第3章

Vue.js渐进式框架

Vue.js是一个渐进式JavaScript框架。本章主要介绍了Vue基本原理、基础语法、组合式和响应性函数、事件绑定及触发、自定义元素、组件、渲染函数、聚合器、Pinia状态管理等内容。熟练掌握Vue.js,将为开发基于响应式数据驱动的Web前端页面提供强大支撑。

3.1 Vue概述

Vue.js简称Vue(官方拼读/vju:/,发音类似于英文单词view),是一个专注于视图层数据展示的数据驱动、渐进式JavaScript框架,官网地址为https://vuejs.org/。所谓数据驱动,就是只需要改变数据,Vue就会自动渲染并展示新内容页面。渐进式的意思则是可以分阶段、有选择性地使用Vue,无须使用其全部特性,小到输出"Hello,Vue!"的简单页面,大到复杂的Web应用系统,增量使用。

很多人注意到Vue是一种渐进式框架,却对其核心特性之一的响应性有所忽略。在第2章中已经学习到RxJS是基于JavaScript进行响应式逻辑处理,Vue的底层也是采用响应式编程的概念思想来构建的。Vue能跟踪数据的变化并即时响应式地更新页面,实现无缝的用户体验,体现出类拔萃的响应式数据管理能力。

Vue易于学习,很容易与其他JavaScript库集成使用,在实践中得到了广泛应用,受到业界极高的关注,被誉为JavaScript热门框架的三大巨头之一。Vue为高效率开发Web应用系统的前端页面处理提供了强大动力。

为更方便地使用Vue,建议在IntelliJ IDEA中安装Vue.js插件,如图3-1所示。

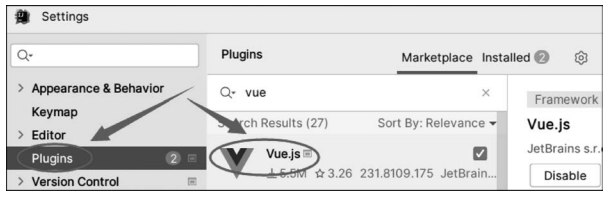

图3-1 Vue.js插件

3.2 Vue 应用基础

3.2.1 创建 Vue 应用

1. 安装

可以采用 NPM(不适合非 Node.js 环境),或在页面引入 CDN 地址方式,例如:

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
```

但这种方式无法使用单文件组件 SFC 的语法。还有一种是下载 JS 文件存入项目文件夹的方式,以 3.3.10 版本为例,下载链接为

```
https://cdnjs.cloudflare.com/ajax/libs/vue/3.3.10/vue.global.prod.min.js
```

vue.global.prod.min.js 是用于项目生产环境的压缩版。在浏览器源码界面上右击另存到项目的 js 文件夹下,然后在页面中引用:

```
<script src="js/vue.global.prod.min.js"></script>
```

这种方式可以直接在脚本中使用全局构建对象 Vue,无须做其他任何安装配置。本书采用这种方式,并使用 3.3.10 版本。

提示:也可到 BootCDN 中文网下载,请参阅 2.1.2 节的内容。

2. 第一个 Vue 应用

下面来创建第一个 Vue 应用,在页面上输出“你好,Vue!”。为便于理解,这个示例给出了完整代码。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>hello Vue</title>
    <script src="js/vue.global.prod.min.js"></script>                    ①
</head>
<body>
<div id="app">                                                           ②
    <span>{{welcome}}</span>
</div>
<script>
    const app = Vue.createApp({                                          ③
        data() {                                                         ④
            return {
                welcome: '你好,Vue!'
            }
        }
    })
    app.mount('#app')                    //将 Vue 应用实例挂载到 id="app"的层上面
</script>
</body>
</html>
```

① 引入 Vue 的支撑库，这个不要忘记了。后面不再重复提示。

② 定义了一个层，用来挂载 Vue 应用的实例，通俗地说，就是让 Vue 来渲染这个层里面所有元素的数据及事件处理。打个比方就是，这个层类似于某个名叫“app”的公司，现在请 Vue 来接管该公司，对公司的各种数据业务（设备、资金、人事等）进行管理。所以，这个层里面标题为< span >的数据，将由 Vue 渲染显示。

③ 利用全局对象 Vue 的 createApp()函数，创建一个 Vue 应用的实例。Vue 会完成一系列初始化过程，例如，设置数据、编译模板等，并运行一些被称为“生命周期钩子”的函数。利用这些钩子函数，就可以在适当的时候处理各种业务逻辑。这有点类似于去坐车（id＝"app"的层），司机（Vue）要运行绕车检查、检查车辆、打火启动、发车等“钩子函数”，而我们（层的内部元素）则可以在司机起步检查时处理自己的业务逻辑“调整坐姿”“系好安全带”。

④ 通过 data()函数返回了一个 welcome 数据。当然，也可以返回多个数据。

还有一种创建 Vue 应用的方式：

```
<div id = "app">
    <span>{{welcome}}</span>
</div>
<script>
    const MyData = {
        data() {
            return {
                welcome: '你好,Vue!'
            }
        }
    }
    Vue.createApp(MyData).mount('#app')
</script>
```

这种方式基于 MyData 组件创建 Vue 应用，运行效果跟第一种方式完全相同。

3.2.2 生命周期

从创建 Vue 应用实例，到初始化、模板编译，再到渲染、挂载 DOM 结点，然后更新界面，最后销毁实例，完成了整个生命周期过程。读者可以去 Vue 官网参阅详细的生命周期流程图，有助于更好地理解 Vue 的整个运行原理。

在整个生命周期过程中，Vue 提供了 8 个重要的钩子函数。下面选取了其中 4 个比较常用的钩子函数做简要说明。

（1）created()：Vue 实例创建完成后调用。此时一些数据和函数已经创建好，但还没有挂载到页面上。可在这个函数里面进行一些初始化处理工作。

（2）mounted()：模板挂载到 Vue 实例后调用，HTML 页面渲染已经完成。可以在这个函数中开始业务逻辑处理。

（3）beforeUpdate()：页面更新之前被调用。此时数据状态值已经是最新的，但并没有更新到页面上。

（4）beforeDestroy()：解除组件绑定、事件监听等。在实例销毁之前调用。

示例：*在待输出数据“你好，Vue!”后加上“渐进式框架!”字样。*

```
const app = Vue.createApp({
```

```
    data() {
        return {
            welcome: '你好,Vue!'
        }
    },
    mounted() {
        this.addStr()
    },
    methods: {
        addStr: function () {
            this.welcome += '渐进式框架!'
        }
    }
})
app.mount('#app')
```

自行编写的若干函数,可以放置在 methods 里面。这里利用 mounted()钩子函数,调用 addStr()函数,对原有的数据进行修改。打开页面后,将显示“你好,Vue! 渐进式框架!”。

提示:要使用数据或函数,需要用 this 进行限定,this 代表当前 Vue 实例。

3.2.3　组合式函数 setup()

为了更好地管理代码,Vue 提供了组合式函数 setup()。setup()能够很好地将代码整合在一起,还可以有选择性地对外暴露我们定义的变量、常量和函数,基本语法格式如下。

```
const app = Vue.createApp({
    props: {
        addr: {type: String}
    },
    setup(props, context) {
        …
    }
})
```

- props:外部传入的数据,可以是数组或对象,这里传入的是一个 addr 字符串数据。后面结合数据绑定知识再举例。
- context:上下文对象,可用来获取应用的一些属性,context 和 props 都是可选参数。

示例:用 setup()函数,重写 3.2.2 节的例子。

```
const app = Vue.createApp({
    setup() {
        const welcome = Vue.ref('你好,Vue!')
        const addStr = () => welcome.value += '渐进式框架!'
        Vue.onMounted(addStr)
        return {welcome}
    }
})
app.mount('#app')
```

较 3.2.2 节写法,现在简洁明了很多。在 setup()函数里面,生命周期钩子函数的写法有所变化,原来的 mounted()变成了 onMounted(),类似的还有 onBeforeMount()、onBeforeUpdate()等。

Vue 是一个全局量,里面定义了很多常量、函数或方法,例如,ref()、onMounted()等。在 Vue.onMounted()里面调用 addStr()函数,返回 welcome 对象添加内容后的值。

通过 setup()函数的 return,可以有选择性地对外暴露某些值或方法。这里如果不返回 welcome,< span >中是无法插值显示 welcome 的值的。读者可能对代码中的 Vue. ref()感到疑惑,这是一个响应性函数,下面将会介绍。

提示:推荐使用 setup()函数组合式写法! 注意,setup()函数里面不能使用 this。

3.2.4 插值

语法格式:

```
{{message}}
```

插值属于 Vue 的模板语法,是数据绑定最常见的形式。可以简单理解为:将< script >脚本中的 message 变量(或常量)的内容"插入"到 HTML 指定位置。实际上,这是一种数据绑定,页面绑定了 message 数据属性的值。当 message 的值发生改变时,插值处的内容会自动发生改变。在前面的示例中其实已经使用过了。

在插值里面,甚至可以使用 JavaScript 表达式(不能是语句)进行某些处理。例如:

```
< div id = "app">
    < span >
        {{welcome. indexOf('Vue') > 0 ? welcome + '我要努力学习之!' : welcome}}
    </span >
</div >
< script >
    Vue. createApp({
        setup() {
            const welcome = Vue. ref('你好,Vue!')
            return {welcome}
        }
    }). mount('#app')
</script >
```

这里的插值使用了三目条件运算来输出不同字符串。现在,页面显示内容将变成"你好,Vue! 我要努力学习之!"。

3.2.5 响应性函数

在 Vue 应用中,当脚本中的数据(例如,服务器推送数据,或用户手工改变)被修改时,页面就会自动发生改变,体现了高度的响应性。要为对象创建响应性状态,需要使用 ref()、reactive()等函数。

1. ref()

ref()是 Vue 响应式 API 的核心函数。如果希望将字符串'你好,Vue!'变成响应式对象,则可使用 ref()函数。ref()会创建一个值为"你好,Vue!"的字符串对象,并将该字符串"包裹"在一个带有 .value 属性的 ref 对象中返回。因此,ref 返回对该响应式对象(只包含一个 value 属性)的引用,ref 一般用于结构较简单的类型。例如下面的 welcome:

```
const welcome = Vue. ref('你好,Vue!')
```

字符串'你好,Vue!'被包裹在一个带有.value 属性的 ref 对象中返回。因此,若要访问 welcome 的值,则需要通过". value"的形式:

```
console.log('welcome 的值:' + welcome.value)
welcome.value += '努力践行!'
```

这里的“.value”形式是指在<script>脚本中。在 HTML 中访问 welcome 的值，并不需要这个“.value”。

```
<div id = "app">
    <span>{{welcome}}</span>
</div>
```

对于常见的简单数据类型，如 string、number 等，一般习惯性使用 ref。但是，实际上 ref 适用于任何类型的值。除了简单数据类型外，ref 也适用于数组、对象等数据结构。只不过，如果将一个非简单数据类型的对象赋值给 ref，例如：

```
const welcome = Vue.ref({title:'你好,Vue!'})
```

ref()函数的内部会通过 reactive()将其转换为响应式代理，这将使该对象变成一种深层次的响应式对象。“深层次的响应式对象”意味着当嵌套对象的值发生改变时，变化能被 Vue 监测到并可反映到页面效果上，例如：

```
const welcome = Vue.ref({
    author:{
            no: '23070156',
            name:'张三丰',
            caption: '响应式项目'
        },
    address: '杭州教工路 10 号'
})
```

welcome 所代理的是一个具有嵌套子对象 author 的对象。如果改变 author 的属性值：

```
welcome.value.author.no = '23070158'
welcome.value.author.name = '杨过'
welcome.value.author.caption = '响应式项目开发实战'
welcome.value.address = '武汉解放大道 10 号'
```

子对象属性值的变化会被 Vue 监测到并影响页面效果，体现了深层次响应性。读者可能注意到，当比较多的地方都需要使用 welcome 时，如果每次都要通过“.value”的形式存取数据，显然比较烦琐！这时候可通过代理函数 proxyRefs()进行简化。来看下面的示例。

```
<div id = "app">
    主题 - {{welcome.author.caption}} 地址 - {{welcome.address}}
    <button id = "resetCapAddr">重置主题和地址</button>
</div>
<script>
    Vue.createApp({
        setup() {
            const welcome = Vue.ref({
                author: {
                    no: '23070156',
                    name: '张三丰',
                    caption: '响应式项目'
                },
                address: '杭州教工路 10 号'
            })
```

```
            Vue.onMounted(() => {
                const {fromEvent, tap} = rxjs
                const proxyWelcome = Vue.proxyRefs(Vue.unref(welcome))          ①
                fromEvent(document.querySelector("#resetCapAddr"), 'click').pipe(
                    tap(() => {                                                  ②
                        proxyWelcome.author.caption = '响应式项目开发实战'
                        proxyWelcome.address = '武汉解放大道 10 号'
                    })
                ).subscribe()
            })
            return {welcome}
        }
    }).mount('#app')
</script>
```

① 函数 proxyRefs()创建了一个对目标对象(welcome 所指引的对象)的代理(Proxy)访问。代理 Proxy 类似于在目标对象之前架设了一层“捕捉器”,外界对目标对象的访问都必须先通过这层“捕捉器”,这有利于保证目标对象原始信息的安全性。该函数首先判断传递过来的参数值是否是响应式对象:如果是,则直接返回该对象;否则,利用 new Proxy()函数对参数值进行“包裹”后返回。那为什么 proxyWelcome 获取数据不再需要“.value”?因为这里使用了 unref()函数进行解除处理。

