

# javascript 面向对象技术基础(一)

看了很多介绍 javascript 面向对象技术的文章,很晕.为什么?不是因为写得不好,而是因为太深奥.

javascript 中的对象还没解释清楚怎么回事,一上来就直奔主题,类/继承/原型/私有变量....

结果呢,看了大半天,有了一个大概的了解,细细一回味,好像什么都没懂...

这篇文章是参考<<javascript-the definitive guide,5th edition>>第7,8,9章而写成的,我也

会尽量按照原书的结构来说明 javascript 的面向对象技术(对象/数组->函数-->类/构造函数/原型).对一些我自己也拿捏不准的地方,我会附上原文的英文语句,供大家参考.

如果不做说明,则文中出现的所有英文语句(程序体除外)都是引自<<javascript-the definitive guide,5th edition>>.

-----  
对象和数组(Objects and Arrays)

什么是对象?把一些"名字-属性"的组合放在一个单元里面,就组成了一个对象.我们可以理解为 javascript 中

的对象就是一些"键-值"对的集合(An object is a collection of named values. These named values are usually referred to as properties of the object.--Section3.5).

"名字"只能是 string 类型,不能是其他类型,而属性的类型则是

任意的(数字/字符串/其他对象..).可以用 new Object() 来创建一个空对象,也可以简单的用"{}"来创建一个

空对象,这两者的作用是等同的.

Js 代码

```
1 var emptyObject1 = {};           //创建空对象
2 var emptyObject2 = new Object(); //创建空对象
3 var person = {"name":"sdcyst",
4               "age":18,
5               "sex":"male"};      //创建一个包含初始值的对象 person
6 alert(person.name);               //sdcyst
7 alert(person["age"]);              //18
```

从上面的例子我们也可以看到,访问一个对象的属性,可以简单的用对象名加"."后加属性的名字,也

可以用"[]"操作符来获取,此时在[]里面的属性名字要加引号,这是因为对象中的索引都是字符串类型的.

javascript 对象中属性的个数是可变的,在创建了一个对象之后可以随时对它赋予任何的属性.

Js 代码

```
8  var person = {};  
9  person.name = "sdcyst";  
10 person["age"] = 18;  
11 alert(person.name + "__" + person.age); //sdcyst__18  
12  
13 var _person = {name:"balala","age":23}; //在构建一个对象时,属性的  
    名字可以不用引号来标注(name),  
14                                     //但是仍旧是一个字符串类型.在访问的时候[]内仍旧需要  
    引号  
15 alert(_person["name"] + "__" + person.age); //balala__23  
16 alert(_person[name]); //undefined
```

通过"."操作符获取对象的属性,必须得知道属性的名字.一般来说"[]"操作符获取对象属性的功能更强大一些,  
可以在[]中放入一些表达式来取属性的值,  
比如可以用在循环控制语句中,而"."操作符则没有这种灵活性。

Js 代码

```
17 var                                     name                                     =  
    {"name1":"NAME1","name2":"NAME2","name3":"NAME3","name4":"NAME4"  
    "};  
18 var namestring = "";  
19 for(var props in name) { //循环 name 对象中的属性名字  
20     namestring += name[props];  
21 }  
22 alert(namestring); //NAME1NAME2NAME3NAME4  
23  
24 namestring = "";  
25 for(var i=0; i<4; i++) {  
26     namestring += name["name"+(i+1)];  
27 }  
28 alert(namestring); //NAME1NAME2NAME3NAME4
```

delete 操作符可以删除对象中的某个属性,判断某个属性是否存在可以使用"in"操作符.

Js 代码

```
29 var                                     name                                     =  
    {"name1":"NAME1","name2":"NAME2","name3":"NAME3","name4":"NAME4"
```

```

    "};
30 var namestring = "";
31 for(var props in name) { //循环 name 对象中的属性名字
32     namestring += name[props];
33 }
34 alert(namestring); //NAME1NAME2NAME3NAME4
35
36 delete name.name1; //删除 name1 属性
37 delete name["name3"]; //删除 name3 属性
38 namestring = "";
39 for(var props in name) { //循环 name 对象中的属性名字
40     namestring += name[props];
41 }
42 alert(namestring); //NAME2NAME4
43
44 alert("name1" in name); //false
45 alert("name4" in name); //true

```

需要注意,对象中的属性是没有顺序的.

对象的 constructor 属性

每一个 javascript 对象都有一个 constructor 属性.这个属性对应了对象初始化时的构造函数(函数也是对象).

Js 代码

```

46 var date = new Date();
47 alert(date.constructor); //Date
48 alert(date.constructor == "Date"); //false
49 alert(date.constructor == Date); //true

```

## javascript 面向对象技术基础(二)

数组

我们已经提到过,对象是无序数据的集合,而数组则是有序数据的集合,数组中的数据(元素)通过索引(从0开始)来访问,数组中的数据可以是任何的数据类型.数组本身仍旧是对象,但是由于数组的很多特性,通常情况下把数组和对象区别

开来分别对待 (Throughout this book, objects and arrays are often treated as distinct datatypes.

This is a useful and reasonable simplification; you can treat objects and arrays as separate types

for most of your JavaScript programming. To fully understand the behavior of objects and arrays, however, you have to know the truth: an array is nothing more than an object with a thin layer of extra functionality. You can see this with the `typeof` operator: applied to an array value, it returns the string "object". --section 7.5).

创建数组可以用 "[]" 操作符, 或者是用 `Array()` 构造函数来 new 一个.

