* heroes = Hero[]

不能这样写。必须heroes : Heroes[]

* 在新建项目之前设置ng set –global packageManager=cnpm
* 我们是这样写 Angular 应用的：用 Angular 扩展语法编写 HTML模板template， 用组件类Component管理这些模板，用服务service添加应用逻辑，用模块module打包发布组件与服务。然后，我们通过引导根模块来启动该应用
* Angular 应用程序的八个主要构造块：模块module、组件component、模板template、元数据metadata、数据绑定、指令、服务service、依赖注入depedency injection。
* 安装完antD或其他组件后要重新ng serve
* http模块是HttpClientModule，很久以前用的是httpModule
* import .ts文件时不需要加.ts后缀
* <div [hidden]="submitted">
* formsModule如果在子模块中使用，需要再引入一次

怪事儿：listOfDate需要在属性声明的时候就赋值，否则就不能绑定了

formdata和request payload还是没搞明白

文件上传也是一知半解

为啥button没有type=submit，但是一点就触发了form的ngsubmit事件？

监听input的onchange事件，两种办法，一种formControl.registerOnChange(fn)，没用，不知道为啥,好像是只能监听setValue引起的值的变化？；一种formControl.valueChanges()

providers: [{ provide: NZ\_I18N, useValue: zh\_CN }],antdesign的一句话，什么意思

ng new hello-angular

创建项目

慢的一匹，但是运行命令

ng new hello --skip-install

在新建项目时，就会忽略安装依赖包，创建速度就很快。等创建完成后，再把别的项目中 node\_modules 文件夹拷过来，用 npm install执行依赖安装时，就没多少需要下载了

ng serve

或者

npm start

启动开发服务器，浏览器输入localhost:4200

# component

* 增加一个**Component**

ng generate component login

ng g c login

generate 是用来生成文件的

component 说明要生成一个组件

login 组件名称，也可以是loginComp/login这样自动创建一个loginComp文件夹，然后loginComp里边又有一个login文件夹

创建

src/app/login/login.component.ts

src/app/login/login.component.html

src/app/login/login.component.css

ng generate component login --inline-template --inline-style

ng g c login -it -is

--inline-template --inline-style告诉angular-cli：生成组件时，把组件的HTML模板和CSS样式和组件放在.ts中

ng generate component login –module app

现在的版本，如果有两个及以上module，创建组件时需要指定所属的module

hello-angular/src/app/login/login.component.ts

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

@Component({

selector: 'app-login',

template: ` //反引号注意

<p>

login Works!

</p>

`,

styles: []

})

export class LoginComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

* @Component 装饰器Decorator
* selector: 'app-login'，在其他组件的template中使用 <app-login></app-login> 来引
* 如果不是内联形式的template和styles时

templateUrl: './hero-list.component.html',

styleUrls: ['./hero-list.component.css']

* 元数据Metadata：selector, templateUrl, styleUrls

@Component({

providers: [ HeroService ]

})

元数据还可以有providers配置项

装饰器种类

@Component

@Service

@NgModule

* 组件的类名：大驼峰形式 ，并且以 Component 结尾。例如HeroListComponent。
* 组件的文件名：小写中线形式，每个单词使用中线分隔，并且以.component.ts 结尾。

例如hero-list.component.ts

* 当前版本中，用ng g c HeroList 创建组件，angular直接转换为hero-list.component

# 类

ng generate class hero

在src/app/位置下生成一个hero.ts

constructor(public id:number, public name: string){}

这样写想当于声明了两个属性，还有一个构造器。

var3=new Class1(3, 'Zhanglin')

export interface Hero {

id: number;

name: string;

}

写接口也行

export class Hero {

id: number;

name: string;

}

一般我们总是 export 组件类，因为肯定会在别的地方import它。

import { Hero } from '../../domain/hero';

hero: Hero = {

id: 1,

name: 'The Monkey King'

}

Angular官方指导建议一个ts文件中只有一个类

hero-list.component.html

<h1>Hello {{hero.name}}!</h1>

<h2>Your id is: {{hero.id}}</h2>

其他文件引入这个类

import { Hero } from '../../domain/hero'

mock-heroes.ts

import { Hero } from './hero';

export const HEROES: Hero[] = [

{ id: 11, name: 'Dr Nice' },

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

{ id: 13, name: 'Bombasto' }

];

一个ts文件，不仅可以导出类、接口，还可以导出一个对象、数组，看到了吗

这儿是一个怎样定义interface和全局常量的示例

hero.ts

export interface Hero {

name: string;

}

export const HEROES = [

{name: 'Dr IQ'},

{name: 'Magneta'},

{name: 'Bombasto'}

];

# template

* 插值表达式interpolation expression

{{ ... }}

* 模板表达式template expression

{{ ... }}中的 ...

和 [hidden]="ischanged" 中的ischanged

* 表达式上下文 Expression context

这个组件

* 模板变量
* template input variable 模板输入变量

<div \*ngFor="let hero of heroes">{{hero.name}}</div>

hero就是一个模板输入变量

* template reference variable (#heroInput) 模板引用变量

<input #heroInput> {{heroInput.value}}

* 模板语句 Template statements

用来响应事件 (event)="statement"

<button (click)="deleteHero()">Delete hero</button>

语句上下文 statement context 就是这个组件

# 绑定

* 从数据源到视图

{{expression}}

[target]="expression"

bind-target="expression"

上两种等价！

* 从视图到数据源

(click)="statement"

on-click="statement"

* 双向的，从视图到数据源再到视图

[(target)]="expression"

bindon-target="expression"

* attribute和property的区别

attribute 由 HTML 定义。property 由 DOM (Document Object Model) 定义。

* getAttribute, setAttribute是获取、更改attribute的方法
* eleName.propertyName来获取、修改property

attribute的值只能是字符串。在标签里边写的东西都是给attribute赋值

* 对于class、id、name

<button class="hello">nihao</button>

var a=document.getElementsByTagName('button')[0]

a.className

a.getAttribute('class')

a.classList.add('shijie')

a.setAttribute('class', 'temp')

不能a.class获取class属性，回避JavaScript关键字。用a.className查看

class的attribute和property，不管其中的一个怎么改，另一个也跟着改，这俩始终一样。id和name也是如此

* 对于vaue

<input type="text" value="Bob">

value 这个 property 被初始化为 “Bob”。

当在输入框中输入 “Sally” 时，DOM 元素的 value 这个 property 变成了 “Sally”。 但是value 这个 attribute 保持不变。

var a=document.getElementsByTagName('input')[0]

a.value='Alice' //value property被改变，输入框内容变为Alice

a.getAttribute('value') //返回Bob

* 对于disabled、hidden

一个button，默认情况下disabled属性值为false，可以点击。

<button disabled>b</button>

添加 disabled 这个 attribute，将 disabled 这个 property 初始化为 true，按钮被禁用。不论disabled attribute的值为"false"、"true"还是空串""，disable property都是true。只有用

a.removeAttribute('disabled')

删掉disabled attribute，disabled property就变成了false，可以点击。

<button disabled="false"></button>

上面这种写法是错误的，或者说没用，即使写成这样仍被禁用

a.disabled=false

控制台输入这一句按钮也能用，同时disabled attribute不改变，可以是"niubi", "true"等，说明真正决定按钮能不能按的是disable property，而不是disable attribute。

* 有些 HTML attribute 没有对应的 property，如 colspan。
* 有些 DOM property 没有对应的 attribute，如 textContent。

## 绑定目标的类型

* 属性 Property
* 元素的 property

<button [disabled]="isUnchanged">Cancel is disabled</button>

<img [src]="heroImageUrl">

* 组件的 property

模板表达式应该返回目标属性所需类型的值。 如果目标属性想要个字符串，就返回字符串。 如果目标属性想要个数字，就返回数字。如果目标属性想要个对象，就返回对象。

HeroDetail 组件的 hero 属性想要一个 Hero 对象，那就在属性绑定中精确地给它一个 Hero 对象

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

可是直接这样写报错……

* 指令的 property

<div [ngClass]="classes">[ngClass] binding to the classes property</div>

* CSS 类 class

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

isSpecial为true或者是个string等等，div都将带有special类。如果是false，则没有。

这样只能操作单个class。

<div **[ngClass]**="classes">nihaoshijie</div>

classes : {};

this.classes={

red:true,

big: true

}

}

这样div就具有red和big两个class

* 样式 style

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

相当于写了行间的style样式 <button style="color: red"> 设置单一样式值

currentStyles: {};

setCurrentStyles() {

this.currentStyles = {

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

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

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

};

}

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

绑定的对象属性名和属性值都加''

## 绑定事件

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

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

onKey(event: any) {

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

}

# service

* 创建一个Service

ng generate service hero

ng g s hero

生成src/app/hero.service.ts

* 使用根注入器注册

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

@Injectable({

providedIn: 'root',

})

export class HeroService {

constructor() { }

}

* 必须先创建/注册服务提供商create/register a provider，来让HeroService可用
* 服务提供商用来交付或提供一个服务。
* 使用注入器injector来注册服务提供商。注入器是一个对象，负责当应用要求获取它的实例时选择和注入该提供商。
* 默认情况下，ng generate service命令通过给@Injectable()添加providedIn: 'root'，用根注入器注册服务提供商。
* 在顶层提供该服务时，Angular 就会为 HeroService 创建一个单一的、共享的实例，并把它注入到任何想要它的类上。在 @Injectable 元数据中注册该提供商，还能允许 Angular 通过移除那些完全没有用过的服务来进行优化。
* 特性模块中注册

@Injectable({

providedIn: UserModule,

})

export class UserService {}

可以把服务注册到指定模块下，只有此模块能使用该服务。就这一行就行了，这是首选方法。下面是另一种方法。

user.module.ts

@NgModule({

providers: [UserService],

})

export class UserModule {}

也可以在模块中为该服务声明一个提供商。表示这个模块中要使用这个服务。但是这样写服务本身的@Injectable()里怎么写就不知道了。应该啥也不写？而且这种写法是Angular6.0以前的写法了，忘了它吧

* 在login.component.ts中import引入AuthService，在构造中初始化service，在onClick中调用service。

import { AuthService } from '../core/auth.service';

export class LoginComponent implements OnInit {

service: AuthService;

constructor() {

this.service = new AuthService();

}

ngOnInit() {

}

onClick(username, password) {

console.log(this.service.loginWithCredentials(username, password));

}

}

最好不要使用这种写法，不然后面的都白看了。用依赖注入写法

# Dependency Injection依赖注入

用服务商provider配置注入器injector

可以通过在以下三种位置之一设置元数据，来在不同层级使用服务商配置注入器

* 在服务本身的 @Injectable() 装饰器的providedIn属性
* NgModule 的 @NgModule() 装饰器的providers选项
* 在组件的 @Component() 装饰器的providers选项

前两种在应用结构中的层级是一样的，都是ModuleInjector。推荐第一种

后一种配置的是ElementInjector，查找服务时优先于前两种。

在某个注入器的范围内，服务是单例的。在指定的注入器中最多只有某个服务的一个实例。

应用只有一个根注入器。如果服务注册到根注入器root上，则root的范围内只有一个实例，也就是整个应用中只用中只有一个实例。

举个例子。有服务S1Service，S1Service有属性name，是个数字。有方法changeName()，调用一次name+1

export class S1Service {

name: number = 1

changeName(): void{

this.name=this.name+1;

}

}

有组件CccComponent注入了S1Service，可以看到name值

export class CccComponent implements OnInit {

constructor(public s1: S1Service) { }

click(): void{

this.s1.changeName();

