驴妈妈H5开发规范

> 程序是写给人读的，只是偶尔让计算机执行一下。 —— Donald Knuth

本规范的主要目标：

- 代码风格一致

  通过保持代码风格的一致，使代码具有良好的可读性，降低代码的维护成本，提高团队间的协作效率。

- 最佳实践

  通过遵守最佳实践，可以降低 bug 引入的可能，提高页面性能，编写出可扩展可维护的代码。

本规范的编写原则：努力兼顾代码风格与开发效率的平衡，既要努力做到 "使每一行代码都像是同一个人编写的"，又要尽量避免 "大家都觉得这样很麻烦"。

本规范的更新：本规范是当前阶段的一个总结，不是最终版，也不会有最终版，随着代码风格的优化、最佳实践的发展，本规范也会变化。

本规范的代码说明：不符合规范的代码写法，会有注释 不好，符合规范的代码写法，会有注释 好，最佳写法，会有注释 最佳；在空格相关的规范中，使用 · 号来强调表示空格。

规范最重要的是执行。

目录

如何使用

1. 通读规范文档
2. 安装 ESlint 并配置本规范对应的 共享规则
3. 在编辑器中将代码格式化

    > 以 Visual Studio Code 为例，快键键为 Alt+Shift+F。

4. 使用 ESlint 进行语法检查
5. 自动+手动修复检查出的问题

> 以 Visual Studio Code 为例，按 F1 打开命令面板输入 eslint 选择 Fix all auto-fixable Problems 可以修复所有能自动修复的问题

最终做到直接写出的代码就是符合规范的代码。

JavaScript

命名

- 避免使用单一字母命名，让你的名字具有实际含义。[eslint: id-length]

```
// 不好
function q() {
    // ...
}

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

- 使用 小驼峰命名法 来命名基本类型、对象和函数。[eslint: camelcase]

> 即使是常量也不要使用全部大写，大写的名称难以阅读。在 ES5 中由于没有 const 关键字，将一个实际不会改变的变量命名为全部大写，有助于在视觉上与普通变量加以区分，并方便语法检测工具进行检查。而在 ES6 中，这是没有必要的。

```
// 不好
const TAXFORPRODUCT = 0.9;
let this_is_my_object = {};
function c() {}

// 好
const taxForProduct = 0.9;
let thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 使用 Pascal命名法 来命名构造函数和类。[eslint: new-cap]

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

const bad = new user({
    name: 'Jack',
```

```
});

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

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

- 缩略词应该保持全部大写或者全部小写。

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

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

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

// 好
const HTTPRequests = [
    // ...
];
```

注释

- 对于类、方法、复杂的代码逻辑、从名称上难以理解的变量、正则表达式等，尽可能加上注释，以便于后期维护。

```
// 是否是由 tab 切换引起的页面滚动
let isScrollingByTab = false;

/**
 * 获取对象的指定属性
 * 属性可以是一个用点号连接的多层级路径
 * @param {object} object 对象
 * @param {string} path 属性值，可以是路径，如：'a.b.c[0].d'
 * @param {any} [defaultVal=''] 取不到属性时的默认值
 * @returns {any} 获取到的属性值
 */
```

```
function getPathValue(object, path, defaultVal = '') {
    // ...
}
```

- 单行注释使用 //，总是在需要注释的内容的上方加入注释，并在注释之前保留一个空行，除非该注释是在当前代码块的第一行。[eslint: no-inline-comments, line-comment-position, lines-around-comment]

```
// 不好
const active = true;  // 当前 tab 状态

// 好
// 当前 tab 状态
const active = true;

// 不好
function getType() {
    console.log('获取type...');
    // 设置默认 type 为 'no type'
    const type = this.type || 'no type';

    return type;
}

// 好
function getType() {
    console.log('获取type...');

    // 设置默认 type 为 'no type'
    const type = this.type || 'no type';

    return type;
}

// 好
function getType() {
    // 设置默认 type 为 'no type'
    const type = this.type || 'no type';

    return type;
}
```

- 多行注释使用 /** ... */，并尽可能遵循 jsdoc 注释规范。

> 遵循 jsdoc 规范的注释，能够很方便的自动生成 api 文档；同时，也能提升编码体验，以 Visual Studio Code 为例，当你调用函数时，会浮动提示该函数的描述、参数类型、返回值等信息。

```
// 不好
```

```
// 获取对象的指定属性
// 属性可以是一个用点号连接的多层级路径
function getPathValue(object, path, defaultVal = '') {
    // ...
}

// 好
/**
* 获取对象的指定属性
* 属性可以是一个用点号连接的多层级路径
*/
function getPathValue(object, path, defaultVal = '') {
    // ...
}

// 最佳
/**
 * 获取对象的指定属性
 * 属性可以是一个用点号连接的多层级路径
 * @param {object} object 对象
 * @param {string} path 属性值，可以是路径，如：'a.b.c[0].d'
 * @param {any} [defaultVal=''] 取不到属性时的默认值
 * @returns {any} 获取到的属性值
 */
function getPathValue(object, path, defaultVal = '') {
    // ...
}
```

- 在注释内容之前加上 FIXME: 可以提醒自己或其他开发人员这是一个需要修改的问题。

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

        // FIXME: 不应该使用一个全局变量
        total = 0;
    }
}
```

- 在注释内容之前加上 TODO: 可以提醒自己或其他开发人员这里需要一些额外工作。

> 在 Visual Studio Code 中，可以安装 TODO Highlight 扩展，支持高亮显示 FIXME:、TODO: 等特殊注释，并自动生成列表展示。

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

```
        // TODO: total 要写成可以被传入的参数修改
        this.total = 0;
    }
}
```

