目录

[第1章 Vue.js介绍 6](#_Toc66390460)

[1.1什么是Vue.js 6](#_Toc66390461)

[1.2 Vue与其它框架 6](#_Toc66390462)

[1.2.1 Vue与React 6](#_Toc66390463)

[1.2.2 Vue与AngularJS 7](#_Toc66390464)

[1.2.3 Vue与Angular（Angular2） 7](#_Toc66390465)

[1.2.4 Vue与其他框架 8](#_Toc66390466)

[1.3 Vue 3的简述 8](#_Toc66390467)

[1.3.1 Vue 3的新特性 8](#_Toc66390468)

[1.3.2从Vue 2开始的重大改变 9](#_Toc66390469)

[1.3.3库与工具的支持 9](#_Toc66390470)

[1.4 本章小结 9](#_Toc66390471)

[1.5 练习题 10](#_Toc66390472)

[第2章 第一个Vue应用 11](#_Toc66390473)

[2.1准备Vue 3 11](#_Toc66390474)

[2.1.1引入Vue.js 11](#_Toc66390475)

[2.1.2安装Vue Devtools 12](#_Toc66390476)

[2.2 Vue实例和数据绑定 14](#_Toc66390477)

[2.2.1构建Vue 3项目 14](#_Toc66390478)

[2.2.2项目实例：Hello Vue3 15](#_Toc66390479)

[2.2.3 setup与生命周期 16](#_Toc66390480)

[2.2.4数据 18](#_Toc66390481)

[2.2.5数据实例：显示响应式对象 19](#_Toc66390482)

[2.2.6方法 20](#_Toc66390483)

[2.2.7方法实例：修改响应式对象的值 20](#_Toc66390484)

[2.3 本章小结 21](#_Toc66390485)

[2.4 练习题 22](#_Toc66390486)

[第3章 Vue的内置指令与语法 23](#_Toc66390487)

[3.1插值绑定 23](#_Toc66390488)

[3.1.1文本插值与表达式 23](#_Toc66390489)

[3.1.2过滤器 25](#_Toc66390490)

[3.1.3 HTML插值 25](#_Toc66390491)

[3.2计算属性 26](#_Toc66390492)

[3.2.1计算属性的概念 26](#_Toc66390493)

[3.2.2计算属性的setter 28](#_Toc66390494)

[3.2.3侦听属性watch 29](#_Toc66390495)

[3.3 v-bind属性绑定 31](#_Toc66390496)

[3.3.1 v-bind指令 31](#_Toc66390497)

[3.3.2绑定class、style与prop 32](#_Toc66390498)

[3.4 v-model双向绑定 33](#_Toc66390499)

[3.4.1 v-model指令 33](#_Toc66390500)

[3.4.2 v-model与修饰符 34](#_Toc66390501)

[3.4.3双向绑定实例：制作问卷 35](#_Toc66390502)

[3.5 v-if/v-show条件渲染 39](#_Toc66390503)

[3.5.1 v-if，v-else与v-else-if 39](#_Toc66390504)

[3.5.2 v-show 41](#_Toc66390505)

[3.5.3 v-if对比v-show 41](#_Toc66390506)

[3.5.4条件渲染实例：按钮权限控制 41](#_Toc66390507)

[3.6 v-for列表渲染 44](#_Toc66390508)

[3.6.1 v-for指令 44](#_Toc66390509)

[3.6.2在v-for里使用对象 45](#_Toc66390510)

[3.6.3列表的更新 46](#_Toc66390511)

[3.6.4列表渲染的key 47](#_Toc66390512)

[3.6.5过滤与排序 48](#_Toc66390513)

[3.6.6 v-for与v-if共用 49](#_Toc66390514)

[3.6.7列表渲染实例：帖子列表 50](#_Toc66390515)

[3.7 v-on事件绑定 52](#_Toc66390516)

[3.7.1 v-on指令 52](#_Toc66390517)

[3.7.2事件修饰符 53](#_Toc66390518)

[3.8指令在Vue 3.x中的变化 54](#_Toc66390519)

[3.8.1 v-if与v-for的key 54](#_Toc66390520)

[3.8.2 v-if与v-for的优先级 55](#_Toc66390521)

[3.8.3 v-bind合并行为 55](#_Toc66390522)

[3.8.4 v-for中的Ref数组 55](#_Toc66390523)

[3.8.5 v-model 56](#_Toc66390524)

[3.9本章小结 57](#_Toc66390525)

[3.10练习题 58](#_Toc66390526)

[第4章 Class与Style绑定 59](#_Toc66390527)

[4.1绑定HTML Class 59](#_Toc66390528)

[4.1.1对象语法 59](#_Toc66390529)

[4.1.2数组语法 61](#_Toc66390530)

[4.2绑定内联样式 62](#_Toc66390531)

[4.2.1对象语法 62](#_Toc66390532)

[4.2.2数组语法 64](#_Toc66390533)

[4.3本章小结 64](#_Toc66390534)

[4.4练习题 64](#_Toc66390535)

[第5章 过滤器 65](#_Toc66390536)

[5.1内置过滤器 65](#_Toc66390537)

[5.1.1字母操作 65](#_Toc66390538)

[5.1.2 json过滤器 66](#_Toc66390539)

[5.1.3限制过滤器 66](#_Toc66390540)

[5.1.4 currency过滤器 68](#_Toc66390541)

[5.1.5 debounce过滤器 68](#_Toc66390542)

[5.2本章小结 69](#_Toc66390543)

[5.3练习题 69](#_Toc66390544)

[第6章 过渡与动画 70](#_Toc66390545)

[6.1过渡与动画概述 70](#_Toc66390546)

[6.1.1基于class的动画和过渡 70](#_Toc66390547)

[6.1.2基于style的动画和过渡 73](#_Toc66390548)

[6.2单元素的过渡 76](#_Toc66390549)

[6.2.1进入与离开过渡 76](#_Toc66390550)

[6.2.2 CSS过渡与动画 78](#_Toc66390551)

[6.2.3自定义过渡class类名 80](#_Toc66390552)

[6.2.4 JavaScript过渡 81](#_Toc66390553)

[6.3其他过渡 82](#_Toc66390554)

[6.3.1多元素过渡 82](#_Toc66390555)

[6.3.2过渡模式 84](#_Toc66390556)

[6.3.3列表过渡 86](#_Toc66390557)

[6.3.4列表过渡案例：打乱列表 90](#_Toc66390558)

[6.4本章小结 94](#_Toc66390559)

[6.5练习题 94](#_Toc66390560)

[第7章 组件 95](#_Toc66390561)

[7.1组件的注册 95](#_Toc66390562)

[7.1.1全局注册 95](#_Toc66390563)

[7.1.2局部注册 97](#_Toc66390564)

[7.2组件的数据传递 99](#_Toc66390565)

[7.2.1 props参数 99](#_Toc66390566)

[7.2.2组件通信 101](#_Toc66390567)

[7.2.3 v-model参数 105](#_Toc66390568)

[7.2.4 Vue 3中的v-model修饰符 106](#_Toc66390569)

[7.3 slot内容分发 109](#_Toc66390570)

[7.3.1 slot的基本用法 109](#_Toc66390571)

[7.3.2 slot的作用域 110](#_Toc66390572)

[7.3.3 slot的后备内容 110](#_Toc66390573)

[7.3.3具名插槽 111](#_Toc66390574)

[7.3.4作用域插槽 114](#_Toc66390575)

[7.4 动态组件 115](#_Toc66390576)

[7.4.1基础用法 115](#_Toc66390577)

[7.4.2 keep-alive 119](#_Toc66390578)

[7.5组件案例：完善标签页组件 124](#_Toc66390579)

[7.6组件在Vue 3中的变化 139](#_Toc66390580)

[7.6.1函数式组件 139](#_Toc66390581)

[7.6.2内联模板Attribute 139](#_Toc66390582)

[7.7本章小结 139](#_Toc66390583)

[7.8练习题 140](#_Toc66390584)

[第8章 前端路由 141](#_Toc66390585)

[8.1 vue-router的基本用法 141](#_Toc66390586)

[8.1.1安装vue-router 141](#_Toc66390587)

[8.1.2 vue-router的基本使用 142](#_Toc66390588)

[8.1.3跳转 143](#_Toc66390589)

[8.2 动态路由匹配 144](#_Toc66390590)

[8.2.1带参数的动态路由匹配 144](#_Toc66390591)

[8.2.2响应参数变化 145](#_Toc66390592)

[8.2.3参数全匹配 146](#_Toc66390593)

[8.3路由匹配的语法 147](#_Toc66390594)

[8.3.1自定义正则表达式 147](#_Toc66390595)

[8.3.2可选参数 148](#_Toc66390596)

[8.3.3可重复参数 149](#_Toc66390597)

[8.4嵌套路由 150](#_Toc66390598)

[8.5命名路由 152](#_Toc66390599)

[8.6重定向和别名 152](#_Toc66390600)

[8.6.1重定向 152](#_Toc66390601)

[8.6.2别名 153](#_Toc66390602)

[8.7向路由组件传递参数 155](#_Toc66390603)

[8.7.1基本语法 155](#_Toc66390604)

[8.7.2传递参数的模式 155](#_Toc66390605)

[8.8 vue-router 4的变化 157](#_Toc66390606)

[8.8.1 Vue Router的创建 157](#_Toc66390607)

[8.8.2新的history选项 157](#_Toc66390608)

[8.8.3删除\*路由 157](#_Toc66390609)

[8.8.4 <router-link>的修改 158](#_Toc66390610)

[8.8.5去除router.app 159](#_Toc66390611)

[8.8.6向route组件的<slot>传递内容 159](#_Toc66390612)

[8.8.7 $route属性编码 159](#_Toc66390613)

[8.9本章小结 159](#_Toc66390614)

[8.10练习题 160](#_Toc66390615)

[第9章 状态管理与Vuex 161](#_Toc66390616)

[9.1 Vuex简介 161](#_Toc66390617)

[9.1.1状态管理模式 161](#_Toc66390618)

[9.1.2安装Vuex 162](#_Toc66390619)

[9.1.3 Vuex的基本使用 163](#_Toc66390620)

[9.2 Vuex中的状态 165](#_Toc66390621)

[9.2.1单一状态树 165](#_Toc66390622)

[9.2.2将Vuex状态加入Vue组件 165](#_Toc66390623)

[9.2.3 mapState的使用 166](#_Toc66390624)

[9.2.4组件的本地状态 167](#_Toc66390625)

[9.3 Vuex中的getter 167](#_Toc66390626)

[9.3.1仓库的getter 167](#_Toc66390627)

[9.3.2属性式访问 168](#_Toc66390628)

[9.3.3方法式访问 169](#_Toc66390629)

[9.3.4 mapGetter的使用 169](#_Toc66390630)

[9.4 Vuex中的mutation 170](#_Toc66390631)

[9.4.1 mutation的有效负载 170](#_Toc66390632)

[9.4.2通过对象提交 171](#_Toc66390633)

[9.4.3 mutation的同步 171](#_Toc66390634)

[9.5 Vuex中的action 172](#_Toc66390635)

[9.5.1 action的基本使用 172](#_Toc66390636)

[9.5.2调度action 172](#_Toc66390637)

[9.5.3组成action 173](#_Toc66390638)

[9.6 Vuex中的模块 175](#_Toc66390639)

[9.7本章小结 176](#_Toc66390640)

[9.8练习题 177](#_Toc66390641)

[第10章 Vue工程实例 178](#_Toc66390642)

[10.1项目目录介绍 178](#_Toc66390643)

[10.1.1 dist文件夹 178](#_Toc66390644)

[10.1.2 node modules文件夹 179](#_Toc66390645)

[10.1.3 src文件夹 179](#_Toc66390646)

[10.2前端页面开发 180](#_Toc66390647)

[10.2.1 Vue文件 180](#_Toc66390648)

[10.2.2导入import 182](#_Toc66390649)

[10.3打包与部署 182](#_Toc66390650)

[10.3.1项目打包 182](#_Toc66390651)

[10.3.2项目部署 183](#_Toc66390652)

[10.3.3通过Github Action自动部署 185](#_Toc66390653)

[10.4本章小结 189](#_Toc66390654)

[10.5练习题 189](#_Toc66390655)

# 第1章 Vue.js介绍

本章将从Vue的概念出发，介绍Vue的特色以及与其他前端框架的不同。对于已经了解Vue 2的读者，本章也会介绍Vue 3的一些新特性。

## 1.1什么是Vue.js

Vue (读音类似view)是一种模型-视图-视图模型（Model-View-ViewModel，MVVM），View和Model是独立的，ViewModel是View 和Model交互的桥梁。当View的某个部分需要更新时，Vue会自动选择恰当的方法和时机进行更新。

Vue的开发者将其称为渐进式框架。与其他重量级框架不同的是，Vue的核心库只关注视图层，而且被设计为可以自底向上逐层应用。得益于Vue简单的API，以及可以阶段性使用的特点，使得Vue十分容易上手。同时，与第三方库或既有项目的整合也十分便捷。

## 1.2 Vue与其它框架

在 MVVM 框架中，Vue的表现十分优秀。原因在于Vue吸取了其他框架，例如React、Angular等的优势，在这一节将会介绍Vue与其他框架的对比。

### 1.2.1 Vue与React

React的特点是使用JavaScript一种语言就可以写前后端，但是因此HTML代码需要写在JavaScript文件中（现在的潮流也越来越多地将CSS也纳入到 JavaScript中来处理），同时前后端代码需要写在一起。这种编程方式也叫“多语言混合式编程” ，会导致代码难以理解、调试。

React与Vue有许多相似之处。首先它们都将注意力集中保持在核心库，使用虚拟DOM以降低页面开销，并且提供了响应式和组件化的视图组件。

在React应用中，当某个组件的状态发生变化时，它会以该组件为根，重新渲染整个组件子树。对于不必要子组件的重渲染则需要手动控制。而在Vue 应用中，组件的依赖是在渲染过程中自动追踪的，所以系统能精确知晓哪个组件确实需要被重渲染。Vue的这个特点使得开发者不再需要考虑此类优化。

在React中，所有的组件的渲染功能都依靠 JSX。但在Vue中也提供了渲染函数，也支持 JSX，同时HTML都是合法的Vue模板。对于CSS而言，在 React中是通过CSS-in-JS实现CSS作用域的，在Vue中则通过单文件组件里类似style的标签。

在应用开发方面，React Native支持使用相同的组件模型编写有本地渲染能力的iOS和 Android软件，并且能同时跨多平台开发。而Vue方面，Weex支持的原生应用的组件开发有待进一步成熟。

### 1.2.2 Vue与AngularJS

Vue的灵感来源于AngularJS，但是Vue解决了AngularJS中存在的许多问题。相比于AngularJS，Vue在速度与体积上都有很多优势。

Vue 与AngularJS相比最大的区别在于没有脏检测机制。在AngularJS中存在多个watcher，当watcher越来越多时，检测耗时会越来越长。因为作用域内的每一次变化，所有 watcher都要重新计算。并且，如果一些 watcher触发另一个更新，就会引发所有watcher重新计算，从而进入一种无限循环的脏检测。这种脏检测机制性能低下，而且有时没有简单的办法来优化有大量 watcher的作用域，并不适合大型Web应用。而Vue使用基于依赖追踪的观察系统并且异步队列更新，全局只设置一个watcher，用这一个watcher来记录和更新一组关联对象的值，因此所有的数据变化都是独立触发，除非它们之间有明确的依赖关系。

在数据流方面，Vue在不同组件间强制使用单向数据流，使得数据流更加清晰易懂。而AngularJS在单向数据流的视图渲染、事件绑定之外，还参与了View对Model层的数据更新，即双向数据绑定。

### 1.2.3 Vue与Angular（Angular2）

Angular具有优秀的组件系统，在AngularJS中的许多实现已经完全重写，API也完全改变了。重写过的Angular有着很快的速度，在性能测试数据上与Vue十分接近。

在体积方面，最近的 Angular版本中在使用了AOT和tree-shaking技术后使得最终的代码体积减小了许多。但Vue 项目还是要小得多。

在学习曲线上，学习Vue，只需要HTML和 JavaScript基础。而Angular 的学习曲线则比较陡峭，由于Angular的设计目标就是只针对大型的复杂应用，它的API更多，概念也更多。因此初学者上手时会有比较多的困难。

### 1.2.4 Vue与其他框架

Polymer也是Vue的一个灵感来源，两者具有相似的开发风格。最大的不同之处在于，Polymer需要重量级的polyfills来帮助工作，一方面导致了性能的下降，另一方面浏览器本身并不支持这些功能。相比而言，Vue不依赖polyfills，并且支持到了IE9。

Ember是一个全能框架。它提供了大量的约定，同时也就有学习曲线高、不灵活的问题。在性能上，Vue比 Ember好很多，Vue能够自动批量更新，而 Ember在性能敏感的场景时需要手动管理。Vue在普通JavaScript对象上建立响应，Ember中需要放在Ember对象内，并且手动为计算属性声明依赖。

## 1.3 Vue 3的简述

本章将会列举Vue 3的一些新特性以及与Vue 2不兼容的一些变更，以便有 Vue 2 经验的读者在开发过程中注意到这些改变。Vue 3所做改变的细节将会在每个对应章节内进行详述。

对比Vue 2，Vue 3通过tree-shaking减小约41%捆绑包体积、初始渲染速度提升约55%、更新速度提升约133%、内存使用率降低约54%等。

### 1.3.1 Vue 3的新特性

在基于对象的2.x API基本没有变化的情况下，Vue 3引入了Composition API。Composition API在代码组织模式上更灵活，类型推导也更稳定，使得Vue在大型应用中更有竞争力。目前已经有适用于Vue 2和Vue 3的Composition API实用程序库。

在Vue 3中提供了Teleport，允许开发者控制在DOM中哪个父节点下呈现 HTML，实现时将不必求助于全局状态或将其拆分为两个组件。

在Vue 3中，组件现在可以有多个根节点，也就是片段。前提是开发者明确定义属性应该分布在哪里。

Vue3中的<style scoped>现在可以包含全局规则或只针对插槽内容的规则

目前Vue 3有一些实验性特性，包括：单文件组件组合式 API 语法糖 (<script setup>) 、单文件组件状态驱动的 CSS 变量 (<style vars>) 。

### 1.3.2从Vue 2开始的重大改变

Global API方面，全局Vue API已更改为使用应用程序实例，全局和内部API已经被重构为可tree-shakable，通过tree-shakable可以减少多达一半的 runtime大小。

模板指令里，v-model的用法，<template v-for>和非v-for节点上key的用法，v-if和v-for的优先级都已经更改。v-bind="object"排序敏感，并且v-for中的ref不再注册ref数组。

Vue 3还移除了Vue 2中的一些API：keyCode支持作为v-on的修饰符、$on，$off和 $once实例方法、过滤、内联模板attribute、$destroy实例方法。

除了以上这些，Vue 3在组件、渲染函数等部分也做了一些改变，将会在对应的章节进行说明。

### 1.3.3库与工具的支持

Vue所有的官方库和工具都支持 Vue 3，并且计划在 2020 年底前稳定所有项目。

vue-cli现在提供了内置选项，可以在创建新项目时选择Vue 3预设。升级vue-cli并运行vue create来创建Vue 3项目。

Vue Router 4.0提供了Vue 3支持。新版本的Devtools目前只支持 Vue 3。但经过新的UI设计和内部结构重构，将会在未来的更新中支持多个Vue版本。

VSCode和Vetur都提供了对Vue 3的全面支持。

## 1.4 本章小结

本章从Vue的快速、轻量化特点作为切入，介绍了它作为渐进式的MVVM框架的特性。通过与其他框架，例如React、Angular的对比，读者可以更直观地选择适合实际开发的框架。而Vue作为渐进式框架，其平稳的学习曲线对于前端框架的新手而言是十分友好的，只需要HTML和 JavaScript基础知识便可以轻松上手。

新的Vue 3则在基于对象的Vue 2.x的基础上加入了Composition API，即组合式API。组合式API在代码组织模式上的灵活性，帮助Vue 3在大型应用中更有竞争力。其他的改变，例如组件、渲染、指令等，则会在本书后续的章节中进行详述。Vue所有的官方库和工具都支持 Vue 3。

在下一章中，将会介绍Vue 3的安装，并构建Vue 3的第一个项目。

## 1.5 练习题

* **问答题：**

1. 什么是MVVN框架？
2. 在学习Vue之前需要哪些基础知识？
3. Vue的灵感来源于哪个框架？Vue中组件的重渲染需要手动控制吗？
4. Vue 3对比Vue 2.x在那些性能方面有所提升？

# 第2章 第一个Vue应用

本章将会对Vue 3和工具的安装进行介绍，并在此基础上完成第一个Vue 3应用。Vue设计的初衷包括可以被渐进式地采用，这意味着它可以根据需求以多种方式集成到一个项目中。因此在最开始，我们可以很简单地引入Vue。

## 2.1准备Vue 3

### 2.1.1引入Vue.js

目前Vue的最新版本为3.0.2，将 Vue 3添加到项目中有三种主要方式。

第一种方式是在页面上以CDN package的形式导入，添加的位置如例**2-1**所示。

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <title>Vue</title>  <script src="https://unpkg.com/vue@next"></script>  </head>  <body>  </body>  </html> |

**例2-1 以CDN形式引入**

对于学习或制作原型，可以使用一下语句：

|  |
| --- |
| <script src="https://unpkg.com/vue@next"></script> |

对于生产环境，则推荐链接到一个明确的版本号和构建文件，例如：

|  |
| --- |
| <script src="https://unpkg.com/vue@3.0.2"></script> |

第二种方式是通过npm安装Vue 3，推荐在构建大型应用时使用。在Node的官网下载Node，同时会得到npm工具。npm 能很好地和Webpack、Browserify等模块打包器配合使用。在命令行输入以下指令安装Vue 3最新稳定版：

|  |
| --- |
| npm install vue@next |

由于npm的仓库源布置在国外，传输速度较慢，作为替代可以使用淘宝镜像源的cnpm：

|  |
| --- |
| npm install -g cnpm --registry=https://registry.npm.taobao.org |

之后，就可以用 cnpm 指令替代 npm 指令进行安装。

第三种方式是使用命令行工具Vue CLI。Vue CLI是一个由官方提供的基于Vue.js进行快速开发的完整系统，为单页面应用(SPA)快速搭建繁杂的脚手架。它确保了各种构建工具能够基于智能的默认配置平稳衔接，这样开发者可以专注在开发应用上。Vue CLI可以快速运行并带有热重载、保存时lint校验，以及生产环境可用的构建版本。

对于Vue 3，应该使用npm上可用的 Vue CLI v4.5 作为 @vue/cli@next。可以通过全局重新安装最新版本的 @vue/cli 进行升级。使用下列任意一条指令进行安装。

|  |
| --- |
| yarn global add @vue/cli@next  npm install -g @vue/cli@next |

然后在Vue项目中运行：

|  |
| --- |
| vue upgrade --next |

### 2.1.2安装Vue Devtools

在开始开发Vue之前，推荐在浏览器中安装Vue Devtools，它可以方便开发者在一个更友好的界面中调试 Vue 应用。Vue Devtools提供了一个查看Vue组件和全局状态管理器Vuex中数据的界面。

对于Google Chrome和Firefox，都可以在浏览器的应用商店中搜索Vue Devtools直接安装。

访问商店有困难的情况下也可以根据以下步骤进行手动安装：

1. 在git bash中输入以下指令，克隆Vue Devtools仓库

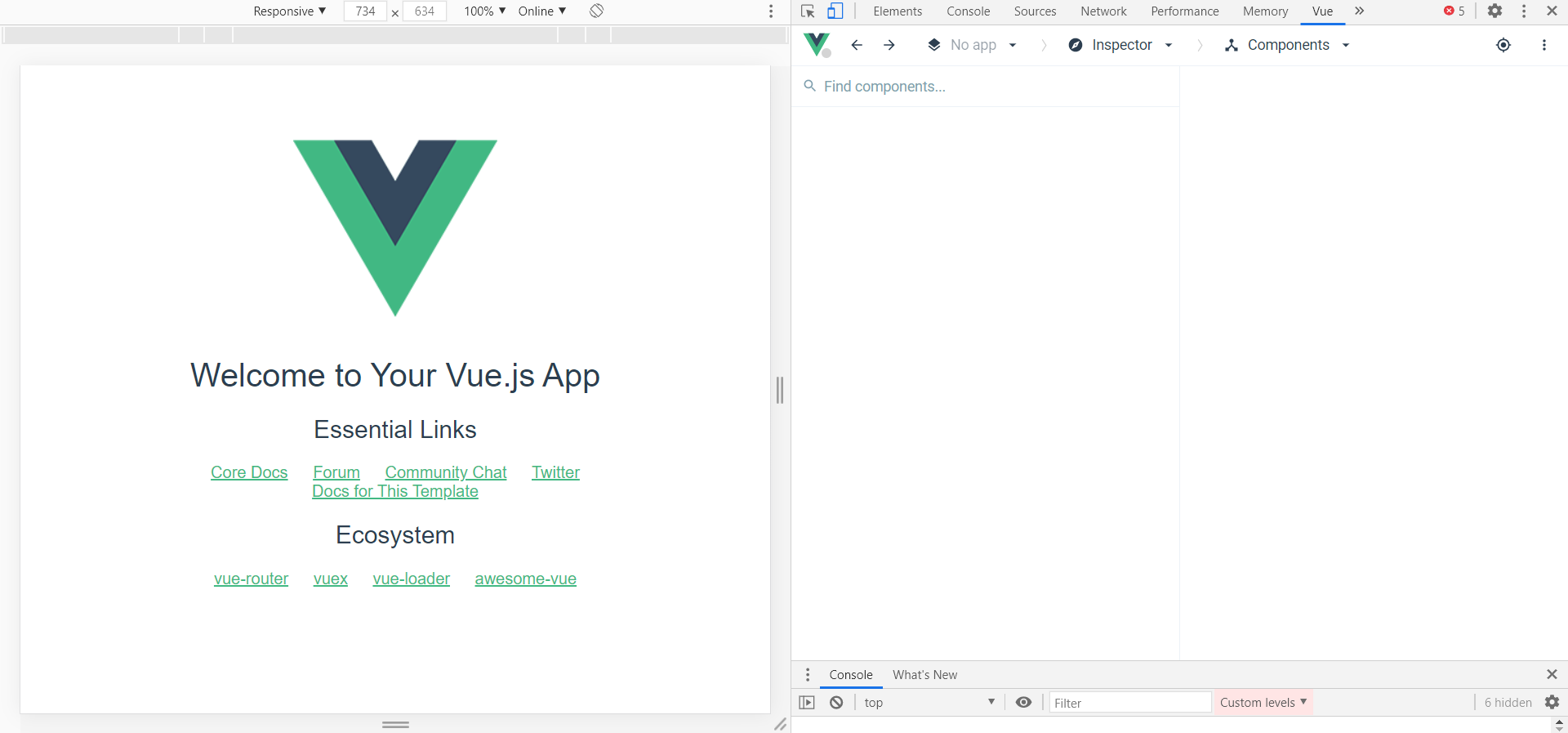
|  |
| --- |
| git clone https://github.com/vuejs/vue-devtools.git |

