# 基本语法

## JavaScript组成

JavaScript是一种基于对象和事件驱动编程语言。一个完整的JavaScript应该由三个不同的部分组成：

核心（ECMAScript)，提供核心语言功能  
浏览器对象模型(BOM)，提供与浏览器交互的方法和接口  
文档对象模型(DOM)，提供访问和操作网页内容的方法和接口

### 核心（ECMAScript)

ECMAScript是一种语法标准，提供核心语言功能

ECMAScript与Web浏览器没有依赖关系。ECMA定义的只是这门语言的基础,而在此基础之上可以构建更完善的脚本语言。我们常见的Web浏览器只是ECMAScript实现可能的宿主环境之一。宿主环境不仅提供基本的ECMAScript实现,同时也会提供该语言的扩展,以便语言与环境之间对接交互。而这些扩展——如DOM,则利用ECMAScript的核心类型和语法提供更多更具体的功能,以便实现针对环境的操作。其他宿主环境包括Node（一种服务器端JavaScript平台）

ECMAScript规定了这门语言的下列组成部分：语法、变量和数据类型、运算符、逻辑控制语句、关键字、保留字、对象

### 浏览器对象模型(BOM)

BOM：Browser Object Model（浏览器对象模型）  
提供与浏览器交互的方法和接口

### 文档对象模型(DOM)

DOM：Document Object Model（文档对象模型）  
提供访问和操作网页内容的方法和接口

## 在Web网页里添加JavaScript

### 方法一

把js语句写在<script></script>里面，

<script>

...javscript语句...

</script>

<script>元素没有任何必须设置的属性（在HTML5里，type属性是可选的）。如果在HTML4.x或XHTML页面里添加JavaScript，就需要使用type属性了。

### 方法二

把JavaScript代码保存到单独的文件（后缀是.js），然后利用<script>标签的src（源）属性把这个文件包含到页面里

<script src="mycode.js"></script>

注意：外部文件里面不能使用<script></script>标签，也不能使用任何HTML标签，只能是纯粹的JavaScript代码。

# 词法结构

词法结构就是一套基础性规则，用来描述如何使用这门语言来编写程序

## 变量

ECMAScript的变量是松散类型的，所谓松散类型就是可以用来保存任何类型的数据。换句话说，每个变量仅仅是一个用于保存值的占位符而已。定义变量时要使用var操作符（注意var是一个关键字），后跟变量名（即一个标识符），如下所示：var message;

### 什么是变量

变量是一个值的符号名称，可以通过名称来获得对值的引用。变量就是值的引用。

### 为什么要使用变量

值太长，每次书写不方便；将值保存起来以备将来使用，也方便以后多次使用

### 如何声明变量

应当先声明变量，再使用变量；变量是使用关键字var或let来声明的，后跟变量名(即一个标识符)  
可以一次声明多个变量，用逗号隔开；可以在声明变量的同时给变量赋值

var message = "hi",

found = false,

age = 29;

这个例子定义并初始化了3个变量。同样由于ECMAScript是松散类型的，因而使用不同类型初始化变量的操作可以放在一条语句中来完成。虽然代码里的换行和变量缩进不是必需的，但这样做可以提高可读性。

在严格模式下，不能定义名为eval或者arguments的变量，否则会导致语法错误。

可以在修改变量的同时修改值的类型,如下所示:

var message = "h1";  
message = 100; // 有效，但不推荐

在这个例子，变量message一开始保存了字符串值”h1”, 然后该值又被一个数字值100取代。虽然不建议修改变量所保存值的类型，但这种操作在ECMAScript中完全有效。

JS中的变量可以用来保存任何类型的数据，变量的数据类型由其存储的值的类型决定；使用变量名等效于直接使用变量中存储的数据

注意

使用var或let声明了变量，但是没有给变量指定初始值，它的初始值就是undefined  
给一个未声明的变量赋值是一个不好的习惯，在严格模式下也会导致抛出ReferenceError错误。因此，应当始终使用var或let来声明变量  
使用既没有声明也没有赋值的变量，会报错

### var，let和const区别

var会变量提升；let不会变量提升，  
var支持重复声明；let 不支持重复声明（声明变量之后就不能再次声明它。虽然可以给变量重新赋值，但不能重新声明变量，否则将导致编译阶段错误。）  
var没有块级作用域；let有块级作用域(if ，for循环，do...while的花括号)  
var 兼容IE7，let兼容IE11

## 常量

使用const关键字声明的是常量，

特性同let:不支持预解析，也就是不支持声明提前；不支持重复声明

区别是一旦声明，常量的值就不可以改变  
每个通过const声明的常量必须进行初始化，也就是必须赋值，否则会抛出语法错误

## 区分大小写

ECMAScript中的一切（变量、函数名和操作符）都区分大小。这就是说, 变量名test和变量名Test分别表示两个不同的变量，而函数名不能使用typeof, 因为它是一个关键字, 但typeOf则完全可以是一个有效的函数名

## 标识符

标识符就是指变量、函数，属性的名字，或者函数的参数

标识符可以按照下列格式规则组合起来的一或者多个字符  
由字母、下划线、美元符号和数字组成，且首字符不能是数字  
不能把关键字，保留字、true、false和null，以及预定义的全局变量和函数作为标识符

为了与ECMAScript内置的函数和对象命名格式保持一致，推荐使用驼峰式命名法：第一个字母小写，剩下的每个单词的首字母大写，如：firstSecond , myCar , doSomethingImport

## 关键字和保留字

ECMAScript描述了一组具有特定用途的关键字, 这些关键字可用于表示控制语句的开始或者结束, 或者用于执行特定操作等。按照规则, 关键字也是语言保留的, 不能用作标识符。

ECMAScript还描述了另外一组不能用作标识符的保留字。尽管保留字在这门语言中还没有任何特定的用途, 但它们有可能将来被用作关键字

## 语句

ECMAScript中的语句以一个分号结尾；如果省略分号，则由解析器确定语句的结尾

<script>  
 var *sum* = a + b *//即使没有分号也是有效的语句---不推荐* var *diff* = a - b; *// 有效的语句---推荐*</script>

JS使用分号将语句分隔开；为了方便阅读，我们建议每条JavaScript的每条语句独占一行，且都加上分号

虽然语句结尾的分号不是必需的，但建议任何时候都不要省略它。加上这个分号可以避免很多错误, 开发人员也可以放心地通过删除多余的空格来压缩JS代码。另外，加上分号也会在某些情况下增进代码的性能, 因为这样解析器就不必再花时间推测应该在哪里插入分号了

可以使用C风格的语法把多条语句组合到一个代码中, 即代码块以左花括号({)开头, 以右花括号(})结尾:

<script>  
 if (test) {  
 test = false;  
 alert(test);  
 }  
</script>

虽然条件控制语句(如if语句)只在执行多条语句的情况下才要求使用代码块, 但最佳实践是始终在控制语句中使用代码块——即使代码中只用一条语句, 例如:

if (test)

alert(test); // 有效但容易出错, 不要使用

if (test) {  
 alert(test); // 推荐使用  
}

在控制语句中使用代码块可以让编码意图更加清晰, 而且也能降低修改时出错的几率。

## 注释

有些语句的作用并不是为了让浏览器执行，而是为了方便需要阅读代码的人。我们把这些语句称为“注释”

写在注释里面的内容，JavaScript的解释器不会去执行

### 单行注释

以两个斜杠开头, 如下所示  
// 单行注释

### 块级注释

以一个斜杠和一个星号(/\*)开头, 以一个星号和一个斜杠(\*/)结尾, 不能嵌套。

/\*

\* 这是一个多行  
 \* (块级)注释

\*/

虽然上面注释中的第二和第三行都以一个星号开头, 但这不是必需的。之所以添加那两个星号, 纯粹是为了提高注释的可读性。

注释的确会略微增加JavaScript源文件的大小，从而对页面加载时间产生不好的影响。一般来说，这种影响小到可以忽略不计，但如果的确需要消除这种影响，我们可以清除JavaScript文件里的全部注释，形成所谓的“运行”版本，用于实际的站点。

## 严格模式

ECMASript5引入严格模式(strict mode)的概念。严格模式是为JavaScript定义了一种不同的解析与执行模型。在严格模式下，ECMAScript3中的一些不确定的行为将得到处理，而且对某些不安全的操作也会报错。要在整个脚本中启动严格模式，可以在顶部添加如下代码：

"use strict";

这行代码看起来像是字符串，而且也没有赋值给任何变量，但其实它是一个编译指示（pragma），用于告诉支持的JavaScript引擎切换到严格模式。这是为不破坏ECMAScript 3 语法而特意选定的语法。

