25 入手装饰器，给凡人添加超能力

更新时间：2019-07-05 10:07:10

> 人的差异在于业余时间。
>
> ——爱因斯坦

ECMAScript 的装饰器提案到现在还没有定案，所以我们直接看 TS 中的装饰器。同样在 TS 中，装饰器仍然是一项实验性特性，未来可能有所改变，所以如果你要使用装饰器，需要在 tsconfig.json 的编译配置中开启 experimentalDecorators，将它设为 true。

3.12.1. 基础

(1) 装饰器定义

装饰器是一种新的声明，它能够作用于**类声明、方法、访问符、属性和参数**上。使用 @ 符号加一个名字来定义，如 @decorat，这的 decorat 必须是一个函数或者求值后是一个函数，这个 decorat 命名不是写死的，是你自己定义的，这个函数在运行的时候被调用，被装饰的声明作为参数会自动传入。要注意**装饰器要紧挨着要修饰的内容的前面**，而且所有的装饰器不能用在声明文件(.d.ts)中，和任何外部上下文中（比如 declare，关于.d.ts 和 declare，我们都会在讲声明文件一课时学习）。比如下面的这个函数，就可以作为装饰器使用：

```
function setProp (target) {
  // ...
}
@setProp
```

先定义一个函数，然后这个函数有一个参数，就是要装饰的目标，装饰的作用不同，这个target代表的东西也不同，下面我们具体讲的时候会讲。定义了这个函数之后，它就可以作为装饰器，使用 @函数名 的形式，写在要装饰的内容前面。

(2) 装饰器工厂

装饰器工厂也是一个函数，它的返回值是一个函数，返回的函数作为装饰器的调用函数。如果使用装饰器工厂，那么在使用的时候，就要加上函数调用，如下：

```
function setProp () {
    return function (target) {
        // ...
    }
}

@setProp()
```

(3) 装饰器组合

装饰器可以组合，也就是对于同一个目标，引用多个装饰器：

```
// 可以写在一行
@setName @setAge target
// 可以换行
@setName
@setAge
target
```

但是这里要格外注意的是，多个装饰器的执行顺序：

- 装饰器工厂从上到下依次执行，但是只是用于返回函数但不调用函数；
- 装饰器函数从下到上依次执行，也就是执行工厂函数返回的函数。

我们以下面的两个装饰器工厂为例：

```
function setName () {
    console.log('get setName')
    return function (target) {
        console.log('setName')
    }
}
function setAge () {
    console.log('get setAge')
    return function (target) {
        console.log('setAge')
    }
}
@setName()
@setAge()
class Test {}
// 打印出来的内容如下：
/**
 'get setName'
 'get setAge'
 'setAge'
 'setName'
*/
```

可以看到，多个装饰器，会先执行装饰器工厂函数获取所有装饰器，然后再从后往前执行装饰器的逻辑。

(4) 装饰器求值

类的定义中不同声明上的装饰器将按以下规定的顺序引用：

1. 参数装饰器，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，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。

3.12.2. 类装饰器

*类装饰器*在类声明之前声明，要记着装饰器要紧挨着要修饰的内容，类装饰器应用于类的声明。

类装饰器表达式会在运行时当做函数被调用，它由唯一一个参数，就是装饰的这个类。

```
let sign = null;
function setName(name: string) {
  return function(target: Function) {
    sign = target;
    console.log(target.name);
  };
}
@setName("lison") // Info
class Info {
  constructor() {}
}
console.log(sign === Info); // true
console.log(sign === Info.prototype.constructor); // true
```

可以看到，我们在装饰器里打印出类的 name 属性值，也就是类的名字，我们没有使用 Info 创建实例，控制台也打印了"Info"，因为装饰器作用与装饰的目标声明时。而且我们将装饰器里获取的参数 target 赋值给 sign，最后判断 sign 和定义的类 Info 是不是相等，如果相等说明它们是同一个对象，结果是 true。而且类 Info 的原型对象的 constructor 属性指向的其实就是 Info 本身。

通过装饰器，我们就可以修改类的原型对象和构造函数：

```
function addName(constructor: { new (): any }) {
  constructor.prototype.name = "lison";
}
@addName
class A {}
const a = new A();
console.log(a.name); // error 类型“A”上不存在属性“name”
```

上面例子中，我们通过 addName 修饰符可以在类 A 的原型对象上添加一个 name 属性，这样使用 A 创建的实例，应该可以继承这个 name 属性，访问实例对象的 name 属性应该返回"lison"，但是这里报错，是因为我们定义的类 A 并没有定义属性 name，所以我们可以定义一个同名接口，通过声明合并解决这个问题：

```
function addName(constructor: { new (): any }) {
  constructor.prototype.name = "lison";
}
@addName
class A {}
interface A {
  name: string;
}
const a = new A();
console.log(a.name); // "lison"
```

