JavaScript 编码规范

目录
代码重构
间距
对象声明
数组与函数调用
Examples of Good Spacing
Semicolons
Indentation and Line Breaks
Blocks and Curly Braces
多行语句
链式方法调用
赋值与全局变量
使用 const 和 let 声明变量
使用 var 声明变量
全局变量
Common Libraries
命名规范
缩写与首字母缩略词
类定义
常量
Comments
相等性
类型检查
字符串
Switch Statements
最佳实践
数组
对象
Iteration

- Underscore.js 集合函数
- Iterating Over jQuery Collections
- JSHint
  - 安装与运行 JSHint
  - JSHint 设置
  - 针对单个文件
  - JSHint Overrides: Ignore Blocks
- 致谢

JavaScript 已成为开发基于 WordPress 的应用程序（主题和插件）以及 WordPress 核心的关键组成部分。需要制定 JavaScript 代码格式化和样式规范，以保持与 WordPress 核心 PHP、HTML 和 CSS 代码规范相同的代码一致性。

> 任何代码库中的所有代码都应看起来像是同一个人编写的，无论有多少人参与贡献。 —— 编写一致、惯用 JavaScript 的原则

WordPress JavaScript 编码规范改编自 jQuery JavaScript 风格指南。我们的标准在以下方面与 jQuery 指南有所不同：

- WordPress 在字符串声明中使用单引号。
- case 语句在 switch 块内缩进。
- 函数内容（包括全文件闭包包装器）保持一致的缩进。
- 部分空格规则有所不同，以与 WordPress PHP 编码规范保持一致。
- 鼓励遵循 jQuery 的 100 字符硬性行宽限制，但非严格强制。

以下许多示例直接改编自 jQuery 风格指南；这些差异已全部整合到本页的示例中。除非明确标注为反模式，以下所有标准和示例均应视为 WordPress 代码的最佳实践。

代码重构

> "不应仅仅因为我们能够重构代码就进行重构。" - 首席开发者 Andrew Nacin

WordPress 的 JavaScript 代码结构在许多部分的风格上并不一致。 WordPress 正在努力逐步改进这一点，以使代码整洁且易于一目了然地阅读。

尽管编码标准很重要，但仅仅为了符合标准而重构旧的 .js 文件并非紧急事项。强烈不建议为旧文件提交 “ 仅修改空白字符 ”的补丁。

所有新增或更新的 JavaScript 代码都将经过审查，以确保其符合标准，并通过 JSHint 检查。

间距
在代码中应自由使用空格。 "如有疑问，留出空间。 "

这些规则鼓励使用宽松的间距以提升开发者可读性。压缩过程会生成针对浏览器读取和处理优化的文件。

使用制表符进行缩进。
行尾或空行上不应有空白字符。
行长度通常不应超过 80 个字符，且不应超过 100 个字符（将制表符计为 4 个空格）。 此为 "软性 "
规则，但长行通常表示代码难以阅读或结构混乱。
if / else / for / while / try 代码块应始终使用大括号，且始终跨越多行。
一元特殊字符运算符（例如 ++、 --）不得在操作数旁添加空格。
任何, 和; 前不得有空格。
用作语句终止符的任何 ; 必须位于行尾。
对象定义中属性名后的任何 : 前不得有空格。
三元条件运算符中的 ? 和: 两侧必须有空格。
空构造中不得有填充空格（例如 {}、 [] 、 fn()）。
每个文件末尾应有空行。
任何! 否定运算符后应有空格。 *
所有函数体应缩进一个制表符，即使整个文件被包裹在闭包中。 *
空格可用于在文档块内或行内对齐代码，但行首应仅使用制表符。 *

*：WordPress JavaScript 标准比 jQuery 风格指南更倾向于稍宽松的空白字符规则。这些差异是为了保持 WordPress 代码库中 PHP 和 JavaScript 文件的一致性。

