第 22 章　代码重构

本书并非志在讨论重构的书，但我们到目前为止，实际上一直在不停地进行代码级别上的优化。在讲设计模式的章节中，我们总是先写一段反例代码，而后再介绍一段通过设计模式重构之后的更好的代码。这种强烈的对比会加深我们对该模式的理解。

模式和重构之间有着一种与生俱来的关系。从某种角度来看，设计模式的目的就是为许多重构行为提供目标。

在实际的项目开发中，除了使用设计模式进行重构之外，还有一些常见而容易忽略的细节，这些细节也是帮助我们达到重构目标的重要手段。本章将挑选一些进行介绍，其中有一部分思想来自Martin Fowler的名著《重构：改善既有代码的设计》，虽然该书是使用Java语言写成的，但这些重构的技巧，有很大一部分可以为JavaScript语言所借鉴。

虽然本章会提出一些重构的目标和手段，但它们都是建议，没有哪些是必须严格遵守的标准。具体是否需要重构，以及如何进行重构，这需要我们根据系统的类型、项目工期、人力等外界因素一起决定。

22.1 提炼函数

在JavaScript开发中，我们大部分时间都在与函数打交道，所以我们希望这些函数有着良好的命名，函数体内包含的逻辑清晰明了。如果一个函数过长，不得不加上若干注释才能让这个函数显得易读一些，那这些函数就很有必要进行重构。

如果在函数中有一段代码可以被独立出来，那我们最好把这些代码放进另外一个独立的函数中。这是一种很常见的优化工作，这样做的好处主要有以下几点。

- 避免出现超大函数。
- 独立出来的函数有助于代码复用。
- 独立出来的函数更容易被覆写。
- 独立出来的函数如果拥有一个良好的命名，它本身就起到了注释的作用。

比如在一个负责取得用户信息的函数里面，我们还需要打印跟用户信息有关的log，那么打印log的语句就可以被封装在一个独立的函数里：

```
var getUserInfo = function(){
    ajax( 'http:// xxx.com/userInfo', function( data ){
        console.log( 'userId: ' + data.userId );
        console.log( 'userName: ' + data.userName );
        console.log( 'nickName: ' + data.nickName );
    });
};
```

改成：

```
var getUserInfo = function(){
    ajax( 'http:// xxx.com/userInfo', function( data ){
        printDetails( data );
```

```
    });
};

var printDetails = function( data ){
    console.log( 'userId: ' + data.userId );
    console.log( 'userName: ' + data.userName );
    console.log( 'nickName: ' + data.nickName );
};
```

22.2 合并重复的条件片段

如果一个函数体内有一些条件分支语句，而这些条件分支语句内部散布了一些重复的代码，那么就有必要进行合并去重工作。假如我们有一个分页函数paging，该函数接收一个参数currPage，currPage表示即将跳转的页码。在跳转之前，为防止currPage传入过小或者过大的数字，我们要手动对它的值进行修正，详见如下伪代码：

```
var paging = function( currPage ){
    if ( currPage <= 0 ){
        currPage = 0;
        jump( currPage );    // 跳转
    }else if ( currPage >= totalPage ){
        currPage = totalPage;
        jump( currPage );    // 跳转
    }else{
        jump( currPage );    // 跳转
    }
};
```

可以看到，负责跳转的代码jump(currPage)在每个条件分支内都出现了，所以完全可以把这句代码独立出来：

```
var paging = function( currPage ){
    if ( currPage <= 0 ){
        currPage = 0;
    }else if ( currPage >= totalPage ){
        currPage = totalPage;
    }
    jump( currPage );    // 把jump函数独立出来
};
```

22.3 把条件分支语句提炼成函数

在程序设计中，复杂的条件分支语句是导致程序难以阅读和理解的重要原因，而且容易导致一个庞大的函数。假设现在有一个需求是编写一个计算商品价格的getPrice函数，商品的计算只有一个规则：如果当前正处于夏季，那么全部商品将以8折出售。代码如下：

```
var getPrice = function( price ){
    var date = new Date();
    if ( date.getMonth() >= 6 && date.getMonth() <= 9 ){    // 夏天
        return price * 0.8;
    }
    return price;
};
```

观察这句代码：

```
if ( date.getMonth() >= 6 && date.getMonth() <= 9 ){
    // ...
}
```