② 通过代理对象 proxyWelcome,改变响应对象 welcome 所“包裹”对象的 author、address 的值,以便观察代理对象 proxyWelcome 的属性值改变时,是否会影响到 welcome。从运行结果来看,这种改变会直接反映到 welcome。当然,使用代理对象 proxyWelcome 后,就不再需要每次带上“.value”了,代码简洁了很多。也可以“return { proxyWelcome}”,并在< div >中使用对应插值:

```
主题 - {{proxyWelcome.author.caption}} 地址 - {{proxyWelcome.address}}
```

效果是一样的。那么,如果希望放弃深层次响应性,怎么办?可使用 shallowRef()。

提示:不要忘了引入 RxJS 支撑库文件< script src="js/rxjs.umd.min.js"></script >。

2. shallowRef()

shallowRef()函数是 ref()函数的浅层作用形式,其内部值以原值形式存储和暴露,无法从顶层到子层,递归地转换为响应式。仍以上面的重置主题和地址为例,这里不再需要使用 proxyRefs()代理函数了。如果修改按钮的 click 事件代码,直接像下面这样赋值,不会触发任何变化,是没有效果的。

```
welcome.value.author.caption = '响应式项目开发实战'
welcome.value.address = '武汉解放大道 10 号'
```

因为这种浅层作用形式,不会将对象内部属性的访问处理成响应式,而只有顶层才是响应式的,即只有对目标对象 welcome 的“.value”进行赋值,才是响应式的,如下。

```
...
const welcome = Vue.shallowRef({
   author: {
       no: '23070156',
       name: '张三丰',
       caption: '响应式项目'
   },
   address: '杭州教工路 10 号'
```

```
})
const {fromEvent, tap} = rxjs
fromEvent(document.querySelector("#resetCapAddr"), 'click').pipe(
    tap(() => {
        const tmp = {
            author: {
                no: '23070158',
                name: '杨过',
                caption: '响应式项目开发实战'
            },
            address: '武汉解放大道10号'
        }
        welcome.value = {...tmp}
    })
).subscribe()
…
```

这里构造了一个临时对象tmp，通过对象展开运算符，将tmp的值赋值给welcome.value，将触发页面变化，达到了目的。

3. reactive()

reactive()函数不像ref()函数那样对对象进行“包裹”，而是直接使得对象本身就具有响应性。例如：

```
const addr = Vue.reactive({
    webSocket: 'ws://192.168.1.5',
    role: 'student',
    endpoint: 'schat'
})
```

代码创建了一个有三个属性的响应对象addr。究其本质，reactive()是对原数据对象(reactive()函数括号中的内容)的代理(Proxy)，即Vue将数据对象包装在代理中，而代理对象可检测属性的更新或删除，依赖于addr的全部界面就会自动更新以反映这些更改。

reactive()常用在创建需要具备响应性状态的结构较复杂的类型上，例如，对象、数组、列表等。一般不建议用于原始数据类型，如number、boolean、string等。与ref()不同，若要访问addr的值，并不需要采用“addr.value”的形式，直接获取即可。

```
<div id="app"><span>{{url}}</span></div>
<script>
    Vue.createApp({
        setup() {
            const addr = Vue.reactive({
                webSocket: 'ws://192.168.1.5',
                role: 'student',
                endpoint: 'schat'
            })
            const url = addr.webSocket + '/' + addr.role + '/' + addr.endpoint
            return {url}
        }
    }).mount('#app')
</script>
```

与前文所述的shallowRef()一样，reactive()也有对应的浅层作用形式shallowReactive()，其原理类似，在此不再赘述。

3.2.6 解构

读者可能对前面代码中的诸如“Vue.createApp”“Vue.ref”“Vue.onMounted”“Vue.reactive”等使用方式感到烦琐。确实如此！使用解构，就可写得更为简便。解构，通俗一点的说法就是将一些属性、函数或方法从某个定义中“抠”出来以方便使用。例如，没有解构前：

```
<div id="app"><span>{{str}}</span></div>
<script>
    Vue.createApp({
        setup() {
            const guoy = Vue.ref({
                name: '杨过',
                email: 'guoy@126.com'
            })
            const str = guoy.value.name + ';' + guoy.value.email
            return {str}
        }
    }).mount('#app')
</script>
```

解构后，访问形式就简单些。

```
<div id="app"><span>{{str}}</span></div>
<script>
    const {createApp, ref} = Vue                                    //解构
    createApp({
        setup() {
            const guoy = ref({
                name: '杨过',
                email: 'guoy@126.com'
            })
            const {name, email} = guoy.value                        //解构
            const str = name + ';' + email
            return {str}
        }
    }).mount('#app')
</script>
```

从全局对象 Vue 中解构出 createApp()、ref()函数，从 guoy 中解构出 name、email 属性。而 3.2.3 节的 setup()函数则可修改成这样：

```
setup() {
    const {ref, onMounted} = Vue
    const welcome = ref('你好,Vue!')
    onMounted(() => welcome.value += '渐进式框架!')
    return {welcome}
}
```

解构，是一个很方便的做法。以后，也可以将一些对象、函数等放在一个专门的自定义集合器里面，统一管理。当需要使用某个函数或对象的时候，从集合器中解构出来即可。

3.3 基础语法

Vue 使用了基于 HTML 的模板语法。模板泛指任何合法的 HTML。Vue 将模板编译并渲

染成虚拟 DOM。

3.3.1　模板语法

Vue 的指令，通常以 v-为前缀，如 v-html、v-bind、v-if、v-show 等。

1. v-html

通常情况下，插值语句会将里面的内容解读为纯文本。如果前面的示例中 message 的内容是这样的：

```
'你好,<b>Vue!</b>'
```

页面上将会原样输出上述内容，而不是期望的加粗的“Vue!”。v-html 指令可以实现 HTML 内容的解析输出：

```
<span v-html="welcome.indexOf('Vue') > 0 ? welcome + '我要努力学习之!' : welcome"></span>
```

现在，页面上的“Vue!”将粗体显示。

2. v-bind

绑定指令 v-bind 日常使用非常频繁。v-bind 用于绑定数据以便动态更新 HTML 元素的状态。当绑定的表达式的值发生改变时，这种改变将实时应用到页面元素上。例如：

```
<img v-bind:src="myimg">
```

可在 setup()函数里面定义 myimg。

```
const myimg = Vue.ref('image/username.png')
return {myimg}
```

页面上将显示 username.png 图片。下面再举一个稍微综合点的示例：一个能够输入文字的文本框。①要求文本框的文字颜色、边框、阴影的样式能够组合变化，例如，可只设定边框、文字颜色，或者只设定文字颜色、阴影样式。②某些场景下改变的是文本框的 size 属性，而另外一些场景下改变的则是 maxlength 属性。因为文本框的 size 是设定外观长度的，而 maxlength 则表示在文本框内允许输入的最大字符长度，所以具体需要设置哪个属性，要求能够灵活变化。

上面这些处理需求，显然是需要动态绑定属性的。下面先来简单定义三个 CSS 样式名称。

```
<style>
    .myColor { color: #00f; }
    .myBorder { border: 1px dotted #f00; }
    .myShadow { box-shadow: 2px 2px 2px #a9a5a5; }
</style>
```

然后，在页面上放置一个包含文本框、按钮的层。

```
<div id="app">
    <input v-bind:class="myStyle" v-bind:[attr]="10" value="你好,Vue!"/>
     <button id="changeStyle">单击改变样式</button>
</div>
```

最后，使用 Vue 进行处理。

```
<script>
    const {createApp, reactive, ref, onMounted} = Vue
    createApp({
```

```
        setup() {
            const myStyle = reactive({
                myColor: true,                                  //颜色
                myBorder: false,                                //边框
                myShadow: false                                 //阴影
            })
            const attr = ref('size')                            //默认值为文本框的外观长度
            onMounted(() => {
                const {fromEvent, tap} = rxjs
                fromEvent(document.querySelector("#changeStyle"), 'click').pipe(
                    tap(() => {
                        myStyle.myColor = !myStyle.myColor      //原值的否定
                        myStyle.myBorder = !myStyle.myBorder
                        myStyle.myShadow = !myStyle.myShadow
                        attr.value = attr.value === 'size'? 'maxlength' : 'size'
                    })
                ).subscribe()
            })
            return {myStyle, attr}
        }
    }).mount('#app')
</script>
```

单击按钮时，myStyle 对象三个属性的值会改变为原值的否定，文本框的样式就会发生组合式变化。

v-bind:[attr]中 attr 是一种动态参数，是根据响应数据 attr 的不同值设置不同的属性。单击按钮时，attr 的值在 size、maxlength 之间切换。当 attr 的值切换到 maxlength 时，意味着文本框中总计最多只能输入 10 个字符。

3. 语法糖

v-前缀对标识 Vue 行为很有帮助，但也稍显烦琐，Vue 为 v-bind 和 v-on 这两个最常用的指令提供了语法糖来简化写法，直接写一个“:”号：

```
<input :class="myStyle" :[attr]="10" value="你好,Vue!"/>
```

4. style 绑定

有时候，对于一些 HTML 元素的简单修饰，常用 style 直接定义，而非采用 CSS 文件方式，例如：

```
<span style="color:#f00">你好</span>
```

Vue 提供了方便的 style 样式绑定处理，先在层里面添加一个< span >：

```
<div id="app">
    <span :style="[style1,style2]">{{welcome}}</span>
</div>
```

这个< span >叠加绑定了两个样式对象：style1、style2。再看 setup()函数代码：

```
setup() {
    const {ref, reactive} = Vue
    const welcome = ref('你好,Vue!')
    const style1 = reactive({
        background: '#090',
        borderRadius: '3px'
```

```
    })
    const style2 = reactive({
        fontWeight: 'bold',
        color: welcome.value.indexOf('Vue') > 0 ? '#fff' : '#f5c609',
        fontSize: '1.2em',
        textShadow: '3px 3px 3px #605d5d'
    })
    return {welcome, style1, style2}
}
```

读者可能觉得用CSS直接修饰效果也一样。确实如此。不过请注意粗体代码，这里是由Vue根据情况进行控制，有些场景就需要这种处理。读者可运行看看最终效果。

5. template

字符串模板template在运行时即时编译，可用于显示HTML内容，该模板将会替换已经挂载的HTML元素。

示例：用template显示两句古诗。

```
<div id = "app">
    <span>{{welcome}}</span>
</div>
<script>
    Vue.createApp({
        setup() {
            const welcome = Vue.ref('你好,Vue!')
            return {welcome}
        },
        template:
                `<div>
                    <img src = "image/username.png" width = "32" height = "32">
                    <span>时人不识凌云木,直待凌云始道高</span>
                 </div>`
    }).mount('#app')
</script>
```

页面上不会显示“你好，Vue!”，而是显示一张图片和那两句诗。

提示：template内容中 前面的引号不是英文的单引号“'”，也不是中文的“‘”，而是键盘数字1旁边的“`”符号！

3.3.2　计算属性 computed

computed常用于简单计算。当数据发生改变时，计算属性会自动重新执行并刷新页面数据。下面通过示例来学习其应用。

示例1：对count进行赋值、取值运算。

```
setup() {
    const {ref, computed} = Vue
    const count = ref(5)
    const comp = computed({
        get: () => ++count.value,
        set: val => count.value = val - 20
    })
    console.log(count.value)                ①
```

```
        console.log(comp.value)          ②
        comp.value = 10
        console.log(count.value)         ③
        console.log(comp.value)          ④
        return {count}
}
```

get()函数用于获取 count 的值：将 count 的值加 1 后返回。而 set()函数则设置 count 的值：将 count 的值减去传递过来的参数值 val。

① 浏览器控制台输出 5。数据并没发生改变，所以日志输出 5。

② 输出 6。通过 get()获取到计算结果。count 的值先加 1，返回给 comp。此时，count 的值也为 6。如果是后加“count.value＋＋”，则输出 5，然后 count 的值仍然是 6。

③ 输出－10。给 comp 赋值为 10，将触发 set()函数进行计算，传递给 val 的值为 10，所以最终输出为－10。

④ 输出－9。同前面一样，因为触发 get()函数得到的计算结果为(－10＋1)。

如果在页面用插值显示 count 的值：

```
<span>{{count}}</span>
```

这时候会显示多少？请读者考虑。

示例 2：计算图书总金额。

```
<div id="app">
    总金额:<span>{{total}}</span>
</div>
<script>
    Vue.createApp({
        setup() {
            const {reactive, computed} = Vue
            const books = reactive([
                {
                    bname: 'Java Web 开发教程',
                    price: 48.9,
                    count: 135
                },
                {
                    bname: 'PostgreSQL 技术及应用',
                    price: 58.9,
                    count: 140
                }
            ])
            const total = computed(() => {
                let sum = 0
                books.forEach(book => sum += book.price * book.count)
                return sum
            })
            return {total}
        }
    }).mount('#app')
</script>
```

代码定义了一个响应式数组 books，然后迭代 books 中的每个对象，计算总金额，返回一个具体值给 total。

3.3.3　侦听 watch

对指定的数据源进行侦听，以便做出反应。基本用法：

```
const count = ref(0)
watch(count, oldCount => {
    //做出反应，执行处理
    }
)
```

在这个基本用法里面，是对单个数据 count 进行侦听，一旦其值发生变化，就自动进行相应的处理。

示例 1：单击按钮，依次输出数字“0　1　3　6　10　15　21　28　36…”。

```
<div id = "app">{{num}} <button id = "add">单击增加</button>
</div>
<script>
    Vue.createApp({
        setup() {
            const {ref, watch, onMounted} = Vue
            const num = ref(0)                                    //输出的数字
            const count = ref(0)                                  //每次增加的数字
            watch(count, oldCount => num.value += oldCount)       //侦听 count 的变化
            onMounted(() => {
                const {fromEvent, tap} = rxjs
                fromEvent(document.querySelector("#add"), 'click').pipe(
                    tap(() => count.value++)                      //单击改变 count 值，激发侦听事件
                ).subscribe()
            })
            return {num}
        }
    }).mount('#app')
</script>
```

