# 第1章 全局API 应用程序API

创建应用程序（）#

创建一个应用程序实例。

类型

function createApp(rootComponent: Component, rootProps?: object): App

细节

第一个参数是根组件。第二个可选参数是要传递给根组件的道具。

例子

使用内联根组件：

import { createApp } from 'vue'

const app = createApp({

/\* root component options \*/

})

使用进口组件：

import { createApp } from 'vue'

import App from './App.vue'

const app = createApp(App)

另请参阅： 指南 - 创建 Vue 应用程序

创建SSRApp()#

在SSR Hydration模式下创建应用程序实例。用法与createApp().

app.mount()#

在容器元素中安装应用程序实例。

类型

interface App {

mount(rootContainer: Element | string): ComponentPublicInstance

}

细节

参数可以是实际的 DOM 元素或 CSS 选择器（将使用第一个匹配的元素）。返回根组件实例。

如果组件定义了模板或渲染函数，它将替换容器内任何现有的 DOM 节点。否则，如果运行时编译器可用，innerHTML则容器的 将用作模板。

在 SSR 水化模式下，它将水化容器内现有的 DOM 节点。如果有mismatches，现有的 DOM 节点将被变形以匹配预期的输出。

对于每个应用实例，mount()只能调用一次。

例子

import { createApp } from 'vue'

const app = createApp(/\* ... \*/)

app.mount('#app')

也可以挂载到实际的 DOM 元素：

app.mount(document.body.firstChild)

app.unmount()#

卸载已安装的应用程序实例，触发应用程序组件树中所有组件的卸载生命周期挂钩。

类型

interface App {

unmount(): void

}

app.provide()#

提供一个可以注入应用程序内所有后代组件的值。

类型

interface App {

provide<T>(key: InjectionKey<T> | symbol | string, value: T): this

}

细节

期望注入键作为第一个参数，提供的值作为第二个参数。返回应用程序实例本身。

例子

import { createApp } from 'vue'

const app = createApp(/\* ... \*/)

app.provide('message', 'hello')

在应用程序的组件内部：

export default {

inject: ['message'],

created() {

console.log(this.message) // 'hello'

}

}

也可以看看：

提供/注入

应用级提供

app.component()#

如果同时传递名称字符串和组件定义，则注册一个全局组件，如果只传递名称，则检索已注册的组件。

类型

interface App {

component(name: string): Component | undefined

component(name: string, component: Component): this

}

例子

import { createApp } from 'vue'

const app = createApp({})

// register an options object

app.component('my-component', {

/\* ... \*/

})

// retrieve a registered component

const MyComponent = app.component('my-component')

另请参阅： 组件注册

app.directive()#

如果同时传递名称字符串和指令定义，则注册全局自定义指令，如果仅传递名称，则检索已注册的指令。

类型

interface App {

directive(name: string): Directive | undefined

directive(name: string, directive: Directive): this

}

例子

import { createApp } from 'vue'

const app = createApp({

/\* ... \*/

})

// register (object directive)

app.directive('my-directive', {

/\* custom directive hooks \*/

})

// register (function directive shorthand)

app.directive('my-directive', () => {

/\* ... \*/

})

// retrieve a registered directive

const myDirective = app.directive('my-directive')

另请参阅： 自定义指令

应用程序.use()#

安装插件。

类型

interface App {

use(plugin: Plugin, ...options: any[]): this

}

细节

期望插件作为第一个参数，可选插件选项作为第二个参数。

插件可以是带有install()方法的对象，也可以只是用作install()方法的函数。选项（的第二个参数app.use()）将传递给插件的install()方法。

当app.use()在同一个插件上多次调用时，插件只会安装一次。

例子

import { createApp } from 'vue'

import MyPlugin from './plugins/MyPlugin'

const app = createApp({

/\* ... \*/

})

app.use(MyPlugin)

另请参阅： 插件

app.mixin()#

应用一个全局 mixin（作用于应用程序）。全局 mixin 将其包含的选项应用于应用程序中的每个组件实例。

不建议

Vue 3 支持 Mixin 主要是为了向后兼容，因为它们在生态系统库中广泛使用。应避免在应用程序代码中使用混入，尤其是全局混入。

对于逻辑重用，更喜欢Composables。

类型

interface App {

mixin(mixin: ComponentOptions): this

}

应用程序版本#

提供创建应用程序时使用的 Vue 版本。这在plugins中很有用，您可能需要基于不同 Vue 版本的条件逻辑。

类型

interface App {

version: string

}

例子

在插件中执行版本检查：

export default {

install(app) {

const version = Number(app.version.split('.')[0])

if (version < 3) {

console.warn('This plugin requires Vue 3')

}

}

}

另请参阅： 全局 API - 版本

应用程序配置#

每个应用程序实例都会公开一个config包含该应用程序配置设置的对象。您可以在安装应用程序之前修改其属性（如下所述）。

import { createApp } from 'vue'

const app = createApp(/\* ... \*/)

console.log(app.config)

app.config.errorHandler#

为从应用程序内部传播的未捕获错误分配一个全局处理程序。

类型

interface AppConfig {

errorHandler?: (

err: unknown,

instance: ComponentPublicInstance | null,

// `info` is a Vue-specific error info,

// e.g. which lifecycle hook the error was thrown in

info: string

) => void

}

细节

错误处理程序接收三个参数：错误、触发错误的组件实例和指定错误源类型的信息字符串。

它可以从以下来源捕获错误：

组件渲染

事件处理程序

生命周期钩子

setup()功能

观察者

自定义指令钩子

过渡挂钩

例子

app.config.errorHandler = (err, instance, info) => {

// handle error, e.g. report to a service

}

app.config.warnHandler#

为来自 Vue 的运行时警告分配一个自定义处理程序。

类型

interface AppConfig {

warnHandler?: (

msg: string,

instance: ComponentPublicInstance | null,

trace: string

) => void

}

细节

警告处理程序接收警告消息作为第一个参数，源组件实例作为第二个参数，组件跟踪字符串作为第三个参数。

它可用于过滤掉特定的警告以减少控制台的冗长。所有 Vue 警告都应在开发期间解决，因此仅建议在调试会话期间关注许多特定警告，并且应在调试完成后将其删除。

小费

警告仅在开发期间有效，因此在生产模式下会忽略此配置。

例子

app.config.warnHandler = (msg, instance, trace) => {

// `trace` is the component hierarchy trace

}

app.config.performance#

将此设置为true在浏览器开发工具性能/时间线面板中启用组件初始化、编译、渲染和修补性能跟踪。仅适用于开发模式和支持performance.mark API 的浏览器。

类型：boolean

另请参阅： 指南 - 性能

app.config.compilerOptions#

配置运行时编译器选项。在此对象上设置的值将传递给浏览器内模板编译器，并影响已配置应用程序中的每个组件。请注意，您还可以使用选项在每个组件的基础上覆盖这些compilerOptions选项。

重要的

此配置选项仅在使用完整版本（即vue.js可以在浏览器中编译模板的独立版本）时受到尊重。如果您在构建设置中使用仅运行时构建，则必须改为@vue/compiler-dom通过构建工具配置传递编译器选项。

For vue-loader：通过compilerOptions加载器选项传递。另请参阅如何在vue-cli.

对于vite：通过@vitejs/plugin-vue选项。

app.config.compilerOptions.isCustomElement#

指定一个检查方法来识别本机自定义元素。

类型： (tag: string) => boolean

细节

true如果标签应被视为本机自定义元素，则应返回。对于匹配的标签，Vue 会将其呈现为原生元素，而不是尝试将其解析为 Vue 组件。

本机 HTML 和 SVG 标签不需要在此函数中匹配 - Vue 的解析器会自动识别它们。

例子

// treat all tags starting with 'ion-' as custom elements

app.config.compilerOptions.isCustomElement = (tag) => {

return tag.startsWith('ion-')

}

另请参阅： Vue 和 Web 组件

app.config.compilerOptions.whitespace#

调整模板空白处理行为。

类型： 'condense' | 'preserve'

默认： 'condense'

细节

Vue 删除/压缩模板中的空白字符以产生更有效的编译输出。默认策略是“浓缩”，具有以下行为：

元素内的前导/结束空白字符被压缩成一个空格。

包含换行符的元素之间的空白字符被删除。

文本节点中连续的空白字符被压缩成一个空格。

将此选项设置为'preserve'将禁用 (2) 和 (3)。

例子

app.config.compilerOptions.whitespace = 'preserve'

app.config.compilerOptions.delimiters#

调整用于模板内文本插值的分隔符。

类型： [string, string]

默认： ['{{', '}}']

细节

这通常用于避免与也使用 mustache 语法的服务器端框架发生冲突。

例子

// Delimiters changed to ES6 template string style

app.config.compilerOptions.delimiters = ['${', '}']

app.config.compilerOptions.comments#

调整模板中 HTML 注释的处理方式。

类型： boolean

默认： false

细节

默认情况下，Vue 会删除生产环境中的注释。将此选项设置为true将强制 Vue 即使在生产中也保留评论。在开发过程中始终保留注释。当 Vue 与其他依赖 HTML 注释的库一起使用时，通常会使用此选项。

例子

app.config.compilerOptions.comments = true

app.config.globalProperties#

可用于注册全局属性的对象，可在应用程序内的任何组件实例上访问这些属性。

类型

interface AppConfig {

globalProperties: Record<string, any>

}

细节

Vue.prototype这是 Vue 3中不再存在的 Vue 2 的替代品。与任何全局的东西一样，应该谨慎使用它。

如果全局属性与组件自己的属性冲突，组件自己的属性将具有更高的优先级。

用法

app.config.globalProperties.msg = 'hello'

这使得msg应用程序中的任何组件模板以及this任何组件实例中都可用：

export default {

mounted() {

console.log(this.msg) // 'hello'

}

}

app.config.optionMergeStrategies#

用于定义自定义组件选项的合并策略的对象。

类型

interface AppConfig {

optionMergeStrategies: Record<string, OptionMergeFunction>

}

type OptionMergeFunction = (to: unknown, from: unknown) => any

细节

一些插件/库添加了对自定义组件选项的支持（通过注入全局混合）。当同一个选项需要从多个来源（例如 mixins 或组件继承）“合并”时，这些选项可能需要特殊的合并逻辑。

app.config.optionMergeStrategies通过使用选项名称作为键将其分配到对象上，可以为自定义选项注册合并策略功能。

合并策略函数接收在父实例和子实例上定义的选项的值，分别作为第一个和第二个参数。

例子

const app = createApp({

// option from self

msg: 'Vue',

// option from a mixin

mixins: [

{

msg: 'Hello '

}

],

mounted() {

// merged options exposed on this.$options

console.log(this.$options.msg)

}

})

// define a custom merge strategy for `msg`

app.config.optionMergeStrategies.msg = (parent, child) => {

return (parent || '') + (child || '')

}

app.mount('#app')

// logs 'Hello Vue'

# 全局 API：通用

版本#

公开当前版本的 Vue。

类型： string

例子

import { version } from 'vue'

console.log(version)

下一个滴答（）#

等待下一次DOM更新刷新的实用程序。

类型

function nextTick(callback?: () => void): Promise<void>

细节

当您在 Vue 中更改状态响应时，生成的 DOM 不会更新同步应用。相反，Vue 会立即更新它们直到“下一个回答”，以确保每个组件只更新一次，无论您进行了多少状态更改。

nextTick()可以在您的状态下使用参数以等待 DOM 更新完成。可以立即更改后更新，也可以等待返回的 Promise。

例子

<script>

import { nextTick } from 'vue'

export default {

data() {

return {

count: 0

}

},

methods: {

async increment() {

this.count++

// DOM not yet updated

console.log(document.getElementById('counter').textContent) // 0

await nextTick()

// DOM is now updated

console.log(document.getElementById('counter').textContent) // 1

}

}

}

</script>

<template>

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

</template>

也可以看看： this.$nextTick()

定义组件（）#

定义具有类型类型推断的 Vue 组件的助手。

类型

function defineComponent(

component: ComponentOptions | ComponentOptions['setup']

): ComponentConstructor

便于阅读，提高类型。

细节

第一个参数需要一个组件选项对象。返回值将是相同的选项对象，因为该函数本质上是一个运行时无操作，仅用于类型推断目的。

请注意，返回类型有点特殊：它将是一个构造函数类型，其实例类型是根据选项推断出的组件实例类型。当返回的类型用作 TSX 中的标记时，这用于类型推断。

您可以从返回类型中提取组件的实例类型（相当于this其选项中的类型），defineComponent()如下所示：

const Foo = defineComponent(/\* ... \*/)

type FooInstance = InstanceType<typeof Foo>

关于 webpack Treeshaking 的注意事项#

因为defineComponent()是一个函数调用，它看起来会对一些构建工具产生副作用，例如 webpack。即使组件从未使用过，这也将防止组件被摇树。

为了告诉 webpack 这个函数调用可以安全地进行 tree-shaking，你可以/\*#\_\_PURE\_\_\*/在函数调用之前添加一个注释符号：

export default /\*#\_\_PURE\_\_\*/ defineComponent(/\* ... \*/)

请注意，如果您使用 Vite，这不是必需的，因为 Rollup（Vite 使用的底层生产捆绑器）足够聪明，可以确定defineComponent()实际上没有副作用，而无需手动注释。

另请参阅： 指南 - 将 Vue 与 TypeScript 结合使用

定义异步组件（）#

定义一个仅在渲染时才延迟加载的异步组件。参数可以是加载器函数，也可以是用于更高级控制加载行为的选项对象。

类型

function defineAsyncComponent(

source: AsyncComponentLoader | AsyncComponentOptions

): Component

type AsyncComponentLoader = () => Promise<Component>

interface AsyncComponentOptions {

loader: AsyncComponentLoader

loadingComponent?: Component

errorComponent?: Component

delay?: number

timeout?: number

suspensible?: boolean

onError?: (

error: Error,

retry: () => void,

fail: () => void,

attempts: number

) => any

}

另请参阅： 指南 - 异步组件

定义自定义元素（）#

这种方法接受与相同的参数，但返回本机自定义的元素类构造函数。defineComponent

类型

function defineCustomElement(

component:

| (ComponentOptions & { styles?: string[] })

| ComponentOptions['setup']

): {

new (props?: object): HTMLElement

}

便于阅读，提高类型。

细节

除了普通的组件选项，defineCustomElement()还支持一个特殊的选项styles，它是一个内联的 CSS 字符串，应该用来提供注入元素外根的 CSS。

返回值是一个自定义的元素构造函数，可以使用。customElements.define()

例子

import { defineCustomElement } from 'vue'

const MyVueElement = defineCustomElement({

/\* component options \*/

})

// Register the custom element.

customElements.define('my-vue-element', MyVueElement)

也可以看看：

指南 - 使用 Vue 制造自定义元素

另请注意，与单个文件组件一起使用时defineCustomElement()需要特殊配置。

# 第3章 组合 API：setup()

笔记

此页面记录了setup组件选项的使用。如果您将 Composition API 与单文件组件一起使用，<script setup>建议使用更简洁和符合人体工程学的语法。

在以下情况下，该setup()钩子用作组件中组合 API 使用的入口点：

在没有构建步骤的情况下使用 Composition API；

在 Options API 组件中与基于 Composition-API 的代码集成。

基本用法#

我们可以使用Reactivity API声明反应状态，并通过从setup(). 返回对象的属性也将在组件实例上可用（如果使用其他选项）：

<script>

import { ref } from 'vue'

export default {

setup() {

const count = ref(0)

// expose to template and other options API hooks

return {

count

}

},

mounted() {

console.log(this.count) // 0

}

}

</script>

<template>

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

</template>

请注意，在模板中访问时，从返回的refssetup会自动浅展开.value，因此您在访问它们时不需要使用。当在 上访问时，它们也以相同的方式展开this。

小费

setup()本身无权访问组件实例 -this将具有undefinedinside的值setup()。您可以从 Options API 访问 Composition-API 公开的值，但反之则不行。

访问道具#

函数中的第一个参数setup是props参数。正如您在标准组件中所期望的那样，函数props内部setup是响应式的，并且会在传入新的 props 时更新。

export default {

props: {

title: String

},

setup(props) {

console.log(props.title)

}

}

请注意，如果您对props对象进行解构，则被解构的变量将失去反应性。因此建议始终以props.xxx.

如果您确实需要解构 props，或者需要将 props 传递给外部函数同时保持反应性，您可以使用toRefs()和toRef()实用程序 API 执行此操作：

import { toRefs, toRef } from 'vue'

export default {

setup(props) {

// turn `props` into an object of refs, then destructure

const { title } = toRefs(props)

// `title` is a ref that tracks `props.title`

console.log(title.value)

// OR, turn a single property on `props` into a ref

const title = toRef(props, 'title')

}

}

设置上下文#

传递给setup函数的第二个参数是一个设置上下文对象。上下文对象公开了其他可能在内部有用的值setup：

export default {

setup(props, context) {

// Attributes (Non-reactive object, equivalent to $attrs)

console.log(context.attrs)

// Slots (Non-reactive object, equivalent to $slots)

console.log(context.slots)

// Emit events (Function, equivalent to $emit)

console.log(context.emit)

// Expose public properties (Function)

console.log(context.expose)

}

}

上下文对象不是反应式的，可以安全地解构：

export default {

setup(props, { attrs, slots, emit, expose }) {

...

