# 介绍

AngularJS是一款由Google公司开发维护的前端MVC框架，其克服了[HTML](http://baike.baidu.com/view/692.htm)在构建应用上的诸多不足，从而降低了开发成本提升了开发效率。

## 特点

AngularJS与我们之前学习的jQuery是有一定的区别的，jQuery更准确来说只是一个类库（类库指的是一系列函数的集合）以DOM做为驱动（核心），而AngularJS则一个框架（诸多类库的集合）以数据和逻辑做为驱动（核心）。

框架对开发的流程和模式做了约束，开发者遵照约束进行开发，更注重的实际的业务逻辑。

AngularJS有着诸多特性，最为核心的是：模块化、双向数据绑定、语义化标签、依赖注入等。

与之类似的框架还有BackBone、KnockoutJS、Vue、React等。

## 下载

1、通过[AngularJS](https://angularjs.org/)官网下载，不过由于国内特殊的国情，需要翻墙才能访问。

2、通过npm下载，npm install angular

3、通过bower下载，bower install angular

bower是什么？

## 体验AngularJS

<div class="container" ng-app>

<input type="text" ng-model="xianjs">

<h1>{{xianjs}}</h1>

</div>

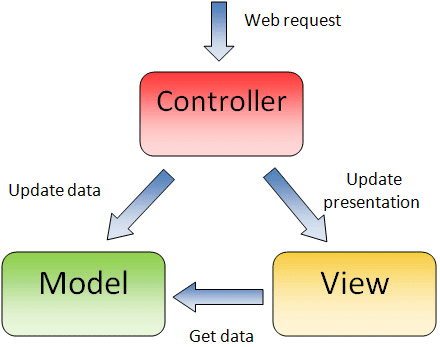
## MVC

MVC是一种开发模式，由模型（Model）、视图（View）、控制器（Controller）3部分构成，采用这种开发模式为合理组织代码提供了方便、降低了代码间的耦合度、功能结构清晰可见。

模型（Model）一般用来处理数据（读取/设置），一般指操作数据库。

视图（View）一般用来展示数据，比如通过HTML展示。

控制器（Controller）一般用做连接模型和视图的桥梁。



MVC更多应用在后端开发程序里，后被引入到前端开发中，由于受到前端技术的限制便有了一些细节的调整，进而出现了很多MVC的衍生版（子集）如MVVM、MVW、MVP、MV\*等。

注：做为初学可以不必过于在意这些概念。

## $scope

View(视图), 即 HTML。

Model(模型), 当前视图中可用的数据。

Controller(控制器), 即 JavaScript 函数，可以添加或修改属性。

scope 是模型。

scope 是一个 JavaScript 对象，带有属性和方法，这些属性和方法可以在视图和控制器中使用

当你在 AngularJS 创建控制器时，你可以将 $scope 对象当作一个参数传递：

## 根作用域

所有的应用都有一个 $rootScope，它可以作用在 ng-app 指令包含的所有 HTML 元素中。

$rootScope 可作用于整个应用中。是各个 controller 中 scope 的桥梁。用 rootscope 定义的值，可以在各个 controller 中使用。



# 模块化

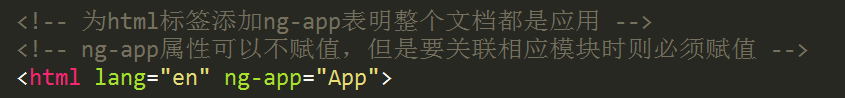
使用AngularJS构建应用（App）时是以模块化（Module）的方式组织的，即将整个应用划分成若干模块，每个模块都有各自的职责，最终组合成一个整体。

采用模块化的组织方式，可以最大程度的实现代码的复用，可以像搭积木一样进行开发。



## 定义应用

通过为任一HTML标签添加ng-app属性，可以指定一个应用，表示此标签所包裹的内容都属于应用（App）的一部分。



## 定义模块

AngularJS提供了一个全局对象angular，在此全局对象下存在若干的方法，其中angular.module()方法用来定义一个模块。



注：应用（App）其本质也是一个模块（一个比较大的模块）。

## 定义控制器

控制器（Controller）作为连接模型（Model）和视图（View）的桥梁存在，所以当我们定义好了控制器以后也就定义好了模型和视图。

<!--控制器的调用-->

<div class="container" ng-controller="firstController">

<input type="text" ng-model="xianjs">

<!--{{}} xianjs视图调用数据进行展示-->

<h1>{{xianjs}}</h1>

</div>

<script type="text/javascript">

/\*定义模块

\*

\* 参数1：应用的名称

\* 参数2：模块的依赖

\* \*/

let app = angular.module("myApp", []);

/\* 定义控制器

\* 参数1：控制器的名称

\* 参数2：依赖参数，是一个数组

\* \*/

app.controller('firstController', ['$scope', function ($scope) {

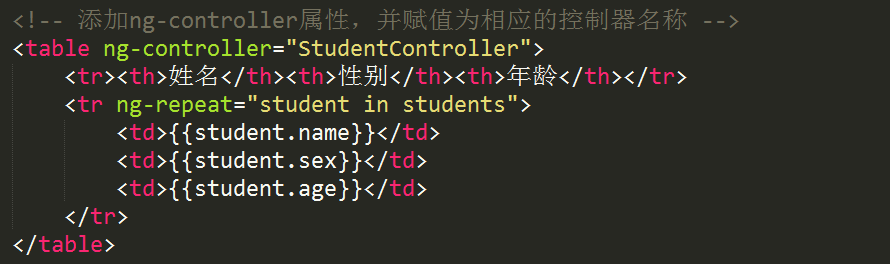
$scope.xianjs = "我是xianjs";

}]);

</script>



模型（Model）数据是要展示到视图（View）上的，所以需要将控制器（Controller）关联到视图（View）上，通过为HTML标签添加ng-controller属性并赋值相应的控制器（Controller）的名称，就确立了关联关系。



<div class="container" ng-controller="firstController">

<table class="table table-bordered">

<tr>

<td>序号</td>

<td>ID</td>

<td>Name</td>

<td>Age</td>

</tr>

<!--index为索引值，item为对象的key-->

<tr ng-repeat="(index,item) in users">

<td>{{index}}</td>

<td>{{item.id}}</td>

<td>{{item.name}}</td>

<td>{{item.age}}</td>

</tr>

</table>

</div>

<script type="text/javascript">

let app = angular.module("myApp", []);//模块

app.controller('firstController', ['$scope', function ($scope) {//控制器

$scope.users = [

{id: 'A01', name: "Tom", age: 25},

{id: 'A02', name: "jack", age: 16},

{id: 'A03', name: "pis", age: 22},

{id: 'A04', name: "kobe", age: 39}

];

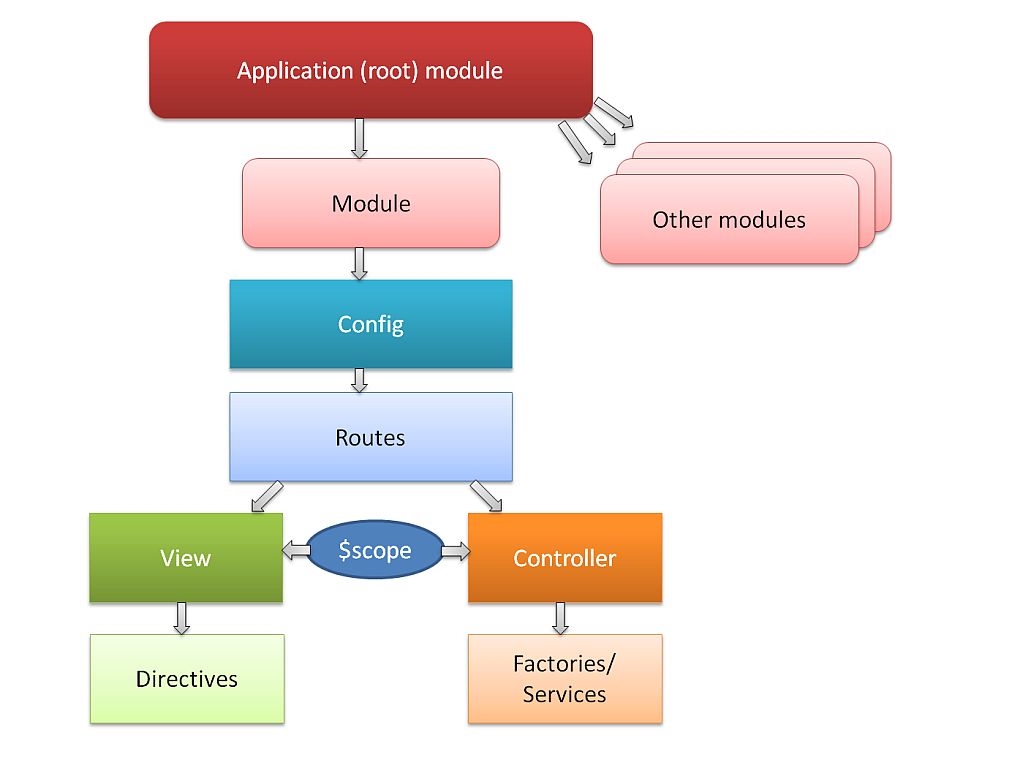
}]);

</script>



以上步骤就是AngularJS最基本的MVC工作模式。

下图是AngularJS的结构，学习AngularJS会围绕下图的结构展开。



## 变量的输出

<**div ng-app="app"**>

*<!--ng-app:表示angular的可以控制的范围-->*

<**div ng-controller="firstController"**>

<**p**><**input type="text" ng-model="*xianjs*"**></**p**>

<**p**>xianjs = {{ ***xianjs*** }}</**p**>

<**p**>1+2 {{ 1+2 }} ,三元表达式: {{ 1 > 2 ? "yes" : "NO" }}</**p**>

<**p ng-bind="*xianjs*"**></**p**>

</**div**>

</**div**>

<**script type="text/javascript"**>

**let *app*** = angular.*module*(**'app'**, []);

***app***.**controller**(**'firstController'**, [**'$scope'**, **function** ($scope) {

$scope.**xianjs** = **'我是xianjs'**;

}]);

</**script**>

# 指令

<http://www.angularjs.net.cn/api/ng/directive>

HTML在构建应用（App）时存在诸多不足之处，AngularJS通过扩展一系列的HTML属性或标签来弥补这些缺陷，所谓指令就是AngularJS自定义的HTML属性或标签，这些指令都是以ng-做为前缀的，例如ng-app、ng-controller、ng-repeat等。

## 内置指令

ng-app 指定应用根元素，至少有一个元素指定了此属性。

ng-controller 指定控制器

ng-show控制元素是否显示，true显示、false不显示

ng-hide控制元素是否隐藏，true隐藏、false不隐藏

ng-if控制元素是否“存在”，true存在、false不存在

ng-src增强图片路径

ng-href增强地址

ng-class控制类名

ng-include引入模板

ng-disabled表单禁用

ng-readonly表单只读

ng-checked单/复选框表单选中

ng-selected下拉框表单选中

见代码示例03Angular内置指令.html

注：后续学习过程中还会介绍其它指令。

<style>

.red {

color: red;

}

</style>

<!--控制器的调用-->

<div class="container" ng-controller="firstController">

<li ng-if="a">ng-if控制元素是否“存在”，true存在、false不存在</li>

<li ng-hide="b">ng-hide控制元素是否隐藏，true隐藏、false不隐藏</li>

<li ng-show="b">ng-show控制元素是否显示，true显示、false不显示</li>

<!--使用src会报错-->

<li><a href="#"><img ng-src="{{src}}" alt=""></a></li>

<li><a ng-href="{{baidu}}">百度</a></li>

<!--ng-class的值为对象，对象的属性为存在的类名，值为布尔值，true添加类名，false不会添加-->

<li ng-class="{red:true}">red</li>

<!--表单禁用-->

<p><input type="text" ng-disabled="true" value="禁止使用"></p>

<!--只读属性-->

<p>money: <input type="text" ng-readonly="true" ng-value="money"></p>

<p><input type="checkbox" ng-checked="true">男</p>

<p><input type="checkbox" ng-checked="false">女</p>

</div>

<script type="text/javascript">

let app = angular.module("myApp", []);//模块

app.controller('firstController', ['$scope', function ($scope) {//控制器

$scope.a = "1";

// $scope.b = false;//显示

$scope.b = true;//隐藏

$scope.src = "./img/1.gif";

$scope.baidu = 'http://www.baidu.com';

$scope.money = "99994.12元";

}]);

</script>

当网页加载完毕，AngularJS 自动开启。

ng-app 指令告诉 AngularJS，<div> 元素是 AngularJS 应用程序 的"所有者"。

ng-model 指令把输入域的值绑定到应用程序变量 name。

ng-bind 指令把应用程序变量 name 绑定到某个段落的 innerHTML。

AngularJS 通过 ng-directives 扩展了 HTML。

ng-app 指令定义一个 AngularJS 应用程序。

ng-init 指令初始化 AngularJS 应用程序变量。

## 自定义指令

AngularJS允许根据实际业务需要自定义指令，通过angular全局对象下的directive方法实现。



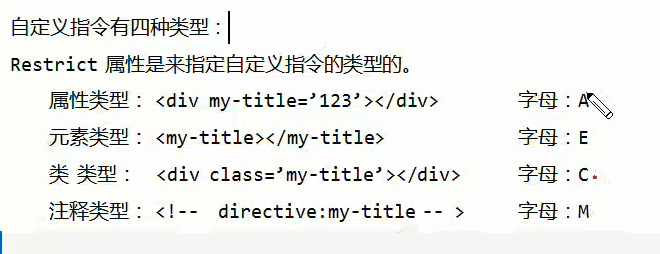
E 代表元素指令 Element

A 代表属性执行 Attribute

M 代表注释指令 Mark

C 代表class执行 Class

<!--控制器的调用-->



<div class="container" ng-controller="firstController">

<div tag></div>

<tag></tag>

<p class="tag"></p>

<!-- directive:tag -->

</div>

<script type="text/javascript">

let app = angular.module("myApp", []);//模块

app.controller('firstController', ['$scope', function ($scope) {//控制器

}]);

/\*

\* 自定义指令

\* 参数1：指令的名称

\* 参数2：回调函数

\* \*/

app.directive("tag", function () {

return {

// 指令类型 E：element

// A：attribute

// M：mark

// C: class

restrict: 'ECMA',

replace: true,//替换

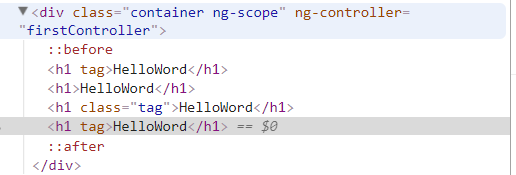
// template: "<h1>HelloWord</h1>",

templateUrl: "./head.html",//加载外部文件

};

});

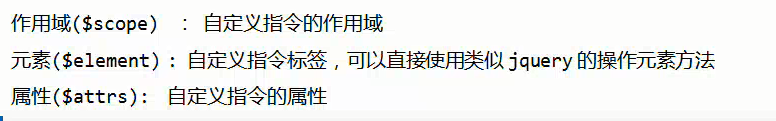
</script>

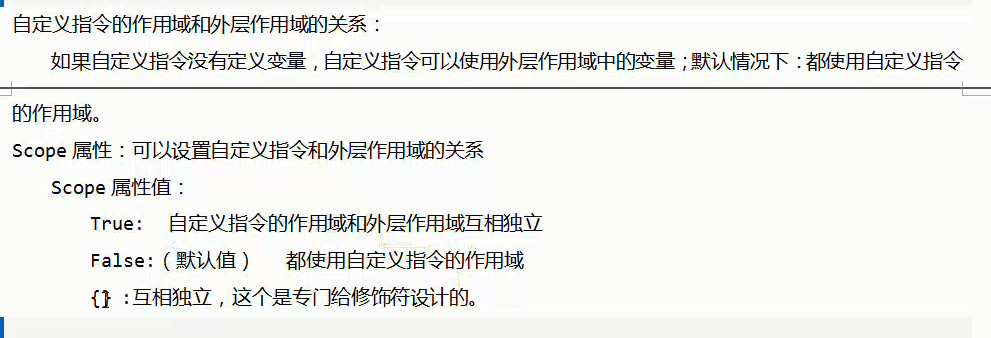




## 指定自定义指令的作用域







**let *app*** = angular.*module*(**'app'**, []);

***app***.**controller**(**'myCtr'**, [**'$scope'**, **function** ($scope) {

}]);

