河南中医药大学信息技术学院（智能医疗行业学院）智能医学工程专业《互联网医疗服务开发》课程

第09章：语法与指令

冯顺磊

河南中医药大学信息技术学院（智能医疗行业学院）
河南中医药大学信息技术学院智能医疗教研室
https://aitcm.hactcm.edu.cn
2025/4/11

本章概要

- 概述
- 创建Vue应用
- 语法
- 指令

1. 概述 1.1 Vue

- Vue是一个用于构建用户界面的渐进式 JavaScript 框架，基于标准 HTML、CSS 和 JavaScript 构建，提供了一套声明式的、组件化的编程模型。它采用了数据驱动和组件化的开发理念，通过简洁灵活的 API，帮助开发者高效地创建交互式的 Web 应用。

1. 概述 1.1 Vue

```
import { createApp, ref } from 'vue'

createApp({
  setup() {
    return {
      count: ref(0)
    }
  }
}).mount('#app')
```

```
<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

- 声明式渲染：Vue 基于标准 HTML 拓展了一套模板语法，使得可以声明式地描述最终输出的 HTML 和 JavaScript 状态之间的关系。
- 响应性：Vue 会自动跟踪 JavaScript 状态并在其发生变化时响应式地更新 DOM。

1. 概述 1.1 Vue

- Vue的发展历史
  - 初始阶段（2013-2014）
    - 2013年：Vue.js 由尤雨溪创建，提供一个轻量级、易用的框架。
    - 2014年：Vue.js 正式发布。
  - 快速发展（2015-2016）
    - 2015年：Vue.js 1.0发布，引入了虚拟 DOM 和响应式系统。
    - 2016年：Vue.js 2.0发布，进一步优化了性能，并改进了开发体验。
  - 生态扩展（2017-2018）
    - 2017年：Vue.js 社区迅速壮大，涌现了大量工具和库，如：Vuex、Vue Router。
    - 2018年：Vue CLI 3.0 发布，提供了更强大的项目脚手架和插件系统。
  - 持续优化（2019-2020）
    - 2019年：Vue.js 3.0 进入开发阶段，引入了 Composition API 等新特性。
    - 2020年：Vue.js 3.0 正式发布，带来了性能提升和更好的TypeScript支持。
  - 现代发展（2021-至今）
    - Vue 3 生态逐步完善，Vite 成为推荐的构建工具， 继续保持稳定更新。

1. 概述 1.1 Vue

- 特点
  - 易用性：若已掌握 HTML、CSS 和 JavaScript，可以很快上手 Vue。Vue 的核心库只关注视图层，易于理解和集成。
  - 灵活性：Vue 是渐进式的，可以根据项目需求逐步采用它的功能。既可以用它构建简单的交互组件，也可以将其集成到大型项目中。
  - 高效性：Vue 采用虚拟 DOM（Virtual DOM）技术，它能在数据发生变化时高效地更新实际的 DOM。虚拟 DOM 是一种轻量级的 JavaScript 对象，是真实 DOM 的抽象表示，通过对比新旧虚拟 DOM 的差异，只更新需要更新的真实 DOM 部分，从而提高渲染效率。
  - 响应式数据绑定：Vue 的响应式系统允许声明式地将数据绑定到 DOM 上。当数据发生变化时，DOM 会自动更新。
  - 组件化开发：Vue 允许将应用拆分成多个小的、可复用的组件。每个组件都有自己的模板、样式和逻辑，使得代码更易于维护和测试。

1. 概述 1.1 Vue

- 生态系统
  - **Vue Router**：Vue 官方的路由管理器，用于实现单页面应用（SPA）的路由功能。
  - **Vuex**：专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。
  - **Vue CLI**：Vue 官方提供的快速项目搭建工具，能帮助你快速创建一个基于 Vue 的项目模板。
  - **Vue Test Utils**：Vue 官方的测试工具库，用于编写和运行 Vue 组件的单元测试。
- 应用场景
  - 单页面应用（SPA）：借助 Vue Router 和 Vuex，能构建出高性能、用户体验良好的单页面应用。
  - 移动端应用：结合 Vue 和一些移动端开发框架（如 Weex、Uni - App），可以开发跨平台的移动端应用。
  - 数据可视化：Vue 和一些图表库（如 ECharts）结合，能够方便地实现数据可视化。

1. 概述 1.1 Vue

- React
  - React 是由 Facebook 开发并维护的一个用于构建用户界面的 JavaScript 库，它采用虚拟 DOM 和组件化开发的思想，极大地提高了前端开发的效率和可维护性。
  - 特点
    - 虚拟 DOM：React 使用虚拟 DOM 来提高渲染效率。虚拟 DOM 是真实 DOM 的抽象表示，通过对比新旧虚拟 DOM 的差异，只更新需要更新的真实 DOM 部分，从而减少了 DOM 操作的次数。
    - 组件化开发：React 将界面拆分成多个小的、可复用的组件。每个组件都有自己的状态和逻辑，使得代码的可维护性和可测试性大大提高。
    - 单向数据流：React 采用单向数据流的设计，数据的流动是单向的，易于理解和调试。
    - 生态系统丰富：React 拥有庞大的生态系统，有许多第三方库和工具可供选择，如 Redux 用于状态管理、React Router 用于路由管理等。

1. 概述 1.1 Vue

- Angular
  - Angular 是由 Google 开发和维护的一个完整的前端框架，它提供了一套完整的解决方案，包括路由、表单处理、HTTP 客户端等。
  - 特点
    - 模块化开发：Angular 使用模块来组织代码，每个模块包含一组相关的组件、指令、管道和服务等。模块化开发使得代码结构清晰，易于维护和扩展。
    - 双向数据绑定：Angular 支持双向数据绑定，即数据的变化会自动反映到视图上，视图的变化也会自动更新数据。
    - 依赖注入：Angular 使用依赖注入来管理组件和服务之间的依赖关系，提高了代码的可测试性和可维护性。
    - TypeScript 支持：Angular 使用 TypeScript 作为开发语言，TypeScript 是 JavaScript 的超集，提供了静态类型检查和更好的代码提示功能。

1. 概述 1.1 Vue

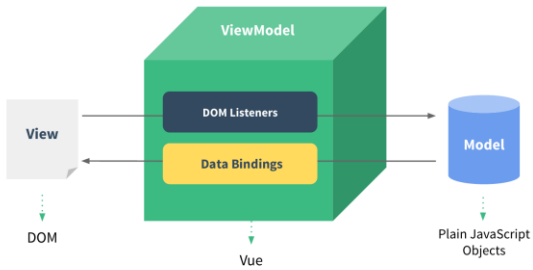
- Vue vs React vs Angular

