



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08章：Vue.js组件

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>

2025/12/3

本章概要

- 组件基础
- 路由管理
- 状态管理
- 数据请求





1. 组件基础

1.1 组件简介

- 组件是对用户界面（UI）进行拆分与封装而形成的独立、可复用且自带逻辑的模块。每个组件皆为一个独立的 Vue.js 3 实例，具备独立的模板、数据、方法以及生命周期钩子，因而能够自行定义和管理其功能与样式。以下为组件的关键特性：
 - 模块化设计：组件开发旨在将 HTML、CSS 和 JavaScript 代码封装于组件内部，用于状态管理和逻辑处理，进而形成相互隔离的独立模块。
 - 可复用性：组件一经注册或引用，便可在应用的多个场景中反复使用。通过参数化配置，同一组件能够呈现出不同的形态与行为。
 - 嵌套结构：Vue.js 3 支持组件嵌套使用，可构建出层次清晰的组件树。这种结构有助于将用户界面拆分为多个易于管理的小型后代组件。
 - 生命周期钩子函数：生命周期钩子（Lifecycle Hooks）贯穿组件的整个生命周期，涵盖组件实例化前（beforeCreate）、组件创建并初始化完毕（created）、组件模板编译完成且挂载到 DOM 前（beforeMount）与挂载到 DOM 后（mounted）、响应式数据更新周期（beforeUpdate、updated），直至组件销毁前（beforeUnmount）和销毁后（unmounted）。
 - 事件通信机制：Vue.js 3 组件支持自定义事件的发送与监听，以此实现组件间的有效通信。

1. 组件基础

1.2 组件组成

- Vue.js 3 组件的主要结构包含三部分，分别为模板（Template）、逻辑（Script）和样式（Styles）。模板用于编辑组件的 HTML 结构；逻辑负责处理组件的 JavaScript 逻辑与数据等；样式则用于定义组件的 CSS 样式。

```
1. <template>
2. <!-- HTML结构 -->
3.   <div class="info">
4.     <h3>{{ infoData.name }}</h3>
5.     <div class="num">
6.       <p>
7.         <span>文章</span><span>{{ infoData.essayNum }}</span>
8.       </p>
9.       <p>
10.        <span>标签</span><span>{{ infoData.labelNum }}</span>
11.      </p>
12.      <p>
13.        <span>分类</span><span>{{ infoData.classifyNum }}</span>
14.      </p>
15.    </div>
16.  </div>
17.</template>
18.<script setup>
19. //JavaScript逻辑
20.console.log('个人信息展示组件');
21. //数据声明
22.const infoData = {
23.  name: "MyGlog",
24.  essayNum: 10,
25.  labelNum: 10,
26.  classifyNum: 10,
27. };
28.</script>
29.<style scoped>
30. /* 局部样式 */
31.</style>
```



1. 组件基础

1.2 组件组成

- Vue.js 3 的组件系统支持开发者运用独立且可复用的组件来构建复杂页面。在编写组件时，需遵循以下组件定义规范。
 - 单一职责原则：一个组件仅承担一项任务，以此确保组件的简洁性与可维护性。
 - 清晰的层次结构：对于大型组件，可将其拆分为更小的子组件，从而提升复用性和可读性。
 - 使用 props 进行数据传递：父组件能够通过 props 向子组件传递数据。
 - 使用事件回调进行通信：子组件可通过触发“emit”事件与父组件进行通信。



1. 组件基础

1.3 组件样式

- 组件样式可分为全局样式控制与局部样式控制。
- 全局样式控制指的是，在组件中定义的样式默认具备全局有效性，即该样式可作用于当前组件的标签、子组件的根标签以及外部标签。
- 而局部样式控制是通过在样式标签上添加scoped属性，使样式仅对当前组件的元素生效，从而避免样式对其他组件造成污染，确保组件样式的独立性与可控性。

1. 组件基础

1.3 组件样式

- 当style声明为scoped时，当前组件的所有标签和子组件的根标签会自动添加名为data-v-xxx的唯一标识属性。
- 在项目打包运行的页面中，style里的样式选择器最右侧会添加名为data-v-xxx的属性选择器。这使得局部作用域样式仅能作用于带有data-v-xxx属性的标签，而此时仅有当前组件的标签和子组件的根标签具备该属性，子组件的子标签和外部标签均无此属性，所以局部作用域样式仅能影响当前组件的标签和子组件的根标签。



- 若需进行深度样式控制，可使用“:deep()”来包含需要深度选择的标签，如此便能匹配并影响子组件的子标签。



1. 组件基础

1.4 组件注册

- 在 Vue.js 3 中，组件注册的目的在于确保程序在构建和渲染 DOM 时，能够准确识别并应用相应组件。
- 注册组件时，需遵循 W3C 规范中的 PascalCase（首字母大写）来命名自定义组件，例如“GlobalComponent”，以保证组件名具有描述性，能清晰反映组件的功能或内容。
- 组件注册主要分为全局注册和局部注册两种模式。



1. 组件基础

1.4 组件注册

- 全局注册

- 全局注册可使全局组件在任何模板中均可使用。在 Vue.js 3 里，通过 “component()” 方法实现组件的全局注册。下面以 “registerComponents” 示例来展示 Vue.js 3 的全局注册方法，具体步骤如下。
- 创建一个名为 “registerComponents” 的项目。



1. 组件基础

1.4 组件注册

- 全局注册

- 打开“registerComponents”项目，在“src\components\”目录下创建“GlobalComponent”文件夹，并在该文件夹下创建“GlobalComponentA.vue”文件，“GlobalComponentA”组件的内容如下。

```
1. <script setup>
2. //JS处理程序
3. console.log('你成功全局注册了GlobalComponentA组件');
4. </script>

5. <template>
6.   <!-- HTML结构 -->
7.   <div class="greetings">
8.     <h3>你成功全局注册了GlobalComponentA组件</h3>
9.   </div>
10. </template>

11. <style scoped>
12. /* 局部样式 */
13. h3 {
14.   font-weight: 500;
15.   font-size: 2.6rem;
16.   position: absolute;
17.   top: 45%; /* 垂直居中 */
18.   left: 50%; /* 水平居中 */
19.   transform: translate(-50%, -50%); /* 修正元素位置使其精确居中 */
20.   text-align: center; /* 文本水平居中 */
21. }
22. </style>
```



1. 组件基础

1.4 组件注册

- 全局注册

- 修改“src\main.js”文件，借助 Vue.js 3 应用实例中的“component()”方法实现组件的全局注册，文件内容如下。

```
1. import { createApp } from 'vue'
2. import App from './App.vue'
3. import GlobalComponentA from “./components/GlobalComponent/GlobalComponentA.vue “;
4. //创建应用实例, app是通过 “createApp” 函数创建一个新的应用实例
5. const app = createApp(App)
6. app.component(“GlobalComponentA “, GlobalComponentA);
7. app.mount('#app');
```

1. 组件基础

1.4 组件注册

- 全局注册

- 修改“src\App.vue”文件，在文件中使用已全局注册的“GlobalComponentA”组件，文件内容如下。

```
1. <template>
2.   <div id= "app ">
3.     <header>
4.       <h1>GlobalComponent</h1>
5.     </header>
6.     <main>
7.       <!-- 引用的全局组件 -->
8.       <GlobalComponentA />
9.     </main>
10.  </div>
11.</template>

12.<style scoped>
13.h1 {
14.  font-weight: 500;
15.  font-size: 3rem;
16.  position: absolute;
17.  top: 20%; /* 垂直居中 */
18.  left: 50%; /* 水平居中 */
19.  transform: translate(-50%, -50%); /* 修正元素位置使其精确居中 */
20.  text-align: center; /* 文本水平居中 */
21.}
22.</style>
```

- 需注意的是，采用全局注册组件的方式，若此类组件在应用的最终版本中未被实际引用或使用，生产环境的打包过程不会自动对其进行优化移除，这将导致最终打包文件的体积增大。此外，全局注册会使组件之间的依赖关系不够清晰，不利于项目的维护。

