Airbnb JavaScript Style Guide() {

用更合理的方式写 JavaScript

自 Airbnb JavaScript Style Guide 。

目录

类型

- 1.1 **基本类型**: 直接存取基本类型。
  - `字符串`
  - `数值`
  - `布尔类型`
  - `null`
  - `undefined`

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

bar = 9;
```

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

- 1.2 **复杂类型**: 通过引用的方式存取复杂类型。
  - `对象`
  - `数组`
  - `函数`

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

bar[0] = 9;

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

返回目录

引用

- 2.1 对**所有的引用使用** `const` ;**不要使用** `var` 。

> 为什么？这能确保你无法对引用重新赋值，也不会导致出现 bug 或难以理解。

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

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

- 2.2 **如果你一定需要可变动的引用，使用** `let` **代替** `var` 。

> 为什么？因为 `let` 是块级作用域，而 `var` 是函数作用域。

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

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

- 2.3 **注意** `let` **和** `const` **都是块级作用域。**

```
// const 和 let 只存在于它们被定义的区块内。
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

返回目录

对象

- 3.1 使用字面值创建对象。

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

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

- 3.2 如果你的代码在浏览器环境下执行，别使用 保留字 作为键值。这样的话在 IE8 不会运行。更多信息。但在 ES6 模块和服务器端中使用没有问题。

```
// bad
const superman = {
  default: { clark: 'kent' },
  private: true,
};

// good
const superman = {
  defaults: { clark: 'kent' },
  hidden: true,
};
```

- 3.3 使用同义词替换需要使用的保留字。

```
// bad
const superman = {
  class: 'alien',
};

// bad
const superman = {
  klass: 'alien',
};

// good
const superman = {
  type: 'alien',
};
```

- 3.4 创建有动态属性名的对象时，使用可被计算的属性名称。

> 为什么？因为这样可以让你在一个地方定义所有的对象属性。

``javascript 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,
};
```

- 3.5 使用对象方法的简写。

```
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
```

```
  },
};

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

- 3.6 **使用对象属性值的简写。**

> 为什么？因为这样更短更有描述性。

```
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

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

- 3.7 **在对象属性声明前把简写的属性分组。**

> 为什么？因为这样能清楚地看出哪些属性使用了简写。

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

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

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

[⬆ 返回目录]

数组

- 4.1 **使用字面值创建数组。**

```
// bad
const items = new Array();

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

- 4.2 向数组添加元素时使用 Arrary#push 替代直接赋值。

  ```javascript const someStack = [];

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

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

- 4.3 使用拓展运算符 `...` 复制数组。

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

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

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

- 4.4 使用 Array#from 把一个类数组对象转换成数组。

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

1 返回目录

解构

- 5.1 使用解构存取和使用多属性对象。

> 为什么？因为解构能减少临时引用属性。

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

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

// good
function getFullName(obj) {
  const { firstName, lastName } = obj;
  return `${firstName} ${lastName}`;
}

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

- 5.2 对数组使用解构赋值。

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

// bad
```

```
const first = arr[0];
const second = arr[1];

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

- 5.3 需要回传多个值时，使用对象解构，而不是数组解构。

> 为什么？增加属性或者改变排序不会改变调用时的位置。

```
// bad
function processInput(input) {
  // then a miracle occurs
  return [left, right, top, bottom];
}

// 调用时需要考虑回调数据的顺序。
const [left, __, top] = processInput(input);

// good
function processInput(input) {
  // then a miracle occurs
  return { left, right, top, bottom };
}

// 调用时只选择需要的数据
const { left, right } = processInput(input);
```

[1] 返回目录

Strings

- 6.1 字符串使用单引号 `''` 。

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

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

- 6.2 字符串超过 80 个字节应该使用字符串连接号换行。

- 6.3 注：过度使用字串连接符号可能会对性能造成影响。jsPerf 和 讨论.

```
// bad
const errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about how Batman had anyth

// 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.';
```

- 6.4 程序化生成字符串时，使用模板字符串代替字符串连接。

> 为什么？模板字符串更为简洁，更具可读性。

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

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

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

[⬆ 返回目录]

函数

- 7.1 **使用函数声明代替函数表达式。**

> 为什么？因为函数声明是可命名的，所以他们在调用栈中更容易被识别。此外，函数声明会把整个函数提升（hoisted），而函数表达式只会把函数的引用变量名提升。这条规则使得箭头函数可以取代函数表达式。

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

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

- 7.2 **函数表达式**:

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

- 7.3 **永远不要在一个非函数代码块（`if`、`while` 等）中声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但它们的解析表现不一致。**

- 7.4 **注意：ECMA-262 把 `block` 定义为一组语句。函数声明不是语句。阅读 ECMA-262 关于这个问题的说明。**

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

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

- 7.5 **永远不要把参数命名为 `arguments`。这将取代原来函数作用域内的 `arguments` 对象。**

