JavaScript Learning

变量

入门

变量就是存储数据的容器

```
var aa;  //声明变量
aa = 18; //赋值

//声明并赋值 一行
var age = 20;

//一次性声明多个变量并赋值
var h1 = 12, h2 = 13, h3 = 15;
```

命名规则

可以有：

- 数字
- 字母
- 下划线
- $

注意：

- 不能以数字开头
- 不能是关键字（for white if....）
- 必须有意义
- 大小写敏感
- 如果变量重名，后面的会替换前面的

驼峰法

> 小驼峰法：除第一个单词之外，其他单词首字母大写。 变量一般用小驼峰法标识。
>
> 譬如

```
var myStudentCount;
```

变量 myStudentCount 第一个单词是全部小写，后面的单词首字母大写。

大驼峰法：相比小驼峰法，大驼峰法把第一个单词的首字母也大写了。

常用于类名，函数名，属性，命名空间。

譬如

```
public class DataBaseUser;
```

交换变量值

```
var a = 1;
var b = 2;
var c = a;

a = b;  //从右向左流动
b = c;
```

数据类型

- 数值 Number
- 字符串 String
- 布尔 Boolen（true、false）
- undefined
- null
- 对象 Object （数组、函数）

原始类型

数值、字符串、布尔

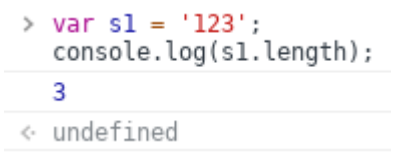
字符串

```
var name = 'lin' //使用单引号或双引号 引起来的就是字符串

//转义        \ -> 转义符号
var outPut = '你好，我是\'鞠亚\''
```

其他数据类型

```
var s1 = '123';
console.log(s1.length); //在控制台输出 s1 的长度
```




```
var s1 = '123';
var s2 = '456';
var s3 = s1+s2 //字符串拼接
console.log(s3);

var s4 = 1;
console.log(s4+s3); //仍是字符串拼接

var s5 = 2;
console.log(s4+s5); //加法运算
```

- +号既可以是数学加法运算 也可以是字符串拼接
- 变量数据类型为数值时为加法运算
- 变量数据类型为字符串时为字符串拼接
- 从前往后进行运算

布尔

ture false 大小写敏感

true 1

false 0

Undefined and Null

Undefined 表示一个声明了但没有赋值的变量

null 表示空， 变量的值若要想为null，必须手动设置

注释

单行与多行

```
// i'm cat tom

/*
  111
  111
*/
```

数据类型转换

转为字符串 (String)

数值转为字符串

```
var n = 5;
var s = n.toString(); // 数值转为字符串
console.log(typeof s); // typeof 显示数据类型

String(n); // 直接转化
```

另外一种方式

```
var n = 5;
var s = ''+n;  // 空字符串+数值（n）原数值（n）变字符串
console.log(typeof s);
```

布尔转为字符串

```
var n = true;
console.log(typeof n.toString());  //使用 .toString() 的方法
```

字符串转为数值

```
var a = '1';
var b = Number(a); // 注意这里
console.log(b);
```

转为数值

其一

Number()

```
var c = Number('c');
var d = Number(null);
var e = Number(undefined);

console.log(c,d,e);
```

```
> var c = Number('c');
  var d = Number(null);
  var e = Number(undefined);

  console.log(c,d,e);
  NaN 0 NaN
```

NaN -> Not a Number.

其二

parseInt() - 整数

```
var a = parseInt('2');
var b = parseInt('k23');
var c = parseInt(null);
var d = parseInt(undefined);

console.log(a,b,c,d);
```

```
var a = parseInt('2');
var b = parseInt('k23');
var c = parseInt(null);
var d = parseInt(undefined);

console.log(a,b,c,d);
2 NaN NaN NaN
```

