第5章 组件应用

Vue 3 的组件系统是构建现代化前端应用的核心机制，通过将 UI 拆分为独立可复用的代码单元实现模块化开发。每个组件都是具有完整功能的 Vue 3 实例，支持与根组件相同的配置选项（如 data、methods、生命周期钩子等），同时具备独立的样式作用域和模板封装。这种设计允许开发者通过组合嵌套组件的方式构建复杂的组件树架构，既能实现功能模块的高度复用，又能保持项目的可维护性和扩展性。组件化开发模式使得大型应用可以被分解为多个职责单一的独立单元，通过清晰的接口（props/events）进行通信，既降低了系统内部耦合度，又提高了开发效率。几乎所有现代 Web 界面都可以通过这种组件树的形式进行抽象和实现，从简单的 UI 控件到完整的路由页面都能被封装为标准化组件，配合 Vue 3 的响应式系统和虚拟 DOM 机制，形成高效的可维护前端架构。


视频讲解

5.1 组件的基础概念

5.1.1 基本使用方法

以下是定义基本组件的示例代码：

```
// 定义一个名为 <button-counter> 的新组件
Vue.component('button-counter', {
  data: function() {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'
})
```

组件是可复用的 Vue 3 实例，开发者可以将这个组件作为自定义元素来使用，代码如下：

```
<div id="components-demo">
  <button-counter></button-counter>
</div>
```

组件可以接收选项，如 data、computed、watch、methods 以及钩子函数等，但 el 选项例外。

5.1.2 组件复用

在 Vue 3 组件中，data 选项必须声明为函数而非对象，这是组件复用机制的核心设计。当组件被复用时，每个实例都需要维护独立的数据状态，通过函数形式返回数据对象可确保每次组件实例化时都获得全新的数据副本。如果直接使用对象形式定义 data，所有组件实例将共享同一数据引用，导致状态污染。这种函数式声明方式保证了组件的封装性和独立性，是构建可复用组件的基础规范，示例代码如下：

```
data: function() {
  return {
    count: 0
  }
}
```

5.1.3 组织结构

通常情况下，应用程序会以一棵嵌套的组件树的形式进行组织，如图 5.1 所示。

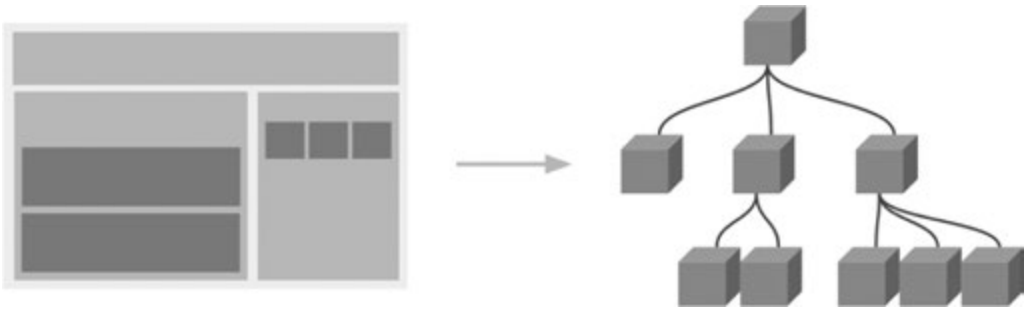

图 5.1 组件嵌套

需要将组件注册到 Vue 3 实例中才能使用，组件的注册方式分为全局注册和局部注册两种，全局注册组件使用 Vue 3 实例的 component()函数，该函数接收两个参数，第一个参数是组件的名称，第二个参数是组件的配置对象或组件的选项。注册的语法形式如下：

```
app.component({ string } name, { Function I Object } definition(optional))
```

下面是一个全局注册组件的例子代码：

```
const app = Vue.createApp({});
app.component('ButtonCounter', {
    data() {
    return {
        count: 0
    }
    },
    template:
     '< button @click = "count++">
        You clicked me {{ count }} times.
    </button >'
});

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

渲染结果如图 5.2 所示。

图 5.2 ButtonCounter 组件的渲染结果

组件的内容可以通过 template 选项定义，在使用组件时，组件所在位置会被 template 选项的内容所替换。组件注册完成后，可以将组件视为自定义元素，在需要的地方按照元素的方式使用，元素的名称就是注册时指定的组件名称，代码示例如下：

