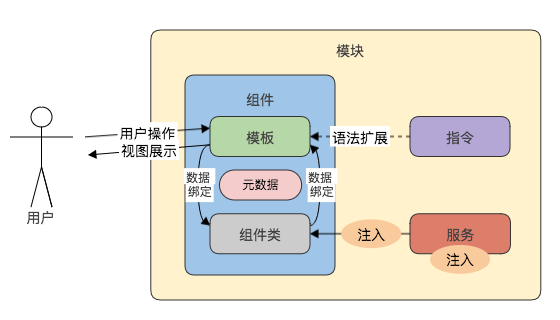
**angular5.0**

1. **简介**

**Angular 2.0 于去年 9 月正式发布，新版 Angular 推出后，官方不再采取AngularJS 这个名称了，而是推荐直接使用 Angular，AngularJS 变为专指第一代框架，所以后文里采用 Angular 均指代 Angular 2+，Angular 彻底重写了 AngularJS采用新架构设计的 Angular 代码更简洁易读、性能更高，更加贴合新时代前端的发展趋势，如基于组件的设计、响应式编程等。除此之外，Angular 适用场景更广，如支持服务端渲染，能更好地适配 Mobile 应用（Mobile Toolkit）、支持离线编译等**

**Angular 官方文档列出了 8 个核心概念，分别是模块、组件、模板、元数据、数据绑定、指令、服务、依赖注入。**



**借助上图，我们来大致解读一下这些核心概念：**

与用户直接交互的是模板，模板并不是独立的模块，它是组成组件的要素之一。另一要素是组件类，用以维护组件的数据模型及功能逻辑；

模板是通过元数据指定的，元数据还包含很多其他的重要信息，这些信息是用来告诉 Angular 如何去解释一个普通的类，如上图所示，元数据结合普通类而构成组件；

指令是 Angular 里的独立构成，他与模板密切关联，用来增强模板特性，间接扩展了模板的语法；

服务也是 Angular 里的独立构成，他是封装单一功能逻辑的单元，通常为组件提供功能扩展；

服务要能为组件所使用，是通过“依赖注入”机制把服务引入到组件内部，服务既可以单独注入到某一组件，亦可注入到模块，两种注入方式使得服务的作用域不一样，后文详解。

Angular 的概念虽多，但理解起来并不难，这些概念中最重要的就是组件。 纵观整个 Angular 应用，接收用户指令，加工处理后输出相应视图的过程中，组件始终处于这个交互的出入口，这正是 Angular 基于组件设计的体现。

**Angular 与 AngularJS 有什么区别**

不再有Controller和 Scope

更好的组件化及代码复用

更好的移动端支持

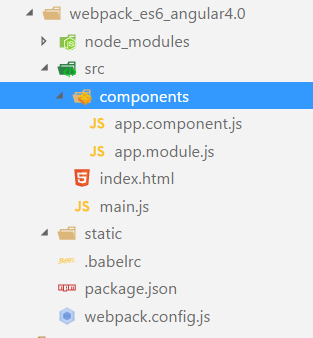
引入了 RxJS 与 Observable

引入了 Zone.js，提供更加智能的变化检测

**typescript规范进行开发：**

由于angular2.0是有谷歌和微软联合出版，所以采用了微软的typescripte规范；

1. **项目构建**
2. **webpack&es6&angular5**
3. **项目目录**



1. **需要的依赖（package.json）**

{

"name": "angular5-es6-webpack",

"version": "1.0.0",

"description": "The Angular 5 Quickstart tutorial done in ES6 with webpack and Babel, without using TypeScript.",

"scripts": {

"start": "webpack-dev-server --progress --open",

"build": "webpack --progress"

},

"repository": {

"type": "git",

"url": ""

},

"author": "",

"license": "ISC",

"bugs": {

"url": ""

},

"homepage": "",

"dependencies": {

"@angular/animations": "^5.0.0",

"@angular/common": "^5.0.0",

"@angular/compiler": "^5.0.0",

"@angular/core": "^5.0.0",

"@angular/forms": "^5.0.0",

"@angular/http": "^5.0.0",

"@angular/platform-browser": "^5.0.0",

"@angular/platform-browser-dynamic": "^5.0.0",

"@angular/router": "^5.0.0",

"core-js": "^2.4.1",

"rxjs": "^5.5.2",

"zone.js": "^0.8.4",

"reflect-metadata": "0.1.2"

},

"devDependencies": {

"babel-core": "^6.7.0",

"babel-loader": "^6.2.4",

"babel-preset-es2015": "^6.6.0",

"concurrently": "^3.5.1",

"css-loader": "^0.28.7",

"html-loader": "^0.5.1",

"html-webpack-plugin": "^2.30.1",

"style-loader": "^0.19.1",

"url-loader": "^0.6.2",

"webpack": "^3.10.0",

"webpack-dev-server": "^2.9.7"

}

}

1. **webpack.confg.js配置**

var webpack = require("webpack")

var HtmlWebpackPlugin = require("html-webpack-plugin")

module.exports = {

entry: \_\_dirname + "/src/main.js",

output: {

path: \_\_dirname + "/dist",

filename: "[name].js",

// chunkFilename: "[name][hash].js"

},

module: {

rules: [

{

test: /\.js$/,

use: ["babel-loader"]

},

{

test: /\.html$/,

loader: 'html-loader',

// query: {

// minimize: true

// }

},

{

test: /\.css$/,

use: ["style-loader", "css-loader"]

},

{

test: /\.png|.jpg$/,

loader: "url-loader"

}]

},

devtool: "cheap-module-eval-source-map",

devServer: {

compress: true,

// 模块的热加载，必须结合 HotModuleReplacementPlugin使用

hot: true,

// inline: true,

port: 9900,

host: 'localhost',

},

plugins: [

// new webpack.optimize.UglifyJsPlugin(),

new webpack.HotModuleReplacementPlugin(),

new webpack.optimize.CommonsChunkPlugin({

name: "commons",

// ( 公共chunk(commnons chunk) 的名称)

filename: "commons.js",

// ( 公共chunk 的文件名)

}),

new HtmlWebpackPlugin({

template: "./src/index.html"

})

]

}

1. **wepack的主入口文件（main.js）**

import 'zone.js/dist/zone.js'

import 'rxjs/bundles/Rx.min.js'

import '@angular/core/bundles/core.umd.js'

import '@angular/common/bundles/common.umd.js'

import '@angular/compiler/bundles/compiler.umd.js'

import '@angular/platform-browser/bundles/platform-browser.umd.js'

import '@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js'

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

import {AppModule} from './components/app.module';

document.addEventListener('DOMContentLoaded', () => {

platformBrowserDynamic().bootstrapModule(AppModule);

});

1. **app组件（app.component.js）**

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

class AppComponent {

static get annotations() {

return [

new Component({

selector: "my-app",

template: '<h1>My First Angular 5 App</h1>'

}),

];

}

constructor () {}

}

export {AppComponent};

1. **app模块（app.module.js）**

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

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

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

class AppModule{

static get annotations() {

return [

new NgModule({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

];

}

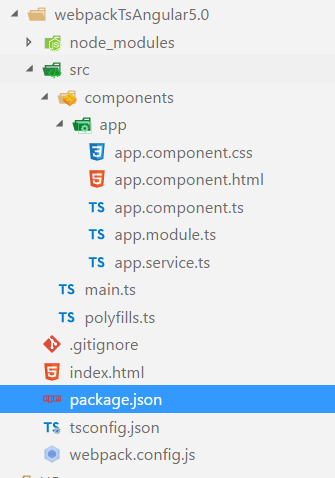
}

export {AppModule};

1. **webpackTsAngular5.0(手动搭建)**

**注：相比上面使用es6搭建，修改webpack.config.js部分loader，即可完成搭建**

1. **项目目录**



1. **改变package.json的依赖项，只需要更改devDependencies即可：**

"devDependencies": {

"@types/jasmine": "^2.5.36",

"@types/node": "^6.0.95",

"angular2-template-loader": "^0.6.2",

"awesome-typescript-loader": "^3.4.1",

"css-loader": "^0.28.7",

"html-loader": "^0.5.1",

"html-webpack-plugin": "^2.30.1",

"source-map-loader": "^0.2.3",

"style-loader": "^0.19.1",

"stylus-loader": "^3.0.1",

"to-string-loader": "^1.1.5",

"typescript": "^2.6.1",

"webpack": "^3.8.1",

"webpack-dev-server": "^2.9.4"

