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。

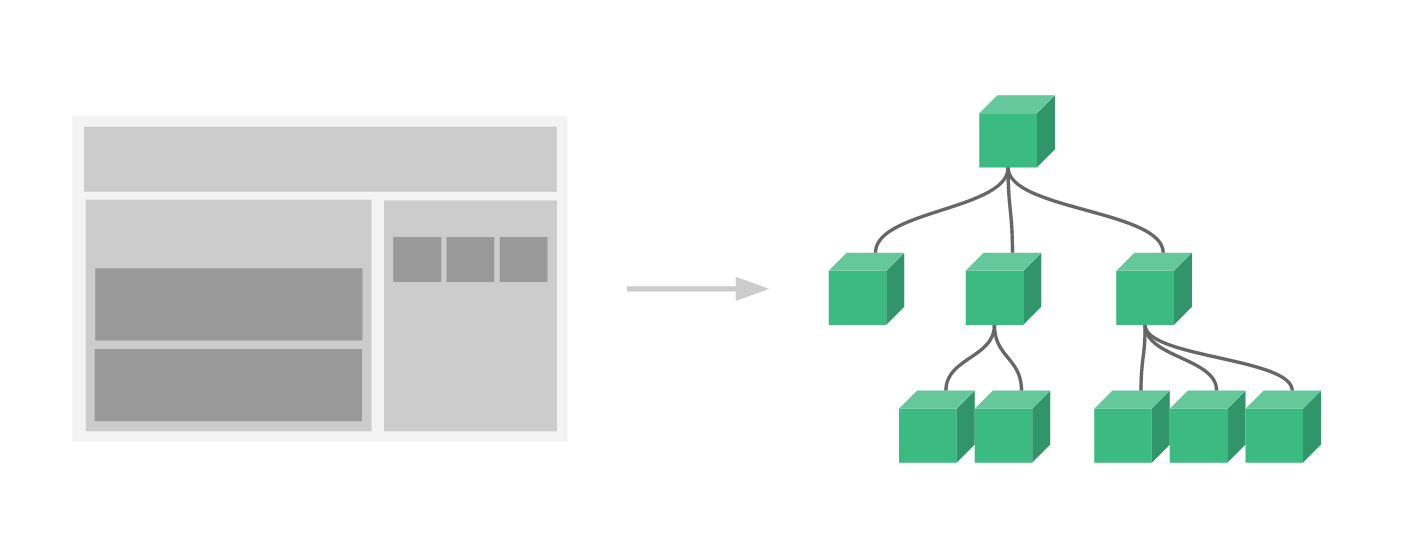
|  |
| --- |
| <div id="app-6">  <p>{{ message }}</p>  <input v-model="message">  </div> |

|  |
| --- |
| 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}}  vue.js代码  var vm=new Vue({  el:"#app",  data:{  ok:true  }  }) |

在 Vue.js ，我们使用 v-if 指令实现同样的功能：

|  |
| --- |
| <h1 v-if="ok">Yes</h1> |

也可以用 v-else 添加一个 “else” 块：

|  |
| --- |
| <h1 v-if="ok">Yes</h1>  <h1 v-else>No</h1> |

### template 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。

使用v-if时根据条件可能dom不包含相应的元素

### 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)

组件未学习

//======================20171011

# 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  }  } |

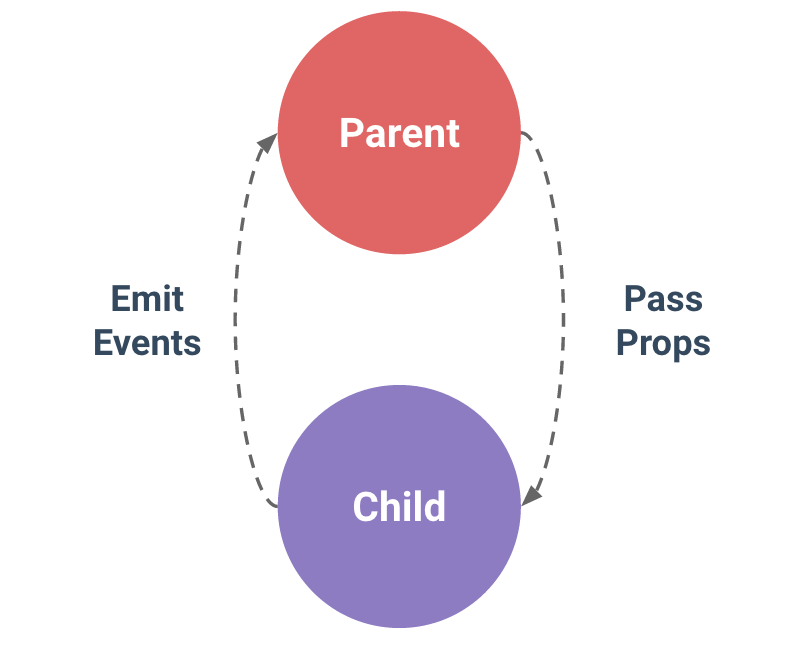
现在每个 counter 都有它自己内部的状态了：

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)

### 过渡 & 动画

# 进入/离开 & 列表过渡

## 概述

Vue 在插入、更新或者移除 DOM 时，提供多种不同方式的应用过渡效果。  
包括以下工具：

* 在 CSS 过渡和动画中自动应用 class
* 可以配合使用第三方 CSS 动画库，如 Animate.css
* 在过渡钩子函数中使用 JavaScript 直接操作 DOM
* 可以配合使用第三方 JavaScript 动画库，如 Velocity.js

在这里，我们只会讲到进入、离开和列表的过渡，你也可以看下一节的 [**管理过渡状态**](https://cn.vuejs.org/v2/guide/transitioning-state.html)。

## 单元素/组件的过渡

Vue 提供了 transition 的封装组件，在下列情形中，可以给任何元素和组件添加 entering/leaving 过渡

* 条件渲染 (使用 v-if)
* 条件展示 (使用 v-show)
* 动态组件
* 组件根节点

这里是一个典型的例子：

|  |
| --- |
| <div id="demo">  <button v-on:click="show = !show">  Toggle  </button>  <transition name="fade">  <p v-if="show">hello</p>  </transition>  </div> |

|  |
| --- |
| new Vue({  el: '#demo',  data: {  show: true  }  }) |

|  |
| --- |
| .fade-enter-active, .fade-leave-active {  transition: opacity .5s  }  .fade-enter, .fade-leave-to /\* .fade-leave-active in below version 2.1.8 \*/ {  opacity: 0  } |

Toggle

hello

当插入或删除包含在 transition 组件中的元素时，Vue 将会做以下处理：

1. 自动嗅探目标元素是否应用了 CSS 过渡或动画，如果是，在恰当的时机添加/删除 CSS 类名。
2. 如果过渡组件提供了 [**JavaScript 钩子函数**](https://cn.vuejs.org/v2/guide/transitions.html#JavaScript-钩子)，这些钩子函数将在恰当的时机被调用。
3. 如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作 (插入/删除) 在下一帧中立即执行。(注意：此指浏览器逐帧动画机制，和 Vue 的 nextTick 概念不同)

### 过渡的类名

在进入/离开的过渡中，会有 6 个 class 切换。

1. v-enter：定义进入过渡的开始状态。在元素被插入时生效，在下一个帧移除。
2. v-enter-active：定义过渡的状态。在元素整个过渡过程中作用，在元素被插入时生效，在 transition/animation 完成之后移除。这个类可以被用来定义过渡的过程时间，延迟和曲线函数。
3. v-enter-to: **2.1.8版及以上** 定义进入过渡的结束状态。在元素被插入一帧后生效 (于此同时 v-enter 被删除)，在 transition/animation 完成之后移除。
4. v-leave: 定义离开过渡的开始状态。在离开过渡被触发时生效，在下一个帧移除。
5. v-leave-active：定义过渡的状态。在元素整个过渡过程中作用，在离开过渡被触发后立即生效，在 transition/animation 完成之后移除。这个类可以被用来定义过渡的过程时间，延迟和曲线函数。
6. v-leave-to: **2.1.8版及以上** 定义离开过渡的结束状态。在离开过渡被触发一帧后生效 (于此同时 v-leave 被删除)，在 transition/animation 完成之后移除。

对于这些在 enter/leave 过渡中切换的类名，v- 是这些类名的前缀。使用 <transition name="my-transition"> 可以重置前缀，比如 v-enter 替换为 my-transition-enter。

v-enter-active 和 v-leave-active 可以控制 进入/离开 过渡的不同阶段，在下面章节会有个示例说明。

### CSS 过渡

常用的过渡都是使用 CSS 过渡。

下面是一个简单例子：

|  |
| --- |
| <div id="example-1">  <button @click="show = !show">  Toggle render  </button>  <transition name="slide-fade">  <p v-if="show">hello</p>  </transition>  </div> |

|  |
| --- |
| new Vue({  el: '#example-1',  data: {  show: true  }  }) |

|  |
| --- |
| /\* 可以设置不同的进入和离开动画 \*/  /\* 设置持续时间和动画函数 \*/  .slide-fade-enter-active {  transition: all .3s ease;  }  .slide-fade-leave-active {  transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);  }  .slide-fade-enter, .slide-fade-leave-to  /\* .slide-fade-leave-active for below version 2.1.8 \*/ {  transform: translateX(10px);  opacity: 0;  } |

Toggle render

hello

### CSS 动画

CSS 动画用法同 CSS 过渡，区别是在动画中 v-enter 类名在节点插入 DOM 后不会立即删除，而是在 animationend 事件触发时删除。

示例：(省略了兼容性前缀)

|  |
| --- |
| <div id="example-2">  <button @click="show = !show">Toggle show</button>  <transition name="bounce">  <p v-if="show">Look at me!</p>  </transition>  </div> |

|  |
| --- |
| new Vue({  el: '#example-2',  data: {  show: true  }  }) |

|  |
| --- |
| .bounce-enter-active {  animation: bounce-in .5s;  }  .bounce-leave-active {  animation: bounce-in .5s reverse;  }  @keyframes bounce-in {  0% {  transform: scale(0);  }  50% {  transform: scale(1.5);  }  100% {  transform: scale(1);  }  } |

Toggle show

Look at me!

### 自定义过渡的类名

我们可以通过以下特性来自定义过渡类名：

* enter-class
* enter-active-class
* enter-to-class (2.1.8+)
* leave-class
* leave-active-class
* leave-to-class (2.1.8+)

他们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库，如 [**Animate.css**](https://daneden.github.io/animate.css/) 结合使用十分有用。

示例：

|  |
| --- |
| <link href="https://cdn.jsdelivr.net/npm/animate.css@3.5.1" rel="stylesheet" type="text/css">  <div id="example-3">  <button @click="show = !show">  Toggle render  </button>  <transition  name="custom-classes-transition"  enter-active-class="animated tada"  leave-active-class="animated bounceOutRight"  >  <p v-if="show">hello</p>  </transition>  </div> |

|  |
| --- |
| new Vue({  el: '#example-3',  data: {  show: true  }  }) |

Toggle render

hello

### 同时使用过渡和动画

Vue 为了知道过渡的完成，必须设置相应的事件监听器。它可以是 transitionend 或 animationend ，这取决于给元素应用的 CSS 规则。如果你使用其中任何一种，Vue 能自动识别类型并设置监听。

但是，在一些场景中，你需要给同一个元素同时设置两种过渡动效，比如 animation 很快的被触发并完成了，而 transition 效果还没结束。在这种情况中，你就需要使用 type特性并设置 animation 或 transition 来明确声明你需要 Vue 监听的类型。

### 显性的过渡持续时间

**2.2.0 新增**

在很多情况下，Vue 可以自动得出过渡效果的完成时机。默认情况下，Vue 会等待其在过渡效果的根元素的第一个 transitionend 或 animationend 事件。然而也可以不这样设定——比如，我们可以拥有一个精心编排的一序列过渡效果，其中一些嵌套的内部元素相比于过渡效果的根元素有延迟的或更长的过渡效果。

