Airbnb JavaScript 风格指南() {

JavaScript 最合理的方法 *A mostly reasonable approach to JavaScript*

> **注意**: 这个指南假定你正在使用 Babel， 并且需要你使用或等效的使用 babel-preset-airbnb。同时假定你在你的应用里安装了带有或等效的 airbnb-browser-shims 的 `shims/polyfills`

目录

1.Types

- 1.1 基本类型: 你可以直接获取到基本类型的值
    - `string`

    - number
    - boolean
    - null
    - undefined
    - symbol

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

    - Symbols 不能被正确的 polyfill。 所以在不能原生支持 symbol 类型的环境[浏览器]中，不应该使用 symbol 类型。

- 1.2 复杂类型: 复杂类型赋值是获取到他的引用的值。 相当于传引用

    - object
    - array
    - function

```
const foo = [1, 2];
const bar = foo;
bar[0] = 9;
console.log(foo[0], bar[0]); // => 9, 9
```

back to top

2.References

- 2.1 所有的赋值都用 const，避免使用 var. eslint: prefer-const, no-const-assign

    Why? 因为这个确保你不会改变你的初始值，重复引用会导致 bug 和代码难以理解

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

- 2.2 如果你一定要对参数重新赋值，那就用 let，而不是 var. eslint: no-var

    Why? 因为 let 是块级作用域，而 var 是函数级作用域

```
// bad
var count = 1;
```

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

- 2.3 注意： let、const 都是块级作用域

```
// const 和 let 都只存在于它定义的那个块级作用域
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

back to top

3.Objects

- 3.1 使用字面值创建对象. eslint: `no-new-object`

```
// bad
const item = new Object();
// good
const item = {};
```

- 3.2 当创建一个带有动态属性名的对象时，用计算后属性名

  Why? 这可以使你将定义的所有属性放在对象的一个地方.

```
function getKey(k) {
  return `a key named ${k}`;
}
// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;
// good getKey('enabled')是动态属性名
const obj = {
```

```
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```

- 3.3 用对象方法简写. eslint: `object-shorthand`

```
// bad
const atom = {
  value: 1,
  addValue: function (value) {
    return atom.value + value;
  },
};
// good
const atom = {
  value: 1,
  // 对象的方法
  addValue(value) {
    return atom.value + value;
  },
};
```

- 3.4 用属性值缩写. eslint: `object-shorthand`

    Why? 这样写的更少且更可读

```
const lukeSkywalker = 'Luke Skywalker';
// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};
// good
const obj = {
  lukeSkywalker,
};
```

- 3.5 将你的所有缩写放在对象声明的开始.

    Why? 这样也是为了更方便的知道有哪些属性用了缩写.

```
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 只对那些无效的标示使用引号 `''`. eslint: `quote-props`

  > Why? 通常我们认为这种方式主观上易读。他优化了代码高亮，并且页更容易被许多 JS 引擎压缩。

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

- 3.7 不要直接调用 `Object.prototype` 上的方法，如 `hasOwnProperty`, `propertyIsEnumerable`, `isPrototypeOf`。

  > Why? 在一些有问题的对象上， 这些方法可能会被屏蔽掉 - 如：`{ hasOwnProperty: false }` - 或这是一个空对象 `Object.create(null)`

```
// bad
console.log(object.hasOwnProperty(key));
// good
console.log(Object.prototype.hasOwnProperty.call(object, key));
// best
const has = Object.prototype.hasOwnProperty; // 在模块作用内做一次缓存
```

```
/* or */
import has from 'has'; // https://www.npmjs.com/package/has
// ...
console.log(has.call(object, key));
```

- 3.8 对象浅拷贝时，更推荐使用扩展运算符[就是...运算符]，而不是 Object.assign。获取对象指定的几个属性时，用对象的 rest 解构运算符[也是...运算符]更好。
    - 这一段不太好翻译出来， 大家看下面的例子就懂了。^.^

```
// very bad
const original = { a: 1, b: 2 };
const copy = Object.assign(original, { c: 3 }); // this mutates `original` ?_?
delete copy.a; // so does this
// bad
const original = { a: 1, b: 2 };
const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }
// good es6 扩展运算符 ...
const original = { a: 1, b: 2 };
// 浅拷贝
const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }
// rest 赋值运算符
const { a, ...noA } = copy; // noA => { b: 2, c: 3 }
```

back to top

4.Arrays

- 4.1 用字面量赋值。 eslint: no-array-constructor

```
// bad
const items = new Array();
// good
const items = [];
```

- 4.2 用 Array#push 代替直接向数组中添加一个值。

```
const someStack = [];
// bad
someStack[someStack.length] = 'abracadabra';
// good
someStack.push('abracadabra');
```

- 4.3 用扩展运算符做数组浅拷贝，类似上面的对象浅拷贝

```
// 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 用 `...` 运算符而不是 `Array.from` 来将一个可迭代的对象转换成数组。

```
const foo = document.querySelectorAll('.foo');
// good
const nodes = Array.from(foo);
// best
const nodes = [...foo];
```

- 4.5 用 `Array.from` 去将一个类数组对象转成一个数组。

```
const arrLike = { 0: 'foo', 1: 'bar', 2: 'baz', length: 3 };
// bad
const arr = Array.prototype.slice.call(arrLike);
// good
const arr = Array.from(arrLike);
```

