编程风格

目录

- 概述
- 缩进
- 区块
- 圆括号
- 行尾的分号
  - 不使用分号的情况
  - 分号的自动添加
- 全局变量
- 变量声明
- with 语句
- 相等和严格相等
- 语句的合并
- 自增和自减运算符
- switch...case 结构
- 参考链接

概述

“编程风格”（programming style）指的是编写代码的样式规则。不同的程序员，往往有不同的编程风格。

有人说，编译器的规范叫做“语法规则”（grammar），这是程序员必须遵守的；而编译器忽略的部分，就叫“编程风格”（programming style），这是程序员可以自由选择的。这种说法不完全正确，程序员固然可以自由选择编程风格，但是好的编程风格有助于写出质量更高、错误更少、更易于维护的程序。

所以，编程风格的选择不应该基于个人爱好、熟悉程度、打字量等因素，而要考虑如何尽量使代码清晰易读、减少出错。你选择的，不是你喜欢的风格，而是一种能够清晰表达你的意图的风格。这一点，对于JavaScript这种语法自由度很高的语言尤其重要。

必须牢记的一点是，如果你选定了一种“编程风格”，就应该坚持遵守，切忌多种风格混用。如果你加入他人的项目，就应该遵守现有的风格。

缩进

行首的空格和 Tab 键，都可以产生代码缩进效果（indent）。

Tab 键可以节省击键次数，但不同的文本编辑器对 Tab 的显示不尽相同，有的显示四个空格，有的显示两个空格，所以有人觉得，空格键可以使得显示效果更统一。

无论你选择哪一种方法，都是可以接受的，要做的就是始终坚持这一种选择。不要一会使用 Tab 键，一会使用空格键。

区块

如果循环和判断的代码体只有一行，JavaScript 允许该区块（block）省略大括号。

```
if (a)
  b();
  c();
```

上面代码的原意可能是下面这样。

```
if (a) {
  b();
  c();
}
```

但是，实际效果却是下面这样。

```
if (a) {
  b();
}
  c();
```

因此，建议总是使用大括号表示区块。

另外，区块起首的大括号的位置，有许多不同的写法。最流行的有两种，一种是起首的大括号另起一行。

```
block
{
  // ...
}
```

另一种是起首的大括号跟在关键字的后面。

```
block {
  // ...
}
```

一般来说，这两种写法都可以接受。但是，JavaScript 要使用后一种，因为 JavaScript 会自动添加句末的分号，导致一些难以察觉的错误。

```
return
{
  key: value
};

// 相当于
return;
{
  key: value
};
```

上面的代码的原意，是要返回一个对象，但实际上返回的是 `undefined`，因为 JavaScript 自动在 `return` 语句后面添加了分号。为了避免这一类错误，需要写成下面这样。

```
return {
  key : value
};
```

因此，表示区块起首的大括号，不要另起一行。

圆括号

圆括号（parentheses）在 JavaScript 中有两种作用，一种表示函数的调用，另一种表示表达式的组合（grouping）。

```
// 圆括号表示函数的调用
console.log('abc');

// 圆括号表示表达式的组合
(1 + 2) * 3
```

建议可以用空格，区分这两种不同的括号。

> 1. 表示函数调用时，函数名与左括号之间没有空格。
> 2. 表示函数定义时，函数名与左括号之间没有空格。
> 3. 其他情况时，前面位置的语法元素与左括号之间，都有一个空格。

按照上面的规则，下面的写法都是不规范的。

```
foo (bar)
return(a+b);
if(a === 0) {...}
function foo (b) {...}
function(x) {...}
```

上面代码的最后一行是一个匿名函数，`function`是语法关键字，不是函数名，所以与左括号之间应该要有一个空格。

行尾的分号

分号表示一条语句的结束。JavaScript允许省略行尾的分号。事实上，确实有一些开发者行尾从来不写分号。但是，由于下面要讨论的原因，建议还是不要省略这个分号。

不使用分号的情况

首先，以下三种情况，语法规定本来就不需要在结尾添加分号。

（1）for 和 while 循环

```
for ( ; ; ) {
} // 没有分号
```

```
while (true) {
} // 没有分号
```

