# 前端开发规范

## JavaScript 编码规范

[TOC]

### 前言

为前端开发提供良好的基础编码风格修行指南。

基于 Airbnb编码规范。所有规范分为三个等级：**必须**、**推荐**、**可选**。

**必须（Mandatory）** 级别要求员工必须严格按照规范的编码格式编写，否则会在代码扫描和自动化构建中报错。

**推荐（Preferable）** 级别希望员工尽量按照规范编写，但如有特殊情况，可以不采用。

**可选（Optional）** 级别并不对员工编码提出要求，一般是 JavaScript 常识以及 ES6、ES7 的新语法，但仍希望员工按参考规范编写。委员会将定期 review 编码规范，不断提高规范等级要求。

### 相关工具

#### npm eslint 规则包

安装:

javascript

yarn add eslint eslint-config-airbnb-base -D

*.eslintrc.js:*

module.exports = {  
 extends: ["airbnb-base"],  
};

### 1. 类型

* [1.1](#types--primitives) **【可选】 基本类型**: 当你访问一个基本类型时，直接操作它的值。
  + string
  + number
  + boolean
  + null
  + undefined
  + symbol
* const foo = 1;  
  let bar = foo;  
    
  bar = 9;  
    
  console.log(foo, bar); // => 1, 9
  + 符号（Symbols）不能完全的被 [polyfill](https://en.wikipedia.org/wiki/Polyfill_(programming))，因此在不能原生支持 symbol 类型的浏览器或环境中，不应该使用 symbol 类型。
* [1.2](#types--complex) **【可选】 复杂类型**: 当你访问一个复杂类型时，直接操作其值的引用。
  + object
  + array
  + function
* const foo = [1, 2];  
  const bar = foo;  
    
  bar[0] = 9;  
    
  console.log(foo[0], bar[0]); // => 9, 9

### 2. 引用

* [2.1](#references--prefer-const) **【必须】** 使用 const 定义你的所有引用；避免使用 var。 eslint: prefer-const, no-const-assign
* 原因? 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](#references--disallow-var) **【必须】** 如果你必须重新赋值你的引用， 使用 let 代替 var。 eslint: no-var
* 原因? 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](#references--block-scope) **【可选】** 注意，let 和 const 都是块级作用域。
* // const 和 let 只存在于他们定义的块级作用域中。  
  {  
   let a = 1;  
   const b = 1;  
  }  
  console.log(a); // ReferenceError  
  console.log(b); // ReferenceError

### 3. 对象

* [3.1](#objects--no-new) **【必须】** 使用字面量语法创建对象。 eslint: [no-new-object](https://eslint.org/docs/rules/no-new-object.html)
* // bad  
  const item = new Object();  
    
  // good  
  const item = {};
* [3.2](#es6-computed-properties) **【推荐】** 在创建具有动态属性名称的对象时使用计算属性名。
* 原因? 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;  
    
  // good  
  const obj = {  
   id: 5,  
   name: "San Francisco",  
   [getKey("enabled")]: true,  
  };
* [3.3](#es6-object-shorthand) **【推荐】** 用对象方法简写。 eslint: [object-shorthand](https://eslint.org/docs/rules/object-shorthand.html)
* // bad  
  const value = 1;  
  const atom = {  
   value: value,  
   addValue: function(newValue) {  
   return atom.value + newValue;  
   },  
  };  
    
  // good  
  const value = 1;  
  const atom = {  
   value,  
   addValue(newValue) {  
   return atom.value + newValue;  
   },  
  };
* [3.4](#es6-object-concise) **【推荐】** 用属性值简写。 eslint: [object-shorthand](https://eslint.org/docs/rules/object-shorthand.html)
* 原因? 它更加简洁并更具描述性。
* const lukeSkywalker = "Luke Skywalker";  
    
  // bad  
  const obj = {  
   lukeSkywalker: lukeSkywalker,  
  };  
    
  // good  
  const obj = {  
   lukeSkywalker,  
  };
* [3.5](#objects--grouped-shorthand) **【推荐】** 声明对象时，将简写的属性放在前面。
* 原因? 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](#objects--quoted-props) **【必须】** 只使用引号标注无效标识符的属性。 eslint: [quote-props](https://eslint.org/docs/rules/quote-props.html)
* 原因? 一般来说，我们认为这样更容易阅读。 它能更好地适配语法高亮显示功能，并且更容易通过许多 JS 引擎进行优化。
* // bad  
  const bad = {  
   foo: 3,  
   bar: 4,  
   "data-blah": 5,  
  };  
    
  // good  
  const good = {  
   foo: 3,  
   bar: 4,  
   "data-blah": 5,  
  };
* [3.7](#objects--prototype-builtins) **【推荐】** 不能直接调用 Object.prototype 的方法，如： hasOwnProperty 、 propertyIsEnumerable 和 isPrototypeOf。 eslint: [no-prototype-builtins](https://eslint.org/docs/rules/no-prototype-builtins.html)
* 原因? 这些方法可能被有问题的对象上的属性覆盖 - 如 { 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  
  console.log(has.call(object, key));  
    
  /\* or \*/  
  import has from "has"; // https://www.npmjs.com/package/has  
  console.log(has(object, key));
* [3.8](#objects--rest-spread) **【推荐】** 使用对象扩展操作符（spread operator）浅拷贝对象，而不是用 [Object.assign](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) 方法。 使用对象的剩余操作符（rest operator）来获得一个新对象，该对象省略了某些属性。
* // very bad  
  const original = { a: 1, b: 2 };  
  const copy = Object.assign(original, { c: 3 }); // 变异的 `original` ಠ\_ಠ  
  delete copy.a; // 这....  
    
  // bad  
  const original = { a: 1, b: 2 };  
  const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }  
    
  // good  
  const original = { a: 1, b: 2 };  
  const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }  
    
  const { a, ...noA } = copy; // noA => { b: 2, c: 3 }

### 4. 数组

* [4.1](#arrays--literals) **【必须】** 使用字面量语法创建数组。 eslint: [no-array-constructor](https://eslint.org/docs/rules/no-array-constructor.html)
* // bad  
  const items = new Array();  
    
  // good  
  const items = [];
* [4.2](#arrays--push) **【必须】** 使用 [Array#push](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/push) 代替直接赋值来给数组添加项。
* const someStack = [];  
    
  // bad  
  someStack[someStack.length] = "abracadabra";  
    
  // good  
  someStack.push("abracadabra");
* [4.3](#es6-array-spreads) **【必须】** 使用数组展开符 ... 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](#arrays--from) **【推荐】** 使用展开符 ... 代替 [Array.from](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/from)，将一个可迭代对象转换成一个数组。
* const foo = document.querySelectorAll(".foo");  
    
  // good  
  const nodes = Array.from(foo);  
    
  // best  
  const nodes = [...foo];
* [4.5](#arrays--mapping) **【必须】** 使用 [Array.from](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/from) 将一个类数组（array-like）对象转换成一个数组。
* 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](#arrays--mapping) **【必须】** 使用 [Array.from](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/from) 代替展开符 ... 映射迭代器，因为它避免了创建一个中间数组。
* // bad  
  const baz = [...foo].map(bar);  
    
  // good  
  const baz = Array.from(foo, bar);
* [4.7](#arrays--callback-return) **【推荐】** 在数组回调函数中使用 return 语句。 如果函数体由单个语句的返回表达式组成，并且无副作用，那么可以省略返回值， 具体查看 [8.2](#arrows--implicit-return)。 eslint: [array-callback-return](https://eslint.org/docs/rules/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;  
  });
* [4.8](#arrays--bracket-newline) **【推荐】** 如果数组有多行，则在数组开始括号 [ 的时候换行，然后在数组结束括号 ] 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];

### 5. 解构

* [5.1](#destructuring--object) **【推荐】** 在访问和使用对象的多个属性时使用对象解构。 eslint: [prefer-destructuring](https://eslint.org/docs/rules/prefer-destructuring)
* 原因? 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](#destructuring--array) **【推荐】** 使用数组解构。 eslint: [prefer-destructuring](https://eslint.org/docs/rules/prefer-destructuring)
* const arr = [1, 2, 3, 4];  
    
  // bad  
  const first = arr[0];  
  const second = arr[1];  
    
  // good  
  const [first, second] = arr;
* [5.3](#destructuring--object-over-array) **【必须】** 在有多个返回值时, 使用对象解构，而不是数组解构。
* 原因? 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);