日志

- 在必要的地方记录日志，方便问题排查。包括：
    - try catch 语句捕获到了异常
    - ajax 调用接口成功
    - ajax 调用接口失败
    - 得到的结果与预期不符
    - 其它需要记录日志的情况
- 使用 lajax 而非 console 记录日志。

```
// 不好
try {
    const result = JSON.parse(data);
} catch(err) {
    console.error(err);
}

// 好
try {
    const result = JSON.parse(data);
} catch(err) {
    logger.error(`接口返回结果转换为JSON时出错：${err}`);
}
```

变量

- 总是使用 const 来定义常量，避免使用 var。[eslint: prefer-const, no-const-assign]

> 这能够确保你无法对常量重新赋值，在开发阶段就发现可能的问题。

```
// 不好
var num1 = 1;
var num2 = 2;

// 好
const num1 = 1;
const num2 = 2;
```

- 如果你需要为引用重新赋值，使用 let 而不是 var。[eslint: no-var]

> let 是块作用域，var 是函数作用域。

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

// 好
let count = 1;
if (true) {
    count += 1;
}

// 不好
for (var i = 0; i < 3; i++) {
    // ...
}

// 好
for (let i = 0; i < 3; i++) {
    // ...
}
```

- 请注意：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;
}

// 引用错误：a未定义
console.log(a);

// 引用错误：b未定义
console.log(b);
```

- 变量声明不要省略关键字。[eslint: no-undef]

```
// 不好
cat = new Cat();

// 好
const cat = new Cat();

// 好
```

```
window.cat = new Cat();
```

- 不要使用链式声明。[eslint: no-multi-assign]

> 链式声明会创建隐式全局变量，可能引入 bug。

```
// 不好
(function example() {
    /**
    * JavaScript 会将这行代码解析为：
    * let a = (b = (c = 1));
    * let 关键字仅作用于变量 a，变量 b 和 c 将定义于全局作用域。
    */
    let a = b = c = 1;
}());

// 出错：a 未定义
console.log(a);

// 输出 1
console.log(b);

// 输出 1
console.log(c);

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

// 出错：a 未定义
console.log(a);

// 出错：b 未定义
console.log(b);

// 出错：c 未定义
console.log(c);
```

- 不要使用 魔术数字，应该使用意义明确的常量代替。[eslint: no-magic-numbers]

> 这会使代码更加可读且易于重构。

```
// 不好
const finalPrice = originalPrice * 0.8;
```

```
// 好
const tax = 0.8;
const finalPrice = originalPrice * tax;
```

字符串

- 字符串使用单引号 '' 。[eslint: quotes]

```
// 不好
const name = "Kathy";

// 不好 - 仅当字符串中需要插入变量或换行时才使用模板字符串
const name = `Kathy`;

// 好
const name = 'Kathy';
```

- 对于长字符串不要人为分割。

> 很多规范都不允许字符串过长，推荐进行字符串拼接分割，但事实上，长字符串分割之后将难以维护，并可能导致关键字搜索无效（关键字被分割了）；而长字符串难以阅读的问题，使用现代编辑器就可以解决，以 Visual Studio Code 为例：点击查看 -> 切换自动换行。

```
// 不好
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.';

// 不好
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.';

// 好
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.';
```

- 构建包含变量或换行的字符串时，使用模板字符串而不是字符串拼接。[eslint: prefer-template, template-curly-spacing]

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

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

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

- 不要对字符串使用 eval() 方法，它可能会导致很多漏洞。[eslint: no-eval]

```
// 不好
const result = eval('(function() { const a = 1; return a; }());');
```

- 避免无意义的转义符 \。[eslint: no-useless-escape]

> 转义符会降低可读性，应该仅在必要时才存在。

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

// 好 - 仅 this 前后的引号才需要转义
const foo = '\'this\' is "quoted"';
```

对象

- 使用简洁语法创建对象。[eslint: no-new-object]

```
// 不好
const item = new Object();

// 好
const item = {};
```

- 使用对象方法的简写。[eslint: object-shorthand]

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

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

// 好
const atom = {
    value: 1,

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

- 使用属性值的简写。[eslint: object-shorthand]

```
const name = 'Kathy';

// 不好
const obj = {
    name: name,
};

// 好
const obj = {
    name,
};
```

- 仅在无效的属性名上加引号。[eslint: quote-props]

> 一般来说，我们认为这在主观上更容易阅读。它改进了语法高亮，并且更容易被许多JS引擎优化。

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

// 好
const good = {
```

```
    foo: 3,
    bar: 4,
    'data-blah': 5,
};
```

- 不要直接使用 Object.prototype 上的方法，例如 hasOwnProperty，isPrototypeOf 等。[eslint: no-prototype-builtins]

> 这些方法可能会被对象的自定义属性覆盖，例如对象：{hasOwnProperty: false}，或者当该对象的 __proto__ 属性为 null 时，例如使用 Object.create(null) 创建的对象，就不存在这些方法。

```
// 不好
console.log(obj.hasOwnProperty(key));

// 好
console.log(Object.prototype.hasOwnProperty.call(obj, key));
```

数组

- 使用简洁语法创建数组。[eslint: no-array-constructor]

```
// 不好
const items = new Array();

// 好
const items = [];
```

- 使用 Array#push 在数组末尾添加元素。

```
const someStack = [];

// 不好
someStack[someStack.length] = 'abc';

// 好
someStack.push('abc');
```

- 使用 扩展运算符 ... 来复制数组。

> 注意：这些示例写法都是浅拷贝，不是深拷贝。

```
const items = [1, 2, 3];
```

```
// 不好
const itemsCopy = [];
for (let i = 0; i < items.length; i++) {
    itemsCopy[i] = items[i];
}

// 好
const itemsCopy = items.slice();

// 好
const itemsCopy = Array.from(items);

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

- 使用 Array.from 将类数组对象（arguments、NodeList 等）转换为标准数组。