Js 代码

```
1 var array1 = []; //创建空数组
2 var array2 = new Array(); //创建空数组
3 array1 = [1,"s",[3,4],{"name1":"NAME1"}]; //
4 alert(array1[2][1]); //4 访问数组中的数组元素
5 alert(array1[3].name1); //NAME1 访问数组中的对象
6 alert(array1[8]); //undefined
7 array2 = [, ,]; //没有数值填入只有逗号, 则对应索引处的元素为
  undefined
8 alert(array2.length); //3
9 alert(array2[1]); //undefined
```

用 `new Array()` 来创建数组时, 可以指定一个默认的大小, 其中的值此时为 `undefined`, 以后可以再给他们赋值. 但是由于 javascript 中的数组的长度是可以任意改变的, 同时数组中的内容也是可以任意改变的, 因此这个初始化的长度实际上对数组没有任何的约束力. 对于一个数组, 如果对超过它最大长度的索引赋值, 则会改变数组的长度, 同时会对没有赋值的索引处赋值 `undefined`, 看下面的例子.

Js 代码

```
10 var array = new Array(10);
11 alert(array.length); //10
12 alert(array[4]); //undefined
13 array[100] = "100th"; //这个操作会改变数组的长度, 同时将10-99索引
  对应的值设为 undefined
14 alert(array.length); //101
15 alert(array[87]); //undefined
```

可以用 `delete` 操作符删除数组的元素, 注意这个删除仅仅是将数组在该位置的元素设为 `undefined`, 数组的长度并没有改变.

我们已经使用过了数组的 `length` 属性, `length` 属性是一个可以读/写的属性, 也就是说我们可以通过改变数组的 `length` 属性来任意的改变数组的长度. 如果将 `length` 设为小于数组长度的值, 则原数组中索引大于

length-1 的值都会被删除. 如果 length 的值大于原始数组的长度, 则在它们之间的值设为 undefined.

Js 代码

```
16 var array = new Array("n1", "n2", "n3", "n4", "n5"); // 五个元素的数组
17 var astring = "";
18 for(var i=0; i<array.length; i++) { // 循环数组元素
19     astring += array[i];
20 }
21 alert(astring); // n1n2n3n4n5
22 delete array[3]; // 删除数组元素的值
23 alert(array.length + "_" + array[3]) // 5_undefined
24
25 array.length = 3; // 缩减数组的长度
26 alert(array[3]); // undefined
27 array.length = 8; // 扩充数组的长度
28 alert(array[4]); // undefined
```

对于数组的其他方法诸如 join/reverse 等等, 在这就不再一一举例.

通过上面的解释, 我们已经知道, 对象的属性值是通过属性的名字 (字符串类型) 来获取, 而数组的元素是通过索引 (整数型 0~2\*\*32-1) 来得到值. 数组本身也是一个对象, 所以对象属性的操作也完全适合于数组.

Js 代码

```
29 var array = new Array("no1", "no2");
30 array["po"] = "props1";
31 alert(array.length); // 2
32 // 对于数组来说, array[0] 同 array["0"] 效果是一样的 (? 不确定, 测试时如此)
33 alert(array[0] + "_" + array["1"] + "_" + array.po); // no1_no2_props1
```

## javascript 面向对象技术基础 (三)

函数

javascript 函数相信大家都写过不少了, 所以我们这里只是简单介绍一下.

创建函数:

```
function f(x) {.....}
```

```
var f = function(x) {.....}
```

上面这两种形式都可以创建名为 `f()` 的函数,不过后一种形式可以创建匿名函数  
函数定义时可以设置参数,如果传给函数的参数个数不够,则从最左边起依次对应,其余的用 `undefined` 赋值,如果传给函数的参数多于函数定义参数的个数,则多出的参数被忽略。

Js 代码

```
1  function myprint(s1,s2,s3) {  
2      alert(s1+"_"+s2+"_"+s3);  
3  }  
4  myprint();           //undefined_undefined_undefined  
5  myprint("string1","string2"); //string1_string2_undefined  
6  myprint("string1","string2","string3","string4");  
   //string1_string2_string3
```

因此,对于定义好的函数,我们不能指望调用者将所有的参数全部传进来.对于那些必须用到的参数应该在函数体中加以检测(用!操作符),或者设置默认值然后同参数进行或(`||`)操作来取得参数。

Js 代码

```
7  function myprint(s1,person) {  
8      var defaultperson = {    //默认 person 对象  
9          "name":"name1",  
10         "age":18,  
11         "sex":"female"  
12     };  
13     if(!s1) {    //s1不允许为空  
14         alert("s1 must be input!");  
15         return false;  
16     }  
17     person = person || defaultperson; //接受 person 对象参数  
18     alert(s1+"_"+person.name+": "+person.age+": "+person.sex);  
19 };  
20  
21 myprint(); //s1 must be input!  
22 myprint("s1"); //s1_name1:18:female  
23 myprint("s1",{"name":"sdcyst","age":23,"sex":"male"});
```

```
//s1_sdcyst:23:male
```

函数的 arguments 属性

在每一个函数体的内部,都有一个 arguments 标识符,这个标识符代表了一个 Arguments 对象.Arguments 对象非常类似于 Array (数组) 对象,比如都有 length 属性,访问它的值用"[]"操作符利用索引来访问参数值,但是,二者是完全不同的东西,仅仅是表面上有共同点而已(比如说修改 Arguments 对象的 length 属性并不会改变它的长度)。

Js 代码

```
24 function myargs() {
25     alert(arguments.length);
26     alert(arguments[0]);
27 }
28 myargs(); //0 --- undefined
29 myargs("1",[1,2]); //2 --- 1
```

Arguments 对象有一个 callee 属性,标示了当前 Arguments 对象所在的方法.可以使用它来实现匿名函数的内部递归调用。

Js 代码

```
30 function(x) {
31     if (x <= 1) return 1;
32     return x * arguments.callee(x-1);
33 } (section8.2)
```

