



第10课：函数调用和 作用域

■ 主讲老师：万章



目录

不同创造函数的方法

函数的多种执行方式


作用域

函数参数的多种方式



不同创造函数的方式

函数声明



```
关键字 函数名( ){  
    执行代码块  
}  
function fun( ){  
    console.log(1)  
}
```

函数表达式

除了函数声明的函数都叫函数表达式

```
let fun = function(){  
    console.log(1)  
}
```

有名函数

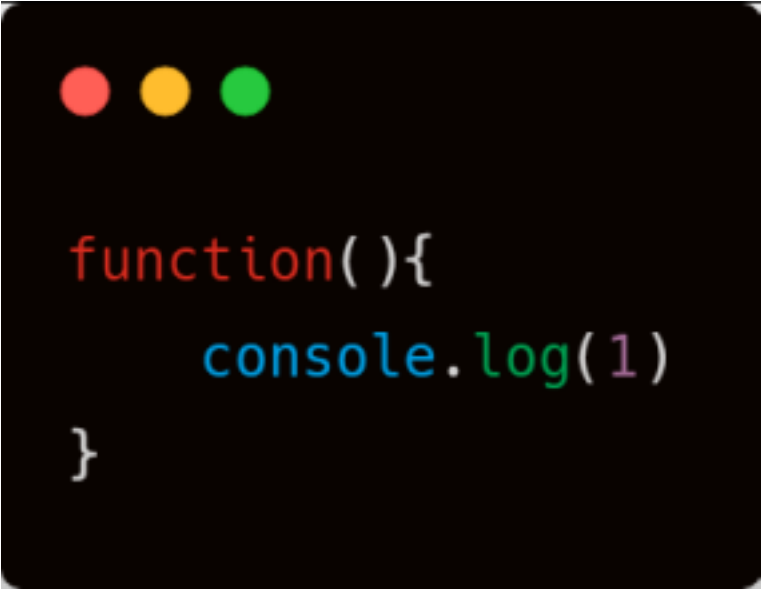


带有名称的函数

```
let fun1 = function(){} //函数名 fun1
```

```
function fun2(){} //函数名 fun2
```

匿名函数



```
function(){  
    console.log(1)  
}
```



函数的多种执行方式

函数的多种执行方式之事件函数



```
<div></div>
```

```
<script>
```

```
    let oDiv = document.getElementsByTagName("div")[0]
```

```
    function fun(){
```

```
        console.log(this.innerHTML)
```

```
    }
```

```
    oDiv.onclick = fun;
```

```
    /*
```

```
        元素.事件名称=函数名称
```

```
    */
```

```
</script>
```

函数的多种执行方式之函数名+括号执行

```
<div></div>
<script>
  var oDiv = document.getElementsByTagName("div")[0];
  function fun(){
    console.log("1");
  }
  fun();//输出1 变量fun的结果是一个函数

  var fun1 = function(){
    console.log(1);
  }();//函数立刻执行, 输出1, 而变量fun1的结果是undefined

  (function fn(){
    console.log(1);
  })();//输出1
</script>
```

该块代码我们可以进行细分研究

首先这是一个赋值语句

等号的左侧是变量fun1
等号右侧的值是一个匿名函数, 而且这个函数还执行了

所以变量fun1 的值 是右侧匿名函数的返回值, 而该匿名函数又没有返回值, 所以就是undefined

函数的多种执行方式之IIFE

IIFE(Immediately-invoked function expression)立即执行函数表达式

```
function(){
    console.log("匿名函数错误执行方法");
}();//此时会报错:

//需要将匿名函数用( )包裹
(function()
    console.log("匿名函数正确执行方式, 也叫IIFE");
})();//打印文字xxx

//还有别的写法
+function(){console.log(1)}();//也是
-function(){console.log(2)}();//也是
~function(){console.log(2)}();//也是
!function(){console.log(2)}();//也是
```

运算符 函数;

