class , [class] , [ngClass] 的使用与区别：

* class

<input [(ngModel)]="name" class="nFont" class="nColor" /><br>

最后的会覆盖前面的

<input [(ngModel)]="name" class="nFont nBorder" class="nColor nBold" /><br>

最后结果： <input \_ngcontent-c0="" class="nColor nBold ng-valid ng-dirty ng-touched">

* class {{ }}

<input [(ngModel)]="name" class="{{myClass}}" class="nColor nBold" /><br>

class=”{{ class\_variable }}” 会覆盖 class=”xxx xxx”

public myClass = "nFont nBorder";

输出结果是：

<input \_ngcontent-c0="" class="**nFont nBorder** ng-valid ng-dirty ng-touched">

相同的， 后面的会覆盖前面的

<input [(ngModel)]="name" class="{{myClass}}" class="{{hisClass}}" /><br>

public myClass = "nFont nBorder";

public hisClass = "nColor nBold";

输出结果：

<input \_ngcontent-c0="" class="nColor nBold ng-valid ng-dirty ng-touched">

* [class] = “表达式或者变量”

[class] 覆盖 class=”xxx”

<input [(ngModel)]="name" [class]="myClass" class="nColor nBold" /><br>

public myClass = "nFont nBorder";

输出结果： <input \_ngcontent-c0="" class="**nFont nBorder** ng-valid ng-dirty ng-touched">

<input [(ngModel)]="name" [class]="'nFont nBorder nBackground'" class="nColor nBold" />

输出结果：<input \_ngcontent-c0="" class="**nFont nBorder nBackground** ng-pristine ng-valid ng-touched">

如果不使用变量，则必须使用 **‘ ’**

<input [(ngModel)]="name" [class]="'nFont nBorder nBackground'" class="{{hisClass}}" /><br>

public hisClass = "nColor nBold";

[class]= 与 class=”{{ }}” 是相同等级的

所以取后者

<input \_ngcontent-c0="" class="nColor nBold ng-untouched ng-pristine ng-valid">

* [class.xxxx] = “true|false”

<input [(ngModel)]="name" [class.nFont]="true" [class.nBorder]="true" class="nColor nBold" />

合并：

<input \_ngcontent-c0="" class="nColor nBold nFont nBorder">

class=”{{ }}” 覆盖前面[class.xxx] 合并后面的 [class.xxx]：

<input [(ngModel)]="name" [class.nFont]="true" [class.nBorder]="true" class="{{hisClass}}" />

覆盖前面：

<input \_ngcontent-c0="" class="nColor nBold">

覆盖前面[class.xxx] 合并后面的 [class.xxx]：

<input [(ngModel)]="name" [class.nBorder]="true" class="{{hisClass}}" [class.nFont]="true" />

<input \_ngcontent-c0="" class="nColor nBold nFont">

合并后面所有的[class.xxx]

<input [(ngModel)]="name" class="{{hisClass}}" [class.nFont]="true" [class.nBorder]="true" />

<input \_ngcontent-c0="" class="nColor nBold nFont nBorder">

* [class.xxxx] , [class] 与 class=”xxx”

<input [(ngModel)]="name" [class.nFont]="true" class="nColor nBold" [class.nBorder]="true" />

Class=”” 与只有 [class.xxx] 则是合并所有：

<input \_ngcontent-c0="" class="nColor nBold nFont nBorder">

[class] 彻底覆盖 class=”xxx”, 然后合并 [class.xxx]

public hisClass = "nColor nBold";

<input [(ngModel)]="name" [class]="'nFont'" class="{{hisClass}}" [class.nBorder]="true" />

<input [(ngModel)]="name" [class]="'nFont'" class="nColor nBold" [class.nBorder]="true" />

<input \_ngcontent-c0="" class="nFont nBorder">

* [class], [class.xxx], class=”{{ }}” class=”xxx”

<input [(ngModel)]="name"

class="{{'nColor nBold'}}"

[class]="'nFont'"

[class.nBorder]="true"

class="nBackground"

/>

合并原则： [class] 与 class=”{{ }}” 同级，后者覆盖前者， 然后覆盖掉 class=”xxx” ， 再合并[class.xxx]

<input \_ngcontent-c0="" class="**nFont nBorder**">

<input [(ngModel)]="name"

[class]="'nFont'"

class="{{'nColor nBold'}}"

[class.nBorder]="true"

class="nBackground"

/>

合并原则： [class] 与 class=”{{ }}” 同级，后者覆盖前者， 然后覆盖掉 class=”xxx” ， 再合并[class.xxx]

<input \_ngcontent-c0="" class="nColor nBold nBorder">

总结： [class] , [class.xxx] ，class=”{{ }}” class=”xxx”

合并原则： [class] 与 class=”{{ }}” 同级，后者覆盖前者， 然后覆盖掉 class=”xxx” ， 再合并[class.xxx]

<input [(ngModel)]="name"

[class.nBorder]="true"

[class.nBold]="true"

class="nFont"

class="nText" - 后面覆盖前面

/>

<input \_ngcontent-c0="" class="**nText nBorder nBold**">

<input [(ngModel)]="name"

[class]="'nColor'"

class="{{'nFont'}}" - 后面覆盖前面

[class.nBorder]="true"

[class.nBorder]="false" – 后面覆盖前面

class="nFont"

class="nBackground" - 被 [class] 或者 class=”{{xxx}}” 覆盖掉

/>

<input \_ngcontent-c0="" class="nFont">

<input [(ngModel)]="name"

[class]="'nColor'"

class="{{'nFont'}}" – 后面覆盖前面

[class.nBorder]="true"

[class.nText]="true" - 合并

class="nFont"

class="nBackground" – 被覆盖

/>

<input \_ngcontent-c0="" class="**nFont nBorder nText**">

总结：[ngClass], [class] , [class.xxx] ，class=”{{ }}” class=”xxx”

<input [(ngModel)]="name"

class="{{'nColor'}}"

[class]="'nFont'"

[class.nBorder]="true"

[ngClass]="{nBold:true}"

class="nBackground"

/>

<input \_ngcontent-c0="" class="**nBackground nColor nBold nBorder**">

合并原则： 只有 [class] 与 class=”{{ }}” 是相互排斥的，最前面的覆盖所有后面的，

<input [(ngModel)]="name"

[class]="'nColor'" – 前面的覆盖后面的

class="{{'nFont'}}"

[class.nBorder]="true"

//[class.nBorder]="false" - 如果有重复，则后面的覆盖前面的

[ngClass]="{nBold:true, nText:true}" - 前面的覆盖后面的

[ngClass]="{nBackground:true, nText:false}"

class="nFont"

class="nText" – 后面的覆盖前面的

/>

合并原则： 只有 [class] 与 class=”{{ }}” 是相互排斥的，最前面的覆盖所有后面的

[ngClass] 有多个， 则是前面覆盖后面所有的，

class=”xxx” 有多个则， 后面覆盖前面的

<input \_ngcontent-c0="" class="**nText nColor nBold nBorder**">

1）

<span [class.text-big]="true" [class.text-red]="true" [class]="tline">Text 222</span>

[class.text-big] [class.text-red] 都不起作用

[class]="tline" - 只有这个起作用

2）

<span class="text-big" [class]="tline">Text 1111</span><br>

[class]="tline" 优先级高，会覆盖掉 class="text-big"

3）

.text-line.blue {

text-decoration: line-through;

color: blue;

font-size:24px;

}

<span [class.text-line]="true" [class.blue]="true">Text 1111</span><br>

.text-line && .blue 同时都有

~~Text 1111~~

<span class="text-line" [class.text-big]="true" [class.text-red]="true">Text 1111</span>

<span [class.text-big]="true" [class.text-red]="true" class="text-line">Text 1111</span><br>

<span [class]="tline" [class.text-big]="true" [class.text-red]="true">Text 222</span>

都起作用，不覆盖

* Style

<input [(ngModel)]="name"

style="color:orange;"

style="font-weight:bold;" - 后面覆盖前面

[style.fontSize]="'36px'"

[style.background-color]="'grey'" - 合并

[ngStyle]="{border:'5px solid green'}" – 前面覆盖后面

[ngStyle]="{fontSize:'30px'}"

/>

合并原则： 不像 class, style 没有 [style]

[style.fontSize]="'36px'" - 优先级比 [ngStyle] 的设置要高

[ngStyle]="{fontSize:'30px'}"

* \*ngIf

<span \*ngIf="isShow; else elseBlock">

<span>Sale Hello World</span>

</span>

<ng-template #elseBlock><span>Else Case</span></ng-template>

public isShow:boolean = true;

