Vue.js

# 基础

## 介绍

Vue (读音 /vjuː/，类似于 view) 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

## 安装

### 安装Vue.js

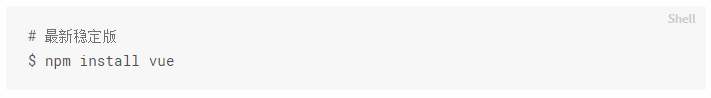
* **<script> 标签引入**

在项目中使用Vue.js框架需要提前下载Vue.js框架代码。然后像在页面中引入普通的JavaScript代码一样使用<script> 标签引入。Vue 会被注册为一个全局变量。



* **NPM安装**

在用 Vue 构建大型应用时推荐使用 NPM 安装[1]。NPM 能很好地和诸如 webpack 或 Browserify 模块打包器配合使用。同时 Vue 也提供配套工具来开发单文件组件。





* **命令行工具 (CLI)**

Vue 提供了一个官方的 CLI，为单页面应用 (SPA) 快速搭建繁杂的脚手架。它为现代前端工作流提供了 batteries-included 的构建设置。只需要几分钟的时间就可以运行起来并带有热重载、保存时 lint 校验，以及生产环境可用的构建版本。

### Vue Devtools

Vue Devtools是Vue提供的用于调试Vue项目的调试工具，在使用 Vue 时，我们推荐在你的浏览器上安装 Vue Devtools。它允许你在一个更友好的界面中审查和调试 Vue 应用。

1. 我们需要到GitHub（https://github.com/vuejs/vue-devtools#vue-devtools）中克隆Vue Devtools的源代码到本地。

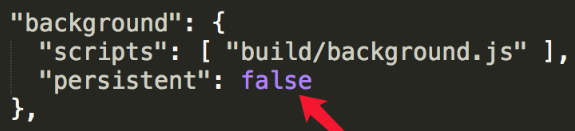


1. 使用命令行进入到本地的Vue Devtools中，使用cnpm install命令安装项目所需要的npm包。



1. 修改manifest.json文件

进入项目中找到manifest.json文件，将其中的"persistent":false改成true。



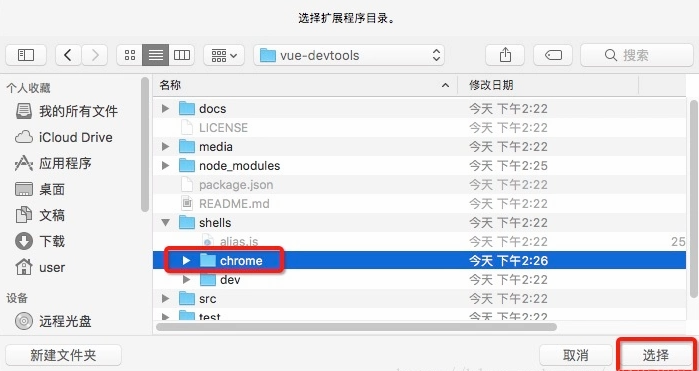
1. 开始编译项目文件

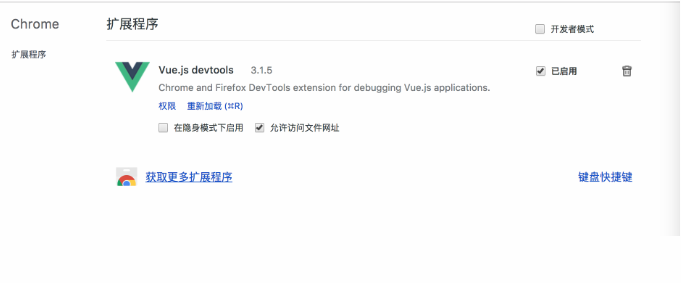


1. 添加至chrome游览器

浏览器输入地址“chrome://extensions/”进入扩展程序页面，点击“加载已解压的扩展程序...”按钮，选择vue-devtools>shells下的chrome文件夹。如果看不见“加载已解压的扩展程序...”按钮，则需要勾选“开发者模式”。



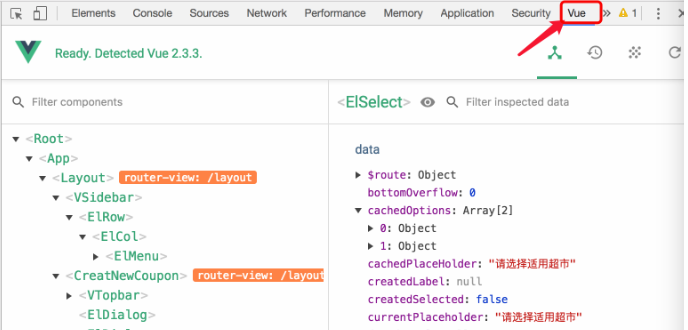




1. Vue-Devtools使用

vue项目, 打开f12, 选择vue就可以使用了.

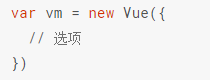
vue是数据驱动的, 这样就能看到对应数据了, 方便我们进行调试



## Vue实例

### 创建Vue实例

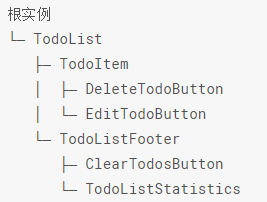
每个 Vue 应用都是通过用 Vue 函数创建一个新的 Vue 实例开始的：



虽然没有完全遵循 MVVM 模型，但是 Vue 的设计也受到了它的启发。因此在文档中经常会使用 vm (ViewModel 的缩写) 这个变量名表示 Vue 实例。

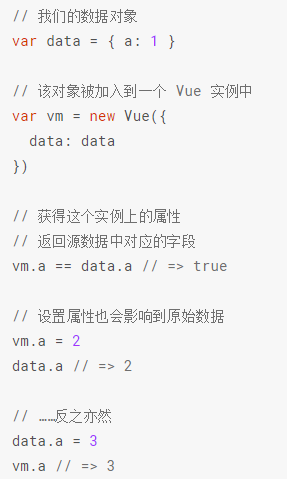
当创建一个 Vue 实例时，你可以传入一个选项对象。这篇教程主要描述的就是如何使用这些选项来创建你想要的行为。

