微信小程序业务开发规范与最佳实践

> 💡 本文用于记录基于公司技术选型与业务特点的开发规范、最佳实践沉淀过程

文中出现：只能、不能、必须、应该、不应、不得、禁止、推荐 、建议，关键词时应给予关注

一、工程规范

1.1 项目配置

常见的 uni-app 工程结构：

Bash

```
├──.husky                 # git hook 配置文件，确保提交的文件 必须 通过 ESLint 检测
│   └──pre-commit         # 预提交执行命令：如 yarn lint
├──.vscode                # VSCode 配置
│   ├──settings.json      # 团队 必须 统一配置
│   └──extensions.json    # 团队 必须 安装推荐插件
├──public                 # vue 打包入口文件
├──src                    # 业务核心目录
├──.editorconfig          # 代码风格配置
├──.eslintrc.js           # ESLint 配置文件
├──.prettierrc.js         # 代码风格配置
└──babel.config.js        # babel 配置文件，与 ES 转换有关
└──postcss.config.js      # postcss 配置文件，与 css 转换有关
└──babel.config.js        # babel 配置文件
└──vue.config.js          # vue 打包配置文件
└──package.json           # 项目信息
```

❗ **规范：**

1. **必须** 安装 VSCode 编辑器用于编码，并且安装工程目录推荐插件 (如：ESLint)
2. **不得** 关闭 ESLint 检测，如遇 ESLint 失效应及时修复
3. **不得** 跳过 Husky 中 ESLint 检测提交代码
4. **不得** 随意修改项目任何配置文件，如 eslintrc.js、vue.config.js
5. **建议** 在各核心业务模块的 `README.md` 文件中添加描述信息，如：文档地址、关联模块等

1.2 业务目录

业务核心目录 (src) 结构：

Bash

```
├──components            # 符合 vue 组件规范的 uni-app 组件目录
│   └──uni-icons         # uni-app 对应组件
│   └──ym-swiper         # 公司对应组件
├──pages                 # 主包业务页面，放 tab 页内容
│   ├──app
│   └──order
├──sub-pages             # 分包内容，以 sub 开头
│   ├──app
│   └──order
├──static                # 存放应用引用的本地静态资源（如图片、视频等）的目录，注意：静态资源只能存放于此
│   └──images            # 图片
│   └──styles            # 样式
├──store                 # 全局数据
│   └──modules
├──mixins                # 复用逻辑集合
│   └──timer-mixin.js
├──utils                 # 辅助集合
│   └──common
│   └──constants
│   └──enums
│   └──request
├──configs                # 配置集合
│   └──config.default.js  # 默认
│   └──config.dev.js      # 开发
│   └──config.test.js     # 测试
│   └──config.pre.js      # 预发
│   └──config.prod.js     # 正式
│   └──env.js             # 基础环境
├──main.js               # Vue初始化入口文件
├──App.vue               # 应用配置，用来配置App全局样式以及监听 应用生命周期
├──manifest.json         # 配置应用名称、appid、logo、版本等打包信息，详见
└──pages.json            # 配置页面路由、导航条、选项卡等页面类信息，详见
```

❗ **规范：**

1. 主包文件 **只能** 放在 `pages` 目录下
2. 分包目录 **必须** 以 sub 开头，如：`sub-pages`，且只能存放与该分包相关文件
3. 全局数据 **必须** 按模块放在 `store/modules` 目录下
4. `components` 目录下 **只能** 放公共展示 UI 组件
5. 业务页面中的组件，放在各自模块的 `./components` 目录下
6. 非三方组件 **必须** 加前缀区分，如 `ym-swiper`
7. 公共函数、变量、枚举等 **只能** 放在 `utils` 对应目录下面，如：`constants`
8. 静态资源 **只能** 存放在 `static` 目录下
9. 公共业务逻辑 **只能** 存放在 `mixins`，并且文件名以 mixin 结尾
10. 类的文件名 **必须** 以大写开头，如：`Time.js`
11. 文件命名 **推荐** 使用 **`kebab-case`**，如：`timer-mixin.js`

