1、使用 Object.freeze()，这会阻止修改现有的属性，也意味着响应系统无法再追踪变化。

var obj = {

foo: 'bar'

}

Object.freeze(obj)

new Vue({

el: '#app',

data: obj

})

2、Vue 实例还暴露了一些有用的实例属性与方法。它们都有前缀 $，以便与用户定义的属性区分开来。例如：

var data = { a: 1 }var vm = new Vue({

el: '#example',

data: data

})

vm.$data === data // => true

vm.$el === document.getElementById('example') // => true

**// $watch 是一个实例方法**

vm.$watch('a', function (newValue, oldValue) {

// 这个回调将在 `vm.a` 改变后调用

})

3、当这些数据改变时，视图会进行重渲染。值得注意的是只有当实例被创建时就已经存在于 data 中的属性才是****响应式****的。也就是说如果你添加一个新的属性，比如：

vm.b = 'hi'

那么对 b 的改动将不会触发任何视图的更新。如果你知道你会在晚些时候需要一个属性，但是一开始它为空或不存在，那么你仅需要设置一些初始值。

## 4、**[实例生命周期钩子](https://vuejs.bootcss.com/guide/instance.html" \l "%E5%AE%9E%E4%BE%8B%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E9%92%A9%E5%AD%90" \o "实例生命周期钩子)**

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做****生命周期钩子****的函数，这给了用户在不同阶段添加自己的代码的机会。

比如 **[created](https://vuejs.bootcss.com/api/" \l "created)** 钩子可以用来在一个实例被创建之后执行代码：

new Vue({

data: {

a: 1

},

created: function () {

// `this` 指向 vm 实例

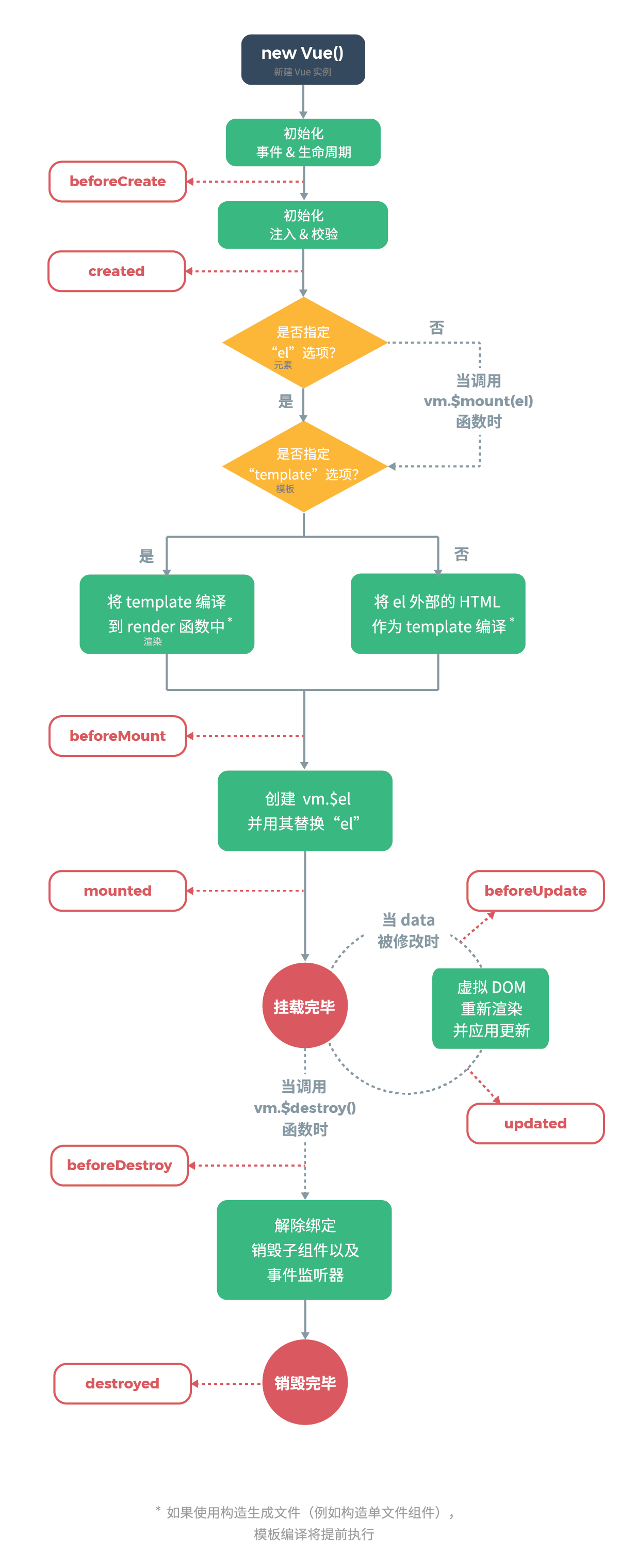
console.log('a is: ' + this.a)

}

})// => "a is: 1"

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 **[mounted](https://vuejs.bootcss.com/api/" \l "mounted)**、**[updated](https://vuejs.bootcss.com/api/" \l "updated)** 和 **[destroyed](https://vuejs.bootcss.com/api/" \l "destroyed)**。生命周期钩子的 this 上下文指向调用它的 Vue 实例。