```
// bad
```

```
function nope(name, options, arguments) {
  // ...stuff...
}

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

- 7.6 **不要使用 `arguments`。可以选择 rest 语法 `...` 替代。**

> 为什么？使用 `...` 能明确你要传入的参数。另外 rest 参数是一个真正的数组，而 `arguments` 是一个类数组。

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

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

- 7.7 **直接给函数的参数指定默认值，不要使用一个变化的函数参数。**

```
// really bad
function handleThings(opts) {
  // 不！我们不应该改变函数参数。
  // 更加糟糕：如果参数 opts 是 false 的话，它就会被设定为一个对象。
  // 但这样的写法会造成一些 Bugs。
  //（译注：例如当 opts 被赋值为空字符串，opts 仍然会被下一行代码设定为一个空对象。）
  opts = opts || {};
  // ...
}

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

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

- 7.8 **直接给函数参数赋值时需要避免副作用。**

> 为什么？因为这样的写法让人感到很困惑。

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

[1] 返回目录

箭头函数

- 8.1 **当你必须使用函数表达式（或传递一个匿名函数）时，使用箭头函数符号。**

> 为什么？因为箭头函数创造了新的一个 `this` 执行环境（译注：参考 Arrow functions - JavaScript | MDN 和 ES6 arrow functions, syntax and lexical scoping），通常情况下都能满足你的需求，而且这样的写法更为简洁。
>
> 为什么不？如果你有一个相当复杂的函数，你或许可以把逻辑部分转移到一个函数声明上。

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

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

- 8.2 **如果一个函数适合用一行写出并且只有一个参数，那就把花括号、圆括号和 `return` 都省略掉。如果不是，那就不要省略。**

> 为什么？语法糖。在链式调用中可读性很高。
>
> 为什么不？当你打算回传一个对象的时候。

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

// good
[1, 2, 3].reduce((total, n) => {
  return total + n;
}, 0);
```

返回目录

构造器

- 9.1 **总是使用 `class`。避免直接操作 `prototype`。**

> 为什么？因为 `class` 语法更为简洁更易读。

```javascript // bad function Queue(contents = []) { this._queue = [...contents]; } Queue.prototype.pop = function() { const value = this._queue[0]; this._queue.splice(0, 1); return value; }

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

- 9.2 **使用 `extends` 继承。**

> 为什么？因为 `extends` 是一个内建的原型继承方法并且不会破坏 `instanceof`。

```javascript
// 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` 来帮助链式调用。**

```
// 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 **可以写一个自定义的 `toString()` 方法，但要确保它能正常运行并且不会引起副作用。**

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

  getName() {
    return this.name;
  }

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

返回目录

模块

- 10.1 **总是使用模组 (`import` / `export`) 而不是其他非标准模块系统。你可以编译为你喜欢的模块系统。**

> 为什么？模块就是未来，让我们开始迈向未来吧。

```
// 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 **不要使用通配符** import。

> 为什么？这样能确保你只有一个默认 export。

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

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

- 10.3 **不要从** import **中直接** export。

> 为什么？虽然一行代码简洁明了，但让 import 和 export 各司其职让事情能保持一致。

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

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

[1] 返回目录

Iterators and Generators

- 11.1 **不要使用** iterators。**使用高阶函数例如** `map()` **和** `reduce()` **替代** `for-of`。

> 为什么？这加强了我们不变的规则。处理纯函数的回调值更易读，这比它带来的副作用更重要。

```
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;
```

- 11.2 现在还不要使用 generators。

> 为什么？因为它们现在还没法很好地编译到 ES5。(译者注：目前(2016/03) Chrome 和 Node.js 的稳定版本都已支持 generators)

返回目录

属性

- 12.1 使用 `.` 来访问对象的属性。

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

// bad
const isJedi = luke['jedi'];

// good
const isJedi = luke.jedi;
```

- 12.2 当通过变量访问属性时使用中括号 `[]`。

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

function getProp(prop) {
  return luke[prop];
}

const isJedi = getProp('jedi');
```

返回目录

变量

- 13.1 一直使用 `const` 来声明变量，如果不这样做就会产生全局变量。我们需要避免全局命名空间的污染。地球队长已经警告过我们了。(译注：全局，global 亦有全球的意思。地球队长的责任是保卫地球环境，所以他警告我们不要造成「全球」污染。)

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

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

- 13.2 使用 `const` 声明每一个变量。

> 为什么？增加新变量将变的更加容易，而且你永远不用再担心调换错 `;` 跟 `,`。

```
// 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 **将所有的 `const` 和 `let` 分组**

> **为什么？当你需要把已赋值变量赋值给未赋值变量时非常有用。**

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

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

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

- 13.4 **在你需要的地方给变量赋值，但请把它们放在一个合理的位置。**

> **为什么？`let` 和 `const` 是块级作用域而不是函数作用域。**

