第一章 Getting Started 开始

1.1 introduction

You are reading the documentation for Vue 3!

- Vue 2 support will end on Dec 31, 2023. Learn more about Vue 2 Extended LTS.
- Vue 2 documentation has been moved to v2.vuejs.org.
- Upgrading from Vue 2? Check out the Migration Guide.

1.1.1 What is Vue?

Vue (pronounced /vjuː/, like **view**) is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be they simple or complex.

Here is a minimal example:

js

```js
import { createApp, ref } from 'vue'

createApp({
  setup() {
    return {
      count: ref(0)
    }
  }
}).mount('#app')
```

1.1 介绍

你正在阅读的是 Vue 3 的文档！

- Vue 2 将于 2023 年 12 月 31 日停止维护。详见 Vue 2 延长 LTS。
- Vue 2 中文文档已迁移至 v2.cn.vuejs.org。
- 想从 Vue 2 升级？请参考迁移指南。

1.1.1 什么是 Vue?

Vue (发音为 /vjuː/，类似 **view**) 是用于构建用户界面的 JavaScript 框架。它基于标准 HTML、CSS 和 JavaScript 构建，并提供了一套声明式的、组件化的编程模型，助你高效地开发用户界面。无论是简单还是复杂的界面，Vue 都可以胜任。

下面是一个最基本的示例：

js:UMD 浏览器引用 JS 方式

```js
const {createApp,ref} = Vue;

createApp({
    setup() {
    return {
        count: ref(0)
    }
    }
}).mount('#app')
```

template

```
<div id="app">
    <button @click="count++">
        Count is: {{ count }}
    </button>
</div>
```

The above example demonstrates the two core features of Vue:

- **Declarative Rendering**: Vue extends standard HTML with a template syntax that allows us to declaratively describe HTML output based on JavaScript state.
- **Reactivity**: Vue automatically tracks JavaScript state changes and efficiently updates the DOM when changes happen.

You may already have questions - don't worry. We will cover every little detail in the rest of the documentation. For now, please read along so you can have a high-level understanding of what Vue offers.

> **Prerequisites**
> The rest of the documentation assumes basic familiarity with HTML, CSS, and JavaScript. If you are totally new to frontend development, it might not be the best idea to jump right into a framework as your first step - grasp the basics and then come back! You can check your knowledge level with this JavaScript overview. Prior experience with other frameworks helps, but is not required.

1.1.2 The Progressive Framework

Vue is a framework and ecosystem that covers most of the common features needed in frontend development. But the web is extremely diverse - the things we build on the web may vary drastically in form and scale. With that in mind, Vue is designed to be flexible and incrementally adoptable. Depending on your use case, Vue can be used in different ways:

- Enhancing static HTML without a build step
- Embedding as Web Components on any page
- Single-Page Application (SPA)

template

```
<div id="app">
    <button @click="count++">
        Count is: {{ count }}
    </button>
</div>
```

上面的示例展示了 Vue 的两个核心功能：

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

你可能已经有了些疑问——先别急，在后续的文档中我们会详细介绍每一个细节。现在，请继续看下去，以确保你对 Vue 作为一个框架到底提供了什么有一个宏观的了解。

> **预备知识**
> 文档接下来的内容会假设你对 HTML、CSS 和 JavaScript 已经基本熟悉。如果你对前端开发完全陌生，最好不要直接从一个框架开始进行入门学习——最好是掌握了基础知识再回到这里。你可以通过这篇 JavaScript 概述来检验你的 JavaScript 知识水平。如果之前有其他框架的经验会很有帮助，但也不是必须的。

1.1.2 渐进式框架

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

- 无需构建步骤，渐进式增强静态的 HTML
- 在任何页面中作为 Web Components 嵌入
- 单页应用 (SPA)

- Fullstack / Server-Side Rendering (SSR)
- Jamstack / Static Site Generation (SSG)
- Targeting desktop, mobile, WebGL, and even the terminal

If you find these concepts intimidating, don't worry! The tutorial and guide only require basic HTML and JavaScript knowledge, and you should be able to follow along without being an expert in any of these.

If you are an experienced developer interested in how to best integrate Vue into your stack, or you are curious about what these terms mean, we discuss them in more detail in Ways of Using Vue.

Despite the flexibility, the core knowledge about how Vue works is shared across all these use cases. Even if you are just a beginner now, the knowledge gained along the way will stay useful as you grow to tackle more ambitious goals in the future. If you are a veteran, you can pick the optimal way to leverage Vue based on the problems you are trying to solve, while retaining the same productivity. This is why we call Vue "The Progressive Framework": it's a framework that can grow with you and adapt to your needs.

1.1.3 Single-File Components

In most build-tool-enabled Vue projects, we author Vue components using an HTML-like file format called **Single-File Component** (also known as `*.vue` files, abbreviated as **SFC**). A Vue SFC, as the name suggests, encapsulates the component's logic (JavaScript), template (HTML), and styles (CSS) in a single file. Here's the previous example, written in SFC format:

vue

```vue
<script setup>
import { ref } from 'vue'
const count = ref(0)
</script>

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

- 全栈 / 服务端渲染 (SSR)
- Jamstack / 静态站点生成 (SSG)
- 开发桌面端、移动端、WebGL，甚至是命令行终端中的界面

如果你是初学者，可能会觉得这些概念有些复杂。别担心！理解教程和指南的内容只需要具备基础的 HTML 和 JavaScript 知识。即使你不是这些方面的专家，也能够跟得上。

如果你是有经验的开发者，希望了解如何以最合适的方式在项目中引入 Vue，或者是对上述的这些概念感到好奇，我们在使用 Vue 的多种方式中讨论了有关它们的更多细节。

无论再怎么灵活，Vue 的核心知识在所有这些用例中都是通用的。即使你现在只是一个初学者，随着你的不断成长，到未来有能力实现更复杂的项目时，这一路上获得的知识依然会适用。如果你已经是一个老手，你可以根据实际场景来选择使用 Vue 的最佳方式，在各种场景下都可以保持同样的开发效率。这就是为什么我们将 Vue 称为"渐进式框架"：它是一个可以与你共同成长、适应你不同需求的框架。

1.1.3 单文件组件

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

vue

```vue
<script setup>
import { ref } from 'vue'
const count = ref(0)
</script>

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

```
<style scoped>
button {
    font-weight: bold;
}
</style>
```

SFC is a defining feature of Vue and is the recommended way to author Vue components **if** your use case warrants a build setup. You can learn more about the how and why of SFC in its dedicated section - but for now, just know that Vue will handle all the build tools setup for you.

1.1.4 API Styles

Vue components can be authored in two different API styles:**Options API** and **Composition API**.

Options API

With Options API, we define a component's logic using an object of options such as `data`, `methods`, and `mounted`. Properties defined by options are exposed on `this` inside functions, which points to the component instance:

```vue
  <script>
  export default {
    // Properties returned from data() become reactive state
    // and will be exposed on `this`.
    data() {
      return {
        count: 0
      }
    },

    // Methods are functions that mutate state and trigger updates.
    // They can be bound as event handlers in templates.
    methods: {
      increment() {
```

```
<style scoped>
button {
    font-weight: bold;
}
</style>
```

单文件组件是 Vue 的标志性功能。如果你的用例需要进行构建，我们推荐用它来编写 Vue 组件。你可以在后续相关章节里了解更多关于单文件组件的用法及用途。但你暂时只需要知道 Vue 会帮忙处理所有这些构建工具的配置就好。

1.1.4 API 风格

Vue 的组件可以按两种不同的风格书写：**选项式 API** 和**组合式 API**。

选项式 API (Options API)

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

```vue
  <script>
  export default {
    // data() 返回的属性将会成为响应式的状态
    // 并且暴露在 `this` 上
    data() {
      return {
        count: 0
      }
    },

    // methods 是一些用来更改状态与触发更新的函数
    // 它们可以在模板中作为事件处理器绑定
    methods: {
      increment() {
```

```
      this.count++
    }
  },

  // Lifecycle hooks are called at different stages
  // of a component's lifecycle.
  // This function will be called when the component is mounted.
  mounted() {
    console.log(`The initial count is ${this.count}.`)
  }
}
</script>

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

Composition API

With Composition API, we define a component's logic using imported API functions. In SFCs, Composition API is typically used with “. The `setup` attribute is a hint that makes Vue perform compile-time transforms that allow us to use Composition API with less boilerplate. For example, imports and top-level variables / functions declared in `<script setup>` are directly usable in the template.

Here is the same component, with the exact same template, but using Composition API and `<script setup>` instead:

vue

```
<script setup>
import { ref, onMounted } from 'vue'

// reactive state
const count = ref(0)

// functions that mutate state and trigger updates
function increment() {
```

```
      this.count++
    }
  },

  // 生命周期钩子会在组件生命周期的各个不同阶段被调用
  // 例如这个函数就会在组件挂载完成后被调用
  mounted() {
    console.log(`The initial count is ${this.count}.`)
  }
}
</script>

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

组合式 API (Composition API)

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

下面是使用了组合式 API 与 `<script setup>` 改造后和上面的模板完全一样的组件：

vue

```
<script setup>
import { ref, onMounted } from 'vue'

// 响应式状态
const count = ref(0)

// 用来修改状态、触发更新的函数
function increment() {
    count.value++
  }

  // lifecycle hooks
  onMounted(() => {
    console.log(`The initial count is ${count.value}.`)
  })
</script>

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

Which to Choose?

Both API styles are fully capable of covering common use cases. They are different interfaces powered by the exact same underlying system. In fact, the Options API is implemented on top of the Composition API! The fundamental concepts and knowledge about Vue are shared across the two styles.

The Options API is centered around the concept of a ”component instance” (`this` as seen in the example), which typically aligns better with a class-based mental model for users coming from OOP language backgrounds. It is also more beginner-friendly by abstracting away the reactivity details and enforcing code organization via option groups.

The Composition API is centered around declaring reactive state variables directly in a function scope and composing state from multiple functions together to handle complexity. It is more free-form and requires an understanding of how reactivity works in Vue to be used effectively. In return, its flexibility enables more powerful patterns for organizing and reusing logic.

You can learn more about the comparison between the two styles and the potential benefits of Composition API in the Composition API FAQ.

If you are new to Vue, here’s our general recommendation:

- For learning purposes, go with the style that looks easier to understand to you. Again, most of the core concepts are shared between the two styles. You can always pick up the other style

```
    count.value++
  }

  // 生命周期钩子
  onMounted(() => {
    console.log(`The initial count is ${count.value}.`)
  })
</script>

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

该选哪一个？

两种 API 风格都能够覆盖大部分的应用场景。它们只是同一个底层系统所提供的两套不同的接口。实际上，选项式 API 是在组合式 API 的基础上实现的！关于 Vue 的基础概念和知识在它们之间都是通用的。

选项式 API 以 “组件实例” 的概念为中心 (即上述例子中的 `this`)，对于有面向对象语言背景的用户来说，这通常与基于类的心智模型更为一致。同时，它将响应性相关的细节抽象出来，并强制按照选项来组织代码，从而对初学者而言更为友好。

组合式 API 的核心思想是直接在函数作用域内定义响应式状态变量，并将从多个函数中得到的状态组合起来处理复杂问题。这种形式更加自由，也需要你对 Vue 的响应式系统有更深的理解才能高效使用。相应的，它的灵活性也使得组织和重用逻辑的模式变得更加强大。

在组合式 API FAQ 章节中，你可以了解更多关于这两种 API 风格的对比以及组合式 API 所带来的潜在收益。

如果你是使用 Vue 的新手，这里是我们的大致建议：

- 在学习的过程中，推荐采用更易于自己理解的风格。再强调一下，大部分的核心概念在这两种风格之间都是通用的。熟悉了一种风格以后，你也能够很

later.

- For production use:
  - Go with Options API if you are not using build tools, or plan to use Vue primarily in low-complexity scenarios, e.g. progressive enhancement.
  - Go with Composition API + Single-File Components if you plan to build full applications with Vue.

You don't have to commit to only one style during the learning phase. The rest of the documentation will provide code samples in both styles where applicable, and you can toggle between them at any time using the **API Preference switches** at the top of the left sidebar.

1.1.5 Still Got Questions?

Check out our FAQ.

1.1.6 Pick Your Learning Path

Different developers have different learning styles. Feel free to pick a learning path that suits your preference - although we do recommend going over all of the content, if possible!

Try the TutorialFor those who prefer learning things hands-on.Read the GuideThe guide walks you through every aspect of the framework in full detail.Check out the ExamplesExplore examples of core features and common UI tasks.

1.2 Quick Start

1.2.1 Try Vue Online

- To quickly get a taste of Vue, you can try it directly in our Playground.
- If you prefer a plain HTML setup without any build steps, you can use this JSFiddle as your starting point.
- If you are already familiar with Node.js and the concept of build tools, you can also try a

快地理解另一种风格。

- 在生产项目中：
  - 当你不需要使用构建工具，或者打算主要在低复杂度的场景中使用 Vue，例如渐进增强的应用场景，推荐采用选项式 API。
  - 当你打算用 Vue 构建完整的单页应用，推荐采用组合式 API + 单文件组件。

在学习阶段，你不必只固守一种风格。在接下来的文档中我们会为你提供一系列两种风格的代码供你参考，你可以随时通过左上角的 **API 风格偏好**来做切换。

1.1.5 还有其他问题？

请查看我们的 FAQ。

1.1.6 选择你的学习路径

不同的开发者有不同的学习方式。尽管在可能的情况下，我们推荐你通读所有内容，但你还是可以自由地选择一种自己喜欢的学习路径！

尝试互动教程适合喜欢边动手边学的读者。继续阅读该指南该指南会带你深入了解框架所有方面的细节。查看示例浏览核心功能和常见用户界面的示例。

1.2 快速上手

1.2.1 线上尝试 Vue

- 想要快速体验 Vue，你可以直接试试我们的演练场。
- 如果你更喜欢不用任何构建的原始 HTML，可以使用 JSFiddle 入门。
- 如果你已经比较熟悉 Node.js 和构建工具等概念，还可以直接在浏览器中打开 StackBlitz 来尝试完整的构建设置。

complete build setup right within your browser on StackBlitz.

1.2.2 Creating a Vue Application

Prerequisites

- Familiarity with the command line
- Install Node.js version 16.0 or higher

In this section we will introduce how to scaffold a Vue Single Page Application on your local machine. The created project will be using a build setup based on Vite and allow us to use Vue Single-File Components (SFCs).

Make sure you have an up-to-date version of Node.js installed and your current working directory is the one where you intend to create a project. Run the following command in your command line (without the > sign):

```
virhuiai %> npm create vue@latest
```

This command will install and execute create-vue, the official Vue project scaffolding tool. You will be presented with prompts for several optional features such as TypeScript and testing support:

```
✓ Project name: … <your-project-name>
✓ Add TypeScript? … No / Yes
✓ Add JSX Support? … No / Yes
✓ Add Vue Router for Single Page Application development? … No / Yes
✓ Add Pinia for state management? … No / Yes
✓ Add Vitest for Unit testing? … No / Yes
✓ Add an End-to-End Testing Solution? … No / Cypress / Playwright
✓ Add ESLint for code quality? … No / Yes
✓ Add Prettier for code formatting? … No / Yes

Scaffolding project in ./<your-project-name>...
Done.
```

If you are unsure about an option, simply choose `No` by hitting enter for now. Once the project is created, follow the instructions to install dependencies and start the dev server:

1.2.2 创建一个 Vue 应用

前提条件

- 熟悉命令行
- 已安装 16.0 或更高版本的 Node.js

在本节中，我们将介绍如何在本地搭建 Vue 单页应用。创建的项目将使用基于 Vite 的构建设置，并允许我们使用 Vue 的单文件组件 (SFC)。

确保你安装了最新版本的 Node.js，并且你的当前工作目录正是打算创建项目的目录。在命令行中运行以下命令 (不要带上 > 符号)：

```
virhuiai %> npm create vue@latest
```

这一指令将会安装并执行 create-vue，它是 Vue 官方的项目脚手架工具。你将会看到一些诸如 TypeScript 和测试支持之类的可选功能提示：

```
✓ Project name: … <your-project-name>
✓ Add TypeScript? … No / Yes
✓ Add JSX Support? … No / Yes
✓ Add Vue Router for Single Page Application development? … No / Yes
✓ Add Pinia for state management? … No / Yes
✓ Add Vitest for Unit testing? … No / Yes
✓ Add an End-to-End Testing Solution? … No / Cypress / Playwright
✓ Add ESLint for code quality? … No / Yes
✓ Add Prettier for code formatting? … No / Yes

Scaffolding project in ./<your-project-name>...
Done.
```

如果不确定是否要开启某个功能，你可以直接按下回车键选择 `No`。在项目被创建后，通过以下步骤安装依赖并启动开发服务器：

```
virhuiai $> cd <your-project-name>
virhuiai $> npm install
virhuiai $> npm run dev
```

You should now have your first Vue project running! Note that the example components in the generated project are written using the Composition API and `<script setup>`, rather than the Options API. Here are some additional tips:

- The recommended IDE setup is Visual Studio Code + Volar extension. If you use other editors, check out the IDE support section.
- More tooling details, including integration with backend frameworks, are discussed in the Tooling Guide.
- To learn more about the underlying build tool Vite, check out the Vite docs.
- If you choose to use TypeScript, check out the TypeScript Usage Guide.

When you are ready to ship your app to production, run the following:

```
virhuiai $> npm run build
```

This will create a production-ready build of your app in the project's `./dist` directory. Check out the Production Deployment Guide to learn more about shipping your app to production.

Next Steps >

1.2.3 Using Vue from CDN

You can use Vue directly from a CDN via a script tag:

```html
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
```

Here we are using unpkg, but you can also use any CDN that serves npm packages, for example jsdelivr or cdnjs. Of course, you can also download this file and serve it yourself.

```
virhuiai $> cd <your-project-name>
virhuiai $> npm install
virhuiai $> npm run dev
```

你现在应该已经运行起来了你的第一个 Vue 项目！请注意，生成的项目中的示例组件使用的是组合式 API 和 `<script setup>`，而非选项式 API。下面是一些补充提示：

- 推荐的 IDE 配置是 Visual Studio Code + Volar 扩展。如果使用其他编辑器，参考 IDE 支持章节。
- 更多工具细节，包括与后端框架的整合，我们会在工具链指南进行讨论。
- 要了解构建工具 Vite 更多背后的细节，请查看 Vite 文档。
- 如果你选择使用 TypeScript，请阅读 TypeScript 使用指南。

当你准备将应用发布到生产环境时，请运行：

```
virhuiai $> npm run build
```

此命令会在 `./dist` 文件夹中为你的应用创建一个生产环境的构建版本。关于将应用上线生产环境的更多内容，请阅读生产环境部署指南。

下一步 >

1.2.3 通过 CDN 使用 Vue

你可以借助 script 标签直接通过 CDN 来使用 Vue：

```html
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
```

这里我们使用了 unpkg，但你也可以使用任何提供 npm 包服务的 CDN，例如 jsdelivr 或 cdnjs。当然，你也可以下载此文件并自行提供服务。

When using Vue from a CDN, there is no ”build step” involved. This makes the setup a lot simpler, and is suitable for enhancing static HTML or integrating with a backend framework. However, you won’t be able to use the Single-File Component (SFC) syntax.

Using the Global Build

The above link loads the *global build* of Vue, where all top-level APIs are exposed as properties on the global `Vue` object. Here is a full example using the global build:

```html
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

<div id="app">{{ message }}</div>

<script>
    const { createApp, ref } = Vue

    createApp({
    setup() {
        const message = ref('Hello vue!')
        return {
        message
        }
    }
    }).mount('#app')
</script>
```

Codepen demo

TIP

Many of the examples for Composition API throughout the guide will be using the `<script setup>` syntax, which requires build tools. If you intend to use Composition API without a build step, consult the usage of the `setup()` option.

Using the ES Module Build

Throughout the rest of the documentation, we will be primarily using ES modules syntax. Most modern browsers now support ES modules natively, so we can use Vue from a CDN via native ES

通过 CDN 使用 Vue 时，不涉及 “构建步骤”。这使得设置更加简单，并且可以用于增强静态的 HTML 或与后端框架集成。但是，你将无法使用单文件组件 (SFC) 语法。

使用全局构建版本

上面的链接使用了*全局构建版本*的 Vue，该版本的所有顶层 API 都以属性的形式暴露在了全局的 `Vue` 对象上。这里有一个使用全局构建版本的例子：

```html
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

<div id="app">{{ message }}</div>

<script>
    const { createApp, ref } = Vue

    createApp({
    setup() {
        const message = ref('Hello vue!')
        return {
        message
        }
    }
    }).mount('#app')
</script>
```

Codepen 示例

TIP

本指南中许多关于组合式 API 的例子将使用 `<script setup>` 语法，这需要构建工具。如果你打算在没有构建步骤的情况下使用组合式 API，请参考 `setup()` 选项的用法。

使用 ES 模块构建版本

在本文档的其余部分我们使用的主要是 ES 模块语法。现代浏览器大多都已原生支持 ES 模块。因此我们可以像这样通过 CDN 以及原生 ES 模块使用 Vue：

modules like this:

```html
<div id="app">{{ message }}</div>

<script type="module">
    import { createApp, ref } from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'

    createApp({
    setup() {
        const message = ref('Hello Vue!')
        return {
        message
        }
    }
    }).mount('#app')
</script>
```

Notice that we are using `<script type="module">`, and the imported CDN URL is pointing to the **ES modules build** of Vue instead.

Codepen demo

Enabling Import maps

In the above example, we are importing from the full CDN URL, but in the rest of the documentation you will see code like this:

```js
import { createApp } from 'vue'
```

We can teach the browser where to locate the `vue` import by using Import Maps:

```html
<script type="importmap">
{
    "imports": {
    "vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"
    }
}
</script>
```

```html
<div id="app">{{ message }}</div>

<script type="module">
    import { createApp, ref } from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'

    createApp({
    setup() {
        const message = ref('Hello Vue!')
        return {
        message
        }
    }
    }).mount('#app')
</script>
```

注意我们使用了 `<script type="module">`，且导入的 CDN URL 指向的是 Vue 的 **ES 模块构建版本**。

Codepen 示例

启用 Import maps

在上面的示例中，我们使用了完整的 CDN URL 来导入，但在文档的其余部分中，你将看到如下代码：

```js
import { createApp } from 'vue'
```

我们可以使用导入映射表 (Import Maps) 来告诉浏览器如何定位到导入的 `vue`：

```html
<script type="importmap">
{
    "imports": {
    "vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"
    }
}
</script>
```

```
<div id="app">{{ message }}</div>

<script type="module">
import { createApp, ref } from 'vue'

createApp({
    setup() {
    const message = ref('Hello Vue!')
    return {
        message
    }
    }
}).mount('#app')
</script>
```

Codepen demo

You can also add entries for other dependencies to the import map - but make sure they point to the ES modules version of the library you intend to use.

Import Maps Browser Support
Import Maps is a relatively new browser feature. Make sure to use a browser within its support range. In particular, it is only supported in Safari 16.4+.

Notes on Production Use
The examples so far are using the development build of Vue - if you intend to use Vue from a CDN in production, make sure to check out the Production Deployment Guide.

Splitting Up the Modules

As we dive deeper into the guide, we may need to split our code into separate JavaScript files so that they are easier to manage. For example:

html

```
<!-- index.html -->
<div id="app"></div>
```

```
<div id="app">{{ message }}</div>

<script type="module">
import { createApp, ref } from 'vue'

createApp({
    setup() {
    const message = ref('Hello Vue!')
    return {
        message
    }
    }
}).mount('#app')
</script>
```

Codepen demo

你也可以在映射表中添加其他的依赖——但请务必确保你使用的是该库的 ES 模块版本。

导入映射表的浏览器支持情况
导入映射表是一个相对较新的浏览器功能。请确保使用其支持范围内的浏览器。请注意，只有 Safari 16.4 以上版本支持。

生产环境中的注意事项
到目前为止示例中使用的都是 Vue 的开发构建版本——如果你打算在生产中通过 CDN 使用 Vue，请务必查看生产环境部署指南。

拆分模块

随着对这份指南的逐步深入，我们可能需要将代码分割成单独的 JavaScript 文件，以便更容易管理。例如：

html

```
<!-- index.html -->
<div id="app"></div>
```

```
<script type="module">
    import { createApp } from 'vue'
    import MyComponent from './my-component.js'

    createApp(MyComponent).mount('#app')
</script>
```

js

```js
// my-component.js
import { ref } from 'vue'
export default {
    setup() {
    const count = ref(0)
    return { count }
    },
    template: `<div>count is {{ count }}</div>`
}
```

If you directly open the above `index.html` in your browser, you will find that it throws an error because ES modules cannot work over the `file://` protocol, which is the protocol the browser uses when you open a local file.

Due to security reasons, ES modules can only work over the `http://` protocol, which is what the browsers use when opening pages on the web. In order for ES modules to work on our local machine, we need to serve the `index.html` over the `http://` protocol, with a local HTTP server.

To start a local HTTP server, first make sure you have Node.js installed, then run `npx serve` from the command line in the same directory where your HTML file is. You can also use any other HTTP server that can serve static files with the correct MIME types.

You may have noticed that the imported component's template is inlined as a JavaScript string. If you are using VSCode, you can install the es6-string-html extension and prefix the strings with a `/*html*/` comment to get syntax highlighting for them.

1.2.4 Next Steps

If you skipped the Introduction, we strongly recommend reading it before moving on to the rest of the documentation.

```
<script type="module">
    import { createApp } from 'vue'
    import MyComponent from './my-component.js'

    createApp(MyComponent).mount('#app')
</script>
```

js

```js
// my-component.js
import { ref } from 'vue'
export default {
    setup() {
    const count = ref(0)
    return { count }
    },
    template: `<div>count is {{ count }}</div>`
}
```

如果直接在浏览器中打开了上面的 `index.html`，你会发现它抛出了一个错误，因为 ES 模块不能通过 `file://` 协议工作，也即是当你打开一个本地文件时浏览器使用的协议。

由于安全原因，ES 模块只能通过 `http://` 协议工作，也即是浏览器在打开网页时使用的协议。为了使 ES 模块在我们的本地机器上工作，我们需要使用本地的 HTTP 服务器，通过 `http://` 协议来提供 `index.html`。

要启动一个本地的 HTTP 服务器，请先安装 Node.js，然后通过命令行在 HTML 文件所在文件夹下运行 `npx serve`。你也可以使用其他任何可以基于正确的 MIME 类型服务静态文件的 HTTP 服务器。

可能你也注意到了，这里导入的组件模板是内联的 JavaScript 字符串。如果你正在使用 VSCode，你可以安装 es6-string-html 扩展，然后在字符串前加上一个前缀注释 `/*html*/` 以高亮语法。

1.2.4 下一步

如果你尚未阅读简介，我们强烈推荐你在移步到后续文档之前返回去阅读一下。

Continue with the GuideThe guide walks you through every aspect of the framework in full detail.Try the TutorialFor those who prefer learning things hands-on.Check out the ExamplesExplore examples of core features and common UI tasks.

继续阅读该指南该指南会带你深入了解框架所有方面的细节。尝试互动教程适合喜欢边动手边学的读者。查看示例浏览核心功能和常见用户界面的示例。

第二章　Essentials

2.1　Creating a Vue Application

2.1.1　The application instance

Every Vue application starts by creating a new **application instance** with the `createApp` function:

```js
import { createApp } from 'vue'

const app = createApp({
    /* root component options */
})
```

2.1.2　The Root Component

The object we are passing into `createApp` is in fact a component. Every app requires a ”root component” that can contain other components as its children.

If you are using Single-File Components, we typically import the root component from another file:

```js
import { createApp } from 'vue'
// import the root component App from a single-file component.
import App from './App.vue'

const app = createApp(App)
```

While many examples in this guide only need a single component, most real applications are or-

基础

2.1　创建一个 Vue 应用

2.1.1　应用实例

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

```js
import { createApp } from 'vue'

const app = createApp({
    /* root component options */
})
```

2.1.2　根组件

我们传入 `createApp` 的对象实际上是一个组件，每个应用都需要一个“根组件”，其他组件将作为其子组件。

如果你使用的是单文件组件，我们可以直接从另一个文件中导入根组件。

```js
import { createApp } from 'vue'
// import the root component App from a single-file component.
import App from './App.vue'

const app = createApp(App)
```

虽然本指南中的许多示例只需要一个组件，但大多数真实的应用都是由一棵嵌套

ganized into a tree of nested, reusable components. For example, a Todo application's component tree might look like this:

```
App (root component)
  TodoList
     TodoItem
        TodoDeleteButton
        TodoEditButton
  TodoFooter
      TodoClearButton
      TodoStatistics
```

In later sections of the guide, we will discuss how to define and compose multiple components together. Before that, we will focus on what happens inside a single component.

2.1.3 Mounting the App

An application instance won't render anything until its `.mount()` method is called. It expects a ”container” argument, which can either be an actual DOM element or a selector string:

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

```js
app.mount('#app')
```

The content of the app's root component will be rendered inside the container element. The container element itself is not considered part of the app.

The `.mount()` method should always be called after all app configurations and asset registrations are done. Also note that its return value, unlike the asset registration methods, is the root component instance instead of the application instance.

In-DOM Root Component Template

The template for the root component is usually part of the component itself, but it is also possible to provide the template separately by writing it directly inside the mount container:

```html
<div id="app">
<button @click="count++">{{ count }}</button>
```

的、可重用的组件树组成的。例如，一个待办事项 (Todos) 应用的组件树可能是这样的：

```
App (root component)
  TodoList
     TodoItem
        TodoDeleteButton
        TodoEditButton
  TodoFooter
      TodoClearButton
      TodoStatistics
```

我们会在指南的后续章节中讨论如何定义和组合多个组件。在那之前，我们得先关注一个组件内到底发生了什么。

2.1.3 挂载应用

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

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

```js
app.mount('#app')
```

应用根组件的内容将会被渲染在容器元素里面。容器元素自己将**不会**被视为应用的一部分。

`.mount()` 方法应该始终在整个应用配置和资源注册完成后被调用。同时请注意，不同于其他资源注册方法，它的返回值是根组件实例而非应用实例。

DOM 中的根组件模板

根组件的模板通常是组件本身的一部分，但也可以直接通过在挂载容器内编写模板来单独提供：

```html
<div id="app">
<button @click="count++">{{ count }}</button>
```

```
</div>
```

js

```
import { createApp } from 'vue'

const app = createApp({
    data() {
    return {
        count: 0
    }
    }
})

app.mount('#app')
```

Vue will automatically use the container's `innerHTML` as the template if the root component does not already have a `template` option.

In-DOM templates are often used in applications that are using Vue without a build step. They can also be used in conjunction with server-side frameworks, where the root template might be generated dynamically by the server.

2.1.4 App Configurations

The application instance exposes a `.config` object that allows us to configure a few app-level options, for example, defining an app-level error handler that captures errors from all descendant components:

js

```
app.config.errorHandler = (err) => {
/* handle error */
}
```

The application instance also provides a few methods for registering app-scoped assets. For example, registering a component:

js

```
app.component('TodoDeleteButton', TodoDeleteButton)
```

This makes the `TodoDeleteButton` available for use anywhere in our app. We will discuss registra-

```
</div>
```

js

```
import { createApp } from 'vue'

const app = createApp({
    data() {
    return {
        count: 0
    }
    }
})

app.mount('#app')
```

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

DOM 内模板通常用于无构建步骤的 Vue 应用程序。它们也可以与服务器端框架一起使用，其中根模板可能是由服务器动态生成的。

2.1.4 应用配置

应用实例会暴露一个 `.config` 对象允许我们配置一些应用级的选项，例如定义一个应用级的错误处理器，用来捕获所有子组件上的错误：

js

```
    app.config.errorHandler = (err) => {
/* handle error */
}
```

应用实例还提供了一些方法来注册应用范围内可用的资源，例如注册一个组件：

js

```
app.component('TodoDeleteButton', TodoDeleteButton)
```

这使得 `TodoDeleteButton` 在应用的任何地方都是可用的。我们会在指南的后续

tion for components and other types of assets in later sections of the guide. You can also browse the full list of application instance APIs in its API reference.

Make sure to apply all app configurations before mounting the app!

2.1.5 Multiple application instances

You are not limited to a single application instance on the same page. The `createApp` API allows multiple Vue applications to co-exist on the same page, each with its own scope for configuration and global assets:

js

```js
const app1 = createApp({
    /* ... */
})
app1.mount('#container-1')

const app2 = createApp({
    /* ... */
})
app2.mount('#container-2')
```

If you are using Vue to enhance server-rendered HTML and only need Vue to control specific parts of a large page, avoid mounting a single Vue application instance on the entire page. Instead, create multiple small application instances and mount them on the elements they are responsible for.

2.2 Template Syntax

Vue uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying component instance's data. All Vue templates are syntactically valid HTML that can be parsed by spec-compliant browsers and HTML parsers.

Under the hood, Vue compiles the templates into highly-optimized JavaScript code. Combined with the reactivity system, Vue can intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes.

If you are familiar with Virtual DOM concepts and prefer the raw power of JavaScript, you can also

章节中讨论关于组件和其他资源的注册。你也可以在 API 参考中浏览应用实例 API 的完整列表。

确保在挂载应用实例之前完成所有应用配置！

2.1.5 多个应用实例

应用实例并不只限于一个。`createApp` API 允许你在同一个页面中创建多个共存的 Vue 应用，而且每个应用都拥有自己的用于配置和全局资源的作用域。

js

```js
const app1 = createApp({
    /* ... */
})
app1.mount('#container-1')

const app2 = createApp({
    /* ... */
})
app2.mount('#container-2')
```

如果你正在使用 Vue 来增强服务端渲染 HTML，并且只想要 Vue 去控制一个大型页面中特殊的一小部分，应避免将一个单独的 Vue 应用实例挂载到整个页面上，而是应该创建多个小的应用实例，将它们分别挂载到所需的元素上去。

2.2 模板语法

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

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

如果你对虚拟 DOM 的概念比较熟悉，并且偏好直接使用 JavaScript，你也可以

directly write render functions instead of templates, with optional JSX support. However, do note that they do not enjoy the same level of compile-time optimizations as templates.

2.2.1 Text Interpolation

The most basic form of data binding is text interpolation using the ”Mustache” syntax (double curly braces):

template

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

The mustache tag will be replaced with the value of the `msg` property from the corresponding component instance. It will also be updated whenever the `msg` property changes.

2.2.2 Raw HTML

The double mustaches interpret the data as plain text, not HTML. In order to output real HTML, you will need to use the `v-html` directive:

html

```
<p>Using text interpolation: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

result

Using text interpolation: This should be red.

Using v-html directive: This should be red.

Here we're encountering something new. The `v-html` attribute you're seeing is called a **directive**. Directives are prefixed with `v-` to indicate that they are special attributes provided by Vue, and as you may have guessed, they apply special reactive behavior to the rendered DOM. Here, we're basically saying ”keep this element's inner HTML up-to-date with the `rawHtml` property on the current active instance.”

The contents of the `span` will be replaced with the value of the `rawHtml` property, interpreted as plain HTML - data bindings are ignored. Note that you cannot use `v-html` to compose template partials, because Vue is not a string-based templating engine. Instead, components are preferred as the fundamental unit for UI reuse and composition.

结合可选的 JSX 支持直接手写渲染函数而不采用模板。但请注意，这将不会享受到和模板同等级别的编译时优化。

2.2.1 文本插值

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

template

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

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

2.2.2 原始 HTML

双大括号会将数据解释为纯文本，而不是 HTML。若想插入 HTML，你需要使用 `v-html` 指令：

html

```
<p>Using text interpolation: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

结果

Using text interpolation: This should be red.

Using v-html directive: This should be red.

这里我们遇到了一个新的概念。这里看到的 `v-html` attribute 被称为一个**指令**。指令由 `v-` 作为前缀，表明它们是一些由 Vue 提供的特殊 attribute，你可能已经猜到了，它们将为渲染的 DOM 应用特殊的响应式行为。这里我们做的事情简单来说就是：在当前组件实例上，将此元素的 innerHTML 与 `rawHtml` 属性保持同步。

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

> **Security Warning**
> Dynamically rendering arbitrary HTML on your website can be very dangerous because it can easily lead to XSS vulnerabilities. Only use `v-html` on trusted content and **never** on user-provided content.

2.2.3 Attribute Bindings

Mustaches cannot be used inside HTML attributes. Instead, use a `v-bind` directive:

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

The `v-bind` directive instructs Vue to keep the element's `id` attribute in sync with the component's `dynamicId` property. If the bound value is `null` or `undefined`, then the attribute will be removed from the rendered element.

Shorthand

Because `v-bind` is so commonly used, it has a dedicated shorthand syntax:

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

Attributes that start with `:` may look a bit different from normal HTML, but it is in fact a valid character for attribute names and all Vue-supported browsers can parse it correctly. In addition, they do not appear in the final rendered markup. The shorthand syntax is optional, but you will likely appreciate it when you learn more about its usage later.

> For the rest of the guide, we will be using the shorthand syntax in code examples, as that's the most common usage for Vue developers.

Boolean Attributes

Boolean attributes are attributes that can indicate true / false values by their presence on an element. For example, `disabled` is one of the most commonly used boolean attributes.

`v-bind` works a bit differently in this case:

> **安全警告**
> 在网站上动态渲染任意 HTML 是非常危险的，因为这非常容易造成 XSS 漏洞。请仅在内容安全可信时再使用 `v-html`，并且**永远不要**使用用户提供的 HTML 内容。

2.2.3 Attribute 绑定

双大括号不能在 HTML attributes 中使用。想要响应式地绑定一个 attribute，应该使用 `v-bind` 指令：

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

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

简写

因为 `v-bind` 非常常用，我们提供了特定的简写语法：

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

开头为 `:` 的 attribute 可能和一般的 HTML attribute 看起来不太一样，但它的确是合法的 attribute 名称字符，并且所有支持 Vue 的浏览器都能正确解析它。此外，他们不会出现在最终渲染的 DOM 中。简写语法是可选的，但相信在你了解了它更多的用处后，你应该会更喜欢它。

> 接下来的指引中，我们都将在示例中使用简写语法，因为这是在实际开发中更常见的用法。

布尔型 Attribute

布尔型 attribute 依据 true / false 值来决定 attribute 是否应该存在于该元素上。`disabled` 就是最常见的例子之一。

`v-bind` 在这种场景下的行为略有不同：

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

The `disabled` attribute will be included if `isButtonDisabled` has a truthy value. It will also be included if the value is an empty string, maintaining consistency with `<button disabled="">`. For other falsy values the attribute will be omitted.

Dynamically Binding Multiple Attributes

If you have a JavaScript object representing multiple attributes that looks like this:

```js
const objectOfAttrs = {
  id: 'container',
  class: 'wrapper'
}
```

You can bind them to a single element by using `v-bind` without an argument:

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

2.2.4 Using JavaScript Expressions

So far we've only been binding to simple property keys in our templates. But Vue actually supports the full power of JavaScript expressions inside all data bindings:

```html
{{ number + 1 }}

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

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

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

These expressions will be evaluated as JavaScript in the data scope of the current component instance.

In Vue templates, JavaScript expressions can be used in the following positions:

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

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

动态绑定多个值

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

```js
const objectOfAttrs = {
  id: 'container',
  class: 'wrapper'
}
```

通过不带参数的 `v-bind`，你可以将它们绑定到单个元素上：

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

2.2.4 使用 JavaScript 表达式

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

```html
{{ number + 1 }}

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

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

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

这些表达式都会被作为 JavaScript ，以当前组件实例为作用域解析执行。

在 Vue 模板内，JavaScript 表达式可以被使用在如下场景上：

- Inside text interpolations (mustaches)
- In the attribute value of any Vue directives (special attributes that start with `v-`)

Expressions Only

Each binding can only contain **one single expression**. An expression is a piece of code that can be evaluated to a value. A simple check is whether it can be used after `return`.

Therefore, the following will **NOT** work:

html
```html
<!-- this is a statement, not an expression: -->
{{ var a = 1 }}

<!-- flow control won't work either, use ternary expressions -->
{{ if (ok) { return message } }}
```

Calling Functions

It is possible to call a component-exposed method inside a binding expression:

html
```html
<time :title="toTitleDate(date)" :datetime="date">
{{ formatDate(date) }}
</time>
```

> **TIP**
> Functions called inside binding expressions will be called every time the component updates, so they should **not** have any side effects, such as changing data or triggering asynchronous operations.

Restricted Globals Access

Template expressions are sandboxed and only have access to a restricted list of globals. The list exposes commonly used built-in globals such as `Math` and `Date`.

Globals not explicitly included in the list, for example user-attached properties on `window`, will not

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

仅支持表达式

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

因此，下面的例子都是**无效**的：

html
```html
<!-- 这是一个语句，而非表达式 -->
{{ var a = 1 }}

<!-- 条件控制也不支持，请使用三元表达式 -->
{{ if (ok) { return message } }}
```

调用函数

可以在绑定的表达式中使用一个组件暴露的方法：

html
```html
<time :title="toTitleDate(date)" :datetime="date">
{{ formatDate(date) }}
</time>
```

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

受限的全局访问

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

没有显式包含在列表中的全局对象将不能在模板内表达式中访问，例如用户附加

be accessible in template expressions. You can, however, explicitly define additional globals for all Vue expressions by adding them to `app.config.globalProperties`.

2.2.5 Directives

Directives are special attributes with the `v-` prefix. Vue provides a number of built-in directives, including `v-html` and `v-bind` which we have introduced above.

Directive attribute values are expected to be single JavaScript expressions (with the exception of `v-for`, `v-on` and `v-slot`, which will be discussed in their respective sections later). A directive's job is to reactively apply updates to the DOM when the value of its expression changes. Take `v-if` as an example:

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

Here, the `v-if` directive would remove / insert the `<p>` element based on the truthiness of the value of the expression `seen`.

Arguments

Some directives can take an ”argument”, denoted by a colon after the directive name. For example, the `v-bind` directive is used to reactively update an HTML attribute:

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

<!-- shorthand -->
<a :href="url"> ... </a>
```

Here, `href` is the argument, which tells the `v-bind` directive to bind the element's `href` attribute to the value of the expression `url`. In the shorthand, everything before the argument (i.e., `v-bind:`) is condensed into a single character, `:`.

Another example is the `v-on` directive, which listens to DOM events:

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

<!-- shorthand -->
```

在 `window` 上的属性。然而，你也可以自行在 `app.config.globalProperties` 上显式地添加它们，供所有的 Vue 表达式使用。

2.2.5 指令 Directives

指令是带有 `v-` 前缀的特殊 attribute。Vue 提供了许多内置指令，包括上面我们所介绍的 `v-bind` 和 `v-html`。

指令 attribute 的期望值为一个 JavaScript 表达式 (除了少数几个例外，即之后要讨论到的 `v-for`、`v-on` 和 `v-slot`)。一个指令的任务是在其表达式的值变化时响应式地更新 DOM。以 `v-if` 为例：

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

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

参数 Arguments

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

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

<!-- 简写 -->
<a :href="url"> ... </a>
```

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

另一个例子是 `v-on` 指令，它将监听 DOM 事件：

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

<!-- 简写 -->
```

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

Here, the argument is the event name to listen to: `click`. `v-on` has a corresponding shorthand, namely the `@` character. We will talk about event handling in more detail too.

Dynamic Arguments

It is also possible to use a JavaScript expression in a directive argument by wrapping it with square brackets:

```html
<!--
Note that there are some constraints to the argument expression,
as explained in the "Dynamic Argument Value Constraints"
and "Dynamic Argument Syntax Constraints" sections below.
-->
<a v-bind:[attributeName]="url"> ... </a>

<!-- shorthand -->
<a :[attributeName]="url"> ... </a>
```

Here, `attributeName` will be dynamically evaluated as a JavaScript expression, and its evaluated value will be used as the final value for the argument. For example, if your component instance has a data property, `attributeName`, whose value is `"href"`, then this binding will be equivalent to `v-bind:href`.

Similarly, you can use dynamic arguments to bind a handler to a dynamic event name:

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

<!-- shorthand -->
<a @[eventName]="doSomething">
```

In this example, when `eventName`'s value is `"focus"`, `v-on:[eventName]` will be equivalent to `v-on:focus`.

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

这里的参数是要监听的事件名称：`click`。`v-on` 有一个相应的缩写，即 `@` 字符。我们之后也会讨论关于事件处理的更多细节。

动态参数

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

```html
<!--
注意，参数表达式有一些约束，
参见下面“动态参数值的限制”与“动态参数语法的限制”章节的解释
-->
<a v-bind:[attributeName]="url"> ... </a>

<!-- 简写 -->
<a :[attributeName]="url"> ... </a>
```

这里的 `attributeName` 会作为一个 JavaScript 表达式被动态执行，计算得到的值会被用作最终的参数。举例来说，如果你的组件实例有一个数据属性 `attributeName`，其值为 `"href"`，那么这个绑定就等价于 `v-bind:href`。

相似地，你还可以将一个函数绑定到动态的事件名称上：

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

<!-- 简写 -->
<a @[eventName]="doSomething">
```

在此示例中，当 `eventName` 的值是 `"focus"` 时，`v-on:[eventName]` 就等价于 `v-on:focus`。

Dynamic Argument Value Constraints

Dynamic arguments are expected to evaluate to a string, with the exception of `null`. The special value `null` can be used to explicitly remove the binding. Any other non-string value will trigger a warning.

Dynamic Argument Syntax Constraints

Dynamic argument expressions have some syntax constraints because certain characters, such as spaces and quotes, are invalid inside HTML attribute names. For example, the following is invalid:

```html
<!-- This will trigger a compiler warning. -->
<a :['foo' + bar]="value"> ... </a>
```

If you need to pass a complex dynamic argument, it's probably better to use a computed property, which we will cover shortly.

When using in-DOM templates (templates directly written in an HTML file), you should also avoid naming keys with uppercase characters, as browsers will coerce attribute names into lowercase:

```html
<a :[someAttr]="value"> ... </a>
```

The above will be converted to `:[someattr]` in in-DOM templates. If your component has a `someAttr` property instead of `someattr`, your code won't work. Templates inside Single-File Components are **not** subject to this constraint.

Modifiers

Modifiers are special postfixes denoted by a dot, which indicate that a directive should be bound in some special way. For example, the `.prevent` modifier tells the `v-on` directive to call `event.preventDefault()` on the triggered event:

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

You'll see other examples of modifiers later, for `v-on` and for `v-model`, when we explore those features.

And finally, here's the full directive syntax visualized:

动态参数值的限制

动态参数中表达式的值应当是一个字符串，或者是 `null`。特殊值 `null` 意为显式移除该绑定。其他非字符串的值会触发警告。

动态参数语法的限制

动态参数表达式因为某些字符的缘故有一些语法限制，比如空格和引号，在 HTML attribute 名称中都是不合法的。例如下面的示例：

```html
<!-- 这会触发一个编译器警告 -->
<a :['foo' + bar]="value"> ... </a>
```

如果你需要传入一个复杂的动态参数，我们推荐使用计算属性替换复杂的表达式，也是 Vue 最基础的概念之一，我们很快就会讲到。

当使用 DOM 内嵌模板 (直接写在 HTML 文件里的模板) 时，我们需要避免在名称中使用大写字母，因为浏览器会强制将其转换为小写：

```html
<a :[someAttr]="value"> ... </a>
```

上面的例子将会在 DOM 内嵌模板中被转换为 `:[someattr]`。如果你的组件拥有 "someAttr" 属性而非 "someattr"，这段代码将不会工作。单文件组件内的模板**不**受此限制。

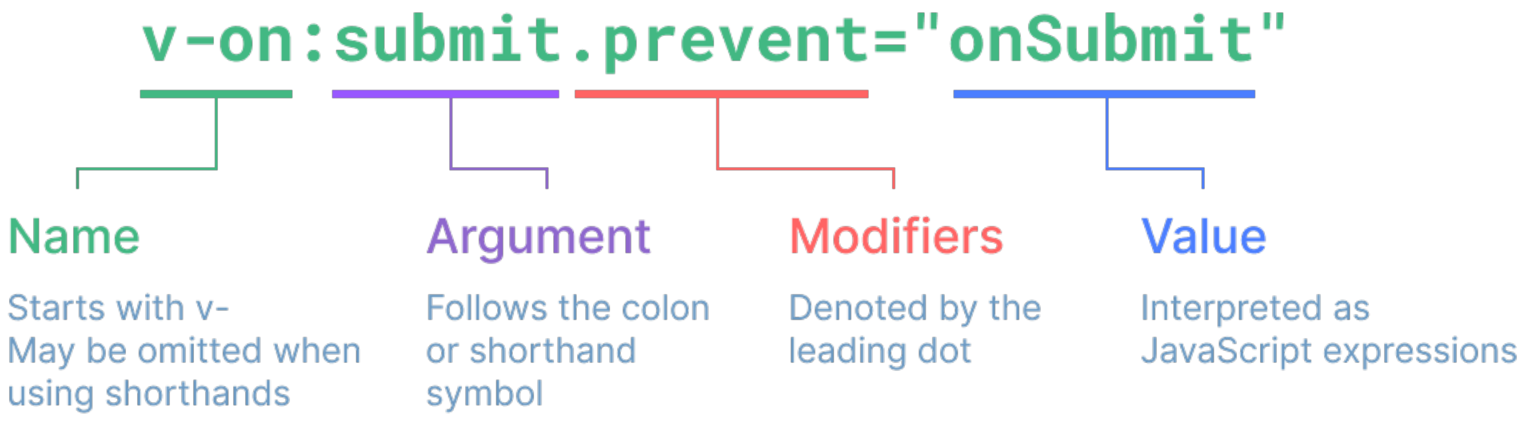
修饰符 Modifiers

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

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

之后在讲到 `v-on` 和 `v-model` 的功能时，你将会看到其他修饰符的例子。

最后，在这里你可以直观地看到完整的指令语法：


v-on:submit.prevent="onSubmit"
Name
Starts with v-
May be omitted when using shorthands
Argument
Follows the colon or shorthand symbol
Modifiers
Denoted by the leading dot
Value
Interpreted as JavaScript expressions

2.3 Reactivity Fundamentals

API Preference
This page and many other chapters later in the guide contain different content for the Options API and the Composition API. Your current preference is Composition API. You can toggle between the API styles using the "API Preference" switches at the top of the left sidebar.

2.3.1 Declaring Reactive State

ref()

In Composition API, the recommended way to declare reactive state is using the `ref()` function:

```js
import { ref } from 'vue'

const count = ref(0)
```

`ref()` takes the argument and returns it wrapped within a ref object with a `.value` property:

2.3 响应式基础

API 参考
本页和后面很多页面中都分别包含了选项式 API 和组合式 API 的示例代码。现在你选择的是组合式 API。你可以使用左侧侧边栏顶部的 “API 风格偏好” 开关在 API 风格之间切换。

2.3.1 声明响应式状态

ref()

在组合式 API 中，推荐使用 `ref()` 函数来声明响应式状态：

```js
import { ref } from 'vue'

const count = ref(0)
```

`ref()` 接收参数，并将其包裹在一个带有 `.value` 属性的 ref 对象中返回：

```js
const count = ref(0)

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

count.value++
console.log(count.value) // 1
```

> See also: Typing Refs

To access refs in a component's template, declare and return them from a component's `setup()` function:

```js
import { ref } from 'vue'

export default {
    // `setup` 是一个特殊的钩子，专门用于组合式 API。
    setup() {
    const count = ref(0)

    // 将 ref 暴露给模板
    return {
        count
    }
    }
}
```

```html
<div>{{ count }}</div>
```

Notice that we did **not** need to append `.value` when using the ref in the template. For convenience, refs are automatically unwrapped when used inside templates (with a few caveats).

You can also mutate a ref directly in event handlers:

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

```js
const count = ref(0)

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

count.value++
console.log(count.value) // 1
```

> 参考：为 refs 标注类型

要在组件模板中访问 ref，请从组件的 `setup()` 函数中声明并返回它们：

```js
import { ref } from 'vue'

export default {
    // `setup` 是一个特殊的钩子，专门用于组合式 API。
    setup() {
    const count = ref(0)

    // 将 ref 暴露给模板
    return {
        count
    }
    }
}
```

```html
<div>{{ count }}</div>
```

注意，在模板中使用 ref 时，我们**不**需要附加 `.value`。为了方便起见，当在模板中使用时，ref 会自动解包 (有一些注意事项)。

你也可以直接在事件监听器中改变一个 ref：

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

For more complex logic, we can declare functions that mutate refs in the same scope and expose them as methods alongside the state:

```js
import { ref } from 'vue'

export default {
    setup() {
    const count = ref(0)

    function increment() {
        // .value is needed in JavaScript
        count.value++
    }

    // don't forget to expose the function as well.
    return {
        count,
        increment
    }
    }
}
```

Exposed methods can then be used as event handlers:

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

Here's the example live on Codepen, without using any build tools.

<script setup>

Manually exposing state and methods via `setup()` can be verbose. Luckily, it can be avoided when using Single-File Components (SFCs). We can simplify the usage with `<script setup>`:

对于更复杂的逻辑，我们可以在同一作用域内声明更改 ref 的函数，并将它们作为方法与状态一起公开：

```js
import { ref } from 'vue'

export default {
    setup() {
    const count = ref(0)

    function increment() {
        // 在 JavaScript 中需要 .value
        count.value++
    }

    // 不要忘记同时暴露 increment 函数
    return {
        count,
        increment
    }
    }
}
```

然后，暴露的方法可以被用作事件监听器：

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

这里是 Codepen 上的例子，没有使用任何构建工具。

<script setup>

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

vue

```vue
<script setup>
import { ref } from 'vue'

const count = ref(0)

function increment() {
    count.value++
}
</script>

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

Try it in the Playground

Top-level imports, variables and functions declared in `<script setup>` are automatically usable in the template of the same component. Think of the template as a JavaScript function declared in the same scope - it naturally has access to everything declared alongside it.

> **TIP**
> For the rest of the guide, we will be primarily using SFC + `<script setup>` syntax for the Composition API code examples, as that is the most common usage for Vue developers.
> If you are not using SFC, you can still use Composition API with the `setup()` option.

Why Refs?

You might be wondering why we need refs with the `.value` instead of plain variables. To explain that, we will need to briefly discuss how Vue's reactivity system works.

When you use a ref in a template, and change the ref's value later, Vue automatically detects the change and updates the DOM accordingly. This is made possible with a dependency-tracking based

vue

```vue
<script setup>
import { ref } from 'vue'

const count = ref(0)

function increment() {
    count.value++
}
</script>

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

在演练场中尝试一下

`<script setup>` 中的顶层的导入、声明的变量和函数可在同一组件的模板中直接使用。你可以理解为模板是在同一作用域内声明的一个 JavaScript 函数——它自然可以访问与它一起声明的所有内容。

> **TIP**
> 在指南的后续章节中，我们基本上都会在组合式 API 示例中使用单文件组件 + `<script setup>` 的语法，因为大多数 Vue 开发者都会这样使用。
> 如果你没有使用单文件组件，你仍然可以在 `setup()` 选项中使用组合式 API。

为什么要使用 ref?

你可能会好奇：为什么我们需要使用带有 `.value` 的 ref，而不是普通的变量？为了解释这一点，我们需要简单地讨论一下 Vue 的响应式系统是如何工作的。

当你在模板中使用了一个 ref，然后改变了这个 ref 的值时，Vue 会自动检测到这个变化，并且相应地更新 DOM。这是通过一个基于依赖追踪的响应式系统实现的。

reactivity system. When a component is rendered for the first time, Vue **tracks** every ref that was used during the render. Later on, when a ref is mutated, it will **trigger** a re-render for components that are tracking it.

In standard JavaScript, there is no way to detect the access or mutation of plain variables. However, we can intercept the get and set operations of an object's properties using getter and setter methods.

The `.value` property gives Vue the opportunity to detect when a ref has been accessed or mutated. Under the hood, Vue performs the tracking in its getter, and performs triggering in its setter. Conceptually, you can think of a ref as an object that looks like this:

```js
// pseudo code, not actual implementation
const myRef = {
    _value: 0,
    get value() {
    track()
    return this._value
    },
    set value(newValue) {
    this._value = newValue
    trigger()
    }
}
```

Another nice trait of refs is that unlike plain variables, you can pass refs into functions while retaining access to the latest value and the reactivity connection. This is particularly useful when refactoring complex logic into reusable code.

The reactivity system is discussed in more details in the Reactivity in Depth section.

Deep Reactivity

Refs can hold any value type, including deeply nested objects, arrays, or JavaScript built-in data structures like `Map`.

A ref will make its value deeply reactive. This means you can expect changes to be detected even when you mutate nested objects or arrays:

当一个组件首次渲染时，Vue 会**追踪**在渲染过程中使用的每一个 ref。然后，当一个 ref 被修改时，它会**触发**追踪它的组件的一次重新渲染。

在标准的 JavaScript 中，检测普通变量的访问或修改是行不通的。然而，我们可以通过 getter 和 setter 方法来拦截对象属性的 get 和 set 操作。

该 `.value` 属性给予了 Vue 一个机会来检测 ref 何时被访问或修改。在其内部，Vue 在它的 getter 中执行追踪，在它的 setter 中执行触发。从概念上讲，你可以将 ref 看作是一个像这样的对象：

```js
// 伪代码，不是真正的实现
const myRef = {
    _value: 0,
    get value() {
    track()
    return this._value
    },
    set value(newValue) {
    this._value = newValue
    trigger()
    }
}
```

另一个 ref 的好处是，与普通变量不同，你可以将 ref 传递给函数，同时保留对最新值和响应式连接的访问。当将复杂的逻辑重构为可重用的代码时，这将非常有用。

该响应性系统在深入响应式原理章节中有更详细的讨论。

深层响应性

Ref 可以持有任何类型的值，包括深层嵌套的对象、数组或者 JavaScript 内置的数据结构，比如 `Map`。

Ref 会使它的值具有深层响应性。这意味着即使改变嵌套对象或数组时，变化也会被检测到：

```html
import { ref } from 'vue'

const obj = ref({
    nested: { count: 0 },
    arr: ['foo', 'bar']
})

function mutateDeeply() {
    // these will work as expected.
    obj.value.nested.count++
    obj.value.arr.push('baz')
}
```

Non-primitive values are turned into reactive proxies via `reactive()`, which is discussed below.

It is also possible to opt-out of deep reactivity with shallow refs. For shallow refs, only `.value` access is tracked for reactivity. Shallow refs can be used for optimizing performance by avoiding the observation cost of large objects, or in cases where the inner state is managed by an external library.

Further reading:

- Reduce Reactivity Overhead for Large Immutable Structures
- Integration with External State Systems

DOM Update Timing

When you mutate reactive state, the DOM is updated automatically. However, it should be noted that the DOM updates are not applied synchronously. Instead, Vue buffers them until the ”next tick” in the update cycle to ensure that each component updates only once no matter how many state changes you have made.

To wait for the DOM update to complete after a state change, you can use the nextTick() global API:

```js
import { nextTick } from 'vue'
```

```html
import { ref } from 'vue'

const obj = ref({
    nested: { count: 0 },
    arr: ['foo', 'bar']
})

function mutateDeeply() {
    // 以下都会按照期望工作
    obj.value.nested.count++
    obj.value.arr.push('baz')
}
```

非原始值将通过 `reactive()` 转换为响应式代理，该函数将在后面讨论。

也可以通过 shallow ref 来放弃深层响应性。对于浅层 ref，只有 `.value` 的访问会被追踪。浅层 ref 可以用于避免对大型数据的响应性开销来优化性能、或者有外部库管理其内部状态的情况。

阅读更多：

- 减少大型不可变数据的响应性开销
- 与外部状态系统集成

DOM 更新时机

当你修改了响应式状态时，DOM 会被自动更新。但是需要注意的是，DOM 更新不是同步的。Vue 会在 “next tick” 更新周期中缓冲所有状态的修改，以确保不管你进行了多少次状态修改，每个组件都只会被更新一次。

要等待 DOM 更新完成后再执行额外的代码，可以使用 nextTick() 全局 API：

```js
import { nextTick } from 'vue'
```

```
async function increment() {
    count.value++
    await nextTick()
    // Now the DOM is updated
}
```

2.3.2 reactive()

There is another way to declare reactive state, with the `reactive()` API. Unlike a ref which wraps the inner value in a special object, `reactive()` makes an object itself reactive:

```js
import { reactive } from 'vue'

const state = reactive({ count: 0 })
```

2.3.3 reactive()

There is another way to declare reactive state, with the `reactive()` API. Unlike a ref which wraps the inner value in a special object, `reactive()` makes an object itself reactive:

```js
import { reactive } from 'vue'

const state = reactive({ count: 0 })
```

> See also: Typing Reactive

Usage in template:

```html
<button @click="state.count++">
{{ state.count }}
</button>
```

Reactive objects are JavaScript Proxies and behave just like normal objects. The difference is that Vue is able to intercept the access and mutation of all properties of a reactive object for reactivity tracking and triggering.

```
async function increment() {
    count.value++
    await nextTick()
    // 现在 DOM 已经更新了
}
```

2.3.2 reactive()

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

```js
import { reactive } from 'vue'

const state = reactive({ count: 0 })
```

2.3.3 reactive()

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

```js
import { reactive } from 'vue'

const state = reactive({ count: 0 })
```

> 参考：为 `reactive()` 标注类型

在模板中使用：

```html
<button @click="state.count++">
{{ state.count }}
</button>
```

响应式对象是 JavaScript 代理，其行为就和普通对象一样。不同的是，Vue 能够拦截对响应式对象所有属性的访问和修改，以便进行依赖追踪和触发更新。

`reactive()` converts the object deeply: nested objects are also wrapped with `reactive()` when accessed. It is also called by `ref()` internally when the ref value is an object. Similar to shallow refs, there is also the `shallowReactive()` API for opting-out of deep reactivity.

Reactive Proxy vs. Original

It is important to note that the returned value from `reactive()` is a Proxy of the original object, which is not equal to the original object:

js

```js
const raw = {}
const proxy = reactive(raw)

// proxy is NOT equal to the original.
console.log(proxy === raw) // false
```

Only the proxy is reactive - mutating the original object will not trigger updates. Therefore, the best practice when working with Vue's reactivity system is to **exclusively use the proxied versions of your state**.