不要在选项属性或回调上使用**[箭头函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions" \t "https://vuejs.bootcss.com/guide/_blank)**，比如 created: () => console.log(this.a) 或 vm.$watch('a', newValue => this.myMethod())。因为箭头函数并没有 this，this会作为变量一直向上级词法作用域查找，直至找到为止，经常导致 Uncaught TypeError: Cannot read property of undefined 或 Uncaught TypeError: this.myMethod is not a function 之类的错误。



## **[插值](https://vuejs.bootcss.com/guide/syntax.html" \l "%E6%8F%92%E5%80%BC" \o "插值)**

### **[文本](https://vuejs.bootcss.com/guide/syntax.html" \l "%E6%96%87%E6%9C%AC" \o "文本)**

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>

Mustache 标签将会被替代为对应数据对象上 msg 属性的值。无论何时，绑定的数据对象上 msg属性发生了改变，插值处的内容都会更新。

通过使用 **[v-once 指令](https://vuejs.bootcss.com/api/" \l "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的其它数据绑定：

<span v-once>这个将不会改变: {{ msg }}</span>

## **[插值](https://vuejs.bootcss.com/guide/syntax.html" \l "%E6%8F%92%E5%80%BC" \o "插值)**

### **[文本](https://vuejs.bootcss.com/guide/syntax.html" \l "%E6%96%87%E6%9C%AC" \o "文本)**

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>

Mustache 标签将会被替代为对应数据对象上 msg 属性的值。无论何时，绑定的数据对象上 msg属性发生了改变，插值处的内容都会更新。

通过使用 **[v-once 指令](https://vuejs.bootcss.com/api/" \l "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的其它数据绑定：

<span v-once>这个将不会改变: {{ msg }}</span>

### **[原始 HTML](https://vuejs.bootcss.com/guide/syntax.html" \l "%E5%8E%9F%E5%A7%8B-HTML" \o "原始 HTML)**

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 **[v-html 指令](https://vuejs.bootcss.com/api/" \l "v-html)**：

<p>Using mustaches: {{ rawHtml }}</p><p>Using v-html directive: <span v-html="rawHtml"></span></p>

Using mustaches: <span style="color: red">This should be red.</span>

Using v-html directive: This should be red.

这个 span 的内容将会被替换成为属性值 rawHtml，直接作为 HTML——会忽略解析属性值中的数据绑定。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。反之，对于用户界面 (UI)，组件更适合作为可重用和可组合的基本单位。

你的站点上动态渲染的任意 HTML 可能会非常危险，因为它很容易导致 **[XSS 攻击](https://en.wikipedia.org/wiki/Cross-site_scripting" \t "https://vuejs.bootcss.com/guide/_blank)**。请只对可信内容使用 HTML 插值，****绝不要****对用户提供的内容使用插值。

### **[Attribute](https://vuejs.bootcss.com/guide/syntax.html" \l "Attribute" \o "Attribute)**

Mustache 语法不能作用在 HTML attribute 上，遇到这种情况应该使用 **[v-bind 指令](https://vuejs.bootcss.com/api/" \l "v-bind)**：

<div v-bind:id="dynamicId"></div>

对于布尔 attribute (它们只要存在就意味着值为 true)，v-bind 工作起来略有不同，在这个例子中：

<button v-bind:disabled="isButtonDisabled">Button</button>

如果 isButtonDisabled 的值是 null、undefined 或 false，则 disabled attribute 甚至不会被包含在渲染出来的 <button> 元素中。

### **[使用 JavaScript 表达式](https://vuejs.bootcss.com/guide/syntax.html" \l "%E4%BD%BF%E7%94%A8-JavaScript-%E8%A1%A8%E8%BE%BE%E5%BC%8F" \o "使用 JavaScript 表达式)**

迄今为止，在我们的模板中，我们一直都只绑定简单的属性键值。但实际上，对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持。

{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含****单个表达式****，所以下面的例子都****不会****生效。

<!-- 这是语句，不是表达式 -->

{{ var a = 1 }}

<!-- 流控制也不会生效，请使用三元表达式 -->

{{ if (ok) { return message } }}

模板表达式都被放在沙盒中，只能访问**[全局变量的一个白名单](https://github.com/vuejs/vue/blob/v2.6.10/src/core/instance/proxy.js" \l "L9" \t "https://vuejs.bootcss.com/guide/_blank)**，如 Math 和 Date 。你不应该在模板表达式中试图访问用户定义的全局变量。

## **[指令](https://vuejs.bootcss.com/guide/syntax.html" \l "%E6%8C%87%E4%BB%A4" \o "指令)**

指令 (Directives) 是带有 v- 前缀的特殊 attribute。指令 attribute 的值预期是****单个 JavaScript 表达式**** (v-for 是例外情况，稍后我们再讨论)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。回顾我们在介绍中看到的例子：

<p v-if="seen">现在你看到我了</p>

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

### **[参数](https://vuejs.bootcss.com/guide/syntax.html" \l "%E5%8F%82%E6%95%B0" \o "参数)**

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。例如，v-bind 指令可以用于响应式地更新 HTML attribute：

<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 attribute 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">...</a>

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。

### **[动态参数](https://vuejs.bootcss.com/guide/syntax.html" \l "%E5%8A%A8%E6%80%81%E5%8F%82%E6%95%B0" \o "动态参数)**

**2.6.0 新增**

从 2.6.0 开始，可以用方括号括起来的 JavaScript 表达式作为一个指令的参数：

<!--

注意，参数表达式的写法存在一些约束，如之后的“对动态参数表达式的约束”章节所述。

--><a v-bind:[attributeName]="url"> ... </a>

这里的 attributeName 会被作为一个 JavaScript 表达式进行动态求值，求得的值将会作为最终的参数来使用。例如，如果你的 Vue 实例有一个 data 属性 attributeName，其值为 "href"，那么这个绑定将等价于 v-bind:href。

