JavaScript的定义：

* + - 脚本语言，解释性语言
    - 动态语言
    - 支持对象的语言.（不支持面向对象）

如何引入Javascript代码的方式：

1.在一对script标记中写代码

2.在html的行内编写代码

3.用script标签以及src属性引入指定js文件

常见输出信息方法：

alert()

document. write()

console.log( ) 这个是我们以后常用的（能自动把8进制16进制转化为10进制，能把科学计数法展开）

变量命名规则：

使用描述性的名称，比如: age, sum。

变量必须以字母、$、\_开头， 后面可以接字母、$、\_数字

变量也能以$和\_符号开头(不推荐这么做)（不能以数字开头，容易引起语义二义性）

变量不能使用关键字，保留字

变量名称对大小写敏感(y和Y是不同的变量)

提示: JavaScript 语句和JavaScript变量都对大小写敏感。

数据类型

原始类型

* + - 数字: number

能取到的最大值：Number.MAX\_VALUE

能取到的最小值：Number.MIN\_VALUE

无穷：Number.POSITIVE\_INFINITY

判断一个数 是否为为无穷大：isFinity()（无穷大返回False）

NaN:一般出现在期望是个数字，但是并不是个数的时候，表示一个底层的错误(not a number)

判断一个数是否为为NaN：isNaN()（NaN返回True）

* + - 字符串: string

Js是弱类型语言

字符串的拼接：

1. str=str1.concat(str2)
2. str=str1+str2

计算字符串的长度（中文一个字算一个字符）

console.log(‘字符串长度为：’,str.length)

indexOf(‘a’[,start]):用于确定某个字符串是否含有某个子字符串，以及它的起始位置.

如果没有找到匹配的字符串则返回 -1。str.indexOf(‘x’)

charAt(n):用于获得指定位置的字符

var ch = str.charAt(3)

substr(start,length)

substring(start,end):用于获得指定位置的子字符串（左闭右开）

var ch = str.substring(start,end)

console.log (typeof ch):判断ch的类型

* + - 布尔: boolean
    - undefined类型为undefined var a;只定义不赋值a:undefined

所有的undefined是一致的console.log(undefined==undefined)结果为true,

其他例如NaN要用isNaN(()函数判断、Infinity要用isfinity()函数判断

null==undefined undefined==undefined null==null

* + - null类型为object

引用类型

array, object, function

运算符

算数运算符

一元（目）运算符：++ --（只需要一个操作数）（在JS中++与+1无区别--与-1无区别）

二元（目）运算符：+ - \* /（需要两个操作数）

三元（目）运算符

对于加减，返回结果为infinity的情况

var num3 = 1E307 + Number . MAX\_ VALUE - Number . MAX\_ VALUE

console. log(num3);

//如果Infinity参加一般计算，结果永远是Infinity（Infinity-Infinity=NaN）

对于加减，返回结果为NaN的情况

1. var num = undefined + 1;

console. log(num);

1. var num4 ;

num=num4+1;

console . log(num);

赋值运算符：

= (表达式的值就是被赋值的值)

复合运算符：+= -+ \*= /= %=

变量+=表达式==>变量=变量+(表达式)

例：num/=2+1🡺num=num(2+1)

比较运算符：

数字的比较：> < >= <=表达式的值是boolean型的

字符串的比较：> < >= <=

编码: ASCII, 比较字符串的过程就是按照顺序比较ASCII的过程

十六进制:

0:0x30 a:0x41 A:0x61

1. 0x61 JS支持的，从C/C++继承来的

2. 61h JS不支持，从汇编那里来的

console.log( 'aaa'> ' aa'); 0x61 0x61 0x61> .0x61 0x61 0

两个字符串比较不足补0

若字符串与数字比较试图将两边都变成数字，如果遇到不能变成数字的情况，返回NaN,无论如何，表达式都是false;

1. console.log( 'aaa'> 7);（不可以转换，返回NaN，表达式结果为false）
2. console.log( '123'> 7);（可以转换表，达式结果为true）

（工作中不要使用这种隐式类型转换）

== === != !==

看上去不相等实际上相等：

undefined==null ‘4’==4 false==0 true==1 []==’’ {}=='[ object Object]' [5]==’5’

看上去相等实际上不等

NaN!=NaN ‘NaN’!=NaN undefined!=0 null!=0

=== !==与== !=的区别：=== !==是严格相等，严格不相等，不进行类型转换

转换规则

对象\字符串作为运算数，另一个运算数是数字，试图把对象\字符串转化为数字，失败的话返回NaN

对象作为运算数，另一个运算数是字符串，试图把对象转化为字符串

条件运算符：？：例：console.log( 25>4 ? 'A': 'B');

逻辑运算符：&& || ！

X && y:表达式的值，如果x等价于false,表达式的值是x;否则，表达式是y（若x为false，y不执行）

应用：if(data !== null){

return fn(data);

}else{

return data;

}

相当于return data && fn(data);

x || y:表达式的值，如果x等价于false, 表达式的值是y;否则,表达式是x（若x为true，y不执行）

源码框架经常使用，是一种JS特有的非常地道的写法

哪些东西等价于false? ：0, undefined,null, NaN,’’

应用: var event = e||window.event

1.如果e不等价于false, 返回e,

2.如果e等价于false, 返回window. Event

运算顺序：

高优先的优先于低优先级的运算符进行运算

同优先级的运算符从左到右

赋值运算符从右到左

加法，只要有字符串，就全都转化为字符串

例：得出58：var str=’’+5+8;

