装饰器

一、简介

1. 装饰器本质是一种特殊的函数，它可以对：类、属性、方法、参数进行扩展，同时能让代码更简洁。
2. 装饰器自 `2015` 年在 `ECMAScript-6` 中被提出到现在，已将近10年。

3. 截止目前，装饰器依然是实验性特性 ， 需要开发者手动调整配置，来开启装饰器支持。
4. 装饰器有 5 种：

   1️⃣类装饰器

   2️⃣属性装饰器

   3️⃣方法装饰器

   4️⃣访问器装饰器

   5️⃣参数装饰器

> 备注：虽然 `TypeScript5.0` 中可以直接使用 **`类装饰器`** ，但为了确保其他装饰器可用，现阶段使用时，仍建议使用 `experimentalDecorators` 配置来开启装饰器支持，而且不排除在来的版本中，官方会**进一步调整**装饰器的相关语法！
>
> 参考： 《TypeScript 5.0发版公告》

二、类装饰器

1. 基本语法

类装饰器是一个应用在**类声明**上的**函数**，可以为类添加额外的功能，或添加额外的逻辑。

基本语法 TypeScript

```typescript
/*
  Demo函数会在Person类定义时执行
  参数说明:
    ○ target参数是被装饰的类，即: Person
*/
function Demo(target: Function) {
  console.log(target)
}

// 使用装饰器
@Demo
class Person { }
```

2. 应用举例

需求：定义一个装饰器，实现 `Person` 实例调用 `toString` 时返回 `JSON.stringify` 的执行结果。

应用举例 TypeScript

```typescript
// 使用装饰器重写toString方法 + 封闭其原型对象
function CustomString(target: Function) {
  // 向被装饰类的原型上添加自定义的 toString 方法
  target.prototype.toString = function () {
    return JSON.stringify(this)
  }
  // 封闭其原型对象，禁止随意操作其原型对象
  Object.seal(target.prototype)
}

// 使用 CustomString 装饰器
@CustomString
class Person {
  constructor(public name: string, public age: number) { }
  speak() {
    console.log('你好呀! ')
  }
}

/* 测试代码如下 */
let p1 = new Person('张三', 18)
// 输出：{"name":"张三","age":18}
console.log(p1.toString())
// 禁止随意操作其原型对象
interface Person {
  a: any
}
// Person.prototype.a = 100 // 此行会报错：Cannot add property a, object is not extensible
// console.log(p1.a)
```

3. 关于返回值

类装饰器有返回值：若类装饰器返回一个新的类，那这个新类将**替换**掉被装饰的类。

类装饰器无返回值：若类装饰器无返回值或返回 `undefined` ，那被装饰的类**不会**被替换。

```
function demo(target:Function){
  // 装饰器有返回值时，该返回值会替换掉被装饰的类
  return class {
    test(){
      console.log(200)
      console.log(300)
      console.log(400)
    }
  }
}

@demo
class Person {
  test(){
    console.log(100)
  }
}

console.log(Person)
```

4. 关于构造类型

> 在 TypeScript 中， `Function` 类型所表示的范围十分广泛，包括：普通函数、箭头函数、方法等等。但并非 `Function` 类型的函数都可以被 `new` 关键字实例化，例如箭头函数是不能被实例化的，那么 TypeScript 中概如何声明一个构造类型呢？有以下两种方式：

仅声明构造类型 TypeScript

```typescript
/*
  ○ new      表示：该类型是可以用new操作符调用。
  ○ ...args  表示：构造器可以接受【任意数量】的参数。
  ○ any[]    表示：构造器可以接受【任意类型】的参数。
  ○ {}       表示：返回类型是对象(非null、非undefined的对象)。
*/

// 定义Constructor类型，其含义是构造类型
type Constructor = new (...args: any[]) => {};

function test(fn:Constructor){}
class Person {}
test(Person)
```

声明构造类型 + 指定静态属性 TypeScript

```typescript
// 定义一个构造类型，且包含一个静态属性 wife
type Constructor = {
  new(...args: any[]): {}; // 构造签名
  wife: string; // wife属性
};

function test(fn:Constructor){}
class Person {
  static wife = 'asd'
}
test(Person)
```


