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。

```
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.

```

首先创建配置文件 tsconfig.js

```
tsc --init

```

然后开启以下配置项，来启动装饰器这个实验性的功能。

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true

```

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

```
tsc -w

```

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

```
<html>
<head>
 <script src="index.js"></script>
</head>
</html>

```

#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

```
const MoveDecorator: ClassDecorator = (: Function): void => {
    console.log(`装饰器 RoleDecorator `);
}

@MoveDecorator
class Tank {
    () {
        console.log('tank 构造函数');
    }
```

```
}

```

即使把装饰器定义放在类的后面也是先执行装饰器

```
@MoveDecorator
class Tank {
    () {
        console.log('tank 构造函数');
    }
}

function MoveDecorator(: Function): void {
    console.log(`装饰器 RoleDecorator `);
}

```

#原型对象

因为可以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

```
const MoveDecorator: ClassDecorator = (: Function) => {
    constructor.prototype.hd = '后盾人'
    constructor.prototype.getPosition = (): { x: number, y: number } => {
        return { x: 100, y: 100 }
    }
}

@MoveDecorator
class Tank {
    () {
        console.log('tank 构造函数');
    }
}
const tank = new Tank()
console.log(tank.getPosition());

```

不过在编译阶段会提示错误，但这不影响编译生成 js 文件

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

我们可以通过为类添加默认属性来解决这个错误

```
class Tank {
    public hd: string | undefined
    public getPosition() { }
    () {
        console.log('tank 构造函数');
    }
}
```

或者在调用时使用断言处理

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

#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。

```
const MoveDecorator: ClassDecorator = (: Function) => {
    constructor.prototype.hd = '后盾人'
    constructor.prototype.getPosition = (): { x: number, y: number } => {
        return { x: 100, y: 100 }
    }
}

class Tank {
    () {
        console.log('tank 构造函数');
    }
```

```
}

MoveDecorator(Tank);
const tank = new Tank()
console.log(tank.getPosition());

```

#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

```
//位置控制
const MoveDecorator: ClassDecorator = (: Function): void => {
    constructor.prototype.hd = '后盾人'
 console.log('MoveDecorator');
    constructor.prototype.getPosition = (): void => {
        console.log('获取坐标');
    }
}

//音乐播放
const MusicDecorator: ClassDecorator = (: Function): void => {
 console.log('MusicDecorator');
    constructor.prototype.playMusic = (): void => {
        console.log('播放音乐');
    }
}

@MoveDecorator
@MusicDecorator
class Tank {
    () {
    }
}
const tank = new Tank();
(<any>tank).playMusic();
(<any>tank).getPosition();

```

#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

```
//位置控制
const MoveDecorator: ClassDecorator = (: Function): void => {
    constructor.prototype.hd = '后盾人'
    constructor.prototype.getPosition = (): void => {
        console.log('获取坐标');
    }
}
//音乐播放
const MusicDecorator: ClassDecorator = (: Function): void => {
    constructor.prototype.playMusic = (): void => {
        console.log('播放音乐');
    }
}

@MoveDecorator
@MusicDecorator
class Tank {
    () {
    }
}
const tank = new Tank();
(<any>tank).playMusic();
(<any>tank).getPosition();

@MoveDecorator
class Player {
}

const xj: Player = new Player();
(xj as any).getPosition()

```

#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。

```
//消息响应
const MessageDecorator: ClassDecorator = (: Function): void => {
    constructor.prototype.message = (message: string): void => {
        document.body.insertAdjacentHTML('afterbegin', `<h2>${message}</h2>`)
    }

}

@MessageDecorator
class LoginController {
    login() {
        console.log('登录逻辑');
        this.message('登录成功')
    }
}
const controller = new LoginController();

controller.login()

```

#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。

```
const MusicDecorator = (type: string): ClassDecorator => {
    switch (type) {
        case 'player':
            return (: Function) => {
                constructor.prototype.playMusic = (): void => {
                    console.log(`播放【海阔天空】音乐`);
                }
            }
            break;
        default:
            return (: Function) => {
                constructor.prototype.playMusic = (): void => {
```

```
                    console.log(`播放【喜洋洋】音乐`);
                }
            }

    }
}

@MusicDecorator('tank')
class Tank {
    () {
    }
}
const tank = new Tank();
(<any>tank).playMusic();

@MusicDecorator('player')
class Player {
}

const xj: Player = new Player();
(xj as any).playMusic()

```

#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

