**Vue**

Vue 是一款用于构建用户界面的 JavaScript 框架。它基于标准 HTML、CSS 和 JavaScript 构建，并提供了一套声明式的、组件化的编程模型，帮助你高效地开发用户界面。

Vue 的两个核心功能（特点）：

* 声明式渲染：Vue 基于标准 HTML 拓展了一套模板语法，使得我们可以声明式地描述最终输出的 HTML 和 JavaScript 状态之间的关系。
* 响应性：Vue 会自动跟踪 JavaScript 状态并在其发生变化时响应式地更新 DOM。

渐进式框架​

Vue 是一个框架，也是一个生态。其功能覆盖了大部分前端开发常见的需求。但 Web 世界是十分多样化的，不同的开发者在 Web 上构建的东西可能在形式和规模上会有很大的不同。考虑到这一点，Vue 的设计非常注重灵活性和“可以被逐步集成”这个特点。根据你的需求场景，你可以用不同的方式使用 Vue：

* 无需构建步骤，渐进式增强静态的 HTML
* 在任何页面中作为 Web Components 嵌入
* 单页应用 (SPA)
* 全栈 / 服务端渲染 (SSR)
* Jamstack / 静态站点生成 (SSG)
* 开发桌面端、移动端、WebGL，甚至是命令行终端中的界面

Vue 的组件可以按两种不同的风格书写：选项式 API 和组合式 API。

选项式 API (Options API)​

使用选项式 API，我们可以用包含多个选项的对象来描述组件的逻辑，例如 data、methods 和 mounted。选项所定义的属性都会暴露在函数内部的 this 上，它会指向当前的组件实例。

组合式 API (Composition API)​

通过组合式 API，我们可以使用导入的 API 函数来描述组件逻辑。在单文件组件中，组合式 API 通常会与 <script setup> 搭配使用。这个 setup attribute 是一个标识，告诉 Vue 需要在编译时进行一些处理，让我们可以更简洁地使用组合式 API。比如，<script setup> 中的导入和顶层变量/函数都能够在模板中直接使用。

**Vue写法总成**

<template>

<div>

<p>{{ message }}</p>

<button @click="reverseMessage">反转</button>

</div>

</template>

<script setup>

import { ref } from 'vue'

const message = ref('你好 Vue 3')

function reverseMessage() {

message.value = message.value.split('').reverse().join('')

}

</script>

<style scoped>

/\* 样式写这里 \*/

</style>

<template>标签内写如html元素，<script>标签内写js逻辑，并用ref绑定html内的元素，<style>标签内写css样式

**模板语法**

**文本插值**

（将文本内的某值设为随函数改变的变量，即绑定到某个函数），用双大括号：”{{...}}”

1. 文本插值 ({{ rawHtml }})

<p>Using text interpolation: {{ rawHtml }}</p>

这是标准的 Vue 插值语法，用来渲染数据。

效果：如果 rawHtml 包含 HTML 标签，它们将被当作普通文本处理，不会作为 HTML 解析显示。也就是说，如果 rawHtml = "<b>加粗文字</b>"，它会显示为字符串 "<b>加粗文字</b>"，而不会渲染成加粗的文字。

2. 使用 v-html 指令 (<span v-html="rawHtml"></span>)

<p>Using v-html directive: <span v-html="rawHtml"></span></p>

v-html 指令用于渲染原始 HTML 内容。

效果：rawHtml 的值会被解析并插入到 DOM 中，作为真正的 HTML 渲染。例如，如果 rawHtml = "<b>加粗文字</b>"，它将渲染为加粗的文字，而不是显示 <b>加粗文字</b>。

**Attribute 绑定**

双括号可以绑定文本，但无法绑定html标签的属性，即HTML的attribute，此时就要用”v-bind:”的方式来绑定属性，来实现属性值的更新，简写为”:”

例如：

<template>

<div>

<!-- 绑定图片的 src -->

<img v-bind:src="imageSrc" alt="Dynamic Image" />

<button @click="changeImage">Change Image</button>

</div>

</template>

<script setup>

import { ref } from "vue";

// 定义 imageSrc 作为响应式数据

const imageSrc = ref("https://example.com/initial-image.jpg");

// 定义一个函数来改变 imageSrc

function changeImage() {

imageSrc.value = "https://example.com/another-image.jpg";

}

</script>

工作原理：

1. 初始时，imageSrc 的值是 https://example.com/initial-image.jpg，因此 <img> 标签的 src 属性会被绑定到这个地址，显示初始的图片。
2. 当你点击 "Change Image" 按钮时，changeImage 函数被调用，imageSrc 的值被更新为 https://example.com/another-image.jpg。
3. 因为 imageSrc 是响应式的，Vue 会自动检测到数据的变化，并更新 src 属性。于是，图片会切换为新的图片。

**布尔型 Attribute​**

布尔型 attribute 依据 true / false 值来决定 attribute 是否应该存在于该元素上。

即：只要这个属性存在，它就是 true，不存在就是 false。

disabled 就是最常见的例子之一：

* disabled（禁用输入框或按钮）
* checked（勾选状态）
* readonly（只读）
* required（表单必须填写）
* autofocus 等

v-bind 在这种场景下的行为略有不同：

template

<button :disabled="isDisabled">Button</button>

<script setup>

import { ref } from 'vue'

const isDisabled = ref(true) // 或 false

</script>

当 isDisabled 为真值或一个空字符串 (即 <button disabled="">) 时，元素会包含这个 disabled 属性。而当其为其他假值时 attribute 将被忽略。

即：

* 为true时：<button disabled>按钮</button>（禁用）
* 为false时：<button>按钮</button>（不禁用）

**动态绑定多个值​**

如果你有像这样的一个包含多个 attribute 的 JavaScript 对象：

const objectOfAttrs = {

id: 'container',

class: 'wrapper',

style: 'background-color:green'

}

通过不带参数的 v-bind，你可以将它们绑定到单个元素上：

template

<div v-bind="objectOfAttrs"></div>

等价于:

<div id="container" class="wrapper" style="background-color:green"></div>

可以随时更新对象的内容：

objectOfAttrs.class = 'new-class'

objectOfAttrs.style = 'background-color:blue'

