第 15 章　装饰者模式

我们玩魔兽争霸的任务关时，对15级乱加技能点的野生英雄普遍没有好感，而是喜欢留着技能点，在游戏的进行过程中按需加技能。同样，在程序开发中，许多时候都并不希望某个类天生就非常庞大，一次性包含许多职责。那么我们就可以使用装饰者模式。装饰者模式可以动态地给某个对象添加一些额外的职责，而不会影响从这个类中派生的其他对象。

在传统的面向对象语言中，给对象添加功能常常使用继承的方式，但是继承的方式并不灵活，还会带来许多问题：一方面会导致超类和子类之间存在强耦合性，当超类改变时，子类也会随之改变；另一方面，继承这种功能复用方式通常被称为“白箱复用”，“白箱”是相对可见性而言的，在继承方式中，超类的内部细节是对子类可见的，继承常常被认为破坏了封装性。

使用继承还会带来另外一个问题，在完成一些功能复用的同时，有可能创建出大量的子类，使子类的数量呈爆炸性增长。比如现在有4种型号的自行车，我们为每种自行车都定义了一个单独的类。现在要给每种自行车都装上前灯、尾灯和铃铛这3种配件。如果使用继承的方式来给每种自行车创建子类，则需要 4×3 = 12 个子类。但是如果把前灯、尾灯、铃铛这些对象动态组合到自行车上面，则只需要额外增加3个类。

这种给对象动态地增加职责的方式称为装饰者（decorator）模式 。装饰者模式能够在不改变对象自身的基础上，在程序运行期间给对象动态地添加职责。跟继承相比，装饰者是一种更轻便灵活的做法，这是一种“即用即付”的方式，比如天冷了就多穿一件外套，需要飞行时就在头上插一支竹蜻蜓，遇到一堆食尸鬼时就点开AOE（范围攻击）技能。


天冷了，
要多加件
衣服哟！

15.1 模拟传统面向对象语言的装饰者模式

首先要提出来的是，作为一门解释执行的语言，给JavaScript中的对象动态添加或者改变职责是一件再简单不过的事情，虽然这种做法改动了对象自身，跟传统定义中的装饰者模式并不一样，但这无疑更符合JavaScript的语言特色。代码如下：

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

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

传统面向对象语言中的装饰者模式在JavaScript中适用的场景并不多，如上面代码所示，通常我们并不太介意改动对象自身。尽管如此，本节我们还是稍微模拟一下传统面向对象语言中的装饰者模式实现。

假设我们在编写一个飞机大战的游戏，随着经验值的增加，我们操作的飞机对象可以升级成更厉害的飞机，一开始这些飞机只能发射普通的子弹，升到第二级时可以发射导弹，升到第三级时可以发射原子弹。

下面来看代码实现，首先是原始的飞机类：

```
var Plane = function(){}

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

接下来增加两个装饰类，分别是导弹和原子弹：

```
var MissileDecorator = function( plane ){
```

```
    this.plane = plane;
}

MissileDecorator.prototype.fire = function(){
    this.plane.fire();
    console.log( '发射导弹' );
}

var AtomDecorator = function( plane ){
    this.plane = plane;
}

AtomDecorator.prototype.fire = function(){
    this.plane.fire();
    console.log( '发射原子弹' );
}
```

导弹类和原子弹类的构造函数都接受参数plane 对象，并且保存好这个参数，在它们的fire 方法中，除了执行自身的操作之外，还调用plane 对象的fire 方法。

这种给对象动态增加职责的方式，并没有真正地改动对象自身，而是将对象放入另一个对象之中，这些对象以一条链的方式进行引用，形成一个聚合对象。这些对象都拥有相同的接口（fire 方法），当请求达到链中的某个对象时，这个对象会执行自身的操作，随后把请求转发给链中的下一个对象。

因为装饰者对象和它所装饰的对象拥有一致的接口，所以它们对使用该对象的客户来说是透明的，被装饰的对象也并不需要了解它曾经被装饰过，这种透明性使得我们可以递归地嵌套任意多个装饰者对象，如图15-1所示。

图　15-1

最后看看测试结果：

