VUE官网笔记整理

# 基础

## 安装

兼容：VUE不兼容IE8以下的浏览器

插件：Vue Devtools

CND方式: <script src="https://cdn.jsdelivr.net/npm/vue@2.5.17/dist/vue.js"></script> //通过script直接引入

NPM方式：npm install vue //最新稳定版

CLI（命令行工具）：npm i vue-cli

## 介绍

Script引入：<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script> //此为开发版本

<script src="https://cdn.jsdelivr.net/npm/vue"></script> //此为生产版本

## 声明式渲染

<div id="app">

<p>{{ message }}</p> //直接用模板渲染数据

<input type="text" name="" id="" value="" v-model="message"/> //用model双向数据绑定

<p v-bind:title="message2">悬停查看提示信息</p> //bind数据绑定属性

<p v-if="seen">you can see me</p> //判断是否显示

<ul>

<li v-for="item in list">{{item.text}}</li> //for循环数据

<mycomponent v-for="item in list" v-bind:data='item' v-bind:key="item.id"></mycomponet> //vue组件

</ul>

<button v-on:click="reset">reset message</button> //按钮事件

</div>

Vue.component('mycomponent',{

props: ['data'],

template: '<li>{{data.text}}</li>'

}) //设置自定义组件

var app = new Vue({

el: '#app',

data: {

message: 'Hello Vue!', //定义数据

message2: '页面加载于 ' + new Date().toLocaleString(),

seen: true,

list: [

{text: 'asd',id: '01'},

{text: 'bdf',id: '02'},

{text: 'dagad',id: '03'}

]

},

methods: { //挂载

reset: function(){

var \_self =this;

\_self.message = \_self.message.split('').reverse().join('')

}

}

})

## 模板语法

### 缩写

v-bind:attr='' === :attr=''

v-on:click='' === @click=''

## 计算属性和侦听器

### 计算属性

<p>Original message: "{{ reversedMessage }}"</p>

computed: {  
 reversedMessage: function () {  
 return this.message.split('').reverse().join('')  
 }  
 }

当 vm.message 发生改变时，所有依赖 vm.reversedMessage 的绑定也会更新

#### 计算属性缓存

计算属性是基于它们的依赖进行缓存的。只在相关依赖发生改变时它们才会重新求值。例如

computed: {  
 now: function () {  
 return Date.now()  
 }  
}

Date.now() 不是响应式依赖，所以重复使用now获取到的都是同一个时间，相比之下，每当触发重新渲染时，调用方法将**总会**再次执行函数。

#### 计算属性与侦听器

watch: {  
 firstName: function (val) {  
 this.fullName = val + ' ' + this.lastName  
 },  
 lastName: function (val) {  
 this.fullName = this.firstName + ' ' + val  
 }  
 }

computed: {  
 fullName: function () {  
 return this.firstName + ' ' + this.lastName  
 }  
 }

当你有一些数据需要随着其它数据变动而变动时，更好的做法是使用计算属性而不是命令式的 watch 回调。

### 侦听器

需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的

<input v-model="question">  
watch: {  
   // 如果 `question` 发生改变，这个函数就会运行  
 question: function (newQuestion, oldQuestion) {  
 this.answer = 'Waiting for you to stop typing...'  
 this.debouncedGetAnswer()  
 }  
 }

## Class与style绑定

### 绑定HTML Class

#### 对象语法

<div v-bind:class="{ active: isActive }"></div>

active 这个 class 存在与否将取决于数据属性 isActive 的 [**truthiness**](https://developer.mozilla.org/zh-CN/docs/Glossary/Truthy)

<div class="static"  
 v-bind:class="{ active: isActive, 'text-danger': hasError }">  
</div>

Class与:class**共存**，切:class可以存在多个变量

|  |
| --- |
| <div v-bind:class="classObject"></div> |

|  |
| --- |
| data: {  classObject: {  active: true,  'text-danger': false  } } |

也可不必写在**内联**里

computed: {  
 classObject: function () {  
 return {  
 active: this.isActive && !this.error,  
 'text-danger': this.error && this.error.type === 'fatal'  
 }  
 }  
}

也可以是**计算属性**

#### 数组语法

<div v-bind:class="[activeClass, errorClass]"></div>

<div v-bind:class="[isActive ? activeClass : '', errorClass]"></div>

可以是三元

<div v-bind:class="[{ active: isActive }, errorClass]"></div>

数组中写三元语法

#### 组件上使用

Vue.component('my-component', {  
 template: '<p class="foo bar">Hi</p>'  
})

<my-component class="baz boo"></my-component>

直接添加，也可以使用对象或数组语法

### 绑定内联样式

#### 对象语法

CSS 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用单引号括起来) 来命名：

<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

直接绑定对象

|  |
| --- |
| <div v-bind:style="styleObject"></div> |

|  |
| --- |
| data: {  styleObject: {  color: 'red',  fontSize: '13px'  } } |

#### 数组语法

可以将多个样式绑定到元素上

<div v-bind:style="[baseStyles, overridingStyles]"></div>

#### 自动添加前缀

当 v-bind:style 使用需要添加[**浏览器引擎前缀**](https://developer.mozilla.org/zh-CN/docs/Glossary/Vendor_Prefix)的 CSS 属性时，如 transform，Vue.js 会自动侦测并添加相应的前缀。