***app***.**directive**(**'myTitle'**, **function** () {

**return** {

*/\* 指令类型 E：element A：attribute M：mark C: class\*/*

**restrict**: **'ECMA'**,*//以怎样的形式进行调用*

**replace**: **true**,*//替换*

**template**: **'<h1>HelloWord {{ xianjs }}</h1>'**,

*/\*\**

*\**

*\** ***@param*** *$scope 作用域,自定义指令的作用域*

*\** ***@param*** *$element 标签,当前自定义指令的标签,可以直接使用类似jq的方法*

*\** ***@param*** *$attrs 属性,自定义指令标签的属性*

*\*/*

controller: **function** ($scope, $element, $attrs) {

$scope.**xianjs** = **'局部指令的注册'**;

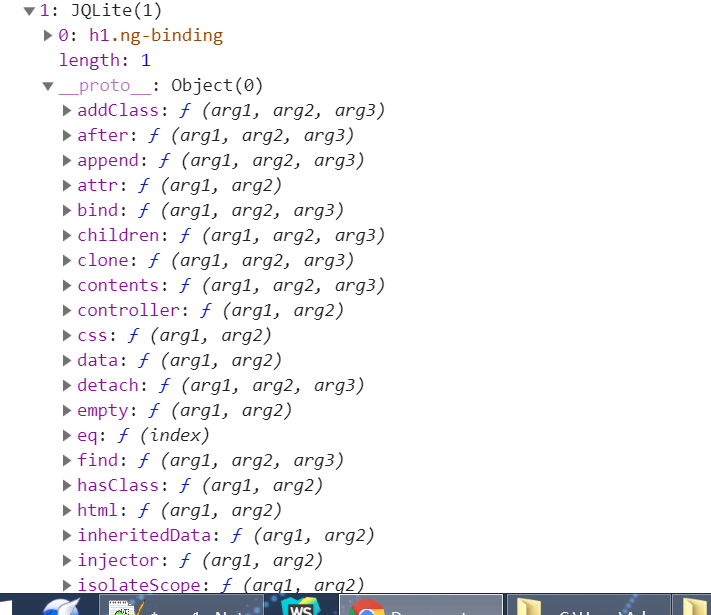
$element.css(**'background'**, **'red'**);

***console***.log(***arguments***);

}

};

});



# 数据绑定

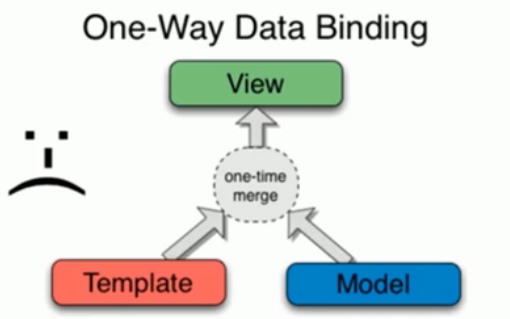
AngularJS是以数据做为驱动的MVC框架，所有模型（Model）里的数据经由控制器（Controller）展示到视图（View）中。

所谓数据绑定指的就是将模型（Model）中的数据与相应的视图（View）进行关联，分为单向绑定和双向绑定两种方式。

## 单向绑定

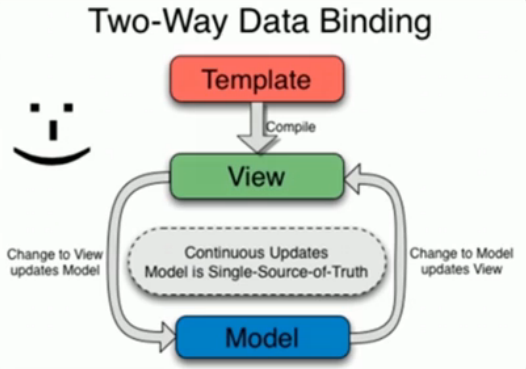
单向数据绑定是指将模型（Model）数据，按着写好的视图（View）模板生成HTML标签，然后追加到DOM中显示，如之前所学的artTemplate 模板引擎的工作方式。

如下图所示，只能模型（Model）数据向视图（View）传递。



## 双向绑定

双向绑定则可以实现模型（Model）数据和视图（View）模板的双向传递，如下图所示。



## 相关指令

在AngularJS中通过“{{}}”和ng-bind指令来实现模型（Model）数据向视图模板（View）的绑定，模型数据通过一个内置服务$scope来提供，这个$scope是一个空对象，通过为这个对象添加属性或者方法便可以在相应的视图（View）模板里被访问。

注：“{{}}”是ng-bind的简写形式，其区别在于通过“{{}}”绑定数据时会有“闪烁”现象，添加ng-cloak也可以解决“闪烁”现象。

<style type="text/css">[ng\:cloak],[ng-cloak],[data-ng-cloak],[x-ng-cloak],.ng-cloak,.x-ng-cloak,.ng-hide:not(.ng-hide-animate){display:none !important;}ng\:form{display:block;}.ng-animate-shim{visibility:hidden;}.ng-anchor{position:absolute;}</style>

<!--控制器的调用-->

<div class="container" ng-controller="firstController">

<h1 ng-cloak>{{data.name}}</h1>

<ul>

<!--<li ng-repeat="item in data.courcess">{{item}}</li>-->

<li ng-repeat="item in data.courcess" ng-bind="item"></li>

</ul>

</div>

<script type="text/javascript">

let app = angular.module("myApp", []);//模块

/\*

\* 单向数据绑定：数据=>视图，可以使用ng-bind

\* {{}}是ng-bind的简写,在网速不好时会出现闪烁现象，但是ng-bind不会出现闪烁现象

\* 可以使用ng-clock修复{{}}出现的闪烁现象

\* \*/

app.controller('firstController', ['$scope', function ($scope) {//控制器

$scope.data = {

name: "Tom",

courcess: ['java', 'css', 'php', 'go']

}

}]);

</script>

通过为表单元素添加ng-model指令实现视图（View）模板向模型（Model）数据的绑定。

<div class="container" ng-controller="firstController">

<input type="text" ng-model="msg">

<span>msg = {{msg}}</span>

<!--

视图=>模型

模型=>视图

-->

<button ng-click="show()">测试</button>

</div>

<script type="text/javascript">

let app = angular.module("myApp", []);//模块

/\*

\* 单向数据绑定：数据=>视图，可以使用ng-bind

\* {{}}是ng-bind的简写,在网速不好时会出现闪烁现象，但是ng-bind不会出现闪烁现象

\* 可以使用ng-clock修复{{}}出现的闪烁现象

\* \*/

app.controller('firstController', ['$scope', function ($scope) {//控制器

$scope.msg = "";

$scope.show = function () {

alert($scope.msg);

}

}]);

</script>

通过ng-init可以初始化模型（Model）也就是$scope。

<**div** ng-init=**"name='lisi';age='26'"**>

<**p**>**name = {{name}}**</**p**>

<**p**>**age={{age}}**</**p**>

</**div**>

AngularJS对事件也进行了扩展，无需显式的获取DOM元素便可以添加事件，易用性变的更强。通过在原有事件名称基础上添加ng-做为前缀，然后以属性的形式添加到相应的HTML标签上即可。如ng-click、ng-dblclick、ng-blur等。

<**div** class=**"container"** ng-controller=**"firstController"**>

<**input** type=**"button"** value=**"增加"** ng-click=**" count = count + 1"**>

<**p**>**count = {{***count***}}**</**p**>

<**hr**>

<**button** ng-click=**"toggle()"**>**隐藏/显示**</**button**>

<**p** ng-hide=**"myVar"**>

**名:** <**input** type=**"text"** ng-model=**"firstName"**><**br**>

**姓名:** <**input** type=**"text"** ng-model=**"lastName"**><**br**>

<**br**>

**Full Name: {{firstName + " " + lastName}}**

</**p**>

</**div**>

<**script** type=**"text/javascript"**>

let *app* = *angular*.module("myApp", []);//模块

*app*.controller('firstController', ['$scope', function (*$scope*) {//控制器

*$scope*.firstName = "John";

*$scope*.lastName = "Doe";

*$scope*.myVar = false;

*$scope*.toggle = function () {

//实现显示和隐藏的切换

*$scope*.myVar = !*$scope*.myVar;

};

}]);

</**script**>

通过ng-repeat可以将数组或对象数据迭代到视图模板中，ng-switch、on、ng-switch-when可以对数据进行筛选。

<**div** class=**"container"** ng-controller=**"firstController"**>

<**ul**>

<!--(key,value)输出对象的key和value-->

<**li** ng-repeat=**"(***k***,***v***) in** *info***"**>**{{***k***}} -- {{***v***}}**</**li**>

</**ul**>

<**ul**>

<**li** ng-repeat=**"(***index***,***item***) in course"** ng-switch=**"item"**>

<**span** ng-switch-when=**"html"**>**{{***item***}}**</**span**>

<**span** ng-switch-when=**"css"**>**{{***item***}}**</**span**>

<**span** ng-switch-when=**"php"**>**{{***item***}}**</**span**>

</**li**>

</**ul**>

</**div**>

<**script** type=**"text/javascript"**>

let *app* = *angular*.module("myApp", []);//模块

*app*.controller('firstController', ['$scope', function (*$scope*) {//控制器

*$scope*.course = ['html', 'css', 'php'];

*$scope*.info = {

name: "Tom",

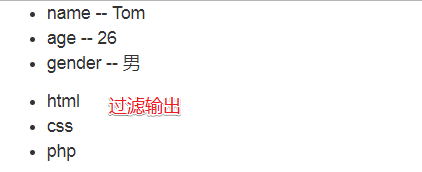
age: 26,

gender: '男'

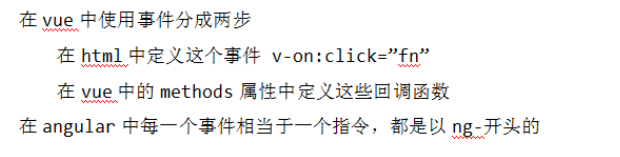
}

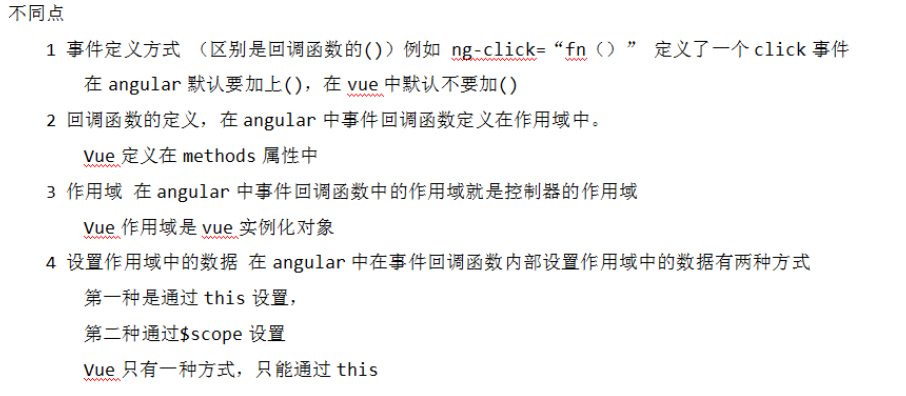
}]);

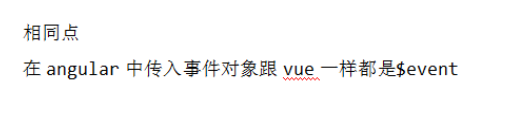
</**script**>



## 事件







<**div ng-app="app"**>

*<!--ng-app:表示angular的可以控制的范围-->*

<**div ng-controller="firstController"**>

*<!--ng-click:绑定事件,定义事件:$scope.show=function(){}-->*

<**button ng-click="***show***(1,2,3)"**>事件的测试</**button**>

*<!--ng-model:双向数据绑定-->*

<**input type="text" ng-model="*xianjs*"**>

*<!--可以进行运算-->*

<**p**>a+b = {{ ***a***+***b*** }}</**p**>

</**div**>

</**div**>

<**script type="text/javascript"**>

**let *app*** = angular.*module*(**'app'**, []);

***app***.**controller**(**'firstController'**, [**'$scope'**, **function** ($scope) {

$scope.**xianjs** = **'我是xianjs'**;

$scope.**a** = 100;

$scope.**b** = 200;

$scope.show = **function** () {

***console***.log(***arguments***);

*alert*($scope.**xianjs**);

};

}]);

</**script**>

<div ng-controller="firstController">

*<!--*

*事件调用可以传递事件对象,字符串,变量:数组都是可以的*

*-->*

<button ng-click="*show*($event,1,2,3,*arr*)">测试按钮事件</button>

</div>

## ng-if和ng-show的比较

<**body ng-app="app"**>

<**button ng-click="flag=!flag"**>切换吧</**button**>

<!--

ng-if:操作的是DOM的移除,加载的速度比较的快的

ng-show:操作的是样式,开始的时候就会加载进来的,切换的代价比较的地

-->

<**p ng-if="flag"**>显示</**p**>

<**p ng-show="flag"**>show显示的形式</**p**>

<**script**>

//1.指定app控制的范围

//2.创建module

**let app** = angular.module(**'app'**, []);

//3.创建控制器

**app**.**controller**(**'firstController'**, [**'$scope'**, **function** ($scope) {

$scope.**flag** = **true**;

}]);

</**script**>

## ng-repeat

<**ul**>

<!--index:从0开始的索引,对象的话为对象的属性

row:数组的值,对象的话为对象的属性值

$index:索引

$first:第一个

$last:最后一个

$Middle:中间的

$odd:奇数

$even:偶数

-->

<**li ng-repeat="item in list"**>{{ **item** }} -

{{ **$index** }} -

{{ **$first** }} -

{{ **$last** }} -

{{ **$middle** }} -

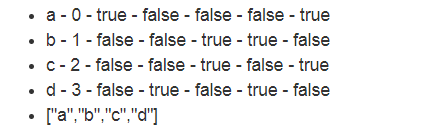
{{ **$odd** }} -

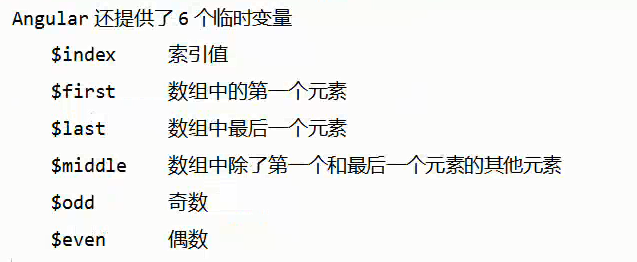
{{ **$even** }}

</**li**>

<**li**>{{ **list** }}</**li**>

</**ul**>





## ng-switch

<div ng-controller="myCtr">

<p ng-if="*flag*">第一个标签</p>

<p ng-if="!*flag*">第二个标签</p>

<ul>

*<!--index:从0开始的索引,对象的话为对象的属性*

*row:数组的值,对象的话为对象的属性值-->*

<li ng-repeat="(*index*,*row*) in *list*">{{ *index* }} - {{ *row* }}</li>

</ul>

<div>

<input type="text" ng-model="*msg*">

*<!--*

*ng-switch :外层的包裹,on指定判断的数据*

*ng-switch-default:默认值*

*ng-switch-when:指定的值*

*-->*

<div ng-switch on="msg">

<div ng-switch-default="">默认的页面展示</div>

<div ng-switch-when="page1">第一个页面</div>

<div ng-switch-when="page2">第二个页面</div>

</div>

</p>

</div>

</div>

<script>

let *app* = angular.*module*('app', []);