一个 Vue 应用由一个通过 new Vue 创建的根 Vue 实例，以及可选的嵌套的、可复用的组件树组成。举个例子，一个 todo 应用的组件树可以是这样的：



### 数据与方法

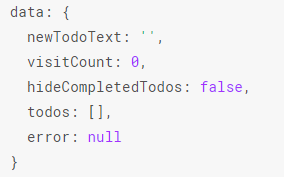
当一个 Vue 实例被创建时，它将 data 对象中的所有的属性加入到 Vue 的响应式系统中。当这些属性的值发生改变时，视图将会产生“响应”，即匹配更新为新的值。



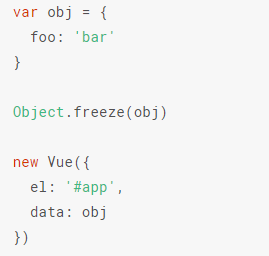
当这些数据改变时，视图会进行重渲染。值得注意的是只有当实例被创建时 data 中存在的属性才是响应式的。也就是说如果你添加一个新的属性，比如：

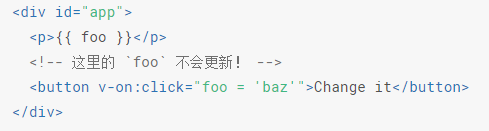


那么对 b 的改动将不会触发任何视图的更新。如果你知道你会在晚些时候需要一个属性，但是一开始它为空或不存在，那么你仅需要设置一些初始值。比如：



这里唯一的例外是使用 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。





除了数据属性，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：



### 生命周期

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 钩子可以用来在一个实例被创建之后执行代码：



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、updated 和 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，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。

**生命周期图示：**



## 模板语法

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML ，所以能被遵循规范的浏览器和 HTML 解析器解析。

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，直接写渲染 (render) 函数，使用可选的 JSX 语法。

### 插值

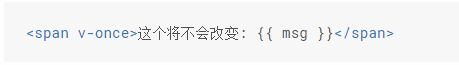
* + **文本**

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



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，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：



* + **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：





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。

* + **属性**

Mustache 语法不能作用在 HTML 属性上，遇到这种情况应该使用 v-bind 指令：



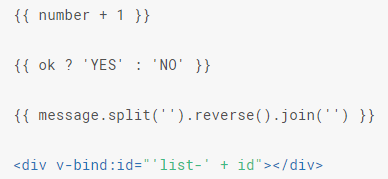
对于布尔特性 (它们只要存在就意味着值为 true)，v-bind 工作起来略有不同，在这个例子中：



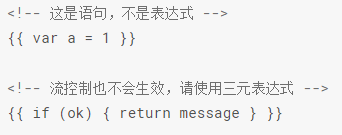
如果 isButtonDisabled 的值是 null、undefined 或 false，则 disabled 特性甚至不会被包含在渲染出来的 <button> 元素中。

* + **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。



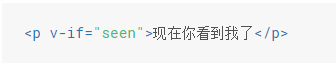
这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。



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，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

指令 (Directives) 是带有 v- 前缀的特殊属性。指令属性的值预期是单个 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：



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**

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 特性：



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：



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**

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：



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：



同样地，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，例如空格和引号，放在 HTML 特性名里是无效的。同样，在 DOM 中使用模板时你需要回避大写键名。

例如，下面的代码是无效的：



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。

另外，如果你在 DOM 中使用模板 (直接在一个 HTML 文件里撰写模板)，需要留意浏览器会把特性名全部强制转为小写：



* + **修饰符**

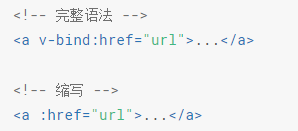
修饰符 (modifier) 是以半角句号 . 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。例如，.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()：



### 缩写

v- 前缀作为一种视觉提示，用来识别模板中 Vue 特定的属性。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) 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**



* + **v-on 缩写**



它们看起来可能与普通的 HTML 略有不同，但 : 与 @ 对于属性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

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：



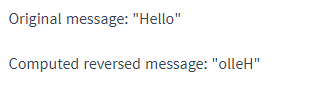
在这个地方，模板不再是简单的声明式逻辑。你必须看一段时间才能意识到，这里是想要显示变量 message 的翻转字符串。当你想要在模板中多次引用此处的翻转字符串时，就会更加难以处理。

所以，对于任何复杂逻辑，你都应当使用计算属性。

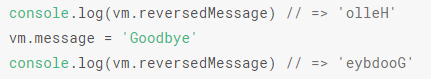
#### 基础例子







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：



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

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：





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



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

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。





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：



#### 计算属性的 setter

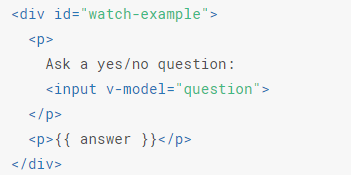
计算属性默认只有 getter ，不过在需要时你也可以提供一个 setter ：

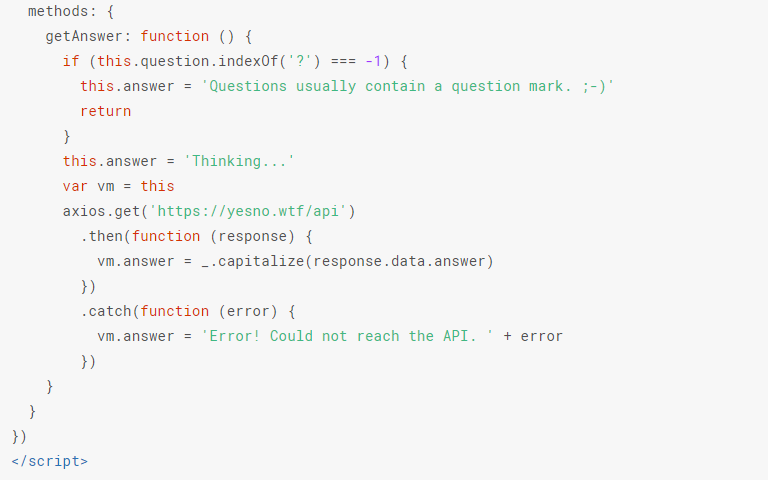


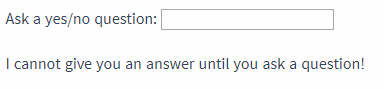
现在再运行 vm.fullName = 'John Doe' 时，setter 会被调用，vm.firstName 和 vm.lastName 也会相应地被更新。

