

01 快速入门

JavaScript代码可以直接嵌在网页的任何地方，不过通常我们都把JavaScript代码放到`中：

```
<html>
<head>
  <script>
    alert('Hello, world');
  </script>
</head>
<body>
  ...
</body>
</html>
```

由...包含的代码就是JavaScript代码，它将直接被浏览器执行。

第二种方法是把JavaScript代码放到一个单独的.js文件，然后在HTML中通过`引入这个文件：

```
<html>
<head>
  <script src="/static/js/abc.js"></script>
</head>
<body>
  ...
</body>
</html>
```

这样，/static/js/abc.js就会被浏览器执行。

把JavaScript代码放入一个单独的.js文件中更利于维护代码，并且多个页面可以各自引用同一份.js文件。

可以在同一个页面中引入多个.js文件，还可以在页面中多次编写js代码...，浏览器按照顺序依次执行。

有些时候你会看到`标签还设置了一个type`属性：

```
<script type="text/javascript">
  ...
</script>
```

但这是没有必要的，因为默认的type就是JavaScript，所以不必显式地把type指定为JavaScript。

如何编写JavaScript

可以用任何文本编辑器来编写JavaScript代码。这里我们推荐以下几种文本编辑器：

Visual Studio Code

微软出的[Visual Studio Code](#)，可以看做迷你版Visual Studio，免费！跨平台！内置JavaScript支持，强烈推荐使用！

Sublime Text

[Sublime Text](#)是一个好用的文本编辑器，免费，但不注册会不定时弹出提示框。

Notepad++

[Notepad++](#)也是免费的文本编辑器，但仅限Windows下使用。

*注意：*不可以用Word或写字板来编写JavaScript或HTML，因为带格式的文本保存后不是*纯文本文件*，无法被浏览器正常读取。也尽量不要用记事本编写，它会自作聪明地在保存UTF-8格式文本时添加BOM头。

如何运行JavaScript

要让浏览器运行JavaScript，必须先有一个HTML页面，在HTML页面中引入JavaScript，然后，让浏览器加载该HTML页面，就可以执行JavaScript代码。

你也许会想，直接在我的硬盘上创建好HTML和JavaScript文件，然后用浏览器打开，不就可以看到效果了吗？

这种方式运行部分JavaScript代码没有问题，但由于浏览器的安全限制，以`file://`开头的地址无法执行如联网等JavaScript代码，最终，你还是需要架设一个Web服务器，然后以`http://`开头的地址来正常执行所有JavaScript代码。

不过，开始学习阶段，你无须关心如何搭建开发环境的问题，我们提供在页面输入JavaScript代码并直接运行的功能，让你专注于JavaScript的学习。

试试直接点击“Run”按钮执行下面的JavaScript代码：

```
// 以双斜杠开头直到行末的是注释，注释是给人看的，会被浏览器忽略
/* 在这中间的也是注释，将被浏览器忽略 */
// 第一个JavaScript代码：
alert('Hello, world'); // 观察执行效果
```

浏览器将弹出一个对话框，显示“Hello, world”。你也可以修改两个单引号中间的内容，再试着运行。

调试

俗话说得好，“工欲善其事，必先利其器。”，写JavaScript的时候，如果期望显示ABC，结果却显示XYZ，到底代码哪里出了问题？不要抓狂，也不要泄气，作为小白，要坚信：JavaScript本身没有问题，浏览器执行也没有问题，有问题的一定是我的代码。

如何找出问题代码？这就需要调试。

怎么在浏览器中调试JavaScript代码呢？

首先，你需要安装Google Chrome浏览器，Chrome浏览器对开发者非常友好，可以让你方便地调试JavaScript代码。从这里[下载Chrome浏览器](#)。打开网页出问题的童鞋请移步[国内镜像](#)。

安装后，随便打开一个网页，然后点击菜单“查看(View)”-“开发者(Developer)”-“开发者工具(Developer Tools)”，浏览器窗口就会一分为二，下方就是开发者工具：



先点击“控制台(Console)”，在这个面板里可以直接输入JavaScript代码，按回车后执行。

要查看一个变量的内容，在Console中输入`console.log(a);`，回车后显示的值就是变量的内容。

关闭Console请点击右上角的“×”按钮。请熟练掌握Console的使用方法，在编写JavaScript代码时，经常需要在Console运行测试代码。

如果你对自己还有更高的要求，可以研究开发者工具的“源码(Sources)”，掌握断点、单步执行等高级调试技巧。

练习

打开[新浪首页](#)，然后查看页面源代码，找一找引入的JavaScript文件和直接编写在页面中的JavaScript代码。然后在Chrome中打开开发者工具，在控制台输入`console.log('Hello');`，回车查看JavaScript代码执行结果。

基本语法

语法

JavaScript的语法和Java语言类似，每个语句以`;`结束，语句块用`{...}`。但是，JavaScript并不强制要求在每个语句的结尾加`;`，浏览器中负责执行JavaScript代码的引擎会自动在每个语句的结尾补上`;`。

让JavaScript引擎自动加分号在某些情况下会改变程序的语义，导致运行结果与期望不一致。在本教程中，我们不会省略`;`，所有语句都会添加`;`。

例如，下面的一行代码就是一个完整的赋值语句：

```
var x = 1;
```

下面的一行代码是一个字符串，但仍然可以视为一个完整的语句：

```
'Hello, world';
```

下面的一行代码包含两个语句，每个语句用`;`表示语句结束：

```
var x = 1; var y = 2; // 不建议一行写多个语句！
```

语句块是一组语句的集合，例如，下面的代码先做了一个判断，如果判断成立，将执行`{...}`中的所有语句：

```
if (2 > 1) {  
    x = 1;  
    y = 2;  
    z = 3;  
}
```

