装饰器（旧语法）

上一章介绍了装饰器的标准语法，那是在 2022 年通过成为标准的。但是在此之前，TypeScript 早在 2014 年就支持装饰器，不过使用的是旧语法。

装饰器的旧语法与标准语法，有相当大的差异。旧语法以后会被淘汰，但是目前大量现有项目依然在使用它，本章就介绍旧语法下的装饰器。

experimentalDecorators 编译选项

使用装饰器的旧语法，需要打开 `--experimentalDecorators` 编译选项。

bash

```
$ tsc --target ES5 --experimentalDecorators
```

此外，还有另外一个编译选项 `--emitDecoratorMetadata` ，用来产生一些装饰器的元数据，供其他工具或某些模块（比如 reflect-metadata ）使用。

这两个编译选项可以在命令行设置，也可以在 `tsconfig.json` 文件里面进行设置。

javascript

```
{
  "compilerOptions": {
    "target": "ES6",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

装饰器的种类

按照所装饰的不同对象，装饰器可以分成五类。

- 类装饰器（Class Decorators）：用于类。
- 属性装饰器（Property Decorators）：用于属性。
- 方法装饰器（Method Decorators）：用于方法。
- 存取器装饰器（Accessor Decorators）：用于类的 set 或 get 方法。
- 参数装饰器（Parameter Decorators）：用于方法的参数。

下面是这五种装饰器一起使用的一个示例。

typescript

```
@ClassDecorator() //  （A）
class A {
  @PropertyDecorator() //  （B）
  name: string;

  @MethodDecorator() //（C）
  fly(
    @ParameterDecorator() //  （D）
    meters: number
  ) {
    // code
  }

  @AccessorDecorator() //  （E）
  get egg() {
    // code
  }
  set egg(e) {
    // code
  }
}
```

上面示例中，A 是类装饰器，B 是属性装饰器，C 是方法装饰器，D 是参数装饰器，E 是存取器装饰器。

注意，构造方法没有方法装饰器，只有参数装饰器。类装饰器其实就是在装饰构造方法。

另外，装饰器只能用于类，要么应用于类的整体，要么应用于类的内部成员，不能用于独立的函数。

typescript

```
function Decorator() {
  console.log('In Decorator');
}

@Decorator // 报错
function decorated() {
  console.log('in decorated');
}
```

上面示例中，装饰器用于一个普通函数，这是无效的，结果报错。

类装饰器

类装饰器应用于类（class），但实际上是应用于类的构造方法。

类装饰器有唯一参数，就是构造方法，可以在装饰器内部，对构造方法进行各种改造。如果类装饰器有返回值，就会替换掉原来的构造方法。

类装饰器的类型定义如下。

typescript

```
type ClassDecorator = <TFunction extends Function>(
  target: TFunction
) => TFunction | void;
```

上面定义中，类型参数 `TFunction` 必须是函数，实际上就是构造方法。类装饰器的返回值，要么是返回处理后的原始构造方法，要么返回一个新的构造方法。

下面就是一个示例。

typescript

```
function f(target: any) {
  console.log("apply decorator");
  return target;
}

@f
class A {}
// 输出：apply decorator
```

上面示例中，使用了装饰器 `@f` ，因此类 `A` 的构造方法会自动传入 `f` 。

类 `A` 不需要新建实例，装饰器也会执行。装饰器会在代码加载阶段执行，而不是在运行时执行，而且只会执行一次。

由于 TypeScript 存在编译阶段，所以装饰器对类的行为的改变，实际上发生在编译阶段。这意味着，TypeScript 装饰器能在编译阶段运行代码，也就是说，它本质就是编译时执行的函数。

下面再看一个示例。

typescript

```
@sealed
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }
}

function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}
```

上面示例中，装饰器 `@sealed()` 会锁定 `BugReport` 这个类，使得它无法新增或删除静态成员和实例成员。

如果除了构造方法，类装饰器还需要其他参数，可以采取“工厂模式”，即把装饰器写在一个函数里面，该函数可以接受其他参数，执行后返回装饰器。但是，这样就需要调用装饰器的时候，先执行一次工厂函数。

typescript

