官方指南：<http://cn.vuejs.org/v2/guide/>

# 基础

## 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/#feat=es5)。

### 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), 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。

Me：可以使用script引入网络上的unpkg包，因为是cdn所以很快

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/) 或 [**Browserify**](http://browserify.org/) 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)

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) 或者 [**Browserify + vueify**](https://github.com/vuejs-templates/browserify-simple) 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)，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（me：类似其他的nodejs项目）  $ 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/)，可以大幅提升安装速度。

### [对不同构建版本的解释](https://cn.vuejs.org/v2/guide/installation.html" \l "对不同构建版本的解释)

在 [**NPM 包的 dist/ 目录**](https://cdn.jsdelivr.net/npm/vue/dist/)你将会找到很多不同的 Vue.js 构建。这里列出了他们之间的差别：

|  | **UMD** | **CommonJS** | **ES Module** |
| --- | --- | --- | --- |
| **完整版** | vue.js | vue.common.js | vue.esm.js |
| **只包含运行时** | vue.runtime.js | vue.runtime.common.js | vue.runtime.esm.js |
| **完整版 (生产环境)** | vue.min.js | - | - |
| **只包含运行时 (生产环境)** | vue.runtime.min.js | - | - |

### 术语

* **完整版**：同时包含编译器和运行时的构建。
* **编译器**：用来将模板字符串编译成为 JavaScript 渲染函数的代码。
* **运行时**：用来创建 Vue 实例，渲染并处理 virtual DOM 等行为的代码。基本上就是除去编译器的其他一切。
* [**UMD**](https://github.com/umdjs/umd)：UMD 构建可以直接通过 <script> 标签用在浏览器中。jsDelivr CDN 的 [**https://cdn.jsdelivr.net/npm/vue**](https://cdn.jsdelivr.net/npm/vue) 默认文件就是运行时 + 编译器的 UMD 构建 (vue.js)。
* [**CommonJS**](http://wiki.commonjs.org/wiki/Modules/1.1)：CommonJS 构建用来配合老的打包工具比如 [**browserify**](http://browserify.org/) 或 [**webpack 1**](https://webpack.github.io/)。这些打包工具的默认文件 (pkg.main) 是只包含运行时的 CommonJS 构建 (vue.runtime.common.js)。
* [**ES Module**](http://exploringjs.com/es6/ch_modules.html)：ES module 构建用来配合现代打包工具比如 [**webpack 2**](https://webpack.js.org/) 或 [**rollup**](https://rollupjs.org/)。这些打包工具的默认文件 (pkg.module) 是只包含运行时的 ES Module 构建 (vue.runtime.esm.js)。

### 运行时 + 编译器 vs. 只包含运行时

如果你需要在客户端编译模板 (比如传入一个字符串给 template 选项，或挂载到一个元素上并以其内部的 HTML 作为模板)，你将需要加上编译器，即完整版的构建：

|  |
| --- |
| // 需要编译器  new Vue({  template: '<div>{{ hi }}</div>'  })  // 不需要编译器  new Vue({  render (h) {  return h('div', this.hi)  }  }) |

当使用 vue-loader 或 vueify 的时候，\*.vue 文件内部的模板会在构建时预编译成 JavaScript。你在最终打好的包里实际上是不需要编译器的，因为只是用运行时构建即可。

因为运行时构建相比完整版缩减了 30% 的体积，你应该尽可能使用这个版本。如果你仍然希望使用完整版构建，你需要在你的打包工具里配置一个别名：

### Webpack

|  |
| --- |
| module.exports = {  // ...  resolve: {  alias: {  'vue$': 'vue/dist/vue.esm.js' // 'vue/dist/vue.common.js' for webpack 1  }  }  } |

### Rollup

|  |
| --- |
| const alias = require('rollup-plugin-alias')  rollup({  // ...  plugins: [  alias({  'vue': 'vue/dist/vue.esm.js'  })  ]  }) |

