

环境配置

装饰器（Decorators）为我们在类的声明及成员上通过编程语法扩展其功能，装饰器以函数的形式声明。

#装饰器类型

可用装饰器包括以下几种

装饰器	说明
ClassDecorator	类装饰器
MethodDecorator	方法装饰器
PropertyDecorator	属性装饰器
ParameterDecorator	参数装饰器

#实验性

Decorators 是实验性的功能，所以开发时会提示错误，我们需要启动 Decorator 这个实验性的功能。

```
1 error TS1219: Experimental support for decorators is a feature that is subject
  to change in a future release. Set the 'experimentalDecorators' option in your
  'tsconfig' or 'jsconfig' to remove this warning.
2
```

首先创建配置文件 tsconfig.js

```
1 tsc --init
2
```

然后开启以下配置项，来启动装饰器这个实验性的功能。

```
1 "experimentalDecorators": true,
2 "emitDecoratorMetadata": true
3
```

然后执行命令，错误就消失了，如果没有 [ts-node \(opens new window\)](#)请先安装

```
1 tsc -W
2
```

比如下面测试都写在`index.ts`，你要定义`index.html`内容如下

```
1 <html>
2 <head>
3   <script src="index.js"></script>
4 </head>
5 </html>
6
```

#类装饰器

类装饰器是对类的功能进行扩展

- 首先执行 `RoleDecorator` 装饰器，然后执行类的构造函数
- 装饰器会优先执行，这与装饰器与类的顺序无关

装饰器参数

首先介绍装饰器函数参数说明

参数	说明
参数一	构造函数

- 普通方法是构造函数的原型对象 `Prototype`
- 静态方法是构造函数

```
1 const MoveDecorator: ClassDecorator = (: Function): void => {
2   console.log(`装饰器 RoleDecorator`);
3 }
4
5 @MoveDecorator
6 class Tank {
7   () {
8     console.log('tank 构造函数');
9   }
}
```

```
10 }  
11
```

即使把装饰器定义放在类的后面也是先执行装饰器

```
1 @MoveDecorator  
2 class Tank {  
3     () {  
4         console.log('tank 构造函数');  
5     }  
6 }  
7  
8 function MoveDecorator(: Function): void {  
9     console.log(`装饰器 RoleDecorator`);  
10 }  
11
```

#原型对象

因为可以在装饰器上得到构造函数，所以可以通过原型对象来添加方法或属性，供实例对象使用

```
1 const MoveDecorator: ClassDecorator = (: Function) => {  
2     constructor.prototype.hd = '后盾人'  
3     constructor.prototype.getPosition = (): { x: number, y: number } => {  
4         return { x: 100, y: 100 }  
5     }  
6 }  
7  
8 @MoveDecorator  
9 class Tank {  
10     () {  
11         console.log('tank 构造函数');  
12     }  
13 }  
14 const tank = new Tank()  
15 console.log(tank.getPosition());  
16
```

不过在编译阶段会提示错误，但这不影响编译生成 js 文件

```
1 Property 'getPosition' does not exist on type 'Tank'
2
```

我们可以通过为类添加默认属性来解决这个错误

```
1 class Tank {
2     public hd: string | undefined
3     public getPosition() { }
4     () {
5         console.log('tank 构造函数');
6     }
7 }
8
```

或者在调用时使用断言处理

```
1 const tank = new Tank()
2 console.log((tank as any).getPosition());
3 //或使用以下方式断言
4 console.log(<any>tank).getPosition());
5
```

#语法糖

不需要把装饰器想的很复杂，下面是同样实现了装饰器的功能。只不过是我们人为调用函数，所以可以把装饰器理解为这种调用的语法糖，这样理解就简单些。

```
1 const MoveDecorator: ClassDecorator = (: Function) => {
2     constructor.prototype.hd = '后盾人'
3     constructor.prototype.getPosition = (): { x: number, y: number } => {
4         return { x: 100, y: 100 }
5     }
6 }
7
8 class Tank {
9     () {
10         console.log('tank 构造函数');
11     }
12 }
```

```

12 }
13
14 MoveDecorator(Tank);
15 const tank = new Tank()
16 console.log(tank.getPosition());
17