注意，`do...while`循环是有分号的。

```
do {
  a--;
} while(a > 0); // 分号不能省略
```

（**2**）分支语句：**if**，**switch**，**try**

```
if (true) {
} // 没有分号

switch () {
} // 没有分号

try {
} catch {
} // 没有分号
```

（**3**）函数的声明语句

```
function f() {
} // 没有分号
```

注意，函数表达式仍然要使用分号。

```
var f = function f() {
};
```

以上三种情况，如果使用了分号，并不会出错。因为，解释引擎会把这个分号解释为空语句。

分号的自动添加

除了上一节的三种情况，所有语句都应该使用分号。但是，如果没有使用分号，大多数情况下，JavaScript会自动添加。

```
var a = 1
// 等同于
var a = 1;
```

这种语法特性被称为 “分号的自动添加”（Automatic Semicolon Insertion，简称 ASI）。

因此，有人提倡省略句尾的分号。麻烦的是，如果下一行的开始可以与本行的结尾连在一起解释，JavaScript 就不会自动添加分号。

```
// 等同于 var a = 3
var
a
=
3

// 等同于 'abc'.length
'abc'
.length

// 等同于 return a + b;
return a +
b;

// 等同于 obj.foo(arg1, arg2);
obj.foo(arg1,
arg2);

// 等同于 3 * 2 + 10 * (27 / 6)
3 * 2
+
10 * (27 / 6)
```

上面代码都会多行放在一起解释，不会每一行自动添加分号。这些例子还是比较容易看出来的，但是下面这个例子就不那么容易看出来了。

```
x = y
(function () {
  // ...
})();

// 等同于
x = y(function () {...})();
```

下面是更多不会自动添加分号的例子。

```
// 引擎解释为 c(d+e)
var a = b + c
(d+e).toString();

// 引擎解释为 a = b/hi/g.exec(c).map(d)
// 正则表达式的斜杠，会当作除法运算符
a = b
/hi/g.exec(c).map(d);

// 解释为'b'['red', 'green']，
// 即把字符串当作一个数组，按索引取值
var a = 'b'
['red', 'green'].forEach(function (c) {
  console.log(c);
})

// 解释为 function (x) { return x }(a++)
// 即调用匿名函数，结果f等于0
var a = 0;
var f = function (x) { return x }
(a++)
```

只有下一行的开始与本行的结尾，无法放在一起解释，JavaScript 引擎才会自动添加分号。

```
if (a < 0) a = 0
console.log(a)

// 等同于下面的代码，
// 因为 0console 没有意义
if (a < 0) a = 0;
console.log(a)
```

另外，如果一行的起首是“自增”（`++`）或“自减”（`--`）运算符，则它们的前面会自动添加分号。

```
a = b = c = 1

a
++
b
--
```

```
c

console.log(a, b, c)
// 1 2 0
```

上面代码之所以会得到 `1 2 0`
的结果，原因是自增和自减运算符前，自动加上了分号。上面的代码实际上等同于下面的形式。

```
a = b = c = 1;
a;
++b;
--c;
```

如果 `continue`、`break`、`return`和 `throw`
这四个语句后面，直接跟换行符，则会自动添加分号。这意味着，如果 `return`
语句返回的是一个对象的字面量，起首的大括号一定要写在同一行，否则得不到预期结果。

```
return
{ first: 'Jane' };

// 解释成
return;
{ first: 'Jane' };
```

由于解释引擎自动添加分号的行为难以预测，因此编写代码的时候不应该省略行尾的分号。

不应该省略结尾的分号，还有一个原因。有些 JavaScript 代码压缩器（ uglifier
）不会自动添加分号，因此遇到没有分号的结尾，就会让代码保持原状，而不是压缩成一行，使得压缩无法得
到最优的结果。