注意花括号`{...}`内的语句具有缩进，通常是4个空格。缩进不是JavaScript语法要求必须的，但缩进有助于我们理解代码的层次，所以编写代码时要遵守缩进规则。很多文本编辑器具有“自动缩进”的功能，可以帮助整理代码。

`{...}`还可以嵌套，形成层级结构：

```
if (2 > 1) {  
    x = 1;  
    y = 2;  
    z = 3;  
    if (x < y) {  
        z = 4;  
    }  
    if (x > y) {  
        z = 5;  
    }  
}
```

JavaScript本身对嵌套的层级没有限制，但是过多的嵌套无疑会大大增加看懂代码的难度。遇到这种情况，需要把部分代码抽出来，作为函数来调用，这样可以减少代码的复杂度。

注释

以`//`开头直到行末的字符被视为行注释，注释是给开发人员看到，JavaScript引擎会自动忽略：

```
// 这是一行注释
alert('hello'); // 这也是注释
```

另一种块注释是用`/*...*/`把多行字符包裹起来，把一大“块”视为一个注释：

```
/* 从这里开始是块注释
   仍然是注释
   仍然是注释
   注释结束 */
```

练习：

分别利用行注释和块注释把下面的语句注释掉，使它不再执行：

```
// 请注释掉下面的语句：
alert('我不想执行');
alert('我也不想执行');
```

大小写

请注意，JavaScript严格区分大小写，如果弄错了大小写，程序将报错或者运行不正常。

数据类型和变量

数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在JavaScript中定义了以下几种数据类型：

Number

JavaScript不区分整数和浮点数，统一用Number表示，以下都是合法的Number类型：

```
123; // 整数123
0.456; // 浮点数0.456
1.2345e3; // 科学计数法表示1.2345x1000，等同于1234.5
-99; // 负数
NaN; // NaN表示Not a Number，当无法计算结果时用NaN表示
Infinity; // Infinity表示无限大，当数值超过了JavaScript的
Number所能表示的最大值时，就表示为Infinity
```

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：`0xff00`，`0xa5b4c3d2`，等等，它们和十进制表示的数值完全一样。

Number可以直接做四则运算，规则和数学一致：

```
1 + 2; // 3
(1 + 2) * 5 / 2; // 7.5
2 / 0; // Infinity
0 / 0; // NaN
10 % 3; // 1
10.5 % 3; // 1.5
```

注意`%`是求余运算。

字符串

字符串是以单引号'或双引号"括起来的任意文本，比如'`abc`'，"`xyz`"等等。请注意，`''`或`""`本身只是一种表示方式，不是字符串的一部分，因此，字符串'`abc`'只有`a`，`b`，`c`这3个字符。

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有`true`、`false`两种值，要么是`true`，要么是`false`，可以直接用`true`、`false`表示布尔值，也可以通过布尔运算计算出来：

```
true; // 这是一个true值
false; // 这是一个false值
2 > 1; // 这是一个true值
2 >= 3; // 这是一个false值
```

`&&`运算是与运算，只有所有都为`true`，`&&`运算结果才是`true`：

```
true && true; // 这个&&语句计算结果为true
true && false; // 这个&&语句计算结果为false
false && true && false; // 这个&&语句计算结果为false
```

`||`运算是或运算，只要其中有一个为`true`，`||`运算结果就是`true`：

```
false || false; // 这个||语句计算结果为false
true || false; // 这个||语句计算结果为true
false || true || false; // 这个||语句计算结果为true
```

! 运算是非运算，它是一个单目运算符，把 **true** 变成 **false**，**false** 变成 **true**：

```
! true; // 结果为false
! false; // 结果为true
!(2 > 5); // 结果为true
```

布尔值经常用在条件判断中，比如：

```
var age = 15;
if (age >= 18) {
    alert('adult');
} else {
    alert('teenager');
}
```

比较运算符

当我们对 **Number** 做比较时，可以通过比较运算符得到一个布尔值：

```
2 > 5; // false
5 >= 2; // true
7 == 7; // true
```

实际上，JavaScript 允许对任意数据类型做比较：

```
false == 0; // true
false === 0; // false
```

要特别注意相等运算符 **==**。JavaScript 在设计时，有两种比较运算符：

第一种是 **==** 比较，它会自动转换数据类型再比较，很多时候，会得到非常诡异的结果：

第二种是 **===** 比较，它不会自动转换数据类型，如果数据类型不一致，返回 **false**，如果一致，再比较。

由于 JavaScript 这个设计缺陷，不要使用 **==** 比较，始终坚持使用 **===** 比较。

另一个例外是 **NaN** 这个特殊的 **Number** 与所有其他值都不相等，包括它自己：

```
NaN === NaN; // false
```

唯一能判断 **NaN** 的方法是通过 **isNaN()** 函数：

```
isNaN(NaN); // true
```

最后要注意浮点数的相等比较：

```
1 / 3 === (1 - 2 / 3); // false
```

这不是JavaScript的设计缺陷。浮点数在运算过程中会产生误差，因为计算机无法精确表示无限循环小数。要比较两个浮点数是否相等，只能计算它们之差的绝对值，看是否小于某个阈值：

```
Math.abs(1 / 3 - (1 - 2 / 3)) < 0.0000001; // true
```

null和undefined

`null`表示一个“空”的值，它和`0`以及空字符串`''`不同，`0`是一个数值，`''`表示长度为0的字符串，而`null`表示“空”。

在其他语言中，也有类似JavaScript的`null`的表示，例如Java也用`null`，Swift用`nil`，Python用`None`表示。但是，在JavaScript中，还有一个和`null`类似的`undefined`，它表示“未定义”。