#### 多重值

从 2.3.0 起你可以为 style 绑定中的属性提供一个包含多个值的数组，常用于提供多个带前缀的值，例如：

<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>

只会渲染数组中最后一个被浏览器支持的值

## 条件渲染

### v-if

<h1 v-if="ok">Yes</h1>

<h1 v-else>No</h1>

#### Template元素上使用

切换多个元素时，可以把一个 <template> 元素当做不可见的包裹元素，并在上面使用 v-if。最终的渲染结果将不包含 <template> 元素。

<template v-if="ok">  
 <h1>Title</h1>  
 <p>Paragraph 1</p>  
 <p>Paragraph 2</p>  
</template>

#### v-else

v-else 元素必须紧跟在带 v-if 或者 v-else-if 的元素的后面，否则它将不会被识别。

#### v-else-if

版本：**2.1.0 新增**

类似于 v-else，v-else-if 也必须紧跟在带 v-if 或者 v-else-if 的元素之后。

#### 用key管理可复用的元素

Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。这么做除了使 Vue 变得非常快之外，还有其它一些好处。例如，如果你允许用户在不同的登录方式之间切换：

<template v-if="loginType === 'username'">  
 <label>Username</label>  
 <input placeholder="Enter your username">  
</template>  
<template v-else>  
 <label>Email</label>  
 <input placeholder="Enter your email address">  
</template>

再重复的input上增加key，切换时输入框都将被重新渲染

<template v-if="loginType === 'username'">  
 <label>Username</label>  
 <input placeholder="Enter your username" key="username-input">  
</template>  
<template v-else>  
 <label>Email</label>  
 <input placeholder="Enter your email address" key="email-input">  
</template>

### v-show

v-show 的元素始终会被渲染并保留在 DOM 中。v-show 只是简单地切换元素的 CSS 属性 display

<h1 v-show="ok">Hello!</h1>

### v-if与v-show

v-if 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块

v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好

### v-if与v-for

**不推荐**同时使用 v-if 和 v-for

## 列表渲染

### 遍历数组

items 是源数据数组并且 item 是数组元素迭代的别名

<li v-for="item in items">  
 {{ item.message }}  
</li>

data: {  
 items: [  
 { message: 'Foo' },  
 { message: 'Bar' }  
 ]  
 }

v-for 还支持一个可选的第二个参数为当前项的索引

<li v-for="(item, index) in items">  
 {{ parentMessage }} - {{ index }} - {{ item.message }}  
</li>

可以用 of 替代 in 作为分隔符，因为它是最接近 JavaScript 迭代器的语法

<div v-for="item of items"></div>

### 遍历对象

<li v-for="value in object">  
 {{ value }}  
</li>

object: {  
 firstName: 'John',  
 lastName: 'Doe',  
 age: 30  
}

第二个的参数为键名

<div v-for="(value, key) in object">  
 {{ key }}: {{ value }}  
</div>

第三个参数为索引

<div v-for="(value, key, index) in object">  
 {{ index }}. {{ key }}: {{ value }}  
</div>

Ps: 在遍历对象时，是按 Object.keys() 的结果遍历，但是不能保证它的结果在不同的 JavaScript 引擎下是一致的

### key

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素.

这个默认的模式是高效的，但是只适用于**不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出**。

它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key 属性。理想的 key 值是每项都有的唯一 id。

<div v-for="item in items" :key="item.id">  
 <!-- 内容 -->  
</div>

### 数组更新检测

#### 变异方法

以下方法会触发视图更新

push、pop、shift、unshift、splice、sort、reverse

#### 替换数组

变异方法 (mutation method)，顾名思义，会改变被这些方法调用的原始数组。相比之下，也有非变异 (non-mutating method) 方法，例如：filter(), concat() 和 slice() 。这些不会改变原始数组，但**总是返回一个新数组**。当使用非变异方法时，可以用新数组替换旧数组：

example1.items = example1.items.filter(function (item) {  
 return item.message.match(/Foo/)  
})

#### 注意事项

由于 JavaScript 的限制，Vue 不能检测以下变动的数组：

1，当你利用索引直接设置一个项时，例如：vm.items[indexOfItem] = newValue

2，,当你修改数组的长度时，例如：vm.items.length = newLength

var vm = new Vue({  
 data: {  
 items: ['a', 'b', 'c']  
 }  
})  
vm.items[1] = 'x' // 不是响应性的  
vm.items.length = 2 // 不是响应性的

可以使用以下方法

|  |
| --- |
| // Vue.set Vue.set(vm.items, indexOfItem, newValue) |

|  |
| --- |
| // Array.prototype.splice vm.items.splice(indexOfItem, 1, newValue) |

vm.$set(vm.items, indexOfItem, newValue)

### 对象更改检测注意事项

还是由于 JavaScript 的限制，**Vue 不能检测对象属性的添加或删除**：

可以使用Vue.set(object, key, value)

使用 Object.assign() 或 \_.extend()时应该使用：

vm.userProfile = Object.assign({}, vm.userProfile, {  
 age: 27,  
 favoriteColor: 'Vue Green'  
})

