第 3 章 Vue 2.x 组件开发

3.1 Vue 组件定义方案

组件定义是 Vue 框架中最具特点的开发方案，也是 MVVM 框架中的经典实现。当使用 Vue 框架进行项目开发时，Vue 允许开发者在项目中定义非 HTML 自有标签节点。并为其赋予相应的功能，用来将传统的 HTML 页面打散，按照页面中的不同部分和功能进行拆分，实现页面的功能细化和更友好的结构化。在学习 Vue 的组件开发前首先要了解什么是单页面应用。

1. 什么是单页面应用

单页面应用也称为 SPA(Single Page Application)。单页面应用与传统的 HTML 网页应用有明显的不同，在一个单页面应用项目中，有且只有一个 HTML 物理页面文件，其他的页面都以逻辑划分，通过 JavaScript 动态渲染到同一个 HTML 文件内。

单页面应用致力于更友好地切换页面及更快速地渲染视图。在单页面应用中，可以将页面拆分成一个或多个自定义组件，当页面的某个组件内部的视图更新时，只有被更新的部分会被重新渲染一次，其他页面部分保持原有的状态，所以单页面应用在进行页面跳转时并不会触发传统 HTML 重新加载静态资源的过程。由于单页面应用的一切页面都是通过 JavaScript 进行切换和渲染的，所以单页面应用在页面跳转时，可以结合 CSS 动画实现跳转页面的交互动画。

2. 单页面应用与传统 Web 应用的对比

单页面应用与传统 Web 应用相比，优点如下：

(1) 由于单页面应用只有一个 HTML 物理页面文件，所以一个单页面应用的整个项目占用空间和文件数量要比传统的 Web 应用小。

(2) 单页面应用通过结合组件化的开发方式，可以更好地对项目进行工程化管理，提升了项目的可维护性和持续迭代性。

(3) 由于单页面应用通过 JavaScript 渲染页面，所以在页面跳转时，可以更快速地加载新页面，无须触发网络请求与 HTML 文件的重新加载。

（4）单页面应用可以在页面跳转的过程中插入动态的过渡动画，提升项目的友好度和交互效果。

单页面应用与传统 Web 应用相比，缺点如下：

（1）单页面应用的核心渲染能力由 JavaScript 提供，如果开发者对依赖包的规划没有到位，在项目的 JavaScript 核心文件未下载完成前，用户只能看见白屏，这种现象叫作白屏时间。白屏时间极大地降低了项目的首屏加载速度和交互体验。

（2）单页面应用在浏览器的兼容性上不如传统的 Web 应用，由于单页面应用的数据响应式系统及其他的新式开发模式，大多会依赖现代浏览器才支持的核心 API，所以单页面应用不支持较早的浏览器版本。

（3）单页面应用对开发者的编程水平和设计能力有较高的要求，如果开发者没有较强的前端开发能力，则其构建的单页面应用会产生更多的问题并且更加难以维护。

3.1.1　自定义组件介绍

Vue 框架的组件系统允许开发者通过 Vue 代码创建一个 HTML 标签系统中原本不存在的标签对象。开发者不仅可以定义标签的名称，还可以为标签设计它特有的样式和功能。这样便可以将原本需要大量 HTML 基础标签堆叠起来的页面，根据结构划分成多个区域，针对区域的结构和功能单独创建不同的标签节点，分散在不同的组件代码中进行管理，最终使一个页面的源代码可以通过几个简单的基础标签实现，具体的页面划分方式如图 3-1 所示。

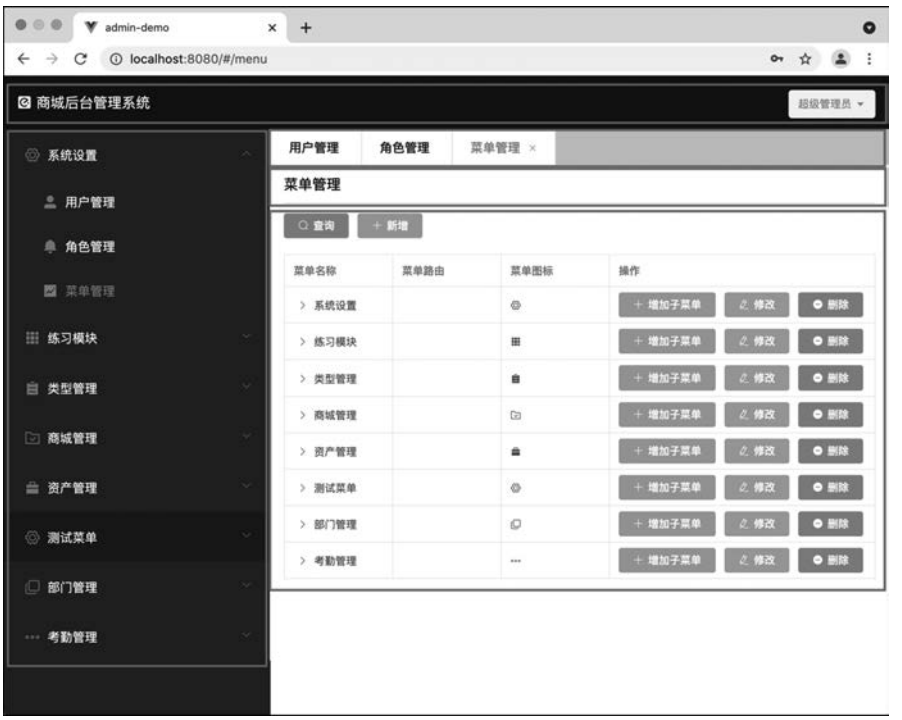


图 3-1　页面组件划分效果图

使用合理的页面划分思想，结合 Vue 的组件系统，可以将页面的基础标签分成几个高级组件来构建，不同的组件内部编写的代码只服务于组件本身，这个操作使页面代码也能变成可插拔式代码。Vue 的自定义组件系统可以理解为，在组装一台个人计算机时，设备厂商为了让不懂硬件生产和开发的普通人，能针对自己的需求独立拼装计算机，将一台计算机合理地分成内存、显卡、硬盘、主板、电源等大的模块单元，这样售卖计算机的卖家及购买计算机的用户都不需要去关注电路板上的精细电子元件，只需合理地将几个大部件组合便能快速地组装一台计算机。Vue 的自定义组件利用了相同的思想，当页面按照大的区域划分后，开发者如果想改动页面的某个部分，则只需修改指定区域对应的组件代码或将这部分直接替换成一个新的组件，便可以完成项目的重构，整个过程无须关注与修改部分无关的代码和组件，所以利用 Vue 的自定义组件方式，可以将图 3-1 的页面划分，构建成极其简单易读的样子，代码如下：

```
<!-- 第 3 章 自定义组件的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <!-- 头部组件 -->
      <p-header></p-header>
      <!-- 空间划分组件 -->
      <p-container>
        <!-- 菜单组件 -->
        <p-menu>
          菜单部分……
        </p-menu>
        <!-- 主体部分 -->
        <p-main>
          <!-- 标签页 -->
          <p-tabs></p-tabs>
          <!-- 页面主题 -->
          <p-title></p-title>
          <!-- 表格部分 -->
          <p-table></p-table>
        </p-main>
      </p-header>
    </div>
  </body>
</html>
```

自定义组件在 Vue 2.x 中的定义方式，代码如下：

```
Vue.component('组件名',{
    组件功能
})
```

自定义组件使用 Vue.component()进行创建，组件名不可以使用 HTML 中的自有标签(包括 SVG 系统中的所有标签)。自定义组件的命名可以使用以下两种方式：

(1) 全小写并使用“-”进行分词的方式。例如：当组件命名为 mycomponent 时，由于 my 和 component 为两个单词，所以最终命名应为 my-component。

(2) 首字母大写的驼峰形式。如：当组件命名为 mycomponent 时，由于 my 和 component 为两个单词，所以最终命名应为 MyComponent。

自定义组件的名称应见名知意，通常的命名方式为“作者代号-组件分类-组件功能……”不推荐单词过长或分词过多。

Vue.component()的第 2 个参数为组件的功能描述对象，该对象与 new Vue()时直接传入的对象结构基本一致，可以直接使用 Vue 框架的 data、computed、watch、methods 等选项及完整的生命周期。接下来以自定义按钮为例，学习如何创建第 1 个 Vue 的自定义组件，代码如下：

```
<!-- 第 3 章 自定义按钮代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <!-- Vue 渲染内容依赖的 HTML 容器 -->
    <div id="app">
      <h4>
        自定义按钮
      </h4>
      <p-button></p-button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-button',{
        template:`
          <button>自定义按钮</button>
        `
      })
      new Vue({}).$mount('#app')
    </script>
  </body>
</html>
```

运行案例的效果如图 3-2 所示。

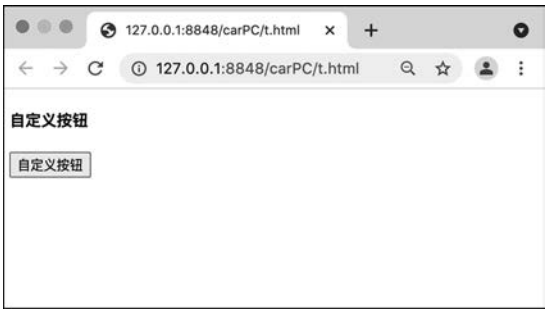


图 3-2　自定义按钮的效果图

自定义组件中可以使用 template 属性来描绘自定义组件的视图部分，template 接收一个字符串类型的值，该值支持 HTML 基本标签语法和 Vue 中的所有指令。Vue.component()必须在 new Vue()前进行创建，否则自定义组件会无法绑定在 Vue 实例上，导致在页面中应用自定义组件失败。在 Vue 2.x 自定义组件的 template 属性中还规定，在其内部定义的 HTML 标签有且只有一个根节点，不可以在 template 中同时创建并列的兄弟节点，否则会出现错误，代码如下：

```
<!-- 第 3 章 自定义组件错误代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <!-- Vue 渲染内容依赖的 HTML 容器 -->
    <div id="app">
      <h4>
        自定义按钮
      </h4>
      <p-button></p-button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-button',{
        template:`
          <button>自定义按钮</button>
          <button>我也是自定义按钮</button>
        `
      })
      new Vue({}).$mount('#app')
```

```
    </script>
  </body>
</html>
```

运行案例，结果如图 3-3 所示。

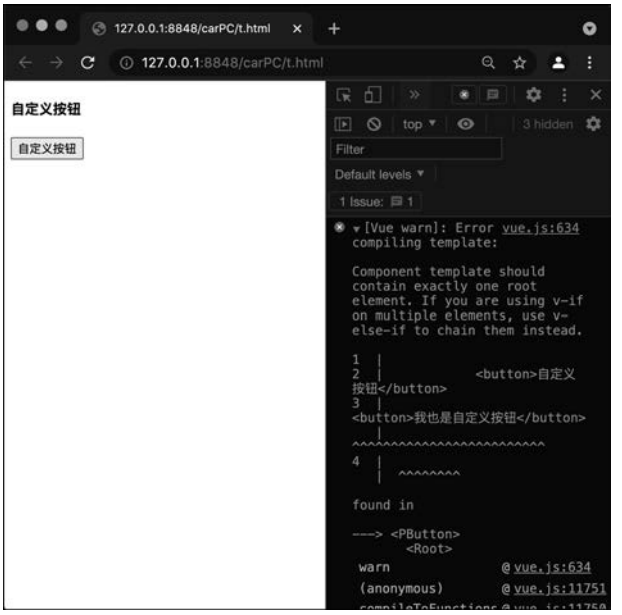


图 3-3　自定义组件错误案例的效果图

控制台上会出现错误：Component template should contain exactly one root element. If you are using v-if on multiple elements, use v-else-if to chain them instead。该错误明确指出：自定义组件应该有且只有一个根节点，如果开发者想要使用多个兄弟节点，则必须结合 v-if、v-else-if 等指令来保证通过条件筛选，最终也只有一个节点作为根节点。

3.1.2　组件的属性介绍

虽然自定义组件是 HTML 中并不存在的标签节点，但自定义组件完全遵守 HTML 标签的所有规则。通过学习 3.1.1 节，创建一个自定义组件已经不存在问题，不过在创建组件时会发现，如果将代码直接写死在 template 属性内（如自定义按钮的案例），则无论组件被引用多少次，在页面上呈现的结果都是固定的，这显然并不适合真实的开发场景。

针对以上问题，Vue 在组件系统中提供了 props 选项，来支持开发者对创建的组件定义组件需要的属性及属性应具备的功能。在 props 中声明的属性会自动应用于自定义组件的标签上，开发者可以像使用 HTML 标签自有属性一样对其设置结果，自定义组件会针对属性不同的结果展示不同的形态。自定义组件的 props 属性的应用方式，代码如下：

```
<!-- 第 3 章 props 属性应用方式代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>props 应用方式</title>
  </head>
  <body>
    <!-- Vue 渲染内容依赖的 HTML 容器 -->
    <div id="app">
      <h4>
        props 应用方式
      </h4>
      <p-button></p-button>
      <p-button label="我是按钮的名字"></p-button>
      <p-button :label="label"></p-button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-button',{
        props:{
          label:{                          //为组件定义自定义属性 props
            required:true,                 //label 属性是否必填
            type:String,                   //label 属性支持的数据类型
            default(){                     //label 为空时的默认值
              return '默认值'
            }
          }
        },
        //在 props 中定义的属性会自动挂载到组件实例中,可以直接在 template 中使用
        template:`
          <button>{{label}}</button>
        `
      })
      new Vue({
        data(){
          return {
            label:'我是动态绑定的内容'
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

运行案例代码,结果如图 3-4 所示。

在 props 中定义的对象包含三个常用属性。

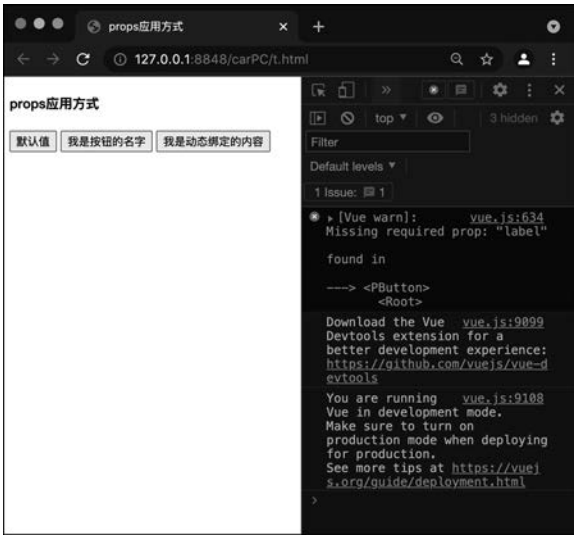


图 3-4　props 属性应用方式的效果图

1）required

required 代表该属性是否必填。它接收一个布尔类型的值，默认值为 false，当值为 true 时，若属性在组件中没有被使用，则控制台会抛出警告信息，告知开发者有一个必填的属性没有使用，警告信息抛出并不会中断程序运行，所以当设置了默认值时，组件依然会按照默认值展示。

2）type

type 代表自定义属性可接受的类型。可以直接对其设置 JavaScript 中的类型包装类，若希望属性的类型只接收字符串型，则将 type 的值设置为 String 即可。只有使用 v-bind 传入的属性，才能保持属性的数据类型，若希望自定义属性接收更多类型，则可以使用数组的方式定义 type，代码如下：

```
//第 3 章 自定义组件 props 中的 type 代码案例
props:{
  label:{                                    //为组件定义自定义属性 props
    required:true,                           //label 属性是否必填
    type:[String,Number,Boolean],            //label 属性支持的数据类型
    default(){                               //label 为空时的默认值
      return '默认值'
    }
  }
}
```

3）default

default 代表自定义属性的默认值。当该属性未在自定义组件中使用时自动称为属性

的默认值，default 属性可直接设置为默认值的结果，也可使用函数的方式将默认值返回去。当自定义属性的类型为数组或对象时，必须使用函数的方式返回默认值，否则当有多个组件副本时，default 会引用同一块内存地址而造成数据隔离失败。

在 props 中定义的属性会自动挂载到组件的实例对象上，可以直接在 template 中使用插值表达式引用，也可以在组件内部的函数中通过 this 进行引用。在 props 中定义的属性是响应式的，若其绑定的值发生变化，则组件会感知到变化并触发视图的更新。props 的特性与 data 选项中定义的数据类似。不同于 data 选项，props 中声明的自定义属性为只读属性，Vue 不推荐在组件内部通过 JavaScript 改变 props 中定义的属性的值，props 中属性的综合使用案例，代码如下：

```
<!-- 第 3 章 props 中属性的综合使用案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>props 综合应用案例</title>
  </head>
  <body>
    <!-- Vue 渲染内容依赖的 HTML 容器 -->
    <div id="app">
      <h4>
        props 综合应用案例
      </h4>
      <p-button :label="label"></p-button>
      <br>
      <!-- 在文本框中改动 -->
      <input type="text" v-model="label"/>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-button',{
        props:{
          label:{                                    //为组件定义自定义属性 props
            required:true,                           //label 属性是否必填
            type:[String,Number,Boolean],            //label 属性支持的数据类型
            default(){                               //label 为空时的默认值
              return '默认值'
            }
          }
        },
        created(){
          this.label='这是一个不推荐的操作'
        },
        updated(){
          console.log('组件发生了变更')
```

```
        },
        //在 props 中定义的属性会自动挂载到组件实例中,可以直接在 template 中使用
        template:`
          <button>{{label}}</button>
        `
      })
      new Vue({
        data(){
          return {
            label:'我是动态绑定的内容'
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

案例运行结果如图 3-5 所示。

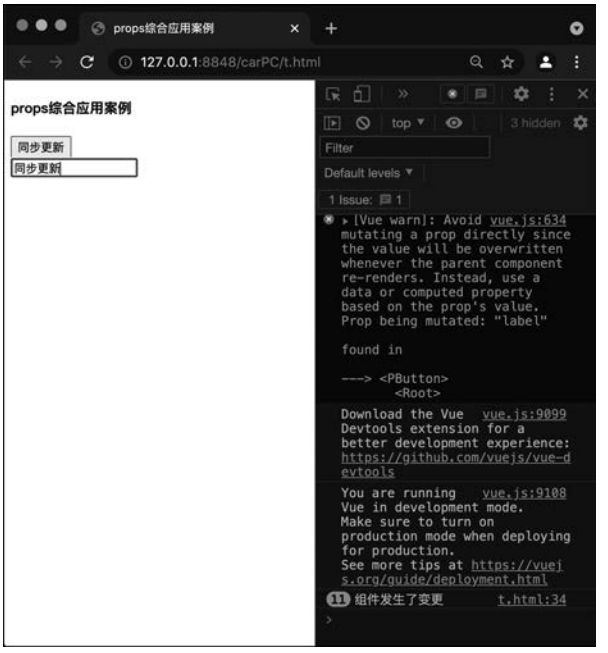


图 3-5　props 中属性的综合使用案例的效果图

在运行结果中会发现控制台抛出如下警告：Avoid vue.js:634 mutating a prop directly since the value will be overwritten whenever the parent component re-renders. Instead, use a data or computed property based on the prop's value。该警告提示开发者，不推荐开发者在组件内部主动修改 props 中创建的自定义属性的值，因为属性的值由组件外绑定并

