

04 | 模版语法和JSX语法：你知道Vue可以用JSX写吗？

2022-11-28 杨文坚 来自北京



天下无鱼

<https://shikey.com/>

《Vue 3 企业级项目实战课》

[课程介绍 >](#)



讲述：杨文坚

时长 16:48 大小 15.34M



你好，我是杨文坚。

前面几节课，我们讲解了很多 Vue.js 3 编译相关的内容，了解完 Vue.js 两个编译打包工具后，我们是时候要开始学习如何使用 Vue.js 3 进行实际的代码开发了。

这节课，我主要会从 Vue.js 的两种主要开发语法进行讲解，它们分别是模板语法和 JSX 语法。从中你不仅能了解到两种开发语法的差异，还可以知道怎么因地制宜地根据需求场景选择合适的语法，从而扩大个人的技术知识储备，更从容地应对用 Vue.js 3 开发项目过程中遇到的各种问题。

Vue.js 从版本 1.x 到版本 3.x，官方代码案例和推荐使用都是模板语法，那么这里我们也根据官方的推荐，优先来了解一下模板语法是怎么一回事。

什么是模板语法？

我们可以把 Vue.js 的模板语法，直接理解为 **HTML 语法的一种扩展**，它所有的模板节点声明、属性设置和事件注册等都是按照 HTML 的语法来进行扩展设计的。按照官方的说法就是“所有的 Vue 模板都是语法层面合法的 HTML，可以被符合规范的浏览器和 HTML 解析器解析”。

现在我举个例子，带你了解下模板语法的概念及其不同内容的作用。代码如下所示：

 复制代码

```
1 <template>
2   <div class="counter">
3     <div class="text">Count: {{state.count}}</div>
4     <button class="btn" v-on:click="onClick">Add</button>
5   </div>
6 </template>
7
8 <script setup>
9   import { reactive } from 'vue';
10  const state = reactive({
11    count: 0
12  });
13  const onClick = () => {
14    state.count ++;
15  }
16 </script>
17
18 <style>
19 .counter {
20   padding: 10px;
21   margin: 10px auto;
22   text-align: center;
23 }
24
25 .counter .text {
26   font-size: 28px;
27   font-weight: bolder;
28   color: #666666;
29 }
30
31 .counter .btn {
32   font-size: 20px;
33   padding: 0 10px;
34   height: 32px;
35   min-width: 80px;
36   cursor: pointer;
37 }
38 </style>
39
```

这是基于 Vue.js 3 的模板语法实现的加数器组件，代码基础结构都是基于 HTML 语法实现的，主要由视图模板、JavaScript 脚本代码和 CSS 样式代码构成的。我们拆分开来具体看看。



首先我们来看看视图层代码：

 复制代码

```
1 <template>
2   <div class="counter">
3     <div class="text">Count: {{state.count}}</div>
4     <button class="btn" v-on:click="onClick">Add</button>
5   </div>
6 </template>
```

上述视图代码中，只能有一个最外层的 `template` 标签，`template` 内部可以允许存在多个 `template` 标签，用来定义模板的“插槽”位置（slot）等更多插槽相关信息，你可以查看 [官方对插槽的说明](#) 了解一下。

Vue.js 的模板可以直接使用 HTML 语法的属性（Attribute），例如 `class` 也可以直接在 Vue.js 的模板中使用，但是 Vue.js 自己定义了一些属性语法，例如 `v-on`，这个就是 Vue.js 模板绑定事件的语法。以此类推，你大概可以猜到大部分 Vue.js 自定义的模板属性语法，都是以“v-”为前缀的。更多 Vue.js 的模板语法，你可以查看 [官方文档](#)。

接下来我们看看 JavaScript 脚本代码：

 复制代码

```
1 <script setup>
2   import { reactive } from 'vue';
3   const state = reactive({
4     count: 0
5   });
6   const onClick = () => {
7     state.count ++;
8   }
9 </script>
10
```

在模板语法中，JavaScript 代码只能放在 `script` 标签里，而且同一个文件里只能有一个顶级的 `script` 标签。



上述代码使用的是 Vue.js 的 Composition API，所以必须在 `script` 标签中声明 `setup` 属性。我们后续所有内容都默认是基于 Composition API 来讲解 Vue.js 3 里的 JavaScript 代码操作。这是因为它是官方推荐的 API 使用方式，使用起来简单清晰，方便复用逻辑代码，同时这也是 Vue.js 3 诞生的特色。