}

}

CccComponent.html

<p>

{{s1.name}}

<button (click)="click()">点击增加</button>

</p>

app.component.html

<app-ccc></app-ccc>

<app-ccc></app-ccc>

1.如果服务注册到root上，则全应用范围内S1Service是单例的，所有组件共享一个S1Service

@Injectable({

providedIn: 'root'

})

显示形如

13

13

上下两个数字永远保持一致

2.如果服务注册到ElementInjector上，使用@Component装饰器的providers属性配置服务商

@Component({

providers: [S1Service]

})

则显示形如

6

12

两个组件有各自的服务实例，数字可以不同

此时也没有必要把@Injector的providedIn清空，因为ElementInjector优先于ModuleInjector

Angular 中有两个注入器层次结构：

* ModuleInjector 层次结构
* ElementInjector 层次结构

## ModuleInjector

可以通过以下两种方式之一配置 ModuleInjector ：

* 使用 @Injectable() 的 providedIn 属性引用 NgModule 或 root 。

@Injectable({

providedIn: 'root'

})

export class ItemService {}

providedIn 属性配置 ModuleInjector。此例中，在root ModuleInjector根注入器范围内，服务可用。这里没有出现providers提供商，但其实就相当于AppModule提供了ItemService

* 使用 @NgModule() 的 providers 数组。

优先第一种，因为使用 @Injectable() 的 providedIn 时，优化工具可以进行摇树优化tree-shaking，从而删除应用中未使用的服务，减小捆绑包尺寸bundle sizes。

ModuleInjector是providers数组的扁平化，这些数组可以通过递归追溯 NgModule.providers和NgModule.imports 找到。

子ModuleInjectors在惰性加载其他模块时创建

* 平台注入器Platform injector

PlatFormInjector也是ModuleInjector，相当于root的父注入器

在 root 之上还有两个注入器，

* 一个额外的 ModuleInjector，即Platform Injector
* NullInjector()

Angular 通过 main.ts 中的如下代码引导应用程序：

platformBrowserDynamic().bootstrapModule(AppModule).then(ref => {...})

* platformBrowserDynamic() 方法创建一个由 PlatformModule 配置的ModuleInjector，该注入器包含特定平台的依赖项。即Platform Injector
* bootstrapModule() 方法会创建一个由AppModule配置的ModuleInjector作为Platform Injector的子注入器。也就是root。
* PlatFormInjector的父注入器是 NullInjector()，它是树的顶部。如果您在树中向上走了很远，以至于要在 NullInjector() 中寻找服务，那么返回错误
* 在依赖注入的这章教程里，总是会说一个Module配置configure一个ModuleInjector。
* 虽然 root 是一个特殊的别名，但其他 ModuleInjector 都没有别名。每当创建动态加载组件时，还会创建 ModuleInjector，比如路由器，它还会创建子 ModuleInjector。

@Injectable() vs. @NgModule()

如果在 AppModule 的 @NgModule() 中配置应用级提供商

src/app/app.module.ts

providers: [

{ provide: LocationStrategy, useClass: HashLocationStrategy }

]

它就会覆盖一个在 @Injectable() 的 root 元数据中配置的提供商。可以用这种方式，来配置root级别的的非默认提供商。

比如上述这样配置，全局范围内LocationStrategy变成HashLocationStrategy

## ElementInjector

Angular 会为每个DOM元素隐式创建 ElementInjector。这个Dom元素应该指的是指令或组件

可以用 @Component() 装饰器中的 providers 或 viewProviders 属性来配置 ElementInjector 以提供服务。

@Component({

providers: [{ provide: ItemService, useValue: { name: 'lamp' } }]

})

export class TestComponent

* 组件@Conponent是一种特殊类型的指令@Directive
* @Directive() 具有 providers 属性，@Component()也继承下来。都可以使用 providers 属性来配置提供商。
* 当使用providers属性为组件或指令配置提供商时，该提供程商就属于该组件或指令的 ElementInjector。
* 同一元素上的组件和指令共享同一个注入器。这句话是真看不懂，啥叫同一元素上的组件和指令。啥是元素
* 这里明确说明了，providers属性配置提供商。提供商属于belong to注入器。Component或Directive有注入器

组件类可以通过两种方式提供服务：

* 使用 providers 数组

@Component({

providers: [

{provide: FlowerService, useValue: {emoji: '🌺'}}

]

})

* 使用 viewProviders 数组

@Component({

viewProviders: [

{provide: AnimalService, useValue: {emoji: '🐶'}}

]

})

* providers和viewProviders基本上差不多，providers从当前组件的逻辑树中搜索服务，比如AaaComponent是BbbComponent的父组件，则BbbComponent里的服务解析顺序就是BbbComponent -> AaaComponent ->AppModule
* ViewProviders是从渲染树中搜索服务，每个组件有一个#view的概念，#view就能渲染组件。有一种特殊情况是BbbComponent显示在AaaComponent当中，但是不在AaaComponent的#view当中，服务的解析顺序就跳过了AaaComponent，变成BbbComponent 🡪AppModule
* 就用providers吧
* providers: [ AnimalService ] 这种方式可以直接注册
* providers: [ { provider: AnimalService, useValue:{ emoji: 'gou'}] ]这种必须要设置useValue属性，给Service的属性赋值

@Inject(Token)=>Value 表示，如果要将 Token 注入逻辑树中的此位置，则它的值为 Value。

@Provide(Token=Value) 表示，在逻辑树中的此位置存在一个值为 Value 的 Token 提供商的声明。

@NgModule(Token) 表示，应在此位置使用后备的 NgModule 注入器。

当使用提供商配置注入器时，就会把提供商和一个 DI Token令牌关联起来。注入器维护一个令牌-提供商的map，当请求一个依赖项，即组件constructor()中注入了服务时，就会引用它。令牌就是这个map的键。

当 AppComponent 请求 FlowerService 时，注入器的工作就是解析resolve FlowerService 令牌。令牌的解析分为两个阶段：

* 注入器确定逻辑树中搜索的开始位置和结束位置。注入程序从起始位置开始，并在逻辑树的每个级别上查找令牌。如果找到令牌，则将其返回。
* 如果未找到令牌，则注入程序将寻找最接近的父 @NgModule() 委派该请求。

@Component({

selector: 'app-villains-list',

templateUrl: './villains-list.component.html',

providers: [ VillainsService ]

})

VillainService 对于 VillainsListComponent 来说是单例的，因为它就是在这里声明的。只要 VillainsListComponent 没有销毁，它就始终是 VillainService 的同一个实例。但是对于 VillainsListComponent 的多个实例，每个 VillainsListComponent 的实例都会有自己的 VillainService 实例。

## 解析规则

先解析ElementInjector 层次结构

后解析ModuleInjector 层次结构

* 当服务注入后，angular开始寻找并试图满足依赖。首先尝试使用组件自己的 ElementInjector 来满足该依赖。
* 如果组件的注入器缺少提供商，它将把请求传给其父组件的 ElementInjector 。
* 如果在任何 ElementInjector 中都找不到提供商，它将返回到发起请求的元素，并在 ModuleInjector 层次结构中进行查找。如果仍然找不到，它将引发错误。
* Angular 会用遇到的第一个提供商来解析该依赖。找到了之后不会再寻找同一服务的其他提供商。

示例：

使用providedIn配置ModuleInjector

flower.service.ts

@Injectable({

providedIn: 'root'

})

export class FlowerService {

emoji = '🌺';

}

配置ModuleInjector，配置在根注入器中。

app.component.ts

export class AppComponent {

constructor(public flower: FlowerService) {}

}

<p>Emoji from FlowerService: {{flower.emoji}} </p>

显示Emoji from FlowerService: 🌺

* 使用@Component( providers:[])配置ElementInjector

child.component.ts

@Component({

selector: 'app-child',

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

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

providers: [{ provide: FlowerService, useValue: { emoji: '🌻' } }]

})

export class ChildComponent {

constructor( public flower: FlowerService) { }

}

显示Emoji from FlowerService: 🌻

因为ElementInjector优先于ModuleInjector

请求 FlowerService 时，FlowerService 在ChildComponent的 providers 数组中解析为向日葵🌻。注入器找到之后就不在注入器树中进一步查找。一旦找到 FlowerService ，它便停止运行，再也看不到🌺（红芙蓉）。

## 解析修饰符

可以使用 @Optional()，@Self()，@SkipSelf() 和 @Host() 来修饰 Angular 的解析行为。

修饰符的类型

* @Optional()

@Optional() 允许 Angular 将您注入的服务视为可选服务。这样，如果无法在运行时解析它，Angular 只会将服务解析为 null，而不会抛出错误。

这儿有一个没有注册的服务S2Service

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

@Injectable()

export class S2Service {

name: string = 'giao'

constructor() { }

}

现在往AaaComponent里注入S2Service，会报错

import { S2Service } from '../s2.service';

export class AaaComponent implements OnInit {

constructor(private s2: S2Service) { }

ngOnInit() { }

}

NullInjectorError: StaticInjectorError(AppModule)[AaaComponent -> S2Service]:

从这里也可以看出，查找服务最终的一层是NullInjector

加上@Optional()修饰符，就不报错了

constructor(@Optional() private s2: S2Service) {

}

不报错也没d用，s2里所有的属性和方法都不能用

* @Self()

使用@Self() 让 Angular 仅查看当前组件或指令的 ElementInjector 。

@Self() 的一个好例子是要注入某个服务，但只有当该服务在当前宿主元素上可用时才行。

constructor(@Self() private s1: S1Service) { }

当注入一个在根注入器注册的服务时，将会报错

@Injectable({

providedIn: 'root'

})

又是一个没D用的，不写@Self还不是先检查ElementInjector？

* @SkipSelf()

@SkipSelf() 与 @Self() 相反。使用 @SkipSelf() ，Angular 在父 ElementInjector 中开始搜索服务。

假设有服务S1注册在根注入器中

@Injectable({

providedIn: 'root'

})

export class S1Service {

name: string = 'hello'

constructor() { }

showName(): void {

console.log(this.name)

}

}

直接注入或使用@Self()修饰符注入，效果是一样的

constructor(@Self() private s1: S1Service) {}

ngOnInit() {

console.log(this.s1.name);

}

显示hello

此时提供Component级别的ElementInjector，用providers属性配置

@Component({

providers: [{provide: S1Service, useValue:{name: 'world'}}]

})

显示world

使用@SkipSelf修饰符注入

constructor(@SkipSelf() private s1: S1Service) { }

则又显示hello

* @Host()

@Host() 使您可以在搜索提供商时将当前组件指定为注入器树的最后一站。即使树的更上级有一个服务实例，Angular 也不会继续寻找。这tm和@Self有啥区别……服了

@Host和@Self应该是有区别的，而且一般和viewProviders配合。

可能有只限定在ElementInjector中搜索服务的作用，但是也不是很清楚。

## 提供商

* 依赖提供商会使用 DI 令牌来配置注入器，注入器会用它来提供这个依赖值的具体的、运行时版本。
* 注入器依靠 "提供商配置" 来创建依赖的实例，并把该实例注入到组件、指令、管道和其它服务中。你必须使用提供商来配置注入器，否则注入器就无法知道如何创建此依赖。
* 注入器创建服务实例的最简单方法，就是用这个服务类本身来创建它，注入器的默认行为就是 new 出这个类实例。这样配置就是默认创建方式

providers: [Logger]

