3 闭包和高阶函数

虽然JS是面向对象语言，但是也有很多函数式语言的特征。设计之初参考了LISP引入了Lamba表达式，闭包，高阶函数等特性

闭包

闭包的形成主要与变量的作用域与生命周期有关

变量的作用域

最长谈到的通常是在函数中的作用域，函数可以用来创造函数作用于，函数外面无法看到函数里面的变量，变量搜索是从内到外的

变量的生存周期

- 全局变量是永久，除非主动销毁
- 局部变量会在退出函数时销毁

看一个例子引出闭包：

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

var f = func();

f(); //2
f(); //3
```

这里虽然退出了函数，但是a一直都在，这是因为f返回了一个匿名函数的引用，这个匿名函数可以访问func()被调用时的环境，而a就在这里，所以有了不能销毁的理由 --->产生了一个闭包结构

闭包的作用

- 封装变量：将全局变量变为私有，看下面i输出值的例子

```
<div>1</div>
<div>2</div>
<div>3</div>

var nodes = document.getElementByTagName('div');
for(var i = 0, len=nodes.length; i< len;i++){
    nodes[i].onclick = function(){
        alert(i)
    }
}
```

点击之后输出的都是3，这是因为div节点是异步触发，而i在循环结束之后从onclick函数内部找到外部时查到的value都是3.

但是换成下面这种写法就不是了：

```
var nodes = document.getElementByTagName('div');
for(var i = 0, len=nodes.length; i< len;i++){
    (function(i){
        nodes[i].onclick = function(){
            alert(i)
        }
    })(i)
}
```

这样i就变成了内部私有变量了，封闭在了onclick中。

当然现在ES5我们可以使用let来实现一样的效果

- 封装函数：同样可以使用闭包来封装一些大函数中小的独立函数
- 延续局部变量的寿命

闭包和面向对象设计

- 对象以方法的形式包含了过程
- 闭包在过程中以环境的形式包含了数据

通常用面向对象思想实现的功能，都可以通过闭包来实现

用闭包实现命令模式

JS的各种设计模式中，闭包的运用很常见，这里我们来看一下两种方法来实现命令模式：

命令模式的意图是将请求封装，并分离请求的发起者和请求的执行者之间的耦合关系，可以在命令被执行前预先植入命令的执行者

- 面向对象的方式：

```
var Tv = {
    // 命令真正的执行者
    open:function(){
        console.log('打开电视')
    },
    close:function(){
        console.log('关闭电视')
    }
}

// 封装的命令
var OpenTvCommand = function(receiver){
    this.receiver = receiver
}

OpenTvCommand.prototype.execute = function(){
    this.recevier.open();
}

OpenTvCommand.prototype.undo = function(){
    this.receiver.close();
}

// 发起命令
var setCommand = function(command){
    document.getElementById('execute').onclick = function(){
        command.execute();
    }
    document.getElementById('undo').onclick = function(){
        command.undo();
    }
}

setCommand(new OpenTvCommand(Tv));
```

- JS中函数作为一等对象，本身就可以传递，所以我们可以使用函数对象而非普通对象来封装请求，这里就需要借助闭包了，命令执行者会被封闭在闭包环境中：

```
var Tv = {
    // 命令真正的执行者
    open:function(){
        console.log('打开电视')
    },
    close:function(){
        console.log('关闭电视')
    }
}

// 封装的命令 - 这里通过闭包保存了receiver，而非普通对象this.receiver来保存
var createCommand = function(receiver){
    var execute = function(){
        return receiver.open();
    }

    var undo = function(){
        return receiver.undo();
    }

    return {
        execute: execute,
        undo: undo
    }
}

// 发起命令
var setCommand = function(command){
    document.getElementById('execute').onclick = function(){
        command.execute();
    }
    document.getElementById('undo').onclick = function(){
        command.undo();
    }
}

setCommand(createCommand(Tv));
```

闭包与内存管理

- 人们有一种误解：闭包会造成内存泄漏，必须减少闭包使用，的确闭包会让一部分我们主动选择的变量活着，但是对于需要的变量其实存放在闭包还是全局作用域中，对内存影响是一致的，不存在内存泄露问题

- 主要问题是闭包有可能造成循环引用，尤其是闭包作用域链中存在DOM节点，就可能内存泄露，但是这其实是基于引用计数策略的垃圾回收机制问题，本质也不是闭包造成的 -- 解决方法只需要将变量设置为null解开变量和其之前引用值的关系

高阶函数

高阶函数是指至少满足下列条件之一的函数：

- 函数可以作为参数传递
- 函数可以作为返回值输出

而JS中的函数显然满足条件，下面我们看看几个应用场景

函数作为参数传递

这个能力方便我们抽离出一部分容易变化的业务逻辑并放在函数参数中 -- 分离业务中变与不变的部分。

最重要的应用场景就是回调函数了，例如把callback函数当作参数传入发起ajax请求方法中，带请求完成之后再执行：

```
var getUserInfo = function(userId, callback){
    $.ajax(....,function(data){
        if(typeof callback === 'function'){
            callback(data)
        }
    })
}
```

还有一些情况，当一个函数不适合执行部分逻辑时，我们可以将其封装成一个函数并将其传递个原函数来调用，这样原函数的灵活性更高，看下面例子：