最后再来看下 CSS 样式代码：

复制代码

```
1 <style>
2 .counter {
3   padding: 10px;
4   margin: 10px auto;
5   text-align: center;
6 }
7
8 .counter .text {
9   font-size: 28px;
10  font-weight: bolder;
11  color: #666666;
12 }
13
14 .counter .btn {
15   font-size: 20px;
16   padding: 0 10px;
17   height: 32px;
18   min-width: 80px;
19   cursor: pointer;
20 }
21 </style>
```

这些代码是模板语法里的 CSS 样式代码，具体使用方式跟 HTML 里使用 CSS 代码一致，唯一不同是可以加上 `scoped` 和 `lang` 属性。

`scoped` 属性可以在编译 Vue.js 模板语法代码的时候，用随机数来定义样式选择器名称，保证 CSS 不会干扰页面上同名的 CSS 选择器，例如下面代码所示：

复制代码

```
1 <div class="counter"></div>
```

```
2 <style scoped>
3 .counter { /*...*/ }
4 </style>
5
6 <!-- 当style加上scoped 后编译成 -->
7
8 <div class="counter" data-v-xxxxx></div>
9 <style>
10 .counter[data-v-xxxxx] { /*...*/ }
11 </style>
12
```



而 `lang` 属性可以赋值声明定义用了其它 CSS 语法，例如 `lang="less"` 就是用了 Less 语法来写的 CSS，但是需要在 Vite 等对应编译配置加上 Less 编译插件。

上面的整体代码就是用一个 Vue.js 3 模板语法实现一个组件，如果有一个组件需要引用这个“计数器”组件，就直接用 `import` 来引用就好了，代码如下：

复制代码

```
1 <template>
2   <div class="app">
3     <Counter />
4   </div>
5 </template>
6
7 <script setup>
8 import Counter from './counter.vue'
9 </script>
10
11 <style>
12 .app {
13   width: 200px;
14   padding: 10px;
15   margin: 10px auto;
16   box-shadow: 0px 0px 9px #00000066;
17   text-align: center;
18 }
19 </style>
```

讲到这里，你是不是反应过来了，其实只要了解过 HTML 语法，就能很容易上手 Vue.js 的模板语法。而且，Vue.js 从版本 1.x 到版本 3.x，官方代码案例和推荐使用都是模板语法，因为模板语法更加简单易用。

不过，既然 Vue.js 官方代码案例和推荐使用都是模板语法，为什么官方还要实现一套与模板语法不同的 JSX 语法呢？



其实这个问题我们可以直接在 Vue.js 官网找到答案，官网就这么写着：“在绝大多数情况下，Vue 推荐使用模板语法来创建应用。然而在某些使用场景下，我们真的需要用到 JavaScript 完全的编程能力。这时渲染函数就派上用场了。”

这就是说，虽然官方推荐你用模板语法来写 Vue.js 3 代码，但是有些功能场景用模板语法可能会很难实现，甚至不能实现，那么就需要用到 JSX 语法来辅助实现了。而且，Vue.js 在 2.x 版本时候已经开始支持 JSX 语法了。那么，Vue.js 3 的 JSX 语法是怎样的呢？

Vue.js 3 的 JSX 语法是怎样的？

在讲解 Vue.js 3 之前，我先来给你分享一下，什么是 JSX 语法。

JSX 语法，是 JavaScript 语法的一种语法扩展，支持在 JavaScript 直接写类似 HTML 的模板代码，你可以直接理解为“HTML in JavaScript”。从目前在网上能找到的资料来看，JSX 语法最早用于 React.js，但不是 React.js 独有的写法，目前有很多框架支持 JSX 写法，例如 Vue.js 和 Solid.js（一种类似 React.js 写法的前端框架）等。

现在，我把上面的 Vue.js 3 的模板语法实现的“加数器”组件换成 JSX 语法实现，你可以对比看看这两个语法的实现差异，如下所示：

复制代码

```
1 import { defineComponent, reactive } from 'vue';
2
3 const Counter = defineComponent({
4
5   setup() {
6     const state = reactive({
7       count: 0
8     });
9     const onClick = () => {
10       state.count ++;
11     }
12     return {
13       state,
14       onClick,
15     }
16   },
17
```

```

18   render(ctx) {
19     const { state, onClick } = ctx;
20     return (
21       <div class="counter">
22         <div class="text">Count: {state.count}</div>
23         <button class="btn" onClick={onClick}>Add</button>
24       </div>
25     )
26   }
27 });
28
29 export default Counter;