在这种情况下你可以用 <transition> 组件上的 duration 属性定制一个显性的过渡持续时间 (以毫秒计)：

|  |
| --- |
| <transition :duration="1000">...</transition> |

你也可以定制进入和移出的持续时间：

|  |
| --- |
| <transition :duration="{ enter: 500, leave: 800 }">...</transition> |

### JavaScript 钩子

可以在属性中声明 JavaScript 钩子

|  |
| --- |
| <transition  v-on:before-enter="beforeEnter"  v-on:enter="enter"  v-on:after-enter="afterEnter"  v-on:enter-cancelled="enterCancelled"  v-on:before-leave="beforeLeave"  v-on:leave="leave"  v-on:after-leave="afterLeave"  v-on:leave-cancelled="leaveCancelled"  >  <!-- ... -->  </transition> |

|  |
| --- |
| // ...  methods: {  // --------  // 进入中  // --------  beforeEnter: function (el) {  // ...  },  // 此回调函数是可选项的设置  // 与 CSS 结合时使用  enter: function (el, done) {  // ...  done()  },  afterEnter: function (el) {  // ...  },  enterCancelled: function (el) {  // ...  },  // --------  // 离开时  // --------  beforeLeave: function (el) {  // ...  },  // 此回调函数是可选项的设置  // 与 CSS 结合时使用  leave: function (el, done) {  // ...  done()  },  afterLeave: function (el) {  // ...  },  // leaveCancelled 只用于 v-show 中  leaveCancelled: function (el) {  // ...  }  } |

这些钩子函数可以结合 CSS transitions/animations 使用，也可以单独使用。

当只用 JavaScript 过渡的时候，**在 enter 和 leave 中，回调函数 done 是必须的**。否则，它们会被同步调用，过渡会立即完成。

推荐对于仅使用 JavaScript 过渡的元素添加 v-bind:css="false"，Vue 会跳过 CSS 的检测。这也可以避免过渡过程中 CSS 的影响。

一个使用 Velocity.js 的简单例子：

|  |
| --- |
| <!--  Velocity works very much like jQuery.animate and is  a great option for JavaScript animations  -->  <script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>  <div id="example-4">  <button @click="show = !show">  Toggle  </button>  <transition  v-on:before-enter="beforeEnter"  v-on:enter="enter"  v-on:leave="leave"  v-bind:css="false"  >  <p v-if="show">  Demo  </p>  </transition>  </div> |

|  |
| --- |
| new Vue({  el: '#example-4',  data: {  show: false  },  methods: {  beforeEnter: function (el) {  el.style.opacity = 0  el.style.transformOrigin = 'left'  },  enter: function (el, done) {  Velocity(el, { opacity: 1, fontSize: '1.4em' }, { duration: 300 })  Velocity(el, { fontSize: '1em' }, { complete: done })  },  leave: function (el, done) {  Velocity(el, { translateX: '15px', rotateZ: '50deg' }, { duration: 600 })  Velocity(el, { rotateZ: '100deg' }, { loop: 2 })  Velocity(el, {  rotateZ: '45deg',  translateY: '30px',  translateX: '30px',  opacity: 0  }, { complete: done })  }  }  }) |

Toggle

## 初始渲染的过渡

可以通过 appear 特性设置节点的在初始渲染的过渡

|  |
| --- |
| <transition appear>  <!-- ... -->  </transition> |

这里默认和进入和离开过渡一样，同样也可以自定义 CSS 类名。

|  |
| --- |
| <transition  appear  appear-class="custom-appear-class"  appear-to-class="custom-appear-to-class" (2.1.8+)  appear-active-class="custom-appear-active-class"  >  <!-- ... -->  </transition> |

自定义 JavaScript 钩子：

|  |
| --- |
| <transition  appear  v-on:before-appear="customBeforeAppearHook"  v-on:appear="customAppearHook"  v-on:after-appear="customAfterAppearHook"  v-on:appear-cancelled="customAppearCancelledHook"  >  <!-- ... -->  </transition> |

## 多个元素的过渡

我们之后讨论[**多个组件的过渡**](https://cn.vuejs.org/v2/guide/transitions.html#多个组件的过渡)，对于原生标签可以使用 v-if/v-else 。最常见的多标签过渡是一个列表和描述这个列表为空消息的元素：

|  |
| --- |
| <transition>  <table v-if="items.length > 0">  <!-- ... -->  </table>  <p v-else>Sorry, no items found.</p>  </transition> |

可以这样使用，但是有一点需要注意：

当有**相同标签名**的元素切换时，需要通过 key 特性设置唯一的值来标记以让 Vue 区分它们，否则 Vue 为了效率只会替换相同标签内部的内容。即使在技术上没有必要，**给在 <transition> 组件中的多个元素设置 key 是一个更好的实践。**

示例：

|  |
| --- |
| <transition>  <button v-if="isEditing" key="save">  Save  </button>  <button v-else key="edit">  Edit  </button>  </transition> |

在一些场景中，也可以给通过给同一个元素的 key 特性设置不同的状态来代替 v-if 和 v-else，上面的例子可以重写为：

|  |
| --- |
| <transition>  <button v-bind:key="isEditing">  {{ isEditing ? 'Save' : 'Edit' }}  </button>  </transition> |

使用多个 v-if 的多个元素的过渡可以重写为绑定了动态属性的单个元素过渡。例如：

|  |
| --- |
| <transition>  <button v-if="docState === 'saved'" key="saved">  Edit  </button>  <button v-if="docState === 'edited'" key="edited">  Save  </button>  <button v-if="docState === 'editing'" key="editing">  Cancel  </button>  </transition> |

可以重写为：

|  |
| --- |
| <transition>  <button v-bind:key="docState">  {{ buttonMessage }}  </button>  </transition> |

|  |
| --- |
| // ...  computed: {  buttonMessage: function () {  switch (this.docState) {  case 'saved': return 'Edit'  case 'edited': return 'Save'  case 'editing': return 'Cancel'  }  }  } |

### 过渡模式

这里还有一个问题，试着点击下面的按钮：

off

在 “on” 按钮和 “off” 按钮的过渡中，两个按钮都被重绘了，一个离开过渡的时候另一个开始进入过渡。这是 <transition> 的默认行为 - 进入和离开同时发生。

在元素绝对定位在彼此之上的时候运行正常：

off

然后，我们加上 translate 让它们运动像滑动过渡：

off

同时生效的进入和离开的过渡不能满足所有要求，所以 Vue 提供了 **过渡模式**

* in-out：新元素先进行过渡，完成之后当前元素过渡离开。
* out-in：当前元素先进行过渡，完成之后新元素过渡进入。

用 out-in 重写之前的开关按钮过渡：

|  |
| --- |
| <transition name="fade" mode="out-in">  <!-- ... the buttons ... -->  </transition> |

off

只用添加一个简单的特性，就解决了之前的过渡问题而无需任何额外的代码。

in-out 模式不是经常用到，但对于一些稍微不同的过渡效果还是有用的。  
将之前滑动淡出的例子结合：

off

很酷吧？

## 多个组件的过渡

多个组件的过渡简单很多 - 我们不需要使用 key 特性。相反，我们只需要使用[**动态组件**](https://cn.vuejs.org/v2/guide/components.html#动态组件)：

|  |
| --- |
| <transition name="component-fade" mode="out-in">  <component v-bind:is="view"></component>  </transition> |

|  |
| --- |
| new Vue({  el: '#transition-components-demo',  data: {  view: 'v-a'  },  components: {  'v-a': {  template: '<div>Component A</div>'  },  'v-b': {  template: '<div>Component B</div>'  }  }  }) |

|  |
| --- |
| .component-fade-enter-active, .component-fade-leave-active {  transition: opacity .3s ease;  }  .component-fade-enter, .component-fade-leave-to  /\* .component-fade-leave-active for below version 2.1.8 \*/ {  opacity: 0;  } |

A B

Component A

## 列表过渡

目前为止，关于过渡我们已经讲到：

* 单个节点
* 同一时间渲染多个节点中的一个

那么怎么同时渲染整个列表，比如使用 v-for ？在这种场景中，使用 <transition-group> 组件。在我们深入例子之前，先了解关于这个组件的几个特点：

* 不同于 <transition>，它会以一个真实元素呈现：默认为一个 <span>。你也可以通过 tag 特性更换为其他元素。
* 内部元素 **总是需要** 提供唯一的 key 属性值

### 列表的进入/离开过渡

现在让我们由一个简单的例子深入，进入和离开的过渡使用之前一样的 CSS 类名。

|  |
| --- |
| <div id="list-demo" class="demo">  <button v-on:click="add">Add</button>  <button v-on:click="remove">Remove</button>  <transition-group name="list" tag="p">  <span v-for="item in items" v-bind:key="item" class="list-item">  {{ item }}  </span>  </transition-group>  </div> |

|  |
| --- |
| new Vue({  el: '#list-demo',  data: {  items: [1,2,3,4,5,6,7,8,9],  nextNum: 10  },  methods: {  randomIndex: function () {  return Math.floor(Math.random() \* this.items.length)  },  add: function () {  this.items.splice(this.randomIndex(), 0, this.nextNum++)  },  remove: function () {  this.items.splice(this.randomIndex(), 1)  },  }  }) |

|  |
| --- |
| .list-item {  display: inline-block;  margin-right: 10px;  }  .list-enter-active, .list-leave-active {  transition: all 1s;  }  .list-enter, .list-leave-to  /\* .list-leave-active for below version 2.1.8 \*/ {  opacity: 0;  transform: translateY(30px);  } |

Add Remove

123456789

这个例子有个问题，当添加和移除元素的时候，周围的元素会瞬间移动到他们的新布局的位置，而不是平滑的过渡，我们下面会解决这个问题。

### 列表的排序过渡

<transition-group> 组件还有一个特殊之处。不仅可以进入和离开动画，还可以改变定位。要使用这个新功能只需了解新增的 **v-move 特性**，它会在元素的改变定位的过程中应用。像之前的类名一样，可以通过 name 属性来自定义前缀，也可以通过 move-class 属性手动设置。

v-move 对于设置过渡的切换时机和过渡曲线非常有用，你会看到如下的例子：

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script>  <div id="flip-list-demo" class="demo">  <button v-on:click="shuffle">Shuffle</button>  <transition-group name="flip-list" tag="ul">  <li v-for="item in items" v-bind:key="item">  {{ item }}  </li>  </transition-group>  </div> |

|  |
| --- |
| new Vue({  el: '#flip-list-demo',  data: {  items: [1,2,3,4,5,6,7,8,9]  },  methods: {  shuffle: function () {  this.items = \_.shuffle(this.items)  }  }  }) |

|  |
| --- |
| .flip-list-move {  transition: transform 1s;  } |

Shuffle

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

这个看起来很神奇，内部的实现，Vue 使用了一个叫 [**FLIP**](https://aerotwist.com/blog/flip-your-animations/) 简单的动画队列  
使用 transforms 将元素从之前的位置平滑过渡新的位置。

我们将之前实现的例子和这个技术结合，使我们列表的一切变动都会有动画过渡。

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js"></script>  <div id="list-complete-demo" class="demo">  <button v-on:click="shuffle">Shuffle</button>  <button v-on:click="add">Add</button>  <button v-on:click="remove">Remove</button>  <transition-group name="list-complete" tag="p">  <span  v-for="item in items"  v-bind:key="item"  class="list-complete-item"  >  {{ item }}  </span>  </transition-group>  </div> |

|  |
| --- |
| new Vue({  el: '#list-complete-demo',  data: {  items: [1,2,3,4,5,6,7,8,9],  nextNum: 10  },  methods: {  randomIndex: function () {  return Math.floor(Math.random() \* this.items.length)  },  add: function () {  this.items.splice(this.randomIndex(), 0, this.nextNum++)  },  remove: function () {  this.items.splice(this.randomIndex(), 1)  },  shuffle: function () {  this.items = \_.shuffle(this.items)  }  }  }) |

|  |
| --- |
| .list-complete-item {  transition: all 1s;  display: inline-block;  margin-right: 10px;  }  .list-complete-enter, .list-complete-leave-to  /\* .list-complete-leave-active for below version 2.1.8 \*/ {  opacity: 0;  transform: translateY(30px);  }  .list-complete-leave-active {  position: absolute;  } |

Shuffle Add Remove

123456789

需要注意的是使用 FLIP 过渡的元素不能设置为 display: inline 。作为替代方案，可以设置为 display: inline-block 或者放置于 flex 中

FLIP 动画不仅可以实现单列过渡，多维网格也[**同样可以过渡**](https://jsfiddle.net/chrisvfritz/sLrhk1bc/)：

**Lazy Sudoku**

Keep hitting the shuffle button until you win.