### 侦听器

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。







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。

## 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

* + **对象语法**

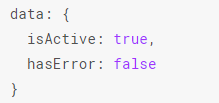
我们可以传给 v-bind:class 一个对象，以动态地切换 class：



上面的语法表示 active 这个 class 存在与否将取决于数据属性 isActive返回的布尔值类型。

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 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"。

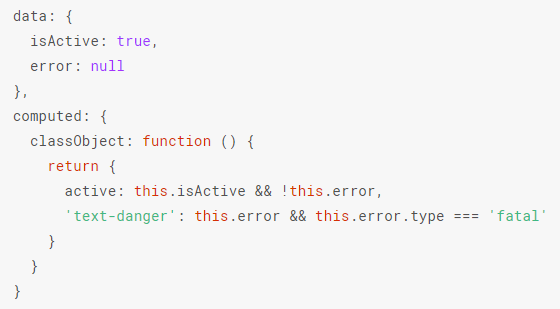
绑定的数据对象不必内联定义在模板里：





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。这是一个常用且强大的模式：





* + **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：







如果你也想根据条件切换列表中的 class，可以用三元表达式：



这样写将始终添加 errorClass，但是只有在 isActive 是 truthy[1] 时才添加 activeClass。

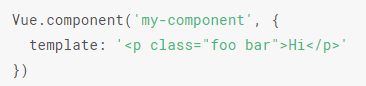
不过，当有多个条件 class 时这样写有些繁琐。所以在数组语法中也可以使用对象语法：



* + **用在组件上**

当在一个自定义组件上使用 class 属性时，这些类将被添加到该组件的根元素上面。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：



然后在使用它的时候添加一些 class：



结果为：



对于带数据绑定 class 也同样适用：



当 isActive 为 truthy[1] 时，HTML 将被渲染成为：



### 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，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：





直接绑定到一个样式对象通常更好，这会让模板更清晰：





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同一个元素上：



* + **自动添加前缀**

当 v-bind:style 使用需要添加浏览器引擎前缀的 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：



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。

## 条件渲染

### 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 布尔值的时候被渲染。

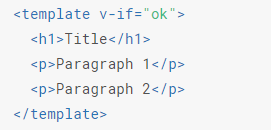


也可以用 v-else 添加一个“else 块”：



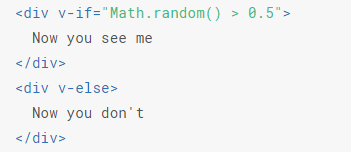
* + **在 <template> 元素上使用 v-if 条件渲染分组**

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。



* + **v-else**

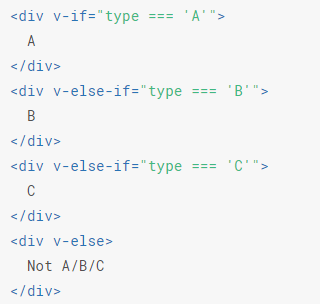
你可以使用 v-else 指令来表示 v-if 的“else 块”：



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**

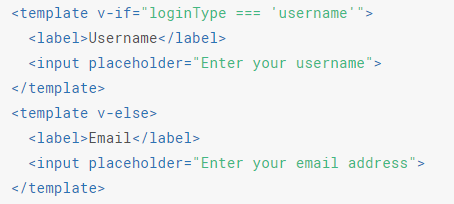
v-else-if，顾名思义，充当 v-if 的“else-if 块”，可以连续使用：



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。

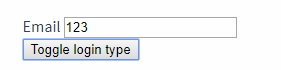
* + **用 key 管理可复用的元素**

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：

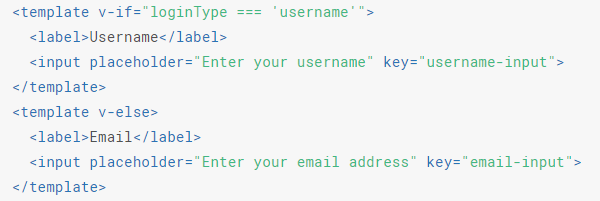


那么在上面的代码中切换 loginType 将不会清除用户已经输入的内容。因为两个模板使用了相同的元素，<input> 不会被替换掉——仅仅是替换了它的 placeholder。

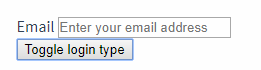
自己动手试一试，在输入框中输入一些文本，然后按下切换按钮：



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：



现在，每次切换时，输入框都将被重新渲染。请看：



### 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：



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

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。

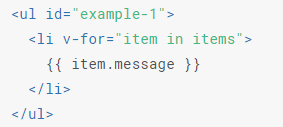
相比之下，v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

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。

## 循环渲染

### v-for遍历数组

v-for指令用于对数据进行循环操作，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。



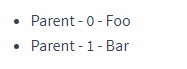




在 v-for 块中，我们可以访问所有父作用域的属性。v-for 还支持一个可选的第二个参数，即当前项的索引。





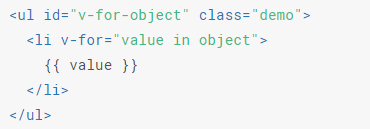


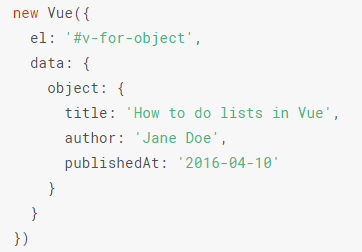
你也可以用 of 替代 in 作为分隔符，因为它更接近 JavaScript 迭代器的语法：

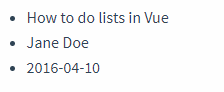


### v-for 遍历对象

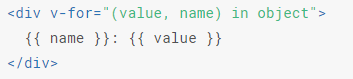
你也可以用 v-for 来遍历一个对象的属性。

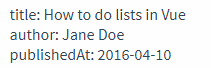




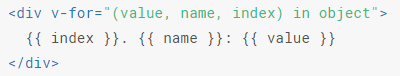


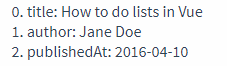
你也可以提供第二个的参数为 property 名称 (也就是键名)：





还可以用第三个参数作为索引：





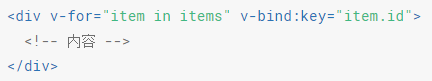
在遍历对象时，会按 Object.keys() 的结果遍历，但是不能保证它的结果在不同的 JavaScript 引擎下都一致。