不过，也可以用一个替代提供商来配置注入器

替代类

不同的类都可用于提供相同的服务。 比如，下面的代码告诉注入器，当组件使用 Logger 令牌请求日志对象时，给它返回一个 BetterLogger 实例。

[{ provide: Logger, useClass: BetterLogger }]

你可以提供一个类似于 Logger 的对象。

你的提供商可以调用一个工厂函数来创建 logger。

# module

模块**Module**

提供相对独立功能的功能块，每块聚焦于一个特定业务领域。

每个Angular应用都有根模块，通过引导根模块来启动应用。根模块叫做AppModule，被放在 app.module.ts 文件中

生成带路由的 NgModule

ng generate module heroes --routing

src/app/heroes/heroes-routing.module.ts

src/app/heroes/heroes.module.ts

注意，创建module时会新建一个文件夹。component同理

ng generate module heroes/heroes --module app --routing

src/app/heroes/heroes/heroes-routing.module.ts

src/app/heroes/heroes/heroes.module.ts

如果这样使用，则创件两个新文件夹

ng generate module heroes/heroes --module app –-flat --routing

src/app/heroes/heroes-routing.module.ts

src/app/heroes/heroes.module.ts

* 加上--flat参数，发现少创建了一个文件夹

--flat参数的作用，其实就是少创建一个文件夹，在src/app/heroes/heroes文件夹的父目录，也就是src/app/heroes中创建module.ts和routing-module.ts

* 此指令的效果等同于第一个示范
* --module app参数的作用，表示直接注册在AppModule的imports中

ng g c user-page/statistics --module user-page.module

@NgModule({

declarations: [

AppComponent,

LoginComponent

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

* @NgModule装饰器。接收一个元数据对象
* declarations 可声明对象。属于本 NgModule 的组件、指令、管道。在module里面声明的组件在module范围内都可以直接使用，不用import { LoginComponent }

declarations中的组件在当前模块中是可见的，但是对其它模块中的组件是不可见的，除非把它们从当前模块导出exports，并让对方模块导入本模块。

* imports 引入此模块需要的其他模块
* providers 暂时看不懂
* bootstrap 引导性组件/根组件。当Angular引导应用时，它会在DOM中渲染这个引导性组件，并把结果放进index.html的该组件的元素标签中（app-root）。所以bootstrap: [AppComponent]将会是99%Angular应用的配置
* exports 导出表。declarations的子集，能在其它模块使用的的组件。普通模块默认都没有exports数组，根模块exports没用
* 所有模块都得导入CommonModule，而BrowserModule只应该在根模块中导入一次。 CommonModule只包含常用指令的信息，比如 ngIf 和 ngFor，它们在大多数模板中都要用到，而BrowserModule 为浏览器所做的应用配置只会使用一次。
* 导入特性模块组件

即使AppModule中imports了CustomerModule，但是CustomerModule声明了的组件并不对外暴露，如果CustomerModule中有个CustomerDashboardComponent组件，则必须注册到exports数组中，然后app.component.html才能使用

exports: [CustomerDashboardComponent]

app.component.html

<app-customer-dashboard></app-customer-dashboard >

## 引导过程

Angular2通过在main.ts中引导AppModule来启动应用。针对不同的平台，Angular提供了很多引导选项。下面的代码是通过即时（JiT）编译器动态引导，一般在进行开发调试时，默认采用这种方式。

//main.ts

import './polyfills.ts';

// 连同Angular编译器一起发布到浏览器

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

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

import { environment } from './environments/environment';

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

if (environment.production) {

enableProdMode();

}

//Angular编译器在浏览器中编译并引导该应用

platformBrowserDynamic().bootstrapModule(AppModule);

另一种方式是使用预编译器（AoT - Ahead-Of-Time）进行静态引导，静态方案可以生成更小、启动更快的应用，建议优先使用它，特别是在移动设备或高延迟网络下。使用static选项，Angular编译器作为构建流程的一部分提前运行，生成一组类工厂。它们的核心就是AppModuleNgFactory。引导预编译的AppModuleNgFactory的语法和动态引导AppModule类的方式很相似。

// 不把编译器发布到浏览器

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

// 静态编译器会生成一个AppModule的工厂AppModuleNgFactory

import { AppModuleNgFactory } from './app.module.ngfactory';

// 引导AppModuleNgFactory

platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);

## 双向数据绑定

在组件的LoginComponent类中提供成员数据变量，然后在模板template中引用这个数据变量

login.component.ts

@Component({

selector: 'app-login',

template:`

<div>

<input type='text' [(ngModel)]='username'/>

<button (click)='onClick()'>Login</button>

</div>

`,//试了一下别的符号都不行，只有Tab键上面这个`才行

styles: [],

providers: []

})

export class LoginComponent implements OnInit {

username = "";//没有var，写分号

onClick(){

console.log(this.service.loginWithCredentials(this.username));

}

}

[(ngModel)]="username"

* 方括号[]的作用是单向绑定，把等号后面当成表达式来解析而不是当成字符串，username的值设置到HTML的input控件中，就是input.value和username绑定，就是username是啥，input里面就是啥。
* [()]，双向绑定，就是HTML对应控件的状态的改变会设置到username中。只有input里边才用到双向绑定，其他标签都是[]单向绑定，用的很多很多
* ngModel是FormModule中提供的指令，它负责从Domain Model（这里就是username或password，以后我们可用绑定更复杂的对象）中创建一个FormControl的实例，并将这个实例和表单的控件绑定起来。

添加验证

<div>

<input required type="text" [(ngModel)]="username" #usernameRef="ngModel" />

{{usernameRef.errors | json}}

<input required type="password" [(ngModel)]="password" #passwordRef="ngModel" minlength="3" />

{{passwordRef.valid}}

<button (click)="onClick()">Login</button>

</div>

* 给input标签加上required属性，表示表单必填
* 通过#usernameRef="ngModel"又加入了引用，这次的引用指向了ngModel，ngModel就是一个与当前input绑定了的，FormControl的实例，这个引用是要在模板中使用的，所以才加入这个引用，如果不需要在模板中使用，可以不要这句
* 这么说的话，这个FormControll实例有属性

valid：验证通没通过

errors：没通过的话，问题在哪儿

errors包含required，minlength等属性，值为false/true

* {{表达式}}表示解析括号中的表达式。初始状态可以看到2个控件的验证状态都是false。
* passwordRef.valid返回true或false；{{usernameRef.errors | json}}将error对象转换成json格式输出。|是管道操作符，用于将前面的结果通过管道输出成另一种格式。

验证不通过时显示原因

<input type="text" [(ngModel)]="username" #usernameRef="ngModel" required

minlength="3"/>

<div \*ngIf="usernameRef.errors?.required">this is required</div>

<div \*ngIf="usernameRef.errors?.minlength">should be at least 3 charactors</div>

* \*ngIf="usernameRef.errors?.required"的意思是当usernameRef.errors.required为true时显示div标签。
* ?的意思是，因为errors可能是个null，如果这个时候调用errors的required属性肯定会引发异常，'?'就是标明在errors为空时就不调用后面的属性了。errors为空时\*ngIf=false，div不显示。其实就相当于外面的标签，加一个属性ngIf='usernameRef.errors'
* \*ngIf="a"，如果a为null空则不显示

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

</ul>

把表单放在<form>标签中

加入一个表单的引用formRef，就能用下边这俩读form里边input的数据

* formRef.value.username
* formRef.value.password

<div>

<form #formRef="ngForm">

…

</form>

</div>

运行后会发现原本好用的代码出错了，这是由于如果在一个大的表单中，ngModel会注册成Form的一个子控件，注册子控件需要一个name，这要求我们显式的指定对应控件的name，因此我们需要为input增加name属性

<div>

<form #formRef="ngForm">

<input type="text" name="username" [(ngModel)]="username" #usernameRef="ngModel" required minlength="3" />

</form>

</div>

使用fieldset标签来对表单项进行分组

<div>

<form #formRef="ngForm" (ngSubmit)="onSubmit(formRef.value)">

<fieldset ngModelGroup="login">

<input type="text" name="username" [(ngModel)]="username" #usernameRef="ngModel" required minlength="3" />

<div \*ngIf="usernameRef.errors?.required">this is required</div>

<div \*ngIf="usernameRef.errors?.minlength">should be at least 3 charactors</div>

<button type="submit">Login</button>

</fieldset>

</form>

</div>

onSubmit(formValue) {

console.log(this.service.loginWithCredentials(formValue.login.username, formValue.login.password));

}

* fieldset是一个和angular一点关系都没有的HTML标签，这里用到它纯粹是为了分组方便
* ngModelGroup="login"意味着，对于fieldset之内的数据，都分组到了login对象中。
* (ngSubmit)="onSubmit(formRef.value)这样的注册事件的方法和<button (click)="onClick()">差不多，为啥一个有ng另一个没有我就不知道了
* 然后设置button的type=submit，这样button就能点击了而且触发的是form的ngSubmit事件。至于为什么我也不知道
* 本来读取的数据方法是formRef.value.username和formRef.value.password，有了fieldset之后变成了formRef.value.login.username和formRef.value.login.password，所以实参是formRef.value，但是感觉这样做除了增加理解难度没屁用，因为如果再设置一个<fieldset ngModelGroup="login2">，然后点击button，触发的还是onSubmit事件，进了同一个函数，函数里边还不好判断来的是哪个表单里的内容。除非全都要。
* 在LoginComponent中写上onSubmit方法，就是form的ngSubmit事件处理函数onSubmit

# 组件交互

一旦加了路由，父组件中不再是显式的<app-child></app-child>，而是<router-outlet>，就失去了所有父子关系，通信的所有方法也随即失效。那么<router-outlet>的组件如何通信呢？请参见路由章节，参数的传递

* 父组件传到子组件

hero-child.component.ts

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

@Component({

selector: 'app-hero-child',

template: `

<h3>{{hero.name}}</h3>

`

})

export class HeroChildComponent {

@Input() hero: Hero;

}

通过输入型绑定，把数据从父组件传到子组件。

输入型属性带@Input 装饰器。

hero-parent.component.ts

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

import { HEROES } from './hero';

@Component({

selector: 'app-hero-parent',

template: `

<h2>{{master}} controls {{heroes.length}} heroes</h2>

<app-hero-child \*ngFor="let hero of heroes"

[hero]="hero">

</app-hero-child>

`

})

export class HeroParentComponent {

heroes = HEROES;

master = 'Master';

}

父组件把每个循环的 hero 实例绑定到子组件的 hero 属性。

绑定赋值的过程应该是在constructor构造过程中发生的

* 通过 setter 截听输入属性值的变化

name-child.component.ts

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

@Component({

selector: 'app-name-child',

template: '<h3>"{{name}}"</h3>'

})

export class NameChildComponent {

private \_name = '';

@Input()

set name(name: string) {

this.\_name = (name && name.trim()) || '<no name set>';

}

get name(): string { return this.\_name; }

}

* 如果name.trim()不为空，则使用name.trim()，所以我实在不知道教程上这句this.\_name = (name && name.trim()) || '<no name set>';和this.\_name = name.trim() || '<no name set>';有什么区别
* 子组件的模板中，{{name}}绑定的是get name()的返回值
* 带有@Input装饰器的set name()函数，表示这是一个setter。拦截了父组件传过来的值，然后进行处理。
* 所以最终还是需要一个\_name属性，这个属性的值就是最后{{name}}展示的值。之所以使用setter和getter，可能是因为便于数据处理，还有\_name可以声明为private吧
* 过程：父组件传入name值🡪setter函数拦截，参数为name值🡪处理得到name1值🡪通过getter函数返回🡪{{name}}显示

name-parent.component.ts

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

@Component({

selector: 'app-name-parent',

template: `

<app-name-child \*ngFor="let nameValue of names" [name]="nameValue"></app-name-child>

