javascript设计模式

单一职责原则（SRP）：应该仅有一个引起它变化的原因，被定义为“引起变化的原因”。

开放---封闭原则（OCP）：把经常改变的和基本不变的分开进行封装。

最少知识原则（LKP）：如果两个对象之间不必彼此直接通信，那么这两个对象就不要发生直接的相互联系。常见的做法是引入一个第三者对象，来承担这些对象之间的通信作用。

代码重构技巧：

1提炼函数（ 避免出现超大函数、独立出来的函数有助于代码复用、独立出来的函数更容易被覆写）

2合并重复的条件片段

3把条件分支语句提炼成函数

4合理使用循环

5提前让函数退出代替嵌套条件分支

6传递对象参数代替过长的参数列表

7尽量减少参数数量

8少用三目运算符

9合理使用链式调用

10分解大型类

11用 return 退出多重循环

各种模式的使用场景：

1原型模式----飞机分身技能

2单例模式----登录浮窗（只可能出现一次的）、

3策略模式----等级年终奖、实现缓动动画、表单校验

4代理模式----小明追 MM、图片预加载、缓存代理（计算乘积）

5迭代器模式----循环访问聚合对象中的各个元素、$.each

6发布--订阅模式----售楼处预约房子、DOM事件、网站登录（其他模块获取登录信息）

7命令模式----菜单程序

8组合模式----万能遥控器、扫描文件夹

9模板方法模式----Coffee or Tea、搭建项目的框架

10享元模式----文件上传、对象池

11职责链模式----手机商城订单

12中介者模式----机场指挥塔、泡泡堂游戏、购买商品

13装饰者模式----飞机大战的游戏（升级子弹变化）、插件式的表单验证

14状态模式----电灯开关、文件上传

15适配器模式----港式插头转换器、电源适配器、数据适配器

1. 原型模式

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。

代码：var plane1 = Object.create( plane );

1. 命令模式

使用用面向对象的方式实现命令模式

## 客户创建命令；调用者执行该命令；接收者在命令执行时执行相应操作。

应用场景：

1.使用命令模式作为"CallBack"在面向对象系统中的替代。

2.需要在不同的时间指定请求、将请求排队。

3.系统需要支持命令的撤消(undo)。

4.如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志修复。

## 作用：命令模式的意图是把请求封装为对象，从而分离请求的发起者和请求的接收者（执行者）之间的耦合关系。在命令被执行之前，可以预先往命令对象中植入命令的接收者。命令模式，通过command对象，连接接收者和调用者，降低了两者的耦合度，便于代码的修改和维护。命令模式提高了代码的模块化程度，但降低了代码的可阅读性（毕竟谁都不愿意看个代码的具体实现要逐层找半天，从调用者，到接收者），增加了代码的难度，所以只有在需要把操作和调用分离时，或需要对操作进行规范化处理时再用它。

代码：

var Tv = { //接收者-receiving,命令接收者对象提供实现具体的功能的方法

open:function(){

console.log( '打开电视机' );

},

close:function(){

console.log( '关闭电视机' );

}

};

var OpenTvCommand = function( receiver ){ //调用者-invoking,用来实现客户对象与接收者之间的低耦合，也就是弱化按钮之类的用户界面元素与其功能之间的耦合。

this.receiver = receiver; //预先植入的命令接收者被当成对象的属性保存起来

};

OpenTvCommand.prototype.execute = function(){

this.receiver.open(); // 执行命令，打开电视机

};

OpenTvCommand.prototype.undo = function(){

this.receiver.close(); // 撤销命令，关闭电视机

};

var setCommand = function( command ){ //客户-client,命令发布者客户对象就是页面上的控件（能绑定鼠标键盘等事件的dom元素）

document.getElementById( 'execute' ).onclick = function(){

command.execute(); // 输出：打开电视机

};

document.getElementById( 'undo' ).onclick = function(){

command.undo(); // 输出：关闭电视机

};

};

setCommand( new OpenTvCommand( Tv ) ); //执行命令

使用闭包方式实现命令模式

var Tv = {

open:function(){

console.log( '打开电视机' );

},

close:function(){

console.log( '关闭电视机' );

}

};

var createCommand = function( receiver ){ //命令接收者会被封闭在闭包形成的环境中

var execute = function(){

return receiver.open(); // 执行命令，打开电视机

};

var undo = function(){

return receiver.close(); // 执行命令，关闭电视机

};

return {

execute:execute,

undo:undo

};

};

var setCommand = function( command ){

document.getElementById( 'execute' ).onclick = function(){

command.execute(); // 输出：打开电视机

};

document.getElementById( 'undo' ).onclick = function(){

command.undo(); // 输出：关闭电视机

};

};

setCommand( createCommand( Tv ) );

1. 单例模式

定义：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

例子：这个登录浮窗是唯一的，无论单击多少次登录按钮，这个浮窗都只会被创建一次，那么这个登录浮窗就适合用单例模式来创建。

代码：

用代理实现单例模式

//单一职责原则----其只负责创建div

var CreateDiv = function( html ){

this.html = html;

this.init();

};

CreateDiv.prototype.init = function(){

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

div .innerHTML = this.html;

document.body.appendChild( div );

};

//使用代理----负责管理单例的逻辑

var ProxySingletonCreateDiv = (function(){

var instance;

return function( html ){

if( !instance ){

instance = new CreateDiv( html );

}

return instance;

};

})();

var d1 = ProxySingletonCreateDiv( '123' );

var d2 = ProxySingletonCreateDiv( '1233' );

console.log( d1===d2 ); //true

惰性单例-----WebQQ登录弹窗（需要的时候才创建）

//单一职责-----创建登录窗口

var createDiv = function( html ){

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

div.innerHTML = html;

document.body.appendChild(div);

div.style.display = 'none';

return div;

}

//代理保存单例逻辑

var createLoginLayer = (function(){

var instance;

return function( html ){

if( !instance ){

instance = createDiv( html );

}

return instance;

};

})();

document.getElementById('dd').onclick = function(){

var login = createLoginLayer( 'woshi dawang' );

login.style.display = 'block';

};

通用惰性单例

//单一职责-----创建登录窗口

var createLoginLayer = function(){

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

div.innerHTML = '余真帆';

document.body.appendChild(div);

div.style.display = 'none';

return div;

}

//单例逻辑

var getSingle = function( fn ){

var result;

return function(){

return result || ( result = fn.apply( this,arguments ) );

};

};

var createSingleLoginLayer = getSingle( createLoginLayer );

document.getElementById('dd').onclick = function(){

var login = createSingleLoginLayer();

login.style.display = 'block';

};

应用：绑定 click 事件

$( 'div' ).one( 'click', function(){ //jquery实现

alert ( 'click' );

});

//单例模式

var getSingle = function( fn ){

var result;

return function(){

return result || ( result = fn.apply( this,arguments ) );

};

};

var bindEvent = getSingle( function(){

document.getElementById('dd').onclick = function(){

alert('click');

};

return true;

} );

var render = function(){

console.log( '开始渲染列表' );

bindEvent();

};

render(); //其实只绑定了一个事件

render();

1. 策略模式

定义：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。

条条大路通罗马。

应用：使用策略模式计算年终奖。

目的：策略模式的目的就是将算法的使用与算法的实现分离开来

组成：第一个部分是一组策略类，策略类封装了具体的算法，并负责具体的计算过程。 第二个部分是环境类 Context，Context接受客户的请求，随后把请求委托给某一个策略类。

例子：绩效为 S的人年终奖有 4倍工资，绩效为 A的人年终奖有 3倍工资，而绩效为 B的人年终奖是 2倍工资。假设财务部要求我们提供一段代码，来方便他们计算员工的年终奖。

传统面向对象策略模式

//第一部分----一组策略类(具体的算法)

var PerformanceS = function(){};

PerformanceS.prototype.calculate = function( salary ){

return salary \* 4;

};

var PerformanceA = function(){};

PerformanceA.prototype.calculate = function( salary ){

return salary \* 3;

};

var PerformanceB = function(){};