```
// 不好
const divs =
Array.prototype.slice.call(document.querySelectorAll('div'));

// 好
const divs = Array.from(document.querySelectorAll('div'));

// 好
function something() {
    const args = Array.from(arguments);
    // ...
}
```

函数

- 不要在非函数代码块（if、while 等）的内部定义函数。[eslint: no-loop-func]

```
// 不好
if (true) {
    function test() {
        console.log(111);
    }
    // ...
}

// 好
let test;
if (true) {
    test = () => {
        console.log(111);
    };
```

```
    // ...
}
```

- 将自执行函数用括号包起来。[eslint: wrap-iife]

```
// 不好
const x = function() {
    console.log(1);
}();

// 不好
(function() {
    console.log(1);
})();

// 好
(function() {
    console.log(1);
}());
```

- 不要将参数命名为 arguments，这将覆盖掉传入函数作用域的 arguments 对象。

```
// 不好
function foo(arguments) {
    // ...
}

// 好
function foo(args) {
    // ...
}
```

- 不要使用 arguments 对象，可以用 剩余参数 替代。[eslint: prefer-rest-params]

> 剩余参数明确表示函数会接收一系列参数，在参数列表中就能看到，不像 arguments 那样隐晦；另外，剩余参数是标准数组，而 arguments 是类数组对象，要使用数组的方法还要先转为数组。

```
// 不好
function joinAll() {
    const args = Array.from(arguments);
    return args.join('');
}

// 好
function joinAll(...args) {
```

```
    return args.join('');
}
```

- 使用 默认参数 语法而不是在函数内部处理默认值。

```
// 不好
function handleThings(opts) {
    // 如果 opts 传值为 false，这里就会错误的重新指定默认值
    opts = opts || {};

    // ...
}

// 不好
function handleThings(opts) {
    if (typeof opts === 'undefined') {
        opts = {};
    }

    // ...
}

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

- 将默认参数放置于最后。

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

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

- 不要使用 Function 构造函数生成一个函数。[eslint: no-new-func]

> 用这种方式生成函数来计算字符串类似于 eval，可能造成很多漏洞。

```
// 不好
const add = new Function('a', 'b', 'return a + b');
```

- 使用 扩展运算符 ... 给函数传参。[eslint: prefer-spread]

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

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

箭头函数

- 当你使用一个函数表达式（或者传递一个匿名函数）时，请使用 箭头函数。[eslint: prefer-arrow-callback, arrow-spacing]

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

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

- 避免可能让人混淆箭头函数符号 => 和比较运算符 >=、<= 的代码。[eslint: no-confusing-arrow]

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

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

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

类

- 总是使用 class，避免直接操作 prototype。

  > 这是因为 class 更简洁易懂。

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

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

- 使用 extends 实现继承。

```
class Parent {
    constructor(name) {
        this.name = name;
    }

    print() {
        console.log(this.name);
    }
}

// 不好
function Child(name) {
    this.name = name;
}

Child.prototype = Object.create(Parent.prototype);

// 好
class Child extends Parent {}
```

- 如果类的方法没有显式返回值，建议返回 this 。

> 这样可以帮助构建方法链，实现链式调用。

```
// 好
class Div {
    setHeight(height) {
        this.height = height;
    }

    setWidth(width) {
        this.width = width;
    }
}

const div = new Div();

// 返回 undefined
div.setHeigth(200);

// 返回 undefined
div.setWidth(300);

// 最佳
class Div {
    setHeight(height) {
        this.height = height;
        return this;
    }

    setWidth(width) {
        this.width = width;
        return this;
    }
}

const div = new Div();
div.setHeight(200).setWidth(300);
```

- 如果没有给类显式指定构造函数，那它会有一个预设的默认构造函数。一个空的或者仅仅委派给父类的构造函数是不必要的。[eslint: no-useless-constructor]

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

    getName() {
        return this.name;
    }
```

```
}

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

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

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

> 类的重复成员会默认最后一个为有效，重复成员多数情况下都是 bug。

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

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

属性

- 访问多层级属性时，每层属性都需要判断是否存在。

```
const data = {
    code: 1,
    data: {
        list: [{
            productId: '001'
        }]
    }
}

// 不好 - 当 list 是一个空数组时就会报错
const productId = data.data.list[0].productId;
```

```
// 不好 - 过多的 try catch 会影响性能
try {
    const productId = data.data.list[0].productId;
} catch (e) {
    const productId = '未命名';
    console.error('获取属性 productId 时出错! ');
}

// 好 - 确保不会出现异常，任意一次的判断结果为 false 则 productId = undefined
const productId = data && data.data && data.data.list &&
data.data.list[0] && data.data.list[0].productId;

// 最佳 - 采用我们获取属性值的公共方法
const productId = commonUtil.getPathValue(data, 'data.list[0].productId',
'未命名');
```

- 使用 . 号访问属性，仅对无效的属性名才使用方括号。[eslint: dot-notation]

```
const person = {
    age: 20,
    'first-name': 'Ada'
}

// 不好
const age = person['age'];

// 好
const age = person.age;
const firstName = person['first-name'];
```

模块

- 不要在多个地方导入同一路径。[eslint: no-duplicate-imports]

> 这会让代码难以维护。

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

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

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

- 将所有的 import 放置于顶部。[eslint: import/first]

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

import bar from 'bar';

// 好
import foo from 'foo';
import bar from 'bar';

foo.init();
```

迭代器

- 避免使用迭代器，尽量用 JavaScript 的高阶函数来替代类似 for-in、for-of 这样的循环。[eslint: no-iterator, no-restricted-syntax]

> 使用 forEach、some、every、filter、map、reduce、find、findIndex 等处理一个数组；使用 Object.keys 生成对象属性的数组。

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

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

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

// 最佳
const sum = numbers.reduce((total, num) => total + num, 0);

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

// 好
```