同样地，你可以使用动态参数为一个动态的事件名绑定处理函数：

<a v-on:[eventName]="doSomething"> ... </a>

在这个示例中，当 eventName 的值为 "focus" 时，v-on:[eventName] 将等价于 v-on:focus。

#### **对动态参数的值的约束**

动态参数预期会求出一个字符串，异常情况下值为 null。这个特殊的 null 值可以被显性地用于移除绑定。任何其它非字符串类型的值都将会触发一个警告。

#### **对动态参数表达式的约束**

动态参数表达式有一些语法约束，因为某些字符，如空格和引号，放在 HTML attribute 名里是无效的。例如：

<!-- 这会触发一个编译警告 --><a v-bind:['foo' + bar]="value"> ... </a>

变通的办法是使用没有空格或引号的表达式，或用计算属性替代这种复杂表达式。

在 DOM 中使用模板时 (直接在一个 HTML 文件里撰写模板)，还需要避免使用大写字符来命名键名，因为浏览器会把 attribute 名全部强制转为小写：

<!--

在 DOM 中使用模板时这段代码会被转换为 `v-bind:[someattr]`。

除非在实例中有一个名为“someattr”的 property，否则代码不会工作。

--><a v-bind:[someAttr]="value"> ... </a>

### **[修饰符](https://vuejs.bootcss.com/guide/syntax.html" \l "%E4%BF%AE%E9%A5%B0%E7%AC%A6" \o "修饰符)**

修饰符 (modifier) 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>

## **[缩写](https://vuejs.bootcss.com/guide/syntax.html" \l "%E7%BC%A9%E5%86%99" \o "缩写)**

v- 前缀作为一种视觉提示，用来识别模板中 Vue 特定的 attribute。当你在使用 Vue.js 为现有标签添加动态行为 (dynamic behavior) 时，v- 前缀很有帮助，然而，对于一些频繁用到的指令来说，就会感到使用繁琐。同时，在构建由 Vue 管理所有模板的**[单页面应用程序 (SPA - single page application)](https://en.wikipedia.org/wiki/Single-page_application" \t "https://vuejs.bootcss.com/guide/_blank)** 时，v- 前缀也变得没那么重要了。因此，Vue 为 v-bind 和 v-on 这两个最常用的指令，提供了特定简写：

### **[v-bind 缩写](https://vuejs.bootcss.com/guide/syntax.html" \l "v-bind-%E7%BC%A9%E5%86%99" \o "v-bind 缩写)**

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

<!-- 缩写 --><a :href="url">...</a>

<!-- 动态参数的缩写 (2.6.0+) --><a :[key]="url"> ... </a>

### **[v-on 缩写](https://vuejs.bootcss.com/guide/syntax.html" \l "v-on-%E7%BC%A9%E5%86%99" \o "v-on 缩写)**

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

<!-- 缩写 --><a @click="doSomething">...</a>

<!-- 动态参数的缩写 (2.6.0+) --><a @[event]="doSomething"> ... </a>

它们看起来可能与普通的 HTML 略有不同，但 : 与 @ 对于 attribute 名来说都是合法字符，在所有支持 Vue 的浏览器都能被正确地解析。而且，它们不会出现在最终渲染的标记中。缩写语法是完全可选的，但随着你更深入地了解它们的作用，你会庆幸拥有它们。

# **计算属性和侦听器**

## **[计算属性](https://vuejs.bootcss.com/guide/computed.html" \l "%E8%AE%A1%E7%AE%97%E5%B1%9E%E6%80%A7" \o "计算属性)**

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。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>

在这个地方，模板不再是简单的声明式逻辑。你必须看一段时间才能意识到，这里是想要显示变量 message 的翻转字符串。当你想要在模板中多次引用此处的翻转字符串时，就会更加难以处理。

所以，对于任何复杂逻辑，你都应当使用****计算属性****。

### **[基础例子](https://vuejs.bootcss.com/guide/computed.html" \l "%E5%9F%BA%E7%A1%80%E4%BE%8B%E5%AD%90" \o "基础例子)**

<div id="example">

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

<p>Computed reversed message: "{{ reversedMessage }}"</p></div>

var vm = new Vue({

el: '#example',

data: {

message: 'Hello'

},

computed: {

// 计算属性的 getter

reversedMessage: function () {

// `this` 指向 vm 实例

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

}

}

})

结果：

Original message: "Hello"

Computed reversed message: "olleH"

这里我们声明了一个计算属性 reversedMessage。我们提供的函数将用作属性 vm.reversedMessage 的 getter 函数：

console.log(vm.reversedMessage) // => 'olleH'

vm.message = 'Goodbye'console.log(vm.reversedMessage) // => 'eybdooG'

你可以打开浏览器的控制台，自行修改例子中的 vm。vm.reversedMessage 的值始终取决于 vm.message 的值。

你可以像绑定普通属性一样在模板中绑定计算属性。Vue 知道 vm.reversedMessage 依赖于 vm.message，因此当 vm.message 发生改变时，所有依赖 vm.reversedMessage 的绑定也会更新。而且最妙的是我们已经以声明的方式创建了这种依赖关系：计算属性的 getter 函数是没有副作用 (side effect) 的，这使它更易于测试和理解。