传入，当自定义属性绑定的值发生变更时，组件外部对属性设置的新值会在组件重新渲染时覆盖组件内部的更改。这个规则在 Vue 3.x 版本中被定义得更加严格，Vue 3.x 中严格禁止了开发者在组件内部修改 props 中定义的属性的值。

3.1.3 组件的事件绑定介绍

自定义组件除自定义属性外还支持自定义事件系统。自定义事件系统可直接应用 Vue 内置的事件绑定方式。与原生 HTML 标签不同的是，自定义组件在实际渲染到网页时会被解析为其内部 template 中设置的内容，所以在自定义组件上绑定的事件默认为无法被挂载到其内部的 HTML 标签上，代码如下：

```
<!-- 第 3 章 自定义组件的事件绑定案例 1 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-button label="自定义按钮" @click="handleClick"></p-button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script type="text/javascript">
      Vue.component('p-button',{
        props:{
          label:{
            type:String,
          }
        },
        template:`
          <button>{{label}}</button>
        `
      })
      new Vue({
        methods:{
          handleClick(){
            console.log('自定义事件')
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

运行案例中的代码，会发现页面中并不会报错并且单击事件并没有成功地绑定在按钮上。发生这种现象主要因为，虽然自定义组件在源代码中当作真实存在的 HTML 标签使

用，但是实际上视图渲染的内容是 template 中编写的 button 标签本身。这意味着 p-button 实际上只是逻辑上的存在，所以绑定在 p-button 上的单击事件并不会发生任何作用。

由于 p-button 组件实际渲染在页面中的是其内部的 button 标签，所以在网页中实际单击的也是 button 标签，如果想要让自定义事件成功执行，则需要将 click 事件绑定在组件内部的 button 标签上，代码如下：

```
<!-- 第 3 章 自定义组件的事件绑定案例 2 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-button label="自定义按钮" @click="handleClick"></p-button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script type="text/javascript">
      Vue.component('p-button',{
        props:{
          label:{
            type:String,
          }
        },
        methods:{
          handleClick(e){
            //组件实际执行的单击事件
            console.log(e)
            console.log('组件内部的自定义事件')
          }
        },
        template:`
          <button @click="handleClick">{{label}}</button>
        `
      })
      new Vue({
        methods:{
          handleClick(){
            console.log('自定义事件')
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

运行自定义组件的事件绑定案例 2 并查看控制台，如图 3-6 所示。

图 3-6　自定义组件的事件绑定案例 2 的效果图

改造代码后发现单击按钮可以触发单击事件执行了。不过当前执行的单击事件并不是在 p-button 上绑定的事件，仅仅让组件内部的单击事件执行仍然无法解决问题。Vue 框架为所有自定义组件内置了 $emit()函数，代码如下：

```
this.$emit('事件名',参数)
```

如果想要触发自定义组件 p-button 上绑定的单击事件，则需要在 p-button 内部的单击事件中通过 this.$emit('click',e)来触发 p-button 上绑定的单击事件，通过传递第 2 个参数 e 可以将单击事件的默认参数传入 click 事件中，这样便可触发自定义组件上绑定的自定义事件，代码如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-button label="自定义按钮" @click="handleClick"></p-button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script type="text/javascript">
```

```
        Vue.component('p-button',{
          props:{
            label:{
              type:String,
            }
          },
          methods:{
            handleClick(e){
              //console.log(e)
              //console.log('组件内部的自定义事件')
              //该函数会触发 p-button 上定义的 click 事件,并将 e 对象传入其中
              this.$emit('click',e)
            }
          },
          template:`
            <button @click="handleClick">{{label}}</button>
          `
        })
        new Vue({
          methods:{
            handleClick(e){
              console.log(e)
              console.log('自定义事件,真正执行.')
            }
          }
        }).$mount('#app')
      </script>
    </body>
  </html>
```

自定义组件的事件绑定方式编写起来稍微有些复杂,不过它可以实现在事件触发时,将自定义组件的内部信息,通过参数传递的形式传递到自定义组件外部。这种方式可以灵活地处理 Vue 组件系统中的数据流向,让不同组件可以通过 props 与事件系统进行相互通信。

3.1.4　组件属性的双向绑定

在第 2 章中对 Vue 的双向绑定已经做了初步介绍,在 2.4.6 节中介绍了 v-model 除了可作用在 input、textarea 及 select 外,还可以作用在自定义组件上,所以在自定义组件中除了基本的属性与事件系统外,还可以通过 v-model 为自定义组件提供双向绑定的能力。接下来以自定义输入框的案例学习 v-model 在自定义组件中的应用。

1. 创建一个自定义 input 标签

在编辑器中创建一个自定义组件 p-input,为其定义属性 label 并作为 p-input 输入框中展示的数据。在 p-input 外部定义 str 变量,通过 v-bind 的方式将 str 变量绑定在 label 属性上传入 p-input 中,由于 label 属性在组件内部不推荐修改,所以在组件内部的 input 标签中

使用 v-bind 的方式将 label 的值绑定在 input 上，最后在视图中打印 str 的初始结果，代码如下：

```
<!-- 第 3 章 创建一个自定义 input 标签案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-input :label="str"></p-input>{{str}}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script type="text/javascript">
      Vue.component('p-input',{
        props:{
          label:{
            type:String,
          }
        },
        template:`
          <input :value="label"/>
        `
      })
      new Vue({
        data(){
          return {
            str:"默认值"
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

案例代码运行结果如图 3-7 所示。

2. 对 p-input 标签绑定自定义事件

p-input 标签定义后会发现自定义属性 label 的值可以被渲染在输入框的内部，不过在页面中修改输入框的值时并不会使外部的 str 属性的结果同步变化，这是因为以 v-bind 方式传入的属性是从外到内的，并不是双向绑定的。想要实现在 p-input 中改动输入框内容时，让 label 属性绑定的 str 的值同步更新可以利用事件触发的原理实现：在 p-input 内部的 input 输入框中绑定输入事件，用于监听文本框实时输入的内容，通过 $emit() 函数通知 p-input 上的自定义事件执行，并将输入框内部的值传递到组件外部，最后将组件外部得到的结果赋值给 str，这样便会触发 str 与 p-input 中的数据同步更新，代码如下：

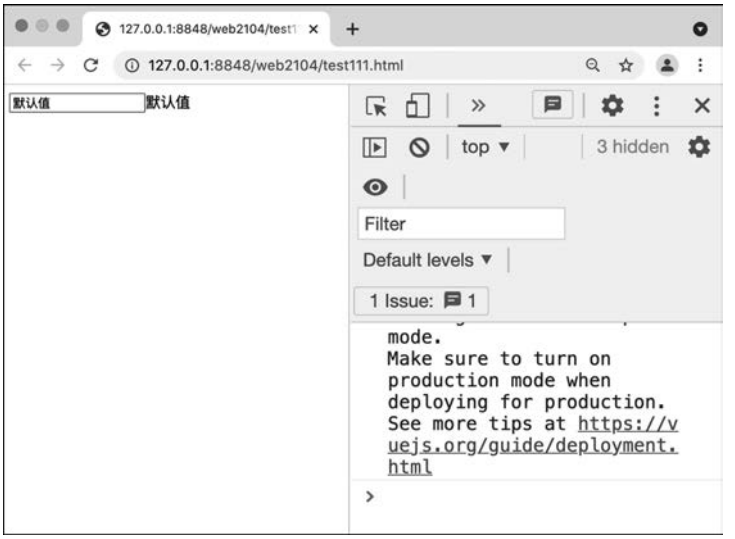


图 3-7　运行自定义 input 标签案例的效果图

```
<!-- 第 3 章 对 p-input 标签绑定自定义事件案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-input :label="str" @value-change="handleValueChange"></p-input>{{str}}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script type="text/javascript">
      Vue.component('p-input',{
        props:{
          label:{
            type:String,
          }
        },
        methods:{
          handleInput(e){
            //通过 e.target.value 获取文本框输入的值
            //当自定义事件的名称为多个单词时使用"-"进行分词
            this.$emit('value-change',e.target.value)
          }
        },
        template:`
          <input :value="label" @input="handleInput"/>
```

```
        `
      })
      new Vue({
        data(){
          return {
            str:"默认值"
          }
        },
        methods:{
          handleValueChange(v){
            this.str=v
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

通过自定义事件系统，已经可以将 p-input 中输入的内容同步到 str 内部，实现了双向绑定的效果，不过这种方式耦合性偏高，必须在自定义事件中编写代码逻辑才能将 p-input 中输入的内容实时地同步到 str 属性中。

3. 对 p-input 定义 v-model 指令

如果想要进一步简化代码并实现更理想的双向绑定，则需要对 p-input 定义 v-model 指令。v-model 指令在自定义组件中，需要开发者手动编写代码才能实现对应的能力。如果想要让自定义组件支持 v-model，则需要在组件内部定义 model 选项，用来告诉组件当前组件的 v-model 指令对应的值默认存储到组件的哪个 prop 上。还需要在 model 选项内部定义触发双向绑定的事件名称，代码如下：

```
<!-- 第3章 对p-input定义v-model指令案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <!-- 在自定义标签上直接使用v-model即可 -->
      <p-input v-model="str"></p-input>{{str}}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script type="text/javascript">
      Vue.component('p-input',{
        props:{
          label:{
```

```
            type:String,
          }
        },
        model:{
          prop:'label',//定义在 v-model 中设置的 str 的值默认绑定在哪个属性上
          event:'value-change'//定义触发 str 变化的函数名称,函数功能已由 v-model 在内部实现
        },
        methods:{
          handleInput(e){
            //直接调用 model 中 event 配置的函数名称,并将目标结果传入,即可实现在输入内容时
            //直接更改 str 的值
            this.$emit('value-change',e.target.value)
          }
        },
        template:`
          <input :value="label" @input="handleInput"/>
        `
      })
      new Vue({
        data(){
          return {
            str:"默认值"
          }
        },
        methods:{
          handleValueChange(v){
            this.str=v
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

使用 model 对象改造代码后，只需将 str 设置在 p-input 的 v-model 指令中，便可以实现与原生 input 标签中 v-model 完全相同的效果。由此可以分析 v-model 的实际原理：在使用 v-model 修饰 p-input 标签后，str 的值实际上保存在 label 属性中，所以在初始化时，str 属性的值可以直接显示在 p-input 内部的 value 属性位置。当通过输入触发 p-input 内部数据变更时，优先触发 p-input 内部的输入事件，并在事件中获得输入框实时输入的结果，将结果通过 model 中自带的 event 事件直接设置到 v-model 对应的 str 属性中，str 属性的值变更又会触发 p-input 组件自动更新，所以 v-model 实现的双向绑定实际上是一个环形的数据。接下来在组件内部和外部做一些改造，以此来更彻底地理解 v-model 的原理，代码如下：

```
<!DOCTYPE html>
<html>
```

```
<head>
  <meta charset="utf-8">
  <title></title>
</head>
<body>
  <div id="app">
    <!-- 在自定义标签上直接使用 v-model 即可 -->
    <p-input v-model="str" ></p-input>{{str}}
  </div>
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script type="text/javascript">
    Vue.component('p-input',{
      props:{
        label:{
          type:String,
        }
      },
      watch:{
        label(v){
          console.log('p-input 组件接收了新的值: '+v)
        }
      },
      model:{
        prop:'label',//定义在 v-model 中设置的 str 的值默认绑定在哪个属性上
        event:'value-change'//定义触发 str 变化的函数名称,函数功能已由 v-model 在内部实现
      },
      methods:{
        handleInput(e){
          console.log('通过输入改变了 p-input 组件的内容: '+e.target.value)
          //直接调用 model 中 event 配置的函数名称,并将目标结果传入,即可实现在输入内容时
          //直接更改 str 的值
          this.$emit('value-change',e.target.value)
        }
      },
      template:`
        <input :value="label" @input="handleInput"/>
      `
    })
    var vm=new Vue({
      data(){
        return {
          str:"默认值"
        }
      },
      watch:{
        str(v){
```

```
            console.log('str 的值发生了变化:' + v)
          }
        }
      }).$mount('#app')

    </script>
  </body>
</html>
```

运行案例代码，首先在输入框中改变文字内容并观察控制台的输出顺序，如图 3-8 所示。

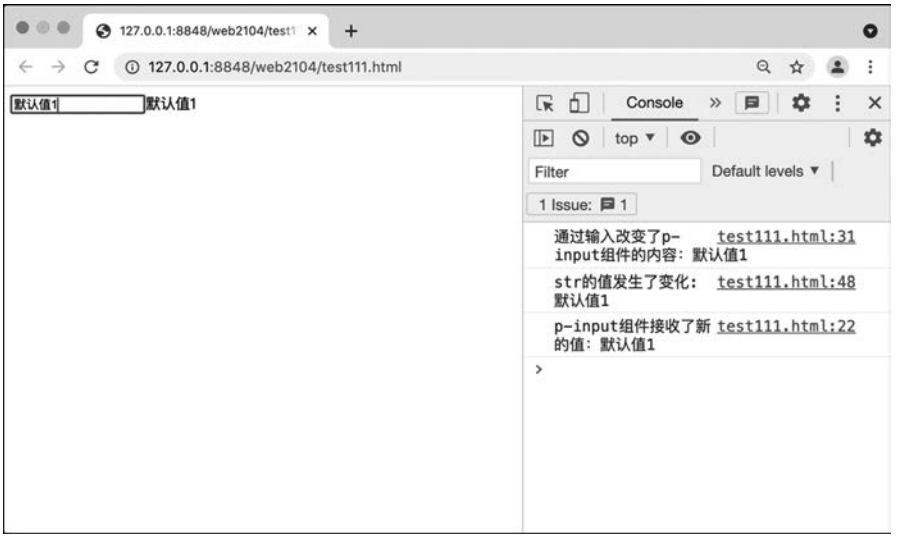


图 3-8　在输入框中输入的效果图

在输入框中输入触发数据变更后会发现，优先触发的是内部 input 标签的输入事件，在输入事件中可以直接获取本次输入的结果。接下来通过 $emit()函数调用，直接改变了 str 属性的值，进而触发了 str 的监听函数。最后由于 str 的值发生了变化，又触发了 p-input 组件的重新渲染，所以组件内部对 label 的监听又执行了一次。接下来在控制台中操作 vm 对象将 str 的值改变后观察控制台的输出，如图 3-9 所示。

通过 vm.str 修改了属性的值后，优先触发 str 属性的监听函数，输出 str 的值发生变化。由于 str 的变化触发了 label 属性的监听，进而 p-input 重新渲染，这样便实现了通过改变 str 触发 p-input 更新。结合图 3-8 与图 3-9 的对比结果会发现，无论通过页面输入还是对 vm.str 进行值的改变，最后两个步骤都是完全一样的，所以双向绑定的本质实际上是一个环形的路线，当通过自定义组件本身影响 v-model 绑定的属性时，数据会完整地走一整圈。当直接修改 v-model 绑定的属性时，数据只走半圈，如图 3-10 所示。

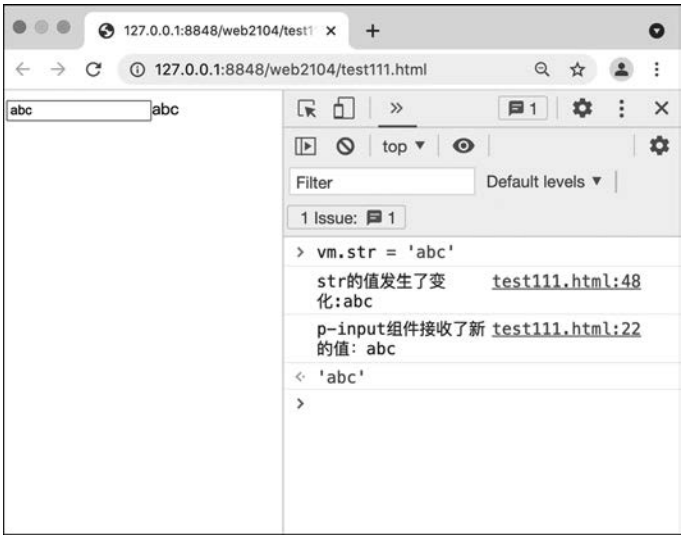


图 3-9　在控制台修改 str 的效果图

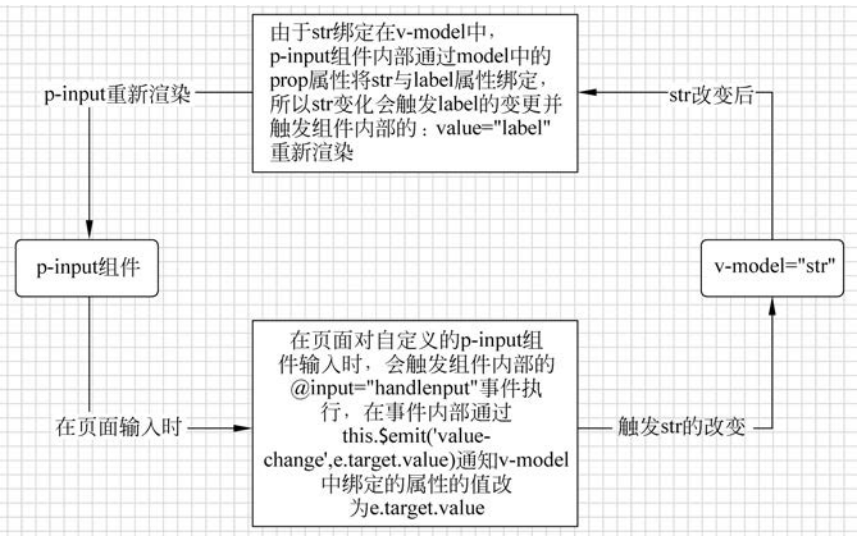


图 3-10　v-model 数据流向的效果图

3.1.5　组件属性的多重双向绑定

有些场景仅使用 v-model 进行数据的双向绑定并不能完美地解决场景的需求，当需要在同一个组件上实现多个属性的双向绑定时，一个 v-model 是完全不够用的。Vue 2. x 对多个属性的双向绑定需求也提供了完美的解决方案，代码如下：

```
<!-- 第 3 章 自定义组件多重双向绑定的格式 -->
<自定义组件 :属性 1.sync="变量 1" :属性 2.sync="变量 2"></自定义组件>
```