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

// 最佳
const increasedByOne = numbers.map(num => num + 1);
```

- 如果你一定要使用 for in，必须判断遍历的属性是否是对象自身的属性。[eslint: guard-for-in]

  > 这是因为在使用 for in 遍历对象时，会把从原型链继承来的属性也包括进来，这样会导致意想不到的项出现。

```
// 不好
for (let key in foo) {
    doSomething(key);
}

// 好
for (let key in foo) {
    if (Object.prototype.hasOwnProperty.call(foo, key)) {
        doSomething(key);
    }
}
```

条件语句

- 根据实际情况，恰当选用 if 或 switch 来构建条件语句。

  一般来说，下面几种情况适合使用 switch：

  - 枚举表达式的值。这种枚举是可以期望的、平行逻辑关系的。
  - 表达式的值是固定的，不是动态变化的。
  - 表达式的值是有限的，而不是无限的。
  - 表达式的值一般为整数、字符串等类型的数据。

  而 if 结构更适合这些情况：

  - 具有复杂的逻辑关系。
  - 表达式的值具有线性特征，如对连续的区间值进行判断。
  - 表达式的值是动态的。
  - 测试任意类型的数据。

```
// 好
let msg;
if (score < 60) {
    msg = '不及格';
} else if (score >= 60 && score < 75) {
    msg = '合格';
} else if (score >= 75 && score < 90) {
    msg = '良好';
} else {
    msg = '优秀';
}

// 好
let msg;
switch (sex) {
    case '女':
        msg = '女士';
        break;
    case '男':
        msg = '先生';
        break;
    default:
        msg = '请选择性别';
}
```

- 所有的 switch 语句都要包含 default 情况，即使 default 内容为空。[eslint: default-case]

> 总是明确说明默认情况是什么，可以提醒开发人员是否忘记了去处理默认情况，让逻辑更严谨。

```
// 不好
switch (code) {
    case '1':
        // ...
        break;
}

// 好
switch (code) {
    case '1':
        // ...
        break;
    default:
        // ...
}

// 好 - 默认情况就是什么都不做
switch (code) {
    case '1':
        // ...
```

```
        break;
    default:
}
```

运算符

- 使用 === 和 !== 进行比较运算。[eslint: eqeqeq]

```
// 不好
if (code == '1') {
    // ...
}

// 好
if (code === '1') {
    // ...
}
```

- 避免不必要的三元运算符。[eslint: no-unneeded-ternary]

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

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

类型转换

- 转换为字符串。

```
const num = 9;

// 不好
const score = num + '';

// 不好 - num 不一定有 toString 方法，比如：num 为 null
const score = num.toString();

// 好
const score = String(num);
```

- 转换为数字。

```
const inputValue = '4';

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

// 不好
const val = +inputValue;

// 不好
const val = inputValue >> 0;

// 好
const val = parseInt(inputValue);

// 好
const val = Number(inputValue);
```

- 转换为布尔值。

```
const age = 0;

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

// 好
const hasAge = Boolean(age);

// 最佳
const hasAge = !!age;
```

- 条件语句如 if 会自动进行类型转换，并遵循如下规则：
  - 字符串：空字符串转换为 false，否则为 true
  - 数字：+0，-0，NaN 转换为 false，否则为 true
  - 布尔值：返回原布尔值
  - undefined：转换为 false
  - null：转换为 false
  - 对象：转换为 true

```
const obj = {};
const arr = [];

// 不好 - 对象、数组始终为 true
if (obj) {
```

```
    // ...
}

if (arr) {
    // ...
}

// 好
if (Object.keys(obj).length) {
    // ...
}

if (arr.length) {
    // ...
}
```

空格

- 使用 4 个空格作为缩进。[eslint: indent]

  > 主流的编辑器一般都支持设定 Tab 对应的空格数，以 Visual Studio Code 为例，设置方式：点击文件 -> 首选项 -> 设置，搜索 "editor.tabSize"。

```
// 不好
function foo() {
·let name;
}

// 不好
function bar() {
··let name;
}

// 好
function foo() {
····let name;
}
```

- 不要混用 Tab 和空格。[eslint: no-mixed-spaces-and-tabs]

  > 这可能会导致一些格式上的异常，例如：在 Jade 中混用 Tab 和空格就会出错。

- 在左大括号之前加 1 个空格。[eslint: space-before-blocks]

```
// 不好
function test(){
    console.log('test');
}
```

```
// 好
function test()·{
    console.log('test');
}
```

- 在流程控制语句（如 if, while 等）的左小括号之前加 1 个空格。[eslint: keyword-spacing]

```
// 不好
if(true) {
    done();
}

// 好
if·(true) {
    done();
}
```

- 将 else 放在与前面 } 同一行并加 1 个空格。[eslint: keyword-spacing]

```
// 不好
if (true) {
    // ...
}else {
    // ...
}

// 不好
if (true) {
    // ...
}
else {
    // ...
}

// 好
if (true) {
    // ...
}·else {
    // ...
}
```

- 在函数的定义和调用中，函数名与参数列表之间不要有空格。[eslint: keyword-spacing]

```
// 不好
function test·() {
```

```
    console.log·('test');
}

// 好
function test() {
    console.log('test');
}
```

- 运算符之间使用空格隔开。[eslint: space-infix-ops]

```
// 不好
const x=y+5;

// 好
const x·=·y·+·5;
```

- 在文件的末尾加上一行空白行。[eslint: eol-last]

```
// 不好
import { es6 } from './test';
    // ...
export default es6;
```

```
// 不好
import { es6 } from './test';
    // ...
export default es6;↵
↵
```

```
// 好
import { es6 } from './test';
    // ...
export default es6;↵
```

- 多个方法（大于 2 个）形成的方法链调用，从第 1 或第 2 个方法开始换行调用，并将 . 号置于行首以说明该行是方法调用，而不是开始一个新的语句。[eslint: newline-per-chained-call]