1. 组件基础

1.4 组件注册

● 局部注册

- 局部注册组件时，需要在它的父组件中进行引入或显式导入，注册后该组件仅在当前组件内可用。下面通过“registerComponents”项目展示局部注册。
- 打开“registerComponents”项目，在“src\components\”下创建“PartComponent”文件夹，并在“PartComponent”文件夹下创建“PartComponentA.vue”文件，“PartComponentA”组件的内容如下。

```
1. <template>
2.   <!-- HTML结构 -->
3.   <div>
4.     <h3>你成功局部注册了PartComponentA组件</h3>
5.   </div>
6. </template>
7. <style scoped>
8. h3 {
9.   font-weight: 500;
10.  font-size: 2.6rem;
11.  position: absolute;
12.  top: 45%; /* 垂直居中 */
13.  left: 50%; /* 水平居中 */
14.  transform: translate(-50%, -50%); /* 修正元素位置使其精确居中 */
15.  text-align: center; /* 文本水平居中 */
16. }
17. </style>
```



1. 组件基础

1.4 组件注册

● 局部注册

- 在“src\App.vue”文件中设置“<script setup>”，引入“PartComponentA”组件，文件内容如下。

```
1. <template>
2.   <div id= "app ">
3.     <header>
4.       <h1>PartComponent</h1>
5.     </header>
6.     <main>
7.       <!-- 引用的局部组件 -->
8.       <PartComponentA />
9.     </main>
10.  </div>
11.</template>
12.<script setup>
13.//引入组件PartComponentA，在使用“<script setup>”的单文件组件中，可直接引入组件无需注册。
14.import PartComponentA from “./components/PartComponent/PartComponentA.vue”;
15.</script>
16.<style scoped>
17.h1 {
18.  font-weight: 500;
19.  font-size: 3rem;
20.  position: absolute;
21.  top: 20%; /* 垂直居中 */
22.  left: 50%; /* 水平居中 */
23.  transform: translate(-50%, -50%); /* 修正元素位置使其精确居中 */
24.  text-align: center; /* 文本水平居中 */
25.}
26.</style>
```

- 需要留意的是，局部注册的组件在后代组件中不可用。在本示例中，“PartComponentA”注册后仅在当前组件可用，在任何子组件或更深层的子组件中均不可用。



1. 组件基础

1.4 组件注册

- 局部注册

- 在“src\App.vue”文件中使用“components”选项显式注册自定义组件。

```
1. //使用ES2015 的缩写语法
2. export default {
3.   name: "App ",
4.   components: {
5.     PartComponentA,
6.   },
7. };
8. //正常写法
9. export default {
10.  name: "App ",
11.  components: {
12.    PartComponentA: PartComponentA
13.  }
14. }
```



1. 组件基础

1.5 组件通信

- Vue 中的组件通信可实现组件间的数据传递与事件交互，达成组件状态同步。高效的组件通信机制能促使各组件协同运作，充分发挥 Vue.js 3 组件化开发的优势，提高项目的可维护性与开发效率。
- 在 Vue 里，组件通信通常分为三类：父子组件通信、兄弟组件通信和跨层级组件通信。

1. 组件基础

1.5 组件通信

● 父子组件通信

- 父组件向子组件传递数据借助“props”属性达成。父组件在模板中运用“v-bind”将数据绑定至子组件的“props”，子组件于“<script setup>”语法糖中通过“defineProps”接收数据。

```
1. <!--父组件-->
2. <template>
3.   <ChildComponent
4.     :title="parentTitle"
5.     :message="parentMessage"
6.     @childReceive="childReceive"
7.   />
8. <div>{{ childData }}</div>
9. </template>

10. <script setup>
11. const parentTitle = "子组件标题";
12. const parentMessage = "你好子组件";
13. </script>
```

```
1. <!--子组件-->
2. <template>
3.   <div>
4.     <h1>{{ title }}</h1>
5.     <p>{{ message }}</p>
6.   </div>
7. </template>

8. <script setup>
9. // 使用defineProps定义接收的props
10. const props = defineProps({
11.   title: String, // 假设title是一个字符串
12.   message: {
13.     // message可以是一个对象，更详细地定义prop
14.     type: String,
15.     required: true, // 标记为必需
16.     default: "Hello from default" // 默认值
17.   },
18. });
19. </script>
```

1. 组件基础

1.5 组件通信

● 父子组件通信

- 子组件向父组件传递数据则通过自定义事件实现，子组件在“<script setup>”中利用“defineEmits”定义自定义事件，通过“\$emit”触发该事件并传递数据，父组件在模板中使用“v-on”监听该事件并接收数据。

```
1. <!--父组件-->
2. <template>
3.   <div>
4.     <h2>父组件</h2>
5.     <ChildComponent @send-data="handleData" />
6.     <p>接收数据: {{ receivedData }}</p>
7.   </div>
8. </template>

9. <script setup>
10. import { ref } from "vue";
11. import ChildComponent from "../ChildComponent.vue";

12. // 接收子组件传递的数据
13. const receivedData = ref(null);

14. // 处理子组件发送的数据
15. const handleData = (data) => {
16.   receivedData.value = data;
17. };
18. </script>
```

```
1. <!--子组件-->
2. <template>
3.   <button @click="sendData">发送数据</button>
4. </template>

5. <script setup>
6. import { ref, defineEmits } from "vue";

7. // 定义可发射的事件
8. const emit = defineEmits(["send-data"]);

9. // 发送数据的方法
10. const sendData = () => {
11.   emit("send-data", "Hello from child!");
12. };
13. </script>
```



1. 组件基础

1.5 组件通信

● 兄弟组件通信

- 兄弟组件无法直接通信，需先将数据通过“emit”发送给父组件，再由父组件通过“props”将数据传递给另一个兄弟组件。

```
1. <!--父组件-->
2. <template>
3.   <div>
4.     <h2>父组件</h2>
5.     <ChildA @send-data="handleDataFromChildA" />
6.     <ChildB :dataFromChildA="receivedDataFromChildA" />
7.   </div>
8. </template>

9. <script setup>
10. import { ref } from "vue";
11. import ChildA from "../ChildComponent.vue";
12. import ChildB from "../ChildComponent2.vue";

13. // 接收来自ChildA的数据
14. const receivedDataFromChildA = ref(null);
15. // 处理ChildA发送的数据
16. const handleDataFromChildA = (data) => {
17.   receivedDataFromChildA.value = data;
18. };
19. </script>
```

```
1. <!--子组件A-->
2. <template>
3.   <button @click="sendData">发送数据</button>
4. </template>

5. <script setup>
6. import { ref, defineEmits } from "vue";

7. // 定义可发射的事件
8. const emit = defineEmits(["sen-data"]);

9. // 发送数据的方法
10. const sendData = () => {
11.   emit("send-data", "你好子组件A!");
12. };
13. </script>
```

```
1. <!--子组件B-->
2. <template>
3.   <div>
4.     <h2>子组件B</h2>
5.     <p>来自子组件A: {{ dataFromChildA }}</p>
6.   </div>
7. </template>

8. <script setup>
9. // 定义 props
10. defineProps({
11.   dataFromChildA: {
12.     type: String,
13.     default: "",
14.   },
15. });
16. </script>
```



1. 组件基础

1.5 组件通信

● 跨层级组件通信

- Vue 提供了“provide”和“inject”选项以实现跨层级的组件通信。祖先组件可使用“provide”选项提供数据，后代组件则能通过“inject”选项接收数据。

```
1. <!--父组件-->
2. <template>
3.   <div>
4.     <h2>父组件</h2>
5.     <Parent />
6.   </div>
7. </template>

8. <script setup>
9. import { ref, provide } from 'vue';
10. import Parent from './Parent.vue';

11. // 定义要传递的数据
12. const providedData = ref('来自父组件!');

13. // 使用 provide 提供数据
14. provide('sharedData', providedData);
15. </script>
```

```
1. <!--子组件-->
2. <template>
3.   <div>
4.     <h2>子组件</h2>
5.     <Child />
6.   </div>
7. </template>

8. <script setup>
9. import Child from './Child.vue';
10. import { inject } from 'vue';

11. // 使用 inject 接收数据
12. const sharedData = inject('sharedData');
13. </script>
```

```
1. <!--孙组件-->
2. <template>
3.   <div>
4.     <h2>孙组件</h2>
5.     <p>来自父组件的数据: {{ sharedData }}</p>
6.   </div>
7. </template>

8. <script setup>
9. import { inject } from 'vue';
10. // 使用 inject 接收数据
11. const sharedData = inject('sharedData');
12. </script>
```



