第 5 章 组　　件

在前端应用程序开发中，如果所有的实例都写在一起，则必然会导致代码既长又不好理解。组件的出现刚好解决了这一问题，它是带有名字的可复用实例，不仅可以重复使用，还可以扩展。组件(Component)是 Vue 最核心的功能，也是整个框架设计最精彩的地方，当然也是比较难掌握的。每个开发者都想在软件开发过程中使用之前写好的代码，但又担心引入这段代码对现有的程序产生影响。Web Components 的出现提供了一种新的思路，可以自定义 tag 标签，并拥有自身的模板、样式和交互。另外，Vue 3. x 新增了组合 API，它是一组附加的、基于函数的 API，允许灵活地组合组件逻辑。

5.1　什么是组件

在正式介绍组件前，先看一个 Vue 组件的简单示例，大家先感受一下，代码如下：

```
//定义一个名为 button - counter 的组件
const vm = Vue.createApp({});
vm.component('button - counter', {
   data() {
       return {
           count: 0
       }
     },
template: '< button v - on:click = "count++"> You clicked me {{ count }} times.</button >'
})
vm.mount('#app');
```

组件是可复用的 Vue 实例，并且带有一个名字，在这个示例中组件是< button-counter >。x 把这个组件作为自定义标签来使用，代码如下：

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

完整的组件使用示例代码如例 5-1 所示。

【例 5-1】 自定义组件

```
//第 5 章/自定义组件.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <button-counter></button-counter>
    <button-counter></button-counter>
</div>
<script>
    //定义一个名为 button-counter 的新组件
    const vm = Vue.createApp({});
    vm.component('button-counter', {
       data() {
          return {
             count: 0
          }
       },
    template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'
    })
    vm.mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-1 所示。

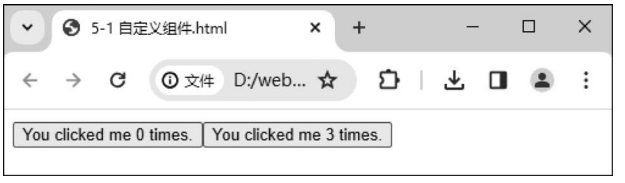

图 5-1　自定义组件

这些类似于<button-counter>自定义的标签就是组件，每个标签代表一个组件，这样就可以将组件进行任意次数的复用。

Web 的组件其实就是页面组成的一部分，好比是计算机中的每个元器件（如硬盘、键盘、鼠标等），它有一个独立的逻辑和功能或界面，同时又能根据规定的接口规则进行相互融合，从而变成一个完整的应用。

Web 页面就是由一个个类似这样的部分组成的，例如导航、列表、弹窗、下拉菜单等。

页面只不过是这些组件的容器,组件自由组合形成功能完整的界面,当不需要某个组件或者想要替换某个组件时,可以随时进行替换和删除,而不影响整个应用的运行。

前端组件化的核心思路就是将一个巨大复杂的逻辑和功能分成粒度合理的小逻辑和功能。使用组件的好处如下:

(1) 提高开发效率。

(2) 方便重复使用。

(3) 简化调试步骤。

(4) 提升整个项目的可维护性。

(5) 便于协同开发。

组件是 Vue.js 最强大的功能之一。组件可以扩展 HTML 元素,封装可重用的代码。在较高层面上,组件是自定义元素,Vue.js 的编译器为它添加特殊功能。在有些情况下,组件也可以采用原生 HTML 元素的形式,以 is 特性扩展。

组件系统让可以用独立可复用的小组件来构建大型应用,几乎任意类型的应用界面都可以抽象为一棵组件树,如图 5-2 所示。

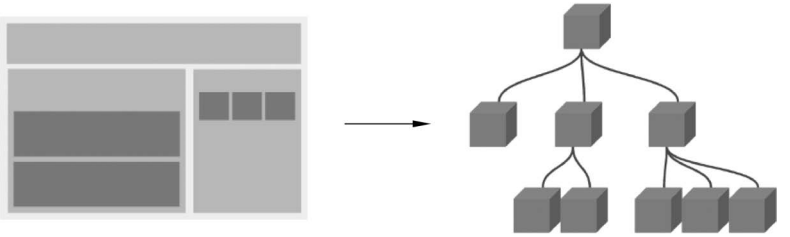

图 5-2 Vue 组件树

5.2 组件的基本使用

为了能在模板中使用,这些组件必须先注册以便 Vue 能够识别。这里有两种组件的注册类型:全局注册和局部注册。

5.2.1 全局注册

全局注册组件使用应用程序实例的 component()方法来注册组件。全局注册有以下两种方式。

1. 第 1 种方式

注册全局组件,代码如下:

```
const vm = Vue.createApp({})
  vm.component('my-component', {
     //选项
  })
```

my-component 就是注册的组件自定义标签名称，推荐使用小写加分隔符的形式命名。因为组件最后会被解析成自定义的 HTML 代码，所以可以直接在 HTML 中使用组件名称作为标签使用，如例 5-2 所示。

【例 5-2】 全局注册组件方式一

```
//第 5 章/全局注册组件方式一.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="https://unpkg.com/vue@next"></script>
    <title>Document</title>
</head>
<body>
    <div id="app">
        <my-component></my-component>
    </div>
    <script>
        const vm = Vue.createApp({})
        vm.component('my-component', {
            template: '<h1>注册</h1>'
        })
        vm.mount('#app');
    </script>
</body>
</html>
```

在浏览器中的显示效果如图 5-3 所示。

图 5-3 全局注册组件方式一

注意：

(1) template 的 DOM 结构必须被一个而且是唯一的根元素包含，如果直接引用，则不被 包裹是无法被渲染的。

(2) 模板(template)声明了数据和最终展现给用户的 DOM 之间的映射关系。

除了 template 选项外，组件中还可以有其他的选项，例如 data、computed、methods 等，代码如下：

```
<div id="app">
    <my-component></my-component>
</div>
```

```
<script>
    const vm = Vue.createApp({})
    vm.component('my-component', {
        template: '<h1>{{message}}</h1>',
        data:function () {
            return{
                message:'注册'
            }
        }
    })
    vm.mount('#app');
</script>
```

Vue组件中data为函数的原因。data为函数,通过return返回对象的复制,使每个实例都有自己独立的对象,实例之间可以互不影响地改变data属性值。

2. 第2种方式

将模板字符串直接定义到script标签中,代码如下:

```
<script id="tmpl" type="text/x-template">
   <div><a href="#">登录</a>|<a href="#">注册</a></div>
</script>
```

同时,使用component()将其定义在组件中,代码如下:

```
<script>
    const vm = Vue.createApp({})
    vm.component("account",{
      template: '#tmpl'
    });
    vm.mount('#app');
</script>
```

完整示例代码如例5-3所示。

【例5-3】 全局注册组件方式二

```
//第5章/全局注册组件方式二.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="https://unpkg.com/vue@next"></script>
    <title>Document</title>
</head>
<body>
<div id="app">
    <account></account>
</div>
<!-- <script id="tmpl" type="text/x-template">
```

```
    <div><a href="#">登录</a> | <a href="#">注册</a></div>
</script> -->

<template id="tmpl">
    <div><a href="#">登录</a> | <a href="#">注册</a></div>
</template>
<script>
    const vm = Vue.createApp({})
    vm.component("account",{
      template: '#tmpl'
    });
    vm.mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-4 所示。

图 5-4　全局注册组件方式二

5.2.2　局部注册

如果不需要全局注册，或者只想在一个 Vue 实例中使用，则可以使用局部注册的方式注册组件。在 Vue 实例中，可以通过对象的 components 属性实现局部注册，如例 5-4 所示。

【例 5-4】 局部注册组件

```
//第 5 章/局部注册组件.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="https://unpkg.com/vue@next"></script>
    <title>Document</title>
</head>
<body>
<div id="app">
    <account></account>
</div>
<script>
    const vm = Vue.createApp({
        components: {                    //定义子组件
          account: {                     //account 组件
          //在这里使用定义的子组件
```

```
            template: '<div><h1>这是 Account 组
                  </h1><login></login></div>',
                components: {          //定义子组件的子组件
                  login: {             //login 组件
                      template: "<h3>这是登录组件</h3>"
                  }
              }
          }
        }
    }).mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-5 所示。

图 5-5　局部注册组件

可以使用 flag 标识符结合 v-if 和 v-else 切换组件,如例 5-5 所示。

【例 5-5】 使用标识符切换组件

