link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=179 sentences=330, words=2881

1. Reflect

1.1 Reflect

- Reflect对象与Proxy对象一样，也是 ES6 为了操作对象而提供的新 API
- JS 的装饰器更多的是存在于对函数或者属性进行一些操作，比如修改他们的值，代理变量，自动绑定 this 等等功能
- 但是却无法实现通过反射来获取究竟有哪些装饰器添加到这个类/方法上,于是 Reflect Metadata 应运而生
- Reflect Metadata简单来说，你可以通过装饰器来给类添加一些自定义的信息
- 然后通过反射将这些信息提取出来

```
Reflect.defineMetadata(metadataKey, metadataValue, target);
Reflect.defineMetadata(metadataKey, metadataValue, target, propertyKey);

let result = Reflect.getMetadata(metadataKey, target);
let result = Reflect.getMetadata(metadataKey, target, propertyKey);
```

```
let target = {};
Reflect.defineMetadata("name", "zhufeng", target);
Reflect.defineMetadata("name", "world", target, 'hello');
console.log(Reflect.getOwnMetadata("name", target));
console.log(Reflect.getOwnMetadata("name", target, "hello"));
```

1.2.2 decorator

- 所有的对类的修饰，都是定义在类这个对象上面的
- 而所有的对类的属性或者方法的修饰，都是定义在类的原型上面的，并且以属性或者方法的 key 作为 property

```
@Reflect.metadata(metadataKey, metadataValue)
class C {

  @Reflect.metadata(metadataKey, metadataValue)
  method() {}
}
```

```
import 'reflect-metadata';
let target = {};
Reflect.defineMetadata('name','zhufeng',target);
Reflect.defineMetadata('name', 'world', target,'hello');
console.log(Reflect.getOwnMetadata('name',target));
console.log(Reflect.getOwnMetadata('name', target, 'hello'));
console.dir(target);

function classMetadata(key,value){
     return function(target){
         Reflect.defineMetadata(key, value, target);
     }
}
function methodMetadata(key, value) {

    return function (target,propertyName) {

        Reflect.defineMetadata(key, value, target, propertyName);
    }
}

@classMetadata('name', 'Person')
class Person{

   @methodMetadata('name', 'world')
   hello():string{ return 'world'}
}
console.log(Reflect.getMetadata('name', Person));
console.log(Reflect.getMetadata('name', new Person(),'hello'));
```

1.2 tsconfig.json

1.2.1 tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "declaration": true,
    "removeComments": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es2017",
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": "./",
    "incremental": true
  },
  "exclude": [
    "node_modules",
    "dist"
  ]
}
```

1.2.2 tsconfig.build.json

```
{
    "extends": "./tsconfig.json",
    "exclude": [
        "node_modules",
        "test",
        "dist",
        "**/*spec.ts"
    ]
}
```

2. IOC和DI

2.1 创建电脑

```
export interface Monitor{}
class Monitor27inch implements Monitor{}
interface Host{}
class LegendHost implements Host { }
class Computer{
    monitor:Monitor;
    host:Host;
    constructor(){
        this.monitor = new Monitor27inch();
        this.host = new LegendHost();
    }
    startup(){
        console.log('组装好了,可以开机了');
    }
}
let computer = new Computer();
computer.startup();
```

- 问题
    - 无法传递不同的零件实例
    - 需要自己手工创建零件实例

2.2 可以传递零件

