总结javascript设计模式使用方法

第一部分、面向对象的基础知识

1. 、面向对象的 JavaScript

1.1 动态类型语言和鸭子类型

定义：如果它走起路来像鸭子，叫起来也像鸭子，那么它就是鸭子。

鸭子类型指导我们只关注对象的行为，而不关注对象本身，也就是关注 HAS-A, 而不是 IS-A。

例如：

if( animal && animal.duckSinging === ‘function’ ){ //典型的鸭子类型检测

choir.push( animal ) ;

}

1.2 多态

定义：多一操作作用于不同对象，产生不同的解释和不同的执行结果。

给不同的对象发送同一个消息的时候，这些对象会根据这个消息分别给出不同的

反馈。

实际作用：多态背后的思想是将“做什么”和“谁去做以及怎样去做”分离开来，也就是将“不变的事物”与 “可能改变的事物”分离开来。

例如：当主人向它们发出“叫”的命令时，鸭会“嘎嘎嘎”地叫，而鸡会“咯咯咯”地叫。

其中动物都会叫，这是不变的；但是不同类型的动物具体怎么叫是可变的。

//不变的部分

var makeSound = function( animal ){

Animal.sound();

};

//可变的部分

var Duck = function(){};

Duck.prototype.sound = function(){

Console.log( ‘嘎嘎嘎嘎’ );

};

Var Chiken = function(){};

Chiken.prototype.sound = function(){

Console.log( ‘咯咯咯咯’ );

};

1.2.3 类型检查和多态 java中向上转型来实现多态。

1.2.4 使用继承得到多态效果

1.2.5 JavaScript的多态

一个 JavaScript对象，既可以表示 Duck 类型的对象，又可以表示 Chicken 类型的对象，这意味着 JavaScript对象的多态性是与生俱来的。

作用：换句话说，多态最根本的作用就是通过把过程化的条件分支语句转化为对象的多态性，从而消除这些条件分支语句。

1.3 封装

封装的目的是将信息隐藏。一般而言，我们讨论的封装是封装数据和封装实现。

1.3.1 封装数据

例如：

var myObject = (function(){

var \_name = 'fan'; //私有（private）变量--局部变量

return { //更好的方案 ECMAScript 6中提供的 let

getName:function(){

return \_name; //公开（public）方法--访问局部变量的公共接口

}

}

})(); //自执行函数

console.log( myObject.getName() ); //'fan'

console.log( myObject.\_name ); //undefined

封装不仅仅是隐藏数据，还包括隐藏实现细节、设计细节以及隐藏对象的类型等。

封装使得对象之间的耦合变松散，对象之间只通过暴露的 API接口来通信。当我们修改一个对象时，可以随意地修改它的内部实现，只要对外的接口没有变化，就不会影响到程序的其他功能。

1.3.3 封装类型

在封装类型方面，JavaScript没有能力，也没有必要做得更多。对于 JavaScript的设计模式实现来说，不区分类型是一种失色，也可以说是一种解脱。

1.3.4 封装变化

通过封装变化的方式，把系统中稳定不变的部分和容易变化的部分隔离开来。

1.4 原型模式和基于原型继承的 JavaScript 对象系统

而在原型编程的思想中，类并不是必需的，对象未必需要从类中创建而来，一个对象是通过克隆另外一个对象所得到的。

1.4.1 使用克隆的原型模式

作用：既然原型模式是通过克隆来创建对象的，那么很自然地会想到，如果需要一个跟某个对象一模一样的对象，就可以使用原型模式。

例子：某种飞机拥有分身技能，当它使用分身技能的时候，要在页面中创建一些跟它一模一样的飞机。

原型模式的实现关键，是语言本身是否提供了 clone 方法。ECMAScript 5提供了 Object.create方法，可以用来克隆对象。

var plane1 = Object.create( plane );

1.4.2 克隆是创建对象的手段

1.4.4 原型编程范型的一些规则

原型编程范型至少包括以下基本规则。

1.所有的数据都是对象。

2.要得到一个对象，不是通过实例化类，而是找到一个对象作为原型并克隆它。

3.对象会记住它的原型。

4.如果对象无法响应某个请求，它会把这个请求委托给它自己的原型。

1.4.5 JavaScript中的原型继承

JavaScript 中的根对象是 Object.prototype 对象。 Object.prototype 对象是一个空的对象。

利用 ECMAScript 5提供的 Object.getPrototypeOf 来查看这个对象的原型：

例如：

console.log( Object.getPrototypeOf( obj1 ) === Object.prototype ); // 输出：true

JavaScript 给对象提供了一个名为 \_\_proto\_\_ 的隐藏属性，某个对象的 \_\_proto\_\_ 属性默认记录着会指向它的构造器的原型对象，即 {Constructor}.prototype 。