}

1. **添加tsconfig.json:**

{

"compilerOptions": {

"outDir": "./dist", //文件输出目标文件夹，和webpack的output路径一致

"allowJs": true,

"target": "es5" //解析成es5

},

"include": [

"./src/\*\*/\*" //需要解析的文件夹

]

}

1. **webpack.config.js的配置，修改如下loader：**

module: {

rules: [

{

test: /\.tsx?$/,

use: ['awesome-typescript-loader', 'angular2-template-loader']

},

{

test: /\.js$/,

enforce: "pre",

loader: "source-map-loader"

},

{

test: /\.html$/,

loader: 'html-loader',

// query: {

// minimize: true

// }

},

{

test: /\.styl$/,

loader: 'css-loader!stylus-loader'

},

{

test: /\.css$/,

use: ['to-string-loader', 'css-loader']

},

{

test: /\.png|.jpg$/,

loader: "url-loader"

}

]

},

1. **架构：**

**核心：**用 Angular 扩展语法编写 HTML *模板*， 用*组件*类管理这些模板，用*服务*添加应用逻辑， 用*模块*打包发布组件与服务。

注：webpack的入口文件：（main.ts）

import './polyfills' //对于es6支持的接口注册

import '@angular/core/bundles/core.umd.js'

import '@angular/common/bundles/common.umd.js'

import '@angular/compiler/bundles/compiler.umd.js'

import '@angular/platform-browser/bundles/platform-browser.umd.js'

import '@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js'

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

import {AppModule} from './components/app/app.module';

document.addEventListener('DOMContentLoaded', () => {

platformBrowserDynamic().bootstrapModule(AppModule);

});

main.ts的依赖文件（polyfills.ts）

import 'core-js/es6/symbol';

import 'core-js/es6/object';

import 'core-js/es6/function';

import 'core-js/es6/parse-int';

import 'core-js/es6/parse-float';

import 'core-js/es6/number';

import 'core-js/es6/math';

import 'core-js/es6/string';

import 'core-js/es6/date';

import 'core-js/es6/array';

import 'core-js/es6/regexp';

import 'core-js/es6/map';

import 'core-js/es6/weak-map';

import 'core-js/es6/set';

import 'core-js/es7/reflect';

import 'zone.js/dist/zone'; // Included with Angular CLI.

1. **模块：**

每个 Angular 应用至少有一个模块（[根模块](https://www.angular.cn/guide/bootstrapping)），习惯上命名为AppModule。

根模块在一些小型应用中可能是唯一的模块，大多数应用会有很多特性模块，每个模块都是一个内聚的代码块专注于某个应用领域、工作流或紧密相关的功能。

Angular 模块（无论是根模块还是特性模块）都是一个带有@[NgModule](https://www.angular.cn/api/core/NgModule)装饰器的类。

app.module.ts(根模块)

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

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

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

// class AppModule{

// static get annotations() {

// return [

// new NgModule({

// imports: [ BrowserModule ],

// declarations: [ AppComponent ],

// bootstrap: [ AppComponent ]

// })

// ];

// }

// }

@NgModule({

imports: [ BrowserModule ],//本模块声明的组件模板需要的类所在的其它模块。

// providers: [ Logger ], //服务的创建者，并加入到全局服务列表中，可用于应用任何部分。

declarations: [ AppComponent ], // 声明本模块中拥有的视图类。Angular 有三种视图类：组件、指令和管道。

exports: [ AppComponent ], //declarations 的子集，可用于其它模块的组件模板。

bootstrap: [ AppComponent ] // 指定应用的主视图（称为根组件），它是所有其它视图的宿主。只有根模块才能设置bootstrap属性

})

class AppModule{}

export {AppModule};

**注：**

1. ngModule和javaScript模块对比：

NgModule（一个带@[NgModule](https://www.angular.cn/api/core/NgModule)装饰器的类）是 Angular 的基础特性之一；

js的模块，采用的是es6的模块规范，完全遵循es6；

两者需要结合使用

1. angular的模块库：

Angular 提供了一组 JavaScript 模块。可以把它们看做库模块。

每个 Angular 库的名字都带有@angular前缀。

**从angular的模块库中导入模块需要使用：**

import { [BrowserModule](https://www.angular.cn/api/platform-browser/BrowserModule) } from '@angular/platform-[browser](https://www.angular.cn/api/animations/browser)';

**将angular模块库中的模块导入到当前模块：**

imports: [ [BrowserModule](https://www.angular.cn/api/platform-browser/BrowserModule) ],

注意区别两个模块系统；

1. **组件：**

***组件*负责控制屏幕上的一小块区域，我们称之为*视图，*在类中定义组件的应用逻辑，为视图提供支持。 组件通过一些由属性和方法组成的 API 与视图交互，Angular 会创建、更新和销毁组件。 应用可以通过**[**生命周期钩子**](https://www.angular.cn/guide/lifecycle-hooks)**在组件生命周期的各个时间点上插入自己的操作，例如上面声明的ngOnInit()**

app.component.ts:

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

import {NameList} from "./app.service" //引入的服务

export class AppComponent implements OnInit {

name:string

list:Array<any>=[]

constructor(private service:NameList ){

this.list=service.list

}

ngOnInit() {

this.name="test one"

}

}

1. **模板：**

**通过组件的自带的**模板**来定义组件视图，可以使用像<h2>和<p>这样的典型的 HTML 元素，还能使用其它元素。 例如，像\*ngFor、{{hero.name}}、(click)、[hero]和<hero-detail>这样的代码使用了 Angular 的**[**模板语法**](https://www.angular.cn/guide/template-syntax)**。**

app.component.html:

<div>

<h1> {{name}}</h1>

<ul>

<li \*ngFor="let i of list">{{i}}</li>

</ul>

</div>

1. **元数据：**

**只有像下面那样，将元数据添加到组件类上，这个类才能变成真正的组件，否则**app.component.ts**也只能是一个类而已**

app.component.ts（添加元数据后，此时才能为一个真正的组件）

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

import {NameList} from "./app.service"

@Component({

selector: "my-app",

// template: '<h1> Angular 5 App</h1>',

templateUrl: 'app.component.html',

styleUrls: ['app.component.css'],

providers:[NameList]

})

export class AppComponent implements OnInit {

name:string

list:Array<any>=[]

constructor(private service:NameList ){

this.list=service.list

}

ngOnInit() {

this.name="test one"

}

}

**@**[**Component**](https://www.angular.cn/api/core/Component)**装饰器能接受一个配置对象， Angular 会基于这些信息创建和展示组件及其视图。**

**@**[**Component**](https://www.angular.cn/api/core/Component)**的配置项包括：**

**selector**： CSS 选择器，它告诉 Angular 在父级 HTML 中查找<hero-list>标签，创建并插入该组件。

**templateUrl：**组件 HTML 模板的模块相对地址**。**

**providers：** 组件所需服务的依赖注入提供商数组。 这是在告诉 Angular：该组件的构造函数需要一个NameList服务，这样组件就可以从服务中获得数据。

**@**[**Component**](https://www.angular.cn/api/core/Component)**里面的元数据会告诉 Angular 从哪里获取你为组件指定的主要的构建块。**

**模板、元数据和组件类共同描绘出这个视图，叫做组件。**

1. **模板与数据绑定**
2. **数据绑定：（双向数据绑定）**

**{{hero.name}}：**

[插值表达式](https://www.angular.cn/guide/displaying-data#interpolation)在<li>标签中显示组件的hero.name属性的值。

**[hero]：**

[属性绑定](https://www.angular.cn/guide/template-syntax#property-binding)把父组件的值传到子组件属性中。

**(click)：**

[事件绑定](https://www.angular.cn/guide/user-input#click)在用户点击时调用组件的方法。

**[(**[**ngModel**](https://www.angular.cn/api/forms/NgControlStatus)**)]：**

在双向绑定中，数据属性值通过属性绑定从组件流到输入框。用户的修改通过事件绑定流回组件，把属性值设置为最新的值。

1. **显示数据：**
2. **{{}}：**插值
3. **<app-root></app-root> ：**组件的使用，这个过程自动实例化类；
4. **模板类型：**内联模板和外联模板（template,templateUrl）
5. **定义组件变量：**