二、代码规范

项目使用 uni-app 框架，更多关注的是基于 **vue 语法 (组件)** 以及差异化的微信小程序规范。项目通过 ESLint + Prettier 来保证团队协作时遵循代码规范。

2.1 命名风格

- 单文件组件的文件名 **必须** 全部小写且遵循 **`kebab-case`**，如：`button-base.vue`

- 组件名 **推荐** 为多个单词 (这样做可以避免跟现有的以及未来的 HTML 元素相冲突)

JavaScript

```javascript
// 反例
// todo.vue
export default {
    name: 'Todo'
}

// 正例
// todo-item.vue
export default {
    name: 'TodoItem'
}

```

- 组件的命名需遵从以下原则：
  - **有意义的**: 不过于具体，也不过于抽象
  - **简短**: 2 到 3 个单词
  - **具有可读性**: 以便于沟通交流

HTML

```html
<!-- 反例 -->
<btn-group></btn-group> <!-- 虽然简短但是可读性差，使用 `button-group` 替代 -->
<ui-slider></ui-slider> <!-- ui 前缀太过于宽泛，在这里意义不明确 -->
<slider></slider> <!-- 与自定义元素规范不兼容 -->

<!-- 正例 -->
<app-header></app-header>
<user-list></user-list>
<range-slider></range-slider>

```

- 自定义私有 property 使用 `$_` 前缀，并附带一个命名空间以回避和其它作者的冲突，如：`$_yourPluginName_`

JavaScript

```javascript
// 反例
const timerMixin =  {
    methods: {
        setTimeout(){
            //....
        }
    }
}

// 正例
const timerMixin =  {
    methods: {
        $setTimeout(){
            //....
        }
    }
}
```

- 在声明 prop 的时，命名 **必须** 使用 **camelCase**，而在模板和 JSX 中应该始终使用 **kebab-case**

JavaScript

```javascript
// 反例
props: {
  'greeting-text': String
}
// <WelcomeMessage greetingText="hi"/>


// 正例
props: {
  greetingText: String
}
// <WelcomeMessage greeting-text="hi"/>
```

- 应用特定样式和约定的基础组件 (也就是展示类的、无逻辑的或无状态的组件) 应该全部以一个特定的前缀开头，比如 `Base`、`App` 或 `V`

Bash

```
# 反例
components/
|- my-button.vue
|- vue-table.vue
|- icon.vue

# 好例子
components/
|- base-button.vue
|- base-table.vue
|- base-icon.vue
```

- 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

Bash

```
# 反例
components/
|- todo-list.vue
|- todo-item.vue
|- todo-button.vue

# 好例子
components/
|- todo-list.vue
|- todo-list-item.vue
|- todo-list-item-button.vue
```

组件名 **必须** 以高级别的 (通常是一般化描述的) 单词开头，以描述性的修饰词结尾

Bash

```
# 反例
components/
|- clear-search-button.vue
|- run-search-button.vue

# 正例
components/
|- search-button-clear.vue
|- search-button-run.vue
```

- 模板中的组件名 **必须** 遵循 **kebab-case**

JavaScript

```
// 反例
<MyComponent></MyComponent>

// 正例
<my-component></my-component>
```

- 组件名应该倾向于完整单词而不是缩写

Bash

```
# 反例
components/
|- sd-settings.vue

# 正例
components/
|- student-dashboard-settings.vue
```

- 方法名、参数名、成员变量、局部变量 **必须** 都统一使用 lowerCamelCase 风格，如：`localValue / getHttpMessage() / inputUserId`