### 6. 字符

* [6.1](#strings--quotes) **【推荐】** 使用单引号 '' 定义字符串。 eslint: [quotes](https://eslint.org/docs/rules/quotes.html)
* // 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](#strings--line-length) **【必须】** 不应该用字符串跨行连接符的格式来跨行编写,这样会使当前行长度超过 100 个字符。
* 原因? 断开的字符串维护起来很痛苦，并且会提高索引难度。
* // 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](#es6-template-literals) **【必须】** 构建字符串时，使用字符串模板代替字符串拼接。 eslint: [prefer-template](https://eslint.org/docs/rules/prefer-template.html) [template-curly-spacing](https://eslint.org/docs/rules/template-curly-spacing)
* 原因? 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](#strings--eval) **【必须】** 永远不要使用 eval() 执行放在字符串中的代码，它导致了太多的漏洞。 eslint: [no-eval](https://eslint.org/docs/rules/no-eval)
* [6.5](#strings--escaping) **【必须】** 不要在字符串中转义不必要的字符。 eslint: [no-useless-escape](https://eslint.org/docs/rules/no-useless-escape)
* 原因? 反斜杠损害了可读性，因此只有在必要的时候才可以出现。
* // bad  
  const foo = "'this' is \"quoted\"";  
    
  // good  
  const foo = "'this' is \"quoted\"";  
  const foo = `my name is '${name}'`;

### 7. 函数

* [7.1](#functions--declarations) **【可选】** 使用命名的函数表达式代替函数声明。 eslint: [func-style](https://eslint.org/docs/rules/func-style)
* 原因? 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，并且它干扰了对这个文件其他部分的理解，那么是时候把这个函数单独抽成一个模块了！别忘了给表达式显式的命名，不用管这个名字是不是由一个确定的变量推断出来的（这在现代浏览器和类似 babel 编译器中很常见）。这消除了由匿名函数在错误调用栈产生的所有假设。 \* ([Discussion](https://github.com/airbnb/javascript/issues/794))
* // bad  
  function foo() {  
   // ...  
  }  
    
  // also good \*  
  const foo = function() {  
   // ...  
  };  
    
  // good  
  const short = function longUniqueMoreDescriptiveLexicalFoo() {  
   // ...  
  };
* [7.2](#functions--iife) **【必须】** 把立即执行函数包裹在圆括号里。 eslint: [wrap-iife](https://eslint.org/docs/rules/wrap-iife.html)
* 原因? 立即调用的函数表达式是个独立的单元 - 将它和它的调用括号还有入参包装在一起可以非常清晰的表明这一点。请注意，在一个到处都是模块的世界中，您几乎用不到 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](#functions--in-blocks) **【必须】** 切记不要在非功能块中声明函数 (if, while, 等)。 请将函数赋值给变量。 浏览器允许你这样做， 但是不同浏览器会有不同的行为， 这并不是什么好事。 eslint: [no-loop-func](https://eslint.org/docs/rules/no-loop-func.html)
* [7.4](#functions--note-on-blocks) **【必须】** ECMA-262 将 block 定义为语句列表。 而函数声明并不是语句。
* // bad  
  if (currentUser) {  
   function test() {  
   console.log("Nope.");  
   }  
  }  
    
  // good  
  let test;  
  if (currentUser) {  
   test = () => {  
   console.log("Yup.");  
   };  
  }
* [7.5](#functions--arguments-shadow) **【必须】** 永远不要给一个参数命名为 arguments。 这将会覆盖函数默认的 arguments 对象。
* // bad  
  function foo(name, options, arguments) {  
   // ...  
  }  
    
  // good  
  function foo(name, options, args) {  
   // ...  
  }
* [7.6](#es6-rest) **【推荐】** 使用 rest 语法 ... 代替 arguments。 eslint: [prefer-rest-params](https://eslint.org/docs/rules/prefer-rest-params)
* 原因? ... 明确了你想要拉取什么参数。 而且, 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](#es6-default-parameters) **【推荐】** 使用默认的参数语法，而不是改变函数参数。
* // really bad  
  function handleThings(opts) {  
   // 不！我们不应该修改参数。  
   // 更加错误的是： 如果 opts 是一个 "非正值"（falsy）它将被设置成一个对象  
   // 这或许正是你想要的，但它可能会导致一些难以察觉的错误。  
   opts = opts || {};  
   // ...  
  }  
    
  // still bad  
  function handleThings(opts) {  
   if (opts === void 0) {  
   opts = {};  
   }  
   // ...  
  }  
    
  // good  
  function handleThings(opts = {}) {  
   // ...  
  }
* [7.8](#functions--default-side-effects) **【必须】** 使用默认参数时避免副作用。
* 原因? 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](#functions--defaults-last) **【推荐】** 总是把默认参数放在最后。
* // bad  
  function handleThings(opts = {}, name) {  
   // ...  
  }  
    
  // good  
  function handleThings(name, opts = {}) {  
   // ...  
  }
* [7.10](#functions--constructor) **【推荐】** 永远不要使用函数构造器来创建一个新函数。 eslint: [no-new-func](https://eslint.org/docs/rules/no-new-func)
* 原因? 以这种方式创建一个函数跟 eval() 差不多，将会导致漏洞。
* // bad  
  var add = new Function("a", "b", "return a + b");  
    
  // still bad  
  var subtract = Function("a", "b", "return a - b");
* [7.11](#functions--signature-spacing) **【必须】** 函数声明语句中需要空格。 eslint: [space-before-function-paren](https://eslint.org/docs/rules/space-before-function-paren) [space-before-blocks](https://eslint.org/docs/rules/space-before-blocks)
* 原因? 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](#functions--mutate-params) **【推荐】** 不要改变入参。 eslint: [no-param-reassign](https://eslint.org/docs/rules/no-param-reassign.html)
* 原因? 操作入参对象会导致原始调用位置出现意想不到的副作用。
* // bad  
  function f1(obj) {  
   obj.key = 1;  
  }  
    
  // good  
  function f2(obj) {  
   const key = Object.prototype.hasOwnProperty.call(obj, "key") ? obj.key : 1;  
  }
* [7.13](#functions--reassign-params) **【推荐】** 不要对入参重新赋值，也不要给入参的属性赋值。部分要求修改入参的常用库（如 Koa、Vuex）可以豁免。 eslint: [no-param-reassign](https://eslint.org/docs/rules/no-param-reassign.html)
* 原因? 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](#functions--spread-vs-apply) **【推荐】** 优先使用扩展运算符 ... 来调用可变参数函数。 eslint: [prefer-spread](https://eslint.org/docs/rules/prefer-spread)
* 原因? 它更加清晰，你不需要提供上下文，并且能比用 apply 来执行可变参数的 new 操作更容易些。
* // 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](#X54ebc4d9475d9b90670d7deca7f90764ea592ee) **【推荐】** 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，并且最后一行也要以逗号结尾。eslint: [function-paren-newline](https://eslint.org/docs/rules/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);

### 8. 箭头函数

* [8.1](#arrows--use-them) **【推荐】** 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](https://eslint.org/docs/rules/prefer-arrow-callback.html), [arrow-spacing](https://eslint.org/docs/rules/arrow-spacing.html)
* 原因? 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。
* 什么时候不适用? 如果你有一个相当复杂的函数，你可能会把这些逻辑转移到它自己的命名函数表达式里。
* // 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](#arrows--implicit-return) **【推荐】** 如果函数体由一个没有副作用的 [表达式](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) 语句组成，删除大括号和 return。否则，保留括号并继续使用 return 语句。 eslint: [arrow-parens](https://eslint.org/docs/rules/arrow-parens.html), [arrow-body-style](https://eslint.org/docs/rules/arrow-body-style.html)
* 原因? 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 + 1}.`);  
    
  // 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  
   }  
  }  
    
  let bool = false;  
    
  // bad  
  foo(() => (bool = true));  
    
  // good  
  foo(() => {  
   bool = true;  
  });
* [8.3](#arrows--paren-wrap) **【推荐】** 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。
* 原因? 它清楚地表明了函数的起点和终点。

// 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](#arrows--one-arg-parens) **【推荐】** 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，为了保证清晰和一致性，请给参数加上括号。 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](https://eslint.org/docs/rules/arrow-parens#always) 来配置 eslint. eslint: [arrow-parens](https://eslint.org/docs/rules/arrow-parens.html)
* 原因？ 让代码看上去不那么乱。
* // 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](#arrows--confusing) **【推荐】** 避免搞混箭头函数符号 (=>) 和比较运算符 (<=, >=)。 eslint: [no-confusing-arrow](https://eslint.org/docs/rules/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](#whitespace--implicit-arrow-linebreak) **【推荐】** 在箭头函数用隐式 return 时强制将函数体的位置约束在箭头后。 eslint: [implicit-arrow-linebreak](https://eslint.org/docs/rules/implicit-arrow-linebreak)
* // bad  
  (foo) => bar;  
    