### Browserify

添加到你项目的 package.json：

|  |
| --- |
| {  // ...  "browser": {  "vue": "vue/dist/vue.common.js"  }  } |

### 开发环境 vs. 生产环境模式

开发环境/生产环境模式是硬编码的 UMD 构建：开发环境下不压缩代码，生产环境下压缩代码。

CommonJS 和 ES Module 构建是用于打包工具的，因此我们不提供压缩后的版本。你有必要在打最终包的时候压缩它们。

CommonJS 和 ES Module 构建同时保留原始的 process.env.NODE\_ENV 检测，以决定它们应该运行在什么模式下。你应该使用适当的打包工具配置来替换它们的环境变量以便控制 Vue 所运行的模式。把 process.env.NODE\_ENV 替换为字符串字面量同样可以让 UglifyJS 之类的压缩工具完全丢掉仅供开发环境的代码段，减少最终的文件尺寸。

### Webpack

使用 Webpack 的 [**DefinePlugin**](https://webpack.js.org/plugins/define-plugin/)：

|  |
| --- |
| var webpack = require('webpack')  module.exports = {  // ...  plugins: [  // ...  new webpack.DefinePlugin({  'process.env': {  NODE\_ENV: JSON.stringify('production')  }  })  ]  } |

### Rollup

使用 [**rollup-plugin-replace**](https://github.com/rollup/rollup-plugin-replace)：

|  |
| --- |
| const replace = require('rollup-plugin-replace')  rollup({  // ...  plugins: [  replace({  'process.env.NODE\_ENV': JSON.stringify('production')  })  ]  }).then(...) |

### Browserify

为你的包提供一个全局的 [**envify**](https://github.com/hughsk/envify) 转换。

|  |
| --- |
| NODE\_ENV=production browserify -g envify -e main.js | uglifyjs -c -m > build.js |

也可以移步到[**生产环境部署提示**](https://cn.vuejs.org/v2/guide/deployment.html)。

### 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) 或者 [**Browserify + vueify**](https://github.com/vuejs-templates/browserify-simple) 构建时，在 CSP 环境中模板将被完美预编译到 render 函数中。

### 开发版构建

**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

Bower 只提供 UMD 构建。

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

### AMD 模块加载器

所有 UMD 构建都可以直接用作 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（me：即双向的数据绑定）**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/)。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”>//这里是通过cdn来引入的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({//vue引入就会在全局注册了一个全局变量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**。（me:双向绑定）我们如何知道？打开你的浏览器的控制台（me：当在浏览器的控制台执行语句时相当于是将js代码注入到了页面中，所以在请求源网页的domain时并不是跨域请求，所以electron可以在里面加载其他的页面并请求对应的网站），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' }  ]  }  }) |

Learn JavaScript

Learn Vue

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（me:同理如果对数组进行pop也会看到数据减少了）。

## [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: {//定义vue的方法  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。