* **变量初始化：**

export class AppComponent {

title = 'Tour of Heroes';

myHero = 'Windstorm';

}

* **构造函数初始化变量：**

export class AppCtorComponent {

title: string;

myHero: string;

constructor() {

this.title = 'Tour of Heroes';

this.myHero = 'Windstorm';

}

}

1. **ngFor渲染列表：**

<li \*ngFor="let i of list" (click)="changeHandle($event,i)">{{i}}</li>

1. **ngIf进行条件显示：**

<p \*ngIf="heroes.length > 3">There are many heroes!</p>

1. **模板语法：**
2. **样式的绑定：**

**CSS 类：**

**如果isSpecial为true时，才会显示special类名**

<div [class.special]="isSpecial">Special</div>

**多个类名的添加**

<div class="bad curly special" [class]="badCurly">Bad curly</div>

**样式：**

**使用三目运算符：**

<button [style.color]="isSpecial ? 'red' : 'green'">

**设置em单位**

<button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>

**设置%单位**

<button [style.font-size.%]="!isSpecial ? 150 : 50" >Small</button>

1. **Property（数据绑定）：**

**元素的 property：**

<img [src]="heroImageUrl">

**组件的 property：**

<app-hero-detail [hero]="currentHero"></app-hero-detail>

**指令的 property**

<div [[ngClass](https://www.angular.cn/api/common/NgClass)]="{'special': isSpecial}"></div>

1. **事件绑定：**

**元素的事件**

<button (click)="onSave()">Save</button>

**组件的事件**

<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>

**指令的事件**

<div (myClick)="clicked=$event" clickable>click me</div>

**事件的另外一种写法：（on）**

<button on-click="onSave()">On Save</button>

1. **模板引用变量**

<input #phone placeholder="phone number">

<button (click)="callPhone(phone.value)">Call</button>

1. **管道操作符 ( | )**

<div>

Title through a pipe chain:

{{title | uppercase | lowercase}}

</div>

1. **安全操作符（？.）**

The current hero's name is {{currentHero?.name}}

1. **组件样式**
2. **styles属性：（字符串数组）**

@Component({

selector: "list",

template: `

<section>

<h2>list列表页</h2>

<ul>

<li \*ngFor='let i of list'>{{i.name}}</li>

</ul>

</section>

`,

styles: [

'h2 {color:red }',

'ul {list-style:none}'

],

providers:[ListService]

})

1. **styleUrls属性：（css文件路径，数组格式）**

**注：url的根路径是相对于当前component文件的**

@Component({

selector: "my-app",

// template: '<h1>My First Angular 5 App</h1>'

templateUrl:"./app.component.html",

styleUrls:["./app.component.css"]

})

1. **模板内联样式**

@Component({

selector: 'app-hero-controls',

template: `

<style>

button {

background-color: white;

border: 1px solid #777;

}

</style>

<h3>Controls</h3>

<button (click)="activate()">Activate</button>

`

})

1. **模板中的link标签：**

**注：URL是*相对于应用程序根目录的*，它通常是应用的宿主页面index.html所在的地方。 这个样式文件的 URL *不是*相对于组件文件的。这就是为什么范例中的 URL 用src/app/开头**

@Component({

selector: 'app-hero-team',

template: `

<link rel="stylesheet" href="assets/hero-team.component.css">

<h3>Team</h3>

<ul>

<li \*ngFor="let member of hero.team">

{{member}}

</li>

</ul>`

})

1. **CSS @imports 语法**

@import 'hero-details-box.css';

1. **指令**

**在 Angular 中有三种类型的指令：**

组件 — 拥有模板的指令

结构型指令 — 通过添加和移除 DOM 元素改变 DOM 布局的指令

属性型指令 — 改变元素、组件或其它指令的外观和行为的指令。

1. **属性指令**
2. **内置属性指令：**
3. [**NgClass**](https://www.angular.cn/guide/template-syntax#ngClass)**：添加或移除一组CSS类**

如果想要同时添加多个类时，指令可能是最好的选择，不推荐使用css类绑定[class]proporty方式，而添加和删除单个的最佳选择是css类绑定

this.currentClasses = {

'saveable': this.canSave,

'modified': !this.isUnchanged,

'special': this.isSpecial

};

<div [ngClass]="currentClasses"> </div>

1. [**NgStyle**](https://www.angular.cn/guide/template-syntax#ngStyle)**:添加或移除一组CSS样式**

如果想要同时添加多个类时，指令可能是最好的选择，不推荐使用css类绑定[class]proporty方式，而添加和删除单个的最佳选择是css类绑定

this.currentStyles = {

'font-style': this.canSave ? 'italic' : 'normal',

'font-weight': !this.isUnchanged ? 'bold' : 'normal',

'font-size': this.isSpecial ? '24px' : '12px'

};

<div [ngStyle]="currentStyles"></div>

1. [**NgModel**](https://www.angular.cn/guide/template-syntax#ngModel)**- 双向绑定到HTML表单元素**

在使用[ngModel](https://www.angular.cn/api/forms/NgControlStatus)指令进行双向数据绑定之前，我们必须导入[FormsModule](https://www.angular.cn/api/forms/FormsModule)并把它添加到Angular模块的imports列表中

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

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

import { FormsModule } from '@angular/forms'; // <--- JavaScript import from Angular

/\* Other imports \*/

@NgModule({

imports: [

BrowserModule,

FormsModule // <--- import into the NgModule

],

/\* Other module metadata \*/

})

export class AppModule { }

<input [([ngModel](https://www.angular.cn/api/forms/NgControlStatus))]="currentHero.name">

1. **自定义属性指令**

import语句指定了从 Angular 的core库导入的一些符号。

[Directive](https://www.angular.cn/api/core/Directive)提供@[Directive](https://www.angular.cn/api/core/Directive)装饰器功能。

[ElementRef](https://www.angular.cn/api/core/ElementRef)[注入](https://www.angular.cn/guide/dependency-injection)到指令构造函数中。这样代码就可以访问 DOM 元素了。

[Input](https://www.angular.cn/api/core/Input)将数据从绑定表达式传达到指令中。

然后，@[Directive](https://www.angular.cn/api/core/Directive)装饰器函数以配置对象参数的形式，包含了指令的元数据。

@[Directive](https://www.angular.cn/api/core/Directive)装饰器需要一个 CSS 选择器，以便从模板中识别出关联到这个指令的 HTML。

[用于 attribute 的 CSS 选择器](https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors)就是属性名称加方括号。 这里，指令的选择器是[myHighlight]，Angular 将会在模板中找到所有带myHighlight属性的元素。

Angular 会为每个匹配的元素创建一个指令控制器类的实例，并把 Angular 的[ElementRef](https://www.angular.cn/api/core/ElementRef)和[Renderer](https://www.angular.cn/api/core/Renderer)注入进构造函数。 [ElementRef](https://www.angular.cn/api/core/ElementRef)是一个服务，它赋予我们通过它的nativeElement属性直接访问 DOM 元素的能力。[Renderer](https://www.angular.cn/api/core/Renderer)服务允许通过代码设置元素的样式。

**注：render方法：查看官网api**

**指令：heightligth.ts**

import { Directive, ElementRef, Input } from '@angular/core';

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(el: ElementRef，Renderer: Renderer) {

el.nativeElement.style.backgroundColor = 'yellow';

Renderer.setElementStyle(el.nativeElement,"fontSize","30px")

}

}

**注入module中（app.module.ts）**

// 指令

import {HighlightDirective} from "./directive/heightligth"

@NgModule({

imports: [ BrowserModule,FormsModule],

declarations: [

HighlightDirective

],

// providers:[ListService],

bootstrap: [ AppComponent ]

})

**使用的组件（directive.component.ts）**

@[HostListener](https://www.angular.cn/api/core/HostListener)装饰器引用属性型指令的宿主元素，在这个例子中就是<p>。

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

import {ListService} from "./../service/list.service"

@Component({

selector: "directiveEl",

template: `

<section>

<p appHighlight>{{name}}</p>

</section>

`,

styles: [

'h3 {color:gray }',

'ul {list-style:none}',

'.special {color:blue}'

],

providers:[ListService]

})

class DirectiveComponent implements OnInit{

name:string="操作指令"

constructor(){

}

ngOnInit() {

}

}

export {DirectiveComponent};