```
//第 5 章/使用标识符切换组件.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="https://unpkg.com/vue@next"></script>
    <title>Document</title>
</head>
<body>
<div id="app">
    <input type="button" value="切换" @click="flag=!flag">
    <account v-if="flag"></account>
    <login v-else="flag"></login>
</div>
<script>
    const vm = Vue.createApp({
        data(){
          return {
              flag: true
          }
```

```
        },
        methods: {},
        components: {                    //定义子组件
          account: {                     //account 组件
              //在这里使用定义的子组件
              template: '<div><h1>这是 Account 组件</h1></div>',
          },
          login: {                       //login 组件
              template: "<h3>这是登录组件</h3>"
          }
        }
    }).mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-6 所示。

图 5-6　使用标识符切换组件效果

5.3　使用 prop 向子组件传递数据

组件是当作元素来使用的，而元素一般是有属性的，同样组件也可以有属性。在使用组件时，给元素设置属性，组件内部如何接受呢？首先需要在组件代码中注册一些自定义的属性，称为 prop，这些 prop 是在组件 props 选项中定义的，之后在使用组件时，就可以把这些 prop 的名字作为元素的属性来使用，通过属性向组件传递数据，这些数据将作为组件实例的属性被使用。

5.3.1　prop 的基本用法

以上章节使用组件主要是将组件模板的内容进行复用，代码如下：

```
<!-- 父组件 -->
<div id="app">
    <my-component></my-component>
    <my-component></my-component>
</div>
<script>
```

```
    //子组件
    const vm = Vue.createApp({})
    vm.component("my-component",{
      template: '<h1>注册</h1>'
    });
    vm.mount('#app');
</script>
```

但是组件中更重要的是组件间进行通信,选项 props 是组件中非常重要的一个选项,起到父子组件间桥梁的作用。

以下是组件选项 props 常用的几种传值方式。

1. 静态 props

组件实例的作用域是孤立的,这意味着不能(也不应该)在子组件的模板内直接引用父组件的数据,示例代码如下:

```
<!-- 父组件 -->
<div id = "app">
    <my-componet message = "来自父组件的数据!"></my-componet>
</div>
<script>
    //子组件
    const vm = Vue.createApp({})
    vm.component("my-component",{
      template: '<span>{{ message }}</span>'
    });
    vm.mount('#app');
</script>
```

在以上代码中子组件是接收不到父组件 message 数据的。

如果要想让子组件使用父组件的数据,则需要通过子组件的 props 选项实现。子组件要显式地用 props 选项声明它期待获得的数据,如例 5-6 所示。

【例 5-6】 子组件接受父组件的数据

```
//第5章/子组件接受父组件的数据.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset = "utf-8"/>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id = "app">
        <my-componet message = "来自父组件的数据!"></my-componet>
    </div>
    <script>
```

```
        const vm = Vue.createApp({})
        vm.component('my-componet', {
          //声明 props
          props: ['message'],
          template: '<span>{{ message }}</span>'
        })
        vm.mount('#app');
    </script>
</body>
</html>
```

在浏览器中的显示效果如图 5-7 所示。

图 5-7　子组件接受父组件的数据效果

由于 HTML 特性是不区分大小写的，因此当使用的不是字符串模板，而是驼峰式命名的 props 时需要转换为相对应的 kebab-case(短横线隔开式) 命名，示例代码如下：

```
<div id="app">
    <my-componet my-message="来自父组件的数据!"></my-componet>
</div>
<script>
    const vm = Vue.createApp({});
    vm.component('my-componet', {
        //声明 props
        props: ['myMessage'],
        template: '<span>{{ myMessage }}</span>'
    })
    vm.mount('#app');
</script>
```

2. 动态 props

在模板中，有时传递的数据并不一定是固定的，而是要动态地将父组件的数据绑定到子模板的 props，与绑定到任何普通的 HTML 特性相类似，即使用 v-bind。当父组件的数据变化时，该变化也会传递给子组件，如例 5-7 所示。

【例 5-7】 动态 props

```
//第 5 章/动态 props.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
```

```
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <input type="text" v-model="parentMessage">
    <my-componet :message="parentMessage"></my-componet>
</div>
<script>
    const vm = Vue.createApp({
        data(){
          return{
              parentMessage:''
          }
        }
    });
    vm.component('my-componet', {
        props: ['message'],
        template: '<span>{{ message }}</span>'
    })
    vm.mount('#app');
</script>
</body>
</html>
```

在文本框中输入信息,子组件就会动态地接收到父组件的数据。如在文本框中输入“贝西奇谈”,组件即可接收到该数据。在浏览器中的显示效果如图 5-8 所示。

图 5-8 动态接收父组件数据效果图

这里使用 v-model 绑定了父组件数据 parentMessage,当在输入框中输入数据时,子组件接收的 props:['message']也会实时响应,并更新组件模板。

对于初学者常犯的一个错误,如果在父组件中直接传递数字、布尔值、数组、对象,则所传递的值均为字符串;如果想传递一个实际的值,则需要使用 v-bind,从而让它的值被当作 JavaScript 表达式进行计算,示例代码如下:

```
<!-- 父组件 -->
<div id="app">
    <my-componet message="1+1"></my-componet><br>
    <my-componet :message="1+1"></my-componet>
</div>
<script>
    //子组件
    const vm = Vue.createApp({});
```

```
    vm.component("my-componet",{
        props: ['message'],
        template: '<span>{{ message }}</span>'
    });
    vm.mount('#app');
</script>
```

如果不使用 v-bind 指令，页面显示结果是字符串“1＋1”。如果使用 v-bind 指令，则会当作 JavaScript 表达式计算结果，即数字 2。

3. 多值传递

如果组件需要传递多个值，则可以定义多个 prop 属性，如例 5-8 所示。

【例 5-8】 传递多个值

```
//第5章/传递多个值.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="https://unpkg.com/vue@next"></script>
    <title>Document</title>
</head>
<body>
 <!-- 父组件 -->
<div id="app">
    <my-book></my-book>
</div>
<script>
    //子组件 book
    const vm = Vue.createApp({});
    vm.component("my-book",{
        template:
'<div><book-title :name="name" :price="price" :author="author"></blog-title></div>',
        data(){
           return{
               name:"剑指大前端全栈工程师",
               price:"219元",
               author:"贾志杰"
           }
        }
    });
    vm.component("book-title",{
        props: ['name','price','author'],
        template:
'<ul><li>{{name}}</li><li>{{price}}</li><li>{{author}}</li></ul>'
    })
    vm.mount('#app');
</script>
```

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

在浏览器中的显示效果如图 5-9 所示。

图 5-9　传递多个值

5.3.2　prop 验证

当开发一个可复用的组件时,父组件希望通过 prop 属性传递的数据类型符合要求。例如组件定义一个 prop 属性是一个对象类型,结果父组件传递的是一个字符串的值,这明显不合适。Vue 提供了 prop 属性的验证规则,在定义 props 选项时,使用一个带验证需求的对象来代替之前使用的字符串组(props:['name','price','author'])。

当组件的 prop 指定验证要求后,如果有一个需求没有被满足,则 Vue 会在浏览器控制台中发出警告,示例代码如下:

```
vm.component('example', {
        props: {
            //基础类型检测 (null 的意思是任何类型都可以)
            propA: Number,
            //多种类型
            propB: [String, Number],
            //必传且是字符串
            propC: {
                type: String,
            required: true
            },
            //数字,有默认值
            propD: {
                type: Number,
            default: 100
            },
            //数组和对象的默认值应当由一个工厂函数返回
            propE: {
                type: Object,
                default: function () {
                    return { message: 'hello' }
                }
            },
            //自定义验证函数
```

```
            propF: {
                validator: function (value) {
                return value > 10
            }
        }
    }
    })
```

验证的 type 类型可以是：String、Number、Boolean、Function、Object、Array 等，type 也可以是一个自定义构造器函数，使用 instanceof 检测。

当 prop 验证失败时，Vue 会抛出警告（如果使用的是开发版本）。由于 prop 会在组件实例创建之前进行校验，所以在 default 或 validator 函数里，诸如 data、computed 或 methods 等实例属性还无法使用。

【例 5-9】 prop 验证

```
//第 5 章/prop 验证.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <parent></parent>
</div>
<script>
    var childNode = {
       template: '<div>{{message}}</div>',
       props:{
          'message':Number
       }
    }
    var parentNode = {
    template:'<div
      class="parent"><child :message="msg"></child></div>',
        components: {
          'child': childNode
        },
        data(){
          return{
             msg: '123'
          }
        }
    };
    const vm = Vue.createApp({
       components: {
```

```
            'parent': parentNode
        }
    }).mount('#app');
</script>
</body>
</html>
```

