**一个简单的轻量级JavaScript面向对象系统扩展**

[[复制]](http://bbs.51js.com/viewthread.php?tid=78071&highlight=###)

**Code:**

/\*\*   
 \* @fileoverview Object的OOS面向对象系统扩展  
 \* @author {@link mailto:akira.cn@gmail.com Akira Wu}  
 \* @version 1.0   
 \* @create-date : 2008-5-9  
 \* @last-modified : 2008-5-9  
 \*/  
  
/\*\*$import 在这里加载需要加载的包（路径）  
    modules.x\_function.function  
$\*/  
  
/\*\*  
 \* 根据一个对象创建它的原型类型  
 {%example  
<script>  
    var Point = function(x, y){  
        this.x = x;  
        this.y = y;  
    }  
    Point.prototype.x = 0;  
    Point.prototype.y = 0;  
  
    var p = new Point(10,20);  
    alert(p.x);  
    var p2 = new (Object.asPrototype(Point.prototype));  
    alert(p2.x);  
</script>  
 %}  
 \*/  
Object.asPrototype = function(obj){  
    var ret = function(){};  
    ret.prototype = obj;  
    return ret;  
};  
  
/\*\*  
 \* 根据现有的object建立一个"平面的"类型(非原型而是直接抄写对象的属性)  
 \* @param {Object} obj 要创建类型的对象  
 \* @returns {Function} 返回创建的类型  
 \*/  
Object.createFlatType = function(obj){  
    return function(){  
        for(var each in obj){  
            this[each] = obj[each];  
        }  
    }  
};  
      
/\*\*  
 \* 判断一个对象是否是JavaScript核心对象  
 \* @param {Object} obj 指定对象  
 \* @returns {boolean} 当该对象是JavaScript的核心对象时，返回true，否则返回false  
 \*/  
Object.definedInCore = function(obj){  
    while(obj.$superClass){  
        if(obj == obj.$superClass) break;  
        obj = obj.$superClass;  
    }  
    return obj === Array ||   
        obj === Boolean ||   
        obj === Date ||   
        obj === Error ||  
        obj === EvalError ||  
        obj === Function ||  
        obj === Math ||  
        obj === Number ||  
        obj === Object ||  
        obj === RangeError ||   
        obj === RegExp ||  
        obj === String ||      
        obj === SyntaxError ||  
        obj === TypeError ||  
        obj === URIError;   
};  
  
/\*\*  
 \* 判断一个object是否是某个接口的实现  
 \* @param {Object} obj 被检测的对象  
 \* @param {Function} inter 被检测的类型  
 \* @returns {boolean} 如果被检测的对象是被检测的类型或者其基类的原型，返回true，否则返回false  
 \*/  
Object.implementationOf = function(obj, inter){  
    var \_interfaces = obj && Object.getClass(obj).\_\_interfaces\_\_;  
  
    if(\_interfaces)  
        for(var i = 0, len = \_interfaces.length; i < len; i++){  
            if(\_interfaces[i] == inter)  
                return true;  
        }  
    return false;  
};  
  
/\*\*  
 \* 判断一个对象是否是一个类型/接口的实例/实现  
 {%example  
<script>  
    var a = 10;  
    alert(Object.instanceOf(a,"number"));  
    var b = new Object();  
    alert(Object.instanceOf(b, Object));  
</script>  
 %}  
 \* @param {Object} obj 要判断类型的对象  
 \* @param {string|Function|Interface} c 类型/接口  
 \* @returns {boolean} 当c是obj的类型或者obj实现的接口时，返回true，否则返回false  
 \*/  
Object.instanceOf = function(obj, c){  
    if(c == null) return false;  
    if(typeof(c) == "string")  
        return typeof(obj) == c;  
    if(c && c instanceof Function && obj instanceof c)  
        return true;  
    if(c && obj && obj.constructor == c)  
        return true;  
    if(c && obj && c.constructor && obj.constructor && Object.prototypeOf(c.prototype, obj.constructor.prototype.constructor))  
        return true;  
  
    return Object.implementationOf(obj, c);  
};  
  