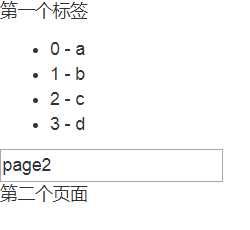
*app*.controller('myCtr', ['$scope', function ($scope) {

$scope.flag = true;

$scope.list = ['a', 'b', 'c', 'd'];

}]);

</script>



# 作用域

通常AngularJS中应用（App）是由若干个视图（View）组合成而成的，而视图（View）又都是HTML元素，并且HTML元素是可以互相嵌套的，另一方面视图都隶属于某个控制器（Controller），进而控制器之间也必然会产生嵌套关系。

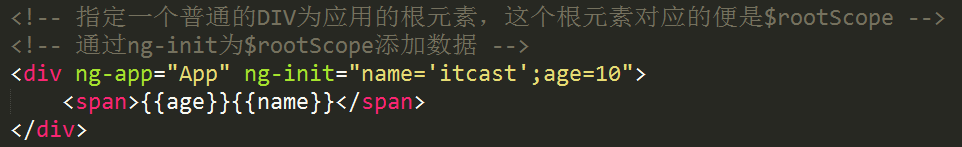
每个控制器（Controller）又都对应一个模型（Model）也就是$scope对象，不同层级控制器（Controller）下的$scope便产生了作用域。

## 根作用域

一个AngularJS的应用（App）在启动时会自动创建一个根作用域$rootScope，这个根作用域在整个应用范围（ng-app所在标签以内）都是可以被访问到的。

所有的应用都有一个 $rootScope，它可以作用在 ng-app 指令包含的所有 HTML 元素中。

$rootScope 可作用于整个应用中。是各个 controller 中 scope 的桥梁。用 rootscope 定义的值，可以在各个 controller 中使用。



## 子作用域

通过ng-controller指令可以创建一个子作用域，新建的作用域可以访问其父作用域的数据。

<**div** class=**"container"** ng-controller=**"firstController"**>

<**p**>**父name = {{***name***}}**</**p**>

<**p**>**全局Money= {{***money***}}**</**p**>

<**div** ng-controller=**"childController"**>

<!--内部作用域没有name的话会向上层查找

本层作用域有的话就会适应本层作用域的name

-->

<**p**> **name = {{***name***}}** </**p**>

</**div**>

</**div**>

<**div**>

**全局的 money = {{***$root***.money}}**

</**div**>

<**script** type=**"text/javascript"**>

let *app* = *angular*.module("myApp", []);//模块

*app*.controller('firstController', ['$scope', function (*$scope*) {//控制器

*$scope*.name = '父亲';

}]);

*app*.controller("childController", ['$scope', '$rootScope', function (*$scope*, *$rootScope*) {

*$scope*.name = "儿子";

*$rootScope*.money = 1000;

}]);

</**script**>

# 过滤器

在AngularJS中使用过滤器格式化展示数据，在“{{}}”中使用“|”来调用过滤器，使用“:”传递参数。

## 内置过滤器

|  |  |
| --- | --- |
| **过滤器** | **描述** |
| currency | 格式化数字为货币格式。 |
| filter | 从数组项中选择一个子集。 |
| lowercase | 格式化字符串为小写。 |
| orderBy | 根据某个表达式排列数组。 |
| uppercase | 格式化字符串为大写。 |

1、currency[货币] 将数值格式化为货币格式

2、date日期格式化，年（y）、月（M）、日（d）、星期（EEEE/EEE）、时（H/h）、分（m）、秒（s）、毫秒（.sss），也可以组合到一起使用。

3、filter在给定数组中选择满足条件的一个子集，并返回一个新数组，其条件可以是一个字符串、对象、函数

4、json将Javascrip对象转成JSON字符串。

5、limitTo取出字符串或数组的前（正数）几位或后（负数）几位

6、lowercase将文本转换成小写格式

7、uppercase将文本转换成大写格式

8、number数字格式化，可控制小位位数

9、orderBy对数组进行排序，第2个参数是布尔值可控制方向(正序或倒序)

<**ul**>

<**li**>**小数点3位：{{***money***|number:3}}**</**li**>

<**li**>**货币：{{***money***|currency:'￥'}}**</**li**>

<**li**>**小写：{{***str***|lowercase}}**</**li**>

<**li**>**大写：{{***str***|uppercase}}**</**li**>

<**li**>**日期：{{currentDate | date:'yyyy-MM-dd HH:mm:ss Z'}}**</**li**>

<**li**>**json格式数据：{{person|json}}**</**li**>

<**li**>**字符串前三个字符：{{***str***|limitTo:3}}**</**li**>

<**li**>**数组前三项：{{***arr***|limitTo:3}}**</**li**>

</**ul**>

<!--小数点3位：95,984.260-->

<!--货币：￥95,984.26-->

<!--小写：abcdefxianjs-->

<!--大写：ABCDEFXIANJS-->

<!--日期：2018-03-15 18:24:01 +0800-->

<!--json格式数据：{ "name": "Tom", "age": "26岁", "gender": "男" }-->

<!--字符串前三个字符：Abc-->

<!--数组前三项：[1,2,3]-->

<**li**>**{{***color***|filter:'e'|filter:'o'}}**</**li**>

<!--

["red","green","yello","oragle"]

["yello","oragle"]

-->

<**li**>**{{***users***|orderBy:'age':true}}**</**li**>

<!--

true:倒叙

false:正序（默认）

-->

<**li**>**{{***users***|orderBy:'age':false}}**</**li**>

## 自定义过滤器

除了使用AngularJS内建过滤器外，还可以根业务需要自定义过滤器，通过模块对象实例提供的filter方法自定义过滤器。



<**div** class=**"container"** ng-controller=**"firstController"**>

<**p**>**xianjs反转 = {{***str***|revers}}**</**p**>

</**div**>

<**script** type=**"text/javascript"**>

let *app* = *angular*.module("myApp", []);//模块

*app*.controller('firstController', ['$scope', function (*$scope*) {//控制器

*$scope*.str = "Xianjs";

}]);

*app*.filter('revers', function () {//依赖注入

return function (*text*) {

return *text*.split("").reverse().join("");

};

});

*console*.log(*app*);

</**script**>

## 日期过滤器的参数说明

<**div ng-controller="firstController"**>

<**p**> datatime = {{ **datatime**|formatDate }}</**p**>

*<!--内置的日期过滤器-->*

<**p**> datatime = {{ **datatime**|**date**:"yyyy-MM-dd HH:mm:ss" }}</**p**>

</**div**>

<**script**>

**let *app*** = angular.*module*(**'app'**, []);

***app***.**controller**(**'firstController'**, [**'$scope'**, **function** ($scope) {

$scope.**datatime** = ***Date***.now();

}]);

*//自定义事件格式化的过滤器*

***app***.**filter**(**'formatDate'**, **function** () {

**return function** (input) {

**let** t = **new *Date***(input);

**let** Y = t.getFullYear();

**let** M = t.getMonth() + 1;

**let** D = t.getDate();

**let** h = t.getHours();

**let** m = t.getMinutes();

**let** s = t.getSeconds();

***console***.log(h, m, s);

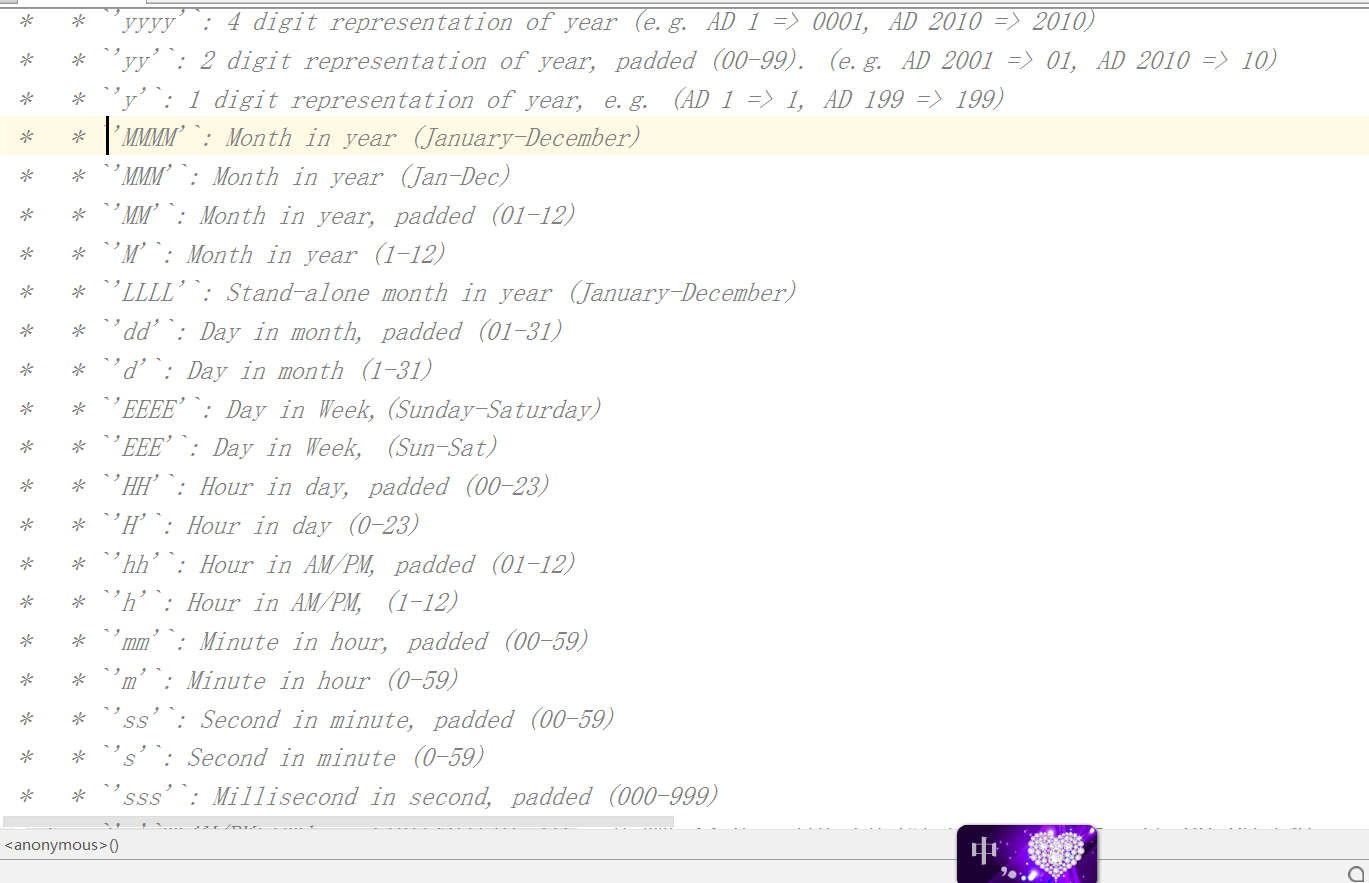
**return `**${ Y }**-**${ M }**-**${ D } ${ h }**-**${ m }**-**${ s }**`**;

};

});

</**script**>





# 表单

## 表单基础

<**form** action=**""**>

<**div** class=**"form-group"**>

<**label** for=**"username"**>**user**</**label**>

<**input** type=**"text"** ng-model=**"user"** class=**"form-control"** id=**"username"**>

</**div**>

<**div** class=**"checkbox"**>

<**input** type=**"checkbox"** ng-model=**"myVal1"**>

</**div**>

<**div** class=**"form-group"**>

<**input** type=**"radio"** ng-model=**"myVar"** value=**"dogs"**>**Dogs**

<**input** type=**"radio"** ng-model=**"myVar"** value=**"tuts"**>**Tutorials**

<**input** type=**"radio"** ng-model=**"myVar"** value=**"cars"**>**Cars**

</**div**>

<**div** class=**"form-group"**>

<**select** ng-model=**"myVars"**>

<**option** value=**""**>

<**option** value=**"dogs"**>**Dogs**

<**option** value=**"tuts"**>**Tutorials**

<**option** value=**"cars"**>**Cars**

</**select**>

</**div**>

<**h3** class=**"page-header"**>**user = {{***user***}}**</**h3**>

<**h3** class=**"page-header"** ng-show=**"myVal1"**>**myVal = {{myVal1}}**</**h3**>

<**h3** class=**"page-header"** ng-show=**"myVar"**>**myVar = {{myVar}}**</**h3**>

<**h3** class=**"page-header"** ng-show=**"myVars"**>**myVar = {{myVars}}**</**h3**>

</**form**>

<**script**>

**let *app*** = angular.*module*(**'app'**, []);

***app***.**controller**(**'myCtr'**, [**'$scope'**, **function** ($scope) {

$scope.**user** = **''**;

$scope.**myVal1** = [];

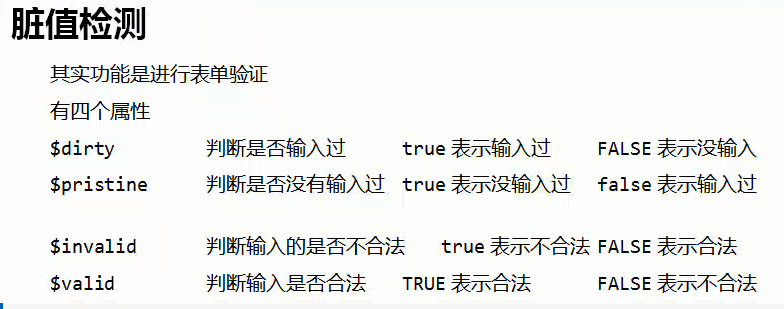
$scope.**myVars** = [];

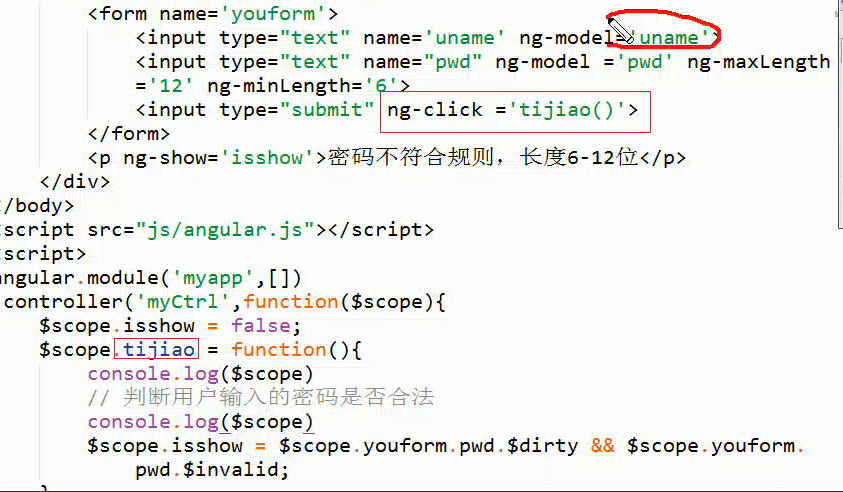
}]);

</**script**>



## 脏值检测

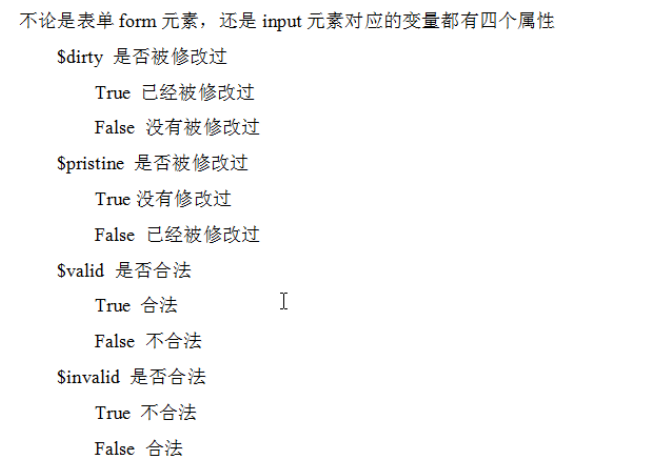




## 输入验证

ng-maxlength="6"：最大长度为6

ng-minlength="4"：最小的长度为4



<**form** action=**""** name=**"regist"** class=**"form-horizontal container-fluid"**>

