第3章 Vue 3基本指令

指令是Vue 3模板中常用的功能之一，它们是带有v-前缀的特殊属性。指令的主要职责是在其值发生改变时，将相应的影响作用于DOM对象。Vue 3的指令在HTML中以页面元素的属性的方式使用，指令属性的值是JavaScript表达式。Vue 3的指令数量相对较少，本章将逐一介绍这些指令。


视频讲解

3.1 条件渲染指令

条件渲染指令的主要功能是根据指令的值为true或false进而触发组件不同的表现形式。

3.1.1 v-if、v-else-if、v-else

v-if、v-else-if和v-else这三个指令用于实现条件判断。v-if根据其值有条件地渲染元素，当v-if的值在true和false之间切换时，元素或组件将被销毁或重建。在组件被销毁或重建的过程中，会执行该组件相应的钩子函数，示例代码如下：

```
<div id = "app">
 <h1 v - if = "display"> Display </h1 >
 <h1 v - if = "hide"> Hide </h1 >
 <h1 v - if = "age > = 25"> Age: {{ age }}</h1 >
 <h1 v - if = "name.indexOf('Tom')> = 0"> Name:{{name}}</h1 >
</div >

<script >
 const vm = Vue.createApp({
  data() {
   return {
    display: true,
    hide: false,
    age: 28,
    name: 'Tom Cruise'
   }
  }
 }).mount('#app');
</script >
```

在浏览器中打开上述代码组成的页面，其渲染结果如图 3.1 所示。

图 3.1　v-if 渲染结果

当 v-if 的值被设置为 hide(即为 false)时，对应的< h1 >元素并没有实际生成，而其他 v-if 的值为 true 的< h1 >元素正常生成。也就是说，当 v-if 的值为 false 时，v-if 不会创建该元素；当 v-if 的值为 true 时，v-if 才会真正创建该元素。

切换到控制台窗口，将 age 属性的值修改为 20(即 vm.age＝20)，然后切换回元素窗口，渲染结果如图 3.2 所示。

图 3.2　修改 age 属性值后 v-if 页面的渲染结果

如果需要控制多个元素的创建或删除，可以使用< template >元素将这些元素包装起来，然后在< template >元素上使用 v-if，代码如下：

```
<div id = "app">
 <template v - if = "!isLogin">
  <form>
   <p> username:< input type = "text"></p>
   <p> password:< input type = "password"></p>
  </form>
 </template>
</div>

<script>
 const vm = Vue.createApp({
  data() {
```

```
    return {
     isLogin: false
    }
   }
 }).mount('#app');
</script>
```

v-else-if 和 v-else 是 v-if 的逻辑补充。示例代码如下：

```
<div id="app">
 <div v-if="Math.random() > 0.5">
  随机数大于 0.5 时可以看到这个元素
 </div>
 <div v-else>
  随机数小于 0.5 时可以看到这个元素
 </div>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {}
  }
 }).mount('#app');
</script>
```

v-else-if 与 v-if 一起使用，可以实现互斥的条件判断，代码如下：

```
<div id="app">
 <span v-if="score >= 85">优秀</span>
 <span v-else-if="score >= 75">良好</span>
 <span v-else-if="score >= 60">及格</span>
 <span v-else>不及格
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    score: 90
   }
  }
 }).mount('#app');
</script>
```

需要注意的是，当一个条件被满足时，后续的条件判断都不会再执行，v-else-if 和 v-else 需要紧跟在 v-if 或 v-else-if 之后。

3.1.2 v-show

v-show 根据其值切换元素的 CSS 样式中的 display 属性，当条件变化时，v-show 会触发过渡效果，代码如下：

```
<div id = "app">
 <h1 v - show = "display"> Display </h1 >
 <h1 v - show = "hide"> Hide </h1 >
 <h1 v - show = "age > = 25"> Age: {{ age }}</h1 >
 <h1 v - show = "name.indexOf('Tom')> = 0"> Name:{{name}}</h1 >
</div >

<script >
 const vm = Vue.createApp({
  data() {
   return {
    display: true,
    hide: false,
    age: 28,
    name: 'Tom Cruise'
   }
  }
 }).mount('#app');
</script >
```

除了指令不同外，本节代码与 3.1.1 节中的 v-if 代码完全相同。接下来观察 DOM 结构在执行之后有何不同，v-show 测试页面的渲染结果如图 3.3 所示。

图 3.3　v-show 测试页面的渲染结果

对比图 3.1 和图 3.3 的展示效果，v-show 与 v-if 似乎没有不同，但在页面结构中可以发现，v-show 并没有根据条件不同而改变页面结构，它在 HTML 元素是否显示的实现机制上与 v-if 不同。无论 v-show 的值是 true 还是 false，v-show 都会创建元素，它通过 CSS 样式中的 display 属性来控制元素是否显示。

3.1.3　v-show 与 v-if 的选择

一般来说，v-if 有更高的切换开销，因为在切换时需要销毁和重新创建元素及其子组件，而 v-show 只需要改变 CSS 样式属性，因此在需要频繁地切换元素的显示或隐藏时，使用 v-show 更好。但在初始渲染时，v-show 存在更高的开销，因为它需要先创建元素，然后再根据其值设置 CSS 样式属性，而 v-if 只有在值为 true 时才会创建元素。因此，在条件改变较少的情况下，使用 v-if 更好。

3.2 列表渲染指令 v-for

3.2.1 基本用法

在 Vue 3 中,可以使用 v-for 基于一个数组来渲染一个列表。v-for 需要使用 item in items 形式的特殊语法,其中 items 是源数据数组,item 是被迭代的数组元素的别名,示例代码如下:

```
<ul id = "array - rendering">
 <li v - for = "item in items">
  {{ item.message }}
 </li>
</ul>
<script>
 const vm = Vue.createApp({
  data() {
   return {
    items: [{ message: 'Foo' }, { message: 'Bar' }]
   }
  }
 }).mount('#app');
</script>
```

可以看到,组件实例的数据对象中定义了一个数组 items,然后在<li>元素上使用 v-for 遍历数组,这将循环渲染<li>元素。在 v-for 块中,可以访问所有父作用域的属性,在每次循环时,item 的值为数组当前索引的值,在<li>元素内部,可以通过 Mustache 语法引用变量 item。

最终渲染结果如图 3.4 所示。

图 3.4 v-for 测试页面的渲染结果

除此之外,v-for 还支持一个可选的第二个参数,即当前项的索引,代码如下:

```
<ul id = "array - with - index">
 <li v - for = "(item, index) in items">
  {{ index }} - {{ item.message }}
```

```
 </li>
</ul>
```