### 维护状态

我们使用v-for指令遍历数组，当该数组的值发生变化，Vue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：



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。

### 数组更新检测

* **变异方法**

Vue重写了JavaScript中对数组操作的方法，使用Vue重写的这些方法对数组操作，可以触发视图的更新。如果直接对原始数组进行操作，则不会触发视图的更新。这些方法包括以下：

push()

pop()

shift()

unshift()

splice()

sort()

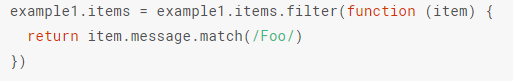
reverse()

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



* **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：

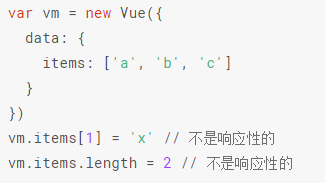


你可能认为这将导致 Vue 丢弃现有 DOM 并重新渲染整个列表。幸运的是，事实并非如此。Vue 为了使得 DOM 元素得到最大范围的重用而实现了一些智能的启发式方法，所以用一个含有相同元素的数组去替换原来的数组是非常高效的操作。

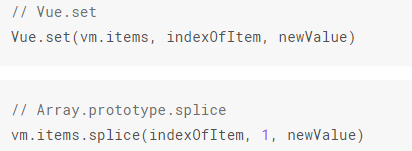
* **注意事项**

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



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：



你也可以使用 vm.$set 实例方法，该方法是全局方法 Vue.set 的一个别名：

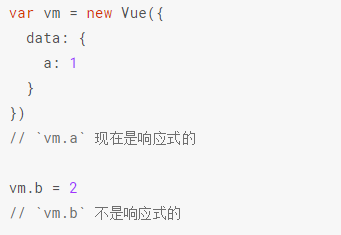


为了解决第二类问题，你可以使用 splice：

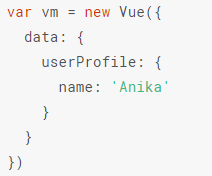


### 对象变更检测注意事项

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



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：



你可以添加一个新的 age 属性到嵌套的 userProfile 对象：



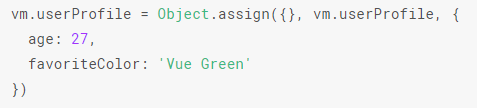
你还可以使用 vm.$set 实例方法，它只是全局 Vue.set 的别名：



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：



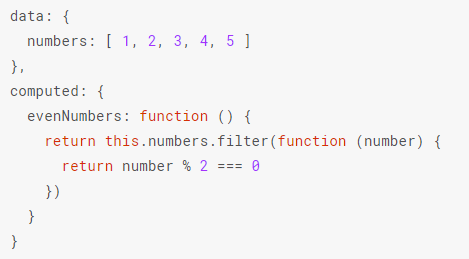
你应该这样做：



### 显示过滤/排序后的结果

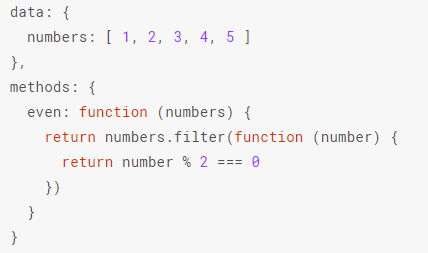
有时，我们想要显示一个数组经过过滤或排序后的版本，而不实际改变或重置原始数据。在这种情况下，可以创建一个计算属性，来返回过滤或排序后的数组。





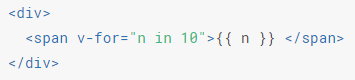
在计算属性不适用的情况下 (例如，在嵌套 v-for 循环中) 你可以使用一个方法：





### 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。

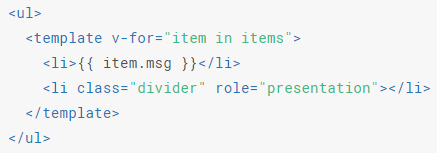


结果：



### <template> 使用 v-for

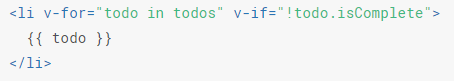
类似于 v-if，你也可以利用带有 v-for 的 <template> 来循环渲染一段包含多个元素的内容。



### v-for 与 v-if 一同使用

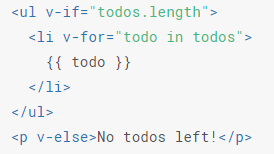
注意我们不推荐在同一元素上使用 v-if 和 v-for。

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：



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>)上。如：



### 在组件上使用 v-for

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 。



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：



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。

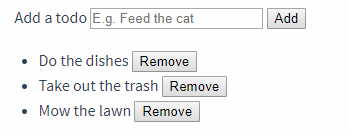
下面是一个简单的 todo 列表的完整例子：



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。







## 事件处理

### 监听事件

可以用 v-on 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码。





### 事件处理方法

然而许多事件处理逻辑会更为复杂，所以直接把 JavaScript 代码写在 v-on 指令中是不可行的。因此 v-on 还可以接收一个需要调用的方法名称。





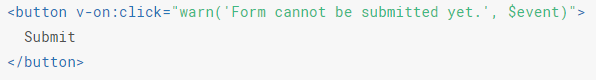
### 内联处理器中的方法

除了直接绑定到一个方法，也可以在内联 JavaScript 语句中调用方法：





有时也需要在内联语句处理器中访问原始的 DOM 事件。可以用特殊变量 $event 把它传入方法：





### 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。尽管我们可以在方法中轻松实现这点，但更好的方式是：方法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。之前提过，修饰符是由点开头的指令后缀来表示的。

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



使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 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上。如果你还没有阅读关于组件的文档，现在大可不必担心。

Vue 还对应 addEventListener 中的 passive 选项提供了 .passive 修饰符。



这个 .passive 修饰符尤其能够提升移动端的性能。

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

在监听键盘事件时，我们经常需要检查详细的按键。Vue 允许为 v-on 在监听键盘事件时添加按键修饰符：



你可以直接将 KeyboardEvent.key 暴露的任意有效按键名转换为 kebab-case 来作为修饰符。



在上述示例中，处理函数只会在 $event.key 等于 PageDown 时被调用。

为了在必要的情况下支持旧浏览器，Vue 提供了绝大多数常用的按键码的别名：

* .enter
* .tab
* .delete (捕获“删除”和“退格”键)
* .esc
* .space
* .up
* .down
* .left
* .right

