前言

1、控制反转：Inversion of Control，简称IoC

2、依赖注入：Dependency Injection，简称DI，依赖注入实际上是控制反转概念的一种实现模式。一种消除类之间依赖关系的设计模式。

假设我们有一个类`Human`，要实例一个`Human`，我们需要实例一个类`Clothes`，而实例化衣服`Clothes`，我们又需要实例化`Cloth`，实例化纽扣等等。

当需求达到一定复杂的程度时，我们不能为了一个人穿衣服去从布从纽扣开始从头实现，最好能把所有的需求放到一个工厂或者仓库，我们需要什么直接从工厂的仓库里面直接拿（与直接封装成一个`Clothes`类，我们需要时再去new有什么区别? -- 假如去`new`实例的话，那类与类之间还是存在着依赖关系）

这个时候就需要依赖注入了，我们实现一个IoC容器（仓库），然后需要衣服就从仓库里面直接拿实例好的衣服给人作为属性穿上去。

将所有的类与类之间的依赖关系交给IoC容器去管理，**之后真正使用的是IoC容器中已存在依赖关系的类的实例**（并非再去`new`新的实例，因此并不会影响类源代码逻辑，个人猜想：若后面需要扩展需求可以根据源类再创建别的容器）

IoC是一种很好的解耦合思想，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。在软件开发中有很好的作用。

基于ts中装饰器的使用

类装饰器

类装饰器表达式会在运行时当作函数被调用，**类的构造函数作为其唯一的参数**

可以用来监视，修改或替换类定义

修改类的构造器：

```
function classDecorator<T extends {new(...args:any[]):{}}>(constructor:T) {
  console.log('装饰器已执行');
  return class extends constructor {
    newProperty = "new property";
    hello = "override";
  }
}
@classDecorator
class Greeter {
  property = "property";
  hello: string;
  constructor(m: string) {
      this.hello = m;
      console.log(this.hello);
  }
}
console.log(new Greeter("world"));
// 装饰器已执行
// world
// Greeter {
```

```
//   property: 'property',
//   hello: 'override',
//   newProperty: 'new property'
// }
```

程序运行顺序:
1、为类Greeter添加装饰内容（加在原来类的构造函数后面，因此能起到重写构造函数属性的效果）
2、实例化类；

修改类的某个方法：

```
function cheating(target: any) {
  target.prototype.hit = function(rival: Somebody) {
    const hitDamage: number = 100;
    console.log(`${this.name}对${rival.name}造成一次伤害: ${hitDamage}`);
  }
}
class Somebody {
  speed: number = 10;
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  hit(rival: Somebody) {
    const hitDamage: number = 10;
    console.log(`${this.name}对${rival.name}造成一次伤害：${hitDamage}`);
  }
}
@cheating
class SBody extends Somebody{
}
const s0 = new Somebody('小红0');
const s1 = new SBody('小红1');
const rival = new Somebody('小明');
s0.hit(rival); // 小红0对小明造成一次伤害：10
s1.hit(rival); // 小红1对小明造成一次伤害: 100
```

属性装饰器

当要装饰的属性为实例属性时（没有`static`修饰），装饰的是该类原型上的属性

```
function logProperty(params: any) {
  // target--->类的原型对象；attr--->装饰的属性名
  return function (target: any, attr: any) {
    target[attr] = params;
  }
}

class HttpClient1 {
  @logProperty('http://www.baidu1.com')
  url: any | undefined;
  constructor() {
  }
  getUrl() {
    console.log(this.url);
  }
```

```
}
class HttpClient2 {
  @logProperty('http://www.baidu22.com')
  url: any | undefined = 'http://www.baidu2.com';
  constructor() {
  }
  getUrl() {
    console.log(this.url);
  }
}
let http1 = new HttpClient1();
http1.getUrl(); // http://www.baidu1.com  HttpClient1类中url没有初始化，因此获取的是原型链上的值

let http2 = new HttpClient2();
http2.getUrl(); // http://www.baidu2.com 获取的是HttpClient2中url初始化的值（先确定自身对象中是否有这个值，没有的话再去原型链上找，再没有就返回undefined）
```

