林不渡 Lv2
2020年08月10日 阅读 1268

走近MidwayJS：初识TS装饰器与IoC机制

前言

很惭愧在阿里实习将近三个月没有一点文章产出，同期入职的 炽翎 和 炬透 都产出了不少优秀的文章，如不想痛失薪资普调和年终奖？试试自动化测试！（基础篇），不禁感慨优秀的人都是有共同点的：善于总结沉淀，而且文笔还好（这点太羡慕了）。入职即将满三个月，也就是说我三个多月没写过文章了。文笔拙劣，还请见谅。

本篇文章是 MidwayJS 的系列推广文章第一篇，原本我打算直接一篇搞定，做个MidwayJS开发后台应用的教程就好了。但是在提笔前询问过一些同学，发现即使是已经有工作经验的前端同学中也有一部分没有了解过TS装饰器相关的知识，对于IoC机制也知之甚少（虽然没学过Java的我同样只是一知半解），因此这篇文章会首先讲解**IoC机制（依赖注入）**与**TS装饰器**相关的知识，力求内容不枯燥，并使各位成功的对MidwayJS来电~

MidwayJS简介

> MidwayJS目前已经升级到Midway-Serverless体系，这可能会给没接触过Serverless、只是想学习框架本身的你带来一些困扰。你可以先阅读其框架本身文档，来只体验框架本身作为后端应用的能力。

你可能没有听过 `Egg` ，但你一定听过或者使用过 `Koa` / `Express` ， `Egg` 基于 `Koa` 并在其能力上做了增强，奉行**【约定优于配置】**，同时它又能作为一款定制能力强的基础框架，来使得你能基于自己的技术架构封装出一套适合自己业务场景的框架。 `MidwayJS` 正是基于 `Egg` ，但在 `Egg` 的基础上做了一些较大的变动：

- 更好的TS支持，可以说写MidwayJS比较舒服的一个地方就是它的**TypeScript支持**了，比如会作为服务的接口定义会单独存放于 `interface` ，提供的能力强大的装饰器，与TypeORM这种TS支持好的框架协作起来更是愉悦。
- IoC机制的路由，以我们下篇文章将要实现的接口为例：

```
export class UserController {

  @get('/all')
  async getUser(): Promise<void> {
    // ...
  }

  @get('/uid/:uid')
  async findUserByUid(): Promise<void> {
    // ...
  }

  @post('/uid/:uid')
  async updateUser(): Promise<void> {
    // ...
  }

  // ...

}
```

（Midway同时保留了Egg的路由能力，即 `src/app/router.ts` 的路由配置方式）

这里是否会让你想到 `NestJS` ？的确在路由这里二者的思想基本是相同的，但Midway的IoC机制底层基于 Injection，同样是Midway团队的作品。并且，Midway的IoC机制也是 `Midway-Serverless` 能力的重要支持（这个我们下篇文章才会讲到）。

- 生态复用，Egg与Koa的中间件大部分能在Midway应用中完美兼容,少部分暂不支持的也由官方团队在快速兼容。

- 稳定支持，MidwayJS至今仍在快速发展迭代，同时也在阿里内部作为Serverless基建的重要成员而受到相当的重视，所以你不用担心它后续的维护情况。

下面的部分里，我们会讲解这些东西：

- **TS装饰器 基本语法、类型**
- **Reflect 元编程**
- **IoC机制与依赖注入（Dependence Injection）**
- **实现简单的基于IoC的路由**

TS装饰器的那些事儿

首先我们需要知道，JS与TS中的装饰器不是一回事，JS中的装饰器目前依然停留在 stage 2 阶段，并且目前版本的草案与TS中的实现差异相当之大（TS是基于第一版，JS目前已经第三版了），所以二者最终的装饰器实现必然有非常大的差异。

其次，装饰器不是TS所提供的特性（如类型、接口），而是TS实现的ECMAScript提案（就像类的私有成员一样）。TS实际上只会对**stage-3**以上的语言提供支持，比如TS3.7.5引入了可选链（Optional chaining）与空值合并（Nullish-Coalescing）。而当TS引入装饰器时（大约在15年左右），JS中的装饰器依然处于 **stage-1** 阶段。其原因是TS与Angular团队PY成功了，Ng团队不再维护 AtScript，而TS引入了注解语法（**Annotation**）及相关特性。

