

5. 模板语法与组件通信



Vue 的模板语法是其声明式渲染的核心，而组件通信则是构建可扩展应用的基础。这两者都是 Vue 面试中的绝对高频考点。

1. 模板语法深入解析

Vue 的指令（Directives）是带有 `v-` 前缀的特殊属性，用于在渲染的 DOM 上应用响应式行为。

`v-bind` (缩写 `:`) - 动态属性绑定

用于动态地绑定一个或多个 HTML 属性。

代码块

```
1  <script setup>
2  import { ref, reactive } from 'vue';
3
4  const imageUrl = ref('logo.png');
5  const imageAlt = ref('Company Logo');
6  const isDisabled = ref(true);
7
8  // 使用对象一次性绑定多个属性
9  const elementAttributes = reactive({
10    id: 'unique-element',
11    'data-index': 123
12  });
13 </script>
14
15 <template>
16   <!-- 基础用法 -->
17   
18
19   <!-- 绑定布尔值属性 -->
20   <button :disabled="isDisabled">Click Me</button>
21
22   <!-- 绑定一个包含多个属性的对象 -->
23   <div v-bind="elementAttributes"></div>
24 </template>
```

v-model - 双向数据绑定

`v-model` 是一个语法糖，用于在表单输入元素或自定义组件上创建双向数据绑定。它会根据元素类型自动选择正确的方式来更新值。

核心原理: `:value` + `@input` 事件的结合。

修饰符:

- `.lazy`: 将 `input` 事件同步改为 `change` 事件同步。
- `.number`: 将输入值自动转为数字。
- `.trim`: 自动过滤用户输入的首尾空白字符。

代码块

```
1  <script setup>
2  import { ref } from 'vue';
3
4  const message = ref('');
5  const age = ref(0);
6  const username = ref('');
7  </script>
8
9  <template>
10   <!-- 基础用法 -->
11   <input v-model="message" placeholder="输入内容..." />
12   <p>Message is: {{ message }}</p>
13
14   <!-- .number 修饰符 -->
15   <input v-model.number="age" type="number" />
16
17   <!-- .trim 修饰符 -->
18   <input v-model.trim="username" />
19 </template>
```

v-for - 列表渲染

用于基于一个数组来渲染一个列表。

- **关键:** 必须使用 `key` 属性为每个节点提供一个唯一的标识，以帮助 Vue 高效地更新 DOM。
- **适用范围:** 可以遍历数组、对象、数字和字符串。

代码块

```
1  <script setup>
2  import { ref } from 'vue';
3
4  const items = ref([
5    { id: 1, name: 'Apple' },
6    { id: 2, name: 'Banana' },
7    { id: 3, name: 'Cherry' }
8  ]);
9  </script>
10
11 <template>
12   <ul>
13     <!-- 遍历数组 -->
14     <li v-for="(item, index) in items" :key="item.id">
15       {{ index }} - {{ item.name }}
16     </li>
17   </ul>
18 </template>
```

`v-if` / `v-show` - 条件渲染

用于根据条件决定是否渲染或显示一个元素。

- `v-if` (真正的条件渲染):
 - 当条件为 `false` 时，元素及其子组件会被**销毁**和**从 DOM 中移除**。
 - 切换开销较高，但初始渲染开销较低（如果条件为 `false`）。
 - 支持与 `v-else` 和 `v-else-if` 配合使用。
- `v-show` (基于 CSS 的切换):
 - 元素始终会被渲染到 DOM 中。
 - 通过切换元素的 CSS `display` 属性来控制其显示和隐藏。
 - 切换开销较低，但初始渲染开销较高。

使用原则:

- **频繁切换**: 使用 `v-show`。
- **运行时条件很少改变**: 使用 `v-if`。

代码块

```
1  <script setup>
```

```
2  import { ref } from 'vue';
3  const isLoggedIn = ref(true);
4  const showDetails = ref(false);
5  </script>
6
7  <template>
8    <!-- v-if 示例 -->
9    <div v-if="isLoggedIn">
10      Welcome back, User!
11    </div>
12    <div v-else>
13      Please log in.
14    </div>
15
16    <!-- v-show 示例 -->
17    <button @click="showDetails = !showDetails">Toggle Details</button>
18    <div v-show="showDetails">
19      This is a detail panel.
20    </div>
21  </template>
```

2. 组件通信模式详解

父子通信: `props` / `emit` / `slots`

这是最常用和最基础的通信方式。

- `props` (父 -> 子): 父组件通过属性将数据向下传递给子组件。
- `emit` (子 -> 父): 子组件通过触发事件将信息发送给父组件。
- `slots` (父 -> 子内容分发): 父组件可以将模板片段（内容）插入到子组件指定的位置。
-

父组件 `ParentComponent.vue`

代码块

```
1  <script setup>
2  import { ref } from 'vue';
3  import UserCard from './UserCard.vue';
4
```

```

5  const user = ref({ id: 1, name: 'Alice' });
6
7  function handleUpdate(newName) {
8      user.value.name = newName;
9      alert(`User name updated to: ${newName}`);
10 }
11 </script>
12
13 <template>
14     <UserCard :user-name="user.name" @name-updated="handleUpdate">
15         <!-- 将内容插入到子组件的默认插槽中 -->
16         <p>This is some extra information about the user.</p>
17     </UserCard>
18 </template>