其三

parseFloat() - 浮点

```
var a = parseFloat('1.23df');
var b = parseFloat('1.3.4.5');
var c = parseFloat('h34');
var d = parseFloat(null);
var e = parseFloat(undefined);

console.log(a,b,c,d,e);
```

```
var a = parseFloat('1.23df');
var b = parseFloat('1.3.4.5');
var c = parseFloat('h34');
var d = parseFloat(null);
var e = parseFloat(undefined);

console.log(a,b,c,d,e);
```

```
1.23 1.3 NaN NaN NaN
```

转为布尔

```
var a = Boolean('0');  // 字符串
var b = Boolean(0);  // 数值
var c = Boolean('2');  // 字符串（非 0 1 ）
var d = Boolean(null);
var e = Boolean(undefined);
var f = Boolean(2); // 数值

console.log(a,b,c,d,e,f);
```

```
var a = Boolean('0');  // 字符串
var b = Boolean(0);  // 数值
var c = Boolean('2');  // 字符串（非 0 1 ）
var d = Boolean(null);
var e = Boolean(undefined);
var f = Boolean(2); // 数值

console.log(a,b,c,d,e,f);
```

```
true false true false false true
```

解释...

> 字符串转布尔，只判断字符串是否为空，为空则 ***false*** ，否则为 ***true***
>
> 数值转布尔，**0** 则 ***false*** ，**1 或 以上** 则为 ***true***
>
> **null** 和 **undefined** 转布尔，二者表“空”的意思，则均为 ***false***

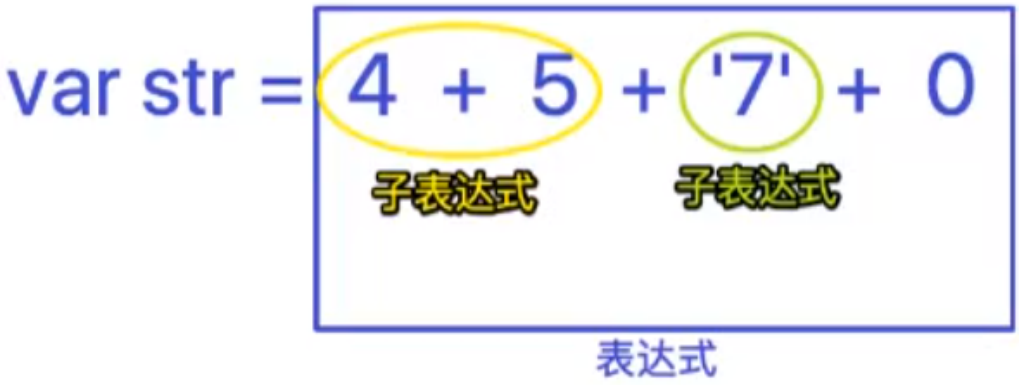
操作符

算数操作符

- 加（+）
- 减（-）
- 乘（*）
- 除（/）
- 取余数（%）

表达式：由 *值* 和 *操作符* 组成，运算有结果

子表达式：表达式中的每个数值及部分表达式




```
var s1 = 6;
var s2 = 3;

console.log(s1+s2);
```

一元运算符

适用于只有一个值的运算

对自身操作

操作符在变量前，先进行自身运算，再进行其他运算

操作符在变量后，则反之

```
n1++  ++n1 //等价 n1+1
n1--  --n1 //等价 n1-1
```

Example 1：

```
var n1 = 5;
++n1;
var n2 = 6;
console.log(n1+ ++n2); //等于13
console.log(n1+ n2++); //等于12
```

Example 2：

```
var a = 1;
var b = ++a + a++;
console.log(b);  //等于4
```

Example 3：

```
var a = 1;
var b = a++ + ++a;
console.log(b); //等于4
```

Example 4：

```
var a = 1;
//        2  + 2+1
var b = ++a + ++a;
console.log(b); // 等于5
```

逻辑运算符（布尔运算符）

&&

与：两个操作数同时为 **true**，结果为 **true**，否则为 **false**

||

或：两个操作数有一个为 **true**，结果则为 **true**，否则为 **false**

!