但是并不需要担心，即使装饰器永远到达不了stage-3/4阶段，它也不会消失的。有相当多的框架都是装饰器的重度用户，如 `Angular` 、 `Nest` 、 `Midway` 等。对于装饰器的实现与编译结果会始终保留，就像 `JSX` 一样。如果你对它的历史与发展方向有兴趣，可以读一读 是否应该在production里使用typescript的decorator? （贺师俊贺老的回答）

为什么我们需要装饰器？在后面的例子中我们会体会到装饰器的强大与魅力，基于装饰器我们能够**快速优雅的复用逻辑**，**提供注释一般的解释说明效果**，以及**对业务代码进行能力增强**。同时我们本文的重点：**依赖注入**也可以通过装饰器来非常简洁的实现。现在我们可能暂时体会不到 **强大**、**简洁** 这些关键词，不急，安心读下去。我会尝试通过这篇文章让你对TS装饰器整体建立起一个认知，并在日常开发里也爱上使用装饰器。

装饰器与注解

> **由于我本身并没学习过Java以及Spring IoC，因此我的理解可能存在一些偏差，还请在评论区指出错误之处~**

装饰器与注解实际上也有一定区别，由于并没有学过Java，这里就不与Java中的注解进行比较了。而只是说我所认为的二者差异：

- **注解** 应该如同字面意义一样， 只是为某个被注解的对象提供元数据（ `metadata` ）的注入，本质上不能起到任何修改行为的操作，需要 `scanner` 去进行扫描获得元数据并基于其去执行操作，注解的元数据才有实际意义。
- **装饰器** 没法添加元数据，只能基于已经由注解注入的元数据来执行操作，来对类、方法、属性、参数进

不同类型的装饰器及使用

> 在开始前，你需要确保在 `tsconfig.json` 中设置了 `experimentalDecorators` 与 `emitDecoratorMetadata` 为 true。

首先要明确地是，TS中的装饰器实现本质是一个语法糖，它的本质是一个函数，如果调用形式为 `@deco()` ，那么这个函数应该再返回一个函数来实现调用。

其次，你应该明白ES6中class的实质，如果不明白，推荐阅读我的这篇文章: 从Babel编译结果看ES6的Class实质

类装饰器

ts

```ts
function addProp(constructor: Function) {
  constructor.prototype.job = 'fe';
}

@addProp
class P {
  job: string;
  constructor(public name: string) {}
}

let p = new P('林不渡');

console.log(p.job); // fe
```

我们发现，在以单纯装饰器方式 `@addProp` 调用时，不管用它来装饰哪个类，起到的作用都是相同的，因为其中要复用的逻辑是固定的。我们试试以 `@addProp()` 的方式来调用：

typescript

```typescript
function addProp(param: string): ClassDecorator {
  return (constructor: Function) => {
    constructor.prototype.job = param;
  };
}
```

```
let p = new P('林不渡');

console.log(p.job); // fe+be
```

现在我们想要添加的属性值就可以由我们决定了，实际上由于我们拿到了原型对象，还可以进行花式操作，能够解锁更多神秘姿势~

方法装饰器

方法装饰器的入参为 **类的原型对象 属性名** 以及**属性描述符(descriptor)**，其属性描述符包含 `writable` `enumerable` `configurable` ，我们可以在这里去配置其相关信息。

> 注意，对于静态成员来说，首个参数会是类的构造函数。而对于实例成员（比如下面的例子），则是类的原型对象

ts

```
function addProps(): MethodDecorator {
  return (target, propertyKey, descriptor) => {
    console.log(target);
    console.log(propertyKey);
    console.log(JSON.stringify(descriptor));

    descriptor.writable = false;
  };
}

class A {
  @addProps()
  originMethod() {
    console.log("I'm Original!");
  }
}

const a = new A();

a.originMethod = () => {
  console.log("I'm Changed!");
```

你是否觉得有点想起来 `Object.defineProperty()` ？ 的确方法装饰器也是借助它来修改类和方法的属性的，你可以去TypeScript Playground看看TS对上面代码的编译结果。

属性装饰器

类似于方法装饰器，但它的入参少了属性描述符。原因则是目前没有方法在定义原型对象成员同时去描述一个实例的属性（创建描述符）。

```typescript
function addProps(): PropertyDecorator {
  return (target, propertyKey) => {
    console.log(target);
    console.log(propertyKey);
  };
}

class A {
  @addProps()
  originProps: any;
}
```