```
<div id = "app">
  <ButtonCounter></ButtonCounter>
</div>
```

上述代码并不能正常工作，因为 HTML 并不区分元素和属性的大小写，浏览器会把所有大写字符解释为小写字符，例如，会把<ButtonCounter>解释为<buttoncounter>，这就导致找不到组件而出现错误，解决办法是在 HTML 模板中采用 kebab-case 命名引用组件。命名代码如下：

```
<div id = "app">
  <button - counter></button - counter>
</div>
```

只要组件注册时采用的是 PascalCase(首字母大写)命名，就可以采用 kebab-case 命名来引用。在非 HTML 模板中可以使用组件的原始名称，即<ButtonCounter>和<button-counter>都是可以的。如果要保持名字的统一性，可以在注册组件时，直接使用 kebab-case 命名为组件命名，例如：

```
app.component('button - counter', …)
```

由于 HTML 不支持自闭合的自定义元素，在 HTML 模板中不能将<ButtonCounter>组件当作自闭合元素使用。例如，不能使用<button-counter/>，而应该使用<button-counter></button-counter>。在非 HTML 模板中不存在这个限制，相反还鼓励将没有内容的组件作为自闭合元素使用，这可以明确表示该组件没有内容，并且省略了结束标记，代码也更加简洁。

局部注册是在组件实例的选项对象中使用 component 选项注册，示例代码如下：

```
const MyComponent = {
    data() {
        return {
            count: 0
        }
    },
    template:
       '<button v-on:click="count++">
            You clicked me {{ count }} times.
        </button>'
}
const app = Vue.createApp({
    components: {
        ButtonCounter: MyComponent
    }
}).mount('#app');
```

对于 components 选项对象，每个属性的名称就是自定义元素的名称，其属性值就是组件实例。全局注册的组件可以在应用程序的任何组件实例的模板中使用，而局部注册的组件只能在父组件的模板中使用。

5.1.4 钩子函数

在 Vue 3 中针对钩子函数设计了新的函数，这些函数可以帮助开发者编写更好的代码。Vue 3 的 Composition API 提供了一个 setup()函数封装了大部分组件代码，并处理了响应式、钩子函数等，可以取代之前的 beforeCreate()函数和 Create()函数。钩子函数是必须导入项目中的，这是为了使项目尽可能轻量化。导入方式如下：

```
import { onMounted, onUpdated, onUnmounted } from 'vue'
```

旧的钩子函数可以在 setup()函数中访问，代码如下：

```
import {
  onBeforeMount,
  onMounted,
  onBeforeUpdate,
  onUpdated,
  onBeforeUnmount,
  onUnmounted,
  onActivated,
  onDeactivated,
  onErrorCaptured
} from 'vue'
export default {
  setup() {
    onBeforeMount(() => {
      // ...
    })
    onMounted(() => {
      // ...
    })
    onBeforeUpdate(() => {
      // ...
```

```
    })
    onUpdated(() => {
      // ...
    })
    onBeforeUnmount(() => {
      // ...
    })
    onUnmounted(() => {
      // ...
    })
    onActivated(() => {
      // ...
    })
    onDeactivated(() => {
      // ...
    })
    onErrorCaptured(() => {
      // ...
    })
  }}
```

5.2 组件间的数据传递


视频讲解

5.2.1 通过 props 属性向子组件传递数据

使用组件时可以为组件元素设置属性。首先需要在组件内部注册一些自定义属性，称为 prop，这些 prop 是在组件的 props 选项中定义的，然后就可以将这些 prop 名称作为元素的属性名来使用，通过属性向子组件传递数据。

```
Vue. component(blog - post, {
    props: ['title '],
    template: '< h3 > {{ title }} /h3 > ',
})
```

一个组件默认可以拥有任意数量的 prop，任何值都可以传递给 prop。在上述模板中能够在组件实例中访问 title 这个值。一个 prop 被注册之后，就可以像这样把数据作为一个自定义属性传递进来。

下面通过一个“传统文化与生态保护”的例子说明如何通过 props 属性传递参数。首先，在子组件 SeasonCard. vue 中定义属性 seasonData，在子组件模板中定义要显示的节气内容结构，代码如下：

