# 第1章 安装

## 1.1 兼容性

Vue 不支持 IE8 及以下版本，因为 Vue 使用了 IE8 无法模拟的 ECMAScript 5 特性。但它支持所有兼容 ECMAScript 5 的浏览器。

## 1.2 语义化版本控制

Vue 在其所有项目中公布的功能和行为都遵循语义化版本控制。对于未公布的或内部暴露的行为，其变更会描述在发布说明中。

## 1.3 更新日志

最新稳定版本：2.7.8

每个版本的更新日志见 GitHub。

## 1.4 VueDevtools

在使用 Vue 时，我们推荐在你的浏览器上安装 Vue Devtools。它允许你在一个更友好的界面中审查和调试 Vue 应用。

## 1.5 直接用 <script> 引入

直接下载并用 <script> 标签引入，Vue 会被注册为一个全局变量。

在开发环境下不要使用压缩版本，不然你就失去了所有常见错误相关的警告!

### 1.5.1 开发版本

https://cn.vuejs.org/js/vue.js

### 1.5.2 生产版本

https://cn.vuejs.org/js/vue.min.js

### 1.5.3 CDN

对于制作原型或学习，你可以这样使用最新版本：

<script src="https://cdn.jsdelivr.net/npm/vue@2.7.8/dist/vue.js"></script>

对于生产环境，我们推荐链接到一个明确的版本号和构建文件，以避免新版本造成的不可预期的破坏：

<script src="https://cdn.jsdelivr.net/npm/vue@2.7.8"></script>

如果你使用原生 ES Modules，这里也有一个兼容 ES Module 的构建文件：

<script type="module">

import Vue from 'https://cdn.jsdelivr.net/npm/vue@2.7.8/dist/vue.esm.browser.js'

</script>

你可以在 cdn.jsdelivr.net/npm/vue 浏览 NPM 包的源代码。

Vue 也可以在 unpkg 和 cdnjs 上获取 (cdnjs 的版本更新可能略滞后)。

请确认了解不同构建版本并在你发布的站点中使用生产环境版本，把 vue.js 换成 vue.min.js。这是一个更小的构建，可以带来比开发环境下更快的速度体验。

## 1.6 NPM

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

# 最新稳定版

$ npm install vue

## 1.7 命令行工具 (CLI)

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

CLI 工具假定用户对 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，在熟悉 Vue 本身之后再使用 CLI。

## 1.8 对不同构建版本的解释

在 NPM 包的 dist/ 目录你将会找到很多不同的 Vue.js 构建版本。这里列出了它们之间的差别：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | UMD | CommonJS | ES Module (基于构建工具使用) | ES Module (直接用于浏览器) |
| 完整版 | vue.js | vue.common.js | vue.esm.js | vue.esm.browser.js |
| 只包含运行时版 | vue.runtime.js | vue.runtime.common.js | vue.runtime.esm.js |  |
| 完整版 (生产环境) | vue.min.js |  |  | vue.esm.browser.min.js |
| 只包含运行时版 (生产环境) | vue.runtime.min.js |  |  |  |

### 1.8.1 术语

完整版：同时包含编译器和运行时的版本。

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、渲染并处理虚拟 DOM 等的代码。基本上就是除去编译器的其它一切。

UMD：UMD 版本可以通过 <script> 标签直接用在浏览器中。jsDelivr CDN 的 https://cdn.jsdelivr.net/npm/vue@2.7.8 默认文件就是运行时 + 编译器的 UMD 版本 (vue.js)。

CommonJS：CommonJS 版本用来配合老的打包工具比如 Browserify 或 webpack 1。这些打包工具的默认文件 (pkg.main) 是只包含运行时的 CommonJS 版本 (vue.runtime.common.js)。

ES Module：从 2.6 开始 Vue 会提供两个 ES Modules (ESM) 构建文件：

为打包工具提供的 ESM：为诸如 webpack 2 或 Rollup 提供的现代打包工具。ESM 格式被设计为可以被静态分析，所以打包工具可以利用这一点来进行“tree-shaking”并将用不到的代码排除出最终的包。为这些打包工具提供的默认文件 (pkg.module) 是只有运行时的 ES Module 构建 (vue.runtime.esm.js)。

为浏览器提供的 ESM (2.6+)：用于在现代浏览器中通过 <script type="module"> 直接导入。

### 1.8.2 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 DOM 内部的 HTML 作为模板)，就将需要加上编译器，即完整版：

// 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，所以只用运行时版本即可。

因为运行时版本相比完整版体积要小大约 30%，所以应该尽可能使用这个版本。如果你仍然希望使用完整版，则需要在打包工具里配置一个别名：

**webpack**

module.exports = {

// ...

resolve: {

alias: {

'vue$': 'vue/dist/vue.esm.js' // 用 webpack 1 时需用 'vue/dist/vue.common.js'

}

}

}

**Rollup**

const alias = require('rollup-plugin-alias')

rollup({

// ...

plugins: [

alias({

'vue': require.resolve('vue/dist/vue.esm.js')

})

]

})

**Browserify**

添加到你项目的 package.json：

{

// ...

"browser": {

"vue": "vue/dist/vue.common.js"

}

}

**Parcel**

在你项目的 package.json 中添加：

{

// ...

"alias": {

"vue" : "./node\_modules/vue/dist/vue.common.js"

}

}

### 1.8.3 开发环境 vs. 生产环境模式

对于 UMD 版本来说，开发环境/生产环境模式是硬编码好的：开发环境下用未压缩的代码，生产环境下使用压缩后的代码。

CommonJS 和 ES Module 版本是用于打包工具的，因此我们不提供压缩后的版本。你需要自行将最终的包进行压缩。

CommonJS 和 ES Module 版本同时保留原始的 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这些环境变量以便控制 Vue 所运行的模式。把 process.env.NODE\_ENV 替换为字符串字面量同时可以让 UglifyJS 之类的压缩工具完全丢掉仅供开发环境的代码块，以减少最终的文件尺寸。

**webpack**

在 webpack 4+ 中，你可以使用 mode 选项：

module.exports = {

mode: 'production'

}

但是在 webpack 3 及其更低版本中，你需要使用 DefinePlugin：

var webpack = require('webpack')

module.exports = {

// ...

plugins: [

// ...

new webpack.DefinePlugin({

'process.env': {

NODE\_ENV: JSON.stringify('production')

}

})

]

}

**Rollup**

使用 rollup-plugin-replace：

const replace = require('rollup-plugin-replace')

rollup({

// ...

plugins: [

replace({

'process.env.NODE\_ENV': JSON.stringify('production')

})

]

}).then(...)

**Browserify**

为你的包应用一次全局的 envify 转换。

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

也可以移步生产环境部署。

### 1.8.4 CSP 环境

有些环境，如 Google Chrome Apps，会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。完整版本依赖于该功能来编译模板，所以无法在这些环境下使用。

另一方面，运行时版本则是完全兼容 CSP 的。当通过 webpack + vue-loader 或者 Browserify + vueify 构建时，模板将被预编译为 render 函数，可以在 CSP 环境中完美运行。

## 1.9 开发版本

重要：GitHub 仓库的 /dist 文件夹只有在新版本发布时才会提交。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

## 1.10 Bower

Bower 只提供 UMD 版本。

# 最新稳定版本

$ bower install vue

## 1.11 AMD 模块加载器

所有 UMD 版本都可以直接用作 AMD 模块。

# 第2章 介绍

## 2.1 Vue.js 是什么

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

如果你想在深入学习 Vue 之前对它有更多了解，我们制作了一个视频，带您了解其核心概念和一个示例工程。

如果你已经是有经验的前端开发者，想知道 Vue 与其它库/框架有哪些区别，请查看对比其它框架。

## 2.2 起步

官方指南假设你已了解关于 HTML、CSS 和 JavaScript 的中级知识。如果你刚开始学习前端开发，将框架作为你的第一步可能不是最好的主意——掌握好基础知识再来吧！之前有其它框架的使用经验会有帮助，但这不是必需的。

尝试 Vue.js 最简单的方法是使用 Hello World 例子。你可以在浏览器新标签页中打开它，跟着例子学习一些基础用法。或者你也可以创建一个 .html 文件，然后通过如下方式引入 Vue：

<!-- 开发环境版本，包含了有帮助的命令行警告 -->

<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

或者：

<!-- 生产环境版本，优化了尺寸和速度 -->

<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>

安装教程给出了更多安装 Vue 的方式。请注意我们不推荐新手直接使用 vue-cli，尤其是在你还不熟悉基于 Node.js 的构建工具时。

如果你喜欢交互式的东西，你也可以查阅这个 Scrimba 上的系列教程，它揉合了录屏和代码试验田，并允许你随时暂停和播放。

## 2.3 声明式渲染

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

<div id="app">

{{ message }}

</div>

var app = new Vue({

el: '#app',

data: {

message: 'Hello Vue!'

}

})

我们已经成功创建了第一个 Vue 应用！看起来这跟渲染一个字符串模板非常类似，但是 Vue 在背后做了大量工作。现在数据和 DOM 已经被建立了关联，所有东西都是响应式的。我们要怎么确认呢？打开你的浏览器的 JavaScript 控制台 (就在这个页面打开)，并修改 app.message 的值，你将看到上例相应地更新。

注意我们不再和 HTML 直接交互了。一个 Vue 应用会将其挂载到一个 DOM 元素上 (对于这个例子是 #app) 然后对其进行完全控制。那个 HTML 是我们的入口，但其余都会发生在新创建的 Vue 实例内部。

除了文本插值，我们还可以像这样来绑定元素 attribute：

<div id="app-2">

<span v-bind:title="message">

鼠标悬停几秒钟查看此处动态绑定的提示信息！

</span>

</div>

var app2 = new Vue({

el: '#app-2',

data: {

message: '页面加载于 ' + new Date().toLocaleString()

}

})

这里我们遇到了一点新东西。你看到的 v-bind attribute 被称为指令。指令带有前缀 v-，以表示它们是 Vue 提供的特殊 attribute。可能你已经猜到了，它们会在渲染的 DOM 上应用特殊的响应式行为。在这里，该指令的意思是：“将这个元素节点的 title attribute 和 Vue 实例的 message property 保持一致”。

如果你再次打开浏览器的 JavaScript 控制台，输入 app2.message = '新消息'，就会再一次看到这个绑定了 title attribute 的 HTML 已经进行了更新。

## 2.4 条件与循环

控制切换一个元素是否显示也相当简单：

<div id="app-3">

<p v-if="seen">现在你看到我了</p>

</div>

var app3 = new Vue({

el: '#app-3',

data: {

seen: true

}

})

继续在控制台输入 app3.seen = false，你会发现之前显示的消息消失了。

这个例子演示了我们不仅可以把数据绑定到 DOM 文本或 attribute，还可以绑定到 DOM 结构。此外，Vue 也提供一个强大的过渡效果系统，可以在 Vue 插入/更新/移除元素时自动应用过渡效果。

还有其它很多指令，每个都有特殊的功能。例如，v-for 指令可以绑定数组的数据来渲染一个项目列表：

<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: '学习 JavaScript' },

{ text: '学习 Vue' },

{ text: '整个牛项目' }

]

}

})

在控制台里，输入 app4.todos.push({ text: '新项目' })，你会发现列表最后添加了一个新项目。

## 2.5 处理用户输入

为了让用户和你的应用进行交互，我们可以用 v-on 指令添加一个事件监听器，通过它调用在 Vue 实例中定义的方法：

<div id="app-5">

<p>{{ message }}</p>

<button v-on:click="reverseMessage">反转消息</button>

</div>

var app5 = new Vue({

el: '#app-5',

data: {

message: 'Hello Vue.js!'

},

methods: {

reverseMessage: function () {

this.message = this.message.split('').reverse().join('')

}

}

})

注意在 reverseMessage 方法中，我们更新了应用的状态，但没有触碰 DOM——所有的 DOM 操作都由 Vue 来处理，你编写的代码只需要关注逻辑层面即可。

Vue 还提供了 v-model 指令，它能轻松实现表单输入和应用状态之间的双向绑定。

<div id="app-6">

<p>{{ message }}</p>

<input v-model="message">

</div>

var app6 = new Vue({

el: '#app-6',

data: {

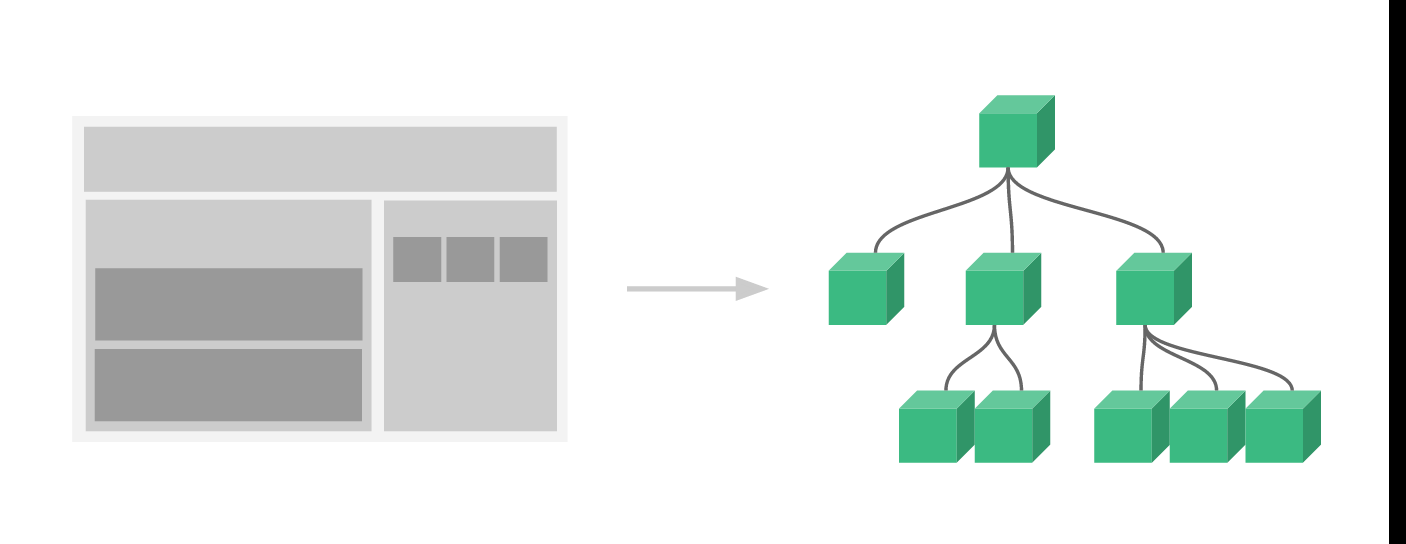
message: 'Hello Vue!'

}

})

## 2.6 组件化应用构建

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。仔细想想，几乎任意类型的应用界面都可以抽象为一个组件树：



在 Vue 里，一个组件本质上是一个拥有预定义选项的一个 Vue 实例。在 Vue 中注册组件很简单：

// 定义名为 todo-item 的新组件

Vue.component('todo-item', {

template: '<li>这是个待办项</li>'

})

var app = new Vue(...)

现在你可以用它构建另一个组件模板：

<ol>

<!-- 创建一个 todo-item 组件的实例 -->

<todo-item></todo-item>

</ol>

但是这样会为每个待办项渲染同样的文本，这看起来并不炫酷。我们应该能从父作用域将数据传到子组件才对。让我们来修改一下组件的定义，使之能够接受一个 prop：

Vue.component('todo-item', {

// todo-item 组件现在接受一个

// "prop"，类似于一个自定义 attribute。

// 这个 prop 名为 todo。

props: ['todo'],

template: '<li>{{ todo.text }}</li>'

})

现在，我们可以使用 v-bind 指令将待办项传到循环输出的每个组件中：

<div id="app-7">

<ol>

<!--

现在我们为每个 todo-item 提供 todo 对象

todo 对象是变量，即其内容可以是动态的。

我们也需要为每个组件提供一个“key”，稍后再

作详细解释。

-->

<todo-item

v-for="item in groceryList"

v-bind:todo="item"

v-bind:key="item.id"

></todo-item>

</ol>

</div>

Vue.component('todo-item', {

props: ['todo'],

template: '<li>{{ todo.text }}</li>'

})

var app7 = new Vue({

el: '#app-7',

data: {

groceryList: [

{ id: 0, text: '蔬菜' },

{ id: 1, text: '奶酪' },

{ id: 2, text: '随便其它什么人吃的东西' }

]

}

})

尽管这只是一个刻意设计的例子，但是我们已经设法将应用分割成了两个更小的单元。子单元通过 prop 接口与父单元进行了良好的解耦。我们现在可以进一步改进 <todo-item> 组件，提供更为复杂的模板和逻辑，而不会影响到父单元。

在一个大型应用中，有必要将整个应用程序划分为组件，以使开发更易管理。在后续教程中我们将详述组件，不过这里有一个 (假想的) 例子，以展示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>

### 2.6.1 与自定义元素的关系

你可能已经注意到 Vue 组件非常类似于自定义元素——它是 Web 组件规范的一部分，这是因为 Vue 的组件语法部分参考了该规范。例如 Vue 组件实现了 Slot API 与 is attribute。但是，还是有几个关键差别：

1. Web Components 规范已经完成并通过，但未被所有浏览器原生实现。目前 Safari 10.1+、Chrome 54+ 和 Firefox 63+ 原生支持 Web Components。相比之下，Vue 组件不需要任何 polyfill，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 组件也可以包装于原生自定义元素之内。
2. Vue 组件提供了纯自定义元素所不具备的一些重要功能，最突出的是跨组件数据流、自定义事件通信以及构建工具集成。

虽然 Vue 内部没有使用自定义元素，不过在应用使用自定义元素、或以自定义元素形式发布时，依然有很好的互操作性。Vue CLI 也支持将 Vue 组件构建成为原生的自定义元素。

# 第3章 Vue 实例

## 3.1 创建一个 Vue 实例

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

var vm = new Vue({

// 选项

})

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

当创建一个 Vue 实例时，你可以传入一个选项对象。这篇教程主要描述的就是如何使用这些选项来创建你想要的行为。作为参考，你也可以在 API 文档中浏览完整的选项列表。

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

根实例

└─ TodoList

├─ TodoItem

│ ├─ TodoButtonDelete

│ └─ TodoButtonEdit

└─ TodoListFooter

├─ TodosButtonClear

└─ TodoListStatistics

我们会在稍后的组件系统章节具体展开。不过现在，你只需要明白所有的 Vue 组件都是 Vue 实例，并且接受相同的选项对象 (一些根实例特有的选项除外)。

## 3.2 数据与方法

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

// 我们的数据对象

var data = { a: 1 }

// 该对象被加入到一个 Vue 实例中

var vm = new Vue({

data: data

})

// 获得这个实例上的 property

// 返回源数据中对应的字段

vm.a == data.a // => true

// 设置 property 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

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

vm.b = 'hi'

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

data: {

newTodoText: '',

visitCount: 0,

hideCompletedTodos: false,

todos: [],

error: null

}

这里唯一的例外是使用 Object.freeze()，这会阻止修改现有的 property，也意味着响应系统无法再追踪变化。

var obj = {

foo: 'bar'

}

Object.freeze(obj)

new Vue({

el: '#app',

data: obj

})

<div id="app">

<p>{{ foo }}</p>

<!-- 这里的 `foo` 不会更新！ -->

<button v-on:click="foo = 'baz'">Change it</button>

</div>

除了数据 property，Vue 实例还暴露了一些有用的实例 property 与方法。它们都有前缀 $，以便与用户定义的 property 区分开来。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 (newValue, oldValue) {

// 这个回调将在 `vm.a` 改变后调用

})

以后你可以在 API 参考中查阅到完整的实例 property 和方法的列表。

## 3.3 实例生命周期钩子

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，这给了用户在不同阶段添加自己的代码的机会。

比如 created 钩子可以用来在一个实例被创建之后执行代码：

new Vue({

data: {

a: 1

},

created: function () {

// `this` 指向 vm 实例

console.log('a is: ' + this.a)

}

})

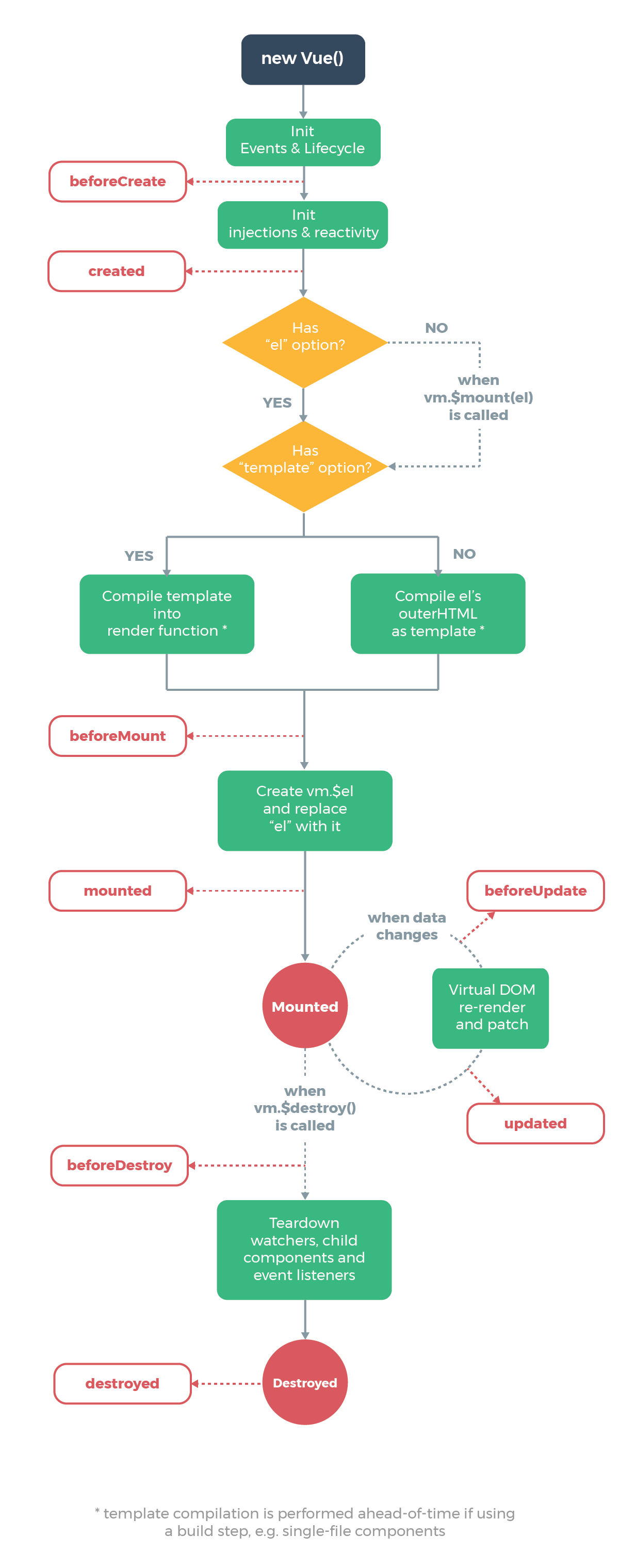
// => "a is: 1"

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 mounted、updated 和 destroyed。生命周期钩子的 this 上下文指向调用它的 Vue 实例。

不要在选项 property 或回调上使用箭头函数，比如 created: () => console.log(this.a) 或 vm.$watch('a', newValue => this.myMethod())。因为箭头函数并没有 this，this 会作为变量一直向上级词法作用域查找，直至找到为止，经常导致 Uncaught TypeError: Cannot read property of undefined 或 Uncaught TypeError: this.myMethod is not a function 之类的错误。

## 3.4 生命周期图示

下图展示了实例的生命周期。你不需要立马弄明白所有的东西，不过随着你的不断学习和使用，它的参考价值会越来越高。



# 第4章 模板语法

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

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

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

## 4.1 插值

### 4.1.1 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 property 的值。无论何时，绑定的数据对象上 msg property 发生了改变，插值处的内容都会更新。

通过使用 v-once 指令，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。但请留心这会影响到该节点上的其它数据绑定：