属性与方法装饰器有一个重要作用是注入与提取元数据，这点我们在后面会体现到。

参数装饰器

参数装饰器的入参首要两位与属性装饰器相同，第三个参数则是参数在当前函数参数中的**索引**。

```ts
function paramDeco(params?: any): ParameterDecorator {
  return (target, propertyKey, index) => {
    console.log(target);
    console.log(propertyKey);
    console.log(index);
    target.constructor.prototype.fromParamDeco = '呀呼！';
  };
}

class B {
```

```
new B().someMethod('啊哈', '林不渡！');
// @ts-ignore
console.log(B.prototype.fromParamDeco);
```

参数装饰器与属性装饰器都有个特别之处，他们都不能获取到描述符descriptor，因此也就不能去修改其参数/属性的行为。但是我们可以这么做：**给类原型添加某个属性，携带上与参数/属性/装饰器相关的元数据，并由下一个执行的装饰器来读取。**(装饰器的执行顺序请参见下一节)

当然像例子中这样直接在原型上添加属性的方式是十分不推荐的，后面我们会使用ES7的 `Reflect Metadata` 来进行元数据的读/写。

装饰器工厂

假设现在我们同时需要四种装饰器，你会怎么做？定义四种装饰器然后分别使用吗？也行，但后续你看着这一堆装饰器可能会感觉有点头疼...，因此我们可以考虑接入工厂模式，使用一个装饰器工厂来为我们根据条件吐出不同的装饰器。

首先我们准备好各个装饰器函数：

（不建议把功能也写在装饰器工厂中，会造成耦合）

ts

```
// @ts-nocheck

function classDeco(): ClassDecorator {
  return (target: Object) => {
    console.log('Class Decorator Invoked');
    console.log(target);
  };
}

function propDeco(): PropertyDecorator {
  return (target: Object, propertyKey: string) => {
    console.log('Property Decorator Invoked');
    console.log(propertyKey);
  };
}
```

```
    descriptor: PropertyDescriptor
  ) => {
    console.log('Method Decorator Invoked');
    console.log(propertyKey);
  };
}

function paramDeco(): ParameterDecorator {
  return (target: Object, propertyKey: string, index: number) => {
    console.log('Param Decorator Invoked');
    console.log(propertyKey);
    console.log(index);
  };
}
```

接着，我们实现一个工厂函数来根据不同条件返回不同的装饰器：

typescript

```
enum DecoratorType {
  CLASS = 'CLASS',
  METHOD = 'METHOD',
  PROPERTY = 'PROPERTY',
  PARAM = 'PARAM',
}

type FactoryReturnType =
  | ClassDecorator
  | MethodDecorator
  | PropertyDecorator
  | ParameterDecorator;

function decoFactory(type: DecoratorType, ...args: any[]): FactoryReturnType {
  switch (type) {
    case DecoratorType.CLASS:
      return classDeco.apply(this, args);

    case DecoratorType.METHOD:
      return methodDeco.apply(this, args);

    case DecoratorType.PROPERTY:
      return propDeco.apply(this, args);
```

```
      throw new Error('Invalid DecoratorType');
    }
  }

  @decoFactory(DecoratorType.CLASS)
  class C {
    @decoFactory(DecoratorType.PROPERTY)
    prop: any;

    @decoFactory(DecoratorType.METHOD)
    method(@decoFactory(DecoratorType.PARAM) param: string) {}
  }

  new C().method();
```

（注意，这里在TS类型定义上似乎有些问题，所以需要带上顶部的 `@ts-nocheck` ，在后续解决了类型报错后，我会及时更新的TAT）

多个装饰器声明

> 装饰器求值顺序来自于TypeScript官方文档一节中的装饰器说明。

类中不同声明上的装饰器将按以下规定的顺序应用：

1. *参数装饰器*，然后依次是*方法装饰器*，*访问符装饰器*，或*属性装饰器*应用到每个实例成员。
2. *参数装饰器*，然后依次是*方法装饰器*，*访问符装饰器*，或*属性装饰器*应用到每个静态成员。
3. *参数装饰器*应用到构造函数。
4. *类装饰器*应用到类。