```



现在，我们类比模板语法，逐步分析下这个 JSX 语法实现的“加数器”组件。

JSX 语法其实可以直接看做是纯 JavaScript 文件代码，在 JavaScript 文件代码里定义 Vue.js 3 组件可以通过 API `defineComponent` 来进行声明定义：

复制代码

```

1 import { defineComponent } from 'vue';
2 const Counter = defineComponent({
3   // ...
4 })

```

而模板语法有组件视图层相关的代码，类比 JSX 语法里定义组件中的 `render` 方法，如下述代码所示：

复制代码

```

1 const Counter = defineComponent({
2   // ...
3   render(ctx) {
4     const { state, onClick } = ctx;
5     return (
6       <div class="counter">
7         <div class="text">Count: {state.count}</div>
8         <button class="btn" onClick={onClick}>Add</button>
9       </div>
10    )
11  }
12  // ...
13 });
14

```

上述代码中，render函数返回的代码，就是 JSX 的写法，用来描述 HTML 模板内容。这里需要注意的是，所有 JSX 写法中都是用单大括号“{state.count}”来作为内部变量处理，而 **Vue.js 3 模板语法是通过双大括号来表示“{{state.count}}”**，单大括号描述变量这个是 JSX 通用写法，Vue.js 的 JSX 语法也是遵循了这个通用写法。

在模板语法中，模板的 <script> 标签里有一段 JavaScript 逻辑代码，这段 JavaScript 的逻辑代码，就是 JSX 语法中的 defineComponent 里除掉 render 函数外剩下的代码内容，如下代码：

复制代码


```
1 const Counter = defineComponent({
2   // 这里还可以是定义属性和组件引用
3   props: {},
4   components: {},
5
6   // ...
7   setup() {
8     const state = reactive({
9       count: 0
10    });
11    const onClick = () => {
12      state.count ++;
13    }
14    return {
15      state,
16      onClick,
17    }
18  },
19  // ...
20 });
```

看在这里，你会不会觉得少了什么东西？哈哈，是不是觉得少了 CSS 样式代码？模板语法中 style 存放的 CSS 代码，在 JSX 语法中，又是在哪个位置呢？

我们先回到最开始的 JSX 介绍中看看。我们说了，JSX 其实也是 JavaScript 代码，在 JavaScript 代码中引用 CSS 代码，一般都是直接 import 对应的 CSS 文件。所以，在 Vue.js 中通过 JSX 语法开发组件，组件的 CSS 代码也是放在独立的 CSS 文件，最后通过 import 引用的，如下代码所示：

复制代码

```
1 import './counter.css'
```


到了这里，你是不是觉得 JSX 语法跟模板语法类比起来，都能找到一一对应关系，**差别好像**不是很大？ <https://shikey.com/>

其实差别还是有的。只是因为上述的“加数器”组件案例只是简单的组件场景，而实际企业项目开发中我们会遇到很多五花八门的需求场景，这个时候模板语法和 JSX 语法的区别就体现出来了。接下来我就来讲解模板语法和 JSX 语法在实际项目中开发的有什么区别。

模板语法和 JSX 语法有什么区别？

首先，最大的区别就是模板语法能通过设置**标签 `<style>` 属性 `scoped`**，让 CSS 和对应的 DOM 在编译后能加上随机的 CSS 属性选择器，避免干扰其它同名 class 名称的样式。

而在 JSX 语法中并没有可以设置 `scoped` 的地方，所以 JSX 语法在使用样式 class 名称的时候，不能配置 `scoped` 避免 CSS 样式干扰。

除了样式的 `scoped` 配置差异外，还有更大的差异是体现在**实现需求场景**上，例如动态的组件渲染。假设我们现在有这么个需求，可以动态对组件进行顺序颠倒，如下述两张效果图所示：