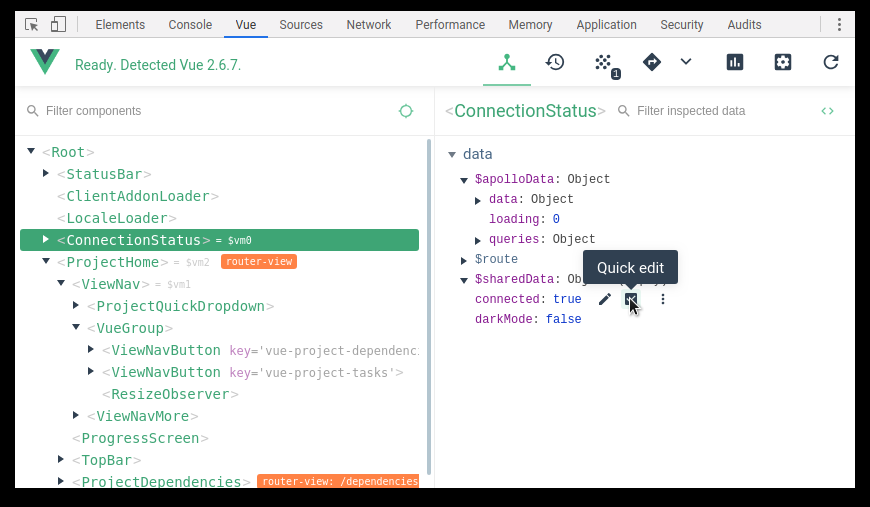
1. 进入新创建的vue-devtools文件夹
2. 在命令行中分别运行以下指令

|  |
| --- |
| yarn install  yarn run build |

1. 打开Chrome浏览器
2. 选择【菜单】→【更多工具】→【扩展程序】命令
3. 单击右上角的【开发者模式】按钮
4. 单击左侧的【加载已解压的扩展程序】按钮
5. 选择文件夹：vue-devtools/packages/shell-chrome/
6. 等待安装完成

安装完成后可以在网页点击F12呼出调试工具，查看Vue Devtools，其界面如图**2-1**所示。





**图2-1 Vue Devtools界面**

## 2.2 Vue实例和数据绑定

### 2.2.1构建Vue 3项目

Vue 3支持使用Vite构建一个项目。Vite是一个web开发构建工具，可以快速提供代码。通过在命令行中运行以下命令，可以使用Vite快速构建Vue项目。

使用npm：

|  |
| --- |
| npm init vite-app <project-name>  cd <project-name>  npm install  npm run dev |

或者使用yarn：

|  |
| --- |
| yarn create vite-app <project-name>  cd <project-name>  yarn  yarn dev |

### 2.2.2项目实例：Hello Vue3

在项目创建完成后，在***App.vue***与***main.js***中编写第一个Vue 3应用，代码分别如例**2-2**与例**2-3**所示。

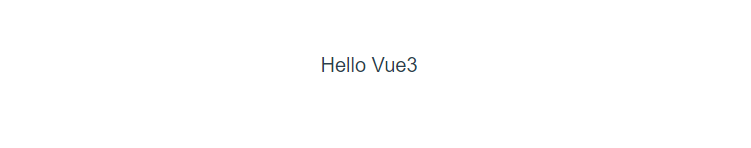
|  |
| --- |
| <template>  <div id="app">  <p>{{ hello }}</p>  </div>  </template>  <script>  import { ref } from "vue";  export default {  setup() {  const hello = ref("Hello Vue3");  return {  hello,  };  },  };  </script> |

**例2-2 App.vue中的代码**

|  |
| --- |
| import { createApp } from 'vue'  import App from './App.vue'  import './index.css'  createApp(App).mount('#app') |

**例2-3 main.js中的代码**

使用npm run dev指令正确运行后将会在网页上看到Hello Vue3的字样，如图**2-2**所**示**。至此，第一个Vue 3应用创建成功。



**图2-2 第一个Hello Vue3**

对于这个实例，在main.js中使用了Vue 3新增的createApp函数创建了一个新的应用实例，并挂载到名为app的DOM元素上。

在App.vue中定义了*hello*，并通过setup函数将其初始值设为'Hello Vue3'。其中，setup函数是组合API的入口点，而ref函数则帮助Vue监听*hello*，当它发生变化，Vue会自动更新视图。Vue会声明式地将数据渲染进DOM的系统，所有东西都是响应式的，也就是说当数据发生变化时，会同步更新其数据链和作用域中所有的相关状态。

### 2.2.3 setup与生命周期

setup函数是Vue 3中新增的一个组件选项，为使用Vue 3的组合式API 新特性提供了统一的入口。会在原有的beforeCreate函数之后、created函数之前执行。 setup的参数有两个：props和context。

在props中定义外界传递过来的参数名称和类型，并在setup函数的第一个形参接收。

setup函数的第二个形参context是一个上下文对象，包含了attrs、slots等属性。由于在setup函数中无法访问到this，因此使用形如context.attrs的方式访问这些属性。对于setup函数与props、context的使用，如例**2-4**所示。

|  |
| --- |
| <script>  export default {  setup(props, context) {  console.log(props.arg1)  console.log(context)  },  props: {  arg1: string,  arg2: int,  },  // context中的内容：  // context :{  // attrs: Data  // slots: Slots  // emit: (event: string, ...args: unknown[]) => void  // 以及parent, refs, root...  // }  };  </script> |

**例2-4 setup函数与props、context**

在Vue 3中，生命周期钩子注册函数也有所变化。在setup函数中可以使用直接导入的onX函数注册生命周期钩子。这些生命周期钩子注册函数只能setup期间同步使用，因为它们依赖于内部全局状态来定位当前活动实例，也就是正在调用其setup的组件实例。

在Vue 2.x的生命周期钩子注册函数中，beforeCreate与created函数删除，直接使用setup，其他生命周期钩子注册函数可以加上前缀on后在setup函数中使用，例如beforeMount变为onBeforeMount、mounted变为onMounted。如例2-5所示。

|  |
| --- |
| import { onMounted, onUpdated, onUnmounted } from ‘vue’;  export default {  setup(props, context) {  onMounted(() => {  console.log('mounted!')  })  onUpdated(() => {  console.log('updated!')  })  onUnmounted(() => {  console.log('unmounted!')  })  }  }; |

**例2-5 生命周期钩子注册函数**

### 2.2.4数据

在Vue 3中，通过ref函数与reactive函数在setup内创建响应式的对象。

ref函数接受一个内部值并返回一个响应式且可变的ref对象，这个对象上只包含一个.value属性。想要访问ref对象的内部值需要使用property .value。

reactive函数会返回原始对象的响应式副本，它影响所有嵌套property。等价于vue 2.x中的Vue.observable函数。

toRefs函数可以将reactive创建出来的响应式对象，转换为普通的对象，这个对象上的每个属性节点，都是ref类型的响应式数据。结果对象的每个property都是指向原始对象相应property的ref。

当把ref对象，挂载到reactive上时，会自动把响应式数据对象展开为原始的值，不需通过.value就可以直接被访问。

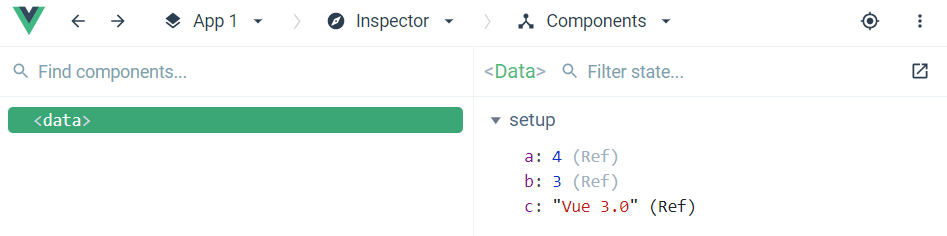
### 2.2.5数据实例：显示响应式对象

从vue中导入ref，reactive，toRefs函数后，在setup函数中创建ref对象*a*，以及reactive对象*state*，其中*state*包含*b*与*c*属性。在return中，将*state*转换为ref对象，使其具有响应式的属性。完整代码如例**2-6**所示。

|  |
| --- |
| <template>  <div id="app">  <p>{{ a }}</p>  <p>{{ b }}</p>  <p>{{ c }}</p>  </div>  </template>  <script>  import { ref, reactive, toRefs } from "vue";  export default {  setup() {  const a = ref(2); //一个响应式且可变的ref对象,此时a的值为2  //创建对象的响应式副本，并非直接响应式  const state = reactive({  b: 3,  c: "Vue 3.0",  });  a.value = 4;  return {  a,  ...toRefs(state), //toRef()函数将data转化为响应式  };  },  };  </script> |

**例2-6 main.js中的代码**

使用Vue Devtools可以查看*a*，*b*，*c*的值以及它们的类型，如图**2-3**所示。



**图2-3 使用Vue Devtools查看样例**

### 2.2.6方法

在Vue 2.x中，可以用methods选项向组件实例添加方法，在Vue 3中仍然支持。而Vue 3中，还可以将原来methods中的方法写在setup函数中，与Vue 2.x一样使用。需要注意的是，想要访问到方法，同样需要在setup的return中写入方法名。

### 2.2.7方法实例：修改响应式对象的值

在之前的数据实例中，再添加一个按钮，并赋予一个@click事件，在点击之后修改*a*，*b*，*c*的值。template中添加的代码如例**2-7**所示。

|  |
| --- |
| <button @click="changeBtn">改变</button> |

**例2-7 在template中添加按钮**

之后，在setup方法中编写修改值得方法，其中修改*a*的值需要通过*a*.value。编写完成后，在return中返回方法名称。setup中加入的代码如例**2-8**所示。

|  |
| --- |
| const changeBtn = () => {  console.log("---changing------");  state.b = 5;  state.c = "changed";  a.value = 8;  };  return {  a,  ...toRefs(state), //toRef()函数将data转化为响应式  changeBtn  }; |

**例2-8 在*setup*中编写修改方法**

在网页中点击按钮，可以看到*a*，*b*，*c*的值被修改。修改前后的对比如图**2-4**所示。

**图2-4 值修改前后对比**

## 2.3 本章小结

本章介绍了Vue 3的三种引入方式，分别为：CDN Package导入、npm安装以及命令行工具Vue CLI构建项目。此外，还介绍了浏览器调试工具Vue Devtools，它允许开发者在一个更友好的界面中审查和调试Vue应用。

在本章中，使用Vite快速构建了一个Vue 3项目。Vite是一个web开发构建工具，可以快速提供代码。在项目构建的实例中，使用了Vue 3新增的createApp函数创建了一个新的应用实例。

作为Vue 3的组合式AP的统一入口，本章介绍了setup函数的使用方法、其参数props和context，以及Vue 2.x中的生命周期钩子注册函数在setup函数中的使用方法。

对于数据与方法，Vue 3则在支持Vue 2.x中data与methods写法的同时，加入了通过ref函数与reactive函数在setup内创建响应式的对象，以及在setup函数内编写方法的语法。

最后，本章还提供了项目构建、数据、方法的实例，读者可以通过这些实例认识到Vue 3在实际项目中的使用。

在下一章中，将会介绍Vue的语法和指令，以及一些语法在Vue 3相对Vue 2.x的更新。

## 2.4 练习题

* **问答题：**

1. 引入Vue的方式有哪三种？分别尝试安装一下。
2. 简要说明ref、reactive与toRefs函数的作用。
3. *A*是一个ref对象，初始值为0，如何将*A*的值修改为2？

* **动手做：**

1. 尝试新建一个Vue 3项目，并在页面上打印“GOOD JOB!”。
2. 尝试在页面上实现一个计时器，点击按钮开始/暂停计时。

# 第3章 Vue的内置指令与语法

本章将会对Vue 3中的语法和内置指令进行介绍，包括差值绑定、计算属性、条件渲染指令、列表渲染指令等，对于在Vue 3中有变更的部分会在对应的小节中进行说明。

## 3.1插值绑定

### 3.1.1文本插值与表达式

文本插值最基本的方法是使用双大括号（Mustache语法）“{{ }}”，Vue将会获取计算后的值，将大括号里的内容替换为设定值，以文本的形式将其展示出来。通过任何方法修改数据设定值，大括号的内容都会被实时替换。例**2-2**中的*hello*，例**2-5**中的*a*，*b*，*c*都是通过这种方式在页面中显示数据的。

除了直接赋值，Mustache语法也接受表达式形式的值。表达式可由JavaScript表达式和过滤器构成。表达式可以有变量、数值、运算符等，表达式的值是它的运算结果。虽然不支持条件语句，但可以通过三元式实现简单的条件判断。

例**3-1**展示了通过文本插值与表达式计算变量、表达式、条件运算符的值，在页面中的效果如图**3-1**所示。

|  |
| --- |
| <template>  <div id="app">  <p><label>变量:</label> {{ num }}</p>  <p><label>表达式:</label> {{ 5 + 10 }}</p>  <p><label>条件运算符:</label> {{ true ? 5 : 10 }}</p>  </div>  </template>  <script>  import { ref } from "vue";  export default {  setup() {  const num = ref("2");  return {  num,  };  },  };  </script> </body>  </html> |

**例3-1 通过文本插值计算变量、表达式、条件运算符的值**



**图3-1 通过文本插值显示变量、表达式、条件运算符的值**

Vue的“{{ }}”内只支持单个表达式，不支持语句和流控制。并且在表达式中，不能使用用户自定义的全局变量。“{{ }}”可以放在HTML标签内，但Vue指令和自身特性内是不可以插值的。

如果想显示“{{ }}”标签，而不进行替换，可以使用v-pre跳过这个元素和它的子元素的编译过程。此外，HTML绑定Vue实例，在页面加载时可能会闪烁。原因是Vue来不及渲染，页面显示出了Vue源代码，可以使用v-cloak指令隐藏未编译的Mustache标签直到实例准备完毕。

### 3.1.2过滤器

在Vue 2.x中支持在“{{ }}”插值的尾部添加过滤器，用管道符“ |” 表示。经常用于格式化文本，比如字母全部大写、格式化日期等。过滤的规则是可以自定义的，通过给Vue实例添加filters来设置。在Vue 3.0中，过滤器已被移除，建议使用方法或计算属性来实现。

下面的例**3-2**实现了内置过滤器、过滤器串联与过滤器传参。uppercase是Vue的一个内置过滤器，可以将字符串转换为大写。通过使用多个管道符号可以将多个过滤器进行串联。

|  |
| --- |
| {{ string | uppercase }}  {{ string | filterA | filterB }}  {{ string | filter arg1 arg2 }} |

**例3-2 内置过滤器、过滤器串联与过滤器传参**

当有多个参数时，可以通过空格将参数分开，过滤器会将string作为第一个参数，arg1、arg2分别作为第二个与第三个参数传入。参数可以是表达式，也可以使用单引号传入字符串。

包括uppercase，Vue总共内置了10种过滤器，将会在过滤器的章节进行详细介绍。

### 3.1.3 HTML插值

HTML插值可以动态渲染DOM 节点，常用于处理开发者无可预知和难以控制的DOM 结构。与文本插值不同的是，文本插值中的代码被解释为节点的文本内容，而HTML插值中的代码则被渲染为视图节点。

对于值是HTML的片段，可以使用三个大括号“{{{ }}}”来绑定。也可以在标签内使用v-html=””的形式。所接受的字符串不会进行编译等操作，Vue会把被绑定的内容解析为DOM节点，按照普通HTML处理，从而实现动态渲染视图的效果。

需要注意的是，在网站上直接动态渲染任意HTML片段，容易导致XSS攻击。因此，开发者应尽量多地使用Vue 自身的模板机制，减少对HTML插值的使用，并且只对可信内容使用HTML插值。

## 3.2计算属性

在项目开发中，往往会在模板中使用表达式或过滤器来对数据进行处理。当表达式过长或者逻辑更复杂时，模板就会变得难以维护。为了避免这种问题，Vue提供了计算属性，对逻辑进行简化。

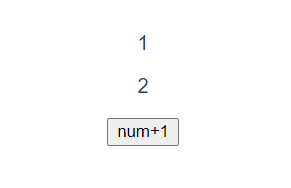
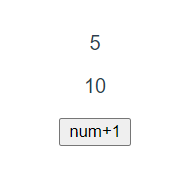
### 3.2.1计算属性的概念

计算属性会在其依赖属性的值发生变化时，对属性的值进行自动更新，同时更新相关的DOM 部分。通过从vue中导入computed来使用计算属性。

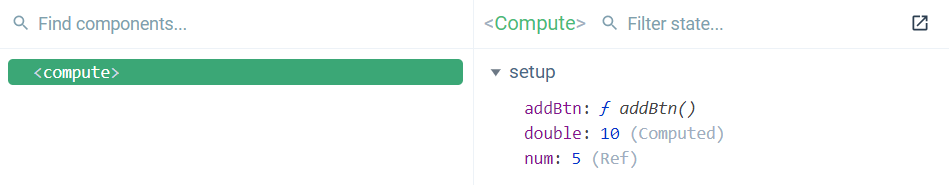
在例**3-3**中给出了计算属性的例子，*double*会始终保持为*num*的两倍，使用按钮增加*num*的值，*double*也会随之改变。显示结果如图**3-2**所示。在Vue Devtools中可以查看计算属性的值，如图**3-3**所示。

|  |
| --- |
| <template>  <div id="app">  <p>{{ num }}</p>  <p>{{ double }}</p>  <button @click="addBtn">num+1</button>  </div>  </template>  <script>  import { computed, toRefs, reactive, ref } from "vue";  export default {  setup() {  const data = reactive({  num: 1,  });  const double = computed(() => {  return data.num \* 2;  });  const addBtn = () => {  data.num++;  };  return {  ...toRefs(data),  double,  addBtn,  };  },  };  </script> |

**例3-3 计算属性**

**图3-2 计算属性的显示**



**图3-3 在Vue Devtools中查看属性**

除了以上这种写法，也可以在将computed写在*data*中，同样可以达到相同的效果。

|  |
| --- |
| const data = reactive({  num: 1,  doubleNum: computed(() => data.num \* 2),  }); |

**例3-4 计算属性的另一种写法**

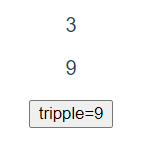
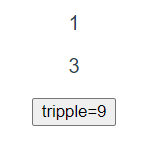
### 3.2.2计算属性的setter

在例3-3中，如果尝试修改*double*的值，会发现Vue会给出一个warning，原因是所创建的*double*是只读的计算属性。通过传入一个包含get和set函数的对象，可以得到一个可读可写的计算属性。

在例3-5中，创建了一个可读可写的计算属性*tripple*，它的值会保持为*num*的三倍。点击按钮将*tripple*的值修改为9，此时*num*的值也会被一同修改为3。修改前后的结果如图**3-4**所示。

|  |
| --- |
| const tripple = computed({  get: () => data.num \* 3,  set: (val) => {  data.num = val / 3;  },  }); |

**例3-5 可读可写的计算属性**



**图3-4 修改tripple的结果**

在computed中，get是取值函数，set是赋值函数。为计算属性赋值时，会触发set函数，触发set函数后，*num*的值会被更新。

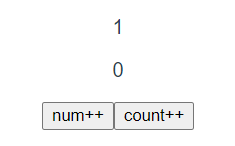
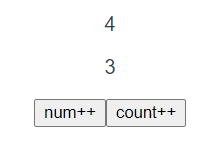
### 3.2.3侦听属性watch

watch函数用来监视指定数据项的变化，从而触发用户自定的操作。watch API完全等同于选项式API this.$watch。watch需要指定侦听的数据源，并在回调函数中执行副作用。默认情况下，回调仅在侦听的数据源发生改变时调用。通过从vue中导入watch来使用侦听属性。

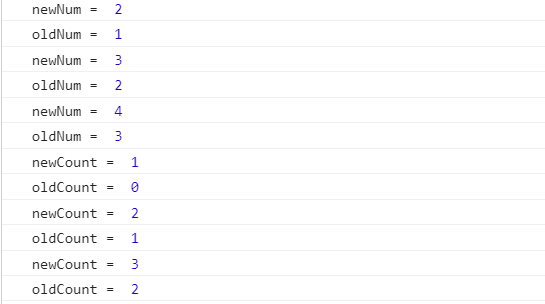
例**3-6**给出了使用watch侦听reactive类型的*num*与ref类型的*count*的例子。watch 可以获取到新值与更新前的值，当*num*与*count*改变时，会执行回调函数，在控制台打印更新前后的值。页面布局与控制台输出结果分别如图**3-5**与图**3-6**所示。

|  |
| --- |
| //侦听reactive类型的数据源  const data = reactive({  num: 1,  });  watch(  () => data.num,  (newNum, oldNum) => {  console.log("newNum = ", newNum);  console.log("oldNum = ", oldNum);  }  );  //侦听ref对象  const count = ref(0);  watch(count, (newCount, oldCount) => {  console.log("newCount = ", newCount);  console.log("oldCount = ", oldCount);  }); |

**例3-6 侦听num与count并打印变化**

**图3-5 通过按钮改变num与count**



**图3-6 控制台输出结果**

侦听器还可以使用数组同时侦听多个源，如例**3-7**所示。

|  |
| --- |
| //侦听多个reactive类型  watch(  [() => data.num1, () => data.num2],  ([newNum1, newNum2], [prevNum1, prevNum2]) => {  //Do something  }  );  //侦听多个ref类型  watch(  [count1, count2],  ([newCount1, newCount2], [prevCount1, prevCount2]) => {  //Do something  }  ); |

**例3-7 侦听多个数据源**

## 3.3 v-bind属性绑定

除了文本之外，DOM节点还有其他一些重要的属性，这些属性基本都可以用指令v-bind进行绑定。

### 3.3.1 v-bind指令

v-bind指令主要用于动态绑定DOM元素属性，可以将一个或多个attribute，或一个组件prop动态地绑定到表达式。如例**3-8**所示。

|  |
| --- |
| <!-- 绑定 attribute -->  <img v-bind:src="imageSrc" />  <!-- 缩写 -->  <img :src="imageSrc" />  <!-- 内联字符串拼接 -->  <img :src="'/path/to/images/' + fileName" />  <!-- 动态 attribute 名 -->  <button v-bind:[key]="value"></button>  <!-- 动态 attribute 名缩写 -->  <button :[key]="value"></button> |

**例3-8 v-bind示例**

### 3.3.2绑定class、style与prop

v-bind在绑定class或style的attribute时，支持其它类型的值，如数组或对象。虽然类名class和样式style可接收的类型都是字符串，但类名实际上是由数组拼接而成，而样式则是由对象键值对拼接而成的。

在绑定 prop时，prop必须在子组件中声明。可以用修饰符指定不同的绑定类型。如例3-9所示。

|  |
| --- |
| <!-- class 绑定 -->  <div :class="[classA, { classB: isB, classC: isC }]">  <!-- style 绑定 -->  <div :style="{ fontSize: size + 'px' }"></div>  <div :style="[styleObjectA, styleObjectB]"></div>  <!-- prop 绑定。"prop" 必须在 my-component 声明 -->  <my-component :prop="someThing"></my-component>  <!-- 通过 $props 将父组件的 props 一起传给子组件 -->  <child-component v-bind="$props"></child-component>  </div> |

**例3-9 绑定class、style、prop的attribute**

其中，类名*classB*与*classC*分别依赖于数据*isB*和*isC*，当*isB*和*isC*为true时，div会拥有类名*classB*与*classC*，为false时则没有。也可以使用三元表达式来根据条件切换类名。

使用:style时，CSS属性名称使用驼峰命名（camelCase）或短横分隔命名（kebab-case）。Vue会自动给特殊的CSS属性名称增加前缀，比如transform。

## 3.4 v-model双向绑定

表单控件在项目中十分常用，例如输入框、选择等，用它们可以完成数据的录入、提交等操作。通过使用指令v-model可以完成表单的数据双向绑定。

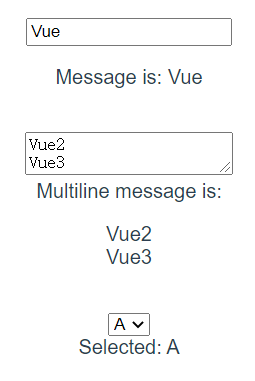
### 3.4.1 v-model指令

v-model指令用于在<input>、<textarea>及<select>等表单控件元素上创建双向绑定，它会根据控件类型自动选取正确的方法来更新元素。在修改表单元素值时，对应的实例中的属性值会被同时更新。反之，更改实例中的属性值，表单元素值也会被更新。

在例**3-10**中，给出了用v-model绑定<input>、<textarea>及<select>的简单例子，当表单中元素变化时，会更新属性值与视图。元素值更新前后如图**3-7**所示。

|  |
| --- |
| <!--输入-->  <input v-model="message" placeholder="请输入" />  <p>Message is: {{ message }}</p>  <br />    <!--多行文本-->  <textarea v-model="mulMessage" placeholder="请输入" ></textarea>  <br />  <span>Multiline message is:</span>  <p style="white-space: pre-line">{{ mulMessage }}</p>  <br />  <!--选择-->  <select v-model="selected">  <option>A</option>  <option>B</option>  <option>C</option>  </select>  <br />  <span>Selected: {{ selected }}</span> |

**例3-10 双向绑定message**

**图3-7 双向绑定message**

需要注意的是，在文本区域<textarea>中插值不起作用，应该使用v-model来代替。

另外，由于<select>的视图太差，而当前也不允许开发者自定义option的样式，所以一般都会使用其他元素来模拟下拉选择框。

在例**3-10**中，v-model绑定的值是一个字符串或布尔值。当需要绑定一个动态的数据，可以用v-bind来实现。

### 3.4.2 v-model与修饰符

v-model的修饰符可以用于控制数据同步的时机。各个修饰符的功能如表3-1所示。

|  |  |
| --- | --- |
| 修饰符 | 功能 |
| .lazy | 将用户输入的数据赋值于变量的时机延迟到数据改变时 |
| .number | 将用户输入转换为数值类型 |
| .tmn | 删除用户输入的首尾空白字符 |

