JavaScript规范

全局命名空间的污染与IIFE（函数表达式）

总是将代码包裹成一个 IIFE(Immediately-Invoked Function Expression)，用以创建独立隔绝的定义域。这一举措可防止全局命名空间被污染。
IIFE 还可确保你的代码不会轻易被其它全局命名空间里的代码所修改（i.e. 第三方库，window 引用，被覆盖的未定义的关键字等等）。

不推荐

```
var x = 20,
y = 30;
console.log(x+y);
```

推荐

```
(function(log,w){
      var x = 20,
      y = 30;
      log(x+y);
}(window.console.log,window))
```

IIFE(立即执行的函数代码)

无论何时，想要创建一个新的封闭的定义域，那就用 IIFE。它不仅避免了干扰，也使得内存在执行完后立即释放。
所有脚本文件建议都从 IIFE 开始。
立即执行的函数表达式的执行括号应该写在外包括号内。虽然写在内还是写在外都是有效的，但写在内使得整个表达式看起来更像一个整体，因此推荐这么做。

不推荐

```
(function(){})()
```

推荐

```
(function(){}())
```

所以，用下列写法格式化你的IIFE代码

```
(function(){
    //Code goes here
}())
```

如果你想引用全局变量或者外部IIFE的变量，可以通过下列方式传参

```
(function($,w,d){
     $(function(){
            w.alert(d.querySelectorAll("div").length);
     })
}(jquery,window,document))
```

变量声明

总是使用 `var` 来声明变量。如不指定 var，变量将被隐式地声明为全局变量，这将对变量难以控制。如果没有声明，变量处于什么定义域就变得不清（可以是在 Document 或 Window 中，也可以很容易地进入本地定义域）。所以，请总是使用 var 来声明变量。
采用严格模式带来的好处是，当你手误输入错误的变量名时，它可以通过报错信息来帮助你定位错误出处。

不推荐

```
x = 10;
y = 20;
```

推荐

```
var x = 10,
y = 20;
```

提升声明

为避免上一章节所述的变量和方法定义被自动提升造成误解，把风险降到最低，我们应该手动地显示地去声明变量与方法。也就是说，所有的变量以及方法，应当定义在 function 内的首行。
只用一个 `var` 关键字声明，多个变量用逗号隔开。

不推荐

```
(function(log){
```

```
    "use strict";
    var a = 10;
    var b = 10;
    for (var i = 0; i<a;i++){
        var  c = a * b *i;
   }
  function f(){
  }
  var d = 100;
  var x = function(){
       return d * d;
  }
  log(x());
}(widnow.console.log));
```

推荐

```
(function(log){
    var a = 10,
    b=10,
    i,
    c,
    d,
    x;
    function f(){
    }
    for(i = 0;i<>a;i++){
        c = a * b * i;
  }
  d = 100;
  x = function(){
       return d * d;
  }
  log(x());
}(window.consloe.log));
```

把赋值尽量写在变量申明中。

不推荐

```
var a,
b,
c;
a = 10;
b = 20;
c = 30;
```

推荐

```
var a =10,
b = 20,
c = 30;
```

总是使用带类型判断的比较判断

总是使用 `===` 精确的比较操作符，避免在判断的过程中，由 JavaScript 的强制类型转换所造成的困扰。
如果你使用 `===` 操作符，那比较的双方必须是同一类型为前提的条件下才会有效。
在只使用 `==` 的情况下，JavaScript 所带来的强制类型转换使得判断结果跟踪变得复杂。
下面的例子可以看出这样的结果有多怪了：

```
(function(log){
      "use strict";
      log("0"==0);//true
      log(""==false);//true
      log("1"==true);//true
      log("null"==undefined);//true
      var x = {
           valueOf:function(){
                return "X";
           }
      };
      log(x == "X")
```

```
}(window.console.log));
```