对比维度	Vue	React	Angular
学习曲线	较为平缓。核心概念易理解，模板语法类似 HTML，官方文档详细且中文资料丰富，上手快	相对较陡。需适应 JSX 语法，生态系统庞大，额外学习成本高	最陡。概念和特性多，使用 TypeScript，学习成本高
性能	采用虚拟 DOM 和高效 Diff 算法，性能出色，响应速度快	使用虚拟 DOM 和 Diff 算法，大型项目中因不可变数据思想有性能开销，可优化	使用脏检查机制，大型应用性能较弱，有优化选项
生态系统	不断发展壮大，有官方路由和状态管理库，还有众多第三方库和插件	非常庞大成熟，有丰富的第三方库和工具，UI 组件库众多	完善，由 Google 维护，提供完整解决方案，有第三方库
灵活性	渐进式框架，灵活性高，可按需引入功能	灵活性高，专注视图层，可自由选择配套库	全功能框架，灵活性低，但规范性和一致性强
社区支持	社区活跃，官方文档详细，国内有很多中文资源	社区庞大活跃，GitHub 开源项目多，社区解答丰富	由 Google 支持，社区稳定，有专业开发者分享经验

国内使用最多的是Vue

1. 概述 1.2 MVVM

- MVVM（Model-View-ViewModel ），是一种基于前端开发的架构模式，由数据模型（Model）、视图（View）和视图数据模型（ViewModel）组成，实现数据模型与界面的分离，提升代码解耦性、可测试性和可维护性。MVVM支持双向数据绑定、事件驱动和命令绑定，简化视图与模型的交互。
  - 数据模型（Model）：负责存储应用程序的数据，独立于视图（View）和视图模型（ViewModel），可在不同的视图间共享。
  - 视图（View）：用户界面，负责数据的展示。
  - 视图数据模型（ViewModel）：负责处理View和Model之间的交互。

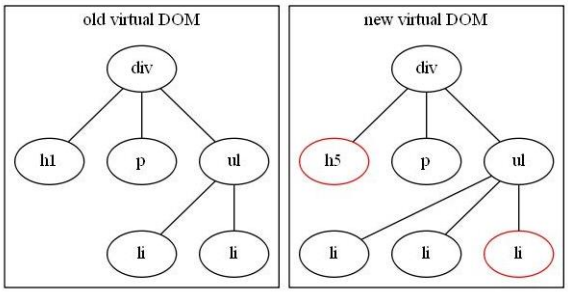


1. 概述 1.2 MVVM

- 工作原理
  - MVVM 模式的核心是数据绑定和视图模型的双向数据绑定机制。当 Model 层的数据发生变化时，ViewModel 会自动感知到这些变化，并将更新后的数据反映到 View 上，实现视图的自动更新；而当用户在 View 上进行操作（如点击按钮、输入文本等）时，这些操作会触发 ViewModel 中的相应方法，进而对 Model 层的数据进行修改。
- 优点
  - 分离关注点：MVVM 模式将视图和业务逻辑分离，使得代码的可维护性和可测试性大大提高。开发人员可以专注于业务逻辑的实现，而设计人员可以专注于界面的设计。
  - 数据绑定：双向数据绑定机制使得数据的变化能够自动反映到视图上，减少了手动操作 DOM 的代码，提高了开发效率，同时也降低了出错的概率。
  - 可测试性：由于 ViewModel 与 View 是分离的，因此可以方便地对 ViewModel 进行单元测试，确保业务逻辑的正确性。
- 缺点
  - 学习曲线较陡：对于初学者来说，理解 MVVM 模式的概念和双向数据绑定机制可能需要花费一定的时间和精力。
  - 性能开销：双向数据绑定机制需要不断地监听数据的变化，这可能会带来一定的性能开销，尤其是在处理大量数据时。

1. 概述 1.3 虚拟DOM

- 虚拟 DOM (Virtual DOM，VDOM) 是将状态映射成视图的一种解决方案，工作原理是使用状态生成虚拟节点，使用虚拟节点渲染视图，它是现代前端框架（如 React、Vue 等）中的一个重要概念。
- 虚拟 DOM 提供了一种管理 DOM （Document Object Model，文档对象模型，是 Web 上构成文档结构和内容的对象的数据表示，作为一个面向对象的网页表示，可以用脚本语言修改。）的策略。与直接创建 DOM 节点不同，Vue 组件会生成 DOM 节点的描述，这些描述符是普通的 JavaScript 对象，称为虚拟 DOM 节点 (Virtual Node， VNode)。每次组件重新渲染时，都会将新的 VNode 树与先前的VNode 树进行比较，然后将它们之间的差异应用于真实 DOM。如果没有任何更改，则不需要修改 DOM。



1. 概述 1.3 虚拟DOM

- 优势
  - 提高性能：直接操作真实 DOM 的代价较高，因为每次操作都会触发浏览器的重排和重绘。而虚拟 DOM 通过差异计算和批量更新，可以减少对真实 DOM 的操作次数，从而提高性能。
  - 跨平台：由于虚拟 DOM 是一个 JavaScript 对象，它可以在不同的平台上进行渲染，例如浏览器、服务器端（SSR）、移动端等。
  - 方便测试：虚拟 DOM 使得代码的测试更加容易，因为可以直接对 JavaScript 对象进行测试，而不需要依赖真实的 DOM 环境。
- 工作流程
  - 初始化：在应用初始化时，框架会根据组件的状态生成一个虚拟 DOM 树，并将其渲染到真实的 DOM 上。
  - 状态更新：当应用的状态发生变化时，框架会重新生成一个新的虚拟 DOM 树。
  - 差异比较：使用算法（如 Diff 算法）对比新旧虚拟 DOM 树，找出它们之间的差异。
  - 更新真实 DOM：将差异部分批量应用到真实的 DOM 上，完成页面的更新。

1. 概述 1.4 数据绑定

- 数据绑定是Vue.js的核心特性之一，将数据与视图进行关联，当数据发生变化时，视图会自动更新，实现数据与视图的同步，提高前端开发的效率和用户体验。
- 在Vue.js 3 中，数据绑定采用了一种名为“响应式”的机制来实现，当开发者在Vue.js 3 中声明一个属性时，Vue.js 3 会使用Proxy对象将该属性转换为响应式对象。Vue.js 3 中的数据绑定有两种绑定方式：单向数据绑定、双向数据绑定。

1. 概述 1.4 数据绑定

- 单项数据绑定
  - 单向数据绑定指的是数据的流动是单向的，即数据从数据源（通常是 Vue 实例中的 data 属性）流向视图。当数据源中的数据发生变化时，视图会自动更新以反映这些变化；但视图上的操作不会直接影响数据源中的数据。
  - 在 Vue 里，常见的单向数据绑定实现方式有插值表达式和 v-bind 指令。
    - 插值表达式：使用双大括号 {{ }} 将数据插入到 HTML 模板中。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>插值表达式单向数据绑定示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <p>{{ message }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                message: 'Hello, Vue!'
            }
        });
    </script>
</body>

</html>
```