**表3-1 v-mdoel的修饰符**

在输入框中，v-model默认是在每次input事件触发后将输入框的值与数据进行同步。使用修饰符.lazy会转变为在change事件中同步。数据不是实时改变的，而是在失焦或按回车时才更新。在例**3-10**的<input>中加入.lazy，在输入框内输入时，*message*不会立刻更新并显示，而是在输入回车，或是点击页面其他部分时更新*message*。

使用修饰符.number可以将输入转换为Number类型，默认情况下会将输入当做String类型。

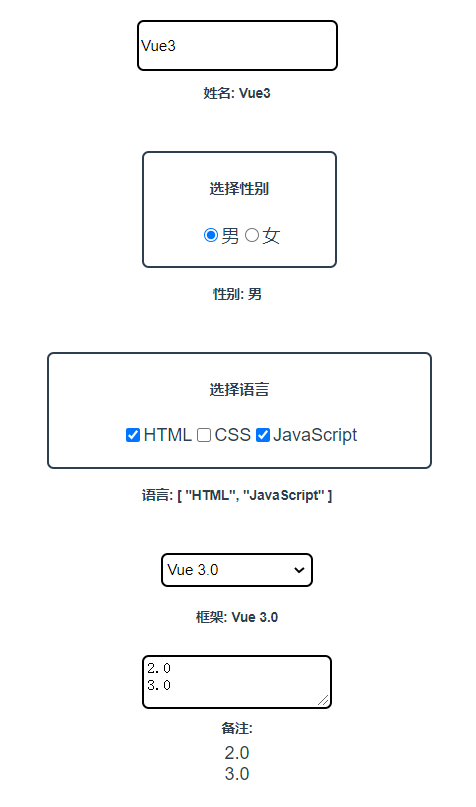
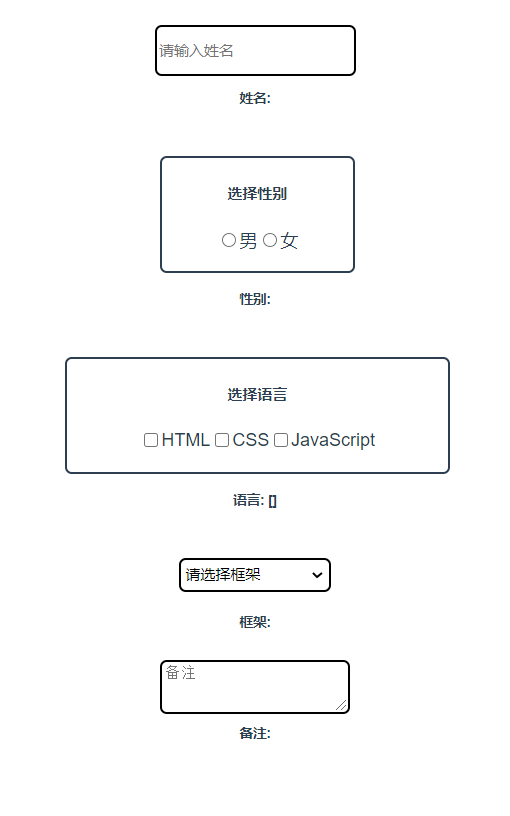
### 3.4.3双向绑定实例：制作问卷

学习了使用v-model进行双向绑定，现在可以制作一份实时显示各个表单控件元素中所填数据的问卷。通过指定<input>中的type，可以实现输入、单选、多选等。

通过问卷，采集了姓名、性别、编程语言、框架以及备注信息。代码如例3-11所示。代码运行结果如图3-8所示。

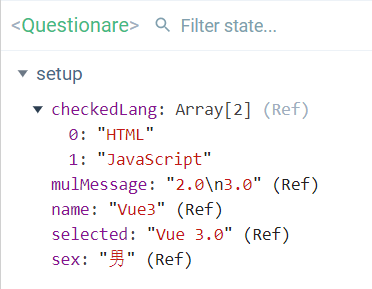
|  |
| --- |
| <template>  <div id="app">  <input v-model.trim="name" placeholder="请输入姓名" class="inText" />  <h6>姓名: {{ name }}</h6>  <br />  <div class="sex">  <h5>选择性别</h5>  <input type="radio" id="男" value="男" v-model="sex" />  <label for="男">男</label>  <input type="radio" id="女" value="女" v-model="sex" />  <label for="女">女</label>  </div>  <h6>性别: {{ sex }}</h6>  <br />  <div class="mulChoice">  <h5>选择语言</h5>  <input type="checkbox" id="HTML" value="HTML" v-model="checkedLang" />  <label for="HTML">HTML</label>  <input type="checkbox" id="CSS" value="CSS" v-model="checkedLang" />  <label for="CSS">CSS</label>  <input  type="checkbox"  id="JavaScript"  value="JavaScript"  v-model="checkedLang"  />  <label for="JavaScript">JavaScript</label>  </div>  <h6>语言: {{ checkedLang }}</h6>  <br />  <select v-model="selected" class="select">  <option disabled selected value style="display: none">请选择框架</option>  <option>Vue 2.x</option>  <option>Vue 3.0</option>  </select>  <h6>框架: {{ selected }}</h6>  <textarea  v-model="mulMessage"  placeholder="备注"  class="mulText"  ></textarea>  <br />  <h6>备注:</h6>  <p style="white-space: pre-line;margin-top: -20px ">{{ mulMessage }}</p>  <br />  </div>  </template>  <script>  import { reactive, ref, toRefs } from "vue";  export default {  setup() {  const data = reactive({  name: "",  mulMessage: "",  selected: "",  sex: "",  checkedLang: [],  });  return {  ...toRefs(data),  };  },  };  </script> |

**例3-11 问卷**



**图3-8 问卷前后效果**

可以看到，当每次在表单控件中输入信息时，数据会实时地被显示在页面中，而不需要对页面进行刷新操作。打开Vue Devtools，可以看到具体的数据内容以及对应的类型，每次对问卷中信息的修改也会直接反映在数据上。如图3-9所示。



**图3-9 在Vue Devtools中查看问卷数据**

## 3.5 v-if/v-show条件渲染

类似于其他程序中的条件语句if、else 、else if，Vue的条件指令同样可以根据表达式的值，在DOM中渲染或销毁元素与组件，称为条件渲染。

### 3.5.1 v-if，v-else与v-else-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真值的时候被渲染。

v-else-if用于充当v-if的“else-if块”，要紧跟v-if。当v-if中的表达式返回值为假时，开始判断v-else-if的表达式值，并根据返回值真假进行渲染。v-else-if可以连续使用。

v-else用来表示v-if的“else 块”，要紧跟v-else-if或v-if，当v-else-if与v-if的表达式值均为假时，渲染v-else中的内容。

在例3-12中使用了v-if、v-else-if与v-else进行条件渲染，当*type*的值为‘A’时，在页面中显示‘A’，为‘B’时显示‘B’，为‘C’时显示‘C’，其他情况下显示‘Not A/B/C’。

|  |
| --- |
| <template>  <div id="app">  <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>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  type: 'A',  });  return {  ...toRefs(data),  };  },  };  </script> |

**例3-12 根据*type*进行条件渲染**

其中，在进行相等的条件判断时，应该使用===。===是Ecmascript的语言，表示严格等于判断，由于JS中的==有缺陷，因此在绝大多数情况下，都是使用三个等号的形式。

因为v-if是一个指令，所以必须将它添加到一个元素上。因此，如果想要一次判断多个元素，可以把一个Vue内置的<template>元素当做不可见的包裹元素，在上面使用v-if。最终的渲染结果将不包含<template>元素。如例3-13所示，Content A，B，C将会由同一个v-if指令决定是否渲染。

|  |
| --- |
| <template v-if="true">  <h1>Content A</h1>  <h1>Content B</h1>  <h1>Content C</h1>  </template> |

**例3-13 在 <template>上使用条件渲染**

### 3.5.2 v-show

v-show指令同样可以用于根据条件展示元素。用法与v-if大致一样。不同的是带有 v-show的元素始终会被渲染并保留在DOM中，因为v-show只是简单地切换元素CSS属性的display。当条件判定为假时，元素的display将被赋值为none；反之，元素的display将被设置为原有值。另外，v-show不支持<template>元素，也不支持v-else。

### 3.5.3 v-if对比v-show

v-if 和v-show 具有类似的功能，不过v-if 才是真正的条件渲染，它会根据表达式适当地销毁或重建元素及绑定的事件或子组件。若表达式初始值为false ，则一开始元素或组件并不会渲染，只有当条件第一次变为真时才开始编译。

而v-show只是简单的CSS属性切换，无论条件真与否，都会被编译。元素或组件保留在DOM中。

根据各自的特性，不难发现v-if更适合条件不经常改变的场景，因为它切换开销相对较大，而v-show 适用于频繁切换条件。

### 3.5.4条件渲染实例：按钮权限控制

在实际的网站开发中，往往会遇到需要对用户进行区分的情况，不同的用户也往往拥有不同的权限或功能。此时，通过条件渲染，可以实现对于不同的用户种类渲染出不同页面的效果。

在例3-14中，实现了根据用户作为学生、老师、助教的不同身份进行不同前端页面显示的效果。代码的运行结果如图3-10所示。

|  |
| --- |
| <template>  <div id="app">  <h1>User Type:</h1>  <template v-if="user === 'student'">  <h2>{{ user }}</h2>  <br />  <button @click="joinClass">加入课程</button>  </template>  <template v-if="user === 'teacher'">  <h2>{{ user }}</h2>  <br />  <button @click="startClass">开始上课</button>  </template>  <template v-if="user === 'teaching assistant'">  <h2>{{ user }}</h2>  <br />  <button @click="manageStudent">管理学生</button>  </template>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  user: "teacher",  });  const joinClass = () => {  console.log("Request already sent!");  };  const startClass = () => {  console.log("Class begin!");  };  const manageStudent = () => {  console.log("Checking requests!");  };  return {  ...toRefs(data),  joinClass,  startClass,  manageStudent,  };  },  };  </script> |

**例3-14 根据用户身份显示不同前端页面**

**图3-10 不同用户身份下的显示与输出**

通过修改按钮的方法，可以实现对不同用户进行功能区分的效果。

## 3.6 v-for列表渲染

### 3.6.1 v-for指令

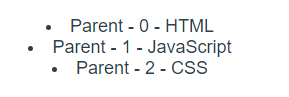
当需要将一个数组遍历或枚举一个对象循环显示时，就会用到列表渲染指令v-for来渲染一个列表。它的表达式需结合in来使用，类似item in items的形式。其中items是源数据数组，而item则是被迭代的数组元素的别名。也可以用of替代in作为分隔符。

v-for也可以像v-if一样用在内置标签＜template＞上，渲染多个元素。在自定义组件上同样可以使用v-for。但是由于组件有自己独立的作用域，数据不会被自动传递到组件中。此时，需要使用props把迭代数据传递到组件里。

在v-for块中，我们可以访问所有父作用域的属性。v-for还支持一个可选的第二个参数index，即当前项的索引。索引从0开始计数。具体代码如例3-15所示。

|  |
| --- |
| <template>  <div id="app">  <li v-for="(item,index) in items">  {{ parentMessage }} - {{ index }} - {{ item.message }}  </li>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  parentMessage: "Parent",  items: [  { message: "HTML" },  { message: "JavaScript" },  { message: "CSS" },  ],  });  return {  ...toRefs(data),  };  },  };  </script> |

**例3-15 用v-for显示数组中的message元素**



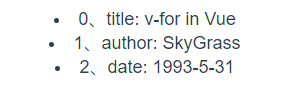
**图3-11 列表渲染**

### 3.6.2在v-for里使用对象

使用v-for可以遍历一个对象的属性。可以提供第二个的参数为属性的名称，即键名key。可以用第三个参数作为索引index。

|  |
| --- |
| <template>  <div id="app">  <li v-for="(value, name, index) in obj">  {{ index }}、{{ name }}: {{ value }}  </li>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  obj: {  title: "v-for in Vue",  author: "SkyGrass",  date: "1993-5-31",  },  });  return {  ...toRefs(data),  };  },  };  </script> |

**例3-16 用v-for遍历对象属性**



**图3-12 遍历对象属性结果**

### 3.6.3列表的更新

Vue的核心是数据与视图的双向绑定，当修改数组时，Vue会检测到数据变化，所以用v-for渲染的视图也会立即更新。Vue将被侦听的数组的变更方法进行了包裹，所以使用它们改变数组会触发视图更新。这些被包裹的方法与说明如表3-1所示。

|  |  |
| --- | --- |
| **名称** | **说明** |
| push() | 将一个或多个元素添加至数组末尾，返回新数组的长度 |
| pop() | 从数组中删除并返回最后一个元素 |
| shift() | 从数组中删除并返回第一个元素 |
| unshift() | 将一个或多个元素添加至数组开头，并返回新数组的长度 |
| splice() | 从数组中删除元素或向数组添加元素 |
| sort() | 对数组元素排序，默认按照Unicode编码排序，返回排序后的数组 |
| reverse() | 将数组中的元素位置颠倒，返回颠倒后的数组 |

**表3-1 数组的变更方法**

使用以上方法会改变被这些方法调用的原始数组，此外还有一些方法不会改变原数组，例如filter()、concat()、slice()，它们返回的是一个新数组。

需要注意的是，当直接使用下标或键名为数组或对象设置成员，以及修改数组长度时，Vue并不会将其加入数据响应式系统，如：arr[‘index’] = 'value' 、obj [' key '] = 'value'或arr.length=1。此时即使数据被修改，视图也不会进行更新。

### 3.6.4列表渲染的key

在使用v-for时，最好为每个迭代元素提供一个值不重复的key，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。因为，当列表渲染被重新执行(数组内容发生改变)时，如果不使用key，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。

通过为每项提供一个唯一的key，给 Vue进行提示，Vue就会根据key的变化重新排列节点顺序，并移除key不存在的节点。实质上，key的存在是为DOM节点标注了一个身份信息，让Vue能够有迹可循追踪到数据对应的节点。

在实战开发中，是否使用key都不会影响功能的实现。建议尽可能在使用v-for时提供key，如例3-17所示。

|  |
| --- |
| <div v-for="item in items" :key="item.id">  <!-- content -->  </div> |

**例3-17 指定v-for的key**

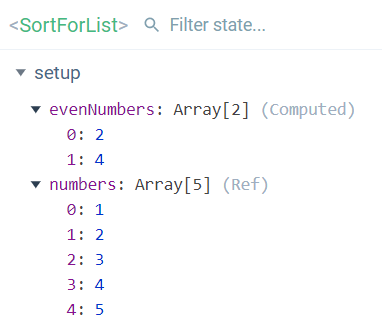
### 3.6.5过滤与排序

在展示数据时，往往需要在不变更或重置原始数据的情况下，显示经过过滤或排序的数组。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。

在例3-18中，对原有数组进行了过滤，筛选出了其中的偶数，Vue Devtools中的结果如图3-13所示。

|  |
| --- |
| <template>  <div id="app">  <li v-for="n in evenNumbers">  {{ n }}  </li>  </div>  </template>  <script>  import { computed, reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  numbers: [1, 2, 3, 4, 5],  });  const evenNumbers = computed(() => {  return data.numbers.filter((number) => number % 2 === 0);  });  return {  ...toRefs(data),  evenNumbers,  };  },  };  </script> |

**例3-18 过滤出数组中的偶数**



**图3-13 过滤数组的结果**

在计算属性不适用的情况下，例如，在嵌套列表渲染v-for的循环中，可以使用一个方法来达到相同的效果。

### 3.6.6 v-for与v-if共用

在多数情况下，不推荐在同一元素上同时使用v-if和v-for。当它们处于同一节点时，v-if的优先级比v-for更高，因此v-if将没有权限访问v-for里的变量。可以通过把v-for移动到外层的<template>标签中来实现相同的效果。

### 3.6.7列表渲染实例：帖子列表

学习完列表渲染v-for与条件渲染v-if，可以尝试制作一个帖子列表。帖子带有收藏功能，被收藏的帖子标题将会以红色显示，并带有收藏与取消收藏按钮。

具体代码如例3-19所示，代码的运行结果如图3-14所示。通过修改addStar(item)方法的内容，可以实现修改收藏状态的效果。在此基础上，也可以引申出置顶、删除帖子等许多方式。

|  |
| --- |
| <template>  <div id="app">  <div v-for="(item, index) in posts" :key="index" class="post">  <h1 v-if="item.isStar === true" style="color: red">  {{ item.title }}  </h1>  <h1 v-if="item.isStar === false">  {{ item.title }}  </h1>  <div class="content">  {{ item.detail }}  </div>  <button class="star" v-if="item.isStar === false" @click="addStar(item)">  收藏  </button>  <button class="star" v-if="item.isStar === true" @click="addStar(item)">  取消收藏  </button>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  posts: [  { title: "HTML", detail: "Something about HTML.", isStar: false },  {  title: "JavaScript",  detail: "Something about JavaScript.",  isStar: true,  },  { title: "CSS", detail: "Something about CSS.", isStar: false },  ],  });  const addStar = (item) => {  //do something  };  return {  ...toRefs(data),  addStar,  };  },  };  </script> |

**例3-19 帖子列表**



**图3-14 帖子列表的显示**

## 3.7 v-on事件绑定

### 3.7.1 v-on指令

v-on指令用于监听 DOM事件，并在触发事件时执行一些JavaScript，通常缩写为@符号。直接把JavaScript代码写在v-on指令中是一种方式。但是许多事件的处理过程十分复杂，因此v-on还可以接收一个需要调用的方法名称。用法为v-on:click="methodName" 或使用快捷方式@click="methodName"。

不难发现，事件绑定已经在之前的例子中出现过很多次。按钮的点击事件@click即等同于v-on:click。除了点击事件，v-on后面还可以接其他HTML的标准事件，如表3-2所示。

|  |  |
| --- | --- |
| **事件** | **说明** |
| click | 单击鼠标左键 |
| dblclick | 双击鼠标左键 |
| contextmenu | 单击鼠标右键 |
| mouseover | 指针移到有事件监听的元素或其子元素内 |
| mouseout | 指针移出元素，或者移到其子元素上 |
| keydown | 键盘动作：按下任意键 |
| keyup | 键盘动作：释放任意键 |

**表3-2 部分HTML的标准事件**

### 3.7.2事件修饰符

Vue为v-on提供了事件修饰符，修饰符是由点开头的指令后缀来表示的。使用多个修饰符时，顺序很重要。常见的修饰符见表3-3。

|  |  |  |
| --- | --- | --- |
| **名称** | **可用事件** | **说明** |
| .stop | 任意 | .当事件触发时，阻止事件冒泡 |
| .prevent | 任意 | 当事件触发时，阻止兀素默认行为 |
| .capture | 任意 | 当事件触发时，阻止事件捕获 |
| .self自身 | 任意 | 限制事件仅作用于节点自身 |
| .once | 任意 | 事件被触发一次后即解除监昕 |
| .passlve | 滚动 | 移动端，限制事件永不调用preventDefault ()方法 |

**表3-3 常见的事件修饰符**

Vue 允许为v-on或者@在监听键盘事件时添加按键修饰符，用于检查详细的按键。使用方法为：@keyup.enter=" methodName"。以下是全部的别名：

• .enter

• .tab

• .delete （“删除”和“退格”）

• .esc

• .space

• .up

• .down

• .left

• .right

除了键盘按键之外， Vue 也为鼠标按键配置了修饰符，包括.left、.right与.middle。

当需要按住多键或者鼠标与键盘共用以实现操作时, 可以使用组合修饰符，需要配合系统按键修饰使用，包括.ctrl、.alt、.shift与.meta。在Mac系统键盘上，meta对应command键。在Windows系统键盘上，meta对应Window键。

## 3.8指令在Vue 3.x中的变化

### 3.8.1 v-if与v-for的key

对于v-if/v-else/v-else-if的各分支项key属性不再是必须的，因为，即使没有为条件分支提供key，Vue 3.x也会自动生成唯一的key。因此，不建议在v-if/v-else/v-else-if的分支中继续使用key属性。如果主动提供key，那么每个分支必须使用唯一的key。故意使用相同的key也无法强制重用分支。

在Vue 3.x中，<template v-for>的key 应该设置在<template>标签上 ，而不是设置在它的子节点上。在Vue 2.x中，由于<template>标签不能拥有key，可以为其每个子节点分别设置key。同样，当使用<template v-for>时存在使用v-if的子节点，key应改为设置在<template>标签上。

### 3.8.2 v-if与v-for的优先级

在Vue 2.x版本中，在一个元素上同时使用v-if和v-for时，v-for会优先作用。在Vue 3.x版本中，v-if会拥有比v-for更高的优先级。

由于语法上存在歧义，建议避免在同一元素上同时使用两者。具体可以参考v-for章节中提到的方法。

### 3.8.3 v-bind合并行为

在元素上动态绑定attribute时，在一个元素中同时使用v-bind="object"语法和单独的property是一种常用的做法。这就涉及到了合并的优先级的问题。

在Vue 2.x，如果一个元素同时定义了v-bind="object" 和一个相同的单独的property，那么这个单独的property总是会覆盖object中的绑定。在Vue 3.x，声明绑定的顺序决定了它们如何合并。现在开发者对自己所希望的合并行为有了更好的控制。

如果希望依赖v-bind的覆盖功能，建议确保在单独的property之前定义v-bind属性。

### 3.8.4 v-for中的Ref数组

在Vue 2.x中，在v-for里使用的ref attribute会用ref数组填充相应的$refs property。但是，当存在嵌套的v-for时，会变得不明确且效率低下。

在 Vue 3 中，将不再在$ref中自动创建数组。要从单个绑定获取多个ref，可以将ref绑定到一个更灵活的函数上。如例3-20所示。

|  |
| --- |
| <template>  <div id="app">  <div v-for="item in list" :ref="setItemRef"></div>  </div>  </template>  <script>  import { ref, onBeforeUpdate, onUpdated } from 'vue'  export default {  setup() {  let itemRefs = []  const setItemRef = el => {  itemRefs.push(el)  }  onBeforeUpdate(() => {  itemRefs = []  })  onUpdated(() => {  console.log(itemRefs)  })  return {  itemRefs,  setItemRef  }  }  }  </script> |

**例3-20 ref绑定到函数**

值得注意的是，itemRefs不必是数组，它也可以是一个对象，其ref会通过迭代的 key被设置。itemRef也可以是响应式的且可以被监听。

### 3.8.5 v-model

在用于自定义组件时，v-model的prop和事件默认名称已更改。prop从value变更为modelValue，event从inpu变更为update:modelValue。v-bind的.sync修饰符和组件的model选项已移除，用v-model作为代替。现在可以在同一个组件上使用多个v-model进行双向绑定。

除了像.trim这样的Vue 2.x硬编码的v-model修饰符外，现在Vue 3.x还支持自定义v-model 修饰符。

迁移时，可以将所有使用.sync的部分替换为v-model。对于所有不带参数的v-model，确保分别将prop和event命名更改为modelValue和update:modelValue。

更多关于v-model与组件的更新将会在组件的章节进行说明。

## 3.9本章小结

本章介绍了插值绑定、计算属性、属性绑定、双向绑定、条件渲染、列表渲染、事件绑定以及Vue的内置指令从Vue 2.x到Vue 3.x的变化。

文本插值最基本的方法是使用双大括号（Mustache语法）“{{ }}”，Vue会将大括号里的内容替换为表达式值，以文本的形式将其展示出来。表达式可由JavaScript表达式和过滤器构成。通过任何方法修改数据设定值，大括号的内容都会被实时替换。Vue支持在“{{ }}”插值的尾部添加过滤器，用管道符“ |” 表示。经常用于格式化文本，

计算属性会在其依赖属性的值发生变化时，对属性的值进行自动更新，同时更新相关的DOM 部分。通过从vue中导入computed来使用计算属性。侦听属性watch用来监视指定数据项的变化，从而触发用户自定的操作。默认情况下，回调仅在侦听的数据源发生改变时调用。通过从vue中导入watch来使用侦听属性。

v-bind指令主要用于动态绑定DOM元素属性，可以将一个或多个attribute，或一个组件prop动态地绑定到表达式。

v-model指令用于在<input>、<textarea>及<select>等表单控件元素上创建双向绑定，它会根据控件类型自动选取正确的方法来更新元素。在修改表单元素值时，对应的实例中的属性值会被同时更新。

Vue根据表达式的值，在DOM中渲染或销毁元素与组件，称为条件渲染。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真值的时候被渲染。当v-if中的表达式返回值为假时，根据v-else-if的表达式值的真假进行渲染。当v-else-if与v-if的表达式值均为假时，渲染v-else中的内容。v-show用法与v-if大致一样。因为v-show只是切换元素CSS属性的display，所以v-show的元素始终会被渲染并保留在DOM中。

使用列表渲染指令v-for可以渲染一个列表，用于遍历一个数组或枚举一个对象循环显示。它的表达式需结合in来使用。

v-on指令用于监听 DOM事件，并在触发事件时执行一些JavaScript，通常缩写为@符号。

在指令方面，Vue 3.x的更新集中在v-if与v-for的key、v-if与v-for的优先级、v-bind合并行为、v-for中的Ref数组和v-model。

在下一章中，将会介绍Class与Style绑定的问题。

## 3.10练习题

* **问答题：**

1. v-if与v-for在同一元素上使用时优先级是怎样的？如何避免这种情况？
2. 在文本插值中，如何实现简单的条件判断？
3. 简要说明如何创建仅可读和可读可写的计算属性。
4. 分别说明几种修饰符如何控制v-model中数据同步的时机。
5. 列举v-if与v-show的异同。
6. 解释v-for中各参数以及key的作用。
7. 列举什么情况下对v-for数组与对象的更新不会导致视图的更新。

* **动手做：**

1. 尝试使用v-model、v-if与v-for的知识，制作一个简易贴吧。有普通用户和管理员两种角色，普通用户可以发帖、收藏帖子。管理员可以发帖、置顶帖子。

# 第4章 Class与Style绑定

在实际开发的过程中，DOM元素经常会动态地绑定一些class类名或style样式，本章将会对Vue 3中的绑定HTML Class与绑定内联样式进行介绍。在将v-bind用于class和style时，Vue做了专门的增强，本章也会展示使用v-bind指令来绑定class和style的多种方法。

## 4.1绑定HTML Class

### 4.1.1对象语法

通过传给v-bind:class一个对象，用于动态地切换class，可以将v-bind:class简写为:class。Vue也支持在对象中传入更多字段来动态切换多个class。:class指令也可以与普通的class attribute同时存在，如例4-1所示。

|  |
| --- |
| <template>  <div class="app">  <div  class="static"  :class="{ active: isActive, 'text-danger': hasError }"  ></div>  </div>  </template>  <script>  import { toRefs,reactive } from "vue";  export default {  setup() {  const data = reactive({  isActive: true,  hasError: false,  });  return {  ...toRefs(data),  };  },  };  </script> |