To ensure consistent access to the proxy, calling `reactive()` on the same object always returns the same proxy, and calling `reactive()` on an existing proxy also returns that same proxy:

js

```js
// calling reactive() on the same object returns the same proxy
console.log(reactive(raw) === proxy) // true

// calling reactive() on a proxy returns itself
console.log(reactive(proxy) === proxy) // true
```

This rule applies to nested objects as well. Due to deep reactivity, nested objects inside a reactive object are also proxies:

js

```js
const proxy = reactive({})

const raw = {}
proxy.nested = raw

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

`reactive()` 将深层地转换对象：当访问嵌套对象时，它们也会被 `reactive()` 包装。当 ref 的值是一个对象时，`ref()` 也会在内部调用它。与浅层 ref 类似，这里也有一个 `shallowReactive()` API 可以选择退出深层响应性。

Reactive Proxy vs. Original

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

js

```js
const raw = {}
const proxy = reactive(raw)

// 代理对象和原始对象不是全等的
console.log(proxy === raw) // false
```

只有代理对象是响应式的，更改原始对象不会触发更新。因此，使用 Vue 的响应式系统的最佳实践是 **仅使用你声明对象的代理版本**。

为保证访问代理的一致性，对同一个原始对象调用 `reactive()` 会总是返回同样的代理对象，而对一个已存在的代理对象调用 `reactive()` 会返回其本身：

js

```js
// 在同一个对象上调用 reactive() 会返回相同的代理
console.log(reactive(raw) === proxy) // true

// 在一个代理上调用 reactive() 会返回它自己
console.log(reactive(proxy) === proxy) // true
```

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

js

```js
const proxy = reactive({})

const raw = {}
proxy.nested = raw

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

Limitations of reactive()

The `reactive()` API has a few limitations:

1. **Limited value types:** it only works for object types (objects, arrays, and collection types such as `Map` and `Set`). It cannot hold primitive types such as `string`, `number` or `boolean`.

2. **Cannot replace entire object:** since Vue's reactivity tracking works over property access, we must always keep the same reference to the reactive object. This means we can't easily ”replace” a reactive object because the reactivity connection to the first reference is lost:

```js
let state = reactive({ count: 0 })

// the above reference ({ count: 0 }) is no longer being tracked
// (reactivity connection is lost!)
state = reactive({ count: 1 })
```

3. **Not destructure-friendly:** when we destructure a reactive object's primitive type property into local variables, or when we pass that property into a function, we will lose the reactivity connection:

```js
const state = reactive({ count: 0 })

// count is disconnected from state.count when destructured.
let { count } = state
// does not affect original state
count++

// the function receives a plain number and
// won't be able to track changes to state.count
// we have to pass the entire object in to retain reactivity
callSomeFunction(state.count)
```

Due to these limitations, we recommend using `ref()` as the primary API for declaring reactive state.

2.3.4 Additional Ref Unwrapping Details

reactive() 的局限性

`reactive()` API 有一些局限性：

1. **有限的值类型**：它只能用于对象类型 (对象、数组和如 `Map`、`Set` 这样的集合类型)。它不能持有如 `string`、`number` 或 `boolean` 这样的原始类型。

2. **不能替换整个对象**：由于 Vue 的响应式跟踪是通过属性访问实现的，因此我们必须始终保持对响应式对象的相同引用。这意味着我们不能轻易地 “替换” 响应式对象，因为这样的话与第一个引用的响应性连接将丢失：

```js
let state = reactive({ count: 0 })

// 上面的 ({ count: 0 }) 引用将不再被追踪
// (响应性连接已丢失！)
state = reactive({ count: 1 })
```

3. **对解构操作不友好**：当我们将响应式对象的原始类型属性解构为本地变量时，或者将该属性传递给函数时，我们将丢失响应性连接：

```js
const state = reactive({ count: 0 })

// 当解构时，count 已经与 state.count 断开连接
let { count } = state
// 不会影响原始的 state
count++

// 该函数接收到的是一个普通的数字
// 并且无法追踪 state.count 的变化
// 我们必须传入整个对象以保持响应性
callSomeFunction(state.count)
```

由于这些限制，我们建议使用 `ref()` 作为声明响应式状态的主要 API。

2.3.4 额外的 ref 解包细节

As Reactive Object Property

A ref is automatically unwrapped when accessed or mutated as a property of a reactive object. In other words, it behaves like a normal property :

```js
const count = ref(0)
const state = reactive({
    count
})

console.log(state.count) // 0

state.count = 1
console.log(count.value) // 1
```

If a new ref is assigned to a property linked to an existing ref, it will replace the old ref:

```js
const otherCount = ref(2)

state.count = otherCount
console.log(state.count) // 2
// original ref is now disconnected from state.count
console.log(count.value) // 1
```

Ref unwrapping only happens when nested inside a deep reactive object. It does not apply when it is accessed as a property of a shallow reactive object.

Caveat in Arrays and Collections

Unlike reactive objects, there is **no** unwrapping performed when the ref is accessed as an element of a reactive array or a native collection type like `Map`:

```js
const books = reactive([ref('Vue 3 Guide')])
// need .value here
console.log(books[0].value)

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

作为 reactive 对象的属性

一个 ref 会在作为响应式对象的属性被访问或修改时自动解包。换句话说，它的行为就像一个普通的属性：

```js
const count = ref(0)
const state = reactive({
    count
})

console.log(state.count) // 0

state.count = 1
console.log(count.value) // 1
```

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

```js
const otherCount = ref(2)

state.count = otherCount
console.log(state.count) // 2
// 原始 ref 现在已经和 state.count 失去联系
console.log(count.value) // 1
```

只有当嵌套在一个深层响应式对象内时，才会发生 ref 解包。当其作为浅层响应式对象的属性被访问时不会解包。

数组和集合的注意事项

与 reactive 对象不同的是，当 ref 作为响应式数组或原生集合类型 (如 `Map`) 中的元素被访问时，它**不会**被解包：

```js
const books = reactive([ref('Vue 3 Guide')])
// 这里需要 .value
console.log(books[0].value)

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

```js
// need .value here
console.log(map.get('count').value)
```

Caveat when Unwrapping in Templates

Ref unwrapping in templates only applies if the ref is a top-level property in the template render context.

In the example below, `count` and `object` are top-level properties, but `object.id` is not:

```js
const count = ref(0)
const object = { id: ref(1) }
```

Therefore, this expression works as expected:

```html
{{ count + 1 }}
```

...while this one does **NOT**:

```html
{{ object.id + 1 }}
```

The rendered result will be `[object Object]1` because `object.id` is not unwrapped when evaluating the expression and remains a ref object. To fix this, we can destructure `id` into a top-level property:

```js
const { id } = object
```

```html
{{ id + 1 }}
```

Now the render result will be `2`.

Another thing to note is that a ref does get unwrapped if it is the final evaluated value of a text interpolation (i.e. a `{{ }}` tag), so the following will render `1`:

```html
{{ object.id }}
```

This is just a convenience feature of text interpolation and is equivalent to `{{ object.id.value }}`.

```js
// 这里需要 .value
console.log(map.get('count').value)
```

在模板中解包的注意事项

在模板渲染上下文中，只有顶级的 ref 属性才会被解包。

在下面的例子中，`count` 和 `object` 是顶级属性，但 `object.id` 不是：

```js
const count = ref(0)
const object = { id: ref(1) }
```

因此，这个表达式按预期工作：

```html
{{ count + 1 }}
```

... 但这个**不会**：

```html
{{ object.id + 1 }}
```

渲染的结果将是 `[object Object]1`，因为在计算表达式时 `object.id` 没有被解包，仍然是一个 ref 对象。为了解决这个问题，我们可以将 `id` 解构为一个顶级属性：

```js
const { id } = object
```

```html
{{ id + 1 }}
```

现在渲染的结果将是 `2`。

另一个需要注意的点是，如果 ref 是文本插值的最终计算值 (即 `{{ }}` 标签)，那么它将被解包，因此以下内容将渲染为 `1`：

```html
{{ object.id }}
```

该特性仅仅是文本插值的一个便利特性，等价于 `{{ object.id.value }}`。

2.4 Computed Properties

2.4.1 Basic Example

In-template expressions are very convenient, but they are meant for simple operations. Putting too much logic in your templates can make them bloated and hard to maintain. For example, if we have an object with a nested array:

```js
const author = reactive({
    name: 'John Doe',
    books: [
        'Vue 2 - Advanced Guide',
        'Vue 3 - Basic Guide',
        'Vue 4 - The Mystery'
    ]
})
```

And we want to display different messages depending on if `author` already has some books or not:

```html
<p>Has published books:</p>
<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
```

At this point, the template is getting a bit cluttered. We have to look at it for a second before realizing that it performs a calculation depending on `author.books`. More importantly, we probably don't want to repeat ourselves if we need to include this calculation in the template more than once.

That's why for complex logic that includes reactive data, it is recommended to use a **computed property**. Here's the same example, refactored:

```html
<script setup>
import { reactive, computed } from 'vue'

const author = reactive({
    name: 'John Doe',
    books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
```

2.4 计算属性

2.4.1 基础示例

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

```js
const author = reactive({
    name: 'John Doe',
    books: [
        'Vue 2 - Advanced Guide',
        'Vue 3 - Basic Guide',
        'Vue 4 - The Mystery'
    ]
})
```

我们想根据 `author` 是否已有一些书籍来展示不同的信息：

```html
<p>Has published books:</p>
<span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
```

这里的模板看起来有些复杂。我们必须认真看好一会儿才能明白它的计算依赖于 `author.books`。更重要的是，如果在模板中需要不止一次这样的计算，我们可不想将这样的代码在模板里重复好多遍。

因此我们推荐使用**计算属性**来描述依赖响应式状态的复杂逻辑。这是重构后的示例：

```html
<script setup>
import { reactive, computed } from 'vue'

const author = reactive({
    name: 'John Doe',
    books: [
    'Vue 2 - Advanced Guide',
    'Vue 3 - Basic Guide',
    'Vue 4 - The Mystery'
    ]
})

// 一个计算属性 ref
const publishedBooksMessage = computed(() => {
    return author.books.length > 0 ? 'Yes' : 'No'
})
</script>

<template>
    <p>Has published books:</p>
    <span>{{ publishedBooksMessage }}</span>
</template>
```

Try it in the Playground

Here we have declared a computed property `publishedBooksMessage`. The `computed()` function expects to be passed a getter function, and the returned value is a **computed ref**. Similar to normal refs, you can access the computed result as `publishedBooksMessage.value`. Computed refs are also auto-unwrapped in templates so you can reference them without `.value` in template expressions.

A computed property automatically tracks its reactive dependencies. Vue is aware that the computation of `publishedBooksMessage` depends on `author.books`, so it will update any bindings that depend on `publishedBooksMessage` when `author.books` changes.

See also: Typing Computed

2.4.2 Computed Caching vs. Methods

You may have noticed we can achieve the same result by invoking a method in the expression:

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

```
    'Vue 4 - The Mystery'
    ]
})

// 一个计算属性 ref
const publishedBooksMessage = computed(() => {
    return author.books.length > 0 ? 'Yes' : 'No'
})
</script>

<template>
    <p>Has published books:</p>
    <span>{{ publishedBooksMessage }}</span>
</template>
```

在演练场中尝试一下

我们在这里定义了一个计算属性 `publishedBooksMessage`。`computed()` 方法期望接收一个 getter 函数，返回值为一个**计算属性 ref**。和其他一般的 ref 类似，你可以通过 `publishedBooksMessage.value` 访问计算结果。计算属性 ref 也会在模板中自动解包，因此在模板表达式中引用时无需添加 `.value`。

Vue 的计算属性会自动追踪响应式依赖。它会检测到 `publishedBooksMessage` 依赖于 `author.books`，所以任何依赖于 `publishedBooksMessage` 的绑定，都会在 `author.books` 改变时同时更新。

也可参考：为计算属性标注类型

2.4.2 计算属性缓存 vs 方法

你可能注意到我们在表达式中像这样调用一个函数也会获得和计算属性相同的结果：

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

```js
// in component
function calculateBooksMessage() {
    return author.books.length > 0 ? 'Yes' : 'No'
}
```

Instead of a computed property, we can define the same function as a method. For the end result, the two approaches are indeed exactly the same. However, the difference is that **computed properties are cached based on their reactive dependencies.** A computed property will only re-evaluate when some of its reactive dependencies have changed. This means as long as `author.books` has not changed, multiple access to `publishedBooksMessage` will immediately return the previously computed result without having to run the getter function again.

This also means the following computed property will never update, because `Date.now()` is not a reactive dependency:

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

In comparison, a method invocation will **always** run the function whenever a re-render happens.

Why do we need caching? Imagine we have an expensive computed property `list`, which requires looping through a huge array and doing a lot of computations. Then we may have other computed properties that in turn depend on `list`. Without caching, we would be executing `list`'s getter many more times than necessary! In cases where you do not want caching, use a method call instead.

2.4.3 Writable Computed

Computed properties are by default getter-only. If you attempt to assign a new value to a computed property, you will receive a runtime warning. In the rare cases where you need a "writable" computed property, you can create one by providing both a getter and a setter:

```html
<script setup>
import { ref, computed } from 'vue'

const firstName = ref('John')
const lastName = ref('Doe')
```

```js
// 组件中
function calculateBooksMessage() {
    return author.books.length > 0 ? 'Yes' : 'No'
}
```

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

这也解释了为什么下面的计算属性永远不会更新，因为 `Date.now()` 并不是一个响应式依赖：

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

相比之下，方法调用**总是**会在重渲染发生时再次执行函数。

为什么需要缓存呢？想象一下我们有一个非常耗性能的计算属性 `list`，需要循环一个巨大的数组并做许多计算逻辑，并且可能也有其他计算属性依赖于 `list`。没有缓存的话，我们会重复执行非常多次 `list` 的 getter，然而这实际上没有必要！如果你确定不需要缓存，那么也可以使用方法调用。

2.4.3 可写计算属性

计算属性默认是只读的。当你尝试修改一个计算属性时，你会收到一个运行时警告。只在某些特殊场景中你可能才需要用到"可写"的属性，你可以通过同时提供 getter 和 setter 来创建：

```html
<script setup>
import { ref, computed } from 'vue'

const firstName = ref('John')
const lastName = ref('Doe')
const fullName = computed({
    // getter
    get() {
    return firstName.value + ' ' + lastName.value
    },
    // setter
    set(newValue) {
    // Note: we are using destructuring assignment syntax here.
    [firstName.value, lastName.value] = newValue.split(' ')
    }
})
</script>
```

Now when you run `fullName.value = 'John Doe'`, the setter will be invoked and `firstName` and `lastName` will be updated accordingly.

2.4.4 Best Practices

Getters should be side-effect free

It is important to remember that computed getter functions should only perform pure computation and be free of side effects. For example, **don't make async requests or mutate the DOM inside a computed getter!** Think of a computed property as declaratively describing how to derive a value based on other values - its only responsibility should be computing and returning that value. Later in the guide we will discuss how we can perform side effects in reaction to state changes with watchers.

Avoid mutating computed value

The returned value from a computed property is derived state. Think of it as a temporary snapshot - every time the source state changes, a new snapshot is created. It does not make sense to mutate a snapshot, so a computed return value should be treated as read-only and never be mutated - instead, update the source state it depends on to trigger new computations.

```
const fullName = computed({
    // getter
    get() {
    return firstName.value + ' ' + lastName.value
    },
    // setter
    set(newValue) {
    // 注意：我们这里使用的是解构赋值语法
    [firstName.value, lastName.value] = newValue.split(' ')
    }
})
</script>
```

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

2.4.4 最佳实践

Getter 不应有副作用

计算属性的 getter 应只做计算而没有任何其他的副作用，这一点非常重要，请务必牢记。举例来说，**不要在 getter 中做异步请求或者更改 DOM！** 一个计算属性的声明中描述的是如何根据其他值派生一个值。因此 getter 的职责应该仅为计算和返回该值。在之后的指引中我们会讨论如何使用侦听器根据其他响应式状态的变更来创建副作用。

避免直接修改计算属性值

从计算属性返回的值是派生状态。可以把它看作是一个"临时快照"，每当源状态发生变化时，就会创建一个新的快照。更改快照是没有意义的，因此计算属性的返回值应该被视为只读的，并且永远不应该被更改——应该更新它所依赖的源状态以触发新的计算。

2.5 Class and Style Bindings

A common need for data binding is manipulating an element's class list and inline styles. Since `class` and `style` are both attributes, we can use `v-bind` to assign them a string value dynamically, much like with other attributes. However, trying to generate those values using string concatenation can be annoying and error-prone. For this reason, Vue provides special enhancements when `v-bind` is used with `class` and `style`. In addition to strings, the expressions can also evaluate to objects or arrays.

2.5.1 Binding HTML Classes

Binding to Objects

We can pass an object to `:class` (short for `v-bind:class`) to dynamically toggle classes:

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

The above syntax means the presence of the `active` class will be determined by the truthiness of the data property `isActive`.

You can have multiple classes toggled by having more fields in the object. In addition, the `:class` directive can also co-exist with the plain `class` attribute. So given the following state:

```js
const isActive = ref(true)
const hasError = ref(false)
```

And the following template:

```html
<div
  class="static"
  :class="{ active: isActive, 'text-danger': hasError }"
></div>
```

It will render:

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

When `isActive` or `hasError` changes, the class list will be updated accordingly. For example, if

2.5 Class 与 Style 绑定

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

2.5.1 绑定 HTML class

绑定对象

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

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

上面的语法表示 `active` 是否存在取决于数据属性 `isActive` 的真假值。

你可以在对象中写多个字段来操作多个 class。此外，`:class` 指令也可以和一般的 `class` attribute 共存。举例来说，下面这样的状态：

```js
const isActive = ref(true)
const hasError = ref(false)
```

配合以下模板：

```html
<div
  class="static"
  :class="{ active: isActive, 'text-danger': hasError }"
></div>
```

渲染的结果会是：

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

当 `isActive` 或者 `hasError` 改变时，class 列表会随之更新。举例来说，如果

`hasError` becomes `true`, the class list will become `"static active text-danger"`.

The bound object doesn't have to be inline:

```js
const classObject = reactive({
  active: true,
  'text-danger': false
})
```

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

This will render:

```html
<div class="active"></div>
```

We can also bind to a computed property that returns an object. This is a common and powerful pattern:

```js
const isActive = ref(true)
const error = ref(null)

const classObject = computed(() => ({
    active: isActive.value && !error.value,
    'text-danger': error.value && error.value.type === 'fatal'
}))
```

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

Binding to Arrays

We can bind `:class` to an array to apply a list of classes:

```js
const activeClass = ref('active')
const errorClass = ref('text-danger')
```

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

Which will render:

`hasError` 变为 `true`，class 列表也会变成 `"static active text-danger"`。

绑定的对象并不一定需要写成内联字面量的形式，也可以直接绑定一个对象：

```js
const classObject = reactive({
  active: true,
  'text-danger': false
})
```

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

这将渲染：

```html
<div class="active"></div>
```

我们也可以绑定一个返回对象的计算属性。这是一个常见且很有用的技巧：

```js
const isActive = ref(true)
const error = ref(null)

const classObject = computed(() => ({
    active: isActive.value && !error.value,
    'text-danger': error.value && error.value.type === 'fatal'
}))
```

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

绑定数组

我们可以给 `:class` 绑定一个数组来渲染多个 CSS class：

```js
const activeClass = ref('active')
const errorClass = ref('text-danger')
```

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

渲染的结果是：

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

If you would like to also toggle a class in the list conditionally, you can do it with a ternary expression:

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

This will always apply `errorClass`, but `activeClass` will only be applied when `isActive` is truthy.

However, this can be a bit verbose if you have multiple conditional classes. That's why it's also possible to use the object syntax inside the array syntax:

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

With Components

> This section assumes knowledge of Components. Feel free to skip it and come back later.

When you use the `class` attribute on a component with a single root element, those classes will be added to the component's root element and merged with any existing class already on it.

For example, if we have a component named `MyComponent` with the following template:

```html
<!-- child component template -->
<p class="foo bar">Hi!</p>
```

Then add some classes when using it:

```html
<!-- when using the component -->
<MyComponent class="baz boo" />
```

The rendered HTML will be:

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

The same is true for class bindings:

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

如果你也想在数组中有条件地渲染某个 class，你可以使用三元表达式：

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

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

然而，这可能在有多个依赖条件的 class 时会有些冗长。因此也可以在数组中嵌套对象：

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

在组件上使用

> 本节假设你已经有 Vue 组件的知识基础。如果没有，你也可以暂时跳过，以后再阅读。

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

举例来说，如果你声明了一个组件名叫 `MyComponent`，模板如下：

```html
<!-- 子组件模板 -->
<p class="foo bar">Hi!</p>
```

在使用时添加一些 class：

```html
<!-- 在使用组件时 -->
<MyComponent class="baz boo" />
```

渲染出的 HTML 为：

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

Class 的绑定也是同样的：

```html
<MyComponent :class="{ active: isActive }" />
```

When `isActive` is truthy, the rendered HTML will be:

```html
<p class="foo bar active">Hi!</p>
```

If your component has multiple root elements, you would need to define which element will receive this class. You can do this using the `$attrs` component property:

```html
<!-- MyComponent template using $attrs -->
<p :class="$attrs.class">Hi!</p>
<span>This is a child component</span>
```

```html
<MyComponent class="baz" />
```

Will render:

```html
<p class="baz">Hi!</p>
<span>This is a child component</span>
```

You can learn more about component attribute inheritance in Fallthrough Attributes section.

2.5.2 Binding Inline Styles

Binding to Objects

`:style` supports binding to JavaScript object values - it corresponds to an HTML element's `style` property:

```js
const activeColor = ref('red')
const fontSize = ref(30)
```

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

Although camelCase keys are recommended, `:style` also supports kebab-cased CSS property keys (corresponds to how they are used in actual CSS) - for example:

```html
<MyComponent :class="{ active: isActive }" />
```

当 `isActive` 为真时，被渲染的 HTML 会是：

```html
<p class="foo bar active">Hi!</p>
```

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

```html
<!-- MyComponent 模板使用 $attrs 时 -->
<p :class="$attrs.class">Hi!</p>
<span>This is a child component</span>
```

```html
<MyComponent class="baz" />
```

这将被渲染为：

```html
<p class="baz">Hi!</p>
<span>This is a child component</span>
```

你可以在透传 Attribute 一章中了解更多组件的 attribute 继承的细节。

2.5.2 绑定内联样式

绑定对象

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

```js
const activeColor = ref('red')
const fontSize = ref(30)
```

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

尽管推荐使用 camelCase，但 `:style` 也支持 kebab-cased 形式的 CSS 属性 key (对应其 CSS 中的实际名称)，例如：

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

It is often a good idea to bind to a style object directly so that the template is cleaner:

```js
const styleObject = reactive({
    color: 'red',
    fontSize: '13px'
})
```

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

Again, object style binding is often used in conjunction with computed properties that return objects.

Binding to Arrays

We can bind `:style` to an array of multiple style objects. These objects will be merged and applied to the same element:

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

Auto-prefixing

When you use a CSS property that requires a vendor prefix in `:style`, Vue will automatically add the appropriate prefix. Vue does this by checking at runtime to see which style properties are supported in the current browser. If the browser doesn't support a particular property then various prefixed variants will be tested to try to find one that is supported.

Multiple Values

You can provide an array of multiple (prefixed) values to a style property, for example:

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

This will only render the last value in the array which the browser supports. In this example, it

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

直接绑定一个样式对象通常是一个好主意，这样可以使模板更加简洁：

```js
const styleObject = reactive({
    color: 'red',
    fontSize: '13px'
})
```

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

同样的，如果样式对象需要更复杂的逻辑，也可以使用返回样式对象的计算属性。

绑定数组

我们还可以给 `:style` 绑定一个包含多个样式对象的数组。这些对象会被合并后渲染到同一元素上：

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

自动前缀

当你在 `:style` 中使用了需要浏览器特殊前缀的 CSS 属性时，Vue 会自动为他们加上相应的前缀。Vue 是在运行时检查该属性是否支持在当前浏览器中使用。如果浏览器不支持某个属性，那么将尝试加上各个浏览器特殊前缀，以找到哪一个是被支持的。

样式多值

你可以对一个样式属性提供多个 (不同前缀的) 值，举例来说：

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

数组仅会渲染浏览器支持的最后一个值。在这个示例中，在支持不需要特别前缀

will render `display: flex` for browsers that support the unprefixed version of flexbox.

2.6 Conditional Rendering

2.6.1 v-if

The directive `v-if` is used to conditionally render a block. The block will only be rendered if the directive's expression returns a truthy value.

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

2.6.2 v-else

You can use the `v-else` directive to indicate an "else block" for `v-if`:

```html
<button @click="awesome = !awesome">Toggle</button>
<h1 v-if="awesome">Vue is awesome!</h1>
<h1 v-else>Oh no  </h1>
```

Try it in the Playground

A `v-else` element must immediately follow a `v-if` or a `v-else-if` element - otherwise it will not be recognized.

2.6.3 v-else-if

The `v-else-if`, as the name suggests, serves as an "else if block" for `v-if`. It can also be chained multiple times:

```html
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
```

的浏览器中都会渲染为 `display: flex`。

2.6 条件渲染

2.6.1 v-if

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

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

2.6.2 v-else

你也可以使用 `v-else` 为 `v-if` 添加一个 “else 区块”。

```html
<button @click="awesome = !awesome">Toggle</button>
<h1 v-if="awesome">Vue is awesome!</h1>
<h1 v-else>Oh no  </h1>
```

在演练场中尝试一下

一个 `v-else` 元素必须跟在一个 `v-if` 或者 `v-else-if` 元素后面，否则它将不会被识别。

2.6.3 v-else-if

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

```html
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
```

```html
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

Similar to `v-else`, a `v-else-if` element must immediately follow a `v-if` or a `v-else-if` element.

2.6.4 v-if on <template>

Because `v-if` is a directive, it has to be attached to a single element. But what if we want to toggle more than one element? In this case we can use `v-if` on a `<template>` element, which serves as an invisible wrapper. The final rendered result will not include the `<template>` element.

html

```html
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

`v-else` and `v-else-if` can also be used on `<template>`.

2.6.5 v-show

Another option for conditionally displaying an element is the `v-show` directive. The usage is largely the same:

html

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

The difference is that an element with `v-show` will always be rendered and remain in the DOM; `v-show` only toggles the `display` CSS property of the element.

`v-show` doesn't support the `<template>` element, nor does it work with `v-else`.

```html
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

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

2.6.4 <template> 上的 v-if

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

html

```html
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

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

2.6.5 v-show

另一个可以用来按条件显示一个元素的指令是 `v-show`。其用法基本一样：

html

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

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

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

2.6.6 v-if vs. v-show

`v-if` is ”real” conditional rendering because it ensures that event listeners and child components inside the conditional block are properly destroyed and re-created during toggles.

`v-if` is also **lazy**: if the condition is false on initial render, it will not do anything - the conditional block won’t be rendered until the condition becomes true for the first time.

In comparison, `v-show` is much simpler - the element is always rendered regardless of initial condition, with CSS-based toggling.

Generally speaking, `v-if` has higher toggle costs while `v-show` has higher initial render costs. So prefer `v-show` if you need to toggle something very often, and prefer `v-if` if the condition is unlikely to change at runtime.

2.6.7 v-if with v-for

> **Note**
> It’s **not** recommended to use `v-if` and `v-for` on the same element due to implicit precedence. Refer to style guide for details.

When `v-if` and `v-for` are both used on the same element, `v-if` will be evaluated first. See the list rendering guide for details.

2.7 List Rendering

2.7.1 v-for

We can use the `v-for` directive to render a list of items based on an array. The `v-for` directive requires a special syntax in the form of `item in items`, where `items` is the source data array and `item` is an **alias** for the array element being iterated on:

js
```js
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

html
```html
<li v-for="item in items">
  {{ item.message }}
```

2.6.6 v-if vs. v-show

`v-if` 是 “真实的” 按条件渲染，因为它确保了在切换时，条件区块内的事件监听器和子组件都会被销毁与重建。

`v-if` 也是**惰性**的：如果在初次渲染时条件值为 false，则不会做任何事。条件区块只有当条件首次变为 true 时才被渲染。

相比之下，`v-show` 简单许多，元素无论初始条件如何，始终会被渲染，只有 CSS `display` 属性会被切换。

总的来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要频繁切换，则使用 `v-show` 较好；如果在运行时绑定条件很少改变，则 `v-if` 会更合适。

2.6.7 v-if 和 v-for

> **警告**
> 同时使用 `v-if` 和 `v-for` 是**不推荐的**，因为这样二者的优先级不明显。请查看风格指南获得更多信息。

当 `v-if` 和 `v-for` 同时存在于一个元素上的时候，`v-if` 会首先被执行。请查看列表渲染指南获取更多细节。

2.7 列表渲染

2.7.1 v-for

我们可以使用 `v-for` 指令基于一个数组来渲染一个列表。`v-for` 指令的值需要使用 `item in items` 形式的特殊语法，其中 `items` 是源数据的数组，而 `item` 是迭代项的**别名**：

js
```js
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

html
```html
<li v-for="item in items">
  {{ item.message }}
```

```
</li>
```

Inside the `v-for` scope, template expressions have access to all parent scope properties. In addition, `v-for` also supports an optional second alias for the index of the current item:

```js
const parentMessage = ref('Parent')
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

```html
<li v-for="(item, index) in items">
  {{ parentMessage }} - {{ index }} - {{ item.message }}
</li>
```

Try it in the Playground

The variable scoping of `v-for` is similar to the following JavaScript:

```js
const parentMessage = 'Parent'
const items = [
  /* ... */
]
items.forEach((item, index) => {
  // 可以访问外层的 `parentMessage`
  // 而 `item` 和 `index` 只在这个作用域可用
  console.log(parentMessage, item.message, index)
})
```

Notice how the `v-for` value matches the function signature of the `forEach` callback. In fact, you can use destructuring on the `v-for` item alias similar to destructuring function arguments:

```html
<li v-for="{ message } in items">
  {{ message }}
</li>
<!-- 有 index 索引时 -->
<li v-for="({ message }, index) in items">
  {{ message }} {{ index }}
</li>
```

For nested `v-for`, scoping also works similar to nested functions. Each `v-for` scope has access to

```
</li>
```

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

```js
const parentMessage = ref('Parent')
const items = ref([{ message: 'Foo' }, { message: 'Bar' }])
```

```html
<li v-for="(item, index) in items">
  {{ parentMessage }} - {{ index }} - {{ item.message }}
</li>
```

在演练场中尝试一下

`v-for` 变量的作用域和下面的 JavaScript 代码很类似：

```js
const parentMessage = 'Parent'
const items = [
  /* ... */
]
items.forEach((item, index) => {
  // 可以访问外层的 `parentMessage`
  // 而 `item` 和 `index` 只在这个作用域可用
  console.log(parentMessage, item.message, index)
})
```

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

```html
<li v-for="{ message } in items">
  {{ message }}
</li>
<!-- 有 index 索引时 -->
<li v-for="({ message }, index) in items">
  {{ message }} {{ index }}
</li>
```

对于多层嵌套的 `v-for`，作用域的工作方式和函数的作用域很类似。每个 `v-for` parent scopes:

```html
<li v-for="item in items">
  <span v-for="childItem in item.children">
    {{ item.message }} {{ childItem }}
  </span>
</li>
```

You can also use `of` as the delimiter instead of `in`, so that it is closer to JavaScript's syntax for iterators:

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

2.7.2 v-for with an Object

You can also use `v-for` to iterate through the properties of an object. The iteration order will be based on the result of calling `Object.keys()` on the object:

```js
const myObject = reactive({
    title: 'How to do lists in Vue',
    author: 'Jane Doe',
    publishedAt: '2016-04-10'
})
```

```html
<ul>
    <li v-for="value in myObject">
    {{ value }}
    </li>
</ul>
```

You can also provide a second alias for the property's name (a.k.a. key):

```html
<li v-for="(value, key) in myObject">
    {{ key }}: {{ value }}
</li>
```

And another for the index:

作用域都可以访问到父级作用域：

```html
<li v-for="item in items">
  <span v-for="childItem in item.children">
    {{ item.message }} {{ childItem }}
  </span>
</li>
```

你也可以使用 `of` 作为分隔符来替代 `in`，这更接近 JavaScript 的迭代器语法：

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

2.7.2 v-for 与对象

你也可以使用 `v-for` 来遍历一个对象的所有属性。遍历的顺序会基于对该对象调用 `Object.keys()` 的返回值来决定。

```js
const myObject = reactive({
    title: 'How to do lists in Vue',
    author: 'Jane Doe',
    publishedAt: '2016-04-10'
})
```

```html
<ul>
    <li v-for="value in myObject">
    {{ value }}
    </li>
</ul>
```

可以通过提供第二个参数表示属性名 (例如 key)：

```html
<li v-for="(value, key) in myObject">
    {{ key }}: {{ value }}
</li>
```

第三个参数表示位置索引：

```html
<li v-for="(value, key, index) in myObject">
    {{ index }}. {{ key }}: {{ value }}
</li>
```

Try it in the Playground

2.7.3 v-for with a Range

`v-for` can also take an integer. In this case it will repeat the template that many times, based on a range of `1...n`.

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

Note here `n` starts with an initial value of `1` instead of `0`.

2.7.4 v-for on <template>

Similar to template `v-if`, you can also use a `<template>` tag with `v-for` to render a block of multiple elements. For example:

```html
<ul>
  <template v-for="item in items">
    <li>{{ item.msg }}</li>
    <li class="divider" role="presentation"></li>
  </template>
</ul>
```

2.7.5 v-for with v-if

Note

It's **not** recommended to use `v-if` and `v-for` on the same element due to implicit precedence. Refer to style guide for details.

When they exist on the same node, `v-if` has a higher priority than `v-for`. That means the `v-if`

```html
<li v-for="(value, key, index) in myObject">
    {{ index }}. {{ key }}: {{ value }}
</li>
```

在演练场中尝试一下

2.7.3 在 v-for 里使用范围值

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

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

注意此处 `n` 的初值是从 `1` 开始而非 `0`。

2.7.4 <template> 上的 v-for

与模板上的 `v-if` 类似，你也可以在 `<template>` 标签上使用 `v-for` 来渲染一个包含多个元素的块。例如：

```html
<ul>
  <template v-for="item in items">
    <li>{{ item.msg }}</li>
    <li class="divider" role="presentation"></li>
  </template>
</ul>
```

2.7.5 v-for 与 v-if

注意

同时使用 `v-if` 和 `v-for` 是**不推荐的**，因为这样二者的优先级不明显。请转阅风格指南查看更多细节。

当它们同时存在于一个节点上时，`v-if` 比 `v-for` 的优先级更高。这意味着 `v-if` condition will not have access to variables from the scope of the v-for:

```html
<!--
 这会抛出一个错误，因为属性 todo 此时
 没有在该实例上定义
-->
<li v-for="todo in todos" v-if="!todo.isComplete">
  {{ todo.name }}
</li>
```

This can be fixed by moving v-for to a wrapping <template> tag (which is also more explicit):

```html
<template v-for="todo in todos">
  <li v-if="!todo.isComplete">
    {{ todo.name }}
  </li>
</template>
```

2.7.6 Maintaining State with key

When Vue is updating a list of elements rendered with v-for, by default it uses an "in-place patch" strategy. If the order of the data items has changed, instead of moving the DOM elements to match the order of the items, Vue will patch each element in-place and make sure it reflects what should be rendered at that particular index.

This default mode is efficient, but **only suitable when your list render output does not rely on child component state or temporary DOM state (e.g. form input values)**.

To give Vue a hint so that it can track each node's identity, and thus reuse and reorder existing elements, you need to provide a unique key attribute for each item:

```html
<div v-for="item in items" :key="item.id">
    <!-- 内容 -->
</div>
```

When using <template v-for>, the key should be placed on the <template> container:

的条件将无法访问到 v-for 作用域内定义的变量别名：

```html
<!--
 这会抛出一个错误，因为属性 todo 此时
 没有在该实例上定义
-->
<li v-for="todo in todos" v-if="!todo.isComplete">
  {{ todo.name }}
</li>
```

在外新包装一层 <template> 再在其上使用 v-for 可以解决这个问题 (这也更加明显易读)：

```html
<template v-for="todo in todos">
  <li v-if="!todo.isComplete">
    {{ todo.name }}
  </li>
</template>
```

2.7.6 通过 key 管理状态

Vue 默认按照 “就地更新” 的策略来更新通过 v-for 渲染的元素列表。当数据项的顺序改变时，Vue 不会随之移动 DOM 元素的顺序，而是就地更新每个元素，确保它们在原本指定的索引位置上渲染。

默认模式是高效的，但**只适用于列表渲染输出的结果不依赖子组件状态或者临时 DOM 状态 (例如表单输入值) 的情况**。

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

```html
<div v-for="item in items" :key="item.id">
    <!-- 内容 -->
</div>
```

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

```html
<template v-for="todo in todos" :key="todo.name">
    <li>{{ todo.name }}</li>
</template>
```

Note

`key` here is a special attribute being bound with `v-bind`. It should not be confused with the property key variable when using `v-for` with an object.

It is recommended to provide a `key` attribute with `v-for` whenever possible, unless the iterated DOM content is simple (i.e. contains no components or stateful DOM elements), or you are intentionally relying on the default behavior for performance gains.

The `key` binding expects primitive values - i.e. strings and numbers. Do not use objects as `v-for` keys. For detailed usage of the `key` attribute, please see the `key` API documentation.

2.7.7 v-for with a Component

> This section assumes knowledge of Components. Feel free to skip it and come back later.

You can directly use `v-for` on a component, like any normal element (don't forget to provide a `key`):

```html
<MyComponent v-for="item in items" :key="item.id" />
```

However, this won't automatically pass any data to the component, because components have isolated scopes of their own. In order to pass the iterated data into the component, we should also use props:

```html
<MyComponent
  v-for="(item, index) in items"
  :item="item"
  :index="index"
  :key="item.id"
/>
```

```html
<template v-for="todo in todos" :key="todo.name">
    <li>{{ todo.name }}</li>
</template>
```

注意

`key` 在这里是一个通过 `v-bind` 绑定的特殊 attribute。请不要和在 `v-for` 中使用对象里所提到的对象属性名相混淆。

推荐在任何可行的时候为 `v-for` 提供一个 `key` attribute，除非所迭代的 DOM 内容非常简单 (例如：不包含组件或有状态的 DOM 元素)，或者你想有意采用默认行为来提高性能。

`key` 绑定的值期望是一个基础类型的值，例如字符串或 number 类型。不要用对象作为 `v-for` 的 key。关于 `key` attribute 的更多用途细节，请参阅 `key` API 文档。

2.7.7 组件上使用 v-for

> 这一小节假设你已了解组件的相关知识，或者你也可以先跳过这里，之后再回来看。

我们可以直接在组件上使用 `v-for`，和在一般的元素上使用没有区别 (别忘记提供一个 `key`)：

```html
<MyComponent v-for="item in items" :key="item.id" />
```

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

```html
<MyComponent
  v-for="(item, index) in items"
  :item="item"
  :index="index"
  :key="item.id"
/>
```

The reason for not automatically injecting `item` into the component is because that makes the component tightly coupled to how `v-for` works. Being explicit about where its data comes from makes the component reusable in other situations.

Check out this example of a simple todo list to see how to render a list of components using `v-for`, passing different data to each instance.

2.7.8 Array Change Detection

Mutation Methods

Vue is able to detect when a reactive array's mutation methods are called and trigger necessary updates. These mutation methods are:

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

Replacing an Array

Mutation methods, as the name suggests, mutate the original array they are called on. In comparison, there are also non-mutating methods, e.g. `filter()`, `concat()` and `slice()`, which do not mutate the original array but **always return a new array**. When working with non-mutating methods, we should replace the old array with the new one:

```js
// `items` 是一个数组的 ref
items.value = items.value.filter((item) => item.message.match(/Foo/))
```

You might think this will cause Vue to throw away the existing DOM and re-render the entire list - luckily, that is not the case. Vue implements some smart heuristics to maximize DOM element

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

这里是一个简单的 Todo List 的例子，展示了如何通过 `v-for` 来渲染一个组件列表，并向每个实例中传入不同的数据。

2.7.8 数组变化侦测

变更方法

Vue 能够侦听响应式数组的变更方法，并在它们被调用时触发相关的更新。这些变更方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

替换一个数组

变更方法，顾名思义，就是会对调用它们的原数组进行变更。相对地，也有一些不可变 (immutable) 方法，例如 `filter()`，`concat()` 和 `slice()`，这些都不会更改原数组，而总是**返回一个新数组**。当遇到的是非变更方法时，我们需要将旧的数组替换为新的：

```js
// `items` 是一个数组的 ref
items.value = items.value.filter((item) => item.message.match(/Foo/))
```

你可能认为这将导致 Vue 丢弃现有的 DOM 并重新渲染整个列表——幸运的是，情况并非如此。Vue 实现了一些巧妙的方法来最大化对 DOM 元素的重用，因此

reuse, so replacing an array with another array containing overlapping objects is a very efficient operation.

2.7.9 Displaying Filtered/Sorted Results

Sometimes we want to display a filtered or sorted version of an array without actually mutating or resetting the original data. In this case, you can create a computed property that returns the filtered or sorted array.

For example:

js
```
const numbers = ref([1, 2, 3, 4, 5])
const evenNumbers = computed(() => {
  return numbers.value.filter((n) => n % 2 === 0)
})
```

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

In situations where computed properties are not feasible (e.g. inside nested `v-for` loops), you can use a method:

js
```
const sets = ref([
  [1, 2, 3, 4, 5],
  [6, 7, 8, 9, 10]
])
function even(numbers) {
  return numbers.filter((number) => number % 2 === 0)
}
```

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

Be careful with `reverse()` and `sort()` in a computed property! These two methods will mutate the original array, which should be avoided in computed getters. Create a copy of the original array before calling these methods:

```
- return numbers.reverse()
```

用另一个包含部分重叠对象的数组来做替换，仍会是一种非常高效的操作。

2.7.9 展示过滤或排序后的结果

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

举例来说：

js
```
const numbers = ref([1, 2, 3, 4, 5])
const evenNumbers = computed(() => {
  return numbers.value.filter((n) => n % 2 === 0)
})
```

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

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

js
```
const sets = ref([
  [1, 2, 3, 4, 5],
  [6, 7, 8, 9, 10]
])
function even(numbers) {
  return numbers.filter((number) => number % 2 === 0)
}
```

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

在计算属性中使用 `reverse()` 和 `sort()` 的时候务必小心！这两个方法将变更原始数组，计算函数中不应该这么做。请在调用这些方法之前创建一个原数组的副本：

```
- return numbers.reverse()
```

```
+ return [...numbers].reverse()
```

2.8 Event Handling

2.8.1 Listening to Events

We can use the `v-on` directive, which we typically shorten to the `@` symbol, to listen to DOM events and run some JavaScript when they're triggered. The usage would be `v-on:click="handler"` or with the shortcut, `@click="handler"`.

The handler value can be one of the following:

1. **Inline handlers:** Inline JavaScript to be executed when the event is triggered (similar to the native `onclick` attribute).
2. **Method handlers:** A property name or path that points to a method defined on the component.

2.8.2 Inline Handlers

Inline handlers are typically used in simple cases, for example:

```js
const count = ref(0)
```

```html
<button @click="count++">Add 1</button>
<p>Count is: {{ count }}</p>
```

Try it in the Playground

2.8.3 Method Handlers

The logic for many event handlers will be more complex though, and likely isn't feasible with inline handlers. That's why `v-on` can also accept the name or path of a component method you'd like to call.

For example:

```
+ return [...numbers].reverse()
```

2.8 事件处理

2.8.1 监听事件

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

事件处理器 (handler) 的值可以是：

1. **内联事件处理器**：事件被触发时执行的内联 JavaScript 语句 (与 `onclick` 类似)。
2. **方法事件处理器**：一个指向组件上定义的方法的属性名或是路径。

2.8.2 内联事件处理器

内联事件处理器通常用于简单场景，例如：

```js
const count = ref(0)
```

```html
<button @click="count++">Add 1</button>
<p>Count is: {{ count }}</p>
```

在演练场中尝试一下

2.8.3 方法事件处理器

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

举例来说：

```js
const name = ref('Vue.js')
function greet(event) {
  alert(`Hello ${name.value}!`)
  // `event` 是 DOM 原生事件
  if (event) {
    alert(event.target.tagName)
  }
}
```

```html
<!-- `greet` 是上面定义过的方法名 -->
<button @click="greet">Greet</button>
```

Try it in the Playground

A method handler automatically receives the native DOM Event object that triggers it - in the example above, we are able to access the element dispatching the event via `event.target.tagName`.

See also: Typing Event Handlers

Method vs. Inline Detection

The template compiler detects method handlers by checking whether the `v-on` value string is a valid JavaScript identifier or property access path. For example, `foo`, `foo.bar` and `foo['bar']` are treated as method handlers, while `foo()` and `count++` are treated as inline handlers.

2.8.4 Calling Methods in Inline Handlers

Instead of binding directly to a method name, we can also call methods in an inline handler. This allows us to pass the method custom arguments instead of the native event:

```js
function say(message) {
    alert(message)
}
```

```js
const name = ref('Vue.js')
function greet(event) {
  alert(`Hello ${name.value}!`)
  // `event` 是 DOM 原生事件
  if (event) {
    alert(event.target.tagName)
  }
}
```

```html
<!-- `greet` 是上面定义过的方法名 -->
<button @click="greet">Greet</button>
```

在演练场中尝试一下

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

你也可以看看为事件处理器标注类型这一章了解更多。

方法与内联事件判断

模板编译器会通过检查 `v-on` 的值是否是合法的 JavaScript 标识符或属性访问路径来断定是何种形式的事件处理器。举例来说，`foo`、`foo.bar` 和 `foo['bar']` 会被视为方法事件处理器，而 `foo()` 和 `count++` 会被视为内联事件处理器。

2.8.4 在内联处理器中调用方法

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

```js
function say(message) {
    alert(message)
}
```

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

Try it in the Playground

2.8.5 Accessing Event Argument in Inline Handlers

Sometimes we also need to access the original DOM event in an inline handler. You can pass it into a method using the special `$event` variable, or use an inline arrow function:

```html
<!-- 使用特殊的 $event 变量 -->
<button @click="warn('Form cannot be submitted yet.', $event)">
    Submit
</button>
<!-- 使用内联箭头函数 -->
<button @click="(event) => warn('Form cannot be submitted yet.', event)">
    Submit
</button>
```

```js
function warn(message, event) {
    // 这里可以访问原生事件
    if (event) {
    event.preventDefault()
    }
    alert(message)
}
```

2.8.6 Event Modifiers

It is a very common need to call `event.preventDefault()` or `event.stopPropagation()` inside event handlers. Although we can do this easily inside methods, it would be better if the methods can be purely about data logic rather than having to deal with DOM event details.

To address this problem, Vue provides **event modifiers** for `v-on`. Recall that modifiers are directive postfixes denoted by a dot.

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

在演练场中尝试一下

2.8.5 在内联事件处理器中访问事件参数

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

```html
<!-- 使用特殊的 $event 变量 -->
<button @click="warn('Form cannot be submitted yet.', $event)">
    Submit
</button>
<!-- 使用内联箭头函数 -->
<button @click="(event) => warn('Form cannot be submitted yet.', event)">
    Submit
</button>
```

```js
function warn(message, event) {
    // 这里可以访问原生事件
    if (event) {
    event.preventDefault()
    }
    alert(message)
}
```

2.8.6 事件修饰符

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

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

- .stop
- .prevent
- .self
- .capture
- .once
- .passive

```html
<!-- 单击事件将停止传递 -->
<a @click.stop="doThis"></a>
<!-- 提交事件将不再重新加载页面 -->
<form @submit.prevent="onSubmit"></form>
<!-- 修饰语可以使用链式书写 -->
<a @click.stop.prevent="doThat"></a>
<!-- 也可以只有修饰符 -->
<form @submit.prevent></form>
<!-- 仅当 event.target 是元素本身时才会触发事件处理器 -->
<!-- 例如：事件处理器不来自子元素 -->
<div @click.self="doThat">...</div>
```

TIP

Order matters when using modifiers because the relevant code is generated in the same order. Therefore using `@click.prevent.self` will prevent **click's default action on the element itself and its children**, while `@click.self.prevent` will only prevent click's default action on the element itself.

The `.capture`, `.once`, and `.passive` modifiers mirror the options of the native `addEventListener` method:

```html
<!-- 添加事件监听器时，使用 `capture` 捕获模式 -->
<!-- 例如：指向内部元素的事件，在被内部元素处理前，先被外部处理 -->
<div @click.capture="doThis">...</div>
<!-- 点击事件最多被触发一次 -->
<a @click.once="doThis"></a>
<!-- 滚动事件的默认行为 (scrolling) 将立即发生而非等待 `onScroll` 完成 -->
```

- .stop
- .prevent
- .self
- .capture
- .once
- .passive

```html
<!-- 单击事件将停止传递 -->
<a @click.stop="doThis"></a>
<!-- 提交事件将不再重新加载页面 -->
<form @submit.prevent="onSubmit"></form>
<!-- 修饰语可以使用链式书写 -->
<a @click.stop.prevent="doThat"></a>
<!-- 也可以只有修饰符 -->
<form @submit.prevent></form>
<!-- 仅当 event.target 是元素本身时才会触发事件处理器 -->
<!-- 例如：事件处理器不来自子元素 -->
<div @click.self="doThat">...</div>
```

TIP

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

`.capture`、`.once` 和 `.passive` 修饰符与原生 `addEventListener` 事件相对应：

```html
<!-- 添加事件监听器时，使用 `capture` 捕获模式 -->
<!-- 例如：指向内部元素的事件，在被内部元素处理前，先被外部处理 -->
<div @click.capture="doThis">...</div>
<!-- 点击事件最多被触发一次 -->
<a @click.once="doThis"></a>
<!-- 滚动事件的默认行为 (scrolling) 将立即发生而非等待 `onScroll` 完成 -->
```

```html
<!-- 以防其中包含 `event.preventDefault()` -->
<div @scroll.passive="onScroll">...</div>
```

The `.passive` modifier is typically used with touch event listeners for improving performance on mobile devices.

> **TIP**
>
> Do not use `.passive` and `.prevent` together, because `.passive` already indicates to the browser that you *do not* intend to prevent the event's default behavior, and you will likely see a warning from the browser if you do so.

2.8.7 Key Modifiers

When listening for keyboard events, we often need to check for specific keys. Vue allows adding key modifiers for `v-on` or `@` when listening for key events:

```html
<!-- 仅在 `key` 为 `Enter` 时调用 `submit` -->
<input @keyup.enter="submit" />
```

You can directly use any valid key names exposed via `KeyboardEvent.key` as modifiers by converting them to kebab-case.

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

In the above example, the handler will only be called if `$event.key` is equal to `'PageDown'`.

Key Aliases

Vue provides aliases for the most commonly used keys:

- `.enter`
- `.tab`
- `.delete` (captures both ”Delete” and ”Backspace” keys)
- `.esc`

```html
<!-- 以防其中包含 `event.preventDefault()` -->
<div @scroll.passive="onScroll">...</div>
```

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

> **TIP**
>
> 请勿同时使用 `.passive` 和 `.prevent`，因为 `.passive` 已经向浏览器表明了你不想阻止事件的默认行为。如果你这么做了，则 `.prevent` 会被忽略，并且浏览器会抛出警告。

2.8.7 按键修饰符

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

```html
<!-- 仅在 `key` 为 `Enter` 时调用 `submit` -->
<input @keyup.enter="submit" />
```

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

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

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

按键别名

Vue 为一些常用的按键提供了别名：

- `.enter`
- `.tab`
- `.delete` (捕获 “Delete” 和 “Backspace” 两个按键)
- `.esc`

- .space
- .up
- .down
- .left
- .right

System Modifier Keys

You can use the following modifiers to trigger mouse or keyboard event listeners only when the corresponding modifier key is pressed:

- .ctrl
- .alt
- .shift
- .meta

Note

On Macintosh keyboards, meta is the command key ( ). On Windows keyboards, meta is the Windows key ( ). On Sun Microsystems keyboards, meta is marked as a solid diamond ( ). On certain keyboards, specifically MIT and Lisp machine keyboards and successors, such as the Knight keyboard, space-cadet keyboard, meta is labeled “META”. On Symbolics keyboards, meta is labeled “META” or “Meta”.

For example:

html

```html
<!-- Alt + Enter -->
<input @keyup.alt.enter="clear" />
<!-- Ctrl + 点击 -->
<div @click.ctrl="doSomething">Do something</div>
```

- .space
- .up
- .down
- .left
- .right

系统按键修饰符

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

- .ctrl
- .alt
- .shift
- .meta

注意

在 Mac 键盘上，meta 是 Command 键 ( )。在 Windows 键盘上，meta 键是 Windows 键 ( )。在 Sun 微机系统键盘上，meta 是钻石键 ( )。在某些键盘上，特别是 MIT 和 Lisp 机器的键盘及其后代版本的键盘，如 Knight 键盘，space-cadet 键盘，meta 都被标记为 “META”。在 Symbolics 键盘上，meta 也被标识为 “META” 或 “Meta”。

举例来说：

html

```html
<!-- Alt + Enter -->
<input @keyup.alt.enter="clear" />
<!-- Ctrl + 点击 -->
<div @click.ctrl="doSomething">Do something</div>
```

TIP

Note that modifier keys are different from regular keys and when used with `keyup` events, they have to be pressed when the event is emitted. In other words, `keyup.ctrl` will only trigger if you release a key while holding down `ctrl`. It won't trigger if you release the `ctrl` key alone.

.exact Modifier

The `.exact` modifier allows control of the exact combination of system modifiers needed to trigger an event.

```html
<!-- 当按下 Ctrl 时，即使同时按下 Alt 或 Shift 也会触发 -->
<button @click.ctrl="onClick">A</button>
<!-- 仅当按下 Ctrl 且未按任何其他键时才会触发 -->
<button @click.ctrl.exact="onCtrlClick">A</button>
<!-- 仅当没有按下任何系统按键时触发 -->
<button @click.exact="onClick">A</button>
```

2.8.8 Mouse Button Modifiers

- `.left`
- `.right`
- `.middle`

These modifiers restrict the handler to events triggered by a specific mouse button.

2.9 Form Input Bindings

When dealing with forms on the frontend, we often need to sync the state of form input elements with corresponding state in JavaScript. It can be cumbersome to manually wire up value bindings and change event listeners:

TIP

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

.exact 修饰符

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

```html
<!-- 当按下 Ctrl 时，即使同时按下 Alt 或 Shift 也会触发 -->
<button @click.ctrl="onClick">A</button>
<!-- 仅当按下 Ctrl 且未按任何其他键时才会触发 -->
<button @click.ctrl.exact="onCtrlClick">A</button>
<!-- 仅当没有按下任何系统按键时触发 -->
<button @click.exact="onClick">A</button>
```

2.8.8 鼠标按键修饰符

- `.left`
- `.right`
- `.middle`

这些修饰符将处理程序限定为由特定鼠标按键触发的事件。

2.9 表单输入绑定

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

html

```html
  <input
    :value="text"
    @input="event => text = event.target.value">
```

The `v-model` directive helps us simplify the above to:

html

```html
  <input v-model="text">
```

In addition, `v-model` can be used on inputs of different types, `<textarea>`, and `<select>` elements. It automatically expands to different DOM property and event pairs based on the element it is used on:

- `<input>` with text types and `<textarea>` elements use `value` property and `input` event;
- `<input type="checkbox">` and `<input type="radio">` use `checked` property and `change` event;
- `<select>` use `value` as a prop and `change` as an event.

Note
`v-model` will ignore the initial `value`, `checked` or `selected` attributes found on any form elements. It will always treat the current bound JavaScript state as the source of truth. You should declare the initial value on the JavaScript side, using reactivity APIs.

2.9.1 Basic Usage

Text

html

```html
<p>Message is: {{ message }}</p>
<input v-model="message" placeholder="edit me" />
```

Try it in the Playground

html

```html
<input
    :value="text"
    @input="event => text = event.target.value">
```

`v-model` 指令帮我们简化了这一步骤：

html

```html
<input v-model="text">
```

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

- 文本类型的 `<input>` 和 `<textarea>` 元素会绑定 `value` property 并侦听 `input` 事件；
- `<input type="checkbox">` 和 `<input type="radio">` 会绑定 `checked` property 并侦听 `change` 事件；
- `<select>` 会绑定 `value` property 并侦听 `change` 事件。

注意
`v-model` 会忽略任何表单元素上初始的 `value`、`checked` 或 `selected` attribute。它将始终将当前绑定的 JavaScript 状态视为数据的正确来源。你应该在 JavaScript 中使用响应式系统的 API来声明该初始值。

2.9.1 基本用法

文本

html

```html
<p>Message is: {{ message }}</p>
<input v-model="message" placeholder="edit me" />
```

在演练场中尝试一下

Note
For languages that require an IME (Chinese, Japanese, Korean etc.), you'll notice that `v-model` doesn't get updated during IME composition. If you want to respond to these updates as well, use your own `input` event listener and `value` binding instead of using `v-model`.

Multiline text

```html
<span>Multiline message is:</span>
<p style="white-space: pre-line;">{{ message }}</p>
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```

Try it in the Playground

Note that interpolation inside `<textarea>` won't work. Use `v-model` instead.

```html
<!-- 错误 -->
<textarea>{{ text }}</textarea>
<!-- 正确 -->
<textarea v-model="text"></textarea>
```

Checkbox

Single checkbox, boolean value:

```html
<input type="checkbox" id="checkbox" v-model="checked" />
<label for="checkbox">{{ checked }}</label>
```

Try it in the Playground

We can also bind multiple checkboxes to the same array or Set value:

```js
const checkedNames = ref([])
```

```html
<div>Checked names: {{ checkedNames }}</div>
<!-- -->
```

注意
对于需要使用 IME 的语言 (中文，日文和韩文等)，你会发现 `v-model` 不会在 IME 输入还在拼字阶段时触发更新。如果你的确想在拼字阶段也触发更新，请直接使用自己的 `input` 事件监听器和 `value` 绑定而不要使用 `v-model`。

多行文本

```html
<span>Multiline message is:</span>
<p style="white-space: pre-line;">{{ message }}</p>
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```

在演练场中尝试一下

注意在 `<textarea>` 中是不支持插值表达式的。请使用 `v-model` 来替代：

```html
<!-- 错误 -->
<textarea>{{ text }}</textarea>
<!-- 正确 -->
<textarea v-model="text"></textarea>
```

复选框

单一的复选框，绑定布尔类型值：

```html
<input type="checkbox" id="checkbox" v-model="checked" />
<label for="checkbox">{{ checked }}</label>
```

在演练场中尝试一下

我们也可以将多个复选框绑定到同一个数组或集合的值：

```js
const checkedNames = ref([])
```

```html
<div>Checked names: {{ checkedNames }}</div>
<!-- -->
```

```html
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<!-- -->
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>
<!-- -->
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
<label for="mike">Mike</label>
```

In this case, the `checkedNames` array will always contain the values from the currently checked boxes.

Try it in the Playground

Radio

html
```html
<div>Picked: {{ picked }}</div>
<!-- -->
<input type="radio" id="one" value="One" v-model="picked" />
<label for="one">One</label>
<!-- -->
<input type="radio" id="two" value="Two" v-model="picked" />
<label for="two">Two</label>
```

Try it in the Playground

Select

Single select:

html
```html
<div>Selected: {{ selected }}</div>
<!-- -->
<select v-model="selected">
  <option disabled value="">Please select one</option>
  <option>A</option>
  <option>B</option>
```

```html
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<!-- -->
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>
<!-- -->
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
<label for="mike">Mike</label>
```

在这个例子中，`checkedNames` 数组将始终包含所有当前被选中的框的值。

在演练场中尝试一下

单选按钮

html

```html
<div>Picked: {{ picked }}</div>
<!-- -->
<input type="radio" id="one" value="One" v-model="picked" />
<label for="one">One</label>
<!-- -->
<input type="radio" id="two" value="Two" v-model="picked" />
<label for="two">Two</label>
```

在演练场中尝试一下

选择器

单个选择器的示例如下：

html

```html
<div>Selected: {{ selected }}</div>
<!-- -->
<select v-model="selected">
  <option disabled value="">Please select one</option>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
```

Try it in the Playground

Note

If the initial value of your `v-model` expression does not match any of the options, the `<select>` element will render in an "unselected" state. On iOS this will cause the user not being able to select the first item because iOS does not fire a change event in this case. It is therefore recommended to provide a disabled option with an empty value, as demonstrated in the example above.

Multiple select (bound to array):

html

```html
<div>Selected: {{ selected }}</div>
<!-- -->
<select v-model="selected" multiple>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
```

Try it in the Playground

Select options can be dynamically rendered with `v-for`:

js

```js
const selected = ref('A')
//
const options = ref([
  { text: 'One', value: 'A' },
  { text: 'Two', value: 'B' },
  { text: 'Three', value: 'C' }
])
```

html

```html
<select v-model="selected">
  <option v-for="option in options" :value="option.value">
    {{ option.text }}
```

```html
  <option>C</option>
</select>
```

在演练场中尝试一下

注意

如果 `v-model` 表达式的初始值不匹配任何一个选择项，`<select>` 元素会渲染成一个 “未选择” 的状态。在 iOS 上，这将导致用户无法选择第一项，因为 iOS 在这种情况下不会触发一个 change 事件。因此，我们建议提供一个空值的禁用选项，如上面的例子所示。

多选 (值绑定到一个数组)：

html

```html
<div>Selected: {{ selected }}</div>
<!-- -->
<select v-model="selected" multiple>
  <option>A</option>
  <option>B</option>
  <option>C</option>
</select>
```

在演练场中尝试一下

选择器的选项可以使用 `v-for` 动态渲染：

js

```js
const selected = ref('A')
//
const options = ref([
  { text: 'One', value: 'A' },
  { text: 'Two', value: 'B' },
  { text: 'Three', value: 'C' }
])
```

html

```html
<select v-model="selected">
  <option v-for="option in options" :value="option.value">
    {{ option.text }}
  </option>
</select>
<!-- -->
<div>Selected: {{ selected }}</div>
```

Try it in the Playground

2.9.2 Value Bindings

For radio, checkbox and select options, the `v-model` binding values are usually static strings (or booleans for checkbox):

```html
<!-- `picked` 在被选择时是字符串 "a" -->
<input type="radio" v-model="picked" value="a" />
<!-- -->
<!-- `toggle` 只会为 true 或 false -->
<input type="checkbox" v-model="toggle" />
<!-- -->
<!-- `selected` 在第一项被选中时为字符串 "abc" -->
<select v-model="selected">
    <option value="abc">ABC</option>
</select>
```

But sometimes we may want to bind the value to a dynamic property on the current active instance. We can use `v-bind` to achieve that. In addition, using `v-bind` allows us to bind the input value to non-string values.

Checkbox

```html
<input
  type="checkbox"
  v-model="toggle"
  true-value="yes"
  false-value="no" />
```

```
  </option>
</select>
<!-- -->
<div>Selected: {{ selected }}</div>
```

在演练场中尝试一下

2.9.2 值绑定

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

```html
<!-- `picked` 在被选择时是字符串 "a" -->
<input type="radio" v-model="picked" value="a" />
<!-- -->
<!-- `toggle` 只会为 true 或 false -->
<input type="checkbox" v-model="toggle" />
<!-- -->
<!-- `selected` 在第一项被选中时为字符串 "abc" -->
<select v-model="selected">
    <option value="abc">ABC</option>
</select>
```

但有时我们可能希望将该值绑定到当前组件实例上的动态数据。这可以通过使用 `v-bind` 来实现。此外，使用 `v-bind` 还使我们可以将选项值绑定为非字符串的数据类型。

复选框