-----  
-

Method-- 方法

方法就是函数.我们知道,每一个对象都包含 0 个或多个属性,属性可以是任意类型,当然也包括对象.函数本身就是一种对象,因此我们完全可以把一个函数放到一个对象里面,此时,这个函数就成了对象的一个方法.此后如果要使用该方法,则可以通过对象名利用"."操作符来实现。

Js 代码

```
34 var obj = {f0:function(){alert("f0");}}; //对象包含一个方法
```

```

35 function f1() {alert("f1");}
36 obj.f1 = f1;    //为对象添加方法
37
38 obj.f0(); //f0 f0是 obj 的方法
39 obj.f1(); //f1 f1是 obj 的方法
40 f1();        //f1 f1同时又是一个函数,可以直接调用
41 f0();        //f0 仅仅是 obj 的方法,只能通过对象来调用

```

方法的调用需要对象的支持,那么在方法中如何获取对象的属性呢?this!this 关键字我们已经很熟悉了,在 javascript 的方法中,我们可以用 this 来取得对方法调用者(对象)的引用,从而获取方法调用者的各种属性.

Js 代码

```

42 var obj = {"name":"NAME","sex":"female"};
43 obj.print = function() { //为对象添加方法
44     alert(this.name + "_" + this["sex"]);
45 };
46 obj.print(); //NAME_female
47 obj.sex = "male";
48 obj.print(); //NAME_male

```

下面我们来一个更加面向对象的例子.

Js 代码

```

49 var person = {name:"defaultname",
50     setName:function(s){
51         this.name = s;
52     },
53     "printName":function(){
54         alert(this.name);
55     }}
56 person.printName(); //defaultname
57 person.setName("newName");
58 person.printName(); //newName

```



在上面的例子中,完全可以用 `person.name=.` 来直接改变 `person` 的 `name` 属性,在此我们只是为了展示一下刚才提到的内容.

另一种改变 `person` 属性的方法就是:定义一个 `function`, 接收两个参数,一个是 `person`, 一个是 `name` 的值,看起来像是这样:

`changeName(person, "newName")`. 哪种方法好呢?很明显,例子中的方法更形象,更直观一些,而且好像有了那么一点面向对象影子.

再次强调一下,方法 (Method) 本身就是函数 (function), 只不过方法的使用更受限制. 在后面的篇幅中,如果提到函数,那么提到的内容同样适用于方法,反之则不尽然.

函数的 `prototype` 属性

每一个函数都包含了一个 `prototype` (原型) 属性,这个属性构成了 javascript 面向对象的核心基础.在后面我们会详细讨论.

## javascript 面向对象技术基础(四)

类、构造函数、原型

先来说明一点:在上面的内容中提到,每一个函数都包含了一个 `prototype` 属性,这个属性指向了一个 `prototype` 对象(Every

`function has a prototype property that refers to a predefined prototype object` --section8.6.2).注意不要搞混了.

构造函数:

`new` 操作符用来生成一个新的对象.`new` 后面必须要跟上一个函数,也就是我们常说的构造函数.构造函数的工作原理又是怎样的呢?

先看一个例子:

Js 代码

```
1  function Person(name,sex) {
2      this.name = name;
3      this.sex = sex;
4  }
5  var per = new Person("sdcyst","male");
6  alert("name:"+per.name+"_sex:"+per.sex); //name:sdcyst_sex:male
```

下面说明一下这个工作的步骤:

开始创建了一个函数(不是方法,只是一个普通的函数),注意用到了 `this` 关键字.以前我们提到过 `this` 关键字表示调用该方法的对象,也就

是说通过对象调用"方法"的时候,`this` 关键字会指向该对象(不使用对象直接调用该函数则 `this` 指向整个的 `script` 域,或者函数所在的域,在此

我们不做详细的讨论).当我们使用 `new` 操作符时,`javascript` 会先创建一个空的对象,然后这个对象被 `new` 后面的方法(函数)的 `this` 关键字引用!然后在方法中

通过操作 `this`,就给这个新创建的对象相应的赋予了属性.最后返回这个经过处理的对象.这样上面的例子就很清楚:先创建一个空对象,然后

调用 `Person` 方法对其进行赋值,最后返回该对象,我们就得到了一个 `per` 对象.

`prototype`(原型)--在这里会反复提到"原型对象"和"原型属性",注意区分这两个概念.

在 `javascript` 中,每个对象都有一个 `prototype` 属性,这个属性指向了一个 `prototype` 对象.

上面我们提到了用 `new` 来创建一个对象的过程,事实上在这个过程中,当创建了空对象后,`new` 会接着操作刚生成的这个对象的 `prototype` 属性.

每个方法都有一个 `prototype` 属性(因为方法本身也是对象),`new` 操作符生成的新对象的 `prototype` 属性值和构造方法的 `prototype` 属性值是一致的.构造方

法的 `prototype` 属性指向了一个 `prototype` 对象,这个 `prototype` 对象初始只有一个属性 `constructor`,而这个 `constructor` 属性又指向了 `prototype` 属性所在的方法

(In the previous section, I showed that the `new` operator creates a new, empty object and then invokes a constructor

function as a method of that object. This is not the complete story, however. After creating the empty object,

`new` sets the `prototype` of that object. The `prototype` of an object is the value of the `prototype` property of its

constructor function. All functions have a `prototype` property that is automatically created and initialized when

