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

E:

cd Codes/angular/hello-angular

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、数据绑定、指令、服务service、依赖注入depedency injection。

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

* src/app/app.component.ts

修改title = 'This is a hello-angular app';

import { Component, OnInit } from '@angular/core';/\*所有组件\*/

import { NgModule } from '@angular/core'; /\*所有模块\*/

import { Injectable } from '@angular/core';/\*所有服务\*/

import { FormsModule } from '@angular/forms'; /\*双向绑定、表格\*/

import { ReactiveFormsModule } from '@angular/forms'; /\*表格\*/

import { Input } from '@angular/core';/\*子组件接收父组件数据\*/

import { RouterModule, Routes } from '@angular/router'; /\*路由相关模块\*/

import { HttpClientModule } from '@angular/common/http';/\*http模块，应写在app.module.ts里，把以前用的httpModule丢掉\*/

# 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

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

Angular文件命名方式：组件名称.component.ts

组件的HTML模板命名为： 组件名称.component.html

组件的样式文件命名为： 组件名称.component.css

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

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

//@Component是Angular提供的装饰器函数，用来描述Compoent的元数据

@Component({

selector: 'app-login',

template: `

<p>

login Works!

</p>

`,

styles: []

})

export class LoginComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

* @Component修饰配置中selector: 'app-login'，在其他组件的template中使用 <app-login></app-login> 来引用这个组件。

比如在hello-angular\src\app\app.component.html中

<app-login></app-login>

* 如果不是内联形式的template和styles时

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

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

* 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

在.ts中声明一个**类**

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

声明变量时不用var

hero-list.component.html

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

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

新建Hero类的文件

ng g class domain/hero

生成以后是一个.ts文件

其他文件引入这个类

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

# service

* 创建一个**Service**

创建一个AuthService

ng g s core\auth

s是service的缩写

core\auth是说在core的目录下建立auth服务相关文件，教程上说先创建一个core目录然后再用这个命令，实际测试不创建core也能自动生成core

为service添加一个方法。这里为这个方法指定了返回类型和参数类型。这就是TypeScript带来的好处，有了类型约束，你在别处调用这个方法时，如果给出的参数类型或返回类型不正确，IDE就可以直接告诉你错了。

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

@Injectable()

export class AuthService {

constructor() { }

loginWithCredentials(username: string, password: string): boolean {

if(username === 'wangpeng')//userName password是两个参数

return true;

return false;

}

}

getHeroes(): Promise<Hero[]> {

return Promise.resolve(HEROES);

}

目前英雄的数据从本地数组中获取，但是终究会从远端服务器获取。当使用远端服务器时，用户不会等待服务器的响应。换句话说，你没法在等待期间阻塞浏览器界面。为了协调视图与响应，我们可以使用承诺（Promise），它是一种异步技术，它需要getHeroes()方法的签名为Promise。

HeroService会生成一个承诺，在有了结果时，它承诺会回调我们。我们请求一个异步服务去做A，并且给它一个回调函数f1。它会去做A（在某个地方），一旦完成，它就会调用f1，并通过参数把工作结果或者错误信息传给我们。

## 在组件component中使用service服务/依赖注入

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

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

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

export class LoginComponent implements OnInit {

//声明成员变量，其类型为AuthService

service: AuthService;//这样声明的意思是a int

constructor() {

this.service = new AuthService();

}

ngOnInit() {

}

onClick(username, password) {

//调用service的方法

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

}

}

* Dependency Injection依赖注入

在组件的修饰器providers中配置AuthService，然后在组件的构造函数中使用参数进行依赖注入

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

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

@Component({

providers:[AuthService]

})

export class LoginComponent implements OnInit {

//这样写的作用：1.声明了一个私有 heroService 属性，2. 把它标记为一个 HeroService 的注入点。

constructor(private service: AuthService) {

//构造函数里不应该写任何事情，应该写在生命周期钩子ngOnInit(){}里边，angular选择合适的时机执行ngOnInit

}

ngOnInit() {

}

onClick(username, password) {

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

//由于service已经成为一个私有属性，调用时直接this.service.function()即可

}

}

* 不import AuthService；在根模块app.module.ts中配置全局可用的service或参数。

providers: [

{provide: 'auth', useClass: AuthService}

]