```
// 好
const result1 = data.replace('[str1]', 'str1').replace('[str2]', 'str2');

// 不好
```

```
const result2 = data.replace('[str1]', 'str1').replace('[str2]',
'str2').replace('[str3]', 'str3');

// 好
const result3 = data
    .replace('[str1]', 'str1')
    .replace('[str2]', 'str2')
    .replace('[str3]', 'str3');

// 好
const result4 = data.replace('[str1]', 'str1')
    .replace('[str2]', 'str2')
    .replace('[str3]', 'str3');
```

- 不要在语句块的开始和结尾处放置空行。[eslint: padded-blocks]

```
// 不好
function bar() {

    console.log(foo);

}

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

- 不要在小括号内的两侧放置空格。[eslint: space-in-parens]

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

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

// 不好
if ( foo ) {
    console.log(foo);
}

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

```
}
```

- 不要在中括号内的两侧放置空格。[eslint: array-bracket-spacing]

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

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

- 在大括号内的两侧放置空格。[eslint: object-curly-spacing]

```
// 不好
const person = {name: 'Kathy'};

// 好
const person = { name: 'Kathy' };
```

- 对象的键和冒号间不加空格，冒号和值间加空格。[eslint: object-curly-spacing]

```
// 不好
const foo = {
    key1 : value1,
    key2:value2,
    key3 :value3,
};

// 好
const foo = {
    key1: value1,
    key2: value2,
    key3: value3,
};
```

- 逗号之前不加空格，逗号之后加空格。[eslint: comma-spacing]

```
// 不好
const foo = 1,bar = 2;
const arr = [1 ,2];
const obj = {'foo': 'bar' ,'baz': 'qur'};
foo(a,b);
```

```
new Foo(a·,b);
function foo(a·,·b){}
a,b

// 好
const foo = 1,·bar = 2;
const arr = [1,·2];
const obj = {'foo': 'bar',·'baz': 'qur'};
foo(a,·b);
new Foo(a,·b);
function foo(a,·b){}
a,·b
```

- 在注释内容之前加 1 个空格。[eslint: spaced-comment]

```
// 不好
//这里控制是否有效
const active = true;

// 好
//·这里控制是否有效
const active = true;

// 不好
/**
*make() 返回一个新的元素
*基于传进来的标签名称
*/
function make(tag) {

    // ...

    return element;
}

// 好
/**
*·make() 返回一个新的元素
*·基于传进来的标签名称
*/
function make(tag) {

    // ...

    return element;
}
```

逗号

- 不要将逗号放在前面。[eslint: comma-style]

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

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

- 多行情况下，建议在最后一行的参数或属性的末尾添加一个额外逗号。[eslint: comma-dangle]

> 这会方便你新增或删除项，并让 git 的差异列表更清晰。

```
// 不好 - 事实上只新增了一个属性，但 git 差异列表会显示有 3 行改动
const hero = {
     firstName: 'Florence',
-    lastName: 'Nightingale'
+    lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing']
};

// 好 - git 差异列表会展示 1 行改动
const hero = {
     firstName: 'Florence',
     lastName: 'Nightingale',
+    inventorOf: ['coxcomb chart', 'modern nursing'],
};
```

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

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

// 好
```

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

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

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

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

// 好（注意：可变参数后面不可以有逗号）
function createHero(
    firstName,
    lastName,
    inventorOf,
    ...heroArgs
) {
    // does nothing
}

// 好（因为该条规则仅适用于多行）
function createHero(firstName, lastName, inventorOf) {
    // does nothing
}
```

分号

- 语句的结尾处必须要有分号。[eslint: semi]

```
// 不好
const name = 'ESLint'
```

```
object.method = () => {
    // ...
}

// 好
const name = 'ESLint';

object.method = () => {
    // ...
};
```

调试

- 不要使用 debugger 语句进行调试。[eslint: no-debugger]

> 现代开发工具可以通过设置断点的方式进行调试，无须修改源码。

```
// 不好
function test(num) {
    debugger;
    return Boolean(num);
}

// 好
function test(num) {
    // 在下面这行设置断点
    return Boolean(num);
}
```

- 在部署到生产环境之前移除项目中的 console、alert 等调试代码。[eslint: no-console, no-alert]

编码

- 使用 encodeURIComponent 而不是 escape 对 URL 中的汉字进行编码。[eslint: no-escape（自定义规则）]

> escape, encodeURI, encodeURIComponent 三者区别可以查看这里。

```
// 不好
const keyword = escape('上海');

// 好
const keyword = encodeURIComponent('上海');
```

HTML

命名

- 标签名全部小写。

```
<!--不好-->
<DIV>这是一个块级元素</DIV>

<!--好-->
<div>这是一个块级元素</div>
```

- 属性名全部小写，使用 - 作为分隔符。

```
<!--不好-->
<div Class="contain" itemCode="1"></div>

<!--好-->
<div class="contain" item-code="1"></div>
```

- 样式类全部小写，使用 - 作为分隔符。

```
<!--不好-->
<button class="btnPrimay btn_outline">选择城市</button>

<!--好-->
<button class="btn-primay btn-outline">选择城市</button>
```

空格

- 使用 4 个空格作为缩进。

> 主流的编辑器一般都支持设定 Tab 对应的空格数，以 Visual Studio Code 为例，设置方式：点击文件 -> 首选项 -> 设置，搜索 "editor.tabSize"。

```
<!--不好-->
<div>
··<span>这是一行文本</span>
</div>

<!--好-->
<div>
····<span>这是一行文本</span>
</div>
```