/\*\*  
 \* 判断一个object是否在某个类型的原型链上  
 {%example  
<script>  
    var Test = function(){  
    };  
    var p = {};  
    Test.prototype = p;  
    var t = new Test();  
    alert(Object.prototypeOf(p, Test));  
</script>  
 %}   
 \* @param {Object} obj 被检测的对象  
 \* @param {Function} c 被检测的类型  
 \* @returns {boolean} 如果被检测的对象是被检测的类型或者其基类的原型，返回true，否则返回false  
 \*/  
Object.prototypeOf = function(obj, c){  
    if(!obj || !c) return false;  
    var \_c = c;  
    while(true){  
        if(!\_c.prototype) return false;  
        if(obj == \_c.prototype){   
            return true;  
        }  
        if(!\_c.prototype.constructor || \_c == \_c.prototype.constructor)  
            return false;  
        \_c = \_c.prototype.constructor;  
    }  
};  
  
/\*\*  
 \* 得到指定对象的类型  
 \* @param {Object} obj 要得到类型的对象  
 \* @returns {Function} 对象的类型  
 \*/  
Object.getClass = function(obj){  
    if(obj.$class instanceof Function) return obj.$class;  
    return obj.constructor;  
};  
  
/\*\*  
 \* {META}改进的实例继承，用来扩展核心对象 <br/>  
 \*   
 \* 第三范式：new T <=> T.apply || new T(T.apply) <br/>  
 \* 所有的JS核心对象都满足这个范式： T.apply(this, arguments) || new T(T.apply(this, arguments))<br/>  
 \* new T(T.apply(a,b,c)) <=> T.apply(a,b,c) || new T(a,b,c) <br/>  
 {%example  
<script>  
    var MyDate = function(){  
        this.cname = "MyDate";  
    }.$cextends(Date);  
      
    var d = new MyDate();  
    alert(d.cname);  
    alert(d.toGMTString());  
</script>  
 %}  
 \* @param {Function} p 当前类型的父类  
 \* @returns {Function} 返回扩展了对象p的自身类型  
 \*/  
Function.prototype.$cextends = function(p){  
    if(!Object.definedInCore(p))  
        throw new Error("不能对非核心对象使用$cextends继承，请使用$pextends或$extends");  
      
    var me = this;  
  
    var ret = function()  
    {  
        var ins = p.apply(this, arguments);  
        ins instanceof p || (ins = new p(ins));  
        for(var each in me.prototype){  
            if(ins[each] == null)  
                ins[each] = me.prototype[each];  
        }  
        me.apply(ins,arguments);  
        ins.constructor = p;  
        ins.$class = ret;  
        return ins;  
    }  
  
    ret.$superClass = p;  
  
    ret.$cextends = ret.$pextends = Function.Error(TypeError, "不能对已经继承过的对象再次进行继承!");  
      
    return ret;  
};  
  
/\*\*  
 \* 如果是核心对象，调用$cextends，否则调用$pextends  
 \* @see Function#$cextends  
 \* @see Function#$pextends  
 \*/  
Function.prototype.$extends = function(p){  
    if(Object.definedInCore(p) && p !== Object)  
        return this.$cextends(p);  
    return this.$pextends(p);  
};  
  
/\*\*  
 \* {META} 实现接口，可支持一组参数<br/>  
 \* 实现了接口的方法在构造对象的时候会对接口强制进行实现检查<br/>  
 \* 如果未实现，会抛出Error<br/>  
 \* 接口可以是任意对象，如果它的某个field值为空，则在构造对象时对该域进行强制检查<br/>  
 \* 否则将接口的值作为对象域的默认值  
 \* @param more... 要实现的接口  
 \* @returns 返回声明要实现接口的自身类型  
 \* @type Function  
 \* @throws Error 当试图对一个核心对象进行接口声明时抛出Error  
 \*/  
Function.prototype.$implements = function(){  
    if(Object.definedInCore(this))  
        throw Error("不能为核心对象声明用户接口!");  
    var me = this;  
    var \_args = Array.prototype.slice.call(arguments);  
    var mins = function(){  
        var ret = me.apply(this, arguments);  
        for(var i = 0, len = \_args.length; i < len; i++)  
        {  
            var p = \_args[i];  
            for(var each in p)  
            {  
                if(this[each] == null && p[each] == null){  
                    throw new Error("接口方法 "+each+" 未实现");  
                }  
                else if(this[each] == null){  
                    this[each] = p[each];  
                }  
            }  
        }  
        this.$class = mins;  
        return ret;  
    }  
      
    mins.$cextends = mins.$pextends = Function.Error("继承不应当出现在接口声明之后!");  
      
    if(me.$superClass){  
        mins.\_\_interfaces\_\_ = me.$superClass.\_\_interfaces\_\_;  
        mins.$superClass = me.$superClass;  
    }  
    mins.\_\_interfaces\_\_ = mins.\_\_interfaces\_\_ ? mins.\_\_interfaces\_\_.concat(\_args) : \_args;  
    mins.prototype = me.prototype;  
      
    return mins;  
};  
  