这句代码要表达的意思很简单，就是判断当前是否正处于夏天（7~10月）。尽管这句代码很短小，但代码表达的意图和代码自身还存在一些距离，阅读代码的人必须要多花一些精力才能明白它传达的意图。其实可以把这句代码提炼成一个单独的函数，既能更准确地表达代码的意思，函数名本身又能起到注释的作用。代码如下：

```
var isSummer = function(){
    var date = new Date();
    return date.getMonth() >= 6 && date.getMonth() <= 9;
};

var getPrice = function( price ){
    if ( isSummer() ){    // 夏天
```

```
        return price * 0.8;
    }
    return price;
};
```

22.4 合理使用循环

在函数体内，如果有些代码实际上负责的是一些重复性的工作，那么合理利用循环不仅可以完成同样的功能，还可以使代码量更少。下面有一段创建XHR 对象的代码，为了简化示例，我们只考虑版本9以下的IE浏览器，代码如下：

```
var createXHR = function(){
    var xhr;
    try{
        xhr = new ActiveXObject( 'MSXML2.XMLHttp.6.0' );
    }catch(e){
        try{
            xhr = new ActiveXObject( 'MSXML2.XMLHttp.3.0' );
        }catch(e){
            xhr = new ActiveXObject( 'MSXML2.XMLHttp' );
        }
    }
    return xhr;
};

var xhr = createXHR();
```

下面我们灵活地运用循环，可以得到跟上面代码一样的效果：

```
var createXHR = function(){
var versions= [ 'MSXML2.XMLHttp.6.0ddd', 'MSXML2.XMLHttp.3.0', 'MSXML2.X
    for ( var i = 0, version; version = versions[ i++ ]; ){
        try{
            return new ActiveXObject( version );
        }catch(e){

        }
    }
};

var xhr = createXHR();
```

22.5 提前让函数退出代替嵌套条件分支

许多程序员都有这样一种观念：“每个函数只能有一个入口和一个出口。”现代编程语言都会限制函数只有一个入口。但关于“函数只有一个出口”，往往会有一些不同的看法。

下面这段伪代码是遵守“函数只有一个出口的”的典型代码：

```
var del = function( obj ){
    var ret;
    if ( !obj.isReadOnly ){     // 不为只读的才能被删除
        if ( obj.isFolder ){     // 如果是文件夹
            ret = deleteFolder( obj );
        }else if ( obj.isFile ){     // 如果是文件
            ret = deleteFile( obj );
        }
    }
    return ret;
};
```

嵌套的条件分支语句绝对是代码维护者的噩梦，对于阅读代码的人来说，嵌套的if 、else 语句相比平铺的if 、else ，在阅读和理解上更加困难，有时候一个外层if 分支的左括号和右括号之间相隔500米之远。用《重构》里的话说，嵌套的条件分支往往是由一些深信“每个函数只能有一个出口的”程序员写出的。但实际上，如果对函数的剩余部分不感兴趣，那就应该立即退出。引导阅读者去看一些没有用的else 片段，只会妨碍他们对程序的理解。

于是我们可以挑选一些条件分支，在进入这些条件分支之后，就立即让这个函数退出。要做到这一点，有一个常见的技巧，即在面对一个嵌套的if 分支时，我们可以把外层if 表达式进行反转。重构后的del 函数如下：

```
var del = function( obj ){
    if ( obj.isReadOnly ){     // 反转if表达式
        return;
    }
```

```
    if ( obj.isFolder ){
        return deleteFolder( obj );
    }
    if ( obj.isFile ){
        return deleteFile( obj );
    }
};
```

22.6 传递对象参数代替过长的参数列表

有时候一个函数有可能接收多个参数，而参数的数量越多，函数就越难理解和使用。使用该函数的人首先得搞明白全部参数的含义，在使用的时候，还要小心翼翼，以免少传了某个参数或者把两个参数搞反了位置。如果我们想在第3个参数和第4个参数之中增加一个新的参数，就会涉及许多代码的修改，代码如下：

```
var setUserInfo = function( id, name, address, sex, mobile, qq ){
    console.log( 'id= ' + id );
    console.log( 'name= ' +name );
    console.log( 'address= ' + address );
    console.log( 'sex= ' + sex );
    console.log( 'mobile= ' + mobile );
    console.log( 'qq= ' + qq );
};

setUserInfo( 1314, 'sven', 'shenzhen', 'male', '137********', 377876679
```