多重双向绑定的封装方式比 v-model 更加简洁，不需要在组件内部定义 model 选项，只需要在组件内部配合 this.$emit('update:要更新的属性',要更新的值)进行调用，便可以直接更改要更新的属性的值。硬性要求：必须在自定义组件中使用 props 提前定义好组件属性，在自定义组件上应用属性时必须携带.sync 字样。在 Vue 3.x 中多重双向绑定的实现方式与 Vue 2.x 不同，在后续的章节中会补充介绍。Vue 2.x 的多重双向绑定案例，代码如下：

```
<!-- 第 3 章 Vue 2.x 多重双向绑定案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <!-- 在自定义标签上直接使用 v-model 即可 -->
      <p-login :username.sync="username" :password.sync="password"></p-login>
      账号为{{username}}
      密码为{{password}}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script type="text/javascript">
      Vue.component('p-login',{
        props:{
          username:{
            type:String,
          },
          password:{
            type:String
          }
        },
        methods:{
          handleInput(e,key){
            this.$emit(`update:${key}`,e.target.value)
          }
        },
        template:`
          <form>
            账号:<input :value="username" @input="handleInput($event,'username')"/><br/>
            密码:<input :value="password" @input="handleInput($event,'password')"/><br/>
          </form>
        `
```

```
        })
        var vm=new Vue({
          data(){
            return {
              username:'',
              password:''
            }
          }
        }). $mount('#app')

      </script>
    </body>
  </html>
```

p-login 组件在内部定义了两个输入框组件，所以该组件需要实时地向组件外部反馈两个输入框中的结果。针对账号和密码两个输入框，在组件内部声明 username 和 password 属性进行一一对应。账号输入框在进行输入时，会触发对 username 属性的更新，this. $emit('update:${key}',e. target. value)会根据两个输入框触发输入时，分别解释成 this. $emit('update:username',e. target. value)及 this. $emit('update:password',e. target. value)，所以在修改账号时，组件外部绑定的 username 会自动更新，在修改密码时，组件外部的 password 会自动更新。具体的操作效果如图 3-11 所示。

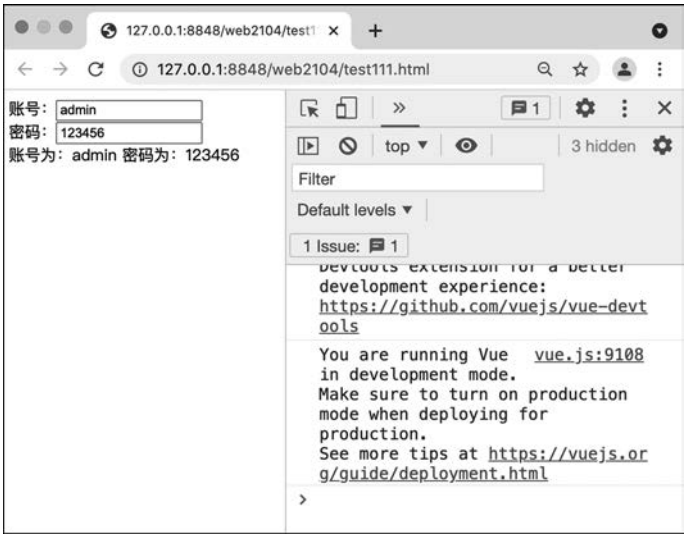


图 3-11 多重双向绑定操作的效果图

3.1.6 实现一个自定义 confirm 组件

下面通过一个实际的组件案例，进一步地夯实自定义组件的基础。confirm 询问框是

网页开发中最常用的组件之一，它的特点是可提升交互友好度，便于对操作行为进行判断。询问框结构主要包含标题部分、内容部分、操作按钮部分及关闭按钮部分。由于不同浏览器自带的询问框样式不统一并缺少定制功能，所以在实际开发中大多通过 HTML、CSS 及 JavaScript 结合手动实现询问框组件，在 Vue 框架中可以通过自定义组件的能力将询问框变成一个可复用的组件。

想要开发一个询问框组件，首先应根据实际开发经验分析一个询问框组件所具备的结构和功能。从结构和功能上来看，一个询问框应具备以下内容：

（1）有标题，可以展示当前弹出的询问框的所属类型。

（2）有关闭按钮，单击关闭按钮可以将询问框隐藏，通常在组件的右上角。

（3）有明确的内容提示，通过内容可以让用户知道当前询问框弹出的意图。

（4）有“确定”和“取消”两个按钮，询问框组件可以根据用户单击的按钮来自行判断用户的反馈结果。

（5）有背景遮罩，弹出层背景应是半透明遮罩，一方面可以突出询问框的展示效果，另一方面可以防止用户单击穿透。

（6）有事件反馈，开发者可以根据事件来确定用户单击了“确定”还是“取消”按钮。

分析完询问框的基本结构和能力，便可以开始进行询问框的组件构建了。在编写 Vue 代码前，首先要在 HTML 文件中单独通过 HTML 与 CSS 语言实现询问框的基本样式，代码如下：

```
<!-- 第 3 章 询问框基本样式的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .p-btn{
        padding: 5px 10px;
        background: #eee;
        outline: none;
        border:1px solid #ddd;
        color:#555;
        border-radius: 5px;
        cursor: pointer;
        position: relative;
      }
      .p-btn:hover:after{
        content: '';
        position: absolute;
        left: 0;
        top: 0;
        width: 100%;
```

```
    height: 100%;
    background-color: rgba(100,100,100,.2);
  }
  .p-btn:active:after{
    content: '';
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    background-color: rgba(100,100,100,.3);
  }
  .p-btn.primary{
    color: #fff;
    background: #409EFF;
    border-color: #409EFF;
  }
  .p-btn.success{
    color: #fff;
    background: #67C23A;
    border-color: #67C23A;
  }
  .p-btn.danger{
    color: #fff;
    background: #F56C6C;
    border-color: #F56C6C;
  }
  .p-btn.warning{
    color: #fff;
    background: #E6A23C;
    border-color: #E6A23C;
  }
  .p-confirm-bg{
    position: fixed;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    background-color: rgba(0,0,0,0.3);
    text-align: center;
  }
  .p-confirm-bg .p-confirm{
    text-align: left;
    background-color: #fff;
    display: inline-block;
    min-width: 300px;
    margin-top: 150px;
    border-radius: 9px;
    padding: 10px 15px;
```

```
        }
        .p-confirm-bg .p-confirm .p-confirm-title{
          color: #222;
          position: relative;
        }
        .p-confirm-bg .p-confirm .p-confirm-title .p-close{
          position: absolute;
          top: 0px;
          right: 0px;
          cursor: pointer;
          font-weight: bold;
        }
        .p-confirm-bg .p-confirm .p-confirm-title .p-close:hover{
          color: #666;
        }
        .p-confirm-bg .p-confirm .p-confirm-content{
          font-size: 14px;
          color: #444;
          padding: 5px 0px;
        }
      </style>
    </head>
    <body>
      <!--
        confirm 样式模板,CSS 文件在 assets 中
        运行本页并阅读代码,之后转到笔记中继续阅读
       -->
      <div class="p-confirm-bg">
        <div class="p-confirm">
          <div class="p-confirm-title">
            title
            <div class="p-close">
              &times;
            </div>
          </div>
          <div class="p-confirm-content">
            内容
          </div>
          <div class="p-confirm-btn">
            <button class="p-btn primary">确定</button>
            <button class="p-btn">取消</button>
          </div>
        </div>
      </div>
    </body>
  </html>
```

编写案例代码并使用浏览器运行,可以提前完成弹出后的询问框基本样式,如图 3-12 所示。

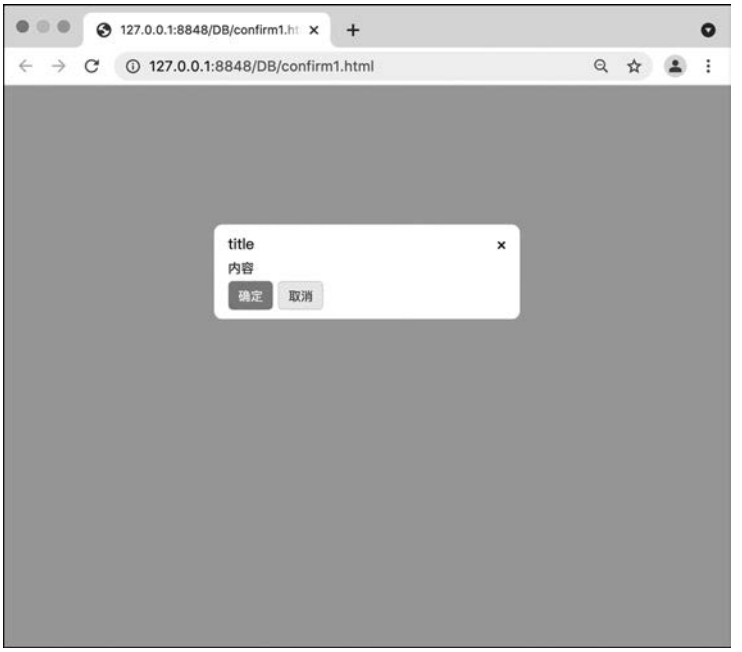


图 3-12　询问框基本样式效果图

接下来正式进入询问框组件的封装过程。

(1) 创建一个 HTML 文件并命名为 confirm.html,在文件中初始化 Vue 框架的基本结构,代码如下:

```
<!-- 第 3 章 初始化 Vue 框架基本结构的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">

    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">

      new Vue({}).$mount('#app')
    </script>
  </body>
</html>
```

（2）将询问框基本样式的 CSS 样式部分代码分别粘贴到基本代码结构中，代码如下：

```
<!-- 第 3 章 修改样式部分的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .p-btn{
        padding: 5px 10px;
        background: #eee;
        outline: none;
        border:1px solid #ddd;
        color:#555;
        border-radius: 5px;
        cursor: pointer;
        position: relative;
      }
      .p-btn:hover:after{
        content: '';
        position: absolute;
        left: 0;
        top: 0;
        width: 100%;
        height: 100%;
        background-color: rgba(100,100,100,.2);
      }
      .p-btn:active:after{
        content: '';
        position: absolute;
        left: 0;
        top: 0;
        width: 100%;
        height: 100%;
        background-color: rgba(100,100,100,.3);
      }
      .p-btn.primary{
        color: #fff;
        background: #409EFF;
        border-color: #409EFF;
      }
      .p-btn.success{
        color: #fff;
        background: #67C23A;
        border-color: #67C23A;
      }
      .p-btn.danger{
        color: #fff;
```

```
            background: #F56C6C;
            border-color: #F56C6C;
          }
          .p-btn.warning{
            color: #fff;
            background: #E6A23C;
            border-color: #E6A23C;
          }
          .p-confirm-bg{
            position: fixed;
            left: 0;
            top: 0;
            width: 100%;
            height: 100%;
            background-color: rgba(0,0,0,0.3);
            text-align: center;
          }
          .p-confirm-bg .p-confirm{
            text-align: left;
            background-color: #fff;
            display: inline-block;
            min-width: 300px;
            margin-top: 150px;
            border-radius: 9px;
            padding: 10px 15px;
          }
          .p-confirm-bg .p-confirm .p-confirm-title{
            color: #222;
            position: relative;
          }
          .p-confirm-bg .p-confirm .p-confirm-title .p-close{
            position: absolute;
            top: 0px;
            right: 0px;
            cursor: pointer;
            font-weight: bold;
          }
          .p-confirm-bg .p-confirm .p-confirm-title .p-close:hover{
            color: #666;
          }
          .p-confirm-bg .p-confirm .p-confirm-content{
            font-size: 14px;
            color: #444;
            padding: 5px 0px;
          }
        </style>
      </head>
      <body>
        <div id="app">
```

```
    </div>
    < script src =" https://cdn. jsdelivr. net/npm/vue/dist/vue. js" type =" text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">

      new Vue({}). $mount('#app')
    </script>
  </body>
</html>
```

（3）在 JavaScript 部分创建名为 p-confirm 的自定义组件，并将询问框基本样式案例中的 HTML 部分代码粘贴到组件的 template 属性中，代码如下：

```
<!-- 第 3 章 初始化 p-confirm 组件的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .p-btn{
        padding: 5px 10px;
        background: #eee;
        outline: none;
        border:1px solid #ddd;
        color:#555;
        border-radius: 5px;
        cursor: pointer;
        position: relative;
      }
      .p-btn:hover:after{
        content: '';
        position: absolute;
        left: 0;
        top: 0;
        width: 100%;
        height: 100%;
        background-color: rgba(100,100,100,.2);
      }
      .p-btn:active:after{
        content: '';
        position: absolute;
        left: 0;
        top: 0;
        width: 100%;
        height: 100%;
        background-color: rgba(100,100,100,.3);
```

```
}
.p-btn.primary{
  color: #fff;
  background: #409EFF;
  border-color: #409EFF;
}
.p-btn.success{
  color: #fff;
  background: #67C23A;
  border-color: #67C23A;
}
.p-btn.danger{
  color: #fff;
  background: #F56C6C;
  border-color: #F56C6C;
}
.p-btn.warning{
  color: #fff;
  background: #E6A23C;
  border-color: #E6A23C;
}
.p-confirm-bg{
  position: fixed;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0,0,0,0.3);
  text-align: center;
}
.p-confirm-bg .p-confirm{
  text-align: left;
  background-color: #fff;
  display: inline-block;
  min-width: 300px;
  margin-top: 150px;
  border-radius: 9px;
  padding: 10px 15px;
}
.p-confirm-bg .p-confirm .p-confirm-title{
  color: #222;
  position: relative;
}
.p-confirm-bg .p-confirm .p-confirm-title .p-close{
  position: absolute;
  top: 0px;
  right: 0px;
  cursor: pointer;
  font-weight: bold;
```

```
        }
        .p-confirm-bg .p-confirm .p-confirm-title .p-close:hover{
          color: #666;
        }
        .p-confirm-bg .p-confirm .p-confirm-content{
          font-size: 14px;
          color: #444;
          padding: 5px 0px;
        }
      </style>
    </head>
    <body>
      <div id="app">
      </div>
      <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
      <script type="text/javascript">
        //初始化 p-confirm 组件
        Vue.component('p-confirm',{
          template:`
            <div class="p-confirm-bg">
              <div class="p-confirm">
                <div class="p-confirm-title">
                  title
                  <div class="p-close">
                    &times;
                  </div>
                </div>
                <div class="p-confirm-content">
                  内容
                </div>
                <div class="p-confirm-btn">
                  <button class="p-btn primary">确定</button>
                  <button class="p-btn">取消</button>
                </div>
              </div>
            </div>
          `
        })
        new Vue({}).$mount('#app')
      </script>
    </body>
  </html>
```

(4) 在组件中加入开关功能。由于询问框是以弹出形式展示的,所以询问框默认不显示在页面中,通过开关控制其显示或隐藏。由于组件的开关应在组件内部和外部都可操作,所以组件的开关应为双向绑定数据。接下来要对组件定义 props 选项及 model 选项实现组件的开关功能,代码如下:

```
<!-- 第 3 章 实现组件开关功能的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <link rel="stylesheet" type="text/css"
     href="assets/confirm.css"/>
    <style type="text/css">
      .p-btn{
        padding: 5px 10px;
        background: #eee;
        outline: none;
        border:1px solid #ddd;
        color:#555;
        border-radius: 5px;
        cursor: pointer;
        position: relative;
      }
      .p-btn:hover:after{
        content: '';
        position: absolute;
        left: 0;
        top: 0;
        width: 100%;
        height: 100%;
        background-color: rgba(100,100,100,.2);
      }
      .p-btn:active:after{
        content: '';
        position: absolute;
        left: 0;
        top: 0;
        width: 100%;
        height: 100%;
        background-color: rgba(100,100,100,.3);
      }
      .p-btn.primary{
        color: #fff;
        background: #409EFF;
        border-color: #409EFF;
      }
      .p-btn.success{
        color: #fff;
        background: #67C23A;
        border-color: #67C23A;
      }
      .p-btn.danger{
        color: #fff;
```

```
        background: #F56C6C;
        border-color: #F56C6C;
      }
      .p-btn.warning{
        color: #fff;
        background: #E6A23C;
        border-color: #E6A23C;
      }
      .p-confirm-bg{
        position: fixed;
        left: 0;
        top: 0;
        width: 100%;
        height: 100%;
        background-color: rgba(0,0,0,0.3);
        text-align: center;
      }
      .p-confirm-bg .p-confirm{
        text-align: left;
        background-color: #fff;
        display: inline-block;
        min-width: 300px;
        margin-top: 150px;
        border-radius: 9px;
        padding: 10px 15px;
      }
      .p-confirm-bg .p-confirm .p-confirm-title{
        color: #222;
        position: relative;
      }
      .p-confirm-bg .p-confirm .p-confirm-title .p-close{
        position: absolute;
        top: 0px;
        right: 0px;
        cursor: pointer;
        font-weight: bold;
      }
      .p-confirm-bg .p-confirm .p-confirm-title .p-close:hover{
        color: #666;
      }
      .p-confirm-bg .p-confirm .p-confirm-content{
        font-size: 14px;
        color: #444;
        padding: 5px 0px;
      }
    </style>
  </head>
  <body>
    <div id="app">
```

```
        <button @click="handleClick">切换展示 confirm</button>
        <p-confirm v-model="show"></p-confirm>
      </div>
      <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
      <script type="text/javascript">
        Vue.component('p-confirm',{
          props:{
            show:{
              type:Boolean,
              default:false
            },
          },
          model:{
            prop:'show',
            event:'change-show'
          },
          methods:{
            handleClose(){
              //通知 v-model 将 show 的值更新为 false
              this.$emit('change-show',false)
              //通知关闭的回调函数执行
              this.$emit('close')
            }
          },
          template:`
            <div v-if="show" class="p-confirm-bg">
              <div class="p-confirm">
                <div class="p-confirm-title">
                  title
                  <div class="p-close" @click="handleClose">
                    &times;
                  </div>
                </div>
                <div class="p-confirm-content">
                  内容
                </div>
                <div class="p-confirm-btn">
                  <button class="p-btn primary">确定</button>
                  <button class="p-btn">取消</button>
                </div>
              </div>
            </div>
          `,
        })
        new Vue({
          data(){
            return {
              show:false,
```

```
            title:'系统提示',
            content:`
              <p>
                <span
                  style="color:red;font-weight:bold"
                >正在进行 xx 操作,单击"确定"按钮继续</span>

              </p>

            `
          }
        },
        methods:{
          handleClick(){
            this.show=!this.show
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

(5) 为了完善组件的完整功能,接下来需要将组件的标题、组件的提示内容及按钮文字等全部变成动态传入的参数,还要完善组件的事件系统,最后可以再补充组件的过渡动画。开发完整的询问框组件,代码如下:

```
<!-- 第 3 章 完整的询问框组件的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .p-btn{
        padding: 5px 10px;
        background: #eee;
        outline: none;
        border:1px solid #ddd;
        color:#555;
        border-radius: 5px;
        cursor: pointer;
        position: relative;
      }
      .p-btn:hover:after{
        content: '';
        position: absolute;
        left: 0;
```

