# 函数

## 函数的声明

函数对任何语言来说都是一个核心的概念。通过函数可以封装任意多条语句，而且可以在任何地方、任何时候调用执行。ECMAScript中的函数使用function关键字来声明，后跟一组参数以及函数体。

函数的构成：

1.使用function声明（必须）

2.函数名

3.参数列表

4.函数体（必须）

5.返回值

function myFunction() { //没有参数的函数

alert('只有函数被调用，我才会被之执行');

}

myFunction(); //直接调用函数

function myFunction(name, age) { //带参数的函数

alert('姓名：'+name+'，年龄：'+age);

}

myFunction('马云',50); //调用函数，并传参

声明方法：

1、函数声明：

function name(([param[, param[, ... param]]]) {

statements

}

2、函数表达式

var func = function [name]([param] [, param] [..., param]) {

statements

}

3、通过Function对象的构造函数声明

[new] Function (arg1, arg2, ... argN, functionBody)

## arguments对象（了解）

JavaScript中函数不介意传递进来多少参数，也不会因为参数不统一而错误。实际上，函数体内可以通过arguments数组对象来接收传递进来的参数。

function myFunction() {

return arguments[0]+' | '+arguments[1]; //得到每次参数的值

}

alert(myFunction(1,2,3,4,5,6)); //传递参数

arguments对象的length属性可以得到参数的数量。

function myFunction() {

return arguments.length; //得到6

}

alert(myFunction(1,2,3,4,5,6));

我们可以利用length这个属性，来智能的判断有多少参数，然后把参数进行合理的应用。比如，要实现一个加法运算，将所有传进来的数字累加，而数字的个数又不确定。

function myFunction() {

var sum = 0;

if (arguments.length == 0) {

return sum;

}

for(var i = 0;i < arguments.length; i++) { //如果有，就累加

sum = sum + arguments[i];

} return sum; //返回累加结果

}

alert(myFunction(5,9,12));

**注意：Javascript中的函数，没有像其他高级语言那种**函数重载功能。

function myFunction(num) {

return num + 100;

}

function myFunction (num) { //会执行这个函数,执行最后一个；

return num + 200;

}

alert(myFunction(50)); //返回结果

## 函数return返回值

上面的例子中带参和不带参的函数，都没有定义返回值，而是调用后直接执行的。实际上，任何函数都可以通过return语句跟后面的要返回的值来实现返回值。

function myFunction() {

var str= '我被调用了！'; //没有参数的函数

return str ; //通过return把函数的最终值返回

}

alert(myFunction()); //调用函数会得到返回值，然后外面输出

function myFunction(name, age) { //有参数的函数

var str='你的姓名：'+name+'，年龄：'+age;

return str; //通过return 把函数的最终值返回

}

alert(myFunction('马云', 50)); //调用函数得到返回值，然后外面输出

我们还可以把函数的返回值赋给一个变量，然后通过变量进行操作。

function myFunction(num1, num2) {

return num1 \* num2;

}

var num = myFunction(10, 5); //函数得到的返回值赋给变量

alert(num);

return语句还有一个功能就是退出当前函数，注意和break的区别。

PS：break用在循环和switch分支语句里。

function myFunction(num) {

if(num<5){

return num;

}else{

return 100;

}

}

Alert(myFunction(10));

## 函数调用

函数定义后并不会自动执行，函数需要被显示调用才可以执行；函数调用的方法：函数调用模式，方法调用模式，构造器调用模式，apply/call调用模式。

1.函数调用模式  
定义一个函数，设置一个变量名保存函数，这时this指向到window对象。

var myfunc = function(a,b){

return a+b;

}

alert(myfunc(3,4));

2.方法调用模式：

先定义一个对象，然后在对象的属性中定义方法，通过myobject.property来执行方法，this即指当前的myobject对象。

var blogInfo={  
　　blogId:123,  
　　blogName:"werwr",  
　　showBlog:function(){alert(this.blogId);}  
};

blogInfo.showBlog();

