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：