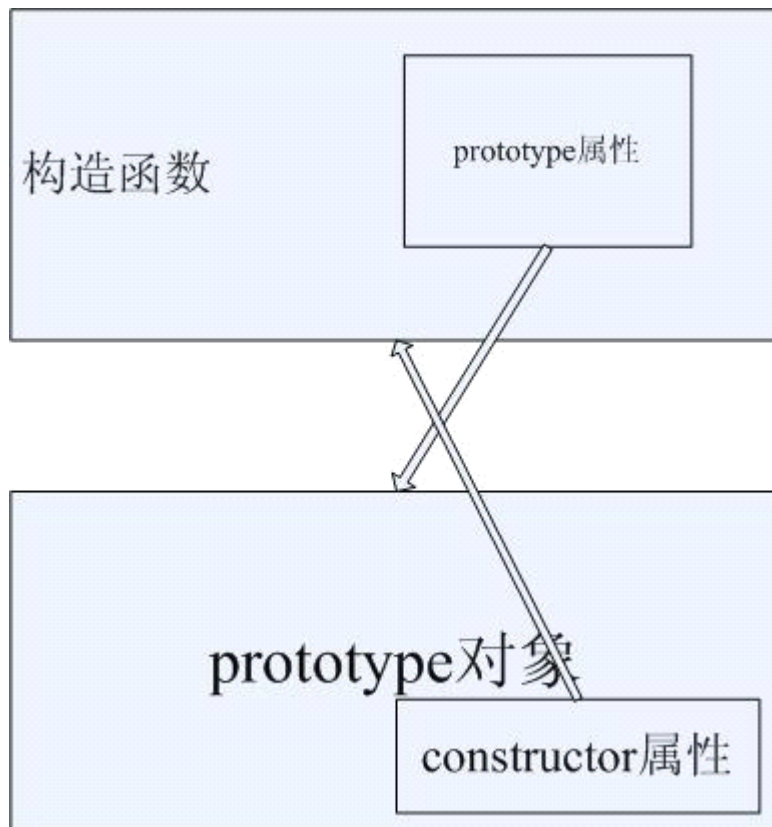
the function is defined. The initial value of the `prototype` property is an object with a single property. This property

is named `constructor` and refers back to the constructor function with which the `prototype` is associated. this is why every

object has a `constructor` property. Any properties you add to this `prototype` object will appear to be properties of

objects initialized by the constructor. ----section9.2)

有点晕,看下面的图:



这样,当用构造函数创建一个新的对象时,它会获取构造函数的 `prototype` 属性所指向的 `prototype` 对象的所有属性.对构造函数对应的 `prototype` 对象所做的任何操作都会反应到它所生成的对象身上,所有的这些对象共享构造函数对应的 `prototype` 对象的属性(包括方法).  
看个具体的例子吧:

Js 代码

```
7  function Person(name,sex) { //构造函数
8      this.name = name;
9      this.sex = sex;
10 }
11 Person.prototype.age = 12; //为 prototype 属性对应的 prototype 对象的属性赋值
12 Person.prototype.print = function() { //添加方法
13     alert(this.name+"_"+this.sex+"_"+this.age);
14 };
15
16 var p1 = new Person("name1","male");
17 var p2 = new Person("name2","male");
18 p1.print(); //name1_male_12
19 p2.print(); //name2_male_12
20
```

```

21 Person.prototype.age = 18; //改变 prototype 对象的属性值,注意是操作构造函数的
    prototype 属性
22 p1.print(); //name1_male_18
23 p2.print(); //name2_male_18

```

到目前为止,我们已经模拟出了简单的类的实现,我们有了构造函数,有了类属性,有了类方法,可以创建"实例".

在下面的文章中,我们就用"类"这个名字来代替构造方法,但是,这仅仅是模拟,并不是真正的面向对象的"类".

在下一步的介绍之前,我们先来看看改变对象的 prototype 属性和设置 prototype 属性的注意事项:

给出一种不是很恰当的解释,或许有助于我们理解:当我们 new 了一个对象之后,这个对象就会获得构造函数的 prototype 属

性(包括函数和变量),可以认为是构造函数(类)继承了它的 prototype 属性对应的 prototype 对象的函数和变量,也就是说,

prototype 对象模拟了一个超类的效果.听着比较拗口,我们直接看个实例吧:

Js 代码

```

24 function Person(name,sex) { //Person 类的构造函数
25     this.name = name;
26     this.sex = sex;
27 }
28 Person.prototype.age = 12; //为 Person 类的 prototype 属性对应的 prototype 对象的
    属性赋值,
29                                     //相当于为 Person 类的父类添加属性
30 Person.prototype.print = function() { //为 Person 类的父类添加方法
31     alert(this.name+"_"+this.sex+"_"+this.age);
32 };
33
34 var p1 = new Person("name1","male"); //p1 的 age 属性继承子 Person 类的父类 (即
    prototype 对象)
35 var p2 = new Person("name2","male");
36
37 p1.print(); //name1_male_12
38 p2.print(); //name2_male_12
39
40 p1.age = 34; //改变 p1 实例的 age 属性
41 p1.print(); //name1_male_34
42 p2.print(); //name2_male_12
43
44 Person.prototype.age = 22; //改变 Person 类的超类的 age 属性
45 p1.print(); //name1_male_34(p1 的 age 属性并没有随着 prototype 属性的改变而改变)

```

```

46 p2.print(); //name2_male_22(p2的 age 属性发生了改变)
47
48 p1.print = function() { //改变 p1对象的 print 方法
49     alert("i am p1");
50 }
51
52 p1.print(); //i am p1(p1的方法发生了改变)
53 p2.print(); //name2_male_22(p2的方法并没有改变)
54
55 Person.prototype.print = function() { //改变 Person 超类的 print 方法
56     alert("new print method!");
57 }
58
59 p1.print(); //i am p1(p1的 print 方法仍旧是自己的方法)
60 p2.print(); //new print method!(p2的 print 方法随着超类方法的改变而改变)

```

看过一篇文章介绍说 javascript 中对象的 prototype 属性相当于 java 中的 static 变量,可以被这个类下的所有对象

共用.而上面的例子似乎表明实际情况并不是这样:

在 JS 中,当我们用 new 操作符创建了一个类的实例对象后,它的方法和属性确实继承了类的 prototype 属性,类的 prototype 属性

中定义的方法和属性,确实可以被这些实例对象直接引用.但是,当我们对这些实例对象的属性和方法重新赋值或定义后,那么

实例对象的属性或方法就不再指向类的 prototype 属性中定义的属性和方法,此时,即使再对类的 prototype 属性中相应的方法或

属性做修改,也不会反应在实例对象身上.这就解释了上面的例子:

一开始,用 new 操作符生成了两个对象 p1,p2,他们的 age 属性和 print 方法都来自(继承于)Person 类的 prototype 属性.然后,我们

修改了 p1 的 age 属性,后面对 Person 类的 prototype 属性中的 age 重新赋值(Person.prototype.age = 22),p1 的 age 属性并不会

随之改变,但是 p2 的 age 属性却随之发生了变化,因为 p2 的 age 属性还是引自 Person 类的 prototype 属性.同样的情况在后面的 print 方法中也体现了出来.

通过上面的介绍,我们知道 prototype 属性在 javascript 中模拟了父类(超类)的角色,在 js 中体现面向对象的思想,prototype 属性是非常关键的.

## javascript 面向对象技术基础(五)

类变量/类方法/实例变量/实例方法

先补充一下以前写过的方法:

在 javascript 中,所有的方法都有一个 call 方法和 apply 方法.这两个方法可以模拟对象调用方法.它的第一个参数是对象,后面的

参数表示对象调用这个方法时的参数 (ECMAScript specifies two methods that are defined for all functions, call()

and apply(). These methods allow you to invoke a function as if it were a method of some other object. The first

argument to both call() and apply() is the object on which the function is to be invoked; this argument becomes

the value of the this keyword within the body of the function. Any remaining arguments to call() are the values

that are passed to the function that is invoked). 比如我们定义了一个方法 f(),然后调用下面的语句:

```
f.call(o, 1, 2);
```

作用就相当于

```
o.m = f;
```

```
o.m(1,2);
```

```
delete o.m;
```

举个例子:

Js 代码

```
1  function Person(name,age) { //定义方法
2      this.name = name;
3      this.age = age;
4  }
5  var o = new Object(); //空对象
6  alert(o.name + "_" + o.age); //undefined_undefined
7
8  Person.call(o,"sdcyst",18); //相当于调用:o.Person("sdcyst",18)
9  alert(o.name + "_" + o.age); //sdcyst_18
10
11 Person.apply(o,["name",89]); //apply 方法作用同 call,不同之处在于传递参数的形式是
    用数组来传递
12 alert(o.name + "_" + o.age); //name_89
```

实例变量和实例方法都是通过实例对象加"."操作符然后跟上属性名或方法名来访问的,但是

我们也可以为类来设置方法或变量,这样就可以直接用类名加"."操作符然后跟上属性名或方法名来访问.定义类属性和类方法很简单:

Js 代码

```

13 Person.counter = 0; //定义类变量,创建的 Person 实例的个数
14 function Person(name,age) {
15     this.name = name;
16     this.age = age;
17     Person.counter++; //没创建一个实例,类变量 counter 加1
18 };
19
20 Person.wholsOlder = function(p1,p2) { //类方法,判断谁的年龄较大
21     if(p1.age > p2.age) {
22         return p1;
23     } else {
24         return p2;
25     }
26 }
27
28 var p1 = new Person("p1",18);
29 var p2 = new Person("p2",22);
30
31 alert("现在有 " + Person.counter + "个人"); //现在有2个人
32 var p = Person.wholsOlder(p1,p2);
33 alert(p.name + "的年龄较大"); //p2的年龄较大

```

prototype 属性的应用:

下面这个例子是根据原书改过来的.

假设我们定义了一个 Circle 类,有一个 radius 属性和 area 方法,实现如下:

Js 代码

```

34 function Circle(radius) {
35     this.radius = radius;
36     this.area = function() {
37         return 3.14 * this.radius * this.radius;
38     }
39 }
40 var c = new Circle(1);
41 alert(c.area()); //3.14

```

假设我们定义了 100 个 Circle 类的实例对象,那么每个实例对象都有一个 radius 属性和 area 方法,

实际上,除了 radius 属性,每个 Circle 类的实例对象的 area 方法都是一样,这样的话,我们就可以把 area 方法抽出来定义在 Circle 类的 prototype 属性中,这样所有的实例对象就可以调用这个方法,

从而节省空间.

Js 代码

```
42 function Circle(radius) {
43     this.radius = radius;
44 }
45 Circle.prototype.area = function() {
46     return 3.14 * this.radius * this.radius;
47 }
48 var c = new Circle(1);
49 alert(c.area()); //3.14
```

现在,让我们用 `prototype` 属性来模拟一下类的继承:首先定义一个 `Circle` 类作为父类,然后定义子类

`PositionCircle`.

Js 代码

```
50 function Circle(radius) { //定义父类 Circle
51     this.radius = radius;
52 }
53 Circle.prototype.area = function() { //定义父类的方法 area 计算面积
54     return this.radius * this.radius * 3.14;
55 }
56
57 function PositionCircle(x,y,radius) { //定义类 PositionCircle
58     this.x = x; //属性横坐标
59     this.y = y; //属性纵坐标
60     Circle.call(this,radius); //调用父类的方法,相当于调用 this.Circle(radius),设置 PositionCircle 类的
61                                //radius 属性
62 }
63 PositionCircle.prototype = new Circle(); //设置 PositionCircle 的父类为 Circle 类
64
65 var pc = new PositionCircle(1,2,1);
66 alert(pc.area()); //3.14
67 //PositionCircle 类的 area 方法继承自 Circle 类,而 Circle 类的
68 //area 方法又继承自它的 prototype 属性对应的 prototype 对象
69 alert(pc.radius); //1 PositionCircle 类的 radius 属性继承自 Circle 类
70
71 /*
72 注意:在前面我们设置 PositionCircle 类的 prototype 属性指向了一个 Circle 对象,
73 因此 pc 的 prototype 属性继承了 Circle 对象的 prototype 属性,而 Circle 对象的
74 constructor 属
```



```

74 性(即 Circle 对象对应的 prototype 对象的 constructor 属性)是指向 Circle 的,所以此处
    弹出
75 的是 Circ.
76 */
77 alert(pc.constructor); //Circle
78
79 /*为此,我们在设计好了类的继承关系后,还要设置子类的 constructor 属性,否则它会
    指向父类
80 的 constructor 属性
81 */
82 PositionCircle.prototype.constructor = PositionCircle
83 alert(pc.constructor); //PositionCircle

