1. 用对象收编变量
2. 真假对象（简单地复制）

Function a(){

return {

}

}

1. 类this/prototype

Function a (){}

a.getName=function(){}

b = new a

b.getName

Function a (){this.getName = function(){}}

b = new a (复制，造成浪费，于是有了prototype)

b.getName

Function a (){}

a.prototype.getName = function(){}

b = new a

b.getName

1）链式(return this)

1. 函数的祖先，抽象一个统一添加方法的功能方法
2. （不允许Function.prototype.checkEmail=function(){}）
3. 链式
4. 类式调用（return this）
5. 面向对象编程

封装：

属性和方法封装

1. 闭包，在闭包内部实现一个完整的类然后将其返回。

var Book = (function(){

//

var bookNum = 0;

//

function checkBook(name){

}

//

function \_book(newId,newName,newPrice){

var name,price;

function checkID(id){}

this.getName = function(){console.log(newName)};

this.getPrice = function(){console.log(newPrice)};

this.setName = function(){};

this.setPrice = function(){};

this.id = newId;

this.copy = function(){};

bookNum++

if(bookNum > 2)

throw new Error('only sale 2');

this.setName(name);

this.setPrice(price);

}

\_book.prototype = {

isJSBook: false,

Display:function(){}

};

return \_book;

})();

1. 安全检察长----创建对象的安全模式

Var book = function(){

If(this instanceof book){

This.title = title;

}else{

Return new book()

}

}

Var \_book = book()

继承：

1. 类式继承：instanceof是判断前面的对象是否是后面类（对象）的实例。

Function superClass(){

this.superValue=true;

}

superClass.prototype.getsupervalue = function(){return this.superValue}

Function subClass(){this.subValue=false;}

subClass.prototype = new superClass();

subClass.prototype.getSubValue = function(){return this.subValue;}

Var instance = new subclass();

Instance.getsupervalue();--- true

Instance.getSubValue ();--- false

Instance instanceof superClass true

Instance instanceof subClass true

subClass instanceof superClass false 只是继承并不是实例

缺陷：1）父类中的共有属性如果是引用类型，就会在子类中被所有实例公用。一个子类实例更改子类原型从父类构造函数中继承来的共有属性就会直接影响其他子类。

1. 由于子类实现的继承是靠其原型对父类的实例化实现的，所有没办法向父类传递参数因而也无法对父类构造函数中的属性初始化
2. 构造函数继承

Function superClass(id){

This.books=[];

This.id = id;

}

superClass.prototype.getsupervalue = function(){return this.superValue}

Function subClass(id){

superClass.call(this,id);

}

但是继承无法涉及到prototype，无法继承到getsupervalue

1. 组合继承

Function superClass(id){

This.books=[];

This.id = id;

}

superClass.prototype.getsupervalue = function(){return this.superValue}

Function subClass(id){

superClass.call(this,id);

}

subClass.prototype = new superClass();

subClass.prototype.getTime = function(){};

父类构造函数调用了两遍

//格拉斯.克罗克福德

1、原型式 function inheritObject(o){

Function f(){}

F.prototype = o;

return new F();

}

//目的是为了创建要返回的新的实例化对象-------缺陷父类引用类型的属性被公用

1. 寄生式继承

function createBook(obj){

var o=inheritObject(obj);

o .getName = function(){console.log(name)};

return o;

}

1. 寄生组合式继承 终极

Function inheritprototype(subClass, superClass){

//Var p = inheritobject(superClass.prototype);

Var p;

Function f(){}

F.prototype=superClass.prototype;

P= new f();

P.constructor = subclass;

Subclass.prototype = p;

}

Function superClass(id){

This.books=[];

This.id = id;

}

superClass.prototype.getsupervalue = function(){return this.superValue}

Function subClass(id){

superClass.call(this,id);//否则无法传参；

}

Inheritprototype(subClass, superClass);

subClass.prototype.getTime=function(){}

//注意：子类再想添加原型方法必须通过proptype.的方式否则会将父类原型继承的对象覆盖掉

多继承 ：

Extend方法//此方法比较浅复制

Var mix = function(){

Var i=1, len=argument.length, target = arguments[0], arg;

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

Arg = arguments[i];

For(var prototype in arg){

Target[prototype] = arg[prototype];

}

}

Return target;

}

深拷贝：

Var mix = function(){

Var i=1, len=argument.length, target = arguments[0], arg;

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

Arg = arguments[i];

For(var prototype in arg){

If(typeof arg[prototype] === “object”){

Target[prototype] = arg[prototype].constructor===”Array”?[]:{};

mix(Target[prototype],arg[prototype]);

}else{

Target[prototype] = arg[prototype];

}

}

}

Return target;