这个需求如果要通过 Vue.js 3 的模板语法实现，可以这么写：

复制代码

```
1 <template>
2   <div class="app">
3     <div v-if="isReverse === false">
4       <Module01 />
5       <Module02 />
6       <Module03 />
7       <Module04 />
8     </div>
9     <div v-else>
10      <Module04 />
11      <Module03 />
12      <Module02 />
13      <Module01 />
14    </div>
15    <button class="btn" @click="onClick">转换顺序: {{isReverse}}</button>
16  </div>
17 </template>
18
19 <script setup>
20 import { ref } from 'vue';
```

```
21 import Module01 from './module01.vue';
22 import Module02 from './module02.vue';
23 import Module03 from './module03.vue';
24 import Module04 from './module04.vue';
25
26 const isReverse = ref(false);
27 const onClick = () => {
28   isReverse.value = !isReverse.value;
29 }
30 </script>
31 <style>
32 /* 完整样式代码请看后续附带仓库链接 */
33 </style>
```



你可以看到，这个代码是通过一个变量 `isReverse` 来控制显示组件的正序和倒序，但是要写两次的顺序的模板代码，如下所示：

```
1 <div v-if="isReverse === false">
2   <Module01 />
3   <Module02 />
4   <Module03 />
5   <Module04 />
6 </div>
7 <div v-else>
8   <Module04 />
9   <Module03 />
10  <Module02 />
11  <Module01 />
12 </div>
```

 复制代码

这样子写代码虽然可以完成功能需求，但是会给后续的维护带来一定的难度。为什么这么说呢？这不是明明已经完成功能了吗，而且代码也很清晰呀，怎么会有后续维护难度呢？

这是因为企业中的需求是一直变化的。例如这次需求是实现组件的顺序的正序和倒序操作，那么如果下次要实现组件的其它排序，是不是意味着要多个变量来控制多个顺序的组件布局模板呢？这就会导致相关组件顺序控制的代码量翻倍增长。

这时候，**JSX** 语法就可以来解决这种“动态”的问题了。我们再用 **JSX** 实现一次上述功能的代码，如下所示：



```
1 import { defineComponent, ref } from 'vue';
2 import Module01 from './module01.vue';
3 import Module02 from './module02.vue';
4 import Module03 from './module03.vue';
5 import Module04 from './module04.vue';
6
7 const App = defineComponent({
8
9   setup() {
10     const isReverse = ref(false);
11     const onClick = () => {
12       isReverse.value = !isReverse.value;
13     }
14     return {
15       isReverse,
16       onClick,
17     }
18   },
19
20   render(ctx) {
21     const { isReverse, onClick } = ctx;
22     const mods = [
23       <Module01 />,
24       <Module02 />,
25       <Module03 />,
26       <Module04 />
27     ]
28     isReverse === true && mods.reverse();
29     return (
30       <div class="app">
31         {mods.map((mod) => {
32           return mod;
33         })}
34         <button class="btn" onClick={onClick}>
35           转换顺序: `${isReverse}`
36         </button>
37       </div>
38     )
39   }
40 });
41
42 export default App;
```

上述代码中，控制组件的动态顺序核心代码是这样的：

```
1 const mods = [
2   <Module01 />,
3   <Module02 />,
4   <Module03 />,
5   <Module04 />
```