PerformanceB.prototype.calculate = function( salary ){

return salary \* 2;

};

//第二部分-----contextContext接受客户的请求，随后把请求委托给某一个策略类。要做到这点，说明 Context中要维持对某个策略对象的引用。

var Bouns = function(){

this.salary = null; //原始工资

this.strategy = null; //绩效等级对应的策略对象

};

Bouns.prototype.setSalary = function( salary ){ //设置员工的原始工资

this.salary = salary;

};

Bouns.prototype.setStrategy = function( strategy ){ //设置员工绩效等级对应的策略对象

this.strategy = strategy;

};

Bouns.prototype.getBouns = function(){

return this.strategy.calculate( this.salary );

};

var bouns = new Bouns();

bouns.setSalary( 10000 );

bouns.setStrategy( new PerformanceA() );

console.log( bouns.getBouns() );

JS中的策略模式

//第一部分-----一组策略类，在js中函数也是对象

var strategies = {

"S":function( salary ){

return salary \* 4;

},

"A":function( salary ){

return salary \* 3;

},

"B":function( salary ){

return salary \* 2;

}

};

//第二部分-----context

var getBouns = function( salary,level ){

return strategies[level]( salary );

};

var dd = getBouns(10000,"A"); //对象的多态的体现

console.log( dd );

例子：策略模式实现缓动动画

//定义一组策略类

var tween = {

linear:function( t,b,c,d ){

return c\*t/d + b;

},

easeIn:function( t,b,c,d ){

return c\*(t/=d)\*t+b;

},

strongEaseIn:function( t,b,c,d ){

return c\*(t/=d)\*t\*t\*t\*t+b;

},

strongEaseOut:function( t,b,c,d ){

return c \* ( ( t = t / d - 1) \* t \* t \* t \* t + 1 ) + b;

},

sineaseIn: function( t, b, c, d ){

return c \* ( t /= d) \* t \* t + b;

},

sineaseOut: function(t,b,c,d){

return c \* ( ( t = t / d - 1) \* t \* t + 1 ) + b;

}

};

//context类

var Animate = function( dom ){

this.dom = dom; //进行运动的 dom 节点

this.startTime = 0; // 动画开始时间

this.startPos = 0; // 动画开始时，dom 节点的位置，即 dom 的初始位置

this.endPos = 0; // 动画结束时，dom 节点的位置，即 dom 的目标位置

this.propertyName = null; // dom 节点需要被改变的 css 属性名

this.easing = null; // 缓动算法

this.duration = null; // 动画持续时间

};

Animate.prototype.start = function( propertyName, endPos, duration, easing ){

this.startTime = +new Date; // 动画启动时间

this.startPos = this.dom.getBoundingClientRect()[ propertyName ]; //dom节点初始位置

this.propertyName = propertyName; // dom 节点需要被改变的 CSS 属性名

this.endPos = endPos; // dom 节点目标位置

this.duration = duration; // 动画持续事件

this.easing = tween[ easing ]; //缓动算法

var self = this;

var timeId = setInterval(function(){ // 启动定时器，开始执行动画

if( self.step() === false ){ // 如果动画已结束，则清除定时器

clearInterval( timeId );

}

},19);

};

Animate.prototype.step = function(){

var t = +new Date; // 取得当前时间获得毫秒数

if( t >= this.startTime + this.duration ){

this.update( this.endPos ); // 更新小球的 CSS 属性值

return false;

}

var pos = this.easing( t - this.startTime, this.startPos,this.endPos - this.startPos,this.duration );

//pos为小球的当前位置

this.update( pos ); // 更新小球的 CSS 属性值

};

Animate.prototype.update = function( pos ){

this.dom.style[ this.propertyName ] = pos + 'px';

};

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

var animate = new Animate( div );

animate.start( 'left',500,1000,'strongEaseOut' );

例子：表单校验----不光是封装算法，也可封装业务规则

很有用的例子

//第一部分--策略类

var strategies = {

isNonEmpty: function( value, errorMsg ){

if ( value === '' ){

return errorMsg;

}

},

minLength: function( value, length, errorMsg ){

if ( value.length < length ){

return errorMsg;

}

},

isMobile: function( value, errorMsg ){

if ( !/(^1[3|5|8][0-9]{9}$)/.test( value ) ){

return errorMsg;

}

}

};

//第二部分--context

var Validator = function(){

this.cache = [];

};

Validator.prototype.add = function( dom, rules ){

var self = this;

for ( var i = 0, rule; rule = rules[ i++ ]; ){

(function( rule ){

var strategyAry = rule.strategy.split( ':' ); //获得其对应的策略类方法

var errorMsg = rule.errorMsg;

self.cache.push(function(){

var strategy = strategyAry.shift();

strategyAry.unshift( dom.value ); //在头部插入其的值

strategyAry.push( errorMsg ); //在尾部插入错误信息

return strategies[ strategy ].apply( dom, strategyAry );

});

})( rule )

}

};

Validator.prototype.start = function(){

for ( var i = 0, validatorFunc; validatorFunc = this.cache[ i++ ]; ){

var errorMsg = validatorFunc();

if ( errorMsg ){

return errorMsg; //遇到第一个错误就会停止

}

}

};

//客户调用

var validataFunc = function(){

var validator = new Validator(); // 创建一个 validator 对象

validator.add( registerForm.userName,[{

strategy:'isNonEmpty',

errorMsg:'用户名不可为空'

},{

strategy:'minLength:10',

errorMsg:'用户名长度不能小于 10 位'

}]);

validator.add( registerForm.password, [{

strategy: 'minLength:6',

errorMsg: '密码长度不能小于 6 位'

}]);

validator.add( registerForm.phoneNumber, [{

strategy: 'isMobile',

errorMsg: '手机号码格式不正确'

}]);

var errorMsg = validator.start(); // 获得校验结果

return errorMsg; // 返回校验结果

};

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

registerForm.onsubmit = function(){

var errorMsg = validataFunc(); //如果 errorMsg 有确切的返回值，说明未通过校验

if( errorMsg ){

alert( errorMsg );

return false; // 阻止表单提交

}

};

1. 代理模式

定义：代理模式是为一个对象提供一个代用品或占位符，以便控制对它的访问。

生活实例：明星都有经纪人作为代理。

实例：小明→B(代理)→A(女神)；小明追女神的示例

var Flower = function(){};

var XiaoMing = {

sendFlower:function( target ){

var flower = new Flower();

target.receiverFlower( flower );

}

};

var B = { //此为代理对象

receiverFlower:function( flower ){

A.listenGoodMood(function(){

A.receiverFlower( flower );

});

}

};

var A = {

receiverFlower:function(){

console.log('收到花了');

},

listenGoodMood:function( fn ){

setTimeout(function(){

fn();

},5000);

}

};

XiaoMing.sendFlower( A );

保护代理和虚拟代理

保护代理----这种请求就可以直接在代理 B处被拒绝掉，这种代理叫作。

虚拟代理----把一些开销很大的对象，延迟到真正需要它的时候才去创建。

虚拟代理实现图片预加载

var myImage = (function(){

var imgNode = document.createElement( 'img' );

document.body.appendChild( imgNode );

return {

setSrc:function( src ){

imgNode.src = src;

}

}

})();

var proxyImage = (function(){

var img = new Image();

img.onload = function(){

myImage.setSrc( this.src );

};

return {

setSrc:function( src ){

myImage.setSrc( 'loading.gif' ); //菊花图

img.src = src;

}

};

})();

//代理对象和本体对象具有相同的对外接口setSrc函数

proxyImage.setSrc( 'big.png' );

虚拟代理合并 HTTP 请求

var synchronousFile = function( id ){

console.log( '开始同步文件，id为：' + id );

};

var proxySynchronousFile = (function(){

var cache = [], // 保存一段时间内需要同步的 ID

timer; //定时器

return function( id ){

cache.push( id );

if( timer ){ // 保证不会覆盖已经启动的定时器

return;