- 不要混用 Tab 和空格。

  > 这可能会导致一些格式上的异常，例如：在 Jade 中混用 Tab 和空格就会出错。

- 标签内的文本避免使用空格，应该使用样式来实现。

```
<!--不好-->
<p>
      这是一些文本
</p>

<!--好-->
<p style="text-indent: 2em;">
    这是一些文本
</p>

<!--不好-->
<div>￥ 1000</div>

<!--好-->
<div>￥<span style="margin-left: 5px">1000</span></div>
```

引号

- 属性值的最外层使用双引号。

```
<!--不好-->
<div class=container></div>

<!--不好-->
<div class='container'></div>

<!--好-->
<div class="container"></div>
```

标签

- 不要在自闭合标签的结尾使用 / 。

```
<!--不好-->
<img src="logo.png" alt="company"/>
<br/>

<!--好-->
<img src="logo.png" alt="company">
<br>
```

- 不要省略可选的关闭标签。

> HTML5规范 指出一些标签可以省略关闭标签，但我们认为这样做降低了可读性。

```
<!--不好-->
<body>
    <ul>
        <li>111
        <li>222
        <li>333
    </ul>

<!--好-->
<body>
    <ul>
        <li>111</li>
        <li>222</li>
        <li>333</li>
    </ul>
</body>
```

属性

- 引入 css 和 js 时不需要指明 type 属性。

> text/css 和 text/javascript 分别是它们的默认值。

```
<!--不好-->
<link type="text/css" rel="stylesheet" href="global.css">
<script type="text/javascript" src="public.js"></script>

<!--好-->
<link rel="stylesheet" href="global.css">
<script src="public.js"></script>
```

- 布尔属性不需要指明属性的值。

> 布尔属性存在代表取值为 true，属性不存在代表取值为 false。

```
<!--不好-->
<input type="text" disabled="disabled">
<input type="checkbox" value="1" checked="checked">

<!--好-->
<input type="text" disabled>
<input type="checkbox" value="1" checked>
```

- 尽量将 class、id 等使用频率高的属性放在前面。

```
<!--不好-->
<input type="text" placeholder="请输入关键字" id="search-input" disabled
class="form-control">

<!--好-->
<input class="form-control" id="search-input" type="text" placeholder="请
输入关键字" disabled>
```

- 使用 data-* 创建标准化的自定义属性。

> 这是创建自定义属性的标准方式；加上 data- 前缀能够明确表示这是一个自定义属性，而非原生属性；另外，在 JavaScript 中还可以通过 document.querySelector(selector).dataset 方便的访问它们。

```
<!--不好-->
<div class="container" code="freetour"></div>

<!--好-->
<div class="container" data-code="freetour"></div>
```

注释

- 对于重要的元素，尽可能加上注释，以便于后期维护。

```
<!--筛选面板上方的加载中动画，主要用于：1. 点击筛选项，2. 点击清空筛选 时显示-->
<div id="loading2" style="display:none;">
    <div></div>
</div>

<!--筛选面板上方的提示语，主要用于无筛选数据时显示-->
<div id="filterTip" style="display:none;">
    <span>该筛选项无数据，已恢复至默认选项！</span>
</div>
```

CSS

分离

- 对于较简单的规则，尽量分离成通用样式，面向属性，而非面向业务。

> 面向业务的css设计，将导致大量的样式重复定义，难以维护；而面向属性的设计能大大提高重用

> 性，也有利于统一维护修改。

```
/* 不好 */
/* 用户登录按钮 */
.btn-user-login {
    background-color: #d30779;
    text-align: center;
    padding-bottom: 10px;
}

/* 好 */
.background-highlight {
    background-color: #d30779;
}

.text-center {
    text-align: center;
}

.p-b-10 {
    padding-bottom: 10px;
}
```

空格

- 使用 4 个空格作为缩进。

> 主流的编辑器一般都支持设定 Tab 对应的空格数，以 Visual Studio Code 为例，设置方式：点击文件 -> 首选项 -> 设置，搜索 "editor.tabSize"。

```
/* 不好 */
p {
··color: #555;
}

/* 好 */
p {
····color: #555;
}
```

- 在 { 前加 1 个空格。

```
/* 不好 */
p{
    color: #555;
}
```

```
/* 不好 */
p
{
    color: #555;
}

/* 好 */
p·{
    color: #555;
}
```

- 将 } 放置于新行。

```
/* 不好 */
p {
    color: #555;}

/* 好 */
p {
    color: #555;
}
```

- 在属性值前加 1 个空格。

```
/* 不好 */
p {
    color:#555;
}

/* 好 */
p {
    color:·#555;
}
```

- 在属性值中的 , 后面加 1 个空格。

```
/* 不好 */
.element {
    background-color: rgba(0,0,0,.5);
}

/* 好 */
.element {
    background-color: rgba(0,·0,·0,·.5);
}
```

- 在注释的 /* 后和 */ 前各加 1 个空格。

```
/* 不好 */
/*这是一行注释*/
p {
    color: #555;
}

/* 好 */
/*·这是一行注释·*/
p {
    color: #555;
}
```

- 属性值中的 (前不要加空格。

```
/* 不好 */
.element {
    background-color: rgba·(0, 0, 0, .5);
}

/* 好 */
.element {
    background-color: rgba(0, 0, 0, .5);
}
```

- 选择器 >、+、~、: 等前后都不要加空格。

```
/* 不好 */
div·>·p {
    color: #555;
}

div·:·after {
    content: "";
}

/* 好 */
div>p {
    color: #555;
}

div:after {
    content: "";
}
```

- 各规则之间保留一个空行。

```
/* 不好 */
p {
    color: #555;
}
.element {
    background-color: rgba(0, 0, 0, .5);
}

/* 好 */
p {
    color: #555;
}

.element {
    background-color: rgba(0, 0, 0, .5);
}
```