### 显示过滤/排序结果

可以创建返回过滤或排序数组的计算属性

<li v-for="n in evenNumbers">{{ n }}</li>

data: {  
 numbers: [ 1, 2, 3, 4, 5 ]  
},  
computed: {  
 evenNumbers: function () {  
 return this.numbers.filter(function (number) {  
 return number % 2 === 0  
 })  
 }  
}

在计算属性不适用的情况下可以使用method方法

methods: {  
 even: function (numbers) {  
 return numbers.filter(function (number) {  
 return number % 2 === 0  
 })  
 }  
}

### 有范围的v-for

<span v-for="n in 10">{{ n }} </span>

### Template上使用v-for

<ul>  
 <template v-for="item in items">  
 <li>{{ item.msg }}</li>  
 <li class="divider" role="presentation"></li>  
 </template>  
</ul>

### v-for与v-if

当它们处于同一节点，v-for 的优先级比 v-if 更高，这意味着 v-if 将分别重复运行于每个 v-for 循环中。当你想为仅有的一些项渲染节点时，这种优先级的机制会十分有用

<li v-for="todo in todos" v-if="!todo.isComplete">  
 {{ todo }}  
</li>

### 一个组件的v-for

可以正常使用， 只不过需要通过组件的porp传入值

<my-component v-for="item in items" :key="item.id"></my-component>

<my-component  
 v-for="(item, index) in items"  
 v-bind:item="item"  
 v-bind:index="index"  
 v-bind:key="item.id"  
></my-component>

is="todo-item" 属性可以避开一些潜在的浏览器解析错误

<ul>  
 <li  
 is="todo-item"  
 v-for="(todo, index) in todos"  
 v-bind:key="todo.id"  
 v-bind:title="todo.title"  
 v-on:remove="todos.splice(index, 1)"  
 ></li>  
 </ul>

## 事件处理

### 监听事件

<button v-on:click="counter += 1">Add 1</button>

### 事件处理方法

<button v-on:click="greet">Greet</button>

methods: {  
 greet: function (event) {  
 // `this` 在方法里指向当前 Vue 实例  
 alert('Hello ' + this.name + '!')  
 // `event` 是原生 DOM 事件  
 if (event) {  
 alert(event.target.tagName)  
 }  
 }  
 }

### 内联处理器中的方法

<button v-on:click="say('hi')">Say hi</button>

methods: {  
 say: function (message) {  
 alert(message)  
 }  
 }

可以用特殊变量 $event 把它传入方法

<button v-on:click="warn('Form cannot be submitted yet.', $event)">  
 Submit  
</button>

methods: {  
 warn: function (message, event) {  
 // 现在我们可以访问原生事件对象  
 if (event) event.preventDefault()  
 alert(message)  
 }  
}

### 事件修饰符

Vue提供了以下修饰符

.stop 阻止单击事件继续传播

.prevent 阻止默认事件，例如提交是不刷新页面

.captrue 事件捕获模式，触发的事件现在此处处理，然后才由内部元素处理

.self 仅在当前元素触发

.once 只会触发一次 ps:2.1.3新增

.passive 触发事件的默认行为，主要用在移动端滚动事件

修饰符顺序很重要v-on:click.prevent.self 会阻止**所有的点击**，而 v-on:click.self.prevent 只会阻止对元素自身的点击。

不要把 .passive 和 .prevent 一起使用，因为 .prevent 将会被忽略，同时浏览器可能会向你展示一个警告。请记住，.passive 会告诉浏览器你不想阻止事件的默认行为。

### 按键修饰符

<!-- 只有在 `keyCode` 是 13 时调用 `vm.submit()` -->  
<input v-on:keyup.13="submit">

别名

<!-- 同上 -->  
<input v-on:keyup.enter="submit">

全部默认按键别名有

.enter

.tab

.delete 捕获删除与退格键

.esc

.space

.up

.down

.left

.right

可以通过全局 config.keyCodes 对象**[自定义按键修饰符别名](https://cn.vuejs.org/v2/api/" \l "keyCodes)**：

|  |
| --- |
| // 可以使用 `v-on:keyup.f1` Vue.config.keyCodes.f1 = 112 |

#### 自动匹配按键修饰符

可直接将 **[KeyboardEvent.key](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key_Values" \t "_blank)** 暴露的任意有效按键名转换为 kebab-case 来作为修饰符：

<input @keyup.page-down="onPageDown">

有一些按键 (.esc 以及所有的方向键) 在 IE9 中有不同的 key 值, 如果你想支持 IE9，它们的内置别名应该是首选

### 系统修饰键

2.1.0新增

.ctrl

.alt

.shift

.meta

**Ps：在 Mac 系统键盘上，meta 对应 command 键 (⌘)。在 Windows 系统键盘 meta 对应 Windows 徽标键 (⊞)。在 Sun 操作系统键盘上，meta 对应实心宝石键 (◆)。在其他特定键盘上，尤其在 MIT 和 Lisp 机器的键盘、以及其后继产品，比如 Knight 键盘、space-cadet 键盘，meta 被标记为“META”。在 Symbolics 键盘上，meta 被标记为“META”或者“Meta”**

实例

<!-- Alt + C -->  
<input @keyup.alt.67="clear">  
<!-- Ctrl + Click -->  
<div @click.ctrl="doSomething">Do something</div>

