参考官方指南：<http://cn.vuejs.org/v2/guide/>

# 安装

### 兼容性

Vue.js 不支持 IE8 及其以下版本，因为 Vue.js 使用了 IE8 不能模拟的 ECMAScript 5 特性。 Vue.js 支持所有**[兼容 ECMAScript 5 的浏览器](http://caniuse.com/" \l "feat=es5" \t "http://cn.vuejs.org/v2/guide/_blank)**。

### 更新日志

每个版本的更新日志见 [**GitHub**](https://github.com/vuejs/vue/releases)。

## [独立版本](http://cn.vuejs.org/v2/guide/installation.html" \l "独立版本)

直接下载并用 <script> 标签引入，Vue 会被注册为一个全局变量。**重要提示：在开发时请用开发版本，遇到常见错误它会给出友好的警告。**

开发环境不要用最小压缩版，不然就失去了错误提示和警告!

[**开发版本**](http://vuejs.org/js/vue.js)包含完整的警告和调试模式  
  
[**生产版本**](http://vuejs.org/js/vue.min.js)删除了警告，23.55kb min+gzip

### [CDN](http://cn.vuejs.org/v2/guide/installation.html" \l "CDN)

推荐：**[unpkg](https://unpkg.com/vue/dist/vue.js" \t "http://cn.vuejs.org/v2/guide/_blank)**, 会保持和 npm 发布的最新的版本一致。可以在 [**unpkg.com/vue/**](https://unpkg.com/vue/) 浏览 npm 包资源。

也可以从 **[jsdelivr](http://cdn.jsdelivr.net/vue/2.0.5/vue.js)** 或 **[cdnjs](http://cdnjs.cloudflare.com/ajax/libs/vue/2.0.5/vue.js)** 获取，不过这两个服务版本更新可能略滞后。

## [NPM](http://cn.vuejs.org/v2/guide/installation.html" \l "NPM)

在用 Vue.js 构建大型应用时推荐使用 NPM 安装， NPM 能很好地和诸如 **[Webpack](http://webpack.github.io/" \t "http://cn.vuejs.org/v2/guide/_blank)** 或 **[Browserify](http://browserify.org/" \t "http://cn.vuejs.org/v2/guide/_blank)** 模块打包器配合使用。 Vue.js 也提供配套工具来开发**[单文件组件](http://cn.vuejs.org/v2/guide/single-file-components.html)**。

|  |
| --- |
| # 最新稳定版  $ npm install vue |

### [独立构建 vs 运行时构建](http://cn.vuejs.org/v2/guide/installation.html" \l "独立构建-vs-运行时构建)

有两种构建方式，独立构建和运行构建。它们的区别在于前者包含**模板编译器**而后者不包含。

模板编译用于编译 Vue 模板字符串成纯 JavaScript 渲染函数。如果你想用 template 选项， 你需要编译。

模板编译器的职责是将模板字符串编译为纯 JavaScript 的渲染函数。如果你想要在组件中使用template 选项，你就需要编译器。

独立构建包含模板编译器并支持 template 选项。 **它也依赖于浏览器的接口的存在，所以你不能使用它来为服务器端渲染。**

运行时构建不包含模板编译器，因此不支持 template 选项，只能用 render 选项，但即使使用运行时构建，在单文件组件中也依然可以写模板，因为单文件组件的模板会在构建时预编译为 render 函数。运行时构建比独立构建要轻量30%，只有 16.39 Kb min+gzip大小。

默认 NPM 包导出的是 **运行时** 构建。为了使用独立构建，在 webpack 配置中添加下面的别名：

|  |
| --- |
| resolve: {  alias: {  'vue$': 'vue/dist/vue.js'  }  } |

对于Browserify，可以用 **[aliasify](https://github.com/benbria/aliasify" \t "http://cn.vuejs.org/v2/guide/_blank)**

不要用 import Vue from 'vue/dist/vue.js' - 用一些工具或第三方库引入 Vue ，这可能会导致应用程序在同一时间加载运行时和独立构建并造成错误。

