JavaScript 的基本语法

语句

JavaScript 程序的执行单位为行（line），也就是一行一行地执行。一般情况下，每一行就是一个语句。

语句（statement）是为了完成某种任务而进行的操作，比如下面就是一行赋值语句。

```javascript
var a = 1 + 3;
```

这条语句先用`var`命令，声明了变量`a`，然后将`1 + 3`的运算结果赋值给变量`a`。

`1 + 3`叫做表达式（expression），指一个为了得到返回值的计算式。语句和表达式的区别在于，前者主要为了进行某种操作，一般情况下不需要返回值；后者则是为了得到返回值，一定会返回一个值。凡是 JavaScript 语言中预期为值的地方，都可以使用表达式。比如，赋值语句的等号右边，预期是一个值，因此可以放置各种表达式。

语句以分号结尾，一个分号就表示一个语句结束。多个语句可以写在一行内。

```javascript
var a = 1 + 3 ; var b = 'abc';
```

分号前面可以没有任何内容，JavaScript 引擎将其视为空语句。

```javascript
;;;
```

上面的代码就表示3个空语句。

表达式不需要分号结尾。一旦在表达式后面添加分号，则 JavaScript 引擎就将表达式视为语句，这样会产生一些没有任何意义的语句。

```javascript
1 + 3;
'abc';
```

上面两行语句只是单纯地产生一个值，并没有任何实际的意义。

变量

概念

变量是对“值”的具名引用。变量就是为“值”起名，然后引用这个名字，就等同于引用这个值。变量的名字就是变量名。

```javascript
var a = 1;
```

上面的代码先声明变量`a`，然后在变量`a`与数值1之间建立引用关系，称为将数值1“赋值”给变量`a`。以后，引用变量名`a`就会得到数值1。最前面的`var`，是变量声明命令。它表示通知解释引擎，要创建一个变量`a`。

注意，JavaScript 的变量名区分大小写，`A`和`a`是两个不同的变量。

变量的声明和赋值，是分开的两个步骤，上面的代码将它们合在了一起，实际的步骤是下面这样。

```javascript
var a;
a = 1;
```

如果只是声明变量而没有赋值，则该变量的值是`undefined`。`undefined`是一个特殊的值，表示“无定义”。

```javascript
var a;
a // undefined
```

如果变量赋值的时候，忘了写`var`命令，这条语句也是有效的。

```javascript
var a = 1;
// 基本等同
a = 1;
```

但是，不写`var`的做法，不利于表达意图，而且容易不知不觉地创建全局变量，所以建议总是使用`var`命令声明变量。

如果一个变量没有声明就直接使用，JavaScript 会报错，告诉你变量未定义。

```javascript
x
// ReferenceError: x is not defined
```

上面代码直接使用变量`x`，系统就报错，告诉你变量`x`没有声明。

可以在同一条`var`命令中声明多个变量。

```javascript
var a, b;
```

JavaScript 是一种动态类型语言，也就是说，变量的类型没有限制，变量可以随时更改类型。

```javascript
var a = 1;
a = 'hello';
```

上面代码中，变量`a`起先被赋值为一个数值，后来又被重新赋值为一个字符串。第二次赋值的时候，因为变量`a`已经存在，所以不需要使用`var`命令。

如果使用`var`重新声明一个已经存在的变量，是无效的。

```javascript
var x = 1;
var x;
x // 1
```

上面代码中，变量`x`声明了两次，第二次声明是无效的。

但是，如果第二次声明的时候还进行了赋值，则会覆盖掉前面的值。

```javascript
var x = 1;
var x = 2;

// 等同于

var x = 1;
var x;
x = 2;
```

变量提升

JavaScript 引擎的工作方式是，先解析代码，获取所有被声明的变量，然后再一行一行地运行。这造成的结果，就是所有的变量的声明语句，都会被提升到代码的头部，这就叫做变量提升（hoisting）。

```javascript
console.log(a);
```

```
var a = 1;
```

上面代码首先使用`console.log`方法，在控制台（console）显示变量`a`的值。这时变量`a`还没有声明和赋值，所以这是一种错误的做法，但是实际上不会报错。因为存在变量提升，真正运行的是下面的代码。

```javascript
var a;
console.log(a);
a = 1;
```

最后的结果是显示`undefined`，表示变量`a`已声明，但还未赋值。

标识符

标识符（identifier）指的是用来识别各种值的合法名称。最常见的标识符就是变量名，以及后面要提到的函数名。JavaScript 语言的标识符对大小写敏感，所以`a`和`A`是两个不同的标识符。