5. 替换被装饰的类

对于高级一些的装饰器，不仅仅是覆盖一个原型上的方法，还要有更多功能，例如添加新的方法和状态。

需求：设计一个 `LogTime` 装饰器，可以给实例添加一个属性，用于记录实例对象的创建时间，再添加一个方法用于读取创建时间。

```ts
// User接口
interface User {
  getTime(): Date
  log(): void
}

// 自定义类型Class
type Constructor = new (...args: any[]) => {}

// 创建一个装饰器，为类添加日志功能和创建时间
function LogTime<T extends Constructor>(target: T) {
  return class extends target {
    createdTime: Date;
    constructor(...args: any[]) {
      super(...args);
      this.createdTime = new Date(); // 记录对象创建时间
    }
    getTime() {
      return `该对象创建时间为: ${this.createdTime}`;
    }
  };
}

@LogTime
class User {
  constructor(
    public name: string,
    public age: number
  ) { }
  speak() {
    console.log(`${this.name}说: 你好啊! `)
  }
}

const user1 = new User('张三', 13);
user1.speak()
console.log(user1.getTime())
```

三、装饰器工厂

装饰器工厂是一个返回装饰器函数的函数，可以为装饰器添加参数，可以更灵活地控制装饰器的行为。

需求：定义一个 `LogInfo` 类装饰器工厂，实现 `Person` 实例可以调用到 `introduce` 方法，且 `introduce` 中输出内容的次数，由 `LogInfo` 接收的参数决定。

```
interface Person {
  introduce: () => void
}

// 定义一个装饰器工厂 LogInfo，它接受一个参数 n，返回一个类装饰器
function LogInfo(n:number) {
  // 装饰器函数，target 是被装饰的类
  return function(target: Function){
    target.prototype.introduce = function () {
      for (let i = 0; i < n; i++) {
        console.log(`我的名字：${this.name}，我的年龄：${this.age}`)
      }
    }
  }
}

@LogInfo(5)
class Person {
  constructor(
    public name: string,
    public age: number
  ) { }
  speak() {
    console.log('你好呀！')
  }
}

let p1 = new Person('张三', 18)
// console.log(p1) // 打印的p1是：_classThis，转换的JS版本比较旧时，会出现，不必纠结
p1.speak()
p1.introduce()
```

四、装饰器组合

装饰器可以组合使用，执行顺序为：先【由上到下】的执行所有的装饰器工厂，依次获取到装饰器，然后再【由下到上】执行所有的装饰器。

装饰器组合 —— 执行顺序 TypeScript

```typescript
//装饰器
function test1(target:Function) {
  console.log('test1')
}
//装饰器工厂
function test2() {
  console.log('test2工厂')
  return function (target:Function) {
    console.log('test2')
  }
}
//装饰器工厂
function test3() {
  console.log('test3工厂')
  return function (target:Function) {
    console.log('test3')
  }
}
//装饰器
function test4(target:Function) {
  console.log('test4')
}

@test1
@test2()
@test3()
@test4
class Person { }

/*
  控制台打印:
    test2工厂
    test3工厂
    test4
    test3
    test2
    test1
*/
```

装饰器组合 —— 应用 TypeScript

```typescript
// 自定义类型Class
type Constructor = new (...args: any[]) => {}

interface Person {
  introduce():void
  getTime():void
}

// 使用装饰器重写toString方法 + 封闭其原型对象
function customToString(target: Function) {
  // 向被装饰类的原型上添加自定义的 toString 方法
  target.prototype.toString = function () {
    return JSON.stringify(this)
  }
  // 封闭其原型对象，禁止随意操作其原型对象
  Object.seal(target.prototype)
}

// 创建一个装饰器，为类添加日志功能和创建时间
function LogTime<T extends Constructor>(target: T) {
  return class extends target {
    createdTime: Date;
    constructor(...args: any[]) {
      super(...args);
      this.createdTime = new Date(); // 记录对象创建时间
    }
    getTime() {
      return `该对象创建时间为: ${this.createdTime}`;
    }
  };
}

// 定义一个装饰器工厂 LogInfo，它接受一个参数 n，返回一个类装饰器
function LogInfo(n:number) {
  // 装饰器函数，target 是被装饰的类
  return function(target: Function){
    target.prototype.introduce = function () {
      for (let i = 0; i < n; i++) {
        console.log(`我的名字: ${this.name}, 我的年龄: ${this.age}`)
      }
    }
  }
}

@customToString
```

```
@LogInfo(3)
@LogTime
class Person {
  constructor(
    public name: string,
    public age: number
  ) { }
  speak() {
    console.log('你好呀! ')
  }
}

const p1 = new Person('张三',18)
console.log(p1.toString())
p1.introduce()
console.log(p1.getTime())
```

