# 概述

Vue (发音为 /vjuː/，类似 view) 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，无论任务是简单还是复杂。

最基本的示例：

<div id="app">

<button @click="count++">

Count is: {{ count }}

</button>

</div>

import { createApp } from 'vue'

createApp({

data() {

return {

count: 0

}

}

}).mount('#app')

上面的示例展示了两个 Vue 的核心功能：

* 声明式渲染：Vue 通过自己的模板语法扩展了标准 HTML，使得我们可以声明式地描述基于 JavaScript 状态输出的 HTML。
* 响应性：Vue 会自动跟踪 JavaScript 状态变化并在改变发生时响应式地更新 DOM。

## 渐进式框架

Vue 是一个框架和生态，功能覆盖了大部分前端开发常见的需求。但 Web 世界又是十分多样化的，我们在 Web 上构建的东西可能在形式和规模上有很大不同。考虑到这一点，Vue 被设计成具有灵活性和可逐步集成的特点。根据你的需求场景，Vue 可以按不同的方式使用：

* 增强静态的 HTML 而无需构建步骤
* 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)
* 目标为桌面端、移动端、WebGL，甚至是命令行终端

## 单文件组件

在大多数启用了构建工具的 Vue 项目中，我们可以使用一种类似 HTML 格式的文件来书写 Vue 组件，它被称为单文件组件 (也被称为 \*.vue 文件，英文缩写为 SFC)。顾名思义，Vue 的单文件组件会将一个组件的逻辑 (JavaScript)，模板 (HTML) 和样式 (CSS) 封装在同一个文件里。下面将用单文件组件的格式重写上面的计数器示例：

<script>

export default {

data() {

return {

count: 0

}

}

}

</script>

<template>

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

</template>

<style scoped>

button {

font-weight: bold;

}

</style>

## API 风格

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

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。

<script>

export default {

// data() 返回的属性将会成为响应式的状态

// 并且暴露在 `this` 上

data() {

return {

count: 0

}

},

// methods 是一些用来更改状态与触发更新的函数

// 它们可以在模板中作为事件监听器绑定

methods: {

increment() {

this.count++

}

},

// 生命周期钩子会在组件生命周期的各个不同阶段被调用

// 例如这个函数就会在组件挂载完成后被调用

mounted() {

console.log(`The initial count is ${this.count}.`)

}

}

</script>

<template>

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

</template>

### 组合式 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 需要在编译时进行转换，来减少使用组合式 API 时的样板代码。例如，<script setup> 中的导入和顶层变量/函数都能够在模板中直接使用。下面是使用了组合式 API 与 <script setup> 改造后和上面的模板完全一样的组件：

<script setup>

import { ref, onMounted } from 'vue'

// 响应式状态

const count = ref(0)

// 用来修改状态、触发更新的函数

function increment() {

count.value++

}

// 生命周期钩子

onMounted(() => {

console.log(`The initial count is ${count.value}.`)

})

</script>

<template>

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

</template>

## 构建

启用构建工具的 Vue 项目：

npm init vue@latest

npm install

npm run dev

# Vue 应用

## 应用实例

每个 Vue 应用都是通过 createApp 函数创建一个新的 应用实例：

import { createApp } from 'vue'

const app = createApp({

/\* 根组件选项 \*/

})

## 根组件

我们传入 createApp 的对象实际上是一个组件，每个应用都需要一个“根组件”，其他组件将作为其子组件。如果你使用的是单文件组件，我们可以直接从另一个文件中导入根组件。

import { createApp } from 'vue'

// 从一个单文件组件中导入根组件

import App from './App.vue'

const app = createApp(App)

大多数真实的应用都是由一棵嵌套的、可重用的组件树组成的。例如，待办事项应用程序的组件树可能是这样的：

App (root component)

├─ TodoList

│ └─ TodoItem

│ ├─ TodoDeleteButton

│ └─ TodoEditButton

└─ TodoFooter

├─ TodoClearButton

└─ TodoStatistics

## 挂载应用

应用实例必须在调用了 .mount() 方法后才会渲染出来。该方法接收一个“容器”参数，可以是一个实际的 DOM 元素或是一个 CSS 选择器字符串：

<div id="app"></div>

app.mount('#app')

应用根组件的内容将会被渲染在容器元素里面。容器元素自己将不会被视为应用的一部分。.mount() 方法应该始终在整个应用配置和资源注册完成后被调用。不同于其他资源注册方法，它的返回值是根组件实例而非应用实例。

## DOM 中的根组件模板

当在未采用构建流程的情况下使用 Vue 时，我们可以在挂载容器中直接书写根组件模板：

<div id="app">

<button @click="count++">{{ count }}</button>

</div>

import { createApp } from 'vue'

const app = createApp({

data() {

return {

count: 0

}

}

})

app.mount('#app')

当根组件没有设置 template 选项时，Vue 将自动使用容器的 innerHTML 作为模板。

## 应用配置

应用实例会暴露一个 .config 对象允许我们配置一些应用级的选项，例如定义一个应用级的错误处理器，它将捕获所有由子组件上抛而未被处理的错误：

app.config.errorHandler = (err) => {

/\* 处理错误 \*/

}

应用实例还提供了一些方法来注册应用范围内可用的资源，例如注册一个组件：

app.component('TodoDeleteButton', TodoDeleteButton)

这使得 TodoDeleteButton 在应用的任何地方都是可用的。

## 多个应用实例

你不必再受限于一个页面只能拥有一个应用实例。createApp API 允许多个 Vue 应用共存于同一个页面上，而且每个应用都拥有自己的用于配置和全局资源的作用域。

const app1 = createApp({

/\* ... \*/

})

app1.mount('#container-1')

const app2 = createApp({

/\* ... \*/

})

app2.mount('#container-2')

# 模板语法

Vue 使用一种基于 HTML 的模板语法，使我们能够声明式地将其组件实例的数据绑定到呈现的 DOM 上。所有的 Vue 模板都是语法上合法的 HTML，可以被符合规范的浏览器和 HTML 解析器解析。

在底层机制中，Vue 会将模板编译成高度优化的 JavaScript 代码。结合响应式系统，当应用状态变更时，Vue 能够智能地推导出需要重新渲染的组件的最少数量，并应用最少的 DOM 操作。

## 文本插值

最基本的数据绑定形式是文本插值，它使用的是“Mustache”语法 (即双大括号)：

<span>Message: {{ msg }}</span>

双大括号标签会被替换为相应组件实例中 msg property 的值。同时每次 msg property 更改时它也会同步更新。

## 原始 HTML

双大括号将会将数据插值为纯文本，而不是 HTML。若想插入 HTML，你需要使用 v-html 指令：

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

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

v-html attribute 被称为一个指令。指令由 v- 作为前缀，表明它们是一些由 Vue 提供的特殊 attribuite, 它们将为渲染的 DOM 应用特殊的响应式行为。这里做的事情简单来说就是：在当前组件实例上，将此元素的 innerHTML 与 rawHtml property 保持同步。

span 的内容将会被替换为 rawHtml property 的值，插值为纯 HTML——数据绑定将会被忽略。注意，你不能使用 v-html 来拼接组合模板，因为 Vue 不是一个基于字符串的模板引擎。相反，组件更适合作为 UI 重用和组合的基本单元。

## Attribute 绑定

双大括号不能在 HTML attributes 中使用。相应的，应该使用 v-bind 指令：

<div v-bind:id="dynamicId"></div>

v-bind 指令指示 Vue 将元素的 id attribute 与组件的 dynamicId property 保持一致。如果绑定的值是 null 或者 undefined，那么该 attribute 将会从渲染的元素上移除。

### 简写

因为 v-bind 非常常用，我们提供了特定的简写语法：

<div :id="dynamicId"></div>

开头为 : 的 attribute 可能和一般的 HTML attribute 看起来不太一样，但它的确是合法的 attribute 名称字符，并且所有支持 Vue 的浏览器都能正确解析它。此外，他们不会出现在最终渲染的标签中。

### 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。disabled 就是最常见的例子之一。

v-bind 在这种场景下的行为略有不同：

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

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

### 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：

data() {

return {

objectOfAttrs: {

id: 'container',

class: 'wrapper'

}

}

}

通过不带参数的 v-bind，你可以将它们绑定到单个元素上：

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

## 使用 JavaScript 表达式

至此，我们仅在模板中绑定了一些简单的 property 键。但是 Vue 实际上在所有的数据绑定中都支持完整的 JavaScript 表达式：

{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div :id="`list-${id}`"></div>

这些表达式都会被作为 JavaScript ，以组件为作用域解析执行。在 Vue 模板内，JavaScript 表达式可以被使用在如下场景上：

* 在文本插值中 (双大括号)
* 在任何 Vue 指令 (以 v- 开头的特殊 attribute) attribute 的值中

### 仅支持表达式

每个绑定仅支持单一表达式，所以下面的例子都是无效的：

<!-- 这是一个语句，而非表达式 -->

{{ var a = 1 }}

<!-- 条件控制同样不会工作，请使用三元表达式 -->

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

### 调用函数

可以在绑定的表达式中使用一个组件暴露的方法：

<span :title="toTitleDate(date)">

{{ formatDate(date) }}

</span>

绑定在表达式中的方法在组件每次更新时都会被重新调用，因此不应该产生任何副作用，比如改变数据或触发异步操作。

### 受限的全局访问

模板中的表达式将被沙盒化，仅能够访问到有限的全局对象列表。该列表中会暴露常用的内置全局对象，比如 Math 和 Date。

没有显式包含在列表中的全局对象将不能在模板内表达式中访问，例如用户附加在 window 上的 property。然而，你也可以自行在 app.config.globalProperties 上显式地添加他们，供所有的 Vue 表达式使用。

## 指令

指令是带有 v- 前缀的特殊 attribute。Vue 提供了许多内置指令，包括上面我们所介绍的 v-bind 和 v-html。指令 attribute 的期望值为一个 JavaScript 表达式 (之后要讨论到的 v-for、v-on 和 v-slot 将会是例外)。使用指令是为了在其表达式值变化时响应式地对 DOM 应用更新。以 v-if 为例：

<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。

### 参数

某些指令会需要一个“参数”，在指令名后通过一个冒号隔开做标识。例如用 v-bind 指令来响应式地更新一个 HTML attribute：

<a v-bind:href="url"> ... </a>

<!-- 简写 -->

<a :href="url"> ... </a>

这里 href 就是一个参数，它告诉 v-bind 指令将表达式 url 的值绑定到元素的 href attribute 上。在简写中，参数前的一切 (例如 v-bind:) 都会被缩略为一个 : 字符。另一个例子是 v-on 指令，它将监听 DOM 事件：

<a v-on:click="doSomething"> ... </a>

<!-- 简写 -->

<a @click="doSomething"> ... </a>

这里的参数是要监听的事件名称：click。v-on 也是少部分含有简写的指令之一，简写字符为 @。

### 动态参数

同样在指令参数上也可以使用一个 JavaScript 表达式，需要包含在一对方括号内：

<!--

注意，参数表达式有一些约束，

参见下面“动态参数值的限制”与“动态参数语法的限制”章节的解释

-->

<a v-bind:[attributeName]="url"> ... </a>

<!-- 简写 -->

<a :[attributeName]="url"> ... </a>

这里的 attributeName 会作为一个 JavaScript 表达式被动态执行，计算得到的值会被用作最终的参数。举个例子，如果你的组件实例有一个数据 property attributeName，其值为 "href"，那么这个绑定就等价于 v-bind:href。相似地，你还可以将一个函数绑定到动态的事件名称上：

<a v-on:[eventName]="doSomething"> ... </a>

<!-- 简写 -->

<a @[eventName]="doSomething">

动态参限制：

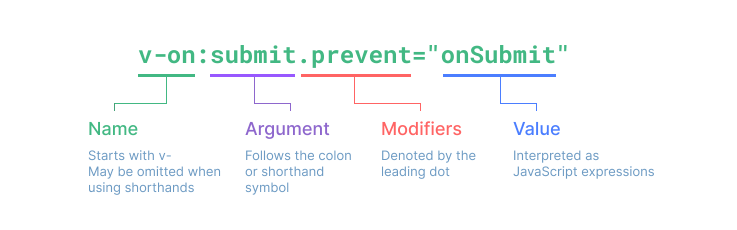
* 动态参数期望结果为一个字符串，或者是 null。特殊值 null 意为显式移除该绑定。任何其他非字符串的值都将触发一个警告。
* 动态参数表达式因为某些字符的缘故有一些语法限制，比如空格和引号，在 HTML attribute 名称中都是不合法的。
* 当使用 DOM 内嵌模板 (直接写在 HTML 文件里的模板) 时，我们需要避免在名称中使用大写字母，因为浏览器会强制将其转换为小写

### 修饰符

修饰符是以点开头的特殊后缀，表明指令需要以一些特殊的方式被绑定。例如 .prevent 修饰符会告知 v-on 指令对触发的事件调用 event.preventDefault()：

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

完整的指令语法：



# 响应式基础（选项式）

## 声明响应式状态

选用选项式 API 时，会用 data 选项来声明组件的响应式状态。此选项的值应为返回一个对象的函数。Vue 将在创建新组件实例的时候调用此函数，并将函数返回的对象封装到其响应式系统中。此对象的任何顶层 property 都被代理到组件实例 (即方法和生命周期钩子中的 this) 上。

export default {

data() {

return {

count: 1

}

},

// `mounted` 是生命周期钩子，之后我们会讲到

mounted() {

// `this` 指向当前组件实例

console.log(this.count) // => 1

// 数据属性也可以被更改

this.count = 2

}

}

## 响应式代理 vs. 原始值

在 Vue 3 中，数据是基于 JavaScript Proxy（代理） 实现响应式的。使用过 Vue 2 的用户可能需要注意下面这样的边界情况：

export default {

data() {

return {

someObject: {}

}

},

mounted() {

const newObject = {}

this.someObject = newObject

console.log(newObject === this.someObject) // false

}

}

当你在赋值后再访问 this.someObject，此值已经是原来的 newObject 的一个响应式代理。这与 Vue 2 中原始的 newObject 不会变为响应式完全不同：请确保始终通过 this 来访问响应式状态。

## 声明方法

为组件添加方法，我们需要用到 methods 选项。它应该是一个包含所有方法的对象：

export default {

data() {

return {

count: 0

}

},

methods: {

increment() {

this.count++

}

},

mounted() {

// 在其他方法或是生命周期中也可以调用方法

this.increment()

}

}

Vue 自动为 methods 中的方法绑定了永远指向组件实例的 this。这确保了方法在作为事件监听器或回调函数时始终保持正确的 this。你不应该在定义 methods 时使用箭头函数，因为这会阻止 Vue 的自动绑定。

export default {

methods: {

increment: () => {

// 反例：无法访问此处的 `this`!

}

}

}

### DOM 更新时机

当你更改响应式状态后，DOM 也会自动更新。然而，你得注意 DOM 的更新并不是同步的。相反，Vue 将缓冲它们直到更新周期的 “下个时机” 以确保无论你进行了多少次声明更改，每个组件都只需要更新一次。若要等待一个状态改变后的 DOM 更新完成，你可以使用 nextTick() 这个全局 API：

import { nextTick } from 'vue'

export default {

methods: {

increment() {

this.count++

nextTick(() => {

// 访问更新后的 DOM

})

}

}

}

### 深层响应性

在 Vue 中，状态都是默认深层响应式的。这意味着即使在更改深层次的对象或数组，你的改动也能被检测到。

export default {

data() {

return {

obj: {

nested: { count: 0 },

arr: ['foo', 'bar']

}

}

},

methods: {

mutateDeeply() {

// 以下都会按照期望工作

this.obj.nested.count++

this.obj.arr.push('baz')

}

}

}

## 有状态方法

某些情况下，我们可能需要动态地创建一个方法函数，比如创建一个预置防抖的事件处理器：

import { debounce } from 'lodash-es'

export default {

methods: {

// 使用 Lodash 的防抖函数

click: debounce(function () {

// ... 对点击的响应 ...

}, 500)

}

}

不过这种方法对于被重用的组件来说是有问题的，因为这个预置防抖的函数是 有状态的：它在运行时维护着一个内部状态。如果多个组件实例都共享这同一个预置防抖的函数，那么它们之间将会互相影响。要保持每个组件实例的防抖函数都彼此独立，我们可以改为在 created 生命周期钩子中创建这个预置防抖的函数：

export default {

created() {

// 每个实例都有了自己的预置防抖的处理函数

this.debouncedClick = \_.debounce(this.click, 500)

},

unmounted() {

// 最好是在组件卸载时

// 清除掉防抖计时器

this.debouncedClick.cancel()

},

methods: {

click() {

// ... 对点击的响应 ...

}

}

}

# 响应式基础（组合式）

## 声明响应式状态

我们可以使用 reactive() 函数创建一个响应式对象或数组：

import { reactive } from 'vue'

const state = reactive({ count: 0 })

响应式对象其实是 JavaScript Proxy，其行为表现与一般对象相似。不同之处在于 Vue 能够跟踪对响应式对象 property 的访问与更改操作。要在组件模板中使用响应式状态，请在 setup() 函数中定义并返回。

import { reactive } from 'vue'

export default {

// `setup` 是一个专门用于组合式 API 的特殊钩子

setup() {

const state = reactive({ count: 0 })

// 暴露 state 到模板

return {

state

}

}

}

<div>{{ state.count }}</div>

相似地，我们也可以在这个作用域下定义可更改响应式 state 的函数，并作为一个方法与 state 一起暴露出去：

import { reactive } from 'vue'

export default {

setup() {

const state = reactive({ count: 0 })

function increment() {

state.count++

}

// 不要忘记同时暴露 increment 函数

return {

state,

increment

}

}

}

暴露的方法通常会被用作事件监听器：

<button @click="increment">

{{ state.count }}

</button>

### <script setup>

在 setup() 函数中手动暴露状态和方法可能非常繁琐。幸运的是，你可以通过使用构建工具来简化该操作。当使用单文件组件（SFC）时，我们可以使用 <script setup> 来简化大量样板代码。

<script setup>

import { reactive } from 'vue'

const state = reactive({ count: 0 })

function increment() {

state.count++

}

</script>

<template>

<button @click="increment">

{{ state.count }}

</button>

</template>

<script setup> 中的顶层的导入和变量声明可在同一组件的模板中自动使用。

### DOM 更新时机

当你更改响应式状态后，DOM 也会自动更新。然而，你得注意 DOM 的更新并不是同步的。相反，Vue 将缓冲它们直到更新周期的 “下个时机” 以确保无论你进行了多少次声明更改，每个组件都只需要更新一次。若要等待一个状态改变后的 DOM 更新完成，你可以使用 nextTick() 这个全局 API：

import { nextTick } from 'vue'

function increment() {

state.count++

nextTick(() => {

// 访问更新后的 DOM

})

}

### 深层响应性

在 Vue 中，状态都是默认深层响应式的。这意味着即使在更改深层次的对象或数组，你的改动也能被检测到。

import { reactive } from 'vue'

const obj = reactive({

nested: { count: 0 },

arr: ['foo', 'bar']

})

function mutateDeeply() {

// 以下都会按照期望工作

obj.nested.count++

obj.arr.push('baz')

}

你也可以直接创建一个浅层响应式对象。它们仅在顶层具有响应性，一般仅在某些特殊场景中需要。

### 响应式代理 vs. 原始对象

值得注意的是，reactive() 返回的是一个原始对象的 Proxy，它和原始对象是不相等的：

const raw = {}

const proxy = reactive(raw)

// 代理和原始对象不是全等的

console.log(proxy === raw) // false

只有代理是响应式的，更改原始对象不会触发更新。因此，使用 Vue 的响应式系统的最佳实践是 仅使用你声明对象的代理版本。为保证访问代理的一致性，对同一个对象调用 reactive() 会总是返回同样的代理，而对一个已存在代理调用 reactive() 也是返回同样的代理：

// 在同一个对象上调用 reactive() 会返回相同的代理

console.log(reactive(raw) === proxy) // true

// 在一个代理上调用 reactive() 会返回它自己

console.log(reactive(proxy) === proxy) // true

这个规则对嵌套对象也适用。依靠深层响应性，响应式对象内的嵌套对象依然是代理：

const proxy = reactive({})

const raw = {}

proxy.nested = raw

console.log(proxy.nested === raw) // false

### reactive() 的局限性

reactive() API 有两条限制：

* 仅对对象类型有效（对象、数组和 Map、Set 这样的集合类型），而对 string、number 和 boolean 这样的 原始类型 无效。
* 因为 Vue 的响应式系统是通过 property 访问进行追踪的，因此我们必须始终保持对该响应式对象的相同引用。这意味着我们不可以随意地“替换”一个响应式对象，因为这将导致对初始引用的响应性连接丢失。

同时这也意味着当我们将响应式对象的 property 赋值或解构至本地变量时，或是将该 property 传入一个函数时，我们会失去响应性：

const state = reactive({ count: 0 })

// n 是一个局部变量，同 state.count

// 失去响应性连接

let n = state.count

// 不影响原始的 state

n++

// count 也和 state.count 失去了响应性连接

let { count } = state

// 不会影响原始的 state

count++

// 该函数接收一个普通数字，并且

// 将无法跟踪 state.count 的变化

callSomeFunction(state.count)

## ref() 定义响应式变量

为了解决 reactive() 带来的限制，Vue 也提供了一个 ref() 方法来允许我们创建可以使用任何值类型的响应式 ref：

import { ref } from 'vue'

const count = ref(0)

ref() 从参数中获取到值，将其包装为一个带 .value property 的 ref 对象：

const count = ref(0)

console.log(count) // { value: 0 }

console.log(count.value) // 0

count.value++

console.log(count.value) // 1

和响应式对象的 property 类似，ref 的 .value property 也是响应式的。同时，当值为对象类型时，会用 reactive() 自动转换它的 .value。一个包含对象类型值的 ref 可以响应式地替换整个对象：

const objectRef = ref({ count: 0 })

// 这是响应式的替换

objectRef.value = { count: 1 }

ref 被传递给函数或是从一般对象上被解构时，不会丢失响应性：

const obj = {

foo: ref(1),

bar: ref(2)

}

// 该函数接收一个 ref

// 需要通过 .value 取值

// 但它会保持响应性

callSomeFunction(obj.foo)

// 仍然是响应式的

const { foo, bar } = obj

一言以蔽之，ref() 使我们能创造一种任意值的 “引用” 并能够不丢失响应性地随意传递。这个功能非常重要，因为它经常用于将逻辑提取到 组合函数 中。

### ref 在模板中的解包

当 ref 在模板中作为顶层 property 被访问时，它们会被自动“解包”，所以不需要使用 .value。下面是之前的计数器例子，用 ref() 代替：

<script setup>

import { ref } from 'vue'

const count = ref(0)

function increment() {

count.value++

}

</script>

<template>

<button @click="increment">

{{ count }} <!-- 无需 .value -->

</button>

</template>

请注意，仅当 ref 是模板渲染上下文的顶层 property 时才适用自动“解包”。 例如， foo 是顶层 property，但 object.foo 不是。所以我们给出以下 object：

const object = { foo: ref(1) }

下面的表达式将不会像预期的那样工作：

{{ object.foo + 1 }}

渲染的结果会是一个 [object Object]，因为 object.foo 是一个 ref 对象。我们可以通过让 foo 成为顶级 property 来解决这个问题：

const { foo } = object

{{ foo + 1 }}

现在渲染结果将是 2。

需要注意的是，如果一个 ref 是文本插值（即一个 {{ }} 符号）计算的最终值，它也将被解包。因此下面的渲染结果将为 1：

{{ object.foo }}

这只是文本插值的一个方便功能，相当于 {{ object.foo.value }}。

### ref 在响应式对象中的解包

当一个 ref 作为一个响应式对象的 property 被访问或更改时，它会自动解包，因此会表现得和一般的 property 一样：

const count = ref(0)

const state = reactive({

count

})

console.log(state.count) // 0

state.count = 1

console.log(count.value) // 1

如果将一个新的 ref 赋值给一个关联了已有 ref 的 property，那么它会替换掉旧的 ref：

const otherCount = ref(2)

state.count = otherCount

console.log(state.count) // 2

// 原始 ref 现在已经和 state.count 失去联系

console.log(count.value) // 1

只有当嵌套在一个深层响应式对象内时，才会发生 ref 解包。

### 数组和集合类型的 ref 解包

不像响应式对象，当 ref 作为响应式数组或像 Map 这种原生集合类型的元素被访问时，不会进行解包。

const books = reactive([ref('Vue 3 Guide')])

// 这里需要 .value

console.log(books[0].value)

const map = reactive(new Map([['count', ref(0)]]))

// 这里需要 .value

console.log(map.get('count').value)

## 响应性语法糖

不得不对 ref 使用 .value 是一个受限于 JavaScript 语言限制的缺点。然而，通过编译时转换，我们可以在适当的位置自动添加 .value 来提升开发体验。Vue 提供了一种编译时转换，使得可以像这样书写之前的“计数器”示例：

<script setup>

let count = $ref(0)

function increment() {

// 无需 .value

count++

}

</script>

<template>

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

</template>

# 计算属性

## 基础示例

模板中的表达式虽然方便，但也只能用来做简单的操作。如果在模板中写太多逻辑，会让使其变得臃肿，难以维护。比如说，我们有这样一个包含嵌套数组的对象：

export default {

data() {

return {

author: {

name: 'John Doe',

books: [

'Vue 2 - Advanced Guide',

'Vue 3 - Basic Guide',

'Vue 4 - The Mystery'

]

}

}

}

}

我们想根据 author 是否已有一些书籍来展示不同的信息：

<p>Has published books:</p>

<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>

这里的模板看起来有些复杂。我们必须认真看好一会儿才能明白它的计算依赖于 author.books。更重要的是，如果在模板中需要不止一次这样的计算，我们可能不想写重复的代码。因此我们推荐使用计算属性来描述依赖响应式状态的复杂逻辑。这是重构后的示例：

export default {

data() {

return {

author: {

name: 'John Doe',

books: [

'Vue 2 - Advanced Guide',

'Vue 3 - Basic Guide',

'Vue 4 - The Mystery'

]

}

}

},

computed: {

// 一个计算属性的 getter

publishedBooksMessage() {

// `this` 指向当前组件实例

return this.author.books.length > 0 ? 'Yes' : 'No'

}

}

}

<p>Has published books:</p>

<span>{{ publishedBooksMessage }}</span>

在这里定义了一个计算属性 publishedBooksMessage。更改此应用的 data 中 books 数组的值后，可以看到 publishedBooksMessage 也会随之改变。在模板中使用计算属性的方式和一般的 property 并无二致。Vue 会检测到 this.publishedBooksMessage 依赖于 this.author.books，所以当 this.author.books 改变时，任何依赖于 this.publishedBooksMessage 的绑定都将同时更新。

## 计算属性缓存 vs 方法

在表达式中像这样调用一个函数也会获得和计算属性相同的结果：

<p>{{ calculateBooksMessage() }}</p>

// 组件中

methods: {

calculateBooksMessage() {

return this.author.books.length > 0 ? 'Yes' : 'No'

}

}

若我们将同样的函数定义为一个方法而不是计算属性，两种方式在结果上确实是完全相同的，然而，不同之处在于计算属性值会基于其响应式依赖被缓存。一个计算属性仅会在其响应式依赖更新时才重新计算。这意味着只要 author.books 不改变，无论多少次访问 publishedBooksMessage 都会立即返回先前的计算结果，而不用重复执行 getter 函数。

这也意味着下面的计算属性永远不会更新，因为 Date.now() 并不是一个响应式依赖：

computed: {

now() {

return Date.now()

}

}

相比之下，方法调用总是会在重渲染发生时再次执行函数。

## 可写计算属性

计算属性默认仅能通过计算函数得出结果。当你尝试修改一个计算属性时，你会收到一个运行时警告。只在某些特殊场景中你可能才需要用到“可写”的属性，你可以通过同时提供 getter 和 setter 来创建：

export default {

data() {

return {

firstName: 'John',

lastName: 'Doe'

}

},

computed: {

fullName: {

// getter

get() {

return this.firstName + ' ' + this.lastName

},

// setter

set(newValue) {

// 注意：我们这里使用的是解构赋值语法

[this.firstName, this.lastName] = newValue.split(' ')

}

}

}

}

现在当你再运行 this.fullName = 'John Doe' 时，setter 会被调用而 this.firstName 和 this.lastName 会随之更新。

# Class 与 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。因为它们都是 attribute，我们可以使用 v-bind 来做这件事：我们只需要通过表达式计算出一个字符串作为最终结果即可。然而频繁地连接字符串让人很闹心，也很容易出错。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，表达式的结果还可以是对象或数组。

## 绑定 HTML class

我们可以给 :class (v-bind:class 的缩写) 传递一个对象来动态切换 class：

<div :class="{ active: isActive }"></div>

上面的语法表示 active 是否存在取决于数据属性 isActive 的真假值。你可以在对象中写多个字段来操作多个 class。此外，:class 指令也可以和一般的 class attribute 共存。所以可以有下面这样的状态：

data() {

return {

isActive: true,

hasError: false

}

}

使用的模板如下：

<div

class="static"

:class="{ active: isActive, 'text-danger': hasError }"

></div>

它将会被渲染成：

<div class="static active"></div>

这也会渲染出相同的结果。我们也可以绑定一个返回对象的计算属性。这才是一个通用且好用的实践。

data() {

return {

isActive: true,

error: null

}

},

computed: {

classObject() {

return {

active: this.isActive && !this.error,

'text-danger': this.error && this.error.type === 'fatal'

}

}

}

<div :class="classObject"></div>

### 绑定数组

我们可以给 :class 绑定一个数组以应用一系列 CSS class：

data() {

return {

activeClass: 'active',

errorClass: 'text-danger'

}

}

<div :class="[activeClass, errorClass]"></div>

渲染的结果是：

<div class="active text-danger"></div>

如果你也想在数组中按条件触发某个 class，你可以使用三元表达式：

<div :class="[isActive ? activeClass : '', errorClass]"></div>

errorClass 会一直存在，但 activeClass 只会在 isActive 为真时才存在。

然而，这可能在有多个依赖条件的 class 时会有些冗长。因此也可以在数组中使用对象语法：

<div :class="[{ active: isActive }, errorClass]"></div>

### 和组件配合

对于只有一个根元素的组件，当你使用了 class attribute 时，这些 class 会被添加到根元素上，并与该元素上已有的 class 合并。

<!-- 子组件模板 -->

<p class="foo bar">Hi!</p>

<!-- 在使用组件时 -->

<my-component class="baz boo"></my-component>

渲染出的 HTML 为：

<p class="foo bar baz boo">Hi</p>

如果你的组件有多个根元素，你将需要指定哪个根元素来接收这个 class。你可以通过组件的 $attrs property 来实现指定：

<!-- my-component 模板使用 $attrs 时 -->

<p :class="$attrs.class">Hi!</p>

<span>This is a child component</span>

<my-component class="baz"></my-component>

这将被渲染为：

<p class="baz">Hi!</p>

<span>This is a child component</span>

## 绑定内联样式

### 绑定对象

:style 支持绑定 JavaScript 对象值，对应的是 HTML 元素的 style 属性：

data() {

return {

activeColor: 'red',

fontSize: 30

}

}

<div :style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>

推荐使用 camelCase，但 :style 也支持 kebab-cased 形式的 CSS 属性 key (对应其 CSS 中的实际名称)，举个例子：

<div :style="{ 'font-size': fontSize + 'px' }"></div>

直接绑定一个样式对象通常是一个好主意，这样可以使模板更加简洁：

data() {

return {

styleObject: {

color: 'red',

fontSize: '13px'

}

}

}

<div :style="styleObject"></div>

### 绑定数组

我们还可以给 :style 绑定一个包含多个样式对象的数组。这些对象会被合并和应用到同一元素上：

<div :style="[baseStyles, overridingStyles]"></div>

* 自动前缀：当你在 :style 中使用了需要浏览器特殊前缀的 CSS 属性时，Vue 会自动为他们加上相应的前缀。Vue 是在运行时检查该属性是否支持在当前浏览器中使用。如果浏览器不支持某个属性，那么将测试加上各个浏览器特殊前缀，以找到哪一个是被支持的。
* 样式多值：你可以对一个样式属性提供多个 (不同前缀的) 值，举个例子：

<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>

数组仅会渲染浏览器支持的最后一个值。在这个示例中，在支持不需要特别加前缀的浏览器中都会渲染为 display: flex 的弹性盒子。

# 条件渲染

## v-if

v-if 指令被用于按条件渲染一个区块。这个区块只会在指令的表达式为真时才被渲染。

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

## v-else

你也可以使用 v-else 为 v-if 添加一个“else 区块”。

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

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

<h1 v-else>Oh no 😢</h1>

一个 v-else 元素必须跟在一个 v-if 或者 v-else-if 元素后面，否则将不会识别它。

## 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：

<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>

和 v-else 相似，一个使用 v-else-if 的元素必须紧跟在一个 v-if 或一个 v-else-if 元素后面。

## <template> 上的 v-if

因为 v-if 是一个指令，他必须依附于某个元素。但如果我们想要切换不止一个元素呢？在这种情况下我们可以在一个 <template> 元素上使用 v-if，这只是一个不可见的包装器元素，最后渲染的结果并不会包含这个 <template> 元素。

<template v-if="ok">

<h1>Title</h1>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</template>

v-else 和 v-else-if 也可以在 <template> 上使用。

## 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。v-show 不支持在 <template> 元素上使用，也没有 v-else 来配合。

## v-if vs v-show

* v-if 是“真实的”按条件渲染，因为它确保了条件区块内的事件监听器和子组件都会在切换时被销毁与重建。
* v-if 也是懒加载的：如果在初次渲染时条件值为 false，则不会做任何事。条件区块会直到条件首次变为 true 时才渲染。
* 相比之下，v-show 简单许多，元素无论初始条件如何，始终会被渲染，仅作 CSS class 的切换。
* 总的来说，v-if 在首次渲染时的切换成本比 v-show 更高。因此当你需要非常频繁切换时 v-show 会更好，而运行时不常改变的时候 v-if 会更合适。

## v-if 和 v-for

当 v-if 和 v-for 同时存在于一个元素上的时候，v-if 会首先被执行。

# 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 指令需要一种特殊的语法形式 item in items，其中 items 是源数据的数组，而 item 是迭代项的别名：

data() {

return {

items: [{ message: 'Foo' }, { message: 'Bar' }]