### [CSP 环境](http://cn.vuejs.org/v2/guide/installation.html" \l "CSP-环境)

有些环境，如 Google Chrome Apps ，强制应用内容安全策略 (CSP) ，不能使用 new Function() 对表达式求值。这时可以用 CSP 兼容版本。独立的构建取决于该功能编译模板，所以无法使用这些环境。

另一方面，运行时构建的是完全兼容 CSP 的。当通过 **[Webpack + vue-loader](https://github.com/vuejs-templates/webpack-simple" \t "http://cn.vuejs.org/v2/guide/_blank)** 或者 **[Browserify + vueify](https://github.com/vuejs-templates/browserify-simple" \t "http://cn.vuejs.org/v2/guide/_blank)** 构建时，在 CSP 环境中模板将被完美预编译到 render 函数中。

## [命令行工具](http://cn.vuejs.org/v2/guide/installation.html" \l "命令行工具)

Vue.js 提供一个**[官方命令行工具](https://github.com/vuejs/vue-cli" \t "http://cn.vuejs.org/v2/guide/_blank)**，可用于快速搭建大型单页应用。该工具提供开箱即用的构建工具配置，带来现代化的前端开发流程。只需一分钟即可启动带热重载、保存时静态检查以及可用于生产环境的构建配置的项目：

|  |
| --- |
| # 全局安装 vue-cli  $ npm install --global vue-cli  # 创建一个基于 webpack 模板的新项目  $ vue init webpack my-project  # 安装依赖，走你  $ cd my-project  $ npm install  $ npm run dev |

The CLI assumes prior knowledge of Node.js and the associated build tools. If you are new to Vue or front-end build tools, we strongly suggest going through [**the guide**](http://cn.vuejs.org/v2/guide/) without any build tools before using the CLI.

CLI 默认了用户对 Node.js 和相关构建工具有一定程度的了解。如果你是新手，我们强烈建议先在不用构建工具的情况下通读**[指南](http://cn.vuejs.org/guide/)**，熟悉 Vue 本身之后再研究 CLI。

对于大陆用户，建议将 npm 的注册表源**[设置为国内的镜像](http://riny.net/2014/cnpm/" \t "http://cn.vuejs.org/v2/guide/_blank)**，可以大幅提升安装速度。

## [开发版本](http://cn.vuejs.org/v2/guide/installation.html" \l "开发版本)

**重要**: Github 仓库的 /dist 文件夹只有在新版本发布时才会更新。如果想要使用 Github 上 Vue 最新的源码，你需要自己构建。

|  |
| --- |
| git clone https://github.com/vuejs/vue.git node\_modules/vue  cd node\_modules/vue  npm install  npm run build |

## [Bower](http://cn.vuejs.org/v2/guide/installation.html" \l "Bower)

|  |
| --- |
| # 最新稳定版本  $ bower install vue |

## [AMD 模块加载器](http://cn.vuejs.org/v2/guide/installation.html" \l "AMD-模块加载器)

独立下载版本或通过 Bower 安装的版本已用 UMD 包装，因此它们可以直接用作 AMD 模块。

**原文：**[**http://vuejs.org/guide/installation.html**](http://vuejs.org/guide/installation.html)

# 介绍

## [Vue.js 是什么](http://cn.vuejs.org/v2/guide/index.html" \l "Vue-js-是什么)

Vue.js（读音 /vjuː/, 类似于 **view**） 是一套构建用户界面的 **渐进式框架**。与其他重量级框架不同的是，Vue 采用自底向上增量开发的设计。Vue 的核心库只关注视图层，并且非常容易学习，非常容易与其它库或已有项目整合。另一方面，Vue 完全有能力驱动采用**[单文件组件](http://cn.vuejs.org/v2/guide/single-file-components.html)**和[**Vue生态系统支持的库**](http://github.com/vuejs/awesome-vue#libraries--plugins)开发的复杂单页应用。

Vue.js 的目标是通过尽可能简单的 API 实现**响应的数据绑定**和**组合的视图组件**。

如果你是有经验的前端开发者，想知道 Vue.js 与其它库/框架的区别，查看**[对比其它框架](http://cn.vuejs.org/v2/guide/comparison.html)**。