`

})

export class NameParentComponent {

names = ['Dr IQ', ' ', ' Bombasto '];

}

Displays 'Dr IQ', '<no name set>', 'Bombasto'

* 父组件监听子组件的事件

voter.component.ts

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

@Component({

selector: 'app-voter',

template: `

<h4>{{name}}</h4>

<button (click)="vote(true)" [disabled]="didVote">Agree</button>

<button (click)="vote(false)" [disabled]="didVote">Disagree</button>

`

})

export class VoterComponent {

@Input() name: string;

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

didVote = false;

vote(agreed: boolean) {

this.voted.emit(agreed);

this.didVote = true;

}

}

* 子组件暴露一个 EventEmitter 属性，当事件发生时，子组件利用该属性 emits向上弹射值或事件。
* EventEmitter 属性是一个输出属性，带有@Output 装饰器
* 点击按钮会触发 true 或 false(布尔型有效载荷)的事件。

votetaker.component.ts

@Component({

selector: 'app-vote-taker',

template: `

<h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>

<app-voter \*ngFor="let voter of voters"

[name]="voter"

(voted)="onVoted($event)">

</app-voter>

`

})

export class VoteTakerComponent {

agreed = 0;

disagreed = 0;

voters = ['Narco', 'Celeritas', 'Bombasto'];

onVoted(agreed: boolean) {

agreed ? this.agreed++ : this.disagreed++;

}

}

* 子组件有一个EventEmitter属性是一个事件发射器，可以发射任何类型的值，此例中发射boolean值
* 父组件将事件发射器绑定到一个事件处理器onVoted()上，用来响应子组件发射的数据

**本地变量**

父组件的模板中用本地变量引用子组件，就可以得到子组件的属性值和方法

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

子组件的name属性值为{{child.name}}

#child就是本地变量

**@ViewChild()**

本地变量方法是个简单便利的方法。但是父组件-子组件的连接必须全部在父组件的模板中进行。父组件components.ts的代码对子组件没有访问权。

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。

countdown-parent.component.ts

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

export class CountdownViewChildParentComponent{

@ViewChild(CountdownTimerComponent)

private timerComponent: CountdownTimerComponent;

}

* 导入对装饰器 ViewChild 和生命周期钩子 AfterViewInit。
* 通过 @ViewChild 装饰器，将子组件 CountdownTimerComponent 注入到私有属性 timerComponent 里面。
* 不需要本地变量引用

**通过服务来通讯**

父组件和它的子组件共享同一个服务，利用该服务在组件家族内部实现双向通讯。

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。

这个 MissionService 把 MissionControlComponent 和多个 AstronautComponent 子组件连接起来。

component-interaction/src/app/mission.service.ts

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

import { Subject } from 'rxjs';

@Injectable()

export class MissionService {

// Observable string sources

private missionAnnouncedSource = new Subject<string>();

private missionConfirmedSource = new Subject<string>();

// Observable string streams

missionAnnounced$ = this.missionAnnouncedSource.asObservable();

missionConfirmed$ = this.missionConfirmedSource.asObservable();

// Service message commands

announceMission(mission: string) {

this.missionAnnouncedSource.next(mission);

}

confirmMission(astronaut: string) {

this.missionConfirmedSource.next(astronaut);

}

}

# route路由

以下摘自官方教程：

带路由的应用都要在index.html的 <head> 标签中先添加一个 <base> 元素，来告诉路由器该如何合成URL。

如果 app 文件夹是该应用的根目录，那就把 href 的值设置

<base href="/">

但是实践发现，设置<base href=''>，并且使用HashLocationStrategy策略，即在app-routing.module.ts中

imports: [RouterModule.forRoot(routes, {useHash: true})],

最终URL在域名之后导航之前会添加#，形如<http://linran1998.top/subAndCheck/#/login>，

应用才能正常工作

大致过程：遇到<router-outlet>的时候——>进入routing-module，imports中的path中查找——>当前url后边的地址对应了一个path——>转到path对应的Component

* 生成一个路由模块

ng generate module app-routing --flat --module=app

或

ng generate module app-routing --module app --flat

--flat 表示

--module=app 表示属于app模块，CLI把模块自动注册到AppModule的imports数组中

* src/app/app-routing.module.ts

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

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

import { HeroListComponent } from './heroes/hero-list/hero-list.component';

const routes: Routes = [

{ path: 'crisis-center', component: CrisisListComponent },

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

{

path: 'heroes',

component: HeroListComponent,

data: { title: 'Heroes List' }

},

{ path: '', redirectTo: '/heroes', pathMatch: 'full' },

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

];

@NgModule({

imports: [ RouterModule.forRoot(routes) ],

exports: [ RouterModule ]

})

export class AppRoutingModule {}

* path不以'/'开头
* 注意顺序，先正常路由，然后重定向，然后通配符
* 需要import相应的组件
* imports: [RouterModule.forRoot(routes, {useHash: true})] 使用哈希策略
* pathMatch 的另一个可能的值是'prefix'，当剩下的URL 以这个路由中的path开头时，就会匹配上这个跳转路由，进而重定向。

例如{ path: 'aaa', redirectTo: '/heroes', pathMatch: 'prefix' },输入url: '/aaabbccc'，没有匹配到，不跳转；输入url: '/aaa/bb/cc'，匹配成功，跳转。

* RouterModule.forRoot是一个静态的工厂方法，返回一个ModuleWithProviders<RouterModule>，所以才能放在imports数组中

Angular API的RouterModule.forRoot的定义：

static forRoot(routes: Route[], config?: ExtraOptions): ModuleWithProviders<RouterModule>

forRoot接收的第一个参数是一个数组，叫做路由定义RouteConfig数组，数组元素是Route类型的对象。

Route类型对象包括属性：

* path：路由器用它来匹配路径和浏览器url
* component：导航到此路由时，路由器需要创建的组件
* redirectTo：重定向
* pathMatch：路径的字符匹配策略
* children：子路由数组

当注册在根模块时

@NgModule({

imports: [RouterModule.forRoot(ROUTES)]

})

对于子模块和惰性加载的子模块

@NgModule({

imports: [RouterModule.forChild(ROUTES)]

})

* 第三个路由中的 data 属性用来存放于每个具体路由有关的任意信息。该数据可以被任何一个激活路由activated route访问，并能用来保存诸如页标题、面包屑以及其它静态只读数据。
* RouterModule.forRoot(routes,{ enableTracing: true })

enableTracing: true会把每个导航生命周期中的事件输出到控制台。仅用于调试

app.module.ts

@NgModule({

imports: [

HeroesModule,

AppRoutingModule

],

providers: [{ provide: NZ\_I18N, useValue: zh\_CN }],

bootstrap: [AppComponent]

})

export class AppModule { }

要注意根模块imports模块的顺序，AppRoutingModule必须必须在最下面!否则，如果在AppRoutingModule前有一个TempModule，这个module自己的子路由就永远无效了。因为根模块加载时按照import的顺序加载，先加载到AppRoutingModule的路由，匹配到通配符，把所有的url请求都拦截了

* 把模块的根组件中需要路由的部分换成

<router-outlet></router-outlet>

**路由参数**

this.router.navigate(['/heroes']);

无参数导航

必备参数

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

<a [routerLink]="['/hero', hero.id]">

* 第一种，定义path形如hero/:id类型，:id就是一个路由参数的占位符。
* 跳转时，用一个数组给routerLink属性绑定参数。一旦路由配置中声明，此参数必须提供，否则无法匹配。

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

import { switchMap } from 'rxjs/operators';

constructor(

private route: ActivatedRoute,

private router: Router

) {}

this.route.paramMap.pipe(

switchMap((params: ParamMap) =>

this.service.getHero(params.get('id')))

);

* 接收参数
* paramMap是一个observable，观察路由参数
* 直接用subscribe订阅这个observable，可以接受到一个data

this.route.paramMap.subscribe((data) => {

console.log(data)

console.log(data.get('id'))

this.userId = data['params'].id;

})

这个data是一个ParamsAsMap或者ParamMap类型的对象，有params属性

我怀疑是ParamsAsMap是一个类，实现了ParamMap接口

params属性的值是一个对象，包括了路由参数。比如此例中的id。所以data['params'].id可以提取到路由参数中id的值

但是不推荐这种方式提取，至少官方文档上没有写这种方法。

ParamMap是一个接口，data实现了接口的get()方法。所以可以使用data.get('id')提取到路由参数中id的值

* 这种方法有利于复用组件
* switchMap 和map的区别是它具有取消效果。在每次发出时，会取消前一个内部 observable的订阅，然后订阅一个新的
* switchMap操作符接受一个函数A作为参数，A的参数就是observable发布的值，即ParamMap类型的params。函数A返回的值作为新的发布的流。
* pipe操作符处理之后，不管pipe里边是map、switchMap还是其他什么乱七八糟的东西，永远返回一个新的observable

this.userId= +this.route.snapshot.paramMap.get('id');

* 接收参数方法二，用snapShot快照提取url里的:id，但前提是这个组件永远不会被复用。
* 复用的意思是，同一个组件HeroDetailComponent，hero/1如果可以直接导航到hero/2，则倾向于复用组件，只改一改变量；但是如果必须通过hero/1->heroList->hero/2，则每次打开heroDetail都会重新创建组件。复用时，ngInit不再执行，所以用snapShot得到的id值不更新
* +加号目的是string转number
* 实践得知这样拿不到父路由的参数，比如/hero/12/detail，如果:id放在了中间，则取不到id的值。必须设置为hero/detail/12，:id放在最后，HeroDetailComponent才能拿到id。换句话说，每个组件只能拿到给自己组件传递的参数，父组件也拿不到子组件的路由参数。

可选参数

this.router.navigate(['userPage', 'upload', this.userID, {p1: 'aaa'}]);

生成链接/userPage/upload/12;p1=aaa

this.route.snapshot.paramMap.get('p1')

同样的，接收不到父路由的参数

* 可选的路由参数没有使用“？”和“&”符号分隔，它们是用“;”分隔的。这是**矩阵 URL标记法** Matrix URL

虽然 Matrix 写法未曾进入过 HTML 标准，但它是合法的。

使用它可以分别给父路由和子路由单独提供参数

* 还有一个this.router.navigateByUrl('/userPage')方法，没navigate()好用，不要用这个了

this.router.navigate(['../', { id: crisisId, foo: 'foo' }], { relative不知道什么意思To: this.route });

**ActivatedRoute**

路由的路径和参数可以通过ActivatedRoute路由服务来获取。

它有如下属性，可以做到提取父子路由参数。非常关键的一个功能

* url

路由路径的 Observable 对象，是一个数组，数组元素为UrlSegment类型的对象。

每个UrlSegment对象有path属性，连在一起就是当前激活的路由的路径

但并不是完整的url，比如有路由配置path:'user', children: [{path:'name'}]

则url为'/user'时，ActivatedRoute.url包含路径user，url为'/user/name'时，只包含name

* data

暂时不知道有什么用

* 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组成的map对象

* queryParamMap

一个 Observable，其中包含一个对所有路由都有效的查询参数组成的map对象。

* fragment

一个适用于所有路由的 URL 的 fragment（片段）的 Observable。

* outlet

要把该路由渲染到的 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时，表示该路由的父级 ActivatedRoute。