1. 组件基础

1.6 第三方组件库

- 在 Vue.js 3 项目开发中，尽管 Vue 核心框架以及 Vue Router、Pinia 等官方生态已提供完善的基础功能，实际项目中仍常面临以下需求：
 - UI 组件需求：快速搭建美观且功能完备的用户界面
 - 开发效率需求：避免重复开发基础组件，节约开发时间
 - 功能扩展需求：实现诸如文件上传、图表展示、国际化等特定功能
 - 性能优化需求：借助专业工具提升应用性能表现
- 第三方组件库与插件正是为解决上述问题而设计，它们经过社区广泛验证，具备高可靠性、完善的文档支持与活跃的维护，能有效助力开发者快速构建高质量的 Vue.js 3 应用程序。



1. 组件基础

1.6 第三方组件库

- Vue 集成第三方组件库主要包括以下三种方式。

- 通过 CDN 引入外部插件
- 使用 CDN 引入外部插件时，只需在 HTML 文件中添加相应的 CDN 链接。

```
1. <!-- 引入 jQuery -->
2. <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
3. <!-- 引入其他第三方插件 -->
4. <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

- 之后，即可在 Vue 组件中直接使用该插件。

```
1. mounted() {
2.   // 使用 jQuery
3.   $('#my-element').hide();

4.   // 使用 Chart.js
5.   new Chart(document.getElementById('myChart'), { /* chart configuration */ });
6. }
```

- 该方法配置简单，但不太适用于大型项目或对可维护性要求较高的场景。



1. 组件基础

1.6 第三方组件库

- Vue 集成第三方组件库主要包括以下三种方式。
 - 使用 npm 安装并引入插件
 - 推荐通过 npm 安装第三方库，并在项目中引入，以便更好地管理依赖，提升代码可维护性。

```
1. npm install chart.js --save
```

- 安装完成后，在 Vue 组件中直接使用即可。

```
1. import Chart from 'chart.js';  
2. export default {  
3.   mounted() {  
4.     new Chart(document.getElementById('myChart'), { /* chart configuration */ });  
5.   }  
6. }
```

1. 组件基础

1.6 第三方组件库

- Vue 集成第三方组件库主要包括以下三种方式。
 - 通过 Vue 插件系统集成第三方库
 - Vue 提供了标准的插件机制，便于将第三方库集成至项目中。通过 npm 安装组件库后，可使用 `Vue.use()` 方法将外部插件注册到 Vue 中。Vuetify、Axios 等许多流行库均支持该集成方式。

```
1. import Vue from 'vue';  
2. import SomeLibrary from 'some-library';  
  
3. Vue.use(SomeLibrary);
```



2. 路由管理

2.1 路由简介

- 路由的基本思想是关于数据定向和转发的一种机制，前端路由作为单页应用（SPA）中的页面导航管理技术，本质上是路径与对应组件之间构建的一组映射关系。
 - 传统地址跳转方式下，用户点击链接或在地址栏输入URL后，浏览器会向服务器发送请求，服务器返回新的页面，导致每次页面切换都需重新加载整个页面。
 - 在SPA中，所有页面内容均在一个HTML文件中加载，页面切换由前端路由实现：点击链接或输入URL时，前端路由会拦截请求，依据URL地址按配置规则加载对应的页面内容。
- Vue.js 3应用启动时会创建一个全局路由实例（通过 `new VueRouter()`），该实例负责监听URL变化。一旦URL发生改变，路由实例将根据当前URL匹配预设的路由规则，进而完成页面渲染。



2. 路由管理

2.1 路由简介

- 目前前端路由主要包含两种实现方式：Hash模式和History模式。
 - Hash模式
 - Hash模式借助URL中hash部分（即“#”号后的内容）设置路由路径。hash值变化时，浏览器不会向服务器发送请求，而是触发“hashchange”事件，通过监听该事件可捕获URL变更并更新页面内容。
 - History模式
 - History模式则基于HTML5 History API实现，该API提供的“pushState”和“replaceState”方法能够在不刷新页面的情况下，向浏览器历史记录添加或替换状态。同时，浏览器历史记录变化（如前进或后退操作）会触发“popstate”事件。



2. 路由管理

2.2 Vue Router

- Vue Router 是 Vue.js 3 的官方路由管理器，具备包括嵌套路由映射、动态路由选择、模块化设计、基于组件的路由配置、路由参数传递、查询参数处理、通配符匹配、过渡效果、导航控制、自动激活 CSS 类的链接、HTML5 history 模式与 hash 模式切换、可定制的滚动行为，以及 URL 的正确编码等一系列功能。
- Vue Router 与 Vue.js 3 核心深度集成，能够更便捷地构建单页面应用。其主要特点如下：
 - 声明式导航：通过 `<RouterLink>` 组件创建 a 标签，以声明方式定义导航链接。
 - 程式化导航：提供如 `router.push()`、`router.replace()`、`router.go()` 等 API，支持在 JavaScript 中灵活控制路由跳转。
 - 嵌套路由：支持将路由映射至嵌套的组件结构，适应复杂页面布局。
 - 路由守卫：允许在路由切换的不同阶段（如切换前、切换后、切换失败）执行特定逻辑。
 - 路由参数：支持在 URL 中传递参数，并可在对应组件中获取和使用。
 - 路由懒加载：支持按需加载路由组件，优化应用性能。



2. 路由管理

2.2 Vue Router

- 创建 Vue Router 项目，深入理解并掌握其使用方法，具体步骤如下：
 - 创建名为“MyVueRouter”的项目。
 - 打开项目，在目录空白处右键单击，选择“在集成终端中打开”。
 - 在集成终端中运行命令“`npm install vue-router@latest`”，安装 Vue Router。

2. 路由管理

2.2 Vue Router

- 创建 Vue Router 项目，深入理解并掌握其使用方法，具体步骤如下：
 - 在“src”目录下创建“router”文件夹，并在其中创建“index.js”文件作为路由配置。

```
1. // 导入 Vue Router 的 createRouter 和 createWebHashHistory 函数
2. import { createRouter, createWebHashHistory } from 'vue-router';

3. // 导入首页组件
4. import HomeView from '../views/HomeView.vue';

5. // 创建一个 Vue Router 实例
6. const router = createRouter({
7.   // 设置 history 模式为 hash 模式
8.   history: createWebHashHistory(import.meta.env.BASE_URL),

9.   // 定义路由规则
10.  routes: [
11.    // 首页路由配置
12.    {
13.      // 路由路径
14.      path: '/',
15.      // 路由名称
16.      name: 'home',
17.      // 对应的组件
18.      component: HomeView
19.    },

20.    // 关于页面路由配置
21.    {
22.      // 路由路径
23.      path: '/about',
24.      // 路由名称
25.      name: 'about',
26.      // 动态导入组件，实现懒加载
27.      component: () => import('../views/AboutView.vue')
28.    }
29.  ]
30. });

31. // 导出默认的 router 实例供其他模块使用
32. export default router;
```

2. 路由管理

2.2 Vue Router

- 创建 Vue Router 项目，深入理解并掌握其使用方法，具体步骤如下：
- 修改 “src\main.js” 文件，引入并全局注册路由组件，内容如下。

```
1. // 导入 Vue Router 的 createRouter 和 createWebHashHistory 函数
2. import { createRouter, createWebHashHistory } from 'vue-router';

3. // 导入首页组件
4. import HomeView from '../views/HomeView.vue';

5. // 创建一个 Vue Router 实例
6. const router = createRouter({
7.   // 设置 history 模式为 hash 模式
8.   history: createWebHashHistory(import.meta.env.BASE_URL),

9.   // 定义路由规则
10.  routes: [
11.    // 首页路由配置
12.    {
13.      // 路由路径
14.      path: '/',
15.      // 路由名称
16.      name: 'home',
17.      // 对应的组件
18.      component: HomeView
19.    },