在函数内部的上方包含这条编译指示，也可以指定函数在严格模式下执行：

function doSomething () {

"use strict";

//函数体

}

严格模式下，JavaScript的执行结果会有很大不同，支持严格模式的浏览器包括IE10+、Firefox4+、Safari5.1+、Opera12+和chrome。

# 数据类型

## 什么是数据类型

计算机的运行需要对各种各样的 值 进行操作，比如数字3.14或文本“大家好”。在编程语言中，这些值的类型称为数据类型。

## JS有哪几种数据类型

JS的数据类型分为两类：原始类型（基本数据类型，简单数据类型）和对象类型 （引用数据类型，复杂数据类型）

### 原始类型

数字（Number)  
字符串 (String)  
布尔值 (Boolean)  
空 (Null)  
未定义 (Undefined )

### 对象类型

除了5种原始数据类型之外的都是对象（Object）,对象是属性的集合，Object本质是由一组无序的键值对组成的

ECMAScript不支持任何创建自定义类型的机制，而所有值最终都将是上述6种数据类型之一

## 使用typeof操作符检测数据类型

使用typeof操作符检测数据的类型，可能返回下列某个字符串:

"undefined"——值未定义;  
"boolean"——布尔值;  
"string"——字符串;  
"number"——数字;  
"object"——对象或者null;  
"function"——函数

调用typeof null会返回"object",因为特殊值null被认为是一个空的对象引用。

函数在ECMAScript中是对象, 不是一种数据类型。但是因为函数有一些特殊的属性，所以通过typeof操作符来检测函数会返回function 而不是 object

## Undefined类型

Undefined类型只有一个值，即undefined

声明了一个变量但没有给它赋值，这个变量的默认值就是 undefined

<script>

var message;

alert(message); // "undefined"

</script>

既没有声明又没有赋值的变量，会导致一个错误。此时只能执行一项操作，即使用typeof操作符检测其数据类型

<script>

alert(age); // 报错：age is not defined

</script>

没有声明但是赋值了是有效的，但是不推荐这么做。建议加上var声明

<script>

num = 100; //有效但是不推荐

alert(num)

</script>

## Null类型

Null类型只有一个值即null。

当我们想让变量具有有效值，却又不是任何具体值时，就把null赋给变量。

从逻辑角度来看，null值表示一个空对象指针，而这也正是使用typeof操作符检测null值时会返回"object"的原因。如果定义的变量准备在将来用于保存对象，那么最好将该变量初始化为null而不是其他值。这样一来，只要直接检查null值就可以知道相应的变量是否已经保存了一个对象的引用，

如下面的例子所示:

<script>

if (car != null) {

// 对car对象执行某些操作

}

</script>

实际上, undefined值是派生自null值，因此ECMA-262规定对它们的相等性测试要返回true

<script>

alert(null == undefined); // true

</script>

尽管null和undefined有这样的关系，但它们的用途完全不同。  
如前所属，无论什么情况下都没有必要把一个变量的值显式地设置为undefined,可是同样的规则对null却不适用。换句话说，只要意在保存对象的变量还没有真正保存对象，就应该明确地让该变量保存null值。  
这样做不仅可以体现null作为空对象指针的惯例，而且也有助于进一步区分null和undefined。

## Boolean类型

boolean类型只有两个值: true和false，注意必须小写  
这两个值与数字值不是一回事，因此 true不一定等于 1，而 false也不一定等于 0

## 数字类型

数字类型包含整型和浮点型：

### 整型

十进制整数，如正整数、负整数和0  
八进制（以0开头），如0123（严格模式下无效，不推荐使用）  
十六进制（以0x或0X开头），如0x8a, 0xab0080  
在进行算术运算时，所有八进制和十六进制表示的数值最终都将被转换成十进制数值

### 浮点型

浮点数表示小数点前后可以有任意位，好像小数点可以浮动到任何数字的任何位置一样。

就是该数值中必须包含一个小数点，且小数点后面必须至少有一个数字。小数部分可以为0  
浮点数的表示形式可以是传统方式，比如3.14159，也可以是指数形式，如35.4e5

在指数表示方法中，e表示“10的幂”，所以35.4e5表示：35.4乘以10的5次幂。利用指数表示法，可以方便地表示特别大或特别小的数值。

下面都是有效的浮点数：3.0 ， 0.00001， -99.99， 2.5e12 , 1e-12

浮点数值的最高精度是17位小数，由于这个因素，程序中有时不能准确处理浮点数运算。如2-1.1=0.8999999999999999  
修正：toFixed(num) 将数字保留小数点后num位且四舍五入，并转化为字符串

<**script**>  
 **var *jg***=2-1.1;  
 ***console***.log(***jg***.toFixed(2))  
</**script**>

### NaN

NaN表示非数值，not a number 不是个数字

当脚本尝试把一些非数字当做数字进行处理，却无法得到数字时，其返回值就是NaN。比如拿一个整数乘以一个字符串，结果就是NaN

NaN和任何数据比较，永远返回false   
任何与NaN进行运算的结果均为NaN  
NaN不与任何值相等，包括自身  
它属于Number类型

一旦程序中出现：NaN,肯定进行了非法的运算操作，如：alert('200px'-100);

### isNaN

利用isNaN()函数能够检测其参数是否为一个“数字”，是数字返回false（不喜欢数字），非数字返回为true

alert(isNaN(100)); false  
alert(isNaN('你好')); true  
alert(isNaN('200')); 内部运行规律：先用Number去转

<script>  
 *//用户输入数字* var num = prompt("请输入一个数字：");  
 if(isNaN(num)){  
 document.write("您输入的不是数字，请确认！");  
 }else{  
 document.write("您输入的数字是："+num);  
 }  
</script>

## 字符串类型

定义：  
字符串由零个或多个字符构成。  
字符包括（但不限于）字母、数字、标点符号和空格。  
字符串必须包含在引号里，单引号或双引号都可以。  
如果字符串包含双引号，就把整个字符串放在单引号里  
如果字符串包含单引号，就把整个字符串放在双引号里。

如果字符串中包含包含与语法冲突的特殊字符，可以使用转义字符  
console.log('\t大家好，\n欢迎学习\'JavaScript!\'');

### 模板字符串

**放在反引号里面，**  
可以当作普通字符串使用；  
可以用来定义多行字符串；  
可以解析变量（把变量放在${}里面

## 数据类型转换

### 转换为布尔值

ECMAScript中所有类型的值都有与这两个Boolean值等价的值。可以对任何数据类型的值调用Boolean()函数，而且总会返回一个Boolean值。至于返回的这个值是true还是false，取决于要转换值的数据类型及其实际值。参见对应的转换规则图



<script>

var message = "Hello world!";

var messageAsBoolean = Boolean(message);

</script>

这些转换规则对理解流控制语句(如if语句)自动执行相应的Boolean转换非常重要，请看下面的代码:

<script>

var message = "Hello world!";

if (message) {

alert("Value is true");

}

</script>

因为字符串message被自动转换成了对应的Boolean值(true)

### 转换为数字类型

把其他类型数据转换为数字类型，主要有Number方法，parseInt()和parseFloat() 方法

Number()方法：可以用于任何数据，如果包含非法字符，则返回NaN

JS提供了两个可以把字符串强制转换为数值格式的函数。

parseFloat()函数解析字符串并返回一个浮点数。  
用法与parseInt完全相同，唯一区别：认识第一个小数点（开头是+，-，空格时，可以转）

<**script**>  
 ***console***.log(*parseFloat*(**"21.4"**)); *// 返回21.4* ***console***.log(*parseFloat*(**"76 trombones"**)); *// 返回76* ***console***.log(*parseFloat*(**"The magnificent 7"**)); *// 返回NaN*</**script**>

parseInt()函数解析字符串并返回整数或NaN。它还可以有第二个可选参数，用于指定数值的基，从而返回二进制、八进制或其他进制数值所对应的十进制数值。  
从左到右一个一个转换，遇到不是数字的字符（包括小数点）就停止转换（开头是+，-，空格时，可以转）。  
不能转，则返回NaN

<**script**>  
 ***console***.log(*parseInt*(**"18.95, 10"**)); *// 返回18* ***console***.log(*parseInt*(**"12px"**, 10)); *// 返回12* ***console***.log(*parseInt*(**"1110"**, 2)); *// 返回14* ***console***.log(*parseInt*(**"Hello"**)); *// 返回NaN*</**script**>