}

}

<li v-for="item in items">

{{ item.message }}

</li>

在 v-for 块中可以完整地访问父作用域内的 property。v-for 也支持使用可选的第二个参数表示当前项的位置索引。

data() {

return {

parentMessage: 'Parent',

items: [{ message: 'Foo' }, { message: 'Bar' }]

}

}

<li v-for="(item, index) in items">

{{ parentMessage }} - {{ index }} - {{ item.message }}

</li>

注意 v-for 是如何对应 forEach 回调的函数签名的。实际上，你也可以在定义 v-for 的变量别名时使用解构，和解构函数参数类似：

<li v-for="{ message } in items">

{{ message }}

</li>

<!-- 有 index 索引时 -->

<li v-for="({ message }, index) in items">

{{ message }} {{ index }}

</li>

对于多层嵌套的 v-for，作用域的工作方式和函数的作用域很类似。每个 v-for 作用域都可以访问到父级作用域：

<li v-for="item in items">

<span v-for="childItem in item.children">

{{ item.message }} {{ childItem }}

</span>

</li>

你也可以使用 of 作为分隔符来替代 in，这也和 JavaScript 的迭代器语法非常相似：

<div v-for="item of items"></div>

## v-for与对象

你也可以使用 v-for 来遍历一个对象的所有属性。

data() {

return {

myObject: {

title: 'How to do lists in Vue',

author: 'Jane Doe',

publishedAt: '2016-04-10'

}

}

}

<ul>

<li v-for="value in myObject">

{{ value }}

</li>

</ul>

你也可以提供第二个参数表示属性名 (例如 key)：

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

{{ key }}: {{ value }}

</li>

第三个参数表示位置索引：

<li v-for="(value, key, index) in myObject">

{{ index }}. {{ key }}: {{ value }}

</li>

注意：当遍历一个对象时，顺序是依据 Object.keys() 的枚举顺序，由于不同的 JavaScript 引擎可能会有不同的实现，所以顺序可能会不一致。

## 在 v-for 里使用值范围

可以直接传给 v-for 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。

## 通过 key 管理状态

Vue 默认按照“就地更新”的策略来更新通过 v-for 渲染的元素列表。当数据项的顺序改变时，Vue 不会随之移动 DOM 元素的顺序，而是就地更新每个元素，确保它们在原本指定的索引位置上渲染。默认模式是高效的，但只适用于列表渲染输出不依赖子组件状态或者临时 DOM 状态 (例如表单输入值)。

为了给 Vue 一个提示，以便它可以跟踪每个节点的标识，从而重用和重新排序现有的元素，你需要为每个项目提供一个唯一的 key attribute：

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

<!-- 内容 -->

</div>

当你使用 <template v-for> 时，key 应该被放置在这个 <template> 容器上：

<template v-for="todo in todos" :key="todo.name">

<li>{{ todo.name }}</li>

</template>

推荐在任何可行的时候为 v-for 提供一个 key attribute，除非所迭代的 DOM 内容非常简单 (例如：不包含组件或有状态的 DOM 元素)，或者有意依赖默认行为来获得性能增益。key 绑定的值期望是一个基础类型的值，例如字符串或 number 类型。不要用对象作为 v-for 的 key。

## 组件上使用 v-for

可以直接在组件上使用 v-for，和其他任何一般的元素没有区别 (别忘记提供一个 key)：

<my-component v-for="item in items" :key="item.id"></my-component>

但是，这不会自动将任何数据传递给组件，因为组件有自己独立的作用域。为了将迭代后的数据传递到组件中，我们还是应该使用 prop：

<my-component

v-for="(item, index) in items"

:item="item"

:index="index"

:key="item.id"

></my-component>

不自动将 item 注入组件的原因是，这会使组件与 v-for 的工作方式紧密耦合。明确其数据的来源可以使组件在其他情况下重用。

## 数组变化侦测

Vue 包装了一批侦听数组的变更方法，以至于这些方法可以触发视图更新。被包装的变更方法如下：

push()

pop()

shift()

unshift()

splice()

sort()

reverse()

变更方法，就是会对调用它们的原数组进行变更。相对地，也有一些非变更方法，例如 filter()，concat() 和 slice()，这些都不会更改原数组，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：

this.items = this.items.filter((item) => item.message.match(/Foo/))

这将导致 Vue 丢弃现有的 DOM 并重新渲染整个列表，但是，情况并非如此。Vue 实现了一些巧妙的方法来最大化对 DOM 元素的重用，因此用另一个包含部分重叠对象的数组来做替换，仍会是一种非常高效的操作。

## 过滤或排序后的结果

希望显示数组经过过滤或排序后的内容，而不实际变更或重置原始数据。在这种情况下，你可以创建返回已过滤或已排序数组的计算属性。

data() {

return {

numbers: [1, 2, 3, 4, 5]

}

},

computed: {

evenNumbers() {

return this.numbers.filter(n => n % 2 === 0)

}

}

<li v-for="n in evenNumbers">{{ n }}</li>

在计算属性不可行的情况下 (例如在多层嵌套的 v-for 循环中)，你可以使用以下方法：

data() {

return {

sets: [[ 1, 2, 3, 4, 5 ], [6, 7, 8, 9, 10]]

}

},

methods: {

even(numbers) {

return numbers.filter(number => number % 2 === 0)

}

}

<ul v-for="numbers in sets">

<li v-for="n in even(numbers)">{{ n }}</li>

</ul>

事件处理

# 事件处理

## 监听事件

你可以使用 v-on 指令 (简写为 @) 来监听 DOM 事件和运行 JavaScript 代码。用法：v-on:click="methodName" 或 @click="handler"。事件处理器的值可以是：

* 内联事件处理器：事件被触发时执行的内联 JavaScript 语句 (与 onclick 类似)。
* 方法事件处理器：一个组件的属性名、或对某个方法的访问。

## 内联事件处理器

内联事件处理器通常用于简单场景，例如：

data() {

return {

count: 0

}

}

<button @click="count++">Add 1</button>

<p>Count is: {{ count }}</p>

## 方法事件处理器

随着事件处理器的逻辑变得愈发复杂，内联代码方式变得不够灵活。因此 v-on 也可以接受一个方法名或对某个方法的调用。

data() {

return {

name: 'Vue.js'

}

},

methods: {

greet(event) {

// 方法中的 `this` 指向当前活跃的组件实例

alert(`Hello ${this.name}!`)

// `event` 是 DOM 原生事件

if (event) {

alert(event.target.tagName)

}

}

}

<!-- `greet` 是上面定义过的方法名 -->

<button @click="greet">Greet</button>

方法事件处理器会自动接收原生 DOM 事件并触发执行。在上面的例子中，我们能够通过被触发事件的 event.target.tagName 访问到该 DOM 元素。

**方法与内联事件判断**：模板编译器会通过检查 v-on 的值是否是合法的 JavaScript 标识符或属性访问来断定是何种形式的事件处理器。举个例子，foo、foo.bar 和 foo['bar'] 会被视为方法事件处理器，而 foo() 和 count++ 会被视为内联事件处理器。

## 在内联处理器中调用方法

除了直接绑定方法名，你还可以在内联事件处理器中调用方法。这允许我们向方法传入自定义参数以代替原生事件：

methods: {

say(message) {

alert(message)

}

}

<button @click="say('hello')">Say hello</button>

<button @click="say('bye')">Say bye</button>

## 在内联事件处理器中访问事件参数

有时我们需要在内联事件处理器中访问原生 DOM 事件。你可以向该处理器方法传入一个特殊的 $event 变量，或者使用内联箭头函数：

<!-- 使用特殊的 $event 变量 -->

<button @click="warn('Form cannot be submitted yet.', $event)">

Submit

</button>

<!-- 使用内联箭头函数 -->

<button @click="(event) => warn('Form cannot be submitted yet.', event)">

Submit

</button>

methods: {

warn(message, event) {

// 这里可以访问 DOM 原生事件

if (event) {

event.preventDefault()

}

alert(message)

}

}

## 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。修饰符是用点表示的指令后缀。

* .stop
* .prevent
* .self
* .capture
* .once
* .passive

<!-- 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>

使用修饰符时需要注意调用顺序，因为相关代码是以相同的顺序生成的。因此使用 @click.prevent.self 会阻止元素及其子元素的所有点击事件的默认行为而 @click.self.prevent 则只会阻止对元素本身的点击事件的默认行为。

.capture、.once 和 .passive 修饰符与原生 addEventListener 事件相同：

<!-- 添加事件监听器时，使用 `capture` 捕获模式 -->

<!-- 例如：指向内部元素的事件，在被内部元素处理前，先被外部处理 -->

<div @click.capture="doThis">...</div>

<!-- 点击事件最多被触发一次 -->

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

<!-- 滚动事件的默认行为 (scrolling) 将立即发生而非等待 `onScroll` 完成 -->

<!-- 以防其中包含 `event.preventDefault()` -->

<div @scroll.passive="onScroll">...</div>

.passive 修饰符一般用于触摸事件的监听器，可以用来改善移动端设备的滚屏性能。

请勿同时使用.passive和.prevent，因为.prevent 会被忽略并且你的浏览器可能会抛出警告。.passive 是向浏览器表明你不想阻止事件的默认行为。并且如果你这样做，可能在浏览器中收到一个警告。

## 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。

<!-- 仅在 `key` 为 `Enter` 时调用 `vm.submit()` -->

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

你可以直接使用 KeyboardEvent.key 暴露的按键名称作为修饰符，但需要转为 kebab-case 形式。

<input @keyup.page-down="onPageDown" />

在上面的例子中，仅会在 $event.key 为 'PageDown' 时调用事件处理。

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

你可以使用以下系统按键修饰符来触发鼠标或键盘事件监听器，只有当按键被按下时才会触发。

* .ctrl
* .alt
* .shift
* .meta

<!-- Alt + Enter -->

<input @keyup.alt.enter="clear" />

<!-- Ctrl + 点击 -->

<div @click.ctrl="doSomething">Do something</div>

注意，系统按键修饰符和常规按键不同。与 keyup事件一起使用时，该按键必须在事件发出时处于按下状态。换句话说，keyup.ctrl 只会在你仍然按住 ctrl 但松开了另一个键时被触发。若你单独松开 ctrl 键将不会触发。

.exact 修饰符：.exact 修饰符允许控制触发一个事件所需的确定组合的系统按键修饰符。

## 鼠标按键修饰符

* .left
* .right
* .middle

这些修饰符将处理程序限定为由特定鼠标按键触发的事件。

# 表单输入绑定

在前端处理表单时，我们常常需要将表单输入框的内容同步给 JavaScript 中相应的变量。手动连接值绑定和更改事件监听器可能会很麻烦：

<input

:value="text"

@input="event => text = event.target.value">

v-model 指令帮我们简化了这一步骤：

<input v-model="text">

另外，v-model 还可以用于各种不同类型的输入，<textarea>、<select> 元素。它会根据所使用的元素自动扩展到不同的 DOM 属性和事件组合：

* 文本类型的 <input> 和 <textarea> 元素会使用到 value 属性和 input 事件；
* <input type="checkbox"> 和 <input type="radio"> 使用 checked 属性和 change 事件；
* <select> 使用的 value 作为 prop，change 作为事件：
* v-model 会忽略任何表单元素上初始的 value、checked 或 selected attribute。它将始终将当前绑定的 JavaScript 状态视为数据的正确来源。

## 基本用法

### 文本

<p>Message is: {{ message }}</p>

<input v-model="message" placeholder="edit me" />

对于需要使用 IME 的语言 (中文，日文和韩文等)，你会发现 v-model 不会在 IME 输入的组合状态时触发更新。如果你的确想在此时也触发更新，请使用 input 事件监听器和 value 绑定值而不要使用 v-model。

### 多行文本

<span>Multiline message is:</span>

<p style="white-space: pre-line;">{{ message }}</p>

<textarea v-model="message" placeholder="add multiple lines"></textarea>

注意插值表达式在 <textarea> 中将不会工作。请使用 v-model 来替代。

<!-- 错误 -->

<textarea>{{ text }}</textarea>

<!-- 正确 -->

<textarea v-model="text"></textarea>

### 复选框

单一的复选框，绑定的是布尔类型值：

<input type="checkbox" id="checkbox" v-model="checked" />

<label for="checkbox">{{ checked }}</label>

我们还可以将多个复选框绑定到同一个数组或集合的值：

export default {

data() {

return {

checkedNames: []

}

}

}

<div>Checked names: {{ checkedNames }}</div>

<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">

<label for="jack">Jack</label>

<input type="checkbox" id="john" value="John" v-model="checkedNames">

<label for="john">John</label>

<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">

<label for="mike">Mike</label>

### 单选按钮

<div>Picked: {{ picked }}</div>

<input type="radio" id="one" value="One" v-model="picked" />

<label for="one">One</label>

<input type="radio" id="two" value="Two" v-model="picked" />

<label for="two">Two</label>

### 选择器

单个选择器的示例如下：

<div>Selected: {{ selected }}</div>

<select v-model="selected">

<option disabled value="">Please select one</option>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

如果 v-model 表达式的初始值不匹配任何一个选择项，<select> 元素会渲染成一个“未选择”的状态。

## 值绑定

对于单选按钮，复选框和选择器选项，v-model 绑定的值通常是静态的字符串 (或者对复选框是布尔值)：

<!-- `picked` 在被选择时是字符串 "a" -->

<input type="radio" v-model="picked" value="a" />

<!-- `toggle` 只会为 true 或 false -->

<input type="checkbox" v-model="toggle" />

<!-- `selected` 在第一项被选中时为字符串 "abc" -->

<select v-model="selected">

<option value="abc">ABC</option>

</select>

但有时我们可能希望将该值绑定到当前活动实例上的动态属性，那么可以使用 v-bind 来做到。此外使用 v-bind 还使我们可以将选项值绑定为非字符串类型。

## 复选框

<input

type="checkbox"

v-model="toggle"

true-value="yes"

false-value="no" />

true-value 和 false-value 是 Vue 特有的 attributes 且仅会在 v-model 存在时工作。这里 toggle 属性的值会在选中时被设为 'yes'，取消选择时设为 'no'。你同样可以通过 v-bind 将其绑定为其他动态值：

<input

type="checkbox"

v-model="toggle"

:true-value="dynamicTrueValue"

:false-value="dynamicFalseValue" />

true-value 和 false-value attributes 不会影响 value attribute，因为浏览器在表单提交时，并不会包含未选择的复选框。为了保证这两个值 (例如：“yes”和“no”) 的其中之一被表单提交，请使用单选按钮作为替代。

### 单选按钮

<input type="radio" v-model="pick" :value="first" />

<input type="radio" v-model="pick" :value="second" />

pick 会在第一个按钮选中时被设为 first，在第二个按钮选中时被设为 second。

### 选择器选项

<select v-model="selected">

<!-- 内联对象字面量 -->

<option :value="{ number: 123 }">123</option>

</select>

v-model 同样也支持非字符串类型的值绑定！在上面这个例子中，当某个选项被选中，selected 会被设为该对象字面量值 { number: 123 }。

## 修饰符

### .lazy

默认情况下，v-model 会在每次 input 事件后更新数据 (IME composition 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。number 修饰符会在输入框有 type="number" 时自动应用。

### .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 修饰符来管理输入：

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

## 组件上的 v-model

可以使用 Vue 构建具有完全自定义行为的可复用输入组件，并且这些输入组件也可以使用 v-model。

# 生命周期

每个 Vue 组件实例在创建时都需要经历一系列的初始化步骤，比如设置好数据侦听，编译模板，挂载实例到 DOM 以及数据改变时更新 DOM。在此过程中，它也会运行称为生命周期钩子的函数，让开发者有机会在特定阶段添加自己的代码。

## 注册周期钩子

举个例子，onMounted 钩子可以用来在组件完成初始渲染并创建 DOM 节点后运行代码。

<script setup>

import { onMounted } from 'vue'

onMounted(() => {

console.log(`the component is now mounted.`)

})

</script>

还有其他一些钩子，会在实例生命周期的不同阶段被调用，最常用的是 onMounted，onUpdated 和 onUnmounted。当调用 onMounted 时，Vue 会自动将注册的回调函数与当前活动组件实例相关联。这就要求这些钩子在组件设置时同步注册。例如请不要这样做：

setTimeout(() => {

onMounted(() => {

// 这将不会正常工作

})

}, 100)

请注意，这并不意味着对 onMounted 的调用必须放在 setup() 或 <script setup> 内的词法环境下。onMounted() 也可以在一个外部函数中调用，只要调用栈是同步的，且最终起源自 setup()。

## 生命周期图示

下面是实例生命周期的图表。

[生命周期](生命周期.png)

# 侦听器

## 基本示例

计算属性允许我们声明性地计算推导值。然而，在有些情况下，为了应对一些状态的变化，我们需要运行些“副作用”：例如更改 DOM，或者根据异步操作的结果，去修改另一处的状态。在组合式 API 中，我们可以使用 watch 函数在每次响应式状态发生变化时触发回调函数：

<script setup>

import { ref, watch } from 'vue'

const question = ref('')

const answer = ref('Questions usually contain a question mark. ;-)')

// 可以直接侦听一个 ref

watch(question, async (newQuestion, oldQuestion) => {

if (newQuestion.indexOf('?') > -1) {

answer.value = 'Thinking...'

try {

const res = await fetch('https://yesno.wtf/api')

answer.value = (await res.json()).answer

} catch (error) {

answer.value = 'Error! Could not reach the API. ' + error

}

}

})

</script>

<template>

<p>

Ask a yes/no question:

<input v-model="question" />

</p>

<p>{{ answer }}</p>

</template>

watch 的第一个参数可以是不同形式的“来源”：它可以是一个 ref (包括计算属性)、一个响应式对象、一个 getter 函数、或多个来源组成的数组：

const x = ref(0)

const y = ref(0)

// 单个 ref

watch(x, (newX) => {

console.log(`x is ${newX}`)

})

// getter 函数

watch(

() => x.value + y.value,

(sum) => {

console.log(`sum of x + y is: ${sum}`)

}

)

// 多个来源组成的数组

watch([x, () => y.value], ([newX, newY]) => {

console.log(`x is ${newX} and y is ${newY}`)

})

注意，你不能侦听响应式对象的 property，例如:

const obj = reactive({ count: 0 })

// 这不起作用，因为你向 watch() 传入了一个 number

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

console.log(`count is: ${count}`)

})

而是用 getter 函数：

// 提供一个 getter 函数

watch(

() => obj.count,

(count) => {

console.log(`count is: ${count}`)

}

)

## 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：

const obj = reactive({ count: 0 })

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

// 在嵌套的 property 变更时触发

// 注意：`newValue` 此处和 `oldValue` 是相等的

// 因为它们是同一个对象！

})

obj.count++

这不同于返回响应式对象的 getter 函数：只有在 getter 函数返回不同的对象时，才会触发回调：

watch(

() => state.someObject,

() => {

// 仅当 state.someObject 被替换时触发

}

)

在上面的例子里，你可以显式地加上 deep 选项，强制转成深层侦听器：

watch(

() => state.someObject,

(newValue, oldValue) => {

// 注意：`newValue` 此处和 `oldValue` 是相等的

// \*除非\* state.someObject 被整个替换了

},

{ deep: true }

)

## watchEffect()

watch() 是懒执行的：仅在侦听源变化时，才会执行回调。但在某些场景中，我们希望在创建侦听器时，立即执行一遍回调。

const url = ref('https://...')

const data = ref(null)

async function fetchData() {

const response = await fetch(url.value)

data.value = await response.json()

}

// 立即获取

fetchData()

// ...再侦听 url 变化

watch(url, fetchData)

还可以用 watchEffect 函数 来简化。watchEffect() 会立即执行一遍回调函数，如果这时函数产生了副作用，Vue 会自动追踪副作用的依赖关系，自动分析出响应源。上面的例子可以重写为：

watchEffect(async () => {

const response = await fetch(url.value)

data.value = await response.json()

})

这个例子中，回调会立即执行。在执行期间，它会自动追踪 url.value 作为依赖（近似于计算属性）。每当 url.value 变化时，回调会再次执行。

watch 和 watchEffect 都能响应式地执行有副作用的回调。它们之间的主要区别是追踪响应式依赖的方式：

* watch 只追踪明确侦听的源。它不会追踪任何在回调中访问到的东西。另外，仅在响应源确实改变时才会触发回调。watch 会避免在发生副作用时追踪依赖，因此，我们能更加精确地控制回调函数的触发时机。
* watchEffect，则会在副作用发生期间追踪依赖。它会在同步执行过程中，自动追踪所有能访问到的响应式 property。这更方便，而且代码往往更简洁，但其响应性依赖关系不那么明确。

## 回调的刷新时机

当你更改了响应式状态，它可能会同时触发 Vue 组件更新和侦听器回调。默认情况下，用户创建的侦听器回调，都会在 Vue 组件更新之前被调用。这意味着你在侦听器回调中访问的 DOM 将是被 Vue 更新之前的状态。如果想在侦听器回调中能访问被 Vue 更新之后的DOM，你需要指明 flush: 'post' 选项：

watch(source, callback, {

flush: 'post'

})

watchEffect(callback, {

flush: 'post'

})

后置刷新的 watchEffect() 有个更方便的别名 watchPostEffect()：

import { watchPostEffect } from 'vue'

watchPostEffect(() => {

/\* 在 Vue 更新后执行 \*/

})

## 停止侦听器

在 setup() 或 <script setup> 中用同步语句创建的侦听器，会自动绑定到宿主组件实例上，并且会在宿主组件卸载时自动停止。因此，在大多数情况下，你无需关心怎么停止一个侦听器。一个关键点是，侦听器必须用同步语句创建：如果用异步回调创建一个侦听器，那么它不会绑定到当前组件上，你必须手动停止它，以防内存泄漏。如下方这个例子：

<script setup>

import { watchEffect } from 'vue'

// 它会自动停止

watchEffect(() => {})

// ...这个则不会！

setTimeout(() => {

watchEffect(() => {})

}, 100)

</script>

要手动停止一个侦听器，请调用 watch 或 watchEffect 返回的函数：

const unwatch = watchEffect(() => {})

// ...当该侦听器不再需要时

unwatch()

注意，需要异步创建侦听器的情况很少，请尽可能选择同步创建。如果需要等待一些异步数据，你可以使用条件式的侦听逻辑：

// 需要异步请求得到的数据

const data = ref(null)

watchEffect(() => {

if (data.value) {

// 数据加载后执行某些操作...

}

})

# 模板 ref

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 attribute：

<input ref="input">

ref 是一个特殊的 attribute，和 v-for 章节中提到的 key 类似。它允许我们在一个特定的 DOM 元素或子组件实例被挂载后，获得对它的直接引用。这可能很有用，比如说在组件挂载时编程式地聚焦到一个 input 元素上，或在一个元素上初始化一个第三方库。

## 访问模板 ref

为了通过组合式 API 获得该模板 ref，我们需要声明一个同名的 ref：

<script setup>

import { ref, onMounted } from 'vue'

// 声明一个 ref 来存放该元素的引用

// 必须和模板 ref 同名

const input = ref(null)

onMounted(() => {

input.value.focus()

})

</script>

<template>

<input ref="input" />

</template>

如果不使用 <script setup>，需确保从 setup() 返回 ref：

export default {

setup() {

const input = ref(null)

// ...

return {

input

}

}

}

注意，你只可以在组件挂载后才能访问 ref。如果你想在模板中的表达式上访问 input，在初次渲染时会是 null。这是因为在初次渲染前这个元素还压根不存在呢！如果你正试图观察一个模板 ref 的变化，确保考虑到 ref 的值为 null 的情况：

watchEffect(() => {

if (input.value) {

input.value.focus()

} else {

// 此时还未挂载，或此元素已经被卸载（例如通过 v-if 控制）

}

})

## v-for 中的 ref

当 ref 在 v-for 中使用时，相应的 ref 中包含的值是一个数组，它将在元素被挂载后填充：

<script setup>

import { ref, onMounted } from 'vue'

const list = ref([

/\* ... \*/

])

const itemRefs = ref([])

onMounted(() => console.log(itemRefs.value))

</script>

<template>

<ul>

<li v-for="item in list" ref="itemRefs">

{{ item }}

</li>

</ul>

</template>

ref 数组不能保证与源数组相同的顺序。

## 函数型 ref

除了使用字符串值作名字，ref attribute 还可以绑定为一个函数，会在每次组件更新时都被调用。函数接受该元素引用作为第一个参数：

<input :ref="(el) => { /\* 将 el 分配给 property 或 ref \*/ }">

如果你正在使用一个动态的 :ref 绑定，我们也可以传一个函数。当元素卸载时，这个 el 参数会是 null。你当然也可以使用一个方法而不是内联函数。

## 组件上的 ref

ref 也可以被用在一个子组件上。此时 ref 中引用的是组件实例：

<script setup>

import { ref, onMounted } from 'vue'

import Child from './Child.vue'

const child = ref(null)

onMounted(() => {

// child.value 是 <Child /> 组件的实例

})

</script>

<template>

<Child ref="child" />

</template>

如果一个子组件使用的是选项式 API 或没有使用 <script setup>，被引用的组件实例和该子组件的 this 完全一致，这意味着父组件对子组件的每一个属性和方法都有完全的访问权。这使得在父组件和子组件之间创建紧密耦合的实现细节变得很容易，当然也因此，应该只在绝对需要时才使用组件引用。大多数情况下，你应该首先使用标准的 props 和 emit 接口来实现父子组件交互。

有一个例外的情况，使用了 <script setup> 的组件是默认私有的：一个父组件无法访问到一个使用了 <script setup> 的子组件中的任何东西，除非子组件在其中通过 defineExpose 宏显式暴露：

<script setup>

import { ref } from 'vue'

const a = 1

const b = ref(2)

defineExpose({

a,

b

})

</script>

当父组件通过模板 ref 获取到了该组件的实例时，得到的实例类型为 { a: number, b: number } (ref 都会自动解包，和一般的实例一样)。

# 组件基础

组件允许我们将 UI 划分为独立的、可重用的部分来思考。组件在应用程序中常常被组织成层层嵌套的树状结构：

[组件嵌套树](组件嵌套树.png)

这和我们嵌套 HTML 元素的方式类似，Vue 实现了自己的组件数据模型，使我们可以在每个组件内封装自定义内容与逻辑。

## 定义组件

当使用构建步骤时，我们一般会将 Vue 组件定义在一个单独的 .vue 文件中，这被叫做单文件组件 (简称 SFC)：

<script setup>

import { ref } from 'vue'

const count = ref(0)

</script>

<template>

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

</template>

当不使用构建步骤时，一个 Vue 组件以一个包含 Vue 特定选项的 JavaScript 对象来定义：

import { ref } from 'vue'

export default {

setup() {

const count = ref(0)

return { count }

},

template: `

<button @click="count++">

You clicked me {{ count }} times.

</button>`

// 或者 `template: '#my-template-element'`

}

## 使用组件

要使用一个子组件，我们需要在父组件中导入它。假设我们把计数器组件放在了一个叫做 ButtonCounter.vue 的文件中，这个组件将会以默认导出的形式被暴露给外部。

<script setup>

import ButtonCounter from './ButtonCounter.vue'

</script>

<template>

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

<ButtonCounter />

</template>

通过 <script setup>，导入的组件都在模板中直接可用。当然，你也可以全局地注册一个组件，使得它在当前应用中的任何组件上都可以使用，而不需要额外再导入。组件可以被重用任意多次：

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

<ButtonCounter />

<ButtonCounter />

<ButtonCounter />

每当点击这些按钮时，每一个组件都维护着自己的状态，是不同的 count。这是因为每当你使用一个组件，就创建了一个新的实例。

在单文件组件中，推荐为子组件使用 PascalCase 的标签名，以此来和原生的 HTML 元素作区分。虽然原生 HTML 标签名是不区分大小写的，但 Vue 单文件组件是可以在编译中区分大小写的。我们也可以使用 /> 来关闭一个标签。如果你是直接在 DOM 中书写模板 (例如原生 <template> 元素的内容)，模板的编译将服从于原生 HTML 的解析行为。在这种情况下，你应该需要使用 kebab-case 形式并显式地关闭这些组件的标签。

<!-- 如果是在 DOM 中书写该模板 -->

<button-counter></button-counter>

<button-counter></button-counter>

<button-counter></button-counter>

## 传递 props

Props 是一种特别的 attributes，你可以在组件上声明注册。要传递给博客文章组件一个标题，我们必须在组件的 props 列表上声明它，使用 defineProps 宏：

<!-- BlogPost.vue -->

<script setup>

defineProps(['title'])

</script>

<template>

<h4>{{ title }}</h4>

</template>

defineProps 是一个仅 <script setup> 中可用的编译宏命令，并不需要显式地导入。声明的 props 会自动暴露给模板。defineProps 会返回一个对象，其中包含了可以传递给组件的所有 props，因此我们在 JavaScript 中使用：

const props = defineProps(['title'])

console.log(props.title)

如果你没有使用 <script setup>，props 必须以 props 选项的方式声明，props 对象会作为 setup() 函数的第一个参数：

export default {

props: ['title'],

setup(props) {

console.log(props.title)

}

}

一个组件可以有任意多的 props，默认情况下，任何值都可以传递给任何 prop。当一个 prop 被注册后，可以像这样以自定义 attribute 的形式传递数据给它：

<BlogPost title="My journey with Vue" />

<BlogPost title="Blogging with Vue" />

<BlogPost title="Why Vue is so fun" />

## 监听事件

组件添加一个按钮：

<!-- BlogPost.vue, 省略了 <script> -->

<template>

<div class="blog-post">

<h4>{{ title }}</h4>

<button>Enlarge text</button>

</div>

</template>

这个按钮目前还没有做任何事情，我们想要点击这个按钮来告诉父组件它应该放大所有博客文章的文字。要解决这个问题，组件实例提供了一个自定义事件系统。父组件可以通过 v-on 或 @ 来选择性地监听子组件上抛的事件，就像监听原生 DOM 事件那样：

<BlogPost

...

@enlarge-text="postFontSize += 0.1"

/>

子组件可以通过调用内置的 $emit 方法，通过传入事件名称来抛出一个事件：

<!-- BlogPost.vue, 省略了 <script> -->

<template>

<div class="blog-post">

<h4>{{ title }}</h4>

<button @click="$emit('enlarge-text')">Enlarge text</button>

</div>

</template>

我们可以通过 defineEmits 宏来选择性地声明需要抛出的事件：

<!-- BlogPost.vue -->

<script setup>

defineProps(['title'])

defineEmits(['enlarge-text'])

</script>

这记录了一个组件发出的所有事件，并可选择对其进行验证。这还使得 Vue 避免了将它们作为原生事件监听器隐式地应用于子组件的根元素。和 defineProps 类似，defineEmits 仅可用于 <script setup> 之中，并且不需要导入，返回的 emit 函数可以被用于在 JavaScript 代码中抛出事件：

const emit = defineEmits(['enlarge-text'])

emit('enlarge-text')

如果你没有在使用 <script setup>，你可以通过 emits 选项定义组件会抛出的事件。你可以从 setup() 函数的第二个参数，即 setup 上下文对象上访问到 emit 函数：

export default {

emits: ['enlarge-text'],

setup(props, ctx) {

ctx.emit('enlarge-text')

}

}

## 通过插槽来分配内容

和 HTML 元素一样，像这样能够向组件中传递内容是非常有用的：

<AlertBox>

Something bad happened.

</AlertBox>

这可以通过 Vue 的自定义 <slot> 元素来实现：

<template>

<div class="alert-box">

<strong>This is an Error for Demo Purposes</strong>

<slot />

</div>

</template>

<style scoped>

.alert-box {

/\* ... \*/

}

</style>

正如你上面所看到的，我们使用 <slot> 作为一个占位符，之后的内容就会放在这里。

## 动态组件

有的需求会想要在两个组件间来回切换，比如 Tab 界面，是通过 Vue 的 <component> 元素和特殊的 is attribute 实现的：

<!-- currentTab 改变时组件也改变 -->

<component :is="tabs[currentTab]"></component>

在上面的例子中，被传给 :is 的值可以是以下几种：

* 被注册的组件名
* 导入的组件对象

你也可以使用 is attribute 来创建一般的 HTML 元素。当使用 <component :is="..."> 来在多个组件间作切换时，组件会在被切换掉后卸载。我们可以通过 <KeepAlive> 组件强制不活跃的组件仍然保持“存活”的状态。

## DOM 模板解析注意事项

如果你想在 DOM 中直接书写 Vue 模板，Vue 则必须从 DOM 中获取模板字符串。因为浏览器的原生 HTML 解析行为，因此有一些需要注意的事项。

请注意下面讨论只适用于直接在 DOM 中编写模板的情况。如果你使用来自以下来源的字符串模板，它们不适用：

* 单文件组件
* 内联模板字符串 (例如 template: '...')
* <script type="text/x-template">

### 大小写区分

HTML 标签和属性名称是不分大小写的，所以浏览器会把任何大写的字符解释为小写。这意味着当你使用 DOM 内的模板时，无论是 PascalCase 形式的组件名称、camelCase 形式的 prop 名称还是 v-on 的事件名称，都需要转换为相应等价的 kebab-case (短横线连字符) 形式：

// JavaScript 中的 camelCase

const BlogPost = {

props: ['postTitle'],

emits: ['updatePost'],

template: `

<h3>{{ postTitle }}</h3>

`

}

<!-- HTML 中的 kebab-case -->

<blog-post post-title="hello!" @update-post="onUpdatePost"></blog-post>

### 闭合标签

我们在上面的例子中已经使用过了闭合标签 (self-closing tag)：

<MyComponent />

这是因为 Vue 的模板解析器将 /> 作为标签关闭的标志，无关其类型。然而在 DOM 模板中，我们必须显式地写出关闭标签：

<my-component></my-component>

这是由于 HTML 只允许一小部分特殊的元素省略其关闭标签，最常见的就是 <input> 和 <img>。对于其他的元素来说，如果你省略了关闭标签，原生的 HTML 解析器会认为开启的标签永远没有结束

### 元素位置限制

某些 HTML 元素对于放在其中的元素类型有限制，例如 <ul>，<ol>，<table> 和 <select>，相应的，某些元素仅在放置于特定元素中时才会显示，例如 <li>，<tr> 和 <option>。这将导致在使用带有此类限制元素的组件时出现问题。例如：

<table>

<blog-post-row></blog-post-row>

</table>

自定义的组件 <blog-post-row> 将作为无效的内容被忽略，因而在最终呈现的输出中造成错误。我们可以使用特殊的 is attribute 作为一种解决方案：

<table>

<tr is="vue:blog-post-row"></tr>

</table>

# 深入组件

## 组件注册

一个 Vue 组件需要被“注册”使得 Vue 在渲染模板时能找到其实现。有两种方式来注册组件：全局注册和局部注册。

### 全局注册

我们可以使用 app.component() 方法，让组件在当前 Vue 应用中全局可用。

import { createApp } from 'vue'

const app = createApp({})

app.component(

// 注册的名字

'MyComponent',

// 组件的实现

{

/\* ... \*/

}

)

如果使用单文件组件，你可以注册被导入的 .vue 文件：

import MyComponent from './App.vue'

app.component('MyComponent', MyComponent)

app.component() 方法可以被链式调用：

app

.component('ComponentA', ComponentA)

.component('ComponentB', ComponentB)

.component('ComponentC', ComponentC)

全局注册的组件可以在此应用的任意组件的模板中使用：

<!-- 这在当前应用的任意组件中都可用 -->

<ComponentA/>

<ComponentB/>

<ComponentC/>

所有的子组件也可以使用全局注册的组件，这意味着这三个组件也都可以在彼此内部使用。

### 局部注册

全局注册虽然很方便，但有以下几个短板：

* 全局注册使构建系统无法移除未使用的组件 (也叫“tree-shaking”)。如果你全局注册了一个组件，却一次都没有使用，它仍然会出现在最终的构建产物中。
* 全局注册在大型项目中使项目的依赖关系变得不那么明确。在父组件中使用子组件时，很难定位子组件的实现。这可能会影响未来长期的可维护性，类似于使用过多的全局变量。

局部注册将注册组件的可用性限定在当前组件的范围内。它使依赖关系更加明确，并且对 tree-shaking 更加友好。当你在单文件组件中使用了 <script setup>，导入的组件可以在本地使用而无需注册：

<script setup>

import ComponentA from './ComponentA.vue'

</script>

<template>

<ComponentA />

</template>

如果不在 <script setup> 中，你将需要使用 components 选项：

import ComponentA from './ComponentA.js'

export default {

components: {

ComponentA

},

setup() {

// ...

