Vue.js

组件和for组件总结完了回头看

VUE MVVM 模式 绑定数据后 操作数据进行 结构的重新渲染 不刷新页面

**钩子函数**

对于自定义指令的定义，Vue2 有 5 个可选的钩子函数。

* **bind** : 只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。
* **inserted** : 被绑定元素插入父节点时调用（父节点存在即可调用，不必存在于 document 中）。
* **update** : 被绑定元素所在的模板更新时调用，而不论绑定值是否变化。
* **componentUpdated** : 被绑定元素所在模板完成一次更新周期时调用。
* **unbind** : 只调用一次，指令与元素解绑时调用。

v-model 指令在表单控件元素上创建双向数据绑定,所谓双向绑定，指的就是我们在js中的vue实例中的data与其渲染的dom元素上的内容保持一致，两者无论谁被改变，另一方也会相应的更新为相同的数据

最基础的就是实现一个联动的效果

v-bind是可以绑定属性和数据的，而v-model也一样可以绑定数据，那么v-bind和v-model的区别在哪里？

1：如果  id={{xxx}}这种写法，则HTML实际上是这样的：id="{{ xxx}}"

2:v-bind用于绑定属性和数据 ，其缩写为“ : ” 也就是v-bind:id  ===   :id

3:v-model也可以绑定数据，但是他是用在表单控件上的，用于实现双向数据绑定，所以如果你用在除了表单控件以外的标签是没有任何效果的

组件架构：

var app5 = new Vue({

el:'', //元素 选择器

data:{}, //数据

methods:{// 事件监听

reverseMessage:function()// methods 事件绑定函数内容

},

created:function(){`this指向vm 实例console.log(1)}, //created 这钩子在实例被创建之后被调用 其他时期的钩子 mounted、 updated 、destroyed

computed: {//计算属性

reversedMessage: function () {

return this.message.split('').reverse().join('')//操作结果返回

}

}，

watch: {}//相当于onchange

})

Component 声明一个新组建

Vue.component('todo-item', { //创建一个新的可共用的组件

props: ['todo'],

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

})

watch 是监听数据变化的 方法 数据变了 就重新渲染结构

v-model 绑定数据<input type="text" name="li" v-model="firstName">

v-bind 指令用于响应地更新 HTML 特性 形式如：v-bind:href 缩写为 :href;

v-on 指令用于监听DOM事件 形式如：v-on:click 缩写为 @click;

v-bind 属性被称为指令 这里该指令的作用是：“将这个元素节点的 title 属性和 Vue 实例的 message 属性保持一致”

v-for="t in todos" 循环指令 t是自定义的 for in 的 var s t ....... 绑定到循环元素上

app4.todos.push({ text: '新项目' })

v-on:click="reverseMessage" 事件绑定不带on 后面是事件名v-on:click 缩写为 @click;

v-model 指令，使表单输入和应用状态间的双向绑定变得轻而易举

数据绑定最常见的形式就是使用 “Mustache” 语法（双大括号）的文本插值：

|  |
| --- |
| <span>Message: {{ msg }}</span> |
| 通过使用 [v-once 指令](https://cn.vuejs.org/v2/api/#v-once)，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。  但请留心这会影响到该节点上所有的数据绑定：   |  | | --- | | <span v-once>This will never change: {{ msg }}</span> | |

双大括号会将数据解释为纯文本，而非 HTML 。为了输出真正的 HTML ，你需要使用 v-html 指令：

|  |
| --- |
| <div v-html="rawHtml"></div> |

被插入的内容都会被当做 HTML —— 数据绑定会被忽略。注意，你不能使用 v-html 来复合局部模板，因为 Vue 不是基于字符串的模板引擎。组件更适合担任 UI 重用与复合的基本单元。

