Vue.js https://cn.vuejs.org/v2/guide/installation.html

组件和for组件 v-model与组件 总结完了回头看

Vue为 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。
* v-model
* **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

**组织架构：**

// 里面的this指向app5实例

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(){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监听数据改变的对象的属性名

数据名：function(){}

}, 就是监听的数据名

components: {

//可以声明一个只能父级调用的 前提是在全局有个Child组件 如下实例2

// <my-component> 将只在父模板可用

'my-component': Child

}

})

Component 声明一个新组件

Vue.component('todo-item', {

props: ['todo'], //这是向子组件通信的

template: '<li>{{ todo.text }}</li>' //写组件结构

//这里有data的话这里的data 必须是个函数

})

实例2

var Child = {

template: '<div>A custom component!</div>'

}

new Vue({

// ...

components: {

// <my-component> 将只在父模板可用

'my-component': Child

}

})

要注册一个全局组件，你可以使用 Vue.component(tagName, options)。 例如：

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

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。

# Vue 实例

## 构造器

每个 Vue.js 应用都是通过构造函数 Vue 创建一个 **Vue 的根实例** 启动的：

|  |
| --- |
| var vm = new Vue({// 选项}) |

虽然没有完全遵循 [MVVM 模式](https://en.wikipedia.org/wiki/Model_View_ViewModel)， Vue 的设计无疑受到了它的启发。因此在文档中经常会使用 vm (ViewModel 的简称) 这个变量名表示 Vue 实例。

在实例化 Vue 时，需要传入一个**选项对象**，它可以包含数据、模板、挂载元素、方法、生命周期钩子等选项。全部的选项可以在 [API 文档](https://cn.vuejs.org/v2/api)中查看。

可以扩展 Vue 构造器，从而用预定义选项创建可复用的**组件构造器**：

|  |
| --- |
| var MyComponent = Vue.extend({  // 扩展选项  })  // 所有的 `MyComponent` 实例都将以预定义的扩展选项被创建  var myComponentInstance = new MyComponent() |

尽管可以命令式地创建扩展实例，不过在多数情况下建议将组件构造器注册为一个自定义元素，然后声明式地用在模板中。我们将在后面详细说明[组件系统](https://cn.vuejs.org/guide/components.html)。现在你只需知道所有的 Vue.js 组件其实都是被扩展的 Vue 实例。

## 属性与方法

每个 Vue 实例都会**代理**其 data 对象里所有的属性：

|  |
| --- |
| var data = { a: 1 }  var vm = new Vue({  data: data  })  vm.a === data.a // -> true  // 设置属性也会影响到原始数据  vm.a = 2  data.a // -> 2  // ... 反之亦然  data.a = 3  vm.a // -> 3 |

注意只有这些被代理的属性是**响应的**。如果在实例创建之后添加新的属性到实例上，它不会触发视图更新。我们将在后面详细讨论响应系统。

除了 data 属性， Vue 实例暴露了一些有用的实例属性与方法。这些属性与方法都有前缀 $，以便与代理的 data 属性区分。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 (newVal, oldVal) {  // 这个回调将在 `vm.a` 改变后调用  }) |

注意，不要在实例属性或者回调函数中（如 vm.$watch('a', newVal => this.myMethod())）使用[箭头函数](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions)。因为箭头函数绑定父上下文，所以 this 不会像预想的一样是 Vue 实例，而是 this.myMethod 未被定义。

实例属性和方法的完整列表中查阅 [API 参考](https://cn.vuejs.org/v2/api)。

## 实例生命周期

每个 Vue 实例在被创建之前都要经过一系列的初始化过程。例如，实例需要配置数据观测(data observer)、编译模版、挂载实例到 DOM ，然后在数据变化时更新 DOM 。在这个过程中，实例也会调用一些 **生命周期钩子** ，这就给我们提供了执行自定义逻辑的机会。例如，created 这个钩子在实例被创建之后被调用：

|  |
| --- |
| var vm = 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调用，如 mounted、 updated 、destroyed 。钩子的 this 指向调用它的 Vue 实例。一些用户可能会问 Vue.js 是否有“控制器”的概念？答案是，没有。组件的自定义逻辑可以分布在这些钩子中。

## 生命周期图示

下图说明了实例的生命周期。你不需要立马弄明白所有的东西，不过以后它会有帮助。

# 模板语法

Vue.js 使用了基于 HTML 的模版语法，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，在应用状态改变时， Vue 能够智能地计算出重新渲染组件的最小代价并应用到 DOM 操作上。

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](https://cn.vuejs.org/v2/guide/render-function.html)，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://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> |

### 纯 HTML

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。

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)。请只对可信内容使用 HTML 插值，**绝不要**对用户提供的内容插值。

### 属性

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)：

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

这对布尔值的属性也有效 —— 如果条件被求值为 false 的话该属性会被移除：

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

### 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，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**单一 JavaScript 表达式**（除了 v-for，之后再讨论）。指令的职责就是当其表达式的值改变时相应地将某些行为应用到 DOM 上。让我们回顾一下在介绍里的例子：

|  |
| --- |
| <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

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) 。

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 阻止单击事件冒泡
* .prevent 提交事件不再重载页面
* .capture 添加事件侦听器时使用事件捕获模式
* .self 只当事件在该元素本身（而不是子元素）触发时触发回调
* .once 事件将只会触发一次

|  |
| --- |
| <!-- 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#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/#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。