```
// good
function() {
  test();
  console.log('doing stuff..');

  //..other stuff..

  const name = getName();

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

  return name;
}

// bad - unnecessary function call
function(hasName) {
  const name = getName();

  if (!hasName) {
    return false;
  }

  this.setFirstName(name);

  return true;
}

// good
function(hasName) {
  if (!hasName) {
    return false;
  }

  const name = getName();
  this.setFirstName(name);

  return true;
}
```

🔝 返回目录

Hoisting

- 14.1 `var` 声明会被提升至该作用域的顶部，但它们赋值不会提升。`let` 和 `const` 被赋予了一种称为「暂时性死区（Temporal Dead Zones，TDZ）」的概念。这对于了解为什么 type of 不再安全相当重要。

```
// 我们知道这样运行不了
// (假设 notDefined 不是全局变量)
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// 由于变量提升的原因，
// 在引用变量后再声明变量是可以运行的。
// 注：变量的赋值 `true` 不会被提升。
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// 编译器会把函数声明提升到作用域的顶层，
// 这意味着我们的例子可以改写成这样：
function example() {
  let declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}

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

- 14.2 匿名函数表达式的变量名会被提升，但函数内容并不会。

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

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

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

- 14.3 命名的函数表达式的变量名会被提升，但函数名和函数函数内容并不会。

```
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');
  };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
  console.log(named); // => undefined

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

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

- 14.4 函数声明的名称和函数体都会被提升。

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

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

- 想了解更多信息，参考 Ben Cherry 的 JavaScript Scoping & Hoisting。

返回目录

比较运算符和等号

- 15.1 优先使用 `===` 和 `!==` 而不是 `==` 和 `!=`.

- 15.2 条件表达式例如 `if` 语句通过抽象方法 `ToBoolean` 强制计算它们的表达式并且总是遵守下面的规则：

  - **对象** 被计算为 **true**
  - **Undefined** 被计算为 **false**
  - **Null** 被计算为 **false**
  - **布尔值** 被计算为 **布尔的值**
  - **数字** 如果是 **+0、-0、或 NaN** 被计算为 **false**，否则为 **true**
  - **字符串** 如果是空字符串 `''` 被计算为 **false**，否则为 **true**

```
if ([0]) {
  // true
  // An array is an object, objects evaluate to true
}
```

- 15.3 使用简写。

```
// bad
if (name !== '') {
  // ...stuff...
}

// good
if (name) {
  // ...stuff...
}

// bad
if (collection.length > 0) {
  // ...stuff...
}

// good
if (collection.length) {
  // ...stuff...
}
```

- 15.4 想了解更多信息，参考 Angus Croll 的 Truth Equality and JavaScript。

返回目录

代码块

- 16.1 使用大括号包裹所有的多行代码块。

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

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
function() { return false; }

// good
function() {
  return false;
}
```

- 16.2 如果通过 `if` 和 `else` 使用多行代码块，把 `else` 放在 `if` 代码块关闭括号的同一行。

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

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

返回目录

注释

- 17.1 使用 `/** ... */` 作为多行注释。包含描述、指定所有参数和返回值的类型和值。

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

  // ...stuff...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed in tag name
 *
 * @param {String} tag
```

```
 * @return {Element} element
 */
function make(tag) {

  // ...stuff...

  return element;
}
```

- 17.2 使用 `//` 作为单行注释。在评论对象上面另起一行使用单行注释。在注释前插入空行。

```
// 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;
}
```

- 17.3 给注释增加 `FIXME` 或 `TODO` 的前缀可以帮助其他开发者快速了解这是一个需要复查的问题，或是给需要实现的功能提供一个解决方式。这将有别于常见的注释，因为它们是可操作的。使用 `FIXME -- need to figure this out` 或者 `TODO -- need to implement`。

- 17.4 使用 `// FIXME`：标注问题。

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

- 17.5 使用 `// TODO`：标注问题的解决方式。

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

返回目录

空白

- 18.1 使用 2 个空格作为缩进。

```
// bad
function() {
····const name;
}

// bad
function() {
·const name;
}

// good
function() {
··const name;
}
```

- 18.2 **在花括号前放一个空格。**

```
// 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',
});
```

- 18.3 **在控制语句(`if`、`while` 等)的小括号前放一个空格。在函数调用及声明中，不在函数的参数列表前加空格。**

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

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

- 18.4 **使用空格把运算符隔开。**

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

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

- 18.5 **在文件末尾插入一个空行。**

