javascript设计模式

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");

console.log( dd );