**例4-1 动态切换class**

上面的语法表示active这个class的存在与否取决于isActive的真假，text-danger的存在与否取决于hasError的真假。当isActive或者hasError变化时，class的值也将相应地更新。上面代码的渲染结果为：

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

当变量值为undefined、null、值为0的数字、空字符串时，会被判定为假。[]、{}、-1、-0.1 也会被判定为真。

除了这种形式，也可以直接绑定一个Object数据或绑定一个返回对象的计算属性。一般当条件多于两个时，往往会使用计算属性，是一种很便利和常见的用法，如例4-2所示。

|  |
| --- |
| <div :class="classObject"></div>  const data = reactive({  isActive: true,  hasError: false,  });  const classObject = computed(() => {  return {  active: data.isActive && !data.error,  "text-danger": data.error && data.error.type === "fatal",  };  }); |

**例4-2 通过计算属性切换class**

### 4.1.2数组语法

与对象语法类似，可以把一个数组传给:class，用来应用一个class列表。想根据条件切换列表中的class，也可以使用三元表达式。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。在例4-3中分别使用了普通数组、三元表达式与对象语法。

|  |
| --- |
| <template>  <div class="app">  <div :class="[activeClass, errorClass]"></div>  <div :class="[{ active: isActive }, hasError ? errorClass : '']"></div>  </template>  <script>  import { toRefs,reactive } from "vue";  export default {  setup() {  const data = reactive({  activeClass: "active",  errorClass: "text-danger",  });  return {  ...toRefs(data),  };  },  };  </script> |

**例4-3 通过计算属性切换class**

上面代码的渲染结果为：

<div class="active text-danger"></div>

与对象语法一样，数组语法也可以使用data 、computed 和methods三种方法。在开发过程中，如果表达式较长或逻辑复杂，应该尽可能优先使用计算属性给元素动态设置类名。

## 4.2绑定内联样式

### 4.2.1对象语法

使用v-bind:style可以给元素绑定内联样式，方法与:class 类似，也有对象语法和数组语法，类似直接在元素上写CSS，同样可以简写为:style。

:style的语法十分直观，十分接近CSS，但其实是一个JavaScript对象。其中CSS property名可以用驼峰式(camelCase)或短横线分隔(kebab-case)来命名。

由于直接绑定到一个样式对象会让模板更清晰，因此通常使用对象的方式。同样的，对象语法也常常结合返回对象的计算属性使用。

在例4-4中分别使用了两种绑定方式，结果如图4-1所示。

|  |
| --- |
| <template>  <div class="app">  <div :style="{ color: activeColor, fontSize: fontSize + 'px' }">string style</div>  <div :style="styleObject">object style</div>  </div>  </template>  <script>  import { toRefs, reactive } from "vue";  export default {  setup() {  const data = reactive({  activeColor: "red",  fontSize: 30,  styleObject: {  color: 'red',  fontSize: '13px'  }  });  return {  ...toRefs(data),  };  },  };  </script> |

**例4-4 对象绑定样式**



**图4-1 对象绑定样式结果**

### 4.2.2数组语法

:style的数组语法可以将多个样式对象应用到同一个元素上，如下所示。

<div :style="[styleA , styleB]"></div>

在实际开发中，:style的数组语法并不常用，因为往往可以写在一个对象里面，这样对代码的理解更为直观。较为常用的是计算属性。

在:style中使用特殊的CSS property时，例如transform，Vue将自动添加相应的前缀。

## 4.3本章小结

本章介绍了Class与Style绑定，包括绑定HTML Class与绑定内联样式。分别可以通过对象语法和数组语法实现对class与style的动态切换。对象语法与数组语法一样，都可以使用data 、computed 和methods三种方法，其中最为常用与便捷的是计算属性。

通过为表达式结果的类型增加对象与数组，避免了将v-bind用于class和style时表达式字符串拼接麻烦且易错的问题。

在下一章中，将会进一步介绍过滤器。

## 4.4练习题

* **问答题：**

1. 列举:style中CSS property名的命名方法。
2. 分别说明对象语法中，条件判断结果为真或假的值有那些情况。
3. 简要说明绑定class与style使用计算属性的优势有哪些。

* **动手做：**

1. 设计一个按钮，让它可以根据两种用户身份切换激活与否。
2. 修改上一章的练习，通过绑定内联样式，让被收藏的帖子标题显示为黄色、置顶帖子的标题显示为红色。

# 第5章 过滤器

从Vue 3.0开始，过滤器已删除，不再支持。建议用计算属性或方法代替过滤器，而不是使用过滤器。因此，本章将简单介绍Vue原有的内置过滤器，而不对过滤器做过多的深入。

在第3章插值绑定中提到过，Vue 2.x允许在表达式后面添加可选的过滤器，以管道符|表示。事实上，过滤器的本质是一个函数，Vue 2.x支持在任何出现表达式的地方添加过滤器。过滤器接受管道符前面的值作为初始值，返回值为经过处理后的输出值。同时，过滤器也能接受额外的参数，始终以表达式的值作为第一个参数，其他参数跟在过滤器名称后面，参数之间以空格分隔。多个过滤器也可以进行串联，上一个过滤器的输出结果可以作为下一个过滤器的输入。在本章中将进一步介绍过滤器。

## 5.1内置过滤器

### 5.1.1字母操作

Vue 2.x内置了三个字母过滤器capitalize、uppercase、lowercase用于处理英文字符。

capitalize过滤器用于将表达式中的首字母转换为大写。

uppercase过滤器用于将表达式中的所有字母转换为大写。

lowercase过滤器用于将表达式中的所有字母转换为小写。

三种过滤器的用法如例5-1所示。

|  |
| --- |
| （｛ ’ abcd ’ | capitalize }}  ｛｛ ’ abcd ’ | uppercase }}  ｛｛ ’ ABCD ’ | lowercase }} |

**例5-1 字母过滤器的使用**

上面三段代码将分别输出‘Abcd’、‘ABCD’和‘abcd’。

### 5.1.2 json过滤器

Vue 2.x的json过滤器可以将表达式的值转换为JSON字符串，输出的结果等于表达式经过JSON.stringify()处理后的结果。json过滤器可接受一个类型为Number的参数，用于决定转换后的JSON字符串的缩进距离，默认的缩进距离为2。使用方式如例5-2：

|  |
| --- |
| {{ student | json 4 }} |

**例5-2 json过滤器的使用**

### 5.1.3限制过滤器

Vue 2.x内置了三个限制过滤器limitBy、filterBy、orderBy用于处理井返回过滤后的数组。这三个过滤器所处理的表达式的值必须是数组。

limitBy过滤器的作用是限制数组为前N个元素，通过传入的第一个参数指定N。第二个参数可选，用于指定开始的偏移量，默认为0不偏移。

filterBy过滤器的第一个参数可以是字符串或者函数。如果第一个参数是字符串，那么将在每个数组元素中搜索它，返回包含该字符串的元素组成的数组。如果filterBy的第一个参数是函数，则过滤器将根据函数的返回结果进行过滤。此时filterBy过滤器将调用JavaScript组中内置的函数filter()对数组进行处理，待过滤数组中的每个元素都将作为参数输入并执行传入filterBy中的函数。只有函数返回结果为true的数组元素才符合条件并将存入一个新的数组，最终filterBy的返回结果即为这个新的数组。

在对象中，过滤器将在所有属性中搜索。可以通过指定搜索宇段的方式缩小搜索范围。在多个字段中搜索时，字段与字段之间通过空格分隔，或者将搜索字段存放在一个数组中。如例5-3所示。

|  |
| --- |
| <div v-for＝”item in items | limitBy 10 5”  ＞＜/div>  <div v-for＝”student in students | filterBy ’sky’ in ’name’ ’nickname’” ＞＜／div> |

**例5-3 limitBy与filterBy过滤器的使用**

此外，还可以使用动态参数作为搜索目标或搜索字段。结合v-model实现输入提示效果。通过将输入值作为检索字段，实现输入提示。如例5-4所示。

|  |
| --- |
| <template>  <div class="app">  <input v-model="input" />  <li v-for="user in users | filterBy input in 'name'">  { { user. name } }  </li>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  input: "",  users: [{ name: "SKY" }, { name: "GRASS" }, { name: "NANA" }],  });  return {  ...toRefs(data),  };  },  };  </script> |

**例5-4 动态提示**

orderBy过滤器的作用是返回排序后的数组。第一个参数可以是字符串、数组或者函数。第二个参数order可选，order<0为降序，其他为升序，默认为1升序排列。

若输入参数为字符串，则可同时传入多个字符串作为排序键，字符串之间以空格分隔，或者存入一个数组中。此时将按照传入键名的顺序或数组的顺序进行排序。如例5-5所示。

|  |
| --- |
| <li v-for=" user in users | orderBy ' lastName '' firstName '' age ' "></li>  <li v-for=" user in users | orderBy sortList"></li> |

**例5-5 orderBy过滤器的使用**

### 5.1.4 currency过滤器

currency过滤器的作用是将数字转换为货币形式输出。第一个参数为String类型的货币符号，默认为美元符号$。第二个参数为Number类型，表示保留的小数位，默认为2。

|  |
| --- |
| <div>{{ amount | currency}}</div>  <div>{{ amount | currency '￥' }}</div>  <div>{{ amount | currency '$' 3}}</div> |

**例5-6 currency过滤器的使用**

当amount为12345时，上面得代码将分别显示$12,345.00、￥12,345.00和$12,345.000

### 5.1.5 debounce过滤器

debounce过滤器的作用是延迟执行时间。接受的表达式的值必须为函数，一般与v-on等指令结合使用。debounce接受一个可选的参数作为延迟时间，单位为毫秒，默认为300毫秒。经过debounce包装的处理器在调用之后将至少延迟设定的时间再执行。如果在延迟结束前再次调用，则延迟时长将重置为设定的时间。使用方法如例5-7所示,实现对了按键事件处理的延迟。

|  |
| --- |
| <input @keyup= "onKeyup | debounce 500"> |

**例5-6** debounce**过滤器的使用**

## 5.2本章小结

本章介绍了Vue原有的内置过滤器，字母过滤器、json过滤器、限制过滤器、currency过滤器和debounce过滤器。它们在处理表达式时有很便利的效果，但是这种方式打破了大括号内表达式“只是JavaScript”的原则。因此从Vue 3.0开始，建议用计算属性或方法替换过滤器。

在下一章中，将会进一步介绍过渡与动画的效果。

## 5.3练习题

* **问答题：**

1. 说明三种字母过滤器的作用。
2. 如何使用限制过滤器限制数组为第3个元素开始的10个元素？
3. currency过滤器的第一个参数采取默认形式，想要修改第二个参数（也就是小数位）时，该如何表示？

* **动手做：**

1. 使用计算属性，实现三种字母过滤器的效果。
2. 使用计算属性，在第三章的所有帖子标题与内容中搜索包含“sky”的部分，并返回包含“sky”的元素组成的数组。

# 第6章 过渡与动画

过渡效果在在用户与网站交互的过程中提供了很好的体验。前面介绍的class和style声明就可以应用于动画和过渡，用于简单的用例。除此之外，Vue还内置了一套过渡系统用于帮助处理过渡和动画，可以在元素从DOM中插入或移除时触发CSS过渡或动画，自动应用过渡效果。Vue的过渡系统也支持JavaScript的过渡，使用钩子函数在过渡过程中执行自定义的DOM操作。

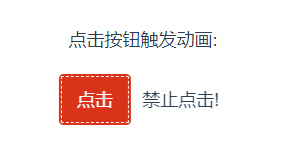
## 6.1过渡与动画概述

### 6.1.1基于class的动画和过渡

<transition> 组件对于组件的进入和离开非常有用，但也可以通过添加一个条件class来激活动画，而无需挂载组件。通过class来激活动画的案例如例6-1所示。

|  |
| --- |
| <template>  <div id="app">  点击按钮触发动画:<br />  <div :class="{ shake: noActivated }">  <button @click="noActivated = true">点击</button>  <span v-if="noActivated">禁止点击!</span>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  noActivated: false,  });  return {  ...toRefs(data),  };  },  };  </script>  <style>  body {  margin: 30px;  }  button {  background: #d93419;  border-radius: 4px;  display: inline-block;  border: none;  padding: 0.75rem 1rem;  margin: 20px 10px 0 0;  text-decoration: none;  color: #ffffff;  font-family: sans-serif;  font-size: 1rem;  cursor: pointer;  text-align: center;  -webkit-appearance: none;  -moz-appearance: none;  }  button:focus {  outline: 1px dashed #fff;  outline-offset: -3px;  }  .shake {  animation: shake 0.82s cubic-bezier(0.36, 0.07, 0.19, 0.97) both;  transform: translate3d(0, 0, 0);  backface-visibility: hidden;  perspective: 1000px;  }  @keyframes shake {  10%,  90% {  transform: translate3d(-1px, 0, 0);  }  20%,  80% {  transform: translate3d(2px, 0, 0);  }  30%,  50%,  70% {  transform: translate3d(-4px, 0, 0);  }  40%,  60% {  transform: translate3d(4px, 0, 0);  }  }  </style> |

**例6-1 通过class激活动画**



**图6-1 动画前后效果**

点击按钮后的前后效果如图6-1所示。通过点击按钮会触发事件，将*noActived*属性的值修改为true，此时class的shake属性被激活，触发在<style>或CSS文件中的shake样式，展现出动画效果。同时v-if条件渲染的判断为真，在视图中渲染出“禁止点击”的字样。

### 6.1.2基于style的动画和过渡

一些过渡效果可以通过插值的方式来实现，这会导致在发生交互时将样式绑定到元素上。使用插值来创建动画的案例如例6-2所示。

|  |
| --- |
| <template>  <div id="app">  <div  @mousemove="xCoordinate"  :style="{ backgroundColor: `hsl(${x}, 80%, 50%)` }"  class="movearea"  >  <h3>在组件中左右移动鼠标</h3>  <p>x: {{ x }}</p>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  x: 0,  });  const xCoordinate = (e) => {  data.x = e.offsetX;  };  return {  ...toRefs(data),  xCoordinate,  };  },  };  </script>  <style>  .movearea {  transition: 0.2s background-color ease;  height: 100px;  width: 40%;  position: absolute;  left: 30%;  color: orange;  }  </style> |

**例6-2 将样式绑定到元素上**









**图6-2 不同鼠标位置时的效果**

在这个例子中使用了插值来创建动画，并且将动画的触发条件添加到了鼠标的移动过程上。同时将CSS过渡属性应用在元素上，让元素知道在更新时要使用什么过渡效果。在图6-2中展示了鼠标在块中左右移动时，<div>的背景颜色会随着*x*坐标不断变化。在JavaScript中，使用了event.offsetX来获取鼠标在<div>中的*x*坐标值，并将*x*值的变化通过hsl映射到背景颜色的变化上。

## 6.2单元素的过渡

### 6.2.1进入与离开过渡

Vue 提供了transition的封装组件，在使用包括条件渲染 (v-if)、条件展示 (v-show)、动态组件与组件根节点时，可以给任何元素和组件添加过渡的进入和离开。过渡动画的触发时机包括元素或组件初始渲染时、元素或组件显示或隐藏时(包括条件渲染时)、元素或组件切换时。在例6-3中，展示了在条件渲染情况下，元素渐进渐出效果的过渡。

|  |
| --- |
| <template>  <div id="app">  <button @click="show = !show">切换</button>  <transition name="fade">  <p v-if="show">Hello</p>  </transition>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  show: true,  });  return {  ...toRefs(data),  };  },  };  </script>  <style>  .fade-enter-active,  .fade-leave-active {  transition: opacity 0.5s ease;  }  .fade-enter-from,  .fade-leave-to {  opacity: 0;  }  </style> |

**例6-3 渐进渐出的过渡**

每当点击按钮时，会切换一次Hello的显示或隐藏。通过在0.5秒的持续时间内，对Hello的透明度进行改变，以实现渐进渐出的效果。

当插入或删除包含在transition组件中的元素时，Vue将会自动判断目标元素是否应用了CSS过渡动画，并在恰当的时机添加或删除带有过渡动画的CSS类名。另外，如果过渡组件提供了JavaScript钩子函数，钩子函数也将在恰当的时机被调用。如果没有找到JavaScript钩子函数，并且也没有检测到CSS过渡动画，则会在浏览器逐帧动画机制的下一帧中立即执行插入或删除的DOM操作。

在例6-3中，使用了fade-enter-from与fade-leave-to等类名，这些类名用于定义不同过渡阶段的样式。在进入或离开的过渡中，总共有6个class可以切换。其中，在原来Vue 2.x中的转换类名\*-enter与\*-leave在Vue 3中被重命名为了\*-enter-from与\*-leave-from。

\*-enter-from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。

\*-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。

\*-enter-to定义进入过渡的结束状态。在元素被插入之后下一帧生效，即\*-enter-from被移除的同时，在过渡动画完成之后移除。

\*-leave-from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。

\*-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。

\*-leave-to：离开过渡的结束状态。在离开过渡被触发之后下一帧生效，即\*-leave-from被删除的同时，在过渡动画完成之后移除。

如果使用一个没有名字的<transition>，则v-是这些类名的默认前缀。如果使用了 <transition name="my-transition">，那么\*-enter-from会替换为my-transition-enter-from。

### 6.2.2 CSS过渡与动画

常用的过渡都是使用CSS过渡。除了直接在元素上添加transition="name" 外，Vue也支持动态绑定CSS名称，便于元素根据场景使用多个过渡效果。如例6-4所示，在data中指定transitionName的值后就可以实现动态绑定。

|  |
| --- |
| <div v-if="show" :transition="transitionName"></div> |

**例6-4 动态绑定过渡CSS名称**

CSS动画用法类似CSS过渡，区别是在动画中，\*-enter-from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。在例6-5中展示了使用CSS动画制作弹跳显示效果。

|  |
| --- |
| <template>  <div id="app">  <button @click="show = !show">切换</button>  <transition name="bounce">  <p v-if="show">弹跳显示</p>  </transition>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  show: true,  });  return {  ...toRefs(data),  };  },  };  </script>  <style>  .bounce-enter-active {  animation: bounce-in 0.5s;  }  .bounce-leave-active {  animation: bounce-in 0.5s reverse;  }  @keyframes bounce-in {  0% {  transform: scale(0);  }  50% {  transform: scale(1.25);  }  100% {  transform: scale(1);  }  }  </style> |

**例6-5 弹跳显示效果**

当点击切换按钮后，会首先放大再缩小“弹跳显示”的字样，直至消失。再次点击时，会反向完成之前的操作，实现弹跳显示的动画效果。

### 6.2.3自定义过渡class类名

在过渡的JavaScript中，可以通过enter-from-class、enter-active-class、enter-to-class、leave-from-class、leave-active-class、leave-to-class这几个attribute来自定义CSS过渡类名。他们的优先级高于普通的类名，会覆盖默认的类名。这样的方式有利于结合使用Vue的过渡系统和其他第三方CSS动画库。例如Animate.css是Vue官方推荐的CSS动画库，通过引入Animate.css的CSS文件就可以使用提供的预设动画。配合自定义过渡类名，可以达到非常不错的效果。

### 6.2.4 JavaScript过渡

类似CSS，同样可以在attribute中声明JavaScript钩子。只使用JavaScript过渡时，不需要定义任何CSS样式，但enter和leave钩子需要调用done函数，用来明确过渡结束时间，否则它们将被同步调用，导致过渡立即结束。推荐在只使用JavaScript钩子时，显式声明css: false, 此时Vue将跳过CSS检测，除了性能略高之外，还可以避免CSS规则干扰过渡。JavaScript钩子的使用如例6-6所示。

|  |
| --- |
| HTML  <transition  @before-enter="beforeEnter"  @enter="enter"  @after-enter="afterEnter"  @enter-cancelled="enterCancelled"  @before-leave="beforeLeave"  @leave="leave"  @after-leave="afterLeave"  @leave-cancelled="leaveCancelled"  :css="false"  >  </transition>  JS  beforeEnter(el) {  },  enter(el, done) {  done()  },  afterEnter(el) {  },  enterCancelled(el) {  },  beforeLeave(el) {  },  leave(el, done) {  done()  },  afterLeave(el) {  },  // leaveCancelled 只用于 v-show 中  leaveCancelled(el) {  } |

**例6-6 使用JavaScript钩子**

## 6.3其他过渡

### 6.3.1多元素过渡

最常见的使用多标签过渡的方法是通过一个列表和描述这个列表为空消息的元素，如例6-7所示。

|  |
| --- |
| <transition>  <table v-if="items.length > 0">  <!-- ... -->  </table>  <p v-else>没有元素</p>  </transition> |

**例6-7 多标签过渡**

除此之外，如果使用多个v-if或将单个元素绑定到一个动态property，也可以实现任意个元素之间的过渡，如例6-8所示。

|  |
| --- |
| HTML  <transition>  <button v-if="status === 'up'" key="up">  上升  </button>  <button v-if=" status === 'down'" key="down">  下降  </button>  <button v-if=" status === 'stay'" key="stay">  悬停  </button>  </transition>  //另一种写法  HTML  <transition>  <button :key="status">  {{ buttonMessage }}  </button>  </transition>  JS  computed: {  buttonMessage() {  switch (this.status) {  case 'up': return '上升'  case 'down': return '下降'  case 'stay': return '悬停'  }  }  } |

**例6-8 多v-if与动态property绑定**

### 6.3.2过渡模式

在例6-8中，如果使用v-if与v-else的条件触发过渡，当判断的数据值发生变化而触发过渡时，两个按钮都会被重绘。<transition>的默认行为会导致当一个按钮离开过渡时另一个按钮才开始进入过渡。

有时这很有效，但这样进入和离开同时生效的过渡不能满足所有要求，因此Vue提供了过渡模式。使用in-out表示新元素先进行过渡，完成之后当前元素过渡离开。out-in:表示当前元素先进行过渡，完成之后新元素过渡进入。使用过渡模式完成按钮平滑切换的实例如例6-9所示。

|  |
| --- |
| <template>  <div id="app">  <transition name="mode-fade" mode="out-in">  <button v-if="on" key="on" @click="on = false">开</button>  <button v-else key="off" @click="on = true">关</button>  </transition>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  on: false,  });  return {  ...toRefs(data),  };  },  };  </script>  <style>  body {  margin: 30px;  }  #demo {  position: relative;  }  button {  position: absolute;  }  .mode-fade-enter-active,  .mode-fade-leave-active {  transition: opacity 0.5s ease;  }  .mode-fade-enter-from,  .mode-fade-leave-to {  opacity: 0;  }  button {  background: #05ae7f;  border-radius: 4px;  display: inline-block;  border: none;  padding: 0.75rem 1rem;  text-decoration: none;  color: #ffffff;  font-family: sans-serif;  font-size: 1rem;  cursor: pointer;  text-align: center;  -webkit-appearance: none;  -moz-appearance: none;  }  </style> |

**例6-9 使用过渡模式更新按钮**

通过给按钮添加一个mode的attribute，就可以修改原来的过渡，而不必添加其他style，这十分有助于UI的微交互。

### 6.3.3列表过渡

在上一节已经介绍了通过v-if与v-show实现单元素的过渡，以及同一时间渲染多个节点中的一个，但是并没有提到使用v-for同时为列表元素添加过渡效果。事实上，<transition>与v-for并不兼容。在这种情况下，Vue提供了<transition-group>用以实现列表过渡。

与<transition>不同的是，<transition-group>会以一个真实元素渲染，默认为<span>，可以通过tag属性更换为其他元素。同时，过渡模式并不可用，内部元素必须提供唯一的key attribute值。此外，CSS过渡的类将会应用在内部的元素中，而不是这个组本身。

在例6-10中，使用<transition-group>实现了列表过渡，点击添加或移除按钮会在随机位置随机添加或删除一个数组元素。代码运行的效果如图6-3所示。

|  |
| --- |
| <template>  <div id="app">  <button @click="add">添加</button>  <button @click="remove">移除</button>  <transition-group name="list" tag="p">  <span v-for="item in items" :key="item" class="list-item">  {{ item }}  </span>  </transition-group>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  items: [1, 2, 3, 4, 5, 6, 7, 8, 9],  nextNum: 10,  });  const randomIndex = () => {  return Math.floor(Math.random() \* data.items.length);  };  const add = () => {  data.items.splice(randomIndex(), 0, data.nextNum++);  };  const remove = () => {  data.items.splice(randomIndex(), 1);  };  return {  ...toRefs(data),  randomIndex,  add,  remove,  };  },  };  </script>  <style>  .list-item {  display: inline-block;  margin-right: 10px;  }  .list-enter-active,  .list-leave-active {  transition: all 1s ease;  }  .list-enter-from,  .list-leave-to {  opacity: 0;  transform: translateY(30px);  }  button {  background: #05ae7f;  border-radius: 4px;  border: none;  padding: 0.75rem 1rem;  margin: 1rem;  color: #ffffff;  font-size: 1rem;  }  </style> |

**例6-10 列表过渡**



**图6-3 列表过渡的显示结果**

在图6-3中，第一个幅为初始状态，包含元素1到9。第二幅为在随机位置添加10、11、12的结果。第三幅为从随机位置删除3个元素的结果。

值得注意的是，由于过渡模式不可用，在添加元素时会表现为首先空出元素的位置，随后元素从下方渐入列表中，而删除时则表现为元素渐出列表后，列表合并空出的位置。因此，看上去添加或删除的元素周围的元素会瞬间移动到新布局的位置，而不是平滑的过渡，为了解决这个问题，就需要引入列表的排序过渡。

除了用以实现进出动画之外，<transition-group>组件还有一个特殊的可以用于改变元素定位的动画。要使用这个新功能需要了解新增的 \*-move 特性，它会在元素改变定位的过程中应用，达到平滑地移动列表元素位置的效果。\*-move动画效果的定义方式与\*-enter-from、\*-leave-from等类名一致，可以通过name属性来自定义前缀，也可以通过move-class属性手动设置。