这时我们可以把参数都放入一个对象内，然后把该对象传入`setUserInfo`函数，`setUserInfo`函数需要的数据可以自行从该对象里获取。现在不用再关心参数的数量和顺序，只要保证参数对应的`key`值不变就可以了：

```
var setUserInfo = function( obj ){
    console.log( 'id= ' + obj.id );
    console.log( 'name= ' + obj.name );
    console.log( 'address= ' + obj.address );
    console.log( 'sex= ' + obj.sex );
    console.log( 'mobile= ' + obj.mobile );
    console.log( 'qq= ' + obj.qq );
};

setUserInfo({
    id: 1314,
    name: 'sven',
    address: 'shenzhen',
    sex: 'male',
    mobile: '137********',
```

22.7 尽量减少参数数量

如果调用一个函数时需要传入多个参数，那这个函数是让人望而生畏的，我们必须搞清楚这些参数代表的含义，必须小心翼翼地把它们按照顺序传入该函数。而如果一个函数不需要传入任何参数就可以使用，这种函数是深受人们喜爱的。在实际开发中，向函数传递参数不可避免，但我们应该尽量减少函数接收的参数数量。下面举个非常简单的示例。 有一个画图函数draw ，它现在只能绘制正方形，接收了3个参数，分别是图形的width 、heigth 以及square ：

```
var draw = function( width, height, square ){};
```

但实际上正方形的面积是可以通过width 和height 计算出来的，于是我们可以把参数square 从draw 函数中去掉：

```
var draw = function( width, height ){
    var square = width * height;
};
```

假设以后这个draw 函数开始支持绘制圆形，我们需要把参数width 和height 换成半径radius ， 但图形的面积square 始终不应该由客户传入，而是应该在draw 函数内部，由传入的参数加上一定的规则计算得来。此时，我们可以使用策略模式，让draw 函数成为一个支持绘制多种图形的函数。

22.8　少用三目运算符

有一些程序员喜欢大规模地使用三目运算符，来代替传统的`if`、`else`。理由是三目运算符性能高，代码量少。不过，这两个理由其实都很难站得住脚。

即使我们假设三目运算符的效率真的比`if`、`else`高，这点差距也是完全可以忽略不计的。在实际的开发中，即使把一段代码循环一百万次，使用三目运算符和使用`if`、`else`的时间开销处在同一个级别里。

同样，相比损失的代码可读性和可维护性，三目运算符节省的代码量也可以忽略不计。让JS文件加载更快的办法有很多种，如压缩、缓存、使用CDN和分域名等。把注意力只放在使用三目运算符节省的字符数量上，无异于一个300斤重的人把超重的原因归罪于头皮屑。

如果条件分支逻辑简单且清晰，这无碍我们使用三目运算符：

```
var global = typeof window !== "undefined" ? window : this;
```

但如果条件分支逻辑非常复杂，如下段代码所示，那我们最好的选择还是按部就班地编写`if`、`else`。`if`、`else`语句的好处很多，一是阅读相对容易，二是修改的时候比修改三目运算符周围的代码更加方便：

```
if ( !aup || !bup ) {
    return a === doc ? -1 :
        b === doc ? 1 :
        aup ? -1 :
        bup ? 1 :
        sortInput ?
        ( indexOf.call( sortInput, a ) - indexOf.call( sortInput, b ) ) :
        0;
}
```

22.9 合理使用链式调用

经常使用jQuery的程序员相当习惯链式调用方法，在JavaScript中，可以很容易地实现方法的链式调用，即让方法调用结束后返回对象自身，如下代码所示：

```
var User = function(){
    this.id = null;
    this.name = null;
};

User.prototype.setId = function( id ){
    this.id = id;
    return this;
};

User.prototype.setName = function( name ){
    this.name = name;
    return this;
};

console.log( new User().setId( 1314 ).setName( 'sven' ) );
```

或者：

```
var User = {
    id: null,
    name: null,
    setId: function( id ){
        this.id = id;
        return this;
    },
    setName: function( name ){
        this.name = name;
        return this;
    }
};

console.log( User.setId( 1314 ).setName( 'sven' ) );
```

使用链式调用的方式并不会造成太多阅读上的困难，也确实能省下一些字符和中间变量，但节省下来的字符数量同样是微不足道的。链式调用带来的坏处就是在调试的时候非常不方便，如果我们知道一条链中有错误出现，必须得先把这条链拆开才能加上一些调试log或者增加断点，这样才能定位错误出现的地方。