3.构造器调用模式

定义一个函数对象，在对象中定义属性，在其原型对象中定义方法。在使用prototype的方法时，必须实例化该对象才能调用其方法。

var myfunc = function(a){

this.a = a;

};

myfunc.prototype = {

show:function(){alert(this.a);}

}

var newfunc = new myfunc("123123123");

newfunc.show();

4.apply,call调用模式

以call函数为例，call的第一个参数，就是改变函数的作用域，后面的参数为传入函数的所需的参数，必须与原函数的参数一致。而apply与call的区别仅在于，apply的第二个参数可以是数组形式，而且不必一一指出参数

var myobject={c:5};

var sum = function(a,b){

return a+b+this.c;

};

var sum2 = sum.call(myobject,10,30);

//var sum2 = sum.apply(myobject,[10,30]);

alert(sum2);

**this关键字**

在函数内部存在this关键字，在JavaScript中this关键字有特殊的含义。它代表函数运行时，自动生成的一个内部对象，只能在函数内部使用, 随着函数使用场合的不同，this的值会发生变化。但是有一个总的原则，那就是this指的是，调用函数的那个对象(离它最近的那个对象)。

示例：

var ok = ‘ok’;

var obj = {ok:'I am Ok'};

var sayOK = function(){

console.log(this['ok']);

}

obj.sayOk = sayOK;

obj.sayOk();

sayOK();

**情况一：纯粹的函数调用**

这是函数的最通常用法，属于全局性调用，因此this就代表全局对象Global。请看下面这段代码，它的运行结果是1。

代码如下:

var x=1;

function test(){

　　alert(this.x);

}

test(); // 1

**情况二：作为对象方法的调用**

函数还可以作为某个对象的方法调用，这时this就指向调用它的对象。

代码如下:

var x=0;

function test(){

alert(this.x);

}

var o = {};

o.x = 1;

o.m = test;

o.m(); //1

## 匿名函数

匿名函数：故名思议就是没有名字的函数；可以有效的保证在页面上写入Javascript，而不会造成全局变量的污染。

//普通函数

function myFunction() { //函数名是myFunction

return 'Hello world！';

}

//匿名函数

function () { //匿名函数，会报错

return 'Hello world !';

}

**匿名函数的执行：**

1.通过表达式自我执行

(function (a,b) { //封装成表达式

alert('Hello world !');

})(10,20); //()表示执行函数，并且传参

2.把匿名函数赋值给变量

var myFunction = function () { //将匿名函数赋给变量

return 'Hello world !';

};

alert(myFunction()); //调用方式和函数调用相似

3.函数里的匿名函数(了解)

function myFunction () {

return function () { //函数里的匿名函数，产生闭包

return 'Hello world!';

}

}

alert(myFunction()());

## 作用域和闭包

闭包（closure）是Javascript语言的一个难点，也是它的特色，很多高级应用都要依靠闭包实现。

### 变量的作用域

要理解闭包，首先必须理解Javascript特殊的变量作用域。

变量的作用域无非就是两种：全局变量和局部变量。

"全局变量"：声明在函数之外的变量，或者不使用var声明的变量。

"局部变量"：声明在函数体中的变量，并且只能在当前函数体内访问。

Javascript语言的特殊之处，就在于函数内部可以直接读取全局变量。

　　var n=999;

　　function f1(){

　　　　alert(n);

　　}

f1();

另一方面，在函数外部自然无法读取函数内的局部变量。

　　function f1(){

　　　　var n=999;

　　}

　　alert(n); // error

这里有一个地方需要注意，函数内部声明变量的时候，一定要使用var命令。如果不用的话，你实际上声明了一个全局变量！

　　function f1(){

　　　　n=999;

　　}

　　f1();

alert(n); // 999

### 如何从外部读取局部变量

出于种种原因，我们有时候需要得到函数内的局部变量。但是，前面已经说过了，正常情况下，这是办不到的，只有通过变通方法才能实现。

那就是在函数的内部，再定义一个函数。

　　function f1(){

　　　　var n=999;

　　　　function f2(){

　　　　　　alert(n); // 999

　　　　}

　　}

在上面的代码中，函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变量，对f1就是不可见的。这就是Javascript语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗！

　　function f1(){

　　　　var n=999;

　　　　function f2(){

　　　　　　alert(n);