}

多态：

对象继承 类继承

1. 简单工厂模式 -----创建型模式

1、 Function factory(type){

Switch(Type){

Case ‘alert’:

return new LoginAlert();

Case ‘confirm:

return new LoginConfirm();

}

}

通过类实例化对象创建，如果继承一个父类则父类原型上的方法是可以公用的

2、function creatBook(type, text){

Var o = new Object()

O.content = text;

O.show = function(){}

If(type==’alert’){

}

return O;

}

通过创建一个新对象然后包装增强其属性和功能实现，每一个都是新的个体方法不能共用

1. 工厂方法模式 -----创建型模式

安全模式---安全工厂方法

Var factory = function(type, content){

If(this instanceof factory){

Var s = new this[type](content);

Return s;

}else{

Return new Factory(type, content);

}

}

Factory.prototype={

Java: function(){},

Ui: function(){}

}

1. 抽象工厂模式 -----创建型模式

产品类簇的创建，而不负责创建某一类产品的实例，它定制类的结构。

简单工厂创建单一对象，工厂方法模式创建多类对象，抽象工厂模式抽象化创建与虚拟

Var VehicleFactory = fuction(subtype,supertype){

If(typeof VehicleFactory[supertype] === ‘function’){

Inheritprototype(subtype,supertype);

}else{

Throw new Error(‘未创建该抽象类’)；

}

}

VehicleFactory.car = function(){this.type=”car”}

VehicleFactory.car.prototype = {getPrice:function(){ return new Error(‘抽象方法不能调用’)}}

Var lamborghini=function(price, speed){

This.price = price;

This.speed = speed;

}

VehicleFactory(lamborghini, ‘car’)

Lamborghini.type car

1. 建造者模式 -----创建型模式

关心创建过程，因此通常将创建对象的类模块化，使被创建的类的每一个模块都可以得到灵活的运用和高质量复用。

function human(){this.skill = param && param.skill || ‘保密’}

Human.prototype = {

Getskill: function (){return this.skill}

}

Function name(name){

Var that = this;

(function(name, that){

That.wholename = name

If(name.indexof(‘ ’) >-1){

That.firstname = name.slice(0, name.indexof(‘ ’));

That.secondname = name.slice(name.indexof(‘ ’));

}

})(name,that);

}

Var person = function(name, work){

Var \_person = new Human();

\_person.name = new Named(name);

\_person.work = new work(work);

Return \_person

}

1. 原型模式 -----创建型模式

将可复用的、可共享的、耗时大的从基类中提出来然后放在其原型中，然后子类通过组合继承或者寄生组合式继承将方法和属性继承下来。

var LoopImages = function(imgArr, container){

this.imagesArray = imgArr;

this.container = container;

}

LoopImages .prototype = {

createImage: function(){},

changeImage: function(){}

}

Var FadeLoopImg = function(imgArr, container, arrow){

LoopImages.call(this,imgArr,container);

This.arrow = arrow;

}

FadeLoopImg.prototype = new LoopImages();

FadeLoopImg.prototype.changeImage = function(){

Console.log(“eee”);

}

Var slideLoopImg = function(imgArr, container,){

LoopImages.call(this,imgArr,container);

}

slideLoopImg.prototype = new LoopImages();

slideLoopImg.prototype.changeImage = function(){

Console.log(“aaa”);

}

原型对象是一个共享的对象，对原型对象的拓展不管是子类或者父类的实例都会继承下来。

如果创建一个实例对象的构造函数比较复杂或者耗时比较长或者通过创建多对象来实现，最好用复制的方式

1. 单例模式-----创建型模式

用一个对象规划一个命名空间，井井有条管理对象的属性和方法。

模块分明

Baidu.dom.addclass;

Baidu.dim.append;

Baidu.event.trim;

Baidu.event.preventdefault;

模拟无法修改的静态变量

Var conf = (function(){

Var conf = {

MAX\_NUM: 100,

MIN\_NUM: 1,

COUNT:1000

};

Return {

Get: function(name){

Return conf[name]?conf[name]:null;

}

}

})()

惰性单例-----延迟创建

Var lazySingle = (function(){

Var \_instance = null;

Function single(){

Return {

publicMethod: function(){},

publicProperty: ‘1.0’

}

}

Return function()(

If(!\_instance){\_instance = Single();}

Return \_instance;

}

})();

套餐服务---外观模式 结构型设计模式

Onclick是DOM0级事件，如果重复定义方法会将之前定义的click覆盖掉；所以应该使用**addEventListner attachEvent是DOM2级事件处理程序提供的方法**

获取css样式

Var A={

Css: function(ele){

var style = null;

i**f(window.getComputedStyle) {**

**style = window.getComputedStyle(ele, null);**

**}else{**

**style = ele.currentStyle;**

**}**

return style;

