javascript设计模式

单一职责原则：应该仅有一个引起它变化的原因。

开放---封闭原则：把经常改变的和基本不变的分开进行封装。

最少知识原则：

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。