第 3 章 组件化开发

学习目的与要求

本章主要讲解组件的注册和引用、组件间的数据传递、动态组件与异步组件、插槽、自定义指令、Element Plus 组件库等内容。通过本章的学习，掌握注册组件的方法，能够运用全局注册或者局部注册的方式完成组件的注册；掌握组件的引用方法，能够在 Vue.js 项目中以标签的形式引用组件；掌握父组件向子组件传递数据的方法，能够使用 props 实现数据传递；掌握子组件向父组件传递数据的方法，能够使用自定义事件实现数据传递；掌握跨级组件之间传递数据的方法，能够使用依赖注入实现数据共享；掌握动态组件的使用方法，能够实现动态组件的渲染；掌握插槽的定义和使用；了解自定义指令的声明和使用；掌握 Element Plus 中常用组件的使用方法，能够实现按钮、表格、表单和菜单效果。

本章主要内容

- 组件的注册和引用
- 组件间的样式冲突
- 组件间的数据传递
- 动态组件与异步组件
- 插槽
- 自定义指令
- 引用静态资源
- Element Plus 组件库

组件(Component)是 Vue.js 最核心的功能，是可扩展的 HTML 元素(可看作自定义的 HTML 元素)，是封装可重用的代码，同时也是 Vue.js 实例。

组件系统是 Vue.js 中的一个重要概念，它提供了一种抽象，让人们可以使用独立可复用的组件来构建大型应用，任意类型的应用界面都可以抽象为一个组件树。这种前端组件化方便 UI 组件的重用。

本章将和读者一起由浅入深地学习组件的相关内容。

扫一扫

视频讲解

3.1 组件的注册和引用

为了能在 UI 模板中使用组件，必须先注册组件，以便 Vue.js 识别。

▶ 3.1.1 组件的注册

组件有两种注册类型，分别为全局注册和局部注册。在注册组件的时候，需要给组件取一个名字，从而区分每个组件，可以采用帕斯卡命名法(驼峰式)为组件命名。

❶ 全局注册

在实际工程中，如果某个组件的使用频率很高，即许多组件中都会引用该组件，则推荐将

该组件全局注册。被全局注册的组件可以在当前 Vue.js 项目的任何一个组件内引用。

在 Vue.js 项目的 main.js 文件中，通过 Vue.js 应用实例的 component()方法可以全局注册组件，该注册方法的语法格式如下。

```
component('组件名称', 需要被注册的组件)
```

component()方法接收两个参数，第 1 个参数为组件名称，在注册完成后即可全局使用该组件名称，第 2 个参数为需要被注册的组件。示例代码如下。

```
import { createApp } from 'vue';
import App from './App.vue'
import MyComponent from './MyComponent.vue'
const app = createApp(App)
app.component('MyComponent', MyComponent)
app.mount('#app')
```

component()方法支持链式调用，可以连续注册多个组件，示例代码如下。

```
app.component('ComponentA', ComponentA).component('ComponentB', ComponentB)
```

❷ 局部注册

在实际工程中，如果某个组件只在特定的情况下被用到，推荐进行局部注册。局部注册即在某个组件中注册，被局部注册的组件只能在当前注册范围内使用。局部注册组件的示例代码如下。

```
<script>
import ComponentA from './ComponentA.vue'
export default {
  components: { ComponentA: ComponentA }
}
</script>
```

在使用 setup 语法糖时，导入的组件会被自动注册，导入后可以直接在模板中使用，示例代码如下。

```
<script setup>
import ComponentA from './ComponentA.vue'
</script>
```

▶ 3.1.2 组件的引用

在将组件注册完成后，若要将组件在页面中渲染出来，需要引用组件。

在组件的<template>标签中可以引用其他组件，被引用的组件需要写成标签的形式，标签名应与组件名对应。组件的标签名可以使用短横线分隔或使用帕斯卡命名法命名。例如，<my-component>标签和<MyComponent>标签都表示引用 MyComponent 组件。一个组件可以被引用多次，但不可以自我引用和互相引用，否则会出现死循环。

下面通过一个实例演示组件的注册和引用。

【例 3-1】 演示组件的注册和引用。在该实例中首先创建 GlobalComponent 全局组件和 LocalComponent 局部组件，然后创建 ComponentUse 组件，并在该组件中引用全局组件和局部组件。

其具体实现步骤如下。

（1）打开命令行窗口，切换到 D:\vuevite 目录，在该目录下执行 npm create vite@latest 命令创建项目 ch3。在项目创建完成后，执行 cd ch3 命令切换到项目目录，继续执行 npm install 命令安装项目依赖。

（2）使用 VS Code 打开 D:\vuevite\ch3 目录。

（3）在项目 ch3 的 src\components 目录中创建 GlobalComponent.vue 文件（全局组件）和 LocalComponent.vue 文件（局部组件）。

GlobalComponent.vue 文件的代码如下。

```
<template>
    <div id="global-component">
      <h5>Global Component</h5>
    </div>
  </template>
  <script setup></script>
  <style>
  #global-component {
    border: 1px solid red;
    height: 50px;
    flex: 1;
  }
  </style>
```

LocalComponent.vue 文件的代码如下。

```
<template>
    <div id="local-component">
      <h5>This is a local component</h5>
    </div>
  </template>
  <script setup></script>
  <style>
  #local-component {
    border: 1px dashed green;
    height: 50px;
    flex: 1;
  }
  </style>
```

（4）在项目 ch3 的 src 目录中创建 ComponentUse.vue 文件，并在该文件中引用全局组件和局部组件，代码如下。

```
<template>
    <div>
      <h5>Component Use</h5>
      <div class="box">
          <global-component />
          <LocalComponent />
     </div>
    </div>
  </template>
  <script setup>
    import LocalComponent from './components/LocalComponent.vue'
```

```
</script>
<style>
.box {
  display: flex;
}
</style>
```

(5) 修改 main.js 中的全局注册组件 GlobalComponent，并切换页面中显示的组件 ComponentUse，代码如下。

```
import { createApp } from 'vue'
//import './style.css'
import App from './ComponentUse.vue'
import GlobalComponent from './components/GlobalComponent.vue'
createApp(App).component('global-component', GlobalComponent).mount('#app')
```

(6) 在 Terminal 终端执行 npm run dev 命令启动服务。

(7) 在浏览器中访问 http://localhost:5173/，页面效果如图 3.1 所示。

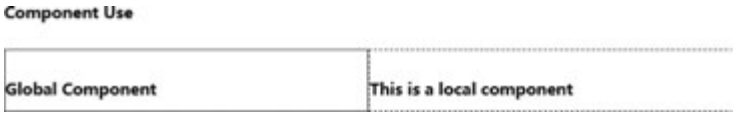

图 3.1 例 3-1 的页面效果

从图 3.1 可以看出，已经将全局组件 GlobalComponent 和局部组件 LocalComponent 引用并显示到 ComponentUse 组件中。

3.2 组件间的样式冲突

在默认情况下，写在 Vue.js 组件中的样式会全局生效，很容易造成多个组件之间出现样式冲突的问题。例如，为例 3-1 的 ComponentUse 组件中的 h5 元素添加边框样式，具体代码如下。

```
h5 {
 border: 1px dotted black;
}
```

这时，在浏览器中访问 http://localhost:5173/，页面效果如图 3.2 所示。

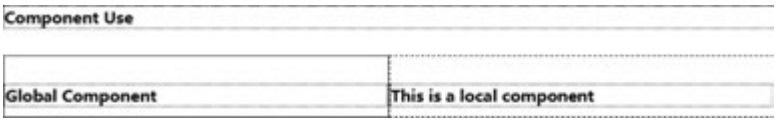

图 3.2 例 3-1 修改后的页面效果

从图 3.2 可以看出，为 ComponentUse 组件中的 h5 元素添加的边框样式也作用到 GlobalComponent 全局组件和 LocalComponent 局部组件的 h5 元素上，进而引起组件样式冲突的问题。

导致组件间出现样式冲突的原因是，在单页应用中所有组件的 DOM 结构都是基于唯一的 index.html 页面呈现的。每个组件中的样式都可以影响整个页面中的 DOM 元素。在

Vue.js 中可以使用< style >标签的 scoped 属性来解决组件间样式冲突的问题。

在为< style >标签添加 scoped 属性后，Vue.js 将自动为当前组件的 DOM 元素添加一个唯一的自定义属性，并在样式中为选择器添加自定义属性，从而限制样式的作用范围，防止组件间出现样式冲突的问题。

修改 ComponentUse 组件，为< style >标签添加 scoped 属性，即< style scoped >。然后在浏览器中再次访问 http://localhost:5173/，页面效果如图 3.3 所示。

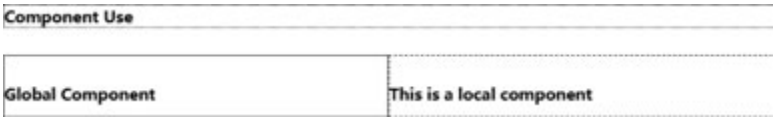

图 3.3　解决样式冲突后的页面效果

从图 3.3 可以看出，为 ComponentUse 组件中的 h5 元素添加的边框样式并没有作用到 GlobalComponent 全局组件和 LocalComponent 局部组件的 h5 元素上，因此解决了组件间样式冲突的问题。

3.3　组件间的数据传递

Vue.js 组件间的数据传递有多种场景，包括父子组件、兄弟组件、组件链以及任意组件间的数据传递。

扫一扫

视频讲解

▶ 3.3.1　父组件向子组件传递数据

如果需要实现父组件向子组件传递数据，需要先在子组件中声明 props，表示子组件从父组件中接收哪些数据。

在不使用 setup 语法糖的情况下，子组件可以使用 props 选项声明 props。props 选项的形式可以是对象或字符串数组。示例代码如下。

```
< script >
export default {
  props: {
    自定义属性 A: 类型,
    自定义属性 B: 类型,
    ……
  }
}
</script >
```

如果不需要 props 的类型，子组件可以声明字符串数组形式的 props，示例代码如下。

```
props: ['自定义属性 A', '自定义属性 B']
```

如果使用 setup 语法糖，子组件可以使用 defineProps()函数声明 props，示例代码如下。

```
< script setup >
    const props = defineProps({'自定义属性 A': 类型}, {'自定义属性 B': 类型})
</script >
```

子组件使用 defineProps()函数声明字符串数组形式的 props，示例代码如下。

```
const props = defineProps(['自定义属性 A', '自定义属性 B'])
```

在子组件中声明了 props 之后，可以直接在子组件模板中输出每个 prop 的值（从父组件接收的数据），示例代码如下。

```
<template>
 {{ 自定义属性 A }}
 {{ 自定义属性 B }}
</template>
```

当在父组件中引用了子组件之后，如果在子组件中声明了 props，则可以在父组件中向子组件传递数据。如果传递的数据是固定值，则可以通过静态绑定 props 的方式为子组件传递数据。