- 所有编程相关的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式

- 类名使用 UpperCamelCase 风格，如：`Timer、TimerStore`

- 常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长，如：`MAX_TIMER_LIMIT`

2.2 代码格式

代码风格通过 prettier 进行约束，配置如下：

JavaScript

```
module.exports = {
  printWidth: 100, // 单行代码超出 100 个字符自动换行
  tabWidth: 2, //（默认值）一个 tab 键缩进相当于 2 个空格
  useTabs: true, // 行缩进使用 tab 键代替空格
  semi: false, //（默认值）语句的末尾加上分号
  singleQuote: true, // 使用单引号
  quoteProps: 'as-needed', //（默认值）仅仅当必须的时候才会加上双引号
  jsxSingleQuote: true, // 在 JSX 中使用单引号
  trailingComma: 'all', // 不用在多行的逗号分隔的句法结构的最后一行的末尾加上逗号
  bracketSpacing: true, //（默认值）在括号和对象的文字之间加上一个空格
  arrowParens: 'avoid', // 当箭头函数中只有一个参数的时候可以忽略括弧
  htmlWhitespaceSensitivity: 'ignore', // vue template 中的结束标签结尾尖括号掉到了下一行
  vueIndentScriptAndStyle: false, //（默认值）对于 .vue 文件，不缩进 <script> 和 <style> 里的内容
  embeddedLanguageFormatting: 'auto', //（默认值）允许自动格式化内嵌的代码块
}
```

2.3 注释

- 类、类属性、类方法的注释必须使用 Jsdoc 规范，使用/**内容*/格式，不得使用 // xxx 方式。(便于编辑器智能提示)，如：

Kotlin

```

export default class Timer {
  /**
   * 构造函数
   * @param {Boolean} isInterval 是否是 setInterval
   * @param {Function} fn 回调函数
   * @param {Number} timeout 定时器执行时间间隔
   * @param  {...any} arg 定时器其他参数
   */
  constructor(isInterval = false, fn = () => {}, timeout = 0, ...arg) {}
  /**
   * 启动定时器
   * @param {Object} timerStore 定时器管理器
   */
  start(timerStore) {}

  /* 暂停定时器 */
  suspend() {}
}
```

- 方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使 用/* */注释，注意与代码对齐

JavaScript

```
export default {
    methods:{
        fetchUserInfo(){
          // 单行注释
          let lock = false
          /**
          * 多行注释
          */
          const { query,isLogin } = this
        }
    }
}
```

- 所有的枚举类型字段必须要有注释，说明每个数据项的用途

JavaScript

```
export const TAKE_WAY_MODE = {
  SERVE: 1, // 送餐到桌
  SELF: 2, // 到店自取
};
```

- 谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。如果无用，则删除

- 代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑 等的修改

- 对于注释的要求：第一、能够准确反映设计思想和代码逻辑；第二、能够描述业务含 义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同 天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路；注释也是给继任者看 的，使其能够快速接替自己的工作

- 好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一 个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释又是相当大的负担

- 特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫描， 经常清理此类标记。线上故障有时候就是来源于这些标记处的代码。如：`待办事宜（TODO）：（标记人，标记时间，[预计处理时间]）`、`错误，不能工作（FIXME）：（标记人，标记时间，[预计处理时间]）`

2.4 业务编码

- **必须** 基于模块开发，每一个 Vue 组件（等同于模块）首先必须专注于解决一个单一的问题，*独立的*、*可复用的*、*微小的* 和 *可测试的*

- **必须** 只在需要时创建组件
  - 如果组件太大, 可能很难重用和维护
  - 如果组件粒度太小，项目就会（因为深层次的嵌套而）被淹没，也更难使组件间通信以及潜在性能问题
  - 避免那些 “以后可能会有用” 的组件污染项目