```
var plane = new Plane();
plane = new MissileDecorator( plane );
plane = new AtomDecorator( plane );

plane.fire();
// 分别输出：  发射普通子弹、发射导弹、发射原子弹
```

15.2　装饰者也是包装器

在《设计模式》成书之前，GoF原想把装饰者（decorator）模式称为包装器（wrapper）模式。

从功能上而言，decorator能很好地描述这个模式，但从结构上看，wrapper的说法更加贴切。装饰者模式将一个对象嵌入另一个对象之中，实际上相当于这个对象被另一个对象包装起来，形成一条包装链。请求随着这条链依次传递到所有的对象，每个对象都有处理这条请求的机会，如图15-2所示。

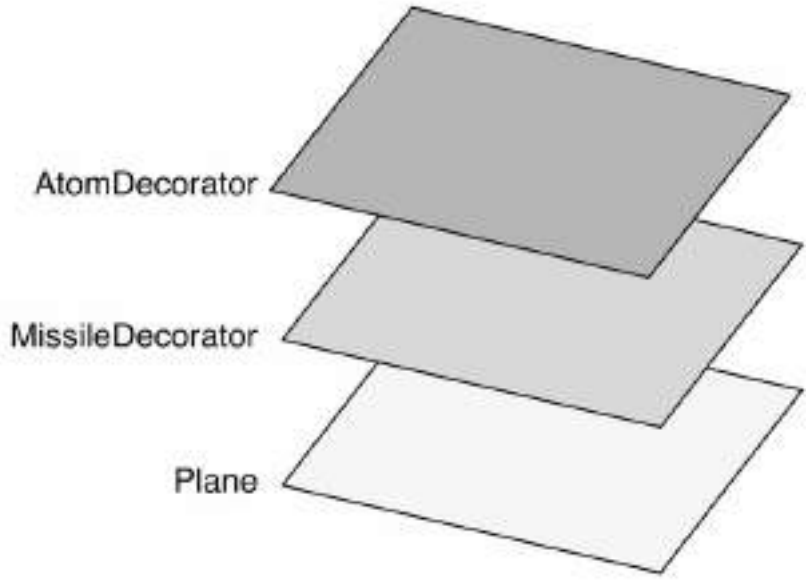

图　15-2

15.3　回到JavaScript的装饰者

JavaScript语言动态改变对象相当容易，我们可以直接改写对象或者对象的某个方法，并不需要使用“类”来实现装饰者模式，代码如下：

```
var plane = {
    fire: function(){
        console.log( '发射普通子弹' );
    }
}

var missileDecorator = function(){
    console.log( '发射导弹' );
}

var atomDecorator = function(){
    console.log( '发射原子弹' );
}

var fire1 = plane.fire;

plane.fire = function(){
    fire1();
    missileDecorator();
}

var fire2 = plane.fire;

plane.fire = function(){
    fire2();
    atomDecorator();
}

plane.fire();
// 分别输出：  发射普通子弹、发射导弹、发射原子弹
```

15.4 装饰函数

在JavaScript中，几乎一切都是对象，其中函数又被称为一等对象。在平时的开发工作中，也许大部分时间都在和函数打交道。在JavaScript中可以很方便地给某个对象扩展属性和方法，但却很难在不改动某个函数源代码的情况下，给该函数添加一些额外的功能。在代码的运行期间，我们很难切入某个函数的执行环境。

要想为函数添加一些功能，最简单粗暴的方式就是直接改写该函数，但这是最差的办法，直接违反了开放-封闭原则：

```
var a = function(){
    alert (1);
}

// 改成：

var a = function(){
    alert (1);
    alert (2);
}
```

很多时候我们不想去碰原函数，也许原函数是由其他同事编写的，里面的实现非常杂乱。甚至在一个古老的项目中，这个函数的源代码被隐藏在一个我们不愿碰触的阴暗角落里。现在需要一个办法，在不改变函数源代码的情况下，能给函数增加功能，这正是开放-封闭原则给我们指出的光明道路。

其实在15.3节的代码中，我们已经找到了一种答案，通过保存原引用的方式就可以改写某个函数：