当要装饰的属性为静态属性时（`static`修饰），装饰的是该类上的静态属性

```
function logProperty(params: any) {
  // target--->类；attr--->装饰的属性名
  return function (target: any, attr: any) {
    target[attr] = params;
  }
}

class HttpClient1 {
  @logProperty('http://www.baidu1.com')
  static url: any | undefined;
  constructor() {
  }
  getUrl() {
    console.log(HttpClient1.url);
  }
}
class HttpClient2 {
  @logProperty('http://www.baidu22.com')
  static url: any | undefined = 'http://www.baidu2.com';
  constructor() {
  }
  getUrl() {
    console.log(HttpClient2.url);
  }
}
console.log(HttpClient1.url); // http://www.baidu1.com
console.log(HttpClient2.url); // http://www.baidu22.com
```

程序运行顺序：

1、为属性执行装饰器函数（给原型上对应属性赋值 || 给类静态属性赋值 || 其他逻辑操作）；

2、实例化类；

参数装饰器

```
// 参数装饰器 接受3个参数
// target: Object -- 被装饰的参数所在的方法的类
// methodName: string | symbol -- 方法名
// paramsIndex: number -- 方法中参数的索引值
// 在构造器中的参数使用
function logParams(params: any) {
  return function (target: any, methodName: any, paramsIndex: any) {
    console.log(1, params);
    console.log(2, target);
    console.log(3, methodName);
    console.log(4, paramsIndex);
  }
}
class HttpClient {
  public url: any | undefined;
  constructor() {
  }
  getData(@logParams('uuid') uuid: any) {
    console.log(uuid);
  }
}
let http = new HttpClient();
http.getData(123456);
// 1 uuid
// 2 HttpClient {}
// 3 getData
// 4 0
// 123456
```

reflect-metadata

当想要给类中的某个属性或类构造函数中某个参数设置元数据时，一般采用`reflect-metadata`这种方式（**暂未找到其他方式，原生js如何给一个对象属性设置元数据???，通过反射来获取类属性上面的批注**）

```
import 'reflect-metadata';
class Post {
  @Reflect.metadata('role', 'admin')
  haha: {haha?: string; test?: number}
}
const metadata = Reflect.getMetadata('role', Post.prototype, 'haha');
console.log(metadata);  // admin
```

1、可以用于给类中一些属性设置一些元数据。元数据指的是表述东西时用的数据。
2、`Reflect.metadata`：可以用于给某个类中的某个属性值添加一个元数据（键值对）
3、`Reflect.getMetadata`：获取某个类中某个属性值的某个元数据（并不会干扰到原来的属性值）

```
import "reflect-metadata";
// 用来记录某个属性的元数据
const formatMetadataKey = Symbol("format");
function format(formatString: string) {
  // Reflect.metadata返回一个函数 (target: Object, propertyKey: string | symbol):
void
```

```
  return Reflect.metadata(formatMetadataKey, formatString);
}
function getFormat(target: any, propertyKey: string) {
  // 获取类中propertyKey的某个元数据(批注)
  return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}
class Greeter {
  @format("Hello, %s")
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    let formatString = getFormat(this, "greeting");
    return formatString.replace("%s", this.greeting);
  }
}
const test = new Greeter('syz').greet();
console.log(test); // Hello, syz
```

reflect-metadata参考1
reflect-metadata参考2

程序调试技巧

1、使用VS Code配合插件`Code Runner`（鼠标框选程序右键`Run Code`即可），因为程序使用的是typescript，使用`Code Runner`运行需要在全局安装`ts-node`（`npm install ts-node -g`）和`typescript`（`npm install typescript -g`）
在文件同级目录下配置一下`tsconfig.json`文件即可

```
{
  "compileOnSave": false,
  "buildOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "module": "commonjs",
    "moduleResolution": "node",
    "target": "ES6",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "noImplicitAny": true,
    "removeComments": false,
    "sourceMap": false,
    "inlineSourceMap": true,
    "noEmitHelpers": false,
    "declaration": true,
  }
}
```

