第 18 章　单一职责原则

就一个类而言，应该仅有一个引起它变化的原因。在JavaScript中，需要用到类的场景并不太多，单一职责原则更多地是被运用在对象或者方法级别上，因此本节我们的讨论大多基于对象和方法。

单一职责原则（SRP）的职责被定义为“引起变化的原因”。如果我们有两个动机去改写一个方法，那么这个方法就具有两个职责。每个职责都是变化的一个轴线，如果一个方法承担了过多的职责，那么在需求的变迁过程中，需要改写这个方法的可能性就越大。

此时，这个方法通常是一个不稳定的方法，修改代码总是一件危险的事情，特别是当两个职责耦合在一起的时候，一个职责发生变化可能会影响到其他职责的实现，造成意想不到的破坏，这种耦合性得到的是低内聚和脆弱的设计。

因此，SRP原则体现为：一个对象（方法）只做一件事情。

18.1 设计模式中的SRP原则

SRP原则在很多设计模式中都有着广泛的运用，例如代理模式、迭代器模式、单例模式和装饰者模式。

1. 代理模式

我们在第6章中已经见过这个图片预加载的例子了。通过增加虚拟代理的方式，把预加载图片的职责放到代理对象中，而本体仅仅负责往页面中添加img 标签，这也是它最原始的职责。

myImage 负责往页面中添加img 标签：

```
var myImage = (function(){
    var imgNode = document.createElement( 'img' );
    document.body.appendChild( imgNode );
    return {
        setSrc: function( src ){
            imgNode.src = src;
        }
    }
})();
```

proxyImage 负责预加载图片，并在预加载完成之后把请求交给本体myImage：

```
var proxyImage = (function(){
    var img = new Image;
    img.onload = function(){
        myImage.setSrc( this.src );
    }
    return {
        setSrc: function( src ){
            myImage.setSrc( 'file:// /C:/Users/svenzeng/Desktop/loading.
            img.src = src;
        }
    }
})();
```

```
proxyImage.setSrc( 'http:// imgcache.qq.com/music/photo/000GGDys0yA0Nk.j
```

把添加img 标签的功能和预加载图片的职责分开放到两个对象中，这两个对象各自都只有一个被修改的动机。在它们各自发生改变的时候，也不会影响另外的对象。

2. 迭代器模式

我们有这样一段代码，先遍历一个集合，然后往页面中添加一些div，这些div的innerHTML 分别对应集合里的元素：

```
var appendDiv = function( data ){
    for ( var i = 0, l = data.length; i < l; i++ ){
        var div = document.createElement( 'div' );
        div.innerHTML = data[ i ];
        document.body.appendChild( div );
    }
};

appendDiv( [ 1, 2, 3, 4, 5, 6 ] );
```

这其实是一段很常见的代码，经常用于ajax请求之后，在回调函数中遍历ajax请求返回的数据，然后在页面中渲染节点。

appendDiv 函数本来只是负责渲染数据，但是在这里它还承担了遍历聚合对象data 的职责。我们想象一下，如果有一天cgi返回的data 数据格式从array 变成了object ，那我们遍历data 的代码就会出现问题，必须改成 for (var i in data) 的方式，这时候必须去修改appendDiv 里的代码，否则因为遍历方式的改变，导致不能顺利往页面中添加div节点。

我们有必要把遍历data 的职责提取出来，这正是迭代器模式的意义，迭代器模式提供了一种方法来访问聚合对象，而不用暴露这个对象的内部表

示。

当把迭代聚合对象的职责单独封装在each 函数中后，即使以后还要增加新的迭代方式，我们只需要修改each 函数即可，appendDiv 函数不会受到牵连，代码如下：

```
var each = function( obj, callback ) {
    var value,
            i = 0,
        length = obj.length,
        isArray = isArraylike( obj );    // isArraylike函数未实现，可以翻阅

    if ( isArray ) {    // 迭代类数组
         for ( ; i < length; i++ ) {
             callback.call( obj[ i ], i, obj[ i ] );
         }
    } else {
         for ( i in obj ) {    // 迭代object对象
              value = callback.call( obj[ i ], i, obj[ i ] );
         }
     }
       return obj;
   };

 var appendDiv = function( data ){
    each( data, function( i, n ){
        var div = document.createElement( 'div' );
        div.innerHTML = n;
        document.body.appendChild( div );
    });
 };

appendDiv( [ 1, 2, 3, 4, 5, 6 ] );
appendDiv({a:1,b:2,c:3,d:4} );
```