## [起步](http://cn.vuejs.org/v2/guide/index.html" \l "起步)

官方指南假设你已有HTML、CSS和JavaScript中级前端知识。如果你是全新的前端开发者，它可能不是最好的想法，将一个框架作为你的第一步-最好掌握基本知识再来！其他框架以前的经验是有帮助的，但不是必需的。

尝试 Vue.js 最简单的方法是使用 **[JSFiddle Hello World 例子](https://jsfiddle.net/chrisvfritz/50wL7mdz/" \t "http://cn.vuejs.org/v2/guide/_blank)**。你可以在浏览器新标签页中打开它，跟着我们学习一些基础示例。或者你也可以创建一个本地的 .html 文件，然后引入 Vue:

|  |
| --- |
| <script src="https://unpkg.com/vue/dist/vue.js"> |

你可以查看**[安装指南](http://cn.vuejs.org/guide/installation.html)**来了解其他安装 Vue 的选项。请注意我们**不推荐**新手直接使用 vue-cli，尤其是对 Node.js 构建工具不够了解的同学。

## [声明式渲染](http://cn.vuejs.org/v2/guide/index.html" \l "声明式渲染)

Vue.js 的核心是一个允许你采用简洁的模板语法来声明式的将数据渲染进 DOM 的系统：

|  |
| --- |
| <div id="app">  {{ message }}  </div> |

|  |
| --- |
| var app = new Vue({  el: '#app',  data: {  message: 'Hello Vue!'  }  }) |

Hello Vue!

我们已经生成了我们的第一个 Vue 应用！看起来这跟单单渲染一个字符串模板非常类似，但是 Vue.js 在背后做了大量工作。现在数据和 DOM 已经被绑定在一起，所有的元素都是**响应式的**。我们如何知道？打开你的浏览器的控制台，并修改 app.message，你将看到上例相应地更新。

除了绑定插入的文本内容，我们还可以采用这样的方式绑定 DOM 元素属性：

|  |
| --- |
| <div id="app-2">  <span v-bind:title="message">  Hover your mouse over me for a few seconds to see my dynamically bound title!  </span>  </div> |

|  |
| --- |
| var app2 = new Vue({  el: '#app-2',  data: {  message: 'You loaded this page on ' + new Date()  }  }) |

Hover your mouse over me for a few seconds to see my dynamically bound title!

这里我们遇到点新东西。你看到的 v-bind 属性被称为**指令**。指令带有前缀 v-，以表示它们是 Vue.js 提供的特殊属性。可能你已经猜到了，它们会在渲染过的 DOM 上应用特殊的响应式行为。这个指令的简单含义是说：将这个元素节点的 title 属性和 Vue 实例的 message 属性绑定到一起。

你再次打开浏览器的控制台输入 app2.message = 'some new message'，你就会再一次看到这个绑定了title属性的HTML已经进行了更新。

## [条件与循环](http://cn.vuejs.org/v2/guide/index.html" \l "条件与循环)

控制切换一个元素的显示也相当简单：

|  |
| --- |
| <div id="app-3">  <p v-if="seen">Now you see me</p>  </div> |

|  |
| --- |
| var app3 = new Vue({  el: '#app-3',  data: {  seen: true  }  }) |

Now you see me

继续在控制台设置 app3.seen = false，你会发现 “Now you see me” 消失了。

这个例子演示了我们不仅可以绑定 DOM 文本到数据，也可以绑定 DOM **结构**到数据。而且，Vue.js 也提供一个强大的过渡效果系统，可以在 Vue 插入/删除元素时自动应用**[过渡效果](http://cn.vuejs.org/v2/guide/transitions.html)**。

也有一些其它指令，每个都有特殊的功能。例如， v-for 指令可以绑定数据到数据来渲染一个列表：

|  |
| --- |
| <div id="app-4">  <ol>  <li v-for="todo in todos">  {{ todo.text }}  </li>  </ol>  </div> |

|  |
| --- |
| var app4 = new Vue({  el: '#app-4',  data: {  todos: [  { text: 'Learn JavaScript' },  { text: 'Learn Vue' },  { text: 'Build something awesome' }  ]  }  }) |