```

## javascript 面向对象技术基础(六)

作用域、闭包、模拟私有属性

先来简单说一下变量作用域，这些东西我们都很熟悉了，所以也不详细介绍。

Js 代码

```

1  var sco = "global"; //全局变量
2  function t() {
3      var sco = "local"; //函数内部的局部变量
4      alert(sco);          //local 优先调用局部变量
5  }
6  t();                      //local
7  alert(sco);              //global 不能使用函数内的局部变量

```

注意一点，在 javascript 中没有块级别的作用域，也就是说在 java 或 c/c++中我们可以用"{}"来包围一个块，从而在其中定义块内的局部变量，在"{}"块外部,这些变量不再起作用,同时,也可以在 for 循环等控制语句中定义局部的变量,但在 javascript 中没有此项特性:

Js 代码

```

8  function f(props) {
9      for(var i=0; i<10; i++) {}
10     alert(i);          //10 虽然 i 定义在 for 循环的控制语句中,但在函数
11                        //的其他位置仍旧可以访问该变量.
12     if(props == "local") {

```

```

13         var sco = "local";
14     alert(sco);
15 }
16     alert(sco);        //同样,函数仍可引用 if 语句内定义的变量
17 }
18 f("local");           //10  local  local

```

在函数内部定义局部变量时要格外小心:

Js 代码

```

19 var sco = "global";
20 function print1() {
21     alert(sco);    //global
22 }
23 function print2() {
24     var sco = "local";
25     alert(sco);    //local
26 }
27 function print3() {
28     alert(sco);    //undefined
29     var sco = "local";
30     alert(sco);    local
31 }
32
33 print1(); //global
34 print2(); //local
35 print3(); //undefined local

```

前面两个函数都很容易理解，关键是第三个:第一个 alert 语句并没有把全局变量 "global" 显示出来，

而是 undefined，这是因为在 print3 函数中，我们定义了 sco 局部变量(不管位置在何处),那么全局的

sco 属性在函数内部将不起作用，所以第一个 alert 中 sco 其实是局部 sco 变量，相当于：

Js 代码

```

36 function print3() {
37     var sco;
38     alert(sco);
39     sco = "local";
40     alert(sco);
41 }

```

从这个例子我们得出，在函数内部定义局部变量时，最好是在开头就把所需的变量定义好，以免出错。

函数的作用域在定义函数的时候已经确定了，例如：

Js 代码

```
42 var scope = "global"    //定义全局变量
43 function print() {
44     alert(scope);
45 }
46 function change() {
47     var scope = "local"; //定义局部变量
48     print();             //虽然是在 change 函数的作用域内调用 print 函数，
49                           //但是 print 函数执行时仍旧按照它定义时的作用域起
作用
50 }
51 change();               //global
```

闭包

闭包是拥有变量、代码和作用域的表达式.在 javascript 中,函数就是变量、代码和函数的作用域的组合物,因此所有

的函数都是闭包 (JavaScript functions are a combination of code to be executed and the scope in which to

execute them. This combination of code and scope is known as a closure in the computer science literature.

All JavaScript functions are closures).好像挺简单.但是闭包到底有什么作用呢?看一个例子。

我们想写一个方法，每次都得到一个整数，这个整数是每次加1的，没有思索，马上下笔：

Js 代码

```
52 var i = 0;
53 function getNext() {
54     i++;
55     return i;
56 }
57 alert(getNext()); //1
58 alert(getNext()); //2
59 alert(getNext()); //3
```

一直用 getNext 函数得到下一个整数,而后不小心或者故意的将全局变量 i 的值设为0,然后再

次调用 getNext,

你会发现又从1开始了.....这时你会想到,要是把i设置成一个私有变量该多好,这样只有在方法内部才

可能改变它,在函数之外就没有办法修改了.下面的代码就是按照这个要求来做得,后面我们详细讨论。

为了解释方便,我们就把下面的代码称为 demo1.

Js 代码

```
60 function temp() {  
61     var i = 0;  
62     function b() {  
63         return ++i;  
64     }  
65     return b;  
66 }  
67 var getNext = temp();  
68 alert(getNext());    //1  
69 alert(getNext());    //2  
70 alert(getNext());    //3  
71 alert(getNext());    //4
```

因为我们平时所说的 javascript 绝大多数都是指在客户端(浏览器)下,所以这里也不例外。在 javascript 解释器启动时,会首先创建一个全局的对象(global object),也就是"window"所引用的对象。

然后我们定义的所有全局属性和方法等都会成为这个对象的属性。

不同的函数和变量的作用域是不同的,因而构成了一个作用域链(scope chain).很显然,在 javascript 解释器启动时,

这个作用域链只有一个对象:window(Window Object, 即 global object)。

在 demo1中,temp 函数是一个全局函数,因此 temp()函数的作用域(scope)对应的作用域链就是 js 解释器启动时的作用域链,只有一个 window 对象。

