这里是官方的 Vue 特有代码的风格指南。如果在工程中使用 Vue，为了回避错误、小纠结和反模式，该指南是份不错的参考。不过我们也不确信风格指南的所有内容对于所有的团队或工程都是理想的。所以根据过去的经验、周围的技术栈、个人价值观做出有意义的偏差是可取的。

对于其绝大部分，我们也总体上避免就 JavaScript 或 HTML 的本身提出建议。我们不介意你是否使用分号或结尾的逗号。我们不介意你在 HTML 特性中使用单引号还是双引号。不过当我们发现在 Vue 的情景下有帮助的特定模式时，也会存在例外。

> **不久之后，我们还会提供操作层面的技巧。有的时候你只需要遵守规则，而我们会尽可能向你展示如何使用 ESLint 及其它自动化程序把操作层面弄得更简单。**

最终，我们把所有的规则归为了四个大类：

规则归类

优先级 A：必要的

这些规则会帮你规避错误，所以学习并接受它们带来的全部代价吧。这里面可能存在例外，但应该非常少，且只有你同时精通 JavaScript 和 Vue 才可以这样做。

优先级 B：强烈推荐

这些规则能够在绝大多数工程中改善可读性和开发体验。即使你违反了，代码还是能照常运行，但例外应该尽可能少且有合理的理由。

优先级 C：推荐

当存在多个同样好的选项，选任意一个都可以确保一致性。在这些规则里，我们描述了每个选项并建议一个默认的选择。也就是说只要保持一致且理由充分，你可以随意在你的代码库中做出不同的选择。请务必给出一个好的理由！通过接受社区的标准，你将会：

1. 训练你的大脑，以便更容易的处理你在社区遇到的代码；
2. 不做修改就可以直接复制粘贴社区的代码示例；
3. 能够经常招聘到和你编码习惯相同的新人，至少跟 Vue 相关的东西是这样的。

优先级 D：谨慎使用

有些 Vue 特性的存在是为了照顾极端情况或帮助老代码的平稳迁移。当被过度使用时，这些特性会让你的代码难于维护甚至变成 bug 的来源。这些规则是为了给有潜在风险的特性敲个警钟，并说明它们什么时候不应该使用以及为什么。

优先级 A 的规则：必要的 (规避错误)

组件名为多个单词 (必要)

组件名应该始终是多个单词的，根组件 `App` 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。

反例

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

```
export default {
  name: 'Todo',
  // ...
}
```

好例子

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

```
export default {
  name: 'TodoItem',
  // ...
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
}
```

组件数据 (必要)

组件的 `data` 必须是一个函数。

当在组件中使用 `data` 属性的时候 (除了 `new Vue` 外的任何地方)，它的值必须是返回一个对象的函数。

详解

当 `data` 的值是一个对象时，它会在这个组件的所有实例之间共享。想象一下，假如一个 `TodoList` 组件的数据是这样的：

```
data: {
  listTitle: '',
  todos: []
}
```

我们可能希望重用这个组件，允许用户维护多个列表 (比如分为购物、心愿单、日常事务等)。这时就会产生问题。因为每个组件的实例都引用了相同的数据对象，更改其中一个列表的标题就会改变其它每一个列表的标题。增删改一个待办事项的时候也是如此。

取而代之的是，我们希望每个组件实例都管理其自己的数据。为了做到这一点，每个实例必须生成一个独立的数据对象。在 JavaScript 中，在一个函数中返回这个对象就可以了：

```
data: function () {
  return {
    listTitle: '',
    todos: []
  }
}
```

反例

```
Vue.component('some-comp', {
  data: {
    foo: 'bar'
  }
})
```

```
export default {
  data: {
    foo: 'bar'
  }
}
```

好例子

```
Vue.component('some-comp', {
  data: function () {
    return {
      foo: 'bar'
    }
  }
})
```

```
// In a .vue file
export default {
  data () {
    return {
      foo: 'bar'
    }
  }
}
```

```
// 在一个 Vue 的根实例上直接使用对象是可以的，
// 因为只存在一个这样的实例。
new Vue({
  data: {
    foo: 'bar'
  }
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
})
```

Prop 定义 (必要)