# 表单控件绑定

## 基础用法

你可以用 v-model 指令在表单控件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，并特别处理一些极端的例子。

v-model 并不关心表单控件初始化所生成的值。因为它会选择 Vue 实例数据来作为具体的值。

对于要求 IME （如中文、 日语、 韩语等） 的语言，你会发现那v-model不会在 ime 构成中得到更新。如果你也想实现更新，请使用 input事件。

### 文本

|  |
| --- |
| <input v-model="message" placeholder="edit me">  <p>Message is: {{ message }}</p> |

### 多行文本

|  |
| --- |
| <span>Multiline message is:</span>  <p style="white-space: pre">{{ message }}</p>  <br>  <textarea v-model="message" placeholder="add multiple lines"></textarea> |

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

### 复选框

单个勾选框，逻辑值：

|  |
| --- |
| <input type="checkbox" id="checkbox" v-model="checked">  <label for="checkbox">{{ checked }}</label> |

多个勾选框，绑定到同一个数组：

|  |
| --- |
| <input type="checkbox" id="jack" value="Jack" v-model="checkedNames">  <label for="jack">Jack</label>  <input type="checkbox" id="john" value="John" v-model="checkedNames">  <label for="john">John</label>  <input type="checkbox" id="mike" value="Mike" v-model="checkedNames">  <label for="mike">Mike</label>  <br>  <span>Checked names: {{ checkedNames }}</span> |

|  |
| --- |
| new Vue({  el: '...',  data: {  checkedNames: []  }  }) |

### 单选按钮

|  |
| --- |
| <div id="example-4" class="demo">  <input type="radio" id="one" value="One" v-model="picked">  <label for="one">One</label>  <br>  <input type="radio" id="two" value="Two" v-model="picked">  <label for="two">Two</label>  <br>  <span>Picked: {{ picked }}</span>  </div> |

|  |
| --- |
| new Vue({  el: '#example-4',  data: {  picked: ''  }  }) |

### 选择列表

单选列表:

|  |
| --- |
| <div id="example-5" class="demo">  <select v-model="selected">  <option>A</option>  <option>B</option>  <option>C</option>  </select>  <span>Selected: {{ selected }}</span>  </div> |

|  |
| --- |
| new Vue({  el: '#example-5',  data: {  selected: null  }  }) |

多选列表（绑定到一个数组）：

|  |
| --- |
| <div id="example-6" class="demo">  <select v-model="selected" multiple style="width: 50px">  <option>A</option>  <option>B</option>  <option>C</option>  </select>  <br>  <span>Selected: {{ selected }}</span>  </div> |

|  |
| --- |
| new Vue({  el: '#example-6',  data: {  selected: []  }  }) |

动态选项，用 v-for 渲染：

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

|  |
| --- |
| new Vue({  el: '...',  data: {  selected: 'A',  options: [  { text: 'One', value: 'A' },  { text: 'Two', value: 'B' },  { text: 'Three', value: 'C' }  ]  }  }) |

## 绑定 value

对于单选按钮，勾选框及选择列表选项， v-model 绑定的 value 通常是静态字符串（对于勾选框是逻辑值）：

|  |
| --- |
| <!-- 当选中时，`picked` 为字符串 "a" -->  <input type="radio" v-model="picked" value="a">  <!-- `toggle` 为 true 或 false -->  <input type="checkbox" v-model="toggle">  <!-- 当选中时，`selected` 为字符串 "abc" -->  <select v-model="selected">  <option value="abc">ABC</option>  </select> |

但是有时我们想绑定 value 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

|  |
| --- |
| <input type="checkbox" v-model="toggle" v-bind:true-value="a" v-bind:false-value="b"> |

|  |
| --- |
| // 当选中时vm.toggle === vm.a  // 当没有选中时vm.toggle === vm.b |

### 单选按钮

|  |
| --- |
| <input type="radio" v-model="pick" v-bind:value="a">// 当选中时vm.pick === vm.a |

|  |
| --- |
|  |

### 选择列表设置

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

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

## 修饰符

### .lazy

在默认情况下， v-model 在 input 事件中同步输入框的值与数据 (除了 [上述](https://cn.vuejs.org/v2/guide/forms.html#vmodel-ime-tip) IME 部分)，但你可以添加一个修饰符lazy ，从而转变为在 change 事件中同步：

|  |
| --- |
| <!-- 在 "change" 而不是 "input" 事件中更新 --> <input v-model.lazy="msg" > |

### .number

如果想自动将用户的输入值转为 Number 类型（如果原值的转换结果为 NaN 则返回原值），可以添加一个修饰符 number给 v-model 来处理输入值：

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

这通常很有用，因为在 type="number" 时 HTML 中输入的值也总是会返回字符串类型。

### .trim

如果要自动过滤用户输入的首尾空格，可以添加 trim 修饰符到 v-model 上过滤输入：

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

## v-model 与组件

**如果你还不熟悉Vue的组件，跳过这里即可。**

HTML 内建的 input 类型有时不能满足你的需求。还好，Vue 的组件系统允许你创建一个具有自定义行为可复用的 input 类型，这些 input 类型甚至可以和 v-model 一起使用！