2、备注：`tsc xxx.js`可将ts文件编译为js文件，可以查看编译后的js代码，可能更容易理解！

简易DI实现

涉及到类装饰器 & 属性装饰器的使用

思路流程

1、实现一个IoC容器`Injector`，并实例化一个根容器`rootInjector`（用于存放各个依赖的工厂容器）
2、实现一个依赖注入方法`Injectable(...)`（用于将各个依赖类注入根容器）
3、实现基于注解的属性注入方法`Inject(...)`（将类需要用到的依赖从根容器取出来并注入到类中，若根容器不存在则创建此依赖）

```
import 'reflect-metadata';
// 工厂里面的各种操作
export class Injector {
  private readonly providerMap: Map<any, any> = new Map();
  private readonly instanceMap: Map<any, any> = new Map();
  public setProvider(key: any, value: any): void {
    if (!this.providerMap.has(key)) {
      this.providerMap.set(key, value);
    }
  }
  public getProvider(key: any): any {
    return this.providerMap.get(key);
  }
  public setInstance(key: any, value: any): void {
    if (!this.instanceMap.has(key)) this.instanceMap.set(key, value);
  }
  public getInstance(key: any): any {
    if (this.instanceMap.has(key)) return this.instanceMap.get(key);
    return null;
  }
  public setValue(key: any, value: any): void {
    if (!this.instanceMap.has(key)) this.instanceMap.set(key, value);
  }
}
// 表示根注入器(用于存放各个依赖的根容器)
export const rootInjector = new Injector();
// 将类注入到工厂中 类装饰器返回一个值，它会使用提供的构造函数来替换原来类的声明
export function Injectable(): (_constructor: any) => any {
  return function (_constructor: any): any {
    rootInjector.setProvider(_constructor, _constructor);
    return _constructor;
  };
}
// 将依赖注入到生产者
export function Inject(): (_constructor: any, propertyName: string) => any {
  return function (_constructor: any, propertyName: string): any {
    // 获取属性定义时的类型
    const  propertyType: any = Reflect.getMetadata('design:type', _constructor,
propertyName);
    const injector: Injector = rootInjector;
    let providerInsntance = injector.getInstance(propertyType);
    if (!providerInsntance) {
      const providerClass = injector.getProvider(propertyType);
      providerInsntance = new providerClass();
```

```
      injector.setInstance(propertyType, providerInsntance);
    }
    _constructor[propertyName] = providerInsntance;
  };
}
@Injectable()
class Cloth {
  name: string = '麻布';
}
@Injectable()
class Clothes {
  // 为类Clothes注入类Cloth 之后类Clothes就拥有了使用类Cloth的能力
  @Inject()
  cloth: Cloth;
  clotheName: string;
  constructor() {
    this.cloth = this.cloth;
    this.clotheName = this.clotheName;
  }
  updateName(name: string) {
    this.clotheName = name;
  }
}
class Human1 {
  @Inject()
  clothes: Clothes;
  name: string;
  constructor(name: string) {
    this.clothes = this.clothes;
    this.name = name;
  }
  update(name: string) {
    this.clothes.updateName(name);
  }
}
// 单例：用于数据状态的维护(一个变 所有变)
const pepe1 = new Human1('syz');
console.log(pepe1);
// Human1 {
//   clothes: Clothes { cloth: Cloth { name: '麻布' }, clotheName: undefined }
// }
pepe1.update('耐克');
console.log(pepe1);
// Human1 {
//   clothes: Clothes { cloth: Cloth { name: '麻布' }, clotheName: '耐克' }
// }
```

简易DI参考

仿Angular中的DI

service依赖注入，使用单例模式，以后单个或多个组件需要使用这个服务时，都是共享同一个实例（**因此可用于状态共享**），极大的减少了内存资源的损耗（正常情况每次使用类都得实例化的）