JavaScript的设计者希望用`null`表示一个空的值，而`undefined`表示值未定义。事实证明，这并没有什么卵用，区分两者的意义不大。大多数情况下，我们都应该用`null`。`undefined`仅仅在判断函数参数是否传递的情况下有用。

数组

数组是一组按顺序排列的集合，集合的每个值称为元素。JavaScript的数组可以包括任意数据类型。例如：

```
[1, 2, 3.14, 'Hello', null, true];
```

上述数组包含6个元素。数组用`[]`表示，元素之间用`,`分隔。

另一种创建数组的方法是通过`Array()`函数实现：

```
new Array(1, 2, 3); // 创建了数组[1, 2, 3]
```

然而，出于代码的可读性考虑，强烈建议直接使用`[]`。

数组的元素可以通过索引来访问。请注意，索引的起始值为`0`：

```
var arr = [1, 2, 3.14, 'Hello', null, true];  
arr[0]; // 返回索引为0的元素，即1  
arr[5]; // 返回索引为5的元素，即true  
arr[6]; // 索引超出了范围，返回undefined
```


对象

JavaScript的对象是一组由键-值组成的无序集合，例如：

```
var person = {  
  name: 'Bob',  
  age: 20,  
  tags: ['js', 'web', 'mobile'],  
  city: 'Beijing',  
  hasCar: true,  
  zipcode: null  
};
```

JavaScript对象的键都是字符串类型，值可以是任意数据类型。上述 `person` 对象一共定义了6个键值对，其中每个键又称为对象的属性，例如，`person` 的 `name` 属性为 `'Bob'`，`zipcode` 属性为 `null`。

要获取一个对象的属性，我们用 `对象变量.属性名` 的方式：

```
person.name; // 'Bob'  
person.zipcode; // null
```

变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在JavaScript中就是用一个变量名表示，变量名是大小写英文、数字、`$` 和 `_` 的组合，且不能用数字开头。变量名也不能是JavaScript的关键词，如 `if`、`while` 等。申明一个变量用 `var` 语句，比如：

```
var a; // 申明了变量a，此时a的值为undefined  
var $b = 1; // 申明了变量$b，同时给$b赋值，此时$b的值为1  
var s_007 = '007'; // s_007是一个字符串  
var Answer = true; // Answer是一个布尔值true  
var t = null; // t的值是null
```

变量名也可以用中文，但是，请不要给自己找麻烦。

在JavaScript中，使用等号 `=` 对变量进行赋值。可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，但是要注意只能用 `var` 申明一次，例如：

```
var a = 123; // a的值是整数123  
a = 'ABC'; // a变为字符串
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下：

```
int a = 123; // a是整数类型变量，类型用int申明
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
var x = 10;
x = x + 2;
```

如果从数学上理解 $x = x + 2$ 那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式 $x + 2$ ，得到结果 **12**，再赋给变量 **x**。由于 **x** 之前的值是 **10**，重新赋值后，**x** 的值变成 **12**。

要显示变量的内容，可以用 `console.log(x)`，打开Chrome的控制台就可以看到结果。

```
// 打印变量x
var x = 100;
console.log(x);
```

使用 `console.log()` 代替 `alert()` 的好处是可以避免弹出烦人的对话框。

strict模式

JavaScript在设计之初，为了方便初学者学习，并不强制要求用 `var` 申明变量。这个设计错误带来了严重的后果：如果一个变量没有通过 `var` 申明就被使用，那么该变量就自动被申明为全局变量：

```
i = 10; // i现在是全局变量
```

在同一个页面的不同的JavaScript文件中，如果都不用 `var` 申明，恰好都使用了变量 **i**，将造成变量 **i** 互相影响，产生难以调试的错误结果。

使用 `var` 申明的变量则不是全局变量，它的范围被限制在该变量被申明的函数体内（函数的概念将稍后讲解），同名变量在不同的函数体内互不冲突。

为了修补JavaScript这一严重设计缺陷，ECMA在后续规范中推出了 **strict** 模式，在 **strict** 模式下运行的JavaScript代码，强制通过 `var` 申明变量，未使用 `var` 申明变量就使用的，将导致运行错误。

启用 **strict** 模式的方法是在JavaScript代码的第一行写上：

```
'use strict';
```

这是一个字符串，不支持 **strict** 模式的浏览器会把它当做一个字符串语句执行，支持 **strict** 模式的浏览器将开启 **strict** 模式运行JavaScript。

来测试一下你的浏览器是否能支持strict模式：

```
'use strict';  
// 如果浏览器支持strict模式，  
// 下面的代码将报ReferenceError错误：  
abc = 'Hello, world';  
console.log(abc);
```

运行代码，如果浏览器报错，请修复后再运行。如果浏览器不报错，说明你的浏览器太古老了，需要尽快升级。

不用var声明的变量会被视为全局变量，为了避免这一缺陷，所有的JavaScript代码都应该使用strict模式。我们在后面编写的JavaScript代码将全部采用strict模式。

字符串

JavaScript的字符串就是用''或""括起来的字符表示。

如果'本身也是一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是I, ' , m, 空格, O, K这6个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

```
'I\'m \"OK\"!';
```

表示的字符串内容是：I'm "OK"!

转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以\\表示的字符就是\。

ASCII字符可以以\x##形式的十六进制表示，例如：

```
'\x41'; // 完全等同于 'A'
```

还可以用\u####表示一个Unicode字符：

```
'\u4e2d\u6587'; // 完全等同于 '中文'
```

多行字符串

由于多行字符串用\n写起来比较费事，所以最新的ES6标准新增了一种多行字符串的表示方法，用反引号`*...`表示：