- 在规则声明中有多个选择器时，每个选择器独占一行。

```
/* 不好 */
div, p, span {
    color: #555;
}

/* 好 */
div,
p,
span {
    color: #555;
}
```

引号

- 字符串类型的值使用双引号。

```
/* 不好 */
.element:after {
    content: '';
    background-image: url(logo.png);
}
```

```
/* 好 */
.element:after {
    content: "";
    background-image: url("logo.png");
}
```

小数

- 不要设置小数作为像素值。

> 像素值为小数时会被解析成整数，且各浏览器解析方式存在差异。

```
/* 不好 */
div {
    width: 9.5px;
}

/* 好 */
div {
    width: 10px;
}
```

- 去掉小数点前面的 0。

```
/* 不好 */
.element {
    color: rgba(0, 0, 0, 0.5);
}

/* 好 */
.element {
    color: rgba(0, 0, 0, .5);
}
```

零值

- 属性值为 0 时，后面不要加单位。

```
/* 不好 */
.element {
    width: 0px;
}

/* 好 */
.element {
```

```
    width: 0;
}
```

- 定义无边框的样式时，将属性值设为 0 而不是 none。

```
/* 不好 */
div {
    border: none;
}

/* 好 */
div {
    border: 0;
}
```

注释

- 对于较生僻的样式、解决特定问题的样式、需要特别注明的样式等，尽可能加上注释，以便于后期维护。

```
button {
    /* 移除iOS下的原生样式 */
    -webkit-appearance: none;
}

.modal {
    /* 蒙版的层级必须大于700，以放在列表之上；同时必须小于900，以放在导航之下 */
    z-index: 701;
}
```

- 总是使用 /* ... */ 创建标准注释，即便是在 sass 中也不要使用 // 。

性能相关

- html 加载时间不超过 1s（Good3G）
- 白屏时间不超过 1.5s（Good3G）
- 首屏时间不超过 2s（Good3G）
- 页面加载时间不超过 5s（Good3G）
- 页面大小不超过 1M
- 图片使用合适尺寸，且不超过 100KB
- 手机上打开页面，没有明显卡顿

- 避免重定向

页面相关

整体结构

- 页面 html 模板如下：

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <meta content="initial-scale=1.0,user-scalable=no,maximum-
scale=1,width=device-width" name="viewport">
        <meta http-equiv="Cache-Control" content="no-transform" />
        <meta http-equiv="Cache-Control" content="no-siteapp" />
        <meta http-equiv="pragma" content="no-cache">
        <meta http-equiv="expires" content="0">
        <meta name="format-detection" content="telephone=no"/>
        <meta name="apple-mobile-web-app-status-bar-style" content="black" />
        <meta name="apple-mobile-web-app-capable" content="yes" />
        <meta name="applicable-device" content="mobile">
        <link rel="apple-touch-icon" href="
//pics.lvjs.com.cn/img/mobile/touch/img/lvmama_v1_png.png">
        <link rel="apple-touch-icon" sizes="72x72" href="
//pics.lvjs.com.cn/img/mobile/touch/img/lvmama_v1_png.png ">
        <link rel="apple-touch-icon" sizes="114x114" href="
//pics.lvjs.com.cn/img/mobile/touch/img/lvmama_v1_png.png ">
        <link rel="apple-touch-icon" sizes="144x144" href="
//pics.lvjs.com.cn/img/mobile/touch/img/lvmama_v1_png.png ">
        <link rel="stylesheet" type="text/css"
href="//pics.lvjs.com.cn/mobile/lib/css/common.css"/>
        <link rel="stylesheet" type="text/css" href="业务css地址"/>
        <title>驴妈妈无线官网-景区门票_自助游_旅游度假_酒店预订</title>
        <meta name="Keywords" content="驴妈妈无线,景区门票,旅游度假"/>
        <meta name="Description" content="驴妈妈无线官网:支持手机快捷预订景区门票,
自助游线路,全国酒店,豪华邮轮等各种旅游产品.(中国最领先的在线旅游预订服务商!) "/>
    </head>
    <body>
        <!--正文 code here-->
        <script type="text/javascript"
src="//pics.lvjs.com.cn/mobile/lib/plugins/public/1.0/public.min.js"></script>
        <script type="text/javascript" src="插件或业务js地址"></script>
    </body>
</html>
```

> TDK：专题找运营产品提供，其他找陈程或对应产品提供，如不提供写如上默认 TDK。

- CSS：css 在 head 部分引用，业务 css 前需引用全局 css (common.css)，不要使用 minify 合并请求；

- JS：js 在 body 闭合标签前引用，插件、业务 js 前需引用全局 js (public.min.js)，不要使用 minify 合并请求；
- 资源文件 css、js、image 需**压缩**且使用 **pics.lvjs.com.cn** 域名引用，
  如：//pics.lvjs.com.cn/mobile/node_pro/public/min/js/mHomePage/proList.js 。

SEO

> 页面开发前，先和产品确认有无 SEO 需求，若有则需后端渲染，并遵循以下规则。

- 首屏必须后端渲染，不能使用 ajax；
- 能使用异步 js 的使用异步，如：<script async src="js地址"></script> ；
- html 及其他资源文件中删除不必要的注释；
- 页面的产品标题用 h1 标签，副标题用 h2、h3 标签；
- 图片使用 img 标签，并使用 alt 属性，避免使用 background，如：<img src="图片地址" alt="上海欢乐谷"> ；
- 链接不用 javascript控制，用标准的超链接，如：链接文字 ；
- 提测后，即让产品通知 SEO 验收，并跟进 SEO 验收情况，及时修复问题；
- 当SEO提出对线上链接做修改的需求时，注意让SEO产品和分销产品确认是否涉及分销推广链接，如涉及，则需分销测试回归；

头部

- 如无特殊要求，业务类页面头部使用 lvheader 插件，专题使用 ztheader 插件。

定位

- 城市数据的获取和赋值统一使用前后端的定位选择插件 positionUtil（location），实现整站城市的数据流转（特殊城市列表除外）。

统计

- 业务类布码，在调用统计方法之前引入 statisticsUtil-x.x.min.js ；
- 专题类布码，在页面底部引入 mlosc.js 。

多页加载

- 对于超过两屏的页面，需在页面底部异步引用 toTop 插件；
- 加载下一页数据时，需使用 toTop 插件中相应的类名展示不同的状态（lvLoading-over、lvLoading-hide）；
- 列表子元素需新增 lvAddBgcolor 类名，点击时会有统一底色（需引用 toTop 插件）。

loading

> 使用方法：调用接口时，传 loadingType: '' 参数，即可自定义使用不同加载效果（需引用 public.min.js）。

- 主接口请求中： 页面主接口请求开始到返回前，显示小驴转转转加载动画；
  页面部分模块数据请求中，对应位置显示 "三个红点" 加载动画；

- tab 切换接口请求中：
  当点击 tab 后，对应列表数据未返回时，应设置一定的高度，防止页面上滑，并显示 "三个红点" 加载动画，使用方法同上。

图片

- 图片样式自适应
  - img：width 百分比，height 自适应；
  - div：background + padding-bottom 。
- 有超过一屏的图片展示，需使用 lazyload 插件，实现懒加载；
- 产品图片背景
  - 背景统一为灰色小驴，居中显示（使用 common.css 相应类名）；
  - 图片元素的外层需嵌套一个盒子。