使用parseInt()方法把以下数据转为数字类型：  
'100px'; '100px125633'; '+100'; 22.5 ， 空字符串

一般情况下parseInt和parseFloat都是用来去转字符串，不要用它们去转换函数，布尔值等等，转不了的，因为他们俩是从左到右一个一个转

### 转换为字符串

String() 任何的数据类型都能转（全局函数，属于window对象的方法，相当于window.String() ）  
toString() 不能转null和undefined类型，其他类型都能转（属于字符串的方法，必须str.toString() ）

# 运算符

## 算术运算符

+ 加、- 减、\* 乘、/ 除、% 取模（求余数）

### 加(+)

用来做加法运算

alert(100+200); 结果：300  
alert(100+'200')； 结果：100200  
alert(10+5+'abc') 结果：15abc

用来连接字符串：字符串和任何类型的数据相连接，最终都是字符串

数字+字符串：100+’100’ 5+5+'hello' 'hello'+5+5 234 +‘ok’ + 111+ 222，NaN+字符串  
null+字符串，undefined+字符串，布尔值+字符串，数组+字符串，对象+字符串,函数+字符串

### 减(-)

用来做减法运算  
数字和字符串相减，会把字符串自动变成数字进行运算

alert(200-100); 结果：100  
alert(200-'100'); 结果：100

### 乘(\*)

用来数学乘法运算。  
数字和字符串相乘，会把字符串自动变成数字进行运算

alert(3\*7); 结果：21  
alert(3\*'7'); 结果：21

### 除(/)

用来数学除法运算。  
数字和字符串相除，会把字符串自动变成数字进行运算，  
如果除数为0，结果为Infinity 无穷大；Infinity 为数字类型

alert(81/9); 结果：9  
alert('81'/9); 结果：9  
alert(5/0); 结果：Infinity  
0/0 为 NaN

### 求余(%)

模运算符 %，就是求余数，不够除为它自己，0%3=0 ，1%3=1 ，2%3=2，3%3=0

用于被整除  
求一段范围（如要让一个数永远不超过10，那么取模10）

if(i===5){i=0}和 i%=5 是一个意思  
alert(8%5) 为3  
alert(10%-3) 为1，求余的正负数不是由后面的正负号决定，而是由前面的正负号决定

秒转分  
var s = 3605;   
alert( math.floor(s/60) + '分' + s%60 + '秒' );

### 递增(++)

递增运算符（++) 为其操作数增加1，返回一个数值。

### 递减(--)

递减运算符（--) 为其操作数减1，返回一个数值。

## 赋值运算符

就是把右边的值赋给左边的变量

=、+=、-=、\*=、/=、%=

如果变量num要加 1 ，有三种写法：  
num = num+1   
num += 1  
num++

如果变量num要减 1 ，有三种写法：  
num = num-1   
num -= 1  
num--

如果变量值得增加或减少不是1 ，而是其他数值，JavaScript还允许把算术操作符与等号结合使用：  
total = total + 5 相当于 total += 5

counter = counter - step 相当于 counter -= step

## 比较运算符

<、>、<=、>=、（等于：==、不等于!=）、（全等于===、全不等于!==）

返回的是一个 布尔值

== 等于，判断值是否相等（数据类型不同时会先转换类型，然后比较值）  
=== 全等，判断值和类型是否完全相等（数据类型不同时不会转换类型）

例：

var a = 3;  
var b = "3";

a==b 返回 true  
a===b 返回 false ，因为a,b的类型不一样

隐式类型转换：

> < 运算符 一定要注意是数字的比较还是字符串的比较

alert('10'>9); 显示为true 此时为数字的比较  
alert('10'>'9'); 显示为false 两个字符串则按照字符的Unicode编码去显示

数字的比较和字符串的比较是不一样的  
字符串比较是一位一位的比较 即'10'>'9' 先拿1和9 比较，当然是false了

## 逻辑运算符

### 且运算

运算符：&&  
一假且必假:两个有一个假的，值就是假的那个；  
两个都真，值为后；

### 或运算

运算符：||  
一真或必真:两个有一个是真的，值就是真的那个；  
两个都真，值为前；

### 取反

alert(!true);  
alert(!'ok');  
alert(!100); 把右边的数据类型转成布尔值

所有逻辑运算符的优先级都低于比较运算符

## 三目运算符

和if判断的第2种写法作用是一样的。优先级高于赋值，低于其他运算符

条件 ? 语句1 : 语句2

var num=51;  
num%2==0 ? alert('双数'):alert('单数');

# 程序结构

程序的三种基本结构

## 顺序结构

从上到下，从左到右执行的顺序，就叫做顺序结构，程序默认就是顺序结构

## 选择结构

选择结构也叫分支结构。在任何的编程语言中，代码需要依靠不同的输入作出决定并且采取行动。例如，在游戏中，如果玩家的生命值变成了0，那么游戏就结束了。在天气应用中，如果在早晨运行，就显示一张日出的图片；如果在晚上，就显示星星和月亮的图片。这就要用到条件语句。

### if语句

if语句括号里面的表达式，会自动调用Boolean()转型函数将这个表达式转换成一个布尔值

if ( 条件 ) { 语句块 }  
if ( 条件 ) { 语句块 } else { 语句块 }  
if ( 条件 ) { 语句块 } else if (条件) { 语句块 } ... else { 语句块 }

当 条件表达式成立时（值为true），执行 语句块。

注意：所有的相对路径（src路径，href路径）、颜色值、innerHTML 值 都别拿来做判断！  
除非是绝对路径：如src=‘http://www.baidu.com/logo.jpg’

我们可以把代码缩的更短一些。当检查某项数据是否是null值时，其实是在检查它是否存在。这种检查可以简化为直接把被检查的数据用作if语句的条件。if（something）与if（something ！= null）完全等价，但是前者显然更简明。此时，如果something存在，则if语句的条件为真；如果something不存在，则if语句的条件为假。

### switch语句

和多分支if...else if...else语句 作用是一样的  
switch 语句评估一个表达式，将表达式的值与case子句匹配，并执行与该情况相关联的语句。

语法：

<script>  
 switch (expression) {  
 case value1:  
 *// 当 expression 的结果与 value1 匹配时，执行此处语句* [break;]  
 case value2:  
 *// 当 expression 的结果与 value2 匹配时，执行此处语句* [break;]  
 ...  
 case valueN:  
 *// 当 expression 的结果与 valueN 匹配时，执行此处语句* [break;]  
 [default:  
 *// 如果 expression 与上面的 value 值都不匹配，执行此处语句* [break;]]  
 }  
</script>

expression  
一个用来与 case 子语句匹配的表达式。

case valueN 可选  
用于匹配 expression 的 case 子句。如果 expression 与给定的 valueN 相匹配，则执行该 case 子句中的语句直到该 switch 语句结束或遇到一个 break 。

default 可选  
一个 default 子句；如果给定，这条子句会在 expression 的值与任一 case 语句均不匹配时执行。  
default在switch语句中的最末尾可以不加break,但是在中间或头部要加break

## 循环结构

### for循环

想做这么一件事：  
重复执行某些代码  
每次执行的时候，有个数字在变化

思考：让alert(1);重复执行3次

var i=0;

for(;i<3;){

alert(1);

i++

}

以上代码还可以这么写：

for(i=0;i<3;i++){

alert(1);

}

再看如下例子：

for(i=0;i<3;i++){

alert(i);

}

在循环体内i为0，1，2  
但是：如果在for循环体外alert(i);那么i为3  
去掉i++ 变成死循环

### for...in循环

已知数组：var arr=['15','11','999']

x指代数组里面的下标0 ， 1， 2...

for( x in arr ){

alert('第'+x+'个东西是:'+arr[x]);

}

### while循环

while 循环会一直循环代码块，只要指定的条件为 true（先判断再执行）

语法

while (条件) {  
 要执行的代码块  
}

当条件为真时，

<**script**>  
 **var *a*** = 0;  
 **while** (***a*** < 3) {  
 ***console***.log(***a***);  
 ***a***++;  
 }  
</**script**>

### do...while循环

在检查条件是否为真之前，会先执行一次do里面的代码，然后只要条件为真就会重复循环。（先执行再判断）

如下： a 的值为 3 ，不满足条件 a < 3, 但是仍然会执行一次 do 里面的代码

<**script**>  
 **var *a*** = 3;  
 **do** {  
 ***console***.log(***a***);  
 ***a***++  
 } **while** (***a*** < 3)  
</**script**>

### 双重循环

什么是双重循环

一个循环体内又包含另一个完整的循环结构，比如：

<script type="text/javascript">  
 while(循环条件1){  
 *//循环操作1* while(循环条件2){  
 *//循环操作2* }  
 }  
</script>