五、属性装饰器

1. 基本语法

```
/*
  参数说明:
    ○ target: 对于静态属性来说值是类，对于实例属性来说值是类的原型对象。
    ○ propertyKey: 属性名。
*/
function Demo(target: object, propertyKey: string) {
  console.log(target,propertyKey)
}

class Person {
  @Demo name: string
  @Demo age: number
  @Demo static school:string

  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }
}

const p1 = new Person('张三', 18)
```

2. 关于属性遮蔽

> 如下代码中：当构造器中的 `this.age = age` 试图在实例上赋值时，实际上是调用了原型上 `age` 属性的 `set` 方法。

```
class Person {
  name: string
  age: number
  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }
}

let value = 99
// 使用defineProperty给Person原型添加age属性，并配置对应的get与set
Object.defineProperty(Person.prototype, 'age', {
  get() {
    return value
  },
  set(val) {
    value = val
  }
})

const p1 = new Person('张三', 18)
console.log(p1.age) //18
console.log(Person.prototype.age)//18
```

在 JavaScript（以及其他一些语言）中，当在一个内部作用域（如函数内部）定义了一个与外部作用域中同名的变量时，内部作用域中的变量会“遮蔽”外部作用域中的同名变量。这意味着在内部作用域中，对这个变量的引用和操作只会影响内部定义的变量，而不会影响外部的同名变量。

3. 应用举例

需求：定义一个 `State` 属性装饰器，来监视属性的修改。

```ts
// 声明一个装饰器函数 State，用于捕获数据的修改
function State(target: object, propertyKey: string) {
  // 存储属性的内部值
  let key = `__${propertyKey}`;

  // 使用 Object.defineProperty 替换类的原始属性
  // 重新定义属性，使其使用自定义的 getter 和 setter
  Object.defineProperty(target, propertyKey, {
    get () {
      return this[key]
    },
    set(newVal: string){
      console.log(`${propertyKey}的最新值为：${newVal}`);
      this[key] = newVal
    },
    enumerable: true,   枚举
    configurable: true, 遍历
  });
}

class Person {
  name: string;
  //使用State装饰器
  @State age: number;
  school = 'atguigu';
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
}

const p1 = new Person('张三', 18);
const p2 = new Person('李四', 30);

p1.age = 80
p2.age = 90

console.log('------------------')
console.log(p1.age) //80
console.log(p2.age) //90
```

六、方法装饰器

1. 基本语法

```
/*
  参数说明:
    ○ target: 对于静态方法来说值是类，对于实例方法来说值是原型对象。
    ○ propertyKey:方法的名称。
    ○ descriptor: 方法的描述对象，其中value属性是被装饰的方法。
*/
function Demo(target: object, propertyKey: string, descriptor: PropertyDescrip
tor){
  console.log(target)
  console.log(propertyKey)
  console.log(descriptor)
}

class Person {
  constructor(
    public name:string,
    public age:number,
  ){}
  // Demo装饰实例方法
  @Demo speak(){
    console.log(`你好，我的名字: ${this.name}，我的年龄: ${this.age}`)
  }
  // Demo装饰静态方法
  @Demo static isAdult(age:number) {
    return age >= 18;
  }
}

const p1 = new Person('张三',18)
p1.speak()
```

2. 应用举例

需求:

1. 定义一个 `Logger` 方法装饰器，用于在方法执行前和执行后，均追加一些额外逻辑。
2. 定义一个 `Validate` 方法装饰器，用于验证数据。

```ts
function Logger(target: object, propertyKey: string, descriptor: PropertyDescr
iptor){
  // 保存原始方法
  const original = descriptor.value;
  // 替换原始方法
  descriptor.value = function (...args:any[]) {
    console.log(`${propertyKey}开始执行.....`)
    const result = original.call(this, ...args)
    console.log(`${propertyKey}执行完毕.....`)
    return result;
  }
}

function Validate(maxValue:number){
  return function (target: object, propertyKey: string, descriptor: PropertyDe
scriptor){
    // 保存原始方法
    const original = descriptor.value;
    // 替换原始方法
    descriptor.value = function (...args: any[]) {
      // 自定义的验证逻辑
      if (args[0] > maxValue) {
        throw new Error('年龄非法! ')
      }
      // 如果所有参数都符合要求，则调用原始方法
      return original.apply(this, args);
    };
  }
}

class Person {
  constructor(
    public name:string,
    public age:number,
  ){}
  @Logger speak(){
    console.log(`你好，我的名字：${this.name}，我的年龄：${this.age}`)
  }
  @Validate(120)
  static isAdult(age:number) {
    return age >= 18;
  }
}

const p1 = new Person('张三',18)
p1.speak()
console.log(Person.isAdult(100))
```