Mustache 不能在 HTML 属性中使用，应使用 [v-bind 指令](https://cn.vuejs.org/v2/api/#v-bind)：

|  |
| --- |
| <p v-if="seen">Now you see me</p> |

这里， v-if 指令将根据表达式 seen 的值的真假来移除/插入 <p> 元素。

### [参数](https://cn.vuejs.org/v2/guide/syntax.html" \l "参数)

一些指令能接受一个“参数”，在指令后以冒号指明。例如， v-bind 指令被用来响应地更新 HTML 属性：

|  |
| --- |
| <a v-bind:href="url"></a> |

在这里 href 是参数，告知 v-bind 指令将该元素的 href 属性与表达式 url 的值绑定。

另一个例子是 v-on 指令，它用于监听 DOM 事件：

|  |
| --- |
| <a v-on:click="doSomething"> |

在这里参数是监听的事件名。我们也会更详细地讨论事件处理。

### [修饰符](https://cn.vuejs.org/v2/guide/syntax.html" \l "修饰符)

修饰符（Modifiers）是以半角句号 . 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。例如，.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()：

|  |
| --- |
| <form v-on:submit.prevent="onSubmit"></form> |

之后当我们更深入地了解 v-on 与 v-model时，会看到更多修饰符的使用。

过滤器 需要仔细看看 在整理、

[v-bind 缩写](https://cn.vuejs.org/v2/guide/syntax.html" \l "v-bind-缩写)

|  |
| --- |
| <!-- 完整语法 -->  <a v-bind:href="url"></a>  <!-- 缩写 -->  <a :href="url"></a> |

### [v-on 缩写](https://cn.vuejs.org/v2/guide/syntax.html" \l "v-on-缩写)

|  |
| --- |
| <!-- 完整语法 -->  <a v-on:click="doSomething"></a>  <!-- 缩写 -->  <a @click="doSomething"></a> |

## [计算属性 computed：{}](https://cn.vuejs.org/v2/guide/computed.html" \l "计算属性)

模板内的表达式是非常便利的，但是它们实际上只用于简单的运算。在模板中放入太多的逻辑会让模板过重且难以维护。例如：

<div id="example">{{ message.split('').reverse().join('') }}</div>

### [计算缓存 与 Methods](https://cn.vuejs.org/v2/guide/computed.html" \l "计算缓存-vs-Methods)

你可能已经注意到我们可以通过调用表达式中的 method 来达到同样的效果：

methods: {

reversedMessage: function () {

return this.message.split('').reverse().join('')

}

}

我们可以将同一函数定义为一个 method 而不是一个计算属性。对于最终的结果，两种方式确实是相同的。然而，不同的是**计算属性是基于它们的依赖进行缓存的**。计算属性只有在它的相关依赖发生改变时才会重新求值。这就意味着只要message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。

### [Computed 属性 vs Watched 属性](https://cn.vuejs.org/v2/guide/computed.html" \l "Computed-属性-vs-Watched-属性)

Vue 确实提供了一种更通用的方式来观察和响应 Vue 实例上的数据变动：watch 属性。当你有一些数据需要随着其它数据变动而变动时，你很容易滥用 watch——特别是如果你之前使用过 AngularJS。然而，通常更好的想法是使用 computed 属性而不是命令式的 watch 回调。细想一下这个例子：

计算属性默认只有 getter ，不过在需要时你也可以提供一个 setter ：

|  |
| --- |
| // ...  computed: {  fullName: {  // getter  get: function () {  return this.firstName + ' ' + this.lastName  },  // setter  set: function (newValue) {  var names = newValue.split(' ')  this.firstName = names[0]  this.lastName = names[names.length - 1]  }  }  }  // ... |

现在在运行 vm.fullName = 'John Doe' 时， setter 会被调用， vm.firstName 和 vm.lastName 也相应地会被更新。

## [观察 Watchers](https://cn.vuejs.org/v2/guide/computed.html" \l "观察-Watchers)

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的 watcher 。这是为什么 Vue 提供一个更通用的方法通过watch 选项，来响应数据的变化。当你想要在数据变化响应时，执行异步操作或开销较大的操作，这是很有用的。