  (foo) => bar;  
    
  // good  
  (foo) => bar;  
  (foo) => bar;  
  (foo) => bar;

### 9. 类和构造器

* [9.1](#constructors--use-class) **【推荐】** 尽量使用 class. 避免直接操作 prototype .
* 原因？ class 语法更简洁，更容易看懂。
* // 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](#constructors--extends) **【推荐】** 使用 extends 来实现继承。
* 原因？ 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](#constructors--chaining) **【可选】** 类的成员方法，可以返回 this， 来实现链式调用。
* // 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](#constructors--tostring) **【可选】** 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](#constructors--no-useless) **【推荐】** 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](https://eslint.org/docs/rules/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](#classes--no-duplicate-members) **【必须】** 避免定义重复的类成员。 eslint: [no-dupe-class-members](https://eslint.org/docs/rules/no-dupe-class-members)
* 原因？ 重复的类成员声明将会默认使用最后一个 - 具有重复的类成员可以说是一个 bug。
* // bad  
  class Foo {  
   bar() {  
   return 1;  
   }  
   bar() {  
   return 2;  
   }  
  }  
    
  // good  
  class Foo {  
   bar() {  
   return 1;  
   }  
  }
* [9.7] **【推荐】** 类成员要么引用 this ，要么声明为静态方法，除非一个外部库或框架需要使用某些非静态方法。当一个方法为非静态方法时，一般表明它在不同的实例上会表现得不同。

### 10. 模块

* [10.1](#modules--use-them) **【可选】** 使用 ES6 的模块 (import/export) 语法来定义模块。
* // 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](#modules--no-wildcard) **【推荐】** 不要使用 import \* 通配符
* 原因？ 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](#modules--no-export-from-import) **【推荐】** 不要在 import 语句中直接 export。
* 原因？ 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](#import/no-duplicates) **【必须】** 对于同一个路径，只在一个地方引入所有需要的东西。 eslint: [no-duplicates](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/no-duplicates.md)
* 原因？ 对于同一个路径，如果存在多行引入，会使代码更难以维护。
* // 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](#modules--no-mutable-exports) **【推荐】** 不要导出可变的引用。 eslint: [import/no-mutable-exports](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/no-mutable-exports.md)
* 原因？ 在一般情况下，应该避免导出可变引用。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](#modules--prefer-default-export) **【可选】** 在只有单一导出的模块里，用 export default 更好。 eslint: [import/prefer-default-export](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/prefer-default-export.md)
* 原因？ 鼓励更多的模块只做单一导出，会增强代码的可读性和可维护性。
* // bad  
  export function foo() {}  
    
  // good  
  export default function foo() {}
* [10.7](#modules--imports-first) **【必须】** 将所有的 imports 语句放在其他语句之前。 eslint: [import/first](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/first.md)
* 原因？ 将所有的 imports 提到顶部，可以防止某些诡异行为的发生。
* // bad  
  import foo from "foo";  
  foo.init();  
    
  import bar from "bar";  
    
  // good  
  import foo from "foo";  
  import bar from "bar";  
    
  foo.init();
* [10.8](#modules--multiline-imports-over-newlines) **【可选】** 多行引入应该像多行数组和对象字面量一样缩进。
* 原因？ 这里的花括号和其他地方的花括号是一样的，遵循相同的缩进规则。末尾的逗号也是一样的。
* // bad  
  import { longNameA, longNameB, longNameC, longNameD, longNameE } from "path";  
    
  // good  
  import { longNameA, longNameB, longNameC, longNameD, longNameE } from "path";
* [10.9](#modules--no-webpack-loader-syntax) **【推荐】** 在模块导入语句中禁止使用 Webpack 加载器语法。 eslint: [import/no-webpack-loader-syntax](https://github.com/benmosher/eslint-plugin-import/blob/master/docs/rules/no-webpack-loader-syntax.md)
* 原因？ 因为在导入语句中使用 webpack 语法，会将代码和打包工具耦合在一起。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";

### 11. 迭代器和发生器

* [11.1](#iterators--nope) **【推荐】** 不要使用迭代器。 推荐使用 JavaScript 的高阶函数代替 for-in 。 eslint: [no-iterator](https://eslint.org/docs/rules/no-iterator.html) [no-restricted-syntax](https://eslint.org/docs/rules/no-restricted-syntax)
* 原因？ 这有助于不可变性原则。 使用带有返回值的纯函数比使用那些带有副作用的方法，更具有可读性。
* 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](#generators--nope) **【可选】** 现在不要使用 generator。
* 原因？ 它们不能很好的转译为 ES5。但可以在 Nodejs 中使用。\*
* [11.3](#generators--spacing) **【推荐】** 如果你必须要使用 generator，请确保正确使用空格。 eslint: [generator-star-spacing](https://eslint.org/docs/rules/generator-star-spacing)
* 原因？ 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\*() {  
   // ...  
  };

### 12. 属性

* [12.1](#properties--dot) **【推荐】** 访问属性时使用点符号。 eslint: [dot-notation](https://eslint.org/docs/rules/dot-notation.html)
* const luke = {  
   jedi: true,  
   age: 28,  
  };  
    
  // bad  
  const isJedi = luke["jedi"];  
    
  // good  
  const isJedi = luke.jedi;
* [12.2](#properties--bracket) **【可选】** 使用变量访问属性时，用 []表示法。
* const luke = {  
   jedi: true,  
   age: 28,  
  };  
    
  function getProp(prop) {  
   return luke[prop];  
  }  
    
  const isJedi = getProp("jedi");
* [12.3](#Xe061860b53d7a4ef3f880600b6f43efe2c3bc53) **【推荐】** 计算指数时，可以使用 \*\* 运算符。 eslint: [no-restricted-properties](https://eslint.org/docs/rules/no-restricted-properties).
* // bad  
  const binary = Math.pow(2, 10);  
    
  // good  
  const binary = 2 \*\* 10;

### 13. 变量

* [13.1](#variables--const) **【必须】** 变量应先声明再使用，禁止引用任何未声明的变量，除非你明确知道引用的变量存在于当前作用域链上。禁止不带任何关键词定义变量，这样做将会创建一个全局变量，污染全局命名空间，造成程序意料之外的错误。 eslint: [no-undef](https://eslint.org/docs/rules/no-undef) [prefer-const](https://eslint.org/docs/rules/prefer-const)
* // bad, 这会创建一个全局变量  
  superPower = new SuperPower();  
    
  // good  
  const superPower = new SuperPower();  
    
  // bad, 容易污染外部变量  
  let superPower = "a";  
  (function() {  
   superPower = "b";  
  })();  
  console.log(superPower);  
    
  // good  
  let superPower = "a";  
  (function() {  
   let superPower = "b";  
  })();  
  console.log(superPower);  
    
  // bad, 更常见的情况是这样的，在 for 循环里的 i 将会污染外部的变量 i  
  let i = 1;  
  (function() {  
   for (i = 0; i < 10; i++) {  
   console.log("inside", i);  
   }  
    
   console.log("outside", i);  
  })();  
  console.log("global", i);  
    
  // good  
  let i = 1;  
  (function() {  
   // i 的作用域在 for 循环内  
   for (let i = 0; i < 10; i++) {  
   console.log("inside i", i);  
   }  
    
   // 如果真的需要在 for 循环外使用循环变量，应该先定义在外部  
   let j;  
    
   for (j = 0; j < 10; j++) {  
   console.log("inside j:", j);  
   }  
    
   console.log("outside j", j);  
  })();  
  console.log("global", i);
* [13.2](#variables--one-const) **【推荐】** 声明多个变量应该分开声明，避免使用 , 一次声明多个变量。 eslint: [one-var](https://eslint.org/docs/rules/one-var.html)
* 原因？ 这样更容易添加新的变量声明，不必担心是使用 ; 还是使用 , 所带来的代码差异。使用版本管理工具如 git ，最后那行的 ; 就不会被标记为修改成 ,。 并且可以通过 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  
  const items = getItems(),  
   goSportsTeam = true;  
  // 定义成了全局变量  
  dragonball = "z";  
    
  // good  
  const items = getItems();  
  const goSportsTeam = true;  
  const dragonball = "z";
* [13.3](#variables--const-let-group) **【推荐】** 把 const 声明语句放在一起，把 let 声明语句放在一起。
* 原因？ 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，且更容易知道哪些变量是不希望被修改的。
* // 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](#variables--define-where-used) **【推荐】** 在你真正需要使用到变量的代码块内定义变量。
* 原因？ let 和 const 是块级作用域而不是函数作用域，不存在变量提升的情况。
* // 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](#variables--no-chain-assignment) **【必须】** 不要链式变量赋值。 eslint: [no-multi-assign](https://eslint.org/docs/rules/no-multi-assign)
* 原因？ 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);  
  })();  
    