Ps：请注意修饰键与常规按键不同，在和 keyup 事件一起用时，事件触发时修饰键必须处于按下状态。换句话说，只有在按住 ctrl 的情况下释放其它按键，才能触发 keyup.ctrl。而单单释放 ctrl 也不会触发事件。如果你想要这样的行为，请为 ctrl 换用 keyCode：keyup.17

#### .exact修饰符

2.5.0新增

.exact 修饰符允许你控制由精确的系统修饰符组合触发的事件。

<!-- 即使 Alt 或 Shift 被一同按下时也会触发 -->  
<button @click.ctrl="onClick">A</button>  
  
<!-- 有且只有 Ctrl 被按下的时候才触发 -->  
<button @click.ctrl.exact="onCtrlClick">A</button>  
  
<!-- 没有任何系统修饰符被按下的时候才触发 -->  
<button @click.exact="onClick">A</button>

#### 鼠标按钮修饰符

2.2.0新增

这些修饰符会限制处理函数仅响应特定的鼠标按钮。

.left

.right

.middle

### 为什么在HTML中监听事件

你可能注意到这种事件监听的方式违背了关注点分离 (separation of concern) 这个长期以来的优良传统。但不必担心，因为所有的 Vue.js 事件处理方法和表达式都严格绑定在当前视图的 ViewModel 上，它不会导致任何维护上的困难。实际上，使用 v-on 有几个好处：

1，扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。

2，因为你无须在 JavaScript 里手动绑定事件，你的 ViewModel 代码可以是非常纯粹的逻辑，和 DOM 完全解耦，更易于测试。

3，当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何清理它们。

## 表单输入绑定

### 基础用法

v-model, 本质上不过是语法糖。它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理。

Ps:v-model 会忽略所有表单元素的 value、checked、selected 特性的初始值而总是将 Ps:Vue 实例的数据作为数据来源

对于需要使用[输入法](https://zh.wikipedia.org/wiki/%E8%BE%93%E5%85%A5%E6%B3%95) (如中文、日文、韩文等) 的语言，你会发现 v-model 不会在输入法组合文字过程中得到更新。如果你也想处理这个过程，请使用 input 事件。

#### 文本

<input v-model="message" placeholder="edit me">

#### 多行文本

<textarea v-model="message" placeholder="add multiple lines"></textarea>

Ps：在文本区域插值 (<textarea></textarea>) 并不会生效，应用 v-model 来代替

#### 复选框

<input type="checkbox" id="checkbox" v-model="checked">

多个复选框，绑定到同一个数组

data: {  
 checkedNames: []  
}

#### 单选框

<input type="radio" id="one" value="One" v-model="picked">

#### 下拉框

<select v-model="selected">  
 <option disabled value="">请选择</option>  
 <option>A</option>  
 <option>B</option>  
 <option>C</option>  
 </select>

Ps：如果 v-model 表达式的初始值未能匹配任何选项，<select> 元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第一个选项。因为这样的情况下，iOS 不会触发 change 事件。因此，更推荐像上面这样提供一个值为空的禁用选项。

多选时 (绑定到一个数组)：

<select v-model="selected" multiple style="width: 50px;">  
 <option>A</option>  
 <option>B</option>  
 <option>C</option>  
 </select>

data: {  
 selected: []  
 }

v-for渲染的动态选项

<select v-model="selected">  
 <option v-for="option in options" v-bind:value="option.value">  
 {{ option.text }}  
 </option>  
</select>

data: {  
 selected: 'A',  
 options: [  
 { text: 'One', value: 'A' },  
 { text: 'Two', value: 'B' },  
 { text: 'Three', value: 'C' }  
 ]  
 }

### 绑定值

v-model 绑定的值通常是静态字符串 (对于复选框也可以是布尔值)

绑定动态属性用 v-bind 实现，并且这个属性的值可以不是字符串

#### 复选框

<input  
 type="checkbox"  
 v-model="toggle"  
 true-value="yes"  
 false-value="no"  
>

// 当选中时  
vm.toggle === 'yes'  
// 当没有选中时  
vm.toggle === 'no'

Ps：这里的 true-value 和 false-value 特性并不会影响输入控件的 value 特性，因为浏览器在提交表单时并不会包含未被选中的复选框。如果要确保表单中这两个值中的一个能够被提交，(比如“yes”或“no”)，请换用单选按钮。

#### 单选按钮

<input type="radio" v-model="pick" v-bind:value="a">

// 当选中时  
vm.pick === vm.a

#### 下拉框选项

<select v-model="selected">  
 <!-- 内联对象字面量 -->  
 <option v-bind:value="{ number: 123 }">123</option>  
</select>

// 当选中时  
typeof vm.selected // => 'object'  
vm.selected.number // => 123

### 修饰符

#### .lazy

在默认情况下，v-model 在每次 input 事件触发后将输入框的值与数据进行同步 (除了[**上述**](https://cn.vuejs.org/v2/guide/forms.html#vmodel-ime-tip)输入法组合文字时)。你可以添加 lazy 修饰符，从而转变为使用 change事件进行同步：

<!-- 在“change”时而非“input”时更新 -->  
<input v-model.lazy="msg" >

#### .number

如果想自动将用户的输入值转为数值类型，可以给 v-model 添加 number 修饰符

<input v-model.number="age" type="number">

如果这个值无法被 parseFloat() 解析，则会返回原始的值。

#### .trim

如果要自动过滤用户输入的首尾空白字符，可以给 v-model 添加 trim 修饰符

<input v-model.trim="msg">

### 在组件上使用v-model

HTML 原生的输入元素类型并不总能满足需求。幸好，Vue 的组件系统允许你创建具有完全自定义行为且可复用的输入组件。这些输入组件甚至可以和 v-model 一起使用