空白字符容易在行尾累积 ——请避免这种情况，因为尾随空格在 JSHint 中会被视为错误。检测空白字符累积的一种方法是在文本编辑器中启用可见空白字符显示。

对象声明

如果对象声明较短（请遵循行长度指南），可以单行声明。当对象声明过长无法单行显示时，每个属性必须独
占一行且每行以逗号结尾。属性名仅在保留字或包含特殊字符时才需要引号：

如果数组较短（请遵循行长度指南），可以单行声明。当数组过长无法单行显示时，每个元素必须独占一行且
每行以逗号结尾。

```
// □□□□
var obj = {
□ready: 9,
□when: 4,
□'you are': 15,
};
var arr = [
□9,
□4,
□15,
];

// □□□□□□□□□□□
var obj = { ready: 9, when: 4, 'you are': 15 };
var arr = [ 9, 4, 15 ];

// □□□□
var obj = { ready: 9,
□when: 4, 'you are': 15 };
var arr = [ 9,
□4, 15 ];
```

□□□□□□

□□□□□□□□□□□□□□□□□□

```
array = [ a, b ];


foo( arg );


foo( 'string', object );


foo( options, object[ property ] );


foo( node, 'property', 2 );


prop = object[ 'default' ];


firstArrayElement = arr[ 0 ];
```

Examples of Good Spacing

```
var i;

if ( condition ) {
	doSomething( 'with a string' );
} else if ( otherCondition ) {
	otherThing( {
		key: value,
		otherKey: otherValue
	} );
} else {
	somethingElse( true );
}

// Unlike jQuery, WordPress prefers a space after the ! negation operator.
// This is also done to conform to our PHP standards.
while ( ! condition ) {
	iterating++;
}

for ( i = 0; i &lt; 100; i++ ) {
	object[ array[ i ] ] = someFn( i );
	$( '.container' ).val( array[ i ] );
}

try {
	// Expressions
} catch ( e ) {
	// Expressions
}
```

Semicolons

Use them. Never rely on Automatic Semicolon Insertion (ASI).

Indentation and Line Breaks

Indentation and line breaks add readability to complex statements.

Tabs should be used for indentation. Even if the entire file is contained in a closure (i.e., an immediately invoked function), the contents of that function should be indented by one tab:

```
( function ( $ ) {
	// Expressions indented

	function doSomething() {
		// Expressions indented
	}
} )( jQuery );
```

Blocks and Curly Braces

`if`, `else`, `for`, `while`, and `try` blocks should always use braces, and always go on multiple lines. The opening brace should be on the same line as the function definition, the conditional, or the loop. The closing brace should be on the line directly following the last statement of the block.

```
var a, b, c;

if ( myFunction() ) {
	// Expressions
} else if ( ( a &amp;&amp; b ) || c ) {
	// Expressions
} else {
	// Expressions
}
```

□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
// □□□□
var html = '&lt;p>The sum of ' + a + ' and ' + b + ' plus ' + c
	+ ' is ' + ( a + b + c ) + '&lt;/p>';
```

```
// 正确示例
var html = '&lt;p>The sum of ' + a + ' and ' + b + ' plus ' + c +
' is ' + ( a + b + c ) + '&lt;/p>';
```

如果有助于提高可读性，应将行拆分为逻辑组，例如将三元运算符的每个表达式拆分到单独一行，即使两者都能放在同一行内。

```
// 可接受
var baz = ( true === conditionalStatement() ) ? 'thing 1' : 'thing 2';

// 更佳
var baz = firstCondition( foo ) &amp;&amp; secondCondition( bar ) ?
qux( foo, bar ) :
foo;
```

当条件语句过长无法在一行内显示时，布尔表达式中逻辑运算符的每个操作数都必须独占一行，并比开括号和闭括号多缩进一级。

```
if (
firstCondition() &amp;&amp;
secondCondition() &amp;&amp;
thirdCondition()
) {
    doStuff();
}
```

链式方法调用

当方法调用链过长无法单行显示时，每个方法调用必须独占一行，且首个方法调用应与调用对象所在行分离。若方法改变了上下文，则需额外增加一级缩进。