以上示例中校验的是数字,当传入数字 123 时,无警告提示。当传入字符串"123"时,控制台会发出警告,如图 5-10 所示。

图 5-10 prop 验证失败时的警告信息

5.3.3 单项数据流

所有的 prop 都使其父子 prop 之间形成了一个单向下行绑定。父级 prop 的更新会向下流动到子组件中,但是反过来则不行。之所以这样设计,是因为要尽可能地将父子组件解耦,避免子组件无意中修改了父组件的状态。

另外,每次父级组件发生变更时,子组件中所有的 prop 都将被刷新为最新的值。这意味着不应该在一个子组件的内部改变 prop。如果这样做了,则 Vue 会在浏览器的控制台中发出警告,如例 5-10 所示。

【例 5-10】 单项数据传递

```
//第 5 章/单项数据传递.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <parent></parent>
</div>
<script>
```

```
    var childNode = {
        template:
        '<div class="child"><div><span>子组件数据</span>' +
            '<input v-model="childMsg"></div><p>{{childMsg}}</p></div>',
            props:['childMsg']
        }
        var parentNode = {
            template:
            '<div class="parent"><div><span>父组件数据</span>' +
                  '<input
                  v-model="msg"></div><p>{{msg}}</p><child :child-msg="msg">
                  </child></div>',
            components: {
                  'child': childNode
            },
            data(){
                  return {
                      'msg':'match'
                  }
            }
    };
    const vm = Vue.createApp({
        components: {
            'parent': parentNode
        }
    }).mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-11 所示。

图 5-11 单项数据传递页面显示效果

当父组件数据变化时，子组件数据会实时地响应数据，而当子组件数据变化时，父组件数据不变，并在控制台显示警告，如图 5-12 所示。

注意：JavaScript 中对象和数组是引用类型，指向同一个内存空间，如果 prop 是一个对象或数组，则在子组件内部改变它时会影响父组件的状态。

有两种情况可能需要改变组件的 prop 属性。

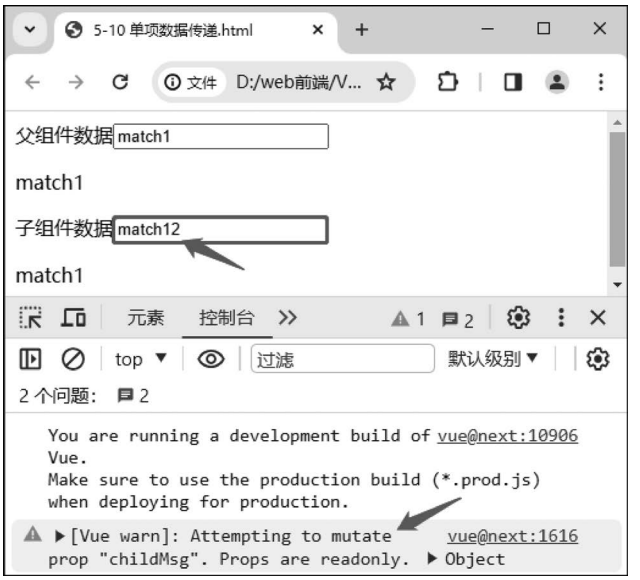

图 5-12 更改子组件时的警告信息

第 1 种情况是定义一个 prop 属性,以方便父组件传递初始值,在子组件内将这个 prop 作为一个本地的 prop 数据来使用。遇到这种情况,解决办法是在本地 data 选项中定义一个属性,然后将 prop 属性值作为其初始值,后续操作只访问这个 data 属性,示例代码如下:

```
props: ['initData'],
data: function () {
   return {
      counter: this.initData
      }
}
```

但定义的局部变量 counter 只能接收 initData 的初始值,当父组件要传递的值发生变化时,counter 无法接收到最新值,如例 5-11 所示。

【例 5-11】 改变组件的 prop 属性(1)

```
//第 5 章//改变组件的 prop 属性(1).html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
```

```
    <parent></parent>
</div>
<script>
    var childNode = {
        template:
        '<div class="child"><div><span>子组件数据</span>' +
              '<input v-model="temp"></div><p>{{temp}}</p></div>',
        props:['childMsg'],
        data(){
           return{
              temp:this.childMsg
           }
        },
    };
    var parentNode = {
        template:
    '<div class="parent"><div><span>父组件数据</span><input
    v-model="msg"> ' +
          '</div><p>{{msg}}</p><child :child-msg="msg"></child></div>',
          components: {
              'child': childNode
          },
        data(){
          return {
              'msg':'match'
          }
        }
    };
    const vm = Vue.createApp({
        components: {
           'parent': parentNode
        }
    }).mount('#app');
</script>
</body>
</html>
```

以上示例除初始值外，父组件更改的值无法更新到子组件中，如图 5-13 所示。

图 5-13　改变组件的 prop 属性(1)

第 2 种情况是 prop 属性接收到数据后需要转换后使用,这种情况可以使用计算属性来解决此问题,示例代码如下:

```
props: ['size'],
computed: {
   normalizedSize: function () {
    return this.size.trim().toLowerCase()
   }
  }
```

但是,由于是计算属性,所以只能显示值,而不能设置值,如例 5-12 所示。

【例 5-12】 改变组件的 prop 属性(2)

```
//第 5 章//改变组件的 prop 属性(2).html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <parent></parent>
</div>
<script>
    var childNode = {
        template:
        '<div class="child"><div><span>子组件数据</span>' +
                '<input v-model="temp"></div><p>{{temp}}</p></div>',
        props:['childMsg'],
        computed:{
             temp(){
                return this.childMsg
             }
        },
    };
    var parentNode = {
        template:
    '<div class="parent"><div><span>父组件数据</span><input
    v-model="msg"> ' +
          '</div><p>{{msg}}</p><child :child-msg="msg"></child></div>',
        components: {
            'child': childNode
        },
        data(){
            return {
                'msg':'match'
            }
        }
```

```
    };
    const vm = Vue.createApp({
        components: {
            'parent': parentNode
        }
    }).mount('#app');
</script>
</body>
</html>
```

以上示例由于子组件使用的是计算属性，所以子组件的数据无法手动修改，只能通过父组件数据更改来实时获取，如图 5-14 所示。

图 5-14　改变组件的 prop 属性(2)

综上方案，更加稳妥的方法是使用变量储存 prop 的初始值，并使用 watch 来监听 prop 的值的变化。当发生变化时，更新变量的值，代码如下：

```
<div id = "app">
    <parent></parent>
</div>
<script>
    var childNode = {
        template:
        '<div class = "child"><div><span>子组件数据</span>' +
              '<input v - model = "temp"></div><p>{{temp}}</p></div>',
        props:['childMsg'],
        data(){
          return{
              temp:this.childMsg
          }
        },
        watch:{
            childMsg(){
                this.temp = this.childMsg
            }
        }
    };
    var parentNode = {
```

```
        template:
    '<div class="parent"><div><span>父组件数据</span><input
    v-model="msg"> ' +
        '</div><p>{{msg}}</p><child :child-msg="msg"></child></div>',
        components: {
            'child': childNode
        },
        data(){
           return {
              'msg':'match'
           }
        }
    };
    const vm = Vue.createApp({
        components: {
           'parent': parentNode
        }
    }).mount('#app');
</script>
```

5.4 子组件向父组件传递数据

在 Vue 组件通信中其中最常见的通信方式就是父子组件之中的通信，而父子组件的设定方式在不同情况下又各有不同，如图 5-15 所示。父组件传递数据给子组件使用，当遇到业务逻辑操作时子组件触发父组件的自定义事件。

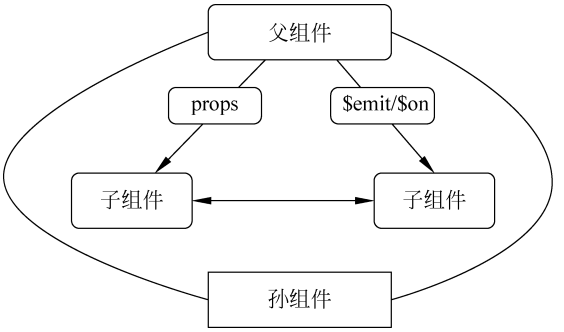

图 5-15 组件通信

作为一个 Vue 初学者不得不了解的就是组件间的数据通信(暂且不谈 Vuex，后面章节会讲到)。通信方式根据组件之间的关系有不同之处。组件关系有 3 种：父→子、子→父、非父子。

5.3 节已讲解，父组件向子组件通信是通过 props 传递数据的，就好像方法的传参一样，父组件调用子组件并传入数据，子组件接收到父组件传递的数据后进行验证使用。

那么子组件如何向父组件传递数据呢？具体可看下面的讲解。

5.4.1 自定义事件

当子组件需要向父组件传递数据时，就要用到自定义事件。v-on 指令除了可以监听 DOM 事件外，还可以用于组件之间的自定义事件。

在子组件中用 $emit()来触发事件以便将内部的数据报告给父组件，如例 5-13 所示。

【例 5-13】 子组件向父组件传递数据