Shuffle

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

1

2

3

4

5

6

7

8

9

### 列表的交错过渡

通过 data 属性与 JavaScript 通信 ，就可以实现列表的交错过渡：

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>  <div id="staggered-list-demo">  <input v-model="query">  <transition-group  name="staggered-fade"  tag="ul"  v-bind:css="false"  v-on:before-enter="beforeEnter"  v-on:enter="enter"  v-on:leave="leave"  >  <li  v-for="(item, index) in computedList"  v-bind:key="item.msg"  v-bind:data-index="index"  >{{ item.msg }}</li>  </transition-group>  </div> |

|  |
| --- |
| new Vue({  el: '#staggered-list-demo',  data: {  query: '',  list: [  { msg: 'Bruce Lee' },  { msg: 'Jackie Chan' },  { msg: 'Chuck Norris' },  { msg: 'Jet Li' },  { msg: 'Kung Fury' }  ]  },  computed: {  computedList: function () {  var vm = this  return this.list.filter(function (item) {  return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !== -1  })  }  },  methods: {  beforeEnter: function (el) {  el.style.opacity = 0  el.style.height = 0  },  enter: function (el, done) {  var delay = el.dataset.index \* 150  setTimeout(function () {  Velocity(  el,  { opacity: 1, height: '1.6em' },  { complete: done }  )  }, delay)  },  leave: function (el, done) {  var delay = el.dataset.index \* 150  setTimeout(function () {  Velocity(  el,  { opacity: 0, height: 0 },  { complete: done }  )  }, delay)  }  }  }) |



* Bruce Lee
* Jackie Chan
* Chuck Norris
* Jet Li
* Kung Fury

## 可复用的过渡

过渡可以通过 Vue 的组件系统实现复用。要创建一个可复用过渡组件，你需要做的就是将 <transition> 或者 <transition-group> 作为根组件，然后将任何子组件放置在其中就可以了。

使用 template 的简单例子：

|  |
| --- |
| Vue.component('my-special-transition', {  template: '\  <transition\  name="very-special-transition"\  mode="out-in"\  v-on:before-enter="beforeEnter"\  v-on:after-enter="afterEnter"\  >\  <slot></slot>\  </transition>\  ',  methods: {  beforeEnter: function (el) {  // ...  },  afterEnter: function (el) {  // ...  }  }  }) |

函数组件更适合完成这个任务：

|  |
| --- |
| Vue.component('my-special-transition', {  functional: true,  render: function (createElement, context) {  var data = {  props: {  name: 'very-special-transition',  mode: 'out-in'  },  on: {  beforeEnter: function (el) {  // ...  },  afterEnter: function (el) {  // ...  }  }  }  return createElement('transition', data, context.children)  }  }) |

## 动态过渡

在 Vue 中即使是过渡也是数据驱动的！动态过渡最基本的例子是通过 name 特性来绑定动态值。

|  |
| --- |
| <transition v-bind:name="transitionName">  <!-- ... -->  </transition> |

当你想用 Vue 的过渡系统来定义的 CSS 过渡/动画 在不同过渡间切换会非常有用。

所有的过渡特性都是动态绑定。它不仅是简单的特性，通过事件的钩子函数方法，可以在获取到相应上下文数据。这意味着，可以根据组件的状态通过 JavaScript 过渡设置不同的过渡效果。

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js"></script>  <div id="dynamic-fade-demo" class="demo">  Fade In: <input type="range" v-model="fadeInDuration" min="0" v-bind:max="maxFadeDuration">  Fade Out: <input type="range" v-model="fadeOutDuration" min="0" v-bind:max="maxFadeDuration">  <transition  v-bind:css="false"  v-on:before-enter="beforeEnter"  v-on:enter="enter"  v-on:leave="leave"  >  <p v-if="show">hello</p>  </transition>  <button  v-if="stop"  v-on:click="stop = false; show = false"  >Start animating</button>  <button  v-else  v-on:click="stop = true"  >Stop it!</button>  </div> |

|  |
| --- |
| new Vue({  el: '#dynamic-fade-demo',  data: {  show: true,  fadeInDuration: 1000,  fadeOutDuration: 1000,  maxFadeDuration: 1500,  stop: true  },  mounted: function () {  this.show = false  },  methods: {  beforeEnter: function (el) {  el.style.opacity = 0  },  enter: function (el, done) {  var vm = this  Velocity(el,  { opacity: 1 },  {  duration: this.fadeInDuration,  complete: function () {  done()  if (!vm.stop) vm.show = false  }  }  )  },  leave: function (el, done) {  var vm = this  Velocity(el,  { opacity: 0 },  {  duration: this.fadeOutDuration,  complete: function () {  done()  vm.show = true  }  }  )  }  }  }) |

Fade In:  Fade Out:

hello

Start animating

最后，创建动态过渡的最终方案是组件通过接受 props 来动态修改之前的过渡。一句老话，唯一的限制是你的想象力。

# 状态过渡

Vue 的过渡系统提供了非常多简单的方法设置进入、离开和列表的动效。那么对于数据元素本身的动效呢，比如：

* 数字和运算
* 颜色的显示
* SVG 节点的位置
* 元素的大小和其他的属性

所有的原始数字都被事先存储起来，可以直接转换到数字。做到这一步，我们就可以结合 Vue 的响应式和组件系统，使用第三方库来实现切换元素的过渡状态。

## 状态动画与观察者

通过观察者我们能监听到任何数值属性的数值更新。可能听起来很抽象，所以让我们先来看看使用 [**Tweenjs**](https://github.com/tweenjs/tween.js) 一个例子：

|  |
| --- |
| <script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>  <div id="animated-number-demo">  <input v-model.number="number" type="number" step="20">  <p>{{ animatedNumber }}</p>  </div> |

|  |
| --- |
| new Vue({  el: '#animated-number-demo',  data: {  number: 0,  animatedNumber: 0  },  watch: {  number: function(newValue, oldValue) {  var vm = this  function animate () {  if (TWEEN.update()) {  requestAnimationFrame(animate)  }  }  new TWEEN.Tween({ tweeningNumber: oldValue })  .easing(TWEEN.Easing.Quadratic.Out)  .to({ tweeningNumber: newValue }, 500)  .onUpdate(function () {  vm.animatedNumber = this.tweeningNumber.toFixed(0)  })  .start()  animate()  }  }  }) |

0

当你把数值更新时，就会触发动画。这个是一个不错的演示，但是对于不能直接像数字一样存储的值，比如 CSS 中的 color 的值，通过下面的例子我们来通过 [**Color.js**](https://github.com/brehaut/color-js) 实现一个例子：

|  |
| --- |
| <script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>  <script src="https://cdn.jsdelivr.net/npm/color-js@1.0.3"></script>  <div id="example-7">  <input  v-model="colorQuery"  v-on:keyup.enter="updateColor"  placeholder="Enter a color"  >  <button v-on:click="updateColor">Update</button>  <p>Preview:</p>  <span  v-bind:style="{ backgroundColor: tweenedCSSColor }"  class="example-7-color-preview"  ></span>  <p>{{ tweenedCSSColor }}</p>  </div> |

|  |
| --- |
| var Color = net.brehaut.Color  new Vue({  el: '#example-7',  data: {  colorQuery: '',  color: {  red: 0,  green: 0,  blue: 0,  alpha: 1  },  tweenedColor: {}  },  created: function () {  this.tweenedColor = Object.assign({}, this.color)  },  watch: {  color: function () {  function animate () {  if (TWEEN.update()) {  requestAnimationFrame(animate)  }  }  new TWEEN.Tween(this.tweenedColor)  .to(this.color, 750)  .start()  animate()  }  },  computed: {  tweenedCSSColor: function () {  return new Color({  red: this.tweenedColor.red,  green: this.tweenedColor.green,  blue: this.tweenedColor.blue,  alpha: this.tweenedColor.alpha  }).toCSS()  }  },  methods: {  updateColor: function () {  this.color = new Color(this.colorQuery).toRGB()  this.colorQuery = ''  }  }  }) |

|  |
| --- |
| .example-7-color-preview {  display: inline-block;  width: 50px;  height: 50px;  } |

 Update

Preview:

#000000

## 动态状态过渡

就像 Vue 的过渡组件一样，数据背后状态过渡会实时更新，这对于原型设计十分有用。当你修改一些变量，即使是一个简单的 SVG 多边形也可实现很多难以想象的效果。

Sides: 10Minimum Radius: 50%Update Interval: 500 milliseconds

上述 demo 背后的代码可以通过[**这个 fiddle**](https://jsfiddle.net/chrisvfritz/65gLu2b6/) 进行详阅。

## 把过渡放到组件里

管理太多的状态过渡会很快的增加 Vue 实例或者组件的复杂性，幸好很多的动画可以提取到专用的子组件。  
我们来将之前的示例改写一下：

|  |
| --- |
| <script src="https://cdn.jsdelivr.net/npm/tween.js@16.3.4"></script>  <div id="example-8">  <input v-model.number="firstNumber" type="number" step="20"> +  <input v-model.number="secondNumber" type="number" step="20"> =  {{ result }}  <p>  <animated-integer v-bind:value="firstNumber"></animated-integer> +  <animated-integer v-bind:value="secondNumber"></animated-integer> =  <animated-integer v-bind:value="result"></animated-integer>  </p>  </div> |

|  |
| --- |
| // 这种复杂的补间动画逻辑可以被复用  // 任何整数都可以执行动画  // 组件化使我们的界面十分清晰  // 可以支持更多更复杂的动态过渡  // 策略。  Vue.component('animated-integer', {  template: '<span>{{ tweeningValue }}</span>',  props: {  value: {  type: Number,  required: true  }  },  data: function () {  return {  tweeningValue: 0  }  },  watch: {  value: function (newValue, oldValue) {  this.tween(oldValue, newValue)  }  },  mounted: function () {  this.tween(0, this.value)  },  methods: {  tween: function (startValue, endValue) {  var vm = this  function animate () {  if (TWEEN.update()) {  requestAnimationFrame(animate)  }  }  new TWEEN.Tween({ tweeningValue: startValue })  .to({ tweeningValue: endValue }, 500)  .onUpdate(function () {  vm.tweeningValue = this.tweeningValue.toFixed(0)  })  .start()  animate()  }  }  })  // 所有的复杂度都已经从 Vue 的主实例中移除！  new Vue({  el: '#example-8',  data: {  firstNumber: 20,  secondNumber: 40  },  computed: {  result: function () {  return this.firstNumber + this.secondNumber  }  }  }) |

 +  = 60

20 + 40 = 60

我们能在组件中结合使用这一节讲到各种过渡策略和 Vue [**内建的过渡系统**](https://cn.vuejs.org/v2/guide/transitions.html)。总之，对于完成各种过渡动效几乎没有阻碍。

## 赋予设计以生命

只要一个动画，就可以带来生命。不幸的是，当设计师创建图标、logo 和吉祥物的时候，他们交付的通常都是图片或静态的 SVG。所以，虽然 GitHub 的章鱼猫、Twitter 的小鸟以及其它许多 logo 类似于生灵，它们看上去实际上并不是活着的。

Vue 可以帮到你。因为 SVG 的本质是数据，我们只需要这些动物兴奋、思考或境界的样例。然后 Vue 就可以辅助完成这几种状态之间的过渡动画，来制作你的欢迎页面、加载指示、以及更加带有情感的提示。