```
`这是一个  
多行  
字符串`;
```

注意：反引号在键盘的ESC下方，数字键1的左边：

ESC	F1	F2	F3	F4
~	!	@	#	\$
`	1	2	3	4
tab	Q	W	E	
caps lock	A	S		

练习：测试你的浏览器是否支持ES6标准，如果不支持，请把多行字符串用\n重新表示出来：

```
// 如果浏览器不支持ES6，将报SyntaxError错误：  
console.log(`多行  
字符串  
测试`);
```

模板字符串

要把多个字符串连接起来，可以用+号连接：

```
var name = '小明';  
var age = 20;  
var message = '你好，' + name + '，你今年' + age + '岁了!';  
alert(message);
```

如果有很多变量需要连接，用+号就比较麻烦。ES6新增了一种模板字符串，表示方法和上面的多行字符串一样，但是它会自动替换字符串中的变量：

```
var name = '小明';  
var age = 20;  
var message = `你好，${name}，你今年${age}岁了!`;   
alert(message);
```

练习：测试你的浏览器是否支持ES6模板字符串，如果不支持，请把模板字符串改为+连接的普通字符串：

```
'use strict';

// 如果浏览器支持模板字符串，将会替换字符串内部的变量：
var name = '小明';
var age = 20;
console.log(`你好，${name}，你今年${age}岁了!`);
```

操作字符串

字符串常见的操作如下：

```
var s = 'Hello, world!';
s.length; // 13
```

要获取字符串某个指定位置的字符，使用类似Array的下标操作，索引号从0开始：

```
var s = 'Hello, world!';

s[0]; // 'H'
s[6]; // ' '
s[7]; // 'w'
s[12]; // '!'
s[13]; // undefined 超出范围的索引不会报错，但一律返回undefined
```

需要特别注意的是，字符串是不可变的，如果对字符串的某个索引赋值，不会有任何错误，但是，也没有任何效果：

```
var s = 'Test';
s[0] = 'X';
alert(s); // s仍然为'Test'
```

JavaScript为字符串提供了一些常用方法，注意，调用这些方法本身不会改变原有字符串的内容，而是返回一个新字符串：

toUpperCase

`toUpperCase()` 把一个字符串全部变为大写：

```
var s = 'Hello';
s.toUpperCase(); // 返回'HELLO'
```

toLowerCase

`toLowerCase()` 把一个字符串全部变为小写：

```
var s = 'Hello';  
var lower = s.toLowerCase(); // 返回'hello'并赋值给变量lower  
lower; // 'hello'
```

indexOf

`indexOf()` 会搜索指定字符串出现的位置：

```
var s = 'hello, world';  
s.indexOf('world'); // 返回7  
s.indexOf('world'); // 没有找到指定的子串，返回-1
```

substring

`substring()` 返回指定索引区间的子串：

```
var s = 'hello, world'  
s.substring(0, 5); // 从索引0开始到5（不包括5），返回'hello'  
s.substring(7); // 从索引7开始到结束，返回'world'
```

数组

JavaScript的 `Array` 可以包含任意数据类型，并通过索引来访问每个元素。

要取得 `Array` 的长度，直接访问 `length` 属性：

```
var arr = [1, 2, 3.14, 'Hello', null, true];  
arr.length; // 6
```

请注意，直接给 `Array` 的 `length` 赋一个新的值会导致 `Array` 大小的变化：

```
var arr = [1, 2, 3];  
arr.length; // 3  
arr.length = 6;  
arr; // arr变为[1, 2, 3, undefined, undefined, undefined]  
arr.length = 2;  
arr; // arr变为[1, 2]
```

`Array` 可以通过索引把对应的元素修改为新的值，因此，对 `Array` 的索引进行赋值会直接修改这个 `Array`：

```
var arr = ['A', 'B', 'C'];  
arr[1] = 99;  
arr; // arr现在变为['A', 99, 'C']
```

请注意，如果通过索引赋值时，索引超过了范围，同样会引起 `Array` 大小的变化：

```
var arr = [1, 2, 3];
arr[5] = 'x';
arr; // arr变为[1, 2, 3, undefined, undefined, 'x']
```

大多数其他编程语言不允许直接改变数组的大小，越界访问索引会报错。然而，JavaScript的Array却不会有任何错误。在编写代码时，不建议直接修改Array的大小，访问索引时要确保索引不会越界。

indexOf

与String类似，Array也可以通过indexOf()来搜索一个指定的元素的位置：

```
var arr = [10, 20, '30', 'xyz'];
arr.indexOf(10); // 元素10的索引为0
arr.indexOf(20); // 元素20的索引为1
arr.indexOf(30); // 元素30没有找到，返回-1
arr.indexOf('30'); // 元素'30'的索引为2
```

注意了，数字30和字符串'30'是不同的元素。

slice

slice()就是对应String的substring()版本，它截取Array的部分元素，然后返回一个新的Array：

```
var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
arr.slice(0, 3); // 从索引0开始，到索引3结束，但不包括索引3：
['A', 'B', 'C']
arr.slice(3); // 从索引3开始到结束：['D', 'E', 'F', 'G']
```

注意到slice()的起止参数包括开始索引，不包括结束索引。

如果不给slice()传递任何参数，它就会从头到尾截取所有元素。利用这一点，我们可以很容易地复制一个Array：

```
var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
var aCopy = arr.slice();
aCopy; // ['A', 'B', 'C', 'D', 'E', 'F', 'G']
aCopy === arr; // false
```

push和pop

push()向Array的末尾添加若干元素，pop()则把Array的最后一个元素删除掉：