```
//第 5 章//子组件向父组件传递数据.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <my-component v-on:myclick="onClick"></my-component>
</div>
<script>
    const vm = Vue.createApp({
        methods: {
            onClick () {
                console.log(arguments)
            }
        }
    })
    vm.component('my-component', {
       template:'<div>' +
       '<button type="button" @click="childClick">单击我触发自定义事件</button></div>',
       methods: {
           childClick () {
           this.$emit('myclick', '这是我暴露出去的数据 1', '这是我暴露出去的数据 2')
           }
       }
    })
    vm.mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-16 所示。

接下来解析上面示例中的代码。

(1) 子组件在自己的方法中将自定义事件及需要发出的数据通过以下代码发送出去：

```
this.$emit('myclick','这是我暴露出去的数据 1', '这是我暴露出去的数据 2')
```

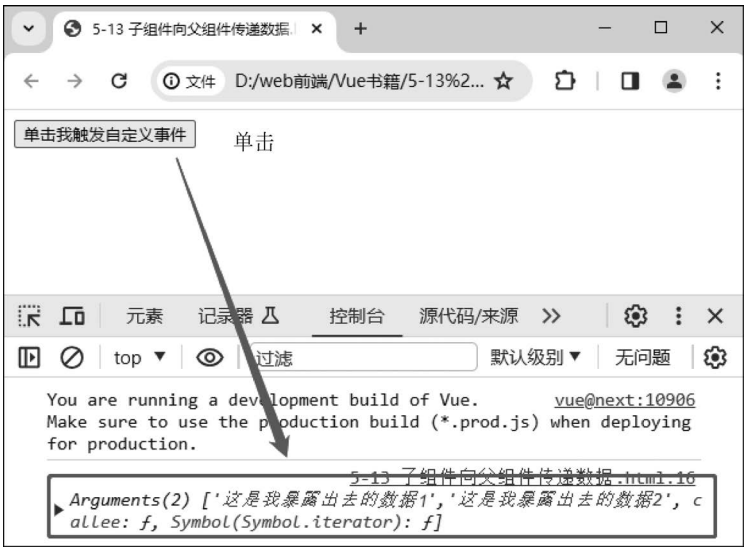

图 5-16　子组件向父组件传递数据效果

➢ 第 1 个参数是自定义事件的名字。

➢ 后面的参数是依次想要发送出去的数据。

(2) 父组件利用 v-on 为事件绑定处理器,代码如下:

```
<my-component v-on:myclick="onClick"></my-component>
```

这样,在 Vue 实例的 methods 方法中就可以调用传进来的参数了。

5.4.2　sync 修饰符

在有些情况下,可能需要对一个 prop 属性进行“双向绑定”,但是真正的双向绑定会带来维护上的问题,因为子组件可以变更父组件,并且父组件和子组件都没有明显的变更来源。Vue 推荐 update.myPropName 模式触发事件实现,如例 5-14 所示。

【例 5-14】 设计购物的数量

其中子组件中的代码如下:

```
<script>
    const vm = Vue.createApp({
        data(){
            return{
                counter:0
            }
        }
    })
    vm.component('child',{
        data(){
```

```
            return{
                count:this.value
            }
        },
        props:{
          value:{
              type:Number,
              default:0
          }
        },
        methods:{
            handleClick(){
                this.$emit("update:value",++this.count)
            },
        },
        template:`
            <div>
             <span>子组件:购买 {{value}}件</span>
             <button v-on:click="handleClick">增加</button>
            </div>
        `
    })
    vm.mount('#app');
</script>
```

在这个子组件中有一个 prop 属性 value,在按钮的 click 事件处理器中,调用 $emit()方法触发 update:value 事件,并将加 1 后的计数值作为事件的附加参数。

在父组件中,使用 v-on 指令监听 update:value 事件,这样就可以接收到子组件传来的数据,然后使用 v-bing 指令绑定子组件的 prop 属性 value,这样就可以给子组件传递父组件的数据了,从而实现了双向数据绑定。父组件中的代码如下:

```
<div id="app">
    父组件:购买{{counter}}件
    <child v-bind:value="counter"
        v-on:update:value="counter= $event"></child>
</div>
```

其中,$event 是自定义事件的附加参数。

完整的示例代码如下:

```
//第5章/设计购物的数量.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
```

```
<body>
<div id="app">
    父组件:购买{{counter}}件
    <child v-bind:value="counter"
        v-on:update:value="counter = $event"></child>
</div>
<script>
    const vm = Vue.createApp({
        data(){
           return{
              counter:0
           }
        }
    })
    vm.component('child',{
        data(){
           return{
              count:this.value
           }
        },
        props:{
          value:{
              type:Number,
              default:0
          }
        },
        methods:{
           handleClick(){
              this.$emit("update:value",++this.count)
           },
        },
        template:`
            <div>
              <span>子组件:购买{{value}}件</span>
              <button v-on:click="handleClick">增加</button>
            </div>
        `
    })
    vm.mount('#app');
</script>
</body>
</html>
```

在浏览器中运行程序,单击5次“增加”按钮,可以看到父组件和子组件中的购买数量是同步变化的,结果如图5-17所示。

为了方便起见,Vue为prop属性的“双向绑定”提供了一个缩写,即.sync修饰符,修改上面示例的<child>的代码如下:

```
<child v-bind.sync="counter"></child>
```

图 5-17 同步更新父组件和子组件的数据

注意，带有.sync 修饰符的 v-bind 不能和表达式一起使用，bind:title.sync="doc.title+'!'"是无效的，代码如下：

```
v-bind:value.sync="doc.title+'!'"
```

上面的代码是无效的，取而代之的是，只提供想要绑定的属性名，类似于 v-bind。

当用一个对象同时设置多个 prop 属性时，也可以将.sync 修饰符和 v-bind 配合使用：

```
<child v-bind.sync="doc"></child>
```

这样会把 doc 对象中的每个属性都作为一个独立的 prop 传进去，然后各自添加，用于更新 v-on 监听器。

5.5 内容分发

在实际项目开发中，时常会把父组件的内容与子组件自己的模板混合起来使用，而这样的一个过程在 Vue 中被称为内容分发。也常常被称为 slot(插槽)，其主要参照了当前 Web Components 规范草案，使用特殊的< slot >元素作为原始内容的插槽。

5.5.1 基础用法

由于 slot 是一块模板，因此对于任何一个组件，从模板种类的角度来分，其实都可分为非插槽模板和插槽模板，其中非插槽模板指的是 HTML 模板(也就是由 HTML 的一些元素构成的，例如 div、span 等)，其显示与否及怎么显示完全由插件自身控制，但插槽模板(也就是 slot)是一个空壳子，它显示与否及怎么显示完全是由父组件来控制的。不过，插槽显示的位置由子组件自身决定，slot 写在组件 template 的哪块，父组件传过来的模板将来就显示在哪块。

一般定义子组件的代码如下：

```
//省略部分代码
<div id="app">
    <child>
        <span>123456</span>
    </child>
</div>
```

```
<script>
    const vm = Vue.createApp({
       components: {
          child:{
             template:"<div>这是子组件内容</div>"
          }
       }
    }).mount('#app');
</script>
```

以上代码在浏览器页面的显示结果为“这是子组件内容”,而123456内容并不会显示。

注意:虽然标签被子组件的child标签所包含,但由于它不在子组件的template属性中,因此不属于子组件。

如果要想标签的内容被显示,则需要在template中添加插槽<slot>标签,代码如下:

```
//省略部分代码
<div id = "app">
    <child>
        <span>123456</span>
    </child>
</div>
<script>
    const vm = Vue.createApp({
       components: {
          child:{
             template:"<div><slot></slot>这是子组件内容</div>"
          }
       }
    }).mount('#app');
</script>
```

以上代码在浏览器页面的显示结果为“123456这是子组件内容”。

我们分步解析内容分发,现在看一个架空的例子,帮助我们理解刚刚说过的严谨而难懂的定义。假设有一个组件,名为my-component,其使用上下文,代码如下:

```
<my-component>
    <p>hi,slots</p>
</my-component>
```

