# 20 开放-封闭原则

在面向对象程序中，开放封闭原则（OCP）是最重要的一条原则，往往好的设计都需要符合这一条

## 示例：扩展window.onLoad函数

来了一个需求需要在window.onLoad中打印页面中所有节点数量：

```
window.onLoad = function(){
    // ...
    console.log(document.getElementsByTagName('*').length);
}
```

接着在需求变化过程中，可能经常需要找到相关代码并改写他们，这个是正常的，想要扩展或者修改功能最直接的就是修改源代码，但是这也是一种危险的欣慰，经常刚刚改好一个Bug，又在不知不觉中引发了其他Bug。

尤其是上面例子中，如果onLoad原本就是一个500-1000行的函数了，遍布各种业务逻辑，那么我们改动很有可能引起副作用。

那有没有什么方法可以解决问题并满足需求呢?

其实之前在15章已经有了答案，我们可以通过动态装饰函数方式来忽略onLoad原先的实现但是也可以新增我们的逻辑：

```
Function.prototype.after = function(fn){
    var _self = this;
    return function(){
        var ret = _self.apply(this, arguemnts);
        fn.apply(this, arguments);
        return ret;
    }
}

window.onload = (window.onload || function(){}).after(function {
    //...我们的逻辑
})
```

# 开放和封闭

上面对比明显后一种效果更好，更稳定。这也是开放-封闭的思想：为程序新增或者修改功能时，可以增加代码但是不允许改动程序源代码

# 用对象的多态性消除条件分支

我们经常会遇到过多的if-else语句，也是造成违反该原则的常见原因，我们不得不修改他们。

这里我们可以尝试使用对象多态性来让程序遵守这一原则

举个例子，原先的makeSound函数：

```
const makeSound = function(animal){
    if(animal instanceof Duck){
        console.log('嘎嘎嘎')
    }else if(animal instanceof Chicken){
        console.log('咯咯咯')
    }
}

// 新增动物都需要该上面的makeSound函数
```

这里我们利用多态的思想，把程序中变化的部分封装起来（动物叫声不一样），不变的部分隔离出来（动物都会叫），这样程序就有了扩展性：

```
const makeSound = function(animal){
    animal.sound();
}

class Duck {
    sound(){
        console.log('嘎嘎嘎')
    }
}

class Chicken{
    // ...
}

// ...
```

这样就不需要修改makeSound函数，同时也方便新增动物和修改动物叫声

# 找出变化的地方

这个原则实际应用可能有一些虚幻，并没有实际的教学模板叫我们怎么发现，但是我们可以尝试找出程序中将要发生变化的地方，然后把变化封装起来。

通过封装变化，可以隔离系统中稳定的和不稳定的部分，可以在后续演变过程种，将容易变化的部分进行替换和修改，当然具体哪些容易变化就跟具体的业务系统有关系了，需要对业务比较了解

除了利用对象多态性之外，还有一些方法可以帮助我们编写遵守原则的代码：

1. 放置挂钩hook

我们可以在程序可能变化的地方放一个hook，通过hook的返回结果来决定程序下一步，在代码路径上预埋分叉路口，方便为未来扩展，让程序有了变化的可能。例如webpack的plugin就是如此，以及之前的模板方法模式中

2. 使用回调函数

在JS中函数也可以传递那么我们的策略模式和命令模式都可以通过回调函数轻松实现，这其实也是一种特殊的挂钩，我们可以把变化逻辑封装在回调函数中，然后程序就可以根据回调函数内部逻辑不同，有不同的结果

很多工具函数和库都支持通过callback来进行用户扩展