1. **响应用户引发的事件**

先把[HostListener](https://www.angular.cn/api/core/HostListener)加进导入列表中，同时再添加[Input](https://www.angular.cn/api/core/Input)符号，因为我们很快就要用到它。

**指令：**

import { Directive, ElementRef, Renderer, HostListener, Input } from '@angular/core';

@HostListener('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

@HostListener('click') onClick() {

this.highlight("red");

}

1. **使用数据绑定向指令传递值**

**绑定到@Input别名**

**指令：**

@[Input](https://www.angular.cn/api/core/Input)() highlightColor: string;

**组件：**

<p appHightlight highlightColor="yellow">Highlighted in yellow</p>

<p appHightlight [highlightColor]="'orange'">Highlighted in orange</p>

<p [appHighlight]="color">Highlight me!</p>

**绑定到***@Input***别名**

**指令：**

@[Input](https://www.angular.cn/api/core/Input)('appHighlight') highlightColor: string;

访问属性值时，在指令中使用this.highlightColor,可以访问到red

**组件：**

<p appHighlight='red' [highlightColor]="'orange'">{{name}}</p>

1. **结构指令**

结构型指令的职责是HTML布局。 它们塑造或重塑DOM的结构，比如添加、移除或维护这些元素。

[*属性型*指令](https://www.angular.cn/guide/attribute-directives)会改变某个元素、组件或其它指令的外观或行为。 比如，内置的[NgStyle](https://www.angular.cn/guide/template-syntax#ngStyle)指令可以同时修改元素的多个样式。

我们可以在一个宿主元素上应用多个*属性型*指令，但[只能应用一个](https://www.angular.cn/guide/structural-directives#one-per-element)*结构型*指令

1. **内置结构指令：**
2. [**NgIf**](https://www.angular.cn/guide/template-syntax#ngIf)**- 根据条件把一个元素添加到DOM中或从DOM移除**

<app-hero-detail \*[ngIf](https://www.angular.cn/api/common/NgIf)="isActive"></app-hero-detail>

1. [**NgSwitch**](https://www.angular.cn/guide/template-syntax#ngSwitch)**- 一组指令，用于切换一组视图**

<div [ngSwitch]="currentHero.emotion">

<div \*ngSwitchCase="'happy'" [hero]="currentHero"></div>

<div \*ngSwitchCase="'sad'" [hero]="currentHero"></div>

<div \*ngSwitchCase="'confused'" [hero]="currentHero"></div>

<div \*ngSwitchDefault [hero]="currentHero"></div>

</div>

1. [**NgForOf**](https://www.angular.cn/guide/template-syntax#ngFor)**- 对列表中的每个条目重复套用同一个模板**

<div \*ngFor="let hero of heroes; let i=index">{{i + 1}} - {{hero.name}}</div>

为了提高渲染性能，可以借助trackBy属性来监听改变了得id，然后只针对id改变的dom节点重绘

**组件：**

trackByHeroes(index: number, hero: Hero): number { return hero.id; }

**模板：**

**1、**

<div \*ngFor="let hero of heroes; trackBy: trackByHeroes">

({{hero.id}}) {{hero.name}}

</div>

**2、**

<div \*ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById" [class.odd]="odd">

({{i}}) {{hero.name}}

</div>

**3、**

<ng-template ngFor let-hero [ngForOf]="heroes" let-i="index" let-odd="odd" [ngForTrackBy]="trackById">

<div [class.odd]="odd">({{i}}) {{hero.name}}</div>

<div [class.odd]="odd">({{i}}) {{hero.name}}</div>

</ng-template>

**注：微语法：**

let关键字声明一个[模板输入变量](https://www.angular.cn/guide/structural-directives#template-input-variable)，我们会在模板中引用它。本例子中，这个输入变量就是hero、i和odd。 解析器会把let hero、let i和let odd翻译成命名变量let-hero、let-i和let-odd。

微语法解析器接收of和trackby，把它们首字母大写（of -> Of, trackBy -> TrackBy）， 并且给它们加上指令的属性名（ngFor）前缀，最终生成的名字是ngForOf和ngForTrackBy。 还有两个NgFor的*输入属性*，指令据此了解到列表是heroes，而track-by函数是trackById。

NgFor指令在列表上循环，每个循环中都会设置和重置它自己的*上下文*对象上的属性。 这些属性包括index和odd以及一个特殊的属性名$implicit（隐式变量）。

let-i和let-odd变量是通过let i=index和let odd=odd来定义的。 Angular把它们设置为*上下文*对象中的index和odd属性的当前值。

上下文中的属性let-hero没有指定过，实际上它来自一个隐式变量。 Angular会把let-hero设置为上下文对象中的$implicit属性，NgFor会用当前迭代中的英雄初始化它。

1. **ng-template指令**

Angular会在真正渲染的时候填充<ng-template>的内容，并且把<ng-template>替换为一个供诊断用的注释。

<ng-template [ngIf]="hero">

<div>{{hero.name}}</div>

</ng-template>

1. **<ng-container>**

Angular的<ng-container>是一个分组元素，但它不会污染样式或元素布局，因为 Angular *压根不会把它放进 DOM* 中。

<ng-container>是一个由 Angular 解析器负责识别处理的语法元素。 它不是一个指令、组件、类或接口，更像是 JavaScript 中 if 块中的花括号。

<select [(ngModel)]="hero">

<ng-container \*ngFor="let h of heroes">

<ng-container \*ngIf="showSad || h.emotion !== 'sad'">

<option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>

</ng-container>

</ng-container>

</select>

1. **自定义结构指令：**

导入[Directive](https://www.angular.cn/api/core/Directive)装饰器（而不再是[Component](https://www.angular.cn/api/core/Component)）。

导入符号[Input](https://www.angular.cn/api/core/Input)、[TemplateRef](https://www.angular.cn/api/core/TemplateRef) 和 [ViewContainerRef](https://www.angular.cn/api/core/ViewContainerRef)，我们在*任何*结构型指令中都会需要它们。

给指令类添加装饰器。

设置 CSS *属性选择器* ，以便在模板中标识出这个指令该应用于哪个元素。

以使用[TemplateRef](https://www.angular.cn/api/core/TemplateRef)取得<ng-template>的内容，并通过[ViewContainerRef](https://www.angular.cn/api/core/ViewContainerRef)来访问这个*视图容器*

指令：

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({ selector: '[domHandle]'})

export class DomDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) {

}

@Input() set domHandle(condition: boolean) {

if (condition ) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear();

}

}

}

组件：

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

@Component({

selector: "directiveDom",

template: `

<p \*domHandle="condition">

指令实现dom的操作

</p>

<button (click)="fn()">点击</button>

`,

styles: [

],

})

class domDirectiveComponent implements OnInit{

name:string="操作指令"

condition:boolean=true

constructor(){}

fn(){

this.condition=!this.condition

}

ngOnInit() {}

}

export {domDirectiveComponent};

1. **生命周期函数：**

|  |  |
| --- | --- |
| 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://www.angular.cn/api/core/SimpleChanges)对象  当被绑定的输入属性的值发生变化时调用，首次调用一定会发生在ngOnInit()之前。 |
| 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。是组件获取初始数据的好地方  在第一轮ngOnChanges()完成之后调用，只调用一次。 |
| 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Angular变更检测周期中调用，ngOnChanges()和ngOnInit()之后。 |
| ngAfterContentInit() | 当把内容投影进组件之后调用。  第一次ngDoCheck()之后调用，只调用一次。  只适用于组件。 |
| ngAfterContentChecked() | 每次完成被投影组件内容的变更检测之后调用。  ngAfterContentInit()和每次ngDoCheck()之后调用  只适合组件。 |
| ngAfterViewInit() | 初始化完组件视图及其子视图之后调用。  第一次ngAfterContentChecked()之后调用，只调用一次。  只适合组件。 |
| ngAfterViewChecked() | 每次做完组件视图和子视图的变更检测之后调用。  ngAfterViewInit()和每次ngAfterContentChecked()之后调用。  只适合组件。 |
| ngOnDestroy | 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。 |

1. **组件交互**
2. **输入型绑定把数据从父组件传到子组件**

**子组件：**

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

@Component({

selector: "props-childe",

template: `

<h2> 子组件</h2>

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

<p>{{parentdata}}</p>

`,

})

export class PropsChildeComponent implements OnInit {

@Input() parentdata:string

constructor() {

}

ngOnChanges(changes: SimpleChanges) {

console.log("ngOnChanges")

console.log(changes)

