# #### 1，ng-if 跟 ng-show/hide 的区别？

第一点区别是，ng-if 在后面表达式为 true 的时候才创建这个 dom 节点，ng-show 是初始时就创建了，用 display:block 和 display:none 来控制显示和不显示。

第二点区别是，ng-if 会（隐式地）产生新作用域，ng-switch 、 ng-include 等会动态创建一块界面的也是如此。

这样会导致，在 ng-if 中用基本变量绑定 ng-model，并在外层 div 中把此 model 绑定给另一个显示区域，内层改变时，外层不会同步改变，因为此时已经是两个变量了。

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

<div ng-if="true">

<input type="text" ng-model="name">

</div>

ng-show 不存在此问题，因为它不自带一级作用域。

避免这类问题出现的办法是，始终将页面中的元素绑定到对象的属性（data.x）而不是直接绑定到基本变量（x）上。

# #### 2，ng-repeat迭代数组的时候，如果数组中有相同值，会有什么问题，如何解决？

会提示 Duplicates in a repeater are not allowed. 加 track by $index 可解决。当然，也可以 trace by 任何一个普通的值，只要能唯一性标识数组中的每一项即可（建立 dom 和数据之间的关联）。

# #### 3，ng-click 中写的表达式，能使用 JS 原生对象上的方法吗？

不止是 ng-click 中的表达式，只要是在页面中，都不能直接调用原生的 JS 方法，因为这些并不存在于与页面对应的 Controller 的 $scope 中。

# #### 4、factory、service 和 provider 是什么关系？

factory：把 service 的方法和数据放在一个对象里，并返回这个对象

service：通过构造函数方式创建 service，返回一个实例化对象

provider：创建一个可通过 config 配置的 service，$get 中返回的，就是用 factory 创建 service 的内容

从底层实现上来看，service 调用了 factory，返回其实例；factory 调用了 provider，返回其 $get 中定义的内容。factory 和 service 功能类似，只不过 factory 是普通 function，可以返回任何东西（return 的都可以被访问，所以那些私有变量怎么写，你懂的）；

service 是构造器，可以不返回（绑定到 this 的都可以被访问）；

provider 是加强版 factory，返回一个可配置的 factory。

# #### 5、angular 中控制器之间如何通信？

1、Service

2、events,指定绑定的事件

3、使用 $rootScope

4、controller之间直接使用$parent, $$childHead等

5、directive 指定属性进行数据绑定

# #### 6，angular 的数据绑定采用什么机制？详述原理

使用的脏检查机制，所谓的双向绑定，其实就是从界面的操作能实时反映到数据，数据的变更能实时展现到界面。

AngularJS在$scope变量中使用脏值检查来实现了数据双向绑定，并且可以通过$scope.$watch来监听变化触发回调；

angular中使用的是脏检查机制，在angular中每次你绑定一些东西到你的UI上时你就会往$watch队列里插入一条$watch，当我们的模版加载完毕时，也就是在linking阶段（Angular分为compile阶段和linking阶段—译者注），Angular解释器会寻找每个directive，然后生成每个需要的$watch。当浏览器接受到可以被angular context处理的事件时就会触发digest循环，这个循环是由两个更小的循环组合起来的，一个是$watch列表，一个是$evalAsync列表，而$watch列表在$digest循环中被“脏值检查”解析，在digest将会遍历我们的watch，然后询问它是否有属性和值的变化，直到$watch队列都检查过，在检查数据变化的时候，由于并不知道这个事件是对哪些数据进行了更改，以及这个事件有可能造成事件之外的其他任何地方的数据更改，所以必须进行一次大检查，将所有“注册”过的值全部检查一遍，一次检查称为一个周期，每次最少检查两遍，因为第二遍用来确认，前一遍的变动中是否有数据的变动，导致了其他数据的变动，如果第二次有变动的话，会再执行一遍，直到最后两次完全一致，则停止检查（其实就是个（递归（遍历））的过程），考虑到内存的消耗和死循环的风险，脏检查每个周期最多递归执行10遍，如果超过10遍就会抛出一个错误。当$digest循环结束时，DOM相应地变化。