  // throws ReferenceError  
  console.log(a);  
  // 1  
  console.log(b);  
  // 1  
  console.log(c);  
    
  // good  
  (function example() {  
   let a = 1;  
   let b = a;  
   let c = a;  
  })();  
    
  // throws ReferenceError  
  console.log(a);  
  // throws ReferenceError  
  console.log(b);  
  // throws ReferenceError  
  console.log(c);  
    
  // 对于 `const` 也一样
* [13.6](#variables--unary-increment-decrement) **【必须】** 避免使用不必要的递增和递减操作符 (++, --)。 eslint [no-plusplus](https://eslint.org/docs/rules/no-plusplus)
* 原因？ 在 eslint 文档中，一元操作符 ++ 和 --会自动添加分号，不同的空白可能会改变源代码的语义。建议使用 num += 1 这样的语句来做递增和递减，而不是使用 num++ 或 num ++ 。同时 ++num 和 num++ 的差异也使代码的可读性变差。不必要的增量和减量语句会导致无法预先明确递增/预递减值，这可能会导致程序中的意外行为。
* 但目前依然允许在 for loop 中使用 ++、-- 的语法，但依然建议尽快迁移到 += 1、-= 1 的语法。 [#22](https://git.code.oa.com/standards/javascript/issues/22)
* // bad, i = 11, j = 20  
  let i = 10;  
  let j = 20;  
  i++;  
  j;  
    
  // bad, i = 10, j = 21  
  let i = 10;  
  let j = 20;  
  i;  
  ++j;  
    
  // bad  
  const array = [1, 2, 3];  
  let num = 1;  
  num++;  
  --num;  
    
  // not good, just acceptable for upforward compatible.  
  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;  
    
  // good  
  const sum = array.reduce((a, b) => a + b, 0);  
  const truthyCount = array.filter(Boolean).length;
* [13.7](#variables--linebreak) **【必须】** 避免在赋值语句 = 前后换行。如果你的代码单行长度超过了 [max-len](https://eslint.org/docs/rules/max-len.html) 定义的长度而不得不换行，那么使用括号包裹。 eslint [operator-linebreak](https://eslint.org/docs/rules/operator-linebreak.html).
* 原因？ 在 = 前后换行，可能混淆赋的值。
* // bad  
  const foo = "superLongLongLongLongLongLongLongLongString";  
    
  // bad  
  const bar = superLongLongLongLongLongLongLongLongFunctionName();  
    
  // bad  
  const fullHeight = borderTop + innerHeight + borderBottom;  
    
  // bad  
  const anotherHeight = borderTop + innerHeight + borderBottom;  
    
  // bad  
  const thirdHeight = borderTop + innerHeight + borderBottom;  
    
  // good - max-len 会忽略字符串，直接写后面即可。  
  const foo = "superLongLongLongLongLongLongLongLongString";  
    
  // good  
  const bar = superLongLongLongLongLongLongLongLongFunctionName();  
    
  // good  
  const fullHeight = borderTop + innerHeight + borderBottom;  
    
  // good  
  const anotherHeight = borderTop + innerHeight + borderBottom;  
    
  // good  
  const thirdHeight = borderTop + innerHeight + borderBottom;
* [13.8](#variables--no-unused-vars) **【必须】** 禁止定义了变量却不使用它。 eslint: [no-unused-vars](https://eslint.org/docs/rules/no-unused-vars)
* 原因? 在代码里到处定义变量却没有使用它，不完整的代码结构看起来像是个代码错误。即使没有使用，但是定义变量仍然需要消耗资源，并且对阅读代码的人也会造成困惑，不知道这些变量是要做什么的。
* // bad  
  let some\_unused\_var = 42;  
    
  // bad，定义了变量不意味着就是使用了  
  let y = 10;  
  y = 5;  
    
  // bad，对自身的操作并不意味着使用了  
  let z = 0;  
  z = z + 1;  
    
  // bad, 未使用的函数参数  
  function getX(x, y) {  
   return x;  
  }  
    
  // good  
  function getXPlusY(x, y) {  
   return x + y;  
  }  
    
  let x = 1;  
  let y = a + 2;  
    
  alert(getXPlusY(x, y));  
    
  /\*\*  
   \* 有时候我们想要提取某个对象排除了某个属性外的其他属性，会用 rest 参数解构对象  
   \* 这时候 type 虽然未使用，但是仍然被定义和赋值，这也是一种空间的浪费  
   \* type 的值是 'a'  
   \* coords 的值是 data 对象，但是没有 type 属性 { example1: 'b', example2: 'c' }  
   \*/  
  let data = { type: "a", example1: "b", example2: "c" };  
  let { type, ...coords } = data;

### 14. 变量提升

* [14.1](#hoisting--about) **【可选】** var 定义的变量会被提升到函数作用域范围内的最顶部，但是对它的赋值是不会被提升的，因此在函数顶部相当于定义了变量，但是值是 undefined。const 和 let 声明的变量受到一个称之为 “暂时性死区” [(Temporal Dead Zones ，简称 TDZ)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#Temporal_dead_zone) 的新概念保护，因此在 “暂时性死区” 内部的 const 和 let 变量，都需要先声明再使用，否则会报错。详情可以阅读 [typeof 不再安全](https://web.archive.org/web/20200121061528/http://es-discourse.com/t/why-typeof-is-no-longer-safe/15) 这篇文章。
* // notDefined 未定义 (假设没有定义的同名全局变量)  
  function example() {  
   // => throws a ReferenceError  
   console.log(notDefined);  
  }  
    
  /\*\*  
   \* 函数体内部因存在 var 变量声明，因此在引用变量的语句之前，变量提升就已经起作用了  
   \* 注意: 真正的值 `true` 不会被提升。  
   \*/  
  function example() {  
   // => undefined  
   console.log(declaredButNotAssigned);  
   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;  
   // => undefined  
   console.log(declaredButNotAssigned);  
   declaredButNotAssigned = true;  
  }  
    
  // 使用 const 和 let  
  function example() {  
   // => throws a ReferenceError  
   console.log(declaredButNotAssigned);  
   // => throws a ReferenceError  
   console.log(typeof declaredButNotAssigned);  
   const declaredButNotAssigned = true;  
  }
* [14.2](#hoisting--anon-expressions) **【可选】** 匿名函数赋值表达式提升变量名，而不是函数赋值。
* function example() {  
   // => undefined  
   console.log(anonymous);  
    
   // => TypeError anonymous is not a function  
   anonymous();  
    
   var anonymous = function() {  
   console.log("anonymous function expression");  
   };  
  }
* [14.3](#hoisting--named-expressions) **【可选】** 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() {  
   // => undefined  
   console.log(named);  
    
   // => TypeError named is not a function  
   named();  
    
   // => ReferenceError superPower is not defined  
   superPower();  
    
   var named = function superPower() {  
   console.log("Flying");  
   };  
  }  
    
  // 当函数名和变量名相同时也是如此。  
  function example() {  
   // => undefined  
   console.log(named);  
    
   // => TypeError named is not a function  
   named();  
    
   var named = function named() {  
   console.log("named");  
   };  
  }
* [14.4](#hoisting--declarations) **【可选】** 函数声明提升其名称和函数体。
* function example() {  
   // => Flying  
   superPower();  
    
   function superPower() {  
   console.log("Flying");  
   }  
  }
  + 更多信息请参考 [Ben Cherry](http://www.adequatelygood.com/) 的 [JavaScript Scoping & Hoisting](http://www.adequatelygood.com/2010/2/JavaScript-Scoping-and-Hoisting/)。

### 15. 比较运算符和等号

* [15.1](#comparison--eqeqeq) **【推荐】** 使用 === 和 !== 而不是 == 和 !=。 eslint: [eqeqeq](https://eslint.org/docs/rules/eqeqeq.html)
* 原因？== 和 != 存在类型转换，会得到和 ===、!== 不一样的结果
* // bad, true  
  undefined == null;  
  // good, false  
  undefined === null;  
    
  // bad, true  
  "0" == 0;  
  // good, false  
  "0" === 0;  
    
  // bad, true  
  0 == false;  
  // good, false  
  0 === false;  
    