非：获取相反的结果

在 JavaScript 中，逻辑运算的结果是决定整个表达式的子表达式的值

运算优先级：先运算 && 再运算 ||

```
var a = 1;
var b = 2;
var c = 0;

console.log(a && b);  //结果为2
console.log(c && b);  //结果为0

console.log(a || b);  //结果为1
console.log(a || c);  //结果为1

console.log(a || c && b);
```

关系运算符

大于号 >

小于号 <

大于等于号 >=

小于等于号 <=

相等运算符

等于 == **只比较值**

不等于 !=

全等于 === **比较值和数据类型**

不全等 !==

```
var a = '2';
var b = 2;
console.log(a==b)
```

赋值运算

= 赋值

+= 加等于

-= 减等于

/= 除等于

%= 余数等于

```
var a = 1;
a+=4; // 等价于 a = a+4
```

运算符的优先级

从上到下

1. ( )
2. 一元运算符 ++ -- ！(非)
3. 算术运算符 先 * / % 后 + -
4. 关系运算符
5. 相等运算符
6. 逻辑运算符 **运算优先级：先运算 && 再运算 ||**
7. 赋值运算符

Example 1 :

```
var s = 4>=6 || 'human' != 'haha' && !(12*2+3 == 122) && true;
console.log(s); // 结果为true
```

Example 2 :

```
var n = 10;
var f = 5 == n / 2 && (2+2*n);
console.log(f); // 结果为22
```

流程控制

三种基本结构：

- 顺序结构：从上到下执行（默认）
- 分支结构：根据不同情况及判断，执行对应代码
- 循环结构：重复执行一段代码

分支结构

if

```
if(判断条件){
    要执行的代码 // 条件成立代码执行
}else if(判断条件){
  // 前方哪个条件满足，哪个执行
  // 若前方代码已有代码执行，则后面的代码无论是否满足条件，均不执行
}else{
  // 若前面所有条件均不成立，则执行 else 部分的代码
}
```

Example 1 :

```
var n1 = 100;
var n2 = 20;
if(n1>n2){
    console.log(n1);
}else{
    console.log(n2);
}
```

Example 2 :

```
var n = 10;
if(n%2 == 0){
    console.log('偶数')
}else{
    console.log('奇数')
}
```

Example 3 - 判断某一年是不是闰年：

闰年：能被4整除，但不能被100整除，亦或者是能被400整除

```
var Y = 2016;
if(Y%4 == 0){
    if(Y%100 != 0){
        console.log('闰年')
    }else if(Y%400 == 0){
        console.log('闰年')
    }else{
        console.log('平年')
    }
}else{
    console.log('平年')
}
```

switch - case

```
switch(值){
    case 值1:
        code;
        break;
    case 值2;
        code;
        break;
    default:
        code;
        break;
}
```

依次判断 switch 中的值与每个 case 的值是否相等，相等则执行对应的代码，若均不相等则执行 default 中的代码

break: 判断若符合，执行代码，后跳出，不再继续判断

Example 1 :

```
var grade = 'B';
switch(grade){
    case 'A':
        console.log('90分以上');
        break;
```

```
    case 'B':
        console.log('80-90之间');
        break;
    case 'C':
        console.log('70-80之间');
        break;
    default:
        console.log('不及格');
        break;
}
```

循环结构

三种

- while
- do...while
- for

while 和 do...while 一般用来解决无法确认次数的循环

for 循环一般在循环次数确定的时候比较方便

while

若条件成立，里面的代码就会一直反复执行，直到不满足条件

每次执行代码都要对条件进行重新判断

```
while(条件){
    code
}
```

Example 1 :

```
var i = 0;
while(i<10){
    console.log(i);
    i++;
}
console.log('End.')
```

Example 2 : **计算1-100之间所有数字之和**

```
var i = 1;
var s = 0;
while(i<=100){
    s = s+i
    i++;
}
console.log(s);
```

Example 3: **计算1-100以内7的倍数**

```
var i = 1;
while(i<=100){
    if(i%7 == 0){
        console.log(i);
    }
    i++;
}
```

Example 3: **计算1-100以内所有偶数的和**

```
var i = 0;
var s = 0;
while(i<=100){
    if(i%2 == 0){
        s+=i
    }
    i++;
}
console.log(s);
```