你还可以通过全局 config.keyCodes 对象自定义按键修饰符别名：

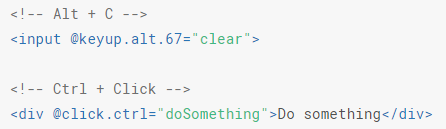


### 系统修饰键

可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。

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

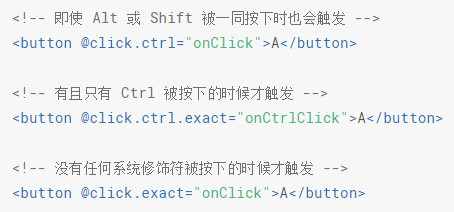
注意：在 Mac 系统键盘上，meta 对应 command 键 (⌘)。在 Windows 系统键盘 meta 对应 Windows 徽标键 (⊞)。在 Sun 操作系统键盘上，meta 对应实心宝石键 (◆)。在其他特定键盘上，尤其在 MIT 和 Lisp 机器的键盘、以及其后继产品，比如 Knight 键盘、space-cadet 键盘，meta 被标记为“META”。在 Symbolics 键盘上，meta 被标记为“META”或者“Meta”。



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**

.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。



* **鼠标按钮修饰符**
* .left
* .right
* .middle

这些修饰符会限制处理函数仅响应特定的鼠标按钮。

### 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

你可以用 v-model 指令在表单 <input>、<textarea> 及 <select> 元素上创建双向数据绑定。它会根据控件类型自动选取正确的方法来更新元素。尽管有些神奇，但 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。

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 Vue 实例的数据作为数据来源。你应该通过 JavaScript 在组件的 data 选项中声明初始值。

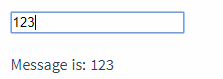
v-model 在内部为不同的输入元素使用不同的属性并抛出不同的事件：

* text 和 textarea 元素使用 value 属性和 input 事件；
* checkbox 和 radio 使用 checked 属性和 change 事件；
* select 字段将 value 作为 prop 并将 change 作为事件。

对于需要使用输入法 (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。

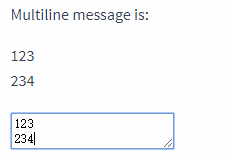
#### 文本





#### 多行文本





在文本区域插值 (<textarea>{{text}}</textarea>) 并不会生效，应用 v-model 来代替。

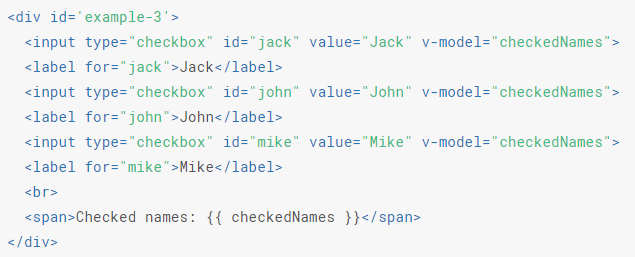
#### 复选框

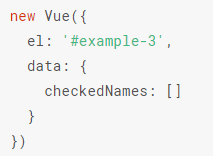
单个复选框，绑定到布尔值：





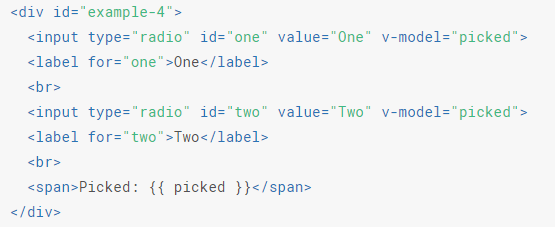
多个复选框，绑定到同一个数组：



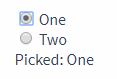




#### 单选按钮







#### 选择框

* **单选时：**



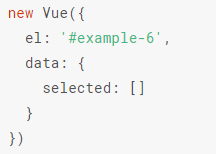


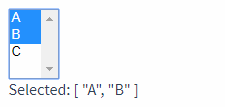


如果 v-model 表达式的初始值未能匹配任何选项，<select> 元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第一个选项。因为这样的情况下，iOS 不会触发 change 事件。因此，更推荐像上面这样提供一个值为空的禁用选项。

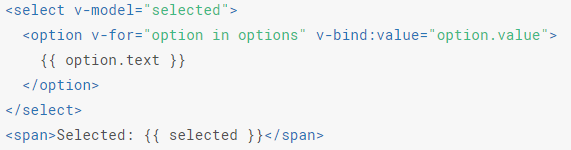
* **多选时 (绑定到一个数组)：**

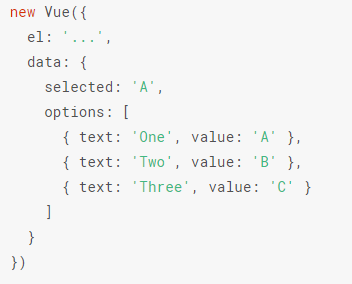






* **用 v-for 渲染的动态选项：**







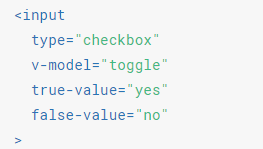
### 值绑定

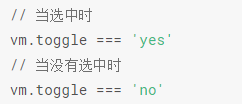
对于单选按钮，复选框及选择框的选项，v-model 绑定的值通常是静态字符串 (对于复选框也可以是布尔值)：



但是有时我们可能想把值绑定到 Vue 实例的一个动态属性上，这时可以用 v-bind 实现，并且这个属性的值可以不是字符串。

#### 复选框





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。

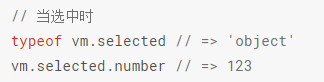
#### 单选按钮





#### 选择框的选项





### 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：



#### .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：



这通常很有用，因为即使在 type="number" 时，HTML 输入元素的值也总会返回字符串。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：