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。

详解

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 将会告警，以帮助你捕获潜在的错误来源。

反例

```
// 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
    }
  }
}
```

为 `v-for` 设置键值 (必要)

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。

详解

假设你有一个待办事项列表：

```
data: function () {
  return {
    todos: [
      {
        id: 1,
        text: '学习使用 v-for'
      },
      {
        id: 2,
        text: '学习使用 key'
      }
    ]
  }
}
```

然后你把它们按照字母顺序排序。在更新 DOM 的时候，Vue 将会优化渲染把可能的 DOM 变动降到最低。即可能删掉第一个待办事项元素，然后把它重新加回

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

到列表的最末尾。

这里的问题在于，不要删除仍然会留在 DOM 中的元素。比如你想使用 `<transition-group>` 给列表加过渡动画，或想在被渲染元素是 `<input>` 时保持聚焦。在这些情况下，为每一个项目添加一个唯一的键值 (比如 `:key="todo.id"`) 将会让 Vue 知道如何使行为更容易预测。

根据我们的经验，最好*始终*添加一个唯一的键值，以便你和你的团队永远不必担心这些极端情况。也在少数对性能有严格要求的情况下，为了避免对象固化，你可以刻意做一些非常规的处理。

反例

```
<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>
```

避免 `v-if` 和 `v-for` 用在一起 **(必要)**

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`)。

详解

当 Vue 处理指令时，`v-for` 比 `v-if` 具有更高的优先级，所以这个模板：

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

将会经过如下运算：

```
this.users.map(function (user) {
  if (user.isActive) {
    return user.name
  }
})
```

因此哪怕我们只渲染出一小部分用户的元素，也得在每次重渲染的时候遍历整个列表，不论活跃用户是否发生了变化。

通过将其更换为在如下的一个计算属性上遍历：

```
computed: {
  activeUsers: function () {
    return this.users.filter(function (user) {
      return user.isActive
    })
  }
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
}
```

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

我们将会获得如下好处：

- 过滤后的列表*只*会在 `users` 数组发生相关变化时才被重新运算，过滤更高效。
- 使用 `v-for="user in activeUsers"` 之后，我们在渲染的时候*只*遍历活跃用户，渲染更高效。
- 解藕渲染层的逻辑，可维护性 (对逻辑的更改和扩展) 更强。

为了获得同样的好处，我们也可以把：

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

更新为：

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

通过将 `v-if` 移动到容器元素，我们不会再对列表中的*每个*用户检查 `shouldShowUsers`。取而代之的是，我们只检查它一次，且不会在 `shouldShowUsers` 为否的时候运算 `v-for`。

反例

```
<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"
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
  >
    {{ user.name }}
  </li>
</ul>
```

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

为组件样式设置作用域 (必要)

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` 特性。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` 特性。

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。

详解

如果你和其他开发者一起开发一个大型工程，或有时引入三方 HTML/CSS (比如来自 Auth0)，设置一致的作用域会确保你的样式只会运用在它们想要作用的组件上。

不止要使用 `scoped` 特性，使用唯一的 class 名可以帮你确保那些三方库的 CSS 不会运用在你自己的 HTML 上。比如许多工程都使用了 `button`、`btn` 或 `icon` class 名，所以即便你不使用类似 BEM 的策略，添加一个 app 专属或组件专属的前缀 (比如 `ButtonClose-icon`) 也可以提供很多保护。

反例

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

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

好例子

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

<!-- 使用 `scoped` 特性 -->
<style scoped>
.button {
  border: none;
  border-radius: 2px;
}

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

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

<!-- 使用 CSS Modules -->
<style module>
.button {
  border: none;
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
  border-radius: 2px;
}

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

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

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

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

私有属性名 (必要)

在插件、混入等扩展中始终为自定义的私有属性使用 `$_` 前缀。并附带一个命名空间以回避和其它作者的冲突 (比如 `$_yourPluginName_`)。

详解

Vue 使用 `_` 前缀来定义其自身的私有属性，所以使用相同的前缀 (比如 `_update`) 有覆写实例属性的风险。即便你检查确认 Vue 当前版本没有用到这个属性名，也不能保证和将来的版本没有冲突。