* firstChild

包含该路由的子路由列表中的第一个 ActivatedRoute。

* children

包含当前路由下所有已激活的子路由。

因为activitedRoute只能查找自己路由所在的参数，所以通过parent和firstChild在所有activatedRoute之间穿梭，寻找参数

**routerLink** 属性

<a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>

<a routerLink="/heroes" routerLinkActive="active">Heroes</a>

<router-outlet></router-outlet>

最简单使用方法，给a标签加routerLink属性。字符串以/开头，输入完整路径

RouterLinkActive 指令会基于当前的 RouterState 为活动的 RouterLink 切换所绑定的 css 类。当这个链接激活时，路由器将会给a标签加上active的CSS类，并在处于非活动状态时移除

可以用一组路径片段动态生成链接

<a [routerLink] = "['/team', teamId, 'user', userName, {details: true}]">click</a>

x.component.ts中

teamId:number = 9;

userName:string = 'zhang';

生成一个到/team/9/user/zhang;details=true的链接。

<a [routerLink]="['/user/bob']" [queryParams]="{debug: true}" fragment="education">

link to user component

</a>

教程上说生成/user/bob#education?debug=true

实践得到 /user/bob?debug=true#education

routerLink属性/navigate()方法第一个参数名可以使用 /、./ 或 ../ 前缀：

* 如果第一个片段用 / 开头，则路由器会从应用的根路由开始查找。
* 如果第一个片段用 ./ 开头或者没有用斜杠开头，路由器就会从当前激活路由开始查找。
* 如果第一个片段以 ../ 开头，则路由器将会向上找一级。

一个特性模块的路由模块

特性模块即非根模块，也就是子模块

heroes-routing.module.ts

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

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

const routes: Routes = [

{ path: 'heroes', component: HeroListComponent },

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

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class HeroesRoutingModule { }

* 之后需要在heroes.module.ts模块引入此路由模块，再在AppModule中引入heroes.module。AppModule中引入HeroesModule之后，再无需引入HeroesRoutingModule.
* 非常重要的一步，就是把原来app-routing.module.ts中配置的关于'heroes'路由删除。因为即使是子模块的路由，也匹配的是完整的url。比如这个例子，会匹配'/heroes'和'/heroes/14'的url，和app-routing.module是并列关系，不是父子关系。只有声明出children数组后，才会发展出父子关系。

**路由动画**

子模块的路由模块，包含一些子路由

const crisisCenterRoutes: Routes = [

{

path: 'center',

component: CenterComponent,

children: [

{

path: '',

component: ListComponent,

children: [

{

path: ':id',

component: DetailComponent

},

{

path: '',

component: CenterHomeComponent

}

]

}

]

}

];

* 注意，如果路由配置形如这样，有children数组
* CenterRoutingModule的第一个配置项，path随意，Component必然是CenterComponent。

但是，app-routing.module.ts中，从来不会出现{ path: 'a', component: 'AppComponent' }。

因为此项目必定加载AppComponent，通过引导AppComponent来启动程序。

其实，app-routing.module.ts就隐含了{ path:'', component: AppComponent }的配置项

所以CenterRoutingModule必然通过引导CenterComponent来移动模块，但是配置这一路由的文件本应该是app-routing.module，但是由于它独立出一个模块，所以由模块自身包含的CenterRoutingModule配置

这也是forRoot和forChild是两个方法的原因

CenterComponent之于CenterModule，就如同AppComponent之于AppModule，是“根”模块

* 注意这个示例配置项，有两个children数组。第一个children数组只有一个成员，path为''，可以匹配所有的url。所以ListComponent必然显示。也就是说，CenterComponent中的<router-outlet>标签，永远都是ListComponent
* 第二个children数组是ListCompnent的子路由，在ListComponent的<router-outlet>中切换显示的组件
* 这个示例其实很有意思，有些写法觉得自然而然就是这样，但其实不是理所应当的。比如顶层数组crisisCenterRoutes中只有一个元素，不是巧合，而是每个模块就应该只有一个。比如crisisCenter模块，就应该管url为/crisisCenter/...形式的路由，虽然也可以添加其他路由的配置，但是不符合模块化的规范了。模块只应该管自己模块内部的东西。

**相对导航**

使用绝对路径导航，修改父路径 /crisis-center，之后，就不得不修改每一个<a>标签routerLink的属性。使用相对路径导航，可以避免这一问题

路由器支持在链接参数数组中使用“目录式”语法

* ./ 或 无前导斜线 形式是相对于当前级别的。
* ../ 会回到当前路由路径的上一级。

在routerLink属性和navigate()方法中支持使用链接参数属组

navigateByUrl() 方法总是要指定完整的绝对路径。

this.router.navigate(['../', { id: crisisId, foo: 'foo' }], { relativeTo: this.route });

* 在链接参数数组后面，添加一个带有 relativeTo 属性的对象，并把它设置为当前的 ActivatedRoute。这样路由器就会基于当前激活路由的位置来计算出目标 URL。
* ActivatedRoute 已经隐含在了 RouterLink 指令中，所以在RouterLink指令中不需要添加relativeTo

用**命名出口**（outlet）显示多重路由

**路由守卫**

* CanActivate导航到某路由
* CanActivateChild导航到某子路由
* CanDeactivate当前路由离开
* Resolve在路由激活之前获取路由数据
* CanLoad处理异步导航到某特性模块

路由器会按照从最深的子路由由下往上检查的顺序，来检查 CanDeactivate() 和 CanActivateChild() 守卫。 然后它会按照从上到下的顺序检查 CanActivate() 守卫。

路由器会先按照从最深的子路由由下往上检查的顺序来检查 CanDeactivate() 和 CanActivateChild() 守卫。 然后它会按照从上到下的顺序检查 CanActivate() 守卫。 如果特性模块是异步加载的，在加载它之前还会检查 CanLoad() 守卫。 如果任何一个守卫返回 false，其它尚未完成的守卫会被取消，这样整个导航就被取消了。

如果url形如 user/12，那么就会出现一个问题，光凭根路由守卫无法解决登录的问题。根路由守卫只能保证控制进入user/12页面，进入后则service.isLoggedIn变为1，现在更改url为user/13，就可以畅行无阻。相当于只要有一个用户的账号密码就可以登录其他账号，当然一个用户信息都不知道的人还是无法登录。解决这个问题暂时想到两个方法，第一url里不添加id，但页面还是需要获取用户id，所以需要其他页面间传参数的方法。第二用子路由守卫可能解决。

不对，可以把判断方法从isLoggedIn=1变为userId=urlID，判断url的id参数和登录时保存的参数是否一致

<a routerLink="./" routerLinkActive="active"

[routerLinkActiveOptions]="{ exact: true }">Dashboard</a>

虽然管理仪表盘中的 RouterLink 只包含一个没有其它 URL 段的斜杠 /，但它能匹配管理特性区下的任何路由。 但你只希望在访问 Dashboard 路由时才激活该链接。往 Dashboard 这个 routerLink 上添加另一个绑定 [routerLinkActiveOptions]="{ exact: true }"， 这样就只有当用户导航到 /admin 这个 URL 时才会激活它，而不会在导航到它的某个子路由时。不懂？？？？？？？？？？？？？？

const adminRoutes: Routes = [

{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard],

children: [

{

path: '',

children: [

{ path: 'crises', component: ManageCrisesComponent },

{ path: 'heroes', component: ManageHeroesComponent },

{ path: '', component: AdminDashboardComponent }

]

}

]

}

];

admin有一个自路由，path是''，有children数组，但没有使用 component，就是无组件路由。一个无组件的路由能让守卫子路由变得更容易。

ng generate guard auth/auth

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

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable({

providedIn: 'root',

})

export class AuthGuard implements CanActivate {

canActivate(

next: ActivatedRouteSnapshot,

state: RouterStateSnapshot): boolean {

return true;

}

}

每个CanActivate类都有一个canActivate()方法

AuthService服务能让用户登录，并且保存当前用户的信息。

admin-routing.module.ts

import { AuthGuard } from '../auth/auth.guard';

路由模块导入守卫类

const adminRoutes: Routes = [

{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard],

}

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router) {}

canActivate(

next: ActivatedRouteSnapshot,

state: RouterStateSnapshot): boolean {

let url: string = state.url;

return this.checkLogin(url);

}

checkLogin(url: string): boolean {

if (this.authService.isLoggedIn) { return true; }

// Store the attempted URL for redirecting

this.authService.redirectUrl = url;

// Navigate to the login page with extras

this.router.navigate(['/login']);

return false;

}

}

canActivate返回一个boolean值，true继续，false中断

不对，获取isLoggedIn不是一个异步的过程么，如果还没取到isLoggedIn，后面的代码已经执行到return false怎么办。

答：获取isLoggedIn不是一个异步过程，就是直接获取。点击login按钮后发生了异步过程，更改isLoggedIn，导航。导航的每一次守卫都会检查isLoggedIn，所以login改变了状态，以后就能随意切换。但是退出系统后，再次打开会重新加载，isLoggedIn又成了false，需要重新登录

**惰性加载**

ng generate module aaaa --route cccc --module app.module

这一条指令创建了文件夹aaaa和文件

src/app/aaaa/aaaa-routing.module.ts (336 bytes)

src/app/aaaa/aaaa.module.ts (335 bytes)

src/app/aaaa/aaaa.component.html (19 bytes)

src/app/aaaa/aaaa.component.spec.ts (614 bytes)

src/app/aaaa/aaaa.component.ts (267 bytes)

src/app/aaaa/aaaa.component.css (0 bytes)

* generate module创建了Aaaa模块
* --route cccc 创建了AaaaComponent，还创建了AaaaModule的路由模块AaaaRoutingModule。--module app.module更新了AppRoutingModule，在根路由模块中添加了一行配置

{ path: 'cccc', loadChildren: () => import('./aaaa/aaaa.module').then(m => m.AaaaModule) }

* aaaa-routing.module.ts

const routes: Routes = [{ path: '', component: AaaaComponent }];

默认的配置是空路径，加载AaaaComponent。这样，地址栏输入http://localhost:4200/#/cccc

，就会加载aaaa模块，显示AaaaComponent

如果以前app.module.ts中有对AaaaModule的导入，那么全部删除。因为如果app.module.ts中imports了AaaaModule的话，加载app.module就会立即加载导入的模块。而现在是惰性加载

测试发现惰性加载的模块，像FormsMosule、ReactiveFormsModule、AntDesign都必须重新imports一遍，不然显示不出来。

# 属性型指令

hero-list.component.html里有一行

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

* 虽然没有明确说hero-detail是hero-list的子组件，但是一个组件里用到了另一个组件，就形成了父子关系，所以[hero]="selectedHero"其实是向子组件传递数据
* 这个[hero]表示绑定父组件的selectedHero属性

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

@Input() hero: Hero;

* 子组件变量声明时必须@Input()，表示数据绑定的是父组件里的属性变量，实现数据从上而下流入

hero-detail.component.html

<div \*ngIf="hero"> <!--如果selectedHero为空，则不显示-->

<div>

<label>desc：</label>

<input [(ngModel)]="hero.desc" placeholder="description">

</div>

</div> <!—显示英雄详情-->

<input [(ngModule)]="属性值">动态双向绑定

hero-list.component.ts父组件：

onSelect(hero: Hero): void {

this.selectedHero = hero;