## [绑定 HTML Class](https://cn.vuejs.org/v2/guide/class-and-style.html" \l "绑定-HTML-Class)

https://cn.vuejs.org/v2/guide/class-and-style.html#对象语法

我们可以传给 v-bind:class 一个对象，以动态地切换 class 。

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

上面的语法表示 classactive 的更新将取决于数据属性 isActive 是否为[真值](https://developer.mozilla.org/en-US/docs/Glossary/Truthy" \t "_blank) 。

我们也可以在对象中传入更多属性用来动态切换多个 class 。此外， v-bind:class 指令可以与普通的 class 属性共存。如下模板:

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

如下 data:

|  |
| --- |
| data: {isActive: true, hasError: false} |

渲染为:

|  |
| --- |
| <div class="static active"></div> |

当 isActive 或者 hasError 变化时，class 列表将相应地更新。例如，如果 hasError 的值为 true ， class列表将变为 "static active text-danger"。

你也可以直接绑定数据里的一个对象：

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

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

渲染的结果和上面一样。我们也可以在这里绑定返回对象的[计算属性](https://cn.vuejs.org/v2/guide/computed.html)。这是一个常用且强大的模式：

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

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

### 数组语法

我们可以把一个数组传给 v-bind:class ，以应用一个 class 列表：

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

|  |
| --- |
| data: {  activeClass: 'active',  errorClass: 'text-danger'  } |

渲染为:

|  |
| --- |
| <div class="active text-danger"></div> |

如果你也想根据条件切换列表中的 class ，可以用三元表达式：

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

此例始终添加 errorClass ，但是只有在 isActive 是 true 时添加 activeClass 。

不过，当有多个条件 class 时这样写有些繁琐。可以在数组语法中使用对象语法：

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

### 用在组件上

**这个章节假设你已经对**[Vue 组件](https://cn.vuejs.org/v2/guide/components.html)**有一定的了解。当然你也可以跳过这里，稍后再回过头来看。**

当你在一个定制的组件上用到 class 属性的时候，这些类将被添加到根元素上面，这个元素上已经存在的类不会被覆盖。

例如，如果你声明了这个组件:

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

然后在使用它的时候添加一些 class：

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

HTML 最终将被渲染成为:

|  |
| --- |
| <p class="foo bar baz boo">Hi</p> |

同样的适用于绑定 HTML class :

|  |
| --- |
| <my-component v-bind:class="{ active: isActive }"></my-component> |

当 isActive 为 true 的时候，HTML 将被渲染成为:

|  |
| --- |
| <p class="foo bar active">Hi</p> |

## 绑定内联样式

### 对象语法

v-bind:style 的对象语法十分直观——看着非常像 CSS ，其实它是一个 JavaScript 对象。 CSS 属性名可以用驼峰式（camelCase）或短横分隔命名（kebab-case）：

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

|  |
| --- |
| data: {  activeColor: 'red',  fontSize: 30  } |

直接绑定到一个样式对象通常更好，让模板更清晰：

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

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

同样的，对象语法常常结合返回对象的计算属性使用。

### 数组语法

v-bind:style 的数组语法可以将多个样式对象应用到一个元素上：

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

### 自动添加前缀

当 v-bind:style 使用需要特定前缀的 CSS 属性时，如 transform ，Vue.js 会自动侦测并添加相应的前缀。

# 条件渲染

## v-if

在字符串模板中，如 Handlebars ，我们得像这样写一个条件块：

|  |
| --- |
| <!-- Handlebars 模板 -->  {{#if ok}}  <h1>Yes</h1>  {{/if}} |

在 Vue.js ，我们使用 v-if 指令实现同样的功能：

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

也可以用 v-else 添加一个 “else” 块：

|  |
| --- |
| <h1 v-if="ok">Yes</h1>  <h1 v-else>No</h1> |

### <template> 中 v-if 条件组

因为 v-if 是一个指令，需要将它添加到一个元素上。但是如果我们想切换多个元素呢？此时我们可以把一个<template> 元素当做包装元素，并在上面使用 v-if。最终的渲染结果不会包含 <template> 元素。

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

### v-else

你可以使用 v-else 指令来表示 v-if 的“else 块”：

|  |
| --- |
| <div v-if="Math.random() > 0.5">  Now you see me  </div>  <div v-else>  Now you don't  </div> |