## 组件基础

### 基本示例

// 定义一个名为 button-counter 的新组件  
Vue.component('button-counter', {  
 data: function () {  
 return {  
 count: 0  
 }  
 },  
 template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'  
})

组件是可复用的 Vue 实例，且带有一个名字

|  |
| --- |
| <div id="components-demo">  <button-counter></button-counter> </div> |

|  |
| --- |
| new Vue({ el: '#components-demo' }) |

通过初始化根实例来吧组件引入

组件是可复用的 Vue 实例，所以它们与 new Vue 接收相同的选项，例如 data、computed、watch、methods 以及生命周期钩子等。仅有的例外是像 el这样根实例特有的选项

### 组件的复用

可以将组件进行任意次数的复用

<div id="components-demo">  
 <button-counter></button-counter>  
 <button-counter></button-counter>  
 <button-counter></button-counter>  
</div>

你每用一次组件，就会有一个它的新**实例**被创建

#### Data必须是一个函数

**一个组件的 data 选项必须是一个函数**，因此每个实例可以维护一份被返回对象的独立的拷贝，没有这条规则，点击一个按钮会影响到其它所有实例

data: function () {  
 return {  
 count: 0  
 }  
}

### 组件的组织

通常一个应用会以一棵嵌套的组件树的形式来组织



为了能在模板中使用，这些组件必须先注册以便 Vue 能够识别。这里有两种组件的注册类型：**全局注册**和**局部注册**。

通过 Vue.component 全局注册的

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

全局注册的组件可以用在其被注册之后的任何 (通过 new Vue) 新创建的 Vue 根实例，也包括其组件树中的所有子组件的模板中。

### 通过prop向子组件传递数据

Prop 是你可以在组件上注册的一些自定义特性。当一个值传递给一个 prop 特性的时候，它就变成了那个组件实例的一个属性。

Vue.component('blog-post', {  
 props: ['title'],  
 template: '<h3>{{ title }}</h3>'  
})

一个组件默认可以拥有任意数量的 prop，任何值都可以传递给任何 prop。

<blog-post title="My journey with Vue"></blog-post>

批量渲染组件时

<blog-post  
 v-for="post in posts"  
 v-bind:key="post.id"  
 v-bind:title="post.title"  
></blog-post>

### 单个根元素

**every component must have a single root element (每个组件必须只有一个根元素)**

Vue.component('blog-post', {  
 props: ['post'],  
 template: `  
 <div class="blog-post">  
 <h3>{{ post.title }}</h3>  
 <div v-html="post.content"></div>  
 </div>  
})

Ps：上述的这个和一些接下来的示例使用了 JavaScript 的**[模板字符串](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Template_literals" \t "_blank)**来让多行的模板更易读。它们在 IE 下并没有被支持，所以如果你需要在不 (经过 Babel 或 TypeScript 之类的工具) 编译的情况下支持 IE，请使用**[折行转义字符](https://css-tricks.com/snippets/javascript/multiline-string-variables-in-javascript/" \t "_blank)**取而代之。

不论何时为 post 对象添加一个新的属性，它都会自动地在 <blog-post> 内可用

### 通过事件向父组件发送消息

调用内建的 [**$emit 方法**](https://cn.vuejs.org/v2/api/#vm-emit)并传入事件的名字，来向父级组件触发一个事件

<button v-on:click="$emit('enlarge-text')">  
 Enlarge text  
</button>

然后用 v-on 在组件上监听事件，就像监听一个原生 DOM 事件一样

<blog-post  
 ...  
 v-on:enlarge-text="postFontSize += 0.1"  
></blog-post>

#### 使用事件抛出一个值

$emit 的第二个参数来提供这个值

<button v-on:click="$emit('enlarge-text', 0.1)">  
 Enlarge text  
</button>

父级组件监听这个事件的时候，我们可以通过 $event 访问到被抛出的这个值

<blog-post  
 ...  
 v-on:enlarge-text="postFontSize += $event"  
></blog-post>

或者，事件处理函数是一个方法：

<blog-post  
 ...  
 v-on:enlarge-text="onEnlargeText"  
></blog-post>

这个值将会作为第一个参数传入这个方法：

methods: {  
 onEnlargeText: function (enlargeAmount) {  
 this.postFontSize += enlargeAmount  
 }  
}

#### 在组件上使用v-model

为了让它正常工作，这个组件内的 <input> 必须：

1，将其 value 特性绑定到一个名叫 value 的 prop 上

2，在其 input 事件被触发时，将新的值通过自定义的 input 事件抛出

Vue.component('custom-input', {  
 props: ['value'],  
 template: `  
 <input  
 v-bind:value="value"  
      v-on:input="$emit('input', $event.target.value)"  
    >  
 `  
})

<custom-input v-model="searchText"></custom-input>

### 通过插槽分发内容

Vue 自定义的 <slot> 元素让这变得非常简单

Vue.component('alert-box', {  
 template: `  
 <div class="demo-alert-box">  
 <strong>Error!</strong>  
 <slot></slot>  
 </div>  
 `  
})

### 动态组件

<!-- 组件会在 `currentTabComponent` 改变时改变 -->  
<component v-bind:is="currentTabComponent"></component>

currentTabComponent 可以包括

1，已注册组件的名字，或,

2，一个组件的选项对象

### 解析DOM模板时的注意事项

有些 HTML 元素，诸如 <ul>、<ol>、<table> 和 <select>，对于哪些元素可以出现在其内部是有严格限制的。而有些元素，诸如 <li>、<tr> 和 <option>，只能出现在其它某些特定的元素内部

Ps：需要注意的是**如果我们从以下来源使用模板的话，这条限制是不存在的**：

1，字符串 (例如：template: '...')