- 4.6 用 `Array.from` 而不是 `...` 运算符去做 map 遍历。 因为这样可以避免创建一个临时数组。

```
// bad
const baz = [...foo].map(bar);
// good
const baz = Array.from(foo, bar);
```

- 4.7 在数组方法的回调函数中使用 return 语句。 如果函数体由一条返回一个表达式的语句组成， 并且这个表达式没有副作用， 这个时候可以忽略 return，详见 8.2. eslint: `array-callback-return`

```
// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
// good 函数只有一个语句
[1, 2, 3].map(x => x + 1);
// bad - 没有返回值， 因为在第一次迭代后 acc 就变成 undefined 了
[[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;
});
```

- 4.8 如果一个数组有很多行，在数组的 [后和] 前断行。 请看下面示例

```
// 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,
];
```

4.9 用 Array.includes 而不是 Array.indexOf

```
const arr = [0,3,5,2,'8',NaN];
// bad
console.log(ary.indexOf(NaN)) //-1
// good
console.log(ary.includes(NaN))//true
```

back to top

5.Destructuring

5.1 用对象的解构赋值来获取和使用对象某个或多个属性值。 eslint: prefer-destructuring

> Why? 解构保存了这些属性的临时值/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 用数组解构.

```
const arr = [1, 2, 3, 4];
// bad
const first = arr[0];
const second = arr[1];
// good
const [first, second] = arr;
```

- 5.3 多个返回值用对象的解构，而不是数据解构。

    Why? 你可以在后期添加新的属性或者变换变量的顺序而不会打破原有的调用

```
// bad
function processInput(input) {
  // 然后就是见证奇迹的时刻
  return [left, right, top, bottom];
}
// 调用者需要想一想返回值的顺序
const [left, __, top] = processInput(input);
// good
function processInput(input) {
  // oops， 奇迹又发生了
  return { left, right, top, bottom };
}
// 调用者只需要选择他想用的值就好了
const { left, top } = processInput(input);
```

back to top

6.Strings

- 6.1 对string用单引号 '' 。 eslint: quotes

```
// bad
const name = "Capt. Janeway";
// bad - 样例应该包含插入文字或换行
const name = `Capt. Janeway`;
// good
const name = 'Capt. Janeway';
```

- 6.2 超过100个字符的字符串不应该用string串联成多行。

    Why? 被折断的字符串工作起来是糟糕的而且使得代码更不易被搜索。

```
// 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 because ' +
  '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 用字符串模板而不是字符串拼接来组织可编程字符串。 eslint: prefer-template template-curly-spacing

    Why? 模板字符串更具可读性、语法简洁、字符串插入参数。

```
// 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 永远不要在字符串中用 eval()，他就是潘多拉盒子。 eslint: no-eval

- 6.5 不要使用不必要的转义字符。eslint: no-useless-escape

    Why? 反斜线可读性差，所以他们只在必须使用时才出现哦

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

```
//best
const foo = `my name is '${name}'`;
```

back to top

7.Functions

- 7.1 用命名函数表达式而不是函数声明。eslint: func-style

  函数表达式： const func = function () {}

  函数声明： function func() {}

  Why? 函数声明时作用域被提前了，这意味着在一个文件里函数很容易（太容易了）在其定义之前被引用。这样伤害了代码可读性和可维护性。如果你发现一个函数又大又复杂，这个函数妨碍这个文件其他部分的理解性，这可能就是时候把这个函数单独抽成一个模块了。别忘了给表达式显示的命名，不用管这个名字是不是由一个确定的变量推断出来的，这消除了由匿名函数在错误调用栈产生的所有假设，这在现代浏览器和类似 babel 编译器中很常见 (Discussion)

  Why? 这一段还不理解这种错误发生的场景，所以只能直译过来了， 另附原文 Why? Function declarations are hoisted, which means that it’s easy - too easy - to reference the function before it is defined in the file. This harms readability and maintainability. If you find that a function’s definition is large or complex enough that it is interfering with understanding the rest of the file, then perhaps it’s time to extract it to its own module! Don’t forget to explicitly name the expression, regardless of whether or not the name is inferred from the containing variable (which is often the case in modern browsers or when using compilers such as Babel). This eliminates any assumptions made about the Error’s call stack. (Discussion)

```
// bad
function foo() {
  // ...
}
// bad  *****
** 虽然此规范未推荐此写法，但最终讨论决定采用此写法
// ********
const foo = function () {
  // ...
};
// good
// lexical name distinguished from the variable-referenced invocation(s)
// 函数表达式名和声明的函数名是不一样的
```

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

- 7.2 把立即执行函数包裹在圆括号里。 eslint: wrap-iife

  > Why? immediately invoked function expression = IIFE Why? 一个立即调用的函数表达式是一个单元 - 把它和他的调用者（圆括号）包裹起来，在括号中可以清晰的地表达这些。 Why? 注意：在模块化世界里，你几乎用不着 IIFE

```
// immediately-invoked function expression (IIFE)
(function () {
  console.log('Welcome to the Internet. Please follow me.');
}());
```

- 7.3 不要在非函数块（if、while 等等）内声明函数。把这个函数分配给一个变量。浏览器会允许你这样做，但浏览器解析方式不同，这是一个坏消息。【详见 no-loop-func】 eslint: no-loop-func