<span v-once>这个将不会改变: {{ msg }}</span>

### 4.1.2 原始 HTML

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 v-html 指令：

<p>Using mustaches: {{ rawHtml }}</p>

<p>Using v-html directive: <span v-html="rawHtml"></span></p>

这个 span 的内容将会被替换成为 property 值 rawHtml，直接作为 HTML——会忽略解析 property 值中的数据绑定。注意，你不能使用 v-html 来复合局部模板，因为 Vue 不是基于字符串的模板引擎。反之，对于用户界面 (UI)，组件更适合作为可重用和可组合的基本单位。

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。请只对可信内容使用 HTML 插值，绝不要对用户提供的内容使用插值。

### 4.1.3 Attribute

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

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

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

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

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

### 4.1.4 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 property 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。

## 4.2 指令

指令 (Directives) 是带有 v- 前缀的特殊 attribute。指令 attribute 的值预期是单个 JavaScript 表达式 (v-for 是例外情况，稍后我们再讨论)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。回顾我们在介绍中看到的例子：

<p v-if="seen">现在你看到我了</p>

这里，v-if 指令将根据表达式 seen 的值的真假来插入/移除 <p> 元素。

### 4.2.1 参数

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。例如，v-bind 指令可以用于响应式地更新 HTML attribute：

<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 attribute 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">...</a>

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。

### 4.2.2 动态参数

2.6.0 新增

从 2.6.0 开始，可以用方括号括起来的 JavaScript 表达式作为一个指令的参数：

<!--

注意，参数表达式的写法存在一些约束，如之后的“对动态参数表达式的约束”章节所述。

-->

<a v-bind:[attributeName]="url"> ... </a>

这里的 attributeName 会被作为一个 JavaScript 表达式进行动态求值，求得的值将会作为最终的参数来使用。例如，如果你的 Vue 实例有一个 data property attributeName，其值为 "href"，那么这个绑定将等价于 v-bind:href。

同样地，你可以使用动态参数为一个动态的事件名绑定处理函数：

<a v-on:[eventName]="doSomething"> ... </a>

在这个示例中，当 eventName 的值为 "focus" 时，v-on:[eventName] 将等价于 v-on:focus。

**对动态参数的值的约束**

动态参数预期会求出一个字符串，异常情况下值为 null。这个特殊的 null 值可以被显性地用于移除绑定。任何其它非字符串类型的值都将会触发一个警告。

**对动态参数表达式的约束**

动态参数表达式有一些语法约束，因为某些字符，如空格和引号，放在 HTML attribute 名里是无效的。例如：

<!-- 这会触发一个编译警告 -->

<a v-bind:['foo' + bar]="value"> ... </a>

变通的办法是使用没有空格或引号的表达式，或用计算属性替代这种复杂表达式。

在 DOM 中使用模板时 (直接在一个 HTML 文件里撰写模板)，还需要避免使用大写字符来命名键名，因为浏览器会把 attribute 名全部强制转为小写：

<!--

在 DOM 中使用模板时这段代码会被转换为 `v-bind:[someattr]`。

除非在实例中有一个名为“someattr”的 property，否则代码不会工作。

-->

<a v-bind:[someAttr]="value"> ... </a>

### 4.2.3 修饰符

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

<form v-on:submit.prevent="onSubmit">...</form>

在接下来对 v-on 和 v-for 等功能的探索中，你会看到修饰符的其它例子。

## 4.3 缩写

缩写

v- 前缀作为一种视觉提示，用来识别模板中 Vue 特定的 attribute。当你在使用 Vue.js 为现有标签添加动态行为 (dynamic behavior) 时，v- 前缀很有帮助，然而，对于一些频繁用到的指令来说，就会感到使用繁琐。同时，在构建由 Vue 管理所有模板的单页面应用程序 (SPA - single page application) 时，v- 前缀也变得没那么重要了。因此，Vue 为 v-bind 和 v-on 这两个最常用的指令，提供了特定简写：

### 4.3.1 v-bind 缩写

<!-- 完整语法 -->

<a v-bind:href="url">...</a>

<!-- 缩写 -->

<a :href="url">...</a>

<!-- 动态参数的缩写 (2.6.0+) -->

<a :[key]="url"> ... </a>

### 4.3.2 v-on 缩写

<!-- 完整语法 -->

<a v-on:click="doSomething">...</a>

<!-- 缩写 -->

<a @click="doSomething">...</a>

<!-- 动态参数的缩写 (2.6.0+) -->

<a @[event]="doSomething"> ... </a>

它们看起来可能与普通的 HTML 略有不同，但 : 与 @ 对于 attribute 名来说都是合法字符，在所有支持 Vue 的浏览器都能被正确地解析。而且，它们不会出现在最终渲染的标记中。缩写语法是完全可选的，但随着你更深入地了解它们的作用，你会庆幸拥有它们。

# 第5章 计算属性和侦听器

## 5.1 计算属性

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。在模板中放入太多的逻辑会让模板过重且难以维护。例如：

<div id="example">

{{ message.split('').reverse().join('') }}

</div>

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

所以，对于任何复杂逻辑，你都应当使用计算属性。

### 5.1.1 基础例子

<div id="example">

<p>Original message: "{{ message }}"</p>

<p>Computed reversed message: "{{ reversedMessage }}"</p>

</div>

var vm = new Vue({

el: '#example',

data: {

message: 'Hello'

},

computed: {

// 计算属性的 getter

reversedMessage: function () {

// `this` 指向 vm 实例

return this.message.split('').reverse().join('')

}

}

})

这里我们声明了一个计算属性 reversedMessage。我们提供的函数将用作 property vm.reversedMessage 的 getter 函数：

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，自行修改例子中的 vm。vm.reversedMessage 的值始终取决于 vm.message 的值。

你可以像绑定普通 property 一样在模板中绑定计算属性。Vue 知道 vm.reversedMessage 依赖于 vm.message，因此当 vm.message 发生改变时，所有依赖 vm.reversedMessage 的绑定也会更新。而且最妙的是我们已经以声明的方式创建了这种依赖关系：计算属性的 getter 函数是没有副作用 (side effect) 的，这使它更易于测试和理解。

### 5.1.2 计算属性缓存 vs 方法

你可能已经注意到我们可以通过在表达式中调用方法来达到同样的效果：

<p>Reversed message: "{{ reversedMessage() }}"</p>

// 在组件中

methods: {

reversedMessage: function () {

return this.message.split('').reverse().join('')

}

}

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是计算属性是基于它们的响应式依赖进行缓存的。只在相关响应式依赖发生改变时它们才会重新求值。这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。

这也同样意味着下面的计算属性将不再更新，因为 Date.now() 不是响应式依赖：

computed: {

now: function () {

return Date.now()

}

}

相比之下，每当触发重新渲染时，调用方法将总会再次执行函数。

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 A，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 A。如果没有缓存，我们将不可避免的多次执行 A 的 getter！如果你不希望有缓存，请用方法来替代。

### 5.1.3 计算属性 vs 侦听属性

Vue 提供了一种更通用的方式来观察和响应 Vue 实例上的数据变动：侦听属性。当你有一些数据需要随着其它数据变动而变动时，你很容易滥用 watch——特别是如果你之前使用过 AngularJS。然而，通常更好的做法是使用计算属性而不是命令式的 watch 回调。细想一下这个例子：

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

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar',

fullName: 'Foo Bar'

},

watch: {

firstName: function (val) {

this.fullName = val + ' ' + this.lastName

},

lastName: function (val) {

this.fullName = this.firstName + ' ' + val

}

}

})

上面代码是命令式且重复的。将它与计算属性的版本进行比较：

var vm = new Vue({

el: '#demo',

data: {

firstName: 'Foo',

lastName: 'Bar'

},

computed: {

fullName: function () {

return this.firstName + ' ' + this.lastName

}

}

})

好得多了，不是吗？

### 5.1.4 计算属性的 setter

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

// ...

computed: {

fullName: {

// getter

get: function () {

return this.firstName + ' ' + this.lastName

},

// setter

set: function (newValue) {

var names = newValue.split(' ')

this.firstName = names[0]

this.lastName = names[names.length - 1]

}

}

}

// ...

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

## 5.2 侦听器

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的侦听器。这就是为什么 Vue 通过 watch 选项提供了一个更通用的方法，来响应数据的变化。当需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的。

例如：

<div id="watch-example">

<p>

Ask a yes/no question:

<input v-model="question">

</p>

<p>{{ answer }}</p>

</div>

<!-- 因为 AJAX 库和通用工具的生态已经相当丰富，Vue 核心代码没有重复 -->

<!-- 提供这些功能以保持精简。这也可以让你自由选择自己更熟悉的工具。 -->

<script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/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, oldQuestion) {

this.answer = 'Waiting for you to stop typing...'

this.debouncedGetAnswer()

}

},

created: function () {

// `\_.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。想要了解更多关于

// `\_.debounce` 函数 (及其近亲 `\_.throttle`) 的知识，

// 请参考：https://lodash.com/docs#debounce

this.debouncedGetAnswer = \_.debounce(this.getAnswer, 500)

},

methods: {

getAnswer: function () {

if (this.question.indexOf('?') === -1) {

this.answer = 'Questions usually contain a question mark. ;-)'

return

}

this.answer = 'Thinking...'

var vm = this

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

})

}

}

})

</script>

在这个示例中，使用 watch 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

除了 watch 选项之外，您还可以使用命令式的 vm.$watch API。

# 第6章 Class 与 Style 绑定

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是 attribute，所以我们可以用 v-bind 处理它们：只需要通过表达式计算出字符串结果即可。不过，字符串拼接麻烦且易错。因此，在将 v-bind 用于 class 和 style 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，还可以是对象或数组。

## 6.1 绑定 HTML Class

### 6.1.1 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 active 这个 class 存在与否将取决于数据 property isActive 的 truthiness。

你可以在对象中传入更多字段来动态切换多个 class。此外，v-bind:class 指令也可以与普通的 class attribute 共存。当有如下模板：

<div

class="static"

v-bind:class="{ active: isActive, 'text-danger': hasError }"

></div>

和如下 data：

data: {

isActive: true,

hasError: false

}

结果渲染为：

<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"。

绑定的数据对象不必内联定义在模板里：

<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一个返回对象的计算属性。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'

}

}

}

### 6.1.2 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]"></div>

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]"></div>

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

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

<div v-bind:class="[{ active: isActive }, errorClass]"></div>

### 6.1.3 用在组件上

这个章节假设你已经对 Vue 组件有一定的了解。当然你也可以先跳过这里，稍后再回过头来看。

当在一个自定义组件上使用 class property 时，这些 class 将被添加到该组件的根元素上面。这个元素上已经存在的 class 不会被覆盖。

例如，如果你声明了这个组件：

Vue.component('my-component', {

template: '<p class="foo bar">Hi</p>'

})

然后在使用它的时候添加一些 class：

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

HTML 将被渲染为：

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

对于带数据绑定 class 也同样适用：

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

当 isActive 为 truthy[1] 时，HTML 将被渲染成为：

<p class="foo bar active">Hi</p>

## 6.2 绑定内联样式

### 6.2.1 对象语法

v-bind:style 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS property 名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用引号括起来) 来命名：

<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

data: {

activeColor: 'red',

fontSize: 30

}

直接绑定到一个样式对象通常更好，这会让模板更清晰：

<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。

### 6.2.2 数组语法

v-bind:style 的数组语法可以将多个样式对象应用到同一个元素上：

<div v-bind:style="[baseStyles, overridingStyles]"></div>

### 6.2.3 自动添加前缀

当 v-bind:style 使用需要添加浏览器引擎前缀的 CSS property 时，如 transform，Vue.js 会自动侦测并添加相应的前缀。

### 6.2.4 多重值

2.3.0+

从 2.3.0 起你可以为 style 绑定中的 property 提供一个包含多个值的数组，常用于提供多个带前缀的值，例如：

<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>

这样写只会渲染数组中最后一个被浏览器支持的值。在本例中，如果浏览器支持不带浏览器前缀的 flexbox，那么就只会渲染 display: flex。

# 第7章 条件渲染

## 7.1 v-if

v-if 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回 truthy 值的时候被渲染。

<h1 v-if="awesome">Vue is awesome!</h1>

也可以用 v-else 添加一个“else 块”：

<h1 v-if="awesome">Vue is awesome!</h1>

<h1 v-else>Oh no 😢</h1>

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

因为 v-if 是一个指令，所以必须将它添加到一个元素上。但是如果想切换多个元素呢？此时可以把一个 <template> 元素当做不可见的包裹元素，并在上面使用 v-if。最终的渲染结果将不包含 <template> 元素。

<template v-if="ok">

<h1>Title</h1>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</template>

### 7.1.2 v-else

你可以使用 v-else 指令来表示 v-if 的“else 块”：

<div v-if="Math.random() > 0.5">

Now you see me

</div>

<div v-else>

Now you don't

</div>

v-else 元素必须紧跟在带 v-if 或者 v-else-if 的元素的后面，否则它将不会被识别。

### 7.1.3 v-else-if

2.1.0 新增

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

<div v-if="type === 'A'">

A

</div>

<div v-else-if="type === 'B'">

B

</div>

<div v-else-if="type === 'C'">

C

</div>

<div v-else>

Not A/B/C

</div>

类似于 v-else，v-else-if 也必须紧跟在带 v-if 或者 v-else-if 的元素之后。

### 7.1.4 用 key 管理可复用的元素

Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。这么做除了使 Vue 变得非常快之外，还有其它一些好处。例如，如果你允许用户在不同的登录方式之间切换：

<template v-if="loginType === 'username'">

<label>Username</label>

<input placeholder="Enter your username">

</template>

<template v-else>

<label>Email</label>

<input placeholder="Enter your email address">

</template>

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

自己动手试一试，在输入框中输入一些文本，然后按下切换按钮：

这样也不总是符合实际需求，所以 Vue 为你提供了一种方式来表达“这两个元素是完全独立的，不要复用它们”。只需添加一个具有唯一值的 key attribute 即可：

<template v-if="loginType === 'username'">

<label>Username</label>

<input placeholder="Enter your username" key="username-input">

</template>

<template v-else>

<label>Email</label>

<input placeholder="Enter your email address" key="email-input">

</template>

现在，每次切换时，输入框都将被重新渲染。请看：

注意，<label> 元素仍然会被高效地复用，因为它们没有添加 key attribute。

## 7.2 v-show

另一个用于根据条件展示元素的选项是 v-show 指令。用法大致一样：

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

不同的是带有 v-show 的元素始终会被渲染并保留在 DOM 中。v-show 只是简单地切换元素的 CSS property display。

注意，v-show 不支持 <template> 元素，也不支持 v-else。

## 7.3 v-if vs v-show

v-if 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

v-if 也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

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

一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。

## 7.4 v-if 与 v-for 一起使用

不推荐同时使用 v-if 和 v-for。请查阅风格指南以获取更多信息。

当 v-if 与 v-for 一起使用时，v-for 具有比 v-if 更高的优先级。请查阅列表渲染指南以获取详细信息。

# 第8章 列表渲染

## 8.1 用 v-for 把一个数组对应为一组元素

我们可以用 v-for 指令基于一个数组来渲染一个列表。v-for 指令需要使用 item in items 形式的特殊语法，其中 items 是源数据数组，而 item 则是被迭代的数组元素的别名。

<ul id="example-1">

<li v-for="item in items" :key="item.message">

{{ item.message }}

</li>

</ul>

var example1 = new Vue({

el: '#example-1',

data: {

items: [

{ message: 'Foo' },

{ message: 'Bar' }

]

}

})

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

<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' }

]

}

})

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

<div v-for="item of items"></div>

## 8.2 在 v-for 里使用对象

你也可以用 v-for 来遍历一个对象的 property。

<ul id="v-for-object" class="demo">

<li v-for="value in object">

{{ value }}

</li>

</ul>

new Vue({

el: '#v-for-object',

data: {

object: {

title: 'How to do lists in Vue',

author: 'Jane Doe',

publishedAt: '2016-04-10'

}

}

})

你也可以提供第二个的参数为 property 名称 (也就是键名)：

<div v-for="(value, name) in object">

{{ name }}: {{ value }}

</div>

还可以用第三个参数作为索引：

<div v-for="(value, name, index) in object">

{{ index }}. {{ name }}: {{ value }}

</div>

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

## 8.3 维护状态

当 Vue 正在更新使用 v-for 渲染的元素列表时，它默认使用“就地更新”的策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序，而是就地更新每个元素，并且确保它们在每个索引位置正确渲染。这个类似 Vue 1.x 的 track-by="$index"。

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出。

为了给 Vue 一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key attribute：

<div v-for="item in items" v-bind:key="item.id">

<!-- 内容 -->

</div>

建议尽可能在使用 v-for 时提供 key attribute，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升。

因为它是 Vue 识别节点的一个通用机制，key 并不仅与 v-for 特别关联。后面我们将在指南中看到，它还具有其它用途。

不要使用对象或数组之类的非基本类型值作为 v-for 的 key。请用字符串或数值类型的值。

更多 key attribute 的细节用法请移步至 key 的 API 文档。

## 8.4 数组更新检测

### 8.4.1 变更方法

Vue 将被侦听的数组的变更方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

push()

pop()

shift()

unshift()

splice()

sort()

reverse()

你可以打开控制台，然后对前面例子的 items 数组尝试调用变更方法。比如 example1.items.push({ message: 'Baz' })。

### 8.4.2 替换数组

变更方法，顾名思义，会变更调用了这些方法的原始数组。相比之下，也有非变更方法，例如 filter()、concat() 和 slice()。它们不会变更原始数组，而总是返回一个新数组。当使用非变更方法时，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 为了使得 DOM 元素得到最大范围的重用而实现了一些智能的启发式方法，所以用一个含有相同元素的数组去替换原来的数组是非常高效的操作。

### 8.4.3 注意事项

由于 JavaScript 的限制，Vue 不能检测数组和对象的变化。深入响应式原理中有相关的讨论。

## 8.5 显示过滤/排序后的结果

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

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

})

}

}

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

<ul v-for="set in sets">

<li v-for="n in even(set)">{{ n }}</li>

</ul>

data: {

sets: [[ 1, 2, 3, 4, 5 ], [6, 7, 8, 9, 10]]

},

methods: {

even: function (numbers) {

return numbers.filter(function (number) {

return number % 2 === 0

})

}

}

## 8.6 在 v-for 里使用值范围

v-for 也可以接受整数。在这种情况下，它会把模板重复对应次数。

<div>

<span v-for="n in 10">{{ n }} </span>

</div>

## 8.7 在 <template> 上使用 v-for

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

<ul>

<template v-for="item in items">

<li>{{ item.msg }}</li>

<li class="divider" role="presentation"></li>

</template>

</ul>

## 8.8 v-for 与 v-if 一同使用

注意我们不推荐在同一元素上使用 v-if 和 v-for。更多细节可查阅风格指南。

当它们处于同一节点，v-for 的优先级比 v-if 更高，这意味着 v-if 将分别重复运行于每个 v-for 循环中。当你只想为部分项渲染节点时，这种优先级的机制会十分有用，如下：

<li v-for="todo in todos" v-if="!todo.isComplete">

{{ todo }}

</li>

上面的代码将只渲染未完成的 todo。

而如果你的目的是有条件地跳过循环的执行，那么可以将 v-if 置于外层元素 (或 <template>) 上。如：

<ul v-if="todos.length">

<li v-for="todo in todos">

{{ todo }}

</li>

</ul>

<p v-else>No todos left!</p>

## 8.9 在组件上使用 v-for

这部分内容假定你已经了解组件相关知识。你也完全可以先跳过它，以后再回来查看。

在自定义组件上，你可以像在任何普通元素上一样使用 v-for。

<my-component v-for="item in items" :key="item.id"></my-component>

2.2.0+ 的版本里，当在组件上使用 v-for 时，key 现在是必须的。

然而，任何数据都不会被自动传递到组件里，因为组件有自己独立的作用域。为了把迭代数据传递到组件里，我们要使用 prop：

<my-component

v-for="(item, index) in items"

v-bind:item="item"

v-bind:index="index"

v-bind:key="item.id"

></my-component>

不自动将 item 注入到组件里的原因是，这会使得组件与 v-for 的运作紧密耦合。明确组件数据的来源能够使组件在其他场合重复使用。

下面是一个简单的 todo 列表的完整例子：

<div id="todo-list-example">

<form v-on:submit.prevent="addNewTodo">

<label for="new-todo">Add a todo</label>

<input

v-model="newTodoText"

id="new-todo"

placeholder="E.g. Feed the cat"

>

<button>Add</button>

</form>

<ul>

<li

is="todo-item"

v-for="(todo, index) in todos"

v-bind:key="todo.id"

v-bind:title="todo.title"

v-on:remove="todos.splice(index, 1)"

></li>

</ul>

</div>

注意这里的 is="todo-item" attribute。这种做法在使用 DOM 模板时是十分必要的，因为在 <ul> 元素内只有 <li> 元素会被看作有效内容。这样做实现的效果与 <todo-item> 相同，但是可以避开一些潜在的浏览器解析错误。查看 DOM 模板解析说明 来了解更多信息。

Vue.component('todo-item', {

template: '\

<li>\

{{ title }}\

<button v-on:click="$emit(\'remove\')">Remove</button>\

</li>\

',

props: ['title']

})

new Vue({

el: '#todo-list-example',

data: {

newTodoText: '',

todos: [

{

id: 1,

title: 'Do the dishes',

},

{

id: 2,

title: 'Take out the trash',

},

{

id: 3,

title: 'Mow the lawn'

}

],

nextTodoId: 4

},

methods: {

addNewTodo: function () {

this.todos.push({

id: this.nextTodoId++,

title: this.newTodoText

})

this.newTodoText = ''

}

}

})

# 第9章 事件处理

## 9.1 监听事件

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

示例：

<div id="example-1">

<button v-on:click="counter += 1">Add 1</button>

<p>The button above has been clicked {{ counter }} times.</p>

</div>

var example1 = new Vue({

el: '#example-1',

data: {

counter: 0

}

})

## 9.2 事件处理方法

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

示例：

<div id="example-2">

<!-- `greet` 是在下面定义的方法名 -->

<button v-on:click="greet">Greet</button>

</div>

var example2 = new Vue({

el: '#example-2',

data: {

name: 'Vue.js'

},

// 在 `methods` 对象中定义方法

methods: {

greet: function (event) {

// `this` 在方法里指向当前 Vue 实例

alert('Hello ' + this.name + '!')

// `event` 是原生 DOM 事件

if (event) {

alert(event.target.tagName)

}

}

}

})

// 也可以用 JavaScript 直接调用方法

example2.greet() // => 'Hello Vue.js!'