再假设此组件的模板如下:

```
<div>
    <slot></slot>
</div>
```

那么注入后的组件 HTML 相当于：

```
<div>
    <p>hi,slots</p>
</div>
```

<slot>标签会把组件使用上下文的内容注入此标签所占据的位置上，可以把<slot>标签理解为占位。组件分发的概念简单而强大，因为它意味着对一个隔离的组件除了通过属性、事件交互之外，还可以注入内容，如例 5-15 所示。

【例 5-15】 插槽

```
//第 5 章/插槽.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <my-component>
        <p>实力打造大前端时代，走在时代的前端</p>
<p>《剑指大前端全栈工程师》</p>
    </my-component>
</div>
<script>
    const vm = Vue.createApp({})
    vm.component('my-component', {
        template:'<div><slot></slot></div>'
    });
    vm.mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-18 所示。

图 5-18 插槽应用效果图

一个组件如果需要外部传数据，如数字、字符串等，则可以使用 property；如果需要传入 JavaScript 表达式或对象，则可以使用事件；如果希望传入的是 HTML 标签，则使用内

容分发就再好不过了,所以尽管内容分发这个概念看起来极为复杂,但实际上可以简化理解为把 HTML 标签传入组件的一种方法,所以归根结底,内容分发是一种为组件传递参数的方法。

5.5.2 编译作用域

在深入讲解内容分发 API 之前,先明确内容在哪个作用域里编译。假定模板如下:

```
<child-component>
   {{ message }}
</child-component>
```

这里的 message 应该是绑定到父组件的数据,还是绑定到子组件的数据? 答案是 message 就是一个 slot,但它绑定的是父组件的数据,而不是组件< child-component >的数据。

组件作用域简单地说就是父组件模板的内容在父组件作用域内编译,子组件模板的内容在子组件作用域内编译,如例 5-16 所示。

【例 5-16】 编译作用域

```
//第 5 章/编译作用域.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <child-component v-show="someChildProperty"></child-component>
</div>
<script>
    const vm = Vue.createApp({})
    vm.component('child-component', {
        template: '<div>这是子组件内容</div>',
        data() {
           return {
                 someChildProperty: true
          }
        }
    })
    vm.mount('#app');
</script>
</body>
</html>
```

由于这里 someChildProperty 绑定的是父组件的数据,所以是无效的,获取不到数据,如图 5-19 所示。

图 5-19 编译作用域

如果想获取数据，则 someChildProperty 需绑定在子组件上，修改后的代码如下：

```
<div id = "app">
    <child - component ></child - component >
</div >
<script >
    const vm = Vue.createApp({})
    vm.component('child - component', {
       template: '<div v - show = "someChildProperty">这是子组件内容</div >',
       data() {
          return {
             someChildProperty: true
          }
       }
    })
    vm.mount('#app');
</script >
```

这样便可获取数据，因此，slot 分发的内容是在父作用域内编译的。

5.5.3 默认内容

如果要父组件在子组件中插入内容，则必须在子组件中声明 slot 标签；如果子组件模板不包含< slot >插口，则父组件的内容将会被丢弃，如例 5-17 所示。

【例 5-17】 默认内容

```
//第 5 章/默认内容.html
<!DOCTYPE html >
<html >
<head >
    <title ></title >
    <meta charset = "utf - 8"/>
```

```
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
<!-- 那组件 innerHTML 位置以后不管有何代码都会被放进插槽里面 -->
    <index>
        <p>首页</p>
        <p>新闻</p>
        <p>军事</p>
        <h1>手机</h1>
    </index>
</div>
<script>
    //插槽的作用就是从组件外部取代码片段放到组件内部来
/*定义默认插槽通过 slot 组件定义,定义好了之后就相当于一个坑,可以把它理解为计算机上 USB 插口*/
    const vm = Vue.createApp({})
    vm.component('index', {
       template:'<div>index</div>'
    })
    vm.mount('#app');
</script>
</body>
</html>
```

浏览器页面的显示结果为 index。所有子组件中的内容都不会被显示,即被丢弃。如果要想父组件在子组件中插入内容,则必须在子组件中声明 slot 标签,更改后的示例代码如下:

```
<script>
    const vm = Vue.createApp({})
    vm.component('index', {
       template:'<div><slot></slot>index</div>'
    })
    vm.mount('#app');
</script>
```

在浏览器中的显示效果如图 5-20 所示。

图 5-20 默认内容效果图

5.5.4 命名插槽

在组件开发中,有时需要使用多个插槽。< slot >元素有一个特殊的属性 name,它用来命名插槽,因此,可以定义多个名字不同的插槽。如例 5-18 所示。

具体使用方法如下:

(1) 父组件在命名插槽提供内容时,可以在一个< template >元素上使用 v-slot 指令,并以 v-slot 参数的形式提供其名称,如“v-slot: 名称”。

(2) 子组件在对应分发位置上的 slot 标签添加属性"name=名称"。

【例 5-18】 命名插槽

```
//第 5 章/命名插槽.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <child>
        <template v-slot:one>
            <h2>静夜思</h2>
        </template>
        <template v-slot:two>
            <h3>床前明月光,疑是地上霜</h3>
        </template>
        <template v-slot:three>
            <h3>举头望明月,低头思故乡</h3>
        </template>
    </child>
</div>
<script>
    const vm = Vue.createApp({
       components:{
          child:{
             template:`
<div>
<slot name='one'></slot>
<slot name='two'></slot>
<slot name='three'></slot>
</div>`
          }
       }
    }).mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-21 所示。

图 5-21　命名插槽效果图

总结：slot 分发其实就是父组件需要在子组件内放一些 DOM，它负责这些 DOM 是否显示，以及在哪个地方显示。

5.5.5　作用域插槽

在父级作用域下，在插槽的内容中是无法访问子组件的数据属性的，但有时需要在父级的插槽内容中访问子组件的属性，可以在子组件的< slot >元素上使用 v-bind 指令绑定一个 prop 属性。例如以下组件代码：

```
<div id = "app">
    <child></child>
</div>
<script>
    const vm = Vue.createApp({})
    vm.component('child', {
        data:function(){
          return{
              info:{
              book:"剑指大前端全栈工程师",
              author:"贾志杰",
              sex:"男"
          }
      }
    },
    template: `
        <button>
           <slot v-bind:values = "info">
                {{info.book}}
           </slot>
        </button>
      `,
  }).mount('#app');
</script>
```

在以上代码中 button 按钮可以显示 info 对象中的任意一个，为了让父组件可以访问 info 对象，在< slot >元素上使用 v-bind 指令绑定一个 values 属性，称为插槽 prop，这个

prop 不需要在 props 选项中声明。

在父级作用域下使用该组件时，可以给 v-slot 指令一个值来定义组件提供的插槽 prop 的名字，代码如下：

```
<div id = "app">
    <child>
        <template v - slot:default = "slotProps">
            {{slotProps.values.book}}
        </template>
    </child>
</div>
```

因为<child>组件内的插槽是默认插槽，所以这里使用其默认的名字 default，然后给出一个名字 slotProps(自定义，随便取)，代表的是包含组件内所有插槽 prop 的一个对象，然后就可以在父组件中利用这个对象访问子组件的插槽 prop，prop 是被绑定到 info 数据属性上的，所以可以进一步访问 info 的内容，如例 5-19 所示。

【例 5-19】 作用域插槽

```
//第 5 章/作用域插槽.html
<!DOCTYPE html>
<html lang = "en">
<head>
    <meta charset = "UTF - 8">
    <title></title>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
<div id = "app">
    <child>
        <template v - slot:default = "slotProps">
            {{slotProps.values.author}}
        </template>
    </child>
</div>
<script>
    const vm = Vue.createApp({})
    vm.component('child', {
        data:function(){
          return{
              info:{
                  book:"剑指大前端全栈工程师",
                  author:"贾志杰",
                  sex:"男"
              }
          }
        },
        template: `
            <button>
```

```
                <slot v-bind:values="info">
                    {{info.book}}
                </slot>
            </button>
        `,
    }).mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-22 所示。

图 5-22　作用域插槽效果图

5.6　动态组件

让多个组件使用同一个挂载点,并动态切换,这就是动态组件。通过保留的< component >元素,动态地绑定到它的 is 特性,可以实现动态组件。它的应用场景往往应用在路由控制或者 tab 切换中。

5.6.1　基本用法

本节通过一个切换页面的例子来讲解动态组件的基本用法,如例 5-20 所示。

【例 5-20】 切换页面

```
//第 5 章/切换页面.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <button @click="change">切换页面</button>
    <component :is="currentView"></component>
</div>
<script>
    const vm = Vue.createApp({
        data(){
```