注意这个顺序，后面我们能够实现元数据读写，也正是因为这个顺序。

当存在多个装饰器来装饰同一个声明时，则会有以下的顺序：

- 首先，由上至下依次对装饰器表达式求值，得到返回的真实函数（如果有的话）
- 而后，求值的结果会由下至上依次调用

（有点类似洋葱模型）

```
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("foo out");
    }
}

function bar() {
    console.log("bar in");
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("bar out");
    }
}

class A {
    @foo()
    @bar()
    method() {}
}

// foo in
// bar in
// bar out
// foo out
```

Reflect Metadata

基本元数据读写

`Reflect Metadata` 是属于ES7的一个提案，其主要作用是在声明时去读写元数据。TS早在1.5+版本就已经支持反射元数据的使用，目前想要使用，我们还需要安装 `reflect-metadata` 与在 `tsconfig.json` 中启用 `emitDecoratorMetadata` 选项。

你可以将元数据理解为用于描述数据的数据，如某个对象的键、键值、类型等等就可称之为该对象的元数据。我们先不用太在意元数据定义的位置，先做一个简单的阐述：

*为类或类属性添加了元数据后，构造函数的原型（或是构造函数，根据静态成员还是实例成员决定）会具有 `[[Metadata]]` 属性，该属性内部包含一个**Map结构，键为属性键，值为元数据键值对。***

ts

```ts
import 'reflect-metadata';

@Reflect.metadata('className', 'D')
class D {
  @Reflect.metadata('methodName', 'hello')
  public hello(): string {
    return 'hello world';
  }
}

const d = new D();
console.log(Reflect.getMetadata('className', D));
console.log(Reflect.getMetadata('methodName', d));
```

可以看到，我们给类D与D内部的方法hello都注入了元数据，并通过 `getMetadata(metadataKey, target)` 这个方式取出了存放的元数据。

> Reflect-metadata支持命令式(`Reflect.defineMetadata`)与声明式（上面的装饰器方式）的元数据定义

我们注意到，注入在类上的元数据在取出时target为这个类D，而注入在方法上的元数据在取出时target则为实例d。原因其实我们实际上在上面的装饰器执行顺序提到了，这是由于**注入在方法、属性、参数上的元数据实际上是被添加在了实例对应的位置上，因此需要实例化才能取出。**

内置元数据

Reflect允许程序去检视自身，基于这个效果，我们可以在装饰器运行时去检查其类型相关信息，如目标类型、目标参数的类型以及方法返回值的类型，这需要借助TS内置的**元数据metadataKey**来实现，以一个检查入参的例子为例：

> 访问符装饰器的属性描述符会额外拥有 `get` 与 `set` 方法，其他与属性装饰器相同

typescript

```typescript
import 'reflect-metadata';

class Point {
```

```
class Line {
  private _p0: Point;
  private _p1: Point;

  @validate
  set p0(value: Point) {
    this._p0 = value;
  }
  get p0() {
    return this._p0;
  }

  @validate
  set p1(value: Point) {
    this._p1 = value;
  }
  get p1() {
    return this._p1;
  }
}

function validate<T>(
  target: any,
  propertyKey: string,
  descriptor: TypedPropertyDescriptor<T>
) {
  let set = descriptor.set!;
  descriptor.set = function (value: T) {
    let type = Reflect.getMetadata('design:type', target, propertyKey);
    if (!(value instanceof type)) {
      throw new TypeError('Invalid type.');
    }
    set(value);
  };
}
```

> 这个例子来自于TypeScript官方文档，但实际上不能正常执行。因为在经过装饰器处理后，set方法的this将会丢失。但我猜想官方的用意只是展示 `design:type` 的用法。

在这个例子中，我们基于 `Reflect.getMetadata('design:type', target, propertyKey);` 获取到了装饰器对应声明

`design:returntype`（**获取方法返回值类型**）这两种元数据字段来提供帮助。但有一点需要注意，**即使对于基本类型，这些元数据也返回对应的包装类型，如** `number -> [Function: Number]`

IoC

IoC、依赖注入、容器

IoC的全称为 **Inversion of Control**，意为**控制反转**，它是OOP中的一种原则（虽然不在n大设计模式中，但实际上IoC也属于一种设计模式），它可以很好的解耦代码。

