目录 Contents

动态组件

1. 什么是动态组件

动态组件指的是动态切换组件的显示与隐藏

动态组件

2. 如何实现动态组件渲染

vue 提供了一个内置的 <component> 组件，专门用来实现动态组件的渲染。示例代码如下：

```
data() {
  // 1. 当前要渲染的组件名称
  return { comName: 'Left' }
}

<!-- 2. 通过 is 属性，动态指定要渲染的组件 -->
<component :is="comName"></component>

<!-- 3. 点击按钮，动态切换组件的名称 -->
<button @click="comName = 'Left'">展示 Left 组件</button>
<button @click="comName = 'Right'">展示 Right 组件</button>
```

动态组件

3. 使用 keep-alive 保持状态

默认情况下，切换动态组件时无法保持组件的状态。此时可以使用 vue 内置的 <keep-alive> 组件保持动态组件的状态。示例代码如下：

```
<keep-alive>
  <component :is="comName"></component>
</keep-alive>
```

动态组件

4. keep-alive 对应的生命周期函数

当组件被缓存时，会自动触发组件的 deactivated 生命周期函数。

当组件被激活时，会自动触发组件的 activated 生命周期函数。

```
export default {
  created() { console.log('组件被创建了') },
  destroyed() { console.log('组件被销毁了') },

  activated() { console.log('Left 组件被激活了！') },
  deactivated() { console.log('Left 组件被缓存了！') }
}
```

动态组件

5. keep-alive 的 include 属性

include 属性用来指定：只有名称匹配的组件会被缓存。多个组件名之间使用英文的逗号分隔：

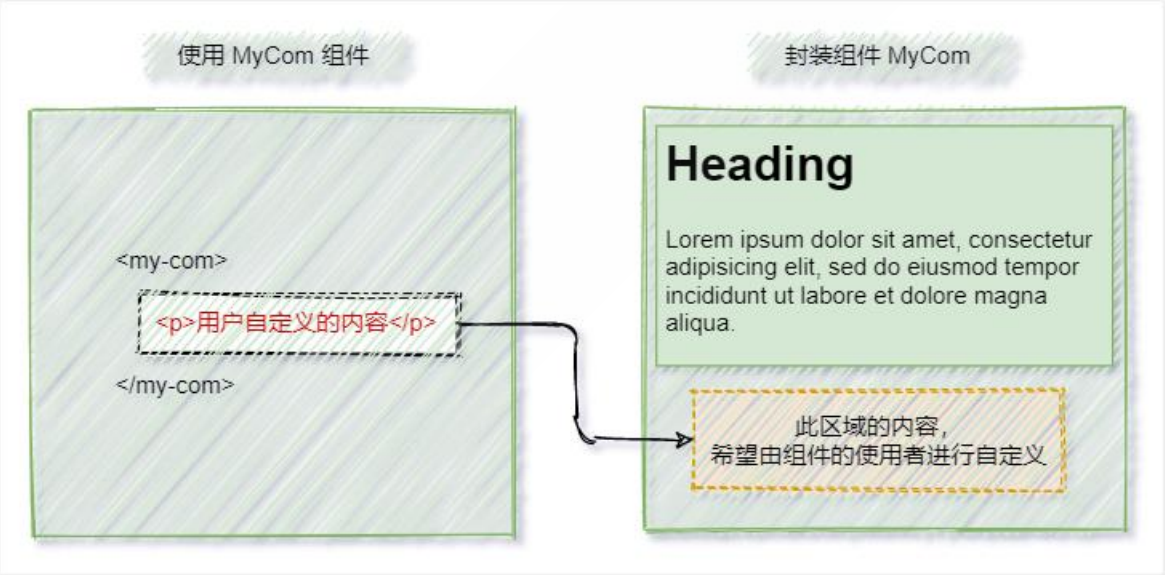
```
1 <keep-alive include="MyLeft,MyRight">
2   <component :is="comName"></component>
3 </keep-alive>
```

