

# Vue3组件化开发（二）

王红元 coderwhy

# 非父子组件的通信

- 在开发中，我们构建了组件树之后，除了**父子组件之间的通信**之外，还会有**非父子组件之间的通信**。
- 这里我们主要讲两种方式：
  - Provide/Inject ；
  - Mitt全局事件总线 ；

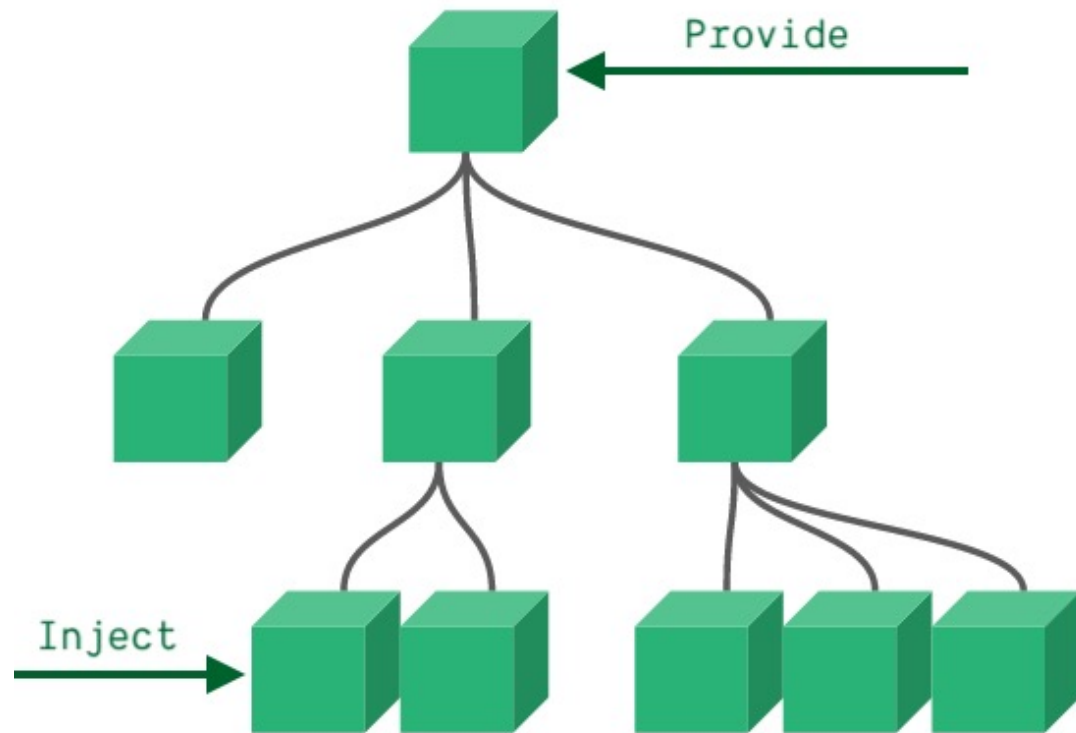
# Provide和Inject

## ■ Provide/Inject用于非父子组件之间共享数据：

- 比如有一些深度嵌套的组件，子组件想要获取父组件的部分内容；
- 在这种情况下，如果我们仍然将props沿着组件链逐级传递下去，就会非常的麻烦；

## ■ 对于这种情况下，我们可以使用 Provide 和 Inject：

- 无论层级结构有多深，父组件都可以作为其所有子组件的依赖提供者；
  - 父组件有一个 **provide** 选项来提供数据；
  - 子组件有一个 **inject** 选项来开始使用这些数据；
- ## ■ 实际上，你可以将依赖注入看作是 “long range props”，除了：
- 父组件不需要知道哪些子组件使用它 provide 的 property
  - 子组件不需要知道 inject 的 property 来自哪里



# Provide和Inject基本使用

■ 我们开发一个这样的结构：



```
App.vue
1 <template>
2   <div>
3     <home></home>
4   </div>
5 </template>
6
7 <script>
8   import Home from './Home.vue';
9
10  export default {
11    components: {
12      Home
13    },
14    provide: {
15      name: "why",
16      age: 18
17    }
18  }
19 </script>
```

```
HomeContent.vue
1 <template>
2   <div>
3     <h2>HomeContent</h2>
4     <h2>{{name}}-{{age}}</h2>
5   </div>
6 </template>
7
8 <script>
9   export default {
10     inject: ["name", "age"]
11   }
12 </script>
13
14 <style scoped>
15
16 </style>
```

