第 20 章　开放-封闭原则

在面向对象的程序设计中，开放-封闭原则（OCP）是最重要的一条原则。很多时候，一个程序具有良好的设计，往往说明它是符合开放-封闭原则的。

开放-封闭原则最早由Eiffel语言的设计者Bertrand Meyer在其著作***Object-Oriented Software Construction*** 中提出。它的定义如下：

> 软件实体（类、模块、函数）等应该是可以扩展的，但是不可修改。

本节我们不采用顺述的方式。在明白开放-封闭原则的定义之前，先看一个示例，这个示例曾经出现在第15章中，我们需要再往`window.onload` 函数中添加一些新的功能。

20.1 扩展window.onload 函数

假设我们是一个大型Web项目的维护人员，在接手这个项目时，发现它已经拥有10万行以上的JavaScript代码和数百个JS文件。

不久后接到了一个新的需求，即在window.onload 函数中打印出页面中的所有节点数量。这当然难不倒我们了。于是我们打开文本编辑器，搜索出window.onload 函数在文件中的位置，在函数内部添加以下代码：

```
window.onload = function(){
    // 原有代码略
    console.log( document.getElementsByTagName( '*' ).length );
};
```

在项目需求变迁的过程中，我们经常会找到相关代码，然后改写它们。这似乎是理所当然的事情，不改动代码怎么满足新的需求呢？想要扩展一个模块，最常用的方式当然是修改它的源代码。如果一个模块不允许修改，那么它的行为常常是固定的。然而，改动代码是一种危险的行为，也许我们都遇到过bug越改越多的场景。刚刚改好了一个bug，但是又在不知不觉中引发了其他的bug。

如果目前的window.onload 函数是一个拥有500行代码的巨型函数，里面密布着各种变量和交叉的业务逻辑，而我们的需求又不仅仅是打印一个log这么简单。那么“改好一个bug，引发其他bug”这样的事情就很可能会发生。我们永远不知道刚刚的改动会有什么副作用，很可能会引发一系列的连锁反应。

那么，有没有办法在不修改代码的情况下，就能满足新需求呢？在第15章中，我们已经得到了答案，通过增加代码，而不是修改代码的方式，来给window.onload 函数添加新的功能，代码如下：

```
Function.prototype.after = function( afterfn ){
    var __self = this;
    return function(){
        var ret = __self.apply( this, arguments );
```

```
        afterfn.apply( this, arguments );
        return ret;
    }
};

window.onload = ( window.onload || function(){} ).after(function(){
    console.log( document.getElementsByTagName( '*' ).length );
});
```

通过动态装饰函数的方式，我们完全不用理会从前window.onload 函数的内部实现，无论它的实现优雅或是丑陋。就算我们作为维护者，拿到的是一份混淆压缩过的代码也没有关系。只要它从前是个稳定运行的函数，那么以后也不会因为我们的新增需求而产生错误。新增的代码和原有的代码可以井水不犯河水。

20.2　开放和封闭

上一节为window.onload 函数扩展功能时，用到了两种方式。一种是修改原有的代码，另一种是增加一段新的代码。使用哪种方式效果更好，已经不言而喻。

现在可以引出开放-封闭原则的思想：当需要改变一个程序的功能或者给这个程序增加新功能的时候，可以使用增加代码的方式，但是不允许改动程序的源代码。

在现实生活中，我们也能找到一些跟开放-封闭原则相关的故事。下面这个故事人尽皆知，且跟肥皂相关。

> 有一家生产肥皂的大企业，从欧洲花巨资引入了一条生产线。这条生产线可以自动完成从原材料加工到包装成箱的整个流程，但美中不足的是，生产出来的肥皂有一定的空盒几率。于是老板又从欧洲找来一支专家团队，花费数百万元改造这一生产线，终于解决了生产出空盒肥皂的问题。
>
> 另一家企业也引入了这条生产线，他们同样遇到了空盒肥皂的问题。但他们的解决办法很简单：用一个大风扇在生产线旁边吹，空盒肥皂就会被吹走。

这个故事告诉我们，相比修改源程序，如果通过增加几行代码就能解决问题，那这显然更加简单和优雅，而且增加代码并不会影响原系统的稳定。讲述这个故事，我们的目的不在于说明风扇的成本有多低，而是想说明，如果使用风扇这样简单的方式可以解决问题，根本没有必要去大动干戈地改造原有的生产线。