}

ngOnInit() {

}

}

**父组件：**

**模板：**

<div>

<h2> props组件通讯</h2>

<main>

<ul>

<li>

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

</li>

</ul>

<p>{{data}}</p>

<hr />

<props-childe [parentdata]="data"></props-childe>

</main>

</div>

**组件：**

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

@Component({

selector: "props-el",

templateUrl: 'props.component.html',

styleUrls: ['props.component.css'],

})

export class PropsComponent implements OnInit {

public data:string="zhangsan"

constructor() {

}

ngOnInit() {

}

}

1. **通过setter截听输入属性值的变化**

**子组件的改变：**

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

@Component({

selector: "props-childe",

template: `

<h2> 子组件</h2>

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

<p>{{parentdata}}</p>

`,

})

export class PropsChildeComponent implements OnInit {

private \_parentdata:string=""

// @Input() parentdata:string

@Input()

set parentdata(parentdata:string){

this.\_parentdata="$"+parentdata

}

get parentdata():string{return this.\_parentdata}

constructor() {

}

ngOnChanges(changes: SimpleChanges) {

console.log("ngOnChanges")

console.log(changes)

}

ngOnInit() {

}

}

1. **通过***ngOnChanges()***来截听输入属性值的变化**

**子组件的变化：**

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

@Component({

selector: "props-childe",

template: `

<h2> 子组件</h2>

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

<p>{{parentdata}}</p>

`,

})

export class PropsChildeComponent implements OnInit {

@Input() parentdata:string

constructor() {

}

ngOnChanges(changes: SimpleChanges) {

console.log("ngOnChanges")

console.log(changes)

}

ngOnInit() {

}

}

1. **父组件监听子组件的事件**

**子组件：**

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

@Component({

selector: "props-childe",

template: `

<h2> 子组件</h2>

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

<p>{{parentdata}}</p>

`,

})

// 父组件监听子组件的事件

export class PropsChildeComponent implements OnInit {

@Input() parentdata:string

@Output() onVoted=new EventEmitter<boolean>();

ngOnInit() {

}

}

**父组件：**

**模板：**

<div>

<h2> props组件通讯</h2>

<main>

<ul>

<li>

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

</li>

</ul>

<p>{{data}}</p>

<hr />

<props-childe [parentdata]="data" (onVoted)="onVoted($event)"></props-childe>

</main>

</div>

**组件：**

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

@Component({

selector: "props-el",

templateUrl: 'props.component.html',

styleUrls: ['props.component.css'],

})

export class PropsComponent implements OnInit {

public data:string="zhangsan"

constructor() {

}

onVoted(state:boolean):void{

console.log(state)

}

ngOnInit() {

}

}

1. **父组件与子组件通过***本地变量***互动**

**子组件：**

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

@Component({

selector: "props-childe",

template: `

<h2> 子组件</h2>

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

<p>{{parentdata}}</p>

<button (click)="handle()">点击</button>

`,

})

// 父组件监听子组件的事件

export class PropsChildeComponent implements OnInit {

childFn():void{

console.log("由父组件来执行子组件的函数")

}

ngOnInit() {

}

}

**父组件：**

// 父组件与子组件通过本地变量互动

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

@Component({

selector: "props-el",

template: `

<div>

<h2> 父组件与子组件通过本地变量互动</h2>

<main>

<button (click)="child.childFn()">点击子组件的函数</button>

<hr />

<props-childe #child></props-childe>

</main>

</div>

`,

styleUrls: ['props.component.css'],

})

export class PropsComponent implements OnInit {

public data:string="zhangsan"

constructor() {

}

ngOnInit() {

}

}

1. **父组件调用***@ViewChild()*

***本地变量*方法是个简单便利的方法。但是它也有局限性，因为父组件-子组件的连接必须全部在父组件的模板中进行。父组件本身的代码对子组件没有访问权。**

**如果父组件的*类*需要读取子组件的属性值或调用子组件的方法，就不能使用*本地变量*方法。**

**当父组件*类*需要这种访问时，可以把子组件作为*ViewChild*，*注入*到父组件里面。**

**父组件：**

// 父组件调用@ViewChild()

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

import {PropsChildeComponent} from "./child/propsChilde.component"

@Component({

selector: "props-el",

template: `

<div>

<h2> 父组件与子组件通过本地变量互动</h2>

<main>

<button (click)="childFn()">点击触发子组件的函数</button>

<hr />

<props-childe ></props-childe>

</main>

</div>

`,

styleUrls: ['props.component.css'],

})

export class PropsComponent implements OnInit {

public data:string="zhangsan"

@ViewChild(PropsChildeComponent)

private childComponent:PropsChildeComponent

constructor() {

}

childFn(){

this.childComponent.childFn()

}

ngOnInit() {

}

}

**子组件：**

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

@Component({

selector: "props-childe",

template: `

<h2> 子组件</h2>

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

<p>{{parentdata}}</p>

<button (click)="handle()">点击</button>

`,

})

// 父组件监听子组件的事件

export class PropsChildeComponent implements OnInit {

childFn():void{

console.log("由父组件来执行子组件的函数")

}

ngOnInit() {

}

}

1. **父组件和子组件通过服务来通讯 （服务学完后再讲）**

**父组件和它的子组件共享同一个服务，利用该服务在家庭内部实现双向通讯。**

**该服务实例的作用域被限制在父组件和其子组件内。这个组件子树之外的组件将无法访问该服务或者与它们通讯。**

1. **动态组件**

**在这里需要借助服务：**

[ComponentFactoryResolver](https://www.angular.cn/api/core/ComponentFactoryResolver)：动态添加组件；

ViewContainerRef：使用指令来获取dom节点

**封装指令（ad-host）:**

import { Directive, ViewContainerRef } from '@angular/core';

@Directive({

selector: '[ad-host]',

})

export class AdDirective {

constructor(public viewContainerRef: ViewContainerRef) {

// console.log(viewContainerRef)

}

}

**展示组件：**

import { Component,ViewChild ,ComponentFactoryResolver,Input,ComponentRef} from '@angular/core';

import {AdDirective} from "../directives/ad-host"

import {Slot1Component} from './slot1';

import {Slot2Component} from './slot2';

@Component({

selector: "slot",

entryComponents: [ Slot1Component, Slot2Component ],

template: `

<div>

<h2>slot组件</h2>

<button (click)="change()" >切换</button>

<ng-template ad-host></ng-template>

</div>

`,

styles: [

'ul {list-style: none}'

],

})

export class SlotComponent {

private state:boolean=true

@ViewChild(AdDirective) adHost: AdDirective

compRef:ComponentRef<any>

constructor(private componentFactoryResolver: ComponentFactoryResolver) {

}

ngOnInit() {

let factory= this.componentFactoryResolver.resolveComponentFactory(Slot1Component)

this.compRef=this.adHost.viewContainerRef.createComponent(factory)

}

change(){

if (this.compRef) {

this.compRef.destroy();

}

if(this.state){

let factory= this.componentFactoryResolver.resolveComponentFactory(Slot1Component)

this.compRef=this.adHost.viewContainerRef.createComponent(factory)

}else{

let factory= this.componentFactoryResolver.resolveComponentFactory(Slot2Component)

this.compRef=this.adHost.viewContainerRef.createComponent(factory)

}

this.state=!this.state

}

}

**动态展示的组件slot1,slot2**

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

@Component({

template: `

<div>

slot1组件

</div>

`,

styles: [

'ul {list-style: none}'

],

})

export class Slot1Component {

}

1. **管道：**

**管道把数据作为输入，然后转换它，给出期望的输出**

1. **内置的管道(相当于angular1中的过滤器)**

[**DatePipe**](https://www.angular.cn/api/common/DatePipe)**、**[**UpperCasePipe**](https://www.angular.cn/api/common/UpperCasePipe)**、**[**LowerCasePipe**](https://www.angular.cn/api/common/LowerCasePipe)**、**[**CurrencyPipe**](https://www.angular.cn/api/common/CurrencyPipe)**和**[**PercentPipe**](https://www.angular.cn/api/common/PercentPipe)