1. 概述 1.4 数据绑定

- 单项数据绑定
  - v-bind 指令：用于将表达式的值绑定到 HTML 元素的属性上，可简写为“:”。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-bind单向数据绑定示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <img :src="imageUrl" alt="示例图片">
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                imageUrl: 'https://picsum.photos/200/300'
            }
        });
    </script>
</body>

</html>
```

  - 单向数据绑定适用于数据展示类的场景，比如展示商品信息、文章内容等，只需将数据呈现给用户，而不需要用户对数据进行修改。

1. 概述 1.4 数据绑定

- 双向数据绑定
  - 双向数据绑定意味着数据在数据源和视图之间可以双向流动。当数据源中的数据发生变化时，视图会更新；当用户在视图上进行操作（如输入文本、选择选项等）时，数据源中的数据也会相应更新。
  - 在 Vue 中，使用 v-model 指令来实现双向数据绑定，它主要用于表单元素（如 input、textarea、select 等）。
  - 双向数据绑定适用于需要用户输入数据并实时更新数据源的场景，如表单填写、用户信息编辑等。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model双向数据绑定示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input v-model="message" type="text">
        <p>输入的内容是：{{ message }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                message: ''
            }
        });
    </script>
</body>

</html>
```

1. 概述 1.5 API风格

- Vue 的组件可以按两种不同的风格书写：
  - 选项式 API (Options API)
  - 选项式 API 把不同功能的代码放在不同选项里，像 data 用于定义数据，methods 用于定义方法，computed 用于定义计算属性等。
  - 适用场景
    - 小型项目：对于功能简单、代码量少的小型项目，选项式 API 的结构清晰，易于理解和上手。
    - 初学者：选项式 API 的语法简单，对于刚接触 Vue 的开发者来说，更容易理解和掌握。
  - 优缺点
    - 优点：代码结构清晰，易于理解和维护，适合初学者快速上手。
    - 缺点：当组件变得复杂时，同一个逻辑关注点的代码会分散在不同的选项中，导致代码的可读性和可维护性下降。

```vue
<script>
export default {
  // data() 返回的属性将会成为响应式的状态
  // 并且暴露在 `this` 上
  data() {
    return {
      count: 0
    }
  },

  // methods 是一些用来更改状态与触发更新的函数
  // 它们可以在模板中作为事件处理器绑定
  methods: {
    increment() {
      this.count++
    }
  },

  // 生命周期钩子会在组件生命周期的各个不同阶段被调用
  // 例如这个函数就会在组件挂载完成后被调用
  mounted() {
    console.log(`The initial count is ${this.count}.`)
  }
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```


1. 概述 1.5 API风格

- Vue 的组件可以按两种不同的风格书写：
  - 组合式 API (Composition API)
  - 组合式 API 将相关的逻辑代码组合在一起，而不是像选项式 API 那样按选项来组织。它通过 setup 函数来实现，setup 函数是一个新的组件选项，在组件创建之前执行。
  - 适用场景
    - 大型项目：对于功能复杂、代码量较大的大型项目，组合式 API 可以更好地组织代码，提高代码的可读性和可维护性。
    - 逻辑复用：当需要在多个组件中复用逻辑时，组合式 API 是更好的选择。
  - 优缺点
    - 优点：逻辑更加清晰，方便代码复用和维护，提高了代码的可测试性。
    - 缺点：学习曲线相对较陡，对于初学者来说可能需要花费更多的时间来理解和掌握。

```vue
<script setup>
import { ref, onMounted } from 'vue'

// 响应式状态
const count = ref(0)

// 用来修改状态、触发更新的函数
function increment() {
  count.value++
}

// 生命周期钩子
onMounted(() => {
  console.log(`The initial count is ${count.value}.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

2. 创建Vue应用

- 创建一个 Vue 应用
  - 步骤1：下载并安装Node.js
  - 步骤2：下载并安装Visual Studio Code
  - 步骤3：创建Vue代码目录
  - 步骤4：在终端中执行“npm create vue@latest”创建Vue项目
  - 步骤5：在终端中执行“npm install”命令，为Vue项目安装依赖
  - 步骤6：在终端中执行“npm run dev”命令，运行Vue项目
  - 步骤7：在Visual Studio Code创建HelloWorld.vue文件并完成对应编码开发。
  - 步骤8：在浏览器中查看页面。

2. 创建Vue应用

- 通过 CDN 使用 Vue
  - 步骤1：创建Vue代码目录
  - 步骤2：新建index.html文件
  - 步骤3：使用 script 标签通过 CDN 来使用 Vue
  - 步骤4：完成对应Vue编码开发
  - 步骤5：在浏览器中查看页面

3. 语法 3.1 模板语法

- Vue使用了一种基于HTML的模板语法，使用户能够声明式地将应用或组件实例的数据绑定到呈现的DOM上。
- 模板中除了HTML的结构，还包含两个Vue模板的特有语法：插值和指令。
  - 插值语法只有一个功能，就是向标签体中插入一个动态的值。
  - 指令语法用来操作其所在的标签，比如指定动态属性值、绑定事件监听、控制显示隐藏等。
- 文本插值
  - 文本插值是将组件中的数据动态地显示在模板的文本内容中，使用“{{ }}” 实现文本插值，将数据动态显示在模板中。

```
<span>{{ message }}</span>
```