20.3 用对象的多态性消除条件分支

过多的条件分支语句是造成程序违反开放-封闭原则的一个常见原因。每当需要增加一个新的if 语句时，都要被迫改动原函数。把if 换成switch-case 是没有用的，这是一种换汤不换药的做法。实际上，每当我们看到一大片的if 或者swtich-case 语句时，第一时间就应该考虑，能否利用对象的多态性来重构它们。

利用对象的多态性来让程序遵守开放-封闭原则，是一个常用的技巧。我们依然选用1.2节中让动物发出叫声的例子。下面先提供一段不符合开放-封闭原则的代码。每当我们增加一种新的动物时，都需要改动makeSound 函数的内部实现：

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

var Duck = function(){};
var Chicken = function(){};

makeSound( new Duck() );       // 输出：嘎嘎嘎
makeSound( new Chicken() );    // 输出：咯咯咯
```

动物世界里增加一只狗之后，makeSound 函数必须改成：

```
var makeSound = function( animal ){
    if ( animal instanceof Duck ){
        console.log( '嘎嘎嘎' );
    }else if ( animal instanceof Chicken ){
        console.log( '咯咯咯' );
    }else if ( animal instanceof Dog ){      // 增加跟狗叫声相关的代码
       console.log('汪汪汪' );
   }
};
```

```
var Dog = function(){};
makeSound( new Dog() );    // 增加一只狗
```

利用多态的思想，我们把程序中不变的部分隔离出来（动物都会叫），然后把可变的部分封装起来（不同类型的动物发出不同的叫声），这样一来程序就具有了可扩展性。当我们想让一只狗发出叫声时，只需增加一段代码即可，而不用去改动原有的makeSound 函数：

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

var Duck = function(){};

Duck.prototype.sound = function(){
    console.log( '嘎嘎嘎' );
};

var Chicken = function(){};

Chicken.prototype.sound = function(){
    console.log( '咯咯咯' );
};

makeSound( new Duck() );     // 嘎嘎嘎
makeSound( new Chicken() );  // 咯咯咯


/********* 增加动物狗，不用改动原有的makeSound函数 ***************/

var Dog = function(){};
Dog.prototype.sound = function(){
    console.log( '汪汪汪' );
};

makeSound( new Dog() );     // 汪汪汪
```

20.4　找出变化的地方

开放-封闭原则是一个看起来比较虚幻的原则，并没有实际的模板教导我们怎样亦步亦趋地实现它。但我们还是能找到一些让程序尽量遵守开放-封闭原则的规律，最明显的就是找出程序中将要发生变化的地方，然后把变化封装起来。

通过封装变化的方式，可以把系统中稳定不变的部分和容易变化的部分隔离开来。在系统的演变过程中，我们只需要替换那些容易变化的部分，如果这些部分是已经被封装好的，那么替换起来也相对容易。而变化部分之外的就是稳定的部分。在系统的演变过程中，稳定的部分是不需要改变的。

在上一节的例子中，由于每种动物的叫声都不同，所以动物具体怎么叫是可变的，于是我们把动物具体怎么叫的逻辑从`makeSound` 函数中分离出来。

而动物都会叫这是不变的，`makeSound` 函数里的实现逻辑只跟动物都会叫有关，这样一来，`makeSound` 就成了一个稳定和封闭的函数。

除了利用对象的多态性之外，还有其他方式可以帮助我们编写遵守开放-封闭原则的代码，下面将详细介绍。

1. 放置挂钩

放置挂钩（hook）也是分离变化的一种方式。我们在程序有可能发生变化的地方放置一个挂钩，挂钩的返回结果决定了程序的下一步走向。这样一来，原本的代码执行路径上就出现了一个分叉路口，程序未来的执行方向被预埋下多种可能性。

翻阅过jQuery源代码的读者也许会留意，jQuery从1.4版本开始，陆续加入了`fixHooks`、`keyHooks`、`mouseHooks`、`cssHooks` 等挂钩。在第11章中我们已经见过`hook` 的作用，Template Method模式中的父类是一个相当稳定的类，它封装了子类的算法骨架和执行步骤。