```
    top: 0;
    width: 100%;
    height: 100%;
    background-color: rgba(100,100,100,.2);
  }
  .p-btn:active:after{
    content: '';
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    background-color: rgba(100,100,100,.3);
  }
  .p-btn.primary{
    color: #fff;
    background: #409EFF;
    border-color: #409EFF;
  }
  .p-btn.success{
    color: #fff;
    background: #67C23A;
    border-color: #67C23A;
  }
  .p-btn.danger{
    color: #fff;
    background: #F56C6C;
    border-color: #F56C6C;
  }
  .p-btn.warning{
    color: #fff;
    background: #E6A23C;
    border-color: #E6A23C;
  }
  .p-confirm-bg{
    position: fixed;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    background-color: rgba(0,0,0,0.3);
    text-align: center;
  }
  .p-confirm-bg .p-confirm{
    text-align: left;
    background-color: #fff;
    display: inline-block;
    min-width: 300px;
    margin-top: 150px;
```

```
        border-radius: 9px;
        padding: 10px 15px;
      }
      .p-confirm-bg .p-confirm .p-confirm-title{
        color: #222;
        position: relative;
      }
      .p-confirm-bg .p-confirm .p-confirm-title .p-close{
        position: absolute;
        top: 0px;
        right: 0px;
        cursor: pointer;
        font-weight: bold;
      }
      .p-confirm-bg .p-confirm .p-confirm-title .p-close:hover{
        color: #666;
      }
      .p-confirm-bg .p-confirm .p-confirm-content{
        font-size: 14px;
        color: #444;
        padding: 5px 0px;
      }
      .fade-enter-active{
        animation-name: fade-in;
        animation-duration: 0.5s;
      }
      .fade-leave-active{
        animation-name: fade-in;
        animation-duration: 0.5s;
        animation-direction: reverse;
      }
      @keyframes fade-in{
        from{opacity:0}
        to{opacity:1}
      }
      .p-confirm{
        animation-name: slide-in;
        animation-duration: .3s;
      }
      @keyframes slide-in{
         from{opacity: 0; transform: translateY( -30px);}
         to{opacity: 1; transform: translateY(0px);}
      }
    </style>
  </head>
  <body>
    <div id="app">
      <button @click="handleClick">切换展示 confirm </button>
      {{show}}
```

```
        <p-confirm v-model="show"
          :title="title"
          :content="content"
          :confirm-button-text="'确定'"
          :cancel-button-text="'关闭'"
          @confirm="handleConfirm"
          @close="handleClose"></p-confirm>
      </div>
      <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
      <script type="text/javascript">
        Vue.component('p-confirm',{
          props:{
            show:{
              type:Boolean,
              default:false
            },
            title:{
              type:String,
              default:''
            },
            content:{
              type:String,
              default:''
            },
            confirmButtonText:{
              type:String,
              default:'确定'
            },
            cancelButtonText:{
              type:String,
              default:'取消'
            }
          },
          model:{
            prop:'show',
            event:'change-show'
          },
          methods:{
            handleClose(){
              //通知 v-model 将 show 的值更新为 false
              this.$emit('change-show',false)
              //通知关闭的回调函数执行
              this.$emit('close')
            },
            handleConfirm(){
              this.$emit('change-show',false)
              //通知确定的回调函数执行
              this.$emit('confirm')
```

```
    }
  },
  template:`
    <transition name="fade">
      <div v-if="show" class="p-confirm-bg">
        <div class="p-confirm">
          <div class="p-confirm-title">
            {{title}}
            <div class="p-close"
              @click="handleClose">
              &times;
            </div>
          </div>
          <div class="p-confirm-content">

            <div v-html="content">
            </div>
          </div>
          <div class="p-confirm-btn">
            <button class="p-btn primary"
              @click="handleConfirm">
              {{confirmButtonText}}
            </button>
            <button @click="handleClose" class="p-btn">
              {{cancelButtonText}}
            </button>
          </div>
        </div>
      </div>
    </transition>
  `,
})
new Vue({
  data(){
    return {
      show:false,
      title:'系统提示',
      content:`
        <p>
          <span
            style="color:red;font-weight:bold"
          >正在进行 xx 操作,单击"确定"按钮继续</span>

        </p>

      `
    }
  },
  methods:{
```

```
            handleClick(){
              console.log('danji')
              this.show=!this.show
            },
            handleConfirm(){
              console.log('单击了确定')
            },
            handleClose(){
              console.log('单击了关闭')
            }
          }
        }).$mount('#app')
      </script>
    </body>
</html>
```

完整的 p-confirm 组件运行结果如图 3-13 所示。

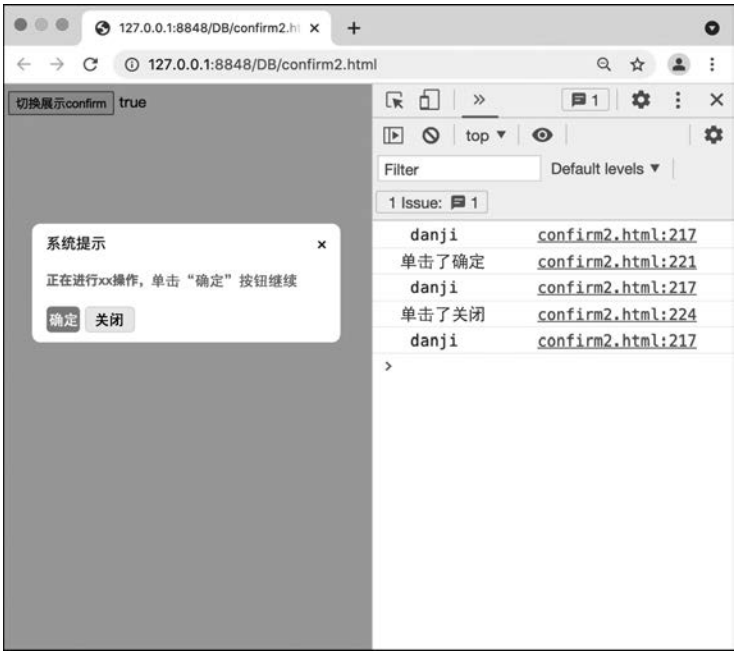


图 3-13　完整的询问框组件的效果图

3.2　Vue 的插槽介绍

插槽是 Vue 组件化开发场景中必不可少的组件之一，在仅使用 props 选项和 this.$emit()配合开发的组件存在一个缺陷：当开发按钮或可嵌套组件时，使用 props 选项定义

的属性必须在组件标签上定义和传入。如果希望自定义组件是可嵌套组件，则在组件双标签结构中间编写的内容就会不翼而飞。插槽就是为了解决自定义可嵌套组件而开发的解决方案。

1. 插槽的概念介绍

Vue 实现了一套内容分发的 API，这套 API 的设计灵感源自 Web Components 规范草案，将 <slot> 元素作为承载分发内容的出口。在 Vue 中使用插槽的示例，代码如下：

```
<!-- 第 3 章 在 Vue 中使用插槽的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-button>
        我是按钮名称
      </p-button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-button',{
        template:`
          <button>
            <slot></slot>
          </button>
        `
      })
      new Vue({}).$mount('#app')
    </script>
  </body>
</html>
```

如果定义的组件是双标签组件，则标签中间输入的内容会自动分发到<slot>组件中，案例中的自定义组件无须使用 props，便可以将<p-button>标签中间输入的内容渲染到实际的<button>按钮内，这就是插槽的最基本使用方式。

2. 具名插槽

Vue 的插槽系统可以实现自定义组件管理双标签内部代码的功能，这样开发者便可以创建可嵌套的自定义组件。插槽在默认的使用场景可以解决大部分开发需求，考虑到特殊场景，<slot>还可以通过命名实现，将不同的标签分发到组件内部的不同位置，例如在开发后台管理系统时，经常需要设计上、左、右结构的页面，开发者可先定义好布局组件的样式，再通过具名插槽，将组件内容指定分配到页面的相应部分。具名插槽使用方式的代码如下：

```
<!-- 第 3 章 具名插槽使用方式的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      html,body,#app{
        margin: 0;
        height: 100%;
      }
      .p-layout{
        width: 100%;
        height: 100%;
        display: flex;
        flex-direction: column;
      }
      .p-layout .p-header{
        height: 60px;
        background-color: #409EFF;
      }
      .p-layout .p-container{
        flex-grow: 1;
        display: flex;
      }
      .p-layout .p-container .p-left{
        width: 300px;
        height: 100%;
        background-color: #E6A23C;
      }
      .p-layout .p-container .p-right{
        flex-grow: 1;
        height: 100%;
        background-color: antiquewhite;
      }
    </style>
  </head>
  <body>
    <div id="app">
      <p-layout>
        <!-- 分配到头部的部分 -->
        <template v-slot:header>
          头部
        </template>
        <!-- 分配到左侧的部分 -->
        <template v-slot:left>
```

```
            左侧部分
          </template>
          <!-- 分配到右侧的部分 -->
          <template v-slot:right>
            右侧部分
          </template>
        </p-layout>
      </div>
      <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
      <script type="text/javascript">
        Vue.component('p-layout',{
          template:`
            <div class="p-layout">
              <div class="p-header">
                <slot name="header"></slot>
              </div>
              <div class="p-container">
                <div class="p-left">
                  <slot name="left"></slot>
                </div>
                <div class="p-right">
                  <slot name="right"></slot>
                </div>
              </div>
            </div>
          `
        })
        new Vue({}).$mount('#app')
      </script>
    </body>
  </html>
```

具名插槽的使用方式非常简单，在组件内部使用<slot>标签时需要对其定义的 name 属性进行命名，在组件应用的代码内部需要通过<template>标签的 v-slot 指令来配置内容分发。使用“v-slot:名称”的方式，将当前编写的<template>中的内容分发到命名相同的<slot>标签内部。具名插槽代码案例中通过定义<p-layout>实现了自动分发内容的上、左、右布局组件，其运行效果如图 3-14 所示。

3. 作用域插槽

作用域插槽是插槽的高级使用方式，它主要应用于动态数据处理，例如在原生 HTML 代码中使用<table>表格结合列表渲染，当将 JavaScript 中的列表数据结构渲染到网页时，可以将表格结合列表渲染的部分进一步抽象成自定义表格组件。在这个应用场景中，作用域插槽便可以实现组件的高级功能。在学习作用域插槽前，应先通过代码实现自定义表格组件，代码如下：

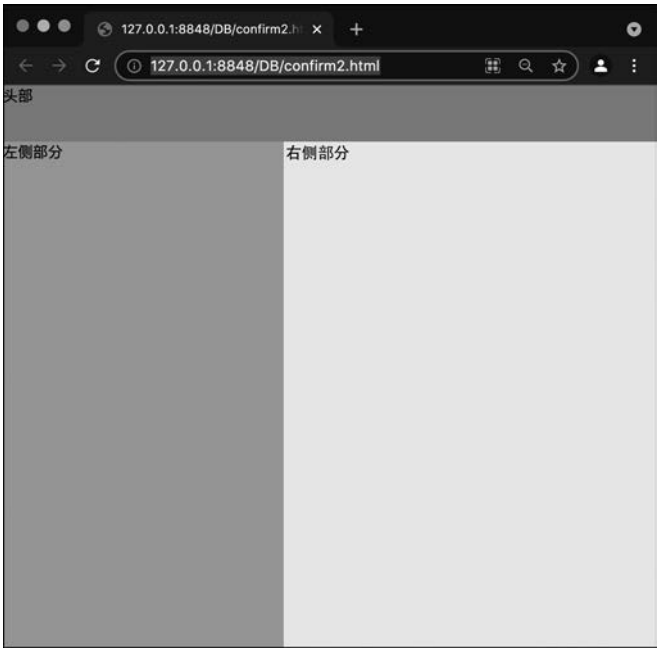


图 3-14　具名插槽案例的效果图

```
<!-- 第 3 章 自定义表格组件的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-table :data="list" :header="header"></p-table>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-table',{
        props:{
          data:{                                    //表格的数据参数
            type:Array,
            default(){
              return []
            }
          },
          header:{                                  //表头的描述参数
            type:Array,
```

```
        default(){
          return []
        }
      }
    },
    template:`
      <table border="1">
        <thead>
          <tr>
            <th v-for="item in header" :key="item.label">
              {{item.label}}
            </th>
          </tr>
        </thead>
        <tbody>
          <tr v-for="item in data" :key="item.id">
            <td v-for="itemHead in header" :key="itemHead.label">
              {{item[itemHead.prop]}}
            </td>
          </tr>
        </tbody>
      </table>
    `
  })
  new Vue({
    data(){
      return {
        //表格的初始数据
        list:[
          {
            id:'a001',
            name:'阳顶天',
            phone:'17388889887',
            email:'xxx@xxx.com',
            birthday:'20xx-0x-0x'
          },
          {
            id:'a002',
            name:'杨逍',
            phone:'17388889888',
            email:'xxx@xxx.com',
            birthday:'20xx-0x-0x'
          },
          {
            id:'a003',
            name:'张无忌',
            phone:'17388889889',
            email:'xxx@xxx.com',
            birthday:'20xx-0x-0x'
```

```
            },
            {
              id:'a004',
              name:'张三丰',
              phone:'17388889880',
              email:'xxx@xxx.com',
              birthday:'20xx-0x-0x'
            }
          ],
          //表头的初始数据,prop 代表本列从数组中提取指定属性名称
          //label 代表当前列名
          header:[
            {
              prop:'name',
              label:'姓名'
            },
            {
              prop:'phone',
              label:'电话'
            },
            {
              prop:'email',
              label:'邮箱'
            },
            {
              prop:'birthday',
              label:'生日'
            },
            {
              label:'操作'
            }
          ]
        }
      }
    }).$mount('#app')
  </script>
 </body>
</html>
```

自定义表格组件中包含两个属性,即 data 和 header。data 代表表格中渲染的数据内容,数据格式是一个对象数组。header 代表表格的表头部分数据描述,用来配置表头的列名及当前列在 data 中所对应的属性名称。两个属性按照要求设置好初始值后,表格便会自动按照数据渲染到网页中,如图 3-15 所示。

在当前的自定义表格中,已经实现了自动渲染数据的能力,由于在 header 中定义了一个名为“操作”的列,该列在 list 属性中没有任何可对应的数据,所以该列在展示时内部为空值。根据通用组件封装的要求,如果想让自定义表格在开发中可以适应更多的业务场景,开

图 3-15　自定义表格案例的效果图

发者则应对自定义表格的列进行自定义。除最基本的数据展示外，开发者应有权向表格的任意单元格内增加自定义内容，所以表格中的每个单元格都应允许通过具名插槽进行自定义。接下来需要在组件内改造代码，为每个单元格定义插槽，代码如下：

```
<!-- 第 3 章 加入具名插槽的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-table :data="list" :header="header">
        <!-- 以英雄名称为例子,将插槽内容分发到自定义表格内部 -->
        <template v-slot:name>
          英雄名称
        </template>
      </p-table>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-table',{
        props:{
          data:{                                    //表格的数据参数
            type:Array,
            default(){
              return []
            }
          },
```

```
    header:{                        //表头的描述参数
      type:Array,
      default(){
        return []
      }
    }
  },
  template:`
    <table border="1">
      <thead>
        <tr>
          <th v-for="item in header" :key="item.label" >
            {{item.label}}
          </th>
        </tr>
      </thead>
      <tbody>
        <tr v-for="item in data" :key="item.id">
          <td v-for="itemHead in header" :key="itemHead.label">
            <!--
              在组件应用时如果未使用对应的具名插槽,则会展示<slot>内部的数据
              如果使用了具名插槽,则<slot>会被渲染为外部传入的数据
             -->
            <slot :name="itemHead.prop">
              {{item[itemHead.prop]}}
            </slot>
          </td>
        </tr>
      </tbody>
    </table>
  `
})
new Vue({
  data(){
    return {
      //表格的初始数据
      list:[
        {
          id:'a001',
          name:'阳顶天',
          phone:'17388889887',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        },
        {
          id:'a002',
          name:'杨逍',
          phone:'17388889888',
          email:'xxx@xxx.com',
```

```
                birthday:'20xx-0x-0x'
              },
              {
                id:'a003',
                name:'张无忌',
                phone:'17388889889',
                email:'xxx@xxx.com',
                birthday:'20xx-0x-0x'
              },
              {
                id:'a004',
                name:'张三丰',
                phone:'17388889880',
                email:'xxx@xxx.com',
                birthday:'20xx-0x-0x'
              }
            ],
            //表头的初始数据,prop 代表本列从数组中提取指定属性名称
            //label 代表当前列名
            header:[
              {
                prop:'name',
                label:'姓名'
              },
              {
                prop:'phone',
                label:'电话'
              },
              {
                prop:'email',
                label:'邮箱'
              },
              {
                prop:'birthday',
                label:'生日'
              },
              {
                prop:'control',
                label:'操作'
              }
            ]
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

改造案例代码后,list 属性传入的数据仍然能正常触发表格内容渲染,只有指定了插槽

的列才会应用外部传入的插槽内容，不过应用了具名插槽的列的数据会出现问题。具体效果如图 3-16 所示。

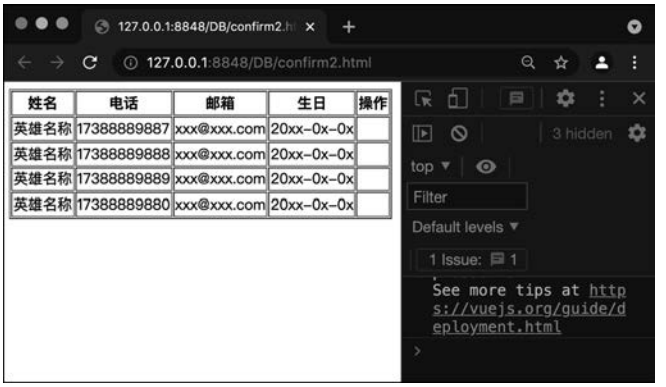


图 3-16　增加具名插槽的效果图

增加具名插槽后，可以实现插槽的内容分发，由于表格内容数据是数组类型，定义的<template v-slot:name>插槽内容会在列表渲染时渲染成多行数据。虽然传入插槽中的“英雄名称”可以在每行中展示，但使用插槽时会覆盖当前列的原始数据，这导致通过插槽渲染的数据无法获取 name 属性每行的原始结果，作用域插槽便是在这种高级组件的应用场景中解决此问题而存在的。<slot>标签上除 name 属性外，还可以在组件源代码中通过 v-bind 绑定自定义属性，一旦绑定了自定义属性，<template>标签上便可以获取组件内部绑定的自定义属性。作用域插槽实际应用的代码如下：

```
<!-- 第 3 章 作用域插槽实际应用的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-table :data="list" :header="header">
        <!-- 在 v-slot:name="row"中可以获取 slot 中绑定的 row 属性的值 -->
        <template v-slot:name="row">
          英雄名称{{row}}
        </template>
      </p-table>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-table',{
```

```
  props:{
    data:{                              //表格的数据参数
      type:Array,
      default(){
        return []
      }
    },
    header:{                            //表头的描述参数
      type:Array,
      default(){
        return []
      }
    }
  },
  template:`
    <table border="1">
      <thead>
        <tr>
          <th v-for="item in header" :key="item.label" >
            {{item.label}}
          </th>
        </tr>
      </thead>
      <tbody>
        <tr v-for="item in data" :key="item.id">
          <td v-for="itemHead in header" :key="itemHead.label">
            <!--
              在 slot 上绑定了 row 属性,并将 data 中的每一行数据 item 设置到 row 中
            -->
            <slot :name="itemHead.prop" :row="item">
              {{item[itemHead.prop]}}
            </slot>
          </td>
        </tr>
      </tbody>
    </table>
  `
})
new Vue({
  data(){
    return {
      //表格的初始数据
      list:[
        {
          id:'a001',
          name:'阳顶天',
          phone:'17388889887',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
```

```
        },
        {
          id:'a002',
          name:'杨逍',
          phone:'17388889888',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        },
        {
          id:'a003',
          name:'张无忌',
          phone:'17388889889',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        },
        {
          id:'a004',
          name:'张三丰',
          phone:'17388889880',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        }
      ],
      //表头的初始数据,prop 代表本列从数组中提取指定属性名称
      //label 代表当前列名
      header:[
        {
          prop:'name',
          label:'姓名'
        },
        {
          prop:'phone',
          label:'电话'
        },
        {
          prop:'email',
          label:'邮箱'
        },
        {
          prop:'birthday',
          label:'生日'
        },
        {
          prop:'control',
          label:'操作'
        }
      ]
    }
  }
```

```
        }). $mount('#app')
      </script>
    </body>
  </html>
```

