基本语法

编写位置

我们目前学习的 JS 全都是客户端的 JS，也就是说全都是需要在浏览器中运行的，所以我们我们的 JS 代码全都需要在网页中编写。

我们的 JS 代码需要编写到<script>标签中。

我们一般将 script 标签写到 head 中。（和 style 标签有点像）

属性： 1. type：默认值 text/javascript 可以不写，不写也是这个值。 2. src：当需要引入一个外部的 js 文件时，使用该属性指向文件的地址。

案例

1. 创建一个 html 文件。
2. 在 html 文件的的 head 标签中创建一个 script 标签.。
3. console.log("Hello World");

严格区分大小写

JavaScript 是严格区分大小写的，也就是 abc 和 Abc 会被解析器认为是两个不同的东西。

注释

注释中的内容不会被解析器解析执行，但是会在源码中显示，我们一般会使用注释对程序中的内容进行解释。

```
// 单行注释
/**
 * 多行注释
 */
```

标识符

所谓标识符，就是指变量、函数、属性的名字，或函数的参数。

标识符的组成：数字、字母、下划线、$，不能以数字开头

按照惯例，ECMAScript 标识符采用**驼峰命名法。**

但是要注意的是*JavaScript*中的标识符不能是`关键字`和`保留字`符。

关键字：系统规定有特殊意义的词。

break、do、instanceof、typeof、case、else、new、var、catch、finally、return、void、continue、for、Switch、while、default、if、throw、delete、in、try、function、this、with、debugger、false、true、null

保留字：系统留用(可能永远也不会用, 但是你不能用)

class、enum、extends、super、const、export、import、inplements、let、private、public、yield、interface、package、protected、static

输出

```
console.log('输出')

alert('弹出')

document.write('写入')
```

变量

变量的作用是给某一个值或对象标注名称。

变量的声明：

```
// 使用var关键字声明一个变量
var a;
```

变量的赋值：

```
// 使用=为变量赋值
a = 123;
```

将上述结合起来就是：

```
var a = 123;
```

数据类型

数据类型决定了一个数据的特征，比如：123 和”123”，直观上看这两个数据都是 123，但实际上前者是一个数字，而后者是一个字符串。

对于不同的数据类型我们在进行操作时会有很大的不同。

五种基本数据类型：

1. 字符串（String）
2. 数值（Number）
3. 布尔值（Boolean）
4. Null
5. Undefined

对象类型：Object

typeof 运算符

使用 typeof 操作符可以用来检查一个变量的数据类型。

```
// 检查数值123的类型
typeof 123; // => 'number'
typeof "123"; // => 'string'
typeof true; // => 'boolean'
typeof undefined; // => undefined
typeof null; // => object
```

数据类型的转换

String

String 用于表示一个字符序列，即字符串。

字符串需要使用 '或" 括起来。

其他数值转换成字符串：

```
/**
 * 数值 => 字符串
 * 布尔值 => 字符串
 */
var num = 12;
typeof (num + "");
typeof String(num);
typeof num.toString();
```

思考：转义字符

转义字符	含义	转义字符	含义
\n	换行	\ \	斜杠
\t	制表	'	单引号
\b	空格	"	双引号
\r	回车		

Number

Number 类型用来表示整数和浮点数，最常用的功能就是用来表示 10 进制的整数和浮点数。

Number 表示的数字大小是有限的，范围是：
– ± 1.7976931348623157e+308
– 如果超过了这个范围，则会返回 ± Infinity。

NaN，即非数值（Not a Number）是一个特殊的数值，JS 中当对数值进行计算时没有结果返回，则返回 NaN。

```
/**
 * 字符串 => 数值
 * 布尔值 => 数值
 */
var s1 = "123ab";
var s2 = "ab123";
var s3 = "123";
var b1 = true;

typeof Number(s1); // => NaN
typeof Number(s2); // => NaN
typeof Number(s3); // => 123

typeof Number(b1);
```

```
parseInt(s1); // => 123
parseInt(s2); // => NaN
parseInt(s3); // => 123
parseInt(b1); // => NaN

b1 + 0; // => 1
```