在父组件中通过静态绑定 props 的方式为子组件传递数据，属性值默认为字符串类型。示例代码如下。

```
<子组件标签名 自定义属性 A="数据" 自定义属性 B="数据" />
```

在父组件中可以使用 v-bind 为子组件动态绑定 props，任意类型的值都可以传递给子组件的 props。示例代码如下。

```
<子组件标签名 :自定义属性 A="变量 A" :自定义属性 B="变量 B" />
```

下面通过一个实例演示父组件向子组件如何传递数据。

【例 3-2】 演示父组件向子组件如何传递数据。在 Vue.js 项目 ch3 的 src 目录中创建 FatherProps.vue 文件（父组件），在项目 ch3 的 src\components 目录中创建 SonProps.vue 文件（子组件），并通过声明 props 实现父组件向子组件传递数据。

其具体实现步骤如下。

（1）在项目 ch3 的 src\components 目录中创建 SonProps.vue 文件（子组件），在子组件中使用 defineProps()函数声明 props 接收父组件传递的数据。SonProps.vue 文件的代码如下。

```
<template>
    <div>
        <h2>SonProps</h2>
        <p>Name: {{ name }}</p>
        <p>Age: {{ age }}</p>
        <p>User Age: {{ userAge }}</p>
        <p>PI: {{ constPI }}</p>
        <p>User Prop: {{ userProp }}</p>
        <p>Hobby Prop: {{ hobbyProp }}</p>
    </div>
</template>
<script setup>
//const props = defineProps(['name', 'age', 'userAge', 'constPI', 'userProp', 'hobbyProp'])
const props = defineProps({
    name: {
        type: String,
        default: 'Son'
    },
    age: {
        type: Number,
```

```
            default: 18
        },
        userAge: {
            type: Number,
            default: 20
        },
        constPI: {
            type: Number,
            default: 3.14
        },
        userProp: {
            type: Object,
            default: () => ({})
        },
        hobbyProp: {
            type: Array,
            default: () => []
        }
    })
    </script>
```

（2）在项目 ch3 的 src 目录中创建 FatherProps.vue 文件（父组件），在父组件中使用 v-bind 为子组件动态绑定 props 传递数据。FatherProps.vue 文件的代码如下。

```
<template>
    <div>
        <h1>FatherProps</h1>
        <!-- 动态绑定父组件的属性、常量、对象属性、数组属性 -->
        <SonProps :name="name" :age="age"
        :userAge="user.age" constPI="3.1425926"
        :userProp="user" :hobbyProp="hobby"/>
    </div>
</template>
<script setup>
  import { ref, reactive } from 'vue';
  import SonProps from './components/SonProps.vue'
  const name = ref('John')
  const age = ref(25)
  const user = reactive({
    name: 'Tom',
    age: 30
  })
  const hobby = ref(['唱歌', '跳舞', '滑冰'])
</script>
```

（3）修改 main.js，切换页面中显示的组件文件 FatherProps.vue，代码如下。

```
import { createApp } from 'vue'
import App from './FatherProps.vue'
createApp(App).mount('#app')
```

（4）在 Terminal 终端执行 npm run dev 命令启动服务。

（5）在浏览器中访问 http://localhost:5173/，页面效果如图 3.4 所示。

从图 3.4 可以看出，父组件 FatherProps 引用了子组件 SonProps，并为子组件动态绑定

props 传递数据。

需要注意的是，在 Vue.js 中所有的 props 都遵循单向数据流原则，props 数据因父组件的更新而变化，变化后的数据将向下传递给子组件，而且不会逆向传递，这样可以防止因子组件意外变更 props 导致数据流向难以理解的问题。每次父组件绑定的 props 发生变更时，子组件中的 props 都将自动刷新为最新的值。在子组件内部不能改变 props，如果这样做，Vue.js 会在浏览器的控制台中发出警告。

图 3.4 父组件 FatherProps 的页面效果

扫一扫 视频讲解

3.3.2 子组件向父组件传递数据

通过 3.3.1 节的学习，大家知道在子组件中声明 props 只能实现父组件向子组件传递数据，并且这种数据传递是单向的。那么如何实现子组件向父组件传递数据呢？

可以通过自定义事件的方式实现子组件向父组件传递数据，具体步骤为首先在子组件中声明自定义事件；然后在子组件中触发自定义事件；最后在父组件中监听自定义事件。

❶ 声明自定义事件

当在子组件中不使用 setup 语法糖时，可以通过 emits 选项声明自定义事件，示例代码如下。

```
<script>
export default {
  emits: ['myself']
}
</script>
```

当在子组件中使用 setup 语法糖时，需要通过调用 defineEmits()函数声明自定义事件，示例代码如下。

```
<script setup>
    const emit = defineEmits(['myself'])
</script>
```

上述代码中的 myself 为事件名称。

❷ 触发自定义事件

当使用场景简单时，可以通过内联方式调用 $emit()方法触发自定义事件，将数据传递给使用的组件(父组件)，示例代码如下。

```
<button @click="$emit('myself', data)">传递</button>
```

在上述代码中，$emit()方法的第 1 个参数为自定义事件的名称(字符串类型)，第 2 个参数为需要传递的数据，当触发当前组件(子组件)的事件时，该数据(data)将传递给父组件。

除了内联方式外，还可以直接定义方法来触发自定义事件。在不使用 setup 语法糖时，可以从 setup()函数的第 2 个参数(即 setup 上下文对象)来访问 emit()方法，示例代码如下。

```
export default {
  setup(props, ctx) {
    const update = () => {
```

```
            ctx.emit('myself', 10)
        }
        return { update }
    }
}
```

如果使用 setup 语法糖，可以调用 emit()函数来实现，示例代码如下。

```
<script setup>
const transfer = () => {
    emit('myself', 10)
}
</script>
```

❸ 监听自定义事件

在父组件中可以通过 v-on 监听子组件中抛出的事件，示例代码如下。

```
<子组件名 @myself = "handle" />
```

在上述示例代码中，当子组件触发 myself 事件时，父组件将接收到从子组件中传递的参数数据，同时执行 handle()方法。在 handle()方法中，父组件可以通过参数接收从子组件中传递的数据。示例代码如下。

```
const handle = (value) => {
    console.log(value)
}
```

下面通过一个实例演示子组件如何向父组件传递数据。

【例 3-3】 演示子组件如何向父组件传递数据。在 Vue.js 项目 ch3 的 src 目录中创建 FatherEmit.vue 文件(父组件)，在项目 ch3 的 src\components 目录中创建 SonEmit.vue 文件(子组件)，并在子组件 SonEmit 中通过自定义事件向父组件 FatherEmit 传递数据。

其具体实现步骤如下。

(1) 在项目 ch3 的 src\components 目录中创建 SonEmit.vue 文件(子组件)，在子组件中使用 defineEmits()函数声明自定义事件，并调用 $emit()方法触发自定义事件。SonEmit.vue 文件的代码如下。

```
<template>
    <h2>子组件</h2>
    <button @click = "$emit('myself1', data1)">传递 1</button>
    <br><br>
    <button @click = "$emit('myself2', data2)">传递 2</button>
</template>
<script setup>
import { ref, reactive } from 'vue';
const emit = defineEmits(['myself1', 'myself2'])
const data1 = ref('传递的数据 1')
const data2 = reactive({ name: '传递的数据 2', age: 20 })
</script>
```

(2) 在项目 ch3 的 src 目录中创建 FatherEmit.vue 文件(父组件)，在父组件中使用 v-on 监听子组件中抛出的事件，接收并显示子组件传递的数据。FatherEmit.vue 文件的代码如下。

```
<template>
  <div>
    <h1>父组件</h1>
    <SonEmit @myself1 = "handle1" @myself2 = "handle2" />
    <h1>父组件接收到子组件的自定义事件1传递的数据：{{ d1 }}</h1>
    <h1>父组件接收到子组件的自定义事件2传递的数据：{{ d2.name }} -- {{ d2.age }}</h1>
  </div>
</template>
<script setup>
 import { ref,reactive } from 'vue';
 import SonEmit from './components/SonEmit.vue'
 const d1 = ref('')
 const d2 = reactive({
    name: '',
    age: ''
 })
 const handle1 = (msg) => {
    d1.value = msg
 }
 const handle2 = (msg) => {
    d2.name = msg.name
    d2.age = msg.age
 }
</script>
```

（3）修改 main.js，切换页面中显示的组件文件 FatherEmit.vue，代码如下。

```
import { createApp } from 'vue'
import App from './FatherEmit.vue'
createApp(App).mount('#app')
```

（4）在 Terminal 终端执行 npm run dev 命令启动服务。

（5）在浏览器中访问 http://localhost:5173/，页面效果如图 3.5 所示。

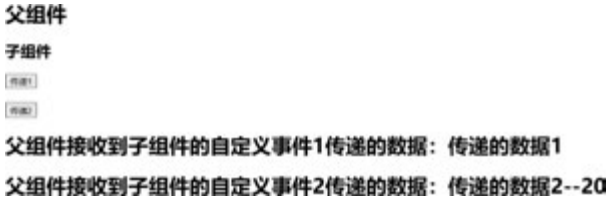

图 3.5 父组件 FatherEmit 的页面效果

单击图 3.5 中子组件的“传递 1”按钮与“传递 2”按钮，可以在父组件 FatherEmit 中显示子组件 SonEmit 传递的数据。

扫一扫 视频讲解

▶ 3.3.3 跨级组件之间的数据传递

Vue.js 提供了跨级组件间数据传递的方式——依赖注入。一个父组件相对于其所有后代组件而言，可作为依赖提供者，而任何后代的组件树，无论层级多深，都可以注入由父组件提供的依赖。

对于父组件而言，如果要为后代组件提供数据，需要使用 provide()函数。对于子组件而言，如果想要注入上层组件或整个应用提供的数据，需要使用 inject()函数。

❶ provide()函数

在父组件中使用provide()函数可以提供一个值，用于被其后代组件注入。provide()函数的语法格式如下。

```
provide(注入名, 注入值)
```

provide()函数可以接收两个参数，第1个参数是注入名，后代组件将通过注入名查找所需的注入值；第2个参数是注入值，值可以是任意类型，包括响应式数据。

在不使用setup语法糖的情况下，provide()函数必须在组件的setup()函数中调用。示例代码如下。

```
<script>
import { ref, provide } from 'vue'
export default {
  setup() {
    const count = ref(100)
    provide( 'message', count )
  }
}
</script>
```

在使用setup语法糖时，父组件使用provide()函数的示例代码如下。

```
<script setup>
import { provide } from 'vue'
provide('message', 'Hello World')
</script>
```

❷ inject()函数

在子组件中通过inject()函数可以注入上层组件或者整个应用提供的数据。inject()函数的语法格式如下。

```
inject(注入值, 默认值, 布尔值)
```

第1个参数是注入值，Vue.js向上遍历父组件，通过匹配注入的值来确定所提供的值，如果父组件链上有多个组件为同一个数据变量提供了值，那么距离更近的父组件将会覆盖更远的父组件所提供的值。