Example 4: **计算1-100以内能被3整除的数的和**

```
var i = 0;
var s = 0;
while(i<=100){
    if(i%3 == 0){
        s+=i
    }
    i++;
}
console.log(s);
```

do...while

先执行代码，再判断条件

若条件成立，代码继续执行；不成立，代码不执行

```
do{code}while(条件);
```

Example 1 :

```
var i = 10;
do{
    console.log(i);
    i++;
}while(i>10);
```

for

步骤：判断 -> 执行代码 -> 自增

```
for(初始表达式;判断表达式;自增自减运算){
    code
}
```

Example 1 :

```
for(var i = 1;i<10;i++){
    console.log(i);
}
```

Example 2 : **计算1-100之间所有数字之和**

```
var s = 0;
for(var i = 0;i<=100;i++){
    s+=i;
}
console.log(s);
```

Example 3 : **计算1-100以内所有偶数的和**

```
var s = 0;
for(var i = 0;i<=100;i++){
    if(i%2 == 0){
        s+=i
    }
}
console.log(s);
```

Example 4 : **正方形**

```
var s = '';
for(var i = 0;i<10;i++){
    for(var h = 0;h<10;h++){
        s+=' * ';
    }
    s+='\n';
}
console.log(s);
```

Example 4 : **三角形**

```
var s = '';
for(var i = 0;i<10;i++){
    for(var h = i;h<10;h++){
        s+='*';
    }
    s+='\n';
}
console.log(s);
```

Example 4 : **九九乘法表**

```
var str = '';
for(var i = 1;i<=9;i++){
    for(var j = i;j<=9;j++){
    str += i + '*' + j + '=' + (i*j) + '\t';
    }
    str += '\n'
}
console.log(str);
```

continue & break

break:立即跳出整个循环，即循环结束，开始执行循环后面的内容（直接跳到大括号）

continue:立即跳出当前循环，继续下一次循环（跳到 `i++` 的地方）

Example 1 : **100以内 *不能* 被7整除的所有数的和**

```
var s = 0;
for(var i = 0;i<=100;i++){
    if(i%7 == 0){
        continue;
    }
    s+=i;
}
console.log(s)
```

Example 2 : **200-300之间 *第一个* 可被7整除的数字**

```
for(var i = 200;i<300;i++){
    if(i%7 == 0){
        console.log(i);
        break;
    }
}
```

数组（特殊的对象）

> 数组，就是将多个元素（通常是同一类型）按一定顺序排列放在一个集合中。

```
// 存储
//          0   1   2
var arr = ['a','b','c'];
var arr2 = [1,2,3];
// 读取
console.log(arr[1]);
```

创建数组

自变量方式

```
var a1 = []; // 空数组
var a1 = ['a',1];

console.log(a1);
```

构造函数方式

```
var a1 = new Array(1,3,'h','k');
console.log(a1);
```

获取数组长度

```
var a1 = ['a',1];
var l = a1.length;
console.log(l);
```

多维数组

> 多维数组：数组中又有数组（套娃

```
var a1 = [1,4,'k','l'];
var a2 = [6,7,a1,'t']; // 二维数组
var a3 = [4,a2,'p']; // 三维数组
console.log(a2);
console.log(a3);
```

获取数组元素

```
var a1 = ['red','green','yellow'];
console.log(a1[0]);
//          0    1    2(整个数组)
//                    0     1
var a2 = ['i','am',['so','tired',
['javascript','is','best']]];
console.log(a2[2][1]);
```

> 下标：数组内容的标号

遍历数组元素

三个方法

```
var a1 = ['a','b','c','d','e','f'];
// 将数组中所有元素逐个打印
// for
for(var i = 0;i<=a1.length;i++){
    console.log(a1[i]);
}
// while
var i = 0;
while(i<=a1.length){
   console.log(a1[i]);
    i++;
}
// for...in
for(var n in a1){
    console.log(a1[n]);
}
```