页面上的 DOM 会自动响应更新

**在绑定中使用js表达式**

Vue支持在绑定中插入完整的js表达式，比如

{{ number + 1 }} //number属性的值加1

{{ ok ? 'YES' : 'NO' }} //如果ok为ture，输出YES，为false则输出NO

{{ message.split('').reverse().join('') }} //假设message的值为hello

//将message的值反转输出，即olleh

//.split('') ['h', 'e', 'l', 'l', 'o'] 把字符串拆成字符数组

//.reverse() ['o', 'l', 'l', 'e', 'h'] 数组反转

//.join('') 'olleh' 再拼接成字符串

注意：此方式仅支持表达式​

每个绑定仅支持单一表达式，也就是一段能够被求值的 JavaScript 代码。一个简单的判断方法是是否可以合法地写在 return 后面。

因此，下面的例子都是无效的：

<!-- 这是一个语句，而非表达式 -->

{{ var a = 1 }}

<!-- 条件控制也不支持，请使用三元表达式 -->

{{ if (ok) { return message } }}

**v-if绑定元素**

v-if可以控制元素的显示和隐藏

比如：

<p v-if="seen">Now you see me</p>

这里，v-if 指令会基于表达式 seen 的值的真/假来移除/插入该 <p> 元素。

响应式基础

**ref()​**

在组合式 API 中，推荐使用 ref() 函数来声明响应式状态

**<script setup>​**

在 setup() 函数中手动暴露大量的状态和方法非常繁琐。我们可以通过使用单文件组件 (SFC) 来避免这种情况。我们可以使用 <script setup> 来大幅度地简化代码

<script setup>

import { ref } from 'vue'

const count = ref(0)

const increment = () => count.value++

</script>

<template>

<button @click="increment">Count: {{ count }}</button>

</template>

变量和函数 自动暴露到模板中，不需要 return，代码更清爽！

**DOM 更新的时机：“Next Tick”原理**

在 Vue 中，响应式数据发生变化时，DOM 并不会立即更新，而是会等当前同步代码执行完之后，统一在下一个“事件循环 (tick)”中一次性更新所有受影响的 DOM。也就是说，虽然修改了响应式状态时，DOM 会被自动更新，但是需要注意的是，DOM 更新不是同步的。Vue 会在“next tick”更新周期中缓冲所有状态的修改，以确保不管你进行了多少次状态修改，每个组件都只会被更新一次。

**reactive()​**

还有另一种声明响应式状态的方式，即使用 reactive() API。与将内部值包装在特殊对象中的 ref 不同，reactive() 将使对象本身具有响应性

* ref()：适合定义基本类型（数字、字符串、布尔值）或单个值
* reactive()：适合定义“对象或数组”这样的结构，直接让整个对象响应式化

**ref（）示例：**

import { ref } from 'vue'

const count = ref(0)

*// 使用时需要 `.value`*

console.log(count.value) *// 0*

count.value++ *// 响应式更新*

**reactive() 示例：**

import { reactive } from 'vue'

const state = reactive({

count: 0,

name: 'Vue',

})

*// 使用时直接访问*

console.log(state.count) *// 0*

state.count++ *// 自动追踪响应*

注意！！虽然ref() 是设计来包装“基本类型”的，但它其实也可以用来包装数组、对象等复杂类型 —— 因为它的内部机制支持这一点。

Ref也能定义数组，因为本质上是包了一层响应式对象：

import { ref } from 'vue'

const todos = ref([]) //<--这是合法的！

todos.value.push({ text: '学习 Vue', done: false })

为什么这样写是有效的？

ref([]) 实际上返回的是：

{ value: reactive([]) }

所以 ref([]) 实际上是：把这个数组变成了响应式的，然后再包在一个 value 属性里。

**计算属性computed()**

computed() 是用来声明“计算属性”的工具，它的值会根据它依赖的响应式数据自动更新，但只有在依赖变更时才重新计算（有缓存）。

比如：

const todos = ref([

{ text: '学习 Vue', done: false },

{ text: '吃饭', done: true }

])

const unfinishedCount = computed(() => {

return todos.value.filter(todo => !todo.done).length

})

这样在模板中你可以直接使用：

<p>未完成任务数：{{ unfinishedCount }}</p>

注意！！计算属性 computed() 只有在它依赖的响应式数据发生变化时才会重新计算。

当computed()内不是响应式数据时，它就不会被vue的响应式系统追踪。

比如：

const now = computed(() => Date.now())

这个now的计算值只会更新一次。因为date.now只是一个普通的函数调用。

**类（Class） 与 样式（Style) 绑定**

数据绑定的一个常见需求场景是操纵元素的 CSS class 列表和内联样式。因为 class 和 style 都是 attribute，我们可以和其他 attribute 一样使用 v-bind 将它们和动态的字符串绑定。但是，在处理比较复杂的绑定时，通过拼接生成字符串是麻烦且易出错的。因此，Vue 专门为 class 和 style 的 v-bind 用法提供了特殊的功能增强。除了字符串外，表达式的值也可以是对象或数组。

举个例子：

你想让一个按钮点击后变“高亮”，如果用 Vue，可以这么写

<template>

<button :class="{ active: isActive }" @click="toggle">

点击切换状态

</button>

</template>

<script setup>

import { ref } from 'vue'

const isActive = ref(false)

function toggle() {

isActive.value = !isActive.value

}

</script>

<style>

.active {

background-color: green;

color: white;

}

</style>

上面这个按钮：

* 一开始没有 active class，所以是普通样式。
* 点击后 isActive 变成 true，Vue 自动帮你把 class="active" 加上！
* 再点击，又去掉 class="active"，样式就变回来了。

**条件渲染**

**v-if**​

v-if 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回真值时才被渲染。

<h1 v-if="awesome">Vue is awesome!</h1>

也就是说只有当awesome的值为true时，Vue is awesome!才会被渲染

**v-else**​

也可以使用 v-else 为 v-if 添加一个“else 区块”。类似于java的if...else

<button @click="awesome = !awesome">Toggle</button>

<h1 v-if="awesome">Vue is awesome!</h1>

<h1 v-else>Oh no</h1>

也就是说，当awesome为true时渲染Vue is awesome!，为false时渲染Oh no

**v-else-if**​

顾名思义，v-else-if 提供的是相应于 v-if 的“else if 区块”。它可以连续多次重复使用：（相当于java的else if）

<div v-if="type === 'A'">

A

</div>

<div v-else-if="type === 'B'">

B

</div>

<div v-else-if="type === 'C'">

C

</div>

<div v-else>

Not A/B/C

</div>

注意！！v-else-if和v-else都是v-if的区块指令，也就是说必须紧跟在一个 v-if 或一个 v-else-if 元素后面。

**v-show**

另一个可以用来按条件显示一个元素的指令是 v-show。其用法基本一样：

<h1 v-show="ok">Hello!</h1>

不同之处在于 v-show 会在 DOM 渲染中保留该元素；v-show 仅切换了该元素上名为 display 的 CSS 属性。

而且v-show 不支持在 <template> 元素上使用，也不能和 v-else 搭配使用。

**v-if vs. v-show​**

| **特性** | **v-if** | **v-show** |
| --- | --- | --- |
| DOM 是否存在 | 不满足条件时元素 **完全不渲染** | 元素一直在 DOM，只是隐藏 (display: none) |
| 初始渲染性能 | 较慢，因需判断并构建 DOM | 快，因默认已渲染 |
| 切换时性能 | 较慢（创建或销毁 DOM） | 快（仅切换样式） |
| 适合的场景 | 条件**偶尔变化**（如登录成功显示欢迎界面） | 条件**频繁变化**（如 tab 切换） |