<**div** class=**"form-group"**>

<**label** for=**"username"** class=**"col-xs-2 control-label"**>**用户名**</**label**>

<**div** class=**"col-xs-8"**>

<**input** type=**"text"** id=**"username"** name=**"username"** class=**"form-control"** ng-model=**"data.username"** ng-maxlength=**"6"**

ng-minlength=**"4"**>

</**div**>

</**div**>

<**div** class=**"form-group"**>

<**label** for=**"tel"** class=**"col-xs-2 control-label"**>**手机号**</**label**>

<**div** class=**"col-xs-8"**><**input** type=**"text"** id=**"tel"** name=**"tel"** ng-model=**"data.tel"** class=**"form-control"**

ng-pattern=**"/^1\d{10}$/"**></**div**>

</**div**>

<**div** class=**"form-group"**>

<**label** for=**"address"** class=**"col-xs-2 control-label"**>**地&emsp;址**</**label**>

<**div** class=**"col-xs-8"**><**input** type=**"text"** id=**"address"** name=**"address"** ng-model=**"data.address"** class=**"form-control"**

ng-required=**"true"**>

</**div**>

</**div**>

<**div** class=**"form-group"**>

<**label** for=**"email"** class=**"col-xs-2 control-label"**>**邮&emsp;箱**</**label**>

<**div** class=**"col-xs-8"**><**input** type=**"text"** id=**"email"** name=**"email"** ng-model=**"data.email"** class=**"form-control"**

ng-pattern=**"/^\w+@[\w\.]/"**></**div**>

</**div**>

<**div** class=**"warning"**>

<**p** class=**"text-danger"** ng-show=**"regist.username.$dirty && regist.username.$invalid"**>**输入正确的用户名称**</**p**>

<**p** class=**"text-danger"** ng-show=**"regist.tel.$dirty && regist.tel.$invalid"**>**输入正确的手机号**</**p**>

<**p** class=**"text-danger"** ng-show=**"regist.address.$dirty && regist.address.$invalid"**>**地址必须填写**</**p**>

<**p** class=**"text-danger"** ng-show=**"regist.email.$dirty && regist.email.$invalid"**>**邮箱格式**</**p**>

<**p** class=**"text-danger"** ng-show=**"showErr"**>**信息不完善**</**p**>

</**div**>

<**div** class=**"form-group col-sm-8 "**>

<**button** class=**"btn btn-danger btn-lg btn-block"** ng-click=**"sub()"** type=**"submit"**>**提交**</**button**>

</**div**>

</**form**>

</**div**>

<**script** type=**"text/javascript"**>

let *app* = *angular*.module('myApp', []);

*app*.controller('firstController', ['$scope', function (*$scope*) {//控制器

*$scope*.data = {};

*$scope*.showErr = false;

//点击提交按钮

*$scope*.sub = function () {

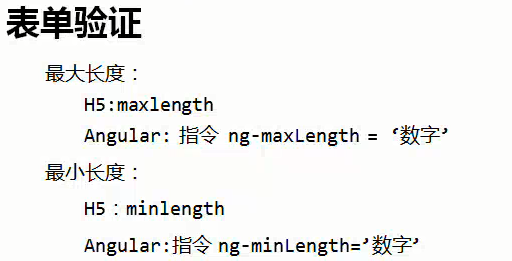
*$scope*.showErr = *$scope*.regist.$invalid;

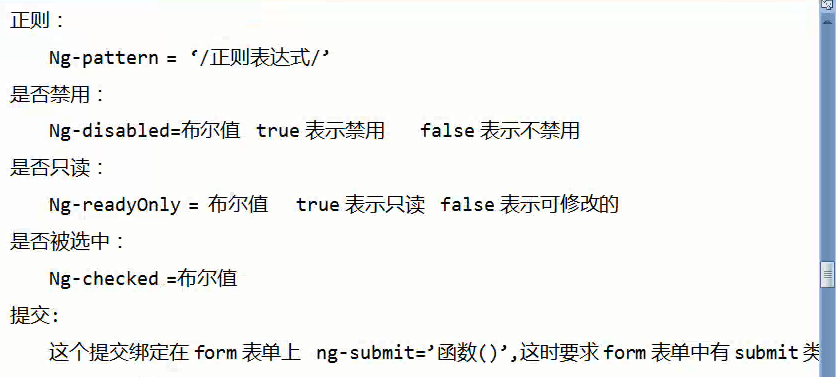
*console*.log(*$scope*.regist.$invalid);

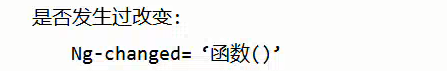
}

}]);

</**script**>







## 特殊属性

### ng-href/ng-src

添加属性和连接使用的

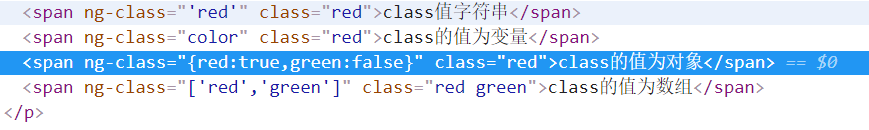
### ng-class

<span ng-class="'red'">class值字符串</span>

<span ng-class="color">class的值为变量</span>

<span ng-class="{red:true,green:false}">class的值为对象</span>

<span ng-class="['red','green']">class的值为数组</span>





### ng-style

**对象形式和变量的形式**

<div ng-controller="myCtr">

<p>

<span ng-style="{fontSize:'60px',color:'red'}">对象的方式设置样式</span><br>

<span ng-style="txtStyle">变量的方式设置样式</span>

</p>

</div>

<script>

let *app* = angular.*module*('app', []);

*app*.controller('myCtr', ['$scope', function ($scope) {

*//变量的方式设置样式*

$scope.txtStyle = { color: 'red', background: '#666' };

}]);

</script>

# 依赖注入

AngularJS采用模块化的方式组织代码，将一些通用逻辑封装成一个对象或函数，实现最大程度的复用，这导致了使用者和被使用者之间存在依赖关系。

所谓依赖注入是指在运行时自动查找依赖关系，然后将查找到依赖传递给使用者的一种机制。

通俗的讲就是通入注入的方式解决依赖关系。

常见的AngularJS内置服务有$http、$location、$timeout、$rootScope等

依赖注入（Dependency Injection，简称DI）是一种软件设计模式，在这种模式下，一个或更多的依赖（或服务）被注入（或者通过引用传递）到一个独立的对象（或客户端）中，然后成为了该客户端状态的一部分；该模式分离了客户端依赖本身行为的创建，这使得程序设计变得松耦合，并遵循了依赖反转和单一职责原则。与服务定位器模式形成直接对比的是，它允许客户端了解客户端如何使用该系统找到依赖；

一句话 --- 没事你不要来找我，有事我会去找你。

<http://www.angularjs.net.cn/api/105.html>

$http是Angular的一个核心服务，它有利于浏览器通过XMLHttpRequest 对象或者 JSONP和远程HTTP服务器交互。

$HTTP API 是基于 $q服务暴露的deferred/promise APIs。

快捷使用方式：

$http.get

$http.head

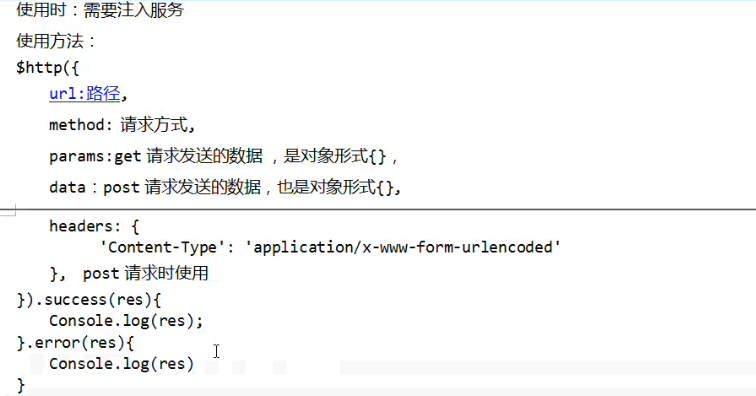
$http.post

$http.put

$http.delete

$http.jsonp

$http.patch



## get请求

<**script**>

**let *app*** = angular.*module*(**'app'**, []);

*//依赖注入使用$http服务*

***app***.**controller**(**'myCtr'**, [**'$scope'**, **'$http'**, **function** ($scope, $http) {

$http({

**url**: **'./carts.json'**,

**method**: **'get'**,

**params**: {*//get请求参数的配置*

**name**: **'lisi'**,

**age**: 26

},

*//支持.then()和success().error()*

}).then(res => {

***console***.log(res);

}, reason => {

***console***.log(reason);

});

}]);

</**script**>

## 行内注入

以数组形式明确声明依赖，数组元素都是包含依赖名称的字符串，数组最后一个元素是依赖注入的目标函数。



推荐使用这种方式声明依赖



## 推断式注入

没有明确声明依赖，AngularJS会将函数参数名称当成是依赖的名称。



这种方式会带来一个问题，当代码经过压缩后函数的参数被压缩，这样便会造成依赖无法找到。

见代码示例14 AngularJS依赖注入(推断式).html

**服务**

服务是一个对象或函数，对外提供特定的功能。

/\*行内式依赖注入\*/

/\*app.controller('firstController', ['$scope', function ($scope) {//控制器

$scope.str = "Xianjs";

}]);\*/

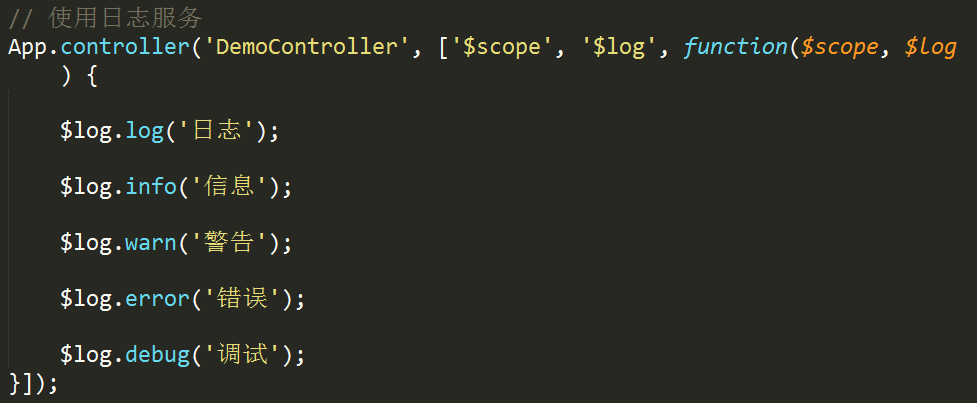
*app*.controller('firstController', '$http', function (*$scope*, *$http*) {

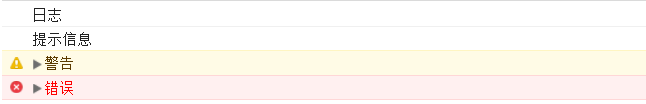
*$scope*.str = "xxxxxx";

});

# 内置服务

## $log打印调试信息





## angular定时器封装

$timeout&$interval对原生Javascript中的setTimeout和setInterval进行了封装。



*app*.controller('firstController', ['$scope', '$timeout', '$interval', function (*$scope*, *$timeout*, *$interval*) {

*$scope*.n = 0;

*$timeout*(function () {

*console*.log("3s后执行了");

}, 2002);

*$interval*(function () {

*console*.log('0.5秒执行一次', *$scope*.n);

*$scope*.n++;

}, 500);

}]);

let timer = *$interval*(function () {

*$scope*.now = new *Date*();

}, 1000);

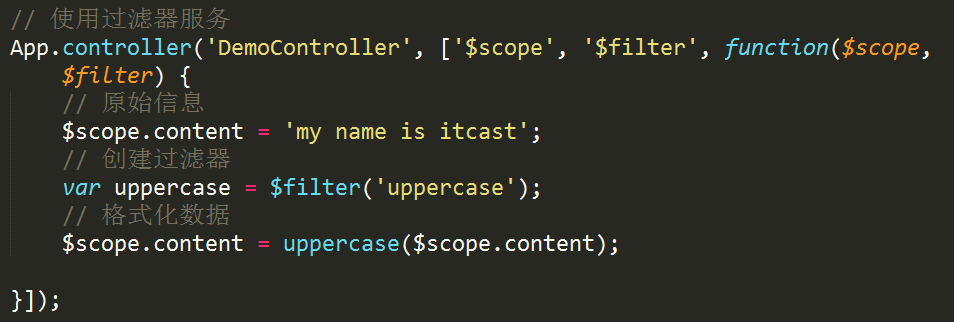
//停止定时器

*$scope*.stop = function () {

*$interval*.cancel(timer);

}

## $filter在控制器中格式化数据。



见代码示例17 AngularJS内置服务filter.html

## $http用于向服务端发起异步请求。



*app*.controller('firstController', ['$scope', '$http', function (*$scope*, *$http*) {

*$http*({

method: "get",//请求方法

url: "19.php",//请求地址

}).then(function success(*res*) {

*console*.log(*res*.data);

*$scope*.data = *res*.data;

}, function error() {

*console*.log("错误");

});

}]);

同时还支持多种快捷方式如$http.get()、$http.post()、$http.jsonp。

注：各参数含义见代码注释。

## get方式



*app*.controller('firstController', ['$scope', '$http', '$log', function (*$scope*, *$http*, *$log*) {

*$http*({

method: "get",//请求方法

url: "19.php",//请求地址

params: {

name: "李四",

age: 26,

gender: '男'

}

}).then(function success(*res*) {

*console*.log(*res*.data);

*$scope*.data = *res*.data;

}, function error() {

*console*.log("错误");

});

}]);

## pot方式

data: "name=李四&age=26&gender=男",

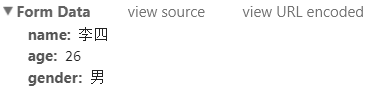
//设置请求头

headers: {

//Content-Type:application/x-www-form-urlencoded

'Content-Type': 'application/x-www-form-urlencoded'

}





## 获取服务端数据

App.controller('DemoCtrl', ['$scope', '$http', function ($scope, $http) {

$scope.get = function () {

$http({

url: './data.json',

method: 'get'

}).then(function (info) {

console.log(info);

$scope.stars = info.data;//返回的数据进行处理

});

}

}]);

## 获取天气预报

Weather.controller('WeatherCtrl', ['$scope', '$http', function ($scope, $http) {

// 使用jsonp

$http({

url: 'http://api.map.baidu.com/telematics/v3/weather',

method: 'jsonp',

params: {

ak: '0A5bc3c4fb543c8f9bc54b77bc155724',

location: '西安市',

output: 'json',

callback: 'JSON\_CALLBACK'

}

}).then(function (info) {

$scope.wether = info.data.results;

console.log(info)

})

}])

## 聊天机器人

<script>

let Chat = angular.module('Chat', []);

Chat.controller('ChatCtrl', ['$scope', '$http', function ($scope, $http) {

$scope.messages = [];

// [

// {text: '我说的话', role: '自已'},

// {text: '对方说的话', role: '对方'}

// ]

$scope.send = function () {

// alert($scope.msg);

$scope.messages.push({

text: $scope.msg,

role: '我说',

cls: 'self'

});

$scope.msg = '';

$http({

url: './chat.php'

}).then(function (info) {

console.log(info.data)

$scope.messages.push({

text: info.data,

role: '对方说',

cls: 'other'

});

})

}

}])