```
interface Monitor{}
class Monitor27inch implements Monitor{}
interface Host{}
class LegendHost implements Host { }
export class Computer{
    monitor:Monitor;
    host:Host;
    constructor(monitor, host){
        this.monitor = monitor;
        this.host = host;
    }
    startup(){
        console.log('组装好了,可以开机了');
    }
}
+let monitor = new Monitor27inch();
+let host = new LegendHost();
+let computer = new Computer(monitor, host);
computer.startup();
```

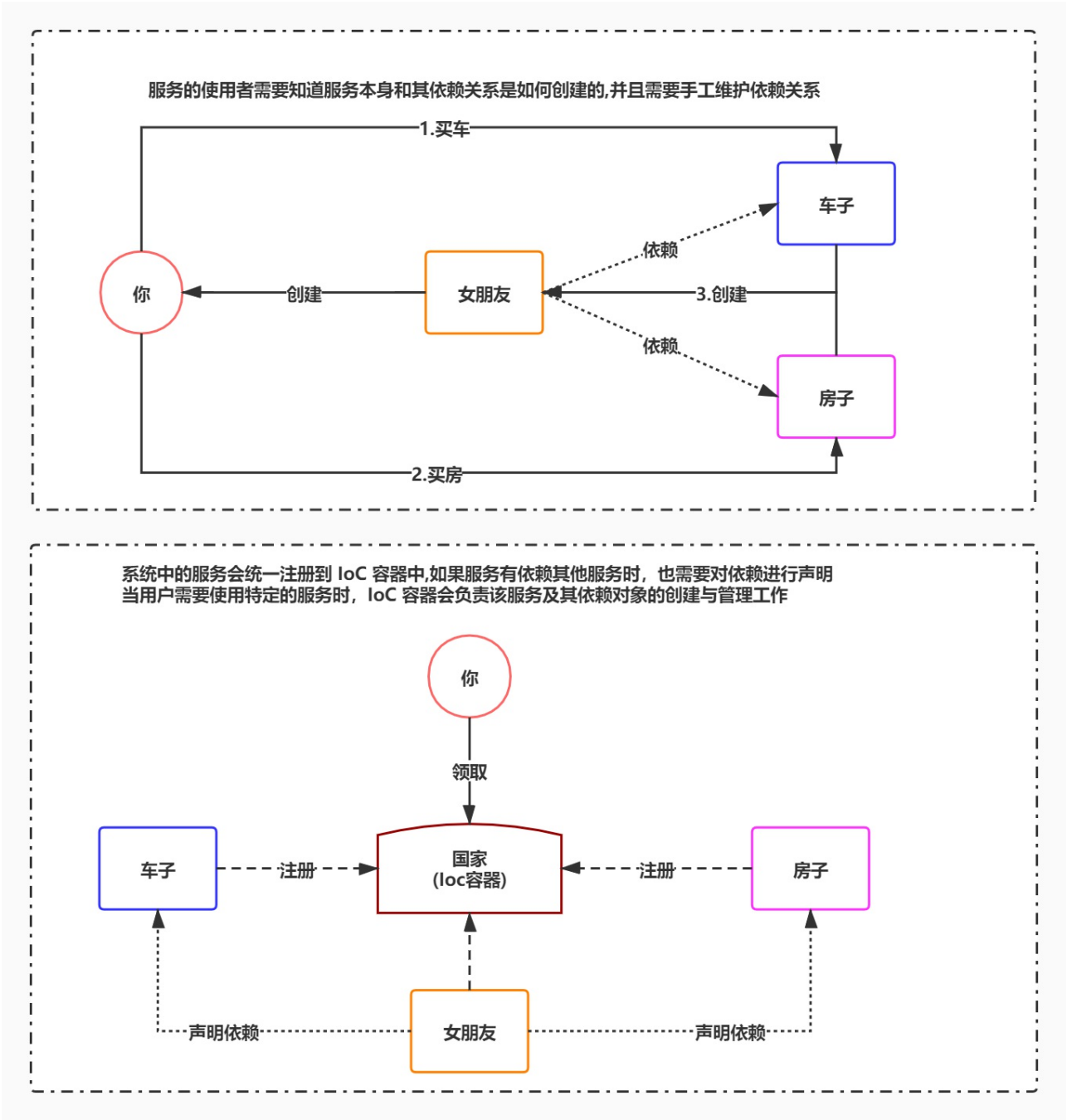
2.3 Ioc和DI

2.3.1 IoC(Inversion of Control)

- IoC(Inversion of Control)即 控制反转。在开发中， IoC 意味着你设计好的对象交给容器控制，而不是使用传统的方式，在对象内部直接控制
- 谁控制谁，控制什么，为何是反转，哪些方面反转了
    - 谁控制谁，控制什么：在传统的程序设计中，我们直接在对象内部通过 new 的方式创建对象，是程序主动创建依赖对象； 而 IoC 是有专门一个容器来创建这些对象，即由 IoC 容器控制对象的创建,谁控制谁？当然是 IoC 容器控制了对象；控制什么？主要是控制外部资源(依赖对象)获取
    - 为何是反转了，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在程序中主动控制去获取依赖对象，也就是正转； 而反转则是由容器来帮忙创建及注入依赖对象,为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转了；哪些方面反转了？依赖对象的获取被反转了
- IoC是一种思想，是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度
- 传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试； 有了 IoC 容器后，把创建和查找依赖对象的控制权交给了容器，由容器注入组合对象，所以对象之间是松散耦合。 这样也便于测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活
- 其实 IoC 对编程带来的最大改变不是从代码上，而是思想上，发生了主从换位的变化。应用程序本来是老大，要获取什么资源都是主动出击，但在 IoC 思想中，应用程序就变成被动了，被动的等待 IoC 容器来创建并注入它所需的资源了

2.3.2 DI(Dependency Injection)

- 对于控制反转来说，其中最常见的方式叫做 依赖注入，简称为 DI（Dependency Injection）
- 组件之间的依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中
- 通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现
- 理解 DI 的关键是 谁依赖了谁，为什么需要依赖，谁注入了谁&#
    - 谁依赖了谁：当然是应用程序依赖 IoC 容器
    - 为什么需要依赖：应用程序需要 IoC 容器来提供对象需要的外部资源（包括对象、资源、常量数据）
    - 谁注入谁：很明显是 IoC 容器注入应用程序依赖的对象；
    - 注入了什么：注入某个对象所需的外部资源（包括对象、资源、常量数据）
- IoC 和 DI 是同一个概念的不同角度描述,依赖注入明确描述了被注入对象依赖 IoC 容器配置依赖对象



3. Nest.js

- NestJS 是构建高效，可扩展的 Node.js Web 应用程序的框架
- 它使用现代的 JavaScript 或 TypeScript（保留与纯 JavaScript 的兼容性），并结合 OOP（面向对象编程），FP（函数式编程）和FRP（函数响应式编程）的元素
- NestJS 旨在提供一个开箱即用的应用程序体系结构，允许轻松创建高度可测试，可扩展，松散耦合且易于维护的应用程序

3.1 安装依赖