七、访问器装饰器

1. 基本语法

```
/*
  参数说明:
    ○ target:
        1. 对于实例访问器来说值是【所属类的原型对象】。
        2. 对于静态访问器来说值是【所属类】。
    ○ propertyKey:访问器的名称。
    ○ descriptor: 描述对象。
*/
function Demo(target: object, propertyKey: string, descriptor: PropertyDescrip
tor) {
  console.log(target)
  console.log(propertyKey)
  console.log(descriptor)
}

class Person {
  @Demo
  get address(){
    return '北京宏福科技园'
  }
  @Demo
  static get country(){
    return '中国'
  }
}
```

2. 应用举例

需求：对 `Weather` 类的 `temp` 属性的 `set` 访问器进行限制，设置的最低温度 `-50` ，最高温度 `50`

```ts
function RangeValidate(min: number, max: number) {
  return function (target: object, propertyKey: string, descriptor: PropertyDescriptor) {
    // 保存原始的 setter 方法，以便在后续调用中使用
    const originalSetter = descriptor.set;

    // 重写 setter 方法，加入范围验证逻辑
    descriptor.set = function (value: number) {
      // 检查设置的值是否在指定的最小值和最大值之间
      if (value < min || value > max) {
        // 如果值不在范围内，抛出错误
        throw new Error(`${propertyKey}的值应该在 ${min} 到 ${max}之间! `);
      }

      // 如果值在范围内，且原始 setter 方法存在，则调用原始 setter 方法
      if (originalSetter) {
        originalSetter.call(this, value);
      }
    };
  };
}

class Weather {
  private _temp: number;
  constructor(_temp: number) {
    this._temp = _temp;
  }
  // 设置温度范围在 -50 到 50 之间
  @RangeValidate(-50,50)
  set temp(value) {
    this._temp = value;
  }
  get temp() {
    return this._temp;
  }
}

const w1 = new Weather(25);
console.log(w1)
w1.temp = 67
console.log(w1)
```


八、参数装饰器

1. 基本语法

```
/*
  参数说明:
    ○ target:
      1.如果修饰的是【实例方法】的参数，target 是类的【原型对象】。
      2.如果修饰的是【静态方法】的参数，target 是【类】。
    ○ propertyKey: 参数所在的方法的名称。
    ○ parameterIndex: 参数在函数参数列表中的索引，从 0 开始。
*/
function Demo(target: object, propertyKey: string, parameterIndex: number) {
  console.log(target)
  console.log(propertyKey)
  console.log(parameterIndex)
}

// 类定义
class Person {
  constructor(public name: string) { }
  speak(@Demo message1: any, mesage2: any) {
    console.log(`${this.name}想对说: ${message1}, ${mesage2}`);
  }
}
```

2. 应用举例

需求：定义方法装饰器 `Validate`，同时搭配参数装饰器 `NotNumber`，来对 `speak` 方法的参数类型进行限制。

```ts
function NotNumber(target: any, propertyKey: string, parameterIndex: number) {
  // 初始化或获取当前方法的参数索引列表
  let notNumberArr: number[] = target[`__notNumber_${propertyKey}`] || [];
  // 将当前参数索引添加到列表中
  notNumberArr.push(parameterIndex);
  // 将列表存储回目标对象
  target[`__notNumber_${propertyKey}`] = notNumberArr;
}

// 方法装饰器定义
function Validate(target: any, propertyKey: string, descriptor: PropertyDescri
ptor) {
  const method = descriptor.value;
  descriptor.value = function (...args: any[]) {
    // 获取被标记为不能为空的参数索引列表
    const notNumberArr: number[] = target[`__notNumber_${propertyKey}`] || [];
    // 检查参数是否为 null 或 undefined
    for (const index of notNumberArr) {
      if (typeof args[index] === 'number') {
        throw new Error(`方法 ${propertyKey} 中索引为 ${index} 的参数不能是数字! `)
      }
    }
    // 调用原始方法
    return method.apply(this, args);
  };

  return descriptor;
}

// 类定义
class Student {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  @Validate
  speak(@NotNumber message1: any, mesage2: any) {
    console.log(`${this.name}想对说: ${message1}, ${mesage2}`);
  }
}

// 使用
const s1 = new Student("张三");
s1.speak(100, 200);
```