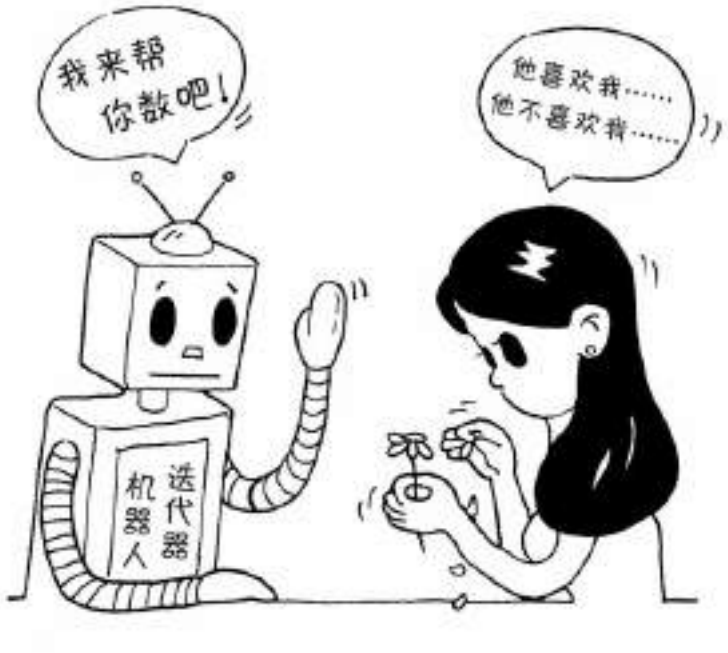
第 7 章　迭代器模式

迭代器模式是指提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。迭代器模式可以把迭代的过程从业务逻辑中分离出来，在使用迭代器模式之后，即使不关心对象的内部构造，也可以按顺序访问其中的每个元素。



目前，恐怕只有在一些“古董级”的语言中才会为实现一个迭代器模式而烦恼，现在流行的大部分语言如Java、Ruby等都已经有了内置的迭代器实现，许多浏览器也支持JavaScript的`Array.prototype.forEach`。

7.1 jQuery中的迭代器

迭代器模式无非就是循环访问聚合对象中的各个元素。比如jQuery中的`$.each`函数，其中回调函数中的参数`i`为当前索引，`n`为当前元素，代码如下：

```
$.each( [1, 2, 3], function( i, n ){
    console.log( '当前下标为：  '+ i  );
    console.log( '当前值为:' + n );
});
```

7.2　实现自己的迭代器

现在我们来自己实现一个each 函数，each 函数接受2个参数，第一个为被循环的数组，第二个为循环中的每一步后将被触发的回调函数：

```
var each = function( ary, callback ){
    for ( var i = 0, l = ary.length; i < l; i++ ){
        callback.call( ary[i], i, ary[ i ] );  // 把下标和元素当作参数传给ca
    }
};

each( [ 1, 2, 3 ], function( i, n ){
    alert ( [ i, n ] );
});
```

7.3 内部迭代器和外部迭代器

迭代器可以分为内部迭代器和外部迭代器，它们有各自的适用场景。这一节我们将分别讨论这两种迭代器。

1. 内部迭代器

我们刚刚编写的each 函数属于内部迭代器，each 函数的内部已经定义好了迭代规则，它完全接手整个迭代过程，外部只需要一次初始调用。

内部迭代器在调用的时候非常方便，外界不用关心迭代器内部的实现，跟迭代器的交互也仅仅是一次初始调用，但这也刚好是内部迭代器的缺点。由于内部迭代器的迭代规则已经被提前规定，上面的each 函数就无法同时迭代2个数组了。

比如现在有个需求，要判断2个数组里元素的值是否完全相等， 如果不改写each 函数本身的代码，我们能够入手的地方似乎只剩下each 的回调函数了，代码如下：

```
var compare = function( ary1, ary2 ){
    if ( ary1.length !== ary2.length ){
        throw new Error ( 'ary1和ary2不相等' );
    }
    each( ary1, function( i, n ){
        if ( n !== ary2[ i ] ){
            throw new Error ( 'ary1和ary2不相等' );
        }
    });
    alert ( 'ary1和ary2相等' );
};

compare( [ 1, 2, 3 ], [ 1, 2, 4 ] );   // throw new Error ( 'ary1和ary2不
```

