JavaScript 代码规范

目录

30. 测试

类型

- 1.1 **原始值**: 当你访问一个原始类型的时候，你可以直接使用它的值。
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`
  - `symbol`

```
const foo = 1;
let bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

  - 标识符不能完全被支持，因此在针对不支持的浏览器或者环境时不应该使用它们。

- 1.2 **复杂类型**: 当你访问一个复杂类型的时候，你需要一个值得引用。
  - `object`
  - `array`
  - `function`

```
const foo = [1, 2];
const bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

引用

- 2.1 使用 `const` 定义你的所有引用；避免使用 `var`。 eslint: `prefer-const`，`no-const-assign`

  > 为什么？这样能够确保你不能重新赋值你的引用，否则可能导致错误或者产生难以理解的代码。.

```
// bad
var a = 1;
var b = 2;

// good
const a = 1;
const b = 2;
```

- 2.2 如果你必须重新赋值你的引用， 使用 `let` 代替 `var`。 eslint: `no-var`

  > 为什么？`let` 是块级作用域，而不像 `var` 是函数作用域.

```
// bad
var count = 1;
if (true) {
  count += 1;
}

// good, use the let.
let count = 1;
if (true) {
  count += 1;
}
```

- 2.3 注意，let 和 const 都是块级范围的。

```
// const 和 let 只存在于他们定义的块中。
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

对象

- 3.1 使用字面语法来创建对象。 eslint: `no-new-object`

```
// bad
const item = new Object();

// good
const item = {};
```

- 3.2 在创建具有动态属性名称的对象时使用计算属性名。

> 为什么？它允许你在一个地方定义对象的所有属性。

```

function getKey(k) {
  return `a key named ${k}`;
}

// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;
```

```
12
13 // good
14 const obj = {
15   id: 5,
16   name: 'San Francisco',
17   [getKey('enabled')]: true,
18 };
```

- 3.3 使用对象方法的缩写。 eslint: `object-shorthand`

```
1 // bad
2 const atom = {
3   value: 1,
4
5   addValue: function (value) {
6     return atom.value + value;
7   },
8 };
9
10 // good
11 const atom = {
12   value: 1,
13
14   addValue(value) {
15     return atom.value + value;
16   },
17 };
```

- 3.4 使用属性值的缩写。 eslint: `object-shorthand`

> 为什么？它的写法和描述较短。

```
1 const lukeSkywalker = 'Luke Skywalker';
2
3 // bad
```

```
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

- 3.5 在对象声明的时候将简写的属性进行分组。

  > 为什么？这样更容易的判断哪些属性使用的简写。

```
const anakinSkywalker = 'Anakin Skywalker';
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker,
};

// good
const obj = {
  lukeSkywalker,
  anakinSkywalker,
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4,
};
```

- 3.6 只使用引号标注无效标识符的属性。 eslint: `quote-props`

  > 为什么？总的来说，我们认为这样更容易阅读。 它提升了语法高亮显示，并且更容易通过许多 JS 引擎优化。

```
// bad
const bad = {
  'foo': 3,
  'bar': 4,
  'data-blah': 5,
};

// good
const good = {
  foo: 3,
  bar: 4,
  'data-blah': 5,
};
```

- 3.7 不能直接调用 `Object.prototype` 的方法，如： `hasOwnProperty` 、 `propertyIsEnumerable` 和 `isPrototypeOf`。

  > 为什么？这些方法可能被一下问题对象的属性追踪 – 相应的有 `{ hasOwnProperty: false }` – 或者，对象是一个空对象 (`Object.create(null)`)。

```
// bad
console.log(object.hasOwnProperty(key));

// good
console.log(Object.prototype.hasOwnProperty.call(object, key));

// best
const has = Object.prototype.hasOwnProperty; // 在模块范围内的缓存中查找一次
/* or */
import has from 'has'; // https://www.npmjs.com/package/has
// ...
```

```
12 console.log(has.call(object, key));
```

- 3.8 更喜欢对象扩展操作符，而不是用 `Object.assign` 浅拷贝一个对象。 使用对象的 rest 操作符来获得一个具有某些属性的新对象。

```
1 // very bad
2 const original = { a: 1, b: 2 };
3 const copy = Object.assign(original, { c: 3 }); // 变异的 `original
  ` ಠ_ಠ
4 delete copy.a; // 这....
5
6 // bad
7 const original = { a: 1, b: 2 };
8 const copy = Object.assign({}, original, { c: 3 }); // copy => {
   a: 1, b: 2, c: 3 }
9
10 // good
11 const original = { a: 1, b: 2 };
12 const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3
   }
13
14 const { a, ...noA } = copy; // noA => { b: 2, c: 3 }
```

数组

- 4.1 使用字面语法创建数组。 eslint: `no-array-constructor`

```
1 // bad
2 const items = new Array();
3
4 // good
5 const items = [];
```

- 4.2 使用 Array#push 取代直接赋值来给数组添加项。

```
const someStack = [];

// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

- 4.3 使用数组展开方法 `...` 来拷贝数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i += 1) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

- 4.4 将一个类数组对象转换成一个数组， 使用展开方法 `...` 代替 `Array.from`。

```
const foo = document.querySelectorAll('.foo');

// good
const nodes = Array.from(foo);

```

```
// best
const nodes = [...foo];
```

- 4.5 对于对迭代器的映射，使用 Array.from 替代展开方法 `...` ， 因为它避免了创建中间数组。

```
// bad
const baz = [...foo].map(bar);

// good
const baz = Array.from(foo, bar);
```

- 4.6 在数组回调方法中使用 return 语句。 如果函数体由一个返回无副作用的表达式的单个语句组成，那么可以省略返回值， 具体查看 8.2。 eslint: `array-callback-return`

```
// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map(x => x + 1);

// bad - 没有返回值，意味着在第一次迭代后 `acc` 没有被定义
[[0, 1], [2, 3], [4, 5]].reduce((acc, item, index) => {
  const flatten = acc.concat(item);
  acc[index] = flatten;
});

// good
[[0, 1], [2, 3], [4, 5]].reduce((acc, item, index) => {
  const flatten = acc.concat(item);
  acc[index] = flatten;
  return flatten;
```

```
});

// bad
inbox.filter((msg) => {
  const { subject, author } = msg;
  if (subject === 'Mockingbird') {
    return author === 'Harper Lee';
  } else {
    return false;
  }
});

// good
inbox.filter((msg) => {
  const { subject, author } = msg;
  if (subject === 'Mockingbird') {
    return author === 'Harper Lee';
  }

  return false;
});
```

- 4.7 如果数组有多行，则在开始的时候换行，然后在结束的时候换行。

```
// bad
const arr = [
  [0, 1], [2, 3], [4, 5],
];

const objectInArray = [{
  id: 1,
}, {
  id: 2,
}];

const numberInArray = [
  1, 2,
```

```
];

// good
const arr = [[0, 1], [2, 3], [4, 5]];

const objectInArray = [
  {
    id: 1,
  },
  {
    id: 2,
  },
];

const numberInArray = [
  1,
  2,
];
```

解构

- 5.1 在访问和使用对象的多个属性的时候使用对象的解构。 eslint: `prefer-destructuring`

  > 为什么？解构可以避免为这些属性创建临时引用。

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// good
function getFullName(user) {
  const { firstName, lastName } = user;
```

```
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

- 5.2 使用数组解构。 eslint: `prefer-destructuring`

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

- 5.3 对于多个返回值使用对象解构，而不是数组解构。

> 为什么？你可以随时添加新的属性或者改变属性的顺序，而不用修改调用方。

```
// bad
function processInput(input) {
  // 处理代码...
  return [left, right, top, bottom];
}

// 调用者需要考虑返回数据的顺序。
const [left, __, top] = processInput(input);

// good
function processInput(input) {
  // 处理代码...
```

```
  return { left, right, top, bottom };
}

// 调用者只选择他们需要的数据。
const { left, top } = processInput(input);
```

字符

- 6.1 使用单引号 `''` 定义字符串。 eslint: `quotes`

```
// bad
const name = "Capt. Janeway";

// bad - 模板文字应该包含插值或换行。
const name = `Capt. Janeway`;