1. Learn JavaScript
2. Learn Vue
3. Build something awesome

在控制台里，输入 app4.todos.push({ text: 'New item' })。你会发现列表中多了一栏新内容。

## [处理用户输入](http://cn.vuejs.org/v2/guide/index.html" \l "处理用户输入)

为了让用户和你的应用进行互动，我们可以用 v-on 指令绑定一个监听事件用于调用我们 Vue 实例中定义的方法：

|  |
| --- |
| <div id="app-5">  <p>{{ message }}</p>  <button v-on:click="reverseMessage">Reverse Message</button>  </div> |

|  |
| --- |
| var app5 = new Vue({  el: '#app-5',  data: {  message: 'Hello Vue.js!'  },  methods: {  reverseMessage: function () {  this.message = this.message.split('').reverse().join('')  }  }  }) |

Hello Vue.js!

Reverse Message

在 reverseMessage 方法中，我们在没有接触 DOM 的情况下更新了应用的状态 - 所有的 DOM 操作都由 Vue 来处理，你写的代码只需要关注基本逻辑。

Vue 也提供了 v-model 指令，它使得在表单输入和应用状态中做双向数据绑定变得非常轻巧。

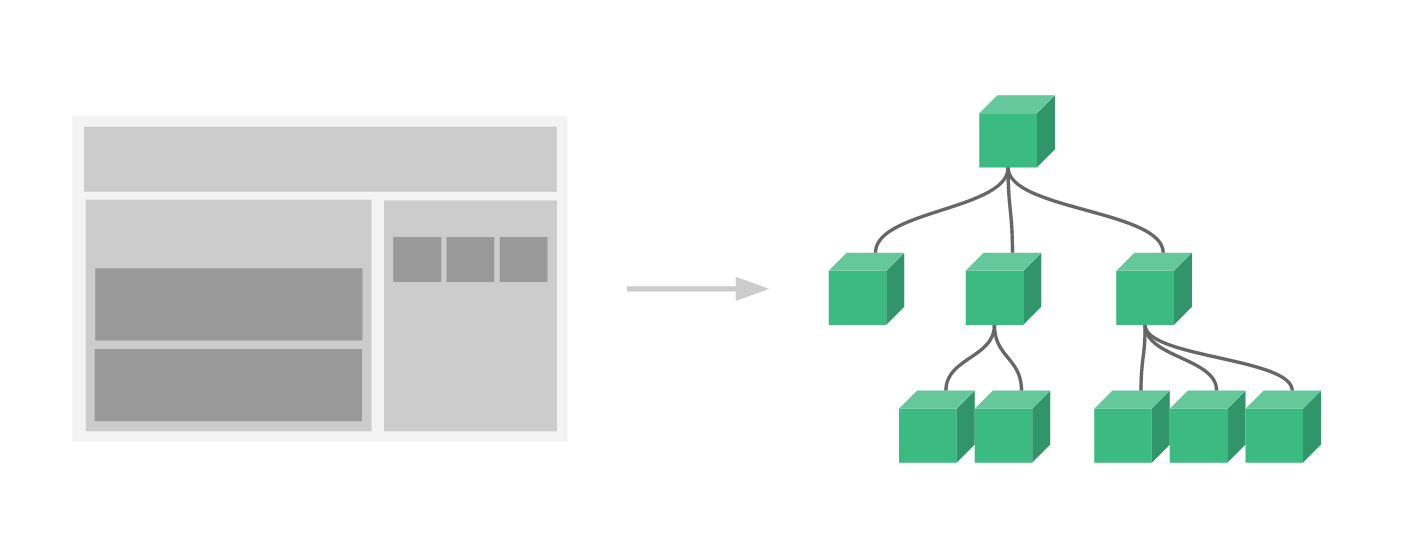
|  |
| --- |
| <div id="app-6">  <p>{{ message }}</p>  <input v-model="message">  </div> |

|  |
| --- |
| var app6 = new Vue({  el: '#app-6',  data: {  message: 'Hello Vue!'  }  }) |

Hello Vue!