<span \*ngIf="isShow; then trueblock; else elseBlock">

<span>Sale Hello World</span>

</span> -- span 元素完全不会出现在页面上

<ng-template #trueblock><span>Show Case</span></ng-template>

<ng-template #elseBlock><span>Else Case</span></ng-template>

* [ngSwitch] - \*ngSwitchCase \*ngSwitchDefault

<div [ngSwitch]="color">

<span \*ngSwitchCase="'yellow'">You Pick {{color}}</span>

<span \*ngSwitchCase="'red'">You Pick {{color}}</span>

<span \*ngSwitchCase="'blue'">You Pick {{color}}</span>

<span \*ngSwitchDefault>You pick default color</span>

</div>

public color:string = "yellow";

* \*ngFor

let item of items

let item of items; index as idx

选项有：index, first, last, odd, even

<ul>

<li \*ngFor="let person of Persons;

index as idx;

first as f;

last as l;

odd as o;

even as e">

{{idx}} - {{f}} - {{l}} - {{o}} - {{e}} - {{person.name}} - {{person.age}}

</li>

</ul>

public Persons: iPerson[] = [

{name:"Green", age: 15},

{name:"Purple", age: 12},

{name:"Yellow", age: 23} ];

输出如下：

* 0 - true - false - false - true - Green - 15
* 1 - false - false - true - false - Purple - 12
* 2 - false - true - false - true - Yellow - 23
* @Input() public var\_xxx
* @Input(“var\_xxx”) public var\_yyy

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

app.component.ts

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title = 'AppComponent Title';

mydata = {name: "Christine Bai", age:21};

}

app.component.html

<div style="text-align:center">

<h1>

Welcome to {{ title }}!

</h1>

<br>

ParentData: {{mydata|json}}

<br>

<br>

<app-sale [mydata]="mydata">Hello World</app-sale>

</div>

sale.component.ts

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

import \* as $ from "jquery";

@Component({

selector: 'app-sale',

template:

`

Name: <input [(ngModel)]="herdata.name" />

Age: <input [(ngModel)]="herdata.age" />

<br>

Sale Mydata: {{herdata|json}}

`,

styles: [

]

})

export class SaleComponent implements OnInit {

@Input("mydata") public herdata;

constructor() {

console.log(`consturctor()`);

$("body").append("<h1>This is good morning</h1>");

}

ngOnInit() {

console.log("ng init");

}

}

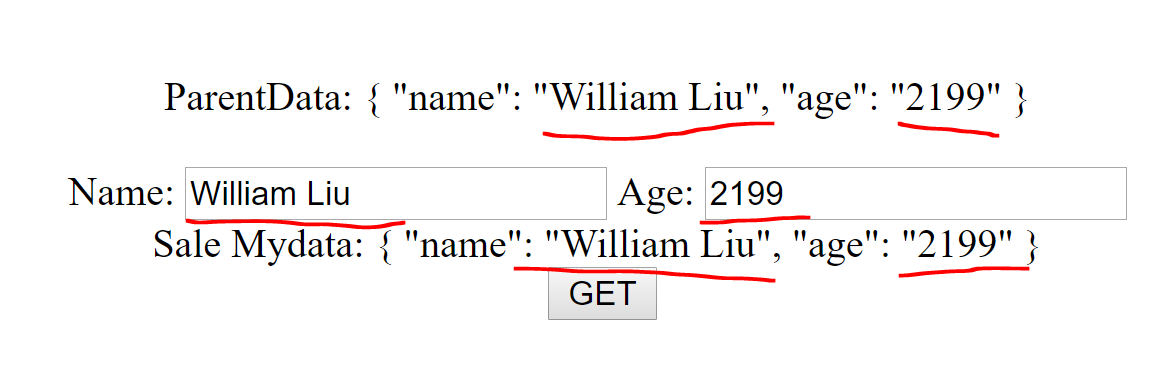
mydata = {name: "Christine Bai", age:21};

<app-sale [mydata]="mydata">Hello World</app-sale>

@Input("mydata") public herdata;

如果传递的是 简单类型： 字符串，数值， 日期，布尔。 则是传递值。 独立隔离的不同变量

如果是传递对象， 如数组， 对象，{ } ， 则是传址， 也就是说子组件的值改变， 父组件的值也改变。



export class SaleComponent implements OnInit {

@Input("mydata") public herdata;

public eid:string="myid";

public hello:string = "Yepp";

constructor() {

console.log(`consturctor()`);

$("body").append("<h1>This is good morning</h1>");

}

ngOnInit() {

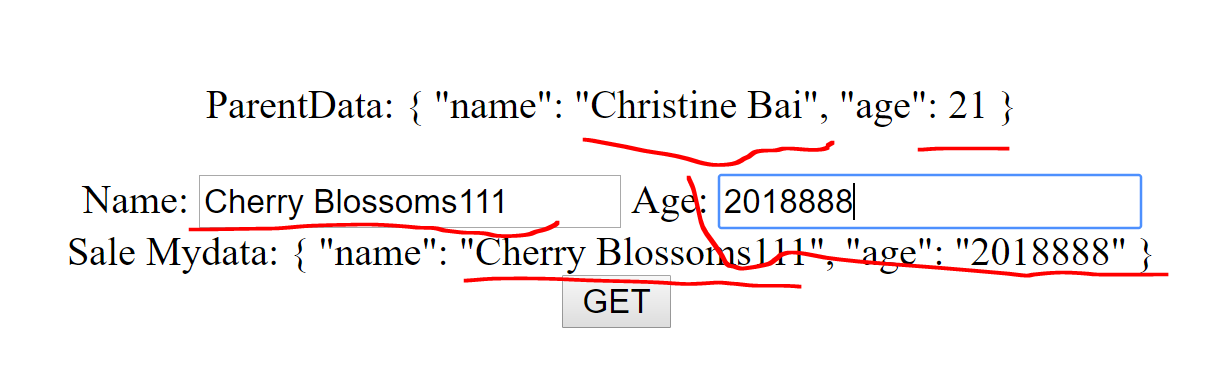
this.herdata = {name:"Cherry Blossoms", age:2018};

console.log("ng init");

}

}

如果我们在 ngOnInit 里重新赋予新的对象。 则和父组件没有任何关系了



如果在 constructor 里操作传入的对象：

export class SaleComponent implements OnInit {

@Input("mydata") public herdata;

public eid:string="myid";

public hello:string = "Yepp";

constructor() {

this.hello = "Hello World"; - 这个是允许设置的

// 实际上是不允许的， 即使如此重新设置，也等于没有执行任何

this.herdata = {name:"", age:666} || this.herdata;

this.herdata.name = "goodDay";

---------------------------------------

console.log(`consturctor()`);

$("body").append("<h1>This is good morning</h1>");

}

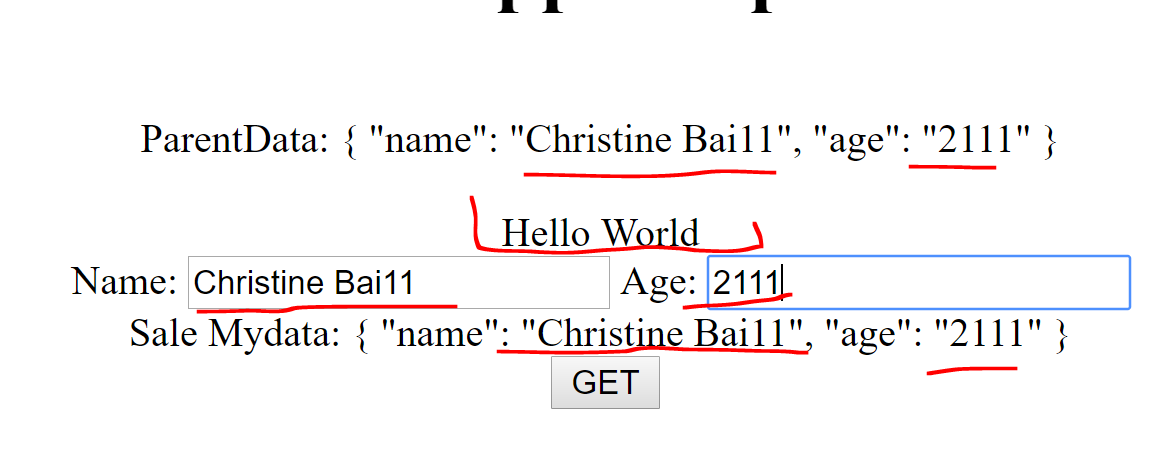
ngOnInit() {

//this.herdata = {name:"Cherry Blossoms", age:2018};

console.log("ng init");