Boolean

布尔型也被称为逻辑值类型或者真假值类型。

布尔型只能够取真（true）和假（false）两种数值。除此以外，其他的值都不被支持。

其他的数据类型也可以通过 Boolean()函数转换为布尔类型

数据类型	true	false
Boolean	true	false
String	任何非空字符	空字符串
Number	任何非 0 数字	0 和 NaN
Object	任何对象	-
Undefined		undefined

Undefined

Undefined 类型只有一个值，即特殊的 undefined 。

在使用 var 声明变量但未对其加以初始化时，这个变量的值就是 undefined。

typeof 对没有初始化和没有声明的变量都会返回 undefined

Null

Null 类型是第二个只有一个值的数据类型，这个特殊的值是 null 。

运算符

JS 中为我们定义了一套对数据进行运算的运算符。包括：算数运算符、位运算符、关系运算符等。

算数运算符

算数运算符顾名思义就是进行算数操作的运算符。

运算符	说明	运算符	说明
+	加法	++（前置）	自增
-	减法	++（后置）	自增
*	乘法	--（前置）	自减
/	除法	--（后置）	自减
%	取模	+	符号不变
		-	符号取反

自增和自减

自增和自减分为前置运算和后置元素

符号放在前面先进行自增或自减再进行运算，符号放在后面先运算再自增或者自减

运算之后，变量的值发生变化，这种效应叫做运算的副作用（side effect）。自增和自减运算符是仅有的两个具有副作用的运算符，其他运算符都不会改变变量的值。

逻辑操作符

一般情况下使用逻辑运算符会返回一个布尔值

逻辑运算符主要有三个：非、与、或。

在进行逻辑操作时如果操作数不是布尔类型则会将其转换布尔类型在进行计算。

运算符	说明	短路规则
!	逻辑非（not）	-
&&	逻辑与（AND）	第一个 false
||	逻辑或（or）	第一个 true

非

非运算符使用 ! 表示。

非运算符可以应用于任意值，无论值是什么类型，这个运算符都会返回一个布尔值。

非运算符会对原值取反

与

与运算符使用 && 表示。

与运算符可以应用于任何数据类型，且不一定返回布尔值。

或

或运算符使用 || 表示。

或运算符可以应用于任何数据类型，且不一定返回布尔值。

赋值运算符

简单的赋值操作符由等于号 （ = ） 表示，其作用就是把右侧的值赋给左侧的变量。

如果在等于号左边添加加减乘除等运算符，就可以完成复合赋值操作。

+=、*=、-=、/=、%=

关系运算符

小于（<） 、大于（>） 、小于等于（<=）和大于等于（>=）这几个关系运算符用于对两个值进行比较，比较的规则与我们在数学课上所学的一样。

JS 中使用==来判断两个值是否相等，如果相等则返回 true。

使用!=来表示两个值是否不相等，如果不等则返回 true。

表达式	值	表达式	值
null == undefined	true	true == 1	true
“NaN” == NaN	false	true == 2	false
5 == NaN	false	undefined == 0	false
NaN == NaN	false	null == 0	false
NaN != NaN	true	“5” == 5	true
false == 0	true		

除了==以外，JS 中还提供了===。

===表示全等，他和==基本一致，不过==在判断两个值时会进行自动的类型转换，而===不会。

也就是说”55”==55 会返回 true，而”55”===55 会返回 false；

同样我们还有!==表示不全等，同样比较时不会自动转型。

逗号

使用逗号可以在一条语句中执行多次操作。

比如：

```
var num1 = 1,
  num2 = 2,
  num3 = 3;
```

条件运算符

条件运算符也称为三元运算符。通常运算符写为?:。

这个运算符需要三个操作数，第一个操作数在?之前，第二个操作数在?和:之间，第三个操作数在:之后。

```
x > 0 ? x : -x; // 求x的绝对值
```