Sarah Drasner 展示了下面这个 demo，这个 demo 结合了时间和交互相关的状态改变：

### 可复用性 & 组合

# 混合

## 基础

混合 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混合对象可以包含任意组件选项。以组件使用混合对象时，所有混合对象的选项将被混入该组件本身的选项。

例子：

|  |
| --- |
| // 定义一个混合对象  var myMixin = {  created: function () {  this.hello()  },  methods: {  hello: function () {  console.log('hello from mixin!')  }  }  }  // 定义一个使用混合对象的组件  var Component = Vue.extend({  mixins: [myMixin]  })  var component = new Component() // => "hello from mixin!" |

## 选项合并

当组件和混合对象含有同名选项时，这些选项将以恰当的方式混合。比如，同名钩子函数将混合为一个数组，因此都将被调用。另外，混合对象的 钩子将在组件自身钩子 **之前** 调用 ：

|  |
| --- |
| var mixin = {  created: function () {  console.log('混合对象的钩子被调用')  }  }  new Vue({  mixins: [mixin],  created: function () {  console.log('组件钩子被调用')  }  })  // => "混合对象的钩子被调用"  // => "组件钩子被调用" |

值为对象的选项，例如 methods, components 和 directives，将被混合为同一个对象。两个对象键名冲突时，取组件对象的键值对。

|  |
| --- |
| var mixin = {  methods: {  foo: function () {  console.log('foo')  },  conflicting: function () {  console.log('from mixin')  }  }  }  var vm = new Vue({  mixins: [mixin],  methods: {  bar: function () {  console.log('bar')  },  conflicting: function () {  console.log('from self')  }  }  })  vm.foo() // => "foo"  vm.bar() // => "bar"  vm.conflicting() // => "from self" |

注意：Vue.extend() 也使用同样的策略进行合并。

## 全局混合

也可以全局注册混合对象。注意使用！ 一旦使用全局混合对象，将会影响到 **所有** 之后创建的 Vue 实例。使用恰当时，可以为自定义对象注入处理逻辑。

|  |
| --- |
| // 为自定义的选项 'myOption' 注入一个处理器。  Vue.mixin({  created: function () {  var myOption = this.$options.myOption  if (myOption) {  console.log(myOption)  }  }  })  new Vue({  myOption: 'hello!'  })  // => "hello!" |

谨慎使用全局混合对象，因为会影响到每个单独创建的 Vue 实例 (包括第三方模板)。大多数情况下，只应当应用于自定义选项，就像上面示例一样。也可以将其用作 [**Plugins**](https://cn.vuejs.org/v2/guide/plugins.html) 以避免产生重复应用

## 自定义选项合并策略

自定义选项将使用默认策略，即简单地覆盖已有值。如果想让自定义选项以自定义逻辑合并，可以向 Vue.config.optionMergeStrategies 添加一个函数：

|  |
| --- |
| Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal) {  // return mergedVal  } |

对于大多数对象选项，可以使用 methods 的合并策略：

|  |
| --- |
| var strategies = Vue.config.optionMergeStrategies  strategies.myOption = strategies.methods |

更多高级的例子可以在 [**Vuex**](https://github.com/vuejs/vuex) 的 1.x 混合策略里找到：

|  |
| --- |
| const merge = Vue.config.optionMergeStrategies.computed  Vue.config.optionMergeStrategies.vuex = function (toVal, fromVal) {  if (!toVal) return fromVal  if (!fromVal) return toVal  return {  getters: merge(toVal.getters, fromVal.getters),  state: merge(toVal.state, fromVal.state),  actions: merge(toVal.actions, fromVal.actions)  }  } |

# 自定义指令

## 简介

除了默认设置的核心指令 (v-model 和 v-show)，Vue 也允许注册自定义指令。注意，在 Vue2.0 里面，代码复用的主要形式和抽象是组件——然而，有的情况下，你仍然需要对纯 DOM 元素进行底层操作，这时候就会用到自定义指令。下面这个例子将聚焦一个 input 元素，像这样：



当页面加载时，元素将获得焦点 (注意：autofocus 在移动版 Safari 上不工作)。事实上，你访问后还没点击任何内容，input 就获得了焦点。现在让我们完善这个指令：

|  |
| --- |
| // 注册一个全局自定义指令 v-focus  Vue.directive('focus', {  // 当绑定元素插入到 DOM 中。  inserted: function (el) {  // 聚焦元素  el.focus()  }  }) |

也可以注册局部指令，组件中接受一个 directives 的选项：

|  |
| --- |
| directives: {  focus: {  // 指令的定义---  inserted: function (el) {}  }  } |

然后你可以在模板中任何元素上使用新的 v-focus 属性：

|  |
| --- |
| <input v-focus> |

## 钩子函数

指令定义函数提供了几个钩子函数 (可选)：

* bind：只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。
* inserted：被绑定元素插入父节点时调用 (父节点存在即可调用，不必存在于 document 中)。
* update：所在组件的 VNode 更新时调用，**但是可能发生在其孩子的 VNode 更新之前**。指令的值可能发生了改变也可能没有。但是你可以通过比较更新前后的值来忽略不必要的模板更新 (详细的钩子函数参数见下)。
* componentUpdated：所在组件的 VNode **及其孩子的 VNode** 全部更新时调用。
* unbind：只调用一次，指令与元素解绑时调用。

接下来我们来看一下钩子函数的参数 (包括 el，binding，vnode，oldVnode)。

## 钩子函数参数

钩子函数被赋予了以下参数：

* **el**：指令所绑定的元素，可以用来直接操作 DOM 。
* **binding**：一个对象，包含以下属性：
  + **name**：指令名，不包括 v- 前缀。
  + **value**：指令的绑定值，例如：v-my-directive="1 + 1", value 的值是 2。
  + **oldValue**：指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。
  + **expression**：绑定值的字符串形式。例如 v-my-directive="1 + 1" ，expression 的值是 "1 + 1"。
  + **arg**：传给指令的参数。例如 v-my-directive:foo，arg 的值是 "foo"。
  + **modifiers**：一个包含修饰符的对象。例如：v-my-directive.foo.bar, 修饰符对象 modifiers 的值是 { foo: true, bar: true }。
* **vnode**：Vue 编译生成的虚拟节点，查阅 [**VNode API**](https://cn.vuejs.org/v2/api/#VNode-接口) 了解更多详情。
* **oldVnode**：上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。

除了 el 之外，其它参数都应该是只读的，尽量不要修改他们。如果需要在钩子之间共享数据，建议通过元素的 [**dataset**](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset) 来进行。

一个使用了这些参数的自定义钩子样例：

|  |
| --- |
| <div id="hook-arguments-example" v-demo:foo.a.b="message"></div> |

|  |
| --- |
| Vue.directive('demo', {  bind: function (el, binding, vnode) {  var s = JSON.stringify  el.innerHTML =  'name: ' + s(binding.name) + '<br>' +  'value: ' + s(binding.value) + '<br>' +  'expression: ' + s(binding.expression) + '<br>' +  'argument: ' + s(binding.arg) + '<br>' +  'modifiers: ' + s(binding.modifiers) + '<br>' +  'vnode keys: ' + Object.keys(vnode).join(', ')  }  })  new Vue({  el: '#hook-arguments-example',  data: {  message: 'hello!'  }  }) |

name: "demo"  
value: "hello!"  
expression: "message"  
argument: "foo"  
modifiers: {"a":true,"b":true}  
vnode keys: tag, data, children, text, elm, ns, context, functionalContext, key, componentOptions, componentInstance, parent, raw, isStatic, isRootInsert, isComment, isCloned, isOnce, asyncFactory, asyncMeta, isAsyncPlaceholder

## 函数简写

大多数情况下，我们可能想在 bind 和 update 钩子上做重复动作，并且不想关心其它的钩子函数。可以这样写:

|  |
| --- |
| Vue.directive('color-swatch', function (el, binding) {  el.style.backgroundColor = binding.value  }) |

## 对象字面量

如果指令需要多个值，可以传入一个 JavaScript 对象字面量。记住，指令函数能够接受所有合法类型的 JavaScript 表达式。

|  |
| --- |
| <div v-demo="{ color: 'white', text: 'hello!' }"></div> |

|  |
| --- |
| Vue.directive('demo', function (el, binding) {  console.log(binding.value.color) // => "white"  console.log(binding.value.text) // => "hello!"  }) |

# 渲染函数 & JSX

## 基础

Vue 推荐在绝大多数情况下使用 template 来创建你的 HTML。然而在一些场景中，你真的需要 JavaScript 的完全编程的能力，这就是 **render 函数**，它比 template 更接近编译器。

|  |
| --- |
| <h1>  <a name="hello-world" href="#hello-world">  Hello world!  </a>  </h1> |

在 HTML 层，我们决定这样定义组件接口：

|  |
| --- |
| <anchored-heading :level="1">Hello world!</anchored-heading> |

当我们开始写一个通过 level prop 动态生成 heading 标签的组件，你可能很快想到这样实现：

|  |
| --- |
| <script type="text/x-template" id="anchored-heading-template">  <h1 v-if="level === 1">  <slot></slot>  </h1>  <h2 v-else-if="level === 2">  <slot></slot>  </h2>  <h3 v-else-if="level === 3">  <slot></slot>  </h3>  <h4 v-else-if="level === 4">  <slot></slot>  </h4>  <h5 v-else-if="level === 5">  <slot></slot>  </h5>  <h6 v-else-if="level === 6">  <slot></slot>  </h6>  </script> |

|  |
| --- |
| Vue.component('anchored-heading', {  template: '#anchored-heading-template',  props: {  level: {  type: Number,  required: true  }  }  }) |

在这种场景中使用 template 并不是最好的选择：首先代码冗长，为了在不同级别的标题中插入锚点元素，我们需要重复地使用 <slot></slot>。

虽然模板在大多数组件中都非常好用，但是在这里它就不是很简洁的了。那么，我们来尝试使用 render 函数重写上面的例子：

|  |
| --- |
| Vue.component('anchored-heading', {  render: function (createElement) {  return createElement(  'h' + this.level, // tag name 标签名称  this.$slots.default // 子组件中的阵列  )  },  props: {  level: {  type: Number,  required: true  }  }  }) |

简单清晰很多！简单来说，这样代码精简很多，但是需要非常熟悉 Vue 的实例属性。在这个例子中，你需要知道当你不使用 slot 属性向组件中传递内容时，比如 anchored-heading 中的 Hello world!，这些子元素被存储在组件实例中的 $slots.default中。如果你还不了解，**在深入 render 函数之前推荐阅读**[**实例属性 API**](https://cn.vuejs.org/v2/api/#实例属性)**。**

## 节点、树以及虚拟 DOM

在深入渲染函数之前，了解一些浏览器的工作原理是很重要的。以下面这段 HTML 为例：

|  |
| --- |
| <div>  <h1>My title</h1>  Some text content  <!-- TODO: Add tagline -->  </div> |

当浏览器读到这些代码时，它会建立一个[**“DOM 节点”树**](https://javascript.info/dom-nodes)来保持追踪，如同你会画一张家谱树来追踪家庭成员的发展一样。

HTML 的 DOM 节点树如下图所示：

每个元素都是一个节点。每片文字也是一个节点。甚至注释也都是节点。一个节点就是页面的一个部分。就像家谱树一样，每个节点都可以有孩子节点 (也就是说每个部分可以包含其它的一些部分)。

高效的更新所有这些节点会是比较困难的，不过所幸你不必再手动完成这个工作了。你只需要告诉 Vue 你希望页面上的 HTML 是什么，这可以是在一个模板里：

|  |
| --- |
| <h1>{{ blogTitle }}</h1> |

或者一个渲染函数里：

|  |
| --- |
| render: function (createElement) {  return createElement('h1', this.blogTitle)  } |