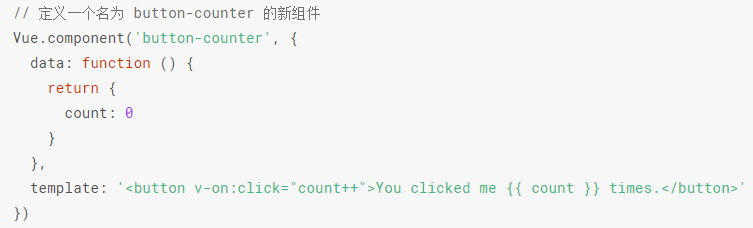
### 在组件上使用 v-model

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！

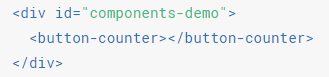
## 组件基础

### 基本示例

这里有一个 Vue 组件的示例：



组件是可复用的 Vue 实例，且带有一个名字：在这个例子中是 <button-counter>。我们可以在一个通过 new Vue 创建的 Vue 根实例中，把这个组件作为自定义元素来使用：



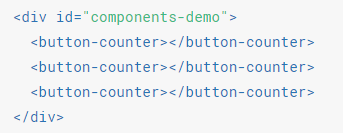




因为组件是可复用的 Vue 实例，所以它们与 new Vue 接收相同的选项，例如 data、computed、watch、methods 以及生命周期钩子等。仅有的例外是像 el 这样根实例特有的选项。

### 组件的复用

你可以将组件进行任意次数的复用：

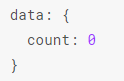




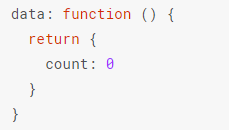
注意当点击按钮时，每个组件都会各自独立维护它的 count。因为你每用一次组件，就会有一个它的新实例被创建。

***data 必须是一个函数***

当我们定义这个 <button-counter> 组件时，你可能会发现它的 data 并不是像这样直接提供一个对象：



取而代之的是，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：

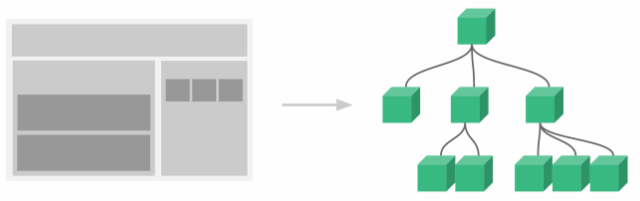


如果 Vue 没有这条规则，点击一个按钮就可能会像如下代码一样影响到其它所有实例：



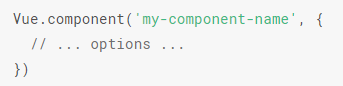
### 组件的组织

通常一个应用会以一棵嵌套的组件树的形式来组织：



例如，你可能会有页头、侧边栏、内容区等组件，每个组件又包含了其它的像导航链接、博文之类的组件。

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。至此，我们的组件都只是通过 Vue.component 全局注册的：



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

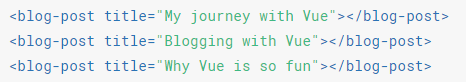
早些时候，我们提到了创建一个博文组件的事情。问题是如果你不能向这个组件传递某一篇博文的标题或内容之类的我们想展示的数据的话，它是没有办法使用的。这也正是 prop 的由来。

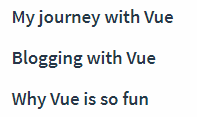
Prop 是你可以在组件上注册的一些自定义特性。当一个值传递给一个 prop 特性的时候，它就变成了那个组件实例的一个属性。为了给博文组件传递一个标题，我们可以用一个 props 选项将其包含在该组件可接受的 prop 列表中：



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。在上述模板中，你会发现我们能够在组件实例中访问这个值，就像访问 data 中的值一样。

一个 prop 被注册之后，你就可以像这样把数据作为一个自定义特性传递进来：





然而在一个典型的应用中，你可能在 data 里有一个博文的数组：



并想要为每篇博文渲染一个组件：



如上所示，你会发现我们可以使用 v-bind 来动态传递 prop。这在你一开始不清楚要渲染的具体内容，比如从一个 API 获取博文列表的时候，是非常有用的。

### 单个根元素

当构建一个 <blog-post> 组件时，你的模板最终会包含的东西远不止一个标题：



最最起码，你会包含这篇博文的正文：



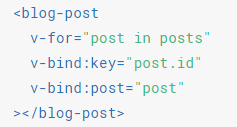
然而如果你在模板中尝试这样写，Vue 会显示一个错误，并解释道 every component must have a single root element (每个组件必须只有一个根元素)。你可以将模板的内容包裹在一个父元素内，来修复这个问题，例如：



看起来当组件变得越来越复杂的时候，我们的博文不只需要标题和内容，还需要发布日期、评论等等。为每个相关的信息定义一个 prop 会变得很麻烦：



所以是时候重构一下这个 <blog-post> 组件了，让它变成接受一个单独的 post prop：





上述的这个和一些接下来的示例使用了 JavaScript 的模板字符串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取而代之。

现在，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。

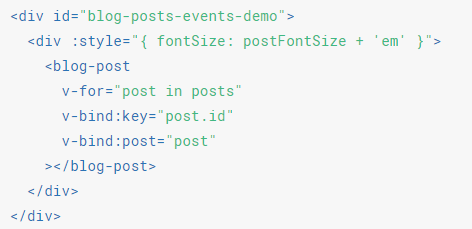
### 监听子组件事件

在我们开发 <blog-post> 组件时，它的一些功能可能要求我们和父级组件进行沟通。例如我们可能会引入一个辅助功能来放大博文的字号，同时让页面的其它部分保持默认的字号。

在其父组件中，我们可以通过添加一个 postFontSize 数据属性来支持这个功能：



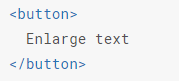
它可以在模板中用来控制所有博文的字号：



现在我们在每篇博文正文之前添加一个按钮来放大字号：



问题是这个按钮不会做任何事：



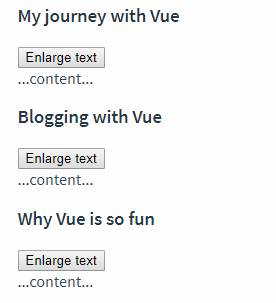
当点击这个按钮时，我们需要告诉父级组件放大所有博文的文本。幸好 Vue 实例提供了一个自定义事件的系统来解决这个问题。父级组件可以像处理 native DOM 事件一样通过 v-on 监听子组件实例的任意事件：