这种类型的立即执行函数其实可以从计算公式去理解

比如:

```
function add(){
    return 3;
}
var a=1+add();//结果是4
```

add函数执行一次然后将返回值用来计算, 那么我们去掉1
var b=+add();//b就等于3
咱们如果不把+add()的值赋值左侧的变量, 那么+add();依然会执行, 然后在内存里面存了一个数字3

add是一个函数名称 表示 function(){return 3;}
所以 +add()就变成
+function(){return 3;}():



作用域

全局作用域

全局作用域：

一个页面就是一个完整的执行环境，里面就存在唯一的一个作用域，就是全局作用域，全局作用域的本质是全局对象的属性

浏览器中全局对象是window，我们申明的变量都相当于在全局对象window下添加属性

```
var a = 2;  
console.log(a === window.a);  
//true 直接添加到window对象下成了其属性
```

函数作用域

函数作用域，函数里面的所有定义的新变量只有在函数作用域内才可以调用，外部不能访问内部函数作用域的变量。但是函数内部可以访问函数外部的变量
不同的函数的作用域不一样

```
var a2 = "这是foo外";

function foo() {
    var a1 = "这是foo内";
    console.log(a1, a2);
}

foo(); // "这是foo内" "这是foo外"
console.log(a1, a2); // "报错" "这是foo外"
```



函数参数的多种方式

函数参数的多种方式之标准传参

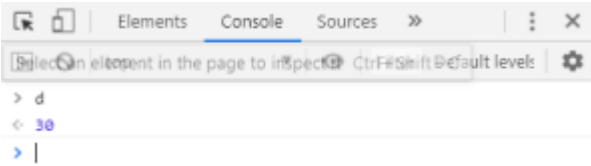
关键点:

1:函数只有在执行的时候才会在内存空间中新建一个函数的作用域, 函数执行完成之后就会销毁

```
<script>
  var a=10;
  var b=20;

  function fn(x,y){
    var c=x+y;
    return c;
  }

  var d=fn(a,b);
</script>
```



全局作用域(变量区)

```
var a;
var b;
```

```
var x;
var y;
var c;
```

函数作用域(变量区)