例如：

var a = new Object();

console.log ( a.\_\_proto\_\_=== Object.prototype ); // 输出：true

最后还要留意一点，原型链并不是无限长的，但 Object.prototype 的原型是 null，此时就结束了。

而通过 Object.create( null ) 可以创建出没有原型的对象。

ECMAScript 6带来了新的 Class语法。

例如：

class Animal{

constructor( name ){

this.name = name;

}

getName(){

return this.name;

}

}

class Dog extends Animal{

constructor( name ){

super( name );

}

sound(){

return '嘎嘎';

}

}

var dog = new Dog( '小黄' );

console.log( dog.getName() + dog.sound() ); //小黄嘎嘎

1. 、this 、 call 和 apply

2.1 this

1.作为对象的方法调用。 //var obj = { this === obj };

2.作为普通函数调用。 //var name = function(){ this === window || global };

但在ECMAScript 5的 strict 模式下指向undefined

3.构造器调用。 --new 运算符调用， 一般this === 该对象

4.Function.prototype.call 或 Function.prototype.apply 调用。可以动态地

改变传入函数的 this。

2.2 call 和 apply

如果我们传入的第一个参数为 null ，函数体内的 this 会指向默认的宿主对象，在浏览器中则是 window，但在”use strict”下this还是指向null；

2.2.2 call 和 apply 的用途

1. 改变 this 指向-- getName.call( obj1 )，将getName的this指向obj1。

2. Function.prototype.bind

例如：

var obj1 = {

name:'fan'

};

var func = function(){

console.log( this.name );

}.bind( obj1 );

func(); //’fan’

1. 借用其他对象的方法

借用方法的第一种场景是“借用构造函数”，通过这种技术

var A = function( name ){

this.name = name;

};

var B = function(){

A.apply( this,arguments ); //借用a的this.name属性

};

B.prototype.getName = function(){

return this.name;

};

var b = new B( 'fan' );

console.log( b.getName() );

Array.prototype.push ：

(function(){

Array.prototype.push.call( arguments, 3 );

console.log ( arguments ); // 输出[1,2,3]

})( 1, 2 );

1. 、闭包和高阶函数

3.1 闭包

闭包的形成与变量的作用域以及变量的生存周期密切相关。

3.1.1 变量的作用域

3.1.2 变量的生存周期

var fuc = function(){

var a = 1;

return function(){

a++;

console.log( a );

};

};

var f = fuc();

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

3.1.3 闭包的更多作用

1. 封装变量

var mult = (function(){

var cache = {}; //用于缓存之前计算的结果

var calculate = function(){ //单独封装成函数----进行计算部分

var a = 1;

for(var i = 0, len = arguments.length; i < len ; i++){

a \*= arguments[ i ];

}

return a;

};

return function(){ //闭包，事cache一直保存在内存中

var arg = Array.prototype.join.call( arguments,',' );

if( arg in cache ){

return cache[ arg ];

}

return cache[ arg ] = calculate.apply( null,arguments );

};

})();

console.log( mult( 1,2,3 ) );

1. 延续局部变量的寿命

img 对象经常用于进行数据上报

Var report = (function(){

Var imgs = [];

Return function( src ){

Var img = new Image();

Imgs.push( img );

Img.src = src;

};

})();

3.1.4 闭包和面向对象设计

//方式1

var extent = {

value: 0,

call: function(){

T his.value++;

console.log( this.value );

}

};

extent.call(); // 输出：1

extent.call(); // 输出：2

//方式2

var Evtent = function( val ){

this.val = val;

};

Evtent.prototype.ff = function(){

this.val++;

console.log( this.val )

};

var evtent = new Evtent( 12 );

evtent.ff(); //13

evtent.ff(); //14

3.2 高阶函数

高阶函数是指至少满足下列条件之一的函数。

1. 函数可以作为参数被传递。
2. 回调函数--ajax等等

var appendDiv = function( callback ){

for ( var i = 0; i < 100; i++ ){

var div = document.createElement( 'div' );

div.innerHTML = i;

document.body.appendChild( div );

if ( typeof callback === 'function' ){

callback( div );

}

}

};

appendDiv(function( node ){

node.style.display = 'none';

});

可以看到，隐藏节点的请求实际上是由客户发起的，但是客户并不知道节点什么时候会创建好，于是把隐藏节点的逻辑放在回调函数中，“委托”给 appendDiv 方法。 appendDiv 方法当然知道节点什么时候创建好，所以在节点创建好的时候， appendDiv 会执行之前客户传入的回调函数。

1. Array.prototype.sort

[ 1, 4, 3 ].sort( function( a, b ){ //大到小，如果return a - b则小到大

return b - a;

});

1. 函数可以作为返回值输出。

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。

1. 判断数据的类型

var isType = function( type ){

return function( obj ){

return Object.prototype.toString.call( obj ) === '[object ' + type +']';

};

};

var isString = isType( 'String' ); //第一个参数对应type

var dd = isString( 'str' ); //第二个参数对应obj

1. getSingle

3.2.3 高阶函数实现AOP

AOP（面向切面编程）的主要作用是把一些跟核心业务逻辑模块无关的功能抽离出来，这些跟业务逻辑无关的功能通常包括日志统计、安全控制、异常处理等。把这些功能抽离出来之后，再通过“动态织入”的方式掺入业务逻辑模块中。