如果类装饰器返回一个值，那么会使用这个返回的值替换被装饰的类的声明，所以我们可以使用此特性修改类的实现。但是要注意的是，我们需要自己处理原有的原型链。我们可以通过装饰器，来覆盖类里一些操作，来看官方的这个例子：

```
function classDecorator<T extends { new (...args: any[]): {} }>(target: T) {
  return class extends target {
    newProperty = "new property";
    hello = "override";
  };
}
@classDecorator
class Greeter {
  property = "property";
  hello: string;
  constructor(m: string) {
    this.hello = m;
  }
}
console.log(new Greeter("world"));
/*
{
    hello: "override"
    newProperty: "new property"
    property: "property"
}
*/
```

首先我们定义了一个装饰器，它返回一个类，这个类继承要修饰的类，所以最后创建的实例不仅包含原 Greeter 类中定义的实例属性，还包含装饰器中定义的实例属性。还有一个点，我们在装饰器里给实例添加的属性，设置的属性值会覆盖被修饰的类里定义的实例属性，所以我们创建实例的时候虽然传入了字符串，但是 hello 还是装饰器里设置的"override"。我们把这个例子改一下：

```
function classDecorator(target: any): any {
  return class {
    newProperty = "new property";
    hello = "override";
  };
}
@classDecorator
class Greeter {
  property = "property";
  hello: string;
  constructor(m: string) {
    this.hello = m;
  }
}
console.log(new Greeter("world"));
/*
{
    hello: "override"
    newProperty: "new property"
}
*/
```

在这个例子中，我们装饰器的返回值还是返回一个类，但是这个类不继承被修饰的类了，所以最后打印出来的实例，只包含装饰器中返回的类定义的实例属性，被装饰的类的定义被替换了。

如果我们的类装饰器有返回值，但返回的不是一个构造函数（类），那就会报错了。

3.12.3. 方法装饰器

方法装饰器用来处理类中方法，它可以处理方法的属性描述符，可以处理方法定义。方法装饰器在运行时也是被当做函数调用，含 3 个参数：

- 装饰静态成员时是类的构造函数，装饰实例成员时是类的原型对象；
- 成员的名字；
- 成员的属性描述符。

讲到这里，我们先补充个 JS 的知识——属性描述符。对象可以设置属性，如果属性值是函数，那这个函数称为方法。每一个属性和方法在定义的时候，都伴随三个属性描述符 configurable、writable 和 enumerable，分别用来描述这个属性的可配置性、可写性和可枚举性。这三个描述符，需要使用 ES5 才有的 Object.defineProperty 方法来设置，我们来看下如何使用：

```
var obj = {};
Object.defineProperty(obj, "name", {
  value: "lison",
  writable: false,
  configurable: true,
  enumerable: true
});
console.log(obj);
// { name: 'lison' }
obj.name = "test";
console.log(obj);
// { name: 'lison' }
for (let key in obj) {
  console.log(key);
}
// 'name'
Object.defineProperty(obj, "name", {
  enumerable: false
});
for (let key in obj) {
  console.log(key);
}
// 什么都没打印
Object.defineProperty(obj, "name", {
  writable: true
});
obj.name = "test";
console.log(obj);
// { name: 'test' }
Object.defineProperty(obj, "name", {
  configurable: false
});
Object.defineProperty(obj, "name", {
  writable: false
});
// error Cannot redefine property: name
```

通过这个例子，我们分别体验了这三个属性修饰符，还要一个字段是 value，用来设置属性的值。首先当我们设置 writable 为 false 时，通过给 obj.name 赋值是没法修改它起初定义的属性值的；普通的属性在 for in 等迭代器中是可以遍历到的，但是如果设置了 enumerable 为 false，即为不可枚举的，就遍历不到了；最后如果设置 configurable 为 false，那么就再也无法通过 Object.defineProperty 修改该属性的三个描述符的值了，所以这是个不可逆的设置。正是因为设置属性的属性描述符需要用 Object.defineProperty 方法，而这个方法又没法通过 ES3 的语言模拟，所以不支持 ES5 的浏览器是没法使用属性描述符的。

讲完属性描述符，就要注意方法装饰器对于属性描述符相关的一些操作了。如果代码输出目标小于 ES5，属性描述符会是 undefined。

来看例子：

```
function enumerable(bool: boolean) {
  return function(
    target: any,
    propertyName: string,
    descriptor: PropertyDescriptor
  ) {
    console.log(target); // { getAge: f, constructor: f }
    descriptor.enumerable = bool;
  };
}
class Info {
  constructor(public age: number) {}
  @enumerable(false)
  getAge() {
    return this.age;
  }
}
const info = new Info(18);
console.log(info);
// { age: 18 }
for (let propertyName in info) {
  console.log(propertyName);
}
// "age"
```

这个例子中通过我们定义了一个方法装饰器工厂，装饰器工厂返回一个装饰器；因为这个装饰器修饰在下面使用的时候修饰的是实例(或者实例继承的)的方法，所以装饰器的第一个参数是类的原型对象；第二个参数是这个方法名；第三个参数是这个属性的属性描述符的对象，可以直接通过设置这个对象上包含的属性描述符的值，来控制这个属性的行为。我们这里定义的这个方法装饰器，通过传入装饰器工厂的一个布尔值，来设置这个装饰器修饰的方法的可枚举性。如果去掉@enumerable(false)，那么最后 for in 循环打印的结果，会既有"age"又有"getAge"。

