途虎养车

养车 就是途虎

装饰器——拓展功能的好方法

商务应用前端组 吴双承

内容大纲

- 装饰器模式
- 装饰器是什么？

- 类装饰器
- 类方法装饰器
- 属性装饰器
- 参数装饰器

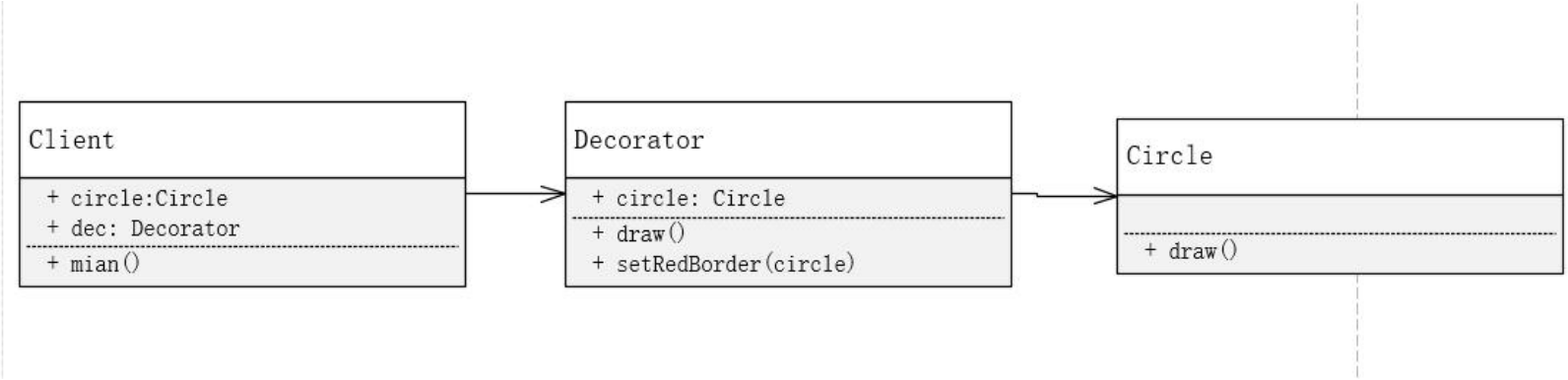
- 装饰器的应用

装饰器初识

装饰器模式

装饰器模式（Decorator Pattern）是一种用于替代继承的技术（不是完全替代），它通过一种无须定义子类的方式来给对象动态增加职责，使用对象之间的关联关系取代类之间的继承关系。

特点：给现有的对象添加新的功能，同时又不改变其结构。



装饰器模式

```
// 圆类
class Circle {
  draw() {
    console.log('画圆形');
  }
}
// 装饰器
class Decorator {
  constructor(circle) {
    this.circle = circle
  }
  draw() {
    // 原有的功能
    this.circle.draw()
    // 加装饰
    this.setRedBorder(circle)
  }
  setRedBorder(circle) {
    console.log('设置了红色边框')
  }
}
```

装饰器是什么？

简单来说，装饰器是一个对类（包括内部的属性和方法）进行处理的函数。
使用时，形如@+函数名，如下test装饰器：

```
@test
class Person {}
// 等同于
class Person{};
Person = test(Person) || Person; // 此时的装饰器就是对类处理的一个函数

function test(target) { // 此处的target为类Person本身
  target.isTestable = true; // 增加静态属性
}
```

装饰器写法

类装饰器

参数：target。指代要修饰的类。

```
@dec
class Demo {}
function dec(target) { // target为要修饰的类Demo
  // todo
}
```

如果想传入多个参数，可以使用装饰器工厂：

```
// 通过装饰器工厂处理更多参数
function dec(arg1, arg2) {
  return function(target) { // target为要修饰的类
    console.log(arg1, arg2)
    // todo...
  }
}
```

类装饰器

```
// mixins
function mixins(...list) {
  return function (target) { // 这个 target 在这里就是 MyClass 这个类
    Object.assign(target.prototype, ...list)
  }
}
// 定义一个方法集
const Foo = {
  foo() {
    console.log('foo')
  }
}
// 使用装饰器
@mixins(Foo)
class MyClass { }

let obj = new MyClass();
obj.foo() // 'foo'
```

类方法装饰器

对类方法的装饰本质是操作其描述符。可以把此时的装饰器理解成是Object.defineProperty(obj, prop, descriptor)的语法糖。

方法装饰器表达式会在运行时当作函数被调用，传入下列3个参数：

1. Tartget:对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. Name:成员的名字。
3. Descriptor:成员的*属性描述符*。

```
class Person {
  @readonly
  method() {  }
}

function readonly(target, name, descriptor){ // target为Person.prototype
  // todo...
  return descriptor; // 最后返回属性描述符
}
```

类方法装饰器

注意：

此时操作的对象是*属性描述符*。它会被应用到方法的属性描述符上，可以用来监听，修改或者替换方法定义。

```
class Person {
  constructor(first, last) {
    this.first = first;
    this.last = last;
  }
  @readonly
  method() { return `${this.first} ${this.last}` }
}
function readonly(target, name, descriptor){ // 此处 target 为 Person.prototype; name 为 method;
  /* descriptor对象原来的值如下
    {
      value: f,
      enumerable: false,
      configurable: true,
      writable: true
    };
  */
  descriptor.writable = false; // 可读写描述置为false
  return descriptor; // 此时返回的是属性描述符
}
readonly(Person.prototype, 'name', descriptor);
// 类似于
Object.defineProperty(Person.prototype, 'name', descriptor);
```

属性装饰器

与方法装饰器一样，该装饰器接收三个参数：

1. Target: 类的原型对象
2. Name: 属性名
3. Descriptor：属性描述符

属性装饰器

```
function nonenumerable(target, key, descriptor) {
  // 设置不可遍历属性
  descriptor.enumerable = false;
  return descriptor;
}

class newNum {
  num1 = 101;
  @nonenumerable
  num2 = 102;
}

let nums = new newNum();
console.log('nums: ', nums); // newNum {num1: 101, num2: 102}
console.log(Object.keys(nums)); // ["num1"]
```

参数装饰器

参数装饰器接收三个参数：

1. Target: 类
2. MethodName: 所在方法的名字
3. ParamsIndex：参数所在的索引

```
function addAge(target: any, methodName: string, paramsIndex: number) {
  target.age = 10;
}
class Person {
  login(username: string, @addAge password: string) { // 对方法login中的password进行修饰
  }
  aaa(@addAge a: string) { // 对方法aaa中的a参数进行修饰
  }
}
```

参数装饰器

```
// 声明合并，同时拥有两个声明的属性
interface nPerson {
  age: number;
}
function addAge(target: any, methodName: string, paramsIndex: number) {
  target.age = 10;
}
class nPerson {
  login(username: string, @addAge password: string) {
    console.log('login', this.age, username, password);
  }
  aaa(@addAge a: string) {
    console.log(a, this.age);
  }
}
let p = new nPerson();
p.login('yehuozhili', '123456'); // login 10 yehuozhili 123456
p.aaa('44'); // 44 10
```

装饰器的应用

装饰器的应用

- 让人更加关注业务代码的开发，封装功能辅助性的代码。

- 封装写日志的代码
- 处理异常的代码
- 节流
- 防抖
- 缓存、权限校验

- 第三方的装饰器库：

  core-decorators.js：https://github.com/jayphelps/core-decorators

  lodash-decorators：https://www.npmjs.com/package/lodash-decorators

途虎养车

养 车　就 是 途 虎