- 7.4 **Note:** 在 ECMA-262 中 [块 block] 的定义是： 一系列的语句； 但是函数声明不是一个语句。函数表达式是一个语句。

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

- 7.5 不要用 arguments 命名参数。他的优先级高于每个函数作用域自带的 arguments 对象， 这会导致函数自带的 arguments 值被覆盖

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

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

  Why? ...明确你想用那个参数。而且 rest 参数是真数组，而不是类似数组的 arguments

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

- 7.7 用默认参数语法而不是在函数里对参数重新赋值。

```
// really bad
function handleThings(opts) {
  // 不， 我们不该改 arguments
  // 第二： 如果 opts 的值为 false, 它会被赋值为 {}
  // 虽然你想这么写， 但是这个会带来一些细微的 bug
  opts = opts || {};
  // ...
}
// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}
// good
function handleThings(opts = {}) {
  // ...
}
```

- 7.8 默认参数避免副作用

  Why? 他会令人迷惑不解， 比如下面这个， a 到底等于几， 这个需要想一下。

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

```
count();  // 2
count(3); // 3
count();  // 3
```

- 7.9 把默认参数赋值放在最后

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

- 7.10 函数签名部分要有空格。eslint: space-before-function-paren space-before-blocks

  Why? 统一性好，而且在你添加/删除一个名字的时候不需要添加/删除空格

```
// bad
const f = function(){};
const g = function (){};
const h = function() {};
// good
const x = function () {};
const y = function a() {};
```

- 7.11 不要改参数. eslint: no-param-reassign

  Why? 操作参数对象对原始调用者会导致意想不到的副作用。 就是不要改参数的数据结构，保留参数原始值和数据结构。

```
// bad
function f1(obj) {
  obj.key = 1;
};
// good
function f2(obj) {
  const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? obj.key : 1;
};
```

- 7.12 不要对参数重新赋值。 eslint: no-param-reassign

  Why? 参数重新赋值会导致意外行为，尤其是对 arguments。这也会导致优化问题，特别是在 V8 里

```
// 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) {
  // ...
}
```

- 7.13 用 spread 操作符...去调用多变的函数更好。 eslint: prefer-spread

  Why? 这样更清晰，你不必提供上下文，而且你不能轻易地用 apply 来组成 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]);
```

- 7.14 调用或者书写一个包含多个参数的函数应该像这个指南里的其他多行代码写法一样： 每行值包含一个参数，每行逗号结尾。

```
// bad
function foo(bar,
             baz,
             quux) {
  // ...
}
// good 缩进不要太过分
function foo(
  bar,
  baz,
  quux,
```

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

back to top

8.Arrow Functions

- 8.1 当你一定要用函数表达式（在回调函数里）的时候就用箭头表达式吧。 eslint: prefer-arrow-callback, arrow-spacing

  Why? 他创建了一个 this 的当前执行上下文的函数的版本，这通常就是你想要的；而且箭头函数是更简洁的语法

  Why? 什么时候不用箭头函数： 如果你有一个相当复杂的函数，你可能会把这个逻辑移出到他自己的函数声明里。

```
// 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 如果函数体由一个没有副作用的表达式语句组成，删除大括号和 return。否则，继续用大括号和 return 语句。 eslint: arrow-parens, arrow-body-style

  Why? 语法糖，当多个函数链在一起的时候好读

```
// 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
}));
// 表达式有副作用就不要用隐式 return
function foo(callback) {
  const val = callback();
  if (val === true) {
    // Do something if callback returns true
  }
}
let bool = false;
// bad
// 这种情况会 return bool = true, 不好
foo(() => bool = true);
// good
foo(() => {
  bool = true;
});
```

- 8.3 万一表达式涉及多行，把他包裹在圆括号里更可读。

    Why? 这样清晰的显示函数的开始和结束

```
// bad
['get', 'post', 'put'].map(httpMethod => Object.prototype.hasOwnProperty.call(
    httpMagicObjectWithAVeryLongName,
    httpMethod
  )
);
// good
['get', 'post', 'put'].map(httpMethod => (
  Object.prototype.hasOwnProperty.call(
    httpMagicObjectWithAVeryLongName,
    httpMethod
  )
```

```
));
```

- 8.4 如果你的函数只有一个参数并且函数体没有大括号，就删除圆括号。否则，参数总是放在圆括号里。注意： 一直用圆括号也是没问题，只需要配置 “always” option for eslint. eslint: arrow-parens

    Why? 这样少一些混乱， 其实没啥语法上的讲究，就保持一个风格。

```
// 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(=>)和比较操作符（<=, >=）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 : item.smallSize);
// good
const itemHeight = (item) => {
  const { height, largeSize, smallSize } = item;
  return height <= 256 ? largeSize : smallSize;
};
```

- 8.6 在隐式 return 中强制约束函数体的位置， 就写在箭头后面。 eslint: implicit-arrow-

linebreak

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

back to top

9.Classes & Constructors

- 9.1 常用 class，避免直接操作 prototype

  Why? class 语法更简洁更易理解

```
// 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 用 extends 实现继承

  Why? 它是一种内置的方法来继承原型功能而不打破 instanceof

```
// 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 方法可以返回 this 来实现方法链

```
// 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 写一个定制的 toString()方法是可以的，只要保证它是可以正常工作且没有副作用的