**组件：pipe.component.ts**

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

@Component({

selector: "pipe",

template: `

<section>

<h3>pipe组件</h3>

<p>{{date|date}}</p>

<p>{{date| date:"MM/dd/yy" }}</p>

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

<p>{{date| date:format }}</p>

</section>

`,

styles: [

'h3 {color:gray }',

'ul {list-style:none}',

'.special {color:blue}'

]

})

class PipeComponent {

date=new Date(1988, 3, 15)

format:string="shortTime"

}

export {PipeComponent};

1. **链式操作**

<p>{{ date | date | uppercase}}</p>

1. **自定义管道**

**管道是一个带有“管道元数据(pipe metadata)”装饰器的类。**

**这个管道类实现了**[**PipeTransform**](https://www.angular.cn/api/core/PipeTransform)**接口的transform方法，该方法接受一个输入值和一些可选参数，并返回转换后的值。**

**当每个输入值被传给transform方法时，还会带上另一个参数，比如我们这个管道中的exponent(放大指数)。**

**我们通过@**[**Pipe**](https://www.angular.cn/api/core/Pipe)**装饰器告诉Angular：这是一个管道。该装饰器是从Angular的core库中引入的。**

**这个@**[**Pipe**](https://www.angular.cn/api/core/Pipe)**装饰器允许我们定义管道的名字，这个名字会被用在模板表达式中。它必须是一个有效的JavaScript标识符。 比如，我们这个管道的名字是exponentialStrength。**

**组件：**

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

@Component({

selector: "pipe",

template: `

<section>

<h3>pipe组件</h3>

<ul>

<li \*ngFor="let i of list | upToCase:'2'">{{i}}</li>

</ul>

</section>

`,

styles: [

'h3 {color:gray }',

'ul {list-style:none}',

'.special {color:blue}'

]

})

class PipeComponent {

list:Array<any>=[1,2,3,4,5]

}

export {PipeComponent};

**模块：**

// pipe管道

import {upToCase} from "./pipe/upToCase"

@NgModule({

imports: [ BrowserModule,FormsModule],

declarations: [

upToCase

],

bootstrap: [ AppComponent ]

})

class AppModule{

}

export {AppModule}

**管道：（upToCase.ts）**

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

@Pipe({name: 'upToCase'})

export class upToCase implements PipeTransform {

transform(value: Array<any>, exponent: string): Array<any> {

let data=null

if(!isNaN(parseInt(exponent))){

data=value.splice(0,parseInt(exponent))

}else{

data=value

}

console.log(222)

return data

}

}

1. **form表单**
2. **用户输入获取值**
3. **$event获取值：**

template: `

<input (keyup)="onKey($event)">

<p>{{values}}</p>

export class KeyUpComponent\_v1 {

values = '';

onKey(event: any) { // without type info

this.values += event.target.value + ' | ';

}

}

1. **从一个模板引用变量中获得用户输入**

@Component({

selector: 'app-loop-back',

template: `

<input #box (keyup)="0">

<p>{{box.value}}</p>

`

})

export class LoopbackComponent { }

1. **按键事件过滤（通过**key.enter**）**

@Component({

selector: 'app-key-up3',

template: `

<input #box (keyup.enter)="onEnter(box.value)">

<p>{{value}}</p>

`

})

export class KeyUpComponent\_v3 {

value = '';

onEnter(value: string) { this.value = value; }

}

1. **失去焦点事件 (blur)**

@Component({

selector: 'app-key-up4',

template: `

<input #box

(keyup.enter)="update(box.value)"

(blur)="update(box.value)">

<p>{{value}}</p>

`

})

export class KeyUpComponent\_v4 {

value = '';

update(value: string) { this.value = value; }

}

1. **表单验证：**
2. **模板驱动验证**

**<input>元素带有一些HTML验证属性：required 和 minlength。它还带有一个自定义的验证器指令forbiddenName。#name="**[**ngModel**](https://www.angular.cn/api/forms/NgControlStatus)**"把**[**NgModel**](https://www.angular.cn/api/forms/NgModel)**导出成了一个名叫name的局部变量。**[**NgModel**](https://www.angular.cn/api/forms/NgModel)**把自己控制的**[**FormControl**](https://www.angular.cn/api/forms/FormControl)**实例的属性映射出去，让我们能在模板中检查控件的状态，比如valid和dirty。要了解完整的控件属性，参见 API 参考手册中的**[**AbstractControl**](https://www.angular.cn/api/forms/AbstractControl)**。**

**<div>元素的揭露了一套嵌套消息，但是只在有“name”错误和控制器为或者。**

**每个嵌套的<div>为其中一个可能出现的验证错误显示一条自定义消息。比如 required、minlength和 forbiddenName。**

**案例：**

**form1.component.ts**

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

@Component({

selector: "form-com1",

template: `

<input id="name" name="name" class="form-control" required minlength="4" [(ngModel)]="hero.name" #name="ngModel">

<div \*ngIf="name.invalid && (name.dirty || name.touched)">

<div \*ngIf="name.errors.required">

这里为必填项

</div>

<div \*ngIf="name.errors.minlength">

最少不能4位

</div>

</div>

`,

styles: [

'h2 {color:red }',

'ul {list-style:none}',

'label {color:blue}',

'li div {font-size:12px; color:red;padding-left:30px;}'

]

})

class FormComponent1 implements OnInit {

constructor() {}

ngOnInit() {}

}

export { FormComponent1 };

1. **自定义校验器与组件类中进行数据的验证：**

**真正的源码都在组件类中。我们不应该通过模板上的属性来添加验证器，而应该在组件类中直接把验证器函数添加到表单控件模型上（**[**FormControl**](https://www.angular.cn/api/forms/FormControl)**）。然后，一旦控件发生了变化，Angular 就会调用这些函数。**

**有两种验证器函数：同步验证器和异步验证器。**

a.同步验证器函数接受一个控件实例，然后返回一组验证错误或null。我们可以在实例化一个[FormControl](https://www.angular.cn/api/forms/FormControl)时把它作为构造函数的第二个参数传进去。

b.异步验证器函数接受一个控件实例，并返回一个承诺（Promise）或可观察对象（Observable），它们稍后会发出一组验证错误或者null。我们可以在实例化一个[FormControl](https://www.angular.cn/api/forms/FormControl)时把它作为构造函数的第三个参数传进去。

**注：所有的异步验证器都会在同步验证器后执行**

**form2.component.ts**

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

import { FormControl, FormGroup, Validators,FormBuilder, ValidatorFn,AbstractControl } from '@angular/forms';

// 自定义校验器（同步）

let forbiddenNameValidator= function (nameRe: RegExp): ValidatorFn {

return (control: AbstractControl): {[key: string]: any} => {

const forbidden = nameRe.test(control.value);

console.log(forbidden)

return forbidden ? null : {'forbiddenName': control.value};

};

}

@Component({

selector: "form-com2",

template: `

<div [formGroup]="heroForm">

<input id="name" class="form-control" formControlName="name" (blur)="fn()">

<div \*ngIf="name.invalid && (name.dirty || name.touched)">

<div \*ngIf="name.errors.required">

这里为必填项

</div>

<div \*ngIf="name.errors.minlength">

最少不能4位

</div>

<div \*ngIf="name.errors.forbiddenName">

没有包含"广"的字段

</div>

</div>

</div>

`,

styles: [

'h2 {color:red }',

'ul {list-style:none}',

'label {color:blue}',

'li div {font-size:12px; color:red;padding-left:30px;}'

]

})

class FormComponent2 implements OnInit {

constructor() {}

heroForm:FormGroup

get name() { return this.heroForm.get('name'); }

ngOnInit(): void {

this.heroForm = new FormGroup({

'name': new FormControl("", [

Validators.required,

Validators.minLength(4),

forbiddenNameValidator(/广/i)

])

});

}

fn(){

console.log(this.heroForm.get("name").errors)

}

}

export { FormComponent2 };

1. **实现form表单的响应式与常用验证**

**组件（form.component.ts）**

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

import { FormControl, FormGroup, Validators,FormBuilder } from '@angular/forms';

import { SelfValidator } from '../validator/validators';

@Component({

selector: "form-com",

template: `

<form [formGroup]="\_formData" novalidate>

<ul >

<li >

<label for="name">姓名</label>

<input formControlName="name" (blur)="changeFn('name')">

<div [hidden]='formData.name.state'>用户名是必填项</div>

</li>

<li>

<label for="phone">手机号</label>

<input formControlName="phone" id="phone" (blur)="changeFn('phone')">

<div [hidden]='formData.phone.state'>请正确输入手机号</div>

</li>

<li>

<label for="sex">性别：</label>

<p>

<span>男：</span><input type="radio" formControlName="sex" value="男" name="sex" (blur)="changeFn('sex')">