  // bad, true  
  "" == false;  
  // good, false  
  "" === false;
* [15.2](#comparison--if) **【可选】** 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**，{} 和 [] 的取值也为: **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， 数组（即使是空的）是一个对象，对象的取值为 true  
  }
* [15.3](#comparison--shortcuts) **【推荐】** 对于布尔值（在明确知道是布尔值的情况下）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](#comparison--moreinfo) **【可选】** 关于布尔值转换和条件语句，详细信息可参阅 Angus Croll 的 [Truth Equality and JavaScript](https://javascriptweblog.wordpress.com/2011/02/07/truth-equality-and-javascript/#more-2108) 这篇文章。
* [15.5](#comparison--switch-blocks) **【必须】** 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级作用域。 eslint: [no-case-declarations](https://eslint.org/docs/rules/no-case-declarations.html)
* 原因？ 变量声明的作用域在整个 switch 语句内，但是只有在 case 条件为真时变量才会被初始化。 当多个 case 语句定义相同的变量时，就会导致变量覆盖的问题。
* // 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](#comparison--nested-ternaries) **【推荐】** 三元表达式不应该嵌套，通常是单行表达式，如果确实需要多行表达式，那么应该考虑使用条件语句。 eslint: [no-nested-ternary](https://eslint.org/docs/rules/no-nested-ternary.html)
* // bad  
  const foo = maybe1 > maybe2 ? "bar" : value1 > value2 ? "baz" : null;  
    
  // 分离为两个三元表达式  
  const maybeNull = value1 > value2 ? "baz" : null;  
    
  // better  
  const foo = maybe1 > maybe2 ? "bar" : maybeNull;  
    
  // best  
  const foo = maybe1 > maybe2 ? "bar" : maybeNull;
* [15.7](#comparison--unneeded-ternary) **【推荐】** 避免不必要的三元表达式。 eslint: [no-unneeded-ternary](https://eslint.org/docs/rules/no-unneeded-ternary.html)
* // 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](#comparison--no-mixed-operators) **【必须】** 使用混合运算符时，使用小括号括起来需要一起计算的部分，只要觉得有必要，那么尽可能地用括号让代码的优先级更明显。大家都能理解的运算符 +、-、\*\* 不要求添加括号。但我们建议在 \*、/ 之间添加括号，因为乘除运算符混写写的比较长的时候容易产生歧义。 eslint: [no-mixed-operators](https://eslint.org/docs/rules/no-mixed-operators.html)
* 原因？ 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;  
  }  
    
  // bad  
  const bar1 = a + (b / c) \* (d / e);  
    
  // 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 bar1 = a + (b / c) \* (d / e);

### 16. 代码块

* [16.1](#blocks--braces) **【必须】** 当有多行代码块的时候，应使用大括号包裹。 eslint: [nonblock-statement-body-position](https://eslint.org/docs/rules/nonblock-statement-body-position)
* // bad  
  if (test) return false;  
    
  // bad  
  let condition = true;  
  let test = 1;  
  // 在缩进不规范的时候，容易造成误解  
  if (condition) condition = false;  
  test = 2;  
    
  // good  
  if (test) return false;  
    
  // good  
  if (test) {  
   return false;  
  }  
    
  // bad  
  function foo() {  
   return false;  
  }  
    
  // good  
  function bar() {  
   return false;  
  }
* [16.2](#blocks--cuddled-elses) **【必须】** 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](https://eslint.org/docs/rules/brace-style.html)
* // bad  
  if (test) {  
   thing1();  
   thing2();  
  } else {  
   thing3();  
  }  
    
  // good  
  if (test) {  
   thing1();  
   thing2();  
  } else {  
   thing3();  
  }
* [16.3](#blocks--no-else-return) **【推荐】** 如果一个 if 块总是会执行 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](https://eslint.org/docs/rules/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;  
   }  
  }

### 17. 控制语句

* [17.1](#control-statements) **【推荐】** 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。
* 原因？ 在行的开头要求运算符保持对齐，并遵循类似于方法链的模式。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") &&  
   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](#control-statements--value-selection) **【必须】** 不要使用选择操作符代替控制语句。
* // bad  
  !isRunning && startRunning();  
    
  // good  
  if (!isRunning) {  
   startRunning();  
  }

### 18. 注释

* [18.1](#comments--multiline) **【必须】** 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](#comments--singleline) **【推荐】** 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。 建议在注释之前放一个空行，除非它在块的第一行。但如果一段代码每行都包含注释，允许不加分行。
* // 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](#comments--spaces) **【必须】** 用一个空格开始所有的注释，使它更容易阅读。 eslint: [spaced-comment](https://eslint.org/docs/rules/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](#comments--actionitems) **【推荐】** 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相应代码，如果你提出了一个需要重新讨论的问题，或者你对需要解决的问题提出的解决方案。 这些不同于其他普通注释，因为它们是可操作的。 这些操作是 FIXME: -- 需要解决这个问题 或者 TODO: -- 需要被实现。
* [18.5](#comments--fixme) **【推荐】** 使用 // FIXME: 注释问题。
* class Calculator extends Abacus {  
   constructor() {  
   super();  
    
   // FIXME: 这里不应该使用全局变量  
   total = 0;  
   }  
  }
* [18.6](#comments--todo) **【推荐】** 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;  
   }  
  }

### 19. 空白

* [19.1](#whitespace--spaces) **【推荐】** 使用 tabs (空格字符) 设置为 2 个空格。 eslint: [indent](https://eslint.org/docs/rules/indent.html)
* // bad  
  function foo() {  
  ∙∙∙∙let name;  
  }  
    
  // bad  
  function bar() {  
  ∙let name;  
  }  
    
  // good  
  function baz() {  
  ∙∙let name;  
  }
* [19.2](#whitespace--before-blocks) **【推荐】** 在花括号前放置一个空格。 eslint: [space-before-blocks](https://eslint.org/docs/rules/space-before-blocks.html)
* // 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](#whitespace--around-keywords) **【必须】** 在控制语句中的左括号前放置 1 个空格（if，while 等）。在函数调用和声明中，参数列表和函数名之间不能留空格。[keyword-spacing](https://eslint.org/docs/rules/keyword-spacing.html)
* // bad  
  if (isJedi) {  
   fight();  
  }  
    
  // good  
  if (isJedi) {  
   fight();  
  }  
    
  // bad  
  function fight() {  
   console.log("Swooosh!");  
  }  
    
  // good  
  function fight() {  
   console.log("Swooosh!");  
  }
* [19.4](#whitespace--infix-ops) **【必须】** 运算符左右设置各设置一个空格 eslint: [space-infix-ops](https://eslint.org/docs/rules/space-infix-ops.html)
* // bad  
  const x = y + 5;  
    
  // good  
  const x = y + 5;
* [19.5](#whitespace--newline-at-end) **【必须】** 在文件的结尾需要保留一个空行 eslint: [eol-last](https://github.com/eslint/eslint/blob/master/docs/rules/eol-last.md)
* // 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](#whitespace--chains) **【推荐】** 在编写多个方法链式调用(超过两个方法链式调用)时。 使用前导点，强调这行是一个方法调用，而不是一个语句。 eslint: [newline-per-chained-call](https://eslint.org/docs/rules/newline-per-chained-call) [no-whitespace-before-property](https://eslint.org/docs/rules/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](#whitespace--after-blocks) **【推荐】** 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;  
    
  // bad  
  const arr = [function foo() {}, function bar() {}];  
  return arr;  
    
  // good  
  const arr = [function foo() {}, function bar() {}];  
    
  return arr;
* [19.8](#whitespace--padded-blocks) **【必须】** 不要在块的开头使用空白行。 eslint: [padded-blocks](https://eslint.org/docs/rules/padded-blocks.html)
* // bad  
  function bar() {  
   console.log(foo);  
  }  
    
  // bad  
  if (baz) {  
   console.log(qux);  
  } else {  
   console.log(foo);  
  }  
    