在 Vue 3 中，开发者不仅可以使用 v-for 遍历数组，也可以用 v-for 来遍历一个对象的所有可枚举属性。具体的使用方法就是使用 of 替代 in 作为分隔符，示例代码如下：

```
<ul id = "v - for - object" class = "demo">
  <li v - for = "value in myObject">
    {{ value }}
  </li>
</ul>
const vm = Vue.createApp({
    data() {
    return {
        myObject: {
        title: 'How to do lists in Vue',
        author: 'Jane Doe',
        publishedAt: '2016 - 04 - 10'
        }
    }
    }
}).mount('#app');
```

可以增加第二个参数来获取属性的名称（即键名），代码如下：

```
<li v - for = "(value, name) in myObject">
 {{ name }}: {{ value }}
</li>
```

还可以增加第三个参数来获取索引，代码如下：

```
<li v - for = "(value, name, index) in myObject">
 {{ index }}. {{ name }}: {{ value }}
</li>
```

在 Vue 3 中，当使用 v-for 渲染元素列表时，默认采用“就地更新”策略。如果数据项的顺序被更改，Vue 3 将不会移动页面元素来匹配数据项的顺序，而是就地更新每个元素，并确保它们在每个索引位置都被正确渲染。为了告知 Vue 3 每个节点的身份，以便能够重用和重新排序现有元素，需要为每个项提供一个唯一的 key 值，建议在使用 v-for 时尽可能提供 key 值，这样可以提高 v-for 的渲染效率，代码如下：

```
<div v - for = "item in items" :key = "item.id">
 <!-- 内容 -->
</div>
```

3.2.2 数组更新

Vue 3 的核心是数据与视图的双向绑定，为了监测数组中元素的变化并及时将变化反映到视图中，Vue 3 对以下 7 个数组变更函数进行了封装。

(1) push()。

(2) pop()。

(3) shift()。

(4) unshift()。

(5) splice()。

(6) sort()。

(7) reverse()。

使用浏览器打开 3.2.1 节中的页面,在开发者工具中切换到控制台窗口,然后输入以下命令:

```
vm.items.push({ message: 'Baz' });
```

数组更新的结果如图 3.5 所示。

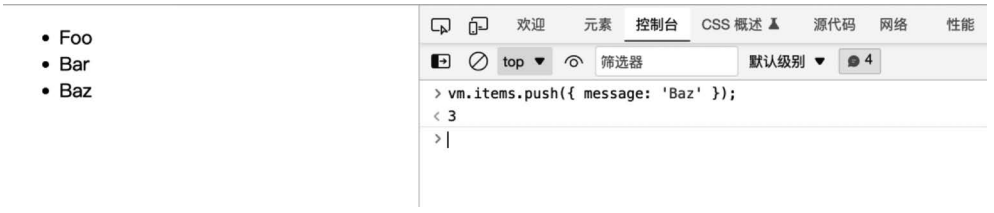

图 3.5 数组更新的结果

上述的 push()函数会改变参数中的原始数组。此外,JavaScript 语言也有原生数组的非变更函数,如 filter()、concat()和 slice(),它们不会改变原始数组,而是返回一个新数组。当使用非变更函数时,可以用新数组替换旧数组,代码如下:

```
vm.items = vm.items.filter(item => item.message.match(/Foo/));
```

数组变更的结果如图 3.6 所示。

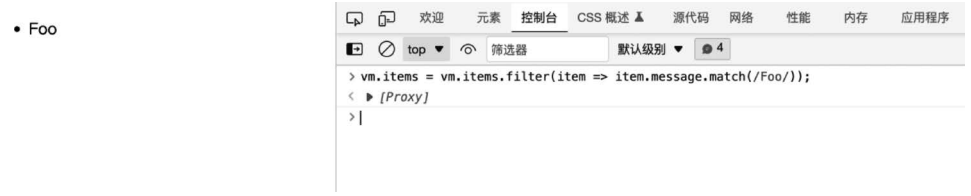

图 3.6 数组变更的结果

有些开发者担心这种操作会造成性能问题,实际上这种操作并不会导致 Vue 3 丢弃现有的页面元素并重新渲染整个列表。Vue 3 为了使页面元素得到最大范围的重用而进行了针对性的优化,用一个含有相同元素的数组去替换原来的数组是非常高效的操作。

3.2.3 v-for 的其他操作

v-for 可以用来显示数组过滤或排序后的结果,如果要显示一个数组经过过滤或排序后的版本,而不改变原始数据,可以创建一个计算属性来返回处理后的数组,代码如下:

```
<div id = "app">
 <li v - for = "n in evenNumbers" :key = "n">{{ n }}</li>
</div>

<script>
 const vm = Vue.createApp({
```

```
    data() {
     return {
      numbers: [1, 2, 3, 4, 5]
     }
    },
    computed: {
     evenNumbers() {
      return this.numbers.filter(number => number % 2 === 0)
     }
    }
   }).mount('#app');
</script>
```

如果在嵌套的 v-for 循环中无法使用计算属性，可以使用 methods()函数来解决，代码如下：

```
<div id="app">
 <ul v-for="numbers in sets">
  <li v-for="n in even(numbers)" :key="n">{{ n }}</li>
 </ul>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    sets: [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]
   }
  },
  methods: {
   even(numbers) {
    return numbers.filter(number => number % 2 === 0)
   }
  }
 }).mount('#app');
</script>
```

v-for 也可以接受整数 n 作为迭代参数，在这种情况下模板会重复循环 n 次，代码如下：

```
<div id="range" class="demo">
 <span v-for="n in 10" :key="n">{{ n }} </span>
</div>
```

页面渲染结果如图 3.7 所示。

和 v-if 类似，开发者也可以利用带有 v-for 的 <template> 来循环渲染一段包含多个元素的内容，代码如下：

```
<ul>
 <template v-for="item in items" :key="item.msg">
  <li>{{ item.msg }}</li>
  <li class="divider" role="presentation"></li>
```

图 3.7 v-for 接受整数的页面渲染结果

```
 </template>
</ul>
```

当 v-for 与 v-if 同时使用时,需要注意当它们处于同一节点时,v-if 的优先级比 v-for 更高,这意味着 v-if 将没有权限访问 v-for 中的变量,代码如下:

```
<!-- 这将抛出一个错误,因为"todo" property 没有在实例上定义 -->
<li v-for="todo in todos" v-if="!todo.isComplete">
 {{ todo.name }}
</li>
```