<script type="text/javascript">  
 for(循环条件1){  
 *//循环操作1* for(循环条件2){  
 *//循环操作2* }  
 }  
</script>

<script type="text/javascript">  
 while(循环条件1){  
 *//循环操作1* for(循环条件2){  
 *//循环操作2* }  
 }  
</script>

双重循环注意：

各循环可互相嵌套  
一般不超过三层  
外层循环变量变化一次，内层循环变量要变化一遍

使用双重循环输出九九乘法表

<script type="text/javascript">  
 for(var *i*=1;*i*<=9;*i*++){  
 for(var *j*=1;*j*<=*i*;*j*++){  
 *document*.write(*i*+"\*"+*j*+"="+*i*\**j*+"&nbsp;&nbsp;&nbsp;")  
 }  
 *document*.write("<br/>");  
 }  
</script>

## break

break 跳出当前循环结构或switch分支结构

不要再执行这个循环结构了

循环五次录入五个人的成绩，如果录入的成绩小于0，则退出循环结构：

<script type="text/javascript">  
 for (let *i* = 1; *i* <= 5; *i*++) {  
 fenshu = prompt('请输入成绩',"");  
 if (fenshu < 0) {  
 *document*.write(`第${*i*}名同学的成绩输入有误，强制退出！`);  
 break;  
 }  
 *document*.write(`第${*i*}名同学的成绩为：${fenshu} <br>`);  
 }  
</script>

## continue

continue 跳过本次循环，执行下一次循环。（就是不要再走continue后面的语句了）  
注意：只能在循环结构内使用（while,do/while,for,for/in）

### for循环跳过

<script type="text/javascript">  
 for (let *i* = 0; *i* < 5; *i*++) {  
 if (*i* == 3) {  
 continue;  
 }  
 console.log(*i*)  
 }  
</script>

### 例子

<script type="text/javascript">  
 for (let *i* = 1; *i* <= 5; *i*++) {  
 fenshu = prompt('请输入成绩',"");  
 if (fenshu < 0) {  
 *document*.write(`第${*i*}名同学的成绩输入有误，该成绩作废！<br>`);  
 continue;  
 }  
 *document*.write(`第${*i*}名同学的成绩为：${fenshu} <br>`);  
 }  
</script>

# 数组对象

## 什么是数组

数组是一组数据的集合。数组中的每个元素可以保存任何类型（比如布尔值，数值，字符串，函数，对象，数组等）

## 怎么创建数组

### 三种方法

<script>  
 var *arr1* = new *Array*('hello', 100, '张三'); *// new方法* var *arr2* = ['hello', 100, '张三']; *// 字面量方法* var *arr3* = *Array*.of('hello', 100, '张三'); *// ES6方法 IE11不支持*</script>

## 数组的长度

数组的长度就是数组中元素的数量，数组的长度可读可写

<script>  
 var *arr* = ['a', 'b', 'c']  
 *arr*.length = 1  
 console.log(*arr*) *// ["a"]*</script>

## 数组的下标

数组中元素的下标从0开始，最后一个元素的下标：数组的长度减1：arr.length-1

### 通过下标添加元素

<script>  
 var *arr1* = new *Array*();  
 *arr1*[0] = 'hello';  
 *arr1*[1] = 100;  
 *arr1*[2] = '张三';  
 console.log(*arr1*);  
  
 var *arr2* = [];  
 *arr2*[0] = 'hello';  
 *arr2*[1] = 100;  
 *arr2*[2] = '张三';  
 console.log(*arr2*);  
  
 var *arr3* = *Array*.of();  
 *arr3*[0] = 'hello';  
 *arr3*[1] = 100;  
 *arr3*[2] = '张三';  
 console.log(*arr3*);  
</script>

### 通过下标修改元素：

<script>  
 var *arr* = ['a', 'b', 'c', 'd'];  
 *arr*[2] = 'xyz';  
 console.log(*arr*);  
</script>

通过下标在数组的最后增加一个元素 :

<script>  
 let *arr* = ['张三', '李四', '王五', 1, 234, 55, 'last'];  
 *arr*[*arr*.length] = '888'  
 console.log(*arr*[*arr*.length - 1])  
</script>

## 快速清空数组

<script>  
 let *arr* = ['张三', '李四', '王五', 1, 234, 55, 'last'];  
 console.log(*arr*.length)  
 *// arr.length = 0;  
 arr* = []; *// 效率会高些* console.log(*arr*)  
</script>

## 二维数组

<script>  
 let *arr* = [[1, 3, 4], ['a', 'b'], ['张三', '李四', '王五', '赵六']];  
 console.log(*arr*.length);  
 console.log(*arr*[1]);  
 console.log(*arr*[2][0]);  
</script>

## 数组的遍历

记住：只要你想依次把数组里面每一个元素都取出来，那么就要用到for循环了

for, while , for...in

<script>  
 let *arr* = ['张三', '李四', '王五', 'a', 'b', 'c'];  
  
 for (let *i* = 0; *i* < *arr*.length; *i*++) {  
 const *element* = *arr*[*i*];  
 console.log(*element*);  
  
 }  
  
 let *i* = 0;  
 while (*i* < *arr*.length) {  
 console.log(*arr*[*i*]);  
 *i*++;  
 }  
  
 for (const *i* in *arr*) {  
 *//console.log(i); //i 下标* console.log(*arr*[*i*]);  
 }  
</script>

二维数组的遍历

<script>  
 let *arr* = [[1, 3, 4], ['a', 'b'], ['小明', '小李', '张三', '李四']];  
 for (let *i* = 0; *i* < *arr*.length; *i*++) {  
 for (let *j* = 0; *j* < *arr*[*i*].length; *j*++) {  
 console.log(*arr*[*i*][*j*]);  
 }  
 }  
</script>

## 数组的方法

### 数组中元素的添加和删除

开头添加和删除：unshift() 和shift()，  
末尾添加和删除：push() 和 pop()

### slice()

slice(start,end) 截取数组中的元素；不操作数组本身，（字符串也有这个方法，用法一样,但是数组没有substring方法）

返回start（包含）到end（不包含）之间的元素组成的数组，位置从0开始  
如果start大于或等于end，返回一个空数组  
如果只有一个参数，则默认为start，则返回start（包含）到数组的末尾  
如果值为负数，会被当做 数组长度+负数 处理

### splice()

操作数组本身  
兼具删除，添加和替换功能  
splice(索引,删除的数量，添加的元素)  
索引：从0开始  
删除的数量：可选；整数，表示要移除的数组元素的个数。  
添加的元素：可选；一个或多个元素

### indexOf()

返回某个指定的元素在数组中首次出现的位置

indexOf(element,n) 从下标 n开始向右搜索element，找到后将其下标返回；如果没有找到则返回-1

### lastIndexOf()

返回某个指定的元素在数组中最后出现的位置

lastIndexOf(element,n) 从下标n开始向左搜索element，找到后将其下标返回；如果没有找到则返回-1

### concat()

用于合并两个或多个数组。此方法不会更改现有数组，而是返回一个新数组。

### join()

join()方法把数组元素按指定分隔符组合成一个字符串；如果没有分隔符，默认是逗号。不操作数组本身

### reverse()

将数组中元素的位置颠倒，并返回该数组。数组的第一个元素会变成最后一个，数组的最后一个元素变成第一个。该方法会改变原数组

### includes()

includes() 方法用来判断一个数组是否包含一个指定的值，如果包含则返回 true，否则返回false。

### sort()

按照字符的unicode编码排序，默认升序，先排第一个字符，以此类推

<**script**>  
 **var *arr***=[4,3,5,76,2,0,8];  
 ***arr***.sort( **function**(a,b){  
 **return** a-b;  
 });  
 *alert*(***arr***);  
</**script**>

结果：0,2,3,4,5,8,76

## 迭代方法

所谓的迭代，就是对数组的每一项都进行相应的操作。

### forEach()

循环数组或集合中的对象、数据

arr.forEach(callback[, thisArg]);  
arr.forEach(回调函数,this对象);

### map()

返回一个由原数组每个元素执行回调函数的结果组成的新数组。

<script>  
 var *array1* = [1, 4, 9, 16];  
  
 *// pass a function to map* const *map1* = *array1*.map(x => x \* 2);  
  
 console.log(*map1*);  
 *// expected output: Array [2, 8, 18, 32]*</script>

### filter()

返回让回调函数为true的元素组成的数组

<script>  
 var *words* = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];  
  
 const *result* = *words*.filter(word => word.length > 6);  
  
 console.log(*result*);  
 *// expected output: Array ["exuberant", "destruction", "present"]*</script>