对于 `$` 前缀来说，其在 Vue 生态系统中的目的是暴露给用户的一个特殊的实例属性，所以把它用于*私有*属性并不合适。

不过，我们推荐把这两个前缀结合为 `$_`，作为一个用户定义的私有属性的约定，以确保不会和 Vue 自身相冲突。

反例

```
var myGreatMixin = {
  // ...
  methods: {
    update: function () {
      // ...
    }
  }
}
```

```
var myGreatMixin = {
  // ...
  methods: {
    _update: function () {
      // ...
    }
  }
}
```

```
var myGreatMixin = {
  // ...
  methods: {
    $update: function () {
      // ...
    }
  }
}
```

```
var myGreatMixin = {
  // ...
  methods: {
    $_update: function () {
      // ...
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
    }
  }
}
```

好例子

```
var myGreatMixin = {
  // ...
  methods: {
    $_myGreatMixin_update: function () {
      // ...
    }
  }
}
```

优先级 B 的规则：强烈推荐 (增强可读性)

组件文件 强烈推荐

只要有能够拼接文件的构建系统，就把每个组件单独分成文件。

当你需要编辑一个组件或查阅一个组件的用法时，可以更快速的找到它。

反例

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

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

好例子

```
components/
|- TodoList.js
|- TodoItem.js
```

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

单文件组件文件的大小写 强烈推荐

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。

反例

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

```
components/
|- myComponent.vue
```

好例子

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
components/
|- MyComponent.vue
```

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

基础组件名 强烈推荐

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`。

详解

这些组件为你的应用奠定了一致的基础样式和行为。它们可能**只**包括：

- HTML 元素
- 其它基础组件
- 第三方 UI 组件库

但是它们**绝不会**包括全局状态 (比如来自 Vuex store)。

它们的名字通常包含所包裹元素的名字 (比如 `BaseButton`、`BaseTable`)，除非没有现成的对应功能的元素 (比如 `BaseIcon`)。如果你为特定的上下文构建类似的组件，那它们几乎总会消费这些组件 (比如 `BaseButton` 可能会用在 `ButtonSubmit` 上)。

这样做的几个好处：

- 当你在编辑器中以字母顺序排序时，你的应用的基础组件会全部列在一起，这样更容易识别。
- 因为组件名应该始终是多个单词，所以这样做可以避免你在包裹简单组件时随意选择前缀 (比如 `MyButton`、`VueButton`)。
- 因为这些组件会被频繁使用，所以你可能想把它们放到全局而不是在各处分别导入它们。使用相同的前缀可以让 webpack 这样工作：

```
var requireComponent = require.context("./src", true, /^Base[A-Z]/)
requireComponent.keys().forEach(function (fileName) {
  var baseComponentConfig = requireComponent(fileName)
  baseComponentConfig = baseComponentConfig.default || baseComponentConfig
  var baseComponentName = baseComponentConfig.name || (
    fileName
      .replace(/^.+\//, '')
      .replace(/\.\w+$/, '')
  )
  Vue.component(baseComponentName, baseComponentConfig)
})
```

反例

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

好例子

```
components/
|- BaseButton.vue
|- BaseTable.vue
|- BaseIcon.vue
```

```
components/
|- AppButton.vue
|- AppTable.vue
|- AppIcon.vue
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
components/
|- VButton.vue
|- VTable.vue
|- VIcon.vue
```

单例组件名 强烈推荐

只应该拥有单个活跃实例的组件应该以 `The` 前缀命名，以示其唯一性。

这不意味着组件只可用于一个单页面，而是*每个页面*只使用一次。这些组件永远不接受任何 prop，因为它们是为你的应用定制的，而不是它们在你的应用中的上下文。如果你发现有必要添加 prop，那就表明这实际上是一个可复用的组件，*只是目前*在每个页面里只使用一次。

反例

```
components/
|- Heading.vue
|- MySidebar.vue
```

好例子

```
components/
|- TheHeading.vue
|- TheSidebar.vue
```

紧密耦合的组件名 强烈推荐

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。

详解

你可以试着通过在其父组件命名的目录中嵌套子组件以解决这个问题。比如：

```
components/
|- TodoList/
   |- Item/
      |- index.vue
      |- Button.vue
   |- index.vue
```

或：