}

timer = setTimeout(function(){

synchronousFile( cache.join( ',' ) ); // 2 秒后向本体发送需要同步的 ID 集合

clearTimeout( timer ); // 清空计时器

timer = null;

cache.length = 0; // 清空 ID 集合

},2000);

};

})();

var checkbox = document.getElementsByTagName( 'input' );

for( var i = 0, c; c = checkbox[ i++ ]; ){

c.onclick = function(){

if( this.checked === true ){

proxySynchronousFile( this.id );

}

};

}

缓存代理

缓存代理可以为一些开销大的运算结果提供暂时的存储，在下次运算时，如果传递进来的参数跟之前一致，则可以直接返回前面存储的运算结果。

var mult = function(){

var a = 1;

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

a = a \* arguments[i];

}

return a;

};

var proxyMult = (function(){

var cache = {};

return function(){

var arg = [].join.call(this,',');

if( arg in cache ){

return cache[ arg ];

}

return cache[ arg ] = mult.apply( this,arguments );

};

})();

var dd = proxyMult(2,3,4);

console.log( dd );

缓存代理用于ajax异步请求数据

我们在常常在项目中遇到分页的需求，同一页的数据理论上只需要去后台拉取一次，这些已经拉取到的数据在某个地方被缓存之后，下次再请求同一页的时候，便可以直接使用之前的数据。

1. 代理模式

定义：迭代器模式是指提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。

内部迭代器的迭代规则已经被提前规定，下面为两个内部迭代器

jQuery中的 $.each 函数

$.each( [1,2,3],function( i,n ){

console.log('下标为：' + i + '值为：' + n);

} );

实现自己的迭代器

var each = function( ary,callback ){

for( var i = 0,l=ary.length ; i < l; i++ ){

callback.call( ary[i], i, ary[i] );

}

};

//使用

var dd = [1,2,3];

each( dd,function( i,n ){

console.log( i+'----' + n );

});

外部迭代器必须显式地请求迭代下一个元素。

var Iterator = function( obj ){

var current = 0;

var next = function(){

current ++;

};

var isDone = function(){

return current >= obj.length;

};

var getCurrItem = function(){

return obj[ current ];

};

return {

next:next,

isDone:isDone,

getCurrItem:getCurrItem

};

};

var compare = function( iterator1,iterator2 ){

while( !iterator1.isDone() && !iterator2.isDone() ){

if( iterator1.getCurrItem() !== iterator2.getCurrItem() ){

throw new Error('不相等');

}

iterator1.next();

iterator2.next();

}

return true;

};

var d1 = new Iterator([1,2,3]);

var d2 = new Iterator([1,2,3]);

console.log( compare( d1,d2 ) );

迭代类数组对象和字面量对象

$.each = function( obj,callback ){

Var value,

i = 0;

Length = obj.length,

isArray = isArraylike( obj );

If( isArray ){ // 迭代类数组

For( ; i < length; i++ ){

Value = callback.call( obj[ i ],i,obj[i] );

If( value === false ){

Break;

}

}

}else {

For( i in obj ){ // 迭代 object 对象

Value = callback.call( obj[i],i,obj[i] );

If( value === false ){

Break;

}

}

}

Return obj;

};

倒序迭代器

Var reverseEach = function( ary,callback ){

For( var l = ary.length - 1; l >= 0; l-- ){

Callback( l,ary[ l ] );

}

};

//使用

ReverseEach( [0,1,2],function( i,n ){

Console.log( n );

} );

中止迭代器

var each = function( ary,callback ){

for( var i = 0,l = ary.length; i < l; i++ ){

if( callback( i,ary[i] ) === false ){

break;

}

}

}

each( [1,2,3,4],function( i,n ){

if( n > 3 ){

return false;

}

console.log( n );

} );

例子：迭代器实例--3种方式上传文件

Var getActiveUploadObj = function(){

Try{

Return new ActiveXobject( ‘TXFTNActiveX.FTNUpload’ ); // IE 上传控件

}catch( e ){

Reutn false;

}

};

Var getFlashUploadObj = function(){

If( supportFlash() ){

Var str = ‘<object type=”application/x-shockwave-flash”></object>’;

Return $(str).appendChild( $(‘body’) );

}

Return false;

};

Var getFormUploadObj = function(){

Var str = ‘<input name=”file” type=”file”>’;

Return $( str ).appendChild( $(‘body’) );

};

//迭代器

Var iteratorUploadObj = function(){

For( var i = 0,fn; fn = arguments[ i++ ]; ){

Var uploadObj = fn();

if( fn !== false ){

Return uploadObj;

}

}

};

//调用

var uploadObj = iteratorUploadObj( getActiveUploadObj, getFlashUploadObj, getFormUpladObj );

1. 发布--订阅模式(观察者模式)

它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知。在 JavaScript开发中，我们一般用事件模型来替代传统的发布 — 订阅模式。

现实例子：小明，小红，小王与售楼处买房的故事。小明等位订阅者，售楼处作为发布者。

作用：发布 — 订阅模式可以广泛应用于异步编程中，这是一种替代传递回调函数的方案。

例子：

1、DOM事件 （发布 — 订阅模式）

document.body.addEventListener( 'click',function(){

alert(11);

},false );

document.body.click(); // 模拟用户点击---手动触发事件dispatchEvent

//我们订阅 document.body 上的 click 事件，当 body 节点被点击时， body 节点便会向订阅者发布这个消息。

2、自定义事件 （实现发布订阅模式的步骤）

1首先要指定好谁充当发布者（比如售楼处）；

2然后给发布者添加一个缓存列表，用于存放回调函数以便通知订阅者（售楼处的花名册）；

3最后发布消息的时候，发布者会遍历这个缓存列表，依次触发里面存放的订阅者回调函数（遍历花名册，挨个发短信）。

//通用的发布--订阅模式

var event = {

clientList: {}, // 缓存列表，存放订阅者的回调函数

listen: function( key, fn ){ //添加订阅者

if ( !this.clientList[ key ] ){ // 如果还没有订阅过此类消息，给该类消息创建一个缓存列表

this.clientList[ key ] = [];

}

this.clientList[ key ].push( fn ); // 订阅的消息添加进缓存列表

},

trigger: function(){ //发布消息

var key = Array.prototype.shift.call( arguments ), // 取出第一个参数

fns = this.clientList[ key ];

if ( !fns || fns.length === 0 ){ // 如果没有绑定对应的消息

return false;

}

for( var i = 0, fn; fn = fns[ i++ ]; ){

fn.apply( this, arguments ); // arguments 是 trigger 时带上的参数

}

},

remove: function( key,fn ){

var fns = this.clientList[ key ];

if ( !fns ){ // 如果 key 对应的消息没有被人订阅，则直接返回

return false;

}

if ( !fn ){ // 如果没有传入具体的回调函数，表示需要取消 key 对应消息的所有订阅

fns && ( fns.length = 0 );

}else{

for ( var l = fns.length - 1; l >=0; l-- ){ // 反向遍历订阅的回调函数列表

var \_fn = fns[ l ];

if ( \_fn === fn ){

fns.splice( l, 1 ); // 删除订阅者的回调函数

}

}

}

}

};

//再定义一个 installEvent 函数，这个函数可以给所有的对象都动态安装发布 — 订阅功能

var installEvent = function( obj ){

for( var i in event ){

obj[ i ] = event[ i ];

}

};

//测试

var salesOffices = {};

installEvent( salesOffices );

salesOffices.listen( 'squareMeter88',fn1 = function( price ){ // 小明订阅消息

console.log('价格=' + price);

} );

salesOffices.listen( 'squareMeter88',fn2 = function( price ){ // 小红订阅消息

console.log('价格1=' + price);

} );

salesOffices.remove('squareMeter88',fn1); // 删除小明的订阅

salesOffices.trigger( 'squareMeter88',2000000 );

例子：网站登录

我们正在开发一个商城网站，网站里有 header头部、nav导航、消息列表、购物车等模块，这几个模块的渲染有一个共同的前提条件，就是必须先用 ajax异步请求获取用户的登录信息。

//登录模块

$.ajax( 'http:// xxx.com?login', function(data){ // 登录成功

login.trigger( 'loginSucc', data); // 发布登录成功的消息

});