```

子组件 `UserCard.vue`

代码块

```

1  <script setup>
2  // 1. 定义接收的 props
3  const props = defineProps({
4      userName: {
5          type: String,
6          required: true
7      }
8  });
9
10 // 2. 定义可以触发的事件
11 const emit = defineEmits(['name-updated']);
12
13 function updateUser() {
14     const newName = props.userName + '!';
15     // 3. 触发事件，将数据传递给父组件
16     emit('name-updated', newName);
17 }
18 </script>
19
20 <template>
21     <div class="card">
22         <h3>{{ userName }}</h3>
23         <!-- 4. 渲染父组件传递过来的插槽内容 -->
24         <slot></slot>
25         <button @click="updateUser">Update Name</button>
26     </div>

```

双向绑定：在自定义组件上使用 `v-model`

你可以在自己的组件上实现 `v-model`，这对于创建自定义表单控件尤其有用。

- **约定**: 组件接收一个名为 `modelValue` 的 prop，并通过 `update:modelValue` 事件来更新它。

父组件 `ParentForm.vue`

代码块

```
1  <script setup>
2  import { ref } from 'vue';
3  import CustomInput from './CustomInput.vue';
4
5  const searchText = ref('Initial Text');
6  </script>
7
8  <template>
9    <CustomInput v-model="searchText" />
10    <p>Current search text: {{ searchText }}</p>
11  </template>
```

子组件 `CustomInput.vue`

代码块

```
1  <script setup>
2  defineProps(['modelValue']);
3  const emit = defineEmits(['update:modelValue']);
4
5  function onInput(event) {
6    emit('update:modelValue', event.target.value);
7  }
8  </script>
9
10 <template>
11   <input :value="modelValue" @input="onInput" />
12 </template>
```

跨层级通信：provide / inject

当需要从祖先组件向其所有后代组件传递数据时，使用 `provide` 和 `inject` 可以避免逐层传递 `props`（即“prop drilling”）。

- **provide**：在祖先组件中提供数据或方法。
- **inject**：在任何后代组件中注入（接收）这些数据或方法。

祖先组件 `App.vue`

代码块

```
1  <script setup>
2  import { ref, provide } from 'vue';
3  import DeepChild from './DeepChild.vue';
4
5  const theme = ref('light');
6  function toggleTheme() {
7    theme.value = theme.value === 'light' ? 'dark' : 'light';
8  }
9
10 // 提供数据和方法给所有后代
11 provide('theme', theme);
12 provide('toggleTheme', toggleTheme);
13 </script>
14
15 <template>
16   <div :class="theme">
17     <DeepChild />
18   </div>
19 </template>
```

后代组件 `DeepChild.vue`

代码块

```
1  <script setup>
2  import { inject } from 'vue';
3
4  // 注入来自祖先的数据和方法
5  const theme = inject('theme', 'light'); // 'light' 是默认值
6  const toggleTheme = inject('toggleTheme');
7  </script>
8
```

```
9   <template>
10   <p>Current theme is: {{ theme }}</p>
11   <button @click="toggleTheme">Toggle Theme</button>
12 </template>
```

3. 面试核心问题与最佳实践

Q1: "v-if 和 v-show 有什么区别，应该如何选择?"

- **区别:**
 - **原理:** `v-if` 是真正的条件渲染，它会确保在切换过程中条件块内的事件监听器和子组件被适当地**销毁和重建**。`v-show` 只是简单地切换元素的 **CSS `display` 属性**。
 - **性能:** `v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。
- **选择:**
 - 如果需要**频繁地切换**，使用 `v-show` 性能更好。
 - 如果条件在运行时**很少改变**，或者初始为假时不需要渲染任何东西，使用 `v-if` 更合适。


Q2: "在 Vue 3 中，v-if 和 v-for 一起使用时，哪个优先级更高?"

- 在 Vue 3 中，`v-if` 的优先级高于 `v-for`。这意味着 `v-if` 会先生效，此时它还无法访问到 `v-for` 作用域中的变量。
- **最佳实践**是避免将它们放在同一个元素上。推荐的做法是：
 - 使用一个计算属性（`computed`）来预先过滤掉不需要显示的项，然后在过滤后的列表上使用 `v-for`。
 - 将 `v-for` 移到 `<template>` 标签上，然后在内部的元素上使用 `v-if`。

代码块

```
1  <!-- 推荐：使用计算属性 -->
2  <div v-for="item in visibleItems" :key="item.id">{{ item.name }}</div>
3
4  <!-- 备选：使用 <template> 标签 -->
5  <template v-for="item in items" :key="item.id">
6    <div v-if="item.isVisible">{{ item.name }}</div>
7  </template>
```

Q3: "除了 `props` / `emit`，你还知道哪些组件通信方式？"

- `v-model`: 用于在父子组件间创建双向绑定，非常适合封装表单控件。
 - `provide` / `inject`: 用于跨越多层的祖先到后代的通信，可以有效解决“prop drilling”问题，常用于传递全局配置或主题信息。
 - **状态管理库 (Pinia/Vuex)**: 当多个组件需要共享和操作同一份复杂状态时，应使用集中的状态管理方案，以保证数据流的可预测性和可维护性。
-
-  **最佳实践**
 1. **数据单向流动**: 始终坚持父组件通过 `props` 向下传递数据，子组件通过 `emit` 事件通知父组件进行状态变更。子组件永远不应直接修改 `props`。
 2. **key 的重要性**: 在使用 `v-for` 时，总是提供一个唯一的、稳定的 `key` 值（如 `item.id`），而不是使用 `index`，这对于性能优化和避免状态混淆至关重要。
 3. **选择合适的通信方式**:
 - **父子**: 默认使用 `props` / `emit`。
 - **深层嵌套**: 考虑 `provide` / `inject`。
 - **兄弟或远亲**: 提升状态到共同的父组件，或使用 Pinia。
 4. **组件接口清晰**: 为 `props` 提供明确的类型、默认值和校验规则。为 `emits` 做出明确的声明。这使得组件像一份清晰的 API 文档，易于理解和使用。