3. 语法 3.1 模板语法

- 指令语法
  - 指令的作用是在其表达式的值变化时响应式地更新 DOM。
  - 其中，“Name”代表指令名称，“Argument”代表参数，“Modifiers”代表修饰符，“Value”代表JavaScript表达式。

3. 语法 3.1 模板语法

- 原始HTML
  - 原始HTML是指插入HTML片段到HTML元素内部，Vue中使用“v-html”指令将数据绑定为HTML片段，并将其插入到元素内部。

```
<p>使用 v-html 指令：<span v-html="rawHtml"></span></p>
```


3. 语法 3.1 模板语法

- 绑定属性
  - 绑定属性使用“v-bind”指令，将元素的值绑定为变量。例如：使用“v-bind:id=“dynamicId”””，“v-bind”指令将元素id属性的值绑定为“dynamicId”变量。

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

```
<div :id="dynamicId"></div>
```

3. 语法 3.1 模板语法

- 布尔型属性
  - 布尔型属性是依据true/false值来决定属性是否在该元素上。例如：使用“:disabled=“isButtonDisabled”””，当“isButtonDisabled”为true时元素会包含“disabled”属性，为false时“disabled”将被忽略。

```
<button :disabled="isButtonDisabled">Button</button>
```


3. 语法 3.1 模板语法

- 动态绑定多个值
  - 使用对象实现把多个属性绑定到单个元素上。例如：定义一个“objectOfAttrs”对象。

```
const objectOfAttrs = {
  id: 'container',
  class: 'wrapper',
  style: 'background-color:green'
}
```

```
<div v-bind="objectOfAttrs"></div>
```

3. 语法 3.1 模板语法

- 使用 JavaScript 表达式：Vue 在所有的数据绑定中都支持完整的 JavaScript 表达式。
- 每个绑定仅支持单一表达式，也就是一段能够被求值的 JavaScript 代码。一个简单的判断方法是是否可以合法地写在 return 后面。

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

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

<div :id="`list-${id}`"></div>
```

示例9-3-1：模板语法

3. 语法 3.2 响应式基础

- 响应式数据绑定是Vue.js开发中一项重要的技术，用于实现页面数据的实时更新，当数据发生变化时，页面能够自动更新，而不需要手动重新渲染。
- Vue.js中主要使用了 Proxy对象来实现响应式，提供了四个函数定义响应式数据，分别是“ref()”函数、“reactive()”函数、“toRef()”函数、“toRefs()”函数。

3. 语法 3.2 响应式基础

- ref()函数
  - ref()函数用于将数据转换成响应式数据。调用ref并传入一个参数时，返回一个响应式的引用对象。例如：在Vue.js 3中使用ref()函数来创建响应式数据，并在页面加载后的某个时间点更新这些数据。

```vue
<template>
    <!-- 输出message响应式数据 -->
    {{ message }}
</template>

<script setup>
import { ref } from 'vue';

const message = ref("不积跬步，无以至千里。不积小流，无以成江河")
// 在页面加载完成后3秒修改响应式数据内容
setTimeout(() => message.value = '会当凌绝顶，一览众山小' , 3000)

</script>
<style>
```


3. 语法 3.2 响应式基础

- reactive()函数
  - reactive()函数用于创建响应式的对象或数组，通过reactive()函数传递一个对象或数组作为参数时，返回一个代理对象（或代理数组），该代理对象（或数组）的所有属性（或索引）都将变为响应式的。

```vue
<template>
    {{ obj.message }}
</template>
<script setup>
import { reactive } from 'vue'
const obj = reactive({ message: '穷且益坚，不坠青云之志' })

setTimeout(() => obj.message = '人有逆天之时，天无绝人之路', 3000)
```

3. 语法 3.2 响应式基础

- toRef()函数
  - toRef()函数用于将响应式对象中的单个属性转换为响应式数据。

```vue
<template>
  <div>message的值：{{ message }}</div>
  <div>obj.message的值：{{ obj.message }}</div>
</template>
<script setup>
import { reactive, toRef } from "vue";

const obj = reactive({ message: "三更灯火五更鸡，正是男儿读书时" });

const message = toRef(obj, "message");
// 延时3秒修改响应式对象属性
setTimeout(() => (message.value = "咬定青山不放松，立根原在破岩中"), 3000);
</script>
```


3. 语法 3.2 响应式基础

- toRefs()函数
  - toRefs()函数用于将响应式对象中的所有属性转换为ref对象。

```vue
<template>
    <div>message的值：{{ message }}</div>
    <div>obj.message的值：{{ obj.message }}</div>
</template>
<script setup>
import { reactive, toRefs } from 'vue'

const obj = reactive({ message: '盛年不重来，一日难再晨' })

let { message } = toRefs(obj)

setTimeout(() => message.value = '及时当勉励，岁月不待人', 3000)
</script>
```

示例9-3-2：响应式基础

3. 语法 3.3 计算属性

- 在模板中直接表达式虽然方便，但在模板中写太多逻辑，会让模板难以维护。
- 例如：一个包含嵌套数组的“author”对象，需求是根据“author”是否有书籍来展示不同的信息，计算依赖于“author.books”需要阅读对象后才能辨别其作用。更重要的是，如果在模板中需要不止一次这样的计算，这样的代码将在模板里重复好多遍。

```
const author = reactive({
  name: 'John Doe',
  books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
    'Vue 4 - The Mystery'
  ]
})

<p>Has published books:</p>
<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
```


3. 语法 3.3 计算属性

- 推荐使用计算属性来描述依赖响应式状态的复杂逻辑。将上述示例用计算属性重构，定义一个计算属性“publishedBooksMessage”，使用“computed()”方法实现。
- 在表达式中像这样调用一个函数也会获得和计算属性相同的结果。

```
// 组件中
function calculateBooksMessage() {
  return author.books.length > 0 ? 'Yes' : 'No'
}

<p>{{ calculateBooksMessage() }}</p>
```

- 两种方式在结果上确实是完全相同的，然而，不同之处在于计算属性值会基于其响应式依赖被缓存。一个计算属性仅会在其响应式依赖更新时才重新计算。

```
<template>
  <p>Has published books:</p>
  <span>{{ publishedBooksMessage }}</span>
</template>

<script setup>
import { reactive, computed } from 'vue'

const author = reactive({
  name: 'John Doe',
  books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
    'Vue 4 - The Mystery'
  ]
})