}

}

对于每个 components 对象里的属性，它们的 key 名就是注册的组件名，而值就是相应组件的实现。上面的例子中使用的是 ES2015 的缩写语法，等价于：

export default {

components: {

ComponentA: ComponentA

}

// ...

}

请注意：局部注册组件在后代组件中并不可用。

### 组件名格式

在整个指引中，我们都使用 PascalCase 作为组件名的注册格式，这是因为：

* PascalCase 是合法的 JavaScript 标识符。这使得在 JavaScript 中导入和注册组件都很容易，同时 IDE 也能提供较好的自动补全。
* <PascalCase /> 在模板中更明显地表明了这是一个 Vue 组件，而不是原生 HTML 元素。同时也能够将 Vue 组件和自定义元素 (web components) 区分开来。

在单文件组件和内联字符串模板中，我们都推荐这样做。但是，PascalCase 的标签名在 DOM 模板中是不可用的。

Vue 支持将使用 kebab-case 的标签解析为使用 PascalCase 注册的组件。这意味着一个以 MyComponent 为名注册的组件，在模板中可以通过 <MyComponent> 或 <my-component> 引用。这允许我们在不同来源的模板中始终使用同一份 JavaScript 组件注册代码。

## Props

### Props声明

组件需要显式声明 prop，这样 Vue 才能知道外部传入的哪些是 prop，哪些是透传 attribute (下一章中我们会讨论到它)。在单文件组件中使用 <script setup> 时，prop 可以使用 defineProps() 宏来定义：

<script setup>

const props = defineProps(['foo'])

console.log(props.foo)

</script>

在没有使用 <script setup> 的组件中，prop 可以使用 props 选项来定义：

export default {

props: ['foo'],

setup(props) {

// setup() 将 props 作为第一个参数

console.log(props.foo)

}

}

请注意，传递给 defineProps() 的参数和提供给 props 选项的值是相同的，两种声明方式背后其实使用的都是 prop 选项。除了使用字符串数组来声明 prop 外，还可以使用对象的形式：

// 使用 <script setup>

defineProps({

title: String,

likes: Number

})

// 非 <script setup>

export default {

props: {

title: String,

likes: Number

}

}

如果你正在搭配 TypeScript 使用 <script setup>，也可以使用类型标注来声明 prop：

<script setup lang="ts">

defineProps<{

title?: string

likes?: number

}>()

</script>

### 传递 prop 的细节

#### Prop 名字格式

如果 prop 的名字很长，应使用 camelCase 形式，因为它们是合法的 JavaScript 标识符，可以直接在模板的表达式中使用，避免在作为属性 key 名时必须带引号。

defineProps({

greetingMessage: String

})

<span>{{ greetingMessage }}</span>

从技术上来讲，你也可以在子组件传递 prop 时使用 camelCase 形式。(使用 DOM 模板时例外) 而实际上为了和 HTML attribute 对齐，都会将其转为 kebab-case 形式：

<MyComponent greeting-message="hello" />

一般情况下都会使用 PascalCase 作为组件标签名，因为这提高了模板的可读性，能很好地区分出 Vue 组件和原生 HTML 元素。然而这对于传递 prop 来说收效并不高，因此我们选择对其进行转换。

#### 静态 vs. 动态 Prop

至此，你已经见过了很多像这样的静态值形式的 prop：

<BlogPost title="My journey with Vue" />

相应地还有使用 v-bind 或缩写 : 所动态绑定的 prop 值：

<!-- 根据一个变量的值动态传入 -->

<BlogPost :title="post.title" />

<!-- 根据一个更复杂表达式的值动态传入 -->

<BlogPost :title="post.title + ' by ' + post.author.name" />

#### 传递不同的值类型

在上述的两个例子中，我们只传入了字符串值，但实际上任何类型的值都可以作为一个 prop。

* Number

<!-- 虽然 `42` 是静态的值，我们还是需要使用 v-bind -->

<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->

<BlogPost :likes="42" />

<!-- 根据一个变量的值动态传入 -->

<BlogPost :likes="post.likes" />

* Boolean

<!-- 仅写上 prop 但不传值，会隐式表达为 `true` -->

<BlogPost is-published />

<!-- 虽然 `false` 是静态的值，我们还是需要使用 v-bind -->

<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->

<BlogPost :is-published="false" />

<!-- 根据一个变量的值动态传入 -->

<BlogPost :is-published="post.isPublished" />

* Array

<!-- 虽然这个数组是静态不变的值，我们还是需要使用 v-bind -->

<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->

<BlogPost :comment-ids="[234, 266, 273]" />

<!-- 根据一个变量的值动态传入 -->

<BlogPost :comment-ids="post.commentIds" />

* Object

<!-- 虽然这个对象字面量是静态不变的值，我们还是需要使用 v-bind -->

<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->

<BlogPost

:author="{

name: 'Veronica',

company: 'Veridian Dynamics'

}"

/>

<!-- 根据一个变量的值动态传入 -->

<BlogPost :author="post.author" />

#### 使用一个对象绑定多个 prop

如果你想要将一个对象的所有属性都当作 prop 传入，你可以使用没有参数的 v-bind，即只是 v-bind 而非 :prop-name。例如，这里有一个 post 对象：

const post = {

id: 1,

title: 'My Journey with Vue'

}

以及下面的模板：

<BlogPost v-bind="post" />

而这实际上等价于：

<BlogPost :id="post.id" :title="post.title" />

### 单向数据流

所有的 prop 都遵循着**单向绑定**原则，prop 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了父组件的状态。另外，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'

想要更改 prop 通常都符合以下两种场景：

* prop 被用于传入初始值；而子组件想在之后将其作为一个局部数据属性。在这种情况下，最好是新定义一个局部数据属性，从 prop 上获取初始值即可：

const props = defineProps(['initialCounter'])

// 计数器只是将 props.initialCounter 作为初始值

// 像下面这样做就使 prop 和后续更新无关了

const counter = ref(props.initialCounter)

* prop 以原始的形式传入，但还需作转换。在这种情况中，最好是基于该 prop 值定义一个计算属性：

const props = defineProps(['size'])

// 该 prop 变更时计算属性也会自动更新

const normalizedSize = computed(() => props.size.trim().toLowerCase())

当对象或数组作为 prop 被传入时，虽然子组件无法更改 prop 绑定，但仍然可以更改对象或数组内部的值。这是因为 JavaScript 的对象和数组是按引用传递，而对 Vue 来说，阻止这样的改变不但非常昂贵，也不合理。这种更改的主要缺陷是它允许了子组件以某种不明显的方式影响了父组件的状态，可能会使数据流在将来变得更难以推理。

### Prop 校验

组件可以更细致地指定对其 prop 的需求，比如你上面看到的类型限制，如果并没有指定要求，Vue 会在浏览器的 JavaScript 控制台中抛出警告来提醒你。这在开发为其他人提供的组件时非常有用。要描述对 prop 的校验，你可以向 defineProps() 宏提供一个带有 prop 校验的对象，而不是一个字符串数组，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

},

// Number 类型的默认值

propD: {

type: Number,

default: 100

},

// 对象类型的默认值

propE: {

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

propF: {

validator(value) {

// The value must match one of these strings

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

}

},

// 函数类型的默认值

propG: {

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'

}

}

})

defineProps() 宏中的参数不可以访问 <script setup> 中定义的其他变量，因为在编译时整个表达式都会被移到外部的函数中。

补充细节：

* 所有 prop 默认都是可选的，除非声明了 required: true。
* 除 Boolean 外的未传递的可选 prop 将会有一个默认值 undefined。
* Boolean 类型的未传递 prop 将被转换为 false。你应该为它设置一个 default 值来确保行为符合预期。
* 如果声明了 default 值，那么在 prop 的值被解析为 undefined 时，无论 prop 是未被传递还是显式指明的 undefined，都会改为 default 值。

type 可以是下列这些原生构造器：

* String
* Number
* Boolean
* Array
* Object
* Date
* Function
* Symbol

另外，type 也可以是自定义的类或构造函数，可以通过 instanceof 来检查、断言。例如下面这个类：

class Person {

constructor(firstName, lastName) {

this.firstName = firstName

this.lastName = lastName

}

}

你可以将其作为一个 prop 的类型：

defineProps({

author: Person

})

这会校验 author prop 的值是否是由 new Person 创建的。

### Boolean 类型转换

为了更贴近原生 boolean attributes 的行为，声明为 Boolean 类型的 props 有特别的类型转换规则。以带有如下声明的 <MyComponent> 组件为例：

defineProps({

disabled: Boolean

})

该组件可以被这样使用：

<!-- 等同于传入 :disabled="true" -->

<MyComponent disabled />

<!-- 等同于传入 :disabled="false" -->

<MyComponent />

当需要一个 prop 在声明时允许多种类型时，应该像这样：

defineProps({

disabled: [Boolean, Number]

})

无论声明类型的顺序如何，Boolean 类型的特殊转换规则都会被应用。

## 透传 Attribute

### Attribute 继承

“透传 attribute”是传递给组件的 attribute 或者 v-on 事件监听器，但并没有显式地声明在所接收组件的 props 或 emits 上。最常见的例子就是 class、style 和 id。当一个组件以单个元素为根作渲染时，透传的 attribute 会自动添加到根元素的 attribute 中。举个例子，下面这个 <MyButton> 组件有这样的模板：

<!-- <MyButton> 的模板 -->

<button>click me</button>

一个父组件使用了这个组件：

<MyButton class="large" />

最后渲染出的 DOM 结果是：

<button class="large">click me</button>

#### 对 class 和 style 的合并

如果一个子组件的根元素已经有了 class 或 style attribute，它会和从父组件上继承的值合并。将之前的 <MyButton> 组件的模板改成这样：

<!-- <MyButton> 的模板 -->

<button class="btn">click me</button>

最后渲染出的 DOM 结果是：

<button class="btn large">click me</button>

#### v-on 监听器继承

同样的规则也适用于 v-on 事件监听器：

<MyButton @click="onClick" />

监听器 click 会被添加到 <MyButton> 的根元素，即那个原生的 <button> 元素之上。当原生的 <button> 被点击，会触发父组件的 onClick 方法。如果原生 button 元素已经通过 v-on 绑定了一个事件监听器，则这些监听器都会被触发。

#### 深层组件继承

如果一个组件在根节点上渲染另一个组件，例如，我们重构一下 <MyButton>，让它在根节点上渲染 <BaseButton>：

<!-- <MyButton/> 的模板，只是渲染另一个组件 -->

<BaseButton />

此时 <MyButton> 接收的透传 attribute 会直接传向 <BaseButton>。请注意：

* 透传的 attribute 不会包含 <MyButton> 上声明过的 props 或是针对 emits 声明事件的 v-on 侦听函数，换句话说，声明过的 props 和侦听函数被 <MyButton>“消费”了。
* 透传的 attribute 若符合声明，也可以作为 props 传入 <BaseButton>。

### 禁用 Attribute 继承

如果你不想要一个组件自动地继承 attribute，你可以在组件选项中设置 inheritAttrs: false。如果你使用了 <script setup>，你需要一个额外的 <script> 块来书写这个选项声明：

<script>

// 使用一个简单的 <script> to declare options

export default {

inheritAttrs: false

}

</script>

<script setup>

// ...setup 部分逻辑

</script>

最常见的需要禁用 attribute 继承的场景就是 attribute 需要应用在根节点以外的其他元素上。通过设置 inheritAttrs 选项为 false，你可以完全控制透传进来的 attribute 如何应用。这些透传进来的 attribute 可以在模板的表达式中直接用 $attrs 访问到。

<span>Fallthrough attribute: {{ $attrs }}</span>

这个 $attrs 对象包含了除组件的 props 和 emits 属性外的所有其他 attribute，例如 class，style，v-on 监听器等等。

这个 $attrs 对象包含了除组件的 props 和 emits 属性外的所有其他 attribute，例如 class，style，v-on 监听器等等。有几点需要注意：

* 和 props 有所不同，透传 attributes 在 JavaScript 中保留了它们原始的大小写，所以像 foo-bar 这样的一个 attribute 需要通过 $attrs['foo-bar'] 来访问。
* 像 @click 这样的一个 v-on 事件监听器将在此对象下被暴露为一个函数 $attrs.onClick。

我们想要所有像 class 和 v-on 监听器这样的透传 attribute 都应用在内部的 <button> 上而不是外层的 <div> 上。我们可以通过设定 inheritAttrs: false 和使用 v-bind="$attrs" 来实现：

<div class="btn-wrapper">

<button class="btn" v-bind="$attrs">click me</button>

</div>

请记住没有参数的 v-bind 会将一个对象的所有属性都作为 attribute 应用到目标元素上。

### 多根节点的 Attribute 继承

和单根节点组件有所不同，有着多个根节点的组件没有自动 attribute 透传行为。如果 $attrs 没有被显式绑定，将会抛出一个运行时警告。

<CustomLayout id="custom-layout" @click="changeValue" />

如果 <CustomLayout> 有下面这样的多根节点模板，由于 Vue 不知道要将 attribute 透传到哪里，所以会抛出一个警告。

<header>...</header>

<main>...</main>

<footer>...</footer>

如果 $attrs 被显式绑定，则不会有警告：

<header>...</header>

<main v-bind="$attrs">...</main>

<footer>...</footer>

### 在 JavaScript 中访问透传 Attribute

如果需要，你可以在 <script setup> 中使用 useAttrs() API 来访问一个组件的所有透传 attribute：

<script setup>

import { useAttrs } from 'vue'

const attrs = useAttrs()

</script>

如果没有使用 <script setup>，attrs 会作为 setup() 上下文对象的一个属性暴露：

export default {

setup(props, ctx) {

// 透传 attribute 被暴露为 ctx.attrs

console.log(ctx.attrs)

}

}

需要注意的是，虽然这里的 attrs 对象总是反映为最新的透传 attribute，但它并不是响应式的 (考虑到性能因素)。你不能通过侦听器去监听它的变化。如果你需要响应性，可以使用 prop。或者你也可以使用 onUpdated() 使得在每次更新时结合最新的 attrs 执行副作用。

## 插槽

### 插槽内容与插口

我们已经学习过组件能够接收任意类型的 JavaScript 值作为 props，但组件要如何接收模板内容呢？在某些场景中，我们可能想要为子组件传递一些模板片段，让子组件在它们的组件中渲染这些片段。举个例子，这里有一个 <FancyButton> 组件，可以像这样使用：

<FancyButton>

Click me! <!-- 插槽内容 -->

</FancyButton>

而 <FancyButton> 的模板是这样的：

<button class="fancy-btn">

<slot></slot> <!-- 插槽插口 -->

</button>

<slot> 元素是一个插槽的插口，标示了父元素提供的插槽内容将在哪里被渲染。

[插槽的插口](插槽的插口.png)

最终渲染出的 DOM 是这样：

<button class="fancy-btn">

Click me!

</button>

通过使用插槽，让 <FancyButton> 仅负责渲染外层的 <button> (以及相应的样式)，而内部的内容由父组件提供。通过和下面的 JavaScript 函数作对比，来以另一种方式理解插槽：

// 父元素传入插槽内容

FancyButton('Click me!')

// FancyButton 在自己的模板中渲染插槽内容

function FancyButton(slotContent) {

return (

`<button class="fancy-btn">

${slotContent}

</button>`

)

}

除了文本以外，插槽内容还可以是任意合法的模板内容。例如我们可以传入多个元素，甚至是组件：

<FancyButton>

<span style="color:red">Click me!</span>

<AwesomeIcon name="plus" />

</FancyButton>

通过使用插槽，<FancyButton> 组件更加灵活和具有可复用性。现在组件可以用在不同的地方渲染各异的内容，但同时还保证都具有相同的样式。

### 渲染作用域

插槽内容可以访问到父组件的数据作用域，因为插槽内容本身是在父组件模板中定义的。举个例子：

<span>{{ message }}</span>

<FancyButton>{{ message }}</FancyButton>

这里的两个 {{ message }} 插值表达式渲染的内容都是一样的。插槽内容无法访问子组件的数据，请牢记一条规则：

任何父组件模板中的东西都只被编译到父组件的作用域中；而任何子组件模板中的东西都只被编译到子组件的作用域中。

### 默认内容

在外部没有提供任何内容的情况下，为插槽指定默认内容用于渲染是很有用的。比如在 <SubmitButton> 组件中：

<button type="submit">

<slot></slot>

</button>

如果我们想在父组件没有提供任何插槽内容时，把“Submit”文本渲染到 <button> 内。需要将“Submit”写在 <slot> 标签之间，使其成为默认内容：

<button type="submit">

<slot>

Submit <!-- 默认内容 -->

</slot>

</button>

当我们在父组件中使用 <SubmitButton> 但不提供任何插槽内容：

<SubmitButton />

那么将渲染默认的“Submit”单词：

<button type="submit">Submit</button>

但如果我们提供了别的内容：

<SubmitButton>Save</SubmitButton>

那么将渲染提供的内容：

<button type="submit">Save</button>

### 具名插槽

有时在一个组件中包含多个插槽的插口是很有用的。举个例子，在一个 <BaseLayout> 组件中，有如下这样的模板：

<div class="container">

<header>

<!-- 标题内容放这里 -->

</header>

<main>

<!-- 主要内容放这里 -->

</main>

<footer>

<!-- 底部内容放这里 -->

</footer>

</div>

对于这种场景，<slot> 元素可以有一个特殊的 attribute name，用来给各个插槽分配唯一的 ID，以确定每一处要渲染的内容：

<div class="container">

<header>

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

</header>

<main>

<slot></slot>

</main>

<footer>

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

</footer>

</div>

没有提供 name 的 <slot> 插口会隐式地命名为“default”。在父组件中使用 <BaseLayout> 时，我们需要一种方式将多个插槽内容传入到各自目标插槽的插口。此时就需要用到**具名插槽**了。要为具名插槽传入内容，我们需要使用一个含 v-slot 指令的 <template> 元素，并将目标插槽的名字传给该指令：

<BaseLayout>

<template v-slot:header>

<!-- header 插槽的内容放这里 -->

</template>

</BaseLayout>

v-slot 有对应的简写 #，因此 <template v-slot:header> 可以简写为 <template #header>。其意思就是“将这部分模板片段传入子组件的 header 插槽中”。

[具名插槽](具名插槽.png)

使用 JavaScript 函数进行翻译：

// 传入不同的内容给不同名字的插槽

BaseLayout({

header: `...`,

default: `...`,

footer: `...`

})

// <BaseLayout> 渲染插槽内容到对应位置

function BaseLayout(slots) {

return (

`<div class="container">

<header>${slots.header}</header>

<main>${slots.default}</main>

<footer>${slots.footer}</footer>

</div>`

)

}

### 动态插槽名

动态指令参数在 v-slot 上也是有效的，即可以定义下面这样的动态插槽名：

<base-layout>

<template v-slot:[dynamicSlotName]>

...

</template>

<!-- 缩写为 -->

<template #[dynamicSlotName]>

...

</template>

</base-layout>

注意这里的表达式和动态指令参数受相同的语法限制。

### 作用域插槽

在上面的渲染作用域中讨论到，插槽的内容无法访问到子组件的状态。然而在某些场景下插槽的内容可能想要同时使用父组件域内和子组件域内的数据。要做到这一点，我们需要一种方法来让子组件在渲染时将一部分数据提供给插槽。我们也确实有办法这么做！可以像对组件传递 prop 那样，向一个插槽的插口上传递 attribute：

<!-- <MyComponent> 的模板 -->

<div>

<slot :text="greetingMessage" :count="1"></slot>

</div>

当需要接收插槽 prop 时，默认插槽和具名插槽的使用方式有一些小区别。下面我们将先展示默认插槽如何接受 prop，通过子组件标签上的 v-slot 指令，直接接收到了一个插槽 prop 对象：

<MyComponent v-slot="slotProps">

{{ slotProps.text }} {{ slotProps.count }}

</MyComponent>

[插槽作用域](插槽作用域.svg)

子组件传入插槽的 props 作为了 v-slot 指令的值，可以在插槽内的表达式中访问。你可以将作用域插槽类比为一个传入子组件的函数。子组件会将相应的 prop 作为参数去调用它：

MyComponent({

// 类比默认插槽，将其想成一个函数

default: (slotProps) => {

return `${slotProps.text} ${slotProps.count}`

}

})

function MyComponent(slots) {

const greetingMessage = 'hello'

return (

`<div>${

// 在插槽函数调用时传入 props

slots.default({ text: greetingMessage, count: 1 })

}</div>`

)

}

实际上，这已经和作用域插槽的最终代码编译结果、以及手动编写渲染函数时使用作用域插槽的方式非常类似了。v-slot="slotProps" 可以类比这里的函数签名，和函数的参数类似，我们也可以在 v-slot 中使用解构：

<MyComponent v-slot="{ text, count }">

{{ text }} {{ count }}

</MyComponent>

具名作用域插槽的工作方式也是类似的，插槽 props 可以作为 v-slot 指令的值被访问到：v-slot:name="slotProps"。当使用缩写时是这样：

<MyComponent>

<template #header="headerProps">

{{ headerProps }}

</template>

<template #default="defaultProps">

{{ defaultProps }}

</template>

<template #footer="footerProps">

{{ footerProps }}

</template>

</MyComponent>

向具名插槽中传入 props：

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

注意插槽上的 name 是由 Vue 保留的，不会作为 props 传递给插槽。因此最终 headerProps 的结果是 { message: 'hello' }。

## 依赖注入

### Prop Drilling

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，形成了一颗巨大的组件树，而某个深层的子组件需要一个较远的祖先组件中的部分内容。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：

[组件树](组件树.png)

这里的 <Footer> 组件可能其实根本不关心这些 props，但它仍然需要定义并将它们传递下去使得 <DeepChild> 能访问到这些 props，如果组件链路非常长，可能会影响到更多这条路上的组件。这一过程被称为“prop drilling”

为解决这一问题，可以使用 provide 和 inject。一个父组件相对于其所有的后代组件，会作为依赖提供者。任何后代的组件树，无论层级有多深，都可以注入由父组件提供给整条链路的依赖。

[注入依赖](注入依赖.png)

### Provide (供给)

要为组件后代供给数据，需要使用到 provide() 函数：

<script setup>

import { provide } from 'vue'

provide(/\* 注入名 \*/ 'message', /\* 值 \*/ 'hello!')

</script>

如果不使用 <script setup>，请确保 provide() 是在 setup() 同步调用的：

import { provide } from 'vue'

export default {

setup() {

provide(/\* 注入名 \*/ 'message', /\* 值 \*/ 'hello!')

}

}

provide() 函数接收两个参数。第一个参数被称为注入名，可以是一个字符串或是一个 Symbol。后代组件会用注入名来查找期望注入的值。一个组件可以多次调用 provide()，使用不同的注入名，注入不同的依赖值。第二个参数是供给的值，值可以是任意类型，包括响应式的状态，比如一个 ref：

import { ref, provide } from 'vue'

const count = ref(0)

provide('key', count)

供给的响应式状态使后代组件可以由此和供给者建立响应式的联系。

### 应用层 Provide

除了供给一个组件的数据，我们还可以在整个应用层面做供给：

import { createApp } from 'vue'

const app = createApp({})

app.provide(/\* 注入名 \*/ 'message', /\* 值 \*/ 'hello!')

应用级的供给在应用的所有组件中都可以注入。这在你编写插件时会特别有用，因为插件一般都不会使用组件形式来供给值。

### Inject (注入)

要注入祖先组件供给的数据，需使用 inject() 函数：

<script setup>

import { inject } from 'vue'

const message = inject('message')

</script>

如果供给的值是一个 ref，注入进来的就是它本身，而不会自动解包。这使得被注入的组件保持了和供给者的响应性链接。同样的，如果没有使用 <script setup>，inject() 需要在 setup() 同步调用：

import { inject } from 'vue'

export default {

setup() {

const message = inject('message')

return { message }

}

}

默认情况下，inject 假设传入的注入名会被某个祖先链上的组件提供。如果该注入名的确没有任何组件提供，则会抛出一个运行时警告。如果在供给的一侧看来属性是可选提供的，那么注入时我们应该声明一个默认值，和 props 类似：

// 如果没有祖先组件提供 "message"

// `value` 会是 "这是默认值"

const value = inject('message', '这是默认值')

在一些场景中，默认值可能需要通过调用一个函数或初始化一个类来取得。为了避免在不使用可选值的情况下进行不必要的计算或产生副作用，我们可以使用工厂函数来创建默认值：

const value = inject('key', () => new ExpensiveClass())

### 配合响应性

当使用响应式 provide/inject 值时，建议尽可能将任何对响应式状态的变更都保持在 provider 内部。这样可以确保 provide 的状态和变更操作都在同一个组件内，使其更容易维护。有的时候，我们可能需要在 injector 组件中更改数据。在这种情况下，我们推荐在 provider 组件内提供一个更改数据方法：

<!-- 在 provider 组件内 -->

<script setup>

import { provide, ref } from 'vue'

const location = ref('North Pole')

function updateLocation() {

location.value = 'South Pole'

}

provide('location', {

location,

updateLocation

})

</script>

<!-- 在 injector 组件 -->

<script setup>

import { inject } from 'vue'

const { location, updateLocation } = inject('location')

</script>

<template>

<button @click="updateLocation">{{ location }}</button>

</template>

最后，如果你想确保从 provide 传过来的数据不能被 injector 的组件更改，你可以使用readonly() 来包装提供的值。

<script setup>

import { ref, provide, readonly } from 'vue'

const count = ref(0)

provide('read-only-count', readonly(count))

</script>

### 使用 Symbol 作注入名

了解了如何使用字符串作为注入名。但如果你正在构建大型的应用程序，包含非常多的依赖供给，或者你正在编写提供给其他开发者使用的组件库，建议最好使用 Symbol 来作为注入名以避免潜在的冲突。建议在一个单独的文件中导出这些注入名 Symbol：

// keys.js

export const myInjectionKey = Symbol()

// 在供给方组件中

import { provide } from 'vue'

import { myInjectionKey } from './keys.js'

provide(myInjectionKey, { /\*

要供给的数据

\*/ });

// 注入方组件

import { inject } from 'vue'

import { myInjectionKey } from './keys.js'

const injected = inject(myInjectionKey)

## 异步组件

### 基础使用

在大型项目中，我们可能需要拆分应用为更小的块，并仅在需要时再从服务器加载相关组件。为实现这点，Vue 提供了一个 defineAsyncComponent 方法：

import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() => {

return new Promise((resolve, reject) => {

// ...从服务器获取组件

resolve(/\* 获取到的组件 \*/)

})

})

// ... 像使用其他一般组件一样使用 `AsyncComp`

如你所见，defineAsyncComponent 方法接收一个返回 Promise 的加载函数。这个 Promise 的 resolve 回调方法应该在从服务器获得组件定义时调用。你也可以调用 reject(reason) 表明加载失败。

ES 模块动态导入也会返回一个 Promise，所以多数情况下我们会将它和 defineAsyncComponent 搭配使用，类似 Vite 和 Webpack 这样的构建工具也支持这种语法，因此我们也可以用它来导入 Vue 单文件组件：

import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() =>

import('./components/MyComponent.vue')

)

最后得到的 AsyncComp 是一个包装器组件，仅在页面需要它渲染时才调用加载函数。另外，它还会将 props 传给内部的组件，所以你可以使用这个异步的包装器组件无缝地替换原始组件，同时实现延迟加载。

### 加载与错误状态

异步操作不可避免地会涉及到加载和错误状态，因此 defineAsyncComponent() 也支持在高级选项中处理这些状态：

const AsyncComp = defineAsyncComponent({

// 加载函数

loader: () => import('./Foo.vue'),

// 加载异步组件时使用的组件

loadingComponent: LoadingComponent,

// 展示加载组件前的延迟时间，默认为 200ms

delay: 200,

// 加载失败后展示的组件

errorComponent: ErrorComponent,

// 如果提供了一个 timeout 时间限制，并超时了

// 也会显示这里配置的报错组件，默认值是：Infinity

timeout: 3000

})

如果提供了一个加载组件，它将在内部组件加载时先行显示。在加载组件显示之前有一个默认的 200ms 延迟——这是因为在网络状况较好时，加载完成得太快，导致最终组件的替换可能看起来像是闪烁。

如果提供了一个报错组件，当加载器函数返回的 Promise 被 reject 时，它将被显示出来。你还可以指定一个超时时间，在请求耗时过长时显示报错组件。

### 搭配 Suspense 使用

异步组件可以搭配内置的 <Suspense> 组件一起使用，若想了解 <Suspense> 和异步组件之间交互，请参阅文档中 <Suspense> 章节。

# 可重用性

## 组合式函数

在 Vue 应用的概念中，“组合式函数”是一个利用 Vue 组合式 API 来封装和复用有状态逻辑的函数。

当构建前端应用时，我们常常需要复用公共任务的逻辑。例如为了在不同地方格式化时间而抽取一个可复用的函数。这个格式化函数封装了无状态的逻辑：它在接收一些输入后立刻返回所期望的输出。复用无状态逻辑的库有很多，诸如你可能听到过的 lodash 和 date-fns。

### 鼠标跟踪器示例

直接在组件中使用组合式 API 实现鼠标跟踪功能，它会是这样的：

<script setup>

import { ref, onMounted, onUnmounted } from 'vue'

const x = ref(0)

const y = ref(0)

function update(event) {

x.value = event.pageX

y.value = event.pageY

}

onMounted(() => window.addEventListener('mousemove', update))

onUnmounted(() => window.removeEventListener('mousemove', update))

</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>

但是，如果我们想在多个组件中复用这个相同的逻辑呢？我们可以把这个逻辑以一个组合式函数的形式提取到外部文件中：

// mouse.js

import { ref, onMounted, onUnmounted } from 'vue'

// 按照惯例，组合式函数名以“use”开头

export function useMouse() {

// 被组合式函数封装和管理的状态

const x = ref(0)

const y = ref(0)

// 组合式函数可以随时更改其状态。

function update(event) {

x.value = event.pageX

y.value = event.pageY

}

// 一个组合式函数也可以挂靠在所属组件的生命周期上

// 来启动和卸载副作用

onMounted(() => window.addEventListener('mousemove', update))

onUnmounted(() => window.removeEventListener('mousemove', update))

// 通过返回值暴露所管理的状态

return { x, y }

}

下面是它在组件中使用的方式：

<script setup>

import { useMouse } from './mouse.js'

const { x, y } = useMouse()

</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>

可以嵌套多个组合式函数：一个组合式函数可以调用一个或多个其他的组合式函数。这使得我们可以像使用多个组件组合成整个应用一样，用多个较小且逻辑独立的单元来组合形成复杂的逻辑。我们可以将添加和清除 DOM 事件监听器的逻辑放入一个组合式函数中：

// event.js

import { onMounted, onUnmounted } from 'vue'

export function useEventListener(target, event, callback) {

// 如果你想的话，

// 也可以用字符串形式的 CSS 选择器来寻找目标 DOM 元素

onMounted(() => target.addEventListener(event, callback))

onUnmounted(() => target.removeEventListener(event, callback))

}

现在，useMouse() 可以被简化为：

// mouse.js

import { ref } from 'vue'

import { useEventListener } from './event'

export function useMouse() {

const x = ref(0)

const y = ref(0)

useEventListener(window, 'mousemove', (event) => {

x.value = event.pageX

y.value = event.pageY

})

return { x, y }

}

### 异步状态示例

useMouse() 组合式函数没有接收任何参数，因此让我们再来看一个需要接收一个参数的组合式函数示例。在做异步数据请求时，我们常常需要处理不同的状态：加载中、加载成功和加载失败。

<script setup>

import { ref } from 'vue'

const data = ref(null)

const error = ref(null)

fetch('...')

.then((res) => res.json())

.then((json) => (data.value = json))

.catch((err) => (error.value = err))

</script>

<template>

<div v-if="error">Oops! Error encountered: {{ error.message }}</div>

<div v-else-if="data">

Data loaded:

<pre>{{ data }}</pre>

</div>

<div v-else>Loading...</div>

</template>

同样，如果在每个需要获取数据的组件中都要重复这种模式，那就太繁琐了。让我们把它抽取成一个组合式函数：

// fetch.js

import { ref } from 'vue'

export function useFetch(url) {

const data = ref(null)

const error = ref(null)

fetch(url)

.then((res) => res.json())

.then((json) => (data.value = json))

.catch((err) => (error.value = err))

return { data, error }

}

现在我们在组件里只需要：

<script setup>

import { useFetch } from './fetch.js'

const { data, error } = useFetch('...')