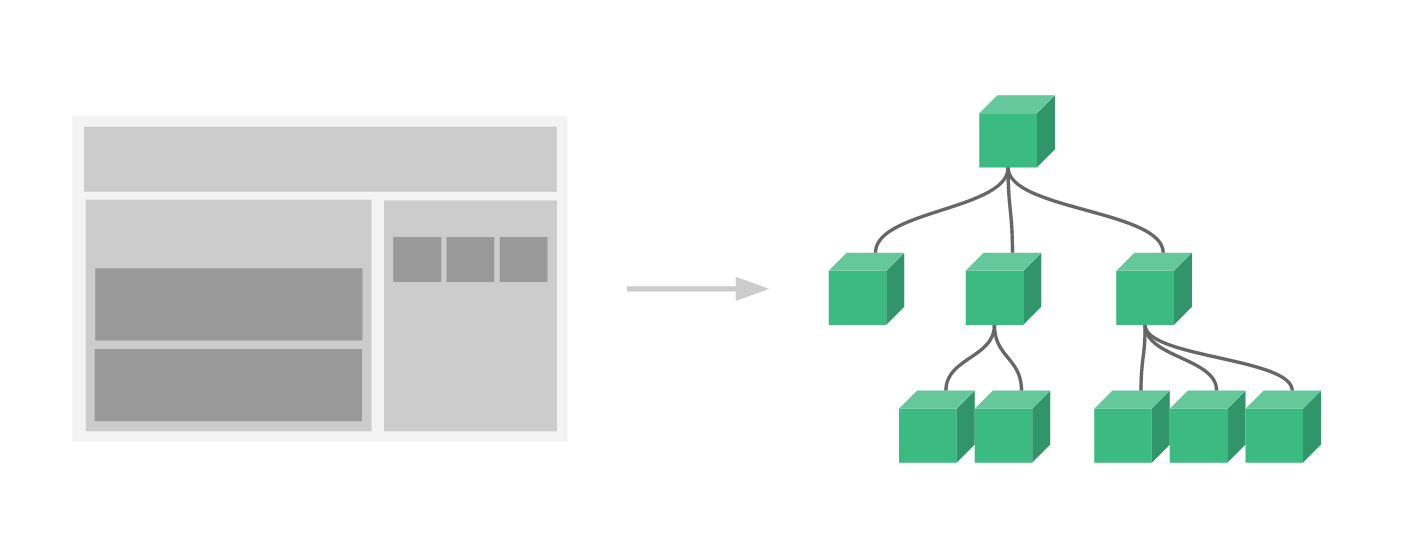
|  |
| --- |
| <div id="app-6">  <p>{{ message }}</p>  <input v-model="message">  </div> |

|  |
| --- |
| 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>'  }) |

现在你可以在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' }  ]  }  }) |

Vegetables

Cheese

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.$data.a或者app.a；app.$data访问到data属性  })  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)。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 () {//该方法会在应用第一次创建的时候执行  // `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（挂载元素）、 updated、destroyed（me：这些钩子方法使得当指定的事件触发时自动执行对应的钩子函数） 。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">{{message}}</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。

通过v-bind使得id动态的变化

### 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)(me：即绑定html的属性时使用v-bind显示指定绑定的数据)：

|  |
| --- |
| <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。

在这里.prevent代表修饰符

### 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({  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  el: '#demo',  data: {  firstName: 'Foo',  lastName: 'Bar',  fullName: 'Foo Bar'  监听firstName，如果该值改变就会执行这个函数  },  watch: {  监听lastName，如果该值改变就会执行这个函数  firstName: function (val) {  this.fullName = val + ' ' + this.lastName  },  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/#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 class active 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**](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

### 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: {  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  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)

# 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：

可加括号也可以不加

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

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

结果：

Greet

## 内联处理器方法

除了直接绑定到一个方法，也可以用内联 JavaScript 语句：

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

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

结果：

Say hi Say what

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

|  |
| --- |
| <button v-on:click="warn('Form cannot be submitted yet.', $event)">Submit</button> |

|  |
| --- |
| // ...  methods: {  warn: function (message, event) {  // 现在我们可以访问原生事件对象  if (event) event.preventDefault()  alert(message)  }  } |

## 事件修饰符

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

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

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

|  |
| --- |
| <!-- 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> |

## 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**](http://cn.vuejs.org/v2/api/#keyCodes)：

|  |
| --- |
| // 可以使用 v-on:keyup.f1  Vue.config.keyCodes.f1 = 112 |

## 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。

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

# 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。

### 文本

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



Message is:

### 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> |

Multiline message is:



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> |

 false

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: []  }  }) |

 Jack  John  Mike   
Checked names: []

### 单选按钮

|  |
| --- |
| <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> |

 One   
 Two   
Picked:

### 选择列表

单选列表:

|  |
| --- |
| <select v-model="selected">  <option>A</option>  <option>B</option>  <option>C</option>  </select>  <span>Selected: {{ selected }}</span> |

 Selected:

多选列表（绑定到一个数组）：

|  |
| --- |
| <select v-model="selected" multiple>  <option>A</option>  <option>B</option>  <option>C</option>  </select>  <br>  <span>Selected: {{ selected }}</span> |

   
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' }  ]  }  }) |

 Selected: A

## 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，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 with Components

**If you’re not yet familiar with Vue’s components, just skip this for now.**

HTML’s built-in input types won’t always meet your needs. Fortunately, Vue components allow you to build reusable inputs with completely customized behavior. These inputs even work with v-model! To learn more, read about [**custom inputs**](http://cn.vuejs.org/v2/guide/components.html#Form-Input-Components-using-Custom-Events) in the Components guide.