### every()

有一个元素让条件为false，则返回false 。且剩余的元素不会再进行检测。

<script>  
 function isBelowThreshold(currentValue) {  
 return currentValue < 40;  
 }  
  
 var *array1* = [1, 30, 39, 29, 10, 13];  
  
 console.log(*array1*.every(isBelowThreshold));  
 *// expected output: true*</script>

### some()

有一个元素让条件为true，则返回true 。 且剩余的元素不会再进行检测。

下面的例子检测在数组中**是否有**元素大于 10。

<script>  
 function isBiggerThan10(element, index, array) {  
 return element > 10;  
 }  
  
 [2, 5, 8, 1, 4].some(isBiggerThan10); *// false* [12, 5, 8, 1, 4].some(isBiggerThan10); *// true*</script>

### reduce()

循环数组中的每一项进行累计操作

### find(）

返回满足条件的第一个元素的值。否则返回 undefined。

<script>  
 var *array1* = [5, 12, 8, 130, 44];  
  
 var *found* = *array1*.find(function(element) {  
 return element > 10;  
 });  
  
 console.log(*found*);  
 *// expected output: 12*</script>

### findIndex(）

返回数组中满足条件的第一个元素的索引。否则返回 -1。

<script>  
 var *array1* = [5, 12, 8, 130, 44];  
  
 function isLargeNumber(element) {  
 return element > 13;  
 }  
  
 console.log(*array1*.findIndex(isLargeNumber));  
 *// expected output: 3*</script>

# 日期对象

## Date对象的构造函数

创建一个日期对象：new Date()

## 实例化Date对象

创建一个新Date对象的唯一方法是通过new 操作符：

<script>  
 let *now* = new *Date*();  
</script>

## 语法

Date()构造函数有四种基本形式

### 没有参数

new Date()

如果没有输入任何参数，则Date的构造器会依据系统设置的当前时间（也就是执行JavaScript的客户端电脑所设置的时间）来创建一个Date对象。

### Unix时间戳

new Date(milliseconds)

一个 Unix 时间戳（Unix Time Stamp），它是一个整数值，表示自1970年1月1日00:00:00 UTC（the Unix epoch）以来的毫秒数，忽略了闰秒。请注意大多数 Unix 时间戳功能仅精确到最接近的秒。

### 时间戳字符串

new Date(dateString)

注意: 由于浏览器之间的差异与不一致性，**强烈不推荐**使用Date构造函数来解析日期字符串 (或使用与其等价的Date.parse)。

<**script**>  
 **var *birthday*** = **new *Date***(**'March 21, 1987 23:15:30'**);  
 **var *day1*** = ***birthday***.getDay();  
 *// Sunday - Saturday : 0 - 6* ***console***.log(***day1***); *// 6* ***console***.log(***birthday***.getHours()); *// 23*</**script**>

### 分别提供日期与时间的每一个成员

new Date(year, monthIndex [, date [, hours [, minutes [, seconds [, milliseconds]]]]]);

当至少提供了年份与月份时，这一形式的 Date() 返回的 Date 对象中的每一个成员都来自下列参数。没有提供的成员将使用最小可能值（对日期为1，其他为0）。

new Date(2018,9,16,0,50,0); //月份从0开始，

## Date对象转换为字符串

它们是很实用的，能够把日期转换为更容易理解的格式。

本地格式：该字符串格式因不同语言而不同

<script>  
 var *myDate* = new *Date*();  
  
 *// 日期和时间：Wed Nov 06 2019 08:44:11 GMT+0800 (中国标准时间)* console.log(*myDate*.toString());  
 console.log(*myDate*.toDateString()); *// 日期部分：Wed Nov 06 2019* console.log(*myDate*.toTimeString()); *// 时间部分：08:49:00 GMT+0800 (中国标准时间)* console.log(*myDate*.toLocaleString()); *// 本地格式的日期和时间：2019/11/6 上午8:56:35* console.log(*myDate*.toLocaleDateString()); *// 本地格式的日期部分：2019/11/6* console.log(*myDate*.toLocaleTimeString()); *// 本地格式的时间部分：上午8:56:35  
  
 // 根据世界时，把Date对象转换为字符串：Wed, 06 Nov 2019 00:49:00 GMT* console.log(*myDate*.toUTCString());   
</script>

## Date对象的方法

### 获取时间

这些方法返回的都是number类型

<script>  
 let *now* = new *Date*();  
 console.log(*now*.getFullYear()); *// 返回一个四位数字的年份* console.log(*now*.getMonth()); *//返回一个 0 到 11的整数值: 0代表一月份，1代表二月份，依次类推* console.log(*now*.getDate()); *// 返回一个 1 到 31 的整数值，表示一个月中的第几天* console.log(*now*.getHours()); *// 返回一个 0 到 23 的整数值，表示小时* console.log(*now*.getMinutes()); *// 返回一个 0 到 59 的整数值，表示分钟数* console.log(*now*.getSeconds()); *// 返回一个 0 到 59 的整数值，表示秒数* console.log(*now*.getMilliseconds()); *// 方法返回一个0 到 999的整数，表示毫秒数* console.log(*now*.getDay()); *// 返回一个 0到 6的整数值： 0 代表星期日，1代表星期一，依次类推* console.log(*now*.getTime()); *// 返回距 1970 年 1 月 1 日之间的毫秒数*</script>

# 字符串对象

## 字符串长度

字符串属性：length，获取字符串的长度

例如：var str = 'www.aixue8.com';  
alert(str.length);

注意:空格也要计算在长度内

## 字符串方法

### str.charAt(下标)

返回指定下标所在的字符，第一个字符的下标为0。

例如：

<**script**>  
 **var *str*** = **'好代码'**;  
 *alert*(***str***.charAt(1));  
 *alert*(***str***.charAt(0)); *//或 alert(str.charAt())*</**script**>

用[]访问单个字符和charAt()这两种方式都可以获取到字符串对应位置的字符，获取的位置都是从0开始。  
区别在于：  
使用string[]的方式，对于超出字符长度范围的，会返回undefined  
而使用charAt()的方式，对于超出范围的会返回一个空的字符串。

-----------------

### str.charCodeAt(下标)

根据下标获取字符的 Unicode 编码，下标从0开始

var str='爱学吧';  
alert( str.charCodeAt(0) );

### slice()和substring()

截取一个字符串的一部分，并返回一新的字符串，主要有两个方法：  
str.substring(start,end)和str.slice(start，end)

str.slice(start，end)  
返回start（包含）到end（不包含）之间的字符串，位置从0开始  
如果start大于或等于end，返回一个字符串  
如果只有一个参数，则默认为start，则返回start（包含）到字符串的末尾  
如果值为负数，会被当做 数组长度+负数 处理

str.substring(start,end)和slice()方法一样，区别：  
如果start大于end，则交换位置  
参数值为负数，则被视为0  
任何大于字符串长度的参数值都被视为字符串长度  
任何NaN参数值都被视为0

### search('string')

找到了返回其下标，找不到返回-1 （类似indexOf()方法）

### match('string')

只能匹配一个。返回数组或null

### replace('string1','string2')

把string1替换成string2

var str = 'www.aixue8.com';  
document.write(str.replace('aixue8','soucloud'))

### split('string'[,limit])

使用指定的分隔符字符串将一个字符串分割成字符串数组；limit一个整数，限定返回的分割片段数量

可以使用正则表达式：str.split(regExp)

### toUpperCase()

转换为大写  
str.toUpperCase()

### 和toLowerCase()

转换为小写  
str.toLowerCase()

### trim()

删除字符串两端的空白字符。在这个上下文中的空白字符是所有的空白字符 (space, tab, no-break space 等) 以及所有行终止符字符（如 LF，CR等）。

### includes()

includes() 方法用于判断一个字符串是否包含另一个字符串，如果当前字符串包含被搜寻的字符串，就返回 true；否则返回 false。区分大小写

### indexOf('string',下标)

返回某个指定的字符串值在字符串中首次出现的位置

### lastIndexOf('string',Number)

str.lastIndexOf('string',Number)  
返回某个指定的字符串值在字符串中最后出现的位置

### startsWith('string')

字符串string是否在原字符串的开头，返回布尔值

### endsWith('string')

字符串string是否在原字符串的结尾，返回布尔值

### concat()

str.concat(str1,str2,...)   
将多个字符串连接成一个字符串  
用+更简单一些，推荐

### repeat()

str.repeat(n)   
将原字符串重复n次后返回一个新字符串

# Math对象

与Date对象不同的是，Math对象不需要创建就可以使用。我们直接调用它的方法就可以了。

Math 是一个内置对象， 它具有数学常数和函数的属性和方法。不是一个函数对象。

## ceil()

语法：Math.ceil(x) 参数x是一个数字，向上取整，返回 大于或等于 给定数字的最小整数

## floor()

语法：Math.floor(x) 参数x是一个数字；向下取整，返回 小于或等于 给定数字的最大整数

## round()

语法：Math.round(x) 参数x是一个数字；返回一个数字四舍五入后最接近的整数

## random()

Math.random() 函数返回一个大于等于0，小于1之间的随机数  
然后您可以缩放到所需的范围。实现将初始种子选择到随机数生成算法;它不能被用户选择或重置。