```
cnpm i @nestjs/core @nestjs/common @nestjs/platform-express rxjs reflect-metadata -D
```

3.2 实现应用

3.2.1 main.ts

src\main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

3.2.2 app.module.ts

src\app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from "./app.controller";
@Module({
  controllers: [AppController]
})
export class AppModule {}
```

3.2.3 app.controller.ts

src\app.controller.ts

```
import { Get, Controller } from "@nestjs/common";

@Controller('/')
export class AppController {
  @Get('/hello')
  hello() {
    return 'hello';
  }
}
```

3.2.4 package.json

```
  "scripts": {
    "start:dev": "nest start --watch"
  },
```

3.3 依赖注入

3.3.1 src\main.ts

```
import { Module } from '@nestjs/common';
import { AppController } from "./app.controller";
+import { AppService } from "./app.service";
+import { UseClassLoggerService, UseValueLoggerService, UseValueLoggerServiceStringToken, UseFactoryLoggerService } from "./logger.service";

@Module({
  controllers: [AppController],
+  providers: [
    AppService,
+    {
+      provide: UseClassLoggerService,
+      useClass: UseClassLoggerService
+    },
+    {
+      provide: UseValueLoggerService,
+      useValue: new UseValueLoggerService()
+    },
+    {
+      provide: 'StringToken',
+      useValue: new UseValueLoggerServiceStringToken()
+    },
+    {
+      provide: 'FactoryToken',
+      useFactory: () => new UseFactoryLoggerService()
+    }
+  ]
+})
export class AppModule {}
```

3.3.2 app.controller.ts

src\app.controller.ts

```
import { Get, Controller,Inject } from "@nestjs/common";
import { AppService } from "./app.service";
+import { UseClassLoggerService, UseValueLoggerService, UseFactoryLoggerService, UseValueLoggerServiceStringToken } from "./logger.service";
@Controller('/')
export class AppController {
  constructor(
+    private readonly appService: AppService,
+    private readonly useClassLoggerService: UseClassLoggerService,
+    private readonly useValueLoggerService: UseValueLoggerService,
+    @Inject("StringToken") private readonly useValueLoggerServiceStringToken: UseValueLoggerServiceStringToken,
+    @Inject("FactoryToken")  private readonly useFactoryLoggerService: UseFactoryLoggerService
+    ) { }
  @Get('/hello')
  hello() {
+    this.useClassLoggerService.log('useClassLoggerService');
+    this.useValueLoggerService.log('useValueLoggerService');
+    this.useValueLoggerServiceStringToken.log('StringToken');
+    this.useFactoryLoggerService.log('FactoryToken');
    return this.appService.getHello();
  }
}
```

3.3.3 src\app.service.ts

src\app.service.ts

```
import { Injectable } from "@nestjs/common";
+import { UseClassLoggerService } from "./logger.service";
@Injectable()
export class AppService {
+  constructor(private readonly useClassLoggerService: UseClassLoggerService,){}
  getHello(): string {
+    this.useClassLoggerService.log('getHello');
    return "Hello";
  }
}
```