//各模块监听登录成功的消息：

var header = (function(){ // header 模块

login.listen( 'loginSucc', function( data){

header.setAvatar( data.avatar );

});

return {

setAvatar: function( data ){

console.log( '设置 header 模块的头像' );

}

}

})();

var nav = (function(){ // nav 模块

login.listen( 'loginSucc', function( data ){

nav.setAvatar( data.avatar );

});

return {

setAvatar: function( avatar ){

console.log( '设置 nav 模块的头像' );

}

}

})();

全局的发布 － 订阅对象

var Event = (function(){

var clientList = {},

listen,

trigger,

remove;

listen = function( key, fn ){

if ( !clientList[ key ] ){

clientList[ key ] = [];

}

clientList[ key ].push( fn );

};

trigger = function(){

var key = Array.prototype.shift.call( arguments ),

fns = clientList[ key ];

if ( !fns || fns.length === 0 ){

return false;

}

for( var i = 0, fn; fn = fns[ i++ ]; ){

fn.apply( this, arguments );

}

};

remove = function( key, fn ){

var fns = clientList[ key ];

if ( !fns ){

return false;

}

if ( !fn ){

fns && ( fns.length = 0 );

}else{

for ( var l = fns.length - 1; l >=0; l-- ){

var \_fn = fns[ l ];

if ( \_fn === fn ){

fns.splice( l, 1 );

}

}

}

};

return {

listen: listen,

trigger: trigger,

remove: remove

}

})();

Event.listen( 'squareMeter88', function( price ){ // 小红订阅消息

console.log( '价格= ' + price ); // 输出：'价格=2000000'

});

Event.trigger( 'squareMeter88', 2000000 ); // 售楼处发布消息

模块间通信

Var a = (function(){

Var count = 0;

Var button = document.getElementById( ‘count’ );

Button.onclick =function(){

Event.trigger( ‘add’,count++ ); //发布消息

};

})();

Var b = (function(){

Var div = document.getElementById( ‘show’ );

Event.listen(‘add’,function(){ //监听消息

div.innerHTML = count;

});

})();

全局事件的命名冲突

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* 使用命名空间 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

var Event = (function(){

var global = this,

Event,

\_default = 'default';

Event = function(){

var \_listen,

\_trigger,

\_remove,

\_slice = Array.prototype.slice,

\_shift = Array.prototype.shift,

\_unshift = Array.prototype.unshift,

namespaceCache = {},

\_create,

find,

each = function( ary, fn ){

var ret;

for ( var i = 0, l = ary.length; i < l; i++ ){

var n = ary[i];

ret = fn.call( n, i, n);

}

return ret;

};

\_listen = function( key, fn, cache ){

if ( !cache[ key ] ){

cache[ key ] = [];

}

cache[key].push( fn );

};

\_remove = function( key, cache ,fn){

if ( cache[ key ] ){

if( fn ){

for( var i = cache[ key ].length; i >= 0; i-- ){

if( cache[ key ][i] === fn ){

cache[ key ].splice( i, 1 );

}

}

}else{

cache[ key ] = [];

}

}

};

\_trigger = function(){

var cache = \_shift.call(arguments),

key = \_shift.call(arguments),

args = arguments,

\_self = this,

ret,

stack = cache[ key ];

if ( !stack || !stack.length ){

return;

}

return each( stack, function(){

return this.apply( \_self, args );

});

};

\_create = function( namespace ){

var namespace = namespace || \_default;

var cache = {},

offlineStack = [], // 离线事件

ret = {

listen: function( key, fn, last ){

\_listen( key, fn, cache );

if ( offlineStack === null ){

return;

}

if ( last === 'last' ){

offlineStack.length && offlineStack.pop()();

}else{

each( offlineStack, function(){

this();

});

}

offlineStack = null;

},

one: function( key, fn, last ){

\_remove( key, cache );

this.listen( key, fn ,last );

},

remove: function( key, fn ){

\_remove( key, cache ,fn);

},

trigger: function(){

var fn,

args,

\_self = this;

\_unshift.call( arguments, cache );

args = arguments;

fn = function(){

return \_trigger.apply( \_self, args );

};

if ( offlineStack ){

return offlineStack.push( fn );

}

return fn();

}

};

return namespace ?

( namespaceCache[ namespace ] ? namespaceCache[ namespace ] :

namespaceCache[ namespace ] = ret )

: ret;

};

return {

create: \_create,

one: function( key,fn, last ){

var event = this.create( );

event.one( key,fn,last );

},

remove: function( key,fn ){

var event = this.create( );

event.remove( key,fn );

},

listen: function( key, fn, last ){

var event = this.create( );

event.listen( key, fn, last );

},

trigger: function(){

var event = this.create( );

event.trigger.apply( this, arguments );

}

};

}();

return Event;

})();

//订阅

Event.create( 'namespace1' ).listen( 'click', function( a ){

console.log( a ); // 输出：1

});

//发布

Event.create( 'namespace1' ).trigger( 'click', 1 );

Event.create( 'namespace2' ).listen( 'click', function( a ){

console.log( a ); // 输出：2

});

Event.create( 'namespace2' ).trigger( 'click', 2 );

1. 命令模式

生活中的例子：这些记录着订餐信息的清单，便是命令模式中的命令对象。

应用场景：有时候需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是什么，此时希望用一种松耦合的方式来设计软件，使得请求发送者和请求接

收者能够消除彼此之间的耦合关系。

作用：通过命令对象将命令发送者和命令请求接收者解耦开的。

例子：——菜单程序（模拟传统面向对象语言的命令模式实现）

var button1 = document.getElementById( 'button1' ),

button2 = document.getElementById( 'button2' ),

button3 = document.getElementById( 'button3' );

//定义 setCommand 函数， setCommand 函数负责往按钮上面安装命令,执行命令的动作被约定为调用 command 对象的 execute() 方法。

//命令发布者--client

var setCommand = function( button,command ){

button.onclick = function(){

command.execute();

};

};

//命令接收者

var MenuBar = {

refresh:function(){

console.log( '刷新菜单目录' );

}

};

var SubMenu = {

add:function(){

console.log( '添加子菜单' );

},

del:function(){

console.log( '删除子菜单' );

}

};

//命令对象，解耦命令发送者和接收者

var RefreshMenuBarCommand = function ( receiver ) {

this.receiver = receiver;

};

RefreshMenuBarCommand.prototype.execute = function(){

this.receiver.refresh();

};

var AddSubMenuCommand = function( receiver ){

this.receiver = receiver;

};

AddSubMenuCommand.prototype.execute = function(){

this.receiver.add();

};

var DelSubMenuCommand = function( receiver ){

this.receiver = receiver;

};

DelSubMenuCommand.prototype.execute = function(){

this.receiver.del();

};

var refreshMenuBarCommand = new RefreshMenuBarCommand( MenuBar );

var addSubMenuCommand = new AddSubMenuCommand( SubMenu );

var delSubMenuCommand = new DelSubMenuCommand( SubMenu );

setCommand( button1, refreshMenuBarCommand );

setCommand( button2, addSubMenuCommand );

setCommand( button3, delSubMenuCommand );

用闭包实现的命令模式

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

var setCommand = function( button,command ){

button.onclick = function(){

command.execute();

};

};

var MenuBar = {

refresh:function(){

console.log( '刷新菜单界面' );

}

};

var RefreshMenuBarCommand = function( receiver ){

return {

execute:function(){

receiver.refresh();

}

};

};

var refreshMenuBarCommand = RefreshMenuBarCommand( MenuBar );

setCommand( button1,refreshMenuBarCommand );

撤消和重做---HTML5版《街头霸王》

var Ryu = {

attack: function(){

console.log( '攻击' );

},

defense: function(){

console.log( '防御' );

},

jump: function(){

console.log( '跳跃' );

},

crouch: function(){

console.log( '蹲下' );

}

};

//命令对象

var makeCommand = function( receiver, state ){ // 创建命令

return function(){

receiver[ state ]();

console.dir( dd );