```
components/
|- TodoList/
   |- Item/
      |- Button.vue
   |- Item.vue
|- TodoList.vue
```

但是这种方式并不推荐，因为这会导致：

- 许多文件的名字相同，使得在编辑器中快速切换文件变得困难。
- 过多嵌套的子目录增加了在编辑器侧边栏中浏览组件所花的时间。

反例

```
components/
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
|- TodoList.vue
|- TodoItem.vue
|- TodoButton.vue
```

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

好例子

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

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

组件名中的单词顺序 强烈推荐

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。

详解

你可能会疑惑：

> “为什么我们给组件命名时不多遵从自然语言呢？”

在自然的英文里，形容词和其它描述语通常都出现在名词之前，否则需要使用连接词。比如：

- Coffee *with* milk
- Soup *of the* day
- Visitor *to the* museum

如果你愿意，你完全可以在组件名里包含这些连接词，但是单词的顺序很重要。

同样要注意**在你的应用中所谓的“高级别”是跟语境有关的**。比如对于一个带搜索表单的应用来说，它可能包含这样的组件：

```
components/
|- ClearSearchButton.vue
|- ExcludeFromSearchInput.vue
|- LaunchOnStartupCheckbox.vue
|- RunSearchButton.vue
|- SearchInput.vue
|- TermsCheckbox.vue
```

你可能注意到了，我们很难看出来哪些组件是针对搜索的。现在我们来根据规则给组件重新命名：

```
components/
|- SearchButtonClear.vue
|- SearchButtonRun.vue
|- SearchInputExcludeGlob.vue
|- SearchInputQuery.vue
|- SettingsCheckboxLaunchOnStartup.vue
|- SettingsCheckboxTerms.vue
```

因为编辑器通常会按字母顺序组织文件，所以现在组件之间的重要关系一目了然。

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

你可能想换成多级目录的方式，把所有的搜索组件放到“search”目录，把所有的设置组件放到“settings”目录。我们只推荐在非常大型 (如有 100+ 个组件) 的应用下才考虑这么做，因为：

- 在多级目录间找来找去，要比在单个 `components` 目录下滚动查找要花费更多的精力。
- 存在组件重名 (比如存在多个 `ButtonDelete` 组件) 的时候在编辑器里更难快速定位。
- 让重构变得更难，因为为一个移动了的组件更新相关引用时，查找/替换通常并不高效。

反例

```
components/
|- ClearSearchButton.vue
|- ExcludeFromSearchInput.vue
|- LaunchOnStartupCheckbox.vue
|- RunSearchButton.vue
|- SearchInput.vue
|- TermsCheckbox.vue
```

好例子

```
components/
|- SearchButtonClear.vue
|- SearchButtonRun.vue
|- SearchInputQuery.vue
|- SearchInputExcludeGlob.vue
|- SettingsCheckboxTerms.vue
|- SettingsCheckboxLaunchOnStartup.vue
```

自闭合组件 强烈推荐

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——但在 DOM 模板里永远不要这样做。

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。

反例

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

```
<!-- 在 DOM 模板中 -->
<my-component/>
```

好例子

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

```
<!-- 在 DOM 模板中 -->
<my-component></my-component>
```

模板中的组件名大小写 强烈推荐

对于绝大多数项目来说，在单文件组件和字符串模板中组件名应该总是 PascalCase 的——但是在 DOM 模板中总是 kebab-case 的。

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。
- 如果你在模板中使用任何非 Vue 的自定义元素，比如一个 Web Component，PascalCase 确保了你的 Vue 组件在视觉上仍然是易识别的。

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

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。**

反例