**列表渲染**​

**v-for​**

我们可以使用 v-for 指令基于一个数组来渲染一个列表。v-for 指令的值需要使用 item in items 形式的特殊语法，其中 items 是源数据的数组，而 item 是迭代项的别名。注意！item和items的值都是可以自定义的：

<template>

<ul>

<li v-for="fruit in fruits" :key="fruit">

{{ fruit }}

</li>

</ul>

</template>

<script setup>

const fruits = ["Apple", "Banana", "Grape"]

</script>

运行后页面会显示：

Apple

Banana

Grape

上文是fruits里面只有一个fruit的情况，但也可以手动插入index：

<template>

<ul>

<li v-for="(fruit, index) in fruits" :key="index">

{{ index + 1 }} - {{ fruit }}

</li>

</ul>

</template>

<script setup>

const fruits = ["Apple", "Banana", "Grape"]

</script>

运行后会显示：

1 - Apple

2 - Banana

3 - Grape

当然也可以遍历对象：

<template>

<ul>

<li v-for="(value, key) in user" :key="key">

{{ key }}: {{ value }}

</li>

</ul>

</template>

<script setup>

const user = {

name: "小明",

age: 20,

hobby: "coding"

}

</script>

渲染结果会是这样的：

name: 小明

age: 20

hobby: coding

**在 v-for 里使用范围值​**

v-for 可以直接接受一个整数值。在这种用例中，会将该模板基于 1...n 的取值范围重复多次。

<span v-for="n in 10">{{ n }}</span>

注意！此处 n 的初值是从 1 开始而非 0。

渲染结果：

<span>1</span>

<span>2</span>

<span>3</span>

<span>4</span>

<span>5</span>

<span>6</span>

<span>7</span>

<span>8</span>

<span>9</span>

<span>10</span>

注意！！v-for 与 v-if​

当它们同时存在于一个节点上时，v-if 比 v-for 的优先级更高。这意味着 v-if 的条件将无法访问到 v-for 作用域内定义的变量别名，因此不推荐v-for 与 v-if一起使用（即使在外先包装一层 <template> 再在其上使用 v-for 可以解决这个问题）。

**数组变化**

**变更方法​**

Vue 能够侦听响应式数组的变更方法，并在它们被调用时触发相关的更新。这些变更方法包括（以下方法都会改变原数组）：

push()：向数组的末尾添加一个或多个元素，并返回新数组的长度

pop()：删除数组的最后一个元素并返回该元素

shift()：删除数组的第一个元素并返回该元素

unshift(): 向数组的开头添加一个或多个元素，并返回新数组的长度

splice(): 修改原数组，可以用来添加、删除、替换数组中的元素

splice(start, deleteCount, item1, item2, ...)

也就是说从第几位开始，删除几个元素，然后插入新元素1，新元素2...

sort(): 对数组进行排序（默认按字符串排序）

reverse(): 反转数组的顺序

不变方法

相对地，也有一些不可变 (immutable) 方法，例如 filter()，concat() 和 slice()，这些方法都不会更改原数组，而总是返回一个新数组。当遇到的是非变更方法时，我们需要将旧的数组替换为新的：

filter()：根据条件筛选元素，返回新数组，不影响原数组

const arr = [1, 2, 3, 4];

const even = arr.filter(n => n % 2 === 0);

console.log(even); *// [2, 4]*

console.log(arr); *// 原数组还是 [1, 2, 3, 4]*

concat()：提取一段子数组，返回新数组，不修改原数组

const arr = [1, 2];

const result = arr.concat([3, 4]);

console.log(result); *// [1, 2, 3, 4]*

console.log(arr); *// 原数组不变 [1, 2]*

slice()：连接两个或多个数组，返回一个新数组

const arr = [1, 2, 3, 4];

const part = arr.slice(1, 3); *// 从下标 1 到 3（不含）*

console.log(part); *// [2, 3]*

console.log(arr); *// 仍然是 [1, 2, 3, 4]*

**事件处理​**

**监听事件​**

我们可以使用 v-on 指令 (简写为 @) 来监听 DOM 事件，并在事件触发时执行对应的 JavaScript。用法：v-on:click="handler" 或 @click="handler"。

**事件修饰符​**

在处理事件时调用 event.preventDefault() 或 event.stopPropagation() 是很常见的。尽管我们可以直接在方法内调用，但如果方法能更专注于数据逻辑而不用去处理 DOM 事件的细节会更好。

为解决这一问题，Vue 为 v-on 提供了事件修饰符。修饰符是用 . 表示的指令后缀，包含以下这些：

.stop 等同于 event.stopPropagation()，阻止事件冒泡

.prevent 等同于 event.preventDefault()，阻止默认行为

.self 只在事件由绑定元素本身触发时才触发回调（不响应子元素冒泡上来的事件）

.capture 添加事件监听时使用捕获模式（默认是冒泡模式）

.once 事件只触发一次

.passive 告诉浏览器这是个不会调用 preventDefault() 的事件处理器（提高性能）

<!-- 单击事件将停止传递 -->

<a @click.stop="doThis"></a>

<!-- 提交事件将不再重新加载页面 -->

<form @submit.prevent="onSubmit"></form>

<!-- 修饰语可以使用链式书写 -->

<a @click.stop.prevent="doThat"></a>

<!-- 也可以只有修饰符 -->

<form @submit.prevent></form>

<!-- 仅当 event.target 是元素本身时才会触发事件处理器 -->

<!-- 例如：事件处理器不来自子元素 -->

<div @click.self="doThat">...</div>

**按键修饰符​**

在监听键盘事件时，我们经常需要检查特定的按键。Vue 允许在 v-on 或 @ 监听按键事件时添加按键修饰符。

比如：

<!-- 仅在 `key` 为 `Enter` 时调用 `submit` -->

<input @keyup.enter="submit" />

**按键别名**

Vue 为一些常用的按键提供了别名：

.enter

.tab

.delete (捕获“Delete”和“Backspace”两个按键)

.esc

.space

.up

.down

.left

.right

.ctrl

.alt

.shift

**鼠标按键修饰符**

.left：左键

.right：右键

.middle：滚轮点击

**表单输入绑定：v-model**

v-model 是 Vue.js 中一个非常常用的指令，它用于在表单控件元素（如 <input>、<textarea>、<select> 等）和数据之间创建双向绑定。通过 v-model，Vue 会自动处理事件和数据的更新。

在表单元素上使用 v-model，你可以将元素的值绑定到 Vue 实例的数据，并且该值会自动更新。

例如：

<template>

<div>

<input v-model="message" />

<p>{{ message }}</p>

</div>

</template>

<script setup>

import { ref } from 'vue';

const message = ref('');