```html
<input
  type="checkbox"
  v-model="toggle"
  true-value="yes"
  false-value="no" />
```

true-value and false-value are Vue-specific attributes that only work with v-model. Here the toggle property's value will be set to 'yes' when the box is checked, and set to 'no' when unchecked. You can also bind them to dynamic values using v-bind:

html
```html
<input
  type="checkbox"
  v-model="toggle"
  :true-value="dynamicTrueValue"
  :false-value="dynamicFalseValue" />
```

Tip

The true-value and false-value attributes don't affect the input's value attribute, because browsers don't include unchecked boxes in form submissions. To guarantee that one of two values is submitted in a form (e.g. ”yes” or ”no”), use radio inputs instead.

Radio

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

pick will be set to the value of first when the first radio input is checked, and set to the value of second when the second one is checked.

Select Options

html
```html
<select v-model="selected">
  <!-- 内联对象字面量 -->
  <option :value="{ number: 123 }">123</option>
</select>
```

v-model supports value bindings of non-string values as well! In the above example, when the option is selected, selected will be set to the object literal value of { number: 123 }.

2.9.3 Modifiers

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

html
```html
<input
  type="checkbox"
  v-model="toggle"
  :true-value="dynamicTrueValue"
  :false-value="dynamicFalseValue" />
```

提示

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

单选按钮

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

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

选择器选项

html
```html
<select v-model="selected">
  <!-- 内联对象字面量 -->
  <option :value="{ number: 123 }">123</option>
</select>
```

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

2.9.3 修饰符

.lazy

By default, `v-model` syncs the input with the data after each `input` event (with the exception of IME composition as stated above). You can add the `lazy` modifier to instead sync after `change` events:

html
```html
<!-- 在 "change" 事件后同步更新而不是 "input" -->
<input v-model.lazy="msg" />
```

.number

If you want user input to be automatically typecast as a number, you can add the `number` modifier to your `v-model` managed inputs:

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

If the value cannot be parsed with `parseFloat()`, then the original value is used instead.

The `number` modifier is applied automatically if the input has `type="number"`.

.trim

If you want whitespace from user input to be trimmed automatically, you can add the `trim` modifier to your `v-model`-managed inputs:

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

2.9.4 v-model with Components

> If you're not yet familiar with Vue's components, you can skip this for now.

HTML's built-in input types won't always meet your needs. Fortunately, Vue components allow you to build reusable inputs with completely customized behavior. These inputs even work with `v-model`! To learn more, read about Usage with `v-model` in the Components guide.

.lazy

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

html
```html
<!-- 在 "change" 事件后同步更新而不是 "input" -->
<input v-model.lazy="msg" />
```

.number

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

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

如果该值无法被 `parseFloat()` 处理，那么将返回原始值。

`number` 修饰符会在输入框有 `type="number"` 时自动启用。

.trim

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

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

2.9.4 组件上的 v-model

> 如果你还不熟悉 Vue 的组件，那么现在可以跳过这个部分。

HTML 的内置表单输入类型并不总能满足所有需求。幸运的是，我们可以使用 Vue 构建具有自定义行为的可复用输入组件，并且这些输入组件也支持 `v-model`！要了解更多关于此的内容，请在组件指引中阅读配合 `v-model` 使用。

2.10 Lifecycle Hooks

Each Vue component instance goes through a series of initialization steps when it's created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called lifecycle hooks, giving users the opportunity to add their own code at specific stages.

2.10.1 Registering Lifecycle Hooks

For example, the `onMounted` hook can be used to run code after the component has finished the initial rendering and created the DOM nodes:

html

```html
<script setup>
import { onMounted } from 'vue'
<!-- -->
onMounted(() => {
  console.log(`the component is now mounted.`)
})
</script>
```

There are also other hooks which will be called at different stages of the instance's lifecycle, with the most commonly used being `onMounted`, `onUpdated`, and `onUnmounted`.

When calling `onMounted`, Vue automatically associates the registered callback function with the current active component instance. This requires these hooks to be registered **synchronously** during component setup. For example, do not do this:

js

```js
setTimeout(() => {
  onMounted(() => {
    // 异步注册时当前组件实例已丢失
    // 这将不会正常工作
  })
}, 100)
```

Do note this doesn't mean that the call must be placed lexically inside `setup()` or `<script setup>`. `onMounted()` can be called in an external function as long as the call stack is synchronous and

2.10 生命周期钩子

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

2.10.1 注册周期钩子

举例来说，`onMounted` 钩子可以用来在组件完成初始渲染并创建 DOM 节点后运行代码：

html

```html
<script setup>
import { onMounted } from 'vue'
<!-- -->
onMounted(() => {
  console.log(`the component is now mounted.`)
})
</script>
```

还有其他一些钩子，会在实例生命周期的不同阶段被调用，最常用的是 `onMounted`、`onUpdated` 和 `onUnmounted`。所有生命周期钩子的完整参考及其用法请参考 API 索引。

当调用 `onMounted` 时，Vue 会自动将回调函数注册到当前正被初始化的组件实例上。这意味着这些钩子应当在组件初始化时被**同步**注册。例如，请不要这样做：

js

```js
setTimeout(() => {
  onMounted(() => {
    // 异步注册时当前组件实例已丢失
    // 这将不会正常工作
  })
}, 100)
```

注意这并不意味着对 `onMounted` 的调用必须放在 `setup()` 或 `<script setup>` 内的词法上下文中。`onMounted()` 也可以在一个外部函数中调用，只要调用栈是同

originates from within `setup()`.

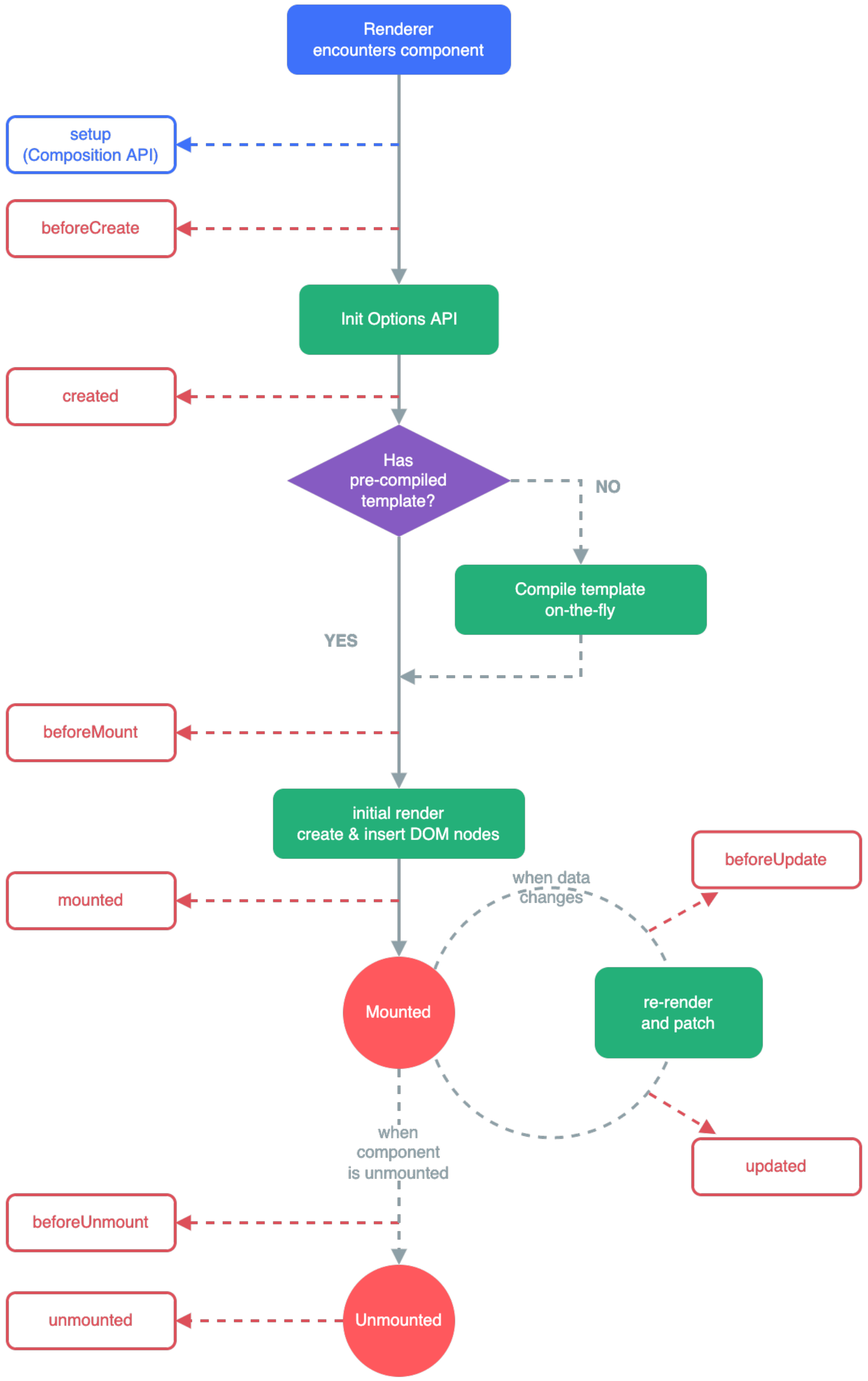
2.10.2 Lifecycle Diagram

Below is a diagram for the instance lifecycle. You don't need to fully understand everything going on right now, but as you learn and build more, it will be a useful reference.

步的，且最终起源自 `setup()` 就可以。

2.10.2 生命周期图示

下面是实例生命周期的图表。你现在并不需要完全理解图中的所有内容，但以后它将是一个有用的参考。


Renderer encounters component
setup (Composition API)
beforeCreate
Init Options API
created
Has pre-compiled template?
NO
Compile template on-the-fly
YES
beforeMount
initial render create & insert DOM nodes
mounted
when data changes
beforeUpdate
re-render and patch
Mounted
updated
when component is unmounted
beforeUnmount
unmounted
Unmounted

Consult the Lifecycle Hooks API reference for details on all lifecycle hooks and their respective use cases.

2.11 Watchers

2.11.1 Basic Example

Computed properties allow us to declaratively compute derived values. However, there are cases where we need to perform ”side effects” in reaction to state changes - for example, mutating the DOM, or changing another piece of state based on the result of an async operation.

With Composition API, we can use the watch function to trigger a callback whenever a piece of reactive state changes:

html

```html
<script setup>
import { ref, watch } from 'vue'
//
const question = ref('')
const answer = ref('Questions usually contain a question mark. ;-)')
//
// 可以直接侦听一个 ref
watch(question, async (newQuestion, oldQuestion) => {
  if (newQuestion.indexOf('?') > -1) {
    answer.value = 'Thinking...'
    try {
      const res = await fetch('https://yesno.wtf/api')
      answer.value = (await res.json()).answer
    } catch (error) {
      answer.value = 'Error! Could not reach the API. ' + error
    }
  }
})
</script>
//
<template>
  <p>
```

有关所有生命周期钩子及其各自用例的详细信息，请参考生命周期钩子 API 索引。

2.11 侦听器

2.11.1 基本示例

计算属性允许我们声明性地计算衍生值。然而在有些情况下，我们需要在状态变化时执行一些“副作用”：例如更改 DOM，或是根据异步操作的结果去修改另一处的状态。

在组合式 API 中，我们可以使用 watch 函数在每次响应式状态发生变化时触发回调函数：

html

```html
<script setup>
import { ref, watch } from 'vue'
//
const question = ref('')
const answer = ref('Questions usually contain a question mark. ;-)')
//
// 可以直接侦听一个 ref
watch(question, async (newQuestion, oldQuestion) => {
  if (newQuestion.indexOf('?') > -1) {
    answer.value = 'Thinking...'
    try {
      const res = await fetch('https://yesno.wtf/api')
      answer.value = (await res.json()).answer
    } catch (error) {
      answer.value = 'Error! Could not reach the API. ' + error
    }
  }
})
</script>
//
<template>
  <p>
```

```
    Ask a yes/no question:
    <input v-model="question" />
  </p>
  <p>{{ answer }}</p>
</template>
```

Try it in the Playground

Watch Source Types

`watch`'s first argument can be different types of reactive "sources": it can be a ref (including computed refs), a reactive object, a getter function, or an array of multiple sources:

```js
const x = ref(0)
const y = ref(0)
//
// 单个 ref
watch(x, (newX) => {
  console.log(`x is ${newX}`)
})
//
// getter 函数
watch(
  () => x.value + y.value,
  (sum) => {
    console.log(`sum of x + y is: ${sum}`)
  }
)
//
// 多个来源组成的数组
watch([x, () => y.value], ([newX, newY]) => {
  console.log(`x is ${newX} and y is ${newY}`)
})
```

Do note that you can't watch a property of a reactive object like this:

```
    Ask a yes/no question:
    <input v-model="question" />
  </p>
  <p>{{ answer }}</p>
</template>
```

在演练场中尝试一下

侦听数据源类型

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

```js
const x = ref(0)
const y = ref(0)
//
// 单个 ref
watch(x, (newX) => {
  console.log(`x is ${newX}`)
})
//
// getter 函数
watch(
  () => x.value + y.value,
  (sum) => {
    console.log(`sum of x + y is: ${sum}`)
  }
)
//
// 多个来源组成的数组
watch([x, () => y.value], ([newX, newY]) => {
  console.log(`x is ${newX} and y is ${newY}`)
})
```

注意，你不能直接侦听响应式对象的属性值，例如:

js

```js
const obj = reactive({ count: 0 })
//
// 错误，因为 watch() 得到的参数是一个 number
watch(obj.count, (count) => {
  console.log(`count is: ${count}`)
})
```

Instead, use a getter:

js

```js
// 提供一个 getter 函数
watch(
  () => obj.count,
  (count) => {
    console.log(`count is: ${count}`)
  }
)
```

2.11.2 Deep Watchers

When you call `watch()` directly on a reactive object, it will implicitly create a deep watcher - the callback will be triggered on all nested mutations:

js

```js
const obj = reactive({ count: 0 })
//
watch(obj, (newValue, oldValue) => {
    // 在嵌套的属性变更时触发
    // 注意：`newValue` 此处和 `oldValue` 是相等的
    // 因为它们是同一个对象！
})
//
obj.count++
```

This should be differentiated with a getter that returns a reactive object - in the latter case, the callback will only fire if the getter returns a different object:

js

```js
const obj = reactive({ count: 0 })
//
// 错误，因为 watch() 得到的参数是一个 number
watch(obj.count, (count) => {
  console.log(`count is: ${count}`)
})
```

这里需要用一个返回该属性的 getter 函数：

js

```js
// 提供一个 getter 函数
watch(
  () => obj.count,
  (count) => {
    console.log(`count is: ${count}`)
  }
)
```

2.11.2 深层侦听器

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

js

```js
const obj = reactive({ count: 0 })
//
watch(obj, (newValue, oldValue) => {
    // 在嵌套的属性变更时触发
    // 注意：`newValue` 此处和 `oldValue` 是相等的
    // 因为它们是同一个对象！
})
//
obj.count++
```

相比之下，一个返回响应式对象的 getter 函数，只有在返回不同的对象时，才会触发回调：

```js
watch(
  () => state.someObject,
  () => {
    // 仅当 state.someObject 被替换时触发
  }
)
```

You can, however, force the second case into a deep watcher by explicitly using the `deep` option:

```js
watch(
  () => state.someObject,
  (newValue, oldValue) => {
    // 注意：`newValue` 此处和 `oldValue` 是相等的
    // * 除非 * state.someObject 被整个替换了
  },
  { deep: true }
)
```

Use with Caution

Deep watch requires traversing all nested properties in the watched object, and can be expensive when used on large data structures. Use it only when necessary and beware of the performance implications.

2.11.3 Eager Watchers

`watch` is lazy by default: the callback won't be called until the watched source has changed. But in some cases we may want the same callback logic to be run eagerly - for example, we may want to fetch some initial data, and then re-fetch the data whenever relevant state changes.

We can force a watcher's callback to be executed immediately by passing the `immediate: true` option:

```js
watch(source, (newValue, oldValue) => {
    // 立即执行，且当 `source` 改变时再次执行
}, { immediate: true })
```

```js
watch(
  () => state.someObject,
  () => {
    // 仅当 state.someObject 被替换时触发
  }
)
```

你也可以给上面这个例子显式地加上 `deep` 选项，强制转成深层侦听器：

```js
watch(
  () => state.someObject,
  (newValue, oldValue) => {
    // 注意：`newValue` 此处和 `oldValue` 是相等的
    // * 除非 * state.someObject 被整个替换了
  },
  { deep: true }
)
```

谨慎使用

深度侦听需要遍历被侦听对象中的所有嵌套的属性，当用于大型数据结构时，开销很大。因此请只在必要时才使用它，并且要留意性能。

2.11.3 即时回调的侦听器

`watch` 默认是懒执行的：仅当数据源变化时，才会执行回调。但在某些场景中，我们希望在创建侦听器时，立即执行一遍回调。举例来说，我们想请求一些初始数据，然后在相关状态更改时重新请求数据。

我们可以通过传入 `immediate: true` 选项来强制侦听器的回调立即执行：

```js
watch(source, (newValue, oldValue) => {
    // 立即执行，且当 `source` 改变时再次执行
}, { immediate: true })
```

2.11.4 watchEffect()

It is common for the watcher callback to use exactly the same reactive state as the source. For example, consider the following code, which uses a watcher to load a remote resource whenever the `todoId` ref changes:

```js
const todoId = ref(1)
const data = ref(null)
//
watch(todoId, async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
}, { immediate: true })
```

In particular, notice how the watcher uses `todoId` twice, once as the source and then again inside the callback.

This can be simplified with `watchEffect()`. `watchEffect()` allows us to track the callback's reactive dependencies automatically. The watcher above can be rewritten as:

```js
watchEffect(async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
})
```

Here, the callback will run immediately, there's no need to specify `immediate: true`. During its execution, it will automatically track `todoId.value` as a dependency (similar to computed properties). Whenever `todoId.value` changes, the callback will be run again. With `watchEffect()`, we no longer need to pass `todoId` explicitly as the source value.

You can check out this example of `watchEffect()` and reactive data-fetching in action.

For examples like these, with only one dependency, the benefit of `watchEffect()` is relatively small. But for watchers that have multiple dependencies, using `watchEffect()` removes the burden

2.11.4 watchEffect()

侦听器的回调使用与源完全相同的响应式状态是很常见的。例如下面的代码，在每当 `todoId` 的引用发生变化时使用侦听器来加载一个远程资源：

```js
const todoId = ref(1)
const data = ref(null)
//
watch(todoId, async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
}, { immediate: true })
```

特别是注意侦听器是如何两次使用 `todoId` 的，一次是作为源，另一次是在回调中。

我们可以用 `watchEffect` 函数 来简化上面的代码。`watchEffect()` 允许我们自动跟踪回调的响应式依赖。上面的侦听器可以重写为：

```js
watchEffect(async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
})
```

这个例子中，回调会立即执行，不需要指定 `immediate: true`。在执行期间，它会自动追踪 `todoId.value` 作为依赖（和计算属性类似）。每当 `todoId.value` 变化时，回调会再次执行。有了 `watchEffect()`，我们不再需要明确传递 `todoId` 作为源值。

你可以参考一下这个例子的 `watchEffect` 和响应式的数据请求的操作。

对于这种只有一个依赖项的例子来说，`watchEffect()` 的好处相对较小。但是对于有多个依赖项的侦听器来说，使用 `watchEffect()` 可以消除手动维护依赖列表

of having to maintain the list of dependencies manually. In addition, if you need to watch several properties in a nested data structure, `watchEffect()` may prove more efficient than a deep watcher, as it will only track the properties that are used in the callback, rather than recursively tracking all of them.

> **TIP**
> `watchEffect` only tracks dependencies during its **synchronous** execution. When using it with an async callback, only properties accessed before the first `await` tick will be tracked.

watch vs. watchEffect

`watch` and `watchEffect` both allow us to reactively perform side effects. Their main difference is the way they track their reactive dependencies:

- `watch` only tracks the explicitly watched source. It won't track anything accessed inside the callback. In addition, the callback only triggers when the source has actually changed. `watch` separates dependency tracking from the side effect, giving us more precise control over when the callback should fire.

- `watchEffect`, on the other hand, combines dependency tracking and side effect into one phase. It automatically tracks every reactive property accessed during its synchronous execution. This is more convenient and typically results in terser code, but makes its reactive dependencies less explicit.

2.11.5 Callback Flush Timing

When you mutate reactive state, it may trigger both Vue component updates and watcher callbacks created by you.

By default, user-created watcher callbacks are called **before** Vue component updates. This means if you attempt to access the DOM inside a watcher callback, the DOM will be in the state before Vue has applied any updates.

If you want to access the DOM in a watcher callback **after** Vue has updated it, you need to specify the `flush: 'post'` option:

```js
watch(source, callback, {
```

的负担。此外，如果你需要侦听一个嵌套数据结构中的几个属性，`watchEffect()` 可能会比深度侦听器更有效，因为它将只跟踪回调中被使用到的属性，而不是递归地跟踪所有的属性。

> **TIP**
> `watchEffect` 仅会在其**同步**执行期间，才追踪依赖。在使用异步回调时，只有在第一个 `await` 正常工作前访问到的属性才会被追踪。

watch vs. watchEffect

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

- `watch` 只追踪明确侦听的数据源。它不会追踪任何在回调中访问到的东西。另外，仅在数据源确实改变时才会触发回调。`watch` 会避免在发生副作用时追踪依赖，因此，我们能更加精确地控制回调函数的触发时机。

- `watchEffect`，则会在副作用发生期间追踪依赖。它会在同步执行过程中，自动追踪所有能访问到的响应式属性。这更方便，而且代码往往更简洁，但有时其响应性依赖关系会不那么明确。

2.11.5 回调的触发时机

当你更改了响应式状态，它可能会同时触发 Vue 组件更新和侦听器回调。

默认情况下，用户创建的侦听器回调，都会在 Vue 组件更新**之前**被调用。这意味着你在侦听器回调中访问的 DOM 将是被 Vue 更新之前的状态。

如果想在侦听器回调中能访问被 Vue 更新**之后**的 DOM，你需要指明 `flush: 'post'` 选项：

```js
watch(source, callback, {
  flush: 'post'
})
//
watchEffect(callback, {
  flush: 'post'
})
```

Post-flush `watchEffect()` also has a convenience alias, `watchPostEffect()`:

js

```
import { watchPostEffect } from 'vue'
//
watchPostEffect(() => {
  /* 在 Vue 更新后执行 */
})
```

2.11.6 Stopping a Watcher

Watchers declared synchronously inside `setup()` or `<script setup>` are bound to the owner component instance, and will be automatically stopped when the owner component is unmounted. In most cases, you don't need to worry about stopping the watcher yourself.

The key here is that the watcher must be created **synchronously**: if the watcher is created in an async callback, it won't be bound to the owner component and must be stopped manually to avoid memory leaks. Here's an example:

html

```
<script setup>
import { watchEffect } from 'vue'
//
// 它会自动停止
watchEffect(() => {})
//
// ... 这个则不会!
setTimeout(() => {
    watchEffect(() => {})
}, 100)
</script>
```

```
  flush: 'post'
})
//
watchEffect(callback, {
  flush: 'post'
})
```

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

js

```
import { watchPostEffect } from 'vue'
//
watchPostEffect(() => {
  /* 在 Vue 更新后执行 */
})
```

2.11.6 停止侦听器

在 `setup()` 或 `<script setup>` 中用同步语句创建的侦听器，会自动绑定到宿主组件实例上，并且会在宿主组件卸载时自动停止。因此，在大多数情况下，你无需关心怎么停止一个侦听器。

一个关键点是，侦听器必须用**同步**语句创建：如果用异步回调创建一个侦听器，那么它不会绑定到当前组件上，你必须手动停止它，以防内存泄漏。如下方这个例子：

html

```
<script setup>
import { watchEffect } from 'vue'
//
// 它会自动停止
watchEffect(() => {})
//
// ... 这个则不会!
setTimeout(() => {
    watchEffect(() => {})
}, 100)
</script>
```

To manually stop a watcher, use the returned handle function. This works for both `watch` and `watchEffect`:

```js
const unwatch = watchEffect(() => {})
//
// ... 当该侦听器不再需要时
unwatch()
```

Note that there should be very few cases where you need to create watchers asynchronously, and synchronous creation should be preferred whenever possible. If you need to wait for some async data, you can make your watch logic conditional instead:

```js
// 需要异步请求得到的数据
const data = ref(null)
//
watchEffect(() => {
  if (data.value) {
    // 数据加载后执行某些操作...
  }
})
```

2.12 Template Refs

While Vue's declarative rendering model abstracts away most of the direct DOM operations for you, there may still be cases where we need direct access to the underlying DOM elements. To achieve this, we can use the special `ref` attribute:

```html
<input ref="input">
```

`ref` is a special attribute, similar to the `key` attribute discussed in the `v-for` chapter. It allows us to obtain a direct reference to a specific DOM element or child component instance after it's mounted. This may be useful when you want to, for example, programmatically focus an input on component mount, or initialize a 3rd party library on an element.

2.12.1 Accessing the Refs

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

```js
const unwatch = watchEffect(() => {})
//
// ... 当该侦听器不再需要时
unwatch()
```

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

```js
// 需要异步请求得到的数据
const data = ref(null)
//
watchEffect(() => {
  if (data.value) {
    // 数据加载后执行某些操作...
  }
})
```

2.12 模板引用

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

```html
<input ref="input">
```

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

2.12.1 访问模板引用

To obtain the reference with Composition API, we need to declare a ref with the same name:

```html
<script setup>
import { ref, onMounted } from 'vue'
//
// 声明一个 ref 来存放该元素的引用
// 必须和模板里的 ref 同名
const input = ref(null)
//
onMounted(() => {
  input.value.focus()
})
</script>
//
<template>
  <input ref="input" />
</template>
```

If not using `<script setup>`, make sure to also return the ref from `setup()`:

```js
export default {
  setup() {
    const input = ref(null)
    // ...
    return {
      input
    }
  }
}
```

Note that you can only access the ref **after the component is mounted.** If you try to access `input` in a template expression, it will be `null` on the first render. This is because the element doesn't exist until after the first render!

If you are trying to watch the changes of a template ref, make sure to account for the case where the ref has `null` value:

```js
watchEffect(() => {
```

为了通过组合式 API 获得该模板引用，我们需要声明一个同名的 ref：

```html
<script setup>
import { ref, onMounted } from 'vue'
//
// 声明一个 ref 来存放该元素的引用
// 必须和模板里的 ref 同名
const input = ref(null)
//
onMounted(() => {
  input.value.focus()
})
</script>
//
<template>
  <input ref="input" />
</template>
```

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

```js
export default {
  setup() {
    const input = ref(null)
    // ...
    return {
      input
    }
  }
}
```

注意，你只可以**在组件挂载后**才能访问模板引用。如果你想在模板中的表达式上访问 `input`，在初次渲染时会是 `null`。这是因为在初次渲染前这个元素还不存在呢！

如果你需要侦听一个模板引用 ref 的变化，确保考虑到其值为 `null` 的情况：

```js
watchEffect(() => {
  if (input.value) {
    input.value.focus()
  } else {
    // 此时还未挂载，或此元素已经被卸载（例如通过 v-if 控制）
  }
})
```

See also: Typing Template Refs

2.12.2 Refs inside v-for

> Requires v3.2.25 or above

When `ref` is used inside `v-for`, the corresponding ref should contain an Array value, which will be populated with the elements after mount:

html

```
<script setup>
import { ref, onMounted } from 'vue'
//
const list = ref([
  /* ... */
])
//
const itemRefs = ref([])
//
onMounted(() => console.log(itemRefs.value))
</script>
<!-- -->
<template>
  <ul>
    <li v-for="item in list" ref="itemRefs">
      {{ item }}
    </li>
  </ul>
</template>
```

```
  if (input.value) {
    input.value.focus()
  } else {
    // 此时还未挂载，或此元素已经被卸载（例如通过 v-if 控制）
  }
})
```

也可参考：为模板引用标注类型

2.12.2 v-for 中的模板引用

> 需要 v3.2.25 及以上版本

当在 `v-for` 中使用模板引用时，对应的 ref 中包含的值是一个数组，它将在元素被挂载后包含对应整个列表的所有元素：

html

```
<script setup>
import { ref, onMounted } from 'vue'
//
const list = ref([
  /* ... */
])
//
const itemRefs = ref([])
//
onMounted(() => console.log(itemRefs.value))
</script>
<!-- -->
<template>
  <ul>
    <li v-for="item in list" ref="itemRefs">
      {{ item }}
    </li>
  </ul>
</template>
```

Try it in the Playground

It should be noted that the ref array does **not** guarantee the same order as the source array.

2.12.3 Function Refs

Instead of a string key, the `ref` attribute can also be bound to a function, which will be called on each component update and gives you full flexibility on where to store the element reference. The function receives the element reference as the first argument:

```html
<input :ref="(el) => { /* 将 el 赋值给一个数据属性或 ref 变量 */ }">
```

Note we are using a dynamic `:ref` binding so we can pass it a function instead of a ref name string. When the element is unmounted, the argument will be `null`. You can, of course, use a method instead of an inline function.

2.12.4 Ref on Component

> This section assumes knowledge of Components. Feel free to skip it and come back later.

`ref` can also be used on a child component. In this case the reference will be that of a component instance:

```html
<script setup>
import { ref, onMounted } from 'vue'
import Child from './Child.vue'
<!-- -->
const child = ref(null)
<!-- -->
onMounted(() => {
  // child.value 是 <Child /> 组件的实例
})
</script>
<!-- -->
<template>
  <Child ref="child" />
```

在演练场中尝试一下

应该注意的是，ref 数组**并不**保证与源数组相同的顺序。

2.12.3 函数模板引用

除了使用字符串值作名字，`ref` attribute 还可以绑定为一个函数，会在每次组件更新时都被调用。该函数会收到元素引用作为其第一个参数：

```html
<input :ref="(el) => { /* 将 el 赋值给一个数据属性或 ref 变量 */ }">
```

注意我们这里需要使用动态的 `:ref` 绑定才能够传入一个函数。当绑定的元素被卸载时，函数也会被调用一次，此时的 `el` 参数会是 `null`。你当然也可以绑定一个组件方法而不是内联函数。

2.12.4 组件上的 ref

> 这一小节假设你已了解组件的相关知识，或者你也可以先跳过这里，之后再回来看。

模板引用也可以被用在一个子组件上。这种情况下引用中获得的值是组件实例：

```html
<script setup>
import { ref, onMounted } from 'vue'
import Child from './Child.vue'
<!-- -->
const child = ref(null)
<!-- -->
onMounted(() => {
  // child.value 是 <Child /> 组件的实例
})
</script>
<!-- -->
<template>
  <Child ref="child" />
```

```
</template>
```

If the child component is using Options API or not using `<script setup>`, the referenced instance will be identical to the child component's `this`, which means the parent component will have full access to every property and method of the child component. This makes it easy to create tightly coupled implementation details between the parent and the child, so component refs should be only used when absolutely needed - in most cases, you should try to implement parent / child interactions using the standard props and emit interfaces first.

An exception here is that components using `<script setup>` are **private by default**: a parent component referencing a child component using `<script setup>` won't be able to access anything unless the child component chooses to expose a public interface using the `defineExpose` macro:

html

```html
<script setup>
import { ref } from 'vue'
<!-- -->
const a = 1
const b = ref(2)
<!-- -->
// 像 defineExpose 这样的编译器宏不需要导入
defineExpose({
  a,
  b
})
</script>
```

When a parent gets an instance of this component via template refs, the retrieved instance will be of the shape `{ a: number, b: number }` (refs are automatically unwrapped just like on normal instances).

See also: Typing Component Template Refs

2.13 Components Basics

Components allow us to split the UI into independent and reusable pieces, and think about each piece in isolation. It's common for an app to be organized into a tree of nested components:

```
</template>
```

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

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

html

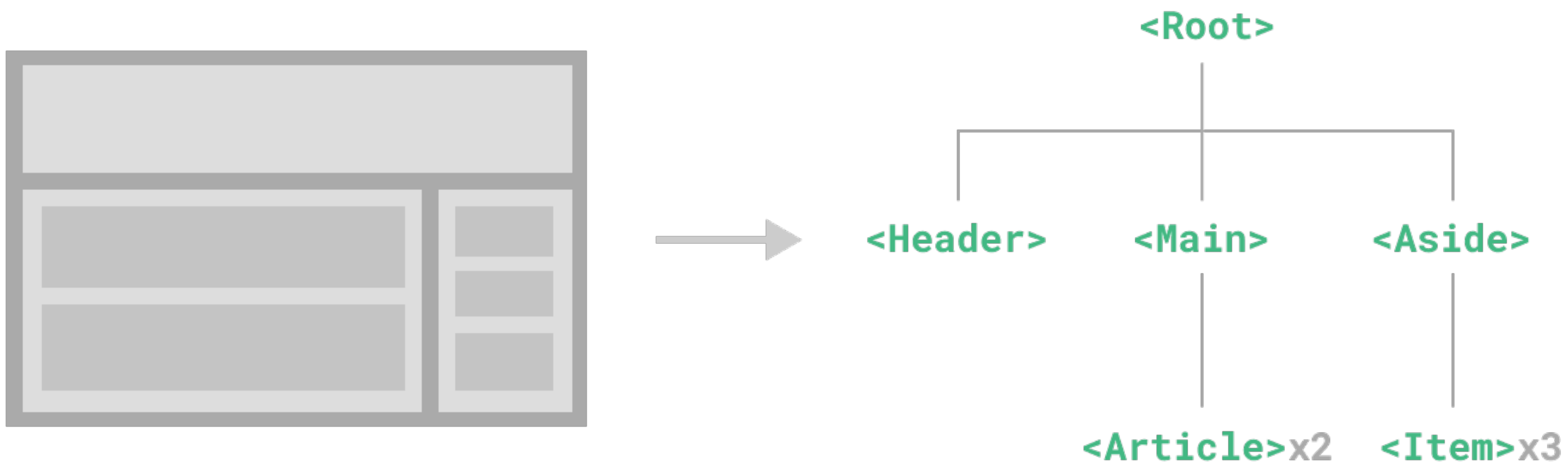
```html
<script setup>
import { ref } from 'vue'
<!-- -->
const a = 1
const b = ref(2)
<!-- -->
// 像 defineExpose 这样的编译器宏不需要导入
defineExpose({
  a,
  b
})
</script>
```

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

TypeScript 用户请参考：为组件的模板引用标注类型

2.13 组件基础

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



This is very similar to how we nest native HTML elements, but Vue implements its own component model that allow us to encapsulate custom content and logic in each component. Vue also plays nicely with native Web Components. If you are curious about the relationship between Vue Components and native Web Components, read more here.

2.13.1 Defining a Component

When using a build step, we typically define each Vue component in a dedicated file using the .vue extension - known as a Single-File Component (SFC for short):

```html
<script setup>
import { ref } from 'vue'
//
const count = ref(0)
</script>
//
<template>
    <button @click="count++">You clicked me {{ count }} times.</button>
```

这和我们嵌套 HTML 元素的方式类似，Vue 实现了自己的组件模型，使我们可以在每个组件内封装自定义内容与逻辑。Vue 同样也能很好地配合原生 Web Components。如果你想知道 Vue 组件与原生 Web Components 之间的关系，可以阅读此章节。

2.13.1 定义一个组件

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

```html
<script setup>
import { ref } from 'vue'
//
const count = ref(0)
</script>
//
<template>
    <button @click="count++">You clicked me {{ count }} times.</button>
```

```
</template>
```

When not using a build step, a Vue component can be defined as a plain JavaScript object containing Vue-specific options:

```js
import { ref } from 'vue'
//
export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  template: `
    <button @click="count++">
      You clicked me {{ count }} times.
    </button>`
  // 也可以针对一个 DOM 内联模板:
  // template: '#my-template-element'
}
```

The template is inlined as a JavaScript string here, which Vue will compile on the fly. You can also use an ID selector pointing to an element (usually native `<template>` elements) - Vue will use its content as the template source.

The example above defines a single component and exports it as the default export of a `.js` file, but you can use named exports to export multiple components from the same file.

2.13.2 Using a Component

> **TIP**
> We will be using SFC syntax for the rest of this guide - the concepts around components are the same regardless of whether you are using a build step or not. The Examples section shows component usage in both scenarios.

To use a child component, we need to import it in the parent component. Assuming we placed our counter component inside a file called `ButtonCounter.vue`, the component will be exposed as the

```
</template>
```

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

```js
import { ref } from 'vue'
//
export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  template: `
    <button @click="count++">
      You clicked me {{ count }} times.
    </button>`
  // 也可以针对一个 DOM 内联模板:
  // template: '#my-template-element'
}
```

这里的模板是一个内联的 JavaScript 字符串，Vue 将会在运行时编译它。你也可以使用 ID 选择器来指向一个元素 (通常是原生的 `<template>` 元素)，Vue 将会使用其内容作为模板来源。

上面的例子中定义了一个组件，并在一个 `.js` 文件里默认导出了它自己，但你也可以通过具名导出在一个文件中导出多个组件。

2.13.2 使用组件

> **TIP**
> 我们会在接下来的指引中使用 SFC 语法，无论你是否使用构建步骤，组件相关的概念都是相同的。示例一节中展示了两种场景中的组件使用情况。

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

file's default export:

html

```html
<script setup>
import ButtonCounter from './ButtonCounter.vue'
</script>
<!-- -->
<template>
  <h1>Here is a child component!</h1>
  <ButtonCounter />
</template>
```

With `<script setup>`, imported components are automatically made available to the template.

It's also possible to globally register a component, making it available to all components in a given app without having to import it. The pros and cons of global vs. local registration is discussed in the dedicated Component Registration section.

Components can be reused as many times as you want:

html

```html
<h1>Here is a child component!</h1>
<ButtonCounter />
<ButtonCounter />
<ButtonCounter />
```

Try it in the Playground

Notice that when clicking on the buttons, each one maintains its own, separate `count`. That's because each time you use a component, a new **instance** of it is created.

In SFCs, it's recommended to use `PascalCase` tag names for child components to differentiate from native HTML elements. Although native HTML tag names are case-insensitive, Vue SFC is a compiled format so we are able to use case-sensitive tag names in it. We are also able to use `/>` to close a tag.

If you are authoring your templates directly in a DOM (e.g. as the content of a native `<template>` element), the template will be subject to the browser's native HTML parsing behavior. In such cases, you will need to use `kebab-case` and explicit closing tags for components:

html

```html
<!-- 如果是在 DOM 中书写该模板 -->
<button-counter></button-counter>
```

给外部。

html

```html
<script setup>
import ButtonCounter from './ButtonCounter.vue'
</script>
<!-- -->
<template>
  <h1>Here is a child component!</h1>
  <ButtonCounter />
</template>
```

通过 `<script setup>`，导入的组件都在模板中直接可用。

当然，你也可以全局地注册一个组件，使得它在当前应用中的任何组件上都可以使用，而不需要额外再导入。关于组件的全局注册和局部注册两种方式的利弊，我们放在了组件注册这一章节中专门讨论。

组件可以被重用任意多次：

html

```html
<h1>Here is a child component!</h1>
<ButtonCounter />
<ButtonCounter />
<ButtonCounter />
```

在演练场中尝试一下

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

在单文件组件中,推荐为子组件使用 `PascalCase` 的标签名,以此来和原生的 HTML 元素作区分。虽然原生 HTML 标签名是不区分大小写的，但 Vue 单文件组件是可以在编译中区分大小写的。我们也可以使用 `/>` 来关闭一个标签。

如果你是直接在 DOM 中书写模板 (例如原生 `<template>` 元素的内容)，模板的编译需要遵从浏览器中 HTML 的解析行为。在这种情况下，你应该需要使用 `kebab-case` 形式并显式地关闭这些组件的标签。

html

```html
<!-- 如果是在 DOM 中书写该模板 -->
<button-counter></button-counter>
```

```
<button-counter></button-counter>
<button-counter></button-counter>
```

See in-DOM template parsing caveats for more details.

2.13.3 Passing Props

If we are building a blog, we will likely need a component representing a blog post. We want all the blog posts to share the same visual layout, but with different content. Such a component won't be useful unless you can pass data to it, such as the title and content of the specific post we want to display. That's where props come in.

Props are custom attributes you can register on a component. To pass a title to our blog post component, we must declare it in the list of props this component accepts, using the `defineProps` macro:

```html
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
</script>
<!-- -->
<template>
  <h4>{{ title }}</h4>
</template>
```

`defineProps` is a compile-time macro that is only available inside `<script setup>` and does not need to be explicitly imported. Declared props are automatically exposed to the template. `defineProps` also returns an object that contains all the props passed to the component, so that we can access them in JavaScript if needed:

```js
const props = defineProps(['title'])
console.log(props.title)
```

See also: Typing Component Props

If you are not using `<script setup>`, props should be declared using the `props` option, and the props object will be passed to `setup()` as the first argument:

```
<button-counter></button-counter>
<button-counter></button-counter>
```

请看 DOM 内模板解析注意事项了解更多细节。

2.13.3 传递 props

如果我们正在构建一个博客，我们可能需要一个表示博客文章的组件。我们希望所有的博客文章分享相同的视觉布局，但有不同的内容。要实现这样的效果自然必须向组件中传递数据，例如每篇文章标题和内容，这就会使用到 props。

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

```html
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
</script>
<!-- -->
<template>
  <h4>{{ title }}</h4>
</template>
```

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

```js
const props = defineProps(['title'])
console.log(props.title)
```

TypeScript 用户请参考：为组件 props 标注类型

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

```js
export default {
  props: ['title'],
  setup(props) {
    console.log(props.title)
  }
}
```

A component can have as many props as you like and, by default, any value can be passed to any prop.

Once a prop is registered, you can pass data to it as a custom attribute, like this:

```html
<BlogPost title="My journey with Vue" />
<BlogPost title="Blogging with Vue" />
<BlogPost title="Why Vue is so fun" />
```

In a typical app, however, you'll likely have an array of posts in your parent component:

```js
const posts = ref([
  { id: 1, title: 'My journey with Vue' },
  { id: 2, title: 'Blogging with Vue' },
  { id: 3, title: 'Why Vue is so fun' }
])
```

Then want to render a component for each one, using `v-for`:

```html
<BlogPost
  v-for="post in posts"
  :key="post.id"
  :title="post.title"
 />
```

Try it in the Playground

Notice how `v-bind` is used to pass dynamic prop values. This is especially useful when you don't know the exact content you're going to render ahead of time.

That's all you need to know about props for now, but once you've finished reading this page and

```js
export default {
  props: ['title'],
  setup(props) {
    console.log(props.title)
  }
}
```

一个组件可以有任意多的 props，默认情况下，所有 prop 都接受任意类型的值。

当一个 prop 被注册后，可以像这样以自定义 attribute 的形式传递数据给它：

```html
<BlogPost title="My journey with Vue" />
<BlogPost title="Blogging with Vue" />
<BlogPost title="Why Vue is so fun" />
```

在实际应用中，我们可能在父组件中会有如下的一个博客文章数组：

```js
const posts = ref([
  { id: 1, title: 'My journey with Vue' },
  { id: 2, title: 'Blogging with Vue' },
  { id: 3, title: 'Why Vue is so fun' }
])
```

这种情况下，我们可以使用 `v-for` 来渲染它们：

```html
<BlogPost
  v-for="post in posts"
  :key="post.id"
  :title="post.title"
 />
```

在演练场中尝试一下

留意我们是如何使用 `v-bind` 来传递动态 prop 值的。当事先不知道要渲染的确切内容时，这一点特别有用。

以上就是目前你需要了解的关于 props 的全部了。如果你看完本章节后还想知道

feel comfortable with its content, we recommend coming back later to read the full guide on Props.

2.13.4 Listening to Events

As we develop our `<BlogPost>` component, some features may require communicating back up to the parent. For example, we may decide to include an accessibility feature to enlarge the text of blog posts, while leaving the rest of the page at its default size.

In the parent, we can support this feature by adding a `postFontSize` ref:

```js
const posts = ref([
  /* ... */
])
//
const postFontSize = ref(1)
```

Which can be used in the template to control the font size of all blog posts:

```html
<div :style="{ fontSize: postFontSize + 'em' }">
  <BlogPost
    v-for="post in posts"
    :key="post.id"
    :title="post.title"
   />
</div>
```

Now let's add a button to the `<BlogPost>` component's template:

```html
<!-- BlogPost.vue, 省略了 <script> -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button>Enlarge text</button>
  </div>
</template>
```

The button doesn't do anything yet - we want clicking the button to communicate to the parent

更多细节，我们推荐你深入阅读关于 props 的完整指引。

2.13.4 监听事件

让我们继续关注我们的 `<BlogPost>` 组件。我们会发现有时候它需要与父组件进行交互。例如，要在此处实现无障碍访问的需求，将博客文章的文字能够放大，而页面的其余部分仍使用默认字号。

在父组件中，我们可以添加一个 `postFontSize` ref 来实现这个效果：

```js
const posts = ref([
  /* ... */
])
//
const postFontSize = ref(1)
```

在模板中用它来控制所有博客文章的字体大小：

```html
<div :style="{ fontSize: postFontSize + 'em' }">
  <BlogPost
    v-for="post in posts"
    :key="post.id"
    :title="post.title"
   />
</div>
```

然后，给 `<BlogPost>` 组件添加一个按钮：

```html
<!-- BlogPost.vue, 省略了 <script> -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button>Enlarge text</button>
  </div>
</template>
```

这个按钮目前还没有做任何事情，我们想要点击这个按钮来告诉父组件它应该放

that it should enlarge the text of all posts. To solve this problem, components provide a custom events system. The parent can choose to listen to any event on the child component instance with `v-on` or `@`, just as we would with a native DOM event:

```html
<BlogPost
  ...
  @enlarge-text="postFontSize += 0.1"
 />
```

Then the child component can emit an event on itself by calling the built-in `$emit` method, passing the name of the event:

```html
<!-- BlogPost.vue, 省略了 <script> -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button @click="$emit('enlarge-text')">Enlarge text</button>
  </div>
</template>
```

Thanks to the `@enlarge-text="postFontSize += 0.1"` listener, the parent will receive the event and update the value of `postFontSize`.

Try it in the Playground

We can optionally declare emitted events using the `defineEmits` macro:

```html
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
defineEmits(['enlarge-text'])
</script>
```

This documents all the events that a component emits and optionally validates them. It also allows Vue to avoid implicitly applying them as native listeners to the child component's root element.

Similar to `defineProps`, `defineEmits` is only usable in `<script setup>` and doesn't need to be imported. It returns an `emit` function that is equivalent to the `$emit` method. It can be used to emit events in the `<script setup>` section of a component, where `$emit` isn't directly accessible:

大所有博客文章的文字。要解决这个问题，组件实例提供了一个自定义事件系统。父组件可以通过 `v-on` 或 `@` 来选择性地监听子组件上抛的事件，就像监听原生 DOM 事件那样：

```html
<BlogPost
  ...
  @enlarge-text="postFontSize += 0.1"
 />
```

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

```html
<!-- BlogPost.vue, 省略了 <script> -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button @click="$emit('enlarge-text')">Enlarge text</button>
  </div>
</template>
```

因为有了 `@enlarge-text="postFontSize += 0.1"` 的监听，父组件会接收这一事件，从而更新 `postFontSize` 的值。

在演练场中尝试一下

我们可以通过 `defineEmits` 宏来声明需要抛出的事件：

```html
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
defineEmits(['enlarge-text'])
</script>
```

这声明了一个组件可能触发的所有事件，还可以对事件的参数进行验证。同时，这还可以让 Vue 避免将它们作为原生事件监听器隐式地应用于子组件的根元素。

和 `defineProps` 类似，`defineEmits` 仅可用于 `<script setup>` 之中，并且不需要导入，它返回一个等同于 `$emit` 方法的 `emit` 函数。它可以被用于在组件的 `<script setup>` 中抛出事件，因为此处无法直接访问 `$emit`：

```html
<script setup>
const emit = defineEmits(['enlarge-text'])
//
emit('enlarge-text')
</script>
```

See also: Typing Component Emits

If you are not using `<script setup>`, you can declare emitted events using the `emits` option. You can access the `emit` function as a property of the setup context (passed to `setup()` as the second argument):

```js
export default {
  emits: ['enlarge-text'],
  setup(props, ctx) {
    ctx.emit('enlarge-text')
  }
}
```

That's all you need to know about custom component events for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Custom Events.

2.13.5 Content Distribution with Slots

Just like with HTML elements, it's often useful to be able to pass content to a component, like this:

```html
<AlertBox>
  Something bad happened.
</AlertBox>
```

Which might render something like:

```html
<script setup>
const emit = defineEmits(['enlarge-text'])
//
emit('enlarge-text')
</script>
```

TypeScript 用户请参考：为组件 emits 标注类型

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

```js
export default {
  emits: ['enlarge-text'],
  setup(props, ctx) {
    ctx.emit('enlarge-text')
  }
}
```

以上就是目前你需要了解的关于组件自定义事件的所有知识了。如果你看完本章节后还想知道更多细节，请深入阅读组件事件章节。

2.13.5 通过插槽来分配内容

一些情况下我们会希望能和 HTML 元素一样向组件中传递内容：

```html
<AlertBox>
  Something bad happened.
</AlertBox>
```

我们期望能渲染成这样：

This is an Error for Demo Purposes
Something bad happened.

This can be achieved using Vue's custom `<slot>` element:

```html
<template>
    <div class="alert-box">
    <strong>This is an Error for Demo Purposes</strong>
    <slot />
    </div>
</template>
<style scoped>
.alert-box {
    /* ... */
}
</style>
```

As you'll see above, we use the `<slot>` as a placeholder where we want the content to go – and that's it. We're done!

Try it in the Playground

That's all you need to know about slots for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Slots.

2.13.6 Dynamic Components

Sometimes, it's useful to dynamically switch between components, like in a tabbed interface:

Open example in the Playground

The above is made possible by Vue's `<component>` element with the special `is` attribute:

```html
<!-- currentTab 改变时组件也改变 -->
<component :is="tabs[currentTab]"></component>
```

In the example above, the value passed to `:is` can contain either:

- the name string of a registered component, OR
- the actual imported component object

You can also use the `is` attribute to create regular HTML elements.

这可以通过 Vue 的自定义 `<slot>` 元素来实现：

```html
<template>
    <div class="alert-box">
    <strong>This is an Error for Demo Purposes</strong>
    <slot />
    </div>
</template>
<style scoped>
.alert-box {
    /* ... */
}
</style>
```

如上所示，我们使用 `<slot>` 作为一个占位符，父组件传递进来的内容就会渲染在这里。

在演练场中尝试一下

以上就是目前你需要了解的关于插槽的所有知识了。如果你看完本章节后还想知道更多细节，请深入阅读组件插槽章节。

2.13.6 动态组件

有些场景会需要在两个组件间来回切换，比如 Tab 界面：

在演练场中查看示例

上面的例子是通过 Vue 的 `<component>` 元素和特殊的 `is` attribute 实现的：

```html
<!-- currentTab 改变时组件也改变 -->
<component :is="tabs[currentTab]"></component>
```

在上面的例子中，被传给 `:is` 的值可以是以下几种：

- 被注册的组件名
- 导入的组件对象

你也可以使用 `is` attribute 来创建一般的 HTML 元素。

When switching between multiple components with `<component :is="...">`, a component will be unmounted when it is switched away from. We can force the inactive components to stay "alive" with the built-in `<KeepAlive> component`.

2.13.7 in-DOM Template Parsing Caveats

If you are writing your Vue templates directly in the DOM, Vue will have to retrieve the template string from the DOM. This leads to some caveats due to browsers' native HTML parsing behavior.

> **TIP**
> It should be noted that the limitations discussed below only apply if you are writing your templates directly in the DOM. They do NOT apply if you are using string templates from the following sources:
> - Single-File Components
> - Inlined template strings (e.g. `template: '...'`)
> - `<script type="text/x-template">`

Case Insensitivity

HTML tags and attribute names are case-insensitive, so browsers will interpret any uppercase characters as lowercase. That means when you're using in-DOM templates, PascalCase component names and camelCased prop names or `v-on` event names all need to use their kebab-cased (hyphen-delimited) equivalents:

js

```js
// JavaScript 中的 camelCase
const BlogPost = {
    props: ['postTitle'],
    emits: ['updatePost'],
    template: `
    <h3>{{ postTitle }}</h3>
    `
}
```

html

```html
<!-- HTML 中的 kebab-case -->
<blog-post post-title="hello!" @update-post="onUpdatePost"></blog-post>
```

当使用 `<component :is="...">` 来在多个组件间作切换时，被切换掉的组件会被卸载。我们可以通过 `<KeepAlive>` 组件强制被切换掉的组件仍然保持 “存活” 的状态。

2.13.7 DOM 内模板解析注意事项

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

> **TIP**
> 请注意下面讨论只适用于直接在 DOM 中编写模板的情况。如果你使用来自以下来源的字符串模板，就不需要顾虑这些限制了：
> - 单文件组件
> - 内联模板字符串 (例如 `template: '...'`)
> - `<script type="text/x-template">`

大小写区分

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

js

```js
// JavaScript 中的 camelCase
const BlogPost = {
    props: ['postTitle'],
    emits: ['updatePost'],
    template: `
    <h3>{{ postTitle }}</h3>
    `
}
```

html

```html
<!-- HTML 中的 kebab-case -->
<blog-post post-title="hello!" @update-post="onUpdatePost"></blog-post>
```

Self Closing Tags

We have been using self-closing tags for components in previous code samples:

```html
<MyComponent />
```

This is because Vue's template parser respects `/>` as an indication to end any tag, regardless of its type.

In in-DOM templates, however, we must always include explicit closing tags:

```html
<my-component></my-component>
```

This is because the HTML spec only allows a few specific elements to omit closing tags, the most common being `<input>` and `<img>`. For all other elements, if you omit the closing tag, the native HTML parser will think you never terminated the opening tag. For example, the following snippet:

```html
<my-component /> <!-- 我们想要在这里关闭标签... -->
<span>hello</span>
```

will be parsed as:

```html
<my-component>
  <span>hello</span>
</my-component> <!-- 但浏览器会在这里关闭标签 -->
```

Element Placement Restrictions

Some HTML elements, such as `<ul>`, `<ol>`, `<table>` and `<select>` have restrictions on what elements can appear inside them, and some elements such as `<li>`, `<tr>`, and `<option>` can only appear inside certain other elements.

This will lead to issues when using components with elements that have such restrictions. For example:

```html
<table>
  <blog-post-row></blog-post-row>
</table>
```

闭合标签

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

```html
<MyComponent />
```

这是因为 Vue 的模板解析器支持任意标签使用 `/>` 作为标签关闭的标志。

然而在 DOM 内模板中，我们必须显式地写出关闭标签：

```html
<my-component></my-component>
```

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

```html
<my-component /> <!-- 我们想要在这里关闭标签... -->
<span>hello</span>
```

将被解析为：

```html
<my-component>
  <span>hello</span>
</my-component> <!-- 但浏览器会在这里关闭标签 -->
```

元素位置限制

某些 HTML 元素对于放在其中的元素类型有限制，例如 `<ul>`，`<ol>`，`<table>` 和 `<select>`，相应的，某些元素仅在放置于特定元素中时才会显示，例如 `<li>`，`<tr>` 和 `<option>`。

这将导致在使用带有此类限制元素的组件时出现问题。例如：

```html
<table>
  <blog-post-row></blog-post-row>
</table>
```

The custom component `<blog-post-row>` will be hoisted out as invalid content, causing errors in the eventual rendered output. We can use the special `is` attribute as a workaround:

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

TIP

When used on native HTML elements, the value of `is` must be prefixed with `vue:` in order to be interpreted as a Vue component. This is required to avoid confusion with native customized built-in elements.

That's all you need to know about in-DOM template parsing caveats for now - and actually, the end of Vue's *Essentials*. Congratulations! There's still more to learn, but first, we recommend taking a break to play with Vue yourself - build something fun, or check out some of the Examples if you haven't already.

Once you feel comfortable with the knowledge you've just digested, move on with the guide to learn more about components in depth.

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

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

TIP

当使用在原生 HTML 元素上时，`is` 的值必须加上前缀 `vue:` 才可以被解析为一个 Vue 组件。这一点是必要的，为了避免和原生的自定义内置元素相混淆。

以上就是你需要了解的关于 DOM 内模板解析的所有注意事项，同时也是 Vue *基础*部分的所有内容。祝贺你！虽然还有很多需要学习的，但你可以先暂停一下，去用 Vue 做一些有趣的东西，或者研究一些示例。

完成了本页的阅读后，回顾一下你刚才所学到的知识，如果还想知道更多细节，我们推荐你继续阅读关于组件的完整指引。

第三章 Components In-Depth

3.1 Component Registration

> This page assumes you've already read the Components Basics. Read that first if you are new to components.

A Vue component needs to be "registered" so that Vue knows where to locate its implementation when it is encountered in a template. There are two ways to register components: global and local.

3.1.1 Global Registration

We can make components available globally in the current Vue application using the `.component()` method:

```js
import { createApp } from 'vue'
//
const app = createApp({})
//
app.component(
  // 注册的名字
  'MyComponent',
  // 组件的实现
  {
    /* ... */
  }
)
```

深入组件

3.1 组件注册

> 此章节假设你已经看过了组件基础。若你还不了解组件是什么，请先阅读该章节。

一个 Vue 组件在使用前需要先被“注册”，这样 Vue 才能在渲染模板时找到其对应的实现。组件注册有两种方式：全局注册和局部注册。

3.1.1 全局注册

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

```js
import { createApp } from 'vue'
//
const app = createApp({})
//
app.component(
  // 注册的名字
  'MyComponent',
  // 组件的实现
  {
    /* ... */
  }
)
```

If using SFCs, you will be registering the imported .vue files:

```js
import MyComponent from './App.vue'
app.component('MyComponent', MyComponent)
```

The .component() method can be chained:

```js
app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)
```

Globally registered components can be used in the template of any component within this application:

```html
<!-- 这在当前应用的任意组件中都可用 -->
<ComponentA/>
<ComponentB/>
<ComponentC/>
```

This even applies to all subcomponents, meaning all three of these components will also be available *inside each other.*

3.1.2 Local Registration

While convenient, global registration has a few drawbacks:

1. Global registration prevents build systems from removing unused components (a.k.a ”tree-shaking”). If you globally register a component but end up not using it anywhere in your app, it will still be included in the final bundle.

2. Global registration makes dependency relationships less explicit in large applications. It makes it difficult to locate a child component’s implementation from a parent component using it. This can affect long-term maintainability similar to using too many global variables.

Local registration scopes the availability of the registered components to the current component only. It makes the dependency relationship more explicit, and is more tree-shaking friendly.

如果使用单文件组件，你可以注册被导入的 .vue 文件：

```js
import MyComponent from './App.vue'
app.component('MyComponent', MyComponent)
```

.component() 方法可以被链式调用：

```js
app
  .component('ComponentA', ComponentA)
  .component('ComponentB', ComponentB)
  .component('ComponentC', ComponentC)
```

全局注册的组件可以在此应用的任意组件的模板中使用：

```html
<!-- 这在当前应用的任意组件中都可用 -->
<ComponentA/>
<ComponentB/>
<ComponentC/>
```

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

3.1.2 局部注册

全局注册虽然很方便，但有以下几个问题：

1. 全局注册，但并没有被使用的组件无法在生产打包时被自动移除 (也叫 “tree-shaking”)。如果你全局注册了一个组件，即使它并没有被实际使用，它仍然会出现在打包后的 JS 文件中。

2. 全局注册在大型项目中使项目的依赖关系变得不那么明确。在父组件中使用子组件时，不太容易定位子组件的实现。和使用过多的全局变量一样，这可能会影响应用长期的可维护性。

相比之下，局部注册的组件需要在使用它的父组件中显式导入，并且只能在该父组件中使用。它的优点是使组件之间的依赖关系更加明确，并且对 tree-shaking 更加友好。

When using SFC with `<script setup>`, imported components can be locally used without registration:

```html
<script setup>
import ComponentA from './ComponentA.vue'
</script>
<template>
  <ComponentA />
</template>
```

In non-`<script setup>`, you will need to use the `components` option:

```js
import ComponentA from './ComponentA.js'
export default {
  components: {
    ComponentA
  },
  setup() {
    // ...
  }
}
```

For each property in the `components` object, the key will be the registered name of the component, while the value will contain the implementation of the component. The above example is using the ES2015 property shorthand and is equivalent to:

```js
export default {
  components: {
    ComponentA: ComponentA
  }
  // ...
}
```

Note that **locally registered components are *not* also available in descendant components**. In this case, `ComponentA` will be made available to the current component only, not any of its child or descendant components.

在使用 `<script setup>` 的单文件组件中，导入的组件可以直接在模板中使用，无需注册：

```html
<script setup>
import ComponentA from './ComponentA.vue'
</script>
<template>
  <ComponentA />
</template>
```

如果没有使用 `<script setup>`，则需要使用 `components` 选项来显式注册：

```js
import ComponentA from './ComponentA.js'
export default {
  components: {
    ComponentA
  },
  setup() {
    // ...
  }
}
```

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

```js
export default {
  components: {
    ComponentA: ComponentA
  }
  // ...
}
```

请注意：**局部注册的组件在后代组件中并 * 不 * 可用**。在这个例子中，`ComponentA` 注册后仅在当前组件可用，而在任何的子组件或更深层的子组件中都不可用。

3.1.3 Component Name Casing

Throughout the guide, we are using PascalCase names when registering components. This is because:

1. PascalCase names are valid JavaScript identifiers. This makes it easier to import and register components in JavaScript. It also helps IDEs with auto-completion.

2. `<PascalCase />` makes it more obvious that this is a Vue component instead of a native HTML element in templates. It also differentiates Vue components from custom elements (web components).

This is the recommended style when working with SFC or string templates. However, as discussed in in-DOM Template Parsing Caveats, PascalCase tags are not usable in in-DOM templates.

Luckily, Vue supports resolving kebab-case tags to components registered using PascalCase. This means a component registered as `MyComponent` can be referenced in the template via both `<MyComponent>` and `<my-component>`. This allows us to use the same JavaScript component registration code regardless of template source.

3.2 Props

> This page assumes you've already read the Components Basics. Read that first if you are new to components.

3.2.1 Props Declaration

Vue components require explicit props declaration so that Vue knows what external props passed to the component should be treated as fallthrough attributes (which will be discussed in its dedicated section).

In SFCs using `<script setup>`, props can be declared using the `defineProps()` macro:

```html
<script setup>
const props = defineProps(['foo'])
console.log(props.foo)
```

3.1.3 组件名格式

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

1. PascalCase 是合法的 JavaScript 标识符。这使得在 JavaScript 中导入和注册组件都很容易，同时 IDE 也能提供较好的自动补全。

2. `<PascalCase />` 在模板中更明显地表明了这是一个 Vue 组件，而不是原生 HTML 元素。同时也能够将 Vue 组件和自定义元素 (web components) 区分开来。

在单文件组件和内联字符串模板中，我们都推荐这样做。但是，PascalCase 的标签名在 DOM 内模板中是不可用的，详情参见 DOM 内模板解析注意事项。

为了方便，Vue 支持将模板中使用 kebab-case 的标签解析为使用 PascalCase 注册的组件。这意味着一个以 `MyComponent` 为名注册的组件，在模板中可以通过 `<MyComponent>` 或 `<my-component>` 引用。这让我们能够使用同样的 JavaScript 组件注册代码来配合不同来源的模板。

3.2 Props

> 此章节假设你已经看过了组件基础。若你还不了解组件是什么，请先阅读该章节。

3.2.1 Props 声明

一个组件需要显式声明它所接受的 props，这样 Vue 才能知道外部传入的哪些是 props，哪些是透传 attribute (关于透传 attribute，我们会在专门的章节中讨论)。

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

```html
<script setup>
const props = defineProps(['foo'])
console.log(props.foo)
```

```
</script>
```

In non-`<script setup>` components, props are declared using the `props` option:

js

```js
export default {
  props: ['foo'],
  setup(props) {
    // setup() 接收 props 作为第一个参数
    console.log(props.foo)
  }
}
```

Notice the argument passed to `defineProps()` is the same as the value provided to the `props` options: the same props options API is shared between the two declaration styles.

In addition to declaring props using an array of strings, we can also use the object syntax:

js

```js
// 使用 <script setup>
defineProps({
  title: String,
  likes: Number
})
```

js

```js
// 非 <script setup>
export default {
  props: {
    title: String,
    likes: Number
  }
}
```

For each property in the object declaration syntax, the key is the name of the prop, while the value should be the constructor function of the expected type.

This not only documents your component, but will also warn other developers using your component in the browser console if they pass the wrong type. We will discuss more details about prop validation further down this page.