  // 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](#whitespace--no-multiple-blanks) **【必须】** 不要使用多个空行填充代码。 eslint: [no-multiple-empty-lines](https://eslint.org/docs/rules/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](#whitespace--in-parens) \*\*【必须】\*\* 不要在括号内添加空格。 eslint: [`space-in-parens`](https://eslint.org/docs/rules/space-in-parens.html)  
  
```javascript  
// 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](#whitespace--in-brackets) **【必须】** 不要在中括号中添加空格。 eslint: [array-bracket-spacing](https://eslint.org/docs/rules/array-bracket-spacing.html)
* // bad  
  const foo = [1, 2, 3];  
  console.log(foo[0]);  
    
  // good  
  const foo = [1, 2, 3];  
  console.log(foo[0]);
* [19.12](#whitespace--in-braces) **【推荐】** 在花括号内添加空格。 eslint: [object-curly-spacing](https://eslint.org/docs/rules/object-curly-spacing.html)
* // bad  
  const foo = { clark: "kent" };  
    
  // good  
  const foo = { clark: "kent" };
* [19.13](#whitespace--max-len) **【必须】** 避免让你的代码行超过 120 个字符(包括空格)。 注意：根据上边的[规则](#strings--line-length)，长字符串编写可不受该规则约束，不应该被分解。 eslint: [max-len](https://eslint.org/docs/rules/max-len.html)
* 原因？ 这样能够提升代码可读性和可维护性。
* // 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](#whitespace--block-spacing) **【必须】** 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](https://eslint.org/docs/rules/block-spacing)
* // bad  
  function foo() {  
   return true;  
  }  
  if (foo) {  
   bar = 0;  
  }  
    
  // good  
  function foo() {  
   return true;  
  }  
  if (foo) {  
   bar = 0;  
  }
* [19.15](#whitespace--comma-spacing) **【必须】** 逗号之前避免使用空格，逗号之后需要使用空格。eslint: [comma-spacing](https://eslint.org/docs/rules/comma-spacing)
* // bad  
  const arr = [1, 2];  
    
  // good  
  const arr = [1, 2];
* [19.16](#whitespace--computed-property-spacing) **【推荐】** 不要在计算属性括号内插入空格。eslint: [computed-property-spacing](https://eslint.org/docs/rules/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](#whitespace--func-call-spacing) **【必须】** 避免在函数名及其入参括号之间插入空格。 eslint: [func-call-spacing](https://eslint.org/docs/rules/func-call-spacing)
* // bad  
  func();  
    
  func();  
    
  // good  
  func();
* [19.18](#whitespace--key-spacing) **【必须】** 在对象的属性和值之间的冒号前不加空格，冒号后加空格。 eslint: [key-spacing](https://eslint.org/docs/rules/key-spacing)
* // bad  
  var obj = { foo: 42 };  
  var obj2 = { foo: 42 };  
    
  // good  
  var obj = { foo: 42 };
* [19.19](#whitespace--no-trailing-spaces) **【必须】** 避免在行尾添加空格。 eslint: [no-trailing-spaces](https://eslint.org/docs/rules/no-trailing-spaces)
* [19.20](#whitespace--no-multiple-empty-lines) **【必须】** 在代码开始处不允许存在空行，行间避免出现多个空行，而结尾处必须保留一个空行。 eslint: [no-multiple-empty-lines](https://eslint.org/docs/rules/no-multiple-empty-lines)
* // bad  
  const x = 1;

const y = 2;

// good const x = 1;

const y = 2;

- [19.21](#linebreak-style) \*\*【推荐】\*\* 推荐使用 Unix 的 LF 作为换行符，而不是 Windows 的 CRLF，这样可以统一文件的换行符，避免因为换行符导致的格式混乱。  
  
### 20. 逗号  
  
- [20.1](#commas--leading-trailing) \*\*【必须】\*\* 逗号\*\*不能\*\*前置 eslint: [`comma-style`](https://eslint.org/docs/rules/comma-style.html)  
  
```javascript  
// 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](#commas--dangling) **【推荐】** 添加尾随逗号： **可以** eslint: [comma-dangle](https://eslint.org/docs/rules/comma-dangle.html)
* 原因？ 在 git diff 时能够更加清晰地查看改动。 另外，像[Babel](https://babeljs.io/)这样的编译器，会在转译时删除代码中的尾逗号，这意味着你不必担心旧版浏览器中的[尾随逗号问题](https://github.com/airbnb/javascript/blob/es5-deprecated/es5/README.md#commas) 。
* // 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);

### 21. jQuery

* [21.1](#jquery--dollar-prefix) **【推荐】** 对于 jQuery 对象一律使用 $ 符作为前缀。
* // bad  
  const sidebar = $(".sidebar");  
    
  // good  
  const $sidebar = $(".sidebar");  
    
  // good  
  const $sidebarBtn = $(".sidebar-btn");
* [21.2](#jquery--cache) **【推荐】** 缓存 jQuery 查询，节省 DOM 查询开销。
* // bad  
  function setSidebar() {  
   $(".sidebar").hide();  
    
   // ...  
    
   $(".sidebar").css({  
   "background-color": "pink",  
   });  
  }  
    
  // good  
  function setSidebar() {  
   const $sidebar = $(".sidebar");  
   $sidebar.hide();  
    
   // ...  
    
   $sidebar.css({  
   "background-color": "pink",  
   });  
  }
* [21.3](#jquery--find) **【推荐】** 能通过一次调用查找到的，不要分多次；在已有对象内查询，使用 find 函数，减少重复查询。
* // bad  
  $("ul", ".sidebar").hide();  
    
  // bad  
  $(".sidebar")  
   .find("ul")  
   .hide();  
    
  // good  
  $(".sidebar ul").hide();  
    
  // good  
  $(".sidebar > ul").hide();  
    
  // good  
  $sidebar.find("ul").hide();

### 22. 类型转换和强制类型转换

* [22.1](#coercion--strings) **【推荐】** 使用 String() 函数将变量转成字符串，比较保险： eslint: [no-new-wrappers](https://eslint.org/docs/rules/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);
* [22.2](#coercion--numbers) **【推荐】** 数字类型转换推荐用 Number() 或者 parseInt() 函数，其中 parseInt() 需显式标明底数。 eslint: [radix](https://eslint.org/docs/rules/radix) [no-new-wrappers](https://eslint.org/docs/rules/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);
* [22.3](#coercion--comment-deviations) **【可选】** 如果你对性能有极高的要求，觉得 [parseInt 性能太低](https://jsperf.com/coercion-vs-casting/3) ，可以使用位运算，但请用注释说明代码含义，否则很难看懂。
* // 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;
* [22.4](#coercion--bitwise) **注意：** **【可选】** 谨慎使用位运算符。 在 js 里，[数字的最大值是 64 位](https://es5.github.io/#x4.3.19) ，但位运算只能返回 32 位的整数 ([来源](https://es5.github.io/#x11.7))。 对于大于 32 位的数值，位运算无法得到预期结果。[讨论](https://github.com/airbnb/javascript/issues/109)。 最大的 32 位整数是： 2,147,483,647。
* 2147483647 >> 0; // => 2147483647  
  2147483648 >> 0; // => -2147483648  
  2147483649 >> 0; // => -2147483647
* [22.5](#coercion--booleans) **【推荐】** 用两个叹号来转换布尔类型： eslint: [no-new-wrappers](https://eslint.org/docs/rules/no-new-wrappers)
* const age = 0;  
    
  // bad  
  const hasAge = new Boolean(age);  
    
  // good  
  const hasAge = Boolean(age);  
    
  // best  
  const hasAge = !!age;

### 23. 命名规范

* [23.1](#naming--descriptive) **【可选】** 避免用单个字母来命名函数或变量。 尽量让名字具有可读性。
* 补充说明：由于 eslint 的 [id-length](https://eslint.org/docs/rules/id-length) 规则会把 for(let i=0; i < len; i++)这种情况也报错，这不太符合我们的习惯，因此在配置中暂时去掉 id-length 检查。
* // bad  
  function q() {  
   // ...  
  }  
    
  // good  
  function query() {  
   // ...  
  }
* [23.2](#naming--camelCase) **【必须】** 使用驼峰命名法（camelCase）命名对象、函数和实例。 eslint: [camelcase](https://eslint.org/docs/rules/camelcase.html)
* // bad  
  const OBJEcttsssss = {};  
  const this\_is\_my\_object = {};  
  function c() {}  
    
  // good  
  const thisIsMyObject = {};  
  function thisIsMyFunction() {}
* [23.3](#naming--PascalCase) **【必须】** 只有在命名构造器或者类的时候，才用帕斯卡命名法（PascalCase），即首字母大写。 eslint: [new-cap](https://eslint.org/docs/rules/new-cap.html)
* // 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](#naming--leading-underscore) **【推荐】** 变量命名时不要使用前置或者后置下划线。 eslint: [no-underscore-dangle](https://eslint.org/docs/rules/no-underscore-dangle.html)
* 因为在 Javascript 里属性和方法没有私有成员一说。 虽然前置下划线通常表示这是私有成员，但实际上还是公开的。 你无法阻止外部调用这类方法。 并且，开发人员容易误以为修改这类函数不需要知会调用方或者不需要测试。 简而言之，如果你想定义私有成员，必须使其不可见。
* // bad  
  this.\_\_firstName\_\_ = "Panda";  
  this.firstName\_ = "Panda";  
  this.\_firstName = "Panda";  
    