</script>

useFetch() 接收一个静态的 URL 字符串作为输入，所以它只执行一次请求，然后就完成了。但如果我们想让它在每次 URL 变化时都重新请求呢？那我们可以让它同时允许接收 ref 作为参数：

// fetch.js

import { ref, isRef, unref, watchEffect } from 'vue'

export function useFetch(url) {

const data = ref(null)

const error = ref(null)

function doFetch() {

// 在请求之前重设状态...

data.value = null

error.value = null

// unref() 解包可能为 ref 的值

fetch(unref(url))

.then((res) => res.json())

.then((json) => (data.value = json))

.catch((err) => (error.value = err))

}

if (isRef(url)) {

// 若输入的 URL 是一个 ref，那么启动一个响应式的请求

watchEffect(doFetch)

} else {

// 否则只请求一次

// 避免监听器的额外开销

doFetch()

}

return { data, error }

}

这个版本的 useFetch() 现在同时可以接收静态的 URL 字符串和 URL 字符串的 ref。当通过 isRef() 检测到 URL 是一个动态 ref 时，它会使用 watchEffect() 启动一个响应式的 effect。该 effect 会立刻执行一次，并在此过程中将 URL 的 ref 作为依赖进行跟踪。当 URL 的 ref 发生改变时，数据就会被重置，并重新请求。

### 约定和最佳实践

组合式函数约定用驼峰命名法命名，并以“use”作为开头。

#### 输入参数

尽管其响应性不依赖 ref，组合式函数仍可接收 ref 参数。如果编写的组合式函数会被其他开发者使用，你最好在处理输入参数时兼容 ref 而不只是原始的值。unref() 工具函数会对此非常有帮助：

import { unref } from 'vue'

function useFeature(maybeRef) {

// 若 maybeRef 确实是一个 ref，它的 .value 会被返回

// 否则，maybeRef 会被原样返回

const value = unref(maybeRef)

}

如果你的组合式函数在接收 ref 为参数时会产生响应式 effect，请确保使用 watch() 显式地监听此 ref，或者在 watchEffect() 中调用 unref() 来进行正确的追踪。

#### 返回值

在组合式函数中使用 ref() 而不是 reactive()。我们推荐的约定是组合式函数始终返回一个 ref 对象，这样该函数在组件中解构之后仍可以保持响应性：

// x 和 y 是两个 ref

const { x, y } = useMouse()

从组合式函数返回一个响应式对象会导致在对象解构过程中丢失与组合式函数内状态的响应性连接。与之相反，ref 则可以维持这一响应性连接。如果你更希望以对象 property 的形式从组合式函数中返回状态，你可以将要返回的对象用 reactive() 包装，这样其中的 ref 会被自动解包，例如：

const mouse = reactive(useMouse())

// mouse.x 链接到了原来的 x ref

console.log(mouse.x)

Mouse position is at: {{ mouse.x }}, {{ mouse.y }}

#### 副作用

在组合式函数中的确可以执行副作用 (例如：添加 DOM 事件监听器或者请求数据)，但请注意以下规则：

* 如果你在一个应用中使用了服务器端渲染 (SSR)，请确保在后置加载的声明钩子上执行 DOM 相关的副作用，例如：onMounted()。这些钩子仅会在浏览器中使用，因此可以确保能访问到 DOM。
* 确保在 onUnmounted() 时清理副作用。举个例子，如果一个组合式函数设置了一个事件监听器，它就应该在 onUnmounted() 中被移除 (就像我们在 useMouse() 示例中看到的一样)。当然也可以像之前的 useEventListener() 示例那样，使用一个组合式函数来自动帮你做这些事。

#### 组合式函数

抽取组合式函数不仅是为了复用，也是为了代码组织。随着组件复杂度的增高，你可能会最终发现组件多得难以查询和理解。组合式 API 会给予你足够的灵活性，让你可以基于逻辑问题将组件代码拆分成更小的函数：

<script setup>

import { useFeatureA } from './featureA.js'

import { useFeatureB } from './featureB.js'

import { useFeatureC } from './featureC.js'

const { foo, bar } = useFeatureA()

const { baz } = useFeatureB(foo)

const { qux } = useFeatureC(baz)

</script>

在某种程度上，你可以将这些提取出的组合式函数看作是可以相互通信的组件范围内的服务。

选项式 API 中使用组合式函数，组合式函数必须在 setup() 中调用。且其返回的绑定必须在 setup() 中返回，以便暴露给 this 及其模板：

import { useMouse } from './mouse.js'

import { useFetch } from './fetch.js'

export default {

setup() {

const { x, y } = useMouse()

const { data, error } = useFetch('...')

return { x, y, data, error }

},

mounted() {

// setup() 暴露的 property 可以在通过 `this` 访问到

console.log(this.x)

}

// ...其他选项

}

#### 其他技巧的比较

* 相比于 Mixin
* 不清晰的 property 来源：当使用了多个 mixin 时，实例 property 来自哪个 mixin 变得不清晰，这使追溯实现和理解组件行为变得困难。这也是我们推荐在组合式函数中使用 ref + 解构模式的理由：让 property 的来源在消费组件时一目了然。
* 命名空间冲突：多个来自不同作者的 mixin 可能会注册相同的 property 键名，造成命名冲突。若使用组合式函数，你可以通过在解构变量时对变量进行重命名来避免相同的键名。
* 隐式的跨 mixin 交流：多个 mixin 需要依赖共享的 property 键名来进行相互作用，这使得它们隐性地耦合在一起。而一个组合式函数的返回值可以作为另一个组合式函数的参数被传入，像普通函数那样。
* 相比于无渲染组件

组合式函数相对于无渲染组件的主要优势是：组合式函数不会产生额外的组件实例开销。当在整个应用中使用时，由无渲染组件产生的额外组件实例会带来无法忽视的性能开销。

* 相比于 React Hook

组合式函数和自定义 React hook 非常相似。组合式 API 的一部分灵感正来自于 React hook，Vue 的组合式函数也的确在逻辑组合能力上与 React hook 相近。然而，Vue 的组合式函数是基于 Vue 细粒度的响应性系统，这和 React hook 的执行模型有本质上的不同。

## 自定义指令

除了 Vue 内置的一系列指令 (比如 v-model 或 v-show) 之外，Vue 还允许你注册自定义的指令。我们已经介绍了两种在 Vue 中重用代码的方式：组件和组合式函数。组件是主要的构建模块，而组合式函数则侧重于有状态的逻辑。另一方面，自定义指令主要是为了重用涉及普通元素的底层 DOM 访问的逻辑。

在 <script setup> 中，任何以 v 开头的驼峰式命名的变量都可以被用作一个自定义指令。在上面的例子中，vFocus 即可以在模板中以 v-focus 的形式使用。。

个自定义指令被定义为一个包含类似于组件的生命周期钩子的对象。钩子接收指令绑定到的元素。下面是一个自定义指令的例子，当一个 input 元素被 Vue 插入到 DOM 中后，它将被聚焦：

<script setup>

// 在模板中启用 v-focus

const vFocus = {

mounted: (el) => el.focus()

}

</script>

<template>

<input v-focus />

</template>

假设你还未点击页面中的其他地方，那么上面这个 input 元素应该会被自动聚焦。该指令比 autofocus attribute 更有用，因为它不仅仅可以在页面加载完成后运行，还可以在 Vue 动态插入元素后运行。

将一个自定义指令全局注册到应用层级也是一种通用的做法：

const app = createApp({})

// 使 v-focus 在所有组件中都可用

app.directive('focus', {

/\* ... \*/

})

### 指令钩子

一个指令的定义对象可以提供几种钩子函数 (都是可选的)：

const myDirective = {

// 在绑定元素的 attribute 前

// 或事件监听器应用前调用

created(el, binding, vnode, prevVnode) {

// 下面会介绍各个参数的细节

},

// 在元素被插入到 DOM 前调用

beforeMount() {},

// 在绑定元素的父组件

// 及他自己的所有子节点都挂载完成后调用

mounted() {},

// 绑定元素的父组件更新前调用

beforeUpdate() {},

// 在绑定元素的父组件

// 及他自己的所有子节点都更新后调用

updated() {},

// 绑定元素的父组件卸载前调用

beforeUnmount() {},

// 绑定元素的父组件卸载后调用

unmounted() {}

}

指令的钩子会传递以下几种参数：

* el：指令绑定到的元素。这可以用于直接操作 DOM。
* binding：一个对象，包含以下 property。
  + value：传递给指令的值。例如在 v-my-directive="1 + 1" 中，值是 2。
  + oldValue：之前的值，仅在 beforeUpdate 和 updated 中可用。无论值是否更改，它都可用。
  + arg：传递给指令的参数 (如果有的话)。例如在 v-my-directive:foo 中，参数是 "foo"。
  + modifiers：一个包含修饰符的对象 (如果有的话)。例如在 v-my-directive.foo.bar 中，修饰符对象是 { foo: true, bar: true }。
  + instance：使用该指令的组件实例。
  + dir：指令的定义对象。
* vnode：代表绑定元素的底层 VNode。
* prevNode：之前的渲染中代表指令所绑定元素的 VNode。仅在 beforeUpdate 和 updated 钩子中可用

像下面这样使用指令：

<div v-example:foo.bar="baz">

binding 参数会是一个这样的对象：

{

arg: 'foo',

modifiers: { bar: true },

value: /\* `baz` 的值 \*/,

oldValue: /\* 上一次更新时 `baz` 的值 \*/

}

和内置指令类似，自定义指令的参数也可以是动态的。举个例子：

<div v-example:[arg]="value"></div>

这里指令的参数会基于组件状态 arg property 响应式地更新。

除了 el 外，你应该将这些参数都视为只读的，并一律不更改它们。若你需要在不同的钩子间共享信息，推荐通过元素的 dataset attribute 实现。

### 简化形式

对于自定义指令来说，需要在 mounted 和 updated 上实现相同的行为、又并不关心其他钩子的情况很常见。此时我们可以将指令定义成一个下面这样的函数：

<div v-color="color"></div>

app.directive('color', (el, binding) => {

// 这会在 `mounted` 和 `updated` 时都调用

el.style.color = binding.value

})

### 对象字面量

如果你的指令需要多个值，你可以向它传递一个 JavaScript 对象字面量。请记住，指令也可以接收任何合法的 JavaScript 表达式。

<div v-demo="{ color: 'white', text: 'hello!' }"></div>

app.directive('demo', (el, binding) => {

console.log(binding.value.color) // => "white"

console.log(binding.value.text) // => "hello!"

})

### 在组件上使用

当在组件上使用自定义指令时，它会始终应用于组件的根节点，和透传 attributes 类似。

<MyComponent v-demo="test" />

<!-- MyComponent 的模板 -->

<div> <!-- v-demo 指令会被应用在此处 -->

<span>My component content</span>

</div>

需要注意的是组件可能含有多个根节点。当应用到一个多根组件时，指令将会被忽略且抛出一个警告。和 attribute 不同，指令不能通过 v-bind="$attrs" 来传递给一个不同的元素。总而言之，不推荐在组件上使用自定义指令。

# 内置组件

## Transition·过渡

Vue 提供了两个内置组件，可以帮助你制作基于状态变化的过渡和动画：

* <Transition> 会在一个元素或组件进入和离开 DOM 时应用动画。当前这一章节正是对它的介绍。
* <TransitionGroup> 会在一个元素或组件被插入到 v-for 列表中，或是被移动或从其中移除时应用动画。
* 除了这两个组件，我们也可以通过其他技术手段来应用动画，比如切换 CSS class 或用状态绑定样式来驱动动画。

### <Transition> 组件

<Transition> 是一个内置组件，这意味着它在任意别的组件中都可以被使用，无需注册。它可以将进入和离开动画应用到通过默认插槽传递给它的元素或组件上。进入或离开可以由以下的条件之一触发：

* 由 v-if 所带来的条件渲染
* 由 v-show 所带来的条件显示
* 由特殊元素 <component> 切换的动态组件

以下是最基本用法的示例：

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

<Transition>

<p v-if="show">hello</p>

</Transition>

.v-enter-active,

.v-leave-active {

transition: opacity 0.5s ease;

}

.v-enter-from,

.v-leave-to {

opacity: 0;

}

<Transition> 仅支持单个元素或组件作为其插槽内容。如果内容是一个组件，这个组件必须仅有一个根元素。当一个 <Transition> 组件中的元素被插入或移除时，会发生下面这些事情：

* Vue 会自动检测目标元素是否应用了 CSS 过渡或动画。如果是，则一些 CSS 过渡 class 会在适当的时机被添加和移除。
* 如果有作为监听器的 JavaScript 钩子，这些钩子函数会在适当时机被调用。
* 如果没有探测到 CSS 过渡或动画、没有提供 JavaScript 钩子，那么 DOM 的插入、删除操作将在浏览器的下一个动画帧上执行。

### 基于 CSS 的过渡

#### CSS 过渡 class

一共有 6 个应用于进入与离开过渡效果的 CSS class。

[CSS 过渡 class](CSS%20过渡%20class.png)

1. v-enter-from：进入动画的起始状态。在元素插入之前添加，在元素插入完成后的下一帧移除。
2. v-enter-active：进入动画的生效状态。应用于整个进入动画阶段。在元素被插入之前添加，在过渡或动画完成之后移除。这个 class 可以被用来定义进入动画的持续时间、延迟与速度曲线类型。
3. v-enter-to：进入动画的结束状态。在元素插入完成后的下一帧被添加 (也就是 v-enter-from 被移除的同时)，在过渡或动画完成之后移除。
4. v-leave-from：离开动画的起始状态。在离开过渡效果被触发时立即添加，在一帧后被移除。
5. v-leave-active：离开动画的生效状态。应用于整个离开动画阶段。在离开过渡效果被触发时立即添加，在过渡或动画完成之后移除。这个 class 可以被用来定义离开动画的持续时间、延迟与速度曲线类型。
6. v-leave-to：离开动画的结束状态。在一个离开动画被触发后的下一帧被添加 (也就是 v-leave-from 被移除的同时)，在过渡或动画完成之后移除。

v-enter-active 和 v-leave-active 给我们提供了为进入和离开动画指定不同速度曲线的能力。

#### 为过渡命名

可以通过一个 name prop 来声明一种过渡：

<Transition name="fade">

...

</Transition>

对于一个已命名的过渡，它的过渡相关 class 会以其名字而不是 v 作为前缀。比如，上方例子中被应用的 class 将会是 fade-enter-active 而不是 v-enter-active。这个“fade”过渡的 class 应该是这样：

.fade-enter-active,

.fade-leave-active {

transition: opacity 0.5s ease;

}

.fade-enter-from,

.fade-leave-to {

opacity: 0;

}

#### CSS 的 transition

<Transition> 一般都会搭配原生 CSS 过渡一起使用，正如你在上面的例子中所看到的那样。这个 transition CSS 属性是一个简写形式，使我们可以一次定义一个过渡的各个方面，包括需要执行动画的属性、持续时间和速度曲线。下面是一个更高级的例子，它使用了不同的持续时间和速度曲线来过渡多个 property：

.fade-enter-active{

transition: all 0.3s ease-out;

}

.fade-leave-active {

transition: all 0.8s cubic-bezier(1, 0.5, 0.8, 1);

}

.fade-enter-from,

.fade-leave-to {

transform: translateX(20px);

opacity: 0;

}

#### CSS 的 animation

原生 CSS 动画和 CSS trasition 的应用方式基本上是相同的，只有一点不同，那就是 \*-enter-from 不是在元素插入后立即移除，而是在一个 animationend 事件触发时被移除。对于大多数的 CSS 动画，我们可以简单地在 \*-enter-active 和 \*-leave-active class 下声明它们。下面是一个示例：

.fade-enter-active{

animation: bounce-in 0.5s;

}

.fade-leave-active {

animation: bounce-in 0.5s reverse;

}

@keyframes bounce-in {

0% {

transform: scale(0);

}

50% {

transform: scale(1.25);

}

100% {

transform: scale(1);

}

}

#### 自定义过渡 class

你也可以向 <Transition> 传递以下的 props 来指定自定义的过渡 class：

* enter-from-class
* enter-active-class
* enter-to-class
* leave-from-class
* leave-active-class
* leave-to-class

你传入的这些 class 会覆盖相应阶段的默认 class 名。这个功能在你想要在 Vue 的动画机制下集成其他的第三方 CSS 动画库时非常有用

#### 同时使用 transition 和 animation

Vue 需要附加事件侦听器，以便知道过渡何时结束。可以是 transitionend 或 animationend，这取决于你所应用的 CSS 规则。如果你仅仅使用二者的其中之一，Vue 可以自动探测到正确的类型。然而在某些场景中，你或许想要在同一个元素上同时使用它们两个。

Vue 触发了一个 CSS 动画，同时鼠标悬停触发另一个 CSS 过渡。此时你需要显式地传入 type prop 来声明，告诉 Vue 需要关心哪种类型，传入的值是 animation 或 transition：

<Transition type="animation">...</Transition>

#### 深层级过渡与显式过渡时间

尽管过渡 class 仅能应用在 <Transition> 的直接子元素上，我们还是可以使用深层级的 CSS 选择器，使深层级的元素发生过渡。可以在嵌套元素上添加一个过渡延迟，这会创建一个交错进入动画序列：

<Transition name="fade">

<div v-if="show" class="outer">

<div class="inner">

Hello

</div>

</div>

</Transition>

.fade-enter-active .inner{

transition-delay: 0.25s;

}

.fade-leave-active .inner {

transition: all 0.3s ease-in-out;

}

.fade-enter-from .inner,

.fade-leave-to .inner {

transform: translateX(30px);

opacity: 0;

}

.outer, .inner {

background: #eee;

padding: 30px;

min-height: 100px;

}

.inner {

background: #ccc;

}

然而，这会带来一个小问题。默认情况下，<Transition> 组件会通过监听过渡根元素上的第一个 transitionend 或者 animationend 事件来尝试自动判断过渡何时结束。而在嵌套的过渡中，期望的行为应该是等待所有内部元素的过渡完成。在这种情况下，你可以通过向 <Transition> 组件传入 duration prop 来显式指定过渡的持续时间 (以毫秒为单位)。

如果有必要的话，你也可以用对象的形式传入，分开指定进入和离开所需的时间：

<Transition :duration="{ enter: 500, leave: 800 }">...</Transition>

#### 性能考量

你可能注意到我们上面例子中展示的动画所用到的属性大多是 transform 和 opacity 之类的。用这些属性制作动画非常高效，因为：

* 他们在动画过程中不会影响到 DOM 结构，因此每一个动画帧都不会触发昂贵的 CSS 布局重新计算。
* 大多数的现代浏览器都可以在执行 transform 动画时利用 GPU 进行硬件加速。

相比之下，像 height 或者 margin 这样的属性会触发 CSS 布局变动，因此执行它们的动画效果更昂贵，需要谨慎使用。

### JavaScript 钩子

你可以通过监听 <Transition> 组件事件的方式在过渡过程中挂上钩子函数：

<Transition

@before-enter="onBeforeEnter"

@enter="onEnter"

@after-enter="onAfterEnter"

@enter-cancelled="onEnterCancelled"

@before-leave="onBeforeLeave"

@leave="onLeave"

@after-leave="onAfterLeave"

@leave-cancelled="onLeaveCancelled"

>

<!-- ... -->

</Transition>

// 在元素被插入到 DOM 之前被调用

// 用这个来设置元素的 "enter-from" 状态

function onBeforeEnter(el) {},

// 在元素被插入到 DOM 之后的下一帧被调用

// 用这个来开始进入动画

function onEnter(el, done) {

// 调用回调函数 done 表示过渡结束

// 如果与 CSS 结合使用，则这个回调是可选参数

done()

}

// 当进入过渡完成时调用。

function onAfterEnter(el) {}

function onEnterCancelled(el) {}

// 在 leave 钩子之前调用

// 大多数时候，你应该只会用到 leave 钩子

function onBeforeLeave(el) {}

// 在离开过渡开始时调用

// 用这个来开始离开动画

function onLeave(el, done) {

// 调用回调函数 done 表示过渡结束

// 如果与 CSS 结合使用，则这个回调是可选参数

done()

}

// 在离开过渡完成、

// 且元素已从 DOM 中移除时调用

function onAfterLeave(el) {}

// 仅在 v-show 过渡中可用

function leaveCancelled(el) {}

这些钩子可以与 CSS 过渡或动画结合使用，也可以单独使用。在使用仅由 JavaScript 执行的动画时，最好是添加一个 :css="false" prop。这显式地向 Vue 表明跳过对 CSS 过渡的自动探测。除了性能稍好一些之外，还可以防止 CSS 规则意外地干扰过渡。

<Transition

...

:css="false"

>

...

</Transition>

在有了 :css="false" 后，我们就自己全权负责控制什么时候过渡结束了。这种情况下对于 @enter 和 @leave 钩子来说，回调函数 done 就是必须的。否则，钩子将被同步调用，过渡将立即完成。

### 可重用过渡

得益于 Vue 的组件系统，过渡是可以被重用的。要创建一个可被重用的过渡，我们需要为 <Transition> 组件创建一个包装组件，并向内传入插槽内容：

<!-- MyTransition.vue -->

<script>

// JavaScript 钩子逻辑...

</script>

<template>

<!-- 包装内置的 Transition 组件 -->

<Transition

name="my-transition"

@enter="onEnter"

@leave="onLeave">

<slot></slot> <!-- 向内传递插槽内容 -->

</Transition>

</template>

<style>

/\*

必要的 CSS...

注意：避免在这里使用 <style scoped>

因为那不会应用到插槽内容上

\*/

</style>

现在 MyTransition 可以在导入后像内置组件那样使用了：

<MyTransition>

<div v-if="show">Hello</div>

</MyTransition>

### 出现时过渡

如果你想在某个节点初次渲染时应用一个过渡效果，你可以添加 appear attribute：

<Transition appear>

...

</Transition>

### 元素间过渡

除了通过 v-if / v-show 切换一个元素，我们也可以通过 v-if / v-else / v-else-if 在几个组件间进行切换过：

<Transition>

<button v-if="docState === 'saved'">Edit</button>

<button v-else-if="docState === 'edited'">Save</button>

<button v-else-if="docState === 'editing'">Cancel</button>

</Transition>

### 过渡模式

在之前的例子中，进入和离开的元素都是在同时开始动画的，并且我们必须将它们设为 position: absolute 以避免二者同时存在时出现的布局问题。然而，在某些场景中这可能不是个好的方案，或者并不能符合行为预期。我们可能想要先执行离开动画，然后在其完成之后再执行元素的进入动画。手动编排这样的动画是非常复杂的，好在我们可以通过向 <Transition> 传入一个 mode prop 来实现这个行为：

### 组件间过渡

<Transition> 也可以用在动态组件之间：

<Transition name="fade" mode="out-in">

<component :is="activeComponent"></component>

</Transition>

### 动态过渡

<Transition> 的 prop (比如 name) 也可以是动态的！这让我们可以根据状态变化动态地应用不同类型的过渡：

<Transition :name="transitionName">

<!-- ... -->

</Transition>

当你使用 Vue 的过渡 class 约定规则定义了 CSS 过渡或动画，并想在它们之间切换时，这可能很有用。你也可以根据你的组件的当前状态在 JavaScript 过渡钩子中应用不同的行为。

## TransitionGroup·过渡组

### 和 <Transition> 的区别

<TransitionGroup> 支持和 <Transition> 基本相同的 prop、CSS 过渡 class 和 JavaScript 钩子监听器，但有以下几点区别：

* 默认情况下，它不会渲染一个包装器元素。但你可以通过传入 tag prop 来指定一个元素作为包装器元素来渲染。
* 过渡模式在这里不可用，因为我们不再是在互斥的元素之间进行切换。
* 其中的元素总是必须有一个独一无二的 key attribute。
* CSS 过渡 class 会被应用在其中的每一个元素上，而不是这个组的容器上。
* 当你是在 DOM 模板中使用时，组件名需要写为 <transition-group>。

### 进入 / 离开过渡

这里是 <TransitionGroup> 对一个 v-for 列表应用进入 / 离开过渡的示例：

import {ref,reactive} from "vue";

const items = reactive([1,2,3,4,5]);

const count = ref(6)

function addItems(){

let index = getRandom(0,items.length - 1)

items.splice(index,0,count.value++);

}

function removeItems(){

if (items.length > 0){

let index = getRandom(0,items.length - 1)

items.splice(index,1);

}

}

//根据最小值和最大值产生一个随机数

function getRandom(min, max) {

return Math.floor(Math.random() \* (max - min) + min);

}

### 移动过渡

上面的示例有一些明显的缺陷：当某一项被插入或移除时，它周围的元素会立即发生“跳跃”而不是平稳地移动。我们可以通过添加一些额外的 CSS 规则来解决这个问题：

.list-move, /\* 对移动中的元素应用的过渡 \*/

.list-enter-active,

.list-leave-active {

transition: all 0.5s ease;

}

.list-enter-from,

.list-leave-to {

opacity: 0;

transform: translateX(30px);

}

/\* 确保将离开的元素从布局流中删除

以便能够正确地计算移动的动画。 \*/

.list-leave-active {

position: absolute;

}

### 交错的列表过渡

如果通过 data attribute 用 JavaScript 来执行过渡时，那么我们也可以实现列表中的交错过渡。首先，我们把某一项的索引作为 DOM 元素上的一个 data attribute 呈现出来。

<input v-model="inputItem">

<TransitionGroup

tag="ul"

:css="false"

@before-enter="onBeforeEnter"

@enter="onEnter"

@leave="onLeave"

>

<li

v-for="(item, index) in computedList"

:key="item"

:data-index="index"

>

{{ item }}

</li>

</TransitionGroup>

const inputItem = ref("");

const itemName = reactive(['Bruce Lee','Jackie Chan','Chuck Norris','Jet Li','Kung Fury']);

const computedList = computed(() => {

return itemName.filter((item) => item.toLowerCase().includes(inputItem.value))

})

function onBeforeEnter(el) {

el.style.opacity = 0

el.style.height = 0

}

function onEnter(el, done) {

gsap.to(el, {

opacity: 1,

height: '1.6em',

delay: el.dataset.index \* 0.15,

onComplete: done

})

}

function onLeave(el, done) {

gsap.to(el, {

opacity: 0,

height: 0,

delay: el.dataset.index \* 0.15,

onComplete: done

})

}

## KeepAlive

<KeepAlive> 是一个内置组件，使我们可以在动态切换多个组件时视情况缓存组件实例。

### 基本使用

默认情况下，一个活跃的组件实例会在切走后被卸载。这会导致它丢失其中所有已变化的状态。在切换时创建新的组件实例通常是有用的行为，但在这个例子中，我们是的确想要组件能在非活跃状态时保留它们的状态。要解决这个问题，我们可以用内置的 <KeepAlive> 组件将这些动态组件包装起来：

<!-- 非活跃的组件将会被缓存！ -->

<KeepAlive>

<component :is="activeComponent" />

</KeepAlive>

DOM 模板中使用时，它应该被写为 <keep-alive>。

### 包含/排除

默认情况下，<KeepAlive> 会缓存内部的任何组件实例。但我们可以通过 include 和 exclude prop 来定制该行为。这两个 prop 的值都可以是一个以英文逗号分隔的字符串、一个正则表达式，或是包含这两种类型的一个数组：

<!-- 以英文逗号分隔的字符串 -->

<KeepAlive include="a,b">

<component :is="view" />

</KeepAlive>

<!-- 正则表达式 (需使用 `v-bind`) -->

<KeepAlive :include="/a|b/">

<component :is="view" />

</KeepAlive>

<!-- 数组 (需使用 `v-bind`) -->

<KeepAlive :include="['a', 'b']">

<component :is="view" />

</KeepAlive>

它会根据组件的 name 选项进行匹配，所以组件如果想要条件性地被 KeepAlive 缓存，就必须显式声明一个 name 选项。

### 最大缓存实例数

我们可以通过传入 max prop 来限制可被缓存的最大组件实例数。<KeepAlive> 的行为在指定了 max 后类似一个 LRU 缓存：如果缓存的实例数量即将超过指定的那个最大数量，则最久没有被访问的缓存实例将被销毁，以便为新的实例腾出空间。

<KeepAlive :max="10">

<component :is="activeComponent" />

</KeepAlive>

### 缓存实例的生命周期

当一个组件实例从 DOM 上移除但因为被 <KeepAlive> 缓存而仍作为组件树的一部分时，它将变为不活跃状态而不是被卸载。当一个组件实例作为缓存树的一部分插入到 DOM 中时，它将重新被激活。一个持续存在的组件可以通过 onActivated() 和 onDeactivated() 注册相应的两个状态的生命周期钩子：

<script setup>

import { onActivated, onDeactivated } from 'vue'

onActivated(() => {

// 调用时机为首次挂载

// 以及每次从缓存中被重新插入时

})

onDeactivated(() => {

// 在从 DOM 上移除、进入缓存

// 以及组件卸载时调用

})

</script>

请注意：

* onActivated 在组件挂载时也会调用，并且 onDeactivated 在组件卸载时也会调用。
* 这两个钩子不仅适用于 <KeepAlive> 缓存的根组件，也适用于缓存树中的后代组件。

## Teleport·传送门

<Teleport> 是一个内置组件，使我们可以将一个组件的一部分模板“传送”到该组件的 DOM 层次结构之外的 DOM 节点中。

### 基本使用

有时我们可能会遇到以下情况：组件模板的一部分在逻辑上属于它，但从视图角度来看，在 DOM 中它应该显示在 Vue 应用之外的其他地方。

最常见的例子是构建一个全屏的模态框时。理想情况下，我们希望模态框的按钮和模态框本身是在同一个组件中，因为它们都与组件的开关状态有关。但这意味着该模态框将与按钮一起呈现，并且位于应用程序的 DOM 更深的层次结构中。在想要通过 CSS 选择器定位该模态框时非常困难。试想下面这样的 HTML 结构：

<div class="outer">

<h3>Tooltips with Vue 3 Teleport</h3>

<div>

<MyModal />

</div>

</div>

看看 modal-button 的实现。

<script setup>

import { ref } from 'vue'

const open = ref(false)

</script>

<template>

<button @click="open = true">Open Modal</button>

<div v-if="open" class="modal">

<p>Hello from the modal!</p>

<button @click="open = false">Close</button>

</div>

</template>

<style scoped>

.modal {

position: fixed;

z-index: 999;

top: 20%;

left: 50%;

width: 300px;

margin-left: -150px;

}

</style>

这个组件中有一个 <button> 按钮来触发打开模态框，和一个 class 名为 .modal 的 <div>，它包含了模态框的内容和一个用来关闭的按钮。当在初始 HTML 结构中使用这个组件时，会有一些潜在的问题：

* position: fixed 能够相对于视口放置的条件是：没有任何祖先元素设置了 transform、perspective 或者 filter 样式属性。而如果我们想要用 CSS transform 为祖先节点 <div class="outer"> 设置动画，则会破坏模态框的布局结构！
* 这个模态框的 z-index 被包含它的元素所制约。如果有其他元素与 <div class="outer"> 重叠并有更高的 z-index，则它会覆盖住我们的模态框。

<Teleport> 提供了一个更简洁的方式来解决此类问题，使我们无需考虑那么多层 DOM 结构的问题。让我们用 <Teleport> 改写一下 <MyModal>：

<button @click="open = true">Open Modal</button>

<Teleport to="body">

<div v-if="open" class="modal">

<p>Hello from the modal!</p>

<button @click="open = false">Close</button>

</div>

</Teleport>

为 <Teleport> 指定的目标 to 期望接收一个 CSS 选择器字符串或者一个真实的 DOM 节点。这里我们其实就是让 Vue 去“传送这部分模板片段到 body 标签下”。

### 搭配组件使用

<Teleport> 只改变了渲染的 DOM 结构，它不会影响组件间的逻辑关系。也就是说，如果 <Teleport> 包含了一个组件，那么该组件始终和这个使用了 <teleport> 的组件保持逻辑上的父子关系。传入的 props 和触发的事件也会照常工作。

这也意味着来自父组件的注入也会按预期工作，子组件将在 Vue Devtools 中嵌套在父级组件下面，而不是放在实际内容移动到的地方。

### 禁用传送门

在某些场景中，你们可能需要视情况禁用 <Teleport>。举个例子，我们想要在桌面端将一个组件当做浮层来渲染，但在移动端则当作行内组件。可以对 <Teleport> 动态地传入一个 disabled prop 来处理这两种不同情况。

<Teleport :disabled="isMobile">

...

</Teleport>

这里的 isMobile 状态可以根据媒体查询的不同结果动态地更新。

### 同一目标上多个传送门

一个常见的应用场景就是写一个可重用的 <Modal> 组件，可能同时存在多个实例。对于此类场景，多个 <teleport> 组件可以将其内容挂载在同一个目标元素上，而顺序就是简单的顺次追加，后挂载的将排在目标元素下更后面的位置上。我们给出下面这样的用例：

<Teleport to="#modals">

<div>A</div>

</Teleport>

<Teleport to="#modals">

<div>B</div>

</Teleport>

渲染的结果为：

<div id="modals">

<div>A</div>

<div>B</div>

</div>

## Suspense

<Suspense> 是一个内置组件，用来在组件树中编排异步依赖。它可以在等待组件树下的多个嵌套异步依赖项解析完成时，呈现加载状态。

### 异步依赖

要了解 <Suspense> 所解决的问题和它是如何与异步依赖进行交互的，我们需要想象这样一种组件层级结构：

<Suspense>

└─ <Dashboard>

├─ <Profile>

│ └─ <FriendStatus>（组件有异步的 setup()）

└─ <Content>

├─ <ActivityFeed> （异步组件）

└─ <Stats>（异步组件）

在这个组件树中有多个嵌套组件，要渲染出它们，首先得解析一些异步资源。如果没有 <Suspense>，则它们每个都需要处理自己的加载、报错和完成状态。在最坏的情况下，我们可能会在页面上看到三个旋转的加载态，在不同的时间显示出内容。有了 <Suspense> 组件后，我们就可以在等待整个多层级组件树中的各个异步依赖获取结果时，在顶层展示出加载中或加载失败的状态。

<Suspense> 可以等待的异步依赖有两种：

* 带有异步 setup() 钩子的组件。这也包含了使用 <script setup> 时有顶层 await 表达式的组件。
* 异步组件。

#### async setup()

组合式 API 中组件的 setup() 钩子可以是异步的：

export default {

async setup() {

const res = await fetch(...)

const posts = await res.json()

return {

posts

}

}

}

如果使用 <script setup>，那么顶层 await 表达式会自动让该组件成为一个异步依赖：

<script setup>

const res = await fetch(...)

const posts = await res.json()

</script>

<template>

{{ posts }}

</template>

#### 异步组件

异步组件默认就是 “suspensible” 的。这意味着如果组件关系链上有一个 <Suspense>，那么这个异步组件就会被当作这个 <Suspense> 的一个异步依赖。在这种情况下，加载状态是由 <Suspense> 控制，而该组件自己的加载、报错、延时和超时等选项都将被忽略。异步组件也可以通过在选项中指定 suspensible: false 表明不用 Suspense 控制，并让组件始终自己控制其加载状态。

### 加载中状态

<Suspense> 组件有两个插槽：#default 和 #fallback。两个插槽都只允许一个直接子节点。在可能的时候都将显示默认槽中的节点。否则将显示后备槽中的节点。

<Suspense>

<!-- 具有深层异步依赖的组件 -->

<Dashboard />

<!-- 在 #fallback 插槽中显示 “正在加载中” -->

<template #fallback>

Loading...