目录 Contents

插槽

1. 什么是插槽

插槽（Slot）是 vue 为组件的封装者提供的能力。允许开发者在封装组件时，把不确定的、希望由用户指定的部分定义为插槽。



可以把插槽认为是组件封装期间，为用户预留的内容的占位符。

插槽

2. 体验插槽的基础用法

在封装组件时，可以通过 <slot> 元素定义插槽，从而为用户预留内容占位符。示例代码如下：

```
<template>
  <p>这是 MyCom1 组件的第 1 个 p 标签</p>
  <!-- 通过 slot 标签，为用户预留内容占位符（插槽） -->
  <slot></slot>
  <p>这是 MyCom1 组件最后一个 p 标签</p>
</template>
```

```
<my-com-1>
  <!-- 在使用 MyCom1 组件时，为插槽指定具体的内容 -->
  <p>~~~用户自定义的内容~~~</p>
</my-com-1>
```

插槽

2.1 没有预留插槽的内容会被丢弃

如果在封装组件时没有预留任何 <slot> 插槽，则用户提供的任何自定义内容都会被丢弃。示例代码如下：

```
<template>
  <p>这是 MyCom1 组件的第 1 个 p 标签</p>
  <!-- 封装组件时吗，没有预留任何插槽 -->
  <p>这是 MyCom1 组件最后一个 p 标签</p>
</template>
```

```
<my-com-1>
  <!-- 自定义的内容会被丢弃 -->
  <p>~~~用户自定义的内容~~~</p>
</my-com-1>
```

插槽

2.2 后备内容

封装组件时，可以为预留的 <slot> 插槽提供后备内容（默认内容）。如果组件的使用者没有为插槽提供任何内容，则后备内容会生效。示例代码如下：

```
<template>
  <p>这是 MyCom1 组件的第 1 个 p 标签</p>
  <slot>这是后备内容</slot>
  <p>这是 MyCom1 组件最后一个 p 标签</p>
</template>
```

插槽

3. 具名插槽

如果在封装组件时需要预留多个插槽节点，则需要为每个 <slot> 插槽指定具体的 name 名称。这种带有具体名称的插槽叫做“具名插槽”。示例代码如下：

```
<div class="container">
  <header>
    <!-- 我们希望把页头放这里 -->
    <slot name="header"></slot>
  </header>
  <main>
    <!-- 我们希望把主要内容放这里 -->
    <slot></slot>
  </main>
  <footer>
    <!-- 我们希望把页脚放这里 -->
    <slot name="footer"></slot>
  </footer>
</div>
```

注意：没有指定 name 名称的插槽，会有隐含的名称叫做 “default”。

插槽

3.1 为具名插槽提供内容

在向具名插槽提供内容的时候，我们可以在一个 <template> 元素上使用 v-slot 指令，并以 v-slot 的参数的形式提供其名称。示例代码如下：

```
<my-com-2>
  <template v-slot:header>
    <h1>滕王阁序</h1>
  </template>

  <template v-slot:default>
    <p>豫章故郡，洪都新府。</p>
    <p>星分翼轸，地接衡庐。</p>
    <p>襟三江而带五湖，控蛮荆而引瓯越。</p>
  </template>

  <template v-slot:footer>
    <p>落款：王勃</p>
  </template>
</my-com-2>
```

插槽

3.2 具名插槽的简写形式

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 #。例如 v-slot:header 可以被重写为 #header：

```
<my-com-2>
  <template #header>
    <h1>滕王阁序</h1>
  </template>

  <template #default>
    <p>豫章故郡，洪都新府。</p>
    <p>星分翼轸，地接衡庐。</p>
    <p>襟三江而带五湖，控蛮荆而引瓯越。</p>
  </template>

  <template #footer>
    <p>落款：王勃</p>
  </template>
</my-com-2>
```

插槽

4. 作用域插槽

在封装组件的过程中，可以为预留的 <slot> 插槽绑定 props 数据，这种带有 props 数据的 <slot> 叫做“作用域插槽”。示例代码如下：

```
<tbody>
  <!-- 下面的 slot 是一个作用域插槽 -->
  <slot v-for="item in list" :user="item"></slot>
</tbody>
```