```

#装饰器叠加

装饰器可以叠加使用，下面是定义了位置管理与音乐播放装饰器

```

1 //位置控制
2 const MoveDecorator: ClassDecorator = (: Function): void => {
3     constructor.prototype.hd = '后盾人'
4     console.log('MoveDecorator');
5     constructor.prototype.getPosition = (): void => {
6         console.log('获取坐标');
7     }
8 }
9
10 //音乐播放
11 const MusicDecorator: ClassDecorator = (: Function): void => {
12     console.log('MusicDecorator');
13     constructor.prototype.playMusic = (): void => {
14         console.log('播放音乐');
15     }
16 }
17
18 @MoveDecorator
19 @MusicDecorator
20 class Tank {
21     () {
22     }
23 }
24 const tank = new Tank();
25 (<any>tank).playMusic();
26 (<any>tank).getPosition();
27

```

#多类复用

定义好装饰器后，可以为多个类复用，比如下面的玩家与坦克

```
1 //位置控制
2 const MoveDecorator: ClassDecorator = (: Function): void => {
3     constructor.prototype.hd = '后盾人'
4     constructor.prototype.getPosition = (): void => {
5         console.log('获取坐标');
6     }
7 }
8 //音乐播放
9 const MusicDecorator: ClassDecorator = (: Function): void => {
10     constructor.prototype.playMusic = (): void => {
11         console.log('播放音乐');
12     }
13 }
14
15 @MoveDecorator
16 @MusicDecorator
17 class Tank {
18     () {
19     }
20 }
21 const tank = new Tank();
22 (<any>tank).playMusic();
23 (<any>tank).getPosition();
24
25 @MoveDecorator
26 class Player {
27 }
28
29 const xj: Player = new Player();
30 (xj as any).getPosition()
31
```

#响应消息

下面是将网站中的响应消息工作，使用装饰器进行复用。

```

1 //消息响应
2 const MessageDecorator: ClassDecorator = (: Function): void => {
3     constructor.prototype.message = (message: string): void => {
4         document.body.insertAdjacentHTML('afterbegin', `<h2>${message}</h2>`)
5     }
6
7 }
8
9 @MessageDecorator
10 class LoginController {
11     login() {
12         console.log('登录逻辑');
13         this.message('登录成功')
14     }
15 }
16 const controller = new LoginController();
17
18 controller.login()
19

```

#装饰器工厂

有时有需要根据条件返回不同的装饰器，这时可以使用装饰器工厂来解决。可以在类、属性、参数等装饰器中使用装饰器工厂。

下例根据 MusicDecorator 工厂函数传递的不同参数，返回不同装饰器函数。

```

1 const MusicDecorator = (type: string): ClassDecorator => {
2     switch (type) {
3         case 'player':
4             return (: Function) => {
5                 constructor.prototype.playMusic = (): void => {
6                     console.log(`播放【海阔天空】音乐`);
7                 }
8             }
9             break;
10         default:
11             return (: Function) => {
12                 constructor.prototype.playMusic = (): void => {

```

```
13         console.log(`播放【喜洋洋】音乐`);
14     }
15 }
16
17 }
18 }
19
20 @MusicDecorator('tank')
21 class Tank {
22     () {
23     }
24 }
25 const tank = new Tank();
26 (<any>tank).playMusic();
27
28 @MusicDecorator('player')
29 class Player {
30 }
31
32 const xj: Player = new Player();
33 (xj as any).playMusic()
34
```

#方法装饰器

装饰器也可以修改方法，首先介绍装饰器函数参数说明

参数	说明
参数一	普通方法是构造函数的原型对象 Prototype，静态方法是构造函数
参数二	方法名称
参数三	属性描述，如果对这个知识点不清楚，请访问 后盾人 (opens new window) 看向军大叔录制的 js 课程

#基本使用

下面使用 ShowDecorator 装饰来修改 show 方法的实现


```
1  const ShowDecorator: MethodDecorator = (  
2    target: Object,  
3    propertyKey: string | Symbol,  
4    descriptor: PropertyDescriptor,  
5  ): void => {  
6    //对象  
7    console.dir(target)  
8    //方法名  
9    console.dir(propertyKey)  
10   //方法实现  
11   console.dir(descriptor)  
12   descriptor.value = () => {  
13     console.log('houdunren.com')  
14   }  
15 }  
16  
17 class Hd {  
18   @ShowDecorator  
19   show() {  
20     console.log('show method')  
21   }  
22 }  
23  
24 const instance = new Hd()  
25 instance.show()  
26
```