### **[计算属性缓存 vs 方法](https://vuejs.bootcss.com/guide/computed.html" \l "%E8%AE%A1%E7%AE%97%E5%B1%9E%E6%80%A7%E7%BC%93%E5%AD%98-vs-%E6%96%B9%E6%B3%95" \o "计算属性缓存 vs 方法)**

你可能已经注意到我们可以通过在表达式中调用方法来达到同样的效果：

<p>Reversed message: "{{ reversedMessage() }}"</p>

// 在组件中methods: {

reversedMessage: function () {

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

}

}

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是****计算属性是基于它们的响应式依赖进行缓存的****。只在相关响应式依赖发生改变时它们才会重新求值。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。

这也同样意味着下面的计算属性将不再更新，因为 Date.now() 不是响应式依赖：

computed: {

now: function () {

return Date.now()

}

}

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。

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 ****A****，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 ****A****。如果没有缓存，我们将不可避免的多次执行 ****A**** 的 getter！如果你不希望有缓存，请用方法来替代。

### **[计算属性 vs 侦听属性](https://vuejs.bootcss.com/guide/computed.html" \l "%E8%AE%A1%E7%AE%97%E5%B1%9E%E6%80%A7-vs-%E4%BE%A6%E5%90%AC%E5%B1%9E%E6%80%A7" \o "计算属性 vs 侦听属性)**

Vue 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：****侦听属性****。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，通常更好的做法是使用计算属性而不是命令式的 watch 回调。细想一下这个例子：

<div id="demo">{{ fullName }}</div>

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar',

fullName: 'Foo Bar'

},

watch: {

firstName: function (val) {

this.fullName = val + ' ' + this.lastName

},

lastName: function (val) {

this.fullName = this.firstName + ' ' + val

}

}

})

上面代码是命令式且重复的。将它与计算属性的版本进行比较：

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar'

},

computed: {

fullName: function () {

return this.firstName + ' ' + this.lastName

}

}

})

好得多了，不是吗？

### **[计算属性的 setter](https://vuejs.bootcss.com/guide/computed.html" \l "%E8%AE%A1%E7%AE%97%E5%B1%9E%E6%80%A7%E7%9A%84-setter" \o "计算属性的 setter)**

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]

}

}

}// ...

现在再运行 vm.fullName = 'John Doe' 时，setter 会被调用，vm.firstName 和 vm.lastName 也会相应地被更新。

## **[侦听器](https://vuejs.bootcss.com/guide/computed.html" \l "%E4%BE%A6%E5%90%AC%E5%99%A8" \o "侦听器)**

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侦听器。这就是为什么 Vue 通过 watch 选项提供了一个更通用的方法，来响应数据的变化。当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。

例如：

<div id="watch-example">

<p>

Ask a yes/no question:

<input v-model="question">

</p>

<p>{{ answer }}</p></div>

<!-- 因为 AJAX 库和通用工具的生态已经相当丰富，Vue 核心代码没有重复 --><!-- 提供这些功能以保持精简。这也可以让你自由选择自己更熟悉的工具。 --><script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js"></script><script src="https://cdn.jsdelivr.net/npm/lodash@4.13.1/lodash.min.js"></script><script>var watchExampleVM = new Vue({

el: '#watch-example',

data: {

question: '',

answer: 'I cannot give you an answer until you ask a 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()

}

},

created: function () {

  // `\_.debounce` 是一个通过 Lodash 限制操作频率的函数。

  // 在这个例子中，我们希望限制访问 yesno.wtf/api 的频率

  // AJAX 请求直到用户输入完毕才会发出。想要了解更多关于

// `\_.debounce` 函数 (及其近亲 `\_.throttle`) 的知识，

// 请参考：https://lodash.com/docs#debounce

this.debouncedGetAnswer = \_.debounce(this.getAnswer, 500)

},

methods: {

getAnswer: function () {

if (this.question.indexOf('?') === -1) {

this.answer = 'Questions usually contain a question mark. ;-)'

return

}

this.answer = 'Thinking...'

var vm = this

axios.get('https://yesno.wtf/api')

.then(function (response) {

vm.answer = \_.capitalize(response.data.answer)

})

.catch(function (error) {

vm.answer = 'Error! Could not reach the API. ' + error

})

}

}

})</script>

结果：

Ask a yes/no question:

I cannot give you an answer until you ask a question!

在这个示例中，使用 watch 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

除了 watch 选项之外，您还可以使用命令式的 **[vm.$watch API](https://vuejs.bootcss.com/api/" \l "vm-watch)**。

# **Class 与 Style 绑定**

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是属性，所以我们可以用 v-bind 处理它们：只需要通过表达式计算出字符串结果即可。不过，字符串拼接麻烦且易错。因此，在将 v-bind 用于 class 和 style 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，还可以是对象或数组。

## **[绑定 HTML Class](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E7%BB%91%E5%AE%9A-HTML-Class" \o "绑定 HTML Class)**

### **[对象语法](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E5%AF%B9%E8%B1%A1%E8%AF%AD%E6%B3%95" \o "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 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" \t "https://vuejs.bootcss.com/guide/_blank)**。

你可以在对象中传入更多属性来动态切换多个 class。此外，v-bind:class 指令也可以与普通的 class 属性共存。当有如下模板：

<div

class="static"

v-bind:class="{ active: isActive, 'text-danger': hasError }"

