基于模块开发

始终基于模块的方式来构建你的 *app*，每一个子模块只做一件事情。

Vue.js 的设计初衷就是帮助开发者更好的开发界面模块。一个模块是应用程序中独立的一个部分。

怎么做？

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。

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。

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。
- ***app-* 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。

如何做？

```
<!-- 推荐 -->

<app-header></app-header>

<user-list></user-list>

<range-slider></range-slider>

<!-- 避免 -->

<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 -->
```

编制：Vue.js 研发群 165862199

```
<slider></slider> <!-- 与自定义元素规范不兼容 -->
```

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。

```
<!-- 推荐 -->

<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>

<!-- 避免 -->

<template>

  <h1>

    {{ `${(new Date()).getUTCFullYear()}-${('0' + ((new Date()).getUTCMonth()+1)).slice(-2)}` }}

  </h1>

</template>
```

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。

怎么做？

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。

```
<!-- 推荐 -->

<range-slider

  :values="[10, 20]"

  min="0"

  max="100"

  step="5"

  :on-slide="updateInputs"

  :on-end="updateResults">

</range-slider>


<!-- 避免 -->

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

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, // 这里添加了数字类型的校验

default() { return 10; },  },  min: {  type: Number,  default() { return 0; },  },  value:

{  type: Number,  default() { return 4; },  },  };</script>
```

↑ 回到目录

将 *this* 赋值给 *component* 变量

在 *Vue.js* 组件上下文中，*this* 指向了组件实例。因此当你切换到了不同的上下文时，要确保 *this* 指向一个可用的 *component* 变量。

换句话说，不要在编写这样的代码 *const self = this;* ，而是应该直接使用变量 *component*。

为什么？

- 将组件 *this* 赋值给变量 *component* 可用让开发者清楚的知道任何一个被使用的地方，它代表的是组件实例。

怎么做？

```
<script type="text/javascript">export default {  methods: {  hello() {  return 'hello';  },  printHello()

{  console.log(this.hello());  },  },};</script>

<!-- 避免 -->

<script type="text/javascript">export default {  methods: {  hello() {  return 'hello';  },  printHello() {  const

self = this; // 没有必要  console.log(self.hello());  },  },};</script>
```

组件结构化

按照一定的结构组织，使得组件便于理解。

编制：Vue.js 研发群 165862199

为什么？

- 导出一个清晰、组织有序的组件，使得代码易于阅读和理解。同时也便于标准化。
- 按首字母排序 *properties*、*data*、*computed*、*watches* 和 *methods* 使得这些对象内的属性便于查找。
- 合理组织，使得组件易于阅读。（*name*; *extends*; *props*, *data* 和 *computed*; *components*; *watch* 和 *methods*; *lifecycle methods* 等）。
- 使用 *name* 属性。借助于 *vue devtools* 可以让你更方便的测试。
- 合理的 *CSS* 结构，如 *BEM* 或 *rscss* - 详情？。
- 使用单文件 *.vue* 文件格式来组件代码。

怎么做？

组件结构化

```
<template lang="html">

  <div class="Ranger_Wrapper">

    <!-- ... -->

  </div>

</template>



<script type="text/javascript">  export default {    // 不要忘记了 name 属性    name: 'RangeSlider',    // 组合其它组件
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    beforeCreate() {},    mounted() {},  };</script>



<style scoped>  .Ranger_Wrapper { /* ... */ }</style>
```

↑ 回到目录

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？

编制：Vue.js 研发群 165862199

- 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 形容词（如 *drive-upload-success*）结尾。（出处）

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。

谨慎使用 *this.$refs*

Vue.js 支持通过 *ref* 属性来访问其它组件和 *HTML* 元素。并通过 *this.$refs* 可以得到组件或 *HTML* 元素的上下文。在大多数情况下，通过 *this.$refs* 来访问其它组件的上下文是可以避免的。在使用的的时候你需要注意避免调用了不恰当的组件 *API*，所以应该尽量避免使用 *this.$refs*。