```
<!-- 子组件: SeasonCard. vue -->
< script setup >
const props = defineProps({
  seasonData: {
    type: Object,
    required: true,
    validator: (value) =>
      'name' in value &&
      'climate' in value &&
      'custom' in value &&
```

```
        'protection' in value
    }
});
</script>
<template>
  <div class="season-card">
    <h3>{{ seasonData.name }}</h3>
    <div class="card-content">
      <p><strong>气候特征：</strong>{{ seasonData.climate }}</p>
      <p><strong>传统习俗：</strong>{{ seasonData.custom }}</p>
      <p><strong>生态保护：</strong>{{ seasonData.protection }}</p>
    </div>
    <div class="thinking-question">
      <h4>思政思考：</h4>
      <p>结合{{ seasonData.name }}的传统智慧，你认为现代人应该如何更好地践行生态保护理念？</p>
    </div>
  </div>
</template>
```

在父组件 SeasonalEducation.vue 中定义一组节气数据，并给子组件的 season-data 属性传递参数，在父组件中引入子组件显示节气内容，代码如下：

```
<!-- 父组件：SeasonalEducation.vue -->
<script setup>
import { ref } from 'vue';
import SeasonCard from './SeasonCard.vue'; // 子组件
// 节气数据
const seasons = ref([
  {
    name: '立春',
    climate: '开始回暖',
    custom: '咬春(吃春饼)',
    protection: '植树造林最佳时期'
  },
  {
    name: '清明',
    climate: '气温转暖',
    custom: '扫墓祭祖',
    protection: '森林防火关键期'
  },
  {
    name: '芒种',
    climate: '高温多雨',
    custom: '煮梅送花神',
    protection: '防汛抗旱准备期'
  }
]);

// 当前选中的节气索引
const currentSeasonIndex = ref(0);
// 切换节气方法
const switchSeason = (index) => {
  currentSeasonIndex.value = index;
};
```

```
</script>
<template>
  <div class="container">
    <h2>传统文化与生态保护</h2>
    <p>通过二十四节气学习传统智慧,践行现代生态保护理念</p>
    <!-- 节气导航 -->
    <div class="nav-buttons">
      <button
        v-for="(season, index) in seasons"
        :key="index"
        @click="switchSeason(index)"
        :class="{ active: currentSeasonIndex === index }"
      >
        {{ season.name }}
      </button>
    </div>
    <!-- 子组件展示 -->
    <SeasonCard
      :season-data="seasons[currentSeasonIndex]"
    />
  </div>
</template>
```

5.2.2 跨层级组件传递数据

Provide 和 Inject 是 Vue 3 提供的一种跨层级组件通信方式,允许祖先组件向其所有子孙后代组件注入依赖,而不必通过每一层组件传递 props。使用 provide 祖先组件提供数据,使用 inject 后代组件注入数据。

示例代码如下:

```
<!-- 祖先组件 -->
<script setup>
import { provide, ref } from 'vue';
const data = ref('Shared data');
provide('sharedData', data);
</script>
<!-- 后代组件 -->
<script setup>
import { inject } from 'vue';
const sharedData = inject('sharedData');
</script>
```

以下是一个通过跨层级组件传递数据更改主题的例子。

```
<div class="code-block">
// 祖先组件
export default {
  data() {
    return {
      theme: 'dark',
      user: { name: '张三' }
    }
  },
  provide() {
```

```
    return {
      theme: this.theme,
      user: this.user,
      changeTheme: this.changeTheme
    }
  },
  methods: {
    changeTheme(newTheme) {
      this.theme = newTheme
    }
  }
}
// 后代组件
export default {
  inject: ['theme', 'user', 'changeTheme'],
  methods: {
    handleClick() {
      this.changeTheme('light')
    }
  }
}
          </div>
```

5.2.3 通过监听事件向父组件传递数据

前面介绍了父组件可以通过 prop 向子组件传递数据，反过来，子组件的某些功能需要与父组件进行通信，则可以通过自定义事件实现。子组件使用$ emit()函数触发事件，父组件使用 v-on 指令监听子组件的自定义事件，$ emit()函数的语法形式如下：

```
$emit(eventName,[... args])
```

eventName 为事件名，args 为附加参数，这些参数会传给事件监听器的回调函数。子组件通过第二个参数向父组件传递数据。例如以下子组件代码：