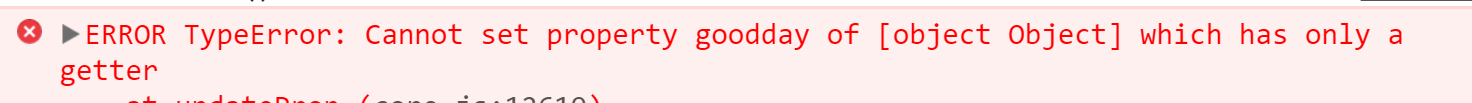
}

}



@Input 和 setter , getter 联合使用

必须要定义 setter , 否则无法赋值。而且没有setter 会出现错误：



getter 是可选的, 如果没有getter 则无法访问其值。

export class AppComponent {

friday = "Good Friday";

}

<app-sale [goodday]="friday"></app-sale>

<app-sale [goodday]="'yes'"></app-sale>

private \_gd:string;

set gday(d:string) {

this.\_gd = d;

}

@Input("goodday") -- 无所谓放在 setter 或者 getter 上都可以，只允许有一个@Input

get gday() {

return this.\_gd;

}

@Input 也可以用来传递函数，注意只是将函数定义传递，函数的context 取决于使用函数时的环境

export class AppComponent {

inputCall(data) {

console.log("input function");

console.log(data);

}

}

<app-sale [mycall]="inputCall"></app-sale>

@Input() public mycall;

<button (click)="sendToParent()">Send</button>

sendToParent() {

this.mycall(this.persons);

this.mycall.call(this, this.persons); -- 或者如此呼叫

}

@Input() – 从父Component 单向改变 子Component的值， 也就是说，父值一改，子值立马跟着变。 如果是子值改变，父值不变，但是子值是改变了。 如果父值接着改变，那么子值不管之前改变与否，都会和父值保持一致。

这就是单向改变。

如果父子都是使用 Object 的属性，因为对象和数组都是通过传址的，所以操作的都是同一个对象。

@Output()

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

@Output 只能借助 EventEmitter 事件来向外传递数据

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

import \* as $ from "jquery";

@Component({

selector: 'app-sale',

template:

`

<br><br><br>

<button (click)="sendToParent()">Send</button>

`,

styles: []

})

export class SaleComponent implements OnInit {

public persons:Array<iPerson> = [

{name:"Red", age: 12},

{name:"Yellow", age: 25}

];

@Output("callMe") public callParent = new EventEmitter();

sendToParent() {

this.callParent.emit(this.persons);

}

constructor() {}

ngOnInit() { }

}

interface iPerson {

name: string;

age: number;

}

<app-sale **(callMe)**="respCall($event);answer($event);">Hello World</app-sale>

answer(data) {

console.log("this is answer");

console.log(data);

}

inputCall(data) {

console.log("input function");

console.log(data);

}

注意： **（callMe）**就是 @Output(“callMe”) 传出来的事件， $event 就是 emit(data) 出来的数据。

Service

@Injectable()

定义服务

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

@Injectable()

export class DataService {

public salary:number = 1000;

public persons:IPerson[] =

[

{name: "Red", age:12, title:"CEO", birth: new Date(2000, 2, 15) },

{name: "Yellow", age: 35, title:"Manager", birth: new Date(1900, 2, 22)}

];

constructor() {

console.log("Data Service is starting");

}

}

export interface IPerson {

name: string;

age: number;

title: string;

birth: Date

}

在模块里引用服务

------------------------------------------------------------

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

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

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

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

import { SaleComponent } from './sale/sale.component';

import { OrderComponent } from './order/order/order.component';

import {DataService} from "./services/data.service";

@NgModule({

declarations: [

AppComponent,

SaleComponent,

OrderComponent

],

imports: [

BrowserModule,

FormsModule

],

providers: [DataService],

bootstrap: [AppComponent]

})

export class AppModule {}

在组件里引用服务

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

import \* as $ from "jquery";

import {DataService, IPerson} from "../services/data.service";

@Component({

selector: 'app-sale',

template:

`

<br><br>

Sale Salary: <input [(ngModel)]="sdataSer.salary" />

<br>

<ul style="text-align:left;">

<li \*ngFor="let person of persons; index as i">

{{i+1}}. {{person.name}} - {{person.age}} - {{person.title}}

</li>

</ul>

`,

styles: []

})

export class SaleComponent implements OnInit {

public persons: IPerson[];

constructor(public sdataSer:DataService) {}

ngOnInit() {

this.persons = this.sdataSer.persons;

}

}

注意：

1）如果使用 constructor 来注入 service，则service 是以singleton 的方式初始化一个对象共享使用。

2）如果使用 new 来实例化service , 则是单独创建一个全新的对象，独立使用数据不会被共享

export class SaleComponent implements OnInit {

public persons: IPerson[];

public sdataSer:DataService = new DataService(); -使用new 则会创建全新隔离的服务

constructor() {}

ngOnInit() {

this.persons = this.sdataSer.persons;

}

}

Http, Observable and RxJS

import { HttpClientModule } from "@angular/common/http";

@NgModule({

declarations: [

AppComponent,

SaleComponent,

OrderComponent

],

imports: [

BrowserModule,

FormsModule,

**HttpClientModule**

],

providers: [DataService],

bootstrap: [AppComponent]

})

export class AppModule {}

Data.Service.ts

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

import { HttpClient } from "@angular/common/http"; -- 导入模块

import { Observable } from "rxjs/Observable"; -- 导入模块

@Injectable()

export class DataService {

constructor(private http: HttpClient) {

console.log("Data Service is starting");

}

ngOnInit() { -- 对于@Injectable ngOnInit 永远不会被执行

console.log("Data Service Init");

}

getData(): Observable<IPerson[]>{

return this.http.get<IPerson[]>("/assets/persons.json");

}

}

export interface IPerson {

name: string;

age: number;

title: string;

birth: Date

}

如何调用

export class SaleComponent implements OnInit {

public persons: IPerson[];

constructor(public sdataSer: DataService) {}

ngOnInit() {

this.sdataSer.getData().subscribe(data=>this.persons=data);

}

}

Error Handler: catch

Routing and Navigation

定义Routing Module:

使用 Angular-CLI:

ng g m yepp –routing --flat

--routing 创建 routing module

-- flat 不创建目录，在当前命令执行的目录创建

我们将看到创建了两个模块文件：

yepp.module.ts -- 如果我们已经有其他主模块，此文件可删除，

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

import { CommonModule } from '@angular/common';

import { YeppRoutingModule } from './yepp-routing.module';

@NgModule({

imports: [

CommonModule,

YeppRoutingModule

],

declarations: []

})

export class YeppModule { }

-----------------------------------------------------------------------------------------------

yepp-routing.module.ts

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

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

const routes: Routes = [];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class YeppRoutingModule { }

为了在路由的组件里，能够正常是用 Angular Core 提供的directive ，需要对路由模块新增

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

import { **CommonModule** } from '@angular/common';

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

const routes: Routes = [];

@NgModule({

imports: [**CommonModule**, RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class YeppRoutingModule { }

创建两个路由组件：

使用 Angular-CLI:

ng g c Country --flat --it --is --spec --prefix world

country.component.ts

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

@Component({

selector: 'world-country',

template: `

<p>

country works!

</p>

`,

styles: []

})

export class CountryComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

ng g c Department --flat --it --is --spec --prefix Company

department.component.ts

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

@Component({

selector: 'Company-department',

template: `

<p>

department works!

</p>

`,

styles: []

})

export class DepartmentComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

定义路由表并注册：yepp-routing.module.ts

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

import { CommonModule } from '@angular/common';

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

import { CountryComponent } from "../bh/country.component"; --导入组件

import { DepartmentComponent } from "../bh/department.component"; --导入组件

const routes: Routes = [

{path: "country", component: CountryComponent}, --定义路由

{path: "depart", component: DepartmentComponent} --定义路由

];

@NgModule({

--在此声明组件以后，此模块被引用，可以不再需要导入声明

declarations:[CountryComponent, DepartmentComponent],

imports: [CommonModule, RouterModule.forRoot(routes)], --forRoot, forChild

exports: [RouterModule] -- 导出RouterModule 很重要

})

export class YeppRoutingModule { }

应用程序主模块：app.module.ts

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

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

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

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

import { YeppRoutingModule } from "./emp/yepp-routing.module";

@NgModule({

declarations: [ -- 这里不需要再定义路由的组件

AppComponent

],

imports: [

BrowserModule,

FormsModule,

YeppRoutingModule -- 只需要导入路由模块即可，也不需要导入 RouterModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule {}

HTML

App.component.html

<a routerLink="/country" routerLinkActive="ok">Click to Country</a>

<a routerLink="/depart" routerLinkActive="ok">Click to Department</a>

<br>

<router-outlet></router-outlet> -- Angular Router View

routerLinkActive - link active css class name 如下：

a.ok {

font-size: 24px;

color: orange;

}

Routing WildCard：

const routes: Routes = [

{path: "country", component: CountryComponent},

{path: "depart", component: DepartmentComponent},

- 这个是默认匹配 pathMatch:"prefix", 将匹配所有的，所以必须使用 full

{path: "", redirectTo:"/country", pathMatch: "full"},

{path: "\*\*", component: BhComponent} --匹配上面无法匹配到的，也就是说Page not found 可放在这里

];