```
var arr = [1, 2];
arr.push('A', 'B'); // 返回Array新的长度: 4
arr; // [1, 2, 'A', 'B']
arr.pop(); // pop()返回'B'
arr; // [1, 2, 'A']
arr.pop(); arr.pop(); arr.pop(); // 连续pop 3次
arr; // []
arr.pop(); // 空数组继续pop不会报错, 而是返回undefined
arr; // []
```

unshift和shift

如果要往Array的头部添加若干元素, 使用unshift()方法, shift()方法则把Array的第一个元素删掉:

```
var arr = [1, 2];
arr.unshift('A', 'B'); // 返回Array新的长度: 4
arr; // ['A', 'B', 1, 2]
arr.shift(); // 'A'
arr; // ['B', 1, 2]
arr.shift(); arr.shift(); arr.shift(); // 连续shift 3次
arr; // []
arr.shift(); // 空数组继续shift不会报错, 而是返回undefined
arr; // []
```

sort

sort()可以对当前Array进行排序, 它会直接修改当前Array的元素位置, 直接调用时, 按照默认排序:

```
var arr = ['B', 'C', 'A'];
arr.sort();
arr; // ['A', 'B', 'C']
```

能否按照我们自己指定的顺序排序呢? 完全可以, 我们将在后面的函数中讲到。

reverse

reverse()把整个Array的元素给掉个个, 也就是反转:

```
var arr = ['one', 'two', 'three'];
arr.reverse();
arr; // ['three', 'two', 'one']
```

splice

splice()方法是修改Array的“万能方法”, 它可以从指定的索引开始删除若干元素, 然后再从该位置添加若干元素:


```

var arr = ['Microsoft', 'Apple', 'Yahoo', 'AOL', 'Excite', 'Oracle'];
// 从索引2开始删除3个元素,然后再添加两个元素:
arr.splice(2, 3, 'Google', 'Facebook'); // 返回删除的元素
['Yahoo', 'AOL', 'Excite']
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
// 只删除,不添加:
arr.splice(2, 2); // ['Google', 'Facebook']
arr; // ['Microsoft', 'Apple', 'Oracle']
// 只添加,不删除:
arr.splice(2, 0, 'Google', 'Facebook'); // 返回[],因为没有删除任何元素
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']

```

concat

`concat()` 方法把当前的 `Array` 和另一个 `Array` 连接起来,并返回一个新的 `Array`:

```

var arr = ['A', 'B', 'C'];
var added = arr.concat([1, 2, 3]);
added; // ['A', 'B', 'C', 1, 2, 3]
arr; // ['A', 'B', 'C']

```

请注意, `concat()` 方法并没有修改当前 `Array`,而是返回了一个新的 `Array`。

实际上, `concat()` 方法可以接收任意个元素和 `Array`,并且自动把 `Array` 拆开,然后全部添加到新的 `Array` 里:

```

var arr = ['A', 'B', 'C'];
arr.concat(1, 2, [3, 4]); // ['A', 'B', 'C', 1, 2, 3, 4]

```

join

`join()` 方法是一个非常实用的方法,它把当前 `Array` 的每个元素都用指定的字符串连接起来,然后返回连接后的字符串:

```

var arr = ['A', 'B', 'C', 1, 2, 3];
arr.join('-'); // 'A-B-C-1-2-3'

```

如果 `Array` 的元素不是字符串,将自动转换为字符串后再连接。

多维数组

如果数组的某个元素又是一个 `Array`,则可以形成多维数组,例如:

```

var arr = [[1, 2, 3], [400, 500, 600], '-'];

```

上述 `Array` 包含3个元素，其中头两个元素本身也是 `Array`。

练习：如何通过索引取到 `500` 这个值：

```
'use strict';
var arr = [[1, 2, 3], [400, 500, 600], '-'];
var x = ??;
```

小结

`Array` 提供了一种顺序存储一组元素的功能，并可以按索引来读写。

练习：在新生欢迎会上，你已经拿到了新同学的名单，请排序后显示： 欢迎
XXX, XXX, XXX和XXX同学！：

```
'use strict';
var arr = ['小明', '小红', '大军', '阿黄'];
console.log('???');
```

对象

JavaScript的对象是一种无序的集合数据类型，它由若干键值对组成。

JavaScript的对象用于描述现实世界中的某个对象。例如，为了描述“小明”这个淘气的小朋友，我们可以用若干键值对来描述他：

```
var xiaoming = {
  name: '小明',
  birth: 1990,
  school: 'No.1 Middle School',
  height: 1.70,
  weight: 65,
  score: null
};
```

JavaScript用一个 `{...}` 表示一个对象，键值对以 `xxx: xxx` 形式申明，用 `,` 隔开。注意，最后一个键值对不需要在末尾加 `,`，如果加了，有的浏览器（如低版本的IE）将报错。

上述对象申明了一个 `name` 属性，值是 `'小明'`，`birth` 属性，值是 `1990`，以及其他一些属性。最后，把这个对象赋值给变量 `xiaoming` 后，就可以通过变量 `xiaoming` 来获取小明的属性了：

```
xiaoming.name; // '小明'
xiaoming.birth; // 1990
```

访问属性是通过 `.` 操作符完成的，但这要求属性名必须是一个有效的变量名。如果属性名包含特殊字符，就必须用 `''` 括起来：

```
var xiaohong = {  
  name: '小红',  
  'middle-school': 'No.1 Middle School'  
};
```

`xiaohong` 的属性名 `middle-school` 不是一个有效的变量，就需要用 `''` 括起来。访问这个属性也无法使用 `.` 操作符，必须用 `['xxx']` 来访问：