- 尽量保证每一个文件的代码行数不要超过 **100** 行。也请保证组件可独立的运行。比较好的做法是增加一个单独的 **demo 示例**

- 组件 **必须** 按照一定的结构组织，使得组件便于理解

HTML

```
<template>
  <view>
    <!-- ... -->
  </view>
</template>

<script>
  export default {
    // 不要忘记了 name 属性
    name: 'RangeSlider',
    // 使用组件 mixins 共享通用功能
    mixins: [],
    // 组成新的组件
    extends: {},
    // 组件属性、变量
    props: {
      bar: {}, // 按字母顺序
      foo: {},
      fooBar: {},
    },
    // 变量
    data() {},
    computed: {},
    // 使用其它组件
    components: {},
    // 方法
    watch: {},
    methods: {},
    // 生命周期函数
    created() {},
    mounted() {},
  };
</script>

<style>
  .Ranger__Wrapper { /* ... */ }
</style>
```

- watch **不能** 使用箭头函数

JavaScript

```
// 反例
export default {
    watch:{
        isLogin:()=>{
            // 此时 this 不是 vue 实例
            console.log(this)
        }
    }
}

// 正例
export default {
    watch:{
        isLogin(){}
    }
}
```

- computed 中 **不能** 使用异步函数，且 **不能** 出现重复的属性名

JavaScript

```javascript
// 反例
export default {
    computed:{
        pro: async function(){
            return await someFun()
        }
        // 不能重复
        pro(){
            return Promise.all([new Promise(resolve,reject)=>{}])
        }
    }
}

// 正例
export default {
    computed:{
        nickname(){
            return this.$store.userInfo.nickname
        }
    }
}
```

- 组件的 `data` **必须**是一个函数

JavaScript

```javascript
// 反例
export default {
    data: {
        foo: 'bar'
    }
}

// 正例
export default {
    data(){
        return {
            foo: 'bar'
        }
    }
}
```

- 组件 props 设计 **必须** 遵循原子化，且 **不得** 冗余 (保证 API **清晰直观**)

HTML

```
<!-- 反例 -->
<range-slider :config="complexConfigObject"></range-slider>

<!-- 正例 -->
<range-slider
  :values="[10, 20]"
  :min="0"
  :max="100"
  :step="5"
  @on-slide="updateInputs"
  @on-end="updateResults">
</range-slider>
```

- Prop 定义应该尽量详细 (便于使用者了解组件的用法，在开发环境避免错误传值)

JavaScript

```
// 反例
props: ['status']

// 正例
props: {
    status: {
        type: String,
        require: true,
        validator: function (value) {
            return [
                'syncing',
                'error'
            ].indexOf(value) !== -1
        }
    }
}

```

- 事件命名应该以动词（如 client-api-load） 或是 名词（如 drive-upload-success）结尾。（出处）

- 为 `v-for` 设置键值 (vue 组件会复用，未设置 key 值会导致 diff 出错)

HTMLBars

```
<!-- 反例 -->
<view v-for="todo in todos">
    {{ todo.text }}
</view>

<!-- 正例 -->
<view v-for="todo in todos" :key="todo.id">
    {{ todo.text }}
</view>

```

- 永远不要把 `v-if` 和 `v-for` 同时用在同一个元素上 (可以使用过滤后的数据用于 v-for 渲染)

HTML

```
<!-- 反例 -->
<view v-for="todo in todos" v-if="todo.done" :key="todo.id" >
    {{ todo.text }}
</view>

<!-- 正例 -->
<view v-for="todo in todosDone" :key="todo.id">
    {{ todo.text }}
</view>

```

- 在单文件组件、字符串模板和 JSX 中没有内容的组件应该是自闭合的

HTML

```
<!-- 反例 -->
<!-- 在单文件组件、字符串模板和 JSX 中 -->
<MyComponent></MyComponent>

<!-- 正例 -->
<!-- 在单文件组件、字符串模板和 JSX 中 -->
<MyComponent/>
```

- 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

HTML

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

<!-- 正例 -->
<view>
    {{ normalizedFullName }}
</view>
```

JavaScript

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

- 指令缩写 (用 `:` 表示 `v-bind:`、用 `@` 表示 `v-on:` 和用 `#` 表示 `v-slot:`) 应该要么都用要么都不用

HTML