```
function factory(info: string) {
  console.log("received: ", info);
  return function (target: any) {
    console.log("apply decorator");
    return target;
  };
}
```

```
@factory("log something")
class A {}
```

上面示例中，函数 `factory()` 的返回值才是装饰器，所以加载装饰器的时候，要先执行一次 `@factory('log something')` ，才能得到装饰器。这样做的好处是，可以加入额外的参数，本例是参数 `info` 。

总之， `@` 后面要么是一个函数名，要么是函数表达式，甚至可以写出下面这样的代码。

typescript

```
@((constructor: Function) => {
  console.log("log something");
})
class InlineDecoratorExample {
  // ...
}
```

上面示例中， `@` 后面是一个箭头函数，这也是合法的。

类装饰器可以没有返回值，如果有返回值，就会替代所装饰的类的构造函数。由于 JavaScript 的类等同于构造函数的语法糖，所以装饰器通常返回一个新的类，对原有的类进行修改或扩展。

typescript

```
function decorator(target: any) {
  return class extends target {
    value = 123;
  };
}

@decorator
class Foo {
  value = 456;
}

const foo = new Foo();
console.log(foo.value); // 123
```

上面示例中，装饰器 `decorator` 返回一个新的类，替代了原来的类。

上例的装饰器参数 `target` 类型是 `any` ，可以改成构造方法，这样就更准确了。

typescript

```typescript
type Constructor = {
  new (...args: any[]): {};
};

function decorator<T extends Constructor>(target: T) {
  return class extends target {
    value = 123;
  };
}
```

这时，装饰器的行为就是下面这样。

javascript

```javascript
@decorator
class A {}

// 等同于
class A {}
A = decorator(A) || A;
```

上面代码中，装饰器要么返回一个新的类 `A` ，要么不返回任何值， `A` 保持装饰器处理后的状态。

方法装饰器

方法装饰器用来装饰类的方法，它的类型定义如下。

typescript

```typescript
type MethodDecorator = <T>(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<T>
) => TypedPropertyDescriptor<T> | void;
```

方法装饰器一共可以接受三个参数。

- target：（对于类的静态方法）类的构造函数，或者（对于类的实例方法）类的原型。
- propertyKey：所装饰方法的方法名，类型为 `string|symbol` 。
- descriptor：所装饰方法的描述对象。

方法装饰器的返回值（如果有的话），就是修改后的该方法的描述对象，可以覆盖原始方法的描述对象。

下面是一个示例。

typescript

```typescript
function enumerable(value: boolean) {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    descriptor.enumerable = value;
  };
}

class Greeter {
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }

  @enumerable(false)
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

上面示例中，方法装饰器 `@enumerable()` 装饰 Greeter 类的 `greet()` 方法，作用是修改该方法的描述对象的可遍历性属性 `enumerable` 。 `@enumerable(false)` 表示将该方法修改成不可遍历。

下面再看一个例子。

typescript

```typescript
function logger(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const original = descriptor.value;
```

```
    descriptor.value = function (...args) {
      console.log("params: ", ...args);
      const result = original.call(this, ...args);
      console.log("result: ", result);
      return result;
    };
  }

  class C {
    @logger
    add(x: number, y: number) {
      return x + y;
    }
  }

  new C().add(1, 2);
  // params:  1 2
  // result:  3
```

上面示例中，方法装饰器 `@logger` 用来装饰 `add()` 方法，它的作用是让该方法输出日志。每当 `add()` 调用一次，控制台就会打印出参数和运行结果。

属性装饰器

属性装饰器用来装饰属性，类型定义如下。

```typescript
type PropertyDecorator = (target: Object, propertyKey: string | symbol) => void;
```

属性装饰器函数接受两个参数。

- target：（对于实例属性）类的原型对象（prototype），或者（对于静态属性）类的构造函数。
- propertyKey：所装饰属性的属性名，注意类型有可能是字符串，也有可能是 Symbol 值。