// 定义一个名为publishedBooksMessage的计算属性
const publishedBooksMessage = computed(() => {
  return author.books.length > 0 ? 'Yes' : 'No'
})
</script>
```

示例9-3-3：计算属性

3. 语法 3.4 侦听器

- 侦听器用于监听响应式数据的变化，并在数据变化时执行相应的回调函数。
  - 选项式 API 中的“watch”选项
    - 监听单个数据源：可以直接指定要监听的数据属性，并提供一个回调函数来处理变化。

```
export default {
  data() {
    return {
      count: 0
    };
  },
  watch: {
    count(newValue, oldValue) {
      // 当 count 变化时执行的逻辑
      console.log(`count 从 ${oldValue} 变为 ${newValue}`);
    }
  }
};
```


3. 语法 3.4 侦听器

- 侦听器用于监听响应式数据的变化，并在数据变化时执行相应的回调函数。
  - 选项式 API 中的“watch”选项
    - 监听多个数据源：可以通过对象形式同时监听多个数据属性。

```
export default {
   data() {
     return {
       count: 0,
       name: ''
     };
   },
   watch: {
     count: function(newValue, oldValue) {
       // 处理 count 变化的逻辑
     },
     name: function(newValue, oldValue) {
       // 处理 name 变化的逻辑
     }
   }
 };
```

3. 语法 3.4 侦听器

- 侦听器用于监听响应式数据的变化，并在数据变化时执行相应的回调函数。
  - 选项式 API 中的“watch”选项
    - 深度监听对象属性：对于对象类型的数据，如果要深度监听其内部属性的变化，可以设置“deep: true”属性。

```
export default {
    data() {
      return {
        user: {
          name: '',
          age: 0
        }
      };
    },
    watch: {
      user: {
        handler(newValue, oldValue) {
          // 深度监听 user 对象的变化
        },
        deep: true
      }
    }
  };
```


3. 语法 3.4 侦听器

- 侦听器用于监听响应式数据的变化，并在数据变化时执行相应的回调函数。
  - 选项式 API 中的“watch”选项
    - 立即执行侦听器：watch 默认是懒执行的，仅当数据源变化时，才会执行回调。通过设置“immediate: true”属性，可以让侦听器在组件创建时立即执行一次。

```
export default {
  // ...
  watch: {
    question: {
      handler(newQuestion) {
        // 在组件实例创建时会立即调用
      },
      // 强制立即执行回调
      immediate: true
    }
  }
  // ...
}
```

3. 语法 3.4 侦听器

- 侦听器用于监听响应式数据的变化，并在数据变化时执行相应的回调函数。
  - 组合式 API 中的“watch”选项
    - 监听单个数据源

```
import { ref, watch } from 'vue';

  export default {
    setup() {
      const count = ref(0);

      watch(count, (newValue, oldValue) => {
        // 处理 count 变化的逻辑
        console.log(`count 从 ${oldValue} 变为 ${newValue}`);
      });

      return { count };
    }
  };
```


3. 语法 3.4 侦听器

- 侦听器用于监听响应式数据的变化，并在数据变化时执行相应的回调函数。
  - 组合式 API 中的“watch”选项
    - 监听多个数据源：可以传递一个数组作为第一个参数来监听多个数据源。

```
import { ref, watch } from 'vue';

  export default {
    setup() {
      const count = ref(0);
      const name = ref('');

      watch([count, name], ([newCount, newName], [oldCount,
oldName]) => {
        // 处理 count 和 name 变化的逻辑
      });

      return { count, name };
    }
  };
```

示例9-3-4：侦听器

3. 语法 3.5 类与样式绑定

- 数据绑定的一个常见需求场景是操纵元素的 CSS class 列表和内联样式。
- 因为 class 和 style 都是 attribute，因此可以和其他 attribute 一样使用 v-bind 将它们和动态的字符串绑定。
- 但是，在处理比较复杂的绑定时，通过拼接生成字符串是麻烦且易出错的。
- 因此，Vue 专门为 class 和 style 的 v-bind 用法提供了特殊的功能增强。除了字符串外，表达式的值也可以是对象或数组。

3. 语法 3.5 类与样式绑定

- class属性绑定：v-bind:class 通过设置一个对象，从而动态的切换 class。

```
<template>
<div class="static" v-bind:class="{ active: isActive }"></div>

//active 是否存在取决于数据属性 isActive 的真假值
//当isActive为真时的渲染结果
<div class="static active"></div>
</template>
<script setup>
import { ref } from "vue";
const isActive=ref(True);
</script>
```

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

//当isActive、hasError为真时渲染结果
<div class="static active text-danger"></div>
</template>
<script setup>
import { ref } from "vue";
const isActive=ref(True);
const hasError=ref(True);
</script>
```

3. 语法 3.5 类与样式绑定

- class属性绑定：v-bind:class 通过设置一个对象，从而动态的切换 class。

```
<template>
<div v-bind:class="[activeClass, errorClass]"></div>

//渲染结果
<div class="active text-danger"></div>
</template>
<script setup>
import { ref } from "vue";
const activeClass = ref('active')
const errorClass = ref('text-danger')
</script>
```


3. 语法 3.5 类与样式绑定

- 绑定内联样式：在Vue中使用v-bind:style 在行内绑定样式。

```
<template>
    <div :style="{ color: activeColor, fontSize: fontSize + 'px'
}"></div>
</template>
<script setup>
import { ref } from "vue";
const activeColor = ref('red')
const fontSize = ref(30)
</script>
```

示例9-3-5：类与样式绑定

4. 指令 4.1 条件渲染指令

- v-if
  - v-if指令能够依据表达式的值来决定是否将元素渲染到 DOM 中。当表达式的值为true时，元素会被渲染；若为false，元素不会被渲染。
- v-else
  - v-else指令必须紧跟在v-if或者v-else-if后面使用，用于表示v-if或者v-else-if条件不满足时要渲染的内容。
- v-else-if
  - v-else-if指令用于在多个条件分支中进行判断，它要紧跟在v-if后面，并且可以有多个v-else-if，最后还能使用v-else作为最终的分支。
- <template> 上的 v-if
  - 因为 v-if 是一个指令，必须依附于某个元素。若想切换不止一个元素时可在 <template> 元素上使用 v-if，这只是一个不可见的包装器元素，最后渲染的结果并不会包含这个 <template> 元素。