}

}

attrs并且slots是在组件本身更新时始终更新的有状态对象。这意味着您应该避免对它们进行解构并始终将属性引用为attrs.x或slots.x。另请注意，与 不同，和的props属性不是反应性的。如果您打算根据对or的更改应用副作用，则应在生命周期挂钩中执行此操作。attrsslotsattrsslotsonBeforeUpdate

公开公共属性#

expose是一个函数，可用于显式限制父组件通过模板 refs访问组件实例时暴露的属性：

export default {

setup(props, { expose }) {

// make the instance "closed" -

// i.e. do not expose anything to the parent

expose()

const publicCount = ref(0)

const privateCount = ref(0)

// selectively expose local state

expose({ count: publicCount })

}

}

与渲染函数一起使用#

setup还可以返回一个渲染函数，该函数可以直接使用在同一范围内声明的反应状态：

import { h, ref } from 'vue'

export default {

setup() {

const count = ref(0)

return () => h('div', count.value)

}

}

返回一个渲染函数可以防止我们返回其他任何东西。在内部这应该不是问题，但是如果我们想通过模板引用将此组件的方法公开给父组件，则可能会出现问题。

我们可以通过调用来解决这个问题expose()：

import { h, ref } from 'vue'

export default {

setup(props, { expose }) {

const count = ref(0)

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

expose({

increment

})

return () => h('div', count.value)

}

}

然后，该increment方法将通过模板引用在父组件中可用。

# 第4章 组合API :反应性 API：核心

也可以看看

为了更好地理解 Reactivity API，建议阅读指南中的以下章节：

Reactivity Fundamentals（API 首选项设置为 Composition API）

深度反应

参考（）#

接受一个内部值并返回一个反应性和可变的 ref 对象，它有一个.value指向内部值的属性。

类型

function ref<T>(value: T): Ref<UnwrapRef<T>>

interface Ref<T> {

value: T

}

细节

ref 对象是可变的 - 即您可以将新值分配给.value. 它也是反应式的——即任何读取操作.value都会被跟踪，而写入操作将触发相关的效果。

如果一个对象被分配为一个 ref 的值，那么该对象将与reactive()进行深度反应。这也意味着如果对象包含嵌套的 ref，它们将被深度展开。

为避免深度转换，请shallowRef()改用。

例子

const count = ref(0)

console.log(count.value) // 0

count.value++

console.log(count.value) // 1

也可以看看：

指南 - 反应性变量ref()

指南 - 打字ref()

计算（）#

接受一个 getter 函数并为 getter 的返回值返回一个只读的响应式ref对象。它还可以使用具有get和set函数的对象来创建可写的 ref 对象。

类型

// read-only

function computed<T>(

getter: () => T,

// see "Computed Debugging" link below

debuggerOptions?: DebuggerOptions

): Readonly<Ref<Readonly<T>>>

// writable

function computed<T>(

options: {

get: () => T

set: (value: T) => void

},

debuggerOptions?: DebuggerOptions

): Ref<T>

例子

创建一个只读的计算参考：

const count = ref(1)

const plusOne = computed(() => count.value + 1)

console.log(plusOne.value) // 2

plusOne.value++ // error

创建一个可写的计算引用：

const count = ref(1)

const plusOne = computed({

get: () => count.value + 1,

set: (val) => {

count.value = val - 1

}

})

plusOne.value = 1

console.log(count.value) // 0

调试：

const plusOne = computed(() => count.value + 1, {

onTrack(e) {

debugger

},

onTrigger(e) {

debugger

}

})

也可以看看：

指南 - 计算属性

指南 - 计算调试

指南 - 打字computed()

反应式（）#

返回对象的反应式代理。

类型

function reactive<T extends object>(target: T): UnwrapNestedRefs<T>

细节

反应式转换是“深度”的：它影响所有嵌套属性。反应式对象还可以在保持反应性的同时深入解开任何作为refs的属性。

还应该注意的是，当引用作为反应数组的元素或本机集合类型（如Map.

为避免深度转换并仅在根级别保留反应性，请改用shallowReactive()。

返回的对象及其嵌套对象被ES Proxy包裹，不等于原始对象。建议只使用响应式代理并避免依赖原始对象。

例子

创建一个反应对象：

const obj = reactive({ count: 0 })

obj.count++

参考展开：

const count = ref(1)

const obj = reactive({ count })

// ref will be unwrapped

console.log(obj.count === count.value) // true

// it will update `obj.count`

count.value++

console.log(count.value) // 2

console.log(obj.count) // 2

// it will also update `count` ref

obj.count++

console.log(obj.count) // 3

console.log(count.value) // 3

请注意，当作为数组或集合元素访问时，refs不会被解包：

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

// need .value here

console.log(books[0].value)

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

// need .value here

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

将ref分配给reactive属性时，该 ref 也将自动展开：

const count = ref(1)

const obj = reactive({})

obj.count = count

console.log(obj.count) // 1

console.log(obj.count === count.value) // true

也可以看看：

指南 - 反应性基础

指南 - 打字reactive()

只读（）#

接受一个对象（反应式或普通）或一个ref并返回一个只读代理到原始的。

类型

function readonly<T extends object>(

target: T

): DeepReadonly<UnwrapNestedRefs<T>>

细节

只读代理很深：访问的任何嵌套属性也将是只读的。它还具有与 相同的引用展开行为reactive()，除了展开的值也将被设为只读。

为避免深度转换，请改用shallowReadonly()。

例子

const original = reactive({ count: 0 })

const copy = readonly(original)

watchEffect(() => {

// works for reactivity tracking

console.log(copy.count)

})

// mutating original will trigger watchers relying on the copy

original.count++

// mutating the copy will fail and result in a warning

copy.count++ // warning!

观看效果（）#

立即运行一个函数，同时响应地跟踪它的依赖关系，并在依赖关系发生变化时重新运行它。

类型

function watchEffect(

effect: (onCleanup: OnCleanup) => void,

options?: WatchEffectOptions

): StopHandle

type OnCleanup = (cleanupFn: () => void) => void

interface WatchEffectOptions {

flush?: 'pre' | 'post' | 'sync' // default: 'pre'

onTrack?: (event: DebuggerEvent) => void

onTrigger?: (event: DebuggerEvent) => void

}

type StopHandle = () => void

细节

第一个参数是要运行的效果函数。效果函数接收可用于注册清理回调的函数。清理回调将在下一次重新运行效果之前调用，并可用于清理无效的副作用，例如挂起的异步请求（参见下面的示例）。

第二个参数是一个可选的选项对象，可用于调整效果的刷新时间或调试效果的依赖关系。

返回值是一个句柄函数，可以调用它来阻止效果再次运行。

例子

const count = ref(0)

watchEffect(() => console.log(count.value))

// -> logs 0

count.value++

// -> logs 1

副作用清理：

watchEffect(async (onCleanup) => {

const { response, cancel } = doAsyncWork(id.value)

// `cancel` will be called if `id` changes

// so that previous pending request will be cancelled

// if not yet completed

onCleanup(cancel)

data.value = await response

})

停止观察者：

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

// when the watcher is no longer needed:

stop()

选项：

watchEffect(() => {}, {

flush: 'post',

onTrack(e) {

debugger

},

onTrigger(e) {

debugger

}

})

另见：

指南 - 观察者

指南 - 观察者调试

watchPostEffect()#

watchEffect()withflush: 'post'选项的别名。

watchSyncEffect()#

watchEffect()withflush: 'sync'选项的别名。

手表（）#

监视一个或多个响应式数据源并在源更改时调用回调函数。

类型

// watching single source

function watch<T>(

source: WatchSource<T>,

callback: WatchCallback<T>,

options?: WatchOptions

): StopHandle

// watching multiple sources

function watch<T>(

sources: WatchSource<T>[],

callback: WatchCallback<T[]>,

options?: WatchOptions

): StopHandle

type WatchCallback<T> = (

value: T,

oldValue: T,

onCleanup: (cleanupFn: () => void) => void

) => void

type WatchSource<T> =

| Ref<T> // ref

| (() => T) // getter

| T extends object

? T

: never // reactive object

interface WatchOptions extends WatchEffectOptions {

immediate?: boolean // default: false

deep?: boolean // default: false

flush?: 'pre' | 'post' | 'sync' // default: 'pre'

onTrack?: (event: DebuggerEvent) => void

onTrigger?: (event: DebuggerEvent) => void

}

为了便于阅读，类型被简化了。

细节

watch()默认情况下是惰性的 - 即仅当监视的源发生更改时才调用回调。

第一个参数是观察者的来源。源可以是以下之一：

返回值的getter函数

一个参考

反应性对象

...或上述数组。

第二个参数是源更改时将调用的回调。回调接收三个参数：新值、旧值和用于注册副作用清除回调的函数。清理回调将在下一次重新运行效果之前调用，并可用于清理无效的副作用，例如挂起的异步请求。

当观察多个源时，回调接收到两个数组，其中包含与源数组对应的新/旧值。

第三个可选参数是一个支持以下选项的选项对象：

immediate：在观察者创建时立即触发回调。旧值将undefined在第一次调用时出现。

deep：如果源是对象，则强制深度遍历源，以便回调触发深度突变。请参阅深度观察者。

flush：调整回调的刷新时间。请参阅回调刷新时间。

onTrack / onTrigger: 调试观察者的依赖。请参阅观察程序调试。

相比watchEffect()，watch()允许我们：

懒惰地执行副作用；

更具体地说明应该触发观察者重新运行的状态；

访问监视状态的先前值和当前值。

例子

观察吸气剂：

const state = reactive({ count: 0 })

watch(

() => state.count,

(count, prevCount) => {

/\* ... \*/

}

)

观看裁判：

const count = ref(0)

watch(count, (count, prevCount) => {

/\* ... \*/

})

当观察多个源时，回调接收包含与源数组对应的新/旧值的数组：

watch([fooRef, barRef], ([foo, bar], [prevFoo, prevBar]) => {

/\* ... \*/

})

使用 getter 源时，仅当 getter 的返回值发生更改时才会触发观察程序。如果您希望回调即使在深度突变时也能触发，您需要使用 . 显式强制观察者进入深度模式{ deep: true }。请注意，在深度模式下，如果回调是由深度突变触发的，则新值和旧值将是同一个对象：

const state = reactive({ count: 0 })

watch(

() => state,

(newValue, oldValue) => {

// newValue === oldValue

},

{ deep: true }

)

当直接观察反应对象时，观察者自动处于深度模式：

const state = reactive({ count: 0 })

watch(state, () => {

/\* triggers on deep mutation to state \*/

})

watch()共享相同的刷新时间和调试选项watchEffect()：

watch(source, callback, {

flush: 'post',

onTrack(e) {

debugger

}

})

# 第5章 组合API：在本页面

当前页面的目录

是参考（）

未引用（）

toRef()

toRefs()

是代理（）

反应式（）

isReadonly()

赞助商

单应用

VueMastery

VueSchool

车辆

热情的人

故事块

离子

纽斯特

翻新的

黄油CMS

赛诺菲

代码字典

vuejs.de 配置

你的标志

前端工程师

KNIME ·德国柏林

Vue/Laravel 全栈开发人员

Delphi Consulting, LLC ·远程

反应性 API：实用程序#

是参考（）#

检查一个值是否是一个 ref 对象。

类型

function isRef<T>(r: Ref<T> | unknown): r is Ref<T>

注意返回类型是一个类型谓词，这意味着isRef可以用作类型保护：

let foo: unknown

if (isRef(foo)) {

// foo's type is narrowed to Ref<unknown>

foo.value

}

未引用（）#

如果参数是 ref，则返回内部值，否则返回参数本身。这是一个糖函数val = isRef(val) ? val.value : val。

类型

function unref<T>(ref: T | Ref<T>): T

例子

function useFoo(x: number | Ref<number>) {

const unwrapped = unref(x)

// unwrapped is guaranteed to be number now

}

toRef()#

可用于为源响应对象上的属性创建 ref。创建的 ref 与其源属性同步：改变源属性将更新 ref，反之亦然。

类型

function toRef<T extends object, K extends keyof T>(

object: T,

key: K,

defaultValue?: T[K]

): ToRef<T[K]>

type ToRef<T> = T extends Ref ? T : Ref<T>

例子

const state = reactive({

foo: 1,

bar: 2

})

const fooRef = toRef(state, 'foo')

// mutating the ref updates the original

fooRef.value++

console.log(state.foo) // 2

// mutating the original also updates the ref

state.foo++

console.log(fooRef.value) // 3

请注意，这不同于：

const fooRef = ref(state.foo)

上面的 ref没有与 同步state.foo，因为ref()接收一个纯数字值。

toRef()当您想将 prop 的 ref 传递给可组合函数时很有用：

<script setup>

import { toRef } from 'vue'

const props = defineProps(/\* ... \*/)

// convert `props.foo` into a ref, then pass into

// a composable

useSomeFeature(toRef(props, 'foo'))

</script>

当toRef与组件道具一起使用时，关于改变道具的通常限制仍然适用。尝试为 ref 分配新值相当于尝试直接修改 prop，并且是不允许的。在这种情况下，您可能需要考虑使用computedwithget和set代替。有关更多信息，请参阅与组件一起使用v-model的指南。

toRef()即使源属性当前不存在，也会返回一个可用的引用。这使得使用可选属性成为可能，这些属性不会被toRefs.

toRefs()#

将响应式对象转换为普通对象，其中结果对象的每个属性都是指向原始对象相应属性的 ref。每个单独的 ref 都是使用创建的toRef()。

类型

function toRefs<T extends object>(

object: T

): {

[K in keyof T]: ToRef<T[K]>

}

type ToRef = T extends Ref ? T : Ref<T>

例子

const state = reactive({

foo: 1,

bar: 2

})

const stateAsRefs = toRefs(state)

/\*

Type of stateAsRefs: {

foo: Ref<number>,

bar: Ref<number>

}

\*/

// The ref and the original property is "linked"

state.foo++

console.log(stateAsRefs.foo.value) // 2

stateAsRefs.foo.value++

console.log(state.foo) // 3

toRefs在从可组合函数返回反应性对象时很有用，以便使用组件可以解构/扩展返回的对象而不会失去反应性：

function useFeatureX() {

const state = reactive({

foo: 1,

bar: 2

})

// ...logic operating on state

// convert to refs when returning

return toRefs(state)

}

// can destructure without losing reactivity

const { foo, bar } = useFeatureX()

toRefs只会在调用时为源对象上可枚举的属性生成引用。要为可能尚不存在的属性创建 ref，请toRef改用。

是代理（）#

检查对象是否是由 、 或 所创建reactive()的readonly()代理。shallowReactive()shallowReadonly()

类型

function isProxy(value: unknown): boolean

反应式（）#

检查对象是否是由reactive()or创建的代理shallowReactive()。

类型

function isReactive(value: unknown): boolean

isReadonly()#

检查对象是否是由readonly()or创建的代理shallowReadonly()。

类型

function isReadonly(value: unknown): boolean

# 第6章 组合API：反应性 API：高级

浅参考（）#

的浅版本ref()。

类型

function shallowRef<T>(value: T): ShallowRef<T>

interface ShallowRef<T> {

value: T

}

细节

与 不同ref()，浅层 ref 的内部值按原样存储和公开，不会被深度响应。只有.value访问是被动的。

shallowRef()通常用于大型数据结构的性能优化，或与外部状态管理系统的集成。

例子

const state = shallowRef({ count: 1 })

// does NOT trigger change

state.value.count = 2

// does trigger change

state.value = { count: 2 }

也可以看看：

指南 - 减少大型不可变结构的反应性开销

指南 - 与外部状态系统集成

触发器引用（）#

强制触发效果取决于浅层 ref。这通常在对浅层参考的内部值进行深度突变后使用。

类型

function triggerRef(ref: ShallowRef): void

例子

const shallow = shallowRef({

greet: 'Hello, world'

})

// Logs "Hello, world" once for the first run-through

watchEffect(() => {

console.log(shallow.value.greet)

})

// This won't trigger the effect because the ref is shallow

shallow.value.greet = 'Hello, universe'

// Logs "Hello, universe"

triggerRef(shallow)

自定义参考（）#

创建一个自定义的 ref，显式控制其依赖跟踪和更新触发。

类型

function customRef<T>(factory: CustomRefFactory<T>): Ref<T>

type CustomRefFactory<T> = (

track: () => void,

trigger: () => void

) => {

get: () => T

set: (value: T) => void

}

细节

customRef()需要一个工厂函数，它接收track和trigger函数作为参数，并应该返回一个带有get和set方法的对象。

一般来说，track()应该叫 inside get()，而且trigger()应该叫 inside set()。但是，您可以完全控制何时调用它们，或者是否应该调用它们。