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

### v-else-if

**2.1.0 新增**

v-else-if，顾名思义，充当 v-if 的“else-if 块”。可以链式地使用多次：

|  |
| --- |
| <div v-if="type === 'A'">  A  </div>  <div v-else-if="type === 'B'">  B  </div>  <div v-else-if="type === 'C'">  C  </div>  <div v-else>  Not A/B/C  </div> |

类似于 v-else,，v-else-if 必须紧跟在 v-if 或者 v-else-if 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，还有一些有用的好处。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> |

那么在上面的代码中切换 loginType 将不会清除用户已经输入的内容。因为两个模版使用了相同的元素，<input> 不会被替换掉——仅仅是替换了它的的 placeholder。

自己动手试一试，在输入框中输入一些文本，然后按下切换按钮：

Username 

Toggle login type

这样也不总是符合实际需求，所以 Vue 为你提供了一种方式来声明“这两个元素是完全独立的——不要复用它们”。只需添加一个具有唯一值的 key 属性即可：

|  |
| --- |
| <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> |

现在，每次切换时，输入框都将被重新渲染。你自己看：

Username 

Toggle login type

注意, <label> 元素仍然会被高效地复用，因为它们没有添加 key 属性。

## v-show

另一个用于根据条件展示元素的选项是 v-show 指令。用法大致一样：

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

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

注意， v-show 不支持 <template> 语法，也不支持 v-else。

## v-if vs 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。

v-if 也是**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。

相比之下， 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。

一般来说， 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；如果在运行时条件不太可能改变，则使用 v-if 较好。

## v-if 与 v-for 一起使用

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。

## [v-for](https://cn.vuejs.org/v2/guide/list.html" \l "v-for)

我们用 v-for 指令根据一组数组的选项列表进行渲染。 v-for 指令需要以 item in items 形式的特殊语法， items是源数据数组并且 item 是数组元素迭代的别名。

### 基本用法

|  |
| --- |
| <ul id="example-1">  <li v-for="item in items">  {{ item.message }}  </li>  </ul> |

|  |
| --- |
| var example1 = new Vue({  el: '#example-1',  data: {  items: [  {message: 'Foo' },  {message: 'Bar' }  ]  }  }) |

结果：

* Foo
* Bar

在 v-for 块中，我们拥有对父作用域属性的完全访问权限。 v-for 还支持一个可选的第二个参数为当前项的索引。

|  |
| --- |
| <ul id="example-2">  <li v-for="(item, index) in items">  {{ parentMessage }} - {{ index }} - {{ item.message }}  </li>  </ul> |

|  |
| --- |
| var example2 = new Vue({  el: '#example-2',  data: {  parentMessage: 'Parent',  items: [  { message: 'Foo' },  { message: 'Bar' }  ]  }  }) |

结果：

* Parent - 0 - Foo
* Parent - 1 - Bar

你也可以用 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> |

### Template v-for

如同 v-if 模板，你也可以用带有 v-for 的 <template> 标签来渲染多个元素块。例如：

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

### 对象迭代 v-for

你也可以用 v-for 通过一个对象的属性来迭代。

|  |
| --- |
| <ul id="repeat-object" class="demo">  <li v-for="value in object">  {{ value }}  </li>  </ul> |

|  |
| --- |
| new Vue({  el: '#repeat-object',  data: {  object: {  FirstName: 'John',  LastName: 'Doe',  Age: 30  }  }  }) |