## [用组件构建（应用）](http://cn.vuejs.org/v2/guide/index.html" \l "用组件构建（应用）)

组件系统是 Vue.js 另一个重要概念，因为它提供了一种抽象，让我们可以用独立可复用的小组件来构建大型应用。如果我们考虑到这点，几乎任意类型的应用的界面都可以抽象为一个组件树：



在 Vue 里，一个组件实质上是一个拥有预定义选项的一个 Vue 实例：

|  |
| --- |
| // Define a new component called todo-item  Vue.component('todo-item', {  template: '<li>This is a todo</li>'  }) |

现在你可以另一个组件模板中写入它：

|  |
| --- |
| <ul>  <!-- Create an instance of the todo-item component -->  <todo-item></todo-item>  </ul> |

但是这样会为每个 todo 渲染同样的文本，这看起来并不是很酷。我们应该将数据从父作用域传到子组件。让我们来修改一下组件的定义，使得它能够接受一个 [**prop**](http://cn.vuejs.org/v2/guide/components.html#Props) 字段：

|  |
| --- |
| Vue.component('todo-item', {  // The todo-item component now accepts a  // "prop", which is like a custom attribute.  // This prop is called todo.  props: ['todo'],  template: '<li>{{ todo.text }}</li>'  }) |

现在，我们可以使用 v-bind 指令将 todo 传到每一个重复的组件中：

|  |
| --- |
| <div id="app-7">  <ol>  <!-- Now we provide each todo-item with the todo object -->  <!-- it's representing, so that its content can be dynamic -->  <todo-item v-for="item in groceryList" v-bind:todo="item"></todo-item>  </ol>  </div> |

|  |
| --- |
| Vue.component('todo-item', {  props: ['todo'],  template: '<li>{{ todo.text }}</li>'  })  var app7 = new Vue({  el: '#app-7',  data: {  groceryList: [  { text: 'Vegetables' },  { text: 'Cheese' },  { text: 'Whatever else humans are supposed to eat' }  ]  }  }) |

1. Vegetables
2. Cheese
3. Whatever else humans are supposed to eat

这只是一个假设的例子，但是我们已经将应用分割成了两个更小的单元，子元素通过 props 接口实现了与父亲元素很好的解耦。我们现在可以在不影响到父应用的基础上，进一步为我们的todo 组件改进更多复杂的模板和逻辑。

在一个大型应用中，为了使得开发过程可控，有必要将应用整体分割成一个个的组件。在[**后面的教程**](http://cn.vuejs.org/v2/guide/components.html)中我们将详述组件，不过这里有一个（假想）的例子，看看使用了组件的应用模板是什么样的：

|  |
| --- |
| <div id="app">  <app-nav></app-nav>  <app-view>  <app-sidebar></app-sidebar>  <app-content></app-content>  </app-view>  </div> |

## [与自定义元素的关系](http://cn.vuejs.org/v2/guide/index.html" \l "与自定义元素的关系)

你可能已经注意到 Vue.js 组件非常类似于**自定义元素**——它是 [**Web 组件规范**](http://www.w3.org/wiki/WebComponents/)的一部分。实际上 Vue.js 的组件语法参考了该规范。例如 Vue 组件实现了 [**Slot API**](http://github.com/w3c/webcomponents/blob/gh-pages/proposals/Slots-Proposal.md) 与 is 特性。但是，有几个关键的不同：

Web 组件规范仍然远未完成，并且没有浏览器实现。相比之下，Vue.js 组件不需要任何补丁，并且在所有支持的浏览器（IE9 及更高版本）之下表现一致。必要时，Vue.js 组件也可以放在原生自定义元素之内。

Vue.js 组件提供了原生自定义元素所不具备的一些重要功能，比如组件间的数据流，自定义事件系统，以及动态的、带特效的组件替换。

## [准备好探索更广阔的世界了？](http://cn.vuejs.org/v2/guide/index.html" \l "准备好探索更广阔的世界了？)

我们刚才简单介绍了 Vue.js 核心的一些最基本的特征 - 本指南的其余部分将用更详尽的篇幅去描述其他的一些高级特性，所以一定要阅读完所有的内容哦！

**原文：**[**http://vuejs.org/guide/index.html**](http://vuejs.org/guide/index.html)

# Vue 实例

## [构造器](http://cn.vuejs.org/v2/guide/instance.html" \l "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（data）、模板(template)、挂载元素(el)、方法(method)、生命周期钩子等选项。全部的选项可以在 [**API 文档**](http://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**](http://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](http://cn.vuejs.org/v2/guide/instance.html" \l "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 //控制台访问时要使用app.$data.a或者app.a  })  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-双向绑定**。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" \t "http://cn.vuejs.org/v2/guide/_blank)**。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**](http://cn.vuejs.org/v2/api)。