例子

创建一个 debounced ref，它只在最近一次 set 调用后的某个超时后更新值：

import { customRef } from 'vue'

export function useDebouncedRef(value, delay = 200) {

let timeout

return customRef((track, trigger) => {

return {

get() {

track()

return value

},

set(newValue) {

clearTimeout(timeout)

timeout = setTimeout(() => {

value = newValue

trigger()

}, delay)

}

}

})

}

在组件中的用法：

<script setup>

import { useDebouncedRef } from './debouncedRef'

const text = useDebouncedRef('hello')

</script>

<template>

<input v-model="text" />

</template>

在操场上试试

浅反应（）#

的浅版本reactive()。

类型

function shallowReactive<T extends object>(target: T): T

细节

与 不同reactive()的是，没有深度转换：只有根级属性对于浅反应对象是反应性的。属性值按原样存储和公开 - 这也意味着具有 ref 值的属性不会自动展开。

谨慎使用

浅层数据结构只能用于组件中的根级状态。避免将其嵌套在深层反应对象中，因为它会创建一个具有不一致反应行为的树，这可能难以理解和调试。

例子

const state = shallowReactive({

foo: 1,

nested: {

bar: 2

}

})

// mutating state's own properties is reactive

state.foo++

// ...but does not convert nested objects

isReactive(state.nested) // false

// NOT reactive

state.nested.bar++

浅只读（）#

的浅版本readonly()。

类型

function shallowReadonly<T extends object>(target: T): Readonly<T>

细节

与 不同readonly()的是，没有深度转换：只有根级属性是只读的。属性值按原样存储和公开 - 这也意味着具有 ref 值的属性不会自动展开。

谨慎使用

浅层数据结构只能用于组件中的根级状态。避免将其嵌套在深层反应对象中，因为它会创建一个具有不一致反应行为的树，这可能难以理解和调试。

例子

const state = shallowReadonly({

foo: 1,

nested: {

bar: 2

}

})

// mutating state's own properties will fail

state.foo++

// ...but works on nested objects

isReadonly(state.nested) // false

// works

state.nested.bar++

toRaw()#

返回 Vue 创建的代理的原始原始对象。

类型

function toRaw<T>(proxy: T): T

细节

toRaw()可以从由 、 或 . 创建的代理reactive()返回readonly()原始shallowReactive()对象shallowReadonly()。

这是一个逃生舱口，可用于临时读取而不会产生代理访问/跟踪开销或写入而不触发更改。不建议持有对原始对象的持久引用。谨慎使用。

例子

const foo = {}

const reactiveFoo = reactive(foo)

console.log(toRaw(reactiveFoo) === foo) // true

标记原始（）#

标记一个对象，使其永远不会转换为代理。返回对象本身。

类型

function markRaw<T extends object>(value: T): T

例子

const foo = markRaw({})

console.log(isReactive(reactive(foo))) // false

// also works when nested inside other reactive objects

const bar = reactive({ foo })

console.log(isReactive(bar.foo)) // false

谨慎使用

markRaw()和浅 API，例如shallowReactive()允许您有选择地退出默认的深度反应/只读转换，并在状态图中嵌入原始的、非代理的对象。它们可以用于多种原因：

有些值根本不应该是被动的，例如复杂的 3rd 方类实例或 Vue 组件对象。

在呈现具有不可变数据源的大型列表时，跳过代理转换可以提高性能。

它们被认为是高级的，因为原始选择退出仅在根级别，因此如果您将嵌套的、未标记的原始对象设置为反应性对象，然后再次访问它，您将获得代理版本。这可能会导致身份风险——即执行依赖于对象身份但同时使用同一对象的原始版本和代理版本的操作：

const foo = markRaw({

nested: {}

})

const bar = reactive({

// although `foo` is marked as raw, foo.nested is not.

nested: foo.nested

})

console.log(foo.nested === bar.nested) // false

身份风险通常很少见。但是，要正确使用这些 API，同时安全地避免身份危害，需要对反应系统的工作原理有深入的了解。

效果范围（）#

创建一个效果范围对象，该对象可以捕获在其中创建的反应性效果（即计算和观察者），以便可以将这些效果放在一起。该 API 的详细用例请查阅其对应的RFC。

类型

function effectScope(detached?: boolean): EffectScope

interface EffectScope {

run<T>(fn: () => T): T | undefined // undefined if scope is inactive

stop(): void

}

例子

const scope = effectScope()

scope.run(() => {

const doubled = computed(() => counter.value \* 2)

watch(doubled, () => console.log(doubled.value))

watchEffect(() => console.log('Count: ', doubled.value))

})

// to dispose all effects in the scope

scope.stop()

获取当前范围（）#

如果有，则返回当前激活的效果范围。

类型

function getCurrentScope(): EffectScope | undefined

onScopeDispose()#

在当前活动效果范围上注册一个 dispose 回调。当关联的效果范围停止时，将调用回调。

此方法可用作可onUnmounted重用组合函数的非组件耦合替换，因为每个 Vue 组件的setup()函数也在效果范围内调用。

类型

function onScopeDispose(fn: () => void): void

# 第7章 组合 API：生命周期挂钩

期挂钩。

onMounted()#

注册一个回调，以便在组件安装后调用。

类型

function onMounted(callback: () => void): void

细节

组件在以下情况下被视为已安装：

它的所有同步子组件都已安装（不包括异步组件或<Suspense>树内的组件）。

它自己的 DOM 树已创建并插入到父容器中。请注意，如果应用程序的根容器也在文档内，它只保证组件的 DOM 树在文档内。

这个钩子通常用于执行需要访问组件渲染的 DOM 的副作用，或者用于将与 DOM 相关的代码限制在服务器渲染应用程序中的客户端。

在服务器端渲染期间不会调用此挂钩。

例子

通过模板引用访问元素：

<script setup>

import { ref, onMounted } from 'vue'

const el = ref()

onMounted(() => {

el.value // <div>

})

</script>

<template>

<div ref="el"></div>

</template>

onUpdated()#

注册一个回调，以便在组件由于响应状态更改而更新其 DOM 树后调用。

类型

function onUpdated(callback: () => void): void

细节

父组件的更新钩子在其子组件之后被调用。

在组件的任何 DOM 更新后调用此钩子，这可能是由不同的状态更改引起的。如果您需要在特定状态更改后访问更新的 DOM，请改用nextTick()。

在服务器端渲染期间不会调用此挂钩。

警告

不要在更新的钩子中改变组件状态 - 这可能会导致无限更新循环！

例子

访问更新的 DOM：

<script setup>

import { ref, onUpdated } from 'vue'

const count = ref(0)

onUpdated(() => {

// text content should be the same as current `count.value`

console.log(document.getElementById('count').textContent)

})

</script>

<template>

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

</template>

onUnmounted()#

注册一个在组件被卸载后调用的回调。

类型

function onUnmounted(callback: () => void): void

细节

组件在以下情况下被视为已卸载：

它的所有子组件都已卸载。

其所有相关的反应效果（渲染效果和在 期间创建的计算/观察者setup()）都已停止。

使用此钩子清除手动创建的副作用，例如计时器、DOM 事件侦听器或服务器连接。

在服务器端渲染期间不会调用此挂钩。

例子

<script setup>

import { onMounted, onUnmounted } from 'vue'

let intervalId

onMounted(() => {

intervalId = setInterval(() => {

// ...

})

})

onUnmounted(() => clearInterval(intervalId))

</script>

onBeforeMount()#

在要安装组件之前注册要调用的钩子。

类型

function onBeforeMount(callback: () => void): void

细节

当这个钩子被调用时，组件已经完成了它的反应状态的设置，但是还没有创建 DOM 节点。它即将第一次执行它的 DOM 渲染效果。

在服务器端渲染期间不会调用此挂钩。

onBeforeUpdate()#

由于响应状态更改，在组件即将更新其 DOM 树之前注册要调用的钩子。

类型

function onBeforeUpdate(callback: () => void): void

细节

这个钩子可用于在 Vue 更新 DOM 之前访问 DOM 状态。在这个钩子中修改组件状态也是安全的。

在服务器端渲染期间不会调用此挂钩。

onBeforeUnmount()#

在要卸载组件实例之前注册要调用的挂钩。

类型

function onBeforeUnmount(callback: () => void): void

细节

调用此钩子时，组件实例仍然可以正常工作。

在服务器端渲染期间不会调用此挂钩。

onErrorCaptured()#

当捕获到从后代组件传播的错误时，注册要调用的钩子。

类型

function onErrorCaptured(callback: ErrorCapturedHook): void

type ErrorCapturedHook = (

err: unknown,

instance: ComponentPublicInstance | null,

info: string

) => boolean | void

细节

可以从以下来源捕获错误：

组件渲染

事件处理程序

生命周期钩子

setup()功能

观察者

自定义指令钩子

过渡挂钩

钩子接收三个参数：错误、触发错误的组件实例和指定错误源类型的信息字符串。

您可以修改组件状态以errorCaptured()向用户显示错误状态。但是，重要的是错误状态不应呈现导致错误的原始内容；否则组件将被抛出无限渲染循环。

钩子可以返回false以阻止错误进一步传播。请参阅下面的错误传播详细信息。

错误传播规则

默认情况下，app.config.errorHandler如果已定义，所有错误仍会发送到应用程序级别，因此这些错误仍可以在一个地方报告给分析服务。

errorCaptured如果组件的继承链或父链上存在多个钩子，则所有这些钩子都会在相同的错误上被调用，按从下到上的顺序。这类似于原生 DOM 事件的冒泡机制。

如果errorCaptured钩子本身抛出一个错误，这个错误和原始捕获的错误都会被发送到app.config.errorHandler.

errorCaptured挂钩可以返回以false防止错误进一步传播。这实质上是在说“这个错误已经被处理，应该被忽略”。它将防止任何额外的errorCaptured钩子或被app.config.errorHandler此错误调用。

onRenderTracked()#

注册一个调试钩子，当组件的渲染效果跟踪到反应性依赖项时调用。

此挂钩仅在开发模式下运行，在服务器端渲染期间不调用。

类型

function onRenderTracked(callback: DebuggerHook): void

type DebuggerHook = (e: DebuggerEvent) => void

type DebuggerEvent = {

effect: ReactiveEffect

target: object

type: TrackOpTypes /\* 'get' | 'has' | 'iterate' \*/

key: any

}

另请参阅： 深度反应性

onRenderTriggered()#

当响应式依赖触发组件的渲染效果重新运行时，注册要调用的调试挂钩。

此挂钩仅在开发模式下运行，在服务器端渲染期间不调用。

类型

function onRenderTriggered(callback: DebuggerHook): void

type DebuggerHook = (e: DebuggerEvent) => void

type DebuggerEvent = {

effect: ReactiveEffect

target: object

type: TriggerOpTypes /\* 'set' | 'add' | 'delete' | 'clear' \*/

key: any

newValue?: any

oldValue?: any

oldTarget?: Map<any, any> | Set<any>

}

另请参阅： 深度反应性

onActivated()#

注册一个回调，以便在组件实例插入 DOM 作为由 缓存的树的一部分后调用<KeepAlive>。

在服务器端渲染期间不会调用此挂钩。

类型

function onActivated(callback: () => void): void

另请参阅： 指南 - 缓存实例的生命周期

onDeactivated()#

注册一个回调，以便在组件实例从 DOM 中删除后调用，作为由 缓存的树的一部分<KeepAlive>。

在服务器端渲染期间不会调用此挂钩。

类型

function onDeactivated(callback: () => void): void

另请参阅： 指南 - 缓存实例的生命周期

onServerPrefetch()#

在组件实例在服务器上呈现之前注册一个要解析的异步函数。

类型

function onServerPrefetch(callback: () => Promise<any>): void

细节

如果回调返回一个 Promise，服务器渲染器将等到 Promise 被解析后再渲染组件。

此钩子仅在服务器端渲染期间调用，可用于执行仅服务器数据获取。

例子

<script setup>

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

const data = ref(null)

onServerPrefetch(async () => {

// component is rendered as part of the initial request

// pre-fetch data on server as it is faster than on the client

data.value = await fetchOnServer(/\* ... \*/)

})

onMounted(async () => {

if (!data.value) {

// if data is null on mount, it means the component

// is dynamically rendered on the client. Perform a

// client-side fetch instead.

data.value = await fetchOnClient(/\* ... \*/)

}

})

</script>

# 第8章 组合 API：依赖注入

提供（）#

提供一个可以由后代组件注入的值。

类型

function provide<T>(key: InjectionKey<T> | string, value: T): void

细节

provide()接受两个参数：键，可以是字符串或符号，以及要注入的值。

使用 TypeScript 时，键可以是转换为的符号InjectionKey- Vue 提供的扩展实用程序类型，Symbol可用于在provide()和之间同步值类型inject()。

与生命周期挂钩注册 API 类似，provide()必须在组件setup()阶段同步调用。

例子

<script setup>

import { ref, provide } from 'vue'

import { fooSymbol } from './injectionSymbols'

// provide static value

provide('foo', 'bar')

// provide reactive value

const count = ref(0)

provide('count', count)

// provide with Symbol keys

provide(fooSymbol, count)

</script>

另见：

指南 - 提供/注入

指南 - 键入提供/注入

注入（）#

注入由祖先组件或应用程序提供的值（通过app.provide()）。

类型

// without default value

function inject<T>(key: InjectionKey<T> | string): T | undefined

// with default value

function inject<T>(key: InjectionKey<T> | string, defaultValue: T): T

// with factory

function inject<T>(

key: InjectionKey<T> | string,

defaultValue: () => T,

treatDefaultAsFactory: true

): T

细节

第一个参数是注入密钥。Vue 将沿着父链查找具有匹配键的提供值。如果父链中的多个组件提供相同的密钥，则最接近注入组件的一个将“遮蔽”链上更高的那些。如果没有找到具有匹配键的值，则inject()返回undefined，除非提供了默认值。

第二个参数是可选的，是在没有找到匹配值时使用的默认值。它也可以是一个工厂函数来返回创建成本高的值。如果默认值是一个函数，则false必须作为第三个参数传递，以指示该函数应该用作值而不是工厂。

与生命周期挂钩注册 API 类似，inject()必须在组件setup()阶段同步调用。

使用 TypeScript 时，键的类型可以是InjectionKey- Vue 提供的扩展实用程序类型，Symbol可用于在provide()和之间同步值类型inject()。

例子

假设父组件已提供如上provide()例所示的值：

<script setup>

import { inject } from 'vue'

import { fooSymbol } from './injectionSymbols'

// inject static value with default

const foo = inject('foo')

// inject reactive value

const count = inject('count')

// inject with Symbol keys

const foo2 = inject(fooSymbol)

// inject with default value

const bar = inject('foo', 'default value')

// inject with default value factory

const baz = inject('foo', () => new Map())

// inject with function default value, by passing the 3rd argument

const fn = inject('function', () => {}, false)

</script>

# 第9章 选项API：状态

数据#

返回组件实例的初始反应状态的函数。

类型

interface ComponentOptions {

data?(

this: ComponentPublicInstance,

vm: ComponentPublicInstance

): object

}

细节

该函数预计将返回一个普通的 JavaScript 对象，该对象将由 Vue.js 进行响应。创建实例后，响应式数据对象可以作为this.$data. 组件实例还代理在数据对象上找到的所有属性，因此this.a将等效于this.$data.a.

所有顶级数据属性都必须包含在返回的数据对象中。可以添加新属性this.$data，但不建议这样做。如果所需的属性值尚不可用，则应包含一个空值（例如undefinedor null）作为占位符，以确保 Vue 知道该属性存在。

以\_或$不会在组件实例上代理的属性，因为它们可能与 Vue 的内部属性和 API 方法发生冲突。您必须以this.$data.\_property.