在这两种情况下，Vue 都会自动保持页面的更新，即便 blogTitle 发生了改变。

### 虚拟 DOM

Vue 通过建立一个**虚拟 DOM** 对真实 DOM 发生的变化保持追踪。请近距离看一下这行代码：

|  |
| --- |
| return createElement('h1', this.blogTitle) |

createElement 到底会返回什么呢？其实不是一个实际的 DOM 元素。它更准确的名字可能是 createNodeDescription，因为它所包含的信息会告诉 Vue 页面上需要渲染什么样的节点，及其子节点。我们把这样的节点描述为“虚拟节点 (Virtual DOM)”，也常简写它为“VNode”。“虚拟 DOM”是我们对由 Vue 组件树建立起来的整个 VNode 树的称呼。

## createElement 参数

接下来你需要熟悉的是如何在 createElement 函数中生成模板。这里是 createElement接受的参数：

|  |
| --- |
| // @returns {VNode}  createElement(  // {String | Object | Function}   // 一个 HTML 标签字符串，组件选项对象，或者一个返回值类型为 String/Object 的函数，必要参数   'div',  // {Object}   // 一个包含模板相关属性的数据对象   // 这样，您可以在 template 中使用这些属性。可选参数。  {     // (详情见下一节)  },  // {String | Array}   // 子节点 (VNodes)，由 `createElement()` 构建而成，  // 或使用字符串来生成“文本节点”。可选参数。  [  '先写一些文字',  createElement('h1', '一则头条'),  createElement(MyComponent, {  props: {  someProp: 'foobar'  }  })  ]  ) |

### 深入 data 对象

有一件事要注意：正如在模板语法中，v-bind:class 和 v-bind:style ，会被特别对待一样，在 VNode 数据对象中，下列属性名是级别最高的字段。该对象也允许你绑定普通的 HTML 特性，就像 DOM 属性一样，比如 innerHTML (这会取代 v-html 指令)。

|  |
| --- |
| {  // 和`v-bind:class`一样的 API  'class': {  foo: true,  bar: false  },  // 和`v-bind:style`一样的 API  style: {  color: 'red',  fontSize: '14px'  },  // 正常的 HTML 特性  attrs: {  id: 'foo'  },  // 组件 props  props: {  myProp: 'bar'  },  // DOM 属性  domProps: {  innerHTML: 'baz'  },  // 事件监听器基于 `on`  // 所以不再支持如 `v-on:keyup.enter` 修饰器  // 需要手动匹配 keyCode。  on: {  click: this.clickHandler  },   // 仅对于组件，用于监听原生事件，而不是组件内部使用 `vm.$emit` 触发的事件。  nativeOn: {  click: this.nativeClickHandler  },  // 自定义指令。注意事项：不能对绑定的旧值设值   // Vue 会为您持续追踪   directives: [  {  name: 'my-custom-directive',  value: '2',  expression: '1 + 1',  arg: 'foo',  modifiers: {  bar: true  }  }  ],  // Scoped slots in the form of  // { name: props => VNode | Array<VNode> }  scopedSlots: {  default: props => createElement('span', props.text)  },   // 如果组件是其他组件的子组件，需为插槽指定名称   slot: 'name-of-slot',  // 其他特殊顶层属性  key: 'myKey',  ref: 'myRef'  } |

### 完整示例

有了这些知识，我们现在可以完成我们最开始想实现的组件：

|  |
| --- |
| var getChildrenTextContent = function (children) {  return children.map(function (node) {  return node.children  ? getChildrenTextContent(node.children)  : node.text  }).join('')  }  Vue.component('anchored-heading', {  render: function (createElement) {  // create kebabCase id  var headingId = getChildrenTextContent(this.$slots.default)  .toLowerCase()  .replace(/\W+/g, '-')  .replace(/(^\-|\-$)/g, '')  return createElement(  'h' + this.level,  [  createElement('a', {  attrs: {  name: headingId,  href: '#' + headingId  }  }, this.$slots.default)  ]  )  },  props: {  level: {  type: Number,  required: true  }  }  }) |

### 约束

#### VNodes 必须唯一

组件树中的所有 VNodes 必须是唯一的。这意味着，下面的 render function 是无效的：

|  |
| --- |
| render: function (createElement) {  var myParagraphVNode = createElement('p', 'hi')  return createElement('div', [     // 错误-重复的 VNodes     myParagraphVNode, myParagraphVNode  ])  } |

如果你真的需要重复很多次的元素/组件，你可以使用工厂函数来实现。例如，下面这个例子 render 函数完美有效地渲染了 20 个重复的段落：

|  |
| --- |
| render: function (createElement) {  return createElement('div',  Array.apply(null, { length: 20 }).map(function () {  return createElement('p', 'hi')  })  )  } |

## 使用 JavaScript 代替模板功能

### v-if 和 v-for

由于使用原生的 JavaScript 来实现某些东西很简单，Vue 的 render 函数没有提供专用的 API。比如，template 中的 v-if 和 v-for：

|  |
| --- |
| <ul v-if="items.length">  <li v-for="item in items">{{ item.name }}</li>  </ul>  <p v-else>No items found.</p> |

这些都会在 render 函数中被 JavaScript 的 if/else 和 map 重写：

|  |
| --- |
| render: function (createElement) {  if (this.items.length) {  return createElement('ul', this.items.map(function (item) {  return createElement('li', item.name)  }))  } else {  return createElement('p', 'No items found.')  }  } |

### v-model

render 函数中没有与 v-model 相应的 api - 你必须自己来实现相应的逻辑：

|  |
| --- |
| render: function (createElement) {  var self = this  return createElement('input', {  domProps: {  value: self.value  },  on: {  input: function (event) {  self.value = event.target.value  self.$emit('input', event.target.value)  }  }  })  } |

这就是深入底层要付出的，尽管麻烦了一些，但相对于 v-model 来说，你可以更灵活地控制。

### 事件 & 按键修饰符

对于 .passive、.capture 和 .once事件修饰符, Vue 提供了相应的前缀可以用于 on：

| **Modifier(s)** | **Prefix** |
| --- | --- |
| .passive | & |
| .capture | ! |
| .once | ~ |
| .capture.once or .once.capture | ~! |

例如:

|  |
| --- |
| on: {  '!click': this.doThisInCapturingMode,  '~keyup': this.doThisOnce,  `~!mouseover`: this.doThisOnceInCapturingMode  } |

对于其他的修饰符，前缀不是很重要，因为你可以在事件处理函数中使用事件方法：

| **Modifier(s)** | **Equivalent in Handler** |
| --- | --- |
| .stop | event.stopPropagation() |
| .prevent | event.preventDefault() |
| .self | if (event.target !== event.currentTarget) return |
| Keys: .enter, .13 | if (event.keyCode !== 13) return (change 13 to [**another key code**](http://keycode.info/) for other key modifiers) |
| Modifiers Keys: .ctrl, .alt, .shift, .meta | if (!event.ctrlKey) return (change ctrlKey to altKey, shiftKey, or metaKey, respectively) |

这里是一个使用所有修饰符的例子：

|  |
| --- |
| on: {  keyup: function (event) {     // 如果触发事件的元素不是事件绑定的元素     // 则返回     if (event.target !== event.currentTarget) return     // 如果按下去的不是 enter 键或者     // 没有同时按下 shift 键     // 则返回     if (!event.shiftKey || event.keyCode !== 13) return     // 阻止 事件冒泡    event.stopPropagation()     // 阻止该元素默认的 keyup 事件     event.preventDefault()  // ...  }  } |

### 插槽

你可以从 [**this.$slots**](https://cn.vuejs.org/v2/api/#vm-slots) 获取 VNodes 列表中的静态内容：

|  |
| --- |
| render: function (createElement) {  // `<div><slot></slot></div>`  return createElement('div', this.$slots.default)  } |

还可以从 [**this.$scopedSlots**](https://cn.vuejs.org/v2/api/#vm-scopedSlots) 中获得能用作函数的作用域插槽，这个函数返回 VNodes：

|  |
| --- |
| render: function (createElement) {  // `<div><slot :text="msg"></slot></div>`  return createElement('div', [  this.$scopedSlots.default({  text: this.msg  })  ])  } |

如果要用渲染函数向子组件中传递作用域插槽，可以利用 VNode 数据中的 scopedSlots域：

|  |
| --- |
| render (createElement) {  return createElement('div', [  createElement('child', {  // pass `scopedSlots` in the data object  // in the form of { name: props => VNode | Array<VNode> }  scopedSlots: {  default: function (props) {  return createElement('span', props.text)  }  }  })  ])  } |

## JSX

如果你写了很多 render 函数，可能会觉得痛苦：

|  |
| --- |
| createElement(  'anchored-heading', {  props: {  level: 1  }  }, [  createElement('span', 'Hello'),  ' world!'  ]  ) |

特别是模板如此简单的情况下：

|  |
| --- |
| <anchored-heading :level="1">  <span>Hello</span> world!  </anchored-heading> |

这就是为什么会有一个 [**Babel 插件**](https://github.com/vuejs/babel-plugin-transform-vue-jsx)，用于在 Vue 中使用 JSX 语法的原因，它可以让我们回到更接近于模板的语法上。

|  |
| --- |
| import AnchoredHeading from './AnchoredHeading.vue'  new Vue({  el: '#demo',  render (h) {  return (  <AnchoredHeading level={1}>  <span>Hello</span> world!  </AnchoredHeading>  )  }  }) |

将 h 作为 createElement 的别名是 Vue 生态系统中的一个通用惯例，实际上也是 JSX 所要求的，如果在作用域中 h 失去作用，在应用中会触发报错。

更多关于 JSX 映射到 JavaScript，阅读 [**使用文档**](https://github.com/vuejs/babel-plugin-transform-vue-jsx#usage)。

## 函数式组件

之前创建的锚点标题组件是比较简单，没有管理或者监听任何传递给他的状态，也没有生命周期方法。它只是一个接收参数的函数。  
在这个例子中，我们标记组件为 functional，这意味它是无状态 (没有 data)，无实例 (没有 this 上下文)。  
一个 **函数式组件** 就像这样：

|  |
| --- |
| Vue.component('my-component', {  functional: true,  // 为了弥补缺少的实例  // 提供第二个参数作为上下文  render: function (createElement, context) {  // ...  },  // Props 可选  props: {  // ...  }  }) |

**注意：在 2.3.0 之前的版本中，如果一个函数式组件想要接受 props，则 props 选项是必须的。在 2.3.0 或以上的版本中，你可以省略 props 选项，所有组件上的属性都会被自动解析为 props。**

组件需要的一切都是通过上下文传递，包括：

* props：提供 props 的对象
* children: VNode 子节点的数组
* slots: slots 对象
* data：传递给组件的 data 对象
* parent：对父组件的引用
* listeners: (2.3.0+) 一个包含了组件上所注册的 v-on 侦听器的对象。这只是一个指向 data.on 的别名。
* injections: (2.3.0+) 如果使用了 [**inject**](https://cn.vuejs.org/v2/api/#provide-inject) 选项，则该对象包含了应当被注入的属性。

在添加 functional: true 之后，锚点标题组件的 render 函数之间简单更新增加 context参数，this.$slots.default 更新为 context.children，之后this.level 更新为 context.props.level。

因为函数式组件只是一个函数，所以渲染开销也低很多。然而，对持久化实例的缺乏也意味着函数式组件不会出现在 [**Vue devtools**](https://github.com/vuejs/vue-devtools) 的组件树里。

在作为包装组件时它们也同样非常有用，比如，当你需要做这些时：

* 程序化地在多个组件中选择一个
* 在将 children, props, data 传递给子组件之前操作它们。

下面是一个依赖传入 props 的值的 smart-list 组件例子，它能代表更多具体的组件：

|  |
| --- |
| var EmptyList = { /\* ... \*/ }  var TableList = { /\* ... \*/ }  var OrderedList = { /\* ... \*/ }  var UnorderedList = { /\* ... \*/ }  Vue.component('smart-list', {  functional: true,  render: function (createElement, context) {  function appropriateListComponent () {  var items = context.props.items  if (items.length === 0) return EmptyList  if (typeof items[0] === 'object') return TableList  if (context.props.isOrdered) return OrderedList  return UnorderedList  }  return createElement(  appropriateListComponent(),  context.data,  context.children  )  },  props: {  items: {  type: Array,  required: true  },  isOrdered: Boolean  }  }) |