如果方法装饰器返回一个值，那么会用这个值作为方法的属性描述符对象：

```
function enumerable(bool: boolean): any {
  return function(
    target: any,
    propertyName: string,
    descriptor: PropertyDescriptor
  ) {
    return {
      value: function() {
        return "not age";
      },
      enumerable: bool
    };
  };
}
class Info {
  constructor(public age: number) {}
  @enumerable(false)
  getAge() {
    return this.age;
  }
}
const info = new Info();
console.log(info.getAge()); // "not age"
```

我们在这个例子中，在方法装饰器中返回一个对象，对象中包含 value 用来修改方法，enumerable 用来设置可枚举性。我们可以看到最后打印出的 info.getAge()的结果为"not age"，说明我们成功使用`function () { return “not age” }`替换了被装饰的方法`getAge () { return this.age }`

注意，当构建目标小于 ES5 的时候，方法装饰器的返回值会被忽略。

3.12.4. 访问器装饰器

访问器也就是我们之前讲过的 set 和 get 方法，一个在设置属性值的时候触发，一个在获取属性值的时候触发。

首先要注意一点的是，TS 不允许同时装饰一个成员的 get 和 set 访问器，只需要这个成员 get/set 访问器中定义在前面的一个即可。

访问器装饰器也有三个参数，和方法装饰器是一模一样的，这里就不再重复列了。来看例子：

```
function enumerable(bool: boolean) {
  return function(
    target: any,
    propertyName: string,
    descriptor: PropertyDescriptor
  ) {
    descriptor.enumerable = bool;
  };
}
class Info {
  private _name: string;
  constructor(name: string) {
    this._name = name;
  }
  @enumerable(false)
  get name() {
    return this._name;
  }
  @enumerable(false) // error 不能向多个同名的 get/set 访问器应用修饰器
  set name(name) {
    this._name = name;
  }
}
```

这里我们同时给 name 属性的 set 和 get 访问器使用了装饰器，所以在给定义在后面的 set 访问器使用装饰器时就会报错。经过 enumerable 访问器装饰器的处理后，name 属性变为了不可枚举属性。同样的，如果访问器装饰器有返回值，这个值会被作为属性的属性描述符。

3.12.5. 属性装饰器

属性装饰器声明在属性声明之前，它有 2 个参数，和方法装饰器的前两个参数是一模一样的。属性装饰器没法操作属性的属性描述符，它只能用来判断某各类中是否声明了某个名字的属性。

```
function printPropertyName(target: any, propertyName: string) {
  console.log(propertyName);
}
class Info {
  @printPropertyName
  name: string;
  @printPropertyName
  age: number;
}
```

3.12.6. 参数装饰器

参数装饰器有 3 个参数，前两个和方法装饰器的前两个参数一模一样：

- 装饰静态成员时是类的构造函数，装饰实例成员时是类的原型对象；
- 成员的名字；
- 参数在函数参数列表中的索引。

参数装饰器的返回值会被忽略，来看下面的例子：

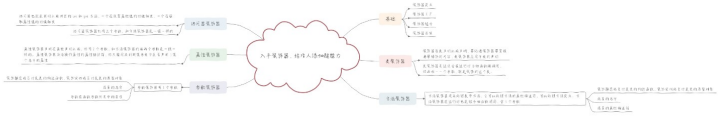
```
function required(target: any, propertName: string, index: number) {
  console.log(`修饰的是${propertName}的第${index + 1}个参数`);
}
class Info {
  name: string = "lison";
  age: number = 18;
  getInfo(prefix: string, @required infoType: string): any {
    return prefix + " " + this[infoType];
  }
}
interface Info {
  [key: string]: string | number | Function;
}
const info = new Info();
info.getInfo("hihi", "age"); // 修饰的是getInfo的第2个参数
```

这里我们在 getInfo 方法的第二个参数之前使用参数装饰器，从而可以在装饰器中获取到一些信息。

本节小结

本小节我们全面学习了装饰器的相关内容，虽然装饰器在ECAMAScript标准的议程中还没有最终确定，但是TypeScript的装饰器却已经被很多人接收，很多库和插件使用装饰器来处理一些值。本小节我们学习了装饰器的定义，以及使用装饰器工厂函数来实现可传参使用装饰器，这里我们着重强调了当一个地方添加了多个装饰器工厂函数和装饰器时的执行顺序，装饰器工厂函数是从上到下执行，装饰器是从下到上执行。我们还分别学习了：类装饰器、方法装饰器、访问器装饰器、属性装饰器和参数装饰器，它们分别处理对应的值。

本章的内容到这里就结束了，这一章的内容都是高阶语法，有一些难度，所以你需要多看多思考，自己动手把提到的例子都实践下，才能更好理解。下一章我们将对前面所学只是进行整合，将一些综合性知识。



24 条件类型，它不是三元操作符的写法吗？

使用模块封装代码

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论