05. Code Review规范整理

1.组件规范

组件拆分基本原则： 每一个 Vue 组件（等同于模块）应该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*。

如果你的组件做了太多的事或是变得臃肿，请将其拆分成更小的组件并保持单一的原则。一般来说，尽量保证每一个文件的css以外代码行数不要超过 200 行。也请保证组件可独立的运行。

1)[建议]Vue 组件命名

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

同时还需要注意：

- 必须符合自定义元素规范: 使用连字符分隔单词，切勿使用保留字。
- `/-` 前缀作为命名空间: 如果非常通用的话可使用一个单词来命名，这样可以方便于其它项目里复用。

为什么?

- 组件是通过组件名来调用的。所以组件名必须简短、富有含义并且具有可读性。
- 如果只写一个单词，不便于识别是自定义组件，且可能和保留字 冲突

如何做?

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

<!--  -->
<btn-group></btn-group> <!-- .  `button-group`  -->
<ui-slider></ui-slider> <!-- ui  -->
<slider></slider> <!--  -->
```

2)[建议]组件表达式简单化

Vue.js 的表达式是 100% 的 Javascript 表达式。这使得其功能性很强大，但也带来潜在的复杂性。因此，你应该尽量保持表达式的简单化。

为什么?

- 复杂的行内表达式难以阅读。
- 行内表达式是不能够通用的，这可能会导致重复编码的问题。
- IDE 基本上不能识别行内表达式语法，所以使用行内表达式 IDE 不能提供自动补全和语法校验功能。

怎么做?

如果你发现写了太多复杂并难以阅读的行内表达式，那么可以使用 method 或是 computed 属性来替代其功能。

```
<!--  -->
<template>
  <h1>
    {{ `${year}-${month}` }}
  </h1>
</template>
<script type="text/javascript">
  export default {
    computed: {
      month() {
        return this.twoDigits((new Date()).getUTCMonth() + 1);
      },
      year() {
        return (new Date()).getUTCFullYear();
      }
    },
    methods: {
      twoDigits(num) {
        return ('0' + num).slice(-2);
      }
    },
  };
</script>

<!--  -->
<template>
  <h1>
    {{ `${(new Date()).getUTCFullYear()}-${('0' + ((new Date()).getUTCMonth()+1)).slice(-2)}` }}
  </h1>
</template>
```

3)[建议]组件 props 原子化

虽然 Vue.js 支持传递复杂的 JavaScript 对象通过 props 属性，但是你应该尽可能的使用原始类型的数据。尽量只使用 JavaScript 原始类型（字符串、数字、布尔值）和函数。尽量避免复杂的对象。

为什么?

- 使得组件 API 清晰直观。
- 只使用原始类型和函数作为 props 使得组件的 API 更接近于 HTML(5) 原生元素。
- 其它开发者更好的理解每一个 prop 的含义、作用。
- 传递过于复杂的对象使得我们不能够清楚的知道哪些属性或方法被自定义组件使用，这使得代码难以重构和维护。
- 便于再组件中直接预验证，增强组件的稳定性

怎么做?

- 尽量组件的每一个属性单独使用一个 props，并且使用函数或是原始类型的值。
- 如果为了简单声明了对象，可以使用 v-bind="propsObj" 的方式结构展开传递给子组件。

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

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

4)[强制]验证组件的 props

在 Vue.js 中，组件的 props 即 API，一个稳定并可预测的 API 会使得你的组件更容易被其他开发者使用。

组件 props 通过自定义标签的属性来传递。属性的值可以是 Vue.js 字符串(`:attr="value"` 或 `v-bind:attr="value"`)或是不传。你需要保证组件的 props 能应对不同的情况。

为什么?

- 验证组件 props 可以保证你的组件永远是可用的（防御性编程）。即使其他开发者并未按照你预想的方法使用时也不会出错。

怎么做?

- 提供默认值。
- 使用 type 属性校验类型。
- 使用 props 之前先检查该 prop 是否存在。