同时子组件可以通过调用内建的 $emit 方法 并传入事件名称来触发一个事件：



有了这个 v-on:enlarge-text="postFontSize += 0.1" 监听器，父级组件就会接收该事件并更新 postFontSize 的值。



* **使用事件抛出一个值**

有的时候用一个事件来抛出一个特定的值是非常有用的。例如我们可能想让 <blog-post> 组件决定它的文本要放大多少。这时可以使用 $emit 的第二个参数来提供这个值：



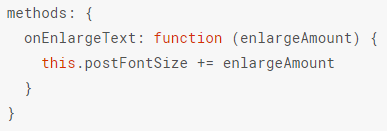
然后当在父级组件监听这个事件的时候，我们可以通过 $event 访问到被抛出的这个值：



或者，如果这个事件处理函数是一个方法：



那么这个值将会作为第一个参数传入这个方法：



* **在组件上使用 v-model**

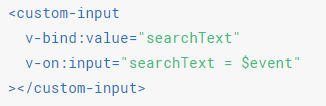
自定义事件也可以用于创建支持 v-model 的自定义输入组件。记住：



等价于：



当用在组件上时，v-model 则会这样：



为了让它正常工作，这个组件内的 <input> 必须：

* 将其 value 特性绑定到一个名叫 value 的 prop 上
* 在其 input 事件被触发时，将新的值通过自定义的 input 事件抛出

写成代码之后是这样的：

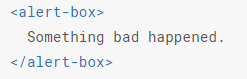


现在 v-model 就应该可以在这个组件上完美地工作起来了：

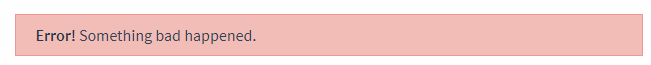


### 通过插槽分发内容

和 HTML 元素一样，我们经常需要向一个组件传递内容，像这样：



可能会渲染出这样的东西：



幸好，Vue 自定义的 <slot> 元素让这变得非常简单：



如你所见，我们只要在需要的地方加入插槽就行了——就这么简单！

### 动态组件

有的时候，在不同组件之间进行动态切换是非常有用的，比如在一个多标签的界面里：



上述内容可以通过 Vue 的 <component> 元素加一个特殊的 is 特性来实现：



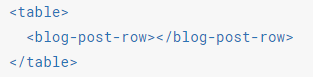
在上述示例中，currentTabComponent 可以包括

* 已注册组件的名字，或
* 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。

这会导致我们使用这些有约束条件的元素时遇到一些问题。例如：



这个自定义组件 <blog-post-row> 会被作为无效的内容提升到外部，并导致最终渲染结果出错。幸好这个特殊的 is 特性给了我们一个变通的办法：



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：

* 字符串 (例如：template: '...')
* 单文件组件 (.vue)
* <script type="text/x-template">

# 进阶

## 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。比如在全局注册的时候我们已经看到了：



该组件名就是 Vue.component 的第一个参数。

你给予组件的名字可能依赖于你打算拿它来做什么。当直接在 DOM 中使用一个组件 (而不是在字符串模板或单文件组件) 的时候，我们强烈推荐遵循 W3C 规范中的自定义组件名 (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。

定义组件名的方式有两种：

**使用 kebab-case**



当使用 kebab-case (短横线分隔命名) 定义一个组件时，你也必须在引用这个自定义元素时使用 kebab-case，例如 <my-component-name>。

**使用 PascalCase**



当使用 PascalCase (首字母大写命名) 定义一个组件时，你在引用这个自定义元素时两种命名法都可以使用。也就是说 <my-component-name> 和 <MyComponentName> 都是可接受的。注意，尽管如此，直接在 DOM (即非字符串的模板) 中使用时只有 kebab-case 是有效的。

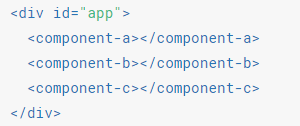
#### 全局注册

到目前为止，我们只用过 Vue.component 来创建组件：



这些组件是全局注册的。也就是说它们在注册之后可以用在任何新创建的 Vue 根实例 (new Vue) 的模板中。比如：



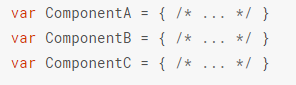


在所有子组件中也是如此，也就是说这三个组件在各自内部也都可以相互使用。

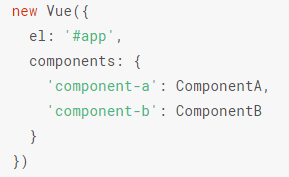
#### 局部注册

全局注册往往是不够理想的。比如，如果你使用一个像 webpack 这样的构建系统，全局注册所有的组件意味着即便你已经不再使用一个组件了，它仍然会被包含在你最终的构建结果中。这造成了用户下载的 JavaScript 的无谓的增加。

在这些情况下，你可以通过一个普通的 JavaScript 对象来定义组件：



然后在 components 选项中定义你想要使用的组件：



对于 components 对象中的每个属性来说，其属性名就是自定义元素的名字，其属性值就是这个组件的选项对象。

注意局部注册的组件在其子组件中不可用。例如，如果你希望 ComponentA 在 ComponentB 中可用，则你需要这样写：



或者如果你通过 Babel 和 webpack 使用 ES2015 模块，那么代码看起来更像：



注意在 ES2015+ 中，在对象中放一个类似 ComponentA 的变量名其实是 ComponentA: ComponentA 的缩写，即这个变量名同时是：

* 用在模板中的自定义元素的名称
* 包含了这个组件选项的变量名

#### 模块系统

***在模块系统中局部注册***

如果你还在阅读，说明你使用了诸如 Babel 和 webpack 的模块系统。在这些情况下，我们推荐创建一个 components 目录，并将每个组件放置在其各自的文件中。

