面向对象…………。

JS面向对象的实现

工厂模式：JS中无法创建类，因此用函数封装以特定接口创建对象，其实现方法非常简单，就是在函数内创建一个对象，给对象赋予属性、方法再将对象返回即可。

function CreateObject(name,url){

var Obj = new Object();

Obj.name = name;

Obj.url = url;

Obj.SayUrl = function(){

console.log(this.url);

};

return Obj;

}

var Cobj = CreateObject(“pschen”,”http://www.pschen.cn”);

构造函数模式：JS中构造函数可以创建特定类型的对象，类似于Array、Date等原生JS的对象

function Blog(name.url){

this.name = name;

this.url = url;

this.SayUrl = function(){

console.log(this.url);

};

}

val blog1 = new Blog(‘pschen’,’http://www.pschen.cn’);

val blog2 = new Blog(‘ellen,’http://www.ellen.cn’);

console.log(blog instaceof Blog); //判断blog是否是Blog的实例

blog.SayUrl();

和工厂模式不同

1、没有显式地创建对象

2、直接将属性和方法赋值给了this对象

3、没有return语句

要创建Blog的新实例，必须使用new操作符，以这种方式调用构造函数分4步：

1、创建一个新对象

2、将构造函数的作用域赋给新对象（因此this就指向了这个新对象）

3、执行构造函数中的代码（为新对象添加属性）

4、返回新对象

将构造函数当作函数

构造函数与其他函数的唯一区别，就在于调用它们的方式不同，任何函数只要通过new操作符来调用，那它就可以作为构造函数。

val blog2 = new Blog(‘ellen,’http://www.ellen.cn’);

blog2.SayUrl();

blog3(“AAAA”,”BBBB”); //添加到window

window.SayUrl();

var obj = new Object();

Blog.call(obj,”XXXX”,”YYYY”);

obj.SayUrl();

构造函数问题：使用构造函数的主要问题，就是每个方法都要在每个实例上重新创建一遍，前面blog1,blog2都有一个名为SayUrl()的方法，但这两个方法不是同一个Function的实例

function Blog(name.url){

this.name = name;

this.url = url;

this.SayUrl = new function(“console.log(this.url);”);

}

从上语句可以看出每个Blog实例都包含一个不同的Function实例 的本质。

可以把函数定义转移到构造函数外部来解决创建不同的实例问题

function Blog(name.url){

this.name = name;

this.url = url;

this.SayUrl = SayUrl;

}

function SayUrl(){ //全局的函数

console.log(this.url);

}

新的问题：全局作用域中定义的函数实际上只能被某个对象调用，更问题的是，如果对象需要定义很多方法，那么就要定义很多个全局函数，就没有什么封装性可言。

原型模式：

我们创建的每个函数都有一个prototype（原型）属性，这个属性是一个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。prototype就是通过调用构造函数而创建的那个对象实例的原型对象。使用原型对象的好处是可以让所有对象实例共享它所包含的属性和方法。不必在构造函数中定义对象实例的信息，而是可以将这些信息直接添加到原型对象中。

function Person(){

}

Person.prototype.name = “”;

Person.prototype.age = 40

Person.prototype.job = “network Engineer”;

Person.prototype.Sayname = function(){

console.log(this.name);

};

var person1 = new Person();

person1.SayName();

var person2 = new Person();

person2.SayName();

console.log(person1.SayName == person2.SayName); //true

理解原型对象

只要创建了一个新函数，就会根据一组特定的规则为该函数创建一个prototype属性，这个属性指向函数的原型对象，默认情况下，所有原型对象都会自动获得一个constructor属性，这个属性包含一个指向prototype属性所在函数的指针。

创建自定义的构造函数之后，其原型对象默认只会取得constructor属性，其他方法则都是从object继承而来，当调用构造函数创建一个新实例后，该实例内部将包含一个指针（\_\_proto\_\_），指向构造函数的原型对象

混合模式（原型+构造）：混合模式中构造函数模式用于定义实例属性，而原型模式用于定义方法和共享属性。每个实例都会有自己的一份实例属性，但同时又共享着方法，最大限度的节省了内存。另外这种模式还支持传递初始参数。优点甚多。这种模式在ECMAScript中是使用最广泛、认同度最高的一种创建自定义对象的方法。

function Blog(name,url,friend){

this.name = name;

this.url = url;

this.friend = friend;

}

Blog.prototype.SayInfo = function(){

console.log(this.name + this.url + this.friend);

};

var blog1 = new Blog(“pschen”,”http://www.pschen.cn”,[‘ellen’,’alon’]);

var blog2 = new Blog(“ellen”,”http://www.ellen.cn”,[‘pschen,’alon’] );

动态原型模式：动态原型模式将所有信息封装在了构造函数中，而通过构造函数中初始化原型（仅第一个对象实例化时初始化原型），这个可以通过判断该方法是否有效而选择是否需要初始化原型。

function Blog(name,url,friend){

this.name = name;

this.url = url;

if(typeof this.SayInfo != ‘function’){

console.log(“one time”);

Blog.prototype.SayInfo = function(){

console.log(this.name + this.url);

};

}

}

var blog1 = new Blog(“pschen”,”http://www.pschen.cn”);

var blog2 = new Blog(“ellen”,”http://www.pschen.cn” );