在不使用IoC的情况下，我们很容易写出来这样的代码：

```typescript
import { A } from './modA';
import { B } from './modB';

class C {
  constructor() {
    this.a = new A();
    this.b = new B();
  }
}
```

乍一看可能没什么，但实际上类C会强依赖于A、B，造成模块之间的耦合。要解决这个问题，我们可以这么做：用一个第三方容器来负责管理容器，当我们需要某个实例时，由这个容器来替我们实例化并交给我们实例。以 `Injcetion` 为例：

```typescript
import { Container } from 'injection';
import { A } from './A';
import { B } from './B';
const container = new Container();
container.bind(A);
container.bind(B);

class C {
  constructor() {
    this.a = container.get('a');
```

现在A、B、C之间没有了耦合，甚至当某个类D需要使用C的实例时，我们也可以把C交给IoC容器。

我们现在能够知道IoC容器大概的作用了：容器内部维护着一个对象池，管理着各个对象实例，当用户需要使用实例时，容器会自动将对象实例化交给用户。

再举个栗子，当我们想要处对象时，会上Soul、Summer、陌陌...等等去一个个找，找哪种的与怎么找是由我自己决定的，这叫 **控制正转**。现在我觉得有点麻烦，直接把自己的介绍上传到世纪佳缘，如果有人看上我了，就会主动向我发起聊天，这叫 **控制反转**。

DI的全称为**Dependency Injection**，即**依赖注入**。依赖注入是控制反转最常见的一种应用方式，就如它的名字一样，它的思路就是在对象创建时自动注入依赖对象。再以 `Injection` 的使用为例：

```typescript
// provide意为当前对象需要被绑定到容器中
// inject意为去容器中取出对应的实例注入到当前属性中
@provide()
export class UserService {

  @inject()
  userModel;

  async getUser(userId) {
    return await this.userModel.get(userId);
  }
}
```

我们不需要在构造函数中去手动 `this.userModel = xxx` 了，容器会自动帮我们做这一步。

实例: 基于IoC的路由简易实现

我们在最开始介绍了MidwayJS的路由机制，大概长这样：

```typescript
@provide()
@controller('/user')
export class UserController {

  @get('/all')
```

```
  async findUserByUid(): Promise<void> {
    // ...
  }

  @post('/uid/:uid')
  async updateUser(): Promise<void> {
    // ...
  }
}
```

（ `@provide()` 来自于底层的IoC支持 `Injection` ， Midway在应用启动时会去扫描被 `@provide()` 装饰的对象，并装载到容器中，这里不是重点，可以暂且跳过，我们主要关注如何**将装饰器路由解析成路由表**的形式）

我们要解析的路由如下：

typescript

```
@controller('/user')
export class UserController {
  @get('/all')
  async getAllUser(): Promise<void> {
    // ...
  }

  @post('/update')
  async updateUser(): Promise<void> {
    // ...
  }
}
```

首先思考 `controller` 和 `get` / `post` 装饰器，我们需要使用这几个装饰器注入哪些信息：

- **路径**
- **方法（方法装饰器）**

首先是对于整个类，我们需要将 `path: "/user"` 这个数据注入：

typescript

```
// 工具常量枚举
export enum METADATA_MAP {
  METHOD = 'method'
```

```
}

const { METHOD, PATH, GET, POST } = METADATA_MAP;

export const controller = (path: string): ClassDecorator => {
  return (target) => {
    Reflect.defineMetadata(PATH, path, target);
  };
};
```

而后是方法装饰器，我们选择一个高阶函数（柯里化）去吐出各个方法的装饰器，而不是为每种方法定义一个。

typescript

```
// 方法装饰器 保存方法与路径
export const methodDecoCreator = (method: string) => {
  return (path: string): MethodDecorator => {
    return (_target, _key, descriptor) => {
      Reflect.defineMetadata(METHOD, method, descriptor.value!);
      Reflect.defineMetadata(PATH, path, descriptor.value!);
    };
  };
};

// 首先确定方法，而后在使用时才去确定路径
const get = methodDecoCreator(GET);
const post = methodDecoCreator(POST);
```