}

};

//动作与keyCode相关联

var commands = {

"119": "jump", // W

"115": "crouch", // S

"97": "defense", // A

"100": "attack" // D

};

var commandStack = []; // 保存命令的堆栈

document.onkeyup = function( ev ){

var keyCode = ev.keyCode,

command = makeCommand( Ryu, commands[ keyCode ] );

if ( command ){

command(); // 执行命令

commandStack.push( command ); // 将刚刚执行过的命令保存进堆栈

}

};

document.getElementById( 'replay' ).onclick = function(){ // 点击播放录像

var command;

while( command = commandStack.shift() ){ // 从堆栈里依次取出命令并执行

command();

}

};

宏命令----通过执行宏命令的方式，可以一次执行一批命令。

//命令接收者

var closeDoorCommand = {

execute:function(){

console.log( '关门' );

}

};

var openPcCommand = {

execute:function(){

console.log( '开电脑' );

}

};

var openQQCommand = {

execute:function(){

console.log( '登录QQ' );

}

};

//定义宏命令对象

var MacroCommand = function(){

return {

commandList : [],

add:function( command ){

this.commandList.push( command );

},

excute:function(){

for( var i = 0,command; command = this.commandList[i++]; ){

command.execute();

}

}

};

};

//命令发布者

var dd = document.getElementById( 'replay' ).onclick = function(){

var macroCommand = MacroCommand();

macroCommand.add( closeDoorCommand );

macroCommand.add( openPcCommand );

macroCommand.add( openQQCommand );

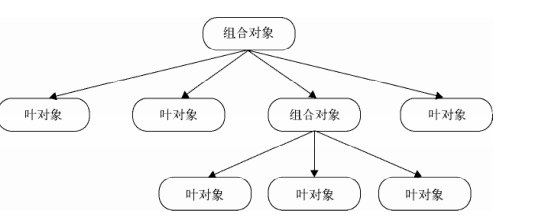
macroCommand.excute();

};

1. 组合模式

定义：组合模式就是用小的子对象来构建更大的对象，而这些小的子对象本身也许是由更小的“孙对象”构成的。

用途：组合模式将对象组合成树形结构，以表示“部分整体”的层次结构。 除了用来表示树形结构之外，组合模式的另一个好处是通过对象的多态性表现，使得用户对单个对象和组合对象的使用具有一致性。



使用场景：

表示对象的部分整体层次结构。特别是我们在开发期间不确定这棵树到底存在多少层次的时候。

客户希望统一对待树中的所有对象。组合模式使客户可以忽略组合对象和叶对象的区别。

代码：

//组合对象

var MacroCommand = function(){

return {

commandList:[],

add:function( command ){

this.commandList.push( command );

},

execute:function(){

for( var i = 0, command; command = this.commandList[i++]; ){

command.execute();

}

},

};

};

//开空调叶对象

var openAcCommand = {

add:function(){

throw new Error( '叶对象不能添加子节点' );

},

execute:function(){

console.log('打开空调');

}

};

//电视、音响用同一个开关

var openTvCommand = {

add:function(){

throw new Error( '叶对象不能添加子节点' );

},

execute:function(){

console.log( '打开电视' );

}

};

var openSoundCommand = {

add:function(){

throw new Error( '叶对象不能添加子节点' );

},

execute:function(){

console.log( '打开音响' );

}

};

//组合命令对象1

var macroCommand1 = MacroCommand();

macroCommand1.add( openTvCommand );

macroCommand1.add( openSoundCommand );

var closeDoorCommand = {

add:function(){

throw new Error( '叶对象不能添加子节点' );

},

execute:function(){

console.log( '关门' );

}

};

var openPcCommand = {

add:function(){

throw new Error( '叶对象不能添加子节点' );

},

execute:function(){

console.log( '开电脑' );

}

};

var openQQCommand = {

add:function(){

throw new Error( '叶对象不能添加子节点' );

},

execute:function(){

console.log( '登录QQ' );

}

};

//组合命令对象2

var macroCommand2 = MacroCommand();

macroCommand2.add( closeDoorCommand );

macroCommand2.add( openPcCommand );

macroCommand2.add( openQQCommand );

//再把两个组合命令再进行组合

var macroCommand = MacroCommand();

macroCommand.add( openAcCommand );

macroCommand.add( macroCommand1 );

macroCommand.add( macroCommand2 );

openTvCommand.add( macroCommand )

//最后给遥控器绑定“超级命令”运行只需要调用最上层对象的 execute 方法。

var setCommand = (function( command ){

document.getElementById('button').onclick = function(){

command.execute();

};

})( macroCommand );

例子：扫描删除文件夹（文件夹与文件的关系类似与上面）

//文件与文件夹

//Folder

var Folder = function( name ){

this.name = name;

this.parent = null; //增加 this.parent 属性

this.files = [];

};

Folder.prototype.add = function( file ){

file.parent = this; //设置父对象

this.files.push( file );

};

Folder.prototype.scan = function(){

for( var i = 0, file, files = this.files; file = files[i++]; ){

file.scan();

}

};

Folder.prototype.remove = function(){

if( !this.parent ){ //根节点或者树外的游离节点

return;

}

for( var files = this.parent.files, l = files.length - 1; l >= 0; l-- ){

var file = files[l];

if( file === this ){

files.splice( l,1 );

}

}

};

//File

var File = function( name ){

this.name = name;

this.parent = null;

};

File.prototype.add = function(){

throw new Error( '文件下面不能再添加文件' );

};

File.prototype.scan = function(){

console.log( '开始扫描文件：' + this.name );

};

File.prototype.remove = function(){

if( !this.parent ){ //根节点或者树外的游离节点

return;

}

for( var files = this.parent.files, l = files.length - 1; l >= 0; l-- ){

var file = files[l];

if( file === this ){

files.splice( l,1 );

}

}

};

var folder = new Folder( '学习资料' );

var folder1 = new Folder( 'js' );

var folder2 = new Folder( 'jq' );

var file1 = new File( 'js实战' );

var file2 = new File( '精通jq' );

var file3 = new File( '重构与模式' );

folder1.add( file1 );

folder2.add( file2 );

folder.add( folder1 );

folder.add( folder2 );

folder.add( file3 );

folder1.remove();

file3.remove();

//启动扫描文件

folder.scan();

1. 模板方法模式

定义：基于继承的设计模式——模板方法（Template Method）模式。

模板方法模式由两部分结构组成，第一部分是抽象父类（相同部分），第二部分是具体的实现子类（不同部分）。通常在抽象父类中封装了子类的算法框架，包括实现一些公共方法以及封装子类中所有方法的执行顺序。子类通过继承这个抽象类，也继承了整个算法结构，并且可以选择重写父类的方法。

例子：茶和coffee

例子中钩子方法：钩子方法的返回结果决定了模板方法后面部分的执行步骤，也就是程序接下来的走向，这样一来，程序就拥有了变化的可能。

基于继承的方式实现

//抽象父类

var Beverage = function(){};

//相同的部分

Beverage.prototype.boilWater = function(){

console.log( '把水煮沸' );

};

//不同的部分，需要子类重写

Beverage.prototype.brew = function(){

throw new Error( '子类必须重写该方法' );

};

Beverage.prototype.pourInCup = function(){

throw new Error( '子类必须重写该方法' );

};

Beverage.prototype.addCondiments = function(){

throw new Error( '子类必须重写该方法' );

};

//钩子函数决定是否执行特定步骤

Beverage.prototype.customerWantsCondiments = function(){

return true; // 默认需要调料

};

//此模式的重点模版方法----父类中封装子类的算法框架

Beverage.prototype.init = function(){

this.boilWater();

this.brew();

this.pourInCup();

if( this.customerWantsCondiments() ){ // 如果挂钩返回 true，则需要调料

this.addCondiments();

}

};

//coffee子类

var Coffee = function(){};

//继承父类

Coffee.prototype = new Beverage();

//重写父类的相关方法

Coffee.prototype.brew = function(){

console.log( '用沸水冲泡coffee' );

};