# 组件

## 什么是组件？

组件（Component）是 Vue.js 最强大的功能之一。组件可以扩展 HTML 元素，封装可重用的代码。在较高层面上，组件是自定义元素， Vue.js 的编译器为它添加特殊功能。在有些情况下，组件也可以是原生 HTML 元素的形式，以 js 特性扩展。

## 使用组件

### 注册

之前说过，我们可以通过以下方式创建一个 Vue 实例：

|  |
| --- |
| new Vue({ el: '#some-element',// 选项}) |

要注册一个全局组件，你可以使用 Vue.component(tagName, options)。 例如：

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

对于自定义标签名，Vue.js 不强制要求遵循 [W3C规则](https://www.w3.org/TR/custom-elements/#concepts) （小写，并且包含一个短杠），尽管遵循这个规则比较好。

组件在注册之后，便可以在父实例的模块中以自定义元素 <my-component></my-component> 的形式使用。要确保在初始化根实例 **之前** 注册了组件：

|  |
| --- |
| <div id="example">  <my-component></my-component>  </div> |

|  |
| --- |
| // 注册  Vue.component('my-component', {  template: '<div>A custom component!</div>'  })  // 创建根实例  new Vue({  el: '#example'  }) |

渲染为：

|  |
| --- |
| <div id="example">  <div>A custom component!</div>  </div> |

A custom component!

### 局部注册

不必在全局注册每个组件。通过使用组件实例选项注册，可以使组件仅在另一个实例/组件的作用域中可用：

|  |
| --- |
| var Child = {  template: '<div>A custom component!</div>'  }  new Vue({  // ...  components: {  // <my-component> 将只在父模板可用  'my-component': Child  }  }) |

这种封装也适用于其它可注册的 Vue 功能，如指令。

### DOM 模版解析说明

当使用 DOM 作为模版时（例如，将 el 选项挂载到一个已存在的元素上）, 你会受到 HTML 的一些限制，因为 Vue 只有在浏览器解析和标准化 HTML 后才能获取模版内容。尤其像这些元素 <ul> ， <ol>， <table> ， <select> 限制了能被它包裹的元素， <option> 只能出现在其它元素内部。

在自定义组件中使用这些受限制的元素时会导致一些问题，例如：

|  |
| --- |
| <table>  <my-row>...</my-row>  </table> |

自定义组件 <my-row> 被认为是无效的内容，因此在渲染的时候会导致错误。变通的方案是使用特殊的 is 属性：

|  |
| --- |
| <table>  <tr is="my-row"></tr>  </table> |

**应当注意，如果您使用来自以下来源之一的字符串模板，这些限制将不适用：**

* <script type="text/x-template">
* JavaScript内联模版字符串
* .vue 组件

因此，有必要的话请使用字符串模版。

### data 必须是函数

通过Vue构造器传入的各种选项大多数都可以在组件里用。 data 是一个例外，它必须是函数。 实际上，如果你这么做：

|  |
| --- |
| Vue.component('my-component', {  template: '<span>{{ message }}</span>',  data: {  message: 'hello'  }  }) |

那么 Vue 会停止，并在控制台发出警告，告诉你在组件中 data 必须是一个函数。理解这种规则的存在意义很有帮助，让我们假设用如下方式来绕开Vue的警告：

|  |
| --- |
| <div id="example-2">  <simple-counter></simple-counter>  <simple-counter></simple-counter>  <simple-counter></simple-counter>  </div> |

|  |
| --- |
| var data = { counter: 0 }  Vue.component('simple-counter', {  template: '<button v-on:click="counter += 1">{{ counter }}</button>',   // 技术上 data 的确是一个函数了，因此 Vue 不会警告，   // 但是我们返回给每个组件的实例的却引用了同一个data对象   data: function () {  return data  }  })  new Vue({  el: '#example-2'  }) |