在angular中

ng-click，ng-change，ng-blur...就是对各类用户事件的封装

$timeout，$http，$window，$location...就是对各种JS/API事件的封装

ng-model，以及控制器中的数据，就是对值的“注册”

$scope 本质是一个总的事件逻辑的封装容器，同时抽象为数据载体，实质上数据都存在于浏览器堆内存中

$scope.apply() & $scope.digest() 即Angular中的“数据大检查”的function

所以如果我们使用了非Angular封装的事件改编数据时，要手动执行一次大检查

由于Angular这种脏检查的方法效率不高，如果一个页面绑定的view超过2000个，就可能存在比较明显的性能问题，官方称之为“脏检查”

举个例子

<button ng-click="val=val+1">increase 1</button>

1

click 时会产生一次更新的操作（至少触发两次 $digest 循环）

按下按钮浏览器接收到一个事件，进入到angular context

$digest 循环开始执行，查询每个 $watch 是否变化

由于监视$scope.val 的 $watch 报告了变化，因此强制再执行一次 $digest 循环 新的 $digest 循环未检测到变化

浏览器拿回控制器，更新 $scope.val 新值对应的 dom

$digest 循环的上限是 10 次（超过 10次后抛出一个异常，防止无限循环）。

# #### 7、 angular 应用分层？

目录结构的划分

对于小型项目，可以按照文件类型组织，比如：

css

js

controllers

models

services

filters

templates

但是对于规模较大的项目，最好按业务模块划分，比如：

css

modules

account

controllers

models

services

filters

templates

disk

controllers

models

services

filters

templates

6

modules 下最好再有一个 common 目录来存放公共的东西。

逻辑代码的拆分

作为一个 MVVM 框架，Angular 应用本身就应该按照 模型，视图模型（控制器），视图来划分。

这里逻辑代码的拆分，主要是指尽量让 controller 这一层很薄。提取共用的逻辑到 service 中 （比如后台数据的请求，数据的共享和缓存，基于事件的模块间通信等），提取共用的界面操作到 directive 中（比如将日期选择、分页等封装成组件等），提取共用的格式化操作到 filter 中等等。

在复杂的应用中，也可以为实体建立对应的构造函数，比如硬盘（Disk）模块，可能有列表、新建、详情这样几个视图，并分别对应的有 controller，那么可以建一个 Disk 构造函数，里面完成数据的增删改查和验证操作，有跟 Disk 相关的 controller，就注入 Disk 构造器并生成一个实例，这个实例就具备了增删改查和验证方法。这样既层次分明，又实现了复用（让 controller 层更薄了）。

# #### 8、angular 应用常用哪些路由库，各自的区别是什么？

Angular1.x 中常用 ngRoute 和 ui.router，还有一种为 Angular2 设计的 new router（面向组件）。后面那个没在实际项目中用过，就不讲了。

无论是 ngRoute 还是 ui.router，作为框架额外的附加功能，都必须以 模块依赖 的形式被引入。

区别

ngRoute 模块是 Angular 自带的路由模块，而 ui.router 模块是基于 ngRoute模块开发的第三方模块。

ui.router 是基于 state （状态）的， ngRoute 是基于 url 的，ui.router模块具有更强大的功能，主要体现在视图的嵌套方面。

使用 ui.router 能够定义有明确父子关系的路由，并通过 ui-view 指令将子路由模版插入到父路由模板的 <div ui-view></div>中去，从而实现视图嵌套。而在 ngRoute 中不能这样定义，如果同时在父子视图中 使用了 <div ng-view></div>会陷入死循环。

分属不同团队进行开发的 angular 应用，如果要做整合，可能会遇到哪些问题，如何解决？

可能会遇到不同模块之间的冲突。

比如一个团队所有的开发在 moduleA 下进行，另一团队开发的代码在 moduleB 下

angular.module('myApp.moduleA', [])

.factory('serviceA', function(){

...