为了解决这个问题,可以把 v-for 移动到 <template> 标签中来修正,代码如下:

```
<template v-for="todo in todos" :key="todo.name">
 <li v-if="!todo.isComplete">
  {{ todo.name }}
 </li>
</template>
```

在自定义组件上,开发者可以像在任何普通元素上一样使用 v-for,代码如下:

```
<my-component v-for="item in items" :key="item.id"></my-component>
```

然而,任何数据都不会被自动传递到组件里,因为组件有自己独立的作用域。为了把迭代数据传递到组件里,需要使用如“:props 名称”的组件属性来传递数据,代码如下:

```
<my-component
 v-for="(item, index) in items"
 :item="item"
 :index="index"
 :key="item.id"
></my-component>
```

3.3 数据绑定指令 v-bind


视频讲解

v-bind 的主要作用是动态更新 HTML 元素上的属性和动态绑定组件的 props 属性，也可以使用简写的符号“:”来代替它。

3.3.1 参数与属性绑定

下面示例中链接的 href 属性通过 v-bind 动态地设置，当数据发生变化时，组件会被重新渲染，代码如下：

```
<div id = "app">
 <a v - bind:href = "url">前往百度</a>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    url: 'https://www.baidu.com'
   }
  }
 }).mount('#app');
</script>
```

3.3.2 动态绑定

示例代码如下：

```
<div id = "app">
 <a v - bind:[attribute] = "url">前往百度</a>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    attribute:'href',
    url: 'https://www.baidu.com'
   }
  }
 }).mount('#app');
</script>
```

与 3.3.1 节中的代码相比，此处将在 HTML 中的属性变成了动态获取。v-bind 还可以直接绑定一个包含属性名和值的对象。在这种情况下，v-bind 指令不需要接收参数就可以直接使用，代码如下：

```
<div id = "app">
 <!-- 绑定一个有属性的对象 -->
```

```
 <form v-bind="formObj">
  <input type="text">
 </form>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    formObj: {
     method: 'get',
     action: '#'
    }
   }
  }
 }).mount('#app');
</script>
```

最终的渲染结果如图 3.8 所示。

图 3.8 v-bind 绑定对象的渲染结果

3.3.3 v-bind 的缩写及合并行为

Vue 3 提供了一个简写方式“:bind”,如果一个元素同时定义了 v-bind="object" 和一个相同的独立属性,后定义的属性值会覆盖之前定义的同名属性值。因此开发者可以通过控制它们的合并行为以满足开发需求,代码如下:

```
<!-- 模板 -->
<div id="red" v-bind="{ id: 'blue' }"></div>
<!-- 结果 -->
<div id="blue"></div>

<!-- 模板 -->
<div v-bind="{ id: 'blue' }" id="red"></div>
<!-- 结果 -->
<div id="red"></div>
```

3.4 v-model 与表单

3.4.1 基本用法

v-model 用于在表单中的< input >、< textarea >和< select >元素上创建双向数据绑定，根据控件类型自动选取正确的方法来更新元素，负责监听用户的输入事件从而更新数据，并对一些极端场景进行特殊处理，代码如下：

```
<div id = "app">
 <input type = "text" v - model = "message">
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    message: 'Hello World'
   }
  }
 }).mount('#app');
</script>
```

渲染效果如图 3.9 所示。

图 3.9 v-model 渲染结果

在开发者工具的控制台窗口中，输入 vm.message ='Welcome to the Vue world'，可以看到 v-model 绑定的表达式数据发生改变，导致页面元素的值随之改变，如图 3.10 所示。

图 3.10 改变表达式数据导致页面元素值改变

接下来在页面控件元素中随意输入一些内容，然后在控制台中输入 vm.message，可以看到表达式数据 message 的值也发生了变化，如图 3.11 所示。

图 3.11 改变页面元素值导致表达式数据改变

3.4.2 值绑定

针对不同的表单控件，v-model 绑定的值都有默认的约定。例如，单个复选框绑定的是布尔值，多个复选框绑定的是一个数组，选中的复选框 value 属性的值被保存到数组中。如果要改变默认的绑定规则，可以使用 v-bind 把值绑定到当前活动实例的一个动态属性上，这个属性的值可以不是字符串。

下面介绍 3 种常用的表单元素是如何绑定值的。

(1) 复选框。在使用单个复选框时，在< input >元素上可以使用两个特殊的属性 true-value 和 false-value 来指定选中状态下和未选中状态下 v-model 绑定的值，代码如下：

```
<div id = "app">
 <input id = "agreement" type = "checkbox" v - model = "isAgree" true - value = "yes" false -
value = "no">
 <label for = "agreement">{{isAgree}}</label>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    isAgree: false
   }
  }
 }).mount('#app');
</script>
```

数据属性 isAgree 的初始值为 false，当选中复选框时，其值为 true-value 的属性值 yes，当取消选中复选框时，其值为 false-value 的属性值 no。true-value 属性和 false-value 属性也可以使用 v-bind 绑定到 data 选项中的某个数据属性上。

(2) 单选按钮。单选按钮被选中时，v-model 绑定的数据属性的值默认被设置为该单选按钮的 value 值。可以使用 v-bind 将< input >元素的 value 属性再绑定到另一个数据属性上，选中后的值就是这个 value 属性绑定的数据属性的值，代码如下：

```
<div id = "app">
 <input id = "male" type = "radio" v - model = "gender" :value = "genderVal[0]">
 <label for = "male">男</label>
```

```
 <br>
 <input id="female" type="radio" v-model="gender" :value="genderVal[1]">
 <label for="female">女</label>
 <br>
 <span>性别：{{gender}}</span>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    gender: '',
    genderVal: ['男', '女']
   }
  }
 }).mount('#app');
</script>
```

图 3.12　使用 v-bind 后单选按钮选中的值

运行效果如图 3.12 所示。

（3）选择框选项。通过选择框选择内容后，其值是选项的值，即< option >元素的 value 属性的值，选项的 value 属性也可以使用 v-bind 指令绑定到一个数据属性上，代码如下：

```
<option v-for="option in options" v-bind:value="option. value"></option>
```

或者将 value 属性绑定到一个对象字面量上，当选项被选中时，vm. selected. number 的值会变更为 2023，代码如下：

```
<select v-model="selected" title="select">
 <!-- 内联对象字面量 -->
 <option v-bind:value="{number:2022}">2023</option>
</select>
```

