前端编码规范及最佳实践整理

一个 developer 个人能力再出众，如果写代码不按团队规范来，凭个人爱好想怎么写就这么写，他对这个团队也是负输出。

因为他的代码不可读、不可改，别人要花翻倍的时间来尝试去阅读，改起来甚至还会带着 Bug。如果这人哪天离职了，那对于他写过的项目来说简直是灾难。

如果你是他的 Leader，是不是很酸爽?

所以有些人在面试时，或者试用期莫名其妙就被淘汰了，可自己实现功能又没问题。面试官又不傻，那问题出在哪里?

他们可能在"编码规范意识"上吃了瘪，能实现功能，不一定能写好代码。

（一）写在前面

行业内流传的知名规范有很多，比如：

- Airbnb JavaScript 规范
- Airbnb React/JSX 规范
- CSS BEM 规范
- 网易 NEC CSS 规范
- 去哪儿网的《HTML/CSS开发规范指南》

如果要穷尽规范点，那可以写很多很多很多。

很少有技术团队把规范点做的非常多。实施了你就知道，如果点特别多，一来难以记忆，二来也难以控制检查成本，对成员来说还是块心理负担。

所以我推荐 Team Leader 将规范归纳得简练、精悍就好。

而且制定规范这件事，重点不在规范本身，而是在规范**制定前的团队内讨论**，以及规范出台后引导团队成员逐渐提高他们的**规范意识**。

所以**本规范只挑选了最必要的点，以及最有价值的点**。不求全，只求精。

（二）编码原则与约定

职责分离原则

数学家会数学，物理学家也会数学。但是遇到一个数学大难题，是给数学家还是物理学家解决?

在编码前端项目时，要注意**结构（HTML）、表现（CSS）和行为（JavaScript）之间的分离**。

另外遇到大型复杂场景，记得及时解耦，将一个大型模块拆分成若干小模块，并且在拆分过程中注意编码复用。

避免过长代码行

如果一行代码过长，在编辑器里就需要左右横向滚动来查看和编写，这样很不方便。

- 一般一行代码最多不超过 80 或 120 字符，看团队要求和个人习惯
- 建议在编辑器设置自动换行功能，超过一行最大字符数就换行显示
- 也可以在编辑器设置指示线，最大字符数那一列会显示一条指示线来警示

待办事项 TODO 约定

注释中用 `TODO` 标识待完成的任务，这样后面可以配合编辑器的搜索快速检索。

HTML 中：

```
<!-- 公司简介 -->
<div class="intro">
  ...
</div>

<!-- TODO: 关于我们 -->

<!-- 联系我们 -->
<div class="contact">
  ...
</div>
```

CSS 中：

```
/* TODO: 测试在非 Chrome 下的浏览器兼容性 */
...
```

JS 中：

```
/**
 * @description: 个人中心
 * @TODO: 接口联调
 */

// TODO: Mock 数据替换为真实接口
fetchMine().then(() => {
  ...
});
```

（三）HTML

文档类型

统一使用 HTML5 的标准文档类型：`<!DOCTYPE html>`。

HTML5 文档类型前后兼容，容易记也容易写，一般编辑器都支持快速生成 H5 文档，比如 VSCode 下保存文件为 `.html` 后缀然后输入 `!` 就可以快速生成。

用双引号包裹属性值

标签的属性值统一使用双引号 `""` 包裹。

不推荐：

```
<div class='intro'>...</div>
```

推荐：

```
<div class="intro">...</div>
```

标签语义化

- 根据 HTML 元素的用途去区分使用它们
- 满足场景时优先使用 HTML5 提供的语义标签：`<header>`、`<footer>`、`<article>`、`<section>`、`<nav>`、`<aside>`、`<address>`

不推荐：

```
<p>新闻标题 XXX</p>
```

推荐：

```
<h1>新闻标题 XXX</h1>
```

虽然使用 `<p>` 配合着 CSS 可以实现标题样式，但 `<p>` 的语义是段落，标题场景应使用 `h1~h6`。

将 `<script>` 标签放在 `<body>` 标签内的结尾处

不推荐：

```
<head>
  <title>Example HTML Page</title>
  <script src="example1.js"></script>
  <script src="example2.js"></script>
</head>
<body>
  ...
</body>
```

很早之前，所有的 <script> 标签都放在 <head> 标签里。这就意味着必须把所有的 JavaScript 代码都下载、解析和解释完成后，才能开始渲染页面（页面在浏览器解析到 <body> 的起始标签时开始渲染）。