></div>

和如下 data：

data: {

isActive: true,

hasError: false

}

结果渲染为：

<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"。

绑定的数据对象不必内联定义在模板里：

<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一个返回对象的**[计算属性](https://vuejs.bootcss.com/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](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E6%95%B0%E7%BB%84%E8%AF%AD%E6%B3%95" \o "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]"></div>

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]"></div>

这样写将始终添加 errorClass，但是只有在 isActive 是 truthy**[[1]](https://vuejs.bootcss.com/guide/class-and-style.html" \l "footnote-1)** 时才添加 activeClass。

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

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

### **[用在组件上](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E7%94%A8%E5%9C%A8%E7%BB%84%E4%BB%B6%E4%B8%8A" \o "用在组件上)**

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

当在一个自定义组件上使用 class 属性时，这些 class 将被添加到该组件的根元素上面。这个元素上已经存在的 class 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 将被渲染为：

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

对于带数据绑定 class 也同样适用：

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

当 isActive 为 truthy**[[1]](https://vuejs.bootcss.com/guide/class-and-style.html" \l "footnote-1)** 时，HTML 将被渲染成为：

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

## **[绑定内联样式](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E7%BB%91%E5%AE%9A%E5%86%85%E8%81%94%E6%A0%B7%E5%BC%8F" \o "绑定内联样式)**

### **[对象语法](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E5%AF%B9%E8%B1%A1%E8%AF%AD%E6%B3%95-1" \o "对象语法)**

v-bind:style 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用引号括起来) 来命名：

<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，这会让模板更清晰：

<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](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E6%95%B0%E7%BB%84%E8%AF%AD%E6%B3%95-1" \o "数组语法)**

v-bind:style 的数组语法可以将多个样式对象应用到同一个元素上：

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

### **[自动添加前缀](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E8%87%AA%E5%8A%A8%E6%B7%BB%E5%8A%A0%E5%89%8D%E7%BC%80" \o "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" \t "https://vuejs.bootcss.com/guide/_blank)**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](https://vuejs.bootcss.com/guide/class-and-style.html" \l "%E5%A4%9A%E9%87%8D%E5%80%BC" \o "多重值)**

**2.3.0+**

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>

这样写只会渲染数组中最后一个被浏览器支持的值。在本例中，如果浏览器支持不带浏览器前缀的 flexbox，那么就只会渲染 display: flex。

****译者注****  
[1] truthy 不是 true，详见 **[MDN](https://developer.mozilla.org/zh-CN/docs/Glossary/Truthy" \t "https://vuejs.bootcss.com/guide/_blank)** 的解释。

# **条件渲染**

## **[v-if](https://vuejs.bootcss.com/guide/conditional.html" \l "v-if" \o "v-if)**

v-if 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回 truthy 值的时候被渲染。

<h1 v-if="awesome">Vue is awesome!</h1>

也可以用 v-else 添加一个“else 块”：

<h1 v-if="awesome">Vue is awesome!</h1><h1 v-else>Oh no </h1>

### **[在 <template> 元素上使用 v-if 条件渲染分组](https://vuejs.bootcss.com/guide/conditional.html" \l "%E5%9C%A8-lt-template-gt-%E5%85%83%E7%B4%A0%E4%B8%8A%E4%BD%BF%E7%94%A8-v-if-%E6%9D%A1%E4%BB%B6%E6%B8%B2%E6%9F%93%E5%88%86%E7%BB%84" \o "在 <template> 元素上使用 v-if 条件渲染分组)**

因为 v-if 是一个指令，所以必须将它添加到一个元素上。但是如果想切换多个元素呢？此时可以把一个 <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](https://vuejs.bootcss.com/guide/conditional.html" \l "v-else" \o "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](https://vuejs.bootcss.com/guide/conditional.html" \l "v-else-if" \o "v-else-if)**

**2.1.0 新增**

v-else-if，顾名思义，充当 v-if 的“else-if 块”，可以连续使用：

<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 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](https://vuejs.bootcss.com/guide/conditional.html" \l "%E7%94%A8-key-%E7%AE%A1%E7%90%86%E5%8F%AF%E5%A4%8D%E7%94%A8%E7%9A%84%E5%85%83%E7%B4%A0" \o "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>

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。因为两个模板使用了相同的元素，<input> 不会被替换掉——仅仅是替换了它的 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表达“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。请看：

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](https://vuejs.bootcss.com/guide/conditional.html" \l "v-show" \o "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 只是简单地切换元素的 CSS 属性 display。

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

## **[v-if vs v-show](https://vuejs.bootcss.com/guide/conditional.html" \l "v-if-vs-v-show" \o "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；如果在运行时条件很少改变，则使用 v-if 较好。

## **[v-if 与 v-for 一起使用](https://vuejs.bootcss.com/guide/conditional.html" \l "v-if-%E4%B8%8E-v-for-%E4%B8%80%E8%B5%B7%E4%BD%BF%E7%94%A8" \o "v-if 与 v-for 一起使用)**

****不推荐****同时使用 v-if 和 v-for。请查阅**[风格指南](https://vuejs.bootcss.com/v2/style-guide/" \l "%E9%81%BF%E5%85%8D-v-if-%E5%92%8C-v-for-%E7%94%A8%E5%9C%A8%E4%B8%80%E8%B5%B7-%E5%BF%85%E8%A6%81)**以获取更多信息。

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。请查阅**[列表渲染指南](https://vuejs.bootcss.com/guide/list.html" \l "v-for-with-v-if)** 以获取详细信息。