## [实例生命周期](http://cn.vuejs.org/v2/guide/instance.html" \l "实例生命周期)

|  |
| --- |
| 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（挂在元素）、 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](http://cn.vuejs.org/v2/guide/instance.html" \l "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。





**原文：**[**http://vuejs.org/guide/instance.html**](http://vuejs.org/guide/instance.html)

### 模板语法

模板就是html

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](http://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—响应/双向绑定。

通过使用 [**v-once 指令**](http://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**](http://cn.vuejs.org/v2/api/#v-bind)：

通过v-bind使得id动态的变化

|  |
| --- |
| <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。

### 过滤器

Vue.js 允许你自定义过滤器，被用作一些常见的文本格式化。过滤器应该被添加在**mustache 插值**的尾部，由“管道符”指示：

|  |
| --- |
| {{ message | capitalize }} |

Vue 2.x 中，过滤器只能在 mustache 绑定中使用。为了在指令绑定中实现同样的行为，你应该使用**[计算属性](http://cn.vuejs.org/v2/guide/computed.html)**。

过滤器函数总接受表达式的值作为第一个参数。

|  |
| --- |
| new Vue({  // ...  filters: {  capitalize: function (value) {  if (!value) return ''  value = value.toString()  return value.charAt(0).toUpperCase() + value.slice(1)  }  }  }) |

过滤器可以串联：

|  |
| --- |
| {{ message | filterA | filterB }} |

过滤器是 JavaScript 函数，因此可以接受参数：

|  |
| --- |
| {{ message | filterA('arg1', arg2) }} |

这里，字符串 'arg1' 将传给过滤器作为第二个参数， arg2 表达式的值将被求值然后传给过滤器作为第三个参数，管道符之前的表达式的值作为第一个参数。

## 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

[在这里.prevent代表修饰符](http://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。

## 缩写

v- 前缀在模板中是作为一个标示 Vue 特殊属性的明显标识。当你使用 Vue.js 为现有的标记添加动态行为时，它会很有用，但对于一些经常使用的指令来说有点繁琐。同时，当搭建 Vue.js 管理所有模板的 [**SPA**](https://en.wikipedia.org/wiki/Single-page_application) 时，v- 前缀也变得没那么重要了。因此，Vue.js 为两个最为常用的指令提供了特别的缩写：

### v-bind 缩写

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

### v-on 缩写

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

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.js 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。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。

**原文：**[**http://vuejs.org/guide/syntax.html**](http://vuejs.org/guide/syntax.html)

# 计算属性

## 计算属性

在模板中绑定表达式是非常便利的，但是它们实际上只用于简单的操作。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> |

在这种情况下，模板不再简单和清晰。在实现反向显示 message 之前，你应该确认它。这个问题在你不止一次反向显示 message 的时候变得更加糟糕。

这就是为什么任何复杂逻辑，你都应当使用**计算属性—计算属性的名字可以被当做data的数据变量来使用，而且是响应式的（自带监听属性，当相关值改变后就会跟着改变）**。

### 基础例子

|  |
| --- |
| <div id="example">  <p>Original message: "{{ message }}"</p>  <p>Computed reversed message: "{{ reversedMessage }}"</p>  </div> |

|  |
| --- |
| var vm = new Vue({  computed修饰的属性不用在data中注册就可以当做data中的变量使用  el: '#example',  data: {  message: 'Hello'  },  computed: {  // a computed getter  reversedMessage: function () {  // `this` points to the vm instance  return this.message.split('').reverse().join('')  }  }  }) |

结果：

通过这样的一个过程，reversedMessage就变为了一个属性

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** 。

|  |
| --- |
| 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，修改 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，依赖于vm.reversedMessage 的绑定也会更新。而且最妙的是我们是声明式地创建这种依赖关系：计算属性的 getter 是干净无副作用的，因此也是易于测试和理解的。