</template>

</Suspense>

在初始渲染时，<Suspense> 将在内存中渲染其默认的插槽内容。如果在这个过程中遇到任何异步依赖，则会进入挂起状态。在挂起状态期间，展示的是后备内容。当所有遇到的异步依赖都完成后，<Suspense> 会进入完成状态，并将展示出默认插槽的内容。

如果在初次渲染时没有遇到异步依赖，<Suspense> 会直接进入完成状态。进入完成状态后，只有当默认插槽的根节点被替换时，<Suspense> 才会回到挂起状态。组件树中新的更深层次的异步依赖不会造成 <Suspense> 回退到挂起状态。

发生回退时，后备内容不会立即展示出来。相反，<Suspense> 在等待新内容和异步依赖完成时，会展示之前 #default 插槽的内容。这个行为可以通过一个 timeout prop 进行配置：在等待渲染新内容耗时超过 timeout 之后，<Suspense> 将会切换为展示后备内容。若 timeout 值为 0 将导致在替换默认内容时立即显示后备内容。

### 事件

除了 pending 事件之外，<suspense> 组件还有 resolve 和 fallback 事件。pending 事件是在进入挂起状态时触发。resolve 事件是在 default 插槽完成获取新内容时触发。fallback 事件则是在 fallback 插槽展示时触发。例如，可以使用这些事件在加载新组件时在之前的 DOM 最上层显示一个加载指示器。

### 错误处理

<Suspense> 组件自身目前还不提供错误处理，不过你可以使用 errorCaptured 选项或者 onErrorCaptured() 钩子，在使用到 <Suspense> 的父组件中捕获和处理异步错误。

### 和其他组件结合

我们常常会将 <Suspense> 和 <Transition>、<KeepAlive> 等组件结合。要保证这些组件都能正常工作，嵌套的顺序非常重要。另外，这些组件都通常与 Vue Router 中的 <RouterView> 组件结合使用。

下面的示例展示了如何嵌套这些组件，使它们都能按照预期的方式运行。若想组合得更简单，你也可以删除一些你不需要的组件：

<RouterView v-slot="{ Component }">

<template v-if="Component">

<Transition mode="out-in">

<KeepAlive>

<Suspense>

<!-- 主要内容 -->

<component :is="Component"></component>

<!-- 加载中状态 -->

<template #fallback>

正在加载...

</template>

</Suspense>

</KeepAlive>

</Transition>

</template>

</RouterView>

Vue Router 使用动态导入对懒加载组件进行了内置支持。这些与异步组件不同，目前他们不会触发 <Suspense>。但是，它们仍然可以有异步组件作为后代，这些组件可以照常触发 <Suspense>。

# Pinia

## 介绍

一个商店（如 Pinia）是一个实体，它持有未绑定到您的组件树的状态和业务逻辑。换句话说，它托管全局状态。它有点像一个始终存在并且每个人都可以读取和写入的组件。它包含三个概念，状态、getter和动作，并且可以安全地假设这些概念等同于data,computed和methods在组件中。

Pinia最初是在 2019 年 11 月左右重新设计使用Composition API的 Vue Store 外观的实验。从那时起，最初的原则仍然相同，但 Pinia 适用于 Vue 2 和 Vue 3 ，并且不需要你使用组合 API。除了安装和SSR之外，两者的 API 都是相同的，并且这些文档针对 Vue 3。

### Pinia特性

Pinia 是 Vue 的存储库，它允许您跨组件/页面共享状态。如果您熟悉 Composition API，您可能会认为您已经可以使用简单的export const state = reactive({}). 这对于单页应用程序来说是正确的，但如果它是服务器端呈现的，则会将您的应用程序暴露给安全漏洞。但即使在小型单页应用程序中，您也可以从使用 Pinia 中获得很多好处：

* 开发工具支持
  + 跟踪动作、突变的时间表
  + 商店出现在使用它们的组件中
  + 时间移动容易的调试
* 热模块更换
  + 在不重新加载页面的情况下修改您的存储值
  + 在开发时保持任何现有状态
* 插件：使用插件扩展 Pinia 功能
* 为 JS 用户提供适当的 TypeScript 支持或自动完成功能
* 服务器端渲染支持

### 基本示例

这就是使用 pinia 在 API 方面的样子（请务必查看入门以获取完整说明）。您首先创建一个商店：

// stores/counter.js

import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', {

state: () => {

return { count: 0 }

},

// could also be defined as

// state: () => ({ count: 0 })

actions: {

increment() {

this.count++

},

},

})

然后在组件中使用它：

import { useCounterStore } from '@/stores/counter'

export default {

setup() {

const counter = useCounterStore()

counter.count++

// with autocompletion ✨

counter.$patch({ count: counter.count + 1 })

// or using an action instead

counter.increment()

},

}

这是一个更完整的 API 示例，您将在 Pinia中使用类型，即使在 JavaScript 中也是如此。

import { defineStore } from 'pinia'

export const useTodos = defineStore('todos', {

state: () => ({

/\*\* @type {{ text: string, id: number, isFinished: boolean }[]} \*/

todos: [],

/\*\* @type {'all' | 'finished' | 'unfinished'} \*/

filter: 'all',

// type will be automatically inferred to number

nextId: 0,

}),

getters: {

finishedTodos(state) {

// autocompletion! ✨

return state.todos.filter((todo) => todo.isFinished)

},

unfinishedTodos(state) {

return state.todos.filter((todo) => !todo.isFinished)

},

/\*\*

\* @returns {{ text: string, id: number, isFinished: boolean }[]}

\*/

filteredTodos(state) {

if (this.filter === 'finished') {

// call other getters with autocompletion ✨

return this.finishedTodos

} else if (this.filter === 'unfinished') {

return this.unfinishedTodos

}

return this.todos

},

},

actions: {

// any amount of arguments, return a promise or not

addTodo(text) {

// you can directly mutate the state

this.todos.push({ text, id: this.nextId++, isFinished: false })

},

},

})

### 与Vuex 3.x/4.x 的比较

Pinia API 与 Vuex ≤4 有很大不同，即：

* 突变不再存在。他们经常被认为是非常冗长的。他们最初带来了 devtools 集成，但这不再是问题。
* 无需创建自定义复杂包装器来支持 TypeScript，所有内容都是类型化的，并且 API 的设计方式尽可能利用 TS 类型推断。
* 不再需要注入魔法字符串、导入函数、调用它们，享受自动完成功能！
* 无需动态添加商店，默认情况下它们都是动态的，您甚至都不会注意到。请注意，您仍然可以随时手动使用商店进行注册，但因为它是自动的，您无需担心。
* 不再有模块的嵌套结构。您仍然可以通过在另一个商店中导入和使用商店来隐式嵌套商店，但 Pinia 通过设计提供平面结构，同时仍然支持商店之间的交叉组合方式。你甚至可以有 store 的循环依赖。
* 没有命名空间的模块。鉴于商店的扁平架构，“命名空间”商店是其定义方式所固有的，您可以说所有商店都是命名空间的。

## 使用Pinia

### 定义store

存储是使用定义的defineStore()，并且它需要一个唯一的名称，作为第一个参数传递：

import { defineStore } from 'pinia'

// useStore could be anything like useUser, useCart

// the first argument is a unique id of the store across your application

export const useStore = defineStore('main', {

// other options...

})

这个名称，也称为id，是必要的，Pania 使用它来将商店连接到 devtools。将返回的函数命名为use...是可组合项之间的约定，以使其使用习惯。

### 使用store

定义了一个useCounterStore的store，以根据需要定义任意数量的store，并且应该在不同的文件中定义每个商店以充分利用 pinia（例如自动允许您的包进行代码拆分和 TypeScript 推理）。实例化store直接在商店中访问定义在state、getters和中的任何属性。

请注意，这store是一个用 包裹的对象reactive，这意味着不需要.value在 getter 之后编写，但是像propsin一样setup，我们不能对其进行解构：

export default defineComponent({

setup() {

const store = useStore()

// ❌ This won't work because it breaks reactivity

// it's the same as destructuring from `props`

const { name, doubleCount } = store

name // "eduardo"

doubleCount // 2

return {

// will always be "eduardo"

name,

// will always be 2

doubleCount,

// this one will be reactive

doubleValue: computed(() => store.doubleCount),

}

},

})

为了从存储中提取属性同时保持其反应性，您需要使用storeToRefs(). 它将为每个反应属性创建参考。当您仅使用store调用任何操作时，这很有用。请注意，您可以直接从store中解构操作，因为它们也绑定到store本身：

import { storeToRefs } from 'pinia'

export default defineComponent({

setup() {

const store = useStore()

// `name` and `doubleCount` are reactive refs

// This will also create refs for properties added by plugins

// but skip any action or non reactive (non ref/reactive) property

const { name, doubleCount } = storeToRefs(store)

// the increment action can be just extracted

const { increment } = store

return {

name,

doubleCount,

increment,

}

},

})

## state状态

在 Pinia 中，状态被定义为返回最终状态的功能。这 Pinia 在服务器端和客户端允许工作。

import { defineStore } from 'pinia'

const useStore = defineStore('storeId', {

// arrow function recommended for full type inference

state: () => {

return {

// all these properties will have their type inferred automatically

counter: 0,

name: 'Eduardo',

isAdmin: true,

}

},

})

### 访问state

默认情况下，您可以通过store实例访问状态来直接读取和写入状态：

const store = useStore()

store.counter++

### 重置状态

您可以通过调用store 上的方法将状态重置为其初始值：$reset()

const store = useStore()

store.$reset()

如果您希望能够写入这些状态属性（例如，如果您有一个表单），您可以mapWritableState()改用。请注意，您不能传递类似 with 的函数mapState()：

import { mapWritableState } from 'pinia'

import { useCounterStore } from '../stores/counterStore'

export default {

computed: {

// gives access to this.counter inside the component and allows setting it

// this.counter++

// same as reading from store.counter

...mapWritableState(useCounterStore, ['counter'])

// same as above but registers it as this.myOwnName

...mapWritableState(useCounterStore, {

myOwnName: 'counter',

}),

},

}

### 改变状态

除了直接用 改变 store 之外store.counter++，您还可以调用该$patch方法。state它允许您对部分对象同时应用多个更改：

store.$patch({

counter: store.counter + 1,

name: 'Abalam',

})

但是，使用这种语法应用某些突变确实很难或成本很高：任何集合修改（例如，从数组中推送、删除、拼接元素）都需要您创建一个新集合。正因为如此，该$patch方法还接受一个函数来对这种难以用补丁对象应用的突变进行分组：

cartStore.$patch((state) => {

state.items.push({ name: 'shoes', quantity: 1 })

state.hasChanged = true

})

这里的主要区别是$patch()允许您将多个更改分组到devtools中的一个条目中。请注意，直接更改state并$patch()出现在devtools 中，并且可以移动时间。

### 更换state

$state您可以通过将商店的属性设置为新对象来替换商店的整个状态：

store.$state = { counter: 666, name: 'Paimon' }

您还可以通过更改实例state的来替换应用程序的整个状态

### 订阅状态

你可以通过 store 的方法观察状态及其变化$subscribe()，类似于 Vuex 的subscribe 方法。$subscribe()与常规相比使用的优点watch()是订阅只会在补丁后触发一次（例如，使用上面的函数版本时）。

cartStore.$subscribe((mutation, state) => {

// import { MutationType } from 'pinia'

mutation.type // 'direct' | 'patch object' | 'patch function'

// same as cartStore.$id

mutation.storeId // 'cart'

// only available with mutation.type === 'patch object'

mutation.payload // patch object passed to cartStore.$patch()

// persist the whole state to the local storage whenever it changes

localStorage.setItem('cart', JSON.stringify(state))

})

默认情况下，状态订阅绑定到添加它们的组件（如果存储在组件的内部setup()）。意思是，当组件被卸载时，它们将被自动删除。如果要在组件卸载后保留它们，请{ detached: true }作为第二个参数传递以从当前组件中分离状态订阅：

export default {

setup() {

const someStore = useSomeStore()

// this subscription will be kept after the component is unmounted

someStore.$subscribe(callback, { detached: true })

// ...

},

}

pinia您可以查看实例上的整个状态：

watch(

pinia.state,

(state) => {

// persist the whole state to the local storage whenever it changes

localStorage.setItem('piniaState', JSON.stringify(state))

},

{ deep: true }

)

## Getters

Getter 完全等同于 Store 状态的计算值。它们可以用 中的getters属性定义defineStore()。他们接收state作为第一个参数来鼓励使用箭头函数：

export const useStore = defineStore('main', {

state: () => ({

counter: 0,

}),

getters: {

doubleCount: (state) => state.counter \* 2,

},

})

大多数时候，getter 只会依赖状态，但是，他们可能需要使用其他 getter。因此，我们可以在定义常规函数时访问整个store 实例，但需要定义返回类型的类型（在 TypeScript 中）。这是由于 TypeScript 中的一个已知限制，不会影响使用箭头函数定义的 getter，也不会影响不使用的 getter：thisthis

export const useStore = defineStore('main', {

state: () => ({

counter: 0,

}),

getters: {

// automatically infers the return type as a number

doubleCount(state) {

return state.counter \* 2

},

// the return type \*\*must\*\* be explicitly set

doublePlusOne(): number {

// autocompletion and typings for the whole store ✨

return this.doubleCount + 1

},

},

})

然后你可以直接在 store 实例上访问 getter：

<template>

<p>Double count is {{ store.doubleCount }}</p>

</template>

<script>

export default {

setup() {

const store = useStore()

return { store }

},

}

</script>

### 多个Getter

与计算属性一样，您可以组合多个 getter。通过 访问任何其他Getter的this。即使您不使用 TypeScript，您也可以使用JSDoc提示您的 IDE 类型：

export const useStore = defineStore('main', {

state: () => ({

counter: 0,

}),

getters: {

// type is automatically inferred because we are not using `this`

doubleCount: (state) => state.counter \* 2,

// here we need to add the type ourselves (using JSDoc in JS). We can also

// use this to document the getter

/\*\*

\* Returns the counter value times two plus one.

\*

\* @returns {number}

\*/

doubleCountPlusOne() {

// autocompletion ✨

return this.doubleCount + 1

},

},

})

### 将参数传递给 getter

Getter只是在幕后计算的属性，因此不可能将任何参数传递给它们。但是，您可以从getter返回一个函数以接受任何参数：

export const useStore = defineStore('main', {

getters: {

getUserById: (state) => {

return (userId) => state.users.find((user) => user.id === userId)

},

},

})

并在组件中使用：

<script>

export default {

setup() {

const store = useStore()

return { getUserById: store.getUserById }

},

}

</script>

<template>

<p>User 2: {{ getUserById(2) }}</p>

</template>

请注意，执行此操作时，getter 不再缓存，它们只是您调用的函数。但是，您可以在 getter 本身内部缓存一些结果，这并不常见，但应该证明性能更高：

export const useStore = defineStore('main', {

getters: {

getActiveUserById(state) {

const activeUsers = state.users.filter((user) => user.active)

return (userId) => activeUsers.find((user) => user.id === userId)

},

},

})

### 其他store的getter

要使用其他存储 getter，您可以直接在getter内部使用它：

import { useOtherStore } from './other-store'

export const useStore = defineStore('main', {

state: () => ({

// ...

}),

getters: {

otherGetter(state) {

const otherStore = useOtherStore()

return state.localData + otherStore.data

},

},

})

## action

动作相当于组件中的方法。它们可以使用actionsin 属性进行定义，defineStore()并且非常适合定义业务逻辑：

export const useStore = defineStore('main', {

state: () => ({

counter: 0,

}),

actions: {

increment() {

this.counter++

},

randomizeCounter() {

this.counter = Math.round(100 \* Math.random())

},

},

})

像getter一样，动作通过完全输入（和自动完成✨）支持访问整个store实例。与 getter 不同，它可以是异步的，您可以在动作内部进行任何 API 调用甚至其他动作！这是一个使用Mande的示例。请注意，您使用的库并不重要，只要您获得 a ，您甚至可以使用本机函数（仅限浏览器）：

import { mande } from 'mande'

const api = mande('/api/users')

export const useUsers = defineStore('users', {

state: () => ({

userData: null,

// ...

}),

actions: {

async registerUser(login, password) {

try {

this.userData = await api.post({ login, password })

showTooltip(`Welcome back ${this.userData.name}!`)

} catch (error) {

showTooltip(error)

// let the form component display the error

return error

}

},

},

})

你也可以完全自由地设置你想要的任何参数并返回任何东西。调用动作时，一切都会自动推断！动作像方法一样被调用：

export default defineComponent({

setup() {

const main = useMainStore()

// call the action as a method of the store

main.randomizeCounter()

return {}

},

})

### 访问其他store操作

要使用另一个store，您可以直接在action内部使用它：

import { useAuthStore } from './auth-store'

export const useSettingsStore = defineStore('settings', {

state: () => ({

preferences: null,

// ...

}),

actions: {

async fetchUserPreferences() {

const auth = useAuthStore()

if (auth.isAuthenticated) {

this.preferences = await fetchPreferences()

} else {

throw new Error('User must be authenticated')

}

},

},

})

### 订阅操作

可以用 观察动作及其结果store.$onAction()。传递给它的回调在操作本身之前执行。after处理承诺并允许您在操作解决后执行功能。以类似的方式，onError允许您在操作抛出或拒绝时执行函数。这是一个在运行操作之前和它们解决/拒绝之后记录的示例。

const unsubscribe = someStore.$onAction(

({

name, // name of the action

store, // store instance, same as `someStore`

args, // array of parameters passed to the action

after, // hook after the action returns or resolves

onError, // hook if the action throws or rejects

}) => {

// a shared variable for this specific action call

const startTime = Date.now()

// this will trigger before an action on `store` is executed

console.log(`Start "${name}" with params [${args.join(', ')}].`)

// this will trigger if the action succeeds and after it has fully run.

// it waits for any returned promised

after((result) => {

console.log(

`Finished "${name}" after ${

Date.now() - startTime

}ms.\nResult: ${result}.`

)

})

// this will trigger if the action throws or returns a promise that rejects

onError((error) => {

console.warn(

`Failed "${name}" after ${Date.now() - startTime}ms.\nError: ${error}.`

)

})

}

)

// manually remove the listener

unsubscribe()

默认情况下，操作订阅绑定到添加它们的组件（如果商店位于组件的 内部setup()）。意思是，当组件被卸载时，它们将被自动删除。如果要在卸载组件后保留它们，请true作为第二个参数传递以将操作订阅与当前组件分离：

export default {

setup() {

const someStore = useSomeStore()

// this subscription will be kept after the component is unmounted

someStore.$onAction(callback, true)

// ...

},

}

## 插件

由于低级 API，Pania 商店可以完全扩展。以下是您可以执行的操作列表：

* 向store添加新属性
* 定义store时添加新选项
* 向store添加新方法
* 包装现有方法
* 更改甚至取消操作
* 实现像本地存储这样的副作用
* 仅适用于特定store

插件被添加到 pinia 实例中pinia.use()。最简单的例子是通过返回一个对象为所有store添加一个静态属性：

import { createPinia } from 'pinia'

// add a property named `secret` to every store that is created after this plugin is installed

// this could be in a different file

function SecretPiniaPlugin() {

return { secret: 'the cake is a lie' }

}

const pinia = createPinia()

// give the plugin to pinia

pinia.use(SecretPiniaPlugin)

// in another file

const store = useStore()

store.secret // 'the cake is a lie'

这对于添加全局对象（如路由器、模式或 toast 管理器）很有用。

Pinia 插件是一个函数，可以选择返回要添加到store的属性。它需要一个可选参数，一个context：

export function myPiniaPlugin(context) {

context.pinia // the pinia created with `createPinia()`

context.app // the current app created with `createApp()` (Vue 3 only)

context.store // the store the plugin is augmenting

context.options // the options object defining the store passed to `defineStore()`

// ...

}

然后将此函数传递给piniawith pinia.use()：

pinia.use(myPiniaPlugin)

插件仅适用于**传递给应用后创建**pinia的store，否则将不会被应用。

### 扩充store

您可以通过简单地在插件中返回它们的对象来为每个store添加属性：

pinia.use(() => ({ hello: 'world' }))

您也可以直接在 上设置属性，store但如果可能，请使用返回版本，以便它们可以被 devtools 自动跟踪：

pinia.use(({ store }) => {

store.hello = 'world'

})

插件返回的任何属性都将由 devtools 自动跟踪，因此为了hello在 devtools 中可见，请确保仅当您想在 devtools 中调试它时才将其添加到store.\_customProperties dev 模式：

// from the example above

pinia.use(({ store }) => {

store.hello = 'world'

// make sure your bundler handle this. webpack and vite should do it by default

if (process.env.NODE\_ENV === 'development') {

// add any keys you set on the store

store.\_customProperties.add('hello')

}

})

请注意，每个商店都用 包装reactive，自动解包装它包含的任何 Ref ( ref(), computed(), ...)：

const sharedRef = ref('shared')

pinia.use(({ store }) => {

// each store has its individual `hello` property

store.hello = ref('secret')

// it gets automatically unwrapped

store.hello // 'secret'

// all stores are sharing the value `shared` property

store.shared = sharedRef

store.shared // 'shared'

})

这就是为什么您可以在没有的情况下访问所有计算属性.value以及它们是反应性的原因。

### 添加state

如果您想将新的状态属性添加到商店或打算在作用期间使用的属性，您必须在两个地方添加它：

* 在store所以你可以访问它store.myState
* store.$state因此它可以在 devtools 中使用，**并在SSR 期间被序列化**。

最重要的是，您肯定必须使用ref()（或其他反应式 API）才能在不同的访问中共享值：

import { toRef, ref } from 'vue'

pinia.use(({ store }) => {

// to correctly handle SSR, we need to make sure we are not overriding an

// existing value

if (!Object.prototype.hasOwnProperty(store.$state, 'hasError')) {

// hasError is defined within the plugin, so each store has their individual

// state property

const hasError = ref(false)

// setting the variable on `$state`, allows it be serialized during SSR

store.$state.hasError = hasError

}

// we need to transfer the ref from the state to the store, this way

// both accesses: store.hasError and store.$state.hasError will work

// and share the same variable

// See https://vuejs.org/api/reactivity-utilities.html#toref

store.hasError = toRef(store.$state, 'hasError')

// in this case it's better not to return `hasError` since it

// will be displayed in the `state` section in the devtools

// anyway and if we return it, devtools will display it twice.

})

请注意，插件中发生的状态更改或添加（包括调用store.$patch()）发生在商店处于活动状态之前，因此**不会触发任何订阅**。

### 添加新的外部属性

当添加外部属性、来自其他库的类实例或简单的非反应性事物时，您应该在将对象markRaw()传递给 pinia 之前将其包装起来。这是一个将路由器添加到每个商店的示例：

import { markRaw } from 'vue'

// adapt this based on where your router is

import { router } from './router'

pinia.use(({ store }) => {

store.router = markRaw(router)

})

### $subscribe在插件内部调用

您也可以在插件中使用store.$subscribe和store.$onAction ：

pinia.use(({ store }) => {

store.$subscribe(() => {

// react to store changes

})

store.$onAction(() => {

// react to store actions

})

})

# Vue Router

## 安装

* 直接下载 / CDN：<https://unpkg.com/vue-router@4>
* npm：npm install vue-router@4
* yarn：yarn add vue-router@4

## 入门

简单的例子：

<div id="app">

<h1>Hello App!</h1>

<p>

<!--使用 router-link 组件进行导航 -->

<!--通过传递 `to` 来指定链接 -->

<!--`<router-link>` 将呈现一个带有正确 `href` 属性的 `<a>` 标签-->

<router-link to="/">Go to Home</router-link>

<router-link to="/about">Go to About</router-link>

</p>

<!-- 路由出口 -->

<!-- 路由匹配到的组件将渲染在这里 -->

<router-view></router-view>

</div>

* router-link：没有使用常规的 a 标签，而是使用一个自定义组件 router-link 来创建链接。这使得 Vue Router 可以在不重新加载页面的情况下更改 URL，处理 URL 的生成以及编码。
* router-view：将显示与 url 对应的组件。你可以把它放在任何地方，以适应你的布局。

其他定义路由方式：

// 1. 定义路由组件.

// 也可以从其他文件导入

const Home = { template: '<div>Home</div>' }

const About = { template: '<div>About</div>' }

// 2. 定义一些路由

// 每个路由都需要映射到一个组件。

// 我们后面再讨论嵌套路由。

const routes = [

{ path: '/', component: Home },

{ path: '/about', component: About },

]

// 3. 创建路由实例并传递 `routes` 配置

// 你可以在这里输入更多的配置，但我们在这里

// 暂时保持简单

const router = VueRouter.createRouter({

// 4. 内部提供了 history 模式的实现。为了简单起见，我们在这里使用 hash 模式。

history: VueRouter.createWebHashHistory(),

routes, // `routes: routes` 的缩写

})

// 5. 创建并挂载根实例

const app = Vue.createApp({})

//确保 \_use\_ 路由实例使

//整个应用支持路由。

app.use(router)

app.mount('#app')

通过调用 app.use(router)，我们可以在任意组件中以 this.$router 的形式访问它，并且以 this.$route 的形式访问当前路由：

// Home.vue

export default {

computed: {

username() {

// 我们很快就会看到 `params` 是什么

return this.$route.params.username

},

},

methods: {

goToDashboard() {

if (isAuthenticated) {

this.$router.push('/dashboard')

} else {

this.$router.push('/login')

}

},

},

}

要在 setup 函数中访问路由，请调用 useRouter 或 useRoute 函数。

## 带参数的动态路由匹配

在 Vue Router 中，我们可以在路径中使用一个动态字段来实现，我们称之为 路径参数 ：

import User from "@/User.vue";

import App from '@/App.vue'

// 这些都会传递给 `createRouter`

const routes = [

// 动态字段以冒号开始

{

path: '/',

name: "App",

component: App,

},

{

path: '/User/:id,

name: "User",

component: User,

},

]

const router = createRouter({

history: createWebHashHistory(),

routes,

})

export default router;

现在像 /users/johnny 和 /users/jolyne 这样的 URL 都会映射到同一个路由。路径参数 用冒号 : 表示。当一个路由被匹配时，它的 params 的值将在每个组件中以 this.$route.params 的形式暴露出来。因此，我们可以通过更新 User 的模板来呈现当前的用户 ID。

<template>

<div>User {{ $route.params.id }}</div>

</template>

你可以在同一个路由中设置有多个 路径参数，它们会映射到 $route.params 上的相应字段。例如：

| **匹配模式** | **匹配路径** | **$route.params** |
| --- | --- | --- |
| /users/:username | /users/eduardo | { username: 'eduardo' } |
| /users/:username/posts/:postId | /users/eduardo/posts/123 | { username: 'eduardo', postId: '123' } |

除了 $route.params 之外，$route 对象还公开了其他有用的信息，如 $route.query（如果 URL 中存在参数）、$route.hash 等。

### 参数的变化

使用带有参数的路由时需要注意的是，当用户从 /users/johnny 导航到 /users/jolyne 时，相同的组件实例将被重复使用。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。不过，这也意味着组件的生命周期钩子不会被调用。

要对同一个组件中参数的变化做出响应的话，你可以简单地 watch $route 对象上的任意属性，在这个场景中，就是 $route.params ：

const User = {

template: '...',

created() {

this.$watch(

() => this.$route.params,

(toParams, previousParams) => {

// 对路由变化做出响应...

}

)

},

}

或者，使用 beforeRouteUpdate 导航守卫，它也可以取消导航：

const User = {

template: '...',

async beforeRouteUpdate(to, from) {

// 对路由变化做出响应...

this.userData = await fetchUser(to.params.id)

},

}

### 捕获路由

常规参数只匹配 url 片段之间的字符，用 / 分隔。如果我们想匹配任意路径，我们可以使用自定义的 路径参数 正则表达式，在 路径参数 后面的括号中加入 正则表达式 :

const routes = [

// 将匹配所有内容并将其放在 `$route.params.pathMatch` 下

{ path: '/:pathMatch(.\*)\*', name: 'NotFound', component: NotFound },

// 将匹配以 `/user-` 开头的所有内容，并将其放在 `$route.params.afterUser` 下

{ path: '/user-:afterUser(.\*)', component: UserGeneric },

]

在这个特定的场景中，我们在括号之间使用了自定义正则表达式，并将pathMatch 参数标记为可选可重复。这样做是为了让我们在需要的时候，可以通过将 path 拆分成一个数组，直接导航到路由：

this.$router.push({

name: 'NotFound',

// 保留当前路径并删除第一个字符，以避免目标 URL 以 `//` 开头。

params: { pathMatch: this.$route.path.substring(1).split('/') },

// 保留现有的查询和 hash 值，如果有的话

query: this.$route.query,

hash: this.$route.hash,

})

### 高级匹配模式

Vue Router 使用自己的路径匹配语法，其灵感来自于 express，因此它支持许多高级匹配模式，如可选的参数，零或多个 / 一个或多个，甚至自定义的正则匹配规则。

## 路由的匹配语法

### 在参数中自定义正则

当定义像 :userId 这样的参数时，我们内部使用以下的正则 ([^/]+) (至少有一个字符不是斜杠 / )来从 URL 中提取参数。两个路由 /:orderId 和 /:productName，两者会匹配完全相同的 URL，所以我们需要一种方法来区分它们。最简单的方法就是在路径中添加一个静态部分来区分它们：

const routes = [

// 匹配 /o/3549

{ path: '/o/:orderId' },

// 匹配 /p/books

{ path: '/p/:productName' },

]

由于，orderId 总是一个数字，而 productName 可以是任何东西，所以我们可以在括号中为参数指定一个自定义的正则：

const routes = [

// /:orderId -> 仅匹配数字

{ path: '/:orderId(\\d+)' },

// /:productName -> 匹配其他任何内容

{ path: '/:productName' },

]

确保转义反斜杠( \ )，就像我们对 \d (变成\\d)所做的那样，在 JavaScript 中实际传递字符串中的反斜杠字符。

### 可重复的参数

如果你需要匹配具有多个部分的路由，如 /first/second/third，你应该用 \*（0 个或多个）和 +（1 个或多个）将参数标记为可重复：

const routes = [

// /:chapters -> 匹配 /one, /one/two, /one/two/three, 等

{ path: '/:chapters+' },

// /:chapters -> 匹配 /, /one, /one/two, /one/two/three, 等

{ path: '/:chapters\*' },

]

这将为你提供一个参数数组，而不是一个字符串，并且在使用命名路由时也需要你传递一个数组：

// 给定 { path: '/:chapters\*', name: 'chapters' },

router.resolve({ name: 'chapters', params: { chapters: [] } }).href

// 产生 /

router.resolve({ name: 'chapters', params: { chapters: ['a', 'b'] } }).href

// 产生 /a/b

// 给定 { path: '/:chapters+', name: 'chapters' },

router.resolve({ name: 'chapters', params: { chapters: [] } }).href

// 抛出错误，因为 `chapters` 为空

这些也可以通过在右括号后添加它们与自定义正则结合使用：

const routes = [

// 仅匹配数字

// 匹配 /1, /1/2, 等

{ path: '/:chapters(\\d+)+' },

// 匹配 /, /1, /1/2, 等

{ path: '/:chapters(\\d+)\*' },

]

### Sensitive 与 strict 路由配置

默认情况下，所有路由是不区分大小写的，并且能匹配带有或不带有尾部斜线的路由。例如，路由 /users 将匹配 /users、/users/、甚至 /Users/。这种行为可以通过 strict 和 sensitive 选项来修改，它们可以既可以应用在整个全局路由上，又可以应用于当前路由上：

const router = createRouter({

history: createWebHistory(),

routes: [

// 将匹配 /users/posva 而非：

// - /users/posva/ 当 strict: true

// - /Users/posva 当 sensitive: true

{ path: '/users/:id', sensitive: true },

// 将匹配 /users, /Users, 以及 /users/42 而非 /users/ 或 /users/42/

{ path: '/users/:id?' },

]

strict: true, // applies to all routes

})

### 可选参数

你也可以通过使用 ? 修饰符(0 个或 1 个)将一个参数标记为可选：

const routes = [

// 匹配 /users 和 /users/posva

{ path: '/users/:userId?' },

// 匹配 /users 和 /users/42

{ path: '/users/:userId(\\d+)?' },

]

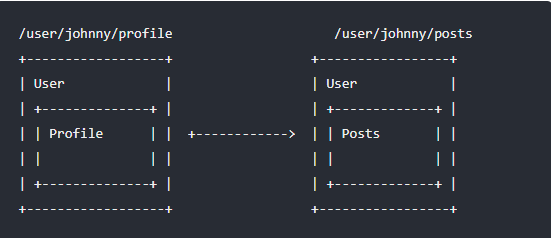
请注意，\* 在技术上也标志着一个参数是可选的，但 ? 参数不能重复。

### 调试

[路径排名工具](https://paths.esm.dev/?p=AAMeJSyAwR4UbFDAFxAcAGAIJXMAAA..)