```
elements
.addClass( 'foo' )
.children()
..html( 'hello' )
.end()
.appendTo( 'body' );
```

赋值与全局变量

使用 `const` 和 `let` 声明变量

对于使用 ES2015 或更新版本编写的代码，应始终使用 `const` 和 `let` 替代 `var`。声明应优先使用 `const`，除非其值会被重新赋值，此时使用 `let` 更为合适。

与 `var` 不同，无需在函数顶部声明所有变量。相反，应在首次使用变量的位置进行声明。

使用 `var` 声明变量

每个函数都应以一个逗号分隔的 `var` 语句开头，用于声明所有必需的局部变量。如果函数未使用 `var` 声明变量，该变量可能会泄漏到外部作用域（通常是全局作用域，这是最坏的情况），并可能无意中引用和修改该数据。

`var` 语句中的赋值应单独成行列出，而声明可以组合在一行中。任何额外的行都应使用一个额外的制表符进行缩进。占用多行的对象和函数应在 `var` 语句之外进行赋值，以避免过度缩进。

```
// 正确
var k, m, length,
    //后续行缩进一个制表符
    value = 'WordPress';

// 错误
var foo = true;
var bar = false;
var a;
var b;
var c;
```

全局变量

过去，WordPress 核心代码大量使用全局变量。由于核心 JavaScript 文件有时会在插件中使用，因此不应移除现有的全局变量。

文件中使用的所有全局变量应在文件顶部进行文档说明。多个全局变量可以用逗号分隔。

以下示例将使 `passwordStrength` 成为该文件中允许的全局变量：

```
/* global passwordStrength:true */
```

`passwordStrength` 后面的 "true"
表示此全局变量是在此文件中定义的。如果你要访问在其他地方定义的全局变量，请省略 `:true`
以将该全局变量指定为只读。

Common Libraries

Backbone, jQuery, Underscore, and the global `wp` object are all registered as allowed globals in the root `.jshintrc` file.

Backbone and Underscore may be accessed directly at any time. jQuery should be accessed through `$` by passing the `jQuery` object into an anonymous function:

```
( function ( $ ) {
// Expressions
} )( jQuery );
```

This will negate the need to call `.noConflict()`, or to set `$` using another variable.

Files which add to, or modify, the `wp` object must safely access the global to avoid overwriting previously set properties:

```
// At the top of the file, set "wp" to its existing value (if present)
window.wp = window.wp || {};
```

命名规范

变量和函数名应使用完整的单词，采用首字母小写的驼峰命名法。这是本标准与 WordPress PHP 编码标准
存在差异的一个方面。

名称应具有描述性，但不应过度冗长。允许对迭代器使用例外情况，例如在循环中使用 `i` 表示索引。

缩写与首字母缩略词

首字母缩略词
必须将其每个组成字母大写。这旨在反映缩略词的每个字母在其展开形式中都是一个独立的单词。

所有其他 缩写 必须采用驼峰式书写，即首字母大写，后续字母小写。

如果缩写或首字母缩略词出现在变量名的开头，其书写必须遵循覆盖变量或类定义首字母的驼峰命名规则。对于变量赋值，这意味着将缩写全部写为小写。对于类定义，其首字母应大写。

```
// "Id" 是 "Identifier" 的缩写:
const userId = 1;

// "DOM" 是 "Document Object Model" 的首字母缩略词:
const currentDOMDocument = window.document;

// 位于变量名开头的首字母缩略词和缩写
// 遵循覆盖变量或类首字母的驼峰规则。
const domDocument = window.document;
class DOMDocument {}
class IdCollection {}
```

类定义

用于 `new` 调用的构造函数首字母应大写（采用 UpperCamelCase 命名法）。

`class` 定义必须使用 UpperCamelCase 命名规范，无论其是否计划用于 `new` 构造。