}

点击事件

hero-list.component.html 模板

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

[子组件属性]="父组件（本组件）属性"

向子组件传递数据，子组件属性绑定父组件属性

[class.selected]="hero === selectedHero"

如果当前hero对象===selectedHero，则class="selected"

子组件向父组件传递数据

父组件

<app-calendar (outputEmitter)="getData($event)"></app-calendar>

getData(day) {

console.log(day.year);

}

子组件

@Output() outputEmitter = new EventEmitter();

onClick(){

this.outputEmitter.emit({"hello"});

}

* 子组件创建一个EventEmitter()，暴露给父组件
* EventEmitter是一个输出属性，带有@Output装饰器
* 父组件绑定一个事件处理器到这个事件属性
* 把触发事件的emit方法放在一个click或者其他事件里头click发生时，EventEmitter属性向上emit弹射事件
* 触发这个事件时，getData()方法响应
* 事件参数(用 $event 表示)传给事件处理方法getData()

# 结构型指令

<li \*ngFor="let user of userObservable | async as users; index as i; first as isFirst; last as isLast">

{{i}}/{{users.length}}. {{user}} <span \*ngIf="isFirst">default</span>

</li>

* index: number 当前条目的索引
* first: boolean 如果是第一个则为 true
* last: boolean 如果是最后一个则为 true
* even: boolean 索引号为偶数则为 true
* odd: boolean 索引号为奇数则为 true

结构型指令就是ngIf ngFor

属性型指令就是html标签上本来的属性加上了()和[]

# Observable & RxJS

有很多Observer观察者观察一个Observable可观察对象。这个Observable的获取数据的方式已经在定义时就已经确定。处理数据的方式由每个observer自己定。

* 创建一个Observable可观察对象实例。其中定义一个订阅者函数subscriberFunction作为创建时的参数。subscriberFunction定义了如何获取值。
* 虽然定义了subscriberFunction，但是不会执行。当有消费者consumer调用subscribe()方法时，subscribeFunction就会执行。
* consumer调用subscribe()方法时，需要传入一个观察者observer作为参数。observer有属性next，error，complete，定义了一些回调函数来处理observable可能会发来的三种通知。其中next必要，剩下两个可选。下面一步一步来
* 创建observable
* of(...items)

注意，这是ES6 rest参数写法

返回一个 Observable 实例，它用同步的方式把参数中提供的这些值发送出来。

* from(iterable)

通常用于把一个数组转换成一个（发送多个值的）observable。

import { Observable, Subscription, of } from 'rxjs';

const myObservable = of(1, 2, 3);

of同步的意思是，一次性把数据全发出来，但是是一个接一个的，触发3次next()和一次complete()

我他妈的，这个问题想了200次终于想通了。of(...item)里面如果是类似(1, 2, 3)这种，subscriberFunction就next()3次发布123，如果是of([1, 2, 3])，of一个数组，则这个数组是一个参数，不是三个参数，next()发布一次，直接发布[1,2,3]数组。所以of([1, 2, 3]是Observable<number[]>类型的，of(1, 2, 3)是observable<number>类型的，他妈的不一样

function sequenceSubscriber(observer) {

observer.next(1);

observer.next(2);

observer.next(3);

observer.complete();

return {unsubscribe() {}};

}

const sequence = new Observable(sequenceSubscriber);

这种定义方式与上面的例子用of操作符等价

subscriberFunction 有参数observer，observer有属性next、error等，但不知道具体内容是什么。

subscriberFunction会在适当的时机下执行observer.next()，来发布流。

如果subscriberFunction中没有定义complete()，那么观察就永远不会结束。

* 创建observer

const myObserver = {

next: x => console.log('Observer got a next value: ' + x),

error: err => console.error('Observer got an error: ' + err),

complete: () => console.log('Observer got a complete notification'),

};

注意，next() 和 error() 方法有参数，complete没有。参数就是Observer发布的流

* 调用subscribe

myObservable.subscribe(myObserver);

订阅。显示如下

Observer got a next value: 1

Observer got a next value: 2

Observer got a next value: 3

Observer got a complete notification

有一种不需要显式定义observer的方法，就是在subscribe()方法中，三个函数作为参数

我发现貌似是subscribe()一次，subscribeFunction会一直循环执行，observer.next()也会一直执行。当然前提是subscribeFunction函数体中没有observer.error()或者complete()

next() 函数可以接受消息字符串、事件对象、数字值或各种结构，具体类型取决于上下文。 为了更通用一点，我们把由可观察对象发布出来的数据统称为**流**。任何类型的值都可以表示为可观察对象，而这些值会被发布为一个流

官方文档建议observable的名字以“$”符号结尾

**多播**

实在是看不懂

创建Observable的其他方法

* interval

import { interval } from 'rxjs';

const secondsCounter = interval(1000);

secondsCounter.subscribe(n =>

console.log(`It's been ${n} seconds since subscribing!`));

next()方法的参数n，就是用interval生成的observable想发布的流，是过了1000ms的次数，就是过了几秒

* 从事件中获得observable

import { fromEvent } from 'rxjs';

const div = document.getElementById('div1');

const mouseMoves = fromEvent(div, 'mousemove');

const subscription = mouseMoves.subscribe((evt: MouseEvent) => {

console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);

if (evt.clientX < 40 && evt.clientY < 40) {

subscription.unsubscribe();

}

});

创造一个发布事件的Observable

上面这一段代码的意思是，观察一个div，开始订阅之后，鼠标在div中移动时控制台输出鼠标坐标。x、y坐标都小于40，则停止观察

到目前为止，observable关于事件能够达到的效果，不用observable也能实现。比如这个功能，直接注册一个事件监听器就可以，用不了这么麻烦

**操作符Operators**

* map()

import { map } from 'rxjs/operators';

const nums = of(1, 2, 3);

const squareValues = map((val: number) => val \* val);

const squaredNums =

squaredNums.subscribe(x => console.log(x));

这个observable本来是要发布1,2,3然后complete()

用map操作符转换流，得到一个新observable，变成发布1,4,9

map操作符接受一个参数，参数是一个函数A。函数A的参数即来源observable要发布的流，函数体的内容就是转换它们，然后返回新值。我称之为，转换流

map操作符返回值也是一个函数B，函数B的参数是来源observable，返回值是新observable

import { filter, map } from 'rxjs/operators';

const squareOddVals = pipe(

filter((n: number) => n % 2 !== 0),

map(n => n \* n)

);

const squareOdd = squareOddVals(nums);

squareOdd.subscribe(x => console.log(x));

* filter过滤要发布的值
* pipe把这些操作符链接起来
* 操作符是处理语句，subscribe是订阅语句
* 使用pipe操作符先设置函数，然后绑定observable；使用observable.pipe()方法相当于先绑定了observable，然后设置处理语句

pipe() 同时是Observable 上的一个方法，所以可以用下列简写形式来达到同样的效果

import { filter, map } from 'rxjs/operators';

const squareOdd = of(1, 2, 3, 4, 5)

.pipe(

filter(n => n % 2 !== 0),

map(n => n \* n)

);

squareOdd.subscribe(x => console.log(x));

错误处理

catchError 操作符

tap()就是窃听，对原数据不干扰，但是可以拿到数据，然后console.log()或者干点别的什么

this.http.get(filename, {responseType: 'text'})

.pipe(

tap( // Log the result or error

data => this.log(filename, data),

error => this.logError(filename, error)

)

);

# 部署

ng build --prod

把index.html改成404

在git上发布

# http

现代浏览器支持使用两种不同的API发起HTTP请求：XMLHttpRequest接口和 fetch() API。Angular提供了一个简化的 API 来实现 HTTP 功能，基于浏览器提供的 XMLHttpRequest 接口。

* 导入httpClientModule

根模块导入httpClientModule，其他子模块不需要导入

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

app.module.ts

imports: [

BrowserModule,

HttpClientModule,

]

* 注入服务

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

constructor(private http: HttpClient) { }

* 得到客户端（浏览器端）的文件

this.http.get('assets/temp.txt').subscribe(data=>{

console.log(data)

})

temp.txt中内容的格式必须为json

最后data会自动转换为对象

也可以是字符串，但字符串的" 必须改成\"。所以还是用json格式

* 请求非 JSON 数据

this.http.get(filename, {responseType: 'text'})

这样就可以拿到纯文本，而不需要json了

* 请求带类型的响应数据

this.configService.getConfig()

.subscribe((data: Config) => this.config = {

heroesUrl: data['heroesUrl'],

textfile: data['textfile']

});

Config是一个interface而不是类，因为响应不能自动转换为类的实例。

this.http.get<Config>(this.configUrl);

注意，使用http.get()方法的时候就可以指定响应类型。相当于指定observable观察的类型

先写<>后写()

* 推荐写法

最好写成服务

如果都写在component.ts里，该组件很快就会因为这些数据方式的细节而变得杂乱不堪。 组件变得难以理解、难以测试，并且这些数据访问逻辑无法被复用，也无法标准化。

把数据展现逻辑从数据访问逻辑中拆分出去，也就是说把数据访问逻辑包装进一个单独的服务中，并且在组件中把数据访问逻辑委托给这个服务。

config.service.ts

@Injectable()

export class ConfigService {

constructor(private http: HttpClient) { }

configUrl = 'assets/config.json';

getConfig() {

return this.http.get(this.configUrl);

}

}

configService里有getConfig()方法，获得一个正在观察config.json的observable

config.component.ts

this.configService.getConfig()

.subscribe((data: Config) => this.config = {

heroesUrl: data['heroesUrl'],

textfile: data['textfile']

});

在组件中进行提取和展示数据，通过订阅服务中的observable

这样就把数据展现逻辑和数据访问逻辑分离

* 读取完整的响应体

this.http.get('assets/temp.txt', { observe: 'response' }).subscribe(

data => {

console.log(data)

})

}

直接get得到的data其实是HttpResponse的body属性

添加{ observe: 'response' }配置可以得到整个响应，类型为HttpResponse，包括

* body: "{"name" : "朔州和天大上了两天热搜"}"
* headers 请求头，类型是HttpHeaders
* ok: true OK值
* status: 200 状态码
* statusText: "OK"
* type: 4
* url: <http://localhost:4200/assets/temp.txt>

getConfigResponse(): Observable<HttpResponse<Config>> {

return this.http.get<Config>(

this.configUrl, { observe: 'response' });

}

观察这一段代码，返回值类型为Observable<HttpResponse<Config>>

最外边一层，是因为http.get()方法返回一个observable，普通情况下观察data，也就是HttpResponse.body添加了observe: 'response'配置项以后观察HttpResponse。

HttpResponse 是一个用于事件响应流的 HttpEvent，可以规定httpResponse<Type>，这样写规定了HttpResponse.body的类型。本例中为Config

* 读取响应报文headers

得到HttpResponse之后，通过

for(let key of res.headers.keys()){

console.log(key,res.headers.get(key))

}

得到headers响应头。res当然是HttpResponse的实例啦

* access-control-allow-origin \*
* date Fri, 28 Feb 2020 10:55:09 GMT
* etag W/"34-2XDlEpe8LlkLRPsV18QQv+GZE/c"
* accept-ranges bytes
* x-powered-by Express
* content-length 52
* content-type text/plain; charset=UTF-8
* 发起JSONP请求

看不懂

* 错误处理

有需要再看

* 重试

this.http.get<Config>(this.configUrl)

.pipe(

retry(3),

catchError(this.handleError) // then handle the error

);

retry操作符，对失败的observable重新订阅几次。retry(3)，最多重试3次

必须声明在catchError的前面。先重试后捕获错误

* 添加请求报文headers

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

const httpOptions = {

headers: new HttpHeaders({

'Content-Type': 'application/json',

'Authorization': 'my-auth-token'

})

};

没法直接修改前述配置对象中的现有头，因为 HttpHeaders 类的实例是不可变的。

用 set() 方法，返回当前实例应用更改后的克隆。

httpOptions.headers = httpOptions.headers.set('Authorization', 'my-new-auth-token');

# 动画

动画状态变换的过程是：没开始动画时，div是本身的样式，即由class属性配置。开始后：state1🡺渐变🡺state2，结束后不变回本身的样式

class🡺突变🡺state1🡺渐变🡺state2🡺结束

* 启用动画模块

导入 BrowserAnimationsModule，它能把动画能力引入 Angular 应用的根模块中。

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

imports: [

BrowserAnimationsModule

]

* 把动画功能导入组件文件中

从 @angular/animations 中导入函数。

import {

trigger,

state,

style,

animate,

transition,

// ...