不建议返回具有自己有状态行为的对象，例如浏览器 API 对象和原型属性。理想情况下，返回的对象应该是一个仅表示组件状态的普通对象。

例子

export default {

data() {

return { a: 1 }

},

created() {

console.log(this.a) // 1

console.log(this.$data) // { a: 1 }

}

}

请注意，如果您使用带有data属性的箭头函数，this则不会是组件的实例，但您仍然可以将实例作为函数的第一个参数访问：

data: (vm) => ({ a: vm.myProp })

另请参阅： 深度反应性

道具#

声明组件的 props。

类型

interface ComponentOptions {

props?: ArrayPropsOptions | ObjectPropsOptions

}

type ArrayPropsOptions = string[]

type ObjectPropsOptions = { [key: string]: Prop }

type Prop<T = any> = PropOptions<T> | PropType<T> | null

interface PropOptions<T> {

type?: PropType<T>

required?: boolean

default?: T | ((rawProps: object) => T)

validator?: (value: unknown) => boolean

}

type PropType<T> = { new (): T } | { new (): T }[]

为了便于阅读，类型被简化了。

细节

在 Vue 中，所有组件的 props 都需要显式声明。组件 props 可以以两种形式声明：

使用字符串数组的简单形式

使用对象的完整形式，其中每个属性键是道具的名称，值是道具的类型（构造函数）或高级选项。

使用基于对象的语法，每个 prop 可以进一步定义以下选项：

type: 可以是以下本机构造函数之一：String, Number, Boolean, Array, Object, Date, Function, Symbol, 任何自定义构造函数或它们的数组。在开发模式下，Vue 会检查 prop 的值是否与声明的类型匹配，如果不匹配则会抛出警告。有关更多详细信息，请参阅道具验证。

另请注意，具有Boolean类型的道具会影响其在开发和生产中的值转换行为。有关更多详细信息，请参阅布尔转换。

default: 指定 prop 没有被父级传递或有值时的默认undefined值。必须使用工厂函数返回对象或数组默认值。工厂函数还接收原始 props 对象作为参数。

required：定义是否需要道具。在非生产环境中，如果该值是真实的并且没有传递道具，则会抛出控制台警告。

validator: 以 prop 值作为唯一参数的自定义验证器函数。在开发模式下，如果此函数返回一个虚假值（即验证失败），将引发控制台警告。

例子

简单声明：

export default {

props: ['size', 'myMessage']

}

带有验证的对象声明：

export default {

props: {

// type check

height: Number,

// type check plus other validations

age: {

type: Number,

default: 0,

required: true,

validator: (value) => {

return value >= 0

}

}

}

}

参见： 道具

计算的#

声明要在组件实例上公开的计算属性。

类型

interface ComponentOptions {

computed?: {

[key: string]: ComputedGetter<any> | WritableComputedOptions<any>

}

}

type ComputedGetter<T> = (

this: ComponentPublicInstance,

vm: ComponentPublicInstance

) => T

type ComputedSetter<T> = (

this: ComponentPublicInstance,

value: T

) => void

type WritableComputedOptions<T> = {

get: ComputedGetter<T>

set: ComputedSetter<T>

}

细节

该选项接受一个对象，其中键是计算属性的名称，值是计算的 getter，或具有get和set方法的对象（用于可写计算属性）。

所有 getter 和 setter 的this上下文都会自动绑定到组件实例。

请注意，如果您使用带有计算属性的箭头函数，this则不会指向组件的实例，但您仍然可以将实例作为函数的第一个参数访问：

export default {

computed: {

aDouble: (vm) => vm.a \* 2

}

}

例子

export default {

data() {

return { a: 1 }

},

computed: {

// readonly

aDouble() {

return this.a \* 2

},

// writable

aPlus: {

get() {

return this.a + 1

},

set(v) {

this.a = v - 1

}

}

},

created() {

console.log(this.aDouble) // => 2

console.log(this.aPlus) // => 2

this.aPlus = 3

console.log(this.a) // => 2

console.log(this.aDouble) // => 4

}

}

另请参阅： 计算属性

方法#

声明要混合到组件实例中的方法。

类型

interface ComponentOptions {

methods?: {

[key: string]: (this: ComponentPublicInstance, ...args: any[]) => any

}

}

细节

声明的方法可以直接在组件实例上访问，也可以在模板表达式中使用。所有方法的this上下文都会自动绑定到组件实例，即使在传递时也是如此。

声明方法时避免使用箭头函数，因为它们无法通过this.

例子

export default {

data() {

return { a: 1 }

},

methods: {

plus() {

this.a++

}

},

created() {

this.plus()

console.log(this.a) // => 2

}

}

另请参阅： 事件处理

手表#

声明要在数据更改时调用的监视回调。

类型

interface ComponentOptions {

watch?: {

[key: string]: WatchOptionItem | WatchOptionItem[]

}

}

type WatchOptionItem = string | WatchCallback | ObjectWatchOptionItem

type WatchCallback<T> = (

value: T,

oldValue: T,

onCleanup: (cleanupFn: () => void) => void

) => void

type ObjectWatchOptionItem = {

handler: WatchCallback | string

immediate?: boolean // default: false

deep?: boolean // default: false

flush?: 'pre' | 'post' | 'sync' // default: 'pre'

onTrack?: (event: DebuggerEvent) => void

onTrigger?: (event: DebuggerEvent) => void

}

为了便于阅读，类型被简化了。

细节

该watch选项需要一个对象，其中键是要观察的反应组件实例属性（例如，通过data或声明的属性computed） - 值是相应的回调。回调接收监视源的新值和旧值。

除了根级属性之外，键还可以是简单的点分隔路径，例如a.b.c. 请注意，此用法不支持复杂表达式 - 仅支持以点分隔的路径。如果您需要查看复杂的数据源，请改用命令式$watch()API。

该值也可以是方法名称的字符串（通过 声明methods），或包含附加选项的对象。使用对象语法时，应在handler字段下声明回调。其他选项包括：

immediate：在观察者创建时立即触发回调。旧值将undefined在第一次调用时出现。

deep：如果源是对象或数组，则强制对源进行深度遍历，以便回调触发深度突变。请参阅深度观察者。

flush：调整回调的刷新时间。请参阅回调刷新时间。

onTrack / onTrigger: 调试观察者的依赖。请参阅观察程序调试。

在声明 watch 回调时避免使用箭头函数，因为它们无法通过this.

例子

export default {

data() {

return {

a: 1,

b: 2,

c: {

d: 4

},

e: 5,

f: 6

}

},

watch: {

// watching top-level property

a(val, oldVal) {

console.log(`new: ${val}, old: ${oldVal}`)

},

// string method name

b: 'someMethod',

// the callback will be called whenever any of the watched object properties change regardless of their nested depth

c: {

handler(val, oldVal) {

console.log('c changed')

},

deep: true

},

// watching a single nested property:

'c.d': function (val, oldVal) {

// do something

},

// the callback will be called immediately after the start of the observation

e: {

handler(val, oldVal) {

console.log('e changed')

},

immediate: true

},

// you can pass array of callbacks, they will be called one-by-one

f: [

'handle1',

function handle2(val, oldVal) {

console.log('handle2 triggered')

},

{

handler: function handle3(val, oldVal) {

console.log('handle3 triggered')

}

/\* ... \*/

}

]

},

methods: {

someMethod() {

console.log('b changed')

},

handle1() {

console.log('handle 1 triggered')

}

},

created() {

this.a = 3 // => new: 3, old: 1

}

}

另见： 观察者

发出#

声明组件发出的自定义事件。

类型

interface ComponentOptions {

emits?: ArrayEmitsOptions | ObjectEmitsOptions

}

type ArrayEmitsOptions = string[]

type ObjectEmitsOptions = { [key: string]: EmitValidator | null }

type EmitValidator = (...args: unknown[]) => boolean

细节

发出的事件可以以两种形式声明：

使用字符串数组的简单形式

使用对象的完整形式，其中每个属性键是事件的名称，值是null或者是验证器函数。

验证函数将接收传递给组件$emit调用的附加参数。例如，如果this.$emit('foo', 1)被调用，则对应的验证器foo将接收参数1。验证器函数应返回一个布尔值以指示事件参数是否有效。

请注意，该emits选项会影响哪些事件侦听器被视为组件事件侦听器，而不是原生 DOM 事件侦听器。声明事件的侦听器将从组件的$attrs对象中删除，因此它们不会传递到组件的根元素。有关更多详细信息，请参见Fallthrough Attributes。

例子

数组语法：

export default {

emits: ['check'],

created() {

this.$emit('check')

}

}

对象语法：

export default {

emits: {

// no validation

click: null,

// with validation

submit: (payload) => {

if (payload.email && payload.password) {

return true

} else {

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

return false

}

}

}

}

另请参阅： Fallthrough 属性

暴露#

当父级通过模板引用访问组件实例时，声明公开的公共属性。

类型

interface ComponentOptions {

expose?: string[]

}

细节

默认情况下，当通过 、 或模板引用访问时，组件实例会将所有实例属性公开给父$parent级$root。这可能是不可取的，因为组件很可能具有应保持私有的内部状态或方法以避免紧密耦合。

该expose选项需要一个属性名称字符串列表。使用时expose，只有明确列出的属性才会在组件的公共实例上公开。

expose仅影响用户定义的属性 - 它不会过滤掉内置组件实例属性。

例子

export default {

// only `publicMethod` will be available on the public instance

expose: ['publicMethod'],

methods: {

publicMethod() {

// ...

},

privateMethod() {

// ...

}

}

}

# 第10章 选项API：渲染

模板#

组件的字符串模板。

类型

interface ComponentOptions {

template?: string

}

细节

通过该template选项提供的模板将在运行时即时编译。只有在使用包含模板编译器的 Vue 构建时才支持它。模板编译器不包含在名称中包含单词的 Vue 构建中runtime，例如vue.runtime.esm-bundler.js. 有关不同构建的更多详细信息，请参阅dist 文件指南。

如果字符串以它开头，#它将用作 aquerySelector并将所选元素的innerHTML用作模板字符串。这允许使用本机<template>元素创作源模板。

如果该render选项也存在于同一组件中，template则将被忽略。

如果您的应用程序的根组件没有指定templateorrender选项，Vue 将尝试使用innerHTML已安装元素的 代替作为模板。

安全说明

仅使用您可以信任的模板来源。不要使用用户提供的内容作为您的模板。有关详细信息，请参阅安全指南。

使成为#

以编程方式返回组件的虚拟 DOM 树的函数。

类型

interface ComponentOptions {

render?(this: ComponentPublicInstance) => VNodeChild

}

type VNodeChild = VNodeChildAtom | VNodeArrayChildren

type VNodeChildAtom =

| VNode

| string

| number

| boolean

| null

| undefined

| void

type VNodeArrayChildren = (VNodeArrayChildren | VNodeChildAtom)[]

细节：

render是字符串模板的替代方案，它允许您利用 JavaScript 的全部编程功能来声明组件的渲染输出。

预编译的模板，例如单文件组件中的模板，render在构建时被编译到选项中。如果组件中同时存在 和 ，render则优先级更高。templaterender

也可以看看：

渲染机制

渲染函数

编译器选项#

为组件的模板配置运行时编译器选项。

类型

interface ComponentOptions {

compilerOptions?: {

isCustomElement?: (tag: string) => boolean

whitespace?: 'condense' | 'preserve' // default: 'condense'

delimiters?: [string, string] // default: ['{{', '}}']

comments?: boolean // default: false

}

}

细节

此配置选项仅在使用完整版本（即vue.js可以在浏览器中编译模板的独立版本）时受到尊重。它支持与应用级app.config.compilerOptions相同的选项，并且对当前组件具有更高的优先级。

另请参阅： app.config.compilerOptions

# 第11章 选项API：生命周期

之前创建#

在实例初始化时调用。

类型

interface ComponentOptions {

beforeCreate?(this: ComponentPublicInstance): void

}

细节

在实例初始化时立即调用，在 props 解析之后，在处理其他选项（例如data()or ）之前computed。

请注意，setup()Composition API 的钩子在任何 Options API 钩子之前被调用，甚至beforeCreate().

创建#

在实例处理完所有与状态相关的选项后调用。

类型

interface ComponentOptions {

created?(this: ComponentPublicInstance): void

}

细节

调用此钩子时，已设置以下内容：反应数据、计算属性、方法和观察者。但是，安装阶段尚未开始，该$el属性将不可用。

安装前#

在要安装组件之前调用。

类型

interface ComponentOptions {

beforeMount?(this: ComponentPublicInstance): void

}

细节

当这个钩子被调用时，组件已经完成了它的反应状态的设置，但是还没有创建 DOM 节点。它即将第一次执行它的 DOM 渲染效果。

在服务器端渲染期间不会调用此挂钩。

安装#

在安装组件后调用。

类型

interface ComponentOptions {

mounted?(this: ComponentPublicInstance): void

}

细节

组件在以下情况下被视为已安装：

它的所有同步子组件都已安装（不包括异步组件或<Suspense>树内的组件）。

它自己的 DOM 树已创建并插入到父容器中。请注意，如果应用程序的根容器也在文档内，它只保证组件的 DOM 树在文档内。

这个钩子通常用于执行需要访问组件渲染的 DOM 的副作用，或者用于将与 DOM 相关的代码限制在服务器渲染应用程序中的客户端。

在服务器端渲染期间不会调用此挂钩。

更新前#

由于响应状态更改，在组件即将更新其 DOM 树之前调用。

类型

interface ComponentOptions {

beforeUpdate?(this: ComponentPublicInstance): void

}

细节

这个钩子可用于在 Vue 更新 DOM 之前访问 DOM 状态。在这个钩子中修改组件状态也是安全的。

在服务器端渲染期间不会调用此挂钩。

更新#

在组件由于响应状态更改而更新其 DOM 树后调用。

类型

interface ComponentOptions {

updated?(this: ComponentPublicInstance): void

}

细节

父组件的更新钩子在其子组件之后被调用。

在组件的任何 DOM 更新后调用此钩子，这可能是由不同的状态更改引起的。如果您需要在特定状态更改后访问更新的 DOM，请改用nextTick()。

在服务器端渲染期间不会调用此挂钩。

警告

不要在更新的钩子中改变组件状态 - 这可能会导致无限更新循环！

卸载前#

在要卸载组件实例之前调用。

类型

interface ComponentOptions {

beforeUnmount?(this: ComponentPublicInstance): void

}

细节

调用此钩子时，组件实例仍然可以正常工作。

在服务器端渲染期间不会调用此挂钩。

卸载#

在组件卸载后调用。

类型

interface ComponentOptions {

unmounted?(this: ComponentPublicInstance): void

}

细节

组件在以下情况下被视为已卸载：

它的所有子组件都已卸载。

其所有相关的反应效果（渲染效果和在 期间创建的计算/观察者setup()）都已停止。

使用此钩子清除手动创建的副作用，例如计时器、DOM 事件侦听器或服务器连接。

在服务器端渲染期间不会调用此挂钩。

错误捕获#

当捕获到从后代组件传播的错误时调用。

类型

interface ComponentOptions {

errorCaptured?(

this: ComponentPublicInstance,

err: unknown,

instance: ComponentPublicInstance | null,

info: string

): boolean | void

}

细节

可以从以下来源捕获错误：

组件渲染

事件处理程序

生命周期钩子

setup()功能

观察者

自定义指令钩子

过渡挂钩

钩子接收三个参数：错误、触发错误的组件实例和指定错误源类型的信息字符串。

您可以修改组件状态以errorCaptured()向用户显示错误状态。但是，重要的是错误状态不应呈现导致错误的原始内容；否则组件将被抛出无限渲染循环。

钩子可以返回false以阻止错误进一步传播。请参阅下面的错误传播详细信息。

错误传播规则

默认情况下，app.config.errorHandler如果已定义，所有错误仍会发送到应用程序级别，因此这些错误仍可以在一个地方报告给分析服务。

errorCaptured如果组件的继承链或父链上存在多个钩子，则所有这些钩子都会在相同的错误上被调用，按从下到上的顺序。这类似于原生 DOM 事件的冒泡机制。

如果errorCaptured钩子本身抛出一个错误，这个错误和原始捕获的错误都会被发送到app.config.errorHandler.

errorCaptured挂钩可以返回以false防止错误进一步传播。这实质上是在说“这个错误已经被处理，应该被忽略”。它将防止任何额外的errorCaptured钩子或被app.config.errorHandler此错误调用。

渲染跟踪#

当组件的渲染效果跟踪到反应性依赖项时调用。

类型

interface ComponentOptions {

renderTracked?(this: ComponentPublicInstance, e: DebuggerEvent): void

}

type DebuggerEvent = {

effect: ReactiveEffect

target: object

type: TrackOpTypes /\* 'get' | 'has' | 'iterate' \*/

key: any

}

另请参阅： 深度反应性

渲染触发#

当响应式依赖触发组件的渲染效果重新运行时调用。

类型

interface ComponentOptions {

renderTriggered?(this: ComponentPublicInstance, e: DebuggerEvent): void

}

type DebuggerEvent = {

effect: ReactiveEffect

target: object

type: TriggerOpTypes /\* 'set' | 'add' | 'delete' | 'clear' \*/

key: any

newValue?: any

oldValue?: any

oldTarget?: Map<any, any> | Set<any>

}

另请参阅： 深度反应性

活性#

在组件实例作为由<KeepAlive>.

在服务器端渲染期间不会调用此挂钩。

类型

interface ComponentOptions {

activated?(this: ComponentPublicInstance): void

}

另请参阅： 指南 - 缓存实例的生命周期

停用#

在组件实例作为由<KeepAlive>.