4. 指令 4.1 条件渲染指令

- v-show
  - v-show 用于根据表达式的值来控制元素的显示与隐藏。
  - v-show 本质上是通过修改元素的 CSS display 属性来实现显示与隐藏的。当表达式的值为 true 时，元素的 display 属性保持默认值；当表达式的值为 false 时，元素的 display 属性会被设置为 none。

4. 指令 4.1 条件渲染指令

- 与 v-if 的对比
  - 渲染方式：
    - v-if 是 “真正” 的条件渲染，它会根据表达式的值来决定是否将元素渲染到 DOM 中。当条件为 false 时，元素不会被渲染到 DOM 里；当条件变为 true 时，元素才会被渲染到 DOM 中。
    - v-show 不管表达式的初始值是什么，元素都会被渲染到 DOM 中，只是通过修改 display 属性来控制元素的显示与隐藏。
  - 性能开销：
    - v-if 有更高的切换开销。因为每次切换时，元素都会被销毁和重新创建，这涉及到创建和销毁 DOM 节点、初始化和销毁组件实例等操作，性能开销较大。
    - v-show 有更高的初始渲染开销，但切换开销较小。由于元素始终存在于 DOM 中，切换时只需修改 CSS 属性，性能开销相对较小。
  - 编译过程：
    - v-if 是惰性的。如果在初始渲染时条件为 false，则什么也不做，直到条件第一次变为 true 时，才会开始渲染条件块。
    - v-show 没有惰性，元素会在初始渲染时就被渲染到 DOM 中。

4. 指令 4.1 条件渲染指令

- v-show
  - 适用场景
    - v-show 适用于需要频繁切换显示状态的场景，因为它的切换开销较小。例如，在一个页面中，根据用户的操作显示或隐藏某个提示框，使用 v-show 可以提高性能。
    - v-if 适用于在运行时条件很少改变的场景。例如，根据用户的权限显示不同的功能模块，由于权限信息通常不会频繁改变，使用 v-if 可以避免不必要的初始渲染开销。

示例9-4-1：条件渲染指令

4. 指令 4.2 列表渲染指令

- v-for
  - v-for 指令的基本语法是 item in items，其中 items 是要遍历的数组或对象，item 是当前遍历到的元素。你可以使用 (item, index) 来同时获取元素和其索引（对于数组），或者使用 (value, key) 来同时获取对象的属性值和属性名（对于对象）。
- <template> 上的 v-for
  - 与模板上的 v-if 类似，也可在 <template> 标签上使用 v-for 来渲染一个包含多个元素的块。

4. 指令 4.2 列表渲染指令

- v-for
  - 遍历数组

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-for 遍历数组示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js">
</head>

<body>
    <div id="app">
        <ul>
            <!-- 使用 v-for 遍历数组 -->
            <li v-for="(item, index) in items" :key="index">
                {{ index }} - {{ item }}
            </li>
        </ul>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                items: ['苹果', '香蕉', '橙子']
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.2 列表渲染指令

- v-for
  - 遍历对象

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-for 遍历对象示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js">
</head>

<body>
    <div id="app">
        <ul>
            <!-- 使用 v-for 遍历对象 -->
            <li v-for="(value, key) in user" :key="key">
                {{ key }}: {{ value }}
            </li>
        </ul>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                user: {
                    name: '张三',
                    age: 25,
                    gender: '男'
                }
            }
        });
    </script>
</body>

</html>
```


4. 指令 4.2 列表渲染指令

- v-for
  - 维护状态（:key 的使用）

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-for 中 key 的使用示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js">
</head>

<body>
    <div id="app">
        <ul>
            <!-- 为每个列表项提供唯一的 key -->
            <li v-for="item in items" :key="item.id">
                {{ item.name }}
            </li>
        </ul>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                items: [
                    { id: 1, name: '商品1' },
                    { id: 2, name: '商品2' },
                    { id: 3, name: '商品3' }
                ]
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.2 列表渲染指令

- v-for
  - 嵌套使用

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-for 嵌套使用示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <ul>
            <!-- 外层 v-for 遍历父数组 -->
            <li v-for="(parentItem, parentIndex) in parentItems" :key="parentIndex">
                {{ parentItem.name }}
                <ul>
                    <!-- 内层 v-for 遍历子数组 -->
                    <li v-for="(childItem, childIndex) in parentItem.children"
:key="childIndex">
                        - {{ childItem }}
                    </li>
                </ul>
            </li>
        </ul>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                parentItems: [
                    {
                        name: '类别1',
                        children: ['子项1', '子项2']
                    },
                    {
                        name: '类别2',
                        children: ['子项3', '子项4']
                    }
                ]
            }
        });
    </script>
</body>

</html>
```

示例9-4-2：列表渲染指令

4. 指令 4.3 事件处理指令

- v-on：v-on 指令的基本语法是 v-on:事件名="方法名" 或者 v-on:事件名="JavaScript 表达式"。
  - “事件名” 指的是标准的 DOM 事件，像 click、input、submit 等。
  - “方法名” 是在 Vue 实例的 methods 选项里定义的方法。
  - “JavaScript 表达式” 则是可以直接执行的 JavaScript 代码。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-on 基本用法示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <!-- 使用 v-on 绑定 click 事件 -->
        <button v-on:click="handleClick">点击我</button>
    </div>
    <script>
        new Vue({
            el: '#app',
            methods: {
                handleClick() {
                    alert('按钮被点击了！');
                }
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.3 事件处理指令

- 简写形式
  - v-on 指令可以简写成 @，所以 v-on:click 可以写成 @click。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-on 简写形式示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <!-- 使用简写形式绑定 click 事件 -->
        <button @click="handleClick">点击我</button>
    </div>
    <script>
        new Vue({
            el: '#app',
            methods: {
                handleClick() {
                    alert('按钮被点击了！');
                }
            }
        });
    </script>
</body>