当 temp 执行时,首先创建一个 call 对象(活动对象),然后把这个 call 对象添加到 temp 函数对应的作用域链的最前头,这是, temp()函数

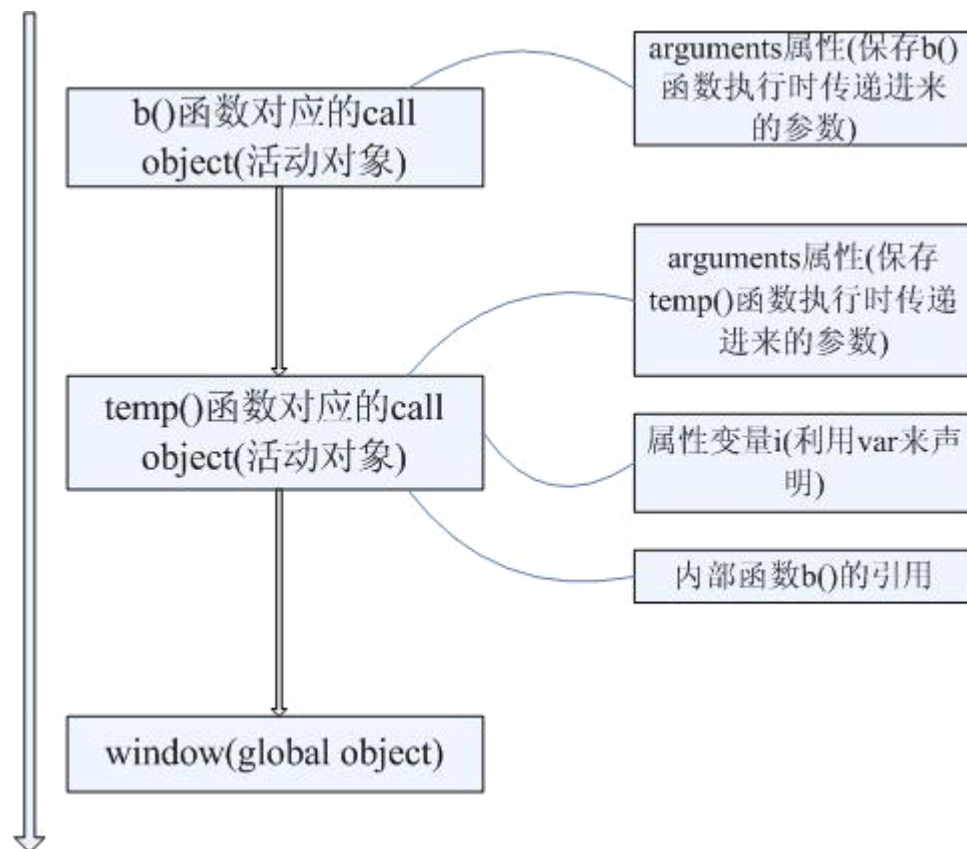
对应的作用域链就包含了两个对象: window 对象和 temp 函数对应的 call object(活动对象)。

然后呢,因为我们在 temp 函数里定义了变量 i,

定义了函数 b(),这些都会成为 call object 的属性。当然,在这之前会首先给 call object 对象添加 arguments 属性,保存了 temp()函数执行时

传递过来的参数。此时,整个的作用域链如下图所示:

同理可以得出函数 b()执行时的整个作用域链:



注意在 `b()` 的作用域链中，`b()` 函数对应的 `call object` 只有一个 `arguments` 属性，并没有 `i` 属性，这是因为在 `b()` 的定义中，并没有用 `var` 关键字来声明 `i` 属性，只有用 `var` 关键字声明的属性才会添加到对应的 `call object` 上。在函数执行时，首先查找对应的 `call object` 有没有需要的属性，如果没有，再往上一级查找，直到找到为止，如果找不到，那就是 `undefined` 了。

这样我们再来看 `demo1` 的执行情况。我们用 `getNext` 引用了 `temp` 函数，而 `temp` 函数返回了函数 `b`，这样 `getNext` 函数其实就是 `b` 函数的引用。执行一次 `getNext`，就执行一次 `b()` 函数。因为函数 `b()` 的作用域依赖于函数 `temp`，因此 `temp` 函数在内存中会一直存在。函数 `b` 执行时，首先查找 `i`，在 `b` 对应的 `call object` 中没有，于是往上一级找，在 `temp` 函数对应的 `call object` 中找到了，于是将其值加1，然后返回这个值。这样，只要 `getNext` 函数有效，那么 `b()` 函数就一直有效，同时，`b()` 函数依赖的 `temp` 函数也不会消失，变量 `i` 也不会消失，而且这个变量在 `temp` 函数外部根本就访问不到，只能在 `temp()` 函数内部访问(`b` 当然可以了)。

来看一个利用闭包来模拟私有属性的例子：

Js 代码

```

72 function Person(name, age) {
73     this.getName = function() { return name; };
74     this.setName = function(newName) { name = newName; };
75     this.getAge = function() { return age; };

```

```

76     this.setAge = function(newAge) { age = newAge; };
77 }
78
79 var p1 = new Person("sdcyst",3);
80 alert(p1.getName()); //sdcyst
81 alert(p1.name);       //undefined    因为 Person('类')没有 name 属性
82 p1.name = "mypara"    //显示的给 p1添加 name 属性
83 alert(p1.getName()); //sdcyst      但是并不会改变 getName 方法的返回值
84 alert(p1.name);       //mypara      显示出 p1对象的 name 属性
85 p1.setName("sss");    //改变私有的"name"属性
86 alert(p1.getName()); //sss
87 alert(p1.name);       //仍旧为 mypara

```

定义了一个 Person 类，有两个私有属性 name，age，分别定义对应的 get/set 方法。虽然可以显示的设置 p1 的 name、age 属性，但是这种显示的设置，并不会改变我们最初设计时模拟出来的"name/age"私有属性。