第2个参数是可选的，为没有匹配到注入的值时使用的默认值。第2个参数可以是工厂函数，用于返回某些创建起来比较复杂的值。如果提供的默认值是函数，还需要将false作为第3个参数传入，表明这个函数就是默认值，而不是工厂函数。

第3个参数是可选的，类型为布尔值，当参数值为false时，表示默认值是函数；当参数值为true时，表示默认值为工厂函数；当省略参数值时，表示默认值为其他类型的数据，不是函数或工厂函数。

在不使用setup语法糖的情况下，inject()函数必须在子组件的setup()函数中调用。使用inject()函数的示例代码如下。

```
<script>
import { inject } from 'vue';
export default {
  setup() {
    const count = inject('mesg1')
```

```
    const foo = inject('mesg2', 'default value')
    const baz = inject('mesg3', () => new Map())
    const fn = inject('mesg4', () => { }, false)
  }
}
</script>
```

在使用 setup 语法糖的情况下，子组件使用 inject()函数的示例代码如下。

```
<script setup>
import { inject } from 'vue';
const count = inject('mesg')
</script>
```

下面通过一个实例演示跨级组件之间的数据传递。

【例 3-4】 演示跨级组件之间的数据传递。在 Vue.js 项目 ch3 的 src 目录中创建 GrandProvide.vue 文件(祖父组件，使用 provide()函数提供依赖数据)，在项目 ch3 的 src\components 目录中创建 FatherInject.vue 文件(父组件)和 SonInject.vue 文件(子组件)，在 FatherInject 和 SonInject 组件中通过 inject()函数注入数据。

其具体实现步骤如下。

(1) 在项目 ch3 的 src\components 目录中创建 FatherInject.vue 文件(父组件)和 SonInject.vue 文件(子组件)，在 FatherInject 和 SonInject 组件中通过 inject()函数注入数据。

FatherInject.vue 文件的代码如下。

```
<template>
    <div>
        <h4>Father Inject</h4>
        <p>{{ mes }}</p>
        <SonInject />
    </div>
</template>
<script setup>
import { inject } from 'vue'
import SonInject from './SonInject.vue';
const mes = inject('message')
</script>
```

SonInject.vue 文件的代码如下。

```
<template>
    <div>
        <h5>SonInject</h5>
        <p>{{ state.weight }}</p>
        <p>{{ state.age }}</p>
        <p>{{ state.height }}</p>
    </div>
</template>
<script setup>
import { inject } from 'vue'
const state = inject('state')
</script>
```

(2) 在项目 ch3 的 src 目录中创建 GrandProvide.vue 文件(祖父组件)，在祖父组件中使

用 provide()函数提供依赖数据。GrandProvide.vue 文件的代码如下。

```
<template>
    <div>
        <h2>Grand Provide</h2>
        <FatherInject />
    </div>
</template>
<script setup>
import { reactive, provide } from 'vue'
import FatherInject from './components/FatherInject.vue';
const state = reactive({
    weight: 190,
    age: 45,
    height: 180
})
provide('state', state)
provide('message', 'This is a message')
</script>
```

（3）修改 main.js，切换页面中显示的组件文件 GrandProvide.vue，代码如下。

```
import { createApp } from 'vue'
import App from './GrandProvide.vue'
createApp(App).mount('#app')
```

（4）在 Terminal 终端执行 npm run dev 命令启动服务。

（5）在浏览器中访问 http://localhost:5173/，页面效果如图 3.6 所示。

图 3.6　祖父组件 GrandProvide 的页面效果

从图 3.6 可以看出，祖父组件 GrandProvide 提供的数据在其后代组件中依次注入并显示。

扫一扫

视频讲解

▶ 3.3.4　任意组件之间的数据传递

在 Vue.js 中，推荐使用一个空的 Vue 实例作为媒介（中央事件总线）实现父子组件、兄弟组件及组件链的数据传递。例如在人们的生活中，买房卖房中介帮忙，买卖双方通过房产中介（中央事件总线）实现需求对接。

在 Vue 2.x 中，Vue 实例可以通过事件触发 API（$on、$off 和 $once）实现中央事件总线，但是在 Vue 3.x 中移除了 $on、$off 和 $once 实例方法，推荐使用外部库 mitt 来代替 $on、$off 和 $once 实例方法。

首先在 Vue.js 项目的 Terminal 终端执行 npm install mitt 命令安装外部库 mitt。

然后在 Vue.js 项目的 main.js 中导入 mitt，并创建中央事件总线。示例代码如下。

```
//引入 mitt
import mitt from 'mitt'
//创建中央总线
const bus = mitt()
//将 bus 挂载成全局变量
app.config.globalProperties.$bus = bus
```

最后使用中央事件总线进行组件间的数据传递,具体使用方式如下。

在一个组件中调用中央事件总线的 emit(event, data)方法触发事件 event(字符串事件名)并向另一个组件传递数据 data。示例代码如下。

```
myBus.emit('send1', myDate.nowDate(new Date()) + " 0101 战情: " + form1.content1)
```

在另一个组件中调用中央事件总线的 on(event, 函数)方法监听事件 event 并接收另一个组件传递的数据(函数的参数)。示例代码如下。

```
myBus.on('send1', (msg) => {
    //msg 为组件传递的数据
})
```

下面通过一个实例演示任意组件之间的数据传递。

【例 3-5】 演示任意组件之间的数据传递。在 Vue.js 项目 ch3 的 src 目录中创建 FamilyMitt.vue 文件;在项目 ch3 的 src\components 目录中创建 BrotherMitt.vue 文件和 SisterMitt.vue 文件,并使用中央事件总线实现 BrotherMitt 组件和 SisterMitt 组件的数据传递;在 FamilyMitt 组件中导入 BrotherMitt 组件和 SisterMitt 组件。

其具体实现步骤如下。

(1) 在项目 ch3 的 Terminal 终端执行 npm install mitt 命令安装外部库 mitt。

(2) 在项目 ch3 的 src\components 目录中创建 BrotherMitt.vue 文件(哥哥组件)和 SisterMitt.vue 文件(妹妹组件)。

BrotherMitt.vue 文件的代码如下。

```
<template>
    <h3>我是老哥</h3>
    <input class="my-input" type="text" v-model="form1.content1" />
    <button class="my-button" @click="onSubmit1">发送</button>
    <br>
    老妹说:<textarea disabled v-model="form1.desc1" cols="50" rows="5"></textarea>
</template>
<script setup>
import { reactive, onMounted } from 'vue'
import { getCurrentInstance } from 'vue'
//获取全局变量 bus
const myBus = getCurrentInstance().appContext.config.globalProperties.$bus
let form1 = reactive({
    content1: '',
    desc1: ''
})
let scenario1 = ''
const onSubmit1 = () => {
    myBus.emit('sendSister', " 老哥说: " + form1.content1 + "\n")
```

```
}
onMounted(() => {
    myBus.on('sendBrother', (msg) => {
        scenario1 = scenario1 + msg
        form1.desc1 = scenario1
    })
})
</script>
<style scoped>
.my-input {
    display: inline-block;
    width: 200px;
}

.my-button {
    display: inline-block;
}
</style>
```

SisterMitt.vue 文件的代码如下。

```
<template>
    <h3>我是老妹</h3>
    <input class="my-input" type="text" v-model="form2.content2" />
    <button class="my-button" @click="onSubmit2">发送</button>
    <br>
    老哥说：<textarea disabled v-model="form2.desc2" cols="50" rows="5"></textarea>
</template>
<script setup>
import { reactive, onMounted } from 'vue'
import { getCurrentInstance } from 'vue'
//获取全局变量 bus
const myBus = getCurrentInstance().appContext.config.globalProperties.$bus
let form2 = reactive({
    content2: '',
    desc2: '',
})
let scenario2 = ''
const onSubmit2 = () => {
    myBus.emit('sendBrother', "老妹说：" + form2.content2 + "\n")
}
onMounted(() => {
    myBus.on('sendSister', (msg) => {
        scenario2 = scenario2 + msg
        form2.desc2 = scenario2
    })
})
</script>
<style scoped>
.my-input {
    display: inline-block;
    width: 200px;
}
.my-button {
    display: inline-block;
}
</style>
```

（3）在项目 ch3 的 src 目录中创建 FamilyMitt.vue 文件（家庭组件），在家庭组件中导入哥哥组件和妹妹组件。FamilyMitt.vue 文件的代码如下。

```
<template>
    <BrotherMitt />
    <SisterMitt />
</template>
<script setup>
import BrotherMitt from "./components/BrotherMitt.vue"
import SisterMitt from "./components/SisterMitt.vue"
</script>
```

（4）修改 main.js，引入 mitt 库，创建中央事件总线，并切换页面中显示的组件文件 FamilyMitt.vue，代码如下。

```
import { createApp } from 'vue'
import App from './FamilyMitt.vue'
//引入 mitt
import mitt from 'mitt'
//创建中央总线
const bus = mitt()
const app = createApp(App)
//将 bus 挂载成全局变量
app.config.globalProperties.$bus = bus
app.mount('#app')
```

（5）在 Terminal 终端执行 npm run dev 命令启动服务。

（6）在浏览器中访问 http://localhost:5173/，页面效果如图 3.7 所示。

图 3.7 使用中央事件总线实现任意组件之间的数据传递

扫一扫 视频讲解

3.4 动态组件与异步组件

组件间切换或异步加载是常见的应用场景，本节将介绍动态与异步组件的实现方法。

▶ 3.4.1 动态组件

在实际应用中，切换界面是常见的场景。那么在 Vue.js 中如何按需切换界面组件呢？虽然可以使用条件渲染指令来实现界面组件的切换，但采用这种方式会使代码变得臃肿。因此，在 Vue.js 中采用动态组件的方式来实现界面组件按需切换。

Vue.js 使用< component >标签可以定义动态组件，语法格式如下。

```
<component :is="要渲染的组件"></component>
```

在上述语法格式中，<component>标签必须配合 is 属性一起使用，is 属性的值表示要渲染的组件，当该属性值发生变化时，页面中渲染的组件也会随之发生变化。is 属性的值可以是字符串或组件，当属性值为组件时，如果要实现组件切换，需要调用 shallowRef()函数将组件保存为响应式数据。

下面通过一个实例演示动态组件的用法。

【例 3-6】 演示动态组件的用法。在 Vue.js 项目 ch3 的 src 目录中创建 DynamicComponent.vue 文件，在项目 ch3 的 src\components 目录中创建 DynamicLeft.vue 文件和 DynamicRight.vue 文件，在 DynamicComponent 组件中通过<component>标签动态显示 DynamicLeft 和 DynamicRight 组件。

其具体实现步骤如下。

(1) 在项目 ch3 的 src\components 目录中创建 DynamicLeft.vue 文件(左侧组件)和 DynamicRight.vue 文件(右侧组件)。