　　　　}

　　　　return f2;

　　}

　　var result=f1();

　　result(); // 999

### 闭包的概念

上一节代码中的f2函数，就是闭包。各种专业文献上的"闭包"（closure）定义非常抽象，很难看懂。我的理解是，闭包就是能够读取其他函数内部变量的函数。

由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。

所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

### 闭包的用途

闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

怎么来理解这句话呢？请看下面的代码。

　　function f1(){

　　　　var n=999;

　　　　nAdd=function(){

n+=1

}

　　　　function f2(){

　　　　　　alert(n);

　　　　}

　　　　return f2;

　　}

　　var result=f1();

　　result(); // 999

　　nAdd();

　　result(); // 1000

在这段代码中，result实际上就是闭包f2函数。它一共运行了两次，第一次的值是999，第二次的值是1000。这证明了，函数f1中的局部变量n一直保存在内存中，并没有在f1调用后被自动清除。

为什么会这样呢？原因就在于f1是f2的父函数，而f2被赋给了result，这导致f2始终在内存中，而f2的存在依赖于f1，因此f1也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

这段代码中另一个值得注意的地方，就是"nAdd=function(){n+=1}"这一行，首先在nAdd前面没有使用var关键字，因此nAdd是一个全局变量，而不是局部变量。其次，nAdd的值是一个匿名函数（anonymous function），而这个匿名函数本身也是一个闭包，所以nAdd相当于是一个设置器，可以在函数外部对函数内部的局部变量进行操作。

### 使用闭包的注意点

1、由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2、闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

### 思考题

如果你能理解下面两段代码的运行结果，应该就算理解闭包的运行机制了。

代码片段一

　　var name = "The Window";

　　var object = {

　　　　name : "My Object",

　　　　getNameFunc : function(){

　　　　　　return function(){

　　　　　　　　return this.name;

　　　　　　};

　　　　}

　　};

　　alert(object.getNameFunc()());

代码片段二

　　var name = "The Window";

　　var object = {

　　　　name : "My Object",

　　　　getNameFunc : function(){

　　　　　　var that = this;

　　　　　　return function(){

　　　　　　　　return that.name;

　　　　　　};

　　　　}

　　};

　　alert(object.getNameFunc()());

# JavaScript对象

### 什么是对象

JavaScript对象就是带有属性和方法的特殊数据类型。对一系列属性和方法的封装。

JS支持对象：内置对象，用户自定义对象，浏览器对象（BOM），文档对象（DOM）

JS对象特点：封装、继承。

### 创建对象

JavaScript创建对象的方式有三种：字面量方式、new运算符和Object.create()。

#### 1、字面量(JSON)的方式创建对象

对象字面量提供了一种非常方便创建对象的方法。一个对象字面量就是一个花括号括起来的一系列键值对。键值对由属性名和属性值构成。

属性名：可以是JavaScript标识符也可以是字符串直接量(包括空字符串)。

属性的值：可以是任意类型的JavaScript表达式，表达式的值(可以是原始值也可以是对象值)，就是这个属性的值。

例子：

var e = {}

var point = {x:0,y:0}

var point2 = {x:point.x,y:point.y+1}

var obj = {a:27, "b":99};

var s = {"":88, "p":99}; // 怎么访问s的第一个属性？

var book = {

"a":"a",

author:{firstname:"F",secondname:"S"} //怎么访问book中author的firstname属 性的第一个属性？

}

当属性中出现空格、斜杠等特殊字符，或者使用的属性与JS关键词冲突时，则

必须使用引号。

var user = {

name:”a”,

say: function(){

alert(“Haha!”);

}

};

#### 2、new运算符创建对象

##### 2.1 通过new Object()创建对象

方法：new运算符后面跟随一个对象类型的标识符（构造函数）。

例如：

var user = new Object();

user.name = "a";

user.age = 21;

user.say=function(){

alert(“haha!”);