```
<template>
  <input type="range" v-model="value" :max="max" :min="min">
</template>
<script type="text/javascript">
  export default {
    props: {
      max: {
        type: Number, //
        default() { return 10; }, //  require
      },
      min: {
        type: Number,
        default() { return 0; },
      },
      value: {
        type: Number,
        default() { return 4; },
      },
    },
  };
</script>
```

5)[强制]组件事件命名

Vue.js 提供的处理函数和表达式都是绑定在 ViewModel 上的，组件的每一个事件都应该按照一个好的命名规范来，这样可以避免不少的开发问题，具体可见如下 为什么。

为什么?

- 开发者可以随意给事件命名，即使是原生事件的名字，这样会带来迷惑性。
- 过于宽松的事件命名可能与 DOM 模板不兼容。

怎么做?

- 事件名也使用连字符命名。
- 一个事件的名字对应组件外的一组意义操作，如：upload-success、upload-error 以及 dropzone-upload-success、dropzone-upload-error（如果需要前缀的话）。

- 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）

6)[建议]避免 this.$parent

Vue.js 支持组件嵌套，并且子组件可访问父组件的上下文。访问组件之外的上下文违反了基于模块开发的第一原则。因此你应该尽量避免使用 `this.$parent`。

为什么?

- 组件必须相互保持独立，Vue 组件也是。如果组件需要访问其父层的上下文就违反了该原则。
- 如果一个组件需要访问其父组件的上下文，那么该组件将不能在其它上下文中复用。

怎么做?

- 通过 props 将值传递给子组件。
- 通过 props 传递回调函数给子组件来达到调用父组件方法的目的。
- 通过在子组件触发事件来通知父组件。

7)[建议]避免在子组件中直接修改父组件的数据

如果传入的是基本数据类型，vue 本身是不允许修改的，但是传入的是对象的情况，修改就不会报错了，但是还是为了数据流向的明确，还是不应该修改。

为什么?

- 所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定，这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解【vue 官方】。

怎么做?

- 通过 props 将值传递给子组件，修改的时候，通过事件传递到父组件中修改。
- 如果子组件需要内部维护或修改，1、定义一个本地的 data 属性并将这个 prop 用作其初始值。2、使用这个 prop 的值来定义一个计算属性。
- 如果嵌套太深，可以谨慎考虑将数据及修改数据的方法 provide给子组件，在子组件中调用方法来修改上级组件的值，而避免多层 emit 事件。

```
// 1 data  prop
props: {
  initialCounter: {
    type: Number,
    default() { return 0; },
  }
},
data: function () {
  return {
    counter: this.initialCounter
  }
}
// 2 prop
props: {
  size: {
    type: String,
    default() { return ''; },
  }
},
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

8)[建议]父组件操作子组件的方式

有时候需要修改子组件的数据，禁止通过$ref.comName 直接修改，也避免使用 watch props 变量的变更来实现内部的某个操作。

为什么?

- 组件必须相互保持独立，Vue 组件也是。如果组件可以直接修改子组件的数据就违反了该原则。
- 通过 watch props 的方式触发某个操作，整体流程太繁琐，不便于维护，以此作为 api 对外不能方便使用。

怎么做?

- 在子组件定义单独对外的方法（不能直接暴露内部的方法），来实现单一功能明确的操作，来实现内部数据的操作，一保证内部功能的拓展下和对外 api 的稳定性。

```
//
/**
 *
 * @param {string|number} key  id
 * @param {boolean} checked
 */
setChecked(key, checked) {
  this.checkoutChange(this.nodesMap[key], checked, false);
},
```

8)[建议]避免定义无用数据

组件中，避免定义无用数据，包括：data字段、props字段、方法。

业务组件不必要定义 name 属性，如果要定义，需要和文件名表达意义一致

2. 缩进

[建议] 在语句的行长度超过 120 时，根据逻辑条件合理缩进。

```
//
//  )  {  if
if (user.isAuthenticated()
    && user.isInRole('admin')
    && user.hasAuthority('add-admin')
    || user.hasAuthority('delete-admin')
) {
        // Code
}

//  )
//
foo(
    aVeryVeryLongArgument,
    anotherVeryLongArgument,
    callback
);

//
// baidu.format""""
baidu.format(
    dateFormatTemplate, //template
    year, month, date, hour, minute, second //data
)

// 3
const result = thisIsAVeryVeryLongCondition
    ? resultA : resultB;
const result = condition
    ? thisIsAVeryVeryLongResult
    : resultB;

//  {  }
const array = [{
        // ...
    },
    {
        // ...
    }];
```