}

}

1. 适配器模式

1、框架之间的适配 比较近的 window.A = A =jQuery

比较远 改写

2、参数适配器 function dosomething(obj){

Var \_adapter = {

Name: ‘雨夜清荷’,

Title: ‘ss’,

Size: ‘22’,

Price: 50

}

For(var i in \_adapter){

\_adapter[i] = obj[i] || \_adpter[i];

}

//extend(\_adapter, obj)注：此时可能会多添加属性

}

1. 数据适配 数组数据结构语义不好，可将其适配成对象模式

4、服务器端数据适配 将后端数据适配成我们可用的数据再使用

1. 代理模式

跨域： img jsonp 代理模板（iframe） postmessage document.domain window.name

1. 装饰者模式

var decorator = function(input, fn){

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

if(typeof input.onclick === 'function'){

var oldclickfun = input.onclick;

input.onclick = function(){

oldclickfun();

fn();

}

}else{

input.onclick = fn();

}

}

1. 桥接模式

1、提取共同点

2、事件与业务逻辑之间的桥梁

Dom.onmouseout = function(){changColor(this, ‘#333’, ‘#555’)}

3、多元化对象

Function speed(x, y){

This.x = x;

This.y = y;

}

Speed.prototype.run = function(){}

Function color(cl){

This.color = cl;

}

color.prototype.draw= function(){}

Function ball (x, y, c){

**This.speed = new speed(x, y, c);**

**This.clolor = new color(c);**

}

Ball.prototype.init = function(){

This.speed.run();

This.color.draw();

}

1. 组合模式

**单个对象 组合对象**

**var base = function(){**

**this.children = [];**

**this.element = null;**

**}**

**base.prototype = {**

**init: function(){**

**throw newError("请重写你的方法");**

**},**

**add: function(){**

**throw newError("请重写你的方法");**

**},**

**getElement: function(){**

**throw newError("请重写你的方法");**

**}**

**}**

var formItem = function(id, parent){

base.call(this);

this.parent = parent;

this.id = id;

this.init();

}

inheritPrototype(formItem, base);

formItem.prototype.init = function(){

this.element = document.createElement('div');

this.element.id = this.id;

}

formItem.prototype.add = function(child){

this.children.push(child);

this.element.appendChild(child.getElement());

return this;

}

formItem.prototype.getElement = function(){

return this.element;

}

formItem.prototype.show = function(){

this.parent.appendChild(this.element);

}

var formsetItem = function(classname, name){

base.call(this);

this.name = name;

this.classname = classname;

this.init();

}

inheritPrototype(formsetItem, base);

formsetItem.prototype.init = function(){

this.element = document.createElement("div");

this.element.innerHTML = this.name;

this.classname = "formset "+this.classname;

}

formsetItem.prototype.add = function(child){

this.element.appendChild(child.getElement());

this.children.push(child);

return this;

}

formsetItem.prototype.getElement = function(){

return this.element;

}

var Group = function(){

base.call(this);

this.init();

}

inheritPrototype(Group, base);

Group.prototype.init = function(){

this.element = document.createElement("div");

}

Group.prototype.add = function(child){

this.element.appendChild(child.getElement());

this.children.push(child);

return this;

}

Group.prototype.getElement = function(){

return this.element;

}

var labelItem = function(text){

base.call(this);

this.text = text || "";

this.init();

}

inheritPrototype(labelItem, base);

labelItem.prototype.init = function(){

this.element = document.createElement("span");

this.element.innerHTML= this.text;

}

labelItem.prototype.add = function(){

}

labelItem.prototype.getElement = function(){

return this.element;

}

var form = new formItem('formitem', document.body);

form.add(new formsetItem('account', '账号').add(

new Group().add(

new labelItem('账户1')

).add(

new labelItem('账户2')

)

).add(

new Group().add(

new labelItem('密码1')

).add(

new labelItem('密码2')

)

)).show();

1. 享元模式

将共有数据与方法提取以提高效率。

**行为型设计模式**

1. 模板方法模式

*继承基类*

模板方法的核心在于对方法的重用，它将核心方法封装在基类中，让子类继承基类的方法，实现基类方法的共享。

Alert.call(this,data);

rightAlert.prototype = new Alert();

**Alert.prototype.init.call(this);**

1. 观察者模式

解决类和对象之间的耦合，解耦两个相互依赖的对象，使其依赖于观察者的消息机制。

**事件监听： 绑定，触发，解除**

**适配**

function EventEmitter() {

this.handlers = {};

}

EventEmitter.prototype = {

constructor: EventEmitter,

on: function(type, handler) {

*if (typeof this.handlers[type] === "undefined") {*

this.handlers[type] = [];

}

this.handlers[type].push(handler);

