第十一课 装饰器模式

学习目录

- **使用场景**
- **使用方式**
- **优缺点**

一. 使用场景

装饰器模式其实是一种 aop 编程的具体体现，所谓 aop 正好与我们之前说的 oop 不同。正式说装饰器模式之前让我们先来了解一下 oop 以及 aop 的含义。

oop

面向对象编程，在 js 中的体现是通过构造函数与原型继承的概念来做到的面向对象编程，用这种编程方式开发的类是一种纵向的对象层次结构。一个类或者对象的扩展基本上是以这个类或者对象的属性或者方法的扩展为主，对他们的属性和方法进行抽象封装形成统一的清晰的逻辑单元，这是面向对象变成 oop 的含义。

aop

面向切面编程，在 js 中的体现是装饰器模式，或者说简单的函数与对象扩展都有面向切面编程 aop 的思想在里面。aop 是一种横向的对象层次结构，它重点不在于对类或者对象的封装，而是体现在运行期间动态的为对象或者类增加额外的功能。让多个对象或者类同时拥有同样额外功能。

简单来说 oop 是为了划分出清晰的功能单元模块，aop 是为了扩展业务逻辑种具体的某些阶段或者额外功能。

aop 与 oop 是一种相互补充和相互促进的关系。

我们要说的装饰器模式就是 aop 思想的最好体现，其实在不同语言中都有装饰器模式的实现，有些语言还有装饰器语法支持，比如 python 中的函数装饰器，对于收集日志和数据信息就非常好用，在 js 中其实也有直接的装饰器语法，但是在 js 引擎中的支持度非常低，几乎没法用 js 提供的装饰器语法来编写装饰器代码结构。

因此我们需要用 js 现有语法来实现装饰器模式，装饰器模式的主要应用场景比较广泛，在任何想动态扩展对象或者函数的地方都可以使用装饰器模式，比如我们在这套课程一开始讲解到的获取函数执行时间的代码，那其实就是装饰器模式在 js 中的应用之一。

又比如我们在日常开发中，在项目快要上线的时候，我们需要在记录登录次数、获取错误日志、记录访问人数等地方使用后端 api 进行统计数据的埋点操作，这些埋点操作实际并没有多少交互，但是却是对应用数据收集的关键步骤，大多数情况下， 我们会进入到需要收集数据的对象或者函数中进行埋点操作，这样就等于改变了原来对象或者函数，同时增加了较多的数据收集重复代码，因此也需要用到装饰器模式来简化这种数据收集步骤与过程。

装饰器模式的核心思想是利用 aop 思想在不改变原有对象或者函数的情况下，动态的给对象或者函数增加额外的功能。

二．使用方式

装饰器模式在 js 中的使用和其他语言有一定的区别，我们可以很简单的使用装饰器模式，也可以比较复杂使用它。

简单的装饰器模式

```
function fn1(){
    console.log('星星课堂');
}
var _fn1 = fn1;
fn1 = function(){
    console.log('web 前端培训工作室');
    _fn1();
    console.log('xingxingclassroom');
}
fn1();
```

这样使用确实是既保证了原函数的调用又增加了额外的功能，但是这种用变量记录原函数的方式，随着装饰器越来越多也会跟着增加变得难以维护，同时还会有找不到 this 的问题。

```
function fn1(num){
    this.num = num;
    console.log(this.num);
    console.log('星星课堂');
}
var _fn1 = fn1;
fn1 = function(){
    console.log('web 前端培训工作室');
```

```
        _fn1.apply(this,arguments);
        console.log('xingxingclassroom');
    }
    fn1(100);
```

可以用函数的 apply 方法可以保证原函数中的 this 引用正确，不会找不到。不过这种方式还是不够通用，每次编写函数的额外功能都要进行记录，非常麻烦。因此我们需要写一套相对通用的方式来实现装饰器模式的方法。

函数原型装饰器模式

我们可以对函数的原型进行方法扩展来满足函数通用装饰器模式的效果，这和我们本套课程一开始的记录函数执行时间的装饰器模式是差不多的方法。

```
Function.prototype.beforeFn = function(fn){
    var _fn = this;
    return function(){
        fn.apply(this,arguments);
        return _fn.apply(this,arguments);
    }
}
Function.prototype.afterFn = function(fn){
    var _fn = this;
    return function(){
        var _fnResult = _fn.apply(this,arguments);
```

```
            fn.apply(this,arguments);
            return _fnResult;
        }
    }
    function fn1(){
        console.log('星星课堂 1');
    }
    fn1 = fn1.beforeFn(function(){
        console.log('fn1 调用之前');
    }).afterFn(function(){
        console.log('fn1 调用之后');
    });
    fn1();
```

以上的写法只是相对通用，因为有时候不同的业务逻辑在函数调用之前和之后有可能会需要增加额外的操作代码，这个可以根据具体要求来进一步扩展相关业务逻辑功能。

三．优缺点

1.装饰器模式可以根据 aop 思想动态的为对象或者函数增加额外功能，这种额外功能的增加是不需要修改原函数的。

2.装饰器模式在封装 ajax 参数、数据记录等方面比较好用，而且相对适用性较好，可以满足大多数函数的额外扩展需求。

3.装饰器模式可能会占用函数原型对象命名空间，从而污染函数原型对象，因此需要根

据需要统一编写与使用。

4.装饰器模式中使用了闭包结构保留了原函数的引用，因此可能会出现一定的性能问题，同时由于返回的是新函数，原函数中如果有属性也会被新函数忽略，因此在使用的时候要尽量避免这些问题。

谢谢观看！

我是星星课堂老师：周小周