对于需要很多 JavaScript 的页面，这会导致页面渲染的明显延迟，在此期间浏览器窗口完全空白。

为解决这个问题，现在通常将 <script> 标签放在 <body> 标签中页面内容的后面。

推荐：

```
<head>
  <title>Example HTML Page</title>
</head>
<body>
  ...
  ...
  <script src="example1.js"></script>
  <script src="example2.js"></script>
</body>
```

这样页面会在处理 JS 代码前渲染出来，由于浏览器显示空白页面的时间短了，用户会觉得页面加载更快了。

使用实体字符

一些字符在 HTML 中拥有特殊的含义，比如小于号 (<) 用于定义标签的开始。如果要正确显示，应优先使用实体字符。

这里列出常用的实体字符：

符号	用途	实体字符
	空格	、 、
©	版权	©
¥	人民币	¥
®	注册商标	®
>	大于号	>
<	小于号	<
&	和号	&

（四）CSS

不要过分依赖后代选择器

先写一个反例：

```
.header .header-title span a {
  ...
}
```

这种写法我给他的定义就是：“写时一时爽，维护火葬场。”

工作中已经见过太多这样写的同事了，简直是在代码里下毒。

从**维护性角度考虑，这样会将 HTML 元素的解构和 CSS 选择器后台关系绑定**。比如上面这个选择器，最终匹配的是个 `<a>` 标签，后续维护中，一旦这个 `<a>` 标签放置位置变了，比如改成 `.header-title` 的直接子元素（也就是和 `<span>` 同级了），那这段 CSS 就废了。

所以这种写法，你改了 HTML 结构，就要改 CSS，反之改了 CSS，HTML 结构可能也要调整，这就**不利于 CSS 选择器的复用**。

从**性能角度考虑，这样会增加 CSS 选择器匹配时间，从而导致渲染变慢**。可能很多前端同学不知道，**CSS 引擎解析后代选择器的顺序是从右向左的**。

意思就是，对于 `.header .header-title span a`，会先匹配所有的 `<a>`，然后匹配它们父级或爷爷中有 `<span>` 的，然后再匹配上级或上上级有 `.header-title` 的，以此类推。这样相比于从左往右解析，相当于先确定个小范围 `<a>`，然后再不断向上级匹配缩小这个范围，而不是先有个 `.header` 然后找它的子子孙孙们，如同大海捞针。

虽然从右往左解析，降低了 CSS 后代选择器匹配的回溯成本。但，为啥不能一步到位，直接用一个 `class` 选择器搞定呢?

当然可以，上面这种情况，你不如直接写一个 `.header-titleLink`，进行 CSS 匹配时就检索一次，完事。

所以**对于绝大部分写业务的场景，推荐选择器命名，仅使用 1 层 `class` 选择器，并通过命名区分它们**（当然还有 CSS Modules 技术可以进一步编译选择器命名，从根本上解决命名冲突问题）。

以下 CSS 选择器书写方式是 OK 的：

```
.news {
  ...
}

.header-titleLink {
  ...
}

.mr12 {
  margin-right: 12px;
}
```

`class` 选择器命名遵守 BEM 规范

知道了优先用 1 层 `class`，但是名字该怎么起才合适?

业内有一个非常知名且广泛使用的 CSS `class` 选择器命名规范——BEM

简单来说，它将需要命名的元素分为 3 部分：

- 块（Block）：独立的一整块区域；
- 元素（Element）：块中的元素，元素不能离开它所在的块而单独使用；
- 修饰（Modifier）：用来区分状态，比如是否已点击、是否 `hover` 等等。

举个例子：`header-link__activated`，其中块是 `header` 头部，头部中有元素 `link`，此时的状态是已激活的 `activated`。

当然，你的团队也不一定非要严格遵守 BEM 规范而使用下划线 `_`，你可以统一只用中划线 `-`，在修饰符前用两个 `-` 就可以了。简化成：

`header-link--activated`

这看起来也很爽。

元素和修饰不是必需的，比如你可以只写一个块：`.news`。所以大多数业务场景，块+块内元素已经可以搞定了。块一般就用 1 个单词，如果元素用 1 个单词不够，我的习惯是使用小驼峰，比如 `.header-titleLink`，这即便在配合 CSS Modules 场景下仍然很适用。

在列举几个使用修饰的例子：

```
/* 登录区域内的提交按钮，禁用状态 */
.login-submitBtn--disabled {
  ...
}

/* 文章区域，暗黑主题风格 */
.article--themeDark {
  ...
}

/* 文章区域内的图片，鼠标 hover 时 */
.article-img--hovered {
  ...
}
```