20.    // 关于页面路由配置
21.    {
22.      // 路由路径
23.      path: '/about',
24.      // 路由名称
25.      name: 'about',
26.      // 动态导入组件，实现懒加载
27.      component: () => import('../views/AboutView.vue')
28.    }
29.  ]
30. });

31. // 导出默认的 router 实例供其他模块使用
32. export default router;
```



2. 路由管理

2.2 Vue Router

- 创建 Vue Router 项目，深入理解并掌握其使用方法，具体步骤如下：
 - 修改 “src\main.js” 文件，引入并全局注册路由组件，内容如下。

```
1. import './assets/main.css'

2. import { createApp } from 'vue'
3. import App from './App.vue'

4. // 导入路由实例，用于管理应用的路由
5. import router from './router';

6. const app = createApp(App)

7. // 使用 app.use() 方法注册路由到 Vue 应用中，在整个应用中使用路由功能
8. app.use(router);

9. app.mount('#app')
```

2. 路由管理

2.2 Vue Router

- 创建 Vue Router 项目，深入理解并掌握其使用方法，具体步骤如下：
 - 在“src”目录下创建“views”文件夹，并分别创建“HomeView.vue”与“AboutView.vue”文件作为视图组件，内容如下。

```
1. <template>
2.   <div class="home">
3.     <h1>这是主页</h1>
4.   </div>
5. </template>

6. <style scoped>
7. .home {
8.   min-height: 100vh;
9.   display: flex;
10.  align-items: center;
11. }
12. </style>
```

```
1. <template>
2.   <div class="about">
3.     <h1>这是关于页</h1>
4.   </div>
5. </template>

6. <style scoped>
7. .about {
8.   min-height: 100vh;
9.   display: flex;
10.  align-items: center;
11. }
12. </style>
```



2. 路由管理

2.2 Vue Router

- 创建 Vue Router 项目，深入理解并掌握其使用方法，具体步骤如下：
 - 在集成终端中执行“npm install”命令，安装项目所需依赖。
 - 运行“npm run dev”启动开发服务器。

2. 路由管理

2.2 Vue Router

- 创建 Vue Router 项目，深入理解并掌握其使用方法，具体步骤如下：
 - 修改 “src\App.vue” 文件，引入并使用 RouterLink 和 RouterView 组件，内容如下。

```
1. <script setup>
2. // 从 'vue-router' 库导入 RouterLink 和 RouterView 组件
3. import { RouterLink, RouterView } from "vue-router";
4. </script>

5. <template>
6.   <!-- 头部区域 -->
7.   <header>
8.     <div class="wrapper">
9.       <!-- 导航栏 -->
10.      <nav>
11.        <!-- RouterLink 组件用于创建链接到不同路由的链接 -->
12.        <!-- 当前页面被激活时，会自动添加 active 类名 -->
13.        <RouterLink to="/">Home</RouterLink>
14.        <RouterLink to="/about">About</RouterLink>
15.      </nav>
16.    </div>
17.  </header>

18.  <!-- RouterView 组件用于渲染当前活动的路由视图 -->
19.  <RouterView />
20.</template>

21.<style scoped>
22.nav {
23.  width: 100vh;
```

```
24.  height: 50%;
25.  font-size: 24px;
26.  text-align: center;
27.  margin-top: 20rem;
28. }
29.nav a.router-link-exact-active {
30.  color: var(--color-text);
31. }

32.nav a.router-link-exact-active:hover {
33.  background-color: transparent;
34. }

35.nav a {
36.  display: inline-block;
37.  padding: 0 1rem;
38.  border-left: 1px solid var(--color-border);
39. }

40.nav a:first-of-type {
41.  border: 0;
42. }
43.</style>
```

2. 路由管理

2.3 路由应用

- 嵌套路由

- 嵌套路由允许开发者将路由映射到嵌套的组件结构中，通过在父路由下定义子路由，能够构建出层次分明的页面架构。
- 举例来说，可以在一个主页面内嵌套呈现多个子页面，这些子页面的内容能够随着主页面路由的变化而动态更新。
- 实现嵌套路由的核心是在父级路由组件中嵌入<router-view>组件，用以承载子路由的渲染内容，具体实现包含以下步骤。
- 创建App.vue文件作为应用的根组件，并在其中配置主路由规则和<router-view>容器。

```
1. <template>
2.   <router-view />
3. </template>
```



2. 路由管理

2.3 路由应用

- 嵌套路由

- 创建User.vue作为子路由组件，定义子路由的具体内容，同时在该组件内部再次使用<router-view>，以实现更深层次的嵌套路由功能。

```
1. <template>
2.   <div>
3.     User
4.     <router-view />
5.   </div>
6. </template>
```

- 创建UserInfo.vue作为更深层级的子路由组件，进一步完善子路由的详细内容展示。

```
1. <template>
2.   <div>
3.     UserName: Tom
4.   </div>
5.   <div>
6.     UserMail: tom@163.com
7.   </div>
8. </template>
```

2. 路由管理

2.3 路由应用

- 嵌套路由

- 修改路由配置文件，明确设定父级路由与子级路由之间的映射规则，从而确保主页面能够正确渲染子页面内容，同时支持子页面根据路由变化实时更新。

```
1. const routes = [  
2.   {  
3.     path: '/user/:id',  
4.     component: User,  
5.     children: [  
6.       {  
7.         // 当 /user/:id/profile 匹配成功  
8.         // UserProfile 将被渲染到 User 的 <router-view> 内部  
9.         path: 'profile',  
10.        component: UserProfile,  
11.      },  
12.      {  
13.        // 当 /user/:id/posts 匹配成功  
14.        // UserPosts 将被渲染到 User 的 <router-view> 内部  
15.        path: 'posts',  
16.        component: UserPosts,  
17.      },  
18.    ],  
19.  },  
20.]
```

2. 路由管理

2.3 路由应用

- 命名路由

- 在Vue.js框架中，命名路由是一种为路由规则分配唯一标识名的机制，此举显著提升了路由使用的灵活性与便捷性。特别是在进行导航控制、参数传递和代码维护时，命名路由展现出巨大优势。在路由配置过程中，可通过name属性为每个路由赋予一个易于识别的名称。

```
1. const routes = [  
2.   {  
3.     path: '/user/:username',  
4.     name: 'profile',  
5.     component: User  
6.   }  
7. ]
```

- 在实际使用过程中，可以借助路由名称实现精确的路由跳转操作。

```
1. <router-link :to="{ name: 'profile', params: { username: 'erina' } }">  
2.   User profile  
3. </router-link>
```

2. 路由管理

2.3 路由应用

● 命名视图

- 在某些复杂的布局需求中，可能需要同时展示多个同级视图而非嵌套视图。例如，创建一个包含侧边栏（sidebar）和主内容区（main）的页面布局时，命名视图技术就能大显身手。通过为多个<router-view>指定不同的名称，可以在界面中同时渲染多个独立命名的视图区域。需要注意的是，未命名的<router-view>将自动获得默认名称default。

```
1. <router-view class="view left-sidebar" name="LeftSidebar" />
2. <router-view class="view main-content" />
3. <router-view class="view right-sidebar" name="RightSidebar" />
```

- 每个命名视图都需要一个对应的组件进行渲染，因此同一路由下的多个视图需要配置多个组件。在配置时务必使用components（带s）属性，这样才能准确为各个视图指定对应的组件，实现灵活多样的页面布局方案，满足复杂项目中多样化的展示需求。

```
1. const router = createRouter({
2.   history: createWebHashHistory(),
3.   routes: [
4.     {
5.       path: '/',
6.       components: {
7.         default: Home,
8.         // LeftSidebar: LeftSidebar 的缩写
9.         LeftSidebar,
10.        // 它们与 `<router-view>` 上的 `name` 属性匹配
11.        RightSidebar,
12.      },
13.    },
14.  ],
15.})
```

2. 路由管理

2.3 路由应用

- 路由组件传参

- 路由组件传参是一种在路由路径中传递动态参数的重要技术，常用于传递资源ID、分类标识等信息，从而实现基于不同参数值的页面内容展示（如/user/1、/product/2等）。Vue路由支持params和query两种传参方式：params参数作为路径的组成部分，在路由配置中使用冒号语法定义；query参数则通过URL问号后的键值对传递，可通过`\$route.query`对象获取，具体实现步骤如下。

- 第1步：实现路由配置