```
var a = function(){
    alert (1);
}

var _a = a;
```

```
a = function(){
    _a();
    alert (2);
}

a();
```

这是实际开发中很常见的一种做法，比如我们想给window 绑定onload 事件，但是又不确定这个事件是不是已经被其他人绑定过，为了避免覆盖掉之前的window.onload 函数中的行为，我们一般都会先保存好原先的window.onload ，把它放入新的window.onload 里执行：

```
window.onload = function(){
    alert (1);
}

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

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

这样的代码当然是符合开放-封闭原则的，我们在增加新功能的时候，确实没有修改原来的window.onload 代码，但是这种方式存在以下两个问题。

- 必须维护_onload 这个中间变量，虽然看起来并不起眼，但如果函数的装饰链较长，或者需要装饰的函数变多，这些中间变量的数量也会越来越多。
- 其实还遇到了this 被劫持的问题，在window.onload 的例子中没有这个烦恼，是因为调用普通函数_onload 时，this 也指向window ，跟调用window.onload 时一样（函数作为对象的方法

被调用时，this 指向该对象，所以此处this 也只指向window ）。现在把window.onload 换成document.getElementById ，代码如下:

```
var _getElementById = document.getElementById;


document.getElementById = function( id ){
    alert (1);
    return _getElementById( id );       // (1)
}


var button = document.getElementById( 'button' );
</script>
```

执行这段代码，我们看到在弹出alert(1) 之后，紧接着控制台抛出了异常:

```
// 输出:  Uncaught TypeError: Illegal invocation
```

异常发生在(1) 处的_getElementById(id) 这句代码上，此时_getElementById 是一个全局函数，当调用一个全局函数时，this 是指向window 的，而document.getElementById 方法的内部实现需要使用this 引用，this 在这个方法内预期是指向document ，而不是window, 这是错误发生的原因，所以使用现在的方式给函数增加功能并不保险。

改进后的代码可以满足需求，我们要手动把document 当作上下文this 传入_getElementById :

```
<html>
    <button id="button"></button>
    <script>
```

```
    var _getElementById = document.getElementById;

    document.getElementById = function(){
        alert (1);
        return _getElementById.apply( document, arguments );
    }

    var button = document.getElementById( 'button' );
    </script>
</html>
```

但这样做显然很不方便，下面我们引入本书3.7节介绍过的AOP，来提供一种完美的方法给函数动态增加功能。

15.5 用AOP装饰函数

首先给出Function.prototype.before 方法和 Function.prototype.after 方法：

```
Function.prototype.before = function( beforefn ){
    var __self = this;  // 保存原函数的引用
    return function(){     // 返回包含了原函数和新函数的"代理"函数
        beforefn.apply( this, arguments );  // 执行新函数，且保证this不被劫持
                                            // 也会被原封不动地传入原函数，新
        return __self.apply( this, arguments );  // 执行原函数并返回原函数的
                                                 // 并且保证this不被劫持
    }
}

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

Function.prototype.before 接受一个函数当作参数，这个函数即为新添加的函数，它装载了新添加的功能代码。

接下来把当前的this 保存起来，这个this 指向原函数，然后返回一个“代理”函数，这个“代理”函数只是结构上像代理而已，并不承担代理的职责（比如控制对象的访问等）。它的工作是把请求分别转发给新添加的函数和原函数，且负责保证它们的执行顺序，让新添加的函数在原函数之前执行（前置装饰），这样就实现了动态装饰的效果。

我们注意到，通过Function.prototype.apply 来动态传入正确的 this ，保证了函数在被装饰之后，this 不会被劫持。

Function.prototype.after 的原理跟

Function.prototype.before 一模一样，唯一不同的地方在于让新添加的函数在原函数执行之后再执行。

下面来试试用Function.prototype.before 的威力：