示例 2：某图书仓库默认有 275 本书。每次单击数字文本框的上下小箭头时，可增减 1 本图书。如果是单击向上的小箭头，增加一本随机生成的样本图书（书名、价格随机模拟生成）；如果是单击向下的小箭头，则删除最后入库的那本图书。文本框下方会实时变化增加的图书数及总金额。当图书数恢复到默认的 275 本时，禁止减少图书，即单击向下小箭头无效，此时单击向上小箭头才有效果。另外，为避免输错，禁止用户在文本框中输入数字。效果如图 3-2 所示。

图 3-2　侦听图书变化

看起来处理有点复杂，但利用 Vue 的响应式技术，结合计算属性、watch 侦听，并没有想象得那么复杂。首先，在页面放置一个层：

```
<div id = "app">
    图书数:<input type = "number" id = "bookInput"
                  v-model = "inBooks.count" min = "0"/><br>
    <span>{{inBooks.message}}</span> 
    总金额:<span>{{inBooks.total}}</span>
</div>
```

层里面的<input>数字型文本框，用来增加或减少图书数，其最小值为0。该文本框的值与inBooks对象的count属性用v-model进行了双向数据绑定，即文本框值的变化会引起count值的变化，从而触发watch()函数，进行相应处理。关于v-model，3.3.4节将会介绍。

插值{{inBooks.message}}用来显示增加了几本书的提示信息。插值{{inBooks.total}}则显示总金额。从这里可以得出，inBooks至少有三个属性：count、message、total。接下来是重头戏，来看<script>代码。

```
<script>
    Vue.createApp({
        setup() {
            const {reactive, computed, watch, onMounted} = Vue
            const books = reactive([            //仓库中存放的图书情况,默认是275本书
                {
                    name: 'Java Web开发教程',
                    price: 48.9,
                    count: 135
                },
                {
                    bname: 'PostgreSQL技术及应用',
                    price: 58.9,
                    count: 140
                }
            ])
            const initBooks = {                 //用来记录仓库原始的图书数、总金额
                count: 0,
                total: 0
            }
            books.forEach(book => {             //计算出仓库原始的图书数、总金额
                initBooks.count += book.count
                initBooks.total += book.price * book.count
            })
            const inBooks = reactive({          //当前仓库情况:图书数量、相关信息、总金额
                count: initBooks.count,
                message: '增加了0本书',
                total: computed(() => {         //利用计算属性计算总金额
                    let sum = 0
                    books.forEach(book => {
                        sum += book.price * book.count
                    })
                    return sum
                })
            });

            watch([books, () => inBooks.count],         ①
              ([newBooks, count], [, oldCount]) => {    ②
                if (count > oldCount && oldCount >= initBooks.count)     //新增图书处理
                    newBooks.push({             //产生一本新书,书名、价格随机生成
                        name: '随机书名' + Math.floor(Math.random() * 100),
                        price: Math.floor(Math.random() * 100),
                        count: 1
                    })
                else if (count < oldCount && count >= initBooks.count) ③
                    newBooks.splice(newBooks.length - 1, 1)       //删除最新入库图书
                if (count <= initBooks.count)                    //当前仓库图书数已达到原始库存数
```

```
                inBooks.count = initBooks.count                    //当前图书数重置为原始库存数
                let num = Math.abs(inBooks.count - initBooks.count)    //计算绝对增加数
                inBooks.message = `增加了${num}本书`
            }
        )
        onMounted(() => {
            //返回 id 为 bookInput 的元素,即调整图书数的文本框
            const bookInput = document.querySelector('#bookInput')
            rxjs.fromEvent(bookInput, 'keydown')                   //添加按键事件监听器
                .subscribe((event) => event.preventDefault())  //禁止键盘输入
        })
        return {inBooks}
    }
}).mount('#app')
</script>
```

① 侦听两个数据 books、count,需要用数组表示。用箭头函数返回 inBooks 对象的 count 属性,对这个属性值进行侦听。

② 这是一个箭头函数,传入的是被侦听的两个数据的新旧状态值。由于两个数据都有新旧两种数据状态,所以需要两个数组。第一个数组[newBooks,count],分别对应所侦听数据的当前值,元素顺序与被侦听数据的顺序一致。第二个数组[,oldCount],对应这两个数据的旧值。同样,第二个数组中的元素顺序也要与被侦听数据的顺序一致。由于程序并不关心 books 的旧值,所以写成"[,oldCount]",省略变量名,表示 books 的旧值被忽视。

③ 减少图书的处理。这时候当前图书数 count 比旧值 oldCount 小,实际上是单击图书文本框向下小箭头的操作,说明需要从 books 数组中删除最新入库的一本书。但是,不允许一直单击向下小箭头,否则图书数可能变成 0,原始图书数也没有了。所以,有一个逻辑与条件"count >= initBooks.count",也就是说,一旦 count 与 oldCount 相等,说明新增的图书已经删完,后续不能再从 books 中删除图书了。

3.3.4　表单域的数据绑定

表单域包含 text(文本框)、textarea(多行文本框)、checkbox(复选框)、radio(单选按钮)、select(下拉选择框)等,用于采集用户输入或选择的数据。

Vue 提供了 v-model 指令,能够实现在这些域元素上的双向数据绑定。下面来看一个包含这几种元素双向绑定的综合性示例。

```
<div id="app">
    姓名:<input v-model="student.name"/>
    简介:<textarea v-model="student.intro"></textarea>
    <p>爱好:<input type="checkbox" v-model="student.fav" value="football"/>足球
        <input type="checkbox" v-model="student.fav" value="dance"/>舞蹈
        <input type="checkbox" v-model="student.fav" value="art"/>艺术
        性别:<input type="radio" v-model="student.sex" value="male"/>男
        <input type="radio" v-model="student.sex" value="female"/>女
        <select v-model="student.major">
            <option value="0701">信息管理</option>
            <option value="0702">电子商务</option>
        </select>
    </p>
    <p>{{student.name}}</p><!-- 这句及下面这句是为了测试双向绑定效果 -->
```

```
    <p style="white-space:pre-wrap">{{student.intro}}</p>
</div>
<script>
    const MyApp = {
        setup() {
            const {reactive, watch} = Vue
            const student = reactive({
                name: '',
                intro: '',
                fav: [],                          //复选框是多个值,所以需要用数组,而非字符串
                sex: '',                          //没有设置默认选中的值
                major: '0701'                     //默认选中"信息管理"
            })
            watch(student, oldStudent => {
                console.log('\u000D')             //换行
                //遍历出选中的全部爱好,在浏览器控制台输出
                student.fav.forEach((s) => console.log(s))
                console.log(newStudent.sex)
                console.log(newStudent.major)
            })
            return {student}
        }
    }
    Vue.createApp(MyApp).mount('#app')
</script>
```

打开页面,如果在姓名、简介文本框中输入内容,将实时同步显示到页面下方。而选中不同爱好、性别、专业后,触发 watch 侦听并在浏览器控制台显示结果。从上面代码中可以总结出一个规律,一般 checkbox、radio、select 的数据绑定常常与 value 属性结合使用。

3.3.5 条件和列表渲染

1. 条件渲染

1) v-if、v-else-if、v-else

v-if 指令用于有条件地渲染内容,只有 v-if 的值为真时才会被渲染,类似于 JavaScript 的 if、else if、else 之类。下面对 3.3.2 节的示例 2 计算图书总金额进行判断输出。

```
<div id="app">
    总金额:<span>{{total}}</span>
    <p v-if="total>=500000">超额</p>
    <p v-else-if="total>=150000">正常</p>
    <p v-else>萎缩</p>
</div>
```

2) v-show

v-show 通过改变元素的 CSS 显示属性来控制 HTML 元素的显示或隐藏,例如:

```
<span v-show="role=='teacher'">欢迎老师加入!</span>
```

当 role 的值为 teacher 时才会显示"欢迎老师加入!"。

2. 列表渲染

v-for 指令对数组中的数据进行循环来渲染列表,常常与 in 或 of 配合使用。来看下面的示例。

```
<div id = "app">
    <ul>
        <li v-for = "college in colleges">                                    ①
            {{college.name}}({{college.no}})
            <ul>
                <li style = 'list-style: none' v-for = "(m,index) of college.major">    ②
                    {{index + 1}}.{{m}}
                </li>
            </ul>
        </li>
    </ul>
</div>
<script>
    Vue.createApp({
        setup() {
            const colleges = Vue.ref([
                {
                    no: '0301',
                    name: '传媒学院',
                    major: ["动画", "广告", "数字媒体", "新闻传播"]
                },
                {
                    no: '0302',
                    name: '经济学院',
                    major: ["国际贸易", "金融学", "财政学"]
                }
            ])
            return {colleges}
        }
    }).mount('#app')
</script>
```

① colleges 是响应式数组对象，代表全部学院，而 college 代表了数组中的某个学院。

② 这里的 m 代表专业名称，而 index 则表示该专业所对应的索引。索引是从 0 开始编号的，这里使用 index+1 以便更符合日常的数字排序习惯。

打开页面后，效果如图 3-3 所示。

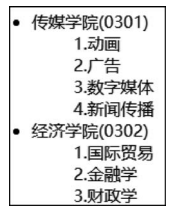

图 3-3　渲染后的列表

3.3.6　对象组件化

组件是一种可重用的独立单元。Vue 可以轻松实现对象的组件化。首先定义一个简单的对象 HelloVue：

```
const HelloVue = {
    props: {
        welcome: String
    },
    template: `<span>{{ welcome }}</span>`
}
```

在前面介绍过 props，用来接收外部传入的数据。这里通过 props 接收字符串数据并传给 welcome 属性。然后，通过 template 模板用插值方式输出到页面上。

组件化对象时，对象的命名请尽量采用驼峰命名法：每个单词的首字母大写，如 HelloVue。

下面构建 Vue 应用。

```
Vue.createApp({
    components: {
        HelloVue
    },
    setup() {
        const hello = Vue.ref('你好,Vue!')
        return {hello}
    }
}).mount('#app')
```

这里最为关键的是用 components 关键字声明了该 Vue 应用要使用的组件是 HelloVue。如果是多个组件,组件之间用逗号隔开。默认情况下,Vue 会将 HelloVue 解析为< hello-vue >元素,即下面的形式。

```
<hello-vue :welcome="hello"></hello-vue>
```

也可以显式指定其他名称的标签名。

```
components: {
        'say-hello':HelloVue
}
```

现在,页面需要这样应用该组件。

```
<div id="app">
    <say-hello :welcome="hello"></say-hello>
</div>
```

由代码可知,HelloVue 中 props 属性 welcome 绑定的是 setup()函数中定义的 hello。就这样,HelloVue 对象被组件化了。

显然,在 HTML 标准里面,并没有 hello-vue(或 say-hello)这样的标签。需要格外注意的是命名方式:将对象名称按单词小写化,单词之间以"-"符号连接。

3.3.7 插槽

很多超市门口放了储物箱(类似于插槽)。超市并不知道顾客会往储物箱里面放什么内容,储物箱里面的内容由顾客决定,且是变化的。计算机主板上有很多内存插槽,插槽上插入什么规格、品牌的内存条,事先无法确定。组装兼容计算机时,有的买家会购买 2GB 金士顿内存条,插入到内存插槽位置,而有的买家则购买 4GB 三星内存条插入到内存插槽。

与上述类似,Vue 的插槽(Slot)允许将不确定的、希望可以动态变化内容的那部分定义为插槽,类似于事先预留的内存插槽。Vue 的插槽实际上是一个内容占位符,可以在这个占位符中填充任何模板代码,例如,一句话、HTML 标签、组件等。当页面渲染显示时,原本 Slot 插槽标记的位置,会显示这些填充的内容。

1. 基本使用

插槽用< slot ></slot >表示,可为其指定默认内容:< slot >插槽默认内容…</slot >。来看下面的完整示例。

```
<div id="app">
    <hello-vue></hello-vue>
```

```
    <hello-vue>你好,Vue!</hello-vue>
    <hello-vue>学而时习之!</hello-vue>
</div>

<script>
    const {createApp, ref} = Vue
    const HelloVue = {
        template: `<div><slot>我是插槽...</slot></div>`
    }
    createApp({
        components: {
            'hello-vue': HelloVue
        }
    }).mount('#app')
</script>
```

代码定义了一个 HelloVue 组件,在组件的模板中使用了插槽<slot>。如果用户未给插槽指定内容,则显示默认内容“我是插槽...”。在 层里面,三次使用了 HelloVue 组件,但插槽内容不一样。在浏览器中打开页面后,会显示如下内容。

```
我是插槽...
你好,Vue!
学而时习之!
```

2. 具名插槽

可以使用带有名称属性的插槽,来划分不同的待显示内容。

```
<slot name="插槽名"></slot>
```

然后,在模板中利用“#插槽名”指定即可。

```
<template #插槽名>插槽内容</template>
```

下面来看一个示例。

```
<div id="app">
    <poem-extract>
        <template #header>望岳</template>
        <template #content>
            会当凌绝顶,一览众山小。
        </template>
    </poem-extract>
    <poem-extract>
        <template #header>忆秦娥·娄山关</template>
        <template #content>
            雄关漫道真如铁,而今迈步从头越。
        </template>
    </poem-extract>
</div>

<script>
    const PoemExtract = {
        template: `<div style="width:300px;text-align:center">
                    <span style="color:#f00;">
                        <slot name="header">标题</slot>
                    </span><br/>
                    <span style="color:#00f">
```

```
                    <slot name="content">诗句</slot>
                </span>
                </div>`
    }
    Vue.createApp({
        components: {
            'poem-extract': PoemExtract
        }
    }).mount('#app')
</script>
```