# **列表渲染**

## **[用 v-for 把一个数组对应为一组元素](https://vuejs.bootcss.com/guide/list.html" \l "%E7%94%A8-v-for-%E6%8A%8A%E4%B8%80%E4%B8%AA%E6%95%B0%E7%BB%84%E5%AF%B9%E5%BA%94%E4%B8%BA%E4%B8%80%E7%BB%84%E5%85%83%E7%B4%A0" \o "用 v-for 把一个数组对应为一组元素)**

我们可以用 v-for 指令基于一个数组来渲染一个列表。v-for 指令需要使用 item in items形式的特殊语法，其中 items 是源数据数组，而 item 则是被迭代的数组元素的****别名****。

<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，我们可以访问所有父作用域的属性。v-for 还支持一个可选的第二个参数，即当前项的索引。

<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，因为它更接近 JavaScript 迭代器的语法：

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

## **[在 v-for 里使用对象](https://vuejs.bootcss.com/guide/list.html" \l "%E5%9C%A8-v-for-%E9%87%8C%E4%BD%BF%E7%94%A8%E5%AF%B9%E8%B1%A1" \o "在 v-for 里使用对象)**

你也可以用 v-for 来遍历一个对象的属性。

<ul id="v-for-object" class="demo">

<li v-for="value in object">

{{ value }}

</li></ul>

new Vue({

el: '#v-for-object',

data: {

object: {

title: 'How to do lists in Vue',

author: 'Jane Doe',

publishedAt: '2016-04-10'

}

}

})

结果：

* How to do lists in Vue
* Jane Doe
* 2016-04-10

你也可以提供第二个的参数为 property 名称 (也就是键名)：

<div v-for="(value, name) in object">

{{ name }}: {{ value }}</div>

title: How to do lists in Vue

author: Jane Doe

publishedAt: 2016-04-10

还可以用第三个参数作为索引：

<div v-for="(value, name, index) in object">

{{ index }}. {{ name }}: {{ value }}</div>

0. title: How to do lists in Vue

1. author: Jane Doe

2. publishedAt: 2016-04-10

在遍历对象时，会按 Object.keys() 的结果遍历，但是****不能****保证它的结果在不同的 JavaScript 引擎下都一致。

## **[维护状态](https://vuejs.bootcss.com/guide/list.html" \l "%E7%BB%B4%E6%8A%A4%E7%8A%B6%E6%80%81" \o "维护状态)**

当 Vue 正在更新使用 v-for 渲染的元素列表时，它默认使用“就地更新”的策略。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，而是就地更新每个元素，并且确保它们在每个索引位置正确渲染。这个类似 Vue 1.x 的 track-by="$index"。

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****。

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：

<div v-for="item in items" v-bind:key="item.id">

<!-- 内容 --></div>

建议尽可能在使用 v-for 时提供 key attribute，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升。

因为它是 Vue 识别节点的一个通用机制，key 并不仅与 v-for 特别关联。后面我们将在指南中看到，它还具有其它用途。

不要使用对象或数组之类的非基本类型值作为 v-for 的 key。请用字符串或数值类型的值。

更多 key attribute 的细节用法请移步至 **[key 的 API 文档](https://cn.vuejs.org/v2/api/" \l "key)**。

## **[数组更新检测](https://vuejs.bootcss.com/guide/list.html" \l "%E6%95%B0%E7%BB%84%E6%9B%B4%E6%96%B0%E6%A3%80%E6%B5%8B" \o "数组更新检测)**

### **[变异方法 (mutation method)](https://vuejs.bootcss.com/guide/list.html" \l "%E5%8F%98%E5%BC%82%E6%96%B9%E6%B3%95-mutation-method" \o "变异方法 (mutation method))**

Vue 将被侦听的数组的变异方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

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

你可以打开控制台，然后对前面例子的 items 数组尝试调用变异方法。比如 example1.items.push({ message: 'Baz' })。

### **[替换数组](https://vuejs.bootcss.com/guide/list.html" \l "%E6%9B%BF%E6%8D%A2%E6%95%B0%E7%BB%84" \o "替换数组)**

变异方法，顾名思义，会改变调用了这些方法的原始数组。相比之下，也有非变异 (non-mutating method) 方法，例如 filter()、concat() 和 slice() 。它们不会改变原始数组，而****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 为了使得 DOM 元素得到最大范围的重用而实现了一些智能的启发式方法，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](https://vuejs.bootcss.com/guide/list.html" \l "%E6%B3%A8%E6%84%8F%E4%BA%8B%E9%A1%B9" \o "注意事项)**

由于 JavaScript 的限制，Vue ****不能****检测以下数组的变动：

1. 当你利用索引直接设置一个数组项时，例如：vm.items[indexOfItem] = newValue
2. 当你修改数组的长度时，例如：vm.items.length = newLength

举个例子：

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

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，同时也将在响应式系统内触发状态更新：

// Vue.set

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

// Array.prototype.splice

vm.items.splice(indexOfItem, 1, newValue)

你也可以使用 **[vm.$set](https://cn.vuejs.org/v2/api/" \l "vm-set)** 实例方法，该方法是全局方法 Vue.set 的一个别名：

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

为了解决第二类问题，你可以使用 splice：

vm.items.splice(newLength)

## **[对象变更检测注意事项](https://vuejs.bootcss.com/guide/list.html" \l "%E5%AF%B9%E8%B1%A1%E5%8F%98%E6%9B%B4%E6%A3%80%E6%B5%8B%E6%B3%A8%E6%84%8F%E4%BA%8B%E9%A1%B9" \o "对象变更检测注意事项)**