2，[单文件组件 (.vue)](https://cn.vuejs.org/v2/guide/single-file-components.html)

3，[<script type="text/x-template">](https://cn.vuejs.org/v2/guide/components-edge-cases.html#X-Templates)

# 深入了解组件

## 组件注册

### 组件名

在注册一个组件的时候，我们始终需要给它一个名字

Vue.component('my-component-name', { /\* ... \*/ })

Ps：组件名 (字母全小写且必须包含一个连字符)。这会帮助你避免和当前以及未来的 HTML 元素相冲突

#### 组件名大小写

定义组件名的方式有两种

Kebab-case(短横线隔开)

Vue.component('my-component-name', { /\* ... \*/ })

pascalBase(首字母大写命名—驼峰式)

### 全局注册

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

子组件之间也可互相用

### 局部注册

**局部注册的组件在其子组件中不可用**

import ComponentA from './ComponentA.vue'

components: {  
 ComponentA  
}

### 模块系统

#### 在模块系统中局部注册

import ComponentA from './ComponentA'  
import ComponentC from './ComponentC'  
  
export default {  
 components: {  
 ComponentA,  
 ComponentC  
 },  
 // ...  
}

#### 基础组件的自动化全局注册

如果你使用了 webpack (或在内部使用了 webpack 的 **[Vue CLI 3+](https://github.com/vuejs/vue-cli" \t "_blank)**)，那么就可以使用 require.context 只全局注册这些非常通用的基础组件

import Vue from 'vue'  
import upperFirst from 'lodash/upperFirst'  
import camelCase from 'lodash/camelCase'  
  
const requireComponent = require.context(  
 // 其组件目录的相对路径  
 './components',  
 // 是否查询其子目录  
 false,  
 // 匹配基础组件文件名的正则表达式  
 /Base[A-Z]\w+\.(vue|js)$/  
)  
  
requireComponent.keys().forEach(fileName => {  
 // 获取组件配置  
 const componentConfig = requireComponent(fileName)  
  
 // 获取组件的 PascalCase 命名  
 const componentName = upperFirst(  
 camelCase(  
 // 剥去文件名开头的 `./` 和结尾的扩展名  
 fileName.replace(/^\.\/(.\*)\.\w+$/, '$1')  
 )  
 )  
  
 // 全局注册组件  
 Vue.component(  
 componentName,  
 // 如果这个组件选项是通过 `export default` 导出的，  
 // 那么就会优先使用 `.default`，  
 // 否则回退到使用模块的根。  
 componentConfig.default || componentConfig  
 )  
})

Ps: **全局注册的行为必须在根 Vue 实例 (通过 new Vue) 创建之前发生**

## Prop

### Prop的大小写(camelCase vs kebab-case)

HTML 中的特性名是大小写不敏感的，所以浏览器会把所有大写字符解释为小写字符。这意味着当你使用 DOM 中的模板时，camelCase (驼峰命名法) 的 prop 名需要使用其等价的 kebab-case (短横线分隔命名) 命名

Vue.component('blog-post', {  
 // 在 JavaScript 中是 camelCase 的  
 props: ['postTitle'],  
 template: '<h3>{{ postTitle }}</h3>'  
})

<!-- 在 HTML 中是 kebab-case 的 -->  
<blog-post post-title="hello!"></blog-post>

### Prop类型

通常你希望每个 prop 都有指定的值类型。这时，你可以以对象形式列出 prop，这些属性的名称和值分别是 prop 各自的名称和类型

props: {  
 title: String,  
 likes: Number,  
 isPublished: Boolean,  
 commentIds: Array,  
 author: Object  
}

props: ['title', 'likes', 'isPublished', 'commentIds', 'author']

也可简写

### 传递静态或动态Prop

<!-- 动态赋予一个变量的值 -->  
<blog-post v-bind:title="post.title"></blog-post>  
<!-- 动态赋予一个复杂表达式的值 -->  
<blog-post v-bind:title="post.title + ' by ' + post.author.name"></blog-post>

#### 传入一个数字

<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-post v-bind:likes="42"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-post v-bind:likes="post.likes"></blog-post>

#### 传入一个布尔值

<!-- 包含该 prop 没有值的情况在内，都意味着 `true`。-->  
<blog-post is-published></blog-post>  
<!-- 即便 `false` 是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->  
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-post v-bind:is-published="false"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-post v-bind:is-published="post.isPublished"></blog-post>

#### 传入一个数组

<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-post v-bind:comment-ids="post.commentIds"></blog-post>