解释闭包的确不是一件容易的事，在网上很多人也是利用例子来说明闭包。如果有地方说的不对，还请指正。

下面是另一篇解释 javascript 闭包的文章，一块儿参考吧。

## 领悟 JavaScript 中的面向对象

JavaScript 是面向对象的。但是不少人对这一点理解得并不全面。

在 JavaScript 中，对象分为两种。一种可以称为“普通对象”，就是我们所普遍理解的那些：数字、日期、用户自定义的对象（如：{}）等等。

还有一种，称为“方法对象”，就是我们通常定义的 function。你可能觉得奇怪：方法就是方法，怎么成了对象了？但是在 JavaScript 中，方法的确是被当成对象来处理的。下面是一个简单的例子：

Js 代码

```

1  function func() {alert('Hello!');}
2  alert(func.toString());

```

在这个例子中，func 虽然是作为一个方法定义的，但它自身却包含一个 toString 方法，说

明 `func` 在这里是被当成一个对象来处理的。更准确的说, `func` 是一个“方法对象”。下面是例子的继续:

Js 代码

```
3 func.name = "I am func.";
4 alert(func.name);
```

我们可以任意的为 `func` 设置属性, 这更加证明了 `func` 就是一个对象。那么方法对象和普通对象的区别在哪里呢? 首先方法对象当然是可以执行的, 在它后面加上一对括号, 就是执行这个方法对象了。

Js 代码

```
5 func();
```

所以, 方法对象具有二重性。一方面它可以被执行, 另一方面它完全可以被当成一个普通对象来使用。这意味着什么呢? 这意味着方法对象是可以完全独立于其他对象存在的。这一点我们可以同 `Java` 比较一下。在 `Java` 中, 方法必须在某一个类中定义, 而不能单独存在。而 `JavaScript` 中就不需要。

方法对象独立于其他方法, 就意味着它能够被任意的引用和传递。下面是一个例子:

Js 代码

```
6 function invoke(f) {
7     f();
8 }
9 invoke(func);
```

将一个方法对象 `func` 传递给另一个方法对象 `invoke`, 让后者在适当的时候执行 `func`。这就是所谓的“回调”了。另外, 方法对象的这种特殊性, 也使得 `this` 关键字不容易把握。这方面相关文章不少, 这里不赘述了。

除了可以被执行以外, 方法对象还有一个特殊的功用, 就是它可以通过 `new` 关键字来创建普通对象。

话说每一个方法对象被创建时, 都会自动的拥有一个叫 `prototype` 的属性。这个属性并没有什么特别之处, 它和其他的属性一样可以访问, 可以赋值。不过当我们用 `new` 关键字来创

建一个对象的时候，`prototype` 就起作用了：它的值（也是一个对象）所包含的所有属性，都会被复制到新建的那个对象上去。下面是一个例子：

Js 代码

```
10 func.prototype.name="prototype of func";
11 var f = new func();
12 alert(f.name);
```

执行的过程中会弹出两个对话框，后一个对话框表示 `f` 这个新建的对象从 `func.prototype` 那里拷贝了 `name` 属性。而前一个对话框则表示 `func` 被作为方法执行了一遍。你可能会问了，为什么这个时候要还把 `func` 执行一遍呢？其实这个时候执行 `func`，就是起“构造函数”的作用。为了形象的说明，我们重新来一遍：

Js 代码

```
13 function func() {
14     this.name="name has been changed."
15 }
16 func.prototype.name="prototype of func";
17 var f = new func();
18 alert(f.name);
```

你就会发现 `f` 的 `name` 属性不再是 "prototype of func"，而是被替换成了 "name has been changed"。这就是 `func` 这个对象方法所起到的“构造函数”的作用。所以，在 JavaScript 中，用 `new` 关键字创建对象是执行了下面三个步骤的：

- 19 创建一个新的普通对象；
- 20 将方法对象的 `prototype` 属性的所有属性复制到新的普通对象中去。
- 21 以新的普通对象作为上下文来执行方法对象。

对于“`new func()`”这样的语句，可以描述为“从 `func` 创建一个新对象”。总之，`prototype` 这个属性的唯一特殊之处，就是在创建新对象的时候了。

那么我们就可以利用这一点。比如有两个方法对象 `A` 和 `B`，既然从 `A` 创建的新对象包含了所有 `A.prototype` 的属性，那么我将它赋给 `B.prototype`，那么从 `B` 创建的新对象不也有同样的属性了？写成代码就是这样：



## Js 代码

```
22 A.prototype.hello = function(){alert('Hello!');}  
23 B.prototype = new A();  
24 new B().hello();
```

这就是 JavaScript 的所谓“继承”了，其实质就是属性的拷贝，这里利用了 `prototype` 来实现。如果不用 `prototype`，那就用循环了，效果是一样的。所谓“多重继承”，自然就是到处拷贝了。

JavaScript 中面向对象的原理，就是上面这些了。自始至终我都没提到“类”的概念，因为 JavaScript 本来就没有“类”这个东西。面向对象可以没有类吗？当然可以。先有类，然后才有对象，这本来就不合理，因为类本来是从对象中归纳出来的，先有对象再有类，这才合理。像下面这样的：

## Js 代码

```
25 var o = {}; // 我发现了一个东西。  
26 o.eat = function(){return "I am eating."} // 我发现它会吃；  
27 o.sleep = function(){return "ZZZZzzz.."} // 我发现它会睡；  
28 o.talk = function(){return "Hi!"} // 我发现它会说话；  
29 o.think = function(){return "Hmmm..."} // 我发现它还会思考。  
30  
31 var Human = new Function(); // 我决定给它起名叫“人”。  
32 Human.prototype = o; // 这个东西就代表了所有“人”的概念。  
33  
34 var h = new Human(); // 当我发现其他同它一样的东西，  
35 alert(h.talk()) // 我就知道它也是“人”了！
```