代码改造后，可直接通过 v-slot:name="row"获取组件内部绑定的 row 属性的值。通过这种方式，可将组件升级为高可定制组件。当开发者不需要自定义表格内部数据时，可直接将数据渲染到表格内部。开发者对任何单元格内部有定制开发需求时，可以通过<template>进行内容分发，以此来自定义每个单元格的数据内容和展示样式。作用域插槽案例的运行结果如图 3-17 所示。

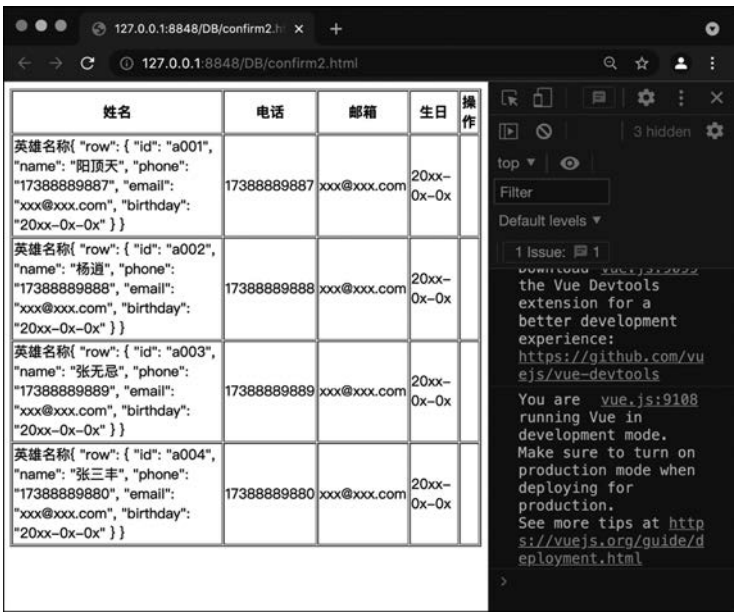


图 3-17　作用域插槽的效果图

接下来将作用域插槽案例代码继续丰富，将姓名列的数据使用插槽展示，将电话号码数据的中间四位数变成 * 号，在操作列中加入删除按钮并实现数据的删除功能。作用域插槽的最终案例，代码如下：

```
<!-- 第 3 章 作用域插槽最终的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
```

```
    <div id="app">
      <p-table :data="list" :header="header">
        <!-- 在 v-slot:name="row"中可以获取 slot 中绑定的 row 属性的值 -->
        <template v-slot:name="{row}">
          {{row.name}}
        </template>
        <!-- 通过计算属性格式化手机号码 -->
        <template v-slot:phone="{row}">
          {{formatPhone(row.phone)}}
        </template>
        <!-- 通过插槽实现对每行定义删除按钮,并将每行的数据传入来确定要删除的数据 -->
        <template v-slot:control="{row}">
          <button @click="handleDelete(row)">删除</button>
        </template>
      </p-table>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-table',{
        props:{
          data:{                                    //表格的数据参数
            type:Array,
            default(){
              return []
            }
          },
          header:{                                  //表头的描述参数
            type:Array,
            default(){
              return []
            }
          }
        },
        template:`
          <table border="1">
            <thead>
              <tr>
                <th v-for="item in header" :key="item.label" >
                  {{item.label}}
                </th>
              </tr>
            </thead>
            <tbody>
              <tr v-for="item in data" :key="item.id">
                <td v-for="itemHead in header" :key="itemHead.label">
                  <!--
                    在 slot 上绑定了 row 属性,并将 data 中的每一行数据 item 设置到 row 中
                  -->
```

```
                <slot :name="itemHead.prop" :row="item">
                  {{item[itemHead.prop]}}
                </slot>
              </td>
            </tr>
          </tbody>
        </table>
      `
})
new Vue({
  data(){
    return {
      //表格的初始数据
      list:[
        {
          id:'a001',
          name:'阳顶天',
          phone:'17388889887',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        },
        {
          id:'a002',
          name:'杨逍',
          phone:'17388889888',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        },
        {
          id:'a003',
          name:'张无忌',
          phone:'17388889889',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        },
        {
          id:'a004',
          name:'张三丰',
          phone:'17388889880',
          email:'xxx@xxx.com',
          birthday:'20xx-0x-0x'
        }
      ],
      //表头的初始数据,prop 代表本列从数组中提取指定属性名称
      //label 代表当前列名
      header:[
        {
          prop:'name',
          label:'姓名'
```

```
            },
            {
              prop:'phone',
              label:'电话'
            },
            {
              prop:'email',
              label:'邮箱'
            },
            {
              prop:'birthday',
              label:'生日'
            },
            {
              prop:'control',
              label:'操作'
            }
          ]
        }
      },
      computed:{
        formatPhone(){
          //计算属性可以使用返回一个函数的方式来接收外部传入的参数
          return function(phone){
            let prefix=phone.substring(0,3)
            let suffix=phone.substring(7)
            return `${prefix}**** ${suffix}`
          }
        }
      },
      methods:{
        handleDelete(row){
          let res=window.confirm('正在删除该数据,是否继续?')
          if(res){
            //通过 filter 循环将要删除的数据过滤掉
            this.list=this.list.filter(item=> item.id !=row.id )
          }
        }
      }
    }).$mount('#app')
  </script>
 </body>
</html>
```

4. 递归组件

熟练掌握插槽用法,可以帮助开发者实现高级组件的定义,这里就包括 Vue 中最常用的递归组件。递归组件允许组件动态解析一个未知深度的属性结构,通常应用在级联菜单或树形组件中,本节以属性组件为例介绍递归组件的实际应用。首先需要定义一组未知深

度的树形数据结构，代码如下：

```
<!-- 第 3 章 树形数据结构的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">

    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      new Vue({
        data(){
          return {
            //初始树形数据
            data:[
              {
                id:'1',
                name:'运营事业部',
                children:[
                  {
                    id:'1-1',
                    name:'IT 研发一部',
                    children:[
                      {
                       id:'1-1-1',
                       name:'研发 1 组'
                      },
                      {
                       id:'1-1-2',
                       name:'研发 2 组'
                      }
                    ]
                  },
                  {
                    id:'1-2',
                    name:'运营部'
                  },
                  {
                    id:'1-3',
                    name:'财务部'
                  }
                ]
              },
```

```
            {
              id:'2',
              name:'营销事业部',
              children:[
                {
                 id:'2-1',
                 name:'售前服务部',
                 children:[
                   {
                    id:'2-1-1',
                    name:'服务 1 组'
                   },
                   {
                    id:'2-1-2',
                    name:'服务 2 组'
                   }
                  ]
                },
                {
                  id:'2 - 2',
                  name:'售后服务部',
                  children:[
                    {
                     id:'2-1-1',
                     name:'反馈 1 组',
                     children:[
                       {
                        id:'2-1-1-1',
                        name:'第一梯队'
                       },
                       {
                        id:'2-1-1-2',
                        name:'第二梯队'
                       }
                      ]
                    },
                    {
                      id:'2-1-2',
                      name:'反馈 2 组'
                    }
                   ]
                  }
                ]
              }
            ]
          }
        }
      }). $mount('#app')
    </script>
  </body>
</html>
```

在 data 中定义的 data 属性是一个非对称树形结构，该树的每个节点的子节点数据深度各不相同，假设 data 属性定义的数据为服务器端返回的数据，每次返回数据的节点内容和深度可能不同，这样就无法通过固定层数的循环来解决数据的渲染。递归组件的设计方案便是解决此类数据渲染的解决方案。接下来需要定义一个名为 p-tree 的自定义组件，并通过组件递归的方式解决初始组件渲染问题，代码如下：

```
<!-- 第 3 章 递归组件的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-tree :data="data"></p-tree>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-tree',{
        props:{
          data:{
            type:Array,
            default(){
              return []
            }
          }
        },
        template:`
          <ul>
            <li v-for="item in data" :key="data.id">
              {{item.name}}
              <!-- 递归组件的用法是当组件内部需要未知层深的内容展示时,可以直接将组件
本身作为渲染内容在组件内部调用 -->
              <!-- 在组件渲染时,p-tree 会自动识别当前节点是否有后代,如果有后代就会触发
链式渲染 -->
              <p-tree v-if="item.children" :data="item.children"><p-tree>
            </li>
          </ul>
        `
      })
      new Vue({
        data(){
          return {
            //初始树形数据
            data:[
```

```
{
  id:'1',
  name:'运营事业部',
  children:[
    {
      id:'1-1',
      name:'IT 研发一部',
      children:[
        {
          id:'1-1-1',
          name:'研发 1 组'
        },
        {
          id:'1-1-2',
          name:'研发 2 组'
        }
      ]
    },
    {
      id:'1-2',
      name:'运营部'
    },
    {
      id:'1-3',
      name:'财务部'
    }
  ]
},
{
  id:'2',
  name:'营销事业部',
  children:[
   {
     id:'2-1',
     name:'售前服务部',
     children:[
       {
        id:'2-1-1',
        name:'服务 1 组'
       },
       {
        id:'2-1-2',
        name:'服务 2 组'
       }
     ]
   },
   {
     id:'2 - 2',
     name:'售后服务部',
```

```
                children:[
                  {
                   id:'2-1-1',
                   name:'反馈 1 组',
                   children:[
                     {
                      id:'2-1-1-1',
                      name:'第一梯队'
                     },
                     {
                      id:'2-1-1-2',
                      name:'第二梯队'
                     }
                    ]
                   },
                   {
                    id:'2-1-2',
                    name:'反馈 2 组'
                   }
                  ]
                 }
                ]
               }
              ]
             }
            }
        }).$mount('#app')
    </script>
  </body>
</html>
```

通过 p-tree 内部引用自己的方式，解决了非对称树形结构的数据展示，这个模式与 JavaScript 中的递归思想完全一致，通过运行过程中链式触发，解决未知深度数据的遍历。递归组件案例的运行结果如图 3-18 所示。

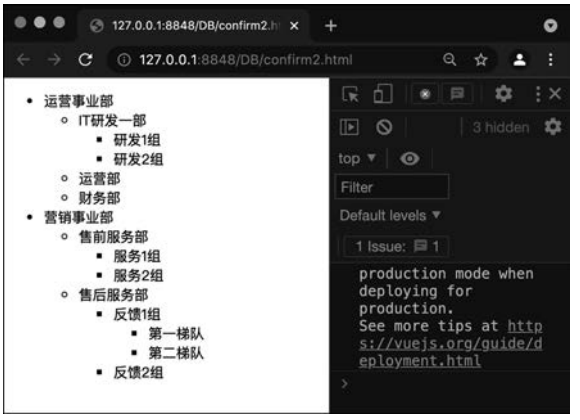


图 3-18　递归组件案例的效果图

完成了递归组件的定义，已经可以实现部门组成的视图展示，根据自定义表格的封装经验，树形组件的内部数据在未来也有很大的定制开发需求，所以每个树形节点的数据和样式都应该是开发者可以自定义的，这里就涉及递归组件中的作用域插槽应用。接下来在树形组件中定义作用域插槽，代码如下：

```
<!-- 第 3 章 在树形组件中定义作用域插槽的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-tree :data="data">
        <!-- 通过作用域插槽自定义树形组件的节点内容 -->
        <template v-slot:node="{row}">
          名称：{{row.name}}
        </template>
      </p-tree>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-tree',{
        props:{
          data:{
            type:Array,
            default(){
              return []
            }
          }
        },
        template:`
          <ul>
            <li v-for="item in data" :key="data.id">
              <!-- 使用作用域插槽实现内容分发和值的传递 -->
              <slot name="node" :row="item">
                {{item.name}}
              </slot>
              <!-- 递归组件的用法是当组件内部需要未知层深的内容展示时，可以直接将组件
本身作为渲染内容在组件内部调用 -->
              <!-- 在组件渲染时，p-tree 会自动识别当前节点是否有后代，如果有后代就会触发
链式渲染 -->
              <p-tree v-if="item.children" :data="item.children"></p-tree>
            </li>
          </ul>
        `
```

```
})
new Vue({
  data(){
    return {
      //初始树形数据
      data:[
        {
          id:'1',
          name:'运营事业部',
          children:[
            {
             id:'1-1',
             name:'IT 研发一部',
             children:[
               {
                id:'1-1-1',
                name:'研发 1 组'
               },
               {
                id:'1-1-2',
                name:'研发 2 组'
               }
             ]
            },
            {
              id:'1-2',
              name:'运营部'
            },
            {
              id:'1-3',
              name:'财务部'
            }
          ]
        },
        {
          id:'2',
          name:'营销事业部',
          children:[
            {
              id:'2-1',
              name:'售前服务部',
              children:[
                {
                 id:'2-1-1',
                 name:'服务 1 组'
                },
```

```
                    {
                     id:'2-1-2',
                     name:'服务 2 组'
                    }
                  ]
                },
                {
                  id:'2 - 2',
                  name:'售后服务部',
                  children:[
                    {
                      id:'2-1-1',
                      name:'反馈 1 组',
                      children:[
                        {
                         id:'2-1-1-1',
                         name:'第一梯队'
                        },
                        {
                         id:'2-1-1-2',
                         name:'第二梯队'
                        }
                      ]
                    },
                    {
                      id:'2-1-2',
                      name:'反馈 2 组'
                    }
                  ]
                }
              ]
            }
          ]
        }
      }
    }). $mount('#app')
  </script>
 </body>
</html>
```

使用作用域插槽改造组件后，可以针对树的每个节点做自定义的改造，但本案例运行结果却不尽人意。具体问题如图 3-19 所示。

按照当前方式改造后，只有树的第一层节点能被插槽分发的内容渲染。这是由于在递归组件中继续引用了组件本身，而案例代码并没有对组件内部引用的组件本身做插槽数据分发，所以造成了只有第一层节点可以应用插槽数据的结果。接下来要将属性组件的内部插槽体系做进一步的改造，最终完成递归组件，代码如下：