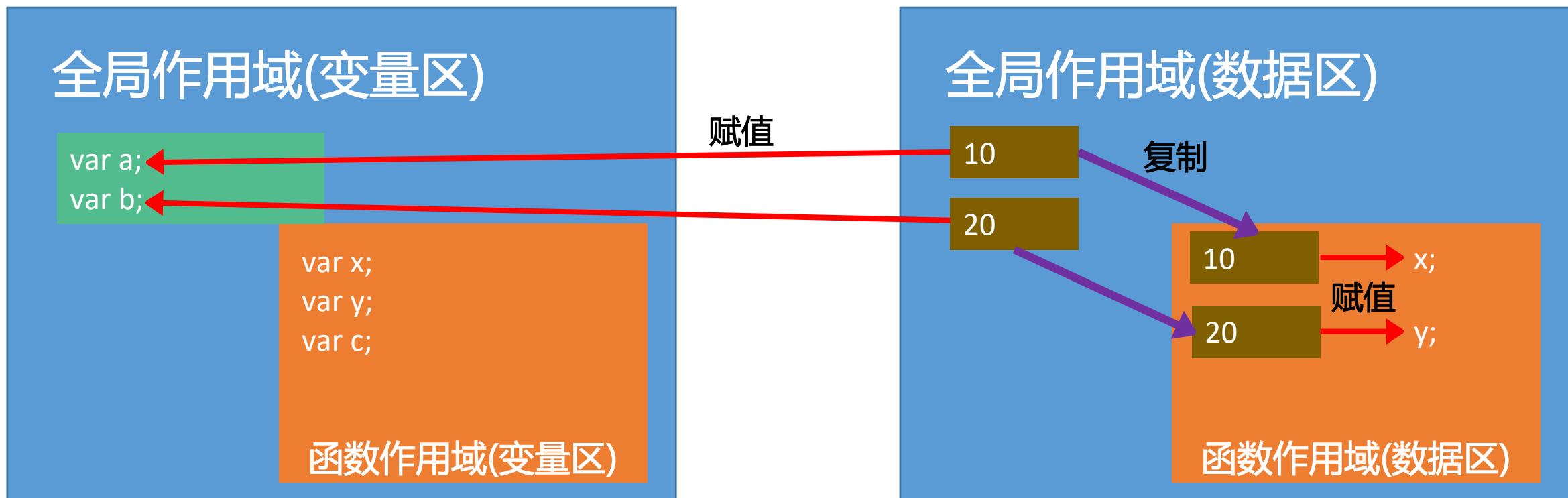
函数执行时

全局作用域(变量区)

```
var a;
var b;
```

函数执行完成后

函数参数的多种方式之标准传参

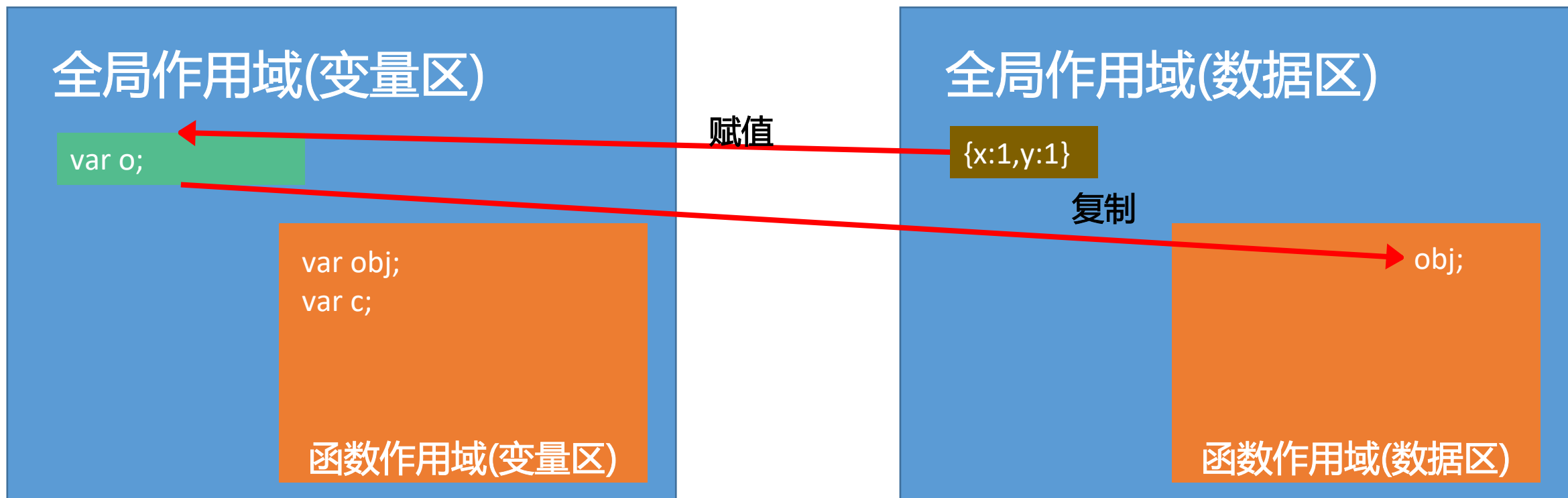


函数执行时

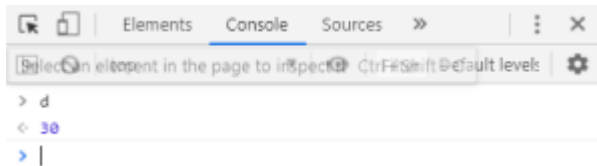
基本类型值的传递如同基本类型变量的复制一样

2:把函数外部的值复制给函数内部的参数, 就和把值从一个变量复制到另一个变量一样

函数参数的多种方式之标准传参



```
var o={a:10,b:20};  
  
function fn(obj){  
    var c=obj.a+obj.b;  
    return c;  
}  
  
var d=fn(o);
```



函数执行时

引用类型值的传递，则如同引用类型变量的复制一样

2:把函数外部的值复制给函数内部的参数，就和把值从一个变量复制到另一个变量一样

函数参数的多种方式之函数内基本类型值参数的操作

全局作用域(变量区)

```
var a;  
var b;
```

```
var x;  
var y;  
var c;
```

函数作用域(变量区)