属性装饰器不需要返回值，如果有的话，也会被忽略。

下面是一个示例。

typescript

```typescript
function ValidRange(min: number, max: number) {
  return (target: Object, key: string) => {
    Object.defineProperty(target, key, {
      set: function (v: number) {
        if (v < min || v > max) {
          throw new Error(`Not allowed value ${v}`);
        }
      },
    });
  };
}

// 输出 Installing ValidRange on year
class Student {
  @ValidRange(1920, 2020)
  year!: number;
}

const stud = new Student();

// 报错 Not allowed value 2022
stud.year = 2022;
```

上面示例中，装饰器 `ValidRange` 对属性 `year` 设立了一个上下限检查器，只要该属性赋值时，超过了上下限，就会报错。

注意，属性装饰器的第一个参数，对于实例属性是类的原型对象，而不是实例对象（即不是 `this` 对象）。这是因为装饰器执行时，类还没有新建实例，所以实例对象不存在。

由于拿不到 `this` ，所以属性装饰器无法获得实例属性的值。这也是它没有在参数里面提供属性描述对象的原因。

typescript

```typescript
function logProperty(target: Object, member: string) {
  const prop = Object.getOwnPropertyDescriptor(target, member);
  console.log(`Property ${member} ${prop}`);
}

class PropertyExample {
  @logProperty
  name: string = "Foo";
```

```
}
// 输出 Property name undefined
```

上面示例中，属性装饰器 `@logProperty` 内部想要获取实例属性 `name` 的属性描述对象，结果拿到的是 `undefined` 。因为上例的 `target` 是类的原型对象，不是实例对象，所以拿不到 `name` 属性，也就是说 `target.name` 是不存在的，所以拿到的是 `undefined` 。只有通过 `this.name` 才能拿到 `name` 属性，但是这时 `this` 还不存在。

属性装饰器不仅无法获得实例属性的值，也不能初始化或修改实例属性，而且它的返回值也会被忽略。因此，它的作用很有限。

不过，如果属性装饰器设置了当前属性的存取器（getter/setter），然后在构造函数里面就可以对实例属性进行读写。

typescript

```typescript
function Min(limit: number) {
  return function (target: Object, propertyKey: string) {
    let value: string;

    const getter = function () {
      return value;
    };

    const setter = function (newVal: string) {
      if (newVal.length < limit) {
        throw new Error(`Your password should be bigger than ${limit}`);
      } else {
        value = newVal;
      }
    };
    Object.defineProperty(target, propertyKey, {
      get: getter,
      set: setter,
    });
  };
}

class User {
  username: string;

  @Min(8)
  password: string;
```

```
  constructor(username: string, password: string) {
    this.username = username;
    this.password = password;
  }
}

const u = new User("Foo", "pass");
// 报错 Your password should be bigger than 8
```

上面示例中，属性装饰器 `@Min` 通过设置存取器，拿到了实例属性的值。

存取器装饰器

存取器装饰器用来装饰类的存取器（accessor）。所谓“存取器”指的是某个属性的取值器（getter）和存值器（setter）。

存取器装饰器的类型定义，与方法装饰器一致。

typescript

```
type AccessorDecorator = <T>(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<T>
) => TypedPropertyDescriptor<T> | void;
```

存取器装饰器有三个参数。

- target：（对于静态属性的存取器）类的构造函数，或者（对于实例属性的存取器）类的原型。
- propertyKey：存取器的属性名。
- descriptor：存取器的属性描述对象。

存取器装饰器的返回值（如果有的话），会作为该属性新的描述对象。

下面是一个示例。

typescript

```
function configurable(value: boolean) {
  return function (
    target: any,
```

```
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    descriptor.configurable = value;
  };
}

class Point {
  private _x: number;
  private _y: number;
  constructor(x: number, y: number) {
    this._x = x;
    this._y = y;
  }

  @configurable(false)
  get x() {
    return this._x;
  }

  @configurable(false)
  get y() {
    return this._y;
  }
}
```

上面示例中，装饰器 `@configurable(false)` 关闭了所装饰属性（ `x` 和 `y` ）的属性描述对象的 `configurable` 键（即关闭了属性的可配置性）。

