1.angular启动过程

1.1启动时加载了那些页面

src/index.html

1.2 启动时加载了那些脚本

src/main.ts

1.3这些脚本做了什么事

引导angular引用启动

2.开发前准备

安装jquery

1)npm install jquery—save

指定版本安装

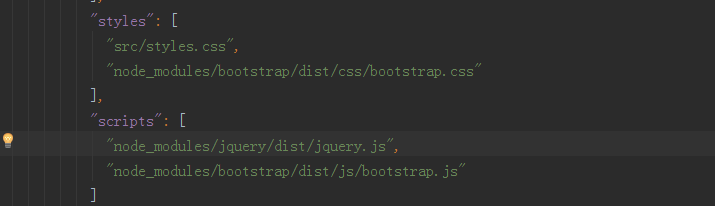
npm install jquery@3.0.0 --save

安装bootstrap

2)npm bootstrap –save

3)项目中引入jquery和bootstrap

在angular.json文件中引入



4）分别安装jquery和bootstrap的类型描述文件，使在typescript文件中可以使用JavaScript文件的代码；

npm install @types/jquery --save-dev

npm install @types/jquery --save-dev

在文件中导入

import \* as $ from 'jquery';

安装 font-Awesome

npm install font-awesome –save

在style.css中引入

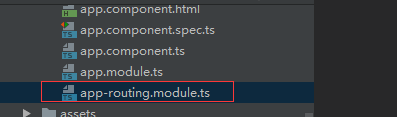
@import '~font-awesome/css/font-awesome.css';

3.angular路由

3.1带路由创建angular项目

ng new 项目名 –routing

项目中多了路由配置文件，在改文件中配置路由的相关信息



3.2了解路由的基本知识

3.2.1路由相关对象

* Routes:路由配置，保存着哪个URL对应展示哪个组件，以及在哪个RouterOutlet中展示组件（必须包含path和component属性，这里path属性中的路径不带”/”）；

使用：

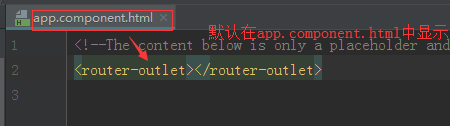
const routes: Routes = [

{path: url, component: componentName},

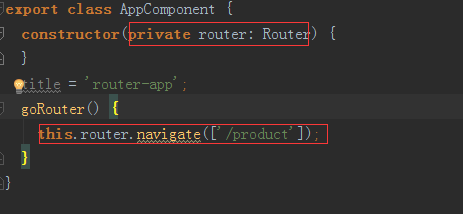
];

**const** routes: Routes = [  
 {path: '', component: HomeComponent},  
 {path: 'product', component: ProductComponent},  
];

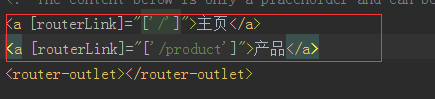
* RouterOutlet：在Html中标记路由内容呈现位置的占位符指令



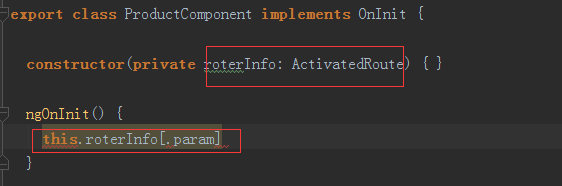
* Router：负责在运行时执行路由的对象，可以通过调用其navigate（）和navigateByUrl（）方法来导航到指定的路由。



* RouterLink：在Html中声明路由导航用的指令



* ActivatedRoute：当钱激活的路由对象，保存这当前路由的信息，如路由地址，路由参数等。



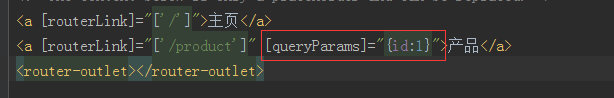
3.2.2通配符路由（）

{path: '\*\*', component: Code404Component}

3.2.3在路由时传递数据

1） 在查询参数中传递数据

* 传递参数：



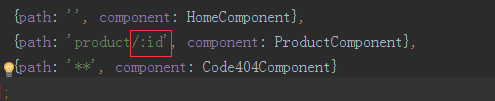
C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\NPACQX@H_DOQJG7%`(0V04A.png

* 接收路由参数：

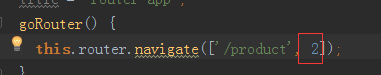
**private** productId: **number**;  
**constructor**(**private** roterInfo: ActivatedRoute) { }  
  
ngOnInit() {  
 **this**.productId = **this**.roterInfo.snapshot.queryParams['id'];  
}

2） 在路由路径中传递参数

* 传递参数



C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\N9_(6DXL)IH`XGPA57M`9S3.png



C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\%6G@8`M9)6_9XA6`697`2`C.png

* 获取参数

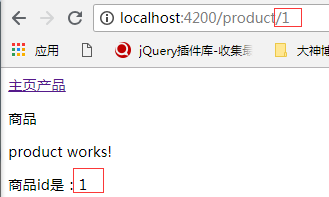
**private** productId: **number**;  
**constructor**(**private** roterInfo: ActivatedRoute) { }  
  
ngOnInit() {  
 **this**.productId = **this**.roterInfo.snapshot.params['id'];  
}

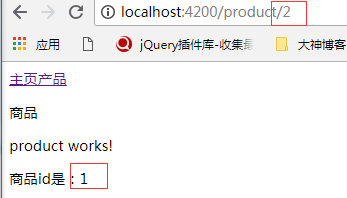
3） 在路由配置中传递数据

4）参数快照和参数订阅

* 参数订阅（从自身路由到自身参数不会改变）

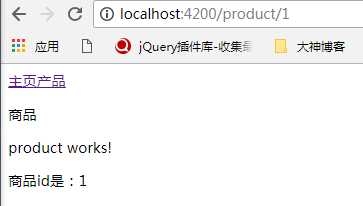
C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\QNEAP42{OF{4VDL}FCRY]LI.png

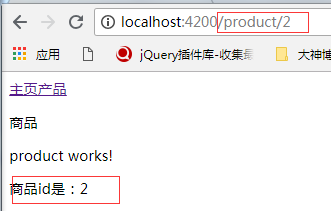




* 参数快照

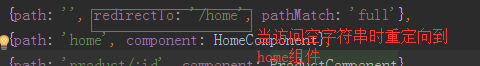
C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\Q_A_)2T0YIA{D}}{{4D1$R2.png