```
const ShowDecorator: MethodDecorator = (
  target: Object,
  propertyKey: string | Symbol,
  descriptor: PropertyDescriptor,
): void => {
  //对象
  console.dir(target)
  //方法名
  console.dir(propertyKey)
  //方法实现
  console.dir(descriptor)
  descriptor.value = () => {
    console.log('houdunren.com')
  }
}

class Hd {
  @ShowDecorator
  show() {
    console.log('show method')
  }
}

const instance = new Hd()
instance.show()

```

输出结果

```
Object
show
Object
houdunren.com

```

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

```

```

```
const ShowDecorator: MethodDecorator = (target: Object, propertyKey: string | Symbol, descriptor: PropertyDescriptor): void => {
    descriptor.writable = false
}

class Hd {
    @ShowDecorator
    show() {
        console.log(33);
    }
}

const instance = new Hd;
instance.show()

//装饰器修改了 writable 描述，所以不能重写函数
instance.show = () => { }

```

#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。

```
const ShowDecorator: MethodDecorator = (target: Object, propertyKey: string | Symbol, descriptor: PropertyDescriptor): void => {
    descriptor.value = () => {
        console.log('houdunren.com');
    }
}

class Hd {
    @ShowDecorator
    static show() {
        console.log('show method');
    }
}

Hd.show()

```

#代码高亮

下面使用装饰器模拟代码高亮

```
const highlightDecorator: MethodDecorator = (target: object, propertyKey: any, descriptor: PropertyDescriptor): any => {
                //保存原型方法
    const method = descriptor.value;

                //重新定义原型方法
    descriptor.value = () => {
        return `<div style="color:red">${method()}</div>`
    }
}

class User {
    @highlightDecorator
    response() {
        return '后盾人 人人做后盾';
    }
}

console.log(new User().response());

```

#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。

```
const SleepDecorator: MethodDecorator = (target: Object, propertyKey: string | symbol, descriptor: PropertyDescriptor) => {
  const method = descriptor.value
  descriptor.value = () => {
    setTimeout(() => {
      method()
    }, 2000)
  }
}
class User {
  @SleepDecorator
```

```
  public response() {
    console.log('houdunren.com')
  }
}

new User().response()

```

下面使用装饰器工厂定义延迟时间

```
const SleepDecorator =
  (times: number): MethodDecorator =>
  (...args: any[]) => {
    const [, , descriptor] = args
    const method = descriptor.value
    descriptor.value = () => {
      setTimeout(() => {
        method()
      }, times)
    }
  }
class User {
  @SleepDecorator(0)
  public response() {
    console.log('houdunren.com')
  }
}

new User().response()

```

#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

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

```
    descriptor.value = () => {
        try {
            method()
        } catch (error: any) {
            //$c 表示 css 样式
            console.log(`%c后盾人 houdunren.com，向军大叔`, "color:green; font-size:20px;");
            console.log(`%c${error.message}`, "color:red;font-size:16px;");
            console.log(`%c${error.stack}`, `color:blue;font-size:12px;`);

        }
    }
}

class Hd {
    @ErrorDecorator
    show() {
        throw new Error('运行失败')
    }
}

const instance = new Hd;
instance.show()

```

对上面的例子使用装饰器工厂来自定义消息内容

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

```
}

class Hd {
    @ErrorDecorator('Oh! 出错了', '向军大叔')
    show() {
        throw new Error('运行失败')
    }
}

const instance = new Hd;
instance.show()

```

#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`

```
//用户资料与登录状态
const user = {
    name: '向军',
    isLogin: true
}

const AccessDecorator: MethodDecorator = (target: Object, propertyKey: string | symbol, descriptor: PropertyDescriptor): void => {
    const method = descriptor.value;
    descriptor.value = () => {
        //登录的用户执行方法
        if (user.isLogin === true) {
            return method()
        }
        //未登录用户跳转到登录页面
        alert('你没有访问权限')
        return location.href = '1.login.html'
    }

}

class Article {
```

```
    @AccessDecorator
    show() {
        console.log('播放视频');
    }

    @AccessDecorator
    store() {
        console.log('保存视频');
    }
}

new Article().store();

```

#权限验证

下面是使用装饰器对用户访问权限的验证

```
//用户类型
type userType = { name: string, isLogin: boolean, permissions: string[] }
//用户数据
const user: userType = {
    name: '向军大叔',
    isLogin: true,
    permissions: ['store', 'manage']
}
//权限验证装饰器工厂
const AccessDecorator = (keys: string[]): MethodDecorator => {
    return (target: Object, propertyKey: string | symbol, descriptor: PropertyDescriptor) => {
        const method = descriptor.value
        const validate = () => keys.every(k => {
            return user.permissions.includes(k)
        })
        descriptor.value = () => {
            if (user.isLogin === true && validate()) {
                alert('验证通过')
                return method()
            }
```