```
</script>
```

在没有使用 `<script setup>` 的组件中，prop 可以使用 `props` 选项来声明：

js

```js
export default {
  props: ['foo'],
  setup(props) {
    // setup() 接收 props 作为第一个参数
    console.log(props.foo)
  }
}
```

注意传递给 `defineProps()` 的参数和提供给 `props` 选项的值是相同的，两种声明方式背后其实使用的都是 prop 选项。

除了使用字符串数组来声明 prop 外，还可以使用对象的形式：

js

```js
// 使用 <script setup>
defineProps({
  title: String,
  likes: Number
})
```

js

```js
// 非 <script setup>
export default {
  props: {
    title: String,
    likes: Number
  }
}
```

对于以对象形式声明中的每个属性，key 是 prop 的名称，而值则是该 prop 预期类型的构造函数。比如，如果要求一个 prop 的值是 `number` 类型，则可使用 `Number` 构造函数作为其声明的值。

对象形式的 props 声明不仅可以一定程度上作为组件的文档，而且如果其他开发者在使用你的组件时传递了错误的类型，也会在浏览器控制台中抛出警告。我们将在本章节稍后进一步讨论有关 prop 校验的更多细节。

If you are using TypeScript with `<script setup>`, it's also possible to declare props using pure type annotations:

```html
<script setup lang="ts">
defineProps<{
  title?: string
  likes?: number
}>()
</script>
```

More details: Typing Component Props

3.2.2 Prop Passing Details

Prop Name Casing

We declare long prop names using camelCase because this avoids having to use quotes when using them as property keys, and allows us to reference them directly in template expressions because they are valid JavaScript identifiers:

```js
defineProps({
  greetingMessage: String
})
```

```html
<span>{{ greetingMessage }}</span>
```

Technically, you can also use camelCase when passing props to a child component (except in in-DOM templates). However, the convention is using kebab-case in all cases to align with HTML attributes:

```html
<MyComponent greeting-message="hello" />
```

We use PascalCase for component tags when possible because it improves template readability by differentiating Vue components from native elements. However, there isn't as much practical benefit in using camelCase when passing props, so we choose to follow each language's conventions.

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

```html
<script setup lang="ts">
defineProps<{
  title?: string
  likes?: number
}>()
</script>
```

更多关于基于类型的声明的细节请参考组件 props 类型标注。

3.2.2 传递 prop 的细节

Prop 名字格式

如果一个 prop 的名字很长，应使用 camelCase 形式，因为它们是合法的 JavaScript 标识符，可以直接在模板的表达式中使用，也可以避免在作为属性 key 名时必须加上引号。

```js
defineProps({
  greetingMessage: String
})
```

```html
<span>{{ greetingMessage }}</span>
```

虽然理论上你也可以在向子组件传递 props 时使用 camelCase 形式 (使用 DOM 内模板时例外)，但实际上为了和 HTML attribute 对齐，我们通常会将其写为 kebab-case 形式：

```html
<MyComponent greeting-message="hello" />
```

对于组件名我们推荐使用 PascalCase，因为这提高了模板的可读性，能帮助我们区分 Vue 组件和原生 HTML 元素。然而对于传递 props 来说，使用 camelCase 并没有太多优势，因此我们推荐更贴近 HTML 的书写风格。

Static vs. Dynamic Props

So far, you've seen props passed as static values, like in:

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

You've also seen props assigned dynamically with `v-bind` or its `:` shortcut, such as in:

```html
<!-- 根据一个变量的值动态传入 -->
<BlogPost :title="post.title" />
<!-- 根据一个更复杂表达式的值动态传入 -->
<BlogPost :title="post.title + ' by ' + post.author.name" />
```

Passing Different Value Types

In the two examples above, we happen to pass string values, but *any* type of value can be passed to a prop.

静态 vs. 动态 Prop

至此，你已经见过了很多像这样的静态值形式的 props：

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

相应地，还有使用 `v-bind` 或缩写 `:` 来进行动态绑定的 props：

```html
<!-- 根据一个变量的值动态传入 -->
<BlogPost :title="post.title" />
<!-- 根据一个更复杂表达式的值动态传入 -->
<BlogPost :title="post.title + ' by ' + post.author.name" />
```

传递不同的值类型

在上述的两个例子中，我们只传入了字符串值，但实际上**任何**类型的值都可以作为 props 的值被传递。

Number

```html
<!-- 虽然 `42` 是个常量，我们还是需要使用 v-bind -->
<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->
<BlogPost :likes="42" />
<!-- 根据一个变量的值动态传入 -->
<BlogPost :likes="post.likes" />
```

Boolean

```html
<!-- 仅写上 prop 但不传值，会隐式转换为 `true` -->
<BlogPost is-published />
<!-- 虽然 `false` 是静态的值，我们还是需要使用 v-bind -->
<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->
<BlogPost :is-published="false" />
<!-- 根据一个变量的值动态传入 -->
<BlogPost :is-published="post.isPublished" />
```

Array

```html
<!-- 虽然这个数组是个常量，我们还是需要使用 v-bind -->
<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->
<BlogPost :comment-ids="[234, 266, 273]" />
<!-- 根据一个变量的值动态传入 -->
<BlogPost :comment-ids="post.commentIds" />
```

Object

```html
<!-- 虽然这个对象字面量是个常量，我们还是需要使用 v-bind -->
<!-- 因为这是一个 JavaScript 表达式而不是一个字符串 -->
<BlogPost
    :author="{
    name: 'Veronica',
    company: 'Veridian Dynamics'
    }"
    />
<!-- 根据一个变量的值动态传入 -->
<BlogPost :author="post.author" />
```

Binding Multiple Properties Using an Object

If you want to pass all the properties of an object as props, you can use `v-bind` without an argument (`v-bind` instead of `:prop-name`). For example, given a `post` object:

```js
const post = {
  id: 1,
  title: 'My Journey with Vue'
}
```

The following template:

```html
<BlogPost v-bind="post" />
```

Will be equivalent to:

使用一个对象绑定多个 prop

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

```js
const post = {
  id: 1,
  title: 'My Journey with Vue'
}
```

以及下面的模板：

```html
<BlogPost v-bind="post" />
```

而这实际上等价于：

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

3.2.3 One-Way Data Flow

All props form a **one-way-down binding** between the child property and the parent one: when the parent property updates, it will flow down to the child, but not the other way around. This prevents child components from accidentally mutating the parent's state, which can make your app's data flow harder to understand.

In addition, every time the parent component is updated, all props in the child component will be refreshed with the latest value. This means you should **not** attempt to mutate a prop inside a child component. If you do, Vue will warn you in the console:

```js
const props = defineProps(['foo'])
//  警告！prop 是只读的！
props.foo = 'bar'
```

There are usually two cases where it's tempting to mutate a prop:

1. **The prop is used to pass in an initial value; the child component wants to use it as a local data property afterwards.** In this case, it's best to define a local data property that uses the prop as its initial value:

   ```js
   const props = defineProps(['initialCounter'])
   // 计数器只是将 props.initialCounter 作为初始值
   // 像下面这样做就使 prop 和后续更新无关了
   const counter = ref(props.initialCounter)
   ```

2. **The prop is passed in as a raw value that needs to be transformed.** In this case, it's best to define a computed property using the prop's value:

   ```js
   const props = defineProps(['size'])
   // 该 prop 变更时计算属性也会自动更新
   const normalizedSize = computed(() => props.size.trim().toLowerCase())
   ```

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

3.2.3 单向数据流

所有的 props 都遵循着**单向绑定**原则，props 因父组件的更新而变化，自然地将新的状态向下流往子组件，而不会逆向传递。这避免了子组件意外修改父组件的状态的情况，不然应用的数据流将很容易变得混乱而难以理解。

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

```js
const props = defineProps(['foo'])
//  警告！prop 是只读的！
props.foo = 'bar'
```

导致你想要更改一个 prop 的需求通常来源于以下两种场景：

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

   ```js
   const props = defineProps(['initialCounter'])
   // 计数器只是将 props.initialCounter 作为初始值
   // 像下面这样做就使 prop 和后续更新无关了
   const counter = ref(props.initialCounter)
   ```

2. **需要对传入的 prop 值做进一步的转换**。在这种情况中，最好是基于该 prop 值定义一个计算属性：

   ```js
   const props = defineProps(['size'])
   // 该 prop 变更时计算属性也会自动更新
   const normalizedSize = computed(() => props.size.trim().toLowerCase())
   ```

Mutating Object / Array Props

When objects and arrays are passed as props, while the child component cannot mutate the prop binding, it **will** be able to mutate the object or array's nested properties. This is because in JavaScript objects and arrays are passed by reference, and it is unreasonably expensive for Vue to prevent such mutations.

The main drawback of such mutations is that it allows the child component to affect parent state in a way that isn't obvious to the parent component, potentially making it more difficult to reason about the data flow in the future. As a best practice, you should avoid such mutations unless the parent and child are tightly coupled by design. In most cases, the child should emit an event to let the parent perform the mutation.

3.2.4 Prop Validation

Components can specify requirements for their props, such as the types you've already seen. If a requirement is not met, Vue will warn you in the browser's JavaScript console. This is especially useful when developing a component that is intended to be used by others.

To specify prop validations, you can provide an object with validation requirements to the `defineProps()` macro, instead of an array of strings. For example:

更改对象 / 数组类型的 props

当对象或数组作为 props 被传入时，虽然子组件无法更改 props 绑定，但仍然**可以**更改对象或数组内部的值。这是因为 JavaScript 的对象和数组是按引用传递，而对 Vue 来说，禁止这样的改动，虽然可能生效，但有很大的性能损耗，比较得不偿失。

这种更改的主要缺陷是它允许了子组件以某种不明显的方式影响父组件的状态，可能会使数据流在将来变得更难以理解。在最佳实践中，你应该尽可能避免这样的更改，除非父子组件在设计上本来就需要紧密耦合。在大多数场景下，子组件应该抛出一个事件来通知父组件做出改变。

3.2.4 Prop 校验

Vue 组件可以更细致地声明对传入的 props 的校验要求。比如我们上面已经看到过的类型声明，如果传入的值不满足类型要求，Vue 会在浏览器控制台中抛出警告来提醒使用者。这在开发给其他开发者使用的组件时非常有用。

要声明对 props 的校验，你可以向 `defineProps()` 宏提供一个带有 props 校验选项的对象，例如：

```js
defineProps({
    // 基础类型检查
    // （给出 `null` 和 `undefined` 值则会跳过任何类型检查）
    propA: Number,
    // 多种可能的类型
    propB: [String, Number],
    // 必传，且为 String 类型
    propC: {
        type: String,
        required: true
    },
    // Number 类型的默认值
    propD: {
        type: Number,
        default: 100
```

```
  },
  // 对象类型的默认值
  propE: {
    type: Object,
    // 对象或数组的默认值
    // 必须从一个工厂函数返回。
    // 该函数接收组件所接收到的原始 prop 作为参数。
    default(rawProps) {
    return { message: 'hello' }
    }
  },
  // 自定义类型校验函数
  propF: {
    validator(value) {
    // The value must match one of these strings
    return ['success', 'warning', 'danger'].includes(value)
    }
  },
  // 函数类型的默认值
  propG: {
    type: Function,
    // 不像对象或数组的默认，这不是一个
    // 工厂函数。这会是一个用来作为默认值的函数
    default() {
    return 'Default function'
    }
  }
  })
```

TIP

Code inside the `defineProps()` argument **cannot access other variables declared in `<script setup>`**, because the entire expression is moved to an outer function scope when compiled.

Additional details:

TIP

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

一些补充细节：

- All props are optional by default, unless `required: true` is specified.
- An absent optional prop other than `Boolean` will have `undefined` value.
- The `Boolean` absent props will be cast to `false`. You can change this by setting a `default` for it — i.e.: `default: undefined` to behave as a non-Boolean prop.
- If a `default` value is specified, it will be used if the resolved prop value is `undefined` - this includes both when the prop is absent, or an explicit `undefined` value is passed.

When prop validation fails, Vue will produce a console warning (if using the development build).

If using Type-based props declarations , Vue will try its best to compile the type annotations into equivalent runtime prop declarations. For example, `defineProps<{ msg: string }>` will be compiled into `{ msg: { type: String, required: true }}`.

Runtime Type Checks

The `type` can be one of the following native constructors:

- `String`
- `Number`
- `Boolean`
- `Array`
- `Object`
- `Date`
- `Function`
- `Symbol`

In addition, `type` can also be a custom class or constructor function and the assertion will be made with an `instanceof` check. For example, given the following class:

js

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName
```

- 所有 prop 默认都是可选的，除非声明了 `required: true`。
- 除 `Boolean` 外的未传递的可选 prop 将会有一个默认值 `undefined`。
- `Boolean` 类型的未传递 prop 将被转换为 `false`。这可以通过为它设置 `default` 来更改——例如：设置为 `default: undefined` 将与非布尔类型的 prop 的行为保持一致。
- 如果声明了 `default` 值，那么在 prop 的值被解析为 `undefined` 时，无论 prop 是未被传递还是显式指明的 `undefined`，都会改为 `default` 值。

当 prop 的校验失败后，Vue 会抛出一个控制台警告 (在开发模式下)。

如果使用了基于类型的 prop 声明 ，Vue 会尽最大努力在运行时按照 prop 的类型标注进行编译。举例来说，`defineProps<{ msg: string }>` 会被编译为 `{ msg: { type: String, required: true }}`。

运行时类型检查

校验选项中的 `type` 可以是下列这些原生构造函数：

- `String`
- `Number`
- `Boolean`
- `Array`
- `Object`
- `Date`
- `Function`
- `Symbol`

另外，`type` 也可以是自定义的类或构造函数，Vue 将会通过 `instanceof` 来检查类型是否匹配。例如下面这个类：

js

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
}
```

You could use it as a prop's type:

```js
defineProps({
  author: Person
})
```

Vue will use `instanceof Person` to validate whether the value of the `author` prop is indeed an instance of the `Person` class.

3.2.5 Boolean Casting

Props with `Boolean` type have special casting rules to mimic the behavior of native boolean attributes. Given a `<MyComponent>` with the following declaration:

```js
defineProps({
  disabled: Boolean
})
```

The component can be used like this:

```html
<!-- 等同于传入 :disabled="true" -->
<MyComponent disabled />
<!-- 等同于传入 :disabled="false" -->
<MyComponent />
```

When a prop is declared to allow multiple types, the casting rules for `Boolean` will also be applied. However, there is an edge when both `String` and `Boolean` are allowed - the Boolean casting rule only applies if Boolean appears before String:

```js
// disabled 将被转换为 true
defineProps({
  disabled: [Boolean, Number]
})
```

```
    this.lastName = lastName
  }
}
```

你可以将其作为一个 prop 的类型：

```js
defineProps({
  author: Person
})
```

Vue 会通过 `instanceof Person` 来校验 `author` prop 的值是否是 `Person` 类的一个实例。

3.2.5 Boolean 类型转换

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

```js
defineProps({
  disabled: Boolean
})
```

该组件可以被这样使用：

```html
<!-- 等同于传入 :disabled="true" -->
<MyComponent disabled />
<!-- 等同于传入 :disabled="false" -->
<MyComponent />
```

当一个 prop 被声明为允许多种类型时，`Boolean` 的转换规则也将被应用。然而，当同时允许 `String` 和 `Boolean` 时，有一种边缘情况——只有当 `Boolean` 出现在 `String` 之前时，`Boolean` 转换规则才适用：

```js
// disabled 将被转换为 true
defineProps({
  disabled: [Boolean, Number]
})
```

```
// disabled 将被转换为 true
defineProps({
  disabled: [Boolean, String]
})
// disabled 将被转换为 true
defineProps({
  disabled: [Number, Boolean]
})
// disabled 将被解析为空字符串 (disabled="")
defineProps({
  disabled: [String, Boolean]
})
```

3.3 Component Events

> This page assumes you've already read the Components Basics. Read that first if you are new to components.

3.3.1 Emitting and Listening to Events

A component can emit custom events directly in template expressions (e.g. in a v-on handler) using the built-in $emit method:

```html
<!-- MyComponent -->
<button @click="$emit('someEvent')">click me</button>
```

The parent can then listen to it using v-on:

```html
<MyComponent @some-event="callback" />
```

The .once modifier is also supported on component event listeners:

```html
<MyComponent @some-event.once="callback" />
```

Like components and props, event names provide an automatic case transformation. Notice we

```
// disabled 将被转换为 true
defineProps({
  disabled: [Boolean, String]
})
// disabled 将被转换为 true
defineProps({
  disabled: [Number, Boolean]
})
// disabled 将被解析为空字符串 (disabled="")
defineProps({
  disabled: [String, Boolean]
})
```

3.3 组件事件

> 此章节假设你已经看过了组件基础。若你还不了解组件是什么，请先阅读该章节。

3.3.1 触发与监听事件

在组件的模板表达式中，可以直接使用 $emit 方法触发自定义事件 (例如：在 v-on 的处理函数中)：

```html
<!-- MyComponent -->
<button @click="$emit('someEvent')">click me</button>
```

父组件可以通过 v-on (缩写为 @) 来监听事件：

```html
<MyComponent @some-event="callback" />
```

同样，组件的事件监听器也支持 .once 修饰符：

```html
<MyComponent @some-event.once="callback" />
```

像组件与 prop 一样，事件的名字也提供了自动的格式转换。注意这里我们触发了

emitted a camelCase event, but can listen for it using a kebab-cased listener in the parent. As with props casing, we recommend using kebab-cased event listeners in templates.

> **TIP**
> Unlike native DOM events, component emitted events do **not** bubble. You can only listen to the events emitted by a direct child component. If there is a need to communicate between sibling or deeply nested components, use an external event bus or a global state management solution.

3.3.2 Event Arguments

It's sometimes useful to emit a specific value with an event. For example, we may want the `<BlogPost>` component to be in charge of how much to enlarge the text by. In those cases, we can pass extra arguments to `$emit` to provide this value:

```html
<button @click="$emit('increaseBy', 1)">
    Increase by 1
</button>
```

Then, when we listen to the event in the parent, we can use an inline arrow function as the listener, which allows us to access the event argument:

```html
<MyButton @increase-by="(n) => count += n" />
```

Or, if the event handler is a method:

```html
<MyButton @increase-by="increaseCount" />
```

Then the value will be passed as the first parameter of that method:

```js
function increaseCount(n) {
  count.value += n
}
```

一个以 camelCase 形式命名的事件，但在父组件中可以使用 kebab-case 形式来监听。与 prop 大小写格式一样，在模板中我们也推荐使用 kebab-case 形式来编写监听器。

> **TIP**
> 和原生 DOM 事件不一样，组件触发的事件**没有冒泡机制**。你只能监听直接子组件触发的事件。平级组件或是跨越多层嵌套的组件间通信，应使用一个外部的事件总线，或是使用一个全局状态管理方案。

3.3.2 事件参数

有时候我们会需要在触发事件时附带一个特定的值。举例来说，我们想要 `<BlogPost>` 组件来管理文本会缩放得多大。在这个场景下，我们可以给 `$emit` 提供一个额外的参数：

```html
<button @click="$emit('increaseBy', 1)">
    Increase by 1
</button>
```

然后我们在父组件中监听事件，我们可以先简单写一个内联的箭头函数作为监听器，此函数会接收到事件附带的参数：

```html
<MyButton @increase-by="(n) => count += n" />
```

或者，也可以用一个组件方法来作为事件处理函数：

```html
<MyButton @increase-by="increaseCount" />
```

该方法也会接收到事件所传递的参数：

```js
function increaseCount(n) {
  count.value += n
}
```

TIP

All extra arguments passed to `$emit()` after the event name will be forwarded to the listener. For example, with `$emit('foo', 1, 2, 3)` the listener function will receive three arguments.

3.3.3 Declaring Emitted Events

A component can explicitly declare the events it will emit using the `defineEmits()` macro:

```html
<script setup>
defineEmits(['inFocus', 'submit'])
</script>
```

The `$emit` method that we used in the `<template>` isn't accessible within the `<script setup>` section of a component, but `defineEmits()` returns an equivalent function that we can use instead:

```html
<script setup>
const emit = defineEmits(['inFocus', 'submit'])
function buttonClick() {
  emit('submit')
}
</script>
```

The `defineEmits()` macro **cannot** be used inside a function, it must be placed directly within `<script setup>`, as in the example above.

If you're using an explicit `setup` function instead of `<script setup>`, events should be declared using the `emits` option, and the `emit` function is exposed on the `setup()` context:

```js
export default {
  emits: ['inFocus', 'submit'],
  setup(props, ctx) {
    ctx.emit('submit')
  }
}
```

TIP

所有传入 `$emit()` 的额外参数都会被直接传向监听器。举例来说，`$emit('foo', 1, 2, 3)` 触发后，监听器函数将会收到这三个参数值。

3.3.3 声明触发的事件

组件可以显式地通过 `defineEmits()` 宏来声明它要触发的事件：

```html
<script setup>
defineEmits(['inFocus', 'submit'])
</script>
```

我们在 `<template>` 中使用的 `$emit` 方法不能在组件的 `<script setup>` 部分中使用，但 `defineEmits()` 会返回一个相同作用的函数供我们使用：

```html
<script setup>
const emit = defineEmits(['inFocus', 'submit'])
function buttonClick() {
  emit('submit')
}
</script>
```

`defineEmits()` 宏**不能**在子函数中使用。如上所示，它必须直接放置在 `<script setup>` 的顶级作用域下。

如果你显式地使用了 `setup` 函数而不是 `<script setup>`，则事件需要通过 `emits` 选项来定义，`emit` 函数也被暴露在 `setup()` 的上下文对象上：

```js
export default {
  emits: ['inFocus', 'submit'],
  setup(props, ctx) {
    ctx.emit('submit')
  }
}
```

As with other properties of the `setup()` context, `emit` can safely be destructured:

```js
export default {
  emits: ['inFocus', 'submit'],
  setup(props, { emit }) {
    emit('submit')
  }
}
```

The `emits` option and `defineEmits()` macro also support an object syntax, which allows us to perform runtime validation of the payload of the emitted events:

```html
<script setup>
const emit = defineEmits({
  submit(payload) {
    // 通过返回值为 `true` 还是为 `false` 来判断
    // 验证是否通过
  }
})
</script>
```

If you are using TypeScript with `<script setup>`, it's also possible to declare emitted events using pure type annotations:

```html
<script setup lang="ts">
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()
</script>
```

More details: Typing Component Emits

Although optional, it is recommended to define all emitted events in order to better document how a component should work. It also allows Vue to exclude known listeners from fallthrough attributes, avoiding edge cases caused by DOM events manually dispatched by 3rd party code.

与 `setup()` 上下文对象中的其他属性一样，`emit` 可以安全地被解构：

```js
export default {
  emits: ['inFocus', 'submit'],
  setup(props, { emit }) {
    emit('submit')
  }
}
```

这个 `emits` 选项和 `defineEmits()` 宏还支持对象语法，它允许我们对触发事件的参数进行验证：

```html
<script setup>
const emit = defineEmits({
  submit(payload) {
    // 通过返回值为 `true` 还是为 `false` 来判断
    // 验证是否通过
  }
})
</script>
```

如果你正在搭配 TypeScript 使用 `<script setup>`，也可以使用纯类型标注来声明触发的事件：

```html
<script setup lang="ts">
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()
</script>
```

TypeScript 用户请参考：如何为组件所抛出事件标注类型

尽管事件声明是可选的，我们还是推荐你完整地声明所有要触发的事件，以此在代码中作为文档记录组件的用法。同时，事件声明能让 Vue 更好地将事件和透传 attribute 作出区分，从而避免一些由第三方代码触发的自定义 DOM 事件所导致的边界情况。

TIP

If a native event (e.g., `click`) is defined in the `emits` option, the listener will now only listen to component-emitted `click` events and no longer respond to native `click` events.

3.3.4 Events Validation

Similar to prop type validation, an emitted event can be validated if it is defined with the object syntax instead of the array syntax.

To add validation, the event is assigned a function that receives the arguments passed to the `emit` call and returns a boolean to indicate whether the event is valid or not.

```html
<script setup>
const emit = defineEmits({
    // 没有校验
    click: null,
    // 校验 submit 事件
    submit: ({ email, password }) => {
    if (email && password) {
        return true
    } else {
        console.warn('Invalid submit event payload!')
        return false
    }
    }
})
function submitForm(email, password) {
    emit('submit', { email, password })
}
</script>
```

3.4 Component v-model

`v-model` can be used on a component to implement a two-way binding.

TIP

如果一个原生事件的名字 (例如 `click`) 被定义在 `emits` 选项中，则监听器只会监听组件触发的 `click` 事件而不会再响应原生的 `click` 事件。

3.3.4 事件校验

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

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

```html
<script setup>
const emit = defineEmits({
    // 没有校验
    click: null,
    // 校验 submit 事件
    submit: ({ email, password }) => {
    if (email && password) {
        return true
    } else {
        console.warn('Invalid submit event payload!')
        return false
    }
    }
})
function submitForm(email, password) {
    emit('submit', { email, password })
}
</script>
```

3.4 组件 v-model

`v-model` 可以在组件上使用以实现双向绑定。

First let's revisit how `v-model` is used on a native element:

```html
<input v-model="searchText" />
```

Under the hood, the template compiler expands `v-model` to the more verbose equivalent for us. So the above code does the same as the following:

```html
<input
  :value="searchText"
  @input="searchText = $event.target.value"
/>
```

When used on a component, `v-model` instead expands to this:

```html
<CustomInput
  :model-value="searchText"
  @update:model-value="newValue => searchText = newValue"
/>
```

For this to actually work though, the `<CustomInput>` component must do two things:

1. Bind the `value` attribute of a native `<input>` element to the `modelValue` prop
2. When a native `input` event is triggered, emit an `update:modelValue` custom event with the new value

Here's that in action:

```html
<!-- CustomInput.vue -->
<script setup>
defineProps(['modelValue'])
defineEmits(['update:modelValue'])
</script>
<template>
  <input
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
  />
</template>
```

首先让我们回忆一下 `v-model` 在原生元素上的用法：

```html
<input v-model="searchText" />
```

在代码背后，模板编译器会对 `v-model` 进行更冗长的等价展开。因此上面的代码其实等价于下面这段：

```html
<input
  :value="searchText"
  @input="searchText = $event.target.value"
/>
```

而当使用在一个组件上时，`v-model` 会被展开为如下的形式：

```html
<CustomInput
  :model-value="searchText"
  @update:model-value="newValue => searchText = newValue"
/>
```

要让这个例子实际工作起来，`<CustomInput>` 组件内部需要做两件事：

1. 将内部原生 `<input>` 元素的 `value` attribute 绑定到 `modelValue` prop
2. 当原生的 `input` 事件触发时，触发一个携带了新值的 `update:modelValue` 自定义事件

这里是相应的代码：

```html
<!-- CustomInput.vue -->
<script setup>
defineProps(['modelValue'])
defineEmits(['update:modelValue'])
</script>
<template>
  <input
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
  />
</template>
```

Now `v-model` should work perfectly with this component:

html

```
<CustomInput v-model="searchText" />
```

Try it in the Playground

Another way of implementing `v-model` within this component is to use a writable `computed` property with both a getter and a setter. The `get` method should return the `modelValue` property and the `set` method should emit the corresponding event:

html

```
<!-- CustomInput.vue -->
<script setup>
import { computed } from 'vue'
const props = defineProps(['modelValue'])
const emit = defineEmits(['update:modelValue'])
const value = computed({
  get() {
    return props.modelValue
  },
  set(value) {
    emit('update:modelValue', value)
  }
})
</script>
<template>
  <input v-model="value" />
</template>
```

3.4.1 v-model arguments

By default, `v-model` on a component uses `modelValue` as the prop and `update:modelValue` as the event. We can modify these names passing an argument to `v-model`:

html

```
<MyComponent v-model:title="bookTitle" />
```

In this case, the child component should expect a `title` prop and emit an `update:title` event to update the parent value:

现在 `v-model` 可以在这个组件上正常工作了：

html

```
<CustomInput v-model="searchText" />
```

在演练场中尝试一下

另一种在组件内实现 `v-model` 的方式是使用一个可写的，同时具有 getter 和 setter 的 `computed` 属性。`get` 方法需返回 `modelValue` prop，而 `set` 方法需触发相应的事件：

html

```
<!-- CustomInput.vue -->
<script setup>
import { computed } from 'vue'
const props = defineProps(['modelValue'])
const emit = defineEmits(['update:modelValue'])
const value = computed({
  get() {
    return props.modelValue
  },
  set(value) {
    emit('update:modelValue', value)
  }
})
</script>
<template>
  <input v-model="value" />
</template>
```

3.4.1 v-model 的参数

默认情况下，`v-model` 在组件上都是使用 `modelValue` 作为 prop，并以 `update:modelValue` 作为对应的事件。我们可以通过给 `v-model` 指定一个参数来更改这些名字：

html

```
<MyComponent v-model:title="bookTitle" />
```

在这个例子中，子组件应声明一个 `title` prop，并通过触发 `update:title` 事件更新父组件值：

html

```html
<!-- MyComponent.vue -->
<script setup>
defineProps(['title'])
defineEmits(['update:title'])
</script>
<template>
  <input
    type="text"
    :value="title"
    @input="$emit('update:title', $event.target.value)"
  />
</template>
```

Try it in the Playground

3.4.2 Multiple v-model bindings

By leveraging the ability to target a particular prop and event as we learned before with `v-model` arguments, we can now create multiple `v-model` bindings on a single component instance.

Each `v-model` will sync to a different prop, without the need for extra options in the component:

html

```html
<UserName
  v-model:first-name="first"
  v-model:last-name="last"
/>
```

html

```html
<script setup>
defineProps({
  firstName: String,
  lastName: String
})
defineEmits(['update:firstName', 'update:lastName'])
</script>
<template>
  <input
```

html

```html
<!-- MyComponent.vue -->
<script setup>
defineProps(['title'])
defineEmits(['update:title'])
</script>
<template>
  <input
    type="text"
    :value="title"
    @input="$emit('update:title', $event.target.value)"
  />
</template>
```

在演练场中尝试一下

3.4.2 多个 v-model 绑定

利用刚才在 `v-model` 参数小节中学到的指定参数与事件名的技巧，我们可以在单个组件实例上创建多个 `v-model` 双向绑定。

组件上的每一个 `v-model` 都会同步不同的 prop，而无需额外的选项：

html

```html
<UserName
  v-model:first-name="first"
  v-model:last-name="last"
/>
```

html

```html
<script setup>
defineProps({
  firstName: String,
  lastName: String
})
defineEmits(['update:firstName', 'update:lastName'])
</script>
<template>
  <input
```

```
    type="text"
    :value="firstName"
    @input="$emit('update:firstName', $event.target.value)"
  />
  <input
    type="text"
    :value="lastName"
    @input="$emit('update:lastName', $event.target.value)"
  />
</template>
```

Try it in the Playground

3.4.3 Handling v-model modifiers

When we were learning about form input bindings, we saw that `v-model` has built-in modifiers - `.trim`, `.number` and `.lazy`. In some cases, you might also want the `v-model` on your custom input component to support custom modifiers.

Let's create an example custom modifier, `capitalize`, that capitalizes the first letter of the string provided by the `v-model` binding:

```html
<MyComponent v-model.capitalize="myText" />
```

Modifiers added to a component `v-model` will be provided to the component via the `modelModifiers` prop. In the below example, we have created a component that contains a `modelModifiers` prop that defaults to an empty object:

```html
<script setup>
const props = defineProps({
  modelValue: String,
  modelModifiers: { default: () => ({}) }
})
defineEmits(['update:modelValue'])
console.log(props.modelModifiers) // { capitalize: true }
</script>
<template>
```

```
    type="text"
    :value="firstName"
    @input="$emit('update:firstName', $event.target.value)"
  />
  <input
    type="text"
    :value="lastName"
    @input="$emit('update:lastName', $event.target.value)"
  />
</template>
```

在演练场中尝试一下

3.4.3 处理 v-model 修饰符

在学习输入绑定时，我们知道了 `v-model` 有一些内置的修饰符，例如 `.trim`，`.number` 和 `.lazy`。在某些场景下，你可能想要一个自定义组件的 `v-model` 支持自定义的修饰符。

我们来创建一个自定义的修饰符 `capitalize`，它会自动将 `v-model` 绑定输入的字符串值第一个字母转为大写：

```html
<MyComponent v-model.capitalize="myText" />
```

组件的 `v-model` 上所添加的修饰符，可以通过 `modelModifiers` prop 在组件内访问到。在下面的组件中，我们声明了 `modelModifiers` 这个 prop，它的默认值是一个空对象：

```html
<script setup>
const props = defineProps({
  modelValue: String,
  modelModifiers: { default: () => ({}) }
})
defineEmits(['update:modelValue'])
console.log(props.modelModifiers) // { capitalize: true }
</script>
<template>
```

```
  <input
    type="text"
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
  />
</template>
```

Notice the component's `modelModifiers` prop contains `capitalize` and its value is `true` - due to it being set on the `v-model` binding `v-model.capitalize="myText"`.

Now that we have our prop set up, we can check the `modelModifiers` object keys and write a handler to change the emitted value. In the code below we will capitalize the string whenever the `<input />` element fires an `input` event.

html

```
<script setup>
const props = defineProps({
  modelValue: String,
  modelModifiers: { default: () => ({}) }
})
const emit = defineEmits(['update:modelValue'])
function emitValue(e) {
  let value = e.target.value
  if (props.modelModifiers.capitalize) {
    value = value.charAt(0).toUpperCase() + value.slice(1)
  }
  emit('update:modelValue', value)
}
</script>
<template>
  <input type="text" :value="modelValue" @input="emitValue" />
</template>
```

Try it in the Playground

Modifiers for v-model with arguments

For `v-model` bindings with both argument and modifiers, the generated prop name will be `arg + "Modifiers"`. For example:

```
  <input
    type="text"
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
  />
</template>
```

注意这里组件的 `modelModifiers` prop 包含了 `capitalize` 且其值为 `true`，因为它在模板中的 `v-model` 绑定 `v-model.capitalize="myText"` 上被使用了。

有了这个 prop，我们就可以检查 `modelModifiers` 对象的键，并编写一个处理函数来改变抛出的值。在下面的代码里，我们就是在每次 `<input />` 元素触发 `input` 事件时将值的首字母大写：

html

```
<script setup>
const props = defineProps({
  modelValue: String,
  modelModifiers: { default: () => ({}) }
})
const emit = defineEmits(['update:modelValue'])
function emitValue(e) {
  let value = e.target.value
  if (props.modelModifiers.capitalize) {
    value = value.charAt(0).toUpperCase() + value.slice(1)
  }
  emit('update:modelValue', value)
}
</script>
<template>
  <input type="text" :value="modelValue" @input="emitValue" />
</template>
```

在演练场中尝试一下

带参数的 v-model 修饰符

对于又有参数又有修饰符的 `v-model` 绑定，生成的 prop 名将是 `arg + "Modifiers"`。举例来说：

```html
<MyComponent v-model:title.capitalize="myText">
```

The corresponding declarations should be:

```js
const props = defineProps(['title', 'titleModifiers'])
defineEmits(['update:title'])
console.log(props.titleModifiers) // { capitalize: true }
```

Here's another example of using modifiers with multiple `v-model` with different arguments:

```html
<UserName
  v-model:first-name.capitalize="first"
  v-model:last-name.uppercase="last"
/>
```

```html
<script setup>
const props = defineProps({
  firstName: String,
  lastName: String,
  firstNameModifiers: { default: () => ({}) },
  lastNameModifiers: { default: () => ({}) }
})
defineEmits(['update:firstName', 'update:lastName'])
console.log(props.firstNameModifiers) // { capitalize: true }
console.log(props.lastNameModifiers) // { uppercase: true}
</script>
```

3.5 Fallthrough Attributes

> This page assumes you've already read the Components Basics. Read that first if you are new to components.

3.5.1 Attribute Inheritance

A "fallthrough attribute" is an attribute or `v-on` event listener that is passed to a component, but is not explicitly declared in the receiving component's props or emits. Common examples of this

```html
<MyComponent v-model:title.capitalize="myText">
```

相应的声明应该是：

```js
const props = defineProps(['title', 'titleModifiers'])
defineEmits(['update:title'])
console.log(props.titleModifiers) // { capitalize: true }
```

这里是另一个例子，展示了如何在使用多个不同参数的 `v-model` 时使用修饰符：

```html
<UserName
  v-model:first-name.capitalize="first"
  v-model:last-name.uppercase="last"
/>
```

```html
<script setup>
const props = defineProps({
  firstName: String,
  lastName: String,
  firstNameModifiers: { default: () => ({}) },
  lastNameModifiers: { default: () => ({}) }
})
defineEmits(['update:firstName', 'update:lastName'])
console.log(props.firstNameModifiers) // { capitalize: true }
console.log(props.lastNameModifiers) // { uppercase: true}
</script>
```

3.5 透传 Attributes

> 此章节假设你已经看过了组件基础。若你还不了解组件是什么，请先阅读该章节。

3.5.1 Attributes 继承

“透传 attribute” 指的是传递给一个组件，却没有被该组件声明为 props 或 emits 的 attribute 或者 `v-on` 事件监听器。最常见的例子就是 `class`、`style` 和 `id`。

include `class`, `style`, and `id` attributes.

When a component renders a single root element, fallthrough attributes will be automatically added to the root element's attributes. For example, given a `<MyButton>` component with the following template:

```html
<!-- <MyButton> 的模板 -->
<button>click me</button>
```

And a parent using this component with:

```html
<MyButton class="large" />
```

The final rendered DOM would be:

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

Here, `<MyButton>` did not declare `class` as an accepted prop. Therefore, `class` is treated as a fallthrough attribute and automatically added to `<MyButton>`'s root element.

class and style Merging

If the child component's root element already has existing `class` or `style` attributes, it will be merged with the `class` and `style` values that are inherited from the parent. Suppose we change the template of `<MyButton>` in the previous example to:

```html
<!-- <MyButton> 的模板 -->
<button class="btn">click me</button>
```

Then the final rendered DOM would now become:

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

v-on Listener Inheritance

The same rule applies to `v-on` event listeners:

```html
<MyButton @click="onClick" />
```

当一个组件以单个元素为根作渲染时，透传的 attribute 会自动被添加到根元素上。举例来说，假如我们有一个 `<MyButton>` 组件，它的模板长这样：

```html
<!-- <MyButton> 的模板 -->
<button>click me</button>
```

一个父组件使用了这个组件，并且传入了 `class`：

```html
<MyButton class="large" />
```

最后渲染出的 DOM 结果是：

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

这里，`<MyButton>` 并没有将 `class` 声明为一个它所接受的 prop，所以 `class` 被视作透传 attribute，自动透传到了 `<MyButton>` 的根元素上。

对 class 和 style 的合并

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

```html
<!-- <MyButton> 的模板 -->
<button class="btn">click me</button>
```

则最后渲染出的 DOM 结果会变成：

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

v-on 监听器继承

同样的规则也适用于 `v-on` 事件监听器：

```html
<MyButton @click="onClick" />
```

The `click` listener will be added to the root element of `<MyButton>`, i.e. the native `<button>` element. When the native `<button>` is clicked, it will trigger the `onClick` method of the parent component. If the native `<button>` already has a `click` listener bound with `v-on`, then both listeners will trigger.

Nested Component Inheritance

If a component renders another component as its root node, for example, we refactored `<MyButton>` to render a `<BaseButton>` as its root:

html
```html
<!-- <MyButton/> 的模板，只是渲染另一个组件 -->
<BaseButton />
```

Then the fallthrough attributes received by `<MyButton>` will be automatically forwarded to `<BaseButton>`.

Note that:

1. Forwarded attributes do not include any attributes that are declared as props, or `v-on` listeners of declared events by `<MyButton>` - in other words, the declared props and listeners have been ”consumed” by `<MyButton>`.

2. Forwarded attributes may be accepted as props by `<BaseButton>`, if declared by it.

3.5.2 Disabling Attribute Inheritance

If you do **not** want a component to automatically inherit attributes, you can set `inheritAttrs: false` in the component’s options.

Since 3.3 you can also use `defineOptions` directly in `<script setup>`:

html
```html
<script setup>
defineOptions({
    inheritAttrs: false
})
// ...setup 逻辑
</script>
```

`click` 监听器会被添加到 `<MyButton>` 的根元素，即那个原生的 `<button>` 元素之上。当原生的 `<button>` 被点击，会触发父组件的 `onClick` 方法。同样的，如果原生 `button` 元素自身也通过 `v-on` 绑定了一个事件监听器，则这个监听器和从父组件继承的监听器都会被触发。

深层组件继承

有些情况下一个组件会在根节点上渲染另一个组件。例如，我们重构一下 `<MyButton>`，让它在根节点上渲染 `<BaseButton>`：

html
```html
<!-- <MyButton/> 的模板，只是渲染另一个组件 -->
<BaseButton />
```

此时 `<MyButton>` 接收的透传 attribute 会直接继续传给 `<BaseButton>`。

请注意：

1. 透传的 attribute 不会包含 `<MyButton>` 上声明过的 props 或是针对 `emits` 声明事件的 `v-on` 侦听函数，换句话说，声明过的 props 和侦听函数被 `<MyButton>`“消费” 了。

2. 透传的 attribute 若符合声明，也可以作为 props 传入 `<BaseButton>`。

3.5.2 禁用 Attributes 继承

如果你**不想要**一个组件自动地继承 attribute，你可以在组件选项中设置 `inheritAttrs: false`。

从 3.3 开始你也可以直接在 `<script setup>` 中使用 `defineOptions`：

html
```html
<script setup>
defineOptions({
    inheritAttrs: false
})
// ...setup 逻辑
</script>
```

The common scenario for disabling attribute inheritance is when attributes need to be applied to other elements besides the root node. By setting the `inheritAttrs` option to `false`, you can take full control over where the fallthrough attributes should be applied.

These fallthrough attributes can be accessed directly in template expressions as `$attrs`:

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

The `$attrs` object includes all attributes that are not declared by the component's `props` or `emits` options (e.g., `class`, `style`, `v-on` listeners, etc.).

Some notes:

- Unlike props, fallthrough attributes preserve their original casing in JavaScript, so an attribute like `foo-bar` needs to be accessed as `$attrs['foo-bar']`.
- A `v-on` event listener like `@click` will be exposed on the object as a function under `$attrs.onClick`.

Using our `<MyButton>` component example from the previous section - sometimes we may need to wrap the actual `<button>` element with an extra `<div>` for styling purposes:

```html
<div class="btn-wrapper">
  <button class="btn">click me</button>
</div>
```

We want all fallthrough attributes like `class` and `v-on` listeners to be applied to the inner `<button>`, not the outer `<div>`. We can achieve this with `inheritAttrs: false` and `v-bind="$attrs"`:

```html
<div class="btn-wrapper">
  <button class="btn" v-bind="$attrs">click me</button>
</div>
```

Remember that `v-bind` without an argument binds all the properties of an object as attributes of the target element.

3.5.3　Attribute Inheritance on Multiple Root Nodes

Unlike components with a single root node, components with multiple root nodes do not have an automatic attribute fallthrough behavior. If `$attrs` are not bound explicitly, a runtime warning

最常见的需要禁用 attribute 继承的场景就是 attribute 需要应用在根节点以外的其他元素上。通过设置 `inheritAttrs` 选项为 `false`，你可以完全控制透传进来的 attribute 被如何使用。

这些透传进来的 attribute 可以在模板的表达式中直接用 `$attrs` 访问到。

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

这个 `$attrs` 对象包含了除组件所声明的 `props` 和 `emits` 之外的所有其他 attribute，例如 `class`，`style`，`v-on` 监听器等等。

有几点需要注意：

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

现在我们要再次使用一下之前小节中的 `<MyButton>` 组件例子。有时候我们可能为了样式，需要在 `<button>` 元素外包装一层 `<div>`：

```html
<div class="btn-wrapper">
  <button class="btn">click me</button>
</div>
```

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

```html
<div class="btn-wrapper">
  <button class="btn" v-bind="$attrs">click me</button>
</div>
```

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

3.5.3　多根节点的 Attributes 继承

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

will be issued.

html

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

If `<CustomLayout>` has the following multi-root template, there will be a warning because Vue cannot be sure where to apply the fallthrough attributes:

html

```html
<header>...</header>
<main>...</main>
<footer>...</footer>
```

The warning will be suppressed if `$attrs` is explicitly bound:

html

```html
<header>...</header>
<main v-bind="$attrs">...</main>
<footer>...</footer>
```

3.5.4 Accessing Fallthrough Attributes in JavaScript

If needed, you can access a component's fallthrough attributes in `<script setup>` using the `useAttrs(` API:

html

```html
<script setup>
import { useAttrs } from 'vue'
const attrs = useAttrs()
</script>
```

If not using `<script setup>`, `attrs` will be exposed as a property of the `setup()` context:

js

```js
export default {
  setup(props, ctx) {
    // 透传 attribute 被暴露为 ctx.attrs
    console.log(ctx.attrs)
  }
}
```

html

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

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

html

```html
<header>...</header>
<main>...</main>
<footer>...</footer>
```

如果 `$attrs` 被显式绑定，则不会有警告：

html

```html
<header>...</header>
<main v-bind="$attrs">...</main>
<footer>...</footer>
```

3.5.4 在 JavaScript 中访问透传 Attributes

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

html

```html
<script setup>
import { useAttrs } from 'vue'
const attrs = useAttrs()
</script>
```

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

js

```js
export default {
  setup(props, ctx) {
    // 透传 attribute 被暴露为 ctx.attrs
    console.log(ctx.attrs)
  }
}
```

Note that although the `attrs` object here always reflects the latest fallthrough attributes, it isn't reactive (for performance reasons). You cannot use watchers to observe its changes. If you need reactivity, use a prop. Alternatively, you can use `onUpdated()` to perform side effects with the latest `attrs` on each update.

3.6 Slots

> This page assumes you've already read the Components Basics. Read that first if you are new to components.

3.6.1 Slot Content and Outlet

We have learned that components can accept props, which can be JavaScript values of any type. But how about template content? In some cases, we may want to pass a template fragment to a child component, and let the child component render the fragment within its own template.

For example, we may have a `<FancyButton>` component that supports usage like this:

```html
<FancyButton>
  Click me! <!-- 插槽内容 -->
</FancyButton>
```

The template of `<FancyButton>` looks like this:

```html
<button class="fancy-btn">
  <slot></slot> <!-- 插槽出口 -->
</button>
```

The `<slot>` element is a **slot outlet** that indicates where the parent-provided **slot content** should be rendered.

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

3.6 插槽 Slots

> 此章节假设你已经看过了组件基础。若你还不了解组件是什么，请先阅读该章节。

3.6.1 插槽内容与出口

在之前的章节中，我们已经了解到组件能够接收任意类型的 JavaScript 值作为 props，但组件要如何接收模板内容呢？在某些场景中，我们可能想要为子组件传递一些模板片段，让子组件在它们的组件中渲染这些片段。

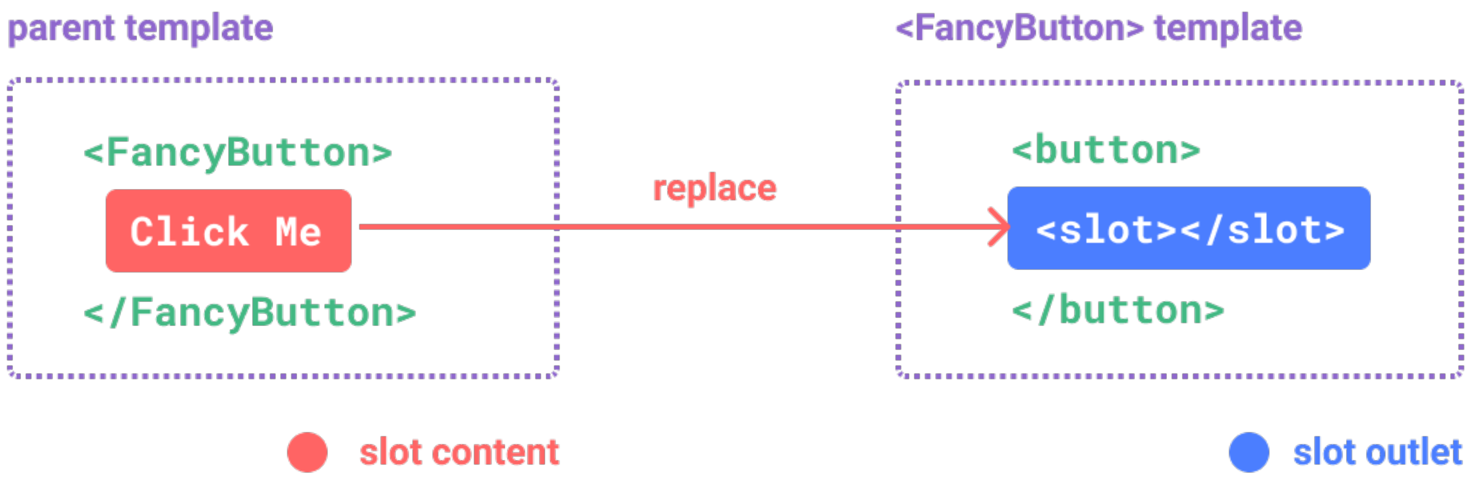
举例来说，这里有一个 `<FancyButton>` 组件，可以像这样使用：

```html
<FancyButton>
  Click me! <!-- 插槽内容 -->
</FancyButton>
```

而 `<FancyButton>` 的模板是这样的：

```html
<button class="fancy-btn">
  <slot></slot> <!-- 插槽出口 -->
</button>
```

`<slot>` 元素是一个**插槽出口** (slot outlet)，标示了父元素提供的**插槽内容** (slot content) 将在哪里被渲染。



And the final rendered DOM:

```html
<button class="fancy-btn">Click me!</button>
```

Try it in the Playground

With slots, the `<FancyButton>` is responsible for rendering the outer `<button>` (and its fancy styling), while the inner content is provided by the parent component.

Another way to understand slots is by comparing them to JavaScript functions:

```js
// 父元素传入插槽内容
FancyButton('Click me!')
// FancyButton 在自己的模板中渲染插槽内容
function FancyButton(slotContent) {
  return `<button class="fancy-btn">
      ${slotContent}
    </button>`
}
```

Slot content is not just limited to text. It can be any valid template content. For example, we can

最终渲染出的 DOM 是这样：

```html
<button class="fancy-btn">Click me!</button>
```

在演练场中尝试一下

通过使用插槽，`<FancyButton>` 仅负责渲染外层的 `<button>`（以及相应的样式），而其内部的内容由父组件提供。

理解插槽的另一种方式是和下面的 JavaScript 函数作类比，其概念是类似的：

```js
// 父元素传入插槽内容
FancyButton('Click me!')
// FancyButton 在自己的模板中渲染插槽内容
function FancyButton(slotContent) {
  return `<button class="fancy-btn">
      ${slotContent}
    </button>`
}
```

插槽内容可以是任意合法的模板内容，不局限于文本。例如我们可以传入多个元

pass in multiple elements, or even other components:

```html
<FancyButton>
  <span style="color:red">Click me!</span>
  <AwesomeIcon name="plus" />
</FancyButton>
```

Try it in the Playground

By using slots, our `<FancyButton>` is more flexible and reusable. We can now use it in different places with different inner content, but all with the same fancy styling.

Vue components' slot mechanism is inspired by the native Web Component `<slot>` element, but with additional capabilities that we will see later.

3.6.2 Render Scope

Slot content has access to the data scope of the parent component, because it is defined in the parent. For example:

```html
<span>{{ message }}</span>
<FancyButton>{{ message }}</FancyButton>
```

Here both `{{ message }}` interpolations will render the same content.

Slot content does **not** have access to the child component's data. Expressions in Vue templates can only access the scope it is defined in, consistent with JavaScript's lexical scoping. In other words:

> Expressions in the parent template only have access to the parent scope; expressions in the child template only have access to the child scope.

3.6.3 Fallback Content

There are cases when it's useful to specify fallback (i.e. default) content for a slot, to be rendered only when no content is provided. For example, in a `<SubmitButton>` component:

```html
<button type="submit">
    <slot></slot>
```

素，甚至是组件：

```html
<FancyButton>
  <span style="color:red">Click me!</span>
  <AwesomeIcon name="plus" />
</FancyButton>
```

在演练场中尝试一下

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

Vue 组件的插槽机制是受原生 Web Component `<slot>` 元素的启发而诞生，同时还做了一些功能拓展，这些拓展的功能我们后面会学习到。

3.6.2 渲染作用域

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

```html
<span>{{ message }}</span>
<FancyButton>{{ message }}</FancyButton>
```

这里的两个 `{{ message }}` 插值表达式渲染的内容都是一样的。

插槽内容**无法访问**子组件的数据。Vue 模板中的表达式只能访问其定义时所处的作用域，这和 JavaScript 的词法作用域规则是一致的。换言之：

> 父组件模板中的表达式只能访问父组件的作用域；子组件模板中的表达式只能访问子组件的作用域。

3.6.3 默认内容

在外部没有提供任何内容的情况下，可以为插槽指定默认内容。比如有这样一个 `<SubmitButton>` 组件：

```html
<button type="submit">
    <slot></slot>
```

```html
</button>
```

We might want the text ”Submit” to be rendered inside the `<button>` if the parent didn’t provide any slot content. To make ”Submit” the fallback content, we can place it in between the `<slot>` tags:

```html
<button type="submit">
    <slot>
    Submit <!-- 默认内容 -->
    </slot>
</button>
```

Now when we use `<SubmitButton>` in a parent component, providing no content for the slot:

```html
<SubmitButton />
```

This will render the fallback content, ”Submit”:

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

But if we provide content:

```html
<SubmitButton>Save</SubmitButton>
```

Then the provided content will be rendered instead:

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

Try it in the Playground

3.6.4 Named Slots

There are times when it’s useful to have multiple slot outlets in a single component. For example, in a `<BaseLayout>` component with the following template:

```html
<div class="container">
    <header>
    <!-- 标题内容放这里 -->
```

```html
</button>
```

如果我们想在父组件没有提供任何插槽内容时在 `<button>` 内渲染 “Submit”，只需要将 “Submit” 写在 `<slot>` 标签之间来作为默认内容：

```html
<button type="submit">
    <slot>
    Submit <!-- 默认内容 -->
    </slot>
</button>
```

现在，当我们在父组件中使用 `<SubmitButton>` 且没有提供任何插槽内容时：

```html
<SubmitButton />
```

“Submit” 将会被作为默认内容渲染：

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

但如果我们提供了插槽内容：

```html
<SubmitButton>Save</SubmitButton>
```

那么被显式提供的内容会取代默认内容：

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

在演练场中尝试一下

3.6.4 具名插槽

有时在一个组件中包含多个插槽出口是很有用的。举例来说，在一个 `<BaseLayout>` 组件中，有如下模板：

```html
<div class="container">
    <header>
    <!-- 标题内容放这里 -->
    </header>
    <main>
    <!-- 主要内容放这里 -->
    </main>
    <footer>
    <!-- 底部内容放这里 -->
    </footer>
</div>
```

For these cases, the `<slot>` element has a special attribute, `name`, which can be used to assign a unique ID to different slots so you can determine where content should be rendered:

```html
<div class="container">
    <header>
    <slot name="header"></slot>
    </header>
    <main>
    <slot></slot>
    </main>
    <footer>
    <slot name="footer"></slot>
    </footer>
</div>
```

A `<slot>` outlet without `name` implicitly has the name "default".

In a parent component using `<BaseLayout>`, we need a way to pass multiple slot content fragments, each targeting a different slot outlet. This is where **named slots** come in.

To pass a named slot, we need to use a `<template>` element with the `v-slot` directive, and then pass the name of the slot as an argument to `v-slot`:

```html
<BaseLayout>
  <template v-slot:header>
    <!-- header 插槽的内容放这里 -->
  </template>
```

```
    </header>
    <main>
    <!-- 主要内容放这里 -->
    </main>
    <footer>
    <!-- 底部内容放这里 -->
    </footer>
</div>
```

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

```html
<div class="container">
    <header>
    <slot name="header"></slot>
    </header>
    <main>
    <slot></slot>
    </main>
    <footer>
    <slot name="footer"></slot>
    </footer>
</div>
```

这类带 `name` 的插槽被称为具名插槽 (named slots)。没有提供 `name` 的 `<slot>` 出口会隐式地命名为 “default”。

在父组件中使用 `<BaseLayout>` 时，我们需要一种方式将多个插槽内容传入到各自目标插槽的出口。此时就需要用到**具名插槽**了：

要为具名插槽传入内容，我们需要使用一个含 `v-slot` 指令的 `<template>` 元素，并将目标插槽的名字传给该指令：

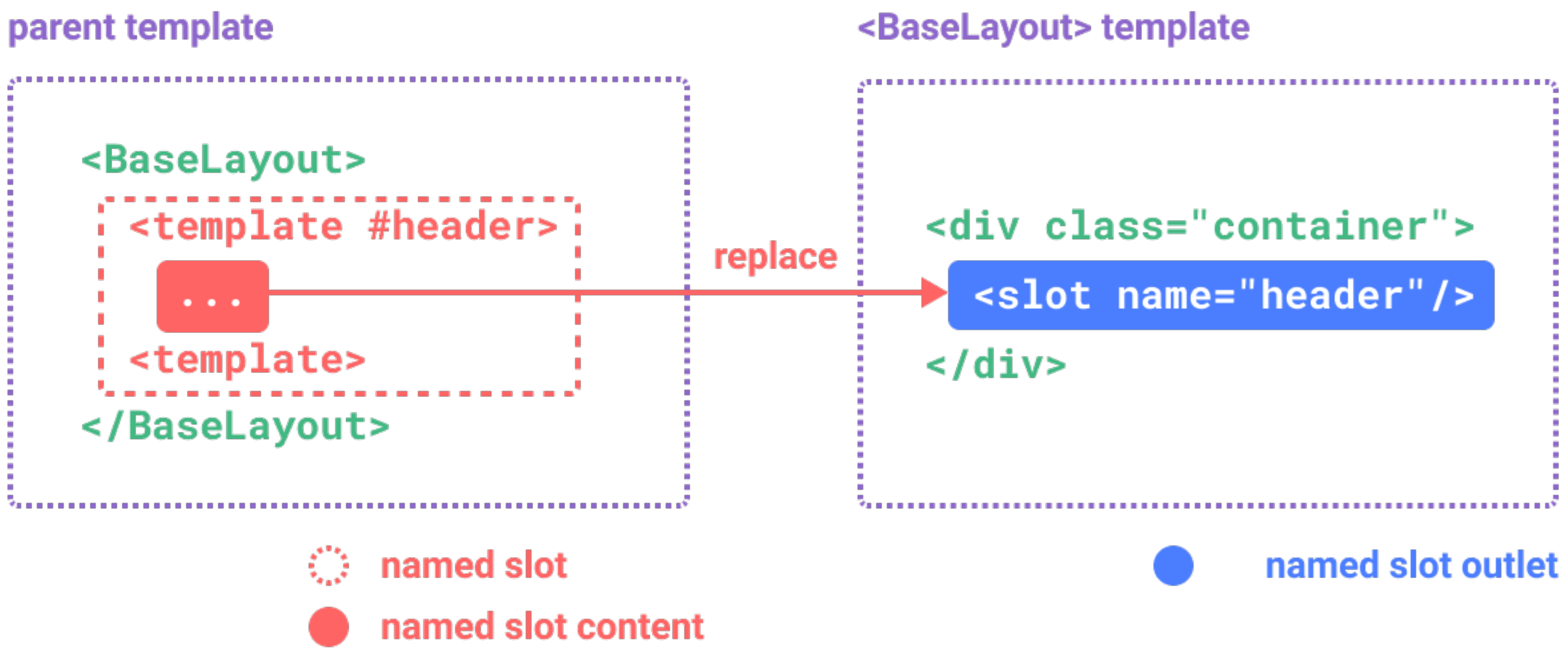
```html
<BaseLayout>
  <template v-slot:header>
    <!-- header 插槽的内容放这里 -->
  </template>
```

```
</BaseLayout>
```

v-slot has a dedicated shorthand #, so <template v-slot:header> can be shortened to just <template #header>. Think of it as ”render this template fragment in the child component’s ’header’ slot”.