</html>
```


4. 指令 4.3 事件处理指令

- 事件修饰符
  - Vue 提供了一些事件修饰符，用于简化事件处理逻辑，常见的事件修饰符有 .stop、.prevent、.capture、.self、.once 等。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-on 事件修饰符示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <!-- 使用 .stop 阻止事件冒泡 -->
        <div @click="outerClick">
            <button @click.stop="innerClick">点击我（阻止冒泡）</button>
        </div>
        <!-- 使用 .prevent 阻止表单提交的默认行为 -->
        <form @submit.prevent="submitForm">
            <input type="submit" value="提交">
        </form>
    </div>
    <script>
        new Vue({
            el: '#app',
            methods: {
                outerClick() {
                    alert('外层 div 被点击了');
                },
                innerClick() {
                    alert('按钮被点击了');
                },
                submitForm() {
                    alert('表单提交被阻止');
                }
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.3 事件处理指令

- 按键修饰符
  - 在监听键盘事件时，Vue 提供了按键修饰符，用于监听特定的按键事件，像 .enter、.tab、.delete、.esc、.space 等。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-on 按键修饰符示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <!-- 使用 .enter 监听回车键事件 -->
        <input @keyup.enter="handleEnter">
    </div>
    <script>
        new Vue({
            el: '#app',
            methods: {
                handleEnter() {
                    alert('回车键被按下了');
                }
            }
        });
    </script>
</body>

</html>
```

示例9-4-3：事件处理器指令

4. 指令 4.4 表单输入指令

- v-model：v-model 是 Vue 中实现双向数据绑定的核心表单输入指令，它可以用在多种表单元素上，包括 input、textarea、select 等。
- 应用于文本输入框（input [type="text"]）

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model 用于文本输入框</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input v-model="message" type="text">
        <p>输入的内容是: {{ message }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                message: ''
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.4 表单输入指令

- 应用于文本域（textarea）

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model 用于文本域</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <textarea v-model="content"></textarea>
        <p>输入的文本是: {{ content }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                content: ''
            }
        });
    </script>
</body>

</html>
```


4. 指令 4.4 表单输入指令

- 应用于单选框（input [type="radio"]）

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model 用于单选框</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input type="radio" id="option1" value="选项1" v-model="selectedOption">
        <label for="option1">选项1</label>
        <input type="radio" id="option2" value="选项2" v-model="selectedOption">
        <label for="option2">选项2</label>
        <p>选择的选项是: {{ selectedOption }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                selectedOption: ''
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.4 表单输入指令

- 应用于复选框（input [type="checkbox"]）
  - 单个复选框

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model 用于单个复选框</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input type="checkbox" v-model="isChecked">
        <label>是否选中</label>
        <p>复选框状态: {{ isChecked }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                isChecked: false
            }
        });
    </script>
</body>

</html>
```


4. 指令 4.4 表单输入指令

- 应用于复选框（input [type="checkbox"]）
  - 多个复选框

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model 用于多个复选框</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input type="checkbox" id="apple" value="苹果" v-model="selectedFruits">
        <label for="apple">苹果</label>
        <input type="checkbox" id="banana" value="香蕉" v-model="selectedFruits">
        <label for="banana">香蕉</label>
        <p>选择的水果是: {{ selectedFruits }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                selectedFruits: []
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.4 表单输入指令

- 应用于下拉选择框（select）

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model 用于下拉选择框</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <select v-model="selectedCity">
            <option value="北京">北京</option>
            <option value="上海">上海</option>
            <option value="广州">广州</option>
        </select>
        <p>选择的城市是: {{ selectedCity }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                selectedCity: ''
            }
        });
    </script>
</body>

</html>
```


4. 指令 4.4 表单输入指令

- v-model 修饰符：v-model 提供了一些修饰符来改变其默认行为。
  - .lazy：默认情况下，v-model 会在每次 input 事件触发时更新数据。使用 .lazy 修饰符后，会转变为在 change 事件触发时更新数据。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model.lazy 示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input v-model.lazy="message" type="text">
        <p>输入的内容是: {{ message }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                message: ''
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.4 表单输入指令

- v-model 修饰符：v-model 提供了一些修饰符来改变其默认行为。
  - .number：如果希望用户输入的数据自动转换为数字类型，可以使用 .number 修饰符。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model.number 示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input v-model.number="age" type="text">
        <p>输入的年龄是: {{ typeof age }}, 值为 {{ age }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                age: 0
            }
        });
    </script>
</body>

</html>
```


4. 指令 4.4 表单输入指令

- v-model 修饰符：v-model 提供了一些修饰符来改变其默认行为。
  - .trim：.trim 修饰符可以自动过滤用户输入内容的首尾空格。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-model.trim 示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <input v-model.trim="inputText" type="text">
        <p>去除首尾空格后的内容是: {{ inputText }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                inputText: ''
            }
        });
    </script>
</body>

</html>
```

示例9-4-4：表单输入指令

4. 指令 4.5 其他指令

- v-pre：跳过这个元素和它的子元素的编译过程，显示原始的 {{ }} 标签。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-pre 示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <span v-pre>{{ this will not be compiled }}</span>
    </div>
    <script>
        new Vue({
            el: '#app'
        });
    </script>
</body>

</html>
```


4. 指令 4.5 其他指令

- v-cloak：v-cloak 指令保持在元素上直到关联实例结束编译。和 CSS 规则如 [v-cloak] { display: none } 一起用时，这个指令可以隐藏未编译的 {{ }} 标签直到实例准备完毕。
  - 避免闪烁：可以避免在 Vue 实例还未编译完成时，用户看到未编译的 {{ }} 标签，提高用户体验。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-cloak 示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <style>
        [v-cloak] {
            display: none;
        }
    </style>
</head>

<body>
    <div id="app">
        <p v-cloak>{{ message }}</p>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                message: 'Hello, Vue!'
            }
        });
    </script>
</body>

</html>
```

4. 指令 4.5 其他指令

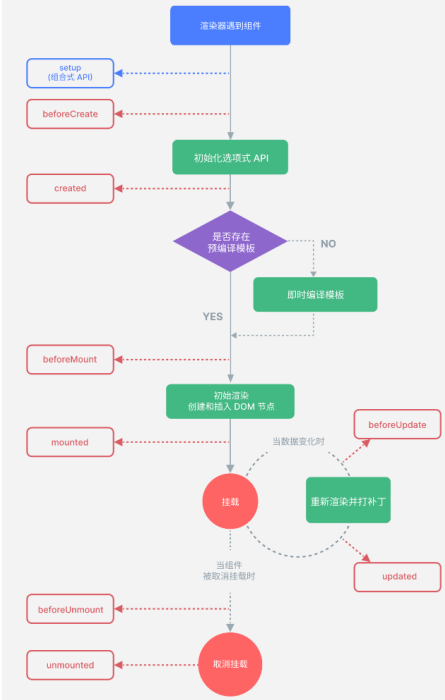
- v-once：v-once 指令只渲染元素和组件一次。随后的重新渲染，元素 / 组件及其所有的子节点将被视为静态内容并跳过。这可以用于优化更新性能。
  - 性能优化：对于一些不需要动态更新的内容，可以使用 v-once 来减少不必要的渲染开销。

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>v-once 示例</title>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<body>
    <div id="app">
        <span v-once>{{ message }}</span>
        <button @click="changeMessage">改变消息</button>
    </div>
    <script>
        new Vue({
            el: '#app',
            data: {
                message: '原始消息'
            },
            methods: {
                changeMessage() {
                    this.message = '新消息';
                }
            }
        });
    </script>
</body>

</html>
```