</script>

## 自定义服务

通过上面例子得知，所谓服务是将一些通用性的功能逻辑进行封装方便使用，AngularJS允许将自定义服务。

### factory方法



factory

factory方式创建的服务，作用就是返回一个有属性有方法的对象。相当于：var f = myFactory();

var app = angular.module('myApp', []);<br>

//通过工厂模式创建自定义服务

app.factory('myFactory', function() {

var service = {};//定义一个Object对象'

service.name = "张三";

var age;//定义一个私有化的变量

//对私有属性写getter和setter方法

service.setAge = function(newAge){

age = newAge;

}

service.getAge = function(){

return age;

}

return service;//返回这个Object对象

});

//创建控制器 回调函数把自定义服务return的对象作为参数传入 可以把他的值作为$scope的一个属性 用于view中

app.controller('myCtrl', function($scope, myFactory) {

myFactory.setAge(20);

$scope.r =myFactory.getAge();

alert(myFactory.name);

});

<**script** type=**"text/javascript"**>

let *app* = *angular*.module("myApp", []);//模块

/\*\*

\* 返回值为函数

\*/

*app*.factory('demo1', [function () {

return function () {

*console*.log('我是自定义服务...');

}

}]);

/\*

\* 返回数据可以是任何的类型

\* \*/

*app*.factory('demo', [function () {

// return "Hello";

// return 1;

// return [1, 2, 43, 5];

return {

name: 'Tom',

age: 26

}

}]);

// 定义一个报时服务

*app*.factory('ShowTime', ['$filter', function (*$filter*) {

// 这里依赖了 $filter 这个服务

// 只要有依赖的地方都可以使用“依赖注入”

// 返回了一个对象，此对象下

// 包含了两个方法

return {

// 用来直接显示当前日期

now: function () {

return *$filter*('date')(new *Date*, 'yyyy-MM-dd');

},

// 根据用户参数显示特定格式日期

format: function (*format*) {

return *$filter*('date')(new *Date*, *format*);

}

}

}]);

*app*.controller('firstController', ['$scope', 'demo', 'ShowTime', function (*$scope*, *demo*, *ShowTime*) {

// demo();

// console.log(demo);

// 调用报时服务的 now 方法

*$scope*.now = *ShowTime*.now();

// 调用服时服务的 format 方法

*$scope*.time = *ShowTime*.format('hh:mm:ss');

}]);

</**script**>

### service方法

通过service方式创建自定义服务，相当于new的一个对象：var s = new myService();，只要把属性和方法添加到this上才可以在controller里调用。

<div ng-app="myApp" ng-controller="myCtrl">

<h1>{{r}}</h1>

</div>

<script>

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

app.service('myService', function($http,$q) {

this.name = "service";

this.myFunc = function (x) {

return x.toString(16);//转16进制

}

this.getData = function(){

var d = $q.defer();

$http.get("ursl")//读取数据的函数。

.success(function(response) {

d.resolve(response);

})

.error(function(){

alert(0)

d.reject("error");

});

return d.promise;

}

});

app.controller('myCtrl', function($scope, myService) {

$scope.r = myService.myFunc(255);

myService.getData().then(function(data){

console.log(data);//正确时走这儿

},function(data){

alert(data)//错误时走这儿

});

});

</script>



见代码示例20 AngularJS自定义服务service.html

在介绍服务时曾提到服务本质就是一个对象或函数，所以自定义服务就是要返回一个对象或函数以供使用。

<**script**>

var *App* = *angular*.module('App', []);

// controller 定义控制器

// directive 定义指令

// filter 定义过滤器

// factory 定义服务

// service 也能定义服务

// 需要两个参数

// 第1个参数 服务名称

// 第2个参数 数组（依赖）

*App*.service('sayHi', ['$http', function (*$http*) {

// 在此写当前服务的具体逻辑

// $http({

// url: ''

// })

// 不再使用 return 将服务结果返回了

// 而是使用 this 来返回

this.num = 1;

this.str = 'abc';

this.arr = ['html', 'js', 'css'];

this.fn = function () {

}

}]);

*App*.controller('DemoCtrl', ['$scope', 'sayHi', function (*$scope*, *sayHi*) {

*console*.log(*sayHi*)

}]);

// 报时服务

*App*.service('ShowTime', ['$filter', function (*$filter*) {

// 为服务添加一个方法

this.now = function () {

return *$filter*('date')(new *Date*, 'yyyy-MM-dd');

}

// 为服务添加一个方法

this.format = function (*format*) {

return *$filter*('date')(new *Date*, *format*);

}

}])

// 测试报时服务

*App*.controller('DemoCtrl2', ['$scope', 'ShowTime', function (*$scope*, *ShowTime*) {

// 将服务执行结果添加至模型上

*$scope*.now = *ShowTime*.now();

*$scope*.time = *ShowTime*.format('hh:mm:ss');

}])

</**script**>

### provder

只有provder是能传 .config() 函数的 service。如果想在 service 对象启用之前，先进行模块范围的配置，那就应该选择 provider。需要注意的是：在config函数里注入provider时，名字应该是：providerName+Provider.

使用Provider的优点就是，你可以在Provider对象传递到应用程序的其他部分之前在app.config函数中对其进行修改。

当你使用Provider创建一个service时，唯一的可以在你的控制器中访问的属性和方法是通过$get()函数返回内容

<body>

<div ng-app="myApp" ng-controller="myCtrl">

</div>

<script>

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

//需要注意的是：在注入provider时，名字应该是：providerName+Provider

app.config(function(myProviderProvider){

myProviderProvider.setName("大圣");

});

app.provider('myProvider', function() {

var name="";

var test={"a":1,"b":2};

//注意的是，setter方法必须是(set+变量首字母大写)格式

this.setName = function(newName){

name = newName

}

this.$get =function($http,$q){

return {

getData : function(){

var d = $q.defer();

$http.get("url")//读取数据的函数。

.success(function(response) {

d.resolve(response);

})

.error(function(){

d.reject("error");

});

return d.promise;

},

"lastName":name,

"test":test

}

}

});

app.controller('myCtrl', function($scope,myProvider) {

alert(myProvider.lastName);

alert(myProvider.test.a)

myProvider.getData().then(function(data){

//alert(data)

},function(data){

//alert(data)

});

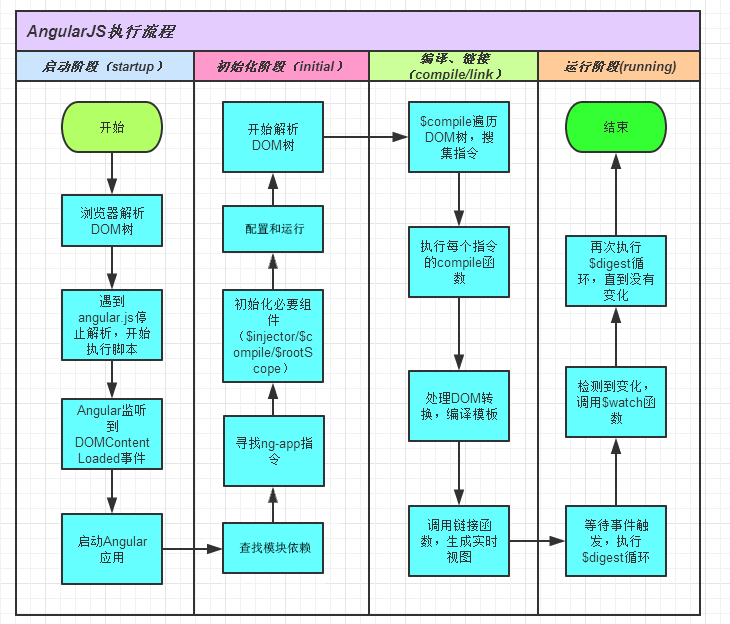
});

</script>

</body>

# 模块加载

AngularJS模块可以在被加载和执行之前对其自身进行配置。我们可以在应用的加载阶段配置不同的逻辑。



## 配置块

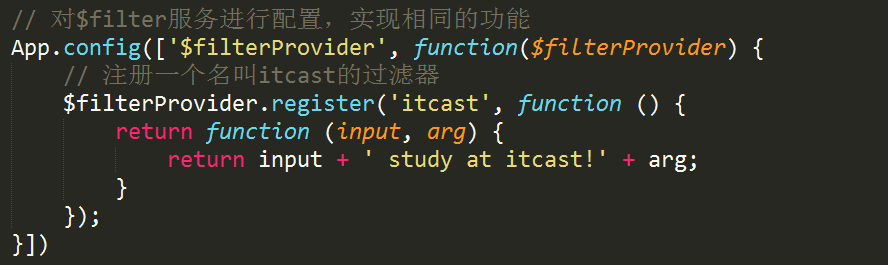
1、通过config方法实现对模块的配置，AngularJS中的服务大部分都对应一个“provider”，用来执行与对应服务相同的功能或对其进行配置。

比如$log、$http、$location都是内置服务，相对应的“provider”分别是$logProvider、$httpProvider、$locationPorvider。

下图以$log为例进行演示，修改了配置



下图以$filter为例进行演示，实现相同功能



见代码示例21 AngularJS配置块.html

## 运行块

服务也是模块形式存在的对且对外提供特定功能，前面学习中都是将服务做为依赖注入进去的，然后再进行调用，除了这种方式外我们也可以直接运行相应的服务模块，AngularJS提供了run方法来实现。

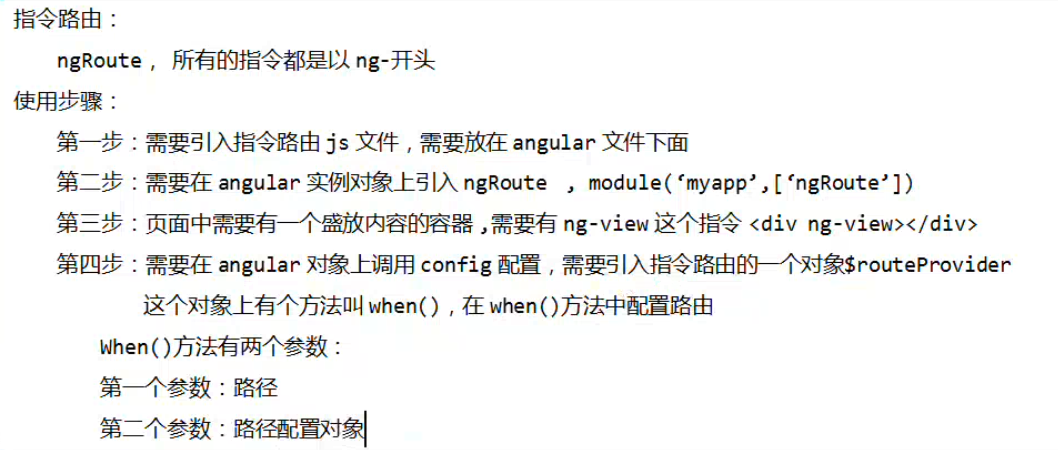


不但如此，run方法还是最先执行的，利用这个特点我们可以将一些需要优先执行的功能通过run方法来运行，比如验证用户是否登录，未登录则不允许进行任何其它操作。

见代码示例22 AngularJS配置块.html

注：此知识点意在了解AngularJS的加载机制。

# 路由



一个应用是由若个视图组合而成的，根据不同的业务逻辑展示给用户不同的视图，路由则是实现这一功能的关键。

## SPA

SPA（Single Page Application）指的是通单一页面展示所有功能，通过Ajax动态获取数据然后进行实时渲染，结合CSS3动画模仿原生App交互，然后再进行打包（使用工具把Web应用包一个壳，这个壳本质上是浏览器）变成一个“原生”应用。

在PC端也有广泛的应用，通常情况下使用Ajax异步请求数据，然后实现内容局部刷新，局部刷新的本质是动态生成DOM，新生成的DOM元素并没有真实存在于文档中，所以当再次刷新页面时新添加的DOM元素会“丢失”，通过单页面应可以很好的解决这个问题。

## 路由

在后端开发中通过URL地址可以实现页面（视图）的切换，但是AngularJS是一个纯前端MVC框架，在开发单页面应用时，所有功能都在同一页面完成，所以无需切换URL地址（即不允许产生跳转），但Web应用中又经常通过链接（a标签）来更新页面（视图），当点击链接时还要阻止其向服务器发起请求，通过锚点（页内跳转）可以实现这一点。

实现单页面应用需要具备：

a、只有一页面

b、链接使用锚点

c、history api

见代码实例23 锚点.html和24 单页面应用原理分析.html

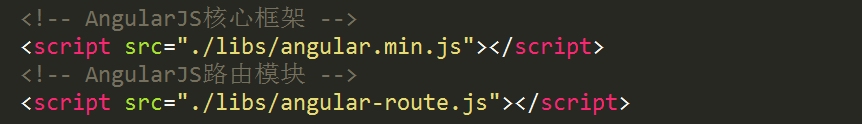
通过上面的例子发现在单一页面中可以能过hashchange事件监听到锚点的变化，进而可以实现为不同的锚点准不同的视图，单页面应用就是基于这一原理实现的。

AngularJS对这一实现原理进行了封装，将锚点的变化封装成路由（Route）,这是与后端路由的根本区别。

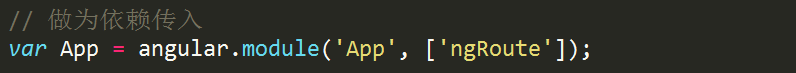
在1.2版前路由功能是包含在AngularJS核心代码当中，之后的版本将路由功能独立成一个模块，[下载angular-route.js](https://code.angularjs.org/)

## 使用

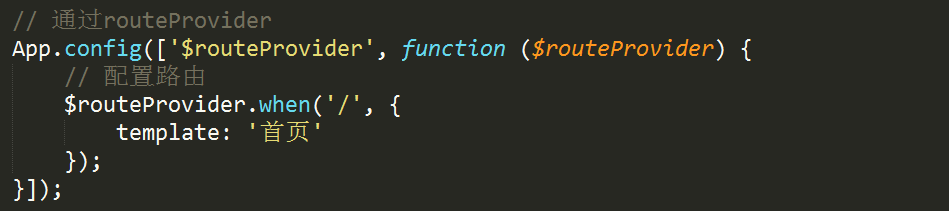
1、引入angular-route.js



2、实例化模块（App）时，当成依赖传进去（模块名称叫ngRoute）。

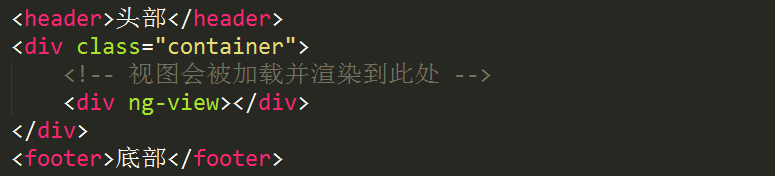


3、配置路由模块



4、布局模板

通过ng-view指令布局模板，路由匹配的视图会被加载渲染到些区域。



见代码实例 25 AngularJS路由.html

## 路由参数

1、提供两个方法匹配路由，分别是when和otherwise，when方法需要两个参数，otherwise方法做为when方法的补充只需要一个参数，其中when方法可以被多次调用。