注意写法区别

provide定义服务名称，有需要注入这个服务的就引用这个名称就好。

useClass指明这个名称对应的服务是一个类 这样定义好之后，我们就可以在任意组件中注入这个依赖了。login.component.ts中，不需要import或者修改providers，更改其构造函数为

constructor(@Inject('auth') private service) {

}

修饰符@Inject('auth')，意思是到系统配置中找到名称为auth的那个依赖 注入到我修饰的变量中。这样改完后你会发现Inject这个修饰符系统不识别，需要在@angular/core中引用这个修饰符，如果没有下面红字，将报错Uncaught ReferenceError: Inject is not defined

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

注意！！ app.module.ts中需要

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

否则报错！这就是我为什么两点04现在才睡的原因！！

* 李罡的教程

import { HeroService } from '../../service/hero.service';

constructor(private heroService: HeroService) { }

app.module.ts

import { HeroService } from './service/hero.service';

providers: [HeroService]

你可以用三种方式之一来设置元数据，以便在应用的不同层级使用提供商来配置各个注入器：

* 在服务 本身的@Injectable() 装饰器中。
* 在模块 NgModule的@NgModule() 装饰器中。
* 在组件 的@Component() 装饰器中。

方法23和李罡的教程，正好是这3中注入器

把 HeroService 注入到 HeroesComponent 之前，你必须先把这个服务提供给依赖注入系统Dependency Injection System。通过注册提供商register a provider来把服务提供给依赖注册系统。A provider is something that can create or deliver a service; in this case, it instantiates the HeroService class to provide the service提供商创建一个服务的实例来提供服务

注册provider🡪provider创建service实例🡪提供服务给依赖注入系统

## Promise & Observable

* Angular HttpClient 的方法会返回 RxJS 的 Observable

导入一些所需的 HTTP 符号：

src/app/hero.service.ts

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

把 HttpClient 注入到构造函数中一个名叫 http 的私有属性中

constructor(

private http: HttpClient,

private messageService: MessageService) { }

保留对 MessageService 的注入。你将会频繁调用它，因此请把它包裹进一个私有的 log 方法中。

/\*\* Log a HeroService message with the MessageService \*/

private log(message: string) {

this.messageService.add(`HeroService: ${message}`);

}

把服务器上英雄数据资源的访问地址 heroesURL 定义为 :base/:collectionName 的形式。 这里的 base 是要请求的资源，而 collectionName 是 in-memory-data-service.ts 中的英雄数据对象。

private heroesUrl = 'api/heroes'; // URL to web api

通过 HttpClient 获取英雄

当前的 HeroService.getHeroes() 使用 RxJS 的 of() 函数来把模拟英雄数据返回为 Observable<Hero[]> 格式

把该方法转换成使用 HttpClient 的

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

}

Http 方法返回单个值

所有的 HttpClient 方法都会返回某个值的 RxJS Observable。

通常，Observable 可以在一段时间内返回多个值。 但来自 HttpClient 的 Observable 总是发出一个值，然后结束，再也不会发出其它值。

具体到这次 HttpClient.get 调用，它返回一个 Observable<Hero[]>，顾名思义就是“一个英雄数组的可观察对象”。在实践中，它也只会返回一个英雄数组。

HttpClient.get 返回响应数据

HttpClient.get 默认情况下把响应体当做无类型的 JSON 对象进行返回。 如果指定了可选的模板类型 <Hero[]>，就会给返回你一个类型化的对象。

JSON 数据的具体形态是由服务器的数据 API 决定的。英雄指南的数据 API 会把英雄数据作为一个数组进行返回。

其它 API 可能在返回对象中深埋着你想要的数据。你可能要借助 RxJS 的 map 操作符对 Observable 的结果进行处理，以便把这些数据挖掘出来。

虽然不打算在此展开讨论，不过你可以到范例源码中的 getHeroNo404() 方法中找到一个使用 map 操作符的例子。

## json-server模拟数据交互

data.json

{

"heroes": [

{

"id": 1,

"name": "The Monkey King",

"desc": "I can fly!"

},

{

"id": 2,

"name": "The Spider Man",

"desc": "I can climb!"