减法，试图把字符串变成数字，然后进行计算

若转化不成功，返回NaN

例：str = ‘a’ – 5;

console.log(str);

除法，两个整数相除可以得出小数（与c/c++不同）

例：4/0为infinity 0/0为NaN null/2=0 null/0=NaN

取余，例：4.5%3为1.5 0%0为NaN 4%0为NaN

用一个不是数的进行数字运算会得到NaN

例如：undefined+1 var a1;var a2=a1+1;console.log(a2)

如果Infinity参 加计算，结果永远是Infinity

例：console.log((1+ 1 + true + null +''CCC'+ 1 + 1 + true + null));

//1+1+true+null+'CCC'+1+1+true+null

//2+true+null+'CCC'+1+1+true+null

//3+null+'CCC'+1+1+true+null

//3+'CCC'+1+1+true+null

// '3CCC11truenull '



流程控制

顺序结构：从上到下，从左到右，顺序执行的过程

分支结构 if、if-else、if-else if-else、if .. else、switch-case（严格比较===）、三元表达式?:

循环结构while、do-while、for、for-in（for(var I in a)i表示下标）

数据类型转换

数据类型的判断

typeof（针对原始类型） instanceof(针对引用类型) a instanceof b

转换为数字

显示类型转化

库函数：Number();把一个能转换为数字的其他类型转换为数字

如：Number(true)=1 Number(‘3’)=3 Number(false)=0 Number(‘’)=0 Number(undefined)=NaN Number(‘xy’)=NaN Number(1.3.2)=NaN

Number(null)=0注意：null != 0

（Number是一个对象而number是一个原始类型）

库函数：parseInt(),把一个字符串按整数的方式“解析”，如果只有开头一部分可以，解析开头的一部分；如果解析不了，返回NaN

如：parseInt(‘10px’)=10 parseInt(‘a10’)=NaN parseInt(‘’)=NaN

对于16进制：parseInt(‘0xA0’)=160 parseInt(‘A0’,16)=160

对于8进制：parseInt(‘70’,8)=56

对于小数（可以当做取整来用但有隐患）：parseInt(‘1.5’)=1 parseInt(‘1.2E6’)=1

对于undefined、null、false、true：parseInt()=NaN

库函数：parseFloat：用于浮点数或者科学计数法

如：parseFloat(’10.00’)=10 parseFloat(‘1.2E3’)=1200 parseFloat(‘1.2aaa’)=1.2 parseFloat(‘10’)=10

isNaN不是数那么会自动调用Number

isNaN(NaN)=true isNaN(‘a10’)=true isNaN(undefined)=true

isNaN(null)=false isNaN(‘12’)=false isNaN(‘’)=false

Number:本质上是否能转化成数字，parseInt: 开头的部分看上去更像数字

隐式类型转换

加法，只要有字符串，就全都转化为字符串

减法，试图把字符串变成数字，然后进行计算

转换为字符串

显式类型转换：

String(),生成一个字符串String(null)=null字符串

toString(),成员函数，将数字转换为字符串 t.toString() 注：null、undefined不可以使用toString()等成员函数 t.toString(8):表示将数字转换为8进制在变为字符串



隐式类型转换：

var t = ‘’+2+undefined+null+false=2undefinednullfalse

转换为boolean值

显式类型转化：

库函数：Boolean()

例：Boolean(0)=false Boolean(undefined)=false Boolean(null)=false Boolean(‘’)=false Boolean(NaN)=false

隐式类型转换：

!0=true !1=false !100=false !!100=true

!undefined=true !null=true !NaN=true !’’=true

&&、||不进行隐式类型转换

数组（js的数组是弱类型）

数组的声明方式

1. 字面方式申请一个数组

var a = [];

数组初始化时，只初始化个数，具体值以后添加 var a = [1,,,,,,4,2]

1. 使用构造函数的方式

var b = new Array();

初始化：var b = new Array(5,4);

var b = new Array(4);（只有一个数字，表示初始化元素个数，是稀疏的）

数组的索引可以不是严格的数字：

如果向一个原来不在范围内的索引写入数据，那么这个位置会产生数据存储，同时调整length

数组的底层是类似key -value形式的mapper

数字，字符串都可以作为索引（如果不存在就写进去，存在就覆盖）

能够parse为数字的字符串会被转化为数字

length只会受到整数的影响.

访问数组中没有初始化过得的位置，第一不会报错，第二返回undefined

如何判断一个变量是数组？

console.log(b.constructor.name === ‘Array’);

console.log(b instanceof Array);

如何轮询数组？

1. for（操纵索引）：轮询的是数字的元素，依赖于length
2. for in（操纵索引）：轮询非稀疏的元素，不依赖于length
3. forEach（操纵元素）：只能轮询数字非稀疏的元素，不依赖于length， forEach方法中的function回调支持3个参数，第1个是遍历的数组内容；第2个是对应的数组索引，第3个是数组本身

例：var a1 = [1,2,3,4];

a1.forEach(function(x){

console.log(x);