在服务器端渲染期间不会调用此挂钩。

类型

interface ComponentOptions {

deactivated?(this: ComponentPublicInstance): void

}

另请参阅： 指南 - 缓存实例的生命周期

服务器预取#

在组件实例在服务器上呈现之前要解析的异步函数。

类型

interface ComponentOptions {

serverPrefetch?(this: ComponentPublicInstance): Promise<any>

}

细节

如果钩子返回一个 Promise，服务器渲染器将等到 Promise 被解析后再渲染组件。

此钩子仅在服务器端渲染期间调用，可用于执行仅服务器数据获取。

例子

export default {

data() {

return {

data: null

}

},

async serverPrefetch() {

// component is rendered as part of the initial request

// pre-fetch data on server as it is faster than on the client

this.data = await fetchOnServer(/\* ... \*/)

},

async mounted() {

if (!this.data) {

// if data is null on mount, it means the component

// is dynamically rendered on the client. Perform a

// client-side fetch instead.

this.data = await fetchOnClient(/\* ... \*/)

}

}

}

# 第12章 选项API：组合

提供#

提供可以由后代组件注入的值。

类型

interface ComponentOptions {

provide?: object | ((this: ComponentPublicInstance) => object)

}

细节：

provide并且inject一起使用以允许祖先组件充当其所有后代的依赖注入器，无论组件层次结构有多深，只要它们位于同一父链中。

该provide选项应该是一个对象或返回一个对象的函数。此对象包含可用于注入其后代的属性。您可以在此对象中使用符号作为键。

例子

基本用法：

const s = Symbol()

export default {

provide: {

foo: 'foo',

[s]: 'bar'

}

}

使用函数提供每个组件的状态：

export default {

data() {

return {

msg: 'foo'

}

}

provide() {

return {

msg: this.msg

}

}

}

请注意，在上面的示例中，提供的msg将不是反应式的。有关更多详细信息，请参阅使用反应性。

另请参阅： 提供/注入

注入#

通过从祖先提供者中定位属性来声明要注入当前组件的属性。

类型

interface ComponentOptions {

inject?: ArrayInjectOptions | ObjectInjectOptions

}

type ArrayInjectOptions = string[]

type ObjectInjectOptions = {

[key: string | symbol]:

| string

| symbol

| { from?: string | symbol; default?: any }

}

细节

inject选项应该是：

字符串数组，或

一个对象，其中键是本地绑定名称，值是：

在可用注入中搜索的键（字符串或符号），或

一个对象，其中：

该from属性是在可用注入中搜索的键（字符串或符号），并且

该default属性用作后备值。类似于 props 的默认值，对象类型需要一个工厂函数来避免多个组件实例之间的值共享。

undefined如果既没有提供匹配的属性也没有提供默认值，则注入属性。

请注意，注入的绑定不是反应式的。这是故意的。但是，如果注入的值是响应式对象，则该对象上的属性确实保持响应式。有关更多详细信息，请参阅使用反应性。

例子

基本用法：

export default {

inject: ['foo'],

created() {

console.log(this.foo)

}

}

使用注入的值作为道具的默认值：

const Child = {

inject: ['foo'],

props: {

bar: {

default() {

return this.foo

}

}

}

}

使用注入值作为数据输入：

const Child = {

inject: ['foo'],

data() {

return {

bar: this.foo

}

}

}

注入可以是可选的，具有默认值：

const Child = {

inject: {

foo: { default: 'foo' }

}

}

如果需要从具有不同名称的属性注入，请使用from来表示源属性：

const Child = {

inject: {

foo: {

from: 'bar',

default: 'foo'

}

}

}

与 prop 默认值类似，您需要对非原始值使用工厂函数：

const Child = {

inject: {

foo: {

from: 'bar',

default: () => [1, 2, 3]

}

}

}

另请参阅： 提供/注入

混入#

要混合到当前组件中的选项对象数组。

类型

interface ComponentOptions {

mixins?: ComponentOptions[]

}

细节：

该mixins选项接受一个 mixin 对象数组。这些 mixin 对象可以像普通实例对象一样包含实例选项，并且它们将使用特定的选项合并逻辑与最终选项合并。例如，如果你的 mixin 包含一个created钩子并且组件本身也有一个，那么这两个函数都会被调用。

Mixin 钩子按照它们提供的顺序被调用，并且在组件自己的钩子之前被调用。

不再推荐

在 Vue 2 中，mixin 是创建可重用组件逻辑块的主要机制。虽然 Vue 3 继续支持 mixin，但Composition API现在是组件之间代码重用的首选方法。

例子：

const mixin = {

created() {

console.log(1)

}

}

createApp({

created() {

console.log(2)

},

mixins: [mixin]

})

// => 1

// => 2

延伸#

要扩展的“基类”组件。

类型：

interface ComponentOptions {

extends?: ComponentOptions

}

细节：

允许一个组件扩展另一个组件，继承其组件选项。

从实现的角度来看，extends几乎与mixins. 指定的组件extends将被视为第一个 mixin。

但是，extends表达mixins不同的意图。该mixins选项主要用于组合功能块，而extends主要关注继承。

与 一样mixins，任何选项都将使用相关的合并策略进行合并。

例子：

const CompA = { ... }

const CompB = {

extends: CompA,

...

}

# 第13章 选项API：杂项

姓名#

显式声明组件的显示名称。

类型

interface ComponentOptions {

name?: string

}

细节

组件的名称用于以下用途：

组件自身模板中的递归自引用

在 Vue DevTools 的组件检查树中显示

在警告组件跟踪中显示

当您使用单文件组件时，组件已经从文件名中推断出自己的名称。例如，一个名为的文件MyComponent.vue将具有推断的显示名称“MyComponent”。

另一种情况是，当一个组件用 全局注册时app.component，全局 ID 会自动设置为其名称。

该name选项允许您覆盖推断的名称，或在无法推断名称时显式提供名称（例如，当不使用构建工具或内联的非 SFC 组件时）。

有一种情况是明确需要的：通过其propsname与可缓存组件匹配时。<KeepAlive>include / exclude

小费

从 3.2.34 版本开始，单文件组件 using<script setup>将根据文件名自动推断其name选项，即使与<KeepAlive>.

继承属性#

控制是否应启用默认组件属性失败行为。

类型

interface ComponentOptions {

inheritAttrs?: boolean // default: true

}

细节

默认情况下，未被识别为道具的父范围属性绑定将“失败”。这意味着当我们有一个单根组件时，这些绑定将作为普通 HTML 属性应用于子组件的根元素。在创作包装目标元素或另一个组件的组件时，这可能并不总是期望的行为。通过设置inheritAttrs为false，可以禁用此默认行为。这些属性可通过$attrs实例属性获得，并且可以使用 . 显式绑定到非根元素v-bind。

例子

<script>

export default {

inheritAttrs: false,

props: ['label', 'value'],

emits: ['input']

}

</script>

<template>

<label>

{{ label }}

<input

v-bind="$attrs"

v-bind:value="value"

v-on:input="$emit('input', $event.target.value)"

/>

</label>

</template>

另请参阅： Fallthrough 属性

成分#

注册组件以供组件实例使用的对象。

类型

interface ComponentOptions {

components?: { [key: string]: Component }

}

例子

import Foo from './Foo.vue'

import Bar from './Bar.vue'

export default {

components: {

// shorthand

Foo,

// register under a different name

RenamedBar: Bar

}

}

另请参阅： 组件注册

指令#

注册指令以供组件实例使用的对象。

类型

interface ComponentOptions {

directives?: { [key: string]: Directive }

}

例子

export default {

directives: {

// enables v-focus in template

focus: {

mounted(el) {

el.focus()

}

}

}

}

<input v-focus>

可用于组件实例的指令散列。

# 第14章 选项API ：组件实例

$数据#

从data选项返回的对象，由组件做出反应。组件实例代理对其数据对象属性的访问。

类型

interface ComponentPublicInstance {

$data: object

}

$道具#

表示组件当前的已解析道具的对象。

类型

interface ComponentPublicInstance {

$props: object

}

细节

props仅包含通过选项声明的道具。组件实例代理访问其 props 对象的属性。

$el#

组件实例正在管理的根 DOM 节点。

类型

interface ComponentPublicInstance {

$el: Node | undefined

}

细节

$elundefined直到组件被安装。

对于具有单个根元素的组件，$el将指向该元素。

对于带有文本根的组件，$el将指向文本节点。

对于具有多个根节点的组件，$el将是 Vue 用来跟踪组件在 DOM 中的位置的占位符 DOM 节点（文本节点，或 SSR 水合模式下的注释节点）。

小费

为了一致性，建议使用模板引用来直接访问元素，而不是依赖$el.

$选项#

用于实例化当前组件实例的已解析组件选项。

类型

interface ComponentPublicInstance {

$options: ComponentOptions

}

细节

该$options对象公开了当前组件的已解析选项，并且是这些可能来源的合并结果：

全局混合

组件extends基础

组件混合

它通常用于支持自定义组件选项：

const app = createApp({

customOption: 'foo',

created() {

console.log(this.$options.customOption) // => 'foo'

}

})

也可以看看： app.config.optionMergeStrategies

$父母#

父实例，如果当前实例有一个。它将null用于根实例本身。

类型

interface ComponentPublicInstance {

$parent: ComponentPublicInstance | null

}

$根#

当前组件树的根组件实例。如果当前实例没有父母，则该值将是它自己。

类型

interface ComponentPublicInstance {

$root: ComponentPublicInstance

}

$插槽#

表示父组件传递的插槽的对象。

类型

interface ComponentPublicInstance {

$slots: { [name: string]: Slot }

}

type Slot = (...args: any[]) => VNode[]

细节

通常在手动创作渲染函数时使用，但也可用于检测插槽是否存在。

每个插槽都this.$slots作为一个函数公开，该函数在对应于该插槽名称的键下返回一个 vnode 数组。默认插槽显示为this.$slots.default.

如果插槽是作用域插槽，则传递给插槽函数的参数可作为插槽道具使用。

另请参阅： 渲染函数 - 渲染槽

$refs#

通过模板 refs注册的 DOM 元素和组件实例的对象。

类型

interface ComponentPublicInstance {

$refs: { [name: string]: Element | ComponentPublicInstance | null }

}

也可以看看：

模板参考

特殊属性 - 参考

$attrs#

包含组件的 fallthrough 属性的对象。

类型

interface ComponentPublicInstance {

$attrs: object

}

细节

Fallthrough Attributes是由父组件传递的属性和事件处理程序，但未声明为 prop 或子组件发出的事件。

默认情况下，$attrs如果只有一个根元素，则在组件的根元素上自动继承其中的所有内容。如果组件有多个根节点，此行为将被禁用，并且可以使用该inheritAttrs选项显式禁用。

也可以看看：

Fallthrough 属性

$手表()#

用于创建观察者的命令式 API。

类型

interface ComponentPublicInstance {

$watch(

source: string | (() => any),

callback: WatchCallback,

options?: WatchOptions

): StopHandle

}

type WatchCallback<T> = (

value: T,

oldValue: T,

onCleanup: (cleanupFn: () => void) => void

) => void

interface WatchOptions {

immediate?: boolean // default: false

deep?: boolean // default: false

flush?: 'pre' | 'post' | 'sync' // default: 'pre'

onTrack?: (event: DebuggerEvent) => void

onTrigger?: (event: DebuggerEvent) => void

}

type StopHandle = () => void

细节

第一个参数是监视源。它可以是组件属性名称字符串、简单的点分隔路径字符串或 getter 函数。

第二个参数是回调函数。回调接收监视源的新值和旧值。

immediate：在观察者创建时立即触发回调。旧值将undefined在第一次调用时出现。

deep：如果源是对象，则强制深度遍历源，以便回调触发深度突变。请参阅深度观察者。

flush：调整回调的刷新时间。请参阅回调刷新时间。

onTrack / onTrigger: 调试观察者的依赖。请参阅观察程序调试。

例子

观察一个属性名称：

this.$watch('a', (newVal, oldVal) => {})

观察一个点分隔的路径：

this.$watch('a.b', (newVal, oldVal) => {})

对更复杂的表达式使用 getter：

this.$watch(

// every time the expression `this.a + this.b` yields

// a different result, the handler will be called.

// It's as if we were watching a computed property

// without defining the computed property itself.

() => this.a + this.b,

(newVal, oldVal) => {}

)

停止观察者：

const unwatch = this.$watch('a', cb)

// later...

unwatch()

也可以看看：

选项 -watch

指南 - 观察者

$发射（）#

在当前实例上触发自定义事件。任何其他参数都将传递到侦听器的回调函数中。

类型

interface ComponentPublicInstance {

$emit(event: string, ...args: any[]): void

}

例子

export default {

created() {

// only event

this.$emit('foo')

// with additional arguments

this.$emit('bar', 1, 2, 3)

}

}

也可以看看：

组件 - 事件

emits选项

$forceUpdate()#

强制组件实例重新渲染。

类型

interface ComponentPublicInstance {

$forceUpdate(): void

}

细节

鉴于 Vue 的全自动反应系统，这应该很少需要。您可能需要它的唯一情况是您使用高级反应性 API 显式创建了非反应性组件状态。

$nextTick()#

全局的实例绑定版本nextTick()。

类型

interface ComponentPublicInstance {

$nextTick(callback?: (this: ComponentPublicInstance) => void): Promise<void>

}

细节

与全局版本的唯一区别nextTick()是传递给的回调this.$nextTick()将其this上下文绑定到当前组件实例。

也可以看看： nextTick()

# 第15章 内置插件：指令

文本#

更新元素的文本内容。

期望： string

细节

v-text通过设置元素的textContent属性来工作，因此它将覆盖元素内的任何现有内容。如果你需要更新 的部分textContent，你应该使用mustache 插值代替。

例子

<span v-text="msg"></span>

<!-- same as -->

<span>{{msg}}</span>

另请参阅： 模板语法 - 文本插值

v-html#

更新元素的innerHTML。

期望： string

细节：

的内容v-html作为纯 HTML 插入 - 不会处理 Vue 模板语法。如果您发现自己尝试使用 编写模板v-html，请尝试改用组件来重新考虑解决方案。

安全说明

在您的网站上动态呈现任意 HTML 可能非常危险，因为它很容易导致XSS 攻击。仅用于v-html受信任的内容，切勿用于用户提供的内容。

在Single-File Components中，scoped样式不会应用于里面的内容v-html，因为 Vue 的模板编译器不会处理 HTML。如果您想v-html使用作用域 CSS 来定位内容，则可以改用CSS 模块<style>或具有手动作用域策略（如 BEM）的附加全局元素。

例子：

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

另请参阅： 模板语法 - 原始 HTML

秀#

根据表达式值的真实性切换元素的可见性。

期望： any

细节

v-show通过内联样式设置CSS 属性来工作，并在元素可见时display尝试尊重初始值。display当其条件发生变化时，它也会触发转换。

另请参阅： 条件渲染 - v-show

v-if#

根据表达式值的真实性有条件地渲染元素或模板片段。

期望： any

细节

当一个v-if元素被切换时，该元素及其包含的指令/组件被销毁并重新构建。如果初始条件为假，则根本不会呈现内部内容。

可用于<template>表示仅包含文本或多个元素的条件块。

该指令在其条件发生变化时触发转换。

一起使用时，v-if优先级高于v-for. 我们不建议在一个元素上同时使用这两个指令——有关详细信息，请参阅列表渲染指南。

另请参阅： 条件渲染 - v-if

v-else#

v-if表示/链的“其他块” v-if。v-else-if

不期待表情

细节

限制：前一个兄弟元素必须有v-ifor v-else-if。

可用于<template>表示仅包含文本或多个元素的条件块。

例子

<div v-if="Math.random() > 0.5">

Now you see me

</div>

<div v-else>

Now you don't

</div>

另请参阅： 条件渲染 - v-else

v-else-if#

为 .表示“else if 块” v-if。可以上链。

期望： any

细节

限制：前一个兄弟元素必须有v-ifor v-else-if。

可用于<template>表示仅包含文本或多个元素的条件块。

例子

<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>

另请参阅： 条件渲染 - v-else-if

v-for#

根据源数据多次渲染元素或模板块。

期望： Array | Object | number | string | Iterable

细节

指令的值必须使用特殊语法alias in expression为当前迭代的元素提供别名：

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

{{ item.text }}

</div>

或者，您还可以为索引指定别名（如果在对象上使用，则为键）：

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

<div v-for="(value, key) in object"></div>

<div v-for="(value, name, index) in object"></div>

的默认行为v-for将尝试在不移动元素的情况下就地修补元素。要强制它重新排序元素，您应该提供一个带有key特殊属性的排序提示：

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

{{ item.text }}

</div>

v-for也可以处理实现Iterable Protocol的值，包括 nativeMap和Set.