}

]

}

json-server data.json

显示

Resources

http://localhost:3000/heroes

Home

http://localhost:3000

json-server默认端口3000

* GET /heroes：列出所有英雄
* POST /heroes：新建一个英雄
* GET /heroes/ID：获取某个指定英雄的信息
* PUT /heroes/ID：更新某个指定英雄的信息（提供该英雄的全部信息）
* PATCH /heroes/ID：更新某个指定英雄的信息（提供该英雄的部分信息）
* DELETE /heroes/ID：删除某个英雄

api\_url = 'http://localhost:3000/heroes';

headers = new Headers({'Content-Type': 'application/json'});

getHeroes(): Promise<Hero[]> {

const url = `${this.api\_url}`;

return this.http.get(url, {headers: this.headers})

.toPromise()

.then(res => res.json() as Hero[])

.catch(this.handleError);

//上边的方法this.http.get<Hero[]>(this.heroesUrl)，这个是.toPromise()，必须转化成Observable或者Promise，要么就是原始json

}

private handleError(error: any): Promise<any> {

console.error('An error occurred', error);

return Promise.reject(error.message || error);

}

# 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。Angular内建的很多库是以模块形式提供的，比如FormsModule封装了表单处理，HttpModule封装了Http的处理等等。

每个Angular应用至少有一个模块类 —— 根模块，通过引导根模块来启动应用。根模块的类名叫做AppModule，被放在 app.module.ts 文件中，位于 hello-angular\src\app\app.module.ts

## app.module.ts

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

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

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

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

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

@NgModule({

declarations: [

AppComponent,

LoginComponent

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

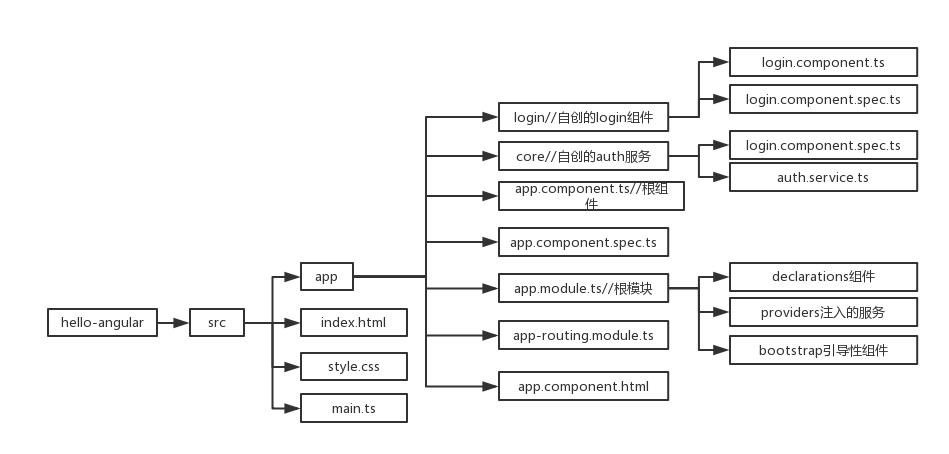
bootstrap: [AppComponent]

})

export class AppModule { }

@NgModule装饰器用来为模块定义元数据。

* declarations 顶层组件。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}
* imports 引入模块BrowserModule FormsModule HttpModule
* providers “注入”的服务（Service），关于依赖性注入会在后面章节中详细解释。
* bootstrap指明哪个组件为引导性组件（本案例中的AppComponent）。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（本案例中的app-root）。



## 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);

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

给login组件的button添加事件。注意括号

<input #usernameRef type="text">

<button (click)="onClick(usernameRef.value)">login</button>

onClick(u){

console.log(u, '被我点击了！')

}

给onClick方法传递一个参数，比如传递上面的文本输入框输入的值：在input标签内加一个#usernameRef，这个叫引用（reference），引用的是input对象

传递input输入框的值，可以用usernameRef.value，然后把onClick()方法改成onClick(usernameRef.value)

## 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？

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

# route

**路由**

1. 在src/index.html中指定基准路径，即<head>第一行<base href="/">

2. 在src/app/app.module.ts中引入RouterModule：import { RouterModule } from '@angular/router';