3.2.4重定向路由

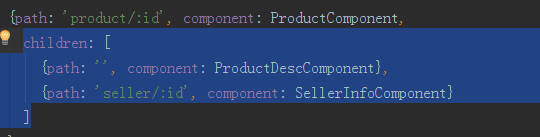
在用户访问一个特定的地址时，将其重定向到另一个指定的地址

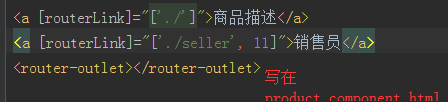


3.3 子路由、保护路由及辅助路由

3.3.1子路由



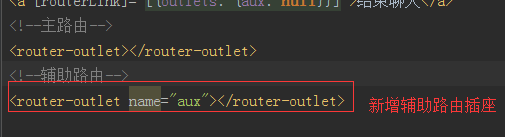




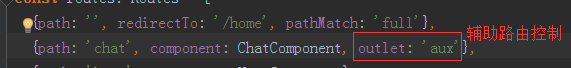
3.3.2辅助路由

同时定义多个路由

* 新增路由插座



* 配置辅助路由



* 添加辅助路由连接



3.3.2路由守卫

当用户已经登录或者拥有某些权限才能进入的路由。

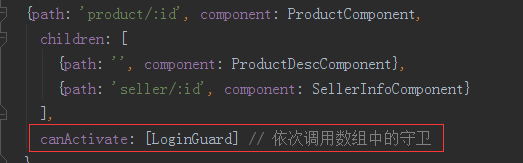
当用户没有保存而离开，提醒用户保存等情况。

* CanActivate:处理导航到某路由的情况

1）写相关程序

**import** {CanActivate} **from** '@angular/router';  
  
**export class** LoginGuard **implements** CanActivate {  
 canActivate() {  
 **const** loggedIn: **boolean** = Math.random() < 0.5;  
 **if** (!loggedIn) {  
 console.log('请先登录');  
 }  
 **return** loggedIn;  
 }  
}

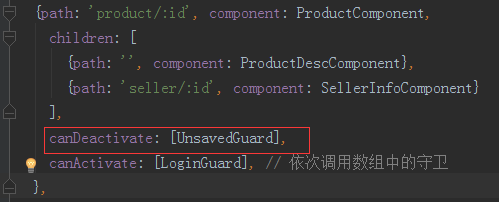
2）在路由配置上使用守卫程序



* CanDeactivate:处理从当前路由离开的情况
* 1）写相关程序

**import** {CanDeactivate} **from** '@angular/router';  
**import** {ProductComponent} **from** '../product/product.component';  
  
**export class** UnsavedGuard **implements** CanDeactivate<ProductComponent> {  
 canDeactivate(component: ProductComponent) {  
 console.log('还没有保存，确定要离开吗');  
 **return** window.confirm('还没有保存，确定要离开吗');  
 }  
}

2）在路由配置上使用守卫程序



* Resolve:在路由激活之前获取路由数据

负责在进入组件之前，把组件需要的数据都加载好，如果拿不到想要的数据或者数据出问题了，就直接跳到错误信息页，或者弹出一些提示，将不再进入目标的路由。

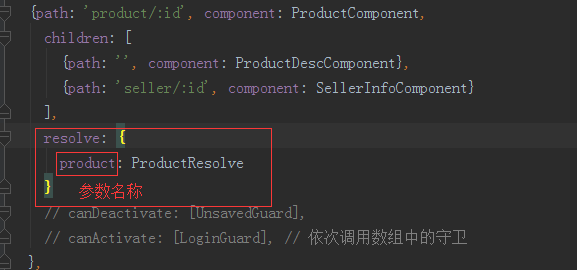
1)编写相关程序代码（守卫代码）

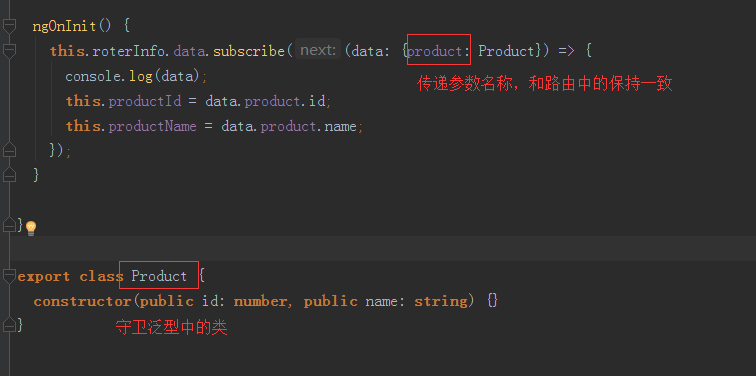
**import** {ActivatedRouteSnapshot, Resolve, Router, RouterStateSnapshot} **from** '@angular/router';  
**import** {Product} **from** '../product/product.component';  
**import** {Observable} **from** 'rxjs/index';  
**import** {Injectable} **from** '@angular/core';  
  
  
@Injectable()  
**export class** ProductResolve **implements** Resolve<Product> {  
 **constructor**(**private** router: Router) { }  
 resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Product> | Promise<Product> | Product {  
 **const** productId: **number** = route.params['id'];  
 **if** ( productId == 1) {  
 **return new** Product(1, 'iphone7');  
 } **else** {  
 **this**.router.navigate(['/home']);  
 }  
 }

要解析的数据类型

**export class** Product {  
 **constructor**(**public** id: **number**, **public** name: **string**) {}  
}

2）在路由中引用





错误:'Injectable' onlyrefers to a type, but is being used as a value here.

解决: 'Injectable'路径引入错误;

4.依赖注入

4.1什么是依赖注入模式及使用依赖注入的好处

4.1.1依赖注入：Dependency Injection,简称DI

依赖注入模式要解决的问题

**var** product = **new** Product;  
createShipmet(product);

使用createShipmet方法需要一个对象参数，这个对象参数通过实例化Product类而得，

也就是说createShipmet方法依赖Product类，但是createShipmet不能创建Product的实例对象。调用createShipmet方法需要通过某种方式创建Product的实例，并将其作为参数传递给createShipmet方法，把一个对象作为一个参数传递给一个方法，也可以叫注入给这个方法；上面采用的是用product变量来接收Product的实例化对象，并注入到createShipmet方法中。如果遇到createShipmet需要传递多个对象参数，或者，传入不同类的参数，则需要大量的手工的实例化对象的代码，如下。

要求：能不能让某人统一创建createShipmet方法所依赖的类，并在需要用的时候完成实例化，我们只需要写 代码createShipmet(product)就可以了。这就是依赖注入要解决的问题。



4.1.2控制反转：Inversion of Control 简称IOC

控制反转是指将依赖的控制权从代码内部转到代码的外部。

以下createShipmet方法需要哪个依赖，是根据createShipmet方法的内部决定的，内部需要什么代码就实例化哪个类。（createShipmet内部需要Product类就实例化Product类，内部需要MockpPoduct类就实例化MockpPoduct类）；

如果实现了控制反转，则内部代码只需要声明需要Product，传入的是Product或者Product的子类MockpPoduct由代码外部决定。依赖注入和控制反转是一体两面，控制反转侧重目的，即目的是将依赖的控制权从代码内部转到代码外部，依赖注入侧重于描述手段，如何实现控制反转，即实现控制反转的手段是依赖注入。实现了控制反转模式的框架被称为IOC容器，angular框架就是一个IOC容器，angular实现控制反转的手段就是依赖注入。