也可以看看：

列表渲染

开启#

将事件侦听器附加到元素。

速记： @

期望： Function | Inline Statement | Object (without argument)

参数：（ event如果使用 Object 语法，则可选）

修饰符：

.stop- 打电话event.stopPropagation()。

.prevent- 打电话event.preventDefault()。

.capture- 在捕获模式下添加事件监听器。

.self- 仅在从该元素分派事件时触发处理程序。

.{keyAlias}- 仅在某些键上触发处理程序。

.once- 最多触发一次处理程序。

.left- 仅触发鼠标左键事件的处理程序。

.right- 仅触发鼠标右键事件的处理程序。

.middle- 仅触发中键鼠标事件的处理程序。

.passive- 附加一个 DOM 事件{ passive: true }。

细节

事件类型由参数表示。表达式可以是方法名称、内联语句，如果存在修饰符，则可以省略。

在普通元素上使用时，它仅侦听本机 DOM 事件。在自定义元素组件上使用时，它会侦听在该子组件上发出的自定义事件。

在侦听本机 DOM 事件时，该方法接收本机事件作为唯一参数。如果使用内联语句，则该语句可以访问特殊$event属性：v-on:click="handle('ok', $event)".

v-on还支持绑定到不带参数的事件/侦听器对的对象。请注意，使用对象语法时，它不支持任何修饰符。

例子：

<!-- method handler -->

<button v-on:click="doThis"></button>

<!-- dynamic event -->

<button v-on:[event]="doThis"></button>

<!-- inline statement -->

<button v-on:click="doThat('hello', $event)"></button>

<!-- shorthand -->

<button @click="doThis"></button>

<!-- shorthand dynamic event -->

<button @[event]="doThis"></button>

<!-- stop propagation -->

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

<!-- prevent default -->

<button @click.prevent="doThis"></button>

<!-- prevent default without expression -->

<form @submit.prevent></form>

<!-- chain modifiers -->

<button @click.stop.prevent="doThis"></button>

<!-- key modifier using keyAlias -->

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

<!-- the click event will be triggered at most once -->

<button v-on:click.once="doThis"></button>

<!-- object syntax -->

<button v-on="{ mousedown: doThis, mouseup: doThat }"></button>

监听子组件上的自定义事件（在子组件上发出“my-event”时调用处理程序）：

<MyComponent @my-event="handleThis" />

<!-- inline statement -->

<MyComponent @my-event="handleThis(123, $event)" />

也可以看看：

事件处理

组件 - 自定义事件

v-绑定#

将一个或多个属性或组件属性动态绑定到表达式。

速记： :或.（使用.prop修饰符时）

期望： any (with argument) | Object (without argument)

争论： attrOrProp (optional)

修饰符：

.camel- 将 kebab-case 属性名称转换为 camelCase。

.prop- 强制将绑定设置为 DOM 属性。3.2+

.attr- 强制将绑定设置为 DOM 属性。3.2+

用法：

用于绑定classorstyle属性时，v-bind支持额外的值类型，例如 Array 或 Objects。有关更多详细信息，请参阅下面的链接指南部分。

在元素上设置绑定时，Vue 默认使用in运算符检查检查元素是否具有定义为属性的键。如果定义了属性，Vue 会将值设置为 DOM 属性而不是属性。这在大多数情况下应该有效，但您可以通过显式使用.propor.attr修饰符来覆盖此行为。这有时是必要的，尤其是在使用自定义元素时。

用于组件 prop 绑定时，必须在子组件中正确声明该 prop。

不带参数使用时，可用于绑定包含属性名称-值对的对象。

例子：

<!-- bind an attribute -->

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

<!-- dynamic attribute name -->

<button v-bind:[key]="value"></button>

<!-- shorthand -->

<img :src="imageSrc" />

<!-- shorthand dynamic attribute name -->

<button :[key]="value"></button>

<!-- with inline string concatenation -->

<img :src="'/path/to/images/' + fileName" />

<!-- class binding -->

<div :class="{ red: isRed }"></div>

<div :class="[classA, classB]"></div>

<div :class="[classA, { classB: isB, classC: isC }]"></div>

<!-- style binding -->

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

<div :style="[styleObjectA, styleObjectB]"></div>

<!-- binding an object of attributes -->

<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>

<!-- prop binding. "prop" must be declared in the child component. -->

<MyComponent :prop="someThing" />

<!-- pass down parent props in common with a child component -->

<MyComponent v-bind="$props" />

<!-- XLink -->

<svg><a :xlink:special="foo"></a></svg>

.prop修饰符还有一个专用的速记.：

<div :someProperty.prop="someObject"></div>

<!-- equivalent to -->

<div .someProperty="someObject"></div>

.camel修饰符允许在使用 DOM 内模板时将属性v-bind名称骆驼化，例如 SVGviewBox属性：

<svg :view-box.camel="viewBox"></svg>

.camel如果您使用字符串模板，或者使用构建步骤预编译模板，则不需要。

也可以看看：

类和样式绑定

组件 - 道具传递细节

v型#

在表单输入元素或组件上创建双向绑定。

预期：根据表单输入元素的值或组件的输出而变化

仅限于：

<input>

<select>

<textarea>

成分

修饰符：

.lazy- 听change事件而不是input

.number- 将有效的输入字符串转换为数字

.trim- 修剪输入

也可以看看：

表单输入绑定

组件事件 - 与v-model

V槽#

表示期望接收道具的命名槽或作用域槽。

速记： #

Expects：在函数参数位置有效的 JavaScript 表达式，包括对解构的支持。可选 - 仅在期望将道具传递到插槽时才需要。

参数：插槽名称（可选，默认为default）

仅限于：

<template>

组件（用于带有道具的唯一默认插槽）

例子：

<!-- Named slots -->

<BaseLayout>

<template v-slot:header>

Header content

</template>

<template v-slot:default>

Default slot content

</template>

<template v-slot:footer>

Footer content

</template>

</BaseLayout>

<!-- Named slot that receives props -->

<InfiniteScroll>

<template v-slot:item="slotProps">

<div class="item">

{{ slotProps.item.text }}

</div>

</template>

</InfiniteScroll>

<!-- Default slot that receive props, with destructuring -->

<Mouse v-slot="{ x, y }">

Mouse position: {{ x }}, {{ y }}

</Mouse>

也可以看看：

组件 - 插槽

v-pre#

跳过此元素及其所有子元素的编译。

不期待表情

细节

在带有 的元素内v-pre，所有 Vue 模板语法都将被保留并按原样呈现。最常见的用例是显示原始胡须标签。

例子：

<span v-pre>{{ this will not be compiled }}</span>

v-一次#

仅渲染元素和组件一次，并跳过以后的更新。

不期待表情

细节

在随后的重新渲染中，元素/组件及其所有子元素将被视为静态内容并被跳过。这可用于优化更新性能。

<!-- single element -->

<span v-once>This will never change: {{msg}}</span>

<!-- the element have children -->

<div v-once>

<h1>comment</h1>

<p>{{msg}}</p>

</div>

<!-- component -->

<MyComponent v-once :comment="msg"></MyComponent>

<!-- `v-for` directive -->

<ul>

<li v-for="i in list" v-once>{{i}}</li>

</ul>

从 3.2 开始，您还可以使用v-memo.

也可以看看：

数据绑定语法 - 插值

备忘录

备忘录#

期望： any[]

细节

记忆模板的子树。可用于元素和组件。该指令需要一个固定长度的依赖值数组来比较记忆。如果数组中的每个值都与上次渲染相同，则将跳过整个子树的更新。例如：

<div v-memo="[valueA, valueB]">

...

</div>

当组件重新渲染时，如果两者valueA和保持不变，则此组件及其子组件valueB的所有更新都将被跳过。<div>事实上，即使是 Virtual DOM VNode 创建也将被跳过，因为子树的记忆副本可以被重用。

正确指定记忆数组很重要，否则我们可能会跳过确实应该应用的更新。v-memo使用空依赖数组 ( v-memo="[]") 在功能上等同于v-once.

用法与v-for

v-memo仅用于性能关键场景中的微优化，应该很少需要。这可能证明有用的最常见情况是在渲染大型v-for列表时（其中length > 1000）：

<div v-for="item in list" :key="item.id" v-memo="[item.id === selected]">

<p>ID: {{ item.id }} - selected: {{ item.id === selected }}</p>

<p>...more child nodes</p>

</div>

当组件的selected状态发生变化时，即使大部分项目保持不变，也会创建大量的 VNode。这里的v-memo用法本质上是说“仅当它从未选中变为选中时才更新此项目，或者相反”。这允许每个未受影响的项目重用其先前的 VNode 并完全跳过差异。请注意，我们不需要item.id在此处包含在 memo 依赖数组中，因为 Vue 会自动从项目的:key.

警告

使用v-memowith 时v-for，请确保它们用于同一元素。v-memo里面不行v-for。

v-memo也可以在组件上使用，以在子组件更新检查已被取消优化的某些边缘情况下手动防止不需要的更新。但同样，开发人员有责任指定正确的依赖数组以避免跳过必要的更新。

也可以看看：

v-一次

v-斗篷#

用于隐藏未编译的模板，直到它准备好。

不期待表情

细节

此指令仅在无构建步骤设置中需要。

当使用 in-DOM 模板时，可能会出现“未编译模板的闪光”：用户可能会看到原始的 mustache 标签，直到安装的组件用渲染的内容替换它们。

v-cloak将保留在元素上，直到关联的组件实例被挂载。结合 CSS 规则如[v-cloak] { display: none }，它可以用来隐藏原始模板，直到组件准备好。

例子：

[v-cloak] {

display: none;

}

<div v-cloak>

{{ message }}

</div>

在<div>编译完成之前将不可见。

# 第16章 内置插件：成分

注册和使用

内置组件可以直接在模板中使用，无需注册。它们也是可摇树的：它们仅在使用时才包含在构建中。

在渲染函数中使用它们时，需要显式导入它们。例如：

import { h, Transition } from 'vue'

h(Transition, {

/\* props \*/

})

<Transition>#

为单个元素或组件提供动画过渡效果。

道具

interface TransitionProps {

/\*\*

\* Used to automatically generate transition CSS class names.

\* e.g. `name: 'fade'` will auto expand to `.fade-enter`,

\* `.fade-enter-active`, etc.

\*/

name?: string

/\*\*

\* Whether to apply CSS transition classes.

\* Default: true

\*/

css?: boolean

/\*\*

\* Specifies the type of transition events to wait for to

\* determine transition end timing.

\* Default behavior is auto detecting the type that has

\* longer duration.

\*/

type?: 'transition' | 'animation'

/\*\*

\* Specifies explicit durations of the transition.

\* Default behavior is wait for the first `transitionend`

\* or `animationend` event on the root transition element.

\*/

duration?: number | { enter: number; leave: number }

/\*\*

\* Controls the timing sequence of leaving/entering transitions.

\* Default behavior is simultaneous.

\*/

mode?: 'in-out' | 'out-in' | 'default'

/\*\*

\* Whether to apply transition on initial render.

\* Default: false

\*/

appear?: boolean

/\*\*

\* Props for customizing transition classes.

\* Use kebab-case in templates, e.g. enter-from-class="xxx"

\*/

enterFromClass?: string

enterActiveClass?: string

enterToClass?: string

appearFromClass?: string

appearActiveClass?: string

appearToClass?: string

leaveFromClass?: string

leaveActiveClass?: string

leaveToClass?: string

}

活动

@before-enter

@before-leave

@enter

@leave

@appear

@after-enter

@after-leave

@after-appear

@enter-cancelled

@leave-cancelled（v-show仅限）

@appear-cancelled

例子

简单元素：

<Transition>

<div v-if="ok">toggled content</div>

</Transition>

动态组件，出现过渡模式+动画：

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

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

</Transition>

监听转换事件：

<Transition @after-enter="onTransitionComplete">

<div v-show="ok">toggled content</div>

</Transition>

另请参阅： <Transition>指南

<TransitionGroup>#

为列表中的多个元素或组件提供过渡效果。

道具

<TransitionGroup><Transition>接受与except相同的道具mode，加上两个额外的道具：

interface TransitionGroupProps extends Omit<TransitionProps, 'mode'> {

/\*\*

\* If not defined, renders as a fragment.

\*/

tag?: string

/\*\*

\* For customizing the CSS class applied during move transitions.

\* Use kebab-case in templates, e.g. move-class="xxx"

\*/

moveClass?: string

}

活动

<TransitionGroup>发出与 相同的事件<Transition>。

细节

默认情况下，<TransitionGroup>不渲染包装 DOM 元素，但可以通过tagprop 定义。

请注意，a 中的每个孩子都<transition-group>必须具有唯一的键控才能使动画正常工作。

<TransitionGroup>支持通过 CSS 变换移动过渡。当更新后孩子在屏幕上的位置发生变化时，它将应用一个移动的 CSS 类（从name属性自动生成或使用move-class道具配置）。如果在应用移动类时 CSStransform属性是“可转换的”，则元素将使用FLIP 技术平滑地动画到其目的地。

例子

<TransitionGroup tag="ul" name="slide">

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

{{ item.text }}

</li>

</TransitionGroup>

另请参阅： 指南 - TransitionGroup

<KeepAlive>#

缓存包裹在里面的动态切换组件。

道具

interface KeepAliveProps {

/\*\*

\* If specified, only components with names matched by

\* `include` will be cached.

\*/

include?: MatchPattern

/\*\*

\* Any component with a name matched by `exclude` will

\* not be cached.

\*/

exclude?: MatchPattern

/\*\*

\* The maximum number of component instances to cache.

\*/

max?: number | string

}

type MatchPattern = string | RegExp | (string | RegExp)[]

细节

当包裹在动态组件上时，<KeepAlive>缓存不活动的组件实例而不破坏它们。

任何时候都只能有一个活动组件实例作为其直接子代<KeepAlive>。

当一个组件在内部切换时<KeepAlive>，它的activated和deactivated生命周期钩子将被相应地调用，提供一个不被调用的mountedand的替代方案unmounted。这适用<KeepAlive>于其直系子代及其所有后代。

例子

基本用法：

<KeepAlive>

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

</KeepAlive>

与v-if/v-else分支一起使用时，一次只能渲染一个组件：

<KeepAlive>

<comp-a v-if="a > 1"></comp-a>

<comp-b v-else></comp-b>

</KeepAlive>

与 一起使用<Transition>：

<Transition>

<KeepAlive>

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

</KeepAlive>

</Transition>

使用include/ exclude：

<!-- comma-delimited string -->

<KeepAlive include="a,b">

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

</KeepAlive>

<!-- regex (use `v-bind`) -->

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

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

</KeepAlive>

<!-- Array (use `v-bind`) -->

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

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

</KeepAlive>

用法与max：

<KeepAlive :max="10">

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

</KeepAlive>

另请参阅： 指南 - KeepAlive

<Teleport>#

将其插槽内容呈现到 DOM 的另一部分。

道具

interface TeleportProps {

/\*\*

\* Required. Specify target container.

\* Can either be a selector or an actual element.

\*/

to: string | HTMLElement

/\*\*

\* When `true`, the content will remain in its original

\* location instead of moved into the target container.

\* Can be changed dynamically.

\*/

disabled?: boolean

}

例子

指定目标容器：

<teleport to="#some-id" />

<teleport to=".some-class" />

<teleport to="[data-teleport]" />

有条件地禁用：

<teleport to="#popup" :disabled="displayVideoInline">

<video src="./my-movie.mp4">

</teleport>

另请参阅： 指南 - 传送

<Suspense> #

用于编排组件树中的嵌套异步依赖项。

道具

interface SuspenseProps {

timeout?: string | number

}

活动

@resolve

@pending

@fallback

细节

<Suspense>接受两个插槽：#default插槽和#fallback插槽。它将在渲染内存中的默认插槽时显示后备插槽的内容。

如果它在渲染默认槽时遇到异步依赖项（Async Components和 components with async setup()），它将等到所有这些都解决后再显示默认槽。

# 第17章 内置插件：特殊元素

不是组件

<component>并且<slot>是类似组件的功能和模板语法的一部分。它们不是真正的组件，并且在模板编译期间被编译掉。因此，它们通常在模板中用小写字母编写。

<component>#

用于渲染动态组件或元素的“元组件”。

道具

interface DynamicComponentProps {

is: string | Component

}

细节

要渲染的实际组件由isprop 确定。

Whenis是一个字符串，它可以是 HTML 标记名称或组件的注册名称。

或者，is也可以直接绑定到一个组件的定义。

例子

按注册名称渲染组件（选项 API）：

<script>

import Foo from './Foo.vue'

import Bar from './Bar.vue'

export default {

components: { Foo, Bar },

data() {

return {

view: 'Foo'

}

}

}

</script>

<template>

<component :is="view" />

</template>

按定义渲染组件（Composition API with <script setup>）：

<script setup>

import Foo from './Foo.vue'

import Bar from './Bar.vue'

</script>

<template>

<component :is="Math.random() > 0.5 ? Foo : Bar" />

</template>

渲染 HTML 元素：

<component :is="href ? 'a' : 'span'"></component>

内置组件都可以传递给is，但是如果要通过名称传递，则必须注册它们。例如：

<script>

import { Transition, TransitionGroup } from 'vue'

export default {

components: {

Transition,

TransitionGroup

}

}

</script>

<template>

<component :is="isGroup ? 'TransitionGroup' : 'Transition'">

...

</component>

</template>

is如果您将组件本身传递给而不是其名称，则不需要注册，例如在<script setup>.