### 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：

|  |
| --- |
| <p>Reversed message: "{{ reverseMessage() }}"</p> |

|  |
| --- |
| // in component  methods: {  会缓存结果，缓存到哪里？localstorge?  reverseMessage: function () {  return this.message.split('').reverse().join('')  }  } |

不经过计算属性，我们可以在 method 中定义一个相同的函数来替代它。对于最终的结果，两种方式确实是相同的。然而，不同的是**计算属性是基于它的依赖缓存**。计算属性只有在它的相关依赖发生改变时才会重新取值。这就意味着只要 message 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。

这也同样意味着如下计算属性将不会更新，因为 Date.now() 不是响应式依赖：

|  |
| --- |
| computed: {  now: function () {  return Date.now()  不依赖于其他项  }  } |

相比而言，每当重新渲染的时候，method 调用**总会**执行函数。

我们为什么需要缓存？假设我们有一个重要的计算属性 **A** ，这个计算属性需要一个巨大的数组遍历和做大量的计算。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 method 替代。

### 计算属性 vs Watched Property

Vue.js 提供了一个方法 $watch ，它用于观察 Vue 实例上的数据变动。当一些数据需要根据其它数据变化时， $watch 很诱人 —— 特别是如果你来自 AngularJS 。不过，通常更好的办法是使用计算属性而不是一个命令式的 $watch 回调。思考下面例子：

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

|  |
| --- |
| var vm = new Vue({  el: '#demo',  data: {  firstName: 'Foo',  lastName: 'Bar',  fullName: 'Foo Bar'  一个参数时代表改变后的值；两个参数时第一个为新值，第二个为旧值  监听firstName，如果该值改变就会执行这个函数  },  watch: {  firstName: function (val) {  this.fullName = val + ' ' + this.lastName  监听lastName，如果该值改变就会执行这个函数  },  lastName: function (val) {  this.fullName = this.firstName + ' ' + val  }  }  }) |

上面代码是命令式的和重复的。跟计算属性对比：

|  |
| --- |
| var vm = new Vue({  el: '#demo',  data: {  firstName: 'Foo',  lastName: 'Bar'  计算属性是响应式的  },  computed: {  fullName: function () {  return this.firstName + ' ' + this.lastName  }  }  }) |

这样更好，不是吗？

### 计算 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]  运行vm.fullName时会调用getter  }  }  }  // ... |

现在在运行 vm.fullName = 'John Doe' 时， setter 会被调用， vm.firstName 和vm.lastName 也会被对应更新。

大多数情况下还是使用计算属性

## 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（计算属性和watch都会是响应式）。当你想要在数据变化响应时，执行异步操作或昂贵操作时，这是很有用的。

例如：

|  |
| --- |
| <div id="watch-example">  <p>  Ask a yes/no question:  <input v-model="question">  </p>  <p>{{ answer }}</p>  </div> |

|  |
| --- |
| <!-- Since there is already a rich ecosystem of ajax libraries -->  <!-- and collections of general-purpose utility methods, Vue core -->  <!-- is able to remain small by not reinventing them（重开发它们）. This also -->  <!-- gives you the freedom to just use what you're familiar with. -->  <script src="https://unpkg.com/axios@0.12.0/dist/axios.min.js"></script>  <script src="https://unpkg.com/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) {  this.answer = 'Waiting for you to stop typing...'  this.getAnswer()  }  },  methods: {    // \_.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    // 学习更多关于 \_.debounce function (and its cousin  // \_.throttle), 参考: https://lodash.com/docs#debounce  getAnswer: \_.debounce(  function () {  var vm = this  if (this.question.indexOf('?') === -1) {  vm.answer = 'Questions usually contain a question mark. ;-)'  return  }  vm.answer = 'Thinking...'  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  })  },  // 这是我们为用户停止输入等待的毫秒数  500  )  }  })  </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，并直到我们得到最终结果时，才设置中间状态。这是计算属性无法做到的。

除了 watch 选项之外，您还可以使用 **[vm.$watch API](http://cn.vuejs.org/v2/api/" \l "vm-watch)** 命令。

