面向对象程序设计

JS中没有类的概念，因此它的对象也与基于类的语言中的对象有所不同。

对象定义：无序属性的集合，其属性可以包含基本值、对象、函数。对象是一组没有特定顺序的值，对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。可以把对象想象成散列表：无非就是一组名值对，其他值可以是数据或函数。

每个对象都是基于一个引用类型创建的，这个引用类型可以是原生类型，也可以是开发人员定义的类型。

JS一般使用原型对象来实现属性和方法的共享（继承）

对象的创建方式：

常规方式：new

var person = new Object();

person.name = “pschen”;

person.age =40;

person.job = “software engineer”;

person.say() = function(){

console.log(this.name + this.age + this.job);

}

对象字面量方式：

var person = {

name: “pschen”,

age:40,

job: “software engineer”,

\_sex:”女”, //前下划线是一种常用的记号，用于表示只能通过对象访问的属性

say: function(){

console.log(this.name + this.age + this.job);

}

};

创建对象：虽然Object构造函数和对象字面量都可以用来创建单个对象，但使用同一个接口创建很多对象，会产生大量的重复代码，为解决这个问题，人们开始使用工厂模式的一种变体。

工厂模式：是软件工程领域一种广为人知的设计模式，这种模式抽象了创建具体对象的过程，JS用函数来封装以特定接口创建对象的细节。

function createperson(name,age,job){

var obj = new Object();

obj.name = name;

obj.age = age;

obj.job = job;

obj.say = function(){

console.log(this.name);

};

return obj;

}

var p1 = createperson(“XXX”,40,”YYY”);

var p2 = createperson(“ZZZ”,40,”SSS”);

每次调用这个函数都会返回一个包含三个属性一个方法的对象。

构造函数模式

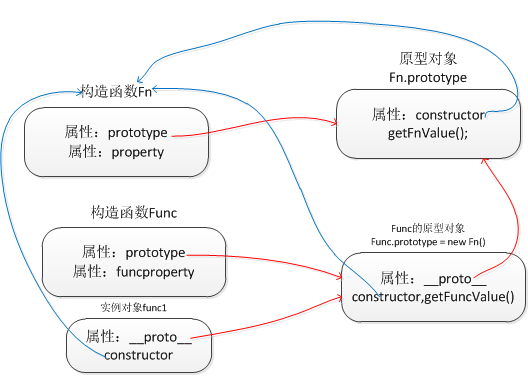
原型模式

继承

继承是OO语言中的一个最为人津津乐道的概念，两种继承方式：接口继承和实现继承

接口继承只继承方法签名，实现继承则继承实际的方法，由于函数没有签名，JS中无法实现接口继承，JS中其实现继承主要是依靠原型链来实现的

原型链：利用原型让一个引用类型继承另一个引用类型的属性和方法。每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针。如果让原型对象等于另一个类型的实例，显然此时的原型对象将包含一个指向另一个原型的指针，相应地另一个原型中也包含一个指向另一个构造函数的指针。如果另一个原型又是另一个类型的实例，那么上述关系依然成立，层层递进，构成了实例与原型的链条。



func1.constructor == Fn

function Fn(){

this.property = true;

}

Fn.prototype.getFnValue = function(){

return this.property;

};

function Func(){

this.funcproperty = false;

}

Func.prototype = new Fn(); //var temp = new Fn(); Func.prototype = temp;重写原型对象

Func.prototype.getFuncValue = function(){

return this.funcproperty;

};

var func1 = new Func();

console.log(func1.getFnValue());

console.log(func1.\_\_proto\_\_); //Fn函数的实例

Fn

getFuncValue: ()

property: true

\_\_proto\_\_: Object

console.log(func1.constructor); //怎么指向的是Fn, func1.constructor最终被Fn. prototype.constructor覆盖

function Fn(){

this.property = true;

}

console.log(Func.prototype.\_\_proto\_\_); //指向Fn的原型对象

Object

constructor: Fn()

getFnValue: ()

\_\_proto\_\_: Object

console.log(Func.prototype.constructor); //实例指向构造函数Fn

function Fn(){

this.property = true;

}

console.log(Func.prototype.getFuncValue);

function (){

return this.funcproperty;

}

确定原型和实例的关系

instanceof 测试实例与原型链中出现过的构造函数

console.log(func1 instanceof Object); //true

console.log(func1 instanceof Fn);

console.log(func1 instanceof Func);

isPrototypeOf(),在原型链中出现过的原型，都可以说是该原型链所派生的实例的原型

即在通过原型链实现继承时，不能使用对象字面量创建原型方法，因为这样就会重写原型链

…………

Func.prototype = new Fn();

Func.prototype = { //重写原型，原型指针指向新的对象，原型对象变为Object

};

原型链的问题：每个实例都会共享超类型的属性，任何一个实例修改超类型的属性时都会对其他实例造成影响（引用类型的属性有影响）

function Fn(){

this.color = ["red","blue","green"];