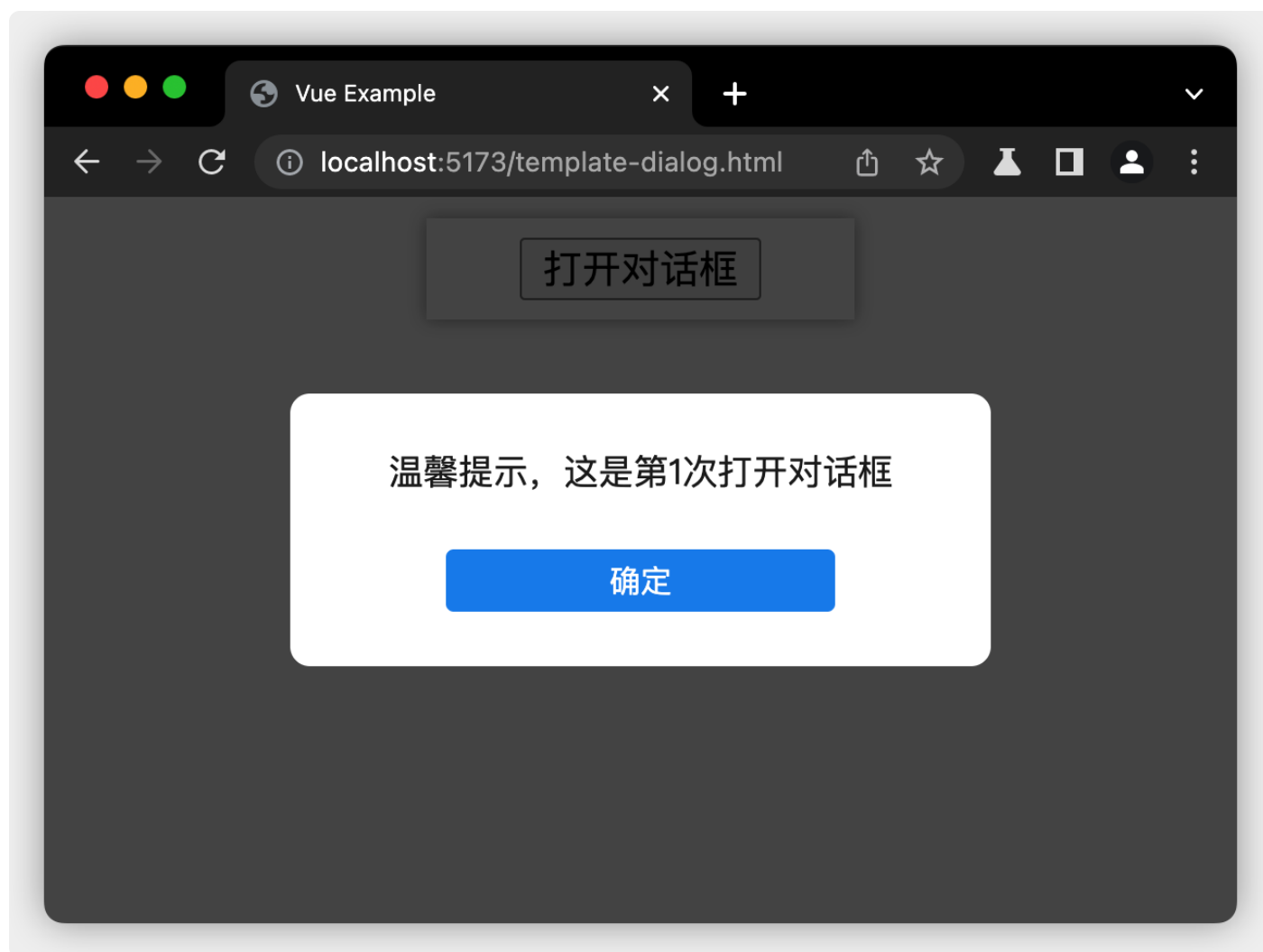
```
4   <Module03 />,
5   <Module04 />
6 ]
7 isReverse === true && mods.reverse();
```



你有没有发现，控制组件顺序的其实就是通过一个 **JSX 的组件数组**来进行的，如果后续遇到项目需求的变化，要求按各种顺序显示组件，那么我们只需要修改这个 **JSX** 数组的顺序就好了，不需要写多套顺序模板，是不是觉得代码量和维度难度一下子就降低很多呢？

不过，这时你可能会想挑战我：像这种动态顺序，如果项目团队用模板语法多写几次组件顺序也能接受的话，是不是等于 **JSX** 语法也没有优势呢？

那我们再来看一种场景，看看你如果不用 **JSX** 语法，能不能接受这样的维护成本。这个场景就是“动态组件的条件渲染”，例如常见的对话框条件显示：



如果用模板语法怎么来实现呢？我们先看对话框代码：

```

1 <template>
2   <div v-if="props.show" class="v-dialog-mask">
3     <div class="v-dialog">
4       <div class="v-dialog-text">
5         {{props.text}}
6       </div>
7       <div class="v-dialog-footer">
8         <button class="v-dialog-btn" @click="onOk">确定</button>
9       </div>
10    </div>
11  </div>
12 </template>
13 <script setup >
14 import { toRef, toRefs, computed } from 'vue';
15 const props = defineProps({
16   text: String,
17   show: Boolean,
18 });
19 const emits = defineEmits(['onOk']);
20
21 const onOk = () => {
22   emits('onOk');
23 }
24 </script>
25 <style>
26 /* 完整样式代码请看后续附带仓库链接 */
27 </style>