下面的示例是将装饰器用来验证属性值，如果赋值不满足条件就报错。

typescript

```
function validator(
  target: Object,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const originalGet = descriptor.get;
  const originalSet = descriptor.set;

  if (originalSet) {
    descriptor.set = function (val) {
      if (val > 100) {
```

```
        throw new Error(`Invalid value for ${propertyKey}`);
      }
      originalSet.call(this, val);
    };
  }
}

class C {
  #foo!: number;

  @validator
  set foo(v) {
    this.#foo = v;
  }

  get foo() {
    return this.#foo;
  }
}

const c = new C();
c.foo = 150;
// 报错
```

上面示例中，装饰器用自己定义的存值器，取代了原来的存值器，加入了验证条件。

TypeScript 不允许对同一个属性的存取器（getter 和 setter）使用同一个装饰器，也就是说只能装饰两个存取器里面的一个，且必须是排在前面的那一个，否则报错。

typescript

```
// 报错
class Person {
  #name: string;

  @Decorator
  set name(n: string) {
    this.#name = n;
  }

  @Decorator // 报错
  get name() {
    return this.#name;
```

```
  }
}
```

上面示例中， `@Decorator` 同时装饰 `name` 属性的存值器和取值器，所以报错。

但是，下面的写法不会报错。

typescript

```typescript
class Person {
  #name: string;

  @Decorator
  set name(n: string) {
    this.#name = n;
  }
  get name() {
    return this.#name;
  }
}
```

上面示例中， `@Decorator` 只装饰它后面第一个出现的存值器（ `set name()` ），并不装饰取值器（ `get name()` ），所以不报错。

装饰器之所以不能同时用于同一个属性的存值器和取值器，原因是装饰器可以从属性描述对象上面，同时拿到取值器和存值器，因此只调用一次就够了。

参数装饰器

参数装饰器用来装饰构造方法或者其他方法的参数。它的类型定义如下。

typescript

```typescript
type ParameterDecorator = (
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number
) => void;
```

参数装饰器接受三个参数。

- target：（对于静态方法）类的构造函数，或者（对于类的实例方法）类的原型对象。

- propertyKey：所装饰的方法的名字，类型为 `string|symbol` 。
- parameterIndex：当前参数在方法的参数序列的位置（从 0 开始）。

该装饰器不需要返回值，如果有的话会被忽略。

下面是一个示例。

typescript

```
function log(
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number
) {
  console.log(`${String(propertyKey)} NO.${parameterIndex} Parameter`);
}

class C {
  member(@log x: number, @log y: number) {
    console.log(`member Paremeters: ${x} ${y}`);
  }
}

const c = new C();
c.member(5, 5);
// member NO.1 Parameter
// member NO.0 Parameter
// member Paremeters: 5 5
```

上面示例中，参数装饰器会输出参数的位置序号。注意，后面的参数会先输出。

跟其他装饰器不同，参数装饰器主要用于输出信息，没有办法修改类的行为。

装饰器的执行顺序

前面说过，装饰器只会执行一次，就是在代码解析时执行，哪怕根本没有调用类新建实例，也会执行，而且从此就不再执行了。

执行装饰器时，按照如下顺序执行。

1. 实例相关的装饰器。
2. 静态相关的装饰器。

3. 构造方法的参数装饰器。
4. 类装饰器。

请看下面的示例。

typescript

```typescript
function f(key: string): any {
  return function () {
    console.log("执行：", key);
  };
}

@f("类装饰器")
class C {
  @f("静态方法")
  static method() {}

  @f("实例方法")
  method() {}

  constructor(@f("构造方法参数") foo: any) {}
}
```

加载上面的示例，输出如下。

typescript

```typescript
执行：  实例方法
执行：  静态方法
执行：  构造方法参数
执行：  类装饰器
```

同一级装饰器的执行顺序，是按照它们的代码顺序。但是，参数装饰器的执行总是早于方法装饰器。

typescript

```typescript
function f(key: string): any {
  return function () {
    console.log("执行：", key);
  };
}

class C {
  @f("方法1")
```

```
  m1(@f("参数1") foo: any) {}

  @f("属性1")
  p1: number;

  @f("方法2")
  m2(@f("参数2") foo: any) {}

  @f("属性2")
  p2: number;
}
```