输出结果

```
1  Object  
2  show  
3  Object  
4  houdunren.com  
5
```

下面是修改方法的属性描述 writable 为 false，这时将不允许修改方法。

如果对属性描述知识点不清楚，请访问 [后盾人 \(opens new window\)](#) 看向军大叔录制的 js 课程

```

1  const ShowDecorator: MethodDecorator = (target: Object, propertyKey: string |
    Symbol, descriptor: PropertyDescriptor): void => {
2      descriptor.writable = false
3  }
4
5  class Hd {
6      @ShowDecorator
7      show() {
8          console.log(33);
9      }
10 }
11
12 const instance = new Hd;
13 instance.show()
14
15 //装饰器修改了 writable 描述, 所以不能重写函数
16 instance.show = () => { }
17

```

#静态方法

静态方法使用装饰器与原型方法相似，在处理静态方法时装饰器的第一个参数是构造函数。

```

1  const ShowDecorator: MethodDecorator = (target: Object, propertyKey: string |
    Symbol, descriptor: PropertyDescriptor): void => {
2      descriptor.value = () => {
3          console.log('houdunren.com');
4      }
5  }
6
7  class Hd {
8      @ShowDecorator
9      static show() {
10         console.log('show method');
11     }
12 }
13
14 Hd.show()
15

```

#代码高亮

下面使用装饰器模拟代码高亮

```
1  const highlightDecorator: MethodDecorator = (target: object, propertyKey: any,
    descriptor: PropertyDescriptor): any => {
2      //保存原型方法
3      const method = descriptor.value;
4
5      //重新定义原型方法
6      descriptor.value = () => {
7          return `

${method()}</div>`;
8      }
9  }
10
11 class User {
12     @highlightDecorator
13     response() {
14         return '后盾人 人人做后盾';
15     }
16 }
17
18 console.log(new User().response());
19


```

#延迟执行

下面是延迟执行方法的装饰器，装饰器参数是延迟的时间，达到时间后才执行方法。

```
1  const SleepDecorator: MethodDecorator = (target: Object, propertyKey: string |
    symbol, descriptor: PropertyDescriptor) => {
2      const method = descriptor.value
3      descriptor.value = () => {
4          setTimeout(() => {
5              method()
6          }, 2000)
7      }
8  }
9  class User {
10     @SleepDecorator
```

```

11 public response() {
12     console.log('houdunren.com')
13 }
14 }
15
16 new User().response()
17

```

下面使用装饰器工厂定义延迟时间

```

1  const SleepDecorator =
2    (times: number): MethodDecorator =>
3    (...args: any[]) => {
4      const [, , descriptor] = args
5      const method = descriptor.value
6      descriptor.value = () => {
7        setTimeout(() => {
8          method()
9        }, times)
10     }
11   }
12  class User {
13    @SleepDecorator(0)
14    public response() {
15      console.log('houdunren.com')
16    }
17  }
18
19  new User().response()
20

```

#自定义错误

下面是使用方法装饰器实现自定义错误

- 任何方法使用 @LogErrorDecorator 装饰器都可以实现自定义错误输出

```

1  const ErrorDecorator: MethodDecorator = (target: Object, propertyKey: string |
    Symbol, descriptor: PropertyDescriptor): void => {
2      const method = descriptor.value;

```

```

3     descriptor.value = () => {
4         try {
5             method()
6         } catch (error: any) {
7             // $c 表示 css 样式
8             console.log(`%c后盾人 houdunren.com, 向军大叔`, "color:green; font-size:20px;");
9             console.log(`%c${error.message}`, "color:red;font-size:16px;");
10            console.log(`%c${error.stack}`, `color:blue;font-size:12px;`);
11        }
12    }
13 }
14 }
15
16 class Hd {
17     @ErrorDecorator
18     show() {
19         throw new Error('运行失败')
20     }
21 }
22
23 const instance = new Hd;
24 instance.show()
25

```

对上面的例子使用装饰器工厂来自定义消息内容

```

1  const ErrorDecorator = (message: string, title: string = '后盾人') =>
    <MethodDecorator>(target: Object, propertyKey: string | Symbol, descriptor:
    PropertyDescriptor): void => {
2      const method = descriptor.value;
3      descriptor.value = () => {
4          try {
5              method()
6          } catch (error: any) {
7              console.log(`%c, ${title || `后盾人 houdunren.com`}`, "color:green;
font-size:20px;");
8              console.log(`%c${message || error.message}`, "color:red;font-
size:16px;");
9          }
10     }