这里使用了两个具名插槽 header、content，分别用来显示某首诗的标题、诗句。标题字体颜色为红色，而诗句内容的字体颜色为蓝色。在< div >层里面显示的两首诗，将会显示同样的 CSS 效果。在浏览器中打开页面后显示如下。

望岳

会当凌绝顶，一览众山小。

忆秦娥·娄山关

雄关漫道真如铁，而今迈步从头越。

3. 具名作用域插槽

作用域的意思是指数据的应用范围。组件所具有的数据，一般情况下都是在各自组件的内部（作用域）使用。而插槽作为占位符，本身没有数据，其数据来自于父组件提供的内容。也就是说，父组件域内的数据，通过插槽，作用到子组件了。

但是，反过来则不行。某些场景下，父组件除了给插槽内容提供数据外，还需要使用子组件内部的数据。也就是说，插槽内容，一部分来自父组件域内的数据，这本身就是可行的。另外一部分插槽内容，需要来自子组件中的数据，这是不允许的，因为 Vue 组件之间的数据流是单向的，只能从父组件流向子组件。这时候，可以先将子组件的数据，附加给具名作用域插槽，然后就可直接通过解构方式，获得所附带的数据。这样一来，插槽内容，有一部分就来自于子组件了。因此，具名作用域插槽相当于延展了数据的作用域，将子组件数据延展到父组件域内，并在父组件域内使用。来看具名作用域插槽的具体使用方法：

```
<slot name="header" topic ="计算机">标题</slot>
```

这里向具名插槽 header 直接传递了一个对象数据{ topic：'计算机' }。

而 template 模板可以解构并使用这个数据，并在页面形成统一数据格式。下面利用具名作用域插槽修改前面的示例。

```
<div id="app">
    <poem-extract>
        <template #header="{message}">
            【{{message}}】望岳
        </template>
        …
        <template #header="{message}">
            【{{message}}】忆秦娥·娄山关
        </template>
        …
</div>
```

```
<script>
    const PoemExtract = {
        setup() {
            const message = Vue.ref('诗词精选')
            return {message}
        },
        template: `
          <div style="width:300px;text-align:center">
          <span style="color:#f00;">
            <slot name="header" :message=message>标题</slot>
          </span><br/>
          <span style="color:#00f">
            <slot name="content">诗句</slot>
          </span>
          </div>`
    }
    …
</script>
```

给插槽绑定了一个 message 属性，并通过“:message=message”将 message 数据绑定到具名插槽 header。然后，将原来的父模板代码：

```
<template #header>望岳</template>
```

修改成了：

```
<template #header="{message}">
    【{{message}}】望岳
</template>
```

读者可能注意到“#header="{message}"”，这里利用{message}从附加在插槽的数据对象中解构出 message，以供父组件使用。接下来，再利用插值{{message}}显示数据，只不过用“【】”包裹一下，做了简单修饰而已。同样，对<template #header>忆秦娥·娄山关</template>也做了类似修改。再次打开页面，将显示如下效果。

【诗词精选】望岳
会当凌绝顶，一览众山小。
【诗词精选】忆秦娥·娄山关
雄关漫道真如铁，而今迈步从头越。

3.3.8　事件绑定和触发

1. 事件绑定 v-on

v-on 用于绑定事件监听器。例如：

```
<button v-on:click="doLogin"></button>
```

可以用语法糖形式简化：

```
<button @click="doLogin"></button>
```

v-on 支持使用以下修饰符。

- .left：单击鼠标左键时触发。
- .right：单击鼠标右键时触发。

- .self：单击当前元素时触发。
- .once：只触发一次。
- .stop：阻止事件继续传播。
- .prevent：阻止元素的默认事件处理

示例 1：使用 v-on 修饰符。

```
<a href = "http://www.hust.edu.cn" @click.prevent = "goUrl">华科</a>
<form name = "form1" action = "usr/issue" @submit.prevent = "check">
    <input type = "submit" value = "开始发送">
</form>
```

单击链接时，默认会跳转到华中科技大学主页，这里阻止了这个默认行为，而是执行 goUrl()函数。同样，单击表单中的“开始发送”按钮，不会执行默认动作 usr/issue，而是执行 check()函数。

示例 2：先准备两张灯泡图片 bulbon.gif(点亮)、bulboff.gif(熄灭)。单击按钮时，实现灯泡的点亮/熄灭状态切换。

```
<div id = "app">
    <bulb - image :my - img = "current"></bulb - image>
    <button @click = "changeImg">点我切换</button>                    ①
</div>
<script>
    const {createApp, ref} = Vue
    const BulbImage = {
        props: {
            myImg: String
        },
        template:`<img :src = 'myImg' width = '133' height = '160'/>`
    }
    createApp({
        components: {
            'bulb - image':BulbImage
        },
        setup() {
            const current = ref('image/bulbon.gif')
            const changeImg = () =>                                  ②
                        current.value = current.value.match('bulbon') ?
                            'image/bulboff.gif' : 'image/bulbon.gif'
            return {current, changeImg}
        }
    }).mount('#app')
</script>
```

① 绑定按钮单击事件，调用 changeImg()函数。

② 使用三元条件运算，判断 current 的内容是否匹配 bulbon，以此切换 current 的图片值。

2. 事件触发 $ emit

也可以使用 $ emit 触发当前对象上的各种标准或自定义事件，基本使用格式：

```
$ emit(事件名,参数)
```

例如 $ emit('toggle')，其中，toggle 是自定义事件名，然后可与具体的某个函数进行绑定。改写上面的灯泡切换示例，去掉“点我切换”按钮，直接单击灯泡图片时实现灯泡点亮/熄灭状态切换。

```
<div id="app">
    <bulb-image width='133' height='160' alt="灯泡切换" style="cursor:pointer"
                :image="current" @switch-bulb="changeImg">
    </bulb-image>
    …
    props: {
        image: String
    },
    emits: ['switchBulb'],                                    //定义事件名数组
    template: `<img :src='image' @click="$emit('switchBulb')" v-bind="$attrs"/>`
    …
</script>
```

代码@switch-bulb="changeImg"将 switchBulb 事件与 changeImg()函数绑定，@click="$emit('switchBulb')"则将图片的 click 事件与事件 switchBulb 绑定。单击图片时，触发 switchBulb 事件，自然就会调用 changeImg()函数实现图片的切换处理。有人可能疑惑<bulb-image>标签中为什么是“@switch-bulb”而不是“@switchBulb”？这正是需要注意的地方：需要将 switchBulb 变形为贴近 HTML 书写风格的 switch-bulb 形式。当然，完全可以在定义事件名数组时直接写成：emits:['switch-bulb']。

细心的读者会注意到 template 模板中的 v-bind="$attrs"，这叫“属性透传”。BulbImage 组件 template 模板中的<img>标签，只是绑定了 src、click 等，而在 层中显然设置了<img>标签的 width、height、alt、style 等属性。这些属性是怎么传递到<img>标签的？Vue 允许通过属性传递：v-bind="$attrs"，直接将父组件提供的数据传递到子组件，非常方便！

请仔细与前面事件绑定 v-on 中的示例 2 进行比较，体会其差异性。

3.3.9　自定义元素

defineCustomElement 用于自定义元素，可实现灵活、高度复用的功能性封装。下面是其基本使用格式，里面的元素读者应该并不会感到特别陌生。

```
const MyElement = Vue.defineCustomElement({
  props: {},                                                  //参数
  setup() {},                                                 //组合函数
  template: `...`,                                            //模板
  styles: [`/*CSS样式*/`]                                     //样式
  …
})
```

可以将 MyElement 作为单独部分存在，然后在需要使用的地方注册即可。

```
customElements.define('my-element', MyElement)
```

现在，就可以在页面中使用了：

```
<my-element></my-element>
```

用 defineCustomElement 重新实现灯泡切换处理：

```
<div>
    <bulb-img></bulb-img>
</div>
<script>
    const {defineCustomElement, ref} = Vue
```

```
    const BulbImage = defineCustomElement({
        setup() {
            const current = ref('image/bulbon.gif')
            const changeImg = () => current.value = current.value.match('bulbon') ?
                'image/bulboff.gif' : 'image/bulbon.gif'
            return {current, changeImg}
        },
        template: `<img :src = 'current' @click = "changeImg" class = 'bulb'/>`,
        styles: [`
                    .bulb {
                        width: 133px;
                        height: 160px;
                        cursor: pointer;
                        box-shadow: 1px 1px 2px #ccc;
                    }
                `]
    })
    customElements.define('bulb-img', BulbImage)
</script>
```

自定义元素 BulbImage 将灯泡图片的切换功能、样式、事件处理等，全部封装在一起了，实现了高度的复用性。代码中的 styles 只适用于 defineCustomElement，用于将 CSS 代码直接封装在自定义元素里面。styles 是一个数组，可以定义若干样式，具体写法与 CSS 完全一样。

当然，这里仍然可以对<img>使用属性透传，请读者自行修改实验。

3.3.10 自定义指令和插件

1. 自定义指令

Vue 内置了 v-html、v-if、v-model、v-show 等一系列指令，也允许自定义指令满足应用需要。一般有两种方式自定义指令，一种是通过 directive 注册为全局指令，另外一种则是通过 directives 注册为当前组件可用的局部指令方式。下面结合示例来说明。

（1）通过 directive 注册为全局指令。

```
<div id = "app">
    <button v-my-button:style.color = "'#00f'">查询</button>          ①
    <button v-my-button:color = "'#f00'">打印</button>                ②
    <button>退出</button>
</div>
<script>
    Vue.createApp()
        .directive('myButton', {
            mounted: (el, binding) => {
                el.style.cursor = 'pointer'                            //鼠标形状
                el.style.color = binding.value                         //字体颜色
                el.style.width = '100px'                               //长度
            }
        }).mount('#app')
</script>
```

这里自定义指令名称为“myButton”，用来定制某类按钮的外观：手形鼠标，字体颜色可定制，长度都是 100px。该指令被注册为全局指令，因为是挂载在 Vue 应用上面。

应用自定义指令时，其名称一般以“v-”开头，全部字母小写，以单词为单位展开并以“-”连接，

例如 v-my-button。当然定义指令时并不需要以“v-”为前缀。

自定义指令一般包含钩子函数以便确定其调用时机，例如，代码中的 mounted 就是钩子函数，表示组件挂载完成后调用自定义指令。可以利用 mounted 的参数进行额外处理，主要有以下可选参数。

- el：自定义指令绑定的元素，例如，这里绑定的元素是< button >。
- binding：一个对象，主要包含 value(传递给指令的值)、oldValue(之前的值)、arg(传递给指令的参数名)、instance(使用指令的组件实例)等属性。
- vnode：指令所绑定元素的底层虚拟结点(VNode)。

其他钩子函数还有 created()、beforeMount()、beforeUpdate()、updated()、beforeUnmount()、unmounted()等，这里不再赘述，请参阅 Vue 官方文档。

① 传递给自定义指令的 value 值'#00f'，arg 参数名 style，修饰符 color。使用这种形式给指令传递数据，可读性比第②种的更强。

② 传递给自定义指令的 value 值'#f00'，arg 参数名 color，修饰符为空。

从页面运行结果看，“查询”“打印”按钮的鼠标手形、长度一致，“查询”按钮的字体颜色为蓝色，“打印”按钮的字体颜色为红色。而“退出”按钮的外观完全不一样，是默认的外观形式。

传递数据还可以使用对象形式，一次传递多个数据。

```
<button v-my-button={color:'#00f',width:'100px'}>查询</button>
<button v-my-button={color:'#f00',width:'70px'}>打印</button>
```

在 mounted 里面，这样赋值：

```
el.style.color = binding.value.color
el.style.width = binding.value.width
```

(2) 通过 directives 注册为局部指令。

```
<div id="app">
    <button v-my-button>查询</button>
    <button v-my-button>打印</button>
    <button v-my-button>退出</button>
</div>
<script>
    Vue.createApp({
        directives: {
            myButton: {
                mounted: (el) => {
                    el.style.cursor = 'pointer'
                    el.style.color = '#00f'
                    el.style.width = '100px'
                }
            }
        }
    }).mount('#app')
</script>
```

这里的 myButton 指令，只能在 app 作用域内局部使用。跟前面的示例稍有不同，这里三个按钮的外观完全一样了。

2. 插件

插件(Plugins)的主要目的是为应用添加能够在任意模板中使用的全局性功能，例如，全局资

源、全局对象、全局方法、全局指令等，而自定义指令有时候是作为局部指令而存在的。

下面将前面注册为局部指令的 myButton，用插件方式注册为全局指令。

```
<div id="app">
    <button v-my-button>查询</button>
    <button v-my-button>打印</button>
    <button v-my-button>退出</button>
</div>
<script>
    const appPlugin = {
        install(app, options) {
            app.directive('myButton', {
                mounted: (el) => {
                    el.style.cursor = options.cursor
                    el.style.color = options.color
                    el.style.width = options.width
                }
            })
        }
    }
    Vue.createApp()
        .use(appPlugin, {                                    //安装插件并传递数据
            cursor: 'pointer',
            color: '#00f',
            width: '100px'
        }).mount('#app')
</script>
```

