Code Style

前言

现在的项目一般都是一个团队共同开发，而每个人都有自己的编码习惯，为了统一格式，项目组在项目开发之前都会制定一系列的规范。

俗话说约定优于配置，但是在执行过程中往往发现效果不是很好（主要是指编码规范这一方面）。所以我们不得不采取一些措施来协助我们统一项目开发人员的编码风格。

项目采用 ECMAScript 6 + Vue.js 开发，规范参考 Google JavaScript Style Guide 。

类型规范

对于常量或不修改的变量声明使用 const ，对于只在当前作用域下有效的变量，应使用 let ，全局变量使用 var 。

- 将所有 const 变量放在一起，然后将所有 let 变量放在一起

```
const foo = 1;

let foo1 = 2;
let bar = foo;
bar = 9;
foo1 = 3;

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

- const 和 let 使用时注意，let 和 const 都是块作用域的

```
// const and let only exist in the blocks they are
defined in.
{
  let a = 1;
  const b = 1;
}
console.log(a); // ReferenceError
console.log(b); // ReferenceError
```

字符串

- 使用单引号 '

```
// bad
var name = "Bob Parr";

// good
var name = 'Bob Parr';

// bad
var fullName = "Bob " + this.lastName;

// good
var fullName = 'Bob ' + this.lastName;
```

- 超过 80 个字符的字符串应该使用字符串连接换行

```
// bad
var 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
var 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
var 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.';
```

- 编程构建字符串时，使用字符串模板而不是字符串连接

```
// 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}?`;
}
```

数组类型

- 使用字面量语法创建数组

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

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

- 如果你不知道数组的长度，使用 push

```
const someStack = [];

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

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

- 使用 ... 来拷贝数组，不要使用 Array.from、Array.of等数组的新的内置API，Array新api用于适合的场景

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

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

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

```
// not good
const foo = [1,2,3];
const nodes = Array.from(foo);
```

解构

使用对象的多个属性时请建议使用对象的解构赋值，解构赋值避免了为这些属性创建临时变量或对象。即使转化成es5都是一样的

- 嵌套结构的对象层数不能超过3层

```
// not good
let obj = {
  'one': [
    {
      'newTwo': [
        {
          'three': [
            'four': '太多层了，头晕晕'
          ]
        }
      ]
    }
  ]
};
// good
let obj = {
  'one': [
    'two',
    {
      'twoObj': '结构清晰'
    }
  ]
```

```
};
```

- 解构语句中统一不使用圆括号

```
// not good
[(a)] = [11]; // a未定义
let { a: (b) } = {}; // 解析出错


// good
let [a, b] = [11, 22];
```

对象解构

- 对象解构元素与顺序无关对象指定默认值时仅对恒等于undefined（!== null）的情况生效

  若函数形参为对象时，使用对象解构赋值

```
  // not good
  function someFun(opt) {
    let opt1 = opt.opt1;
    let opt2 = opt.opt2;
    console.log(op1);
  }


  // good
  function someFun(opt) {
    let { opt1, opt2 } = opt;
    console.log(`$(opt1) 加上 $(opt2)`);
  }

  function someFun({ opt1, opt2 }) {
```

```
    console.log(opt1);
  }
```

- 若函数有多个返回值时，使用对象解构，不使用数组解构，避免添加顺序的问题

```
// not good
function anotherFun() {
  const one = 1, two = 2, three = 3;
  return [one, two, three];
}
const [one, three, two] = anotherFun(); // 顺序乱了
// one = 1, two = 3, three = 2


// good
function anotherFun() {
  const one = 1, two = 2, three = 3;
  return { one, two, three };
}
const { one, three, two } = anotherFun(); // 不用管顺序
// one = 1, two = 2, three = 3
```

- 已声明的变量不能用于解构赋值（语法错误）

```
// 语法错误
let a;
{ a } = { b: 123};
```

- 数组解构时数组元素与顺序相关

  例如交换数组两个元素的值

```

```

```
let x = 1;
let y = 2;

// not good
let temp;
temp = x;
x = y;
y = temp;


// good
[x, y] = [y, x]; // 交换变量
```

- 将数组成员赋值给变量时，使用数组解构

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

// not good
const one = arr[0];
const two = arr[1];


// good
const [one, two] = arr;
```

- 函数有多个返回值时使用对象解构，而不是数组解构。

  这样你就可以随时添加新的返回值或任意改变返回值的顺序，而不会导致调用失败。

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

```
  // the caller needs to think about the order of
return data
  const [left, __, top] = processInput(input);