});

split(‘x’): 把一个字符串以x分割成字符串数组

reverse(): 颠倒数组中元素的顺序

concat():把两个数组化为一个数组

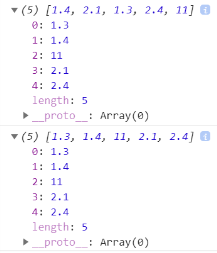
var a = a1.concat(a2);

join: 把数组串行化为一个字符串

var a = [1,2,3];

console.log(a.join(‘-’));

结果为：1-2-3

 sort: (缺省)把数据按照字符串的ASCII码进行排序，在原数组上操作，不生成新的数组

var a1 = [1.4,2.1,1.3,2.4,11];

console.log(a1);

a1.sort();

console.log(a1);

var a1 = [1.4,2.1,1.3,2.4,11];

 console.log(a1.join());

a1.sort();

console.log(a1.join());

对数字排序（第二个数传给a第一数传给了b，这个排序相当于是如果a-b<0就将x移到y的前面一位(比较函数需要返回一个小于0的值)（自己推测，不确定））

a-b:升序b-a:降序

console.log(a1. join (','));

a1. sort(function(a, b){

return a - b;

})

console.log(a1.join(','));

以下数组的函数都是使用数字的方式添加/删除元素，并且length有效

push: 在数组的尾部增加一个元素

pop:把数组尾部的元素去掉（会返回去掉的元素）

shift:把数组的头部的元素去掉（会返回去掉的元素）

unshift: 在数组的头部增加一个元素

应用：

1. 把数组作为一个尾进尾出的栈(后入先出)
2. 把数组作为一个头进头出的栈(后入先出)

数组去重：

思路1,我们准备个新的数组， 轮询原数组元素，如果该元素不存在与新数组中,加入新数组

如果已经在新数组中，不加。

思路2,我们对数组进行排序，相同的元素一定 是挨在一起的（将原数组排序后的元素与新数组最后一个元素比较，不相同就放进去）

引用类型的内存分配

原始类型与引用类型的区别：

原始数据类型变量的内存在栈中分配，数据存在栈中

引用数据类型变量的内存在栈(stack)中分配，实际的内存在堆（heap）中分配，变量的内存中只存储了内存地址( 在堆中的)。

拷贝时，原始数据类型直接拷贝数据，引用数据类型拷贝内存地址

计数器：引用数据类型，每有一个变量存储了它的地址，那么计数器加一，反之减一。计数器为0时，堆中的引用数据被释放。

数组的几个成员函数

slice:从数组中截取一段元素，拷贝，生成新数组，原数组不受影响

a.slice(start):从下标为start开始到数组结束

a.slice(start,end)：从下标为start开始到下标为end-1结束（左闭右开）

splice: 从数组中截取一段元素，原数组受影响

a.splice(start):从start到结尾都截走

a.splice(start,length):从start开始截取长度为length的一截

a.splice(start,length,item1,item2,iten3…): 从start开始截取长度为length的一截,

将item1,item2,iten3…从下标为start处插入到a中

delete:删除数组中的元素，然后删除对应的索引

var a1 = [1,2,3,4,11];

 delete a1[1];

console.log(a1);

数组的性质：

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

a.length=2;

console.log(a);=[1,2]

**有几种清空数组的方法？**

var a = [0,1,2,3,4];

1. a.length = 0;
2. a.splice(0)
3. pop、shift

while(a.length>0){

a.pop();

}

1. a=[];（存在隐患，若b=a那么用这种方法b不会清空）

高维数组：

一维数组

二维数组martrix(矩阵)

三维数组cubic(立方体)

函数：

定义：可以重复运行的子程序

声明：function<函数名>

函数可以有形参,也可以没有

形参是函数定义过程中参与运算的变量

实际参数，是函数调用时传递给函数的参数

函数可以有返回值,return,如果没有return,那么函数返回undefined

如果return的返回值不只一行，返回值必须用()括起来

函数声明：function<函数名>(){}

函数表达式

var x = function<函数名> (){}（函数名不起作用，不会像声明一样是一个变量，所以可以写成：

var x = **function (){}** 这是一个表达式，表达式的返回值是一个函数）

ff(1,function(){},2)

函数表达式不会污染名字空间,这也是为什么很多jS库使用函数表达式

立即执行函数：()表示对函数进行调用，前面的是一个函数表达式，返回值是一个函数

作用：

* + - * 1. 不必为函数命名，避免了污染全局变量
        2. 立即执行函数内部形成了一个单独的作用域，可以封装一些外部无法读取的私有变量
        3. 封装变量

**(function<函数名可省略>(){**

**var a = 2;**

**var b = 3;**

**console.log(a + b);**

**})**();

原始类型和引用类型作为函数参数传递的区别

原始类型：值传递 引用类型：地址传递

函数的嵌套：

任意一个函数，可以调用其外部定义的函数，也可以调用一个自己内部定义的函数

递归：

函数自己调用自己

练习:有一行台阶,N阶,N是正整数，可以一步一阶,一步两阶,一步三阶，一共有多少种走法?

函数名叫step

N:1 => 1,

N:2=>一步两阶;先走一步一阶，step(1)=>1+1=2,

N:3=>一步三阶;先走一步一阶，step(2);先走一步两阶，step(1)=>1+2+1=4;

N:4 =>一步三阶，step(1); 一步两阶，step(2); 一步一阶; step(3)=>1+2+4=7;

N:n:=> : step(n-3) + : step(n-2).+ step(n-1);

在写函数时：

定义输入形参的有效形式，并且想清楚如果形参无效，那么应该用什么样的策略处理?