})

angular.module('myApp.moduleB', [])

.factory('serviceA', function(){

...

})

angular.module('myApp', ['myApp.moduleA', 'myApp.moduleB'])

会导致两个 module 下面的 serviceA 发生了覆盖。

貌似在 Angular1.x 中并没有很好的解决办法，所以最好在前期进行统一规划，做好约定，严格按照约定开发，每个开发人员只写特定区块代码。

# #### 9、angular 的缺点有哪些？

强约束

导致学习成本较高，对前端不友好。

但遵守 AngularJS 的约定时，生产力会很高，对 Java 程序员友好。

不利于 SEO

因为所有内容都是动态获取并渲染生成的，搜索引擎没法爬取。

一种解决办法是，对于正常用户的访问，服务器响应 AngularJS 应用的内容；对于搜索引擎的访问，则响应专门针对 SEO 的HTML页面。

性能问题

作为 MVVM 框架，因为实现了数据的双向绑定，对于大数组、复杂对象会存在性能问题。

可以用来 优化 Angular 应用的性能 的办法：

减少监控项（比如对不会变化的数据采用单向绑定）

主动设置索引（指定 track by，简单类型默认用自身当索引，对象默认使用 $$hashKey，比如改为 track by item.id）

降低渲染数据量（比如分页，或者每次取一小部分数据，根据需要再取）

数据扁平化（比如对于树状结构，使用扁平化结构，构建一个 map 和树状数据，对树操作时，由于跟扁平数据同一引用，树状数据变更会同步到原始的扁平数据）

另外，对于Angular1.x ，存在 脏检查 和 模块机制 的问题。

移动端

可尝试 Ionic，但并不完善。

# #### 10、是$rootScrope以及和$scope的区别？

通俗的说$rootScrope 页面所有$scope的父亲

如何产生$rootScope和$scope吧。

step1:Angular解析ng-app然后在内存中创建$rootScope。

step2:angular会继续解析，找到{{}}表达式，并解析成变量。

step3:接着会解析带有ng-controller的div然后指向到某个controller函数。这个时候在这个controller函数变成一个$scope对象实例。

# #### 11、如何取消 $timeout, 以及停止一个$watch()? \*\*

停止 $timeout我们可以用cancel：

var customTimeout = $timeout(function () {

// your code

}, 1000);

$timeout.cancel(customTimeout);

停掉一个$watch：

// .$watch() 会返回一个停止注册的函数

function that we store to a variable

var deregisterWatchFn = $rootScope.$watch(‘someGloballyAvailableProperty', function (newVal) {

if (newVal) {

// we invoke that deregistration function, to disable the watch

deregisterWatchFn();

...

}

});

# #### 11、Angular Directive中restrict 中分别可以怎样设置？scope中@,=,&有什么区别？

restrict中可以分别设置:

A匹配属性

E匹配标签

C匹配class

M 匹配注释

当然你可以设置多个值比如AEC,进行多个匹配。

在scope中，@,=,&在进行值绑定时分别表示

@获取一个设置的字符串，它可以自己设置的也可以使用{{yourModel}}进行绑定的;

= 双向绑定，绑定scope上的一些属性；

&用于执行父级scope上的一些表达式，常见我们设置一些需要执行的函数

# #### 15、$apply()和 $digest()的区别

安全性：$apply()可以接收一个参数作为function()，这个 function 会被包装到一个 try … catch 块中，所以一旦有异常发生，该异常会被 $exceptionHandler service 处理。