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

  // the caller selects only the data they need
  const { left, right } = processInput(input);
```

函数

- 使用函数声明而不是函数表达式

  函数声明拥有函数名，在调用栈中更加容易识别。并且，函数声明会整体提升，而函数表达式只会提升变量本身。这条规则也可以这样描述，始终使用箭头函数来代替函数表达式。

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

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

- 绝对不要在一个非函数块（if，while，等等）里声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但是它们解析不同注：ECMA-262 把块 定义为一组语句，函数声明不是一个语句。阅读 ECMA-262 对这个问题的说明

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

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

- 绝对不要把参数命名为 arguments, 这将会覆盖函数作用域内传过来的 arguments 对象

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

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

- 永远不要使用 arguments，使用 ... 操作符来代替

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

```
}

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

- 使用函数参数默认值语法，而不是修改函数的实参

```
// really bad
function handleThings(opts) {
  opts = opts || {};
}

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

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

箭头函数 Arrow Functions

- 当必须使用函数表达式时（例如传递一个匿名函数时），请使用箭头函数

```
// bad
"use strict";
var fn = function fn(v) {
  return console.log(v);
```

```
};

// good
var fn= (v=>console.log(v));
```

- 箭头函数总是用括号包裹参数，省略括号只适用于单个参数，并且还降低了程序的可读性

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

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

- 立即执行的匿名函数

```
// 函数表达式
// immediately-invoked function expression (IIFE)
// good, 看起来就很厉害
(() => {
  console.log('Welcome to the Internet. Please follow me.');
})();
```

对象

- 使用对象字面量创建对象

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

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

- 不要使用保留字（reserved words）作为键，否则在 IE8 下将出错

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

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

- 创建对象时使用计算的属性名，而不要在创建对象后使用对象的动态特性，这样可以在同一个位置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
};
```

- 使用定义对象方法的简短形式

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

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

// good
const atom = {
  value: 1,

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

- 使用定义对象属性的简短形式，书写起来更加简单，并且可以自描述。这里和 es5 有些不同，需要注意下

```
const lukeSkywalker = 'Luke Skywalker';

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

// good
```

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

- 将所有简写的属性写在对象定义的最顶部，这样可以更加方便地知道哪些属性使用了简短形式。

```
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
};
```

类

- 总是使用 class 关键字，避免直接修改 prototype，class 语法更简洁，也更易理解。

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

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

- 类名与花括号须保留一个空格间距，类中的方法定义时，括号) 也须与花括号 { 保留一个空格间距

```
// not good
class Foo{
  constructor(){
    // constructor
  }
  sayHi()    {
    // 仅保留一个空格间距
  }
```

```
}

// good
class Foo {
  constructor() {
    // constructor
  }
  sayHi() {
    // 仅保留一个空格间距
  }
}
```

- 定义类时，方法的顺序如下：

constructor

public get/set 公用访问器，set 只能传一个参数

public methods 公用方法，公用相关命名使用小驼峰式写法(lowerCamelCase)

private get/set 私有访问器，私有相关命名应加上下划线 _ 为前缀

private methods 私有方法

```
// good
class SomeClass {
  constructor() {
    // constructor
  }

  get aval() {
    // public getter
  }

  set aval(val) {
```

```
    // public setter
  }

  doSth() {
    // 公用方法
  }

  get _aval() {
    // private getter
  }

  set _aval() {
    // private setter
  }

  _doSth() {
    // 私有方法
  }
}
```

- 如果不是class类，不使用new

```
// not good
function Foo() {

}
const foo = new Foo();

// good
class Foo {

}
const foo = new Foo();
```

- 使用 extends 关键字来继承

  这是一个内置的继承方式，并且不会破坏 instanceof 原型检查。

```
// 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];
  }
}
```

模块

- 总是在非标准的模块系统中使用标准的 import 和 export 语法，我们总是可以将标准的模块语法转换成支持特定模块加载器的语法。

  推荐使用 import和 export 来做模块加载

```
// 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;
```

- import / export 后面采用花括号 { } 引入模块的写法时，建议在花括号内左右各保留一个空格

```
// not good
import {lightRed} from './colors';
import { lightRed} from './colors';

// good
import { lightRed } from './colors';
```

- 不要使用通配符 * 的 import ，这样确保了一个模块只有一个默认的 export 项

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

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

- 不要直接从一个 import 上 export

  虽然一行代码看起来更简洁，但是有一个明确的 import 和一个明确的 export 使得代码行为更加明确。

```
// bad
// filename es6.js
```

```
export default { es6 } from './airbnbStyleGuide';

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

- 多变量要导出时应采用对象解构形式

```
// not good
export const a= 'a';
export const b= 'b';

// good
export const a= 'a';
export const b= 'b';

export default { a, b };
```

- 导出单一一个类时，确保你的文件名就是你的类名

```
// file contents
class CheckBox {
  // ...
}
module.exports = CheckBox;

// in some other file
// bad
const CheckBox = require('./checkBox');

// bad
const CheckBox = require('./check_box');
```

```
// good
const CheckBox = require('./CheckBox');
```

- 导出一个默认小驼峰命名的函数时，文件名应该就是导出的方法名

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

- 导出单例、函数库或裸对象时，使用大驼峰命名规则

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

export default AirbnbStyleGuide;
```