```
class Jedi {
  constructor(options = {}) {
    this.name = options.name || 'no name';
  }
  getName() {
    return this.name;
  }
  toString() {
    return `Jedi - ${this.getName()}`;
  }
}
```

- 9.5 如果没有具体说明，类有默认的构造方法。一个空的构造函数或只是代表父类的构造函数是不需要写的。
  eslint: no-useless-constructor

```
// bad
class Jedi {
  constructor() {}
  getName() {
    return this.name;
  }
}
// bad
class Rey extends Jedi {
  // 这种构造函数是不需要写的
  constructor(...args) {
    super(...args);
  }
}
// good
class Rey extends Jedi {
  constructor(...args) {
    super(...args);
    this.name = 'Rey';
  }
}
```

- 9.6 避免重复类成员。 eslint: no-dupe-class-members

  Why? 重复类成员会默默的执行最后一个 —— 重复本身也是一个 bug

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

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

- 9.7 除非外部库或框架需要使用特定的非静态方法，否则类方法应该使用 this 或被做成静态方法。 作为一个实例方法应该表明它根据接收者的属性有不同的行为。eslint: class-methods-use-this

```
// bad
class Foo {
  bar() {
    console.log('bar');
  }
}
// good - this 被使用了
class Foo {
  bar() {
    console.log(this.bar);
  }
}
// good - constructor 不一定要使用 this
class Foo {
  constructor() {
    // ...
  }
}
// good - 静态方法不需要使用 this
class Foo {
  static bar() {
    console.log('bar');
  }
}
```

back to top

10.Modules

- 10.1 用(import/export) 模块而不是无标准的模块系统。你可以随时转到你喜欢的模块系统。

  Why? 模块化是未来，让我们现在就开启未来吧。

```
// 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 不要用 import 通配符， 就是 * 这种方式

  Why? 这确保你有单个默认的导出

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

- 10.3 不要直接从 import 中直接 export

  Why? 虽然一行是简洁的，有一个明确的方式进口和一个明确的出口方式来保证一致性。

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

- 10.4 一个路径只 import 一次。 eslint: no-duplicate-imports

  Why? 从同一个路径下 import 多行会使代码难以维护

```
// bad
import foo from 'foo';
// … some other imports … //
import { named1, named2 } from 'foo';
// good
import foo, { named1, named2 } from 'foo';
// good
import foo, {
  named1,
  named2,
} from 'foo';
```

- 10.5 不要导出可变的东西 eslint: import/no-mutable-exports

Why? 变化通常都是需要避免，特别是当你要输出可变的绑定。虽然在某些场景下可能需要这种技术，但总的来说应该导出常量。

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

- 10.6 在一个单一导出模块里，用 export default 更好。 eslint: import/prefer-default-export

  Why? 鼓励使用更多文件，每个文件只做一件事情并导出，这样可读性和可维护性更好。

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

- 10.7 import 放在其他所有语句之前。 eslint: import/first

  Why? 让 import 放在最前面防止意外行为。

```
// 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 import 应该缩进，就像多行数组和对象字面量

  Why? 花括号与样式指南中每个其他花括号块遵循相同的缩进规则，逗号也是。

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

- 10.9 在 import 语句里不允许 Webpack loader 语法 eslint: import/no-webpack-loader-syntax

    > Why? 一旦用 Webpack 语法在 import 里会把代码耦合到模块绑定器。最好是在 webpack.config.js 里写 webpack loader 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';
```

back to top

11.Iterators and Generators

- 11.1 不要用遍历器。用 JavaScript 高级函数代替 for-in、 for-of。 eslint: no-iterator no-restricted-syntax

    不要用 for/in 循环数组 数组遍历应该使用 length 属性或者数组的 forEach

    > Why? 这强调了我 们不可变的规则。 处理返回值的纯函数比副作用更容易。

    > Why? 用数组的这些迭代方法： map() / every() / filter() / find() / findIndex() / reduce() / some() / ... , 用对象的这些方法 Object.keys() / Object.values() / Object.entries() 去产生一个数组， 这样你就能去遍历对象了。

```
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 现在不要用 generator

  Why? 它在 es5 上支持的不好

- 11.3 如果你一定要用，或者你忽略我们的建议，请确保它们的函数签名空格是得当的。 eslint: generator-star-spacing

  Why? function 和 * 是同一概念关键字 - *不是 function 的修饰符，function*是一个和 function 不一样的独特结构

```
// 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* () {
  // ...
}
```

back to top

12.Properties

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

```
const luke = {
  jedi: true,
  age: 28,
};
// bad
const isJedi = luke['jedi'];
// good
const isJedi = luke.jedi;
```

- 12.2 当获取的属性是变量时用方括号[]取

```
const luke = {
  jedi: true,
  age: 28,
};
function getProp(prop) {
  return luke[prop];
}
const isJedi = getProp('jedi');
```

- 12.3 做幂运算时用幂操作符 ** 。 eslint: no-restricted-properties.