$apply会使ng进入 $digest cycle , 并从$rootScope开始遍历(深度优先)检查数据变更。

$digest仅会检查该scope和它的子scope，当你确定当前操作仅影响它们时，用$digest可以稍微提升性能。

# #### 16、内置 filter

ng 内置的 filter 有九种：

date（日期）

currency（货币）

limitTo（限制数组或字符串长度）

orderBy（排序）

lowercase（小写）

uppercase（大写）

number（格式化数字，加上千位分隔符，并接收参数限定小数点位数）

filter（处理一个数组，过滤出含有某个子串的元素）

json（格式化 json 对象）

filter 有两种使用方法，一种是直接在页面里：

<p>{{now | date : 'yyyy-MM-dd'}}</p>

另一种是在 js 里面用：

// $filter('过滤器名称')(需要过滤的对象, 参数1, 参数2,...)

$filter('date')(now, 'yyyy-MM-dd hh:mm:ss');

自定义 filter

// 形式

app.filter('过滤器名称',function(){

return function(需要过滤的对象,过滤器参数1,过滤器参数2,...){

//...做一些事情

return 处理后的对象;

}

});

// 栗子

app.filter('timesFilter', function(){

return function(item, times){

var result = '';

for(var i = 0; i < times; i++){

result += item;

}

return result;

}

})

# #### 17、angular 的数据绑定采用什么机制？详述原理

脏检查机制。

双向数据绑定是 AngularJS 的核心机制之一。当 view 中有任何数据变化时，会更新到 model ，当 model 中数据有变化时，view 也会同步更新，显然，这需要一个监控。

原理就是，Angular 在 scope 模型上设置了一个 监听队列，用来监听数据变化并更新 view 。每次绑定一个东西到 view 上时 AngularJS 就会往 $watch 队列里插入一条 $watch，用来检测它监视的 model 里是否有变化的东西。当浏览器接收到可以被 angular context 处理的事件时，$digest 循环就会触发，遍历所有的 $watch，最后更新 dom。

举个栗子

<button ng-click="val=val+1">increase 1</button>

click 时会产生一次更新的操作（至少触发两次 $digest 循环）

按下按钮

浏览器接收到一个事件，进入到 angular context

$digest 循环开始执行，查询每个 $watch 是否变化

由于监视 $scope.val 的 $watch 报告了变化，因此强制再执行一次 $digest 循环

新的 $digest 循环未检测到变化

浏览器拿回控制器，更新 $scope.val 新值对应的 dom

$digest 循环的上限是 10 次（超过 10次后抛出一个异常，防止无限循环）。

详见 关于 AngularJS 的数据绑定

# #### 18、两个平级界面块（通信） a 和 b，如果 a 中触发一个事件，有哪些方式能让 b 知道？详述原理，

这个问题换一种说法就是，如何在平级界面模块间进行通信。有两种方法，

一种是共用服务，一种是基于事件。

共用服务

在 Angular 中，通过 factory 可以生成一个单例对象，在需要通信的模块 a 和 b 中注入这个对象即可。

基于事件

这个又分两种方式

第一种是借助父 controller。在子 controller 中向父 controller 触发（$emit）一个事件，然后在父 controller 中监听（$on）事件，再广播（$broadcast）给子 controller ，这样通过事件携带的参数，实现了数据经过父 controller，在同级 controller 之间传播。

第二种是借助 $rootScope。每个 Angular 应用默认有一个根作用域 $rootScope， 根作用域位于最顶层，从它往下挂着各级作用域。所以，如果子控制器直接使用 $rootScope 广播和接收事件，那么就可实现同级之间的通信。

# #### 22、如何看待 angular 1.2 中引入的 controller as 语法？

最根本的好处

在 angular 1.2 以前，在 view 上的任何绑定都是直接绑定在 $scope 上的

function myCtrl($scope){

$scope.a = 'aaa';

$scope.foo = function(){

...