```
1. import { createRouter, createWebHistory } from 'vue-router';

2. // 定义路由
3. const routes = [
4.   // 动态路由（用于接收 params）
5.   {
6.     path: '/user/:userId', // 动态参数 userId
7.     name: 'UserProfile', // 命名路由（params 必须用 name 跳转）
8.     component: () => import('@/views/UserProfile.vue')
9.   },
10.  // 普通路由（用于接收 query）
11.  {
12.    path: '/search',
13.    name: 'SearchPage',
14.    component: () => import('@/views/SearchPage.vue')
15.  }
16.];

17. const router = createRouter({
18.  history: createWebHistory(),
19.  routes
20.});

21. export default router;
```

2. 路由管理

2.3 路由应用

- 路由组件传参

- 第2步：在路由跳转时传递params参数

```
1. <!-- 组件 A: 触发跳转 -->
2. <template>
3.   <button @click="goToUserProfile">查看用户详情</button>
4. </template>

5. <script setup>
6. import { useRouter } from 'vue-router';

7. const router = useRouter();

8. const goToUserProfile = () => {
9.   // 方式 1: 通过命名路由 + params (推荐, 避免 path 覆盖 params)
10.  router.push({
11.    name: 'UserProfile', // 必须用 name (path 会忽略 params)
12.    params: { userId: 123 } // 传递参数 (键名需与路由动态段一致)
13.  });

14.  // 方式 2: 错误示范 (用 path 会丢失 params!)
15.  // router.push({ path: '/user', params: { userId: 123 } }); // URL 会是 /user, 参数丢失
16. };
17. </script>
```



2. 路由管理

2.3 路由应用

- 路由组件传参

- 第3步：在组件中接收params参数

```
1. <!-- UserProfile.vue -->
2. <template>
3.   <div>用户 ID: {{ userId }}</div>
4. </template>

5. <script setup>
6. import { useRoute } from 'vue-router';

7. const route = useRoute();
8. const userId = route.params.userId; // 获取 params 参数（类型为 string，需手动转换数字）
9. </script>
```

2. 路由管理

2.3 路由应用

- 路由组件传参

- 第4步：跳转时传递query参数

```
1. <!-- 组件 B：触发搜索跳转 -->
2. <template>
3.   <input v-model="keyword" placeholder="输入搜索词" />
4.   <button @click="doSearch">搜索</button>
5. </template>

6. <script setup>
7. import { ref, useRouter } from 'vue-router';

8. const router = useRouter();
9. const keyword = ref('');

10. const doSearch = () => {
11.   // 方式 1：通过 path + query（直接拼接 URL）
12.   router.push({
13.     path: '/search', // 可用 path 或 name
14.     query: { keyword: keyword.value } // 参数会显示为 ?keyword=xxx
15.   });

16.   // 方式 2：通过 name + query（效果同上）
17.   // router.push({
18.   //   name: 'SearchPage',
19.   //   query: { keyword: keyword.value }
20.   // });
21. };
22. </script>
```



2. 路由管理

2.3 路由应用

- 路由组件传参

- 第5步：在组件中接收query参数

```
1. <!-- SearchPage.vue -->
2. <template>
3.   <div>搜索关键词: {{ keyword }}</div>
4. </template>

5. <script setup>
6. import { useRoute } from 'vue-router';

7. const route = useRoute();
8. const keyword = route.query.keyword; // 获取 query 参数 (类型为 string)
9. </script>
```

2. 路由管理

2.3 路由应用

● 编程式路由

- 除了使用 <router-link> 创建 a 标签来定义导航链接，还可以借助 router 的实例方法，通过编写代码来实现。Vue Router 提供了诸如 router.push、router.replace、router.go 等编程式导航的 API，使开发者能够在 JavaScript 代码中灵活控制路由跳转，实现更复杂的路由逻辑。
- 如需导航至不同 URL，可使用 router.push 方法。该方法会向 history 栈添加新记录，因此用户点击浏览器后退按钮时将返回之前的 URL。该方法的参数可以是一个字符串路径，或一个描述地址的对象。

```
1. // 字符串路径
2. router.push('/users/eduardo')

3. // 带有路径的对象
4. router.push({ path: '/users/eduardo' })

5. // 命名的路由，并加上参数，让路由建立 url
6. router.push({ name: 'user', params: { username: 'eduardo' } })

7. // 带查询参数，结果是 /register?plan=private
8. router.push({ path: '/register', query: { plan: 'private' } })

9. // 带 hash，结果是 /about#team
10. router.push({ path: '/about', hash: '#team' })
```

2. 路由管理

2.3 路由应用

- 编程式路由

- `router.replace` 的功能与 `router.push` 类似，不同之处在于它导航时不会向 `history` 添加新记录，正如其名称所示——它会替换当前的历史记录。

```
1. router.push({ path: '/home', replace: true })
2. // 相当于
3. router.replace({ path: '/home' })
```

- `router.go` 方法接受一个整数作为参数，用于在历史堆栈中前进或后退指定步数，其行为类似于 `window.history.go(n)`。

```
1. // 向前移动一条记录，与 router.forward() 相同
2. router.go(1)

3. // 返回一条记录，与 router.back() 相同
4. router.go(-1)

5. // 前进 3 条记录
6. router.go(3)

7. // 如果没有那么多记录，静默失败
8. router.go(-100)
9. router.go(100)
```



2. 路由管理

2.3 路由应用

- 路由守卫

- 路由守卫是 Vue Router 中用于控制导航的钩子函数，可在导航发生前、发生后或结束时触发。借助路由守卫，可执行权限验证、数据预取等操作，确保只有符合特定条件的用户才能访问某些路由，从而增强应用的安全性和用户体验。
- 路由守卫主要分为全局守卫、路由独享守卫和组件内守卫。

2. 路由管理

2.3 路由应用

- 路由守卫

- 全局守卫

- 第1步：实现路由配置

```
1. import { createRouter, createWebHistory } from 'vue-router';

2. // 路由配置数组（按「功能模块/访问层级」排序，便于维护）
3. const routes = [
4.   {
5.     name: 'Home',          // 路由名称（建议唯一且语义化）
6.     path: '/',             // 匹配路径
7.     component: () => import('@views/Home.vue'), // 懒加载组件
8.     meta: {
9.       title: '首页',      // 页面标题（用于全局后置守卫动态修改document.title）
10.      requiresAuth: true // 权限标记：需要登录才能访问
11.    }
12.  },
13.  {
14.    name: 'Login',
15.    path: '/login',
16.    component: () => import('@views/Login.vue'),
17.    meta: {
18.      title: '登录',
19.      requiresAuth: false
20.    }
21.  },
22.  {
23.    name: 'Dashboard',
24.    path: '/dashboard',
25.    component: () => import('@views/Dashboard.vue'),
26.    meta: {
27.      title: '控制台',
28.      requiresAuth: true // 需要登录验证
29.    }
30.  }
31. ];
```

```
32. // 创建路由实例（整合历史模式与路由配置）
33. const router = createRouter({
34.   history: createWebHistory(import.meta.env.BASE_URL),
35.   routes,
36.   scrollBehavior(to, from, savedPosition) {
37.     return savedPosition || { top: 0 };
38.   }
39. });

40. export default router;
```

2. 路由管理

2.3 路由应用

- 路由守卫

- 全局守卫

- 第2步：使用全局前置守卫 `beforeEach` 检查用户是否登录，若未登录则重定向至登录页面，以确保仅登录用户可访问需权限的页面。

```
1. // 在 router/index.js 中添加
2. import { useUserStore } from '@/stores/user'; // 假设用 Pinia 管理用户状态

3. router.beforeEach((to, from, next) => {
4.   const userStore = useUserStore();
5.   const isLoggedIn = userStore.isLoggedIn; // 获取用户登录状态