0 0 0

由于这三个组件共享了同一个 data ， 因此增加一个 counter 会影响所有组件！这不对。我们可以通过为每个组件返回全新的 data 对象来解决这个问题：

|  |
| --- |
| data: function () {  return {  counter: 0  }  } |

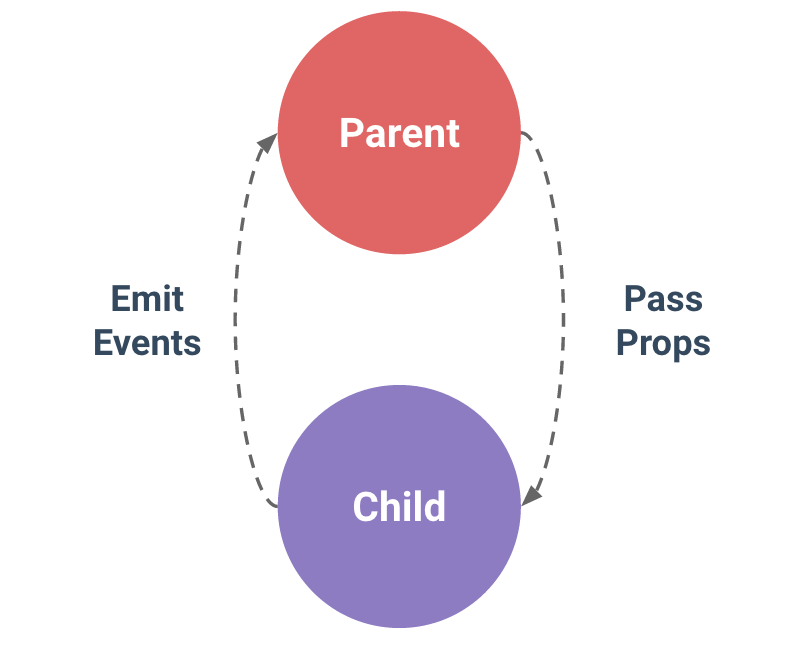
现在每个 counter 都有它自己内部的状态了：

0 0 0

### 构成组件

组件意味着协同工作，通常父子组件会是这样的关系：组件 A 在它的模版中使用了组件 B 。它们之间必然需要相互通信：父组件要给子组件传递数据，子组件需要将它内部发生的事情告知给父组件。然而，在一个良好定义的接口中尽可能将父子组件解耦是很重要的。这保证了每个组件可以在相对隔离的环境中书写和理解，也大幅提高了组件的可维护性和可重用性。

在 Vue.js 中，父子组件的关系可以总结为 **props down, events up** 。父组件通过 **props** 向下传递数据给子组件，子组件通过 **events** 给父组件发送消息。看看它们是怎么工作的。



## Prop

### 使用 Prop 传递数据

组件实例的作用域是**孤立的**。这意味着不能(也不应该)在子组件的模板内直接引用父组件的数据。要让子组件使用父组件的数据，我们需要通过子组件的props选项。

子组件要显式地用 [props 选项](https://cn.vuejs.org/v2/api/#props)声明它期待获得的数据：

|  |
| --- |
| Vue.component('child', {  // 声明 props  props: ['message'],  // 就像 data 一样，prop 可以用在模板内  // 同样也可以在 vm 实例中像 “this.message” 这样使用  template: '<span>{{ message }}</span>'  }) |

然后我们可以这样向它传入一个普通字符串：

|  |
| --- |
| <child message="hello!"></child> |

结果：

hello!

### camelCase vs. kebab-case

HTML 特性是不区分大小写的。所以，当使用的不是字符串模版，camelCased (驼峰式) 命名的 prop 需要转换为相对应的 kebab-case (短横线隔开式) 命名：

|  |
| --- |
| Vue.component('child', {  // camelCase in JavaScript  props: ['myMessage'],  template: '<span>{{ myMessage }}</span>'  }) |

|  |
| --- |
| <!-- kebab-case in HTML -->  <child my-message="hello!"></child> |

如果你使用字符串模版，则没有这些限制。

### 动态 Prop

在模板中，要动态地绑定父组件的数据到子模板的props，与绑定到任何普通的HTML特性相类似，就是用 v-bind。每当父组件的数据变化时，该变化也会传导给子组件：

|  |
| --- |
| <div>  <input v-model="parentMsg">  <br>  <child v-bind:my-message="parentMsg"></child>  </div> |

使用 v-bind 的缩写语法通常更简单：

|  |
| --- |
| <child :my-message="parentMsg"></child> |

结果：

   
Message from parent

### 字面量语法 vs 动态语法

初学者常犯的一个错误是使用字面量语法传递数值：

|  |
| --- |
| <!-- 传递了一个字符串"1" -->  <comp some-prop="1"></comp> |

因为它是一个字面 prop ，它的值是字符串 "1" 而不是number。如果想传递一个实际的number，需要使用 v-bind ，从而让它的值被当作 JavaScript 表达式计算：

|  |
| --- |
| <!-- 传递实际的mumber -->  <comp v-bind:some-prop="1"></comp> |

### 单向数据流

prop 是单向绑定的：当父组件的属性变化时，将传导给子组件，但是不会反过来。这是为了防止子组件无意修改了父组件的状态——这会让应用的数据流难以理解。

另外，每次父组件更新时，子组件的所有 prop 都会更新为最新值。这意味着你**不应该**在子组件内部改变 prop 。如果你这么做了，Vue 会在控制台给出警告。

为什么我们会有修改prop中数据的冲动呢？通常是这两种原因：

1. prop 作为初始值传入后，子组件想把它当作局部数据来用；
2. prop 作为初始值传入，由子组件处理成其它数据输出。

对这两种原因，正确的应对方式是：

1. 定义一个局部变量，并用 prop 的值初始化它：

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

1. 定义一个计算属性，处理 prop 的值并返回。

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

注意在 JavaScript 中对象和数组是引用类型，指向同一个内存空间，如果 prop 是一个对象或数组，在子组件内部改变它**会影响**父组件的状态。

### Prop 验证

我们可以为组件的 props 指定验证规格。如果传入的数据不符合规格，Vue 会发出警告。当组件给其他人使用时，这很有用。

要指定验证规格，需要用对象的形式，而不能用字符串数组：

|  |
| --- |
| Vue.component('example', {  props: {  // 基础类型检测 （`null` 意思是任何类型都可以）  propA: Number,  // 多种类型  propB: [String, Number],  // 必传且是字符串  propC: {  type: String,  required: true  },  // 数字，有默认值  propD: {  type: Number,  default: 100  },  // 数组／对象的默认值应当由一个工厂函数返回  propE: {  type: Object,  default: function () {  return { message: 'hello' }  }  },  // 自定义验证函数  propF: {  validator: function (value) {  return value > 10  }  }  }  }) |