# Provide和Inject函数的写法

■ 如果Provide中提供的一些数据是来自data，那么我们可能会想要通过this来获取：

■ 这个时候会报错：

□ 这里给大家留一个思考题，我们的this使用的是哪里的this？

```
<script>
import Home from './Home.vue';

export default {
  components: {
    Home
  },
  data() {
    return {
      names: ["abc", "cba"]
    },
    provide: {
      name: "why",
      age: 18,
      length: this.names.length
    }
  }
}
```

```
Uncaught TypeError: Cannot read property 'names' of undefined
    at eval (App.vue?af3e:22)
    at Module../node_modules/cache-loader/dist/cjs.js?!./node_modules/babel-loader/lib/index.js!./
```

```
provide() {
  return {
    name: "why",
    age: 18,
    length: this.names.length
  }
}
```

# 处理响应式数据

- 我们先来验证一个结果：如果我们修改了`this.names`的内容，那么使用`length`的子组件会不会是响应式的？
- 我们会发现对应的子组件中是**没有反应的**：
  - 这是因为当我们**修改了names之后**，之前在provide中引入的 `this.names.length` 本身并不是响应式的；
- 那么怎么样可以让我们的数据变成响应式的呢？
  - 非常的简单，我们可以使用**响应式的一些API**来完成这些功能，比如说**computed函数**；
  - 当然，这个computed是**vue3的新特性**，在后面我会专门讲解，这里大家可以先直接使用一下；
- **注意：我们在使用length的时候需要获取其中的value**
  - 这是因为**computed返回的是一个ref对象**，需要取出其中的**value来使用**；

```
provide() {  
  return {  
    name: "why",  
    age: 18,  
    length: computed(() => this.names.length)  
  }  
},  
  
<div>  
  <h2>HomeContent</h2>  
  <h2>{{name}}-{{age}}-{{length.value}}</h2>  
</div>
```

# 全局事件总线mitt库

■ Vue3从实例中移除了 \$on、\$off 和 \$once 方法，所以我们如果希望**继续使用全局事件总线**，要通过**第三方的库**：

□ Vue3官方有推荐一些库，例如 [mitt](#) 或 [tiny-emitter](#)；

□ 这里我们主要讲解一下**mitt库**的使用；

■ 首先，我们需要先安装这个库：

```
npm install mitt
```

■ 其次，我们可以封装一个工具eventbus.js：

```
import mitt from 'mitt';

const emitter = mitt();

export default emitter;
```

# 使用事件总线工具

## ■ 在项目中使用它们：

- 我们在Home.vue中监听事件；
- 我们在App.vue中触发事件；

```
<script>
import emitter from './eventBus';

export default {
  // Home.vue中监听
  created() {
    emitter.on("why", (info) => {
      console.log("why event:", info);
    });

    emitter.on("kobe", (info) => {
      console.log("kobe event:", info);
    });

    emitter.on("*", (type, e) => {
      console.log("* event:", type, e);
    });
  }
}
</script>
```

```
import emitter from './eventBus';

export default {
  components: {
    Home
  },
  methods: {
    triggerEvent() {
      emitter.emit("why", {name: "why", age: 18});
    }
  }
}
```



# Mitt的事件取消

- 在某些情况下我们可能希望取消掉之前注册的函数监听：

```
// 取消emitter中所有的监听
emitter.all.clear()

// 定义一个函数
function onFoo() {}
emitter.on('foo', onFoo) // 监听
emitter.off('foo', onFoo) // 取消监听
```