```
// bad
(function(global) {
  // ...stuff...
})(this);
```

```
// bad
(function(global) {
  // ...stuff...
})(this);↵
↵
```

```
// good
(function(global) {
  // ...stuff...
})(this);↵
```

- 18.5 **在使用长方法链时进行缩进。使用前面的点** `.` **强调这是方法调用而不是新语句。**

```
// 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').class('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);
```

- 18.6 **在块末和新语句前插入空行。**

```
// 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;
```

返回目录

逗号

- 19.1 **行首逗号：不需要。**

```
// 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',
};
```

- 19.2 **增加结尾的逗号：需要。**

> 为什么？这会让 git diffs 更干净。另外，像 babel 这样的转译器会移除结尾多余的逗号，也就是说你不必担心老旧浏览器的尾逗号问题。

```
// bad - git diff without trailing comma
const hero = {
     firstName: 'Florence',
-    lastName: 'Nightingale'
+    lastName: 'Nightingale',
+    inventorOf: ['coxcomb graph', 'modern nursing']
}
```

```
// good - git diff with trailing comma
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',
];
```

返回目录

分号

- 20.1 **使用分号**

```
// bad
(function() {
  const name = 'Skywalker'
  return name
})()

// good
(() => {
  const name = 'Skywalker';
  return name;
})();

// good (防止函数在两个 IIFE 合并时被当成一个参数)
;(() => {
  const name = 'Skywalker';
  return name;
})();
```

Read more.

返回目录

类型转换

- 21.1 **在语句开始时执行类型转换。**
- 21.2 **字符串：**

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

// bad
const totalScore = this.reviewScore + '';
```

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

- 21.3 对数字使用 `parseInt` 转换，并带上类型转换的基数。

```
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);
```

- 21.4 如果因为某些原因 parseInt 成为你所做的事的瓶颈而需要使用位操作解决性能问题时，留个注释说清楚原因和你的目的。

```
// good
/**
 * 使用 parseInt 导致我的程序变慢，
 * 改成使用位操作转换数字快多了。
 */
const val = inputValue >> 0;
```

- 21.5 注：小心使用位操作运算符。数字会被当成 64 位值，但是位操作运算符总是返回 32 位的整数（参考）。位操作处理大于 32 位的整数值时还会导致意料之外的行为。关于这个问题的讨论。最大的 32 位整数是 2,147,483,647：

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

- 21.6 布尔：

```
const age = 0;

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

// good
const hasAge = Boolean(age);

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

返回目录

命名规则

- 22.1 避免单字母命名。命名应具备描述性。

```
// bad
```

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

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

- 22.2 使用驼峰式命名对象、函数和实例。

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

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

- 22.3 使用帕斯卡式命名构造函数或类。

```
// 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',
});
```

- 22.4 不要使用下划线 `_` 结尾或开头来命名属性和方法。

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

// good
this.firstName = 'Panda';
```

- 22.5 别保存 `this` 的引用。使用箭头函数或 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);
  };
}
```

- 22.6 **如果你的文件只输出一个类，那你的文件名必须和类名完全保持一致。**

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

// in some other file
// bad
import CheckBox from './checkBox';

// bad
import CheckBox from './check_box';

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

- 22.7 **当你导出默认的函数时使用驼峰式命名。你的文件名必须和函数名完全保持一致。**

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- 22.8 **当你导出单例、函数库、空对象时使用帕斯卡式命名。**

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

export default AirbnbStyleGuide;
```

返回目录

存取器

- 23.1 **属性的存取函数不是必须的。**

- 23.2 **如果你需要存取函数时使用** `getVal()` **和** `setVal('hello')`**。**

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

- 23.3 如果属性是布尔值，使用 `isVal()` 或 `hasVal()`。

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

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

- 23.4 创建 `get()` 和 `set()` 函数是可以的，但要保持一致。

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

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

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

[⬆ 返回目录](#)

事件

- 24.1 当给事件附加数据时（无论是 DOM 事件还是私有事件），传入一个哈希而不是原始值。这样可以让后面的贡献者增加更多数据到事件数据而无需找出并更新事件的每一个处理器。例如，不好的写法：

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

...

$(this).on('listingUpdated', function(e, listingId) {
  // do something with listingId
});
```

更好的写法：

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

...

$(this).on('listingUpdated', function(e, data) {
  // do something with data.listingId
});
```

[⬆ 返回目录](#)

jQuery

- 25.1 使用 `$` 作为存储 jQuery 对象的变量名前缀。

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

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

- 25.2 **缓存** jQuery 查询。

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

  // ...stuff...

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

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

  // ...stuff...

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

- 25.3 对 DOM 查询**使用层叠** `$('.sidebar ul')` **或 父元素 > 子元素** `$('.sidebar > ul')`。jsPerf

- 25.4 对**有作用域的** jQuery **对象**查询**使用** `find`。

```
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```

返回目录