}

}

使用 controllerAs，不需要再注入 $scope，controller 变成了一个很简单的 javascript 对象（POJO），一个更纯粹的 ViewModel。

function myCtrl(){

// 使用 vm 捕获 this 可避免内部的函数在使用 this 时导致上下文改变

var vm = this;

vm.a = 'aaa';

}

原理

从源码实现上来看，controllerAs 语法只是把 controller 这个对象的实例用 as 别名在 $scope 上创建了一个属性。

if (directive.controllerAs) {

locals.$scope[directive.controllerAs] = controllerInstance;

}

但是这样做，除了上面提到的使 controller 更加 POJO 外，还可以避免遇到 AngularJS 作用域相关的一个坑（就是上文中 ng-if 产生一级作用域的坑，其实也是 javascript 原型链继承中值类型继承的坑。因为使用 controllerAs 的话 view 上所有字段都绑定在一个引用的属性上，比如 vm.xx，所以坑不再存在）。

<div ng-controller="TestCtrl as vm">

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

<div ng-if="vm.name">

<input type="text" ng-model="vm.name">

</div>

</div>

问题

使用 controllerAs 会遇到的一个问题是，因为没有注入 $scope，导致 $emit、 $broadcast、 $on、 $watch 等 $scope 下的方法无法使用。这些跟事件相关的操作可以封装起来统一处理，或者在单个 controller 中引入 $scope，特殊对待。

参考 angular controller as syntax vs scope

# #### 24、详述 angular 的 “依赖注入”

栗子

依赖注入是一种软件设计模式，目的是处理代码之间的依赖关系，减少组件间的耦合。

举个栗子，如果没有使用 AngularJS，想从后台查询数据并在前端显示，可能需要这样做：

var animalBox = document.querySelector('.animal-box');

var httpRequest = {

get: function(url, callback){

console.log(url + ' requested');

var animals = ['cat', 'dog', 'rabbit'];

callback(animals);

}

}

var render = function(el, http){

http.get('/api/animals', function(animals){

el.innerHTML = animals;

})

}

render(httpRequest, animalBox);

但是，如果在调用 render 的时候不传参数，像下面这样，会报错，因为找不到 el 和 http（定义的时候依赖了，运行的时候不会自动查找依赖项）

render();

// TypeError: Cannot read property 'get' of undefined

而使用 AngularJS，可以直接这样

function myCtrl = ($scope, $http){

$http.get('/api/animals').success(function(data){

$scope.animals = data;

})

}

也就是说，在 Angular App 运行的时候，调用 myCtrl，自动做了 $scope 和 $http 两个依赖性的注入。

原理

AngularJS 是通过构造函数的参数名字来推断依赖服务名称的，通过 toString() 来找到这个定义的 function 对应的字符串，然后用正则解析出其中的参数（依赖项），再去依赖映射中取到对应的依赖，实例化之后传入。

简化一下，大概是这样：

var inject = {

// 存储依赖映射关系

storage: {},

// 注册依赖

register: function(name, resource){

this.storage[name] = resource;

},

// 解析出依赖并调用

resolve: function(target){

var self = this;

var FN\_ARGS = /^function\s\*[^\(]\*\(\s\*([^\)]\*)\)/m;

var STRIP\_COMMENTS = /((\/\/.\*$)|(\/\\*[\s\S]\*?\\*\/))/mg;

fnText = target.toString().replace(STRIP\_COMMENTS, '');

argDecl = fnText.match(FN\_ARGS)[1].split(/, ?/g);

var args = [];

argDecl.forEach(function(arg){

if(self.storage[arg]){

args.push(self.storage[arg]);

}

})

return function(){

target.apply({}, args);

}

}

}

使用这个 injector，前面那个不用 AngularJS 的栗子这样改造一下就可以调用了

inject.register('el', animalBox);

inject.register('ajax', httpRequest);

render = inject.resolve(render);

render();

问题

因为 AngularJS 的 injector 是假设函数的参数名就是依赖的名字，然后去查找依赖项，那如果按前面栗子中那样注入依赖，代码压缩后（参数被重命名了），就无法查找到依赖项了。

// 压缩前

function myCtrl = ($scope, $http){

...