由于子类的数量是无限制的，总会有一些“个性化”的子类迫使我们不得不去改变已经封装好的算法骨架。于是我们可以在父类中的某个容易变化的

地方放置挂钩，挂钩的返回结果由具体子类决定。这样一来，程序就拥有了变化的可能。

关于模板方法模式中的挂钩应用，可以参考第11章。

2. 使用回调函数

在JavaScript中，函数可以作为参数传递给另外一个函数，这是高阶函数的意义之一。在这种情况下，我们通常会把这个函数称为回调函数。在JavaScript版本的设计模式中，策略模式和命令模式等都可以用回调函数轻松实现。

回调函数是一种特殊的挂钩。我们可以把一部分易于变化的逻辑封装在回调函数里，然后把回调函数当作参数传入一个稳定和封闭的函数中。当回调函数被执行的时候，程序就可以因为回调函数的内部逻辑不同，而产生不同的结果。

比如，我们通过ajax异步请求用户信息之后要做一些事情，请求用户信息的过程是不变的，而获取到用户信息之后要做什么事情，则是可能变化的：

```
var getUserInfo = function( callback ){
    $.ajax( 'http:// xxx.com/getUserInfo', callback );
};

getUserInfo( function( data ){
    console.log( data.userName );
});

getUserInfo( function( data ){
    console.log( data.userId );
});
```

另外一个例子是关于Array.prototype.map 的。在不支持Array.prototype.map 的浏览器中，我们可以简单地模拟实现一个map 函数。

arrayMap 函数的作用是把一个数组“映射”为另外一个数组。映射的步骤是不变的，而映射的规则是可变的，于是我们把这部分规则放在回调函数中，传入arrayMap 函数：

```
var arrayMap = function( ary, callback ){
    var i = 0,
        length = ary.length,
        value,
        ret = [];

    for ( ; i < length; i++ ){
        value = callback( i, ary[ i ] );
        ret.push( value );
    }

    return ret;
}

var a = arrayMap( [ 1, 2, 3 ], function( i, n ){
    return n * 2;
});

var b = arrayMap( [ 1, 2, 3 ], function( i, n ){
    return n * 3;
});

console.log( a );    // 输出：[ 2, 4, 6 ]
console.log( b );    // 输出：[ 3, 6, 9 ]
```

20.5　设计模式中的开放- 封闭原则

有一种说法是，设计模式就是给做的好的设计取个名字。几乎所有的设计模式都是遵守开放-封闭原则的，我们见到的好设计，通常都经得起开放-封闭原则的考验。不管是具体的各种设计模式，还是更抽象的面向对象设计原则，比如单一职责原则、最少知识原则、依赖倒置原则等，都是为了让程序遵守开放-封闭原则而出现的。可以这样说，开放-封闭原则是编写一个好程序的目标，其他设计原则都是达到这个目标的过程。

本章我们已经讨论过装饰者模式是如何遵守开放-封闭原则的，本节将继续例举几个模式，来更深一步地了解设计模式在遵守开放-封闭原则方面做出的努力。

1. 发布-订阅模式

发布-订阅模式用来降低多个对象之间的依赖关系，它可以取代对象之间硬编码的通知机制，一个对象不用再显式地调用另外一个对象的某个接口。当有新的订阅者出现时，发布者的代码不需要进行任何修改；同样当发布者需要改变时，也不会影响到之前的订阅者。

2. 模板方法模式

在第11章中，我们曾提到，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。在一个运用了模板方法模式的程序中，子类的方法种类和执行顺序都是不变的，所以我们把这部分逻辑抽出来放到父类的模板方法里面；而子类的方法具体怎么实现则是可变的，于是把这部分变化的逻辑封装到子类中。通过增加新的子类，便能给系统增加新的功能，并不需要改动抽象父类以及其他的子类，这也是符合开放-封闭原则的。