赋值

全局作用域(数据区)

10

20

复制

10

20

赋值

x;

y;

函数作用域(数据区)

全局作用域(数据区)

10

20

x++; → 10++

y--; → 20++

函数作用域(数据区)

```
var a=10;  
var b=20;  
  
function fn(x,y){  
  x=x+1;  
  y=y+1;  
  console.log(x);//输出11  
  console.log(y);//输出21  
}  
  
fn(a,b);  
  
console.log(a);//输出10  
console.log(b);//输出10
```

	Elements	Console	Sources	Filter	Default level
11			demo.html:17		
21			demo.html:18		
10			demo.html:23		
20			demo.html:24		

3:如果传输的参数是基本类型(数字,字符串,布尔,null,undefined), 那么在函数内部对参数的操作**不会影响**外部变量的值。

函数内部的参数和函数外部的变量的值是相互独立的

函数参数的多种方式之函数内引用类型值参数的操作

全局作用域(变量区)

var o;

var obj;

函数作用域(变量区)

赋值

全局作用域(数据区)

{a:10,b:20}

复制

obj

函数作用域(数据区)

全局作用域(数据区)

{a:10 b:20}

obj.a="万章"

函数作用域(数据区)

```
var o={a:10,b:20};

function fn(obj){
  obj.a="万章";
  console.log(obj.a);//输出万章
}

fn(o);

console.log(o.a);//输出万章
```

4:如果传输的参数是引用类型(数组,函数,对象), 那么在函数内部对参数的操作**会**
影响外部变量的值.

函数内部的参数和函数外部的变量的值是同一个

函数参数的多种方式之arguments

ECMAScript 函数的参数与大多数其他语言中函数的参数有所不同。ECMAScript 函数不介意传递进来多少个参数，也不在乎传进来参数是什么数据类型。也就是说，即便你定义的函数只接收两个参数，在调用这个函数时也未必一定要传递两个参数。可以传递一个、三个甚至不传递参数，而解析器永远不会有什么怨言

```
var a=10;
var b=20;

function fn(){
    console.log(arguments);
}

fn(a,b);
```



arguments数组只有在函数作用域里面才会生效

原因是ECMAScript 中的参数在内部是用一个数组来表示的。函数接收到的始终都是这个数组，而不关心数组中包含哪些参数（如果有参数的话）

函数参数的多种方式之arguments

关键点:

1:函数在定义时没有规定参数, 函数在执行的时候也可以往里面传入若干个参数, 参数按照顺序存储在arguments数组中

```
var a=10;  
var b=20;  
  
function fn(){  
    console.log(arguments);  
}  
  
fn(a,b);
```

全局作用域(变量区)

```
var a;  
var b;
```

```
var arguments=[]
```

函数作用域(变量区)

默认情况下,函数都有一个数组

全局作用域(变量区)

```
var a;  
var b;
```

```
arguments=[a,b]
```

函数作用域(变量区)

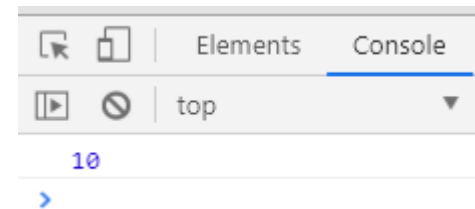
函数执行时,传参进来时,就把参数放进了这个数组

函数参数的多种方式之arguments

关键点:

2:函数在定义时规定了参数, 那么函数在执行的时候会在定义参数的变量同时把传入进来的参数存储在arguments数组中

```
var a=10;  
var b=20;  
  
function fn(x,y){  
    console.log(arguments[0]);  
}  
  
fn(a,b);
```



全局作用域(变量区)

```
var a;  
var b;
```

```
var x;  
var y;  
var arguments=[]
```

函数作用域(变量区)

默认情况下,函数都有一个数组

全局作用域(变量区)

```
var a;  
var b;
```

```
x=a;  
y=b;  
arguments=[a,b]
```

函数作用域(变量区)

函数执行时,传参进来时,就把参数放进了这个数组