**Thinking in JavaScript Meta-Programming**

**Meta-Programming是一个比较广义的概念，你可以将它翻译成元类或者元模型编程，而它实际的意思，是用一系列方法生成类型模板。在JavaScript，所谓的类型模板就是function，而元类，就是Function和由Function引出的一系列扩展。**  
  
关于JavaScript的Meta-Programming思想，其实有很多实现的例子和各种不同的方法，而在本文中，将它们归结为三个范式。所谓范式，你可以将它们理解为一些“公式化”的概念，或者某种模式，当你遇到同一类问题的时候，你应该寻找适合解决此类问题的“范式”。  
  
**第一范式：<new> T <=> <new> R:function(){donothing, return T.apply},R.prototype = T.prototype**  
  
这个范式被称为函数范式，它有如上面所列的标准形式和其他几种变形。标准形式被称为“前束”范式，因为它能够以语法等价的形式在函数T执行前插入一段代码。例如：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

Function.prototype.$verify = function(){  //对函数进行参数类型匹配  
  
    var me = this;  
    var \_args = arguments;  
    var mins = function(){  
        for(var j= 0, len = \_args.length; j< len; j++)  
        {  
            if(!$oneof(arguments[j],\_args[j])){  
                    throw new Error("函数的参数类型不匹配，位置："+(j+1));  
            }  
        }  
        return me.apply(this, arguments);  
    }  
    mins.prototype = me.prototype;  
    return mins;      
}

上面的代码对函数增加了参数类型匹配的扩展，它可以实现强制解释器对某个函数在调用之前进行参数类型匹配。例如：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

var foo = function(x,y){  
    alert(x+y);  
}.$verify("number","number");  
  
foo(1,2);  
  
var foo2 = function(x,y){  
    x(y);  
}.$verify(Function,"number");  
//foo("error",2);  
  
foo2(function(x){alert(x)},10);  
//foo2("x","y");  
  
var Class3 = function(x,y){  
    this.x = x;  
    this.y = y;  
}.$verify("number","number");  
Class3.prototype.dist2 = function(){return this.x\*this.x + this.y\*this.y};  
var c = new Class3(10,20);  
alert(c.dist2());

注意到上面这一段代码，$verify返回一个function，这个function是调用$verify的那个function的一个第一范式迭代，这种处理方法在语法层面上达到很好的效果，而且它是无害的，这意味着你在编写和调试代码的时候可以得到$verify带来的好处，而在你发布代码的时候，你却可以很容易地用文本处理工具将$verify “尾巴”从你的代码中移出出去。  
  
除此以外，第一范式的应用是很广泛的，因为它意味着你可以对函数进行任意扩展并且这些扩展不改变代码本身的语法结构！  
你可以给函数增加某些不同功能的“尾巴”这些尾巴能够很好地帮你收集运行时信息、监视代码或者提供有用的调试信息。而这些所有的“尾巴”在你最终发布代码时，均可以非常方便地去掉，所以它们对实际运行的代码不会带来任何性能上的开销！这一点，对于开发者来说，无疑是非常非常好的消息！  
  
**第二范式：new T <=> T.apply(T.getPrototypeObject())**  
  
在几个月或者一年以前，我就在思考一个问题，我们知道，对于一个function T，脚本既可以把它当作一个方法来执行，又可以把它作为一个类型来构造，然而它们是不同的。除了new之外，其中语法上最显著的一个区别是T作为一个function，既可以用()来直接操作，也可以享受call和apply带来的好处。而new，则受到比较大的限制，例如：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

function List()  
{  
    this.members = Array.prototype.slice.apply(arguments); //Array支持可变参数，因为它可以作为函数来调用  
}  
function $list()  
{  
    return new List(/\*这里的参数应该怎么传？如果我希望$list(1,2,3...) <=> new List(1,2,3...)\*/);   
}

我们看到，相对function来讲，new操作受到较大的限制，当然上面这个实际问题是可以通过别的方式来解决的，但是用第二范式，无疑可以具有通用性地解决此类问题：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