加载上面的示例，输出如下。

typescript

```
执行：  参数1
执行：  方法1
执行：  属性1
执行：  参数2
执行：  方法2
执行：  属性2
```

上面示例中，实例装饰器的执行顺序，完全是按照代码顺序的。但是，同一个方法的参数装饰器，总是早于该方法的方法装饰器执行。

如果同一个方法或属性有多个装饰器，那么装饰器将顺序加载、逆序执行。

typescript

```
function f(key: string): any {
  console.log("加载：", key);
  return function () {
    console.log("执行：", key);
  };
}

class C {
  @f("A")
  @f("B")
  @f("C")
  m1() {}
}
// 加载：  A
// 加载：  B
```

```
// 加载：  C
// 执行：  C
// 执行：  B
// 执行：  A
```

如果同一个方法有多个参数，那么参数也是顺序加载、逆序执行。

typescript

```typescript
function f(key: string): any {
  console.log("加载：", key);
  return function () {
    console.log("执行：", key);
  };
}

class C {
  method(@f("A") a: any, @f("B") b: any, @f("C") c: any) {}
}
// 加载：  A
// 加载：  B
// 加载：  C
// 执行：  C
// 执行：  B
// 执行：  A
```

为什么装饰器不能用于函数?

装饰器只能用于类和类的方法，不能用于函数，主要原因是存在函数提升。

JavaScript 的函数不管在代码的什么位置，都会提升到代码顶部。

typescript

```typescript
addOne(1);
function addOne(n: number) {
  return n + 1;
}
```

上面示例中，函数 `addOne()` 不会因为在定义之前执行而报错，原因就是函数存在提升，会自动提升到代码顶部。

如果允许装饰器可以用于普通函数，那么就有可能导致意想不到的情况。

typescript

```
let counter = 0;

let add = function (target:any) {
  counter++;
};

@add
function foo() {
  //...
}
```

上面示例中，本来的意图是装饰器 `@add` 每使用一次，变量 `counter` 就加 `1` ，但是实际上会报错，因为函数提升的存在，使得实际执行的代码是下面这样。

javascript

```
@add // 报错
function foo() {
  //...
}

let counter = 0;
let add = function (target:any) {
  counter++;
};
```

上面示例中， `@add` 还没有定义就调用了，从而报错。

总之，由于存在函数提升，使得装饰器不能用于函数。类是不会提升的，所以就没有这方面的问题。

另一方面，如果一定要装饰函数，可以采用高阶函数的形式直接执行，没必要写成装饰器。

javascript

```
function doSomething(name) {
  console.log("Hello, " + name);
}

function loggingDecorator(wrapped) {
  return function () {
    console.log("Starting");
```

```
    const result = wrapped.apply(this, arguments);
    console.log("Finished");
    return result;
  };
}

const wrapped = loggingDecorator(doSomething);
```

上面示例中， `loggingDecorator()` 是一个装饰器，只要把原始函数传入它执行，就能起到装饰器的效果。

多个装饰器的合成

多个装饰器可以应用于同一个目标对象，可以写在一行。

typescript

```
@f @g x
```

上面示例中，装饰器 `@f` 和 `@g` 同时装饰目标对象 `x` 。

多个装饰器也可以写成多行。

typescript

```
@f
@g
x
```

多个装饰器的效果，类似于函数的合成，按照从里到外的顺序执行。对于上例来说，就是执行 `f(g(x))` 。

前面也说过，如果 `f` 和 `g` 是表达式，那么需要先从外到里求值。

参考链接

- A Complete Guide to TypeScript Decorators, by Saul Mirone
- Deep introduction to using and implementing TypeScript decorators, by David Herron
- Deep introduction to property decorators in TypeScript, by David Herron

- Deep introduction to accessor decorators in TypeScript, by David Herron
- Using Property Decorators in Typescript with a real example, by Dany Paredes

最后更新: 2023/8/13 15:25

Previous page
装饰器

Next page
declare 关键字