  // good  
  this.firstName = "Panda";  
    
  // 好，在 WeakMap 可用的环境中。 参考 https://kangax.github.io/compat-table/es6/#test-WeakMap  
  const firstNames = new WeakMap();  
  firstNames.set(this, "Panda");
* [23.5](#naming--self-this) **【推荐】** 不要保存 this 的引用，请使用箭头函数或者 [函数#bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/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](#naming--filename-matches-export) **【可选】** 文件名应该和默认导出的名称保持一致（文件名建议使用 kebab-case，后缀为小写）。
* // 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 "./check-box"; // kebab-case export/import/filename  
  import fortyTwo from "./forty-two"; // kebab-case export/import/filename  
  import insideDirectory from "./inside-directory"; // kebab-case export/import/directory name/implicit "index"  
  // ^ supports both inside-directory.js and inside-directory/index.js
* [23.7](#naming--camelCase-default-export) **【必须】** 导出默认函数时使用驼峰命名法，并且文件名应该和方法名相同。文件名建议使用 kebab-case，后缀为小写。
* function makeStyleGuide() {  
   // ...  
  }  
    
  export default makeStyleGuide;
* [23.8](#naming--PascalCase-singleton) **【必须】** 当导出构造器 / 类 / 单例 / 函数库 / 对象时应该使用帕斯卡命名法（首字母大写）。
* const ZyxStyleGuide = {  
   es6: {},  
  };  
    
  export default ZyxStyleGuide;
* [23.9](#naming--Acronyms-and-Initialisms) **【推荐】** 缩略词和缩写都必须是全部大写或者全部小写，可读性更好。
* // 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](#naming--uppercase) **【可选】** 对于 export 的常量，可以用全大写命名，但模块内部的常量名不需要全大写（用驼峰试命名可读性更好）。
* UPPERCASE\_VARIABLES 全大写变量可以让开发者知道这是个常量。 但注意在常量对象内的属性名不需要全大写(如 EXPORTED\_OBJECT.key)。
* // 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",  
  };

### 24. 存取器

* [24.1] **【可选】** 如果没有特殊需要，类属性存取器其实是没有必要的。
* [24.2](#accessors--no-getters-setters) **【可选】** 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，如果你需要处理存取过程的时候可以使用函数 getVal() 和 setVal('hello') 实现。
* // bad  
  class Dragon {  
   get age() {  
   // ...  
   }  
    
   set age(value) {  
   // ...  
   }  
  }  
    
  // good  
  class Dragon {  
   getAge() {  
   // ...  
   }  
    
   setAge(value) {  
   // ...  
   }  
  }
* [24.3](#accessors--boolean-prefix) **【推荐】** 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](#accessors--consistent) **【可选】** 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];  
   }  
  }

### 25. 事件

* [25.1](#events--hash) **【推荐】** 无论原生 DOM 事件还是像 Backbone 事件这样的自定义事件，在给事件传递参数时，参数应该使用对象而不是单个值。这样未来如果参数有所变化，就不用修改事件回调函数本身。
* // bad  
  $(this).trigger("listingUpdated", listing.id);  
    
  // ...  
    
  $(this).on("listingUpdated", (e, listingID) => {  
   // do something with listingID  
  });
* // good  
  $(this).trigger("listingUpdated", { listingID: listing.id });  
    
  // ...  
    
  $(this).on("listingUpdated", (e, data) => {  
   // do something with data.listingID  
  });

### 26. 分号

* [26.1](#semicolons--required) **【必须】** **要加分号** eslint: [semi](https://eslint.org/docs/rules/semi.html)
* 原因？ 当 JavaScript 解析器解析到没有分号的单行代码时，它会使用一个叫做 [自动分号插入算法（Automatic Semicolon Insertion）](https://tc39.github.io/ecma262/#sec-automatic-semicolon-insertion) 来确定是否应该以换行符视为语句的结束，如果判断为语句结束，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。 随着越来越多的新特性成为 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 以捕获缺少分号的代码行，将有助于预防此类问题。
* // bad - 可能异常  
  const jedis = {}[("luke", "leia")].some((name) => {  
   jedis[name] = true;  
   return true;  
  });  
    
  // 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 jedis = {};  
  ["luke", "leia"].some((name) => {  
   jedis[name] = true;  
   return true;  
  });  
    
  // 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";  
  }

## CSS 规范

### 前言

为前端开发提供良好的样式编码风格指南。

基于 Airbnb 编码规范结合实际使用进行制定。

**必须（Mandatory）** 级别要求必须严格按照规范的编码格式编写，否则会在代码扫描和自动化构建中报错。

**推荐（Preferable）** 级别希望尽量按照规范编写，但如有特殊情况，可以不采用。

**可选（Optional）** 级别并不对编码提出要求，但仍希望按参考规范编写。

### 1. 术语

#### 1.1 规则声明

我们把一个（或一组）选择器和一组属性称之为 “规则声明”。举个例子：

.listing {  
 font-size: 18px;  
 line-height: 1.2;  
}

#### 1.2 选择器

在规则声明中，“选择器” 负责选取 DOM 树中的元素，这些元素将被定义的属性所修饰。选择器可以匹配 HTML 元素，也可以匹配一个元素的类名、ID, 或者元素拥有的属性。以下是选择器的例子：

.my-element-class {  
 /\* ... \*/  
}  
  
[aria-hidden] {  
 /\* ... \*/  
}

#### 1.3 属性

最后，属性决定了规则声明里被选择的元素将得到何种样式。属性以键值对形式存在，一个规则声明可以包含一或多个属性定义。以下是属性定义的例子：

.some-selector {  
 background: #f1f1f1;  
 color: #333;  
}

### 2. CSS

本节的规范也适用于 Sass/less 等预编译语言，建议使用 Sass 进行开发。

用Prettier美化器美化后可以让代码满足第 4、5、6、7 点以及第 9 点颜色值变小写的要求。

#### 2.1 格式

2.1.1 **【必须】** 使用 2 个空格作为缩进。stylelint: indentation

**Bad**

.avatar {  
 border-radius: 50%;  
}

**Good**

.avatar {  
 border-radius: 50%;  
}

2.1.2 **【推荐】** 类名建议使用破折号代替驼峰法，推荐使用 BEM 方式命名。

BEM，也就是 “Block-Element-Modifier”，是一种用于 HTML 和 CSS 类名的*命名约定*。BEM 最初是由 Yandex 提出的，要知道他们拥有巨大的代码库和可伸缩性，BEM 就是为此而生的，并且可以作为一套遵循 OOCSS 的参考指导规范。

**Bad**

.listingCard {  
 /\* ... \*/  
}

**Good**

/\* listing-card 是一个块（block），表示高层次的组件 \*/  
.listing-card {  
 /\* ... \*/  
}  
  
/\*listing-card\_\_title 是一个元素（element），它属于 .listing-card 的一部分，因此块是由元素组成的。\*/  
.listing-card\_\_title {  
 /\* ... \*/  
}  
  
/\*listing-card--featured 是一个修饰符（modifier），表示这个块与 .listing-card 有着不同的状态或者变化。\*/  
.listing-card--featured {  
 /\* ... \*/  
}

2.1.3 **【必须】** 不要使用 ID 选择器。 sytlelint: selector-max-id 在 CSS 中，虽然可以通过 ID 选择元素，但大家通常都会把这种方式列为反面教材。ID 选择器给你的规则声明带来了不必要的高优先级，而且 ID 选择器是不可重用的。 想要了解关于这个主题的更多内容，参见 [CSS Wizardry 的文章](http://csswizardry.com/2014/07/hacks-for-dealing-with-specificity/)，文章中有关于如何处理优先级的内容。

ps: selector-no-id 已经从 stylelint v7.12.0 废除，使用 selector-max-id: 0 代替

**Bad**

#title {  
 border-radius: 50%;  
}

**Good**

.title {  
 border-radius: 50%;  
}

2.1.4 **【必须】** 在一个规则声明中应用了多个选择器时，每个选择器独占一行。stylelint: selector-list-comma-newline-after

（本规则可以用Prettier美化器美化器来满足。）

**Bad**

.one,  
.selector,  
.per-line {  
 /\* ... \*/  
}

**Good**

.one,  
.selector,  
.per-line {  
 /\* ... \*/  
}

2.1.5 **【必须】** 在规则声明的左大括号 { 前加上一个空格。 stylelint: block-opening-brace-space-before

（本规则可以用Prettier美化器美化器来满足。）

**Bad**

.one {  
 /\* ... \*/  
}

**Good**

.one {  
 /\* ... \*/  
}

2.1.6 **【必须】** 在属性的冒号 : 后面加上一个空格，前面不加空格。stylelint: declaration-colon-space-after declaration-colon-space-before