3.4.3　修饰符

修饰符主要有以下 3 种。

（1）trim 修饰符。它用于自动过滤用户输入内容首尾两端的空格，使用 v-model 时，代码如下：

```
<input type="text" v-model="inputValue">
<p>{{ inputValue }}</p>
<input type="text" v-model.trim="inputValue">
<p>{{ inputValue }}</p>
```

运行效果如图 3.13 所示。

可以看到，当使用 trim 修饰符后，< p >标签的前后空格和中间多余的空格都被去除了，只显示输入框中的实际内容。

（2）lazy 修饰符。它用于将 v-model 的默认触发方式由 input 事件更改为 change 事件。例如，在使用< input >元素时，每次输入内容都会立即更新数据，使用 lazy 修饰符后，v-model 的双向数据绑定触发方式就变为失去焦点时进行内容检测，从而减少了频繁更新数据的操作，代码如下：

图 3.13 trim 修饰符应用

```
<input type = "text" v - model.lazy = "inputValue">
<p>{{ inputValue }}</p>
```

(3) number 修饰符。它用于自动将用户输入的数据转换为数值类型,如果无法被 parseFloat() 转换,则返回原始内容,代码如下:

```
<input type = "text" v - model.number = "inputValue">
<p>{{ inputValue }}</p>
```

3.5 方法、计算属性与监听属性

3.5.1 Vue 3 中的方法

Vue 3 方法是与 Vue 3 实例关联的对象,本书后续提到的 Vue 3 方法特指 methods 对象。当需要对元素的某些事件做出响应时,可以通过 v-on 来绑定相应的 Vue 3 方法,开发者可以在 Vue 3 方法内定义函数来执行事件响应的操作,下面演示 Vue 3 方法的工作原理,代码如下:

```
<div id = "app">
 <!-- 渲染 DOM 树 -->
 <h1 style = "color: seagreen;">{{title}}</h1>
 <h2>Title : {{name}}</h2>
 <h2>Topic : {{topic}}</h2>
 <!-- 调用 Vue 3 方法中的函数 -->
 <h2>{{show()}}</h2>

</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    title: "Geeks for Geeks",
    name: "Vue.js",
```

```
    topic: "Instances"
   }
  },
  // 创建组件中的 Vue 3 方法
  methods: {
   // 创建函数
   show: function () {
    return "欢迎尝试这个 Vue 例子 "
      + this.name + " - " + this.topic;
   }
  }
 }).mount('#app');
</script>
```

运行效果如图 3.14 所示。

图 3.14　Vue 3 方法工作原理运行效果

3.5.2　计算属性

在模板中使用表达式非常方便，但如果表达式的逻辑比较复杂，使用计算属性会大大降低模板的复杂度，代码如下：

```
<div id = "app">
 <p>{{message. split('). reverse(). join(')}}</p>
</div>
```

Mustache 语法中的表达式调用了 3 个函数来最终实现字符串的反转，逻辑过于复杂，如果在模板中还需要多次引用这个功能，会让模板变得更加复杂并难以处理，这时就可以使用计算属性，它以函数形式在 computed 选项中定义，代码如下：

```
<div id = "app">
 <div id = "app">
  <p>原始字符串：{{message }}</p>
  <p>计算后的反转字符串：{{reversedMessage}}</p>
 </div>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    message: 'Hello!, welcome to Vue!'
   }
  },
  // 创建计算属性
  computed: {
   //计算属性的 getter
   reversedMessage() {
    return this.message.split('').reverse().join('');
   }
  }
}).mount('#app');
</script>
```

在上述示例中，声明了一个计算属性 reversedMessage，它可以像普通的属性一样在模板中绑定数据。在浏览器中的渲染结果如图 3.15 所示。

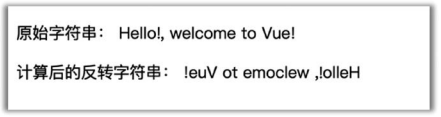

图 3.15 计算属性绑定数据的渲染结果

当 message 属性的值改变时，reversedMessage 的值会自动更新，并且会自动同步更新相应 DOM 对象。在浏览器中修改 vm. message 的值，reversedMessage 的值也会随之改变。

由于计算属性默认只有 getter 方法，因此不能直接修改计算属性，如需修改可以参考以下代码：

```
<div id = "app">
 <div id = "app">
  <p > First name:< input type = "text" v - model = "firstName"></p >
  <p > Last name:< input type = "text" v - model = "lastName"></p >
  <p >{{ fullName }}</p >
 </div >
</div >

<script >
 const vm = Vue.createApp({
  data() {
   return {
    firstName: 'Smith',
    lastName: "Will"
   }
  },
  // 创建计算属性
  computed: {
   fullName: {
    // getter()函数
    get() {
     return this.firstName + '' + this.lastName;
    },
    // setter()函数
    set(newValue) {
     let names = newValue.split(' ');
     this.firstName = names[0];
     this.lastName = names[names, length1];
    }
   }
  }
 }).mount(' #app');
</script >
```

任意修改 firstName 或 lastName 的值，fullName 的值也会自动更新，这是调用 fullName 的 get( ) 实现的。在浏览器的 Console 窗口中输入 vm. fullName = " Bruce Willis"，可以看到 firstName 和 lastName 的值也同时发生了改变，这是调用 fullName 的 set()实现的。

3.5.3 监听属性

为了观察和响应实例上的数据变化，当需要一些数据随着其他数据变化而变化时，可以使用监听属性。尽管这听起来和计算属性的作用很相似，但在实际应用中两者是有很大差别的。Vue 3 实例的选项对象通常是在 watch 选项中定义监听属性。下面的代码演示了如何使用监听属性来实现千米和米之间的换算。

```
<div id = "app">
 <div id = "app">
  千米:< input type = "text" v - model = "kilometers">
  米:< input type = "text" v - model = "meters">
 </div >
</div >

< script >
 const vm = Vue.createApp({
  data() {
   return {
    kilometers: 0,
    meters: 0
   }
  },
  // 监听属性
  watch: {
   kilometers(val) {
    this.meters = val * 1000;
   },
   meters(val, oldVal) {
    this.kilometers = val / 1000;
   }
  }
 }).mount(' #app');
</script >
```

在这个例子中编写了两个监听函数，分别监听数据属性 kilometers 和 meters 的变化。当其中一个数据属性的值发生改变时，对应的监听函数就会被调用进而经过计算得到另一个数据属性的值。注意，不要在监听函数中使用箭头函数，这样会造成 Vue 3 的上下文发生错乱。