Routing Parameters

Routing Parameters: using url/:parameter\_name

const routes: Routes = [

{path: "country", component: CountryComponent},

{path: "depart", component: DepartmentComponent},

{path: "country/:id/:name", component: DetailComponent},

{path: "", redirectTo:"/country", pathMatch: "full"},

{path: "\*\*", component: BhComponent}

];

public countries = [

{name:"China", cid: 100},

{name:"Canada", cid: 144},

{name:"USA", cid: 134},

{name:"UK",cid:173}

];

<ul>

<li \*ngFor="let ccc of countries">

<button (click)="**selectCountry**(ccc)"

[class.ok]="selected(ccc.cid)">

{{ccc.cid}} - {{ccc.name}}

</button>

<a routerLink="/country/{{ccc.cid}}/{{ccc.name}}"

routerLinkActive="ok">

{{ccc.cid}} - {{ccc.name}}

</a>

</li>

</ul>

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

public selectedID:number = 0;

**selectCountry**(theCountry) {

**this.router.navigate**(["/country", theCountry.cid, theCountry.name]);

this.selectedID = theCountry.cid;

}

selected(cid) {

return this.selectedID==cid;

}

constructor(public pdataSer: DataService, public router:Router, public aroute: ActivatedRoute) {

}

ngOnInit() {

this.selectedID = parseInt(this.aroute.snapshot.paramMap.get("id")); --初始化无法获取

console.log(this.selectedID);

}

以上有个缺点： detail 里获取的 路由参数不会动态改变，一直是初始化时的值



Drawbacks of snapshot.

使用 paramMap Observable

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

import { Router, ActivatedRoute, ParamMap } from '@angular/router';

@Component({

selector: 'app-detail',

template: `

<p>

detail works!

<br>

ID: {{id}} Name: {{name}}

</p>

`,

styles:[]

})

export class DetailComponent implements OnInit {

public id:number;

public name:string;

constructor(public router:Router, public aroute:ActivatedRoute) { }

ngOnInit() {

this.aroute.paramMap.subscribe(

(param: ParamMap) => {

let id = parseInt(param.get("id"));

this.id = id;

let name = param.get("name");

this.name = name;

}

);

}

}

使用 paramMap Optional

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

import { Router, ActivatedRoute, ParamMap } from '@angular/router';

@Component({

selector: 'app-detail',

template: `

<p>

detail works!

<br>

ID: {{id}} Name: {{name}} Good

<br>

<button (click)="back()">Back</button>

</p>

`,

styles:[]

})

export class DetailComponent implements OnInit {

public id:number;

public name:string;

constructor(public router:Router, public aroute:ActivatedRoute) { }

ngOnInit() {

this.aroute.paramMap.subscribe(

(param: ParamMap) => {

let id = parseInt(param.get("id"));

this.id = id;

let name = param.get("name");

this.name = name;

}

);

}

back() {

this.router.navigate(["/depart", {id: this.id}]);

}

}

注意区别：

Router.navigate([ “routing\_path”, param1, param2 ]);

-- 这是路由路径 /routing\_path/param1/param2

Router.navigate([ “routing\_path”, { param1 : value1 , param2: value 2} ]);

-- 这是Optional Parameter, 不参与路由匹配

http://localhost:4200/depart**;id=134 返回的格式是 routing\_path;param1=value1;param2=value2**

**如何提取可选参数**，和普通参数一样

ngOnInit() {

this.aroute.paramMap.subscribe(

(param: ParamMap)=>{

this.param1 = parseInt(param.get("param1"));

this.param2 = parseInt(param.get("param2"));

}

);

}

Relative Route:

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

import { Router, ActivatedRoute, ParamMap } from '@angular/router';

@Component({

selector: 'app-detail',

template: `

<p>

detail works!

<br>

ID: {{id}} Name: {{name}} Good

<br>

<button (click)="back()">Back</button>

</p>

`,

styles:[]

})

export class DetailComponent implements OnInit {

public id:number;

public name:string;

constructor(public router:Router, public aroute:ActivatedRoute) { }

ngOnInit() {

this.aroute.paramMap.subscribe(

(param: ParamMap) => {

let id = parseInt(param.get("id"));

this.id = id;

let name = param.get("name");

this.name = name;

}

);

}

back() {

this.router.navigate(

["../../", {id: this.id, name: this.name}],

{relativeTo: this.aroute});

}

}

<http://localhost:4200/country/144/Canada>

after click on Back Button

<http://localhost:4200/country;id=144;name=Canada>

Child Route:

const routes: Routes = [

{ path: "country",

component: CountryComponent,

**children**: [

{path: "overview", component: OverviewComponent},

{path: "contact", component: ContactComponent}

]

},

{ path: "depart",

component: DepartmentComponent,

**children**: [

{path: "overview", component: OverviewComponent},

{path: "contact", component: ContactComponent}

]

},

{ path: "country/:id/:name",

component: DetailComponent,

**children**: [

{path: "overview", component: OverviewComponent},

{path: "contact", component: ContactComponent}

]

},

{path: "", redirectTo:"/country", pathMatch: "full"},

{path: "\*\*", component: BhComponent}

];

app.component.html

<br>

<a routerLink="/country" routerLinkActive="ok">Click to Country</a>

<a routerLink="/depart" routerLinkActive="ok">Click to Department</a>

<br>

country.component.ts

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

import { Router, ActivatedRoute } from "@angular/router";

@Component({

selector: 'world-country',

template: `

<p>

country works!

<button (click)="goOverview()">Go Overview</button>

<button (click)="goContact()">Go Contact</button>

<router-outlet></router-outlet>

</p>

`,

styles: []

})

export class CountryComponent implements OnInit {

constructor(public router: Router, public route: ActivatedRoute) { }

ngOnInit() { }

goOverview() {

this.router.**navigate**(["overview"], {relativeTo: this.route});

}

goContact() {

this.router.**navigate**(["contact"], {relativeTo: this.route});

}

}

如何构建通用的子路由模块：

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

import { CommonModule } from '@angular/common';

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

import { CountryComponent } from "../bh/country.component";

import { DepartmentComponent } from "../bh/department.component";

import { BhComponent } from "../bh/bh.component";

import { DetailComponent } from "../detail/detail.component";

const routes: Routes = [

{ path: "country", component: CountryComponent,

**loadChildren**: "../child-route/child-route-routing.module#ChildRouteRoutingModule"

},

{ path: "lwh",

**loadChildren**: "../child-route/child-route-routing.module#ChildRouteRoutingModule"

},

{ path: "depart", component: DepartmentComponent },

{

path: "country/:id/:name",

component: DetailComponent,

**loadChildren**: "../child-route/child-route-routing.module#ChildRouteRoutingModule"

},

{ path: "", redirectTo:"overview", pathMatch: "full" },

{ path: "\*\*", component: BhComponent }

];

@NgModule({

declarations:[CountryComponent, DepartmentComponent, DetailComponent, BhComponent],

imports: [CommonModule, RouterModule.**forRoot**(routes)],

exports: [RouterModule]

})

export class YeppRoutingModule { }

loadChildren: lazy loading child route module

child route module

child-route-routing.module.ts

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

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

import { ContactComponent } from "../contact/contact.component";

import { OverviewComponent } from "../overview/overview.component";

import { ComComponent } from "../com/com.component";

const childRoutes: Routes = [

{

path:"",

component: **ComComponent**, -- 匹配所有的URL, 也就是ComComponent是共享模板

children: [

{ path: "overview", component: OverviewComponent },

{ path: "contact", component: ContactComponent }

]

}

];

@NgModule({

declarations:[OverviewComponent, ContactComponent, ComComponent],

imports: [RouterModule.**forChild**(childRoutes)], -- 必须设置为forChild

exports: [RouterModule]

})

export class ChildRouteRoutingModule { }

com.module.ts

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

@Component({

selector: 'app-com',

template: `

<p>

common child route template works!<br>

<router-outlet></router-outlet> -- 承接子路由的容器

</p>

`,

styles: []

})

export class **ComComponent** implements OnInit {

constructor() { }

ngOnInit() {}

}

country.module.ts

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

import { Router, ActivatedRoute } from "@angular/router";

@Component({

selector: 'world-country',

template: `

<p>

country works!