然后你需要在局部注册之前导入每个你想使用的组件。例如，在一个假设的 ComponentB.js 或 ComponentB.vue 文件中：

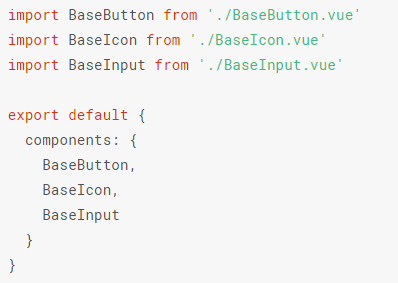


现在 ComponentA 和 ComponentC 都可以在 ComponentB 的模板中使用了。

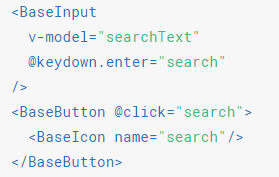
***基础组件的自动化全局注册***

可能你的许多组件只是包裹了一个输入框或按钮之类的元素，是相对通用的。我们有时候会把它们称为基础组件，它们会在各个组件中被频繁的用到。

所以会导致很多组件里都会有一个包含基础组件的长列表：



而只是用于模板中的一小部分：



幸好如果你使用了 webpack (或在内部使用了 webpack 的 Vue CLI 3+)，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。这里有一份可以让你在应用入口文件 (比如 src/main.js) 中全局导入基础组件的示例代码：



记住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 中的属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：





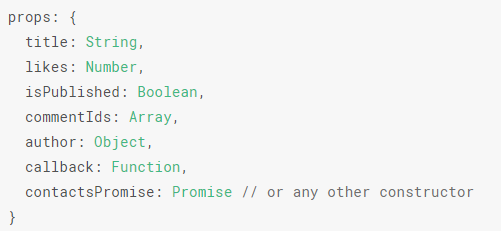
重申一次，如果你使用字符串模板，那么这个限制就不存在了。

#### Prop 类型

到这里，我们只看到了以字符串数组形式列出的 prop：



但是，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：



这不仅为你的组件提供了文档，还会在它们遇到错误的类型时从浏览器的 JavaScript 控制台提示用户。

#### 传递静态或动态 Prop

像这样，你已经知道了可以像这样给 prop 传入一个静态的值：



你也知道 prop 可以通过 v-bind 动态赋值，例如：



在上述两个示例中，我们传入的值都是字符串类型的，但实际上任何类型的值都可以传给一个 prop。

数字：



布尔值：



数组：

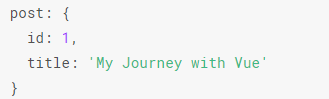


对象：



对象的属性：

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：





等价于：



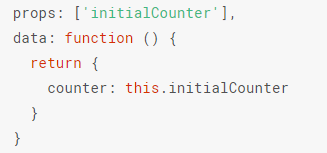
#### 单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

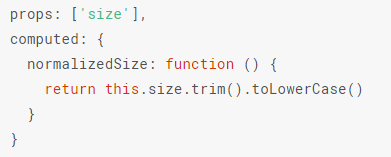
额外的，每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。

这里有两种常见的试图改变一个 prop 的情形：

1. 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：



1. 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：



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

我们可以为组件的 prop 指定验证要求，例如你知道的这些类型。如果有一个需求没有被满足，则 Vue 会在浏览器控制台中警告你。这在开发一个会被别人用到的组件时尤其有帮助。

为了定制 prop 的验证方式，你可以为 props 中的值提供一个带有验证需求的对象，而不是一个字符串数组。例如：



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。

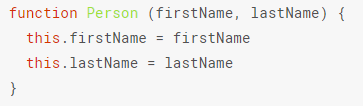
注意那些 prop 会在一个组件实例创建之前进行验证，所以实例的属性 (如 data、computed 等) 在 default 或 validator 函数中是不可用的。

***类型检查***

type 可以是下列原生构造函数中的一个：

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

额外的，type 还可以是一个自定义的构造函数，并且通过 instanceof 来进行检查确认。例如，给定下列现成的构造函数：



你可以使用：



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

#### 非 Prop 的特性

一个非 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。

因为显式定义的 prop 适用于向一个子组件传入信息，然而组件库的作者并不总能预见组件会被用于怎样的场景。这也是为什么组件可以接受任意的特性，而这些特性会被添加到这个组件的根元素上。

例如，想象一下你通过一个 Bootstrap 插件使用了一个第三方的 <bootstrap-date-input> 组件，这个插件需要在其 <input> 上用到一个 data-date-picker 特性。我们可以将这个特性添加到你的组件实例上：



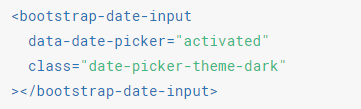
然后这个 data-date-picker="activated" 特性就会自动添加到 <bootstrap-date-input> 的根元素上。

***替换/合并已有的特性***

想象一下 <bootstrap-date-input> 的模板是这样的：



为了给我们的日期选择器插件定制一个主题，我们可能需要像这样添加一个特别的类名：



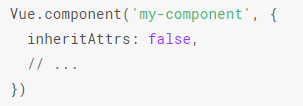
在这种情况下，我们定义了两个不同的 class 的值：

* form-control，这是在组件的模板内设置好的
* date-picker-theme-dark，这是从组件的父级传入的

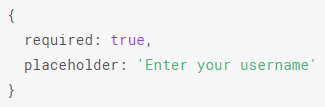
对于绝大多数特性来说，从外部提供给组件的值会替换掉组件内部设置好的值。所以如果传入 type="text" 就会替换掉 type="date" 并把它破坏！庆幸的是，class 和 style 特性会稍微智能一些，即两边的值会被合并起来，从而得到最终的值：form-control date-picker-theme-dark。

***禁用特性继承***

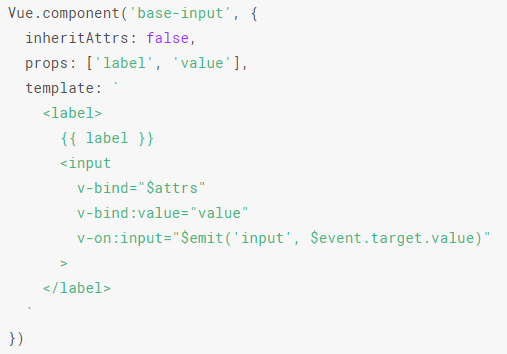
如果你不希望组件的根元素继承特性，你可以在组件的选项中设置 inheritAttrs: false。例如：



这尤其适合配合实例的 $attrs 属性使用，该属性包含了传递给一个组件的特性名和特性值，例如：



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。在撰写基础组件的时候是常会用到的：



注意 inheritAttrs: false 选项不会影响 style 和 class 的绑定。

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：