### slots() 和 children 对比

你可能想知道为什么同时需要 slots() 和 children。slots().default 不是和 children 类似的吗？在一些场景中，是这样，但是如果是函数式组件和下面这样的 children 呢？

|  |
| --- |
| <my-functional-component>  <p slot="foo">  first  </p>  <p>second</p>  </my-functional-component> |

对于这个组件，children 会给你两个段落标签，而 slots().default 只会传递第二个匿名段落标签，slots().foo 会传递第一个具名段落标签。同时拥有 children 和 slots() ，因此你可以选择让组件通过 slot() 系统分发或者简单的通过 children 接收，让其他组件去处理。

## 模板编译

你可能有兴趣知道，Vue 的模板实际是编译成了 render 函数。这是一个实现细节，通常不需要关心，但如果你想看看模板的功能是怎样被编译的，你会发现会非常有趣。下面是一个使用 Vue.compile 来实时编译模板字符串的简单 demo：



render:

function anonymous() {

with(this){return \_c('div',[\_m(0),(message)?\_c('p',[\_v(\_s(message))]):\_c('p',[\_v("No message.")])])}

}

staticRenderFns:

\_m(0): function anonymous() {

with(this){return \_c('header',[\_c('h1',[\_v("I'm a template!")])])}

}

# 插件

## 开发插件

插件通常会为 Vue 添加全局功能。插件的范围没有限制——一般有下面几种：

1. 添加全局方法或者属性，如: [**vue-custom-element**](https://github.com/karol-f/vue-custom-element)
2. 添加全局资源：指令/过滤器/过渡等，如 [**vue-touch**](https://github.com/vuejs/vue-touch)
3. 通过全局 mixin 方法添加一些组件选项，如: [**vue-router**](https://github.com/vuejs/vue-router)
4. 添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现。
5. 一个库，提供自己的 API，同时提供上面提到的一个或多个功能，如 [**vue-router**](https://github.com/vuejs/vue-router)

Vue.js 的插件应当有一个公开方法 install 。这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象：

|  |
| --- |
| MyPlugin.install = function (Vue, options) {  // 1. 添加全局方法或属性  Vue.myGlobalMethod = function () {  // 逻辑...  }  // 2. 添加全局资源  Vue.directive('my-directive', {  bind (el, binding, vnode, oldVnode) {  // 逻辑...  }  ...  })  // 3. 注入组件  Vue.mixin({  created: function () {  // 逻辑...  }  ...  })  // 4. 添加实例方法  Vue.prototype.$myMethod = function (methodOptions) {  // 逻辑...  }  } |

## 使用插件

通过全局方法 Vue.use() 使用插件：

|  |
| --- |
| // 调用 `MyPlugin.install(Vue)`  Vue.use(MyPlugin) |

也可以传入一个选项对象：

|  |
| --- |
| Vue.use(MyPlugin, { someOption: true }) |

Vue.use 会自动阻止注册相同插件多次，届时只会注册一次该插件。

Vue.js 官方提供的一些插件 (例如 vue-router) 在检测到 Vue 是可访问的全局变量时会自动调用 Vue.use()。然而在例如 CommonJS 的模块环境中，你应该始终显式地调用 Vue.use()：

|  |
| --- |
| // 用 Browserify 或 webpack 提供的 CommonJS 模块环境时  var Vue = require('vue')  var VueRouter = require('vue-router')  // 不要忘了调用此方法  Vue.use(VueRouter) |

[**awesome-vue**](https://github.com/vuejs/awesome-vue#components--libraries) 集合了来自社区贡献的数以千计的插件和库。

# 过滤器

Vue.js 允许你自定义过滤器，可被用作一些常见的文本格式化。过滤器可以用在两个地方：**mustache 插值和 v-bind 表达式** (后者从 2.1.0+ 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符指示：

|  |
| --- |
| <!-- in mustaches -->  {{ message | capitalize }}  <!-- in v-bind -->  <div v-bind:id="rawId | formatId"></div> |

过滤器函数总接收表达式的值 (之前的操作链的结果) 作为第一个参数。在这个例子中，capitalize 过滤器函数将会收到 message 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 }} |

在这个例子中，filterA 被定义为接收单个参数的过滤器函数，表达式 message 的值将作为参数传入到函数中，然后继续调用同样被定义为接收单个参数的过滤器函数 filterB，将 filterA 的结果传递到 filterB 中。

过滤器是 JavaScript 函数，因此可以接收参数：

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

这里，filterA 被定义为接收三个参数的过滤器函数。其中 message 的值作为第一个参数，普通字符串 'arg1' 作为第二个参数，表达式 arg2 取值后的值作为第三个参数。

### 工具

# 生产环境部署

## 开启生产环境模式

开发时，Vue 会提供很多警告来帮你解决常见的错误与陷阱。生产时，这些警告语句却没有用，反而会增加你的载荷量。再次，有些警告检查有小的运行时开销，生产环境模式下是可以避免的。

### 不用打包工具

如果用 Vue 完整独立版本 (直接用 <script> 元素引入 Vue)，生产时应该用精简版本 (vue.min.js)。请查看[**安装指导**](https://cn.vuejs.org/v2/guide/installation.html#直接-lt-script-gt-引入)，附有开发与精简版本。

### 用打包工具

如果用 Webpack 或 Browserify 类似的打包工具时，生产状态会在 Vue 源码中由 process.env.NODE\_ENV 决定，默认在开发状态。Webpack 与 Browserify 两个打包工具都提供方法来覆盖此变量并使用生产状态，警告语句也会被精简掉。每一个 vue-cli 模板有预先配置好的打包工具，但了解怎样配置会更好。

#### Webpack

使用 Webpack 的 [**DefinePlugin**](https://webpack.github.io/docs/list-of-plugins.html#defineplugin) 来指定生产环境，以便在压缩时可以让 UglifyJS 自动删除代码块内的警告语句。例如配置：

|  |
| --- |
| var webpack = require('webpack')  module.exports = {  // ...  plugins: [  // ...  new webpack.DefinePlugin({  'process.env': {  NODE\_ENV: '"production"'  }  }),  new webpack.optimize.UglifyJsPlugin({  compress: {  warnings: false  }  })  ]  } |

#### Browserify

* 运行打包命令，设置 NODE\_ENV 为 "production"。等于告诉 vueify 避免引入热重载和开发相关代码。
* 使用一个全局 [**envify**](https://github.com/hughsk/envify) 转换你的 bundle 文件。这可以精简掉包含在 Vue 源码中所有环境变量条件相关代码块内的警告语句。例如：

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

#### 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(...) |

## 预编译模板

当你需要处理 DOM 内或 JavaScript 内的模板时，“从模板到渲染函数”的编译就会在线上发生。通常情况下这种处理是足够快的，但是如果你的应用对性能很敏感最好还是回避。

预编译模板最简单的方式就是使用[**单文件组件**](https://cn.vuejs.org/v2/guide/single-file-components.html) - 相关的构建设置会自动把预编译处理好，所以构建好的代码已经包含了编译出来的渲染函数而不是原始的模板字符串。

如果你使用 Webpack，并且喜欢分离 JavaScript 和模板文件，你可以使用 [**vue-template-loader**](https://github.com/ktsn/vue-template-loader)，它也可以在构建过程中把模板文件转换成为 JavaScript 渲染函数。

## 提取组件的 CSS

当使用单文件组件时，组件内的 CSS 会以 <style> 标签的方式通过 JavaScript 动态注入。这有一些小小的运行时开销，如果你使用服务端渲染，这会导致一段“无样式的内容瞬间 (fouc)”。横跨所有组件提取 CSS 到同一个文件回避这件事情，这也会更好的压缩和缓存 CSS。

可查阅相关构建工具文档：

* [**Webpack + vue-loader**](https://vue-loader.vuejs.org/zh-cn/configurations/extract-css.html) (the vue-cli webpack template has this pre-configured
* [**Browserify + vueify**](https://github.com/vuejs/vueify#css-extraction)
* [**Rollup + rollup-plugin-vue**](https://vuejs.github.io/rollup-plugin-vue/#/en/2.3/?id=custom-handler)

## 跟踪运行时错误

如果在组件渲染时出现运行错误，错误将会被传递至全局 Vue.config.errorHandler 配置函数 (如果已设置)。利用这个钩子函数和错误跟踪服务 (如 [**Sentry**](https://sentry.io/)，它为 Vue 提供[**官方集成**](https://sentry.io/for/vue/))，可能是个不错的主意。

# 单文件组件

## 介绍

在很多 Vue 项目中，我们使用 Vue.component 来定义全局组件，紧接着用 new Vue({ el: '#container '}) 在每个页面内指定一个容器元素。

这种方式在很多中小规模的项目中运作的很好，在这些项目里 JavaScript 只被用来加强特定的视图。但当在更复杂的项目中，或者你的前端完全由 JavaScript 驱动的时候，下面这些缺点将变得非常明显：

* **全局定义 (Global definitions)** 强制要求每个 component 中的命名不得重复
* **字符串模板 (String templates)** 缺乏语法高亮，在 HTML 有多行的时候，需要用到丑陋的 \
* **不支持 CSS (No CSS support)** 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏
* **没有构建步骤 (No build step)** 限制只能使用 HTML 和 ES5 JavaScript, 而不能使用预处理器，如 Pug (formerly Jade) 和 Babel

文件扩展名为 .vue 的 **single-file components(单文件组件)** 为以上所有问题提供了解决方法，并且还可以使用 Webpack 或 Browserify 等构建工具。

这是一个文件名为 Hello.vue 的简单实例：

现在我们获得：

* [**完整语法高亮**](https://github.com/vuejs/awesome-vue#source-code-editing)
* [**CommonJS 模块**](https://webpack.js.org/concepts/modules/#what-is-a-webpack-module)
* [**组件作用域的 CSS**](https://vue-loader.vuejs.org/zh-cn/features/scoped-css.html)

正如我们说过的，我们可以使用预处理器来构建简洁和功能更丰富的组件，比如 Pug，Babel (with ES2015 modules)，和 Stylus。

这些特定的语言只是例子，你可以只是简单地使用 Babel，TypeScript，SCSS，PostCSS - 或者其他任何能够帮助你提高生产力的预处理器。如果搭配 vue-loader 使用 Webpack，它也是把 CSS Modules 当作第一公民来对待的。

### 怎么看待关注点分离？

一个重要的事情值得注意，**关注点分离不等于文件类型分离。**在现代 UI 开发中，我们已经发现相比于把代码库分离成三个大的层次并将其相互交织起来，把它们划分为松散耦合的组件再将其组合起来更合理一些。在一个组件里，其模板、逻辑和样式是内部耦合的，并且把他们搭配在一起实际上使得组件更加内聚且更可维护。

即便你不喜欢单文件组件，你仍然可以把 JavaScript、CSS 分离成独立的文件然后做到热重载和预编译。

|  |
| --- |
| <!-- my-component.vue -->  <template>  <div>This will be pre-compiled</div>  </template>  <script src="./my-component.js"></script>  <style src="./my-component.css"></style> |

## 起步

### 针对刚接触 JavaScript 模块开发系统的用户

有了 .vue 组件，我们就进入了高级 JavaScript 应用领域。如果你没有准备好的话，意味着还需要学会使用一些附加的工具：

* **Node Package Manager (NPM)**：阅读 [**Getting Started guide**](https://docs.npmjs.com/getting-started/what-is-npm) 直到 10: Uninstalling global packages章节。
* **Modern JavaScript with ES2015/16**：阅读 Babel 的 [**Learn ES2015 guide**](https://babeljs.io/docs/learn-es2015/)。你不需要立刻记住每一个方法，但是你可以保留这个页面以便后期参考。

在你花一些时日了解这些资源之后，我们建议你参考 [**webpack-simple**](https://github.com/vuejs-templates/webpack-simple)。只要遵循指示，你就能很快地运行一个用到 .vue 组件，ES2015 和 热重载 (hot-reloading) 的 Vue 项目!