2、第1个参数是一个字符串，代表当前URL中的hash值。

3、第2个参数是一个对象，配置当前路由的参数，如视图、控制器等。

a、template 字符串形式的视图模板

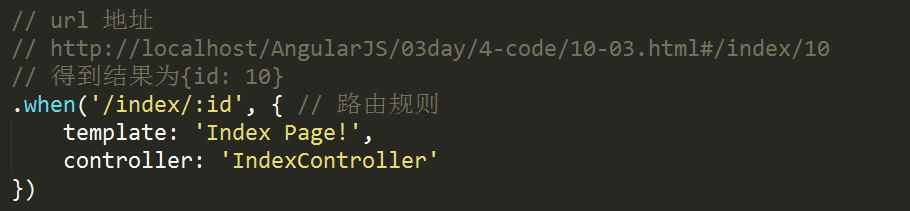
b、templateUrl 引入外部视图模板

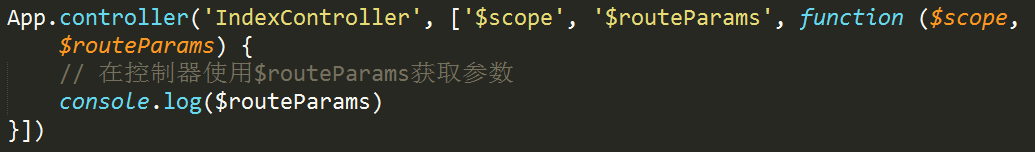
c、controller 视图模板所属的控制器

d、redirectTo跳转到其它路由

见代码实例26 AngularJS路由配置.html

4、获取参数，在控制中注入$routeParams可以获取传递的参数





见代码实例 27 AngularJS路由参数.html

# 其它

## jQuery

在没有引入jQuery的前提下AngularJS实现了简版的jQuery Lite，通过angular.element不能选择元素，但可以将一个DOM元素转成jQuery对象，如果引提前引入了jQuery则angular.element则完全等于jQuery。

见代码示例28AngularJS中使用jQuery.html

## bower

基于NodeJS的一个静态资源管理工具，由twitter公司开发维，解决大型网站中静态资源的依赖问题。

1、依赖NodeJS环境和git工具。

2、npm install -g bower安装bower

3、bower search 查找资源信息

4、bower install 安装（下载）资源，通过#号可以指定版本号

5、bower info 查看资源信息

6、bower uninstall 卸载（删除）资源

# 第二部分angular

<http://www.angularjs.net.cn/api/>

# angular脚手架

<https://www.angular.cn/cli>

## 安装 Angular CLI

Angular CLI 的主版本会跟随它所支持的 Angular 主版本，不过其小版本可能会独立发布。

使用 npm 包管理器来安装 CLI：

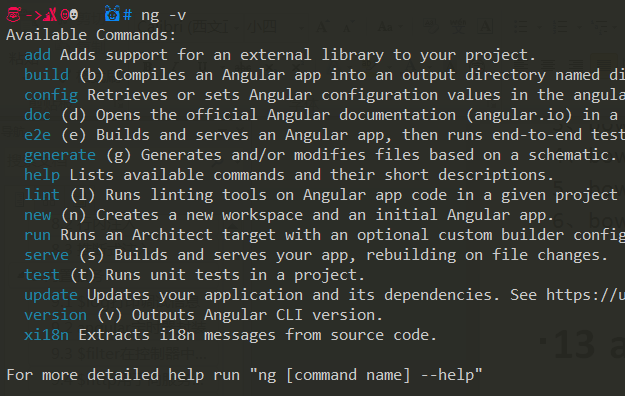
content\_copynpm install -g @angular/cli

有关版本变更的详情，以及如何从以前版本升级的信息，参见 GitHub 上的 Releases 页：<https://github.com/angular/angular-cli/releases>

安装CLI: cnpm install -g @angular/cli

//卸载: npm uninstall -g @angular/cli npm cache clean

检测是否成功: ng –version or ng v



## 新建项目

**新建项目**

ng new my-app

//cd my-app //进入my-app

ng serve //启服务

ng new my-first-project

cd my-first-project

ng serve

## 工作区与项目文件

[ng new](https://www.angular.cn/cli/new) 命令会创建一个 **Angular 工作区**目录，并生成一个新的应用骨架。 每个工作区中可以包含多个应用和库。 由 [ng new](https://www.angular.cn/cli/new) 命令创建的初始应用位于工作区的顶层。 你在工作区中生成的其它应用或库，会放在 projects/ 子目录下。

新生成的应用中包含根模块的源码，还有根组件和模板。 每个应用都有一个 src 目录，其中包含逻辑、数据和静态文件。

你可以直接编辑这些生成的文件，也可以使用 CLI 命令来添加或修改它们。 使用 [ng generate](https://www.angular.cn/cli/generate) 命令也可以添加其它组件和服务，以及管道、指令的源码等。 必须在工作区或项目目录下才能执行 [add](https://www.angular.cn/cli/add) 或 [generate](https://www.angular.cn/cli/generate) 之类的命令，因为这些命令需要在应用或库上进行创建或其它操作。

* 欲知详情，参见[工作区的文件结构](https://www.angular.cn/guide/file-structure)。

工作区的顶级包含一系列工作区级的配置文件。

|  |  |
| --- | --- |
| 工作区配置文件 | 用途 |
| .editorconfig | 代码编辑器配置。参见 [EditorConfig](https://editorconfig.org/) |
| .gitignore | 指定 [Git](https://git-scm.com/) 要忽略的非跟踪的文件。 |
| angular.json | 工作区中所有项目的默认 CLI 配置，包括 CLI 使用的构建选项、运行选项、测试工具选项（比如 [TSLint](https://palantir.github.io/tslint/)、[Karma](https://karma-runner.github.io/)、[Protractor](http://www.protractortest.org/)）等。欲知详情，参见 [Angular 工作区配置](https://www.angular.cn/guide/workspace-config)。 |
| node\_modules | 提供给整个工作区的 [npm 包](https://www.angular.cn/guide/npm-packages)。 |
| package.json | 配置用于工作区中所有项目的包依赖项。有关此文件的特有格式和内容，参见 [npm 文档](https://docs.npmjs.com/files/package.json)。 |
| package-lock.json | 为 npm 客户端安装到 node\_modules 中的所有软件包提供版本信息。详情参见 [npm documentation](https://docs.npmjs.com/files/package-lock.json)。如果你使用 yarn 客户端，此文件会由 [yarn.lock](https://yarnpkg.com/lang/en/docs/yarn-lock/) 代替。 |
| tsconfig.json | 工作区中所有应用的默认 [TypeScript](https://www.typescriptlang.org/) 配置。包括 TypeScript 选项和 Angular 模板编译器选项。参见 [TypeScript 配置](https://www.angular.cn/guide/typescript-configuration)。 |
| tslint.json | 工作区中所有应用的默认 [TSLint](https://palantir.github.io/tslint/) 配置。 |
| README.md | 介绍文档 |

src/ 子目录包含该初始应用的源文件（应用逻辑、数据和资源）以及配置文件。 工作区级的 node\_modules 依赖，对于这个项目也是可见的。

|  |  |
| --- | --- |
| 应用源码与配置文件 | 用途 |
| app/ | 包含组件文件，其中定义了应用逻辑和数据。详情参见稍后的[应用源码目录](https://www.angular.cn/guide/file-structure#app-src)。 |
| assets/ | 包含图像文件和其它文件，当构建应用时会被原样复制到构建目标中。 |
| environments/ | 包含针对特定目标环境的配置选项。默认情况下有一个未命名的标准开发环境和一个名叫 "prod" 的产品环境。你可以定义一些额外的目标环境配置。 |
| browserslist  browserlist | 配置各个目标浏览器和 Node.js 版本之间的市场占有率，供各种前端工具使用。详情参见 [GitHub 上的 Browserlist](https://github.com/browserslist/browserslist)。 |
| favicon.ico | 一个用在书签栏上的应用图标。 |
| index.html | 当有人访问你的应用时给出的主 HTML 文件，你通常不用手动在这里添加任何 <script> 或<link> 标签。 |
| main.ts | 应用的主入口点。使用 [JIT 编译器](https://angular.cn/guide/glossary#jit)编译应用，并引导应用的根模块 AppModule 来运行在浏览器中。你也可以为 CLI 的 build 和 serve 命令添加 --aot 标志，来使用 [AOT 编译器](https://angular.cn/guide/aot-compiler) 而不必修改任何代码。 |
| polyfills.ts | 为浏览器支持提供腻子脚本（polyfill）。 |
| styles.sass | 列出为项目提供样式的 CSS 文件。其扩展名和你为项目配置的样式预处理器保持一致。 |
| test.ts | 单元测试的主入口点，其中带有一些特定于 Angular 的配置。一般来说你不必编辑这个问题。 |
| tsconfig.app.json | 继承自工作区级的 tsconfig.json 文件。 |
| tsconfig.spec.json | 继承自工作区级的 tsconfig.json 文件。 |
| tslint.json | 继承自工作区级的 tsconfig.json 文件。 |

## 应用源码目录

在 src/ 目录下，app/ 目录包含你的应用逻辑和数据。Angular 组件、模板和样式都在这里。 assets/ 子目录包含图片和应用所需的其它文件。src/ 顶层的文件用于支持测试和运行你的应用。



## 脚手架命令

#### 新建组件

ng generate component component-name

也可缩写为：

ng g c component-name

下面是一些命令的缩写：

g —— generate 生成

c —— class 类

d —— directive 指令

m —— module 模块

p —— pipe 管道

s —— service 服务

it —— inline-template 内联模板

is —— inline-style 内联样式

# 指令

## 循环

### ngModel:双向数据绑定

<input type="text" [(ngModel)]="msg">

<p>msg = {{ msg }}</p>

### for...of循环

<ul>

<li *\*ngFor*="let *list* of lists">

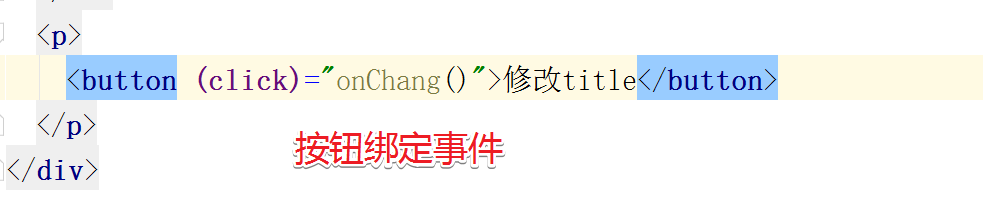
{{ *list*.id }} - {{ *list*.name }}

</li>

</ul>

### (click)事件绑定





### \*ngIf:显示和隐藏

<p *\*ngIf*="msg">显示</p>

<button (click)="msg=!msg">切换</button>

### \*ngSwitch:多分支判断

<div [ngSwitch]="msg">

<div *\*ngSwitchCase*="'xian'">西安</div>

<div *\*ngSwitchCase*="'beijing'">北京</div>

<div *\*ngSwitchDefault*>默认值</div>

</div>

### 动态属性:[]

动态属性: <input type="text" [disabled]="msg" [(ngModel)]="msg">

### 样式的处理ngStyle

**ngStyle**

<**p [style.color]="red"**>样式的设置</**p**>

<**p [style.font-size.px]="**30**"**>样式的设置</**p**>

*<!--style="font-size: 30px;" 需要单位的需要指定单位-->*

<**p [style.background]="bg"**>样式的设置</**p**>

*<!--bg为指定变量对应的值-->*

<**p [ngStyle]="**{**color**: **'green'**,**fontSize**:**'30px'**}**"**>对象的方式进行样式设置</**p**>

变量对应的颜色设置

red: string = 'red';

bg: string = '#333';

### 类设置:ngClass

<**p [ngClass]="**{**red**: **isShow**}**"**>是否显示呀</**p**>

*<!--isShow为true时添加class='red'-->*

### 添加删除的案例

**import** { Component } **from '@angular/core'**;

**interface** USERS {

**id**: **number**;

**name**: **string**;

}

**const** users: USERS[] = [

{ **id**: 11, **name**: **'Mr. Nice'** },

{ **id**: 12, **name**: **'Narco'** },

{ **id**: 13, **name**: **'Bombasto'** },

];

@Component({

**selector**: **'app-root'**,

**template**: **`**

**<p>添加用户: <input type="text" [(ngModel)]="userName">**

**<button (click)="addUser()">添加用户</button>**

**</p>**

**<ul class="users">**

**<li \*ngFor="let *item* of users,let *i*=index" [ngClass]="{bg: *i*%2==0}">**

**{{ *item*.id }}-{{ *item*.name }} - {{ *i* }}**

**<button (click)="del(*i*)">删除</button>**

**</li>**

**</ul>**

**`**,

**styles**: [**`**

**.bg {**

**background-color: #b7b7b7;**

**color: #0cff24;**

**}**

**`**]

})

**export class** AppComponent {

**users**: USERS[] = users;

**userName**: **string** = **''**;*//表单数据双向绑定*

*//添加的功能*

addUser(): **void** {

**let** lastid: **number** = **this**.**users**[**this**.**users**.**length** - 1].**id**;

**let** user: **any** = { **id**: [++lastid], **name**: **this**.**userName** };

**this**.**users**.push(user);

}

*//删除数据*

del(index: **number**): **void** {

**this**.**users**.splice(index, 1);

}

}

# 组件

## 创建组件

### 创建组件

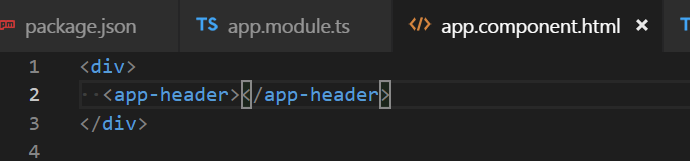
ng g component

ng g c components/header

### 使用组件



在模板使用创建的组件



## 数据

### 数据属性

数据默修饰符为public

// 创建数据默认的修饰符为public

msg = 'xianjs';

num: **number** = 123;//可以知道数据类型

**public** a: **string** = 'public 可以在任何地方使用的';

**protected** b: **string** = 'protected:可以继承使用';

**private** c: **string** = 'private私有的属性,无法继承';

tmp: **string**;//不进行初始化

**constructor**() { //构造函数

//构造函数可以进行数据的初始化

**this**.tmp = 'tmp数据的初始化';

}

### 绑定数据

<**ul**>

<**li**>header子组件的数组展示</**li**>

<**li**>msg = {{ msg }}</**li**>

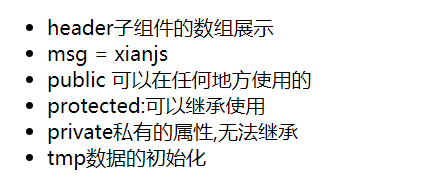
<**li**>{{ a }}</**li**>

<**li**>{{ b }}</**li**>

<**li**>{{ c }}</**li**>

<**li**>{{ tmp }}</**li**>

</**ul**>



### 绑定html

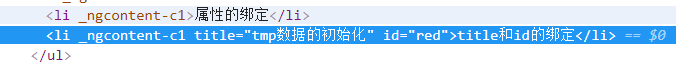
strHtml: **string** = '<h1>世界你好,html可以解析吗</h1>';

<**li [innerHTML]=**"strHtml"></**li**>

## 属性绑定

<**li**>属性的绑定</**li**>

<**li [title]=**"tmp" **[id]=**"red">title和id的绑定</**li**>



## 数据循环

### \*ngFor

//指定数组类型

lists: **string**[] = ['tom', 'jack', 'sam', 'rice'];

<!--可以使用i为索引-->

<**li *\*ngFor*=**"**let** *item* of lists;**let** *i*=index">{{*i*}} - {{ *item* }}</**li**>



### template循环数据

无法使用

## 条件判断

### ngIf判断

<**li *\*ngIf*=**"lists.length===4">长度为4</**li**>

<**li *\*ngIf*=**"lists.length < 4">长度小于4</**li**>

<**li *\*ngIf*=**"lists.length > 1">长度大于1</**li**>

### ngSwitch

<!--[(ngModel)]="msg"双向数据绑定

[ngSwitch]="msg":switch判断的变量

\*ngSwitchCase="2":分支判断

-->

<**input type=**"text" **[(ngModel)]=**"msg">

<**ul [ngSwitch]=**"msg">

<**li *\*ngSwitchCase*=**"1">登陆</**li**>

<**li *\*ngSwitchCase*=**"2">未登录</**li**>

<**li *\*ngSwitchCase*=**"3">注册</**li**>

<**div *\*ngSwitchDefault***>匿名</**div**>

</**ul**>

## 属性绑定

 当要把视图元素的属性 (property) 设置为模板表达式时，就要写模板的属性 (property) 绑定。

    Angular的属性绑定是单向数据绑定，因为值的流动是单向的，从组件的数据属性流动到目标元素的属性。

    对于属性绑定,强调以下几点:

* 不可以使用属性绑定来从目标元素拉取值
* 不可以从绑定目标元素的属性来读取它
* 不可以使用元素绑定来调用目标元素上的方法
* 你只可以设置属性绑定.