## 9.3 内联处理器中的方法

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

<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)

}

}

})

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

<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)

}

}

## 9.4 事件修饰符

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

为了解决这个问题，Vue.js 为 v-on 提供了事件修饰符。之前提过，修饰符是由点开头的指令后缀来表示的。

.stop

.prevent

.capture

.self

.once

.passive

<!-- 阻止单击事件继续传播 -->

<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>

<!-- 添加事件监听器时使用事件捕获模式 -->

<!-- 即内部元素触发的事件先在此处理，然后才交由内部元素进行处理 -->

<div v-on:click.capture="doThis">...</div>

<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->

<!-- 即事件不是从内部元素触发的 -->

<div v-on:click.self="doThat">...</div>

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 v-on:click.prevent.self 会阻止所有的点击，而 v-on:click.self.prevent 只会阻止对元素自身的点击。

**2.1.4 新增**

<!-- 点击事件将只会触发一次 -->

<a v-on:click.once="doThis"></a>

不像其它只能对原生的 DOM 事件起作用的修饰符，.once 修饰符还能被用到自定义的组件事件上。如果你还没有阅读关于组件的文档，现在大可不必担心。

**2.3.0 新增**

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

<!-- 滚动事件的默认行为 (即滚动行为) 将会立即触发 -->

<!-- 而不会等待 `onScroll` 完成 -->

<!-- 这其中包含 `event.preventDefault()` 的情况 -->

<div v-on:scroll.passive="onScroll">...</div>

这个 .passive 修饰符尤其能够提升移动端的性能。

不要把 .passive 和 .prevent 一起使用，因为 .prevent 将会被忽略，同时浏览器可能会向你展示一个警告。请记住，.passive 会告诉浏览器你不想阻止事件的默认行为。

## 9.5 按键修饰符

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

<!-- 只有在 `key` 是 `Enter` 时调用 `vm.submit()` -->

<input v-on:keyup.enter="submit">

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

<input v-on:keyup.page-down="onPageDown">

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

### 9.5.1 按键码

keyCode 的事件用法已经被废弃了并可能不会被最新的浏览器支持。

使用 keyCode attribute 也是允许的：

<input v-on:keyup.13="submit">

为了在必要的情况下支持旧浏览器，Vue 提供了绝大多数常用的按键码的别名：

.enter

.tab

.delete (捕获“删除”和“退格”键)

.esc

.space

.up

.down

.left

.right

有一些按键 (.esc 以及所有的方向键) 在 IE9 中有不同的 key 值, 如果你想支持 IE9，这些内置的别名应该是首选。

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

// 可以使用 `v-on:keyup.f1`

Vue.config.keyCodes.f1 = 112

## 9.6 系统修饰键

**2.1.0 新增**

可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。

.ctrl

.alt

.shift

.meta

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

例如：

<!-- Alt + C -->

<input v-on:keyup.alt.67="clear">

<!-- Ctrl + Click -->

<div v-on:click.ctrl="doSomething">Do something</div>

请注意修饰键与常规按键不同，在和 keyup 事件一起用时，事件触发时修饰键必须处于按下状态。换句话说，只有在按住 ctrl 的情况下释放其它按键，才能触发 keyup.ctrl。而单单释放 ctrl 也不会触发事件。如果你想要这样的行为，请为 ctrl 换用 keyCode：keyup.17。

### 9.6.1 .exact 修饰符

**2.5.0 新增**

.exact 修饰符允许你控制由精确的系统修饰符组合触发的事件。

<!-- 即使 Alt 或 Shift 被一同按下时也会触发 -->

<button v-on:click.ctrl="onClick">A</button>

<!-- 有且只有 Ctrl 被按下的时候才触发 -->

<button v-on:click.ctrl.exact="onCtrlClick">A</button>

<!-- 没有任何系统修饰符被按下的时候才触发 -->

<button v-on:click.exact="onClick">A</button>

### 9.6.2 鼠标按钮修饰符

**2.2.0 新增**

.left

.right

.middle

这些修饰符会限制处理函数仅响应特定的鼠标按钮。

## 9.7 为什么在 HTML 中监听事件？

你可能注意到这种事件监听的方式违背了关注点分离 (separation of concern) 这个长期以来的优良传统。但不必担心，因为所有的 Vue.js 事件处理方法和表达式都严格绑定在当前视图的 ViewModel 上，它不会导致任何维护上的困难。实际上，使用 v-on 有几个好处：

扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。

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。

当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何清理它们。

# 第10章 表单输入绑定

## 10.1 基础用法

你可以用 v-model 指令在表单 <input>、<textarea> 及 <select> 元素上创建双向数据绑定。它会根据控件类型自动选取正确的方法来更新元素。尽管有些神奇，但 v-model 本质上不过是语法糖。它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理。

v-model 会忽略所有表单元素的 value、checked、selected attribute 的初始值而总是将 Vue 实例的数据作为数据来源。你应该通过 JavaScript 在组件的 data 选项中声明初始值。

v-model 在内部为不同的输入元素使用不同的 property 并抛出不同的事件：

text 和 textarea 元素使用 value property 和 input 事件；

checkbox 和 radio 使用 checked property 和 change 事件；

select 字段将 value 作为 prop 并将 change 作为事件。

对于需要使用输入法 (如中文、日文、韩文等) 的语言，你会发现 v-model 不会在输入法组合文字过程中得到更新。如果你也想处理这个过程，请使用 input 事件。

### 10.1.1 文本

<input v-model="message" placeholder="edit me">

<p>Message is: {{ message }}</p>

### 10.1.2 多行文本

<span>Multiline message is:</span>

<p style="white-space: pre-line;">{{ message }}</p>

<br>

<textarea v-model="message" placeholder="add multiple lines"></textarea>

### 10.1.3 复选框

**单个复选框，绑定到布尔值：**

<input type="checkbox" id="checkbox" v-model="checked">

<label for="checkbox">{{ checked }}</label>

**多个复选框，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: []

}

})

### 10.1.4 单选按钮

<div id="example-4">

<input type="radio" id="one" value="One" v-model="picked">

<label for="one">One</label>

<br>

<input type="radio" id="two" value="Two" v-model="picked">

<label for="two">Two</label>

<br>

<span>Picked: {{ picked }}</span>

</div>

new Vue({

el: '#example-4',

data: {

picked: ''

}

})

### 10.1.5 选择框

**单选时：**

<div id="example-5">

<select v-model="selected">

<option disabled value="">请选择</option>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

<span>Selected: {{ selected }}</span>

</div>

new Vue({

el: '...',

data: {

selected: ''

}

})

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

**多选时 (绑定到一个数组)：**

<div id="example-6">

<select v-model="selected" multiple style="width: 50px;">

<option>A</option>

<option>B</option>

<option>C</option>

</select>

<br>

<span>Selected: {{ selected }}</span>

</div>

new Vue({

el: '#example-6',

data: {

selected: []

}

})

**用 v-for 渲染的动态选项：**

<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' }

]

}

})

## 10.2 值绑定

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

<!-- 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">

<!-- 当选中第一个选项时，`selected` 为字符串 "abc" -->

<select v-model="selected">

<option value="abc">ABC</option>

</select>

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

### 10.2.1 复选框

<input

type="checkbox"

v-model="toggle"

true-value="yes"

false-value="no"

>

// 当选中时

vm.toggle === 'yes'

// 当没有选中时

vm.toggle === 'no'

这里的 true-value 和 false-value attribute 并不会影响输入控件的 value attribute，因为浏览器在提交表单时并不会包含未被选中的复选框。如果要确保表单中这两个值中的一个能够被提交，(即“yes”或“no”)，请换用单选按钮。

### 10.2.2 单选按钮

<input type="radio" v-model="pick" v-bind:value="a">

// 当选中时

vm.pick === vm.a

### 10.2.3 选择框的选项

<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

## 10.3 修饰符

### 10.3.1 .lazy

在默认情况下，v-model 在每次 input 事件触发后将输入框的值与数据进行同步 (除了上述输入法组合文字时)。你可以添加 lazy 修饰符，从而转为在 change 事件\_之后\_进行同步：

<!-- 在“change”时而非“input”时更新 -->

<input v-model.lazy="msg">

### 10.3.2 .number

如果想自动将用户的输入值转为数值类型，可以给 v-model 添加 number 修饰符：

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

这通常很有用，因为即使在 type="number" 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 parseFloat() 解析，则会返回原始的值。

### 10.3.3 .trim

如果要自动过滤用户输入的首尾空白字符，可以给 v-model 添加 trim 修饰符：

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

## 10.4 在组件上使用 v-model

如果你还不熟悉 Vue 的组件，可以暂且跳过这里。

HTML 原生的输入元素类型并不总能满足需求。幸好，Vue 的组件系统允许你创建具有完全自定义行为且可复用的输入组件。这些输入组件甚至可以和 v-model 一起使用！

要了解更多，请参阅组件指南中的自定义输入组件。

# 第11章 组件基础

## 11.1 这里有一个 Vue 组件的示例：

这里有一个 Vue 组件的示例：

// 定义一个名为 button-counter 的新组件

Vue.component('button-counter', {

data: function () {

return {

count: 0

}

},

template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'

})

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

<div id="components-demo">

<button-counter></button-counter>

</div>

new Vue({ el: '#components-demo' })

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

## 11.2 组件的复用

你可以将组件进行任意次数的复用：

<div id="components-demo">

<button-counter></button-counter>

<button-counter></button-counter>

<button-counter></button-counter>

</div>

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

### 11.2.1 data 必须是一个函数

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

data: {

count: 0

}

取而代之的是，一个组件的 data 选项必须是一个函数，因此每个实例可以维护一份被返回对象的独立的拷贝：

data: function () {

return {

count: 0

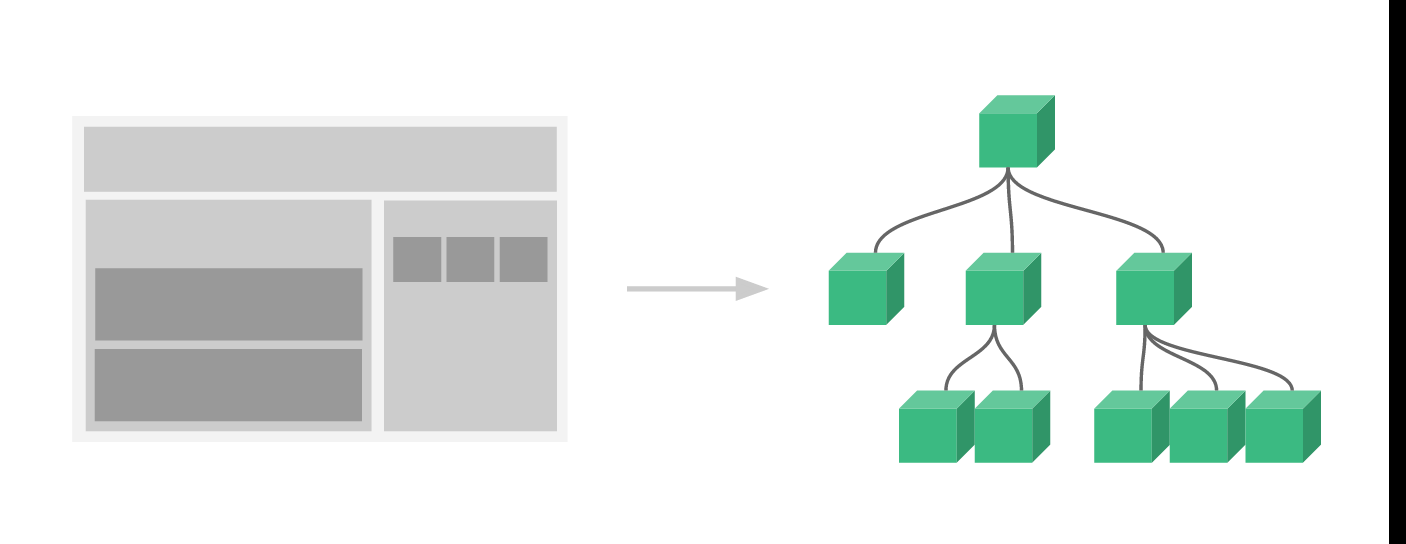
}

}

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

## 11.3 组件的组织

通常一个应用会以一棵嵌套的组件树的形式来组织：



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

为了能在模板中使用，这些组件必须先注册以便 Vue 能够识别。这里有两种组件的注册类型：全局注册和局部注册。至此，我们的组件都只是通过 Vue.component 全局注册的：

Vue.component('my-component-name', {

// ... options ...

})

全局注册的组件可以用在其被注册之后的任何 (通过 new Vue) 新创建的 Vue 根实例，也包括其组件树中的所有子组件的模板中。

到目前为止，关于组件注册你需要了解的就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把组件注册读完。

## 11.4 通过 Prop 向子组件传递数据

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

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

Vue.component('blog-post', {

props: ['title'],

template: '<h3>{{ title }}</h3>'

})

一个组件默认可以拥有任意数量的 prop，任何值都可以传递给任何 prop。在上述模板中，你会发现我们能够在组件实例中访问这个值，就像访问 data 中的值一样。

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

<blog-post title="My journey with Vue"></blog-post>

<blog-post title="Blogging with Vue"></blog-post>

<blog-post title="Why Vue is so fun"></blog-post>

然而在一个典型的应用中，你可能在 data 里有一个博文的数组：

new Vue({

el: '#blog-post-demo',

data: {

posts: [

{ id: 1, title: 'My journey with Vue' },

{ id: 2, title: 'Blogging with Vue' },

{ id: 3, title: 'Why Vue is so fun' }

]

}

})

并想要为每篇博文渲染一个组件：

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:title="post.title"

></blog-post>

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

到目前为止，关于 prop 你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把 prop 读完。

## 11.5 单个根元素

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

<h3>{{ title }}</h3>

最最起码，你会包含这篇博文的正文：

<h3>{{ title }}</h3>

<div v-html="content"></div>

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

<div class="blog-post">

<h3>{{ title }}</h3>

<div v-html="content"></div>

</div>

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

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:title="post.title"

v-bind:content="post.content"

v-bind:publishedAt="post.publishedAt"

v-bind:comments="post.comments"

></blog-post>

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

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:post="post"

></blog-post>

Vue.component('blog-post', {

props: ['post'],

template: `

<div class="blog-post">

<h3>{{ post.title }}</h3>

<div v-html="post.content"></div>

</div>

`

})

上述的这个和一些接下来的示例使用了 JavaScript 的模板字符串来让多行的模板更易读。它们在 IE 下并没有被支持，所以如果你需要在不 (经过 Babel 或 TypeScript 之类的工具) 编译的情况下支持 IE，请使用折行转义字符取而代之。

现在，不论何时为 post 对象添加一个新的 property，它都会自动地在 <blog-post> 内可用。

## 11.6 监听子组件事件

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

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

new Vue({

el: '#blog-posts-events-demo',

data: {

posts: [/\* ... \*/],

postFontSize: 1

}

})

它可以在模板中用来控制所有博文的字号：

<div id="blog-posts-events-demo">

<div :style="{ fontSize: postFontSize + 'em' }">

<blog-post

v-for="post in posts"

v-bind:key="post.id"

v-bind:post="post"

></blog-post>

</div>

</div>

现在我们在每篇博文正文之前添加一个按钮来放大字号：

Vue.component('blog-post', {

props: ['post'],

template: `

<div class="blog-post">

<h3>{{ post.title }}</h3>

<button>

Enlarge text

</button>

<div v-html="post.content"></div>

</div>

`

})

问题是这个按钮不会做任何事：

<button>

Enlarge text

</button>

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

<blog-post

...

v-on:enlarge-text="postFontSize += 0.1"

></blog-post>

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

<button v-on:click="$emit('enlarge-text')">

Enlarge text

</button>

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

### 11.6.1 使用事件抛出一个值

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

<button v-on:click="$emit('enlarge-text', 0.1)">

Enlarge text

</button>

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

<blog-post

...

v-on:enlarge-text="postFontSize += $event"

></blog-post>

或者，如果这个事件处理函数是一个方法：

<blog-post

...

v-on:enlarge-text="onEnlargeText"

></blog-post>

那么这个值将会作为第一个参数传入这个方法：

methods: {

onEnlargeText: function (enlargeAmount) {

this.postFontSize += enlargeAmount

}

}

### 11.6.2 在组件上使用 v-model

自定义事件也可以用于创建支持 v-model 的自定义输入组件。记住：

<input v-model="searchText">

等价于：

<input

v-bind:value="searchText"

v-on:input="searchText = $event.target.value"

>

当用在组件上时，v-model 则会这样：

<custom-input

v-bind:value="searchText"

v-on:input="searchText = $event"

></custom-input>

为了让它正常工作，这个组件内的 <input> 必须：

将其 value attribute 绑定到一个名叫 value 的 prop 上

在其 input 事件被触发时，将新的值通过自定义的 input 事件抛出

写成代码之后是这样的：

Vue.component('custom-input', {

props: ['value'],

template: `

<input

v-bind:value="value"

v-on:input="$emit('input', $event.target.value)"

>

`

})

现在 v-model 就应该可以在这个组件上完美地工作起来了：

<custom-input v-model="searchText"></custom-input>

到目前为止，关于组件自定义事件你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把自定义事件读完。

## 11.7 通过插槽分发内容

和 HTML 元素一样，我们经常需要向一个组件传递内容，像这样：

<alert-box>

Something bad happened.

</alert-box>

可能会渲染出这样的东西：

幸好，Vue 自定义的 <slot> 元素让这变得非常简单：

Vue.component('alert-box', {

template: `

<div class="demo-alert-box">

<strong>Error!</strong>

<slot></slot>

</div>

`

})

如你所见，我们只要在需要的地方加入插槽就行了——就这么简单！

到目前为止，关于插槽你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把插槽读完。

## 11.8 动态组件

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

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

<!-- 组件会在 `currentTabComponent` 改变时改变 -->

<component v-bind:is="currentTabComponent"></component>

在上述示例中，currentTabComponent 可以包括

已注册组件的名字，或

一个组件的选项对象

你可以在这里查阅并体验完整的代码，或在这个版本了解绑定组件选项对象，而不是已注册组件名的示例。

请留意，这个 attribute 可以用于常规 HTML 元素，但这些元素将被视为组件，这意味着所有的 attribute 都会作为 DOM attribute 被绑定。对于像 value 这样的 property，若想让其如预期般工作，你需要使用 .prop 修饰器。

到目前为止，关于动态组件你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把动态和异步组件读完。

## 11.9 解析 DOM 模板时的注意事项

有些 HTML 元素，诸如 <ul>、<ol>、<table> 和 <select>，对于哪些元素可以出现在其内部是有严格限制的。而有些元素，诸如 <li>、<tr> 和 <option>，只能出现在其它某些特定的元素内部。

这会导致我们使用这些有约束条件的元素时遇到一些问题。例如：

<table>

<blog-post-row></blog-post-row>

</table>

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

<table>

<tr is="blog-post-row"></tr>

</table>

需要注意的是如果我们从以下来源使用模板的话，这条限制是不存在的：

字符串 (例如：template: '...')

单文件组件 (.vue)

<script type="text/x-template">

到这里，你需要了解的解析 DOM 模板时的注意事项——实际上也是 Vue 的全部必要内容，大概就是这些了。恭喜你！接下来还有很多东西要去学习，不过首先，我们推荐你先休息一下，试用一下 Vue，自己随意做些好玩的东西。

如果你感觉已经掌握了这些知识，我们推荐你再回来把完整的组件指南，包括侧边栏中组件深入章节的所有页面读完。

# 第12章 组件注册

## 12.1 组件名

在注册一个组件的时候，我们始终需要给它一个名字。比如在全局注册的时候我们已经看到了：

Vue.component('my-component-name', { /\* ... \*/ })

该组件名就是 Vue.component 的第一个参数。

你给予组件的名字可能依赖于你打算拿它来做什么。当直接在 DOM 中使用一个组件 (而不是在字符串模板或单文件组件) 的时候，我们强烈推荐遵循 W3C 规范中的自定义组件名 (字母全小写且必须包含一个连字符)。这会帮助你避免和当前以及未来的 HTML 元素相冲突。

你可以在风格指南中查阅到关于组件名的其它建议。

### 12.1.1 组件名大小写

定义组件名的方式有两种：

使用 kebab-case

Vue.component('my-component-name', { /\* ... \*/ })

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

使用 PascalCase

Vue.component('MyComponentName', { /\* ... \*/ })

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

## 12.2 全局注册

到目前为止，我们只用过 Vue.component 来创建组件：

Vue.component('my-component-name', {

// ... 选项 ...

})

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

Vue.component('component-a', { /\* ... \*/ })

Vue.component('component-b', { /\* ... \*/ })

Vue.component('component-c', { /\* ... \*/ })

new Vue({ el: '#app' })

<div id="app">

<component-a></component-a>

<component-b></component-b>

<component-c></component-c>

</div>

在所有子组件中也是如此，也就是说这三个组件在各自内部也都可以相互使用。

## 12.3 局部注册

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

在这些情况下，你可以通过一个普通的 JavaScript 对象来定义组件：

var ComponentA = { /\* ... \*/ }

var ComponentB = { /\* ... \*/ }

var ComponentC = { /\* ... \*/ }

然后在 components 选项中定义你想要使用的组件：

new Vue({

el: '#app',

components: {

'component-a': ComponentA,

'component-b': ComponentB

}

})

对于 components 对象中的每个 property 来说，其 property 名就是自定义元素的名字，其 property 值就是这个组件的选项对象。

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

var ComponentA = { /\* ... \*/ }

var ComponentB = {

components: {

'component-a': ComponentA

},

// ...

}

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

import ComponentA from './ComponentA.vue'

export default {

components: {

ComponentA

},

// ...

}

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

用在模板中的自定义元素的名称

包含了这个组件选项的变量名

## 12.4 模块系统

如果你没有通过 import/require 使用一个模块系统，也许可以暂且跳过这个章节。如果你使用了，那么我们会为你提供一些特殊的使用说明和注意事项。

### 12.4.1 在模块系统中局部注册

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

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

import ComponentA from './ComponentA'

import ComponentC from './ComponentC'

export default {

components: {

ComponentA,

ComponentC

},

// ...