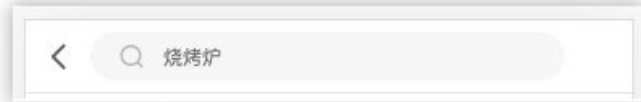
# 认识插槽Slot

## ■ 在开发中，我们会经常封装一个个可复用的组件：

- 前面我们会通过props传递给组件一些数据，让组件来进行展示；
- 但是为了让这个组件具备更强的通用性，我们不能将组件中的内容限制为固定的div、span等等这些元素；
- 比如某种情况下我们使用组件，希望组件显示的是一个按钮，某种情况下我们使用组件希望显示的是一张图片；
- 我们应该让使用者可以决定某一块区域到底存放什么内容和元素；

## ■ 举个栗子：假如我们定制一个通用的导航组件 - NavBar

- 这个组件分成三块区域：左边-中间-右边，每块区域的内容是不固定；
- 左边区域可能显示一个菜单图标，也可能显示一个返回按钮，可能什么都不显示；
- 中间区域可能显示一个搜索框，也可能是一个列表，也可能是一个标题，等等；
- 右边可能是一个文字，也可能是一个图标，也可能什么都不显示；



# 如何使用插槽slot？

## ■ 这个时候我们就可以来定义插槽slot：

- 插槽的使用过程其实是抽取共性、预留不同；
- 我们会将共同的元素、内容依然在组件内进行封装；
- 同时会将不同的元素使用slot作为占位，让外部决定到底显示什么样的元素；

## ■ 如何使用slot呢？

- Vue中将 `<slot>` 元素作为承载分发内容的出口；
- 在封装组件中，使用特殊的元素`<slot>`就可以为封装组件开启一个插槽；
- 该插槽插入什么内容取决于父组件如何使用；

# 插槽的基本使用

- 我们一个组件MySlotCpn.vue：该组件中有一个插槽，我们可以在插槽中放入需要显示的内容；
- 我们在App.vue中使用它们：我们可以插入普通的内容、html元素、组件元素，都可以是可以的；

```
MySlotCpn.vue U x
src > 07_插槽的使用 > MySlotCpn.vue > {} "MySlotCpn.vue" >
1 <template>
2   <div>
3     <h2>MySlotCpn开始</h2>
4     <slot></slot>
5     <h2>MySlotCpn结尾</h2>
6   </div>
7 </template>
```

```
App.vue U x
src > 07_插槽的使用 > App.vue > {} "App.vue" > script
1 <template>
2   <div>
3     <my-slot-cpn>
4       <!-- 1. 普通的内容 -->
5       Hello World
6       <!-- 2. html元素 -->
7       <button>我是按钮</button>
8       <!-- 3. 组件元素 -->
9       <my-button></my-button>
10    </my-slot-cpn>
11  </div>
12 </template>
```

# 插槽的默认内容

- 有时候我们希望在使用插槽时，如果没有插入对应的内容，那么我们需要显示一个**默认的内容**：
  - 当然这个默认的内容只会在没有提供插入的内容时，才会显示；

```
MySlotCpn.vue
<template>
  <div>
    <h2>MySlotCpn开始</h2>
    <slot>
      <h2>我是默认显示的内容</h2>
    </slot>
    <h2>MySlotCpn结尾</h2>
  </div>
</template>
```

默认内容会显示

```
App.vue
1 <template>
2   <div>
3     <my-slot-cpn>
4     </my-slot-cpn>
5   </div>
6 </template>
```

没有插入内容

# 多个插槽的效果

■ 我们先测试一个知识点：如果一个组件中含有**多个插槽**，我们插入多个内容时是什么效果？

□ 我们会发现默认情况下每个插槽都会获取到我们插入的内容来显示；

The image shows a code editor with two files: `App.vue` and `NavBar.vue`.

**NavBar.vue** (Left Panel):

```
1 <template>
2   <div class="nav-bar">
3     <div class="left">
4       <slot></slot>
5     </div>
6     <div class="center">
7       <slot></slot>
8     </div>
9     <div class="right">
10      <slot></slot>
11    </div>
12  </div>
13 </template>
14
15 <script>
16   export default {
```

**App.vue** (Right Panel):

```
1 <template>
2   <div>
3     <nav-bar>
4       <button>左边按钮</button>
5       <h2>中间标题</h2>
6       <i>右边i元素</i>
7     </nav-bar>
8   </div>
9 </template>
```

Arrows indicate the mapping of content from `App.vue` to the slots in `NavBar.vue`:

- Arrow 1: From `<button>左边按钮</button>` to the first `<slot>` in `NavBar.vue`.
- Arrow 2: From `<h2>中间标题</h2>` to the second `<slot>` in `NavBar.vue`.
- Arrow 3: From `<i>右边i元素</i>` to the third `<slot>` in `NavBar.vue`.

