[**JavaScript规范**](http://cdcf.retail-tek.com/pages/viewpage.action?pageId=4883143)

**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**

1. var x = 20,
2. y = 30;
3. console.log(x+y);

**推荐**

1. (function(log,w){
2. var x = 20,
3. y = 30;
4. log(x+y);
5. }(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**

1. (function(){})()

**推荐**

1. (function(){}())

所以，用下列写法格式化你的IIFE代码

1. (function(){
2. //Code goes here
3. }())

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

1. (function($,w,d){
2. $(function(){
3. w.alert(d.querySelectorAll("div").length);
4. })
5. }(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**

1. x = 10;
2. y = 20;

**推荐**

1. var x = 10,
2. 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**

1. (function(log){
2. "use strict";
3. var a = 10;
4. var b = 10;
5. for (var i = 0; i<a;i++){
6. var c = a \* b \*i;
7. }
8. function f(){
9. }
10. var d = 100;
11. var x = function(){
12. return d \* d;
13. }
14. log(x());
15. }(widnow.console.log));

**推荐**

1. (function(log){
2. var a = 10,
3. b=10,
4. i,
5. c,
6. d,
7. x;
8. function f(){
9. }
10. for(i = 0;i<>a;i++){
11. c = a \* b \* i;
12. }
13. d = 100;
14. x = function(){
15. return d \* d;
16. }
17. log(x());
18. }(window.consloe.log));

把赋值尽量写在变量申明中。

**不推荐**

1. var a,
2. b,
3. c;
4. a = 10;
5. b = 20;
6. c = 30;

**推荐**

1. var a =10,
2. b = 20,
3. 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：

1. (function(log){
2. "use strict";
3. log("0"==0);//true
4. log(""==false);//true
5. log("1"==true);//true
6. log("null"==undefined);//true
7. var x = {
8. valueOf:function(){
9. return "X";
10. }
11. };
12. log(x == "X")
13. }(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：

1. function(log){
2. 'use strict';
3. function logTruthyFalsy(expr) {
4. if(expr) {
5. log('truthy');
6. } else {
7. log('falsy');
8. }
9. }
11. logTruthyFalsy(true); // truthy
12. logTruthyFalsy(1); // truthy
13. logTruthyFalsy({}); // truthy
14. logTruthyFalsy([]); // truthy
15. logTruthyFalsy('0'); // truthy
17. logTruthyFalsy(false); // falsy
18. logTruthyFalsy(0); // falsy
19. logTruthyFalsy(undefined); // falsy
20. logTruthyFalsy(null); // falsy
21. logTruthyFalsy(NaN); // falsy
22. logTruthyFalsy(''); // falsy
24. }(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**

1. if(!x) {
2. if(!y) {
3. x = 1;
4. } else {
5. x = y;
6. }
7. }

**推荐**

1. x = x || y || 1;

这一小技巧经常用来给方法设定默认的参数。

1. (function(log){
2. 'use strict';
3. function multiply(a, b) {
4. a = a || 1;
5. b = b || 1;
6. log('Result ' + a \* b);
7. }
8. multiply(); // Result 1
9. multiply(10); // Result 10
10. multiply(3, NaN); // Result 3
11. multiply(9, 5); // Result 45
12. }(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：

1. var foo = function() {
2. return true;
3. }; // semicolon here.
5. function foo() {
6. return true;
7. } // 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**

1. if (x) {
2. function foo() {}
3. }

**推荐**

1. if (x) {
2. var foo = function() {};
3. }

**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：

1. 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**

1. if(x === 10) {
2. return 'valid';
3. } else {
4. return 'invalid';
5. }
6. **推荐**
8. return x === 10 ? 'valid' : 'invalid';