}

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

### 12.4.2 基础组件的自动化全局注册

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

所以会导致很多组件里都会有一个包含基础组件的长列表：

import BaseButton from './BaseButton.vue'

import BaseIcon from './BaseIcon.vue'

import BaseInput from './BaseInput.vue'

export default {

components: {

BaseButton,

BaseIcon,

BaseInput

}

}

而只是用于模板中的一小部分：

<BaseInput

v-model="searchText"

@keydown.enter="search"

/>

<BaseButton @click="search">

<BaseIcon name="search"/>

</BaseButton>

如果你恰好使用了 webpack (或在内部使用了 webpack 的 Vue CLI 3+)，那么就可以使用 require.context 只全局注册这些非常通用的基础组件。这里有一份可以让你在应用入口文件 (比如 src/main.js) 中全局导入基础组件的示例代码：

import Vue from 'vue'

import upperFirst from 'lodash/upperFirst'

import camelCase from 'lodash/camelCase'

const requireComponent = require.context(

// 其组件目录的相对路径

'./components',

// 是否查询其子目录

false,

// 匹配基础组件文件名的正则表达式

/Base[A-Z]\w+\.(vue|js)$/

)

requireComponent.keys().forEach(fileName => {

// 获取组件配置

const componentConfig = requireComponent(fileName)

// 获取组件的 PascalCase 命名

const componentName = upperFirst(

camelCase(

// 获取和目录深度无关的文件名

fileName

.split('/')

.pop()

.replace(/\.\w+$/, '')

)

)

// 全局注册组件

Vue.component(

componentName,

// 如果这个组件选项是通过 `export default` 导出的，

// 那么就会优先使用 `.default`，

// 否则回退到使用模块的根。

componentConfig.default || componentConfig

)

})

记住全局注册的行为必须在根 Vue 实例 (通过 new Vue) 创建之前发生。这里有一个真实项目情景下的示例。

# 第13 章 Prop

Prop 的大小写 (camelCase vs kebab-case)

HTML 中的 attribute 名是大小写不敏感的，所以浏览器会把所有大写字符解释为小写字符。这意味着当你使用 DOM 中的模板时，camelCase (驼峰命名法) 的 prop 名需要使用其等价的 kebab-case (短横线分隔命名) 命名：

Vue.component('blog-post', {

// 在 JavaScript 中是 camelCase 的

props: ['postTitle'],

template: '<h3>{{ postTitle }}</h3>'

})

<!-- 在 HTML 中是 kebab-case 的 -->

<blog-post post-title="hello!"></blog-post>

重申一次，如果你使用字符串模板，那么这个限制就不存在了。

## 13.1 Prop 类型

到这里，我们只看到了以字符串数组形式列出的 prop：

props: ['title', 'likes', 'isPublished', 'commentIds', 'author']

但是，通常你希望每个 prop 都有指定的值类型。这时，你可以以对象形式列出 prop，这些 property 的名称和值分别是 prop 各自的名称和类型：

props: {

title: String,

likes: Number,

isPublished: Boolean,

commentIds: Array,

author: Object,

callback: Function,

contactsPromise: Promise // or any other constructor

}

这不仅为你的组件提供了文档，还会在它们遇到错误的类型时从浏览器的 JavaScript 控制台提示用户。你会在这个页面接下来的部分看到类型检查和其它 prop 验证。

## 13.2 传递静态或动态 Prop

像这样，你已经知道了可以像这样给 prop 传入一个静态的值：

<blog-post title="My journey with Vue"></blog-post>

你也知道 prop 可以通过 v-bind 动态赋值，例如：

<!-- 动态赋予一个变量的值 -->

<blog-post v-bind:title="post.title"></blog-post>

<!-- 动态赋予一个复杂表达式的值 -->

<blog-post

v-bind:title="post.title + ' by ' + post.author.name"

></blog-post>

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

### 13.2.1 传入一个数字

<!-- 即便 `42` 是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->

<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->

<blog-post v-bind:likes="42"></blog-post>

<!-- 用一个变量进行动态赋值。-->

<blog-post v-bind:likes="post.likes"></blog-post>

### 13.2.2 传入一个布尔值

<!-- 包含该 prop 没有值的情况在内，都意味着 `true`。-->

<blog-post is-published></blog-post>

<!-- 即便 `false` 是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->

<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->

<blog-post v-bind:is-published="false"></blog-post>

<!-- 用一个变量进行动态赋值。-->

<blog-post v-bind:is-published="post.isPublished"></blog-post>

### 13.2.3 传入一个数组

<!-- 即便数组是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->

<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->

<blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>

<!-- 用一个变量进行动态赋值。-->

<blog-post v-bind:comment-ids="post.commentIds"></blog-post>

### 13.2.4 传入一个对象

<!-- 即便对象是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->

<!-- 这是一个 JavaScript 表达式而不是一个字符串。-->

<blog-post

v-bind:author="{

name: 'Veronica',

company: 'Veridian Dynamics'

}"

></blog-post>

<!-- 用一个变量进行动态赋值。-->

<blog-post v-bind:author="post.author"></blog-post>

### 13.2.5 传入一个对象的所有 property

如果你想要将一个对象的所有 property 都作为 prop 传入，你可以使用不带参数的 v-bind (取代 v-bind:prop-name)。例如，对于一个给定的对象 post：

post: {

id: 1,

title: 'My Journey with Vue'

}

下面的模板：

<blog-post v-bind="post"></blog-post>

等价于：

<blog-post

v-bind:id="post.id"

v-bind:title="post.title"

></blog-post>

## 13.3 单向数据流

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

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

这里有两种常见的试图变更一个 prop 的情形：

这个 prop 用来传递一个初始值；这个子组件接下来希望将其作为一个本地的 prop 数据来使用。在这种情况下，最好定义一个本地的 data property 并将这个 prop 用作其初始值：

props: ['initialCounter'],

data: function () {

return {

counter: this.initialCounter

}

}

这个 prop 以一种原始的值传入且需要进行转换。在这种情况下，最好使用这个 prop 的值来定义一个计算属性：

props: ['size'],

computed: {

normalizedSize: function () {

return this.size.trim().toLowerCase()

}

}

注意在 JavaScript 中对象和数组是通过引用传入的，所以对于一个数组或对象类型的 prop 来说，在子组件中改变变更这个对象或数组本身将会影响到父组件的状态。

## 13.4 Prop 验证

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

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

Vue.component('my-component', {

props: {

// 基础的类型检查 (`null` 和 `undefined` 会通过任何类型验证)

propA: Number,

// 多个可能的类型

propB: [String, Number],

// 必填的字符串

propC: {

type: String,

required: true

},

// 带有默认值的数字

propD: {

type: Number,

default: 100

},

// 带有默认值的对象

propE: {

type: Object,

// 对象或数组默认值必须从一个工厂函数获取

default: function () {

return { message: 'hello' }

}

},

// 自定义验证函数

propF: {

validator: function (value) {

// 这个值必须匹配下列字符串中的一个

return ['success', 'warning', 'danger'].includes(value)

}

}

}

})

当 prop 验证失败的时候，(开发环境构建版本的) Vue 将会产生一个控制台的警告。

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

### 13.4.1 type 可以是下列原生构造函数中的一个：

String

Number

Boolean

Array

Object

Date

Function

Symbol

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

function Person (firstName, lastName) {

this.firstName = firstName

this.lastName = lastName

}

你可以使用：

Vue.component('blog-post', {

props: {

author: Person

}

})

来验证 author prop 的值是否是通过 new Person 创建的。type 可以是下列原生构造函数中的一个：

String

Number

Boolean

Array

Object

Date

Function

Symbol

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

function Person (firstName, lastName) {

this.firstName = firstName

this.lastName = lastName

}

你可以使用：

Vue.component('blog-post', {

props: {

author: Person

}

})

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

## 13.5 非 Prop 的 Attribute

一个非 prop 的 attribute 是指传向一个组件，但是该组件并没有相应 prop 定义的 attribute。

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

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

<bootstrap-date-input data-date-picker="activated"></bootstrap-date-input>

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

### 13.5.1 替换/合并已有的 Attribute

想象一下 <bootstrap-date-input> 的模板是这样的：

<input type="date" class="form-control">

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

<bootstrap-date-input

data-date-picker="activated"

class="date-picker-theme-dark"

></bootstrap-date-input>

在这种情况下，我们定义了两个不同的 class 的值：

form-control，这是在组件的模板内设置好的

date-picker-theme-dark，这是从组件的父级传入的

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

### 13.5.2 禁用 Attribute 继承

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

Vue.component('my-component', {

inheritAttrs: false,

// ...

})

这尤其适合配合实例的 $attrs property 使用，该 property 包含了传递给一个组件的 attribute 名和 attribute 值，例如：

{

required: true,

placeholder: 'Enter your username'

}

有了 inheritAttrs: false 和 $attrs，你就可以手动决定这些 attribute 会被赋予哪个元素。在撰写基础组件的时候是常会用到的：

Vue.component('base-input', {

inheritAttrs: false,

props: ['label', 'value'],

template: `

<label>

{{ label }}

<input

v-bind="$attrs"

v-bind:value="value"

v-on:input="$emit('input', $event.target.value)"

>

</label>

`

})

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

这个模式允许你在使用基础组件的时候更像是使用原始的 HTML 元素，而不会担心哪个元素是真正的根元素：

<base-input

label="Username:"

v-model="username"

required

placeholder="Enter your username"

></base-input>

# 第14章 自定义事件

## 14.1 事件名

不同于组件和 prop，事件名不存在任何自动化的大小写转换。而是触发的事件名需要完全匹配监听这个事件所用的名称。举个例子，如果触发一个 camelCase 名字的事件：

this.$emit('myEvent')

则监听这个名字的 kebab-case 版本是不会有任何效果的：

<!-- 没有效果 -->

<my-component v-on:my-event="doSomething"></my-component>

不同于组件和 prop，事件名不会被用作一个 JavaScript 变量名或 property 名，所以就没有理由使用 camelCase 或 PascalCase 了。并且 v-on 事件监听器在 DOM 模板中会被自动转换为全小写 (因为 HTML 是大小写不敏感的)，所以 v-on:myEvent 将会变成 v-on:myevent——导致 myEvent 不可能被监听到。

因此，我们推荐你始终使用 kebab-case 的事件名。

## 14.2 自定义组件的 v-model

**2.2.0+ 新增**

一个组件上的 v-model 默认会利用名为 value 的 prop 和名为 input 的事件，但是像单选框、复选框等类型的输入控件可能会将 value attribute 用于不同的目的。model 选项可以用来避免这样的冲突：

Vue.component('base-checkbox', {

model: {

prop: 'checked',

event: 'change'

},

props: {

checked: Boolean

},

template: `

<input

type="checkbox"

v-bind:checked="checked"

v-on:change="$emit('change', $event.target.checked)"

>

`

})

现在在这个组件上使用 v-model 的时候：

<base-checkbox v-model="lovingVue"></base-checkbox>

这里的 lovingVue 的值将会传入这个名为 checked 的 prop。同时当 <base-checkbox> 触发一个 change 事件并附带一个新的值的时候，这个 lovingVue 的 property 将会被更新。

注意你仍然需要在组件的 props 选项里声明 checked 这个 prop。

## 14.3 将原生事件绑定到组件

你可能有很多次想要在一个组件的根元素上直接监听一个原生事件。这时，你可以使用 v-on 的 .native 修饰符：

<base-input v-on:focus.native="onFocus"></base-input>

在有的时候这是很有用的，不过在你尝试监听一个类似 <input> 的非常特定的元素时，这并不是个好主意。比如上述 <base-input> 组件可能做了如下重构，所以根元素实际上是一个 <label> 元素：

<label>

{{ label }}

<input

v-bind="$attrs"

v-bind:value="value"

v-on:input="$emit('input', $event.target.value)"

>

</label>

这时，父级的 .native 监听器将静默失败。它不会产生任何报错，但是 onFocus 处理函数不会如你预期地被调用。

为了解决这个问题，Vue 提供了一个 $listeners property，它是一个对象，里面包含了作用在这个组件上的所有监听器。例如：

{

focus: function (event) { /\* ... \*/ }

input: function (value) { /\* ... \*/ },

}

有了这个 $listeners property，你就可以配合 v-on="$listeners" 将所有的事件监听器指向这个组件的某个特定的子元素。对于类似 <input> 的你希望它也可以配合 v-model 工作的组件来说，为这些监听器创建一个类似下述 inputListeners 的计算属性通常是非常有用的：

Vue.component('base-input', {

inheritAttrs: false,

props: ['label', 'value'],

computed: {

inputListeners: function () {

var vm = this

// `Object.assign` 将所有的对象合并为一个新对象

return Object.assign({},

// 我们从父级添加所有的监听器

this.$listeners,

// 然后我们添加自定义监听器，

// 或覆写一些监听器的行为

{

// 这里确保组件配合 `v-model` 的工作

input: function (event) {

vm.$emit('input', event.target.value)

}

}

)

}

},

template: `

<label>

{{ label }}

<input

v-bind="$attrs"

v-bind:value="value"

v-on="inputListeners"

>

</label>

`

})

现在 <base-input> 组件是一个完全透明的包裹器了，也就是说它可以完全像一个普通的 <input> 元素一样使用了：所有跟它相同的 attribute 和监听器都可以工作，不必再使用 .native 监听器。

## 14.4 .sync 修饰符

**2.3.0+ 新增**

在有些情况下，我们可能需要对一个 prop 进行“双向绑定”。不幸的是，真正的双向绑定会带来维护上的问题，因为子组件可以变更父组件，且在父组件和子组件两侧都没有明显的变更来源。

这也是为什么我们推荐以 update:myPropName 的模式触发事件取而代之。举个例子，在一个包含 title prop 的假设的组件中，我们可以用以下方法表达对其赋新值的意图：

this.$emit('update:title', newTitle)

然后父组件可以监听那个事件并根据需要更新一个本地的数据 property。例如：

<text-document

v-bind:title="doc.title"

v-on:update:title="doc.title = $event"

></text-document>

为了方便起见，我们为这种模式提供一个缩写，即 .sync 修饰符：

<text-document v-bind:title.sync="doc.title"></text-document>

注意带有 .sync 修饰符的 v-bind 不能和表达式一起使用 (例如 v-bind:title.sync=”doc.title + ‘!’” 是无效的)。取而代之的是，你只能提供你想要绑定的 property 名，类似 v-model。

当我们用一个对象同时设置多个 prop 的时候，也可以将这个 .sync 修饰符和 v-bind 配合使用：

<text-document v-bind.sync="doc"></text-document>

这样会把 doc 对象中的每一个 property (如 title) 都作为一个独立的 prop 传进去，然后各自添加用于更新的 v-on 监听器。

将 v-bind.sync 用在一个字面量的对象上，例如 v-bind.sync=”{ title: doc.title }”，是无法正常工作的，因为在解析一个像这样的复杂表达式的时候，有很多边缘情况需要考虑。

# 第15章 插槽

该页面假设你已经阅读过了组件基础。如果你还对组件不太了解，推荐你先阅读它。

在 2.6.0 中，我们为具名插槽和作用域插槽引入了一个新的统一的语法 (即 v-slot 指令)。它取代了 slot 和 slot-scope 这两个目前已被废弃但未被移除且仍在文档中的 attribute。新语法的由来可查阅这份 RFC。

## 15.1 插槽内容

Vue 实现了一套内容分发的 API，这套 API 的设计灵感源自 Web Components 规范草案，将 <slot> 元素作为承载分发内容的出口。

它允许你像这样合成组件：

<navigation-link url="/profile">

Your Profile

</navigation-link>

然后你在 <navigation-link> 的模板中可能会写为：

<a

v-bind:href="url"

class="nav-link"

>

<slot></slot>

</a>

当组件渲染的时候，<slot></slot> 将会被替换为“Your Profile”。插槽内可以包含任何模板代码，包括 HTML：

<navigation-link url="/profile">

<!-- 添加一个 Font Awesome 图标 -->

<span class="fa fa-user"></span>

Your Profile

</navigation-link>

甚至其它的组件：

<navigation-link url="/profile">

<!-- 添加一个图标的组件 -->

<font-awesome-icon name="user"></font-awesome-icon>

Your Profile

</navigation-link>

如果 <navigation-link> 的 template 中没有包含一个 <slot> 元素，则该组件起始标签和结束标签之间的任何内容都会被抛弃。

## 15.2 编译作用域

当你想在一个插槽中使用数据时，例如：

<navigation-link url="/profile">

Logged in as {{ user.name }}

</navigation-link>

该插槽跟模板的其它地方一样可以访问相同的实例 property (也就是相同的“作用域”)，而不能访问 <navigation-link> 的作用域。例如 url 是访问不到的：

<navigation-link url="/profile">

Clicking here will send you to: {{ url }}

<!--

这里的 `url` 会是 undefined，因为其 (指该插槽的) 内容是

\_传递给\_ <navigation-link> 的而不是

在 <navigation-link> 组件\*内部\*定义的。

-->

</navigation-link>

作为一条规则，请记住：

父级模板里的所有内容都是在父级作用域中编译的；子模板里的所有内容都是在子作用域中编译的。

## 15.3 后备内容

有时为一个插槽设置具体的后备 (也就是默认的) 内容是很有用的，它只会在没有提供内容的时候被渲染。例如在一个 <submit-button> 组件中：

<button type="submit">

<slot></slot>

</button>

我们可能希望这个 <button> 内绝大多数情况下都渲染文本“Submit”。为了将“Submit”作为后备内容，我们可以将它放在 <slot> 标签内：

<button type="submit">

<slot>Submit</slot>

</button>

现在当我在一个父级组件中使用 <submit-button> 并且不提供任何插槽内容时：

<submit-button></submit-button>

后备内容“Submit”将会被渲染：

<button type="submit">

Submit

</button>

但是如果我们提供内容：

<submit-button>

Save

</submit-button>

则这个提供的内容将会被渲染从而取代后备内容：

<button type="submit">

Save

</button>

## 15.4 具名插槽

自 2.6.0 起有所更新。已废弃的使用 slot attribute 的语法在这里。

有时我们需要多个插槽。例如对于一个带有如下模板的 <base-layout> 组件：

<div class="container">

<header>

<!-- 我们希望把页头放这里 -->

</header>

<main>

<!-- 我们希望把主要内容放这里 -->

</main>

<footer>

<!-- 我们希望把页脚放这里 -->

</footer>

</div>

对于这样的情况，<slot> 元素有一个特殊的 attribute：name。这个 attribute 可以用来定义额外的插槽：

<div class="container">

<header>

<slot name="header"></slot>

</header>

<main>

<slot></slot>

</main>

<footer>

<slot name="footer"></slot>

</footer>

</div>

一个不带 name 的 <slot> 出口会带有隐含的名字“default”。

在向具名插槽提供内容的时候，我们可以在一个 <template> 元素上使用 v-slot 指令，并以 v-slot 的参数的形式提供其名称：

<base-layout>

<template v-slot:header>

<h1>Here might be a page title</h1>

</template>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

<template v-slot:footer>

<p>Here's some contact info</p>

</template>

</base-layout>

现在 <template> 元素中的所有内容都将会被传入相应的插槽。任何没有被包裹在带有 v-slot 的 <template> 中的内容都会被视为默认插槽的内容。

然而，如果你希望更明确一些，仍然可以在一个 <template> 中包裹默认插槽的内容：

<base-layout>

<template v-slot:header>

<h1>Here might be a page title</h1>

</template>

<template v-slot:default>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

</template>

<template v-slot:footer>

<p>Here's some contact info</p>

</template>

</base-layout>

任何一种写法都会渲染出：

<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>

注意 v-slot 只能添加在 <template> 上 (只有一种例外情况)，这一点和已经废弃的 slot attribute 不同。

## 15.5 作用域插槽

自 2.6.0 起有所更新。已废弃的使用 slot-scope attribute 的语法在这里。

有时让插槽内容能够访问子组件中才有的数据是很有用的。例如，设想一个带有如下模板的 <current-user> 组件：

<span>

<slot>{{ user.lastName }}</slot>

</span>

我们可能想换掉备用内容，用名而非姓来显示。如下：

<current-user>

{{ user.firstName }}

</current-user>

然而上述代码不会正常工作，因为只有 <current-user> 组件可以访问到 user，而我们提供的内容是在父级渲染的。

为了让 user 在父级的插槽内容中可用，我们可以将 user 作为 <slot> 元素的一个 attribute 绑定上去：

<span>

<slot v-bind:user="user">

{{ user.lastName }}

</slot>

</span>

绑定在 <slot> 元素上的 attribute 被称为插槽 prop。现在在父级作用域中，我们可以使用带值的 v-slot 来定义我们提供的插槽 prop 的名字：

<current-user>

<template v-slot:default="slotProps">

{{ slotProps.user.firstName }}

</template>

</current-user>

在这个例子中，我们选择将包含所有插槽 prop 的对象命名为 slotProps，但你也可以使用任意你喜欢的名字。

### 15.5.1 独占默认插槽的缩写语法

在上述情况下，当被提供的内容只有默认插槽时，组件的标签才可以被当作插槽的模板来使用。这样我们就可以把 v-slot 直接用在组件上：

<current-user v-slot:default="slotProps">

{{ slotProps.user.firstName }}

</current-user>

这种写法还可以更简单。就像假定未指明的内容对应默认插槽一样，不带参数的 v-slot 被假定对应默认插槽：

<current-user v-slot="slotProps">

{{ slotProps.user.firstName }}

</current-user>

注意默认插槽的缩写语法不能和具名插槽混用，因为它会导致作用域不明确：

<!-- 无效，会导致警告 -->

<current-user v-slot="slotProps">

{{ slotProps.user.firstName }}

<template v-slot:other="otherSlotProps">

slotProps is NOT available here

</template>

</current-user>

只要出现多个插槽，请始终为所有的插槽使用完整的基于 <template> 的语法：

<current-user>

<template v-slot:default="slotProps">

{{ slotProps.user.firstName }}

</template>

<template v-slot:other="otherSlotProps">

...

</template>

</current-user>

### 15.5.2 解构插槽 Prop

作用域插槽的内部工作原理是将你的插槽内容包裹在一个拥有单个参数的函数里：

function (slotProps) {

// 插槽内容

}

这意味着 v-slot 的值实际上可以是任何能够作为函数定义中的参数的 JavaScript 表达式。所以在支持的环境下 (单文件组件或现代浏览器)，你也可以使用 ES2015 解构来传入具体的插槽 prop，如下：

<current-user v-slot="{ user }">

{{ user.firstName }}

</current-user>

这样可以使模板更简洁，尤其是在该插槽提供了多个 prop 的时候。它同样开启了 prop 重命名等其它可能，例如将 user 重命名为 person：

<current-user v-slot="{ user: person }">

{{ person.firstName }}

</current-user>

你甚至可以定义后备内容，用于插槽 prop 是 undefined 的情形：

<current-user v-slot="{ user = { firstName: 'Guest' } }">

{{ user.firstName }}

</current-user>

## 15.6 动态插槽名

**2.6.0 新增**

动态指令参数也可以用在 v-slot 上，来定义动态的插槽名：

<base-layout>

<template v-slot:[dynamicSlotName]>

...

</template>

</base-layout>

## 15.7 具名插槽的缩写

**2.6.0 新增**

跟 v-on 和 v-bind 一样，v-slot 也有缩写，即把参数之前的所有内容 (v-slot:) 替换为字符 #。例如 v-slot:header 可以被重写为 #header：

<base-layout>

<template #header>

<h1>Here might be a page title</h1>

</template>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

<template #footer>

<p>Here's some contact info</p>

</template>

</base-layout>

然而，和其它指令一样，该缩写只在其有参数的时候才可用。这意味着以下语法是无效的：

<!-- 这样会触发一个警告 -->

<current-user #="{ user }">

{{ user.firstName }}

</current-user>

如果你希望使用缩写的话，你必须始终以明确插槽名取而代之：

<current-user #default="{ user }">

{{ user.firstName }}

</current-user>

## 15.8 其它示例

插槽 prop 允许我们将插槽转换为可复用的模板，这些模板可以基于输入的 prop 渲染出不同的内容。这在设计封装数据逻辑同时允许父级组件自定义部分布局的可复用组件时是最有用的。

例如，我们要实现一个 <todo-list> 组件，它是一个列表且包含布局和过滤逻辑：

<ul>

<li

v-for="todo in filteredTodos"

v-bind:key="todo.id"

>

{{ todo.text }}

</li>

</ul>

我们可以将每个 todo 作为父级组件的插槽，以此通过父级组件对其进行控制，然后将 todo 作为一个插槽 prop 进行绑定：

<ul>

<li

v-for="todo in filteredTodos"

v-bind:key="todo.id"

>

<!--

我们为每个 todo 准备了一个插槽，

将 `todo` 对象作为一个插槽的 prop 传入。

-->

<slot name="todo" v-bind:todo="todo">

<!-- 后备内容 -->

{{ todo.text }}

</slot>

</li>

</ul>

现在当我们使用 <todo-list> 组件的时候，我们可以选择为 todo 定义一个不一样的 <template> 作为替代方案，并且可以从子组件获取数据：

<todo-list v-bind:todos="todos">

<template v-slot:todo="{ todo }">

<span v-if="todo.isComplete">✓</span>

{{ todo.text }}

</template>

</todo-list>

这只是作用域插槽用武之地的冰山一角。想了解更多现实生活中的作用域插槽的用法，我们推荐浏览诸如 Vue Virtual Scroller、Vue Promised 和 Portal Vue 等库。

## 15.9 废弃了的语法

v-slot 指令自 Vue 2.6.0 起被引入，提供更好的支持 slot 和 slot-scope attribute 的 API 替代方案。v-slot 完整的由来参见这份 RFC。在接下来所有的 2.x 版本中 slot 和 slot-scope attribute 仍会被支持，但已经被官方废弃且不会出现在 Vue 3 中。

### 15.9.1 带有 slot attribute 的具名插槽

自 2.6.0 起被废弃。新推荐的语法请查阅这里。

在 <template> 上使用特殊的 slot attribute，可以将内容从父级传给具名插槽 (把这里提到过的 <base-layout> 组件作为示例)：

<base-layout>

<template slot="header">

<h1>Here might be a page title</h1>

</template>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

<template slot="footer">

<p>Here's some contact info</p>

</template>

</base-layout>

或者直接把 slot attribute 用在一个普通元素上：

<base-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>

</base-layout>

这里其实还有一个未命名插槽，也就是默认插槽，捕获所有未被匹配的内容。上述两个示例的 HTML 渲染结果均为：

<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>

### 15.9.2 带有 slot-scope attribute 的作用域插槽

**自 2.6.0 起被废弃。新推荐的语法请查阅这里。**

在 <template> 上使用特殊的 slot-scope attribute，可以接收传递给插槽的 prop (把这里提到过的 <slot-example> 组件作为示例)：