// good
const name = 'Capt. Janeway';
```

- 6.2 使行超过100个字符的字符串不应使用字符串连接跨多行写入。

  > 为什么？断开的字符串更加难以工作，并且使代码搜索更加困难。

```
// bad
const errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// bad
const errorMessage = 'This is a super long error that was thrown b
```

```
ecause ' +
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';

// good
const errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about how Batman had anything to do with this, you would get nowhere fast.';
```

- 6.3 当以编程模式构建字符串时，使用字符串模板代替字符串拼接。 eslint: `prefer-template` `template-curly-spacing`

  > 为什么？字符串模板为您提供了一种可读的、简洁的语法，具有正确的换行和字符串插值特性。

```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// bad
function sayHi(name) {
  return `How are you, ${ name }?`;
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

- 6.4 不要在字符串上使用 `eval()` ，它打开了太多漏洞。 eslint: `no-eval`

- 6.5 不要转义字符串中不必要的字符。 eslint: `no-useless-escape`

  > 为什么？反斜杠损害了可读性，因此只有在必要的时候才会出现。

```
// bad
const foo = '\'this\' \i\s \"quoted\"';

// good
const foo = '\'this\' is "quoted"';
const foo = `my name is '${name}'`;
```

方法

- 7.1 使用命名的函数表达式代替函数声明。 eslint: `func-style`

  > 为什么？函数声明是挂起的，这意味着在它在文件中定义之前，很容易引用函数。这会损害可读性和可维护性。如果您发现函数的定义是大的或复杂的，以至于它干扰了对文件的其余部分的理解，那么也许是时候将它提取到它自己的模块中了!不要忘记显式地命名这个表达式，不管它的名称是否从包含变量(在现代浏览器中经常是这样，或者在使用诸如Babel之类的编译器时)。这消除了对错误的调用堆栈的任何假设。（Discussion）

```
// bad
function foo() {
  // ...
}

// bad
const foo = function () {
  // ...
};

// good
// 从变量引用调用中区分的词汇名称
```

```
const short = function longUniqueMoreDescriptiveLexicalFoo() {
  // ...
};
```

- 7.2 Wrap立即调用函数表达式。 eslint: `wrap-iife`

> 为什么? 立即调用的函数表达式是单个单元 – 包装， 并且拥有括号调用，在括号内，清晰的表达式。 请注意，在一个到处都是模块的世界中，您几乎不需要一个 IIFE 。

```
// immediately-invoked function expression (IIFE) 立即调用的函数表达式
(function () {
  console.log('Welcome to the Internet. Please follow me.');
}());
```

- 7.3 切记不要在非功能块中声明函数 (`if`, `while`, 等)。 将函数赋值给变量。 浏览器允许你这样做，但是他们都有不同的解释，这是个坏消息。 eslint: `no-loop-func`

- 7.4 **注意:** ECMA-262 将 `block` 定义为语句列表。 函数声明不是语句。

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
  };
}
```

- 7.5 永远不要定义一个参数为 `arguments`。 这将会优先于每个函数给定范围的 `arguments` 对象。

```
// bad
function foo(name, options, arguments) {
  // ...
}

// good
function foo(name, options, args) {
  // ...
}
```

- 7.6 不要使用 `arguments`, 选择使用 rest 语法 `...` 代替。 eslint: `prefer-rest-params`

> 为什么? `...` 明确了你想要拉取什么参数。 更甚, rest 参数是一个真正的数组，而不仅仅是类数组的 `arguments` 。

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

- 7.7 使用默认的参数语法，而不是改变函数参数。

```
// really bad
```

```
function handleThings(opts) {
  // No! We shouldn't mutate function arguments.
  // Double bad: if opts is falsy it'll be set to an object which may
  // be what you want but it can introduce subtle bugs.
  opts = opts || {};
  // ...
}

// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}

// good
function handleThings(opts = {}) {
  // ...
}
```

- 7.8 避免使用默认参数的副作用。

  > 为什么？他们很容易混淆。

```
var b = 1;
// bad
function count(a = b++) {
  console.log(a);
}
count();  // 1
count();  // 2
count(3); // 3
count();  // 3
```

- 7.9 总是把默认参数放在最后。

```
// bad
function handleThings(opts = {}, name) {
  // ...
}

// good
function handleThings(name, opts = {}) {
  // ...
}
```

- 7.10 永远不要使用函数构造器来创建一个新函数。 eslint: `no-new-func`

> 为什么？以这种方式创建一个函数将对一个类似于 `eval()` 的字符串进行计算，这将打开漏洞。

```
// bad
var add = new Function('a', 'b', 'return a + b');

// still bad
var subtract = Function('a', 'b', 'return a - b');
```

- 7.11 函数签名中的间距。 eslint: `space-before-function-paren` `space-before-blocks`

> 为什么？一致性很好，在删除或添加名称时不需要添加或删除空格。

```
// bad
const f = function(){};
const g = function (){};
const h = function() {};

// good
const x = function () {};
const y = function a() {};
```

- 7.12 没用变异参数。 eslint: `no-param-reassign`

  > 为什么？将传入的对象作为参数进行操作可能会在原始调用程序中造成不必要的变量副作用。

```
// bad
function f1(obj) {
  obj.key = 1;
}

// good
function f2(obj) {
  const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? ob
j.key : 1;
}
```

- 7.13 不要再赋值参数。 eslint: `no-param-reassign`

  > 为什么？重新赋值参数会导致意外的行为，尤其是在访问 `arguments` 对象的时候。 它还可能导致性能优化问题，尤其是在 V8 中。

```
// bad
function f1(a) {
  a = 1;
  // ...
}

function f2(a) {
  if (!a) { a = 1; }
  // ...
}

// good
function f3(a) {
  const b = a || 1;
  // ...
```

```
}

function f4(a = 1) {
  // ...
}
```

- 7.14 优先使用扩展运算符 `...` 来调用可变参数函数。 eslint: `prefer-spread`

  > 为什么？它更加干净，你不需要提供上下文，并且你不能轻易的使用 `apply` 来 `new` 。

```
// bad
const x = [1, 2, 3, 4, 5];
console.log.apply(console, x);

// good
const x = [1, 2, 3, 4, 5];
console.log(...x);

// bad
new (Function.prototype.bind.apply(Date, [null, 2016, 8, 5]));

// good
new Date(...[2016, 8, 5]);
```

- 7.15 具有多行签名或者调用的函数应该像本指南中的其他多行列表一样缩进：在一行上只有一个条目，并且每个条目最后加上逗号。 eslint: `function-paren-newline`

```
// bad
function foo(bar,
             baz,
             quux) {
  // ...
}

// good
```

```
function foo(
  bar,
  baz,
  quux,
) {
  // ...
}

// bad
console.log(foo,
  bar,
  baz);

// good
console.log(
  foo,
  bar,
  baz,
);
```

箭头函数

- 8.1 当你必须使用匿名函数时 (当传递内联函数时)， 使用箭头函数。 eslint: `prefer-arrow-callback`, `arrow-spacing`

  > 为什么? 它创建了一个在 `this` 上下文中执行的函数版本，它通常是你想要的，并且是一个更简洁的语法。

  > 为什么不? 如果你有一个相当复杂的函数，你可以把这个逻辑转移到它自己的命名函数表达式中。

```
// bad
[1, 2, 3].map(function (x) {
  const y = x + 1;
  return x * y;
});

```

```
// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

- 8.2 如果函数体包含一个单独的语句，返回一个没有副作用的 expression ， 省略括号并使用隐式返回。否则，保留括号并使用 `return` 语句。 eslint: `arrow-parens`, `arrow-body-style`

  > 为什么？语法糖。 多个函数被链接在一起时，提高可读性。

```
// bad
[1, 2, 3].map(number => {
  const nextNumber = number + 1;
  `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map(number => `A string containing the ${number}.`);

// good
[1, 2, 3].map((number) => {
  const nextNumber = number + 1;
  return `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map((number, index) => ({
  [index]: number,
}));

// 没有副作用的隐式返回
function foo(callback) {
  const val = callback();
  if (val === true) {
    // 如果回调返回 true 执行
  }
}
```

```
28
29 let bool = false;
30
31 // bad
32 foo(() => bool = true);
33
34 // good
35 foo(() => {
36   bool = true;
37 });
```