/\*\*  
 \* {META}改进的对象原型继承，延迟执行参数构造，并在子类的实例中添加了$super引用  
 {%example  
<script>  
    function Shape(x, y){  
        this.className = "Shape";  
        this.x = x || 0;  
        this.y = y || 0;  
    }  
    Shape.prototype.getClassName = function(){  
        return this.className;  
    }  
    var Square = function(x, y, w, h){  
        this.$super(x, y);  
        this.className = "Square";  
        this.w = w || 1;  
        this.h = h || 1;  
    }.$pextends(Shape);  
    Square.prototype.leftTop = function(){  
        return [this.x, this.y];  
    }  
    Square.prototype.rightBottom = function(){  
        return [this.x + this.w, this.y + this.h];  
    }  
    var s = new Square(3, 5, 2, 4);  
    alert(s.getClassName());  
    alert(s.rightBottom());  
      
    var ColorSquare = function(c,x,y,w,h){  
        this.$super(x, y);  
        this.c = c || "red";  
        this.className = "CS";  
    }.$pextends(Square);  
    ColorSquare.prototype.color = function(){return this.c}  
    var c = new ColorSquare();  
    alert(c.color());  
    alert(c.leftTop());  
</script>  
 %}  
 \* @param {Function} p 当前类型的父类  
 \* @returns 返回继承了p的自身类型  
 \* @type Function  
 \* @throws Error 当试图对一个系统核心对象使用原型继承时抛出Error  
 \*/  
Function.prototype.$pextends = function(p){  
    if(Object.definedInCore(p))  
        throw new Error("不能对系统核心对象使用原型继承!");  
      
    var me = this;  
    var ins = function()  
    {     
        var \_me = this;  
        this.$super = function(){return p.apply(\_me, arguments)};  
        var ret = me.apply(this, arguments);  
        this.constructor = me;  
        this.$class = ins;  
        this.$super = null;  
        return ret;  
    }  
  
    ins.$superClass = p;  
      
    ins.\_\_interfaces\_\_ = me.\_\_interfaces\_\_;  
    ins.prototype = new (Object.asPrototype(p.prototype));  
  
    ins.$cextends = ins.$pextends = Function.Error(TypeError, "不能对已经继承过的对象再次进行继承!");  
      
    return ins;  
};  
  
/\*\*  
 \* 为类型动态创建一个实例，它和直接new的区别在于instanceof的值<br/>  
 \*  
 \* {第二范式：new T <=> T.apply(T.getPrototypeObject())}  
 \* @param {void}  
 \* @returns 这个类型的一个实例  
 \*/  
Function.prototype.createInstance = function(){  
    var p = new (Object.asPrototype(this.prototype));  
    this.apply(p,arguments);  
    return p;  
};  
  
/\*\*  
 \* function.VOID.$extends的缩写<br/>  
 \* 返回一个继承p的空函数  
 \* @param {Function} p 被继承的类型  
 \* @returns {Function} 继承了指定类型的函数  
 \* @see Function#$extends  
 \*/  
Function.$extends = function(p){  
    return Function.VOID.$extends(p);  
};  
  
/\*\*  
 \* function.VOID.$implements的缩写  
 \* 返回一个实现p接口的空函数  
 \* @param {Object} p 被实现的接口  
 \* @returns {Function} 实现了指定接口的函数  
 \* @see Function#$implements  
 \*/  
Function.$implements = function(p){  
    return Function.VOID.$implements(p);  
};  
  
/\*\*  
 \* 返回一个执行后抛出异常的函数  
 \* @param {Error} err 要抛出的异常类型，默认为Error，可缺省  
 \* @param {string} msg 异常的描述信息  
 \* @returns {Function} 一个执行后抛出异常的函数  
 \*/  
Function.Error = function(err, msg){  
    if(arguments.length < 2){  
        msg = arguments[0];  
        err = Error;  
    }  
    return function(){  
        throw new err(msg);  
    }  
};  
  
/\*\*  
 \* 返回第一个参数的函数  
 \* @param {Object} obj 要返回的对象  
 \* @returns {Object} 要返回的对象  
 \*/  
Function.K = function(obj){return obj};  
  
/\*\*  
 \* 空函数  
 \* @type void  
 \*/  
Function.VOID = function(){};