}

console.log(user);//输出Object {name: "a", age: 21}

var myObject = new Object();

myObject['0'] = 'a';

myObject['1'] = 'b';

myObject['2'] = 'c';

console.log(myObject); //输出 Object {0: "a", 1: "b", 2: "c"}

var myString = new String('abc');

console.log(myString); //输出 String {0: "a", 1: "b", 2: "c", length: 3}

上面的例子中我们使用构造函数的方式创建了对象，然后再给对象的属性赋值。第一种方法调用了Object构造函数来实现；第二种方法使用String构造函数实现。（string对象）。那我们是否可以自己定义构造函数以方便创建对象呢？当然可以。

##### 2.2 通过自定义构造函数

例如：

function person(){

this.name = “hantengfei”;

this.say = function(){

alert(“haha!”);

}

};

var p1=new person();

alert(p1.name);

p1.say();

var Person = function(name, age,) {

this.name = name;

this.age = age;

this.say = function(){

alert(“haha!”);

};

var myPerson = new Person(“yangmengmeng”, 21); //创建了一个person对象；

console.log(myPerson);

上面的例子中我们自定义了Person()构造函数，并使用它创建了一个复杂的Person对象。

#### 3、通过Object.create()创建对象（对象的继承）

在讲述第三种对象创建技术之前，我们应当首先解释一下原型的概念。每一个JavaScript对象都和另一个对象相关联，这里的另一个对象就是原型对象。原型的意思就是每一个对象都从原型继承属性和方法。每个对象都有一个内部属性prototype，我们通常称之为原型。原型的值可以是一个对象，也可以是null。如果它的值是一个对象，则这个对象也一定有自己的原型，这样就形成了一条线性的链，我们称之为原型链。

例如：

var myFunction = function () {}//构造方法

console.log(myFunction.prototype);//输出myFunction

console.log(typeof myFunction.prototype);//输出 'object'

console.log(String.prototype);//输出String

console.log(typeof String.prototype); //输出 'object'

默认的prototype 属性是Object()对象。而通过new String()创建的对象的原型就是String.prototype，由于prototype属性是一个对象，因此原型链查找到最后是Object.prototype。

所以，由new String()创建的String对象的属性同时继承自String.prototype和Object.prototype。当访问一个对象的某一属性，如果这个属性在此对象中不存在，就在它的prototype属性所指的原型对象的属性中寻找，如果找到则返回，否则继续沿着prototype链一直找下去，直到prototype为null的时候停止。

例如：

function foo(){}

foo.prototype.z = 13;

var obj = new foo();

obj.y = 12;

obj.x = 11;

console.log(obj.x);//1

console.log(obj.y);//2

console.log(obj.z);//3

ECMAScript 5 中引入了一个新方法：Object.create。可以调用这个方法来创建一个新对象。

新对象的原型就是调用create方法时传入的第一个参数；

例如：

var a = {a: 1}; // a ---> Object.prototype ---> null

var b = Object.create(a);// b ---> a ---> Object.prototype ---> null

console.log(b.a); // 1 (继承而来)

var c = Object.create(b);

// c ---> b ---> a ---> Object.prototype ---> null

如果想创建一个没有原型的新对象，可以传入参数null，新创建的对象不会继承任何东西，甚至不包括基础方法，比如toString()，

例如：

var d = Object.create(null); // d ---> null

console.log(d.toString()) //TypeError: undefined is not a function

如果想创建一个普通的空对象(比如通过{}或new Object()创建的对象)，需要传入

Object.prototype，比如：

var obj = Object.create(Object.prototype)

补充：（1）工厂方式（每创建一个对象，对象的方法是新对象，浪费资源）

function createObject(name){

       var p = new Object();

       p.name=name;

       p.say = function(){alert(p.name+'ff');}

       return p;

}

var p1 = createObject("p1");

var p2 = createObject("p2");

alert(p1.name+" "+p2.name);

p1.say();p2.say();

alert(p1.say==p2.say); //false

（2）构造函数方式

function Person(name){

this.name = name;

this.say = function(){

alert("I am "+this.name);