3.命名

[强制] `//` 。

[强制] `//` 。

示例:

```
// variable
let loadingModules = {};


// function...
function stringFormat(source) {
}


// params...
function hear(theBells) {
}
```

[强制]公用 [:] 使用 的命名方式。

示例:

```
const HTML_ENTITY = {};
```

[强制] 使用驼峰式命名。

示例:

```
equipments.heavyWeapons = {};
```

[强制] 由多个单词组成的缩写词，在命名中，根据当前命名法和出现的位置，所有字母的大小写与首字母的大小写保持一致。

示例:

```
function XMLParser() {
}

function insertHTML(element, html) {
}

let httpRequest = new HTTPRequest();
```

[建议] 使用 。

示例:

```
function getStyle(element) {
}
```

[建议] boolean 类型的变量使用 is 或 has 开头。

示例：

```
let isReady = false;
let hasMoreCommands = false;
```

[建议] Promise 用 表达。

示例：

```
const loadingData = ajax.get('url');
loadingData.then(callback);
```

[建议] 使用 Pascal， 使用 的命名方式。

示例：

```
const TargetState = {
    READING: 1,
    READED: 2,
    APPLIED: 3,
    READY: 4
};
```

4.注释

单行注释

[强制] 必须独占一行。

// 后跟一个空格，缩进与下一行被注释说明的代码一致。

多行注释

[建议] 避免使用 /*...*/ 这样的多行注释。有多行注释内容时，使用多个单行注释。

文档化注释

[建议] 为了便于代码阅读和自文档化，以下内容必须包含以 /...*/ 形式的块注释中。**

[建议] 自文档化的文档说明 what，而不是 how。

类型定义

[强制] 类型定义都是以{开始, 以}结束。

解释：

常用类型如：{string}, {number}, {boolean}, {Object}, {Function}, {RegExp}, {Array}, {Date}。

类型不仅局限于内置的类型，也可以是自定义的类型。比如定义了一个类 Developer，就可以使用它来定义一个参数和返回值的类型。

[强制] 对于基本类型 {string}, {number}, {boolean}，首字母必须小写。

类型定义	语法示例	解释
String	{string}	--

Number	{number}	--
Boolean	{boolean}	--
Object	{Object}	--
Function	{Function}	--
RegExp	{RegExp}	--
Array	{Array}	--
Date	{Date}	--
单一类型集合	{Array.<string>}	string 类型的数组
多类型	{(number | boolean)}	可能是 number 类型, 也可能是 boolean 类型
允许为null	{?number}	可能是 number, 也可能是 null
不允许为null	{!Object}	Object 类型, 但不是 null
Function类型	{function(number, boolean)}	函数, 形参类型
Function带返回值	{function(number, boolean):string}	函数, 形参, 返回值类型
参数可选	@param {string=} name	可选参数, =为类型后缀
可变参数	@param {...number} args	变长参数, ...为类型前缀
任意类型	{*}	任意类型
可选任意类型	@param {*=} name	可选参数，类型不限
可变任意类型	@param {...*} args	变长参数，类型不限

文件注释

[强制] 文件顶部必须包含文件注释，用 `@file` 标识文件说明。

示例:

```
/**
 * @file Describe the file
 */
```

[建议] 文件注释中可以用 `@author` 标识开发者信息, 可添加多个author(责任划分)

可安装vscode-fileheader, ctrl + option + i (Mac)添加头部文档注释

```
/*
 * @Author: author of the file
 * @Date: 2019-04-01 17:08:07
 * @Last Modified by:
 * @Last Modified time: 2019-04-01 17:08:07
 */
```

示例:

```
/**
 * @file Describe the file
 * @Author author-name
 * author-name2
 */
```

data/props/计算属性注释

[强制]data/props/计算属性 应在变量『上一行』添加单行注释

示例:

```
data() {
    return {
      //
      showStart: 0,
      //
      showEnd: 0,
      //
      itemHeight: 34
    };
  }
```