}

function Func(){

}

Func.prototype = new Fn();

var func1 = new Func();

func1.color.push("black");

console.log(func1.color); // ["red", "blue", "green", "black"]

var func2 = new Func();

console.log(func2.color); // ["red", "blue", "green", "black"]

借用构造函数（伪造对象、经典继承）基本思想：在子类型构造函数的内部调用超类型构造函数，函数只不过是在特定环境中执行代码的对象，通过使用apply()、call()方法也可在新创建的对象上执行构造函数

function Fn(){

this.color = ["red","blue","green"];

}

function Func(){

Fn.call(this);

}

var func1 = new Func();

查看对象中的属性方法

var i;

for(i in func2){

console.log(i);

console.log(func2[i]);

}

color

(3) ["red", "blue", "green"]

func1.color.push("black");

console.log(func1.color); // ["red", "blue", "green", "black"]

var func2 = new Func();

console.log(func2.color); // ["red", "blue", "green"]

Fn.call(this)借调了另一个构造函数，在创建Func实例的环境下调用Fn构造函数，这样每一个新的Func实例对象上执行Fn函数中定义的所有对象初始化代码，每个实例对象都有一个自己的color属性副本

传递参数

相对于原型链，借用构造函数有一个很大的优势，即可以在子类型构造函数中向超类型构造函数传递参数

function Fn(name){

this.name = name;

}

function Func(){

Fn.call(this, "black");

this.age = 40; //实例对象属性

}

var func1 = new Func();

console.log(func1.name);

console.log(func1.age);

组合继承：伪经典继承。指的是将原型链和借用构造函数的技术组合到一块，从而发挥二者之长的一种继承模式，基本思想是使用原型链实现对原型属性和方法的继承。而通过借用构造函数来实现对实例属性的继承，这样既通过在原型上定义方法实现了函数利用，又能够保证每个实例都有它自己的属性

function Fn(name){

this.name = name;

this.color = ["red","blue","green"];

}

Fn.prototype.sayName = function(){

console.log(this.name);

};

function Func(name,age){

Fn.call(this,name);

this.age = age;

}

Func.prototype = new Fn(); //Func的原型对象为Fn的实例对象，继承了Fn的方法和属性

Func.prototype.sayAge = function(){

console.log(this.age);

};

var func1 = new Func("pschen",40);

func1.color.push("black");

console.log(func1.color); // ["red", "blue", "green", "black"]

func1.sayName();

func1.sayAge();

var func2 = new Func("ellen",36);

console.log(func2.color); // ["red", "blue", "green"]

func2.sayName();

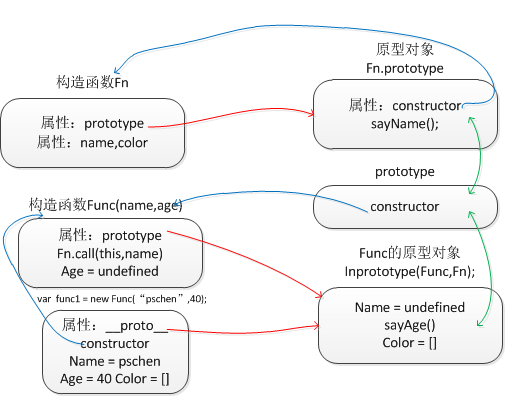
func2.sayAge();

寄生组合继承：通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。基本思想是不必为了指定子类型的原型而调用超类型的构造函数，我们所需要的无非就是超类型原型的一个副本，本质上就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型

function inprototype(Fun1,Fun2){ //Object(对象); 返回原对象，复制参数(对象)的指针

var prototype = Object(Fun2.prototype); //创建参数2原型对象的副本

prototype.constructor = Fun1; //为副本对象添加constructor属性，指向参数1函数



Fun1.prototype = prototype; //将新创建的副本对象赋值给参数1的原型对象

}

function Fn(name){

this.name = name;

this.color = ["red","blue","green"];

}

Fn.prototype.sayName = function(){

console.log(this.name);

};

function Func(name,age){

Fn.call(this,name);

this.age = age;

}

inprototype(Func,Fn);

Func.prototype.sayAge = function(){

console.log(this.age);

};

var func1 = new Func("pschen",40);

func1.color.push("black");

console.log(func1.color); // ["red", "blue", "green", "black"]

func1.sayName();

func1.sayAge();

var func2 = new Func("ellen",36);

console.log(func2.color); // ["red", "blue", "green"]

func2.sayName();

func2.sayAge();

函数表达式

定义函数的两种方式

函数声明

function functionname(arg0,arg1,arg2){

//函数体

}

关于函数声明它的一个重要特征就是函数声明提升，在执行代码之前会先读取函数声明并赋值

sayHi(();

function sayHi(){

//

}

这例子不会抛出错误

函数表达式

var functionname = function(arg0,arg1,arg2){

//函数体

};