插槽

4.1 使用作用域插槽

可以使用 v-slot: 的形式，接收作用域插槽对外提供的数据。示例代码如下：

```
<my-com-3>
  <!-- 1. 接收作用域插槽对外提供的数据 -->
  <template v-slot:default="scope">
    <tr>
      <!-- 2. 使用作用域插槽的数据 -->
      <td>{{scope}}</td>
    </tr>
  </template>
</my-com-3>
```

插槽

4.2 解构插槽 Prop

作用域插槽对外提供的数据对象，可以使用解构赋值简化数据的接收过程。示例代码如下：

```
<my-com-3>
  <!-- v-slot: 可以简写成 # -->
  <!-- 作用域插槽对外提供的数据对象，可以通过“解构赋值”简化接收的过程 -->
  <template #default="{user}">
    <tr>
      <td>{{user.id}}</td>
      <td>{{user.name}}</td>
      <td>{{user.state}}</td>
    </tr>
  </template>
</my-com-3>
```

目录

Contents

自定义指令

1. 什么是自定义指令

vue 官方提供了 v-text、v-for、v-model、v-if 等常用的指令。除此之外 vue 还允许开发者自定义指令。

自定义指令

2. 自定义指令的分类

vue 中的自定义指令分为两类，分别是：

- 私有自定义指令
- 全局自定义指令

自定义指令

3. 私有自定义指令

在每个 vue 组件中，可以在 directives 节点下声明私有自定义指令。示例代码如下：

```
directives: {
  color: {
    // 为绑定到的 HTML 元素设置红色的文字
    bind(el) {
      // 形参中的 el 是绑定了此指令的、原生的 DOM 对象
      el.style.color = 'red'
    }
  }
}
```

自定义指令

4. 使用自定义指令

在使用自定义指令时，需要加上 v- 前缀。示例代码如下：

```
<!-- 声明自定义指令时，指令的名字是 color -->
<!-- 使用自定义指令时，需要加上 v- 指令前缀 -->
<h1 v-color>App 组件</h1>
```

自定义指令

5. 为自定义指令动态绑定参数值

在 template 结构中使用自定义指令时，可以通过等号（=）的方式，为当前指令动态绑定参数值：

```
data() {
  return {
    color: 'red' // 定义 color 颜色值
  }
}

<!-- 在使用指令时，动态为当前指令绑定参数值 color -->
<h1 v-color="color">App 组件</h1>
```

自定义指令

6. 通过 binding 获取指令的参数值

在声明自定义指令时，可以通过形参中的第二个参数，来接收指令的参数值：

```
directives: {
  color: {
    bind(el, binding) {
      // 通过 binding 对象的 .value 属性，获取动态的参数值
      el.style.color = binding.value
    }
  }
}
```

自定义指令

7. update 函数

bind 函数只调用 1 次：当指令第一次绑定到元素时调用，当 DOM 更新时 bind 函数不会被触发。 update 函数会在每次 DOM 更新时被调用。示例代码如下：

```
directives: {
  color: {
    // 当指令第一次被绑定到元素时被调用
    bind(el, binding) {
      el.style.color = binding.value
    },
    // 每次 DOM 更新时被调用
    update(el, binding) {
      el.style.color = binding.value
    }
  }
}
```

自定义指令

8. 函数简写

如果 insert 和update 函数中的逻辑完全相同，则对象格式的自定义指令可以简写成函数格式：

```
directives: {
  // 在 insert 和 update 时，会触发相同的业务逻辑
  color(el, binding) {
    el.style.color = binding.value
  }
}
```

自定义指令

9. 全局自定义指令

全局共享的自定义指令需要通过“Vue.directive()”进行声明，示例代码如下：

```
// 参数1：字符串，表示全局自定义指令的名字
// 参数2：对象，用来接收指令的参数值
Vue.directive('color', function(el, binding) {
  el.style.color = binding.value
})
```

① 能够掌握 keep-alive 元素的基本使用

- <keep-alive> 标签、include 属性

② 能够掌握插槽的基本用

- <slot> 标签、具名插槽、作用域插槽、后备内容

③ 能够知道如何自定义指令

- 私有自定义指令 directives: { }
- 全局自定义指令 Vue.directive()