</p>

<p>

<span>女：</span><input type="radio" formControlName="sex" value="女" name="sex" (blur)="changeFn('sex')">

</p>

<div [hidden]='formData.sex.state'>选择性别</div>

</li>

<li formGroupName="password">

<p><label>密码:</label><input type="text" formControlName="passwordval" (blur)="changeFn('password')"/></p>

<p><label>确认密码:</label><input type="text" formControlName="pwconfrim" (blur)="changeFn('password')"/></p>

<div [hidden]='formData.password.state'>请正确输入密码</div>

</li>

<li>

<label for="area" >地区：</label>

<select name="" id="area" formControlName="area" (blur)="changeFn('area')">

<option \*ngFor="let i of optionList" [value]="i" >{{i}}</option>

</select>

<div [hidden]='formData.area.state'>请选择地区</div>

</li>

<li>

<button (click)="submit()">提交</button>

</li>

</ul>

</form>

<p>{{\_formData.value | json}}</p>

`,

styles: [

'h2 {color:red }',

'ul {list-style:none}',

'label {color:blue}',

'li div {font-size:12px; color:red;padding-left:30px;}'

]

})

class FormComponent implements OnInit {

private optionList: Array<any>=["北京","上海","南京","天津"]

constructor(private fb: FormBuilder) {

this.createForm();

}

// 创建一个formGroup模型

\_formData: FormGroup;

// 校验的状态对象

ValidatorObjState={

love:true,

area:true,

}

// 存储数据的对象

formData={

name:{

val:"张三",

state:true

},

phone:{

val:"",

state:true

},

password:{

val:"",

two:"",

state:true

},

sex:{

val:"",

state:true

},

area:{

val:"",

state:true

},

}

SelfValidator:SelfValidator //封装的校验器类；

createForm() {

this.SelfValidator=new SelfValidator()

this.\_formData = this.fb.group({

name: [this.formData.name.val, Validators.required ],

phone: [this.formData.phone.val, this.SelfValidator.mobileValidator ],

sex: [this.formData.sex.val, Validators.required ],

area: [this.formData.area.val, Validators.required ],

password: this.fb.group({

passwordval: [this.formData.password.val,[Validators.required,Validators.minLength(6)]],

pwconfrim: [this.formData.password.two,Validators.required]

},{validator :this.SelfValidator.euqalValidator})

});

}

changeFn(name:string,event:any){

// 每次失去焦点时，将此表单的校验状态赋给相应的state

this.formData[name].state=this.\_formData.get(name).valid

}

submit(){

// 点击提交时，将\_formData数据模型的数据添加到formData对象上

Object.keys(this.formData).forEach((i)=>{

if(i=="password"){

this.formData[i].val=this.\_formData.get([i]).get("passwordval").value

}else{

this.formData[i].val=this.\_formData.get([i]).value

}

})

if(this.\_formData.valid){ //所有表单都合法才打印表单的值

console.log(this.formData);

// 将所有的数据进行格式化，符合后台提交所用数据；

var arr=Object.keys(this.formData).map((i)=>{

return {[i]:this.formData[i].val}

})

var obj=Object.assign({},...arr)

console.log(obj)

}else{

// 当校验不成功时，将所有的未填项的state变为false，让提示字段显示出来；

Object.keys(this.formData).map((i)=>{

let $i=this.formData[i]

if($i.val==""||!$i.val){

this.formData[i].state=false

}

})

}

}

ngOnInit() {

}

}

export { FormComponent };

**自定义校验器（validators.ts）:**

import { FormControl, FormGroup ,Validators} from '@angular/forms';

export class SelfValidator {

mobileValidator(control: FormControl): any {

let myReg = /^1(3|4|5|7|8)+\d{9}$/;

let valid = myReg.test(control.value);

console.log("moblie的校验结果是" + valid)

return valid ? null : { mobile: true };//如果valid是true 返回是null

}

euqalValidator(group: FormGroup): any {

let password: FormControl = group.get("passwordval") as FormControl;

let pwconfrim: FormControl = group.get("pwconfrim") as FormControl;

let valid: boolean = (password.value === pwconfrim.value);

console.log("密码校验结果是" + valid);

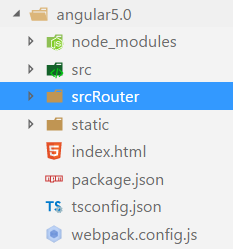
return valid ? null : { equal:true };

}

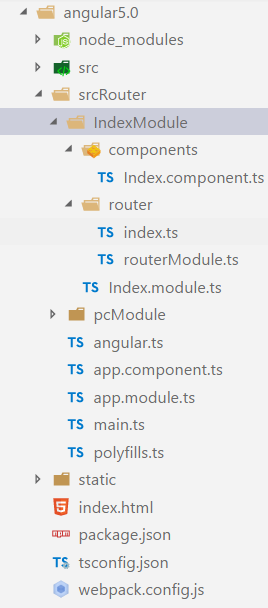
}

1. **路由与导航**

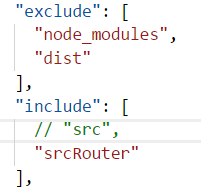
**注：项目结构有所变化：如下：**



**路由之前的代码全部在src中，创建新的srcRouter文件目录：**



**注意将tsconfig.json进行如下修改**



**webpack.config.ts的重新配置：**



1. **基本使用：**

**路由配置文件（router/index.ts）**

import { Routes } from '@angular/router';

import {IndexComponent} from "../components/Index.component"

const Components=[IndexComponent]

const IndexRoutes: Routes = [

{ path: 'tab', component: IndexComponent },

// 表示应用的默认路径，当URL为空时就会访问那里，因此它通常会作为起点。 这个默认路由会重定向到URL /heroes，并显示HeroesListComponent。

{ path: '',

redirectTo: '/heroes',

pathMatch: 'full'

},

// 当所请求的URL不匹配前面定义的路由表中的任何路径时，路由器就会选择此路由。 这个特性可用于显示“404 - Not Found”页，或自动重定向到其它路由。

{ path: '\*\*', component: IndexComponent }

];

export {Components,IndexRoutes}

**模块文件（Index.module.ts）**

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

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

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

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

import { RouterModule } from '@angular/router';

// 路由

import {Components,IndexRoutes} from "./router/index"

// 路由模块的引入

// import { Components,IndexRoutingModule } from "./router/routerModule"

@NgModule({

imports: [BrowserModule, FormsModule, ReactiveFormsModule, HttpClientModule,

RouterModule.forRoot(

IndexRoutes,

// { enableTracing: true } // <-- debugging purposes only

),

// IndexRoutingModule

],

declarations: [...Components]

})

class IndexModule {

}

export { IndexModule }

**路由出口（index.html）:**

**注意：在head中添加<base href="/">，并提供一个根组件；**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

<style>

\*{margin:0;padding:0;}

</style>

<base href="/">

</head>

<body>

<my-app></my-app>

</body>

</html>

**app根组件配置出口（app.component.ts）:**

**路由器会在<router-outlet>标签中显示视图。**

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

@Component({

selector: "my-app",

template: `

<div class="main">

<router-outlet></router-outlet>

</div>`,

styles:[

'\* {margin:0;padding:0}'

]

})

class AppComponent{

}

export {AppComponent};

**创建Index.component.ts展示组件：**

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

import { ActivatedRoute } from '@angular/router';

@Component({

template: `

<section>

<h3>首页，默认路由</h3>

</section>

`,

styles: [

'h3 {color:gray }',

'ul {list-style:none}',

'.special {color:blue}'

]

})

class IndexComponent{

constructor(route: ActivatedRoute){

console.log(1111111)

console.log(route.params)

}

}

export {IndexComponent};

1. **路由的参数获取：**
2. **路由的几种参数形式：**

{ path: 'tab', component: TabComponent },

// 动态路由片段

{ path: 'hero/:id', component: HeroDetailComponent },

// 路由传递数据

{

path: 'heroes',

component: HeroListComponent,

data: { title: 'Heroes List' }

},

// 表示应用的默认路径，当URL为空时就会访问那里，因此它通常会作为起点。 这个默认路由会重定向到URL /heroes，并显示HeroesListComponent。重定向路由需要一个pathMatch属性，来告诉路由器如何用URL去匹配路由的路径，否则路由器就会报错。 在本应用中，路由器应该只有在*完整的URL*等于''时才选择HeroListComponent组件，因此我们要把pathMatch设置为'full'。