创建一个函数并将它赋值给变量，这种情况下创建的函数叫匿名函数

函数表达式与其他表达式一样，都要在使用之前先赋值

sayHi();

var sayHi = function(){

//函数体

};

这个会抛出错误，提示函数不存在

能够创建函数再赋值给变量，也就能够把函数作为其他函数的值返回

递归：递归函数是在一个函数通过名字调用自身的情况下构成的

function func(num){

if( num<= 1){

return 1;

}else{

return num \* func(num-1);

}

}

func(5); //5\*4\*3\*2\*1

5\*func(4) 5\*4\*func(3) 5\*4\*3\*func(2) 5\*4\*3\*2\*func(1)

argument.callee是一个指向正在执行的函数的指针，在编写递归函数时，使用它总比使用函数名更保险，但在严格模式下不能通过脚本访问

function func(num){

if( num<= 1){

return 1;

}else{

return num \* argument.callee (num-1);

}

}

可以使用命名函数表达式来达成相同的递归结果

var func1 = (function f (num){

if( num<= 1){

return 1;

}else{

return num \* f (num-1);

}

});

创建一个名为f()的命名函数表达式，然后将它赋值给变量func1,把函数赋值给另一个变量，函数的名字f仍然有效，递归调用还是能正确完成，在严格和非严格下都可以执行

闭包：指有权访问另一个函数作用域中的变量的函数。首先它是一个函数，其次它可以访问另一个函数内的变量

要理解闭包必须从理解函数第一次被调用的时候都会发生什么入手，理解作用域链

当某个函数第一次被调用时，会创建一个执行环境及相应的作用域链，并把作用域链值给一个特殊的内部属性（[Scope]），然后使用this,arguments和其他命名参数的值来初始化函数的活动对象（activation object），在作用域链中，外部函数的活动对象始终处于第二位，外部函数的外部函数的活动对象处于第三位……直至作为作用域链终点的全局执行环境

函数执行过程中，为读取和写入变量的值，就需要在作用域中查找变量

function func(value1,value2){

if(value1 < value2){

return -1;

} else if(value1 > value2){

return 1;

} else {

return 0;

}

}

var result = compare(5,10);

先定义了func()函数，然后又在全局作用域中调用它，当第一次调用时，会创建一个包含this/arguments/value1/value2的活动对象

后台在每个执行环境都有一个表示变量的对象——变量对象，全局环境的变量对象始终存在，而函数的局部环境的变量对象，则只在函数执行的过程中存在，创建func()函数时，会创建一个预告包含全局变量对象的作用域链，这个作用域链保存在内部的[scope]属性中

作用域链本质上是一个指向变量对象指针列表，它只引用但不实际包含变量对象

一般来讲，当函数执行完毕后，局部活动对象就会被销毁，内存中仅保存全局作用域，但闭包的情况又有所不同

在另一个函数内部定义的函数会将包含函数（即外部函数）的活动对象添加到它的作用域链中。

在后台执行环境中，闭包的作用域链包含着它自己的作用域、包含函数的作用域和全局作用域。

关于this对象

匿名函数的执行环境具有全局性，因此其this对象通常指向window。但有时候由于编写闭包的方式不同，这点可能不会很明显

var name = "The Window";

var object = {

　　name : "My Object",

　　getNameFunc : function(){

　　　　return function(){

　　　　　　return this.name;

　　　　};

　　}

　};

console.log(object.getNameFunc()());

每个函数在被调用时，其活动对象都会自动取得两个特殊变量：this、argument。内部函数在搜索这两个变量时，只会搜索到其活动对象为止，永远不可能直接访问外部函数中的这两个变量

要将函数声明转换成函数表达式，只要给他加上一对圆括号

var temp = function(){};

temp();

function(){}();//报错

(function(){})();

私有变量：函数的参数，局部变量，在函数内部定义的其他函数

funct func(num1,num2){

sum = num1 + num2;

}

创建用于访问私有变量的公有方法（利用闭包），把有权访问私有变量和私有函数的公有方法称为特权方法

一是在构造函数中定义特权方法

function MyObject(){

var privatevariable = 10;

function privatefunction() {

return false;

}

this.publicmethod = function(){

privatevariable++;

return privatefunction();

};

}

构造函数中定义特权方法缺点：针对每个实例都会创建同样一组新方法

静态私有变量

通过在私有作用域中定义私有变量或函数也可以创建特权方法

(fcuntion(){

var privatevariable = 10;

function privatefunction() {

return false;

}

MyObject = function(){}; //构造函数

MyObject.prototype.publicmethod = function(){ //在函数的原型定义方法

privatevariable++;

return privatefunction();

};

})();

定义构造函数时没有使用函数声明，也没有使用var，所有MyObject是一个全局变量