```



再看使用对话框代码：

```

1 <template>
2   <div class="app">
3     <button class="btn" @click="onClickOpenDialog" >打开对话框</button>
4   </div>
5   <Dialog
6     :show="showDialog"
7     :text="showText"
8     @onOk="onDialogOk"
9   />
10 </template>
11 <script setup>
12 import { ref } from 'vue';
13 import Dialog from './dialog.vue';
14
15 const showDialog = ref(false);
16 const showCount = ref(0);
17 const showText = ref('温馨提示，这是一个对话框')

```

```

18  const onClickOpenDialog = () => {
19    showDialog.value = true;
20    showCount.value += 1;
21    showText.value = `温馨提示，这是第${showCount.value}次打开对话框`
22  }
23
24  const onDialogOk = () => {
25    showDialog.value = false;
26  }
27  </script>
28  <style>
29    /* 完整样式代码请看后续附带仓库链接 */
30  </style>
31

```



天下无鱼

<https://shikey.com/>

上述是用模板语法实现的对话框“条件动态”显示，你能看到，如果要控制一个对话框显示，不仅需要有一个变量 `showDialog` 来控制，还需要把一个 `<Dialog>` 标签“埋在”模板里；如果后续有多个对话框显示，就需要控制多个变量和多个对话框标签。这样子代码虽然能运行，但是维护起来就比较冗余了。

那么换成 **JSX** 写法会是怎样呢？我这里给你看一下 **JSX** 写法的对话框组件：

 复制代码

```

1  import { defineComponent, reactive, createApp, h, toRaw } from 'vue';
2
3  const Dialog = defineComponent({
4    props: {
5      text: String,
6    },
7    emits: [ 'onOk' ],
8    setup(props, context) {
9      const { emit } = context;
10     const state = reactive({
11       count: 0
12     });
13     const onOk = () => {
14       emit('onOk')
15     }
16     return {
17       props,
18       onOk,
19     }
20   },
21   render(ctx) {
22     const { props, onOk } = ctx;
23     return (
24       <div class="v-dialog-mask">

```



```

25     <div class="v-dialog">
26         <div class="v-dialog-text">
27             {props.text}
28         </div>
29         <div class="v-dialog-footer">
30             <button class="v-dialog-btn" onClick={onOk}>确定</button>
31         </div>
32     </div>
33 </div>
34 )
35 }
36 });
37
38 export function createDialog(params = {}) {
39     const dom = document.createElement('div');
40     const body = document.querySelector('body');
41     body.appendChild(dom);
42     const app = createApp({
43         render() {
44             return h(Dialog, {
45                 text: params.text,
46                 onOk: params.onOk
47             });
48         }
49     });
50     app.mount(dom)
51
52     return {
53         close: () => {
54             app.unmount();
55             dom.remove();
56         }
57     }
58 };

```



天下无鱼

<https://shikey.com/>

我们来分析上述 JSX 语法实现的对话框组件代码，核心思路是这样子的：

- 提供一个方法直接调用对话框渲染；
- 触发方法时候，在页面<body>标签上创建一个动态<div>标签；
- 用 JSX 生成对话框组件，挂载在这个动态<div>标签上，对话框显示；
- 调用方法返回一个对象，内置一个方法属性提供对话框的关闭操作。

使用时就是按照简单的方法使用，如下代码所示：

```
1 import { createDialog } from './dialog';
2
3 // ...
4 const dialog = createDialog({
5   text: `温馨提示，这是第${showCount.value}次打开对话框`,
6   onOk: () => {
7     dialog.close();
8   }
9 });
```



如果要同时显示多个对话框，就直接执行多次调用，代码如下所示：

```
1 import { createDialog } from './dialog';
2
3 // ...
4 const dialog1 = createDialog({
5   text: `温馨提示，这是第1个对话框`,
6   onOk: () => {
7     dialog1.close();
8   }
9 });
10
11 const dialog2 = createDialog({
12   text: `温馨提示，这是第2个对话框`,
13   onOk: () => {
14     dialog2.close();
15   }
16 });
```

你看这里的代码，是不是比起模板写法维护起来简单得多呢？只需要用单纯的方法调用来触发对话框就行了，不需要像模板语法那样，对每个对话框维护一个变量和标签。

看到这里，我们再来回顾一下刚刚提到的场景。“动态组件”场景下，相比模板语法，JSX 有更加灵活的功能实现和后续代码维护。**但是这个代码的开发和维护的难度并不是绝对的，而是相对的。**