#### 传入一个对象

<!-- 即便对象是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->  
<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->  
<blog-post v-bind:author="{ name: 'Veronica', company: 'Veridian Dynamics' }"></blog-post>  
<!-- 用一个变量进行动态赋值。-->  
<blog-post v-bind:author="post.author"></blog-post>

#### 传入一个对象的所有属性

如果你想要将一个对象的所有属性都作为 prop 传入，你可以使用不带参数的 v-bind(取代 v-bind:prop-name)。例如，对于一个给定的对象 post

post: {  
 id: 1,  
 title: 'My Journey with Vue'  
}

<blog-post v-bind="post"></blog-post>

等价于：

|  |
| --- |
| <blog-post  v-bind:id="post.id"  v-bind:title="post.title" ></blog-post> |

### 单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个**单向下行绑定**：每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你**不**应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告

这个 prop 用来传递一个初始值；这个子组件接下来希望将其作为一个本地的 prop 数据来使用。在这种情况下，最好定义一个本地的 data 属性并将这个 prop 用作其初始值：

|  |
| --- |
| props: ['initialCounter'], data: function () {  return {  counter: this.initialCounter  } } |

这个 prop 以一种原始的值传入且需要进行转换。在这种情况下，最好使用这个 prop 的值来定义一个计算属性：

|  |
| --- |
| props: ['size'], computed: {  normalizedSize: function () {  return this.size.trim().toLowerCase()  } } |

Ps：注意在 JavaScript 中对象和数组是通过引用传入的，所以对于一个数组或对象类型的 prop 来说，在子组件中改变这个对象或数组本身**将会**影响到父组件的状态。

### Prop验证

Vue.component('my-component', {  
 props: {  
 // 基础的类型检查 (`null` 匹配任何类型)  
 propA: Number,  
 // 多个可能的类型  
 propB: [String, Number],  
 // 必填的字符串  
 propC: {  
 type: String,  
 required: true  
 },  
 // 带有默认值的数字  
 propD: {  
 type: Number,  
 default: 100  
 },  
 // 带有默认值的对象  
 propE: {  
 type: Object,  
 // 对象或数组默认值必须从一个工厂函数获取  
 default: function () {  
 return { message: 'hello' }  
 }  
 },  
 // 自定义验证函数  
 propF: {  
 validator: function (value) {  
 // 这个值必须匹配下列字符串中的一个  
 return ['success', 'warning', 'danger'].indexOf(value) !== -1  
 }  
 }  
 }  
})

当 prop 验证失败的时候，(开发环境构建版本的) Vue 将会产生一个控制台的警告。

Ps：注意那些 prop 会在一个组件实例创建**之前**进行验证，所以实例的属性 (如 data、computed 等) 在 default 或 validator 函数中是不可用的

#### 类型检查

String，Number，Boolean，Array，Object，Date，Function，Symbol

通过 instanceof 来进行检查

例如，给定下列现成的构造函数：

function Person (firstName, lastName) {  
 this.firstName = firstName  
 this.lastName = lastName  
}

props: {  
 author: Person  
}

来验证 author prop 的值是否是通过 new Person 创建的

### 非Prop特性

非 prop 特性是指传向一个组件，但是该组件并没有相应 prop 定义的特性

例如

<bootstrap-date-input data-date-picker="activated"></bootstrap-date-input>

然后这个 data-date-picker="activated" 特性就会自动添加到 <bootstrap-date-input> 的根元素上

#### 替换/合并已有的特性

从外部提供给组件的值会替换掉组件内部设置好的值。例如传入的input的type的值是可变的，但传入的class以及style的值是会合并的

#### 禁用特性继承

如果你**不**希望组件的根元素继承特性，你可以在组件的选项中设置

inheritAttrs: false

Vue.component('my-component', {  
 inheritAttrs: false,  
 // ...  
})

有了 inheritAttrs: false 和 $attrs，你就可以手动决定这些特性会被赋予哪个元素。在撰写[基础组件](https://cn.vuejs.org/v2/style-guide/" \l "%E5%9F%BA%E7%A1%80%E7%BB%84%E4%BB%B6%E5%90%8D-%E5%BC%BA%E7%83%88%E6%8E%A8%E8%8D%90)的时候是常会用到的：

|  |
| --- |
| Vue.component('base-input', {  inheritAttrs: false,  props: ['label', 'value'],  template: `  <label>  {{ label }}  <input  v-bind="$attrs"  v-bind:value="value"  v-on:input="$emit('input', $event.target.value)"  >  </label>  ` }) |

这个模式允许你在使用基础组件的时候更像是使用原始的 HTML 元素，而不会担心哪个元素是真正的根元素：

|  |
| --- |
| <base-input  v-model="username"  class="username-input"  placeholder="Enter your username" ></base-input> |

## 自定义事件

### 事件名

触发的事件名需要完全匹配监听这个事件所用的名称

事件名不会被用作一个 JavaScript 变量名或属性名，所以使用 kebab-case 的事件名

### 自定义组件的v-model

2.2.0新增

一个组件上的 v-model 默认会利用名为 value 的 prop 和名为 input 的事件，但是像单选框、复选框等类型的输入控件可能会将 value 特性用于[**不同的目的**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox#Value)。model 选项可以用来避免这样的冲突

template: `  
 <input  
 type="checkbox"  
 v-bind:checked="checked"  
 v-on:change="$emit('change', $event.target.checked)"  
 >  
 `