G.objectAsPrototype = function(obj, c){  
    c = c || function(){};  
    c.prototype = obj;  
    return c;  
};  
Function.prototype.getPrototypeObject = function(){  
    var p = this.\_\_templete\_\_ || (this.\_\_templete\_\_= G.objectAsPrototype(this.prototype));  
    return new p();  
};  
Function.prototype.createInstance = function(){  
    var p = this.getPrototypeObject();  
    this.apply(p,arguments);  
    return p;  
};

我们看到，第二范式解决了这样的问题，它告诉我们，JavaScript的new T操作等价于通过T的prototype创造一个“原型”对象，再用这个对象去执行T的构造函数，最终等价于产生了一个T的实例，但区别是后者不同于new，是以一种函数调用的标准形式来产生的。  
  
利用第二范式，我们可以很容易地增强JavaScript的原型继承，轻易地解决原型继承中关于构造函数延迟执行的需求（具体的将在另外一篇文章《深度探索高效率JavaScript继承》给出详细说明）  
下面给出简单代码：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

Function.prototype.$pextends = function(p){  
    var me = this;  
    var ins = function()  
    {  
        this.$super = function(){  
            p.apply(this, arguments);  
        }  
        me.apply(this, arguments);  
    }  
  
    ins.prototype = p.getPrototypeObject();  
    return ins;  
}

**第三范式：new T <=> T.apply || new T(T.apply)**  
  
记得之前有人问过我一个有意思的问题，是关于js核心对象的扩展的。那个需求是实现一个自定义的Date类型MyDate，并且这个MyDate的所有构造参数都要和Date完全一致。之前，这个问题在解决的时候遇到一个困扰，具体的是这样的：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

function MyDate()  
{  
    var ins = new Date(/\*原生对象的扩展方式，可是如何处理可变参数呢？\*/);  
    ......  
    return ins;  
}

而第三范式的意思是说，对于所有的核心对象，都满足如下两种情况之一：要么函数调用的返回结果等效于new操作，要么函数调用的返回结果利用new构造后等同于原始的new操作。Date的问题满足后者，也就是说，对于这个问题，将上面的代码改写成如下：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

function MyDate()  
{  
    var ins = new Date(Date.apply(this, arguments));  
    ......  
    return ins;  
}

即可满足需求。  
显然，这个第三范式并不是一个JavaScript类型默认遵循的范式，不过，有趣的是，几乎所有的核心对象都遵循第三范式：  
  
Array、Function 满足范式右侧的第一个条件  
Number、Boolean、String、Date、RegExp 满足范式右侧的第二个条件  
  
所以，利用第三范式，我们可以实现核心对象的继承方法（关于核心对象继承的详细内容也会在《深度探索高效率JavaScript继承》给出详细讨论）：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

Function.prototype.$cextends = function(p){  
    var me = this;  
    return function()  
    {  
        var ins = p.apply(this, arguments);  
        ins instanceof p || (ins = new p(ins));  
  
        me.apply(ins,arguments);  
        return ins;  
    }  
}

除了继承核心对象之外，第三范式还有其他很有趣的应用：

[[复制]](http://bbs.51js.com/viewthread.php?tid=77451&highlight=###)

**Code:**

var MyFunction = function(){  
    this.m = function(){alert("static m")}  
    this.prototype.m = function(){alert("m")};  
}.$cextends(Function);  
  
//简单实现的Function Template  
var X = new MyFunction("alert(1)");  
var x = new X();  
X.m();  
x.m();

上面这段代码实现了一个自定义的函数模版，这样使用者就能够很方便地自己扩展Function，这个模式的意义是让Function元类具备有扩展能力，这种能力正是Meta-Programming所需要的。  
  
**总结**  
  
在上面的讨论中，我们给出了三个范式的基本形式，并且分别进行了基本的说明，在实际应用中，通过它们可以实现一些非常Cool的扩展，有兴趣的读者可以研究下面的代码：  
  
   提示：您可以先修改部分代码再运行  
  
今天所讨论的内容只是Meta-Programming的“冰山一角”，算是抛砖引玉吧，希望大家在这方面能够有好的研究和好的结果，也希望大家能够多多进行交流和分享，因为Meta-Programming对于脚本的意义重大，它是一种着眼未来的编程模式。