```
<!-- 反例 -->
<input v-bind:value="newTodoText" :placeholder="newTodoInstructions">

<!-- 正例 -->
<input :value="newTodoText" :placeholder="newTodoInstructions" >
```

- 如果一组 `v-if` + `v-else` 的元素类型相同，最好使用 `key` (比如两个 `<view>` 元素，避免因 key 值引发 意料之外的结果

HTML

```html
<!-- 反例 -->
<view v-if="error">
    {{ error }}
</view>
<view v-else>
    {{ results }}
</view>

<!-- 正例 -->
<view v-if="error" key="search-status">
    {{ error }}
</view>
<view v-else key="search-results">
    {{ results }}
</view>
```

- **应该** 优先通过 prop 和事件进行父子组件之间的通信，而不是 `this.$parent` 或变更 prop

JavaScript

```javascript
// 注意组件定义的差别，这里仅为展示，用 Vue.component 的形式，实际 uni-app 中建议使用单文件组件

// 正例
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }
  },
  template: '<input v-model="todo.text">'
})

// 反例
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }
  },
  template: `
    <input
      :value="todo.text"
      @input="$emit('input', $event.target.value)"
    >
  `
})
```

- **谨慎使用** `this.$refs` 组件必须是保持独立的，如果一个组件的 API 不能够提供所需的功能，那么这个组件在设计、实现上是有问题的。(**注意：小程序里是没有 dom 的**)

HTML

```html
<!-- 使用 this.$refs 的适用情况-->
<modal ref="basicModal">
  <h4>Basic Modal</h4>
  <button class="primary" @click="$refs.basicModal.hide()">Close</button>
</modal>
<button @click="$refs.basicModal.open()">Open modal</button>

<!-- Modal component -->
<template>
  <div v-show="active">
    <!-- ... -->
  </div>
</template>

<script>
  export default {
    // ...
    data() {
      return {
        active: false,
      };
    },
    methods: {
      open() {
        this.active = true;
      },
      hide() {
        this.active = false;
      },
    },
    // ...
  };
</script>
```

- 按如下规范使用数据
  a. data 数据 **只能** 挂与渲染相关的数据，其他数据直接挂在 this 上
  b. computed 中的数据 **只能** 挂载与渲染相关的数据
  c. 使用 mapState 获取数据时，**不能** 全量获取 **只能** 按需获取
  d. watch 数据时，**只能** 监听与渲染相关的数据

HTML

```html
<template>
  <view class="wrap">
    <view> {{ title }} </view>
    <view> {{ nickname }} </view>
  </view>
</template>

<script>
import { mapState } from 'vuex';

export default {
  data() {
    return {
        title: '趣小面' // 只能挂载与渲染有关的视图
    };
  },
  created() {
   this.timer = null; // timer 为临时变量，且与渲染无关。无需挂在 data 上
  },
  mounted() {
    this.timer = setTimeout(() => {
      this.title = '趣小面 +';
    }, 1000);
  },
  computed: {
    ...mapState({
      nickname: (state) => state.userInfo.nickname, // 只能按需获取
    }),
  },
  watch: {
    title(){
        console.log('标题已变更') // 只能 watch 与渲染有关的数据
    }
  }
  // ... 清除定时器相关逻辑
};
</script>

<style>
.wrap {
  padding: 24px;
}
</style>
```

2.5 业务日志

- 业务中不可直接使用系统 Log API，建议创建自定义 Logger 类，规范日志使用

JavaScript

```
// 反例
console.log('123')

// 正例
// src/utils/logger.js
export default {
    log(..arg){ console.log(...arg)}
    //...
}

// 使用
import logger from '@/utils/logger.js'
logger.log('123')
```

- 所有日志 **必须** 有意义，避免无效日志混乱显示

JavaScript