4.1.3使用依赖注入的好处

1）依赖注入会帮助以一种松耦合的方式编写代码，使代码可测性和可从用性更高。

如:在商品组件productComponent中使用一个商品服务来获取商品信息

var productService = new ProductService();

productComponent商品组件将与ProductService紧密耦合在一起。如过在另一个项目中重用productComponent商品组件，但是需要使用另一个不同的服务对象来获取商品信息，必须要修改productComponent组件的代码

var productService = new AnotherProductService();

这说明商品组件productComponent和商品服务ProductService是紧密耦合在一起的，如果在别额项目中用商品组件，需要修改商品组件中的代码才能用。

依赖注入可以帮助接触商品组件和商品服务的紧耦合关系，从而可以在不同的项目中不修改代码的从用商品组件。

4.1.4依赖注入的实现过程



在构造函数内声明需要一个productService的token

当有组件或指令声明自己需要一个类型为productService的token时，实例化一个productService，并将其注入到目标对象。

注册一个类型是productService的token，

providers定义了一个对象在被注入到组件和指令中之前如何实例化

等价于

token：代表可被注入的对象的类型，token的类型由provider配置对象的provider属性来决定。

过程说明：在productComponent组件的构造函数constructor中声明需要一个productService的token，angular看到这样的声明，就会在providers里面找ProductSevice这个类型的token对应的类是哪个（ProductService），并实例化该类，注入到声明的组件或指令中（productComponent组件中）。

如果下其他项目中重用productComponent组件，只需将其声明修改成

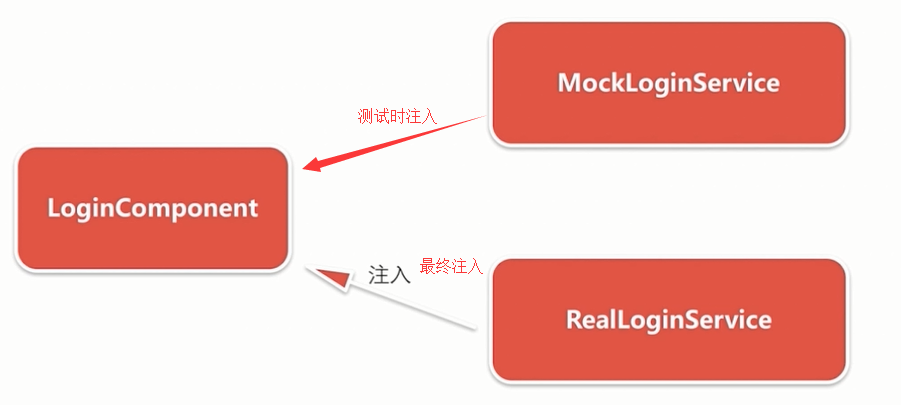


该项目就会实例化AnotherProductService并注入到该组件中。productComponent组件不需要做任何改变，提高了组件的重用性。

2）可测性

依赖注入提高了组件的可测试性，当真实对象还不可用时，可以方便的注入一个虚拟的对象来测试功能。

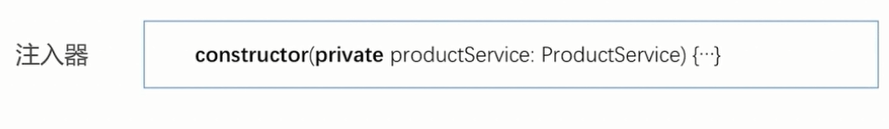
如：为应用添加一个登录功能，创建一个LoginComponent组件来让用户填写用户名和密码，LoginComponent需要依赖一个loginService服务，loginService服务连接一个身份验证服务器，但改服务器还没开发好。可以创建一个MockLoginService，这个服务并不真正的连接身份认证服务器，而是编码一段逻辑来判断是否可以登录，如只有用户名为admin密码为123456时能登录，其他时候都返回用户名或密码错误。使用依赖注入将MockLoginService注入到LoginComponent，不需要等认证服务器开发完才能做测试。当真正的服务器创建好后，将注入改成RealLoginService即可。



4.2介绍angular依赖注入的是实现：注入器和提供器

注入器：

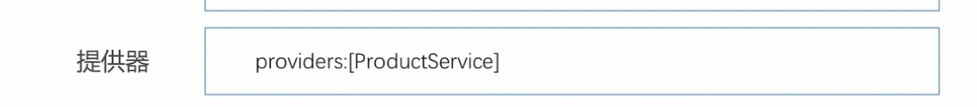
每一个组件都有一个注入器实例负责注入组件需要的对象，注入器是angular提供的一个服务类。一般情况下不需要直接调用注入器的方法。注入器会自动通过组件的构造函数将组件所需的对象注入组件



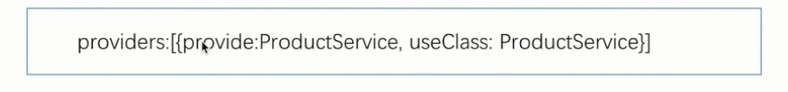
看到以上代码，就会在整个angular应用中寻找ProductService的实例，如果找个这个实例，就会把这个实例注入到该productService对象中，就能使用改服务。

提供器：

要知道如何产生ProductService，需要指定提供器。一般通过组件或者模块的providers属性来声明provider



也可以写成



构造函数中声明中的ProductService和提供器中的provide相同，而根据注入器中的useClass属性来实例化具体的一个类。

还可以通过工厂方法返回一个实例，并把工厂方法返回的实例注入到构造函数的productService属性，可以写一些代码来给工厂做一些初始化的工作。



提供器的 规则：

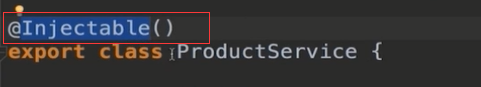
1）.当一个提供器声明在模块时，对所有组件都可见

2）. 当一个提供器声明在组件时，它只对自己和它的子组件可见。

3）. 当声明在组件中的提供器和声明在模块中的提供器有相同的token时，声明在组件中的提供器会覆盖声明在模块中的提供器。

4）. 一般情况下有限将提供器声明在模块中。

4.3服务之间相互注入



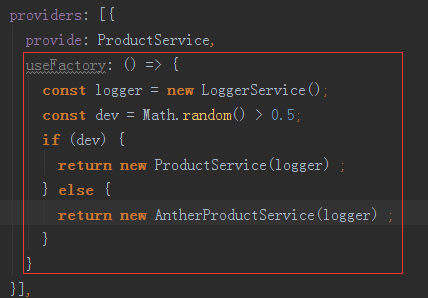
Injectable装饰器表明ProductService这个类也可以通过构造函数注入其它服务