<button (click)="goOverview()">Go Overview</button>

<button (click)="goContact()">Go Contact</button>

<router-outlet></router-outlet>

</p>

`,

styles: []

})

export class CountryComponent implements OnInit {

constructor(public router: Router, public route: ActivatedRoute) { }

ngOnInit() { }

goOverview() {

this.router.navigate(["overview"], {relativeTo: this.route});

}

goContact() {

this.router.navigate(["contact"], {relativeTo: this.route});

}

}

<http://localhost:4200/country/overview> <http://localhost:4200/country/contact>

我们也可以去除共享模板：

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

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

import { ContactComponent } from "../contact/contact.component";

import { OverviewComponent } from "../overview/overview.component";

我们不再是用children, 路由路径直接放在Route[]

const childRoutes: Routes = [

{ path: "overview", component: OverviewComponent },

{ path: "contact", component: ContactComponent }

];

@NgModule({

declarations:[OverviewComponent, ContactComponent],

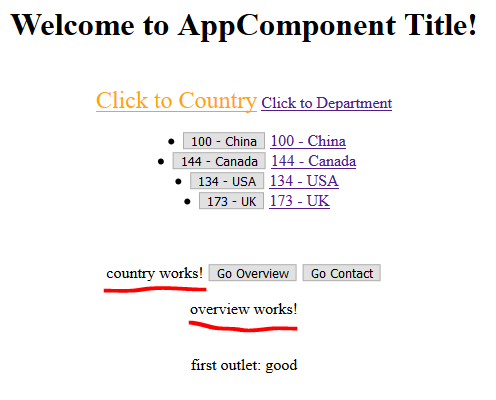
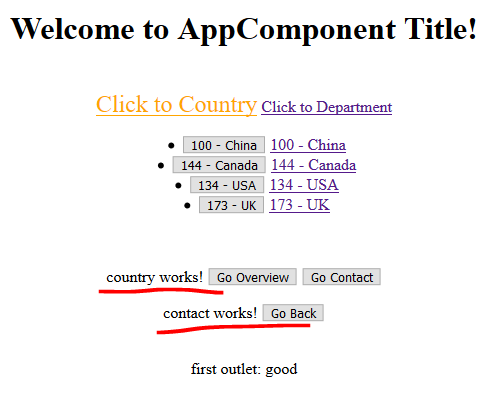
imports: [RouterModule.forChild(childRoutes)],

exports: [RouterModule]

})

export class ChildRouteRoutingModule { }

<http://localhost:4200/country/overview> <http://localhost:4200/country/contact>

* Pipe

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

@Pipe({

name: 'mypip'

})

export class MypipPipe implements PipeTransform {

transform(value: any, arg1: any, arg2:any, arg3:any): any {

return `return value :${ value } arg1: ${arg1} arg2: ${arg2} arg3:${arg3}`;

}

}

title = 'AppComponent Title';

{{title | mypip: 10 : '3kdkd' : 200}}

输出结果：

return value :AppComponent Title arg1: 10 arg2: 3kdkd arg3:200

* Directive

Angular 2+ 的指令分为以下三种：

* 组件(Component directive)：用于构建UI组件，继承于 Directive 类
* 属性指令(Attribute directive): 用于改变组件的外观或行为
* 结构指令(Structural directive): 用于动态添加或删除DOM元素来改变DOM布局

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

@Directive({

selector: '[bgc]'

})

export class BgDirective {

constructor(public elRef: ElementRef, public render: Renderer) {

}

ngOnInit() {

this.render.setText( this.elRef.nativeElement, 'What is good news!' );

this.elRef.nativeElement.style.fontSize = "78px";

}

}

* Module 与 Component 的关系

一、模块

虽然Angular是一款前端框架，但在使用Typescript作为开发语言的同时，也引入了面向对象的开发方式。熟悉Java开发的同学肯定知道依赖注入的概念，幸运的是，Angular的模块化也引用了这一设计模式。模块是Angular中最大的单元，任何应用都应该有一个根模块，同时也可拥有多个模块。Angular中，通过@NgModule装饰器定义模块，示例如下：

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

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

@NgModule({

imports: [ BrowserModule ],

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

其中：

declarations： 用于声明该模块下的组件、指令或管道，上例中的AppComponent便是作为组件声明的，只有在声明之后才能使用。关于组件、指令和管道的概念，将在接下来的部分单独介绍。

exports： 用于导出该模块下的组件、指令或管道，通过导出，这些组件、指令或管道将能在其他模块中使用。

imports： 用于导入其他模块，当模块引用其他模块之后，便可以通过引入组件的方式来使用被引用模块中的组件、指令和管道。上例中由于导入了BrowserModule模块，因此可在该模块中引入和使用BrowserModule模块中的组件、指令和管道。

providers： 用于向该模块中注入服务，服务在注入该模块后，便可在该模块中使用。上例中便注入了Logger服务，之后无需通过构造函数便可以直接在整个模块中的组件直接使用该服务。关于服务的概念，将在接下来的部分单独介绍。

bootstrap： 根模块的根组件 ，作为根模块的入口，只有根模块才有根组件。

关于模块，通俗的将就是将一个应用分成若干个单元，这些单元负责完成某个类型的所有功能，并且这些单元可以相互引用。例如一个简易博客系统，可以分成根模块、用户模块、文章模块。那么文章模块可以引用用户模块，而用户模块和文章模块的入口要通过根模块，所以根模块就得引用另外两个模块。

二、组件

通过@Component装饰器，便可将类定义成一个组件。组件可作为模块中的单元，可以重复使用。使用组件及其视图的方式有多种，可在页面中静态布置，也可通过路由插槽展示。@Component装饰器配置项如下：

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

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

@Component({

selector: 'app-root',

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

// template: '<h1>Welcome to {{ title }}!</h1>',

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

// styles: 'h1{color:#fff}',

providers: [ HttpClient ]

})

export class AppComponent {

title = 'app';

}

其中：

selector： 这是该组件的CSS选择器，在页面中该选择器的位置，将展示该组件的视图。

templateUrl： HTML模板文件的相对路径，也可通过template直接提供HTML模板。HTML模板将用于提供该组件的视图。

styleUrls： 样式文件的相对路径，也可通过styles直接提供样式。该样式仅对该组件的视图有效。

providers： 服务列表，将向该组件注入列表中的服务，此后便可使用该服务了。上例中的Angular被告知，在该组件的构造函数中需要一个HttpClient实例，通过Angular创建了一个HttpClient实例，并直接注入到组件中使用。

---------------------

作者：Tylili.

来源：CSDN

原文：https://blog.csdn.net/Hydrop/article/details/80850073

版权声明：本文为博主原创文章，转载请附上博文链接！

**1.初步了解模块和组件**

在上两篇文章中讲述了构建工程，这篇文章简单讲述一下Angular2中的模块和组件。

app文件夹目录.png

  app文件夹下有五个文件，其中，app.component.spec.ts应该是和模块测试有关的文件，目前不用管它。剩下的四个文件就是典型的**模块+组件**的文件组成模式。  
  Angular2应用由模块和组件构成，每个模块这样明明name.module.ts,组件则是name.component.ts。  
  这几个文件的关系是从属关系：  
|--app.module.ts(模块)  
 |--app.component.ts(组件)  
  |--app.component.html(HTML模板)  
  |--app.component.css(CSS样式表)  
一个模块可以有多个组件，一个组件也可以有多个样式表,但只有一个HTML模板。

**1.1组件的含义**

Angular2中的组件，就是我们自定义的一段HTML代码，给他取个名字，就可以当作HTML的标签使用了。假如组件的名字是my-app，那么就可以这样使用：

<div>

<my-app><my-app>//可以像原生标签一样添加属性，而且组件能接收的属性更丰富

//标签里面也可以放内容

</div>

组件其实是一个类，它定义了一系列方法和属性，并且把HTML模板封装起来。它的意义是，你可以把它当做一个独立的隔离的盒子,并利用其中的方法和属性进行盒子内部的操作，以及和外界的父组件、兄弟组件交互。

**1.2模块的含义**

模块就是将一系列的组件还有指令、服务整合封装到一起，提供一个完整的功能。模块可以被外部模块引用。

**2.模块详解和根模块**

**2.1 根模块**

每个应用会有一个根模块，按照约定，它的类名叫做AppModule，被放在app.module.ts文件中。应用启动时，就会加载这个模块。  
  每个根模块会有一个根组件，默认就是app.component.ts，名字是app-root。  
  查看项目目录中的index.html，会发现有<app-root>Loading...</app-root>这样的代码，就是在加载这个根模块。

**2.2 模块详解**

下面是app.module.ts的代码。

import { BrowserModule } from '@angular/platform-browser';//每个在浏览器中运行

//的应用的根模块都需要引入BrowserModule

import { NgModule } from '@angular/core';//每个模块都需要引入的核心库中的NgModule

import { FormsModule } from '@angular/forms';//表单模块，在应用中使用表单时引入

import { HttpModule } from '@angular/http';//http模块，当需要进行http远程请求时引入

import { AppComponent } from './app.component';//自己创建的组件

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

FormsModule,

HttpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Angular2用TypeScript组织代码的方式大致就像这个文件一样。