策略一，错了，就返回。

策略二，如果错了，能处理多少就处理多少

设定响应的测试用例TDD Test -Driven Development

标识符的命名规则

小驼峰:单词串写，第一个单词所有字母小写，以后的单词第一个字母大写，其他字母小写

大驼峰:单词串写,每个单词第一个字母大写,其他字母小写=>构造函数

匈牙利: window API用的，i\_ 小驼峰fl\_ lpsz ====不要用

面向对象思想

三种不同的语言类型

机器语言+汇编语言

面向过程的语言: C语言

面向对象的语言: C++，Java, C#

面向对象的特征

封装性:一些列的数据和方法封装起来

继承性:父类和子类

多态性: ( 里氏代换原则)任何父类可以正常被处理的代码，子类也可以被处理

JavaScript语言不是严格的面向对象的语言，它是支持对象的一门语言

支持数据、方法、继承、多态

对象Object

对象的创建

1、字面量的方式（推荐） var obj = {};

2、系统构造函数的方式（系统构造函数首字母大写） var obj = new Object();

对象的构成

数据、属性

数据可以是任意的原始类型

var obj = {}; obj.page = 300; obj.name = ‘Book Name’;

数据也可以是一个对象

var author = {}; author.firstName = ‘Li’; author.familyName = ‘Wang’; obj.author=author；

成员函数

obj.query = function(){}

访问数据及成员函数

console.log(obj.name); console.log(obj.query())

也可以写成以下形式（在不确定要打印什么的时候可以用）

var b = ‘name’; var a = ‘query’;

console.log(obj[b]); console.log(obj[a]());

对象的属性或者成员函数的增删改查

增加

obj.b=’bbb’;

1、对象中的属性可以不是标识符

例：增加一个变量名1

obj.[‘1’]=’xx’

2、没有定义的变量直接赋值，会产生怎样的后果（要尽量避免）

function f2(){

aaa=5;相当于写window.aaa = 5;

}

f2();

console.log(aaa);运行结果为5

为什么？

(1)有个对象叫window

(2)所有在脚本中(没有在函数的范围内)通过var定义的变量，都挂在window.上

(3)在任意位置,一个变量赋值都可以看做是在window.上寻找这个变量

删除：delete obj.a;与数组相似

修改：obj.a=’bbb’;

查：

1. in

console.log(‘a’ in obj)

1. hasOwnProperty()函数

console.log(obj.hasOwnProperty(‘a’))

枚举

for(var p in obj){

console.log(p);

}

将obj中的属性全部输出，

for...in循环出来的属性，除了通过原型链继承的属性不是可枚举，其他都是可枚举属性。

对其进行判断：

obj.propertyIsEnumerable(‘toString’)判断toString是不是可枚举的（即判断是不是自己的）

函数

函数的调用方式

1、普通调用

function f1(x){

console.log(x);

}

f1('hello world!');

2、对象的成员函数

var obj = {

f2: function(x){

console.log(x);

}

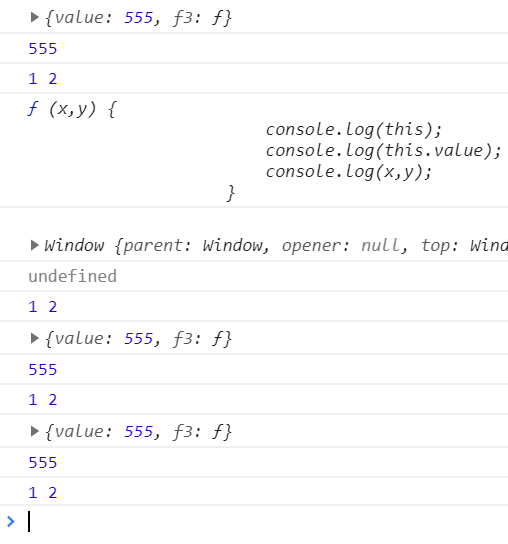
}

obj.f2('I feel cool');

3、构造函数

var a = new Array();

间接调用 使用call或者apply, 主要区别在于可以绑定this

 var obj = {

value: 555,

f3: function (x,y) {

console.log(this);

console.log(this.value);

console.log(x,y);

},

}

obj.f3(1,2);

console.log(obj.f3);

var f4 = obj.f3;//相当于window.f4 = obj.f3;

f4(1,2);//window.f4(1,2);

f4.call(obj,1,2);

f4.apply(obj,[1,2]);

arguments对象

形参与实参:如果函数支持实参数小于形参数，那么在函数中需要使用arguments.length或者对形参判断是否等于undefined来判断实参数

两种情况

1.形参多,但是调用的时候，某些实参可以省略

2.形参数量有限，实参会给如不定个数的更多参数。

需要知道形参的个数: <函数名>. length

需要知道实参的个数: arguments. length

形参和实参是不同的变量,但是是系统自动绑定的

var f = function (a,b,c) {

console.log(a,b,c);

}

f(1);

结果为1 undefined undefined

var f = function (a,b,c) {

console.log(a);

console.log(arguments.length,arguments[1]);

}

f(1,2,3);

结果为1 3 2

当实参数等于或多于形参数，实参形参自动绑定

var f = function (a,b,c) {

console.log(c,arguments[2]);

c = 5;

console.log(c,arguments[2]);

arguments[1] = 9;

console.log(b,arguments[1]);