```
// bad
const binary = Math.pow(2, 10);
```

```
// good
const binary = 2 ** 10;
```

back to top

13.Variables

- 13.1 用 const 或 let 声明变量。不这样做会导致全局变量。 我们想要避免污染全局命名空间。首长这样警告我们。 eslint: no-undef prefer-const

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

- 13.2 每个变量都用一个 const 或 let 。 eslint: one-var

  Why? 这种方式很容易去声明新的变量，你不用去考虑把;调换成,，或者引入一个只有标点的不同的变化。这种做法也可以是你在调试的时候单步每个声明语句，而不是一下跳过所有声明。

```
// 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 const 放一起，let 放一起

  Why? 在你需要分配一个新的变量， 而这个变量依赖之前分配过的变量的时候，这种做法是有帮助的

```
// 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 在你需要的地方声明变量，但是要放在合理的位置

    Why? let 和 const 都是块级作用域而不是函数级作用域

```
// bad - unnecessary function call
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;
  }
  // 在需要的时候分配
  const name = getName();
  if (name === 'test') {
    this.setName('');
    return false;
  }
  return name;
}
```

- 13.5 不要使用链接变量分配。 eslint: no-multi-assign

    Why? 链接变量分配创建隐式全局变量。

```
// bad
(function example() {
  // JavaScript 将这一段解释为
```

```
  // let a = ( b = ( c = 1 ) );
  // let 只对变量 a 起作用; 变量 b 和 c 都变成了全局变量
  let a = b = c = 1;
}());
console.log(a); // undefined
console.log(b); // 1
console.log(c); // 1
// good
(function example() {
  let a = 1;
  let b = a;
  let c = a;
}());
console.log(a); // undefined
console.log(b); // undefined
console.log(c); // undefined
// `const` 也是如此
```

- 13.6 不要使用一元自增自减运算符（++， --）. eslint no-plusplus

  Why? 根据 eslint 文档，一元增量和减量语句受到自动分号插入的影响，并且可能会导致应用程序中的值递增或递减的无声错误。 使用 num + = 1 而不是 num ++或 num ++语句来表达你的值也是更有表现力的。 禁止一元增量和减量语句还会阻止您无意地预增/预减值，这也会导致程序出现意外行为。

```
// bad
const array = [1, 2, 3];
let num = 1;
num++;
--num;
let sum = 0;
let truthyCount = 0;
for (let i = 0; i < array.length; i++) {
  let value = array[i];
  sum += value;
  if (value) {
    truthyCount++;
  }
}
// good
const array = [1, 2, 3];
let num = 1;
num += 1;
num -= 1;
const sum = array.reduce((a, b) => a + b, 0);
```

```
  const truthyCount = array.filter(Boolean).length;
```

- 13.7 在赋值的时候避免在 = 前/后换行。 如果你的赋值语句超出 max-len， 那就用小括号把这个值包起来再换行。 eslint operator-linebreak.

  Why? 在 = 附近换行容易混淆这个赋值语句。

```
// bad
const foo =
  superLongLongLongLongLongLongLongFunctionName();
// bad
const foo
  = 'superLongLongLongLongLongLongLongString';
// good
const foo = (
  superLongLongLongLongLongLongLongFunctionName()
);
// good
const foo = 'superLongLongLongLongLongLongLongString';
```

- 13.8 不允许有未使用的变量。 eslint: no-unused-vars

  Why? 一个声明了但未使用的变量更像是由于重构未完成产生的错误。这种在代码中出现的变量会使阅读者迷惑。

```
// bad
var some_unused_var = 42;
// 写了没用
var y = 10;
y = 5;
// 变量改了自己的值，也没有用这个变量
var z = 0;
z = z + 1;
// 参数定义了但未使用
function getX(x, y) {
    return x;
}
// good
function getXPlusY(x, y) {
  return x + y;
}
var x = 1;
var y = a + 2;
alert(getXPlusY(x, y));
// 'type' 即使没有使用也可以可以被忽略， 因为这个有一个 rest 取值的属性。
// 这是从对象中抽取一个忽略特殊字段的对象的一种形式
```

```
var { type, ...coords } = data;
// 'coords' 现在就是一个没有 'type' 属性的 'data' 对象
```

back to top

14.Hoisting

- 14.1 var 声明会被提前到他的作用域的最前面，它分配的值还没有提前。const 和 let 被赋予了新的调用概念时效区 —— Temporal Dead Zones (TDZ)。 重要的是要知道为什么 typeof 不再安全.

```
// 我们知道这个不会工作，假设没有定义全局的 notDefined
function example() {
  console.log(notDefined); // => throws a ReferenceError
}
// 在你引用的地方之后声明一个变量，他会正常输出是因为变量作用域上升。
// 注意： declaredButNotAssigned 的值没有上升
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}
// 解释器把变量声明提升到作用域最前面，
// 可以重写成如下例子， 二者意义相同
function example() {
  let declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}
// 用 const， let 就不一样了
function example() {
  console.log(declaredButNotAssigned); // => throws a ReferenceError
  console.log(typeof declaredButNotAssigned); // => throws a ReferenceError
  const declaredButNotAssigned = true;
}
```

- 14.2 匿名函数表达式和 var 情况相同

```
function example() {
  console.log(anonymous); // => undefined
  anonymous(); // => TypeError anonymous is not a function
  var anonymous = function () {
    console.log('anonymous function expression');
  };
}
```

- 14.3 已命名函数表达式提升他的变量名，不是函数名或函数体

```
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');
  };
}
// 函数名和变量名一样是也如此
function example() {
  console.log(named); // => undefined
  named(); // => TypeError named is not a function
  var named = function named() {
    console.log('named');
  };
}
```

- 14.4 函数声明则提升了函数名和函数体