当需要在数据变化时执行异步或开销较大的操作时，使用监听属性是最合适的。例如在一个在线问答系统中，用户输入的问题需要到服务器端获取答案，就可以考虑对问题属性进行监听，在异步请求答案的过程中，可以设置中间状态，向用户提示“请稍候...”，而这样的功能使用计算属性无法做到。

下面给出一个使用监听属性实现斐波那契数列的计算的例子，该计算比较耗时，因此采用 HTML 5 新增的 WebWorker 来计算，将斐波那契数列的计算放到一个单独的 fibonacci.js 文件中，代码如下：

```
function fibonacci(n) {
```

```
 return n < 2 ? n : arguments.callee(n - 1) + arguments.callee(n - 2);
}
onmessage = function (event) {
 var num = parseInt(event.data, 10);
 postMessage(fibonacci(num));
}
```

主页面代码如下：

```
<div id = "app">
 <span >请输入要计算斐波那契数列的第几个数: </span >
 <input type = "text" v - model = "num">
 <p v - show = "result">{{result}}</p >
</div >

<script >
 const vm = Vue.createApp({
  data() {
   return {
    num: 0,
    result: ''
   }
  },
  // 监听属性
  watch: {
   num(val) {
    this.result = "请稍候...";
    if (val > 0) {
     const worker = new Worker("fibonacci.js");
     worker.onmessage = (event) => this.result = event.data; worker.postMessage(val);
    }
    else {
     this.result = '';
    }
   }
  }
 }).mount('#app');
</script >
```

在这个例子中，为了防止用户在等待计算时以为程序卡死，为 result 数据属性设置了一个中间状态，从而给用户一个提示。worker 实例是异步执行的，当后台线程完成任务后通过 postMessage()函数来调用，并通过调用创建者线程的 onmessage()回调函数来通知。在此回调函数中，可以通过 event 对象的 data 属性来获取数据。在这种异步回调执行的过程中，this 的指向会发生变化。如果 onmessage()回调函数写成以下形式：

```
worker.onmessage = function (event) { this.result = event.data };
```

那么在执行 onmessage()函数时，this 实际上指向的是 worker 对象，因此 this. result 的值是 undefined。为了解决这个问题，需要在 onmessage()函数处使用箭头函数，因为箭头函数绑定的是父级作用域的上下文，这里绑定的是 vm 对象。在使用 Vue 3 进行开发时，经常会遇到 this 指向的问题，合理地使用箭头函数可以避免很多问题，后面还会遇到类似的情

况。在浏览器中打开页面，输入 40，会看到提示信息“请稍候...”，过一段时间后会显示斐波那契数列的第 40 个数。

当定义监听属性时，除直接编写一个函数外还可以指定一个 Vue 3 方法名，代码如下：

```
<div id = "app">
 年龄：<input type = "text" v - model = "age">
 <p v - if = "info">{{info}}</p>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    age: 0,
    info: ''
   }
  },
  methods: {
   checkAge() {
    if (this.age >= 18)
     this.info = '已成年';
    else
     this.info = '未成年';
   }
  },
  watch: {
   age: 'checkAge'
  }
 }).mount('#app');
</script>
```

监听属性还可以监听一个对象的属性变化，代码如下所示：

```
<div id = "app">
 年龄：<input type = "text" v - model = "person.age">
 <p v - if = "info">{{info}}</p>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    person: {
     name: 'lisi',
     age: 0
    },
    info: ''
   }
  },
  watch: {
   //该回调会在 person 对象的属性改变时被调用，无论该属性被嵌套得多深
```

```
    person: {
     handler(val, oldVal) {
      if (val.age >= 18)
       this.info = '已成年';
      else
       this.info = '未成年';
     },
     deep: true
    }
   }
  }).mount('#app');
</script>
```

需要注意的是,在监听对象属性变化时,使用了两个新选项:handler 和 deep。handler 用于定义数据变化时调用的监听属性函数,而 deep 主要用于监听对象属性的变化。如果将 deep 的值设置为 true,无论对象属性在对象中的层级有多深,只要该属性的值发生变化,都会被监测到。

监听属性函数在初始渲染时不会被调用,只有在后续监听的属性发生变化时才会被调用。如果要在开始监听后立即执行监听属性函数,可以使用 immediate 选项,并将其值设置为 true,代码如下:

```
watch: {
 //该回调会在 person 对象的属性改变时被调用,无论该属性被嵌套得多深
 person: {
  handler(val, oldVal) {
   if (val.age >= 18)
    this.info = '已成年';
   else
    this.info = '未成年';
  },
  deep: true,
  immediate: true
 }
}
```

如果仅需要监听对象的一个属性变化,且变化不影响对象的其他属性,那么可以直接监听该对象的属性,代码如下:

```
watch: {
 'person.age':function(val,oldVal){
  ...
 }
}
```

除了在 watch 选项中定义监听属性,还可以使用组件实例的 $watch()函数观察组件实例上响应式属性或计算属性的更改。注意,对于顶层数据属性、prop 和计算属性,只能以字符串形式传递名字,代码如下:

```
vm.$watch('kilometers', (newValue, oldValue) => {
 //这个回调将在 vm.kilometers 改变后调用
 document.getElementById("info").innerHTML = "修改前值为: " + old Value + "修改后值为:
```

```
" + newValue;
})
```

对于更复杂的表达式或嵌套属性，则需要使用函数形式，代码如下：

```
const app = Vue.createApp({
   data() {
    return {
     a: 1,
     b: 2,
     c: {
      d: 3,
      e: 4
     }
    }
   },
   created() {
    // 顶层属性名
    this.$watch('a', (newVal, old Val) => {

    })
    //用于监听单个嵌套属性的函数
    this.$watch(
     () => this.c.d,
     (newVal, oldVal) => {

     }
    )
    // 用于监听复杂表达式的函数
    this.$watch(
     // 每次表达式"this.a + this.b"产生不同的结果时
     // 都会调用处理程序，就好像在观察一个计算属性而没有定义计算属性本身
     () => this.a + this.b,
     (newVal, oldVal) => {

     }
    )
   }
  })
```

$watch()函数还可以接受一个可选的选项对象作为其第 3 个参数。例如，要监听对象内部嵌套属性的变化，可以在选项参数中传入 deep:true，代码如下：

```
vm.$watch('person', callback, {
 deep: true
});
```