为什么？

- 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。
- 组件的属性和事件必须足够的给大多数的组件使用。

怎么做？

- 提供良好的组件 *API*。
- 总是关注于组件本身的目的。

编制：Vue.js 研发群 165862199

- 拒绝定制代码。如果你在一个通用的组件内部编写特定需求的代码，那么代表这个组件的 API 不够通用，或者你可能需要一个新的组件来应对该需求。
- 检查所有的 *props* 是否有缺失的，如果有提一个 *issue* 或是完善这个组件。
- 检查所有的事件。子组件向父组件通信一般是通过事件来实现的，但是大多数的开发者更多的关注于 *props* 从忽视了这点。
- **Props 向下传递，事件向上传递！**。以此为目标升级你的组件，提供良好的 API 和 独立性。
- 当遇到 *props* 和 *events* 难以实现的功能时，通过 *this.$refs* 来实现。
- 当需要操作 DOM 无法通过指令来做的时候可使用 *this.$ref* 而不是 *JQuery*、*document.getElement**、*document.queryElement*。

```
<!-- 推荐，并未使用 this.$refs -->
<range :max="max"
   :min="min"
   @current-value="currentValue"
   :step="1"></range>
<!-- 使用 this.$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>

<!-- 如果可通过 emited 来做则避免通过 this.$refs 直接访问 -->

<template>

  <range :max="max"

    :min="min"

    ref="range"

    :step="1"></range>

</template>
```

```
<script>  export default {    // ...    methods: {      getRangeCurrentValue() {        return this.$refs.range.currentValue;      },    },    // ...  };</script>
```

使用组件名作为样式作用域空间

Vue.js 的组件是自定义元素，这非常适合用来作为样式的根作用域空间。可以将组件名作为 CSS 类的命名空间。

为什么？

- 给样式加上作用域空间可以避免组件样式影响外部的样式。
- 保持模块名、目录名、样式根作用域名一样，可以很好的将其关联起来，便于开发者理解。

怎么做？

使用组件名作为样式命名的前缀，可基于 BEM 或 OOCSS 范式。同时给 style 标签加上 scoped 属性。加上 scoped 属性编译后会给组件的 class 自动加上唯一的前缀从而避免样式的冲突。

```
<style scoped>  /* 推荐 */  .MyExample {}  .MyExample li {}  .MyExample__item {}  /* 避免 */  .My-Example {} /* 没有用组件名或模块名限制作用域，不符合 BEM 规范 */</style>
```

提供组件 API 文档

使用 Vue.js 组件的过程中会创建 Vue 组件实例，这个实例是通过自定义属性配置的。为了便于其他开发者使用该组件，对于这些自定义属性即组件 API 应该在 README.md 文件中进行说明。

为什么？

- 良好的文档可以让开发者比较容易的对组件有一个整体的认识，而不用去阅读组件的源码，也更方便开发者使用。
- 组件配置属性即组件的 *API*，对于组件的用户来说他们更感兴趣的是 *API* 而不是实现原理。
- 正式的文档会告诉开发者组件 *API* 变更以及向后的兼容性情况。
- *README.md* 是标准的我们应该首先阅读的文档文件。代码托管网站（*GitHub*、*Bitbucket*、*Gitlab* 等）会默认在仓库中展示该文件作为仓库的介绍。

怎么做？

在模块目录中添加 *README.md* 文件：

```
range-slider/
├── range-slider.vue
├── range-slider.less
└── README.md
```

在 *README* 文件中说明模块的功能以及使用场景。对于 *vue* 组件来说，比较有用的描述是组件的自定义属性即 *API* 的描述介绍。

Range slider

功能

range slider 组件可通过拖动的方式来设置一个给定范围内的数值。

该模块使用 *noUiSlider* 来实现跨浏览器和 *touch* 功能的支持。

如何使用

<range-slider> 支持如下的自定义属性：

attribute	type	description
min	Number	可拖动的最小值.
max	Number	可拖动的最大值.
values	Number[]optional	包含最大值和最小值的数组. 如. values="[10, 20]". Defaults to [opts.min, opts.max].
step	Numberoptional	增加减小的数值单位，默认为 1.