运算符的优先级

- .、[]、 new
- ()
- ++、 --
- !、~、+(单目)、-(单目)、typeof、void、delete
- %、*、/
- +(双目)、-(双目)
- << 、 >>、 >>>
- <、<=、>、>=
- ==、!==、===
- &
- ^
- |
- &&
- ||
- ?:
- =、+=、-=、*=、/=、%=、<<=、>>=、>>>=、&=、^=、|=
- ,

语句和代码块

前边我所说表达式和运算符等内容可以理解成是我们一门语言中的单词，短语。而**语句（*statement*）**就是我们这个语言中一句一句完
整的话了。

语句是一个程序的基本单位，JS 的程序就是由一条一条语句构成的，每一条语句使用;结尾。

JS 中的语句默认是由上至下顺序执行的，但是我们也可以通过一些流程控制语句来控制语句的执行顺序。

代码块是在大括号 {} 中所写的语句，以此将多条语句的集合视为一条语句来使用。

```
{
  var a = 123;
  a++;
  alert(a);
}
```

我们一般使用代码块将需要一起执行的语句进行分组，需要注意的是，代码块结尾不需要加分号。

条件语句

条件语句是通过判断指定表达式的值来决定执行还是跳过某些语句。

两种条件语句：

```
if...else
switch...case
```

if...else

if...else 语句是一种最基本的控制语句，它让 JavaScript 可以有条件的执行语句。

```
if (true) {
  //...
}

if (true) {
  //...
} else {
  //...
}

if (true) {
  //...
} else if (false) {
  //...
} else {
  //...
}
```

switch...case

switch...case 是另一种流程控制语句。

switch 语句更适用于多条分支使用同一条语句的情况。

需要注意的是 case 语句只是标识的程序运行的起点，并不是终点，所以一旦符合 case 的条件程序会一直运行到结束。所以我们一般会在 case 中添加 break 作为语句的结束。

```
var number = 10;
switch (number) {
  case 1:
    break;
  case 2:
    break;
  default:
    break;
}
```

循环语句

循环中的语句只要满足一定的条件将会一直执行。

while

while 语句是一个最基本的循环语句，也被称为 while 循环。

语法：

```
while (true) {
  // ...
}
```

和 if 一样 while 中的条件表达式将会被转换为布尔类型，只要该值为真，则代码块将会一直重复执行。

代码块每执行一次，条件表达式将会重新计算。

do...while

do...while 和 while 非常类似，只不过它会在循环的尾部而不是顶部检查表达式的值。

do...while 循环会至少执行一次。

```
do{
    //...
}while()
```

for 循环

for 语句也是循环控制语句，我们也称它为 for 循环。

大部分循环都会有一个计数器用以控制循环执行的次数，计数器的三个关键操作是初始化、检测和更新。for 语句就将这三步操作明确为了语法的一部分。

```
for(初始化表达式 ； 条件表达式 ； 更新表达式){
    语句...
}
```

break 和 continue

break 和 continue 语句用于在循环中精确地控制代码的执行。

使用 break 语句会使程序立刻退出最近的循环，强制执行循环后边的语句。

break 和 continue 语句只在循环和 switch 语句中使用。

使用 continue 语句会使程序跳过当次循环，继续执行下一次循环，并不会结束整个循环。

label

使用 label 语句可以在代码中添加标签，以便将来使用。

```
label: statement
---
```

标签通常与 break 语句和 continue 语句配合使用，跳出特定的循环。

```
top: for (var i = 0; i < 3; i++) {
  for (var j = 0; j < 3; j++) {
    if (i === 1 && j === 1) break top;
    console.log("i=" + i + ", j=" + j);
  }
}
// i=0, j=0
// i=0, j=1
// i=0, j=2
// i=1, j=0
foo: {
  console.log(1);
  break foo;
  console.log("本行不会输出");
```

```
}
console.log(2);
// 1
// 2
top: for (var i = 0; i < 3; i++) {
  for (var j = 0; j < 3; j++) {
    if (i === 1 && j === 1) continue top;
    console.log("i=" + i + ", j=" + j);
  }
}
// i=0, j=0
// i=0, j=1
// i=0, j=2
// i=1, j=0
// i=2, j=0
// i=2, j=1
// i=2, j=2
```