注意：只有声明了Injectable装饰器的服务才能注入其它服务

组件没有Injectable装饰器也能注入服务，因为组件中有Component装饰器，Component是Injectable的子类，因此能注入

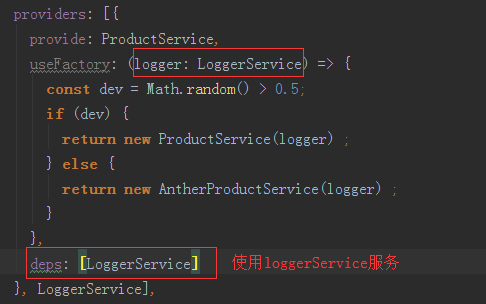
4.4 用工厂方法或之对象来定义提供器

需求：在某些情况下，服务对象需根据不同的条件来决定要实例化那个对象，也有可能在实例化对象时需要传递参数，这时使用工厂提供器。

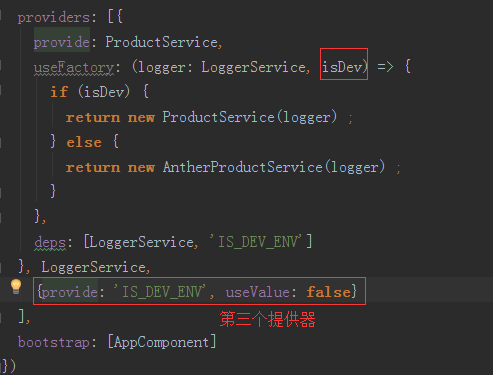


在工厂方法中使用其他服务的提供器

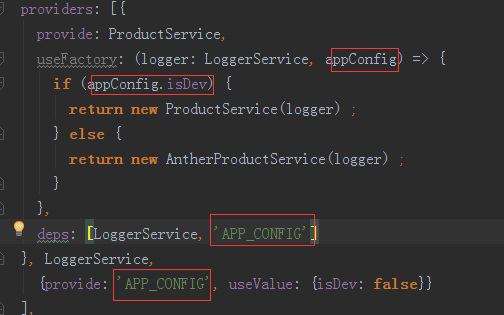
通过deps属性



将变量注入工厂服务



也可以是个对象



4.3注入器的层级关系

注入器:将实例化好的对象注入需要的组件。

注入器的层级关系：

程序启动创建应用级别注入器，将模块中声明的提供器都注册到该注入器中。被注册的提供器包括应用主模块声明的和所有被引用的模块中声明的提供器。

应用级注入器为主组件（AppComponent）创建一个组件及的注入器，并将组件中声明的提供器注册到该组件级的注入器上。

当子组件被创建时，其父组件也为该子组件创建一个注入器，将子组件声明的提供器注册到该子组件上。



注意，angular中，只有一种注入方式，就是构造函数注入方式。

5.数据绑定、响应和编程和管道

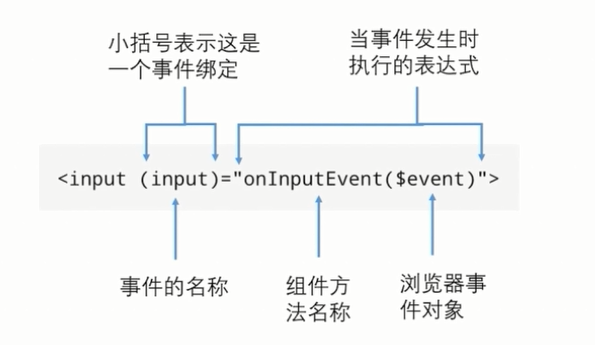
5.1 数据绑定

5.1.1 概述

* {{productTitle}}:使用插值表达式将一个表达式的值显示到模板上。
* <img [src]=”imgUrl” />:使用方括号将HTML标签的一个属性绑定到一个表达式上
* < button (click)=”fn()” ></button>：使用小括号将组件控制器中的一个方法绑定为模板上的一个事件处理器。