- 8.3 如果表达式跨越多个行，用括号将其括起来，以获得更好的可读性。

  为什么？它清楚地显示了函数的起点和终点。

```
1 // bad
2 ['get', 'post', 'put'].map(httpMethod => Object.prototype.hasOwnPr
  operty.call(
3     httpMagicObjectWithAVeryLongName,
4     httpMethod,
5   )
6 );
7
8 // good
9 ['get', 'post', 'put'].map(httpMethod => (
10   Object.prototype.hasOwnProperty.call(
11     httpMagicObjectWithAVeryLongName,
12     httpMethod,
13   )
14 ));
```

- 8.4 如果你的函数接收一个参数，则可以不用括号，省略括号。 否则，为了保证清晰和一致性，需要在参数周围加上括号。 注意：总是使用括号是可以接受的，在这种情况下，我们使用 “always” option 来配置 eslint. eslint: `arrow-parens`

  为什么？减少视觉上的混乱。

```
// bad
[1, 2, 3].map((x) => x * x);

// good
[1, 2, 3].map(x => x * x);

// good
[1, 2, 3].map(number => (
  `A long string with the ${number}. It's so long that we don't want it to take up space on the .map line!`
));

// bad
[1, 2, 3].map(x => {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

- 8.5 避免箭头函数符号 (`=>`) 和比较运算符 (`<=`, `>=`) 的混淆。 eslint: `no-confusing-arrow`

```
// bad
const itemHeight = item => item.height > 256 ? item.largeSize : item.smallSize;

// bad
const itemHeight = (item) => item.height > 256 ? item.largeSize : item.smallSize;

// good
const itemHeight = item => (item.height > 256 ? item.largeSize : i
```

```
  tem.smallSize);

// good
const itemHeight = (item) => {
  const { height, largeSize, smallSize } = item;
  return height > 256 ? largeSize : smallSize;
};
```

- 8.6 注意带有隐式返回的箭头函数函数体的位置。 eslint: `implicit-arrow-linebreak`

```
// bad
(foo) =>
  bar;

(foo) =>
  (bar);

// good
(foo) => bar;
(foo) => (bar);
(foo) => (
   bar
)
```

类和构造器

- 9.1 尽量使用 `class`. 避免直接操作 `prototype` .

  为什么? `class` 语法更简洁，更容易推理。

```
// bad
function Queue(contents = []) {
  this.queue = [...contents];
```

```
}
Queue.prototype.pop = function () {
  const value = this.queue[0];
  this.queue.splice(0, 1);
  return value;
};

// good
class Queue {
  constructor(contents = []) {
    this.queue = [...contents];
  }
  pop() {
    const value = this.queue[0];
    this.queue.splice(0, 1);
    return value;
  }
}
```

- 9.2 使用 `extends` 来扩展继承。

  > 为什么？它是一个内置的方法，可以在不破坏 `instanceof` 的情况下继承原型功能。

```
// bad
const inherits = require('inherits');
function PeekableQueue(contents) {
  Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function () {
  return this.queue[0];
};

// good
class PeekableQueue extends Queue {
  peek() {
    return this.queue[0];
```

```
  }
}
```

- 9.3 方法返回了 `this` 来供其内部方法调用。

```
// bad
Jedi.prototype.jump = function () {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function (height) {
  this.height = height;
};

const luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20); // => undefined

// good
class Jedi {
  jump() {
    this.jumping = true;
    return this;
  }

  setHeight(height) {
    this.height = height;
    return this;
  }
}

const luke = new Jedi();

luke.jump()
  .setHeight(20);
```

- 9.4 只要在确保能正常工作并且不产生任何副作用的情况下，编写一个自定义的 `toString()` 方法也是可以的。

```
class Jedi {
  constructor(options = {}) {
    this.name = options.name || 'no name';
  }

  getName() {
    return this.name;
  }

  toString() {
    return `Jedi - ${this.getName()}`;
  }
}
```

- 9.5 如果没有指定类，则类具有默认的构造器。 一个空的构造器或是一个代表父类的函数是没有必要的。 eslint: `no-useless-constructor`

```
// bad
class Jedi {
  constructor() {}

  getName() {
    return this.name;
  }
}

// bad
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
  }
```

```
}

// good
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
    this.name = 'Rey';
  }
}
```

- 9.6 避免定义重复的类成员。 eslint: `no-dupe-class-members`

  > 为什么？重复的类成员声明将会默认倾向于最后一个 – 具有重复的类成员可以说是一个错误。

```
// bad
class Foo {
  bar() { return 1; }
  bar() { return 2; }
}

// good
class Foo {
  bar() { return 1; }
}

// good
class Foo {
  bar() { return 2; }
}
```

模块

- 10.1 你可能经常使用模块 (`import`/`export`) 在一些非标准模块的系统上。 你也可以在你喜欢的模块系统上相互转换。

  > 为什么？模块是未来的趋势，让我们拥抱未来。

```
// bad
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;

// ok
import AirbnbStyleGuide from './AirbnbStyleGuide';
export default AirbnbStyleGuide.es6;

// best
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- 10.2 不要使用通配符导入。

  > 为什么？这确定你有一个单独的默认导出。

```
// bad
import * as AirbnbStyleGuide from './AirbnbStyleGuide';

// good
import AirbnbStyleGuide from './AirbnbStyleGuide';
```

- 10.3 不要直接从导入导出。

  > 为什么？虽然写在一行很简洁，但是有一个明确的导入和一个明确的导出能够保证一致性。

```
// bad
// filename es6.js
export { es6 as default } from './AirbnbStyleGuide';

// good
```

```
// filename es6.js
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

- 10.4 只从一个路径导入所有需要的东西。
  eslint: `no-duplicate-imports`

  > 为什么？从同一个路径导入多个行，使代码更难以维护。

```
// bad
import foo from 'foo';
// … 其他导入 … //
import { named1, named2 } from 'foo';

// good
import foo, { named1, named2 } from 'foo';

// good
import foo, {
  named1,
  named2,
} from 'foo';
```

- 10.5 不要导出可变的引用。
  eslint: `import/no-mutable-exports`

  > 为什么？在一般情况下，应该避免发生突变，但是在导出可变引用时及其容易发生突变。虽然在某些特殊情况下，可能需要这样，但是一般情况下只需要导出常量引用。

```
// bad
let foo = 3;
export { foo };

// good
const foo = 3;
export { foo };
```

- 10.6 在单个导出的模块中，选择默认模块而不是指定的导出。
  eslint: `import/prefer-default-export`

  > 为什么？为了鼓励更多的文件只导出一件东西，这样可读性和可维护性更好。

```
// bad
export function foo() {}

// good
export default function foo() {}
```

- 10.7 将所有的 `import`s 语句放在所有非导入语句的上边。
  eslint: `import/first`

  > 为什么？由于所有的 `import`s 都被提前，保持他们在顶部是为了防止意外发生。

```
// bad
import foo from 'foo';
foo.init();

import bar from 'bar';

// good
import foo from 'foo';
import bar from 'bar';

foo.init();
```

- 10.8 多行导入应该像多行数组和对象一样缩进。

  > 为什么？花括号和其他规范一样，遵循相同的缩进规则，后边的都好一样。

```
```

```
// bad
import {longNameA, longNameB, longNameC, longNameD, longNameE} fro
m 'path';

// good
import {
  longNameA,
  longNameB,
  longNameC,
  longNameD,
  longNameE,
} from 'path';
```

- 10.9 在模块导入语句中禁止使用 Webpack 加载器语法。
  eslint: `import/no-webpack-loader-syntax`

  > 为什么？因为在导入语句中使用 webpack 语法，将代码和模块绑定在一起。应该在 `webpack.config.js` 中使用加载器语法。

```
// bad
import fooSass from 'css!sass!foo.scss';
import barCss from 'style!css!bar.css';