}

f(1,2,3);

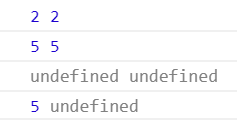
结果为:3 3

5 5

9 9

当实参数小于形参数，与形参相匹配的实参自动绑定，不匹配的不绑定

var f = function (a,b,c) {

 console.log(b,arguments[1]);

b = 5;

console.log(b,arguments[1]);

console.log(c,arguments[2]);

c = 5;

console.log(c,arguments[2]);

}

f(1,2);

函数表达式的递归：arguments.callee表示函数本身（不用函数名的方法）

var fact = function(n){

if(n <= 1){

return 1;

}else{

return n\*arguments.callee(n-1);

}

}

console.log(fact(5));

如何判断一个变量是函数

1. if(typeof f == ‘function’)

typeof一般用于原始类型

null是原始类型，但是typeof null 是object

函数是引用类型，但是typeof f是function

1. if(f.constructor.name == ‘Function’)

（null 和 undefined 无constructor）

练习[js\myMath.html](js/myMath.html)

工厂模式:简单工厂（将整个对象及初始化全部封装在一个方法中）

简单工厂模式属于类的创建型模式，又叫静态工厂方法模式。通过专门定义一个工厂类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

设计模式(Design Pattern):

谁? Gang of Four“四人帮”1994

主张?模式，能够更好的帮你解决一些你面对的问题

针对接口编程;更多的使用组合而不是继承.

构造函数

构造函数内部原理

　 ①.在函数体内部隐式的加上 this = {}

    ②.执行this.xxx = xxx;

    ③.隐式的返回this

【需要特别注意：构造函数需要发现new关键字之后才会有构造函数的上面的三步走原理，否则构造函数就和一个普通的函数一样，没啥区别！】

每次以相同的方式来构造对象;如果改变了构造函数，那么所有此后产生的对象都跟着改变，

特点:大写字母开头

包装类

Number

String

Boolean

一个构造函数，理论上不应该有返回值

如果有了呢?

如果返回原始类型，调用的时候没有用new,函数会把现在要返回的值返回，如果用了,返回this

如果返回引用类型，无论调用时是否用了new,函数会把现在要返回的值返回。(自己写的都是引用类型)

原始值不能有属性和方法，当给它们添加属性的时候系统会自动进行包装类并销毁。

包装类本身内部是基本类型,但是其本身是对象object

var n1=123;

var n2 =new Number(123) ;

typeof n1 = number typeof n2 = object

var str =' 1234' ;

console. log( str. concat( '56'));

str是原始类型为什么可以调用成员函数？

他其实相当于：

var str =' 1234' ;

var strTmp =new String(str);

console. log( strTmp. concat( '56'));

运行完后strTmp被销毁

所有这些都可以调用它对应的包装类上的成员函数

var str = "abcd";

str.a = 2;

new String(str).a=2 delete

console.log(str.a); undefined

var str = "abcd";

str.length = 2;

console.log(str.length); 2

需要添加的属性为包装类上的属性

预编译（两种情况：1、第一次运行这个脚本的时候2、函数调用的时候）（JS-006）

JS引擎处理脚本的过程

预编译（第一次\前置 扫描）将所有变量和函数的声明（不是函数表达式）

脚本：

创建全局对象GO(Global Object) (window)(上下文)

加载脚本文件

预编译:

找出所有的变量声明，按照变量名加入全局对象，如果已经存在，忽略，只保留第一个

找出所有的函数声明，按照函数名加入全局对象，如果已经存在同名变量或者函数，替换。

非声明不予理睬

解释执行（第二次扫描）

脚本的预编译

1. 没有var的变量，都不是变量声明，全部都认为是window的全局变量，不参与预编译
2. 即使aa在函数中,aa也是全局变量,是运行时生效，不是预编译时生效

console.log(aa);

function test(){

aa = 5;

}

会报错

1. 脚本中,所有的变量声明，在脚本的预编译阶段完成,所有变量的声明与实际的书写位置无关（变量的使用只能在初始化之后）

console.log(a);

var a=5;

console.log(a); 结果为

undefined

5

1. 脚本中，所有的函数声明,在脚本的预编译阶段完成,所有函数的声明与实际的书写位置无关（函数在任意位置定义，在任意位置都可以运行）

console.log(f);

function f(){

console.log( 'haha');

}结果为

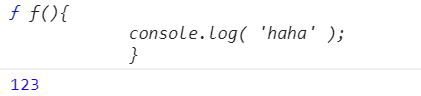
ƒ f(){

console.log( 'haha');

}

1. 脚本中,如果变量与函数同名，那么函数将覆盖变量

console.log(f);

var f = 123;

console.log(f);

function f(){

console.log( 'haha' );

}

1. 脚本中，如果变量与函数同名,函数可以覆盖变量,但是变量无法覆盖函数

console.log(f);

function f(){

console.log( 'haha');

}

var f = 123;

结果为ƒ f(){

console.log( 'haha');

