15 装饰者模式

在程序开发中，我们其实一开始都不希望某个类天生非常庞大，包含许多职责。那么我们就可以使用装饰者模式来动态地给某个对象添加一些额外的职责，从而不会影响从这个类中派生的其他对象。

传统的面向对象给对象添加功能通常通过继承方式，不过这种不太灵活，一方面导致父子类存在耦合性，父类改变子类也改变，另一方面继承这种复用成为“白箱复用”，父类内部细节对子类是可见的，破坏了封装性

这里另外一种动态的给对象添加职责的方式成为装饰者decorator模式。不改变对象自身的基础上，在运行时添加职责和功能，更加轻便灵活。例如如果天冷了，就可以多穿外套。

模拟面向对象的装饰者模式

作为解释执行语言，JS中给对象动态添加职责其实很简单，虽然会改变对象自身，但是更符合JS语言特色：

```
let obj = {
    name: 'sven',
    address: '深圳市',
};

obj.address = obj.address + '福田区';
```

但是这种装饰者模式在JS中适用的场景并不多，例如如下的飞机类，通过装饰者模式升级Fire能力：

```
let Plane = function(){}

Plane.prototype.fire = function(){
    console.log('发射普通子弹')
}

let MissileDecorator = function(plane){
    this.plane = plane;
}

MissileDecorator.prototype.fire = function(){
    this.plane.fire();
```

```
    console.log('发射导弹')
}

let plane = new Plane();
plane = new MissileDecorator(plane);
plane.fire(); // 输出 发射普通子弹 发射导弹
```

这里导弹类接受plane对象并保存了这个参数，并给fire增加职责，通过链式引用形成一个聚合对象。

这里该plane对象的内部装饰过程对用户来说是透明，被装饰者也不需要感知是否被装饰过

装饰者也是包装器

GoF原想把装饰者decorator模式成为包装器wrapper模式，其实从结构上看，wrapper的说法更加贴切。装饰者将一个对象嵌入到另一个对象之中，时机上相当于这个对象被另一个对象包装起来，形成包装链。

装饰函数

在JS中，几乎一切都是对象，函数又是一等对象。JS中可以很方便地给某个对象扩展属性和方法但是很难在不改动某个函数的源码的情况下给此函数添加额外的功能。简单来说，运行时我们很难切入某个函数的执行环境

为函数添加功能最简单粗暴方法就是直接改写该函数，但这是最差的，违反开闭原则：

```
let a = function(){
    console.log(1)
}

// 直接修改
let a = function(){
    console.log(1)
    console.log(2)
}
```

但是很多情况我们不希望碰原函数，可能因为是其他同事编写的，实现杂乱，也可能是老项目，改不动，有没有可能不改动原函数来增加功能?

之前的示例代码演示了一种保存引用的方式：

```
/**
 * 保存引用来新增功能
```

```
 */
let a = function(){
    console.log(1)
}

let _a = a;

a = function(){
    _a();
    console.log(2);
}

a();

// 实际开发中修改全局函数也会这样 - 避免覆盖之前函数的行为
window.onload = function(){
    console.log(1)
}

let _onload = window.onload || function(){};

window.onload = function(){
    _onload();
    console.log(2);
}
```

代码是符合开闭原则的，新增功能不需要修改源代码，但是又两个问题：

- 需要额外维护_onload这个中间变量，如果链条长的话需要装饰的函数就会很多，中间变量数量也越来越多
- 存在this劫持问题，上面onload例子中没有这个烦恼，因为this指向全局window，跟调用window.load一样（函数作为对象的方法被调用，this指向该对象，所以this也是指向window），如果换成document.getElementById的装饰就会出问题，需要手动把document当作上下文传入_getElementById才行

用AOP装饰函数

这里我们用之前的AOP思想来提供一种方案给函数增加额外功能，这种方法比较完美，首先给出before和after方法的实现：

```
/**
 * AOP实现装饰器
 */
Function.prototype.before = function (fn) {
```

```
    // 保存原函数引用
    let _self = this;
    // 返回新的定制函数
    return function () {
        // 执行新的函数并保证this不被劫持，新函数接受的参数也会传入原函数
        fn.apply(this, arguments);
        // 执行原函数
        return _self.apply(this, arguments);
    }
}

Function.prototype.after = function (fn) {
    // 保存原函数引用
    let _self = this;
    // 返回新的定制函数
    return function () {
        // 执行原函数
        const result = _self.apply(this, arguments);
        // 执行新的函数并保证this不被劫持，新函数接受的参数也会传入原函数
        fn.apply(this, arguments);
        return result;
    }
}
```