// good
import fooSass from 'foo.scss';
import barCss from 'bar.css';
```

迭代器和发生器

- 11.1 不要使用迭代器。 你应该使用 JavaScript 的高阶函数代替 `for-in` 或者 `for-of`。 eslint: `no-iterator` `no-restricted-syntax`

  > 为什么？这是我们强制的规则。 拥有返回值得纯函数比这个更容易解释。

使用 `map()` / `every()` / `filter()` / `find()` / `findIndex()` / `reduce()` / `some()` / ... 遍历数组， 和使用 `Object.keys()` / `Object.values()` / `Object.entries()` 迭代你的对象生成数组。

```
const numbers = [1, 2, 3, 4, 5];

// bad
let sum = 0;
for (let num of numbers) {
  sum += num;
}
sum === 15;

// good
let sum = 0;
numbers.forEach((num) => {
  sum += num;
});
sum === 15;

// best (use the functional force)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;

// bad
const increasedByOne = [];
for (let i = 0; i < numbers.length; i++) {
  increasedByOne.push(numbers[i] + 1);
}

// good
const increasedByOne = [];
numbers.forEach((num) => {
  increasedByOne.push(num + 1);
});

// best (keeping it functional)
const increasedByOne = numbers.map(num => num + 1);
```

- 11.2 不要使用发生器。

  > 为什么？它们不能很好的适应 ES5。

- 11.3 如果你必须使用发生器或者无视 我们的建议，请确保他们的函数签名是正常的间隔。 eslint: `generator-star-spacing`

  > 为什么？`function` 和 `*` 是同一个概念关键字的一部分 - `*` 不是 `function` 的修饰符，`function*` 是一个不同于 `function` 的构造器。

```
// bad
function * foo() {
  // ...
}

// bad
const bar = function * () {
  // ...
};

// bad
const baz = function *() {
  // ...
};

// bad
const quux = function*() {
  // ...
};

// bad
function*foo() {
  // ...
}

// bad
function *foo() {
```

```
  // ...
}

// very bad
function
*
foo() {
  // ...
}

// very bad
const wat = function
*
() {
  // ...
};

// good
function* foo() {
  // ...
}

// good
const foo = function* () {
  // ...
};
```

属性

- 12.1 访问属性时使用点符号。 eslint: `dot-notation`

```
const luke = {
  jedi: true,
  age: 28,
};
```

```
5
6 // bad
7 const isJedi = luke['jedi'];
8
9 // good
10 const isJedi = luke.jedi;
```

- 12.2 使用变量访问属性时，使用 `[]` 表示法。

```
1 const luke = {
2   jedi: true,
3   age: 28,
4 };
5
6 function getProp(prop) {
7   return luke[prop];
8 }
9
10 const isJedi = getProp('jedi');
```

- 12.3 计算指数时，可以使用 `**` 运算符。 eslint: `no-restricted-properties`.

```
1 // bad
2 const binary = Math.pow(2, 10);
3
4 // good
5 const binary = 2 ** 10;
```

变量

- 13.1 使用 `const` 或者 `let` 来定义变量。 不这样做将创建一个全局变量。 我们希望避免污染全局命名空间。 Captain Planet 警告过我们。 eslint: `no-undef` `prefer-const`

```
// bad
superPower = new SuperPower();

// good
const superPower = new SuperPower();
```

- 13.2 使用 `const` 或者 `let` 声明每一个变量。 eslint: `one-var`

> 为什么？这样更容易添加新的变量声明，而且你不必担心是使用 `;` 还是使用 `,` 或引入标点符号的差别。 你可以通过 debugger 逐步查看每个声明，而不是立即跳过所有声明。

```
// bad
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

- 13.3 把 `const` 声明的放在一起，把 `let` 声明的放在一起。.

> 为什么？这在后边如果需要根据前边的赋值变量指定一个变量时很有用。

```
// bad
```

```
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// good
const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

- 13.4 在你需要的使用定义变量，但是要把它们放在一个合理的地方。

> 为什么？`let` 和 `const` 是块级作用域而不是函数作用域。

```
// bad - 不必要的函数调用
function checkName(hasName) {
  const name = getName();

  if (hasName === 'test') {
    return false;
  }

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}

```

```
// good
function checkName(hasName) {
  if (hasName === 'test') {
    return false;
  }

  const name = getName();

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}
```

- 13.5 不要链式变量赋值。 eslint: `no-multi-assign`

  > 为什么？链式变量赋值会创建隐式全局变量。

```
// bad
(function example() {
  // JavaScript 把它解释为
  // let a = ( b = ( c = 1 ) );
  // let 关键词只适用于变量 a ；变量 b 和变量 c 则变成了全局变量。
  let a = b = c = 1;
}());

console.log(a); // throws ReferenceError
console.log(b); // 1
console.log(c); // 1

// good
(function example() {
  let a = 1;
  let b = a;
  let c = a;
}());
```

```
19
20 console.log(a); // throws ReferenceError
21 console.log(b); // throws ReferenceError
22 console.log(c); // throws ReferenceError
23
24 // 对于 `const` 也一样
```

- 13.6 避免使用不必要的递增和递减 (`++`, `--`)。 eslint `no-plusplus`

> 为什么？在eslint文档中，一元递增和递减语句以自动分号插入为主题，并且在应用程序中可能会导致默认值的递增或递减。它还可以用像 `num += 1` 这样的语句来改变您的值，而不是使用 `num++` 或 `num ++` 。不允许不必要的增量和减量语句也会使您无法预先递增/预递减值，这也会导致程序中的意外行为。

```
1 // bad
2
3 const array = [1, 2, 3];
4 let num = 1;
5 num++;
6 --num;
7
8 let sum = 0;
9 let truthyCount = 0;
10 for (let i = 0; i < array.length; i++) {
11   let value = array[i];
12   sum += value;
13   if (value) {
14     truthyCount++;
15   }
16 }
17
18 // good
19
20 const array = [1, 2, 3];
21 let num = 1;
22 num += 1;
23 num -= 1;
```

```
24
25 const sum = array.reduce((a, b) => a + b, 0);
26 const truthyCount = array.filter(Boolean).length;
```

- 13.7 避免在赋值语句 `=` 前后换行。如果你的代码违反了 `max-len`， 使用括号包裹。 eslint `operator-linebreak`.

  > 为什么？在 `=` 前后换行，可能混淆赋的值。

```
1 // bad
2 const foo =
3   superLongLongLongLongLongLongLongFunctionName();
4
5 // bad
6 const foo
7   = 'superLongLongLongLongLongLongLongString';
8
9 // good
10 const foo = (
11   superLongLongLongLongLongLongLongFunctionName()
12 );
13
14 // good
15 const foo = 'superLongLongLongLongLongLongLongString';
```

提升

- 14.1 `var` 定义的变量会被提升到函数范围的最顶部，但是它的赋值不会。`const` 和 `let` 声明的变量受到一个称之为 Temporal Dead Zones (TDZ) 的新概念保护。 知道为什么 typeof 不再安全 是很重要的。

```
1 // 我们知道这个行不通 （假设没有未定义的全局变量）
2 function example() {
```

```
  console.log(notDefined); // => throws a ReferenceError
}

// 在引用变量后创建变量声明将会因变量提升而起作用。
// 注意: 真正的值 `true` 不会被提升。
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// 解释器将变量提升到函数的顶部
// 这意味着我们可以将上边的例子重写为:
function example() {
  let declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}

// 使用 const 和 let
function example() {
  console.log(declaredButNotAssigned); // => throws a ReferenceErr
or
  console.log(typeof declaredButNotAssigned); // => throws a Refer
enceError
  const declaredButNotAssigned = true;
}
```

- 14.2 匿名函数表达式提升变量名，而不是函数赋值。

```
function example() {
  console.log(anonymous); // => undefined

  anonymous(); // => TypeError anonymous is not a function

  var anonymous = function () {
    console.log('anonymous function expression');
  };
```

```
}
```

- 14.3 命名函数表达式提升的是变量名，而不是函数名或者函数体。

```
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
    console.log('Flying');
  };
}

// 当函数名和变量名相同时也是如此。
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  var named = function named() {
    console.log('named');
  };
}
```

- 14.4 函数声明提升其名称和函数体。