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

var vm = new Vue({

data: {

a: 1

}

})// `vm.a` 现在是响应式的

vm.b = 2// `vm.b` 不是响应式的

对于已经创建的实例，Vue 不允许动态添加根级别的响应式属性。但是，可以使用 Vue.set(object, propertyName, value) 方法向嵌套对象添加响应式属性。例如，对于：

var vm = new Vue({

data: {

userProfile: {

name: 'Anika'

}

}

})

你可以添加一个新的 age 属性到嵌套的 userProfile 对象：

Vue.set(vm.userProfile, 'age', 27)

你还可以使用 vm.$set 实例方法，它只是全局 Vue.set 的别名：

vm.$set(vm.userProfile, 'age', 27)

有时你可能需要为已有对象赋值多个新属性，比如使用 Object.assign() 或 \_.extend()。在这种情况下，你应该用两个对象的属性创建一个新的对象。所以，如果你想添加新的响应式属性，不要像这样：

Object.assign(vm.userProfile, {

age: 27,

favoriteColor: 'Vue Green'

})

你应该这样做：

vm.userProfile = Object.assign({}, vm.userProfile, {

age: 27,

favoriteColor: 'Vue Green'

})

## **[显示过滤/排序后的结果](https://vuejs.bootcss.com/guide/list.html" \l "%E6%98%BE%E7%A4%BA%E8%BF%87%E6%BB%A4-%E6%8E%92%E5%BA%8F%E5%90%8E%E7%9A%84%E7%BB%93%E6%9E%9C" \o "显示过滤/排序后的结果)**

有时，我们想要显示一个数组经过过滤或排序后的版本，而不实际改变或重置原始数据。在这种情况下，可以创建一个计算属性，来返回过滤或排序后的数组。

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

})

}

}

在计算属性不适用的情况下 (例如，在嵌套 v-for 循环中) 你可以使用一个方法：

<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

})

}

}

## **[在 v-for 里使用值范围](https://vuejs.bootcss.com/guide/list.html" \l "%E5%9C%A8-v-for-%E9%87%8C%E4%BD%BF%E7%94%A8%E5%80%BC%E8%8C%83%E5%9B%B4" \o "在 v-for 里使用值范围)**

v-for 也可以接受整数。在这种情况下，它会把模板重复对应次数。

<div>

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

结果：

1 2 3 4 5 6 7 8 9 10

## **[在 <template> 上使用 v-for](https://vuejs.bootcss.com/guide/list.html" \l "%E5%9C%A8-lt-template-gt-%E4%B8%8A%E4%BD%BF%E7%94%A8-v-for" \o "在 <template> 上使用 v-for)**

类似于 v-if，你也可以利用带有 v-for 的 <template> 来循环渲染一段包含多个元素的内容。比如：

<ul>

<template v-for="item in items">

<li>{{ item.msg }}</li>

<li class="divider" role="presentation"></li>

</template></ul>

## **[v-for 与 v-if 一同使用](https://vuejs.bootcss.com/guide/list.html" \l "v-for-%E4%B8%8E-v-if-%E4%B8%80%E5%90%8C%E4%BD%BF%E7%94%A8" \o "v-for 与 v-if 一同使用)**

注意我们****不****推荐在同一元素上使用 v-if 和 v-for。更多细节可查阅**[风格指南](https://vuejs.bootcss.com/v2/style-guide/" \l "%E9%81%BF%E5%85%8D-v-if-%E5%92%8C-v-for-%E7%94%A8%E5%9C%A8%E4%B8%80%E8%B5%B7-%E5%BF%85%E8%A6%81)**。

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。当你只想为部分项渲染节点时，这种优先级的机制会十分有用，如下：

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

{{ todo }}</li>

上面的代码将只渲染未完成的 todo。

而如果你的目的是有条件地跳过循环的执行，那么可以将 v-if 置于外层元素 (或 **[<template>](https://vuejs.bootcss.com/guide/conditional.html" \l "%E5%9C%A8-lt-template-gt-%E4%B8%AD%E9%85%8D%E5%90%88-v-if-%E6%9D%A1%E4%BB%B6%E6%B8%B2%E6%9F%93%E4%B8%80%E6%95%B4%E7%BB%84)**)上。如：

<ul v-if="todos.length">

<li v-for="todo in todos">

{{ todo }}

</li></ul><p v-else>No todos left!</p>

## **[在组件上使用 v-for](https://vuejs.bootcss.com/guide/list.html" \l "%E5%9C%A8%E7%BB%84%E4%BB%B6%E4%B8%8A%E4%BD%BF%E7%94%A8-v-for" \o "在组件上使用 v-for)**

**这部分内容假定你已经了解[组件](https://vuejs.bootcss.com/guide/components.html)相关知识。你也完全可以先跳过它，以后再回来查看。**

在自定义组件上，你可以像在任何普通元素上一样使用 v-for 。

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

**2.2.0+ 的版本里，当在组件上使用 v-for 时，key 现在是必须的。**

然而，任何数据都不会被自动传递到组件里，因为组件有自己独立的作用域。为了把迭代数据传递到组件里，我们要使用 prop：

<my-component

v-for="(item, index) in items"

v-bind:item="item"

v-bind:index="index"

v-bind:key="item.id"