6.   // 检查目标路由是否需要登录（通过 meta 字段标记）
7.   if (to.meta.requiresAuth && !isLoggedIn) {
8.     // 未登录：跳转到登录页，并记录当前想访问的路径（登录后可跳转回来）
9.     next({
10.      path: '/login',
11.      query: { redirect: to.fullPath } // 携带目标路径作为查询参数
12.    });
13.   } else {
14.     // 已登录或无需验证：放行
15.     next();
16.   }
17. });
```



2. 路由管理

2.3 路由应用

- 路由守卫

- 全局守卫

- 第2步：使用全局后置守卫 `afterEach` 修改页面标题，使用户能清晰了解当前页面内容。

```
1. // 在 router/index.js 中添加
2. router.afterEach((to) => {
3.   // 从路由 meta 中获取标题（未设置则默认「未知页面」）
4.   const title = to.meta.title || '未知页面';
5.   document.title = `${title} - 我的项目`; // 修改页面标题
6. });
```

- 其中关键参数说明如下：
 - `to`：目标路由对象（包含 `path`、`query`、`meta` 等信息）。
 - `from`：当前正要离开的路由对象。
 - `next`：控制导航的函数（必须调用）：
 - `next()`：允许进行导航。
 - `next('/path')`：重定向至指定路径。
 - `next(false)`：取消当前导航（URL 将回退）。

2. 路由管理

2.3 路由应用

● 路由独享守卫

- 路由独享守卫仅在特定路由切换时触发，其定义和使用方式与全局守卫类似，但仅作用于该特定路由，可对该路由的访问执行特殊逻辑处理，例如检查特定权限。例如，在需特定权限访问的路由上设置路由独享守卫，当用户尝试访问时检查权限，若权限不足则提示用户或阻止访问，以保障系统安全。

```
1. import { createRouter, createWebHistory } from 'vue-router';
2. import { useUserStore } from '@/stores/user'; // 假设用 Pinia 管理用户状态

3. // 路由配置 (含独享守卫)
4. const routes = [
5.   {
6.     name: 'Home',
7.     path: '/',
8.     component: () => import('@/views/Home.vue'),
9.     meta: { title: '首页', requiresAuth: true },
10.    // 路由独享守卫: 进入首页前校验权限
11.    beforeEnter: (to, from, next) => {
12.      const userStore = useUserStore();
13.      if (userStore.isLoggedIn) {
14.        next(); // 已登录, 放行
15.      } else {
16.        next({ name: 'Login', query: { redirect: to.fullPath } }); // 未登录跳登录页
17.      }
18.    },
19.  },
20.  {
21.    name: 'Dashboard',
22.    path: '/dashboard',
23.    component: () => import('@/views/Dashboard.vue'),
24.    meta: { title: '控制台', requiresAuth: true, permission: 'admin' }, // 新增权限标识
25.    // 路由独享守卫: 进入控制台前校验管理员权限
26.    beforeEnter: (to, from, next) => {
27.      const userStore = useUserStore();
28.      // 校验登录状态 + 管理员权限
29.    },
30.  },
31.];
```

```
1. if (userStore.isLoggedIn && userStore.role === 'admin') {
2.   next(); // 符合条件, 放行
3. } else if (!userStore.isLoggedIn) {
4.   next({ name: 'Login', query: { redirect: to.fullPath } }); // 未登录跳登录页
5. } else {
6.   next({ name: 'Forbidden', query: { from: to.path } }); // 无权限跳 403 页
7. }
8. },
9. ],
10. {
11.   name: 'Login',
12.   path: '/login',
13.   component: () => import('@/views/Login.vue'),
14.   meta: { title: '登录' },
15. },
16. {
17.   name: 'Forbidden',
18.   path: '/403',
19.   component: () => import('@/views/Forbidden.vue'),
20.   meta: { title: '无权限访问' },
21. },
22. ];

23. const router = createRouter({
24.   history: createWebHistory(import.meta.env.BASE_URL),
25.   routes
26. });

27. export default router;
```

2. 路由管理

2.3 路由应用

● 组件内守卫

- 组件内守卫在组件自身的路由切换时触发，允许在组件内部定义特定的导航逻辑，可在组件创建、挂载、销毁等不同阶段执行路由控制，以满足组件自身的业务需求，例如在组件挂载前预加载数据。此外，可在组件销毁前释放资源，避免资源浪费，确保组件正常运行与应用的性能优化。

```
1. <script setup>
2. import { ref } from 'vue';
3. import { useRoute, useRouter } from 'vue-router';
4. import { fetchArticleDetail, saveDraft } from '@/api/article';

5. const route = useRoute(); // 当前路由对象（含动态参数）
6. const router = useRouter(); // 路由实例
7. const article = ref(null); // 文章数据
8. const draftContent = ref(''); // 草稿内容（模拟未保存的修改）

9. // -----
10. // 1. beforeRouteEnter: 进入组件前触发（组件未创建时）
11. // -----
12. // 注意：此时无法访问组件实例（this 不存在），但可以通过 next 回调访问
13. router.beforeRouteEnter((to, from, next) => {
14.   // 预加载文章数据（根据路由参数 to.params.id）
15.   fetchArticleDetail(to.params.id)
16.     .then(res => {
17.       // 将数据暂存到路由的 meta 中（或通过 Pinia 全局状态管理）
18.       // 因为组件未创建，无法直接赋值给 article.value
19.       next(vm => {
20.         // vm 是组件实例的代理（类似 this），可以访问组件数据
21.         vm.article = res.data;
22.         vm.draftContent = res.data.content; // 初始化草稿内容
23.       });
24.     })
25.     .catch(err => {
26.       next({ name: 'Error', query: { msg: '文章加载失败' } }); // 加载失败跳转错误页
27.     });
28. });
```

```
29. // -----
30. // 2. beforeRouteUpdate: 当前路由改变但组件被复用时触发（如动态参数变化）
31. // -----
32. // 场景：从 /article/1 跳转到 /article/2，组件实例被复用（不销毁重建）
33. router.beforeRouteUpdate(async (to, from, next) => {
34.   // 检测路由参数是否变化（如文章 ID 改变）
35.   if (to.params.id !== from.params.id) {
36.     try {
37.       const res = await fetchArticleDetail(to.params.id);
38.       article.value = res.data; // 更新文章数据
39.       draftContent.value = res.data.content; // 重置草稿为新文章内容
40.     } catch (err) {
41.       next({ name: 'Error', query: { msg: '文章加载失败' } });
42.     }
43.   } else {
44.     next(); // 参数未变化，直接放行
45.   }
46. });
```

```
48. // -----
49. // 3. beforeRouteLeave: 离开当前组件对应的路由前触发（如跳转到其他页面）
50. // -----
51. // 场景：离开文章详情页时，检查是否有未保存的草稿
52. router.beforeRouteLeave((to, from, next) => {
53.   if (draftContent.value !== article.value?.content) {
54.     // 有未保存的修改：弹出确认框
55.     const confirmLeave = window.confirm('当前内容未保存，确定离开吗?');
56.     confirmLeave ? next() : next(false); // 确认离开 / 取消导航
57.   } else {
58.     next(); // 无未保存修改，直接放行
59.   }
60. });

61. // 组件卸载时清理（可选）
62. onUnmounted(() => {
63.   draftContent.value = ''; // 清空草稿缓存
64. });
65. </script>

66. <template>
67.   <div v-if="article">
68.     <h1>{{ article.title }}</h1>
69.     <p>{{ article.content }}</p>
70.     <textarea v-model="draftContent" placeholder="编辑草稿..." />
71.   </div>
72.   <div v-else>加载中...</div>
73. </template>
```



3. 状态管理

3.1 概述

- 随着Vue应用日益复杂，组件间的数据共享可能演变为维护难题。试想：用户登录信息需在多个组件中使用，若通过props层层传递，代码将变得臃肿且难以维护。此时，我们迫切需要一个“中央数据仓库”——这正是Pinia的价值所在。作为Vue官方推荐的状态管理库，Pinia是Vuex的继任者。若曾使用过Vuex，将会发现Pinia具备以下优势：
 - 更简洁的API：摒弃mutations，仅保留state、getters和actions
 - 完善的TypeScript支持：完美兼容TS，提供自动类型推导
 - 模块化设计：无需创建命名空间
 - 轻量级：体积约6KB，比Vuex小近一半
 - 支持Vue DevTools：可追踪状态变化、进行时间旅行调试等



3. 状态管理

3.2 环境配置

- 在项目中使用Pinia前，需先完成安装。可通过npm或yarn进行安装：
- npm安装命令如下。