标识符有一套命名规则，不符合规则的就是非法标识符。JavaScript 引擎遇到非法标识符，就会报错。

简单说，标识符命名规则如下。

- 第一个字符，可以是任意 Unicode 字母（包括英文字母和其他语言的字母），以及美元符号（`$`）和下划线（`_`）。
- 第二个字符及后面的字符，除了 Unicode 字母、美元符号和下划线，还可以用数字`0-9`。

下面这些都是合法的标识符。

```javascript
arg0
_tmp
$elem
π
```

下面这些则是不合法的标识符。

```javascript
1a  // 第一个字符不能是数字
23  // 同上
***  // 标识符不能包含星号
a+b  // 标识符不能包含加号
-d  // 标识符不能包含减号或连词线
```

```

中文是合法的标识符，可以用作变量名。

```javascript
var 临时变量 = 1;
```

> JavaScript 有一些保留字，不能用作标识符：arguments、break、case、catch、class、const、continue、debugger、default、delete、do、else、enum、eval、export、extends、false、finally、for、function、if、implements、import、in、instanceof、interface、let、new、null、package、private、protected、public、return、static、super、switch、this、throw、true、try、typeof、var、void、while、with、yield。

注释

源码中被 JavaScript 引擎忽略的部分就叫做注释，它的作用是对代码进行解释。JavaScript 提供两种注释的写法：一种是单行注释，用`//`起头；另一种是多行注释，放在`/*`和`*/`之间。

```javascript
// 这是单行注释

/*
 这是
 多行
 注释
*/
```

此外，由于历史上 JavaScript 可以兼容 HTML 代码的注释，所以`<!--`和`-->`也被视为合法的单行注释。

```javascript
x = 1; <!-- x = 2;
--> x = 3;
```

上面代码中，只有`x = 1`会执行，其他的部分都被注释掉了。

需要注意的是，`-->`只有在行首，才会被当成单行注释，否则会当作正常的运算。

```javascript
function countdown(n) {
  while (n --> 0) console.log(n);
}
countdown(3)
// 2
```
```

```
// 1
// 0
```

上面代码中，`n --> 0`实际上会当作`n-- > 0`，因此输出2、1、0。

区块

JavaScript 使用大括号，将多个相关的语句组合在一起，称为“区块”（block）。

对于`var`命令来说，JavaScript 的区块不构成单独的作用域（scope）。

```javascript
{
  var a = 1;
}

a // 1
```

上面代码在区块内部，使用`var`命令声明并赋值了变量`a`，然后在区块外部，变量`a`依然有效，区块对于`var`命令不构成单独的作用域，与不使用区块的情况没有任何区别。在 JavaScript 语言中，单独使用区块并不常见，区块往往用来构成其他更复杂的语法结构，比如`for`、`if`、`while`、`function`等。

条件语句

JavaScript 提供`if`结构和`switch`结构，完成条件判断，即只有满足预设的条件，才会执行相应的语句。

if 结构

`if`结构先判断一个表达式的布尔值，然后根据布尔值的真伪，执行不同的语句。所谓布尔值，指的是 JavaScript 的两个特殊值，`true`表示真，`false`表示`伪`。

```javascript
if (布尔值)
  语句;

// 或者
if (布尔值) 语句;
```

上面是`if`结构的基本形式。需要注意的是，“布尔值”往往由一个条件表达式产生的，必须放在圆括号中，表示对表达式求值。如果表达式的求值结果为`true`，就执行紧跟在后面的语句；如果结果为`false`，则跳过紧跟在后面的语句。

```javascript
if (m === 3)
  m = m + 1;
```

上面代码表示，只有在`m`等于3时，才会将其值加上1。

这种写法要求条件表达式后面只能有一个语句。如果想执行多个语句，必须在`if`的条件判断之后，加上大括号，表示代码块（多个语句合并成一个语句）。

```javascript
if (m === 3) {
  m += 1;
}
```

建议总是在`if`语句中使用大括号，因为这样方便插入语句。

注意，`if`后面的表达式之中，不要混淆赋值表达式（`=`）、严格相等运算符（`===`）和相等运算符（`==`）。尤其是赋值表达式不具有比较作用。

```javascript
var x = 1;
var y = 2;
if (x = y) {
  console.log(x);
}
// "2"
```

上面代码的原意是，当`x`等于`y`的时候，才执行相关语句。但是，不小心将严格相等运算符写成赋值表达式，结果变成了将`y`赋值给变量`x`，再判断变量`x`的值（等于2）的布尔值（结果为`true`）。

这种错误可以正常生成一个布尔值，因而不会报错。为了避免这种情况，有些开发者习惯将常量写在运算符的左边，这样的话，一旦不小心将相等运算符写成赋值运算符，就会报错，因为常量不能被赋值。

```javascript
if (x = 2) { // 不报错
if (2 = x) { // 报错
```