## 嵌套路由

一些应用程序的 UI 由多层嵌套的组件组成。在这种情况下，URL 的片段通常对应于特定的嵌套组件结构，例如：



通过 Vue Router，你可以使用嵌套路由配置来表达这种关系。接着上节创建的 app ：

<div id="app">

<router-view></router-view>

</div>

const User = {

template: '<div>User {{ $route.params.id }}</div>',

}

// 这些都会传递给 `createRouter`

const routes = [{ path: '/user/:id', component: User }]

这里的 <router-view> 是一个顶层的 router-view。它渲染顶层路由匹配的组件。同样地，一个被渲染的组件也可以包含自己嵌套的 <router-view>。例如，如果我们在 User 组件的模板内添加一个 <router-view>：

const User = {

template: `

<div class="user">

<h2>User {{ $route.params.id }}</h2>

<router-view></router-view>

</div>

`,

}

要将组件渲染到这个嵌套的 router-view 中，我们需要在路由中配置 children：

const routes = [

{

path: '/user/:id',

component: User,

children: [

{

// 当 /user/:id/profile 匹配成功

// UserProfile 将被渲染到 User 的 <router-view> 内部

path: 'profile',

component: UserProfile,

},

{

// 当 /user/:id/posts 匹配成功

// UserPosts 将被渲染到 User 的 <router-view> 内部

path: 'posts',

component: UserPosts,

},

],

},

]

注意，以 / 开头的嵌套路径将被视为根路径。这允许你利用组件嵌套，而不必使用嵌套的 URL。

children 配置只是另一个路由数组，就像 routes 本身一样。因此，你可以根据自己的需要，不断地嵌套视图。按照上面的配置，当你访问 /user/eduardo 时，在 User 的 router-view 里面什么都不会呈现，因为没有匹配到嵌套路由。也许你确实想在那里渲染一些东西。在这种情况下，你可以提供一个空的嵌套路径：

const routes = [

{

path: '/user/:id',

component: User,

children: [

// 当 /user/:id 匹配成功

// UserHome 将被渲染到 User 的 <router-view> 内部

{ path: '', component: UserHome },

// ...其他子路由

],

},

]

## 编程式导航

除了使用 <router-link> 创建 a 标签来定义导航链接，我们还可以借助 router 的实例方法，通过编写代码来实现。

### 导航到不同的位置

想要导航到不同的 URL，可以使用 router.push 方法。这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，会回到之前的 URL。当你点击 <router-link> 时，内部会调用这个方法，所以点击 <router-link :to="..."> 相当于调用 router.push(...) ：

| **声明式** | **编程式** |
| --- | --- |
| <router-link :to="..."> | router.push(...) |

该方法的参数可以是一个字符串路径，或者一个描述地址的对象。例如：

// 字符串路径

router.push('/users/eduardo')

// 带有路径的对象

router.push({ path: '/users/eduardo' })

// 命名的路由，并加上参数，让路由建立 url

router.push({ name: 'user', params: { username: 'eduardo' } })

// 带查询参数，结果是 /register?plan=private

router.push({ path: '/register', query: { plan: 'private' } })

// 带 hash，结果是 /about#team

router.push({ path: '/about', hash: '#team' })

如果提供了 path，params 会被忽略，上述例子中的 query 并不属于这种情况。取而代之的是下面例子的做法，你需要提供路由的 name 或手写完整的带有参数的 path ：

const username = 'eduardo'

// 我们可以手动建立 url，但我们必须自己处理编码

router.push(`/user/${username}`) // -> /user/eduardo

// 同样

router.push({ path: `/user/${username}` }) // -> /user/eduardo

// 如果可能的话，使用 `name` 和 `params` 从自动 URL 编码中获益

router.push({ name: 'user', params: { username } }) // -> /user/eduardo

// `params` 不能与 `path` 一起使用

router.push({ path: '/user', params: { username } }) // -> /user

当指定 params 时，可提供 string 或 number 参数（或者对于可重复的参数可提供一个数组）。任何其他类型（如 undefined、false 等）都将被自动字符串化。对于可选参数，你可以提供一个空字符串（""）来跳过它。

由于属性 to 与 router.push 接受的对象种类相同，所以两者的规则完全相同。router.push 和所有其他导航方法都会返回一个 Promise，让我们可以等到导航完成后才知道是成功还是失败。

### 替换当前位置

它的作用类似于 router.push，唯一不同的是，它在导航时不会向 history 添加新记录，正如它的名字所暗示的那样——它取代了当前的条目。

| **声明式** | **编程式** |
| --- | --- |
| <router-link :to="..." replace> | router.replace(...) |

也可以直接在传递给 router.push 的 routeLocation 中增加一个属性 replace: true ：

router.push({ path: '/home', replace: true })

// 相当于

router.replace({ path: '/home' })

### 横跨历史

该方法采用一个整数作为参数，表示在历史堆栈中前进或后退多少步，类似于 window.history.go(n)。例子：

// 向前移动一条记录，与 router.forward() 相同

router.go(1)

// 返回一条记录，与 router.back() 相同

router.go(-1)

// 前进 3 条记录

router.go(3)

// 如果没有那么多记录，静默失败

router.go(-100)

router.go(100)

## 命名路由

除了 path 之外，你还可以为任何路由提供 name。这有以下优点：

* 没有硬编码的 URL
* params 的自动编码/解码。
* 防止你在 url 中出现打字错误。
* 绕过路径排序（如显示一个）

const routes = [

{

path: '/user/:username',

name: 'user',

component: User

}

]

要链接到一个命名的路由，可以向 router-link 组件的 to 属性传递一个对象：

<router-link :to="{ name: 'user', params: { username: 'erina' }}">

User

</router-link>

这跟代码调用 router.push() 是一回事：

router.push({ name: 'user', params: { username: 'erina' } })

在这两种情况下，路由将导航到路径 /user/erina。

## 命名视图

有时候想同时 (同级) 展示多个视图，而不是嵌套展示，例如创建一个布局，有 sidebar (侧导航) 和 main (主内容) 两个视图，这个时候命名视图就派上用场了。你可以在界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。

<router-view class="view left-sidebar" name="LeftSidebar"></router-view>

<router-view class="view main-content"></router-view>

<router-view class="view right-sidebar" name="RightSidebar"></router-view>

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components 配置 (带上 s)：

const router = createRouter({

history: createWebHashHistory(),

routes: [

{

path: '/',

components: {

default: Home,

// LeftSidebar: LeftSidebar 的缩写

LeftSidebar,

// 它们与 `<router-view>` 上的 `name` 属性匹配

RightSidebar,

},

},

],

})

我们也有可能使用命名视图创建嵌套视图的复杂布局。这时你也需要命名用到的嵌套 router-view 组件。我们以一个设置面板为例：



* Nav 只是一个常规组件。
* UserSettings 是一个视图组件。
* UserEmailsSubscriptions、UserProfile、UserProfilePreview 是嵌套的视图组件。

UserSettings 组件的 <template> 部分应该是类似下面的这段代码:

<!-- UserSettings.vue -->

<div>

<h1>User Settings</h1>

<NavBar />

<router-view />

<router-view name="helper" />

</div>

那么你就可以通过这个路由配置来实现上面的布局：

{

path: '/settings',

// 你也可以在顶级路由就配置命名视图

component: UserSettings,

children: [{

path: 'emails',

component: UserEmailsSubscriptions

}, {

path: 'profile',

components: {

default: UserProfile,

helper: UserProfilePreview

}

}]

}

## 重定向和别名

### 重定向

重定向也是通过 routes 配置来完成，下面例子是从 /home 重定向到 /：

const routes = [{ path: '/home', redirect: '/' }]

重定向的目标也可以是一个命名的路由：

const routes = [{ path: '/home', redirect: { name: 'homepage' } }]

甚至是一个方法，动态返回重定向目标：

const routes = [

{

// /search/screens -> /search?q=screens

path: '/search/:searchText',

redirect: to => {

// 方法接收目标路由作为参数

// return 重定向的字符串路径/路径对象

return { path: '/search', query: { q: to.params.searchText } }

},

},

{

path: '/search',

// ...

},

]

请注意，导航守卫并没有应用在跳转路由上，而仅仅应用在其目标上。在上面的例子中，在 /home 路由中添加 beforeEnter 守卫不会有任何效果。

在写 redirect 的时候，可以省略 component 配置，因为它从来没有被直接访问过，所以没有组件要渲染。唯一的例外是嵌套路由：如果一个路由记录有 children 和 redirect 属性，它也应该有 component 属性。

也可以重定向到相对位置：

const routes = [

{

// 将总是把/users/123/posts重定向到/users/123/profile。

path: '/users/:id/posts',

redirect: to => {

// 该函数接收目标路由作为参数

// 相对位置不以`/`开头

// 或 { path: 'profile'}

return 'profile'

},

},

]

### 别名

重定向是指当用户访问 /home 时，URL 会被 / 替换，然后匹配成 /。别名是 别名为 /home，意味着当用户访问 /home 时，URL 仍然是 /home，但会被匹配为用户正在访问 /。上面对应的路由配置为：

const routes = [{ path: '/', component: Homepage, alias: '/home' }]

通过别名，你可以自由地将 UI 结构映射到一个任意的 URL，而不受配置的嵌套结构的限制。使别名以 / 开头，以使嵌套路径中的路径成为绝对路径。你甚至可以将两者结合起来，用一个数组提供多个别名：

const routes = [

{

path: '/users',

component: UsersLayout,

children: [

// 为这 3 个 URL 呈现 UserList

// - /users

// - /users/list

// - /people

{ path: '', component: UserList, alias: ['/people', 'list'] },

],

},

]

如果你的路由有参数，请确保在任何绝对别名中包含它们：

const routes = [

{

path: '/users/:id',

component: UsersByIdLayout,

children: [

// 为这 3 个 URL 呈现 UserDetails

// - /users/24

// - /users/24/profile

// - /24

{ path: 'profile', component: UserDetails, alias: ['/:id', ''] },

],

},

]

## 路由组件传参

在你的组件中使用 $route 会与路由紧密耦合，这限制了组件的灵活性，因为它只能用于特定的 URL。虽然这不一定是件坏事，但我们可以通过 props 配置来解除这种行为：

const User = {

template: '<div>User {{ $route.params.id }}</div>'

}

const routes = [{ path: '/user/:id', component: User }]

替换成：

const User = {

// 请确保添加一个与路由参数完全相同的 prop 名

props: ['id'],

template: '<div>User {{ id }}</div>'

}

const routes = [{ path: '/user/:id', component: User, props: true }]

### 布尔模式

当 props 设置为 true 时，route.params 将被设置为组件的 props。

### 命名视图

对于有命名视图的路由，你必须为每个命名视图定义 props 配置：

const routes = [

{

path: '/user/:id',

components: { default: User, sidebar: Sidebar },

props: { default: true, sidebar: false }

}

]

### 对象模式

当 props 是一个对象时，它将原样设置为组件 props。当 props 是静态的时候很有用。

const routes = [

{

path: '/promotion/from-newsletter',

component: Promotion,

props: { newsletterPopup: false }

}

]

### 函数模式

你可以创建一个返回 props 的函数。这允许你将参数转换为其他类型，将静态值与基于路由的值相结合等等。

const routes = [

{

path: '/search',

component: SearchUser,

props: route => ({ query: route.query.q })

}

]

URL /search?q=vue 将传递 {query: 'vue'} 作为 props 传给 SearchUser 组件。请尽可能保持 props 函数为无状态的，因为它只会在路由发生变化时起作用。如果你需要状态来定义 props，请使用包装组件，这样 vue 才可以对状态变化做出反应。

## 历史模式

在创建路由器实例时，history 配置允许我们在不同的历史模式中进行选择。

### HASH模式

hash 模式是用 createWebHashHistory() 创建的：

import { createRouter, createWebHashHistory } from 'vue-router'

const router = createRouter({

history: createWebHashHistory(),

routes: [

//...

],

})

它在内部传递的实际 URL 之前使用了一个哈希字符（#）。由于这部分 URL 从未被发送到服务器，所以它不需要在服务器层面上进行任何特殊处理。不过，它在 SEO 中确实有不好的影响。如果你担心这个问题，可以使用 HTML5 模式。

### HTML5 模式

用 createWebHistory() 创建 HTML5 模式，推荐使用这个模式：

import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({

history: createWebHistory(),

routes: [

//...

],

})

当使用这种历史模式时，URL 会看起来很 "正常"，例如 https://example.com/user/id。漂亮!由于我们的应用是一个单页的客户端应用，如果没有适当的服务器配置，用户在浏览器中直接访问 https://example.com/user/id，就会得到一个 404 错误。这就丑了。

不用担心：要解决这个问题，你需要做的就是在你的服务器上添加一个简单的回退路由。如果 URL 不匹配任何静态资源，它应提供与你的应用程序中的 index.html 相同的页面。漂亮依旧!

### 服务器配置

以下示例假定你正在从根目录提供服务。如果你部署到子目录，你应该使用Vue CLI 的 publicPath 配置和相关的路由器的 base 属性。你还需要调整下面的例子，以使用子目录而不是根目录（例如，将RewriteBase/ 替换为 RewriteBase/name-of-your-subfolder/）。

#### Apache

<IfModule mod\_negotiation.c>

Options -MultiViews

</IfModule>

<IfModule mod\_rewrite.c>

RewriteEngine On

RewriteBase /

RewriteRule ^index\.html$ - [L]

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule . /index.html [L]

</IfModule>

也可以使用 FallbackResource 代替 mod\_rewrite。

#### nginx

location / {

try\_files $uri $uri/ /index.html;

}

#### Node.js

const http = require('http')

const fs = require('fs')

const httpPort = 80

http

.createServer((req, res) => {

fs.readFile('index.html', 'utf-8', (err, content) => {

if (err) {

console.log('We cannot open "index.html" file.')

}

res.writeHead(200, {

'Content-Type': 'text/html; charset=utf-8',

})

res.end(content)

})

})

.listen(httpPort, () => {

console.log('Server listening on: http://localhost:%s', httpPort)

})

#### Express + Node.js

对于 Node.js/Express，可以考虑使用 connect-history-api-fallback 中间件。

#### Internet Information Services (IIS)

安装 IIS UrlRewrite。在网站的根目录下创建一个 web.config 文件，内容如下：

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<system.webServer>

<rewrite>

<rules>

<rule name="Handle History Mode and custom 404/500" stopProcessing="true">

<match url="(.\*)" />

<conditions logicalGrouping="MatchAll">

<add input="{REQUEST\_FILENAME}" matchType="IsFile" negate="true" />

<add input="{REQUEST\_FILENAME}" matchType="IsDirectory" negate="true" />

</conditions>

<action type="Rewrite" url="/" />

</rule>

</rules>

</rewrite>

</system.webServer>

</configuration>

Caddy v2#

try\_files {path} /

Caddy v1#

rewrite {

regexp .\*

to {path} /

}

#### Firebase hosting

将此添加到你的 firebase.json 中：

{

"hosting": {

"public": "dist",

"rewrites": [

{

"source": "\*\*",

"destination": "/index.html"

}

]

}

}

#### Netlify

创建一个 \_redirects 文件，包含在你的部署文件中：

/\* /index.html 200

在 vue-cli、nuxt 和 vite 项目中，这个文件通常放在名为 static 或 public 的目录下。你可以在 Netlify 文档中找到更多关于语法的信息。你也可以创建一个 netlify.toml 来结合其他 Netlify 功能的重定向。

#### Vercel

在项目根目录创建一个vercel.json文件，内容如下：

{

"rewrites": [{ "source": "/:path\*", "destination": "/index.html" }]

}

#### Caveat

这有一个注意事项。你的服务器将不再报告 404 错误，因为现在所有未找到的路径都会显示你的 index.html 文件。为了解决这个问题，你应该在你的 Vue 应用程序中实现一个万能的路由来显示 404 页面。

const router = createRouter({

history: createWebHistory(),

routes: [{ path: '/:pathMatch(.\*)', component: NotFoundComponent }],

})

另外，如果你使用的是 Node.js 服务器，你可以通过在服务器端使用路由器来匹配传入的 URL，如果没有匹配到路由，则用 404 来响应，从而实现回退。

## 导航守卫

vue-router 提供的导航守卫主要用来通过跳转或取消的方式守卫导航。这里有很多方式植入路由导航中：全局的，单个路由独享的，或者组件级的。

### 全局前置守卫

你可以使用 router.beforeEach 注册一个全局前置守卫：

const router = createRouter({ ... })

router.beforeEach((to, from) => {

// ...

// 返回 false 以取消导航

return false

})

当一个导航触发时，全局前置守卫按照创建顺序调用。守卫是异步解析执行，此时导航在所有守卫 resolve 完之前一直处于等待中。每个守卫方法接收两个参数：

* to: 即将要进入的目标 用一种标准化的方式
* from: 当前导航正要离开的路由 用一种标准化的方式

可以返回的值如下:

* false: 取消当前的导航。如果浏览器的 URL 改变了(可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 from 路由对应的地址。
* 一个路由地址: 通过一个路由地址跳转到一个不同的地址，就像你调用 router.push() 一样，你可以设置诸如 replace: true 或 name: 'home' 之类的配置。当前的导航被中断，然后进行一个新的导航，就和 from 一样。

router.beforeEach(async (to, from) => {

if (

// 检查用户是否已登录

!isAuthenticated &&

// ❗️ 避免无限重定向

to.name !== 'Login'

) {

// 将用户重定向到登录页面

return { name: 'Login' }

}

})

如果遇到了意料之外的情况，可能会抛出一个 Error。这会取消导航并且调用 router.onError() 注册过的回调。如果什么都没有，undefined 或返回 true，则导航是有效的，并调用下一个导航守卫以上所有都同 async 函数 和 Promise 工作方式一样：

router.beforeEach(async (to, from) => {

// canUserAccess() 返回 `true` 或 `false`

const canAccess = await canUserAccess(to)

if (!canAccess) return '/login'

})

### 全局解析守卫

你可以用 router.beforeResolve 注册一个全局守卫。这和 router.beforeEach 类似，因为它在 每次导航时都会触发，但是确保在导航被确认之前，**同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被正确调用**。这里有一个例子，确保用户可以访问自定义 meta 属性 requiresCamera 的路由：

router.beforeResolve(async to => {

if (to.meta.requiresCamera) {

try {

await askForCameraPermission()

} catch (error) {

if (error instanceof NotAllowedError) {

// ... 处理错误，然后取消导航

return false

} else {

// 意料之外的错误，取消导航并把错误传给全局处理器

throw error

}

}

}

})

router.beforeResolve 是获取数据或执行任何其他操作（如果用户无法进入页面时你希望避免执行的操作）的理想位置。

### 全局后置钩子

你也可以注册全局后置钩子，然而和守卫不同的是，这些钩子不会接受 next 函数也不会改变导航本身：

router.afterEach((to, from) => {

sendToAnalytics(to.fullPath)

})

它们对于分析、更改页面标题、声明页面等辅助功能以及许多其他事情都很有用。它们也反映了 navigation failures 作为第三个参数：

router.afterEach((to, from, failure) => {

if (!failure) sendToAnalytics(to.fullPath)

})

### 路由独享的守卫

你可以直接在路由配置上定义 beforeEnter 守卫：

const routes = [

{

path: '/users/:id',

component: UserDetails,

beforeEnter: (to, from) => {

// reject the navigation

return false

},

},

]

beforeEnter 守卫 只在进入路由时触发，不会在 params、query 或 hash 改变时触发。它们只有在 从一个不同的 路由导航时，才会被触发。你也可以将一个函数数组传递给 beforeEnter，这在为不同的路由重用守卫时很有用：

function removeQueryParams(to) {

if (Object.keys(to.query).length)

return { path: to.path, query: {}, hash: to.hash }

}

function removeHash(to) {

if (to.hash) return { path: to.path, query: to.query, hash: '' }

}

const routes = [

{

path: '/users/:id',

component: UserDetails,

beforeEnter: [removeQueryParams, removeHash],

},

{

path: '/about',

component: UserDetails,

beforeEnter: [removeQueryParams],

},

]

### 组件内的守卫

最后，你可以在路由组件内直接定义路由导航守卫(传递给路由配置的)，你可以为路由组件添加以下配置：

* beforeRouteEnter
* beforeRouteUpdate
* beforeRouteLeave

const UserDetails = {

template: `...`,

beforeRouteEnter(to, from) {

// 在渲染该组件的对应路由被验证前调用

// 不能获取组件实例 `this` ！

// 因为当守卫执行时，组件实例还没被创建！

},

beforeRouteUpdate(to, from) {

// 在当前路由改变，但是该组件被复用时调用

// 举例来说，对于一个带有动态参数的路径 `/users/:id`，在 `/users/1` 和 `/users/2` 之间跳转的时候，

// 由于会渲染同样的 `UserDetails` 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。

// 因为在这种情况发生的时候，组件已经挂载好了，导航守卫可以访问组件实例 `this`

},

beforeRouteLeave(to, from) {

// 在导航离开渲染该组件的对应路由时调用

// 与 `beforeRouteUpdate` 一样，它可以访问组件实例 `this`

},

}

### 完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用 beforeRouteLeave 守卫。
3. 调用全局的 beforeEach 守卫。
4. 在重用的组件里调用 beforeRouteUpdate 守卫(2.2+)。
5. 在路由配置里调用 beforeEnter。
6. 解析异步路由组件。
7. 在被激活的组件里调用 beforeRouteEnter。
8. 调用全局的 beforeResolve 守卫(2.5+)。
9. 导航被确认。
10. 调用全局的 afterEach 钩子。
11. 触发 DOM 更新。
12. 调用 beforeRouteEnter 守卫中传给 next 的回调函数，创建好的组件实例会作为回调函数的参数传入。

## 路由元信息

可能希望将任意信息附加到路由上，如过渡名称、谁可以访问路由等。这些事情可以通过接收属性对象的meta属性来实现，并且它可以在路由地址和导航守卫上都被访问到。定义路由的时候你可以这样配置 meta 字段：

const routes = [

{

path: '/posts',

component: PostsLayout,

children: [

{

path: 'new',

component: PostsNew,

// 只有经过身份验证的用户才能创建帖子

meta: { requiresAuth: true }

},

{

path: ':id',

component: PostsDetail

// 任何人都可以阅读文章

meta: { requiresAuth: false }

}

]

}

]

一个路由匹配到的所有路由记录会暴露为 $route 对象(还有在导航守卫中的路由对象)的$route.matched 数组。我们需要遍历这个数组来检查路由记录中的 meta 字段，但是 Vue Router 还为你提供了一个 $route.meta 方法，它是一个非递归合并所有 meta 字段的（从父字段到子字段）的方法。

router.beforeEach((to, from) => {

// 而不是去检查每条路由记录

// to.matched.some(record => record.meta.requiresAuth)

if (to.meta.requiresAuth && !auth.isLoggedIn()) {

// 此路由需要授权，请检查是否已登录

// 如果没有，则重定向到登录页面

return {

path: '/login',

// 保存我们所在的位置，以便以后再来

query: { redirect: to.fullPath },

}

}

})

可以通过扩展 RouteMeta 接口来输入 meta 字段：

// typings.d.ts or router.ts

import 'vue-router'

declare module 'vue-router' {

interface RouteMeta {

// 是可选的

isAdmin?: boolean

// 每个路由都必须声明

requiresAuth: boolean

}

}

## 数据获取

在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现：

* **导航完成之后获取**：先完成导航，然后在接下来的组件生命周期钩子中获取数据。在数据获取期间显示“加载中”之类的指示。
* **导航完成之前获取**：导航完成前，在路由进入的守卫中获取数据，在数据获取成功后执行导航。

### 导航完成后获取数据

当你使用这种方式时，我们会马上导航和渲染组件，然后在组件的 created 钩子中获取数据。这让我们有机会在数据获取期间展示一个 loading 状态，还可以在不同视图间展示不同的 loading 状态。假设我们有一个 Post 组件，需要基于 $route.params.id 获取文章数据：

<template>

<div class="post">

<div v-if="loading" class="loading">Loading...</div>

<div v-if="error" class="error">{{ error }}</div>

<div v-if="post" class="content">

<h2>{{ post.title }}</h2>

<p>{{ post.body }}</p>

</div>

</div>

</template>

export default {

data() {

return {

loading: false,

post: null,

error: null,

}

},

created() {

// watch 路由的参数，以便再次获取数据

this.$watch(

() => this.$route.params,

() => {

this.fetchData()

},

// 组件创建完后获取数据，

// 此时 data 已经被 observed 了

{ immediate: true }

)

},

methods: {

fetchData() {

this.error = this.post = null

this.loading = true

// replace `getPost` with your data fetching util / API wrapper

getPost(this.$route.params.id, (err, post) => {

this.loading = false

if (err) {

this.error = err.toString()

} else {

this.post = post

}

})

},

},

}

### 在导航完成前获取数据

通过这种方式，我们在导航转入新的路由前获取数据。我们可以在接下来的组件的 beforeRouteEnter 守卫中获取数据，当数据获取成功后只调用 next 方法：

export default {

data() {

return {

post: null,

error: null,

}

},

beforeRouteEnter(to, from, next) {

getPost(to.params.id, (err, post) => {

next(vm => vm.setData(err, post))

})

},

// 路由改变前，组件就已经渲染完了

// 逻辑稍稍不同

async beforeRouteUpdate(to, from) {

this.post = null

try {

this.post = await getPost(to.params.id)

} catch (error) {

this.error = error.toString()

}

},

}

在为后面的视图获取数据时，用户会停留在当前的界面，因此建议在数据获取期间，显示一些进度条或者别的指示。如果数据获取失败，同样有必要展示一些全局的错误提醒。

## 组合式 API

### 在 setup 中访问路由和当前路由

因为我们在 setup 里面没有访问 this，所以我们不能再直接访问 this.$router 或 this.$route。作为替代，我们使用 useRouter 函数：

import { useRouter, useRoute } from 'vue-router'

export default {

setup() {

const router = useRouter()

const route = useRoute()

function pushWithQuery(query) {

router.push({

name: 'search',

query: {

...route.query,

},

})

}

},

}

route 对象是一个响应式对象，所以它的任何属性都可以被监听，但你应该避免监听整个 route 对象。在大多数情况下，你应该直接监听你期望改变的参数。

import { useRoute } from 'vue-router'

import { ref, watch } from 'vue'

export default {

setup() {

const route = useRoute()

const userData = ref()

// 当参数更改时获取用户信息

watch(

() => route.params.id,

async newId => {

userData.value = await fetchUser(newId)

}

)

},

}

### 导航守卫

虽然你仍然可以通过 setup 函数来使用组件内的导航守卫，但 Vue Router 将更新和离开守卫作为 组合式 API 函数公开：

import { onBeforeRouteLeave, onBeforeRouteUpdate } from 'vue-router'

import { ref } from 'vue'

export default {

setup() {

// 与 beforeRouteLeave 相同，无法访问 `this`

onBeforeRouteLeave((to, from) => {

const answer = window.confirm(

'Do you really want to leave? you have unsaved changes!'

)

// 取消导航并停留在同一页面上

if (!answer) return false

})

const userData = ref()

// 与 beforeRouteUpdate 相同，无法访问 `this`

onBeforeRouteUpdate(async (to, from) => {

//仅当 id 更改时才获取用户，例如仅 query 或 hash 值已更改

if (to.params.id !== from.params.id) {

userData.value = await fetchUser(to.params.id)

}

})

},

}

组合式 API 守卫也可以用在任何由 <router-view> 渲染的组件中，它们不必像组件内守卫那样直接用在路由组件上。

Vue Router 将 RouterLink 的内部行为作为一个组合式 API 函数公开。它提供了与 v-slot API 相同的访问属性：

import { RouterLink, useLink } from 'vue-router'

import { computed } from 'vue'

export default {

name: 'AppLink',

props: {

// 如果使用 TypeScript，请添加 @ts-ignore

...RouterLink.props,

inactiveClass: String,

},

setup(props) {

const { route, href, isActive, isExactActive, navigate } = useLink(props)

const isExternalLink = computed(

() => typeof props.to === 'string' && props.to.startsWith('http')

)

return { isExternalLink, href, navigate, isActive }

},

}

## 过渡动效

想要在你的路径组件上使用转场，并对导航进行动画处理，你需要使用 v-slot API：

<router-view v-slot="{ Component }">

<transition name="fade">

<component :is="Component" />

</transition>

</router-view>

### 单个路由的过渡

上面的用法会对所有的路由使用相同的过渡。如果你想让每个路由的组件有不同的过渡，你可以将元信息和动态的 name 结合在一起，放在<transition> 上：

const routes = [

{

path: '/custom-transition',

component: PanelLeft,

meta: { transition: 'slide-left' },

},

{

path: '/other-transition',

component: PanelRight,

meta: { transition: 'slide-right' },

},

]

<router-view v-slot="{ Component, route }">

<!-- 使用任何自定义过渡和回退到 `fade` -->

<transition :name="route.meta.transition || 'fade'">

<component :is="Component" />

</transition>

</router-view>

### 基于路由的动态过渡

也可以根据目标路由和当前路由之间的关系，动态地确定使用的过渡。使用和刚才非常相似的片段：

<!-- 使用动态过渡名称 -->

<router-view v-slot="{ Component, route }">

<transition :name="route.meta.transition">

<component :is="Component" />

</transition>

</router-view>

我们可以添加一个 after navigation hook，根据路径的深度动态添加信息到 meta 字段。

router.afterEach((to, from) => {

const toDepth = to.path.split('/').length

const fromDepth = from.path.split('/').length

to.meta.transitionName = toDepth < fromDepth ? 'slide-right' : 'slide-left'

})

### 强制在复用的视图之间进行过渡

Vue 可能会自动复用看起来相似的组件，从而忽略了任何过渡。幸运的是，可以添加一个 key 属性来强制过渡。这也允许你在相同路由上使用不同的参数触发过渡：

<router-view v-slot="{ Component, route }">

<transition name="fade">

<component :is="Component" :key="route.path" />

</transition>

</router-view>

## 滚动行为

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。 vue-router 能做到，而且更好，它让你可以自定义路由切换时页面如何滚动。注意: 这个功能只在支持 history.pushState 的浏览器中可用。

当创建一个 Router 实例，你可以提供一个 scrollBehavior 方法：

const router = createRouter({

history: createWebHashHistory(),

routes: [...],

scrollBehavior (to, from, savedPosition) {

// return 期望滚动到哪个的位置

}

})

scrollBehavior 函数接收 to和 from 路由对象，如 Navigation Guards。第三个参数 savedPosition，只有当这是一个 popstate 导航时才可用（由浏览器的后退/前进按钮触发）。该函数可以返回一个 ScrollToOptions 位置对象:

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

// 始终滚动到顶部

return { top: 0 }

},

})

你也可以通过 el 传递一个 CSS 选择器或一个 DOM 元素。在这种情况下，top 和 left 将被视为该元素的相对偏移量。

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

// 始终在元素 #main 上方滚动 10px

return {

// 也可以这么写

// el: document.getElementById('main'),

el: '#main',

top: -10,

}

},

})

如果返回一个 falsy 的值，或者是一个空对象，那么不会发生滚动。返回 savedPosition，在按下后退/前进按钮时，就会像浏览器的原生表现那样：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (savedPosition) {

return savedPosition

} else {

return { top: 0 }

}

},

})

如果你要模拟 “滚动到锚点” 的行为：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (to.hash) {

return {

el: to.hash,

}

}

},

})

如果你的浏览器支持滚动行为，你可以让它变得更流畅：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (to.hash) {

return {

el: to.hash,

behavior: 'smooth',

}

}

}

})

有时候，我们需要在页面中滚动之前稍作等待。例如，当处理过渡时，我们希望等待过渡结束后再滚动。要做到这一点，你可以返回一个 Promise，它可以返回所需的位置描述符。下面是一个例子，我们在滚动前等待 500ms：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve({ left: 0, top: 0 })

}, 500)

})

},

})

可以将其与页面级过渡组件的事件挂钩，以使滚动行为与你的页面过渡很好地结合起来，但由于使用场景可能存在的差异和复杂性，只是提供了这个基础来实现特定的用户场景。

## 路由懒加载

当打包构建应用时，JavaScript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就会更加高效。Vue Router 支持开箱即用的动态导入，这意味着你可以用动态导入代替静态导入：

// 将

// import UserDetails from './views/UserDetails'

// 替换成

const UserDetails = () => import('./views/UserDetails')

const router = createRouter({

// ...

routes: [{ path: '/users/:id', component: UserDetails }],

})

component (和 components) 配置接收一个返回 Promise 组件的函数，Vue Router 只会在第一次进入页面时才会获取这个函数，然后使用缓存数据。这意味着你也可以使用更复杂的函数，只要它们返回一个 Promise ：

const UserDetails = () =>

Promise.resolve({

/\* 组件定义 \*/

})

一般来说，对所有的路由都使用动态导入是个好主意。

* 使用 webpack：把某个路由下的所有组件都打包在同个异步块 (chunk) 中。只需要使用命名 chunk，一个特殊的注释语法来提供 chunk name (需要 Webpack > 2.4)：

const UserDetails = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserDetails.vue')

const UserDashboard = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserDashboard.vue')

const UserProfileEdit = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserProfileEdit.vue')

* 使用 Vite：在Vite中，你可以在rollupOptions下定义分块：

// vite.config.js

export default defineConfig({

build: {

rollupOptions: {

// https://rollupjs.org/guide/en/#outputmanualchunks

output: {

manualChunks: {

'group-user': [

'./src/UserDetails',

'./src/UserDashboard',

'./src/UserProfileEdit',

],

},

},

},

})

## 导航故障

如果我们想在一个导航完成后做一些事情，我们需要一个在调用 router.push 后进行等待的方法。想象一下，我们有一个移动手机菜单，它允许我们进入不同的页面，而我们只想在导航到新页面后隐藏菜单，我们可能想这样做：

router.push('/my-profile')

this.isMenuOpen = false

但是这样做会马上关闭菜单，因为 导航是异步的，我们需要 await router.push 返回的 promise ：

await router.push('/my-profile')

this.isMenuOpen = false

现在，一旦导航完成，菜单就会关闭，但如果导航被阻止，它也会关闭。我们需要一种方法来检测我们是否真的改变了页面。

### 检测导航故障

如果导航被阻止，导致用户停留在同一个页面上，由 router.push 返回的 Promise 的解析值将是 Navigation Failure。否则，它将是一个 falsy 值(通常是 undefined)。这样我们就可以区分我们导航是否离开了当前位置：

const navigationResult = await router.push('/my-profile')

if (navigationResult) {

// 导航被阻止

} else {

// 导航成功 (包括重新导航的情况)

this.isMenuOpen = false

}

Navigation Failure 是带有一些额外属性的 Error 实例，这些属性为我们提供了足够的信息，让我们知道哪些导航被阻止了以及为什么被阻止了。要检查导航结果的性质，请使用 isNavigationFailure 函数：

import { NavigationFailureType, isNavigationFailure } from 'vue-router'

// 试图离开未保存的编辑文本界面

const failure = await router.push('/articles/2')

if (isNavigationFailure(failure, NavigationFailureType.aborted)) {

// 给用户显示一个小通知

showToast('You have unsaved changes, discard and leave anyway?')