```
class Earth {
  static addHuman( human ) {
    Earth.humans.push( human );
  }

  static getHumans() {
    return Earth.humans;
  }
}

Earth.humans = [];
```

所有 `@wordpress/element` 组件（包括无状态函数组件）都应遵循类定义的命名规则。这既是为了保持一致性，也考虑到组件未来可能需要从函数形式转换为类形式而不破坏兼容性。

常量

对于永不打算重新赋值或修改的常量值，驼峰命名法存在例外情况。此类变量必须使用 全大写蛇形命名法_____。

几乎所有情况下，常量都应在文件的最顶层作用域中定义。需要特别注意的是，在概念上比此处暗示的含义更为有限——JavaScript 中通过 `const` 赋值的变量实际上仍可被修改，仅受保护不被重新赋值。本编码规范定义的常量仅适用于预期永不改变的值，这更多是开发者传达意图的策略，而非技术限制。

JavaScript 的 `const` 赋值__

Comments

Comments come before the code to which they refer, and should always be preceded by a blank line. Capitalize the first letter of the comment, and include a period at the end when writing full sentences. There must be a single space between the comment token (`//`) and the comment text.

```
someStatement();

// Explanation of something complex on the next line
$( 'p' ).doSomething();

// This is a comment that is long enough to warrant being stretched
// over the span of multiple lines.
```

JSDoc comments should use the `/**` multi-line comment opening. Refer to the JavaScript Documentation Standards for more information.

Inline comments are allowed as an exception when used to annotate special arguments in formal parameter lists:

```
function foo( types, selector, data, fn, /* INTERNAL */ one ) {
	// Do stuff
}
```

相等性

必须使用严格相等检查（`===`），而不是抽象相等检查（`==`）。

类型检查

□□□□□□□□□□□□□□□

- □□□□ `typeof object === 'string'`
- □□□ `typeof object === 'number'`
- □□□□ `typeof object === 'boolean'`
- □□□ `typeof object === 'object'` □ `_.isObject( object )`
- □□□□ `jQuery.isPlainObject( object )`
- □□□ `_.isFunction( object )` □ `jQuery.isFunction( object )`
- □□□ `_.isArray( object )` □ `jQuery.isArray( object )`
- □□□ `object.nodeType` □ `_.isElement( object )`
- null□ `object === null`
- null □ undefined□ `object == null`
- undefined□
  - □□□□□ `typeof variable === 'undefined'`
  - □□□□□ `variable === undefined`
  - □□□ `object.prop === undefined`
  - □□□□□□□ `_.isUndefined( object )`

□□□□ Backbone □ Underscore □□□□□□□ Underscore.js □□□□□□□□□□ jQuery □□

□□□

□□□□□□□□□□□□□

```
var myStr = 'strings should be contained in single quotes';
```

□□□□□□□□□□□□□□□□□□ `\`□□□□□

```
// □□□□□□□□□□□□
'Note the backslash before the \'single quotes\'';
```

Switch Statements

The usage of `switch` statements is generally discouraged, but can be useful when there are a large number of cases - especially when multiple cases can be handled by the same block, or fall-through logic (the `default` case) can be leveraged.

When using `switch` statements:

- Use a `break` for each case other than `default`. When allowing statements to "fall through," note that explicitly.
- Indent `case` statements one tab within the `switch`.

```
switch ( event.keyCode ) {
	// ENTER and SPACE both trigger x()
	case $.ui.keyCode.ENTER:
	case $.ui.keyCode.SPACE:
		x();
		break;
	case $.ui.keyCode.ESCAPE:
		y();
		break;
	default:
		z();
}
```

It is not recommended to return a value from within a switch statement: use the `case` blocks to set values, then `return` those values at the end.

```
function getKeyCode( keyCode ) {
	var result;

	switch ( event.keyCode ) {
		case $.ui.keyCode.ENTER:
		case $.ui.keyCode.SPACE:
			result = 'commit';
			break;
		case $.ui.keyCode.ESCAPE:
			result = 'exit';
			break;
		default:
			result = 'default';
	}

	return result;
}
```

最佳实践

数组

在 JavaScript 中创建数组应使用简写 `[]` 构造函数，而非 `new Array()` 表示法。

```
var myArray = [];
```

你可以在构造期间初始化数组：

```
var myArray = [ 1, 'WordPress', 2, 'Blog' ];
```

在 JavaScript 中，关联数组被定义为对象。

对象

在 JavaScript 中有多种创建对象的方式。对象字面量表示法 `{}` 不仅性能最佳，也最易于阅读。