说实话，这个compare 函数一点都算不上好看，我们目前能够顺利完成需求，还要感谢在JavaScript里可以把函数当作参数传递的特性，但在其他语言中未必就能如此幸运。

在一些没有闭包的语言中，内部迭代器本身的实现也相当复杂。比如C语言中的内部迭代器是用函数指针来实现的，循环处理所需要的数据都要以参数的形式明确地从外面传递进去。

2. 外部迭代器

外部迭代器必须显式地请求迭代下一个元素。

外部迭代器增加了一些调用的复杂度，但相对也增强了迭代器的灵活性，我们可以手工控制迭代的过程或者顺序。

下面这个外部迭代器的实现来自《松本行弘的程序世界》第4章，原例用Ruby写成，这里我们翻译成JavaScript：

```
var Iterator = function( obj ){
    var current = 0;

    var next = function(){
        current += 1;
    };

    var isDone = function(){
        return current >= obj.length;
    };

    var getCurrItem = function(){
        return obj[ current ];
    };

    return {
        next: next,
        isDone: isDone,
        getCurrItem: getCurrItem
    }
};
```

再看看如何改写compare 函数：

```
var compare = function( iterator1, iterator2 ){
    while( !iterator1.isDone() && !iterator2.isDone() ){
```

```
        if ( iterator1.getCurrItem() !== iterator2.getCurrItem() ){
             throw new Error ( 'iterator1和iterator2不相等' );
        }
        iterator1.next();
        iterator2.next();
    }

    alert ( 'iterator1和iterator2相等' );
}

var iterator1 = Iterator( [ 1, 2, 3 ] );
var iterator2 = Iterator( [ 1, 2, 3 ] );

compare( iterator1, iterator2 );  // 输出：iterator1和iterator2相等
```

外部迭代器虽然调用方式相对复杂，但它的适用面更广，也能满足更多变的需求。内部迭代器和外部迭代器在实际生产中没有优劣之分，究竟使用哪个要根据需求场景而定。

7.4 迭代类数组对象和字面量对象

迭代器模式不仅可以迭代数组，还可以迭代一些类数组的对象。比如`arguments`、`{"0":'a',"1":'b'}`等。 通过上面的代码可以观察到，无论是内部迭代器还是外部迭代器，只要被迭代的聚合对象拥有`length`属性而且可以用下标访问，那它就可以被迭代。

在JavaScript中，`for in`语句可以用来迭代普通字面量对象的属性。jQuery中提供了$.each`函数来封装各种迭代行为:

```
$.each = function( obj, callback ) {
    var value,
        i = 0,
        length = obj.length,
        isArray = isArraylike( obj );

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

                if ( value === false ) {
                    break;
                }
            }
        } else {
            for ( i in obj ) {    // 迭代object对象
                value = callback.call( obj[ i ], i, obj[ i ] );
                if ( value === false ) {
                    break;
                }
            }
        }
    return obj;
};
```

7.5 倒序迭代器

由于GoF中对迭代器模式的定义非常松散，所以我们可以有多种多样的迭代器实现。总的来说， 迭代器模式提供了循环访问一个聚合对象中每个元素的方法，但它没有规定我们以顺序、倒序还是中序来循环遍历聚合对象。

下面我们分分钟实现一个倒序访问的迭代器：

```
var reverseEach = function( ary, callback ){
    for ( var l = ary.length - 1; l >= 0; l-- ){
        callback( l, ary[ l ] );
    }
};

reverseEach( [ 0, 1, 2 ], function( i, n ){
    console.log( n );  // 分别输出：2, 1 ,0
});
```

7.6 中止迭代器

迭代器可以像普通for 循环中的break 一样，提供一种跳出循环的方法。在1.4节jQuery的each 函数里有这样一句：

```
if ( value === false ) {
    break;
 }
```

这句代码的意思是，约定如果回调函数的执行结果返回false ，则提前终止循环。下面我们把之前的each 函数改写一下：

```
var each = function( ary, callback ){
    for ( var i = 0, l = ary.length; i < l; i++ ){
        if ( callback( i, ary[ i ] ) === false ){    // callback的执行结果
            break;
        }
    }
};

each( [ 1, 2, 3, 4, 5 ], function( i, n ){
    if ( n > 3 ){          // n大于3的时候终止循环
        return false;
    }
    console.log( n );    // 分别输出：1, 2, 3
});
```