},

emit: function(event) {

var i;

var len;

i*f (!event.target) {*

event.target = this;

}

*if (this.handlers[event.type] instanceof Array) {*

var handlers = this.handlers[event.type];

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

handlers[i](event);

}

}

},

off: function(type, handler) {

var i;

var len;

if (!type && !handler) {

this.handlers = {};

}

if (type && !handler) {

delete this.handlers[type];

}

if (this.handlers[type] instanceof Array) {

var handlers = this.handlers[type];

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

if (handlers[i] === handler) {

break;

}

}

**handlers.splice(i, 1);**

}

}

}; // InheritsUtil.js

return EventEmitter;

1. 超级玛丽---状态模式

当一个对象的内部状态发生改变时，会导致其行为的改变，这看起来像是改变了对象。

案例：美图评选结果

超级玛丽

Var marrystate = function(){

Function show(result){

States[“state”+result]&&States[“state”+result]();

}

Retrun:{

Show: show

}

}

Resultstate.show()

1. 策略模式

将定义的一组算法封装起来，使其相互之间可以替换

**价格：price\*100\*80/10000**

第二十章 职责链模式

解决请求发送者与接受者之间的耦合，通过职责链上的多个对象对分解请求流程，实现请求在多个对象之间的传递，直到最后一个对象完成处理请求。

例：从服务器拉取新闻数据，适配并分析新闻数据，创建新闻模块，为新闻模块添加交互，从页面中展现新闻模块。

**XHR**

第二十一章 命令模式

将请求和实现解耦并封装成独立对象，从而使不同的请求对客户端的实现 参数化。

**Callee**

**Caller**

**判断array类型：**

**Object.prototype.toString.call(): [object Array] [object object]**

**Length**

命令接口

excute: function (msg){

if(!msg){

return;

}

if(msg.length){

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

arguments.callee(msg[i]);

}else{

msg.param = Object.prototype.toString.call(msg.param) === "[object Array]"? msg.param: [msg.param];

Action[msg.command].apply(Action, msg.param);

}

}

第二十二章 访问者模式

针对于对象结构中的元素，定义在不改变该对象的前提下访问结构中元素的新方法。

Ie中的dom2级事件为元素添加行为时，attachEvent方法this=window

**Array.prototype.slice.call(arguments,0)类数组对象转换成一个真正的数组**

Function bindIEEvent(dom, type, fn, data){

Var data = data||{};

dom.attachEvent(“on”+type, function(e){

Fn.call(dom, e, data);

})

}

第二十三章中介者模式

Var Mediator: funtion(){

var \_\_message = {};

return {

regist: function(type , fn){

if(typeof \_\_message[type] === 'undefind'){

\_\_message[type] = [fn];

}else{

\_\_message[type].push(fn);

}

},

send: function(type){

if(\_\_message[type]){

Var i=0,

len = \_\_message[type].length;

for(;i<len;i++){

\_\_message[type][i]&&\_\_message[type][i](); }

}

}

}

第二十四章 备忘录模式

在不破坏对象的封装性的前提下，在对象之外捕获并保存该对象内部的状态以便日后对象使用或者对象恢复到以前的某个状态。

第二十五章 迭代器模式

顺序访问一个聚合对象中的每个元素，简化代码中的循环语句 类数组

第二十六章 解释器模式

对客户提出的需求，经过解析而形成一个抽象解释程序

第五篇技巧性设计模式

第二十七章 链模式

通过在对象中的每个方法调用执行完毕后返回当前对象 this 来实现。

第二十八章 委托模式

多个对象接收并处理同一请求，他们将请求委托给另一个对象统一处理请求。

第二十九章 数据管理器 数据访问对象 DAO//

抽象和封装对数据源的访问和存储，DAO通过对数据源链接的管理，方便对数据的访问与存储

localStorage、cookie、 userdata

Nodejs 的mongoDB

第三十章 节流模式

对重复的业务逻辑进行节流控制，执行最后一次操作并取消其他操作，以提高性能。

节流器

回到顶部icon

鼠标滑过

图片延迟加载

第三十一章 卡片拼图---简单模板模式

通过格式化字符串拼凑出视图避免创建视图时大量节点操作。 优化内存开销。

第三十二章 惰性模式

减少每次代码执行时的重复性的分支判断，通过对对象重定义来屏蔽原对象中的分支判断。

重复判断事件绑定方法

解决方案 ：对方法进行重写，1、加载时重写2、第一次执行时写

第三十三章 异国战场---参与者模式

参与者：在特定的作用域中执行给定的函数，并将参数原封不动的传递。

第三十四章 等待者模式

Promise settimeout deferred

第六篇 架构型设计模式

框架结构

第三十五章 同步模块模式

第三十五章 异步模块模式

requirejs