```
    <div class="img-wrap">
        <img src="http://pic.lvmama.com/pics//uploads/pc/place2/2017-06-
23/294d8e11-7cb7-4e46-a8e5-96c237775de9_480_320.jpg">
        <div class="price"><p>¥<i>170</i>起</p></div>
    </div>
```

- 产品图片尺寸后缀
  - java 接口返回的图片路径无尺寸后缀时（如 480_320），建议对应接口开发添加尺寸后缀，如 http://pic.lvmama.com/pics//uploads/pc/place2/2017-06-23/294d8e11-7cb7-4e46-a8e5-96c237775de9_480_320.jpg ； *php 接口对应的 cms 后台已对上传图片尺寸和大小做限制，故无需后缀*
  - 100% 宽度图片建议使用宽 720 以上后缀，50% 使用 480 以上；30% 使用 300 以上；25% 使用 200 以上；
  - 图库支持尺寸如下：
    1200_480|720_540|480_320|360_270|300_200|200_150|200_80|180_120|121_91|1280_|960_|720_|480_|1080_432|530_353|347_231 。
- 防止轮播图平铺

  > 在网速慢的情况下，swiper 里面的图片会出现平铺显示的情况

  需在 swiper 外层盒子限制其高度（padding-bottom 或 "固定高度"）。
- 不要使用雪碧图

分享

- app 分享
  内嵌 app 的页面如果想具有 app 的分享功能，必须具有 share meta，

如：<meta id="share" share-linkurl="https://zt1.lvmama.com/template4/index/2255" name="share" share-title="暑期热爱季【驴妈妈】" share-content="驴妈妈暑期热爱季，千万红包狂撒一夏！最高立减2000元！" share-imageurl="http://pics.lvjs.com.cn/pics/allin/back/201706/1498031694711.jpg">

- 微信 (QQ) 分享

> 如果页面中存在 share meta，且没有显式调用 share 自定义分享方法，share 插件会默认使用 share meta 中的内容作为微信分享的内容。

  - 业务类页面使用 share 插件，专题类页面使用 newBase(本质还是调用了 share 插件)，可自定义微信 (QQ) 分享图片、标题、描述等；
  - 分享图片，300*300 以上图片（如不自定义设置，微信会默认取页面中第一张大于 300*300 的图片）；
  - 分享标题，字数限制待定（如不自定义设置，微信会默认取页面 title 内容）；
  - 分享描述，不得超过 25 个中文字符（50 个英文字符），若超过 share 插件将会进行截取（如不自定义设置，微信会默认取页面 url 进行显示）。

错误处理

- 主接口超时，待定；
- 主接口 500，待定；
- 接口返回数据为空，小灰驴+产品定制化文字；
- 接口-1，弹出错误信息黑框；
- 接口-2，弹出错误信息黑框。

其他

- 全站所有链接使用 https；
- 调用 php 的 cms 运营配置类接口，需使用 cmsUtil 插件，将运营配置信息转化为最终链接；
- 频道、列表、详情类页面需在页面底部异步引用 downloadBar 下载条插件；
- 接口请求方式：订单相关提交数据接口使用post请求，其他均使用get请求；

ESlint

在 eslint 子目录 中可以找到本规范 JavaScript 部分对应的 eslint 共享规则。

参考

本规范的制定主要参考了这些文档：

- Airbnb JavaScript Style Guide
- Airbnb CSS Style Guide
- Alloyteam 代码规范
- 精简高效的CSS命名准则

相关资源

ES6

- 阮一峰：ECMAScript 6 入门
- es6 特性概述
- es6 兼容性列表

工具

- 推荐使用 Visual Studio Code 作为开发和调试工具
- 由于规范文档和 eslint 规则 都是 es6 优先，可以使用 ES5 to ES6 插件方便地将 ES5 源码一键转换为 ES6
- 在 VSCode 中安装 ESlint 插件，可以实时显示检查出的问题，配置方法见 这里
- TODO Highlight 插件能够高亮显示 TODO:、FIXME: 这样的特殊代码注释

其它代码规范

- Airbnb 代码规范
- Alloyteam 代码规范
- Google 代码规范