```
function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
```

```
  }
}
```

- 更多信息请参考 Ben Cherry 的 JavaScript Scoping & Hoisting。

比较运算符和等号

- 15.1 使用 `===` 和 `!==` 而不是 `==` 和 `!=`。 eslint: `eqeqeq`

- 15.2 条件语句，例如 `if` 语句使用 `ToBoolean` 的抽象方法来计算表达式的结果，并始终遵循以下简单的规则：
  - **Objects** 的取值为： **true**
  - **Undefined** 的取值为： **false**
  - **Null** 的取值为： **false**
  - **Booleans** 的取值为： **布尔值的取值**
  - **Numbers** 的取值为：如果为 **+0, -0, or NaN** 值为 **false** 否则为 **true**
  - **Strings** 的取值为: 如果是一个空字符串 `''` 值为 **false** 否则为 **true**

```
if ([0] && []) {
  // true
  // 一个数组（即使是空的）是一个对象，对象的取值为 true
}
```

- 15.3 对于布尔值使用简写，但是对于字符串和数字进行显式比较。

```
// bad
if (isValid === true) {
  // ...
}

// good
if (isValid) {
```

```
  // ...
}

// bad
if (name) {
  // ...
}

// good
if (name !== '') {
  // ...
}

// bad
if (collection.length) {
  // ...
}

// good
if (collection.length > 0) {
  // ...
}
```

- 15.4 获取更多信息请查看 Angus Croll 的 Truth Equality and JavaScript 。

- 15.5 在 `case` 和 `default` 的子句中，如果存在声明 (例如. `let`, `const`, `function`, 和 `class`)，使用大括号来创建块 。 eslint: `no-case-declarations`

  > 为什么？语法声明在整个 `switch` 块中都是可见的，但是只有在赋值的时候才会被初始化，这种情况只有在 `case` 条件达到才会发生。 当多个 `case` 语句定义相同的东西是，这会导致问题问题。

```
// bad
switch (foo) {
  case 1:
    let x = 1;
    break;
```

```
  case 2:
    const y = 2;
    break;
  case 3:
    function f() {
      // ...
    }
    break;
  default:
    class C {}
}

// good
switch (foo) {
  case 1: {
    let x = 1;
    break;
  }
  case 2: {
    const y = 2;
    break;
  }
  case 3: {
    function f() {
      // ...
    }
    break;
  }
  case 4:
    bar();
    break;
  default: {
    class C {}
  }
}
```

- 15.6 三目表达式不应该嵌套，通常是单行表达式。 eslint: `no-nested-ternary`

```
// bad
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null;

// 分离为两个三目表达式
const maybeNull = value1 > value2 ? 'baz' : null;

// better
const foo = maybe1 > maybe2
  ? 'bar'
  : maybeNull;

// best
const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

- 15.7 避免不必要的三目表达式。 eslint: `no-unneeded-ternary`

```
// bad
const foo = a ? a : b;
const bar = c ? true : false;
const baz = c ? false : true;

// good
const foo = a || b;
const bar = !!c;
const baz = !c;
```

- 15.8 使用该混合运算符时，使用括号括起来。 唯一例外的是标准算数运算符 (`+`, `-`, `*`, & `/`) 因为他们的优先级被广泛理解。 eslint: `no-mixed-operators`

> 为什么？这能提高可读性并且表明开发人员的意图。

```
// bad
```

```
const foo = a && b < 0 || c > 0 || d + 1 === 0;

// bad
const bar = a ** b - 5 % d;

// bad
// 可能陷入一种 (a || b) && c 的思考
if (a || b && c) {
  return d;
}

// good
const foo = (a && b < 0) || c > 0 || (d + 1 === 0);

// good
const bar = (a ** b) - (5 % d);

// good
if (a || (b && c)) {
  return d;
}

// good
const bar = a + b / c * d;
```

块

- 16.1 当有多行代码块的时候，使用大括号包裹。 eslint: `nonblock-statement-body-position`

```
// bad
if (test)
  return false;

// good
if (test) return false;
```

```
7
8  // good
9  if (test) {
10   return false;
11 }
12
13 // bad
14 function foo() { return false; }
15
16 // good
17 function bar() {
18   return false;
19 }
```

- 16.2 如果你使用的是 `if` 和 `else` 的多行代码块，则将 `else` 语句放在 `if` 块闭括号同一行的位置。 eslint: `brace-style`

```
1  // bad
2  if (test) {
3    thing1();
4    thing2();
5  }
6  else {
7    thing3();
8  }
9
10 // good
11 if (test) {
12   thing1();
13   thing2();
14 } else {
15   thing3();
16 }
```

- 16.3 如果一个 `if` 块总是执行一个 `return` 语句，那么接下来的 `else` 块就没有必要了。 如果一个包含 `return` 语句的 `else if` 块，在一个包含了 `return` 语句的 `if` 块之后，那么可以拆成多个 `if` 块。 eslint: `no-else-return`

```
// bad
function foo() {
  if (x) {
    return x;
  } else {
    return y;
  }
}

// bad
function cats() {
  if (x) {
    return x;
  } else if (y) {
    return y;
  }
}

// bad
function dogs() {
  if (x) {
    return x;
  } else {
    if (y) {
      return y;
    }
  }
}

// good
function foo() {
  if (x) {
    return x;
  }

```

```
  return y;
}

// good
function cats() {
  if (x) {
    return x;
  }

  if (y) {
    return y;
  }
}

// good
function dogs(x) {
  if (x) {
    if (z) {
      return y;
    }
  } else {
    return z;
  }
}
```

控制语句

- 17.1 如果你的控制语句（`if`，`while` 等）太长或者超过了一行最大长度的限制，则可以将每个条件（或组）放入一个新的行。 逻辑运算符应该在行的开始。

  > 为什么？要求操作符在行的开始保持对齐并遵循类似方法衔接的模式。 这提高了可读性，并且使更复杂的逻辑更容易直观的被理解。

```
// bad
if ((foo === 123 || bar === 'abc') && doesItLookGoodWhenItBecomesT
hatLong() && isThisReallyHappening()) {
```

```
  thing1();
}

// bad
if (foo === 123 &&
  bar === 'abc') {
  thing1();
}

// bad
if (foo === 123
  && bar === 'abc') {
  thing1();
}

// bad
if (
  foo === 123 &&
  bar === 'abc'
) {
  thing1();
}

// good
if (
  foo === 123
  && bar === 'abc'
) {
  thing1();
}

// good
if (
  (foo === 123 || bar === 'abc')
  && doesItLookGoodWhenItBecomesThatLong()
  && isThisReallyHappening()
) {
  thing1();
}

```

```
// good
if (foo === 123 && bar === 'abc') {
  thing1();
}
```

- 17.2 不要使用选择操作符代替控制语句。

```
// bad
!isRunning && startRunning();

// good
if (!isRunning) {
  startRunning();
}
```

注释

- 18.1 使用 `/** ... */` 来进行多行注释。

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...

  return element;
}

```

```
// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}
```

- 18.2 使用 `//` 进行单行注释。 将单行注释放在需要注释的行的上方新行。 在注释之前放一个空行，除非它在块的第一行。

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this.type || 'no type';
```

```
23
24   return type;
25 }
26
27 // also good
28 function getType() {
29   // set the default type to 'no type'
30   const type = this.type || 'no type';
31
32   return type;
33 }
```

- 18.3 用一个空格开始所有的注释，使它更容易阅读。 eslint: `spaced-comment`

```
1  // bad
2  //is current tab
3  const active = true;
4
5  // good
6  // is current tab
7  const active = true;
8
9  // bad
10 /**
11  *make() returns a new element
12  *based on the passed-in tag name
13  */
14 function make(tag) {
15
16   // ...
17
18   return element;
19 }
20
21 // good
22 /**
23  * make() returns a new element
```

```
 * based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}
```

- 18.4 使用 `FIXME` 或者 `TODO` 开始你的注释可以帮助其他开发人员快速了解，如果你提出了一个需要重新审视的问题，或者你对需要实现的问题提出的解决方案。 这些不同于其他评论，因为他们是可操作的。 这些行为是 `FIXME: -- 需要解决这个问题` 或者 `TODO: -- 需要被实现`。

- 18.5 使用 `// FIXME:` 注释一个问题。

```
class Calculator extends Abacus {
  constructor() {
    super();

    // FIXME: 这里不应该使用全局变量
    total = 0;
  }
}
```