如果v-model在<component>标签上使用，模板编译器会将其扩展为modelValue道具和update:modelValue事件侦听器，就像任何其他组件一样。但是，这将与原生 HTML 元素不兼容，例如<input>或<select>。因此，v-model与动态创建的本机元素一起使用将不起作用：

<script setup>

import { ref } from 'vue'

const tag = ref('input')

const username = ref('')

</script>

<template>

<!-- This won't work as 'input' is a native HTML element -->

<component :is="tag" v-model="username" />

</template>

实际上，这种极端情况并不常见，因为本机表单字段通常包装在实际应用程序的组件中。如果您确实需要直接使用本机元素，则可以v-model手动将其拆分为属性和事件。

另请参阅： 动态组件

<slot>#

表示模板中的插槽内容出口。

道具

interface SlotProps {

/\*\*

\* Any props passed to <slot> to passed as arguments

\* for scoped slots

\*/

[key: string]: any

/\*\*

\* Reserved for specifying slot name.

\*/

name?: string

}

细节

该<slot>元素可以使用该name属性来指定插槽名称。指定no 时name，它将呈现默认插槽。传递给 slot 元素的附加属性将作为 slot props 传递给父级中定义的作用域插槽。

元素本身将被其匹配的插槽内容替换。

<slot>Vue 模板中的元素被编译成 JavaScript，所以不要与原生<slot>元素混淆。

# 第18章 内置插件：特殊属性

钥匙#

key特殊属性主要用作 Vue 的虚拟 DOM 算法在将新节点列表与旧列表进行比较时识别 vnode 的提示。

期望： number | string | symbol

细节

在没有键的情况下，Vue 使用一种算法来最小化元素移动，并尝试尽可能多地就地修补/重用相同类型的元素。使用键，它将根据键的顺序更改对元素进行重新排序，并且不再存在键的元素将始终被删除/销毁。

同一个共同父母的孩子必须有唯一的钥匙。重复的键会导致渲染错误。

最常见的用例与v-for：

<ul>

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

</ul>

它还可以用于强制替换元素/组件而不是重用它。当您想要：

正确触发组件的生命周期钩子

触发转换

例如：

<transition>

<span :key="text">{{ text }}</span>

</transition>

text更改时，总是<span>会替换而不是修补，因此将触发转换。

另请参阅： 指南 - 列表渲染 - 维护状态key

参考#

表示模板 ref。

期望： string | Function

细节

ref用于注册对元素或子组件的引用。

在 Options API 中，引用将注册在组件的this.$refs对象下：

<!-- stored as this.$refs.p -->

<p ref="p">hello</p>

在 Composition API 中，引用将存储在具有匹配名称的 ref 中：

<script setup>

import { ref } from 'vue'

const p = ref()

</script>

<template>

<p ref="p">hello</p>

</template>

如果在普通 DOM 元素上使用，则引用将是该元素；如果在子组件上使用，则引用将是子组件实例。

或者ref，可以接受一个函数值，该函数值可以完全控制引用的存储位置：

<ChildComponent :ref="(el) => child = el" />

关于 ref 注册时间的重要说明：因为 ref 本身是作为 render 函数的结果创建的，所以您必须等到组件安装后才能访问它们。

this.$refs也是非反应性的，因此您不应尝试在模板中使用它进行数据绑定。

另请参阅： 模板参考

是#

用于绑定动态组件。

期望： string | Component

在原生元素上的使用 3.1+

当在is原生 HTML 元素上使用该属性时，它将被解释为自定义的内置元素，这是一个原生 Web 平台功能。

但是，有一个用例，您可能需要 Vue 用 Vue 组件替换本机元素，如DOM 模板解析警告中所述。您可以在is属性的值前面加上前缀，vue:以便 Vue 将元素呈现为 Vue 组件：

<table>

<tr is="vue:my-row-component"></tr>

</table>

也可以看看：

内置特殊元件 -<component>

动态组件

# 第19章 单文件组件：SFC 语法规范

概述#

Vue 单文件组件 (SFC)，通常使用\*.vue文件扩展名，是一种自定义文件格式，它使用类似 HTML 的语法来描述 Vue 组件。Vue SFC 在语法上与 HTML 兼容。

每个\*.vue文件包含三种类型的顶级语言块：<template>、<script>和<style>，以及可选的附加自定义块：

<template>

<div class="example">{{ msg }}</div>

</template>

<script>

export default {

data() {

return {

msg: 'Hello world!'

}

}

}

</script>

<style>

.example {

color: red;

}

</style>

<custom1>

This could be e.g. documentation for the component.

</custom1>

语言块#

<template>#

每个\*.vue文件一次最多可以包含一个顶级<template>块。

内容将被提取并传递给@vue/compiler-dom，预编译为 JavaScript 渲染函数，并作为render选项附加到导出的组件。

<script>#

每个\*.vue文件一次最多可以包含一个<script>块（不包括<script setup>）。

该脚本作为 ES 模块执行。

默认导出应该是一个 Vue 组件选项对象，可以是普通对象，也可以是 defineComponent 的返回值。

<script setup>#

每个\*.vue文件一次最多可以包含一个<script setup>块（不包括 normal <script>）。

该脚本被预处理并用作组件的setup()函数，这意味着它将为组件的每个实例执行。中的顶级绑定<script setup>会自动暴露给模板。有关更多详细信息，请参阅<script setup>.

<style>#

一个\*.vue文件可以包含多个<style>标签。

<style>标记可以具有或scoped属性module（有关详细信息，请参阅SFC 样式特征）以帮助将样式封装到当前组件。<style>在同一个组件中可以混合使用不同封装方式的多个标签。

自定义块#

其他自定义块可以包含在\*.vue文件中以满足任何项目特定的需求，例如<docs>块。自定义块的一些真实示例包括：

格子：<page-query>

vite-plugin-vue-gql：<gql>

Vue-i18n：<i18n>

自定义块的处理将取决于工具 - 如果您想构建自己的自定义块集成，请参阅相关工具部分了解更多详细信息。

自动名称推断#

在以下情况下，SFC 会自动从其文件名推断组件的名称：

开发警告格式

开发工具检查

递归自引用。例如，一个名为的文件可以在其模板FooBar.vue中引用它自己。<FooBar/>这比显式注册/导入的组件具有更低的优先级。

预处理器#

lang块可以使用该属性声明预处理器语言。最常见的情况是使用 TypeScript 作为<script>块：

<script lang="ts">

// use TypeScript

</script>

lang可以应用于任何块 - 例如我们可以<style>与SASS和Pug<template>一起使用：

<template lang="pug">

p {{ msg }}

</template>

<style lang="scss">

$primary-color: #333;

body {

color: $primary-color;

}

</style>

请注意，与各种预处理器的集成可能因工具链而异。查看相应的文档以获取示例：

维特

Vue CLI

webpack + vue-loader

来源进口#

如果您更喜欢将\*.vue组件拆分为多个文件，则可以使用该src属性为语言块导入外部文件：

<template src="./template.html"></template>

<style src="./style.css"></style>

<script src="./script.js"></script>

注意src导入遵循与 webpack 模块请求相同的路径解析规则，这意味着：

相对路径需要以./

您可以从 npm 依赖项中导入资源：

<!-- import a file from the installed "todomvc-app-css" npm package -->

<style src="todomvc-app-css/index.css" />

src导入也适用于自定义块，例如：

<unit-test src="./unit-test.js">

</unit-test>

注释#

在每个块中，您应使用所使用语言的注释语法（HTML、CSS、JavaScript、Pug 等）。对于顶级注释，使用 HTML 注释语法：<!-- comment contents here -->

# 第20章 单文件组件：<脚本设置>

<script setup>是一种编译时语法糖，用于在单文件组件 (SFC) 中使用组合 API。如果您同时使用 SFC 和 Composition API，建议使用该语法。<script>与普通语法相比，它提供了许多优点：

更简洁的代码，更少的样板

能够使用纯 TypeScript 声明道具和发出的事件

更好的运行时性能（模板编译成同范围内的渲染函数，无需中间代理）

更好的 IDE 类型推断性能（语言服务器从代码中提取类型的工作更少）

基本语法#

要选择加入语法，请将setup属性添加到<script>块中：

<script setup>

console.log('hello script setup')

</script>

里面的代码被编译为组件setup()函数的内容。这意味着与通常<script>仅在首次导入组件时执行一次不同，内部代码<script setup>将在每次创建组件实例时执行。

顶级绑定暴露给模板#

在 using 时<script setup>，任何在内部声明的顶级绑定（包括变量、函数声明和导入）<script setup>都可以直接在模板中使用：

<script setup>

// variable

const msg = 'Hello!'

// functions

function log() {

console.log(msg)

}

</script>

<template>

<button @click="log">{{ msg }}</button>

</template>

进口以同样的方式暴露。这意味着您可以直接在模板表达式中使用导入的辅助函数，而无需通过methods选项公开它：

<script setup>

import { capitalize } from './helpers'

</script>

<template>

<div>{{ capitalize('hello') }}</div>

</template>

反应性#

需要使用Reactivity API显式创建反应状态。类似于从setup()函数返回的值，引用在模板中引用时会自动展开：

<script setup>

import { ref } from 'vue'

const count = ref(0)

</script>

<template>

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

</template>

使用组件#

范围内的<script setup>值也可以直接用作自定义组件标签名称：

<script setup>

import MyComponent from './MyComponent.vue'

</script>

<template>

<MyComponent />

</template>

将MyComponent其视为被引用为变量。如果你使用过 JSX，这里的心智模型是相似的。kebab-case 等效项<my-component>也适用于模板 - 但是强烈建议使用 PascalCase 组件标签以保持一致性。它还有助于区分原生自定义元素。

动态组件#

由于组件被引用为变量而不是在字符串键下注册，因此:is在内部使用动态组件时应该使用动态绑定<script setup>：

<script setup>

import Foo from './Foo.vue'

import Bar from './Bar.vue'

</script>

<template>

<component :is="Foo" />

<component :is="someCondition ? Foo : Bar" />

</template>

请注意如何将组件用作三元表达式中的变量。

递归组件#

SFC 可以通过其文件名隐式引用自身。例如，一个名为的文件可以在其模板FooBar.vue中引用它自己。<FooBar/>

请注意，它的优先级低于导入的组件。如果您的命名导入与组件的推断名称冲突，您可以为导入设置别名：

import { FooBar as FooBarChild } from './components'

命名空间组件#

您可以使用带有点的组件标签<Foo.Bar>来引用嵌套在对象属性下的组件。当您从单个文件导入多个组件时，这很有用：

<script setup>

import \* as Form from './form-components'

</script>

<template>

<Form.Input>

<Form.Label>label</Form.Label>

</Form.Input>

</template>

使用自定义指令#

全局注册的自定义指令正常工作。本地自定义指令不需要显式注册<script setup>，但它们必须遵循命名方案vNameOfDirective：

<script setup>

const vMyDirective = {

beforeMount: (el) => {

// do something with the element

}

}

</script>

<template>

<h1 v-my-directive>This is a Heading</h1>

</template>

如果您从其他地方导入指令，可以将其重命名以适应所需的命名方案：

<script setup>

import { myDirective as vMyDirective } from './MyDirective.js'

</script>

defineProps() & defineEmits()#

要声明具有完整类型推断支持的选项props，emits我们可以使用defineProps和defineEmitsAPI，它们在内部自动可用<script setup>：

<script setup>

const props = defineProps({

foo: String

})

const emit = defineEmits(['change', 'delete'])

// setup code

</script>

defineProps并且defineEmits是编译器宏只能在<script setup>. 它们不需要导入，在<script setup>处理时会被编译掉。

defineProps接受与props选项相同的值，而defineEmits接受与选项相同的值emits。

defineProps并defineEmits根据传递的选项提供适当的类型推断。

传递给defineProps并将defineEmits从设置中提升到模块范围的选项。因此，选项不能引用在设置范围内声明的局部变量。这样做会导致编译错误。但是，它可以引用导入的绑定，因为它们也在模块范围内。

如果你使用 TypeScript，也可以使用纯类型注释来声明 props 和 emits。

定义暴露（）#

使用的组件默认<script setup>是关闭的- 即通过模板引用或$parent链检索的组件的公共实例不会暴露内部声明的任何绑定<script setup>。

要显式公开<script setup>组件中的属性，请使用defineExpose编译器宏：

<script setup>

import { ref } from 'vue'

const a = 1

const b = ref(2)

defineExpose({

a,

b

})

</script>

当父级通过模板 refs 获取此组件的实例时，检索到的实例将具有形状{ a: number, b: number }（就像在正常实例上一样，refs 会自动展开）。

useSlots()&useAttrs()#

slots和attrsinside的使用<script setup>应该相对较少，因为您可以直接在模板$slots中访问它们。$attrs在您确实需要它们的极少数情况下，请分别使用useSlots和useAttrs助手：

<script setup>

import { useSlots, useAttrs } from 'vue'

const slots = useSlots()

const attrs = useAttrs()

</script>

useSlotsanduseAttrs是实际的运行时函数，它返回等效于setupContext.slotsand setupContext.attrs。它们也可以用于普通的组合 API 函数。

正常使用<script>#

<script setup>可以和正常一起使用<script>。<script>在我们需要的情况下，可能需要正常：

声明不能用 表示的选项<script setup>，例如inheritAttrs或通过插件启用的自定义选项。

声明命名导出。

运行副作用或创建只应执行一次的对象。

<script>

// normal <script>, executed in module scope (only once)

runSideEffectOnce()

// declare additional options

export default {

inheritAttrs: false,

customOptions: {}

}

</script>

<script setup>

// executed in setup() scope (for each instance)

</script>

顶层await#

顶层await可以在里面使用<script setup>。生成的代码将编译为async setup()：

<script setup>

const post = await fetch(`/api/post/1`).then((r) => r.json())

</script>

此外，等待的表达式将自动编译为保留当前组件实例上下文的格式await。

笔记

async setup()必须与 结合使用Suspense，目前仍是实验性功能。我们计划在未来的版本中完成并记录它——但如果你现在好奇，你可以参考它的测试来看看它是如何工作的。

仅限 TypeScript 的功能#

仅类型的 props/emit 声明#

Props 和 emits 也可以通过将文字类型参数传递给definePropsor来使用纯类型语法声明defineEmits：

const props = defineProps<{

foo: string

bar?: number

}>()

const emit = defineEmits<{

(e: 'change', id: number): void

(e: 'update', value: string): void

}>()

defineProps或者defineEmits只能使用运行时声明或类型声明。同时使用两者会导致编译错误。

使用类型声明时，静态分析会自动生成等效的运行时声明，以消除双重声明的需要，并仍然确保正确的运行时行为。

在开发模式下，编译器将尝试从类型中推断出相应的运行时验证。例如这里foo: String是从类型推断出来的foo: string。如果类型是对导入类型的引用，则推断结果将是foo: null（等于any类型），因为编译器没有外部文件的信息。

在 prod 模式下，编译器会生成数组格式声明以减小 bundle 大小（这里的 props 会被编译成['foo', 'bar']）

发出的代码仍然是具有有效类型的 TypeScript，可以由其他工具进一步处理。

到目前为止，类型声明参数必须是以下之一，以确保正确的静态分析：

类型文字

对同一文件中的接口或类型文字的引用

目前不支持从其他文件导入复杂类型和类型。将来可以支持类型导入。

使用类型声明时的默认道具值#

仅类型声明的一个缺点defineProps是它无法为 props 提供默认值。为了解决这个问题，withDefaults还提供了一个编译器宏：

export interface Props {

msg?: string

labels?: string[]

}

const props = withDefaults(defineProps<Props>(), {

msg: 'hello',

labels: () => ['one', 'two']

})

这将被编译为等效的运行时道具default选项。此外，withDefaults帮助程序提供默认值的类型检查，并确保返回的props类型已删除了已声明默认值的属性的可选标志。

限制#

由于模块执行语义的不同，内部代码<script setup>依赖于 SFC 的上下文。当移动到外部.js或.ts文件中时，可能会导致开发人员和工具的混乱。因此，<script setup>不能与src属性一起使用。

# 第21章 单文件组件：证监会 CSS 功能

作用域 CSS#

当<style>标签具有该scoped属性时，其 CSS 将仅适用于当前组件的元素。这类似于 Shadow DOM 中的样式封装。它带有一些警告，但不需要任何 polyfill。它是通过使用 PostCSS 转换以下内容来实现的：

<style scoped>

.example {

color: red;

}

</style>

<template>

<div class="example">hi</div>

</template>

进入以下：

<style>

.example[data-v-f3f3eg9] {

color: red;

}

</style>

<template>

<div class="example" data-v-f3f3eg9>hi</div>

</template>

子组件根元素#

使用scoped，父组件的样式不会泄漏到子组件中。但是，子组件的根节点会同时受到父级 CSS 和子级 CSS 的影响。这是设计使然，以便父级可以为子根元素设置样式以用于布局目的。

深度选择器#

如果您希望scoped样式中的选择器“深”，即影响子组件，则可以使用:deep()伪类：

<style scoped>

.a :deep(.b) {

/\* ... \*/

}

</style>

以上将被编译成：

.a[data-v-f3f3eg9] .b {

/\* ... \*/

}

小费

创建的 DOM 内容v-html不受作用域样式的影响，但您仍然可以使用深度选择器对其进行样式设置。

开槽选择器#

默认情况下，作用域样式不会影响由 呈现的内容<slot/>，因为它们被认为是由传递它们的父组件拥有。要显式定位插槽内容，请使用:slotted伪类：

<style scoped>

:slotted(div) {

color: red;

}

</style>

全局选择器#

如果您只想全局应用一个规则，则可以使用:global伪类而不是创建另一个规则<style>（见下文）：

<style scoped>

:global(.red) {

color: red;

}

</style>

混合本地和全球风格#

您还可以在同一个组件中同时包含作用域和非作用域样式：

<style>

/\* global styles \*/

</style>

<style scoped>

/\* local styles \*/

</style>

范围样式提示#

范围样式不会消除对类的需要。由于浏览器呈现各种 CSS 选择器的方式，p { color: red }在作用域（即与属性选择器组合时）会慢很多倍。如果您改用类或 ID，例如 in .example { color: red }，那么您实际上消除了对性能的影响。

小心递归组件中的后代选择器！对于带有 selector 的 CSS 规则.a .b，如果匹配的元素.a包含递归子组件，则.b该子组件中的所有内容都将被该规则匹配。

CSS 模块#

<style module>标签被编译为CSS 模块，并将生成的 CSS 类作为以下键下的对象公开给组件$style：

<template>

<p :class="$style.red">This should be red</p>

</template>

<style module>

.red {

color: red;

}

</style>

生成的类被散列以避免冲突，达到将 CSS 仅作用于当前组件的相同效果。

有关全局异常和组合等更多详细信息，请参阅CSS 模块规范。

自定义注入名称#

module您可以通过给属性一个值来自定义注入的类对象的属性键：

<template>

<p :class="classes.red">red</p>

</template>

<style module="classes">

.red {

color: red;

}

</style>

使用组合 API#

可以通过API访问注入的setup()类。对于具有自定义注入名称的块，接受匹配的属性值作为第一个参数：<script setup>useCssModule<style module>useCssModulemodule

import { useCssModule } from 'vue'

// inside setup() scope...