}

1. 脚本中，如果有多个函数同名，最后声明的函数将覆盖所有前面的同名函数声明

并且,参数个数是忽略的，也就是说,JS压根不支持重载



函数调用的预编译

创建活动对象AO（Active Objext）（上下文）

预编译：

scope chain（作用域链）

初始化arguments

初始化形参，将arguments中的值赋值给形参

找出所有的变量声明，按照变量名加入A0，如果已经存在，忽略。

找出所有的函数声明，按照函数名加入A0，如果已经存在同名变量或者函数,替换。

this初始化

解释执行

1. 函数中，所有变量声明，在函数的预编译阶段完成，所有变量的声明与实际的书写位置无关.
2. 函数中，所有函数声明，在函数的预编译阶段完成，所有函数的声明与实际的书写位置无关，
3. 函数中，如果变量与函数同名，那么函数将覆盖变量
4. 函数中,只有函数能够覆盖变量，变量无法覆盖函数
5. 函数中，如果有多个函数同名,最后声明的函数将覆盖所有前面的同名函数声明并且,参数个数是忽略的，也就是说,JS压根不支持重载
6. 当函数预编译后，遇到需要访问的变量或者函数，优先考虑自己A0中定义的变量和函数如果找不到,才会在其定义的上一层A0中寻找，直至到G0,再找不到才报错.



练习1

function test(x, x){

console.log(x);// function( )

x=5;

console.log(arguments); // [12, 5]

function x(){

console.log('haha');

}

}

test(12,13);

// GO.test .-> function()

// test.AO. arguments ->[12,13]->[12,function()]->[ 12,5]

// test.A0.x -> 12 -> 13 -> function() -> 5

练习2

b = 'cba';//不算声明，不用管

function a(a, a){

console.log(a); // function()

console.log(b); // undefined

var b='abc';

a();

function a(){

console.log(a); // function()

console.log(b); // ' abc'

}

}

a(5, 10);

//GO.a -> function( )

//a .A0. arguments - >[5,10) ] ->[5，function( ) ]

//a.A0.a -> 5 -> 10 -> function( )

//a.AO.b -abc

//a.a.AO

练习3

var x = 1, y=z=0; ========🡺 // var x;

function add(n){ //x = 1;

return n=n+1; // var y;

} //z=0;

y = add(x); //y=Z;

function add(n){

return n=n+3;

}

z = add(x);

结果：x = 1 y = 4 z = 4

作用域链：Scope Chain（特点：为JS所做的技术准备）

作用域链的作用：以另一种方式理解预编译

作用域Scope：有效范围

函数作用域

外部对内部可见

内部对外部不可见

内部优先

作用域链是函数级别的

if的{ }块无法影响函数的作用域

var scope=g;

if(true) {

console.log(scope); // g

var scope =‘ l' ;

console.log(scope); // l

}

console.log(scope); // l

循环的{}无法影响函数的作用域

for(var i = 0;i<a.length;i++)i是全局变量

执行环境和作用域链

执行环境(execution context)(JS中就是作用域链)，定义了执行期间可以访问的变量和函数。

全局执行环境

Global Object (window)

从见到JS代码开始创建

到网页关闭时销毁

函数执行环境

Activation Object

从函数调用开始创建

到函数调用结束时销毁（AO引用计数为0）

作用域[scope]], 每个函数都有。

作用域是私有属性，只能由JS引擎访问

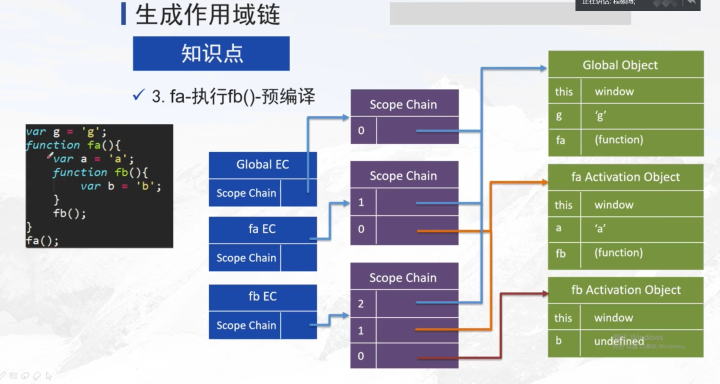
作用域链，是AO和GO构成的链

所谓执行环境，就是根据作用域链依次查找变量和函数:

找到即停;

全部找完无果，报错。

作用域链每个函数都有



生成作用域链

每个函数在定义(函数声明\函数表达式)时会拷贝其父亲函数的作用域链;

在函数被调用时，生成AO然后将AO压入作用域链的栈顶。

函数多次调用时，是产生相同的AO还是不同的AO?不同

function fa(){

console.log(a);

var a = 100;

a++ ;

console.log(a);

}

fa();

fa();

fa();

函数递归调用时，是产生相同的AO还是不同的AO?不同

function fa(x){

if(x > 2){

var a = 100;

a++ ;

fa(x-1);

}

return 0;

}

fa(3);

with：生成新的with variable object， 放在作用域链表顶端。（自己写尽量不要用）

1、

function f(p1, x){

with(p1){//优先从p1中找x

//5000行代码

console.log(x);

}

}

var p={

value1: 3,

value2: 5,

//如果加上x:25,结果为25

}

f(p, 5);

2、

var p={

values: {

value1: 1

}

}

with(p.values) { //使用with后,即使对象改变依然能访问p.values 里的value1

// 5000行

p.values = {

value2: 15

}//p.values对象的地址发生改变，

// 5000行

console . log(value1); //1

}

console. log(p. values . value1); //undefined

console. log(p. values . value2); //15

作用域链的应用:（问原理的时候，不是因为效率就是因为安全）

效率:

尽量少使用靠近上层的变量，多使用自己的局部变量

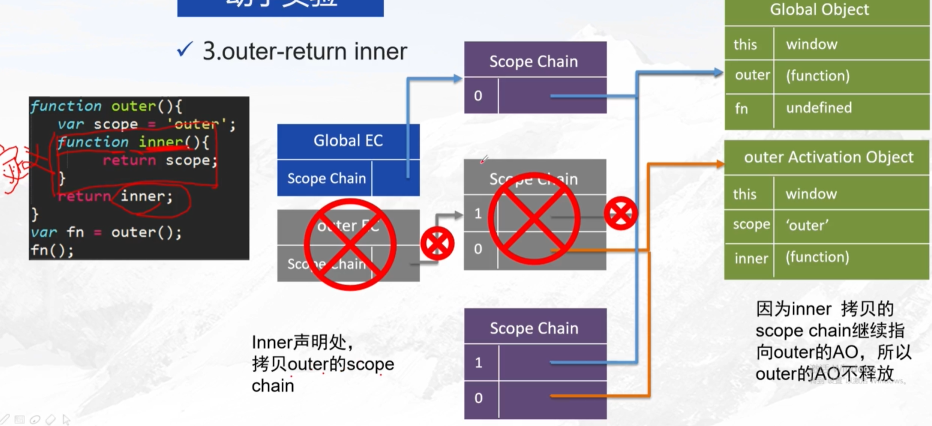
重名，容易出错:

尽量减少不同层次函数使用相同的变量名

避免函数名与变量名一样。

函数退出以后AO是否一定被释放?

不一定，当函数退出后其子函数没退出，引用计数不为0不释放



闭包<https://zhuanlan.zhihu.com/p/54666882> [https://blog.csdn.net](https://blog.csdn.net/huakaiwuxing/article/details/78968642)

函数的AO通过scope chain相互连接起来，使得函数体内的变量都可以保存在函数的AO，这样的特性称为“闭包

闭包的危险:

闭包会造成原有AO不释放，产生内存泄漏

闭包的应用:

实现公有变量

缓存存储结构

封装，实现属性私有化

模块化开发

防止污染全局变量

应用

特点：封装变量，只暴露函数

1. 共有变量：一个变量一个函数

function createCounter( ){

var count = 0;

function counterAdd( ) {

count++ ;

console. log( count);

return count;

}

return counterAdd;

}

var counter = createCounter();

counter( ) ;//1

counter( ) ;//2

counter( );//3

1. 缓存存储结构：多个变量多个函数

function counter(){

var count = 0;

function countAdd(){

count++;

console.log(count);

}

function countAddTwo(){

count += 2;

console.log(count);

}

function clear(){

count = 0;

console.log(count);

}

return [countAdd,countAddTwo,clear];

}

var cou = counter();

cou[0]();

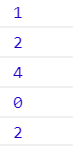
cou[0]();

cou[1]();

cou[2]();

cou[1]();

结果为：



1. 模块化

function creatCount(x){

var count = 0;

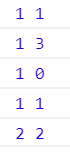
var counter = {

countAdd: function(){

count++;

console.log(x,count);

},

 countAddTwo: function(){

count += 2;

console.log(x,count);

},

clear: function(){

count = 0;

console.log(x,count);

},

};

return counter;

}

var cou = creatCount(1);

cou.countAdd();

cou.countAddTwo();

cou.clear();

cou.countAdd();

var cou1 = creatCount(2); //不会影响cou里面的count

cou1.countAddTwo();

内置对象

常量

console.log(Math.PI);

console.log(Math.E);

console.log(Math.LN2);

console.log(Math.LN10);

console.log(Math. LOG2E ) ;

console.log(Math. LOG10E) ;

console.log(Math.SQRT2);

console.log(Math.SQRT1\_ 2);sqrt(1/2)

成员函数

绝对值

console.log(Math.abs(10)) ;

console.log(Math.abs(-5));

三角函数 弧度

弧度--角度 2\*PI—360deg

console. log(Math.sin(Math.PI/3)); //sin60度

console. log(Math.cos(Math.PI/3));

console. log(Math.tan(Math.PI/3));

反三角函数 弧度值 sin(arcsinx)=x

console.log(Math.asin(1));

console.log(Math.acos(1));

console.log(Math.atan(1));

console.log(Math.atan2(1, 1)); //atan2(纵坐标,横坐标)

四舍五入

console . log(Math.round(13.4));

console. log(Math.round(13.5));

取整

console. log(Math. floor(12.7)); //向下取整

console. log(Math.ceil(12.7)); //向上取整

找到最接近的单精度数

console. log(Math.fround(13.4));

指数与对数 都以e为底

console. log (Math.exp(2));

console. log (Math. log(25));

任意数的指数

console. log(Math. pow(10, 2));

任意对数：如**log101000**

console. log(Math.log(1000)/Math.log(10));

根号

console. log(Math. sqrt(4));

求最大值最小值

console.log(Math.max(1,2,3,5,44));

console.log(Math . min(1,2,3,5,44));

随机值[0~1)之间的平均分布的小数

console . log(Math. random());