```
app.component('child', {
  data: function() {
    return {
      name: '张三'
    }
  },
  methods: {
    handleClick() {
      // 调用实例的$emit()函数触发自定义事件 greet,并传递参数
      this.$emit('greet', this.name);
    }
  },
  template: '<button @click="handleClick">开始欢迎 </button>'
})
```

子组件中的按钮接收到 click 事件后，使用$ emit()函数触发一个自定义事件，使用组件时可以使用 v-on 指令监听 greet 事件，实现子组件向父组件传递数据，代码如下：

```
<div id = "app">
  <child @greet = "sayHello"></child>
</div>
const app = Vue.createApp({
  methods: {
    // 自定义事件的附加参数会自动传入函数
    sayHello(name) {
      alert("Hello," + name);
    }
  }
});
```

与组件和 prop 不同,事件名不会自动转换大小写。调用$ emit()函数触发的事件名称需要与用于监听该事件的名称完全匹配。如果在 v-on 指令中直接使用 JavaScript 语句,则可以通过$ emit()函数访问自定义事件的附加参数,示例代码如下:

```
<button @click = "$emit('enlarge - text',0.1)">
  Enlarge text
</button>
```

5.3 插槽


视频讲解

在 Vue 组件化开发过程中,我们经常面临一个核心设计挑战:如何在保持组件结构稳定性的同时,为特定使用场景提供足够的定制能力。传统方案通过 props 传递复杂数据结构往往导致代码冗余和维护困难。

Vue 插槽(Slot)机制为此提供了优雅的解决方案。通过在组件模板中声明一个或多个内容占位区,允许父组件将自定义模板片段注入这些预定位置。这种设计模式巧妙平衡了组件的内聚性与扩展性,使组件在维持核心功能稳定的同时,具备了高度的可定制能力。

5.3.1 基本插槽

基本插槽(默认插槽)是 Vue 内容分发的基础形式,它允许父组件向子组件传递任意模板内容,这些内容会替换子组件中的< slot ></slot >占位符。

基本插槽使用步骤如下。

(1) 在子组件中定义插槽占位符< slot ></slot >。

```
<!-- ChildComponent.vue -->
<div class = "card">
  <div class = "card - header">
    <h3 >{{ title }}</h3 >
  </div >
  <!-- 插槽占位符 -->
  <div class = "card - body">
    <slot ></slot > <!-- 内容插入点 -->
  </div >
  <div class = "card - footer">
    页脚信息
  </div >
</div >
```

（2）在父组件中向子组件传递内容，在< ChildComponent >标签中的内容会被替换。

```
<!-- ParentComponent.vue -->
<ChildComponent title="用户资料">
  <!-- 插入内容 -->
  <img src="avatar.jpg" alt="头像">
  <p>用户名：{{ userName }}</p>
  <button @click="editProfile">编辑资料</button>
</ChildComponent>
```

（3）渲染结果。

```
<div class="card">
  <div class="card-header">
    <h3>用户资料</h3>
  </div>
  <div class="card-body">
    <img src="avatar.jpg" alt="头像">
    <p>用户名：张三</p>
    <button>编辑资料</button>
  </div>
  <div class="card-footer">
    页脚信息
  </div>
</div>
```

当父组件未提供内容时，插槽可显示预设的默认内容：

```
<!-- 子组件 -->
<slot>
  <!-- 默认内容 -->
  <div class="empty-state">
    <p>暂无内容</p>
    <button>刷新</button>
  </div>
</slot>
```

5.3.2 编译作用域

在插槽中使用数据的模板代码如下：

```
<navigation-link url="/profile">
  Logged in as {{ user.name }}
</navigation-link>
```

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

```
<navigation-link url="/profile">
  Clicking here will send you to: {{ url }}
  // 这里的 'url' 会是 undefined
</navigation-link>
```

5.3.3 后备内容

可以给一个插槽设置默认内容，这个内容只会在没有提供内容时被渲染，例如

<button>组件大部分时候渲染文本 Submit 作为默认内容，示例代码如下：

```
<button type = "submit">
  <slot> Submit </slot>
</button>
```

当在一个父组件中使用<submit-button>并且没有提供任何插槽内容时，将会渲染默认内容 Submit，如果提供了内容，则会渲染提供的内容，代码如下：