3. 策略模式

策略模式和模板方法模式是一对竞争者。在大多数情况下，它们可以相互替换使用。模板方法模式基于继承的思想，而策略模式则偏重于组合和委托。

策略模式将各种算法都封装成单独的策略类，这些策略类可以被交换使

用。策略和使用策略的客户代码可以分别独立进行修改而互不影响。我们增加一个新的策略类也非常方便，完全不用修改之前的代码。

4. 代理模式

我们在第6章中举了几个例子，开放-封闭原则在它们之中都得到了体现。拿预加载图片举例，我们现在已有一个给图片设置src 的函数myImage，当我们想为它增加图片预加载功能时，一种做法是改动myImage 函数内部的代码，更好的做法是提供一个代理函数proxyMyImage，代理函数负责图片预加载，在图片预加载完成之后，再将请求转交给原来的myImage 函数，myImage 在这个过程中不需要任何改动。

预加载图片的功能和给图片设置src 的功能被隔离在两个函数里，它们可以单独改变而互不影响。myImage 不知晓代理的存在，它可以继续专注于自己的职责——给图片设置src 。

5. 职责链模式

在第14章的学习中，我们遇到过一个例子，把一个巨大的订单函数分别拆成了500元订单、200元订单以及普通订单的3个函数。这3个函数通过职责链连接在一起，客户的请求会在这条链条里面依次传递：

```
var order500yuan = new Chain(function( orderType, pay, stock ){
    // 具体代码略
});

var order200yuan = new Chain(function( orderType, pay, stock ){
    // 具体代码略
});

var orderNormal = new Chain(function( orderType, pay, stock ){
    // 具体代码略
});

order500yuan.setNextSuccessor( order200yuan ).setNextSuccessor( orderNor
order500yuan.passRequest( 1, true, 10 );    // 500元定金预购，得到100优惠券
```

可以看到，当我们增加一个新类型的订单函数时，不需要改动原有的订单

函数代码，只需要在链条中增加一个新的节点。

20.6 开放- 封闭原则的相对性

在职责链模式代码中，大家也许会产生这个疑问：开放-封闭原则要求我们只能通过增加源代码的方式扩展程序的功能，而不允许修改源代码。那当我们往职责链中增加一个新的100元订单函数节点时，不也必须改动设置链条的代码吗？代码如下：

```
order500yuan.setNextSuccessor( order200yuan ).setNextSuccessor( orderNor
```

变为：

```
order500yuan.setNextSuccessor( order200yuan ).setNextSuccessor( order100
```

实际上，让程序保持完全封闭是不容易做到的。就算技术上做得到，也需要花费太多的时间和精力。而且让程序符合开放-封闭原则的代价是引入更多的抽象层次，更多的抽象有可能会增大代码的复杂度。

更何况，有一些代码是无论如何也不能完全封闭的，总会存在一些无法对其封闭的变化。作为程序员，我们可以做到的有下面两点。

- 挑选出最容易发生变化的地方，然后构造抽象来封闭这些变化。
- 在不可避免发生修改的时候，尽量修改那些相对容易修改的地方。拿一个开源库来说，修改它提供的配置文件，总比修改它的源代码来得简单。

比如在第14章中出现的那个巨大的订单函数，它包含了各种订单的逻辑，有500元和200元的，也有普通订单的。这个函数是最有可能发生变化的，一旦增加新的订单，就必须修改这个巨大的函数。而用职责链模式重构之后，我们只需要新增一个节点，然后重新设置链条中节点的连接顺序。重构后的修改方式显然更加清晰简单 。

20.7　接受第一次愚弄

下面这段话引自Bob大叔的《敏捷软件开发原则、模式与实践》。

> 有句古老的谚语说：“愚弄我一次，应该羞愧的是你。再次愚弄我，应该羞愧的是我。”这也是一种有效的对待软件设计的态度。为了防止软件背着不必要的复杂性，我们会允许自己被愚弄一次。

让程序一开始就尽量遵守开放-封闭原则，并不是一件很容易的事情。一方面，我们需要尽快知道程序在哪些地方会发生变化，这要求我们有一些“未卜先知”的能力。另一方面，留给程序员的需求排期并不是无限的，所以我们可以说服自己去接受不合理的代码带来的第一次愚弄。在最初编写代码的时候，先假设变化永远不会发生，这有利于我们迅速完成需求。当变化发生并且对我们接下来的工作造成影响的时候，可以再回过头来封装这些变化的地方。然后确保我们不会掉进同一个坑里，这有点像星矢说的：“圣斗士不会被同样的招数击倒第二次。”