至于为什么优先采用“严格相等运算符”（`===`），而不是“相等运算符”（`==`），请参考《运算符》章节。

if...else 结构

`if`代码块后面，还可以跟一个`else`代码块，表示不满足条件时，所要执行的代码。

```javascript
if (m === 3) {
  // 满足条件时，执行的语句
} else {
  // 不满足条件时，执行的语句
}
```

上面代码判断变量`m`是否等于3，如果等于就执行`if`代码块，否则执行`else`代码块。

对同一个变量进行多次判断时，多个`if...else`语句可以连写在一起。

```javascript
if (m === 0) {
  // ...
} else if (m === 1) {
  // ...
} else if (m === 2) {
  // ...
} else {
  // ...
}
```

`else`代码块总是与离自己最近的那个`if`语句配对。

```javascript
var m = 1;
var n = 2;

if (m !== 1)
if (n === 2) console.log('hello');
else console.log('world');
```

上面代码不会有任何输出，`else`代码块不会得到执行，因为它跟着的是最近的那个`if`语句，相当于下面这样。

```javascript
if (m !== 1) {
  if (n === 2) {
    console.log('hello');
  } else {
    console.log('world');
  }
}
```

如果想让`else`代码块跟随最上面的那个`if`语句，就要改变大括号的位置。

```javascript
if (m !== 1) {
  if (n === 2) {
    console.log('hello');
  }
} else {
  console.log('world');
}
// world
```

switch 结构

多个`if...else`连在一起使用的时候，可以转为使用更方便的`switch`结构。

```javascript
switch (fruit) {
  case "banana":
    // ...
    break;
  case "apple":
    // ...
    break;
  default:
    // ...
}
```

上面代码根据变量`fruit`的值，选择执行相应的`case`。如果所有`case`都不符合，则执行最后的`default`部分。需要注意的是，每个`case`代码块内部的`break`语句不能少，否则会接下去执行下一个`case`代码块，而不是跳出`switch`结构。

```javascript
var x = 1;

switch (x) {
  case 1:
    console.log('x 等于1');
  case 2:
    console.log('x 等于2');
  default:
    console.log('x 等于其他值');
}
// x等于1
// x等于2
// x等于其他值
```

上面代码中，`case`代码块之中没有`break`语句，导致不会跳出`switch`结构，而会一直执行下去。正确的写法是像下面这样。

```javascript
switch (x) {
  case 1:
    console.log('x 等于1');
    break;
  case 2:
    console.log('x 等于2');
    break;
  default:
    console.log('x 等于其他值');
}
```

`switch`语句部分和`case`语句部分，都可以使用表达式。

```javascript
switch (1 + 3) {
  case 2 + 2:
    f();
    break;
  default:
    neverHappens();
}
```

上面代码的`default`部分，是永远不会执行到的。

需要注意的是，`switch`语句后面的表达式，与`case`语句后面的表示式比较运行结果时，采用的是严格相等运算符（`===`），而不是相等运算符（`==`），这意味着比较时不会发生类型转换。

```javascript
var x = 1;

switch (x) {
  case true:
    console.log('x 发生类型转换');
    break;
  default:
    console.log('x 没有发生类型转换');
}
// x 没有发生类型转换
```

上面代码中，由于变量`x`没有发生类型转换，所以不会执行`case true`的情况。这表明，`switch`语句内部采用的是“严格相等运算符”，详细解释请参考《运算符》一节。

三元运算符 ?:

JavaScript 还有一个三元运算符（即该运算符需要三个运算子）`?:`，也可以用于逻辑判断。

```javascript
(条件) ? 表达式1 : 表达式2
```

上面代码中，如果“条件”为`true`，则返回“表达式1”的值，否则返回“表达式2”的值。

```javascript
var even = (n % 2 === 0) ? true : false;
```

上面代码中，如果`n`可以被2整除，则`even`等于`true`，否则等于`false`。它等同于下面的形式。

```javascript
var even;
if (n % 2 === 0) {
  even = true;
} else {
  even = false;
}
```

这个三元运算符可以被视为`if...else...`的简写形式，因此可以用于多种场合。

```javascript
var myVar;
console.log(
  myVar ?
  'myVar has a value' :
  'myVar does not have a value'
)
// myVar does not have a value
```

上面代码利用三元运算符，输出相应的提示。

```javascript
var msg = '数字' + n + '是' + (n % 2 === 0 ? '偶数' : '奇数');
```

上面代码利用三元运算符，在字符串之中插入不同的值。

循环语句

循环语句用于重复执行某个操作，它有多种形式。

while 循环

`While`语句包括一个循环条件和一段代码块，只要条件为真，就不断循环执行代码块。

```javascript
while (条件)
  语句;

// 或者
while (条件) 语句;
```

`while`语句的循环条件是一个表达式，必须放在圆括号中。代码块部分，如果只有一条语句，可以省略大括号，否则就必须加上大括号。

```javascript
while (条件) {
  语句;
}
```