5.1.2 事件绑定

格式：<input (input)=”onInputEvent($event)”>



注意：

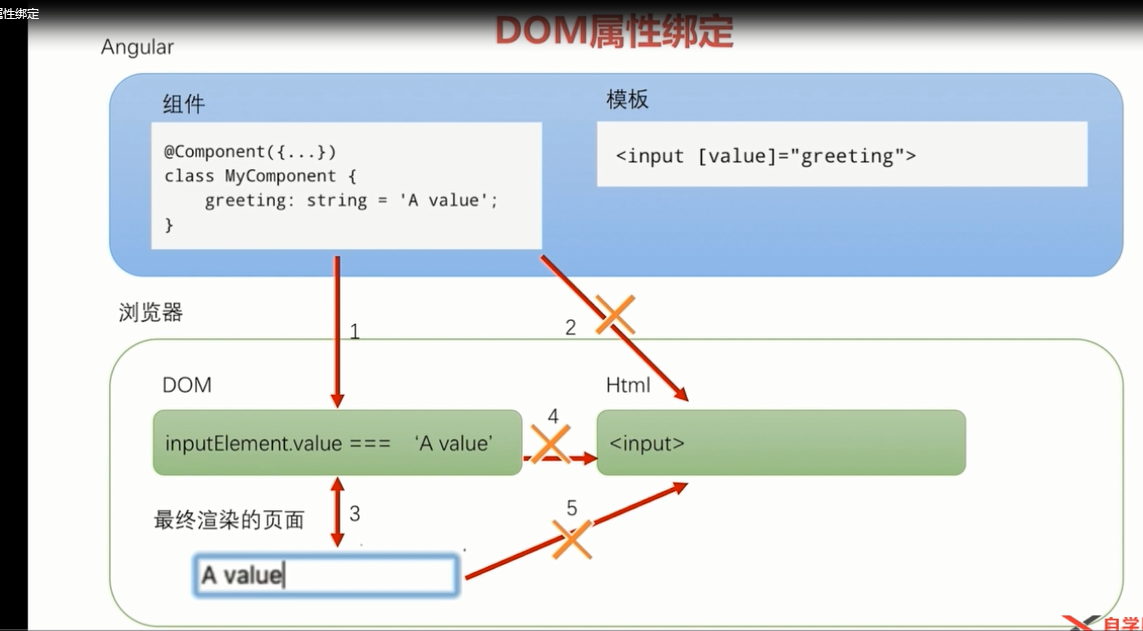
* + 等号右侧的表达式也可以是一个属性赋值：

<button click=”saved=true”></button>

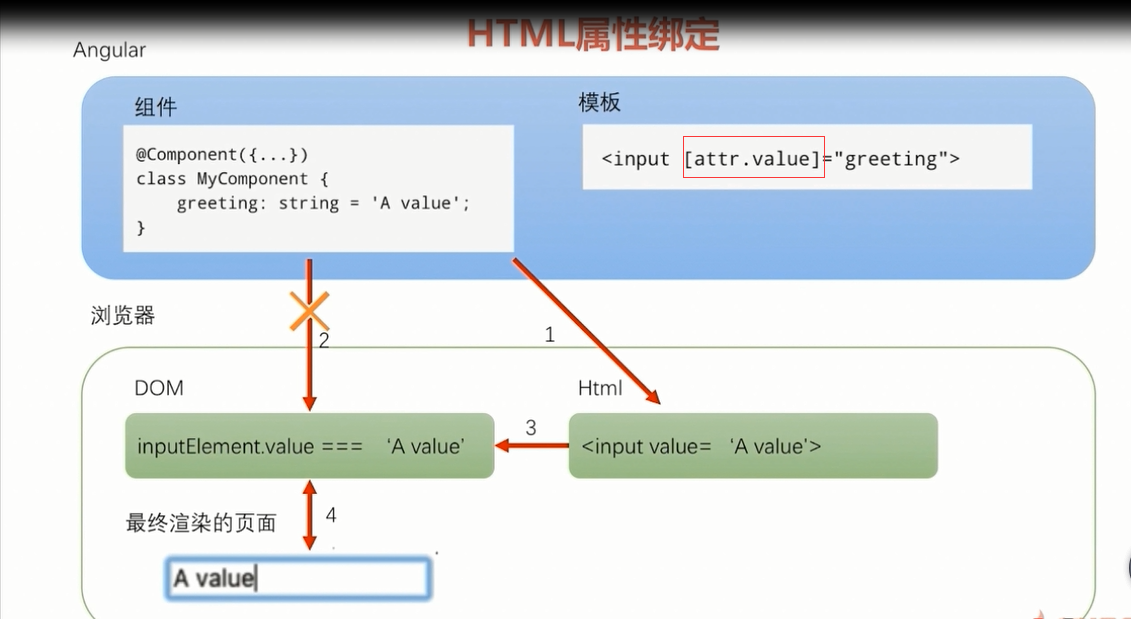
5.1.3 属性绑定

<img [src]="imgSrc"><br>  
<img src="{{imgSrc}}">

angulardom属性绑定



angular的html属性绑定



基本的html属性绑定

<td [attr.colspan]=”tableColspan”>Something</td>

css类绑定

<div class="aaa bbb" [class]="someExpreesion">Something</div>  
<div [class.special]="isSpecial">Something</div>  
<div [ngClass]="{aaa:isA, bbb:isB }"></div>

1） 完全替换

<div [class]="divClass">慕课网</div>

**private** divClass: **string**;  
**constructor**() {  
 setTimeout(() => {  
 **this**.divClass = 'a b c';  
 }, 3000);  
}

2） 替换其中一个

<div class="a b" [class.c]="isBig">慕课网</div>

**private** isBig = **false**;  
**constructor**() {  
 setTimeout(() => {  
 **this**.isBig = **true**;  
 }, 3000);  
}

3）替换多个

<div class="c" [ngClass]="{a:isA , b:isB}">慕课网</div>

**private** isA = **false**;  
**private** isB = **false**;  
**constructor**() {  
 setTimeout(() => {  
 **this**.isA = **true**;  
 **this**.isB = **true**;  
 }, 3000);  
}

样式绑定

<button [style.color]="isSpecial ? 'red': 'green'">Red</button>  
<div [ngStyle]="{'font-size':this.canSave? 'italic': 'normal'}"></div>

html属性和dom属性的关系

* 少量HTML属性和DOM属性之间有着1:1的映射，如id
* 有些html属性没有对应的dom属性，如colspan
* 有些dom属性没有对应的html属性，如textContent
* 就算名字相同，html属性和dom属性也不是同一样东西
* html属性的值指定了初始值，dom属性的值表示当前值；dom属性的值可以改变，html属性的值不能改变。
* 模板绑定是通过dom属性和事件来工作的，而不是HTML属性

5.1.4 双向数据绑定

<input type="text" [(ngModel)]="name">  
{{name}}

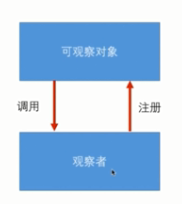
**public** name: **string**;  
**constructor**() {  
 setTimeout(() => {  
 **this**.name = 'Tom';  
 }, 3000);  
}

5.2 响应式编程

响应式编程是以观察者为核心的。

观察者模式和rxjs

观察者模式：由两个对象组成，一个是可观察对象，另一个是观察者。在初始化可观察者对象时，会向可观察对象里注册一些观察者对象。注册完成后，当可观察者对象发生改变时，可调用观察者对象中的一些方法来将变化告诉观察者并进行相应的处理。



rxjs的模型

**var** subscription = Observable.from([1, 2, 3, 4])组成可观察对象  
 .filter((e) => e%2 == 0)  
 .map((e) => e\*e)  
 .subscribe( 被订阅，调用该方法返回一个subscription  
 e => console.log(e),  
 error => console.error(error), 观察者  
 () => console.log('结束了')  
 );

操作符

* 可观察对象Observable（流）:表示一组值或事件的集合。
* 观察者Observer: 一个回调函数集合，他知道怎样监听被Observable发送的值。
* 订阅Subscription：表示一个可观察对象，主要用于取消注册。
* 操作符Operators：纯粹的函数，使开发者可以以函数编程的方式处理集合。

错误：

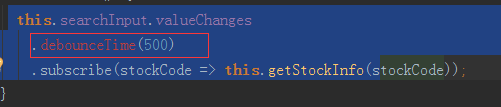
C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\~A@`4YP)X{O8_FO{W26_2IG.png

解决：

**import** 'rxjs/Rx';

改为：

**import** { debounceTime } **from** 'rxjs/operators';



改为：

.pipe(debounceTime(500))

5.3 管道

负责从原始值到管道值的转换

单个使用

{{birthdata | date}}

多个使用

{{birthdata | date | uppercase}}

可接受参数管道

{{birthdata | date:'yyyy-MM-dd HH:mm:ss'}}

自定义管道

命令安装

ng g pipe pipe/multiple

**import** { Pipe, PipeTransform } **from** '@angular/core';  
  
@Pipe({  
 name: 'multiple'  
})  
**export class** MultiplePipe **implements** PipeTransform {  
  
 transform(value: **number**, args?: **number**): **any** {  
 **if** (!args) {  
 args = 1;  
 }  
 **return** value \* args;  
 }  
  
}

说明：通过Pipe管道装饰器来装饰管道

value：接收到的值

args：可选参数

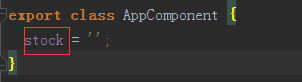
使用：{{size | multiple:4}}

6.组件之间的通讯

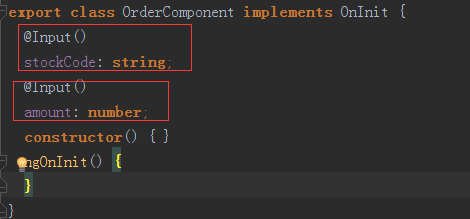
6.1 组件的输入输出

6.1.1 父组件传到子组件（@Input（）装饰器）;

1) 父组件中声明要传递的参数



2) 子组件中通过装饰器@Input（）修饰接收参数属性

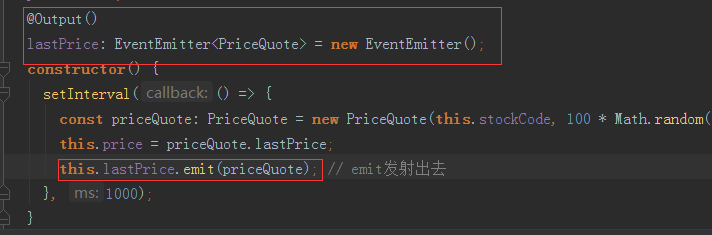


3)父组件调用子组件传递数据

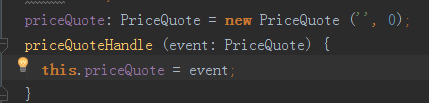
C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\0F`@Q3P_V_@S0K5]RTD40SO.png