**UI Preview (Bottom):**

左边按钮	左边按钮	左边按钮
中间标题	中间标题	中间标题
右边i元素	右边i元素	右边i元素



# 具名插槽的使用

- 事实上，我们希望达到的效果是插槽对应的显示，这个时候我们就可以使用 **具名插槽**：
  - 具名插槽顾名思义就是给插槽起一个名字，`<slot>` 元素有一个特殊的 attribute : `name`；
  - 一个不带 `name` 的 slot，会带有隐含的名字 `default`；

```
src > 07_插槽的使用 > NavBar.vue > {} "NavBar.vue" > script
1 <template>
2   <div class="nav-bar">
3     <div class="left">
4       <slot name="left"></slot>
5     </div>
6     <div class="center">
7       <slot name="center"></slot>
8     </div>
9     <div class="right">
10      <slot name="right"></slot>
11    </div>
12  </div>
13 </template>
14
15 <script>
16   export default {
    made by coderwhy
  }
</script>
```


```
src > 07_插槽的使用 > App.vue > {} "App.vue" > template
1 <template>
2   <div>
3     <nav-bar>
4       <template v-slot:left>
5         <button>左边按钮</button>
6       </template>
7       <template v-slot:center>
8         <h2>中间标题</h2>
9       </template>
10      <template v-slot:right>
11        <i>右边i元素</i>
12      </template>
13    </nav-bar>
14  </div>
15 </template>
```

左边按钮 中间标题 右边元素

## ■ 什么是动态插槽名呢？

- 目前我们使用的插槽名称都是固定的；
- 比如 v-slot:left、v-slot:center等等；
- 我们可以通过 **v-slot:[dynamicSlotName]** 方式动态绑定一个名称；

```
<template>
  <div>
    <nav-bar>
      <template v-slot:[name]>
        <button>左边按钮</button>
      </template>
      <template v-slot:center>
        <h2>中间标题</h2>
      </template>
      <template v-slot:right>
        <i>右边i元素</i>
      </template>
    </nav-bar>
  </div>
</template>
```



```
data() {
  return {
    name: "left"
  }
}
```



# 具名插槽使用的时候缩写

## ■ 具名插槽使用的时候缩写：

- 跟 v-on 和 v-bind 一样，v-slot 也有缩写；
- 即把参数之前的所有内容 (v-slot:) 替换为字符 # ；

```
<template>
  <div>
    <nav-bar>
      <template #left>
        <button>左边按钮</button>
      </template>
      <template #center>
        <h2>中间标题</h2>
      </template>
      <template #right>
        <i>右边i元素</i>
      </template>
    </nav-bar>
  </div>
</template>
```

## ■ 在Vue中有渲染作用域的概念：

- 父级模板里的所有内容都是在父级作用域中编译的；
- 子模板里的所有内容都是在子作用域中编译的；

## ■ 如何理解这句话呢？我们来看一个案例：

- 在我们的案例中ChildCpn自然是可以让问自己作用域中的title内容的；
- 但是在App中，是访问不了ChildCpn中的内容的，因为它们是跨作用域的访问；

## ■ 案例见下页

# 渲染作用域案例

ChildCpn.vue — 03\_learn\_component

App.vue U ChildCpn.vue U ×

src > 08\_作用域插槽 > ChildCpn.vue > {} "ChildCpn.vue" > script > default

```
1 <template>
2   <div>
3     <h2>{{title}}</h2>
4     <slot></slot>
5   </div>
6 </template>
7
8 <script>
9   export default {
10     data() {
11       return {
12         title: "Hello Child Cpn"
13       }
14     }
15   }
16 </script>
```

h2元素中可以访问title

made by coderwhy

App.vue U × ChildCpn.vue U

src > 08\_作用域插槽 > App.vue > {} "App.vue" > template

```
1 <template>
2   <div>
3     <child-cpn>
4       <span>{{title}}</span>
5     </child-cpn>
6   </div>
7 </template>
```