（本规则可以用Prettier美化器美化器来满足。）

**Bad**

.avatar {  
 border-radius: 50%;  
 background: #ffffff;  
 border: 0.02rem solid white;  
}

**Good**

.avatar {  
 border-radius: 50%;  
 background: #fff;  
 border: 0.02rem solid white;  
}

2.1.7 **【必须】** 规则声明的右大括号 } 独占一行。stylelint: block-closing-brace-newline-before

（本规则可以用Prettier美化器美化器来满足。）

**Bad**

.avatar {  
 border-radius: 50%;  
 background: #ffffff;  
 border: 0.02rem solid white;  
}

**Good**

.avatar {  
 border-radius: 50%;  
 background: #fff;  
 border: 0.02rem solid white;  
}

2.1.8 **【推荐】** 颜色值用小写，能用三个字来表示，则推荐用三个字来表示。stylelint color-hex-length: sort color-hex-case: lower

**Bad**

.foo {  
 background: #abcdef;  
}  
  
.bar {  
 background: #ffffff;  
}

**Good**

.foo {  
 background: #abcdef;  
}  
  
.bar {  
 background: #fff;  
}

2.1.9 **【必须】** 不要使用!important。stylelint: declaration-no-important

**Bad**

.avatar {  
 border: 0.02rem solid white !important;  
}

**Good**

.avatar {  
 border: 0.02rem solid white;  
}

2.1.10 **【必须】** 在定义无边框样式时，使用 0 代替 none。stylelint: declaration-property-value-disallowed-list

uses:

{  
 // ...  
 "declaration-property-value-disallowed-list": [  
 {  
 "/^border/": ["none"]  
 }  
 ]  
}

**Bad**

.foo {  
 border: none;  
}

**Good**

.foo {  
 border: 0;  
}

2.1.11 **【推荐】** before、after、active、focus 等状态只使用一个:。 stylelint: selector-pseudo-element-colon-notation

**Bad**

.item {  
 &::before {  
 content: "";  
 }  
 &::after {  
 content: "";  
 }  
}  
  
input {  
 &::active {  
 background: #f00;  
 }  
 &::focus {  
 background: #ff0;  
 }  
}

**Good**

.item {  
 &:before {  
 content: "";  
 }  
 &:after {  
 content: "";  
 }  
}  
  
input {  
 &:active {  
 background: #f00;  
 }  
 &:focus {  
 background: #ff0;  
 }  
}

2.1.12 **【推荐】** 样式中不要添加浏览器前缀，由编译工具自动添加。stylelint: value-no-vendor-prefix

**Bad**

.item {  
 -webkit-box-sizing: border-box;  
 -webkit-transform: scale(0.5);  
 -moz-transform: scale(0.5);  
 -ms-transform: scale(0.5);  
}

**Good**

.item {  
 box-sizing: border-box;  
 transform: scale(0.5);  
}

2.1.13 **【必须】** 禁止小于 1 的小数前加 0。stylelint: number-leading-zero: never

**Bad**

.item {  
 line-height: 0.5;  
}

**Good**

.item {  
 line-height: 0.5;  
}

2.1.14 **【必须】** 字符串使用单引号。stylelint: string-quotes: single

**Bad**

.item::before {  
 content: "x";  
}

**Good**

.item:before {  
 content: "x";  
}

2.1.15 **【必须】** 单行属性声明块中只允许声明一个属性。stylelint: declaration-block-single-line-max-declarations: 1

**Bad**

.item { color: red, top: 3px }

**Good**

.item {  
 color: red;  
 top: 3px;  
}

2.1.16 **【推荐】** 最大嵌套深度。stylelint: max-nesting-depth: 10

#### 2.2 注释

2.2.1 **【推荐】** 建议使用行注释 (在 Sass 中是 //) 代替块注释。

**Bad**

/\*  
\* 注释1  
\* 注释2  
\*/  
.avatar {  
 border: 0.02rem solid white;  
}  
  
.avatar {  
 border: 0.02rem solid white; /\*注释\*/  
}

**Good**

/\* 注释 \*/  
.avatar {  
 border: 0.02rem solid white;  
}

2.2.2 **【推荐】** 给需要注释的代码写上详细说明，比如：

* 为什么用到了 z-index
* 兼容性处理或者针对特定浏览器的 hack

**Good**

/\* 用到了z-index的原因 \*/  
.avatar {  
 z-index: 3;  
}  
  
/\* 针对华为手机的兼容性处理 \*/  
.huawei-hack {  
 /\* ... \*/  
}

2.2.3 **【推荐】** 在 html 上说明加上某个类有什么用，例如：加上 show 类时显示弹窗，加上 me 类显示“我的”样式。

**Good**

<!-- 加上show类时显示弹窗 -->  
<div class="dialog show"></div>  
  
<!-- 加上me类时表示我的消息的样式 -->  
<div class="message me"></div>

2.2.4 **【推荐】** 非首行注释，需要在注释前空行。 stylelint: comment-empty-line-before

**Bad**

.item {  
}  
/\* comment \*/

**Good**

.item {  
}  
  
/\* comment \*/

#### 2.3 JavaScript 钩子

2.3.1 **【推荐】** 避免在 CSS 和 JavaScript 中绑定相同的类。否则开发者在重构时通常会出现以下情况：轻则浪费时间在对照查找每个要改变的类，重则因为害怕破坏功能而不敢作出更改。

我们推荐在创建用于特定 JavaScript 的类名时，添加 js- 前缀：

**Good**

<button class="btn btn-primary js-request-to-book">Request to Book</button>

### 3. less

#### 3.1 代码组织

3.1.1 **【推荐】** 代码按如下形式按顺序组织

1. import
2. 变量声明
3. 样式声明

@import "est/all.less";  
  
@default-text-color: #333;  
  
.page {  
 width: 960px;  
 margin: 0 auto;  
}

#### 3.2 @import 语句

3.2.1 **【推荐】** @import语句引用的文件必须（MUST）写在一对引号内，.less后缀不得省略（与引入 CSS 文件时的路径格式一致），引号使用 "。

**Bad**

@import "est/all";  
@import "my/mixins";

**Good**

@import "est/all.less";  
@import "my/mixins.less";

#### 3.3 变量

3.3.1 **【必须】** 变量名应使用破折号（例如 @my-variable）代替 camelCased 和 snake\_cased 风格。对于仅用在当前文件的变量，可以在变量名之前添加下划线前缀（例如 @\_my-variable）。

**Bad**

@my\_variable = 0；  
@myVariable = 0；;

**Good**

@my-variable = 0；  
@\_my-variable = 0；;

#### 3.4 运算

3.4.1 **【推荐】** +, -, \*, / 四个运算符两侧保留一个空格。+, - 两侧的操作数有相同的单位，如果其中一个是变量，另一个数值书写单位。

**Bad**

@a: 200px;  
@b: (@a+100) \* 2;

**Good**

@a: 200px;  
@b: (@a + 100px) \* 2;

#### 3.5 Mixins

3.5.1 **【推荐】** Mixin 和后面的括号之间不得包含空格。在给 mixin 传递参数时，在参数分隔符 (,, ;) 后要保留一个空格：

**Bad**

.box {  
 .size(30px, 20px);  
 .clearfix ();  
}

**Good**

.box {  
 .size(30px, 20px);  
 .clearfix();  
}

3.5.2 **【推荐】** 在定义 mixin 时，如果 mixin 名称不是一个需要使用的 className，加上括号，否则即使不被调用也会输出到 CSS 中。

**Bad**

.big-text {  
 font-size: 2em;  
}  
  
h3 {  
 .big-text;  
}

**Good**

.big-text() {  
 font-size: 2em;  
}  
  
h3 {  
 .big-text();  
}

3.5.3 **【推荐】** 如果混入的是本身不输出内容的 mixin，在 mixin 后添加括号（即使不传参数），以区分这是否是一个 className（修改以后是否会影响 HTML）。

**Bad**

.box {  
 .clearfix;  
 .size (20px);  
}

**Good**

.box {  
 .clearfix();  
 .size(20px);  
}

#### 3.6 继承

3.6.1 **【推荐】** 使用继承时，如果在声明块内书写 :extend 语句，写在开头：

**Bad**

.sub {  
 color: red;  
 &:extend(.mod all);  
}

**Good**

.sub {  
 &:extend(.mod all);  
 color: red;  
}