DynamicLeft.vue 文件的代码如下。

```
<template>
    <div>左侧动态组件</div>
</template>
<script setup>
</script>
```

DynamicRight.vue 文件的代码如下。

```
<template>
    <div>右侧动态组件</div>
</template>
<script setup>
</script>
```

(2) 在项目 ch3 的 src 目录中创建 DynamicComponent.vue 文件(动态组件)，在动态组件中通过<component>标签动态显示左侧组件和右侧组件。DynamicComponent.vue 文件的代码如下。

```
<template>
    <button @click="change('left')">左侧组件</button>
      
    <button @click="change('right')">右侧组件</button>
    <div>
        <component :is="showMyComponent"></component>
    </div>
</template>
<script setup>
import DynamicLeft from './components/DynamicLeft.vue'
import DynamicRight from './components/DynamicRight.vue'
import { shallowRef } from 'vue'
const showMyComponent = shallowRef(DynamicLeft)
const change = (direction) => {
    if (direction === 'left') {
        showMyComponent.value = DynamicLeft
```

```
        } else {
            showMyComponent.value = DynamicRight
        }
    }
    </script>
```

(3) 修改 main.js,切换页面显示的组件文件 DynamicComponent.vue,代码如下。

```
import { createApp } from 'vue'
import App from './DynamicComponent.vue'
const app = createApp(App)
app.mount('#app')
```

(4) 在 Terminal 终端执行 npm run dev 命令启动服务。

(5) 在浏览器中访问 http://localhost:5173/,页面效果如图 3.8 所示。

图 3.8 动态组件的页面效果

单击图 3.8 中的“左侧组件”按钮,将显示左侧组件的界面内容;单击图 3.8 中的“右侧组件”按钮,将显示右侧组件的界面内容,这说明动态组件实现了界面组件按需切换。

▶ 3.4.2 使用 KeepAlive 组件实现组件缓存

在使用动态组件实现组件间的按需切换时,隐藏的组件被销毁,显示的组件被重新创建。因此,当一个组件被销毁后又重新创建时,组件无法保持销毁前的状态。如果在多个组件之间进行动态切换时想要保持这些组件的状态,以及避免重复渲染导致的性能问题,可以通过组件缓存来实现。

在 Vue.js 中可以通过< KeepAlive >标签来实现组件缓存。使用< KeepAlive >标签包裹需要被缓存的组件。< KeepAlive >标签的语法格式如下。

```
<KeepAlive>
     被缓存的组件
</KeepAlive>
```

下面通过一个实例演示组件缓存的用法。

【例 3-7】 演示组件缓存的用法。

该实例在例 3-6 的基础上实现,具体实现步骤如下。

(1) 修改 DynamicRight.vue 文件,为了方便演示组件缓存,添加响应式数据 state,修改后的代码如下。

```
<template>
    <div>右侧动态组件</div>
    <div>
        <!-- state 变量是为了演示组件缓存而定义的 -->
        state 值为: {{ state }}
        <button @click="state++">改变状态</button>
    </div>
</template>
<script setup>
import { ref } from 'vue'
const state = ref(0)
</script>
```

（2）修改 DynamicComponent.vue 文件，使用<KeepAlive>标签实现动态组件缓存，修改后的代码如下。

```
<template>
    <button @click="change('left')">左侧组件</button>
      
    <button @click="change('right')">右侧组件</button>
    <div>
        <!-- 缓存组件 -->
        <KeepAlive>
            <component :is="showMyComponent"></component>
        </KeepAlive>
    </div>
</template>
<script setup>
import DynamicLeft from './components/DynamicLeft.vue'
import DynamicRight from './components/DynamicRight.vue'
import { shallowRef } from 'vue'
const showMyComponent = shallowRef(DynamicLeft)
const change = (direction) => {
    if (direction === 'left') {
        showMyComponent.value = DynamicLeft
    } else {
        showMyComponent.value = DynamicRight
    }
}
</script>
```

（3）在浏览器中访问 http://localhost:5173/，页面效果与图 3.8 相同，这时单击“右侧组件”按钮，页面效果如图 3.9 所示。

连续单击图 3.9 中的“改变状态”按钮 3 次，state 值为 3，页面效果如图 3.10 所示。

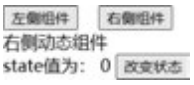

图 3.9　右侧组件的页面效果

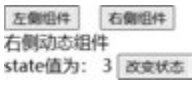

图 3.10　state 值发生变化

在单击图 3.10 中的“左侧组件”按钮后单击“右侧组件”按钮，页面效果与图 3.10 一样，这说明<KeepAlive>标签实现了动态组件的状态缓存。

▶ 3.4.3　异步组件

在大型应用中，可能需要将应用分割成许多小的代码块，并且只在需要时从服务器异步加载一个模块，这样可以避免一开始就把所有组件加载，浪费不必要的开销。

Vue.js 提供了 defineAsyncComponent()函数将组件定义为一个工厂函数，动态地解析组件。异步组件，只在组件需要渲染时触发工厂函数，并把结果缓存起来，以备再次渲染。

下面通过一个实例演示异步组件的用法。

【例 3-8】　演示异步组件的用法。在 Vue.js 项目 ch3 的 src 目录中创建 AsyncComponentTest.vue 文件（测试异步组件），在项目 ch3 的 src\components 目录中创建 AsyncComponent.vue 文件（异步组件）和 NotAsyncComponent.vue 文件（非异步组件）。

其具体实现步骤如下。

（1）在项目 ch3 的 src\components 目录中创建 AsyncComponent.vue 文件（异步组件）和 NotAsyncComponent.vue 文件（非异步组件）。

AsyncComponent.vue 文件的代码如下。

```
<template>
    <div id="async-component">
        <h4>异步组件,5 秒钟后显示</h4>
    </div>
</template>
<script setup>
</script>
<style>
#async-component {
    background-color: red;
}
</style>
```

NotAsyncComponent.vue 文件的代码如下。

```
<template>
    <div id="not-async-component">
         Not Async Component
    </div>
</template>
<script setup>
</script>
<style>
#not-async-component {
    background-color: greenyellow;
}
</style>
```

（2）在项目 ch3 的 src 目录中创建 AsyncComponentTest.vue 文件，在 AsyncComponentTest.vue 文件中使用 defineAsyncComponent() 函数将 AsyncComponent 组件定义为异步组件。AsyncComponentTest.vue 文件的代码如下。

```
<template>
    <div>
        <NotAsyncComponent />
        <MyAsyncComponent />
    </div>
</template>
<script setup>
import AsyncComponent from './components/AsyncComponent.vue';
import NotAsyncComponent from './components/NotAsyncComponent.vue';
import { defineAsyncComponent } from 'vue'
const MyAsyncComponent = defineAsyncComponent(() => //定义异步组件
    new Promise((resolve, reject) => {              //返回 Promise 的工厂函数
        window.setTimeout(() => {                   //window.setTimeout 只是演示异步
            /*在从服务器接收到加载组件定义后,调用 Promise 的 resolve 方法异步加载组件,
            也可以调用 reject(reason)指示加载失败.*/
            resolve(AsyncComponent)
        }, 5000)
    })
)
</script>
```

（3）修改 main.js，切换页面中显示的组件文件 AsyncComponentTest.vue，代码如下。

```
import { createApp } from 'vue'
import App from './AsyncComponentTest.vue'
const app = createApp(App)
app.mount('#app')
```

（4）在 Terminal 终端执行 npm run dev 命令启动服务。

（5）在浏览器中访问 http://localhost:5173/，页面效果如图 3.11 所示。

图 3.11 显示 5 秒钟后，页面效果变成如图 3.12 所示。

图 3.11　异步组件显示前的页面效果

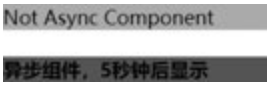

图 3.12　异步组件显示的页面效果

从图 3.11 和图 3.12 可以看出，5 秒钟后才从服务器异步加载异步组件，这样就可以避免一开始就把所有组件加载，浪费不必要的开销。

3.5　插槽

一个网页有时由多个模块组成，例如：

```
<div class = "container">
    <header>
        <!-- 我们希望把页头放这里 -->
    </header>
    <main>
        <!-- 我们希望把主要内容放这里 -->
    </main>
    <footer>
        <!-- 我们希望把页脚放这里 -->
    </footer>
</div>
```

这时需要使用插槽混合父组件的内容与子组件的模板。那么插槽如何定义和使用呢？下面进行学习。

扫一扫

视频讲解

▶ 3.5.1　插槽的定义与使用

插槽是组件封装时为组件的使用者预留的占位符，允许组件的使用者在组件内展示特定的内容。通过插槽，可以使组件更灵活、更具有可复用性。注意，插槽在定义后才能使用。

❶ 插槽的定义

在封装组件时，可以通过<slot>标签定义插槽，从而在组件中预留占位符。在<slot>标签内可以添加插槽的默认内容。如果组件的使用者没有为插槽提供任何内容，则默认内容生效；如果组件的使用者为插槽提供了内容，则该插槽内容会替代默认内容。示例代码如下。

```
<slot>
      <p>插槽内容，默认内容!</p>
</slot>
```

❷ 插槽的使用

使用插槽，需要在父组件中将子组件写成双标签的形式，然后在双标签内传入内容，这些内容会被填充到子组件定义的相应插槽中。示例代码如下。

```
<SlotSubComponent>
    <div>
        <h2>给插槽组件传递的内容</h2>
    </div>
</SlotSubComponent>
```

因为插槽内容是在父组件模板中定义的，所以在插槽内容中可以访问到父组件的数据。插槽内容可以是任意合法的模板内容，不局限于文本，可以使用多个元素或者组件作为插槽内容，示例代码如下。

```
<SlotSubComponent>
    <div>
        <h2>给插槽组件传递的内容</h2>
        <!-- 在插槽内容中访问父组件的数据 -->
        <p>{{ message }}</p>
    </div>
    <!-- 在插槽内容中使用组件 -->
    <DynamicLeft />
</SlotSubComponent>
```

下面通过一个实例演示插槽的定义与使用。

【例 3-9】 演示插槽的定义与使用。在 Vue.js 项目 ch3 的 src 目录中创建 SlotUser.vue 文件，在项目 ch3 的 src\components 目录中创建 SlotSubComponent.vue 文件，在 SlotSubComponent 组件中定义插槽，在 SlotUser 组件中使用插槽。

其具体实现步骤如下。

(1) 在项目 ch3 的 src\components 目录中创建 SlotSubComponent.vue 文件(插槽组件)，在 SlotSubComponent.vue 中定义插槽。SlotSubComponent.vue 文件的代码如下。

```
<template>
    <div>
        <h2>SlotSubComponent</h2>
    </div>
    <slot>
        <p>插槽内容，默认内容!</p>
    </slot>
</template>
<script setup>
</script>
```