保存的this用于获取原函数的引用，最后返回一个“代理”函数，工作就是将请求分别转发给新添加的函数与原函数并保证执行顺序 -- 最终实现了动态装饰的效果

下面来看看一个demo：

```
// 适用before来增加新的window.onload
window.onload = function(){
    alert(1);
}

window.onload = (window.onload || function(){}).after(function(){
    alert(2);
});
```

这个时候定制window.onload就很简单了。

当然这里还有一种不污染原型的写法，把新旧函数作为参数处理传入：

```
// 不污染原型
let before = function(fn, beforeFn){
```

```
    return function(){
        beforeFn.apply(this, arguments);
        return fn.apply(this, arguments);
    }
}
```

✍️ 现在也有很多TS借助装饰器@特性来实现AOP，也是一种比较好的实践方案

AOP的应用实例

无论是业务代码还是框架，用AOP装饰函数在实际中还是有用的，我们可以把行为职责拆分细粒度，然后通过装饰把他们组合，编写一个松耦合和高复用的系统。

数据统计上报

分离业务代码和数据统计代码无论在什么语言，都是AOP经典应用。

有时候在项目结尾阶段，我们可能不得不在早已封装好的函数中加上很多统计数据的代码，例如页面中有一个button，我们需要在onclick时候进行数据上报来统计适用button的用户数，一般情况下我们这样写：

```
/**
 * 数据上报示例
 */
const showLogin = function () {
    console.log('打开浮层');
    log(this.getAttribute('tag'));
}

const log = function (tag) {
    console.log('上报标签为' + tag);
    // 上报代码
}

document.getElementById('button').onclick = showLogin;
```

这里不太好的地方就是上报的函数耦合到了onclick中，我们可以尝试适用AOP来解耦：

```
// AOP优化
let showLogin2 = function(){
    // 不耦合上报
```

```
    console.log('打开浮层')
}

let log2 = function () {
    console.log('上报标签为' + this.getAttribute('tag'));
    // 上报代码
}

// 打开登陆浮层之后上报数据
showLogin2 = showLogin2.after(log);

document.getElementById('button').onclick = showLogin2;
```

用AOP动态改变函数的参数

现在有一个ajax函数，负责项目中所有的ajax异步请求：

```
let ajax = function(type,url,param){
    // 略
}
```

有一次我们遭受了CSRF攻击。导致我们需要为这个运行良好的ajax函数添加一个Token参数来防御攻击，比较一般的方法是我们直接给所有请求都加上这个参数：

```
let ajax = function(type,url,param){
    param = param || {};
    param.Token = getToken();
    // 略
}
```

但是这种方式使得ajax函数变得僵硬，一方面所有的请求都带上了Token，另一方面移植会很困难。

这里我们可以借助AOP来保证ajax函数本身的纯净性：

```
let ajax = function(type,url,param){
    param = param || {};
    // 略
}

ajax = ajax.before(function(type, url, param){
```

```
7     param.Token = getToken();
8 })
```

装饰者模式和代理模式

初看起来装饰者和之前的代理模式有点像，其实都是描述了为对象提供间接引用而不直接修改影响对象

最主要的区别其实在于设计的目的：

- 代理模式：直接访问本体不方便或者不满足时候，提供一个代理对象来访问，通常是本体定义了关键功能，代理提供或拒绝对它的访问。通常也只有一层代理->本体的引用
  - 代理一般功能不会超过本体，只是提供一些辅助功能
- 装饰者模式：为对象本体动态添加行为和功能，一开始并不能确定对象的全部功能，按照所需进行添加，最终形成一条装饰链

小结

JS中独特的装饰者模式体现在装饰函数上，实际开发中也非常常用，例如框架开发中，我们可能继续让函数提供稳定且方便移植的功能，那些个性化和定制功能可以在框架之外提供动态装饰的方法，避免框架变得臃肿