}

}

var p1 = new Person("wang");

var p2 = new Person("li");

p1.say();

p2.say();

alert(p1.say==p2.say); //false

创建对象时比工厂方法更易于理解。和工厂方法一样，每个对象都有自己的方法，浪费资源。

 （3）原型方式

function Person(){}

Person.prototype.name = "";

Person.prototype.say = function(){

alert("I am "+this.name);

}

var p1 = new Person();

var p2 = new Person();

alert(p1.say == p2.say);//true

缺点：无法在构造方法中传递参数，所有对象共享属性。优点：对象共方法，节约资源。

 （4）构造方法+原型方式

function Person(name){

       this.name = name;

}

Person.prototype.say = function(){

       alert("I am "+this.name);

}

var p1 = new Person("wang");

var p2 = new Person("li");

p1.say();

p2.say();

alert(p1.say==p2.say); //true

 PS：优点：解决了前面提到的问题。缺点：封装不够完美。

 （5）动态原形方式

function Person(name){

this.name = name;

if(Person.prototype.say == undefined){

       Person.prototype.say = function(){

              alert("I am "+this.name);

              }

       }

}

var p1 = new Person("wang");

var p2 = new Person("li");

p1.say();

p2.say();

alert(p1.say==p2.say); //true

### 访问属性

访问对象的属性和设置对象的属性可以使用**点运算符**或**方括号运算符**，比如：

var obj = {x:2,y:3};

obj.x;

obj["y"];

obj["x"] = 4;

obj.y = 5;

这两种方式的不同之处是方括号内还可以是一个计算结果为字符串的表达式

例如：

var obj = {y1:1,y2:2};

var i = 1, n = 2;

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

console.log(obj['y’+i]);

}

如果读取undefined上的属性，则会出现属性读写异常。

var obj = {x:1};

obj.y;

obj.y.z =2;

可以先对对象进行判断，再访问其属性，

例如：

if( obj.y) {

var yz = obj.y.z;

}

### 删除属性

delete运算符用来删除一个属性的定义，delete的返回值不是表示删除成功了，而是表示没有这个属性了。

例如

var obj = new Object();

obj.name='zs';

delete obj.name;// 输出true

obj.name;// 输出undefined

delete obj.name//输出true;

delete只是断开属性和宿主对象的联系，而不会在内存中删除该属性。

例如：

var a = {};

a.b = {x:1,y:2};

var c = a.b;

delete a.b;

console.log(c.x);//输出1

### 检测属性

有时我们需要对属性做一些检测，比如对象是否拥有某个属性，这个属性是对象自己的还是原型对象的，这个属性是否可以枚举等。

属性不仅包含标识它们可枚举的特性，还包括它们是否可写、可配置的特性。可枚举特性控制着当使用for/in遍历对象的属性时是否可以遍历该属性，可配置性控制着其他特性（包括属性是否可以删除）的修改。

例如：

var cat = new Object();

cat.legs = 4;

cat.name = "tom";

'legs' in cat;//true

'abc' in cat;//false

"toString" in cat;//true

cat.hasOwnProperty('legs');//true

cat.hasOwnProperty('toString');//false

cat.propertyIsEnumerable('legs');//true

cat.propertyIsEnumerable('toString');//false

如果想查看属性是否可枚举、可写、可配置，可以使用Object. getOwnPropertyDescriptor()方法得到对属性的描述信息。

默认情况下，属性的可枚举、可写、可配置都是false。

语法：

Object.getOwnPropertyDescriptor(obj,prop)

如果想设置属性是否可枚举、可写、可配置，可以使用Object.defineProperty ()方法.

语法：

Object.defineProperty(obj,prop,descriptor)

obj——对象

prop——需要定义或修改的属性名

descriptor——是一个对象，描述需要被定义或修改的属性。

示例：

var a = {x:1}

Object.getOwnPropertyDescriptor(a,'x');

Object.defineProperty(a,'y',{value:2,writable:false,enumerable:false,configurable:true});

a.y=3;

console.log(a.y); //输出2

Object.defineProperty(a,'y',{value:3,writable:false });

console.log(a.y); //输出3