}

如果你忽略第二个参数：isNavigationFailure(failure)，那么就只会检查这个 failure 是不是一个 Navigation Failure。

### 鉴别导航故障

正如我们在一开始所说的，有不同的情况会导致导航的中止，所有这些情况都会导致不同的 Navigation Failure。它们可以用 isNavigationFailure 和 NavigationFailureType 来区分。总共有三种不同的类型：

* aborted：在导航守卫中返回 false 中断了本次导航。
* cancelled： 在当前导航还没有完成之前又有了一个新的导航。比如，在等待导航守卫的过程中又调用了 router.push。
* duplicated：导航被阻止，因为我们已经在目标位置了。

### 导航故障的属性

所有的导航失败都会暴露 to 和 from 属性，以反映失败导航的当前位置和目标位置：

// 正在尝试访问 admin 页面

router.push('/admin').then(failure => {

if (isNavigationFailure(failure, NavigationFailureType.redirected)) {

failure.to.path // '/admin'

failure.from.path // '/'

}

})

在所有情况下，to 和 from 都是规范化的路由地址。

### 检测重定向

当在导航守卫中返回一个新的位置时，我们会触发一个新的导航，覆盖正在进行的导航。与其他返回值不同的是，重定向不会阻止导航，而是创建一个新的导航。因此，通过读取路由地址中的 redirectedFrom 属性，对其进行不同的检查：

await router.push('/my-profile')

if (router.currentRoute.value.redirectedFrom) {

// redirectedFrom 是解析出的路由地址，就像导航守卫中的 to和 from

}

## 动态路由

对路由的添加通常是通过 routes 选项来完成的，但是在某些情况下，你可能想在应用程序已经运行的时候添加或删除路由。具有可扩展接口(如 Vue CLI UI )这样的应用程序可以使用它来扩展应用程序。

### 添加路由

动态路由主要通过两个函数实现。router.addRoute() 和 router.removeRoute()。它们只注册一个新的路由，也就是说，如果新增加的路由与当前位置相匹配，就需要你用 router.push() 或 router.replace() 来手动导航，才能显示该新路由。看一个例子：

const router = createRouter({

history: createWebHistory(),

routes: [{ path: '/:articleName', component: Article }],

})

进入任何页面，/about，/store，或者 /3-tricks-to-improve-your-routing-code 最终都会呈现 Article 组件。如果我们在 /about 上添加一个新的路由：

router.addRoute({ path: '/about', component: About })

页面仍然会显示 Article 组件，我们需要手动调用 router.replace() 来改变当前的位置，并覆盖我们原来的位置（而不是添加一个新的路由，最后在我们的历史中两次出现在同一个位置）：

router.addRoute({ path: '/about', component: About })

// 我们也可以使用 this.$route 或 route = useRoute() （在 setup 中）

router.replace(router.currentRoute.value.fullPath)

记住，如果你需要等待新的路由显示，可以使用 await router.replace()。

### 在导航守卫中添加路由

如果你决定在导航守卫内部添加或删除路由，你不应该调用 router.replace()，而是通过返回新的位置来触发重定向：

router.beforeEach(to => {

if (!hasNecessaryRoute(to)) {

router.addRoute(generateRoute(to))

// 触发重定向

return to.fullPath

}

})

上面的例子有两个假设：第一，新添加的路由记录将与 to 位置相匹配，实际上导致与我们试图访问的位置不同。第二，hasNecessaryRoute() 在添加新的路由后返回 false，以避免无限重定向。

因为是在重定向中，所以我们是在替换将要跳转的导航，实际上行为就像之前的例子一样。而在实际场景中，添加路由的行为更有可能发生在导航守卫之外，例如，当一个视图组件挂载时，它会注册新的路由。

### 删除路由

有几个不同的方法来删除现有的路由：

* 通过添加一个名称冲突的路由。如果添加与现有途径名称相同的途径，会先删除路由，再添加路由：

router.addRoute({ path: '/about', name: 'about', component: About })

// 这将会删除之前已经添加的路由，因为他们具有相同的名字且名字必须是唯一的

router.addRoute({ path: '/other', name: 'about', component: Other })

* 通过调用 router.addRoute() 返回的回调：

const removeRoute = router.addRoute(routeRecord)

removeRoute() // 删除路由如果存在的话

当路由没有名称时，这很有用。

* 通过使用 router.removeRoute() 按名称删除路由：

router.addRoute({ path: '/about', name: 'about', component: About })

// 删除路由

router.removeRoute('about')

需要注意的是，如果你想使用这个功能，但又想避免名字的冲突，可以在路由中使用 Symbol 作为名字。

当路由被删除时，所有的别名和子路由也会被同时删除

### 添加嵌套路由

要将嵌套路由添加到现有的路由中，可以将路由的 name 作为第一个参数传递给 router.addRoute()，这将有效地添加路由，就像通过 children 添加的一样：

router.addRoute({ name: 'admin', path: '/admin', component: Admin })

router.addRoute('admin', { path: 'settings', component: AdminSettings })

这等效于：

router.addRoute({

name: 'admin',

path: '/admin',

component: Admin,

children: [{ path: 'settings', component: AdminSettings }],

})

### 查看现有路由

Vue Router 提供了两个功能来查看现有的路由：

* router.hasRoute()：检查路由是否存在。
* router.getRoutes()：获取一个包含所有路由记录的数组。

# 升级规模

## 单文件组件

Vue 的单文件组件 (即 \*.vue 文件，简称 SFC) 是一种特殊的文件格式，使我们能够将一个 Vue 组件的模板、逻辑与样式封装在单个文件中。下面是一个单文件组件的示例：

<script>

export default {

data() {

return {

greeting: 'Hello World!'

}

}

}

</script>

<template>

<p class="greeting">{{ greeting }}</p>

</template>

<style>

.greeting {

color: red;

font-weight: bold;

}

</style>

如你所见，Vue 的单文件组件是 HTML、CSS 和 JavaScript 三种元素的自然延伸。<template>、<script> 和 <style> 三个块在同一个文件中封装、组合了组件的视图、逻辑和样式。

### 使用 SFC 优势

管 SFC 需要一套构建步骤，但也相应地有非常多优势：

* 使用熟悉的 HTML、CSS 和 JavaScript 语法编写模块化的组件
* 罗列本身就强耦合的逻辑关注点
* 预编译模板
* 组件级隔离的 CSS
* 在使用组合式 API 时语法更简单
* 通过交叉分析模板和逻辑代码能进行更多编译时优化
* 更好的 IDE 支持，提供自动补全和对模板中表达式的类型检查
* 开箱即用的模块热更新 (HMR) 支持

SFC 是 Vue 框架提供的一个功能，并且在下列场景中都是官方推荐的项目组织方式：

* 单页面应用 (SPA)
* 静态站点生成 (SSG)
* 大型的前端项目，若使用构建步骤可以获得更好的开发体验 (DX)

### SFC工作流程

Vue SFC 是一个框架指定的文件格式，因此必须交由 @vue/compiler-sfc 编译为标准的 JavaScript 和 CSS，一个编译后的 SFC 是一个标准的 JavaScript(ES) 模块，这也意味着通过适当的构建配置，你可以像导入其他 ES 模块一样导入 SFC：

import MyComponent from './MyComponent.vue'

export default {

components: {

MyComponent

}

}

SFC 中的 <style> 标签一般会在开发时注入成原生的 <style> 标签以支持热更新，而生产环境下它们会被抽取、合并成单独的 CSS 文件。

## 工具链

### 在线尝试

不需要在机器上安装任何东西，也可以尝试单文件组件的使用。提供了一个在线的 Playground，可以在浏览器中访问：

* Vue SFC Playground
  + 始终根据最新的提交部署
  + 用来检查编译输出的结果
* StackBlitz 中的 Vue + Vite
  + 类似 IDE 的环境，但实际是在浏览器中运行 Vite 开发服务器
  + 和本地启动效果最接近

### 项目脚手架

#### Vite

Vite 是一个轻量级的、速度极快的构建工具，对 Vue SFC 提供第一优先级支持。要使用 Vite 来启动一个 Vue 项目，非常简单：

$ npm init vue@latest

这个命令同时还会安装和执行 create-vue，它是 Vue 提供的官方脚手架工具。

* 要学习更多关于 Vite 的知识，请查看 [Vite 官方文档](https://cn.vitejs.dev/)。
* 若要了解如何为一个 Vite 项目配置 Vue 相关的特殊行为，比如向 Vue 编译器传递相关选项，请查看 [@vitejs/plugin-vue 的文档](https://github.com/vitejs/vite/tree/main/packages/plugin-vue#readme)。

上面提到的两种在线 Playground 也支持下载文件为一个 Vite 项目。

#### Vue CLI

Vue CLI 是官方提供的基于 Webpack 的 Vue 工具链。它现在处于维护模式，我们建议使用 Vite 开始新的项目，除非你依赖特定的 Webpack 的特性。在大多数情况下，Vite 将提供更优秀的开发体验。

#### IDE 支持

* 推荐使用的 IDE 是 VSCode+ 配备 Volar 插件。Volar 提供了语法高亮、TypeScript 支持，和模板内表达式与组件 props 的智能提示。
* Volar 替代了 Vetur，那是我们之前为 Vue 2 提供的官方 VSCode 扩展。如果你之前已经安装了 Vetur，请确保在 Vue 3 的项目中禁用它。
  + WebStorm 同样也为 Vue 的单文件组件提供了很好的内置支持。
* 其他支持语言服务协议 (LSP) 的 IDE 也可以通过 LSP 享受到 Volar 所提供的的核心功能。例如 coc-volar，它是为 Vim/NeoVim 提供更多 Vue SFC 支持。

#### 浏览器开发者插件

Vue 的浏览器开发者插件使我们可以浏览一个 Vue 应用的组件树，查看各个组件的状态，追踪状态管理的事件，还有测评性能表现。

* [文档](https://devtools.vuejs.org/)
* [Chrome 插件商店页](https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd)
* [Firefox 所属插件页](https://addons.mozilla.org/en-US/firefox/addon/vue-js-devtools/)
* [独立的 Electron 应用所属插件](https://devtools.vuejs.org/guide/installation.html#standalone)

## 状态管理

从技术上讲，每一个 Vue 组件实例都已经在“管理”它自己的响应式状态了。我们以一个简单的计数器组件为例：

<script setup>

import { ref } from 'vue'

// 状态

const count = ref(0)

// 动作

function increment() {

count.value++

}

</script>

<!-- 视图 -->

<template>{{ count }}</template>

它是一个独立的单元，由以下几个部分组成：

* 状态：驱动整个应用的数据源；
* 视图：对状态的一种声明式映射；
* 动作：状态根据用户在视图中的输入而作出相应变更的可能方式。

下面是“单向数据流”这一概念的简单图示：

[单向数据流](单向数据流.png)

然而，当我们有多个组件共享一个共同的状态时，就没有这么简单了：

* 多个视图可能都依赖于同一部分状态。
* 不同视图中的动作可能需要更改同一部分状态。

### 响应性 API

如果你有一部分状态需要在多个组件实例间共享，你可以使用 reactive() 来创建一个响应式对象，并在不同组件中导入它：

// store.js

import { reactive } from 'vue'

export const store = reactive({

count: 0

})

<!-- ComponentA.vue -->

<script setup>

import { store } from './store.js'

</script>

<template>From A: {{ store.count }}</template>

<!-- ComponentB.vue -->

<script setup>

import { store } from './store.js'

</script>

<template>From B: {{ store.count }}</template>

现在每当 store 对象被更改时，<ComponentA> 与 <ComponentB> 都会自动更新它们的视图。现在我们有了单一的数据源。然而，这也意味着任意一个导入了 store 的组件都可以随意修改它的状态：

<template>

<button @click="store.count++">

From B: {{ store.count }}

</button>

</template>

请注意这里点击的处理函数使用了 store.increment()，带上了圆括号作为内联表达式调用，因为它并不是组件的方法，并且必须要以正确的 this 上下文来调用。

除了我们这里用到的单个响应式对象作为一个 store 之外，你还可以使用其他响应性 API 例如 ref() 或是 computed()，或是甚至通过一个组合式函数来返回一个全局状态：

import { ref } from 'vue'

// 全局状态，创建在模块作用域下

const globalCount = ref(1)

export function useCount() {

// 局部状态，每个组件都会创建

const localCount = ref(1)

return {

globalCount,

localCount

}

}

### Pinia

虽然我们的手动状态管理解决方案在简单的场景中已经足够了，但是在大规模的生产应用中还有很多其他事项需要考虑：

* 更强的团队协作约定
* 与 Vue DevTools 集成，包括时间轴、组件内部审查和时间旅行调试
* 模块热更新 (HMR)
* 服务端渲染支持

Pinia 就是一个实现了上述需求的状态管理库，由 Vue 核心团队维护，对 Vue 2 和 Vue 3 都可用。

现有用户可能对 Vuex 更熟悉，它是 Vue 之前的官方状态管理库。由于 Pinia 在生态系统中能够承担相同的职责且能做得更好，因此 Vuex 现在处于维护模式。它仍然可以工作，但不再接受新的功能。对于新的应用程序，建议使用 Pinia。

实上，Pinia 这款产品最初是为了探索 Vuex 的下一个版本，整合了核心团队关于 Vuex 5 的许多想法。最终，我们意识到 Pinia 已经实现了我们想要在 Vuex 5 中提供的大部分内容，因此决定将其作为新的官方推荐。

相比于 Vuex，Pinia 提供了更简洁直接的 API，并提供了组合式风格的 API，最重要的是，在使用 TypeScript 时它提供了非常好的类型推导。

## 服务端渲染 (SSR)

https://staging-cn.vuejs.org/guide/scaling-up/ssr.html#higher-level-solutions

### 总览

Vue.js 是一个用于构建客户端应用的框架。默认情况下，Vue 组件在浏览器中生成和操作 DOM 作为输出。然而，我们也可以将相同的组件在服务端渲染成 HTML 字符串，直接返回给浏览器，最后再将静态的 HTML“激活” (hydrate) 为完全交互式的客户端应用。

一个由服务端渲染的 Vue.js 应用也可以被认为是“同构的”或“通用的”，因为应用的大部分代码同时运行在服务端和客户端。

与客户端的单页应用 (SPA) 相比，SSR 的优势主要在于：

* 更快的内容到达时间：这一点在慢网速或者运行缓慢的设备上尤为重要。服务端渲染的 HTML 无需等到所有的 JavaScript 都下载并执行完成之后才显示，所以你的用户将会更快地看到完整渲染的页面。除此之外，数据获取过程在首次访问时在服务端完成，相比于从客户端获取，可能有更快的数据库连接。这通常可以带来更高的核心 Web 指标评分、更好的用户体验，而对于那些“内容到达时间与转化率直接相关”的应用来说，这点可能至关重要。
* 统一的心智模型：你可以使用相同的语言以及相同的声明式、面向组件的心智模型来开发整个应用，而不需要在后端模板系统和前端框架之间来回切换。
* 更好的 SEO：搜索引擎爬虫可以直接看到完全渲染的页面。

使用 SSR 时还有一些权衡之处需要考量：

* 开发中的限制。浏览器端特定的代码只能在某些生命周期钩子中使用；一些外部库可能需要特殊处理才能在服务端渲染的应用中运行。
* 更多的与构建配置和部署相关的要求。服务端渲染的应用需要一个能让 Node.js 服务器运行的环境，不像完全静态的 SPA 那样可以部署在任意的静态文件服务器上
* 更高的服务端负载。在 Node.js 中渲染一个完整的应用要比仅仅托管静态文件更加占用 CPU 资源，因此如果你预期有高流量，请为相应的服务器负载做好准备，并采用合理的缓存策略。

在为你的应用使用 SSR 之前，你首先应该问自己是否真的需要它。这主要取决于内容到达时间对应用的重要程度。例如，如果你正在构建一个内部的仪表盘，初始加载时的那额外几百毫秒对你来说并不重要，这种情况下使用 SSR 就有点小题大作了。然而，在内容到达时间极其重要的场景下，SSR 可以尽可能地帮你实现最优的初始加载性能。

静态站点生成 (SSG)，也被称为预渲染，是另一种流行的构建快速网站的技术。如果用服务端渲染一个页面所需的数据对每个用户来说都是相同的，那么我们可以只渲染一次，提前在构建过程中完成，而不是每次请求进来就重新渲染页面。预渲染的页面生成后作为静态 HTML 文件被服务器托管。

SSG 保留了和 SSR 应用相同的性能表现：它带来了优秀的内容到达耗时性能。同时，它比 SSR 应用的花销更小，也更容易部署，因为它输出的是静态 HTML 和资源文件。这里的关键词是静态：SSG 仅可以用于消费静态数据的页面，即数据在构建期间就是已知的，并且在多次部署期间不会改变。每当数据变化时，都需要重新部署。

### 基础教程

#### 渲染一个应用

让我们来看一个 Vue SSR 最基础的实战示例。

1. 创建一个新的文件夹，cd 进入
2. 执行 npm init -y
3. 在 package.json 中添加 "type"："module" 使 Node.js 以 ES modules mode 运行
4. 执行 npm install vue
5. 创建一个 example.js 文件：

// 此文件运行在 Node.js 服务器上

import { createSSRApp } from 'vue'

// Vue 的服务端渲染 API 位于 `vue/server-renderer` 路径下

import { renderToString } from 'vue/server-renderer'

const app = createSSRApp({

data: () => ({ count: 1 }),

template: `<button @click="count++">{{ count }}</button>`

})

renderToString(app).then((html) => {

console.log(html)

})

接着运行：

> node example.js

它应该会在命令行中打印出如下内容：

<button>1</button>

renderToString() 接收一个 Vue 应用实例作为参数，返回一个 Promise，当 Promise resolve 时得到应用渲染的 HTML。当然你也可以使用 Node.js Stream API 或者 Web Streams API 来执行流式渲染。查看 SSR API 参考获取完整的相关细节。

然后我们可以把 Vue SSR 的代码移动到一个服务器请求处理函数里，它将应用的 HTML 片段包装为完整的页面 HTML。接下来的几步我们将会使用 express：

* 执行 npm install express
* 创建下面的 server.js 文件：

import express from 'express'

import { createSSRApp } from 'vue'

import { renderToString } from 'vue/server-renderer'

const server = express()

server.get('/', (req, res) => {

const app = createSSRApp({

data: () => ({ count: 1 }),

template: `<button @click="count++">{{ count }}</button>`

})

renderToString(app).then((html) => {

res.send(`

<!DOCTYPE html>

<html>

<head>

<title>Vue SSR Example</title>

</head>

<body>

<div id="app">${html}</div>

</body>

</html>

`)

})

})

server.listen(3000, () => {

console.log('ready')

})

最后，执行 node server.js，访问 http://localhost:3000。你应该可以看到页面中的按钮了。

#### 客户端激活

如果你点击该按钮，你会发现数字并没有改变。这段 HTML 在客户端是完全静态的，因为我们没有在浏览器中加载 Vue。

为了使客户端的应用可交互，Vue 需要执行一个激活步骤。在激活过程中，Vue 会创建一个与服务端完全相同的应用实例，然后将每个组件与它应该控制的 DOM 节点相匹配，并添加 DOM 事件监听器。

为了在激活模式下挂载应用，我们应该使用 createSSRApp() 而不是 createApp()：

// 该文件运行在浏览器中

import { createSSRApp } from 'vue'

const app = createSSRApp({

// ...和服务端完全一致的应用实例

})

// 在客户端挂载一个 SSR 应用时会假定

// HTML 是预渲染的，然后执行激活过程，

// 而不是挂载新的 DOM 节点

app.mount('#app')

#### 代码结构

这里我们将演示最基础的设置。首先，让我们将应用的创建逻辑拆分到一个单独的文件 app.js 中：

// app.js (在服务器和客户端之间共享)

import { createSSRApp } from 'vue'

export function createApp() {

return createSSRApp({

data: () => ({ count: 1 }),

template: `<button @click="count++">{{ count }}</button>`

})

}

该文件及其依赖项在服务器和客户端之间共享——我们称它们为通用代码。编写通用代码时需要注意许多事项，我们将在下面讨论。我们在客户端入口导入通用代码，创建应用程序并执行挂载：

// client.js

import { createApp } from './app.js'

createApp().mount('#app')

服务器在请求处理函数中使用相同的应用创建逻辑：

// server.js (不相关的代码省略)

import { createApp } from './app.js'

server.get('/', (req, res) => {

const app = createApp()

renderToString(app).then(html => {

// ...

})

})

此外，为了在浏览器中加载客户端文件，我们还需要：

* 在 server.js 中添加 server.use(express.static('.')) 来托管客户端文件。
* 将 <script type="module" src="/client.js"></script> 添加到 HTML 外壳以加载客户端入口文件。
* 通过在 HTML 外壳中添加 Import Map 以支持在浏览器中使用 import \* from 'vue'。

### 更通用的解决方案

# TypeScript

## TypeScript 使用 Vue

像 TypeScript 这样的类型系统可以在编译时通过静态分析检测出很多常见错误。这减少了生产环境中的运行时错误，也让我们在重构大型项目的时候更有信心。通过 IDE 中基于类型的自动补全，TypeScript 还改善了开发体验和效率。

### 项目启动

create-vue，即官方的项目脚手架工具，提供了搭建基于 Vite 且 TypeScript 就绪的 Vue 项目的选项。在基于 Vite 的配置中，开发服务器和打包器将只会对 TypeScript 文件执行语法转译，而不会执行任何类型检查，这保证了 Vite 开发服务器在使用 TypeScript 时也能始终保持飞快的速度。

* 在开发阶段，我们推荐你依赖一个好的 IDE 配置来获取即时的类型错误反馈。
* 对于单文件组件，你可以使用工具 vue-tsc 在命令行检查类型和生成类型声明文件。vue-tsc 是对 TypeScript 自身命令行界面 tsc 的一个封装。它的工作方式基本和 tsc 一致。除了 TypeScript 文件，它还支持 Vue 的单文件组件。
* vue-tsc 目前还不支持 watch 模式，但这已经在计划之中。与此同时，如果你想要类型检查成为 dev 命令的一部分，可以看看 vite-plugin-checker。
* Vue CLI 也提供了对 TypeScript 的支持，但是已经不推荐了。

#### IDE 支持

* 强烈推荐 Visual Studio Code (VSCode)，因为它对 TypeScript 有着很好的内置支持。
  + Volar 是官方的 VSCode 扩展，提供了 Vue 单文件组件中的 TypeScript 支持，还伴随着一些其他非常棒的特性。
  + TypeScript Vue Plugin 用于支持在 TS 中 import \*.vue 文件。
* WebStorm 对 TypeScript 和 Vue 也都提供了开箱即用的支持。其他的 JetBrains IDE 也同样可以通过一个免费插件支持。

#### 配置 tsconfig.json

通过 create-vue 搭建的项目包含了预置的 tsconfig.json。其底层配置抽象于 @vue/tsconfig 包中。在项目内我们使用 Project References 来确保运行在不同环境下的代码的类型正确 (比如应用代码 vs. 测试代码)。在手动配置 tsconfig.json 时，请留意以下选项：

* compilerOptions.isolatedModules 被设置为了 true，因为 Vite 使用 esbuild 来转译 TypeScript，并受限于单文件转译的限制。
* 如果你正在使用选项式 API，需要将 compilerOptions.strict 设置为 true (或者至少开启 compilerOptions.noImplicitThis，它是 strict 模式的一部分)，才可以获得对组件选项中 this 的类型检查。否则 this 会被认为是 any。
* 如果你在构建工具中配置了路径解析别名，例如 @/\* 这个别名被默认配置在了 create-vue 项目中，你需要通过 compilerOptions.paths 选项为 TypeScript 再配置一遍。

#### 托管模式

为了让 Vue 单文件组件和 TypeScript 一起工作，Volar 创建了一个针对 Vue 的 TS 语言服务实例，将其用于 Vue 单文件组件。同时，普通的 TS 文件依然由 VSCode 内置的 TS 语言服务来处理。这也是为什么我们需要安装 TypeScript Vue Plugin 来支持在 TS 文件中引入 Vue 单文件组件。这套默认设置能够工作，但在每个项目里我们都运行了两个语言服务实例：一个来自 Volar，一个来自 VSCode 的内置服务。这在大型项目里可能会带来一些性能问题。为了优化性能，Volar 提供了一个叫做“托管模式”的功能。在托管模式下，Volar 使用单个 TS 语言服务实例同时为 Vue 和 TS 文件提供支持。

要开启托管模式，你需要执行以下步骤来在你的项目的工作空间中禁用 VSCode 的内置 TS 语言服务：

1. 在当前项目的工作空间下，用 Ctrl + Shift + P (macOS：Cmd + Shift + P) 唤起命令面板。
2. 输入 built，然后选择“Extensions：Show Built-in Extensions”。
3. 在插件搜索框内输入 typescript (不要删除 @builtin 前缀)。
4. 点击“TypeScript and JavaScript Language Features”右下角的小齿轮，然后选择“Disable (Workspace)”。
5. 重新加载工作空间。托管模式将会在你打开一个 Vue 或者 TS 文件时自动启用。

#### Vue CLI 和 ts-loader

在像 Vue CLI 这样的基于 webpack 搭建的项目中，一般是在模块转换的管道中执行类型检查，例如使用 ts-loader。然而这并不是一个简洁的解决方案，因为类型系统需要了解整个模块关系才能执行类型检查。单个模块的转换步骤并不适合该任务。这导致了下面的问题：

* ts-loader 只能对转换后的代码执行类型检查，这和我们在 IDE 或 vue-tsc 中看到的可以映射回源代码的错误并不一致。
* 类型检查可能会很慢。当它和代码转换在相同的线程/进程中执行时，它会显著影响整个应用程序的构建速度。
* 我们已经在 IDE 中通过单独的进程运行着类型检查了，因此这一步降低了开发体验但没有带来足够的收益。

### 使用说明

#### defineComponent()

为了让TypeScript 正确地推导出组件选项内的类型，我们需要通过 defineComponent() 这个全局 API 来定义组件：

import { defineComponent } from 'vue'

export default defineComponent({

// 启用了类型推导

props: {

name: String,

msg: { type: String, required: true }

},

data() {

return {

count: 1

}

},

mounted() {

this.name // 类型：string | undefined

this.msg // 类型：string

this.count // 类型：number

}

})

当没有结合 <script setup> 使用组合式 API 时，defineComponent() 也支持对传递给 setup() 的 prop 的推导：

import { defineComponent } from 'vue'

export default defineComponent({

// 启用了类型推导

props: {

message: String

},

setup(props) {

props.message // 类型：string | undefined

}

})

#### 在单文件组件中的用法

要在单文件组件中使用 TypeScript，需要在 <script> 标签上加上 lang="ts" 的 attribute。当 lang="ts" 存在时，所有的模板内表达式都将享受到更严格的类型检查。

<script lang="ts">

import { defineComponent } from 'vue'

export default defineComponent({

data() {

return {

count: 1

}

}

})

</script>

<template>

<!-- 启用了类型检查和自动补全 -->

{{ count.toFixed(2) }}

</template>

lang="ts" 也可以用于 <script setup>：

<script setup lang="ts">

// 启用了 TypeScript

import { ref } from 'vue'

const count = ref(1)

</script>

<template>

<!-- 启用了类型检查和自动补全 -->

{{ count.toFixed(2) }}

</template>

#### 模板中的 TypeScript

在使用了 <script lang="ts"> 或 <script setup lang="ts"> 后，<template> 在绑定表达式中也支持 TypeScript。这对需要在模板表达式中执行类型转换的情况下非常有用。这里有一个假想的例子：

<script setup lang="ts">

let x: string | number = 1

</script>

<template>

<!-- 出错，因为 x 可能是字符串 -->

{{ x.toFixed(2) }}

</template>

可以使用内联类型强制转换解决此问题：

<script setup lang="ts">

let x: string | number = 1

</script>

<template>

{{ (x as number).toFixed(2) }}

</template>

## TypeScript 与组合式 API

### 为组件的 prop 标注类型

当使用 <script setup> 时，这个 defineProps() 宏函数支持从它的参数中推导类型：

<script setup lang="ts">

const props = defineProps({

foo: { type: String, required: true },

bar: Number

})

props.foo // string

props.bar // number | undefined

</script>

这被称之为“运行时声明”，因为传递给 defineProps() 的参数会作为运行时的 props 选项使用。然而，通过泛型参数来定义 prop 的类型通常更直接：

<script setup lang="ts">

const props = defineProps<{

foo: string

bar?: number

}>()

</script>

这被称之为“基于类型的声明”。编译器会尽可能地尝试根据类型参数推导出等价的运行时选项。在这种场景下，我们第二个例子中编译出的运行时选项和第一个是完全一致的。基于类型的声明或者运行时声明都可以使用，但是你不能同时使用两者。我们也可以将 prop 的类型移入一个单独的接口中：

<script setup lang="ts">

interface Props {

foo: string

bar?: number

}

const props = defineProps<Props>()

</script>

为了生成正确的运行时代码，传给 defineProps() 的泛型参数必须是以下之一：

* 一个类型字面量：

defineProps<{ /\*... \*/ }>()

* 对同一个文件中的一个接口或对象类型字面量的引用：

interface Props {/\* ... \*/}

defineProps<Props>()

接口或对象字面类型可以包含从其他文件导入的类型引用，但是，传递给 defineProps 的泛型参数本身不能是一个导入的类型：

import { Props } from './other-file'

// 不支持！

defineProps<Props>()

当使用基于类型的声明时，我们失去了对 prop 定义默认值的能力。这可以通过目前实验性的响应性语法糖来解决（这个行为目前需要显式地选择开启）：

<script setup lang="ts">

interface Props {

foo: string

bar?: number

}

// 对 defineProps() 的响应性解构

// 默认值会被编译为等价的运行时选项

const { foo, bar = 100 } = defineProps<Props>()

</script>

### 为组件的 emit 标注类型

在 <script setup> 中，emit 函数的类型标注可以通过运行时声明或类型声明进行：

<script setup lang="ts">

// 运行时

const emit = defineEmits(['change', 'update'])

// 基于类型

const emit = defineEmits<{

(e: 'change', id: number): void

(e: 'update', value: string): void

}>()

</script>

这个类型参数应该是一个带调用签名的类型字面量。这个类型字面量的类型就是返回的 emit 函数的类型。我们可以看到，基于类型的声明使我们可以对所触发事件的类型进行更细粒度的控制。

若没有使用 <script setup>，defineComponent() 也可以根据 emits 选项推导暴露在 setup 上下文中的 emit 函数的类型：

import { defineComponent } from 'vue'

export default defineComponent({

emits: ['change'],

setup(props, { emit }) {

emit('change') // <-- 类型检查 / 自动补全

}

})

### 为 ref() 标注类型

ref 会根据初始化时的值推导其类型：

import { ref } from 'vue'

// 推导出的类型：Ref<number>

const year = ref(2020)

// => TS Error: Type 'string' is not assignable to type 'number'.

year.value = '2020'

有时我们可能想为 ref 内的值指定一个更复杂的类型，可以通过使用 Ref 这个类型：

import { ref } from 'vue'

import type { Ref } from 'vue'

const year: Ref<string | number> = ref('2020')

year.value = 2020 // 成功！

如果你指定了一个泛型参数但没有给出初始值，那么最后得到的就将是一个包含 undefined 的联合类型：

// 推导得到的类型：Ref<number | undefined>

const n = ref<number>()

### 为 reactive() 标注类型

reactive() 也会隐式地从它的参数中推导类型：

import { reactive } from 'vue'

// 推导得到的类型：{ title: string }

const book = reactive({ title: 'Vue 3 指引' })

要显式地标注一个 reactive property 的类型，我们可以使用接口：

import { reactive } from 'vue'

interface Book {

title: string

year?: number

}

const book: Book = reactive({ title: 'Vue 3 指引' })

### 为 computed() 标注类型

computed() 会从其计算函数的返回值上推导出类型：

import { ref, computed } from 'vue'

const count = ref(0)

// 推导得到的类型：ComputedRef<number>

const double = computed(() => count.value \* 2)

// => TS Error: Property 'split' does not exist on type 'number'

const result = double.value.split('')

你还可以通过泛型参数显式指定类型：

const double = computed<number>(() => {

// 若返回值不是 number 类型则会报错

})

### 为事件处理器标注类型

在处理原生 DOM 事件时，应该为我们传递给事件处理器的参数正确地标注类型。让我们看一下这个例子：

<script setup lang="ts">

function handleChange(event) {

// `event` 隐式地标注为 `any` 类型

console.log(event.target.value)

}

</script>

<template>

<input type="text" @change="handleChange" />

</template>

没有类型标注时，这个 event 参数会隐式地标注为 any 类型。这也会在 tsconfig.json 中配置了 "strict": true 或 "noImplicitAny": true 时报出一个 TS 错误。因此，建议显式地为事件处理器的参数标注类型。此外，你可能需要显式地强制转换 event 上的 property：

function handleChange(event: Event) {

console.log((event.target as HTMLInputElement).value)