type 可以是下面原生构造器：

* String
* Number
* Boolean
* Function
* Object
* Array

type 也可以是一个自定义构造器函数，使用 instanceof 检测。

当 prop 验证失败，Vue会在抛出警告 (如果使用的是开发版本)。

## 自定义事件

我们知道，父组件是使用 props 传递数据给子组件，但如果子组件要把数据传递回去，应该怎样做？那就是自定义事件！

### 使用 v-on 绑定自定义事件

每个 Vue 实例都实现了[事件接口(Events interface)](https://cn.vuejs.org/v2/api/#Instance-Methods-Events)，即：

* 使用 $on(eventName) 监听事件
* 使用 $emit(eventName) 触发事件

Vue的事件系统分离自浏览器的[EventTarget API](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget)。尽管它们的运行类似，但是$on 和 $emit **不是**addEventListener 和 dispatchEvent 的别名。

另外，父组件可以在使用子组件的地方直接用 v-on 来监听子组件触发的事件。

不能用$on侦听子组件抛出的事件，而必须在模板里直接用v-on绑定，就像以下的例子：

下面是一个例子：

|  |
| --- |
| <div id="counter-event-example">  <p>{{ total }}</p>  <button-counter v-on:increment="incrementTotal"></button-counter>  <button-counter v-on:increment="incrementTotal"></button-counter>  </div> |

|  |
| --- |
| Vue.component('button-counter', {  template: '<button v-on:click="increment">{{ counter }}</button>',  data: function () {  return {  counter: 0  }  },  methods: {  increment: function () {  this.counter += 1  this.$emit('increment')  }  },  })  new Vue({  el: '#counter-event-example',  data: {  total: 0  },  methods: {  incrementTotal: function () {  this.total += 1  }  }  }) |

0

0 0

在本例中，子组件已经和它外部完全解耦了。它所做的只是报告自己的内部事件，至于父组件是否关心则与它无关。留意到这一点很重要。

#### 给组件绑定原生事件

有时候，你可能想在某个组件的根元素上监听一个原生事件。可以使用 .native 修饰 v-on 。例如：

|  |
| --- |
| <my-component v-on:click.native="doTheThing"></my-component> |

### 使用自定义事件的表单输入组件

自定义事件可以用来创建自定义的表单输入组件，使用 v-model 来进行数据双向绑定。看看这个：

|  |
| --- |
| <input v-model="something"> |

这不过是以下示例的语法糖：

|  |
| --- |
| <input v-bind:value="something" v-on:input="something = $event.target.value"> |

所以在组件中使用时，它相当于下面的简写：

|  |
| --- |
| <custom-input v-bind:value="something" v-on:input="something = arguments[0]"></custom-input> |

所以要让组件的 v-model 生效，它必须：

* 接受一个 value 属性
* 在有新的 value 时触发 input 事件

我们来看一个非常简单的货币输入的自定义控件：

|  |
| --- |
| <currency-input v-model="price"></currency-input> |

|  |
| --- |
| Vue.component('currency-input', {  template: '\  <span>\  $\  <input\  ref="input"\  v-bind:value="value"\  v-on:input="updateValue($event.target.value)"\  >\  </span>\  ',  props: ['value'],  methods: {  // 不是直接更新值，而是使用此方法来对输入值进行格式化和位数限制  updateValue: function (value) {  var formattedValue = value  // 删除两侧的空格符  .trim()  // 保留 2 小数位  .slice(0, value.indexOf('.') + 3)  // 如果值不统一，手动覆盖以保持一致  if (formattedValue !== value) {  this.$refs.input.value = formattedValue  }  // 通过 input 事件发出数值  this.$emit('input', Number(formattedValue))  }  }  }) |

$ 

当然，上面的例子是比较幼稚的。 比如，用户甚至可以输入多个小数点或句号 - 哦哦！因此我们需要一个更有意义的例子，下面是一个更加完善的货币过滤器：

事件接口不仅仅可以用来连接组件内部的表单输入，也很容易集成你自己创造的输入类型。想象一下：

|  |
| --- |
| <voice-recognizer v-model="question"></voice-recognizer>  <webcam-gesture-reader v-model="gesture"></webcam-gesture-reader>  <webcam-retinal-scanner v-model="retinalImage"></webcam-retinal-scanner> |

### 非父子组件通信

有时候两个组件也需要通信(非父子关系)。在简单的场景下，可以使用一个空的 Vue 实例作为中央事件总线：

|  |
| --- |
| var bus = new Vue() |

|  |
| --- |
| // 触发组件 A 中的事件  bus.$emit('id-selected', 1) |

|  |
| --- |
| // 在组件 B 创建的钩子中监听事件  bus.$on('id-selected', function (id) {  // ...  }) |

在复杂的情况下，我们应该考虑使用专门的 [状态管理模式](https://cn.vuejs.org/v2/guide/state-management.html).