```
</BaseLayout>
```

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



Here’s the code passing content for all three slots to <BaseLayout> using the shorthand syntax:

```html
<BaseLayout>
    <template #header>
    <h1>Here might be a page title</h1>
    </template>
    <template #default>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
    </template>
```

下面我们给出完整的、向 <BaseLayout> 传递插槽内容的代码，指令均使用的是缩写形式：

```html
<BaseLayout>
    <template #header>
    <h1>Here might be a page title</h1>
    </template>
    <template #default>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
    </template>
```

```html
  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

When a component accepts both a default slot and named slots, all top-level non-`<template>` nodes are implicitly treated as content for the default slot. So the above can also be written as:

html

```html
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>
  <!-- 隐式的默认插槽 -->
  <p>A paragraph for the main content.</p>
  <p>And another one.</p>
  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

Now everything inside the `<template>` elements will be passed to the corresponding slots. The final rendered HTML will be:

html

```html
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

Try it in the Playground

```html
  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

当一个组件同时接收默认插槽和具名插槽时，所有位于顶级的非 `<template>` 节点都被隐式地视为默认插槽的内容。所以上面也可以写成：

html

```html
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>
  <!-- 隐式的默认插槽 -->
  <p>A paragraph for the main content.</p>
  <p>And another one.</p>
  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

现在 `<template>` 元素中的所有内容都将被传递到相应的插槽。最终渲染出的 HTML 如下：

html

```html
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

在演练场中尝试一下

Again, it may help you understand named slots better using the JavaScript function analogy:

```js
// 传入不同的内容给不同名字的插槽
BaseLayout({
  header: `...`,
  default: `...`,
  footer: `...`
})
// <BaseLayout> 渲染插槽内容到对应位置
function BaseLayout(slots) {
  return `<div class="container">
      <header>${slots.header}</header>
      <main>${slots.default}</main>
      <footer>${slots.footer}</footer>
    </div>`
}
```

3.6.5 Dynamic Slot Names

Dynamic directive arguments also work on `v-slot`, allowing the definition of dynamic slot names:

```html
<base-layout>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>
  <!-- 缩写为 -->
  <template #[dynamicSlotName]>
    ...
  </template>
</base-layout>
```

Do note the expression is subject to the syntax constraints of dynamic directive arguments.

3.6.6 Scoped Slots

As discussed in Render Scope, slot content does not have access to state in the child component.

使用 JavaScript 函数来类比可能更有助于你来理解具名插槽：

```js
// 传入不同的内容给不同名字的插槽
BaseLayout({
  header: `...`,
  default: `...`,
  footer: `...`
})
// <BaseLayout> 渲染插槽内容到对应位置
function BaseLayout(slots) {
  return `<div class="container">
      <header>${slots.header}</header>
      <main>${slots.default}</main>
      <footer>${slots.footer}</footer>
    </div>`
}
```

3.6.5 动态插槽名

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

```html
<base-layout>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>
  <!-- 缩写为 -->
  <template #[dynamicSlotName]>
    ...
  </template>
</base-layout>
```

注意这里的表达式和动态指令参数受相同的语法限制。

3.6.6 作用域插槽

在上面的渲染作用域中我们讨论到，插槽的内容无法访问到子组件的状态。

However, there are cases where it could be useful if a slot's content can make use of data from both the parent scope and the child scope. To achieve that, we need a way for the child to pass data to a slot when rendering it.

In fact, we can do exactly that - we can pass attributes to a slot outlet just like passing props to a component:

html
```html
<!-- <MyComponent> 的模板 -->
<div>
  <slot :text="greetingMessage" :count="1"></slot>
</div>
```

Receiving the slot props is a bit different when using a single default slot vs. using named slots. We are going to show how to receive props using a single default slot first, by using `v-slot` directly on the child component tag:

html
```html
<MyComponent v-slot="slotProps">
  {{ slotProps.text }} {{ slotProps.count }}
</MyComponent>
```

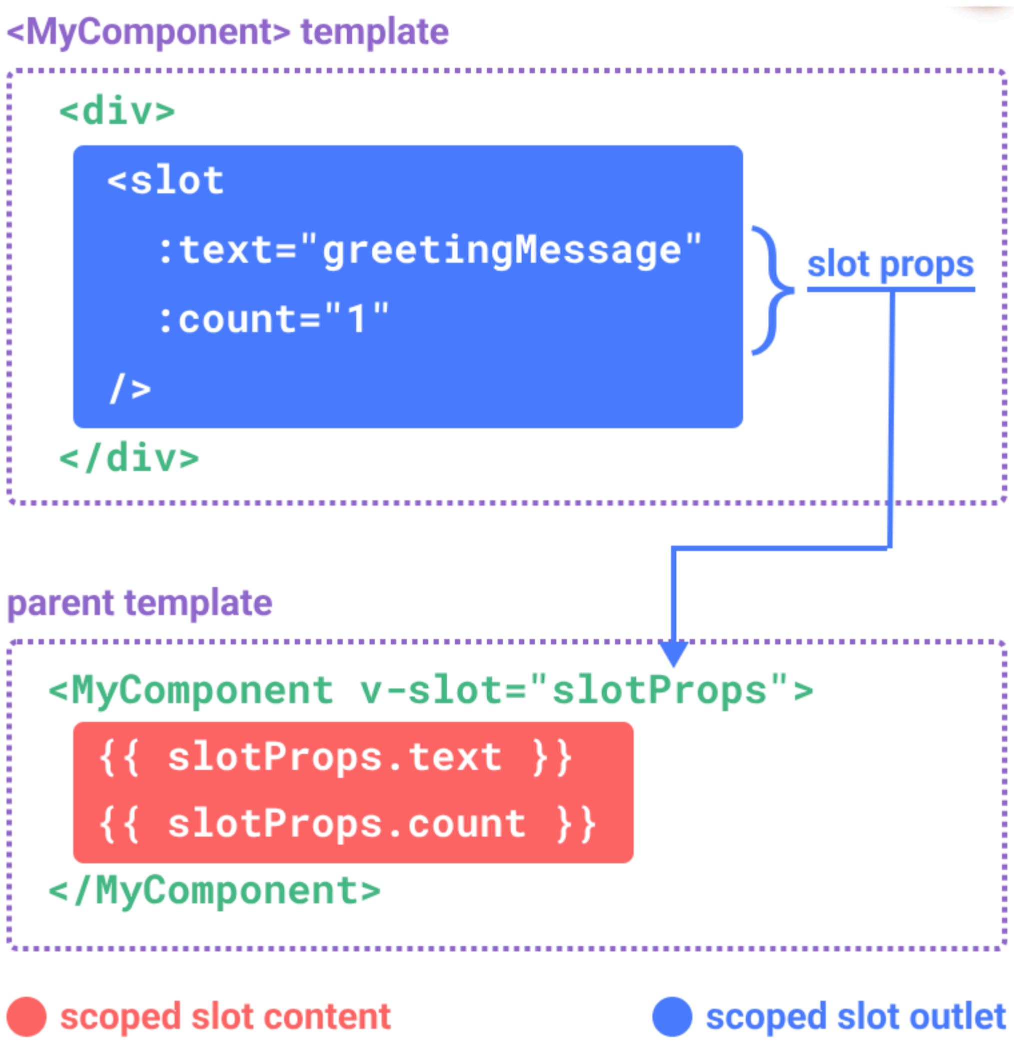
然而在某些场景下插槽的内容可能想要同时使用父组件域内和子组件域内的数据。要做到这一点，我们需要一种方法来让子组件在渲染时将一部分数据提供给插槽。

我们也确实有办法这么做！可以像对组件传递 props 那样，向一个插槽的出口上传递 attributes：

html
```html
<!-- <MyComponent> 的模板 -->
<div>
  <slot :text="greetingMessage" :count="1"></slot>
</div>
```

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

html
```html
<MyComponent v-slot="slotProps">
  {{ slotProps.text }} {{ slotProps.count }}
</MyComponent>
```



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

```
<MyComponent v-slot="slotProps">
  {{ slotProps.text }}
  {{ slotProps.count }}
</MyComponent>
```

Try it in the Playground

The props passed to the slot by the child are available as the value of the corresponding `v-slot` directive, which can be accessed by expressions inside the slot.

You can think of a scoped slot as a function being passed into the child component. The child component then calls it, passing props as arguments:

```js
MyComponent({
  // 类比默认插槽，将其想成一个函数
  default: (slotProps) => {
    return `${slotProps.text} ${slotProps.count}`
```

在演练场中尝试一下

子组件传入插槽的 props 作为了 `v-slot` 指令的值，可以在插槽内的表达式中访问。

你可以将作用域插槽类比为一个传入子组件的函数。子组件会将相应的 props 作为参数传给它：

```js
MyComponent({
  // 类比默认插槽，将其想成一个函数
  default: (slotProps) => {
    return `${slotProps.text} ${slotProps.count}`
  }
})
function MyComponent(slots) {
  const greetingMessage = 'hello'
  return `<div>${
    // 在插槽函数调用时传入 props
    slots.default({ text: greetingMessage, count: 1 })
  }</div>`
}
```

In fact, this is very close to how scoped slots are compiled, and how you would use scoped slots in manual render functions.

Notice how `v-slot="slotProps"` matches the slot function signature. Just like with function arguments, we can use destructuring in `v-slot`:

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

Named Scoped Slots

Named scoped slots work similarly - slot props are accessible as the value of the `v-slot` directive: `v-slot:name="slotProps"`. When using the shorthand, it looks like this:

```html
<MyComponent>
  <template #header="headerProps">
    {{ headerProps }}
  </template>
  <template #default="defaultProps">
    {{ defaultProps }}
  </template>
  <template #footer="footerProps">
    {{ footerProps }}
  </template>
</MyComponent>
```

```
  }
})
function MyComponent(slots) {
  const greetingMessage = 'hello'
  return `<div>${
    // 在插槽函数调用时传入 props
    slots.default({ text: greetingMessage, count: 1 })
  }</div>`
}
```

实际上，这已经和作用域插槽的最终代码编译结果、以及手动编写渲染函数时使用作用域插槽的方式非常类似了。

`v-slot="slotProps"` 可以类比这里的函数签名，和函数的参数类似，我们也可以在 `v-slot` 中使用解构：

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

具名作用域插槽

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

```html
<MyComponent>
  <template #header="headerProps">
    {{ headerProps }}
  </template>
  <template #default="defaultProps">
    {{ defaultProps }}
  </template>
  <template #footer="footerProps">
    {{ footerProps }}
  </template>
</MyComponent>
```

Passing props to a named slot:

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

Note the `name` of a slot won't be included in the props because it is reserved - so the resulting `headerProps` would be `{ message: 'hello' }`.

If you are mixing named slots with the default scoped slot, you need to use an explicit `<template>` tag for the default slot. Attempting to place the `v-slot` directive directly on the component will result in a compilation error. This is to avoid any ambiguity about the scope of the props of the default slot. For example:

```html
<!-- 该模板无法编译 -->
<template>
  <MyComponent v-slot="{ message }">
    <p>{{ message }}</p>
    <template #footer>
      <!-- message 属于默认插槽，此处不可用 -->
      <p>{{ message }}</p>
    </template>
  </MyComponent>
</template>
```

Using an explicit `<template>` tag for the default slot helps to make it clear that the `message` prop is not available inside the other slot:

```html
<template>
  <MyComponent>
    <!-- 使用显式的默认插槽 -->
    <template #default="{ message }">
      <p>{{ message }}</p>
    </template>
    <template #footer>
      <p>Here's some contact info</p>
    </template>
  </MyComponent>
</template>
```

向具名插槽中传入 props：

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

注意插槽上的 `name` 是一个 Vue 特别保留的 attribute，不会作为 props 传递给插槽。因此最终 `headerProps` 的结果是 `{ message: 'hello' }`。

如果你同时使用了具名插槽与默认插槽，则需要为默认插槽使用显式的 `<template>` 标签。尝试直接为组件添加 `v-slot` 指令将导致编译错误。这是为了避免因默认插槽的 props 的作用域而困惑。举例：

```html
<!-- 该模板无法编译 -->
<template>
  <MyComponent v-slot="{ message }">
    <p>{{ message }}</p>
    <template #footer>
      <!-- message 属于默认插槽，此处不可用 -->
      <p>{{ message }}</p>
    </template>
  </MyComponent>
</template>
```

为默认插槽使用显式的 `<template>` 标签有助于更清晰地指出 `message` 属性在其他插槽中不可用：

```html
<template>
  <MyComponent>
    <!-- 使用显式的默认插槽 -->
    <template #default="{ message }">
      <p>{{ message }}</p>
    </template>
    <template #footer>
      <p>Here's some contact info</p>
    </template>
  </MyComponent>
</template>
```

Fancy List Example

You may be wondering what would be a good use case for scoped slots. Here's an example: imagine a `<FancyList>` component that renders a list of items - it may encapsulate the logic for loading remote data, using the data to display a list, or even advanced features like pagination or infinite scrolling. However, we want it to be flexible with how each item looks and leave the styling of each item to the parent component consuming it. So the desired usage may look like this:

html

```html
<FancyList :api-url="url" :per-page="10">
  <template #item="{ body, username, likes }">
    <div class="item">
      <p>{{ body }}</p>
      <p>by {{ username }} | {{ likes }} likes</p>
    </div>
  </template>
</FancyList>
```

Inside `<FancyList>`, we can render the same `<slot>` multiple times with different item data (notice we are using `v-bind` to pass an object as slot props):

html

```html
<ul>
  <li v-for="item in items">
    <slot name="item" v-bind="item"></slot>
  </li>
</ul>
```

Try it in the Playground

Renderless Components

The `<FancyList>` use case we discussed above encapsulates both reusable logic (data fetching, pagination etc.) and visual output, while delegating part of the visual output to the consumer component via scoped slots.

If we push this concept a bit further, we can come up with components that only encapsulate logic and do not render anything by themselves - visual output is fully delegated to the consumer component with scoped slots. We call this type of component a **Renderless Component**.

高级列表组件示例

你可能想问什么样的场景才适合用到作用域插槽,这里我们来看一个 `<FancyList>` 组件的例子。它会渲染一个列表,并同时会封装一些加载远端数据的逻辑、使用数据进行列表渲染、或者是像分页或无限滚动这样更进阶的功能。然而我们希望它能够保留足够的灵活性,将对单个列表元素内容和样式的控制权留给使用它的父组件。我们期望的用法可能是这样的:

html

```html
<FancyList :api-url="url" :per-page="10">
  <template #item="{ body, username, likes }">
    <div class="item">
      <p>{{ body }}</p>
      <p>by {{ username }} | {{ likes }} likes</p>
    </div>
  </template>
</FancyList>
```

在 `<FancyList>` 之中,我们可以多次渲染 `<slot>` 并每次都提供不同的数据 (注意我们这里使用了 `v-bind` 来传递插槽的 props):

html

```html
<ul>
  <li v-for="item in items">
    <slot name="item" v-bind="item"></slot>
  </li>
</ul>
```

在演练场中尝试一下

无渲染组件

上面的 `<FancyList>` 案例同时封装了可重用的逻辑 (数据获取、分页等) 和视图输出,但也将部分视图输出通过作用域插槽交给了消费者组件来管理。

如果我们将这个概念拓展一下,可以想象的是,一些组件可能只包括了逻辑而不需要自己渲染内容,视图输出通过作用域插槽全权交给了消费者组件。我们将这种类型的组件称为**无渲染组件**。

An example renderless component could be one that encapsulates the logic of tracking the current mouse position:

html
```html
<MouseTracker v-slot="{ x, y }">
  Mouse is at: {{ x }}, {{ y }}
</MouseTracker>
```

Try it in the Playground

While an interesting pattern, most of what can be achieved with Renderless Components can be achieved in a more efficient fashion with Composition API, without incurring the overhead of extra component nesting. Later, we will see how we can implement the same mouse tracking functionality as a Composable.

That said, scoped slots are still useful in cases where we need to both encapsulate logic **and** compose visual output, like in the `<FancyList>` example.

3.7 Provide / Inject

> This page assumes you've already read the Components Basics. Read that first if you are new to components.

3.7.1 Prop Drilling

Usually, when we need to pass data from the parent to a child component, we use props. However, imagine the case where we have a large component tree, and a deeply nested component needs something from a distant ancestor component. With only props, we would have to pass the same prop across the entire parent chain:

这里有一个无渲染组件的例子，一个封装了追踪当前鼠标位置逻辑的组件：

html
```html
<MouseTracker v-slot="{ x, y }">
  Mouse is at: {{ x }}, {{ y }}
</MouseTracker>
```

在演练场中尝试一下

虽然这个模式很有趣,但大部分能用无渲染组件实现的功能都可以通过组合式 API 以另一种更高效的方式实现，并且还不会带来额外组件嵌套的开销。之后我们会在组合式函数一章中介绍如何更高效地实现追踪鼠标位置的功能。

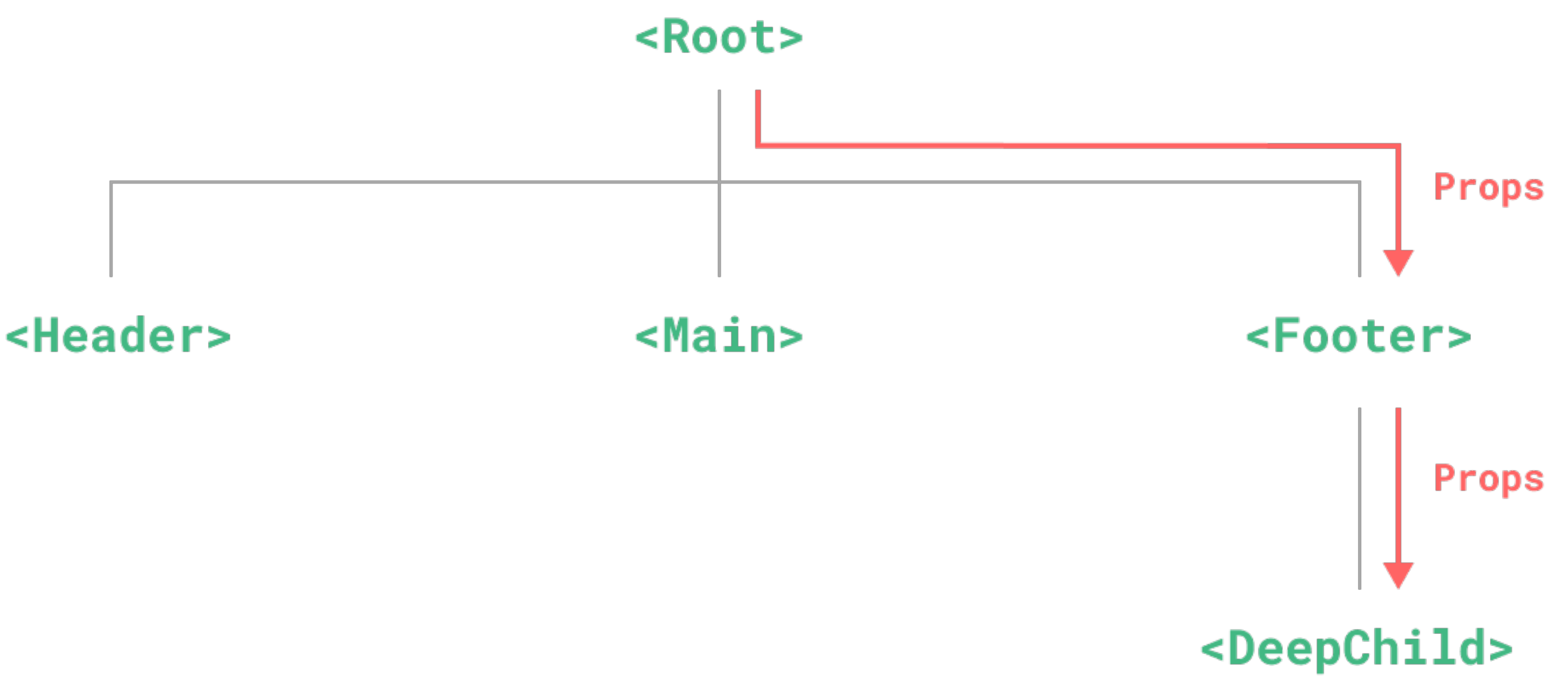
尽管如此，作用域插槽在需要**同时**封装逻辑、组合视图界面时还是很有用，就像上面的 `<FancyList>` 组件那样。

3.7 依赖注入

> 此章节假设你已经看过了组件基础。若你还不了解组件是什么，请先阅读该章节。

3.7.1 Prop 逐级透传问题

通常情况下，当我们需要从父组件向子组件传递数据时，会使用 props。想象一下这样的结构：有一些多层级嵌套的组件，形成了一颗巨大的组件树，而某个深层的子组件需要一个较远的祖先组件中的部分数据。在这种情况下，如果仅使用 props 则必须将其沿着组件链逐级传递下去，这会非常麻烦：

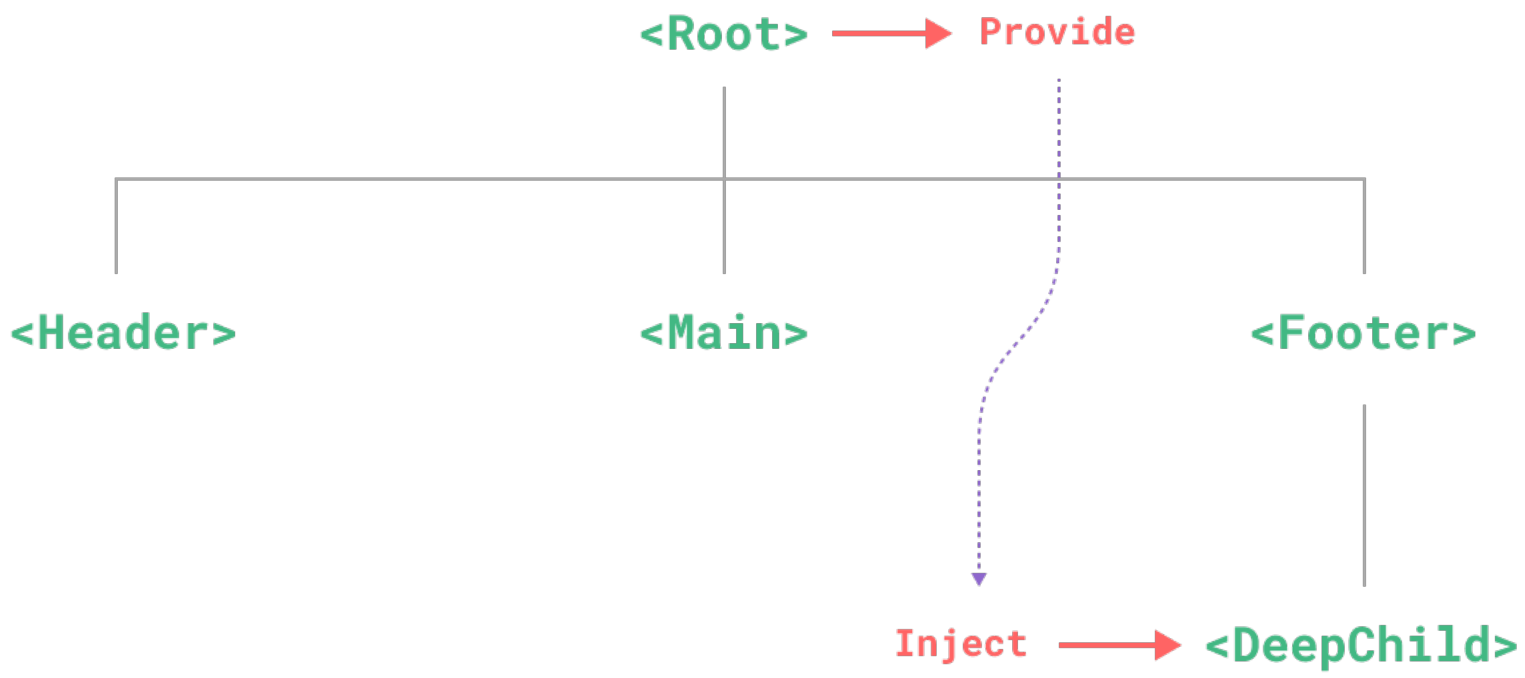


Notice although the `<Footer>` component may not care about these props at all, it still needs to declare and pass them along just so `<DeepChild>` can access them. If there is a longer parent chain, more components would be affected along the way. This is called ”props drilling” and definitely isn’t fun to deal with.

We can solve props drilling with `provide` and `inject`. A parent component can serve as a **dependency provider** for all its descendants. Any component in the descendant tree, regardless of how deep it is, can **inject** dependencies provided by components up in its parent chain.

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

`provide` 和 `inject` 可以帮助我们解决这一问题。[1] 一个父组件相对于其所有的后代组件，会作为**依赖提供者**。任何后代的组件树，无论层级有多深，都可以**注入**由父组件提供给整条链路的依赖。



3.7.2 Provide

To provide data to a component's descendants, use the `provide()` function:

```html
<script setup>
import { provide } from 'vue'
provide(/* 注入名 */ 'message', /* 值 */ 'hello!')
</script>
```

If not using `<script setup>`, make sure `provide()` is called synchronously inside `setup()`:

```js
import { provide } from 'vue'
export default {
  setup() {
    provide(/* 注入名 */ 'message', /* 值 */ 'hello!')
  }
}
```

3.7.2 Provide (提供)

要为组件后代提供数据，需要使用到 `provide()` 函数：

```html
<script setup>
import { provide } from 'vue'
provide(/* 注入名 */ 'message', /* 值 */ 'hello!')
</script>
```

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

```js
import { provide } from 'vue'
export default {
  setup() {
    provide(/* 注入名 */ 'message', /* 值 */ 'hello!')
  }
}
```

The `provide()` function accepts two arguments. The first argument is called the **injection key**, which can be a string or a `Symbol`. The injection key is used by descendant components to lookup the desired value to inject. A single component can call `provide()` multiple times with different injection keys to provide different values.

The second argument is the provided value. The value can be of any type, including reactive state such as refs:

js
```js
import { ref, provide } from 'vue'
const count = ref(0)
provide('key', count)
```

Providing reactive values allows the descendant components using the provided value to establish a reactive connection to the provider component.

3.7.3 App-level Provide

In addition to providing data in a component, we can also provide at the app level:

js
```js
import { createApp } from 'vue'
const app = createApp({})
app.provide(/* 注入名 */ 'message', /* 值 */ 'hello!')
```

App-level provides are available to all components rendered in the app. This is especially useful when writing plugins, as plugins typically wouldn't be able to provide values using components.

3.7.4 Inject

To inject data provided by an ancestor component, use the `inject()` function:

html
```html
<script setup>
import { inject } from 'vue'
const message = inject('message')
</script>
```

If the provided value is a ref, it will be injected as-is and will **not** be automatically unwrapped. This allows the injector component to retain the reactivity connection to the provider component.

`provide()` 函数接收两个参数。第一个参数被称为**注入名**，可以是一个字符串或是一个 `Symbol`。后代组件会用注入名来查找期望注入的值。一个组件可以多次调用 `provide()`，使用不同的注入名，注入不同的依赖值。

第二个参数是提供的值，值可以是任意类型，包括响应式的状态，比如一个 ref：

js
```js
import { ref, provide } from 'vue'
const count = ref(0)
provide('key', count)
```

提供的响应式状态使后代组件可以由此和提供者建立响应式的联系。

3.7.3 应用层 Provide

除了在一个组件中提供依赖，我们还可以在整个应用层面提供依赖：

js
```js
import { createApp } from 'vue'
const app = createApp({})
app.provide(/* 注入名 */ 'message', /* 值 */ 'hello!')
```

在应用级别提供的数据在该应用内的所有组件中都可以注入。这在你编写插件时会特别有用，因为插件一般都不会使用组件形式来提供值。

3.7.4 Inject (注入)

要注入上层组件提供的数据，需使用 `inject()` 函数：

html
```html
<script setup>
import { inject } from 'vue'
const message = inject('message')
</script>
```

如果提供的值是一个 ref，注入进来的会是该 ref 对象，而**不会**自动解包为其内部的值。这使得注入方组件能够通过 ref 对象保持了和供给方的响应性链接。

Full provide + inject Example with Reactivity

Again, if not using `<script setup>`, `inject()` should only be called synchronously inside `setup()`:

js

```js
import { inject } from 'vue'
export default {
  setup() {
    const message = inject('message')
    return { message }
  }
}
```

Injection Default Values

By default, `inject` assumes that the injected key is provided somewhere in the parent chain. In the case where the key is not provided, there will be a runtime warning.

If we want to make an injected property work with optional providers, we need to declare a default value, similar to props:

js

```js
// 如果没有祖先组件提供 "message"
// `value` 会是 " 这是默认值"
const value = inject('message', '这是默认值')
```

In some cases, the default value may need to be created by calling a function or instantiating a new class. To avoid unnecessary computation or side effects in case the optional value is not used, we can use a factory function for creating the default value:

js

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

The third parameter indicates the default value should be treated as a factory function.

3.7.5 Working with Reactivity

When using reactive provide / inject values, **it is recommended to keep any mutations to reactive state inside of the *provider* whenever possible**. This ensures that the provided state and its possible mutations are co-located in the same component, making it easier to maintain

带有响应性的 provide + inject 完整示例

同样的，如果没有使用 `<script setup>`，`inject()` 需要在 `setup()` 内同步调用：

js

```js
import { inject } from 'vue'
export default {
  setup() {
    const message = inject('message')
    return { message }
  }
}
```

注入默认值

默认情况下，`inject` 假设传入的注入名会被某个祖先链上的组件提供。如果该注入名的确没有任何组件提供，则会抛出一个运行时警告。

如果在注入一个值时不要求必须有提供者，那么我们应该声明一个默认值，和 props 类似：

js

```js
// 如果没有祖先组件提供 "message"
// `value` 会是 " 这是默认值"
const value = inject('message', '这是默认值')
```

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

js

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

第三个参数表示默认值应该被当作一个工厂函数。

3.7.5 和响应式数据配合使用

当提供 / 注入响应式的数据时，**建议尽可能将任何对响应式状态的变更都保持在供给方组件中**。这样可以确保所提供状态的声明和变更操作都内聚在同一个组件内，使其更容易维护。

in the future.

There may be times when we need to update the data from an injector component. In such cases, we recommend providing a function that is responsible for mutating the state:

```html
<!-- 在供给方组件内 -->
<script setup>
import { provide, ref } from 'vue'
const location = ref('North Pole')
function updateLocation() {
  location.value = 'South Pole'
}
provide('location', {
  location,
  updateLocation
})
</script>
```

```html
<!-- 在注入方组件 -->
<script setup>
import { inject } from 'vue'
const { location, updateLocation } = inject('location')
</script>
<template>
  <button @click="updateLocation">{{ location }}</button>
</template>
```

Finally, you can wrap the provided value with `readonly()` if you want to ensure that the data passed through `provide` cannot be mutated by the injector component.

```html
<script setup>
import { ref, provide, readonly } from 'vue'
const count = ref(0)
provide('read-only-count', readonly(count))
</script>
```

有的时候，我们可能需要在注入方组件中更改数据。在这种情况下，我们推荐在供给方组件内声明并提供一个更改数据的方法函数：

```html
<!-- 在供给方组件内 -->
<script setup>
import { provide, ref } from 'vue'
const location = ref('North Pole')
function updateLocation() {
  location.value = 'South Pole'
}
provide('location', {
  location,
  updateLocation
})
</script>
```

```html
<!-- 在注入方组件 -->
<script setup>
import { inject } from 'vue'
const { location, updateLocation } = inject('location')
</script>
<template>
  <button @click="updateLocation">{{ location }}</button>
</template>
```

最后，如果你想确保提供的数据不能被注入方的组件更改，你可以使用 `readonly()` 来包装提供的值。

```html
<script setup>
import { ref, provide, readonly } from 'vue'
const count = ref(0)
provide('read-only-count', readonly(count))
</script>
```

3.7.6 Working with Symbol Keys

So far, we have been using string injection keys in the examples. If you are working in a large application with many dependency providers, or you are authoring components that are going to be used by other developers, it is best to use Symbol injection keys to avoid potential collisions.

It's recommended to export the Symbols in a dedicated file:

```js
// keys.js
export const myInjectionKey = Symbol()
```

```html
// 在供给方组件中
import { provide } from 'vue'
import { myInjectionKey } from './keys.js'
provide(myInjectionKey, { /*
  要提供的数据
*/ });
```

```js
// 注入方组件
import { inject } from 'vue'
import { myInjectionKey } from './keys.js'
const injected = inject(myInjectionKey)
```

See also: Typing Provide / Inject

Edit this page on GitHub

3.8 Async Components

3.8.1 Basic Usage

In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it's needed. To make that possible, Vue has a `defineAsyncComponent` function:

```js
import { defineAsyncComponent } from 'vue'
const AsyncComp = defineAsyncComponent(() => {
```

3.7.6 使用 Symbol 作注入名

至此，我们已经了解了如何使用字符串作为注入名。但如果你正在构建大型的应用，包含非常多的依赖提供，或者你正在编写提供给其他开发者使用的组件库，建议最好使用 Symbol 来作为注入名以避免潜在的冲突。

我们通常推荐在一个单独的文件中导出这些注入名 Symbol：

```js
// keys.js
export const myInjectionKey = Symbol()
```

```html
// 在供给方组件中
import { provide } from 'vue'
import { myInjectionKey } from './keys.js'
provide(myInjectionKey, { /*
  要提供的数据
*/ });
```

```js
// 注入方组件
import { inject } from 'vue'
import { myInjectionKey } from './keys.js'
const injected = inject(myInjectionKey)
```

TypeScript 用户请参考：为 Provide / Inject 标注类型

译者注 [1] 在本章及后续章节中，“**提供**” 将成为对应 Provide 的一个专有概念

3.8 异步组件

3.8.1 基本用法

在大型项目中，我们可能需要拆分应用为更小的块，并仅在需要时再从服务器加载相关组件。Vue 提供了 `defineAsyncComponent` 方法来实现此功能：

```js
import { defineAsyncComponent } from 'vue'
const AsyncComp = defineAsyncComponent(() => {
  return new Promise((resolve, reject) => {
    // ... 从服务器获取组件
    resolve(/* 获取到的组件 */)
  })
})
// ... 像使用其他一般组件一样使用 `AsyncComp`
```

As you can see, `defineAsyncComponent` accepts a loader function that returns a Promise. The Promise's `resolve` callback should be called when you have retrieved your component definition from the server. You can also call `reject(reason)` to indicate the load has failed.

ES module dynamic import also returns a Promise, so most of the time we will use it in combination with `defineAsyncComponent`. Bundlers like Vite and webpack also support the syntax (and will use it as bundle split points), so we can use it to import Vue SFCs:

js

```js
import { defineAsyncComponent } from 'vue'
const AsyncComp = defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
)
```

The resulting `AsyncComp` is a wrapper component that only calls the loader function when it is actually rendered on the page. In addition, it will pass along any props and slots to the inner component, so you can use the async wrapper to seamlessly replace the original component while achieving lazy loading.

As with normal components, async components can be registered globally using `app.component()`:

js

```js
app.component('MyComponent', defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
))
```

They can also be defined directly inside their parent component:

html

```html
<script setup>
import { defineAsyncComponent } from 'vue'
const AdminPage = defineAsyncComponent(() =>
  import('./components/AdminPageComponent.vue')
)
```

```
  return new Promise((resolve, reject) => {
    // ... 从服务器获取组件
    resolve(/* 获取到的组件 */)
  })
})
// ... 像使用其他一般组件一样使用 `AsyncComp`
```

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

ES 模块动态导入也会返回一个 Promise，所以多数情况下我们会将它和 `defineAsyncComponent` 搭配使用。类似 Vite 和 Webpack 这样的构建工具也支持此语法 (并且会将它们作为打包时的代码分割点)，因此我们也可以用它来导入 Vue 单文件组件：

js

```js
import { defineAsyncComponent } from 'vue'
const AsyncComp = defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
)
```

最后得到的 `AsyncComp` 是一个外层包装过的组件，仅在页面需要它渲染时才会调用加载内部实际组件的函数。它会将接收到的 props 和插槽传给内部组件，所以你可以使用这个异步的包装组件无缝地替换原始组件，同时实现延迟加载。

与普通组件一样，异步组件可以使用 `app.component()` 全局注册：

js

```js
app.component('MyComponent', defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
))
```

也可以直接在父组件中直接定义它们：

html

```html
<script setup>
import { defineAsyncComponent } from 'vue'
const AdminPage = defineAsyncComponent(() =>
  import('./components/AdminPageComponent.vue')
)
```

```
</script>
<template>
  <AdminPage />
</template>
```

3.8.2 Loading and Error States

Asynchronous operations inevitably involve loading and error states - `defineAsyncComponent()` supports handling these states via advanced options:

```js
const AsyncComp = defineAsyncComponent({
  // 加载函数
  loader: () => import('./Foo.vue'),
  // 加载异步组件时使用的组件
  loadingComponent: LoadingComponent,
  // 展示加载组件前的延迟时间，默认为 200ms
  delay: 200,
  // 加载失败后展示的组件
  errorComponent: ErrorComponent,
  // 如果提供了一个 timeout 时间限制，并超时了
  // 也会显示这里配置的报错组件，默认值是：Infinity
  timeout: 3000
})
```

If a loading component is provided, it will be displayed first while the inner component is being loaded. There is a default 200ms delay before the loading component is shown - this is because on fast networks, an instant loading state may get replaced too fast and end up looking like a flicker.

If an error component is provided, it will be displayed when the Promise returned by the loader function is rejected. You can also specify a timeout to show the error component when the request is taking too long.

3.8.3 Using with Suspense

Async components can be used with the `<Suspense>` built-in component. The interaction between `<Suspense>` and async components is documented in the dedicated chapter for “.

```
</script>
<template>
  <AdminPage />
</template>
```

3.8.2 加载与错误状态

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

```js
const AsyncComp = defineAsyncComponent({
  // 加载函数
  loader: () => import('./Foo.vue'),
  // 加载异步组件时使用的组件
  loadingComponent: LoadingComponent,
  // 展示加载组件前的延迟时间，默认为 200ms
  delay: 200,
  // 加载失败后展示的组件
  errorComponent: ErrorComponent,
  // 如果提供了一个 timeout 时间限制，并超时了
  // 也会显示这里配置的报错组件，默认值是：Infinity
  timeout: 3000
})
```

如果提供了一个加载组件，它将在内部组件加载时先行显示。在加载组件显示之前有一个默认的 200ms 延迟——这是因为在网络状况较好时，加载完成得很快，加载组件和最终组件之间的替换太快可能产生闪烁，反而影响用户感受。

如果提供了一个报错组件，则它会在加载器函数返回的 Promise 抛错时被渲染。你还可以指定一个超时时间，在请求耗时超过指定时间时也会渲染报错组件。

3.8.3 搭配 Suspense 使用

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

第四章 Reusability

4.1 Composables

> **TIP**
> This section assumes basic knowledge of Composition API. If you have been learning Vue with Options API only, you can set the API Preference to Composition API (using the toggle at the top of the left sidebar) and re-read the Reactivity Fundamentals and Lifecycle Hooks chapters.

4.1.1 What is a ”Composable”?

In the context of Vue applications, a ”composable” is a function that leverages Vue’s Composition API to encapsulate and reuse **stateful logic**.

When building frontend applications, we often need to reuse logic for common tasks. For example, we may need to format dates in many places, so we extract a reusable function for that. This formatter function encapsulates **stateless logic**: it takes some input and immediately returns expected output. There are many libraries out there for reusing stateless logic - for example lodash and date-fns, which you may have heard of.

By contrast, stateful logic involves managing state that changes over time. A simple example would be tracking the current position of the mouse on a page. In real-world scenarios, it could also be more complex logic such as touch gestures or connection status to a database.

4.1.2 Mouse Tracker Example

逻辑复用

4.1 组合式函数

> **TIP**
> 此章节假设你已经对组合式 API 有了基本的了解。如果你只学习过选项式 API，你可以使用左侧边栏上方的切换按钮将 API 风格切换为组合式 API 后，重新阅读响应性基础和生命周期钩子两个章节。

4.1.1 什么是 “组合式函数”？

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

当构建前端应用时，我们常常需要复用公共任务的逻辑。例如为了在不同地方格式化时间，我们可能会抽取一个可复用的日期格式化函数。这个函数封装了**无状态的逻辑**：它在接收一些输入后立刻返回所期望的输出。复用无状态逻辑的库有很多，比如你可能已经用过的 lodash 或是 date-fns。

相比之下，有状态逻辑负责管理会随时间而变化的状态。一个简单的例子是跟踪当前鼠标在页面中的位置。在实际应用中，也可能是像触摸手势或与数据库的连接状态这样的更复杂的逻辑。

4.1.2 鼠标跟踪器示例

If we were to implement the mouse tracking functionality using the Composition API directly inside a component, it would look like this:

```html
<script setup>
import { ref, onMounted, onUnmounted } from 'vue'
const x = ref(0)
const y = ref(0)
function update(event) {
  x.value = event.pageX
  y.value = event.pageY
}
onMounted(() => window.addEventListener('mousemove', update))
onUnmounted(() => window.removeEventListener('mousemove', update))
</script>
<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

But what if we want to reuse the same logic in multiple components? We can extract the logic into an external file, as a composable function:

```js
// mouse.js
import { ref, onMounted, onUnmounted } from 'vue'
// 按照惯例，组合式函数名以“use”开头
export function useMouse() {
  // 被组合式函数封装和管理的状态
  const x = ref(0)
  const y = ref(0)
  // 组合式函数可以随时更改其状态。
  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }
  // 一个组合式函数也可以挂靠在所属组件的生命周期上
  // 来启动和卸载副作用
  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))
  // 通过返回值暴露所管理的状态
  return { x, y }
```

如果我们要直接在组件中使用组合式 API 实现鼠标跟踪功能，它会是这样的：

```html
<script setup>
import { ref, onMounted, onUnmounted } from 'vue'
const x = ref(0)
const y = ref(0)
function update(event) {
  x.value = event.pageX
  y.value = event.pageY
}
onMounted(() => window.addEventListener('mousemove', update))
onUnmounted(() => window.removeEventListener('mousemove', update))
</script>
<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

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

```js
// mouse.js
import { ref, onMounted, onUnmounted } from 'vue'
// 按照惯例，组合式函数名以“use”开头
export function useMouse() {
  // 被组合式函数封装和管理的状态
  const x = ref(0)
  const y = ref(0)
  // 组合式函数可以随时更改其状态。
  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }
  // 一个组合式函数也可以挂靠在所属组件的生命周期上
  // 来启动和卸载副作用
  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))
  // 通过返回值暴露所管理的状态
  return { x, y }
}
```

And this is how it can be used in components:

```html
<script setup>
import { useMouse } from './mouse.js'
const { x, y } = useMouse()
</script>
<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

Try it in the Playground

As we can see, the core logic remains identical - all we had to do was move it into an external function and return the state that should be exposed. Just like inside a component, you can use the full range of Composition API functions in composables. The same `useMouse()` functionality can now be used in any component.

The cooler part about composables though, is that you can also nest them: one composable function can call one or more other composable functions. This enables us to compose complex logic using small, isolated units, similar to how we compose an entire application using components. In fact, this is why we decided to call the collection of APIs that make this pattern possible Composition API.

For example, we can extract the logic of adding and removing a DOM event listener into its own composable:

```js
// event.js
import { onMounted, onUnmounted } from 'vue'
export function useEventListener(target, event, callback) {
  // 如果你想的话，
  // 也可以用字符串形式的 CSS 选择器来寻找目标 DOM 元素
  onMounted(() => target.addEventListener(event, callback))
  onUnmounted(() => target.removeEventListener(event, callback))
}
```

And now our `useMouse()` composable can be simplified to:

```js
// mouse.js
import { ref } from 'vue'
```

```
}
```

下面是它在组件中使用的方式：

```html
<script setup>
import { useMouse } from './mouse.js'
const { x, y } = useMouse()
</script>
<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

在演练场中尝试一下

如你所见，核心逻辑完全一致，我们做的只是把它移到一个外部函数中去，并返回需要暴露的状态。和在组件中一样，你也可以在组合式函数中使用所有的组合式 API。现在，`useMouse()` 的功能可以在任何组件中轻易复用了。

更酷的是，你还可以嵌套多个组合式函数：一个组合式函数可以调用一个或多个其他的组合式函数。这使得我们可以像使用多个组件组合成整个应用一样，用多个较小且逻辑独立的单元来组合形成复杂的逻辑。实际上，这正是为什么我们决定将实现了这一设计模式的 API 集合命名为组合式 API。

举例来说，我们可以将添加和清除 DOM 事件监听器的逻辑也封装进一个组合式函数中：

```js
// event.js
import { onMounted, onUnmounted } from 'vue'
export function useEventListener(target, event, callback) {
  // 如果你想的话，
  // 也可以用字符串形式的 CSS 选择器来寻找目标 DOM 元素
  onMounted(() => target.addEventListener(event, callback))
  onUnmounted(() => target.removeEventListener(event, callback))
}
```

有了它，之前的 `useMouse()` 组合式函数可以被简化为：

```js
// mouse.js
import { ref } from 'vue'
import { useEventListener } from './event'
export function useMouse() {
  const x = ref(0)
  const y = ref(0)
  useEventListener(window, 'mousemove', (event) => {
    x.value = event.pageX
    y.value = event.pageY
  })
  return { x, y }
}
```

TIP

Each component instance calling `useMouse()` will create its own copies of `x` and `y` state so they won't interfere with one another. If you want to manage shared state between components, read the State Management chapter.

4.1.3 Async State Example

The `useMouse()` composable doesn't take any arguments, so let's take a look at another example that makes use of one. When doing async data fetching, we often need to handle different states: loading, success, and error:

html

```html
<script setup>
import { ref } from 'vue'
const data = ref(null)
const error = ref(null)
fetch('...')
  .then((res) => res.json())
  .then((json) => (data.value = json))
  .catch((err) => (error.value = err))
</script>
<template>
  <div v-if="error">Oops! Error encountered: {{ error.message }}</div>
  <div v-else-if="data">
    Data loaded:
```

```
import { useEventListener } from './event'
export function useMouse() {
  const x = ref(0)
  const y = ref(0)
  useEventListener(window, 'mousemove', (event) => {
    x.value = event.pageX
    y.value = event.pageY
  })
  return { x, y }
}
```

TIP

每一个调用 `useMouse()` 的组件实例会创建其独有的 `x`、`y` 状态拷贝，因此他们不会互相影响。如果你想要在组件之间共享状态，请阅读状态管理这一章。

4.1.3 异步状态示例

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

html

```html
<script setup>
import { ref } from 'vue'
const data = ref(null)
const error = ref(null)
fetch('...')
  .then((res) => res.json())
  .then((json) => (data.value = json))
  .catch((err) => (error.value = err))
</script>
<template>
  <div v-if="error">Oops! Error encountered: {{ error.message }}</div>
  <div v-else-if="data">
    Data loaded:
```

```
    <pre>{{ data }}</pre>
  </div>
  <div v-else>Loading...</div>
</template>
```

It would be tedious to have to repeat this pattern in every component that needs to fetch data. Let's extract it into a composable:

js

```
// fetch.js
import { ref } from 'vue'
export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)
  fetch(url)
    .then((res) => res.json())
    .then((json) => (data.value = json))
    .catch((err) => (error.value = err))
  return { data, error }
}
```

Now in our component we can just do:

html

```
<script setup>
import { useFetch } from './fetch.js'
const { data, error } = useFetch('...')
</script>
```

Accepting Reactive State

`useFetch()` takes a static URL string as input - so it performs the fetch only once and is then done. What if we want it to re-fetch whenever the URL changes? In order to achieve this, we need to pass reactive state into the composable function, and let the composable create watchers that perform actions using the passed state.

For example, `useFetch()` should be able to accept a ref:

js

```
const url = ref('/initial-url')
```

```
    <pre>{{ data }}</pre>
  </div>
  <div v-else>Loading...</div>
</template>
```

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

js

```
// fetch.js
import { ref } from 'vue'
export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)
  fetch(url)
    .then((res) => res.json())
    .then((json) => (data.value = json))
    .catch((err) => (error.value = err))
  return { data, error }
}
```

现在我们在组件里只需要：

html

```
<script setup>
import { useFetch } from './fetch.js'
const { data, error } = useFetch('...')
</script>
```

接收响应式状态

`useFetch()` 接收一个静态 URL 字符串作为输入——因此它只会执行一次 fetch 并且就此结束。如果我们想要在 URL 改变时重新 fetch 呢？为了实现这一点，我们需要将响应式状态传入组合式函数，并让它基于传入的状态来创建执行操作的侦听器。

举例来说，`useFetch()` 应该能够接收一个 ref：

js

```
const url = ref('/initial-url')
```

```js
const { data, error } = useFetch(url)
// 这将会重新触发 fetch
url.value = '/new-url'
```

Or, accept a getter function:

```js
// 当 props.id 改变时重新 fetch
const { data, error } = useFetch(() => `/posts/${props.id}`)
```

We can refactor our existing implementation with the `watchEffect()` and `toValue()` APIs:

```js
// fetch.js
import { ref, watchEffect, toValue } from 'vue'
export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)
  const fetchData = () => {
    // reset state before fetching..
    data.value = null
    error.value = null
    fetch(toValue(url))
      .then((res) => res.json())
      .then((json) => (data.value = json))
      .catch((err) => (error.value = err))
  }
  watchEffect(() => {
    fetchData()
  })
  return { data, error }
}
```

`toValue()` is an API added in 3.3. It is designed to normalize refs or getters into values. If the argument is a ref, it returns the ref's value; if the argument is a function, it will call the function and return its return value. Otherwise, it returns the argument as-is. It works similarly to `unref()`, but with special treatment for functions.

Notice that `toValue(url)` is called **inside** the `watchEffect` callback. This ensures that any reactive

```js
const { data, error } = useFetch(url)
// 这将会重新触发 fetch
url.value = '/new-url'
```

或者接收一个 getter 函数：

```js
// 当 props.id 改变时重新 fetch
const { data, error } = useFetch(() => `/posts/${props.id}`)
```

我们可以用 `watchEffect()` 和 `toValue()` API 来重构我们现有的实现：

```js
// fetch.js
import { ref, watchEffect, toValue } from 'vue'
export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)
  const fetchData = () => {
    // reset state before fetching..
    data.value = null
    error.value = null
    fetch(toValue(url))
      .then((res) => res.json())
      .then((json) => (data.value = json))
      .catch((err) => (error.value = err))
  }
  watchEffect(() => {
    fetchData()
  })
  return { data, error }
}
```

`toValue()` 是一个在 3.3 版本中新增的 API。它的设计目的是将 ref 或 getter 规范化为值。如果参数是 ref，它会返回 ref 的值；如果参数是函数，它会调用函数并返回其返回值。否则，它会原样返回参数。它的工作方式类似于 `unref()`，但对函数有特殊处理。

注意 `toValue(url)` 是在 `watchEffect` 回调函数的**内部**调用的。这确保了在 `toValue()`

dependencies accessed during the `toValue()` normalization are tracked by the watcher.

This version of `useFetch()` now accepts static URL strings, refs, and getters, making it much more flexible. The watch effect will run immediately, and will track any dependencies accessed during `toValue(url)`. If no dependencies are tracked (e.g. url is already a string), the effect runs only once; otherwise, it will re-run whenever a tracked dependency changes.

Here's the updated version of `useFetch()`, with an artificial delay and randomized error for demo purposes.

4.1.4 Conventions and Best Practices

Naming

It is a convention to name composable functions with camelCase names that start with ”use”.

Input Arguments

A composable can accept ref or getter arguments even if it doesn't rely on them for reactivity. If you are writing a composable that may be used by other developers, it's a good idea to handle the case of input arguments being refs or getters instead of raw values. The `toValue()` utility function will come in handy for this purpose:

js

```js
import { toValue } from 'vue'
function useFeature(maybeRefOrGetter) {
  // 如果 maybeRefOrGetter 是一个 ref 或 getter,
  // 将返回它的规范化值。
  // 否则原样返回。
  const value = toValue(maybeRefOrGetter)
}
```

If your composable creates reactive effects when the input is a ref or a getter, make sure to either explicitly watch the ref / getter with `watch()`, or call `toValue()` inside a `watchEffect()` so that it is properly tracked.

The useFetch() implementation discussed earlier provides a concrete example of a composable that accepts refs, getters and plain values as input argument.

规范化期间访问的任何响应式依赖项都会被侦听器跟踪。

这个版本的 `useFetch()` 现在能接收静态 URL 字符串、ref 和 getter，使其更加灵活。watch effect 会立即运行，并且会跟踪 `toValue(url)` 期间访问的任何依赖项。如果没有跟踪到依赖项（例如 url 已经是字符串），则 effect 只会运行一次；否则，它将在跟踪到的任何依赖项更改时重新运行。

这是更新后的 `useFetch()`，为了便于演示，添加了人为延迟和随机错误。

4.1.4 约定和最佳实践

命名

组合式函数约定用驼峰命名法命名，并以 “use” 作为开头。

输入参数

即便不依赖于 ref 或 getter 的响应性，组合式函数也可以接收它们作为参数。如果你正在编写一个可能被其他开发者使用的组合式函数，最好处理一下输入参数是 ref 或 getter 而非原始值的情况。可以利用 `toValue()` 工具函数来实现：

js

```js
import { toValue } from 'vue'
function useFeature(maybeRefOrGetter) {
  // 如果 maybeRefOrGetter 是一个 ref 或 getter,
  // 将返回它的规范化值。
  // 否则原样返回。
  const value = toValue(maybeRefOrGetter)
}
```

如果你的组合式函数在输入参数是 ref 或 getter 的情况下创建了响应式 effect，为了让它能够被正确追踪，请确保要么使用 `watch()` 显式地监视 ref 或 getter，要么在 `watchEffect()` 中调用 `toValue()`。

前面讨论过的 useFetch() 实现提供了一个接受 ref、getter 或普通值作为输入参数的组合式函数的具体示例。

Return Values

You have probably noticed that we have been exclusively using `ref()` instead of `reactive()` in composables. The recommended convention is for composables to always return a plain, non-reactive object containing multiple refs. This allows it to be destructured in components while retaining reactivity:

```js
// x 和 y 是两个 ref
const { x, y } = useMouse()
```

Returning a reactive object from a composable will cause such destructures to lose the reactivity connection to the state inside the composable, while the refs will retain that connection.

If you prefer to use returned state from composables as object properties, you can wrap the returned object with `reactive()` so that the refs are unwrapped. For example:

```js
const mouse = reactive(useMouse())
// mouse.x 链接到了原来的 x ref
console.log(mouse.x)
```

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

Side Effects

It is OK to perform side effects (e.g. adding DOM event listeners or fetching data) in composables, but pay attention to the following rules:

- If you are working on an application that uses Server-Side Rendering (SSR), make sure to perform DOM-specific side effects in post-mount lifecycle hooks, e.g. `onMounted()`. These hooks are only called in the browser, so you can be sure that code inside them has access to the DOM.
- Remember to clean up side effects in `onUnmounted()`. For example, if a composable sets up a DOM event listener, it should remove that listener in `onUnmounted()` as we have seen in the `useMouse()` example. It can be a good idea to use a composable that automatically does this for you, like the `useEventListener()` example.

返回值

你可能已经注意到了，我们一直在组合式函数中使用 `ref()` 而不是 `reactive()`。我们推荐的约定是组合式函数始终返回一个包含多个 ref 的普通的非响应式对象，这样该对象在组件中被解构为 ref 之后仍可以保持响应性：

```js
// x 和 y 是两个 ref
const { x, y } = useMouse()
```

从组合式函数返回一个响应式对象会导致在对象解构过程中丢失与组合式函数内状态的响应性连接。与之相反，ref 则可以维持这一响应性连接。

如果你更希望以对象属性的形式来使用组合式函数中返回的状态，你可以将返回的对象用 `reactive()` 包装一次，这样其中的 ref 会被自动解包，例如：

```js
const mouse = reactive(useMouse())
// mouse.x 链接到了原来的 x ref
console.log(mouse.x)
```

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

副作用

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

- 如果你的应用用到了服务端渲染 (SSR)，请确保在组件挂载后才调用的生命周期钩子中执行 DOM 相关的副作用，例如：`onMounted()`。这些钩子仅会在浏览器中被调用，因此可以确保能访问到 DOM。
- 确保在 `onUnmounted()` 时清理副作用。举例来说，如果一个组合式函数设置了一个事件监听器，它就应该在 `onUnmounted()` 中被移除 (就像我们在 `useMouse()` 示例中看到的一样)。当然也可以像之前的 `useEventListener()` 示例那样，使用一个组合式函数来自动帮你做这些事。

Usage Restrictions

Composables should only be called in `<script setup>` or the `setup()` hook. They should also be called **synchronously** in these contexts. In some cases, you can also call them in lifecycle hooks like `onMounted()`.

These restrictions are important because these are the contexts where Vue is able to determine the current active component instance. Access to an active component instance is necessary so that:

1. Lifecycle hooks can be registered to it.
2. Computed properties and watchers can be linked to it, so that they can be disposed when the instance is unmounted to prevent memory leaks.

> **TIP**
> <script setup> is the only place where you can call composables after using await. The compiler automatically restores the active instance context for you after the async operation.

4.1.5 Extracting Composables for Code Organization

Composables can be extracted not only for reuse, but also for code organization. As the complexity of your components grow, you may end up with components that are too large to navigate and reason about. Composition API gives you the full flexibility to organize your component code into smaller functions based on logical concerns:

html

```html
<script setup>
import { useFeatureA } from './featureA.js'
import { useFeatureB } from './featureB.js'
import { useFeatureC } from './featureC.js'
const { foo, bar } = useFeatureA()
const { baz } = useFeatureB(foo)
const { qux } = useFeatureC(baz)
</script>
```

To some extent, you can think of these extracted composables as component-scoped services that can talk to one another.

使用限制

组合式函数只能在 `<script setup>` 或 `setup()` 钩子中被调用。在这些上下文中，它们也只能被**同步**调用。在某些情况下，你也可以在像 `onMounted()` 这样的生命周期钩子中调用它们。

这些限制很重要，因为这些是 Vue 用于确定当前活跃的组件实例的上下文。访问活跃的组件实例很有必要，这样才能：

1. 将生命周期钩子注册到该组件实例上
2. 将计算属性和监听器注册到该组件实例上，以便在该组件被卸载时停止监听，避免内存泄漏。

> **TIP**
> <script setup> 是唯一在调用 await 之后仍可调用组合式函数的地方。编译器会在异步操作之后自动为你恢复当前的组件实例。

4.1.5 通过抽取组合式函数改善代码结构

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

html

```html
<script setup>
import { useFeatureA } from './featureA.js'
import { useFeatureB } from './featureB.js'
import { useFeatureC } from './featureC.js'
const { foo, bar } = useFeatureA()
const { baz } = useFeatureB(foo)
const { qux } = useFeatureC(baz)
</script>
```

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

4.1.6 Using Composables in Options API

If you are using Options API, composables must be called inside `setup()`, and the returned bindings must be returned from `setup()` so that they are exposed to `this` and the template:

```js
import { useMouse } from './mouse.js'
import { useFetch } from './fetch.js'
export default {
  setup() {
    const { x, y } = useMouse()
    const { data, error } = useFetch('...')
    return { x, y, data, error }
  },
  mounted() {
    // setup() 暴露的属性可以在通过 `this` 访问到
    console.log(this.x)
  }
  // ... 其他选项
}
```

4.1.7 Comparisons with Other Techniques

vs. Mixins

Users coming from Vue 2 may be familiar with the mixins option, which also allows us to extract component logic into reusable units. There are three primary drawbacks to mixins:

1. **Unclear source of properties**: when using many mixins, it becomes unclear which instance property is injected by which mixin, making it difficult to trace the implementation and understand the component's behavior. This is also why we recommend using the refs + destructure pattern for composables: it makes the property source clear in consuming components.
2. **Namespace collisions**: multiple mixins from different authors can potentially register the same property keys, causing namespace collisions. With composables, you can rename the destructured variables if there are conflicting keys from different composables.
3. **Implicit cross-mixin communication**: multiple mixins that need to interact with one

4.1.6 在选项式 API 中使用组合式函数

如果你正在使用选项式 API，组合式函数必须在 `setup()` 中调用。且其返回的绑定必须在 `setup()` 中返回，以便暴露给 `this` 及其模板：

```js
import { useMouse } from './mouse.js'
import { useFetch } from './fetch.js'
export default {
  setup() {
    const { x, y } = useMouse()
    const { data, error } = useFetch('...')
    return { x, y, data, error }
  },
  mounted() {
    // setup() 暴露的属性可以在通过 `this` 访问到
    console.log(this.x)
  }
  // ... 其他选项
}
```

4.1.7 与其他模式的比较

和 Mixin 的对比

Vue 2 的用户可能会对 mixins 选项比较熟悉。它也让我们能够把组件逻辑提取到可复用的单元里。然而 mixins 有三个主要的短板：

1. **不清晰的数据来源**：当使用了多个 mixin 时，实例上的数据属性来自哪个 mixin 变得不清晰，这使追溯实现和理解组件行为变得困难。这也是我们推荐在组合式函数中使用 ref + 解构模式的理由：让属性的来源在消费组件时一目了然。
2. **命名空间冲突**：多个来自不同作者的 mixin 可能会注册相同的属性名，造成命名冲突。若使用组合式函数，你可以通过在解构变量时对变量进行重命名来避免相同的键名。
3. **隐式的跨 mixin 交流**：多个 mixin 需要依赖共享的属性名来进行相互作用，

another have to rely on shared property keys, making them implicitly coupled. With composables, values returned from one composable can be passed into another as arguments, just like normal functions.

For the above reasons, we no longer recommend using mixins in Vue 3. The feature is kept only for migration and familiarity reasons.

vs. Renderless Components

In the component slots chapter, we discussed the Renderless Component pattern based on scoped slots. We even implemented the same mouse tracking demo using renderless components.

The main advantage of composables over renderless components is that composables do not incur the extra component instance overhead. When used across an entire application, the amount of extra component instances created by the renderless component pattern can become a noticeable performance overhead.

The recommendation is to use composables when reusing pure logic, and use components when reusing both logic and visual layout.

vs. React Hooks

If you have experience with React, you may notice that this looks very similar to custom React hooks. Composition API was in part inspired by React hooks, and Vue composables are indeed similar to React hooks in terms of logic composition capabilities. However, Vue composables are based on Vue's fine-grained reactivity system, which is fundamentally different from React hooks' execution model. This is discussed in more detail in the Composition API FAQ.

4.1.8 Further Reading

- Reactivity In Depth: for a low-level understanding of how Vue's reactivity system works.
- State Management: for patterns of managing state shared by multiple components.
- Testing Composables: tips on unit testing composables.
- VueUse: an ever-growing collection of Vue composables. The source code is also a great learning resource.

这使得它们隐性地耦合在一起。而一个组合式函数的返回值可以作为另一个组合式函数的参数被传入，像普通函数那样。

基于上述理由，我们不再推荐在 Vue 3 中继续使用 mixin。保留该功能只是为了项目迁移的需求和照顾熟悉它的用户。

和无渲染组件的对比

在组件插槽一章中，我们讨论过了基于作用域插槽的无渲染组件。我们甚至用它实现了一样的鼠标追踪器示例。

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

我们推荐在纯逻辑复用时使用组合式函数，在需要同时复用逻辑和视图布局时使用无渲染组件。

和 React Hooks 的对比

如果你有 React 的开发经验，你可能注意到组合式函数和自定义 React hooks 非常相似。组合式 API 的一部分灵感正来自于 React hooks，Vue 的组合式函数也的确在逻辑组合能力上与 React hooks 相近。然而，Vue 的组合式函数是基于 Vue 细粒度的响应性系统，这和 React hooks 的执行模型有本质上的不同。这一话题在组合式 API 的常见问题中有更细致的讨论。

4.1.8 延伸阅读

- 深入响应性原理：理解 Vue 响应性系统的底层细节。
- 状态管理：多个组件间共享状态的管理模式。
- 测试组合式函数：组合式函数的单元测试技巧。
- VueUse：一个日益增长的 Vue 组合式函数集合。源代码本身就是一份不错的学习资料。

4.2 Custom Directives

In addition to the default set of directives shipped in core (like `v-model` or `v-show`), Vue also allows you to register your own custom directives.

4.2.1 Introduction

We have introduced two forms of code reuse in Vue: components and composables. Components are the main building blocks, while composables are focused on reusing stateful logic. Custom directives, on the other hand, are mainly intended for reusing logic that involves low-level DOM access on plain elements.

A custom directive is defined as an object containing lifecycle hooks similar to those of a component. The hooks receive the element the directive is bound to. Here is an example of a directive that focuses an input when the element is inserted into the DOM by Vue:

```html
<script setup>
// 在模板中启用 v-focus
const vFocus = {
  mounted: (el) => el.focus()
}
</script>
<template>
  <input v-focus />
</template>
```

Assuming you haven't clicked elsewhere on the page, the input above should be auto-focused. This directive is more useful than the `autofocus` attribute because it works not just on page load - it also works when the element is dynamically inserted by Vue.

In `<script setup>`, any camelCase variable that starts with the `v` prefix can be used as a custom directive. In the example above, `vFocus` can be used in the template as `v-focus`.

If not using `<script setup>`, custom directives can be registered using the `directives` option:

```js
export default {
  setup() {
```

4.2 自定义指令

4.2.1 介绍

除了 Vue 内置的一系列指令 (比如 `v-model` 或 `v-show`) 之外，Vue 还允许你注册自定义的指令 (Custom Directives)。

我们已经介绍了两种在 Vue 中重用代码的方式：组件和组合式函数。组件是主要的构建模块，而组合式函数则侧重于有状态的逻辑。另一方面，自定义指令主要是为了重用涉及普通元素的底层 DOM 访问的逻辑。

一个自定义指令由一个包含类似组件生命周期钩子的对象来定义。钩子函数会接收到指令所绑定元素作为其参数。下面是一个自定义指令的例子，当一个 input 元素被 Vue 插入到 DOM 中后，它会被自动聚焦：

```html
<script setup>
// 在模板中启用 v-focus
const vFocus = {
  mounted: (el) => el.focus()
}
</script>
<template>
  <input v-focus />
</template>
```

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

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

在没有使用 `<script setup>` 的情况下，自定义指令需要通过 `directives` 选项注册：

```js
export default {
  setup() {
    /*...*/
  },
  directives: {
    // 在模板中启用 v-focus
    focus: {
      /* ... */
    }
  }
}
```

It is also common to globally register custom directives at the app level:

js
```
const app = createApp({})
// 使 v-focus 在所有组件中都可用
app.directive('focus', {
  /* ... */
})
```

TIP
Custom directives should only be used when the desired functionality can only be achieved via direct DOM manipulation. Prefer declarative templating using built-in directives such as `v-bind` when possible because they are more efficient and server-rendering friendly.

4.2.2 Directive Hooks

A directive definition object can provide several hook functions (all optional):

js
```
const myDirective = {
  // 在绑定元素的 attribute 前
  // 或事件监听器应用前调用
  created(el, binding, vnode, prevVnode) {
    // 下面会介绍各个参数的细节
  },
  // 在元素被插入到 DOM 前调用
  beforeMount(el, binding, vnode, prevVnode) {},
```

```
    /*...*/
  },
  directives: {
    // 在模板中启用 v-focus
    focus: {
      /* ... */
    }
  }
}
```

将一个自定义指令全局注册到应用层级也是一种常见的做法：

js

```
const app = createApp({})
// 使 v-focus 在所有组件中都可用
app.directive('focus', {
  /* ... */
})
```

TIP
只有当所需功能只能通过直接的 DOM 操作来实现时，才应该使用自定义指令。其他情况下应该尽可能地使用 `v-bind` 这样的内置指令来声明式地使用模板，这样更高效，也对服务端渲染更友好。

4.2.2 指令钩子

一个指令的定义对象可以提供几种钩子函数 (都是可选的)：

js

```
const myDirective = {
  // 在绑定元素的 attribute 前
  // 或事件监听器应用前调用
  created(el, binding, vnode, prevVnode) {
    // 下面会介绍各个参数的细节
  },
  // 在元素被插入到 DOM 前调用
  beforeMount(el, binding, vnode, prevVnode) {},
  // 在绑定元素的父组件
  // 及他自己的所有子节点都挂载完成后调用
  mounted(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件更新前调用
  beforeUpdate(el, binding, vnode, prevVnode) {},
  // 在绑定元素的父组件
  // 及他自己的所有子节点都更新后调用
  updated(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件卸载前调用
  beforeUnmount(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件卸载后调用
  unmounted(el, binding, vnode, prevVnode) {}
}
```

Hook Arguments

Directive hooks are passed these arguments:

- `el`: the element the directive is bound to. This can be used to directly manipulate the DOM.
- `binding`: an object containing the following properties.
  - `value`: The value passed to the directive. For example in `v-my-directive="1 + 1"`, the value would be `2`.
  - `oldValue`: The previous value, only available in `beforeUpdate` and `updated`. It is available whether or not the value has changed.
  - `arg`: The argument passed to the directive, if any. For example in `v-my-directive:foo`, the arg would be `"foo"`.
  - `modifiers`: An object containing modifiers, if any. For example in `v-my-directive.foo.bar`, the modifiers object would be `{ foo: true, bar: true }`.
  - `instance`: The instance of the component where the directive is used.
  - `dir`: the directive definition object.
- `vnode`: the underlying VNode representing the bound element.