```
<!-- 在单文件组件和字符串模板中 -->
<mycomponent/>
```

```
<!-- 在单文件组件和字符串模板中 -->
<myComponent/>
```

```
<!-- 在 DOM 模板中 -->
<MyComponent></MyComponent>
```

好例子

```
<!-- 在单文件组件和字符串模板中 -->
<MyComponent/>
```

```
<!-- 在 DOM 模板中 -->
<my-component></my-component>
```

或者

```
<!-- 在所有地方 -->
<my-component></my-component>
```

JS/JSX 中的组件名大小写 强烈推荐

JS/JSX 中的组件名应该始终是 PascalCase 的，尽管在较为简单的应用中只使用 `Vue.component` 进行全局组件注册时，可以使用 kebab-case 字符串。

详解

在 JavaScript 中，PascalCase 是类和构造函数 (本质上任何可以产生多份不同实例的东西) 的命名约定。Vue 组件也有多份实例，所以同样使用 PascalCase 是有意义的。额外的好处是，在 JSX (和模板) 里使用 PascalCase 使得代码的读者更容易分辨 Vue 组件和 HTML 元素。

然而，对于只通过 `Vue.component` 定义全局组件的应用来说，我们推荐 kebab-case 作为替代。原因是：

- 全局组件很少被 JavaScript 引用，所以遵守 JavaScript 的命名约定意义不大。
- 这些应用往往包含许多 DOM 内的模板，这种情况下是**必须**使用 kebab-case 的。

反例

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

```
import myComponent from './MyComponent.vue'
```

```
export default {
  name: 'myComponent',
  // ...
}
```

```
export default {
  name: 'my-component',
  // ...
}
```

好例子

```
Vue.component('MyComponent', {
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
  // ...
})
```

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

```
import MyComponent from './MyComponent.vue'
```

```
export default {
  name: 'MyComponent',
  // ...
}
```

完整单词的组件名 强烈推荐

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。

反例

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

好例子

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

Prop 名大小写 强烈推荐

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。

反例

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

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

好例子

```
props: {
  greetingText: String
}
```

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

多个特性的元素 强烈推荐

多个特性的元素应该分多行撰写，每个特性一行。

在 JavaScript 中，用多行分隔对象的多个属性是很常见的最佳实践，因为这样更易读。模板和 JSX 值得我们做相同的考虑。

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

反例

```
<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"
/>
```

模板中简单的表达式 强烈推荐

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。

反例

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

好例子

```
<!-- 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(' ')
  }
}
```

简单的计算属性 强烈推荐

应该把复杂计算属性分割为尽可能多的更简单的属性。

详解

更简单、命名得当的计算属性是这样的：

- **易于测试**

  当每个计算属性都包含一个非常简单且很少依赖的表达式时，撰写测试以确保其正确工作就会更加容易。

- **易于阅读**

  简化计算属性要求你为每一个值都起一个描述性的名称，即便它不可复用。这使得其他开发者 (以及未来的你) 更容易专注在他们关心的代码上并搞清楚发生了什么。

- **更好的“拥抱变化”**

  任何能够命名的值都可能用在视图上。举个例子，我们可能打算展示一个信息，告诉用户他们存了多少钱；也可能打算计算税费，但是可能会分开展现，而

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

不是作为总价的一部分。

小的、专注的计算属性减少了信息使用时的假设性限制，所以需求变更时也用不着那么多重构了。

反例

```
computed: {
  price: function () {
    var basePrice = this.manufactureCost / (1 - this.profitMargin)
    return (
      basePrice -
      basePrice * (this.discountPercent || 0)
    )
  }
}
```

好例子

```
computed: {
  basePrice: function () {
    return this.manufactureCost / (1 - this.profitMargin)
  },
  discount: function () {
    return this.basePrice * (this.discountPercent || 0)
  },
  finalPrice: function () {
    return this.basePrice - this.discount
  }
}
```

带引号的特性值 强烈推荐

非空 HTML 特性值应该始终带引号 (单引号或双引号，选你 JS 里不用的那个)。

在 HTML 中不带空格的特性值是可以没有引号的，但这样做常常导致带空格的特征值*被回避*，导致其可读性变差。

反例

```
<input type=text>
```

```
<AppSidebar :style={width:sidebarWidth+'px'}>
```

好例子

```
<input type="text">
```

```
<AppSidebar :style="{ width: sidebarWidth + 'px' }">
```

指令缩写 强烈推荐

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

反例

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

```
<input
  v-on:input="onInput"
  @focus="onFocus"
>
```

好例子

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
<input
  :value="newTodoText"
  :placeholder="newTodoInstructions"
>
```

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

```
<input
  @input="onInput"
  @focus="onFocus"
>
```

```
<input
  v-on:input="onInput"
  v-on:focus="onFocus"
>
```

优先级 C 的规则：推荐 (将选择和认知成本最小化)

组件/实例的选项的顺序 推荐

组件/实例的选项应该有统一的顺序。

这是我们推荐的组件选项默认顺序。它们被划分为几大类，所以你也能知道从插件里添加的新属性应该放到哪里。

1. **副作用** (触发组件外的影响)