## 使用 Slot 分发内容

在使用组件时，我们常常要像这样组合它们：

|  |
| --- |
| <app>  <app-header></app-header>  <app-footer></app-footer>  </app> |

注意两点：

1. <app> 组件不知道它的挂载点会有什么内容。挂载点的内容是由<app>的父组件决定的。
2. <app> 组件很可能有它自己的模版。

为了让组件可以组合，我们需要一种方式来混合父组件的内容与子组件自己的模板。这个过程被称为 **内容分发** (或 “transclusion” 如果你熟悉 Angular)。Vue.js 实现了一个内容分发 API ，参照了当前 [Web 组件规范草案](https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Slots-Proposal.md)，使用特殊的<slot> 元素作为原始内容的插槽。

### 编译作用域

在深入内容分发 API 之前，我们先明确内容在哪个作用域里编译。假定模板为：

|  |
| --- |
| <child-component>  {{ message }}  </child-component> |

message 应该绑定到父组件的数据，还是绑定到子组件的数据？答案是父组件。组件作用域简单地说是：

父组件模板的内容在父组件作用域内编译；子组件模板的内容在子组件作用域内编译。

一个常见错误是试图在父组件模板内将一个指令绑定到子组件的属性/方法：

|  |
| --- |
| <!-- 无效 -->  <child-component v-show="someChildProperty"></child-component> |

假定 someChildProperty 是子组件的属性，上例不会如预期那样工作。父组件模板不应该知道子组件的状态。

如果要绑定作用域内的指令到一个组件的根节点，你应当在组件自己的模板上做：

|  |
| --- |
| Vue.component('child-component', {   // 有效，因为是在正确的作用域内  template: '<div v-show="someChildProperty">Child</div>',  data: function () {  return {  someChildProperty: true  }  }  }) |

类似地，分发内容是在父作用域内编译。

### 单个 Slot

除非子组件模板包含至少一个 <slot> 插口，否则父组件的内容将会被**丢弃**。当子组件模板只有一个没有属性的 slot 时，父组件整个内容片段将插入到 slot 所在的 DOM 位置，并替换掉 slot 标签本身。

最初在 <slot> 标签中的任何内容都被视为**备用内容**。备用内容在子组件的作用域内编译，并且只有在宿主元素为空，且没有要插入的内容时才显示备用内容。

假定 my-component 组件有下面模板：

|  |
| --- |
| <div>  <h2>我是子组件的标题</h2>  <slot>  只有在没有要分发的内容时才会显示。  </slot>  </div> |

父组件模版：

|  |
| --- |
| <div>  <h1>我是父组件的标题</h1>  <my-component>  <p>这是一些初始内容</p>  <p>这是更多的初始内容</p>  </my-component>  </div> |

渲染结果：

|  |
| --- |
| <div>  <h1>我是父组件的标题</h1>  <div>  <h2>我是子组件的标题</h2>  <p>这是一些初始内容</p>  <p>这是更多的初始内容</p>  </div>  </div> |

### 具名 Slot

<slot> 元素可以用一个特殊的属性 name 来配置如何分发内容。多个 slot 可以有不同的名字。具名 slot 将匹配内容片段中有对应 slot 特性的元素。

仍然可以有一个匿名 slot ，它是**默认 slot** ，作为找不到匹配的内容片段的备用插槽。如果没有默认的 slot ，这些找不到匹配的内容片段将被抛弃。

例如，假定我们有一个 app-layout 组件，它的模板为：

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

父组件模版：

|  |
| --- |
| <app-layout>  <h1 slot="header">这里可能是一个页面标题</h1>  <p>主要内容的一个段落。</p>  <p>另一个主要段落。</p>  <p slot="footer">这里有一些联系信息</p>  </app-layout> |

渲染结果为：

|  |
| --- |
| <div class="container">  <header>  <h1>这里可能是一个页面标题</h1>  </header>  <main>  <p>主要内容的一个段落。</p>  <p>另一个主要段落。</p>  </main>  <footer>  <p>这里有一些联系信息</p>  </footer>  </div> |

在组合组件时，内容分发 API 是非常有用的机制。

### 作用域插槽

**2.1.0 新增**

作用域插槽是一种特殊类型的插槽，用作使用一个（能够传递数据到）可重用模板替换已渲染元素。

在子组件中，只需将数据传递到插槽，就像你将 prop 传递给组件一样：

|  |
| --- |
| <div class="child">  <slot text="hello from child"></slot>  </div> |

在父级中，具有特殊属性 scope 的 <template> 元素，表示它是作用域插槽的模板。scope 的值对应一个临时变量名，此变量接收从子组件中传递的 prop 对象：

|  |
| --- |
| <div class="parent">  <child>  <template scope="props">  <span>hello from parent</span>  <span>{{ props.text }}</span>  </template>  </child>  </div> |

如果我们渲染以上结果，得到的输出会是：

|  |
| --- |
| <div class="parent">  <div class="child">  <span>hello from parent</span>  <span>hello from child</span>  </div>  </div> |

作用域插槽更具代表性的用例是列表组件，允许组件自定义应该如何渲染列表每一项：

|  |
| --- |
| <my-awesome-list :items="items">  <!-- 作用域插槽也可以在这里命名 -->  <template slot="item" scope="props">  <li class="my-fancy-item">{{ props.text }}</li>  </template>  </my-awesome-list> |