```
  // 在绑定元素的父组件
  // 及他自己的所有子节点都挂载完成后调用
  mounted(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件更新前调用
  beforeUpdate(el, binding, vnode, prevVnode) {},
  // 在绑定元素的父组件
  // 及他自己的所有子节点都更新后调用
  updated(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件卸载前调用
  beforeUnmount(el, binding, vnode, prevVnode) {},
  // 绑定元素的父组件卸载后调用
  unmounted(el, binding, vnode, prevVnode) {}
}
```

钩子参数

指令的钩子会传递以下几种参数：

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

- `prevNode`: the VNode representing the bound element from the previous render. Only available in the `beforeUpdate` and `updated` hooks.

As an example, consider the following directive usage:

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

The `binding` argument would be an object in the shape of:

```js
{
  arg: 'foo',
  modifiers: { bar: true },
  value: /* `baz` 的值 */,
  oldValue: /* 上一次更新时 `baz` 的值 */
}
```

Similar to built-in directives, custom directive arguments can be dynamic. For example:

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

Here the directive argument will be reactively updated based on `arg` property in our component state.

> **Note**
> Apart from `el`, you should treat these arguments as read-only and never modify them. If you need to share information across hooks, it is recommended to do so through element's dataset.

4.2.3 Function Shorthand

It's common for a custom directive to have the same behavior for `mounted` and `updated`, with no need for the other hooks. In such cases we can define the directive as a function:

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

```js
app.directive('color', (el, binding) => {
  // 这会在 `mounted` 和 `updated` 时都调用
```

- `prevNode`:代表之前的渲染中指令所绑定元素的 VNode。仅在 `beforeUpdate` 和 `updated` 钩子中可用。

举例来说，像下面这样使用指令：

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

`binding` 参数会是一个这样的对象：

```js
{
  arg: 'foo',
  modifiers: { bar: true },
  value: /* `baz` 的值 */,
  oldValue: /* 上一次更新时 `baz` 的值 */
}
```

和内置指令类似，自定义指令的参数也可以是动态的。举例来说：

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

这里指令的参数会基于组件的 `arg` 数据属性响应式地更新。

> **Note**
> 除了 `el` 外，其他参数都是只读的，不要更改它们。若你需要在不同的钩子间共享信息，推荐通过元素的 dataset attribute 实现。

4.2.3 简化形式

对于自定义指令来说，一个很常见的情况是仅仅需要在 `mounted` 和 `updated` 上实现相同的行为，除此之外并不需要其他钩子。这种情况下我们可以直接用一个函数来定义指令，如下所示：

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

```js
app.directive('color', (el, binding) => {
  // 这会在 `mounted` 和 `updated` 时都调用
  el.style.color = binding.value
})
```

4.2.4 Object Literals

If your directive needs multiple values, you can also pass in a JavaScript object literal. Remember, directives can take any valid JavaScript expression.

html

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

js

```js
app.directive('demo', (el, binding) => {
  console.log(binding.value.color) // => "white"
  console.log(binding.value.text) // => "hello!"
})
```

4.2.5 Usage on Components

When used on components, custom directives will always apply to a component's root node, similar to Fallthrough Attributes.

html

```html
<MyComponent v-demo="test" />
```

html

```html
<!-- MyComponent 的模板 -->
<div> <!-- v-demo 指令会被应用在此处 -->
  <span>My component content</span>
</div>
```

Note that components can potentially have more than one root node. When applied to a multi-root component, a directive will be ignored and a warning will be thrown. Unlike attributes, directives can't be passed to a different element with `v-bind="$attrs"`. In general, it is **not** recommended to use custom directives on components.

4.3 Plugins

```
  el.style.color = binding.value
})
```

4.2.4 对象字面量

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

html

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

js

```js
app.directive('demo', (el, binding) => {
  console.log(binding.value.color) // => "white"
  console.log(binding.value.text) // => "hello!"
})
```

4.2.5 在组件上使用

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

html

```html
<MyComponent v-demo="test" />
```

html

```html
<!-- MyComponent 的模板 -->
<div> <!-- v-demo 指令会被应用在此处 -->
  <span>My component content</span>
</div>
```

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

4.3 插件

4.3.1 Introduction

Plugins are self-contained code that usually add app-level functionality to Vue. This is how we install a plugin:

```js
import { createApp } from 'vue'
const app = createApp({})
app.use(myPlugin, {
  /* 可选的选项 */
})
```

A plugin is defined as either an object that exposes an `install()` method, or simply a function that acts as the install function itself. The install function receives the app instance along with additional options passed to `app.use()`, if any:

```js
const myPlugin = {
  install(app, options) {
    // 配置此应用
  }
}
```

There is no strictly defined scope for a plugin, but common scenarios where plugins are useful include:

1. Register one or more global components or custom directives with `app.component()` and `app.directive()`.
2. Make a resource injectable throughout the app by calling `app.provide()`.
3. Add some global instance properties or methods by attaching them to `app.config.globalProperties`.
4. A library that needs to perform some combination of the above (e.g. vue-router).

4.3.2 Writing a Plugin

In order to better understand how to create your own Vue.js plugins, we will create a very simplified version of a plugin that displays `i18n` (short for Internationalization) strings.

Let's begin by setting up the plugin object. It is recommended to create it in a separate file and

4.3.1 介绍

插件 (Plugins) 是一种能为 Vue 添加全局功能的工具代码。下面是如何安装一个插件的示例：

```js
import { createApp } from 'vue'
const app = createApp({})
app.use(myPlugin, {
  /* 可选的选项 */
})
```

一个插件可以是一个拥有 `install()` 方法的对象，也可以直接是一个安装函数本身。安装函数会接收到安装它的应用实例和传递给 `app.use()` 的额外选项作为参数：

```js
const myPlugin = {
  install(app, options) {
    // 配置此应用
  }
}
```

插件没有严格定义的使用范围，但是插件发挥作用的常见场景主要包括以下几种：

1. 通过 `app.component()` 和 `app.directive()` 注册一到多个全局组件或自定义指令。
2. 通过 `app.provide()` 使一个资源可被注入进整个应用。
3. 向 `app.config.globalProperties` 中添加一些全局实例属性或方法
4. 一个可能上述三种都包含了的功能库 (例如 vue-router)。

4.3.2 编写一个插件

为了更好地理解如何构建 Vue.js 插件，我们可以试着写一个简单的 `i18n` (国际化 (Internationalization) 的缩写) 插件。

让我们从设置插件对象开始。建议在一个单独的文件中创建并导出它，以保证更

export it, as shown below to keep the logic contained and separate.

```js
// plugins/i18n.js
export default {
  install: (app, options) => {
    // 在这里编写插件代码
  }
}
```

We want to create a translation function. This function will receive a dot-delimited `key` string, which we will use to look up the translated string in the user-provided options. This is the intended usage in templates:

```html
<h1>{{ $translate('greetings.hello') }}</h1>
```

Since this function should be globally available in all templates, we will make it so by attaching it to `app.config.globalProperties` in our plugin:

```js
// plugins/i18n.js
export default {
  install: (app, options) => {
    // 注入一个全局可用的 $translate() 方法
    app.config.globalProperties.$translate = (key) => {
      // 获取 `options` 对象的深层属性
      // 使用 `key` 作为索引
      return key.split('.').reduce((o, i) => {
        if (o) return o[i]
      }, options)
    }
  }
}
```

Our `$translate` function will take a string such as `greetings.hello`, look inside the user provided configuration and return the translated value.

The object containing the translated keys should be passed to the plugin during installation via additional parameters to `app.use()`:

好地管理逻辑，如下所示：

```js
// plugins/i18n.js
export default {
  install: (app, options) => {
    // 在这里编写插件代码
  }
}
```

我们希望有一个翻译函数，这个函数接收一个以 . 作为分隔符的 `key` 字符串，用来在用户提供的翻译字典中查找对应语言的文本。期望的使用方式如下：

```html
<h1>{{ $translate('greetings.hello') }}</h1>
```

这个函数应当能够在任意模板中被全局调用。这一点可以通过在插件中将它添加到 `app.config.globalProperties` 上来实现：

```js
// plugins/i18n.js
export default {
  install: (app, options) => {
    // 注入一个全局可用的 $translate() 方法
    app.config.globalProperties.$translate = (key) => {
      // 获取 `options` 对象的深层属性
      // 使用 `key` 作为索引
      return key.split('.').reduce((o, i) => {
        if (o) return o[i]
      }, options)
    }
  }
}
```

我们的 `$translate` 函数会接收一个例如 `greetings.hello` 的字符串，在用户提供的翻译字典中查找，并返回翻译得到的值。

用于查找的翻译字典对象则应当在插件被安装时作为 `app.use()` 的额外参数被传入：

```js
import i18nPlugin from './plugins/i18n'
app.use(i18nPlugin, {
  greetings: {
    hello: 'Bonjour!'
  }
})
```

Now, our initial expression `$translate('greetings.hello')` will be replaced by `Bonjour!` at runtime.

See also: Augmenting Global Properties

> **TIP**
> Use global properties scarcely, since it can quickly become confusing if too many global properties injected by different plugins are used throughout an app.

Provide / Inject with Plugins

Plugins also allow us to use `inject` to provide a function or attribute to the plugin's users. For example, we can allow the application to have access to the `options` parameter to be able to use the translations object.

```js
// plugins/i18n.js
export default {
  install: (app, options) => {
    app.provide('i18n', options)
  }
}
```

Plugin users will now be able to inject the plugin options into their components using the `i18n` key:

```html
<script setup>
import { inject } from 'vue'
const i18n = inject('i18n')
console.log(i18n.greetings.hello)
```

```js
import i18nPlugin from './plugins/i18n'
app.use(i18nPlugin, {
  greetings: {
    hello: 'Bonjour!'
  }
})
```

这样，我们一开始的表达式 `$translate('greetings.hello')` 就会在运行时被替换为 `Bonjour!` 了。

TypeScript 用户请参考：扩展全局属性

> **TIP**
> 请谨慎使用全局属性，如果在整个应用中使用不同插件注入的太多全局属性，很容易让应用变得难以理解和维护。

插件中的 Provide / Inject

在插件中，我们可以通过 `provide` 来为插件用户供给一些内容。举例来说，我们可以将插件接收到的 `options` 参数提供给整个应用，让任何组件都能使用这个翻译字典对象。

```js
// plugins/i18n.js
export default {
  install: (app, options) => {
    app.provide('i18n', options)
  }
}
```

现在，插件用户就可以在他们的组件中以 `i18n` 为 key 注入并访问插件的选项对象了。

```html
<script setup>
import { inject } from 'vue'
const i18n = inject('i18n')
console.log(i18n.greetings.hello)
```

```
</script>
```

```
</script>
```

第五章 Built-in Components

5.1 Transition

Vue offers two built-in components that can help work with transitions and animations in response to changing state:

- `<Transition>` for applying animations when an element or component is entering and leaving the DOM. This is covered on this page.
- `<TransitionGroup>` for applying animations when an element or component is inserted into, removed from, or moved within a `v-for` list. This is covered in the next chapter.

Aside from these two components, we can also apply animations in Vue using other techniques such as toggling CSS classes or state-driven animations via style bindings. These additional techniques are covered in the Animation Techniques chapter.

5.1.1 The <Transition> Component

`<Transition>` is a built-in component: this means it is available in any component's template without having to register it. It can be used to apply enter and leave animations on elements or components passed to it via its default slot. The enter or leave can be triggered by one of the following:

- Conditional rendering via `v-if`
- Conditional display via `v-show`
- Dynamic components toggling via the `<component>` special element
- Changing the special `key` attribute

内置组件

5.1 Transition

Vue 提供了两个内置组件，可以帮助你制作基于状态变化的过渡和动画：

- `<Transition>` 会在一个元素或组件进入和离开 DOM 时应用动画。本章节会介绍如何使用它。
- `<TransitionGroup>` 会在一个 `v-for` 列表中的元素或组件被插入，移动，或移除时应用动画。我们将在下一章节中介绍。

除了这两个组件，我们也可以通过其他技术手段来应用动画，比如切换 CSS class 或用状态绑定样式来驱动动画。这些其他的方法会在动画技巧章节中展开。

5.1.1 <Transition> 组件

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

- 由 `v-if` 所触发的切换
- 由 `v-show` 所触发的切换
- 由特殊元素 `<component>` 切换的动态组件
- 改变特殊的 `key` 属性

This is an example of the most basic usage:

```html
<button @click="show = !show">Toggle</button>
<Transition>
  <p v-if="show">hello</p>
</Transition>
```

```css
/* 下面我们会解释这些 class 是做什么的 */
.v-enter-active,
.v-leave-active {
  transition: opacity 0.5s ease;
}
.v-enter-from,
.v-leave-to {
  opacity: 0;
}
```

Try it in the Playground

> **TIP**
> `<Transition>` only supports a single element or component as its slot content. If the content is a component, the component must also have only one single root element.

When an element in a `<Transition>` component is inserted or removed, this is what happens:

1. Vue will automatically sniff whether the target element has CSS transitions or animations applied. If it does, a number of CSS transition classes will be added / removed at appropriate timings.

2. If there are listeners for JavaScript hooks, these hooks will be called at appropriate timings.

3. If no CSS transitions / animations are detected and no JavaScript hooks are provided, the DOM operations for insertion and/or removal will be executed on the browser's next animation frame.

5.1.2 CSS-Based Transitions

Transition Classes

以下是最基本用法的示例：

```html
<button @click="show = !show">Toggle</button>
<Transition>
  <p v-if="show">hello</p>
</Transition>
```

```css
/* 下面我们会解释这些 class 是做什么的 */
.v-enter-active,
.v-leave-active {
  transition: opacity 0.5s ease;
}
.v-enter-from,
.v-leave-to {
  opacity: 0;
}
```

在演练场中尝试一下

> **TIP**
> `<Transition>` 仅支持单个元素或组件作为其插槽内容。如果内容是一个组件，这个组件必须仅有一个根元素。

当一个 `<Transition>` 组件中的元素被插入或移除时，会发生下面这些事情：

1. Vue 会自动检测目标元素是否应用了 CSS 过渡或动画。如果是，则一些 CSS 过渡 class 会在适当的时机被添加和移除。

2. 如果有作为监听器的 JavaScript 钩子，这些钩子函数会在适当时机被调用。

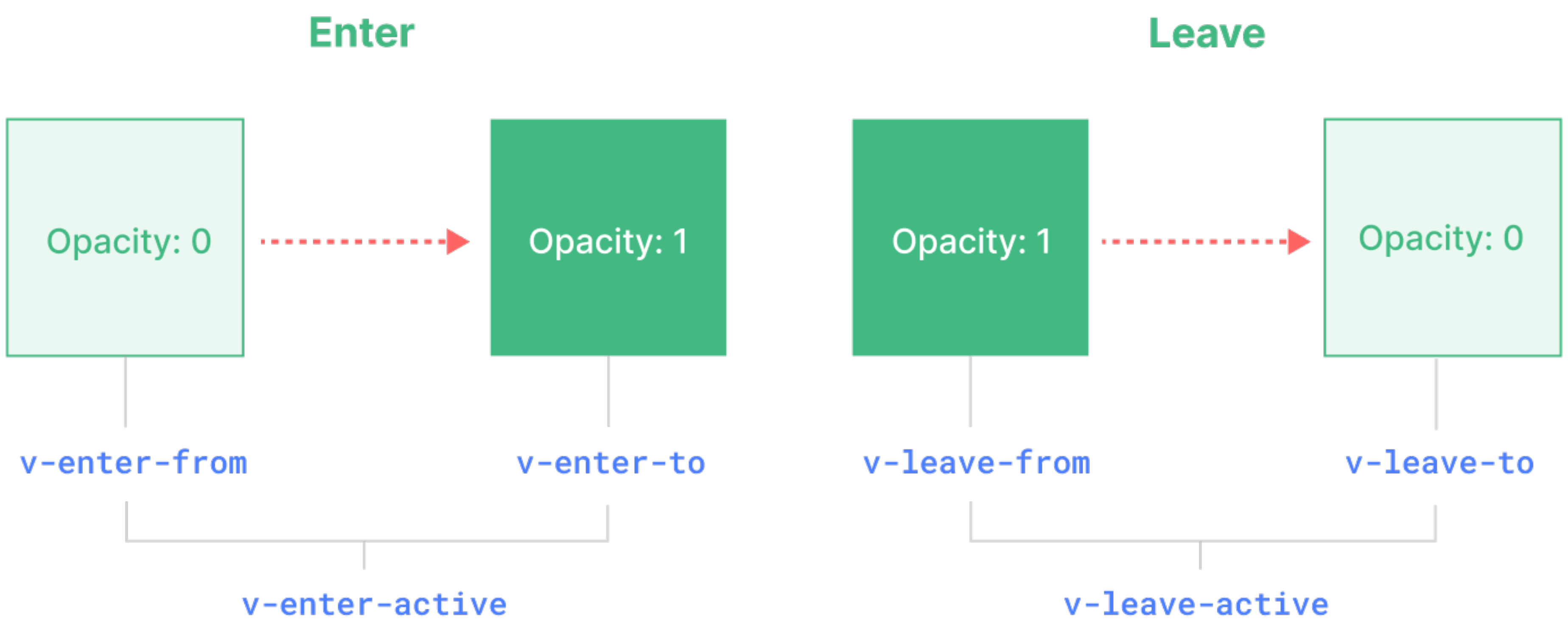
3. 如果没有探测到 CSS 过渡或动画、也没有提供 JavaScript 钩子，那么 DOM 的插入、删除操作将在浏览器的下一个动画帧后执行。

5.1.2 基于 CSS 的过渡效果

CSS 过渡 class

There are six classes applied for enter / leave transitions.

一共有 6 个应用于进入与离开过渡效果的 CSS class。



1. `v-enter-from`: Starting state for enter. Added before the element is inserted, removed one frame after the element is inserted.
2. `v-enter-active`: Active state for enter. Applied during the entire entering phase. Added before the element is inserted, removed when the transition/animation finishes. This class can be used to define the duration, delay and easing curve for the entering transition.
3. `v-enter-to`: Ending state for enter. Added one frame after the element is inserted (at the same time `v-enter-from` is removed), removed when the transition/animation finishes.
4. `v-leave-from`: Starting state for leave. Added immediately when a leaving transition is triggered, removed after one frame.
5. `v-leave-active`: Active state for leave. Applied during the entire leaving phase. Added

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

immediately when a leaving transition is triggered, removed when the transition/animation finishes. This class can be used to define the duration, delay and easing curve for the leaving transition.

6. `v-leave-to`: Ending state for leave. Added one frame after a leaving transition is triggered (at the same time `v-leave-from` is removed), removed when the transition/animation finishes.

`v-enter-active` and `v-leave-active` give us the ability to specify different easing curves for enter / leave transitions, which we'll see an example of in the following sections.

Named Transitions

A transition can be named via the `name` prop:

```html
<Transition name="fade">
  ...
</Transition>
```

For a named transition, its transition classes will be prefixed with its name instead of `v`. For example, the applied class for the above transition will be `fade-enter-active` instead of `v-enter-active`. The CSS for the fade transition should look like this:

```css
.fade-enter-active,
.fade-leave-active {
  transition: opacity 0.5s ease;
}
.fade-enter-from,
.fade-leave-to {
  opacity: 0;
}
```

CSS Transitions

`<Transition>` is most commonly used in combination with native CSS transitions, as seen in the basic example above. The `transition` CSS property is a shorthand that allows us to specify multiple aspects of a transition, including properties that should be animated, duration of the transition, and easing curves.

过渡效果被触发时立即添加，在过渡或动画完成之后移除。这个 class 可以被用来定义离开动画的持续时间、延迟与速度曲线类型。

6. `v-leave-to`：离开动画的结束状态。在一个离开动画被触发后的下一帧被添加 (也就是 `v-leave-from` 被移除的同时)，在过渡或动画完成之后移除。

`v-enter-active` 和 `v-leave-active` 给我们提供了为进入和离开动画指定不同速度曲线的能力，我们将在下面的小节中看到一个示例。

为过渡效果命名

我们可以给 `<Transition>` 组件传一个 `name` prop 来声明一个过渡效果名：

```html
<Transition name="fade">
  ...
</Transition>
```

对于一个有名字的过渡效果，对它起作用的过渡 class 会以其名字而不是 `v` 作为前缀。比如，上方例子中被应用的 class 将会是 `fade-enter-active` 而不是 `v-enter-active`。这个 “fade” 过渡的 class 应该是这样：

```css
.fade-enter-active,
.fade-leave-active {
  transition: opacity 0.5s ease;
}
.fade-enter-from,
.fade-leave-to {
  opacity: 0;
}
```

CSS 的 transition

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

Here is a more advanced example that transitions multiple properties, with different durations and easing curves for enter and leave:

html

```html
<Transition name="slide-fade">
  <p v-if="show">hello</p>
</Transition>
```

css

```css
/*
  进入和离开动画可以使用不同
  持续时间和速度曲线。
*/
.slide-fade-enter-active {
  transition: all 0.3s ease-out;
}
.slide-fade-leave-active {
  transition: all 0.8s cubic-bezier(1, 0.5, 0.8, 1);
}
.slide-fade-enter-from,
.slide-fade-leave-to {
  transform: translateX(20px);
  opacity: 0;
}
```

Try it in the Playground

CSS Animations

Native CSS animations are applied in the same way as CSS transitions, with the difference being that `*-enter-from` is not removed immediately after the element is inserted, but on an `animationend` event.

For most CSS animations, we can simply declare them under the `*-enter-active` and `*-leave-active` classes. Here's an example:

html

```html
<Transition name="bounce">
  <p v-if="show" style="text-align: center;">
    Hello here is some bouncy text!
```

下面是一个更高级的例子，它使用了不同的持续时间和速度曲线来过渡多个属性：

html

```html
<Transition name="slide-fade">
  <p v-if="show">hello</p>
</Transition>
```

css

```css
/*
  进入和离开动画可以使用不同
  持续时间和速度曲线。
*/
.slide-fade-enter-active {
  transition: all 0.3s ease-out;
}
.slide-fade-leave-active {
  transition: all 0.8s cubic-bezier(1, 0.5, 0.8, 1);
}
.slide-fade-enter-from,
.slide-fade-leave-to {
  transform: translateX(20px);
  opacity: 0;
}
```

在演练场中尝试一下

CSS 的 animation

原生 CSS 动画和 CSS transition 的应用方式基本上是相同的，只有一点不同，那就是 `*-enter-from` 不是在元素插入后立即移除，而是在一个 `animationend` 事件触发时被移除。

对于大多数的 CSS 动画，我们可以简单地在 `*-enter-active` 和 `*-leave-active` class 下声明它们。下面是一个示例：

html

```html
<Transition name="bounce">
  <p v-if="show" style="text-align: center;">
    Hello here is some bouncy text!
  </p>
</Transition>
```

css

```css
.bounce-enter-active {
  animation: bounce-in 0.5s;
}
.bounce-leave-active {
  animation: bounce-in 0.5s reverse;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.25);
  }
  100% {
    transform: scale(1);
  }
}
```

Try it in the Playground

Custom Transition Classes

You can also specify custom transition classes by passing the following props to `<Transition>`:

- `enter-from-class`
- `enter-active-class`
- `enter-to-class`
- `leave-from-class`
- `leave-active-class`
- `leave-to-class`

```
  </p>
</Transition>
```

css

```css
.bounce-enter-active {
  animation: bounce-in 0.5s;
}
.bounce-leave-active {
  animation: bounce-in 0.5s reverse;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.25);
  }
  100% {
    transform: scale(1);
  }
}
```

在演练场中尝试一下

自定义过渡 class

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

- `enter-from-class`
- `enter-active-class`
- `enter-to-class`
- `leave-from-class`
- `leave-active-class`
- `leave-to-class`

These will override the conventional class names. This is especially useful when you want to combine Vue's transition system with an existing CSS animation library, such as Animate.css:

```html
<!-- 假设你已经在页面中引入了 Animate.css -->
<Transition
  name="custom-classes"
  enter-active-class="animate__animated animate__tada"
  leave-active-class="animate__animated animate__bounceOutRight"
>
  <p v-if="show">hello</p>
</Transition>
```

Try it in the Playground

Using Transitions and Animations Together

Vue needs to attach event listeners in order to know when a transition has ended. It can either be `transitionend` or `animationend`, depending on the type of CSS rules applied. If you are only using one or the other, Vue can automatically detect the correct type.

However, in some cases you may want to have both on the same element, for example having a CSS animation triggered by Vue, along with a CSS transition effect on hover. In these cases, you will have to explicitly declare the type you want Vue to care about by passing the `type` prop, with a value of either `animation` or `transition`:

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

Nested Transitions and Explicit Transition Durations

Although the transition classes are only applied to the direct child element in `<Transition>`, we can transition nested elements using nested CSS selectors:

```html
<Transition name="nested">
  <div v-if="show" class="outer">
    <div class="inner">
      Hello
    </div>
```

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

```html
<!-- 假设你已经在页面中引入了 Animate.css -->
<Transition
  name="custom-classes"
  enter-active-class="animate__animated animate__tada"
  leave-active-class="animate__animated animate__bounceOutRight"
>
  <p v-if="show">hello</p>
</Transition>
```

在演练场中尝试一下

同时使用 transition 和 animation

Vue 需要附加事件监听器，以便知道过渡何时结束。可以是 `transitionend` 或 `animationend`，这取决于你所应用的 CSS 规则。如果你仅仅使用二者的其中之一，Vue 可以自动探测到正确的类型。

然而在某些场景中，你或许想要在同一个元素上同时使用它们两个。举例来说，Vue 触发了一个 CSS 动画，同时鼠标悬停触发另一个 CSS 过渡。此时你需要显式地传入 `type` prop 来声明，告诉 Vue 需要关心哪种类型，传入的值是 `animation` 或 `transition`：

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

深层级过渡与显式过渡时长

尽管过渡 class 仅能应用在 `<Transition>` 的直接子元素上，我们还是可以使用深层级的 CSS 选择器，在深层级的元素上触发过渡效果。

```html
<Transition name="nested">
  <div v-if="show" class="outer">
    <div class="inner">
      Hello
    </div>
  </div>
</Transition>
```

css

```css
/* 应用于嵌套元素的规则 */
.nested-enter-active .inner,
.nested-leave-active .inner {
  transition: all 0.3s ease-in-out;
}
.nested-enter-from .inner,
.nested-leave-to .inner {
  transform: translateX(30px);
  opacity: 0;
}
/* ... 省略了其他必要的 CSS */
```

We can even add a transition delay to the nested element on enter, which creates a staggered enter animation sequence:

css

```css
/* 延迟嵌套元素的进入以获得交错效果 */
.nested-enter-active .inner {
  transition-delay: 0.25s;
}
```

However, this creates a small issue. By default, the `<Transition>` component attempts to automatically figure out when the transition has finished by listening to the **first** `transitionend` or `animationend` event on the root transition element. With a nested transition, the desired behavior should be waiting until the transitions of all inner elements have finished.

In such cases you can specify an explicit transition duration (in milliseconds) using the `duration` prop on the `<transition>` component. The total duration should match the delay plus transition duration of the inner element:

html

```html
<Transition :duration="550">...</Transition>
```

Try it in the Playground

If necessary, you can also specify separate values for enter and leave durations using an object:

```
  </div>
</Transition>
```

css

```css
/* 应用于嵌套元素的规则 */
.nested-enter-active .inner,
.nested-leave-active .inner {
  transition: all 0.3s ease-in-out;
}
.nested-enter-from .inner,
.nested-leave-to .inner {
  transform: translateX(30px);
  opacity: 0;
}
/* ... 省略了其他必要的 CSS */
```

我们甚至可以在深层元素上添加一个过渡延迟，从而创建一个带渐进延迟的动画序列：

css

```css
/* 延迟嵌套元素的进入以获得交错效果 */
.nested-enter-active .inner {
  transition-delay: 0.25s;
}
```

然而，这会带来一个小问题。默认情况下，`<Transition>` 组件会通过监听过渡根元素上的**第一个** `transitionend` 或者 `animationend` 事件来尝试自动判断过渡何时结束。而在嵌套的过渡中，期望的行为应该是等待所有内部元素的过渡完成。

在这种情况下，你可以通过向 `<Transition>` 组件传入 `duration` prop 来显式指定过渡的持续时间 (以毫秒为单位)。总持续时间应该匹配延迟加上内部元素的过渡持续时间：

html

```html
<Transition :duration="550">...</Transition>
```

在演练场中尝试一下

如果有必要的话，你也可以用对象的形式传入，分开指定进入和离开所需的时间：

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

Performance Considerations

You may notice that the animations shown above are mostly using properties like `transform` and `opacity`. These properties are efficient to animate because:

1. They do not affect the document layout during the animation, so they do not trigger expensive CSS layout calculation on every animation frame.
2. Most modern browsers can leverage GPU hardware acceleration when animating `transform`.

In comparison, properties like `height` or `margin` will trigger CSS layout, so they are much more expensive to animate, and should be used with caution. We can check resources like CSS-Triggers to see which properties will trigger layout if we animate them.

5.1.3 JavaScript Hooks

You can hook into the transition process with JavaScript by listening to events on the `<Transition>` component:

html
```html
<Transition
  @before-enter="onBeforeEnter"
  @enter="onEnter"
  @after-enter="onAfterEnter"
  @enter-cancelled="onEnterCancelled"
  @before-leave="onBeforeLeave"
  @leave="onLeave"
  @after-leave="onAfterLeave"
  @leave-cancelled="onLeaveCancelled"
>
  <!-- ... -->
</Transition>
```

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

性能考量

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

1. 他们在动画过程中不会影响到 DOM 结构，因此不会每一帧都触发昂贵的 CSS 布局重新计算。
2. 大多数的现代浏览器都可以在执行 `transform` 动画时利用 GPU 进行硬件加速。

相比之下，像 `height` 或者 `margin` 这样的属性会触发 CSS 布局变动，因此执行它们的动画效果更昂贵，需要谨慎使用。我们可以在 CSS-Triggers 这类的网站查询哪些属性会在执行动画时触发 CSS 布局变动。

5.1.3 JavaScript 钩子

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

html
```html
<Transition
  @before-enter="onBeforeEnter"
  @enter="onEnter"
  @after-enter="onAfterEnter"
  @enter-cancelled="onEnterCancelled"
  @before-leave="onBeforeLeave"
  @leave="onLeave"
  @after-leave="onAfterLeave"
  @leave-cancelled="onLeaveCancelled"
>
  <!-- ... -->
</Transition>
```

```js
// 在元素被插入到 DOM 之前被调用
// 用这个来设置元素的 "enter-from" 状态
function onBeforeEnter(el) {}
// 在元素被插入到 DOM 之后的下一帧被调用
// 用这个来开始进入动画
function onEnter(el, done) {
  // 调用回调函数 done 表示过渡结束
  // 如果与 CSS 结合使用，则这个回调是可选参数
  done()
}
// 当进入过渡完成时调用。
function onAfterEnter(el) {}
// 当进入过渡在完成之前被取消时调用
function onEnterCancelled(el) {}
// 在 leave 钩子之前调用
// 大多数时候，你应该只会用到 leave 钩子
function onBeforeLeave(el) {}
// 在离开过渡开始时调用
// 用这个来开始离开动画
function onLeave(el, done) {
  // 调用回调函数 done 表示过渡结束
  // 如果与 CSS 结合使用，则这个回调是可选参数
  done()
}
// 在离开过渡完成、
// 且元素已从 DOM 中移除时调用
function onAfterLeave(el) {}
// 仅在 v-show 过渡中可用
function onLeaveCancelled(el) {}
```

These hooks can be used in combination with CSS transitions / animations or on their own.

When using JavaScript-only transitions, it is usually a good idea to add the `:css="false"` prop. This explicitly tells Vue to skip auto CSS transition detection. Aside from being slightly more performant, this also prevents CSS rules from accidentally interfering with the transition:

```js
// 在元素被插入到 DOM 之前被调用
// 用这个来设置元素的 "enter-from" 状态
function onBeforeEnter(el) {}
// 在元素被插入到 DOM 之后的下一帧被调用
// 用这个来开始进入动画
function onEnter(el, done) {
  // 调用回调函数 done 表示过渡结束
  // 如果与 CSS 结合使用，则这个回调是可选参数
  done()
}
// 当进入过渡完成时调用。
function onAfterEnter(el) {}
// 当进入过渡在完成之前被取消时调用
function onEnterCancelled(el) {}
// 在 leave 钩子之前调用
// 大多数时候，你应该只会用到 leave 钩子
function onBeforeLeave(el) {}
// 在离开过渡开始时调用
// 用这个来开始离开动画
function onLeave(el, done) {
  // 调用回调函数 done 表示过渡结束
  // 如果与 CSS 结合使用，则这个回调是可选参数
  done()
}
// 在离开过渡完成、
// 且元素已从 DOM 中移除时调用
function onAfterLeave(el) {}
// 仅在 v-show 过渡中可用
function onLeaveCancelled(el) {}
```

这些钩子可以与 CSS 过渡或动画结合使用，也可以单独使用。

在使用仅由 JavaScript 执行的动画时，最好是添加一个 `:css="false"` prop。这显式地向 Vue 表明可以跳过对 CSS 过渡的自动探测。除了性能稍好一些之外，还可以防止 CSS 规则意外地干扰过渡效果。

css

```
<Transition
  ...
  :css="false"
>
  ...
</Transition>
```

With `:css="false"`, we are also fully responsible for controlling when the transition ends. In this case, the `done` callbacks are required for the `@enter` and `@leave` hooks. Otherwise, the hooks will be called synchronously and the transition will finish immediately.

Here's a demo using the GreenSock library to perform the animations. You can, of course, use any other animation library you want, for example Anime.js or Motion One.

Try it in the Playground

5.1.4　Reusable Transitions

Transitions can be reused through Vue's component system. To create a reusable transition, we can create a component that wraps the `<Transition>` component and passes down the slot content:

html

```
<!-- MyTransition.vue -->
<script>
// JavaScript 钩子逻辑...
</script>
<template>
  <!-- 包装内置的 Transition 组件 -->
  <Transition
    name="my-transition"
    @enter="onEnter"
    @leave="onLeave">
    <slot></slot> <!-- 向内传递插槽内容 -->
  </Transition>
</template>
<style>
/*
```

css

```
<Transition
  ...
  :css="false"
>
  ...
</Transition>
```

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

这里是使用 GreenSock 库执行动画的一个示例，你也可以使用任何你想要的库，比如 Anime.js 或者 Motion One。

在演练场中尝试一下

5.1.4　可复用过渡效果

得益于 Vue 的组件系统，过渡效果是可以被封装复用的。要创建一个可被复用的过渡，我们需要为 `<Transition>` 组件创建一个包装组件，并向内传入插槽内容：

html

```
<!-- MyTransition.vue -->
<script>
// JavaScript 钩子逻辑...
</script>
<template>
  <!-- 包装内置的 Transition 组件 -->
  <Transition
    name="my-transition"
    @enter="onEnter"
    @leave="onLeave">
    <slot></slot> <!-- 向内传递插槽内容 -->
  </Transition>
</template>
<style>
/*
```

```
  必要的 CSS...
  注意：避免在这里使用 <style scoped>
  因为那不会应用到插槽内容上
*/
</style>
```

Now MyTransition can be imported and used just like the built-in version:

```html
<MyTransition>
  <div v-if="show">Hello</div>
</MyTransition>
```

5.1.5 Transition on Appear

If you also want to apply a transition on the initial render of a node, you can add the appear prop:

```html
<Transition appear>
  ...
</Transition>
```

5.1.6 Transition Between Elements

In addition to toggling an element with v-if / v-show, we can also transition between two elements using v-if / v-else / v-else-if, as long as we make sure that there is only one element being shown at any given moment:

```html
<Transition>
  <button v-if="docState === 'saved'">Edit</button>
  <button v-else-if="docState === 'edited'">Save</button>
  <button v-else-if="docState === 'editing'">Cancel</button>
</Transition>
```

Try it in the Playground

```
  必要的 CSS...
  注意：避免在这里使用 <style scoped>
  因为那不会应用到插槽内容上
*/
</style>
```

现在 MyTransition 可以在导入后像内置组件那样使用了：

```html
<MyTransition>
  <div v-if="show">Hello</div>
</MyTransition>
```

5.1.5 出现时过渡

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

```html
<Transition appear>
  ...
</Transition>
```

5.1.6 元素间过渡

除了通过 v-if / v-show 切换一个元素，我们也可以通过 v-if / v-else / v-else-if 在几个组件间进行切换，只要确保任一时刻只会有一个元素被渲染即可：

```html
<Transition>
  <button v-if="docState === 'saved'">Edit</button>
  <button v-else-if="docState === 'edited'">Save</button>
  <button v-else-if="docState === 'editing'">Cancel</button>
</Transition>
```

在演练场中尝试一下

5.1.7 Transition Modes

In the previous example, the entering and leaving elements are animated at the same time, and we had to make them `position: absolute` to avoid the layout issue when both elements are present in the DOM.

However, in some cases this isn't an option, or simply isn't the desired behavior. We may want the leaving element to be animated out first, and for the entering element to only be inserted **after** the leaving animation has finished. Orchestrating such animations manually would be very complicated - luckily, we can enable this behavior by passing `<Transition>` a `mode` prop:

html

```html
<Transition mode="out-in">
  ...
</Transition>
```

Here's the previous demo with `mode="out-in"`:

`<Transition>` also supports `mode="in-out"`, although it's much less frequently used.

5.1.8 Transition Between Components

`<Transition>` can also be used around dynamic components:

html

```html
<Transition name="fade" mode="out-in">
  <component :is="activeComponent"></component>
</Transition>
```

Try it in the Playground

5.1.9 Dynamic Transitions

`<Transition>` props like `name` can also be dynamic! It allows us to dynamically apply different transitions based on state change:

html

```html
<Transition :name="transitionName">
  <!-- ... -->
</Transition>
```

5.1.7 过渡模式

在之前的例子中，进入和离开的元素都是在同时开始动画的，因此我们不得不将它们设为 `position: absolute` 以避免二者同时存在时出现的布局问题。

然而，很多情况下这可能并不符合需求。我们可能想要先执行离开动画，然后在其完成**之后**再执行元素的进入动画。手动编排这样的动画是非常复杂的，好在我们可以通过向 `<Transition>` 传入一个 `mode` prop 来实现这个行为：

html

```html
<Transition mode="out-in">
  ...
</Transition>
```

将之前的例子改为 `mode="out-in"` 后是这样：

`<Transition>` 也支持 `mode="in-out"`，虽然这并不常用。

5.1.8 组件间过渡

`<Transition>` 也可以作用于动态组件之间的切换：

html

```html
<Transition name="fade" mode="out-in">
  <component :is="activeComponent"></component>
</Transition>
```

在演练场中尝试一下

5.1.9 动态过渡

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

html

```html
<Transition :name="transitionName">
  <!-- ... -->
</Transition>
```

This can be useful when you've defined CSS transitions / animations using Vue's transition class conventions and want to switch between them.

You can also apply different behavior in JavaScript transition hooks based on the current state of your component. Finally, the ultimate way of creating dynamic transitions is through reusable transition components that accept props to change the nature of the transition(s) to be used. It may sound cheesy, but the only limit really is your imagination.

Related

- “ API reference

5.2 TransitionGroup

`<TransitionGroup>` is a built-in component designed for animating the insertion, removal, and order change of elements or components that are rendered in a list.

5.2.1 Differences from <Transition>

`<TransitionGroup>` supports the same props, CSS transition classes, and JavaScript hook listeners as `<Transition>`, with the following differences:

- By default, it doesn't render a wrapper element. But you can specify an element to be rendered with the `tag` prop.
- Transition modes are not available, because we are no longer alternating between mutually exclusive elements.
- Elements inside are **always required** to have a unique `key` attribute.
- CSS transition classes will be applied to individual elements in the list, **not** to the group / container itself.

TIP
When used in in-DOM templates, it should be referenced as `<transition-group>`.

这个特性的用处是可以提前定义好多组 CSS 过渡或动画的 class，然后在它们之间动态切换。

你也可以根据你的组件的当前状态在 JavaScript 过渡钩子中应用不同的行为。最后，创建动态过渡的终极方式还是创建可复用的过渡组件，并让这些组件根据动态的 props 来改变过渡的效果。掌握了这些技巧后，就真的只有你想不到，没有做不到的了。

参考

- “ API 参考

5.2 TransitionGroup

`<TransitionGroup>` 是一个内置组件，用于对 `v-for` 列表中的元素或组件的插入、移除和顺序改变添加动画效果。

5.2.1 和 <Transition> 的区别

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

- 默认情况下，它不会渲染一个容器元素。但你可以通过传入 `tag` prop 来指定一个元素作为容器元素来渲染。
- 过渡模式在这里不可用，因为我们不再是在互斥的元素之间进行切换。
- 列表中的每个元素都**必须**有一个独一无二的 `key` attribute。
- CSS 过渡 class 会被应用在列表内的元素上，**而不是**容器元素上。

TIP
当在 DOM 内模板中使用时，组件名需要写为 `<transition-group>`。

5.2.2 Enter / Leave Transitions

Here is an example of applying enter / leave transitions to a v-for list using <TransitionGroup>:

html
```html
<TransitionGroup name="list" tag="ul">
  <li v-for="item in items" :key="item">
    {{ item }}
  </li>
</TransitionGroup>
```

css
```css
.list-enter-active,
.list-leave-active {
  transition: all 0.5s ease;
}
.list-enter-from,
.list-leave-to {
  opacity: 0;
  transform: translateX(30px);
}
```

5.2.3 Move Transitions

The above demo has some obvious flaws: when an item is inserted or removed, its surrounding items instantly "jump" into place instead of moving smoothly. We can fix this by adding a few additional CSS rules:

css
```css
.list-move, /* 对移动中的元素应用的过渡 */
.list-enter-active,
.list-leave-active {
  transition: all 0.5s ease;
}
.list-enter-from,
.list-leave-to {
  opacity: 0;
  transform: translateX(30px);
}
```

5.2.2 进入 / 离开动画

这里是 <TransitionGroup> 对一个 v-for 列表添加进入 / 离开动画的示例：

html
```html
<TransitionGroup name="list" tag="ul">
  <li v-for="item in items" :key="item">
    {{ item }}
  </li>
</TransitionGroup>
```

css
```css
.list-enter-active,
.list-leave-active {
  transition: all 0.5s ease;
}
.list-enter-from,
.list-leave-to {
  opacity: 0;
  transform: translateX(30px);
}
```

5.2.3 移动动画

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

css
```css
.list-move, /* 对移动中的元素应用的过渡 */
.list-enter-active,
.list-leave-active {
  transition: all 0.5s ease;
}
.list-enter-from,
.list-leave-to {
  opacity: 0;
  transform: translateX(30px);
}
```

```
/* 确保将离开的元素从布局流中删除
  以便能够正确地计算移动的动画。 */
.list-leave-active {
  position: absolute;
}
```

Now it looks much better - even animating smoothly when the whole list is shuffled:

Full Example

5.2.4 Staggering List Transitions

By communicating with JavaScript transitions through data attributes, it's also possible to stagger transitions in a list. First, we render the index of an item as a data attribute on the DOM element:

```html
<TransitionGroup
  tag="ul"
  :css="false"
  @before-enter="onBeforeEnter"
  @enter="onEnter"
  @leave="onLeave"
>
  <li
    v-for="(item, index) in computedList"
    :key="item.msg"
    :data-index="index"
  >
    {{ item.msg }}
  </li>
</TransitionGroup>
```

Then, in JavaScript hooks, we animate the element with a delay based on the data attribute. This example is using the GreenSock library to perform the animation:

```js
function onEnter(el, done) {
  gsap.to(el, {
    opacity: 1,
```

```
/* 确保将离开的元素从布局流中删除
  以便能够正确地计算移动的动画。 */
.list-leave-active {
  position: absolute;
}
```

现在它看起来好多了，甚至对整个列表执行洗牌的动画也都非常流畅：

完整的示例

5.2.4 渐进延迟列表动画

通过在 JavaScript 钩子中读取元素的 data attribute，我们可以实现带渐进延迟的列表动画。首先，我们把每一个元素的索引渲染为该元素上的一个 data attribute：

```html
<TransitionGroup
  tag="ul"
  :css="false"
  @before-enter="onBeforeEnter"
  @enter="onEnter"
  @leave="onLeave"
>
  <li
    v-for="(item, index) in computedList"
    :key="item.msg"
    :data-index="index"
  >
    {{ item.msg }}
  </li>
</TransitionGroup>
```

接着，在 JavaScript 钩子中，我们基于当前元素的 data attribute 对该元素的进场动画添加一个延迟。以下是一个基于 GreenSock library 的动画示例：

```js
function onEnter(el, done) {
  gsap.to(el, {
    opacity: 1,
    height: '1.6em',
    delay: el.dataset.index * 0.15,
    onComplete: done
  })
}
```

Full Example in the Playground

Related

- “ API reference

5.3 KeepAlive

`<KeepAlive>` is a built-in component that allows us to conditionally cache component instances when dynamically switching between multiple components.

5.3.1 Basic Usage

In the Component Basics chapter, we introduced the syntax for Dynamic Components, using the `<component>` special element:

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

By default, an active component instance will be unmounted when switching away from it. This will cause any changed state it holds to be lost. When this component is displayed again, a new instance will be created with only the initial state.

In the example below, we have two stateful components - A contains a counter, while B contains a message synced with an input via `v-model`. Try updating the state of one of them, switch away, and then switch back to it:

You’ll notice that when switched back, the previous changed state would have been reset.

Creating fresh component instance on switch is normally useful behavior, but in this case, we’d really like the two component instances to be preserved even when they are inactive. To solve this

```
    height: '1.6em',
    delay: el.dataset.index * 0.15,
    onComplete: done
  })
}
```

在演练场中查看完整示例

参考

- “ API 参考

5.3 KeepAlive

`<KeepAlive>` 是一个内置组件，它的功能是在多个组件间动态切换时缓存被移除的组件实例。

5.3.1 基本使用

在组件基础章节中，我们已经介绍了通过特殊的 `<component>` 元素来实现动态组件的用法：

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

默认情况下，一个组件实例在被替换掉后会被销毁。这会导致它丢失其中所有已变化的状态——当这个组件再一次被显示时，会创建一个只带有初始状态的新实例。

在下面的例子中，你会看到两个有状态的组件——A 有一个计数器，而 B 有一个通过 `v-model` 同步 input 框输入内容的文字展示。尝试先更改一下任意一个组件的状态，然后切走，再切回来：

你会发现在切回来之后，之前已更改的状态都被重置了。

在切换时创建新的组件实例通常是有意义的，但在这个例子中，我们的确想要组件能在被“切走”的时候保留它们的状态。要解决这个问题，我们可以用 `<KeepAlive>`

problem, we can wrap our dynamic component with the `<KeepAlive>` built-in component:

```html
<!-- 非活跃的组件将会被缓存！ -->
<KeepAlive>
  <component :is="activeComponent" />
</KeepAlive>
```

Now, the state will be persisted across component switches:

Try it in the Playground

TIP
When used in in-DOM templates, it should be referenced as `<keep-alive>`.

5.3.2 Include / Exclude

By default, `<KeepAlive>` will cache any component instance inside. We can customize this behavior via the `include` and `exclude` props. Both props can be a comma-delimited string, a `RegExp`, or an array containing either types:

```html
<!-- 以英文逗号分隔的字符串 -->
<KeepAlive include="a,b">
    <component :is="view" />
</KeepAlive>
<!-- 正则表达式 (需使用 `v-bind`) -->
<KeepAlive :include="/a|b/">
    <component :is="view" />
</KeepAlive>
<!-- 数组 (需使用 `v-bind`) -->
<KeepAlive :include="['a', 'b']">
    <component :is="view" />
</KeepAlive>
```

The match is checked against the component's `name` option, so components that need to be conditionally cached by `KeepAlive` must explicitly declare a `name` option.

内置组件将这些动态组件包装起来：

```html
<!-- 非活跃的组件将会被缓存！ -->
<KeepAlive>
  <component :is="activeComponent" />
</KeepAlive>
```

现在，在组件切换时状态也能被保留了：

在演练场中尝试一下

TIP
在 DOM 内模板中使用时，它应该被写为 `<keep-alive>`。

5.3.2 包含/排除

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

```html
<!-- 以英文逗号分隔的字符串 -->
<KeepAlive include="a,b">
  <component :is="view" />
</KeepAlive>
<!-- 正则表达式 (需使用 `v-bind`) -->
<KeepAlive :include="/a|b/">
  <component :is="view" />
</KeepAlive>
<!-- 数组 (需使用 `v-bind`) -->
<KeepAlive :include="['a', 'b']">
  <component :is="view" />
</KeepAlive>
```

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

TIP

Since version 3.2.34, a single-file component using `<script setup>` will automatically infer its `name` option based on the filename, removing the need to manually declare the name.

5.3.3 Max Cached Instances

We can limit the maximum number of component instances that can be cached via the `max` prop. When `max` is specified, `<KeepAlive>` behaves like an LRU cache: if the number of cached instances is about to exceed the specified max count, the least recently accessed cached instance will be destroyed to make room for the new one.

```html
<KeepAlive :max="10">
  <component :is="activeComponent" />
</KeepAlive>
```

5.3.4 Lifecycle of Cached Instance

When a component instance is removed from the DOM but is part of a component tree cached by `<KeepAlive>`, it goes into a **deactivated** state instead of being unmounted. When a component instance is inserted into the DOM as part of a cached tree, it is **activated**.

A kept-alive component can register lifecycle hooks for these two states using `onActivated()` and `onDeactivated()`:

```html
<script setup>
import { onActivated, onDeactivated } from 'vue'
onActivated(() => {
  // 调用时机为首次挂载
  // 以及每次从缓存中被重新插入时
})
onDeactivated(() => {
  // 在从 DOM 上移除、进入缓存
  // 以及组件卸载时调用
})
```

TIP

在 3.2.34 或以上的版本中，使用 `<script setup>` 的单文件组件会自动根据文件名生成对应的 `name` 选项，无需再手动声明。

5.3.3 最大缓存实例数

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

```html
<KeepAlive :max="10">
  <component :is="activeComponent" />
</KeepAlive>
```

5.3.4 缓存实例的生命周期

当一个组件实例从 DOM 上移除但因为被 `<KeepAlive>` 缓存而仍作为组件树的一部分时，它将变为**不活跃**状态而不是被卸载。当一个组件实例作为缓存树的一部分插入到 DOM 中时，它将重新**被激活**。

一个持续存在的组件可以通过 `onActivated()` 和 `onDeactivated()` 注册相应的两个状态的生命周期钩子：

```html
<script setup>
import { onActivated, onDeactivated } from 'vue'
onActivated(() => {
  // 调用时机为首次挂载
  // 以及每次从缓存中被重新插入时
})
onDeactivated(() => {
  // 在从 DOM 上移除、进入缓存
  // 以及组件卸载时调用
})
```

```
</script>
```

Note that:

- `onActivated` is also called on mount, and `onDeactivated` on unmount.
- Both hooks work for not only the root component cached by `<KeepAlive>`, but also descendant components in the cached tree.

Related

- “ API reference

5.4 Teleport

`<Teleport>` is a built-in component that allows us to ”teleport” a part of a component’s template into a DOM node that exists outside the DOM hierarchy of that component.

5.4.1 Basic Usage

Sometimes we may run into the following scenario: a part of a component’s template belongs to it logically, but from a visual standpoint, it should be displayed somewhere else in the DOM, outside of the Vue application.

The most common example of this is when building a full-screen modal. Ideally, we want the modal’s button and the modal itself to live within the same component, since they are both related to the open / close state of the modal. But that means the modal will be rendered alongside the button, deeply nested in the application’s DOM hierarchy. This can create some tricky issues when positioning the modal via CSS.

Consider the following HTML structure.

html

```html
<div class="outer">
  <h3>Tooltips with Vue 3 Teleport</h3>
  <div>
```

```
</script>
```

请注意：

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

参考

- “ API 参考

5.4 Teleport

`<Teleport>` 是一个内置组件，它可以将一个组件内部的一部分模板“传送”到该组件的 DOM 结构外层的位置去。

5.4.1 基本用法

有时我们可能会遇到这样的场景：一个组件模板的一部分在逻辑上从属于该组件，但从整个应用视图的角度来看，它在 DOM 中应该被渲染在整个 Vue 应用外部的其他地方。

这类场景最常见的例子就是全屏的模态框。理想情况下，我们希望触发模态框的按钮和模态框本身是在同一个组件中，因为它们都与组件的开关状态有关。但这意味着该模态框将与按钮一起渲染在应用 DOM 结构里很深的地方。这会导致该模态框的 CSS 布局代码很难写。

试想下面这样的 HTML 结构：

html

```html
<div class="outer">
  <h3>Tooltips with Vue 3 Teleport</h3>
  <div>
```

```
    <MyModal />
  </div>
</div>
```

And here is the implementation of `<MyModal>`:

```html
<script setup>
import { ref } from 'vue'
const open = ref(false)
</script>
<template>
  <button @click="open = true">Open Modal</button>
  <div v-if="open" class="modal">
    <p>Hello from the modal!</p>
    <button @click="open = false">Close</button>
  </div>
</template>
<style scoped>
.modal {
  position: fixed;
  z-index: 999;
  top: 20%;
  left: 50%;
  width: 300px;
  margin-left: -150px;
}
</style>
```

The component contains a `<button>` to trigger the opening of the modal, and a `<div>` with a class of `.modal`, which will contain the modal's content and a button to self-close.

When using this component inside the initial HTML structure, there are a number of potential issues:

- `position: fixed` only places the element relative to the viewport when no ancestor element has `transform`, `perspective` or `filter` property set. If, for example, we intend to animate the ancestor `<div class="outer">` with a CSS transform, it would break the modal layout!