> 在某个注入器的范围内，服务是单例的。也就是说，在指定的注入器中最多只有某个服务的最多一个实例。
> 应用只有一个根注入器。在root或AppModule级提供UserService意味着它注册到了根注入器上。在整个应用中只有一个UserService实例，每个要求注入UserService的类都会得到这一个服务实例，除非你在子注入器中配置了另一个提供者

1、如何将依赖从容器中取出来呢 -- 调用`inject()`方法会返回类的实例（`Reflect.construct(...)`用于实例化，接着调用了类的构造函数，将对应的依赖实例作为构造函数参数传进去）
2、写在类中`constructor()`中的参数都是要注入依赖的。普通类型，例如`string`，`number`不能写进去，否则报错

相关技术：
1、类装饰器，配合`reflect-metadata`
2、参数装饰器，配合`reflect-metadata`

代码实现

1、实现`Injectable()`类装饰器，用于标记类是否可被注入容器中

```
import { Type } from './type';
import 'reflect-metadata';
const INJECTABLE_METADATA_KEY = Symbol('INJECTABLE_KEY');
// 必须被这个装饰器修饰的Class才说明是可以被注入的 (表示Class自己承认是可被注入的)
export function Injectable() {
  return function(target: any) {
    Reflect.defineMetadata(INJECTABLE_METADATA_KEY, true, target);
    return target;
  }
}

// 检查Class是否可被注入
export function isInjectable<T>(target: Type<T>) {
  return Reflect.getMetadata(INJECTABLE_METADATA_KEY, target) === true;
}
```

2、实现`Inject()`参数装饰器，用于将value值注入类中

```
import { Token } from './provider';
import 'reflect-metadata';
const INJECT_METADATA_KEY = Symbol('INJECT_KEY');
// 参数装饰器 接受3个参数
// target: Object -- 被装饰的参数所在的方法的类
// methodName: string | symbol -- 方法名
// parameterIndex: number -- 方法中参数的索引值
// 在构造器中的参数使用
export function Inject(token: Token<any>) {
  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<any> | undefined;
}
```

3、实现`addProvider(...)`方法，用于将所有需要注入到容器的所有类/值添加到`providers`变量中，后续`inject(...)`将从`providers`取值

4、实现`inject(...)`方法，将类注入容器并返回实例（使用`Reflect.construct`方法）

```
export class Container {

  providers = new Map<Token<any>, Provider<any>>();
  instances = new Map<Token<any>, any>();

  addProvider<T>(provider: Provider<T>) {
    this.assertInjectableIfClassProvider(provider);
    // 重复注入同个类的话 貌似是以后面的为准???
    this.providers.set(provider.provide, provider);
  }
  inject<T>(type: Token<T>): T {
    let provider = this.providers.get(type);
    let instance = this.instances.get(type);
    if(provider === undefined && !(type instanceof InjectionToken)) {
      provider = { provide: type, useClass: type };
      this.assertInjectableIfClassProvider(provider);
    }
    // 容器中是否已存在实例 存在的话直接返回原有的实例 共享变量
    if(instance) {
      return instance;
    }
    return this.injectWithProvider(type, provider);
  }
  private assertInjectableIfClassProvider<T>(provider: Provider<T>) {
    if(isClassProvider(provider) && !isInjectable(provider.useClass)) {
      throw new Error(
        `Cannot provide ${this.getTokenName(provider.provide)} using class
${this.getTokenName( provider.useClass )},
        ${this.getTokenName(provider.useClass)} isn't injectable`
      );
    }
  }
  private getTokenName<T>(token: Token<T>) {
    return token instanceof InjectionToken ? token.injectionIdentifier :
token.name
  }
  private injectWithProvider<T>(type: Token<T>, provider?: Provider<T>): T {
    let instance;
    if (provider === undefined) {
      throw new Error(`No provider for type ${this.getTokenName(type)}`);
    }
    if (isClassProvider(provider)) {
      instance = this.injectClass(provider as ClassProvider<T>);
    } else if (isValueProvider(provider)) {
      instance =  this.injectValue(provider as ValueProvider<T>);
    } else {
      instance = this.injectFactory(provider as FactoryProvider<T>);
    }
    this.addInstances(type, instance);
    return instance;
  }
  // 关键 在实例化服务类时 需要构造该服务类依赖的独享(即在构造函数中注入的依赖)
```

```
  private injectClass<T>(classProvider: ClassProvider<T>): T {
    const target = classProvider.useClass;
    const params = this.getInjectedParams(target);
    // 这里的作用 类似于 new target(...params)
    return Reflect.construct(target, params);
  }
  private injectValue<T>(valueProvider: ValueProvider<T>): T {
    return valueProvider.useValue;
  }
  private injectFactory<T>(factoryProvider: FactoryProvider<T>): T {
    return factoryProvider.useFactory();
  }
  // 用于获取类构造函数中声明的依赖对象 重点&难点
  private getInjectedParams<T>(target: Type<T>) {
    // 获取参数的类型 "design:paramtypes" 用于修饰目标对象方法的参数类型
    const argTypes = Reflect.getMetadata(REFLECT_PARAMS, target) as
(InjectableParam | undefined)[];
    if(argTypes === undefined) {
      return [];
    }
    return argTypes.map((argTypes, index) => {
      if(argTypes === undefined) {
        throw new Error (
          `Injection error. Recursive dependency detected in constructor for type
${target.name} with parameter at index ${index}`
        );
      }
      const overrideToken = getInjectionToken(target, index);
      const actualToken = overrideToken === undefined ? argTypes :
overrideToken;
      let provider = this.providers.get(actualToken);
      // 递归调用 一层接一层
      return this.injectWithProvider(actualToken, provider);
    });
  }
  // 已注入容器的实例添加到变量instances
  private addInstances<T>(type: Token<T>, instance: any) {
    this.instances.set(type, instance);
  }
}
```