明智地使用真假判断

当我们在一个 if 条件语句中使用变量或表达式时，会做真假判断。`if(a == true)` 是不同于 `if(a)` 的。后者的判断比较特殊，我们称其为真假判断。这种判断会通过特殊的操作将其转换为 true 或 false，下列表达式统统返回 false：`false`, `0`, `undefined`, `null`, `NaN`, `''`（空字符串）.
这种真假判断在我们只求结果而不关心过程的情况下，非常的有帮助。
以下示例展示了真假判断是如何工作的：

```
function(log){
     'use strict';
      function logTruthyFalsy(expr) {
            if(expr) {
                  log('truthy');
            } else {
                  log('falsy');
           }
      }

     logTruthyFalsy(true); // truthy
     logTruthyFalsy(1); // truthy
     logTruthyFalsy({}); // truthy
     logTruthyFalsy([]); // truthy
     logTruthyFalsy('0'); // truthy

     logTruthyFalsy(false); // falsy
     logTruthyFalsy(0); // falsy
     logTruthyFalsy(undefined); // falsy
     logTruthyFalsy(null); // falsy
     logTruthyFalsy(NaN); // falsy
     logTruthyFalsy(''); // falsy

}(window.console.log));

```

变量赋值时的逻辑操作

逻辑操作符 `||` 和 `&&` 也可被用来返回布尔值。如果操作对象为非布尔对象，那每个表达式将会被自左向右地做真假判断。
基于此操作，最终总有一个表达式被返回回来。这在变量赋值时，是可以用来简化你的代码的。

不推荐

```
if(!x) {
    if(!y) {
          x = 1;
     } else {
          x = y;
     }
}
```

推荐

```
x = x || y || 1;
```

这一小技巧经常用来给方法设定默认的参数。

```
(function(log){
      'use strict';
       function multiply(a, b) {
              a = a || 1;
              b = b || 1;
              log('Result ' + a * b);
      }
      multiply(); // Result 1
      multiply(10); // Result 10
      multiply(3, NaN); // Result 3
      multiply(9, 5); // Result 45
}(window.console.log));
```

分号

总是使用分号，因为隐式的代码嵌套会引发难以察觉的问题。当然我们更要从根本上来杜绝这些问题

Why?

JavaScript 中语句要以分号结束，否则它将会继续执行下去，不管换不换行。

以上的每一个示例中，函数声明或对象或数组，都变成了在一句语句体内。

要知道闭合圆括号并不代表语句结束，JavaScript 不会终结语句，除非它的下一个 token 是一个中缀符或者是圆括号操作符。

澄清：分号与函数

分号需要用在表达式的结尾，而并非函数声明的结尾。区分它们最好的例子是：

```
var foo = function() {
       return true;
}; // semicolon here.

function foo() {
    return true;
} // no semicolon here.

```

嵌套函数

嵌套函数是非常有用的，比如用在持续创建和隐藏辅助函数的任务中。你可以非常自由随意地使用它们。

语句块内的函数声明

切勿在语句块内声明函数，在 ECMAScript 5 的严格模式下，这是不合法的。函数声明应该在定义域的顶层。

但在语句块内可将函数申明转化为函数表达式赋值给变量。

不推荐

```
if (x) {
       function foo() {}
```

```
}
```

推荐

```
if (x) {
      var foo = function() {};
}
```

使用闭包

闭包的创建也许是 JS 最有用也是最易被忽略的能力了。

this 关键字

只在对象构造器、方法和在设定的闭包中使用 `this` 关键字。this 的语义在此有些误导。它时而指向全局对象（大多数时），时而指向调用者的定义域（在 eval 中），时而指向 DOM 树中的某一节点（当用事件处理绑定到 HTML 属性上时），时而指向一个新创建的对象（在构造器中），还时而指向其它的一些对象（如果函数被 `call()` 和 `apply()` 执行和调用时）。

正因为它是如此容易地被搞错，请限制它的使用场景：

- 在构造函数中
- 在对象的方法中（包括由此创建出的闭包内）

字符串

统一使用单引号(‘)，不使用双引号(“)。这在创建 HTML 字符串非常有好处：

```
var msg = 'This is some HTML <div class="makes-sense"></div>';
```

三元条件判断（if 的快捷方法）

用三元操作符分配或返回语句。在比较简单的情况下使用，避免在复杂的情况下使用。没人愿意用 10 行三元操作符把自己的脑子绕晕。

不推荐

```
if(x === 10) {
      return 'valid';
} else {
       return 'invalid';
}
推荐

return x === 10 ? 'valid' : 'invalid';
```