```
    <MyModal />
  </div>
</div>
```

接下来我们来看看 `<MyModal>` 的实现：

```html
<script setup>
import { ref } from 'vue'
const open = ref(false)
</script>
<template>
  <button @click="open = true">Open Modal</button>
  <div v-if="open" class="modal">
    <p>Hello from the modal!</p>
    <button @click="open = false">Close</button>
  </div>
</template>
<style scoped>
.modal {
  position: fixed;
  z-index: 999;
  top: 20%;
  left: 50%;
  width: 300px;
  margin-left: -150px;
}
</style>
```

这个组件中有一个 `<button>` 按钮来触发打开模态框，和一个 class 名为 `.modal` 的 `<div>`，它包含了模态框的内容和一个用来关闭的按钮。

当在初始 HTML 结构中使用这个组件时，会有一些潜在的问题：

- `position: fixed` 能够相对于浏览器窗口放置有一个条件，那就是不能有任何祖先元素设置了 `transform`、`perspective` 或者 `filter` 样式属性。也就是说如果我们想要用 CSS `transform` 为祖先节点 `<div class="outer">` 设

- The modal's `z-index` is constrained by its containing elements. If there is another element that overlaps with `<div class="outer">` and has a higher `z-index`, it would cover our modal.

`<Teleport>` provides a clean way to work around these, by allowing us to break out of the nested DOM structure. Let's modify `<MyModal>` to use `<Teleport>`:

```html
<button @click="open = true">Open Modal</button>
<Teleport to="body">
  <div v-if="open" class="modal">
    <p>Hello from the modal!</p>
    <button @click="open = false">Close</button>
  </div>
</Teleport>
```

The `to` target of `<Teleport>` expects a CSS selector string or an actual DOM node. Here, we are essentially telling Vue to "**teleport** this template fragment **to** the `body` tag".

You can click the button below and inspect the `<body>` tag via your browser's devtools:

You can combine `<Teleport>` with " to create animated modals - see Example here.

TIP
The teleport `to` target must be already in the DOM when the `<Teleport>` component is mounted. Ideally, this should be an element outside the entire Vue application. If targeting another element rendered by Vue, you need to make sure that element is mounted before the `<Teleport>`.

5.4.2 Using with Components

`<Teleport>` only alters the rendered DOM structure - it does not affect the logical hierarchy of the components. That is to say, if `<Teleport>` contains a component, that component will remain a logical child of the parent component containing the `<Teleport>`. Props passing and event emitting

置动画，就会不小心破坏模态框的布局！

- 这个模态框的 `z-index` 受限于它的容器元素。如果有其他元素与 `<div class="outer">` 重叠并有更高的 `z-index`，则它会覆盖住我们的模态框。

`<Teleport>` 提供了一个更简单的方式来解决此类问题，让我们不需要再顾虑 DOM 结构的问题。让我们用 `<Teleport>` 改写一下 `<MyModal>`：

```html
<button @click="open = true">Open Modal</button>
<Teleport to="body">
  <div v-if="open" class="modal">
    <p>Hello from the modal!</p>
    <button @click="open = false">Close</button>
  </div>
</Teleport>
```

`<Teleport>` 接收一个 `to` prop 来指定传送的目标。`to` 的值可以是一个 CSS 选择器字符串，也可以是一个 DOM 元素对象。这段代码的作用就是告诉 Vue"把以下模板片段**传送到** `body` 标签下"。

你可以点击下面这个按钮，然后通过浏览器的开发者工具，在 `<body>` 标签下找到模态框元素：

我们也可以将 `<Teleport>` 和 " 结合使用来创建一个带动画的模态框。你可以看看这个示例。

TIP
`<Teleport>` 挂载时，传送的 `to` 目标必须已经存在于 DOM 中。理想情况下，这应该是整个 Vue 应用 DOM 树外部的一个元素。如果目标元素也是由 Vue 渲染的，你需要确保在挂载 `<Teleport>` 之前先挂载该元素。

5.4.2 搭配组件使用

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

will continue to work the same way.

This also means that injections from a parent component work as expected, and that the child component will be nested below the parent component in the Vue Devtools, instead of being placed where the actual content moved to.

5.4.3 Disabling Teleport

In some cases, we may want to conditionally disable `<Teleport>`. For example, we may want to render a component as an overlay for desktop, but inline on mobile. `<Teleport>` supports the `disabled` prop which can be dynamically toggled:

```html
<Teleport :disabled="isMobile">
  ...
</Teleport>
```

Where the `isMobile` state can be dynamically updated by detecting media query changes.

5.4.4 Multiple Teleports on the Same Target

A common use case would be a reusable `<Modal>` component, with the potential for multiple instances to be active at the same time. For this kind of scenario, multiple `<Teleport>` components can mount their content to the same target element. The order will be a simple append - later mounts will be located after earlier ones within the target element.

Given the following usage:

```html
<Teleport to="#modals">
  <div>A</div>
</Teleport>
<Teleport to="#modals">
  <div>B</div>
</Teleport>
```

The rendered result would be:

```html
<div id="modals">
```

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

5.4.3 禁用 Teleport

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

```html
<Teleport :disabled="isMobile">
  ...
</Teleport>
```

这里的 `isMobile` 状态可以根据 CSS media query 的不同结果动态地更新。

5.4.4 多个 Teleport 共享目标

一个可重用的模态框组件可能同时存在多个实例。对于此类场景，多个 `<Teleport>` 组件可以将其内容挂载在同一个目标元素上，而顺序就是简单的顺次追加，后挂载的将排在目标元素下更后面的位置上。

比如下面这样的用例：

```html
<Teleport to="#modals">
  <div>A</div>
</Teleport>
<Teleport to="#modals">
  <div>B</div>
</Teleport>
```

渲染的结果为：

```html
<div id="modals">
```

```
  <div>A</div>
  <div>B</div>
</div>
```

Related

- “ API reference
- Handling Teleports in SSR

5.5 Suspense

Experimental Feature

`<Suspense>` is an experimental feature. It is not guaranteed to reach stable status and the API may change before it does.

`<Suspense>` is a built-in component for orchestrating async dependencies in a component tree. It can render a loading state while waiting for multiple nested async dependencies down the component tree to be resolved.

5.5.1 Async Dependencies

To explain the problem `<Suspense>` is trying to solve and how it interacts with these async dependencies, let’s imagine a component hierarchy like the following:

```
<Suspense>
<Dashboard>
   <Profile>
     <FriendStatus>（组件有异步的 setup()）
   <Content>
     <ActivityFeed> （异步组件）
     <Stats>（异步组件）
```

In the component tree there are multiple nested components whose rendering depends on some

```
  <div>A</div>
  <div>B</div>
</div>
```

参考

- “ API 参考
- 在 SSR 中处理 Teleports

5.5 Suspense

实验性功能

`<Suspense>` 是一项实验性功能。它不一定会最终成为稳定功能，并且在稳定之前相关 API 也可能会发生变化。

`<Suspense>` 是一个内置组件，用来在组件树中协调对异步依赖的处理。它让我们可以在组件树上层等待下层的多个嵌套异步依赖项解析完成，并可以在等待时渲染一个加载状态。

5.5.1 异步依赖

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

```
<Suspense>
<Dashboard>
   <Profile>
     <FriendStatus>（组件有异步的 setup()）
   <Content>
     <ActivityFeed> （异步组件）
     <Stats>（异步组件）
```

在这个组件树中有多个嵌套组件，要渲染出它们，首先得解析一些异步资源。如果

async resource to be resolved first. Without `<Suspense>`, each of them will need to handle its own loading / error and loaded states. In the worst case scenario, we may see three loading spinners on the page, with content displayed at different times.

The `<Suspense>` component gives us the ability to display top-level loading / error states while we wait on these nested async dependencies to be resolved.

There are two types of async dependencies that `<Suspense>` can wait on:

1. Components with an async `setup()` hook. This includes components using `<script setup>` with top-level `await` expressions.

2. Async Components.

async setup()

A Composition API component's `setup()` hook can be async:

```js
export default {
  async setup() {
    const res = await fetch(...)
    const posts = await res.json()
    return {
      posts
    }
  }
}
```

If using `<script setup>`, the presence of top-level `await` expressions automatically makes the component an async dependency:

```html
<script setup>
const res = await fetch(...)
const posts = await res.json()
</script>
<template>
  {{ posts }}
</template>
```

没有 `<Suspense>`，则它们每个都需要处理自己的加载、报错和完成状态。在最坏的情况下，我们可能会在页面上看到三个旋转的加载态，在不同的时间显示出内容。

有了 `<Suspense>` 组件后，我们就可以在等待整个多层级组件树中的各个异步依赖获取结果时，在顶层展示出加载中或加载失败的状态。

`<Suspense>` 可以等待的异步依赖有两种：

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

2. 异步组件。

async setup()

组合式 API 中组件的 `setup()` 钩子可以是异步的：

```js
export default {
  async setup() {
    const res = await fetch(...)
    const posts = await res.json()
    return {
      posts
    }
  }
}
```

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

```html
<script setup>
const res = await fetch(...)
const posts = await res.json()
</script>
<template>
  {{ posts }}
</template>
```

Async Components

Async components are **”suspensible”** by default. This means that if it has a <Suspense> in the parent chain, it will be treated as an async dependency of that <Suspense>. In this case, the loading state will be controlled by the <Suspense>, and the component's own loading, error, delay and timeout options will be ignored.

The async component can opt-out of Suspense control and let the component always control its own loading state by specifying `suspensible: false` in its options.

5.5.2 Loading State

The <Suspense> component has two slots: #default and #fallback. Both slots only allow for **one** immediate child node. The node in the default slot is shown if possible. If not, the node in the fallback slot will be shown instead.

```html
<Suspense>
  <!-- 具有深层异步依赖的组件 -->
  <Dashboard />
  <!-- 在 #fallback 插槽中显示 “正在加载中” -->
  <template #fallback>
    Loading...
  </template>
</Suspense>
```

On initial render, <Suspense> will render its default slot content in memory. If any async dependencies are encountered during the process, it will enter a **pending** state. During the pending state, the fallback content will be displayed. When all encountered async dependencies have been resolved, <Suspense> enters a **resolved** state and the resolved default slot content is displayed.

If no async dependencies were encountered during the initial render, <Suspense> will directly go into a resolved state.

Once in a resolved state, <Suspense> will only revert to a pending state if the root node of the #default slot is replaced. New async dependencies nested deeper in the tree will **not** cause the <Suspense> to revert to a pending state.

When a revert happens, fallback content will not be immediately displayed. Instead, <Suspense>

异步组件

异步组件默认就是 **“suspensible”** 的。这意味着如果组件关系链上有一个 <Suspense>，那么这个异步组件就会被当作这个 <Suspense> 的一个异步依赖。在这种情况下，加载状态是由 <Suspense> 控制，而该组件自己的加载、报错、延时和超时等选项都将被忽略。

异步组件也可以通过在选项中指定 `suspensible: false` 表明不用 Suspense 控制，并让组件始终自己控制其加载状态。

5.5.2 加载中状态

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

```html
<Suspense>
  <!-- 具有深层异步依赖的组件 -->
  <Dashboard />
  <!-- 在 #fallback 插槽中显示 “正在加载中” -->
  <template #fallback>
    Loading...
  </template>
</Suspense>
```

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

如果在初次渲染时没有遇到异步依赖，<Suspense> 会直接进入完成状态。

进入完成状态后，只有当默认插槽的根节点被替换时，<Suspense> 才会回到挂起状态。组件树中新的更深层次的异步依赖**不会**造成 <Suspense> 回退到挂起状态。

发生回退时，后备内容不会立即展示出来。相反，<Suspense> 在等待新内容和异步

will display the previous #default content while waiting for the new content and its async dependencies to be resolved. This behavior can be configured with the timeout prop: <Suspense> will switch to fallback content if it takes longer than timeout to render the new default content. A timeout value of 0 will cause the fallback content to be displayed immediately when default content is replaced.

5.5.3 Events

The <Suspense> component emits 3 events: pending, resolve and fallback. The pending event occurs when entering a pending state. The resolve event is emitted when new content has finished resolving in the default slot. The fallback event is fired when the contents of the fallback slot are shown.

The events could be used, for example, to show a loading indicator in front of the old DOM while new components are loading.

5.5.4 Error Handling

<Suspense> currently does not provide error handling via the component itself - however, you can use the errorCaptured option or the onErrorCaptured() hook to capture and handle async errors in the parent component of <Suspense>.

5.5.5 Combining with Other Components

It is common to want to use <Suspense> in combination with the “ and “ components. The nesting order of these components is important to get them all working correctly.

In addition, these components are often used in conjunction with the <RouterView> component from Vue Router.

The following example shows how to nest these components so that they all behave as expected. For simpler combinations you can remove the components that you don’t need:

```html
<RouterView v-slot="{ Component }">
  <template v-if="Component">
    <Transition mode="out-in">
```

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

5.5.3 事件

<Suspense> 组件会触发三个事件：pending、resolve 和 fallback。pending 事件是在进入挂起状态时触发。resolve 事件是在 default 插槽完成获取新内容时触发。fallback 事件则是在 fallback 插槽的内容显示时触发。

例如，可以使用这些事件在加载新组件时在之前的 DOM 最上层显示一个加载指示器。

5.5.4 错误处理

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

5.5.5 和其他组件结合

我们常常会将 <Suspense> 和 “、“ 等组件结合。要保证这些组件都能正常工作，嵌套的顺序非常重要。

另外，这些组件都通常与 Vue Router 中的 <RouterView> 组件结合使用。

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

```html
<RouterView v-slot="{ Component }">
  <template v-if="Component">
    <Transition mode="out-in">
```

```
      <KeepAlive>
        <Suspense>
          <!-- 主要内容 -->
          <component :is="Component"></component>
          <!-- 加载中状态 -->
          <template #fallback>
            正在加载...
          </template>
        </Suspense>
      </KeepAlive>
    </Transition>
  </template>
</RouterView>
```

Vue Router has built-in support for lazily loading components using dynamic imports. These are distinct from async components and currently they will not trigger `<Suspense>`. However, they can still have async components as descendants and those can trigger `<Suspense>` in the usual way.

```
      <KeepAlive>
        <Suspense>
          <!-- 主要内容 -->
          <component :is="Component"></component>
          <!-- 加载中状态 -->
          <template #fallback>
            正在加载...
          </template>
        </Suspense>
      </KeepAlive>
    </Transition>
  </template>
</RouterView>
```

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

第六章 Scaling Up

6.1 Single-File Components

6.1.1 Introduction

Vue Single-File Components (a.k.a. `*.vue` files, abbreviated as **SFC**) is a special file format that allows us to encapsulate the template, logic, **and** styling of a Vue component in a single file. Here's an example SFC:

```html
<script setup>
import { ref } from 'vue'
const greeting = ref('Hello World!')
</script>
<template>
  <p class="greeting">{{ greeting }}</p>
</template>
<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

As we can see, Vue SFC is a natural extension of the classic trio of HTML, CSS and JavaScript. The `<template>`, `<script>`, and `<style>` blocks encapsulate and colocate the view, logic and styling of a component in the same file. The full syntax is defined in the SFC Syntax Specification.

应用规模化

6.1 单文件组件

6.1.1 介绍

Vue 的单文件组件 (即 `*.vue` 文件，英文 Single-File Component，简称 **SFC**) 是一种特殊的文件格式，使我们能够将一个 Vue 组件的模板、逻辑与样式封装在单个文件中。下面是一个单文件组件的示例：

```html
<script setup>
import { ref } from 'vue'
const greeting = ref('Hello World!')
</script>
<template>
  <p class="greeting">{{ greeting }}</p>
</template>
<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

如你所见，Vue 的单文件组件是网页开发中 HTML、CSS 和 JavaScript 三种语言经典组合的自然延伸。`<template>`、`<script>` 和 `<style>` 三个块在同一个文件中封装、组合了组件的视图、逻辑和样式。完整的语法定义可以查阅 SFC 语法说明。

6.1.2 Why SFC

While SFCs require a build step, there are numerous benefits in return:

- Author modularized components using familiar HTML, CSS and JavaScript syntax
- Colocation of inherently coupled concerns
- Pre-compiled templates without runtime compilation cost
- Component-scoped CSS
- More ergonomic syntax when working with Composition API
- More compile-time optimizations by cross-analyzing template and script
- IDE support with auto-completion and type-checking for template expressions
- Out-of-the-box Hot-Module Replacement (HMR) support

SFC is a defining feature of Vue as a framework, and is the recommended approach for using Vue in the following scenarios:

- Single-Page Applications (SPA)
- Static Site Generation (SSG)
- Any non-trivial frontend where a build step can be justified for better development experience (DX).

That said, we do realize there are scenarios where SFCs can feel like overkill. This is why Vue can still be used via plain JavaScript without a build step. If you are just looking for enhancing largely static HTML with light interactions, you can also check out petite-vue, a 6 kB subset of Vue optimized for progressive enhancement.

6.1.3 How It Works

Vue SFC is a framework-specific file format and must be pre-compiled by @vue/compiler-sfc into standard JavaScript and CSS. A compiled SFC is a standard JavaScript (ES) module - which means with proper build setup you can import an SFC like a module:

6.1.2 为什么要使用 SFC

使用 SFC 必须使用构建工具，但作为回报带来了以下优点：

- 使用熟悉的 HTML、CSS 和 JavaScript 语法编写模块化的组件
- 让本来就强相关的关注点自然内聚
- 预编译模板，避免运行时的编译开销
- 组件作用域的 CSS
- 在使用组合式 API 时语法更简单
- 通过交叉分析模板和逻辑代码能进行更多编译时优化
- 更好的 IDE 支持，提供自动补全和对模板中表达式的类型检查
- 开箱即用的模块热更新 (HMR) 支持

SFC 是 Vue 框架提供的一个功能，并且在下列场景中都是官方推荐的项目组织方式：

- 单页面应用 (SPA)
- 静态站点生成 (SSG)
- 任何值得引入构建步骤以获得更好的开发体验 (DX) 的项目

当然，在一些轻量级场景下使用 SFC 会显得有些杀鸡用牛刀。因此 Vue 同样也可以在无构建步骤的情况下以纯 JavaScript 方式使用。如果你的用例只需要给静态 HTML 添加一些简单的交互，你可以看看 petite-vue，它是一个 6 kB 左右、预优化过的 Vue 子集，更适合渐进式增强的需求。

6.1.3 SFC 是如何工作的

Vue SFC 是一个框架指定的文件格式，因此必须交由 @vue/compiler-sfc 编译为标准的 JavaScript 和 CSS，一个编译后的 SFC 是一个标准的 JavaScript(ES) 模块，这也意味着在构建配置正确的前提下，你可以像导入其他 ES 模块一样导入 SFC：

js
```js
import MyComponent from './MyComponent.vue'
export default {
  components: {
    MyComponent
  }
}
```

 tags inside SFCs are typically injected as native <style> tags during development to support hot updates. For production they can be extracted and merged into a single CSS file.</body>

You can play with SFCs and explore how they are compiled in the Vue SFC Playground.

In actual projects, we typically integrate the SFC compiler with a build tool such as Vite or Vue CLI (which is based on webpack), and Vue provides official scaffolding tools to get you started with SFCs as fast as possible. Check out more details in the SFC Tooling section.

6.1.4 What About Separation of Concerns?

Some users coming from a traditional web development background may have the concern that SFCs are mixing different concerns in the same place - which HTML/CSS/JS were supposed to separate!

To answer this question, it is important for us to agree that **separation of concerns is not equal to the separation of file types**. The ultimate goal of engineering principles is to improve the maintainability of codebases. Separation of concerns, when applied dogmatically as separation of file types, does not help us reach that goal in the context of increasingly complex frontend applications.

In modern UI development, we have found that instead of dividing the codebase into three huge layers that interweave with one another, it makes much more sense to divide them into loosely-coupled components and compose them. Inside a component, its template, logic, and styles are inherently coupled, and colocating them actually makes the component more cohesive and maintainable.

Note even if you don't like the idea of Single-File Components, you can still leverage its hot-reloading and pre-compilation features by separating your JavaScript and CSS into separate files using Src Imports.

js
```js
import MyComponent from './MyComponent.vue'
export default {
  components: {
    MyComponent
  }
}
```

SFC 中的 `<style>` 标签一般会在开发时注入成原生的 `<style>` 标签以支持热更新，而生产环境下它们会被抽取、合并成单独的 CSS 文件。

你可以在 Vue SFC 演练场中实际使用一下单文件组件，同时可以看到它们最终被编译后的样子。

在实际项目中，我们一般会使用集成了 SFC 编译器的构建工具，比如 Vite 或者 Vue CLI (基于 webpack)，Vue 官方也提供了脚手架工具来帮助你尽可能快速地上手开发 SFC。更多细节请查看 SFC 工具链章节。

6.1.4 如何看待关注点分离?

一些有着传统 Web 开发背景的用户可能会因为 SFC 将不同的关注点集合在一处而有所顾虑，觉得 HTML/CSS/JS 应当是分离开的！

要回答这个问题，我们必须对这一点达成共识：**前端开发的关注点不是完全基于文件类型分离的**。前端工程化的最终目的都是为了能够更好地维护代码。关注点分离不应该是教条式地将其视为文件类型的区别和分离，仅仅这样并不够帮我们在日益复杂的前端应用的背景下提高开发效率。

在现代的 UI 开发中，我们发现与其将代码库划分为三个巨大的层，相互交织在一起，不如将它们划分为松散耦合的组件，再按需组合起来。在一个组件中，其模板、逻辑和样式本就是有内在联系的、是耦合的，将它们放在一起，实际上使组件更有内聚性和可维护性。

即使你不喜欢单文件组件这样的形式而仍然选择拆分单独的 JavaScript 和 CSS 文件，也没关系，你还是可以通过资源导入功能获得热更新和预编译等功能的支持。

6.2 Tooling

6.2.1 Try It Online

You don't need to install anything on your machine to try out Vue SFCs - there are online playgrounds that allow you to do so right in the browser:

- Vue SFC Playground
  - Always deployed from latest commit
  - Designed for inspecting component compilation results
- Vue + Vite on StackBlitz
  - IDE-like environment running actual Vite dev server in the browser
  - Closest to local setup

It is also recommended to use these online playgrounds to provide reproductions when reporting bugs.

6.2.2 Project Scaffolding

Vite

Vite is a lightweight and fast build tool with first-class Vue SFC support. It is created by Evan You, who is also the author of Vue!

To get started with Vite + Vue, simply run:

```
npm create vue@latest
```

This command will install and execute create-vue, the official Vue project scaffolding tool.

- To learn more about Vite, check out the Vite docs.
- To configure Vue-specific behavior in a Vite project, for example passing options to the Vue compiler, check out the docs for @vitejs/plugin-vue.

Both online playgrounds mentioned above also support downloading files as a Vite project.

6.2 工具链

6.2.1 在线尝试

你不需要在机器上安装任何东西，也可以尝试基于单文件组件的 Vue 开发体验。我们提供了一个在线的演练场，可以在浏览器中访问：

- Vue SFC 演练场 `https://play.vuejs.org/`
  - 自动随着 Vue 仓库最新的提交更新
  - 支持检查编译输出的结果
- StackBlitz 中的 Vue + Vite `https://vite.new/vue`
  - 类似 IDE 的环境，但实际是在浏览器中运行 Vite 开发服务器
  - 和本地开发效果更接近

在报告 Bug 时，我们也建议使用这些在线演练场来提供最小化重现。

6.2.2 项目脚手架

Vite

Vite 是一个轻量级的、速度极快的构建工具，对 Vue SFC 提供第一优先级支持。作者是尤雨溪，同时也是 Vue 的作者！

要使用 Vite 来创建一个 Vue 项目，非常简单：

```
npm create vue@latest
```

这个命令会安装和执行 create-vue，它是 Vue 提供的官方脚手架工具。跟随命令行的提示继续操作即可。

- 要学习更多关于 Vite 的知识，请查看 Vite 官方文档。
- 若要了解如何为一个 Vite 项目配置 Vue 相关的特殊行为，比如向 Vue 编译器传递相关选项，请查看 @vitejs/plugin-vue 的文档。

上面提到的两种在线演练场也支持将文件作为一个 Vite 项目下载。

Vue CLI

Vue CLI is the official webpack-based toolchain for Vue. It is now in maintenance mode and we recommend starting new projects with Vite unless you rely on specific webpack-only features. Vite will provide superior developer experience in most cases.

For information on migrating from Vue CLI to Vite:

- Vue CLI -> Vite Migration Guide from VueSchool.io
- Tools / Plugins that help with auto migration

Note on In-Browser Template Compilation

When using Vue without a build step, component templates are written either directly in the page's HTML or as inlined JavaScript strings. In such cases, Vue needs to ship the template compiler to the browser in order to perform on-the-fly template compilation. On the other hand, the compiler would be unnecessary if we pre-compile the templates with a build step. To reduce client bundle size, Vue provides different ”builds” optimized for different use cases.

- Build files that start with `vue.runtime.*` are **runtime-only builds**: they do not include the compiler. When using these builds, all templates must be pre-compiled via a build step.
- Build files that do not include `.runtime` are **full builds**: they include the compiler and support compiling templates directly in the browser. However, they will increase the payload by ~14kb.

Our default tooling setups use the runtime-only build since all templates in SFCs are pre-compiled. If, for some reason, you need in-browser template compilation even with a build step, you can do so by configuring the build tool to alias `vue` to `vue/dist/vue.esm-bundler.js` instead.

If you are looking for a lighter-weight alternative for no-build-step usage, check out petite-vue.

6.2.3 IDE Support

- The recommended IDE setup is VSCode + the Vue Language Features (Volar) extension. The extension provides syntax highlighting, TypeScript support, and intellisense for template expressions and component props.

Vue CLI

Vue CLI 是官方提供的基于 Webpack 的 Vue 工具链，它现在处于维护模式。我们建议使用 Vite 开始新的项目，除非你依赖特定的 Webpack 的特性。在大多数情况下，Vite 将提供更优秀的开发体验。

关于从 Vue CLI 迁移到 Vite 的资源：

- VueSchool.io 的 Vue CLI -> Vite 迁移指南
- 迁移支持工具 / 插件

浏览器内模板编译注意事项

当以无构建步骤方式使用 Vue 时，组件模板要么是写在页面的 HTML 中，要么是内联的 JavaScript 字符串。在这些场景中，为了执行动态模板编译，Vue 需要将模板编译器运行在浏览器中。相对的，如果我们使用了构建步骤，由于提前编译了模板，那么就无须再在浏览器中运行了。为了减小打包出的客户端代码体积，Vue 提供了多种格式的 “构建文件”以适配不同场景下的优化需求。

- 前缀为 `vue.runtime.*` 的文件是**只包含运行时的版本**：不包含编译器，当使用这个版本时，所有的模板都必须由构建步骤预先编译。
- 名称中不包含 `.runtime` 的文件则是**完全版**：即包含了编译器，并支持在浏览器中直接编译模板。然而，体积也会因此增长大约 14kb。

默认的工具链中都会使用仅含运行时的版本，因为所有 SFC 中的模板都已经被预编译了。如果因为某些原因，在有构建步骤时，你仍需要浏览器内的模板编译，你可以更改构建工具配置，将 `vue` 改为相应的版本 `vue/dist/vue.esm-bundler.js`。

如果你需要一种更轻量级，不依赖构建步骤的替代方案，也可以看看 petite-vue。

6.2.3 IDE 支持

- 推荐使用的 IDE 是 VSCode，配合 Vue 语言特性 (Volar) 插件。该插件提供了语法高亮、TypeScript 支持，以及模板内表达式与组件 props 的智能提示。

> **TIP**
>
> Volar replaces Vetur, our previous official VSCode extension for Vue 2. If you have Vetur currently installed, make sure to disable it in Vue 3 projects.

- WebStorm also provides great built-in support for Vue SFCs.
- Other IDEs that support the Language Service Protocol (LSP) can also leverage Volar's core functionalities via LSP:
  - Sublime Text support via LSP-Volar.
  - vim / Neovim support via coc-volar.
  - emacs support via lsp-mode

6.2.4 Browser Devtools

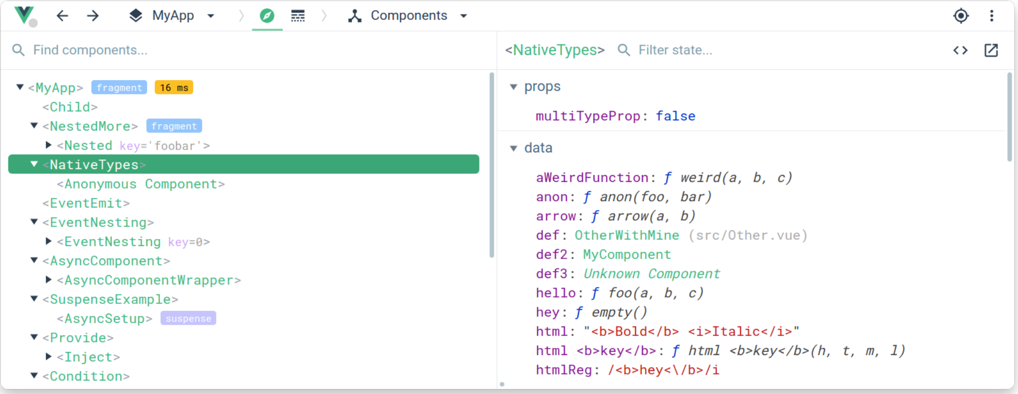
The Vue browser devtools extension allows you to explore a Vue app's component tree, inspect the state of individual components, track state management events, and profile performance.

> **TIP**
>
> Volar 取代了我们之前为 Vue 2 提供的官方 VSCode 扩展 Vetur。如果你之前已经安装了 Vetur，请确保在 Vue 3 的项目中禁用它。

- WebStorm 同样也为 Vue 的单文件组件提供了很好的内置支持。
- 其他支持语言服务协议 (LSP) 的 IDE 也可以通过 LSP 享受到 Volar 所提供的核心功能：
  - Sublime Text 通过 LSP-Volar 支持。
  - vim / Neovim 通过 coc-volar 支持。
  - emacs 通过 lsp-mode 支持。

6.2.4 浏览器开发者插件

Vue 的浏览器开发者插件使我们可以浏览一个 Vue 应用的组件树，查看各个组件的状态，追踪状态管理的事件，还可以进行组件性能分析。



```
▼ <MyApp> fragment 16 ms
    <Child>
  ▼ <NestedMore> fragment
    ▶ <Nested key='foobar'>
  ▼ <NativeTypes>
      <Anonymous Component>
    <EventEmit>
  ▼ <EventNesting>
    ▶ <EventNesting key=0>
  ▼ <AsyncComponent>
    ▶ <AsyncComponentWrapper>
  ▼ <SuspenseExample>
      <AsyncSetup> suspense
  ▼ <Provide>
    ▶ <Inject>
  ▼ <Condition>
```

```
<NativeTypes>
▼ props
    multiTypeProp: false
▼ data
    aWeirdFunction: f weird(a, b, c)
    anon: f anon(foo, bar)
    arrow: f arrow(a, b)
    def: OtherWithMine (src/Other.vue)
    def2: MyComponent
    def3: Unknown Component
    hello: f foo(a, b, c)
    hey: f empty()
    html: "<b>Bold</b> <i>Italic</i>"
    html <b>key</b>: f html <b>key</b>(h, t, m, l)
    htmlReg: /<b>hey<\/b>/i
```

- Documentation
- Chrome Extension
- Firefox Addon
- Edge Extension
- Standalone Electron app

- 文档
- Chrome 扩展商店页
- Firefox 所属插件页
- Edge 扩展
- 独立的 Electron 应用所属插件

6.2.5 TypeScript

Main article: Using Vue with TypeScript.

- Volar provides type checking for SFCs using `<script lang="ts">` blocks, including template expressions and cross-component props validation.
- Use `vue-tsc` for performing the same type checking from the command line, or for generating `d.ts` files for SFCs.

6.2.6 Testing

Main article: Testing Guide.

- Cypress is recommended for E2E tests. It can also be used for component testing for Vue SFCs via the Cypress Component Test Runner.
- Vitest is a test runner created by Vue / Vite team members that focuses on speed. It is specifically designed for Vite-based applications to provide the same instant feedback loop for unit / component testing.
- Jest can be made to work with Vite via vite-jest. However, this is only recommended if you have existing Jest-based test suites that you need to migrate over to a Vite-based setup, as Vitest provides similar functionalities with a much more efficient integration.

6.2.7 Linting

The Vue team maintains eslint-plugin-vue, an ESLint plugin that supports SFC-specific linting rules.

Users previously using Vue CLI may be used to having linters configured via webpack loaders. However when using a Vite-based build setup, our general recommendation is:

1. `npm install -D eslint eslint-plugin-vue`, then follow `eslint-plugin-vue`'s configuration guide.
2. Setup ESLint IDE extensions, for example ESLint for VSCode, so you get linter feedback right in your editor during development. This also avoids unnecessary linting cost when starting the dev server.

6.2.5 TypeScript

具体细节请参考章节：配合 TypeScript 使用 Vue。

- Volar 插件能够为 `<script lang="ts">` 块提供类型检查，也能对模板内表达式和组件之间 props 提供自动补全和类型验证。
- 使用 `vue-tsc` 可以在命令行中执行相同的类型检查，通常用来生成单文件组件的 `d.ts` 文件。

6.2.6 测试

具体细节请参考章节：测试指南。

- Cypress 推荐用于 E2E 测试。也可以通过 Cypress 组件测试运行器来给 Vue SFC 作单文件组件测试。
- Vitest 是一个追求更快运行速度的测试运行器，由 Vue / Vite 团队成员开发。主要针对基于 Vite 的应用设计，可以为组件提供即时响应的测试反馈。
- Jest 可以通过 vite-jest 配合 Vite 使用。不过只推荐在你已经有一套基于 Jest 的测试集、且想要迁移到基于 Vite 的开发配置时使用，因为 Vitest 也能够提供类似的功能，且后者与 Vite 的集成更方便高效。

6.2.7 代码规范

Vue 团队维护着 eslint-plugin-vue 项目，它是一个 ESLint 插件，会提供 SFC 相关规则的定义。

之前使用 Vue CLI 的用户可能习惯于通过 webpack loader 来配置规范检查器。然而，若基于 Vite 构建，我们一般推荐：

1. `npm install -D eslint eslint-plugin-vue`,然后遵照 `eslint-plugin-vue` 的指引进行配置。
2. 启用 ESLint IDE 插件，比如 ESLint for VSCode，然后你就可以在开发时获得规范检查器的反馈。这同时也避免了启动开发服务器时不必要的规范检查。

3. Run ESLint as part of the production build command, so you get full linter feedback before shipping to production.
4. (Optional) Setup tools like lint-staged to automatically lint modified files on git commit.

6.2.8 Formatting

- The Volar VSCode extension provides formatting for Vue SFCs out of the box.
- Alternatively, Prettier provides built-in Vue SFC formatting support.

6.2.9 SFC Custom Block Integrations

Custom blocks are compiled into imports to the same Vue file with different request queries. It is up to the underlying build tool to handle these import requests.

- If using Vite, a custom Vite plugin should be used to transform matched custom blocks into executable JavaScript. Example
- If using Vue CLI or plain webpack, a webpack loader should be configured to transform the matched blocks. Example

6.2.10 Lower-Level Packages

@vue/compiler-sfc

- Docs

This package is part of the Vue core monorepo and is always published with the same version as the main `vue` package. It is included as a dependency of the main `vue` package and proxied under `vue/compiler-sfc` so you don't need to install it individually.

The package itself provides lower-level utilities for processing Vue SFCs and is only meant for tooling authors that need to support Vue SFCs in custom tools.

3. 将 ESLint 格式检查作为一个生产构建的步骤，保证你可以在最终打包时获得完整的规范检查反馈。
4. (可选) 启用类似 lint-staged 一类的工具在 git commit 提交时自动执行规范检查。

6.2.8 格式化

- Volar VSCode 插件为 Vue SFC 提供了开箱即用的格式化功能。
- 除此之外，Prettier 也提供了内置的 Vue SFC 格式化支持。

6.2.9 SFC 自定义块集成

自定义块被编译成导入到同一 Vue 文件的不同请求查询。这取决于底层构建工具如何处理这类导入请求。

- 如果使用 Vite,需使用一个自定义 Vite 插件将自定义块转换为可执行的 JavaScript 代码。示例。
- 如果使用 Vue CLI 或只是 webpack，需要使用一个 loader 来配置如何转换匹配到的自定义块。示例。

6.2.10 底层库

@vue/compiler-sfc

- 文档

这个包是 Vue 核心 monorepo 的一部分，并始终和 `vue` 主包版本号保持一致。它已经成为 `vue` 主包的一个依赖并代理到了 `vue/compiler-sfc` 目录下，因此你无需单独安装它。

这个包本身提供了处理 Vue SFC 的底层的功能，并只适用于需要支持 Vue SFC 相关工具链的开发者。

> **TIP**
> Always prefer using this package via the `vue/compiler-sfc` deep import since this ensures its version is in sync with the Vue runtime.

@vitejs/plugin-vue

- Docs

Official plugin that provides Vue SFC support in Vite.

vue-loader

- Docs

The official loader that provides Vue SFC support in webpack. If you are using Vue CLI, also see docs on modifying `vue-loader` options in Vue CLI.

6.2.11　Other Online Playgrounds

- VueUse Playground
- Vue + Vite on Repl.it
- Vue on CodeSandbox
- Vue on Codepen
- Vue on Components.studio
- Vue on WebComponents.dev

6.3　Routing

6.3.1　Client-Side vs. Server-Side Routing

Routing on the server side means the server sending a response based on the URL path that the user is visiting. When we click on a link in a traditional server-rendered web app, the browser receives

> **TIP**
> 请始终选择通过 `vue/compiler-sfc` 的深度导入来使用这个包，因为这样可以确保其与 Vue 运行时版本同步。

@vitejs/plugin-vue

- 文档

为 Vite 提供 Vue SFC 支持的官方插件。

vue-loader

- 文档

为 webpack 提供 Vue SFC 支持的官方 loader。如果你正在使用 Vue CLI，也可以看看如何在 Vue CLI 中更改 `vue-loader` 选项的文档。

6.2.11　其他在线演练场

- VueUse Playground
- Vue + Vite on Repl.it
- Vue on CodeSandbox
- Vue on Codepen
- Vue on Components.studio
- Vue on WebComponents.dev

6.3　路由

6.3.1　客户端 vs. 服务端路由

服务端路由指的是服务器根据用户访问的 URL 路径返回不同的响应结果。当我们在一个传统的服务端渲染的 web 应用中点击一个链接时，浏览器会从服务端获

an HTML response from the server and reloads the entire page with the new HTML.

In a Single-Page Application (SPA), however, the client-side JavaScript can intercept the navigation, dynamically fetch new data, and update the current page without full page reloads. This typically results in a more snappy user experience, especially for use cases that are more like actual ”applications”, where the user is expected to perform many interactions over a long period of time.

In such SPAs, the ”routing” is done on the client side, in the browser. A client-side router is responsible for managing the application’s rendered view using browser APIs such as History API or the `hashchange` event.

6.3.2 Official Router

Watch a Free Video Course on Vue School

Vue is well-suited for building SPAs. For most SPAs, it’s recommended to use the officially-supported Vue Router library. For more details, see Vue Router’s documentation.

6.3.3 Simple Routing from Scratch

If you only need very simple routing and do not wish to involve a full-featured router library, you can do so with Dynamic Components and update the current component state by listening to browser `hashchange` events or using the History API.

Here’s a bare-bone example:

```html
<script setup>
import { ref, computed } from 'vue'
import Home from './Home.vue'
import About from './About.vue'
import NotFound from './NotFound.vue'
const routes = {
  '/': Home,
  '/about': About
}
const currentPath = ref(window.location.hash)
window.addEventListener('hashchange', () => {
```

得全新的 HTML，然后重新加载整个页面。

然而，在单页面应用中，客户端的 JavaScript 可以拦截页面的跳转请求，动态获取新的数据，然后在无需重新加载的情况下更新当前页面。这样通常可以带来更顺滑的用户体验，尤其是在更偏向 “应用” 的场景下，因为这类场景下用户通常会在很长的一段时间中做出多次交互。

在这类单页应用中，“路由” 是在客户端执行的。一个客户端路由器的职责就是利用诸如 History API 或是 `hashchange` 事件这样的浏览器 API 来管理应用当前应该渲染的视图。

6.3.2 官方路由

在 Vue School 上观看免费的视频课程

Vue 很适合用来构建单页面应用。对于大多数此类应用，都推荐使用官方支持的路由库。要了解更多细节，请查看 Vue Router 的文档。

6.3.3 从头开始实现一个简单的路由

如果你只需要一个简单的页面路由，而不想为此引入一整个路由库，你可以通过动态组件的方式，监听浏览器 `hashchange` 事件或使用 History API 来更新当前组件。

下面是一个简单的例子：

```html
<script setup>
import { ref, computed } from 'vue'
import Home from './Home.vue'
import About from './About.vue'
import NotFound from './NotFound.vue'
const routes = {
  '/': Home,
  '/about': About
}
const currentPath = ref(window.location.hash)
window.addEventListener('hashchange', () => {
  currentPath.value = window.location.hash
})
const currentView = computed(() => {
  return routes[currentPath.value.slice(1) || '/'] || NotFound
})
</script>
<template>
  <a href="#/">Home</a> |
  <a href="#/about">About</a> |
  <a href="#/non-existent-path">Broken Link</a>
  <component :is="currentView" />
</template>
```

Try it in the Playground

6.4 State Management

6.4.1 What is State Management?

Technically, every Vue component instance already ”manages” its own reactive state. Take a simple counter component as an example:

```html
<script setup>
import { ref } from 'vue'
// 状态
const count = ref(0)
// 动作
function increment() {
  count.value++
}
</script>
<!-- 视图 -->
<template>{{ count }}</template>
```

It is a self-contained unit with the following parts:

```
  currentPath.value = window.location.hash
})
const currentView = computed(() => {
  return routes[currentPath.value.slice(1) || '/'] || NotFound
})
</script>
<template>
  <a href="#/">Home</a> |
  <a href="#/about">About</a> |
  <a href="#/non-existent-path">Broken Link</a>
  <component :is="currentView" />
</template>
```

在演练场中尝试一下

6.4 状态管理

6.4.1 什么是状态管理?

理论上来说，每一个 Vue 组件实例都已经在 “管理” 它自己的响应式状态了。我们以一个简单的计数器组件为例：

```html
<script setup>
import { ref } from 'vue'
// 状态
const count = ref(0)
// 动作
function increment() {
  count.value++
}
</script>
<!-- 视图 -->
<template>{{ count }}</template>
```

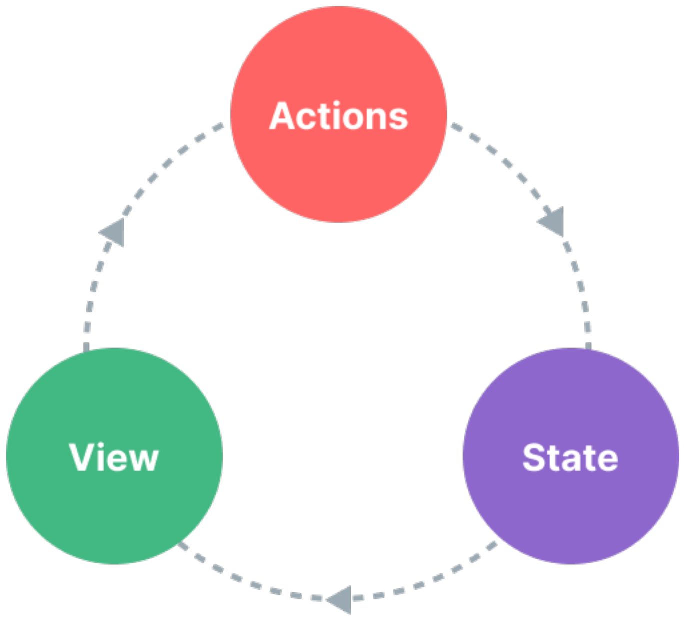
它是一个独立的单元，由以下几个部分组成：

- The **state**, the source of truth that drives our app;
- The **view**, a declarative mapping of the **state**;
- The **actions**, the possible ways the state could change in reaction to user inputs from the **view**.

This is a simple representation of the concept of ”one-way data flow”:

- **状态**：驱动整个应用的数据源；
- **视图**：对**状态**的一种声明式映射；
- **交互**：状态根据用户在**视图**中的输入而作出相应变更的可能方式。

下面是 “单向数据流” 这一概念的简单图示：



However, the simplicity starts to break down when we have **multiple components that share a common state**:

1. Multiple views may depend on the same piece of state.
2. Actions from different views may need to mutate the same piece of state.

For case one, a possible workaround is by ”lifting” the shared state up to a common ancestor component, and then pass it down as props. However, this quickly gets tedious in component trees with deep hierarchies, leading to another problem known as Prop Drilling.

For case two, we often find ourselves resorting to solutions such as reaching for direct parent / child instances via template refs, or trying to mutate and synchronize multiple copies of the state via emitted events. Both of these patterns are brittle and quickly lead to unmaintainable code.

A simpler and more straightforward solution is to extract the shared state out of the components, and manage it in a global singleton. With this, our component tree becomes a big ”view”, and any

然而，当我们有**多个组件共享一个共同的状态**时，就没有这么简单了：

1. 多个视图可能都依赖于同一份状态。
2. 来自不同视图的交互也可能需要更改同一份状态。

对于情景 1，一个可行的办法是将共享状态 “提升” 到共同的祖先组件上去，再通过 props 传递下来。然而在深层次的组件树结构中这么做的话，很快就会使得代码变得繁琐冗长。这会导致另一个问题：Prop 逐级透传问题。

对于情景 2，我们经常发现自己会直接通过模板引用获取父/子实例，或者通过触发的事件尝试改变和同步多个状态的副本。但这些模式的健壮性都不甚理想，很容易就会导致代码难以维护。

一个更简单直接的解决方案是抽取出组件间的共享状态，放在一个全局单例中来管理。这样我们的组件树就变成了一个大的 “视图”，而任何位置上的组件都可以

component can access the state or trigger actions, no matter where they are in the tree!

6.4.2 Simple State Management with Reactivity API

If you have a piece of state that should be shared by multiple instances, you can use `reactive()` to create a reactive object, and then import it into multiple components:

```js
// store.js
import { reactive } from 'vue'
export const store = reactive({
  count: 0
})
```

```html
<!-- ComponentA.vue -->
<script setup>
import { store } from './store.js'
</script>
<template>From A: {{ store.count }}</template>
```

```html
<!-- ComponentB.vue -->
<script setup>
import { store } from './store.js'
</script>
<template>From B: {{ store.count }}</template>
```

Now whenever the `store` object is mutated, both `<ComponentA>` and `<ComponentB>` will update their views automatically - we have a single source of truth now.

However, this also means any component importing `store` can mutate it however they want:

```html
<template>
  <button @click="store.count++">
    From B: {{ store.count }}
  </button>
</template>
```

While this works in simple cases, global state that can be arbitrarily mutated by any component is

访问其中的状态或触发动作。

6.4.2 用响应式 API 做简单状态管理

如果你有一部分状态需要在多个组件实例间共享，你可以使用 `reactive()` 来创建一个响应式对象，并将它导入到多个组件中：

```js
// store.js
import { reactive } from 'vue'
export const store = reactive({
  count: 0
})
```

```html
<!-- ComponentA.vue -->
<script setup>
import { store } from './store.js'
</script>
<template>From A: {{ store.count }}</template>
```

```html
<!-- ComponentB.vue -->
<script setup>
import { store } from './store.js'
</script>
<template>From B: {{ store.count }}</template>
```

现在每当 `store` 对象被更改时，`<ComponentA>` 与 `<ComponentB>` 都会自动更新它们的视图。现在我们有了单一的数据源。

然而，这也意味着任意一个导入了 `store` 的组件都可以随意修改它的状态：

```html
<template>
  <button @click="store.count++">
    From B: {{ store.count }}
  </button>
</template>
```

虽然这在简单的情况下是可行的，但从长远来看，可以被任何组件任意改变的全

not going to be very maintainable in the long run. To ensure the state-mutating logic is centralized like the state itself, it is recommended to define methods on the store with names that express the intention of the actions:

js

```
// store.js
import { reactive } from 'vue'
export const store = reactive({
  count: 0,
  increment() {
    this.count++
  }
})
```

html

```
<template>
  <button @click="store.increment()">
    From B: {{ store.count }}
  </button>
</template>
```

Try it in the Playground

TIP
Note the click handler uses `store.increment()` with parentheses - this is necessary to call the method with the proper `this` context since it's not a component method.

Although here we are using a single reactive object as a store, you can also share reactive state created using other Reactivity APIs such as `ref()` or `computed()`, or even return global state from a Composable:

js

```
import { ref } from 'vue'
// 全局状态，创建在模块作用域下
const globalCount = ref(1)
export function useCount() {
  // 局部状态，每个组件都会创建
  const localCount = ref(1)
  return {
```

局状态是不太容易维护的。为了确保改变状态的逻辑像状态本身一样集中，建议在 store 上定义方法，方法的名称应该要能表达出行动的意图：

js

```
// store.js
import { reactive } from 'vue'
export const store = reactive({
  count: 0,
  increment() {
    this.count++
  }
})
```

html

```
<template>
  <button @click="store.increment()">
    From B: {{ store.count }}
  </button>
</template>
```

在演练场中尝试一下

TIP
请注意这里点击的处理函数使用了 `store.increment()`，带上了圆括号作为内联表达式调用，因为它并不是组件的方法，并且必须要以正确的 `this` 上下文来调用。

除了我们这里用到的单个响应式对象作为一个 store 之外，你还可以使用其他响应式 API 例如 `ref()` 或是 `computed()`，或是甚至通过一个组合式函数来返回一个全局状态：

js

```
import { ref } from 'vue'
// 全局状态，创建在模块作用域下
const globalCount = ref(1)
export function useCount() {
  // 局部状态，每个组件都会创建
  const localCount = ref(1)
  return {
    globalCount,
    localCount
  }
}
```

The fact that Vue's reactivity system is decoupled from the component model makes it extremely flexible.

6.4.3 SSR Considerations

If you are building an application that leverages Server-Side Rendering (SSR), the above pattern can lead to issues due to the store being a singleton shared across multiple requests. This is discussed in more details in the SSR guide.

6.4.4 Pinia

While our hand-rolled state management solution will suffice in simple scenarios, there are many more things to consider in large-scale production applications:

- Stronger conventions for team collaboration
- Integrating with the Vue DevTools, including timeline, in-component inspection, and time-travel debugging
- Hot Module Replacement
- Server-Side Rendering support

Pinia is a state management library that implements all of the above. It is maintained by the Vue core team, and works with both Vue 2 and Vue 3.

Existing users may be familiar with Vuex, the previous official state management library for Vue. With Pinia serving the same role in the ecosystem, Vuex is now in maintenance mode. It still works, but will no longer receive new features. It is recommended to use Pinia for new applications.

Pinia started out as an exploration of what the next iteration of Vuex could look like, incorporating many ideas from core team discussions for Vuex 5. Eventually, we realized that Pinia already implements most of what we wanted in Vuex 5, and decided to make it the new recommendation

```
    globalCount,
    localCount
  }
}
```

事实上，Vue 的响应性系统与组件层是解耦的，这使得它非常灵活。

6.4.3 SSR 相关细节

如果你正在构建一个需要利用服务端渲染 (SSR) 的应用，由于 store 是跨多个请求共享的单例，上述模式可能会导致问题。这在 SSR 指引那一章节会讨论更多细节。

6.4.4 Pinia

虽然我们的手动状态管理解决方案在简单的场景中已经足够了，但是在大规模的生产应用中还有很多其他事项需要考虑：

- 更强的团队协作约定
- 与 Vue DevTools 集成，包括时间轴、组件内部审查和时间旅行调试
- 模块热更新 (HMR)
- 服务端渲染支持

Pinia 就是一个实现了上述需求的状态管理库，由 Vue 核心团队维护，对 Vue 2 和 Vue 3 都可用。

现有用户可能对 Vuex 更熟悉，它是 Vue 之前的官方状态管理库。由于 Pinia 在生态系统中能够承担相同的职责且能做得更好，因此 Vuex 现在处于维护模式。它仍然可以工作，但不再接受新的功能。对于新的应用，建议使用 Pinia。

事实上，Pinia 最初正是为了探索 Vuex 的下一个版本而开发的，因此整合了核心团队关于 Vuex 5 的许多想法。最终，我们意识到 Pinia 已经实现了我们想要在 Vuex 5 中提供的大部分内容，因此决定将其作为新的官方推荐。

instead.

Compared to Vuex, Pinia provides a simpler API with less ceremony, offers Composition-API-style APIs, and most importantly, has solid type inference support when used with TypeScript.

6.5 Testing

6.5.1 Why Test?

Automated tests help you and your team build complex Vue applications quickly and confidently by preventing regressions and encouraging you to break apart your application into testable functions, modules, classes, and components. As with any application, your new Vue app can break in many ways, and it's important that you can catch these issues and fix them before releasing.

In this guide, we'll cover basic terminology and provide our recommendations on which tools to choose for your Vue 3 application.

There is one Vue-specific section covering composables. See Testing Composables below for more details.

6.5.2 When to Test

Start testing early! We recommend you begin writing tests as soon as you can. The longer you wait to add tests to your application, the more dependencies your application will have, and the harder it will be to start.

6.5.3 Testing Types

When designing your Vue application's testing strategy, you should leverage the following testing types:

- **Unit**: Checks that inputs to a given function, class, or composable are producing the expected output or side effects.
- **Component**: Checks that your component mounts, renders, can be interacted with, and behaves as expected. These tests import more code than unit tests, are more complex, and

相比于 Vuex，Pinia 提供了更简洁直接的 API，并提供了组合式风格的 API，最重要的是，在使用 TypeScript 时它提供了更完善的类型推导。

6.5 测试

6.5.1 为什么需要测试

自动化测试能够预防无意引入的 bug，并鼓励开发者将应用分解为可测试、可维护的函数、模块、类和组件。这能够帮助你和你的团队更快速、自信地构建复杂的 Vue 应用。与任何应用一样，新的 Vue 应用可能会以多种方式崩溃，因此，在发布前发现并解决这些问题就变得十分重要。

在本篇指引中，我们将介绍一些基本术语，并就你的 Vue 3 应用应选择哪些工具提供一些建议。

还有一个特定用于 Vue 的小节，介绍了组合式函数的测试，详情请参阅测试组合式函数。

6.5.2 何时测试

越早越好！我们建议你尽快开始编写测试。拖得越久，应用就会有越多的依赖和复杂性，想要开始添加测试也就越困难。

6.5.3 测试的类型

当设计你的 Vue 应用的测试策略时，你应该利用以下几种测试类型：

- **单元测试**：检查给定函数、类或组合式函数的输入是否产生预期的输出或副作用。
- **组件测试**：检查你的组件是否正常挂载和渲染、是否可以与之互动，以及表现是否符合预期。这些测试比单元测试导入了更多的代码，更复杂，需要更

require more time to execute.

- **End-to-end**: Checks features that span multiple pages and makes real network requests against your production-built Vue application. These tests often involve standing up a database or other backend.

Each testing type plays a role in your application's testing strategy, and each will protect you against different types of issues.

6.5.4 Overview

We will briefly discuss what each of these are, how they can be implemented for Vue applications, and provide some general recommendations.

6.5.5 Unit Testing

Unit tests are written to verify that small, isolated units of code are working as expected. A unit test usually covers a single function, class, composable, or module. Unit tests focus on logical correctness and only concern themselves with a small portion of the application's overall functionality. They may mock large parts of your application's environment (e.g. initial state, complex classes, 3rd party modules, and network requests).

In general, unit tests will catch issues with a function's business logic and logical correctness.

Take for example this `increment` function:

js

```js
// helpers.js
export function increment (current, max = 10) {
  if (current < max) {
    return current + 1
  }
  return current
}
```

Because it's very self-contained, it'll be easy to invoke the increment function and assert that it returns what it's supposed to, so we'll write a Unit Test.

If any of these assertions fail, it's clear that the issue is contained within the `increment` function.

多时间来执行。

- **端到端测试**：检查跨越多个页面的功能，并对生产构建的 Vue 应用进行实际的网络请求。这些测试通常涉及到建立一个数据库或其他后端。

每种测试类型在你的应用的测试策略中都发挥着作用，保护你免受不同类型的问题的影响。

6.5.4 总览

我们将简要地讨论这些测试是什么，以及如何在 Vue 应用中实现它们，并提供一些普适性建议。

6.5.5 单元测试

编写单元测试是为了验证小的、独立的代码单元是否按预期工作。一个单元测试通常覆盖一个单个函数、类、组合式函数或模块。单元测试侧重于逻辑上的正确性，只关注应用整体功能的一小部分。他们可能会模拟你的应用环境的很大一部分（如初始状态、复杂的类、第三方模块和网络请求）。

一般来说，单元测试将捕获函数的业务逻辑和逻辑正确性的问题。

以这个 `increment` 函数为例：

js

```js
// helpers.js
export function increment (current, max = 10) {
  if (current < max) {
    return current + 1
  }
  return current
}
```

因为它很独立，可以很容易地调用 `increment` 函数并断言它是否返回了所期望的内容，所以我们将编写一个单元测试。

如果任何一条断言失败了，那么问题一定是出在 `increment` 函数上。

js

```js
// helpers.spec.js
import { increment } from './helpers'
describe('increment', () => {
  test('increments the current number by 1', () => {
    expect(increment(0, 10)).toBe(1)
  })
  test('does not increment the current number over the max', () => {
    expect(increment(10, 10)).toBe(10)
  })
  test('has a default max of 10', () => {
    expect(increment(10)).toBe(10)
  })
})
```

As mentioned previously, unit testing is typically applied to self-contained business logic, components, classes, modules, or functions that do not involve UI rendering, network requests, or other environmental concerns.

These are typically plain JavaScript / TypeScript modules unrelated to Vue. In general, writing unit tests for business logic in Vue applications does not differ significantly from applications using other frameworks.

There are two instances where you DO unit test Vue-specific features:

1. Composables

2. Components

Composables

One category of functions specific to Vue applications is Composables, which may require special handling during tests. See Testing Composables below for more details.

Unit Testing Components

A component can be tested in two ways:

js

```js
// helpers.spec.js
import { increment } from './helpers'
describe('increment', () => {
  test('increments the current number by 1', () => {
    expect(increment(0, 10)).toBe(1)
  })
  test('does not increment the current number over the max', () => {
    expect(increment(10, 10)).toBe(10)
  })
  test('has a default max of 10', () => {
    expect(increment(10)).toBe(10)
  })
})
```

如前所述，单元测试通常适用于独立的业务逻辑、组件、类、模块或函数，不涉及 UI 渲染、网络请求或其他环境问题。

这些通常是与 Vue 无关的纯 JavaScript/TypeScript 模块。一般来说，在 Vue 应用中为业务逻辑编写单元测试与使用其他框架的应用没有明显区别。

但有两种情况，你必须对 Vue 的特定功能进行单元测试：

1. 组合式函数

2. 组件

组合式函数

有一类 Vue 应用中特有的函数被称为 组合式函数，在测试过程中可能需要特殊处理。你可以跳转到下方查看 测试组合式函数 了解更多细节。

组件的单元测试

一个组件可以通过两种方式测试：

1. Whitebox: Unit Testing
   Tests that are ”Whitebox tests” are aware of the implementation details and dependencies of a component. They are focused on **isolating** the component under test. These tests will usually involve mocking some, if not all of your component’s children, as well as setting up plugin state and dependencies (e.g. Pinia).

2. Blackbox: Component Testing
   Tests that are ”Blackbox tests” are unaware of the implementation details of a component. These tests mock as little as possible to test the integration of your component and the entire system. They usually render all child components and are considered more of an ”integration test”. See the Component Testing recommendations below.

Recommendation

- Vitest
  Since the official setup created by `create-vue` is based on Vite, we recommend using a unit testing framework that can leverage the same configuration and transform pipeline directly from Vite. Vitest is a unit testing framework designed specifically for this purpose, created and maintained by Vue / Vite team members. It integrates with Vite-based projects with minimal effort, and is blazing fast.

Other Options

- Peeky is another fast unit test runner with first-class Vite integration. It is also created by a Vue core team member and offers a GUI-based testing interface.

- Jest is a popular unit testing framework, and can be made to work with Vite via the vite-jest package. However, we only recommend Jest if you have an existing Jest test suite that needs to be migrated over to a Vite-based project, as Vitest offers a more seamless integration and better performance.

6.5.6 Component Testing

In Vue applications, components are the main building blocks of the UI. Components are therefore the natural unit of isolation when it comes to validating your application’s behavior. From a granularity perspective, component testing sits somewhere above unit testing and can be considered

1. 白盒：单元测试
   白盒测试知晓一个组件的实现细节和依赖关系。它们更专注于将组件进行更**独立**的测试。这些测试通常会涉及到模拟一些组件的部分子组件，以及设置插件的状态和依赖性（例如 Pinia）。

2. 黑盒：组件测试
   黑盒测试不知晓一个组件的实现细节。这些测试尽可能少地模拟，以测试组件在整个系统中的集成情况。它们通常会渲染所有子组件，因而会被认为更像一种“集成测试”。请查看下方的组件测试建议作进一步了解。

推荐方案

- Vitest
  因为由 `create-vue` 创建的官方项目配置是基于 Vite 的，所以我们推荐你使用一个可以利用同一套 Vite 配置和转换管道的单元测试框架。Vitest 正是一个针对此目标设计的单元测试框架，它由 Vue / Vite 团队成员开发和维护。在 Vite 的项目集成它会非常简单，而且速度非常快。

其他选择

- Peeky 是另一速度极快的单元测试运行器，对 Vite 集成提供第一优先级支持。它也是由 Vue 核心团队成员创建的，并提供了一个基于图形用户界面（GUI）的测试界面。

- Jest 是一个广受欢迎的单元测试框架，并可通过 vite-jest 这个包在 Vite 中使用。不过，我们只推荐你在已有一套 Jest 测试配置、且需要迁移到基于 Vite 的项目时使用它，因为 Vitest 提供了更无缝的集成和更好的性能。

6.5.6 组件测试

在 Vue 应用中，主要用组件来构建用户界面。因此，当验证应用的行为时，组件是一个很自然的独立单元。从粒度的角度来看，组件测试位于单元测试之上，可以被认为是集成测试的一种形式。你的 Vue 应用中大部分内容都应该由组件测试来

a form of integration testing. Much of your Vue Application should be covered by a component test and we recommend that each Vue component has its own spec file.

Component tests should catch issues relating to your component's props, events, slots that it provides, styles, classes, lifecycle hooks, and more.

Component tests should not mock child components, but instead test the interactions between your component and its children by interacting with the components as a user would. For example, a component test should click on an element like a user would instead of programmatically interacting with the component.

Component tests should focus on the component's public interfaces rather than internal implementation details. For most components, the public interface is limited to: events emitted, props, and slots. When testing, remember to **test what a component does, not how it does it**.

DO

- For **Visual** logic: assert correct render output based on inputted props and slots.
- For **Behavioral** logic: assert correct render updates or emitted events in response to user input events.

In the below example, we demonstrate a Stepper component that has a DOM element labeled ”increment” and can be clicked. We pass a prop called `max` that prevents the Stepper from being incremented past 2, so if we click the button 3 times, the UI should still say 2.

We know nothing about the implementation of Stepper, only that the ”input” is the `max` prop and the ”output” is the state of the DOM as the user will see it.

覆盖，我们建议每个 Vue 组件都应有自己的组件测试文件。

组件测试应该捕捉组件中的 prop、事件、提供的插槽、样式、CSS class 名、生命周期钩子，和其他相关的问题。

组件测试不应该模拟子组件，而应该像用户一样，通过与组件互动来测试组件和其子组件之间的交互。例如，组件测试应该像用户那样点击一个元素，而不是编程式地与组件进行交互。

组件测试主要需要关心组件的公开接口而不是内部实现细节。对于大部分的组件来说，公开接口包括触发的事件、prop 和插槽。当进行测试时，请记住，**测试这个组件做了什么，而不是测试它是怎么做到的**。

推荐的做法

- 对于 **视图**的测试：根据输入 prop 和插槽断言渲染输出是否正确。
- 对于 **交互**的测试：断言渲染的更新是否正确或触发的事件是否正确地响应了用户输入事件。

在下面的例子中，我们展示了一个步进器（Stepper）组件，它拥有一个标记为 `increment` 的可点击的 DOM 元素。我们还传入了一个名为 `max` 的 prop 防止步进器增长超过 2，因此如果我们点击了按钮 3 次，视图将仍然显示 2。

我们不了解这个步进器的实现细节，只知道“输入”是这个 `max` prop，“输出”是这个组件状态所呈现出的视图。

Vue Test Utils

Cypress

```
const valueSelector = '[data-testid=stepper-value]'
const buttonSelector = '[data-testid=increment]'

mount(Stepper, {
    props: {
    max: 1
    }
})
```

```
cy.get(valueSelector).should('be.visible').and('contain.text', '0')
    .get(buttonSelector).click()
    .get(valueSelector).should('contain.text', '1')
```

Testing Library

```
const { getByText } = render(Stepper, {
props: {
    max: 1
}
})

getByText('0') // 隐式断言 "0" 在这个组件中

const button = getByText('increment')