```
function example() {
  superPower(); // => Flying
  function superPower() {
    console.log('Flying');
  }
}
```

- 详情请见 JavaScript Scoping & Hoisting by Ben Cherry.

back to top

15.Comparison Operators & Equality

- 15.1 用 === 和 !== 而不是 == 和 !=. eslint: eqeqeq

- 15.2 条件语句如'if'语句使用强制`ToBoolean'抽象方法来评估它们的表达式，并且始终遵循以下简单规则:

    - **Objects** 计算成 **true**
    - **Undefined** 计算成 **false**
    - **Null** 计算成 **false**
    - **Booleans** 计算成 **the value of the boolean**
    - **Numbers**
        - **+0, -0, or NaN** 计算成 **false**
        - 其他 **true**
    - **Strings**
        - '' 计算成 **false**

    - 其他 **true**

```
if ([0] && []) {
  // true
  // 数组（即使是空数组）是对象，对象会计算成 true
}
```

- 15.3 布尔值用缩写，而字符串和数字要明确比较对象

```
// bad
if (isValid === true) {
  // ...
}
// good
if (isValid) {
  // ...
}
// bad
if (name) {
  // ...
}
// good
if (name !== '') {
  // ...
}
// bad
if (collection.length) {
  // ...
}
// good
if (collection.length > 0) {
  // ...
}
```

- 15.4 更多信息请见 Angus Croll 的真理、平等和 JavaScript —— Truth Equality and JavaScript

- 15.5 在 case 和 default 分句里用大括号创建一块包含语法声明的区域(e.g. let, const, function, and class). eslint rules: no-case-declarations.

    Why? 语法声明在整个 switch 的代码块里都可见，但是只有当其被分配后才会初始化，他的初始化时当这个 case 被执行时才产生。 当多个 case 分句试图定义同一个事情时就出问题了

```
// 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 三元表达式不应该嵌套，通常是单行表达式。

  eslint rules: no-nested-ternary.

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

```
// better
const maybeNull = value1 > value2 ? 'baz' : null;
const foo = maybe1 > maybe2
  ? 'bar'
  : maybeNull;
// best
const maybeNull = value1 > value2 ? 'baz' : null;
const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

- 15.7 避免不需要的三元表达式

  eslint rules: 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 用圆括号来混合这些操作符。 只有当标准的算术运算符(+, -, *, & /)， 并且它们的优先级显而易见时，可以不用圆括号括起来。 eslint: no-mixed-operators

  > Why? 这提高了可读性，并且明确了开发者的意图

```
// bad
const foo = a && b < 0 || c > 0 || d + 1 === 0;
// bad
const bar = a ** b - 5 % d;
// bad
// 别人会陷入(a || b) && c 的迷惑中
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;
```

back to top

16.Blocks

- 16.1 用大括号包裹多行代码块。 eslint: nonblock-statement-body-position

```
// bad
if (test)
  return false;
// good
if (test) return false;
// good
if (test) {
  return false;
}
// bad
function foo() { return false; }
// good
function bar() {
  return false;
}
```

- 16.2 if 表达式的 else 和 if 的关闭大括号在一行。 eslint: brace-style

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}
// good
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

- 16.3 如果 if 语句中总是需要用 return 返回， 那后续的 else 就不需要写了。 if 块中包含 return， 它后面的 else if 块中也包含了 return， 这个时候就可以把 return 分到多个 if 语句块中。 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;
  }
}
```

back to top

17.Control Statements

- 17.1 当你的控制语句(if, while 等)太长或者超过最大长度限制的时候， 把每一个(组)判断条件放在单独一行里。 逻辑操作符放在行首。

  > Why? 把逻辑操作符放在行首是让操作符的对齐方式和链式函数保持一致。这提高了可读性，也让复杂逻辑更容易看清楚。

```
// bad
if ((foo === 123 || bar === 'abc') && doesItLookGoodWhenItBecomesThatLong() &&
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 不要用选择操作符代替控制语句。

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

back to top

18.Comments

- 18.1 多行注释用 /** ... */

```
// 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 单行注释用//，将单行注释放在被注释区域上面。如果注释不是在第一行，那么注释前面就空一行

```
// 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';
  return type;
}
// also good
function getType() {
  // set the default type to 'no type'
  const type = this._type || 'no type';
  return type;
}
```

- 18.3 所有注释开头空一个，方便阅读。 eslint: spaced-comment

```
// bad
//is current tab
const active = true;
// good
// is current tab
const active = true;
// bad
/**
 *make() returns a new element
 *based on the passed-in tag name
 */
function make(tag) {
  // ...
  return element;
}
```

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

- 18.4 在你的注释前使用 FIXME'或 TODO'前缀， 这有助于其他开发人员快速理解你指出的需要重新访问的问题， 或者您建议需要实现的问题的解决方案。 这些不同于常规注释，因为它们是可操作的。 动作是 FIXME: - 需要计算出来或 TODO: - 需要实现。

- 18.5 用// FIXME:给问题做注释

```
class Calculator extends Abacus {
  constructor() {
    super();
    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

- 18.6 用// TODO:去注释问题的解决方案

```
class Calculator extends Abacus {
  constructor() {
    super();
    // TODO: total should be configurable by an options param
    this.total = 0;
  }
}
```

back to top

19.Whitespace

- 19.1 tab 用两个空格. eslint: indent

```
// bad
function foo() {
????const name;
}
// bad
function bar() {
```

```
?const name;
}
// good
function baz() {
??const name;
}
```

- 19.2 在大括号前空一格。 eslint: space-before-blocks