4、调用demo

```
import { Container } from './container';
import { Injectable } from './Injectable';
import { Inject } from './Inject';
import { InjectionToken } from './provider';
const API_URL = new InjectionToken('apiUrl');
@Injectable()
class HttpClient{
  get() {
    console.log('get');
  }
}
@Injectable()
class HttpService {
  constructor(
```

```
    // 使用这种方式是如何将HttpClient注入的 使用
Reflect.getMetadata('design:paramtypes', target)获取target类的参数类型信息
    private httpClient: HttpClient,
    @Inject(API_URL) private apiUrl: string,
  ) { }
  test() {
    console.log(this.apiUrl);
    this.httpClient.get();
  }
}
const container = new Container();
// addProvider 这一步就相当于angular中在module中的providers添加Service
container.addProvider({
  provide: API_URL,
  useValue: 'https://www.666.com/'
});
container.addProvider({provide: HttpService, useClass: HttpService });
container.addProvider({provide: HttpClient, useClass: HttpClient });
const httpService = container.inject(HttpService);
httpService.test();

// https://www.666.com/
//get
```

关键点

1、`Reflect.getMetadata('design:paramtypes', value)`：获取类构造函数参数的类型

2、`Reflect.construct(data, params)`：实例化类

3、类中依赖如果存在多层采用递归注入方式

完整代码 查看

疑问

1、一种消除类之间依赖关系的设计模式，具体是如何体现的？貌似还是存在依赖，只是封装起来了而已?
解答：将所有的类与类之间的依赖关系交给IoC容器去管理，之后真正使用的是IoC容器中已存在依赖关系的类的实例（并非再去创建新的实例，因此并不会影响类源代码逻辑，个人猜想：若后面需要扩展需求可以根据源类再创建别的容器）；

2、angular中同个Service在多个地方注入，获取依赖时如何确定要获取哪一个值?
解答：如果是在同一个注入容器重复注入，按照注入的先后顺序，后面注入的会重写前面注入的;

3、在 `construct` 中 `private httpClient: HttpClient` 是如何将 `HttpClient` 注入依赖的?
解答：使用 `Reflect.getMetadata('design:paramtypes', target)` 获取 `target` 类的参数类型信息;