Coffee.prototype.pourInCup = function(){

console.log( '把咖啡倒进杯子' );

};

Coffee.prototype.addCondiments = function(){

console.log( '加糖和牛奶' );

};

Coffee.prototype.customerWantsCondiments = function(){

return confirm( '你确定需要调料吗？' );

};

var coffee = new Coffee();

coffee.init();

高阶函数方式-----挂钩到高层组件中，而高层组件会决定什么时候、以何种方式去使用这些底层组件

var Beverage = function( param ){

var boilWater = function(){

console.log( '把水煮沸' );

};

var brew = param.brew || function(){

throw new Error( '必须传递 brew 方法' );

};

var pourInCup = param.pourInCup || function(){

throw new Error( '必须传递 pourInCup 方法' );

};

var addCondiments = param.addCondiments || function(){

throw new Error( '必须传递 addCondiments 方法' );

};

//钩子函数,默认添加

var addBoolean = param.addBoolean || function(){

return true;

};

var F = function(){};

F.prototype.init = function(){

boilWater();

brew();

pourInCup();

if( addBoolean() ){

addCondiments();

}

};

return F;

};

//coffee

var Coffee = Beverage({

brew:function(){

console.log( '用沸水冲泡咖啡' );

},

pourInCup:function () {

console.log( '把咖啡倒进杯子' );

},

addCondiments:function(){

console.log( '加糖和牛奶' );

},

addBoolean:function(){

return window.confirm( '要调加调料吗？' );

}

});

var coffee = new Coffee();

coffee.init();

1. 享元（flyweight）模式

享元（flyweight）模式是一种用于性能优化的模式，“fly”在这里是苍蝇的意思，意为蝇量

级。享元模式的核心是运用共享技术来有效支持大量细粒度的对象。

享元模式要求将对象的属性划分为内部状态与外部状态（状态在这里通常指属性）。

区分内部状态与外部状态：

内部状态存储于对象内部。

内部状态可以被一些对象共享。

内部状态独立于具体的场景，通常不会改变。

外部状态取决于具体的场景，并根据场景而变化，外部状态不能被共享。

例子：文件上传（微云）

var Upload = function( uploadType ){

//作为内部状态

this.uploadType = uploadType;

};

Upload.prototype.delFile = function( id ){

uploadManager.setExternalState( id, this );

if( this.fileSize < 3000 ){

return this.dom.parentNode.removeChild( this.dom );

}

if( window.confirm( '确定要删除该文件吗? ' + this.fileName ) ){

return this.dom.parentNode.removeChild( this.dom );

}

};

//定义一个工厂来创建 upload 对象

var UploadFactory = (function(){

var createFlyWeightObjs = {};

return {

create:function( uploadType ){

if( createFlyWeightObjs[ uploadType ] ){

return createFlyWeightObjs[ uploadType ];

}

return createFlyWeightObjs[ uploadType ] = new Upload( uploadType );

}

};

})();

//负责向 UploadFactory 提交创建对象的请求，并用一个 uploadDatabase 对象保存所有 upload 对象的外部状态

var uploadManager = (function(){

var uploadDatabase = {};

return {

add:function( id, uploadType, fileName, fileSize ){

var flyWeightObj = UploadFactory.create( uploadType );

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

dom.innerHTML =

'<span>文件名称：' + fileName +',文件大小：' + fileSize + '</span>' +

'<button class="delFile">删除</button>';

dom.querySelector( '.delFile' ).onclick = function(){

flyWeightObj.delFile( id );

}

document.body.appendChild( dom );

uploadDatabase[id] = {

fileName:fileName,

fileSize:fileSize,

dom:dom

};

return flyWeightObj;

},

setExternalState:function( id, flyWeightObj ){

var uploadData = uploadDatabase[id];

for( var i in uploadData ){

flyWeightObj[i] = uploadData[i];

}

}

};

})();

var id = 0;

window.startUpload = function( uploadType, files ){

for( var i = 0, file; file = files[i++]; ){

var uploadObj = uploadManager.add( ++id, uploadType, file.fileName, file.fileSize );

}

};

//测试

startUpload( 'plugin',[{

fileName:'1.txt',

fileSize:1000

},

{

fileName:'2.txt',

fileSize:3000

},

{

fileName:'3.txt',

fileSize:5000

}] );

startUpload( 'flash',[{

fileName:'4.txt',

fileSize:1000

},

{

fileName:'5.html',

fileSize:3000

},

{

fileName:'6.txt',

fileSize:5000

}] )

例子：对象池实现----地图应用

//通用对象池实现

var objectPoolFactory = function( createObjFn ){

var objectPool = [];

return {

create:function(){ //创建对象

var obj = objectPool.length === 0 ?

createObjFn.apply( this,arguments ) : objectPool.shift();

return obj;

},

recover:function( obj ){ //回收对象

objectPool.push( obj );

},

};

};

//objectPoolFactory 来创建一个装载一些 iframe 的对象池

var iframeFactory = objectPoolFactory( function(){

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

document.body.appendChild( iframe );

iframe.onload = function(){

iframe.onload = null; // 防止 iframe 重复加载的 bug

iframeFactory.recover( iframe ); // iframe 加载完成之后回收节点

};

return iframe;

} );

var iframe1 = iframeFactory.create();

iframe1.src = 'https://www.baidu.com/';

var iframe2 = iframeFactory.create();

iframe2.src = 'http://www.runoob.com/';

setTimeout(function(){

var iframe3 = iframeFactory.create();

iframe3.src = 'https://vuefe.cn/';

}, 3000 );

1. 职责链模式

定义：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间

的耦合关系，将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

例子：售卖手机的电商网站

//orderType:表示订单类型1,2,3

//pay:是否支付定金

//stock:库存

var order500 = function( orderType, pay, stock ){

if( orderType === 1 && pay === true ){

console.log( '500 元定金预购, 得到 100 优惠券' );

}else{

return 'nextSuccessor'; // 我不知道下一个节点是谁，反正把请求往后面传递

}

};

var order200 = function( orderType, pay, stock ){

if( orderType === 2 && pay === true ){

console.log( '200 元定金预购, 得到 50 优惠券' );

}else{

return 'nextSuccessor'; // 我不知道下一个节点是谁，反正把请求往

}

};

var orderNormal = function( orderType, pay, stock ){

if( stock > 0 ){

console.log( '普通购买, 无优惠券' );

}else{

console.log( '手机库存不足' );

}

};

//把函数包装进职责链节点，我们定义一个构造函数 Chain

// Chain.prototype.setNextSuccessor 指定在链中的下一个节点

// Chain.prototype.passRequest 传递请求给某个节点

var Chain = function( fn ){

this.fn = fn;

this.successor = null; //保存下个节点

};

Chain.prototype.setNextSuccessor = function( successor ){

return this.successor = successor;

};

Chain.prototype.passRequest = function(){

var ret = this.fn.apply( this, arguments );

if (ret === 'nextSuccessor') {

return this.successor && this.successor.passRequest.apply( this.successor, arguments );

}

return ret;

};

//把 3个订单函数分别包装成职责链的节点

var chainOrder500 = new Chain( order500 );

var chainOrder200 = new Chain( order200 );

var chainOrderNormal = new Chain( orderNormal );

//指定节点中职责链的顺序

chainOrder500.setNextSuccessor( chainOrder200 );

chainOrder200.setNextSuccessor( chainOrderNormal );

//测试，都是从第一个节点开始

chainOrder500.passRequest( 1, true, 500 ); // 输出：500 元定金预购，得到 100 优惠券

chainOrder500.passRequest( 2, true, 500 ); // 输出：200 元定金预购，得到 50 优惠券

chainOrder500.passRequest( 3, true, 500 ); // 输出：普通购买，无优惠券

chainOrder500.passRequest( 1, false, 0 ); // 输出：手机库存不足

//手动指定起始节点----明确其为普通购买

chainNormal.passRequest( 1, false, 500 );

AOP 实现职责链----- JavaScript的函数式特性

Function.prototype.after = function( fn ){

var self = this;

return function(){

var ret = self.apply( this, arguments );

if( ret === 'nextSuccessor' ){

return fn.apply( this, arguments );