### 6.3.4列表过渡案例：打乱列表

在上一节已经提到了例6-10中的过渡不自然问题的解决方法，这里可以尝试制作一个带有添加、删除以及打乱功能的列表，这个列表将会有平滑的渐入渐出效果。具体的代码如例6-11所示。

|  |
| --- |
| <template>  <div id="app">  <button @click="shuffle">打乱</button>  <button @click="add">添加</button>  <button @click="remove">移除</button>  <transition-group name="complete-list" tag="p">  <span v-for="item in items" :key="item" class="complete-list-item">  {{ item }}  </span>  </transition-group>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  import \_ from "lodash";  export default {  setup() {  const data = reactive({  items: [1, 2, 3, 4, 5, 6, 7, 8, 9],  nextNum: 10,  });  const randomIndex = () => {  return Math.floor(Math.random() \* data.items.length);  };  const add = () => {  data.items.splice(randomIndex(), 0, data.nextNum++);  };  const remove = () => {  data.items.splice(randomIndex(), 1);  };  const shuffle = () => {  data.items = \_.shuffle(data.items);  };  return {  ...toRefs(data),  randomIndex,  add,  remove,  shuffle,  };  },  };  </script>  <style>  body {  margin: 30px;  }  .complete-list-item {  transition: all 0.8s ease;  display: inline-block;  margin-right: 10px;  }  .complete-list-enter-from,  .complete-list-leave-to {  opacity: 0;  transform: translateY(30px);  }  .complete-list-leave-active {  position: absolute;  }  button {  background: #05ae7f;  border-radius: 4px;  border: none;  padding: 0.75rem 1rem;  margin: 1rem;  color: #ffffff;  font-size: 1rem;  }  </style> |

**例6-11 列表的增删与打乱**



**图6-4 列表的初始状态**



**图6-5 列表的增删3个元素**



**图6-6 列表的打乱**

如图6-4、图6-5与图6-6所示，分别展示了列表的初始状态、列表在随机位置增加三个元素、列表在随机位置删除三个元素以及打乱列表顺序的状态，过渡动画都达到了渐入渐出的效果。值得注意的是，此处引入了Lodash工具库，用于对列表进行打乱操作。可以通过以下的代码安装lodash库：

npm install --save lodash

npm install --save-dev babel-plugin-lodash

## 6.4本章小结

本章介绍了过渡效果，它在用户与网站交互的过程中提供了很好的体验。Vue内置的过渡系统可以帮助自动处理过渡和动画，支持CSS过渡和JavaScript过渡。

<transition> 组件十分常用于过渡中，可以通过添加一个条件class来激活动画，而无需挂载组件。插值同样也可以应用于过渡动画。通过转换类名、CSS动态绑定、条件渲染等可以控制过渡的进入和离开。使用<transition-group>可以实现列表过渡的效果。

在下一章中，将会进一步介绍组件并补充组件在Class与Style绑定、表单等方面的内容。

## 6.5练习题

* **问答题：**

1. 过渡动画的触发时机包括哪些？
2. 简述6种转换类名的作用分别是什么。
3. 使用JavaScript过渡时，如何避免CSS规则干扰？
4. <transition-group>是否可以使用过渡模式？它如何改变元素定位？

* **动手做：**

1. 为折叠卡片制作过渡动画：点击按钮替换卡片的图片，通过在水平方向上缩放两张图片实现替换时的过渡。

# 第7章 组件

组件是Vue最核心的功能之一，也是最难掌握的，它支持自定义tag和原生HTML元素的扩展。组件的核心目标是为了可重用性高，减少重复性的开发。在Webpack项目中，每一个页面文件.vue都可以认为是一个组件。

本章将介绍组件有关的内容，并通过几个实例练习如何使用Vue组件。

## 7.1组件的注册

### 7.1.1全局注册

Vue的组件可以预定义很多选项，最核心的是以下几个：模板template声明了数据与DOM之间的映射关系。初始数据data一个组件的初始数据状态。接受的外部参数props用于组件间的数据传递和共享，默认是单向绑定，但也可以显式声明为双向绑定。方法methods是对数据的改动操作。生命周期钩子函数lifecycle hooks一个组件会触发多个，在钩子函数中，可以封装一些自定义的逻辑。

全局注册需要确保在根实例初始化之前注册，这样才能使组件在任意实例中被使用，注册方式如例7-1所示。

|  |
| --- |
| // 注册一个名为my-component的组件  app.component('my-component', {  /\* ... \*/  })  // 检索注册的组件(始终返回构造函数)  const MyComponent = app.component('my-component', {}) |

**例7-1 组件的全局注册**

组件的全局注册可以传入两个参数，第一个参数是组件的名称，对于组件的命名，W3C规范是字母全小写且必须包含一个连字符“-”，用于避免和当前以及未来的HTML元素相冲突。，Vue不强制要求，但官方建议遵循这个规则。第二个参数是组件的构造函数definition，可以是Function，也可以是Object。如果传入构造函数参数，返回应用实例。如果不传入，返回组件定义。

组件是可复用的，且带有一个名字。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。

组件在注册之后，可以在父实例的模块中以自定义元素的形式使用。要确保在初始化根实例之前注册了组件，组件的注册与使用代码如例7-2所示。

|  |
| --- |
| HTML  <div id="components-demo">  <my-component></my-component>  </div>  JS  const app = Vue.createApp({});  app.component("my-component", {  data() {  return {  count: 0,  };  },  template: `  <button v-on:click="count++">  You clicked me {{ count }} times.  </>  `,  });  app.mount("#components-demo"); |

**例7-2 组件的全局注册与使用**



**图7-1 按钮组件的效果**

代码运行的效果如图7-1左侧所示，点击两次后数字变为2，如右侧所示。渲染后的结果如例7-3所示。

|  |
| --- |
| <div id="components-demo">  <button v-on:click="count++">  You clicked me {{ count }} times.  </>  </div> |

**例7-3 渲染后的结果**

### 7.1.2局部注册

全局注册往往是不够理想的，因为全局注册所有的组件意味着即便已经不再使用一个组件，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的无谓增加。局部注册则限定了组件只能在被注册的组件中使用，而无法在其他组件中使用，可以通过一个普通的JavaScript对象来定义局部组件。组件中也可以使用components选项注册组件，进行组件嵌套。局部组件的注册方式如例7-4所示。

|  |
| --- |
| JS  const ComponentA = {  /\* ... \*/  }  const ComponentB = {  /\* ... \*/  }  const app = Vue.createApp({  components: {  'component-a': ComponentA,  'component-b': ComponentB  }  })  //另一种写法  import ComponentA from './ComponentA.vue'  export default {  components: {  ComponentA  }} |

**例7-3 局部组件的注册**

对于components对象中的每个property来说，property名就是自定义元素的名字，property值就是组件的选项对象。局部注册的组件在其子组件中不可用。希望ComponentA在ComponentB中使用的代码如例7-4所示。

|  |
| --- |
| JS  const ComponentA = {  }  const ComponentB = {  components: {  'component-a': ComponentA  }} |

**例7-4 在组件中使用组件**

Vue组件的模板在某些情况下会受到HTML限制，例如＜table＞内只允许是<tr>、＜td＞ 、<th＞等表格元素，所以在＜table＞内直接使用组件是无效的。常见的限制元素还有＜ul＞、＜ol＞、＜select>。这种情况下，可以使用is属性来挂载组件，如例7-5所示。tbody在渲染时会被替换为组件的内容。

|  |
| --- |
| HTML  <table>  <tbody is=”my-component” ></tbody>  </table> |

**例7-5 使用is属性挂载组件**

## 7.2组件的数据传递

### 7.2.1 props参数

props是组件中非常重要的一个选项，起到了父子组件间桥梁的作用。

由于组件实例的作用域是独立的，因此子组件中无法直接调用父组件的数据，需要通过props将父组件的数据传递给子组件，子组件显式声明props以接收数据，子组件接收到后就可以根据参数的不同来渲染不同的内容或执行操作。

props的值可以有两种，一种是字符串数组，一种是对象，如例7-6所示。

|  |
| --- |
| JS  //数组形式  props: ['title', 'likes', 'isPublished', 'commentIds', 'author']  //对象形式  props: {  title: [String , Number],  likes: {type: Number, default: 0},  isPublished: Boolean,  commentIds: Array,  author: {Object, required: true},  } |

**例7-6 props参数的两种形式**

对于对象类型的props，Vue会根据验证要求检查组件，保证参数的正确使用。验证的type 类型可以是：String、Number、Boolean、Object、Array、Function。type也可以是一个自定义构造器，使用instanceof检测。

props中的数据与组件data函数的数据的主要区别就是props来自父级，而data中

是组件自己的数据，作用域是组件本身。这两种数据都可以在模板、计算属性和方法中使用。

一个组件默认可以拥有任意数量的props，任何值都可以传递给任何props。一个props被注册之后，就可以把数据作为一个自定义attribute传递进去。可以给props传入一个静态的值，也可以通过v-bind或简写:动态赋予一个变量的值或者一个复杂表达式的值，任何类型的值都可以传给props，如例7-7所示。

|  |
| --- |
| HTML  <my-post title="My journey with Vue"></my-post>  <my-post :title="post.title"></my-post>  <my-post :title="post.title + ' by ' + post.author "></my-post>  JS  app.component('my-post', {  props: ['title'],  template: `<h4>{{ title }}</h4>`  }) |

**例7-7 给props传入值**

每次父级组件发生变更时，子组件中所有的props都将会刷新。因此不应该在一个子组件内部改变props。当子组件希望使用本地的props数据时，最好定义一个本地的data property并传入props的值。当props需要进行转换时，最好使用props的值来定义一个计算属性。如果使用修饰符.sync，在子组件中修改值将会影响父组件的值。修饰符.once 意味着单次绑定，子组件接受一次父组件传递的数据后，单独维护这份数据，既不影响父组件数据也不受其影响而更新。

### 7.2.2组件通信

尽管子组件可以用this. $parent访问它的父组件， 父组件同样有一个数组this.$ children，包含它所有的子组件，根实例的后代可以用this.$root访问根实例，不过子组件应当避免直接依赖父组件的数据，尽量显式地使用props传递数据。在子组件中修改父组件的状态同样是非常糟糕的做法，因为这会导致父组件与子组件高耦合。同时只看父组件将会很难理解父组件的状态，因为它可能被任意子组件修改。在理想情况下，只有组件自己能修改自己的状态。

因此当子组件需要向父组件传递数据时，就要用到自定义事件。如果要通知整个事件系统，就要向下广播。每个Vue 实例都是一个事件触发器。$on()是监昕事件；$emit()是把事件沿着作用域链向上派送；$dispatch() 是派发事件，事件沿着父链冒泡；$broadcast()是广播事件，事件向下传导给所有的后代。

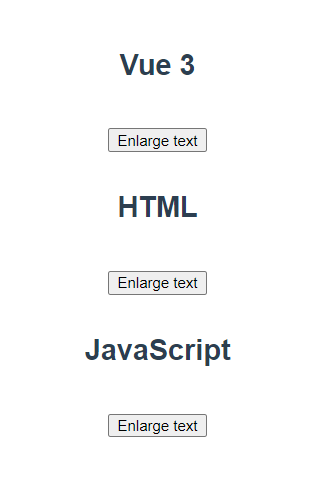
父级组件可以通过v-on或@监听子组件实例的任意事件，同时子组件可以通过调用内建的$emit方法并传入事件名称来触发一个事件，通过子组件中按钮修改父组件字体大小的案例如例7-8与例7-9所示。

|  |
| --- |
| <template>  <div id="app" class="demo">  <div :style="{ fontSize: postFontSize + 'em' }">  <blog-post  v-for="post in posts"  :key="post.id"  :title="post.title"  @enlarge-text="postFontSize += 0.1"  >  </blog-post>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  import BlogPost from './BlogPost.vue'  export default {  components: {  BlogPost,  },  setup() {  const data = reactive({  posts: [  { id: 1, title: "Vue 3" },  { id: 2, title: "HTML" },  { id: 3, title: "JavaScript" },  ],  postFontSize: 1,  });  return {  ...toRefs(data),  };  },  };  </script> |

**例7-8 父组件插入子组件**

|  |
| --- |
| <template>  <div class="blog-post">  <h4>{{ title }}</h4>  <button @click="$emit('enlargeText')">Enlarge text</button>  </div>  </template>  <script>  export default {  props: ["title"],  };  </script> |

**例7-9 blog-post子组件**



**图7-2 加大字号前后效果**

点击子组件中的加大字号按钮后的效果如图7-2右侧所示。经过 @enlarge-text监听器，父级将接收事件并更新postFontSize值，达到增大字号的效果。而$emit()方法的第一个参数是自定义事件的名称，后面的参数可选，表示要传递的数据。带数据的$emit()的使用方法如例7-10所示。

|  |
| --- |
| <!—- 子组件 -->  <button @click="$emit('enlarge-text', 0.1)">  Enlarge text  </button>  <!—- 父组件 -->  <blog-post ... @enlarge-text="postFontSize += $event"></blog-post> |

**例7-10 包含数据的$emit()方法**

当在父级组件监听事件时，可以通过$event访问到$emit()方法的第二个参数值。如果事件的处理函数是一个方法，那么这个值将会作为第一个参数传入这个方法。

除了用v-on在组件上监听自定义事件外，也可以监昕DOM事件，这时可以用.native 修饰符监听一个原生事件，监听的是该组件的根元素。

类似于prop类型验证，如果使用对象语法定义发出的事件，也可以进行验证。通过为事件分配一个函数以添加验证，函数将接收传递给$emit调用的参数，并返回一个布尔值表示事件是否有效。事件验证的案例如例7-11所示。

|  |
| --- |
| JS  app.component('my-component', {  emits: {  // 没有验证  click: null,  // 验证submit 事件  submit: ({ email, password }) => {  if (email && password) {  return true  } else {  console.warn('Invalid submit event payload!')  return false  }  }  },  methods: {  submitForm() {  this.$emit('submit', { email, password })  }  }  }) |

**例7-11 事件验证**

### 7.2.3 v-model参数

自定义事件也可以用于创建支持v-model的自定义输入组件。当用在组件上时，v-model的用法如例7-12所示。

|  |
| --- |
| HTML  <my-input v-model="search"></my-input>  JS  app.component('my-input', {  props: ['modelValue'],  template: `  <input  :value="modelValue"  @input="$emit('update:modelValue', $event.target.value)"  >  `  }) |

**例7-12 v-model自定义输入组件**

以上是一种默认情况，组件上的v-model使用modelValue作为prop、update:modelValue作为事件。除了这样，还可以通过向v-model传递参数来修改这些名称，如例7-13所示。

|  |
| --- |
| HTML  <my-input v-model:sss="search"></my-input>  JS  app.component('my-input', {  props: {  sss: String  },  template: `  <input  type="text"  :value="sss"  @input="$emit('update:sss', $event.target.value)"  >  `  }) |

**例7-13 修改props与事件名称**

子组件将需要一个sss prop并发出update:sss要同步的事件。

正如在v-model参数中，同样可以在单个组件实例上创建多个v-model绑定。每个v-model将自动同步到不同的props，而不需要在组件中添加额外的选项。

### 7.2.4 Vue 3中的v-model修饰符

在Vue 2.x中，支持了组件v-model上的.trim等修饰符。在Vue 3中，添加到组件v-model的修饰符将通过modelModifiers prop提供给组件，便于更好地支持自定义修饰符。

v-model本身有内置修饰符.trim、.number和.lazy。现在还支持添加自定义的修饰符。在例7-14中，创建了一个自定义修饰符capitalize，用于将v-model绑定的字符串的首字母大写。首先创建了一个组件，它包含了modelModifiers，默认为空对象。

|  |
| --- |
| HTML  <div id="app">  <my-component v-model.capitalize="myText"></my-component>  {{ myText }}  </div>  JS  const app = Vue.createApp({  data() {  return {  myText: ''  }  }  })  app.component('my-component', {  props: {  modelValue: String,  modelModifiers: {  default: () => ({})  }  },  methods: {  emitValue(e) {  let value = e.target.value  if (this.modelModifiers.capitalize) {  value = value.charAt(0).toUpperCase() + value.slice(1)  }  this.$emit('update:modelValue', value)  }  },  template: `<input  type="text"  :value="modelValue"  @input="emitValue">`  })  app.mount('#app') |

**例7-14 创建自定义修饰符capitalize**

当组件的created生命周期钩子触发时，modelModifiers包含capitalize，值为 true，因为它被设置在v-model绑定v-model.capitalize="bar"。每当<input/>元素触发input事件时，都会将字符串大写。

对于带参数的v-model绑定，生成的prop名称将为arg + "Modifiers"，如例7-15所示。

|  |
| --- |
| HTML  <my-component v-model:sss.capitalize="bar"></my-component>  JS  app.component('my-component', {  props: ['sss', 'sssModifiers'],  template: `  <input type="text"  :value="sss"  @input="$emit('update:sss', $event.target.value)">  `,  created() {  console.log(this.sssModifiers) // { capitalize: true }  }  }) |

**例7-15 带参数的v-model绑定**

## 7.3 slot内容分发

### 7.3.1 slot的基本用法

类似HTML元素，为了让组件可以组合，也需要一种方式来混合父组件的内容与子组件的模板，这个处理称为内容分发transclusion。Vue实现了内容分发，使用特殊的<slot>元素作为原始内容的插槽，让这变得非常简单。<slot>的基本用法如例7-16所示。

|  |
| --- |
| <!-- 父组件 -->  <my-component>  <i class="fas fa-plus"></i>  <font-awesome-icon name="plus"></font-awesome-icon>  Add something  </my-component>  <!-- 子组件模板 -->  <button class="btn-primary">  <slot></slot>  </button>  <!—- 渲染结果 -->  <button class="btn-primary">  <i class="fas fa-plus"></i>  <font-awesome-icon name="plus"></font-awesome-icon>  Add something  </button> |

**例7-16 slot的基本用法**

插槽可以包含任何模板代码或其他组件，包括 HTML。当组件渲染的时候，<slot>将会被替换。相反，如果子组件的模板中没有包含<slot>元素，则组件起始标签和结束标签之间的任何内容都会被抛弃。

至此，props参数、events事件和slot内容分发构成了Vue组件的3 个API，组件都由这3 部分构成。

### 7.3.2 slot的作用域

插槽可以访问与模板其余部分相同的实例property，也就是相同的“作用域”，但是不能访问所在子组件的作用域。例7-17中的*action*将不能被访问到。

|  |
| --- |
| <!—- 渲染结果 -->  <my-component action="delete">  Clicking here will {{ action }} a number  </my-component> |

**例7-17 slot的作用域**

因此，在此处想要使用*action*则需要在父组件中进行绑定。slot分发的内容，作用域是在父组件上的。

同样，父组件模板里的所有内容都是在父级作用域中编译的，子组件模板里的所有内容都是在子作用域中编译的。

### 7.3.3 slot的后备内容

插槽可以设置后备内容，作为默认的内容。后备内容只会在没有提供内容的时候被渲染。在例7-18中，为按钮设置了一个常见的submit后备。

|  |
| --- |
| <!-- 子组件模板 -->  <button type="submit">  <slot>Submit</slot>  </button>  <!-- 无插槽内容的父组件 -->  <submit-button></submit-button>  <!-- 有插槽内容的父组件 -->  <submit-button> Save </submit-button>  <!-- 渲染结果1 -->  <button type="submit"> Submit </button>  <!-- 渲染结果2 -->  <button type="submit"> Save </button> |

**例7-18 slot的后备内容**

### 7.3.3具名插槽

给<slot>元素指定一个name属性后可以分发多个内容，有name属性的<slot>称为具名插槽。一个不带name属性的匿名<slot>默认带有隐含的名字“default”。具名<slot>将匹配内容片段中有对应<slot>属性的元素。具名<slot>可以与匿名<slot>共存。如果没有匿名slot，找不到匹配插槽的内容片段将被忽略。

在向具名插槽提供内容的时候，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：

具名插槽的使用如例7-19所示。

|  |
| --- |
| <!-- 子组件模板 -->  <template>  <div class="container">  <div style="background-color:red">  <slot name="header"></slot>  </div>  <div style="background-color:yellow">  <slot></slot>  </div>  <div style="background-color:pink">  <slot name="footer"></slot>  </div>  </div>  </template>  <!-- 父组件 -->  <template>  <div id="app" class="demo">  <child-comp>  <template v-slot:footer>  <p>Here's some contact info</p>  </template>  <template v-slot:default>  <p>A paragraph for the main content.</p>  <p>And another one.</p>  </template>  <template v-slot:header>  <h1>Here might be a page title</h1>  </template>  </child-comp>  </div>  </template>  <script>  import ChildComp from "./childComp.vue";  export default {  components: {  ChildComp,  },  };  </script> |

**例7-18 具名插槽的使用**



**图7-3 具名插槽的渲染结果**

插槽渲染后的结果如图7-3所示，红色背景部分的slot被替换为了name为header的匹配内容。黄色背景部分的匿名slot被替换为了name为default的匹配内容，父组件没有指定slot特性的元素与内容都将替换这个slot。粉色背景部分的slot被替换为了name为footer的匹配内容。

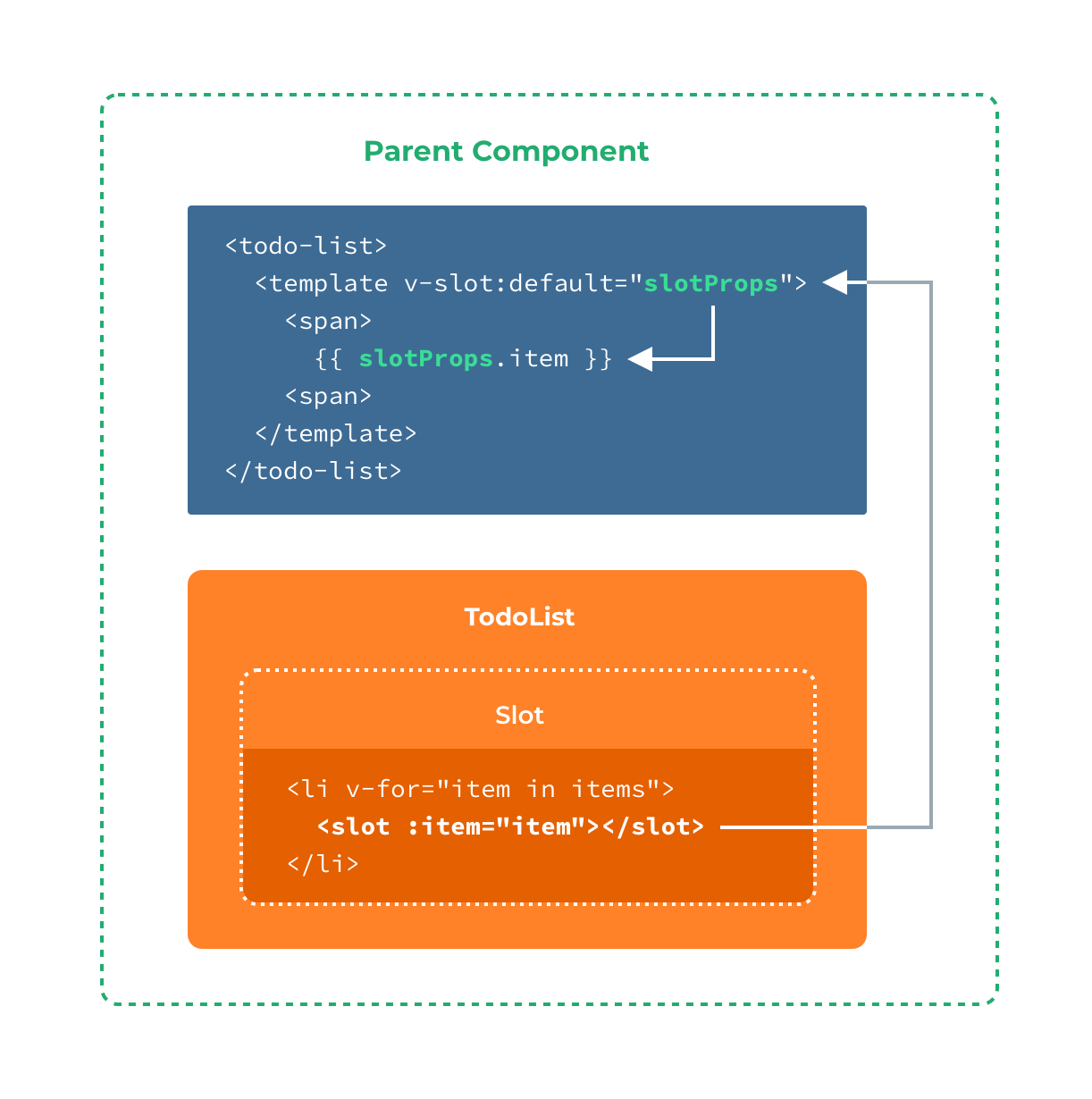
类似v-on和v-bind，v-slot也有缩写，通过把参数之前的所有内容，即v-slot:，替换为字符#。例如，上述的v-slot:header可以被重写为 #header：。需要注意的是，这种缩写只在有参数的时候才可用。如果希望使用缩写的话，就必须始终以明确插槽名代替。

### 7.3.4作用域插槽

前面提到了，插槽访问的是父组件的作用域，但是让插槽内容能够访问子组件中才有的数据的需求也十分常见。例如，当一个组件被用来渲染一个项目数组时，常常希望能够自定义每个项目的渲染方式。这就需要使用作用域插槽。如例7-19所示。

|  |
| --- |
| <!-- 子组件 -->  HTML  <template>  <ul>  <li v-for="( item, index ) in items">  <slot :item="item"></slot>  </li>  </ul>  </template>  JS  const data = reactive({  items: ['Feed a cat', 'Buy milk']  });  <!-- 父组件 -->  <todo-list>  <template v-slot:default="slotProps">  <span class="green">{{ slotProps.item }}</span>  </template>  </todo-list> |

**例7-19 作用域插槽的使用**



**图7-4 作用域插槽中数据的获取关系**

因为在原本的情况下只有<todo-list>组件可以访问item，如果要使item可用于父级提供的slot内容，可以添加一个<slot>元素并将其绑定为属性，绑定在<slot>元素上的属性被称为插槽prop。此时，在父级作用域中，可以使用带值的v-slot来定义所提供的插槽prop的名字。