3.5.4 方法、计算属性与监听属性的区别

在 Vue 3 中，可以通过在表达式中调用 Vue 3 方法来达到与监听属性相同的效果，代码如下：

```
<p>Reversed message: "{{ reversedMessage() }}"</p>
// 在组件中
methods: {
 reversedMessage: function () {
  return this.message.split('').reverse().join('')
 }
}
```

Vue 3 方法和计算属性的区别在于，计算属性是基于响应式依赖进行缓存的，只有在相关响应式依赖发生改变时才会重新求值，只要 message 还没有发生改变，多次访问 reversedMessage 计算属性就会立即返回之前的计算结果，而不会再次执行函数。这意味着下面代码中的计算属性 now 将不再更新，因为 Date.now() 不是响应式依赖：

```
computed: {
 now: function () {
  return Date.now()
 }
}
```

假设有一个计算属性 A 需要遍历一个庞大的数组并进行大量的计算，且可能有其他的计算属性依赖 A，缓存的意义在于系统可避免多次执行 A 的 getter()函数。当某些数据需要随着其他数据的变化而变化时，建议使用计算属性，代码如下：

```
<div id="demo">{{ fullName }}</div>
var vm = new Vue({
 el: '#demo',
 data: {
  firstName: 'Foo',
  lastName: 'Bar',
  fullName: 'Foo Bar'
 },
 watch: {
  firstName: function (val) {
   this.fullName = val + '' + this.lastName
  },
  lastName: function (val) {
   this.fullName = this.firstName + '' + val
  }
 }
})
```

上述代码是命令式的且存在重复代码，采用计算属性的代码如下：

```
var vm = new Vue({
 el: '#demo',
 data: {
  firstName: 'Foo',
  lastName: 'Bar'
 },
 computed: {
  fullName: function () {
   return this.firstName + '' + this.lastName
```

```
  }
 }
})
```

可以看出，计算属性的版本在可读性和逻辑性上更好。

3.6 事件处理

3.6.1 监听事件 v-on

v-on 用于绑定事件监听器，也可以使用简写的符号“@”来代替它。v-on 后面的参数指定了监听的事件类型，参数值可以是一个函数的名称或内联语句，如果没有使用修饰符，那么可以省略。在普通元素上使用 v-on 时，只能监听原生的 DOM 事件。而在自定义元素组件上使用 v-on 时，可以监听子组件触发的自定义事件，示例代码如下：

```
<div id = "app">
 <div>
  <button v - on:click = "counter += 1"> Add 1 </button>
  <p>按钮已经点击了 {{ counter }} 次.</p>
 </div>
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    counter: 0
   }
  }
 }).mount('#app');
</script>
```

当点击按钮时，点击次数都会加 1，运行效果如图 3.16 所示。

图 3.16 v-on 监听点击事件示例

3.6.2 事件处理函数

在实际开发中由于事件处理逻辑通常比较复杂，因此不建议直接将 JavaScript 代码写在 v-on 中。v-on 还可以接收一个用于调用的函数名称，示例代码如下：

```
<div id = "app">
  <!-- greet 是在下面定义的函数名 -->
  <button v - on:click = "greet"> Greet </button>
</div>
const vm = Vue.createApp({
   data() {
    return {
     name: 'Vue.js'
    }
   },
```

```
    // 在 methods 对象中定义函数
    methods: {
     greet: function (event) {
      // this 在函数里指向当前 Vue 实例
      alert('Hello ' + this.name + '!')
      // event 是原生 DOM 事件
      if (event) {
       alert(event.target.tagName)
      }
     }
    }
  }).mount('#app');
```

3.6.3 内联处理函数

除了直接绑定到一个函数，还可以在内联 JavaScript 语句中调用函数，代码如下：

```
<div id = "app">
 <button v - on:click = "say('hi')"> Say hi </button>
 <button v - on:click = "say('what')"> Say what </button>
</div>
const vm = Vue.createApp({
  data() {
   return {}
  },
  methods: {
   say: function (message) {
    alert(message)
   }
  }
 }).mount('#app');
```

渲染结果如图 3.17 所示，这种方式不会在 v-on 所在元素上出现对应的 JavaScript 事件属性。

图 3.17 v-on 内联渲染结果

3.6.4 多事件监听

在 Vue 3 中，可以使用 v-on 来绑定事件，有时一个标签上需要绑定多个事件，可以逐一绑定，也可以使用 v-on 一次绑定多个不同的事件，代码如下：

```
< input v - on = "{focus:focus,blur:blur}"></input >
const vm = Vue.createApp({
   data() {
    return {}
   },
   methods: {
    blur() {
     console.log("输入框失去焦点");
    },
    focus() {
     console.log("输入框获取焦点");
    }
   }
  }).mount('#app');
```

input 获取焦点和 input 失去焦点的运行效果分别如图 3.18 和图 3.19 所示。

图 3.18　input 获取焦点的运行效果

图 3.19　input 失去焦点的运行效果

3.6.5　事件修饰符

在事件处理程序中调用 event.preventDefault() 或 event.stopPropagation()是常见需求，虽然可以在函数中实现，但更好的方式是让函数只处理纯粹的数据逻辑，而不是去处理 DOM 事件细节。为此，Vue 3 给 v-on 提供了事件修饰符，修饰符是由点开头的指令后缀来表示的，主要有以下 6 个。

(1).stop。

(2).prevent。

(3).capture。

(4).self。

(5).once。

(6).passive。

示例代码如下：

```
<!-- 阻止点击事件继续传播 -->
<a v-on:click.stop="doThis"></a>

<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- 修饰符可以串联 -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>

<!-- 添加事件监听器时使用事件捕获模式 -->
<!-- 即内部元素触发的事件先在此处理,然后才交由内部元素进行处理 -->
<div v-on:click.capture="doThis">...</div>

<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->
<!-- 即事件不是从内部元素触发的 -->
<div v-on:click.self="doThat">...</div>

<!-- 点击事件将只会触发一次 -->
<a v-on:click.once="doThis"></a>

<!-- 滚动事件的默认行为(即滚动行为)将会立即触发 -->
<!-- 而不会等待 onScroll 完成 -->
<!-- 这其中包含 event.preventDefault()的情况 -->
<div v-on:scroll.passive="onScroll">...</div>
```

使用事件修饰符时,顺序非常重要,因为相应的代码也会按照同样的顺序生成。使用v-on:click.prevent.self会阻止所有的点击事件,而v-on:click.self.prevent只会阻止对元素本身的点击事件。不要同时使用.prevent和.passive修饰符,因为.prevent修饰符会被忽略,同时浏览器可能会显示一个警告,.passive修饰符会告诉浏览器不要阻止事件的默认行为。