```
xiaohong['middle-school']; // 'No.1 Middle School'  
xiaohong['name']; // '小红'  
xiaohong.name; // '小红'
```

也可以用 `xiaohong['name']` 来访问 `xiaohong` 的 `name` 属性，不过 `xiaohong.name` 的写法更简洁。我们在编写JavaScript代码的时候，属性名尽量使用标准的变量名，这样就可以直接通过 `object.prop` 的形式访问一个属性了。

实际上JavaScript对象的所有属性都是字符串，不过属性对应的值可以是任意数据类型。

如果访问一个不存在的属性会返回什么呢？JavaScript规定，访问不存在的属性不报错，而是返回 `undefined`：

```
'use strict';  
  
var xiaoming = {  
  name: '小明'  
};  
console.log(xiaoming.name);  
console.log(xiaoming.age); // undefined
```

由于JavaScript的对象是动态类型，你可以自由地给一个对象添加或删除属性：

```
var xiaoming = {  
  name: '小明'  
};  
xiaoming.age; // undefined  
xiaoming.age = 18; // 新增一个age属性  
xiaoming.age; // 18  
delete xiaoming.age; // 删除age属性  
xiaoming.age; // undefined  
delete xiaoming['name']; // 删除name属性  
xiaoming.name; // undefined  
delete xiaoming.school; // 删除一个不存在的school属性也不会报错
```

如果我们要检测 `xiaoming` 是否拥有某一属性，可以用 `in` 操作符：

```
var xiaoming = {
  name: '小明',
  birth: 1990,
  school: 'No.1 Middle School',
  height: 1.70,
  weight: 65,
  score: null
};
'name' in xiaoming; // true
'grade' in xiaoming; // false
```

不过要小心，如果`in`判断一个属性存在，这个属性不一定是`xiaoming`的，它可能是`xiaoming`继承得到的：

```
'toString' in xiaoming; // true
```

因为`toString`定义在`object`对象中，而所有对象最终都会在原型链上指向`object`，所以`xiaoming`也拥有`toString`属性。

要判断一个属性是否是`xiaoming`自身拥有的，而不是继承得到的，可以用`hasOwnProperty()`方法：

```
var xiaoming = {
  name: '小明'
};
xiaoming.hasOwnProperty('name'); // true
xiaoming.hasOwnProperty('toString'); // false
```

条件判断

JavaScript使用`if () { ... } else { ... }`来进行条件判断。例如，根据年龄显示不同内容，可以用`if`语句实现如下：

```
var age = 20;
if (age >= 18) { // 如果age >= 18为true，则执行if语句块
  alert('adult');
} else { // 否则执行else语句块
  alert('teenager');
}
```

其中`else`语句是可选的。如果语句块只包含一条语句，那么可以省略`{ }`：

```
var age = 20;
if (age >= 18)
  alert('adult');
else
  alert('teenager');
```

省略 `{}` 的危险之处在于，如果后来想添加一些语句，却忘了写 `{}`，就改变了 `if...else...` 的语义，例如：

```
var age = 20;
if (age >= 18)
    alert('adult');
else
    console.log('age < 18'); // 添加一行日志
    alert('teenager'); // <- 这行语句已经不在else的控制范围了
```

上述代码的 `else` 子句实际上只负责执行 `console.log('age < 18');`，原有的 `alert('teenager');` 已经不属于 `if...else...` 的控制范围了，它每次都会执行。

相反地，有 `{}` 的语句就不会出错：

```
var age = 20;
if (age >= 18) {
    alert('adult');
} else {
    console.log('age < 18');
    alert('teenager');
}
```

这就是为什么我们建议永远都要写上 `{}`。

多行条件判断

如果还要更细致地判断条件，可以使用多个 `if...else...` 的组合：

```
var age = 3;
if (age >= 18) {
    alert('adult');
} else if (age >= 6) {
    alert('teenager');
} else {
    alert('kid');
}
```

上述多个 `if...else...` 的组合实际上相当于两层 `if...else...`：

```

var age = 3;
if (age >= 18) {
    alert('adult');
} else {
    if (age >= 6) {
        alert('teenager');
    } else {
        alert('kid');
    }
}

```

但是我们通常把 `else if` 连写在一起，来增加可读性。这里的 `else` 略掉了 `{}` 是没有问题的，因为它只包含一个 `if` 语句。注意最后一个单独的 `else` 不要略掉 `{}`。

请注意，`if...else...` 语句的执行特点是二选一，在多个 `if...else...` 语句中，如果某个条件成立，则后续就不再继续判断了。

试解释为什么下面的代码显示的是 `teenager`：

```

'use strict';
var age = 20;
if (age >= 6) {
    console.log('teenager');
} else if (age >= 18) {
    console.log('adult');
} else {
    console.log('kid');
}

```

由于 `age` 的值为 `20`，它实际上同时满足条件 `age >= 6` 和 `age >= 18`，这说明条件判断的顺序非常重要。请修复后让其显示 `adult`。

如果 `if` 的条件判断语句结果不是 `true` 或 `false` 怎么办？例如：

```

var s = '123';
if (s.length) { // 条件计算结果为3
    //
}

```

JavaScript 把 `null`、`undefined`、`0`、`NaN` 和空字符串 `''` 视为 `false`，其他值一概视为 `true`，因此上述代码条件判断的结果是 `true`。

练习

小明身高 `1.75`，体重 `80.5kg`。请根据 BMI 公式（体重除以身高的平方）帮小明计算他的 BMI 指数，并根据 BMI 指数：

- 低于 `18.5`：过轻

- 18.5-25: 正常
- 25-28: 过重
- 28-32: 肥胖
- 高于32: 严重肥胖

用 `if...else...` 判断并显示结果:

```
'use strict';

var height = parseFloat(prompt('请输入身高(m):'));
var weight = parseFloat(prompt('请输入体重(kg):'));
var bmi = ???;
if ...
```

循环

要计算 $1+2+3$ ，我们可以直接写表达式:

```
1 + 2 + 3; // 6
```

要计算 $1+2+3+...+10$ ，勉强也能写出来。

但是，要计算 $1+2+3+...+10000$ ，直接写表达式就不可能了。

为了让计算机能计算成千上万次的重复运算，我们就需要循环语句。

JavaScript的循环有两种，一种是 `for` 循环，通过初始条件、结束条件和递增条件来循环执行语句块:

```
var x = 0;
var i;
for (i=1; i<=10000; i++) {
    x = x + i;
}
x; // 50005000
```

让我们来分析一下 `for` 循环的控制条件:

- `i=1` 这是初始条件，将变量*i*置为1;
- `i<=10000` 这是判断条件，满足时就继续循环，不满足就退出循环;
- `i++` 这是每次循环后的递增条件，由于每次循环后变量*i*都会加1，因此它终将在若干次循环后不满足判断条件 `i<=10000` 而退出循环。

练习

利用 `for` 循环计算 `1 * 2 * 3 * ... * 10` 的结果:

```
'use strict';
var x = ?;
var i;
for ...
if (x === 3628800) {
    console.log('1 x 2 x 3 x ... x 10 = ' + x);
}
else {
    console.log('计算错误');
}
```

`for` 循环最常用的地方是利用索引来遍历数组：

```
var arr = ['Apple', 'Google', 'Microsoft'];
var i, x;
for (i=0; i<arr.length; i++) {
    x = arr[i];
    console.log(x);
}
```

`for` 循环的3个条件都是可以省略的，如果没有退出循环的判断条件，就必须使用 `break` 语句退出循环，否则就是死循环：

```
var x = 0;
for (;;) { // 将无限循环下去
    if (x > 100) {
        break; // 通过if判断来退出循环
    }
    x ++;
}
```

for ... in

`for` 循环的一个变体是 `for ... in` 循环，它可以把一个对象的所有属性依次循环出来：

```
var o = {
    name: 'Jack',
    age: 20,
    city: 'Beijing'
};
for (var key in o) {
    console.log(key); // 'name', 'age', 'city'
}
```

要过滤掉对象继承的属性，用 `hasOwnProperty()` 来实现：


```
var o = {
    name: 'Jack',
    age: 20,
    city: 'Beijing'
};
for (var key in o) {
    if (o.hasOwnProperty(key)) {
        console.log(key); // 'name', 'age', 'city'
    }
}
```

由于 `Array` 也是对象，而它的每个元素的索引被视为对象的属性，因此，`for ... in` 循环可以直接循环出 `Array` 的索引：

```
var a = ['A', 'B', 'C'];
for (var i in a) {
    console.log(i); // '0', '1', '2'
    console.log(a[i]); // 'A', 'B', 'C'
}
```

请注意，`for ... in` 对 `Array` 的循环得到的是 `String` 而不是 `Number`。

while

`for` 循环在已知循环的初始和结束条件时非常有用。而上述忽略了条件的 `for` 循环容易让人看不清循环的逻辑，此时用 `while` 循环更佳。

`while` 循环只有一个判断条件，条件满足，就不断循环，条件不满足时则退出循环。比如我们要计算100以内所有奇数之和，可以用 `while` 循环实现：

```
var x = 0;
var n = 99;
while (n > 0) {
    x = x + n;
    n = n - 2;
}
x; // 2500
```

在循环内部变量 `n` 不断自减，直到变为 `-1` 时，不再满足 `while` 条件，循环退出。

do ... while

最后一种循环是 `do { ... } while()` 循环，它和 `while` 循环的唯一区别在于，不是在每次循环开始的时候判断条件，而是在每次循环完成的时候判断条件：

```
var n = 0;
do {
    n = n + 1;
} while (n < 100);
n; // 100
```

用 `do { ... } while()` 循环要小心，循环体会至少执行1次，而 `for` 和 `while` 循环则可能一次都不执行。

练习

请利用循环遍历数组中的每个名字，并显示 `Hello, xxx!`：

```
'use strict';
var arr = ['Bart', 'Lisa', 'Adam'];
for ...
```

请尝试 `for` 循环和 `while` 循环，并以正序、倒序两种方式遍历。

小结

循环是让计算机做重复任务的有效的办法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。JavaScript的死循环会让浏览器无法正常显示或执行当前页面的逻辑，有的浏览器会直接挂掉，有的浏览器会在一段时间后提示你强行终止JavaScript的执行，因此，要特别注意死循环的问题。

在编写循环代码时，务必小心编写初始条件和判断条件，尤其是边界值。特别注意 `i < 100` 和 `i <= 100` 是不同的判断逻辑。

Map和Set

JavaScript的默认对象表示方式 `{}` 可以视为其他语言中的 `Map` 或 `Dictionary` 的数据结构，即一组键值对。

但是JavaScript的对象有个小问题，就是键必须是字符串。但实际上 `Number` 或者其他数据类型作为键也是非常合理的。

为了解决这个问题，最新的ES6规范引入了新的数据类型 `Map`。要测试你的浏览器是否支持ES6规范，请执行以下代码，如果浏览器报 `ReferenceError` 错误，那么你需要换一个支持ES6的浏览器：