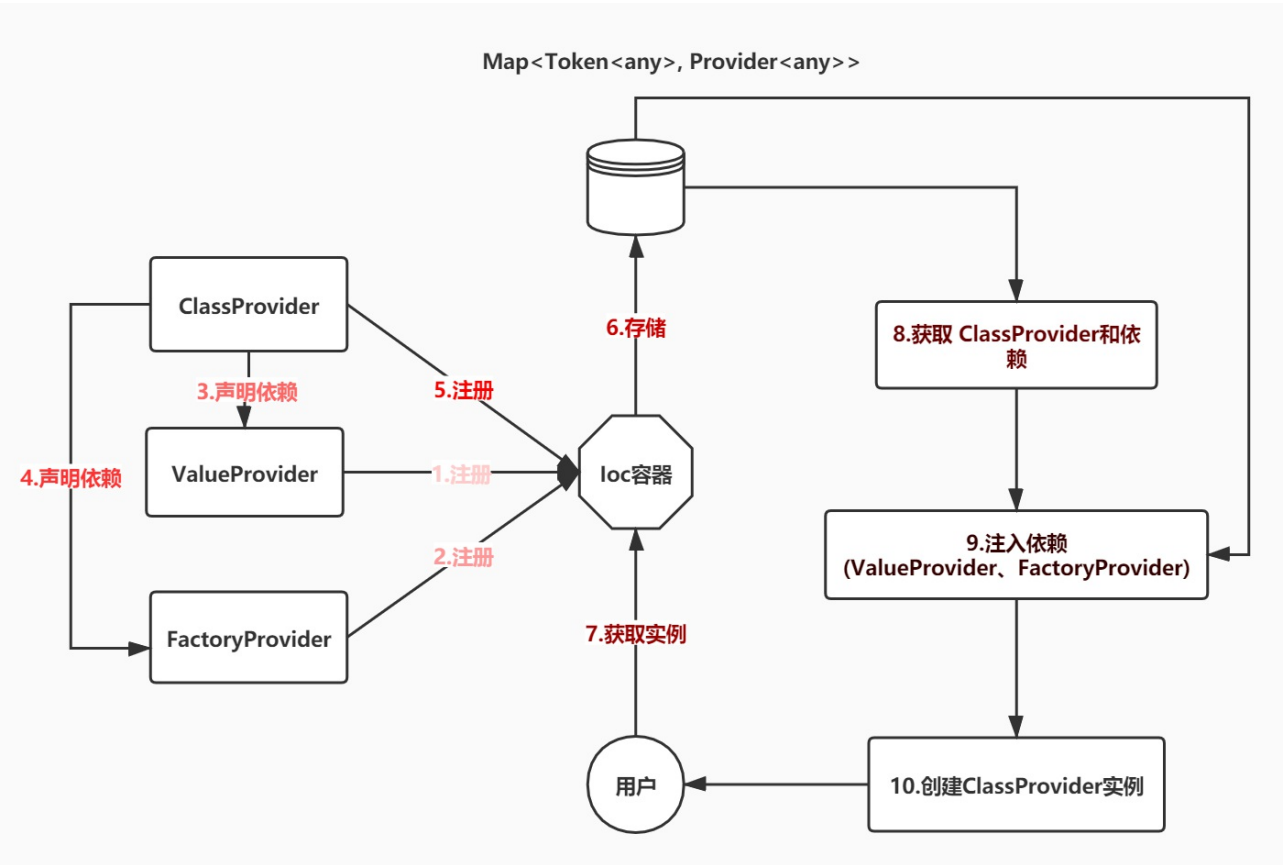
3.3.4 logger.service.ts

src\app.logger.ts

```
import { Injectable } from "@nestjs/common";

@Injectable()
export class UseClassLoggerService {
  constructor(){
    console.log('创建 UseClassLoggerService');
  }
  log(message:string) {
   console.log(message);
  }
}

export class UseValueLoggerService {
  log(message: string) {
    console.log(message);
  }
}
export class UseValueLoggerServiceStringToken {
  log(message: string) {
    console.log(message);
  }
}
export class UseFactoryLoggerService {
  log(message: string) {
    console.log(message);
  }
}
```



4. IOC

4.1 注册Provider

4.1 type.ts

```
export interface Type{
    new(...args: any[]): T;
}
```

4.2 Container.ts

```
import {Provider,Token} from "./provider";
export class Container {
    public providers = new Map, Provider>();
    addProvider(provider: Provider) {
        this.providers.set(provider.provide, provider);
    }
}
```

