**（1） 函数定义形式**

**函数定义**

函数是可以通过外部代码调用的一个“子程序”

一个函数由称为函数体的一系列语句组成。

值可以传递给一个函数，函数将返回一个值。

**函数定义方式（3种）**

**函数声明**

function max(a, b) {

return a > b ? a : b;

}

**函数表达式**

var max = function (a, b) {

return a > b ? a : b;

};

**Function** **构造函数实例化**

var max = new Function("a", "b", "return a > b ? a : b;");

**3. Function构造函数**

可以传入任意数量的实参

**最后一个实参为函数体**

函数体中 javascript 语句之间分号隔开

Function 构造函数创建一个匿名函数

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

**4. 函数定义三要素**

函数名

参数

返回值

**5. 具名/匿名函数**

单独的匿名函数是无法运行的，可以把匿名函数赋值给变量或立即执行

具名函数优势：当遇到错误时，堆栈跟踪会显示函数名，容易寻找错误

**6. 代理函数名**

var f1 = function f2(){};//f2是代理函数名

f2是可有可无的"代理"函数名

**代理函数名的作用域是只能在函数的主体( FunctionBody )内部**

var f1 = function f2() {};

f1();

f2();//error：f2 is not defined

var f1 = function f2() {

console.log(f1);

console.log(f2);

console.log(f1 === f2);//true

};

f1();

**7. name属性**

返回函数实例的名称

var f1 = function f2() {

console.log(f1.name);//f2

console.log(f2.name);//f2

};

console.log(f1.name);//f2

f1();

var f1 = function() {

console.log(f1.name);//f1

};

f1();

console.log(f1.name);//f1

var obj = { method: function() {} };

console.log(obj.method.name);//method

var f1 = new Function();

console.log(f1.name);// anonymous

//使用new Function()语法的函数其名称为“anonymous”

**8. length属性**

**length** **属性**指明函数定义的**形参个数**

function fun1(){}

console.log(fun1.length);//0

function fun2(a,b){}

console.log(fun2.length);//2

**（2） arguments对象**

**1. 函数的参数**

JavaScript 函数在定义时有固定数目的命名参数（形参），但当调用这个函数时，**传递给它的参数（实参）数目却可以是任意的**

**2. Arguments对象**

**代表传入函数的实参**

是函数中的**局部变量**

**系统创建**，不能显式创建，只有函数调用时才可用

它是一个**类数组\*\*\*\*对象**

**3. 类数组对象**

与数组一样具有 length 与 index 属性，本质是个对象object

不能调用push()、slice()等方法

var obj = {

0: "hello",

1: "world",

length: 2

};

console.log(obj[0]);// "hello"

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

for (var i = 0; i < obj.length; i++) {

console.log(obj[i]);

}

//"hello"

//"world"

function fun(a,b){

console.log(a);

console.log(b);

console.dir(arguments);

}

fun(1,2,3);

//1

//2

//arguments(3)

//0: 1

//1: 2

//2: 3

**4. Arguments与形参的双向绑定特性**

在调用时 arguments 对象与**实际传递了值的形参变量**发生双向绑定

arguments 对象中的对应单元会和命名参数建立关联

**5. arguments的length属性**

表示函数调用时传入的**实参**数量

在调用时，实参个数确定， arguments.length 确定， 不会再发生改变

**（3） call/apply/bind方法**

**1. 函数对象**

函数是对象，对象是一系列**属性**和**方法**的集合

属性：name、length

方法：toString、valueOf、call、apply、bind

**2. toString、valueOf方法**

toString方法：返回一个表示当前函数源代码的字符串

valueOf方法：返回函数本身

function add(a,b){

return a+b;

}

console.log(add.toString());

console.log(add.valueOf());

console.log(add + 1);

console.log(add + "1");

console.log(add + true);

//function add(a,b){ return a+b; }

//ƒ add(a,b){ return a+b; }

//function add(a,b){ return a+b; }1

//function add(a,b){ return a+b; }1

//function add(a,b){ return a+b; }true

**3. this关键字**

在 function 内部被创建，**指向调用时所在函数所绑定的对象**，**this** **不能被赋值，this的值取决于函数被调用的方式**

**4. call方法**

**调用函数，并改变函数执行的this指向**

fn.call(thisObj，arg1，arg2，...)