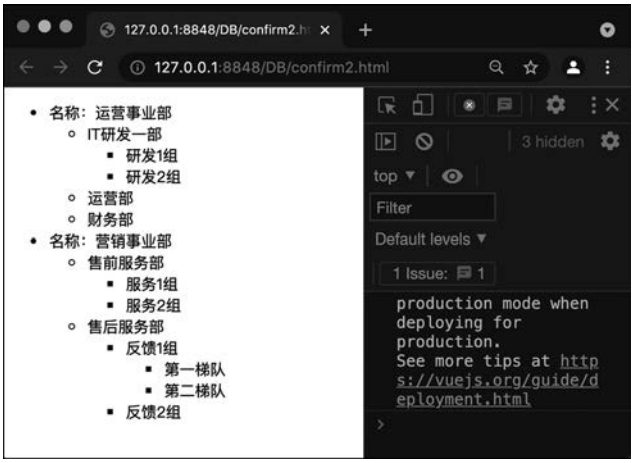


图 3-19　树形组件追加作用域插槽的效果图

```
<!-- 第 3 章 最终递归组件的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="app">
      <p-tree :data="data">
        <!-- 通过作用域插槽自定义树形组件的节点内容 -->
        <template v-slot:node="{row}">
          名称：{{row.name}}
        </template>
      </p-tree>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('p-tree',{
        props:{
          data:{
            type:Array,
            default(){
              return []
            }
          }
        },
        template:`
          <ul>
```

```
            <li v-for="item in data" :key="data.id">
              <!-- 使用作用域插槽实现内容分发和值的传递 -->
              <slot name="node" :row="item">
                {{item.name}}
              </slot>
              <!-- 递归组件的用法是当组件内部需要未知层深的内容展示时,可以直接将组件本身作为渲染内容在组件内部调用 -->
              <!-- 在组件渲染时,p-tree会自动识别当前节点是否有后代,如果有后代就会触发链式渲染 -->
              <p-tree v-if="item.children" :data="item.children">
                <!-- 通过向递归组件中传入template实现插槽的定制分发 -->
                <template v-slot:node="{row}">
                 <!-- 将递归组件外部传入的内容通过slot获得并传入子组件内,实现对任何层级的数据都可以进行高可定制 -->
                  <slot name="node" :row="row">
                    {{row.name}}
                  </slot>
                </template>
              <p-tree>
            </li>
          </ul>
        `
      })
      new Vue({
        data(){
          return {
            //初始树形数据
            data:[
              {
                id:'1',
                name:'运营事业部',
                children:[
                  {
                    id:'1-1',
                    name:'IT研发一部',
                    children:[
                      {
                        id:'1-1-1',
                        name:'研发1组'
                      },
                      {
                        id:'1-1-2',
                        name:'研发2组'
                      }
                    ]
                  },
                  {
                    id:'1-2',
                    name:'运营部'
```

```
    },
    {
      id:'1-3',
      name:'财务部'
    }
  ]
},
{
  id:'2',
  name:'营销事业部',
  children:[
    {
      id:'2-1',
      name:'售前服务部',
      children:[
        {
          id:'2-1-1',
          name:'服务 1 组'
        },
        {
          id:'2-1-2',
          name:'服务 2 组'
        }
      ]
    },
    {
      id:'2 - 2',
      name:'售后服务部',
      children:[
        {
          id:'2-1-1',
          name:'反馈 1 组',
          children:[
            {
              id:'2-1-1-1',
              name:'第一梯队'
            },
            {
              id:'2-1-1-2',
              name:'第二梯队'
            }
          ]
        },
        {
          id:'2-1-2',
          name:'反馈 2 组'
        }
      ]
    }
```

```
                ]
              }
            ]
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

通过最终改造，树形组件可以仅通过一个< template >标签，来高度定制其内部任何子节点的样式和数据内容。树形组件最终改造的结果如图 3-20 所示。

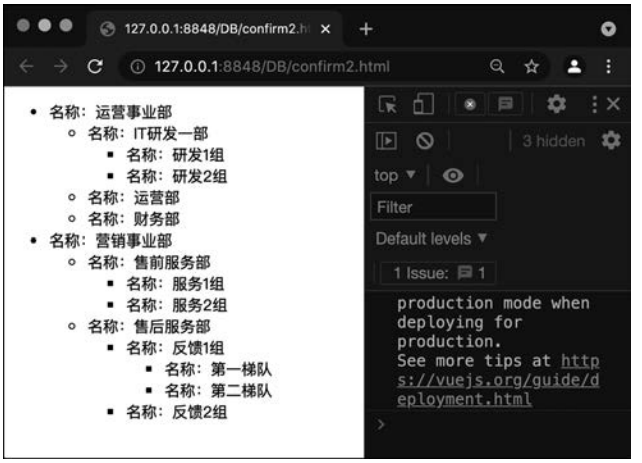


图 3-20　最终递归组件的效果图

3.3　Vue 的动态组件介绍

动态组件是 Vue 组件系统中最具特色的功能，也是单页面应用的核心解决方案。通过动态组件技术，可以实现单页面应用的路由系统。Vue 对动态组件提供了一套完整的解决方案，Vue 在初始化阶段便注册了大量的内置组件，其中就包括< component >组件。

1. < component >组件介绍

Vue 框架内置了< component >组件。在 Vue 初始化完成后，任何组件内部都可以直接应用< component >组件，< component >组件本身没有任何样式和功能，它相当于组件容器，其作用是加载开发者在项目中定义的自定义组件。< component >中包含 is 属性，它的值可以是字符串或自定义组件的 JavaScript 对象。当值为字符串时，只要 is 的值与任何运行环境中已注册的组件名称相同，< component >就会变成对应的组件并渲染其内容。当值为自定义组件的 JavaScript 对象时，< component >会自动渲染该组件内容。动态组件的实际应

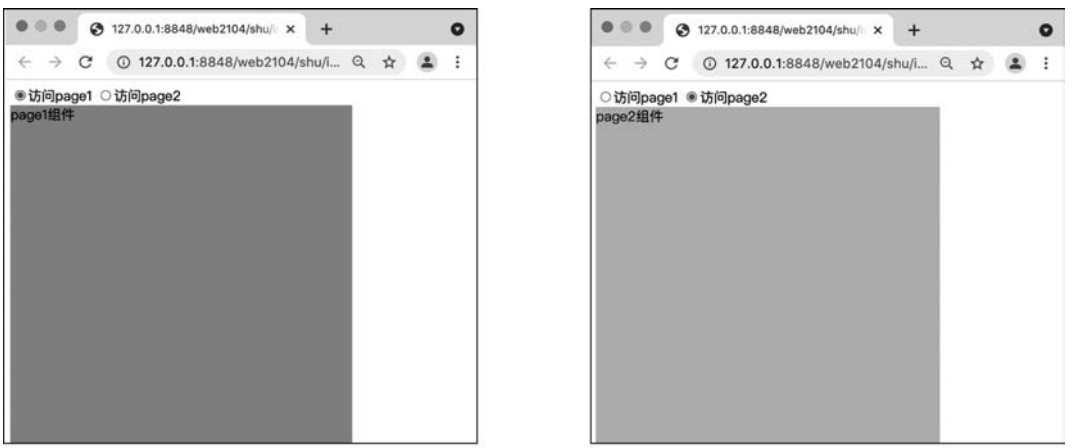
用案例,代码如下:

```
<!-- 第 3 章 动态组件实际应用的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .page{
        width: 400px;
        height: 400px;
      }
      .page1{
        background-color: #42B983;
      }
      .page2{
        background-color: aquamarine;
      }
    </style>
  </head>
  <body>
    <div id="app">
    <!-- 使用 pageName 绑定单选按钮,用来动态切换 pageName 的结果 -->
      <label for="page1">
        <input id="page1" type="radio" v-model="pageName" value="page1">访问 page1
      </label>
      <label for="page2">
        <input id="page2" type="radio" v-model="pageName" value="page2">访问 page2
      </label>
      <!-- 将 pageName 的值绑定在 component 组件上,component 会根据 pageName 的值匹配对应的
组件 -->
      <component :is="pageName"></component>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('page1',{
        template:`
          <div class="page page1">
            page1 组件
          </div>
        `
      })
      Vue.component('page2',{
        template:`
          <div class="page page2">
            page2 组件
          </div>
```

```
        `
      })
      new Vue({
        data(){
          return {
            //通过字符串匹配组件名称,让动态组件可以自动识别
            pageName:'page1'
          }
        }
      }). $mount('#app')

    </script>
  </body>
</html>
```

<component>标签会自动匹配 pageName 对应的值，当 pageName 的值为 page1 时，动态组件展示的就是<page1>标签的结果，当 pageName 的值为 page2 时，动态组件展示的就是<page2>标签的结果。该案例的运行结果如图 3-21 所示。



(a) pageName的值为page1时　　(b) pageName的值为page2时

图 3-21　动态组件实际案例的效果图

2. 动态组件的生命周期

在动态组件切换时，会触发组件自身的生命周期执行，Vue 中的所有自定义组件都具备完整独立的生命周期系统，在从组件加载到组件销毁的过程中，组件内部的生命周期会自动执行。当某一个组件被动态组件渲染时，该组件会触发 beforeCreate()、created()、beforeMount()及 mounted()四个钩子函数。当动态组件切换到其他组件时，该组件会自动触发销毁流程，执行 beforeDestroy()和 destroyed()钩子函数。

根据组件切换的特性，当使用动态组件进行视图切换时，如果想实现类似浏览器窗口切换的效果，则会触发不好的用户体验。浏览器在窗口切换时，用户在不同网页中输入的内容

及位置信息都会记录在独立的窗口内，这样在下一次切换回该网页时，用户可以接着上次的浏览轨迹继续浏览网页。使用动态组件实现的页面切换，仅能实现页面切换功能，Vue 的生命周期规则会导致每次切换组件时组件都会重新创建一次，所以上一次访问该组件的浏览记录和操作记录会全部丢失。根据当前的分析，首先需要了解动态组件的生命周期执行过程，动态组件生命周期的基本案例，代码如下：

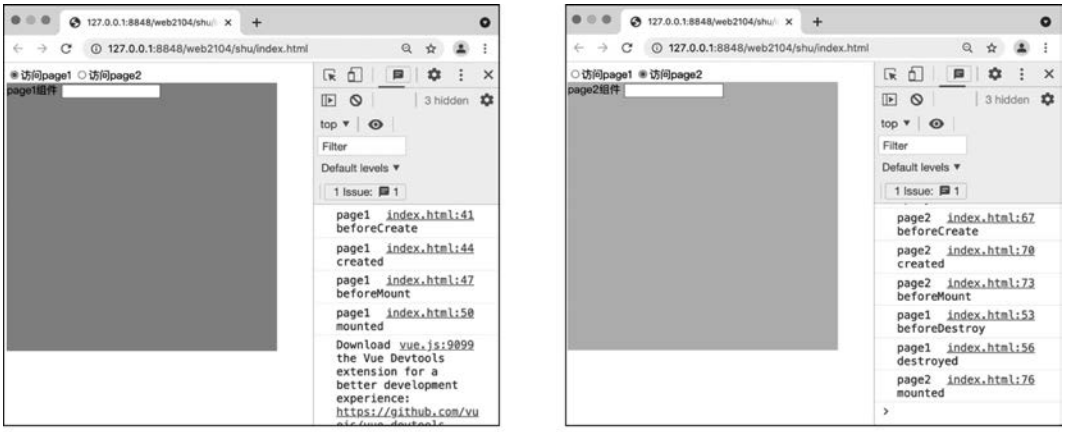
```
<!-- 第 3 章 动态组件生命周期的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .page{
        width: 400px;
        height: 400px;
      }
      .page1{
        background-color: #42B983;
      }
      .page2{
        background-color: aquamarine;
      }
    </style>
  </head>
  <body>
    <div id="app">
    <!-- 使用 pageName 绑定单选按钮，用来动态切换 pageName 的结果 -->
      <label for="page1">
        <input id="page1" type="radio" v-model="pageName" value="page1">访问 page1
      </label>
      <label for="page2">
        <input id="page2" type="radio" v-model="pageName" value="page2">访问 page2
      </label>
      <!-- 将 pageName 的值绑定在 component 组件上，component 会根据 pageName 的值匹配对应的
组件 -->
      <component :is="pageName"></component>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('page1',{
        template:`
          <div class="page page1">
            page1 组件
            <input/>
          </div>
```

```
    `,
    beforeCreate() {
      console.log('page1 beforeCreate')
    },
    created(){
      console.log('page1 created')
    },
    beforeMount(){
      console.log('page1 beforeMount')
    },
    mounted(){
      console.log('page1 mounted')
    },
    beforeDestroy(){
      console.log('page1 beforeDestroy')
    },
    destroyed(){
      console.log('page1 destroyed')
    }
  })
  Vue.component('page2',{
    template:`
      <div class="page page2">
        page2 组件
        <input/>
      </div>
    `,
    beforeCreate() {
      console.log('page2 beforeCreate')
    },
    created(){
      console.log('page2 created')
    },
    beforeMount(){
      console.log('page2 beforeMount')
    },
    mounted(){
      console.log('page2 mounted')
    },
    beforeDestroy(){
      console.log('page2 beforeDestroy')
    },
    destroyed(){
      console.log('page2 destroyed')
    }
  })
  new Vue({
    data(){
      return {
```

```
            //通过字符串匹配组件名称,让动态组件可以自动识别
            pageName:'page1'
          }
        }
      }).$mount('#app')

    </script>
  </body>
</html>
```

动态组件在初次加载时,会执行<page1>组件的生命周期,动态组件从<page1>切换到<page2>时,在触发<page2>组件加载的过程中,会同时触发<page1>组件的销毁流程,这导致无论在组件间切换几次,每次展示的自定义组件都是重新创建的组件,所以在组件切换时,无论是<page1>还是<page2>,都无法保留其内部输入框中输入的内容。Vue之所以这样设计,是为了节省程序在浏览器中占用的内存。组件化可以让视图代码在可维护性上取得更大的优势,由于组件代码都是通过JavaScript执行的,所以所有的视图组件都需要在浏览器内存中进行管理。开发一个完整的项目,会定义大量的组件,如果每个组件创建后不执行销毁操作,则会导致浏览器内部保存大量当前不展示的页面数据,随着时间的推移,触发内存溢出的可能性会变得很高,所以在动态组件系统中,每当<component>加载一个新的组件时,都是将上一次的组件从内存中销毁,以此来释放内存。组件生命周期案例运行的结果如图3-22所示。



(a) <page1>组件加载时　　(b) 从<page1>切换到<page2>组件时

图3-22　动态组件生命周期案例的效果图

3.4　Vue的组件缓存

通过3.3节的学习,了解了组件的生命周期及加载流程,根据当前的情况还无法实现保留每个组件的访问状态。想要实现在Vue的视图切换时,保存组件的访问和操作状态,就

需要借助组件缓存技术实现。组件缓存技术需要借助 Vue 的内置组件< keep-alive >，凡是被< keep-alive >包裹的< component >组件，在第一次加载后，便会被保存在浏览器的内存中，在组件切换时便不会将这些被缓存的组件销毁，所以被缓存后的组件无论如何切换，都不会再触发组件的生命周期钩子函数，这样便实现了保存访问和操作记录。组件缓存的实际案例，代码如下：

```
<!-- 第 3 章 组件缓存的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .page{
        width: 400px;
        height: 400px;
      }
      .page1{
        background-color: #42B983;
      }
      .page2{
        background-color: aquamarine;
      }
    </style>
  </head>
  <body>
    <div id="app">
    <!-- 使用 pageName 绑定单选按钮，用来动态切换 pageName 的结果 -->
      <label for="page1">
        <input id="page1" type="radio" v-model="pageName" value="page1">访问 page1
      </label>
      <label for="page2">
        <input id="page2" type="radio" v-model="pageName" value="page2">访问 page2
      </label>
      <!-- 通过 keep-alive 直接嵌套 component 动态组件后，其内部的组件在切换时便不会再销毁 -->
      <keep-alive>
        <component :is="pageName"></component>
      </keep-alive>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript" charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('page1',{
        template:`
          <div class="page page1">
            page1 组件
```

```
        <input/>
      </div>
    `,
    beforeCreate() {
      console.log('page1 beforeCreate')
    },
    created(){
      console.log('page1 created')
    },
    beforeMount(){
      console.log('page1 beforeMount')
    },
    mounted(){
      console.log('page1 mounted')
    },
    beforeDestroy(){
      console.log('page1 beforeDestroy')
    },
    destroyed(){
      console.log('page1 destroyed')
    }
})
Vue.component('page2',{
  template:`
    <div class="page page2">
      page2 组件
      <input/>
    </div>
  `,
  beforeCreate() {
    console.log('page2 beforeCreate')
  },
  created(){
    console.log('page2 created')
  },
  beforeMount(){
    console.log('page2 beforeMount')
  },
  mounted(){
    console.log('page2 mounted')
  },
  beforeDestroy(){
    console.log('page2 beforeDestroy')
  },
  destroyed(){
    console.log('page2 destroyed')
  }
})
new Vue({
```

```
        data(){
          return {
            //通过字符串匹配组件名称,让动态组件可以自动识别
            pageName:'page1'
          }
        }
      }). $mount('#app')

    </script>
  </body>
</html>
```

运行案例代码后,在< page1 >组件的输入框中输入 123,在< page2 >组件的输入框中输入 456。切换两个组件的展示并观察控制台输出,会发现第一次访问< page1 >或< page2 >时都会触发两个组件的 beforeCreate()、created()、beforeUpdate()及 updated()四个钩子函数,组件的销毁函数不会触发。在这之后继续切换< page1 >和< page2 >时,组件便不会触发任何生命周期钩子函数,< page1 >和< page2 >两个组件内的输入框中输入的内容也不会消失。

以上便是组件缓存技术的实现,缓存技术可以让自定义组件在做视图切换时,保存用户的操作行为和数据,来方便用户在视图切换中有更好的操作体验,但是组件缓存会增加浏览器的内存负担。视图组件对象在内存中持续增加会大量占用 JavaScript 的堆内存空间,一旦超过界限,就会触发内存溢出的错误。当然,Vue 的作者已经考虑到这些问题,所以为了提高组件性能,< keep-alive >内置了以下几个属性:

1) max 属性

max 属性用来控制< keep-alive >组件最多可以缓存多少个组件实例,一旦这个数字达到了阈值,在新实例被创建之前,已缓存组件中最久没有被访问的实例会被销毁。具体的使用方式,代码如下:

```
<keep-alive :max="10">
  <component :is="view"></component>
</keep-alive>
```

2) include 属性

include 属性通常使用数组类型,也可以使用字符串类型,它的作用是选择性缓存自定义组件实例对象。在 include 中包含的组件会自动被缓存,其他不在数组中的视图组件并不会被缓存。选择性缓存可以让开发者在视图组件缓存规划上有更多的选择,用以提高内存利用率和访问体验。选择性缓存的案例,代码如下:

```
<!-- 第 3 章 include 属性的代码案例 -->
<!-- 逗号分隔字符串 -->
<keep-alive include="a,b">
```

```
  <component :is="view"></component>
</keep-alive>

<!-- 正则表达式 (使用 `v-bind`) -->
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- 数组 (使用 `v-bind`) -->
<keep-alive :include="['a', 'b']">
  <component :is="view"></component>
</keep-alive>
```

3) exclude 属性

exclude 属性与 include 属性的使用方式完全一样。不同的是，exclude 代表选择性不缓存，只有定义在 exclude 中的组件才会被< keep-alive >排除，开发者在项目的特殊情况中，可主动排除不想缓存的组件。选择性不缓存的案例，代码如下：

```
<!-- 第 3 章 exclude 属性的代码案例 -->
<!-- 逗号分隔字符串 -->
<keep-alive :exclude="a,b">
  <component :is="view"></component>
</keep-alive>

<!-- 正则表达式 (使用 `v-bind`) -->
<keep-alive :exclude="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- 数组 (使用 `v-bind`) -->
<keep-alive :exclude="['a', 'b']">
  <component :is="view"></component>
</keep-alive>
```

< keep-alive >的优缺点已经介绍得差不多了，接下来需要介绍的是缓存后的组件访问问题。使用组件缓存技术后，视图组件的实例便会被暂存在内存中，切换视图组件也不会触发自定义组件的销毁，所以一旦所有要切换的组件全部被访问过后，再次访问任何一个组件也不会触发其内部的初始化钩子函数。这就导致在切换组件时，应用程序并不知道被缓存的组件何时被访问。为此，Vue 提供了两个补充的生命周期钩子函数，用来在被缓存的组件中使用，具体用法如下。

1) activated()

activated()函数只在被缓存的组件中有效，通过该函数开发者可以完全掌控被缓存后的组件，每当组件被访问时，便会触发 activated()执行。activated()函数的应用场景，代码如下：

```
<!-- 第 3 章 activated()函数应用场景的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .page{
        width: 400px;
        height: 400px;
      }
      .page1{
        background-color: #42B983;
      }
      .page2{
        background-color: aquamarine;
      }
    </style>
  </head>
  <body>
    <div id="app">
    <!-- 使用 pageName 绑定单选按钮,用来动态切换 pageName 的结果 -->
      <label for="page1">
        <input id="page1" type="radio" v-model="pageName" value="page1">访问 page1
      </label>
      <label for="page2">
        <input id="page2" type="radio" v-model="pageName" value="page2">访问 page2
      </label>
      <!-- 通过 keep-alive 直接嵌套 component 动态组件后,其内部的组件在切换时便不会再被
销毁 -->
      <keep-alive>
        <component :is="pageName"></component>
      </keep-alive>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      Vue.component('page1',{
        template:`
          <div class="page page1">
            page1 组件
            <input/>
            <br/>
            {{visitTime}}
          </div>
        `,
        data(){
          return {
            visitTime:new Date()
```

```
    }
  },
  beforeCreate() {
    console.log('page1 beforeCreate')
  },
  created(){
    console.log('page1 created')
  },
  beforeMount(){
    console.log('page1 beforeMount')
  },
  mounted(){
    console.log('page1 mounted')
  },
  beforeDestroy(){
    console.log('page1 beforeDestroy')
  },
  destroyed(){
    console.log('page1 destroyed')
  },
  activated(){
    console.log('page1 被缓存后,被访问')
    this.visitTime=new Date()
  }
})
Vue.component('page2',{
  template:`
    <div class="page page2">
      page2 组件
      <input/>
      <br/>
      {{visitTime}}
    </div>
  `,
  data(){
    return {
      visitTime:new Date()
    }
  },
  beforeCreate() {
    console.log('page2 beforeCreate')
  },
  created(){
    console.log('page2 created')
  },
  beforeMount(){
    console.log('page2 beforeMount')
  },
  mounted(){
```