```
1. npm install pinia
```

- yarn安装命令如下。

```
1. yarn add pinia
```

- 安装完成后，在main.js文件中初始化Pinia并将其挂载至Vue应用。

```
1. import { createApp } from 'vue'
2. import App from './App.vue'
3. import { createPinia } from 'pinia'
4.
5. // 必须要在挂载前注入！！
6. const app = createApp(App)
7. const pinia = createPinia()
8.
9. app.use(pinia)
10. app.mount('#app')
```

- 上述代码中，通过createPinia()创建Pinia实例，再使用app.use(pinia)将其安装到Vue应用中。

3. 状态管理

3.4 核心概念

- Store工厂模式

- 在Pinia中，状态管理的核心是store——一个包含状态（state）、Getters和Actions的对象。
- 建议在项目中创建stores目录，用于存放所有store文件。例如，创建counter.js文件来管理简单的计数器状态。

```
1. import { defineStore } from 'pinia'
2.
3. export const useCounterStore = defineStore('counter', {
4.   state: () => ({
5.     count: 0
6.   }),
7.   getters: {
8.     doubleCount: (state) => state.count * 2
9.   },
10.  actions: {
11.    increment() {
12.      this.count++
13.    },
14.    decrement() {
15.      this.count--
16.    }
17.  }
18.})
```

- defineStore是Pinia提供的函数，用于定义store。首个参数'counter'为store的唯一标识符
- state函数返回一个对象，其属性即为需要管理的状态
- getters定义了基于状态计算的派生状态，类似于Vue组件中的计算属性
- actions定义了修改状态或执行异步操作的方法

3. 状态管理

3.3 初始化Pinia

- 在组件中使用已定义的store非常简单。

```
1. <template>
2.   <div>
3.     <p>Count: {{ counterStore.count }}</p>
4.     <p>Double Count: {{ counterStore.doubleCount }}</p>
5.     <button @click="counterStore.increment">Increment</button>
6.     <button @click="counterStore.decrement">Decrement</button>
7.   </div>
8. </template>
9.
10. <script setup>
11. import { useCounterStore } from '../stores/counter'
12.
13. const counterStore = useCounterStore()
14. </script>
```

- 上述代码中，通过useCounterStore()获取counter store实例，即可在组件中访问和修改store中的状态，并调用其方法

3. 状态管理

3.3 初始化Pinia

- 高阶技巧

- 数据持久化插件可将store状态持久化存储，防止数据丢失。例如使用localStorage存储状态，在组件重新加载时恢复先前状态，确保数据的连续性和可用性。

```
1. // plugins/persist.js
2. export const persistPlugin = ({ store }) => {
3.   // 从localStorage恢复状态
4.   const savedState = localStorage.getItem(store.$id)
5.   if (savedState) {
6.     store.$patch(JSON.parse(savedState))
7.   }
8.
9.   // 监听变化自动保存
10.  store.$subscribe((mutation, state) => {
11.    localStorage.setItem(store.$id, JSON.stringify(state))
12.  })
13. }
```

```
1. import { createPinia } from 'pinia'
2. import piniaPluginPersistedstate from 'pinia-plugin-persistedstate'
3.
4. const pinia = createPinia()
5. pinia.use(piniaPluginPersistedstate)
```

3. 状态管理

3.3 初始化Pinia

- 高阶技巧

- Pinia提供插件系统，允许开发者在全局层面扩展其功能。例如，可创建简单的日志插件，记录每次状态变化。

```
1. import { createPinia } from 'pinia'
2.
3. const loggerPlugin = (context) => {
4.   const { store } = context
5.   const oldState = {...store.$state }
6.
7.   store.$subscribe((mutation, state) => {
8.     console.log(`[Pinia Logger] ${mutation.type} in ${store.$id}`)
9.     console.log('Old State:', oldState)
10.    console.log('New State:', state)
11.    Object.assign(oldState, state)
12.  })
13. }
14.
15. const pinia = createPinia()
16. pinia.use(loggerPlugin)
17.
18. export default pinia
```

- 上述代码中，定义了loggerPlugin插件，通过pinia.use(loggerPlugin)将插件应用到Pinia实例。此后每次store状态发生变化时，控制台都会输出相应的日志信息。



4. 数据请求

4.1 Axios简介

- Axios是一个基于Promise的HTTP客户端库，专为浏览器和Node.js环境设计，用于发送各类HTTP请求。它在本质上是对原生XMLHttpRequest的现代化封装，完全遵循Promise规范并与最新的ECMAScript标准兼容。Axios具备出色的框架适配性，能够与主流前端框架无缝集成。其核心特性包括：
 - 支持在浏览器环境中创建XMLHttpRequests；
 - 支持在Node.js环境中发起HTTP请求；
 - 全面支持Promise API；
 - 提供请求和响应拦截机制；
 - 支持请求数据和响应数据的自动转换；
 - 支持请求取消功能；
 - 自动实现JSON数据转换；
 - 提供客户端XSRF攻击防护支持。



4. 数据请求

4.2 环境配置

- 在项目中集成Axios前，需要通过包管理工具进行安装。以下是两种常用的安装方式：
- 使用npm安装的执行命令如下

```
1. npm install axios --save
```

- 使用yarn安装的执行命令如下

```
1. yarn add axios
```

4. 数据请求

4.3 Axios请求方法

- GET请求

- GET方法主要用于从服务器获取数据资源，请求参数可通过URL查询字符串或params配置项传递。典型使用示例如下：

```
1. // 方式1: 直接拼接参数到 URL
2. axios.get('/api/user?id=123')
3.   .then(response => {
4.     console.log('请求成功: ', response.data); // 响应体数据在 data 中
5.   })
6.   .catch(error => {
7.     console.log('请求失败: ', error.message);
8.   });
9.
10. // 方式2: 通过 params 配置传递参数 (推荐, 自动拼接 URL)
11. axios.get('/api/user', {
12.   params: { id: 123, name: '张三' } // 最终 URL 为 /api/user?id=123&name=张三
13. })
14. .then(res => console.log(res.data))
15. .catch(err => console.log(err));
```

4. 数据请求

4.3 Axios请求方法

● POST请求

- POST方法通常用于向服务器提交数据，特别适用于表单提交和数据创建场景。请求参数通过data配置项传递，默认采用JSON格式进行数据传输。具体实现示例如下：

```
1. // 提交 JSON 数据（默认 Content-Type: application/json）
2. axios.post('/api/user', {
3.   name: '张三',
4.   age: 25,
5.   email: 'zhangsan@example.com'
6. })
7. .then(res => console.log('提交成功: ', res.data))
8. .catch(err => console.log('提交失败: ', err));
9.
10. // 提交表单数据（需手动设置 Content-Type）
11. const formData = new FormData();
12. formData.append('name', '张三');
13. formData.append('file', fileObj); // 上传文件
14.
15. axios.post('/api/upload', formData, {
16.   headers: { 'Content-Type': 'multipart/form-data' } // 表单数据格式
17. })
18. .then(res => console.log('文件上传成功'))
19. .catch(err => console.log('上传失败'));
```

4. 数据请求

4.3 Axios请求方法

- 请求别名

- Axios为所有标准HTTP方法提供了便捷的别名接口，这些别名的使用方法与GET/POST方法类似，极大提升了开发效率。具体应用示例如下：