插槽span不可以访问title  
只能访问App作用域中的内容

# 认识作用域插槽

- 但是有时候我们希望插槽**可以访问到子组件中的内容**是非常重要的：
  - 当一个组件被用来渲染一个**数组元素**时，我们**使用插槽**，并且**希望插槽中没有显示每项的内容**；
  - 这个Vue给我们提供了**作用域插槽**；
- 我们来看下面的一个案例：
  - 1.在App.vue中定义好数据
  - 2.传递给ShowNames组件中
  - 3.ShowNames组件中遍历names数据
  - 4.定义插槽的prop
  - 5.通过v-slot:default的方式获取到slot的props
  - 6.使用slotProps中的item和index

# 作用域插槽的案例

The image shows a code editor with two files: `App.vue` and `ShowNames.vue`. The editor is displaying the `src > 08_作用域插槽 > App.vue` file on the left and the `src > 08_作用域插槽 > ShowNames.vue` file on the right. The code is annotated with orange boxes and arrows, explaining the steps for using scoped slots.

**App.vue**

```
1 <template>
2   <div>
3     <show-names :names="names">
4       <template v-slot:default="slotProps">
5         <span>{{slotProps.item}}-{{slotProps.index}}</span>
6       </template>
7     </show-names>
8   </div>
9 </template>
10
11 <script>
12   import ShowNames from './ShowNames.vue';
13
14   export default {
15     components: {
16       ShowNames,
17     },
18     data() {
19       return {
20         names: ["why", "kobe", "james", "curry"]
21       }
22     }
13
```

Annotations for `App.vue`:

- 1. 在App.vue中定义好数据 (Define data in App.vue)
- 5. 通过v-slot:default的方式获取到slot的props (Get slot props via v-slot:default)
- 6. 使用其中的item和index属性 (Use item and index attributes)

**ShowNames.vue**

```
1 <template>
2   <div>
3     <template v-for="(item, index) in names" :key="item">
4       <!-- 插槽prop -->
5       <slot :item="item" :index="index"></slot>
6     </template>
7   </div>
8 </template>
9
10 <script>
11   export default {
12     props: {
13       names: {
14         type: Array,
15         default: () => []
16       }
17     }
18   }
19 </script>
20
```

Annotations for `ShowNames.vue`:

- 2. 传递到了ShowNames组件中 (Passed to ShowNames component)
- 3. ShowNames组件中遍历names数据 (Iterate names data in ShowNames component)
- 4. 定义插槽prop (Define slot prop)

# 独占默认插槽的缩写

- 如果我们的插槽是默认插槽default，那么在使用的时候 `v-slot:default="slotProps"` 可以简写为 `v-slot="slotProps"`：

```
<show-names :names="names">
  <template v-slot="slotProps">
    <span>{{slotProps.item}}-{{slotProps.index}}</span>
  </template>
</show-names>
```

- 并且如果我们的插槽只有默认插槽时，组件的标签可以被当做插槽的模板来使用，这样，我们就可以将 `v-slot` 直接用在组件上：

```
<show-names :names="names" v-slot="slotProps">
  <span>{{slotProps.item}}-{{slotProps.index}}</span>
</show-names>
```



# 默认插槽和具名插槽混合

- 但是，如果有默认插槽和具名插槽，那么按照完整的template来编写。

```
<template>
  <div>
    <show-names :names="names" v-slot="slotProps">
      <span>{{slotProps.item}}-
    </show-names>
    <template v-slot:why>
      <h2>哈哈</h2>
    </template>
  </div>
</template>
```

[vue/valid-v-slot] Default slot must use '<template>' on a custom element when there are other named slots. eslint-plugin-vue

[View Problem \(⌘F8\)](#) No quick fixes available

- 只要出现多个插槽，请始终为所有的插槽使用完整的基于 <template> 的语法：

```
<show-names :names="names">
  <template v-slot="slotProps">
    <span>{{slotProps.item}}-{{slotProps.index}}</span>
  </template>
  <template v-slot:why>
    <h2>哈哈</h2>
  </template>
</show-names>
```