### 属性绑定的方式

1. 使用[]进行绑定
2. 使用bind-进行绑定

<img [src]="heroImageUrl">

### attribute 绑定

    我们需要attribute 绑定来创建和绑定到这样的 attribute。

    attribute 绑定的语法与属性绑定类似。

    但方括号中的部分不是元素的属性名，而是由attr前缀，一个点 (.) 和attribute 名字组成。 可以通过值为字符串表达式来设置 attribute 的值。

<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>

### 图片资源src

<!--图片路径可以是网络资源路径,也可以是本地资源图片,默认在assets下找图片-->

<**img src=**"https://ss0.bdstatic.com/5aV1bjqh\_Q23odCf/static/superman/img/logo\_top\_86d58ae1.png">

<**img [src]=**"imgSrc" **alt=**"" **width=**"200" **height=**"200">

imgSrc: **string** = 'assets/img/1.jpg';

## 事件绑定

### 点击事件

<!--事件绑定-->

<**button (dblclick)=**"show('123')">双击</**button**>

<**button (click)=**"show('abc')">事件触发</**button**>

//事件函数

show(str: **string**) {

alert('show 方法');

*console*.log(str);

*console*.log(**this**.lists);//数据的输出

}

### 表单事件

keupFn($event: KeyboardEvent) {

// console.log($event.target.tagName);//标签名称

// console.log($event.target.value);//属性名称

*console*.log($event.keyCode);//获取键盘码

**if** ($event.keyCode === 13) {

*console*.log($event.target.value);

}

}

<**input type=**"text" **placeholder=**"输入你的用户名称" **(keyup)=**"keupFn(*$event*)">

## 双向数据绑定

### 前提条件



导入FormsModule

**import** { FormsModule } **from** '@angular/forms';

imports: [

BrowserModule,

FormsModule

],

<**div**>

双向数据绑定

<**input type=**"text" **[(ngModel)]=**"xianjs">

<**span**> xianjs = {{ xianjs }}</**span**>

</**div**>

## 服务

### 创建服务

创建到指定目录下面

ng g service services/storage

### 使用服务

app.module.ts 里面引入创建的服务

1.app.module.ts 里面引入创建的服务

import { StorageService } from './services/storage.service';

2. NgModule 里面的 providers 里面依赖注入服务

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

*//引入表单相关的模块 才可以用双休数据绑定*

import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { SearchComponent } from './components/search/search.component';

import { TodolistComponent } from './components/todolist/todolist.component';

*//引入并且配置服务*

import { StorageService } from './services/storage.service'; //自定义服务的导入

@NgModule({

declarations: [

AppComponent,

SearchComponent,

TodolistComponent

],

imports: [

BrowserModule,

FormsModule

],

providers: [StorageService], // 注入依赖的服务

bootstrap: [AppComponent]

})

export class AppModule { }

### 在页面使用服务

使用的页面引入服务，注册服务

import { StorageService } from '../../services/storage.service';

constructor(private storage: StorageService) {

}

## 什么周期钩子函数

<https://angular.cn/guide/lifecycle-hooks>

### 生命周期钩子

每个组件都有一个被 Angular 管理的生命周期。

Angular 创建它，渲染它，创建并渲染它的子组件，在它被绑定的属性发生变化时检查它，并在它从 DOM 中被移除前销毁它。

Angular 提供了生命周期钩子，把这些关键生命时刻暴露出来，赋予你在它们发生时采取行动的能力。

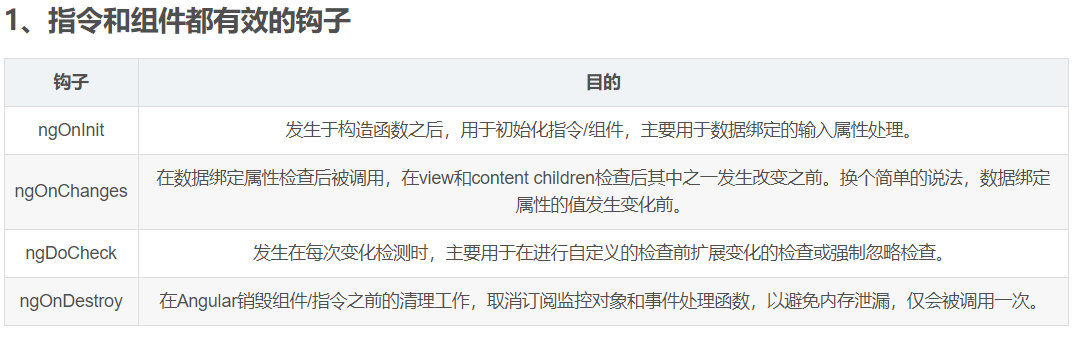
除了那些组件内容和视图相关的钩子外,指令有相同生命周期钩子。

### 生命周期钩子顺序

当 Angular 使用构造函数新建一个组件或指令后，就会按下面的顺序在特定时刻调用这些生命周期钩子方法：

|  |  |
| --- | --- |
| 钩子 | 用途及时机 |
| ngOnChanges() | 当 Angular（重新）设置数据绑定输入属性时响应。 该方法接受当前和上一属性值的 [SimpleChanges](https://angular.cn/api/core/SimpleChanges) 对象  在 ngOnInit() 之前以及所绑定的一个或多个输入属性的值发生变化时都会调用。 |
| ngOnInit() | 在 Angular 第一次显示数据绑定和设置指令/组件的输入属性之后，初始化指令/组件。  在第一轮 ngOnChanges() 完成之后调用，只调用**一次**。  **请求数据在此操作** |
| ngDoCheck() | 检测，并在发生 Angular 无法或不愿意自己检测的变化时作出反应。  在每个变更检测周期中，紧跟在 ngOnChanges() 和 ngOnInit() 后面调用。  数据侦测 |
| [ngAfterContentInit()](https://angular.cn/api/router/RouterLinkActive#ngAfterContentInit) | 当 Angular 把外部内容投影进组件/指令的视图之后调用。  第一次 ngDoCheck() 之后调用，只调用一次。 |
| ngAfterContentChecked() | 每当 Angular 完成被投影组件内容的变更检测之后调用。  [ngAfterContentInit()](https://angular.cn/api/router/RouterLinkActive#ngAfterContentInit) 和每次 ngDoCheck() 之后调用 |
| [ngAfterViewInit()](https://angular.cn/api/forms/NgForm#ngAfterViewInit) | 当 Angular 初始化完组件视图及其子视图之后调用。  ngAfterContentChecked() 之后调用，只调用一次。  DOM操作在此 |
| ngAfterViewChecked() | 每当 Angular 做完组件视图和子视图的变更检测之后调用。  [ngAfterViewInit()](https://angular.cn/api/forms/NgForm#ngAfterViewInit) 和每次 ngAfterContentChecked() 之后调用。 |
| ngOnDestroy() | 每当 Angular 每次销毁指令/组件之前调用并清扫。 在这儿反订阅可观察对象和分离事件处理器，以防内存泄漏。  在 Angular 销毁指令/组件之前调用。  组件销毁操作 , 事件解绑 |

### ngOnInit与constructor的区别



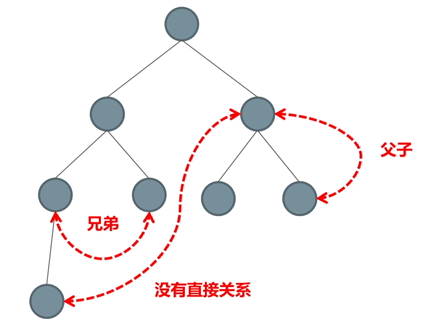




constructor

constructor应该是ES6中明确使用constructor来表示构造函数的，构造函数使用在class中，用来做初始化操作。当包含constructor的类被实例化时，构造函数将被调用。

## 组件通信



### 父子组件中通过 ViewChild 调用子组件的方法

1.调用子组件给子组件定义一个名称

<app-footer #footerChild></app-footer>

2.引入 ViewChild

import { Component, OnInit ,ViewChild} from '@angular/core';

3. ViewChild 和刚才的子组件关联起来

@ViewChild('footerChild') footer;

4.调用子组件

run(){

this.footer.footerRun();

}

*//获取一个组件*

*// @ViewChild('header') header:any;*

getChildRun(){

*//调用子组件里面的方法*

this.header.run();

}

### 父组件给子组件传值-@input

父组件不仅可以给子组件传递简单的数据，还可把自己的方法以及整个父组件传给子组件

1. 父组件调用子组件的时候传入数据

<app-header [msg]="msg"></app-header>

2. 子组件引入 Input 模块

import { Component, OnInit ,Input } from '@angular/core';

3. 子组件中 @Input 接收父组件传过来的数据

export class HeaderComponent implements OnInit {

@Input() msg:string

constructor() { }

ngOnInit() {

}

}

4. 子组件中使用父组件的数据

<h2>这是头部组件--{{msg}}</h2>

### 子组件通过@Output 触发父组件的方法

1. 子组件引入 Output 和 EventEmitter

import { Component, OnInit ,Input,Output,EventEmitter} from '@angular/core';

2.子组件中实例化 EventEmitter

@Output() private outer=new EventEmitter<string>();

/\*用 EventEmitter 和 output 装饰器配合使用 <string>指定类型变量\*/

3. 子组件通过 EventEmitter 对象 outer实例广播数据

sendParent(){

// alert('zhixing');

this.outer.emit('msg from child')

}

4.父组件调用子组件的时候，定义接收事件 , outer 就是子组件的 EventEmitter 对象 outer

<app-header (outer)="runParent($event)"></app-header>

5.父组件接收到数据会调用自己的 runParent 方法，这个时候就能拿到子组件的数据

//接收子组件传递过来的数据

runParent(msg:string){

alert(msg);

}

### 父组件通过@ViewChild 主动获取子组件的数据和方法

1.调用子组件给子组件定义一个名称

<app-footer #footerChild></app-footer>

2. 引入 ViewChild

import { Component, OnInit ,ViewChild} from '@angular/core';

3. ViewChild 和刚才的子组件关联起来

@ViewChild('footerChild') footer;

4.调用子组件

run(){

this.footer.footerRun();

}

### 非父子组件通讯

1、公共的服务

2、Localstorage (推荐)

3、Cookie

## 样式和类

### ngStyle

<**p [style.color]="red"**>样式的设置</**p**>

<**p [style.font-size.px]="**30**"**>样式的设置</**p**>

*<!--style="font-size: 30px;" 需要单位的需要指定单位-->*

<**p [style.background]="bg"**>样式的设置</**p**>

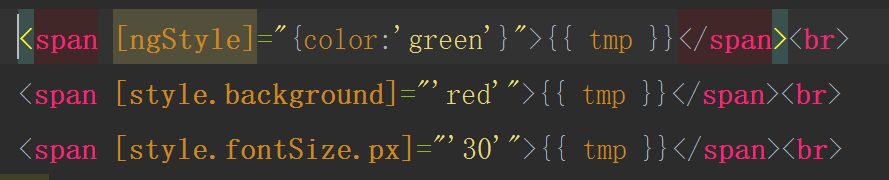
*<!--bg为指定变量对应的值-->*

<**p [ngStyle]="**{**color**: **'green'**,**fontSize**:**'30px'**}**"**>对象的方式进行样式设置</**p**>

变量对应的颜色设置

red: string = 'red';

bg: string = '#333';



### ngClass

<**span [ngClass]=**"{red: **true**}">{{ tmp }}</**span**>

## DOM操作和动画

### DOM操作

#### 原生操作DOM

ngAfterViewInit(){

var boxDom:any=document.getElementById('box');

boxDom.style.color='red';

}

*//视图加载完成以后触发的方法 dom加载完成 （建议把dom操作放在这个里面）*

ngAfterViewInit(){

let oBox1:any=document.getElementById('box1');

console.log(oBox1.innerHTML);

oBox1.style.color="blue";

}

#### Angular 中的 ViewChild操作DOM

ViewChild获取dom节点

1、模板中给dom起一个名字

<div #myBox>

我是一个dom节点

</div>

2、在业务逻辑里面引入ViewChild

import { Component, OnInit,ViewChild} from '@angular/core';

3、 写在类里面 @ViewChild('myBox') myBox:any;

4、ngAfterViewInit生命周期函数里面获取dom

this.myBox.nativeElement

<app-header #header></app-header>

<div #myBox>

我是一个dom节点

</div>

*//获取dom节点*

@ViewChild('myBox') myBox:any;

*//获取一个组件*

@ViewChild('header') header:any;

使用angular进行DOM操作

*// DOM操作*

ngAfterViewInit(): void {

console.log(this.myBox.nativeElement);

this.myBox.nativeElement.style.width='100px';

this.myBox.nativeElement.style.height='100px';

this.myBox.nativeElement.style.background='red';

console.log(this.myBox.nativeElement.innerHTML);

}

### 动画

## RxJS异步编程解决方案

### 简介

RxJS 是 ReactiveX 编程理念的 JavaScript 版本。ReactiveX 来自微软，它是一种针对异步数据流的编程。简单来说，它将一切数据，包括 HTTP 请求，DOM事件或者普通数据等包装成流的形式，然后用强大丰富的操作符对流进行处理，使你能以同步编程的方式处理异步数据，并组合不同的操作符来轻松优雅的实现你所需要的功能。

RxJS 是一种针对异步数据流编程工具，或者叫响应式扩展编程；可不管如何解释 RxJS 其目标就是异步编程，Angular 引入 RxJS 为了就是让异步可控、更简单。

RxJS 里面提供了很多模块。这里我们主要给大家讲 RxJS 里面最常用的 Observable 和fromEvent。

### 异步编程

1、回调函数

2、事件监听/发布订阅

3、Promise

4、Rxjs

### Promise 和 RxJS 处理异步对比

#### Promise 处理异步:

let promise = new Promise(resolve => {

setTimeout(() => {

resolve('---promise timeout---');

}, 2000);

});

promise.then(value => console.log(value));

#### RxJS 处理异步：

import {Observable} from 'rxjs';

let stream = new Observable(observer => {

setTimeout(() => {

observer.next('observable timeout');

}, 2000);

});

stream.subscribe(value => console.log(value));

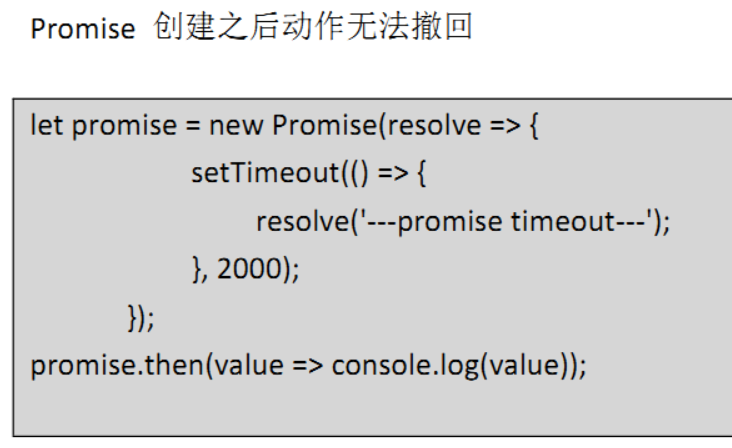
从上面列子可以看到 RxJS 和 Promise 的基本用法非常类似， 除了一些关键词不同。 Promise

里面用的是 then() 和 resolve()，而 RxJS 里面用的是 next() 和 subscribe()。

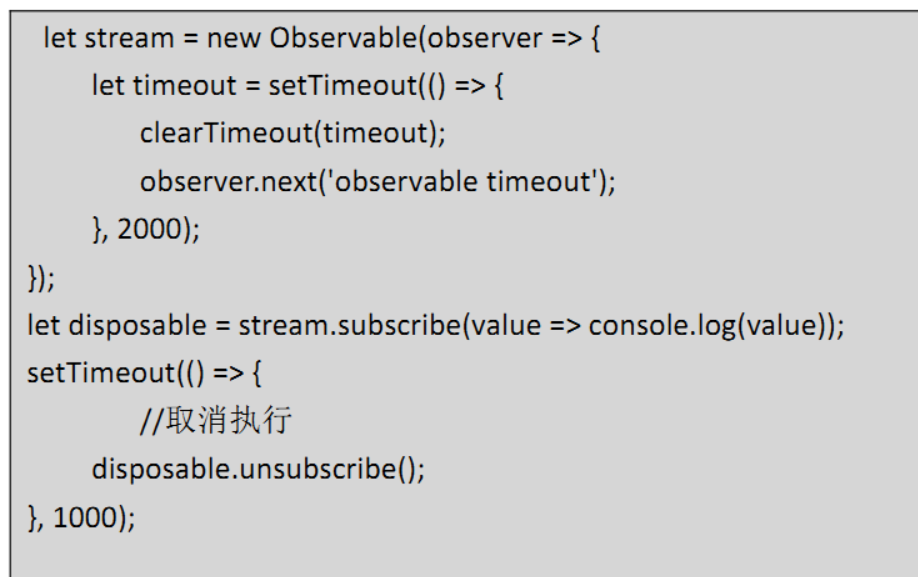
从上面例子我们感觉Promise 和 RxJS 的用法基本相似。 其实Rxjs相比Promise 要强大很多。比如 Rxjs 中可以中途撤回、Rxjs 可以发射多个值、Rxjs 提供了多种工具函数等等。

### Rxjs unsubscribe 取消订阅

Promise的创建之后，动作是无法撤回的。Observable不一样，动作可以通过unsbscribe()方法中途撤回，而且Observable在内部做了智能的处理.