- 18.6 使用 `// TODO:` 注释解决问题的方法。

```
class Calculator extends Abacus {
  constructor() {
    super();

    // TODO: total 应该由一个 param 的选项配置
    this.total = 0;
  }
```

```
8 }
```

空白

- 19.1 使用 tabs (空格字符) 设置为 2 个空格。 eslint: `indent`

```
1 // bad
2 function foo() {
3 ····let name;
4 }
5
6 // bad
7 function bar() {
8 ·let name;
9 }
10
11 // good
12 function baz() {
13 ··let name;
14 }
```

- 19.2 在主体前放置一个空格。 eslint: `space-before-blocks`

```
1 // bad
2 function test(){
3   console.log('test');
4 }
5
6 // good
7 function test() {
8   console.log('test');
9 }
```

```
10
11 // bad
12 dog.set('attr',{
13   age: '1 year',
14   breed: 'Bernese Mountain Dog',
15 });
16
17 // good
18 dog.set('attr', {
19   age: '1 year',
20   breed: 'Bernese Mountain Dog',
21 });
```

- 19.3 在控制语句（`if`, `while` 等）开始括号之前放置一个空格。 在函数调用和是声明中，在参数列表和函数名之间没有空格。 eslint: `keyword-spacing`

```
1 // bad
2 if(isJedi) {
3   fight ();
4 }
5
6 // good
7 if (isJedi) {
8   fight();
9 }
10
11 // bad
12 function fight () {
13   console.log ('Swooosh!');
14 }
15
16 // good
17 function fight() {
18   console.log('Swooosh!');
19 }
```

- 19.4 用空格分离操作符。 eslint: `space-infix-ops`

```
// bad
const x=y+5;

// good
const x = y + 5;
```

- 19.5 使用单个换行符结束文件。 eslint: `eol-last`

```
// bad
import { es6 } from './AirbnbStyleGuide';
  // ...
export default es6;
```

```
// bad
import { es6 } from './AirbnbStyleGuide';
  // ...
export default es6;↵
↵
```

```
// good
import { es6 } from './AirbnbStyleGuide';
  // ...
export default es6;↵
```

- 19.6 在使用链式方法调用的时候使用缩进(超过两个方法链)。 使用一个引导点，强调该行是方法调用，而不是新的语句。 eslint: `newline-per-chained-call` `no-whitespace-before-property`

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// bad
$('#items').
  find('.selected').
    highlight().
    end().
  find('.open').
    updateCount();

// good
$('#items')
  .find('.selected')
    .highlight()
    .end()
  .find('.open')
    .updateCount();

// bad
const leds = stage.selectAll('.led').data(data).enter().append('svg:svg').classed('led', true)
    .attr('width', (radius + margin) * 2).append('svg:g')
    .attr('transform', `translate(${radius + margin},${radius + margin})`)
    .call(tron.led);

// good
const leds = stage.selectAll('.led')
    .data(data)
  .enter().append('svg:svg')
    .classed('led', true)
    .attr('width', (radius + margin) * 2)
  .append('svg:g')
    .attr('transform', `translate(${radius + margin},${radius + margin})`)
    .call(tron.led);

```

```
// good
const leds = stage.selectAll('.led').data(data);
```

- 19.7 在块和下一个语句之前留下一空白行。

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;

// bad
const obj = {
  foo() {
  },
  bar() {
  },
};
return obj;

// good
const obj = {
  foo() {
  },

  bar() {
  },
};

return obj;
```

```
33
34 // bad
35 const arr = [
36   function foo() {
37   },
38   function bar() {
39   },
40 ];
41 return arr;
42
43 // good
44 const arr = [
45   function foo() {
46   },
47
48   function bar() {
49   },
50 ];
51
52 return arr;
```

- 19.8 不要在块的开头使用空白行。 eslint: `padded-blocks`

```
1 // bad
2 function bar() {
3
4   console.log(foo);
5
6 }
7
8 // bad
9 if (baz) {
10
11   console.log(qux);
12 } else {
13   console.log(foo);
14
```

```
}

// bad
class Foo {

  constructor(bar) {
    this.bar = bar;
  }
}

// good
function bar() {
  console.log(foo);
}

// good
if (baz) {
  console.log(qux);
} else {
  console.log(foo);
}
```

- 19.9 不要在括号内添加空格。 eslint: `space-in-parens`

```
// bad
function bar( foo ) {
  return foo;
}

// good
function bar(foo) {
  return foo;
}

// bad
if ( foo ) {
  console.log(foo);
```

```
}

// good
if (foo) {
  console.log(foo);
}
```

- 19.10 不要在中括号中添加空格。 eslint: `array-bracket-spacing`

```
// bad
const foo = [ 1, 2, 3 ];
console.log(foo[ 0 ]);

// good
const foo = [1, 2, 3];
console.log(foo[0]);
```

- 19.11 在花括号内添加空格。 eslint: `object-curly-spacing`

```
// bad
const foo = {clark: 'kent'};

// good
const foo = { clark: 'kent' };
```

- 19.12 避免让你的代码行超过100个字符（包括空格）。 注意：根据上边的 约束，长字符串可免除此规定，不应分解。 eslint: `max-len`

> 为什么？这样能够确保可读性和可维护性。

```
// bad
const foo = jsonData && jsonData.foo && jsonData.foo.bar && jsonDa
```

```
  ta.foo.bar.baz && jsonData.foo.bar.baz.quux && jsonData.foo.bar.ba
  z.quux.xyzzy;

// bad
$.ajax({ method: 'POST', url: 'https://airbnb.com/', data: { name:
  'John' } }).done(() => console.log('Congratulations!')).fail(() =>
  console.log('You have failed this city.'));

// good
const foo = jsonData
  && jsonData.foo
  && jsonData.foo.bar
  && jsonData.foo.bar.baz
  && jsonData.foo.bar.baz.quux
  && jsonData.foo.bar.baz.quux.xyzzy;

// good
$.ajax({
  method: 'POST',
  url: 'https://airbnb.com/',
  data: { name: 'John' },
})
  .done(() => console.log('Congratulations!'))
  .fail(() => console.log('You have failed this city.'));
```

- 19.13 要求打开的块标志和同一行上的标志拥有一致的间距。此规则还会在同一行关闭的块标记和前边的标记强制实施一致的间距。 eslint: `block-spacing`

```
// bad
function foo() {return true;}
if (foo) { bar = 0;}

// good
function foo() { return true; }
if (foo) { bar = 0; }
```

- 19.14 逗号之前避免使用空格，逗号之后需要使用空格。eslint: `comma-spacing`

```
// bad
var foo = 1,bar = 2;
var arr = [1 , 2];

// good
var foo = 1, bar = 2;
var arr = [1, 2];
```

- 19.15 在计算属性之间强化间距。eslint: `computed-property-spacing`

```
// bad
obj[foo ]
obj[ 'foo']
var x = {[ b ]: a}
obj[foo[ bar ]]

// good
obj[foo]
obj['foo']
var x = { [b]: a }
obj[foo[bar]]
```

- 19.16 在函数和它的调用之间强化间距。 eslint: `func-call-spacing`

```
// bad
func ();

func
();

```

```
// good
func();
```

- 19.17 在对象的属性和值之间强化间距。 eslint: `key-spacing`

```
// bad
var obj = { "foo" : 42 };
var obj2 = { "foo":42 };

// good
var obj = { "foo": 42 };
```

- 19.18 在行的末尾避免使用空格。 eslint: `no-trailing-spaces`

- 19.19 避免多个空行，并且只允许在文件末尾添加一个换行符。 eslint: `no-multiple-empty-lines`

```
// bad
var x = 1;



var y = 2;

// good
var x = 1;

var y = 2;
```

逗号

- 20.1 逗号前置： **不行** eslint: `comma-style`

```
// bad
const story = [
    once
  , upon
  , aTime
];

// good
const story = [
  once,
  upon,
  aTime,
];

// bad
const hero = {
    firstName: 'Ada'
  , lastName: 'Lovelace'
  , birthYear: 1815
  , superPower: 'computers'
};

// good
const hero = {
  firstName: 'Ada',
  lastName: 'Lovelace',
  birthYear: 1815,
  superPower: 'computers',
};
```