}

return ret;

};

};

//设置职责链

var order = order500.after( order200 ).after( orderNormal );

order( 1, true, 500 ); // 输出：500 元定金预购，得到 100 优惠券

order( 2, true, 500 ); // 输出：200 元定金预购，得到 50 优惠券

order( 1, false, 500 ); // 输出：普通购买，无优惠券

用职责链模式获取文件上传对象

var getActiveUploadObj = function(){

try{

return new ActiveObject('TXFTNActiveX.FTNUpload'); // IE 上传控件

}catch( e ){

return 'nextSuccessor';

}

};

var getFlashUploadObj = function(){

if( supportFlash() ){

var str = '<object type="application/x-shockwave-flash"></object>';

return $( str ).appendTo( $('body') );

}

return 'nextSuccessor' ;

};

var getFormUpladObj = function(){

return $( '<form><input name="file" type="file"/></form>' ).appendTo( $('body') );

};

//设置职责链

var getUploadObj = getActiveUploadObj.after( getFlashUploadObj ).after( getFormUpladObj );

1. 中介者模式

定义：中介者模式的作用就是解除对象与对象之间的紧耦合关系。增加一个中介者对象后，所有的相关对象都通过中介者对象来通信，而不是互相引用，所以当一个对象发生改变时，只需要通知中介者对象即可。

例子：泡泡堂游戏

//定义玩家

function Player( name, teamColor ){

this.name = name; //角色名字

this.teamColor = teamColor; // 队伍颜色

this.state = 'alive';

}

Player.prototype.win = function(){

console.log( this.name + 'won' );

};

Player.prototype.lose = function(){

console.log( this.name + 'lost' );

};

Player.prototype.die = function(){

this.state = 'dead';

playerDirector.reciveMessage( 'playerDead', this ); // 给中介者发送消息，玩家死亡

};

Player.prototype.remove = function(){

playerDirector.reciveMessage( 'removePlayer', this ); // 给中介者发送消息，移除一个玩家

};

Player.prototype.changeTeam = function( color ){

playerDirector.reciveMessage( 'changeTeam', this, color ); // 给中介者发送消息，玩家换队

};

//创建玩家对象的工厂函数

var playerFactory = function( name, teamColor ){

var newPlayer = new Player( name, teamColor ); // 创造一个新的玩家对象

playerDirector.reciveMessage( 'addPlayer', newPlayer ); // 给中介者发送消息，新增玩家

return newPlayer;

};

//中介对象playerDirector

var playerDirector = (function(){

var players = {}, // 保存所有玩家

operations = {}; // 中介者可以执行的操作

//新增玩家

operations.addPlayer = function( player ){

var teamColor = player.teamColor; // 玩家的队伍颜色

players[teamColor] = players[teamColor] || [];// 如果该颜色的玩家还没有成立队伍，则新成立一个队伍

players[teamColor].push( player ); // 添加玩家进队伍

};

//移除一个玩家

operations.removePlayer = function( player ){

var teamColor = player.teamColor, // 玩家的队伍颜色

teamPlayers = players[teamColor] || []; // 该队伍所有成员

for( var i = teamPlayers.length - 1; i >= 0; i-- ){ // 遍历删除

if( teamPlayers[i] === player ){

teamPlayers.splice( i, 1 );

}

}

};

//玩家换队

operations.changeTeam = function( player, newTeamColor ){ // 玩家换队

operations.removePlayer( player ); // 从原队伍中删除

player.teamColor = newTeamColor; // 改变队伍颜色

operations.addPlayer( player ); // 增加到新队伍中

};

//玩家死亡

operations.playerDead = function( player ){

var teamColor = player.teamColor,

teamPlayers = players[teamColor]; // 玩家所在队伍

var all\_dead = true;

for( var i = 0, player; player =teamPlayers[i++]; ){

if( player.state !== 'dead' ){

all\_dead = false;

break;

}

}

if( all\_dead === true ){ // 全部死亡

for( var i = 0, player; player = teamPlayers[i++]; ){

player.lose(); // 本队所有玩家 lose

}

for( var color in players ){

if( color !== teamColor ){

var teamPlayers = players[color]; // 其他队伍的玩家

for( var i = 0, player; player = teamPlayers[i++]; ){

player.win(); // 其他队伍所有玩家 win

}

}

}

}

};

var reciveMessage = function(){

var message = Array.prototype.shift.call( arguments ); // arguments 的第一个参数为消息名称

operations[ message ].apply( this, arguments );

};

return {

reciveMessage:reciveMessage

};

})();

//测试

// 红队：

var player1 = playerFactory( '皮蛋', 'red' ),

player2 = playerFactory( '小乖', 'red' ),

player3 = playerFactory( '宝宝', 'red' ),

player4 = playerFactory( '小强', 'red' );

// 蓝队：

var player5 = playerFactory( '黑妞', 'blue' ),

player6 = playerFactory( '葱头', 'blue' ),

player7 = playerFactory( '胖墩', 'blue' ),

player8 = playerFactory( '海盗', 'blue' );

player1.remove();

player2.remove();

player3.die();

player4.die();

中介者模式的例子——购买商品

Select标签为onchange事件/input标签为oninput事件

//模拟服务器返回的数据

var goods = { // 手机库存

"red|32G": 3, // 红色 32G，库存数量为 3

"red|16G": 23,

"blue|32G": 12,

"blue|16G": 6

};

var mediator = (function(){

var colorSelect = document.getElementById( 'colorSelect' ),

memorySelect = document.getElementById( 'memorySelect' ),

numberInput = document.getElementById( 'numberInput' ),

colorInfo = document.getElementById( 'colorInfo' ),

memoryInfo = document.getElementById( 'memoryInfo' ),

numberInfo = document.getElementById( 'numberInfo' ),

nextBtn = document.getElementById( 'nextBtn' );

return {

changed:function( obj ){

var color = colorSelect.value, // 颜色

memory = memorySelect.value, // 内存

number = numberInput.value, // 数量

stock = goods[ color + '|' + memory ]; // 颜色和内存对应的手机库存数量

if( obj === colorSelect ){ // 如果改变的是选择颜色下拉框

colorInfo.innerHTML = color;

}else if( obj === memorySelect ){

memoryInfo.innerHTML = memory;

}else if( obj === numberInput ){

numberInfo.innerHTML = number;

}

if( !color ){

nextBtn.disabled = true;

nextBtn.innerHTML = '请选择颜色';

return;

}

if( !memory ){

nextBtn.disabled = true;

nextBtn.innerHTML = '请选择内存';

return;

}

if( !memory ){

nextBtn.disabled = true;

nextBtn.innerHTML = '请选择内存';

return;

}

if ( !/[1-9]{1,}\d{0,}/.test( number )){

nextBtn.disabled = true;

nextBtn.innerHTML = '请输入正确的购买数量';

return;

}

if ( stock < number ){

nextBtn.disabled = true;

nextBtn.innerHTML = '库存不足';

return;

}

nextBtn.disabled = false;

nextBtn.innerHTML = '放入购物车';

},

};

})();

// 事件函数：

colorSelect.onchange = function(){

mediator.changed( this );

};

memorySelect.onchange = function(){

mediator.changed( this );

};

numberInput.oninput = function(){

mediator.changed( this );

};

1. 装饰着模式（包装器）

定义：这种给对象动态地增加职责的方式称为装饰者（decorator）模式。装饰者模式能够在不改变对象自身的基础上，在程序运行期间给对象动态地添加职责。 即用即付

例子：飞机大战的游戏 -----传统面向对象语言中的装饰者模式实现

//原始的飞机类

var Plane = function(){};

Plane.prototype.fire = function(){

console.log( '发射普通子弹' );

};

//两个装饰类，分别是导弹和原子弹

var MissileDecorator = function( plane ){

this.plane = plane;

};

MissileDecorator.prototype.fire = function(){

this.plane.fire();

console.log( '发射导弹' );

};

var AtomDecorator = function( plane ){

this.plane = plane;