列表组件的模板：

|  |
| --- |
| <ul>  <slot name="item"  v-for="item in items"  :text="item.text">  <!-- fallback content here -->  </slot>  </ul> |

## 动态组件

通过使用保留的 <component> 元素，动态地绑定到它的 is 特性，我们让多个组件可以使用同一个挂载点，并动态切换：

|  |
| --- |
| var vm = new Vue({  el: '#example',  data: {  currentView: 'home'  },  components: {  home: { /\* ... \*/ },  posts: { /\* ... \*/ },  archive: { /\* ... \*/ }  }  }) |

|  |
| --- |
| <component v-bind:is="currentView">  <!-- 组件在 vm.currentview 变化时改变！ -->  </component> |

也可以直接绑定到组件对象上：

|  |
| --- |
| var Home = {  template: '<p>Welcome home!</p>'  }  var vm = new Vue({  el: '#example',  data: {  currentView: Home  }  }) |

### keep-alive

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。为此可以添加一个 keep-alive 指令参数：

|  |
| --- |
| <keep-alive>  <component :is="currentView">  <!-- 非活动组件将被缓存！ -->  </component>  </keep-alive> |

在[API 参考](https://cn.vuejs.org/v2/api/#keep-alive)查看更多 <keep-alive> 的细节。

## 杂项

### 编写可复用组件

在编写组件时，记住是否要复用组件有好处。一次性组件跟其它组件紧密耦合没关系，但是可复用组件应当定义一个清晰的公开接口。

Vue 组件的 API 来自三部分 - props, events 和 slots ：

* **Props** 允许外部环境传递数据给组件
* **Events** 允许组件触发外部环境的副作用
* **Slots** 允许外部环境将额外的内容组合在组件中。

使用 v-bind 和 v-on 的简写语法，模板的缩进清楚且简洁：

|  |
| --- |
| <my-component  :foo="baz"  :bar="qux"  @event-a="doThis"  @event-b="doThat"  >  <img slot="icon" src="...">  <p slot="main-text">Hello!</p>  </my-component> |

### 子组件索引

尽管有 props 和 events ，但是有时仍然需要在 JavaScript 中直接访问子组件。为此可以使用 ref 为子组件指定一个索引 ID 。例如：

|  |
| --- |
| <div id="parent">  <user-profile ref="profile"></user-profile>  </div> |

|  |
| --- |
| var parent = new Vue({ el: '#parent' })  // 访问子组件  var child = parent.$refs.profile |

当 ref 和 v-for 一起使用时， ref 是一个数组或对象，包含相应的子组件。

$refs 只在组件渲染完成后才填充，并且它是非响应式的。它仅仅作为一个直接访问子组件的应急方案——应当避免在模版或计算属性中使用 $refs 。

### 异步组件

在大型应用中，我们可能需要将应用拆分为多个小模块，按需从服务器下载。为了让事情更简单， Vue.js 允许将组件定义为一个工厂函数，动态地解析组件的定义。Vue.js 只在组件需要渲染时触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。例如：

|  |
| --- |
| Vue.component('async-example', function (resolve, reject) {  setTimeout(function () {  // Pass the component definition to the resolve callback  resolve({  template: '<div>I am async!</div>'  })  }, 1000)  }) |

工厂函数接收一个 resolve 回调，在收到从服务器下载的组件定义时调用。也可以调用 reject(reason) 指示加载失败。这里 setTimeout 只是为了演示。怎么获取组件完全由你决定。推荐配合使用 ：[Webpack 的代码分割功能](http://webpack.github.io/docs/code-splitting.html)：

|  |
| --- |
| Vue.component('async-webpack-example', function (resolve) {  // 这个特殊的 require 语法告诉 webpack  // 自动将编译后的代码分割成不同的块，  // 这些块将通过 Ajax 请求自动下载。  require(['./my-async-component'], resolve)  }) |

你可以使用 Webpack 2 + ES2015 的语法返回一个 Promise resolve 函数：

|  |
| --- |
| Vue.component(  'async-webpack-example',  () => System.import('./my-async-component')  ) |

如果你是 **Browserify** 用户,可能就无法使用异步组件了,它的作者已经[表明](https://github.com/substack/node-browserify/issues/58#issuecomment-21978224) Browserify 是不支持异步加载的。Browserify 社区发现 [一些解决方法](https://github.com/vuejs/vuejs.org/issues/620)，可能有助于已存在的复杂应用。对于其他场景，我们推荐简单实用 Webpack 构建，一流的异步支持

### 组件命名约定

当注册组件（或者 props）时，可以使用 kebab-case ，camelCase ，或 TitleCase 。Vue 不关心这个。

|  |
| --- |
| // 在组件定义中  components: {  // 使用 kebab-case 形式注册  'kebab-cased-component': { /\* ... \*/ },  // register using camelCase  'camelCasedComponent': { /\* ... \*/ },  // register using TitleCase  'TitleCasedComponent': { /\* ... \*/ }  } |

在 HTML 模版中，请使用 kebab-case 形式：

|  |
| --- |
| <!-- 在HTML模版中始终使用 kebab-case -->  <kebab-cased-component></kebab-cased-component>  <camel-cased-component></camel-cased-component>  <title-cased-component></title-cased-component> |