重点：获取数组的长度 `array.length`

Example 1 - 求数组元素的和

```
var arr = [12,26,1,7,8,4];
var n = 0;
for(var i=0;i<arr.length;i++){
    n+=arr[i];
}
console.log(n);
```

Example 2 - 求数组中最大的值

```
var arr = [12,26,1,7,8,4];
var n = 0;
for(var i=0;i<arr.length;i++){
    if(n<arr[i]){
        n=arr[i];
    }
}
console.log(n);
```

Example 3 - 求数组中所有的偶数

```
var arr = [12,26,1,7,8,4];
for(var i = 0;i<arr.length;i++){
    if(arr[i]%2 == 0){
        console.log(arr[i]);
    }
}
```

Example 4 - 将数组中的元素以 | 分割为一个字符串

```
var arr = [12,26,1,7,8,4];
var s = '';
for(var i=0;i<arr.length;i++){
    s += arr[i] + '|';
}
console.log(s);
```

函数

> 函数：封装一段代码，将来可重复使用

声明

```
function 函数名(形式参数1,形式参数2,...){} // 关键字声明
var 变量名（也可作为函数名） = function(){} // 表达式声明
```

调用

```
function f1(){
    console.log("I'm Cat Tom.")
}
f1(实际参数1,实际参数2,...); //调用
```

形式参数 & 实际参数

形式参数：在声明函数时使用，值不固定，与实际参数传入的值要一一对应

实际参数：函数调用时，实际传入函数中的值，传入后，在函数中使用形式参数获取具体的值

Example:

```
function n(k){
    var s = 0;
for(var i = 0;i<=k;i++){
    s+=i;
}
console.log(s);
}
n(50);
```

返回值

```
function 函数名(形式参数1,形式参数2,...){
    retrun 返回值;
}
var re = f1(实际参数1,实际参数2,...); // re 变量即返回值
```

Example :

```
function f(a,b){
    var c = a-b
    return c;
    console.log("I'm Cat Tom.")
}
var h = f(5,2);
console.log(h);
```

注意

> 若函数中没有 retrun ，那么函数调用后接到的返回值就是 undefined
>
> 若函数中有 retrun ，但 retrun 后无值，那么函数调用后接到的返回值还是 undefined
>
> 函数中 retrun 后，不管有什么代码，均不执行
>
> return 后，函数的调用结束

匿名函数

> 本身没有名字的函数。

```
var 变量名（也可作为函数名） = function(){}
变量名（也可作为函数名）();
```

自调用

```
// 自调用匿名函数
(
    function(){
    alert("I'm Cat Tom.");
}
)();
```

> 第一组括号将匿名函数括起来，视作一个整体
>
> 在这个整体后加 `()` 即可自调用

函数也是一种数据类型

```
function fn(){}
console.log(typeof fn); // 结果为 function
```

Example 1 - 回调

```
function f1(s){
    s();
}
var f2 = function(){
    alert("I'm Cat Tom.");
}
// f2 函数会被当作值，传入 f1 函数内
f1(f2);
```

Example 2 - 闭包

```
function f1(){
    var a = 10;
    var f2 = function(){
    alert("I'm Cat Tom.");
  }
  return f2; // 将函数作为返回值
}
var k = f1();
k();
```

全局变量 & 局部变量

> 在任何地方都可以访问到的变量就是**全局变量**
>
> 全局变量所在的区域就是**全局作用域**
>
> 只在固定的代码片段中才可访问到的变量就是**局部变量**，例如函数内部的变量
>
> 局部变量所在的区域就是**局部作用域**（也称函数作用域）

> 变量退出作用域后会销毁，全局变量退出程序后才会销毁

代码运行的阶段

1. 解析（编译）阶段：
    - 语法检查
    - 变量及函数进行声明
2. 运行阶段
    - 变量赋值
    - 代码流程的执行

```
console.log(a);
var a = 10;
// 等价 ↓
var a;
console.log(a);
a = 2;
```

变量提升

> 在代码执行前，变量在编译阶段已经被声明

Example 1 :

```
var a = 12;
function abc(){
    alert(a);
    var a = 10;
}
abc();
```