- 20.2 添加尾随逗号： **可以** eslint: `comma-dangle`

> 为什么？这个将造成更清洁的 git 扩展差异。 另外，像 Babel 这样的编译器，会在转换后的代码中删除额外的尾随逗号，这意味着你不必担心在浏览器中后面的 尾随逗号问题 。

```
// bad - 没有尾随逗号的 git 差异
const hero = {
     firstName: 'Florence',
-    lastName: 'Nightingale'
+    lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing']
};

// good - 有尾随逗号的 git 差异
const hero = {
     firstName: 'Florence',
     lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing'],
};
```

```
// bad
const hero = {
  firstName: 'Dana',
  lastName: 'Scully'
};

const heroes = [
  'Batman',
  'Superman'
];

// good
const hero = {
  firstName: 'Dana',
  lastName: 'Scully',
};

const heroes = [
  'Batman',
  'Superman',
```

```
];

// bad
function createHero(
  firstName,
  lastName,
  inventorOf
) {
  // does nothing
}

// good
function createHero(
  firstName,
  lastName,
  inventorOf,
) {
  // does nothing
}

// good (注意逗号不能出现在 "rest" 元素后边)
function createHero(
  firstName,
  lastName,
  inventorOf,
  ...heroArgs
) {
  // does nothing
}

// bad
createHero(
  firstName,
  lastName,
  inventorOf
);

// good
createHero(
  firstName,
```

```
  lastName,
  inventorOf,
);

// good (注意逗号不能出现在 "rest" 元素后边)
createHero(
  firstName,
  lastName,
  inventorOf,
  ...heroArgs
);
```

分号

- 21.1 对 eslint: `semi`

> 为什么？当 JavaScript 遇见一个没有分号的换行符时，它会使用一个叫做 Automatic Semicolon Insertion 的规则来确定是否应该以换行符视为语句的结束，并且如果认为如此，会在代码中断前插入一个分号到代码中。 但是，ASI 包含了一些奇怪的行为，如果 JavaScript 错误的解释了你的换行符，你的代码将会中断。 随着新特性成为 JavaScript 的一部分，这些规则将变得更加复杂。 明确地终止你的语句，并配置你的 linter 以捕获缺少的分号将有助于防止你遇到的问题。

```
// bad - 可能异常
const luke = {}
const leia = {}
[luke, leia].forEach(jedi => jedi.father = 'vader')

// bad - 可能异常
const reaction = "No! That's impossible!"
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}())

// bad - 返回 `undefined` 而不是下一行的值 - 当 `return` 单独一行的时候 ASI 总是会发生
```

```
function foo() {
  return
    'search your feelings, you know it to be foo'
}

// good
const luke = {};
const leia = {};
[luke, leia].forEach((jedi) => {
  jedi.father = 'vader';
});

// good
const reaction = "No! That's impossible!";
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
}());

// good
function foo() {
  return 'search your feelings, you know it to be foo';
}
```

更多信息.

类型转换和强制类型转换

- 22.1 在语句开始前进行类型转换。

- 22.2 字符类型： eslint: `no-new-wrappers`

```
// => this.reviewScore = 9;

```

```
// bad
const totalScore = new String(this.reviewScore); // typeof totalScore is "object" not "string"

// bad
const totalScore = this.reviewScore + ''; // invokes this.reviewScore.valueOf()

// bad
const totalScore = this.reviewScore.toString(); // isn't guaranteed to return a string

// good
const totalScore = String(this.reviewScore);
```

- 22.3 数字类型：使用 `Number` 进行类型铸造和 `parseInt` 总是通过一个基数来解析一个字符串。eslint: `radix` `no-new-wrappers`

```
const inputValue = '4';

// bad
const val = new Number(inputValue);

// bad
const val = +inputValue;

// bad
const val = inputValue >> 0;

// bad
const val = parseInt(inputValue);

// good
const val = Number(inputValue);

// good
const val = parseInt(inputValue, 10);
```

- 22.4 如果出于某种原因，你正在做一些疯狂的事情，而 `parseInt` 是你的瓶颈，并且出于 性能问题 需要使用位运算， 请写下注释，说明为什么这样做和你做了什么。

```
// good
/**
 * parseInt 使我的代码变慢。
 * 位运算将一个字符串转换成数字更快。
 */
const val = inputValue >> 0;
```

- 22.5 **注意：** 当你使用位运算的时候要小心。 数字总是被以 64-bit 值 的形式表示，但是位运算总是返回一个 32-bit 的整数 (来源)。 对于大于 32 位的整数值，位运算可能会导致意外行为。讨论。 最大的 32 位整数是： 2,147,483,647。

```
2147483647 >> 0; // => 2147483647
2147483648 >> 0; // => -2147483648
2147483649 >> 0; // => -2147483647
```

- 22.6 布尔类型： eslint: `no-new-wrappers`

```
const age = 0;

// bad
const hasAge = new Boolean(age);

// good
const hasAge = Boolean(age);

// best
const hasAge = !!age;
```

命名规范

- 23.1 避免单字母的名字。用你的命名来描述功能。 eslint: `id-length`

```
// bad
function q() {
  // ...
}

// good
function query() {
  // ...
}
```

- 23.2 在命名对象、函数和实例时使用驼峰命名法（camelCase）。 eslint: `camelcase`

```
// bad
const OBJEcttssssss = {};
const this_is_my_object = {};
function c() {}

// good
const thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 23.3 只有在命名构造器或者类的时候才用帕斯卡拼命名法（PascalCase）。 eslint: `new-cap`

```
// bad
function user(options) {
  this.name = options.name;
```

```
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}

const good = new User({
  name: 'yup',
});
```

- 23.4 不要使用前置或者后置下划线。 eslint: `no-underscore-dangle`

> 为什么? JavaScript 在属性和方法方面没有隐私设置。 虽然前置的下划线是一种常见的惯例，意思是“private”，事实上，这些属性时公开的，因此，它们也是你公共 API 的一部分。 这种约定可能导致开发人员错误的认为更改不会被视为中断，或者不需要测试。建议：如果你想要什么东西是“private”， 那就一定不能有明显的表现。

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';
this._firstName = 'Panda';

// good
this.firstName = 'Panda';

// 好，在 WeakMapx 可用的环境中
// see https://kangax.github.io/compat-table/es6/#test-WeakMap
const firstNames = new WeakMap();
firstNames.set(this, 'Panda');
```

- 23.5 不要保存 `this` 的引用。 使用箭头函数或者 函数#bind。

```
// bad
function foo() {
  const self = this;
  return function () {
    console.log(self);
  };
}

// bad
function foo() {
  const that = this;
  return function () {
    console.log(that);
  };
}

// good
function foo() {
  return () => {
    console.log(this);
  };
}
```

- 23.6 文件名应该和默认导出的名称完全匹配。

```
// file 1 contents
class CheckBox {
  // ...
}
export default CheckBox;

// file 2 contents
```

```
export default function fortyTwo() { return 42; }

// file 3 contents
export default function insideDirectory() {}

// in some other file
// bad
import CheckBox from './checkBox'; // PascalCase import/export, camelCase filename
import FortyTwo from './FortyTwo'; // PascalCase import/filename, camelCase export
import InsideDirectory from './InsideDirectory'; // PascalCase import/filename, camelCase export

// bad
import CheckBox from './check_box'; // PascalCase import/export, snake_case filename
import forty_two from './forty_two'; // snake_case import/filename, camelCase export
import inside_directory from './inside_directory'; // snake_case import, camelCase export
import index from './inside_directory/index'; // requiring the index file explicitly
import insideDirectory from './insideDirectory/index'; // requiring the index file explicitly

// good
import CheckBox from './CheckBox'; // PascalCase export/import/filename
import fortyTwo from './fortyTwo'; // camelCase export/import/filename
import insideDirectory from './insideDirectory'; // camelCase export/import/directory name/implicit "index"
// ^ supports both insideDirectory.js and insideDirectory/index.js
```

- 23.7 当你导出默认函数时使用驼峰命名法。 你的文件名应该和方法名相同。

```
function makeStyleGuide() {
  // ...
}

export default makeStyleGuide;
```