```
// bad
function test(){
  console.log('test');
}
// good
function test() {
  console.log('test');
}
// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});
// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog',
});
```

- 19.3 在控制语句(if, while 等)的圆括号前空一格。在函数调用和定义时，参数列表和函数名之间不空格。 eslint: keyword-spacing

```
// bad
if(isJedi) {
  fight ();
}
// good
if (isJedi) {
  fight();
}
// bad
function fight () {
  console.log ('Swooosh!');
}
// good
function fight() {
  console.log('Swooosh!');
```

```
}
```

- 19.4 用空格来隔开运算符。 eslint: space-infix-ops

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

- 19.5 文件结尾空一行. 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 当出现长的方法链（>2 个）时用缩进。用点开头强调该行是一个方法调用，而不是一个新的语句。
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 在一个代码块后下一条语句前空一行。

```
// 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;
// bad
const arr = [
  function foo() {
  },
  function bar() {
  },
];
return arr;
// good
const arr = [
  function foo() {
  },
  function bar() {
  },
];
return arr;
```

- 19.8 不要用空白行填充块。 eslint: padded-blocks

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

- 19.9 不要在代码之间使用多个空白行填充。 eslint: no-multiple-empty-lines

```
// bad
class Person {
```

```
  constructor(fullName, email, birthday) {
    this.fullName = fullName;
    this.email = email;
    this.setAge(birthday);
  }
  setAge(birthday) {
    const today = new Date();
    const age = this.getAge(today, birthday);
    this.age = age;
  }
  getAge(today, birthday) {
    // ..
  }
}
// good
class Person {
  constructor(fullName, email, birthday) {
    this.fullName = fullName;
    this.email = email;
    this.setAge(birthday);
  }
  setAge(birthday) {
    const today = new Date();
    const age = getAge(today, birthday);
    this.age = age;
  }
  getAge(today, birthday) {
    // ..
  }
}
```

- 19.10 圆括号里不要加空格。 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);
}
```

- 19.11 方括号里不要加空格。看示例。 eslint: array-bracket-spacing

```
// bad
const foo = [ 1, 2, 3 ];
console.log(foo[ 0 ]);
// good， 逗号分隔符还是要空格的
const foo = [1, 2, 3];
console.log(foo[0]);
```

- 19.12 花括号里加空格。 eslint: object-curly-spacing

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

- 19.13 避免一行代码超过 100 个字符（包含空格）。

- 注意： 对于上面——strings--line-length，长字符串不受此规则限制，不应分解。 eslint: max-len

    Why? 这样确保可读性和可维护性

```
// bad
const foo = jsonData && jsonData.foo && jsonData.foo.bar &&
jsonData.foo.bar.baz && jsonData.foo.bar.baz.quux &&
jsonData.foo.bar.baz.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.'));
```

- 19.14 作为语句的花括号内也要加空格 —— { 后和 } 前都需要空格。 eslint: block-spacing

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

- 19.15 , 前不要空格， , 后需要空格。 eslint: comma-spacing

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

- 19.16 计算属性内要空格。参考上述花括号和中括号的规则。 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]]
```

- 19.17 调用函数时，函数名和小括号之间不要空格。 eslint: func-call-spacing

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

- 19.18 在对象的字面量属性中， key value 之间要有空格。 eslint: key-spacing

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

- 19.19 行末不要空格。 eslint: no-trailing-spaces

- 19.20 避免出现多个空行。 在文件末尾只允许空一行。 eslint: no-multiple-empty-lines

```
// bad
var x = 1;
var y = 2;
// good
var x = 1;
var y = 2;
```

back to top

20.Commas

- 20.1 不要前置逗号。 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 额外结尾逗号: **要** eslint: comma-dangle

  Why? 这导致 git diffs 更清洁。 此外，像 Babel 这样的转换器会删除转换代码中的额外的逗号，这意味着你不必担心旧版浏览器中的结尾逗号问题。

```
// bad - 没有结尾逗号的 git diff
const hero = {
     firstName: 'Florence',
-    lastName: 'Nightingale'
+    lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing']
};
// good - 有结尾逗号的 git diff
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 (note that a comma must not appear after a "rest" element)
function createHero(
  firstName,
  lastName,
  inventorOf,
  ...heroArgs
) {
  // does nothing
}
// bad
createHero(
  firstName,
  lastName,
  inventorOf
);
// good
createHero(
  firstName,
  lastName,
  inventorOf,
);
// good (note that a comma must not appear after a "rest" element)
createHero(
  firstName,
  lastName,
  inventorOf,
  ...heroArgs
)
```

back to top

21.Semicolons

- 21.1 **Yup.** eslint: semi

  > Why? 当 JavaScript 遇到没有分号结尾的一行，它会执行自动插入分号 Automatic Semicolon Insertion 这一规则来决定行末是否加分号。如果 JavaScript 在你的断行里错误的插入了分号，就会出现一些古怪的行为。当新的功能加到 JavaScript 里后， 这些规则会变得更复杂难懂。显示的结束语句，并通过配置代码检查去捕获没有带分号的地方可以帮助你防止这种错误。