下面是`while`语句的一个例子。

```javascript
var i = 0;

while (i < 100) {
  console.log('i 当前为：' + i);
  i = i + 1;
}
```

上面的代码将循环100次，直到`i`等于100为止。

下面的例子是一个无限循环，因为循环条件总是为真。

```javascript
while (true) {
  console.log('Hello, world');
}
```

for 循环

`for`语句是循环命令的另一种形式，可以指定循环的起点、终点和终止条件。它的格式如下。

```javascript
```

```
for (初始化表达式; 条件; 递增表达式)
  语句

// 或者

for (初始化表达式; 条件; 递增表达式) {
  语句
}
```

`for`语句后面的括号里面，有三个表达式。

- 初始化表达式（initialize）：确定循环变量的初始值，只在循环开始时执行一次。
- 条件表达式（test）：每轮循环开始时，都要执行这个条件表达式，只有值为真，才继续进行循环。
- 递增表达式（increment）：每轮循环的最后一个操作，通常用来递增循环变量。

下面是一个例子。

```javascript
var x = 3;
for (var i = 0; i < x; i++) {
  console.log(i);
}
// 0
// 1
// 2
```

上面代码中，初始化表达式是`var i = 0`，即初始化一个变量`i`；测试表达式是`i < x`，即只要`i`小于`x`，就会执行循环；递增表达式是`i++`，即每次循环结束后，`i`增大1。

所有`for`循环，都可以改写成`while`循环。上面的例子改为`while`循环，代码如下。

```javascript
var x = 3;
var i = 0;

while (i < x) {
  console.log(i);
  i++;
}
```

`for`语句的三个部分（initialize、test、increment），可以省略任何一个，也可以全部省略。

```javascript
for ( ; ; ){
```

```
  console.log('Hello World');
}
```

上面代码省略了`for`语句表达式的三个部分，结果就导致了一个无限循环。

do...while 循环

`do...while`循环与`while`循环类似，唯一的区别就是先运行一次循环体，然后判断循环条件。

```javascript
do
  语句
while (条件);

// 或者
do {
  语句
} while (条件);
```

不管条件是否为真，`do...while`循环至少运行一次，这是这种结构最大的特点。另外，`while`语句后面的分号注意不要省略。

下面是一个例子。

```javascript
var x = 3;
var i = 0;

do {
  console.log(i);
  i++;
} while(i < x);
```

break 语句和 continue 语句

`break`语句和`continue`语句都具有跳转作用，可以让代码不按既有的顺序执行。

`break`语句用于跳出代码块或循环。

```javascript
var i = 0;

while(i < 100) {
  console.log('i 当前为：' + i);
  i++;
```

```
  if (i === 10) break;
}
```

上面代码只会执行10次循环，一旦`i`等于10，就会跳出循环。

`for`循环也可以使用`break`语句跳出循环。

```javascript
for (var i = 0; i < 5; i++) {
  console.log(i);
  if (i === 3)
    break;
}
// 0
// 1
// 2
// 3
```

上面代码执行到`i`等于3，就会跳出循环。

`continue`语句用于立即终止本轮循环，返回循环结构的头部，开始下一轮循环。

```javascript
var i = 0;

while (i < 100){
  i++;
  if (i % 2 === 0) continue;
  console.log('i 当前为：' + i);
}
```

上面代码只有在`i`为奇数时，才会输出`i`的值。如果`i`为偶数，则直接进入下一轮循环。

如果存在多重循环，不带参数的`break`语句和`continue`语句都只针对最内层循环。

标签（label）

JavaScript 语言允许，语句的前面有标签（label），相当于定位符，用于跳转到程序的任意位置，标签的格式如下。

```javascript
label:
  语句
```

标签可以是任意的标识符，但不能是保留字，语句部分可以是任意语句。

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

```javascript
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
```

上面代码为一个双重循环区块，`break`命令后面加上了`top`标签（注意，`top`不用加引号），满足条件时，直接跳出双层循环。如果`break`语句后面不使用标签，则只能跳出内层循环，进入下一次的外层循环。

标签也可以用于跳出代码块。

```javascript
foo: {
  console.log(1);
  break foo;
  console.log('本行不会输出');
}
console.log(2);
// 1
// 2
```

上面代码执行到`break foo`，就会跳出区块。

`continue`语句也可以与标签配合使用。

```javascript
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
```

上面代码中，`continue`命令后面有一个标签名，满足条件时，会跳过当前循环，直接进入下一轮外层循环。如果`continue`语句后面不使用标签，则只能进入下一轮的内层循环。

参考链接

- Axel Rauschmayer, [A quick overview of JavaScript](http://www.2ality.com/2011/10/javascript-overview.html)