- `el`

2. **全局感知** (要求组件以外的知识)

- `name`
- `parent`

3. **组件类型** (更改组件的类型)

- `functional`

4. **模板修改器** (改变模板的编译方式)

- `delimiters`
- `comments`

5. **模板依赖** (模板内使用的资源)

- `components`
- `directives`
- `filters`

6. **组合** (向选项里合并属性)

- `extends`
- `mixins`

7. **接口** (组件的接口)

- `inheritAttrs`
- `model`
- `props` / `propsData`

8. **本地状态** (本地的响应式属性)

- `data`
- `computed`

9. **事件** (通过响应式事件触发的回调)

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

- `watch`
- 生命周期钩子 (按照它们被调用的顺序)

10. **非响应式的属性** (不依赖响应系统的实例属性)

- `methods`

11. **渲染** (组件输出的声明式描述)

- `template` / `render`
- `renderError`

元素特性的顺序 推荐

元素 (包括组件) 的特性应该有统一的顺序。

这是我们为组件选项推荐的默认顺序。它们被划分为几大类，所以你也能知道新添加的自定义特性和指令应该放到哪里。

1. **定义** (提供组件的选项)

- `is`

2. **列表渲染** (创建多个变化的相同元素)

- `v-for`

3. **条件渲染** (元素是否渲染/显示)

- `v-if`
- `v-else-if`
- `v-else`
- `v-show`
- `v-cloak`

4. **渲染方式** (改变元素的渲染方式)

- `v-pre`
- `v-once`

5. **全局感知** (需要超越组件的知识)

- `id`

6. **唯一的特性** (需要唯一值的特性)

- `ref`
- `key`
- `slot`

7. **双向绑定** (把绑定和事件结合起来)

- `v-model`

8. **其它特性** (所有普通的绑定或未绑定的特性)

9. **事件** (组件事件监听器)

- `v-on`

10. **内容** (覆写元素的内容)

- `v-html`
- `v-text`

组件/实例选项中的空行 推荐

你可能想在多个属性之间增加一个空行，特别是在这些选项一屏放不下，需要滚动才能都看到的时候。

当你的组件开始觉得密集或难以阅读时，在多个属性之间添加空行可以让其变得容易。在一些诸如 Vim 的编辑器里，这样格式化后的选项还能通过键盘被快速导航。

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

好例子

```
props: {
  value: {
    type: String,
    required: true
  },

  focused: {
    type: Boolean,
    default: false
  },

  label: String,
  icon: String
},

computed: {
  formattedValue: function () {
    // ...
  },

  inputClasses: function () {
    // ...
  }
}
```

```
// 没有空行在组件易于阅读和导航时也没问题。
props: {
  value: {
    type: String,
    required: true
  },
  focused: {
    type: Boolean,
    default: false
  },
  label: String,
  icon: String
},
computed: {
  formattedValue: function () {
    // ...
  },
  inputClasses: function () {
    // ...
  }
}
```

单文件组件的顶级元素的顺序 推荐

单文件组件应该总是让 `<script>`、`<template>` 和 `<style>` 标签的顺序保持一致。且 `<style>` 要放在最后，因为另外两个标签至少要有一个。

反例

```
<style>/* ... */</style>
<script>/* ... */</script>
<template>...</template>
```

```
<!-- ComponentA.vue -->
<script>/* ... */</script>
<template>...</template>
<style>/* ... */</style>

<!-- ComponentB.vue -->
<template>...</template>
<script>/* ... */</script>
<style>/* ... */</style>
```

好例子

```
<!-- ComponentA.vue -->
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
<script>/* ... */</script>
<template>...</template>
<style>/* ... */</style>

<!-- ComponentB.vue -->
<script>/* ... */</script>
<template>...</template>
<style>/* ... */</style>
```

```
<!-- ComponentA.vue -->
<template>...</template>
<script>/* ... */</script>
<style>/* ... */</style>

<!-- ComponentB.vue -->
<template>...</template>
<script>/* ... */</script>
<style>/* ... */</style>
```

优先级 D 的规则：谨慎使用 (有潜在危险的模式)