```
            return{
                index:0,
                arr:[
                    {template:'<div>我是主页</div>'},
                    {template:'<div>我是提交页</div>'},
                    {template:'<div>我是存档页</div>'}
                ],
            }
        },
        computed:{
            currentView(){
                return this.arr[this.index];
            }
        },
        methods:{
            change(){
                this.index = (++this.index) % 3;
            }
        }
    }).mount('#app');
</script>
</body>
</html>
```

在浏览器中单击“切换页面”按钮时，可以动态地获取页面信息，如图 5-23 所示。

图 5-23　切换页面显示效果图

<component>标签中 is 属性决定了当前采用的子组件，:is 是 v-bind:is 的简写，绑定了父组件中 data 的 currentView 属性。单击按钮时会更改数组 arr 的索引值，同时也会修改子组件的内容。

5.6.2　keep-alive

动态切换掉的组件(非当前显示的组件)是被移除掉了，如果把切换出去的组件保留在内存中，则可以保留它的状态或避免重新渲染。<keep-alive>包裹动态组件时会缓存不活动的组件实例，而不是销毁它们，以便提高提取效率。和<transition>相似，<keep-alive>是一个抽象组件，它自身不会渲染一个 DOM 元素，也不会出现在父组件链中，如例 5-21 所示。

【例 5-21】 keep-alive 基础用法

```
//第 5 章/ keep-alive 基础用法.html
<!DOCTYPE html>
```

```
<html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <button @click="change">切换页面</button>
    <keep-alive>
        <component :is="currentView"></component>
    </keep-alive>
</div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             index:0,
             arr:[
                {template:'<div>我是主页</div>'},
                {template:'<div>我是提交页</div>'},
                {template:'<div>我是存档页</div>'}
             ],
          }
       },
       computed:{
          currentView(){
             return this.arr[this.index];
          }
       },
       methods:{
          change(){
             let len = this.arr.length;
             this.index = (++this.index) % len;
          }
       }
    }).mount('#app');
</script>
</body>
</html>
```

如果有多个条件性的子元素,则<keep-alive>要求同时只有一个子元素被渲染时可以使用条件判断,如例5-22所示。

【例5-22】 利用条件判断缓冲子元素

```
//第5章/利用条件判断缓冲子元素.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
```

```
    <meta charset = "utf - 8"/>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
<div id = "app">
    <button @click = "change">切换页面</button>
    <keep - alive>
        <home v - if = "index === 0"></home>
        <posts v - else - if = "index === 1"></posts>
        <archive v - else></archive>
    </keep - alive>
</div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             index:0,
          }
       },
       components:{
          home:{template:'<div>我是主页</div>'},
          posts:{template:'<div>我是提交页</div>'},
          archive:{template:'<div>我是存档页</div>'},
       },
       methods:{
          change(){
             //在 data 外面定义的属性和方法通过 $options 可以获取和调用
             let len = Object.keys(this.$options.components).length;
             this.index = (++this.index) % len;
          }
       }
    }).mount('#app');
</script>
</body>
</html>
```

5.6.3 activated 钩子函数

Vue 给组件提供了 activated 钩子函数，作用于动态组件切换或者静态组件初始化的过程中。activated 函数里默认有一个参数，而这个参数也是一个函数，只有执行这个函数时，才会切换组件，从而延迟执行当前的组件，如例 5-23 所示。

【例 5-23】 activated 钩子函数

```
//第 5 章/activated 钩子函数.html
<!DOCTYPE html>
<html>
<head>
    <title></title>
```

```
    <meta charset = "utf - 8"/>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
<div id = "app">
    <button @click = 'toShow'>单击显示子组件</button>
    <!-- 或者<component v - bind:is = "which_to_show" keep - alive></component>也行 -->
    <keep - alive>
        <component v - bind:is = "which_to_show"></component>
    </keep - alive>
</div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             which_to_show: "first"
          }
       },
       methods: {
          toShow: function () {            //切换组件显示
             var arr = ["first", "second", "third", ""];
             var index = arr.indexOf(this.which_to_show);
             if (index < 2) {
                this.which_to_show = arr[index + 1];
             } else {
                this.which_to_show = arr[0];
             }
             console.log(this. $children);
          }
       },
       components: {
          first: {                         //第 1 个子组件
             template: "<div>这里是子组件 1 </div>"
          },
          second: {                        //第 2 个子组件
          template: "<div>这里是子组件 2,这里是延迟后的内容:{{hello}}</div>",
             data: function () {
                return {
                    hello: ""
                }
             },
             activated: function (done) { //只有执行这个参数时才会切换组件
                console.log('beixi')
                var self = this;
                var startTime = new Date().getTime(); //get the current time
                //两秒后执行
                while (new Date().getTime() < startTime + 2000){
```

```
                    self.hello = '我是延迟后的内容';
                }
            }
        },
        third: { //第 3 个子组件
            template: "<div>这里是子组件 3 </div>"
        }
    }
}).mount('#app');
</script>
</body>
</html>
```

单击子组件，当切换到第 2 个组件时会先执行 activated 钩子函数并在两秒后显示组件 2 的内容，从而起到了延迟加载的作用。

5.7　组件 Mixin 技术

使用组件开发的一大优势在于可以提高代码的复用性。通过 Mixin 技术，组件的复用性可以得到进一步提高。

当开发大型前端项目时，可能会定义非常多的组件，这些组件可能有部分功能是通用的，对于这部分通用的功能，如果每个组件都编写一遍，则比较麻烦，而且不易于维护，如例 5-24 所示。

【例 5-24】 定义多个组件

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <my-com1 title="组件 1"></my-com1>
    <my-com2 title="组件 2"></my-com2>
    <my-com3 title="组件 3"></my-com3>
</div>
<script>
    const vm = Vue.createApp({})
    const com1 = {
        props:['title'],
        template:`
          <div style="border:red solid 2px">
```

```
            {{title}}
          </div>
          `
    }
    const com2 = {
        props:['title'],
        template:`
          <div style = "border:red solid 2px">
            {{title}}
          </div>
          `
    }
    const com3 = {
        props:['title'],
        template:`
          <div style = "border:red solid 2px">
          {{title}}
          </div>
          `
    }
    vm.component("my-com1",com1)
    vm.component("my-com2",com2)
    vm.component("my-com3",com3)
    vm.mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-24 所示。

图 5-24　多个组件效果图

在以上示例的 3 个示例组件中，每个组件都定义了一个名为 title 的外部属性，这部分代码其实可以抽离出来，修改例 5-24 中的代码，修改后的代码如下：

```
<script>
    const vm = Vue.createApp({})
    const myMixin = {
        props:['title']
    }
    const com1 = {
```