3.6.6 按键修饰符

Vue 3允许为v-on在监听键盘事件时添加按键修饰符,代码如下:

```
<!-- 只有在 key 是 Enter 时调用 vm.submit() -->
<input v-on:keyup.enter="submit">
<input v-on:keyup.page-down="onPageDown">
```

为了在必要的情况下支持旧浏览器,Vue 3提供了绝大多数常用的按键码的别名,主要有以下9个:

(1).enter。

(2).tab。

(3).delete。

(4).esc。

(5).space。

（6）.up。

（7）.down。

（8）.left。

（9）.right。

一些按键（如 .esc 和所有方向键）在 IE 9 中的 key 值与其他浏览器不同，如果需要支持 IE 9，则应该使用这些内置的别名，还可以通过全局的 config.keyCodes 对象自定义按键修饰符的别名，代码如下：

```
// 可以使用 v-on:keyup.f1
Vue.config.keyCodes.f1 = 112
```

3.6.7 系统修饰键

在 Vue 3 中还可以使用以下 4 个系统修饰符来实现仅在按相应按键时才触发鼠标或键盘事件的监听器。

（1）.ctrl。

（2）.alt。

（3）.shift。

（4）.meta。

注意，在 macOS 系统键盘上，.meta 对应 command 键（⌘）。在 Windows 系统键盘上，.meta 对应 Windows 徽标键（⊞）。在 Sun 操作系统键盘上，meta 对应实心宝石键（◆）。在其他特定键盘上，尤其是在 MIT 和 Lisp 机器的键盘及其后继产品（如 Knight 键盘、space-cadet 键盘），meta 被标记为“META”。在 Symbolics 键盘上，.meta 被标记为“META”或者“Meta”，示例代码如下：

```
<!-- Alt + C -->
<input v-on:keyup.alt.67="clear">

<!-- Ctrl + Click -->
<div v-on:click.ctrl="doSomething">Do something</div>
```

在同时使用修饰键和 keyup 事件时，只有在按 Ctrl 键的情况下释放其他按键才能触发 keyup.ctrl 事件，如果只释放 Ctrl 键不会触发事件。要实现这种行为，需要使用 keyCode 将 keyup.ctrl 替换为 keyup.17。

Vue 3 内有 1 个特别的.exact 修饰符，可以和系统修饰符搭配进行更精确的控制。.exact 修饰符允许控制仅当精确的系统修饰符按键按下时触发事件，代码如下：

```
<!-- 即使 Alt 键或 Shift 键被一同按下时也会触发 -->
<button v-on:click.ctrl="onClick">A</button>

<!-- 有且只有 Ctrl 键被按下的时候才触发 -->
<button v-on:click.ctrl.exact="onCtrlClick">A</button>

<!-- 没有任何系统修饰符被按下的时候才触发 -->
<button v-on:click.exact="onClick">A</button>
```

3.7 其他基本指令

3.7.1 首次渲染 v-once

v-once 可以使元素或组件只被渲染一次，该指令不需要赋值，在之后的重新渲染中元素或组件及其所有子节点将被视为静态内容并跳过，可用于优化更新性能，代码如下：

```
<div id = "app">
 <h1 >{{title}}</h1 >
 <a v - for = "nav in navs" :href = "nav. url" v - once >{{nav.name}}</a >
</div >
<script src = " https://unpkg.com/vue@next "></script >
<script >
 const vm = Vue.createApp({
  data() {
   return {
    title: 'v - once 的用法',
    navs: [
     { name: '首页', url: '/home' },
     { name: '新闻', url: '/news' },
     { name: '视频', url: '/video' },
    ]
   }
  }
 }).mount(' #app');
</script >
```

图 3.20 v-once 渲染结果

渲染结果如图 3.20 所示。

虽然看起来和没有使用 v-once 的渲染结果是相同的，但是 v-once 在首次渲染时不会生成动态绑定的代码，这有助于提高渲染性能。可以在控制台中输入以下语句并按回车键：

```
vm.navs.push({name:'论坛', url:'/bbs'})
```

如图 3.21 所示，可以发现页面没有任何变化。

图 3.21 修改 navs 数组的内容的渲染结果

3.7.2 使用 v-cloak 避免渲染时闪烁

示例代码如下所示：

```
<div id = "app">
 <h1 v - cloak >{{message}}</h1 >
```

```
</div>

<script>
 const vm = Vue.createApp({
  data() {
   return {
    message: '渲染结束可见'
   }
  }
 }).mount('#app');
</script>
```

可以发现，当浏览器加载页面时，如果网速较慢或者页面较大，会出现 DOM 树构建完成后直接显示{{message}}的情况，直到 Vue 3 的 JavaScript 文件加载完毕、组件实例创建和模板编译后，才会将{{message}}替换为数据对象中的内容。这个过程中，页面会出现闪烁，用户体验较差。

为了解决这个问题，可以使用 v-cloak，v-cloak 会一直保留在元素上，直到与其关联的 Vue 3 实例完成编译。当与 CSS 规则 v-cloak{ display：none } 一起使用时，该指令可以隐藏未编译的 Mustache 语法，直到实例准备就绪。

在 Vue 3 独立版本的页面开发中，使用 v-cloak 解决初始化慢所导致的页面闪烁非常有效。但是在较大的项目中，由于采用模块化开发，主页面只包含一个空的 div 元素，剩余的内容是由路由挂载不同的组件完成的，因此没有必要使用 v-cloak。

3.8 自定义指令

除了 Vue 3 内置的核心功能指令（如 v-model 和 v-show）外，Vue 3 还允许注册自定义指令。

3.8.1 注册自定义指令

假设需要开发一个输入框，当页面加载时该输入框元素将获得焦点，只要用户在打开这个页面后还没点击过任何内容，这个输入框就仍处于聚焦状态。可以通过自定义指令来实现这个功能，代码如下：

```
// 注册一个全局自定义指令 v-focus
Vue.directive(focus, {
 // 当被绑定的元素插入 DOM 树中时……
 inserted: function (el) {
  // 聚焦元素
  el.focus()
 }
})
```

如果想要注册局部指令，可以在组件选项中添加 directives 属性，代码如下：

```
directives: {
 focus: {
  // 指令的定义
```

```
  inserted: function (el) {
   el.focus()
  }
 }
}
```