Iterators 和 Generators

Iterators 性能比较差，对于数组来说大致与Array.prototype.forEach 相当，比不过原生的 for 循环，而且用起来比较麻烦，数组提供了 for...of ，对象提供了 for...in ，不推荐使用迭代器。

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

// bad
var iterator = numbers[Symbol.iterator]();
var result = iterator.next();
let sum = 0;
while (!result.done) {
  sum += result.value;
  result = iterator.next();
```

```
}

// good
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;
```

- generators。不推荐使用，或者非常谨慎地使用。

  生成器不是用来写异步的，虽然确实有这样一个效果，但这仅仅是一种Hack。异步在未来一定是属于async和await这两个关键字的，但太多人眼里生成器就是写异步用的，这会导致滥用。暂时推荐用promise来实现异步。

属性访问

- 使用点 . 操作符来访问常量属性

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

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

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

- 使用中括号[] 操作符来访问变量属性

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

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

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

map + set + weakmap + weakset 数据结构

- 新加的集合类型，提供了更加方便的获取属性值的方法，可以检查某个属性是属于原型链上还是当前对象的，并用获取对象的set和get方法

  但是，推荐使用weakmap和weakset，而不是map和set，除非必须使用。普通集合会阻止垃圾回收器对这些作为属性键存在的对象的回收，有造成内存泄漏的危险

```
// not good, Maps
var wm = new Map();
wm.set(key, { extra: 42 });
wm.size === 1

// not good, Sets
```

```
var ws = new Set();
ws.add({ data: 42 });

// good, Weak Maps
var wm = new WeakMap();
wm.set(key, { extra: 42 });
wm.size === undefined

// good, Weak Sets
var ws = new WeakSet();
ws.add({ data: 42 });//因为添加到ws的这个临时对象没有其他变量引用它，所以ws不会保存它的值，也就是说这次添加其实没有意思

// not good
let object = {},
object.hasOwnProperty(key)

// good
let object = new WeakSet();
object.has(key) === true;
```

- 当你的元素或者键值有可能不是字符串时，推荐使用WeakMap和WeakSet。

```
// bad
var obj = { 3:'value' };

// good
var ws = new WeakSet();
ws.add(3, 'value');
有移除操作的需求时，使用WeakMap和WeakSet。
// bad
var obj = { 'key':'value' };
delete obj.key;
```

```
// good
var ws = new WeakSet();
ws.add('key', 'value');
ws.remove('key');
```

- 当仅需要一个不可重复的集合时，使用 WeakSet 优先于普通对象，而不要使用 {foo: true} 这样的对象。

```
// bad
var obj = { 'key':'value' };

// good
var ws = new WeakSet();
ws.add('key', 'value');
```

- 当需要遍历功能时，使用 WeakMap 和 WeakSet ，因为其可以简单地使用 for..of 进行遍历，性能更高

```
// bad
var obj = { key:'value', key1: 'value1' };
for(var key in obj){
}

// good
var ws = new WeakSet();
ws.add('key', 'value').add('key1', 'value1');
for(var key of ws){
}
```

promise 、symbols 、proxies

promise 是一种异步处理模式。发 promise 申明和调用分开，推荐异步方式使用 Promise 。

```
// not good
(new Promise(resolve, reject){})
  .then(cunction(){},function(){})
  .then();

// good
var promise = new Primise(function(resolve, reject){});
promise
  .then(function(){},function(){]})
  .then();
```

- symbol 用于对象的键和私有属性，使用过于复杂，没有使用必要，容易扰乱外层作用域。总之不要使用

  ```
  // good
  function MyClass(privateData) {
    let key = privateData;
  }

  //not good
  const key = Symbol('key');
  function MyClass(privateData) {
    this[key] = privateData;
  }

  const object = new MyClass("hello")
  object['key'] === undefined //无法访问该属性，因为是私有的
  ```

- Proxy可以监听对象身上发生了什么事情，并在这些事情发生后执行一些相应的操作，没有特别要注意的，尽情用吧。

不要使用统一码

- 字符串支持新的 Unicode 文本形式，也增加了新的正则表达式修饰符 u 来

处理码位，但是一般不要这样处理，会减低程序可读性且处理统一码速度会降低

```
// not good
'字符串'.length == 6

// 新加的: 正则支持统一码'u',  但仍建议不使用
// not good
'字符串'.match(/./u)[0].length == 6
'字符串'.codePointAt(0) == 0x20BB7
```

进制数支持

- 加入对二进制(b)和八进制(o)字面量的支持。该特性可以使用

```
// ok
0b111110111 === 503 // true
0o767 === 503 // true
```

不建议使用reflect对象和tail calls尾调用

- 没有使用的必要性