```
// 反例
console.log(userInfo)

// 正例 ( 可通过 logger 单例对象简化操作 )
console.log('调试_当前用户信息为', userInfo)

```

- 临时调试的日志 **必须** 及时移除

- 业务中的扩展日志（如打点、临时监控等）命名方式： `logType_logName.log` 。logType：日志类型，如 `stats/monitor/access` 等；logName：日志描 述。(这种命名的好处：通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找)

- **必须** 通过打包配置，将生产环境的日志输出移除

2.6 样式规范

- 类名 **必须** 使用小写字母，遵循 BEM 规范 如：`form__submit--disabled`

- id **必须** 采用驼峰命名方式，建议尽量少使用 id 选择器

- scss 中的变量、函数、混合、placeholder **必须** 采用驼峰式命名

- id 和 class 的名称总是使用可以反应元素目的和用途的名称，或其他通用的名称，代替表象和晦涩难懂的名称

CSS

```css
/*! 反例 */
.fw-800 {
    font-weight: 800;
}
.red {
    color: red;
}

/*! 正例 */
.heavy {
  font-weight: 800;
}
.important {
  color: red;
}
```

- **禁止** 使用标签选择器

- **禁止** 嵌套层级过多

SCSS

```scss
/*! 反例 */
.main{
  .title{
    .name{
      color:#fff
    }
  }
}

/*! 正例 */
.main-title{
   .name{
      color:#fff
   }
}
```

三、静态资源规范

- 静态资源 **必须** 放在 `static` 目录下，且只能放图片、字体等无需编译的资源

Bash

```bash
┌──static
│   └──iconfont      # iconfont 资源
│   └──images        # 图片资源
```

- `css`、`less/scss` 等资源不要放在 `static` 目录下，建议放在自建的 `common` 目录下

- 本地背景图片的引用路径推荐使用以 ~@ 开头的绝对路径

CSS

```css
 .test2 { background-image: url('~@/static/logo.png'); }
```

- **禁止** 使用太大的背景图或字体文件 (大于等于 40kb)，如必须使用，需将其转换为 base64 格式使用，或将其上传至 OBS 为网络地址使用

- 不使用的静态资源 **必须** 及时删除

- 静态资源网络路径 **必须** 加协议头 `https`

- 图片 **必须** 按设计侧指定的大小使用

CSS

```css
/*
若设计稿宽度为 750px，元素 A 在设计稿上的宽度为 100px，
那么元素 A 在 uni-app 里面的宽度应该设为：750 * 100 / 750，结果为：100rpx
公式：750 * 元素在设计稿中的宽度 / 设计稿基准宽度
参考：rpx 单位转换
*/