编制：Vue.js 研发群 165862199

attribute	type	description
on-slide	Functionoptional	用户拖动开始按钮或者结束按钮时的回调函数，函数接受 (values, HANDLE) 格式的参数。 如：on-slide={ updateInputs }, component.updateInputs = (values, HANDLE) => { const value = values[HANDLE]; }.
on-end	Functionoptional	当用户停止拖动时触发的回调函数，函数接受 (values, HANDLE) 格式的参数。

如需要自定义 slider 的样式可参考 noUiSlider 文档

提供组件 demo

添加 index.html 文件作为组件的 demo 示例，并提供不同配置情况的效果，说明组件是如何使用的。

为什么？

- demo 可以说明组件是独立可使用的。
- demo 可以让开发者预览组件的功能效果。
- demo 可以展示组件各种配置参数下的功能。

↑ 回到目录

对组件文件进行代码校验

代码校验可以保持代码的统一性以及追踪语法错误。.vue 文件可以通过使用 eslint-plugin-html 插件来校验代码。你可以通过 vue-cli 来开始你的项目，vue-cli 默认会开启代码校验功能。

为什么？

- 保证所有的开发者使用同样的编码规范。
- 更早的感知到语法错误。

怎么做？

为了校验工具能够校验 *.vue 文件，你需要将代码编写在 <script>标签中，并使组件表达式简单化，因为校验工具无法理解行内表达式，配置校验工具可以访问全局变量 vue 和组件的 props。

ESLint

ESLint 需要通过 ESLint HTML 插件来抽取组件中的代码。

通过 .eslintrc 文件来配置 ESlint，这样 IDE 可以更好的理解校验配置项：

```
{
```

```
"extends": "eslint:recommended",

"plugins": ["html"],

"env": {

    "browser": true

},

"globals": {

    "opts": true,

    "vue": true

  }

}
```

运行 *ESLint*

```
eslint src/**/*.vue
```

JSHint

JSHint 可以解析 *HTML*（使用 *--extra-ext* 命令参数）和抽取代码（使用 *--extract=auto* 命令参数）。

通过 *.jshintrc* 文件来配置 *ESlint*，这样 *IDE* 可以更好的理解校验配置项。

```
{

  "browser": true,

  "predef": ["opts", "vue"]

}
```

运行 *JSHint*

```
jshint --config modules/.jshintrc --extra-ext=html --extract=auto modules/
```

注：*JSHint* 不接受 *vue* 扩展名的文件，只支持 *html*。

只在需要时创建组件

为什么？

编制：Vue.js 研发群 165862199

Vue.js 是一个基于组件的框架。如果你不知道何时创建组件可能会导致以下问题：

- 如果组件太大，可能很难重用和维护;
- 如果组件太小，你的项目就会（因为深层次的嵌套而）被淹没，也更难使组件间通信;

怎么做?

- 始终记住为你的项目需求构建你的组件，但是你也应该尝试想到它们能够从中脱颖而出（独立于项目之外）。如果它们能够在你项目之外工作，就像一个库那样，就会使得它们更加健壮和一致。
- 尽可能早地构建你的组件总是更好的，因为这样使得你可以在一个已经存在和稳定的组件上构建你的组件间通信（*props & events*）。

规则

- 首先，尽可能早地尝试构建出诸如模态框、提示框、工具条、菜单、头部等这些明显的（通用型）组件。总之，你知道的这些组件以后一定会在当前页面或者是全局范围内需要。
- 第二，在每一个新的开发项目中，对于一整个页面或者其中的一部分，在进行开发前先尝试思考一下。如果你认为它有一部分应该是一个组件，那么就创建它吧。
- 最后，如果你不确定，那就不要。避免那些“以后可能会有用”的组件污染你的项目。它们可能会永远的只是（静静地）待在那里，这一点也不聪明。注意，一旦你意识到应该这么做，最好是就把它打破，以避免与项目的其他部分构成兼容性和复杂性。