}

// 压缩后

function myCtrl = (a, b){

...

}

所以，通常会使用下面两种方式注入依赖（对依赖添加的顺序有要求）。

数组注释法

myApp.controller('myCtrl', ['$scope', '$http', function($scope, $http){

...

}])

显式 $inject

myApp.controller('myCtrl', myCtrl);

function myCtrl = ($scope, $http){

...

}

myCtrl.$inject = ['$scope', '$http'];

补充

对于一个 DI 容器，必须具备三个要素：依赖项的注册，依赖关系的声明和对象的获取。

在 AngularJS 中，module 和 $provide 都可以提供依赖项的注册；内置的 injector 可以获取对象（自动完成依赖注入）；依赖关系的声明，就是前面问题中提到的那样。

下面是个栗子

// 对于 module，传递参数不止一个，代表新建模块，空数组代表不依赖其他模块

// 只有一个参数（模块名），代表获取模块

// 定义 myApp，添加 myApp.services 为其依赖项

angular.module('myApp', ['myApp.services']);

// 定义一个 services module，将 services 都注册在这个 module 下面

angular.module('myApp.services', [])

// $provider 有 factory, service, provider, value, constant

// 定义一个 HttpService

angular.module('myApp.services').service('HttpService', ['$http', function($http){

...

}])

参考

# #### 25、[AngularJS] 自己实现一个简单的依赖注入

理解angular中的module和injector，即依赖注入

AngularJS中的依赖注入实际应用场景

# #### 26、如何看待angular2

相比 Angular1.x，Angular2的改动很大，几乎算是一个全新的框架。

基于 TypeScript（可以使用 TypeScript 进行开发），在大型项目团队协作时，强语言类型更有利。

组件化，提升开发和维护的效率。

还有 module 支持动态加载，new router，promise的原生支持等等。

迎合未来标准，吸纳其他框架的优点，值得期待，不过同时要学习的东西也更多了（ES next、TS、Rx等）。

**#### 27、angular路由守卫**

     路由守卫是指当用户满足了某些要求之后才可以离开或者进入某个页面或者场景的时候使用。比如说只有当用户填写了用户名和密码之后才可以进入首页，比如说用户离开某个页面时明月保存信息提示用户是否保存信息后再离开等操作，控制这些要求的就叫路由守卫。

 1.CanActivate 进入路由守卫（通俗来说：创建一个守卫，将要求写入守卫里面，将守卫注入到使用的页面，在路由里面引用这个守卫，）

这种守卫是指用户必须满足某些要求后才可以进入路由。比如注册登录

1.首先建立一个守卫文件guard，在文件里面建一个守卫guard.ts

2.在guard.ts里面定义一个类，这个类实现一个接口CanActivate：这个接口只有一个方法就是canActivate(){}

这个方法返回一个布尔值true或者是false,来决定是否进入路由；

import {CanActivate} from "@angular/router";

export class LoginGard implements CanActivate{

canActivate(){

let loggedIn:boolean=Math.random()<0.5;

if(!loggedIn){

console.log("用户未登录");

}

return loggedIn;

}

}

3.接下来去配置路由：在配置路由的时候呢在路由后面加一个canActivate属性，这个属性的值是一个数组，可以是多个值，这里的多个值是所有路由的守卫，也就是说该路由需要满足的路由守卫有哪些，路由会依次调用这些守卫，只有当满足所有守卫的要求时路由才有效，也就是所有守卫的返回值都为true的时候才可以同过路由守卫。

4.路由配置好了之后需要利用依赖注入来实现这个类。

import {LoginGard} from "./gard/login.gard";

const routes:Routes=[

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

{path:'chat',component:XhatComponent,outlet:'aux'},

{path:'product',component:ProductComponent,children:[

{path:'',component:ProductdescComponent} ,

{path:'seller/:id',component:SellerComponent}

],canActivate:[LoginGard]},

{path:'home/:id',component:HomeComponent},//整个路径被划分成两段变量，一段是路径，一段时参数

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

];