Function.prototype.before = function( beforefn ){

var that = this; // 保存原函数的引用

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 that.apply( this, arguments ); // 执行原函数

}

};

Function.prototype.after = function( afterfn ){

var that = this; // 保存原函数的引用

return function(){ // 返回包含了原函数和新函数的"代理"函数

var ret = that.apply( this, arguments ); // 执行原函数

afterfn.apply( this, arguments ); // 执行新函数，修正 this

return ret;

}

};

//调用方法

var func = function(){

console.log( 1 );

};

func = func.before( function(){

console.log( 2 );

} ).after( function(){

console.log( 3 );

} );

func();

3.2.4 高阶函数的其他应用

1. currying----函数柯里化（function currying）

假设我们要编写一个计算每月开销的函数。在每天结束之前，我们都要记录今天花掉了多少

钱。通用的 function currying(){}

var currying = function( fn ){

var args = [];

return function(){

if( arguments.length === 0 ){

return fn.apply( this,args );

}else{

[].push.apply( args,arguments );

return arguments.callee;

}

};

};

var cost = (function(){

var money = 0;

return function(){

for( var i =0; i<arguments.length; i++ ){

money += arguments[ i ];

}

return money;

};

})();

var dd = currying( cost );

dd(100); // 未真正求值

dd(200); // 未真正求值

console.log(dd()); //求出真值

1. Uncurrying

//方式一

Function.prototype.uncurrying = function(){

var self = this;

return function(){

var obj = [].shift.call( arguments );

return self.apply( obj,arguments );

};

};

//方式二

Function.prototype.uncurrying = function(){

var self = this;

return function(){

return Function.prototype.call.apply( self, arguments );

}

};

1. 函数节流

函数被频繁调用的场景

Window.resize

Mousemove

上传进度。

解决方案：将即将被执行的函数用setTimeout 延迟一段时间执行。

var throttle = function ( fn, interval ) {

var \_\_self = fn, // 保存需要被延迟执行的函数引用

timer, // 定时器

firstTime = true; // 是否是第一次调用

return function () {

var args = arguments,

\_\_me = this;

if ( firstTime ) { // 如果是第一次调用，不需延迟执行

\_\_self.apply(\_\_me, args);

return firstTime = false;

}

if ( timer ) { // 如果定时器还在，说明前一次延迟执行还没有完成

return false;

}

timer = setTimeout(function () { // 延迟一段时间执行

clearTimeout(timer);

timer = null;

\_\_self.apply(\_\_me, args);

}, interval || 500 );

};

};

window.onresize = throttle(function(){

console.log( 1 );

}, 1000 );

1. 分时函数

一个例子是创建 WebQQ的 QQ好友列表。列表中通常会有成百上千个好友。比如把 1秒钟创建 1000个节点，改为每隔 200毫秒创建 8个节点。

var timeChunk = function( ary, fn, count ){

var obj,

t;

var len = ary.length;

var start = function(){

for ( var i = 0; i < Math.min( count || 1, ary.length ); i++ ){

var obj = ary.shift();

fn( obj );

}

};

return function(){

t = setInterval(function(){

if ( ary.length === 0 ){ // 如果全部节点都已经被创建好

return clearInterval( t );

}

start();

}, 200 ); // 分批执行的时间间隔，也可以用参数的形式传入

};

};

1. 惰性加载函数

但是在第一次进入条件分支之后，在函数内部会重写这个函数，重写之后的函数就是我们期望的 addEvent 函数，在下一次进入 addEvent 函数的时候， addEvent函数里不再存在条件分支语句：

var addEvent = function( elem,type,fn){

if( window.addEventListener ){

addEvent = function( elem,type,fn ){ //重写该方法

elem.addEventListener( type,fn,false );

}

}else if( window.attachEvent ){

addEvent = function( elem,type,fn ){ //重写该方法

elem.attachEvent( 'on' + type,fn );

}

}

addEvent( elem,type,fn );

};

var dd = document.getElementById('dd');

addEvent(dd,'click',function(){

alert('1111');

});

动态创建命名空间

var MyApp = {};

MyApp.namespace = function( name ){

var parts = name.split('.');

var current = MyApp;

for( var i = 0, len = parts.length; i < len; i++ ){

if( !current[ parts[ i ] ] ){

current[ parts[ i ] ] = {};

}

current = current[ parts[ i ] ];

}

};

MyApp.namespace( 'ss' );

MyApp.namespace( 'dd.ww' );

console.log( MyApp );

使用闭包封装私有变量

var user = (function(){

//私有变量

var \_\_name = 'fan',

\_\_age = '22';

return {

getInfo :function(){ //外部访问私有属性的接口

alert( '我的名字：' + \_\_name + '，我的年龄：' + \_\_age );

}

};

})();

user.getInfo();