* 在顶部使用import语句，导入模块或者组件需要使用的外部模块。  
  在模块中需要使用自己创建的组件或者其他服务、指令，也需要import。
* import语句之后，使用@NgModule()语句描述本模块的元数据。
  + declarations：声明属于本模块的组件，每个组件必须在且仅在一个模块中声明。
  + imports:引入买本模块中用到的模块，该模块是处于import语句引入的模块中。
  + providers:声明模块中使用的服务的提供者，暂时还没有用到。
  + bootstrap:根模块中的引导组件，应用启动过程中，会创建这个数组中的组件并插入到HTML中，一般只有一个引导组件。
* 最后，使用export语句，创建模块的类并暴露出去。  
  模块、组件只有暴露出去，其他的模块才能引用。

**3.组件详解**

import { Component } from '@angular/core'; //所有组件必须引入

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title = 'app works! 你已经完成了第一步';

}

* import语句导入需要的模块。
* 使用@Component()来描述本组件的元数据。
  + selector:创建的组件的名字，就像HTML标签的名字一样
  + templateUrl: HTML模板，使用相对路径，./表示这个组件文件所在的当前目录，
  + styleUrls:HTML模板使用的样式表，可以有多个。
* 最后，使用export创建组件的类并暴露出去。在类中，可以创建属性和方法。

**4.应用启动过程**

  整个应用的启动流程：加载根模块，加载所需的其他模块，创建引导组件，显示组件内容。

***注意***：我觉得学习Angular2很好的方法之一就是去看官方教程，有中文版的，很方便。看完前面的开发指南部分就足以了解Angular2的大致原理了，当然官方教程有的地方概念可能很模糊，有地方不了解的同学，可以留言，我会解答。

作者：胡不归vac  
链接：https://www.jianshu.com/p/1102d886aebe  
来源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

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

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { MyrouterRoutingModule } from "./myrouter/myrouter-routing.module";

import { HelpModule } from './help/help.module';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MyrouterRoutingModule,

HelpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Module可以包含多 Components

注意：

如果组件在别的模块里定义了，则不能在别的模块里再次导出使用。

app.module.ts

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

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { MyrouterRoutingModule } from "./myrouter/myrouter-routing.module";

import { HelpModule } from './help/help.module';

import { EmployeeComponent as e1} from "./employee/employee.component";

import { LoginComponent as ll } from "./login/login.component";

@NgModule({

declarations: [

AppComponent,

e1,

ll

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MyrouterRoutingModule,

HelpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

app-routings.module.ts

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

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

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

import { EmployeeComponent } from "./employee/employee.component";

import { LoginComponent } from "./login/login.component";

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

import { DataService } from './Services/data.service';

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

const routes: Routes = [

{ path: "employee", component: EmployeeComponent },

{ path: "login", component: LoginComponent }

];

@NgModule({

declarations: [EmployeeComponent, LoginComponent],

imports: [RouterModule.forRoot(routes), HttpClientModule, FormsModule, BrowserModule],

providers: [DataService],

exports: [RouterModule, EmployeeComponent, LoginComponent]

})

export class AppRoutingModule { }

解决方案：

使用exports : [EmployeeComponent, LoginComponent]

给其他模块使用即可。其他模块不需要再使用

Import 和 dclarations

所以其他的模块，只是需要导入模块

imports:[] 即可，被导入的模块会把可导出的组件一起带过来

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

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { MyrouterRoutingModule } from "./myrouter/myrouter-routing.module";

import { HelpModule } from './help/help.module';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

MyrouterRoutingModule,

HelpModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

# angular 路由跳转以及传参

2018年05月17日 20:42:28 [坑里水库](https://me.csdn.net/qq_27409289) 阅读数 3404

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/qq\_27409289/article/details/80356384

1 获取在路由后面的参数

**<a [routerLink]="['/list/nav','MALFUNCTION','WAITING']"></a>**

**这种方式需要在路由配置像这样：**

{

// 报修列表导航快捷栏

**path**: **'list/nav/:repairType/:repairStatus'**,

},

可以通过这种方式获取传递的参数:

**this**.**activatedRoute**.**params**[**'value'**][**'repairType'**];

通过这种方式传递参数域名是这样的：http://localhost:4200/#/system\_manage/user/position/list/repairType/6

缺点： 不能通过

http://localhost:4200/#/system\_manage/user/position/list/直接访问地址，必须的加上参数否则出错

2 通过域名跳转的方式获取参数（http://localhost:4200/#/system\_manage/repair/list?repairType=REPAIR&repairStatus=WAITING）

这种方式不需要再路由后面配置，解放路由，只需跟平常一样写路由如：

{

**path**: **"list"**,

。。。。。。。。。。。。。

},

只需要在ts代码中写地址跳转：

**this**.**router**.navigate([**'/system\_manage/repair/list'**], { **queryParams**: { **'repairType'**: **'CLEAN'**,**'repairStatus'**:**'WAITING'** } });

上面这种也可以传递多个参数

在comoent中需要这么获取‘?’ 后面的参数

this.**activatedRoute**.**queryParams**.subscribe((params:Params)=>{

**this**.**repairType** = params[**'repairType'**];

**this**.**repairStatus** = params[**'repairStatus'**];

});

这两种的方式获取传递参数的区别在于

this.**activatedRoute**.**queryParams**和this**.activatedRoute.params**

再摘抄一些路由跳转的内容：

1.this.router.navigate(['user', 1]);   
以根路由为起点跳转

2.this.router.navigate(['user', 1],{relativeTo: route});   
默认值为根路由，设置后相对当前路由跳转，route是ActivatedRoute的实例，使用需要导入ActivatedRoute

3.this.router.navigate(['user', 1],{ queryParams: { id: 1 } });   
路由中传参数 /user/1?id=1

4.this.router.navigate(['view', 1], { preserveQueryParams: true });   
默认值为false，设为true，保留之前路由中的查询参数/user?id=1 to /view?id=1

5.this.router.navigate(['user', 1],{ fragment: 'top' });   
路由中锚点跳转 /user/1#top

6.this.router.navigate(['/view'], { preserveFragment: true });   
默认值为false，设为true，保留之前路由中的锚点/user/1#top to /view#top

7.this.router.navigate(['/user',1], { skipLocationChange: true });   
默认值为false，设为true路由跳转时浏览器中的url是跳转前的路径，但是传入的参数依然有效

8.this.router.navigate(['/user',1], { replaceUrl: true });   
未设置时默认为true，设置为false路由不会进行跳转

# [angular路由详解三（路由参数传递）](https://www.cnblogs.com/chzlh/p/7718064.html)

我们经常用路由传递参数，路由主要有三种方式：

第一种：在查询参数中传递数据

{path:"address/:id"}   => address/1  => ActivatedRoute.param[id]

　　在路由中传递

　　<a [routerLink] = "['/address',1]"></a>

　　点击事件传递

　　this.router.navigate([‘/address’,2])

　　//在不同等级页面跳转可以用snapshot（快照方式）

　　this.id = this.routeInfo.snapshot.params["id"]

　　//相同组件跳转需要使用subscribe(订阅方式)

　　 this.routeInfo.params.subscribe((params: Params) => this.id = params["id"]  )

第二种：在路由路径中传递参数数据

<a [routerLink] = "['/address']" queryParams= "{id:1}"></a>

this.id = this.routeInfo.snapshot.queryParams["id"]//拿到路由中的参数

第三种：在路由配置中传递数据

{path:'home', component: HomeComponent，data:[{isPush:true}] }  => ActivatedRoute.data[0][isPush]

　　//同样也是有snapshot和subscribe两种类型

　　this.isPush = this.routeInfo.snapshot.data[0]["isPush"]

Router 传递参数：

<ul>

<li \*ngFor="let e of emps; index as i">

Summary:

<a [routerLink]="['/employee/summary', e.Id]" [queryParams]="{sname: wmname, dname:wmname}" >

{{i}} - {{e.Id}} - {{e.Name}} - {{e.Age}}

</a><br>

Depart:

<a [routerLink]="['/employee/depart', e.Id]" [queryParams]="">

{{i}} - {{e.Id}} - {{e.Name}} - {{e.Age}}

</a><br>

</li>

</ul>

constructor(private \_route: Router, private \_aroute: ActivatedRoute) { }

ngOnInit() {

this.\_aroute.params.subscribe( (p:Params) => {

this.eid = p["id"];

});

this.\_aroute.paramMap.subscribe(p=>{

this.hid = p.get("id");

});

this.\_aroute.queryParams.subscribe(p => {

this.sname = p["sname"];

this.dname = p["dname"];

});

}

Params 和 ParamMap 都是差不多。 queryParams 是专门访问 queryParams: ?id=1000&name=someone&age=99