<slot-example>

<template slot="default" slot-scope="slotProps">

{{ slotProps.msg }}

</template>

</slot-example>

这里的 slot-scope 声明了被接收的 prop 对象会作为 slotProps 变量存在于 <template> 作用域中。你可以像命名 JavaScript 函数参数一样随意命名 slotProps。

这里的 slot="default" 可以被忽略为隐性写法：

<slot-example>

<template slot-scope="slotProps">

{{ slotProps.msg }}

</template>

</slot-example>

slot-scope attribute 也可以直接用于非 <template> 元素 (包括组件)：

<slot-example>

<span slot-scope="slotProps">

{{ slotProps.msg }}

</span>

</slot-example>

slot-scope 的值可以接收任何有效的可以出现在函数定义的参数位置上的 JavaScript 表达式。这意味着在支持的环境下 (单文件组件或现代浏览器)，你也可以在表达式中使用 ES2015 解构，如下：

<slot-example>

<span slot-scope="{ msg }">

{{ msg }}

</span>

</slot-example>

使用这里描述过的 <todo-list> 作为示例，与它等价的使用 slot-scope 的代码是：

<todo-list v-bind:todos="todos">

<template slot="todo" slot-scope="{ todo }">

<span v-if="todo.isComplete">✓</span>

{{ todo.text }}

</template>

</todo-list>

# 第16章 动态组件 & 异步组件

## 16.1 在动态组件上使用 keep-alive

我们之前在一个多标签的界面中使用 is attribute 来切换不同的组件：

<component v-bind:is="currentTabComponent"></component>

当在这些组件之间切换的时候，你有时会想保持这些组件的状态，以避免反复重新渲染导致的性能问题。例如我们来展开说一说这个多标签界面：

你会注意到，如果你选择了一篇文章，切换到 Archive 标签，然后再切换回 Posts，是不会继续展示你之前选择的文章的。这是因为你每次切换新标签的时候，Vue 都创建了一个新的 currentTabComponent 实例。

重新创建动态组件的行为通常是非常有用的，但是在这个案例中，我们更希望那些标签的组件实例能够被在它们第一次被创建的时候缓存下来。为了解决这个问题，我们可以用一个 <keep-alive> 元素将其动态组件包裹起来。

<!-- 失活的组件将会被缓存！-->

<keep-alive>

<component v-bind:is="currentTabComponent"></component>

</keep-alive>

来看看修改后的结果：

现在这个 Posts 标签保持了它的状态 (被选中的文章) 甚至当它未被渲染时也是如此。你可以在这个示例查阅到完整的代码。

注意这个 <keep-alive> 要求被切换到的组件都有自己的名字，不论是通过组件的 name 选项还是局部/全局注册。

你可以在 API 参考文档查阅更多关于 <keep-alive> 的细节。

## 16.2 异步组件

在大型应用中，我们可能需要将应用分割成小一些的代码块，并且只在需要的时候才从服务器加载一个模块。为了简化，Vue 允许你以一个工厂函数的方式定义你的组件，这个工厂函数会异步解析你的组件定义。Vue 只有在这个组件需要被渲染的时候才会触发该工厂函数，且会把结果缓存起来供未来重渲染。例如：

Vue.component('async-example', function (resolve, reject) {

setTimeout(function () {

// 向 `resolve` 回调传递组件定义

resolve({

template: '<div>I am async!</div>'

})

}, 1000)

})

如你所见，这个工厂函数会收到一个 resolve 回调，这个回调函数会在你从服务器得到组件定义的时候被调用。你也可以调用 reject(reason) 来表示加载失败。这里的 setTimeout 是为了演示用的，如何获取组件取决于你自己。一个推荐的做法是将异步组件和 webpack 的 code-splitting 功能一起配合使用：

Vue.component('async-webpack-example', function (resolve) {

// 这个特殊的 `require` 语法将会告诉 webpack

// 自动将你的构建代码切割成多个包，这些包

// 会通过 Ajax 请求加载

require(['./my-async-component'], resolve)

})

你也可以在工厂函数中返回一个 Promise，所以把 webpack 2 和 ES2015 语法加在一起，我们可以这样使用动态导入：

Vue.component(

'async-webpack-example',

// 这个动态导入会返回一个 `Promise` 对象。

() => import('./my-async-component')

)

当使用局部注册的时候，你也可以直接提供一个返回 Promise 的函数：

new Vue({

// ...

components: {

'my-component': () => import('./my-async-component')

}

})

如果你是一个 Browserify 用户同时喜欢使用异步组件，很不幸这个工具的作者明确表示异步加载“并不会被 Browserify 支持”，至少官方不会。Browserify 社区已经找到了一些变通方案，这些方案可能会对已存在的复杂应用有帮助。对于其它的场景，我们推荐直接使用 webpack，以拥有内置的头等异步支持。

### 16.2.1 处理加载状态

**2.3.0+ 新增**

这里的异步组件工厂函数也可以返回一个如下格式的对象：

const AsyncComponent = () => ({

// 需要加载的组件 (应该是一个 `Promise` 对象)

component: import('./MyComponent.vue'),

// 异步组件加载时使用的组件

loading: LoadingComponent,

// 加载失败时使用的组件

error: ErrorComponent,

// 展示加载时组件的延时时间。默认值是 200 (毫秒)

delay: 200,

// 如果提供了超时时间且组件加载也超时了，

// 则使用加载失败时使用的组件。默认值是：`Infinity`

timeout: 3000

})

注意如果你希望在 Vue Router 的路由组件中使用上述语法的话，你必须使用 Vue Router 2.4.0+ 版本。

# 第17章 处理边界情况

这里记录的都是和处理边界情况有关的功能，即一些需要对 Vue 的规则做一些小调整的特殊情况。不过注意这些功能都是有劣势或危险的场景的。我们会在每个案例中注明，所以当你使用每个功能的时候请稍加留意。

## 17.1 访问元素 & 组件

在绝大多数情况下，我们最好不要触达另一个组件实例内部或手动操作 DOM 元素。不过也确实在一些情况下做这些事情是合适的。

### 17.1.1 访问根实例

在每个 new Vue 实例的子组件中，其根实例可以通过 $root property 进行访问。例如，在这个根实例中：

// Vue 根实例

new Vue({

data: {

foo: 1

},

computed: {

bar: function () { /\* ... \*/ }

},

methods: {

baz: function () { /\* ... \*/ }

}

})

所有的子组件都可以将这个实例作为一个全局 store 来访问或使用。

// 获取根组件的数据

this.$root.foo

// 写入根组件的数据

this.$root.foo = 2

// 访问根组件的计算属性

this.$root.bar

// 调用根组件的方法

this.$root.baz()

对于 demo 或非常小型的有少量组件的应用来说这是很方便的。不过这个模式扩展到中大型应用来说就不然了。因此在绝大多数情况下，我们强烈推荐使用 Vuex 来管理应用的状态。

### 17.1.2 访问父级组件实例

和 $root 类似，$parent property 可以用来从一个子组件访问父组件的实例。它提供了一种机会，可以在后期随时触达父级组件，以替代将数据以 prop 的方式传入子组件的方式。

在绝大多数情况下，触达父级组件会使得你的应用更难调试和理解，尤其是当你变更了父级组件的数据的时候。当我们稍后回看那个组件的时候，很难找出那个变更是从哪里发起的。

另外在一些可能适当的时候，你需要特别地共享一些组件库。举个例子，在和 JavaScript API 进行交互而不渲染 HTML 的抽象组件内，诸如这些假设性的 Google 地图组件一样：

<google-map>

<google-map-markers v-bind:places="iceCreamShops"></google-map-markers>

</google-map>

这个 <google-map> 组件可以定义一个 map property，所有的子组件都需要访问它。在这种情况下 <google-map-markers> 可能想要通过类似 this.$parent.getMap 的方式访问那个地图，以便为其添加一组标记。你可以在这里查阅这种模式。

请留意，尽管如此，通过这种模式构建出来的那个组件的内部仍然是容易出现问题的。比如，设想一下我们添加一个新的 <google-map-region> 组件，当 <google-map-markers> 在其内部出现的时候，只会渲染那个区域内的标记：

<google-map>

<google-map-region v-bind:shape="cityBoundaries">

<google-map-markers v-bind:places="iceCreamShops"></google-map-markers>

</google-map-region>

</google-map>

那么在 <google-map-markers> 内部你可能发现自己需要一些类似这样的 hack：

var map = this.$parent.map || this.$parent.$parent.map

很快它就会失控。这也是我们针对需要向任意更深层级的组件提供上下文信息时推荐依赖注入的原因。

### 17.1.3 访问子组件实例或子元素

尽管存在 prop 和事件，有的时候你仍可能需要在 JavaScript 里直接访问一个子组件。为了达到这个目的，你可以通过 ref 这个 attribute 为子组件赋予一个 ID 引用。例如：

<base-input ref="usernameInput"></base-input>

现在在你已经定义了这个 ref 的组件里，你可以使用：

this.$refs.usernameInput

来访问这个 <base-input> 实例，以便不时之需。比如程序化地从一个父级组件聚焦这个输入框。在刚才那个例子中，该 <base-input> 组件也可以使用一个类似的 ref 提供对内部这个指定元素的访问，例如：

<input ref="input">

甚至可以通过其父级组件定义方法：

methods: {

// 用来从父级组件聚焦输入框

focus: function () {

this.$refs.input.focus()

}

}

这样就允许父级组件通过下面的代码聚焦 <base-input> 里的输入框：

this.$refs.usernameInput.focus()

当 ref 和 v-for 一起使用的时候，你得到的 ref 将会是一个包含了对应数据源的这些子组件的数组。

$refs 只会在组件渲染完成之后生效，并且它们不是响应式的。这仅作为一个用于直接操作子组件的“逃生舱”——你应该避免在模板或计算属性中访问 $refs。

### 17.1.4 依赖注入

在此之前，在我们描述访问父级组件实例的时候，展示过一个类似这样的例子：

<google-map>

<google-map-region v-bind:shape="cityBoundaries">

<google-map-markers v-bind:places="iceCreamShops"></google-map-markers>

</google-map-region>

</google-map>

在这个组件里，所有 <google-map> 的后代都需要访问一个 getMap 方法，以便知道要跟哪个地图进行交互。不幸的是，使用 $parent property 无法很好的扩展到更深层级的嵌套组件上。这也是依赖注入的用武之地，它用到了两个新的实例选项：provide 和 inject。

provide 选项允许我们指定我们想要提供给后代组件的数据/方法。在这个例子中，就是 <google-map> 内部的 getMap 方法：

provide: function () {

return {

getMap: this.getMap

}

}

然后在任何后代组件里，我们都可以使用 inject 选项来接收指定的我们想要添加在这个实例上的 property：

inject: ['getMap']

你可以在这里看到完整的示例。相比 $parent 来说，这个用法可以让我们在任意后代组件中访问 getMap，而不需要暴露整个 <google-map> 实例。这允许我们更好的持续研发该组件，而不需要担心我们可能会改变/移除一些子组件依赖的东西。同时这些组件之间的接口是始终明确定义的，就和 props 一样。

实际上，你可以把依赖注入看作一部分“大范围有效的 prop”，除了：

祖先组件不需要知道哪些后代组件使用它提供的 property

后代组件不需要知道被注入的 property 来自哪里

然而，依赖注入还是有负面影响的。它将你应用程序中的组件与它们当前的组织方式耦合起来，使重构变得更加困难。同时所提供的 property 是非响应式的。这是出于设计的考虑，因为使用它们来创建一个中心化规模化的数据跟使用 $root做这件事都是不够好的。如果你想要共享的这个 property 是你的应用特有的，而不是通用化的，或者如果你想在祖先组件中更新所提供的数据，那么这意味着你可能需要换用一个像 Vuex 这样真正的状态管理方案了。

你可以在 API 参考文档学习更多关于依赖注入的知识。

## 17.2 程序化的事件侦听器

现在，你已经知道了 $emit 的用法，它可以被 v-on 侦听，但是 Vue 实例同时在其事件接口中提供了其它的方法。我们可以：

通过 $on(eventName, eventHandler) 侦听一个事件

通过 $once(eventName, eventHandler) 一次性侦听一个事件

通过 $off(eventName, eventHandler) 停止侦听一个事件

你通常不会用到这些，但是当你需要在一个组件实例上手动侦听事件时，它们是派得上用场的。它们也可以用于代码组织工具。例如，你可能经常看到这种集成一个第三方库的模式：

// 一次性将这个日期选择器附加到一个输入框上

// 它会被挂载到 DOM 上。

mounted: function () {

// Pikaday 是一个第三方日期选择器的库

this.picker = new Pikaday({

field: this.$refs.input,

format: 'YYYY-MM-DD'

})

},

// 在组件被销毁之前，

// 也销毁这个日期选择器。

beforeDestroy: function () {

this.picker.destroy()

}

这里有两个潜在的问题：

它需要在这个组件实例中保存这个 picker，如果可以的话最好只有生命周期钩子可以访问到它。这并不算严重的问题，但是它可以被视为杂物。

我们的建立代码独立于我们的清理代码，这使得我们比较难于程序化地清理我们建立的所有东西。

你应该通过一个程序化的侦听器解决这两个问题：

mounted: function () {

var picker = new Pikaday({

field: this.$refs.input,

format: 'YYYY-MM-DD'

})

this.$once('hook:beforeDestroy', function () {

picker.destroy()

})

}

使用了这个策略，我甚至可以让多个输入框元素同时使用不同的 Pikaday，每个新的实例都程序化地在后期清理它自己：

mounted: function () {

this.attachDatepicker('startDateInput')

this.attachDatepicker('endDateInput')

},

methods: {

attachDatepicker: function (refName) {

var picker = new Pikaday({

field: this.$refs[refName],

format: 'YYYY-MM-DD'

})

this.$once('hook:beforeDestroy', function () {

picker.destroy()

})

}

}

查阅这个示例可以了解到完整的代码。注意，即便如此，如果你发现自己不得不在单个组件里做很多建立和清理的工作，最好的方式通常还是创建更多的模块化组件。在这个例子中，我们推荐创建一个可复用的 <input-datepicker> 组件。

想了解更多程序化侦听器的内容，请查阅实例方法 / 事件相关的 API。

注意 Vue 的事件系统不同于浏览器的 EventTarget API。尽管它们工作起来是相似的，但是 $emit、$on, 和 $off 并不是 dispatchEvent、addEventListener 和 removeEventListener 的别名。

## 17.3 循环引用

### 17.3.1 递归组件

组件是可以在它们自己的模板中调用自身的。不过它们只能通过 name 选项来做这件事：

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

当你使用 Vue.component 全局注册一个组件时，这个全局的 ID 会自动设置为该组件的 name 选项。

Vue.component('unique-name-of-my-component', {

// ...

})

稍有不慎，递归组件就可能导致无限循环：

name: 'stack-overflow',

template: '<div><stack-overflow></stack-overflow></div>'

类似上述的组件将会导致“max stack size exceeded”错误，所以请确保递归调用是条件性的 (例如使用一个最终会得到 false 的 v-if)。

### 17.3.2 组件之间的循环引用

假设你需要构建一个文件目录树，像访达或资源管理器那样的。你可能有一个 <tree-folder> 组件，模板是这样的：

<p>

<span>{{ folder.name }}</span>

<tree-folder-contents :children="folder.children"/>

</p>

还有一个 <tree-folder-contents> 组件，模板是这样的：

<ul>

<li v-for="child in children">

<tree-folder v-if="child.children" :folder="child"/>

<span v-else>{{ child.name }}</span>

</li>

</ul>

当你仔细观察的时候，你会发现这些组件在渲染树中互为对方的后代和祖先——一个悖论！当通过 Vue.component 全局注册组件的时候，这个悖论会被自动解开。如果你是这样做的，那么你可以跳过这里。

然而，如果你使用一个模块系统依赖/导入组件，例如通过 webpack 或 Browserify，你会遇到一个错误：

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

为了解释这里发生了什么，我们先把两个组件称为 A 和 B。模块系统发现它需要 A，但是首先 A 依赖 B，但是 B 又依赖 A，但是 A 又依赖 B，如此往复。这变成了一个循环，不知道如何不经过其中一个组件而完全解析出另一个组件。为了解决这个问题，我们需要给模块系统一个点，在那里“A 反正是需要 B 的，但是我们不需要先解析 B。”

在我们的例子中，把 <tree-folder> 组件设为了那个点。我们知道那个产生悖论的子组件是 <tree-folder-contents> 组件，所以我们会等到生命周期钩子 beforeCreate 时去注册它：

beforeCreate: function () {

this.$options.components.TreeFolderContents = require('./tree-folder-contents.vue').default

}

或者，在本地注册组件的时候，你可以使用 webpack 的异步 import：

components: {

TreeFolderContents: () => import('./tree-folder-contents.vue')

}

这样问题就解决了！

## 17.4 模板定义的替代品

### 17.4.1 内联模板

当 inline-template 这个特殊的 attribute 出现在一个子组件上时，这个组件将会使用其里面的内容作为模板，而不是将其作为被分发的内容。这使得模板的撰写工作更加灵活。

<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>

内联模板需要定义在 Vue 所属的 DOM 元素内。

不过，inline-template 会让模板的作用域变得更加难以理解。所以作为最佳实践，请在组件内优先选择 template 选项或 .vue 文件里的一个 <template> 元素来定义模板。

### 17.4.2 X-Template

另一个定义模板的方式是在一个 <script> 元素中，并为其带上 text/x-template 的类型，然后通过一个 id 将模板引用过去。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'

})

x-template 需要定义在 Vue 所属的 DOM 元素外。

这些可以用于模板特别大的 demo 或极小型的应用，但是其它情况下请避免使用，因为这会将模板和该组件的其它定义分离开。

## 17.5 控制更新

感谢 Vue 的响应式系统，它始终知道何时进行更新 (如果你用对了的话)。不过还是有一些边界情况，你想要强制更新，尽管表面上看响应式的数据没有发生改变。也有一些情况是你想阻止不必要的更新。

### 17.5.1 强制更新

如果你发现你自己需要在 Vue 中做一次强制更新，99.9% 的情况，是你在某个地方做错了事。

你可能还没有留意到数组或对象的变更检测注意事项，或者你可能依赖了一个未被 Vue 的响应式系统追踪的状态。

然而，如果你已经做到了上述的事项仍然发现在极少数的情况下需要手动强制更新，那么你可以通过 $forceUpdate 来做这件事。

### 17.5.2 通过 v-once 创建低开销的静态组件

渲染普通的 HTML 元素在 Vue 中是非常快速的，但有的时候你可能有一个组件，这个组件包含了大量静态内容。在这种情况下，你可以在根元素上添加 v-once attribute 以确保这些内容只计算一次然后缓存起来，就像这样：

Vue.component('terms-of-service', {

template: `

<div v-once>

<h1>Terms of Service</h1>

... a lot of static content ...

</div>

`

})

再说一次，试着不要过度使用这个模式。当你需要渲染大量静态内容时，极少数的情况下它会给你带来便利，除非你非常留意渲染变慢了，不然它完全是没有必要的——再加上它在后期会带来很多困惑。例如，设想另一个开发者并不熟悉 v-once 或漏看了它在模板中，他们可能会花很多个小时去找出模板为什么无法正确更新。

# 第18章 进入/离开 & 列表过渡

## 18.1 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。

## 18.2 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进入/离开过渡

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 below version 2.1.8 \*/ {

opacity: 0;

}

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：

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。

如果过渡组件提供了 JavaScript 钩子函数，这些钩子函数将在恰当的时机被调用。

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)

### 18.2.1 过渡的类名

在进入/离开的过渡中，会有 6 个 class 切换。

v-enter：定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。

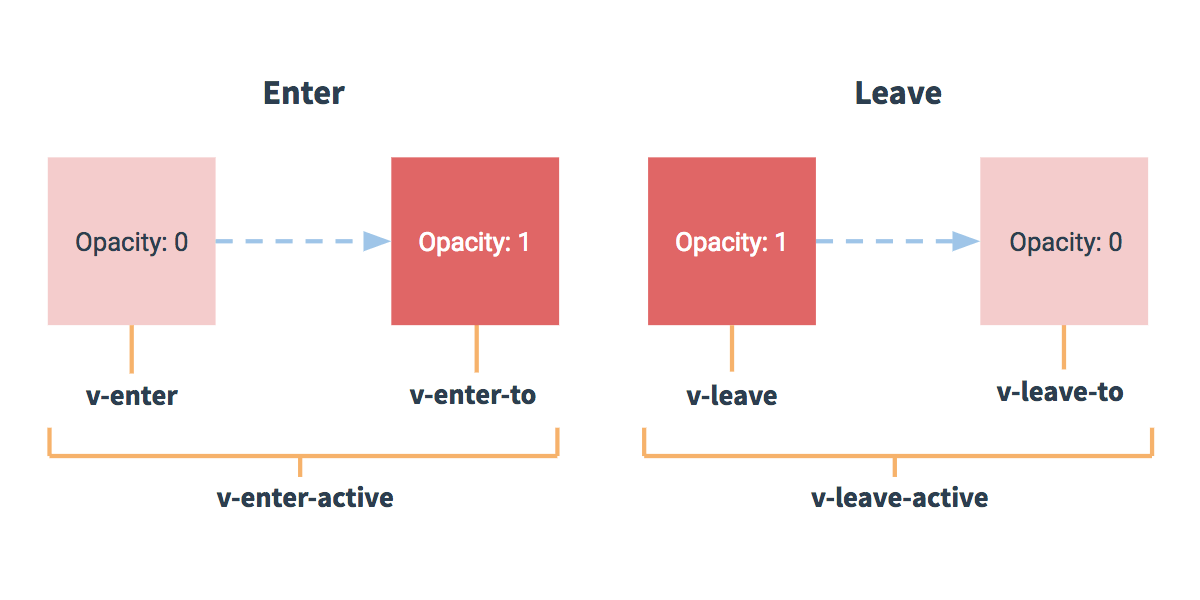
v-enter-active：定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。

v-enter-to：2.1.8 版及以上定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 v-enter 被移除)，在过渡/动画完成之后移除。

v-leave：定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除。

v-leave-active：定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数。

v-leave-to：2.1.8 版及以上定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 v-leave 被删除)，在过渡/动画完成之后移除。



对于这些在过渡中切换的类名来说，如果你使用一个没有名字的 <transition>，则 v- 是这些类名的默认前缀。如果你使用了 <transition name="my-transition">，那么 v-enter 会替换为 my-transition-enter。

v-enter-active 和 v-leave-active 可以控制进入/离开过渡的不同的缓和曲线，在下面章节会有个示例说明。

### 18.2.2 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;

}

### 18.2.3 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">Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris facilisis enim libero, at lacinia diam fermentum id. Pellentesque habitant morbi tristique senectus et netus.</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);

}

}

### 18.2.4 自定义过渡的类名

我们可以通过以下 attribute 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 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

}

})

### 18.2.5 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 attribute 并设置 animation 或 transition 来明确声明你需要 Vue 监听的类型。

### 18.2.6 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系列过渡效果，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 prop 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>

### 18.2.7 JavaScript 钩子

可以在 attribute 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) {

// ...

},

// 当与 CSS 结合使用时

// 回调函数 done 是可选的

enter: function (el, done) {

// ...

done()

},

afterEnter: function (el) {

// ...

},

enterCancelled: function (el) {

// ...

},

// --------

// 离开时

// --------

beforeLeave: function (el) {

// ...