7.7 迭代器模式的应用举例

2013年的一天，当我在重构某个项目中文件上传模块的代码时，发现了下面这段代码，它的目的是根据不同的浏览器获取相应的上传组件对象：

```
var getUploadObj = function(){
    try{
        return new ActiveXObject("TXFTNActiveX.FTNUpload");    // IE上传
    }catch(e){
        if ( supportFlash() ){       // supportFlash函数未提供
            var str = '<object  type="application/x-shockwave-flash"></o
            return $( str ).appendTo( $('body') );
       }else{
            var str = '<input name="file" type="file"/>';  // 表单上传
            return $( str ).appendTo( $('body') );
       }
    }
};
```

在不同的浏览器环境下，选择的上传方式是不一样的。因为使用浏览器的上传控件进行上传速度快，可以暂停和续传，所以我们首先会优先使用控件上传。如果浏览器没有安装上传控件，则使用Flash上传， 如果连Flash也没安装，那就只好使用浏览器原生的表单上传了。

看看上面的代码，为了得到一个upload 对象，这个getUploadObj 函数里面充斥了try ，catch 以及if 条件分支。缺点是显而易见的。第一是很难阅读，第二是严重违反开闭原则。 在开发和调试过程中，我们需要来回切换不同的上传方式，每次改动都相当痛苦。后来我们还增加支持了一些另外的上传方式，比如，HTML5上传，这时候唯一的办法是继续往getUploadObj 函数里增加条件分支。

现在来梳理一下问题，目前一共有3种可能的上传方式，我们不知道目前正在使用的浏览器支持哪几种。就好比我们有一个钥匙串，其中共有3把钥匙，我们想打开一扇门但是不知道该使用哪把钥匙，于是从第一把钥匙开始，迭代钥匙串进行尝试，直到找到了正确的钥匙为止。

同样，我们把每种获取upload 对象的方法都封装在各自的函数里，然后使用一个迭代器，迭代获取这些upload 对象，直到获取到一个可用的为止：

```
var getActiveUploadObj = function(){
    try{
        return new ActiveXObject( "TXFTNActiveX.FTNUpload" );    // IE上
    }catch(e){
        return false;
    }
};

var getFlashUploadObj = function(){
    if ( supportFlash() ){     // supportFlash函数未提供
        var str = '<object type="application/x-shockwave-flash"></object
        return $( str ).appendTo( $('body') );
    }
    return false;
};

var getFormUpladObj = function(){
    var str = '<input name="file" type="file" class="ui-file"/>';  // 表
    return $( str ).appendTo( $('body') );
};
```

在getActiveUploadObj 、getFlashUploadObj 、getFormUpladObj 这3 个函数中都有同一个约定：如果该函数里面的upload 对象是可用的，则让函数返回该对象，反之返回false ，提示迭代器继续往后面进行迭代。

所以我们的迭代器只需进行下面这几步工作。

- 提供一个可以被迭代的方法，使得getActiveUploadObj ，getFlashUploadObj 以及getFlashUploadObj 依照优先级被循环迭代。
- 如果正在被迭代的函数返回一个对象，则表示找到了正确的upload 对象，反之如果该函数返回false ，则让迭代器继续工作。

迭代器代码如下：

```
var iteratorUploadObj = function(){
    for ( var i = 0, fn; fn = arguments[ i++ ]; ){
        var uploadObj = fn();
        if ( uploadObj !== false ){
            return uploadObj;
        }
    }
};

var uploadObj = iteratorUploadObj( getActiveUploadObj, getFlashUploadObj
```

重构代码之后，我们可以看到，获取不同上传对象的方法被隔离在各自的函数里互不干扰，try、catch 和if 分支不再纠缠在一起，使得我们可以很方便地的维护和扩展代码。比如，后来我们又给上传项目增加了Webkit控件上传和HTML5上传，我们要做的仅仅是下面一些工作。

- 增加分别获取Webkit控件上传对象和HTML5上传对象的函数：

```
var getWebkitUploadObj = function(){
    // 具体代码略
};

var getHtml5UploadObj = function(){
    // 具体代码略
};
```

- 依照优先级把它们添加进迭代器：

```
var uploadObj = iteratorUploadObj( getActiveUploadObj, getWebkitUpl
    getFlashUploadObj, getHtml5UploadObj, getFormUpladObj );
```