结果：

* John
* Doe
* 30

你也可以提供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> |

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。

### 整数迭代 v-for

v-for 也可以取整数。在这种情况下，它将重复多次模板。

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

结果：

1 2 3 4 5 6 7 8 9 10

### 组件 和 v-for

**了解组件相关知识，查看**[组件](https://cn.vuejs.org/v2/guide/components.html)**。完全可以先跳过它，以后再回来查看。**

在自定义组件里，你可以像任何普通元素一样用 v-for 。

|  |
| --- |
| <my-component v-for="item in items"></my-component> |

然而他不能自动传递数据到组件里，因为组件有自己独立的作用域。为了传递迭代数据到组件里，我们要用 props ：

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

不自动注入 item 到组件里的原因是，因为这使得组件会紧密耦合到 v-for 如何运作。在一些情况下，明确数据的来源可以使组件可重用。

下面是一个简单的 todo list 完整的例子：

|  |
| --- |
| <div id="todo-list-example">  <input  v-model="newTodoText"  v-on:keyup.enter="addNewTodo"  placeholder="Add a todo"  >  <ul>  <li  is="todo-item"  v-for="(todo, index) in todos"  v-bind:title="todo"  v-on:remove="todos.splice(index, 1)"  ></li>  </ul>  </div> |

|  |
| --- |
| Vue.component('todo-item', {  template: '\  <li>\  {{ title }}\  <button v-on:click="$emit(\'remove\')">X</button>\  </li>\  ',  props: ['title']  })  new Vue({  el: '#todo-list-example',  data: {  newTodoText: '',  todos: [  'Do the dishes',  'Take out the trash',  'Mow the lawn'  ]  },  methods: {  addNewTodo: function () {  this.todos.push(this.newTodoText)  this.newTodoText = ''  }  }  }) |



* Do the dishes X
* Take out the trash X
* Mow the lawn X

## 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将不是移动 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。这个类似 Vue 1.x 的 track-by="$index" 。

这个默认的模式是有效的，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。

为了给 Vue 一个提示，以便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 值是每项都有唯一 id。这个特殊的属性相当于 Vue 1.x 的 track-by ，但它的工作方式类似于一个属性，所以你需要用 v-bind 来绑定动态值（在这里使用简写）：

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

建议尽可能使用 v-for 来提供 key ，除非迭代 DOM 内容足够简单，或者你是故意要依赖于默认行为来获得性能提升。

因为它是 Vue 识别节点的一个通用机制， key 并不特别与 v-for 关联，key 还具有其他用途，我们将在后面的指南中看到其他用途。

## 数组更新检测

### 变异方法

Vue 包含一组观察数组的变异方法，所以它们也将会触发视图更新。这些方法如下：

* push()
* pop()
* shift()
* unshift()
* splice()
* sort()
* reverse()

你打开控制台，然后用前面例子的 items 数组调用变异方法：example1.items.push({ message: 'Baz' }) 。

### 重塑数组

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(), 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/)  }) |

你可能认为这将导致 Vue 丢弃现有 DOM 并重新渲染整个列表。幸运的是，事实并非如此。 Vue 实现了一些智能启发式方法来最大化 DOM 元素重用，所以用一个含有相同元素的数组去替换原来的数组是非常高效的操作。

### 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

为了解决第一类问题，以下两种方式都可以实现和 vm.items[indexOfItem] = newValue 相同的效果， 同时也将触发状态更新：

|  |
| --- |
| // Vue.set  Vue.set(example1.items, indexOfItem, newValue) |

|  |
| --- |
| // Array.prototype.splice`  example1.items.splice(indexOfItem, 1, newValue) |

为了解决第二类问题，你也同样可以使用 splice：

|  |
| --- |
| example1.items.splice(newLength) |

## 显示过滤/排序结果

有时，我们想要显示一个数组的过滤或排序副本，而不实际改变或重置原始数据。在这种情况下，可以创建返回过滤或排序数组的计算属性。

例如：

|  |
| --- |
| <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  })  }  } |

或者，你也可以在计算属性不适用的情况下 (例如，在嵌套 v-for 循环中) 使用 method 方法：

|  |
| --- |
| <li v-for="n in even(numbers)">{{ n }}</li> |

|  |
| --- |
| data: {  numbers: [ 1, 2, 3, 4, 5 ]  },  methods: {  even: function (numbers) {  return numbers.filter(function (number) {  return number % 2 === 0  })  }  } |

# 事件处理器

## 监听事件

可以用 v-on 指令监听 DOM 事件来触发一些 JavaScript 代码。

示例：

|  |
| --- |
| <div id="example-1">  <button v-on:click="counter += 1">增加 1</button>  <p>这个按钮被点击了 {{ counter }} 次。</p>  </div> |

|  |
| --- |
| var example1 = new Vue({  el: '#example-1',  data: {  counter: 0  }  }) |