扩展：理解Vue.js 3的生命周期钩子

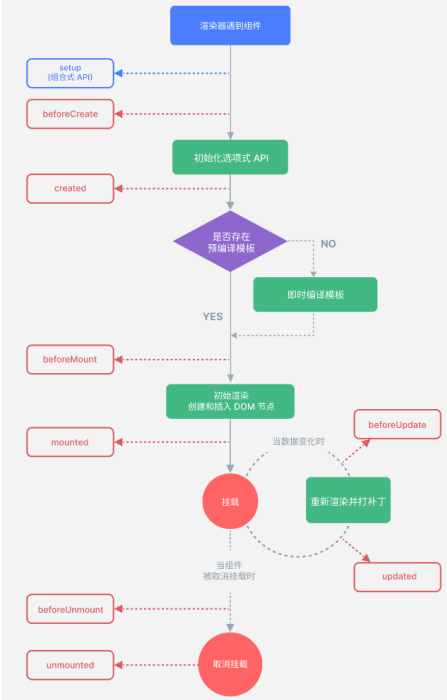
- Vue.js 3中的生命周期钩子是指在组件实例的创建、更新、销毁等过程中，自动调用的一系列方法，包括“beforeCreate”，“created”，“beforeMount”，“mounted”，“beforeUpdate”，“updated”，“beforeUnmount”，“unmounted”。通过这些钩子函数，可以在组件的不同生命周期阶段执行特定的操作。

- beforeCreate：
  - 这个阶段发生在实例初始化的最早期。
  - 此时组件实例刚刚被创建，还没有完成选项的处理和配置。
  - 不能访问“data”、“computed”、“methods”或“ref”。
  - 通常很少在这个钩子中编写实际的业务逻辑。



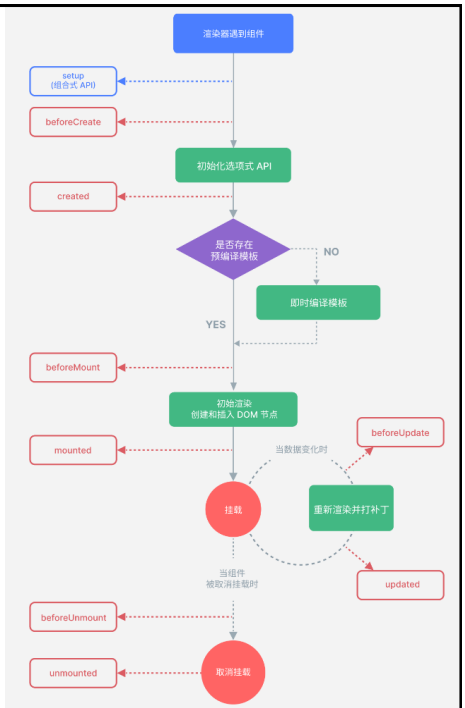
扩展：理解Vue.js 3的生命周期钩子

- created：
  - 组件实例已经创建完成，所有的选项（包括 data、computed、methods 等）都已配置好。
  - 可以进行一些不依赖于DOM的初始化操作，比如发送异步请求获取数据。
  - 仍然无法访问DOM元素。
- beforeMount：
  - 在组件即将被挂载到DOM之前调用。
  - 此时模板已经编译完成，但尚未渲染到真实的DOM中。
  - 可以在这个阶段做一些最后的准备工作。



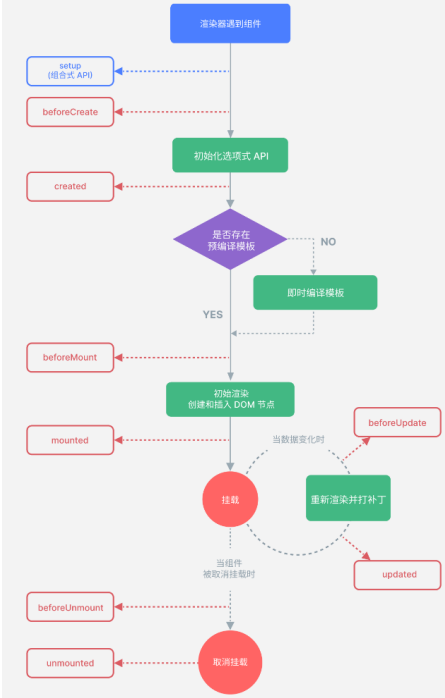
扩展：理解Vue.js 3的生命周期钩子

- mounted：
  - 组件已经被挂载到DOM上，所有的DOM操作都可以在此进行。
  - 通常用于执行与DOM相关的初始化，比如设置事件监听、初始化第三方库等。
  - 可以获取到真实的DOM元素，并对其进行操作。
- beforeUpdate：
  - 当数据发生变化，导致组件需要重新渲染时，在虚拟DOM重新渲染之前调用。
  - 可以在此阶段获取到变化前的状态，以便进行一些比较或处理。



扩展：理解Vue.js 3的生命周期钩子

- updated：
  - 组件重新渲染完成，虚拟DOM已经更新并应用到真实DOM之后调用。
  - 应避免在updated这个钩子中实现会导致组件再次更新的操作，以防出现陷入死循环的情况。
- beforeUnmount：
  - 在组件即将被销毁之前调用。
  - 可以在这里执行一些清理操作，例如移除事件监听、取消定时器等。
- unmounted：
  - 组件已经被销毁，相关的资源和副作用都已被清理。



信创智能医疗系统研发课程体系
河南中医药大学信息技术学院（智能医疗行业学院）

河南中医药大学信息技术学院（智能医疗行业学院）智能医疗教研室
河南中医药大学医疗健康信息工程技术研究所