</script>

在这个例子中，<input> 元素的值会和 message 变量双向绑定。当用户输入内容时，message 会自动更新，并且当 message 的值发生变化时，<input> 中的内容也会自动更新。

**V-model修饰符**

.lazy​

默认情况下，v-model 会在每次 input 事件后更新数据 (IME 拼字阶段的状态例外)。你可以添加 lazy 修饰符来改为在每次 change 事件后更新数据：

<!-- 在 "change" 事件后同步更新而不是 "input" -->

<input v-model.lazy="msg" />

.number​

如果你想让用户输入自动转换为数字，你可以在 v-model 后添加 .number 修饰符来管理输入：

<input v-model.number="age" />

如果该值无法被 parseFloat() 处理，那么将返回原始值。特别是当输入为空时 (例如用户清空输入字段之后)，会返回一个空字符串。这种行为与 DOM 属性 valueAsNumber 有所不同。

number 修饰符会在输入框有 type="number" 时自动启用。

.trim​

如果你想要默认自动去除用户输入内容中两端的空格，你可以在 v-model 后添加 .trim 修饰符：

<input v-model.trim="msg" />

**Vue监听器 watch API**

Vue 提供的 watch API（监听器）可以监听一个或多个响应式数据源，并在它们变化时执行回调函数。

1. **监听一个值（ref）**

const message = ref('Hello')

watch(message, (newVal, oldVal) => {

console.log('message changed:', newVal)

})

注意！！

new：是 **响应式数据变化后的新值**。

old：是 **响应式数据变化前的旧值**。

这两个参数是 watch API 内部传递给回调函数的，**你不能自己规定什么值才会触发回调**。它们会自动随着你监听的 ref 或 reactive 数据的变化而变化。

1. **监听多个值（数组）**

const name = ref('')

const age = ref(0)

watch([name, age], ([newName, newAge], [oldName, oldAge]) => {

console.log('Name or age changed')

})

1. **监听对象的属性（默认浅监听）**

const user = ref({ name: '小明', age: 20 })

watch(user, (newVal, oldVal) => {

console.log('user changed')

})

注意！！但是如果你修改的是 user.value.name，这个监听器不会触发，因为默认是浅监听。

**4.深层侦听器**

直接给 watch() 传入一个响应式对象，会隐式地创建一个深层侦听器——该回调函数在所有嵌套的变更时都会被触发：

* watch() 会监听整个响应式对象的变化，包括嵌套的属性。
* 如果你没有传递 { deep: true }，Vue 会根据实际情况（如对象是否嵌套）自动处理监听。

但要注意！！深层侦听的方式在性能上可能会有开销，特别是当对象非常大时，Vue 会对每一个嵌套属性进行检测。

import { reactive, watch } from 'vue'

*// 创建一个响应式对象*

const obj = reactive({

user: { name: 'Alice', age: 25 },

count: 0

})

*// 监听整个 obj 的变化*

watch(obj, (newValue, oldValue) => {

console.log('obj 发生变化了：', newValue, oldValue)

})

*// 触发嵌套的变化*

obj.user.name = 'Bob' *// 这个变化会触发 watch*

obj.count++ *// 这个变化也会触发 watch*

注意！！当监听一个大对象时，可以通过 getter 函数来监听某个具体属性，避免不必要的深层侦听：

**方式 1：监听具体属性**

watch(() => obj.count, (newVal, oldVal) => {

console.log('count changed:', newVal)

})

**方式 2：监听嵌套属性**

watch(() => obj.user.name, (newVal, oldVal) => {

console.log('user.name changed:', newVal)

})

优势：只在你关心的属性变化时触发，性能更好，逻辑更清晰

**5. 立即执行（immediate）**

如果你希望在一开始就执行一次回调，可以加 immediate: true

watch(count, (newVal, oldVal) => {

console.log('count changed or initialized')

}, { immediate: true })

1. **停止监听（stop handle）**

watch() 返回一个函数，调用这个函数就可以停止监听。

const stop = watch(count, (newVal) => {

console.log('监听中...')

})

// 在某个时机手动停止监听，调用stop方法

stop()

1. **一次性监听器**

如果希望回调只在源变化时触发一次，请使用 once: true 选项。

watch(source, (newValue, oldValue) => {

// 当 `source` 变化时，仅触发一次

},{ once: true })

**watchEffect()指令**

是 Vue 3 提供的一个更简洁的 响应式副作用监听函数，它会在 依赖的响应式数据发生变化时自动重新执行回调函数。

| **功能** | **watch()** | **watchEffect()** |
| --- | --- | --- |
| 用于监听 | 明确指定一个或多个响应式数据 | 自动收集用到的所有响应式数据 |
| 回调参数 | 提供新值和旧值 | 不提供 |
| 自动运行 | 初始化时不会自动触发（除非配置） | **初始化时自动执行一次** |
| 适用场景 | 明确监听某些值变化（可获取旧值） | 快速响应式处理副作用逻辑（更简洁） |

示例：

import { ref, watchEffect } from 'vue'

const count = ref(0)

watchEffect(() => {

console.log(`当前 count 是：${count.value}`)

})

首次执行时自动输出：

当前 count 是：0

且每次 count.value 改变时，都会重新执行这个回调函数。

**停止监听**

你可以手动停止监听（就像 watch() 那样），用stop（）

**watch 或 watchEffect 指令的“副作用”**

在 Vue（或更广义的响应式编程中）提到的 副作用（Side Effect），指的是：**某个响应式数据发生变化后，所引发的额外行为或结果**

比如说：

const count = ref(0)

watchEffect(() => {

console.log(`count 变成了：${count.value}`)

})

这个 console.log 就是一个副作用。它**不是数据本身**，而是**响应数据变化触发的一段逻辑**。

在vue中的副作用并不是“坏事”，它只是指**「对外部世界有影响或依赖外部状态的逻辑」**，而不是纯粹的值变化。也就是说，副作用（side effect）就是：**在响应式数据变化时执行的、对组件或程序“外部状态”产生影响的代码。**

**常用副作用：**

**副作用行为**  **说明**

console.log() 日志输出

localStorage.setItem() 本地存储

DOM.style.xxx 直接修改 DOM

fetch() / axios() 请求 API 接口

clearInterval() 清除定时器等资源回收

总结一句：

只要 watch 或 watchEffect 里面的代码**“影响了外部世界”**，就属于**副作用**。

**模板引用**

虽然 Vue 的声明性渲染模型为你抽象了大部分对 DOM 的直接操作，但在某些情况下，我们仍然需要直接访问底层 DOM 元素。要实现这一点，我们可以使用特殊的 ref 属性

要在组合式 API 中获取引用，我们可以使用辅助函数 useTemplateRef() ：

<script setup>

import { useTemplateRef, onMounted } from 'vue'