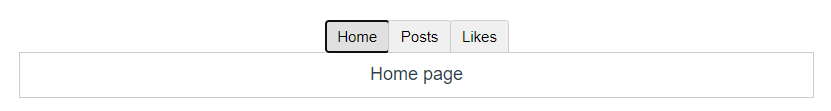
## 7.4 动态组件

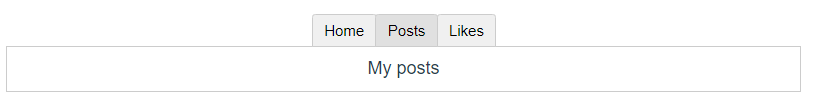
### 7.4.1基础用法

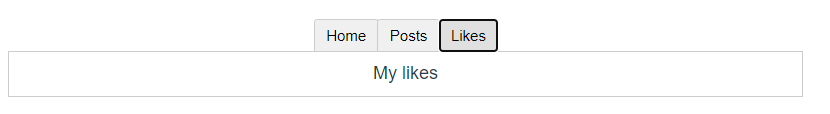
Vue支持动态组件，可以多个组件使用同一挂载点，根据条件动态地切换不同的组件。使用标签<component>，通过绑定到is属性的值来判断挂载哪个组件。基础用法如例7-20所示。

|  |
| --- |
| <!-- 父组件 -->  HTML  <template>  <div id="app" class="demo">  <button  v-for="tab in tabs"  v-bind:key="tab"  v-bind:class="['tab-button', { active: currentTab === tab }]"  v-on:click="currentTab = tab"  >  {{ tab }}  </button>  <component v-bind:is="currentTabComponent" class="tab"></component>  </div>  </template>  <script>  import { computed,reactive, toRefs } from "vue";  import TabPosts from "./tabPosts.vue";  import TabHome from "./tabHome.vue";  import TabLikes from "./tabLikes.vue";  export default {  components: {  TabPosts,  TabHome,  TabLikes,  },  setup() {  const data = reactive({  currentTab: "Home",  tabs: ["Home", "Posts", "Likes"],  });  const currentTabComponent = computed(() => {  return "tab-" + data.currentTab.toLowerCase();  });  return {  ...toRefs(data),  currentTabComponent,  };  },  };  </script>  <style>  .demo {  font-family: sans-serif;  border: 1px solid #eee;  border-radius: 2px;  padding: 20px 30px;  margin-top: 1em;  margin-bottom: 40px;  user-select: none;  overflow-x: auto;  }  .tab-button {  padding: 6px 10px;  border-top-left-radius: 3px;  border-top-right-radius: 3px;  border: 1px solid #ccc;  cursor: pointer;  background: #f0f0f0;  margin-bottom: -1px;  margin-right: -1px;  }  .tab-button:hover {  background: #e0e0e0;  }  .tab-button.active {  background: #e0e0e0;  }  .demo-tab {  border: 1px solid #ccc;  padding: 10px;  }  </style>  <!-- home组件 -->  <template>  <div class="demo-tab">Home page</div>  </template>  <!-- posts组件 -->  <template>  <div class="demo-tab">My posts</div>  </template>  <!-- likes组件 -->  <template>  <div class="demo-tab">My likes</div>  </template> |

**例7-20 动态组件的用法**







**图7-5 动态组件的效果**

通过给Vue的<component>元素加一个is属性来实现动态切换。:is是v-bind:is 的缩写，可以包括已注册组件的名字或一个组件的选项对象。在例7-20中，绑定了父组件中的计算属性currentTabComponent， 需要注意的是，currentTab属性的值需要和父组件实例中的components属性的key相对应。

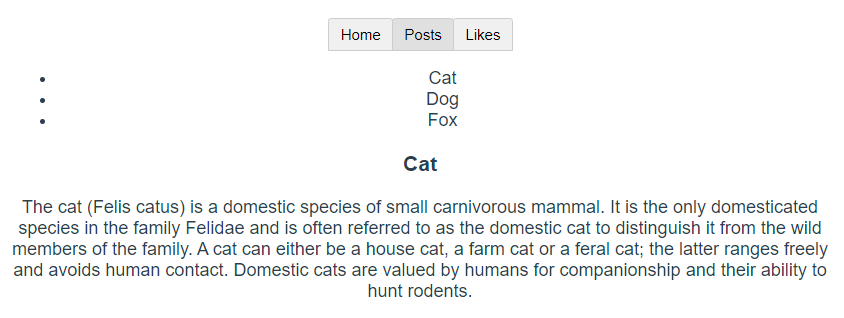
### 7.4.2 keep-alive

component标签可以添加keep-alive属性，用于将切换出去的组件保留在内存中，避免重新渲染。可以使用它缓存一些非动态的组件实例，以保留组件状态。keep-alive应出现在组件被移除之后需要再次挂载的地方。现在，来完善一下例7-20中的posts组件，显示一些博文内容，修改后的posts组件如例7-21所示。

|  |
| --- |
| <template>  <div class="demo-posts-tab">  <ul class="demo-posts-sidebar">  <li  v-for="post in posts"  :key="post.id"  :class="{  'demo-active': post === selectedPost,  }"  @click="selectedPost = post"  >  {{ post.title }}  </li>  </ul>  <div class="demo-post-container">  <div v-if="selectedPost" class="demo-post">  <h3>{{ selectedPost.title }}</h3>  <div v-html="selectedPost.content"></div>  </div>  <strong v-else> Click on a blog title to the left to view it. </strong>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  posts: [  {  id: 1,  title: "Cat",  content:  "<p>The cat (Felis catus) is a domestic species of small carnivorous mammal. It is the only domesticated species in the family Felidae and is often referred to as the domestic cat to distinguish it from the wild members of the family. A cat can either be a house cat, a farm cat or a feral cat; the latter ranges freely and avoids human contact. Domestic cats are valued by humans for companionship and their ability to hunt rodents.</p>",  },  {  id: 2,  title: "Dog",  content:  "<p>The domestic dog (Canis familiaris when considered a separate species or Canis lupus familiaris when considered a subspecies of the wolf) is a wolf-like canid that can be found distributed around the world. The dog descended from an ancient, now-extinct wolf with the modern wolf being the dog's nearest living relative. The dog was the first species to be domesticated by hunter–gatherers more than 15,000 years ago, which predates agriculture.</p>",  },  {  id: 3,  title: "Fox",  content:  "<p>Foxes are small to medium-sized, omnivorous mammals belonging to several genera of the family Canidae. Foxes have a flattened skull, upright triangular ears, a pointed, slightly upturned snout, and a long bushy tail (or brush). Foxes live on every continent except Antarctica. The global distribution of foxes, together with their widespread reputation for cunning, has contributed to their prominence in popular culture and folklore in many societies around the world. The hunting of foxes with packs of hounds, long an established pursuit in Europe, especially in the British Isles, was exported by European settlers to various parts of the New World.</p>",  },  ],  selectedPost: null,  });  return {  ...toRefs(data),  };  },  };  </script> |

**例7-21 带有博文的posts组件**





**图7-6 带有博文的posts**

图7-6展示了带有博文的posts组件在加载时以及在点开cat博文后的效果，此时如果切换到home或likes标签，然后再切换回posts，不会继续展示之前选择的文章，而是回到第一次加载的状态。这是因为每次切换新标签时，Vue都创建了一个新的 currentTabComponent实例。在这个例子中，标签的组件实例能够被保存下来将会是更好的，因此可以在它们第一次被创建的时候缓存下来。通过用一个<keep-alive>元素将其动态组件包裹起来以解决这个问题，如例7-22所示。

|  |
| --- |
| <keep-alive>  <component :is="currentTabComponent"></component>  </keep-alive> |

**例7-22 keep-alive的用法**

现在各个标签就保持了它们的状态，posts标签中被选中的文章将会被保留，即使当它未被渲染时也是。

通常情况下，如果每个组件在激活时并不要求每次都实时请求数据，使用keep-alive就可以避免一些不必要的重复渲染，加快页面的响应速度。

## 7.5组件案例：完善标签页组件

从例7-20到例7-22，已经搭建起了home、posts和likes三个标签页的基本框架，并学会了用keep-alive的方法保留状态。现在可以结合v-for、v-if的知识，完成标签页组件的设计。在home标签显示问候以及热门趋势，在likes显示喜爱的内容。对页面进行一些简单的美化，修改后的代码如例7-23到例7-26所示。

|  |
| --- |
| <template>  <div id="app" class="demo">  <button  v-for="tab in tabs"  v-bind:key="tab"  v-bind:class="['tab-button', { active: currentTab === tab }]"  v-on:click="currentTab = tab"  >  {{ tab }}  </button>  <keep-alive>  <component :is="currentTabComponent"></component>  </keep-alive>  </div>  </template>  <script>  import { computed, reactive, toRefs } from "vue";  import TabPosts from "./tabPosts.vue";  import TabHome from "./tabHome.vue";  import TabLikes from "./tabLikes.vue";  export default {  components: {  TabPosts,  TabHome,  TabLikes,  },  setup() {  const data = reactive({  currentTab: "Home",  tabs: ["Home", "Posts", "Likes"],  });  const currentTabComponent = computed(() => {  return "tab-" + data.currentTab.toLowerCase();  });  return {  ...toRefs(data),  currentTabComponent,  };  },  };  </script>  <style>  .demo {  font-family: sans-serif;  border: 1px solid #eee;  border-radius: 2px;  padding: 20px 30px;  margin-top: 1em;  margin-bottom: 40px;  user-select: none;  overflow-x: auto;  }  .tab-button {  padding: 12px 20px;  border-top-left-radius: 6px;  border-top-right-radius: 6px;  border: 1px solid #ccc;  cursor: pointer;  background: #f0f0f0;  margin-bottom: -2px;  margin-right: -2px;  }  .tab-button:hover {  background: #e0e0e0;  }  .tab-button.active {  background: #e0e0e0;  }  .demo-tab {  border: 1px solid #ccc;  padding: 20px;  }  </style> |

**例7-23 父组件的代码**

|  |
| --- |
| <template>  <div class="demo-tab">  <div class="card" style="background-color: lightskyblue">  <h1>Home</h1>  <h4>Here's the home page!</h4>  </div>  <div class="card" style="background-color: tomato;">  <h2>Trends</h2>  <div  v-for="trend in trends"  :key="trend.id"  :class="{  'demo-active': trend === selectedTrend,  }"  @click="selectedTrend = trend"  >  <h3>{{ trend.title }}</h3>  <div v-if="trend === selectedTrend">  <div  v-html="selectedTrend.content"  style="text-indent: 2em; text-align: left; margin: 0 3em"  ></div>  </div>  </div>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  trends: [  {  id: 2,  title: "Dog",  content:  "<p>The domestic dog (Canis familiaris when considered a separate species or Canis lupus familiaris when considered a subspecies of the wolf) is a wolf-like canid that can be found distributed around the world. The dog descended from an ancient, now-extinct wolf with the modern wolf being the dog's nearest living relative. The dog was the first species to be domesticated by hunter–gatherers more than 15,000 years ago, which predates agriculture.</p>",  },  {  id: 3,  title: "Fox",  content:  "<p>Foxes are small to medium-sized, omnivorous mammals belonging to several genera of the family Canidae. Foxes have a flattened skull, upright triangular ears, a pointed, slightly upturned snout, and a long bushy tail (or brush). Foxes live on every continent except Antarctica. The global distribution of foxes, together with their widespread reputation for cunning, has contributed to their prominence in popular culture and folklore in many societies around the world. The hunting of foxes with packs of hounds, long an established pursuit in Europe, especially in the British Isles, was exported by European settlers to various parts of the New World.</p>",  },  ],  selectedTrend: null,  });  return {  ...toRefs(data),  };  },  };  </script>  <style scoped>  .card {  width: 80%;  min-height: 50px;  border: solid 2px;  border-radius: 4px;  margin-bottom: 10px;  position: relative;  left: 10%;    }  </style> |

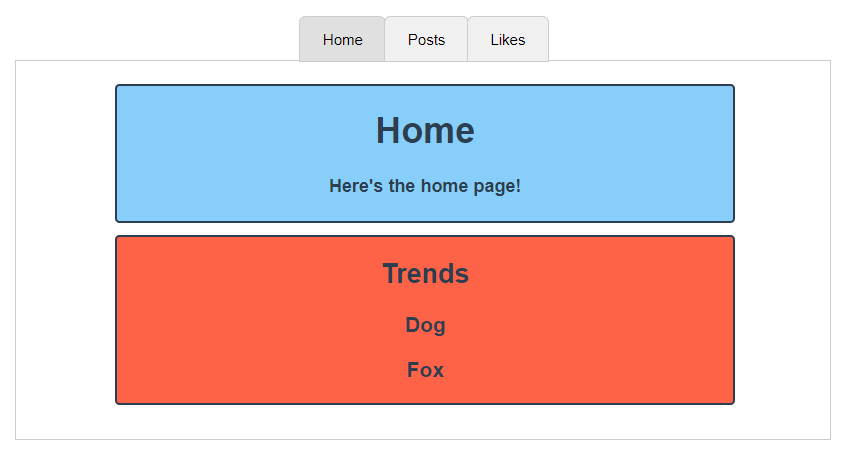
**例7-24 home组件的代码**

|  |
| --- |
| <template>  <div class="demo-posts-tab">  <ul class="demo-posts-sidebar">  <li  v-for="post in posts"  :key="post.id"  :class="{  'demo-active': post === selectedPost,  }"  @click="selectedPost = post"  >  {{ post.title }}  </li>  </ul>  <div class="demo-post-container">  <div v-if="selectedPost" class="demo-post">  <h3>{{ selectedPost.title }}</h3>  <div  v-html="selectedPost.content"  style="  text-indent: 2em;  text-align: left;  margin: 0 3em;  width: 60%;  left: 15%;  position: relative;  "  ></div>  </div>  <strong v-else> Click on a blog title to view it. </strong>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  posts: [  {  id: 1,  title: "Cat",  content:  "<p>The cat (Felis catus) is a domestic species of small carnivorous mammal. It is the only domesticated species in the family Felidae and is often referred to as the domestic cat to distinguish it from the wild members of the family. A cat can either be a house cat, a farm cat or a feral cat; the latter ranges freely and avoids human contact. Domestic cats are valued by humans for companionship and their ability to hunt rodents.</p>",  },  {  id: 2,  title: "Dog",  content:  "<p>The domestic dog (Canis familiaris when considered a separate species or Canis lupus familiaris when considered a subspecies of the wolf) is a wolf-like canid that can be found distributed around the world. The dog descended from an ancient, now-extinct wolf with the modern wolf being the dog's nearest living relative. The dog was the first species to be domesticated by hunter–gatherers more than 15,000 years ago, which predates agriculture.</p>",  },  {  id: 3,  title: "Fox",  content:  "<p>Foxes are small to medium-sized, omnivorous mammals belonging to several genera of the family Canidae. Foxes have a flattened skull, upright triangular ears, a pointed, slightly upturned snout, and a long bushy tail (or brush). Foxes live on every continent except Antarctica. The global distribution of foxes, together with their widespread reputation for cunning, has contributed to their prominence in popular culture and folklore in many societies around the world. The hunting of foxes with packs of hounds, long an established pursuit in Europe, especially in the British Isles, was exported by European settlers to various parts of the New World.</p>",  },  ],  selectedPost: null,  });  return {  ...toRefs(data),  };  },  };  </script>  <style scoped>  li {  min-height: 20px;  margin-bottom: 10px;  width: 50%;  left: 35%;  position: relative;  }  .demo-posts-sidebar {  margin-bottom: 50px;  background-color: greenyellow;  padding: 10px 0px 5px 0;  width: 30%;  left: 35%;  position: relative;  border: solid 2px;  border-radius: 4px;  }  </style> |

**例7-25 posts组件的代码**

|  |
| --- |
| <template>  <div class="demo-tab">  <div class="card">  <h2>Likes</h2>  <div  v-for="like in likes"  :key="like.id"  :class="{  'demo-active': like === selectedLike,  }"  @click="selectedLike = like"  >  <h3>{{ like.title }}</h3>  <div v-if="like === selectedLike">  <div  v-html="selectedLike.content"  style="text-indent: 2em; text-align: left; margin: 0 3em"  ></div>  </div>  </div>  </div>  </div>  </template>  <script>  import { reactive, toRefs } from "vue";  export default {  setup() {  const data = reactive({  likes: [  {  id: 1,  title: "Cat",  content:  "<p>The cat (Felis catus) is a domestic species of small carnivorous mammal. It is the only domesticated species in the family Felidae and is often referred to as the domestic cat to distinguish it from the wild members of the family. A cat can either be a house cat, a farm cat or a feral cat; the latter ranges freely and avoids human contact. Domestic cats are valued by humans for companionship and their ability to hunt rodents.</p>",  },  {  id: 3,  title: "Fox",  content:  "<p>Foxes are small to medium-sized, omnivorous mammals belonging to several genera of the family Canidae. Foxes have a flattened skull, upright triangular ears, a pointed, slightly upturned snout, and a long bushy tail (or brush). Foxes live on every continent except Antarctica. The global distribution of foxes, together with their widespread reputation for cunning, has contributed to their prominence in popular culture and folklore in many societies around the world. The hunting of foxes with packs of hounds, long an established pursuit in Europe, especially in the British Isles, was exported by European settlers to various parts of the New World.</p>",  },  ],  selectedLike: null,  });  return {  ...toRefs(data),  };  },  };  </script>  <style scoped>  .card {  width: 80%;  min-height: 50px;  border: solid 2px;  border-radius: 4px;  margin-bottom: 10px;  position: relative;  left: 10%;  background-color: gold;  }  </style> |

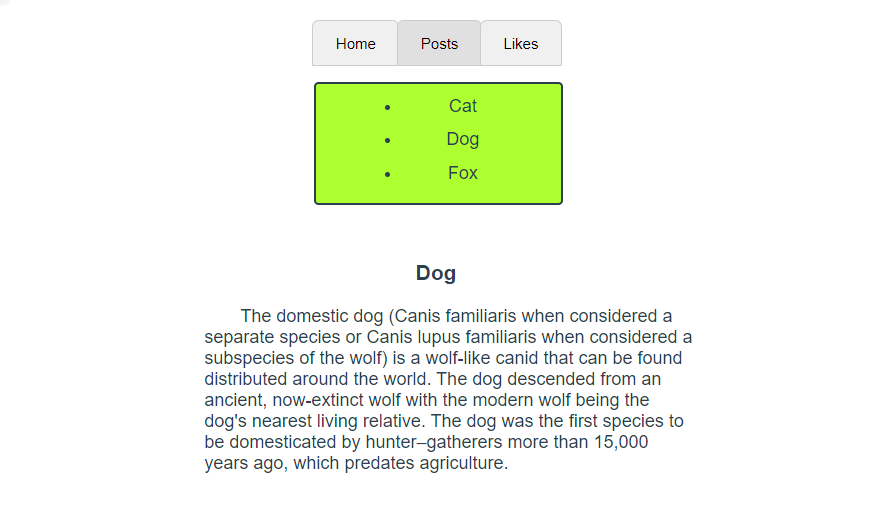
**例7-26 likes组件的代码**



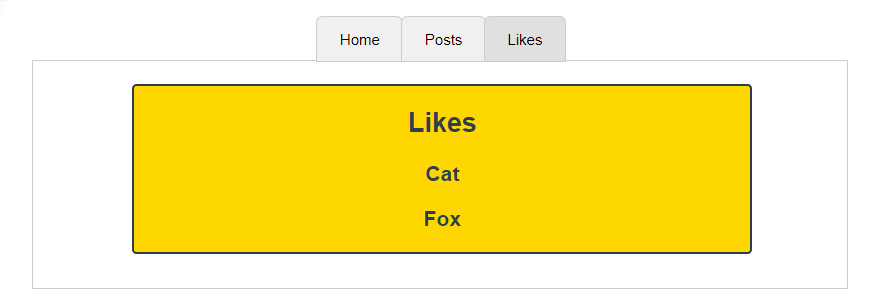


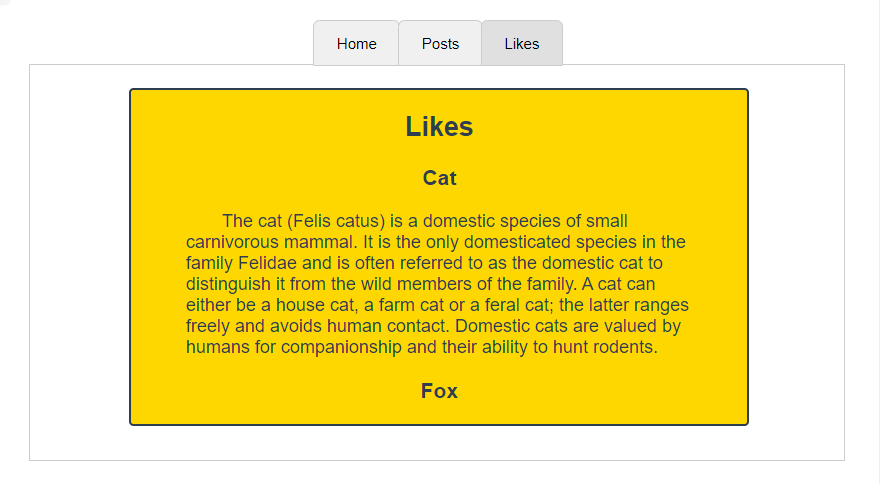
**图7-7 home标签页的效果**





**图7-8 posts标签页的效果**





**图7-9 likes标签页的效果**

最后三个标签页的显示效果如图7-7到图7-9所示，点击标题会切换展示博文的内容，并且保留打开的状态。在接入后端时，各个组件可以向后端请求数据，分别获取趋势、博文、喜爱的列表，并将获取的数据显示在页面上。

## 7.6组件在Vue 3中的变化

### 7.6.1函数式组件

在Vue 2.x中，函数式组件有两个主要作用，分别是性能优化和返回多个根节点，因为它们的初始化速度比有状态组件快得多。然而，在Vue 3中，有状态组件的性能已经提高到可以忽略不计的程度。此外，有状态组件现在也包括返回多个根节点的能力。因此，函数式组件剩下的唯一用例就是简单组件，例如创建动态标题的组件。在其他情况下，建议使用有状态组件。

在语法方面，在Vue 3中，所有的函数式组件都是用普通函数创建的，{ functional: true }选项在通过函数创建组件时已被移除，也就不再需要定义该选项。现在可以接收两个参数：props和context，其中context参数是一个对象，包含组件的attrs，slots，和emit property。同时，functional attribute在单文件组件（SFC）<template> 已被移除，因此在SFCs上使用functional时可以删除该attribute，并将props的所有引用重命名为$props，将attrs重命名为$attrs。

### 7.6.2内联模板Attribute

在Vue 2.x中，为子组件提供了inline-template属性，用于将其内部内容用作模板，而不是将其作为分发内容。在Vue 3中将不再支持此功能。最简单的解决方法是将<script>与其他类型一起使用，并在组件中使用选择器将模板作为目标。另外，以前使用inline-template的组件也可以使用默认slot进行重构，这将会使数据范围更加明确，也保留了内联编写子内容的便利。

## 7.7本章小结

本章介绍了组件的有关内容，组件的核心目标是为了可重用性高，减少重复性的开发。

组件的全局注册可以传入两个参数，第一个参数是组件的名称，第二个参数是组件的构造函数definition，可以是Function，也可以是Object。要确保在根实例初始化之前注册。局部注册则限定了组件只能在被注册的组件中使用，可以通过一个普通的JavaScript对象来定义局部组件。在组件中使用components选项注册组件，可以进行组件嵌套。

子组件中可以通过显式声明props以接收父组件数据，子组件接收到后就可以根据参数的不同来渲染不同的内容或执行操作。props的值可以有两种，一种是字符串数组，一种是对象。除了props，父级组件还可以通过v-on或@监听子组件实例的任意事件，同时子组件也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

Vue通过使用特殊的<slot>元素作为原始内容的插槽实现了内容分发，可以让组件组合，以混合父组件的内容与子组件的模板。插槽可以设置后备内容。通过给<slot>元素指定一个name属性后可以分发多个内容，有name属性的<slot>称为具名插槽。一个不带name属性的匿名<slot>默认带有隐含的名字“default”。

Vue还支持动态组件，多个组件使用同一挂载点，根据条件动态地切换不同的组件。通过keep-alive元素可以使用缓存一些非动态的组件实例，以保留组件状态。

在下一章中，将会开始介绍Vue的前端路由，逐步开始工程化。

## 7.8练习题

* **问答题：**

1. 全局注册组件传入的两个参数分别是什么？都有哪些需要注意的地方？
2. 组件之间的通信有哪几种方式？分别是什么流程？
3. 具名插槽和匿名插槽分别如何匹配父组件中的内容？
4. <transition-group>是否可以使用过渡模式？它如何改变元素定位？

* **动手做：**

1. 设计一个数字输入框组件，只能输入数字，附带两个按钮用于加1 和减1。也可以自行添加初始值、最大值、最小值的按钮。在输入框中数值改变时，触发一个自定义事件来通知父组件。

# 第8章 前端路由

路由是所有前端框架中必须具备的元素。虽然使用后端路由时，页面可以在服务端渲染好直接返回给浏览器，但是模板是由后端来维护，因此前端开发者需要安装整套的后端服务，有时还需要使用非前端语言来改写html结构，难以维护。而如果使用前后端分离的模式时，后端则可以专注在数据上，前端可以专注在交互和可视化上。

广义上的前端路由是指前端根据URL来分发视图，它定义了对于哪个URL应该由哪个文件来处理。路由需要监听浏览器地址的变化，以及动态加载视图。前端路由的优点有很多，包括页面持久、前后端分离等。在Vue中，路由有一个专门的vue-router库用于给Vue提供路由管理。

## 8.1 vue-router的基本用法

### 8.1.1安装vue-router

vue-router提供了CDN引入、npm等安装方式，可以选择其中一种方式进行安装。

Unpkg.com提供基于npm的CDN链接，网址为：https://unpkg.com/vue-router/dist/vue-router.js。这个链接将始终指向npm的最新版本。也可以通过在中间的vue-router后加入版本标签来指定版本，例如…/vue-router@2.0.0/dist…。通过以下代码可以引入vue-router：

<script src="/path/to/vue.js"></script>

<script src="/path/to/vue-router.js"></script>

vue-router在Vue添加之后将自动安装。

当使用Webpack等支持CommonJS规范的模块化打包器来构建时，可以使用npm包的方式来安装，安装代码如下：

npm install vue-router

vue-router如果要使用最新的开发版本，则必须直接从GitHub克隆并自行构建，代码如下：

git clone https://github.com/vuejs/vue-router.git node\_modules/vue-router

cd node\_modules/vue-router

npm install

npm run build

### 8.1.2 vue-router的基本使用

在开始编写vue-router相关的代码之前，可以新建一个带有router的项目，Vue CLI脚手架会自动搭建好项目的结构，并写入一些基础代码，新建项目的代码如下：

vue create router-proj

router-proj可以更换为自定义的项目名称，在后续的步骤中选择manually select features，并勾选上Router选项，选择Vue 3的版本，等待CLI完成搭建。在routes中加入一些路径与组件后，就可以显示网页内容了，修改后的代码如例8-1所示。

|  |
| --- |
| import { createRouter, createWebHashHistory } from 'vue-router'  import Home from '../views/Home.vue'  const routes = [  {  path: '/',  name: 'Home',  component: Home  },  {  path: '/about',  name: 'About',  component: () => import('../views/About.vue')  }  ]  const router = createRouter({  history: createWebHashHistory(),  routes  })  export default router |

**例8-1 router中的代码**

类似于通过is特性来实现动态组件，Vue-router的实现原理就是事实上就是在路由不同的页面动态加载不同的组件。在这段代码中，首先导入了Home与About两个组件，并分别为它们设置了路径。在导出*router*后，可以在main.js中加入以下代码来使用路由：

createApp(App) .use(router)