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

# 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 is 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。出错

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

使用组件时，大多数选项可以被传入到 Vue 构造器中，有一个例外： data 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 会在控制台发出警告，告诉你在组件中 data 必须是一个函数。最好理解这种规则的存在意义。

|  |
| --- |
| <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>',   // data 是一个函数，因此 Vue 不会警告，  // 但是我们为每一个组件返回了同一个对象引用  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！我们可以通过为每个组件返回新的 data 对象来解决这个问题：

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

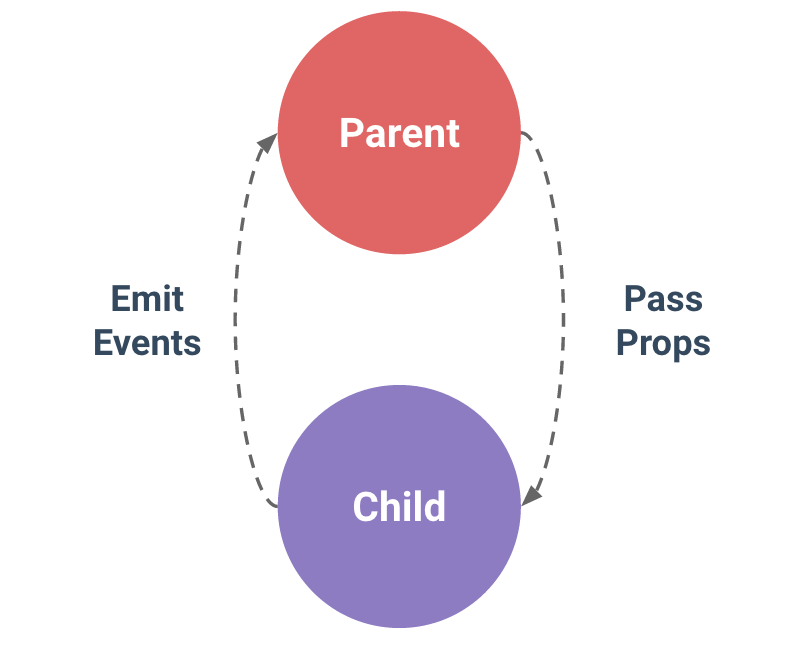
现在每个 counter 都有它自己内部的状态了：

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。



## Props

### 使用Props传递数据

组件实例的作用域是**孤立的**。这意味着不能并且不应该在子组件的模板内直接引用父组件的数据。可以使用 props 把数据传给子组件。

prop 是父组件用来传递数据的一个自定义属性。子组件需要显式地用 [**props 选项**](http://cn.vuejs.org/v2/api/#props) 声明 “prop”：

|  |
| --- |
| 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 特性不区分大小写。当使用非字符串模版时，prop的名字形式会从 camelCase 转为 kebab-case（短横线隔开）：

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

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

再次说明，如果你使用字符串模版，不用在意这些限制。

### 动态 Props

类似于用 v-bind 绑定 HTML 特性到一个表达式，也可以用 v-bind 动态绑定 props 的值到父组件的数据中。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 ，它的值以字符串 "1" 而不是以实际的数字传下去。如果想传递一个实际的 JavaScript 数字，需要使用 v-bind ，从而让它的值被当作 JavaScript 表达式计算：

|  |
| --- |
| <!-- 传递实际的数字 -->  <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。