3. 单例模式

第4章曾实现过一个惰性单例，最开始的代码是这样的：

```
var createLoginLayer = (function(){
    var div;
    return function(){
```

```
        if ( !div ){
            div = document.createElement( 'div' );
            div.innerHTML = '我是登录浮窗';
            div.style.display = 'none';
            document.body.appendChild( div );
        }
        return div;
    }
})();
```

现在我们把管理单例的职责和创建登录浮窗的职责分别封装在两个方法里，这两个方法可以独立变化而互不影响，当它们连接在一起的时候，就完成了创建唯一登录浮窗的功能，下面的代码显然是更好的做法：

```
var getSingle = function( fn ){    // 获取单例
    var result;
       return function(){
          return result || ( result = fn .apply(this, arguments ) );
       }
};

var createLoginLayer = function(){         // 创建登录浮窗
    var div = document.createElement( 'div' );
    div.innerHTML = '我是登录浮窗';
    document.body.appendChild( div );
    return div;
};

var createSingleLoginLayer = getSingle( createLoginLayer );

var loginLayer1 = createSingleLoginLayer();
var loginLayer2 = createSingleLoginLayer();

alert ( loginLayer1 === loginLayer2 );    // 输出： true
```

4. 装饰者模式

使用装饰者模式的时候，我们通常让类或者对象一开始只具有一些基础的

职责，更多的职责在代码运行时被动态装饰到对象上面。装饰者模式可以为对象动态增加职责，从另一个角度来看，这也是分离职责的一种方式。

下面是第15章曾提到的例子，我们把数据上报的功能单独放在一个函数里，然后把这个函数动态装饰到业务函数上面：

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

<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' ) );
};

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

</script>
</html>
```

SRP原则的应用难点就是如何去分离职责，下面的小节我们将开始讨论这点。

18.2　何时应该分离职责

SRP原则是所有原则中最简单也是最难正确运用的原则之一。

要明确的是，并不是所有的职责都应该一一分离。

一方面，如果随着需求的变化，有两个职责总是同时变化，那就不必分离他们。比如在ajax请求的时候，创建xhr 对象和发送xhr 请求几乎总是在一起的，那么创建xhr 对象的职责和发送xhr 请求的职责就没有必要分开。

另一方面，职责的变化轴线仅当它们确定会发生变化时才具有意义，即使两个职责已经被耦合在一起，但它们还没有发生改变的征兆，那么也许没有必要主动分离它们，在代码需要重构的时候再进行分离也不迟。

18.3　违反SRP原则

在人的常规思维中，总是习惯性地把一组相关的行为放到一起，如何正确地分离职责不是一件容易的事情。

我们也许从来没有考虑过如何分离职责，但这并不妨碍我们编写代码完成需求。对于SRP原则，许多专家委婉地表示“This is sometimes hard to see.”。

一方面，我们受设计原则的指导，另一方面，我们未必要在任何时候都一成不变地遵守原则。在实际开发中，因为种种原因违反SRP的情况并不少见。比如jQuery的`attr`等方法，就是明显违反SRP原则的做法。jQuery的`attr`是个非常庞大的方法，既负责赋值，又负责取值，这对于jQuery的维护者来说，会带来一些困难，但对于jQuery的用户来说，却简化了用户的使用。

在方便性与稳定性之间要有一些取舍。具体是选择方便性还是稳定性，并没有标准答案，而是要取决于具体的应用环境。比如如果一个电视机内置了DVD机，当电视机坏了的时候，DVD机也没法正常使用，那么一个DVD发烧友通常不会选择这样的电视机。但如果我们的客厅本来就小得夸张，或者更在意DVD在使用上的方便，那让电视机和DVD机耦合在一起就是更好的选择。

18.4　SRP原则的优缺点

SRP原则的优点是降低了单个类或者对象的复杂度，按照职责把对象分解成更小的粒度，这有助于代码的复用，也有利于进行单元测试。当一个职责需要变更的时候，不会影响到其他的职责。

但SRP原则也有一些缺点，最明显的是会增加编写代码的复杂度。当我们按照职责把对象分解成更小的粒度之后，实际上也增大了这些对象之间相互联系的难度。