// 第一个参数必须与模板中的 ref 值匹配

const input = useTemplateRef('my-input')

onMounted(() => {

input.value.focus()

})

</script>

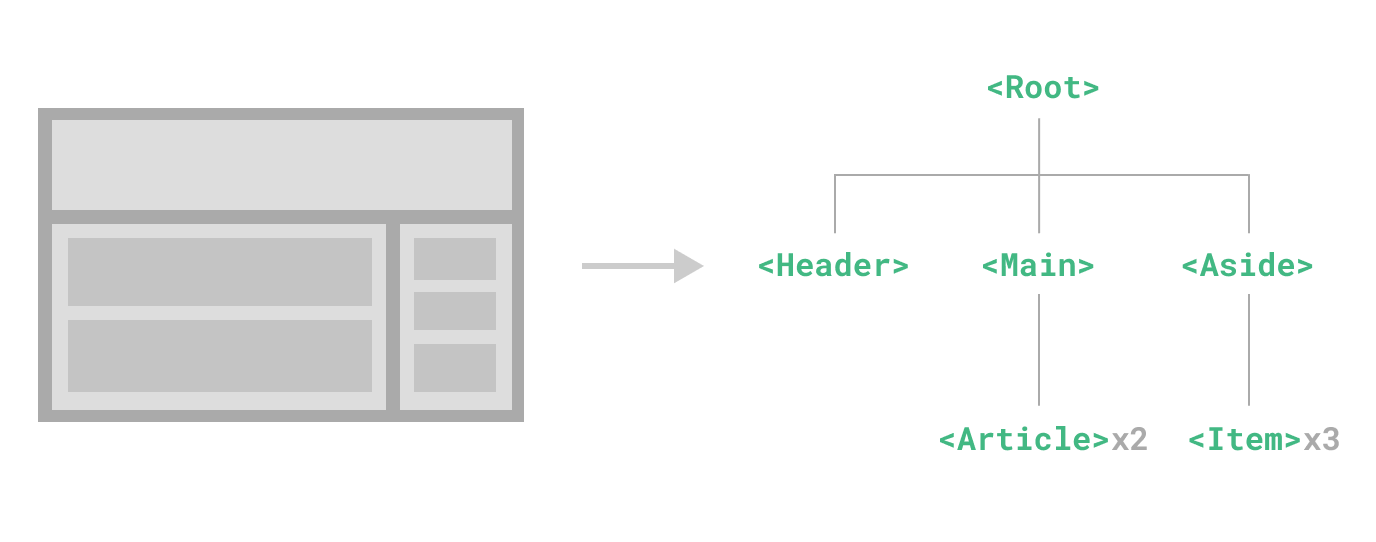
<template>

<input ref="my-input" />

</template>

**组件基础**

组件允许我们将 UI 划分为独立的、可重用的部分，并且可以对每个部分进行单独的思考。在实际应用中，组件常常被组织成一个层层嵌套的树状结构：



如上图所示，各个组件之间可以嵌套，形成父子组件关系，比如新建两个.vue文件，这两个文件可以相互嵌套：

**ButtonCounter.vue文件（子组件）：**

<script setup>

import { ref } from 'vue'

const count = ref(0)

</script>

<template>

<button @click="count++">You clicked me {{ count }} times.</button>

</template>

**App.vue文件（父组件）：**

<script setup>

import ButtonCounter from './ButtonCounter.vue' //导入子组件

</script>

<template>

<h1>Here is a child component!</h1>

<ButtonCounter /> //在模板中渲染子组件（用子组件名本身当标签名）

<ButtonCounter /> //可以多次调用，会生成多个渲染

</template>

**传递 props（父传子）**

当希望从**父组件**向**子组件**中传递数据时，要用props属性。在 Vue 中，props（属性）是父组件用来向子组件传递数据的方式，是组件间通信的基础手段之一。

**1. 父组件传递 Props**

<!-- App.vue -->

<template>

<ChildComponent :username="name" />

</template>

<script setup>

import ChildComponent from './ChildComponent.vue'

import { ref } from 'vue'

const name = ref('小明')

</script>

这里我们把 name 传给子组件 ChildComponent，props 名叫 username。

**2. 子组件接收 Props**

<!-- ChildComponent.vue -->

<template>

<p>欢迎，{{ username }}！</p> //欢迎，小明！

</template>

<script setup>

const props = defineProps({

username: String

})

</script>

现在 username 就是父组件传进来的数据了，子组件可以直接用。

注意！！Props是单向绑定的，也就是说子组件中不能直接修改props

Props特性 说明

单向绑定 父 → 子，子组件不能直接修改 props（除非 .sync）

类型验证 defineProps({ xxx: String }) 可以限制类型

支持默认值 可配合 withDefaults() 设置默认值

响应式更新 父组件更新 props，子组件也会自动更新

**监听事件emit（子传父）**

当希望在子组件的事件对父组件产生影响时，要用emit，也就是说从子组件生成 emit 事件，并且在父组件监听事件

1. **子组件：使用 emit 触发事件**

<!-- ChildComponent.vue -->

<template>

<button @click="sendMessage">点击发送</button>

</template>

<script setup>

const emit = defineEmits(['send']) //定义emit事件send

function sendMessage() {

emit('send', 'Hello from child!') //在函数中触发事件

}

</script>

子组件通过 emit('send', ...) 向父组件发送 send 事件，并可以**附带数据**。

**2. 父组件：监听子组件的事件**

<!-- ParentComponent.vue -->

<template>

<ChildComponent @send="handleSend" /> //用@监听send事件

</template>

<script setup>

import ChildComponent from './ChildComponent.vue'

function handleSend(msg) {

console.log('收到子组件消息:', msg) //当事件发生时，接受参数并调用函数

}

</script>

**生命周期钩子**

生命周期钩子（Lifecycle Hooks）是 Vue 组件在创建、挂载、更新和销毁等阶段所触发的函数。你可以在这些阶段执行特定操作，比如获取数据、清理定时器、打印调试信息等。

在 <script setup> 中使用生命周期钩子：

<script setup>

import { onMounted, onUnmounted, onUpdated, onBeforeMount } from 'vue'

onMounted(() => {

console.log('组件挂载完成！')

})

onUnmounted(() => {

console.log('组件已卸载')

})

onUpdated(() => {

console.log('组件更新了')

})

onBeforeMount(() => {

console.log('组件准备挂载')