```
            console.log('page2 mounted')
          },
          beforeDestroy(){
            console.log('page2 beforeDestroy')
          },
          destroyed(){
            console.log('page2 destroyed')
          },
          activated(){
            console.log('page2 被缓存后,被访问')
            this.visitTime=new Date()
          }
        })
        new Vue({
          data(){
           return {
             //通过字符串匹配组件名称,让动态组件可以自动识别
             pageName:'page1'
           }
          }
        }).$mount('#app')

      </script>
    </body>
  </html>
```

被缓存的<page1>和<page2>组件在后续的访问中不会被重新初始化,所以无法更新visitTime的值。想要更新组件的访问时间,需要在activated()函数中手动进行初始化,这样可以保证用户无论访问任何组件,都可以记录用户的实时访问时间。该案例的运行结果如图3-23所示。

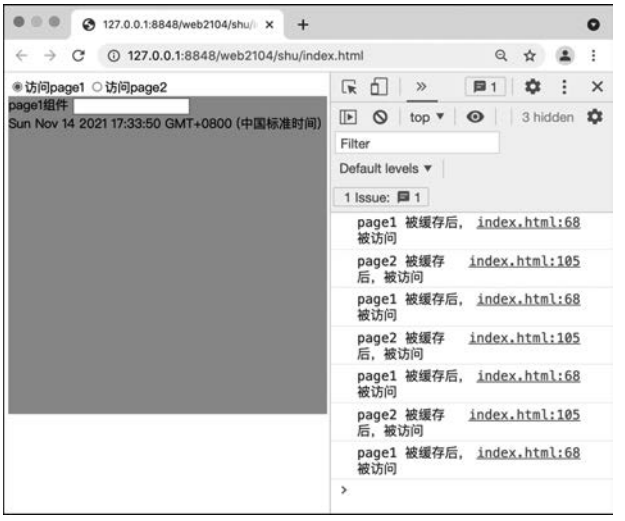


图3-23 activated()钩子函数应用案例的效果图

2）deactivated()

deactivated()函数只在被缓存的组件中有效，当被缓存的组件从缓存列表中移除时，deactivated()函数便会被执行，用来通知开发者该组件已经不再被缓存。

3.5　Vue 的组件过渡

在原生 Web 开发场景中，开发者经常需要来对视图组件进行过渡动画的定义，配合 JavaScript 和 CSS 语言可以快速为 HTML 标签定义好看的动画和交互效果。在 Vue 框架中，配合组件化思想开发时，仅使用原生的过渡动画方式很难写出高效好看的代码，所以 Vue 框架提供了内置的过渡解决方案。

1. ＜transition＞组件介绍

Vue 框架内置了＜transition＞组件，用来管理自定义组件的过渡行为，＜transition＞可以拦截嵌套在它内部的组件的进入和离开状态，在不同的状态切换时补偿过渡动画。＜transition＞内部可以被管理状态变更的组件包括以下几种：

（1）使用 v-if 切换显示和隐藏的组件。

（2）使用 v-show 切换显示和隐藏的组件。

（3）切换过程的动态组件。

（4）根节点的组件。

接下来通过一个简单的案例学习组件过渡的实际应用，代码如下：

```
<!-- 第 3 章 组件过渡实际应用的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      /* 过渡动画的样式状态 */
      .fade-enter{
        opacity: 0;
      }
      .fade-enter-active{
        transition: opacity .5s;
      }
      .fade-enter-to{
        opacity: 1;
      }
      .fade-leave{
        opacity: 1;
      }
      .fade-leave-active{
        transition: opacity .5s;
```

```
        }
        .fade-leave-to{
          opacity: 0;
        }
      </style>
    </head>
    <body>
      <div id="app">
        <button @click="handleClick">执行过渡</button>
        <!-- 使用 transition 嵌套 p 标签捕捉其过渡状态,name 属性为过渡样式的名称 -->
        <transition name="fade">
          <!-- 使用 v-if 切换元素的显示和隐藏 -->
          <p v-if="show">
            我是一段文字
          </p>
        </transition>
      </div>
      <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
      <script type="text/javascript">

        new Vue({
          data(){
            return {
              //开关属性
              show:false
            }
          },
          methods:{
            //单击按钮时切换 show 属性的结果
            handleClick(){
              this.show=!this.show
            }
          }
        }).$mount('#app')

      </script>
    </body>
  </html>
```

当单击按钮时,<transition>会自动感知到<p>标签的显示和隐藏的过程,为其在进入和离开时补偿 CSS 交互效果,实现淡入和淡出的动画。在使用<transition>体系时,并不会打破 Vue 默认的状态切换流程。

当插入或删除包含在<transition>组件中的元素时,Vue 将会做以下处理:

(1) 自动嗅探目标元素是否应用了 CSS 过渡或动画,如果应用了,则在恰当的时机添加/删除 CSS 类名。

(2) 如果过渡组件提供了 JavaScript 钩子函数,则这些钩子函数将在恰当的时机被调用。

（3）如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡动画，则 DOM 操作会在下一帧中立即执行，不会产生任何过渡或动画效果。

2. 过渡的类名

在进入或离开的过渡中，被< transition >组件包裹的元素会有 6 个 class 切换的过程。

1）v-enter

定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。

2）v-enter-active

定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的时间、延迟和曲线函数。

3）v-enter-to

2.1.8 版及以上定义进入过渡的结束状态。在元素被插入之后的下一帧生效（与此同时 v-enter 被移除），在过渡/动画完成之后移除。

4）v-leave

定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除。

5）v-leave-active

定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的时间、延迟和曲线函数。

6）v-leave-to

2.1.8 版及以上定义离开过渡的结束状态。在离开过渡被触发之后的下一帧生效（与此同时 v-leave 被删除），在过渡/动画完成之后移除。

整个状态过渡期，< transition >标签会自动按照顺序对元素应用 6 个 class 名称，这些 class 名称被应用和执行的顺序如图 3-24 所示。

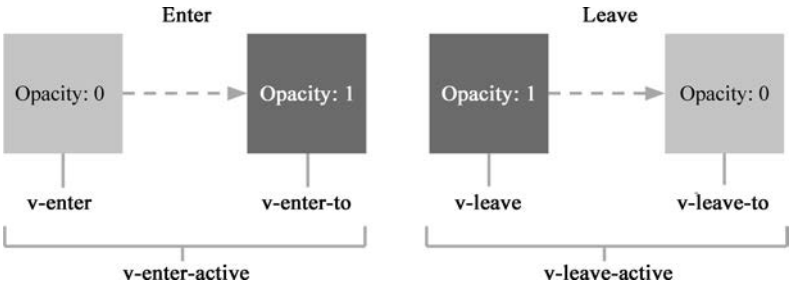


图 3-24　过渡名称应用顺序的效果图

关于过渡名称，如果开发者在使用< transition >组件时没有传递任何参数，则 6 个过渡名称默认以 v-开头，如果在< transition >组件中定义了 name 属性，如在过渡的实际应用中对< transition >标签设置了 name="fade"，则该组件的过渡名称以 fade-开头。v-enter、v-enter-to、v-leave 及 v-leave-to 四个名称为分别进入和离开状态的起点和终点，所以在

v-enter 中定义元素进入前的样式，在 v-enter-to 中定义元素进入后的最终样式，在 v-leave 中定义元素离开前的样式，在 v-leave-to 中定义元素离开后的最终样式。v-enter-active 和 v-leave-active 分别为进入和离开过程中持续应用的样式，所以在过渡中将 CSS 样式的 transition 属性应用在包含-active 的类名中。

3. animation 实现过渡

使用 CSS 样式定义过渡状态，需要定义 6 个类名，虽然这些样式可以进一步简写，但 6 种状态的切换是通过 transition 属性来控制元素过渡的，这导致过渡只能实现从起点到终点的两种状态的线性变化。Vue 在过渡中支持在类名中定义 CSS 动画，实现过渡效果的复杂化。使用 CSS 动画后，原有的 6 个样式名只需要定义两个，CSS 动画实现的过渡，代码如下：

```
<!-- 第 3 章 CSS 动画实现过渡的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      /* 过渡动画的样式状态 */
      .fade-enter-active{
        animation-name: scale-in-out;
        animation-duration: 0.5s ;
      }
      .fade-leave-active{
        animation-name: scale-in-out;
        animation-duration: 0.5s ;
        animation-direction: reverse;
      }
      @keyframes scale-in-out{
        0%{
          transform: scale(0);
        }
        50%{
          transform: scale(1.5);
        }
        100%{
          transform: scale(1);
        }
      }
    </style>
  </head>
  <body>
    <div id="app">
      <button @click="handleClick">执行过渡</button>
      <!-- 使用 transition 嵌套 p 标签捕捉其过渡状态，name 属性为过渡样式的名称 -->
```

```
        <transition name="fade">
          <!-- 使用 v-if 切换元素的显示和隐藏 -->
          <p v-if="show" style="display: inline-block;">
            这是一段文字
          </p>
        </transition>
      </div>
      <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
      <script type="text/javascript">

        new Vue({
          data(){
            return {
              //开关属性
              show:false
            }
          },
          methods:{
            //单击按钮时切换 show 属性的结果
            handleClick(){
              this.show=!this.show
            }
          }
        }).$mount('#app')

      </script>
    </body>
  </html>
```

使用 animation 应用关键帧动画后，只需配合 v-enter-active 及 v-leave-active 两个样式名称便可以实现过渡流程。利用动画可以倒序播放的特点，只需定义一套关键帧，便可以实现进入和离开的两套过渡动作复用。关键帧可以逐帧定义动画的具体动作，所以 animation 动画可以让过渡效果变得更加复杂和自由。

4. 自定义过渡类名

在< transition >组件中，可以通过以下属性来自定义过渡类名：

1）enter-class

2）enter-active-class

3）enter-to-class（2.1.8+）

4）leave-class

5）leave-active-class

6）leave-to-class（2.1.8+）

它们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库（如 Animate.css）结合使用十分有用。自定义类名的应用案例，代码如下：

```
<!-- 第3章 自定义类名的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <link href="https://cdn.jsdelivr.net/npm/animate.css@3.5.1" rel="stylesheet" type=
"text/css">
  </head>
  <body>
    <div id="app">
      <button @click="show=!show">
        Toggle render
      </button>
      <transition
        name="custom-classes-transition"
        enter-active-class="animated tada"
        leave-active-class="animated bounceOutRight"
      >
        <p v-if="show">hello</p>
      </transition>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
    <script type="text/javascript">
      new Vue({
        data(){
          return{
            //开关属性
            show:false
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

5. 过渡的其他使用方式

(1) 同时使用过渡和动画：Vue为了知道过渡是否已完成，必须设置相应的事件监听器。它可以是transitionend或animationend，这取决于给元素应用的CSS规则。如果使用其中任何一种，则Vue能自动识别类型并设置监听，但是，在一些场景中，需要给同一个元素同时设置两种过渡动效，例如animation很快地被触发并完成了，而transition效果还没有结束。在这种情况下就需要使用type attribute并设置animation或transition来明确声明Vue监听的类型。

(2) 显性的过渡持续时间：在很多情况下，Vue可以自动得出过渡效果的完成时机。默认情况下，Vue会等待其在过渡效果的根元素的第1个transitionend或animationend事

件，然而也可以不这样设定，例如精心编排的一系列过渡效果，其中一些嵌套的内部元素，相比于过渡效果的根元素，有延迟或更长的过渡效果。在这种情况下可以用 < transition > 组件上的 duration prop 定制一个显性的过渡持续时间，代码如下：

```
<transition :duration="1000">...</transition>
```

也可以定制进入和移出的持续时间，代码如下：

```
<transition :duration="{ enter: 500, leave: 800 }">...</transition>
```

（3）JavaScript 钩子：< transition >组件上提供了完整的事件系统，以支持 JavaScript 钩子函数驱动过渡执行的代码如下：

```
<!-- 第 3 章 JavaScript 钩子函数驱动过渡执行的代码案例 -->
<!-- HTML 部分写法 -->
<transition
  v-on:before-enter="beforeEnter"
  v-on:enter="enter"
  v-on:after-enter="afterEnter"
  v-on:enter-cancelled="enterCancelled"

  v-on:before-leave="beforeLeave"
  v-on:leave="leave"
  v-on:after-leave="afterLeave"
  v-on:leave-cancelled="leaveCancelled"
>
  <!-- ... -->
</transition>
<!-- JavaScript 部分的写法 -->
//...
methods: {
  //---------
  //进入中
  //---------
  beforeEnter: function (el) {
    //...
  },
  //当与 CSS 结合使用时
  //回调函数 done 是可选的
  enter: function (el, done) {
    //...
    done()
  },
  afterEnter: function (el) {
    //...
  },
  enterCancelled: function (el) {
```

```
    //...
  },

  //---------
  //离开时
  //---------

  beforeLeave: function (el) {
    //...
  },
  //当与 CSS 结合使用时
  //回调函数 done 是可选的
  leave: function (el, done) {
    //...
    done()
  },
  afterLeave: function (el) {
    //...
  },
  //leaveCancelled 只用于 v-show 中
  leaveCancelled: function (el) {
    //...
  }
}
```

(4) 初始渲染的过渡：<transition>组件包含 appear 属性来允许节点在初始化时立即执行过渡动画，代码如下：

```
<transition appear>
  <!-- ... -->
</transition>
```

appear 也支持自定义 CSS 的类名及 JavaScript 的钩子函数，代码如下：

```
<!-- 第 3 章 appear 自定义类名和钩子函数的代码案例 -->
<!-- 自定义 CSS 类名的案例 -->
<transition
  appear
  appear-class="custom-appear-class"
  appear-to-class="custom-appear-to-class" (2.1.8+)
  appear-active-class="custom-appear-active-class"
>
  <!-- ... -->
</transition>
<!-- 自定义 JavaScript 钩子函数的案例 -->
<transition
  appear
  v-on:before-appear="customBeforeAppearHook"
```

```
    v-on:appear="customAppearHook"
    v-on:after-appear="customAfterAppearHook"
    v-on:appear-cancelled="customAppearCancelledHook"
  >
    <!-- ... -->
  </transition>
```

（5）多个元素的过渡：之前讨论的过渡，都是基于一个组件的显示和隐藏进行过渡补偿。当< transition >标签内部嵌套的过渡内容有多个时，例如 v-if 与 v-else 指令同时应用，代码如下：

```
<!-- 第 3 章 v-if 与 v-else 同时应用的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .fade-enter{
        opacity: 0;
      }
      .fade-enter-active,.fade-leave-active{
        transition: all 1s;
      }
      .fade-leave-to{
        opacity: 0;
      }
    </style>
  </head>
  <body>
    <div id="app">
      <button @click="show=!show">
        切换
      </button>
      <div>
        <transition name="fade">
          <!-- 这里必须使用 key 修饰防止视图不更新 -->
          <button v-if="show" key="on">
            show 为 true
          </button>
          <button v-else key="off">
            show 为 false
          </button>
        </transition>
      </div>
    </div>
```

```
    < script src =" https://cdn. jsdelivr. net/npm/vue/dist/vue. js" type =" text/javascript"
charset ="utf-8"></script >
    < script type ="text/javascript">
      new Vue({
        data(){
          return {
            //开关属性
            show:false
          }
        }
      }). $mount('#app')
    </script >
  </body >
</html >
```

运行案例代码，在切换 show 属性的结果时，视图层会根据 show 的值动态切换按钮的展示。当前案例运行结果会出现问题，如图 3-25 所示。

图 3-25　v-if 与 v-else 同时应用案例的效果图

运行案例后，会发现切换过程中，由于使用了 v-if 和 v-else 指令，当前按钮会被从 DOM 树中移除，新的按钮会被添加进 DOM 树中。这个过程被< transition >捕捉到后，当前按钮进入隐藏状态和新按钮进入显示状态会同时进行，这就导致了网页中会同时进行一个按钮正在消失，另一个按钮正在进入的过程。< transition >中提供了 mode 属性来解决多组件切换的问题，为了让两个组件有序地执行动画，可以在< transition >组件中设置 mode="in-out"或 mode="out-in"来切换过渡执行的模式。

1）in-out

in-out 代表先执行新元素的进入动画，新元素进入后再执行当前元素的离开动画。

2）out-in

out-in 代表先执行当前元素的离开动画，当前元素离开后再执行新元素的进入动画。

在多个元素的进入/离开过渡案例中，out-in 模式更加适合。

6. 列表过渡

列表过渡与多元素过渡不同，列表过渡管理了多个元素同时触发的过渡动画，通常与列表渲染同时使用，实现在同一个过渡组件下，对多个元素的进入和离开状态进行管理。列表过渡采用的组件为< transition-group >，不同于< transition >组件，< transition-group >组件在视图中呈现为真实元素，默认为< span >标签，可通过 tag 属性来改变真实呈现的标签类型。列表过渡的案例，代码如下：

```
<!-- 第 3 章 列表过渡的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style type="text/css">
      .list-item{
        display: inline-block;
        margin-right: 15px;
      }
      .slide-enter,.slide-leave-to{
        transform: translate(0px,50px);
        opacity: 0;
      }
      .slide-enter-active,.slide-leave-active{
        transition: all .5s;
      }
    </style>
  </head>
  <body>
    <div id="app">
      <button @click="handleAddRandom">
        随机插入
      </button>
      <button @click="handleRemoveRandom">
        随机删除
      </button>
      <!-- 使用 tag 将 transition-group 设置为 div 标签 -->
      <transition-group name="slide" tag="div">
        <!-- key 不可以绑定为序号，必须绑定为数组的每个元素 -->
        <span v-for="(item,index) in list" :key="item" class="list-item">
          {{item}}
        </span>
      </transition-group>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js" type="text/javascript"
charset="utf-8"></script>
```

```
    <script type="text/javascript">
      new Vue({
        data(){
          return {
            list:[
              1,2,3,4,5,6,7,8,9
            ]
          }
        },
        methods:{
          handleAddRandom(){
            //根据数组获取随机位置
            let randomIndex=parseInt(Math.random()*this.list.length)
            //对数组的指定位置插入新的数字

this.list.splice(randomIndex,1,this.list.length+1,this.list[randomIndex])
          },
          handleRemoveRandom(){
            //获取数组的随机位置
            let randomIndex=parseInt(Math.random()*this.list.length)
            //删除该位置的元素
            this.list.splice(randomIndex,1)
          }
        }

      }).$mount('#app')
    </script>
  </body>
</html>
```

单击“随机插入”按钮，程序会在当前数组内随机获取位置插入新的数字，数字是按照数组的长度动态生成的。单击“随机删除”按钮，程序会在当前数组内随机获取一个位置，并将当前位置的数字删除。两个操作都会触发视图的更新，也会触发列表内的进入和离开过渡，连续单击“随机插入”或“随机删除”按钮时，会触发多个元素同时展示过渡效果，如图 3-26 所示。

