vue代码风格摘要

一：有关组件命名

1.1：组件名应该始终由多个单词的组成

注：根组件 App 以及 `<transition>`、`<component>` 之类的 Vue 内置组件除外。

原因：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。

坏例子

```
Vue.component('name', {
  // ...
})
```

好例子

```
Vue.component('name-item', {
  // ...
})
```

1.2：组件命名法

组件名应采用单词大写开头（PascalCase），或采用横线连接命名（kebab-case）。

原因：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可取的原因。

PascalCase 相比 kebab-case 有一些优势：

编辑器可以在模板里自动补全组件名，因为 PascalCase 同样适用于 JavaScript。
`<MyComponent>` 视觉上比 `<my-component>` 更能够和单个单词的 HTML 元素区别开来，因为前者的不同之处有两个大写字母，后者只有一个横线。

不幸的是，由于 HTML 是大小写不敏感的，在 DOM 模板中必须仍使用 kebab-case。

还请注意，如果你已经是 kebab-case 的重度用户，那么与 HTML 保持一致的命名约定且在多个项目中保持相同的大小写规则就可能比上述优势更为重要了。在这些情况下，在所有的地方都使用 kebab-

case 同样是可以接受的。

坏例子

```
components/
|- mycomponent.vue

components/
|- myComponent.vue
```

好例子

```
<MyComponent/>
```

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

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

1.3：基础组件命名

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。

坏例子

```
components/
|- MyButton.vue
|- VueTable.vue
|- Icon.vue
```

好例子

```
components/
|- BaseButton.vue
|- AppTable.vue
|- VIcon.vue
```

1.4：组件命名的单词顺序

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

坏例子

```
components/
|- ClearSearchButton.vue
```

好例子

```
components/
|- SearchButtonClear.vue
```

1.5：自闭合组件

在单文件组件中，对于没有内容的组件应该是自闭合的。

自闭合组件表示它们不仅没有内容，而且刻意没有内容。其不同之处就好像书上的一页白纸对比贴有“本页有意留白”标签的白纸。而且没有了额外的闭合标签，你的代码也更简洁。

不幸的是，HTML 并不支持自闭合的自定义元素——只有官方的“空”元素。所以上述策略仅适用于进入 DOM 之前 Vue 的模板编译器能够触达的地方，然后再产出符合 DOM 规范的 HTML。

坏例子

```
<MyComponent></MyComponent>
```

好例子

```
<MyComponent/>
```

1.6：组件名单词尽量不要缩写

组件名应该倾向于完整单词而不是缩写。

编辑器中的自动补全已经让书写长命名的代价非常之低了，而其带来的明确性却是非常宝贵的。不常用的缩写尤其应该避免。

宗旨：不同开发成员看到组件名能尽可能快的理解意思

坏例子

```
components/
|- SdSettings.vue
|- UProfOpts.vue
```

好例子

```
components/
|- StudentDashboardSettings.vue
|- UserProfileOptions.vue
```

二：Prop 定义

2.1：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。

坏例子

```
// 这样做只有开发原型系统时可以接受
props: ['status']
```

好例子

```
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
    }
  }
}
```

2.2：Prop 名大小写

在声明 prop 的时候，其命名应该始终使用 camelCase，而在模板和 JSX 中应该始终使用 kebab-case。

我们单纯的遵循每个语言的约定。在 JavaScript 中更自然的是 camelCase。而在 HTML 中则是 kebab-case。

坏例子

```
props: {
  'greeting-text': String
}

<WelcomeMessage greetingText="hi"/>
```

好例子

```
props: {
  greetingText: String
}

<WelcomeMessage greeting-text="hi"/>
```

三：为 v-for 设置键值

总是用 **key** 配合 **v-for**

在组件上总是必须用 key 配合 v-for，以便维护内部组件及其子树的状态。甚至在元素上维护可预测的行为，比如动画中的对象固化 (object constancy)，也是一种好的做法。

坏例子

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

好例子

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

四：避免 **v-if** 和 **v-for** 用在一起

永远不要把 **v-if** 和 **v-for** 同时用在同一个元素上

一般我们在两种常见的情况下会倾向于这样做：

为了过滤一个列表中的项目 (比如 v-for="user in users" v-if="user.isActive")。在这种情形下，请将 users 替换为一个计算属性 (比如 activeUsers)，让其返回过滤后的列表。

为了避免渲染本应该被隐藏的列表 (比如 v-for="user in users" v-if="shouldShowUsers")。这种情形下，请将 v-if 移动至容器元素上 (比如 ul、ol)。

坏例子

```
<ul>
  <li
    v-for="user in users"
    v-if="user.isActive"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>

<ul>
  <li
    v-for="user in users"
    v-if="shouldShowUsers"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

好例子

```
<ul>
  <li
    v-for="user in activeUsers"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>

<ul v-if="shouldShowUsers">
  <li
    v-for="user in users"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

五：为组件样式设置作用域

对于应用来说，顶级 App 组件和布局组件中的样式可以是全局的，但是其它所有组件都应该是有作用域的。

这条规则只和单文件组件有关。你不一定要使用 scoped attribute。设置作用域也可以通过 CSS Modules，那是一个基于 class 的类似 BEM 的策略，当然你也可以使用其它的库或约定。

不管怎样，对于组件库，我们应该更倾向于选用基于 class 的策略而不是 scoped attribute。

这让覆写内部样式更容易：使用了常人可理解的 class 名称且没有太高的选择器优先级，而且不太会导致冲突。

坏例子

```
<template>
  <button class="btn btn-close">X</button>
</template>

<style>
.btn-close {
  background-color: red;
}
</style>
```

好例子

```
<template>
  <button :class="[$style.button, $style.buttonClose]">X</button>
</template>



<style module>
.button {
  border: none;
  border-radius: 2px;
}

.buttonClose {
  background-color: red;
}
</style>

<template>
  <button class="c-Button c-Button--close">X</button>
</template>



<style>
.c-Button {
  border: none;
  border-radius: 2px;
}

.c-Button--close {
  background-color: red;
}
</style>
```

六：多个 **attribute** 的元素

多个 **attribute** 的元素应该分多行撰写，每个 **attribute** 一行

原因：在 JavaScript 中，用多行分隔对象的多个 property 是很常见的最佳实践，因为这样更易读。

坏例子

```
<img src="https://vuejs.org/images/logo.png" alt="Vue Logo">

<MyComponent foo="a" bar="b" baz="c"/>
```

好例子

```
<img
  src="https://vuejs.org/images/logo.png"
  alt="Vue Logo"
>

<MyComponent
  foo="a"
  bar="b"
  baz="c"
/>
```

七：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。

坏例子

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

好例子

```
{{ 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(' ')
  }
}
```

八：指令缩写

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，保持使用的一致性。

九：要谨慎使用 scoped 中的元素选择器

在 scoped 样式中，类选择器比元素选择器更好，因为大量使用元素选择器是很慢的。