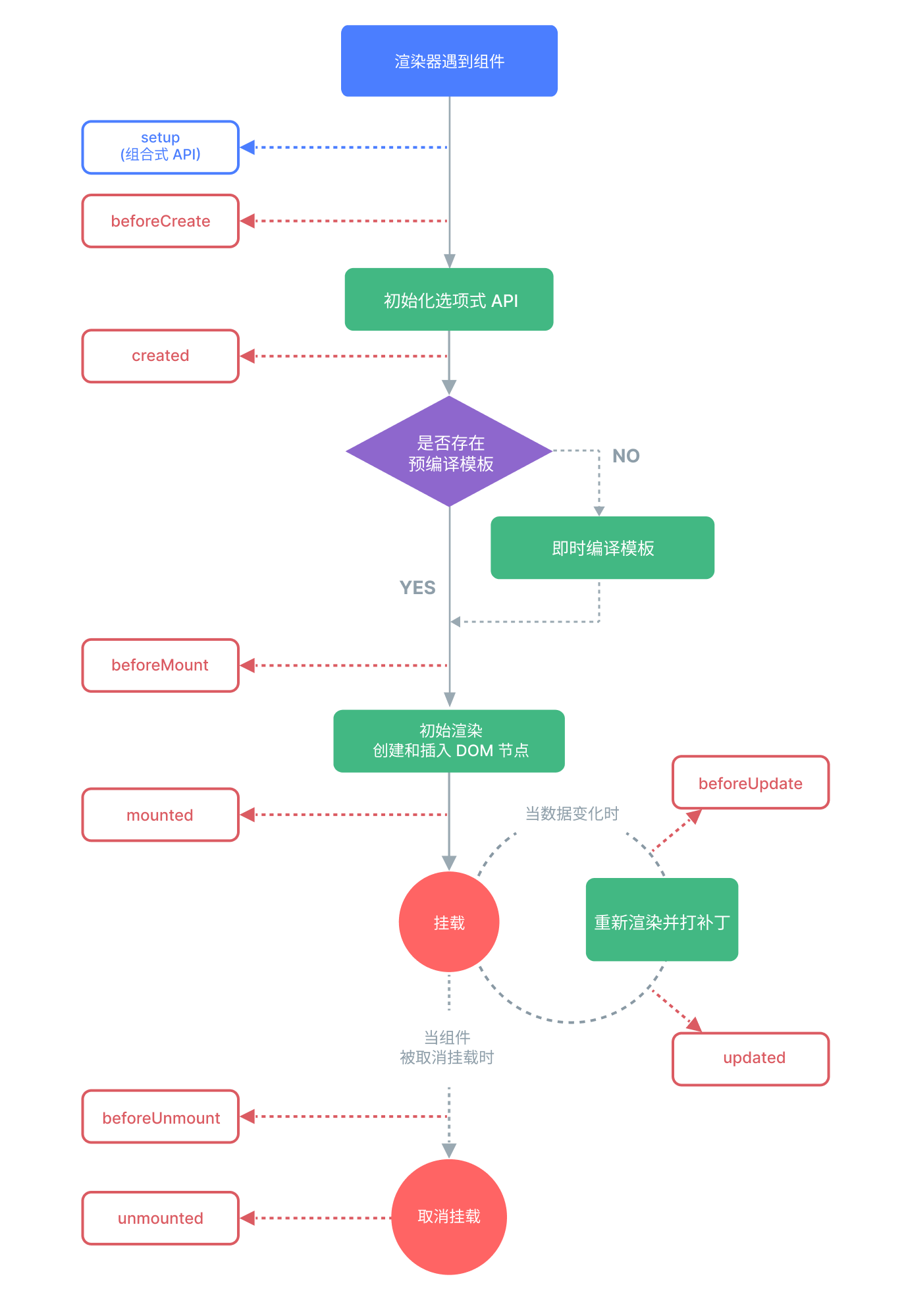
})

</script>

常用生命周期钩子一览表

| **生命周期钩子** | **说明** |
| --- | --- |
| onBeforeMount | 组件挂载前触发 |
| onMounted | 组件挂载后触发 |
| onBeforeUpdate | 数据更新前触发 |
| onUpdated | 数据更新后触发 |
| onBeforeUnmount | 卸载前触发 |
| onUnmounted | 卸载后触发 |

**Vue的生命周期示意图：**



**组件注册**

**1. 局部注册（推荐）**

这是最常用的方式，只在当前组件中使用某个子组件。详细见上文的父子组件嵌套

<!-- ParentComponent.vue -->

<template>

<ChildComponent />

</template>

<script setup>

import ChildComponent from './ChildComponent.vue'

</script>

**2. 全局注册**

如果你有一个组件会在很多地方反复使用，比如一个通用的按钮、对话框等，可以用全局注册。

**第一步：**在 main.js中注册：

import { createApp } from 'vue'

import App from './App.vue'

import GlobalComponent from './components/GlobalComponent.vue'

//导入要注册的组件

const app = createApp(App)

app.component('GlobalComponent', GlobalComponent)

//第一个GlobalComponent为注册的组件名，就是写在模板里的<GlobalComponent />

//第二个GlobalComponent为引入的组件对象：'./components/GlobalComponent.vue'

app.mount('#app')

**第二步：**然后你可以在任何组件中直接使用它：

<template>

<GlobalComponent />

</template>

**Props 声明**

一个组件需要显式声明它所接受的 props，这样 Vue 才能知道外部传入的哪些是 props，哪些是透传 attribute

在使用 <script setup> 的单文件组件中，props 可以使用 defineProps() 宏来声明：

（前文的 父传子时 就用了props）

<script setup>

const props = defineProps(['foo'])

console.log(props.foo)

</script>

除了使用字符串数组来声明 props 外，还可以使用对象的形式：

defineProps({

title: String,

likes: Number

})

**单向数据流**

所有的 props 都遵循着单向绑定原则，props 因父组件的更新而变化，自然地将新的状态向下流往子组件，而不会逆向传递。这避免了子组件意外修改父组件的状态的情况

每次父组件更新后，所有的子组件中的 props 都会被更新到最新值，这意味着你不应该在子组件中去更改一个 prop。若你这么做了，Vue 会在控制台上向你抛出警告：

const props = defineProps(['foo'])

// ❌ 警告！prop 是只读的！

props.foo = 'bar'

注意！！如果我们需要根据子组件的变化更改父组件，在大多数场景下，子组件应该抛出一个emit事件来通知父组件做出改变。

**Prop 校验**

Vue 组件可以更细致地声明对传入的 props 的校验要求。要声明对 props 的校验，你可以向 defineProps() 宏提供一个带有 props 校验选项的对象，例如：

defineProps({

// 基础类型检查

// （给出 `null` 和 `undefined` 值则会跳过任何类型检查）

propA: Number,

// 多种可能的类型

propB: [String, Number],

// 必传，且为 String 类型

propC: {

type: String,

required: true

},

// 必传但可为 null 的字符串

propD: {

type: [String, null],

required: true

},

// Number 类型的默认值

propE: {

type: Number,

default: 100

},

// 对象类型的默认值

propF: {

type: Object,

// 对象或数组的默认值

// 必须从一个工厂函数返回。

// 该函数接收组件所接收到的原始 prop 作为参数。

default(rawProps) {

return { message: 'hello' }

}

},

// 自定义类型校验函数

// 在 3.4+ 中完整的 props 作为第二个参数传入

propG: {

validator(value, props) {

// The value must match one of these strings

return ['success', 'warning', 'danger'].includes(value)