想要在setup函数中使用router时，可以通过useRouter()或useRoute()函数来实现。在之后，router实例将会被经常使用，需要注意的是this.$router与直接使用createRouter()创建的router实例是完全相同的，使用this.$router是因为不想在每个需要操纵路由的组件中都进行router的导入操作。

### 8.1.3跳转

vue-router有两种跳转页面的方法，第一种是使用内置的<router-link>组件，<router-link>默认会被渲染成一个<a>标签，它的to属性用于指定跳转链接。

|  |
| --- |
| <template>  <div id="nav">  <router-link to="/">Home</router-link> |  <router-link to="/about">About</router-link>  </div>  <router-view/>  </template> |

**例8-2 router-link的使用**

除了to，<router-link>还有其他的一些prop ，例如replace，不留下History 记录，也就不能用后退键返回上一个页面；active-class，当路由匹配成功时，会给当前元素设置一个router-link-active的class。

第二种可以使用router实例的方法。例如在点击事件的处理函数中使用this.$router.push(‘/about’)。

打开网页，点击About或在路径后加入/about后，就可以跳转到About组件的页面，如图8-1所示。



**图8-1 跳转到About页面**

## 8.2 动态路由匹配

### 8.2.1带参数的动态路由匹配

很多时候，需要将具有给定模式的多个路由映射到同一组件，例如可能有一个User组件是为所有用户展示的，但不同之处在于具有不同的用户ID。在Vue Router中，可以在路径中使用一个动态段来实现，称为param。Param的使用方法如例8-3所示。

|  |
| --- |
| const routes = [  { path: '/users/:id', component: User },  ] |

**例8-3 带参数的动态组件**

动态段以冒号:开头，这样之后，类似/users/1258和/users/2600的URL都将映射到相同的路由。当路由进行匹配时，路由的params可以用this.$route.params的形式在每个组件中获取。在同一路径中可以有多个参数，它们将分别映射到$route.params上的相应字段，如表8-1所示。

|  |  |  |
| --- | --- | --- |
| **格式** | **匹配路径** | **$ route.params** |
| /users/:username | /users/eduardo | { username: 'eduardo' } |
| /users/:username/posts/:postId | /users/eduardo/posts/123 | { username: 'eduardo', postId: '123' } |

**表8-1 params与路径的匹配关系**

除了$route.params，$route对象还包括了其他有用的信息，例如当URL中存在查询时会有$route.query以及$route.hash等。

### 8.2.2响应参数变化

以例8-2中的动态段举例，使用使用带参数的路由时要注意的一件事是，当用户从/users/1258转到/users/2600时，将重用相同的组件实例。由于两个路由都使用相同的组件，因此这比销毁旧实例然后创建新实例更有效。但是，这也意味着将不会调用组件的生命周期钩子函数。要对同一组件中的参数更改做出响应，可以直接监听$route对象上的内容，或者使用beforeRouteUpdate，它也可以取消导航，如例8-4所示。

|  |
| --- |
| //使用watch监听  const User = {  template: '...',  created() {  this.$watch(  () => this.$route.params,  (toParams, previousParams) => {  // reaction  }  )  },  }  //使用navigation guard  const User = {  template: '...',  async beforeRouteUpdate(to, from) {  // reaction  this.userData = await fetchUser(to.params.id)  },  } |

**例8-4 响应组件参数变化**

### 8.2.3参数全匹配

常规参数只会匹配网址片段之间用/分隔的字符，如果想匹配任何字符，可以使用自定义的参数正则表达式，方法是通过在参数后面紧跟的括号内添加正则表达式，如例8-5所示。

|  |
| --- |
| const routes = [  //匹配所有并放入`$route.params.pathMatch`  { path: '/:pathMatch(.\*)\*', name: 'NotFound', component: NotFound },  // 匹配所有以`/user-`开头的并放入`$route.params.afterUser`  { path: '/user-:afterUser(.\*)', component: UserGeneric },  ] |

**例8-5 参数全匹配**

在这种特定情况下，可以在括号之间使用自定义的正则表达式，并使用?修饰符（0或1）将pathMatch参数标记为可选、使用\*（0或更多）和+（1或更多）将pathMatch参数标记为可重复的参数。通过将path拆分为一个数组可以在需要时直接导航到路由，如例8-6所示。

|  |
| --- |
| this.$router.push({  name: 'NotFound',  //保留当前路径并删除第一个字符，以免目标网址以“//”开头  params: { pathMatch: this.$route.path.substring(1).split('/') },  // 如果有的话保留现有query和hash  query: this.$route.query,  hash: this.$route.hash,  }) |

**例8-6 将path拆分为数组**

## 8.3路由匹配的语法

### 8.3.1自定义正则表达式

大多数应用程序都使用静态路由，例如/about，动态路由的使用如图/users/:userId。除此之外，还可以使用自定义的正则表达式

当定义param时，可以在内部使用[^/]+，其中至少有一个不是/的字符，用于从URL中提取param，除了在需要根据参数内容区分两个路由的情况下，这种方法的效果都很好。当两条路由/:orderId和/:productName将匹配完全相同的URL时，需要一种方法区分它们。最简单的方法是将静态部分添加到它们的路径中，如例8-7所示。

|  |
| --- |
| const routes = [  //匹配 /o/3549  { path: '/o/:orderId' },  //匹配 /p/books  { path: '/p/:productName' },  ] |

**例8-7 区分相同URL的路由**

但是在某些情况下不想添加静态部分。但是，orderId与productName可能是相同的数字，另一种方法是在括号中为参数指定自定义正则表达式，如例8-8所示。

|  |
| --- |
| const routes = [  // /:orderId 只能匹配数字  { path: '/:orderId(\\d+)' },  // /:productName 所有字符都可以  { path: '/:productName' },  ] |

**例8-8 自定义正则表达式**

此时/25将会匹配/:orderId，而其他将会匹配/:productName，routes中的数组顺序将不再重要。

### 8.3.2可选参数

在动态路由中提到了使用?修饰符可以将参数标记为可选参数，如例8-9所示。

|  |
| --- |
| const routes = [  // 将会匹配 /users 和 /users/posva  { path: '/users/:userId?' },  // 将会匹配 /users 和 /users/42  { path: '/users/:userId(\\d+)?' },  ] |

**例8-9 可选参数的使用方法**

### 8.3.3可重复参数

如果需要将路由与多个部分匹配，则可以使用\*（可以有0个或更多）或者+（可以有1个或更多）将参数标记为可重复，如例8-10所示。

|  |
| --- |
| const routes = [  // /:chapters 可以匹配 /one, /one/two, /one/two/three,…  { path: '/:chapters+' },  // /:chapters可以匹配 /, /one, /one/two, /one/two/three,…  { path: '/:chapters\*' },  ] |

**例8-10 可重复参数的使用方法**

这将提供一个参数数组而不是字符串，并且还要求在使用命名路由时传递一个数组，如例8-11所示。

|  |
| --- |
| // 生成 /  router.resolve({ name: 'chapters', params: { chapters: [] } }).href  // 生成 /a/b  router.resolve({ name: 'chapters', params: { chapters: ['a', 'b'] } }).href  router.resolve({ name: 'chapters', params: { chapters: [] } }).href  // 报错，因为 `chapters` 为空 |

**例8-11 给命名路由传递数组**

当与自定义正则表达式结合使用时，需要将\*与+添加在右括号后。同时，\*在使用时也可以认为是将参数标记为可选的，但是?标记的参数不能重复。

## 8.4嵌套路由

很多情况下，应用的UI会由嵌套在多个层级的组件组成。这时，URL的某段对应于嵌套组件的某种结构是很常见的。通过Vue Router，可以使用嵌套的路由配置来表达这种关系。

<router-view>是顶层router-view，它渲染与顶层路由匹配的组件。类似的，渲染的组件也可以包含其自己的嵌套路由，如例8-12所示。

|  |
| --- |
| //组件中  const User = {  template: `  <div class="user">  <h2>User {{ $route.params.id }}</h2>  <router-view></router-view>  </div>  `,  }  //路由中  const routes = [  {  path: '/user/:id',  component: User,  children: [  {  // 匹配时UserProfile将会被渲染入User的<router-view>中  path: 'profile',  component: UserProfile,  },  {  // 匹配时UserPosts将会被渲染入User's <router-view>中  path: 'posts',  component: UserPosts,  },  ],  },  ] |

**例8-12 路由与组件的嵌套匹配**

这里首先在User组件的模板中添加了一个<router-view>，此时要将组件呈现到被嵌套的router-view中，就需要在路由中添加children选项。

需要注意的是，以/开头的嵌套路径将会被视为根路径，这使得利用组件嵌套时不必使用嵌套的URL。

在上面的例子中，当访问/user/eduardo时，User的router-view中的内容不会被渲染，因为没有匹配的嵌套路由。如果想要渲染部分内容，，可以提供一个空的嵌套路径，如例8-13所示。

|  |
| --- |
| const routes = [  {  path: '/user/:id',  component: User,  children: [  // 在匹配时UserHome将会被渲染在User的<router-view>中  { path: '', component: UserHome },  ...  ],  },  ]] |

**例8-13 空嵌套路径**

## 8.5命名路由

除了path，还可以给路由提供name属性。这具有以下优点：没有硬编码的网址、自动编码/解码params、防止在网址中输入错字、绕过路径排名。要链接到命名路由，可以将一个对象传递给router-link组件的to参数，如例8-14所示。

|  |
| --- |
| //路由  const routes = [  {  path: '/user/:username',  name: 'user',  component: User  }  ]  //组件  <router-link :to="{ name: 'user', params: { username: 'erina' }}">  User  </router-link> |

**例8-14 命名路由的使用**

路由将导航到路径 /user/erina，这与下面的写法具有同样的效果。

router.push({ name: 'user', params: { username: 'erina' } })

## 8.6重定向和别名

### 8.6.1重定向

重定向也在routes配置中完成，也可以针对命名路由或使用函数进行动态重定向。如例8-15所示。

|  |
| --- |
| const routes = [{ path: '/home', redirect: '/' }]  // 命名路由  const routes = [{ path: '/home', redirect: { name: 'homepage' } }]  // 使用函数  const routes = [  {  // /search/screens 到 /search?q=screens  path: '/search/:searchText',  redirect: to => {  return { path: '/search', query: { q: to.params.searchText } }  },  },  {  path: '/search',  ...  },  ] |

**例8-15 重定向的使用**

编写redirect时，可以省略component选项，因为不会直接到达，所以没有要渲染的组件。唯一的例外是嵌套路由，如果路由记录有children和redirect属性，则它也应该有component属性。

除了直接定位到绝对位置，也可以重定向到相对位置。

### 8.6.2别名

/的别名/home意味着当用户访问/home时，URL保留/home不变，但将被匹配为用户就像正在访问/时一样。

使用别名可以自由地将UI结构映射到任意URL，而不受配置的嵌套结构约束。将别名的开头设为/，使路径在嵌套路由中成为绝对路径。甚至可以将两者结合起来，为数组提供多个别名。别名的使用如例8-16所示。

|  |
| --- |
| const routes = [{ path: '/', component: Homepage, alias: '/home' }]  // 为数组提供多个别名  const routes = [  {  path: '/users',  component: UsersLayout,  children: [  // UserList 将会渲染出3个URL  // /users 、 /users/list 和 /people  { path: '', component: UserList, alias: ['/people', 'list'] },  ],  },  ] |

**例8-16 别名的使用**

如果路由包含param，确保将其包含在绝对别名中，如例8-17所示。

|  |
| --- |
| const routes = [  {  path: '/users/:id',  component: UsersByIdLayout,  children: [  // UserDetails将会渲染出3个URL  // - /users/24 、 /users/24/profile 和 /24  { path: 'profile', component: UserDetails, alias: ['/:id', ''] },  ],  },  ] |

**例8-17 包含参数的别名**

## 8.7向路由组件传递参数

### 8.7.1基本语法

在组件中使用$route会导致组件与路由紧密耦合，限制组件的灵活性，因为这样组件只能在特定URL上使用。通过使用props选项可以避免这种情况。如例8-18所示。

|  |
| --- |
| const User = {  props: ['id'],  template: '<div>User {{ id }}</div>'  }  const routes = [{ path: '/user/:id', component: User, props: true }] |

**例8-18 向路由组件传递参数**

这样可以在任何地方使用该组件，保证了组件的重用和测试。

### 8.7.2传递参数的模式

传递参数有4种模式，分别为布尔、命名视图、对象、功能模式。

在布尔模式中，当props设置为时true，route.params将被设置为组件props。

在命名视图中，对具有命名视图的路由，必须为每个命名视图定义props选项，如例8-19所示。

在对象模式中，当props是对象时，它将按原样设置为组件props。特别是当props是静态的时十分有用，如例8-19所示。

在功能模式中，可以创建一个返回props的函数。可以用来将参数转换为其他类型、将静态值与基于路由的值结合起来等，如例8-19所示。

|  |
| --- |
| // 命名视图  const routes = [  {  path: '/user/:id',  components: { default: User, sidebar: Sidebar },  props: { default: true, sidebar: false }  }  ]  // 对象模式  const routes = [  {  path: '/promotion/from-newsletter',  component: Promotion,  props: { newsletterPopup: false }  }  ]  // 功能模式  const routes = [  {  path: '/search',  component: SearchUser,  props: route => ({ query: route.query.q })  }  ] |

**例8-19 传递参数的模式**

## 8.8 vue-router 4的变化

Vue Router 4.0提供了Vue 3支持，并有许多突破性的变化。

### 8.8.1 Vue Router的创建

Vue Router不再作为类，而是作为一组函数。因此新建的方法从原先的new Router()改变为createRouter，如例 所示。

|  |
| --- |
| // vue-router 3.x的写法  // import Router from 'vue-router'  // const router = new VueRouter({});  // vue-router 4的写法  import { createRouter } from 'vue-router'  const router = createRouter({}) |

**例8-20 vue-router 4的新建方法**

### 8.8.2新的history选项

原有的mode: 'history'选项已经被更灵活的新选项history取代。迁移时，需要根据使用的模式，用合适的函数替代原来的mode，替换的代码如下：

"history": createWebHistory()

"hash": createWebHashHistory()

"abstract": createMemoryHistory()

### 8.8.3删除\*路由

捕获所有路由（\*，/\*）现在必须使用带有自定义正则表达式的参数进行定义，如例8-21所示。

|  |
| --- |
| onst routes = [  { path: '/:pathMatch(.\*)\*', name: 'not-found', component: NotFound },  //如果省略最后一个`\*`，则解析或推送时将对params中的`/`字符进行编码  { path: '/:pathMatch(.\*)', name: 'bad-not-found', component: NotFound },  ]  // 不好的写法  router.resolve({  name: 'bad-not-found',  params: { pathMatch: 'not/found' },  }).href // '/not%2Ffound'  // 推荐的写法  router.resolve({  name: 'not-found',  params: { pathMatch: ['not', 'found'] },  }).href / |

**例8-21 捕获所有路由的写法**

### 8.8.4 <router-link>的修改

Append选项已从<router-link>中移除，可以手动将值连接到现有path，如下所示。

<router-link :to="append($route.path, 'child-route')"></router-link>

另外，event和tag选项也已经从<router-link>删除。现在可以使用v-slot的API完全自定义<router-link>。

exact选项也已被删除，因为正在修复的警告不再存在，因此可以安全地删除它。

### 8.8.5去除router.app

router.app过去代表注入路由器的最后一个根组件（Vue实例）。现在多个Vue应用可以同时安全地使用Vue Router，不过仍然可以在使用路由器时添加它。

### 8.8.6向route组件的<slot>传递内容

当有路由组件的<slot>嵌套在<router-view>组件下，并且需要直接传递要由<slot>渲染的模板时，需要将它通过v-slot传递给<component>，如例8-22所示。

|  |
| --- |
| <router-view v-slot="{ Component }">  <component :is="Component"></component>  </router-view> |

**例8-22 向route组件的<slot>传递内容**

### 8.8.7 $route属性编码

无论导航在哪里初始化，在params、query和hash中的解码值现在都保持一致。初始导航与应用内导航应产生相同的结果。

Hash现在已经解码，可以使用router.push({ hash: $route.hash })进行复制，并且可以在el中直接使用。

当使用push、resolve和replace并且在对象中提供了string位置或path属性时，必须对其进行编码。而params、query和hash必须在未编码的版本中提供。

现在，斜线字符（/）已在params内部正确解码，同时仍在URL上生成编码后的版本。

## 8.9本章小结

本章介绍了路由的有关内容，路由是所有前端框架中必须具备的元素，它定义了对于哪个URL应该由哪个文件来处理。前端路由的优点包括页面持久、前后端分离等。

路由的跳转可以使用内置的<router-link>组件，<router-link>默认会被渲染成一个<a>标签，它的to属性用于指定跳转链接。此外，还可以使用router实例的方法，例如$router.push()。

在Vue Router中，可以在路径中使用param来实现动态路由，将具有给定模式的多个路由映射到同一组件。路由匹配的语法包括自定义正则表达式、使用?修饰的可选参数以及+和\*修饰的可重复参数。

通过Vue Router，还可以使用嵌套的路由配置来表达URL的某段对应于嵌套组件的某种结构的关系，没成为嵌套路由。

通过使用props选项可以向路由组件传递参数，可以避免组件与路由紧密耦合，限制组件的灵活性。

在下一章中，将会开始介绍状态管理与VueX。

## 8.10练习题

* **问答题：**

1. 路由的跳转有哪些方式？
2. 动态组件参数的可选、0个或以上以及1个或以上的修饰符分别是什么？
3. 简述别名的作用。

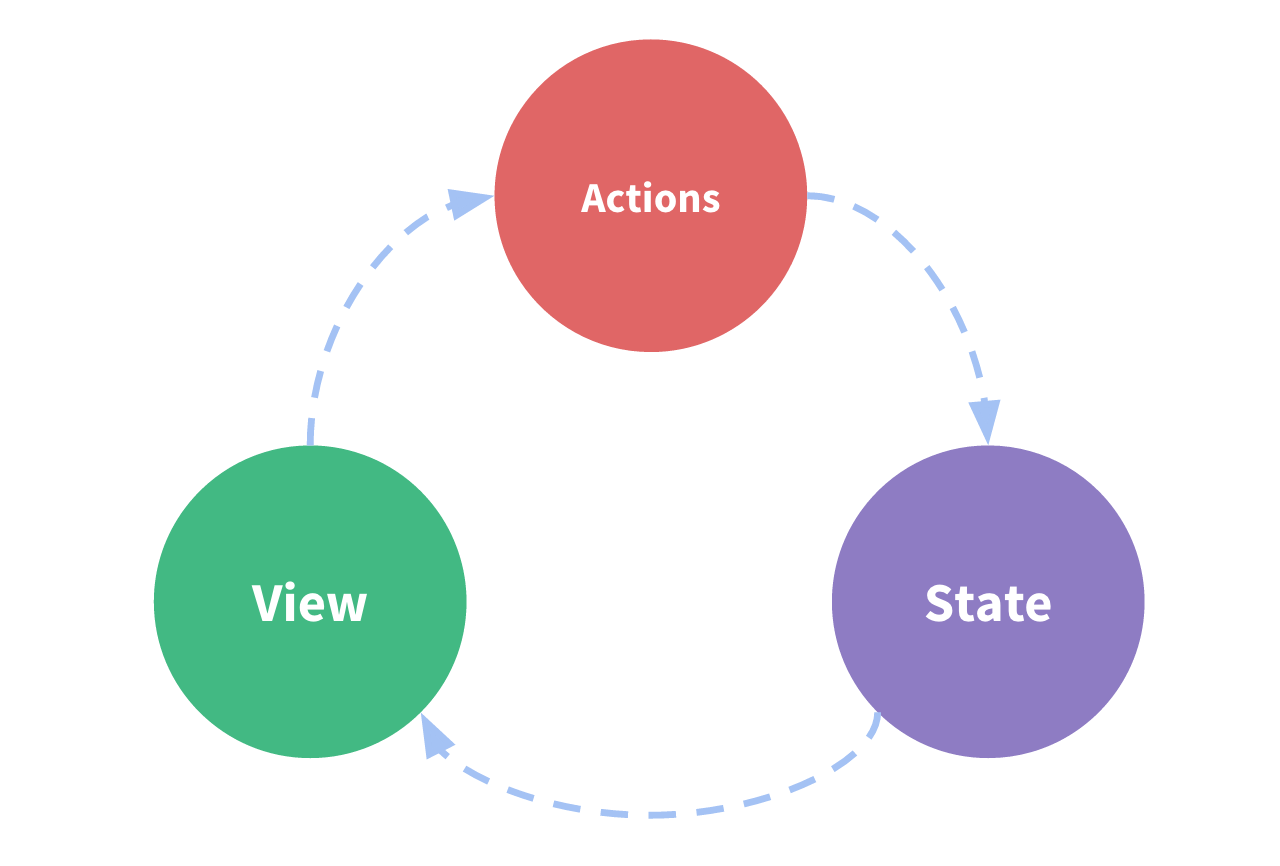
# 第9章 状态管理与Vuex

## 9.1 Vuex简介

### 9.1.1状态管理模式

Vuex是Vue的状态管理模式+库。它作为应用中所有组件的集中仓库，确保只能以可预测的方式更改状态。

在一个自包含应用中，例如最开始编写的计数器，往往包含三个部分。状态部分state，包括驱动应用的数据源；视图部分view，指明状态的映射；动作部分action，指明状态根据用于在视图中的输入进行改变的可能方式。三部分的交互如图9-1所示。



**图9-1 单向数据流的概念**

但是，当存在多个组件共享相同状态时，这种过程将会被打破。因为多个视图可能依赖于同一个状态，并且来自不同视图的动作可能需要改变同一个状态。

传递props可以用一种很繁琐的方式解决深度嵌套组件的情况，但是对于同级组件则根本不起作用。对于第二个问题，经常使用获得直接的父/子实例引用的方法，或者通过事件来对状态的多个副本进行变异和同步。但是这样两种方式都容易导致代码变得难以维护。

通过定义和分离状态管理中涉及的概念，并执行保证视图和状态之间独立性的规则，可以为代码提供更多的结构和高维护性。这就是Vuex的基本思想，它也是专门为Vue定制的库，可以利用细粒度的反应进行高效更新。



**图9-2 Vuex状态管理的概念**

Vuex以更多概念和样板为代价应对共享状态管理。这是一种短期和长期生产力之间的权衡。如果应用很简单，那么不使用Vuex，而是使用一个简单的存储模式，可能会是更合适的。但是，在构建中型到大型的SPA时，很可能会遇到处理Vue组件外部状态的情况，而Vuex将是更好的选择。

### 9.1.2安装Vuex

Vuex 4.0提供了Vue 3支持，其API与Vuex 3.x基本相同。唯一的突破性变化是插件的安装方式。

Vuex提供了CDN引入、npm、yarn等安装方式，可以选择其中一种方式进行安装。

Unpkg.com提供基于npm的CDN链接，网址为：https://unpkg.com/vuex。这个链接将始终指向npm的最新版本。也可以通过在vuex后加入版本标签来指定版本，例如https://unpkg.com/vuex@2.0.0。通过以下代码可以引入Vuex：

<script src="/path/to/vue.js"></script>

<script src="/path/to/vuex.js"></script>

Vuex在Vue添加之后将自动安装。

除了CDN引入，还可以使用npm或yarn包的方式来安装，安装代码分别如下：

npm install vuex@next –save

yarn add vuex@next –save

Vuex需要包含Promise，可以通过CDN包含它：

<script src="https://cdn.jsdelivr.net/npm/es6-promise@4/dist/es6-promise.auto.js"></script>

使用NPM或Yarn之类的程序包管理器可以使用以下命令进行安装Promise：

npm install es6-promise --save # NPM

yarn add es6-promise # Yarn

之后，在使用Vuex时，将以下行添加到代码中的任何位置以使用Promise：

import 'es6-promise/auto'

如果要使用vuex最新的开发版本，必须直接从GitHub克隆并自行构建，代码如下：

git clone https://github.com/vuejs/vuex.git node\_modules/vuex

cd node\_modules/vuex

yarn

yarn build

### 9.1.3 Vuex的基本使用

每个Vuex应用的核心都是仓库store。仓库就是一个保存应用状态的容器。Vuex仓库与普通的全局对象大致有两点不同之处。Vuex仓库是反应式的，当Vue组件从中检索状态时，如果仓库的状态发生变化，它们将快速地更新。仓库的状态不能直接更改，只能通过显式地提交突变mutation。理由是这样可以确保每个状态更改都留下可跟踪的记录，并可以使用工具用于更好地了解应用。

仓库的创建方式如例9-1所示，只需要提供一个初始状态对象和一些mutation。

|  |
| --- |
| import { createApp } from 'vue'  import { createStore } from 'vuex'  // 创建一个新的仓库实例  const store = createStore({  state () {  return {  count: 0  }  },  mutations: {  increment (state) {  state.count++  }  }  })  const app = createApp({ /\* 根组件 \*/ })  // 像插件一样安装仓库实例  app.use(store) |

**例9-1 创建Vuex仓库**

现在可以通过store.state访问状态对象，并使用store.commit方法触发状态更改。在Vue组件中，可以通过this.$store来访问商店。现在，还可以使用component方法进行更改，如例9-2所示。

|  |
| --- |
| JS  // 状态更改  store.commit('increment')  console.log(store.state.count) // -> 1  methods: {  increment() {  this.$store.commit('increment')  console.log(this.$store.state.count)  }  } |

**例9-2 Vuex状态更改**

在这里需要通过提交mutation而不是直接更改store.state.count的原因是想要明确地追踪它，以便在维护代码时可以更容易地推断应用中的状态变更。也就可以通过工具记录每个mutation，拍摄状态快照、执行跨时间调试等。

## 9.2 Vuex中的状态

### 9.2.1单一状态树

Vuex使用单个状态树，即该单个对象包含所有应用的层级状态。这也意味着通常每个应用只有一个仓库。这使查找特定状态以及获取当前应用的状态快照进行调试都变得更容易。单个状态树也不会与模块性冲突。

在Vuex中存储的数据遵循与Vue实例中的数据相同的规则，即状态对象必须是纯文本。

### 9.2.2将Vuex状态加入Vue组件

由于Vuex商店是反应式的，因此从中检索状态的最简单方法是从计算属性中返回一些商店状态，如例9-3示。

|  |
| --- |
| const Counter = {  template: `<div>{{ count }}</div>`,  computed: {  count () {  return store.state.count  }  }  } |