```
var appendDiv = function(){
    var div = document.createElement('div');
    div.innerHTML = 0;
    document.body.appendChild(div);
    div.style.display = 'none';
}
```

其实：div.style.display = 'none';这一段逻辑写在这里有一点硬编码的感觉，也使得appendDiv函数难以服用，我们可以将这里封装为外部的callback：

```
var appendDiv = function(callback){
    var div = document.createElement('div');
    div.innerHTML = 0;
    document.body.appendChild(div);
    callback && callback(div);
}

appendDiv(function(node){
    node.style.display = 'none';
});
```

函数作为返回值输出

函数当作返回值的场景相对更多，体现函数式编程的巧妙。让函数继续返回一个可执行的函数，意味着运算过程是可延续的。

来看一下一个判断数据类型的函数例子：

```
var isString = function(obj){
    return Object.prototype.toString.call(obj) === '[object String]';
}

var isArray = function(obj){
    return Object.prototype.toString.call(obj) === '[object Array]';
}

...
```

我们虽然可以编写一系列isXXX来判断obj类型，但是我们发现其实这些函数都非常类似，唯一不同的就是判断的方式是【Object xxx】，所以这里我们可以借助提出配置参数的方式来减少冗余代码，并且return所需要的函数：

```
var isType= function(type){
    return function(obj){
        return Object.prototype.toString.call(obj) === `[object ${type}]`;
    }
}

var isString = isType('String');
var isArray = isType('isArray');
```

高阶函数实现AOP

AOP面向切面编程的作用是将跟核心业务逻辑无关的功能抽离出来，例如日志统计，安全控制，异常处理等等。抽离出来之后再通过“动态注入”的方式渗入所需要的业务逻辑模块中。

好处：保持业务逻辑模块的纯净和高内聚同时，方便复用日志统计等模块

Java中一般通过反射和动态代理机制来实现AOP，而JS中则简单的多。

JS中实现AOP都是指把一个函数“动态注入”到另外一个函数之中，实现技术其实有很多，这里展示一个通过Function.prototype来实现的例子：

```
Function.prototype.before = function(beforefn){
    var _self = this; // 保存原函数引用
    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，修正this
        return _self.apply(this, arguments); // 执行原函数
    }
}

Function.prototype.after= function(afterfn){
    var _self = this; // 保存原函数引用
    return function(){ // 返回包含了原函数和新函数的代理函数
        var ret = _self.apply(this, arguments);  // 先执行原函数
        afterfn.apply(this, arguments);
        return ret;
    }
}

var func = function(){console.log(2)}

func = func.before(function(){
    console.log(1)
}).after(function(){
    console.log(3)
})

func(); // 输出 1 2 3
```

上述就是通过函数原型添加before和after的职责 --> JS中装饰者模式的一种实现

其他应用

高阶函数除此之外还有一些应用：

函数节流

某些场景下可能会导致函数被触发的频率很高造成性能问题：

- window.onresize事件
- mousemove事件
- 上传进度
- ....

为了解决这个问题，我们通常会使用throttle函数来优化：原理是将函数先调用一遍，接着用setTimeout延迟一段时间再执行，如果延迟时机还没有到就忽略接下来的函数调用,保证interval期间内函数至多触发一次

```
var throttle = function(fn, interval){
    var _self = fn; // 保存原函数引用
    var timer;
    var firstTime = true;

    return function(){
        var args = arguments;
        var _me = this;

        if(firstTime){
            _self.apply(_me, args);
            return firstTime = false;
        }

        if(timer){
            return false;
        }

        timer = setTimeout(function(){
            clearTimeout(timer);
            timer = null;
            _self.apply(_me, args);
        }, interval || 500)
    }

}

window.onresize = throttle(function(){
    console.log(1)
}, 500)
```

其他应用还有诸如函数柯里化，分时函数，懒加载函数等等，都是通过高阶函数方式给原先方式增加新的能力

分时函数

另外一个问题：用户主动调用某些函数，但是因为客观原因可能会影响页面性能，点击的例子例如大数据渲染列表以及短时间内往页面中添加大量DOM节点，浏览器很容易卡死：

```
var ary = [];

for(var i =1;i<=1000;i++){
    ary.push(i)
}

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

renderFriendList(ary);
```

解决方案之一是实现timeChunk函数，将创建节点工作分批进行，例如1s创建1000个节点改为每隔200ms创建10个节点，这样性能就会好很多

```
// arr:原始数据 fn:针对数据item的函数 count:分批数量
var timeChunk = function(arr, fn, count){
    var obj,t;
    var len = arr.length;

    var start = function(){
        for(var i = 0; i < Math.min(count || 1, arr.length); i++){
            var obj = arr.shift();
            fn(obj);
        }
    };

    return function(){
        t = setInterval(function(){
            if(arr.length === 0){
                return clearInterval(t);
            }
```

```
18              start();
19          }, 200); // 也可以以参数传入来控制
20      }
21  }
```

小结

进入设计模式学习之前，本章主要讲解了闭包以及高阶函数。因为JS设计模式之中这些的应用非常广泛。毕竟因为JS自身语言特点和传统面向对象语言实现还是有差别的，很多都依赖闭包和高阶函数来实现的。

相对于设计模式的实现过程，我们更关注的是模式可以帮助我们完成什么