},

// 当与 CSS 结合使用时

// 回调函数 done 是可选的

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 中必须使用 done 进行回调。否则，它们将被同步调用，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 和 jQuery.animate 的工作方式类似，也是用来实现 JavaScript 动画的一个很棒的选择

-->

<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 })

}

}

})

## 18.3 初始渲染的过渡

可以通过 appear attribute 设置节点在初始渲染的过渡

<transition appear>

<!-- ... -->

</transition>

这里默认和进入/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>

在上面的例子中，无论是 appear attribute 还是 v-on:appear 钩子都会生成初始渲染过渡。

## 18.4 多个元素的过渡

我们之后讨论多个组件的过渡，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 attribute 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，也可以通过给同一个元素的 key attribute 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 property 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'

}

}

}

### 18.4.1 过渡模式

这里还有一个问题，试着点击下面的按钮：

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：

然后，我们加上 translate 让它们运动像滑动过渡：

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>

只用添加一个简单的 attribute，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：

很酷吧？

## 18.5 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 attribute。相反，我们只需要使用动态组件：

<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;

}

## 18.6 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 attribute 更换为其他元素。

过渡模式不可用，因为我们不再相互切换特有的元素。

内部元素总是需要提供唯一的 key attribute 值。

CSS 过渡的类将会应用在内部的元素中，而不是这个组/容器本身。

### 18.6.1 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。

### 18.6.2 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 class，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 attribute 来自定义前缀，也可以通过 move-class attribute 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;

}

这个看起来很神奇，内部的实现，Vue 使用了一个叫 FLIP 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;

}

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：

### 18.6.3列表的交错过渡

通过 data attribute 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)

}

}

})

## 18.7 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) {

// ...

}

}

})

函数式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)

}

})

## 18.8 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 attribute 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。

所有过渡 attribute 都可以动态绑定，但我们不仅仅只有 attribute 可以利用，还可以通过事件钩子获取上下文中的所有数据，因为事件钩子都是方法。这意味着，根据组件的状态不同，你的 JavaScript 过渡会有不同的表现。

<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

}

}

)

}

}

})

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。

# 第19章 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 property

这些数据要么本身就以数值形式存储，要么可以转换为数值。有了这些数值后，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。

## 19.1 状态动画与侦听器

通过侦听器我们能监听到任何数值 property 的数值更新。可能听起来很抽象，所以让我们先来看看使用 GreenSock 一个例子：

<script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.2.4/gsap.min.js"></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,

tweenedNumber: 0

},

computed: {

animatedNumber: function() {

return this.tweenedNumber.toFixed(0);

}

},

watch: {

number: function(newValue) {

gsap.to(this.$data, { duration: 0.5, tweenedNumber: newValue });

}

}

})

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 Tween.js 和 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;

}

## 19.2 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。

## 19.3 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

}

}

})

20

40

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。总之，对于完成各种过渡动效几乎没有阻碍。

## 19.4 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、思考或警戒的样例。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：

# 第20章 混入

## 20.1 基础

混入 (mixin) 提供了一种非常灵活的方式，来分发 Vue 组件中的可复用功能。一个混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被“混合”进入该组件本身的选项。

例子：

// 定义一个混入对象

var myMixin = {

created: function () {

this.hello()

},

methods: {

hello: function () {

console.log('hello from mixin!')

}

}

}

// 定义一个使用混入对象的组件

var Component = Vue.extend({

mixins: [myMixin]

})

var component = new Component() // => "hello from mixin!"

## 20.2 选项合并

当组件和混入对象含有同名选项时，这些选项将以恰当的方式进行“合并”。

比如，数据对象在内部会进行递归合并，并在发生冲突时以组件数据优先。

var mixin = {

data: function () {

return {

message: 'hello',

foo: 'abc'

}

}

}

new Vue({

mixins: [mixin],

data: function () {

return {

message: 'goodbye',

bar: 'def'

}

},

created: function () {

console.log(this.$data)

// => { message: "goodbye", foo: "abc", bar: "def" }

}

})

同名钩子函数将合并为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子之前调用。

var mixin = {

created: function () {

console.log('混入对象的钩子被调用')

}

}

new Vue({

mixins: [mixin],

created: function () {

console.log('组件钩子被调用')

}

})

// => "混入对象的钩子被调用"

// => "组件钩子被调用"

值为对象的选项，例如 methods、components 和 directives，将被合并为同一个对象。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。

## 20.3 全局混入

混入也可以进行全局注册。使用时格外小心！一旦使用全局混入，它将影响每一个之后创建的 Vue 实例。使用恰当时，这可以用来为自定义选项注入处理逻辑。

// 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!"

请谨慎使用全局混入，因为它会影响每个单独创建的 Vue 实例 (包括第三方组件)。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。推荐将其作为插件发布，以避免重复应用混入。

## 20.4 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) {

// 返回合并后的值

}

对于多数值为对象的选项，可以使用与 methods 相同的合并策略：

var strategies = Vue.config.optionMergeStrategies

strategies.myOption = strategies.methods

可以在 Vuex 1.x 的混入策略里找到一个更高级的例子：

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)

}

}

# 第21章 自定义指令

## 21.1. 简介

除了核心功能默认内置的指令 (v-model 和 v-show)，Vue 也允许注册自定义指令。注意，在 Vue2.0 中，代码复用和抽象的主要形式是组件。然而，有的情况下，你仍然需要对普通 DOM 元素进行底层操作，这时候就会用到自定义指令。举个聚焦输入框的例子，如下：

当页面加载时，该元素将获得焦点 (注意：autofocus 在移动版 Safari 上不工作)。事实上，只要你在打开这个页面后还没点击过任何内容，这个输入框就应当还是处于聚焦状态。现在让我们用指令来实现这个功能：

// 注册一个全局自定义指令 `v-focus`

Vue.directive('focus', {

// 当被绑定的元素插入到 DOM 中时……

inserted: function (el) {

// 聚焦元素

el.focus()

}

})

如果想注册局部指令，组件中也接受一个 directives 的选项：

directives: {

focus: {

// 指令的定义

inserted: function (el) {

el.focus()

}

}

}

然后你可以在模板中任何元素上使用新的 v-focus property，如下：

<input v-focus>

## 21.2 钩子函数

**一个指令定义对象可以提供如下几个钩子函数 (均为可选)：**

**bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。**

**inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)。**

**update：所在组件的 VNode 更新时调用，但是可能发生在其子 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)。**

**我们会在稍后讨论渲染函数时介绍更多 VNodes 的细节。**

**componentUpdated：指令所在组件的 VNode 及其子 VNode 全部更新后调用。**

**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 (即 el、binding、vnode 和 oldVnode)。**

## 21.3钩子函数参数

指令钩子函数会被传入以下参数：

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 property：

name：指令名，不包括 v- 前缀。

value：指令的绑定值，例如：v-my-directive="1 + 1" 中，绑定值为 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：字符串形式的指令表达式。例如 v-my-directive="1 + 1" 中，表达式为 "1 + 1"。

arg：传给指令的参数，可选。例如 v-my-directive:foo 中，参数为 "foo"。

modifiers：一个包含修饰符的对象。例如：v-my-directive.foo.bar 中，修饰符对象为 { foo: true, bar: true }。

vnode：Vue 编译生成的虚拟节点。移步 VNode API 来了解更多详情。

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，切勿进行修改。如果需要在钩子之间共享数据，建议通过元素的 dataset 来进行。

这是一个使用了这些 property 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!'

}

})

### 21.3.1 动态指令参数

指令的参数可以是动态的。例如，在 v-mydirective:[argument]="value" 中，argument 参数可以根据组件实例数据进行更新！这使得自定义指令可以在应用中被灵活使用。

例如你想要创建一个自定义指令，用来通过固定布局将元素固定在页面上。我们可以像这样创建一个通过指令值来更新竖直位置像素值的自定义指令：

<div id="baseexample">

<p>Scroll down the page</p>

<p v-pin="200">Stick me 200px from the top of the page</p>

</div>

Vue.directive('pin', {

bind: function (el, binding, vnode) {

el.style.position = 'fixed'

el.style.top = binding.value + 'px'

}

})

new Vue({

el: '#baseexample'

})

这会把该元素固定在距离页面顶部 200 像素的位置。但如果场景是我们需要把元素固定在左侧而不是顶部又该怎么办呢？这时使用动态参数就可以非常方便地根据每个组件实例来进行更新。

<div id="dynamicexample">

<h3>Scroll down inside this section ↓</h3>

<p v-pin:[direction]="200">I am pinned onto the page at 200px to the left.</p>

</div>

Vue.directive('pin', {

bind: function (el, binding, vnode) {

el.style.position = 'fixed'

var s = (binding.arg == 'left' ? 'left' : 'top')

el.style[s] = binding.value + 'px'

}

})

new Vue({

el: '#dynamicexample',

data: function () {

return {

direction: 'left'

}

}

})

这样这个自定义指令现在的灵活性就足以支持一些不同的用例了。

## 21.4 函数简写

在很多时候，你可能想在 bind 和 update 时触发相同行为，而不关心其它的钩子。比如这样写：

Vue.directive('color-swatch', function (el, binding) {

el.style.backgroundColor = binding.value

})

## 21.5 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!"

})

# 第22章 渲染函数 & JSX

## 22.1 基础

Vue 推荐在绝大多数情况下使用模板来创建你的 HTML。然而在一些场景中，你真的需要 JavaScript 的完全编程的能力。这时你可以用渲染函数，它比模板更接近编译器。

让我们深入一个简单的例子，这个例子里 render 函数很实用。假设我们要生成一些带锚点的标题：

<h1>

<a name="hello-world" href="#hello-world">

Hello world!

</a>

</h1>

对于上面的 HTML，你决定这样定义组件接口：

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

当开始写一个只能通过 level prop 动态生成标题 (heading) 的组件时，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

}

}

})

这里用模板并不是最好的选择：不但代码冗长，而且在每一个级别的标题中重复书写了 <slot></slot>，在要插入锚点元素时还要再次重复。

虽然模板在大多数组件中都非常好用，但是显然在这里它就不合适了。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, // 标签名称

this.$slots.default // 子节点数组

)

},

props: {

level: {

type: Number,

required: true

}

}

})

看起来简单多了！这样代码精简很多，但是需要非常熟悉 Vue 的实例 property。在这个例子中，你需要知道，向组件中传递不带 v-slot 指令的子节点时，比如 anchored-heading 中的 Hello world!，这些子节点被存储在组件实例中的 $slots.default 中。如果你还不了解，在深入渲染函数之前推荐阅读实例 property API。

## 22.2 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>

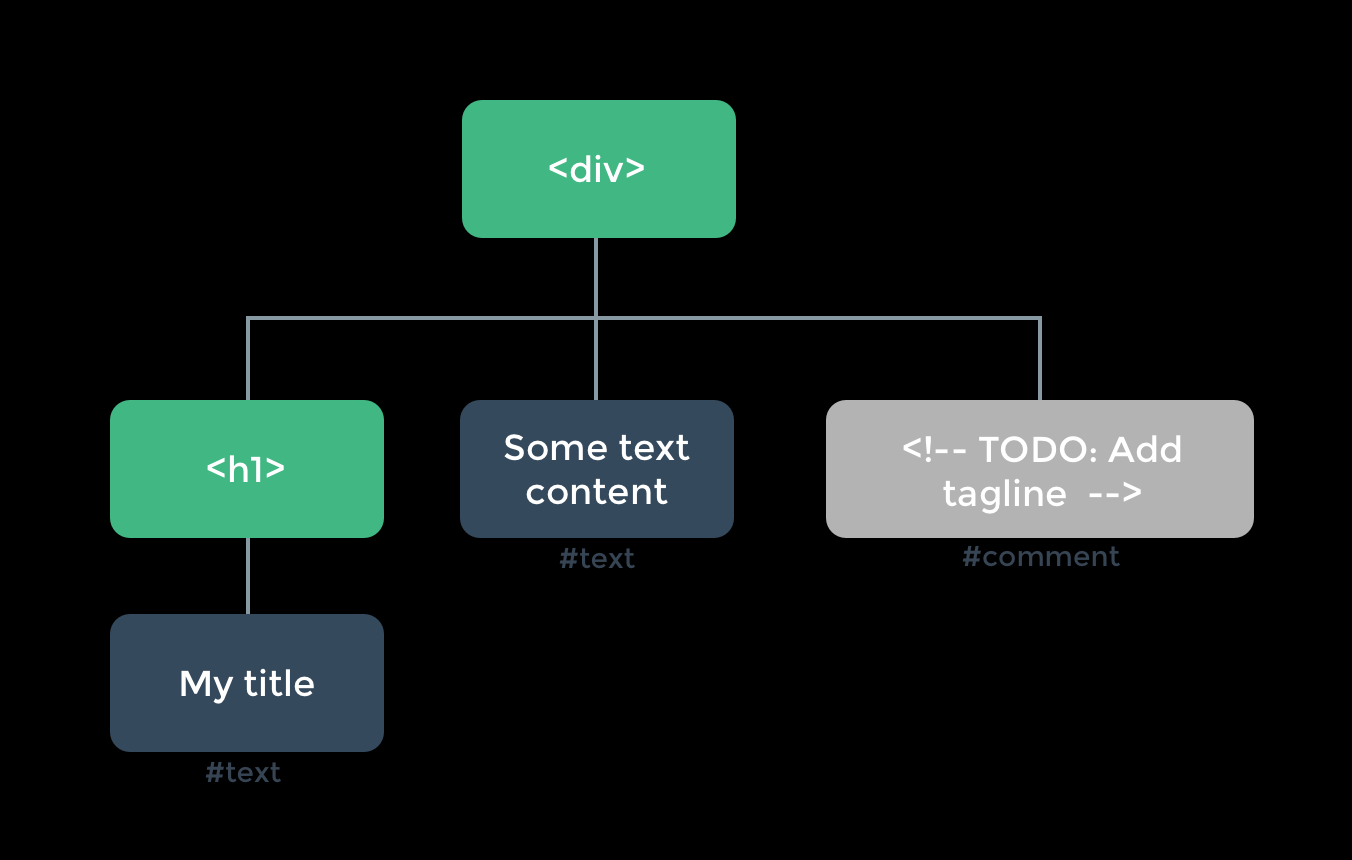
Some text content

<!-- TODO: Add tagline -->

</div>

当浏览器读到这些代码时，它会建立一个“DOM 节点”树来保持追踪所有内容，如同你会画一张家谱树来追踪家庭成员的发展一样。

上述 HTML 对应的 DOM 节点树如下图所示：



每个元素都是一个节点。每段文字也是一个节点。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)。

高效地更新所有这些节点会是比较困难的，不过所幸你不必手动完成这个工作。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。

### 22.2.1 虚拟 DOM

Vue 通过建立一个虚拟 DOM 来追踪自己要如何改变真实 DOM。请仔细看这行代码：

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，包括及其子节点的描述信息。我们把这样的节点描述为“虚拟节点 (virtual node)”，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。

## 22.3 createElement 参数

接下来你需要熟悉的是如何在 createElement 函数中使用模板中的那些功能。这里是 createElement 接受的参数：

// @returns {VNode}

createElement(

// {String | Object | Function}

// 一个 HTML 标签名、组件选项对象，或者

// resolve 了上述任何一种的一个 async 函数。必填项。

'div',

// {Object}

// 一个与模板中 attribute 对应的数据对象。可选。

{

// (详情见下一节)

},

// {String | Array}

// 子级虚拟节点 (VNodes)，由 `createElement()` 构建而成，

// 也可以使用字符串来生成“文本虚拟节点”。可选。

[

'先写一些文字',

createElement('h1', '一则头条'),

createElement(MyComponent, {

props: {

someProp: 'foobar'

}

})

]

)

### 22.3.1 深入数据对象

有一点要注意：正如 v-bind:class 和 v-bind:style 在模板语法中会被特别对待一样，它们在 VNode 数据对象中也有对应的顶层字段。该对象也允许你绑定普通的 HTML attribute，也允许绑定如 innerHTML 这样的 DOM property (这会覆盖 v-html 指令)。

{

// 与 `v-bind:class` 的 API 相同，

// 接受一个字符串、对象或字符串和对象组成的数组

'class': {

foo: true,

bar: false

},

// 与 `v-bind:style` 的 API 相同，

// 接受一个字符串、对象，或对象组成的数组

style: {

color: 'red',

fontSize: '14px'

},

// 普通的 HTML attribute

attrs: {

id: 'foo'

},

// 组件 prop

props: {

myProp: 'bar'

},

// DOM property

domProps: {

innerHTML: 'baz'

},

// 事件监听器在 `on` 内，

// 但不再支持如 `v-on:keyup.enter` 这样的修饰器。

// 需要在处理函数中手动检查 keyCode。

on: {

click: this.clickHandler

},

// 仅用于组件，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，你无法对 `binding` 中的 `oldValue`

// 赋值，因为 Vue 已经自动为你进行了同步。

directives: [

{

name: 'my-custom-directive',

value: '2',

expression: '1 + 1',

arg: 'foo',

modifiers: {

bar: true

}

}

],

// 作用域插槽的格式为

// { name: props => VNode | Array<VNode> }

scopedSlots: {

default: props => createElement('span', props.text)

},

// 如果组件是其它组件的子组件，需为插槽指定名称

slot: 'name-of-slot',

// 其它特殊顶层 property

key: 'myKey',

ref: 'myRef',

// 如果你在渲染函数中给多个元素都应用了相同的 ref 名，

// 那么 `$refs.myRef` 会变成一个数组。

refInFor: true

}

### 22.3.2 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) {

// 创建 kebab-case 风格的 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

}

}

})

### 22.3.3 约束

VNode 必须唯一

组件树中的所有 VNode 必须是唯一的。这意味着，下面的渲染函数是不合法的：

render: function (createElement) {

var myParagraphVNode = createElement('p', 'hi')

return createElement('div', [

// 错误 - 重复的 VNode

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，下面这渲染函数用完全合法的方式渲染了 20 个相同的段落：

render: function (createElement) {

return createElement('div',

Array.apply(null, { length: 20 }).map(function () {

return createElement('p', 'hi')

})

)

}

## 22.4 使用 JavaScript 代替模板功能

### 22.4.1 v-if 和 v-for

只要在原生的 JavaScript 中可以轻松完成的操作，Vue 的渲染函数就不会提供专有的替代方法。比如，在模板中使用的 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>

这些都可以在渲染函数中用 JavaScript 的 if/else 和 map 来重写：

props: ['items'],

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.')

}

}

### 22.4.2 v-model

渲染函数中没有与 v-model 的直接对应——你必须自己实现相应的逻辑：

props: ['value'],

render: function (createElement) {

var self = this

return createElement('input', {

domProps: {

value: self.value

},

on: {

input: function (event) {

self.$emit('input', event.target.value)

}

}

})

}

这就是深入底层的代价，但与 v-model 相比，这可以让你更好地控制交互细节。

### 22.4.3 事件 & 按键修饰符

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

修饰符 前缀

.passive &

.capture !

.once ~

.capture.once 或

.once.capture ~!

例如：

on: {

'!click': this.doThisInCapturingMode,

'~keyup': this.doThisOnce,

'~!mouseover': this.doThisOnceInCapturingMode

}

对于所有其它的修饰符，私有前缀都不是必须的，因为你可以在事件处理函数中使用事件方法：

修饰符 处理函数中的等价操作

.stop event.stopPropagation()

.prevent event.preventDefault()

.self if (event.target !== event.currentTarget) return

按键：

.enter, .13 if (event.keyCode !== 13) return (对于别的按键修饰符来说，可将 13 改为另一个按键码)

修饰键：

.ctrl, .alt, .shift, .meta if (!event.ctrlKey) return (将 ctrlKey 分别修改为 altKey、shiftKey 或者 metaKey)

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()

// ...

}

}

### 22.4.4 插槽

你可以通过 this.$slots 访问静态插槽的内容，每个插槽都是一个 VNode 数组：

render: function (createElement) {

// `<div><slot></slot></div>`

return createElement('div', this.$slots.default)

}

也可以通过 this.$scopedSlots 访问作用域插槽，每个作用域插槽都是一个返回若干 VNode 的函数：

props: ['message'],

render: function (createElement) {

// `<div><slot :text="message"></slot></div>`

return createElement('div', [

this.$scopedSlots.default({

text: this.message

})

])

}

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 数据对象中的 scopedSlots 字段：

render: function (createElement) {

// `<div><child v-slot="props"><span>{{ props.text }}</span></child></div>`

return createElement('div', [

createElement('child', {

// 在数据对象中传递 `scopedSlots`

// 格式为 { name: props => VNode | Array<VNode> }

scopedSlots: {

default: function (props) {

return createElement('span', props.text)

}

}

})

])

}

## 22.5 JSX

如果你写了很多 render 函数，可能会觉得下面这样的代码写起来很痛苦：

createElement(

'anchored-heading', {

props: {

level: 1

}

}, [

createElement('span', 'Hello'),

' world!'

]

)

特别是对应的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，用于在 Vue 中使用 JSX 语法，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: function (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。从 Vue 的 Babel 插件的 3.4.0 版本开始，我们会在以 ES2015 语法声明的含有 JSX 的任何方法和 getter 中 (不是函数或箭头函数中) 自动注入 const h = this.$createElement，这样你就可以去掉 (h) 参数了。对于更早版本的插件，如果 h 在当前作用域中不可用，应用会抛错。

要了解更多关于 JSX 如何映射到 JavaScript，请阅读使用文档。

## 22.6 函数式组件

之前创建的锚点标题组件是比较简单，没有管理任何状态，也没有监听任何传递给它的状态，也没有生命周期方法。实际上，它只是一个接受一些 prop 的函数。在这样的场景下，我们可以将组件标记为 functional，这意味它无状态 (没有响应式数据)，也没有实例 (没有 this 上下文)。一个函数式组件就像这样：

Vue.component('my-component', {

functional: true,

// Props 是可选的

props: {

// ...

},

// 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) {

// ...

}

})

注意：在 2.3.0 之前的版本中，如果一个函数式组件想要接收 prop，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 attribute 都会被自动隐式解析为 prop。

当使用函数式组件时，该引用将会是 HTMLElement，因为他们是无状态的也是无实例的。

在 2.5.0 及以上版本中，如果你使用了单文件组件，那么基于模板的函数式组件可以这样声明：

<template functional>

</template>

组件需要的一切都是通过 context 参数传递，它是一个包括如下字段的对象：

props：提供所有 prop 的对象

children：VNode 子节点的数组

slots：一个函数，返回了包含所有插槽的对象

scopedSlots：(2.6.0+) 一个暴露传入的作用域插槽的对象。也以函数形式暴露普通插槽。

data：传递给组件的整个数据对象，作为 createElement 的第二个参数传入组件

parent：对父组件的引用

listeners：(2.3.0+) 一个包含了所有父组件为当前组件注册的事件监听器的对象。这是 data.on 的一个别名。

injections：(2.3.0+) 如果使用了 inject 选项，则该对象包含了应当被注入的 property。

在添加 functional: true 之后，需要更新我们的锚点标题组件的渲染函数，为其增加 context 参数，并将 this.$slots.default 更新为 context.children，然后将 this.level 更新为 context.props.level。

因为函数式组件只是函数，所以渲染开销也低很多。

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来代为渲染；

在将 children、props、data 传递给子组件之前操作它们。

下面是一个 smart-list 组件的例子，它能根据传入 prop 的值来代为渲染更具体的组件：

var EmptyList = { /\* ... \*/ }

var TableList = { /\* ... \*/ }

var OrderedList = { /\* ... \*/ }

var UnorderedList = { /\* ... \*/ }

Vue.component('smart-list', {

functional: true,

props: {

items: {

type: Array,

required: true

},

isOrdered: Boolean

},

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

)

}

})

### 22.6.1 向子元素或子组件传递 attribute 和事件

在普通组件中，没有被定义为 prop 的 attribute 会自动添加到组件的根元素上，将已有的同名 attribute 进行替换或与其进行智能合并。

然而函数式组件要求你显式定义该行为：

Vue.component('my-functional-button', {

functional: true,

render: function (createElement, context) {

// 完全透传任何 attribute、事件监听器、子节点等。

return createElement('button', context.data, context.children)

}

})

通过向 createElement 传入 context.data 作为第二个参数，我们就把 my-functional-button 上面所有的 attribute 和事件监听器都传递下去了。事实上这是非常透明的，以至于那些事件甚至并不要求 .native 修饰符。

如果你使用基于模板的函数式组件，那么你还需要手动添加 attribute 和监听器。因为我们可以访问到其独立的上下文内容，所以我们可以使用 data.attrs 传递任何 HTML attribute，也可以使用 listeners (即 data.on 的别名) 传递任何事件监听器。

<template functional>

<button

class="btn btn-primary"

v-bind="data.attrs"

v-on="listeners"

>

<slot/>

</button>

</template>

### 22.6.2 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——但如果是如下的带有子节点的函数式组件呢？

<my-functional-component>

<p v-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感知某个插槽机制，还是简单地通过传递 children，移交给其它组件去处理。

## 22.7 模板编译

你可能会有兴趣知道，Vue 的模板实际上被编译成了渲染函数。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具体是怎样被编译的，可能会发现会非常有意思。下面是一个使用 Vue.compile 来实时编译模板字符串的简单示例：

<div>

<header>

<h1>I'm a template!</h1>

</header>

<p v-if="message">{{ message }}</p>

<p v-else>No message.</p>

</div>

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!")])])}

}

# 第23章 插件

插件通常用来为 Vue 添加全局功能。插件的功能范围没有严格的限制——一般有下面几种：

添加全局方法或者 property。如：vue-custom-element

添加全局资源：指令/过滤器/过渡等。如 vue-touch

通过全局混入来添加一些组件选项。如 vue-router

添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现。

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

## 23.1 使用插件

通过全局方法 Vue.use() 使用插件。它需要在你调用 new Vue() 启动应用之前完成：

// 调用 `MyPlugin.install(Vue)`

Vue.use(MyPlugin)

new Vue({

// ...组件选项

})

也可以传入一个可选的选项对象：

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

Vue.use 会自动阻止多次注册相同插件，届时即使多次调用也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()。然而在像 CommonJS 这样的模块环境中，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 集合了大量由社区贡献的插件和库。

## 23.2 开发插件

Vue.js 的插件应该暴露一个 install 方法。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 property

Vue.myGlobalMethod = function () {

// 逻辑...