函数/方法注释

[强制] 函数/方法注释必须包含函数说明，有参数和返回值时必须使用注释标识。

[强制] 参数和返回值注释必须包含类型信息和说明。

示例:

```
/**
 *
 *
 * @param {string} p1 1
 * @param {string} p2 2
 * @return {Object}
 */
function foo(p1, p2) {
    return {
        p1: p1,
        p2: p2
    };
}
```

推荐使用vscode的代码片段，具体设置如下中的note片段

```
{
  // Place your global snippets here. Each snippet is defined under a snippet name and has a scope, prefix,
body and
  // description. Add comma separated ids of the languages where the snippet is applicable in the scope field.
If scope
  // is left empty or omitted, the snippet gets applied to all languages. The prefix is what is
  // used to trigger the snippet and the body will be expanded and inserted. Possible variables are:
  // $1, $2 for tab stops, $0 for the final cursor position, and ${1:label}, ${2:another} for placeholders.
  // Placeholders with the same ids are connected.
  // Example:
  "console": {
    "scope": "javascript,typescript,Vue",
    "prefix": "con",
    "body": [
      "console.log('$1');",
    ],
    "description": "console.log"
  },
  "notes": {
    "scope": "javascript,typescript,Vue",
    "prefix": "note",
    "body": [
      "/**",
      " * @description $1",
      " * @param {any}",
      " * @returns",
      " */"
    ],
    "description": ""
  },
  "$message":{
    "scope": "javascript,typescript,Vue",
    "prefix": "me",
    "body": [
      "if (res.code===0) {",
        "",
      "} else {",
      "  this.$message.error(res.msg)",
      "}"
    ],
    "description": "api"
  }
}
```

[强制] 对 Object 中各项的描述， 必须使用 `@param` 标识。

示例:

```
/**
 *
 *
 * @param {Object} option
 * @param {string} option.url option
 * @param {string=} option.method option
 */
function foo(option) {
    // TODO
}
```

事件注释

[强制] 必须使用 `@event` 标识事件，事件参数的标识与方法描述的参数标识相同。

示例:

```
/**
 *
 *
 * @event
 * @param {Object} e e
 * @param {string} e.before before
 * @param {string} e.after after
 */
onchange: function (e) {
}
```

常量注释

[建议] 常量必须使用 @const 标记，并包含说明和类型信息。

示例:

```
/**
 *
 *
 * @const
 * @type {string}
 */
const REQUEST_URL = 'myurl.do';
```

细节注释

对于内部实现、不容易理解的逻辑说明、摘要信息等，我们可能需要编写细节注释。

[建议] 细节注释遵循单行注释的格式。说明必须换行时，每行是一个单行注释的起始。

示例:

```
function foo(p1, p2, opt_p3) {
    //
    //
    for (...) {
        ....
    }
}
```

[强制] 有时我们会使用一些特殊标记进行说明。特殊标记必须使用单行注释的形式。下面列举了一些常用标记:

解释:

1. TODO: 有功能待实现。此时需要对将要实现的功能进行简单说明。
2. FIXME: 该处代码运行没问题，但可能由于时间赶或者其他原因，需要修正。此时需要对如何修正进行简单说明。
3. HACK: 为修正某些问题而写的不太好或者使用了某些诡异手段的代码。此时需要对思路或诡异手段进行描述。
4. XXX: 该处存在陷阱。此时需要对陷阱进行描述。

5.国际化

[强制] merge-request 之前实现国际化文本提取，以减少后续国际化的实现时间。

[强制] 二选一 1、中文作 key， 2、 key使用大写英文 并用下划线分割，页面中对应位置应该有 "i18n:" 标识的对应中文注释

```
<!--  -->
<p class="title">{{ $t('') }}</p>



<!--  -->
<!-- i18n:  -->
<p class="title">{{ $t('COMMON.LOGIN_TITLE') }}</p>
```

国际化其他注意事项，参见 洞鉴5.1国际化

参考:

https://github.com/fex-team/styleguide/blob/master/javascript.md

https://github.com/pablohpsilva/vuejs-component-style-guide/blob/master/README-CN.md#%E6%8F%90%E4%BE%9B%E7%BB%84%E4%BB%B6-demo