属性书写顺序

有的选择器样式要写好多，很容易就写成了一坨（反例）：

```
.xxx {
  border: 1px solid #e5e5e5;
  line-height: 1.5;
  width: 100px;
  height: 100px;
  color: #333;
  opacity: 0.5;
```

```
  text-align: center;
  position: absolute;
  display: block;
  float: right;
  top: 0;
  right: 0;
  background-color: #f5f5f5;
  border-radius: 3px;
  bottom: 0;
  left: 0;
}
```

这样如果后面要改，不用搜索的话可能要找个几秒才能找到要改的属性。

那不如**编码时给众多 CSS 属性分类**：

```
.xxx {
  /* 布局、位置 */
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  z-index: 100;

  /* 盒模型 */
  display: block;
  float: right;
  width: 100px;
  height: 100px;

  /* 排版 */
  line-height: 1.5;
  color: #333;
  text-align: center;

  /* 视觉效果相关 */
  background-color: #f5f5f5;
  border: 1px solid #e5e5e5;
  border-radius: 3px;

  /* 其他需要额外注意的 */
  opacity: 1;
}
```

编写顺序是按修改时可能对其他元素样式造成的影响大小来排的，影响可能更大的放前面，中间适当加入空行。

这样写，一来可以将重要的属性放在前面，读的时候一目了然，二来等到改的时候也可以按分类去快速找到对应属性。

（五）JavaScript（ES6+）

比较运算符 & 等号

- 优先使用 `===` 和 `!==` 而不是 `==` 和 `!=`
- 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 对应布尔值
  - 数字 如果是 `+0`、`-0` 或 `NaN` 则被计算为 `false`，否则为 `true`
  - 字符串 如果是空字符串 `''` 被计算为 `false`，否则为 `true`

命名规则

- 使用小驼峰命名对象、函数和实例。

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

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

- 使用大驼峰命名构造函数或类。

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

const bad = new user({
  name: 'nope'
});

// good
function User(options) {
  this.name = options.name;
}

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

- 不要使用下划线前/后缀

> 为什么？JavaScript 并没有私有属性或私有方法的概念。虽然使用下划线来表示「私有」是一种共识，但实际上这些属性是完全公开的，它本身就是你公共接口的一部分。这种习惯或许会导致开发者错误的认为改动它不会造成破坏或者不需要去测试。

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

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

函数命名

函数命名大多使用动词开头，例：

- 事件处理函数，可以用 `handle` 开头：`handleSubmitForm`、`handleFieldChange`、`handleOptionChange`；
- 获得整合处理的数据，可以用 `get` 开头，修改数据，可以用 `set` 开头：`getTeacherData`、`setTeacherData`；

变量和函数的命名尽量有区分度、语义化，可以适当使用单词简写。

类和构造函数

- 优先使用类而不是构造函数，因为类的语法更加简洁，避免直接操作原型

```
// 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;
  }
}
```

另外，对于 `class` 要根据实际场景，区分以下 3 种属性/方法：

- 实例属性、实例方法：由 `new` 构造函数得到的实例的属性和方法；
- 静态属性、静态方法：直接挂载构造函数上的属性和方法，比如 `Array.isArray`；
- 原型属性、原型方法：本质上是挂在 `构造函数.prototype` 上的属性和方法，它们被各个实例所共享。

箭头函数

- 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。）

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

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

（六）Vue 技术栈

组件命名

组件名为多个单词

组件名应该始终是多个单词的，根组件 `App` 以及 `<transition>`、`<component>` 之类的 Vue 内置组件除外。

这样做可以避免跟现有的以及未来的 HTML 元素相冲突，因为所有的 HTML 元素名称都是单个单词的。

```
// bad
Vue.component('todo', {
  // ...
})

export default { name: 'Todo' }

// good
Vue.component('todo-item', {
  // ...
})

export default { name: 'TodoItem' }
```

单文件组件的大小写

单文件组件的文件名应该要么始终是单词大写开头 (PascalCase)，要么始终是横线连接 (kebab-case)。

单词大写开头对于代码编辑器的自动补全最为友好，因为这使得我们在 JS(X) 和模板中引用组件的方式尽可能的一致。然而，混用文件命名方式有的时候会导致大小写不敏感的文件系统的问题，这也是横线连接命名同样完全可取的原因。

```
// bad
components/
|- mycomponent.vue

components/
|- myComponent.vue

// good
components/
|- MyComponent.vue

components/
|- my-component.vue
```