<base-checkbox v-model="lovingVue"></base-checkbox>

这里的 lovingVue 的值将会传入这个名为 checked 的 prop。同时当 <base-checkbox> 触发一个 change 事件并附带一个新的值的时候，这个 lovingVue 的属性将会被更新

Ps：注意你仍然需要在组件的 props 选项里声明 checked 这个 prop。

### 将原生事件绑定到组件

Vue 提供了一个 $listeners 属性，它是一个对象，里面包含了作用在这个组件上的所有监听器

computed: {  
 inputListeners: function () {  
 var vm = this  
 // `Object.assign` 将所有的对象合并为一个新对象  
 return Object.assign({},  
 // 我们从父级添加所有的监听器  
 this.$listeners,  
 // 然后我们添加自定义监听器，  
 // 或覆写一些监听器的行为  
 {  
 // 这里确保组件配合 `v-model` 的工作  
 input: function (event) {  
 vm.$emit('input', event.target.value)  
 }  
 }  
 )  
 }  
 },

所有跟它相同的特性和监听器的都可以工作

### .sync修饰符

2.3.0新增

<text-document v-bind:title.sync="doc.title"></text-document>

Ps：注意带有 .sync 修饰符的 v-bind **不能**和表达式一起使用 (例如 v-bind:title.sync=”doc.title + ‘!’” 是无效的)。取而代之的是，你只能提供你想要绑定的属性名，类似 v-model

可以设置多个

<text-document v-bind.sync="doc"></text-document>

PS：将 v-bind.sync 用在一个字面量的对象上，例如 v-bind.sync=”{ title: doc.title }”，是无法正常工作的，因为在解析一个像这样的复杂表达式的时候，有很多边缘情况需要考虑。

## 插槽

### 插槽内容

<slot> 元素作为承载分发内容的出口

<navigation-link url="/profile">  
 Your Profile  
</navigation-link>

<a  
 v-bind:href="url"  
 class="nav-link"  
>  
 <slot></slot>  
</a>

如果组件**没有**包含一个 <slot> 元素，则任何传入它的内容都会被抛弃

### 具名插槽

当需要多个插槽时，这样的情况，<slot> 元素有一个特殊的特性：name

<div class="container">  
 <header>  
 <slot name="header"></slot>  
 </header>  
 <main>  
 <slot></slot>  
 </main>  
 <footer>  
 <slot name="footer"></slot>  
 </footer>  
</div>

<template slot="header">  
 <h1>Here might be a page title</h1>  
 </template>

或者

<h1 slot="header">Here might be a page title</h1>

可以保留一个未命名插槽，这个插槽是**默认插槽**，也就是说它会作为所有未匹配到插槽的内容的统一出口

### 插槽的默认内容

可以在组件模板里的 <slot> 标签内部指定默认的内容来做到这一点

<slot>Submit</slot>

父组件为这个插槽提供了内容，则默认的内容会被替换掉

### 编译作用域

**父组件模板的所有东西都会在父级作用域内编译；子组件模板的所有东西都会在子级作用域内编译**

在插槽内使用数据时

<navigation-link url="/profile">  
 Logged in as {{ user.name }}  
</navigation-link>

请按照上述规则编写

### 作用域插槽

2.1.0新增

当需要在插槽内使用数据

<li  
 v-for="todo in todos"  
 v-bind:key="todo.id"  
 >  
 <!-- 我们为每个 todo 准备了一个插槽，-->  
 <!-- 将 `todo` 对象作为一个插槽的 prop 传入。-->  
 <slot v-bind:todo="todo">  
 <!-- 回退的内容 -->  
 {{ todo.text }}  
 </slot>  
 </li>

可以选择为待办项定义一个不一样的 <template> 作为替代方案，并且可以通过 slot-scope 特性从子组件获取数据

<todo-list v-bind:todos="todos">  
 <!-- 将 `slotProps` 定义为插槽作用域的名字 -->  
 <template slot-scope="slotProps">  
 <!-- 为待办项自定义一个模板，-->  
 <!-- 通过 `slotProps` 定制每个待办项。-->  
 <span v-if="slotProps.todo.isComplete">✓</span>  
 {{ slotProps.todo.text }}  
 </template>  
</todo-list>

PS：**在 2.5.0+，slot-scope 不再限制在 <template> 元素上使用，而可以用在插槽内的任何元素或组件上。**

#### 解构slot-scope

如果一个 JavaScript 表达式在一个函数定义的参数位置有效，那么这个表达式实际上就可以被 slot-scope 接受。也就是说你可以在支持的环境下 ([**单文件组件**](https://cn.vuejs.org/v2/guide/single-file-components.html)或[**现代浏览器**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#%E6%B5%8F%E8%A7%88%E5%99%A8%E5%85%BC%E5%AE%B9))，在这些表达式中使用 [**ES2015 解构语法**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#%E8%A7%A3%E6%9E%84%E5%AF%B9%E8%B1%A1)

<todo-list v-bind:todos="todos">  
 <template slot-scope="{ todo }">  
 <span v-if="todo.isComplete">✓</span>  
 {{ todo.text }}  
 </template>  
</todo-list>

PS：**推荐使用**

## 动态组件&异步组件