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

设计模式中的开放-封闭原则

几乎所有设计模式，好的设计都经得起该原则的考验，无论是具体的设计模式还是抽象面向对象设计原则，都是为了让程序遵守这个，这里我们距离探讨

1. 发布-订阅模式

当出现新的订阅者时候，发布者代码不需要修改，不需要显式调用对象的某个接口

2. 模板方法模式

是一种典型的通过封装变化来提高系统扩展性的设计模式。这种程序中子类的方法种类和执行顺序都是不变的，所以可以把这部分逻辑抽象到父类的模板方法。而具体实现是变化的，封装到子类中就好

3. 策略模式

和上面模式是一对竞争者，很多情况可以相互替换，知识思想不同，算法的独立封装也避免了相互影响

4. 代理模式

通过代理对象将额外功能和本职工作解耦，减少本职对象的改动即可拥有额外能力

5. 职责链模式

    同样新增一条链路时候不会影响独立的环节

相对性

注意这里其实让程序保证完全封闭是很难做到的，就算做得到也会花费很多时间和经理，因为会引入更多抽象层次，增加整体复杂度，降低可读性。也会存在一些无法封闭或者无法预知的变化。

这里我们可以注重两点：

- 挑选出最容易发生变化的地方，然后针对其构造抽象来封闭
- 不可避免发生修改时候，尽量修改相对容易且对外暴露的地方。例如开源库的配置文件而非源代码

接受第一次愚弄

> 为了放置软件背着不必要的复杂度，我们可以允许自己被愚弄一次

让程序一开始就遵循原则很困难。一方面我们只能品经验预测一些可能变化的地方，另一方面需求排期有限。

所以有时候我们可以尝试接受第一次愚弄，假设没有变化，完成需求为主，再迭代过程中对工作造成影响时候可以在回过头来封装起来，确保不会掉进同一个坑里面