插件通常需要利用 install()方法，将其安装到应用实例中去。install()方法有两个参数：目标应用、传递的数据。上面的代码将自定义指令 myButton 安装到 app 应用中，并传递 CSS 属性数据 cursor、color 和 width。现在，myButton 作为全局自定义指令，可在整个 Vue 应用中使用。当然，像下面这样直接使用 install()安装方法也是可以的。

```
...
<script>
    const install = (app, options) => {
            app.directive('myButton', {
            ...
    }
    const app = Vue.createApp()
    install(app, {
        cursor: 'pointer',
        color: '#00f',
        width: '100px'
    })
    app.mount('#app')
</script>
```

3.4 渲染函数

3.4.1 h()函数

h()函数中的“h”是 hyperscript 的简称，即能够生成 HTML 元素的 JavaScript。h()函数返回

一个虚拟结点(Virtual Node,VNode)。虚拟结点,实际上是一个描述 HTML 元素(例如 div、table 等)的 JavaScript 对象。虽然不是真实的 HTML 元素,但又具备 HTML 元素的所有特征。VNode 保存在内存中,通过设计数据结构形式,“虚拟”地表示出 HTML 元素,所以被称为 VNode。虚拟结点能够让我们在内存中,灵活、动态地组合出希望的界面视图效果。

h()函数基本格式如下。

```
h(元素类型,属性,子元素)
```

这个格式是嵌套的,也就是说,子元素一样也可以是 h()函数。如果是多个子元素,则需要用数组,类似于:

```
h(元素类型,属性,[子元素 1, 子元素 2,…])
```

使用 h()函数,可以非常简单:Vue.h('span','你好,Vue!');也可以使用各种复杂的组合。来看下面的示例。

示例 1:创建并显示文字内容为“强大的 h()函数!”的 div 层。

```
<div id = "app"></div>
<script>
    Vue.createApp({
        setup() {
            const {h} = Vue
            //利用 return 直接返回渲染结果
            return () => h('div', {                    //创建 div 层
                innerText: '强大的 h()函数!',
                style: {
                    width: '160px',
                    height: '28px',
                    cursor: 'pointer',                 //鼠标为手形
                    textAlign: 'center',               //文字居中
                    border: '1px dotted #00f',         //边框 1px、点线、红色
                    boxShadow: '2px 2px 3px #ccc' //灰色边框阴影
                }
            })
        }
    }).mount('#app')
</script>
```

代码创建了一个 div 层,利用 style 属性,对该层的 CSS 样式进行了定义,并设置其内部文字为“强大的 h()函数!”。

示例 2:显示图书馆的馆藏图书及索取号,如图 3-4 所示。

```
<div id = "app"></div>
<script>
    Vue.createApp({
        setup() {
            const {h} = Vue
            return () => h('figure', {                 //创建 HTML 的 figure 图片标签元素
                    style: {
                        textAlign: 'center',
                        width: '180px',
                        height: '270px',
                        border: '1px solid #f00',
                        boxShadow: '2px 2px 3px #ccc'
```

```
                    }
                },
                [h('img', {                          //figure 的第 1 个子元素:图片< img >
                    src: 'image/springboot.png',     //图片源
                    decoding: 'async',               //异步解析图像
                    style: {
                        display: 'block',
                        objectFit: 'contain',        //调整图片以适应父容器的长宽比
                        maxWidth: '180px',
                        maxHeight: '240px',
                    }
                }),
                    h('figcaption', {                //figure 的第 2 个子元素:标题< figcaption >
                        style: {textAlign: 'center'}
                    }, '馆藏索取号:TP02 - 101')      //figcaption 的内容
                ])
        }
    }).mount('#app')
</script>
```

图 3-4　h()函数渲染结果

3.4.2　render()函数

渲染函数 render()和 h()函数类似,用法上稍有差异。render()和 h()结合,允许充分利用 JavaScript 的编程能力,灵活实现设计人机交互界面的目的。值得注意的是,render()函数的优先级高于 template 模板,这意味着如果二者同时存在,会优先显示 render()渲染的内容,而非 template 模板的内容。

示例 1:改写 3.4.1 节示例 2,用 render 渲染显示。

```
<div id = "render"></div>
<script>
    Vue.createApp({
        render() {
            const {h} = Vue
```

```
            return h('figure', {
                style: {
                …
            )
        }
    }).mount('#render')
</script>
```

请仔细阅读代码，你能比较出差异么？

示例 2：继续改写上面的示例，通过向 render 传递参数来渲染页面。

```
<div id = "render"></div>
<script>
    Vue.createApp({
        setup() {
            const {render, h} = Vue
            const content = h('figure', {
                …
            )
            //获取页面 id = "render"的<div>层
            const container = document.querySelector('#render')
            render(content, container)              //将虚拟结点渲染到层
        }
    }).mount('#render')
</script>
```

这种情况下，render()语法格式为 render(VNode 虚拟结点，DOM 元素对象)。代码用 h()函数创建了虚拟结点 content，用 document.querySelector 获取到页面中的 层。上面代码的处理思路非常重要，需要熟练掌握。

示例 3：利用 render()、h()、defineCustomElement 组合实现如图 3-5 所示的按钮。单击“消息”按钮时，“消息”二字变成“渲染示例”。

图 3-5　消息按钮

这个效果，如果用 HTML 元素来写，主要代码结构如下。

```
<button>
    <img src = "image/info.png" width = "22" height = "22">消息
</button>
```

注意，这里省略掉了 CSS 修饰代码，以及按钮单击事件的 JavaScript 代码。现在，用一个自定义标签<message-tip>，封装上面的按钮。来看看如何用 h()、defineCustomElement 组合实现。

```
<message-tip></message-tip>  <!-- 自定义元素标签 -->
<script>
    const MessageTip = Vue.defineCustomElement({     //自定义元素
        render() {
            const {h} = Vue
            const button = h('button', {             //最外层的按钮<button>
                    style: {                         //样式修饰
                        border: '0px',
                        width: '70px',
                        height: '45px',
                        borderRadius: '4px',         //边框为圆角
                        background: '#d8d4d4',
                        cursor: 'pointer'
                    }
                }, [
```

```
                h('img', {                          //向 button 里面添加图片
                    src: 'image/info.png',
                    width: 22,
                    height: 22
                }),
                h('br'),                            //在图片后面添加换行
                h('span', {                         //向 button 里面添加 span 元素
                    //单击时,修改 span 内部文字
                    onclick: (event) => event.target.innerText = '渲染示例',
                    innerText: '消息'               //span 默认的文本内容
                }),
                h('div', {                          //向 button 里面添加层,用于显示右上角的数字 6
                    innerText: 6,
                    style: {                        //修饰数字 6:圆形、红底白字、显示于右上角
                        position: 'relative',
                        left: '94%',
                        top: '-50px',
                        borderRadius: '50%',
                        textAlign: 'center',
                        color: '#fff',
                        background: '#f15555',
                        width: '12px',
                        height: '12px',
                        fontSize: '10px',
                    }
                })]
            )
            return h(button)
        }
    })
    customElements.define('message-tip', MessageTip)
</script>
```

上面的处理思路非常简单：定义好 defineCustomElement 元素 MessageTip，该元素由 render()、h()渲染而成，然后用 customElements 注册为 message-tip。再在页面上直接使用< message-tip ></message-tip >标签。甚至都不需要创建 Vue 应用，最大的优势在于体现了良好的封装性，能够带来极高的可复用性。

综上来看，h()、render()函数提供了完全的生成并控制 HTML 页面元素的能力。在某些场景下，可以代替 template，二者配合渲染、构建页面，请读者熟练掌握。

3.5 使用组件

组件能够实现代码的复用，也方便代码的管理。

3.5.1 组件定义及动态化

1. defineComponent 定义组件

定义组件可使用 defineComponent，其参数比较灵活，一般常使用具有组件选项的对象作为其参数。来看下面的示例。

```
<div id="app"></div>
<script>
    const CollegeComponent = Vue.defineComponent({
        setup() {
            const colleges = [
                {id: 'mgc', name: '管理学院'},
                {id: 'mdc', name: '传媒学院'}
            ]
            return {colleges}
        },
        template: `<ul>
                    <li v-for="college of colleges">
                      {{ college.name }}({{ college.id }})
                    </li>
                  </ul>`
    })
    Vue.createApp(CollegeComponent).mount('#app')
</script>
```

在浏览器中打开页面后，将显示如下形式的内容。

- 管理学院(mgc)
- 传媒学院(mdc)

由此可见，前面学过的很多知识点，都可应用在 defineComponent 中。

2. 动态组件

Vue 提供了组件的动态绑定，帮助我们灵活地切换组件。基本格式如下。

```
<component :is="module"></component>
```

利用 is 属性切换不同的组件，实现动态改变效果。只要改变 module 的值，页面就会显示不同组件的内容。下面就利用 is 的这个特性，来实现组件的切换。只要单击“切换组件”按钮，页面内容将在两个组件之间切换。具体代码如下。

```
<div id="app">
    <component :is="curModule"></component>
    <button id="switchBtn">切换组件</button>
</div>

<script>
    const {
        defineComponent, createApp, ref, onMounted, render, h
    } = Vue
    const MgcComponent = defineComponent({            //定义第 1 个组件:管理学院
        template: `<div>管理学院拥有管理科学与工程...</div>`
    })
    const MdcComponent = defineComponent({            //定义第 2 个组件:传媒学院
        template: `<div>传媒学院以新媒体、跨媒体...</div>`
    })
    createApp({
        setup() {
            const curModule = ref('mgc')              //is 默认绑定名为 mgc 的组件,即管理学院
            onMounted(() => {
                document.querySelector("#switchBtn")
                    .addEventListener('click', () => curModule.value === 'mgc'?
                        curModule.value = 'mdc' : curModule.value = 'mgc')     ①
```

```
            })
            return {curModule}
        }
    }).component('mgc', MgcComponent)                                    ②
      .component('mdc', MdcComponent)
      .mount('#app')
</script>
```

① 给 switchBtn 按钮添加 click 事件。单击按钮时，判断< component >的 is 当前所绑定的组件名。若是 mgc(管理学院)，则赋值为 mdc(传媒学院)，否则赋值为 mgc。由于 curModule 为响应性对象，其值的变化，将触发< component >内容改变到传媒学院。

② 将组件 MgcComponent 注册到当前应用实例，注册名为 mgc。component()函数用于获取或注册组件，基本格式如下。

```
component(name:String,component:Component)
```

其中，第 2 个参数可选。如果没有提供第 2 个参数，例如 app. component('mgc')，则查找并返回该名字注册的组件。

这个示例基本展现了一些 Web 应用系统，如何实现在单击菜单时，通过切换组件改变页面内容的基本方法。后续章节所实现的系统导航菜单切换，就采用了这种动态组件。

3.5.2 异步组件

异步组件 defineAsyncComponent 在运行时是“懒加载”，也就是说，只在需要的时候才会去加载内容。一般使用方法：

```
const AsyncComponent = Vue.defineAsyncComponent(
  () => new Promise((resolve, reject) => {
      resolve({
        template: '< span >我是异步组件!</span >'
      })
    })
)
app.component('async-component', AsyncComponent)
```

defineAsyncComponent 返回一个 Promise 对象并利用 resolve 回调内容。关于 Promise，请参阅 JavaScript 相关资料。但这是最基本的使用，下面通过一个详细示例来学习更为具体的用法。

示例：在页面上用 10s 时间显示会计学院文字介绍，随后自动播放学院视频。

```
< div id = "app">
    < acc-college ></acc-college >
</div >

< script >
    const {ref, h, defineAsyncComponent, createApp} = Vue
    const accDescribe = { //会计学院文字介绍组件
        template: `< div >
                    < span >会计学院是中国会计学会理事单位...</span >< br >
                    < span style = 'color:#0000ff'>稍候，请欣赏学院视频...</span >
                </div >`
    }
    const AccCollege = defineAsyncComponent({                    //会计学院处理视频的异步组件
        loader: () => new Promise(resolve => {
            setTimeout(() => {                                   //延迟 10s 后显示视频画面
```

```
                resolve({                                          ①
                 template: `
                   <video width="450" height="350" preload="auto" controls autoplay>
                      <source src="video/acc.mp4" type="video/mp4">
                      您的浏览器不支持视频标签
                   </video>`
                })
            }, 10000)
        }),
        delay: 0,                                    //延迟 0ms,即:立即加载显示 acDescribe 组件内容
        loadingComponent: accDescribe,               //加载会计学院文字介绍组件
        errorComponent: h('div', {                   ②
            style: {color: '#f00'},
        }, '视频加载失败...'),
        onError: (error, retry, fail, attempts) => {  //一旦出错,试着重试加载
            console.log(error.message)               //在浏览器控制台显示出错信息
            console.log(`第${attempts}次重试!`)       //控制台显示重试信息
            attempts < 3 ? retry() : fail()          //重试 3 次
        }
    })
    createApp(AccCollege).mount('#app')
</script>
```

① 回调并解析模板。模板内容为会计学院视频。<video>标签设置了视频自动播放,但是有些浏览器默认不支持视频的自动播放,导致自动播放失效,需要在浏览器里面手工设置允许自动播放视频。

② 若加载出错,则显示此组件的内容。这里直接用 h()函数渲染显示一个 div 层,用红色文字显示“视频加载失败...”字样。

为了观察出错时的处理,将上述代码简单修改一下,来模拟出错效果。将 loader 修改成下面的代码。

```
loader: () => new Promise(resolve => {
    throw new Error('404:无法加载视频资源!')          //抛出错误
})
```

404 是设置的模拟出错状态码,表示无法找到相应资源。在浏览器中打开页面,页面会显示红色文字“视频加载失败...”,而浏览器控制台则会显示如图 3-6 所示信息。