```
<submit - button></submit - button>
<button type = "submit">
  Submit
</button>
<submit - button>
  Save
</submit - button>
<button type = "submit">
  Save
</button>
```

5.3.4 具名插槽

有时在项目中需要使用多个插槽，<slot>元素有一个特殊的属性 name 可以用来定义额外的插槽，没有 name 属性的<slot>元素会有一个隐含的名字 default，代码如下：

```
<div class = "container">
  <header>
    <slot name = "header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name = "footer"></slot>
  </footer>
</div>
```

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

```
<base - layout>
  <template v - slot:header>
    <h1> Here might be a page title </h1>
  </template>
  <p> A paragraph for the main content. </p>
  <p> And another one. </p>
  <template v - slot:footer>
    <p> Here's some contact info </p>
  </template>
</base - layout>
```

<template>元素中的所有内容都将传递到相应的插槽中，任何没有被包裹在带有 v-slot 的<template>中的内容都将被视为默认插槽的内容。如果希望更明确则可在一个<template>中包括默认插槽的内容，注意 v-slot 只能添加在<template>上，代码如下：

```
<base-layout>
  <template v-slot:header>
    <h1>Here might be a page title</h1>
  </template>
  <template v-slot:default>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </template>
  <template v-slot:footer>
    <p>Here's some contact info</p>
  </template>
</base-layout>
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

5.3.5 作用域插槽

有时候让插槽内容能够访问子组件中的数据是非常有用的，例如<current-user>组件想要更改默认内容为 user.firstName，代码如下：

```
<span>
  <slot>{{ user.lastName }}</slot>
</span>
<current-user>
  {{ user.firstName }}
</current-user>
```

上述代码不能正常工作，因为只有<current-user>组件可以访问到 user，而代码中提供的内容是在父级渲染的。为了让 user 在父级的插槽内容中可用，可以将 user 作为<slot>元素的一个属性绑定上去，代码如下：

```
<span>
  <slot v-bind:user="user">
    {{ user.lastName }}
  </slot>
</span>
```

绑定在<slot>元素上的属性被称为插槽 prop，在父级作用域中可以使用带值的 v-slot 来定义插槽 prop 的名字，将包含所有插槽 prop 的对象命名为 slotProps，也可任意命名，代码如下：

```
<current-user>
  <template v-slot:default="slotProps">
    {{ slotProps.user.firstName }}
  </template>
</current-user>
```

5.3.6 动态插槽名

动态指令参数也可以用在 v-slot 上，来定义动态的插槽名，代码如下：

```
<base-layout>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>
</base-layout>
```

5.4 其他应用

5.4.1 动态组件

动态组件是 Vue 3 中实现组件按需动态渲染的核心机制，通过<component>标签的 is 属性绑定变量，在运行时灵活切换组件。其核心价值体现在以下方面。

（1）统一渲染入口：通过单一挂载点管理多个组件的切换。

（2）状态管理优化：结合<KeepAlive>实现组件状态缓存。

（3）代码复用提升：避免为相似功能创建多个独立挂载节点。

（4）性能优化：减少不必要的组件销毁与重建。

下面是一个完整的示例，展示如何使用动态组件，并包含一个简单的选项卡切换界面，代码如下：

```
<template>
  <!-- 动态组件核心结构 -->
  <component :is="currentComponent" />
</template>
<script setup>
import { ref } from 'vue'
import ComponentA from './ComponentA.vue'
import ComponentB from './ComponentB.vue'
// 定义当前组件引用
const currentComponent = ref(ComponentA)
</script>
// 切换逻辑实现
const switchComponent = () => {
  currentComponent.value =
    currentComponent.value === ComponentA
      ? ComponentB
      : ComponentA
}
```

程序具体执行流程为：修改 currentComponent 值→触发组件卸载→创建新组件实例→插入 DOM。

5.4.2 异步组件

异步组件是 Vue 3 中优化应用性能的关键技术，通过按需加载组件代码，显著减少首屏加载时间。其核心价值体现在以下方面。

（1）代码分割：将大型应用拆分为多个小块，实现懒加载。

（2）性能优化：减少初始包体积，加速页面渲染。

（3）按需加载：仅在需要时加载组件，节省带宽资源。

（4）错误边界：支持加载状态和错误处理，提升用户体验。

定义异步组件，示例代码如下：

```
// 方式 1：使用 defineAsyncComponent
import { defineAsyncComponent } from 'vue'
const AsyncComponent = defineAsyncComponent(() =>
  import('./components/HeavyComponent.vue')
)
// 方式 2：直接返回 Promise(Vue 3.2+)
const AsyncComponent = () => import('./HeavyComponent.vue')
```