接下来我们要做的事情就很简单了：

- 拿到注入在类上元数据的根路径
- 拿到每个方法上元数据的方法、路径
- 拼接，生成路由表

typescript

```
const routeGenerator = (ins: Object) => {
  const prototype = Object.getPrototypeOf(ins);

  const rootPath = Reflect.getMetadata(PATH, prototype['constructor']);

  const methods = Object.getOwnPropertyNames(prototype).filter(
```

```
    const path = Reflect.getMetadata(PATH, methodBody);
    const method = Reflect.getMetadata(METHOD, methodBody);
    return {
      path: `${rootPath}${path}`,
      method,
      methodName,
      methodBody,
    };
  });
  console.log(routeGroup);
  return routeGroup;
};
```

生成的结果大概是这样：

typescript

```
[
  {
    path: '/user/all',
    method: 'post',
    methodName: 'getAllUser',
    methodBody: [Function (anonymous)]
  },
  {
    path: '/user/update',
    method: 'get',
    methodName: 'updateUser',
    methodBody: [Function (anonymous)]
  }
]
```

基于这种思路，我们可以很容易的写一个使Koa支持IoC路由的工具。如果你有兴趣，不妨扩展一下。比如说路由还有可能长这样：

typescript

```
@controller('/user', { middleware:[mw1, mw2, ...] })
export class UserController {
  @get('/all', { middleware:[mw11, mw22, ...] })
  async getAllUser(): Promise<void> {
    // ...
```

```
    // ...
  }

  @post('/update')
  async updateUser(): Promise<void> {
    // ...
  }
}
```

新增了几个地方：

- 全局中间件
- 路由级别中间件
- 路由传参

要不要试试整活？

这个例子是否属于IoC机制的体现可能会有争议，但我个人认为 `Reflect Metadata` 的设计本身就是IoC的体现。如果你有别的看法，欢迎在评论区告知我。

依赖注入工具库

我个人了解并使用过的TS依赖注入工具库包括：

- TypeDI，TypeStack出品
- TSYringe，微软出品
- Injection，MidwayJS团队出品，是MidwayJS底层IoC的能力支持

其中 `TypeDI` 也是我日常使用较多的一个，如果你使用基本的Koa开发项目，不妨试一试 `TypeORM + TypeORM-TypeDI-Extensions` 。我们再看看上面呈现过的 `Injection` 的例子：

typescript

```
@provide()
export class UserService {

  @inject()
  userModel;
```

实际上，一个依赖注入工具库必定会提供的就是 **从容器中获取实例** 与 **注入对象到容器中**的两个方法，如上面的 `provide` 与 `inject` ，TypeDI的 `Service` 与 `Inject` 。

总结

读完这篇文章，我想你应该对TypeScript中的装饰器与IoC机制有了大概的了解，如果你意犹未尽，不妨去看一下TypeScript对装饰器、反射元数据的编译结果，见TypeScript Playground。或者，如果你想早点开始了解MidwayJS，在阅读文档的基础上，你也可以瞅瞅我写的这个简单的Demo：Midway-Article-Demo，基于 `Midway` + `TypeORM` + `SQLite3` ，但请注意仍处于雏形，许多Midway的强大能力尚未得到体现，所以不要以这个Demo判定Midway的能力，我会尽快完善这个Demo的。

下一篇，我们会讲解Midway的基础能力，以及对Midway-Serverless：阿里巴巴淘系技术部面对Serverless交出的其中一份答卷的展望。本篇内容可能还是有些枯燥，下一篇我们就会进入欢乐的实战环节啦~

林不渡 Lv2
全干工程师 @ 阿里巴巴-淘系技术...
获得点赞 78 · 获得阅读 3,061

38 4

ssh_晨曦时梦见兮 · 5天前 · Vue.js / TypeScript

TS 4.1 新特性实现 Vuex 无限层级命名空间的 dispatch 类型推断。

50 12

SugarTurboS · 17天前 · JavaScript / TypeScript

你还在为项目里的重复请求发愁吗?

152 65

秋天不落叶 · 11天前 · React.js / TypeScript

多种方式实现自定义 React 路由拦截弹窗

118 25

小姐姐味道 · 11天前 · 前端 / TypeScript

半天掌握TypeScript，感觉就像写Java

45 24

阿里巴巴淘系技术 · 6天前 · TypeScript

TypeScript 核心概念梳理

20 2

ssh_晨曦时梦见兮 · 19天前 · TypeScript / Vue.js

TypeScript 4.1 新特性：字符串模板类型，Vuex 终于有救了?

104 48