6.1.2通过EventEmitter输出到组件外（from '@angular/core'）,并通过Output来注解

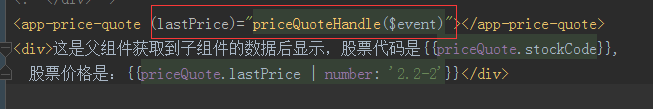
1）子组件中通过Output装饰器并通过EventEmitter来发射数据



2）父组件接收相同类型的数据



3）父组件调用方法，获取数据

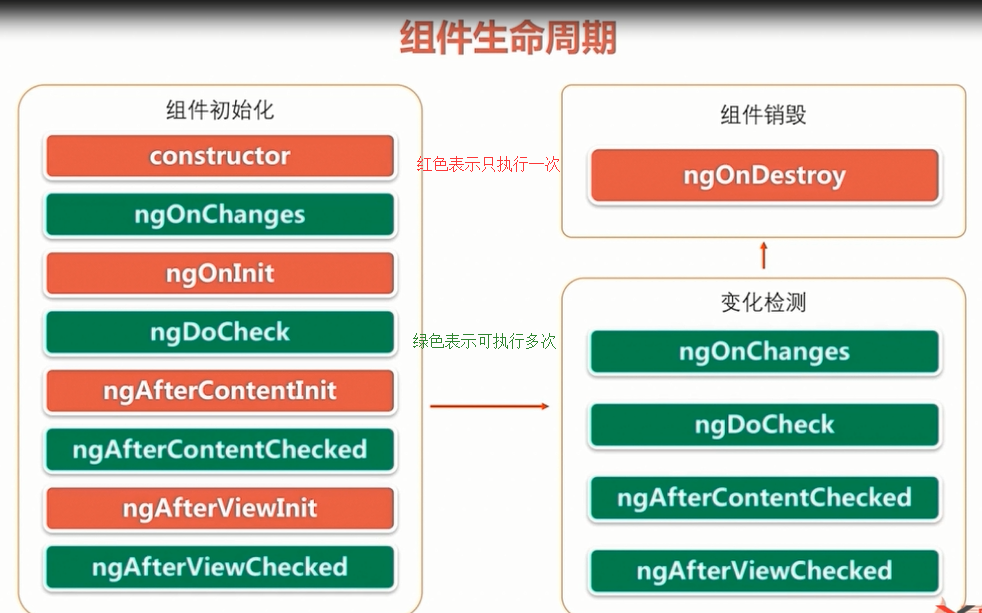


6.2 使用中间人模式传递数据

中间人负责从一个组件接收数据，并将其传递给另一个组件

同时完成以上两步，先将数据从子组件中传给父组件，在由父组件传个另一个子组件

6.3 组件生命周期以及angular的变化发现机制

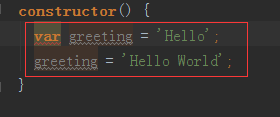


每一个钩子都是'@angular/core'库里定义的一个接口,每一个接口都有一个唯一的钩子方法,他们的名字是由接口的名字加上‘ng’前缀构成的。

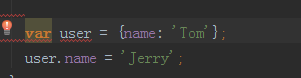
1）ngOnChanges:父组件在初始化或修改子组件的输入参数时被调用。父组件初始化子组件的输入属性（通过@Input装饰器注解的属性）之前，子组件的输入属性是没有值的。初始化时会调用ngOnChanges，将值传到子组件。angular的变更检测机制刷新不可变对象（即不可变对象的值改变），会触发ngOnChanges事件。

可变对象和不可变对象

在JavaScript中，字符串是不可变，当一个字符串在内存中被创建，他的值永远不会改变，var greeting = ‘Hello’在内存中创建了一个值为’Hello’的字符串，greeting = ‘Hello World’并不改变‘Hello’字符串的值，而是在内存中创建一个新的值为’Hello World’的字符串。在内存中就有两个字符串，每个字符串都是不可变的。对于greeting变量来说，值是改变了，它指向的内存地址从第一个字符串的内存地址改为指向第二个字符串的内存地址，这是不可变对象。



在JavaScript中对象是可变的，一个对象的实例在内存中被创建以后，它的属性变化了，他也会保存在固定的内存地址中，第一行代码创建了一个对象，对象的name属性指向了内存中一个值为Tom的字符串，当第二行代码被执行后，一个新的值为’Jerry’的字符串在内存中被创建，并且user的name属性指向新的地址，但是user变量本身依然保持在被创建时的内存地址，也就是说改的是user这个内存地址的内容，user所指向的内存地址并没有改变。这就是可变对象。



2）angular的变更检测机制

angular的变更检测机制是有zone.js来实现的，目的是用来保证组件属性的变化和页面的变化是同步的，浏览器中发生的任何异步事件都会触发变更检测（如点击按钮，输入数据，数据从服务器返回了，调用setTimeOut方法等）。

angular实现了两个变更检测策略。Default和OnPush。如果所有的组件都使用Default策略，不管变更发生在那个组件上，zone.js都会检查整个组件树。如果有一个特定的组件声明自己的变更策略为OnPush，n那么只有当真个组件的输入属性发生变化时，zone.js才会检测这个组件及其后代组件。

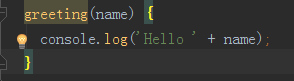


ngDoCheck（其他带有Check的钩子相同）钩子，浏览器中发生的任何异步事件都会触发变更检测。

3）ngAfterViewInit、ngAfterViewChecked

在父组件中调用子组件的代码

子组件中的方法



父组件中通过装饰器@ViewChild();

@ViewChild('child1')  
child: Child2Component;  
**constructor**() {  
}  
ngOnInit(): **void** {  
 **this**.child.greeting('Tom');  
}

C:\Users\Administrator\AppData\Roaming\Tencent\Users\2829122019\QQ\WinTemp\RichOle\A[F%U59@7[`A249SL8RV53G.png

ViewChild中的child1是模板本地变量#child

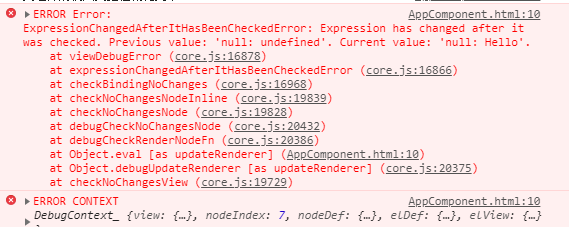
通过模板变量调用

<app-child2 #child2></app-child2>  
<button (click)="child2.greeting('Jerry')">调用child2的方法</button>

ngAfterViewInit、ngAfterViewChecked表示组件的模板所有内容都被组装完成以后，组件模板已经呈现给用户看了，这个动作完成之后，这两个方法会被调用。