@NgModule({

imports: [

RouterModule.forRoot(routes)

],

exports:[RouterModule],

providers: [LoginGard]

})

export class AppRoutingModule { }

2.CanDeactivate 离开路由守卫

这种守卫是指用户如果不能满足当前守卫的要求就不能离开此页面

  1，同CanActivate一样首先也要顶一个守卫这个守卫实现一个接口CanDeactivate ，不一样的是这个接口要实现一个泛型，这个泛型就是指定当前组件的类型，即要守护的组件；

 2.这个接口有一个canDeactivate（）的方法，一样要接受一个参数，这个参数是当前保护的组件的参数信息；用这些信息来判断满足什么样的条件方可离开此组件页面

import {CanDeactivate} from "@angular/router";

import {ProductComponent} from "app/product/product.component";

export class UnsaveGuard implements CanDeactivate<ProductComponent>{

canDeactivate(component: ProductComponent){

return window.confirm("您还没有保存确定要离开吗")//如果用户点否就继续留在本页面，点确定才会离开本页面。window.confirm弹出一个弹框

}

}

 3.配置路由，和canActivate一样的配置方式

const routes:Routes=[

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

{path:'chat',component:XhatComponent,outlet:'aux'},

{path:'product',component:ProductComponent,children:[

{path:'',component:ProductdescComponent} ,

{path:'seller/:id',component:SellerComponent}

],canActivate:[LoginGard]，

canDeactivate:[UnsaveGuard]

},

3.Resolve 激活路由守卫（通俗来说就是首先创建一个路由守卫来提前获取组件参数值，其次将参数值再返回给组件视图）；

这种守卫在路由激活前触发，一般用来获取激活路由对应视图组件数据使用。

进入视图前，利用该守卫将视图所需数据加载完成，

可以避免网络不好的时候用户进入视图后网络不好造成数据加载不出来的情况出现，

此路由守卫在加载视图前请求好数据后才进入视图。

同样的1.创建一个路由守卫实现一个接口Resolve，这个接口接受一个泛型，这个泛型

是该守卫要解析出来的数据的结构和类型。

2.实现一个resolve方法，这个方法可以获取路由携带的参数，获取到参数后才可以进

入视图。

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

export class ProductRsolve implements Resolve<Product>{//这个product泛型类需要自己定义一下引入

@Injectable（）//只有加这个装饰器router才可以被注入进来

constructor(private router:Route){

}

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Product | Observable<Product> | Promise<Product> {//这里的ActivatedRouteSnapshot可以直接获取路由里面的参数，因此接下来就可以直接获取参数值

let productId:number=Route.params["id"];//Route.params直接获取参数

if(productId==1){

return new Product(1,"iphnoe7");

}else {

this.router.navigate(["/home"]);//注入router后才可以使用导航

}

}

}

export class Product{

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

}

3.配置路由不一样的是路由里面resolve接收的是一个对象，这个对象的name就是守卫要接受的参数，值就是用哪个守卫来解析这个name。

4.和其他守卫一样也需要注入

const routes:Routes=[

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

{path:'chat',component:XhatComponent,outlet:'aux'},

{path:'product',component:ProductComponent,children:[

{path:'',component:ProductdescComponent} ,

{path:'seller/:id',component:SellerComponent}

],resolve:{

product:ProductRsolve

}

在组件中定义变量接收参数

export class ProductComponent implements OnInit {

routerInfo: any;

private productId:number;//定义两个参数来接收路由传进来的值

private productName:string;

constructor(private routerIonfo:ActivatedRoute) { }

ngOnInit() {

this.productId=this.routerIonfo.snapshot.queryParams["id"];

this.routerInfo.data.subscribe((data:{product:Product})=>{//获取值

this.productId=data.product.id;

this.productName=data.product.name;

});

}

}

export class Product{

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

}

}