```

```

11 }
12
13 class Hd {
14     @ErrorDecorator('Oh! 出错了', '向军大叔')
15     show() {
16         throw new Error('运行失败')
17     }
18 }
19
20 const instance = new Hd;
21 instance.show()
22

```

#登录验证

本例体验装饰器模拟用户登录判断，如果用户的 isLogin 为 false，则跳转到登录页面 `1.login.html`

```

1 //用户资料与登录状态
2 const user = {
3     name: '向军',
4     isLogin: true
5 }
6
7 const AccessDecorator: MethodDecorator = (target: Object, propertyKey: string |
    symbol, descriptor: PropertyDescriptor): void => {
8     const method = descriptor.value;
9     descriptor.value = () => {
10         //登录的用户执行方法
11         if (user.isLogin === true) {
12             return method()
13         }
14         //未登录用户跳转到登录页面
15         alert('你没有访问权限')
16         return location.href = '1.login.html'
17     }
18
19 }
20
21 class Article {

```

```

22     @AccessDecorator
23     show() {
24         console.log('播放视频');
25     }
26
27     @AccessDecorator
28     store() {
29         console.log('保存视频');
30     }
31 }
32
33 new Article().store();
34

```

#权限验证

下面是使用装饰器对用户访问权限的验证

```

1  //用户类型
2  type userType = { name: string, isLogin: boolean, permissions: string[] }
3  //用户数据
4  const user: userType = {
5      name: '向军大叔',
6      isLogin: true,
7      permissions: ['store', 'manage']
8  }
9  //权限验证装饰器工厂
10 const AccessDecorator = (keys: string[]): MethodDecorator => {
11     return (target: Object, propertyKey: string | symbol, descriptor:
PropertyDescriptor) => {
12         const method = descriptor.value
13         const validate = () => keys.every(k => {
14             return user.permissions.includes(k)
15         })
16         descriptor.value = () => {
17             if (user.isLogin === true && validate()) {
18                 alert('验证通过')
19                 return method()
20             }

```

```

21         alert('验证失败')
22         // location.href = 'login.html'
23     }
24 }
25 }
26
27 class Article {
28     show() {
29         console.log('显示文章')
30     }
31     @AccessDecorator(['store', 'manage'])
32     store() {
33         console.log('保存文章')
34     }
35 }
36 new Article().store()
37

```

#网络异步请求

下面是模拟异步请求的示例

```

1  const RequestDecorator = (url: string): MethodDecorator => {
2      return (target: Object, propertyKey: string | symbol, descriptor:
3          PropertyDescriptor) => {
4          const method = descriptor.value
5          // axios.get(url).then()
6          new Promise<any[]>(resolve => {
7              setTimeout(() => {
8                  resolve([
9                      { name: '向军大叔' },
10                     { name: '后盾人' }
11                 ])
12             }, 2000)
13         }).then(users => {
14             method(users)
15         })
16     }
17 }
18
19 class User {
20     @RequestDecorator('
21     https://www.houdunren.com/api/user
22     @RequestDecorator('

```



```
16 public all(users: any[]) {
17     console.log(users)
18 }
19 }
20
```

#属性装饰器

首先介绍装饰器函数参数说明

参数	说明
参数一	普通方法是构造函数的原型对象 Prototype，静态方法是构造函数
参数二	属性名称

#基本使用

下面是属性装饰器的定义方式

```
1 const PropDecorator: PropertyDecorator = (target: Object, propertyKey: string |
  symbol): void => {
2     console.log(target, propertyKey);
3 }
4
5 class Hd {
6     @PropDecorator
7     public name: string | undefined = '后盾人'
8     show() {
9         console.log(33);
10    }
11 }
12
```

#访问器

下面是定义将属性 name 的值转为小写的装饰器

```
1 const PropDecorator: PropertyDecorator = (target: Object, propertyKey: string |
  symbol): void => {
```

```

2    let value: string;
3    const getter = () => {
4        return value
5    }
6    const setter = (v: string) => {
7        value = v.toLowerCase()
8    }
9
10   Object.defineProperty(target, propertyKey, {
11       set: setter,
12       get: getter
13   })
14 }
15
16 class Hd {
17     @PropDecorator
18     public name: string | undefined
19     show() {
20         console.log(33);
21     }
22 }
23
24 const instance = new Hd;
25 instance.name = 'HouDunRen'
26 console.log(instance.name);
27

