文件及目录规范

1. 文件夹规范

   文件夹格式均以小驼峰形式展现，如：myAccount

2. JS文件

   JS文件格式均以小驼峰形式展现，如myAccount.js

3. CSS文件规范

   CSS文件格式均以小驼峰形式展现

4. img文件

   img文件名格式均以中划线连接

5. 文本内容缩进

   配置统一的.editorconfig

   ```
   root = true

   [*]
   indent_style = space
   indent_size = 2
   end_of_line = lf
   charset = utf-8
   trim_trailing_whitespace = true
   insert_final_newline = true

   [*.md]
   indent_size = 4
   trim_trailing_whitespace = false
   ```

6. vue单文件行数限制

   文件内最大行数：**500**行。

   文件行数过多时，根据情况，可以：

   - 拆分为app.js、index.vue、index.less
   - 拆分出更细粒度的组件

HTML规范

1. `class` 必须单词全字母小写，单词间以 `-` 分隔。

2. 标签名必须使用小写字母。

   示例：

```
<!-- good -->
<p>Hello StyleGuide!</p>

<!-- bad -->
<P>Hello StyleGuide!</P>
```

3. 属性名必须使用小写字母。

   示例：

```
<!-- good -->
<table cellspacing="0">...</table>

<!-- bad -->
<table cellSpacing="0">...</table>
```

4. 属性值必须用双引号包围。

   不允许使用单引号，不允许不使用引号。

   示例：

```
<!-- good -->
<script src="esl.js"></script>

<!-- bad -->
<script src='esl.js'></script>
<script src=esl.js></script>
```

5. 保证 `favicon` 可访问。

   在未指定 favicon 时，大多数浏览器会请求 Web Server 根目录下的 favicon.ico 。为了保证favicon可访问，避免404，必须遵循以下两种方法之一：

   1. 在 Web Server 根目录放置 favicon.ico 文件。
   2. 使用 link 指定 favicon。

   示例：

```
<link rel="shortcut icon" href="path/to/favicon.ico">
```

6. 禁止 `img` 的 `src` 取值为空。延迟加载的图片也要增加默认的 `src`。

   src 取值为空，会导致部分浏览器重新加载一次当前页面，参考：
   https://developer.yahoo.com/performance/rules.html#emptysrc

CSS规范

1. 选择器 与 { 之间必须包含空格。

   示例：

```
.selector {
}
```

2. 属性名 与之后的 : 之间不允许包含空格， : 与 属性值 之间必须包含空格。

   示例：

```
margin: 0;
```

3. 当一个 rule 包含多个 selector 时，每个选择器声明必须独占一行。

   示例：

```
/* good */
.post,
.page,
.comment {
  line-height: 1.5;
}

/* bad */
.post, .page, .comment {
  line-height: 1.5;
}
```

4. >、+、~ 选择器的两边各保留一个空格。

   示例：

```
/* good */
main > nav {
  padding: 10px;
}

label + input {
  margin-left: 5px;
}

input:checked ~ button {
  background-color: #69C;
```

```
}

/* bad */
main>nav {
  padding: 10px;
}

label+input {
  margin-left: 5px;
}

input:checked~button {
  background-color: #69C;
}
```

5. 属性选择器中的值必须用双引号包围。

   > 不允许使用单引号，不允许不使用引号。

   示例：

```
/* good */
article[character="juliet"] {
  family: "Vivien Leigh", victoria, female;
}

/* bad */
article[character='juliet'] {
  family: "Vivien Leigh", victoria, female;
}
```

6. 属性定义必须另起一行。

   示例：

```
/* good */
.selector {
  margin: 0;
  padding: 0;
}

/* bad */
.selector { margin: 0; padding: 0; }
```

7. 属性定义后必须以分号结尾。

   示例：

```
/* good */
.selector {
  margin: 0;
}

/* bad */
.selector {
  margin: 0
}
```

8. 尽可能在浏览器能高效实现的属性上添加过渡和动画。

   在可能的情况下应选择这样四种变换：

   - `transform: translate(npx, npx);`
   - `transform: scale(n);`
   - `transform: rotate(ndeg);`
   - `opacity: 0..1;`

