**浅说泛函与函数变换**

JavaScript是函数第一型(first calss)的语言，JavaScript函数具有两重含义：它既能作为过程调用，又是一个对象。函数本身的可操作性带来了函数变换的设计思想。  
  
**函数变换和泛函**  
  
[有关泛函的概念](http://baike.baidu.com/view/523281.htm)  
  
这里借用数学的概念，简单来说，泛函就是定义域是一个函数，而值域是一个函数，推广开来， 泛函就是从任意的向量空间到标量的映射。  
泛函也是一种“函数”，它的独立变量一般不是通常函数的“自变量”，而是通常函数本身。泛函是函数的函数。  
泛函的英文是 Functional， 所以也可以把函数式编程(Functional Programming)称为泛函编程（对应在函数式编程中也把泛函称为高阶函数(higher-order function) (HOF)的）。  
对JavaScript来说，一个泛函是用来完成一个函数变换的函数。由于变换的结构依然是一个JavaScript函数，所以对于JavaScript来说泛函的值域和定义域是相同的，这意味着泛函变换是可以**迭代**的。  
  
**利用泛函来进行函数变换**  
  
这个思路其实就是通过对函数执行某种泛函过程，把这个函数的输入和输出变为所需要的输入和输出。  
  
**使用泛函的好处**  
  
假如说在一个过程a之后接着要执行一个过程b，并且a和b的执行参数是相同的，用普通的模式来实现如下：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function f(){  
        a.apply(this, arguments);  
        b.apply(this, arguments);  
    }

但是问题来了，如果另一个应用里，过程a之后接着要执行一个过程c，那么：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function f2(){  
        a.apply(this, arguments);  
        c.apply(this, arguments);  
    }

有没有办法把这个过程合二为一呢？答案是有的：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function comb(f1, f2){  
        return function(){  
                f1.apply(this, arguments), f2.apply(this, arguments);  
        }  
    }  
    f = comb(a,b);  
    f2 = comb(a,c);

在这里comb就是一个最基本的“泛函”，它返回将两个函数先后执行的函数，或者也可以看成它对两个函数作了一个“加法变换”： comb: f=f1+f2  
  
**设计实用的函数变换**  
  
函数变换是一种特殊的泛函，它是**以某个函数为中心**的泛函过程。  
通常来说，我们认为函数变换产生新的函数，但是，如果你把JavaScript函数看作一个具有输入和输出的元件的话，那么函数变换产生的结果可以看成是**改变函数的输入或输出**。  
在这里，我们为了简单化设计，把函数变换分成改变输入和改变输出两类，并把这两类变换称为**基本变换**，前面说过函数变换是可以迭代的，所以同时改变输入和输出的函数变换可以通过基本变换组合产生。  
  
接下来，我们主要考虑函数基本变换形式。我通过事件模型来建立基本变换（其实用事件模型不是必须的，只是这么做理解起来稍微简单些）：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function on(func, type, handler){  
    return function(){  
        var evtArgs = {  
            args:[].slice.call(arguments),   
            type:type,  
            owner:this,  
            returnValue:null,   
            target:func,   
            preventDefault: function(){  
                this.target = null  
            }  
        };  
  
        "before" == type && handler.call(evtArgs.owner, evtArgs);  
              
        if(evtArgs.target){  
            evtArgs.returnValue = evtArgs.target.apply(evtArgs.owner, evtArgs.args);  
            "after" == type && handler.call(evtArgs.owner, evtArgs);  
        }  
        return evtArgs.returnValue;  
    }  
}  
on(fn, "before", function(evt){/\*在这里你可以改变输入参数\*/});  
on(fn, "after", function(evt){/\*在这里你可以改变返回值\*/});

OK，现在我们有了第一个泛函，我在这个泛函里给函数本身增加了两个事件，在这两个事件里通过操作事件参数可以改变函数的输入参数和返回值。  
  
现在我要先做一件事情，就是把这个泛函（它本身也是函数）给变换到Function的prototype上去：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function methodize(fn){  
    return on(fn, "before", function(evt){  
        evt.args.unshift(evt.owner);  
    });  
}

所以methodize是我们的第二个泛函。有了它，可以做如下的事情：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

Function.prototype.methodize = methodize(methodize); //先methodize自己^\_^  
Function.prototype.on = on.methodize();