{ path: '',

redirectTo: '/heroes',

pathMatch: 'full'

},

// 当所请求的URL不匹配前面定义的路由表中的任何路径时，路由器就会选择此路由。 这个特性可用于显示“404 - Not Found”页，或自动重定向到其它路由。

{ path: '\*\*', component: PageNotFoundComponent }

1. **路由参数的获取：**

|  |  |
| --- | --- |
| url | 路由路径的Observable对象，是一个由路由路径中的各个部分组成的字符串数组。 |
| data | 一个Observable，其中包含提供给路由的data对象。也包含由[解析守卫（resolve guard）](https://www.angular.cn/guide/router#resolve-guard)解析而来的值。 |
| paramMap | 一个Observable，其中包含一个由当前路由的必要参数和[可选参数](https://www.angular.cn/guide/router#optional-route-parameters)组成的[map](https://www.angular.cn/api/router/ParamMap)对象。用这个map可以获取来自同名参数的单一值或多重值。 |
| queryParamMap | 一个Observable，其中包含一个对所有路由都有效的[查询参数](https://www.angular.cn/guide/router#query-parameters)组成的[map](https://www.angular.cn/api/router/ParamMap)对象。 用这个map可以获取来自查询参数的单一值或多重值。 |
| fragment | An Observable of the URL [fragment](https://www.angular.cn/guide/router#fragment) available to all routes. |
| outlet | 要把该路由渲染到的[RouterOutlet](https://www.angular.cn/api/router/RouterOutlet)的名字。对于无名路由，它的路由名是primary，而不是空串。 |
| routeConfig | 用于该路由的路由配置信息，其中包含原始路径。 |
| parent | 当该路由是一个[子路由](https://www.angular.cn/guide/router#child-routing-component)时，表示该路由的父级[ActivatedRoute](https://www.angular.cn/api/router/ActivatedRoute)。 |
| firstChild | 包含该路由的子路由列表中的第一个[ActivatedRoute](https://www.angular.cn/api/router/ActivatedRoute)。 |
| children | 包含当前路由下所有已激活的[子路由](https://www.angular.cn/guide/router#child-routing-component)。 |

1. **路由事件**

|  |  |
| --- | --- |
| [NavigationStart](https://www.angular.cn/api/router/NavigationStart) | 本[事件](https://www.angular.cn/api/router/NavigationStart)会在导航开始时触发。 |
| [RoutesRecognized](https://www.angular.cn/api/router/RoutesRecognized) | 本[事件](https://www.angular.cn/api/router/RoutesRecognized)会在路由器解析完URL，并识别出了相应的路由时触发 |
| [RouteConfigLoadStart](https://www.angular.cn/api/router/RouteConfigLoadStart) | 本[事件](https://www.angular.cn/api/router/RouteConfigLoadStart)会在[Router](https://www.angular.cn/api/router/Router)对一个路由配置进行[惰性加载](https://www.angular.cn/guide/router#asynchronous-routing)之前触发。 |
| [RouteConfigLoadEnd](https://www.angular.cn/api/router/RouteConfigLoadEnd) | 本[事件](https://www.angular.cn/api/router/RouteConfigLoadEnd)会在路由被惰性加载之后触发。 |
| [NavigationEnd](https://www.angular.cn/api/router/NavigationEnd) | 本[事件](https://www.angular.cn/api/router/NavigationEnd)会在导航成功结束之后触发。 |
| [NavigationCancel](https://www.angular.cn/api/router/NavigationCancel) | 本[事件](https://www.angular.cn/api/router/NavigationCancel)会在导航被取消之后触发。 这可能是因为在导航期间某个[路由守卫](https://www.angular.cn/guide/router#guards)返回了false。 |
| [NavigationError](https://www.angular.cn/api/router/NavigationError) | 这个[事件](https://www.angular.cn/api/router/NavigationError)会在导航由于意料之外的错误而失败时触发。 |

1. **路由指令集：**

[**RouterLink**](https://www.angular.cn/api/router/RouterLink)**、**[**RouterLinkActive**](https://www.angular.cn/api/router/RouterLinkActive)**和**[**RouterOutlet**](https://www.angular.cn/api/router/RouterOutlet)**是由**[**RouterModule**](https://www.angular.cn/api/router/RouterModule)**包提供的指令**

<li routerLink="/pc" routerLinkActive="active">pc页面</li>

1. **将路由配置信息封装成模块**

**Index.module.ts文件：**

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

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

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

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

import { RouterModule } from '@angular/router';

// 路由

// import {Components,IndexRoutes} from "./router/index"

// 路由模块的引入

import { Components,IndexRoutingModule } from "./router/routerModule"

@NgModule({

imports: [BrowserModule, FormsModule, ReactiveFormsModule, HttpClientModule,

// RouterModule.forRoot(

// IndexRoutes,

// // { enableTracing: true } // <-- debugging purposes only

// ),

IndexRoutingModule

],

declarations: [...Components]

})

class IndexModule {

}

export { IndexModule }

**routerModule文件:**

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

import { RouterModule, Routes } from '@angular/router';

import {IndexComponent} from "../components/Index.component"

const Components=[

IndexComponent

]

const IndexRoutes: Routes = [

{ path: '', component: IndexComponent },

// 设置默认路由

{ path: '',

redirectTo: '/tab',

pathMatch: 'full'

},

// 如果404时相应的页面

{ path: '\*\*', component: IndexComponent }

];

@NgModule({

imports: [

RouterModule.forRoot(

IndexRoutes,

// { enableTracing: true } // <-- debugging purposes only

)

],

exports: [

RouterModule

]

})

class IndexRoutingModule {}

export {Components,IndexRoutingModule}

**app.module.ts文件的改变：**

**注意：下面两个路由模块引入app根模块时的顺序，默认路由的配置等权限比较大的路由配置模块一定要往前放；**

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

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

import { RouterModule } from '@angular/router';

// 引入项目的根组件

import {AppComponent} from "./app.component"

// 引入首页部分的模块

import { IndexModule } from "./IndexModule/Index.module"

// 引入个人中心模块的路由

import { PcModule } from "./PcModule/Pc.module"

@NgModule({

imports: [

BrowserModule,

RouterModule,

PcModule, //引入个人中心模块

IndexModule , //引入首页部分的模块

],

declarations:[AppComponent],

bootstrap:[AppComponent]

})

class AppModule {}

export { AppModule }

1. **配置子路由以及动态路由**

**路由配置文件：**

const IndexRoutes: Routes = [

{

path: 'pc',

component: PcComponent ,

children:[

{

path: 'add',

component: AddComponent

},

{

path: 'add/:id',

component: AddComponent

}

]

},

];

**页面设置点击**

<a routerLink="/pc/add" routerLinkActive="active">add页面</a>

<a routerLink="/pc/add/123" routerLinkActive="active">add页面动态路由传参</a>

**add.component.ts文件进行参数的获取**

class AddComponent{

constructor(route: ActivatedRoute){

console.log(route.value) //{id:123}

}

}

export {AddComponent};

1. **多重路由视图的定义：**

**承接路由的接口：**

<router-outlet name="popup"></router-outlet>

**配置第二路由的参数：**

{

path: 'compose',

component: ComposeMessageComponent,

outlet: 'popup'

},

**页面中进行跳转：**

<a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>

1. **对于动态路由片段定义的路由，更新时获取片段新值：**

**注入rx的switchMap，在polyfills文件中注入；**

import 'rxjs/add/operator/switchMap';

**在组件中对路由片段的监听：**

class AddComponent{

constructor(private route: ActivatedRoute,

private router: Router){

}

ngOnInit() {

this.route.paramMap.switchMap((params:ParamMap)=>{

console.log(params.get("id"))

return [{id:params.get("id")}]

}).subscribe(hero =>{

console.log(hero)

});

}

}

1. **路由的跳转**

**动态路由片段的js跳转添加值：**

this.router.navigate(["/pc/add",1234])

***矩阵URL*标记法：**

this.router.navigate(["/pc/add",{name:"zhangsan"}])