```
<html>
    <button id="button"></button>
    <script>
    Function.prototype.before = function( beforefn ){
        var __self = this;
        return function(){
            beforefn.apply( this, arguments );
            return __self.apply( this, arguments );
        }
    }

    document.getElementById = document.getElementById.before(function(){
        alert (1);
    });

    var button = document.getElementById( 'button' );
    console.log( button );
    </script>
</html>
```

再回到window.onload 的例子，看看用Function.prototype.after 来增加新的window.onload 事件是多么简单：

```
window.onload = function(){
    alert (1);
}

window.onload = ( window.onload || function(){} ).after(function(){
    alert (2);
}).after(function(){
    alert (3);
}).after(function(){
    alert (4);
});
```

值得提到的是，上面的AOP实现是在Function.prototype 上添加before 和after 方法，但许多人不喜欢这种污染原型的方式，那么我们可以做一些变通，把原函数和新函数都作为参数传入before 或者after 方法：

```
var before = function( fn, beforefn ){
    return function(){
        beforefn.apply( this, arguments );
        return fn.apply( this, arguments );
    }
}

var a = before(
    function(){alert (3)},
    function(){alert (2)}
);

a = before( a, function(){alert (1);} );
a();
```

15.6 AOP的应用实例

用AOP装饰函数的技巧在实际开发中非常有用。不论是业务代码的编写，还是在框架层面，我们都可以把行为依照职责分成粒度更细的函数，随后通过装饰把它们合并到一起，这有助于我们编写一个松耦合和高复用性的系统。

这一节将介绍几个例子，带大家进一步理解装饰函数的威力。

15.6.1 数据统计上报

分离业务代码和数据统计代码，无论在什么语言中，都是AOP的经典应用之一。在项目开发的结尾阶段难免要加上很多统计数据的代码，这些过程可能让我们被迫改动早已封装好的函数。

比如页面中有一个登录button，点击这个button会弹出登录浮层，与此同时要进行数据上报，来统计有多少用户点击了这个登录button：

```
<html>
    <button tag="login" id="button">点击打开登录浮层</button>
    <script>

    var showLogin = function(){
        console.log( '打开登录浮层' );
        log( this.getAttribute( 'tag' ) );
    }

    var log = function( tag ){
        console.log( '上报标签为: ' + tag );
        // (new Image).src = 'http:// xxx.com/report?tag=' + tag;    //
    }

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

    </script>
</html>
```

我们看到在showLogin 函数里，既要负责打开登录浮层，又要负责数据上报，这是两个层面的功能，在此处却被耦合在一个函数里。使用AOP分离之后，代码如下：

```
<html>
    <button tag="login" id="button">点击打开登录浮层</button>
    <script>

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

    var showLogin = function(){
        console.log( '打开登录浮层' );
    }

    var log = function(){
        console.log( '上报标签为: ' + this.getAttribute( 'tag' ) );
    }

    showLogin = showLogin.after( log );    // 打开登录浮层之后上报数据

    document.getElementById( 'button' ).onclick = showLogin;
    </script>
</html>
```

15.6.2　用AOP动态改变函数的参数

观察Function.prototype.before 方法：

```
Function.prototype.before = function( beforefn ){
    var __self = this;
    return function(){
        beforefn.apply( this, arguments );        // (1)
        return __self.apply( this, arguments );    // (2)
    }
}
```

从这段代码的(1)处和(2)处可以看到，beforefn 和原函数__self 共用一组参数列表arguments ，当我们在beforefn 的函数体内改变 arguments 的时候，原函数__self 接收的参数列表自然也会变化。

下面的例子展示了如何通过Function.prototype.before 方法给函数func 的参数param 动态地添加属性b：

```
var func = function( param ){
    console.log( param );    // 输出： {a: "a", b: "b"}
}

func = func.before( function( param ){
    param.b = 'b';
});

func( {a: 'a'} );
```

现在有一个用于发起ajax请求的函数，这个函数负责项目中所有的ajax异步请求：

```
var ajax = function( type, url, param ){
    console.dir(param);
    // 发送ajax请求的代码略
};

ajax( 'get', 'http:// xxx.com/userinfo', { name: 'sven' } );
```

上面的伪代码表示向后台cgi发起一个请求来获取用户信息，传递给cgi的参数是{ name: 'sven' } 。

ajax函数在项目中一直运转良好，跟cgi的合作也很愉快。直到有一天，我们的网站遭受了CSRF攻击。解决CSRF攻击最简单的一个办法就是在HTTP请求中带上一个Token 参数。

假设我们已经有一个用于生成Token 的函数：