**原文：**[**http://vuejs.org/guide/computed.html**](http://vuejs.org/guide/computed.html)

# Class 与 Style 绑定

数据绑定一个常见需求是操作元素的 class 列表和它的内联样式。因为它们都是属性 ，我们可以用v-bind 处理它们：只需要计算出表达式最终的字符串。不过，字符串拼接麻烦又易错。因此，在v-bind 用于 class 和 style 时， Vue.js 专门增强了它。表达式的结果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 。

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

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

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

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

如下 data:

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

渲染为:

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

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

你也可以直接绑定数据里的一个对象：

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

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

渲染的结果和上面一样。我们也可以在这里绑定返回对象的**[计算属性](http://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',  }  }  } |

数组中的值是data的变量数据

### 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]"> |

### With Components（组件）

**This section assumes knowledge of [Vue Components](http://cn.vuejs.org/v2/guide/components.html). Feel free to skip it and come back later.**

When you use the class attribute on a custom component, those classes will be added to the component’s root element（那些class属性将会被添加到组件的根元素）. Existing classes on this element will not be overwritten（在该组件中已经存在的class属性不会被重写覆盖）.

For example, if you declare this component:

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

Then add some classes when using it:

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

The rendered HTML（html原型） will be:

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

The same is true for class bindings:

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

When isActive is truthy, the rendered HTML will be:

|  |
| --- |
| <div class="foo bar active"></div> |

## 绑定内联样式

[绑定style时，用单引号引起来的直接值显示，没有用单引号括起来就是vue的data数据](http://cn.vuejs.org/v2/guide/class-and-style.html" \l "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。

**原文：**[**http://vuejs.org/guide/class-and-style.html**](http://vuejs.org/guide/class-and-style.html)

# 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：

vue.js代码

var vm=new Vue({

el:"#app",

data:{

ok:true

}

})

|  |
| --- |
| <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 v-if

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它。

<div id="app">

<h1 v-if="ok">show something...</h1>

<h1 v-else>hide all</h1>

</div>

<script>

var vm=new Vue({

el:"#app",

data:{

ok:Math.random()>0.5?true:false

}

})

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

### v-else

可以用 v-else 指令给 v-if 或 v-show 添加一个 “else” 块：

|  |
| --- |
| <div v-if="Math.random() > 0.5">  Sorry  </div>  <div v-else>  Not sorry  </div> |

v-else 元素必须紧跟在 v-if 或 v-show 元素的后面——否则它不能被识别。

### v-show

[使用v-if时根据条件可能dom不包含相应的元素](http://cn.vuejs.org/v2/guide/conditional.html" \l "v-show)

另一个根据条件展示元素的选项是 v-show 指令。用法大体上一样：

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

不同的是有 v-show 的元素会始终渲染并保持在 DOM 中。v-show 是简单的切换元素的 CSS 属性 display 。

注意 v-show 不支持 <template> 语法。

使用的是css的display

## v-if vs. v-show

v-if 是真实的条件渲染，因为它会确保条件块在切换当中适当地销毁与重建条件块内的事件监听器和子组件。

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——在条件第一次变为真时才开始局部编译（编译会被缓存起来）。

相比之下， v-show 简单得多——元素始终被编译并保留，只是简单地基于 CSS 切换。

一般来说， v-if 有更高的切换消耗而 v-show 有更高的初始渲染消耗。因此，如果需要频繁切换使用 v-show 较好，如果在运行时条件不大可能改变则使用 v-if 较好。

**原文：**[**http://vuejs.org/guide/conditional.html**](http://vuejs.org/guide/conditional.html)

# 列表渲染

## 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({  js中构建json时 key可以不用加引号  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**](http://cn.vuejs.org/v2/guide/components.html)**。Feel free to skip it and come back later.**

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，而不是移动 DOM 元素来匹配数据项的顺序， Vue 将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 数组调用突变方法：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. 当你直接设置一个项的索引时，例如： vm.items[indexOfItem] = newValue
2. 当你修改数组的长度时，例如： vm.items.length = newLength

为了避免第一种情况，以下两种方式将达到像 vm.items[indexOfItem] = newValue 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) |

避免第二种情况，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，您也可以使用在计算属性是不可行的 method 方法 (例如，在嵌套 v-for 循环中)：

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

**原文：**[**http://vuejs.org/guide/list.html**](http://vuejs.org/guide/list.html)