当使用字符串模式时，可以不受 HTML 的 case-insensitive 限制。这意味实际上在模版中，你可以使用 camelCase 、 TitleCase 或者 kebab-case 来引用：

|  |
| --- |
| <!-- 在字符串模版中可以用任何你喜欢的方式! -->  <my-component></my-component>  <myComponent></myComponent>  <MyComponent></MyComponent> |

如果组件未经 slot 元素传递内容，你甚至可以在组件名后使用 / 使其自闭合：

|  |
| --- |
| <my-component/> |

当然，这只在字符串模版中有效。因为自闭的自定义元素是无效的 HTML ，浏览器原生的解析器也无法识别它。

### 递归组件

组件在它的模板内可以递归地调用自己，不过，只有当它有 name 选项时才可以：

|  |
| --- |
| name: 'unique-name-of-my-component' |

当你利用Vue.component全局注册了一个组件, 全局的ID作为组件的 name 选项，被自动设置.

|  |
| --- |
| Vue.component('unique-name-of-my-component', {  // ...  }) |

如果你不谨慎, 递归组件可能导致死循环:

|  |
| --- |
| name: 'stack-overflow',  template: '<div><stack-overflow></stack-overflow></div>' |

上面组件会导致一个错误 “max stack size exceeded” ，所以要确保递归调用有终止条件 (比如递归调用时使用 v-if 并让他最终返回 false )。

### 组件间的循环引用Circular References Between Components

假设你正在构建一个文件目录树，像在Finder或文件资源管理器中。你可能有一个 tree-folder组件:

|  |
| --- |
| <p>  <span>{{ folder.name }}</span>  <tree-folder-contents :children="folder.children"/>  </p> |

然后 一个tree-folder-contents组件：

|  |
| --- |
| <ul>  <li v-for="child in children">  <tree-folder v-if="child.children" :folder="child"/>  <span v-else>{{ child.name }}</span>  </li>  </ul> |

When you look closely, you’ll see that these components will actually be each other’s descendent and ancestor in the render tree - a paradox! When registering components globally with Vue.component, this paradox is resolved for you automatically. If that’s you, you can stop reading here.  
当你仔细看时，会发现在渲染树上这两个组件同时为对方的父节点和子节点–这点是矛盾的。当使用Vue.component将这两个组件注册为全局组件的时候，框架会自动为你解决这个矛盾，如果你是这样做的，就不用继续往下看了。  
然而，如果你使用诸如Webpack或者Browserify之类的模块化管理工具来requiring/importing组件的话，就会报错了：

|  |
| --- |
| Failed to mount component: template or render function not defined. |

为了解释为什么会报错，简单的将上面两个组件称为 A 和 B ，模块系统看到它需要 A ，但是首先 A 需要 B ，但是 B 需要 A， 而 A 需要 B，陷入了一个无限循环，因此不知道到底应该先解决哪个。要解决这个问题，我们需要在其中一个组件中（比如 A ）告诉模块化管理系统，“A 虽然需要 B ，但是不需要优先导入 B”  
在我们的例子中，我们选择在tree-folder 组件中来告诉模块化管理系统循环引用的组件间的处理优先级，我们知道引起矛盾的子组件是tree-folder-contents，所以我们在beforeCreate 生命周期钩子中去注册它：

|  |
| --- |
| beforeCreate: function () {  this.$options.components.TreeFolderContents = require('./tree-folder-contents.vue')  } |

问题解决了。

### 内联模版

如果子组件有 inline-template 特性，组件将把它的内容当作它的模板，而不是把它当作分发内容。这让模板更灵活。

|  |
| --- |
| <my-component inline-template>  <div>  <p>These are compiled as the component's own template.</p>  <p>Not parent's transclusion content.</p>  </div>  </my-component> |

但是 inline-template 让模板的作用域难以理解。最佳实践是使用 template 选项在组件内定义模板或者在 .vue 文件中使用 template 元素。

### X-Templates

另一种定义模版的方式是在 JavaScript 标签里使用 text/x-template 类型，并且指定一个id。例如：

|  |
| --- |
| <script type="text/x-template" id="hello-world-template">  <p>Hello hello hello</p>  </script> |

|  |
| --- |
| Vue.component('hello-world', {  template: '#hello-world-template'  }) |

这在有很多模版或者小的应用中有用，否则应该避免使用，因为它将模版和组件的其他定义隔离了。

### 对低开销的静态组件使用 v-once

尽管在 Vue 中渲染 HTML 很快，不过当组件中包含**大量**静态内容时，可以考虑使用 v-once 将渲染结果缓存起来，就像这样：

|  |
| --- |
| Vue.component('terms-of-service', {  template: '\  <div v-once>\  <h1>Terms of Service</h1>\  ... a lot of static content ...\  </div>\  '  }) |

**原文：**<http://vuejs.org/guide/components.html>

← [表单控件绑定](https://cn.vuejs.org/v2/guide/forms.html)[深入响应式原理](https://cn.vuejs.org/v2/guide/reactivity.html) →

发现错误？想参与编辑？ [在 Github 上编辑此页！](https://github.com/vuejs/cn.vuejs.org/blob/master/src/v2/guide/components.md)