图 3-6　控制台出错提示

3.5.3　数据提供和注入

我们常常用 props 在组件之间传递数据。Vue 还提供了另外一个更直截了当的方法：provide

和 inject。

1. 数据提供 provide

provide 用于向后代组件注入数据。注意,这里强调的是后代组件。例如,A 组件提供了数据:

```
provide('caption', '教务辅助管理系统')
```

或者函数形式:

```
provide() {
    return {
        'caption': '教务辅助管理系统'
    }
}
```

子组件 B 可通过 caption 注入并使用“教务辅助管理系统”数据。B 的子组件 C 也可注入使用该数据。

2. 数据注入 inject

注入由祖先组件提供的值,常常与 provide 配合使用。下面注入前面所提供的 caption。

- const caption = inject('caption') //注入 caption
- const caption = inject('caption', '无标题') //注入 caption,若为空则默认值为“无标题”
- const caption =inject('caption', () => '无标题', true)

三种写法各有特点。最后一种写法,在注入 caption 时,若 caption 为空,则默认值通过函数的返回值来确定,例如,这里的箭头函数返回值是“无标题”。第三个参数 true,指示默认值由函数来提供。下面是一个 provide、inject 配合使用的示例。

示例:提供公钥数据给子组件使用。

```
<div id = "app">
    <print - public - key :unit = "'管理学院'"></print - public - key>
</div>

<script>
    const {createApp, provide, inject} = Vue
    function printPublicKey(props) {                          //声明数据传递参数 props
        const pkey = inject('publicKey')                      //注入数据
        return `单位: ${props.unit}\u3000 公钥: ${pkey}`      //\u3000 表示全角空格
    }
    createApp({
        components: {
            'print - public - key': printPublicKey
        },
        setup() {
            provide('publicKey', '0701@2003 $ - Edu!')        //提供数据
        }
    }).mount('#app')
</script>
```

也可以不用 setup(),而使用函数形式:

```
createApp({
    components: {
        'print - public - key': printPublicKey
    },
```

```
    provide() {
        return {
            'publicKey': '@07012003$ - Edu!'
        }
    }
}).mount('#app')
```

读者可能注意到 printPublicKey()函数的用法。这是一个函数式组件。函数式组件使得开发人员无须像前面 defineComponent 那样定义组件,在一些场景下会带来方便。这个示例用了数据提供和注入,也用了参数传递 props。打开页面后,浏览器输出内容如下。

```
单位:管理学院 公钥:0701@2003$ - Edu!
```

3.6 单文件组件

单文件组件(Single File Component,SFC)常用来表示扩展名为".vue"的文件,是 Vue 提供的特殊文件格式。这种文件格式,将 Vue 组件需要的模板、JavaScript 逻辑处理以及 CSS 样式封装在单个文件中。SFC 使得开发人员可以更清晰地组织组件代码,提高代码的复用性、可维护性。

3.6.1 基本结构形式

单文件组件的基本结构形式如下。

```
<template>
    在这里放置 HTML 模板内容
</template>
<script>
    //在这里编写 JavaScript 脚本代码
</script>
<style>
    /* 在这里编写 CSS 样式内容 */
</style>
```

这三部分并不是必须都写全,而是择需而定。在项目里面右击,选择 New→Vue Component,即可创建单文件组件,如图 3-7 所示。

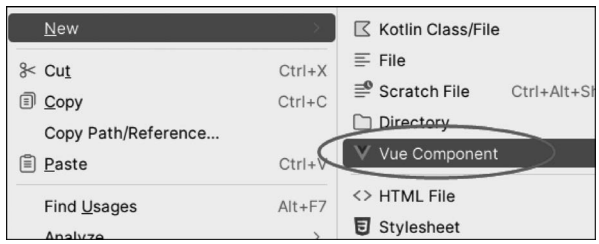

图 3-7　创建单文件组件

示例:管理学院情况介绍的单文件组件 mgc.vue。

```
<template>
    <div class = "college - mgc">
        <header>
```

```
            <img src="image/mgc.png" alt="mgc"/>{{ caption }}
        </header>
        <main>管理学院以开放视角...打开管理大门。</main>
        <footer>官网:www.mgc.edu.cn</footer>
    </div>
</template>
<script>
export default {
    name: "college-mgc",
    setup() {
        const caption = Vue.ref('学院名片')
        return {caption}
    }
}
</script>
<style scoped>
img {
    vertical-align: middle;
}
header {
    font-size: large;
    color: #00f;
}
main {
    font-size: 16px;
}
footer {
    text-align: center;
    font-size: 12px;
}
.college-mgc {
    position: relative;
    width: 300px;
    height: 80px;
    margin: 0 auto;                                    /*自动居中*/
    border: 1px solid #00f;
    box-shadow: 2px 2px 2px #ccc;
}
</style>
```

<style scoped>封装了当前组件的样式,其中,scoped 表示局部样式,仅对当前组件有效。如果去掉了 scoped,则意味着是全局样式。可以同时定义局部样式、全局样式:

```
<style>
img {
    vertical-align:middle;
}
</style>
<style scoped>
.college-mgc {
    position: relative;
    ...
}
```

也可以对 mdc.vue 文件的内容做分散处理,将三部分内容分别保存为相应文件,然后用 src 属性导入进来,例如:

```
<template src="./html/mgc.html"></template> <!-- 前目录子目录 html 下的 mgc.html -->
<style src="./css/mgc.css"></style>     <!-- 前目录子目录 css 下的 mgc.css -->
<script src="./js/mgc.js"></script>     <!-- 前目录子目录 js 下的 mgc.js -->
```

提示：若右键菜单里面没有出现 Vue Component 菜单项，可先创建一个普通文件，手工修改扩展名为“.vue”。再次在项目名称上右击选择菜单 New 时，一般就会出现 Vue Component 菜单项。

3.6.2　样式选择器

在前面使用<style scoped>封装了当前组件使用的样式，但是组件样式是有作用域的，默认情况下子组件并不能从作用域样式中接收样式。不过，通过样式选择器，组件样式能够以不同方式影响到子组件。

1. 深度选择器：deep()

使用：deep()伪类，可将父组件的样式渗透到子组件中，例如：

```
<style scoped>
:deep(.header-title) {
  font-size: 18px;
}
</style>
```

通过使用伪类：deep()，使得特定的 CSS 自定义样式类.header-title 不再限定于父组件，而是可作用于其下的任何子组件。

2. 插槽选择器：slotted()

同样，默认情况下作用域样式也无法影响到插槽<slot>渲染出来的内容。使用：slotted()伪类可以将插槽内容作为 CSS 样式的作用目标。

```
<template>
  <div class="my-title">
    <slot>标题</slot>
  </div>
</template>

<style scoped>
:slotted(h3) {
  font-size: 16px;
  color: #00F;
}
</style>
```

这里通过利用伪类：slotted()，使得插槽<slot>中的<h3>标签具有“文字颜色为红色、16px 大小”的样式。

3. 全局选择器：global()

如果希望将某个样式规则应用到全局，可使用：global()伪类来实现。

```
<style scoped>
:global(h3) {
  font-size: 16px;
  color: #00F;
```

```
}
</style>
```

这类似于使用：

```
<style>
h3{
  font-size: 16px;
  color: #00F;
}
</style>
```

从第 6 章开始，将会结合项目模块实战，展示样式选择器的具体使用。

3.6.3 使用 vue3-sfc-loader 导入 SFC

到目前为止，还无法查看 SFC 的运行效果，因为浏览器并不能识别.vue 文件。需要使用专门的编译打包工具，如 webpack、rollup 等，将其编译成浏览器可识别的代码。这些编译打包工具，往往需要做各种构建配置，还需要 Node.js 支持。

第三方插件 vue3-sfc-loader.js 专门用于在运行时动态加载.vue 文件，能够加载、解析、编译.vue 文件中的模板、JavaScript 脚本和 CSS 样式，并分析依赖项进行递归解析。vue3-sfc-loader.js 不需要安装配置 Node.js，也不需要做任何构建配置。vue3-sfc-loader.js 包含 Vue3(Vue2)编译器、Babel JavaScript 编译器、CSS 转换器 postcss、JavaScript 标准库补丁库 core-js，且内置了对 ES6 的支持。vue3-sfc-loader.js 简单易用，几乎不需要做任何配置。更详细的内容，可通过这个网址 https://www.jsdelivr.com/package/npm/vue3-sfc-loader 进行了解。

有两种方式使用 vue3-sfc-loader.js。一种自然是简单快捷、需要网络支持的 CDN 方式，直接在 HTML 文件的<head>标签中加入：

```
<script src="https://cdn.jsdelivr.net/npm/vue3-sfc-loader/dist/vue3-sfc-loader.js">
</script>
```

另外一种方式仍然是下载 JS 文件到项目里面，随时使用。在浏览器中打开下面两个地址中的某一个均可快速下载。

```
https://cdn.jsdelivr.net/npm/vue3-sfc-loader/dist/vue3-sfc-loader.js
https://unpkg.com/vue3-sfc-loader@0.9.5/dist/vue3-sfc-loader.js
```

同样，在源码界面上右击，另存到项目的 js 文件夹下，然后在页面引入即可，本书采用这种方式。

vue3-sfc-loader.js 提供了一个非常重要的函数 loadModule()，可用来加载.vue 组件。

示例：编写页面主文件 s3-6-3.html，利用 vue3-sfc-loader 编译加载运行 3.6.1 节创建的 mgc.vue 组件。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>sfc 示例</title>
    <script src="js/vue.global.prod.min.js"></script>
    <script src="js/vue3-sfc-loader.js"></script><!--引入 vue3-sfc-loader.js -->
</head>
```

```
<body>
<div id="app">
    <college-mgc></college-mgc>
</div>

<script>
    const options = {
        moduleCache: {vue: Vue},                          //模块缓存器
        getFile: (url) => fetch(url).then(response => response.ok ?
            response.text() : Promise.reject(response)), //文件内容加载器
        addStyle: (styleStr) => {                         //加载 CSS 文件到文档<head>里面
            const style = document.createElement('style')
            style.textContent = styleStr
            const ref = document.head
                .getElementsByTagName('style')[0] || null
            document.head.insertBefore(style, ref)
        }
    }
    //从 window 全局对象 vue3-sfc-loader 中解构 loadModule
    const {loadModule} = window["vue3-sfc-loader"]
    const {createApp, defineAsyncComponent} = Vue
    createApp({
        components: {                                     //异步加载 mgc.vue,并注册为 college-mgc
            'college-mgc': defineAsyncComponent(() => loadModule('./mgc.vue', options))
        }
    }).mount('#app');
</script>
</body></html>
```

为了避免读者出现差错,这里给出了完整代码。在浏览器中打开 s3-6-3.html,效果如图 3-8 所示。

图 3-8　运行效果

3.7 组合式语法糖

3.7.1 基本语法

在前面很多地方写了类似于下面的组合函数语句:

```
<script>
    Vue.createApp({
        setup() {
            …
            return {caption}
        }
        …
    }).mount('#app')
</script>
```

使用单文件组件后，为了简洁代码、提高运行性能，Vue 提供了< script setup >组合式 API 的语法糖。< script setup >里面的代码，会被编译成 setup()函数的内容，并在创建组件实例时执行。而在< script setup >中声明的变量、对象、函数、import 导入的内容等，都能直接在 template 模板中使用，无须使用类似于 return {caption}这样的语句。接下来，将 3.6.1 节 mgc.vue 组件修改成< script setup >形式：

```
< template >…</template >
< script setup >
const caption = Vue.ref('学院名片')
</script >
< style scoped >…</style >
```

当然，只需要修改< script >部分，现在脚本代码只需要一句：

```
const caption = Vue.ref('学院名片')
```

这里的 caption，可以直接在 template 模板中使用。使用< script setup >后的代码，是多么简洁！

3.7.2 属性声明和事件声明

1. 属性声明 defineProps

在 setup()组合式函数里面，经常利用 props 传递数据，请参阅 3.2.3 节的内容。那么在< script setup >里面，可以利用 defineProps()声明数据传递属性。仍以 3.6 节的 mgc.vue 为例，对其稍做修改，通过参数值来控制“学院名片”左边图像的显示或隐藏。

```
< template >
    < div class = "college - mgc">
        < header >
            < img src = "image/mgc.png" v - show = "visible" alt = "mgc"/>
            {{ caption }}
        </header >
        …
    </div >
</template >
< script setup >
const {ref, defineProps} = Vue
const caption = ref('学院名片')
defineProps({
    visible: {type: Boolean, default: true}
})
</script >
< style scoped >…</style >
```

代码中，v-show = " visible" 说明< img >图像是显示还是隐藏，由 visible 的值决定。而 defineProps 定义了 visible 为布尔型，默认值是 true。现在，需要在页面主文件 s3-6-3.html 中，给 mgc.vue 组件的 visible 传递某个值。只需要修改 s3-6-3.html 中的如下代码。

```
< div id = "app">
    < college - mgc :visible = "false"></college - mgc >
</div >
```

其中，:visible = "false"为新增代码。在浏览器中打开 s3-6-3.html，“学院名片”左边图像消失不见。再将 false 改成 true，图像又会出现。这里当然只是简单的模拟操作，以便学习 defineProps

的基本用法，后续章节中会有其实战应用。

2. 事件声明 defineEmits

defineEmits()用于声明自定义事件，并在需要的时候触发事件处理。其用法比较简单：先在父组件中给某个子组件(例如< enroll-combobox >)绑定一个事件处理函数(例如 setVisible)：

```
< enroll - combobox @setVisible = "setVisible"></enroll - combobox >
```

@setVisible 表示绑定的事件名叫“setVisible”，而等号后面则表示该事件绑定的函数叫 setVisible。下面是 setVisible()函数的示例性代码。