**例9-3 在计算属性中返回状态**

每当store.state.count更改时，将触发计算属性进行重新计算，并更新关联的DOM。但是，这样的方式会导致组件依赖全局仓库。使用模块系统时，要求在使用仓库状态的每个组件中导入仓库，并且在测试组件时还需要模拟。

为了解决这个问题，Vuex通过Vue的插件系统将仓库从根组件插入到所有子组件中，并可以通过this.$store的形式在子组件上使用。现在，可以把Counter中的count()改写为this.$store.state.count。

### 9.2.3 mapState的使用

当一个组件需要使用多个仓库状态属性或getter时，声明所有这些计算属性可能会变得重复和冗长。为了解决这个问题，可以利用mapState生成getter函数，如例9-4所示。

|  |
| --- |
| import { mapState } from 'vuex'  export default {  computed: mapState({  count: state => state.count,  countAlias: 'count',  // 为了通过`this`访问本地state，需要使用一个普通函数  countPlusLocalState (state) {  return state.count + this.localCount  }  })  } |

**例9-4 利用mapState生成getter函数**

当映射的计算属性的名称与状态子树名称相同时，还可以将字符串数组传递给mapState，如下所示：

computed: mapState(['count'])

这将会把this.count映射到store.state.count。

### 9.2.4组件的本地状态

使用Vuex并不意味着必须将所有状态都放入Vuex。尽管将更多状态添加到Vuex中可以使状态mutation更明确，并且更容易调试，但有时它也会使代码更冗长和间接。如果一个状态严格地属于单个组件，那么可以将其保留为本地状态。对于不同的组件可以权衡取舍。

## 9.3 Vuex中的getter

### 9.3.1仓库的getter

有时在开发中可能需要根据仓库状态来计算派生状态，此时如果需要使用多个组件，则必须复制该函数，或者将其提取到共享helper中然后将其导入多个位置，这两种方法都比较复杂。Vuex中则允许在仓库中定义getter，类似于视为仓库的计算属性，如例9-5所示。

|  |
| --- |
| const store = createStore({  state: {  todos: [  { id: 1, text: '...', done: true },  ]  },  getters: {  doneTodos (state) {  return state.todos.filter(todo => todo.done)  }  }  }) |

**例9-5 在仓库中定义getter**

### 9.3.2属性式访问

getter将存储在store.getters对象上，可以作为属性访问它的值。getter还可以接收其他getter作为第二个参数，如例9-6所示。

|  |
| --- |
| getters: {  // ...  doneTodosCount (state, getters) {  return getters.doneTodos.length  }  }  store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]  //现在可以在任何组件中使用它  computed: {  doneTodosCount () {  return this.$store.getters.doneTodosCount  }  } |

**例9-6 属性式地使用getter**

### 9.3.3方法式访问

除了属性式，还可以通过返回函数将参数传递给getter。当需要查询仓库中的数组时，将十分高效，如例9-7所示。

|  |
| --- |
| getters: {  // ...  getTodoById: (state) => (id) => {  return state.todos.find(todo => todo.id === id)  }  }  store.getters.getTodoById(1) // -> { id: 1, text: '...', done: true } |

**例9-6 通过方法使用getter**

需要注意的是，通过方法访问的getter会在每次调用它们时运行，并且结果不会被缓存。

### 9.3.4 mapGetter的使用

mapGetter可以简单地将仓库getter映射到本地计算属性，如果要将getter映射到其他名称，可以使用对象，如例9-7所示。

|  |
| --- |
| export default {  computed: {  ...mapGetters([  doneCount: 'doneTodosCount',  'anotherGetter',  ])  }  } |

**例9-7 mapGetter的使用**

## 9.4 Vuex中的mutation

### 9.4.1 mutation的有效负载

在提交mutation时，可以将另一个参数传递给store.commit，称为mutation的有效负载。在大多数情况下，有效载荷可以是一个对象以便包含多个字段，并且记录的mutation也将更详细，如例9-8所示。

|  |
| --- |
| // 一个参数的写法  mutations: {  increment (state, n) {  state.count += n  }  }  store.commit('increment', 10)  // 传递对象的写法  mutations: {  increment (state, payload) {  state.count += payload.amount  }  }  store.commit('increment', {  amount: 10  }) |

**例9-8 mutation的有效负载**

### 9.4.2通过对象提交

提交mutation的另一种方法是直接使用具有type属性的对象，使用对象提交时，整个对象将作为有效载荷传递给mutation处理程序，因此处理程序保持不变，如例9-9所示。

|  |
| --- |
| mutations: {  increment (state, payload) {  state.count += payload.amount  }  }  store.commit({  type: 'increment',  amount: 10  }) |

**例9-9 通过对象提交mutation**

### 9.4.3 mutation的同步

在使用mutation时，一个重要的规则是mutation的处理函数必须是同步的，因为处理函数回调中执行的任何状态mutation是不可追踪的。

异步与状态mutation结合会使代码难以推理。例如，当同时使用两个使状态发生变化的异步回调来调用两个方法时，无法知道它们何时被调用以及哪个回调首先被调用。因此，在Vuex中，mutation是同步事务，为了处理异步操作，需要使用动作Actions。

## 9.5 Vuex中的action

### 9.5.1 action的基本使用

action类似mutation，不同之处在于action不会改变状态，而是执行改变，同时action可以包含任意异步操作。action的使用如例9-10所示。

|  |
| --- |
| const store = createStore({  state: {  count: 0  },  mutations: {  increment (state) {  state.count++  }  },  actions: {  increment (context) {  context.commit('increment')  }  }  }) |

**例9-10 action的基本使用**

Action的处理程序会收到一个上下文对象，该对象在仓库实例上有相同的方法货属性集，因此可以调用context.commit来提交一个mutation，或者通过context.state和context.getters访问状态和getter。

### 9.5.2调度action

通过store.dispatch方法可以触发action，因为mutation必须是同步的，但action不是，可以在一个action中执行异步操作，如例9-11所示。

|  |
| --- |
| store.dispatch('increment')  actions: {  incrementAsync ({ commit }) {  setTimeout(() => {  commit('increment')  }, 1000)  }  } |

**例9-11 调度action**

Action同样支持有效负载和对象式的写法，如例9-12所示。

|  |
| --- |
| // 有效负载  store.dispatch('incrementAsync', {  amount: 10  })  // 对象式  store.dispatch({  type: 'incrementAsync',  amount: 10  }) |

**例9-11 调度action**

### 9.5.3组成action

action通常是异步的，那么了解action的时机，以及将多个action组合在一起以处理更复杂的异步流将会是一个难题。

首先要知道的是，store.dispatch可以处理由触发的action处理程序返回的Promise，并且还可以返回Promise。因此，通过使用async或await，可以编写多个个action，如例9-12所示。

|  |
| --- |
| actions: {  actionA ({ commit }) {  return new Promise((resolve, reject) => {  setTimeout(() => {  commit('someMutation')  resolve()  }, 1000)  })  }  }  store.dispatch('actionA').then(() => {})  // 在另一个动作中  actions: {  actionB ({ dispatch, commit }) {  return dispatch('actionA').then(() => {  commit('someOtherMutation')  })  }  }  //使用async与await  actions: {  async actionA ({ commit }) {  commit('gotData', await getData())  },  async actionB ({ dispatch, commit }) {  await dispatch('actionA') // wait for `actionA` to finish  commit('gotOtherData', await getOtherData())  }  } |

**例9-12 组合action**

## 9.6 Vuex中的模块

由于使用单状态树，因此应用的所有状态都包含在一个对象内。但是，随着应用规模扩大，仓库可能会因此变得庞大。

为了解决这个问题，Vuex允许将仓库划分为模块。每个模块可以包含其自己的状态，mutation、action、getter，甚至是嵌套模块。

在模块的mutation和getter内部，接收到的第一个参数将是模块的本地状态。模块getter内部，根状态是第三个参数，如例9-13所示。

|  |
| --- |
| const moduleA = {  state: () => ({  count: 0  }),  mutations: {  increment (state) {  // `state` is the local module state  state.count++  }  },  actions: {  incrementIfOddOnRootSum ({ state, commit, rootState }) {  if ((state.count + rootState.count) % 2 === 1) {  commit('increment')  }  }  },  getters: {  sumWithRootCount (state, getters, rootState) {  return state.count + rootState.count  }  }  }  const moduleB = {  state: () => ({ ... }),  mutations: { ... },  actions: { ... }  }  const store = createStore({  modules: {  a: moduleA,  b: moduleB  }  })  store.state.a // -> `moduleA`'s state  store.state.b // -> `moduleB`'s state |

**例9-13 模块的使用**

在模块action内部，本地状态可以通过context.state获取，而根状态可以通过context.rootState获取。

## 9.7本章小结

本章介绍了Vuex和状态管理的有关内容，Vuex是Vue的状态管理模式+库。它作为应用中所有组件的集中仓库，确保只能以可预测的方式更改状态。Vuex的基本思想是通过定义和分离状态管理中涉及的概念，并执行保证视图和状态之间独立性的规则，为代码提供更多的结构和高维护性。在构建中型到大型的SPA时，Vuex往往是更好的选择。

仓库store是每个Vuex应用的核心。仓库就是一个保存应用状态的容器。仓库的状态变更需要通过显式地提交mutation，以确保每个状态更改都留下可跟踪的记录。

Vuex的仓库状态使用单个状态树的形式，单个对象包含所有应用的层级状态。这使查找特定状态以及获取当前应用的状态快照都变得容易。但使用Vuex并不必须将所有状态都放入Vuex。如果一个状态严格地属于单个组件，那么可以将其保留为本地状态。

Vuex中允许在仓库中定义getter，类似于仓库的计算属性，用于在开发中根据仓库状态来计算派生状态。

Vuex中提供了mutation，它处理函数必须是同步的。可以通过有效负载和对象的形式提交mutation。为了处理异步操作，需要使用动作Actions。action不会改变状态，而是执行改变，同时action可以包含任意异步操作。Action同样支持有效负载和对象式的写法。

Vuex允许将仓库划分为模块，用于解决随着应用规模扩大，仓库因此变得庞大的问题。每个模块可以包含其自己的状态，mutation、action、getter，甚至是嵌套模块。

在下一章中，将会开始介绍Vue的工程实例，包括目录结构、页面间传参、项目部署等内容。

## 9.8练习题

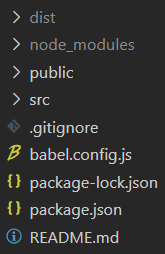
* **问答题：**

1. 仓库的状态如何更改？为什么选择这种方法？
2. 通过方法访问的getter的运行时机是什么？有什么需要注意的？
3. 在模块的getter内部，参数分别代表什么？

# 第10章 Vue工程实例

## 10.1项目目录介绍

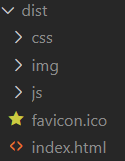
现在，完善了路由与Vuex方面的知识之后，可以新建一个项目，并在自定义配置中勾选上vue-router与Vuex的选项。打开初始化后的项目后，可以发现里面已经存在一些脚手架自动搭建的文件和文件夹，如图10-1所示。



**图10-1 项目的目录结构**

### 10.1.1 dist文件夹

由npm run build 生成，是项目的生产版本，项目完成后，交付该文件夹即可。



**图10-2 dist的目录结构**

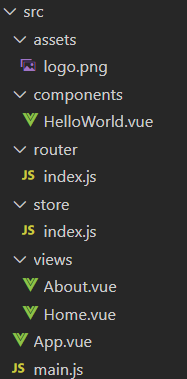
在dist文件夹下，有css、js与img文件夹，如图10-2所示。文件夹中分别存放有css、js与图片文件，其中文件名中无意义的字符串是随机生成的，是为了让文件名发生变化以便部署，方便Nginx 服务器重新对该文件进行缓存。app.css是编译后的css文件，app.js是核心的js文件，打包代码逻辑。.map文件非常重要，浏览器可以先下载整个.js 文件，然后后续通过.map文件部分加载文件。这个文件夹应该加入.gitignore中，因为每次编译这里的文件都会改变。

### 10.1.2 node modules文件夹

node项目往往会使用很多第三方包，通过命令$ npm install可以自定义安装第三方包，同时所有在package.jaon中定义的第三方包也会被自动下载，保存在node modules文件夹下。因此，往往node modules文件夹中的内容有很多，也应该加入.gitignore中，在克隆后可以通过$ npm install再次自动导入第三方包。

### 10.1.3 src文件夹

src文件夹是核心代码和工作空间的所在目录。展开后如图10-3所示。



**图10-3 src的目录结构**

assets文件夹用于存放图片、音频、视频等资源。components文件夹用于存放视图和组件，是开发的核心。router/index.js 用于配置项目前端路由，定义了页面对应的url。App. vue是Vue CLI默认创建的项目根组件，所有的其他页面都从根组件进行延伸。main.js 是webpack的入口文件，支撑Vue框架。

## 10.2前端页面开发

### 10.2.1 Vue文件

在脚手架搭建项目的过程中编写了一些文件，许多文件都很有代表性，可以用做实际开发中的样本。例如HelloWord.vue文件，就是一个较为标准的vue单页面，如例10-1所示。

|  |
| --- |
| <template>  <div class="hello">  <h1>{{ msg }}</h1>  <p>  For a guide and recipes on how to configure / customize this project,<br>  check out the  <a href="https://cli.vuejs.org" target="\_blank" rel="noopener">vue-cli documentation</a>.  </p>  <h3>Installed CLI Plugins</h3>  <ul>  <li...</li>  </ul>  <h3>Essential Links</h3>  <ul>  <li>...</li>  </ul>  <h3>Ecosystem</h3>  <ul>  <li>...</li>  </ul>  </div>  </template>  <script>  export default {  name: 'HelloWorld',  props: {  msg: String  }  }  </script>  <style scoped lang="scss">  h3 {  margin: 40px 0 0;  }  ul {  list-style-type: none;  padding: 0;  }  li {  display: inline-block;  margin: 0 10px;  }  a {  color: #42b983;  }  </style> |

**例10-1 HelloWorld.vue的内容**

一个标准的vue文件包含HTML、JS 和CSS三个部分。Vue CLI采用关注点分离的开发方式，更适合组件化的开发。script标签中的内容为JS下的Vue组件。template标签中的内容为HTML下的组件的DOM 结构。style标签中的内容为CSS 样式表，scoped属性表示CSS的作用域仅限当前组件。在每个vue文件的<script>中，都会存在export default {}以导出Vue组件，方便其他文件引用这部分代码。

在script标签中可以通过箭头表示函数，好处是强制定义了作用域，可以避免很多由作用域产生的问题。

### 10.2.2导入import

import可以用于导入外部代码或包，例如在路由中，就通过import { createRouter, createWebHashHistory } from 'vue-router'与import Home from '../views/Home.vue'分别导入了vue-router中的方法以及home组件。

在from后的字符串开始处添加＠符号，可以在本地文件系统中引入文件。＠代表源代码目录，一般情况下是src文件夹。除了@，也会使用../或./等相对路径的方式引入。

## 10.3打包与部署

### 10.3.1项目打包

使用一下命令可以对项目内容进行打包：

npm run build

在执行完命令后，会生成dist文件夹，在前面已经介绍过，如图10-2所示。应用模式是默认的构建模式，这时index.html会带有注入的资源和resource hint，第三方库会被分到一个独立包以便更好的缓存，小于4kb的静态资源会被内联在JavaScript中，public 中的静态资源会被复制到输出目录中。除此之外，还可以通过--target选项指定不同的构建目标，允许将相同的源代码根据不同的用例生成不同的构建。

除了应用模式，还可以使用库模式，通过下面的命令可以将一个单独的入口构建为一个库：

npm run build --target lib --name myLib [entry]

这个入口可以是一个JS或一个Vue文件。如果没有指定入口，则会使用src/App.vue。

需要注意的是，在库模式中Vue是外置的，因此即便在代码中导入了Vue，包中也不会有Vue。如果这个库会通过一个打包器使用，它将尝试通过打包器以依赖的方式加载 Vue，否则就会回退到一个全局的Vue变量。可以在build命令中添加--inline-vue选项以避免这种情况。

类似库模式，同样可以通过选项构建Web Components版本，可以通过下面的命令将一个单独的入口构建为一个 Web Components 组件：

npm run build --target wc --name my-element [entry]

这里的入口应该是一个Vue文件。Vue CLI会把这个组件自动包裹并注册为Web Components组件，不再需要在main.js里注册。在Web Components模式中，Vue同样是外置的，包会假设在页面中已经有一个可用的全局变量Vue。

如果要在Web Components组件的目标中使用vuex ，还需要在App.vue中初始化仓库store，如例10-2所示。

|  |
| --- |
| import store from './store'  export default {  store,  name: 'App',  } |

**例10-2 在Web Components中使用vuex**

### 10.3.2项目部署

对于项目的部署，如果使用Vue CLI处理静态资源并和后端框架一起部署，那么只需要确保生成的构建文件在正确的位置即可。

如果独立于后端，单独部署前端应用。那么可以将dist目录里构建的内容部署到服务器中，并确保正确的publicPath。publicPath是部署应用包时的基本URL，用法等同于webpack的output.publicPath，但Vue CLI在一些其他地方也需要用到这个值，所以应始终使用publicPath。

当使用路由的history模式时，需要注意此时的后台配置。虽然通过history.pushState API可以实现无须重新加载页面的URL跳转，但是如果是单页客户端应用，用户直接访问URL时就会返回404。所以还需要在服务端增加一个覆盖所有情况的候选资源。当URL匹配不到静态资源时返回同一个index.html页面，也就是app依赖的页面。如例10-3所示。

|  |
| --- |
| // nginx  location / {  try\_files $uri $uri/ /index.html;  }  //原生 Node.js  const http = require('http')  const fs = require('fs')  const httpPort = 80  http.createServer((req, res) => {  fs.readFile('index.html', 'utf-8', (err, content) => {  if (err) {  console.log('We cannot open "index.html" file.')  }  res.writeHead(200, {  'Content-Type': 'text/html; charset=utf-8'  })  res.end(content)  })  }).listen(httpPort, () => {  console.log('Server listening on: http://localhost:%s', httpPort)  }) |

**例10-3 使用根目录时的后端配置方法**

### 10.3.3通过Github Action自动部署

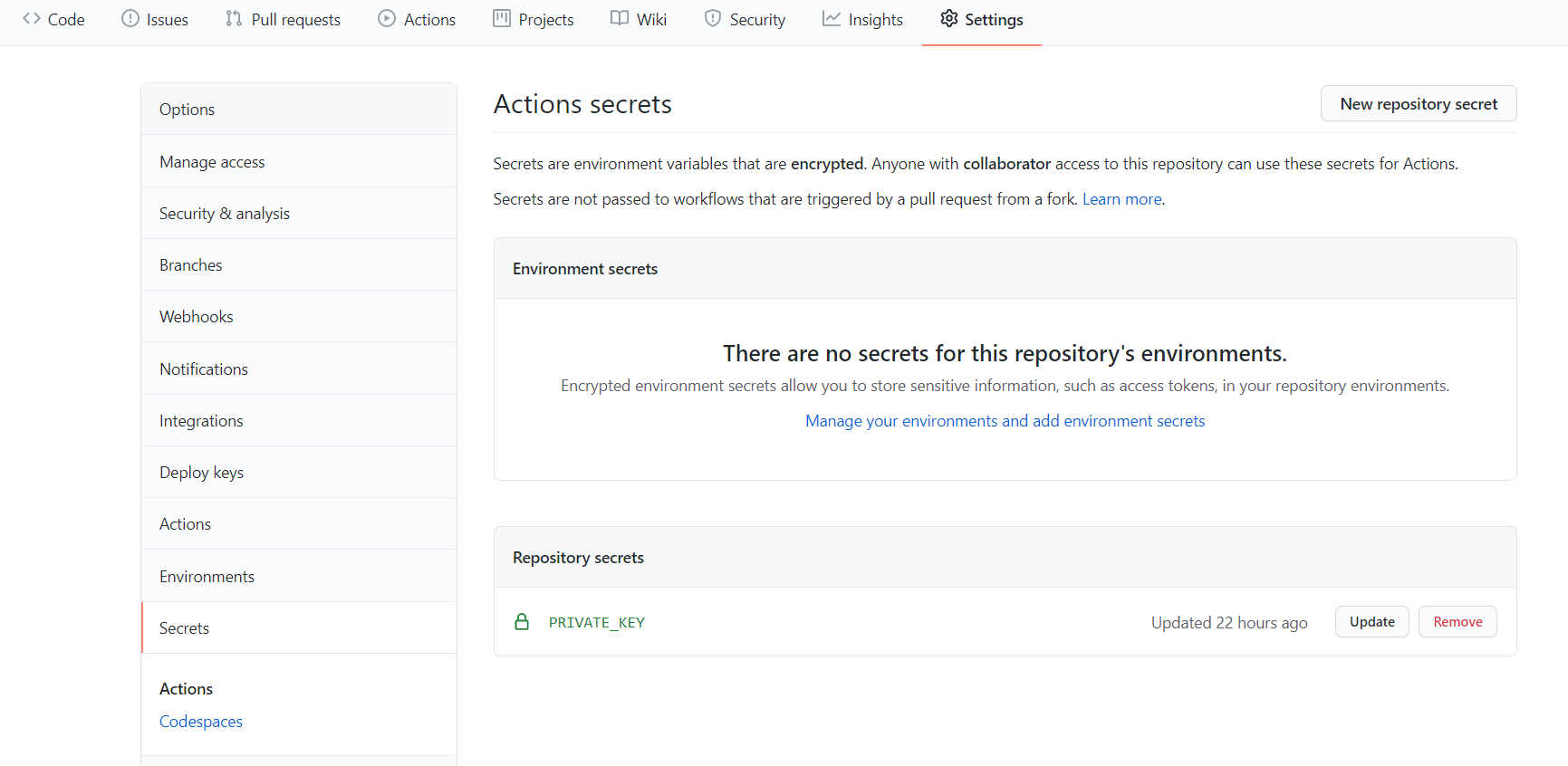
在传统的部署流程中，需要在打包项目后，上传dist文件夹到服务器，再配置好nginx。如果代码经过了多次修改，这样的过程难免会显得有些繁琐，希望寻求自动化的方式处理这一过程。通过Github Actions可以实现代码同步与部署的自动过程，这里以Ubuntu系统的nginx服务器为例介绍配置方法。

首先需要在项目中配置GitHub Actions的workflow。在项目根目录下创建.github/workflows文件夹，并在其中创建一个yml文件，用于编写action的步骤，yml文件中的内容如例10-3所示。

|  |
| --- |
| name: 打包应用并部署  on:  push:  branches:  - main  jobs:  build:  runs-on: ubuntu-20.04  steps:  - name: 获取代码  uses: actions/checkout@main    - name: 安装node  uses: actions/setup-node@v1  with:  node-version: 14.0.0  - name: 安装依赖  run: npm install  - name: 项目打包  run: npm run build  - name: 发布到服务器  uses: easingthemes/ssh-deploy@v2.1.1  env:  # 私钥  SSH\_PRIVATE\_KEY: ${{ secrets.PRIVATE\_KEY }}  # scp参数  ARGS: "-avz --delete"  # 源目录  SOURCE: "dist"  # 服务器ip（写入自己服务器的IP）  REMOTE\_HOST: " "  # 用户  REMOTE\_USER: "root"  # 目标地址  TARGET: "/root/vue-web" |

**例10-4 yml文件中编写GitHub Action**

其中，branches指定了触发action的分支，当代码被推送入main分支时，将会触发action开始运行。runs-on指定了job任务运行时的虚拟机环境，是必填字段，也可以选择例如-latest的版本选项。使用action库中的actions/checkout获取源码，推荐安装大于14.0.0版本的node.js。出于保密与安全性的考虑，不建议直接将私钥写在yml文件中，可以通过在GitHub项目中的secrets填写私钥的方式，在部署过程中通过GitHub获取。



**图10-4 在GitHub项目中设置私钥**

在GitHub的项目页面，单击 Settings→Secrets→New repository secret 按钮，填写名称与私钥内容并保存。私钥可从~/.ssh/id\_rsa文件中获取，名称则需与yml文件中的保持一致。

之后，登录服务器，并确保已经完成nginx的配置，进入/etc/nginx/sites-enabled文件夹，新建一个项目配置文件，以vue-web为例，在其中添加配置，如例10-4所示。

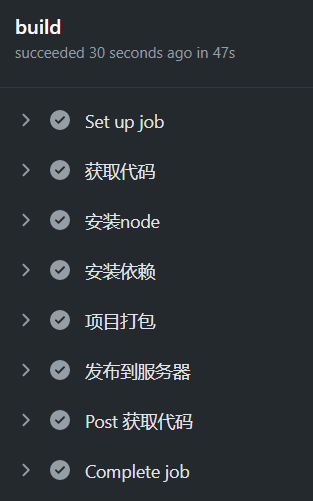
|  |
| --- |
| server {  listen 8080;  server\_name ip;  location / {  root /root/vue-web/dist/;  index index.html index.htm;  }  } |

**例10-5 配置vue-web**

其中，需要将ip替换为服务器ip，并确保root的地址与yml文件中的TARGET保持一致。最后，使用以下代码重启nginx：

nginx -s reload

通过git push指令推送代码到main分支后，将会触发workflow，进入项目的Actions页面可以查看具体状态，如图10-5所示。



**例10-5 等待GitHub Action完成工作**

进入服务器中TARGET对应的目录，可以看到dist中文文件已经被部署到服务器中。由于此处只上传了Vue初始项目，dist中的内容较少，文件夹中的内容如图10-6所示，其中js文件夹的内容如图10-7所示。



**例10-6 dist文件夹中的内容**



**例10-7 js文件夹中的内容**

完成之后，就可以通过服务器ip地址的8080端口访问网站了。

## 10.4本章小结

本章介绍了项目的目录结构、前端开发、项目的打包与部署。

在项目目录中，主要的文件夹有dist、src、public、node\_modules等。dist文件夹是项目构建后的产物，项目完成后，交付dist即可。src则是开发中主要使用的文件夹，其中有存放资源的assets文件夹、存放视图和组件的components文件夹、配置前端路由的router/index.js等。node modules文件夹下保存了项目使用的第三方包。public文件夹提供了一个应急手段，可以将静态资源不经过webpack，只进行简单复制。当需要动态引用图片或库与webpack不兼容时将会十分有效。

在部署项目时，需要确保正确的publicPath。对于使用history模式的路由还需要注意在服务端配置候选资源以避免404的情况。

在下一章中，将会完整地完成一个实际的Vue 3网站开发。

## 10.5练习题

* **问答题：**

1. 项目打包后生成的文件夹是哪个？部署时需要交付哪些文件夹？
2. 简要说明src文件夹中各个文件以及文件夹的作用。
3. 打包时如何指定不同的构建目标？