结果：

增加 1

这个按钮被点击了 0 次。

## 方法事件处理器

许多事件处理的逻辑都很复杂，所以直接把 JavaScript 代码写在 v-on 指令中是不可行的。因此 v-on 可以接收一个定义的方法来调用。

示例：

|  |
| --- |
| <div id="example-2">  <!-- `greet` 是在下面定义的方法名 -->  <button v-on:click="greet">Greet</button>  </div> |

|  |
| --- |
| var example2 = new Vue({  el: '#example-2',  data: {  name: 'Vue.js'  },  // 在 `methods` 对象中定义方法  methods: {  greet: function (event) {  // `this` 在方法里指当前 Vue 实例  alert('Hello ' + this.name + '!')  // `event` 是原生 DOM 事件  alert(event.target.tagName)  }  }  })  // 也可以用 JavaScript 直接调用方法  example2.greet() // -> 'Hello Vue.js!' |

结果：

Greet

## 内联处理器方法

除了直接绑定到一个方法，也可以用内联 JavaScript 语句：

|  |
| --- |
| <div id="example-3">  <button v-on:click="say('hi')">Say hi</button>  <button v-on:click="say('what')">Say what</button>  </div> |

|  |
| --- |
| new Vue({  el: '#example-3',  methods: {  say: function (message) {  alert(message)  }  }  }) |

结果：

Say hi Say what

有时也需要在内联语句处理器中访问原生 DOM 事件。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

在事件处理程序中调用 event.preventDefault() 或 event.stopPropagation() 是非常常见的需求。尽管我们可以在 methods 中轻松实现这点，但更好的方式是：methods 只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

为了解决这个问题， Vue.js 为 v-on 提供了 **事件修饰符**。通过由点(.)表示的指令后缀来调用修饰符。

* .stop
* .prevent
* .capture
* .self
* .once

|  |
| --- |
| <!-- 阻止单击事件冒泡 -->  <a v-on:click.stop="doThis"></a>  <!-- 提交事件不再重载页面 -->  <form v-on:submit.prevent="onSubmit"></form>  <!-- 修饰符可以串联 -->  <a v-on:click.stop.prevent="doThat"></a>  <!-- 只有修饰符 -->  <form v-on:submit.prevent></form>  <!-- 添加事件侦听器时使用事件捕获模式 -->  <div v-on:click.capture="doThis">...</div>  <!-- 只当事件在该元素本身（而不是子元素）触发时触发回调 -->  <div v-on:click.self="doThat">...</div> |

**2.1.4 新增**

|  |
| --- |
| <!-- 点击事件将只会触发一次 -->  <a v-on:click.once="doThis"></a> |

不像其它只能对原生的 DOM 事件起作用的修饰符，.once 修饰符还能被用到自定义的[组件事件](https://cn.vuejs.org/v2/guide/components.html" \l "Using-v-on-with-Custom-Events)上. 如果你还没有阅读关于组件的文档，现在大可不必担心。

## 按键修饰符

在监听键盘事件时，我们经常需要监测常见的键值。 Vue 允许为 v-on 在监听键盘事件时添加按键修饰符：

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

记住所有的 keyCode 比较困难，所以 Vue 为最常用的按键提供了别名：

|  |
| --- |
| <!-- 同上 -->  <input v-on:keyup.enter="submit">  <!-- 缩写语法 -->  <input @keyup.enter="submit"> |

全部的按键别名：

* .enter
* .tab
* .delete (捕获 “删除” 和 “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 |

## 按键修饰符

**2.1.0 新增**

可以用如下修饰符开启鼠标或键盘事件监听，使在按键按下时发生响应。

* .ctrl
* .alt
* .shift
* .meta

**注意：在Mac系统键盘上，meta对应命令键 (⌘)。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键盘及其后续，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”。  
例如:**

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

## 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）传统理念。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。你无须担心如何自己清理它们。