```
const visible = ref(null)
const setVisible = (v) => visible.value = v
```

然后，在< enroll-combobox >标签所对应 enroll-combobox.vue 组件的< script setup >中：

```
const {defineEmits} = Vue                                    //解构出 defineEmits
//声明事件数组，目前只有一个事件 setVisible。多个事件之间用逗号隔开。
const emits = defineEmits(['setVisible'])
```

最后，在 enroll-combobox.vue 组件中某个需要的地方激发事件、传递数据。

```
emits('setVisible', true)
```

激发父组件的 setVisible 事件，传递数据 true。该事件会调用 setVisible()函数，将 visible 赋值为 true。更详细的应用方法，请参阅后续章节。

3.7.3　属性暴露

3.7.1 节说过，在< script setup >中声明的对象、函数、组件等，被暴露出去，可直接在 template 模板中使用。有时候，可能希望显式地指定要暴露出去的属性或函数，就需要利用 defineExpose()了。例如，修改 mgc.vue，指定暴露 caption：

```
< script setup >
const {ref, defineProps, defineExpose} = Vue
const caption = ref('学院名片')
defineProps({
    visible: {type: Boolean, default: true}
})
defineExpose({
    caption
})
</script >
```

当然，因为只有一个 caption 需要暴露，这里其实并不需要设置。如果是多个，则用逗号隔开即可。

3.8　使用聚合器封装内容

可以将某个或某类组件调用的函数、方法或常量等抽取出来，封装在一个单独的聚合器文件里面(例如 print.aggregator.vue)，并定义哪些可以被暴露出去。当需要使用时，从该聚合器中解

构出来即可。定义聚合器的方法如下。

```
export default {
    name: 'print.aggregator',
    aggregator: function (exports) {
        const bookList = []                                    //数组对象
        const buttonStyle = {                                  //样式常量
            width: '220px',
            …
        }
        const methodA = () => { … }                            //方法函数
        const methodB = () => { … }

        exports.buttonStyle = buttonStyle                      //对外暴露
        exports.methodA = methodA
        exports.methodB = methodB
        return exports;
    }({})
}
</script>
```

如果需要使用 buttonStyle、methodB,则导入聚合器再解构需要的元素。

```
import print from './print.aggregator.vue'
const { buttonStyle, methodB} = print.aggregator
```

可以分门别类地构建不同的聚合器,增强代码管理的方便性。

3.9 深入__vue_app__和_vnode

3.9.1 __vue_app__

要使用 Vue,就要先通过 createApp()创建一个应用实例 app,然后将该实例通过 mount 接口挂载在某个容器元素中。

```
<div id = "app"></div>
<script>
const app = Vue.createApp({
        setup() {
            …
        }
    })
app.mount('#app')
</script>
```

这背后发生了什么?不妨来简单了解一下:应用实例创建完成后,Vue 会在 HTML 层 app(容器元素)上添加一个名为“__vue_app__”的属性,其值就是应用实例对象 app。app 对象实际上是 Vue 应用所挂载宿主容器(app 层)而定义的一个对象属性,代表整个 Vue 应用的根。__vue_app__主要提供了以下函数(接口或方法)。

- component:注册一个全局组件。
- directive:注册一个全局指令。

- mount 和 unmount：mount 是将应用实例挂载到某个容器元素中，而 unmount 则卸载已挂载的应用实例。
- provide：数据提供。
- use：安装插件。

这些函数、接口或方法，在前面其实已经接触过。另外，__vue_app__还提供了以下属性。

- _component：代表根组件对象，包含当前应用所注册的组件、数据提供 provides、模板 template 等内容。
- _container：代表容器元素，其值一般是“#元素 tag+id”，例如 div#app。
- _context：代表应用上下文，包括内容较多，例如，当前应用、加载的组件、全局配置信息、全局错误处理、自定义指令等。

可以利用__vue_app__做某些特殊处理，例如，通过 component 接口可动态注册组件，利用 _container 可得到主页的纯文字信息以及构成页面的 HTML 元素，通过_context 可查询到 Vue 应用所注册的组件有哪些，甚至可进行卸载组件的处理。

示例：利用__vue_app__获取应用的 provide 数据，通过子组件显示在页面上。

(1) 创建 s3-9-1.html，主要代码如下。

```
<div id = "app">
    <unit - info></unit - info>
</div>

<script>
    …              //省略掉的代码，与 3.6.3 节 s3 - 6 - 3.html 中的这部分代码完全一样
    createApp({
        provides:{
            unit:'mgc.edu',
            access:'只读',
            issuer:'李超猛'
        },
        components: {
            'unit - info': defineAsyncComponent(() => loadModule('./s3 - 9 - 1.vue', options))
        }
    }).mount('#app');
</script>
```

这里用 provides 提供了三组数据，而 3.5.3 节则是用 provide('publicKey','0701@2003 $-Edu!')提供了一组数据。

(2) 编写 s3-9-1.vue，内容如下。

```
<template>
    <div>{{ info }}</div>
</template>
<script setup>
const provides = document.querySelector('#app')
                         .__vue_app__._component.provides
const unit = provides.unit
const access = provides.access
const issuer = provides.issuer
const info = `单位：${unit}\u3000 权限：${access}\u3000 签发人：${issuer}`
</script>
```

代码利用__vue_app__成功读取了 Vue 应用中的 provides 数据，并在页面显示如下结果。

单位：mgc.edu　权限：只读　签发人：李超猛

提示：该对象名的拼写不要弄错了，是前面连续两个下画线“_”，加上中间的“vue_app”，再加上后面连续两个下画线“_”拼接而成。

3.9.2　_vnode

_vnode 代表了应用的虚拟结点，提供的属性较多，重点关注以下属性。

- appContext：应用上下文，与前面的_context 类似。
- component：应用根组件。component 对象提供了相当丰富的内容，例如，props（传递参数）、components（组件）、ctx（setup 上下文对象）、emit（事件触发）、refs（模板引用）、slots（插槽）等。
- el：DOM 元素，例如，某个< div >层，可利用 el 实现页面元素的修改。

前面所举示例 s3-9-1.vue 中的：

```
const provides = document.querySelector('#app')
                         .__vue_app__._component.provides
```

可修改为以下代码，效果一样。

```
const provides = document.querySelector("#app")
                         ._vnode.appContext.app._component.provides
```

3.9.3　实战组件的动态注册和卸载

组件的动态注册还是比较容易的，使用 app.component()即可，但动态卸载则有难度。不过，使用__vue_app__或_vnode 就能轻松实现。下面的示例演示了__vue_app__和_vnode 结合的实战应用技巧。

示例：图 3-9 是某单位作者材料处理页面。“作者简介”菜单显示作者介绍信息；“材料处理”菜单展示作者代表性材料。每个材料用一个组件来处理。材料审查人员双击材料的某个图片，就可删除并卸载掉对应组件，界面则会自动重排。

图 3-9　作者材料处理

（1）编写页面主文件 s3-9-3.html。

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>组件的动态注册和卸载</title>
    <style>
        .menu-docs {
            position: absolute;
            width: 460px;
            height: 428px;
            font-size: 16px;
            text-align: center;
            top: 10px;
            background: rgba(236, 223, 223, 0.2);
        }
        .menu-ul {
            position: relative;
            background-color: #b0c3f0;
            width: 456px;
            height: 32px;
            top: -20px;
            font-size: 16px;
            padding: 0;
            margin: 5px;
        }
        .menu-ul li {
            font-size: 16px;
            float: left;                                    /* 横向排列 */
            list-style: none;                               /* 去掉列表符号 */
            padding: 5px;
            margin-inline: 50px;                            /* 拉大间距 */
        }
        .menu-ul li:hover {
            background-color: #f1625d;
            border-radius: 4px;
            cursor: pointer;
        }
    </style>
    <script src="js/vue.global.prod.min.js"></script>
</head>
<body>
<div id="myapp">
    <div class="menu-docs">
        <ul class="menu-ul">
            <li v-for="(item,index) in menus" @click="menuIndex = index">
                {{item.name}}
            </li>
        </ul>
        <component :is="current"></component>
    </div>
</div>
<script type="module" src="js/author.main.js"></script>
</body>
</html>
```

代码使用< component >的 is 属性来控制菜单组件的动态切换。单击菜单项时，改变 menuIndex 的值，就可以在 author. main. js 里面用 watch 侦听 menuIndex 值的变化，给 current 赋予新值，从而激发页面变化。

(2) 编写 Vue 主应用脚本文件 js/author. main. js。

```
import {AuthorDocs} from "./author.docs.js"                    //导入 AuthorDocs 组件

const { createApp, defineComponent, h, ref, watch} = Vue
const author = defineComponent({                               //定义“作者简介”菜单下对应内容的组件
    setup() {
        return () =>
            h('div', {
                style: {
                    width: '410px',
                    height: '360px',
                    margin: '0px auto',
                    border: '0px solid #00f',
                    boxShadow: '2px 2px 3px #ccc'
                }
            }, Vue.h('img', {src: 'image/author.png',}))
    }
})

const menus = [                                                //定义菜单项
    {id: 'author', name: '作者简介', module: author},          //作者简介组件
    {id: 'docs', name: '材料处理', module: AuthorDocs}         //材料处理组件
]
const watchMenu = {
    setup() {
        const menuIndex = ref(0)                               //默认显示 menus 的第 1 个元素
        const current = ref('author')                          //默认显示作者简介组件
        //侦听菜单项变化,并触发界面组件的变化
        watch(menuIndex, () => current.value = menus[menuIndex.value].id)
        return {menus, menuIndex, current}
    }
}
const myapp = createApp(watchMenu)
menus.forEach((item) => {
    myapp.component(item.id, item.module)                      //将组件注册到应用
})
myapp.mount('#myapp')
```

(3) 编写材料处理组件脚本文件 js/author. docs. js。

```
export const AuthorDocs = Vue.defineComponent({                //定义材料处理组件
    setup() {
        const docs = [                                         //个人材料将被渲染成 4 个组件并注册到 Vue 应用
            {id: 'computerApp', image: 'image/computerapp.jpg'},
            {id: 'computerGc', image: 'image/computergc.jpg'},
            {id: 'rcp', image: 'image/rcp.jpg'},
            {id: 'springBoot', image: 'image/springboot.png'}
        ]
        const myapp = document.querySelector("#myapp").__vue_app__
        docs.forEach((doc, index) =>
            new Promise(() =>                                  //异步注册组件到 Vue 应用
                myapp.component(doc.id, Vue.h(asyncComponent, {     //h()函数渲染
```

```
                    id: doc.id,
                    image: doc.image
                }))
            ).then()
        )
        return {docs}
    },
    template: `
            <div>
              <template v-for="(doc,index) in docs">
                  <component :is="doc.id"></component><!-- 动态构建组件 -->
              </template>
            </div>
            `
})

const asyncComponent = Vue.defineAsyncComponent(          //异步组件
    () => Promise.resolve({
        props: {
            id: String,                                   //材料的 id
            image: String                                 //材料对应图片 URL
        },
        setup(props) {
            const {id, image} = props
            const delDocs = (id) => {
                const appVNode = document.querySelector("#myapp")._vnode
                const children = appVNode.el.children[1].children                    ①
                const appComponents = appVNode.appContext.components                 ②
                Object.keys(appComponents)
                    .filter(key => appComponents[key].__v_isVNode)                   ③
                    .forEach((key, index) => {            //遍历组件
                        if (key === id) {                 //如果是用户当前双击的组件
                            children[index].remove()      //从数组中移除当前 div 层对象
                            delete appComponents[key]     //删除对应组件
                        }
                    })
            }
            return {id, image, delDocs}
        },
        template: `<div style="float:left;padding:0 20px 10px 2px;">
                    <img :src="image" width="200" height="170"
                      @dblclick="delDocs(id)" title="双击删除">
                  </div>`
    })
)
```

① appVNode.el 是页面的 元素，它有两个子元素：第 1 个子元素是<ul class="menu-ul">，第 2 个子元素是<component :is="current"></component>渲染出的层 ，所以 appVNode.el.children[1]就是指这个渲染出的层。而这个层有 4 个子元素，每个子元素是一个 div 对象，对应了作者的 4 种材料，这就是 appVNode.el.children[1].children 所代表的内容。

② 虚拟结点的 appContext 应用上下文的 components 属性，其值是组件对象的集合，里面包含整个应用的 6 个组件，分别是 author、docs、computerApp、computerGc、rcp、springBoot，双击时需要卸载后面 4 个组件中的某一个。

③ __v_isVNode 是一个布尔型属性，用来判断当前元素是否是虚拟结点。注意其拼写：与__vue_app__拼写类似，前面是两个“_”，再加上“v_isVNode”。这里过滤掉其他非虚拟结点，只保留 4 个分别代表 computerApp、computerGc、rcp、springBoot 组件的虚拟结点(div 层)。为什么这 4 个是虚拟层？因为这 4 个组件是用 h()函数渲染的。

当然，双击删除时，只是从虚拟结点中进行卸载。单击“作者简介”，再单击“材料处理”，会恢复到初始状态，因为并没有真正去修改原始数据。这里只是组件卸载的模拟示例而已，目的是为了更好地学习__vue_app__、_vnode 的应用方法。仔细体会其用法，就可灵活运用到实战中。

3.10 状态管理

什么是状态管理？一个典型的场景是：一些页面需要根据用户登录情况来判断是否允许访问，这就需要用户登录状态数据。用户登录状态可能超时失效，也可能主动退出了，这意味着需要更新状态数据，以便其他页面也能及时响应。状态就像一个仓库，是应用程序的一部分，例如，用户登录信息、网站配色等，可以在需要时从任何地方访问其内容。状态管理是一种设计模式，可管理并同步所有组件中的状态数据。