### 得到一个两数之间的随机数

min≤x＜max。

公式：Math.random() \* (max - min) + min;

### 得到一个两数之间的随机整数

min≤x＜max。  
公式：Math.floor(Math.random()\*(max-min)+min)

min＜x≤max。  
公式：Math.ceil(Math.random()\*(max-min)+min)

## max()

语法：Math.max(value1[,value2, ...])   
参数value1, value2, ... 是一组数值。返回给定的一组数字中的最大值，

## min()

语法：Math.min(value1[,value2, ...])   
参数value1, value2, ... 是一组数值；返回给定的一组数字中的最小值，

## PI

Math.PI

圆周率，一个圆的周长和直径之比，约等于 3.14159

## 平方根

Math.SQRT1\_2

1/2的平方根, 约等于 0.707

Math.SQRT2

2的平方根,约等于 1.414

## abs()

Math.abs(x);   
返回x的绝对值，参数x是一个数字

# 函数

## 什么是函数

JavaScript 函数是用来完成特定任务的代码块。JavaScript把部分代码包装为能够重复使用的模块，称为函数。

## 创建函数

### 函数声明

函数声明的语法：

function 函数名(参数1,参数2,...) {  
 函数体(代码块)  
 return 返回值；  
}

### 函数表达式

var fn = function (参数1,参数2,...) {  
 函数体  
 return 返回值；  
};

推荐这种方式，因为函数声明方式会提前

### ES6箭头函数表达式

箭头函数表达式的语法比函数表达式更简洁，并且没有自己的this，arguments，super或 new.target。这些函数表达式更适用于那些本来需要匿名函数的地方，并且它们不能用作构造函数。

箭头函数不能用作构造器，和 new一起用会抛出错误。  
var Foo = () => {};  
var foo = new Foo(); // TypeError: Foo is not a constructor

箭头函数没有prototype属性。  
var Foo = () => {};  
console.log(Foo.prototype); // undefined

基础语法：  
(参数1, 参数2, …, 参数N) => { 语句 }

(参数1, 参数2, …, 参数N) => 表达式（单一）  
相当于：(参数1, 参数2, …, 参数N) =>{ return 表达式; }

当只有一个参数时，圆括号是可选的：  
(单一参数) => {函数声明}  
单一参数 => {函数声明}

没有参数的函数应该写成一对圆括号。  
() => {函数声明}

箭头函数没有arguments对象。可以用rest参数代替

### 使用Function构造函数

var fn = new Function('arg1','arg2',...,'argN','代码块;returen 返回值');

arg 是参数；必须加引号；return返回值不是必须的  
不建议使用，效率低，影响性能

## 调用函数

函数只有在调用时才能执行函数体里面的代码，不调用就永远不会执行。

### 函数名加小括号

函数名称() 或 变量名()

如 ：fn();

fn存储的是一个地址，代表的是当前的整个函数块  
fn() 表示执行函数里面的代码

### 通过事件调用

obj.事件 = 函数名称 或 变量名 或 整个函数块

### 整个函数加小括号

( function () {alert('ok');} )() ;

可以避免代码库中的函数有重名问题，且只会在运行时执行一次，一般用作初始化工作

### 通过函数的apply或call或bind方法调用

call() 方法使用一个指定的 this 值和单独给出的一个或多个参数来调用一个函数。（就是说可以指定谁来调用这个函数，且调用时可以传参）

语法

fun.call(thisArg, arg1, arg2, ...)

参数

thisArg：在 fun 函数运行时指定的 this 值。  
arg1, arg2, ... ：指定的参数列表。

## 函数传参

### 实参和形参

函数的参数分为实参和形参：  
实参：在调用函数的时候，小括号中传入的变量或值;实际传入的参数  
形参：在创建函数的时候，小括号中定义的变量，用来接收实参。ES6允许为函数的参数设置默认值，在ES6之前，不可以。设置方法：直接在形参里面设置

参数的类型可以是任何数据类型

传参的作用：可以动态的改变函数体内对应的变量的值或类型，使同一函数体得到不同的结果，减少代码重复  
对参数进行判断，根据判断的结果进行相应的处理

### 实参==形参

正好一对一

<**script**>  
 **function** *fn*(a) {  
 ***console***.log(a);  
 }  
 *fn*(100)  
</**script**>

<**script**>  
 *fn*(**'我喜欢'**);  
 *fn*(**'javascript'**);  
 **function** *fn*(a) {  
 ***console***.log(a.charAt(2));  
 }  
</**script**>

多个实参之间用英文逗号隔开：

<**script**>  
 *fn*(100, **'px'**);  
 **function** *fn*(a, b) {  
 ***console***.log(a + b);  
 }  
</**script**>

### 实参<形参

不会报错，会自动赋值为undefined

<**script**>  
 **function** *f*(a) {  
 ***console***.log(a); *// undefined* }  
 *f*();  
</**script**>

### 实参>形参

不会报错，可以用arguments对象或ES6剩余参数 获取到

arguments对象是所有（非箭头函数）函数中都可用的局部变量，只能在函数体内使用。每创建一个函数，该函数就会隐式创建一个arguments对象，它包含了当前函数的所有实参。

arguments 对象具有如下属性：  
arguments.length:返回实参的数量  
arguments.callee:返回当前函数块（就是函数本身）（匿名函数可以使用该属性实现递归调用）

<**script**>  
 *fn*(1, 2, 3); *//实参，实际传递的参数* **function** *fn*(a, b, c) { *//a，b，c是形参，用来接收实参* ***console***.log(***arguments***); *// 实参的集合* ***console***.log(***arguments***.**length**); *// 实参的数量* ***console***.log(***arguments***[0]); *// 第一个实参* ***console***.log(***arguments***[***arguments***.**length** - 1]); *// 最后一个实参* ***console***.log(***arguments* instanceof *Array***); *// false* }  
</**script**>

剩余语法(Rest syntax) 看起来和展开语法完全相同，不同点在于, 剩余参数用于解构数组和对象。从某种意义上说，剩余语法与展开语法是相反的：展开语法将数组展开为其中的各个元素，而剩余语法则是将多个元素收集起来并“凝聚”为单个元素。

ES2015引入了rest参数，形式为：...变量名 ，用于获取函数的多余参数，这样就不需要使用arguments对象了。rest参数的变量是一个数组，里面包含了函数的参数。rest参数之后不能再有其他的参数，即只能是最后一个参数，否则会报错

### 将函数作为值传递

将一个函数作为参数传递给其他函数。这看起来像《盗梦空间》中的情形，但提供了强大的功能。从本质上说，函数视为一种类型，因此可将函数赋给变量，以后再通过变量来调用它们。

函数是一种对象类型，因此可将其传递给其他函数

## this对象

在函数内部除了有arguments这个对象，还有个this对象  
在绝大多数情况下，函数的调用方式决定了this的值。this不能在执行期间被赋值，并且在每次函数被调用时this的值也可能会不同。对于箭头函数，函数体内this对象就是定义时所在的对象，而不是调用时所在对象

### 全局环境

无论是否在严格模式下，在全局执行环境中（在任何函数体外部）this 都指向全局对象。

<script>  
 *// 在浏览器中, window 对象同时也是全局对象：* console.log(this === *window*); *// true* a = 37;  
 console.log(*window*.a); *// 37* this.b = "MDN";  
 console.log(*window*.b) *// "MDN"* console.log(b) *// "MDN"*</script>

### 作为对象的方法

当函数作为对象里的方法被调用时，它们的 this 是调用该函数的对象。

下面的例子中，当 obj.f()被调用时，函数内的this将绑定到obj对象。

<script>  
 var *obj* = {  
 num: 100,  
 f: function () {  
 console.log(this.num)  
 }  
 };  
  
 *obj*.f(); *// 100*</script>

### 作为构造函数

当一个函数用作构造函数时（使用new关键字），它的this被绑定到正在构造的新对象。

### 作为一个DOM事件处理函数

当函数被用作事件处理函数时，它的this指向触发事件的元素

<script>  
 var *mybtn* = *document*.querySelector('#btn');  
 *mybtn*.onclick = function () {  
 console.log(this); *// 是按钮调用了this所在的这个函数, 所以this就是按钮这个元素* }  
  
