# DI依赖注入

2018年11月20日　　8:46

1. 基本概念：

依赖注入：DI 假如你写了一个发法，方法的参数是一个对象，调用这个方法的时候，需要实例化这个对象 ，并把它传递给这个方法。

> 在软件工程中，依赖注入是种实现控制反转用于解决依赖性设计模式。一个依赖关系指的是可被利用的一种对象（即服务提供端）。依赖注入是将所依赖的传递给将使用的从属对象（即客户端）。该服务是将会变成客户端的状态的一部分。传递服务给客户端，而非允许客户端来建立或寻找服务，是本设计模式的基本要求。 —— 维基百科

对于依赖注入还有一个问题就是控制反转(IoC),是说创建对象的控制权发生转移，以前创建对象的主动权和创建时机由应用程序把控，而现在这种权利转交给 IoC 容器，它就是一个专门用来创建对象的工厂，你需要什么对象，它就给你什么对象。有了 IoC 容器，依赖关系就改变了，原先的依赖关系就没了，它们都依赖 IoC容器了，通过 IoC 容器来建立它们之间的关系。

个人理解：

依赖注入：需要实例化

控制反转：只需要注入，自己会去实例化

2. 依赖注入的两个部分：

注入器：constructor(private productService: ProductService) {…}

提供器：提供器有两种方式声明分别为：

单例模式：providers:[ProductService] <==> providers:[{provide:ProductService,useClass:ProductService}]

如果有另一个服务则需要这样定义：providers:[{provide:ProductService,useClass:otherProductService}]

Note：只要provide对应的是同一个值，那么在注入器那就可以找到对应的服务，并按照一定原则去覆盖它

工厂模式：providers:[{provide:ProductService,useFactory:() => {…}]

Note:用useFactory来代表工厂模式，用useClass代表单例模式

3. 例如我们需要一个productService来获取product的信息从而方便在html显示，我们需要去声明各种与product有关的实例来声明product，这样就很繁琐，而对于依赖注入就很简单，下面来看看依赖注入怎么实现的

单例模式声明提供器的方式

a. 生成sever服务 ng g s shared/productService，并在app.module.ts中声明这个提供器

```
@NgModule({
  declarations: [
    AppComponent,
    Product1Component,
    Product2Component
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  // 模块声明
  providers: [ProductService,],
```

当然这种写法也可以改为：providers:[{provide:ProductService,useClass:ProductService}]

b. 在服务模块中编写相应的函数调用

```
import { Injectable } from '@angular/core';
import {LoggerService} from './logger.service';

@Injectable({
  providedIn: 'root'
})
export class ProductService {

  // 编写获取产品信息的函数，从而使product组件调用这个函数获取产品信息
  getProduct(): Product {
    return new Product( id: 0,  title: 'iPhoneX',  price: 9999,  desc: '电子产品');
  }
}

// 定义Product类来作为数据的来源
export class Product {
  constructor(
    public id: number,
    public title: string,
    public price: number,
    public desc: string
  ) {
  }
}
```

c. 在product组件中声明一个属性，然后声明注入器，调用提供器内的方法返回声明的变量到html页面展示

```
export class Product1Component implements OnInit {

  // 声明一个属性变量
  product: Product;

  // 将提供器引入，必须在constructor里引入
  // 格式：constructor(private 变量名: 引入的提供器名) { }
  constructor(private productService: ProductService) { }

  ngOnInit() {
    // 获取返回的product信息，从而掺入html中
```

```
export class Product1Component implements OnInit {

  // 声明一个属性变量
  product: Product;

  // 将提供器引入，必须在constructor里引入
  // 格式：constructor(private 变量名: 引入的提供器名) { }
  constructor(private productService: ProductService) { }

  ngOnInit() {
    // 获取返回的product信息，从而掺入html中
    this.product = this.productService.getProduct();
  }

}
```

d. html文件

```
<div>
  <h1>商品详情</h1>
  <h2>名称：{{product.title}}</h2>
  <h2>价格：{{product.price}}</h2>
  <h2>描述：{{product.desc}}</h2>
</div>
```

4. 另一种单例模式声明提供器的方式：在组件中声明提供器
   a. 依然先生成服务组件：ng g s shared/othrtProductService
   b. 在othrtProductService服务模块中继承之前的ProductService，就省的再重新定义produc模板了

```
import { Injectable } from '@angular/core';
import {Product, ProductService} from './product.service';
import {LoggerService} from './logger.service';

@Injectable({
  providedIn: 'root'
})

// 实现implements，这样就能实现ProductService内的相同方法
export class OtherproductService implements ProductService {

  getProduct(): Product {
    return new Product( id: 1,  title: 'sumsung8',  price: 8888,  desc: '电子产品');
  }
```

   c. 在product组件中更改商品的提供器，此时app.module.ts中依然是providers:[ProductService]这样定义的提供器，但我们要在组件中更改这个提供器换成OtherproductService

```
@Component({
  selector: 'app-product2',
  templateUrl: './product2.component.html',
  styleUrls: ['./product2.component.css'],
  // 更改商品的提供器
  providers: [{
    provide: ProductService, useClass: OtherproductService
  }]
})
// 这个和product1代码一样，只是提供器不同
export class Product2Component implements OnInit {

  product: Product;

  /* private productService: ProductService，冒号前面的productService是个变量名，
  冒号后面的是那个ProductService对应的是 providers中的provide的值，
  而最终把productService这个变量通过provide的token(即ProductService)
  所对应的提供器OtherproductService定义为productService的类型*/
  constructor(private productService: ProductService) { }
  // 这种方式不推荐使用，只是作为一种解释的代码来看注入器是怎么引入的，不方便测试调用之类的问题
  // private productService: ProductService;
  // constructor(private injector: Injector) {
  //   this.productService = injector.get(ProductService);
  // }
  ngOnInit() {
    this.product = this.productService.getProduct();
  }
}
```