Rxjs 可以通过 unsubscribe() 可以撤回 subscribe 的动作



## 管道

1. 大小写转换

<p>{{str | uppercase}}</p>//转换成大写

<p>{{str | lowercase}}</p>//转换成小写

2. 日期格式转换

<p>{{today | date:'yyyy-MM-dd HH:mm:ss' }}</p>

<p>date = {{ date|date:'yyyy-mm-dd HH:mm:ss' }}</p>

<!--date = 2019-40-31 09:40:02-->

3. 小数位数

//接收的参数格式为{最少整数位数}.{最少小数位数}-{最多小数位数} //保留2~4位小数

<p>{{p | number:'1.2-4'}}</p>

4. JavaScript 对象序列化

<p>{{ { name: 'semlinker' } | json }}</p> <!-- Output: { "name": "semlinker" } -->

5. slice

<p>{{ 'semlinker' | slice:0:3 }}</p> <!-- Output: sem -->

6. 管道链

<p>{{ 'semlinker' | slice:0:3 | uppercase }}</p> <!-- Output: SEM -->

7. 自定义管道

自定义管道的步骤：

使用 [@Pipe](http://bbs.itying.com/user/Pipe) 装饰器定义 Pipe 的 metadata 信息，如 Pipe 的名称 - 即 name 属性

实现 PipeTransform 接口中定义的 transform 方法

1.1 WelcomePipe 定义

import { Pipe, PipeTransform } from '@angular/core';

[@Pipe](/user/Pipe)({ name: 'welcome' })

export class WelcomePipe implements PipeTransform {

transform(value: string): string {

if(!value) return value;

if(typeof value !== 'string') {

throw new Error('Invalid pipe argument for WelcomePipe');

}

return "Welcome to " + value;

}

}

1.2 WelcomePipe 使用

<div>

<p ngNonBindable>{{ 'semlinker' | welcome }}</p>

<p>{{ 'semlinker' | welcome }}</p> <!-- Output: Welcome to semlinker -->

</div>

2.1 RepeatPipe 定义

import {Pipe, PipeTransform} from '@angular/core';

[@Pipe](/user/Pipe)({name: 'repeat'})

export class RepeatPipe implements PipeTransform {

transform(value: any, times: number) {

return value.repeat(times);

}

}

2.2 RepeatPipe 使用

<div>

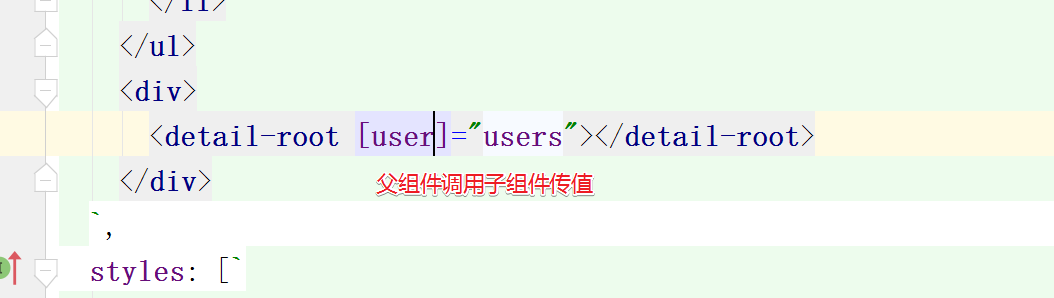
<p ngNonBindable>{{ 'lo' | repeat:3 }}</p>

<p>{{ 'lo' | repeat:3 }}</p> <!-- Output: lololo -->

</div>

## 组件传参:父组件-子组件





组件需要app.module注册



传递的数据过多的时候可以传递对象进行一次传递

### 组件传参:子-->父

#### 子组件

*//相当于发射器*

@Output() **changeNumber**: EventEmitter<**number**> = **new** EventEmitter();

**n**: **number** = 0;

*//子组件的方法*

show(): **void** {

*//触发的父类方法名称,传递参数//触发器*

**this**.**changeNumber**.emit(**this**.**n**++);

}

#### 父组件

父组件的模板文件调用子组件的名称

父组件处理方法对应第一个参数文事件对象

<h1>子组件 app-detail</h1>

<app-detail (changeNumber)="parent(*$event*)"></app-detail>

父组件对应的处理函数

parent(n: number): void {

*console*.log(n);

*console*.log('父组件的parent方法');

}

### #abc获取子组件数据

<app-detail #*abc* (changeNumber)="parent(*$event*)"></app-detail>

*//ViewChild获取组件中的方法属性*

@ViewChild(**'abc'**) **abc**;

ngOnInit(): **void** {*//什么周期初始化,就会调用此函数*

**this**.**num2** = **this**.**abc**.getAbc();

}

#### 子组件

getAbc(): string {

return 'abc';

}



### 组件传参

## 第三方文件的使用

### 安装依赖文件

[angular6 中使用bootstrap](https://www.cnblogs.com/king94Boy/p/9669284.html)

npm install jquery   --save

npm install popper.js   --save

npm install bootstrap   --save

### 配置文件:angular.json

"styles": [

"src/styles.css",

"./node\_modules/bootstrap/dist/css/bootstrap.css"

],

"scripts": [

"./node\_modules/jquery/dist/jquery.min.js",

"./node\_modules/popper.js/dist/umd/popper.js",

"./node\_modules/bootstrap/dist/js/bootstrap.js"

]

### pipe管道

# 数据交互

## 依赖的核心模块

**import** { HttpClientModule } **from** '@angular/common/http';

imports: [

BrowserModule,

FormsModule,

HttpClientModule,

],

## get请求

**import** { HttpClient} **from** '@angular/common/http';

**constructor**(**public** http: HttpClient) {

*console*.log('arguments');

}

getData() {

**let** api: **string** = 'assets/data.json';

**this**.http.get(api).subscribe(res => {

*console*.log(res);

});

}

## post请求

**import** { HttpClient, HttpHeaders } **from** '@angular/common/http';

**constructor**(**public** http: HttpClient) {

*console*.log('arguments');

}

postData() {

**const** httpOption = {

headers: **new** HttpHeaders({ 'Content-Type': 'application/json' })

};

**let** api: **string** = 'http://127.0.0.1/login';

**this**.http.post(api, { username: '李四', pwd: 666 }, httpOption).subscribe(res => {

*console*.log(res);

});

}



## jsonp

配置app.module.ts

HttpClientJsonpModule

imports: [

BrowserModule,

FormsModule,

HttpClientModule,

HttpClientJsonpModule,],

**import** { HttpClient, HttpHeaders } **from** '@angular/common/http';

**constructor**(**public** http: HttpClient) {

}

//jsonp获取数据

jsonpData() {

**let** api: **string** = 'assets/data.json';

**this**.http.jsonp(api, 'callback').subscribe(res => {

*console*.log(res);

});

}

## axios

创建服务模块

🙌 # ng g s services/httpservice

服务的配置

**import** axios **from** 'axios';

// axios获取数据的方法

axiosGet(api: **string**) {

**return new** *Promise*((resolve, reject) => {

axios.get(api).then(data => {

resolve(data);

}, reason => {

reject(reason);

});

});

}

在组件中使用

//导入封装的axios服务

**import** { HttpserviceService } **from** '../../services/httpservice.service';

构造函数配置

**constructor**(**public** http: HttpClient, **public** httpServices: HttpserviceService) {

}

通过方法获取数据

//axios获取数据

axiosData() {

**let** api: **string** = 'assets/data.json';

**this**.httpServices.axiosGet(api).then(data => {

*console*.log(data);

}).catch(error => {

*console*.log(error);

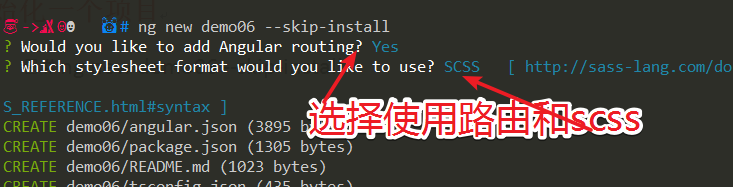
});

}

# 路由

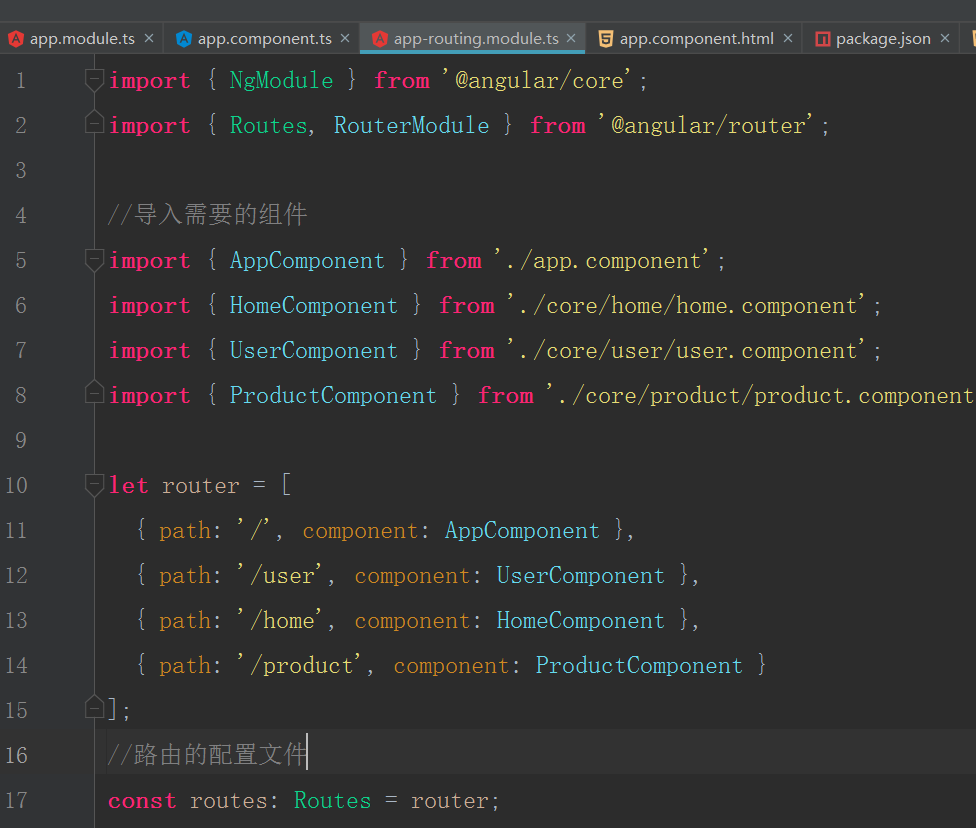
## 初始化一个项目

🎅->🎿☺☻ 🙌 # ng new demo06 --skip-install



## 创建几个组件

## 找到 app-routing.module.ts 配置路由



## 配置

找到 app.component.html 根组件模板，配置 router-outlet 显示动态加载的路由

<**div**>

<!-- <p>

<a href="/">首页</a> |

<a href="/user">用户</a> |

<a href="/product">商品</a> |

<a href="/home">个人中心</a>

</p>-->

<**p**>

<**a href=**"/" **routerLink=**"/">首页</**a**> |

<**a href=**"/" **routerLink=**"/home">用户家目录</**a**> |

<**a href=**"/" **routerLink=**"/user">user</**a**> |

<**a href=**"/" **routerLink=**"/product">商品</**a**> |

</**p**>

</**div**>

<!--路由的挂载点-->

<**router-outlet**></**router-outlet**>

## 路由的配置文件

//导入需要的组件

**import** { AppComponent } **from** './app.component';

**import** { HomeComponent } **from** './core/home/home.component';

**import** { UserComponent } **from** './core/user/user.component';

**import** { ProductComponent } **from** './core/product/product.component';

**let** router = [

//redirectTo :相当于重定向到指定的路由

{ path: 'user', component: UserComponent },

{ path: 'home', component: HomeComponent },

{ path: 'product', component: ProductComponent },

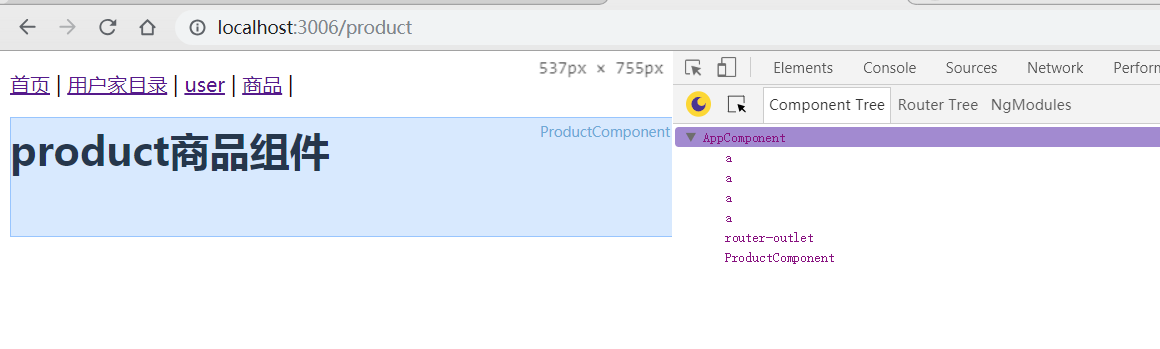
{ path: '\*', redirectTo: 'home' },

];

//路由的配置文件

**const** routes: Routes = router;

## 单页面路由



# 组件的使用

## Ant Design

<https://ng.ant.design/docs/introduce/zh>

<https://ng.ant.design/docs/getting-started/zh>

1.安装脚手架工具

$ npm install -g @angular/cli

2.创建项目

执行以下命令，@angular/cli 会在当前目录下新建一个名称为 PROJECT-NAME 的文件夹，并自动安装好相应依赖。

$ ng new PROJECT-NAME

3.初始化[配置

进入项目文件夹，执行以下命令后将自动完成 ng-zorro-antd 的初始化配置，包括引入国际化文件，导入模块，引入样式文件等工作。

$ ng add ng-zorro-antd

4.启动调试

一键启动调试，运行成功后显示欢迎页面。

$ ng serve --port 0 --open