- 23.8 当你导出一个构造器 / 类 / 单例 / 函数库 / 暴露的对象时应该使用帕斯卡命名法。

```
const AirbnbStyleGuide = {
  es6: {
  },
};

export default AirbnbStyleGuide;
```

- 23.9 缩略词和缩写都必须是全部大写或者全部小写。

> 为什么？名字是为了可读性，不是为了满足计算机算法。

```
// bad
import SmsContainer from './containers/SmsContainer';

// bad
const HttpRequests = [
  // ...
];

// good
import SMSContainer from './containers/SMSContainer';

// good
const HTTPRequests = [
  // ...
];

```

```
// also good
const httpRequests = [
  // ...
];

// best
import TextMessageContainer from './containers/TextMessageContainer';

// best
const requests = [
  // ...
];
```

- 23.10 你可以大写一个常量，如果它：（1）被导出，（2）使用 `const` 定义（不能被重新赋值），（3）程序员可以信任它（以及其嵌套的属性）是不变的。

  > 为什么？这是一个可以帮助程序员确定变量是否会发生变化的辅助工具。
  > UPPERCASE_VARIABLES 可以让程序员知道他们可以相信变量（及其属性）不会改变。

  - 是否是对所有的 `const` 定义的变量？ – 这个是没有必要的，不应该在文件中使用大写。但是，它应该用于导出常量。
  - 导出对象呢？ – 在顶级导出属性 (e.g. `EXPORTED_OBJECT.key`) 并且保持所有嵌套属性不变。

```
// bad
const PRIVATE_VARIABLE = 'should not be unnecessarily uppercased within a file';

// bad
export const THING_TO_BE_CHANGED = 'should obviously not be uppercased';

// bad
export let REASSIGNABLE_VARIABLE = 'do not use let with uppercase variables';

// ---

```

```
// 允许，但是不提供语义值
export const apiKey = 'SOMEKEY';

// 多数情况下，很好
export const API_KEY = 'SOMEKEY';

// ---

// bad - 不必要大写 key 没有增加语义值
export const MAPPING = {
  KEY: 'value'
};

// good
export const MAPPING = {
  key: 'value'
};
```

存取器

- 24.1 对于属性的的存取函数不是必须的。

- 24.2 不要使用 JavaScript 的 getters/setters 方法，因为它们会导致意外的副作用，并且更加难以测试、维护和推敲。 相应的，如果你需要存取函数的时候使用 `getVal()` 和 `setVal('hello')`。

```
// bad
class Dragon {
  get age() {
    // ...
  }

  set age(value) {
    // ...
```

```
  }
}

// good
class Dragon {
  getAge() {
    // ...
  }

  setAge(value) {
    // ...
  }
}
```

- 24.3 如果属性/方法是一个 `boolean` 值，使用 `isVal()` 或者 `hasVal()`。

```
// bad
if (!dragon.age()) {
  return false;
}

// good
if (!dragon.hasAge()) {
  return false;
}
```

- 24.4 可以创建 `get()` 和 `set()` 方法，但是要保证一致性。

```
class Jedi {
  constructor(options = {}) {
    const lightsaber = options.lightsaber || 'blue';
    this.set('lightsaber', lightsaber);
  }

```

```
  set(key, val) {
    this[key] = val;
  }

  get(key) {
    return this[key];
  }
}
```

事件

- 25.1 当给事件（无论是 DOM 事件还是更加私有的事件）附加数据时，传入一个对象（通畅也叫做“hash”） 而不是原始值。 这样可以让后边的贡献者向事件数据添加更多的数据，而不用找出更新事件的每个处理器。 例如，不好的写法：

```
// bad
$(this).trigger('listingUpdated', listing.id);

// ...

$(this).on('listingUpdated', (e, listingID) => {
  // do something with listingID
});
```

更好的写法：

```
// good
$(this).trigger('listingUpdated', { listingID: listing.id });

// ...

$(this).on('listingUpdated', (e, data) => {
  // do something with data.listingID
```

```
});
```

jQuery

- 26.1 对于 jQuery 对象的变量使用 `$` 作为前缀。

```
// bad
const sidebar = $('.sidebar');

// good
const $sidebar = $('.sidebar');

// good
const $sidebarBtn = $('.sidebar-btn');
```

- 26.2 缓存 jQuery 查询。

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...

  $('.sidebar').css({
    'background-color': 'pink',
  });
}

// good
function setSidebar() {
  const $sidebar = $('.sidebar');
  $sidebar.hide();
```

```
16
17   // ...
18
19   $sidebar.css({
20     'background-color': 'pink',
21   });
22 }
```

- 26.3 对于 DOM 查询使用层叠 `$('.sidebar ul')` 或 父元素 > 子元素 `$('.sidebar > ul')` 的格式。 jsPerf

- 26.4 对于有作用域的 jQuery 对象查询使用 `find` 。

```
1 // bad
2 $('ul', '.sidebar').hide();
3
4 // bad
5 $('.sidebar').find('ul').hide();
6
7 // good
8 $('.sidebar ul').hide();
9
10 // good
11 $('.sidebar > ul').hide();
12
13 // good
14 $sidebar.find('ul').hide();
```

ECMAScript 5 兼容性

- 27.1 参考 Kangax的 ES5 兼容性表格。

ECMAScript 6+ (ES 2015+) Styles

- 28.1 这是一个链接到各种 ES6+ 特性的集合。

1. 箭头函数
2. 类
3. 对象简写
4. 对象简洁
5. 对象计算属性
6. 字符串模板
7. 解构
8. 默认参数
9. Rest
10. 数组展开
11. Let 和 Const
12. 求幂运算符
13. 迭代器和发生器
14. 模块

- 28.2 不要使用尚未达到第3阶段的 TC39 建议。

> 为什么？它们没有最终确定， 并且它们可能会被改变或完全撤回。我们希望使用JavaScript，而建议还不是JavaScript。

标准库

标准库
包含功能已损坏的实用工具，但因为遗留原因而保留。

- 29.1 使用 `Number.isNaN` 代替全局的 `isNaN`.
  eslint: `no-restricted-globals`

> 为什么？全局的 `isNaN` 强制非数字转化为数字，对任何强制转化为 NaN 的东西都返回 true。
>
> 如果需要这种行为，请明确说明。

```
// bad
isNaN('1.2'); // false
isNaN('1.2.3'); // true

// good
Number.isNaN('1.2.3'); // false
Number.isNaN(Number('1.2.3')); // true
```

- 29.2 使用 `Number.isFinite` 代替全局的 `isFinite`.
  eslint: `no-restricted-globals`

> 为什么？全局的 `isFinite` 强制非数字转化为数字，对任何强制转化为有限数字的东西都返回 true。
>
> 如果需要这种行为，请明确说明。

```
// bad
isFinite('2e3'); // true

// good
Number.isFinite('2e3'); // false
Number.isFinite(parseInt('2e3', 10)); // true
```

Testing

- 30.1 **是的.**

```
function foo() {
  return true;
}
```

- 30.2 **没有，但是认真**:
  - 无论你使用那种测试框架，都应该编写测试！
  - 努力写出许多小的纯函数，并尽量减少发生错误的地方。
  - 对于静态方法和 mock 要小心----它们会使你的测试更加脆弱。
  - 我们主要在 Airbnb 上使用 `mocha` 和 `jest` 。 `tape` 也会用在一些小的独立模块上。
  - 100%的测试覆盖率是一个很好的目标，即使它并不总是可行的。
  - 无论何时修复bug，都要编写一个回归测试。在没有回归测试的情况下修复的bug在将来几乎肯定会再次崩溃。

性能

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Are Javascript functions like `map()`, `reduce()`, and `filter()` optimized for traversing arrays?
- Loading...

资源

学习 ES6+

- Latest ECMA spec
- ExploringJS
- ES6 Compatibility Table
- Comprehensive Overview of ES6 Features

读这个

- Standard ECMA-262

工具

- Code Style Linters
  - ESlint – Airbnb Style .eslintrc
  - JSHint – Airbnb Style .jshintrc
- Neutrino preset – neutrino-preset-airbnb-base