```

#随机色块

我们使用属性访问器定义随机颜色，并绘制色块，下面是 hd.ts 的内容

```

1  const RandomColorDecorator: PropertyDecorator = (target: Object, propertyKey:
    string | symbol): void => {
2      const colors: string[] = ['red', 'green', 'blue', '#333333'];
3      Object.defineProperty(target, propertyKey, {
4          get: () => {
5              return colors[Math.floor(Math.random() * colors.length)]
6          }
7      })

```

```

8  }
9
10 class Hd {
11     @RandomColorDecorator
12     public color: string | undefined
13
14     public draw() {
15         document.body.insertAdjacentHTML('beforeend', `<div
style="width:200px;height:200px;background-color:${this.color}">houdunren.com 向
军</div>`)
16     }
17 }
18
19 new Hd().draw()
20

```

下面是 hd.html 的模板内容

```

1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="UTF-8" />
5          <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6          <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7          <title>Document</title>
8      </head>
9      <body>
10         <script src="1.js"></script>
11     </body>
12 </html>
13

```

#参数装饰器

可以对方法的参数设置装饰器，参数装饰器的返回值被忽略。

装饰器函数参数说明

参数	说明
参数一	普通方法是构造函数的原型对象 Prototype，静态方法是构造函数

参数二	方法名称
参数三	参数所在索引位置

#基本使用

下面是定义参数装饰器

```
1  const ParameterDecorator: ParameterDecorator = (target: any, propertyKey: string
    | Symbol, parameterIndex: number): void => {
2      console.log(target, propertyKey, parameterIndex);
3  }
4
5  class Hd {
6      show(name: string, @ParameterDecorator url: string) {
7      }
8  }
9
```

#元数据

元数据指对数据的描述，首先需要安装扩展包 [reflect-metadata](#)(opens new window)

```
1  yarn add reflect-metadata
2
```

下面是使用元数据的示例

```
1  //引入支持元数据的扩展名
2  import "reflect-metadata";
3
4  const hd = { name: '向军', city: '北京' }
5  //在对象 hd 的属性 name 上定义元数据（元数据指对数据的描述）
6  Reflect.defineMetadata('xj', 'houdunren.com', hd, 'name')
7
8  let value = Reflect.getMetadata('xj', hd, 'name')
9
10 console.log(value);
11
```

#参数验证

下面是对方法参数的验证，当参数不存在或为 Undefined 时抛出异常。

```
1 //引入支持元数据的扩展名
2 import 'reflect-metadata'
3
4 const requiredMetadataKey = Symbol('required')
5 //哪些参数需要验证，记录参数顺序数字
6 let requiredParameters: number[] = []
7
8 function required(target: Object, propertyKey: string | symbol, parameterIndex:
  number) {
9   //将需要验证的参数索引存入
10   requiredParameters.push(parameterIndex)
11   //在 target 对象的 propertyKey属性上定义元数据 ， 参数为： 键， 值， 对象， 方法
12   Reflect.defineMetadata(requiredMetadataKey, requiredParameters, target,
    propertyKey)
13 }
14
15 const validate: MethodDecorator = (target: object, propertyKey: string | symbol,
  descriptor: PropertyDescriptor) => {
16   const method = descriptor.value
17   descriptor.value = function () {
18     //读取 @required 装饰器定义的元数据
19     let requiredParameters: number[] =
      Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey)
20
21     //如果有值，表示有需要验证的参数
22     if (requiredParameters) {
23       for (const parameterIndex of requiredParameters) {
24         //如果参数不存在或参数值为 undefined 报出错误
25         if (requiredParameters.includes(parameterIndex) &&
          arguments[parameterIndex] === undefined) {
26           throw new Error('验证失败， 参数不能为空。')
27         }
28       }
29     }
30     //验证通过后执行类方法
31     return method.apply(this, arguments)
32 }
```

```
33 }
34
35 class Hd {
36     @validate
37     show(@required name: string, @required id: number) {
38         console.log('验证通过, 执行方法')
39     }
40 }
41
42 const f = new Hd()
43 f.show('后盾人', 18)
44
45 // f.show('后盾人', undefined as any)
46
```

执行命令测试

```
1 ts-node index.ts
```