```
            alert('验证失败')
            // location.href = 'login.html'
        }
    }
}

class Article {
    show() {
        console.log('显示文章')
    }
    @AccessDecorator(['store', 'manage'])
    store() {
        console.log('保存文章')
    }
}
new Article().store()

```

#网络异步请求

下面是模拟异步请求的示例

```
const RequestDecorator = (url: string): MethodDecorator => {
  return (target: Object, propertyKey: string | symbol, descriptor: PropertyDescriptor) => {
    const method = descriptor.value
    // axios.get(url).then()
    new Promise<any[]>(resolve => {
      setTimeout(() => {
        resolve([{ name: '向军大叔' }, { name: '后盾人' }])
      }, 2000)
    }).then(users => {
      method(users)
    })
  }
}
class User {
  @RequestDecorator('https://www.houdunren.com/api/user
  @RequestDecorator('
```

```
  public all(users: any[]) {
    console.log(users)
  }
}

```

#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

```
const PropDecorator: PropertyDecorator = (target: Object, propertyKey: string | symbol): void => {
    console.log(target, propertyKey);
}

class Hd {
    @PropDecorator
    public name: string | undefined = '后盾人'
    show() {
        console.log(33);
    }
}

```

#访问器

下面是定义将属性 name 的值转为小写的装饰器

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

```
    let value: string;
    const getter = () => {
        return value
    }
    const setter = (v: string) => {
        value = v.toLowerCase()
    }

    Object.defineProperty(target, propertyKey, {
        set: setter,
        get: getter
    })
}

class Hd {
    @PropDecorator
    public name: string | undefined
    show() {
        console.log(33);
    }
}

const instance = new Hd;
instance.name = 'HouDunRen'
console.log(instance.name);

```

#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

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

```
}

class Hd {
    @RandomColorDecorator
    public color: string | undefined

    public draw() {
        document.body.insertAdjacentHTML('beforeend', `<div style="width:200px;height:200px;background-color:${this.color}">houdunren.com 向军</div>`)
    }
}

new Hd().draw()

```

下面是 hd.html 的模板内容

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Document</title>
    </head>
    <body>
        <script src="1.js"></script>
    </body>
</html>

```

#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

```
const ParameterDecorator: ParameterDecorator = (target: any, propertyKey: string | Symbol, parameterIndex: number): void => {
    console.log(target, propertyKey, parameterIndex);
}

class Hd {
    show(name: string, @ParameterDecorator url: string) {
    }
}

```

#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)

```
yarn add reflect-metadata

```

下面是使用元数据的示例

```
//引入支持元数据的扩展名
import "reflect-metadata";

const hd = { name: '向军', city: '北京' }
//在对象 hd 的属性 name 上定义元数据（元数据指对数据的描述）
Reflect.defineMetadata('xj', 'houdunren.com', hd, 'name')

let value = Reflect.getMetadata('xj', hd, 'name')

console.log(value);

```

#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。

```
//引入支持元数据的扩展名
import 'reflect-metadata'

const requiredMetadataKey = Symbol('required')
//哪些参数需要验证，记录参数顺序数字
let requiredParameters: number[] = []

function required(target: Object, propertyKey: string | symbol, parameterIndex: number) {
  //将需要验证的参数索引存入
  requiredParameters.push(parameterIndex)
  //在 target 对象的 propertyKey属性上定义元素数据 ，参数为：键，值，对象，方法
  Reflect.defineMetadata(requiredMetadataKey, requiredParameters, target, propertyKey)
}

const validate: MethodDecorator = (target: object, propertyKey: string | symbol, descriptor: PropertyDescriptor) => {
  const method = descriptor.value
  descriptor.value = function () {
    //读取 @required 装饰器定义的元数据
    let requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey)

    //如果有值，表示有需要验证的参数
    if (requiredParameters) {
      for (const parameterIndex of requiredParameters) {
        //如果参数不存在或参数值为 undefined 报出错误
        if (requiredParameters.includes(parameterIndex) && arguments[parameterIndex] === undefined) {
          throw new Error('验证失败，参数不能为空.')
        }
      }
    }
    //验证通过后执行类方法
    return method.apply(this, arguments)
  }
```

```
}

class Hd {
  @validate
  show(@required name: string, @required id: number) {
    console.log('验证通过，执行方法')
  }
}

const f = new Hd()
f.show('后盾人', 18)

// f.show('后盾人', undefined as any)

```

执行命令测试

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