></my-component>

不自动将 item 注入到组件里的原因是，这会使得组件与 v-for 的运作紧密耦合。明确组件数据的来源能够使组件在其他场合重复使用。

下面是一个简单的 todo 列表的完整例子：

<div id="todo-list-example">

<form v-on:submit.prevent="addNewTodo">

<label for="new-todo">Add a todo</label>

<input

v-model="newTodoText"

id="new-todo"

placeholder="E.g. Feed the cat"

>

<button>Add</button>

</form>

<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></div>

注意这里的 is="todo-item" 属性。这种做法在使用 DOM 模板时是十分必要的，因为在 <ul> 元素内只有 <li> 元素会被看作有效内容。这样做实现的效果与 <todo-item> 相同，但是可以避开一些潜在的浏览器解析错误。查看 **[DOM 模板解析说明](https://vuejs.bootcss.com/guide/components.html" \l "%E8%A7%A3%E6%9E%90-DOM-%E6%A8%A1%E6%9D%BF%E6%97%B6%E7%9A%84%E6%B3%A8%E6%84%8F%E4%BA%8B%E9%A1%B9)** 来了解更多信息。

Vue.component('todo-item', {

template: '\

<li>\

{{ title }}\

<button v-on:click="$emit(\'remove\')">Remove</button>\

</li>\

',

props: ['title']

})

new Vue({

el: '#todo-list-example',

data: {

newTodoText: '',

todos: [

{

id: 1,

title: 'Do the dishes',

},

{

id: 2,

title: 'Take out the trash',

},

{

id: 3,

title: 'Mow the lawn'

}

],

nextTodoId: 4

},

methods: {

addNewTodo: function () {

this.todos.push({

id: this.nextTodoId++,

title: this.newTodoText

})

this.newTodoText = ''

}

}

})

# API介绍：

Vue.config是一个对象，包含vue的全局配置，可以在启动之前修改下列属性:

#silent,

类型 boolean，

默认值 false

用法Vue.config.silent = true.取消vue的所有日志和警告

#[optionMergeStrategies](https://vuejs.bootcss.com/api/" \l "optionMergeStrategies" \o "optionMergeStrategies)

类型：{ [key: string]: Function }

默认值：{}

用法：

Vue.config.optionMergeStrategies.\_my\_option = function (parent, child, vm) {

return child + 1

}

const Profile = Vue.extend({

\_my\_option: 1

})

// Profile.options.\_my\_option = 2

自定义合并策略的选项。

合并策略选项分别接收在父实例和子实例上定义的该选项的值作为第一个和第二个参数，Vue 实例上下文被作为第三个参数传入。

#[errorHandler](https://vuejs.bootcss.com/api/" \l "errorHandler" \o "errorHandler)

类型：Function

默认值：undefined

用法：

Vue.config.errorHandler = function (err, vm, info) {

// handle error

// `info` 是 Vue 特定的错误信息，比如错误所在的生命周期钩子

// 只在 2.2.0+ 可用

}

指定组件的渲染和观察期间未捕获错误的处理函数。这个处理函数被调用时，可获取错误信息和 Vue 实例。

从 2.2.0 起，这个钩子也会捕获组件生命周期钩子里的错误。同样的，当这个钩子是 undefined 时，被捕获的错误会通过 console.error 输出而避免应用崩溃。

从 2.4.0 起，这个钩子也会捕获 Vue 自定义事件处理函数内部的错误了。

从 2.6.0 起，这个钩子也会捕获 v-on DOM 监听器内部抛出的错误。另外，如果任何被覆盖的钩子或处理函数返回一个 Promise 链 (例如 async 函数)，则来自其 Promise 链的错误也会被处理。

错误追踪服务 [Sentry](https://sentry.io/" \t "https://vuejs.bootcss.com/api/_blank) 和 [Bugsnag](https://docs.bugsnag.com/platforms/browsers/vue/" \t "https://vuejs.bootcss.com/api/_blank) 都通过此选项提供了官方支持。

参考 [自定义选项的混入策略](https://vuejs.bootcss.com/guide/mixins.html" \l "%E8%87%AA%E5%AE%9A%E4%B9%89%E9%80%89%E9%A1%B9%E6%B7%B7%E5%85%A5%E7%AD%96%E7%95%A5)

#[errorHandler](https://vuejs.bootcss.com/api/" \l "errorHandler" \o "errorHandler)

类型：Function

默认值：undefined

用法：

Vue.config.errorHandler = function (err, vm, info) {

// handle error

// `info` 是 Vue 特定的错误信息，比如错误所在的生命周期钩子

// 只在 2.2.0+ 可用

}

指定组件的渲染和观察期间未捕获错误的处理函数。这个处理函数被调用时，可获取错误信息和 Vue 实例。

从 2.2.0 起，这个钩子也会捕获组件生命周期钩子里的错误。同样的，当这个钩子是 undefined 时，被捕获的错误会通过 console.error 输出而避免应用崩溃。

从 2.4.0 起，这个钩子也会捕获 Vue 自定义事件处理函数内部的错误了。

从 2.6.0 起，这个钩子也会捕获 v-on DOM 监听器内部抛出的错误。另外，如果任何被覆盖的钩子或处理函数返回一个 Promise 链 (例如 async 函数)，则来自其 Promise 链的错误也会被处理。

错误追踪服务 [Sentry](https://sentry.io/" \t "https://vuejs.bootcss.com/api/_blank) 和 [Bugsnag](https://docs.bugsnag.com/platforms/browsers/vue/" \t "https://vuejs.bootcss.com/api/_blank) 都通过此选项提供了官方支持。