4.3 provider.ts

provider.ts

```
import { Type } from "./type";
export class InjectionToken {
    constructor(public injectionIdentifier: string) { }
}

export type Token = Type | InjectionToken;

export interface BaseProvider {
    provide: Token;
}

export interface ClassProvider extends BaseProvider {
    provide: Token;
    useClass: Type;
}

export interface ValueProvider extends BaseProvider {
    provide: Token;
    useValue: T;
}

export interface FactoryProvider extends BaseProvider {
    provide: Token;
    useFactory: () => T;
}

export type Provider =
    | ClassProvider
    | ValueProvider
    | FactoryProvider;
```

4.4 index.ts

```
export * from "./container";
```

4.5 ioc\index.spec.ts

```
import {Container} from './';
let container = new Container();
const point = { x: 100,y:100 };
class BasicClass { }

container.addProvider({ provide: BasicClass, useClass: BasicClass });

container.addProvider({ provide: BasicClass, useValue: point });

container.addProvider({ provide: BasicClass, useFactory: () => point });
console.log(container.providers);
```

4.2 装饰器

4.2.1 Injectable

- Injectable 装饰器用于表示此类可以自动注入其依赖项，该装饰器属于类装饰器
- 类装饰器顾名思义，就是用来装饰类的。它接收一个参数：target: TFunction，表示被装饰的类

```
declare type ClassDecorator = <TFunction extends Function>(target: TFunction) => TFunction;
```

4.2.2 Inject

- Inject装饰器属于参数装饰器
- 参数装饰器顾名思义，是用来装饰函数参数，它接收三个参数
    - target: Object —— 被装饰的类
    - propertyKey: string | symbol —— 方法名
    - parameterIndex: number —— 方法中参数的索引值

```
declare type ParameterDecorator = (target: Object,
  propertyKey: string | symbol, parameterIndex: number ) => void
```

4.2.3 实现

4.2.3.1 Injectable.ts

```
import "reflect-metadata";
const INJECTABLE_METADATA_KEY = Symbol("INJECTABLE_KEY");

export function Injectable() {
    return function (target: any) {
        Reflect.defineMetadata(INJECTABLE_METADATA_KEY, true, target);
        return target;
    };
}
```

4.2.3.2 Inject.ts

```
import 'reflect-metadata';
import { Token } from './provider';

const INJECT_METADATA_KEY = Symbol('INJECT_KEY');

export function Inject(token: Token) {
    return function (target: any, _: string | symbol, index: number) {
        Reflect.defineMetadata(INJECT_METADATA_KEY, token, target, `index-${index}`);
        return target;
    };
}
```