在变更检测机制中，angular禁止在一个视图在已经组装好之后再去更新视图。在这两个方法中去更新视图中绑定的属性，会抛出异常（报错）。

ngAfterViewInit(): **void** {  
 console.log('父组件的视图初始化完毕');  
 **this**.message = 'Hello';  
}



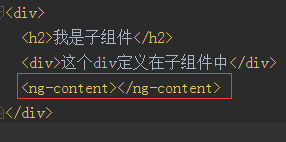
解决：放到另一个时间循环里。

ngAfterViewInit(): **void** {  
 console.log('父组件的视图初始化完毕');  
 setTimeout(() => {  
 **this**.message = 'Hello';  
 }, 0);  
}

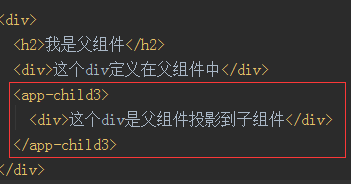
4）Content钩子（AfterContentInit、AfterContentCheck）

投影：在angular里，可以使用ngContent指令将父组件模板的任意片段投影到子组件上。

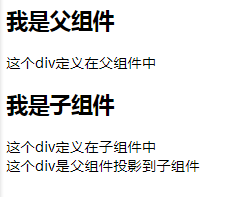
a.在子组件模板上使用ngContent指令来指定一个投影点。



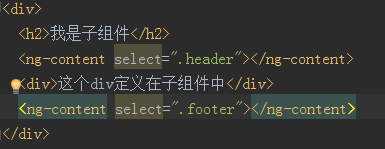
b.在父组件中把要投影到子组件的投影点的html片段写在子组件标签中

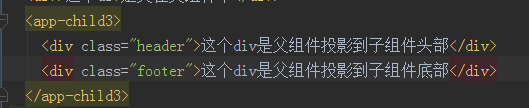


c.效果



注意：一个组件可以声明多个ng-content标签，如下，通过select属性指定对应的class属性





若使用差插值表达式，只能使用父组件中的属性，不能使用子组件中的，它是父组件中的代码，只不过被投影到子组件中。

AfterContentInit、AfterContentCheck 是在被投影的内容组装完成之后调用的。

注意：在AfterContentInit可以改变内容，组件的视图没用组装完毕，只是投影内容组装完成。

5）路由的时候组件被销毁，调用ngOnDestroy

7.angular的表单处理

7.1 模板式表单

表单的数据模型是通过组件模板中的相关指令来定义的，因为使用这种方式定义表单的数据模型时，会受限于HTML的语法，所以，模板驱动方式只适用于一些简单的场景。

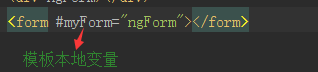
7.1.1使用模板式表单时，只能使用指令来定义模型，可使用的指令有，这些指令都来自FormModule这个模块，需引入该模块。

1） NgForm

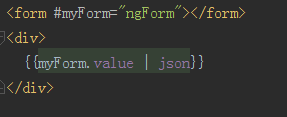
默认情况下，angular自动给form标签挂了一个NgForm指令。

NgForm指令隐式创建一个FormGroup实例，这个类用来代表表单的数据模型并且存储表单的数据。标有NgForm指令的html标签（如form标签）自动发现其所有标有NgModel指令的子元素，并将其值添加到表单的数据模型中。NgForm标签可以在form标签之外的其他标签中使用（如：<div ngForm></div>），如果不希望使用NgForm指令，则明确指明为NgNoForm（<form ngNoForm></form>）。

NgForm指令可以被一个模板本地变量（# + 名称）引用，以便在模板中访问NgForm对象的实例。



引用模板变量



NgForm指令会拦截标准的html表单提交事件，阻止表单的自动提交。因为表单提交会刷新页面。会用一个自定义的ngSubmit的事件来代替，并将表单的值传给自定义事件。

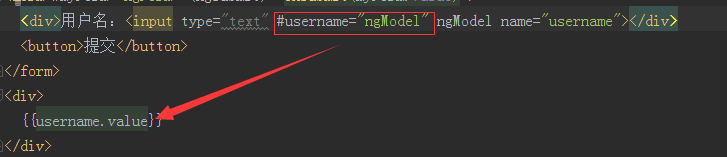
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm.value)"></form>

2） NgModel

在angular中，NgModel指令代表表单的一个字段，这个指令会隐式的创建一个FormControl的实例来代表字段的数据模型，并用这个FormControl类型的对象来存储字段的值。在form里使用NgModel不需要使用[]或（）包起来，直接使用即可，但是需要为使用了NgModel指令的标签指定一个name属性。

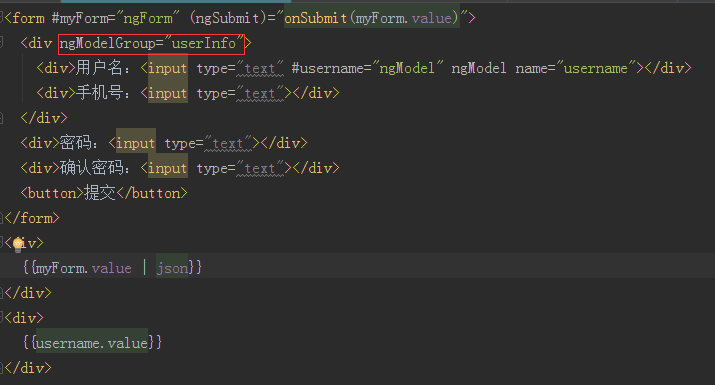
<input type="text" ngModel name="username">

与NgForm一样，NgModel指令创建的对象也可以通过一个模板变量来引用，并通过模板变量的value属性来访问字段的值。

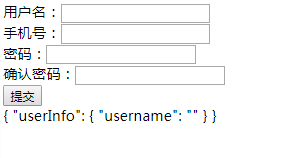


3） NgModelGroup

NgModelGroup代表的是表单的一部分，允许将一些表单字段组织在一起，形成更清晰的层次关系。NgModelGroup也会创建一个FormGroup类的一个实例。这个实例会在NgForm对象中的value属性中表现为一个嵌套的对象，NgModelGroup的子属性都会表现为这个嵌套对象的子属性。



表现为：



7.2 响应式表单

使用响应式表单，通过编写，typescript代码而不是Html代码来创建一个底层数据模型，在这个模型定义好之后，可以使用一些特定的指令，将模板上的html元素于与底层的数据模型连接在一起。