}

### 为 provide/inject 标注类型

provide 和 inject 通常会在不同的组件中运行。要正确地为注入的值标记类型，Vue 提供了一个 InjectionKey 接口，它是一个继承自 Symbol 的泛型类型，可以用来在提供者和消费者之间同步注入值的类型：

import { provide, inject } from 'vue'

import type { InjectionKey } from 'vue'

const key = Symbol() as InjectionKey<string>

provide(key, 'foo') // 若提供的是非字符串值会导致错误

const foo = inject(key) // foo 的类型：string | undefined

建议将注入 key 的类型放在一个单独的文件中，这样它就可以被多个组件导入。当使用字符串注入 key 时，注入值的类型是 unknown，需要通过泛型参数显式声明：

const foo = inject<string>('foo') // 类型：string | undefined

注意注入的值仍然可以是 undefined，因为无法保证提供者一定会在运行时 provide 这个值。当提供了一个默认值后，这个 undefined 类型就可以被移除：

const foo = inject<string>('foo', 'bar') // 类型：string

如果你确定该值将始终被提供，则还可以强制转换该值：

const foo = inject('foo') as string

### 为模板 ref 标注类型

模板 ref 需要通过一个显式指定的泛型参数和一个初始值 null 来创建：

<script setup lang="ts">

import { ref, onMounted } from 'vue'

const el = ref<HTMLInputElement | null>(null)

onMounted(() => {

el.value?.focus()

})

</script>

<template>

<input ref="el" />

</template>

注意为了严格的类型安全，有必要在访问 el.value 时使用可选链或类型守卫。这是因为直到组件被挂载前，这个 ref 的值都是初始的 null，并且在由于 v-if 的行为将引用的元素卸载时也可以被设置为 null。

### 为组件模板 ref 标注类型

有时，你可能需要为一个子组件添加一个模板 ref，以便调用它公开的方法。举个例子，我们有一个 MyModal 子组件，它有一个打开模态框的方法：

<!-- MyModal.vue -->

<script setup lang="ts">

import { ref } from 'vue'

const isContentShown = ref(false)

const open = () => (isContentShown.value = true)

defineExpose({

open

})

</script>

为了获取 MyModal 的类型，我们首先需要通过 typeof 得到其类型，再使用 TypeScript 内置的 InstanceType 工具类型来获取其实例类型：

<!-- App.vue -->

<script setup lang="ts">

import MyModal from './MyModal.vue'

const modal = ref<InstanceType<typeof MyModal> | null>(null)

const openModal = () => {

modal.value?.open()

}

</script>

注意，如果你想在 TypeScript 文件而不是在 Vue SFC 中使用这种技巧，需要开启 Volar 的托管模式。

# 进阶主题

## 使用 Vue 的多种方式

在 Web 的世界中从来就没有可以适配所有场景、解决所有问题的银弹。正因如此，Vue 被设计成一个灵活的、可以渐进式集成的框架。根据使用场景的不同需要，相应地有多种不同的方式来使用 Vue，以此在技术栈复杂度、开发体验和性能表现间取得最佳平衡。

### 独立脚本

Vue 可以作为独立的脚本文件使用，无需构建步骤！如果你有一个后端框架，并且它已经渲染了大部分的 HTML，或者你的前端逻辑并不复杂，不需要构建步骤，这会是你集成 Vue 最容易的方式。你可以在这些场景中将 Vue 想象成 jQuery 的一个更加声明式的替代。

Vue 也提供了另一个适用于此类无构建步骤场景的替代版 petite-vue，主要为渐进式增强已有的 HTML 作了特别的优化。功能更加精简，十分轻量。

### 嵌入Web Component

你可以用 Vue 来构建标准的 Web Component，这些 Web Component 可以嵌入到任何 HTML 页面中，无论它们是如何被渲染的。这个选项允许你以一种完全与消费者无关的方式使用 Vue：因为生成的 Web Component 可以嵌入到旧版本的应用、静态 HTML，甚至用其他框架构建的应用中。

### 单页面应用 (SPA)

一些应用在前端需要具有丰富的交互性和并不简单的有状态逻辑。构建这类应用的最佳方法是使用这样一种架构：Vue 不仅控制整个页面，还可以处理数据更新和导航，而无需重新加载页面。这种类型的应用通常称为单页面应用 (Single-Page application，SPA)。Vue 提供了核心功能库和综合性的工具链支持，为现代 SPA 提供了极佳的开发体验，包括以下内容：

* 客户端路由
* 极其快速的构建工具
* IDE 支持
* 浏览器开发工具
* TypeScript 集成
* 测试工具

SPA 一般都依赖后端来暴露 API 端点，但你也可以将 Vue 和如 Inertia.js 之类的解决方案搭配使用，在保留以服务器为中心的开发模型的同时获得 SPA 的益处。

### 全栈 / SSR

纯客户端的 SPA 在首屏加载和 SEO 方面有显著的问题。这是因为浏览器会收到一个巨大的 HTML 空页面，必须等到 JavaScript 加载完毕再渲染。

Vue 为服务端将一个 Vue 应用渲染成 HTML 字符串提供了第一优先级支持的 API。这使得服务器能够分发渲染好的 HTML，允许用户在 JavaScript 下载时看到页面内容。Vue 之后会在客户端对应用进行“激活 (hydrate)”使其重获可交互性。这被称为服务端渲染 (SSR)，它极大地改善了 Web 应用在核心指标上的性能表现，如最大内容绘制 (LCP)。

也有一些针对此类场景、基于 Vue 构建的框架，比如 NuxtJS，允许你使用 Vue 和 JavaScript 开发一个全栈应用。

### JAMStack / SSG

如果所需的数据是静态的，那么服务端渲染可以提前完成。这意味着我们可以将整个应用预渲染为 HTML，并将其作为静态文件提供。这增强了站点的性能表现，也使部署变得更容易，因为我们无需根据请求动态地渲染页面。Vue 仍可通过激活在客户端提供交互。这一技术通常被称为静态站点生成 (SSG)，也被称为 JAMStack。

Vue 团队维护了一个静态站点生成器 VitePress，你正在阅读的文档也是基于它构建的！另外，NuxtJS 也支持 SSG。你甚至可以在同一个 Nuxt 应用中通过不同的路由提供 SSR 和 SSG。

### Web 之外...

尽管 Vue 主要是为构建 Web 应用而设计的，但它绝不仅仅局限于浏览器。你还可以：

* 配合 Electron 或 Tauri 构建桌面应用
* 配合 Ionic Vue 构建移动端应用
* 使用 Quasar 用同一套代码同时开发桌面端和移动端应用
* 使用 Vue 的自定义渲染 API 来构建不同目标的渲染器，比如 WebGL 甚至是终端命令行！

## 组合式 API FAQ

### 什么是组合式 API

组合式 API 是一系列 API 的集合，使我们可以使用函数而不是声明选项的方式书写 Vue 组件。它是一个概括性的术语，涵盖了以下方面的 API：

* 响应性 API：例如 ref() 和 reactive()，使我们可以直接创建响应式状态、计算属性和侦听器。
* 生命周期钩子：例如 onMounted() 和 onUnmounted()，使我们可以在组件各个生命周期阶段添加逻辑。
* 依赖注入：例如 provide() 和 inject()，使我们可以在使用响应性 API 时，利用 Vue 的依赖注入系统。

组合式 API 是 Vue 3 的内置功能，而要想在 Vue 2 中使用，可以使用官方维护的插件 @vue/composition-api。在 Vue 3 中，组合式 API 基本上都会配合 <script setup> 语法在单文件组件中使用。下面是一个使用组合式 API 的组件示例：

<script setup>

import { ref, onMounted } from 'vue'

// 响应式状态

const count = ref(0)

// 更改状态、触发更新的函数

function increment() {

count.value++

}

// 生命周期钩子

onMounted(() => {

console.log(`计数器初始值为 ${count.value}。`)

})

</script>

<template>

<button @click="increment">点击了：{{ count }} 次</button>

</template>

虽然这套 API 的风格是基于函数的组合，但组合式 API 并不是函数式编程。组合式 API 是以 Vue 中数据可变的、细粒度的响应性系统为基础的，而函数式编程更强调数据不可变。

### 为什么要有组合式 API

#### 更好的逻辑复用

组合式 API 最基本的优势是它使我们能够通过组合函数来实现更加简洁高效的逻辑复用。它解决了所有 mixins 的缺陷，那是选项式 API 中一种逻辑复用机制。

组合式 API 提供的更多逻辑复用可能性孵化了一些非常棒的社区项目，比如 VueUse，一个不断成长的工具型组合式函数集合。组合式 API 还为其他第三方状态管理库集成 Vue 的响应式系统提供了一套简洁清晰的机制，例如 RxJS。

#### 更灵活的代码组织

许多用户都喜欢选项式 API，因为在默认情况下就能够写出有组织的代码：任何东西都有其对应的选项来管理。然而，选项式 API 在单个组件的逻辑复杂到一定程度时，也面临了一些无法忽视的限制。这些限制主要体现在需要处理多个逻辑关注点的组件中，在许多 Vue 2 已经上线的生产应用中可以看到这一点。我们以 Vue CLI GUI 中的文件浏览器组件为例：这个组件承担了以下几个逻辑关注点：

* 追踪当前文件夹的状态，展示其内容
* 处理文件夹的相关操作 (打开、关闭和刷新)
* 支持创建新文件夹
* 可以切换到只展示收藏的文件夹
* 可以开启对隐藏文件夹的展示
* 处理当前工作目录中的变更

处理相同逻辑关注点的代码被强制拆分在了不同的选项中，位于文件的不同部分。在一个几百行的大组件中，要读懂代码中的一个逻辑关注点，只能在文件中反复上下滚动，可这件事不应该这么困难。另外，如果我们还想要将一个逻辑关注点抽离到一个可重用的工具函数中，需要从文件的不同部分中寻找到所需的正确片段。

而如果用组合式 API 重构这个组件，将会变成：现在与同一个逻辑关注点相关的代码被归为了一组：我们无需再为了一个逻辑关注点在不同的选项块间来回滚动切换。此外，我们现在可以不费吹灰之力地将这一组代码移动到一个外部文件中，不再需要为了抽象而重新组织代码，大大降低了重构成本，这在长期维护的大型项目中非常关键。

#### 更好的类型推导

组合式 API 主要利用基本的变量和函数，它们本身就是类型友好的。用组合式 API 重写的代码可以享受到完整的类型推导，不需要书写太多类型标注。大多数时候，用 TypeScript 书写的组合式 API 代码和用 JavaScript 写都差不太多！这也同样让许多纯 JavaScript 用户能从 IDE 中享受到部分类型推导功能。

#### 生产包体积更小

搭配 <script setup> 使用组合式 API 比等价情况下的选项式 API 更高效，对代码压缩也更友好。这是由于 <script setup> 形式书写的组件模板被编译为了一个内联函数，和 <script setup> 中的代码位于同一作用域。不像选项式 API 需要依赖 this 上下文对象访问属性，被编译的模板可以直接访问 <script setup> 中定义的变量，无需一个代码实例从中代理。这对代码压缩更友好，因为变量的名字可以变得更短，但对象的属性名则不能。

### 和 React Hooks 相比

组合式 API 提供了和 React Hooks 相同级别的逻辑组织能力，但它们之间有着一些重要的区别。React Hooks 在组件每次更新时都会重新调用。这就产生了一些即使是经验丰富的 React 开发人员也会感到困惑的问题。这也带来了一些性能问题，严重影响开发的体验。下面一些例子：

* 昂贵的计算需要使用 useMemo，这也需要传入正确的依赖数组。
* 在默认情况下，传递给子组件的事件处理函数会导致子组件进行不必要的更新。子组件默认更新，并需要显式的调用 useCallback 作优化。这几乎是必需的，因此同样需要正确的依赖数组。忽视这一点会导致默认情况下对应用程序进行过度渲染，并可能在不知不觉中导致性能问题。
* 要解决变量闭包导致的问题，再结合并发功能，使得很难推理出一段钩子代码是什么时候运行的，并且很不好处理需要横跨多个渲染 (通过 useRef) 的可变状态。

相比起来，Vue 的组合式 API：

* 仅调用 setup() 或 <script setup> 的代码一次。这使得代码能更好地与 JavaScript 的习惯性使用的直觉结合起来，因为不需要担心闭包变量的问题。组合式 API 也并不限制调用顺序，还可以有条件地进行调用。
* Vue 的响应性系统运行时会自动收集计算属性和侦听器的依赖，因此无需我们手动声明依赖。
* 无需手动缓存回调函数来避免不必要的组件更新。总而言之，Vue 细粒度的响应性系统确保了组件仅执行必要的更新。对 Vue 开发者来说几乎不怎么需要对子组件更新进行手动优化。

## 深入响应式系统

Vue 最有区别性的功能就是其潜藏于底层的响应式系统。组件状态都是响应式的 JavaScript 对象。当更改它们时，视图会随即更新。

### 响应性

如果我们在 JavaScript 写类似的逻辑：

let A0 = 1

let A1 = 2

let A2 = A0 + A1

console.log(A2) // 3

A0 = 2

console.log(A2) // 仍然是 3

当我们更改 A0 后，A2 不会自动更新。那么我们如何在 JavaScript 中做到这一点呢？首先，为了能重新运行计算的代码来更新 A2，我们需要将其包装为一个函数：

let A2

function update() {

A2 = A0 + A1

}

然后，我们需要定义几个术语：

* 这个 update() 函数会产生一个副作用，或者就简称为作用，因为它会更改程序里的状态。
* A0 和 A1 被视为这个作用的依赖，因为它们的值被用来执行这个作用。因此这次作用也可以说是一个它依赖的订阅者。

我们需要一个魔法函数，能够在 A0 或 A1 (这两个依赖) 变化时调用 update() (产生作用)。

whenDepsChange(update)

这个 whenDepsChange() 函数有如下的任务：

1. 当一个变量被读取时进行追踪。例如我们执行了表达式 A0 + A1 的计算，则 A0 和 A1 都被读取到了。
2. 如果一个变量在当前运行的副作用中被读取了，就将该副作用设为此变量的一个订阅者。例如由于 A0 和 A1 在 update() 执行时被访问到了，则 update() 需要在第一次调用之后成为 A0 和 A1 的订阅者。
3. 探测一个变量的变化。例如当我们给 A0 赋了一个新的值后，应该通知其所有订阅了的副作用重新执行。

### 响应性是如何工作的

我们无法直接追踪对上述示例中局部变量的读写过程，在原生 JavaScript 中没有提供这样一种机制。但是，我们是可以追踪一个对象的属性进行读和写的。

在JavaScript中有两种劫持属性访问的方式：getter/setters和Proxies。Vue2使用 getter/setters 完全由于需支持更旧版本浏览器的限制。而在 Vue 3 中使用了 Proxy 来创建响应式对象，将 getter/setter 用于 ref。下面的伪代码将会说明它们是如何工作的：

function reactive(obj) {

return new Proxy(obj, {

get(target, key) {

track(target, key)

return target[key]

},

set(target, key, value) {

trigger(target, key)

target[key] = value

}

})

}

function ref(value) {

const refObject = {

get value() {

track(refObject, 'value')

return value

},

set value(newValue) {

trigger(refObject, 'value')

value = newValue

}

}

return refObject

}

在 track() 内部，我们会检查当前是否有正在运行的副作用。如果有，我们会查找到一个所有追踪了该属性的订阅者，它们存储在一个 Set 中，然后将当前这个副作用添加到该 Set 中。

// 这会在一个副作用就要运行之前被设置

// 我们会在后面处理它

let activeEffect

function track(target, key) {

if (activeEffect) {

const effects = getSubscribersForProperty(target, key)

effects.add(activeEffect)

}

}

副作用订阅将被存储在一个全局的 WeakMap<target, Map<key, Set<effect>>> 数据结构中。如果在第一次追踪时没有找到对相应属性订阅的副作用集合，它将会在这里新建。这就是 getSubscribersForProperty() 函数所做的事。在 trigger() 之中，我们会再查找到该属性的所有订阅副作用。但这一次我们是去调用它们：

function trigger(target, key) {

const effects = getSubscribersForProperty(target, key)

effects.forEach((effect) => effect())

}

回到 whenDepsChange() 函数中：

function whenDepsChange(update) {

const effect = () => {

activeEffect = effect

update()

activeEffect = null

}

effect()

}

它包装了原先的 update 函数到一个副作用中，并在运行实际的更新之前，将它自己设为当前活跃的副作用。而在更新期间开启的 track() 调用，都将能定位到这个当前活跃的副作用。此时，我们已经创建了一个能自动跟踪其依赖关系的副作用，它会在依赖关系更改时重新运行。我们称其为**响应式副作用**。

Vue 提供了一个 API 来让你创建响应式副作用 watchEffect()。事实上，你会发现它的使用方式和我们上面示例中说的魔法函数 whenDepsChange() 非常相似。我们可以用真正的 Vue API 改写上面的例子：

import { ref, watchEffect } from 'vue'

const A0 = ref(0)

const A1 = ref(1)

const A2 = ref()

watchEffect(() => {

// 追踪 A0 和 A1

A2.value = A0.value + A1.value

})

// 将触发副作用

A0.value = 2

使用一个响应式副作用来更改一个 ref 并不是最优解，事实上使用计算属性会更直观简洁：

import { ref, computed } from 'vue'

const A0 = ref(0)

const A1 = ref(1)

const A2 = computed(() => A0.value + A1.value)

A0.value = 2

在内部，computed 会使用响应式副作用来管理失效与重新计算的过程。那么，常见的响应式副作用的用例是什么呢？自然是更新 DOM！我们可以像下面这样实现一个简单的“响应式渲染”：

import { ref, watchEffect } from 'vue'

const count = ref(0)

watchEffect(() => {

document.body.innerHTML = `计数：${count.value}`

})

// 更新 DOM

count.value++

实际上，这与 Vue 组件保持状态和 DOM 同步的方式非常接近。每个组件实例创建一个响应式副作用来渲染和更新 DOM。当然，Vue 组件使用了比 innerHTML 更高效的方式来更新 DOM。

### 运行时 vs. 编译时响应性

Vue 的响应式系统基本是基于运行时的。追踪和触发都是在浏览器中运行时进行的。运行时响应性的优点是，它可以在没有构建步骤的情况下工作，而且边缘情况较少。另一方面，这使得它受到了 JavaScript 语法的制约。通过响应性语法糖这一实验性功能去尝试减少冗余代码：

let A0 = $ref(0)

let A1 = $ref(1)

// 在变量读取时追踪

const A2 = $computed(() => A0 + A1)

// 在变量写入时触发

A0 = 2

这个代码段会被编译成没有该转换时的样子，即自动地为所有变量引用处添加上 .value。

### 响应性调试

Vue 的响应性系统可以自动跟踪依赖关系，但在某些情况下，我们可能希望确切地知道正在跟踪什么，或者是什么导致了组件重新呈现。

#### 组件调试钩子

我们可以在一个组件渲染时调试查看哪些依赖正在被使用，以及使用 onRenderTracked 和 onRenderTriggered 生命周期钩子来确定哪个依赖正在触发更新。这些钩子都会收到一个调试事件，其中包含了所需依赖的信息。推荐在回调中放置一个 debugger 语句，使你可以在开发者工具中交互式地查看依赖（组件调试钩子仅会在开发模式下工作）：

<script setup>

import { onRenderTracked, onRenderTriggered } from 'vue'

onRenderTracked((event) => {

debugger

})

onRenderTriggered((event) => {

debugger

})

</script>

调试事件对象有如下的类型定义：

type DebuggerEvent = {

effect: ReactiveEffect

target: object

type:

| TrackOpTypes /\* 'get' | 'has' | 'iterate' \*/

| TriggerOpTypes /\* 'set' | 'add' | 'delete' | 'clear' \*/

key: any

newValue?: any

oldValue?: any

oldTarget?: Map<any, any> | Set<any>

}

#### 计算属性调试

我们可以向 computed() 传入第二个参数，是一个包含了 onTrack 和 onTrigger 两个回调函数的对象：

* onTrack 将在响应属性或引用作为依赖项被跟踪时被调用。
* onTrigger 将在侦听器回调被依赖项的变更触发时被调用。
* 计算属性的 onTrack 和 onTrigger 选项仅会在开发模式下工作。

const plusOne = computed(() => count.value + 1, {

onTrack(e) {

// 当 count.value 被追踪为依赖时触发

debugger

},

onTrigger(e) {

// 当 count.value 被更改时触发

debugger

}

})

// 访问 plusOne，会触发 onTrack

console.log(plusOne.value)

// 更改 count.value，应该会触发 onTrigger

count.value++

#### 侦听器调试

和 computed() 类似，侦听器也支持 onTrack 和 onTrigger 选项（侦听器的 onTrack 和 onTrigger 选项仅会在开发模式下工作。）：

watch(source, callback, {

onTrack(e) {

debugger

},

onTrigger(e) {

debugger

}

})

watchEffect(callback, {

onTrack(e) {

debugger

},

onTrigger(e) {

debugger

}

})

## 渲染机制

Vue 是如何将一份模板转换为真实的 DOM 节点的呢？又是如何高效地更新 DOM 节点的呢？

### 虚拟 DOM

你可能已经听说过虚拟 DOM 的概念了，Vue 的渲染系统正是基于这个概念构建的。虚拟 DOM(VDOM) 是一种编程概念，意为将将目标所需的 UI 通过数据结构“虚拟”地表示出来，保存在内存中，并与真实的 DOM 保持同步。这个概念是由 React 率先开拓，随后在各个不同的框架中都有不同的实现，当然也包括 Vue。与其说虚拟 DOM 是一种具体的技术，不如说是一种模式。所以没有一个标准的实现。我们可以用一个简单的例子来说明：const vnode = {

type: 'div',

props: {

id: 'hello'

},

children: [

/\* 更多 vnode \*/

]

}

这里所说的 vnode 即一个纯 JavaScript 的对象 (一个“虚拟节点”)，它代表着一个 <div> 元素。它包含我们创建实际元素所需的所有信息。它还包含更多的子节点，这使它成为虚拟 DOM 树的根节点。

一个运行时渲染器将会遍历整个虚拟 DOM 树，并据此构建真实的 DOM 树。这个过程被称为挂载 (mount)。如果我们有两份虚拟 DOM 树，渲染器将会有比较地遍历它们，找出它们之间的区别，并应用这其中的变化到真实的 DOM 上。这个过程被称为修补 (patch)，又被称为“比较差异 (diffing)”或“协调 (reconciliation)”。

虚拟 DOM 带来的主要收益是它赋予了开发者编程式地、声明式地创建、审查和组合所需 UI 结构的能力，而把直接与 DOM 相关的操作交给了渲染器。

### 渲染管线

以更高层面的视角看，Vue 组件挂载后发生了如下这几件事：

1. 编译：Vue 模板被编译为了渲染函数：即用来返回虚拟 DOM 树的函数。这一步骤可以通过构建步骤提前完成，也可以通过使用运行时编译器即时完成。
2. 挂载：运行时渲染器调用渲染函数，遍历返回的虚拟 DOM 树，并基于它创建实际的 DOM 节点。这一步会作为响应式副作用执行，因此它会追踪其中所用到的所有响应式依赖。
3. 修补：当一个依赖发生变化后，副作用会重新运行，这时候会创建一个更新后的虚拟 DOM 树。运行时渲染器遍历这棵新树，将它与旧树进行比较，然后将必要的更新应用到真实 DOM 上去。

[vue组件挂载后流程](Vue%20组件挂载过程.png)

### 模板 vs. 渲染函数

Vue 模板会被预编译成虚拟 DOM 渲染函数。Vue 也提供了 API 使我们可以不使用模板编译，直接手写渲染函数。在处理高度动态的逻辑时，渲染函数相比于模板更加灵活，因为你可以完全地使用 JavaScript 来构造你想要的 vnode。

* 模板更贴近实际的 HTML。这使得我们能够更方便地重用一些已有的 HTML 代码片段，能够带来更好的可访问性体验、能更方便地使用 CSS 应用样式，并且更容易使设计师理解和修改。
* 由于其确定的语法，更容易对模板做静态分析。这使得 Vue 的模板编译器能够应用许多编译时优化来提升虚拟 DOM 的性能表现

### 带编译时信息的虚拟 DOM

虚拟 DOM 在 React 和大多数其他实现中都是纯运行时的：协调算法无法预知新的虚拟 DOM 树会是怎样，因此它总是需要遍历整棵树、比较每个 vnode 上 props 的区别来确保正确性。另外，即使一棵树的某个部分从未改变，还是会在每次重渲染时创建新的 vnode，带来了完全不必要的内存压力。

虚拟 DOM 在 React 和大多数其他实现中都是纯运行时的：协调算法无法预知新的虚拟 DOM 树会是怎样，因此它总是需要遍历整棵树、比较每个 vnode 上 props 的区别来确保正确性。另外，即使一棵树的某个部分从未改变，还是会在每次重渲染时创建新的 vnode，带来了完全不必要的内存压力。

#### 静态提升

在模板中常常有部分内容是不带任何动态绑定的：

<div>

<div>foo</div> <!-- 需提升 -->

<div>bar</div> <!-- 需提升 -->

<div>{{ dynamic }}</div>

</div>

foo 和 bar 这两个 div 是完全静态的，没有必要在重新渲染时再次创建和比对它们。Vue 编译器自动地会提升这部分 vnode 创建函数到这个模板的渲染函数之外，并在每次渲染时都使用这份相同的 vnode，渲染器知道新旧 vnode 在这部分是完全相同的，所以会完全跳过对它们的差异比对。

此外，当有足够多连续的静态元素时，它们还会再被压缩为一个“静态 vnode”，其中包含的是这些节点相应的纯 HTML 字符串。这些静态节点会直接通过 innerHTML 来挂载。同时还会在初次挂载后缓存相应的 DOM 节点。如果这部分内容在应用中其他地方被重用，那么将会使用原生的 cloneNode() 方法来克隆新的 DOM 节点，这会非常高效。

#### 修补标记 Flags

对于单个有动态绑定的元素来说，我们可以在编译时推断出大量信息：

<!-- 仅含 class 绑定 -->

<div :class="{ active }"></div>

<!-- 仅含 id 和 value 绑定 -->

<input :id="id" :value="value">

<!-- 仅含文本子节点 -->

<div>{{ dynamic }}</div>

在为这些元素生成渲染函数时，Vue 在 vnode 创建调用中直接编码了每个元素所需的更新类型：

createElementVNode("div", {

class: \_normalizeClass({ active: \_ctx.active })

}, null, 2 /\* CLASS \*/)

最后这个参数 2 就是一个修补标记 (patch flag)。一个元素可以有多个修补标记，会被合并成一个数字。运行时渲染器也将会使用位运算来检查这些标记，确定相应的更新操作：

if (vnode.patchFlag & PatchFlags.CLASS /\* 2 \*/) {

// 更新节点的 CSS class

}

位运算检查是非常快的。通过这样的修补标记，Vue 能够在更新带有动态绑定的元素时做最少的操作。

Vue 也为 vnode 的子节点标记了类型。举个例子，包含多个根节点的模板被表示为一个片段 (fragment)，大多数情况下，我们可以确定其顺序是永远不变的，所以这部分信息就可以提供给运行时作为一个修补标记。

export function render() {

return (\_openBlock(), \_createElementBlock(\_Fragment, null, [

/\* children \*/

], 64 /\* STABLE\_FRAGMENT \*/))

}

运行时会完全跳过对这个根片段中子元素顺序的重新协调过程。

#### 树结构打平

上面这个例子中生成的代码，所返回的虚拟 DOM 树是经一个特殊的 createElementBlock() 调用创建的：

export function render() {

return (\_openBlock(), \_createElementBlock(\_Fragment, null, [

/\* children \*/

], 64 /\* STABLE\_FRAGMENT \*/))

}

这里我们引入一个概念“区块”，内部结构是稳定的一个部分可被称之为一个区块。在这个用例中，整个模板只有一个区块，因为这里没有用到任何结构性指令 (比如 v-if 或者 v-for)。每一个块都会追踪其所有带修补标记的后代节点 (不只是直接子节点)，举个例子：

<div> <!-- root block -->

<div>...</div> <!-- 不会追踪 -->

<div :id="id"></div> <!-- 要追踪 -->

<div> <!-- 不会追踪 -->

<div>{{ bar }}</div> <!-- 要追踪 -->

</div>

</div>

编译的结果会被打平为一个数组，仅包含所有动态的后代节点：

div (block root)

- div 带有 :id 绑定

- div 带有 {{ bar }} 绑定

当这个组件需要重渲染时，只需要遍历这个打平的树而非整棵树。这也就是我们所说的树结构打平，这大大减少了我们在虚拟 DOM 协调时需要遍历的节点数量。模板中任何的静态部分都会被高效地略过。

v-if 和 v-for 指令会创建新的区块节点：

<div> <!-- 根区块 -->

<div>

<div v-if> <!-- if 区块 -->

...

<div>

</div>

</div>

一个子区块会在父区块的动态子节点数组中被追踪，这为他们的父区块保留了一个稳定的结构

#### 对 SSR 激活的影响

修补标记和树结构打平都大大提升了 Vue SSR 激活的性能表现：

* 单个元素的激活可以基于相应 vnode 的修补标记走更快的捷径。
* 在激活时只有区块节点和其动态子节点需要被遍历，这在模板层面上实现更高效的部分激活。

## 渲染函数 & JSX

在绝大多数情况下，Vue 推荐使用模板语法来搭建 HTML。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。

### 基本用法

#### 创建 Vnodes

Vue 提供了一个 h() 函数用于创建 vnodes：

import { h } from 'vue'

const vnode = h(

'div', // type

{ id: 'foo', class: 'bar' }, // props

[

/\* children \*/

]

)

h() 是 hyperscript 的简称——意思是“能生成 HTML (超文本标记语言) 的 JavaScript”。这个名字来源于许多虚拟 DOM 实现时共享的约定。h() 函数的使用方式非常的灵活：

// 除了类型必填以外，其他的参数都是可选的

h('div')

h('div', { id: 'foo' })

// attribute 和 property 都能在 prop 中书写

// Vue 会自动将它们分配到正确的位置

h('div', { class: 'bar', innerHTML: 'hello' })

// props modifiers such as .prop and .attr can be added

// with '.' and `^' prefixes respectively

h('div', { '.name': 'some-name', '^width': '100' })

// 类与样式可以像在模板中一样

// 用数组或对象的形式书写

h('div', { class: [foo, { bar }], style: { color: 'red' } })

// 事件监听器应以 onXxx 的形式书写

h('div', { onClick: () => {} })

// children 可以是一个字符串

h('div', { id: 'foo' }, 'hello')

// 没有 props 时可以省略不写

h('div', 'hello')

h('div', [h('span', 'hello')])

// children 数组可以同时包含 vnodes 与字符串

h('div', ['hello', h('span', 'hello')])

得到的 vnode 为如下形式：

const vnode = h('div', { id: 'foo' }, [])

vnode.type // 'div'

vnode.props // { id: 'foo' }

vnode.children // []

vnode.key // null

注意事项：完整的 VNode 接口包含其他内部 property，但是强烈建议避免使用这些没有在这里列举出的 property。这样能够避免因内部 property 变更而导致的不兼容性问题。

#### 声明渲染函数

当组合式 API 与模板一起使用时，setup() 钩子的返回值是用于暴露数据给模板。然而当我们使用渲染函数时，可以直接把渲染函数返回：

import { ref, h } from 'vue'

export default {

props: {

/\* ... \*/

},

setup(props) {

const count = ref(1)

// 返回渲染函数

return () => h('div', props.msg + count.value)

}

}

在 setup() 内部声明的渲染函数天生能够访问在同一范围内声明的 props 和许多响应式状态。除了返回一个 vnode，你还可以返回字符串或数组：

export default {

setup() {

return () => 'hello world!'

}

}

import { h } from 'vue'

export default {

setup() {

// 使用数组返回多个根节点

return () => [

h('div'),

h('div'),

h('div')

]

}

}

请确保返回的是一个函数而不是一个值！setup() 函数在每个组件中只会被调用一次，而返回的渲染函数将会被调用多次。

如果一个渲染函数组件不需要任何实例状态，为了简洁起见，它们也可以直接被声明为一个函数：

function Hello() {

return 'hello world!'

}

没错，这就是一个有效的 Vue 组件

#### Vnodes 必须唯一

组件树中的 vnodes 必须是唯一的。下面是错误示范：

function render() {

const p = h('p', 'hi')

return h('div', [

// 啊哦，重复的 vnodes 是无效的

p,

p

])

}

如果你真的非常想在页面上渲染多个重复的元素或者组件，你可以使用一个工厂函数来做这件事。比如下面的这个渲染函数就可以完美渲染出 20 个相同的段落：

function render() {

return h(

'div',

Array.from({ length: 20 }).map(() => {

return h('p', 'hi')

})

)

}

### JSX / TSX

JSX 是 JavaScript 的一个类似 XML 的扩展，有了它，我们可以用以下的方式来书写代码：

const vnode = <div>hello</div>

在 JSX 表达式中，使用大括号来嵌入动态值：

const vnode = <div id={dynamicId}>hello, {userName}</div>

虽然最早是由 React 引入，但实际上 JSX 语法并没有定义运行时语义，并且能被编译成成各种不同的输出形式。如果你之前使用过 JSX 语法，那么请注意 Vue 的 JSX 编译方式与 React 中 JSX 的编译方式不同，因此你不能在 Vue 应用中使用 React 的 JSX 编译。与 React JSX 语法的一些明显区别包括：

* 可以使用 HTML attributes 比如 class 和 for 作为 props - 不需要使用 className 或 htmlFor。
* 传递子元素给组件 (比如 slots) 的方式不同。

Vue 的类型定义也提供了 TSX 语法的类型推断支持。当使用 TSX 语法时，确保在 tsconfig.json 中配置了 "jsx": "preserve"，这样的 TypeScript 就能保证 Vue JSX 语法编译过程中的完整性。

### 函数式组件

函数式组件是自身没有任何状态的组件的另一种形式。它们在渲染过程中不会创建组件实例，并跳过常规的组件生命周期。我们使用的是一个简单函数，而不是一个选项对象，来创建函数式组件。该函数实际上就是该组件的 render 函数。函数式组件的签名与 setup() 钩子相同：

function MyComponent(props, { slots, emit, attrs }) {

// ...

}

大多数常规组件的配置选项在函数式组件中都不可用。然而我们还是可以把 props 和 emits 作为 property 加入，以达到定义它们的目的：

MyComponent.props = ['value']

MyComponent.emits = ['click']

如果这个 props 选项没有被定义，那么被传入函数的 props 对象就会像 attrs 一样会包含所有 attribute。除非指定了 props 选项，否则每个 prop 的名字将不会基于驼峰命名法被一般化处理。函数式组件可以像普通组件一样被注册和消费。如果你将一个函数作为第一个参数传入 h，它将会被当作一个函数式组件来对待。