紧密耦合的组件名

和父组件紧密耦合的子组件应该以父组件名作为前缀命名。

如果一个组件只在某个父组件的场景下有意义，这层关系应该体现在其名字上。因为编辑器通常会按字母顺序组织文件，所以这样做可以把相关联的文件排在一起。

```
// bad
components/
|- TodoList.vue
|- TodoItem.vue
|- TodoButton.vue

components/
|- SearchSidebar.vue
|- NavigationForSearchSidebar.vue

// good
components/
|- TodoList.vue
|- TodoListItem.vue
|- TodoListItemButton.vue

components/
|- SearchSidebar.vue
|- SearchSidebarNavigation.vue
```

Prop 定义

Prop 定义应该尽量详细。

在你提交的代码中，`prop` 的定义应该尽量详细，至少需要指定其类型。

细致的prop 定义有两个好处：

- 它们写明了组件的 API，所以很容易看懂组件的用法；
- 在开发环境下，如果向一个组件提供格式不正确的 `prop`，Vue 将会警告，以帮助捕获潜在的错误来源。

```
// bad
// 这样做只有开发 Demo 时可以接受
props: ['status']

// good
props: {
  status: String
}

// 更好的做法!
props: {
  status: {
    type: String,
    required: true,
    validator: function(value) {
      return [
        'syncing',
        'synced',
        'version-conflict',
        'error'
      ].indexOf(value) !== -1
    }
  }
}
```

为 `v-for` 设置键值

请在组件上总是用 `key` 配合 `v-for`，以便维护内部组件及其子树的状态。

```
// bad
<ul>
  <li v-for="todo in todos">{{ todo.text }}</li>
</ul>

// good
<ul>
  <li v-for="todo in todos" :key="todo.id">{{ todo.text }}</li>
</ul>
```

不要直接使用节点索引作为 `key`，这会导致新旧节点序列的 Diff 算法部分优化失效。

模板中简单的表达式

组件模板应该只包含简单的表达式，复杂的表达式则应该重构为计算属性或方法。

复杂表达式会让你的模板变得不那么声明式。我们应该尽量描述应该出现的**是什么**，而非**如何**计算那个值。而且计算属性和方法使得代码可以重用。

```
<!-- bad -->
{{
  fullName.split(' ').map(function (word) {
    return word[0].toUpperCase() + word.slice(1)
  }).join(' ')
}}
```

```
<!-- good -->
<!-- 在模板中 -->
{{ normalizedFullName }}
```

```
// 复杂表达式已经移入一个计算属性
computed: {
  normalizedFullName: function () {
    return this.fullName.split(' ').map(function (word) {
      return word[0].toUpperCase() + word.slice(1)
    }).join(' ')
  }
}
```

（七）React 技术栈（含 JSX）

命名

- 扩展名: React 组件使用 `.jsx` 扩展名；
- 文件名: 组件文件名使用帕斯卡（大驼峰）命名. 如：`ReservationCard.jsx`；
- 引用命名: React 组件名使用帕斯卡命名，实例使用骆驼式命名：

```
// bad
import reservationCard from './ReservationCard';

// good
import ReservationCard from './ReservationCard';

// bad
const ReservationItem = <ReservationCard />;
```

```
// good
const reservationItem = <ReservationCard />;
```

JSX 缩进对齐

```
// bad
<Foo superLongParam="bar"
     anotherSuperLongParam="baz" />

// good, 有多行属性的话, 新建一行关闭标签
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// 若能在一行中显示, 直接写成一行
<Foo bar="bar" />

// 子元素按照常规方式缩进
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Quux />
</Foo>
```

正确设置 key

避免使用数组的索引 index 来作为属性 key 的值，推荐使用唯一 ID：

```
// bad
{todos.map((todo, index) => <Todo {...todo} key={index} />)}

// good
{todos.map(todo => (<Todo {...todo} key={todo.id} />))}
```

Refs

总是在 Refs 里使用回调函数，字符串形式已弃用：

```
// bad
<Foo ref="myRef" />

// good
<Foo ref={(ref) => { this.myRef = ref; }} />
```

为事件处理函数绑定 this

```
// bad
class extends React.Component {
  onClickDiv() {
    // do stuff
  }

  // 在每次 render 过程中， 再调用 bind 都会新建一个新的函数，浪费资源
  render() {
    return <div onClick={this.onClickDiv.bind(this)} />;
  }
}

// good
class extends React.Component {
  // 通过 class 的实例方法绑定 this
  onClickDiv = () => {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv} />;
  }
}
```