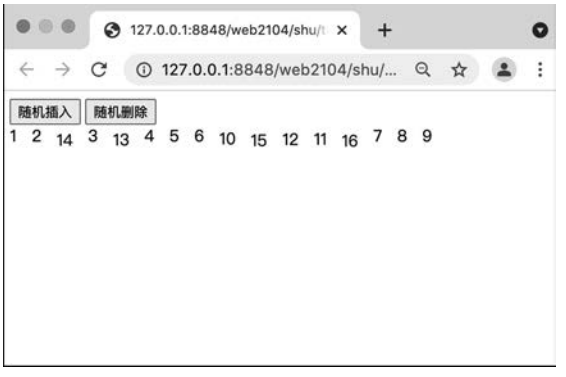


图 3-26　连续单击“随机插入”或“随机删除”按钮的效果图

列表过渡除了可以批量管理多个元素的进入和离开过渡外，还可以触发列表排序的过渡动画管理，当数组内部元素的展示顺序发生变化时，进入和离开的过渡动画无法进行补偿，若想实现排序动画，则需要记录数组变化前每个元素的位置，以及变化后每个元素的位置。Vue 为排序过渡提供了内置动画，Vue 使用了一个叫作 FLIP 的简单的动画队列，使用 transforms 将元素从之前的位置平滑过渡到新的位置。列表排序过渡的案例，代码如下：

```
<!-- 第 3 章 列表排序过渡的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>列表过渡</title>
    <style type="text/css">
      .sort-move{
        transition: all 1s;
      }
    </style>
  </head>
  <body>
    <!-- div id 为 app,它的作用是作为 Vue 框架的渲染容器,只有在这个标签内部编写的 Vue 代码
才能被正确地解析 -->
    <div id="app">
      <button @click="handleSort">随机排序</button>
      <transition-group name="sort">
        <div v-for="item in list" :key="item">
          {{item}}
        </div>
      </transition-group>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      //vm 是 Vue 的实例对象
      var vm=new Vue({
        data(){
          return {
            list:[1,2,3,4,5,6,7,8,9,10]
          }
        },
        methods:{
          handleSort(){
            this.list=this.list.sort(()=>Math.random()>0.5?1:-1)
          }
        }

      }).$mount('#app')
    </script>
  </body>
</html>
```

3.6 其他高级 API 的介绍

除之前章节介绍的特性及编程方式外，Vue 还提供了大量的 API 方便开发者在不同场景中使用，本书着重介绍开发场景常用的 API 及其应用案例，未提及的 API 和应用案例可参考 Vue 的官方文档。本节继续介绍 Vue 框架在开发中实用性比较高的其他框架 API。

1. 响应式数据的更新限制

使用 Vue 框架开发时，开发者可以通过操作 Vue 实例上绑定的数据来触发视图层的更新。Vue 2.x 版本中存在一些特殊情况，会导致开发者即使修改 data 选项中定义的数据也不会触发视图的更新，代码如下：

```
<!-- 第 3 章 Vue 2.x 版本中存在一些特殊情况的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>组件更新策略</title>
  </head>
  <body>
    <!-- div id 为 app,它的作用是作为 Vue 框架的渲染容器,只有在这个标签内部编写的 Vue 代码
才能被正确地解析 -->
    <div id="app">
      用户信息:{{userInfo}}
      <br>
      <button @click="handleUpdateSex">更新 sex 属性</button>
      <button @click="handleUpdateMoney">更新 money 属性</button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      var vm=new Vue({
        data(){
          return {
            userInfo:{
              username:'张三',
              age:18,
              sex:'男'
            }
          }
        },
        methods:{
          handleUpdateSex(){
            this.userInfo.sex='女'
          },
          handleUpdateMoney(){
            this.userInfo.money=10000
```

```
            }
          }
        }).$mount('#app')
      </script>
    </body>
  </html>
```

运行代码后会发现，单击“更新 sex 属性”按钮后，视图中的 sex 属性的值会变成女。单击“更新 money 属性”按钮后，视图中并不会发生任何变化。参考代码中的属性修改方式会发现，两个按钮的代码逻辑是相同的，都是通过 this.userInfo.属性名来修改属性值的。这个案例暴露了 Vue 2.x 在响应式数据更新机制上的瓶颈，具体原因可以参考官方文档中的描述：“Vue 实例的数据对象。Vue 会递归地把 data 的 property 转换为 getter/setter，从而让 data 的 property 能够响应数据变化。对象必须是纯粹的对象（含有零个或多个的 key-value 对）：浏览器 API 创建的原生对象，原型上的 property 会被忽略。大概来讲，data 应该只能是数据，不推荐观察拥有状态行为的对象。一旦观察过，你就无法在根数据对象上添加响应式 property，因此推荐在创建实例之前，就声明所有的根级响应式 property。”

由官方描述得知，在 Vue 2.x 版本中，定义在 data 选项中的属性应提前声明好要使用的属性，否则 Vue 框架无法追踪后添加属性的变化，这种状况很大程度地限制了开发场景的灵活性。对象中初始化的属性与未初始化的属性在 Vue 实例中的描述有所差别，如图 3-27 所示。

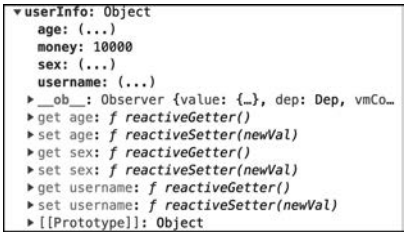


图 3-27　初始化的属性与未初始化的属性差别的效果图

Vue 对象中初始化的对象及其属性，被绑定到 Vue 实例后，会在对象内部存在对应的 set()和 get()函数，当属性的值变化时会直接触发对应属性的 set()和 get()函数，而后创建的 money 属性并没有经过 Vue 对象初始化的过程，所以该属性在对象内并没有对应的 set()和 get()函数，这就是后创建属性不能更新的原因。

2. 解决属性更新问题

直接操作对象中未初始化的属性时，Vue 框架无法感知，所以无法执行视图部分的更新。虽然 Vue 框架推荐开发者应提前声明对象未来要使用的所有属性，但这种方式仍然不适合一些特殊的开发场景，所以 Vue 框架为此提供了其他方式，来保证后创建的对象属性能具备视图更新的能力。Vue 提供了两种解决方案，用以在视图更新上提供补偿措施。

1）$set()函数

$set()函数是 Vue 每个组件的实例对象上自带的一个函数，该函数可以为对象未创建的属性赋值，并触发视图层的更新。$set()的使用方式，代码如下：

```
实例对象.$set(对象,赋值的属性,对应属性的值)
```

使用 $set()实现属性更新的案例,代码如下:

```
<!-- 第 3 章 $set()实现属性更新的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>组件更新策略</title>
  </head>
  <body>
    <!-- div id 为 app,它的作用是作为 Vue 框架的渲染容器,只有在这个标签内部编写的 Vue 代码
才能被正确地解析 -->
    <div id="app">
      用户信息:{{userInfo}}
      <br>
      <button @click="handleUpdateSex">更新 sex 属性</button>
      <button @click="handleUpdateMoney">更新 money 属性</button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      var vm=new Vue({
        data(){
          return {
            userInfo:{
              username:'张三',
              age:18,
              sex:'男'
            }
          }
        },
        methods:{
          handleUpdateSex(){
            this.userInfo.sex='女'
          },
          handleUpdateMoney(){
            //this.$set(this.userInfo,'money',10000)
            //用来将 userInfo 对象的 money 属性的值更新
            this.$set(this.userInfo,'money',10000)
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

2) $forceUpdate()函数

$forceUpdate()函数是 Vue 框架提供的强制更新视图的 API,该函数存在于任何组件的实例对象上,调用该函数会触发整个视图的重新渲染,所以不推荐在实际业务中频繁调用

$forceUpdate()函数。$forceUpdate()函数的使用案例的代码如下：

```
<!-- 第 3 章 $forceUpdate()函数的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>组件更新策略</title>
  </head>
  <body>
    <!-- div id 为 app,它的作用是作为 Vue 框架的渲染容器,只有在这个标签内部编写的 Vue 代码
才能被正确地解析 -->
    <div id="app">
      用户信息:{{userInfo}}
      <br>
      <button @click="handleUpdateSex">更新 sex 属性</button>
      <button @click="handleUpdateMoney">更新 money 属性</button>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      var vm=new Vue({
        data(){
          return {
            userInfo:{
              username:'张三',
              age:18,
              sex:'男'
            }
          }
        },
        methods:{
          handleUpdateSex(){
            this.userInfo.sex='女'
          },
          handleUpdateMoney(){
            //this.$set(this.userInfo,'money',10000)
            //用来将 userInfo 对象的 money 属性的值更新
            this.userInfo.money=10000
            this.userInfo.abcd=1234
            //当执行 this.$forceUpdate()时,整个视图的所有内容都会更新
            this.$forceUpdate()
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

3. mixin 功能的介绍

Vue 提供了 mixin(混入)功能,通过定义一个对象,来分发 Vue 组件中的可复用功能。一个混入对象可以包含任意组件选项。当组件使用混入对象时,所有混入对象的选项将被“混合”进入该组件本身的选项。mixin 有两种使用方式。

1) 选项合并

该方式为局部混入,可以在 Vue 实例外部定义相同结构的对象,并将对象合并到任意组件的 mixins 选项中,Vue 实例会自动将两个对象的结构组合起来,当遇到同名变量触发冲突时,Vue 会优先使用组件内部的原有数据。选项合并的应用案例,代码如下:

```
<!-- 第 3 章 选项合并的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>选项合并案例</title>
  </head>
  <body>
    <div id="app">
      <h4>组件资深属性</h4>
      {{hello}}
      <h4>混入组件的属性</h4>
      <input type="text" v-model="str">{{str}}
      <h4>混入时冲突的属性处理结果</h4>
      <!-- 在混入时由于组件内部存在 userInfo,所以 userInfo 的结果优先展示在组件内部 -->
      <!-- 由于混入对象中的 userInfo 中包含组件内不存在的 sex 属性,所以该属性会被合并 -->
      {{userInfo}}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      //定义一个混入对象
      var mixin={
        data(){
          return {
            str:'混入的字符串',
            userInfo:{
              username:'小黄',
              password:'123456',
              sex:'男'
            }
          }
        },
        watch:{
          //混入对象可以使用 Vue 实例中可用的任何功能
          str(v){
            console.log(v)
          }
```

```
        }
      }
      var vm=new Vue({
        mixins:[mixin],
        data(){
          return {
            hello:'你好 Vue',
            userInfo:{
              username:'小明',
              password:'654321',
            }
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

2）全局混入

可以通过 Vue.mixin()定义一个全局的混入对象，该方式会导致定义的混入对象自动被加载到 Vue 实例对象中，同时也被加载到任何该 Vue 实例的自定义组件中。全局混入的入侵能力极其强，可以在组件开发场景中抽取可复用功能，不推荐在业务开发中使用全局混入，因为在业务开发场景中，开发者定义的全局混入对象也会被应用在开发者使用的第三方框架对象中，这样会导致代码污染。全局混入的应用案例，代码如下：

```
<!-- 第 3 章 全局混入的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>全局混入案例</title>
  </head>
  <body>
    <div id="app">
      {{str}}
      <p-div>
        自定义组件
      </p-div>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      //定义一个全局的混入
      Vue.mixin({
        data(){
          return{
            str:'全局混入的变量'
```

```
        }
      },
      //混入对象中的任何选项都会被注入可作用的对象中
      created(){
      //该生命周期会被注入所有的组件对象中
      //只要组件中存在 name 属性,生命周期中就可以得到不同组件的 name 的值
        console.log('混入组件的生命周期: '+this.name)
      }
    })
    //自定义组件
    Vue.component('p-div',{
      data(){
        return {
          name:'我是 p-div 组件的 name 的值'
        }
      },
      template:`
        <div id="p-div">
          <slot/>
          <br/>
          混入组件的 str: {{str}}
        </div>
      `
    })
    //全局 Vue 实例
    var vm=new Vue({
      data(){
        return {
          name:'我是全局 Vue 实例中的 name 的值'
        }
      }
    }).$mount('#app')
  </script>
 </body>
</html>
```

4. 自定义指令的介绍

在之前的章节中介绍了 Vue 的指令系统,凡是在标签上使用的 v-开头的关键字都是 Vue 框架提供的全局指令,并且每个指令都有不同的能力。Vue 框架除了提供了方便的指令系统外,还为开发者提供了自定义指令的通道,开发者可以根据项目的需要,创建 Vue 框架中本不存在的指令关键字,并为这些关键字赋予能力。自定义指令的使用方式为 Vue.directive(指令名称,指令对象)。指令对象中包含 5 个钩子函数,用来为自定义指令从绑定到渲染的不同时期提供事件的通知,具体的钩子函数介绍如下。

1) bind()

该函数只调用一次,指令第一次绑定到元素时调用,在这里可以进行一次性的初始化设置。

2）inserted()

被绑定元素插入父节点时调用。

3）update()

所在组件的 VNode 更新时调用，但可能发生在其子 VNode 更新之前。指令的值可能发生了改变，也可能没有，但可以通过比较更新前后的值，来忽略不必要的模板更新。

4）componentUpdated()

指令所在组件的 VNode 及其子 VNode 全部更新后调用。

5）unbind()

只调用一次，指令与元素解绑时调用。

每个自定义指令的钩子函数中都会被传入固定的几个参数：

1）el

指令所绑定的元素，可以用来直接操作 DOM。

2）binding

自定义指令的描述对象，其中包含多个属性。

（1）name：指令名，不包含 v-前缀。

（2）value：指令绑定的值，例如：v-my-directive="1＋1"中，绑定的值为 2。

（3）oldValue：指令绑定的前一个值，仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。

（4）expression：字符串形式的指令表达式。例如 v-my-directive="1＋1"中，表达式为"1＋1"。

（5）arg：传给指令的参数，可选。例如 v-my-directive:foo 中，参数为"foo"。

（6）modifiers：一个包含修饰符的对象。例如：v-my-directive.foo.bar 中，修饰符对象为{foo：true，bar：true}。

3）vnode

Vue 编译生成的虚拟节点。

4）oldVnode

上一个虚拟节点，仅在 update 和 componentUpdated 钩子中可用。

除 el 之外，其他参数都应该是只读的，切勿进行修改。如果需要在钩子之间共享数据，则建议通过元素的 dataset 进行设置。自定义指令的实际应用案例的代码如下：

```
<!-- 第 3 章 自定义指令实际应用的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>自定义指令案例</title>
  </head>
  <body>
```

```
<div id="app">
  <h4>自定义列表渲染案例</h4>
  <ul>
    <li v-p-for="list" :key="index"></li>
  </ul>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
<script type="text/javascript">
  Vue.directive('p-for',{
    bind(el,binding,vnode){
      //console.log(el,binding,vnode)
      console.log(vnode.parent,el)

    },
    inserted(el,binding,vnode){
      //console.log(el,binding,vnode)
      console.log(vnode,el.parentNode)
      let parent=el.parentNode
      let str=''
      binding.value.forEach((item,index)=>{
        let prefix=`<li>`
        let suffix=`</li>`
        str+=prefix+item.name+suffix
      })
      parent.innerHTML=str
    },
    update(el,binding,vnode){
      console.log(el,binding,vnode)
    },
    componentUpdated(el,binding,vnode){
      console.log(el,binding,vnode)
    },
    unbind(el,binding,vnode){
      console.log(el,binding,vnode)
    }
  })
  var vm=new Vue({
    data(){
      return{
        list:[
          {id:'01',name:'张三丰'},
          {id:'01',name:'张无忌'},
          {id:'01',name:'张翠山'},
          {id:'01',name:'张远桥'},
          {id:'01',name:'张云鹏'}
        ]
      }
    }
  }).$mount('#app')
```

```
    </script>
  </body>
</html>
```

5. 动态渲染的介绍

Vue 框架提供了 Vue.extend()函数，用以动态地为 Vue 创建一个“子类”。Vue.extend()中可以传入与初始化 Vue 实例相同的参数结构，其中的 data 选项要求只能使用函数形式进行创建。Vue.extend()需要依赖一个物理标签作为容器进行组件的渲染。动态渲染的应用案例的代码如下：

```
<!-- 第 3 章 动态渲染的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Vue.extend()案例</title>
  </head>
  <body>
    <div id="app">
      <h4>动态渲染的案例</h4>
      <button @click="handleClick">单击展示</button>
      <!-- 渲染后 div 容器会被替换成子类组件本身 -->
      <div id="container"></div>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      //创建 Vue 的子类对象,并初始化组件内容
      var Profile=Vue.extend({
        template:`
          <p>动态插入的组件内容</p>
        `
      })

      var vm=new Vue({
        methods:{
          handleClick(){
            //实例化子类并渲染到指定容器中
            new Profile().$mount('#container')
          }
        }
      }).$mount('#app')
    </script>
  </body>
</html>
```

6. 异步更新队列

Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓

存在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入队列中一次。这种在缓存时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的，然后在下一个事件循环 tick 中，Vue 刷新队列并执行实际（已去重的）工作。Vue 在内部对异步队列尝试使用原生的 Promise. then、MutationObserver 和 setImmediate，如果执行环境不支持，则会采用 setTimeout(fn, 0)代替。

当设置 vm. someData='new value'时，组件不会立即被重新渲染。当刷新队列时，组件会在下一个事件循环 tick 中更新。多数情况不需要关心这个过程，但是如果想基于更新后的 DOM 状态做点什么，这就可能会有些棘手。虽然 Vue. js 通常鼓励开发人员使用“数据驱动”的方式思考，避免直接接触 DOM，但是有时必须操作 DOM。为了在数据变化后等待 Vue 完成更新 DOM，可以在数据变化后立即使用 Vue. nextTick(callback)。这样回调函数将在 DOM 更新完成后被调用。异步更新队列的应用案例的代码如下：

```
<!-- 第 3 章 异步更新队列的代码案例 -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Vue.nextTick()案例</title>
  </head>
  <body>
    <div id="app">
      <h4>异步更新队列案例</h4>
      <button @click="handleClick">单击展示</button>
      <div id="n">{{name}}</div>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
    <script type="text/javascript">
      var vm=new Vue({
        data(){
          return {
            name:'初始值'
          }
        },
        methods:{
          handleClick(){

            this.name='修改后的新值'
            var n=document.getElementById('n')
            //查看 DOM 对象是否渲染了新的结果
            //此时会输出'初始值'
            console.log(n.innerHTML)
            //Vue.nextTick()在实例内部可以使用 this.nextTick()进行调用
            this.$nextTick().then(()=>{
              //使用 nextTick 来查看视图是否更新完毕
              //此时会输出'修改后的新值'
```

```
            console.log(n.innerHTML)
          })
        }
      }
    }).$mount('#app')
  </script>
 </body>
</html>
```