}

},

// 函数类型的默认值

propH: {

type: Function,

// 不像对象或数组的默认，这不是一个

// 工厂函数。这会是一个用来作为默认值的函数

default() {

return 'Default function'

}

}

})

**Emit事件校验**​

和对 props 添加类型校验的方式类似，所有触发的事件也可以使用对象形式来描述。

要为事件添加校验，那么事件可以被赋值为一个函数，接受的参数就是抛出事件时传入 emit 的内容，返回一个布尔值来表明事件是否合法。比如：

<script setup>

const emit = defineEmits({

// 没有校验

click: null,

// 校验 submit 事件

submit: ({ email, password }) => {

if (email && password) {

return true

} else {

console.warn('Invalid submit event payload!')

return false

}

}

})

function submitForm(email, password) {

emit('submit', { email, password })

}

</script>

* submitForm函数接收用户的 email 和 password，然后尝试触发 submit 事件。
* 如果 email 或 password 缺失，事件就不会真正发出，并会在控制台看到警告。

**透传 Attributes**

当你在父组件中使用子组件时，传给子组件的 非 props 属性（也就是没有被 props 声明的）默认会“透传”到子组件的 根元素 上。这种机制就叫 **属性透传**（Attribute Inheritance / Attribute Fallthrough）。比如：

<!-- Parent.vue -->（父）

<template>

<MyButton class="btn" type="submit" />

</template>

<!-- MyButton.vue -->（子）

<template>

<button>Click Me</button>

</template>

<script setup>

defineProps([]) // 没有声明任何 props

</script>

虽然 class="btn" 和 type="submit" 没在 MyButton 的 props 中声明，

但它们 会自动加到 <button> 上，因为 <button> 是根节点。

**禁用 Attributes 继承**

如果 不想自动透传，可以使用inheritAttrs，这样处理：

<script setup>

defineOptions({

inheritAttrs: false

})

</script>

<template>

<!-- 手动绑定 $attrs -->

<button v-bind="$attrs">Click Me</button>

</template>

这样你可以决定属性绑定到哪里，比如：

<button v-bind="$attrs" class="custom">Click Me</button>

**$attrs 是什么：**

* $attrs 是一个对象，包含了所有未被 props 接收的属性。
* 可以手动通过 v-bind="$attrs" 传给任何你想要的子节点。
* 常用于封装组件时的样式透传、事件透传等。

**插槽 Slots**

插槽（slot）是 Vue 的一个超强特性，用来让父组件把 HTML 内容“插”到子组件的指定位置，非常适合写可复用的 UI 组件。比如：

<!-- 父组件 -->

<MyCard>

<p>这里是插进 MyCard 里的内容</p>

</MyCard>

<!-- 子组件 MyCard.vue -->

<template>

<div class="card">

<slot></slot> <!-- 插槽占位符 -->

</div>

</template>

注意！！slot 标签的位置就是传入内容显示的位置。

**具名插槽（多个插槽）**

用来定位插入内容的位置

<!-- 父组件 -->

<MyDialog>

<template #header>

<h2>我是标题</h2>

</template>

<template #default>

<p>我是正文内容</p>

</template>

<template #footer>

<button>关闭</button>

</template>

</MyDialog>

<!-- 子组件 MyDialog.vue -->

<template>

<div class="dialog">

<header><slot name="header" /></header>

<main><slot /></main> <!-- 如果未命名就是default 插槽 -->

<footer><slot name="footer" /></footer>

</div>

</template>

**作用域插槽（带数据回传）**

当子组件希望把数据传给插槽时，就要用到 作用域插槽（Scoped Slot）。

<!-- 子组件 ListRenderer.vue -->

<template>

<ul>

<li v-for="item in items" :key="item.id">

<slot :item="item" />

</li>

</ul>

</template>

<script setup>

defineProps({

items: Array

})