```
var getToken = function(){
    return 'Token';
}
```

现在的任务是给每个ajax请求都加上Token 参数：

```
var ajax = function( type, url, param ){
    param = param || {};
    Param.Token = getToken();      // 发送ajax请求的代码略...
};
```

虽然已经解决了问题，但我们的ajax 函数相对变得僵硬了，每个从ajax函数里发出的请求都自动带上了Token 参数，虽然在现在的项目中没有什么问题，但如果将来把这个函数移植到其他项目上，或者把它放到一个开源库中供其他人使用，Token 参数都将是多余的。

也许另一个项目不需要验证Token ，或者是Token 的生成方式不同，无论是哪种情况，都必须重新修改ajax 函数。

为了解决这个问题，先把ajax 函数还原成一个干净的函数：

```
var ajax= function( type, url, param ){
    console.log(param);      // 发送ajax请求的代码略
};
```

然后把Token 参数通过Function.prototyte.before 装饰到ajax 函数的参数param 对象中：

```
var getToken = function(){
    return 'Token';
}

ajax = ajax.before(function( type, url, param ){
    param.Token = getToken();
});

ajax( 'get', 'http:// xxx.com/userinfo', { name: 'sven' } );
```

从ajax 函数打印的log可以看到，Token 参数已经被附加到了ajax 请求的参数中：

```
{name: "sven", Token: "Token"}
```

明显可以看到，用AOP的方式给ajax 函数动态装饰上Token 参数，保证了ajax函数是一个相对纯净的函数，提高了ajax 函数的可复用性，它在被迁往其他项目的时候，不需要做任何修改。

15.6.3 插件式的表单验证

我们很多人都写过许多表单验证的代码，在一个Web项目中，可能存在非常多的表单，如注册、登录、修改用户信息等。在表单数据提交给后台之前，常常要做一些校验，比如登录的时候需要验证用户名和密码是否为空，代码如下：

```
<html>
    <body>
        用户名：<input id="username" type="text"/>
        密码：  <input id="password" type="password"/>
        <input id="submitBtn" type="button" value="提交"/>
```

```
    </body>
    <script>
    var username = document.getElementById( 'username' ),
        password = document.getElementById( 'password' ),
        submitBtn = document.getElementById( 'submitBtn' );

        var formSubmit = function(){
            if ( username.value === '' ){
                return alert ( '用户名不能为空' );
            }
            if ( password.value === '' ){
                return alert ( '密码不能为空' );
            }

            var param = {
                username: username.value,
                password: password.value
            }
            ajax( 'http:// xxx.com/login', param );    // ajax具体实现略
        }

        submitBtn.onclick = function(){
            formSubmit();
        }
    </script>
</html>
```

formSubmit 函数在此处承担了两个职责，除了提交ajax请求之外，还要验证用户输入的合法性。这种代码一来会造成函数臃肿，职责混乱，二来谈不上任何可复用性。

本节的目的是分离校验输入和提交ajax请求的代码，我们把校验输入的逻辑放到validata 函数中，并且约定当validata 函数返回false 的时候，表示校验未通过，代码如下：

```
var validata = function(){
    if ( username.value === '' ){
        alert ( '用户名不能为空' );
        return false;
    }
    if ( password.value === '' ){
        alert ( '密码不能为空' );
```