};

AtomDecorator.prototype.fire = function(){

this.plane.fire();

console.log( '发射原子弹' );

};

var plane = new Plane();

plane = new MissileDecorator( plane );

plane = new AtomDecorator( plane );

plane.fire( );

Js中的装饰者

var plane = {

fire:function(){

console.log( '发射普通子弹' );

}

};

var missileDecorator = function(){

console.log( '发射导弹' );

};

var atomDecorator = function(){

console.log( '发射原子弹' );

};

var fire1 = plane.fire;

plane.fire = function(){

fire1();

missileDecorator();

};

var fire2 = plane.fire;

plane.fire = function(){

fire2();

atomDecorator();

}

plane.fire();

// 分别输出： 发射普通子弹、发射导弹、发射原子弹

装饰者函数

Function.prototype.before = function( beforefn ){

var \_\_self = 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，且保证 this 不被劫持，新函数接受的参数

// 也会被原封不动地传入原函数，新函数在原函数之前执行

return \_\_self.apply( this, arguments ); // 执行原函数并返回原函数的执行结果，

// 并且保证 this 不被劫持

}

}

Function.prototype.after = function( afterfn ){

var \_\_self = this;

return function(){

var ret = \_\_self.apply( this, arguments );

afterfn.apply( this, arguments );

return ret;

}

};

例子：数据统计上报

Function.prototype.after = function( afterfn ){

var \_\_self = this;

return function(){

var ret = \_\_self.apply( this, arguments );

afterfn.apply( this, arguments );

return ret;

}

};

var showLogin = function(){

console.log( '打开登录浮层' );

}

var log = function(){

console.log( '上报标签为: ' + this.getAttribute( 'tag' ) );

}

showLogin = showLogin.after( log ); // 打开登录浮层之后上报数据

document.getElementById( 'button' ).onclick = showLogin;

例子：插件式的表单验证

Function.prototype.before = function( beforefn ){

var \_\_self = this;

return function(){

if ( beforefn.apply( this, arguments ) === false ){

// beforefn 返回 false 的情况直接 return，不再执行后面的原函数

return;

}

return \_\_self.apply( this, arguments );

}

}

var validata = function(){

if( !username.value ){

console.log( '用户名不可为空' );

return false;

}

if ( password.value === '' ){

console.log( '密码不能为空' );

return false;

}

};

var formSubmit = function(){

var param = {

username:username.value,

password:password.value

};

console.dir( param );

};

formSubmit = formSubmit.before( validata );

submitBtn.onclick = function(){

formSubmit();

};

1. 状态模式

定义：将状态封装成独立的类，在不同的状态下具有截然不同的行为，这个对象看起来是从不同的类中实例化而来的，实际上这是使用了委托的效果。

状态模式的关键是区分事物内部的状态，事物内部状态的改变往往会带来事物的行为改变。

特点：状态模式的关键是把事物的每种状态都封装成单独的类，跟此种状态有关的行为都被封装在这个类的内部。

使用场景：文件上传，还是音乐、视频播放器等等。

例子：电灯程序

//定义状态父类

var State = function(){};

State.prototype.buttonWasPressed = function(){

throw new Error( '父类的 buttonWasPressed 方法必须被重写' );

};

//3个状态类

//OffLightState

var OffLightState = function( light ){

this.light = light;

};

OffLightState.prototype = new State(); //继承抽象父类

OffLightState.prototype.buttonWasPressed = function(){

console.log( '弱光' ); // offLightState 对应的行为

this.light.setState( this.light.weakLightState ); // 切换状态到 weakLightState

};

// WeakLightState：

var WeakLightState = function( light ){

this.light = light;

};

WeakLightState.prototype = new State();

WeakLightState.prototype.buttonWasPressed = function(){

console.log( '强光' ); // weakLightState 对应的行为

this.light.setState( this.light.strongLightState ); // 切换状态到 strongLightState

};

// StrongLightState：

var StrongLightState = function( light ){

this.light = light;

};

StrongLightState.prototype = new State();

StrongLightState.prototype.buttonWasPressed = function(){

console.log( '关灯' ); // strongLightState 对应的行为

this.light.setState( this.light.offLightState ); // 切换状态到 offLightState

};

//上下文（Context）

var Light = function(){

//context保持对状态对象的引用

this.offLightState = new OffLightState( this );

this.weakLightState = new WeakLightState( this );

this.strongLightState = new StrongLightState( this );

this.button = null;

};

Light.prototype.init = function(){

var button = document.createElement( 'button' ),

self = this;

this.button = document.body.appendChild( button );

this.button.innerHTML = '开关';

this.currState = this.offLightState; // 设置当前状态

this.button.onclick = function(){

self.currState.buttonWasPressed();

};

};

Light.prototype.setState = function( newState ){

this.currState = newState;

};

//测试

var light = new Light();

light.init();

例子：JavaScript 版本的状态机-----电灯

var Light = function(){

this.currState = FSM.off; // 设置当前状态

this.button = null;

};

Light.prototype.init = function(){

var button = document.createElement( 'button' ),

self = this;

button.innerHTML = '已关灯';

this.button = document.body.appendChild( button );

this.button.onclick = function(){

self.currState.buttonWasPressed.call( self ); // 把请求委托给 FSM 状态机

};

};

var FSM = {

off:{

buttonWasPressed:function(){

console.log( '关灯' );

this.button.innerHTML = '下一次按我是开灯';

this.currState = FSM.on;

}

},

on:{

buttonWasPressed: function(){

console.log( '开灯' );

this.button.innerHTML = '下一次按我是关灯';

this.currState = FSM.off;

}

}

};

var light = new Light();

light.init();

例子：闭包改写-----电灯

var delegate = function( client, delegation ){

return {

buttonWasPressed:function(){ // 将客户的操作委托给 delegation 对象

return delegation.buttonWasPressed.apply( client, arguments );

}

}

};

var FSM = {

off: {

buttonWasPressed: function(){

console.log( '关灯' );

this.button.innerHTML = '下一次按我是开灯';

this.currState = this.onState;

}

},

on: {

buttonWasPressed: function(){

console.log( '开灯' );

this.button.innerHTML = '下一次按我是关灯';

this.currState = this.offState;

}

}

};

var Light = function(){

this.offState = delegate( this, FSM.off );

this.onState = delegate( this, FSM.on );

this.currState = this.offState; // 设置初始状态为关闭状态

this.button = null;

};

Light.prototype.init = function(){

var button = document.createElement( 'button' ),

self = this;

button.innerHTML = '已关灯';

this.button = document.body.appendChild( button );

this.button.onclick = function(){

self.currState.buttonWasPressed();

}

};

var light = new Light();

light.init();

1. 适配器模式

作用：适配器模式的作用是解决两个软件实体间的接口不兼容的问题。

例子：地图应用

var googleMap = {

show:function(){

console.log( '开始渲染谷歌地图' );

}

};

var baiduMap = {

display:function(){

console.log( '开始渲染百度地图' );

}

};

//百度地图适配器，使其拥有show方法

var baiduMapAdapter = {

show:function(){

return baiduMap.display();

}

};

var renderMap = function( obj ){

obj.show();

};

renderMap( googleMap ); // 输出：开始渲染谷歌地图

renderMap( baiduMapAdapter ); // 输出：开始渲染百度地图

例子：数据适配器

var getGuangdongCity = function(){

var guangdongCity = [{

name:'shenzhen',

id:11,

},{

name:'gaungzhou',

id:12,

}];

//将上面数据调整为如下

/\*var guangdongCity = {

shenzhen: 11,

guangzhou: 12,

zhuhai: 13

};\*/

return guangdongCity;

};

var render = function( fn ){

console.log( '开始渲染广东省地图' );

document.write( JSON.stringify( fn() ) );

};

var addressAdapter = function( oldAddressfn ){

var address = {},

oldAddress = oldAddressfn();

for( var i = 0, c; c = oldAddress[i++]; ){

address[c.name] = c.id;

}

return function(){

return address;

};

};

render( addressAdapter( getGuangdongCity ) );