// default, returns classes for <style module>

useCssModule()

// named, returns classes for <style module="classes">

useCssModule('classes')

v-bind()在 CSS 中#

SFC<style>标签支持使用v-bindCSS 函数将 CSS 值链接到动态组件状态：

<template>

<div class="text">hello</div>

</template>

<script>

export default {

data() {

return {

color: 'red'

}

}

}

</script>

<style>

.text {

color: v-bind(color);

}

</style>

该语法适用于<script setup>，并支持 JavaScript 表达式（必须用引号括起来）：

<script setup>

const theme = {

color: 'red'

}

</script>

<template>

<p>hello</p>

</template>

<style scoped>

p {

color: v-bind('theme.color');

}

</style>

实际值将被编译成散列的 CSS 自定义属性，因此 CSS 仍然是静态的。自定义属性将通过内联样式应用于组件的根元素，并在源值更改时进行响应式更新。

# 第22章 高级API：渲染函数 API

H（）#

创建虚拟 DOM 节点（vnodes）。

类型

// full signature

function h(

type: string | Component,

props?: object | null,

children?: Children | Slot | Slots

): VNode

// omitting props

function h(type: string | Component, children?: Children | Slot): VNode

type Children = string | number | boolean | VNode | null | Children[]

type Slot = () => Children

type Slots = { [name: string]: Slot }

为了便于阅读，类型被简化了。

细节

第一个参数可以是字符串（用于原生元素）或 Vue 组件定义。第二个参数是要传递的道具，第三个参数是孩子。

创建组件 vnode 时，子节点必须作为插槽函数传递。如果组件只需要默认槽，则可以传递单个槽函数。否则，槽必须作为槽函数的对象传递。

为方便起见，当 children 不是 slot 对象时，可以省略 props 参数。

例子

创建原生元素：

import { h } from 'vue'

// all arguments except the type are optional

h('div')

h('div', { id: 'foo' })

// both attributes and properties can be used in props

// Vue automatically picks the right way to assign it

h('div', { class: 'bar', innerHTML: 'hello' })

// class and style have the same object / array

// value support like in templates

h('div', { class: [foo, { bar }], style: { color: 'red' } })

// event listeners should be passed as onXxx

h('div', { onClick: () => {} })

// children can be a string

h('div', { id: 'foo' }, 'hello')

// props can be omitted when there are no props

h('div', 'hello')

h('div', [h('span', 'hello')])

// children array can contain mixed vnodes and strings

h('div', ['hello', h('span', 'hello')])

创建组件：

import Foo from './Foo.vue'

// passing props

h(Foo, {

// equivalent of some-prop="hello"

someProp: 'hello',

// equivalent of @update="() => {}"

onUpdate: () => {}

})

// passing single default slot

h(Foo, () => 'default slot')

// passing named slots

// notice the `null` is required to avoid

// slots object being treated as props

h(MyComponent, null, {

default: () => 'default slot',

foo: () => h('div', 'foo'),

bar: () => [h('span', 'one'), h('span', 'two')]

})

另请参阅： 指南 - 渲染函数 - 创建 VNode

合并道具（）#

合并多个道具对象，对某些道具进行特殊处理。

类型

function mergeProps(...args: object[]): object

细节

mergeProps()支持合并多个道具对象，对以下道具进行特殊处理：

class

style

onXxx事件监听器 - 多个具有相同名称的监听器将被合并到一个数组中。

如果您不需要合并行为并且想要简单的覆盖，则可以使用原生对象扩展。

例子

import { mergeProps } from 'vue'

const one = {

class: 'foo',

onClick: handlerA

}

const two = {

class: { bar: true },

onClick: handlerB

}

const merged = mergeProps(one, two)

/\*\*

{

class: 'foo bar',

onClick: [handlerA, handlerB]

}

\*/

克隆VNode()#

克隆一个 vnode。

类型

function cloneVNode(vnode: VNode, extraProps?: object): VNode

细节

返回一个克隆的 vnode，可以选择使用额外的 props 与原始节点合并。

Vnode 一旦创建就应该被认为是不可变的，你不应该改变现有 vnode 的 props。相反，用不同的/额外的道具克隆它。

Vnodes具有特殊的内部属性，因此克隆它们并不像对象传播那么简单。cloneVNode()处理大部分内部逻辑。

例子

import { h, cloneVNode } from 'vue'

const original = h('div')

const cloned = cloneVNode(original, { id: 'foo' })

是VNode()#

检查一个值是否是一个 vnode。

类型

function isVNode(value: unknown): boolean

解决组件（）#

用于按名称手动解析已注册的组件。

类型

function resolveComponent(name: string): Component | string

细节

注意：如果您可以直接导入组件，则不需要此操作。

resolveComponent()必须在渲染函数内部调用才能从正确的组件上下文中解析。

如果未找到该组件，将发出运行时警告，并返回名称字符串。

例子

const { h, resolveComponent } = Vue

export default {

render() {

const ButtonCounter = resolveComponent('ButtonCounter')

return h(ButtonCounter)

}

}

另请参阅： 指南 - 渲染函数 - 组件

解决指令（）#

用于按名称手动解析已注册的指令。

类型

function resolveDirective(name: string): Directive | undefined

细节

注意：如果您可以直接导入组件，则不需要此操作。

resolveDirective()必须在渲染函数内部调用才能从正确的组件上下文中解析。

如果找不到该指令，将发出运行时警告，并且函数返回undefined.

另请参阅： 指南 - 渲染函数 - 自定义指令

withDirectives()#

用于向 vnode 添加自定义指令。

类型

function withDirectives(

vnode: VNode,

directives: DirectiveArguments

): VNode

// [Directive, value, argument, modifiers]

type DirectiveArguments = Array<

| [Directive]

| [Directive, any]

| [Directive, any, string]

| [Directive, any, string, DirectiveModifiers]

>

细节

使用自定义指令包装现有 vnode。第二个参数是一个自定义指令数组。每个自定义指令也表示为 . 形式的数组[Directive, value, argument, modifiers]。如果不需要，可以省略数组的尾元素。

例子

import { h, withDirectives } from 'vue'

// a custom directive

const pin = {

mounted() {

/\* ... \*/

},

updated() {

/\* ... \*/

}

}

// <div v-pin:top.animate="200"></div>

const vnode = withDirectives(h('div'), [

[pin, 200, 'top', { animate: true }]

])

另请参阅： 指南 - 渲染函数 - 自定义指令

withModifiers()#

用于向事件处理函数添加内置v-on修饰符。

类型

function withModifiers(fn: Function, modifiers: string[]): Function

例子

import { h, withModifiers } from 'vue'

const vnode = h('button', {

// equivalent of v-on.stop.prevent

onClick: withModifiers(() => {

// ...

}, ['stop', 'prevent'])

})

# 第23章 高级API：服务器端渲染 API

渲染字符串（）#

导出自vue/server-renderer

类型

function renderToString(

input: App | VNode,

context?: SSRContext

): Promise<string>

例子

import { createSSRApp } from 'vue'

import { renderToString } from 'vue/server-renderer'

const app = createSSRApp({

data: () => ({ msg: 'hello' }),

template: `<div>{{ msg }}</div>`

})

;(async () => {

const html = await renderToString(app)

console.log(html)

})()

SSR 上下文#

您可以传递一个可选的上下文对象，该对象可用于在渲染期间记录其他数据，例如访问 Teleports 的内容：

const ctx = {}

const html = await renderToString(app, ctx)

console.log(ctx.teleports) // { '#teleported': 'teleported content' }

此页面上的大多数其他 SSR API 也可以选择接受上下文对象。可以通过useSSRContext帮助器在组件代码中访问上下文对象。

另请参阅： 指南 - 服务器端渲染

渲染节点流（）#

将输入呈现为Node.js 可读流。

导出自vue/server-renderer

类型

function renderToNodeStream(

input: App | VNode,

context?: SSRContext

): Readable

例子

// inside a Node.js http handler

renderToNodeStream(app).pipe(res)

笔记

vue/server-renderer与 Node.js 环境分离的 ESM 构建不支持此方法。改为使用pipeToNodeWritable。

pipeToNodeWritable()#

渲染和管道到现有的Node.js 可写流实例。

导出自vue/server-renderer

类型

function pipeToNodeWritable(

input: App | VNode,

context: SSRContext = {},

writable: Writable

): void

例子

// inside a Node.js http handler

pipeToNodeWritable(app, {}, res)

渲染到WebStream()#

将输入呈现为Web ReadableStream。

导出自vue/server-renderer

类型

function renderToWebStream(

input: App | VNode,

context?: SSRContext

): ReadableStream

例子

// inside an environment with ReadableStream support

return new Response(renderToWebStream(app))

笔记

ReadableStream在未在全局范围内公开构造函数的环境中，pipeToWebWritable()应改为使用。

pipeToWebWritable()#

渲染和管道到现有的Web WritableStream实例。

导出自vue/server-renderer

类型

function pipeToWebWritable(

input: App | VNode,

context: SSRContext = {},

writable: WritableStream

): void

例子

这通常与以下组合使用TransformStream：

// TransformStream is available in environments such as CloudFlare workers.

// in Node.js, TransformStream needs to be explicitly imported from 'stream/web'

const { readable, writable } = new TransformStream()

pipeToWebWritable(app, {}, writable)

return new Response(readable)

renderToSimpleStream()#

使用简单的可读界面以流模式呈现输入。

导出自vue/server-renderer

类型

function renderToSimpleStream(

input: App | VNode,

context: SSRContext,

options: SimpleReadable

): SimpleReadable

interface SimpleReadable {

push(content: string | null): void

destroy(err: any): void

}

例子

let res = ''

renderToSimpleStream(

app,

{},

{

push(chunk) {

if (chunk === null) {

// done

console(`render complete: ${res}`)

} else {

res += chunk

}

},

destroy(err) {

// error encountered

}

}

)

使用SSRContext()#

用于检索传递给renderToString()或其他服务器呈现 API 的上下文对象的运行时 API。

类型

function useSSRContext<T = Record<string, any>>(): T | undefined

例子

检索到的上下文可用于附加呈现最终 HTML 所需的信息（例如头部元数据）。

<script setup>

import { useSSRContext } from 'vue'

// make sure to only call it during SSR

// https://vitejs.dev/guide/ssr.html#conditional-logic

if (import.meta.env.SSR) {

const ctx = useSSRContext()

// ...attach properties to the context

}

</script>

在 GitHub 上编辑此页面

# 第24章 高级API：TypeScript实用程序类型

道具类型<T>#

用于在使用运行时 props 声明时使用更高级的类型注释 prop。

例子

import { PropType } from 'vue'

interface Book {

title: string

author: string

year: number

}

export default {

props: {

book: {

// provide more specific type to `Object`

type: Object as PropType<Book>,

required: true

}

}

}

另请参阅： 指南 - 键入组件道具

组件自定义属性#

用于扩充组件实例类型以支持自定义全局属性。

例子

import axios from 'axios'

declare module 'vue' {

interface ComponentCustomProperties {

$http: typeof axios

$translate: (key: string) => string

}

}

小费

增强必须放在模块.ts或.d.ts文件中。有关更多详细信息，请参阅类型增强放置。

另请参阅： 指南 - 增强全局属性

组件自定义选项#

用于扩充组件选项类型以支持自定义选项。

例子

import { Route } from 'vue-router'

declare module 'vue' {

interface ComponentCustomOptions {

beforeRouteEnter?(to: any, from: any, next: () => void): void

}

}

小费

增强必须放在模块.ts或.d.ts文件中。有关更多详细信息，请参阅类型增强放置。

另请参阅： 指南 - 增强自定义选项

组件自定义道具#

用于扩充允许的 TSX 道具，以便在 TSX 元素上使用未声明的道具。

例子

declare module 'vue' {

interface ComponentCustomProps {

hello?: string

}

}

export {}

// now works even if hello is not a declared prop

<MyComponent hello="world" />

小费

增强必须放在模块.ts或.d.ts文件中。有关更多详细信息，请参阅类型增强放置。

CSS 属性#

用于增加样式属性绑定中的允许值。

例子

允许任何自定义 CSS 属性

declare module 'vue' {

interface CSSProperties {

[key: `--${string}`]: string

}

}

<div style={ { '--bg-color': 'blue' } }>

<div :style="{ '--bg-color': 'blue' }">

小费

增强必须放在模块.ts或.d.ts文件中。有关更多详细信息，请参阅类型增强放置。

也可以看看

SFC标签支持使用该函数<style>将 CSS 值链接到动态组件状态。v-bind CSS这允许在没有类型扩充的情况下使用自定义属性。

CSS 中的 v-bind()

# 第25章 高级API：自定义渲染器 API

创建渲染器（）#

创建自定义渲染器。通过提供特定于平台的节点创建和操作 API，您可以利用 Vue 的核心运行时来定位非 DOM 环境。

类型

function createRenderer<HostNode, HostElement>(

options: RendererOptions<HostNode, HostElement>

): Renderer<HostElement>

interface Renderer<HostElement> {

render: RootRenderFunction<HostElement>

createApp: CreateAppFunction<HostElement>

}

interface RendererOptions<HostNode, HostElement> {

patchProp(

el: HostElement,

key: string,

prevValue: any,

nextValue: any,

// the rest is unused for most custom renderers

isSVG?: boolean,

prevChildren?: VNode<HostNode, HostElement>[],

parentComponent?: ComponentInternalInstance | null,

parentSuspense?: SuspenseBoundary | null,

unmountChildren?: UnmountChildrenFn

): void

insert(

el: HostNode,

parent: HostElement,

anchor?: HostNode | null

): void

remove(el: HostNode): void

createElement(

type: string,

isSVG?: boolean,

isCustomizedBuiltIn?: string,

vnodeProps?: (VNodeProps & { [key: string]: any }) | null

): HostElement

createText(text: string): HostNode

createComment(text: string): HostNode

setText(node: HostNode, text: string): void

setElementText(node: HostElement, text: string): void

parentNode(node: HostNode): HostElement | null

nextSibling(node: HostNode): HostNode | null

// optional, DOM-specific

querySelector?(selector: string): HostElement | null

setScopeId?(el: HostElement, id: string): void

cloneNode?(node: HostNode): HostNode

insertStaticContent?(

content: string,

parent: HostElement,

anchor: HostNode | null,

isSVG: boolean

): [HostNode, HostNode]

}

例子

import { createRenderer } from '@vue/runtime-core'

const { render, createApp } = createRenderer({

patchProp,

insert,

remove,

createElement

// ...

})

// `render` is the low-level API

// `createApp` returns an app instance

export { render, createApp }

// re-export Vue core APIs

export \* from '@vue/runtime-core'

Vue 自己@vue/runtime-dom是使用相同的 API 实现的。对于更简单的实现，请查看@vue/runtime-test哪个是 Vue 自己的单元测试的私有包。