 *mybtn*.onclick = function () { fn2(); } *// 相当于 window.fn2()* function fn2() {  
 console.log(this); *// this => window* }  
  
 *mybtn*.onclick = function () { fn3(this); }  
 function fn3(a) {  
 console.log(a); *// this => 按钮元素* }  
</script>

## 返回值

return语句终止函数的执行，并返回一个指定的值给函数调用者。

### 语法

return [[expression]];

expression表达式的值会被返回。如果没有expression，则返回 undefined。

任何函数都可以通过return语句来实现返回值，返回值可以是任何js的数据类型

<script>  
 function counter() {  
 for (var count = 1; ; count++) { *// 无限循环* console.log(count + "A"); *// 执行5次* if (count === 5) {  
 return;  
 }  
 console.log(count + "B"); *// 执行4次* }  
 console.log(count + "C"); *// 永远不会执行* }  
 counter();  
</script>

## 作用域

什么是作用域：一条数据起作用的范围

### 全局作用域

声明在任何函数以外的数据，拥有全局作用域。  
全局作用域的数据，在这条数据声明之后的任何地方都能访问和修改，且只有在页面关闭后才被删除。

### 局部作用域

函数的作用域就是局部作用域  
局部作用域的数据，只能在函数内部进行访问和修改，且在函数运行以后被立即删除

局部作用域的数据：  
在函数内部使用var或let声明的变量（函数内未使用var或let声明而直接赋值的变量拥有全局作用域）  
函数的参数；  
函数内的声明函数和函数表达式

### 作用域链

当我们调用一条数据的时候，会先在本作用域进行查找，如果找不到就向上查找父作用域，如果还是没有找到，就继续向上，一直找到全局作用域，还是找不到就报错。

<script>  
 var *authorName*="山边小溪";  
 function doSomething(){  
 var blogName="梦想天空";  
 function innerSay(){  
 alert(blogName);  
 }  
 innerSay();  
 }  
 alert(*authorName*); *//山边小溪* alert(blogName); *//脚本错误* doSomething(); *//梦想天空* innerSay() *//脚本错误*</script>

## 闭包

### 什么是闭包

在一个函数内部创建另一个函数，通过另一个函数访问这个函数的局部变量（参数和变量不会被垃圾回收机制所收回）

从技术的角度讲，所有的JavaScript函数都是闭包：它们都是对象，它们都关联到作用域链。

<script>  
 function init() {  
 var name = "Mozilla"; *// name 是一个被 init 创建的局部变量* function displayName() { *// displayName() 是内部函数,一个闭包* alert(name); *// displayName() 函数使用了父函数中声明的变量* }  
 displayName();  
 }  
 init();  
</script>

因为：一个函数引用另一个函数的变量，因为变量被引用着所以不会被回收  
这是优点也是缺点，不必要的闭包只会增加内存消耗

闭包中的this在运行时指向window

### 闭包的用途

可以读取函数内部的变量，是将函数内部和函数外部连接起来的一座桥梁。  
让这些变量的值始终保持在内存中。

### 闭包的缺点

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。过度使用闭包会导致性能下降，尽量少使用闭包，只在必要时使用。

闭包会在父函数外部，改变父函数内部变量的值

## 回调函数

回调（callback）：把一个函数作为参数传到一个主函数里面，当主函数执行完之后，再执行传进去的这个函数。这个过程就叫做回调。

## 递归函数

在函数内部直接或间接的调用自己  
使用递归时调用次数不要太多，防止内存溢出；要有让调用结束的条件

<**script**>  
 **function** *fn*(num) {  
 **if** (num <= 1) {  
 **return** 1;  
 } **else** {  
 **return** num \* *fn*(num - 1)  
 }  
 }  
 console.log(fn(4));  
</**script**>

## 内置全局函数

内置：其实就是系统已经编写好的一些函数，具有特定的功能，可以直接使用  
顶层：全局函数，可以作用于任何对象



# JS的预解析规则

## 变量的预解析

JS会对整个script块中的代码进行预解析，把var声明的变量提前至作用域的顶部，且变量的值都是undefined  
变量重名的：只留后一个变量  
如果预解析这一步出现错误：整个script块的代码都不会执行，但是不会影响其他script块的代码

预解析完毕后，开始执行代码，表达式可以修改预解析的值，表达式：= + - \* / ++ -- %  
如果执行代码这一步错误：只会影响其后代码的执行，不会影响其他script块的代码

## 函数的预解析

JS会对整个script块中的代码进行预解析，把函数声明的整个函数块（函数名称和函数体）提前至作用域的顶部

函数重名的：只留后一个函数  
变量和函数名重名了，就只留下函数  
如果预解析这一步出现错误：整个script块的代码都不会执行，但是不会影响其他script块的代码

预解析完毕后，执行代码，表达式可以修改预解析的值，表达式：= + - \* / ++ -- %  
如果执行代码这一步错误：只会影响其后代码的执行，不会影响其他script块的代码

函数执行的时候会对函数中的变量和函数进行预解析

# 面向对象

面向对象（Object-Oriented，OO）的语言有一个标志，那就是它们都有类的概念，而通过类可以创建任意多个具有相同属性和方法的对象。ECMAScript5没有类的概念，ES6才有。

## 什么是对象

对象是某个特定引用类型的实例

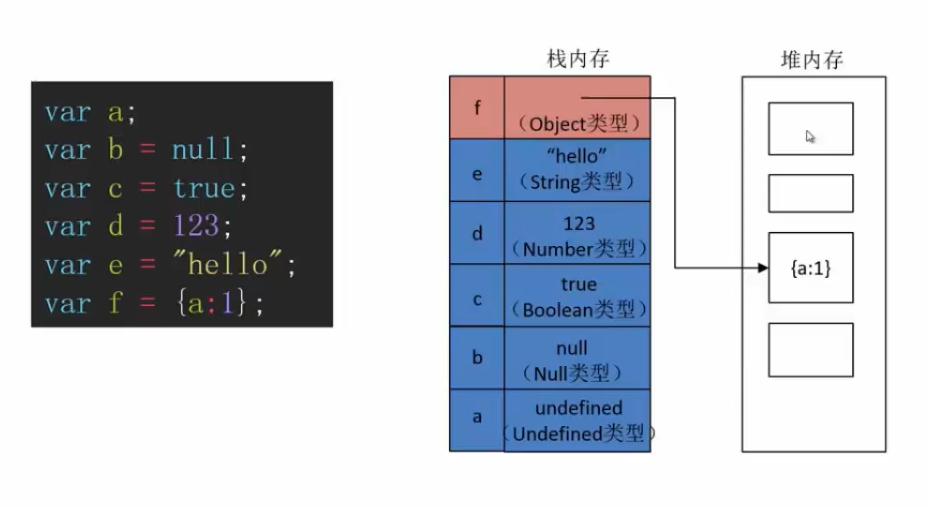
对象类型，也叫引用数据类型，复杂数据类型

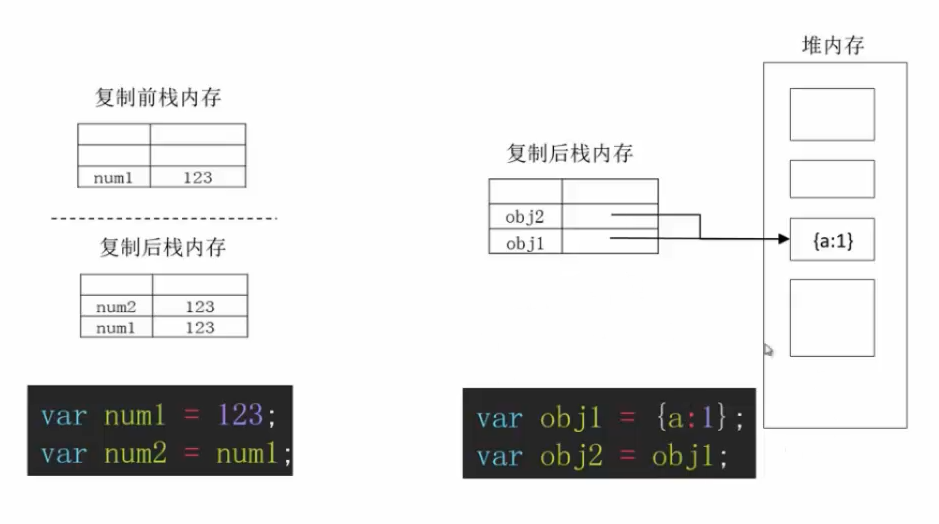
## 对象分类

内置对象，由ECMAScript定义的对象：数组（Array对象)，日期（Date对象），函数（Function对象），Math对象，正则表达式(regExp对象)  
宿主对象：宿主环境提供的对象  
自定义对象