```
            mixins:[myMixin],
            template:`
              <div style = "border:red solid 2px">
                {{title}}
              </div>
              `
        }
        const com2 = {
            mixins:[myMixin],
            template:`
              <div style = "border:red solid 2px">
                {{title}}
              </div>
              `
        }
        const com3 = {
            mixins:[myMixin],
            template:`
              <div style = "border:red solid 2px">
                {{title}}
              </div>
              `
        }
        vm.component("my-com1",com1)
        vm.component("my-com2",com2)
        vm.component("my-com3",com3)
        vm.mount('#app');
    </script>
```

5.8 组合式 API

Vue 3 的组合式 API(Composition API)是一组函数和语法糖,用于更灵活和可组合地组织 Vue 组件的代码逻辑。与 Vue 2 的选项式 API 不同,组合式 API 不再以选项的形式存在,Vue 3.0 提供了一个新的组件选项:setup。

组合式 API 算是 Vue 3 对我们开发者来讲非常有价值的一个 API 更新。注意,Vue 3 中仍可以使用选项式 API。

1. 选项式 API 和组合式 API

1) 选项式 API 的写法

在 Vue 2 中,编写组件的方式是 Options API(选项式 API)。Options API 的一大特点是在对应的属性中编写对应的功能模块,例如 data 选项定义数据、methods 选项定义方法、computed 选项定义计算属性、在 watch 选项中监听属性改变,也包括生命周期钩子。这样一来,一个功能逻辑的代码就被分散了。

(1) 优点:易于学习和使用,写代码的位置已经约定好。

(2) 缺点：

- 当实现某个功能时，这个功能对应的代码逻辑会被拆分到各个属性中。
- 这种碎片化的代码使理解和维护这个复杂的组件变得异常困难，并且隐藏了潜在的逻辑问题。
- 当组件变得更大、更复杂时，逻辑关注点的列表就会增长，那么同一个功能的逻辑就会被拆分得很分散。
- 代码组织性差，相似的逻辑代码不便于复用，逻辑复杂代码多了也不好阅读。
- 当处理单个逻辑关注点时，需要不断地跳到相应的代码块中。

2) 组合式 API 的写法

在 Vue 3.0 项目中将会使用组合 API 写法。代码风格：一个功能逻辑的代码组织在一起(包含数据、函数等)，能将同一个逻辑关注点相关的代码收集在一起会更好，这就是 Composition API 想要做的事情，以及可以帮助我们完成的事情。

(1) 优点：在功能逻辑复杂且繁多的情况下，将各个功能逻辑代码组织在一起，便于阅读和维护。

(2) 缺点：需要有良好的代码组织能力和拆分逻辑能力。

选项式 API 和组合式 API 的区别如图 5-25 所示。

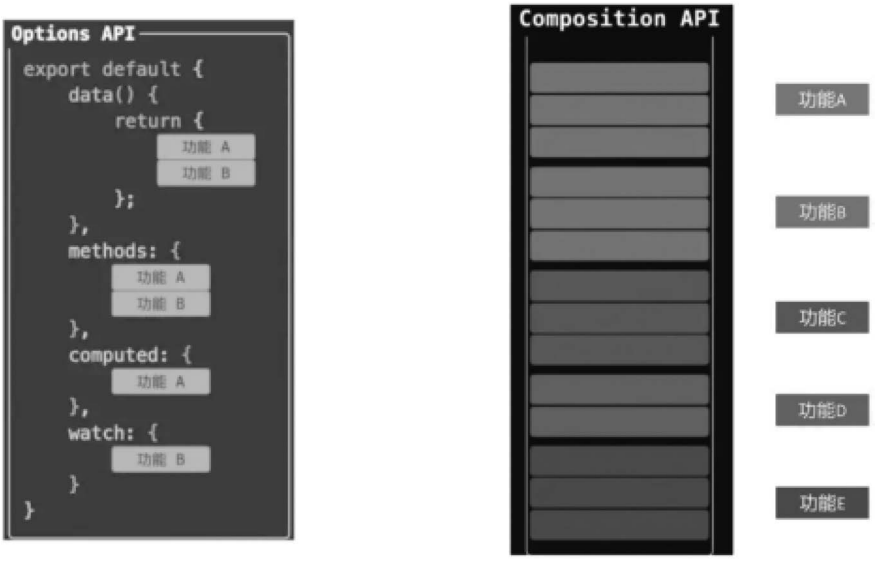

图 5-25　选项式 API 和组合式 API 的区别

组合式 API 的功能抽离、封装如图 5-26 所示。

2. 生命周期函数

Options API 和 Composition API 之间的映射，包含如何在 setup()内部调用生命周期钩子，见表 5-1。

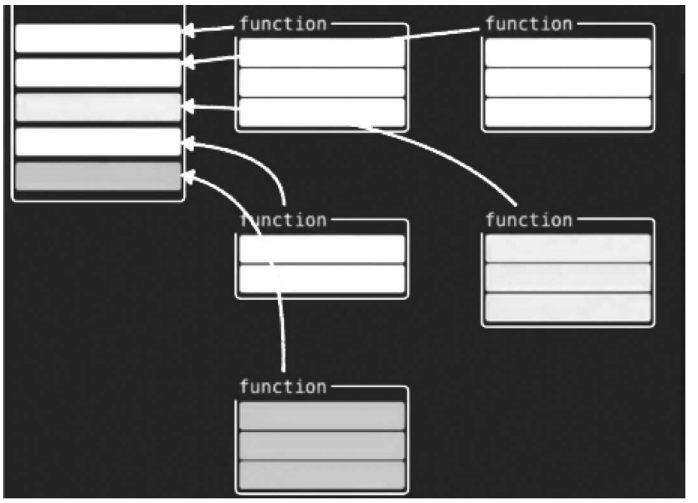

图 5-26 组合式 API 的功能抽离、封装

表 5-1 生命周期函数

选项式 API	组合式 API
beforeCreate	不需要，直接写到 setup(创建实例前)函数中
created	不需要，直接写到 setup(创建实例前)函数中
beforeMount	onBeforeMount 挂载 DOM 前
mounted	onMounted 挂载 DOM 后
beforeUpdate	onBeforeUpdate 更新组件前
updated	onUpdated 更新组件后
beforeDestroyed	onBeforeUnmount 卸载销毁前
destroyed	onUnmounted 卸载销毁后

3. 组合式 API 入门

由于组合式 API 是一套基于函数的 API，因此能够更好地与 TypeScript 集成，使用组合式 API 编写的代码可以享受完整的类型推断。

组合式 API 也可以与现有的基于选项的 API 一起使用，不过组合式 API 是在选项(data、computed、methods)之前解析，因此在组合式 API 中是无法访问这些选项定义的属性的。

setup()函数是一个新的组件选项，它作为在组件内部使用组合式 API 的入口点。setup 函数在初始的 prop 解析之后且在组件实例创建之前被调用。对于组件的生命周期钩子，setup 函数在 beforeCreate 钩子之前调用。

setup()函数主要有两个参数，分别是 props 和 context。

(1) props：props 是 setup 的第 1 个参数，该参数是响应式的，因此不能用 ES6 进行解构，因为会消除它的响应式。

(2) context：context 是 setup 的第 2 个参数，context 是 JavaScript 中的一个普通对象，它不是响应式的，可以放心地使用 ES6 结构。context 对外暴露了一些可能会用到的

值,例如 attrs、slots、emit、expose 等。

setup()函数接受的 props 对象是响应式的,也就是说,在组件外部传入新的 prop 值时,props 对象会更新,可以调用 watchEffect 或 watch 方法监听该对象并对更改作出响应,如例 5-25 所示。

【例 5-25】 setup()函数应用

```
//第 5 章/setup()函数应用.html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <script src="https://unpkg.com/vue@next"></script>
    </head>
    <body>
        <div id="app">
            <post-item :post-title="title"></post-item>
        </div>
        <script>
            const vm = Vue.createApp({
                data(){
                    return {
                        title: 'Java 无难事'
                    }
                }
          });
          vm.component('postItem', {
              //声明 props
              props: ['postTitle'],
              setup(props){
                  Vue.watchEffect(() => {
                      console.log(props.postTitle);
                  })
              },
              template: '<h3>{{ postTitle }}</h3>'
          });
          vm.mount('#app');
        </script>
    </body>
</html>
```

在浏览器中的显示效果如图 5-27 所示。

setup()函数接受的第 2 个可选参数是一个 context 对象,该对象是一个普通的 js 对象,公开了 3 个组件属性,示例代码如下:

```
vm.component('postItem', {
    //声明 props
    props: ['postTitle'],
    setup(props,context){
```

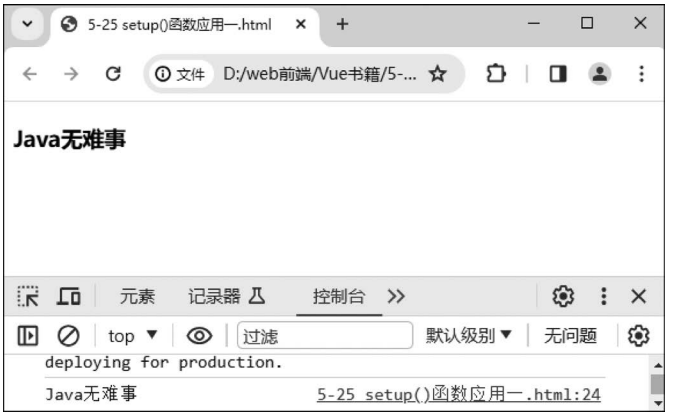

图 5-27 setup()函数

```
        //属性(非响应式对象)
        console.log(context.attrs)
        //插槽(非响应式对象)
        console.log(context.slots)
        //发出的事件(方法)
        console.log(context.emit)
        Vue.watchEffect(() => {
            console.log(props.postTitle);
        })
    },
    template: '<h3>{{ postTitle }}</h3>'
});
```

当 setup 函数和选项式 API 一起使用时,在 setup 函数内部不要使用 this。因为 setup 是在选项被解析之前调用的。也就是说,在 setup 函数内不能访问 data、computed、methods 组件选项。

5.9 响应式 API

Vue 3 的核心功能主要通过响应式 API 实现,组合式 API 将它们公开为独立的函数。

1. reactive()方法和 watchEffect()方法

reactive()方法可以对一个 JavaScript 对象创建响应式状态。在 HTML 页面中可以编写如下代码:

```
<script src="https://unpkg.com/vue@next"></script>
<script>
  //响应式状态
    const state = Vue.reactive({
       count:0
    })
</script>
```

reactive()方法相当于 Vue 2 中的 vue.observable()方法。

watchEffect()方法接收一个函数对象作为参数,它会立即运行该函数,同时响应式地跟踪其依赖项,并在依赖项发生更改时重新运行该函数。watchEffect 方法类似于 Vue 2 中的 watch 选项,但是它不需要分离监听的数据源和副作用回调。组合 API 还提供了一个 watch 方法,其行为与 Vue 2 的 watch 选项完全相同,如例 5-26 所示。

【例 5-26】 reactive()方法和 watchEffect()方法

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
        <script src="https://unpkg.com/vue@next"></script>
    </head>
    <body>
        <script>
            const {reactive, watchEffect} = Vue;
            const state = reactive({
                count: 0
            })
            watchEffect(() => {
              document.body.innerHTML = `count is ${state.count}`
          })
        </script>
    </body>
</html>
```

在浏览器中的显示效果如图 5-28 所示。按 F12 键打开控制台,输入“state.count=100”后按 Enter 键,效果如图 5-29 所示。

图 5-28 初始化状态

2. ref()方法

reactive()方法为一个 JavaScript 对象创建响应式代理,如果需要对一个原始值(如字符串)创建响应式代理对象,则一种方式是将该原始值作为某个对象的属性,调用 reactive()方法为该对象创建响应式代理对象;另一种方式就是使用 Vue 给出的另一种方法 ref(),该方法接收一个原始值,返回一个响应式和可变的 ref 对象,返回的对象只有一个 value 属性指向内部值。如例 5-27 所示。

【例 5-27】 ref()方法

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

图 5-29　响应式对象的依赖跟踪

```
    <head>
        <meta charset="UTF-8">
        <title></title>
        <script src="https://unpkg.com/vue@next"></script>
    </head>
    <body>
        <script>
            const {ref, watchEffect} = Vue;
            const state = ref(0)
            watchEffect(() => {
              document.body.innerHTML = `count is ${state.value}`
            })
        </script>
    </body>
</html>
```

在浏览器中按 F12 键打开控制台，输入“state.count＝666”后按 Enter 键，效果如图 5-30 所示。

图 5-30　使用 ref()方法

此时取值需要访问 state 对象的 value 属性。当 ref 作为渲染上下文的属性返回(从 setup 返回的对象)并在模板中访问时,它将自动展开为内部值,不需要在模板中添加 .value,更改后的示例代码如下:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
        <div id="app">
            <span>{{ count }}</span>
            <button @click="count ++">Increment count</button>
        </div>
            <script src="https://unpkg.com/vue@next"></script>
        <script>
            const {ref} = Vue;
            const app = Vue.createApp({
               setup(){
                  const count = ref(0);
                  return {
                      count
                  }
               }
            })
            app.mount('#app')
      </script>
  </body>
</html>
```

3. readonly()

有时希望跟踪响应对象(ref 或 reactive)的变化,但还希望阻止从应用程序的某个位置对其进行修改。例如,当我们有一个提供的响应式对象时,想要防止它在注入的地方发生更改,为此,可以为原始对象创建一个只读属性,示例代码如下:

```
import {readonly} from 'vue'
export default{
    name:'App',
    setup(){
        let state = readonly({name:"贾志杰",attr:{age:30,height:1.80}});
        function myFn(){
           state.name = 'zhangsan';
           state.attr.age = 18;
           state.attr.height = 1.60;
           console.log(state);                    //数据并没有变化
        }
        return {state,myFn}
    }
}
```

4. computed()方法

computed()方法与 computed 选项的作用一样，用于创建依赖于其他状态的计算属性。该方法接收一个 getter 函数，并为 getter 返回的值返回一个不可变的响应式 ref 对象，我们使用组合式 API 反转字符串，如例 5-28 所示。

【例 5-28】 computed()方法

```
<html>
    <head>
        <meta charset="UTF-8">
        <title>计算属性</title>
        <script src="https://unpkg.com/vue@next"></script>
    </head>
    <body>
        <div id="app">
            <p>原始字符串：{{ message }}</p>
            <p>反转字符串：{{ reversedMessage }}</p>
        </div>
        <script>
            const {ref, computed} = Vue;
            const vm = Vue.createApp({
                setup(){
                    const message = ref('剑指大前端全栈工程师');
                    const reversedMessage = computed(() =>
                        message.value.split('').reverse().join('')
                    );
                    return {
                        message,
                        reversedMessage
                    }
                }
            }).mount('#app');
        </script>
    </body>
</html>
```

在浏览器中的显示效果如图 5-31 所示。

图 5-31 computed()方法

5. watch()方法

watch()方法等同于 Vue 2 的 this.$watch()方法，以及相应的 watch 选项。watch()方法需要监听特定的数据源，并在单独的回调函数中应用副作用。在默认情况下，它也是惰

性的,即只有当被监听的数据源发生变化时,才会调用回调函数。

与 watchEffect()方法相比,watch 方法有以下功能:

(1) 惰性地执行副作用。

(2) 更具体地说明什么状态应该触发监听器重新运行。

(3) 访问被监听状态的前一个值和当前值。

watch()方法与 watchEffect()方法共享行为,包括手动停止、副作用失效(将 onInvalidate 作为第 3 个参数传递给回调)、刷新时间和调试。

监听的数据源可以是返回值的 getter 函数,也可以是直接的 ref 对象。示例代码如下:

```
const state = reactive({ count:0 })
//监听返回值的 getter 函数
watch(
  () => state.count,
    (count,prevCount) => {
       ...
    }
)
const count = ref(0)
//直接监听一个 ref 对象
watch(count,(count,prevCount) => {...})
```

5.10 综合案例

组件添加好后,通过单击“发表评论”按钮来将内容添加到评论列表中,如例 5-29 所示。实现的逻辑如下:

(1) 通过单击“发表评论”按钮触发单击事件调用组件中 methods 中定义的方法。

(2) 在 methods 中定义的方法中将保存的 localStorage 中的列表数据加载到 list 中。

(3) 将录入的信息添加到 list 中,然后将数据保存到 localStorage 中。

(4) 调用父组件中的方法来刷新列表数据。

【例 5-29】 综合案例

```
//第 5 章/综合案例.html
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <!-- 引入 Vue -->
    <script src="https://unpkg.com/vue@next"></script>
    <!-- 引入 Bootstrap -->
    <link rel="stylesheet" href="./lib/Bootstrap-3.3.7.css">
</head>
<body>
<div id="app">
```

```
        <cmt-box @func="loadComments"></cmt-box>
        <ul class="list-group">
            <li class="list-group-item" v-for="item in list" :key="item.id">
                <span class="badge">评论人: {{ item.user }}</span>
                      {{ item.content }}
            </li>
        </ul>
    </div>
    <template id="tmpl">
        <div>
            <div class="form-group">
                <label>评论人:</label>
                <input type="text" class="form-control" v-model="user">
            </div>
            <div class="form-group">
                <label>评论内容:</label>
                <textarea class="form-control" v-model="content"></textarea>
            </div>
            <div class="form-group">
                <input type="button" value="发表评论" class="btn btn-primary"
                    @click="postComment">
            </div>
        </div>
    </template>
    <script>
        var commentBox = {
           data() {
              return {
                 user: '',
                 content: ''
           }
           },
           template: '#tmpl',
           methods: {
              postComment() { //发表评论的方法
              var comment = { id: Date.now(), user: this.user, content: this.content }
                 //从 localStorage 中获取所有的评论
                 var list = JSON.parse(localStorage.getItem('cmts') || '[]')list.unshift(comment)
                 //重新保存最新的评论数据
                 localStorage.setItem('cmts', JSON.stringify(list))
                 this.user = this.content = ''
                 this.$emit('func')
              }
           }
        }
        //创建 Vue 实例,得到 ViewModel
        const vm = Vue.createApp({
           data() {
              return{
```

```
                list: [
                    { id: Date.now(), user: 'beixi', content: '这是我的网名' },
                    { id: Date.now(), user: 'jzj', content: '这是我的真名' },
                 { id: Date.now(), user: '贝西奇谈', content: '有任何问题可以关注公众号' }
                ]
            }
        },
        beforeCreate(){ },
        created(){
            this.loadComments()
        },
        methods: {
            loadComments() {     //从本地的 localStorage 中加载评论列表
                var list = JSON.parse(localStorage.getItem('cmts') || '[]')
                this.list = list
            }
        },
        components: {
            'cmt-box': commentBox
        }
    }).mount('#app');
</script>
</body>
</html>
```

在浏览器中的显示效果如图 5-32 所示。

图 5-32　综合案例实现效果