   典型的，可以使用 translate 来代替 left 作为动画属性。

   示例：

```
/* good */
.box {
  transition: transform 1s;
}
.box:hover {
  transform: translate(20px); /* move right for 20px */
}

/* bad */
.box {
  left: 0;
  transition: left 1s;
}
.box:hover {
  left: 20px; /* move right for 20px */
}
```

JAVASCRIPT规范

1. 变量定义使用`let`和`const`，禁止使用`var`声明变量。如果声明的变量之后不涉及变量修改，一律使用`const`进行变量声明。

```
// bad
var a = 1
// bad 后面不涉及修改
```

```
let noModified = 1
console.log(noModified)

// good
let isDev = false
isDev = process.env.NODE_ENV === 'development'

// good 后面不涉及该变量的修改，使用const声明
const dir = '/home/user'
console.log(dir)
```

2. 对象、函数、实例使用用小驼峰式命名

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

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

3. 常量采用大写字母，下划线连接的方式

```
const MAX_COUNT = 10
```

4. 用大驼峰式命名类

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

5. 使用字面值创建对象和数组

```
// bad
const item = new Object()
const items = new Array()

// good
const item = {}
const items = []
```

6. 字符串使用单引号 ''

```
// bad
const name = "this is a string"

// bad - 样例应该包含插入文字或换行
const name = `this is a string`

// good
const name = 'this is a string'
```

7. 用字符串模板而不是字符串拼接来组织可编程字符串。

> 模板字符串更具可读性、语法简洁、字符串插入参数。

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

8. 禁止对字符串使用eval

9. 不要使用`arguments`，用rest语法`...`代替。

> 明确所用的参数。而且rest参数是真数组，而不是类似数组的`arguments`。

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

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

10. 不要在非函数块（if、while等等）内声明函数。把这个函数分配给一个变量。

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

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

11. 不要用`new Function`创建函数。

> 以这种方式创建函数将类似于字符串eval()，容易产生漏洞。

```
// bad
var add = new Function('a', 'b', 'return a + b')

// still bad
var subtract = Function('a', 'b', 'return a - b')
```

12. 一个路径只 import 一次。

```
// bad
import foo from 'foo'
```

```
// … some other imports … //
import { named1, named2 } from 'foo'

// good
import foo, { named1, named2 } from 'foo'

// good
import foo, {
  named1,
  named2,
} from 'foo'
```

13. import 放在其他所有语句之前。

```
// bad
import foo from 'foo'
foo.init()

import bar from 'bar'

// good
import foo from 'foo'
import bar from 'bar'

foo.init()
```

14. 禁止直接链式获取对象属性值。

> 容易导致异常发生，在node/ssr服务上出现该异常，极容易导致内存溢出

```
const obj = {a: { b : 1 }}

// bad
console.log(obj.a.b)

// good
console.log(obj && obj.a && obj.a.b)

// best
_.get(obj, 'a.b')
```

15. 比较运算时，用 === 和 !== 而不是 == 和 !=。

16. 比较运算时，布尔值用缩写，而字符串和数字要明确比较对象

```
// bad
if (isValid === true) {
```

```
  // ...
}

// good
if (isValid) {
  // ...
}

// bad
if (name) {
  // ...
}

// good
if (name !== '') {
  // ...
}

// bad
if (collection.length) {
  // ...
}

// good
if (collection.length > 0) {
  // ...
}
```

17. 在case和default分句里用大括号创建一块包含语法声明的区域(例如：let, const, function, and class)。

> 语法声明在整个switch的代码块里都可见，但是只有当其被分配后才会初始化，他的初始化是当这个case被执行时才产生。 当多个case分句试图定义同一个变量时就出问题了。

```
// 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 {}
  }
}
```

18. 三元表达式不应该嵌套，通常是单行表达式。

```
// bad
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null

// better
const maybeNull = value1 > value2 ? 'baz' : null

const foo = maybe1 > maybe2
  ? 'bar'
  : maybeNull

// best
const maybeNull = value1 > value2 ? 'baz' : null

const foo = maybe1 > maybe2 ? 'bar' : maybeNull
```