没有在 `v-if` / `v-if-else` / `v-else` 中使用 `key` 谨慎使用

如果一组 `v-if` + `v-else` 的元素类型相同，最好使用 `key` (比如两个 `<div>` 元素)。

默认情况下，Vue 会尽可能高效的更新 DOM。这意味着其在相同类型的元素之间切换时，会修补已存在的元素，而不是将旧的元素移除然后在同一位置添加一个新元素。如果本不相同的元素被识别为相同，则会出现意料之外的副作用。

反例

```
<div v-if="error">
  错误：{{ error }}
</div>
<div v-else>
  {{ results }}
</div>
```

好例子

```
<div
  v-if="error"
  key="search-status"
>
  错误：{{ error }}
</div>
<div
  v-else
  key="search-results"
>
  {{ results }}
</div>
```

```
<p v-if="error">
  错误：{{ error }}
</p>
<div v-else>
  {{ results }}
</div>
```

`scoped` 中的元素选择器 谨慎使用

元素选择器应该避免在 `scoped` 中出现。

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。

详解

为了给样式设置作用域，Vue 会为元素添加一个独一无二的特性，例如 `data-v-f3f3eg9`。然后修改选择器，使得在匹配选择器的元素中，只有带这个特性才会

© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

真正生效 (比如 `button[data-v-f3f3eg9]`)。

问题在于大量的元素和特性组合的选择器 (比如 `button[data-v-f3f3eg9]`) 会比类和特性组合的选择器 慢，所以应该尽可能选用类选择器。

反例

```
<template>
  <button>X</button>
</template>

<style scoped>
button {
  background-color: red;
}
</style>
```

好例子

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

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

隐性的父子组件通信 谨慎使用

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` 或改变 prop。

一个理想的 Vue 应用是 prop 向下传递，事件向上传递的。遵循这一约定会让你的组件更易于理解。然而，在一些边界情况下 prop 的变更或 `this.$parent` 能够简化两个深度耦合的组件。

问题在于，这种做法在很多*简单*的场景下可能会更方便。但请当心，不要为了一时方便 (少写代码) 而牺牲数据流向的简洁性 (易于理解)。

反例

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

```
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }
  },
  methods: {
    removeTodo () {
      var vm = this
      vm.$parent.todos = vm.$parent.todos.filter(function (todo) {
        return todo.id !== vm.todo.id
      })
    }
  },
  template: `
    <span>
      {{ todo.text }}
      <button @click="removeTodo">
        X
      </button>
    </span>
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
  `
})
```

好例子

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

```
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }
  },
  template: `
    <span>
      {{ todo.text }}
      <button @click="$emit('delete')">
        X
      </button>
    </span>
  `
})
```

非 Flux 的全局状态管理 谨慎使用

应该优先通过 Vuex 管理全局状态，而不是通过 `this.$root` 或一个全局事件总线。

通过 `this.$root` 和/或全局事件总线管理状态在很多简单的情况下都是很方便的，但是并不适用于绝大多数的应用。Vuex 提供的不仅是一个管理状态的中心区域，还是组织、追踪和调试状态变更的好工具。

反例

```
// main.js
new Vue({
  data: {
    todos: []
  },
  created: function () {
    this.$on('remove-todo', this.removeTodo)
  },
  methods: {
    removeTodo: function (todo) {
      var todoIdToRemove = todo.id
      this.todos = this.todos.filter(function (todo) {
        return todo.id !== todoIdToRemove
      })
    }
  }
})
```

好例子

```
// store/modules/todos.js
export default {
  state: {
    list: []
  },
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME

```
  mutations: {
    REMOVE_TODO (state, todoId) {
      state.list = state.list.filter(todo => todo.id !== todoId)
    }
  },
  actions: {
    removeTodo ({ commit, state }, todo) {
      commit('REMOVE_TODO', todo.id)
    }
  }
}
```

```
<!-- TodoItem.vue -->
<template>
  <span>
    {{ todo.text }}
    <button @click="removeTodo(todo)">
      X
    </button>
  </span>
</template>

<script>
import { mapActions } from 'vuex'

export default {
  props: {
    todo: {
      type: Object,
      required: true
    }
  },
  methods: mapActions(['removeTodo'])
}
</script>
```


© Copyright Friday, Jun 1, 2018, 9:50 AM by COMPANYNAME