```
var myObj = {};
```

除非对象需要特定的原型，否则应使用对象字面量表示法。如果需要特定原型，则应通过使用 `new` 调用构造函数来创建对象。

```
var myObj = new ConstructorMethod();
```

对象属性应通过点表示法访问，除非键是变量或字符串且不是有效的标识符：

```
prop = object.propertyName;
prop = object[ variableKey ];
prop = object['key-with-hyphens'];
```

Iteration

When iterating over a large collection using a `for` loop, it is recommended to store the loop's max value as a variable rather than re-computing the maximum every time:

```
// Good &amp;amp; Efficient
var i, max;

// getItemCount() gets called once
for ( i = 0, max = getItemCount(); i &amp;lt; max; i++ ) {
  // Do stuff
}

// Bad &amp;amp; Potentially Inefficient:
// getItemCount() gets called every time
for ( i = 0; i &amp;lt; getItemCount(); i++ ) {
  // Do stuff
}
```

Underscore.js □□□□

□□□□□ Underscore □ □□□□□□□ □□□□□□□□ `_.each`□ `_.map` □ `_.reduce` □□□□□□□□□□□□□□□□□□□□□

Underscore □□□□□□ JavaScript □□□□ jQuery □□□□□□□□

```
var obj = {
  first: 'thing 1',
  second: 'thing 2',
  third: 'lox'
};

var arr = _.chain( obj )
  .keys()
  .map( function ( key ) {
    return key + ' comes ' + obj[ key ];
  } )
  //□□□□□□
  .value();

// arr === [ 'first comes thing 1', 'second comes thing 2', 'third comes lox' ]
```

Iterating Over jQuery Collections

The only time jQuery should be used for iteration is when iterating over a collection of jQuery objects:

```
$tabs.each( function ( index, element ) {
	var $element = $( element );

	// Do stuff to $element
} );
```

Never use jQuery to iterate over raw data or vanilla JavaScript objects.

JSHint

JSHint JavaScript WordPress JSHint

JSHint

JSHint Grunt JSHint Grunt Node.js WordPress `package.json`

Node.js Node.js

Node.js WordPress SVN `cd ~/`

`npm install` WordPress Node

`npm run grunt jshint` Grunt WordPress JavaScript `npm run grunt jshint:core` .js `npm run grunt jshint:tests`

JSHint

JSHint WordPress SVN `.jshintrc` JSHint WordPress

□□□□□

□□□ JSHint □□□□□□□□□□□□□□□□ `--file=filename.js` □□□□□□□□□□□□
WordPress □□ JavaScript □□□□ "admin-bar.js" □□□

```
npm run grunt jshint:core --file=admin-bar.js
```

□□□□□□□□□□□□□□□□ "password-strength-meter.js" □□□

```
npm run grunt jshint:tests --file=password-strength-meter.js
```

□□□□□□□□□□□□□□□□□□□□□□ JSHint □□□□□□□□□□□□ JSHint □□□□□□□□□□□□

JSHint Overrides: Ignore Blocks

In some situations, parts of a file should be excluded from JSHint. As an example, the script file for the admin bar contains the minified code for the jQuery HoverIntent plugin - this is third-party code that should not pass through JSHint, even though it is part of a WordPress core JavaScript file.

To exclude a specific file region from being processed by JSHint, enclose it in JSHint directive comments:

```
/* jshint ignore:start */
if ( typeof jQuery.fn.hoverIntent === 'undefined' ) {
□// hoverIntent r6 - Copy of wp-includes/js/hoverIntent.min.js
□(function(a){a.fn.hoverIntent=..............
}
/* jshint ignore:end */
```

□□

- □□□□ jQuery □□□□□ jQuery JavaScript □□□□ □□□□□□ MIT □□□□□□

□□□ #2
□ □□□ □□□ 2026-03-27 01:08:34 UTC
□ □□□ □□□ 2026-03-27 01:08:34 UTC