}

// 2. 添加全局资源

Vue.directive('my-directive', {

bind (el, binding, vnode, oldVnode) {

// 逻辑...

}

...

})

// 3. 注入组件选项

Vue.mixin({

created: function () {

// 逻辑...

}

...

})

// 4. 添加实例方法

Vue.prototype.$myMethod = function (methodOptions) {

// 逻辑...

}

}

# 第24章 过滤器

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：双花括号插值和 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”符号指示：

<!-- 在双花括号中 -->

{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->

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

你可以在一个组件的选项中定义本地的过滤器：

filters: {

capitalize: function (value) {

if (!value) return ''

value = value.toString()

return value.charAt(0).toUpperCase() + value.slice(1)

}

}

或者在创建 Vue 实例之前全局定义过滤器：

Vue.filter('capitalize', function (value) {

if (!value) return ''

value = value.toString()

return value.charAt(0).toUpperCase() + value.slice(1)

})

new Vue({

// ...

})

当全局过滤器和局部过滤器重名时，会采用局部过滤器。

下面这个例子用到了 capitalize 过滤器：

john

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。在上述例子中，capitalize 过滤器函数将会收到 message 的值作为第一个参数。

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 的值作为第三个参数。

# 第25章 单文件组件

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：

1.全局定义 (Global definitions) 强制要求每个 component 中的命名不得重复

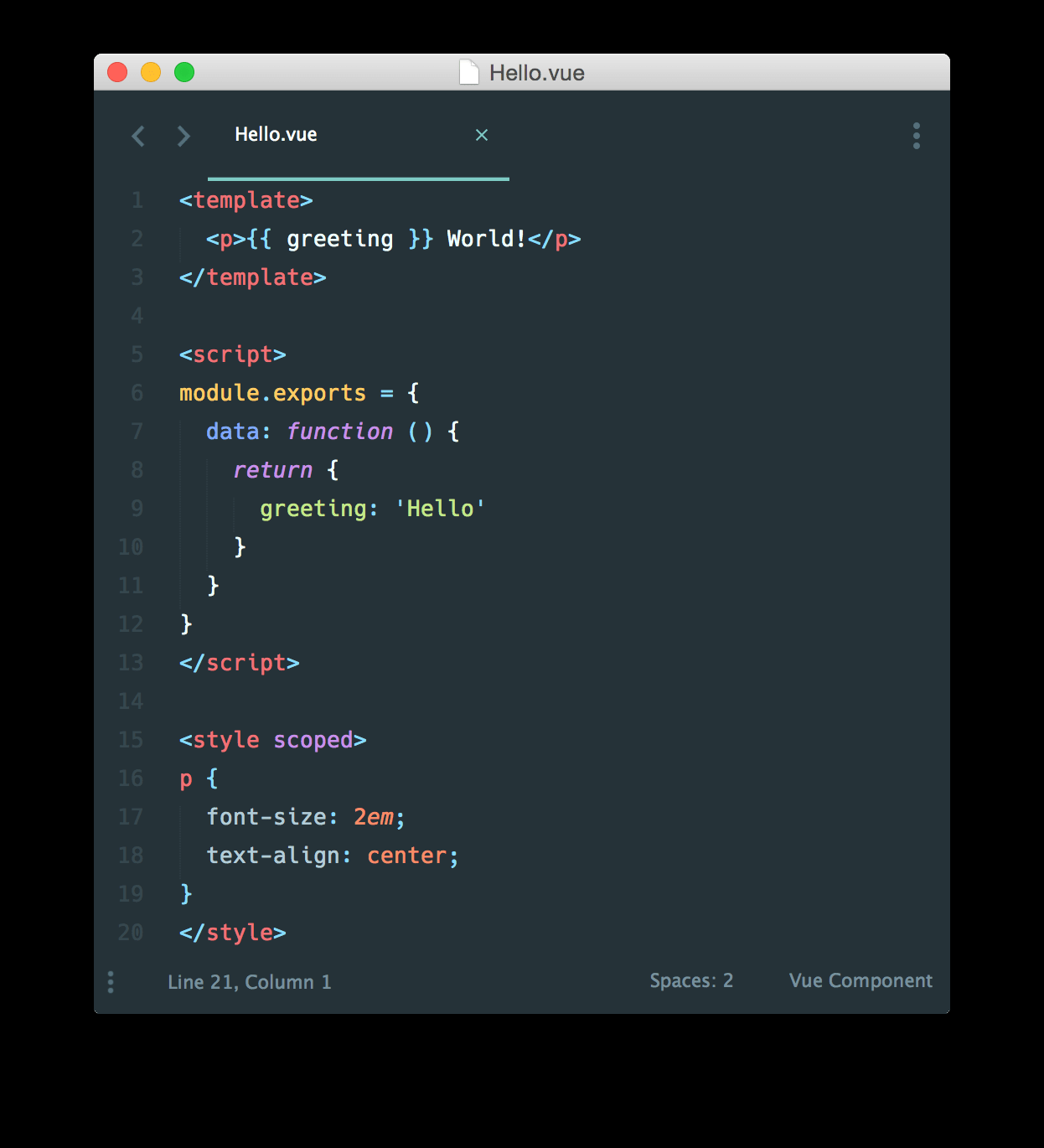
2.字符串模板 (String templates) 缺乏语法高亮，在 HTML 有多行的时候，需要用到丑陋的 \

3.不支持 CSS (No CSS support) 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏

4.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：



单文件组件的示例 (点击查看文本版的代码

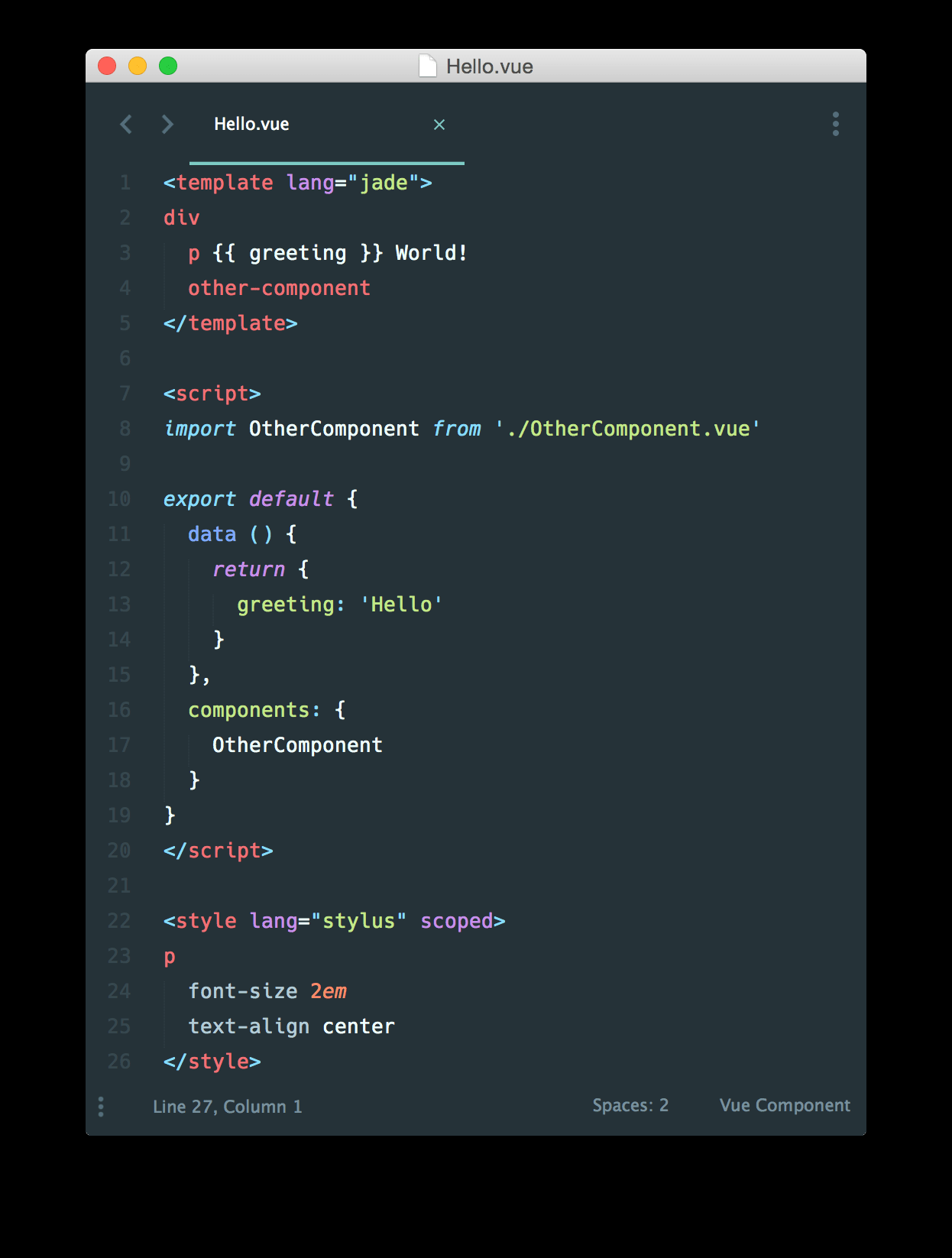
现在我们获得：

完整语法高亮

CommonJS 模块

组件作用域的 CSS

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。



带预处理器的单文件组件的示例 (点击查看文本版的代码)

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，它也能为 CSS Modules 提供头等支持。

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**

***例子沙箱***

如果你希望深入了解并开始使用单文件组件，请来 CodeSandbox 看看这个简单的 todo 应用。

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 中关于如何从注册地 (registry) 获取包的章节。

Modern JavaScript with ES2015/16：阅读 Babel 的 Learn ES2015 guide。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。

在你花一天时间了解这些资源之后，我们建议你参考 Vue CLI 3。只要遵循指示，你就能很快地运行一个带有 .vue 组件、ES2015、webpack 和热重载 (hot-reloading) 的 Vue 项目！

**针对高级用户**

CLI 会为你搞定大多数工具的配置问题，同时也支持细粒度自定义配置项。

有时你会想从零搭建你自己的构建工具，这时你需要通过 Vue Loader 手动配置 webpack。关于学习更多 webpack 的内容，请查阅其官方文档和 Webpack Academy。

# 第26章 测试

**介绍**

当构建可靠的应用时，测试在个人或团队构建新特性、重构代码、修复 bug 等工作中扮演了关键的角色。尽管测试的流派有很多，它们在 web 应用这个领域里主要有三大类：

**单元测试**

**组件测试**

端到端 (E2E，end-to-end) 测试

本章节致力于引导大家了解测试的生态系统并为 Vue 应用或组件库选择适合的工具。

**单元测试**

**介绍**

单元测试允许你将独立单元的代码进行隔离测试，其目的是为开发者提供对代码的信心。通过编写细致且有意义的测试，你能够有信心在构建新特性或重构已有代码的同时，保持应用的功能和稳定。

为一个 Vue 应用做单元测试并没有和为其它类型的应用做测试有什么明显的区别。

**选择框架**

因为单元测试的建议通常是框架无关的，所以下面只是当你在评估应用的单元测试工具时需要的一些基本指引。

**一流的错误报告**

当测试失败时，提供有用的错误信息对于单元测试框架来说至关重要。这是断言库应尽的职责。一个具有高质量错误信息的断言能够最小化调试问题所需的时间。除了简单地告诉你什么测试失败了，断言库还应额外提供上下文以及测试失败的原因，例如预期结果 vs. 实际得到的结果。

一些诸如 Jest 这样的单元测试框架会包含断言库。另一些诸如 Mocha 需要你单独安装断言库 (通常会用 Chai)。

**活跃的社区和团队**

因为主流的单元测试框架都是开源的，所以对于一些旨在长期维护其测试且确保项目本身保持活跃的团队来说，拥有一个活跃的社区是至关重要的。额外的好处是，在任何时候遇到问题时，一个活跃的社区会为你提供更多的支持。

**框架**

尽管生态系统里有很多工具，这里我们列出一些在 Vue 生态系统中常用的单元测试工具。

**Jest**

Jest 是一个专注于简易性的 JavaScript 测试框架。一个其独特的功能是可以为测试生成快照 (snapshot)，以提供另一种验证应用单元的方法。

**资料：**

Jest 官网

Vue CLI 官方插件 - Jest

Mocha

Mocha 是一个专注于灵活性的 JavaScript 测试框架。因为其灵活性，它允许你选择不同的库来满足诸如侦听 (如 Sinon) 和断言 (如 Chai) 等其它常见的功能。另一个 Mocha 独特的功能是它不止可以在 Node.js 里运行测试，还可以在浏览器里运行测试。

**资料：**

Mocha 官网

Vue CLI 官方插件 - Mocha

**组件测试**

**介绍**

测试大多数 Vue 组件时都必须将它们挂载到 DOM (虚拟或真实) 上，才能完全断言它们正在工作。这是另一个与框架无关的概念。因此组件测试框架的诞生，是为了让用户能够以可靠的方式完成这项工作，同时还提供了 Vue 特有的诸如对 Vuex、Vue Router 和其他 Vue 插件的集成的便利性。

**选择框架**

以下章节提供了在评估最适合你的应用的组件测试框架时需要记住的事项。

与 Vue 生态系统的最佳兼容性

毋容置疑，最重要的标准之一就是组件测试库应该尽可能与 Vue 生态系统兼容。虽然这看起来很全面，但需要记住的一些关键集成领域包括单文件组件 (SFC)、Vuex、Vue Router 以及应用所依赖的任何其他特定于 Vue 的插件。

**一流的错误报告**

当测试失败时，提供有用的错误日志以最小化调试问题所需的时间对于组件测试框架来说至关重要。除了简单地告诉你什么测试失败了，他们还应额外提供上下文以及测试失败的原因，例如预期结果 vs. 实际得到的结果。

**推荐**

Vue Testing Library (@testing-library/vue)

Vue Testing Library 是一组专注于测试组件而不依赖实现细节的工具。由于在设计时就充分考虑了可访问性，它采用的方案也使重构变得轻而易举。

它的指导原则是，与软件使用方式相似的测试越多，它们提供的可信度就越高。

**资料：**

Vue Testing Library 官网

Vue Test Utils

Vue Test Utils 是官方的偏底层的组件测试库，它是为用户提供对 Vue 特定 API 的访问而编写的。如果你对测试 Vue 应用不熟悉，我们建议你使用 Vue Testing Library，它是 Vue Test Utils 的抽象。

**资源：**

Vue Test Utils 官方文档

Vue 测试指南 by Lachlan Miller

**端到端 (E2E) 测试**

**介绍**

虽然单元测试为开发者提供了一定程度的信心，但是单元测试和组件测试在部署到生产环境时提供应用整体覆盖的能力是有限的。因此，端到端测试可以说从应用最重要的方面进行测试覆盖：当用户实际使用应用时会发生什么。

换句话说，端到端测试验证应用中的所有层。这不仅包括你的前端代码，还包括所有相关的后端服务和基础设施，它们更能代表你的用户所处的环境。通过测试用户操作如何影响应用，端到端测试通常是提高应用是否正常运行的信心的关键。

**选择框架**

虽然 web 上的端到端测试因不可信赖 (片面的) 测试和减慢开发过程而得到负面的声誉，但现代端到端工具在创建更可靠的、交互的和实用的测试方面取得了长足进步。在选择端到端测试框架时，以下章节在你为应用选择测试框架时提供了一些指导。

**跨浏览器测试**

端到端测试的一个主要优点是它能够跨多个浏览器测试应用。尽管 100% 的跨浏览器覆盖看上去很诱人，但需要注意的是，因为持续运行这些跨浏览器测试需要额外的时间和机器消耗，它会降低团队的资源回报。因此，在选择应用需要的跨浏览器测试数量时，必须注意这种权衡。

针对 E2E 浏览器特定问题的一个最新进展是，针对不常用的浏览器 (如：< IE11、旧版 Safari 等) 使用应用监视和错误报告工具 (如：Sentry、LogRocket 等)。

**更快的反馈路径**

端到端测试和开发的主要问题之一是运行整个套件需要很长时间。通常，这只在持续集成和部署 (CI/CD) 管道中完成。现代的端到端测试框架通过添加类似并行化的特性来帮助解决这个问题，这使得 CI/CD 管道的运行速度通常比以前快。此外，在本地开发时，有选择地为正在处理的页面运行单个测试的能力，同时还提供测试的热重载，将有助于提高开发者的工作流程和工作效率。

**一流的调试经验**

虽然开发者传统上依赖于在终端窗口中扫描日志来帮助确定测试中出了什么问题，但现代端到端测试框架允许开发者利用他们已经熟悉的工具，例如浏览器开发工具。

**推荐**

虽然生态系统中有许多工具，但以下是一些 Vue.js 生态系统中常用的端到端测试框架。

Cypress.io

Cypress.io 是一个测试框架，旨在通过使开发者能够可靠地测试他们的应用，同时提供一流的开发者体验，来提高开发者的生产率。

**资料：**

Cypress 官网

Vue CLI 官方插件 - Cypress

Cypress Testing Library

Nightwatch.js

Nightwatch.js 是一个端到端测试框架，可用于测试 web 应用和网站，以及 Node.js 单元测试和集成测试。

**资料：**

Nightwatch 官网

Vue CLI 官方插件 - Nightwatch

Puppeteer

Puppeteer 是一个 Node.js 库，它提供高阶 API 来控制浏览器，并可以与其他测试运行程序 (例如 Jest) 配对来测试应用。

**资料：**

Puppeteer 官网

TestCafe

TestCafe 是一个基于端到端的 Node.js 框架，旨在提供简单的设置，以便开发者能够专注于创建易于编写和可靠的测试。

**资料：**

TestCafe 官网

# 第27章TypeScript 支持

Vue CLI 提供了内建的 TypeScript 工具支持。

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 的官方类型声明，还为 Vue Router 和 Vuex 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，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": {

// 与 Vue 的浏览器支持保持一致

"target": "es5",

// 这可以对 `this` 上的数据 property 进行更严格的推断

"strict": true,

// 如果使用 webpack 2+ 或 rollup，可以利用 tree-shake:

"module": "es2015",

"moduleResolution": "node"

}

}

注意你需要引入 strict: true (或者至少 noImplicitThis: true，这是 strict 模式的一部分) 以利用组件方法中 this 的类型检查，否则它会始终被看作 any 类型。

参阅 TypeScript 编译器选项文档 (英) 了解更多。

开发工具链

工程创建

Vue CLI 3 可以使用 TypeScript 生成新工程。创建方式：

# 1. 如果没有安装 Vue CLI 就先安装

npm install --global @vue/cli

# 2. 创建一个新工程，并选择 "Manually select features (手动选择特性)" 选项

vue create my-project-name

编辑器支持

要使用 TypeScript 开发 Vue 应用程序，我们强烈建议您使用 Visual Studio Code，它为 TypeScript 提供了极好的“开箱即用”支持。如果你正在使用单文件组件 (SFC)，可以安装提供 SFC 支持以及其他更多实用功能的 Vetur 插件。

WebStorm 同样为 TypeScript 和 Vue 提供了“开箱即用”的支持。

基本用法

要让 TypeScript 正确推断 Vue 组件选项中的类型，您需要使用 Vue.component 或 Vue.extend 定义组件：

import Vue from 'vue'

const Component = Vue.extend({

// 类型推断已启用

})

const Component = {

// 这里不会有类型推断，

// 因为 TypeScript 不能确认这是 Vue 组件的选项

}

基于类的 Vue 组件

如果您在声明组件时更喜欢基于类的 API，则可以使用官方维护的 vue-class-component 装饰器：

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 property

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)

}

}

增强类型以配合插件使用

插件可以增加 Vue 的全局/实例 property 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)。

例如，声明一个 string 类型的实例 property $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 property 和组件选项：

import Vue from 'vue'

declare module 'vue/types/vue' {

// 可以使用 `VueConstructor` 接口

// 来声明全局 property

interface VueConstructor {

$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 property

console.log(Vue.$myGlobal)

// 额外的组件选项

var vm = new Vue({

myOption: 'Hello'

})

标注返回值

因为 Vue 的声明文件天生就具有循环性，TypeScript 可能在推断某个方法的类型的时候存在困难。因此，你可能需要在 render 或 computed 里的方法上标注返回值。

import Vue, { VNode } from 'vue'

const Component = Vue.extend({

data () {

return {

msg: 'Hello'

}

},

methods: {

// 需要标注有 `this` 参与运算的返回值类型

greet (): string {

return this.msg + ' world'

}

},

computed: {

// 需要标注

greeting(): string {

return this.greet() + '!'