3. 定义和配置路由数组，我们暂时只为login来定义路由，仍然在src/app/app.module.ts中的imports中

imports: [

BrowserModule,

FormsModule,

HttpModule,

RouterModule.forRoot([

{

path: 'login',

component: LoginComponent

}

])

],

forRoot是一个静态的工厂方法，返回Module

Angular API的RouterModule.forRoot的定义：

forRoot(routes: Routes, config?: ExtraOptions) : ModuleWithProviders

forRoot接收的参数，参数是一个数组，每个数组元素是一个对象。这个数组就叫做路由定义（RouteConfig）数组，每个数组元素就叫路由定义，目前我们只有一个路由定义。路由定义这个对象包括若干属性：

* path：路由器用它来匹配，路由中指定的路径，和浏览器url，例'login'。
* component：导航到此路由时，路由器需要创建的组件，如 LoginComponent 。
* redirectTo：重定向到某个path，使用场景的话，比如在用户输入不存在的路径时重定向到首页。
* pathMatch：路径的字符匹配策略
* children：子路由数组

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

* 在src/index.html中指定基准路径，即<head>第一行<base href="/">
* 生成一个路由模块

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

--flat 把这个文件放进了 src/app 中，而不是单独的目录中。

--module=app 告诉 CLI 把它注册到 AppModule 的 imports 数组中。

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

import { NgModule } from '@angular/core';//所有模块都要import NgModule

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

//引入路由相关模块RouterModule，Routes

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

const routes: Routes = [

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

//如果啥也没有，重定向到/heroes

{ path: 'heroes',component: HeroListComponent }//heroes之前不能加'/'

//url最后是heroes，则把页面中所有的<router-outlet></router-outlet>变为HeroListComponent(<app-hero-list></app-hero-list>)

];

@NgModule({

imports: [ RouterModule.forRoot(routes) ],

// 配置路由/初始化路由器

exports: [ RouterModule ]

// 能在 AppComponent 中使用 RouterOutlet，是因为 AppModule 导入了 AppRoutingModule，而 AppRoutingModule 中导出了 RouterModule

})

export class AppRoutingModule {}

* 将原来的组件显示方式改为路由形式

<app-hero-list></app-hero-list>删掉。改成<router-outlet></router-outlet>

如果前面没有--module=app，则在app.module.ts中不自动imports AppRoutingModule

# 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

<ul class="heroes">

……

</ul>

<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"

# Observable & RxJS

import { Observable, Subscription } from 'rxjs';

创建一个Observable实例。同时定义一个订阅者subscriber函数

observer调用subscribe()，则这个函数执行

const locations = new Observable((observer) => {

const {next, error} = observer;

//next=observer.next, error=observer.error

let watchId;

// Simple geolocation API check provides values to publish

if ('geolocation' in navigator) {

watchId = navigator.geolocation.watchPosition(next, error);

} else {

error('Geolocation not available');

}

subscribe() 调用会返回一个 Subscription 对象，该对象具有一个 unsubscribe() 方法

return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};

});

调用observable的 subscribe() 方法，需要传入一个观察者对象observer

const locationsSubscription = locations.subscribe({

next: (position) => console.log('Current Position: ', position),

error: () => console.log('Error Getting Location: ', msg),

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

});

// Stop listening for location after 10 seconds

setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);

* of(...items) —— 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) —— 把它的参数转换成一个 Observable 实例。 该方法通常用于把一个数组转换成一个（发送多个值的）可观察对象。

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

// Create observer object

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'),

};

// Execute with the observer object

myObservable.subscribe(myObserver);

// Logs:

// Observer got a next value: 1

// Observer got a next value: 2

// Observer got a next value: 3

// Observer got a complete notification

import { fromEvent } from 'rxjs';

const node = document.querySelector('input[type=text]');

// 第二个参数 input 是事件名，对于input元素有一个 oninput 事件用于接受用户输入

const input$ = fromEvent(node, 'input');

input$.subscribe({

next: (event: any) => console.log(`You just typed ${event.target.value}!`),

error: (err) => console.log(`Oops... ${err}`),

complete: () => console.log(`Complete!`)

});

Observable.fromEvent() 会返回一个Observable，并且监听 input 事件，当事件被触发后会发送一个 Event 给对应的observer观察者

引入of

import { of } from "rxjs";

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

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

const squaredNums = squareValues(nums);

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