(2) 在项目 ch3 的 src 目录中创建 SlotUser.vue 文件，在 SlotUser.vue 中使用插槽组件。SlotUser.vue 文件的代码如下。

```
<template>
    <div>
        <h1>Slot User</h1>
    </div>
    <SlotSubComponent>
```

```
        <div>
            <h2>给插槽组件传递的内容</h2>
            <!-- 在插槽内容中访问父组件的数据 -->
            <p>{{ message }}</p>
        </div>
        <!-- 在插槽内容中使用组件 -->
        <DynamicLeft />
    </SlotSubComponent>
</template>
<script setup>
import SlotSubComponent from './components/SlotSubComponent.vue'
import DynamicLeft from './components/DynamicLeft.vue'
const message = 'Hello, Slot!'
</script>
```

（3）修改 main.js，切换页面中显示的组件文件 SlotUser.vue，代码如下。

```
import { createApp } from 'vue'
import App from './SlotUser.vue'
const app = createApp(App)
app.mount('#app')
```

（4）在 Terminal 终端执行 npm run dev 命令启动服务。

（5）在浏览器中访问 http://localhost:5173/，页面效果如图 3.13 所示。

图 3.13　使用插槽组件的页面效果

从图 3.13 可以看出，父组件 SlotUser 使用了插槽子组件 SlotSubComponent，并在插槽内容中访问了父组件的数据，还使用了 DynamicLeft 组件。

扫一扫 视频讲解

▶ 3.5.2　具名插槽

在 Vue.js 实际工程中，当需要定义多个插槽时，可以通过具名插槽来区分不同的插槽。所谓的具名插槽，就是给每个插槽定义一个名称，这样就可以在对应插槽中提供对应数据。

<slot>标签的 name 属性用于给每个插槽分配唯一的名称，以确定每一处要渲染的内容。定义具名插槽的语法格式如下。

```
<slot name="插槽名称"></slot>
```

在父组件中，可以通过一个包含 v-slot 指令的<template>标签将内容填充到指定名称的插槽中，语法格式如下。

```
<组件名>
    <template v-slot:插槽名称></template>
</组件名>
```

与 v-bind 类似，v-slot 也有简写形式，可以将 v-slot:替换为 #。例如，v-slot:mySlot 可以简写为 #mySlot。

下面通过一个实例演示具名插槽的用法。

【例 3-10】 演示具名插槽的用法。在 Vue.js 项目 ch3 的 src 目录中创建 NameSlotUse.vue 文件，在项目 ch3 的 src\components 目录中创建 NameSlot.vue 文件，在 NameSlot 组件中定义具名插槽，在 NameSlotUse 组件中使用具名插槽组件 NameSlot。

其具体实现步骤如下。

(1) 在项目 ch3 的 src\components 目录中创建 NameSlot.vue 文件(具名插槽组件)，在 NameSlot.vue 文件中定义具名插槽。NameSlot.vue 文件的代码如下。

```
<template>
    <div>
        <div class="header-box">
            <slot name="header"></slot>
        </div>
        <div class="content-box">
            <slot name="center"></slot>
        </div>
        <div class="footer-box">
            <slot name="footer"></slot>
        </div>
    </div>
</template>
<script setup>
</script>
<style>
.header-box {
    background-color: #1a9f3d;
    padding: 10px;
    text-align: center;
    font-size: 24px;
    font-weight: bold;
}
.content-box {
    background-color: #e71717;
    padding: 10px;
    text-align: center;
    font-size: 18px;
}
.footer-box {
    background-color: #655c5c;
    padding: 10px;
    text-align: center;
    font-size: 14px;
}
</style>
```

(2) 在项目 ch3 的 src 目录中创建 NameSlotUse.vue 文件(父组件)，在父组件中使用具名插槽组件 NameSlot。NameSlotUse.vue 文件的代码如下。

```
<template>
    <NameSlot>
        <template v-slot:header>
            <p>这是网页的头部区域</p>
        </template>
        <template #center>
            <p>这是网页的中间区域</p>
        </template>
        <template #footer>
            <p>这是网页的尾部区域</p>
        </template>
    </NameSlot>
</template>
<script setup>
import NameSlot from './components/NameSlot.vue'
</script>
```

（3）修改 main.js，切换页面中显示的组件文件 NameSlotUse.vue，代码如下。

```
import { createApp } from 'vue'
import App from './NameSlotUse.vue'
const app = createApp(App)
app.mount('#app')
```

（4）在 Terminal 终端执行 npm run dev 命令启动服务。

（5）在浏览器中访问 http://localhost:5173/，页面效果如图 3.14 所示。

图 3.14　NameSlotUse 组件的页面效果

扫一扫 视频讲解

▶ 3.5.3　作用域插槽

一般情况下，在父组件中不能使用子组件中定义的数据。如果需要在父组件中使用子组件中定义的数据，可以通过作用域插槽来实现。作用域插槽是带有数据的插槽，子组件提供一部分数据给插槽，父组件接收子组件的数据进行页面渲染。

作用域插槽的使用分为定义数据和接收数据两部分。

❶ 定义数据

在插槽中定义数据供父组件使用，示例代码如下。

```
<slot message="Hello World"></slot>
```

❷ 接收数据

使用默认插槽和具名插槽接收数据的方式不同，下面分别进行讲解。

1）默认插槽

在定义插槽时，如果省略了< slot >标签的 name 属性，则 name 属性默认为 default，这样的插槽称为默认插槽。

在父组件中可以通过 v-slot 指令接收插槽中定义的数据，接收到的数据可以在插槽内通过“{{ }}”语法进行访问。例如，在父组件中使用 MySlot 子组件中的插槽时，通过 v-slot 指令接收数据的示例代码如下。

```
<MySlot v-slot="param">
  <p>{{ param.message }}</p>
</MySlot>
```

在上述代码中 param 为形参，表示从作用域插槽中接收的数据，该形参的名称可以自定义。

作用域插槽对外提供的数据对象可以使用解构赋值，以简化数据的接收过程，示例代码如下。

```
<MySlot v-slot="{ message }">
  <p>{{ message }}</p>
</MySlot>
```

{ message }表示通过解构赋值解构对象，在解构后子组件中定义的数据可以直接访问，而不是以“形参.属性”的方式访问。

2）具名插槽

在具名插槽中也可以向父组件中传递数据。例如，在封装 MySlot 组件时向具名插槽中传入数据的示例代码如下。

```
<slot name="header" message="Hello World"></slot>
```

在使用具名插槽时，插槽属性可以作为 v-slot 的值被访问到，基本语法格式为“v-slot:插槽名称="形参"”，简写形式为“#插槽名称="形参"”，示例代码如下。

```
<MySlot>
      <template #header="{ message }">
          {{ message }}
      </template>
</MySlot>
```

如果在一个子组件中同时定义了默认插槽和具名插槽，并且它们均需要为父组件提供数据，这时就需要为默认插槽显式地使用< template >标签来接收数据，示例代码如下。

```
<MySlot>
  <template #default="{ message }">
    {{ message }}
  </template>
</MySlot>
```

下面通过一个实例演示作用域插槽的用法。

【例 3-11】 演示作用域插槽的用法。

在 Vue.js 项目 ch3 的 src 目录中创建 ScopeSlotUse.vue 文件，在项目 ch3 的 src\components 目录中创建 ScopeSlot.vue 文件，在 ScopeSlot 组件中使用插槽向 ScopeSlotUse 组件传递数据，

在 ScopeSlotUse 组件中接收插槽传递的数据。

其具体实现步骤如下。

（1）在项目 ch3 的 src\components 目录中创建 ScopeSlot.vue 文件（子组件），在子组件中使用作用域插槽向父组件传递数据。ScopeSlot.vue 文件的代码如下。

```
<template>
    <!-- 默认插槽传字符串数据 -->
    <slot message="Hello 默认插槽"></slot>
    <hr>
    <!-- 具名插槽传字符串数据 -->
    <slot message="Hello World 具名插槽 content" name="content"></slot>
    <hr>
    <!-- 具名插槽传对象数据 -->
    <slot :user="user" name="footer"></slot>
</template>
<script setup>
import { reactive } from 'vue'
const user = reactive({ name: 'zhangsan', age: '35' })
</script>
```

（2）在项目 ch3 的 src 目录中创建 ScopeSlotUse.vue 文件（父组件），在父组件中接收子组件通过作用域插槽传递的数据。ScopeSlotUse.vue 文件的代码如下。

```
<template>
    <ScopeSlot>
        <!-- { message }解构赋值 -->
        <template #default="{ message }">
            <p>{{ message }}</p>
        </template>
        <template #content="scope">
            <p>{{ scope }}</p>
            <p>{{ scope.message }}</p>
        </template>
        <!-- { user }解构赋值 -->
        <template #footer="{ user }">
            <p>{{ user.name }}</p>
            <p>{{ user.age }}</p>
        </template>
    </ScopeSlot>
</template>
<script setup>
import ScopeSlot from './components/ScopeSlot.vue'
</script>
```

（3）修改 main.js，切换页面中显示的组件文件 ScopeSlotUse.vue，代码如下。

```
import { createApp } from 'vue'
import App from './ScopeSlotUse.vue'
const app = createApp(App)
app.mount('#app')
```

（4）在 Terminal 终端执行 npm run dev 命令启动服务。

（5）在浏览器中访问 http://localhost:5173/，页面效果如图 3.15 所示。

图 3.15 父组件 ScopeSlotUse 的页面效果

从图 3.15 可以看出，父组件 ScopeSlotUse 接收并显示了子组件 ScopeSlot 通过作用域插槽传递的数据。

扫一扫 视频讲解

3.6 自定义指令

Vue.js 为用户提供了功能丰富的内置指令，例如 v-model、v-show 等。这些内置指令可以满足大部分业务需求，但有时用户需要一些特殊功能，例如对普通 DOM 元素进行底层操作。幸运的是，Vue.js 允许用户自定义指令，实现特殊功能。

▶ 3.6.1 自定义指令的概念

与组件类似，Vue.js 中的自定义指令分为局部自定义指令和全局自定义指令。

局部自定义指令是指在组件内部定义的指令，局部自定义指令只可以在定义该指令的组件内部使用。全局自定义指令是指在全局定义的指令，全局自定义指令可以在任何组件中使用。例如，在 src\main.js 文件中定义全局自定义指令，可以用于任何一个组件。