.left-a{
    width:100 rpx;
    height: 100 rpx;
}
```

- 图片资源显示的 高 / 宽 与原图的 高 / 宽 **不得** 超过 15%

- 图片 **必须** 经过 压缩工具 压缩才能使用，且单张图片不得高于 150 kb

- 图标按钮最小可点击区域 **不得** 低于 为 20*20 px

关于公司内部图片组件的使用方式推荐阅读：

- 图片压缩整体规划方案
- ym-network-image

四、分包规范

所谓的 **主包**，即放置默认启动页面、TabBar 页面，以及一些所有分包都需用到公共资源、JS 脚本；而**分包** 则是根据开发者的配置进行划分。**独立分包** 是小程序中一种特殊类型的分包，可以独立于主包和其他分包运行

推荐阅读：小程序分包加载

> ❗ **规范：**
>
> 1. 整个小程序所有分包大小 **不能** 超过 20M
> 2. 单个分包 / 独立包 / 主包 大小 **不能** 超过 2M
> 3. `subpackage` 的根目录 **不能** 是另外一个 `subpackage` 内的子目录
> 4. `tabBar` 页面 **必须** 在 app（主包）内
> 5. 分包 / 独立包 / 主包 之间的资源 **不能** 互相依赖 (js 文件、wxss、自定义组件、插件等)

五、最佳实践

最佳实践产生背景，推荐阅读：

- 微信小程序 setData 最佳实践
- 微信小程序定时器回收

> ❗ **规范：**
>
> 1. setData **不得** 传送与渲染无关的数据
> 2. 定时器不使用时，**必须** 回收
> 3. 页面非可视状态 (onShow)，**不得** 有任何操作，如：setData、定时器、其他事件监听等
> 4. 父子组件通信，**必须** 遵守单向数据流
> 5. **不得** 滥用全局数据，全局数据 **必须** 符合前端数据流管理

六、其他工具

圈复杂度检测

圈复杂度 (Cyclomatic complexity) 是一种代码复杂度的衡量标准，也称为条件复杂度或循环复杂度，它可以用来衡量一个模块判定结构的复杂程度，数量上表现为独立现行路径条数，也可理解为覆盖所有的可能情况最少使用的测试用例数。简称 CC 。其符号为 VG 或是 M 。

计算规则：

圈复杂度实际上就是等于判定节点（`if-else`、`switch case`、`for` 循环、三元运算符、`||` 和 `&&` 等）的数量再加上 1，示例代码：

JavaScript

```
function testComplexity(*param*) {
    let result = 1;
    if (param > 0) {
        result--;
    }
    for (let i = 0; i < 10; i++) {
        result += Math.random();
    }
    switch (parseInt(result)) {
        case 1:
            result += 20;
            break;
        case 2:
            result += 30;
            break;
        default:
            result += 10;
            break;
    }
    return result > 20 ? result : result;
}
```

如上，代码圈复杂度为：`1+1+2+1+1 = 6`

衡量标准：

Eslint 规则 可用于检验，下面拟定衡量代码质量标准如下：

圈复杂度	代码状况	可测性	维护成本
1-10	清晰	高	低
10-20	复杂	中	中

20-30	非常复杂	低	高
>30	不可读	不可测	非常高

成功 检测完成,耗费9794ms，共检测【287】个文件，【2062】个函数，其中可能存在问题的函数【16】个

复杂度	重构建议	函数名	函数类型	位置
11	建议重构	agreeRender	Method	/src/components/ym-author/ym-author.vue [375,16]
12	建议重构	fetchData		/src/components/ym-popup-make/ym-popup-make.vue [44,20]
20	强烈建议重构	someCallback	Method	/src/components/ym-store-list/ym-store-list.vue [213,17]
13	建议重构	setReport		/src/components/ym-store-list/ym-store-list.vue [290,14]
11	建议重构	needShowScan	Method	/src/pages/order/components/create-info/create-info.vue [118,17]
13	建议重构	isNeedShowShopNotFound		/src/pages/order/components/create-info/create-info.vue [160,27]
39	强烈建议重构	changeDineWay	Method	/src/pages/order/create.vue [738,18]
15	建议重构	changeCartNum		/src/store/modules/cart.js [179,16]
12	建议重构	initCart		/src/store/modules/cart.js [264,17]
34	强烈建议重构	*		/src/store/modules/login.js [266,16]
30	强烈建议重构	initShopInfo	Method	/src/store/modules/order.js [82,15]
11	建议重构	assemble		/src/sub-pages/invoice/detail.vue [79,13]
12	建议重构	submit		/src/sub-pages/order/comment.vue [148,17]
18	强烈建议重构	choose	Method	/src/sub-pages/order/components/package-good/package-good.vue [96,11]
14	建议重构	*		/src/utils/banner/banner.js [88,27]
17	强烈建议重构	request		/src/utils/request/core/request.js [56,16]

七、相关资料

- Vue 官方风格指南
- Angular 风格
- eslint-config-airbnb
- Vue.js 组件编码规范
- ES6 编程风格
- ESLint
- Prettier