然后开发者可以在模板中任何元素上使用新的 v-focus 属性。

3.8.2 钩子函数

一个指令定义对象可以提供以下 5 个钩子函数(均为可选)。

(1) bind：只调用一次,指令第一次绑定到元素时调用,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(仅保证父节点存在,但不一定已被插入文档中)。

(3) update：所在组件的 VNode 更新时调用。

(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。

3.8.3 动态指令参数

指令的参数可以是动态的,例如在 v-mydirective：[argument]="value" 中,argument 参数可以根据组件实例数据进行更新,这使得自定义指令可以在应用中被灵活使用。

例如,可以创建一个自定义指令,用于通过指令值更新垂直坐标,从而将元素以绝对坐标固定在页面上,代码如下：

```
<div id = "baseexample">
  <p>向下滚动页面</p>
  <p v-pin = "200">固定在距离页面顶部 200px 的位置</p>
</div>
Vue.directive('pin', {
 bind: function (el, binding, vnode) {
  el.style.position = 'fixed'
  el.style.top = binding.value + 'px'
 }
})

new Vue({
 el: '#baseexample'
})
```

这会把该元素固定在距离页面顶部 200px 的位置,但如果场景是需要把元素固定在左侧或顶部,则需要使用动态参数根据每个组件实例来进行更新,代码如下：

```
<div id = "dynamicexample">
 <h3>在此区域内向下滚动</h3>
 <p v-pin:[direction] = "200">固定在左侧 200px 的地方</p>
</div>
Vue.directive('pin', {
 bind: function (el, binding, vnode) {
  el.style.position = 'fixed'
```

```
    var s = (binding.arg == 'left'? 'left' : 'top')
    el.style[s] = binding.value + 'px'
  }
})

new Vue({
  el: '#dynamicexample',
  data: function () {
    return {
      direction: 'left'
    }
  }
})
```

3.8.4 函数简写与对象字面量

在许多情况下，开发者可能希望在调用 bind 和 update 钩子函数时触发相同的行为，而不关心其他钩子函数。如果指令需要多个值，可以传入一个 JavaScript 对象字面量，指令函数可以接受所有合法的 JavaScript 表达式，代码如下：

```
<div v-demo="{ color: 'white', text: 'hello!' }"></div>
Vue.directive('demo', function (el, binding) {
  console.log(binding.value.color)          // 输出 white
  console.log(binding.value.text)           // 输出 hello!
})
```

3.9 案例

3.9.1 使用自定义指令实现随机背景色

有时需要在网页中将一张图片作为某个元素的背景图，但当网络状况较差或图片较大时，图片的加载速度可能会很慢。在这种情况下可以先使用随机的背景色填充该元素的区域，等待图片加载完成后再将元素的背景替换为图片。使用自定义指令可以很方便地实现上述功能，代码如下：

```
<div id="app">
  <div v-img="'images/bg.jpg'"></div>
</div>

<script>
  const app = Vue.createApp({});
  app.directive('img', {
    mounted: function (el, binding) {
      let color = Math.floor(Math.random() * 1000000);
      el.style.backgroundColor = '#' + color;
      let img = new Image();
      img.src = binding.value;
      img.onload = function () {
        el.style.backgroundImage = 'url(' + binding.value + ')';
```

```
    }
   }
  })
  app.mount('#app');
</script>
```

3.9.2 注册登录页面信息

在 SPA 中,用户注册时通常会使用 Ajax 将数据以 JSON 格式发送到服务器端。JSON 是 JavaScript 对象字面量语法的子集。在表单提交前通常需要将要发送的数据组织为一个 JavaScript 对象或数组,然后将其转换为 JSON 字符串进行发送。使用 Vue 3 将数据组织为对象的过程非常简单。可以使用 v-model 将输入控件直接绑定到某个对象的属性上,然后使用 v-on 绑定“注册”按钮的 click 事件,在事件处理函数中直接发送该对象即可,完整代码如下所示:

```
<div id = "app">
 <form>
  <table border = "0">
   <tr>
    <td>用户名: </td>
    <td>
     <input type = "text" name = "username" v - model = "user.username">
    </td>
   </tr>
   <tr>
    <td>密码: </td>
    <td>
     <input type = "password" name = "password" v - model = "user.password">
    </td>
   </tr>
   <tr>
    <td>性别: </td>
    <td>
     <input type = "radio" name = "gender" value = "1" v - model = "user.gender">男
     <input type = "radio" name = "gender" value = "0" v - model = "user.gender">女
    </td>
   </tr>
   <tr>
    <td>邮件地址: </td>
    <td>
     <input type = "text" name = "email" v - model = "user.email">
    </td>
   </tr>
   <tr>
    <td>密码问题: </td>
    <td>
     <input type = "text" name = " pwdQuestion " v - model = "user.pwdQuestion">
    </td>
   </tr>
   <tr>
    <td>密码答案: </td>
    <td>
```

```
        <input type="text" name="pwdAnswer" v-model="user.pwdAnswer">
      </td>
     </tr>
     <tr>
      <td>
       <input type="submit" value="注册" @click.prevent="register">
      </td>
      <td><input type="reset" value="重填"></td>
     </tr>
    </table>
   </form>
  </div>

  <script>
   const vm = Vue.createApp({
    data() {
     return {
      user: {
       username: '',
       password: '',
       gender: '',
       email: '',
       pwdQuestion: '',
       pwdAnswer: ''
      }
     }
    },
    methods: {
     register: function () {
      //直接发送 this.user 对象
      // ...
      console.log(this.user);
     }
    }
   }).mount('#app');
  </script
```

在“注册”按钮上绑定 click 事件时使用.prevent 修饰符来阻止表单的默认提交行为，这是因为本案例是在 click 事件响应函数中完成用户注册数据的发送，并不希望发生表单的默认提交行为，浏览器中注册页面的显示效果如图 3.22 所示。

图 3.22　注册页面的显示效果

3.10 本章小节

本章详细介绍了 Vue 3 的内置指令。其中，常用的指令包括 v-if、v-for、v-on、v-bind 和 v-model。读者应该重点掌握这 5 个指令的使用方法。此外，本章还介绍了自定义指令的用法，自定义指令只应用于对 DOM 对象的封装操作。在某些特殊需求下，通过自定义指令封装 DOM 对象操作可以简化代码编写，提高代码的重用性。

习题

1. 描述 v-once 的作用。
2. 描述计算属性和监听属性的异同。
3. 实现一个修改密码页面。