</script>

<!-- 父组件 -->

<ListRenderer :items="[{ id: 1, name: 'Vue' }, { id: 2, name: 'React' }]">

<template #default="{ item }">

<strong>{{ item.name }}</strong>

</template>

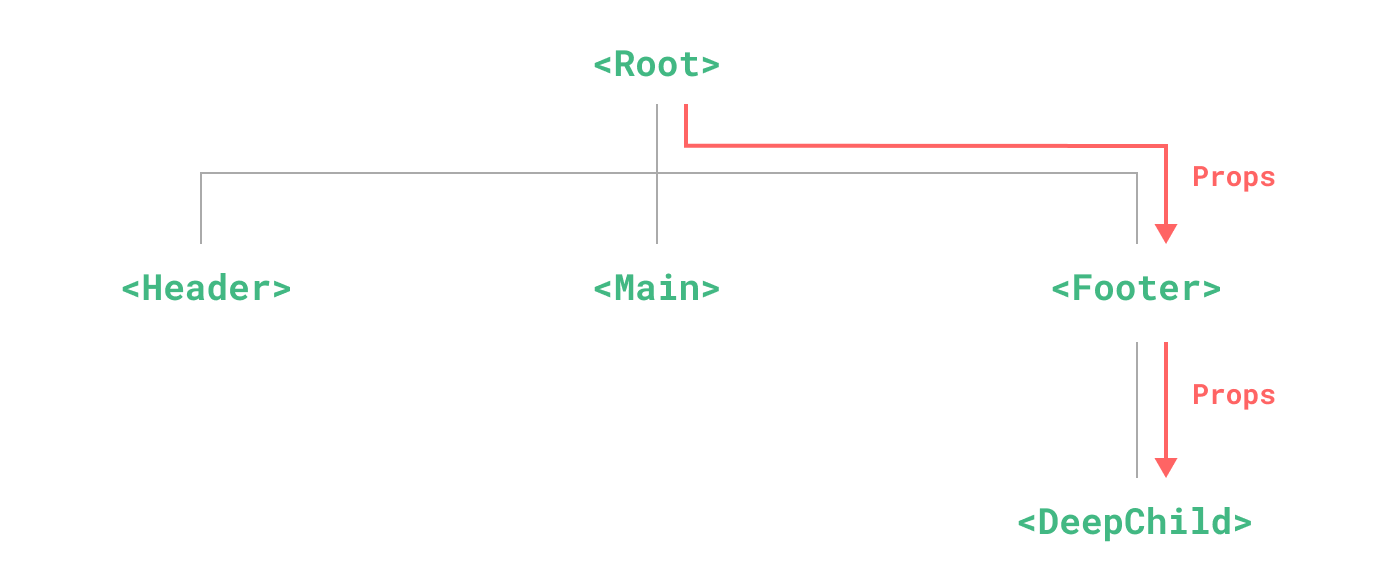
</ListRenderer>

插槽中通过 #default="{ item }" 解构了子组件传过来的对象。

**依赖注入**

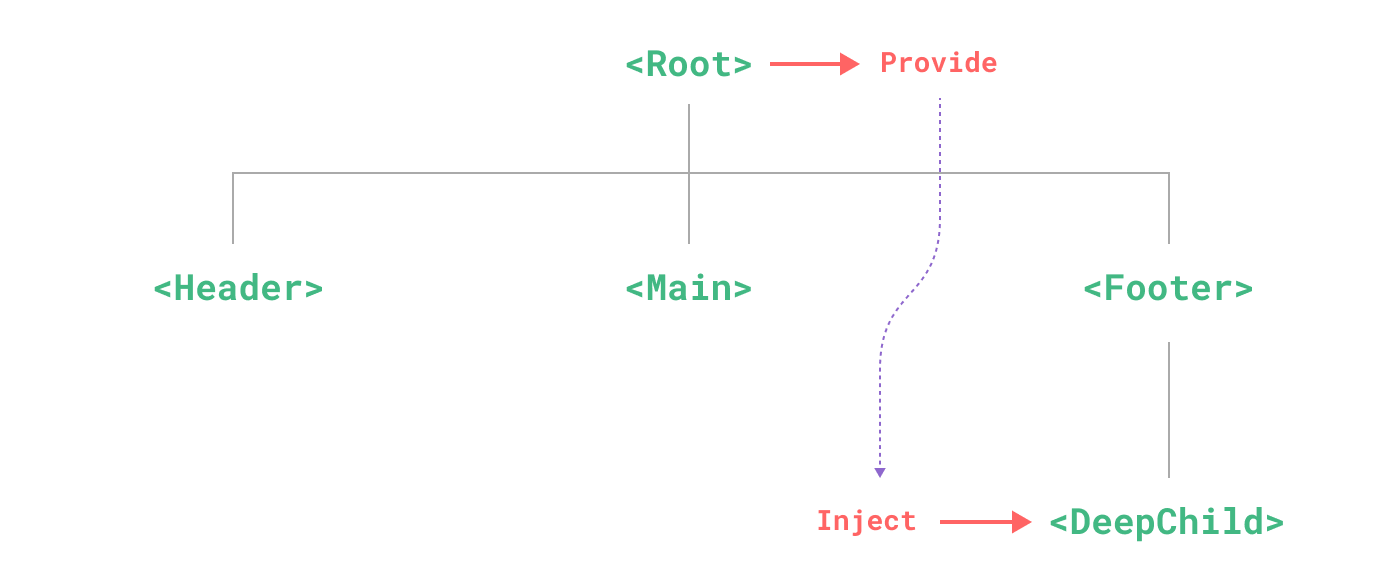
**Prop 逐级透传问题**​

通常情况下，当我们需要从父组件向子组件传递数据时，会使用 props。想象一下这样的结构：有一些多层级嵌套的组件，形成了一棵巨大的组件树，而某个深层的子组件需要一个较远的祖先组件中的部分数据。在这种情况下，如果仅使用 props 则必须将其沿着组件链逐级传递下去，这会非常麻烦：



注意！！虽然这里的 <Footer> 组件可能根本用不到这些 props，但为了使 <DeepChild> 能访问到它们，仍然需要定义并向下传递。如果组件链路非常长，可能会影响到更多这条路上的组件。这一问题被称为“prop 逐级透传”，显然是我们希望尽量避免的情况。

这时候就要用到 **依赖注入** ，Vue 的依赖注入（Dependency Injection）是一种在祖先组件中提供数据，在后代组件中注入使用的机制，非常适合跨层级传值，不用通过 props 一层层传。关键字为provide 和 inject



父组件（祖先）

<script setup>

import { ref, provide } from 'vue'

const user = ref({ name: 'Alice' })

provide('user', user) // key: 'user'，值是响应式对象

</script>

<template>

<ChildComponent />

</template>

子组件（任意层级）

<script setup>

import { inject } from 'vue'

const user = inject('user') // 获取到父组件提供的 user

console.log(user.value.name) // 访问 user 的 name

</script>

<template>

<div>Hello, {{ user?.name }}</div>

</template>

**异步组件**

异步组件就是：当组件需要被用到时，才去加载它的代码，常用于按需加载、优化打包体积等。关键字为：defineAsyncComponent ，基本写法为：

<template>

<Suspense>

<AsyncComponent />

</Suspense>

</template>

<script setup>

import { defineAsyncComponent } from 'vue'

const AsyncComponent = defineAsyncComponent({

loader: () => import('./MyBigComponent.vue'), // 组件懒加载

loadingComponent: LoadingSpinner, // 加载时显示

errorComponent: LoadFailed, // 加载失败时显示

delay: 200, // 延迟多少 ms 再显示 loading

timeout: 3000 // 超过多少 ms 报错

})

</script>

**组合式函数**

组合式函数（也叫 Composables）就是你自己封装的函数，里面用到了 Vue 的响应式 API（比如 ref、reactive、watch、onMounted 等），然后把状态或逻辑暴露出来，供多个组件使用。它的目标是 复用逻辑 + 让组件更清爽

比如：先定义一个组合式函数useCounter.js

// useCounter.js

import { ref } from 'vue'

export function useCounter() {

const count = ref(0)

const increment = () => {

count.value++

}

return { count, increment }

}

然后在组件中使用它：

<script setup>

import { useCounter } from './useCounter'

const { count, increment } = useCounter()

</script>

<template>

<button @click="increment">Clicked: {{ count }}</button>

</template>

**自定义指令**

Vue 自定义指令，就是你可以手动写的 v-xxx 指令，用来给 DOM 元素加上自定义行为，比如自动聚焦、点击外部关闭、懒加载等。Vue 自带一些内置指令（比如 v-model、v-if），而自定义指令让你可以扩展 Vue 的 DOM 操作能力。你可以通过 directive() 或 directives 选项注册属于你自己的 v-xxx。

注册时分为 全局注册指令 和 组件内注册指令 两种形式：

**1，全局注册指令（在main.js中注册）：**

// main.js 或 main.ts

app.directive('focus', {

mounted(el) {

el.focus()

}

})

使用方式：

<input v-focus />

也就是说：在全局中创建了一个 v-focus 指令，元素挂载后自动获得焦点。

**2，组件内注册指令（直接在组件内注册并使用）：**

<script setup>

const vHighlight = {

mounted: (el) => {

el.classList.add('is-highlight')

}

}

</script>

<template>

// 在模板中启用 v-highlight

<p v-highlight>This sentence is important!</p>

</template>