```
        return false;
    }
}

var formSubmit = function(){
    if ( validata() === false ){    // 校验未通过
        return;
    }
    var param = {
        username: username.value,
        password: password.value
    }
    ajax( 'http:// xxx.com/login', param );
}

submitBtn.onclick = function(){
    formSubmit();
}
```

现在的代码已经有了一些改进，我们把校验的逻辑都放到了validata 函数中，但formSubmit 函数的内部还要计算validata 函数的返回值，因为返回值的结果表明了是否通过校验。

接下来进一步优化这段代码，使validata 和formSubmit 完全分离开来。首先要改写Function.prototype.before ， 如果beforefn 的执行结果返回false ，表示不再执行后面的原函数，代码如下：

```
Function.prototype.before = function( beforefn ){
    var __self = this;
    return function(){
        if ( beforefn.apply( this, arguments ) === false ){
            // beforefn返回false的情况直接return，不再执行后面的原函数
            return;
        }
        return __self.apply( this, arguments );
    }
}

var validata = function(){
    if ( username.value === '' ){
        alert ( '用户名不能为空' );
        return false;
```

```
    }
    if ( password.value === '' ){
        alert ( '密码不能为空' );
        return false;
    }
}

var formSubmit = function(){
    var param = {
        username: username.value,
        password: password.value
    }
    ajax( 'http:// xxx.com/login', param );
}

formSubmit = formSubmit.before( validata );

submitBtn.onclick = function(){
    formSubmit();
}
```

在这段代码中，校验输入和提交表单的代码完全分离开来，它们不再有任何耦合关系，formSubmit = formSubmit.before(validata) 这句代码，如同把校验规则动态接在formSubmit 函数之前，validata 成为一个即插即用的函数，它甚至可以被写成配置文件的形式，这有利于我们分开维护这两个函数。再利用策略模式稍加改造，我们就可以把这些校验规则都写成插件的形式，用在不同的项目当中。

值得注意的是，因为函数通过Function.prototype.before 或者Function.prototype.after 被装饰之后，返回的实际上是一个新的函数，如果在原函数上保存了一些属性，那么这些属性会丢失。代码如下：

```
var func = function(){
    alert( 1 );
}
func.a = 'a';

func = func.after( function(){
    alert( 2 );
});
```

```
alert ( func.a );   // 输出：undefined
```

另外，这种装饰方式也叠加了函数的作用域，如果装饰的链条过长，性能上也会受到一些影响。

15.7　装饰者模式和代理模式

装饰者模式和第6章代理模式的结构看起来非常相像，这两种模式都描述了怎样为对象提供一定程度上的间接引用，它们的实现部分都保留了对另外一个对象的引用，并且向那个对象发送请求。

代理模式和装饰者模式最重要的区别在于它们的意图和设计目的。代理模式的目的是，当直接访问本体不方便或者不符合需要时，为这个本体提供一个替代者。本体定义了关键功能，而代理提供或拒绝对它的访问，或者在访问本体之前做一些额外的事情。装饰者模式的作用就是为对象动态加入行为。换句话说，代理模式强调一种关系（Proxy与它的实体之间的关系），这种关系可以静态的表达，也就是说，这种关系在一开始就可以被确定。而装饰者模式用于一开始不能确定对象的全部功能时。代理模式通常只有一层代理-本体的引用，而装饰者模式经常会形成一条长长的装饰链。

在虚拟代理实现图片预加载的例子中，本体负责设置img节点的src，代理则提供了预加载的功能，这看起来也是“加入行为”的一种方式，但这种加入行为的方式和装饰者模式的偏重点是不一样的。装饰者模式是实实在在的为对象增加新的职责和行为，而代理做的事情还是跟本体一样，最终都是设置src。但代理可以加入一些“聪明”的功能，比如在图片真正加载好之前，先使用一张占位的loading图片反馈给客户。

15.8 小结

本章通过数据上报、统计函数的执行时间、动态改变函数参数以及插件式的表单验证这4个例子，我们了解了装饰函数，它是JavaScript中独特的装饰者模式。这种模式在实际开发中非常有用，除了上面提到的例子，它在框架开发中也十分有用。作为框架作者，我们希望框架里的函数提供的是一些稳定而方便移植的功能，那些个性化的功能可以在框架之外动态装饰上去，这可以避免为了让框架拥有更多的功能，而去使用一些`if`、`else`语句预测用户的实际需要。