# Angular2父子组件之间数据传递：@ViewChild获取子组件

<https://blog.csdn.net/wuyuxing24/article/details/84927282>

@ViewChild(ChildenComponent) child: ChildenComponent;//子组件实例引用  
@ViewChild("child") child2; //字符串

<app-son toggle="Diag Toggle" #diag></app-son>

<app-son toggle="Tree Toggle" #tree></app-son>

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

import { SonComponent } from '../son/son.component';

@Component({

selector: 'app-employee',

templateUrl: './employee.component.html',

styleUrls: ['./employee.component.css']

})

export class EmployeeComponent implements OnInit, AfterViewInit {

public fromtree: string;

public fromdiag: string;

@ViewChild("tree") tree: SonComponent;

@ViewChild("diag") diag: SonComponent;

ngOnInit() {

}

ngAfterViewInit() {

this.fromtree = this.tree.tot;

this.fromdiag = this.diag.tot;

}

eclick() {

this.tree.sonClick();

this.diag.sonClick();

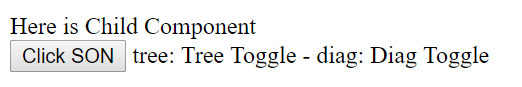
}

}

<button (click)="eclick()">Click SON</button> tree: {{fromtree}} - diag: {{fromdiag}}

<app-son toggle="Diag Toggle" #diag></app-son>

<app-son toggle="Tree Toggle" #tree></app-son>



# Angular2父子组件之间数据传递：@ViewChild获取子组件

2017年07月09日 17:06:46 [小处成就大事](https://me.csdn.net/long328583644) 阅读数 9524

​我们在《Angular2父子组件之间数据传递：局部变量获取子组件》讲到过（如果有不懂的，可以先去看看），通过在子组件模版上设置局部变量的方式获取子组件的成员变量，但是有一个限制，必须在父组件的模版中设置局部变量才能够获取到子组件成员。那有没有办法实现不依赖于局部变量获取子组件成员呢？ 答案：肯定是有的，接下来我们讲下通过@ViewChild来实现！

@ViewChild(className)**当然了这个class也是有条件的,必须是@Component或者@Directive修饰的class。**

@ViewChild(‘diag’)

<app-son toggle="Diag Toggle" #diag></app-son>

淡描@ViewChild

@ViewChild的作用是声明对子组件元素的实例引用，意思是通过注入的方式将子组件注入到@ViewChild容器中，你可以想象成依赖注入的方式注入，只不过@ViewChild不能在构造器constructor中注入，因为@ViewChild会在ngAfterViewInit()回调函数之前执行。

@VIewChild提供了一个参数来选择将要引入的组件元素，可以是一个子组件实例引用， 也可以是一个字符串(两者的区别，后面会讲)

@ViewChild(ChildenComponent) child: ChildenComponent;//子组件实例引用  
@ViewChild("child") child2; //字符串

下面我们来介绍一下两种用法。

1、当传入的是一个子组件实例引用

childenConponetn.ts（子组件）

**import** {Component} **from** '@angular/core';  
  
@Component({  
selector: 'app-childen',  
  templateUrl: './childen.component.html',  
  styleUrls: ['./childen.component.css']  
})  
**export class** ChildenComponent {  
fun1() {  
alert('子组件方法');  
  }  
}

1、定义了一个类方法fun1()，提供给父组件调用

parentComponent.ts（父组件）

**import** {Component, ViewChild} **from** '@angular/core';  
**import** {ChildenComponent} **from** './childen/childen.component'  
  
@Component({  
selector: 'app-parent',  
  templateUrl: './parent.component.html',  
  styleUrls: ['./parent.component.css']  
})  
  
**export class** ParentComponent {  
 @ViewChild(ChildenComponent) child: ChildenComponent;  
  
  OnClick() {  
**this**.child.fun1();  
  }  
}

1、这里传入一个子组件实例引入，定义了一个变量child接收

2、定义了Onclick()方法，用于页面触发点击事件，模拟调用子组件中的方法

parentComponetn.html（父组件模版）

<div class="parent\_div">  
  <p>父组件</p>  
  <div>  
    <input type="button" value="调用子组件方法" (click)="OnClick()">  
  </div>  
  <!---子组件指令 start-->  
  <app-childen></app-childen>  
  <!---子组件指令 end-->  
</div>

1、父组件模版中input绑定了一个click点击事件，页面触发点击调用OnClick()方法

最终效果如下：



2、当传入的是一个字符串

parentComponent.ts（父组件）

**import** {Component, ViewChild} **from** '@angular/core';  
  
@Component({  
selector: 'app-parent',  
  templateUrl: './parent.component.html',  
  styleUrls: ['./parent.component.css']  
})  
  
**export class** ParentComponent {  
  
 @ViewChild('myChild') child;  
  
  OnClick() {  
**this**.child.fun1();  
  }  
}

1、@ViewChild传入一个字符串myChild，变量child接收。其它不变

parentComponent.html（父组件模版）

<div class="parent\_div">  
  <p>父组件</p>  
  <div>  
    <input type="button" value="调用子组件方法" (click)="OnClick()">  
  </div>  
  <!---子组件指令 start-->  
  <app-childen #myChild></app-childen>  
  <!---子组件指令 end-->  
</div>

1、细心的你会发现这里在子组件模版中创建了一个局部变量#myChild，父组件中的myChild字符串解释为一个选择器。也就是一个元素包含模版局部变量#myChild，这就是与第一种方式唯一不同的地方，这里弥补了 《Angular2父子组件之间数据传递：局部变量获取子组件》只能在模版中使用子组件引入的缺陷

最终效果跟上面结果一样

# Angular 正确使用 @ViewChild、@ViewChildren 访问 DOM、组件、指令

2018年12月09日 13:26:00 [tuacy](https://me.csdn.net/wuyuxing24) 阅读数 1282

       @ViewChild和@ViewChildren是Angular提供给我们的装饰器，用于从模板视图中获取匹配的元素。需要注意的是@ViewChild和@ViewChildren会在父组件钩子方法ngAfterViewInit调用之前赋值。接下来我们来看下@ViewChild、@ViewChildren怎么使用。

# 一 选择器

       @ViewChild、@ViewChildren支持的选择器包括以下几种(@ViewChild、@ViewChildren的参数有以下几种)：

## 1.1 class

       当然了这个class也是有条件的，必须是@Component或者@Directive修饰的class。

## 1.2 模板应用变量。

       得解释下什么是模板应用变量。直接用一个例子来说明。  
例如<my-component #cmp>里面的cmp就是模板应用变量。

## 1.3 子组件provider提供的类

       我们还是用实际的例子来说明，比如我们有一个子组件。代码如下

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

import {ChildService} from './child.service';

@Component({

selector: 'app-child',

template: `

<h1>自定义的一个子组件</h1>

`,

providers: [

ChildService

]

})

export class ChildComponent implements OnInit {

constructor(public childService: ChildService) {

}

ngOnInit() {

}

}

       在上面的子组件里面provider里面提供了一个ChildService类。我们也是可以通过@ViewChild来拿到这个ChildService类的。代码如下

@ViewChild(ChildService) childService: ChildService;

## 1.4 子组件provider通过string token提供的类

       我们还是用实际的例子来说明。子组件的pprovider通过 string token valued的形式提供了一个StringTokenValue类，string token 对应的是tokenService。

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

import {StringTokenValue} from './string-token-value';

@Component({

selector: 'app-child',

template: `

<h1>自定义的一个子组件</h1>

`,

styleUrls: ['./child.component.less'],

providers: [

{provide: 'tokenService', useClass: StringTokenValue}

]

})

export class ChildComponent {

}

       在父组件里面我们也是可以拿到子组件provider的这个StringTokenValue类的。方式如下：

@ViewChild('tokenService') tokenService: StringTokenValue;

## 1.5 TemplateRef

       当选择器是TemplateRef的时候，则会获取到html里面所有的ng-template的节点。实际例子如下：

/\*\*\*\* @ViewChild(TemplateRef) @ViewChildren(TemplateRef)获取页面上的ng-template节点信息 \*\*\*\*/

@ViewChild(TemplateRef) template: TemplateRef<any>;

@ViewChildren(TemplateRef) templateList: QueryList<TemplateRef<any>>;

# 二 使用场景

## 2.1 DOM

       如果想通过@ViewChild、@ViewChildren访问到DOM节点的话都得先给相应的DOM节点设置模板应用变量。例如下面的domLabel就是一个模板应用变量。

<div #domLabel>cba</div>

* 1

### 2.1.1 获取DOM里面宿主视图

       咱们可以简单把div，input，label等等，当然了还包括我们自定义的一些组件都认为是宿主视图。用ElementRef变量接收到获取到的宿主视图元素。

ElementRef

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

@Component({

selector: 'app-root',

template: `

<!-- view child dom -->

<div #domLabel>

abc

</div>

`,

styleUrls: ['./app.component.less']