为什么说是相对的呢？其实这个“难度相对”是针对人来说的，而不是技术本身。

因为不同开发者对两种语法的驾驭程度和理解程度不一样，虽然 JSX 语法比较灵活，但是要驾驭好，需要你有比较好的 JavaScript 设计思维。而模板语法虽然没有 JSX 语法那么灵活，

但是它学习成本比较低，同时官方也有大量的模板语法的案例。

那么，现在引申出了一个问题，既然两种语法各有优点，同时它们开发和项目维护难度也是因人而异的，那么我们在企业的项目中如何选择这两种语法呢？

关于怎么选择，我这里就一个观点，用一句网络话语来讲就是“小朋友才做选择，大人们全都要”。

其实两种语法不是互斥的，而是可以共存互相使用的，所以在基于 Vue.js 3 开发的项目里，我们可以这么选择开发语法：

- 普通功能开发以模板语法为主，方便照顾到团队里不同技术能力程度的组员，让项目技术实现沟通起来方便些；
- 模板语法比较难实现的功能就换成 JSX 语法实现，例如一些对话框等动态组件场景，主要为了功能灵活实现和后续代码维护。

另外，你可能还会有疑问，官方推荐的开发语法就是模板语法，那如果我们要学习 JSX 语法有什么渠道呢？我的答案是多去借鉴一些使用 JSX 语法的成熟 Vue.js 3 开源项目，例如，[Ant Design Vue](#)、[Vant UI](#)等。

这些开源项目都是比较流行的 Vue.js 3 UI 组件库，基本能覆盖大部分的企业项目前端开发场景。如果你遇到了某些场景想用 JSX 语法开发，可以去参考对应的组件的 JSX 语法设计。

总结

我们这节课主要介绍和对比了两种 Vue.js 3 的开发语法，模板语法和 JSX 语法。你从中可以理解到两种语法的差异和适用场景。

在面对普通功能开发中，我们可以选择模板语法进行开发，是基于模板语法的简易学习成本，方便团队组员的项目协同合作，在面对一些动态功能开发（例如对话框等动态渲染场景），可以选择 JSX 语法进行开发，让代码更灵活扩展和维护迭代。

但我不希望你只仅仅看到两种语法的使用场景，我希望你能从中理解到 Vue.js 3 开发语法的选择不是绝对根据语法的优缺点，而是要考虑到团队人员对技术的驾驭程度，如果团队成员是

React.js 转 Vue.js，那么估计对 JSX 语法比较熟悉，强行统一用模板语法开发 Vue.js 3 项目估计不是一个最好的选择。



这也引申出一个概念，技术没有绝对的适用场景。在实际团队的项目开发中，要选择某种技术或者某种技术模式，不仅仅要考虑技术优缺点，还要考虑人员的能力程度，综合考虑选择出高效率的技术方案。

思考

前端开发组件经常会遇到组件的“递归使用”，也就是组件内部也循环使用了组件自己，那么，如何用模板语法和 JSX 语法处理组件的“自我递归使用”呢？

[完整的代码在这里](#)

分享给需要的人，Ta购买本课程，你将得 18 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 从Rollup到Vite：如何用Vite构建你的Vue 3项目？

下一篇 05 | 响应式开发操作：如何理解和使用Vue 3的响应式数据？

精选留言 (4)

写留言



飞雪

2022-12-02 来自陕西

想请教一下，我不太明白在用JSX写转换顺序，render函数中-----转换顺序: `{ ${isReverse} }`---
--- 为什么不直接写作 `{ isReverse }`，而是用模板字符串包裹呢？



雪梨.....崔

2022-12-02 来自北京

JSX 不能配置 `scoped` 避免 CSS 样式干扰，那么 JSX 应该如何做才能避免样式干扰呢

共 1 条评论 >



天下无鱼

<https://shikey.com/>



都市夜归人

2022-11-30 来自江苏

用模板方法一样能通过调整数组的顺序去实现，而不是用 `v-if`。这是为了使用 JSX 而这样做的吧，个人觉得例子举的不是很恰当。

共 1 条评论 >



请叫我潜水员

2022-11-29 来自广东

`dialog` 那个例子用模板是一样的效果，`Dialog` 组件从 `dialog.vue` 中引入，然后创建一个相同的 `createDialog`，使用效果一模一样。因为我就是这么用的

共 1 条评论 >