自定义指令由一个包含自定义指令生命周期函数的参数来定义，常用的自定义指令生命周期函数如下。

(1) created()：在绑定元素的属性前调用。

(2) beforeMount()：只调用一次，在绑定元素被挂载前调用，并进行初始化设置。

(3) mounted()：在绑定元素的父组件及自身的所有子节点都挂载完成后调用。

(4) beforeUpdate()：在绑定元素的父组件更新前调用。

(5) updated()：在绑定元素的父组件及自身的所有子节点都更新后调用。

(6) beforeUnmount()：在绑定元素的父组件卸载前调用。

(7) unmounted()：只调用一次，在绑定元素的父组件卸载后调用。

常用的自定义指令生命周期函数的参数如下。

(1) el：指令所绑定的元素，可以直接用于操作 DOM 元素。

(2) binding：一个对象，包含很多属性，用于接收属性的参数值。

(3) vnode：代表绑定元素底层的虚拟节点。

(4) prevNode：之前页面渲染中指令所绑定元素的虚拟节点。

binding 中包含以下 6 个常用属性。

(1) value：传递给指令的值。

(2) arg：传递给指令的参数。

(3) oldValue：之前的值，仅在 beforeUpdate()函数和 updated()函数中可用，无论值是否更改都可用。

(4) modifiers：一个包含修饰符的对象（如果有）。例如，在 v-my-directive.foo.bar 中修

饰符对象是{foo：true，bar：true}。

（5）instance：使用该指令的组件实例。

（6）dir：指令的定义对象。

▶ 3.6.2 局部自定义指令的声明与使用

在不使用 setup 语法糖时，可以在 directives 属性中声明局部自定义指令。例如，声明一个局部自定义指令 focus，示例代码如下。

```
export default {
  directives: {
    focus: {}
  }
}
```

在上述代码中，focus 为自定义指令的名称。名称为 focus 的指令指向一个配置对象，在该对象中可以包含自定义指令的生命周期函数，通过这些生命周期函数可以操作 DOM 元素。在使用自定义指令时，需要以“v-”开头，示例代码如下。

```
<input v-focus/>
```

在使用 setup 语法糖时，任何以“v”开头的采用驼峰式命名方法命名的变量都可以被用作一个自定义指令，示例代码如下。

```
<template>
  <input v-focus/>
</template>
<script setup>
        const vFocus = { }
</script>
```

▶ 3.6.3 全局自定义指令的声明与使用

全局自定义指令需要使用 Vue.js 应用实例的 directive()方法进行声明，语法格式如下。

```
directive('自定义指令名称', 对象)
```

directive()方法的第 1 个参数的类型为字符串，表示全局自定义指令的名称；第 2 个参数的类型为对象或者函数，用于接收指令的参数值。

例如，在 src\main.js 文件中声明全局自定义指令 fontSize。示例代码如下。

```
const app = createApp(App)
app.directive('fontSize', {
    mounted(el, binding) {
      el.style.fontSize = binding.value + 'px'
    }
})
app.mount('#app')
```

下面通过一个实例演示自定义指令的用法。

【例 3-12】 演示自定义指令的用法。在 Vue.js 项目 ch3 的 src 目录中创建 DirectiveUse.vue 文件，在项目的 main.js 中声明全局自定义指令 fontSize，在 DirectiveUse.vue 文件中声明局

部自定义指令 v-focus，在 DirectiveUse.vue 文件中使用自定义指令 fontSize 和 v-focus。

其具体实现步骤如下。

（1）修改 main.js，在项目的 main.js 中声明全局自定义指令 fontSize，并切换页面中显示的组件文件 DirectiveUse.vue。修改后 main.js 的代码如下。

```
import { createApp } from 'vue'
import App from './DirectiveUse.vue'
const app = createApp(App)
//全局自定义指令
app.directive('fontSize', {
    //updated()函数,实现自定义指令绑定的参数改变时页面进行同步更改
    updated(el, binding) {
        el.style.fontSize = binding.value + 'px'
    }
})
app.mount('#app')
```

（2）在项目 ch3 的 src 目录中创建 DirectiveUse.vue 文件，在 DirectiveUse.vue 文件中声明局部自定义指令 v-focus，并使用自定义指令 fontSize 和 v-focus。DirectiveUse.vue 文件的代码如下。

```
<template>
    <!-- 使用局部自定义指令 v-focus -->
    <input v-focus type="text" />
    <hr>
    <!-- 为自定义指令绑定参数 fontSize -->
    <span v-fontSize="fontSize">使用全局自定义指令</span>
    <hr>
    <button @click="change">更改字号大小</button>
</template>
<script setup>
import { ref } from 'vue'
const fontSize = ref(16)
const vFocus = {
    mounted(el) {
        alert('自定义指令 v-focus 被绑定到元素上')
        el.focus()
    }
}
const change = () => {
    fontSize.value++
}
</script>
```

（3）在 Terminal 终端执行 npm run dev 命令启动服务。

（4）在浏览器中访问 http://localhost:5173/，页面效果如图 3.16 所示。

图 3.16 DirectiveUse 组件的页面效果

从图 3.16 可以看出，光标焦点自动作用在文本框上，说明自定义指令 v-focus 已经生效。另外，单击“更改字号大小”按钮后，页面上的“使用全局自定义指令”文本的字号变大，说明全局自定义指令 v-fontSize 也已经生效。

3.7 引用静态资源

在组件中，有时需要使用一些静态资源，例如图片、CSS 资源等。Vue.js 项目的 public 目录和 src\assets 目录都可以存放静态资源，但引用静态资源的方式不同，具体如下。

❶ 引用 public 目录中的静态资源

public 目录用于存放不可编译的静态资源文件，例如图片、样式文件等。该目录下的文件会被复制到打包目录，该目录下的文件需要使用绝对路径访问。例如，在组件中引用 public 目录中的 logo.png 文件，示例代码如下。

```
<img src="/logo.png">
```

❷ 引用 src\assets 目录中的静态资源

src\assets 目录用于存放可编译的静态资源文件，例如图片、样式文件等。该目录下的文件需要使用相对路径访问。在引用 src\assets 目录中的图片时，首先将图片保存到本地，然后使用 import 语法将图片导入需要的组件，最后通过 img 元素的 src 属性添加图片的路径。示例代码如下。

```
<template>
  <img :src="icon">
</template>
<script setup>
import icon from '../assets/vue.svg'
</script>
```

扫一扫

视频讲解

3.8 Element Plus 组件库

Element Plus 是一套为开发者、设计师和产品经理准备的基于 Vue.js 3 的组件库，提供了配套设计资源，简化了常用组件的封装，大大降低了开发难度，助力使用者快速搭建网站。Element Plus 目前还处于快速迭代开发阶段，如果用户需要查阅官方文档，可以登录官网（https://cn.element-plus.org/zh-CN/），如图 3.17 所示。

图 3.17 Element Plus 官网的首页

3.8.1 Element Plus 的安装

在 Vue.js 项目中要想使用 Element Plus 组件库，需要事先安装 Element Plus，方法为首先使用 VS Code 打开需要使用 Element Plus 组件库的 Vue.js 项目，例如 ch3，然后在项目的 Terminal 终端执行 npm install element-plus --save 命令局部安装 Element Plus，--save 表示项目运行时依赖。

Element Plus 提供了一套常用的图标集合，但在默认情况下这些图标没有集成在组件中，需要另外安装才能使用，其安装方式与 Element Plus 的安装方式相同，即在 Vue.js 项目的 Terminal 终端执行 npm install @element-plus/icons-vue 命令安装。

在 Vue.js 项目中安装 Element Plus 和 Element Plus 图标后，还需要在 main.js 中引入 Element Plus 和注册 Element Plus 图标后才能在组件中使用它们。

❶ 引入并挂载 Element Plus 模块

在 Vue.js 项目的 src\main.js 文件中完整导入并挂载 Element Plus 模块。示例代码如下。

```
import {createApp} from 'vue'
import ElementPlus from 'element-plus'
import 'element-plus/dist/index.css'
import App from './App.vue'
const app = createApp(App)
app.use(ElementPlus)
app.mount('#app')
```

❷ 注册 Element Plus 图标

Element Plus 图标在 Vue.js 项目的 src\main.js 文件中注册后才能使用。示例代码如下。

```
import * as ElementPlusIconsVue from '@element-plus/icons-vue'
const app = createApp(App)
for (const [key, component] of Object.entries(ElementPlusIconsVue)) {
  app.component(key, component)
}
app.mount('#app')
```

3.8.2 Element Plus 组件介绍

Element Plus 组件主要包括基础类、表单类、数据展示类、导航类、反馈类等五大类，每一类又包含很多组件。下面简单介绍每一类组件中所包含的组件。

❶ 基础类组件

基础类组件包括 Button(按钮)、Border(边框)、Color(色彩)、Container(布局容器)、Icon(图标)、Layout(布局)、Link(链接)、Scrollbar(滚动条)、Space(间距)、Typography(排版)等组件。

❷ 表单类组件

表单类组件包括 Cascader(级联选择器)、Checkbox(复选框)、ColorPicker(颜色选择器)、DatePicker(日期选择器)、DateTimePicker(日期时间选择器)、Form(表单)、Input(输入框)、Input Number(数字输入框)、Radio(单选按钮)、Rate(评分)、Select(选择器)、Select V2(虚拟

列表选择器）、Slider（滑块）、Switch（开关）、TimePicker（时间选择器）、TimeSelect（时间选择）、Transfer（穿梭框）、Upload（上传）等组件。

❸ 数据展示类组件

数据展示类组件包括 Avatar（头像）、Badge（徽章）、Calendar（日历）、Card（卡片）、Carousel（走马灯）、Collapse（折叠面板）、Descriptions（描述列表）、Empty（空状态）、Image（图片）、Infinite Scroll（无限滚动）、Pagination（分页）、Progress（进度条）、Result（结果）、Skeleton（骨架屏）、Table（表格）、Virtualized Table（虚拟化表格）、Tag（标签）、Timeline（时间线）、Tree（树形控件）、TreeSelect（树形选择）、Tree V2（虚拟化树形控件）等组件。

❹ 导航类组件

导航类组件包括 Affix（固钉）、Backtop（回到顶部）、Breadcrumb（面包屑）、Dropdown（下拉菜单）、Menu（菜单）、Page Header（页头）、Steps（步骤条）、Tabs（标签页）等组件。

❺ 反馈类组件

反馈类组件包括 Alert（提示）、Dialog（对话框）、Drawer（抽屉）、Loading（加载）、Message（消息提示）、MessageBox（消息弹框）、Notification（通知）、Popconfirm（气泡确认框）、Popover（气泡卡片）、Tooltip（文字提示）等组件。

下面简单介绍几个常用的 Element Plus 组件。

3.8.3 Element Plus 中的常用组件之 Button

在 Element Plus 组件库中，Button 组件使用< el-button >标签定义，< el-button >标签的常用属性如表 3.1 所示。

表 3.1 ＜el-button＞标签的常用属性

属 性 名	属 性 值	描 述
type	primary	主要按钮
	success	成功按钮
	info	一般提示信息按钮
	warning	警告按钮
	danger	危险按钮
plain	true 或 false	是否为朴素按钮，默认值为 false
round	true 或 false	是否为圆角按钮，默认值为 false
disabled	true 或 false	是否为禁用按钮，默认值为 false
link	true 或 false	是否为链接按钮，默认值为 false
circle	true 或 false	是否为圆形按钮，默认值为 false

在上述属性中，如果需要设置 plain、round 或 circle 属性的值为 true，可以写成“:属性名＝"true""或“属性名”的形式。这里以 round 属性为例，示例代码如下。

```
< el - button type = "primary" :round = "true"> Primary </el - button >
```