```
'use strict';
var m = new Map();
var s = new Set();
console.log('你的浏览器支持Map和Set! ');
// 直接运行测试
```

Map

Map 是一组键值对的结构，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用 **Array** 实现，需要两个 **Array**：

```
var names = ['Michael', 'Bob', 'Tracy'];  
var scores = [95, 75, 85];
```

给定一个名字，要查找对应的成绩，就先要在 **names** 中找到对应的位置，再从 **scores** 取出对应的成绩，**Array** 越长，耗时越长。

如果用 **Map** 实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用 **JavaScript** 写一个 **Map** 如下：

```
var m = new Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);  
m.get('Michael'); // 95
```

初始化 **Map** 需要一个二维数组，或者直接初始化一个空 **Map**。**Map** 具有以下方法：

```
var m = new Map(); // 空Map  
m.set('Adam', 67); // 添加新的key-value  
m.set('Bob', 59);  
m.has('Adam'); // 是否存在key 'Adam': true  
m.get('Adam'); // 67  
m.delete('Adam'); // 删除key 'Adam'  
m.get('Adam'); // undefined
```

由于一个 **key** 只能对应一个 **value**，所以，多次对一个 **key** 放入 **value**，后面的值会把前面的值冲掉：

```
var m = new Map();  
m.set('Adam', 67);  
m.set('Adam', 88);  
m.get('Adam'); // 88
```

Set

Set 和 **Map** 类似，也是一组 **key** 的集合，但不存储 **value**。由于 **key** 不能重复，所以，在 **Set** 中，没有重复的 **key**。

要创建一个 **Set**，需要提供一个 **Array** 作为输入，或者直接创建一个空 **Set**：

```
var s1 = new Set(); // 空Set  
var s2 = new Set([1, 2, 3]); // 含1, 2, 3
```

重复元素在 **Set** 中自动被过滤：

```
var s = new Set([1, 2, 3, 3, '3']);
s; // Set {1, 2, 3, "3"}
```

注意数字3和字符串'3'是不同的元素。

通过`add(key)`方法可以添加元素到`Set`中，可以重复添加，但不会有效果：

```
s.add(4);
s; // Set {1, 2, 3, 4}
s.add(4);
s; // 仍然是 Set {1, 2, 3, 4}
```

通过`delete(key)`方法可以删除元素：

```
var s = new Set([1, 2, 3]);
s; // Set {1, 2, 3}
s.delete(3);
s; // Set {1, 2}
```

小结

`Map`和`Set`是ES6标准新增的数据类型，请根据浏览器的支持情况决定是否要使用。

iterable

遍历`Array`可以采用下标循环，遍历`Map`和`Set`就无法使用下标。为了统一集合类型，ES6标准引入了新的`iterable`类型，`Array`、`Map`和`Set`都属于`iterable`类型。

具有`iterable`类型的集合可以通过新的`for ... of`循环来遍历。

`for ... of`循环是ES6引入的新的语法，请测试你的浏览器是否支持：

```
'use strict';
var a = [1, 2, 3];
for (var x of a) {
}
console.log('你的浏览器支持for ... of');
// 请直接运行测试
```

用`for ... of`循环遍历集合，用法如下：

```

var a = ['A', 'B', 'C'];
var s = new Set(['A', 'B', 'C']);
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
for (var x of a) { // 遍历Array
    console.log(x);
}
for (var x of s) { // 遍历Set
    console.log(x);
}
for (var x of m) { // 遍历Map
    console.log(x[0] + '=' + x[1]);
}

```

你可能会有疑问，`for ... of`循环和`for ... in`循环有何区别？

`for ... in`循环由于历史遗留问题，它遍历的实际上是对象的属性名称。一个 `Array` 数组实际上也是一个对象，它的每个元素的索引被视为一个属性。

当我们手动给 `Array` 对象添加了额外的属性后，`for ... in` 循环将带来意想不到的意外效果：

```

var a = ['A', 'B', 'C'];
a.name = 'Hello';
for (var x in a) {
    console.log(x); // '0', '1', '2', 'name'
}

```

`for ... in` 循环将把 `name` 包括在内，但 `Array` 的 `length` 属性却不包括在内。

`for ... of` 循环则完全修复了这些问题，它只循环集合本身的元素：

```

var a = ['A', 'B', 'C'];
a.name = 'Hello';
for (var x of a) {
    console.log(x); // 'A', 'B', 'C'
}

```

这就是为什么要引入新的 `for ... of` 循环。

然而，更好的方式是直接使用 `iterable` 内置的 `forEach` 方法，它接收一个函数，每次迭代就自动回调该函数。以 `Array` 为例：

```

'use strict';
var a = ['A', 'B', 'C'];
a.forEach(function (element, index, array) {
    // element: 指向当前元素的值
    // index: 指向当前索引
    // array: 指向Array对象本身
    console.log(element + ', index = ' + index);
});

```

注意，`forEach()`方法是ES5.1标准引入的，你需要测试浏览器是否支持。

`Set`与`Array`类似，但`Set`没有索引，因此回调函数的前两个参数都是元素本身：

```
var s = new Set(['A', 'B', 'C']);
s.forEach(function (element, sameElement, set) {
    console.log(element);
});
```

`Map`的回调函数参数依次为`value`、`key`和`map`本身：

```
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
m.forEach(function (value, key, map) {
    console.log(value);
});
```

如果对某些参数不感兴趣，由于JavaScript的函数调用不要求参数必须一致，因此可以忽略它们。例如，只需要获得`Array`的`element`：

```
var a = ['A', 'B', 'C'];
a.forEach(function (element) {
    console.log(element);
});
```