})

export class AppComponent implements AfterViewInit {

/\*\*\*\* DOM节点对应的 \*\*\*\*/

// DOM节点只能使用模板应用变量来找到

@ViewChild('domLabel') domLabelChild: ElementRef; // 找到第一个符合条件的节点

ngAfterViewInit(): void {

// DOM节点

console.log(this.domLabelChild.nativeElement);

}

}

### 2.1.2 获取DOM里面嵌入视图

       咱们可以简单的吧 ng-template的内容认为是嵌入视图。

import {AfterViewInit, Component, QueryList, TemplateRef, ViewChild, ViewChildren} from '@angular/core';

@Component({

selector: 'app-root',

template: `

<ng-template #domTemplate>

<div>默认我是不会显示的</div>

</ng-template>

<ng-template>

<div></div>

</ng-template>

`,

styleUrls: ['./app.component.less']

})

export class AppComponent implements AfterViewInit {

/\*\*\*\* DOM节点对应的 \*\*\*\*/

@ViewChild('domTemplate') domTemplate: TemplateRef<any>; // 查找嵌入元素

/\*\*\*\* @ViewChild(TemplateRef) @ViewChildren(TemplateRef)获取页面上的ng-template节点信息 \*\*\*\*/

@ViewChild(TemplateRef) template: TemplateRef<any>;

@ViewChildren(TemplateRef) templateList: QueryList<TemplateRef<any>>;

ngAfterViewInit(): void {

// DOM节点

console.log(this.domTemplate);

console.log(this.template);

if (this.templateList != null && this.templateList.length !== 0) {

this.templateList.forEach(elementRef => console.log(elementRef));

}

}

}

## 2.2 组件

       大部分情况下第三方库里面的组件或者我们自定义的一些组件在父组件里面都是需要拿到这个子组件对象的。有两种获取获取到组件：一个是通过模板变量名、另一个是通过组件类。

import {AfterViewInit, Component, ElementRef, QueryList, ViewChild, ViewChildren} from '@angular/core';

import {ChildComponent} from './child.component';

@Component({

selector: 'app-root',

template: `

<!-- view child for Component -->

<app-child #appChild></app-child>

<app-child></app-child>

`,

styleUrls: ['./app.component.less']

})

export class AppComponent implements AfterViewInit {

/\*\*\*\* 组件 \*\*\*\*/

@ViewChild('appChild') componentChild: ChildComponent; // 通过模板变量名获取

@ViewChild(ChildComponent) componentChildByType: ChildComponent; // 直接通过组件类型获取

@ViewChild('appChild', {read: ElementRef}) componentChildElement: ElementRef; // 直接找到子组件对应的DOM

@ViewChildren(ChildComponent) componentChildList: QueryList<ChildComponent>; // 获取所有的

ngAfterViewInit(): void {

// DOM节点

console.log(this.componentChild);

if (this.componentChildList != null && this.componentChildList.length !== 0) {

this.componentChildList.forEach(elementRef => console.log(elementRef));

}

}

}

       有写场景我可能需要直接获取到组件对应的DOM节点，这个时候  
@ViewChild、@ViewChildren就的加上过滤条件{read: ElementRef}。如下所示：

@ViewChild('appChild', {read: ElementRef}) componentChildElement: ElementRef; // 直接找到子组件对应的DOM

## 2.3 指令

       @ViewChild、@ViewChildren也是可以获取到指令对象的。指令对象的获取和组件对象的获取差不多，唯一不同的地方就是用模板变量名获取指令的时候要做一些特殊的处理。我们还是用具体的实例来说明。

       我们自定义一个非常简单的指令TestDirective，添加exportAs属性。代码如下。

exportAs属性很关键

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

/\*\*

\* 指令，测试使用,这里使用了exportAs，就是为了方便我们精确的找到指令

\*/

@Directive({

selector: '[appTestDirective]',

exportAs: 'appTest'

})

export class TestDirective {

constructor(private elementRef: ElementRef) {

elementRef.nativeElement.value = '我添加了指令';

}

}

       获取TestDirective指令，注意单个指令对象获取的时候，模板变量名的写法。比如下面的代码中#divTestDirective=‘appTest’，模板变量名等号右边的就是TestDirective指令exportAs对应的名字。

import {AfterViewInit, Component, QueryList, ViewChild, ViewChildren} from '@angular/core';

import {TestDirective} from './test.directive';

@Component({

selector: 'app-root',

template: `

<!-- view child for Directive -->

<input appTestDirective/>

<br/>

<input appTestDirective #divTestDirective='appTest'/>

`,

styleUrls: ['./app.component.less']

})

export class AppComponent implements AfterViewInit {

/\*\*

\* 获取html里面所有的TestDirective

\*/

@ViewChildren(TestDirective) testDirectiveList: QueryList<TestDirective>;

/\*\*

\* 获取模板变量名为divTestDirective的TestDirective的指令,这个得配合指令的exportAs使用

\*/

@ViewChild('divTestDirective') testDirective: TestDirective;

ngAfterViewInit(): void {

console.log(this.testDirective);

if (this.testDirectiveList != null && this.testDirectiveList.length !== 0) {

this.testDirectiveList.forEach(elementRef => console.log(elementRef));

}

}

}

[实例链接地址](https://github.com/tuacy/ng-view-child)

       总结：获取子元素的的时候，我们用的最多的应该就是通过模板变量名，或者直接通过class来获取了。还有一个特别要注意的地方就是获取单个指令对象的时候需要配合指令的exportAs属性使用，并且把他赋值给模板变量名。

使用@ViewChild、@ViewChildren 的完整案例：

// SonComponent

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

@Component({

selector: 'app-son',

templateUrl: './son.component.html',

styleUrls: ['./son.component.css']

})

export class SonComponent implements OnInit {

@Input("toggle") public tot:string;

constructor() { }

ngOnInit() {

}

sonClick() {

alert("Good Son: " + this.tot);

}

}

--------------------------------------------------

import { Component, OnInit, ViewChild, AfterViewInit, ElementRef,TemplateRef, ViewChildren, QueryList } from '@angular/core';

import { SonComponent } from '../son/son.component';

@Component({

selector: 'app-employee',

templateUrl: './employee.component.html',

styleUrls: ['./employee.component.css']

})

export class EmployeeComponent implements OnInit, AfterViewInit {

@ViewChild("tmp") temp: TemplateRef<any>;

@ViewChild("el1") el1: ElementRef;

@ViewChild("diag") diag: SonComponent;

@ViewChildren(SonComponent) sonList: QueryList<SonComponent>;

ngAfterViewInit() {

console.log(this.temp);

console.log(this.el1);

console.log(this.diag);

console.log("diag toggle: " + this.diag.tot);

for(let son of this.sonList) {

console.log(son.tot)

}

}

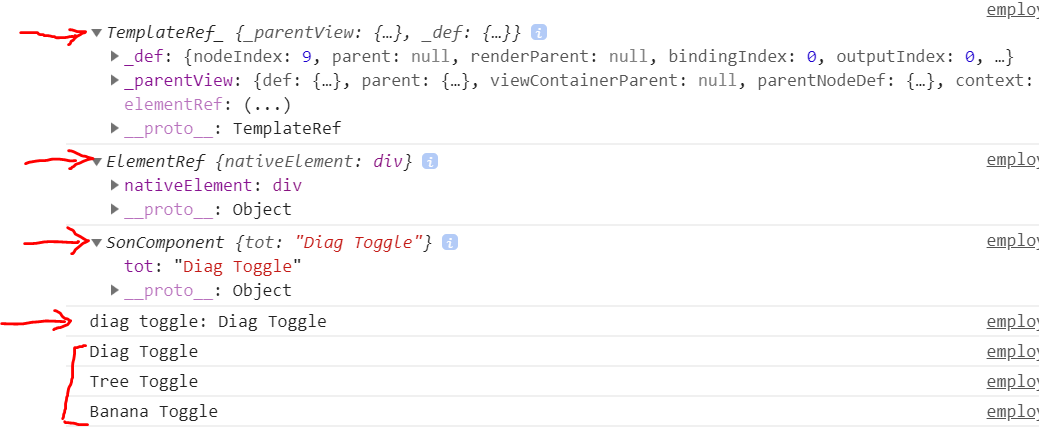
eclick() {

this.diag.sonClick();

}

}

Console.log() output as below



Employee.Component.html

-------------------------------------------------------

c Here is employee Component<br>

<ng-template #tmp>

<span>Hello world</span>

</ng-template>

<div #el1>

Sunny Day!

</div>

<br>

<app-son toggle="Diag Toggle" #diag></app-son>

<app-son toggle="Tree Toggle"></app-son>

<app-son toggle="Banana Toggle"></app-son>

<button (click)="eclick()">Click SON</button>

<br>