这个模板使用 [**Webpack**](https://webpack.github.io/)，一个能将多个模块打包成最终应用的模块打包工具。想学习更多 Webpack 的知识，请移步[**它们的官方文档**](https://webpack.js.org/configuration/)以及 [**Webpack Academy**](https://webpack.academy/p/the-core-concepts)。

在 Webpack 中，每个模块被打包到 bundle 之前都由一个相应的 “loader” 来转换，Vue 也提供 [**vue-loader**](https://github.com/vuejs/vue-loader) 插件来执行 .vue 单文件组件 的转换。这个 [**webpack-simple**](https://github.com/vuejs-templates/webpack-simple) 模板已经为你准备好了所有的东西，但是如果你想了解更多关于 .vue 组件和 Webpack 如何一起运转的信息，你可以阅读 [**vue-loader 的文档**](https://vue-loader.vuejs.org/)。

### 针对高级用户

无论你更钟情 Webpack 或是 Browserify，我们为简单的和更复杂的项目都提供了一些文档模板。我们建议浏览 [**github.com/vuejs-templates**](https://github.com/vuejs-templates)，找到你需要的部分，然后参考 README 中的说明，使用 [**vue-cli**](https://github.com/vuejs/vue-cli) 工具生成新的项目。

模板中使用 [**Webpack**](https://webpack.js.org/)，一个模块加载器加载多个模块然后构建成最终应用。为了进一步了解 Webpack，可以看 [**官方介绍视频**](https://www.youtube.com/watch?v=WQue1AN93YU)。如果你有基础，可以看 [**在 Egghead.io 上的 Webpack 进阶教程**](https://egghead.io/courses/using-webpack-for-production-javascript-applications)。

# 单元测试

## 配置和工具

任何兼容基于模块的构建系统都可以正常使用，但如果你需要一个具体的建议，可以使用 [**Karma**](https://karma-runner.github.io/) 进行自动化测试。它有很多社区版的插件，包括对 [**Webpack**](https://github.com/webpack/karma-webpack) 和 [**Browserify**](https://github.com/Nikku/karma-browserify) 的支持。更多详细的安装步骤，请参考各项目的安装文档，通过这些 Karma 配置的例子可以快速帮助你上手 ([**Webpack**](https://github.com/vuejs-templates/webpack/blob/master/template/test/unit/karma.conf.js) 配置，[**Browserify**](https://github.com/vuejs-templates/browserify/blob/master/template/karma.conf.js) 配置)。

## 简单的断言

你不必为了可测性在组件中做任何特殊的操作，导出原始设置就可以了：

|  |
| --- |
| <template>  <span>{{ message }}</span>  </template>  <script>  export default {  data () {  return {  message: 'hello!'  }  },  created () {  this.message = 'bye!'  }  }  </script> |

然后随着 Vue 导入组件的选项，你可以使用许多常见的断言：

|  |
| --- |
| // 导入 Vue.js 和组件，进行测试  import Vue from 'vue'  import MyComponent from 'path/to/MyComponent.vue'  // 这里是一些 Jasmine 2.0 的测试，你也可以使用你喜欢的任何断言库或测试工具。  describe('MyComponent', () => {  // 检查原始组件选项  it('has a created hook', () => {  expect(typeof MyComponent.created).toBe('function')  })  // 评估原始组件选项中的函数的结果  it('sets the correct default data', () => {  expect(typeof MyComponent.data).toBe('function')  const defaultData = MyComponent.data()  expect(defaultData.message).toBe('hello!')  })  // 检查 mount 中的组件实例  it('correctly sets the message when created', () => {  const vm = new Vue(MyComponent).$mount()  expect(vm.message).toBe('bye!')  })  // 创建一个实例并检查渲染输出  it('renders the correct message', () => {  const Ctor = Vue.extend(MyComponent)  const vm = new Ctor().$mount()  expect(vm.$el.textContent).toBe('bye!')  })  }) |

## 编写可被测试的组件

很多组件的渲染输出由它的 props 决定。事实上，如果一个组件的渲染输出完全取决于它的 props，那么它会让测试变得简单，就好像断言不同参数的纯函数的返回值。看下面这个例子：

|  |
| --- |
| <template>  <p>{{ msg }}</p>  </template>  <script>  export default {  props: ['msg']  }  </script> |

你可以在不同的 props 中，通过 propsData 选项断言它的渲染输出：

|  |
| --- |
| import Vue from 'vue'  import MyComponent from './MyComponent.vue'  // 挂载元素并返回已渲染的文本的工具函数  function getRenderedText (Component, propsData) {  const Ctor = Vue.extend(Component)  const vm = new Ctor({ propsData: propsData }).$mount()  return vm.$el.textContent  }  describe('MyComponent', () => {  it('renders correctly with different props', () => {  expect(getRenderedText(MyComponent, {  msg: 'Hello'  })).toBe('Hello')  expect(getRenderedText(MyComponent, {  msg: 'Bye'  })).toBe('Bye')  })  }) |

## 断言异步更新

由于 Vue 进行 [**异步更新 DOM**](https://cn.vuejs.org/v2/guide/reactivity.html#异步更新队列) 的情况，一些依赖 DOM 更新结果的断言必须在 Vue.nextTick 回调中进行：

|  |
| --- |
| // 在状态更新后检查生成的 HTML  it('updates the rendered message when vm.message updates', done => {  const vm = new Vue(MyComponent).$mount()  vm.message = 'foo'  // 在状态改变后和断言 DOM 更新前等待一刻  Vue.nextTick(() => {  expect(vm.$el.textContent).toBe('foo')  done()  })  }) |

我们计划做一个通用的测试工具集，让不同策略的渲染输出 (例如忽略子组件的基本渲染) 和断言变得更简单。

# TypeScript 支持

## 从 2.2.0 起针对 TS + Webpack 2 用户的重要改动

在 Vue 2.2 里，我们引入了新机制，把 dist 文件都作为 ES 模块发布。这在 webpack 2 中属于默认行为。遗憾的是，这个改动会引入一个会破坏兼容性的意外改动。在 TypeScript + webpack 2 里，import Vue = require('vue') 现在会返回一个综合的 ES 模块对象，而不是 Vue 对象本身了。

我们计划在未来把所有的官方类型声明都改成 ES-风格的导出方式 (译注：export)。请参阅下面的[**推荐配置**](https://cn.vuejs.org/v2/guide/typescript.html#推荐配置)板块，配置一个不易过时的编码方案。

## 发布为 NPM 包的官方声明文件

静态类型系统能帮助你有效防止许多潜在的运行时错误，而且随着你的应用日渐丰满会更加显著。这就是为什么 Vue 不仅仅为 Vue core 提供了针对 [**TypeScript**](https://www.typescriptlang.org/) 的[**官方类型声明**](https://github.com/vuejs/vue/tree/dev/types)，还为 [**Vue Router**](https://github.com/vuejs/vue-router/tree/dev/types) 和 [**Vuex**](https://github.com/vuejs/vuex/tree/dev/types) 也提供了相应的声明文件。

而且，我们已经把它们[**发布到了 NPM**](https://cdn.jsdelivr.net/npm/vue/types/)，最新版本的 TypeScript 也知道该如何自己从 NPM 包里解析类型声明。这意味着只要你成功地通过 NPM 安装了，就不再需要任何额外的工具辅助，即可在 Vue 中使用 TypeScript 了。

## 推荐配置

|  |
| --- |
| // tsconfig.json  {  "compilerOptions": {  // ... 已省略其它选项  "allowSyntheticDefaultImports": true,  "lib": [  "dom",  "es5",  "es2015.promise"  ]  }  } |

请注意 allowSyntheticDefaultImports 选项允许你使用下列语法：

|  |
| --- |
| import Vue from 'vue' |

而不是这种：

|  |
| --- |
| import Vue = require('vue') // 注：老语法 |

我们更为推荐前者 (ES 模块语法)，因为他跟原生的 ES 用法更为一致，而且在未来，我们计划把官方声明全部搬迁到 ES 风格的导出方式。

另外呢，如果你是搭配 webpack 2 使用 TypeScript，那么以下配置也很推荐：

|  |
| --- |
| {  "compilerOptions": {  // ... 已省略其他配置  "module": "es2015",  "moduleResolution": "node"  }  } |

这句选项告诉 TypeScript 不要处理 ES 模块引入语句 (译注：import .. from ..)。这样 webpack 2 就可以充分利用其基于 ES 模块的 tree-shaking (译注一种在抽象语法树中减除未被使用的死代码的优化技术，简称摇树优化)。

参阅 [**TypeScript 编译器选项文档 (英)**](https://www.typescriptlang.org/docs/handbook/compiler-options.html)了解更多。

## 使用 Vue 的类型声明

Vue 的类型声明导出了很多有效的[**类型声明**](https://github.com/vuejs/vue/blob/dev/types/index.d.ts)。比如，标记一个导出的组件选项对象 (e.g. 在 .vue 文件中)：

|  |
| --- |
| import Vue, { ComponentOptions } from 'vue'  export default {  props: ['message'],  template: '<span>{{ message }}</span>'  } as ComponentOptions<Vue> |

## Class 风格的 Vue 组件

我们可以很容易地标记上 Vue 组件选项的类型：

|  |
| --- |
| import Vue, { ComponentOptions } from 'vue'  // 声明该组件的类型  interface MyComponent extends Vue {  message: string  onClick (): void  }  export default {  template: '<button @click="onClick">Click!</button>',  data: function () {  return {  message: 'Hello!'  }  },  methods: {  onClick: function () {  // TypeScript 知道 `this` 是类型为 MyComponent 的对象  // 因此 `this.message` 会是一个 string  window.alert(this.message)  }  }  // 我们需要显式地标注导出选项对象为 MyComponent 类型  } as ComponentOptions<MyComponent> |

不幸的是，这里也有一些局限性：

* **TypeScript 不能推断出 Vue API 里的所有类型**。比如，他们不知道我们 data 函数中返回的 message 属性会被添加到 MyComponent 实例中。这意味着如果我们给 message赋值一个数字或者布尔值，linter 和 编译器并不能抛出一个“该值应该是字符串”的错误。
* 因为第一条的局限, **如上的类型注释可能会很罗嗦**。TypeScript 不能正确推导 message的类型，是唯一迫使我们手动声明它是 string 的原因。

好消息是，[**vue-class-component**](https://github.com/vuejs/vue-class-component) 能解决以上的两个问题。这是一个官方的姐妹库，它能允许你把组件声明为一个原生的 JavaScript 类，外加一个 @Component 的修饰符。为了举例说明，我们把上面的例子重写一下吧：

|  |
| --- |
| import Vue from 'vue'  import Component from 'vue-class-component'  // @Component 修饰符注明了此类为一个 Vue 组件  @Component({  // 所有的组件选项都可以放在这里  template: '<button @click="onClick">Click!</button>'  })  export default class MyComponent extends Vue {  // 初始数据可以直接声明为实例的属性  message: string = 'Hello!'  // 组件方法也可以直接声明为实例的方法  onClick (): void {  window.alert(this.message)  }  } |

有了这种备选语法，我们的组件定义不仅仅更加短小了，而且 TypeScript 也能在无需显式的接口声明的情况下，正确推断 message 和 onClick 的类型了呢。这个策略甚至能让你处理计算属性 (computed)，生命周期钩子以及 render 函数的类型。你可以参阅 [**vue-class-component 文档**](https://github.com/vuejs/vue-class-component#vue-class-component)，来了解完整的细节。

## 声明 Vue 插件补充的类型

插件可以增加 Vue 的全局/实例属性和组件选项。在这些情况下，在 TypeScript 中制作插件需要类型声明。庆幸的是，TypeScript 有一个特性来补充现有的类型，叫做[**模块补充 (module augmentation)**](https://www.typescriptlang.org/docs/handbook/declaration-merging.html#module-augmentation)。