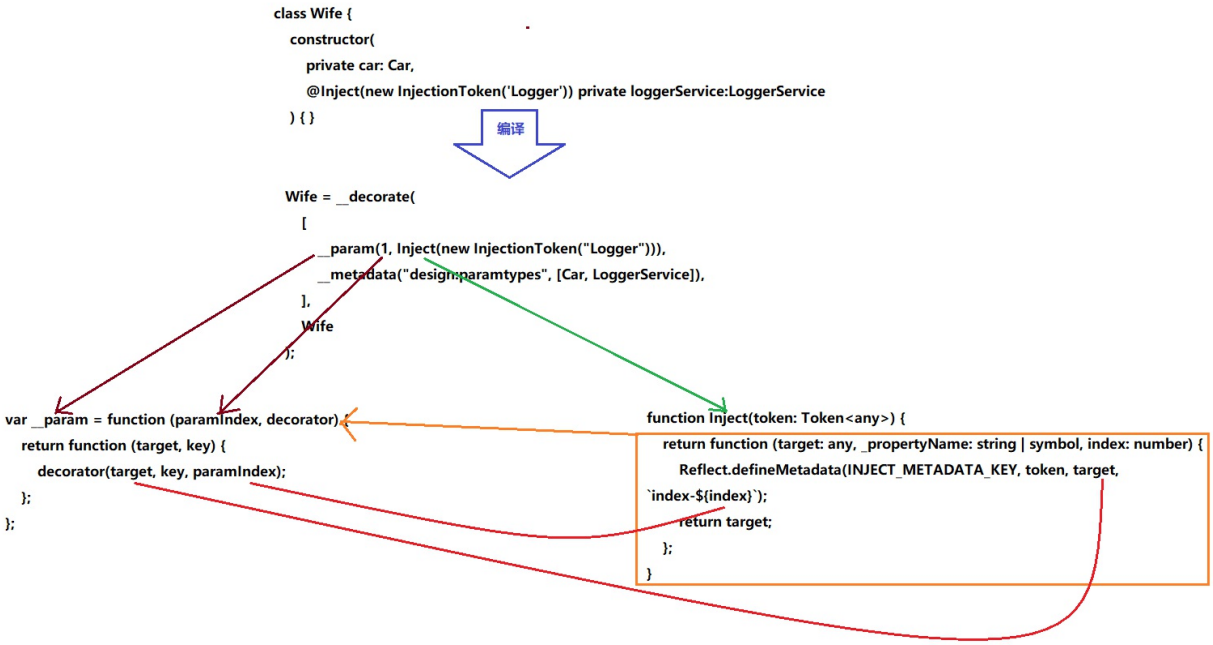
4.3 实现 inject

- inject 方法所实现的功能就是根据 Token 获取与之对应的对象

4.3.1 Container.ts

```
import {
    Provider, Token, InjectionToken,
    ClassProvider, ValueProvider,FactoryProvider,
    isClassProvider, isValueProvider,isFactoryProvider} from "./provider";
import { Type } from "./type";
export class Container {
    public providers = new Map, Provider>();
    addProvider(provider: Provider) {
        this.providers.set(provider.provide, provider);
    }
    inject(type: Token): T {
        let provider = this.providers.get(type);
        return this.injectWithProvider(type, provider);
    }
    private injectWithProvider(type: Token, provider?: Provider): T {
        if (isClassProvider(provider)) {
            //TODO
        } else if (isValueProvider(provider)) {
            return this.injectValue(provider as ValueProvider);
        } else if (isFactoryProvider(provider)){
            return this.injectFactory(provider as FactoryProvider);
        }
    }
    private injectValue(valueProvider: ValueProvider): T {
        return valueProvider.useValue;
    }
    private injectFactory(valueProvider: FactoryProvider): T {
        return valueProvider.useFactory();
    }
}
```

- __decorate:执行装饰器的函数，被执行的装饰器分为四类，类装饰器、参数装饰器、方法装饰器，还有一类特殊的装饰器是ts编译选项 `emitDecoratorMetadata`生成的装饰器，用来定义一些特殊元数据 design:paramtypes等，这些特殊元数据可以获取编译之前的类型信息
    - 参数类型元数据使用元数据键"design:type"
    - 参数类型元数据使用元数据键"design:paramtypes"
    - 返回值类型元数据使用元数据键"design:returntype"
- __metadata: 类装饰器工厂，获取的装饰器会将指定键值对与类关联起来
- __param: 参数装饰器工厂，根据参数下标、参数装饰器、获取最终的装饰器，并且将参数下标传递给装饰器




```
tsc params/index.ts --experimentalDecorators --emitDecoratorMetadata --target es5
```

4.4.1 装饰器

```
function classDecorator(constructor: Function) {}

function propertyDecorator(target: any, property: string) {}

function methodDecorator(target: any, property: string, descriptor: PropertyDescriptor) { }

function paramDecorator(target: any, methodName: string, paramsIndex: number) { }
```

4.4.2 params\index.ts

```ts
import 'reflect-metadata';
interface Type {
    new(...args: any[]): T;
}
class InjectionToken {
    constructor(public injectionIdentifier: string) { }
}
type Token = Type | InjectionToken;

const INJECT_METADATA_KEY = 'INJECT_KEY';

function Inject(token: Token) {
    return function (target: any, _propertyName: string | symbol, index: number) {
        Reflect.defineMetadata(INJECT_METADATA_KEY, token, target, `index-${index}`);
        return target;
    };
}

class Car { }

class LoggerService { }

class Wife {
    constructor(
        private car: Car,
        @Inject(new InjectionToken('Logger')) private loggerService:LoggerService
    ) { }
}
console.log(Wife);
```