   Note: constructor(private productService: ProductService) { }，冒号前面的productService是个变量名，冒号后面的是那个ProductService对应的是 providers中的provide的值,而最终把productService这个变量通过provide的token(即ProductService)所对应的提供器OtherproductService定义为productService的类型

   d. html文件

```
<div>
  <h1>商品详情</h1>
  <h2>名称：{{product.title}}</h2>
  <h2>价格：{{product.price}}</h2>
  <h2>描述：{{product.desc}}</h2>
</div>
```

5. 在提供器注入其他提供器

a. 生成一个logger服务模块：ng g s shared/Logger
然后在app.module.ts中声明提供器

```
@NgModule({
  declarations: [
    AppComponent,
    Product1Component,
    Product2Component
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  // 提供器模块声明
  providers: [ProductService, LoggerService],
```

b. 在logger.service.ts中定义一个函数

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

@Injectable({
  providedIn: 'root'
})
export class LoggerService {

  constructor() { }
  log(message: string) {
    console.log(message);
  }
}
```

c. 将这个提供器镶嵌到之前的ProductService服务器中

```
export class ProductService {

  // 将LoggerService提供器注入到这个提供器中
  constructor(public logger: LoggerService) { }

  // 编写获取产品信息的函数，从而使product组件调用这个函数获取产品信息
  getProduct(): Product {
    // 直接调用LoggerService提供器的方法
    this.logger.log('getProduct方法调用');
    return new Product(0, 'iPhoneX', 9999, '电子产品');
  }
}
```

Note：因为otherProductService是继承ProductService这个提供器的，根据ts的语法要求，子类必须调用父类的构造方法。故也要在otherProductService里进行如下修改，将LoggerService作为父类的构造方法。

```
// 用implements继承，这样就能实现ProductService内的相同方法
export class OtherproductService implements ProductService {

  getProduct(): Product {
    return new Product(1, 'sumsung8', 8888, '电子产品');
  }

  constructor(public logger: LoggerService) { }
}
```

这样就把一个提供器注入到另一个提供器中，如果有被注入的提供器还有注入的提供器，同方式注入

6. 在某些时候服务对象，不是简单new就能满足要求。根据条件决定实例化哪个对象，或者传递参数，这个时候，使用工厂函数指定提供器。根据某个标识即true或者false，来选择注入productService和anohorService的某一个。
工厂模式主要是app.module.ts中的providers的逻辑，更详细的说是providers中的useFactory中的逻辑

```
// 工厂模式实例化ProductService，useFactory就是工厂函数
// 工厂函数调用如果发现需要依赖另一个服务LoggerService，和appConfig，则用deps参数来导入依赖
// 如果LoggerService也需要一个工厂，则继续以这种方式定义直到全部加载完
// appConfig则是定义一个变量，通过deps来加载进依赖中，变量可以是对象或者常量
// 用{provide: 'APP_CONFIG', useValue: {isDEV: false}}来定义，useValue就可以定义是对象或者常量
providers: [{
  provide: ProductService,
  useFactory: (logger: LoggerService, appConfig) => {
    if (appConfig.isDEV) {
      return new ProductService(logger);
    } else {
      return new OtherproductService(logger);
    }
  },
  // 导入上面工厂模式需要的依赖提供器
  deps: [LoggerService, 'APP_CONFIG']
},
  // 这里在定义另一个需要的提供器APP_CONFIG
  {
    // useValue就可以定义是对象或者常量即可以像下面的那样写一个对象
    // 也可以直接写useValue: false
    provide: 'APP_CONFIG', useValue: {isDEV: false}
  }, LoggerService
],
```

这的方式和provide的原理是一样的通过token来定义变量值。即appConfig对应的是'APP_CONGIG'而通过'APP_CONGIG'的token找到所代表的value，并将这个value赋给appConfig

提供器工厂模式注入的前世今生可以访问： https://www.jianshu.com/p/d328f1e9c24b

7. 注入器及层级关系：
a. 在模块(app.module.ts)声明中，对所有组件都可见，都可以注入

b. 在组件(xxx.component.ts)声明时，只对这个组件和其子组件有效，其他的组件无权调用注入
c. 声明在模块(app.module.ts)和组件(xxx.component.ts)中的提供器的token相同时，即定义的providers中的provide相同时，声明在组件的提供器会覆盖掉在模块中生命的提供器，来为本组件调用，但对其他组件无覆盖效果
d. 一般情况下，先声明模块中的提供器，再在组件总声明提供器，只有需要对某些组件不可用时，才在这些组件重定义自己的提供器