> 问：弹窗结果是?
>
> 答：undefined

若局部作用域中已有相同变量声明，那么全局作用域中声明的变量不生效

Example 2 :

```
console.log(a); // 函数已被调用?
function a(){
    console.log('b');
}
var a = 1;
console.log(a); // 输出函数 a
```

若函数与变量同名，那么函数声明会替换变量声明

Example 3 :

```
// console.log(a);
function a(){
    console.log('b');
}
var a = 1;
console.log(a); // 输出 1
```

函数没有被调用，自然输出的是 `1`

作用域 & 作用域链

> 懒得写了，累了...
>
> 只有函数可以制造作用域结构， 那么只要是代码，就至少有一个作用域，即全局作用域。凡是代码中有函数，那么这个函数就构成另一个作用域。如果函数中还有函数，那么在这个函数中就又可以诞生一个作用域。
>
> 将这样的所有的作用域列出来，可以有一个结构：函数内指向函数外的链式结构。就称作作用域链。

```
var a = 1;
function f1(){
    var a = 2;
    function f2(){
        var a = 3;
        function f3(){
```

```
            var a = 4;
            console.log(a);
        }
        f3();
    }
    f2();
}
f1();
```

> 当函数中使用给某个变量时，优先在函数自己的作用域中查询
>
> 若找不到，就向上一层作用域查找，直到全局作用域

对象

概念

看不懂。

不如实操。

声明

对象中的数据都是键值成对存在的

通常来说，若值为函数，那么称为方法(Method)

其他类型的值都称为属性

自变量声明对象

```
var obj1 = {age:18,height:190,name:'Cat Tom',Method:function(){}};
```

实例化 *内置构造函数* 方式声明对象

```
var obj2 = new Object(); // Object() 即是内置构造函数
```

实例化 *自定义构造函数* 方式声明对象

```
function Fun(){} // Fun() 便是我们自定义构造函数
var f new Fun();
```

使用

```
对象.属性名
```

```
var obj1 = {
    age:18,
    height:190,
    name:'Cat Tom',
    Method:function(){
          console.log("I'm Cat Tom.");
    }
};
console.log(obj1.height);
obj1.Method();
```

this

this 是个对象

```
function f(){
    console.log(this);
}
f();
```

`this` 永远指向一个对象

普通的函数中也有 `this` ，此时 `this` 指向全局对象`window`

this 的指向

```
var obj1 = {
    age:18,
    height:190,
    name:'Cat Tom',
    Method:function(){
          var age = this.age;
          console.log(age);
    }
};
obj1.Method();
```

在方法中的 this 指的是该方法所在的对象

```
k = 'a';

function fun(){
    var k = 'b'
    console.log(this.k);
}

var o1 = {
    k:'c',
    f:fun,
}
var o2 = {
    k:'d',
    f:fun,
}

o1.f();
o2.f();
```

this 运行在哪个对象下，就指向哪个对象

遍历

for...in 循环

```
for(键 in 对象);
```

for...in 循环不仅可以循环遍历对象，还可以循环遍历数组

```
var obj1 = {
    age:18,
    height:190,
    name:'Cat Tom',
};
for(var n in obj1){
    console.log(obj1[n]);
}
```

删除

```
var obj1 = {
    age:18,
    height:190,
    name:'Cat Tom',
};
delete obj1.age; // 删除对象的属性
console.log(obj1);
```

包装

原始类型的数据在一定条件下可自动转为对象

可自动当做对象使用，可调用各种属性和方法

包装对象使用完成后，会自动销毁

```
var a = '456';
a.length;

var v1 = new Number(123);
console.log(v1);
```

标准库对象（内置对象）

- Math
- Array
- Number

- String
- Boolean
- ...

有疑问就到 MDN 或 MSDN 上查

活用百度谷歌 /doge

获取 n 至 m 的随机值

```
Math.random() * (m-n) + n;
```

random 方法得到的是浮点数

可利用 floor 方法取整

实例化构造函数获取时间对象

```
var date = new Date();
console.log(date);
```