## 原始数据类型和对象类型

原始数据类型：在内存中，存放在栈中的简单数据段，它们直接存储在变量访问的位置。按照值来操作  
对象数据类型：在内存中，存放在堆中的对象，存储在变量处的值是一个指针，指向存储对象的内存处。按照地址来操作





## 创建对象-New模式

JavaScript有一个内置对象Object，利用它可以创建一个空白的对象：

var myNewObject = new Object();

这样就得到了一个崭新的对象myNewObject，这时它还没有任何属性和方法，因此没有任何实际功能。我们可以使用点方式添加属性：

myNewObject.info = 'I am a shiny object';

## 创建对象-对象字面量模式

早期的JS开发人员经常使用new Object()方式创建新对象，现在，对象字面量成为创建新对象的首选方式。上面的例子用对象字面量语法可以写成这样：

<script>  
 var *person* = {  
 name: "小明",  
 age: 18,  
 job: "前端工程师",  
 say: function () {  
 console.log('我是：' + this.name + ',年龄:' + this.age);  
 }  
 };  
 console.log(*person*.age);  
 console.log(*person*['name']);  
 *person*.say();  
</script>

## 创建对象-工厂模式

使用Object构造函数或对象字面量创建对象的缺点：使用同一个接口创建很多对象，会产生大量的重复代码。为解决这个问题，人们开始使用工厂模式的一种变体。

<**script**>  
 **let *p1*** = {  
 **name**: **'张三'**,  
 **age**: 18,  
 };  
 **let *p2*** = {  
 **name**: **'李四'**,  
 **age**: 26,  
 };  
</**script**>

工厂模式是软件工程领域一种广为人知的设计模式，这种模式抽象了创建具体对象的过程。考虑ES5中无法创建类，开发人员就发明了一种函数，用函数来封装以特定接口创建对象的细节，如下面的例子所示：

<script>  
 function createPerson(name, age,job) {  
 let obj = {};  
 obj.name = name;  
 obj.age = age;  
 obj.job = job;  
 obj.sayName = function () {  
 console.log(this.name);  
 };  
 return obj;  
 }  
  
 let *p1* = createPerson('zhangsan', 29, "WEB"); *// 第一个实例* let *p2* = createPerson('lisi', 27,'UI'); *// 第二个实例  
 p1*.sayName();  
 *p2*.sayName();  
 console.log(*p1*.sayName === *p2*.sayName); *// false*</script>

工厂模式解决了代码重复问题，  
弊端1：看不出对象的类型（都是Object类型），比如 new Date() 为日期Date类型  
弊端2：函数重复，浪费资源，比如console.log(p1.say === p2.say); // false

## 创建对象-构造函数模式

如果只需要某个对象的一个实例，使用直接创建对象实例的方法还算不错。但如果要创建同一个对象的多个实例，使用这种方式就要反复重复整个过程：创建对象、添加属性、定义方法等。

如果要创建可能具有多个实例的对象，更好的方式是“对象构造函数”。它会创建某种模板，方便实现多次实例化。

在把对象实例化时，还可以通过给构造函数传递一个或多个参数来定制对象。

<**script**>  
 **function** *Person*(personname, personage, personjob) {  
 **this**.**name** = personname;  
 **this**.**age** = personage;  
 **this**.**job** = personjob;  
 **this**.sayName = **function** () {  
 ***console***.log(**this**.**name**);  
 }  
 }  
  
 **var *person1*** = **new** *Person*(**'zhangsan'**, 29, **"WEB"**);  
 **var *person2*** = **new** *Person*(**'lisi'**, 27, **'UI'**);  
  
 ***person1***.sayName();  
 ***person2***.sayName();  
  
</**script**>

## 创建对象-原型模式

### prototype

我们创建的每个函数都有一个prototype（原型）属性，这个属性是一个指针，指向一个对象，这个对象的用途是包含所有实例共享的属性和方法。使用原型的好处是可以让所有对象实例共享它所包含的属性和方法。换句话说，不必在构造函数中定义对象实例的信息，而是可以将这些信息直接添加到原型对象中：

<script>  
 function Person() {  
 }  
 Person.prototype.name = 'zhangsan';  
 Person.prototype.age = 28;  
 Person.prototype.job = 'Full Stack Developer';  
 Person.prototype.sayName = function () {  
 console.log(this.name);  
 }  
  
 var *person1* = new Person();  
 *person1*.sayName(); *// zhangsan* var *person2* = new Person();  
 *person2*.sayName(); *// zhangsan* console.log(*person1*.sayName === *person2*.sayName); *// true*</script>

### constructor

无论什么时候，只要创建了一个新函数，就会根据一组特定的规则为该函数创建一个prototype属性，这个属性指向函数的原型对象。在默认情况下，所有的原型对象都会自动获得一个constructor（构造函数）属性，这个属性是一个指向prototype属性所在函数的指针。

### \_\_proto\_\_

Firefox、Safari和Chrome在每个对象上都支持一个属性\_\_proto\_\_；每个实例对象包含一个\_\_proto\_\_属性，该属性仅仅指向了Person.prototype；

### instanceof

前面提到过，在默认情况下，所有的原型对象最初都只包含constructor（构造函数）属性，而该属性也是共享的，因此可以通过对象实例访问。

console.log(person1.constructor === Person.prototype.constructor); // true

提到检测对象类型，还是instanceof操作符要更可靠一些。

## 创建对象-组合模式

创建自定义类型的最常见方式，就是组合使用构造函数模式与原型模式。构造函数模式用于定义实例属性，而原型模式用于定义共享的属性和方法。结果，每个实例都会有自己的一份实例属性的副本，但同时又共享着对方法的引用，最大限度地节省了内存。另外，这种混成模式还支持向构造函数传递参数；可谓是集两种模式之长。

## 继承

继承是OO 语言中的一个最为人津津乐道的概念。许多OO 语言都支持两种继承方式：接口继承和实现继承。接口继承只继承方法签名，而实现继承则继承实际的方法。  
如前所述，由于函数没有签名，在ECMAScript 中无法实现接口继承。ECMAScript 只支持实现继承，而且其实现继承主要是依靠原型链来实现的。

### 原型链继承

ECMAScript 中描述了原型链的概念，并将原型链作为实现继承的主要方法。其基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。  
原型链继承示例：

<script>  
 function Person() {  
 this.country = "China";  
 }  
  
 Person.prototype.getCountry = function () {  
 return "I love " + this.country;  
 }  
  
 function njPerson() {  
 this.city = "NanJing";  
 }  
  
 *// 继承了 Person* njPerson.prototype = new Person();  
  
 njPerson.prototype.getCity = function () {  
 return "I love " + this.city;  
 }  
  
 var *njPerson1* = new njPerson();  
 console.log(*njPerson1*.city); *// NanJing* console.log(*njPerson1*.getCity()); *// I love NanJing* console.log(*njPerson1*.country); *// China* console.log(*njPerson1*.getCountry()); *// I love China*</script>

### 借用构造函数

在解决原型中包含引用类型值所带来问题的过程中，开发人员开始使用一种叫做**借用构造函数**（constructor stealing）的技术（有时候也叫做伪造对象或经典继承）。这种技术的基本思想相当简单，即在子类型构造函数的内部调用超类型构造函数。如下所示：

<script>  
 function SuperType() {  
 this.colors = ["red", "blue", "green"];  
 }  
  
 function SubType() {  
 *// 继承了 SuperType* SuperType.call(this);  
 }  
  
 var *instancel* = new SubType();  
 *instancel*.colors.push("black");  
 console.log(*instancel*.colors); *// ["red", "blue", "green", "black"]* var *instance2* = new SubType();  
 console.log(*instance2*.colors); *// ["red", "blue", "green"]*</script>

**借用构造函数的问题**

如果仅仅是借用构造函数，那么也将无法避免构造函数模式存在的问题——方法都在构造函数中定义，因此函数复用就无从谈起了。考虑到这些问题，借用构造函数的技术也是很少单独使用的。

### 组合继承

组合继承（combination inheritance），有时候也叫做伪经典继承，指的是将原型链和借用构造函数的技术组合到一块，从而发挥二者之长的一种继承模式。其背后的思路是使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又能够保证每个实例都有它自己的属性。