4.4.2 params\index.js

```js
;
var __decorate = (this && this.__decorate) || function (decorators, target, key, desc) {
    var c = arguments.length, r = c < 3 ? target : desc === null ? desc = Object.getOwnPropertyDescriptor(target, key) : desc, d;
    if (typeof Reflect === "object" && typeof Reflect.decorate === "function") r = Reflect.decorate(decorators, target, key, desc);
    else for (var i = decorators.length - 1; i >= 0; i--) if (d = decorators[i]) r = (c < 3 ? d(r) : c > 3 ? d(target, key, r) : d(target, key)) || r;
    return c > 3 && r && Object.defineProperty(target, key, r), r;
};
var __metadata = (this && this.__metadata) || function (k, v) {
    if (typeof Reflect === "object" && typeof Reflect.metadata === "function") return Reflect.metadata(k, v);
};
var __param = (this && this.__param) || function (paramIndex, decorator) {
    return function (target, key) { decorator(target, key, paramIndex); }
};
Object.defineProperty(exports, "__esModule", { value: true });
require("reflect-metadata");
var InjectionToken =  (function () {
    function InjectionToken(injectionIdentifier) {
        this.injectionIdentifier = injectionIdentifier;
    }
    return InjectionToken;
}());
var INJECT_METADATA_KEY = 'INJECT_KEY';
function Inject(token) {
    return function (target, _propertyName, index) {
        Reflect.defineMetadata(INJECT_METADATA_KEY, token, target, "index-" + index);
        return target;
    };
}
var Car =  (function () {
    function Car() {
    }
    return Car;
}());
var LoggerService =  (function () {
    function LoggerService() {
    }
    return LoggerService;
}());
var Wife =  (function () {
    function Wife(car, loggerService) {
        this.car = car;
        this.loggerService = loggerService;
    }
    Wife = __decorate([
        __param(1, Inject(new InjectionToken('Logger'))),
        __metadata("design:paramtypes", [Car,
            LoggerService])
    ], Wife);
    return Wife;
}());
console.log(Wife);
```

4.4.3 params\imp.js

```
;
require("reflect-metadata");

var __decorate = function (decorators, target, key, desc) {

    var argsLength = arguments.length,
        decoratorTarget =
            argsLength < 3
                ? target
                : desc === null
                    ? (desc = Object.getOwnPropertyDescriptor(target, key))
                    : desc;

    if (typeof Reflect === "object" && typeof Reflect.decorate === "function")
      decoratorTarget = Reflect.decorate(decorators, target, key, desc);

    else {
      for (var i = decorators.length - 1; i >= 0; i--) {
        let decorator = decorators[i];
        if (decorator) {
          decoratorTarget =
            (argsLength < 3
              ? decorator(decoratorTarget)
              : argsLength > 3
              ? decorator(target, key, decoratorTarget)
              : decorator(target, key)) || decoratorTarget;
        }
      }
    }

    return (
      argsLength > 3 &&
        decoratorTarget &&
        Object.defineProperty(target, key, decoratorTarget),
      decoratorTarget
    );
}

var __metadata = function (k, v) {
    return Reflect.metadata(k, v);
};

var __param = function (paramIndex, decorator) {
    return function (target, key) {
        decorator(target, key, paramIndex);
    };
};

var InjectionToken = (function () {
    function InjectionToken(injectionIdentifier) {
        this.injectionIdentifier = injectionIdentifier;
    }
    return InjectionToken;
})();
var INJECT_METADATA_KEY = "INJECT_KEY";

function Inject(token) {
    return function (target, _propertyName, index) {
        Reflect.defineMetadata(INJECT_METADATA_KEY,token,target,"index-" + index);
        return target;
    };
}
var Car = (function () {
    function Car() { }
    return Car;
})();
var LoggerService = (function () {
    function LoggerService() { }
    return LoggerService;
})();
var Wife = (function () {
    function Wife(car, loggerService) {
        this.car = car;
        this.loggerService = loggerService;
    }
    Wife = __decorate(
        [
            __param(1, Inject(new InjectionToken("Logger"))),
            __metadata("design:paramtypes", [Car, LoggerService]),
        ],
        Wife
    );
    return Wife;
})();
console.log(Reflect.getMetadata(INJECT_METADATA_KEY, Wife, "index-1"));
console.log(Wife);
```