19. 如果 `if` 语句中总是需要用 `return` 返回， 那后续的 `else` 就不需要写了。 `if` 块中包含 `return`， 它后面的 `else if` 块中也包含了 `return`， 这个时候就可以把 `return` 分到多个 `if` 语句块中。

```
// 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
```

```
  }
}
```

20. 当函数内有多个`return`时，返回值类型应保持一致。

```
// bad
function foo() {
  if (x) {
    return 1
  }

  if (y) {
    return false
  }

  return 'b'
}

// good
function foo() {
  if (x) {
    return 1
  }

  if (y) {
    return 2
  }

  return 3
}
```

注释规范

1. 业务代码中在逻辑上永远不会执行到的代码，应及时删除
2. 已注释掉的代码要及时删除
3. 临时变量必须要注明其业务逻辑用途

```
// bad
let template = ''
list.forEach(item => {
  template += `<div class="${item.type === 1 ? 'red' :
''}">${item.text}</div>`
})

// good
// 要最终展示的质检模型文案
let template = ''
// 从质检模型集合中拼接文案样式
list.forEach(item => {
```

```
  // item.type 0:不涉及违规，正常显示；1:涉及违规，标红；
  // item.text 质检模型文案
  template += `<div class="${item.type === 1 ? 'red' :
''}">${item.text}</div>`
})
```

4. 注释不能文不对题，与代码实现逻辑出现偏差，代码有改动必须更新对应注释

```
// bad
showDialog () {
  // 先重置数据
  this.add()
}

// good
showDialog () {
  // 先重置数据
  this.reset()
}
```

5. 每个api接口的函数定义和通用方法定义必须注释业务逻辑用途

```
// bad
export const countTransferTaskByCondition = body => {
  return
api.JPost(`/api/sdb/crm/mission/agent/task/countTransferTaskByConditio
n`, body)
}

export const reallocationByCondition = body => {
  return
api.JPost(`/api/sdb/crm/mission/agent/task/reallocationByCondition`,
body)
}

// good
// 代理商中转站-分配弹窗-查询
export const countTransferTaskByCondition = body => {
  return
api.JPost(`/api/sdb/crm/mission/agent/task/countTransferTaskByConditio
n`, body)
}

// 代理商中转站-重新分配弹窗-确认
export const reallocationByCondition = body => {
  return
api.JPost(`/api/sdb/crm/mission/agent/task/reallocationByCondition`,
body)
}
```

```
// bad
async getDataByQuery ({
  isLoading,
  apiName,
  dataName,
  emitName,
  hasRemind,
  queryObj = {},
  resDataName,
  defaultData = [],
  successRemind,
  callBackNameList
}) {
  ...
}

// good
// 查询带请求参数的公用方法
/*
  @params options:
  {
    isLoading: 是否有loading
    apiName: 接口名称
    dataName: 数据名称
    emitName: emit名称
    hasRemind:是否有失败提示
    queryObj: 请求的对象数据
    resDataName: 返回值的对象属性
    defaultData: 默认值
    successRemind: 是否有成功提示
    callBackNameList: this上的回调函数名称数组
  }
*/
async getDataByQuery ({
  isLoading,
  apiName,
  dataName,
  emitName,
  hasRemind,
  queryObj = {},
  resDataName,
  defaultData = [],
  successRemind,
  callBackNameList
}) {
  ...
}
```

6. 每个vue文件的props和data都应注明业务逻辑含义及用途

```
// bad
props: {
  mobile: {
    type: String,
    default: ''
  },
  taskId: {
    type: String,
    default: ''
  },
  lpAnswerInfo: {
    type: Object,
    default: () => ({
      answerCalling: false
    })
  },
  isSave: {
    type: Boolean,
    default: false
  },
  isPredictive: {
    type: Boolean,
    default: false
  }
}

// good
// 外呼手机号
mobile: {
  type: String,
  default: ''
},
// 外呼任务id
taskId: {
  type: String,
  default: ''
},
// 坐席振铃待接听相关信息
lpAnswerInfo: {
  type: Object,
  default: () => ({
    // 是否要接听
    answerCalling: false
  })
},
// 通话记录是否已经保存
isSave: {
  type: Boolean,
  default: false
},
// 是否是预测式外呼
isPredictive: {
  type: Boolean,
```

```
    default: false
  }
```