基本使用方式的代码如下：

```
<template>
  <AsyncComponent />
</template>
<script setup>
import AsyncComponent from './AsyncComponent'
</script>
```

下面是一个基本案例，实现一个用户资料编辑器的异步加载场景。当用户单击按钮时，动态加载包含复杂表单的编辑器组件，展示异步组件的加载过程控制与状态管理。示例代码如下：

```
// 父组件(ParentComponent.vue)
<template>
  <div>
    <h1>用户资料编辑器</h1>
    <button @click="showEditor = true">打开编辑器</button>
    <!-- 异步组件容器 -->
    <Suspense>
      <template #default>
        <AsyncUserEditor v-if="showEditor" />
      </template>
      <template #fallback>
        <div class="loading">加载中...</div>
      </template>
    </Suspense>
  </div>
</template>
<script setup>
import { ref } from 'vue'
// 定义异步组件(实际开发中使用真实路径)
const AsyncUserEditor = defineAsyncComponent(() =>
  import('./UserEditor.vue')
```

```
)
const showEditor = ref(false)
</script>
<style>
.loading {
  padding: 20px;
  background: #f0f0f0;
  border-radius: 4px;
}
</style>
// 异步子组件(UserEditor.vue)
<template>
  <div class="editor">
    <h2>编辑用户资料</h2>
    <form @submit.prevent="save">
      <div class="form-group">
        <label>用户名</label>
        <input v-model="user.name" required>
      </div>
      <div class="form-group">
        <label>邮箱</label>
        <input v-model="user.email" type="email" required>
      </div>
      <button type="submit">保存</button>
    </form>
  </div>
</template>
<script setup>
import { ref } from 'vue'
const user = ref({
  name: '',
  email: ''
})
// 模拟保存操作
const save = () => {
  console.log('保存用户数据:', user.value)
  alert('保存成功!')
}
</script>
<style>
.editor {
  max-width: 500px;
  margin-top: 20px;
  padding: 20px;
  border: 1px solid #eee;
}
.form-group {
  margin-bottom: 15px;
}
label {
  display: block;
  margin-bottom: 5px;
  font-weight: 500;
}
</style>
```

5.5 案例

5.5.1 实现日历组件

接下来将实现一个日历组件，以下是核心代码：

```
<template>
  <div id="calender">
    <div class="txt-c p-10">
      <span @click="prev">⬅ </span>
      <input type="text" v-model="year">年
      <input type="text" v-model="month" class="month">月
      <span @click="next">➡ </span>
    </div>
    <div class="weekDay flex jc-sb p-5 day">
      <p v-for="item in weekList" :key="item.id">{{ item }}</p>
    </div>
    <div class="weekDay flex p-5 day">
      <p v-for="item in spaceDay" :key="item.id"></p>
      <p v-for="(item, idx) in (monthDay[this.month - 1] || 30)" @click="setDay(idx)"
:class="idx == activeDay ? 'active' : ''"
        :key="item.id">{{ item }}</p>
    </div>
  </div>
</template>
<script>
export default {
  name: "calender",
  data() {
    return {
      year: '',                // 年份
      month: '',               // 月份
      day: '',                 // 天数
      current: '',             // 当前时间
      weekList: ['周一', '周二', '周三', '周四', '周五', '周六', '周日'],
      monthDay: [31, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
      activeDay: '',           // 选中的日期
      spaceDay: '',            // 每个月第一天是星期几
      February: ''             // 判断2月份的天数
    }
  },
  created() {
    this.current = new Date()
    this.getTheCurrentDate()
    this.getMonthFisrtDay()
    this.February = this.isLeapYear(this.year) ? 28 : 29
    this.monthDay.splice(1, 1, this.February)
  },
  watch: {
    month() {
      if (this.month > 12 || this.month < 1) {
        console.log('请输入正确月份')
        return
```

```
        }
        this.getMonthFisrtDay()
      },
      year() {
        this.getMonthFisrtDay()
      }
    },
    methods: {
      // 判断是不是闰年
      isLeapYear(year) {
        return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
      },
      // 选取特定天数
      setDay(idx) {
        this.activeDay = idx
        this.day = idx + 1
        console.log('选择的日期是' + this.year + '' + this.month + '' + this.day)
      },
      // 判断月份的第一天是星期几
      getMonthFisrtDay() {
        var firstDayOfCurrentMonth = new Date(this.year, this.month - 1, 1) // 某年某月的第一天
        if (firstDayOfCurrentMonth.getDay() == 0) {
          this.spaceDay = 6
        } else {
          this.spaceDay = firstDayOfCurrentMonth.getDay() - 1
        }
      },
      // 获取当前的日期
      getTheCurrentDate() {
        this.year = this.current.getFullYear()
        this.month = this.current.getMonth() + 1
        this.day = this.current.getDate()
      },
      prev() {
        if (this.month == 1) {
          this.year--
          this.month = 12
        } else {
          this.month--
        }
        this.activeDay = 0
        this.getMonthFisrtDay()
      },
      next() {
        if (this.month == 12) {
          this.year++
          this.month = 1
        } else {
          this.month++
        }
        this.activeDay = 0
        this.getMonthFisrtDay()
      }
    }
  }
</script>
```