或

```
< el - button type = "primary" round > Primary </el - button >
```

下面通过一个实例演示< el-button >标签的用法。

【例 3-13】 演示< el-button >标签的用法。

假设项目 ch3 的 main.js 中已经引入 Element Plus 和注册 Element Plus 图标，其他具体实现步骤如下。

（1）在 Vue.js 项目 ch3 的 src 目录中创建 ButtonUse.vue 文件，在 ButtonUse.vue 文件中使用 Element Plus 的 Button 组件。ButtonUse.vue 文件的代码如下。

```
<template>
    <div class="mb-4">
        <el-button>Default</el-button>
        <el-button type="primary">Primary</el-button>
        <el-button type="success">Success</el-button>
        <el-button type="info">Info</el-button>
        <el-button type="warning">Warning</el-button>
        <el-button type="danger">Danger</el-button>
    </div>
    <div class="mb-4">
        <el-button plain>Plain</el-button>
        <el-button type="primary" plain>Primary</el-button>
        <el-button type="success" plain>Success</el-button>
        <el-button type="info" plain>Info</el-button>
        <el-button type="warning" plain>Warning</el-button>
        <el-button type="danger" plain>Danger</el-button>
    </div>
    <div class="mb-4">
        <el-button round>Round</el-button>
        <el-button type="primary" round>Primary</el-button>
        <el-button type="success" round>Success</el-button>
        <el-button type="info" round>Info</el-button>
        <el-button type="warning" round>Warning</el-button>
        <el-button type="danger" round>Danger</el-button>
    </div>
    <div>
        <el-button :icon="Search" circle />
        <el-button type="primary" :icon="Edit" circle />
        <el-button type="success" :icon="Check" circle />
        <el-button type="info" :icon="Message" circle />
        <el-button type="warning" :icon="Star" circle />
        <el-button type="danger" :icon="Delete" circle />
    </div>
</template>
<script setup>
import {
    Check,
    Delete,
    Edit,
    Message,
    Search,
    Star,
} from '@element-plus/icons-vue'
</script>
```

（2）修改 main.js，切换页面中显示的组件文件 ButtonUse.vue，代码如下。

```
import { createApp } from 'vue'
import App from './ButtonUse.vue'
import ElementPlus from 'element-plus'
import 'element-plus/dist/index.css'
import * as ElementPlusIconsVue from '@element-plus/icons-vue'
const app = createApp(App)
app.use(ElementPlus)
for (const [key, component] of Object.entries(ElementPlusIconsVue)) {
    app.component(key, component)
}
app.mount('#app')
```

（3）在 Terminal 终端执行 npm run dev 命令启动服务。

（4）在浏览器中访问 http://localhost:5173/，页面效果如图 3.18 所示。

图 3.18　Element Plus 的 Button 组件的页面效果

3.8.4　Element Plus 中的常用组件之 Table

Element Plus 组件库提供了数据展示组件 Table，用于展示多条结构类似的数据，例如工资表、人员表和计划表等，以便对数据进行排序、筛选、对比等操作。在 Element Plus 组件库中，Table 组件使用< el-table >标签定义，在该标签中绑定 data 对象数组后，在< el-table-column >中使用 prop 属性对应对象中的属性名，即可将数据添加到表格的单元格中；使用 label 属性可以定义表格的列名，使用 width 属性可以定义表格的宽度。< el-table >标签的常用属性如表 3.2 所示。

表 3.2　＜el-table＞标签的常用属性

属　性　名	描　　述
data	显示的数据
stripe	是否添加斑马线，默认值为 false
order	是否带有纵向边框，默认值为 false

下面通过一个实例演示< el-table >标签的用法。

【例 3-14】　演示< el-table >标签的用法。

其具体实现步骤如下。

（1）在 Vue.js 项目 ch3 的 src 目录中创建 TableUse.vue 文件，在 TableUse.vue 文件中使用 Element Plus 的 Table 组件。TableUse.vue 文件的代码如下。

```
<template>
    <el-table :data="userList" style="width: 100%">
            <el-table-column prop="name" label="姓名" width="150" />
            <el-table-column prop="date" label="出生日期" width="150" />
            <el-table-column prop="sex" label="性别" width="150" />
            <el-table-column prop="address" label="住址" />
    </el-table>
</template>
```

```
<script setup>
const userList = [
    { name: '张三', date: '2005-10-01', sex: '男', address: '北京市海淀区' },
    { name: '李四', date: '2006-02-02', sex: '女', address: '上海市浦东新区' },
    { name: '王五', date: '2007-03-03', sex: '男', address: '深圳市南山区' },
    { name: '赵六', date: '2005-07-05', sex: '女', address: '大连市沙河口区' },
    { name: '田七', date: '2004-06-06', sex: '男', address: '大连市中山区' },
]
</script>
```

(2) 修改 main.js,切换页面中显示的组件文件 TableUse.vue,代码如下。

```
import { createApp } from 'vue'
import ElementPlus from 'element-plus'
import 'element-plus/dist/index.css'
import App from './TableUse.vue'
const app = createApp(App)
app.use(ElementPlus)
app.mount('#app')
```

(3) 在 Terminal 终端执行 npm run dev 命令启动服务。

(4) 在浏览器中访问 http://localhost:5173/,页面效果如图 3.19 所示。

姓名	出生日期	性别	住址
张三	2005-10-01	男	北京市海淀区
李四	2006-02-02	女	上海市浦东新区
王五	2007-03-03	男	深圳市南山区
赵六	2005-07-05	女	大连市沙河口区
田七	2004-06-06	男	大连市中山区

图 3.19 Element Plus 的 Table 组件的页面效果

▶ 3.8.5 Element Plus 中的常用组件之 Form

在实际工程中,大多数 Vue.js 项目都涉及表单界面,例如登录和注册界面。Element Plus 组件库提供了 Form 组件实现表单界面,该组件采用 Flex 布局,用于收集、验证和提交数据。基础的表单包含输入框、单选框、多选框等组件。

在 Element Plus 组件库中,Form 组件使用< el-form >标签定义,在该标签中使用< el-form-item >作为输入项的容器,用于获取值和验证值。< el-form >标签的常用属性如表 3.3 所示。

表 3.3 <el-form >标签的常用属性

属 性 名	描 述
inline	行内表单模式,默认值为 false,表示垂直表单
label-position	表单域标签的位置,默认值为 right(右对齐),left 表示左对齐,top 表示位于表单域的顶部
model	表单数据对象

Form 组件提供了表单验证的功能,只需要为< el-form >标签的 rules 属性传入约定的验证规则,并将 form-Item 的 prop 属性设置为需要验证的特殊键值即可。

下面通过一个实例演示<el-form>标签的用法。

【例 3-15】 演示<el-form>标签的用法。

其具体实现步骤如下。

(1) 在 Vue.js 项目 ch3 的 src 目录中创建 FormUse.vue 文件，在 FormUse.vue 文件中使用 Element Plus 的 Form 组件，并使用 rules 属性进行表单校验。FormUse.vue 文件的代码如下。

```
<template>
    <el-form ref="ruleFormRef" style="max-width: 600px" :model="ruleForm" :rules="rules" label-width="auto" class="demo-ruleForm" :size="formSize" status-icon>
        <el-form-item label="Activity name" prop="name">
            <el-input v-model="ruleForm.name" />
        </el-form-item>
        <el-form-item label="Activity zone" prop="region">
            <el-select v-model="ruleForm.region" placeholder="Activity zone">
                <el-option label="Zone one" value="shanghai" />
                <el-option label="Zone two" value="beijing" />
            </el-select>
        </el-form-item>
        <el-form-item label="Activity count" prop="count">
            <el-select-v2 v-model="ruleForm.count" placeholder="Activity count" :options="options" />
        </el-form-item>
        <el-form-item label="Activity time" required>
            <el-col :span="11">
                <el-form-item prop="date1">
                    <el-date-picker v-model="ruleForm.date1" type="date" aria-label="Pick a date" placeholder="Pick a date" style="width: 100%" />
                </el-form-item>
            </el-col>
            <el-col class="text-center" :span="2">
                <span class="text-gray-500">-</span>
            </el-col>
            <el-col :span="11">
                <el-form-item prop="date2">
                    <el-time-picker v-model="ruleForm.date2" aria-label="Pick a time" placeholder="Pick a time" style="width: 100%" />
                </el-form-item>
            </el-col>
        </el-form-item>
        <el-form-item label="Instant delivery" prop="delivery">
            <el-switch v-model="ruleForm.delivery" />
        </el-form-item>
        <el-form-item label="Activity location" prop="location">
            <el-segmented v-model="ruleForm.location" :options="locationOptions" />
        </el-form-item>
        <el-form-item label="Activity type" prop="type">
            <el-checkbox-group v-model="ruleForm.type">
                <el-checkbox value="Online activities" name="type">
                    Online activities
                </el-checkbox>
                <el-checkbox value="Promotion activities" name="type">
```

```
                    Promotion activities
                </el-checkbox>
                <el-checkbox value="Offline activities" name="type">
                    Offline activities
                </el-checkbox>
                <el-checkbox value="Simple brand exposure" name="type">
                    Simple brand exposure
                </el-checkbox>
            </el-checkbox-group>
        </el-form-item>
        <el-form-item label="Resources" prop="resource">
            <el-radio-group v-model="ruleForm.resource">
                <el-radio value="Sponsorship">Sponsorship</el-radio>
                <el-radio value="Venue">Venue</el-radio>
            </el-radio-group>
        </el-form-item>
        <el-form-item label="Activity form" prop="desc">
            <el-input v-model="ruleForm.desc" type="textarea" />
        </el-form-item>
        <el-form-item>
            <el-button type="primary" @click="submitForm(ruleFormRef)">
                Create
            </el-button>
            <el-button @click="resetForm(ruleFormRef)">Reset</el-button>
        </el-form-item>
    </el-form>
</template>
<script setup>
import { reactive, ref } from 'vue'
const formSize = ref('default')
const ruleFormRef = ref()
const ruleForm = reactive({
    name: 'Hello',
    region: '',
    count: '',
    date1: '',
    date2: '',
    delivery: false,
    location: '',
    type: [],
    resource: '',
    desc: '',
})
const locationOptions = ['Home', 'Company', 'School']
const rules = reactive({
    name: [
        { required: true, message: 'Please input Activity name', trigger: 'blur' },
        { min: 3, max: 5, message: 'Length should be 3 to 5', trigger: 'blur' },
    ],
    region: [
        {
            required: true,
            message: 'Please select Activity zone',
            trigger: 'change',
        },
```

```
        ],
        count: [
            {
                required: true,
                message: 'Please select Activity count',
                trigger: 'change',
            },
        ],
        date1: [
            {
                type: 'date',
                required: true,
                message: 'Please pick a date',
                trigger: 'change',
            },
        ],
        date2: [
            {
                type: 'date',
                required: true,
                message: 'Please pick a time',
                trigger: 'change',
            },
        ],
        location: [
            {
                required: true,
                message: 'Please select a location',
                trigger: 'change',
            },
        ],
        type: [
              {
                type: 'array',
                required: true,
                message: 'Please select at least one activity type',
                trigger: 'change',
            },
        ],
        resource: [
            {
                required: true,
                message: 'Please select activity resource',
                trigger: 'change',
            },
        ],
        desc: [
            { required: true, message: 'Please input activity form', trigger: 'blur' },
        ],
    })
    const submitForm = async (formEl) => {
        if (!formEl) return
        await formEl.validate((valid, fields) => {
            if (valid) {
                //提交表单数据
```

```
            console.log('submit!')
        } else {
            console.log('error submit!', fields)
        }
    })
}
const resetForm = (formEl) => {
    if (!formEl) return
    formEl.resetFields()
}
const options = Array.from({ length: 10000 }).map((_, idx) => ({
    value: `${idx + 1}`,
    label: `${idx + 1}`,
}))
</script>
```