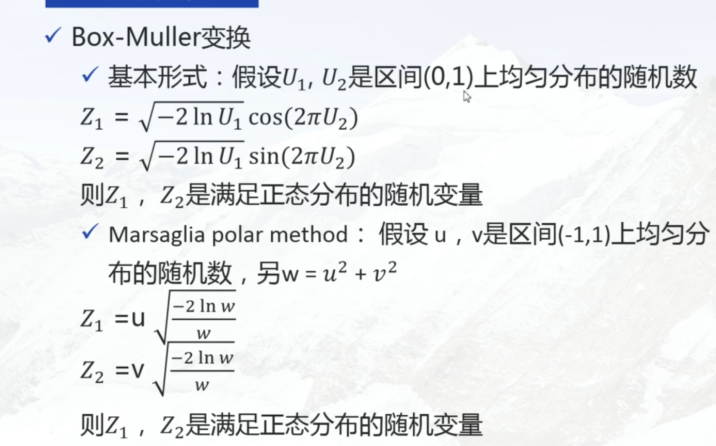
产生离散的数平均分布? 1 ~6

console. log(Math. floor (Math. random( )\*6+1)

产生离散的数3,5,7,平均分布（n:0、1、2 2n+3=3,5,7）

console.log(Math.floor(Math.random()\*3)\*2+3);

如何生成正态分布



练习：产生一个随机颜色

function randomHex( ){

var rdm = Math. random() \* 256;

rdm = Math . floor(rdm);

return rdm . toString(16);

}

function randomColor(){

//随机的R,G, B,两位的16进制数

//1.生成0~255之间的随机数，离散的

// 2.变成16进制的字符串

var r = randomHex( ) ;

var g = randomHex( ) ;

var b = randomHex( ) ;

console.log(r, g, b);

// 3.拼接字符串

return ' background-color: #' + r+g+b+';';

}

console. log(randomColor());

String:

// string是原始类型，String对象

var str =abcde ' ;

//str[0] = 'A';错的:因为包装类(将str包装为一个引用类型,在包装类上str[0] = 'A'然后销毁)

str = ' Abcde' ;

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

console. log(str[i]);

}

字符串的常用方法：

charCodeAt() 方法可返回指定位置的字符的 Unicode 编码

string.length:字符串的长度

string.charAt(索引) = []

String.fromCharCode(ASCII值，可以多个)变成字符串

var str = String.fromCharCode(102,23,65);

string.concat(拼接字符串，可以多个)

string.indexOf()查找字符串(存在返回1，没有返回-1)

string.lastIndexOf()查找字符串，从后往前查

string.replace(要替换的,替换成的)

var str = ‘hello world’;

console.log(str.replace(‘o’,’AA’)); //将第一个’o’替换为’AA’

console.log(str.replace(/o/g,’AA’)); //正则表达式,将所有’o’替换为’AA’

string.slice()截取字符串，但是 不影响原来的字符串（没有splice）

string.split()基于某种分隔用途的字符，把字符串切割成字符串数组

string.substr(start,length)截取字符串

string.substring(start,end)截取字符串

string.toLowerCase()全部转小写

string.toUpperCase()全部转大写

string.trim()去掉两边空格或者等价的 例如\t制表符

string.trimLeft() string.trimRight()单独去掉左边或右边空格

数据通信,只能依靠字符串,我们关心字符串的格式

对象->字符串(叫串行化) 字符串->对象(叫解析)

JSON 是一种格式 与对象或者数组是一致的

var strJSON ='{\

"name": "ABC",\

"age": 25\

}'

string->object

//1.eval() 有安全性问题（将一段字符串当做javascript来运行）

eval() 函数使用的是 JavaScript 编译器，可解析 JSON 文本，然后生成 JavaScript 对象。必须把文本包围在括号中，这样才能避免语法错误：由于json是以”{}”的方式来开始以及结束的，在JS中，它会被当成一个语句块来处理，所以必须强制性的将它转换成一种表达式。

var obj1 = eval('(' + strJSON + ')');

//2.JSON.parse()

var obj2 = JSON.parse(strJSON);

//3.jQuery.parseJSON(string)

//需要jquery-3.5.1.js

var obj3 = jQuery.parseJSON(strJSON);

object->string

//1.JSON.stringify()

var obj = {

name: 'ABC',

age: 25

}

var strJSON = JSON.stringify(obj);

console.log(strJSON);

//2.jQuery.toJSON()

//需要jQuery.json.min.js

strJSON = jQuery.toJSON(obj);

console.log(strJSON);

练习1:把一张九九乘法表(需要用程序生成)放入JSON中,然后解析

var obj={

line1: [ '1\*1=1'],

line2: [ ' 1\*2=2', 2\*2=4' ],

line3:[ '1\*3=3', '2\*3=6', '3\*3=9 '],

…

}

练习2:把多个学生的成绩单放入JSON,然后解析

学生的信息，包括: name, birthday, gender, class, {course, score}

Date 日期和时间

具体知识点：[js\Date.html](js/Date.html)

定时器，异步

setTimeout调用时只是把任务放置到队列中，然后返回,等到规定的时间到了, 会再调用回调函数

时间=程序运行的时间+setTimeout的时间

setInterval在指定的延时间隔重复执行。

区别：

setTimeout会保证在指定好的延时时间后执行，但是setInterval则不会这样。 如果function中的代码有耗时载操作，那么使用setTimeout方法递归，则可能会增加总递归的时间。 而使用setInterval方法，如果程序中耗时比延时间隔长，则会立刻回调函数。

[js\定时器.html](js/定时器.html)