通常有两种改变 prop 的情况：

1. prop 作为初始值传入，子组件之后只是将它的初始值作为本地数据的初始值使用；
2. prop 作为需要被转变的原始值传入。

更确切的说这两种情况是：

1. 定义一个局部 data 属性，并将 prop 的初始值作为局部数据的初始值。

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

1. 定义一个 computed 属性，此属性从 prop 的值计算得出。

|  |
| --- |
| 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

组件可以为 props 指定验证要求。如果未指定验证要求，Vue 会发出警告。当组件给其他人使用时这很有用。

prop 是一个对象而不是字符串数组时，它包含验证要求：

|  |
| --- |
| 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，使用 instanceof 检测。

当 prop 验证失败了， Vue 将拒绝在子组件上设置此值，如果使用的是开发版本会抛出一条警告。

## 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)**](http://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。

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。它所做的只是触发一个父组件关心的内部事件。

#### 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

自定义事件也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。牢记：

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

仅仅是一个语法糖：

|  |
| --- |
| <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：

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

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

一个非常简单的货币输入：

|  |
| --- |
| <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: {  // Instead of updating the value directly, this  // method is used to format and place constraints  // on the input's value  updateValue: function (value) {  var formattedValue = value  // Remove whitespace on either side  .trim()  // Shorten to 2 decimal places  .slice(0, value.indexOf('.') + 3)  // If the value was not already normalized,  // manually override it to conform  if (formattedValue !== value) {  this.$refs.input.value = formattedValue  }  // Emit the number value through the input event  this.$emit('input', Number(formattedValue))  }  }  }) |

$ 

The implementation above is pretty naive though. For example, users are allowed to enter multiple periods and even letters sometimes - yuck! So for those that want to see a non-trivial example, here’s a more robust currency filter:

这个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

有时候非父子关系的组件也需要通信。在简单的场景下，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) {  // ...  }) |

在更多复杂的情况下，你应该考虑使用专门的 [**状态管理模式**](http://cn.vuejs.org/v2/guide/state-management.html).

## 使用 Slots 分发内容

在使用组件时，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，我们先明确内容的编译作用域。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。

如果要绑定子组件内的指令到一个组件的根节点，应当在它的模板内这么做：

|  |
| --- |
| 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，分发内容是在父组件作用域内编译。

### 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>I'm the child title</h2>  <slot>  如果没有分发内容则显示我。  </slot>  </div> |

父组件模版：

|  |
| --- |
| <div>  <h1>I'm the parent title</h1>  <my-component>  <p>This is some original content</p>  <p>This is some more original content</p>  </my-component>  </div> |

渲染结果：

|  |
| --- |
| <div>  <h1>I'm the parent title</h1>  <div>  <h2>I'm the child title</h2>  <p>This is some original content</p>  <p>This is some more original content</p>  </div>  </div> |

### 具名Slots

<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">Here might be a page title</h1>  <p>A paragraph for the main content.</p>  <p>And another one.</p>  <p slot="footer">Here's some contact info</p>  </app-layout> |

渲染结果为：

|  |
| --- |
| <div class="container">  <header>  <h1>Here might be a page title</h1>  </header>  <main>  <p>A paragraph for the main content.</p>  <p>And another one.</p>  </main>  <footer>  <p>Here's some contact info</p>  </footer>  </div> |

在组合组件时，内容分发 API 是非常有用的机制。

## 动态组件

多个组件可以使用同一个挂载点，然后动态地在它们之间切换。使用保留的 <component> 元素，动态地绑定到它的 is 特性：

|  |
| --- |
| 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**](http://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 () {  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。如果这个功能对你很重要，请使用 Webpack。

### 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 camelCase 形式注册  'kebab-cased-component': { /\* ... \*/ },  'camelCasedComponent': { /\* ... \*/ },  '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 、 PascalCase 或者 kebab-case 来引用你的组件和 prop：

|  |
| --- |
| <!-- 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 )。

### 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。

### 使用 v-once 的低级静态组件(Cheap Static Component)

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**](http://vuejs.org/guide/components.html)