```
1. axios.request(config)
2. axios.get(url[, config])
3. axios.delete(url[, config])
4. axios.head(url[, config])
5. axios.options(url[, config])
6. axios.post(url[, data[, config]])
7. axios.put(url[, data[, config]])
8. axios.patch(url[, data[, config]])
9. axios.postForm(url[, data[, config]])
10. axios.putForm(url[, data[, config]])
11. axios.patchForm(url[, data[, config]])
```

4. 数据请求

4.3 Axios请求方法

- 配置式请求

- 所有请求都可以通过统一的axios(config)方法进行配置，这种方式特别适合需要高度自定义请求参数的复杂场景。配置示例如下：

```
1. axios({
2.   method: 'post', // 请求方法 (get/post/put/delete 等)
3.   url: '/api/user', // 请求地址
4.   params: { type: 'admin' }, // URL 参数 (拼接在 URL 后)
5.   data: { name: '张三', age: 25 }, // 请求体数据 (POST/PUT 等用)
6.   headers: { 'Authorization': 'Bearer ' + token }, // 自定义请求头
7.   timeout: 5000 // 请求超时时间 (毫秒)，超过则报错
8. })
9. .then(res => console.log(res.data))
10. .catch(err => console.log(err));
```



4. 数据请求

4.4 拦截器

- Axios的拦截器机制允许开发者在请求发送前和响应接收后进行预处理。
- 这一特性非常适合实现全局性的请求头管理、错误处理等通用功能。
- 通过`interceptors.request.use()`可以添加统一的认证信息。
- `interceptors.response.use()`则可实现响应数据的统一处理。

4. 数据请求

4.4 拦截器

- 请求拦截器

- 请求拦截器在请求被发送到服务器之前执行，常用于添加统一的请求头信息（如身份验证token、内容类型设置）、对敏感数据进行加密处理，以及显示请求加载状态提示。典型实现示例如下：

```
1. // 添加请求拦截器
2. axios.interceptors.request.use(
3.   // 拦截成功（请求发送前执行）
4.   config => {
5.     // 1. 统一添加 token（登录后从本地存储获取）
6.     const token = localStorage.getItem('token');
7.     if (token) {
8.       config.headers.Authorization = 'Bearer ' + token;
9.     }
10.
11.    // 2. 统一设置请求超时时间
12.    config.timeout = 10000;
13.
14.    // 3. 显示加载状态（如调用 UI 库的 loading 组件）
15.    // Loading.show();
16.
17.    return config; // 必须返回 config，否则请求会中断
18.  },
19.  // 拦截失败（请求配置错误时执行）
20.  error => {
21.    // 隐藏加载状态
22.    // Loading.hide();
23.    return Promise.reject(error); // 传递错误，供 catch 捕获
24.  }
25.);
```

4. 数据请求

4.4 拦截器

● 响应拦截器

- 响应拦截器在服务器返回响应数据后、业务逻辑处理前执行，可用于实现响应数据的统一格式化、全局错误处理机制的实施，以及加载状态的隐藏操作，确保前端应用能够更加稳定高效地处理响应数据。具体实现示例如下：

```
1. // 添加响应拦截器
2. axios.interceptors.response.use(
3.   // 拦截成功（响应状态码 2xx 时执行）
4.   response => {
5.     // 1. 隐藏加载状态
6.     // Loading.hide();
7.
8.     // 2. 过滤数据：直接返回响应体的 data 字段，简化业务代码
9.     return response.data;
10.  },
11.   // 拦截失败（响应状态码非 2xx 或网络错误时执行）
12.   error => {
13.     // 1. 隐藏加载状态
14.     // Loading.hide();
15.
16.     // 2. 统一错误处理
17.     if (error.response) {
18.       // 有响应体（状态码 4xx/5xx）
19.       const status = error.response.status;
20.       switch (status) {
21.         case 401: // 未登录或 token 过期
22.           alert('登录已过期，请重新登录');
23.           localStorage.removeItem('token'); // 清除无效 token
24.           window.location.href = '/login'; // 跳转到登录页
25.           break;
26.         case 403: // 权限不足
27.           alert('您没有权限执行此操作');
28.           break;
```

```
29.         case 500: // 服务器错误
30.           alert('服务器繁忙，请稍后再试');
31.           break;
32.         default:
33.           alert('请求失败: ' + error.response.data.message);
34.       }
35.     } else if (error.request) {
36.       // 无响应体（网络错误、超时等）
37.       alert('网络异常，请检查网络连接');
38.     }
39.
40.     return Promise.reject(error); // 传递错误，供业务代码捕获
41.   }
42.);
```



4. 数据请求

4.4 拦截器

- 若需临时禁用某个拦截器，可以通过拦截器实例提供的移除方法来实现，具体操作示例如下：

```
1. // 1. 保存拦截器实例
2. const requestInterceptor = axios.interceptors.request.use(config => config);
3.
4. // 2. 移除拦截器
5. axios.interceptors.request.eject(requestInterceptor);
```

4. 数据请求

4.5 Axios的请求配置

- 在使用Axios发起请求时，可以通过配置对象来自定义请求行为。其中url参数是必填项，如果未指定method参数，则默认使用GET方法。常用的配置选项详细说明如下：

```
1. {
2.   // url 是用于请求的服务器URL
3.   url: '/user',
4.
5.   // method 是创建请求时使用的方法
6.   method: 'get',
7.
8.   // baseURL 将自动加在url前面，除非url是一个绝对URL
9.   // 它可以通过设置一个baseURL 便于为axios实例的方法传递相对URL
10.  baseURL: 'https://some-domain.com/api/',
11.
12.  // headers 是即将被发送的自定义请求头
13.  headers: { 'x-Requested-With': 'XMLHttpRequest' },
14.
15.  // params 是即将与请求一起发送的URL参数
16.  // 必须是一个无格式对象（plain object）或 URLSearchParams 对象
17.  params: {
18.    ID: 12345,
19.  },
20.
21.  // data 是作为请求主体被发送的数据，只适用于这些请求方式：PUT、POST和PATCH
22.  // 在没有设置转换请求时，必须是以下类型之一
23.  // string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams
24.  // 浏览器专属：FormData, File, Blob
25.  // Node 专属：Stream
26.  data: {
27.    firstName: 'First',
28.  },
29.
30.  // timeout 指定请求超时的毫秒数（0 表示无超时时间）
31.  // 如果请求花费了超过 timeout 的时间，请求将被中断
32.  timeout: 1000,
```

```
33.   // responseType 表示服务器响应的数据类型，可以是以下几种
34.   // arraybuffer、blob、document、json、text、stream
35.   responseType: 'json', // default
36.
37.   // proxy 定义代理服务器的主体名称和端口
38.   // auth 表示 HTTP 基础验证应当用于连接代理，并提供凭据
39.   // 这将会设置一个 Proxy-Authorization 头，覆写掉已有的通过使用 header 设置
40.   // 的自定义
41.   // Proxy-Authorization 头
42.   proxy: {
43.     host: '127.0.0.1',
44.     port: 9000,
45.     auth: {
46.       username: 'mikeymike',
47.       password: 'rapunz31'
48.     }
49.   },
50. }
```

4. 数据请求

4.6 Axios的响应结构

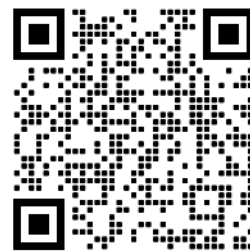
- Axios请求返回的响应对象包含以下重要信息：

```
1. {  
2.   // data 由服务器提供的响应  
3.   data: {},  
4.  
5.   // status 来自服务器响应的 HTTP 状态码  
6.   status: 200,  
  
7.   // statusText 来自服务器响应的 HTTP 状态信息  
8.   statusText: 'OK',  
  
9.   // headers 服务器响应的头  
10.  headers: {},  
11.  
12.  // config 是为请求提供的配置信息  
13.  config: {},  
  
14.  // request 是生成当前响应的请求  
15.  // 在 node.js 中是最后一个 ClientRequest 实例（在重定向中）  
16.  // 在浏览器中是 XMLHttpRequest 示例  
17.  request: {},  
18. }
```

```
1. axios.get('/user/12345')  
2.   .then(function(response) {  
3.     console.log(response.data);  
4.     console.log(response.status);  
5.     console.log(response.statusText);  
6.     console.log(response.headers);  
7.     console.log(response.config);  
8.   })
```

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