(2) 修改 main.js,切换页面中显示的组件文件 FormUse.vue,代码如下。

```
import { createApp } from 'vue'
import ElementPlus from 'element-plus'
import 'element-plus/dist/index.css'
import App from './FormUse.vue'
const app = createApp(App)
app.use(ElementPlus)
app.mount('#app')
```

(3) 在 Terminal 终端执行 npm run dev 命令启动服务。

(4) 在浏览器中访问 http://localhost:5173/,页面效果如图 3.20 所示。

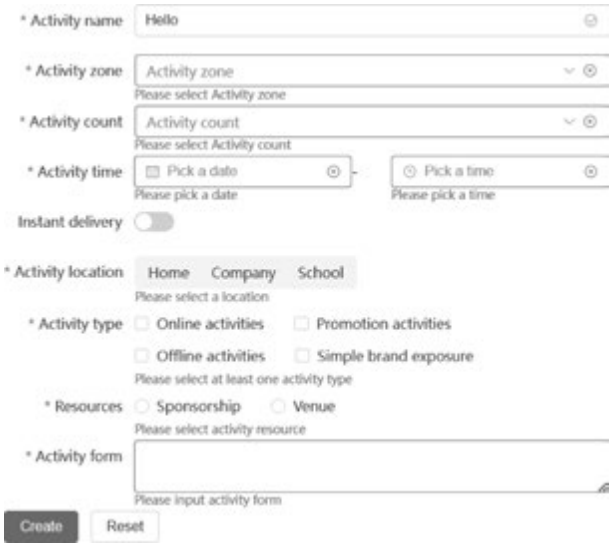

图 3.20 Element Plus 的 Form 组件的页面效果

3.8.6 Element Plus 中的常用组件之 Menu

菜单是网页设计中不可或缺的交互元素,通常位于页面的顶部或左侧,可以帮助用户快速定位和访问目标内容。Element Plus 组件库提供了 Menu 组件,能够高效地为网站提供导航功能。

在 Element Plus 组件库中，Menu 组件使用< el-menu >标签定义，在该标签中包含< el-menu-item >（菜单项）和< sub-menu >（子菜单）标签。菜单默认为垂直模式，通过将 mode 属性值设置为 horizontal，可以将菜单变为水平模式。

< el-menu >标签的常用属性如表 3.4 所示。

表 3.4 ＜el-menu＞标签的常用属性

属 性 名	描 述
mode	菜单展示模式，默认值为 vertical（垂直模式），horizontal 表示菜单为水平模式
collapse	是否水平折叠并收起菜单，默认值为 false
background-color	菜单的背景色
text-color	菜单的文字颜色，默认值为＃303133
active-text-color	当前激活菜单的文字颜色，默认值为＃409eff
default-active	页面加载时默认激活菜单的 index 属性

下面通过一个实例演示< el-menu >标签的用法。

【例 3-16】 演示< el-menu >标签的用法。

其具体实现步骤如下。

（1）在 Vue.js 项目 ch3 的 src 目录中创建 MenuUse.vue 文件，在 MenuUse.vue 文件中使用 Element Plus 的 Menu 组件。MenuUse.vue 文件的代码如下。

```
<template>
    <el-row class="tac">
        <el-col :span="12">
            <h5 class="mb-2">Default colors</h5>
            <el-menu default-active="2" class="el-menu-vertical-demo" @open=
"handleOpen" @close="handleClose">
                <el-sub-menu index="1">
                    <template #title>
                        <el-icon>
                            <location />
                        </el-icon>
                        <span>Navigator One</span>
                    </template>
                    <el-menu-item-group title="Group One">
                        <el-menu-item index="1-1">item one</el-menu-item>
                        <el-menu-item index="1-2">item two</el-menu-item>
                    </el-menu-item-group>
                    <el-menu-item-group title="Group Two">
                        <el-menu-item index="1-3">item three</el-menu-item>
                    </el-menu-item-group>
                    <el-sub-menu index="1-4">
                        <template #title>item four</template>
                        <el-menu-item index="1-4-1">item one</el-menu-item>
                    </el-sub-menu>
                </el-sub-menu>
                <el-menu-item index="2">
                    <el-icon><icon-menu /></el-icon>
                    <span>Navigator Two</span>
                </el-menu-item>
                <el-menu-item index="3" disabled>
                    <el-icon>
```

```
                        <document />
                    </el-icon>
                    <span>Navigator Three</span>
                </el-menu-item>
                <el-menu-item index="4">
                    <el-icon>
                        <setting />
                    </el-icon>
                    <span>Navigator Four</span>
                </el-menu-item>
            </el-menu>
        </el-col>
        <el-col :span="12">
            <h5 class="mb-2">Custom colors</h5>
            <el-menu active-text-color="#ffd04b" background-color="#545c64" class="el-menu-vertical-demo"
                default-active="2" text-color="#fff" @open="handleOpen" @close="handleClose">
                <el-sub-menu index="1">
                    <template #title>
                        <el-icon>
                            <location />
                        </el-icon>
                        <span>Navigator One</span>
                    </template>
                    <el-menu-item-group title="Group One">
                        <el-menu-item index="1-1">item one</el-menu-item>
                        <el-menu-item index="1-2">item two</el-menu-item>
                    </el-menu-item-group>
                    <el-menu-item-group title="Group Two">
                        <el-menu-item index="1-3">item three</el-menu-item>
                    </el-menu-item-group>
                    <el-sub-menu index="1-4">
                        <template #title>item four</template>
                        <el-menu-item index="1-4-1">item one</el-menu-item>
                    </el-sub-menu>
                </el-sub-menu>
                <el-menu-item index="2">
                    <el-icon><icon-menu /></el-icon>
                    <span>Navigator Two</span>
                </el-menu-item>
                <el-menu-item index="3" disabled>
                    <el-icon>
                        <document />
                    </el-icon>
                    <span>Navigator Three</span>
                </el-menu-item>
                <el-menu-item index="4">
                    <el-icon>
                        <setting />
                    </el-icon>
                    <span>Navigator Four</span>
                </el-menu-item>
            </el-menu>
        </el-col>
```

```
        </el-row>
    </template>
    <script setup>
    import {
        Document,
        Menu as IconMenu,
        Location,
        Setting,
    } from '@element-plus/icons-vue'
    const handleOpen = (key, keyPath) => {
        console.log(key, keyPath)
    }
    const handleClose = (key, keyPath) => {
        console.log(key, keyPath)
    }
    </script>
```

（2）修改 main.js，切换页面中显示的组件文件 MenuUse.vue，代码如下。

```
import { createApp } from 'vue'
import ElementPlus from 'element-plus'
import 'element-plus/dist/index.css'
import App from './MenuUse.vue'
const app = createApp(App)
app.use(ElementPlus)
app.mount('#app')
```

（3）在 Terminal 终端执行 npm run dev 命令启动服务。

（4）在浏览器中访问 http://localhost:5173/，页面效果如图 3.21 所示。

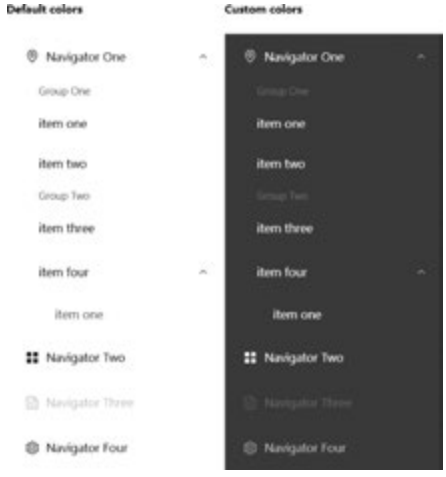

图 3.21　Element Plus 的 Menu 组件的页面效果

本章小结

本章详细介绍了组件的注册和引用、组件间的数据传递、动态组件与异步组件、插槽、自定义指令以及 Element Plus 组件库等内容。通过本章的学习，读者应掌握组件的注册和引用，理解组件间数据传递的基本原理，掌握插槽的定义和使用；了解自定义指令的声明和使用；掌握 Element Plus 中常用组件的使用方法。

习 题 3

1. 在 Vue.js 项目的 main.js 文件中，通过应用实例的(　　)方法可以全局注册组件。

A. components()　　B. registers()　　C. component()　　D. register()

2. 在 Vue.js 中可以使用< style >标签的(　　)属性解决组件间的样式冲突问题。

A. scoped　　B. local　　C. global　　D. private

3. 如果使用 setup 语法糖，子组件可以使用(　　)函数声明 props 接收父组件传递的数据。

A. defineProp()　　B. createProp()　　C. defineProps()　　D. createProps()

4. 对于父组件而言，如果要为后代组件提供数据，需要使用(①)函数。对于子组件而言，如果想要注入上层组件或整个应用提供的数据，需要使用(②)函数。

A. ①provide()　②inject()　　B. ①set()　②get()

C. ①push()　②get()　　D. ①provide()　②props()

5. 具名插槽通过给 slot 元素设置(　　)属性进行定义。

A. v-slot　　B. slot-scope　　C. name　　D. data

6. Vue.js 可以使用(　　)标签定义动态组件。

A. < components >　　B. < component >　　C. < slots >　　D. < slot >

7. 创建 Vue.js 项目 practice3_1；在项目 practice3_1 的 components 目录下创建 4 个组件，分别为 WarFirst.vue、WarSecond.vue、WarThird.vue、WarFourth.vue；在项目 practice3_1 的 App.vue 根组件中将 WarFirst.vue、WarSecond.vue、WarThird.vue、WarFourth.vue 等 4 个组件组合显示，如图 3.22 所示；在 WarFirst.vue、WarSecond.vue、WarThird.vue、WarFourth.vue 等 4 个组件中，使用中央总线分别向另外 3 个组件发送战情消息，同时接收另外 3 个组件发送过来的情报。

图 3.22　战情互报

扫一扫

自测题