OK，现在泛函更加易于使用了，目前我们用到了on-before，还没有用到on-after，但是我觉得传before、after的参数显得太麻烦，因此我再写第三个泛函：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function curry(fn, curryArgs){  
    return fn.on("before",function(evt){  
            var args = [];  
              
            for(var i = 0, len = curryArgs.length; i < len; i++){  
                if(i in curryArgs){  
                    args.push(curryArgs[i]);  
                }else{  
                    if(evt.args.length){  
                        args.push(evt.args.shift());  
                    }  
                }  
            }  
            evt.args = args.concat(evt.args);  
    });  
}   
Function.prototype.curry = curry.methodize();  
Function.prototype.before = on.curry([,"before"]).methodize();  //注意到这里的curry和methodize其实可以互换，但是需要改变curry的参数  
Function.prototype.after = on.curry([,"after"]).methodize();

curry这个泛函进行了**将默认参数先赋予指定函数**的一个变换。到目前为止我们的变换仅限于改变参数（通过on-before），那么改变返回值有什么用呢？  
  
简单举一个例子，事实上，我们可以尝试着为一般函数建立一个归约(reduce)规则：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function reduce(fn){  
    return fn.after(function(evt){  
        var args = evt.args.slice(evt.target.length);  
        if(args.length > 0){  
            if(evt.returnValue)  
                args.unshift(evt.returnValue)  
            evt.returnValue = reduce(evt.target).apply(evt.owner, args);  
        }  
    });  
}

泛函reduce将函数按参数进行**归约**，例如：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

var add = function(x,y){  
    return x+y;  
}.reduce();  
alert(add(1,2,3,4));  //得到10

事实上，除了reduce之外，我们还可以将函数变换作用于需要**链式**调用的场合，所谓链式调用即形如o.a().b().c().d()...的形式，一般来说要实现链式调用有两种情况，一种是函数自身的返回值进行包装后支持链式调用，一种是改变函数的返回值将函数的第一个参数包装后返回，无论怎样的情况，链式调用都能用基本函数变换来实现，这里就不列举了。  
  
前面我们简单用on-before、on-after实现了多个基本函数变换，事实上，函数变换还有更复杂的形式，现在我们写一个稍稍复杂的函数变换——multi

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function multi(func){  
    return function(){  
        var list = arguments[0];  
        if(list instanceof Array){  
              
            var ret = [];  
            var moreArgs = [].slice.call(arguments,1);  
            for(var i = 0, len = list.length; i < len; i++){  
                var r = func.apply(this, [list[i]].concat(moreArgs));  
                ret.push(r);       
            }  
            return ret;  
        }else{  
            return func.apply(this, arguments);  
        }  
    }  
}  
Function.prototype.multi = multi.methodize();

这个变换将第一个参数的list给函数依次执行，执行结果也作为列表返回。这个变换本身其实不是一个基本变换，它既改变了参数（虽然仅仅是类型发生变化），又改变了返回值，而且实际上这个变换执行的是函数的重复迭代。

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

function g(o){  
    return ++o.x;  
}  
var obj = [{x:1},{x:2},{x:3}];  
obj.g = g.multi().methodize();  
obj.g();  //[2,3,4]

**函数变换的组合使用**  
  
实际上，在前面的介绍中我们了解了基本的函数变换，并且在例子中，其实我们在对这些变换进行组合应用，不过在组合应用时必须注意变换本身对参数造成的影响，这些影响会关系到组合变换的次序。例如你先对一个函数进行multi变换再进行methodize变换和先对一个函数进行methodize变换再进行multi变换，将得到截然不同的结果，原因是methodize变换事实上减少了输入参数的个数。  
  
**批量变换**  
  
我们建立一个map方法，就能很方便地将一组函数批量变换到要使用它的对象（或原型）上。

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

var extend = function(des, src, override){  
    for(var i in src){   
        if(override || !des[i])   
            des[i] = src[i];  
    }  
    return des;  
};  
var map = function(obj, fn, thisObj){  
    var ret = {};  
    for(var key in obj){  
        ret[key] = fn.call(thisObj, obj[key], key, obj);  
    }  
    return ret;  
};

现在我们建立执行批量操作的对象，或者方法集合，比如我们可以把刚才分散的Function.prototype属性赋值集中起来：

[[复制]](http://bbs.51js.com/viewthread.php?tid=85538&highlight=###)

**Code:**

var FunctionUtils = {  
    on:on,  
    reduce:reduce,  
    methodize:methodize,  
    after:after,  
    before:before,  
    curry:curry,  
    multi:multi  
}  
extend(Function.prototype, map(FunctionUtils, methodize));

**新的模式**  
  
事实上到这里我们可以建立一个新的模式，提供批量操作方法，将一组简单的函数经过复杂变换复制到我们需要这些操作的对象或集合上去。这种模式可以允许你先建立轻量的，无污染的模型，然后以某些变换组合将这些模型整理成对象或者创建类，具体细节完全可以交给你实现的函数变换器去处理。