例如，声明一个 string 类型的实例属性 $myProperty：

|  |
| --- |
| // 1. 确保在声明补充的类型之前导入 'vue'  import Vue from 'vue'  // 2. 定制一个文件，设置你想要补充的类型  // 在 types/vue.d.ts 里 Vue 有构造函数类型  declare module 'vue/types/vue' {  // 3. 声明为 Vue 补充的东西  interface Vue {  $myProperty: string  }  } |

在你的项目中包含了上述作为声明文件的代码之后 (像 my-property.d.ts)，你就可以在 Vue 实例上使用 $myProperty 了。

|  |
| --- |
| var vm = new Vue()  console.log(vm.$myProperty) // 将会顺利编译通过 |

你也可以声明额外的属性和组件选项：

|  |
| --- |
| import Vue from 'vue'  declare module 'vue/types/vue' {  // 可以使用 `namespace` 替代 `interface`  // 来声明全局属性  namespace Vue {  const $myGlobal: string  }  }  // ComponentOptions 声明于 types/options.d.ts 之中  declare module 'vue/types/options' {  interface ComponentOptions<V extends Vue> {  myOption?: string  }  } |

上述的声明允许下面的代码顺利编译通过：

|  |
| --- |
| // 全局属性  console.log(Vue.$myGlobal)  // 额外的组件选项  var vm = new Vue({  myOption: 'Hello'  }) |

### 规模化

# 路由

## 官方路由

对于大多数单页面应用，都推荐使用官方支持的 [**vue-router 库**](https://github.com/vuejs/vue-router)。更多细节可以看 [**vue-router 文档**](https://router.vuejs.org/)。

## 从零开始简单的路由

如果只需要非常简单的路由而不需要引入整个路由库，可以动态渲染一个页面级的组件像这样：

|  |
| --- |
| const NotFound = { template: '<p>Page not found</p>' }  const Home = { template: '<p>home page</p>' }  const About = { template: '<p>about page</p>' }  const routes = {  '/': Home,  '/about': About  }  new Vue({  el: '#app',  data: {  currentRoute: window.location.pathname  },  computed: {  ViewComponent () {  return routes[this.currentRoute] || NotFound  }  },  render (h) { return h(this.ViewComponent) }  }) |

结合 HTML5 History API，你可以建立一个非常基本但功能齐全的客户端路由器。可以直接看[**实例应用**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example)

## 整合第三方路由

如果有非常喜欢的第三方路由，如 [**Page.js**](https://github.com/visionmedia/page.js) 或者 [**Director**](https://github.com/flatiron/director)，整合[**很简单**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example/compare/master...pagejs)。这有个用了 Page.js 的[**复杂示例**](https://github.com/chrisvfritz/vue-2.0-simple-routing-example/tree/pagejs)。

# 状态管理

## 类 Flux 状态管理的官方实现

由于多个状态分散的跨越在许多组件和交互间各个角落，大型应用复杂度也经常逐渐增长。为了解决这个问题，Vue 提供 [**vuex**](https://github.com/vuejs/vuex)：我们有受到 Elm 启发的状态管理库。vuex 甚至集成到 [**vue-devtools**](https://github.com/vuejs/vue-devtools)，无需配置即可访问时光旅行。

### React 的开发者请参考以下信息

如果你是来自 React 的开发者，你可能会对 Vuex 和 [**Redux**](https://github.com/reactjs/redux) 间的差异表示关注，Redux 是 React 生态环境中最流行的 Flux 实现。Redux 事实上无法感知视图层，所以它能够轻松的通过一些[**简单绑定**](https://github.com/egoist/revue)和 Vue 一起使用。Vuex 区别在于它是一个专门为 Vue 应用所设计。这使得它能够更好地和 Vue 进行整合，同时提供简洁的 API 和改善过的开发体验。

## 简单状态管理起步使用

经常被忽略的是，Vue 应用中原始数据对象的实际来源 - 当访问数据对象时，一个 Vue 实例只是简单的代理访问。所以，如果你有一处需要被多个实例间共享的状态，可以简单地通过维护一份数据来实现共享：

|  |
| --- |
| const sourceOfTruth = {}  const vmA = new Vue({  data: sourceOfTruth  })  const vmB = new Vue({  data: sourceOfTruth  }) |

现在当 sourceOfTruth 发生变化，vmA 和 vmB 都将自动的更新引用它们的视图。子组件们的每个实例也会通过 this.$root.$data 去访问。现在我们有了唯一的实际来源，但是，调试将会变为噩梦。任何时间，我们应用中的任何部分，在任何数据改变后，都不会留下变更过的记录。

为了解决这个问题，我们采用一个简单的 **store 模式**：

|  |
| --- |
| var store = {  debug: true,  state: {  message: 'Hello!'  },  setMessageAction (newValue) {  if (this.debug) console.log('setMessageAction triggered with', newValue)  this.state.message = newValue  },  clearMessageAction () {  if (this.debug) console.log('clearMessageAction triggered')  this.state.message = ''  }  } |

需要注意，所有 store 中 state 的改变，都放置在 store 自身的 action 中去管理。这种集中式状态管理能够被更容易地理解哪种类型的 mutation 将会发生，以及它们是如何被触发。当错误出现时，我们现在也会有一个 log 记录 bug 之前发生了什么。

此外，每个实例/组件仍然可以拥有和管理自己的私有状态：

|  |
| --- |
| var vmA = new Vue({  data: {  privateState: {},  sharedState: store.state  }  })  var vmB = new Vue({  data: {  privateState: {},  sharedState: store.state  }  }) |

重要的是，注意你不应该在 action 中 替换原始的状态对象 - 组件和 store 需要引用同一个共享对象，mutation 才能够被观察

接着我们继续延伸约定，组件不允许直接修改属于 store 实例的 state，而应执行 action 来分发 (dispatch) 事件通知 store 去改变，我们最终达成了 [**Flux**](https://facebook.github.io/flux/) 架构。这样约定的好处是，我们能够记录所有 store 中发生的 state 改变，同时实现能做到记录变更 (mutation)、保存状态快照、历史回滚/时光旅行的先进的调试工具。

说了一圈其实又回到了[**vuex**](https://github.com/vuejs/vuex)，如果你已经读到这儿，或许可以去尝试一下！

# 服务端渲染

## SSR 完全指南

在 2.3 发布后我们发布了一份完整的构建 Vue 服务端渲染应用的指南。这份指南非常深入，适合已经熟悉 Vue, webpack 和 Node.js 开发的开发者阅读。请移步 [**ssr.vuejs.org**](https://ssr.vuejs.org/zh/)。

## Nuxt.js

从头搭建一个服务端渲染的应用是相当复杂的。幸运的是，我们有一个优秀的社区项目 [**Nuxt.js**](https://nuxtjs.org/) 让这一切变得非常简单。Nuxt 是一个基于 Vue 生态的更高层的框架，为开发服务端渲染的 Vue 应用提供了极其便利的开发体验。更酷的是，你甚至可以用它来做为静态站生成器。推荐尝试。

### 内在

# 深入响应式原理

现在是时候深入一下了！Vue 的一个最明显的特性就是其不太引人注意的响应式系统。数据模型仅仅是普通的 JavaScript 对象。而当你修改它们时，视图会进行更新。这使得状态管理非常简单直接，不过理解其工作原理同样重要，这样你可以回避一些常见的问题。在这个章节，我们将进入一些 Vue 响应式系统的底层的细节。

## 如何追踪变化

当你把一个普通的 JavaScript 对象传给 Vue 实例的 data 选项，Vue 将遍历此对象所有的属性，并使用 [**Object.defineProperty**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) 把这些属性全部转为 getter/setter。Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是为什么 Vue 不支持 IE8 以及更低版本浏览器的原因。

用户看不到 getter/setter，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。这里需要注意的问题是浏览器控制台在打印数据对象时 getter/setter 的格式化并不同，所以你可能需要安装 [**vue-devtools**](https://github.com/vuejs/vue-devtools) 来获取更加友好的检查接口。

每个组件实例都有相应的 **watcher** 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。

## 检测变化的注意事项

受现代 JavaScript 的限制 (以及废弃 Object.observe)，Vue **不能检测到对象属性的添加或删除**。由于 Vue 会在初始化实例时对属性执行 getter/setter 转化过程，所以属性必须在 data 对象上存在才能让 Vue 转换它，这样才能让它是响应的。例如：

|  |
| --- |
| var vm = new Vue({  data:{  a:1  }  })  // `vm.a` 是响应的  vm.b = 2  // `vm.b` 是非响应的 |

Vue 不允许在已经创建的实例上动态添加新的根级响应式属性 (root-level reactive property)。然而它可以使用 Vue.set(object, key, value) 方法将响应属性添加到嵌套的对象上：

|  |
| --- |
| Vue.set(vm.someObject, 'b', 2) |

您还可以使用 vm.$set 实例方法，这也是全局 Vue.set 方法的别名：

|  |
| --- |
| this.$set(this.someObject,'b',2) |

有时你想向已有对象上添加一些属性，例如使用 Object.assign() 或 \_.extend() 方法来添加属性。但是，添加到对象上的新属性不会触发更新。在这种情况下可以创建一个新的对象，让它包含原对象的属性和新的属性：

|  |
| --- |
| // 代替 `Object.assign(this.someObject, { a: 1, b: 2 })`  this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 }) |

也有一些数组相关的问题，之前已经在[**列表渲染**](https://cn.vuejs.org/v2/guide/list.html#注意事项)中讲过。

## 声明响应式属性

由于 Vue 不允许动态添加根级响应式属性，所以你必须在初始化实例前声明根级响应式属性，哪怕只是一个空值：

|  |
| --- |
| var vm = new Vue({  data: {  // 声明 message 为一个空值字符串  message: ''  },  template: '<div>{{ message }}</div>'  })  // 之后设置 `message`  vm.message = 'Hello!' |

如果你在 data 选项中未声明 message，Vue 将警告你渲染函数在试图访问的属性不存在。

这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue 实例在类型检查系统的帮助下运行的更高效。而且在代码可维护性方面也有一点重要的考虑：data 对象就像组件状态的概要，提前声明所有的响应式属性，可以让组件代码在以后重新阅读或其他开发人员阅读时更易于被理解。

## 异步更新队列

可能你还没有注意到，Vue **异步**执行 DOM 更新。只要观察到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据改变。如果同一个 watcher 被多次触发，只会一次推入到队列中。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作上非常重要。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部尝试对异步队列使用原生的 Promise.then 和 MutationObserver，如果执行环境不支持，会采用 setTimeout(fn, 0) 代替。

例如，当你设置 vm.someData = 'new value' ，该组件不会立即重新渲染。当刷新队列时，组件会在事件循环队列清空时的下一个“tick”更新。多数情况我们不需要关心这个过程，但是如果你想在 DOM 状态更新后做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员沿着“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们确实要这么做。为了在数据变化之后等待 Vue 完成更新 DOM ，可以在数据变化之后立即使用 Vue.nextTick(callback) 。这样回调函数在 DOM 更新完成后就会调用。例如：

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

|  |
| --- |
| var vm = new Vue({  el: '#example',  data: {  message: '123'  }  })  vm.message = 'new message' // 更改数据  vm.$el.textContent === 'new message' // false  Vue.nextTick(function () {  vm.$el.textContent === 'new message' // true  }) |

在组件内使用 vm.$nextTick() 实例方法特别方便，因为它不需要全局 Vue ，并且回调函数中的 this 将自动绑定到当前的 Vue 实例上：

|  |
| --- |
| Vue.component('example', {  template: '<span>{{ message }}</span>',  data: function () {  return {  message: '没有更新'  }  },  methods: {  updateMessage: function () {  this.message = '更新完成'  console.log(this.$el.textContent) // => '没有更新'  this.$nextTick(function () {  console.log(this.$el.textContent) // => '更新完成'  })  }  }  }) |