4.5 实现注入

4.5.1 Container.ts

```
import {
    Provider, Token, InjectionToken,
    ClassProvider, ValueProvider,FactoryProvider,
+    isClassProvider, isValueProvider,isFactoryProvider} from "./provider";
+import { getInjectionToken, } from "./inject";
+import { Type } from "./type";
+type InjectableParam = Type;
+const REFLECT_PARAMS = "design:paramtypes";
export class Container {
    public providers = new Map, Provider>();
    addProvider(provider: Provider) {
        this.providers.set(provider.provide, provider);
    }
+    inject(type: Token): T {
+        let provider = this.providers.get(type);
+        return this.injectWithProvider(type, provider);
+    }
+    private injectWithProvider(type: Token, provider?: Provider): T {
+        if (provider === undefined) {
+            throw new Error(`No provider for type ${this.getTokenName(type)}`);
+        }
+        if (isClassProvider(provider)) {
+            return this.injectClass(provider as ClassProvider);
+        } else if (isValueProvider(provider)) {
+            return this.injectValue(provider as ValueProvider);
+        } else if (isFactoryProvider(provider)){
+            return this.injectFactory(provider as FactoryProvider);
+        }
+    }
+    private injectValue(valueProvider: ValueProvider): T {
+        return valueProvider.useValue;
+    }
+    private injectFactory(valueProvider: FactoryProvider): T {
+        return valueProvider.useFactory();
+    }
+    private injectClass(classProvider: ClassProvider): T {
+        const target = classProvider.useClass;
+        const params = this.getInjectedParams(target);
+        return Reflect.construct(target, params);
+    }
+    private getInjectedParams(target: Type) {
+        const argTypes = Reflect.getMetadata(REFLECT_PARAMS, target) as (
+            | InjectableParam
+            | undefined)[];
+        if (argTypes === undefined) {
+            return [];
+        }
+        return argTypes.map((argType, index) => {
+            const overrideToken = getInjectionToken(target, index);
+            const actualToken = overrideToken === undefined ? argType : overrideToken;
+            let provider = this.providers.get(actualToken);
+            return this.injectWithProvider(actualToken, provider);
+        });
+    }
+    private getTokenName(token: Token) {
+        return token instanceof InjectionToken
+            ? token.injectionIdentifier
+            : token.name;
+    }
}
```

4.5.2 ioc\provider.ts

```
import { Type } from "./type";
export class InjectionToken {
    constructor(public injectionIdentifier: string) { }
}
//Token 类型是一个联合类型，既可以是一个函数类型也可以是 InjectionToken 类型
export type Token = Type | InjectionToken;

export interface BaseProvider {
    provide: Token;
}

export interface ClassProvider extends BaseProvider {
    provide: Token;
    useClass: Type;
}

export interface ValueProvider extends BaseProvider {
    provide: Token;
    useValue: T;
}

export interface FactoryProvider extends BaseProvider {
    provide: Token;
    useFactory: () => T;
}

export type Provider =
    | ClassProvider
    | ValueProvider
    | FactoryProvider;

+export function isClassProvider(
+    provider: BaseProvider
+): provider is ClassProvider {
+    return (provider as any).useClass !== undefined;
+}

+export function isValueProvider(
+    provider: BaseProvider
+): provider is ValueProvider {
+    return (provider as any).useValue !== undefined;
+}

+export function isFactoryProvider(
+    provider: BaseProvider
+): provider is FactoryProvider {
+    return (provider as any).useFactory !== undefined;
+}
```

4.5.3 Inject.ts

ioc\Inject.ts

```
import 'reflect-metadata';
import { Token } from './provider';

const INJECT_METADATA_KEY = Symbol('INJECT_KEY');

export function Inject(token: Token) {
    return function (target: any, _: string | symbol, index: number) {
        Reflect.defineMetadata(INJECT_METADATA_KEY, token, target, `index-${index}`);
        return target;
    };
}
export function getInjectionToken(target: any, index: number) {
    return Reflect.getMetadata(INJECT_METADATA_KEY, target, `index-${index}`) as
        | Token
        | undefined;
}
```

4.5.4 Injectable.ts

ioc\Injectable.ts

```
import "reflect-metadata";
const INJECTABLE_METADATA_KEY = Symbol("INJECTABLE_KEY");

export function Injectable() {
    return function (target: any) {
        Reflect.defineMetadata(INJECTABLE_METADATA_KEY, true, target);
        return target;
    };
}
```

4.6 index.spec.ts

- 测试一下注释是否正确

```
import { Container } from "./container";
import { Injectable } from "./injectable";
import { Inject } from "./inject";
import { InjectionToken } from "./provider";

const HouseStringToken = new InjectionToken("House");

@Injectable()
class Car { }

@Injectable()
class House { }

@Injectable()
class Wife {
    constructor(
        private car: Car,
        @Inject(HouseStringToken) private house: House
    ) { }
}

const container = new Container();

container.addProvider({
    provide: HouseStringToken,
    useValue: new House(),
});

container.addProvider({ provide: Car, useClass: Car });
container.addProvider({ provide: Wife, useClass: Wife });
const wife = container.inject(Wife);
console.dir(wife);
```

5. debugger

```
cnpm i ts-node typescript reflect-metadata -D
```

.vscode\launch.json

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug ts",
            "type": "node",
            "request": "launch",
            "runtimeArgs": [
                "-r",
                "ts-node/register"
            ],
            "args": [
                "${relativeFile}"
            ]
        }
    ]
}
```