Javascript 面向对象特性实现（封装、继承、接口）

Javascript 作为弱类型语言，和 Java、php 等服务端脚本语言相比，拥有极强的灵活性。对于小型的 web 需求，在编写 javascript 时，可以选择面向过程的方式编程，显得高效；但在实际工作中，遇到的项目需求和框架较大的情况下，选择面向对象的方式编程显得尤其重要，Javascript 原生语法中没有提供表述面向对象语言特性的关键字和语法（如 extends、implement）。为了实现这些面向对象的特性，需要额外编写一些代码，如下。

在开始使用 OO 特性之前，还需要考虑使用接口、继承所带来的弊端。封装、接口、继承都将使代码结构变得复杂，对于编码新手有较高的要求，对于别人接受你的项目成本也提高了，在团队协作中需要根据具体需求斟酌，不要为了秀技术而写代码；同时，封装和接口都将带来一些额外的内存开销，有些可以忽略不计，有些 则是得不偿失，需要注意。

一、封装

```
var book = (function(){
        var COUNT = 0;

        //静态私有方法
        function checkISBN(isbn){
            return true;
        }
        //构造器
        var ctor = function(id,name,isbn){
            var _id,_name,_isbn
            this.setId= function(id){
                _id=id;
            }
            this.getId = function(){
                return _id;
            }
            this.setName = function(name){
                _name = name;
            }
            this.getName = function(){
                return name;
            }
            this.setIsbn = function(isbn){
                _isbn = isbn;
            }
            this.getIsbn = function(){
                return isbn;
            }
            if(checkISBN){
```

```
                COUNT++;
            }
            this.setName(name);
            this.setId(id);
            this.setIsbn(isbn);
        }
        ctor.getCount = function(){
            return COUNT;
    }
        return ctor;
    })();
//静态、共用方法
book.buyCount = function(count){
    return count;
}

book.prototype = {
    display:function(){
    }
}
var b = new book();
b.getCount();
b.display();
```

二、接口

```
/**
 * 接口：实现多个类的共同性。让彼此不想关的对象也能被同等对待。
 */

var Interface = function(name,method){
    if(arguments.length < 2){
        throw new Error("xxx")；
    }
    this.name = name;
    this.method = [];
    for(var i = 0;i<method.length;i++){
        if(typeof method[i] !== 'string'){
            throw new Error("xxx");
        }
        this.method.push(method[i]);
    }
}
//public static method
```

```
Interface.ensureImplement = function(object){
    for(var i = 0;i<arguments.length;i++){
        var interface = arguments[i];
        if(Interface.construction !== Interface){
            throw new Error("xxx";)
        }
        for(var j = 0;j<ingerface.method.length;j++){
            var method = interface.method[j];
            if(!object[method] || typeof object[method] !==='function'){
                throw new Error("xxx");
            }
        }
    }
}

var testResultInstance = new Interface("testResultInstance",["getData","getResults"]);
function testInterface = function(mapInstance){
    Interface.ensureImplement(mapInstance,testResultInstance);
    mapInstance.getData();
    mapInstance.getResults();
}
function Map(name){
    this.name = name;
    this.getData = function(){};
    this.getResults = function(){};
}
var mapInstance = new Map("test");
testInterface(mapInstance);
```

三、继承

```
/**
 * 继承提供一些代码复用的功能。但继承照成两个类间的强耦合
 */
//类式继承
function Person(name){
    this.name = name;

}
function Design(name,book){
    Person.call(this,name);
    this.book = book;
}
extend(Person,Desion);
```

```
Design.prototype.getBooks = function(){
	return this.book;
}
var d = new Design("tim","test");
d.getBooks();
d.name;

function extend(superclass,subclass){
	var F = function(){};
	F.prototype = superclass.prototype;
	subclass.prototype = new F();
	subclass.prototype.constructor = subclass

	subclass.superclass = superclass;
	if(superclass.prototype.constructor == Object.prototype.constructor){
		superclass.prototype.constructor = superclass;
	}
}
/*************************************************************************/
//原型继承
function clone(superclass){
	var F = function(){};
	F.prototype = superclass;
	return new F();
}
var Person = {
	name:"default",
	getName : function(){
		return this.name;
	}
};
var Desion = clone(Person);
Desion.books = ["写给大家看的设计书"];
Desion.getBooks = function(){
	return this.books;
}
/*************************************************************************/
//参元法
var Mimin = function(){};
Mimin.prototype = {
	name : 'default',
	getName : function(){
		return this.name;
	}
```

```
};
function augment(receiveingClass,givingClass){
        for(methodName in givingClass){
                if(!receiveingClass[methodName]){
                        receiveingClass[methodName] = methodName;
                }
        }
}
```