} from '@angular/animations';

* 添加状态

state('open', style({

height: '200px',

opacity: 1,

backgroundColor: 'yellow'

})),

state('closed', style({

height: '100px',

opacity: 0.5,

backgroundColor: 'green'

})),

使用 style() 函数指定一组 CSS 样式，并用state()函数为该状态指定一个唯一的名字

注意，样式的属性必须是小驼峰格式的。比如backgroundColor

如果想把边框从0px变为10px，就得写成

transition("void => \*", [

style({border: "0px solid black"}),

animate("5s 0s linear", style({ border: "10px solid black"}))

])

如果一个状态是border: 0px，另一个状态是border: 10px solid black，那么看到的将是动画进行到一半才有了边框。可能是边框样式默认不是solid，也才开始线性变化。由none变为solid，中间一半的时候突变。当然经过测试border颜色默认也不是black，但主要原因应该还是因为border-style

0px 0%这种，最好设为none

转场状态可以通过 state() 函数进行显式定义，或使用预定义的 \*（通配符）状态和 void 状态。

* 通配符状态

星号 \* 或者叫通配符可以匹配任何一个动画状态。用来定义那些不用在乎起始状态或结束状态的转场动画。

比如，一个 open => \* 转场可应用在当元素的状态从 open 变成任何其它状态时。

transition('\* => closed', [

animate('1s')

])

这样，任意一个状态转为closed的时候都是1s

也可以同时设置open => closed，谁声明靠前谁优先

使用带样式的 \* 通配符来告诉动画使用当前的状态值，并用它进行动画处理。通配符是一个后备值，如果未在触发器中声明动画状态，就会使用这个值。

transition ('\* => open', [

animate ('1s',

style ({ opacity: '\*' }),

),

]),

这个没看懂，应该是自动计算opacity的值

* void 状态

可以使用 void 状态来为进入或离开页面的元素配置转场

* 当元素离开视图时，就会触发 \* => void 转场
* 当元素进入视图时，就会触发 void => \* 转场=

通配符状态 \* 会匹配任何状态 —— 包括 void。

void就是不在视图中的组件，就是用\*ngIf和\*ngFor控制显示

我们把元素进入或离开视图等价于从 DOM 中插入或删除。

<div @myInsertRemoveTrigger \*ngIf="isShown" class="insert-remove-container">

<p>The box is inserted</p>

</div>

trigger('myInsertRemoveTrigger', [

transition(':enter', [

style({ opacity: 0 }),

animate('100ms', style({ opacity: 1 })),

]),

transition(':leave', [

animate('100ms', style({ opacity: 0 }))

])

]),

注意，这个例子不能使用 state()定义状态。

当 HTML 元素没有附着在视图中时，我们就会应用 void 状态。

因为元素不在 DOM 中。 因此，使用别名 :enter 和 :leave 来定位要从视图中插入或删除的 HTML 元素。

transition第二个参数里设置style，然后animate设置第二个参数style来确定状态state。

因为组件的显示和消失是通过\*ngIf来控制的，就不是通过@triggerName的改变来控制的。在组件显示和消失过程中，它没有state，但是有起始和结束状态，通过两个style确定

当任何 \*ngIf 或 \*ngFor 中的视图放进页面中时，会运行 :enter 转场；当移除这些视图时，就会运行 :leave 转场。

:enter 和 :leave 分别是 void => \* 和 \* => void 的别名。 这些别名供多个动画函数使用。

transition ( 'void => \*', [ ... ] );

等同于

transition ( ':enter', [ ... ] );

* 添加转场

transition('open => closed', [

animate('1s')

]),

transition() 动画转场，从 open 到 closed 的持续一秒的状态转场

=> 操作符表示单向转场，而 <=> 表示双向转场。从open到closed和从closed到open一样

用 state() 定义每个转场结束时的样式，这些样式在动画完成后仍会保留。

当禁用了动画时，也会忽略 transition() 中的样式，但 state() 中的样式不会。

animate第一个参数接受一个由三部分组成的字符串。

animate('duration delay easing')

* duration 持续时间 delay 延迟
* 作为纯数字，毫秒为单位：100
* 作为字符串，毫秒为单位：'100ms'
* 作为字符串，秒为单位：'0.1s'
* easing 控制速度，缓动效果
* ease-in 开始慢，后来快
* ease-out 开始快，后来慢
* ease-in-out 开始结束慢，中间快

duration必须，其他可选

使用 animate() 函数来定义长度、延迟和缓动效果

可以在同一个 transition() 参数中包含多个状态对：

transition( 'on => off, off => void' )

* 触发器

trigger() 函数会把状态state()和转场transition()组合在一起，并为这个动画命名

trigger() 函数描述了监听变化时要使用的触发器名称。当这个触发器名称所绑定的值发生了变化时，触发器就会启动它所定义的操作。所以我感觉触发器名称和动画名称是一个

注意：在每个 trigger() 函数调用中，元素在任意时刻只能处于其中的一个状态。但是，元素可以在同一时刻激活多个触发器。也就是说，可以在同一元素上绑定多个动画

<div @ani2 @ani1 \*ngIf="isOpen" class="content">

nihaonihao

</div>

它们可以同时起作用

* 定义动画

在 @Component() 装饰器的 animations: 属性下用trigger()定义动画。

@Component({

animations: [

trigger('openClose', [

state('open', style({

height: '200px',

opacity: 1,

backgroundColor: 'yellow'

})),

state('closed', style({

height: '100px',

opacity: 0.5,

backgroundColor: 'green'

})),

transition('open => closed', [

animate('1s')

]),

transition('closed => open', [

animate('0.5s')

]),

]),

]

})

export class OpenCloseComponent {

isOpen = true;

toggle() {

this.isOpen = !this.isOpen;

}

}

state() 定义了两个状态

transition() 定义了在这两个状态切换时的缓动效果

放到一个trigger里，就组成一个动画

* 把触发器绑定到模板表达式上

<div [@openClose]="isOpen ? 'open' : 'closed'" class="open-close-container">

<p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>

</div>

openClose是触发器名称

open / closed是状态名，[@openClose]取值只能为状态名

当表达式的值变化时，即open变成close或close变成open，动画就会触发。

* 禁用动画

可以把一个名叫 @.disabled 的动画绑定放在 HTML 元素上，以禁用该元素及其子元素上的动画。

<div [@.disabled]="isDisabled">

<div [@childAnimation]="isOpen ? 'open' : 'closed'"

class="open-close-container">

<p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>

</div>

</div>

当 @.disabled 绑定为 true 时，@childAnimation 触发器就不会启动。

* 回调事件

当动画启动和终止时，trigger() 函数会发出一些回调。在下面的例子中，我们有一个包含 openClose 触发器的组件。

export class OpenCloseComponent {

onAnimationEvent ( event: AnimationEvent ) {

}

}

在 HTML 模板中，动画事件通过 $event 传递回来

绑定@trigger.start 和 @trigger.done事件，这里的 trigger 表示所使用的触发器名字。

<div [@openClose]="isOpen ? 'open' : 'closed'"

(@openClose.start)="onAnimationEvent($event)"

(@openClose.done)="onAnimationEvent($event)">

</div>

* 关键帧动画

使用关键帧动画创建一个具有多个顺序执行步骤的动画。

Angular 的 keyframe() 函数类似于 CSS 中的关键帧。关键帧允许在单个时间段内进行多种样式更改。

animate('2s', keyframes([

style({ backgroundColor: 'blue' }),

style({ backgroundColor: 'red' }),

style({ backgroundColor: 'orange' })

])

关键帧包括一个用来定义动画中每个样式何时开始更改的偏移（offset）属性。偏移是个 0 到 1 之间的相对值，分别标记动画的开始和结束时间

只要使用了它，就要同样应用于这个关键帧的每个步骤。

定义关键帧的偏移量是可选的。如果省略它们，就会自动分配均匀间隔的偏移。例如，三个没有预定义偏移的关键帧会分别使用 0、0.5、1 作为偏移。

animate('2s', keyframes([

style({ backgroundColor: 'blue', offset: 0}),

style({ backgroundColor: 'red', offset: 0.8}),

style({ backgroundColor: 'orange', offset: 1.0})

]))

* 复杂转场

trigger("ani3", [

transition(":enter", [

query('li', [

style({width: 0}),

stagger(300, [

animate('300ms', style({width: '\*'}))

])

])

])

])

isOpen = true;

items: Array<string> = [

"还能接受额外的选择",

"rement。",

];

<ul [@ani3]="isOpen">

<li \*ngFor="let item of items">{{item}}</li>

</ul>

* query查找正在进入或离开页面的元素。第一个参数是一个CSS选择器
* <ul>元素绑定了动画ani3，子元素<li>是使用\*ngFor加载进入页面的
* <li>进入页面时，query()由第一个参数'li'查找到正在进入的li，触发动画
* stagger()第一个参数300表示每个li动画间隔300ms。还可以设为-300，先开始最后一个li的动画

开始 \_heros满 heroTotal -1

输入d \_heros变小 newTotal变小成5 heroTotal变小成5

清空 \_heros满 newTotal变大为10 heroTotal变大为10

# 表单

响应式表单 reactive 推荐

模板驱动表单template-driven

## 响应式表单 ReactiveFormsMosule

第一步 注册 ReactiveFormsModule

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

@NgModule({

imports: [

ReactiveFormsModule

],

})

export class AppModule { }

第二步 导入FormControl类，创建form control表单控件，模板中注册

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

<input type="text" [formControl]="name">

name = new FormControl('设置表单初始值');

* 权威数据源The source of truth提供表单元素的value和状态
* 在响应式表单中，表单模型The form model充当权威数据源
* 上例中，FormControl的实例 就是 表单模型。
* 这玩意儿和MVC有点儿像，formControl是model，form是view
* 构造函数优先于placeholder。但是这俩的功能和目的完全不一样。构造函数直接填充了表单，而placeholder只是显示提示，并没有真正输入进去

<input type="text"/>

text 一行纯文本

password 密码

第三步 显示表单控件的值

* 使用 value 属性。它能让你获得当前值的一份快照

<p>

Value: {{ name.value }}

</p>

* 通过可观察对象 valueChanges，你可以在模板中使用 AsyncPipe 或在组件类中使用 subscribe() 方法来监听表单值的变化。

第四步 设置表单控件的值

this.name.setValue('Nancy');

第五步 把表单控件分组

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

profileForm = new FormGroup({

firstName: new FormControl(''),

lastName: new FormControl(''),

});