另外，不写结尾的分号，可能会导致脚本合并出错。所以，有的代码库在第一行语句开始前，会加上一个分号
。

```
;var a = 1;
// ...
```

上面这种写法就可以避免与其他脚本合并时，排在前面的脚本最后一行语句没有分号，导致运行出错的问题。

全局变量

JavaScript

最大的语法缺点，可能就是全局变量对于任何一个代码块，都是可读可写。这对代码的模块化和重复使用，非常不利。

因此，建议避免使用全局变量。如果不得不使用，可以考虑用大写字母表示变量名，这样更容易看出这是全局变量，比如 `UPPER_CASE`。

变量声明

JavaScript 会自动将变量声明“提升”（hoist）到代码块（block）的头部。

```
if (!x) {
  var x = {};
}

// 等同于
var x;
if (!x) {
  x = {};
}
```

这意味着，变量 `x` 是 `if` 代码块之前就存在了。为了避免可能出现的问题，最好把变量声明都放在代码块的头部。

```
for (var i = 0; i < 10; i++) {
  // ...
}

// 写成
var i;
for (i = 0; i < 10; i++) {
  // ...
}
```

上面这样的写法，就容易看出存在一个全局的循环变量 `i`。

另外，所有函数都应该在使用之前定义。函数内部的变量声明，都应该放在函数的头部。

with 语句

`with`可以减少代码的书写，但是会造成混淆。

```
with (o) {
  foo = bar;
}
```

上面的代码，可以有四种运行结果：

```
o.foo = bar;
o.foo = o.bar;
foo = bar;
foo = o.bar;
```

这四种结果都可能发生，取决于不同的变量是否有定义。因此，不要使用`with`语句。

相等和严格相等

JavaScript 有两个表示相等的运算符：“相等”（`==`）和“严格相等”（`===`）。

相等运算符会自动转换变量类型，造成很多意想不到的情况。

```
0 == ''// true
1 == true // true
2 == true // false
0 == '0' // true
false == 'false' // false
false == '0' // true
' \t\r\n ' == 0 // true
```

因此，建议不要使用相等运算符（`==`），只使用严格相等运算符（`===`）。

语句的合并

有些程序员追求简洁，喜欢合并不同目的的语句。比如，原来的语句是

```
a = b;
if (a) {
  // ...
```

```
}
```

他喜欢写成下面这样。

```
if (a = b) {
  // ...
}
```

虽然语句少了一行，但是可读性大打折扣，而且会造成误读，让别人误解这行代码的意思是下面这样。

```
if （a === b）{
  // ...
}
```

建议不要将不同目的的语句，合并成一行。

自增和自减运算符

自增（`++`）和自减（`--`）运算符，放在变量的前面或后面，返回的值不一样，很容易发生错误。事实上，所有的`++`运算符都可以用`+= 1`代替。

```
++x
// 等同于
x += 1;
```

改用`+= 1`，代码变得更清晰了。

建议自增（`++`）和自减（`--`）运算符尽量使用`+=`和`-=`代替。

switch...case 结构

`switch...case`结构要求，在每一个`case`的最后一行必须是`break`语句，否则会接着运行下一个`case`。这样不仅容易忘记，还会造成代码的冗长。

而且，`switch...case`不使用大括号，不利于代码形式的统一。此外，这种结构类似于`goto`语句，容易造成程序流程的混乱，使得代码结构混乱不堪，不符合面向对象编程的原则。

```
function doAction(action) {
  switch (action) {
    case 'hack':
      return 'hack';
    case 'slash':
      return 'slash';
    case 'run':
      return 'run';
    default:
      throw new Error('Invalid action.');
  }
}
```

□□□□□□□□□□□□□□□

```
function doAction(action) {
  var actions = {
    'hack': function () {
      return 'hack';
    },
    'slash': function () {
      return 'slash';
    },
    'run': function () {
      return 'run';
    }
  };

  if (typeof actions[action] !== 'function') {
    throw new Error('Invalid action.');
  }

  return actions[action]();
}
```

□□□□□ `switch...case` □□□□□□□□□□□□□□

□□□□

- Eric Elliott, Programming JavaScript Applications, Chapter 2. JavaScript Style Guide, O'Reilly, 2014
- Axel Rauschmayer, A meta style guide for JavaScript
- Axel Rauschmayer, Automatic semicolon insertion in JavaScript
- Rod Vagg, JavaScript and Semicolons

□□□ #2
□ □□□ □□□ 2018-01-14 19:39:35 UTC
□ □□□ □□□ 2022-03-01 13:14:29 UTC