}

},

// `createElement` 是可推导的，但是 `render` 需要返回值类型

render (createElement): VNode {

return createElement('div', this.greeting)

}

})

如果你发现类型推导或成员补齐不工作了，标注某个方法也许可以帮助你解决这个问题。使用 --noImplicitAny 选项将会帮助你找到这些未标注的方法。

标注 Prop

import Vue, { PropType } from 'vue'

interface ComplexMessage {

title: string,

okMessage: string,

cancelMessage: string

}

const Component = Vue.extend({

props: {

name: String,

success: { type: String },

callback: {

type: Function as PropType<() => void>

},

message: {

type: Object as PropType<ComplexMessage>,

required: true,

validator (message: ComplexMessage) {

return !!message.title;

}

}

}

})

如果你发现校验器并没有得到类型推导或命名补全不工作，用预期的类型标注参数可能会助你解决这类问题。

第28章 生产环境部署

以下大多数内容在你使用 Vue CLI 时都是默认开启的。该章节仅跟你自定义的构建设置有关。

开启生产环境模式

开发环境下，Vue 会提供很多警告来帮你对付常见的错误与陷阱。而在生产环境下，这些警告语句却没有用，反而会增加应用的体积。此外，有些警告检查还有一些小的运行时开销，这在生产环境模式下是可以避免的。

不使用构建工具

如果用 Vue 完整独立版本，即直接用 <script> 元素引入 Vue 而不提前进行构建，请记得在生产环境下使用压缩后的版本 (vue.min.js)。两种版本都可以在安装指导中找到。

使用构建工具

当使用 webpack 或 Browserify 类似的构建工具时，Vue 源码会根据 process.env.NODE\_ENV 决定是否启用生产环境模式，默认情况为开发环境模式。在 webpack 与 Browserify 中都有方法来覆盖此变量，以启用 Vue 的生产环境模式，同时在构建过程中警告语句也会被压缩工具去除。所有这些在 vue-cli 模板中都预先配置好了，但了解一下怎样配置会更好。

webpack

在 webpack 4+ 中，你可以使用 mode 选项：

module.exports = {

mode: 'production'

}

但是在 webpack 3 及其更低版本中，你需要使用 DefinePlugin：

var webpack = require('webpack')

module.exports = {

// ...

plugins: [

// ...

new webpack.DefinePlugin({

'process.env.NODE\_ENV': JSON.stringify('production')

})

]

}

Browserify

在运行打包命令时将 NODE\_ENV 设置为 "production"。这等于告诉 vueify 避免引入热重载和开发相关的代码。

对打包后的文件进行一次全局的 envify 转换。这使得压缩工具能清除掉 Vue 源码中所有用环境变量条件包裹起来的警告语句。例如：

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

或者在 Gulp 中使用 envify：

// 使用 envify 自定义模块指定环境变量

var envify = require('envify/custom')

browserify(browserifyOptions)

.transform(vueify)

.transform(

// 必填项，以处理 node\_modules 里的文件

{ global: true },

envify({ NODE\_ENV: 'production' })

)

.bundle()

或者配合 Grunt 和 grunt-browserify 使用 envify：

// 使用 envify 自定义模块指定环境变量

var envify = require('envify/custom')

browserify: {

dist: {

options: {

// 该函数用来调整 grunt-browserify 的默认指令

configure: b => b

.transform('vueify')

.transform(

// 必填项，以处理 node\_modules 里的文件

{ global: true },

envify({ NODE\_ENV: 'production' })

)

.bundle()

}

}

}

Rollup

使用 @rollup/plugin-replace：

const replace = require('@rollup/plugin-replace')

rollup({

// ...

plugins: [

replace({

'process.env.NODE\_ENV': JSON.stringify( 'production' )

})

]

}).then(...)

模板预编译

当使用 DOM 内模板或 JavaScript 内的字符串模板时，模板会在运行时被编译为渲染函数。通常情况下这个过程已经足够快了，但对性能敏感的应用还是最好避免这种用法。

预编译模板最简单的方式就是使用单文件组件——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，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“无样式内容闪烁 (fouc)”。将所有组件的 CSS 提取到同一个文件可以避免这个问题，也会让 CSS 更好地进行压缩和缓存。

查阅这个构建工具各自的文档来了解更多：

webpack + vue-loader (vue-cli 的 webpack 模板已经预先配置好)

Browserify + vueify

Rollup + rollup-plugin-vue

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)。利用这个钩子函数来配合错误跟踪服务是个不错的主意。比如 Sentry，它为 Vue 提供了官方集成。

# 第28章 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。更多细节可以移步 vue-router 文档。

从零开始简单的路由

如果你只需要非常简单的路由而不想引入一个功能完整的路由库，可以像这样动态渲染一个页面级的组件：

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，你可以建立一个麻雀虽小但是五脏俱全的客户端路由器。可以直接看实例应用。

整合第三方路由

如果你有更偏爱的第三方路由，如 Page.js 或者 Director，整合起来也一样简单。这里有一个使用了 Page.js 的完整示例。

# 第29章 状态管理

类 Flux 状态管理的官方实现

由于状态零散地分布在许多组件和组件之间的交互中，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：我们有受到 Elm 启发的状态管理库。vuex 甚至集成到 vue-devtools，无需配置即可进行时光旅行调试 (time travel debugging)。

在 Vue Mastery 观看视频讲解

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 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 data 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：

var sourceOfTruth = {}

var vmA = new Vue({

data: sourceOfTruth

})

var vmB = new Vue({

data: sourceOfTruth

})

现在当 sourceOfTruth 发生变更，vmA 和 vmB 都将自动地更新它们的视图。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数据来源，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 的变更，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变更将会发生，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({

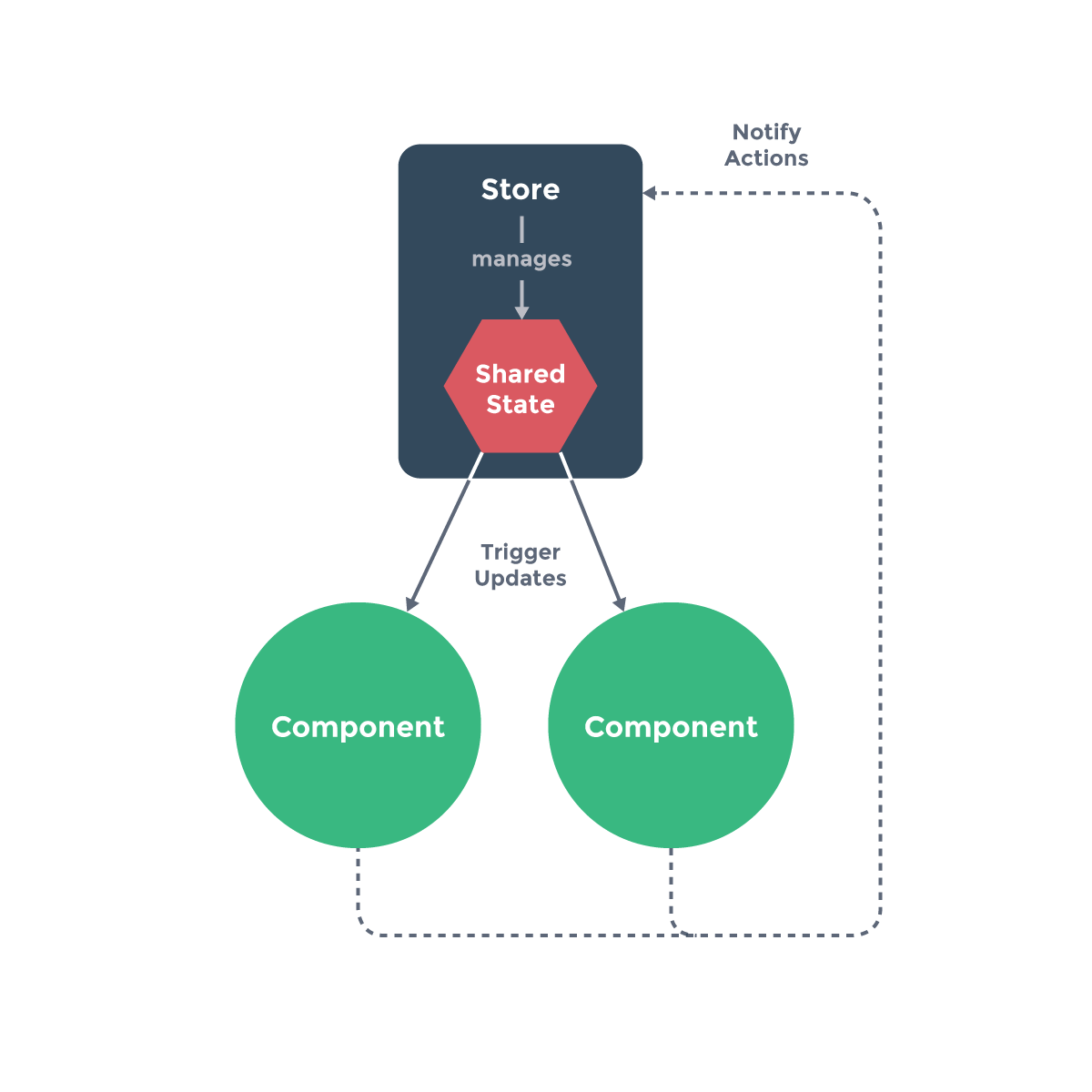
data: {

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，变更才能够被观察到。

接着我们继续延伸约定，组件不允许直接变更属于 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 架构。这样约定的好处是，我们能够记录所有 store 中发生的 state 变更，同时实现能做到记录变更、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，如果你已经读到这儿，或许可以去尝试一下！

# 第30章 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 v2.ssr.vuejs.org。

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 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。

Quasar Framework SSR + PWA

Quasar Framework 可以通过其一流的构建系统、合理的配置和开发者扩展性生成 (可选地和 PWA 互通的) SSR 应用，让你的想法的设计和构建变得轻而易举。你可以在服务端挑选执行超过上百款遵循“Material Design 2.0”的组件，并在浏览器端可用。你甚至可以管理网站的 <meta> 标签。Quasar 是一个基于 Node.js 和 webpack 的开发环境，它可以通过一套代码完成 SPA、PWA、SSR、Electron、Capacitor 和 Cordova 应用的快速开发。

# 第31章 安全

报告安全漏洞

当我们收到一个安全漏洞报告，将给予其最高优先级，并由全职贡献者停下手中的工作处理此事。如发现任何安全漏洞，请邮件给 security@vuejs.org。

虽然发现新安全漏洞是比较罕见的事，我们仍推荐始终使用最新版本的 Vue 及其官方的周边库，以确保应用尽可能安全。

第一原则：永远不要使用不可信任的模板

在使用 Vue 的时候最基本的安全规则是永远不要将不可信任的内容作为模板内容使用。这样做等价于允许在应用程序中执行任意的 JavaScript——甚至更糟的是如果在服务端渲染的话可能导致服务器被攻破。举个例子：

new Vue({

el: '#app',

template: `<div>` + userProvidedString + `</div>` // 永远不要这样做

})

Vue 的模板是被编译为 JavaScript 的，而其中的表达式会作为渲染流程的一部分执行。尽管该表达式是在一个特定的渲染上下文中进行运算的。考虑到潜在的全局运行环境的复杂性，作为类似 Vue 的框架，想要完全让代码远离潜在的恶意代码执行而不导致性能问题，是不切实际的。最直接的回避这类问题的方式就是确保 Vue 模板的内容始终是可信的且完全由你掌控。

Vue 的安全措施

HTML 内容

不论使用模板还是渲染函数，内容都会被自动转义。也就是说对于这份模板：

<h1>{{ userProvidedString }}</h1>

如果 userProvidedString 包含了：

'<script>alert("hi")</script>'

则它会被转义成为如下 HTML：

&lt;script&gt;alert(&quot;hi&quot;)&lt;/script&gt;

因此避免了脚本注入。该转义通过诸如 textContent 的浏览器原生的 API 完成，所以除非浏览器本身存在安全漏洞，否则不会存在安全漏洞。

Attribute 绑定

同样地，动态 attribute 绑定也会自动被转义。也就是说对于这份模板：

<h1 v-bind:title="userProvidedString">

hello

</h1>

如果 userProvidedString 包含了：

'" onclick="alert(\'hi\')'

则它会被转义成为如下 HTML：

&quot; onclick=&quot;alert('hi')

因此避免了通过闭合 title attribute 而注入新的任意 HTML。该转义通过诸如 setAttribute 的浏览器原生的 API 完成，所以除非浏览器本身存在安全漏洞，否则不会存在安全漏洞。

潜在危险

在任何 web 应用中，允许未过滤的用户提供的内容成为 HTML、CSS 或 JavaScript 都有潜在的危险，因此应当尽可能避免。尽管如此，有些情况下的风险是可接受的。

例如，类似 CodePen 和 JSFiddle 这样的服务允许用户提供的内容直接被执行，但这是预期行为，且在 iframe 中以某种程度被隔离在沙箱中。当一些重要功能不可避免地依赖引入一些安全漏洞，您的团队需要自行在该功能的重要性和漏洞带来的最坏场景间进行权衡。

注入 HTML

如你之前学到的，Vue 会自动转义 HTML 内容，以避免向应用意外注入可执行的 HTML。然而，某些情况下你清楚这些 HTML 是安全的，这时你可以显式地渲染 HTML 内容：

使用模板：

<div v-html="userProvidedHtml"></div>

使用渲染函数：

h('div', {

domProps: {

innerHTML: this.userProvidedHtml

}

})

使用基于 JSX 的渲染函数：

<div domPropsInnerHTML={this.userProvidedHtml}></div>

注意永远不要认为用户提供的 HTML 是 100% 安全的，除非它是在一个 iframe 沙盒里或者应用中只有编写这些 HTML 的用户可以接触到它。除此之外，允许用户撰写其自己的 Vue 模板会带来类似的危险。

注入 URL

在类似这样的 URL 中：

<a v-bind:href="userProvidedUrl">

click me

</a>

如果没有对该 URL 进行“过滤”以防止通过 javascript: 来执行 JavaScript，则会有潜在的安全问题。有一些库如 sanitize-url 可以帮助你做这件事，但请注意：

只要你是在前端进行 URL 过滤，那么就已经有安全问题了。用户提供的 URL 永远需要通过后端在入库之前进行过滤。然后这个问题就会在每个客户端连接该 API 时被阻止，包括原生移动应用。还要注意，甚至对于被过滤过的 URL，Vue 仍无法帮助你保证它们会跳转到安全的目的地。

注入样式

来看这个示例：

<a

v-bind:href="sanitizedUrl"

v-bind:style="userProvidedStyles"

>

click me

</a>

让我们假设 sanitizedUrl 已经被过滤过了，所以这已经是一个完全真实的 URL 且没有 JavaScript。但通过 userProvidedStyles，恶意用户仍可以提供 CSS 来进行“点击诈骗”，例如将链接的样式设置为一个透明的方框覆盖在“登录”按钮之上。然后再把 https://user-controlled-website.com/ 做成你的应用的登录页的样子，它们就可能获取一个用户真实的登录信息。

你可以想象到，允许用户为一个 <style> 元素提供内容，将产生甚至更严重的安全漏洞，以使得用户完全控制整个页面的样式。这就是为什么 Vue 要在模板内避免渲染 style 标签，例如：

<style>{{ userProvidedStyles }}</style>

为了确保用户完全远离点击诈骗，我们推荐只允许在一个 iframe 沙盒内进行 CSS 的完全控制。或让用户通过一个样式绑定来控制，我们推荐使用其对象语法且只允许用户提供特定的可以安全控制的 property 的值。例如：

<a

v-bind:href="sanitizedUrl"

v-bind:style="{

color: userProvidedColor,

background: userProvidedBackground

}"

>

click me

</a>

注入 JavaScript

我们强烈不鼓励使用 Vue 渲染 <script> 元素，因为模板和渲染函数永远不应该产生副作用。然而，这并不是唯一包含可能在运行时会被视为 JavaScript 的字符串。

每个 HTML 元素都有接受 JavaScript 字符串作为其值的 attribute，如 onclick、onfocus 和 onmouseenter。将用户提供的 JavaScript 绑定到它们任意当中都是一个潜在的安全风险，因此应该避免。

请注意，永远不要认为用户提供的 JavaScript 是 100% 安全的，除非它是在一个 iframe 沙盒里或者应用中只有编写该 JavaScript 的用户可以接触到它。

有的时候我们会收到在 Vue 模板中可以产生跨站脚本攻击 (XSS) 的安全漏洞报告。一般情况下，我们不会将这样的案例视为真正的安全漏洞，因为从以下两个可能允许 XSS 的场景看，不存在可行的办法来保护开发者：

开发者显式地要求 Vue 将用户提供的、未经过滤的内容作为 Vue 模板进行渲染。这是无法避免的不安全，Vue 没有办法知道其源头。

开发者向 Vue 挂载包含服务端渲染或用户提供的内容的 HTML 的整个页面。这实质上和问题 #1 是相同的，但是有的时候开发者可能没有意识到。这会使得攻击者提供作为普通 HTML 安全但对于 Vue 模板不安全的 HTML 以导致安全漏洞。最佳实践是永远不要向 Vue 挂载可能包含服务端渲染或用户提供的内容。

最佳实践

通用的规则是只要允许执行未过滤的用户提供的内容 (不论作为 HTML、JavaScript 甚至 CSS)，你就可能令自己处于被攻击的境地。这些建议实际上不论使用 Vue 还是别的框架甚至不使用框架，都是成立的。

除了上述关于潜在危险的建议，我们也推荐自行熟悉以下资料：

HTML5 Security Cheat Sheet

OWASP’s Cross Site Scripting (XSS) Prevention Cheat Sheet

然后利用学到的知识，对那些包含了第三方组件或通过其它方式影响渲染到 DOM 的内容的依赖的源代码进行重新审查，以发现潜在的危险模式。

后端协作

HTTP 安全漏洞，诸如伪造跨站请求 (CSRF/XSRF) 和跨站脚本注入 (XSSI)，都是后端重点关注的方向，因此并不是 Vue 所担心的。尽管如此，和后端团队交流学习如何和他们的 API 最好地进行交互，例如在表单提交时提交 CSRF token，永远是件好事。

服务端渲染 (SSR)

使用 SSR 时存在额外的安全考量，因此请确认遵循我们的 SSR 文档中概括出的最佳实践以避免安全漏洞。

# 第32章 深入响应式原理

现在是时候深入一下了！Vue 最独特的特性之一，是其非侵入性的响应式系统。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，这样你可以避开一些常见的问题。在这个章节，我们将研究一下 Vue 响应式系统的底层的细节。

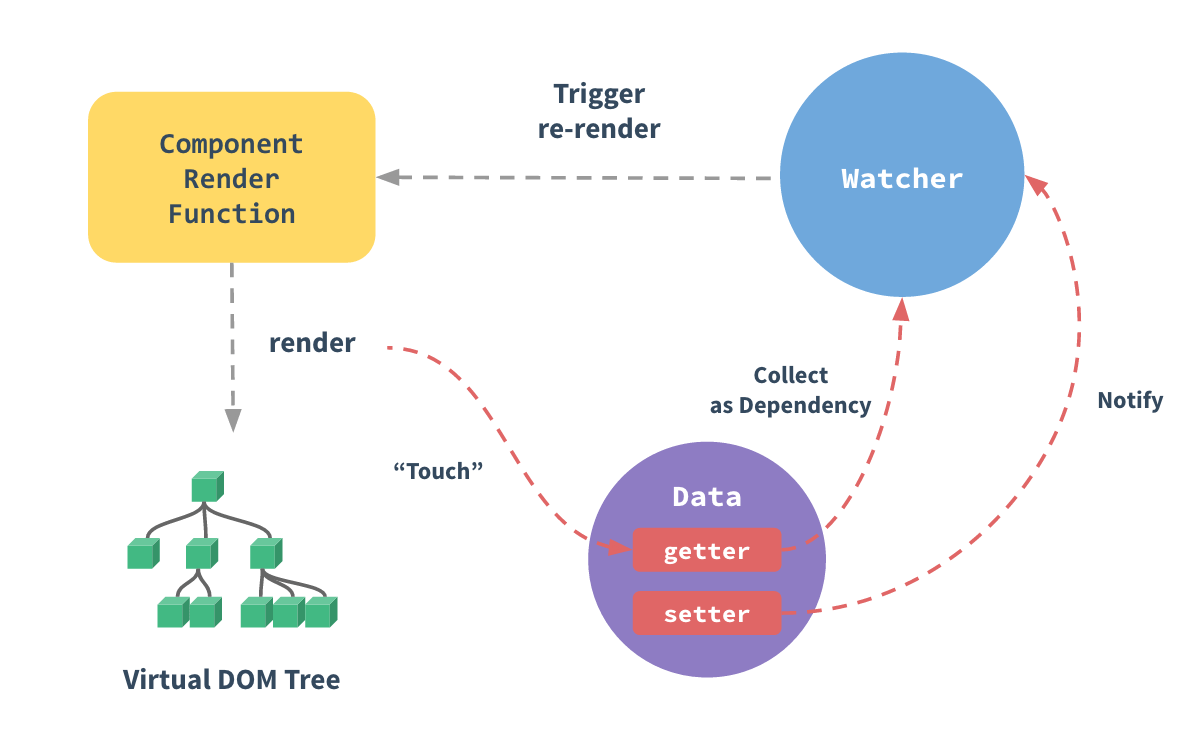
如何追踪变化

在 Vue Mastery 观看视频讲解

当你把一个普通的 JavaScript 对象传入 Vue 实例作为 data 选项，Vue 将遍历此对象所有的 property，并使用 Object.defineProperty 把这些 property 全部转为 getter/setter。Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。

这些 getter/setter 对用户来说是不可见的，但是在内部它们让 Vue 能够追踪依赖，在 property 被访问和修改时通知变更。这里需要注意的是不同浏览器在控制台打印数据对象时对 getter/setter 的格式化并不同，所以建议安装 vue-devtools 来获取对检查数据更加友好的用户界面。

每个组件实例都对应一个 watcher 实例，它会在组件渲染的过程中把“接触”过的数据 property 记录为依赖。之后当依赖项的 setter 触发时，会通知 watcher，从而使它关联的组件重新渲染。



检测变化的注意事项

由于 JavaScript 的限制，Vue 不能检测数组和对象的变化。尽管如此我们还是有一些办法来回避这些限制并保证它们的响应性。

对于对象

Vue 无法检测 property 的添加或移除。由于 Vue 会在初始化实例时对 property 执行 getter/setter 转化，所以 property 必须在 data 对象上存在才能让 Vue 将它转换为响应式的。例如：

var vm = new Vue({

data:{

a:1

}

})

// `vm.a` 是响应式的

vm.b = 2

// `vm.b` 是非响应式的

对于已经创建的实例，Vue 不允许动态添加根级别的响应式 property。但是，可以使用 Vue.set(object, propertyName, value) 方法向嵌套对象添加响应式 property。例如，对于：

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)

有时你可能需要为已有对象赋值多个新 property，比如使用 Object.assign() 或 \_.extend()。但是，这样添加到对象上的新 property 不会触发更新。在这种情况下，你应该用原对象与要混合进去的对象的 property 一起创建一个新的对象。

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

this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })

对于数组

Vue 不能检测以下数组的变动：

当你利用索引直接设置一个数组项时，例如：vm.items[indexOfItem] = newValue

当你修改数组的长度时，例如：vm.items.length = newLength

举个例子：

var vm = new Vue({

data: {

items: ['a', 'b', 'c']

}

})

vm.items[1] = 'x' // 不是响应性的

vm.items.length = 2 // 不是响应性的

为了解决第一类问题，以下两种方式都可以实现和 vm.items[indexOfItem] = newValue 相同的效果，同时也将在响应式系统内触发状态更新：

// Vue.set

Vue.set(vm.items, indexOfItem, newValue)

// Array.prototype.splice

vm.items.splice(indexOfItem, 1, newValue)

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

vm.$set(vm.items, indexOfItem, newValue)

为了解决第二类问题，你可以使用 splice：

vm.items.splice(newLength)

声明响应式 property

由于 Vue 不允许动态添加根级响应式 property，所以你必须在初始化实例前声明所有根级响应式 property，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!'

如果你未在 data 选项中声明 message，Vue 将警告你渲染函数正在试图访问不存在的 property。

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 实例能更好地配合类型检查系统工作。但与此同时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结构 (schema)。提前声明所有的响应式 property，可以让组件代码在未来修改或给其他开发人员阅读时更易于理解。

异步更新队列

可能你还没有注意到，Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化，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变更。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。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 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 和 setImmediate，如果执行环境不支持，则会采用 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，组件会在下一个事件循环“tick”中更新。多数情况我们不需要关心这个过程，但是如果你想基于更新后的 DOM 状态来做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员使用“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们必须要这么做。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)。这样回调函数将在 DOM 更新完成后被调用。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: '未更新'

}

},

methods: {

updateMessage: function () {

this.message = '已更新'

console.log(this.$el.textContent) // => '未更新'

this.$nextTick(function () {

console.log(this.$el.textContent) // => '已更新'

})

}

}

})

因为 $nextTick() 返回一个 Promise 对象，所以你可以使用新的 ES2017 async/await 语法完成相同的事情：

methods: {

updateMessage: async function () {

this.message = '已更新'

console.log(this.$el.textContent) // => '未更新'

await this.$nextTick()

console.log(this.$el.textContent) // => '已更新'

}

}