```
// bad
(function () {
  const name = 'Skywalker'
  return name
})()
// good
(function () {
  const name = 'Skywalker';
  return name;
}());
// good, 行首加分号，避免文件被连接到一起时立即执行函数被当做变量来执行。
;(() => {
  const name = 'Skywalker';
  return name;
}());
```

  Read more.

back to top

22.Type Casting & Coercion

- 22.1 在语句开始执行强制类型转换。

- 22.2 Strings: 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(); // 不保证返回 string
// good
const totalScore = String(this.reviewScore);
```

- 22.3 Numbers: 用 Number 做类型转换，parseInt 转换 string 常需要带上基数。 eslint: radix

```
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 请在注释中解释为什么要用移位运算和你在做什么。无论你做什么狂野的事，比如由于 parseInt 是你的性能瓶颈导致你一定要用移位运算。 请说明这个是因为性能原因，

```
// good
/**
 * parseInt 是代码运行慢的原因
 * 用 Bitshifting 将字符串转成数字使代码运行效率大幅增长
 */
const val = inputValue >> 0;
```

- 22.5 **注意：** 用移位运算要小心. 数字使用 64-位表示的，但移位运算常常返回的是 32 为整形 source)。移位运算对大于 32 位的整数会导致意外行为。Discussion. 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:

```
const age = 0;
// bad
const hasAge = new Boolean(age);
// good
const hasAge = Boolean(age);
// best
const hasAge = !!age;
```

back to top

23.Naming Conventions

- 23.1 避免用一个字母命名，让你的命名可描述。 eslint: id-length

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

- 23.2 用小驼峰式命名你的对象、函数、实例。 eslint: camelcase

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

- 23.3 用大驼峰式命名类。 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 不要用前置或后置下划线。 eslint: no-underscore-dangle

  Why? JavaScript 没有私有属性或私有方法的概念。尽管前置下划线通常的概念上意味着“private”，事实上，这些属性是完全公有的，因此这部分也是你的 API 的内容。这一概念可能会导致开发者误以为更改这个不会导致崩溃或者不需要测试。 如果你想要什么东西变成

“private”，那就不要让它在这里出现。

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

- 23.5 不要保存引用 this， 用箭头函数或函数绑定——Function#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 export default 导出模块 A，则这个文件名也叫 A.*， import 时候的参数也叫 A。 大小写完全一致。

```
// 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 export-default 一个函数时，函数名用小驼峰，文件名需要和函数名一致。

```
function makeStyleGuide() {
  // ...
}
export default makeStyleGuide;
```

- 23.8 当你 export 一个结构体/类/单例/函数库/对象 时用大驼峰，文件名采用小驼峰命名方式

```
const AirbnbStyleGuide = {
  es6: {
  }
};
export default AirbnbStyleGuide;
```

- 23.9 简称和缩写应该全部大写或全部小写。

    Why? 名字都是给人读的，不是为了适应电脑的算法的。

```
// 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 你可以用全大写字母设置静态变量，他需要满足三个条件。

    1. 导出变量
    2. 是 const 定义的， 保证不能被改变
    3. 这个变量是可信的，他的子属性都是不能被改变的

       Why? 这是一个附加工具，帮助开发者去辨识一个变量是不是不可变的。

    4. 对于所有的 const 变量呢？ —— 这个是不必要的。大写变量不应该在同一个文件里定义并使用，它只能用来作为导出变量。 赞同!
    5. 那导出的对象呢？ —— 大写变量处在 export 的最高级(e.g. EXPORTED_OBJECT.key) 并且他包含的所有子属性都是不可变的。

```
// 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';
// ---
// 允许但不够语义化
export const apiKey = 'SOMEKEY';
// 在大多数情况下更好
export const API_KEY = 'SOMEKEY';
// ---
```

```
// bad - 不必要的大写键，没有增加任何语言
export const MAPPING = {
  KEY: 'value'
};
// good
export const MAPPING = {
  key: 'value'
};
```

back to top

24.Events

- 24.1 通过哈希而不是原始值向事件装载数据时(不论是 DOM 事件还是像 Backbone 事件的很多属性)。 这使得后续的贡献者（程序员）向这个事件装载更多的数据时不用去找或者更新每个处理器。例如:

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

prefer:

```
// good
$(this).trigger('listingUpdated', { listingID: listing.id });
// ...
$(this).on('listingUpdated', (e, data) => {
  // do something with data.listingID
});
```

back to top

25.Standard Library

标准库中包含一些功能受损但是由于历史原因遗留的工具类

- 25.1 用 Number.isNaN 代替全局的 isNaN. eslint: no-restricted-globals

  > Why? 全局 isNaN 强制把非数字转成数字， 然后对于任何强转后为 NaN 的变量都返回 true 如果你想用这个功能，就显式的用它。

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

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

- 25.2 用 Number.isFinite 代替 isFinite. eslint: no-restricted-globals

  Why? 理由同上，会把一个非数字变量强转成数字，然后做判断。

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

26.Testing

- 26.1 **Yup.**

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

- 26.2 **No, but seriously**:
- 无论用那个测试框架，你都需要写测试。
- 尽量去写很多小而美的纯函数，减少突变的发生
- 小心 stub 和 mock —— 这会让你的测试变得脆弱。
- 在 Airbnb 首选 mocha。 tape 偶尔被用来测试一些小的，独立的模块。
- 100%测试覆盖率是我们努力的目标，即便实际上很少达到。
- 每当你修了一个 bug， 都要写一个回归测试。 一个 bug 修复了，没有回归测试，很可能以后会再次出问题。

back to top