如果该链条的结构相对稳定，后期不易发生修改，那么使用链式调用无可厚非。但如果该链条很容易发生变化，导致调试和维护困难，那么还是建议使用普通调用的形式：

```
var user = new User();

user.setId( 1314 );
user.setName( 'sven' );
```

22.10 分解大型类

在我编写的HTML5版“街头霸王”的第一版代码中，负责创建游戏人物的 Spirit 类非常庞大，不仅要负责创建人物精灵，还包括了人物的攻击、防御等动作方法，代码如下：

```
var Spirit = function( name ){
    this.name = name;
};

Spirit.prototype.attack = function( type ){    // 攻击
    if ( type === 'waveBoxing' ){
        console.log( this.name + ': 使用波动拳' );
    }else if( type === 'whirlKick' ){
        console.log( this.name + ': 使用旋风腿' );
    }
};

var spirit = new Spirit( 'RYU' );

spirit.attack( 'waveBoxing' );      // 输出：RYU：使用波动拳
spirit.attack( 'whirlKick' );    // 输出：RYU：使用旋风腿
```

后来发现，Spirit.prototype.attack 这个方法实现是太庞大了，实际上它完全有必要作为一个单独的类存在。面向对象设计鼓励将行为分布在合理数量的更小对象之中：

```
var Attack = function( spirit ){
    this.spirit = spirit;
};

Attack.prototype.start = function( type ){
    return this.list[ type ].call( this );
};

Attack.prototype.list = {
    waveBoxing: function(){
        console.log( this.spirit.name + ': 使用波动拳' );
    },
    whirlKick: function(){
```

```
        console.log( this.spirit.name + ': 使用旋风腿' );
    }
};
```

现在的Spirit 类变得精简了很多，不再包括各种各样的攻击方法，而是把攻击动作委托给Attack 类的对象来执行，这段代码也是策略模式的运用之一：

```
var Spirit = function( name ){
    this.name = name;
    this.attackObj = new Attack( this );
};

Spirit.prototype.attack = function( type ){    // 攻击
    this.attackObj.start( type );
};

var spirit = new Spirit( 'RYU' );

spirit.attack( 'waveBoxing' );    // 输出：RYU：使用波动拳
spirit.attack( 'whirlKick' );    // 输出：RYU：使用旋风腿
```

22.11 用return 退出多重循环

假设在函数体内有一个两重循环语句，我们需要在内层循环中判断，当达到某个临界条件时退出外层的循环。我们大多数时候会引入一个控制标记变量：

```
var func = function(){
    var flag = false;
    for ( var i = 0; i < 10; i++ ){
        for ( var j = 0; j < 10; j++ ){
            if ( i * j >30 ){
                flag = true;
                break;
            }
        }
        if ( flag === true ){
            break;
        }
    }
};
```

第二种做法是设置循环标记：

```
var func = function(){
    outerloop:
    for ( var i = 0; i < 10; i++ ){
        innerloop:
        for ( var j = 0; j < 10; j++ ){
            if ( i * j >30 ){
                break outerloop;
            }
        }
    }
};
```

这两种做法无疑都让人头晕目眩，更简单的做法是在需要中止循环的时候

直接退出整个方法：

```
var func = function(){
    for ( var i = 0; i < 10; i++ ){
        for ( var j = 0; j < 10; j++ ){
            if ( i * j >30 ){
                return;
            }
        }
    }
};
```

当然用return 直接退出方法会带来一个问题，如果在循环之后还有一些将被执行的代码呢？如果我们提前退出了整个方法，这些代码就得不到被执行的机会：

```
var func = function(){
    for ( var i = 0; i < 10; i++ ){
        for ( var j = 0; j < 10; j++ ){
            if ( i * j >30 ){
                return;
            }
        }
    }
    console.log( i );     // 这句代码没有机会被执行
};
```

为了解决这个问题，我们可以把循环后面的代码放到return 后面，如果代码比较多，就应该把它们提炼成一个单独的函数：

```
var print = function( i ){
    console.log( i );
};

var func = function(){
    for ( var i = 0; i < 10; i++ ){
        for ( var j = 0; j < 10; j++ ){
```

```
            if ( i * j >30 ){
                return print( i );
            }
        }
    }
};

func();
```