5.5.2 利用组件实现“弹出层”

弹窗类组件的特点是独立于当前 Vue 3 实例之外，不能被写在当前业务的 HTML 模板内，否则样式控制会比较困难，通用性也会变差。通常情况下弹窗类组件会被挂载在 body 上，通过对 Vue 3 实例的创建和挂载，使用 JavaScript 动态地生成和取消，具体实现思路如下。

（1）创建一个 create()函数。

（2）传入一个组件的配置，包括组件本身和 props 参数。

（3）创建组件实例，并将其挂载到 body 上。

（4）最后返回组件实例。

具体代码如下：

```
import Vue from 'vue'
// Component 为传入的组件，props 为传入的组件参数
export default function create(Component, props) {
  // 创建实例
  const vm = new Vue({
    render(h) {
      return h(Component, { props })
    }
  }).$mount()
  // 通过 vm.$el 获取生成的 dom 对象，把生成的真实 dom 对象插入 body 中
  document.body.appendChild(vm.$el)
    // 获取根组件实例
  const comp = vm.$children[0]
  // 弹窗关闭时调用
  comp.remove = () => {
    // 移除本身
    document.body.removeChild(vm.$el)
    // 释放自己所占资源
    vm.$destroy()
  }
  // 返回创建的实例
  return comp
}
```

传入要显示的组件，即上文中提到的 Component 参数，代码如下：

```
<template>
  <div v-if="isShow">
    <h3>{{ title }}</h3>
    <p>{{ message }}</p>
  </div>
</template>
<script>
export default {
  // 上文中传入的 props 参数
  props: {
    title: {
      type: String,
      default: ""
    },
    message: {
      type: String,
```

```
        default: ""
      },
      duration: {
        type: Number,
        default: 1000
      }
    },
    data() {
      return {
        isShow: false
      };
    },
    methods: {
      // 调用 show()函数展示
      show() {
        this.isShow = true;
        // 使用 setTimeout 定时关闭
        setTimeout(this.hide, this.duration);
      },
      // 消失后移除自身所占资源
      hide() {
        this.isShow = false;
        this.remove();
      }
    }
  };
  </script>
```

create()函数返回的是组件自身实例,调用 show()函数即可成功显示弹窗组件,代码如下:

```
create(Notice, {
  title: '我是自定义弹窗组件',
  message: '我成功出现了',
  duration: 3000
}).show()
```

5.6 总结

本章完整地介绍了 Vue 3 中组件开发所涉及的各个知识点,在学习本章时需要保持耐心,仔细阅读章节内容,并尝试动手编写一些实例,以便更好地理解本章的内容。

习题 5


自测题

1. 为什么 data 是一个函数?
2. Vue 3 组件通信有哪些方式?
3. Vue3 中如何通过动态组件实现组件切换?
4. 请描述 Vue 3 的父子组件钩子函数的执行顺序。