3.10.1 Pinia 简介

Pinia 是 Vue 官方推荐的专属状态管理库，是一个类型安全、容易扩展、模块化设计的轻量级状态库，官网地址为 https://pinia.vuejs.org/zh/。在 Node.js 环境下，可通过 npm install pinia 命令进行安装，非 Node.js 环境可通过 CDN 使用。

```
<script src=" https://unpkg.com/pinia@2.1.7/dist/pinia.iife.prod.js"></script>
```

或者，利用下面的链接到 cdnjs 网站下载。

```
https://cdnjs.cloudflare.com/ajax/libs/pinia/2.1.7/pinia.iife.prod.min.js
```

将下载到的 pinia.iife.prod.min.js 保存到项目 js 文件夹下，本书采用这种方式，并使用 2.1.7 这个版本。值得注意的是，Pinia 需要 vue-demi 的支持，下载地址为

```
https://cdnjs.cloudflare.com/ajax/libs/vue-demi/0.14.6/index.iife.min.js
```

最终，页面需要包含以下两个引用。

```
<script src="js/index.iife.min.js"></script>
<script src="js/pinia.iife.prod.min.js"></script>
```

3.10.2 数据状态 State

数据状态其实反映的是用户所关心数据的变化情况，例如，用户 logo 图像从 a.png 变更为 b.png。为了管理数据状态变化，每个 Vue 应用有且仅有一个数据存储仓库(store)的实例，store 就是一个容器，主要包含三个部分的内容：状态数据(state)、计算函数(getter)、数据更改(action)。

Pinia 支持响应式地进行数据状态存储的管理，可同步或异步修改 store 中的状态值。先设计好需要管理的状态数据，然后利用 Pinia 提供的 defineStore()定义 store。defineStore()函数有两

个参数：第一个参数，是一个唯一值 id；第二个参数可传入 setup() 函数或 Option 可选值对象，或者直接传入 Option 对象，将 id 作为该对象的属性。

可以在项目的 store 文件夹下创建一个 index.js 文件，然后定义一个名为 useStore 的状态 store。

```
const {defineStore} = Pinia

export const useStore = defineStore('user', () => {
    const username = null
    const score = 0
    return { username, score }
})
```

这是 setup()组合式函数写法，也可以用 Option 对象方式。

```
export const useStore = defineStore({
     id: 'user',
     state: () => {
        return {
            username: null,
            score: 0
        }
    }
})
```

代码中的 state 是 store 中最重要的状态数据。state 通常使用箭头函数来返回数据。或者简写为

```
export const useStore = defineStore('user', {
     state: () => ({
          username: null,
          score: 0
    })
})
```

不过，推荐使用更清晰、更简化、更方便管理的写法。先定义一个 states.js 文件，专门用来定义各种状态数据量。

```
export default {
    name: 'store.states',
    username: null,
    score: 0,
    …//其他 state 数据
}
```

然后，利用对象展开运算符，这样来定义状态 store：

```
import storeStates from './states.js'

const {defineStore} = Pinia
export const useStore = defineStore('user', {
    state: () => ({...storeStates})
})
```

现在还无法进行有效的状态管理。必须创建一个 Pinia 实例，并将其传递给 Vue 应用。Pinia 提供了 createPinia()函数来创建 Pinia 实例。

```
Vue.createApp({…})
    .use(Pinia.createPinia())
    .mount('#app')
```

到此，Pinia 理论上可进行数据的状态管理了。不过，还需要另外两个部分 Getter 和 Action，才具有实际应用价值。

3.10.3 计算属性 Getter

如果需要从 store 的 score 中派生出一些状态，例如，加减处理，就需要用到计算函数 getter。Getter 是 store 中 state 数据的计算值。Pinia 通过 defineStore()中的 getters 属性来定义这些计算值。一般使用箭头函数，将 state 作为第一个参数。

```
export const useStore = defineStore('user', {
     state: () => ({
          username: null,
          score: 0
        }),
    getters: {
          scorePlus: (state) => state.score++
    }
})
```

同样，也可以定义一个 getters.js 文件，专门用来处理 getter 计算函数。

```
export default {
    name: 'store.getters',
    scorePlus: (state) => state.score++,
    …//其他 getter 函数
}
```

现在，导入 getters.js 文件，store 的定义变成下面这样。

```
import storeStates from './states.js'
import storeGetters from './getters.js'

const {defineStore} = Pinia
export const useStore = defineStore('user', {
    state: () => ({...storeStates}),
    getters: {...storeGetters}
})
```

虽然 Pinia 允许直接对状态数据进行修改，例如 useStore.score=90.6，不过，统一通过 Action 更改状态数据，是更好的选择。

3.10.4 数据更改 Action

Action 类似于 Vue 中的 method，通过 defineStore()中的 actions 属性来定义。

```
export const useStore = defineStore('user', {
     state: () => ({
          username: null,
          score: 0
```

```
    }),
    actions: {
         scorePlus(num){ this.state.score += num}
    }
})
```

除了同步执行，Action 也可以异步执行，例如：

```
actions: {
    async scorePlus(username) {
        this.score += await getScore(username)
    }
}
```

代码通过 getScore()函数获取指定用户名的加分数。同前面一样，也可以定义一个 actions.js 文件，专门用来处理数据更改。

```
export default {
    name: 'store.actions',
    scorePlus: function (num) {
        this.score += num
    },
    …//其他 action 函数
}
```

然后导入 actions.js 文件，最终将 store 的定义确定为如下形式。

```
import storeStates from './states.js'
import storeGetters from './getters.js'
import storeActions from './actions.js'

const {defineStore} = Pinia
export const useStore = defineStore({
    id: 'user',
    state: () => ({...storeStates}),
    getters: {...storeGetters},
    actions: {...storeActions}
})
```

推荐采用这样的处理思路。从第 6 章开始，将使用 Pinia 进行项目的状态管理，请参阅相关章节内容。

3.10.5　项目中的应用方式

项目中可能需要频繁使用状态数据，如何在项目中更好地使用 Pinia 进行状态管理？下面列出 4 种方式供参考。

1. 每次使用时 import

每次组件需要进行状态数据处理时，import 导入：

```
import {useStore} from './store/index.js'
useStore.username = '杨过'
```

这种方式，每次需要导入看起来麻烦，其实还是比较方便的，但某些场景下（例如非 Node.js 环境的 SFC 中）可能存在无法导入问题。

2. 挂载到 window 对象

```
import {useStore} from './store/index.js'
const window.useStore = useStore
```

然后,就可在应用的任何地方利用 window.useStore 或 window['useStore']进行 store 的各种处理。当然,使用这种方式,得忍受其全局污染。

3. 作为 Vue 应用的全局变量

挂载到全局变量上,使用成本并没有增加多少。将全局变量命名为 $useStore:

```
import {useStore} from './store/index.js'
app.config.globalProperties.$useStore = useStore()
```

然后,在组件中需要利用 getCurrentInstance()获取。

```
const { getCurrentInstance } = Vue
const {proxy} = getCurrentInstance()
const useStore = proxy.$useStore
useStore.scorePlus(5)
```

本书在后续章节中采用这种方法。

4. 利用__vue_app__

如果觉得前面三种方式都不理想,而是希望在组件中直接使用,则可借助于 3.9.3 节介绍的__vue_app__,通过 defineStore()中定义的唯一 id 号 user 来处理状态数据。这种方式更底层些。

```
const provides = document.querySelector('#app').__vue_app__._context.provides
const symbolKey = Reflect.ownKeys(provides)
        .find(key => key.toString() === 'Symbol()')      //查找 provides 中的 Symbol 键
const useStore = Vue.toRaw(provides[symbolKey].state._value.user)
```

provides 是一个 Symbol()数据,里面包含具体的状态数据。Symbol 是在 ECMAScript 6 (ES6) 标准中引入的,常用于表示独一无二的标识符。这里的 Symbol 数据无法直接获取,需要先通过 Reflect.ownKeys 获取到对应的 Symbol()键。然后,根据该键的 state._value.user 属性,拿到 user。但是,这个 user 又被"包裹"成了代理对象 Proxy(Object),不能直接访问其值,需要用 toRaw()函数取出其中的数据。

现在可以直接访问状态数据了,例如 useStore.username。

3.11 场景应用实战

下面通过两个场景应用,练习并加深 Vue 应用技巧与方法。

3.11.1 下拉选择框联动

用下拉选择框实现学院、专业之间的联动,如图 3-10 所示。当选择不同院系时,专业名称发生相应变化。

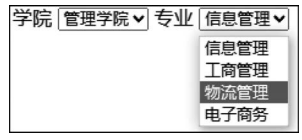

图 3-10 院系专业联动

代码如下。

```
<div id="combo">
    学院
    <select v-model="index"><-- 绑定 index 值 -->
       <option v-for="(college,i) in colleges" :value="i">
           {{college.name}}
       </option>
    </select>
    专业
    <select>
       <option v-for="(major,j) in colleges[index].majors" :value="j">
           {{major}}
       </option>
    </select>
</div>
<script>
    Vue.createApp({
       setup() {
         const {reactive, ref} = Vue
         const colleges = reactive(
           [
              {name:'管理学院',majors:['信息管理','工商管理','物流管理','电子商务',]},
              {name: '会计学院', majors: ['会计学', '工程造价', '财务管理']},
              {name: '传媒学院', majors: ['动画', '广告', '数字媒体', '新闻传播']},
              {name: '经济学院', majors: ['投资学', '财政学', '金融学', '国际贸易']}
           ]
         )
         const index = ref(0)                              //默认管理学院
         return {colleges, index}
       }
    }).mount('#combo')
</script>
```

每个学院用一个 JSON 对象表示，多个学院则用数组存放。当选择不同的学院时，通过 index 的改变，响应式地改变专业下拉选择框的值。简单的代码，很好地阐释了 Vue 数据驱动的响应式处理思想！

3.11.2　动态增删图书

动态增删图书，在 2.4.2 节已经用 RxJS 实现过了，现在要求用 Vue 来实现该功能。具体功能要求请参阅 2.4.2 节，这里只新增一个要求：页面上增加拟入库数、拟删除数，如图 3-11 所示。

图 3-11　新增图书

由于 Vue 是通过数据变化来驱动页面变化的，因此处理思路与 2.4 节有很大不同：用一个响应式的 books 数组管理页面新增、删除图书的数据。图书的增加，只需要向 books 数组中 push 新的图书，删除则是过滤掉 books 中已勾选的图书，然后将剩下的图书 push 到 books 里面。books 数据的改变，自然会引起页面响应式改变。主要代码如下(CSS 代码省略)。

```
<table id="bookTable">
    <tr>
        <td colspan="2" nowrap class="title">新增图书</td>
    </tr>
    <tr>
        <td nowrap>书 名</td>
        <td nowrap id="bookList">
            <div v-for="(book,index) of books"> <!-- 循环 books 数组,构建图书列表 -->
                <input type="text" v-model="book.name" autofocus>
                <input type="radio" :name="'r'+index" value="01" v-model="book.checked"/>社
会科学
                <input type="radio" :name="'r'+index" value="02" v-model="book.checked"/>自
然科学
                <input type="checkbox" :name="'ck'+index" v-model="book.deleted"/>
            </div>
        </td>
    </tr>
    <tr>
        <td colspan=2 class="button">
            <img src="image/delete.png" @click="delBook" title="删除图书">  
            <img src="image/add.png" @click="addBook" title="新增图书"/>  
            <img src="image/save.png" title="保存入库"/><br>
            <!-- 筛选出删除标志 deleted 的值为 false 的图书 -->
            拟入库数:{{books.filter(book =>!book.deleted).length}} 
            <!-- 筛选出删除标志 deleted 的值为 true 的图书 -->
            拟删除数:{{books.filter(book =>book.deleted).length}}
            <br><span class="message">{{message}}</span>
        </td>
    </tr>
</table>
<script>
    Vue.createApp({
        setup() {
            const {ref, reactive, nextTick} = Vue
            const books = reactive([            //响应式数组对象
            {
                name: '',                       //书名
                checked: '01',                  //默认勾选“社会科学”
                deleted: false                  //删除标志,与复选框的值绑定。默认不勾选
            }
            ])
           let message = ref(null)
           const addBook = () =>{
            const newBook = [...books]          //新增的图书,直接用 books 对象的值
            newBook.checked = books[books.length - 1].checked //默认勾选情况
            books.push(newBook)                 //添加到 books 数组中,响应式改变界面数据
            nextTick(() =>{                     //等待 DOM 同步更新完成后设置新增图书文本框焦点
                const s = `#bookList>div:nth-child(${books.length})>input:nth-child(1)`
                const focused = document.querySelector(s)
                focused.focus()                 //“书名”文本框获得焦点
```

```
            })
        }
            const delBook = () => {
            //过滤掉 deleted 为 true 的图书,即移去待删除图书以后,剩下的图书
                const restBooks = books.filter(book => !book.deleted)
                let count = restBooks.length;                          //剩下图书的数量
                if (count === books.length)                            //与 books 长度相同
                    message.value = '未选择待删除图书!'
                else if (count === 0)                                  //剩下图书数量为 0
                    message.value = '不能全删,至少需要保留一本图书!'
                else if (confirm("确认删除所选的全部图书?")) {
                    books.length = 0                                   //清空 books 中的旧数据
                    restBooks.forEach(book => books.push(book))        //加入剩下的图书
                    message.value = ''
                }
            }
            return {books, message, addBook, delBook}
        }
    }).mount('#bookTable')
</script>
```

这个示例再次体现了 Vue 响应式数据驱动的思想!