<form [formGroup]="profileForm">

<label>

First Name:

<input type="text" formControlName="firstName">

</label>

<label>

Last Name:

<input type="text" formControlName="lastName">

</label>

</form>

* FormControl管理一个input控件的值。FormGroup管理多个FormControl的状态
* FormGroup也有value。this.profileForm.value返回

{

firstName: "nihao"

lastName: "shijie"

}

一般来说，取值就用formGoup.value.formControlName

不需要单独创建FormControl，在表单中使用formControlName，创建FormGroup时指定属性名，再new FormControl即可。就像上边那种写法

* 修改部分数据，可以这样

this.fg1.controls.firstName.setValue('111111111111')

FormGroup有controls属性，controls是由所有的FormControl组成的数组。这样通过formGroup.controls.formControl.setValue()修改表单的值。

this.heroForm.get('firstName');

* formGroup.get('formControlName') 官方文档推荐用get()方法访问formControl控件

第六步 提交

<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

<button type="submit">Submit</button>

</form>

onSubmit() : void{

console.warn(this.profileForm.value);

}

* FormGroup指令监听form元素发出的submit事件，并发出ngSubmit事件。ngSubmit事件绑定一个回调函数onSubmit()。
* 点击button触发onSubmit事件
* 这还让用户可以用回车键来提交填完的表单

第七步 嵌套表单组，FormGroup里头套FormGroup

export class ProfileEditorComponent {

profileForm = new FormGroup({

firstName: new FormControl(''),

address: new FormGroup({

street: new FormControl(''),

city: new FormControl('')

})

});

}

<div formGroupName="address">

<label>

Street:

<input type="text" formControlName="street">

</label>

<label>

City:

<input type="text" formControlName="city">

</label>

</div>

* FormGroup 可以同时接纳 FormControl 和 FormGroup 作为子控件
* 里头的FormGroup就不能写<form [formGroupName]>了，要换成<div [formGroupName]>。你懂得，元素里头不能套自己

第八步 修改

可以用上面提到的setValue修改

this.profileForm.patchValue({

firstName: 'Nancy',

address: {

street: '123 Drew Street'

}

});

* patch修改。patchValue修改模型值
* 可以修改部分的formControl
* 注意如果有嵌套的FormGroup，patchValue的参数对象的属性也是嵌套的

第九步 使用 FormBuilder 服务来生成表单控件

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

constructor(private fb: FormBuilder) { }

profileForm = this.fb.group({

firstName: ['', Validators.required],

address: this.fb.group({

street: [''],

zip: ['']

}),

});

FormBuilder 服务有三个方法：control()、group() 和 array()。这些都是工厂方法，用于在组件类中分别生成 FormControl、FormGroup 和 FormArray。

但我也没看出来比直接new方便在哪儿了

第十步 使用表单数组管理动态控件

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

FormArray 是 FormGroup 之外的另一个选择，用于管理任意数量的匿名控件。像 FormGroup 实例一样，可以往 FormArray 中动态插入和移除控件，并且 FormArray 实例的值和验证状态也是根据它的子控件计算得来的。不过，你不需要为每个控件定义一个名字作为 key，因此，如果你事先不知道子控件的数量，这就是一个很好的选择。

用到再说吧

## 表单验证

HTML5 有一组内置的属性，用来进行原生验证，包括 required、minlength、maxlength 等。

这些 HTML5 验证器属性可以和 Angular 响应式表单提供的内置验证器组合使用。组合使用这两种验证器实践，可以防止在模板检查完之后表达式再次被修改导致的错误。但是我现在还不知道这种错误是什么，为什么能避免这种错误。

而且官方文档不同章节上还有一句话，在响应式表单中，权威数据源是其组件类。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）。然后，一旦控件发生了变化，Angular 就会调用这些函数。

<input type="text" formControlName="firstName" required>

**内置验证器**

第一步 导入验证器函数

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

第二步 把字段设为必填required

name = new FormControl('设置表单初始值', Validators.required);

* FormControl.status / FormGroup.status

查看当前状态，不通过验证时为"INVALID"，通过时为"VALID"

* FormControl/FormGroup.valid/invalid

值为false/true

这些属性都可以用来检测输入是否合法

<button type="submit" [disabled]='fg1.invalid'>提交</button>

disabled 属性绑定FormGroup.invalid属性，控制button在验证通过时才能点

**自定义验证器**

export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {

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

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

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

};

}

* 这个函数实际上是一个工厂函数，它接受一个正则表达式，返回一个验证器函数V。
* 这个工厂函数的意义在于，可以生产不同正则表达式的验证器。没有这个工厂一样可以
* 正则表达式用来匹配表单中的字符串
* 验证器函数V的参数是一个AbstractControl类型的对象。因为使用时直接绑定了一个FormControl，所以这个参数由系统自动传入。
* V的返回值是一个对象或null，返回值类型可以写成object | null。本例中如果匹配失败，说明控制器值有效，返回null；如果匹配成功，说明控制器无效，返回验证错误对象。
* 验证错误对象通常有一个名为验证秘钥的属性，此例中为forbiddenName
* 验证错误对象形如{[key: string]: any}，和普通对象不同的是它的key是一个string，和json有点像。但是不知道为什么要这样

添加到响应式表单

this.heroForm = new FormGroup({

'name': new FormControl(this.hero.name, [

Validators.required,

forbiddenNameValidator(/bob/i)

])

});

/bob/i是一个正则表达式，javascript自带的RegExp对象

把这个函数作为参数传给 FormControl。如果有多个验证器，要放到一个数组里

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

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

* 把name设置成一个getter，name相当于一个对象，只不过使用name值的时候通过get()方法得到
* 也可以不设置getter，组件搞一个name属性，赋值this.heroForm.get('name')
* getter函数一般在对象里使用，但是angular有并且热衷于声明在组件类中的写法

<input class="form-control"

formControlName="name" required >

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

class="alert alert-danger">

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

Name is required.

</div>

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

Name cannot be Bob.

</div>

</div>

* \*ngIf="name.invalid && (name.dirty || name.touched) 在表单内容无效、并且表单被修改过或失去焦点后才显示
* formControl.errors就是验证错误对象。只要这个东西不为null，就说明有错。formControl.errors.forbiddenName可以提取具体的错误，因为这是在验证器函数V中设定的
* 不同的错误，会在errors中代表不同的属性。自定义验证器中返回了一个带有forbiddenName属性的对象，errors中就又一条forbiddenName。所以说自定义验证器必须返回对象或null。

以上是官方文档给的示例，由于有工厂函数，所以显得很复杂

如果不要工厂函数，就直接

varify(control: AbstractControl): object | null{

const a = control.value=='Bob';

return a ? {'forbiddenName': '不能写Bob！'} : null;

};

new FormControl('', [

Validators.required,

this.verify

])

**内置验证器**

min() 最小值

max() 最大值

required() 必填

requiredTrue() 必为ture

email() 符合email格式

minLength() 最小长度

maxLength() 最大长度

pattern() 匹配一个正则表达式

# 管道pipe

<p>The hero's birthday is {{ birthday | date }}</p>

birthday = new Date(1988, 3, 15); // April 15, 1988

把组件的birthday变量 通过管道操作符| 流入到Date函数中

# AntD In Angular

$ ng add ng-zorro-antd

## 表单

* <form nz-form>

[nzLayout]="'inline'" 有nzLayout属性。调整布局

[formGroup]="validateForm" 有formGroup属性绑定formGroup

(ngSubmit)="submitForm()" 有ngSubmit属性绑定提交的回调方法

* 表单项 <nz-form-item>

区分表单中不同的区域

* 表单标签 <nz-form-label>

用于标示当前表单项的内容，可选。

* 表单域 <nz-form-control>

表单域可以是输入控件<input>，标准表单域，标签，下拉菜单，文本域等。

nzErrorTip="Please input your username!" 有nzErrorTip属性

[nzSuccessTip] success 时提示信息

[nzWarningTip] warning 时提示信息

[nzErrorTip] error 时提示信息

[nzValidatingTip] 正在校验时提示信息

nzErrorTip和验证器函数一起使用

* 输入组<nz-input-group>

nzPrefixIcon="user" 有nzPrefixIcon属性。应该是决定图标

* 输入框nz-input

<input nz-input name="email" type="email" id="email">

formControlName="userName" 有formControl属性，绑定formControl

<form nz-form>

<nz-form-item>

<nz-form-label [nzSpan]="6" nzFor="email">E-mail</nz-form-label>

<nz-form-control [nzSpan]="14">

<nz-input-group nzPrefixIcon="user">

<input type="text" nz-input formControlName="userName" placeholder="Username" />

</nz-input-group>

</nz-form-control>

</nz-form-item >

</form>

nz-formControl里可以有nz-input-group也可以没有，不知道什么情况下有什么情况下没有

所有nz-col的参数在nz-form-control / nz-form-label上均可直接使用。即可以规定nzSpan

按我的理解，先设置一个form大表格。

然后用nz-form-item分行，一个nz-form-item占一行

所以在同一行的控件，就写到一个nz-form-item里

同一行控件的大小怎么控制？靠nz-form-control / nz-form-label所带的nzSpan控制

nz-form-control / nz-form-label可以不标明nz-col而直接设置[nzSpan]，而且还可以与<div nz-col>组合使用

所以，nz-form-control / nz-form-label都是块级元素是毫无疑问的

nz-form-control里不仅能放input，还可以放nz-cascader级联选择框、nz-date-picker日期选择框、nz-radio-group单选框等等

nzLayout设置为inline后，input变为inline-block。nzSpan就不起作用了，width开始起作用了

表单验证

<nz-form-control [nzSm]="14" [nzXs]="24" [nzErrorTip]="errorTpl">

<input nz-input type="password" formControlName="checkPassword" id="checkPassword" />

<ng-template #errorTpl let-control>

<ng-container \*ngIf="control.hasError('required')">

Please confirm your password!

</ng-container>

<ng-container \*ngIf="control.hasError('confirm')">

Two passwords that you enter is inconsistent!

</ng-container>

</ng-template>

</nz-form-control>

## Grid栅格

24 栅格系统。

根据我多年使用的理解，这东西就是把div的box-sizing设置为border-box，再设置width即可。因为div变为border-box以后，width包括了padding，所以只要width不变，再设置padding是改变不了大小的。但是设置margin会影响，所以一定要避免设置margin

<div nz-row>

<div nz-col nzSpan="12">col-12</div>

<div nz-col nzSpan="12">col-12</div>

</div>

一行24个栅格

通过row在水平方向建立一组column，只有col可以作为row的直接元素

如果一个row中的col总和超过 24，那么多余的col会作为一个整体另起一行排列

<div nz-row nzGutter="16">

使用 nz-row 的 nzGutter 属性设置间隔，推荐使用 (16+8n)px 作为栅格间隔。(n 是自然数)

因为<div nz-col>是box-sizing: border-box，width包括border和padding，所以nzSpan相当于规定了width，而nzGutter实际上是规定了padding。设置nzGutter导致div实际内容区变小，但是width不变，总体宽度不变

<div nz-col nzSpan="8" nzOffset="8">

使用nzOffset="8"设置元素向右侧偏移了8个列（column）的宽度

## Modal模态框

this.modal.success、this.modal.info等创建的模态框也不会通过点击确定销毁。本来以为第二次及以后的创建就是在原来的基础上改文字，不创建新的模态框了，结果就是创建新的模态框，这就很恐怖了，因为可能一直创建然后挤爆内存

# G2