创建一个响应式表单需要两步，首先需要编码来创建一个数据模型，然后需要使用一些指令将模板中的html元素连接到这个数据模型上。

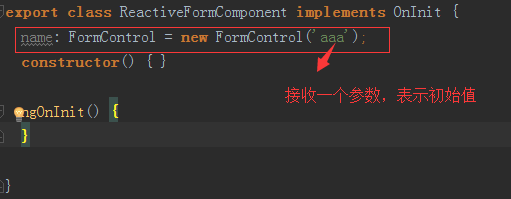
7.2.1如何创建一个数据模型

数据模型是指一个用来保存表单数据的的数据结构，简称模型。由定义在angular模型中的三个类组成。

1） FormControl

FormControl是构成表单的基本单位，通常情况下用来代表一个input元素，也可以用来代表一个更复杂的ui组件，如日历或者下拉选择快。

FormControl保存着与其关联的html元素当前的值以及元素的校验状态，还有元素是否被修改过的信息。



2） FormGroup

FormGroup即可代表表单的一部分，也可以代表整个表单，它是多个FormControl的集合。FormGroup将多个FormControl的值和状态聚合在一起，如果其中一个FormControl是无效的，那么整个FormGroup就是无效的。

formModel: FormGroup = **new** FormGroup ({  
 from: **new** FormControl(),  
 to: **new** FormControl()  
});

3) FormArray

FormArray和FormGroup类似，但是多了一个长度属性。FormArray通常用来代表一个可以增长的字段集合。FormGroup用来代表整个表单或者表单字段的一个固定的子集。

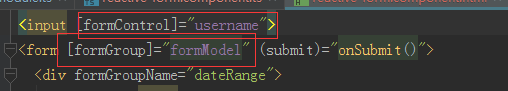
emails: FormArray = **new** FormArray ([  
 **new** FormControl('a@a.com'),  
 **new** FormControl('b@b.com'),  
]);

7.2.2 响应式表单指令

1）FormGroup类的指令：formGroup、formGroupName

2）FormControl类的指令：formControl、formControlName

formControl不能用在FormGroup的内部。



3）FormArray类的指令：formArrayName

4）使用

formModel: FormGroup = **new** FormGroup ({  
 username: **new** FormControl('aaa'),  
 dateRange: **new** FormGroup({  
 from: **new** FormControl(),  
 to: **new** FormControl()  
 }),  
 emails: **new** FormArray ([  
 **new** FormControl('a@a.com'),  
 **new** FormControl('b@b.com'),  
 ])  
});

<!--<input [formControl]="username">-->  
<form [formGroup]="formModel" (submit)="onSubmit()">  
 <input formControlName="username">  
  
 <div formGroupName="dateRange">  
 起始日期: <input type="date" formControlName="from">  
 截止日期：<input type="date" formControlName="to">  
 </div>  
 <div>  
 <ul formArrayName="emails">  
 <li \*ngFor="let e of this.formModel.get('emails').controls;let i=index">  
 <input type="text" [formControlName]="i" >  
 </li>  
 </ul>  
 <button type="button" (click)="addEmail()">增加email</button>  
 </div>  
 <div>  
 <button type="submit">保存</button>  
 </div>  
</form>

7.2.3响应式表单重构

通过注入FormBuilder来重构表单数据模型

7.3模板式表单和响应式表单的异同

不管是那种表单，都会有一个对应的数据模型来存储表单的数据，在模板式表单中，数据模型是由angular基于你组件模板中的指令隐式创建的。而在响应式模板中，可通过编码明确的创建数据模型然后将模板上的html元素与底层的数据模型连接在一起。

数据模型并不是一个任意的对象，它是一个由angular/forms模块中的一些特定的类，如FormControl、FormGroup、FormArray等组成的。在模板式表单中，你是不能直接访问到这些类的。

响应式表单并不会替你生成HTML，模板任然需要自己来写。

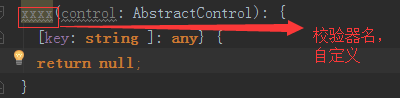
响应式表单的指令一般都以form开头。

响应式表单form开头的指令是不可引用的，不能声明一个模板本地变量来引用该指令的一个实例。

7.4表单验证

7.4.1angular的校验器

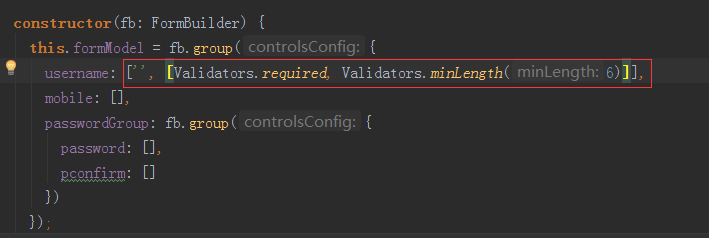
定义校验器格式：



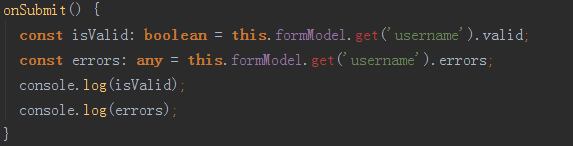
7.4.2校验响应式表单

angular已经定义好的校验器

定义在Validators类中的属性，可直接调用



提交时获取验证信息



自定义校验器：

mobileValidator(control: FormControl): **any** {  
 **var** myreg = /^(((13[0-9]{1})|(15[0-9]{1})|(18[0-9]{1}))+\d{8})$/;  
 **let** valid = myreg.test(control.value);  
 console.log('mobile的校验结果是：' + valid);  
 **return** valid ? **null** : {mobile: **true**} ;  
}

调用：

mobile: ['', **this**.mobileValidator],

状态字段：

touched和untouched

touched和untouched用来判断用户是否访问过一个字段，也就是字段是否获取过焦点。如果获取过，touched为true，untouched为false，反之没获取过焦点touched是false，untouched是true。

pridtine和dirty

如果一个字段的值没被改变过，pridtine就是true，dirty是false，反之字段的值被修改过，pridtine为false，dirty为true

pending

pending是指当一个字段处于异步校验时，字段的pending属性为true，这是可以显示一段文字或图片让用户知道正在校验。

注意：可以给不同状态的表单添加自定义样式。

7.4.3校验模板式表单

模板式表单只能通过指令来校验，因此，需要将校验器方法包装成一个指令，然后才能在模板中使用。在模板式表单校验中不能直接使用状态字段属性。可以通过事件绑定的方式使用。

8.与服务器通讯

8.1 创建web服务器

8.1.1使用node.js创建服务器

创建serve文件夹

初始化文件夹npm init –y

安装typescript类型定义文件：npm i @types/node –save

node不认typescript，需要将typescript编译成JavaScript，通过在配置文件tsconfig.json中编写相关代码来告诉如何将typescript编译成JavaScript

配置IDE,使其知道要用新建的tsconfig.json来配置，webstorm中点击设置，搜索typescript，语言和框架里有个typescript，点击进行配置

创建ts文件，在里面编写服务器相关代码

**import** \* **as** http **from** 'http';  
**const** server = http.createServer((request, response) => {  
 response.end('Hello node!')  
});  
server.listen(8000);

启动服务器 node build/hello\_serve.js

减少手工开发，安装express：npm install express --save

安装其类型定义文件：npm install @types/express --save

自动重启node服务器并加载最新的代码：npm install -g nodemon

通过nodemon启动服务器

nodemon build/hello\_serve.js

8.1.2 使用Express创建restful的Http服务

8.1.3 监控服务器的文件变化

8.2 Http通讯

8.3 websocket通讯