arg1,arg2,...：被调用函数的实参

thisObj：将函数对象中的 this 指向 **thisObj** **对象**

a.如果 thisObj \*\*未传递，this \*\*指向全局对象 **window**

b.如果传递为 undefined/null，this 指向全局对象 window

c.如果传递为数字，字符串，布尔值，this 指向该原始值的包装对象

返回值与fn普通调用相同

**5. apply方法**

**call方法与apply方法唯一不同是传参的形式不同**

fn.apply(thisObj，[arg1，arg2，...])

var x = 100;

var obj = {

x: 50

};

var foo = {

x: 0,

getX: function () {

return this.x;

}

};

//通过对象调用方法

console.log(foo.getX()); //0

//call()、apply()改变调用方法中this的指向为指定对象

console.log(foo.getX.call(obj)); //50

console.log(foo.getX.apply(obj)); //50

//call()、apply()没有指定对象时 默认指向全局对象（window）

console.log(foo.getX.call()); //100

console.log(foo.getX.apply()); //100

call()、apply()使用仍然是执行原来对象的方法里面的代码，只是代码中的this指向改变了。如果调用的对象方法里面没有this，那么使用call()和apply()没有任何改变，也没有意义

apple()方法的应用

var arr1 = [1, 2, 3, 4];

var arr2 = [5, 6, 7, 8];

//拼接数组arr1,arr2

arr1.push(5, 6, 7, 8);

console.log(arr1);//[1, 2, 3, 4, 5, 6, 7, 8]

arr1.push.apply(arr1, arr2);

console.log(arr1);//[1, 2, 3, 4, 5, 6, 7, 8, 5, 6, 7, 8]

var arr3 = [1, 5, 8, 2, 0, -2, 20];

//求数组的最大值和最小值

Math.max.apply(null, arr3);//20

Math.min.apply(null, arr3);//-2

//定义一个log 方法，让它可以代理 console.log 方法

function alog(msg) {

console.log(msg);

}

alog(1); //1

alog(1, 2); //1

function blog() {

console.log.apply(console, arguments);

}

blog(1); //1

blog(1, 2); //1 2

**6. bind方法**

**bind** **不会调用函数，即\*\*\*\*不会执行原函数中的代码**，apply，call 都会立即调用函数执行，bind 不会立即调用函数

fn.bind(thisObj，arg1，arg2,...)

当绑定函数调用时\*\*，\*\***thisObj** **参数作为原函数运行时的** **this** **指向**

arg1,arg2,... 当绑定函数被调用时，这些参数加上绑定函数本身的参数会按照顺序作为原函数运行时的参数。（预设参数）

**返回值：返回一个原函数的拷贝（绑定函数），并拥有指定的** **this** **值和初始参数**

//bind() 方法创建一个新的函数,并不会执行

//案例1

var length = 10;

function fn1() {

console.log(this);

console.log(this.length);

}

function fn2() {

console.log(this);

console.log(this.length);

}

var fn1Bound = fn1.bind(fn2);

console.log(fn1Bound == fn1); //false但是大括号里面的结构是一样的

console.log(fn1Bound);

//ƒ fn1() {console.log(this);console.log(this.length);}

fn1Bound(1, 2);

//ƒ fn2() {console.log(this);console.log(this.length);}

//0

//案例2

function add(a, b) {

console.log(this);

console.log(a + b);

}

function sub(a, b) {

console.log(this);

console.log(a - b);

}

var addBound = add.bind(sub, 3, 1);

console.log(addBound == add);//false

console.log(addBound);

//ƒ add(a, b) {console.log(this);console.log(a + b);}

<body>

<div id="div"></div>

<script>

var div = document.getElementById("div");

div.onclick = function() {

function change() {

this.style.background = "black";

//this指向window对象

}

setTimeout(change, 2000);//改法错误

};

</script>

</body>

<body>

<div id="div"></div>

<script>

var div = document.getElementById("div");

div.onclick = function() {

function change() {

this.style.background = "black";

}

setTimeout(change.bind(this), 2000);

//this指向div，改法正确

};

</script>

</body>

**（4） 函数调用形式（4种）**

**1. 作为函数直接调用 fn()**

**2. 作为对象方法调用 div.onclick**

**3. 作为构造函数调用 Array()**

**4. 通过 call/apply 间接调用 fn.apply()**