// 向我们的增长按钮发送一个点击事件。
await fireEvent.click(button)

getByText('1')

await fireEvent.click(button)
```

DON’T

Don’t assert the private state of a component instance or test the private methods of a component. Testing implementation details makes the tests brittle, as they are more likely to break and require updates when the implementation changes.

The component’s ultimate job is rendering the correct DOM output, so tests focusing on the DOM output provide the same level of correctness assurance (if not more) while being more robust and resilient to change.

Don’t rely exclusively on snapshot tests. Asserting HTML strings does not describe correctness. Write tests with intentionality.

If a method needs to be tested thoroughly, consider extracting it into a standalone utility function and write a dedicated unit test for it. If it cannot be extracted cleanly, it may be tested as a part of a component, integration, or end-to-end test that covers it.

应避免的做法

不要去断言一个组件实例的私有状态或测试一个组件的私有方法。测试实现细节会使测试代码太脆弱，因为当实现发生变化时，它们更有可能失败并需要更新重写。

组件的最终工作是渲染正确的 DOM 输出，所以专注于 DOM 输出的测试提供了足够的正确性保证（如果你不需要更多其他方面测试的话），同时更加健壮、需要的改动更少。

不要完全依赖快照测试。断言 HTML 字符串并不能完全说明正确性。应当编写有意图的测试。

如果一个方法需要测试，把它提取到一个独立的实用函数中，并为它写一个专门的单元测试。如果它不能被直截了当地抽离出来，那么对它的调用应该作为交互测试的一部分。

Recommendation

- Vitest for components or composables that render headlessly (e.g. the `useFavicon` function in VueUse). Components and DOM can be tested using `@vue/test-utils`.
- Cypress Component Testing for components whose expected behavior depends on properly rendering styles or triggering native DOM events. It can be used with Testing Library via @testing-library/cypress.

The main differences between Vitest and browser-based runners are speed and execution context. In short, browser-based runners, like Cypress, can catch issues that node-based runners, like Vitest, cannot (e.g. style issues, real native DOM events, cookies, local storage, and network failures), but browser-based runners are *orders of magnitude slower than Vitest* because they do open a browser, compile your stylesheets, and more. Cypress is a browser-based runner that supports component testing. Please read Vitest's comparison page for the latest information comparing Vitest and Cypress.

Mounting Libraries

Component testing often involves mounting the component being tested in isolation, triggering simulated user input events, and asserting on the rendered DOM output. There are dedicated utility libraries that make these tasks simpler.

- `@vue/test-utils` is the official low-level component testing library that was written to provide users access to Vue specific APIs. It's also the lower-level library `@testing-library/vue` is built on top of.
- `@testing-library/vue` is a Vue testing library focused on testing components without relying on implementation details. Its guiding principle is that the more tests resemble the way software is used, the more confidence they can provide.

We recommend using `@vue/test-utils` for testing components in applications. `@testing-library/vue` has issues with testing asynchronous component with Suspense, so it should be used with caution.

Other Options

- Nightwatch is an E2E test runner with Vue Component Testing support. (Example Project)

推荐方案

- Vitest 对于组件和组合式函数都采用无头渲染的方式 (例如 VueUse 中的 `useFavicon` 函数)。组件和 DOM 都可以通过 @vue/test-utils 来测试。
- Cypress 组件测试 会预期其准确地渲染样式或者触发原生 DOM 事件。可以搭配 @testing-library/cypress 这个库一同进行测试。

Vitest 和基于浏览器的运行器之间的主要区别是速度和执行上下文。简而言之，基于浏览器的运行器，如 Cypress，可以捕捉到基于 Node 的运行器 (如 Vitest) 所不能捕捉的问题 (比如样式问题、原生 DOM 事件、Cookies、本地存储和网络故障)，但基于浏览器的运行器比 Vitest *慢几个数量级*，因为它们要执行打开浏览器，编译样式表以及其他步骤。Cypress 是一个基于浏览器的运行器，支持组件测试。请阅读 Vitest 文档的 “比较” 这一章 了解 Vitest 和 Cypress 最新的比较信息。

组件挂载库

组件测试通常涉及到单独挂载被测试的组件，触发模拟的用户输入事件，并对渲染的 DOM 输出进行断言。有一些专门的工具库可以使这些任务变得更简单。

- `@vue/test-utils` 是官方的底层组件测试库，用来提供给用户访问 Vue 特有的 API。`@testing-library/vue` 也是基于此库构建的。
- `@testing-library/vue` 是一个专注于测试组件而不依赖于实现细节的 Vue 测试库。它的指导原则是：测试越是类似于软件的使用方式，它们就能提供越多的信心。

我们推荐在应用中使用 `@vue/test-utils` 测试组件。`@testing-library/vue` 在测试带有 Suspense 的异步组件时存在问题，在使用时需要谨慎。

其他选择

- Nightwatch 是一个端到端测试运行器，支持 Vue 的组件测试。(Nightwatch v2 版本的 示例项目)

- WebdriverIO for cross-browser component testing that relies on native user interaction based on standardized automation. It can also be used with Testing Library.

6.5.7 E2E Testing

While unit tests provide developers with some degree of confidence, unit and component tests are limited in their abilities to provide holistic coverage of an application when deployed to production. As a result, end-to-end (E2E) tests provide coverage on what is arguably the most important aspect of an application: what happens when users actually use your applications.

End-to-end tests focus on multi-page application behavior that makes network requests against your production-built Vue application. They often involve standing up a database or other backend and may even be run against a live staging environment.

End-to-end tests will often catch issues with your router, state management library, top-level components (e.g. an App or Layout), public assets, or any request handling. As stated above, they catch critical issues that may be impossible to catch with unit tests or component tests.

End-to-end tests do not import any of your Vue application's code but instead rely completely on testing your application by navigating through entire pages in a real browser.

End-to-end tests validate many of the layers in your application. They can either target your locally built application or even a live Staging environment. Testing against your Staging environment not only includes your frontend code and static server but all associated backend services and infrastructure.

> The more your tests resemble how your software is used, the more confidence they can give you. - Kent C. Dodds - Author of the Testing Library

By testing how user actions impact your application, E2E tests are often the key to higher confidence in whether an application is functioning properly or not.

Choosing an E2E Testing Solution

While end-to-end (E2E) testing on the web has gained a negative reputation for unreliable (flaky) tests and slowing down development processes, modern E2E tools have made strides forward to create more reliable, interactive, and useful tests. When choosing an E2E testing framework, the following sections provide some guidance on things to keep in mind when choosing a testing frame-

- WebdriverIO 用于跨浏览器组件测试，该测试依赖于基于标准自动化的原生用户交互。也可以与测试库一起使用。

6.5.7 端到端（E2E）测试

虽然单元测试为所写的代码提供了一定程度的验证，但单元测试和组件测试在部署到生产时，对应用整体覆盖的能力有限。因此，端到端测试针对的可以说是应用最重要的方面：当用户实际使用你的应用时发生了什么。

端到端测试的重点是多页面的应用表现，针对你的应用在生产环境下进行网络请求。他们通常需要建立一个数据库或其他形式的后端，甚至可能针对一个预备上线的环境运行。

端到端测试通常会捕捉到路由、状态管理库、顶级组件（常见为 App 或 Layout）、公共资源或任何请求处理方面的问题。如上所述，它们可以捕捉到单元测试或组件测试无法捕捉的关键问题。

端到端测试不导入任何 Vue 应用的代码，而是完全依靠在真实浏览器中浏览整个页面来测试你的应用。

端到端测试验证了你的应用中的许多层。可以在你的本地构建的应用中，甚至是一个预上线的环境中运行。针对预上线环境的测试不仅包括你的前端代码和静态服务器，还包括所有相关的后端服务和基础设施。

> 你的测试越是类似于你的软件的使用方式，它们就越能值得你信赖。- Kent C. Dodds - Testing Library 的作者

通过测试用户操作如何影响你的应用，端到端测试通常是提高应用能否正常运行的置信度的关键。

选择一个端到端测试解决方案

虽然因为不可靠且拖慢了开发过程，市面上对 Web 上的端到端测试的评价并不好，但现代端到端工具已经在创建更可靠、更有用和交互性更好的测试方面取得了很大进步。在选择端到端测试框架时，以下小节会为你给应用选择测试框架时需要注意的事项提供一些指导。

work for your application.

跨浏览器测试　One of the primary benefits that end-to-end (E2E) testing is known for is its ability to test your application across multiple browsers. While it may seem desirable to have 100% cross-browser coverage, it is important to note that cross browser testing has diminishing returns on a team's resources due to the additional time and machine power required to run them consistently. As a result, it is important to be mindful of this trade-off when choosing the amount of cross-browser testing your application needs.

端到端测试的一个主要优点是你可以了解你的应用在多个不同浏览器上运行的情况。尽管理想情况应该是 100% 的跨浏览器覆盖率，但很重要的一点是跨浏览器测试对团队资源的回报是递减的，因为需要额外的时间和机器来持续运行它们。因此，在选择应用所需的跨浏览器测试的数量时，注意权衡是很有必要的。

更快的反馈　One of the primary problems with end-to-end (E2E) tests and development is that running the entire suite takes a long time. Typically, this is only done in continuous integration and deployment (CI/CD) pipelines. Modern E2E testing frameworks have helped to solve this by adding features like parallelization, which allows for CI/CD pipelines to often run magnitudes faster than before. In addition, when developing locally, the ability to selectively run a single test for the page you are working on while also providing hot reloading of tests can help boost a developer's workflow and productivity.

端到端测试和相应开发过程的主要问题之一是，运行整个套件需要很长的时间。通常情况下，这只在持续集成和部署（CI/CD）管道中进行。现代的端到端测试框架通过增加并行化等功能来帮助解决这个问题，这使得 CI/CD 管道的运行速度比以前快了几倍。此外，在本地开发时，能够有选择地为你正在工作的页面运行单个测试，同时还提供测试的热重载，大大提高了开发者的工作流程和生产力。

第一优先级的调试体验　While developers have traditionally relied on scanning logs in a terminal window to help determine what went wrong in a test, modern end-to-end (E2E) test frameworks allow developers to leverage tools they are already familiar with, e.g. browser developer tools.

传统上，开发者依靠扫描终端窗口中的日志来帮助确定测试中出现的问题，而现代端到端测试框架允许开发者利用他们已经熟悉的工具，例如浏览器开发工具。

无头模式下的可见性　When end-to-end (E2E) tests are run in continuous integration/deployment pipelines, they are often run in headless browsers (i.e., no visible browser is opened for the user to watch). A critical feature of modern E2E testing frameworks is the ability to see snapshots and/or videos of the application during testing, providing some insight into why errors are happening. Historically, it was tedious to maintain these integrations.

当端到端测试在 CI/CD 管道中运行时，它们通常在无头浏览器（即不带界面的浏览器）中运行。因此，当错误发生时，现代端到端测试框架的一个关键特性是能够在不同的测试阶段查看应用的快照、视频，从而深入了解错误的原因。而在很早以前，要手动维护这些集成是非常繁琐的。

Recommendation

- Cypress
  Overall, we believe Cypress provides the most complete E2E solution with features like an informative graphical interface, excellent debuggability, built-in assertions and stubs, flake-resistance, parallelization, and snapshots. As mentioned above, it also provides support for Component Testing. However, it only supports Chromium-based browsers and Firefox.

Other Options

- Playwright is also a great E2E testing solution with a wider range of browser support (mainly WebKit). See Why Playwright for more details.
- Nightwatch is an E2E testing solution based on Selenium WebDriver. This gives it the widest browser support range.
- WebdriverIO is a test automation framework for web and mobile testing based on the WebDriver protocol.

6.5.8 Recipes

Adding Vitest to a Project

In a Vite-based Vue project, run:

```
virhuiai %> npm install -D vitest happy-dom @testing-library/vue
```

Next, update the Vite configuration to add the `test` option block:

js

```js
// vite.config.js
import { defineConfig } from 'vite'
export default defineConfig({
  // ...
  test: {
    // 启用类似 jest 的全局测试 API
```

推荐方案

- Cypress
  总的来说，我们认为 Cypress 提供了最完整的端到端解决方案，其具有信息丰富的图形界面、出色的调试性、内置断言和存根、抗剥落性、并行化和快照等诸多特性。而且如上所述，它还提供对 组件测试 的支持。不过，它只支持测试基于 Chromium 的浏览器和 Firefox。

其他选项

- Playwright 也是一个非常好的端到端测试解决方案，支持测试范围更广的浏览器品类 (主要是 WebKit 型的)。查看这篇文章《为什么选择 Playwright》了解更多细节。
- Nightwatch 是一个基于 Selenium WebDriver 的端到端测试解决方案。它的浏览器品类支持范围是最广的。
- WebdriverIO 是一个基于 WebDriver 协议的网络和移动测试的自动化测试框架。

6.5.8 用例指南

添加 Vitest 到项目中

在一个基于 Vite 的 Vue 项目中，运行如下命令：

```
virhuiai %> npm install -D vitest happy-dom
@testing-library/vue
```

接着，更新你的 Vite 配置，添加上 `test` 选项：

js

```js
// vite.config.js
import { defineConfig } from 'vite'
export default defineConfig({
  // ...
  test: {
    // 启用类似 jest 的全局测试 API
    globals: true,
    // 使用 happy-dom 模拟 DOM
    // 这需要你安装 happy-dom 作为对等依赖（peer dependency）
    environment: 'happy-dom'
  }
})
```

TIP

If you use TypeScript, add `vitest/globals` to the `types` field in your `tsconfig.json`.

css

```
// tsconfig.json
{
  "compilerOptions": {
    "types": ["vitest/globals"]
  }
}
```

Then, create a file ending in `*.test.js` in your project. You can place all test files in a test directory in the project root or in test directories next to your source files. Vitest will automatically search for them using the naming convention.

js

```
// MyComponent.test.js
import { render } from '@testing-library/vue'
import MyComponent from './MyComponent.vue'
test('it should work', () => {
  const { getByText } = render(MyComponent, {
    props: {
      /* ... */
    }
  })
  // 断言输出
  getByText('...')
})
```

Finally, update `package.json` to add the test script and run it:

```
    globals: true,
    // 使用 happy-dom 模拟 DOM
    // 这需要你安装 happy-dom 作为对等依赖（peer dependency）
    environment: 'happy-dom'
  }
})
```

TIP

如果你在使用 TypeScript，请将 `vitest/globals` 添加到 `tsconfig.json` 的 `types` 字段当中。

css

```
// tsconfig.json
{
  "compilerOptions": {
    "types": ["vitest/globals"]
  }
}
```

接着在你的项目中创建名字以 `*.test.js` 结尾的文件。你可以把所有的测试文件放在项目根目录下的 `test` 目录中，或者放在源文件旁边的 `test` 目录中。Vitest 会使用命名规则自动搜索它们。

js

```
// MyComponent.test.js
import { render } from '@testing-library/vue'
import MyComponent from './MyComponent.vue'
test('it should work', () => {
  const { getByText } = render(MyComponent, {
    props: {
      /* ... */
    }
  })
  // 断言输出
  getByText('...')
})
```

最后，在 `package.json` 之中添加测试命令，然后运行它：

```css
{
  // ...
  "scripts": {
    "test": "vitest"
  }
}
```

```
virhuiai %> npm test
```

Testing Composables

> This section assumes you have read the Composables section.

When it comes to testing composables, we can divide them into two categories: composables that do not rely on a host component instance, and composables that do.

A composable depends on a host component instance when it uses the following APIs:

- Lifecycle hooks
- Provide / Inject

If a composable only uses Reactivity APIs, then it can be tested by directly invoking it and asserting its returned state/methods:

```js
// counter.js
import { ref } from 'vue'
export function useCounter() {
  const count = ref(0)
  const increment = () => count.value++
  return {
    count,
    increment
  }
}
```

```js
// counter.test.js
```

```css
{
  // ...
  "scripts": {
    "test": "vitest"
  }
}
```

```
virhuiai %> npm test
```

测试组合式函数

> 这一小节假设你已经读过了组合式函数这一章。

当涉及到测试组合式函数时，我们可以根据是否依赖宿主组件实例把它们分为两类。

当一个组合式函数使用以下 API 时，它依赖于一个宿主组件实例：

- 生命周期钩子
- 供给/注入

如果一个组合式程序只使用响应式 API，那么它可以通过直接调用并断言其返回的状态或方法来进行测试。

```js
// counter.js
import { ref } from 'vue'
export function useCounter() {
  const count = ref(0)
  const increment = () => count.value++
  return {
    count,
    increment
  }
}
```

```js
// counter.test.js
```

```js
import { useCounter } from './counter.js'
test('useCounter', () => {
  const { count, increment } = useCounter()
  expect(count.value).toBe(0)
  increment()
  expect(count.value).toBe(1)
})
```

A composable that relies on lifecycle hooks or Provide / Inject needs to be wrapped in a host component to be tested. We can create a helper like the following:

```js
// test-utils.js
import { createApp } from 'vue'
export function withSetup(composable) {
  let result
  const app = createApp({
    setup() {
      result = composable()
      // 忽略模板警告
      return () => {}
    }
  })
  app.mount(document.createElement('div'))
  // 返回结果与应用实例
  // 用来测试供给和组件卸载
  return [result, app]
}
```

```js
import { withSetup } from './test-utils'
import { useFoo } from './foo'
test('useFoo', () => {
  const [result, app] = withSetup(() => useFoo(123))
  // 为注入的测试模拟一方供给
  app.provide(...)
  // 执行断言
  expect(result.foo.value).toBe(1)
```

```js
import { useCounter } from './counter.js'
test('useCounter', () => {
  const { count, increment } = useCounter()
  expect(count.value).toBe(0)
  increment()
  expect(count.value).toBe(1)
})
```

一个依赖生命周期钩子或供给/注入的组合式函数需要被包装在一个宿主组件中才可以测试。我们可以创建下面这样的帮手函数：

```js
// test-utils.js
import { createApp } from 'vue'
export function withSetup(composable) {
  let result
  const app = createApp({
    setup() {
      result = composable()
      // 忽略模板警告
      return () => {}
    }
  })
  app.mount(document.createElement('div'))
  // 返回结果与应用实例
  // 用来测试供给和组件卸载
  return [result, app]
}
```

```js
import { withSetup } from './test-utils'
import { useFoo } from './foo'
test('useFoo', () => {
  const [result, app] = withSetup(() => useFoo(123))
  // 为注入的测试模拟一方供给
  app.provide(...)
  // 执行断言
  expect(result.foo.value).toBe(1)
  // 如果需要的话可以这样触发
  app.unmount()
})
```

For more complex composables, it could also be easier to test it by writing tests against the wrapper component using Component Testing techniques.

6.6 Server-Side Rendering (SSR)

6.6.1 Overview

What is SSR?

Vue.js is a framework for building client-side applications. By default, Vue components produce and manipulate DOM in the browser as output. However, it is also possible to render the same components into HTML strings on the server, send them directly to the browser, and finally ”hydrate” the static markup into a fully interactive app on the client.

A server-rendered Vue.js app can also be considered ”isomorphic” or ”universal”, in the sense that the majority of your app's code runs on both the server **and** the client.

Why SSR?

Compared to a client-side Single-Page Application (SPA), the advantage of SSR primarily lies in:

- **Faster time-to-content**: this is more prominent on slow internet or slow devices. Server-rendered markup doesn't need to wait until all JavaScript has been downloaded and executed to be displayed, so your user will see a fully-rendered page sooner. In addition, data fetching is done on the server-side for the initial visit, which likely has a faster connection to your database than the client. This generally results in improved Core Web Vitals metrics, better user experience, and can be critical for applications where time-to-content is directly associated with conversion rate.
- **Unified mental model**: you get to use the same language and the same declarative, component-oriented mental model for developing your entire app, instead of jumping back and forth between a backend templating system and a frontend framework.

```
  // 如果需要的话可以这样触发
  app.unmount()
})
```

对于更复杂的组合式函数，通过使用组件测试编写针对这个包装器组件的测试，这会容易很多。

6.6 服务端渲染 (SSR)

6.6.1 总览

什么是 SSR?

Vue.js 是一个用于构建客户端应用的框架。默认情况下，Vue 组件的职责是在浏览器中生成和操作 DOM。然而，Vue 也支持将组件在服务端直接渲染成 HTML 字符串，作为服务端响应返回给浏览器，最后在浏览器端将静态的 HTML“激活”(hydrate) 为能够交互的客户端应用。

一个由服务端渲染的 Vue.js 应用也可以被认为是 “同构的”(Isomorphic) 或 “通用的”(Universal)，因为应用的大部分代码同时运行在服务端**和**客户端。

为什么要用 SSR?

与客户端的单页应用 (SPA) 相比，SSR 的优势主要在于：

- **更快的首屏加载**：这一点在慢网速或者运行缓慢的设备上尤为重要。服务端渲染的 HTML 无需等到所有的 JavaScript 都下载并执行完成之后才显示，所以你的用户将会更快地看到完整渲染的页面。除此之外，数据获取过程在首次访问时在服务端完成，相比于从客户端获取，可能有更快的数据库连接。这通常可以带来更高的核心 Web 指标评分、更好的用户体验，而对于那些 “首屏加载速度与转化率直接相关” 的应用来说，这点可能至关重要。
- **统一的心智模型**：你可以使用相同的语言以及相同的声明式、面向组件的心智模型来开发整个应用，而不需要在后端模板系统和前端框架之间来回切换。
- **更好的 SEO**：搜索引擎爬虫可以直接看到完全渲染的页面。

- **Better SEO**: the search engine crawlers will directly see the fully rendered page.

> **TIP**
> As of now, Google and Bing can index synchronous JavaScript applications just fine. Synchronous being the key word there. If your app starts with a loading spinner, then fetches content via Ajax, the crawler will not wait for you to finish. This means if you have content fetched asynchronously on pages where SEO is important, SSR might be necessary.

There are also some trade-offs to consider when using SSR:

- Development constraints. Browser-specific code can only be used inside certain lifecycle hooks; some external libraries may need special treatment to be able to run in a server-rendered app.
- More involved build setup and deployment requirements. Unlike a fully static SPA that can be deployed on any static file server, a server-rendered app requires an environment where a Node.js server can run.
- More server-side load. Rendering a full app in Node.js is going to be more CPU-intensive than just serving static files, so if you expect high traffic, be prepared for corresponding server load and wisely employ caching strategies.

Before using SSR for your app, the first question you should ask is whether you actually need it. It mostly depends on how important time-to-content is for your app. For example, if you are building an internal dashboard where an extra few hundred milliseconds on initial load doesn't matter that much, SSR would be an overkill. However, in cases where time-to-content is absolutely critical, SSR can help you achieve the best possible initial load performance.

SSR vs. SSG

Static Site Generation (SSG), also referred to as pre-rendering, is another popular technique for building fast websites. If the data needed to server-render a page is the same for every user, then instead of rendering the page every time a request comes in, we can render it only once, ahead of time, during the build process. Pre-rendered pages are generated and served as static HTML files.

SSG retains the same performance characteristics of SSR apps: it provides great time-to-content performance. At the same time, it is cheaper and easier to deploy than SSR apps because the

> **TIP**
> 截至目前，Google 和 Bing 可以很好地对同步 JavaScript 应用进行索引。这里的 “同步” 是关键词。如果你的应用以一个 loading 动画开始，然后通过 Ajax 获取内容，爬虫并不会等到内容加载完成再抓取。也就是说，如果 SEO 对你的页面至关重要，而你的内容又是异步获取的，那么 SSR 可能是必需的。

使用 SSR 时还有一些权衡之处需要考量：

- 开发中的限制。浏览器端特定的代码只能在某些生命周期钩子中使用；一些外部库可能需要特殊处理才能在服务端渲染的应用中运行。
- 更多的与构建配置和部署相关的要求。服务端渲染的应用需要一个能让 Node.js 服务器运行的环境，不像完全静态的 SPA 那样可以部署在任意的静态文件服务器上。
- 更高的服务端负载。在 Node.js 中渲染一个完整的应用要比仅仅托管静态文件更加占用 CPU 资源，因此如果你预期有高流量，请为相应的服务器负载做好准备，并采用合理的缓存策略。

在为你的应用使用 SSR 之前，你首先应该问自己是否真的需要它。这主要取决于首屏加载速度对应用的重要程度。例如，如果你正在开发一个内部的管理面板，初始加载时的那额外几百毫秒对你来说并不重要，这种情况下使用 SSR 就没有太多必要了。然而，在内容展示速度极其重要的场景下，SSR 可以尽可能地帮你实现最优的初始加载性能。

SSR vs. SSG

静态站点生成 (Static-Site Generation，缩写为 SSG)，也被称为预渲染，是另一种流行的构建快速网站的技术。如果用服务端渲染一个页面所需的数据对每个用户来说都是相同的，那么我们可以只渲染一次，提前在构建过程中完成，而不是每次请求进来都重新渲染页面。预渲染的页面生成后作为静态 HTML 文件被服务器托管。

SSG 保留了和 SSR 应用相同的性能表现：它带来了优秀的首屏加载性能。同时，它比 SSR 应用的花销更小，也更容易部署，因为它输出的是静态 HTML 和资源

output is static HTML and assets. The keyword here is **static**: SSG can only be applied to pages consuming static data, i.e. data that is known at build time and does not change between deploys. Every time the data changes, a new deployment is needed.

If you're only investigating SSR to improve the SEO of a handful of marketing pages (e.g. `/`, `/about`, `/contact`, etc.), then you probably want SSG instead of SSR. SSG is also great for content-based websites such as documentation sites or blogs. In fact, this website you are reading right now is statically generated using VitePress, a Vue-powered static site generator.

6.6.2 Basic Tutorial

Rendering an App

Let's take a look at the most bare-bones example of Vue SSR in action.

1. Create a new directory and `cd` into it
2. Run `npm init -y`
3. Add `"type": "module"` in `package.json` so that Node.js runs in ES modules mode.
4. Run `npm install vue`
5. Create an `example.js` file:

```js
// 此文件运行在 Node.js 服务器上
import { createSSRApp } from 'vue'
// Vue 的服务端渲染 API 位于 `vue/server-renderer` 路径下
import { renderToString } from 'vue/server-renderer'
const app = createSSRApp({
  data: () => ({ count: 1 }),
  template: `<button @click="count++">{{ count }}</button>`
})
renderToString(app).then((html) => {
  console.log(html)
})
```

Then run:

文件。这里的关键词是**静态**：SSG 仅可以用于消费静态数据的页面，即数据在构建期间就是已知的，并且在多次部署期间不会改变。每当数据变化时，都需要重新部署。

如果你调研 SSR 只是为了优化为数不多的营销页面的 SEO (例如 `/`、`/about` 和 `/contact` 等)，那么你可能需要 SSG 而不是 SSR。SSG 也非常适合构建基于内容的网站，比如文档站点或者博客。事实上，你现在正在阅读的这个网站就是使用 VitePress 静态生成的，它是一个由 Vue 驱动的静态站点生成器。

6.6.2 基础教程

渲染一个应用

让我们来看一个 Vue SSR 最基础的实战示例。

1. 创建一个新的文件夹，`cd` 进入
2. 执行 `npm init -y`
3. 在 `package.json` 中添加 `"type": "module"` 使 Node.js 以 ES modules mode 运行
4. 执行 `npm install vue`
5. 创建一个 `example.js` 文件：

```js
// 此文件运行在 Node.js 服务器上
import { createSSRApp } from 'vue'
// Vue 的服务端渲染 API 位于 `vue/server-renderer` 路径下
import { renderToString } from 'vue/server-renderer'
const app = createSSRApp({
  data: () => ({ count: 1 }),
  template: `<button @click="count++">{{ count }}</button>`
})
renderToString(app).then((html) => {
  console.log(html)
})
```

接着运行：

```
virhuiai %> node example.js
```

It should print the following to the command line:

`<button>1</button>`

`renderToString()` takes a Vue app instance and returns a Promise that resolves to the rendered HTML of the app. It is also possible to stream rendering using the Node.js Stream API or Web Streams API. Check out the SSR API Reference for full details.

We can then move the Vue SSR code into a server request handler, which wraps the application markup with the full page HTML. We will be using `express` for the next steps:

- Run `npm install express`
- Create the following `server.js` file:

```js
import express from 'express'
import { createSSRApp } from 'vue'
import { renderToString } from 'vue/server-renderer'
const server = express()
server.get('/', (req, res) => {
  const app = createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
  renderToString(app).then((html) => {
    res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>Vue SSR Example</title>
      </head>
      <body>
        <div id="app">${html}</div>
      </body>
    </html>
    `)
```

```
virhuiai %> node example.js
```

它应该会在命令行中打印出如下内容：

`<button>1</button>`

`renderToString()` 接收一个 Vue 应用实例作为参数，返回一个 Promise，当 Promise resolve 时得到应用渲染的 HTML。当然你也可以使用 Node.js Stream API 或者 Web Streams API 来执行流式渲染。查看 SSR API 参考获取完整的相关细节。

然后我们可以把 Vue SSR 的代码移动到一个服务器请求处理函数里，它将应用的 HTML 片段包装为完整的页面 HTML。接下来的几步我们将会使用 `express`：

- 执行 `npm install express`
- 创建下面的 `server.js` 文件：

```js
import express from 'express'
import { createSSRApp } from 'vue'
import { renderToString } from 'vue/server-renderer'
const server = express()
server.get('/', (req, res) => {
  const app = createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
  renderToString(app).then((html) => {
    res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>Vue SSR Example</title>
      </head>
      <body>
        <div id="app">${html}</div>
      </body>
    </html>
    `)
  })
})
server.listen(3000, () => {
  console.log('ready')
})
```

Finally, run `node server.js` and visit `http://localhost:3000`. You should see the page working with the button.

Try it on StackBlitz

Client Hydration

If you click the button, you'll notice the number doesn't change. The HTML is completely static on the client since we are not loading Vue in the browser.

To make the client-side app interactive, Vue needs to perform the **hydration** step. During hydration, it creates the same Vue application that was run on the server, matches each component to the DOM nodes it should control, and attaches DOM event listeners.

To mount an app in hydration mode, we need to use `createSSRApp()` instead of `createApp()`:

```js
// 该文件运行在浏览器中
import { createSSRApp } from 'vue'
const app = createSSRApp({
  // ... 和服务端完全一致的应用实例
})
// 在客户端挂载一个 SSR 应用时会假定
// HTML 是预渲染的，然后执行激活过程，
// 而不是挂载新的 DOM 节点
app.mount('#app')
```

Code Structure

Notice how we need to reuse the same app implementation as on the server. This is where we need to start thinking about code structure in an SSR app - how do we share the same application code between the server and the client?

```
  })
})
server.listen(3000, () => {
  console.log('ready')
})
```

最后，执行 `node server.js`，访问 `http://localhost:3000`。你应该可以看到页面中的按钮了。

在 StackBlitz 上试试

客户端激活

如果你点击该按钮，你会发现数字并没有改变。这段 HTML 在客户端是完全静态的，因为我们没有在浏览器中加载 Vue。

为了使客户端的应用可交互，Vue 需要执行一个**激活**步骤。在激活过程中，Vue 会创建一个与服务端完全相同的应用实例，然后将每个组件与它应该控制的 DOM 节点相匹配，并添加 DOM 事件监听器。

为了在激活模式下挂载应用，我们应该使用 `createSSRApp()` 而不是 `createApp()`：

```js
// 该文件运行在浏览器中
import { createSSRApp } from 'vue'
const app = createSSRApp({
  // ... 和服务端完全一致的应用实例
})
// 在客户端挂载一个 SSR 应用时会假定
// HTML 是预渲染的，然后执行激活过程，
// 而不是挂载新的 DOM 节点
app.mount('#app')
```

代码结构

想想我们该如何在客户端复用服务端的应用实现。这时我们就需要开始考虑 SSR 应用中的代码结构了——我们如何在服务器和客户端之间共享相同的应用代码呢？

Here we will demonstrate the most bare-bones setup. First, let's split the app creation logic into a dedicated file, `app.js`:

js

```
// app.js (在服务器和客户端之间共享)
import { createSSRApp } from 'vue'
export function createApp() {
  return createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
}
```

This file and its dependencies are shared between the server and the client - we call them **universal code**. There are a number of things you need to pay attention to when writing universal code, as we will discuss below.

Our client entry imports the universal code, creates the app, and performs the mount:

js

```
// client.js
import { createApp } from './app.js'
createApp().mount('#app')
```

And the server uses the same app creation logic in the request handler:

js

```
// server.js (不相关的代码省略)
import { createApp } from './app.js'
server.get('/', (req, res) => {
  const app = createApp()
  renderToString(app).then(html => {
    // ...
  })
})
```

In addition, in order to load the client files in the browser, we also need to:

1. Serve client files by adding `server.use(express.static('.'))` in `server.js`.
2. Load the client entry by adding `<script type="module" src="/client.js"></script>` to the HTML shell.

这里我们将演示最基础的设置。首先，让我们将应用的创建逻辑拆分到一个单独的文件 `app.js` 中：

js

```
// app.js (在服务器和客户端之间共享)
import { createSSRApp } from 'vue'
export function createApp() {
  return createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
}
```

该文件及其依赖项在服务器和客户端之间共享——我们称它们为**通用代码**。编写通用代码时有一些注意事项，我们将在下面讨论。

我们在客户端入口导入通用代码，创建应用并执行挂载：

js

```
// client.js
import { createApp } from './app.js'
createApp().mount('#app')
```

服务器在请求处理函数中使用相同的应用创建逻辑：

js

```
// server.js (不相关的代码省略)
import { createApp } from './app.js'
server.get('/', (req, res) => {
  const app = createApp()
  renderToString(app).then(html => {
    // ...
  })
})
```

此外，为了在浏览器中加载客户端文件，我们还需要：

1. 在 `server.js` 中添加 `server.use(express.static('.'))` 来托管客户端文件。
2. 将 `<script type="module" src="/client.js"></script>` 添加到 HTML

3. Support usage like `import * from 'vue'` in the browser by adding an Import Map to the HTML shell.

Try the completed example on StackBlitz. The button is now interactive!

6.6.3 Higher Level Solutions

Moving from the example to a production-ready SSR app involves a lot more. We will need to:

- Support Vue SFCs and other build step requirements. In fact, we will need to coordinate two builds for the same app: one for the client, and one for the server.

  TIP
  Vue components are compiled differently when used for SSR - templates are compiled into string concatenations instead of Virtual DOM render functions for more efficient rendering performance.

- In the server request handler, render the HTML with the correct client-side asset links and optimal resource hints. We may also need to switch between SSR and SSG mode, or even mix both in the same app.
- Manage routing, data fetching, and state management stores in a universal manner.

A complete implementation would be quite complex and depends on the build toolchain you have chosen to work with. Therefore, we highly recommend going with a higher-level, opinionated solution that abstracts away the complexity for you. Below we will introduce a few recommended SSR solutions in the Vue ecosystem.

Nuxt

Nuxt is a higher-level framework built on top of the Vue ecosystem which provides a streamlined development experience for writing universal Vue applications. Better yet, you can also use it as a static site generator! We highly recommend giving it a try.

Quasar

外壳以加载客户端入口文件。

3. 通过在 HTML 外壳中添加 Import Map 以支持在浏览器中使用 `import * from 'vue'`。

在 StackBlitz 上尝试完整的示例。按钮现在可以交互了！

6.6.3 更通用的解决方案

从上面的例子到一个生产就绪的 SSR 应用还需要很多工作。我们将需要：

- 支持 Vue SFC 且满足其他构建步骤要求。事实上，我们需要为同一个应用执行两次构建过程：一次用于客户端，一次用于服务器。

  TIP
  Vue 组件用在 SSR 时的编译产物不同——模板被编译为字符串拼接而不是 render 函数，以此提高渲染性能。

- 在服务器请求处理函数中，确保返回的 HTML 包含正确的客户端资源链接和最优的资源加载提示 (如 prefetch 和 preload)。我们可能还需要在 SSR 和 SSG 模式之间切换，甚至在同一个应用中混合使用这两种模式。
- 以一种通用的方式管理路由、数据获取和状态存储。

完整的实现会非常复杂，并且取决于你选择使用的构建工具链。因此，我们强烈建议你使用一种更通用的、更集成化的解决方案，帮你抽象掉那些复杂的东西。下面推荐几个 Vue 生态中的 SSR 解决方案。

Nuxt

Nuxt 是一个构建于 Vue 生态系统之上的全栈框架，它为编写 Vue SSR 应用提供了丝滑的开发体验。更棒的是，你还可以把它当作一个静态站点生成器来用！我们强烈建议你试一试。

Quasar

Quasar is a complete Vue-based solution that allows you to target SPA, SSR, PWA, mobile app, desktop app, and browser extension all using one codebase. It not only handles the build setup, but also provides a full collection of Material Design compliant UI components.

Vite SSR

Vite provides built-in support for Vue server-side rendering, but it is intentionally low-level. If you wish to go directly with Vite, check out vite-plugin-ssr, a community plugin that abstracts away many challenging details for you.

You can also find an example Vue + Vite SSR project using manual setup here, which can serve as a base to build upon. Note this is only recommended if you are experienced with SSR / build tools and really want to have complete control over the higher-level architecture.

6.6.4 Writing SSR-friendly Code

Regardless of your build setup or higher-level framework choice, there are some principles that apply in all Vue SSR applications.

Reactivity on the Server

During SSR, each request URL maps to a desired state of our application. There is no user interaction and no DOM updates, so reactivity is unnecessary on the server. By default, reactivity is disabled during SSR for better performance.

Component Lifecycle Hooks

Since there are no dynamic updates, lifecycle hooks such as `onMounted` or `onUpdated` will **NOT** be called during SSR and will only be executed on the client.

You should avoid code that produces side effects that need cleanup in `setup()` or the root scope of `<script setup>`. An example of such side effects is setting up timers with `setInterval`. In client-side only code we may setup a timer and then tear it down in `onBeforeUnmount` or `onUnmounted`. However, because the unmount hooks will never be called during SSR, the timers will stay around forever. To avoid this, move your side-effect code into `onMounted` instead.

Quasar 是一个基于 Vue 的完整解决方案，它可以让你用同一套代码库构建不同目标的应用，如 SPA、SSR、PWA、移动端应用、桌面端应用以及浏览器插件。除此之外，它还提供了一整套 Material Design 风格的组件库。

Vite SSR

Vite 提供了内置的 Vue 服务端渲染支持，但它在设计上是偏底层的。如果你想要直接使用 Vite，可以看看 vite-plugin-ssr，一个帮你抽象掉许多复杂细节的社区插件。

你也可以在这里查看一个使用手动配置的 Vue + Vite SSR 的示例项目，以它作为基础来构建。请注意，这种方式只有在你有丰富的 SSR 和构建工具经验，并希望对应用的架构做深入的定制时才推荐使用。

6.6.4 书写 SSR 友好的代码

无论你的构建配置或顶层框架的选择如何，下面的原则在所有 Vue SSR 应用中都适用。

服务端的响应性

在 SSR 期间，每一个请求 URL 都会映射到我们应用中的一个期望状态。因为没有用户交互和 DOM 更新，所以响应性在服务端是不必要的。为了更好的性能，默认情况下响应性在 SSR 期间是禁用的。

组件生命周期钩子

因为没有任何动态更新，所以像 `onMounted` 或者 `onUpdated` 这样的生命周期钩子**不会**在 SSR 期间被调用，而只会在客户端运行。

你应该避免在 `setup()` 或者 `<script setup>` 的根作用域中使用会产生副作用且需要被清理的代码。这类副作用的常见例子是使用 `setInterval` 设置定时器。我们可能会在客户端特有的代码中设置定时器,然后在 `onBeforeUnmount` 或 `onUnmounted` 中清除。然而，由于 unmount 钩子不会在 SSR 期间被调用，所以定时器会永远存在。为了避免这种情况，请将含有副作用的代码放到 `onMounted` 中。

Access to Platform-Specific APIs

Universal code cannot assume access to platform-specific APIs, so if your code directly uses browser-only globals like `window` or `document`, they will throw errors when executed in Node.js, and vice-versa.

For tasks that are shared between server and client but with different platform APIs, it's recommended to wrap the platform-specific implementations inside a universal API, or use libraries that do this for you. For example, you can use `node-fetch` to use the same fetch API on both server and client.

For browser-only APIs, the common approach is to lazily access them inside client-only lifecycle hooks such as `onMounted`.

Note that if a third-party library is not written with universal usage in mind, it could be tricky to integrate it into a server-rendered app. You *might* be able to get it working by mocking some of the globals, but it would be hacky and may interfere with the environment detection code of other libraries.

Cross-Request State Pollution

In the State Management chapter, we introduced a simple state management pattern using Reactivity APIs. In an SSR context, this pattern requires some additional adjustments.

The pattern declares shared state in a JavaScript module's root scope. This makes them **singletons** - i.e. there is only one instance of the reactive object throughout the entire lifecycle of our application. This works as expected in a pure client-side Vue application, since the modules in our application are initialized fresh for each browser page visit.

However, in an SSR context, the application modules are typically initialized only once on the server, when the server boots up. The same module instances will be reused across multiple server requests, and so will our singleton state objects. If we mutate the shared singleton state with data specific to one user, it can be accidentally leaked to a request from another user. We call this **cross-request state pollution.**

We can technically re-initialize all the JavaScript modules on each request, just like we do in browsers. However, initializing JavaScript modules can be costly, so this would significantly affect server performance.

访问平台特有 API

通用代码不能访问平台特有的 API，如果你的代码直接使用了浏览器特有的全局变量，比如 `window` 或 `document`，他们会在 Node.js 运行时报错，反过来也一样。

对于在服务器和客户端之间共享，但使用了不同的平台 API 的任务，建议将平台特定的实现封装在一个通用的 API 中，或者使用能为你做这件事的库。例如你可以使用 `node-fetch` 在服务端和客户端使用相同的 fetch API。

对于浏览器特有的 API，通常的方法是在仅客户端特有的生命周期钩子中惰性地访问它们，例如 `onMounted`。

请注意，如果一个第三方库编写时没有考虑到通用性，那么要将它集成到一个 SSR 应用中可能会很棘手。你*或许*可以通过模拟一些全局变量来让它工作，但这只是一种 hack 手段并且可能会影响到其他库的环境检测代码。

跨请求状态污染

在状态管理一章中，我们介绍了一种使用响应式 API 的简单状态管理模式。而在 SSR 环境中，这种模式需要一些额外的调整。

上述模式在一个 JavaScript 模块的根作用域中声明共享的状态。这是一种**单例模式**——即在应用的整个生命周期中只有一个响应式对象的实例。这在纯客户端的 Vue 应用中是可以的，因为对于浏览器的每一个页面访问，应用模块都会重新初始化。

然而，在 SSR 环境下，应用模块通常只在服务器启动时初始化一次。同一个应用模块会在多个服务器请求之间被复用，而我们的单例状态对象也一样。如果我们用单个用户特定的数据对共享的单例状态进行修改，那么这个状态可能会意外地泄露给另一个用户的请求。我们把这种情况称为**跨请求状态污染**。

从技术上讲，我们可以在每个请求上重新初始化所有 JavaScript 模块，就像我们在浏览器中所做的那样。但是，初始化 JavaScript 模块的成本可能很高，因此这会显著影响服务器性能。

The recommended solution is to create a new instance of the entire application - including the router and global stores - on each request. Then, instead of directly importing it in our components, we provide the shared state using app-level provide and inject it in components that need it:

```js
// app.js （在服务端和客户端间共享）
import { createSSRApp } from 'vue'
import { createStore } from './store.js'
// 每次请求时调用
export function createApp() {
  const app = createSSRApp(/* ... */)
  // 对每个请求都创建新的 store 实例
  const store = createStore(/* ... */)
  // 提供应用级别的 store
  app.provide('store', store)
  // 也为激活过程暴露出 store
  return { app, store }
}
```

State Management libraries like Pinia are designed with this in mind. Consult Pinia's SSR guide for more details.

Hydration Mismatch

If the DOM structure of the pre-rendered HTML does not match the expected output of the client-side app, there will be a hydration mismatch error. Hydration mismatch is most commonly introduced by the following causes:

1. The template contains invalid HTML nesting structure, and the rendered HTML got "corrected" by the browser's native HTML parsing behavior. For example, a common gotcha is that `cannot be placed inside`:

```html
<p><div>hi</div></p>
```

   If we produce this in our server-rendered HTML, the browser will terminate the first `<p>` when `<div>` is encountered and parse it into the following DOM structure:

```html
    <p></p>
```

推荐的解决方案是在每个请求中为整个应用创建一个全新的实例，包括 router 和全局 store。然后，我们使用应用层级的 provide 方法来提供共享状态，并将其注入到需要它的组件中，而不是直接在组件中将其导入：

```js
// app.js （在服务端和客户端间共享）
import { createSSRApp } from 'vue'
import { createStore } from './store.js'
// 每次请求时调用
export function createApp() {
  const app = createSSRApp(/* ... */)
  // 对每个请求都创建新的 store 实例
  const store = createStore(/* ... */)
  // 提供应用级别的 store
  app.provide('store', store)
  // 也为激活过程暴露出 store
  return { app, store }
}
```

像 Pinia 这样的状态管理库在设计时就考虑到了这一点。请参考 Pinia 的 SSR 指南以了解更多细节。

激活不匹配

如果预渲染的 HTML 的 DOM 结构不符合客户端应用的期望，就会出现激活不匹配。最常见的激活不匹配是以下几种原因导致的：

1. 组件模板中存在不符合规范的 HTML 结构，渲染后的 HTML 被浏览器原生的 HTML 解析行为纠正导致不匹配。举例来说，一个常见的错误是 `不能被放在中`：

```html
<p><div>hi</div></p>
```

   如果我们在服务器渲染的 HTML 中出现这样的代码，当遇到 `<div>` 时，浏览器会结束第一个 `<p>`，并解析为以下 DOM 结构：

```html
<p></p>
```

```
    <div>hi</div>
    <p></p>
```

2. The data used during render contains randomly generated values. Since the same application will run twice - once on the server, and once on the client - the random values are not guaranteed to be the same between the two runs. There are two ways to avoid random-value-induced mismatches:

   (a) Use `v-if` + `onMounted` to render the part that depends on random values only on the client. Your framework may also have built-in features to make this easier, for example the `<ClientOnly>` component in VitePress.

   (b) Use a random number generator library that supports generating with seeds, and guarantee the server run and the client run are using the same seed (e.g. by including the seed in serialized state and retrieving it on the client).

3. The server and the client are in different time zones. Sometimes, we may want to convert a timestamp into the user's local time. However, the timezone during the server run and the timezone during the client run are not always the same, and we may not reliably know the user's timezone during the server run. In such cases, the local time conversion should also be performed as a client-only operation.

When Vue encounters a hydration mismatch, it will attempt to automatically recover and adjust the pre-rendered DOM to match the client-side state. This will lead to some rendering performance loss due to incorrect nodes being discarded and new nodes being mounted, but in most cases, the app should continue to work as expected. That said, it is still best to eliminate hydration mismatches during development.

Custom Directives

Since most custom directives involve direct DOM manipulation, they are ignored during SSR. However, if you want to specify how a custom directive should be rendered (i.e. what attributes it should add to the rendered element), you can use the `getSSRProps` directive hook:

js

```js
const myDirective = {
  mounted(el, binding) {
    // 客户端实现:
    // 直接更新 DOM
```

```
<div>hi</div>
<p></p>
```

2. 渲染所用的数据中包含随机生成的值。由于同一个应用会在服务端和客户端执行两次，每次执行生成的随机数都不能保证相同。避免随机数不匹配有两种选择：

   (a) 利用 `v-if` + `onMounted` 让需要用到随机数的模板只在客户端渲染。你所用的上层框架可能也会提供简化这个用例的内置 API，比如 VitePress 的 `<ClientOnly>` 组件。

   (b) 使用一个能够接受随机种子的随机数生成库，并确保服务端和客户端使用同样的随机数种子 (比如把种子包含在序列化的状态中，然后在客户端取回)。

3. 服务端和客户端的时区不一致。有时候我们可能会想要把一个时间转换为用户的当地时间，但在服务端的时区跟用户的时区可能并不一致，我们也并不能可靠的在服务端预先知道用户的时区。这种情况下，当地时间的转换也应该作为纯客户端逻辑去执行。

当 Vue 遇到激活不匹配时，它将尝试自动恢复并调整预渲染的 DOM 以匹配客户端的状态。这将导致一些渲染性能的损失，因为需要丢弃不匹配的节点并渲染新的节点，但大多数情况下，应用应该会如预期一样继续工作。尽管如此，最好还是在开发过程中发现并避免激活不匹配。

自定义指令

因为大多数的自定义指令都包含了对 DOM 的直接操作，所以它们会在 SSR 时被忽略。但如果你想要自己控制一个自定义指令在 SSR 时应该如何被渲染 (即应该在渲染的元素上添加哪些 attribute)，你可以使用 `getSSRProps` 指令钩子：

js

```js
const myDirective = {
  mounted(el, binding) {
    // 客户端实现:
    // 直接更新 DOM
    el.id = binding.value
  },
  getSSRProps(binding) {
    // 服务端实现:
    // 返回需要渲染的 prop
    // getSSRProps 只接收一个 binding 参数
    return {
      id: binding.value
    }
  }
}
```

Teleports

Teleports require special handling during SSR. If the rendered app contains Teleports, the teleported content will not be part of the rendered string. An easier solution is to conditionally render the Teleport on mount.

If you do need to hydrate teleported content, they are exposed under the `teleports` property of the ssr context object:

js

```js
const ctx = {}
const html = await renderToString(app, ctx)
console.log(ctx.teleports) // { '#teleported': 'teleported content' }
```

You need to inject the teleport markup into the correct location in your final page HTML similar to how you need to inject the main app markup.

> **TIP**
> Avoid targeting `body` when using Teleports and SSR together - usually, `<body>` will contain other server-rendered content which makes it impossible for Teleports to determine the correct starting location for hydration.

Instead, prefer a dedicated container, e.g. `<div id="teleported"></div>` which contains only teleported content.

```
    el.id = binding.value
  },
  getSSRProps(binding) {
    // 服务端实现:
    // 返回需要渲染的 prop
    // getSSRProps 只接收一个 binding 参数
    return {
      id: binding.value
    }
  }
}
```

Teleports

在 SSR 的过程中 Teleport 需要特殊处理。如果渲染的应用包含 Teleport，那么其传送的内容将不会包含在主应用渲染出的字符串中。在大多数情况下，更推荐的方案是在客户端挂载时条件式地渲染 Teleport。

如果你需要激活 Teleport 内容,它们会暴露在服务端渲染上下文对象的 `teleports` 属性下：

js

```js
const ctx = {}
const html = await renderToString(app, ctx)
console.log(ctx.teleports) // { '#teleported': 'teleported content' }
```

跟主应用的 HTML 一样，你需要自己将 Teleport 对应的 HTML 嵌入到最终页面上的正确位置处。

> **TIP**
> 请避免在 SSR 的同时把 Teleport 的目标设为 `body`——通常 `<body>` 会包含其他服务端渲染出来的内容，这会使得 Teleport 无法确定激活的正确起始位置。

推荐用一个独立的只包含 teleport 的内容的容器,例如 `<div id="teleported"></div>`。

第七章　Best Practices

7.1　Production Deployment

7.1.1　Development vs. Production

During development, Vue provides a number of features to improve the development experience:

- Warning for common errors and pitfalls
- Props / events validation
- Reactivity debugging hooks
- Devtools integration

However, these features become useless in production. Some of the warning checks can also incur a small amount of performance overhead. When deploying to production, we should drop all the unused, development-only code branches for smaller payload size and better performance.

7.1.2　Without Build Tools

If you are using Vue without a build tool by loading it from a CDN or self-hosted script, make sure to use the production build (dist files that end in `.prod.js`) when deploying to production. Production builds are pre-minified with all development-only code branches removed.

- If using global build (accessing via the `Vue` global): use `vue.global.prod.js`.
- If using ESM build (accessing via native ESM imports): use `vue.esm-browser.prod.js`.

Consult the dist file guide for more details.

最佳实践

7.1　生产部署

7.1.1　开发环境 vs. 生产环境

在开发过程中，Vue 提供了许多功能来提升开发体验：

- 对常见错误和隐患的警告
- 对组件 props / 自定义事件的校验
- 响应性调试钩子
- 开发工具集成

然而，这些功能在生产环境中并不会被使用，一些警告检查也会产生少量的性能开销。当部署到生产环境中时，我们应该移除所有未使用的、仅用于开发环境的代码分支，来获得更小的包体积和更好的性能。

7.1.2　不使用构建工具

如果你没有使用任何构建工具，而是从 CDN 或其他源来加载 Vue，请确保在部署时使用的是生产环境版本（以 `.prod.js` 结尾的构建文件）。生产环境版本会被最小化，并移除了所有仅用于开发环境的代码分支。

- 如果需要使用全局变量版本（通过 `Vue` 全局变量访问）：请使用 `vue.global.prod.js`。
- 如果需要 ESM 版本（通过原生 ESM 导入访问）：请使用 `vue.esm-browser.prod.js`。

更多细节请参考构建文件指南。

7.1.3 With Build Tools

Projects scaffolded via `create-vue` (based on Vite) or Vue CLI (based on webpack) are pre-configured for production builds.

If using a custom setup, make sure that:

1. `vue` resolves to `vue.runtime.esm-bundler.js`.
2. The compile time feature flags are properly configured.
3. `process.env.NODE_ENV` is replaced with `"production"` during build.

Additional references:

- Vite production build guide
- Vite deployment guide
- Vue CLI deployment guide

7.1.4 Tracking Runtime Errors

The app-level error handler can be used to report errors to tracking services:

```js
import { createApp } from 'vue'
const app = createApp(...)
app.config.errorHandler = (err, instance, info) => {
  // 向追踪服务报告错误
}
```

Services such as Sentry and Bugsnag also provide official integrations for Vue.

7.2 Performance

7.2.1 Overview

Vue is designed to be performant for most common use cases without much need for manual optimizations. However, there are always challenging scenarios where extra fine-tuning is needed. In

7.1.3 使用构建工具

通过 `create-vue`（基于 Vite）或是 Vue CLI（基于 webpack）搭建的项目都已经预先做好了针对生产环境的配置。

如果使用了自定义的构建，请确保：

1. `vue` 被解析为 `vue.runtime.esm-bundler.js`。
2. 编译时功能标记已被正确配置。
3. `process.env.NODE_ENV` 会在构建时被替换为 `"production"`。

其他参考：

- Vite 生产环境指南
- Vite 部署指南
- Vue CLI 部署指南

7.1.4 追踪运行时错误

应用级错误处理 可以用来向追踪服务报告错误：

```js
import { createApp } from 'vue'
const app = createApp(...)
app.config.errorHandler = (err, instance, info) => {
  // 向追踪服务报告错误
}
```

诸如 Sentry 和 Bugsnag 等服务也为 Vue 提供了官方集成。

7.2 性能优化

7.2.1 概述

Vue 在大多数常见场景下性能都是很优秀的，通常不需要手动优化。然而，总会有一些具有挑战性的场景需要进行针对性的微调。在本节中，我们将讨论用 Vue 开

this section, we will discuss what you should pay attention to when it comes to performance in a Vue application.

First, let's discuss the two major aspects of web performance:

- **Page Load Performance**: how fast the application shows content and becomes interactive on the initial visit. This is usually measured using web vital metrics like Largest Contentful Paint (LCP) and First Input Delay (FID).
- **Update Performance**: how fast the application updates in response to user input. For example, how fast a list updates when the user types in a search box, or how fast the page switches when the user clicks a navigation link in a Single-Page Application (SPA).

While it would be ideal to maximize both, different frontend architectures tend to affect how easy it is to attain desired performance in these aspects. In addition, the type of application you are building greatly influences what you should prioritize in terms of performance. Therefore, the first step of ensuring optimal performance is picking the right architecture for the type of application you are building:

- Consult Ways of Using Vue to see how you can leverage Vue in different ways.
- Jason Miller discusses the types of web applications and their respective ideal implementation / delivery in Application Holotypes.

7.2.2 Profiling Options

To improve performance, we need to first know how to measure it. There are a number of great tools that can help in this regard:

For profiling load performance of production deployments:

- PageSpeed Insights
- WebPageTest

For profiling performance during local development:

- Chrome DevTools Performance Panel
    - `app.config.performance` enables Vue-specific performance markers in Chrome Dev-

发的应用在性能方面该注意些什么。

首先，让我们区分一下 web 应用性能的两个主要方面：

- **页面加载性能**：首次访问时，应用展示出内容与达到可交互状态的速度。这通常会用 Google 所定义的一系列 Web 指标 (Web Vitals) 来进行衡量，如最大内容绘制 (Largest Contentful Paint，缩写为 LCP) 和首次输入延迟 (First Input Delay，缩写为 FID)。
- **更新性能**：应用响应用户输入更新的速度。比如当用户在搜索框中输入时结果列表的更新速度，或者用户在一个单页面应用 (SPA) 中点击链接跳转页面时的切换速度。

虽然最理想的情况是将两者都最大化，但是不同的前端架构往往会影响到在这些方面是否能达到更理想的性能。此外，你所构建的应用的类型极大地影响了你在性能方面应该优先考虑的问题。因此，优化性能的第一步是为你的应用类型确定合适的架构：

- 查看使用 Vue 的多种方式这一章看看如何用不同的方式围绕 Vue 组织架构。
- Jason Miller 在 Application Holotypes 一文中讨论了 Web 应用的类型以及它们各自的理想实现/交付方式。

7.2.2 分析选项

为了提高性能，我们首先需要知道如何衡量它。在这方面，有一些很棒的工具可以提供帮助：

用于生产部署的负载性能分析：

- PageSpeed Insights
- WebPageTest

用于本地开发期间的性能分析：

- Chrome 开发者工具 “性能” 面板
    - `app.config.performance` 将会开启 Vue 特有的性能标记，标记在 Chrome

Tools' performance timeline.

- Vue DevTools Extension also provides a performance profiling feature.

7.2.3 Page Load Optimizations

There are many framework-agnostic aspects for optimizing page load performance - check out this web.dev guide for a comprehensive round up. Here, we will primarily focus on techniques that are specific to Vue.

Choosing the Right Architecture

If your use case is sensitive to page load performance, avoid shipping it as a pure client-side SPA. You want your server to be directly sending HTML containing the content the users want to see. Pure client-side rendering suffers from slow time-to-content. This can be mitigated with Server-Side Rendering (SSR) or Static Site Generation (SSG). Check out the SSR Guide to learn about performing SSR with Vue. If your app doesn't have rich interactivity requirements, you can also use a traditional backend server to render the HTML and enhance it with Vue on the client.

If your main application has to be an SPA, but has marketing pages (landing, about, blog), ship them separately! Your marketing pages should ideally be deployed as static HTML with minimal JS, by using SSG.

Bundle Size and Tree-shaking

One of the most effective ways to improve page load performance is shipping smaller JavaScript bundles. Here are a few ways to reduce bundle size when using Vue:

- Use a build step if possible.
  - Many of Vue's APIs are "tree-shakable" if bundled via a modern build tool. For example, if you don't use the built-in `<Transition>` component, it won't be included in the final production bundle. Tree-shaking can also remove other unused modules in your source code.
  - When using a build step, templates are pre-compiled so we don't need to ship the Vue compiler to the browser. This saves **14kb** min+gzipped JavaScript and avoids the runtime

开发者工具的性能时间线上。

- Vue 开发者扩展也提供了性能分析的功能。

7.2.3 页面加载优化

页面加载优化有许多跟框架无关的方面 - 这份 web.dev 指南提供了一个全面的总结。这里，我们将主要关注和 Vue 相关的技巧。

选用正确的架构

如果你的用例对页面加载性能很敏感，请避免将其部署为纯客户端的 SPA，而是让服务器直接发送包含用户想要查看的内容的 HTML 代码。纯客户端渲染存在首屏加载缓慢的问题，这可以通过服务器端渲染 (SSR) 或静态站点生成 (SSG) 来缓解。查看 SSR 指南以了解如何使用 Vue 实现 SSR。如果应用对交互性要求不高，你还可以使用传统的后端服务器来渲染 HTML，并在客户端使用 Vue 对其进行增强。

如果你的主应用必须是 SPA，但还有其他的营销相关页面 (落地页、关于页、博客等)，请单独部署这些页面！理想情况下，营销页面应该是包含尽可能少 JS 的静态 HTML，并用 SSG 方式部署。

包体积与 Tree-shaking 优化

一个最有效的提升页面加载速度的方法就是压缩 JavaScript 打包产物的体积。当使用 Vue 时有下面一些办法来减小打包产物体积：

- 尽可能地采用构建步骤
  - 如果使用的是相对现代的打包工具，许多 Vue 的 API 都是可以被 tree-shake 的。举例来说，如果你根本没有使用到内置的 `<Transition>` 组件，它将不会被打包进入最终的产物里。Tree-shaking 也可以移除你源代码中其他未使用到的模块。
  - 当使用了构建步骤时，模板会被预编译，因此我们无须在浏览器中载入 Vue 编译器。这在同样最小化加上 gzip 优化下会相对缩小 **14kb** 并避

compilation cost.

- Be cautious of size when introducing new dependencies! In real-world applications, bloated bundles are most often a result of introducing heavy dependencies without realizing it.
  - If using a build step, prefer dependencies that offer ES module formats and are tree-shaking friendly. For example, prefer `lodash-es` over `lodash`.
  - Check a dependency's size and evaluate whether it is worth the functionality it provides. Note if the dependency is tree-shaking friendly, the actual size increase will depend on the APIs you actually import from it. Tools like bundlejs.com can be used for quick checks, but measuring with your actual build setup will always be the most accurate.
- If you are using Vue primarily for progressive enhancement and prefer to avoid a build step, consider using petite-vue (only **6kb**) instead.

Code Splitting

Code splitting is where a build tool splits the application bundle into multiple smaller chunks, which can then be loaded on demand or in parallel. With proper code splitting, features required at page load can be downloaded immediately, with additional chunks being lazy loaded only when needed, thus improving performance.

Bundlers like Rollup (which Vite is based upon) or webpack can automatically create split chunks by detecting the ESM dynamic import syntax:

js
```js
// lazy.js 及其依赖会被拆分到一个单独的文件中
// 并只在 `loadLazy()` 调用时才加载
function loadLazy() {
  return import('./lazy.js')
}
```

Lazy loading is best used on features that are not immediately needed after initial page load. In Vue applications, this can be used in combination with Vue's Async Component feature to create split chunks for component trees:

js
```js
import { defineAsyncComponent } from 'vue'
// 会为 Foo.vue 及其依赖创建单独的一个块
// 它只会按需加载
```

免运行时的编译开销。

- 在引入新的依赖项时要小心包体积膨胀！在现实的应用中，包体积膨胀通常因为无意识地引入了过重的依赖导致的。
  - 如果使用了构建步骤，应当尽量选择提供 ES 模块格式的依赖，它们对 tree-shaking 更友好。举例来说，选择 `lodash-es` 比 `lodash` 更好。
  - 查看依赖的体积，并评估与其所提供的功能之间的性价比。如果依赖对 tree-shaking 友好，实际增加的体积大小将取决于你从它之中导入的 API。像 bundlejs.com 这样的工具可以用来做快速的检查，但是根据实际的构建设置来评估总是最准确的。
- 如果你只在渐进式增强的场景下使用 Vue，并想要避免使用构建步骤，请考虑使用 petite-vue (只有 **6kb**) 来代替。

代码分割

代码分割是指构建工具将构建后的 JavaScript 包拆分为多个较小的，可以按需或并行加载的文件。通过适当的代码分割，页面加载时需要的功能可以立即下载，而额外的块只在需要时才加载，从而提高性能。

像 Rollup (Vite 就是基于它之上开发的) 或者 webpack 这样的打包工具可以通过分析 ESM 动态导入的语法来自动进行代码分割：

js
```js
// lazy.js 及其依赖会被拆分到一个单独的文件中
// 并只在 `loadLazy()` 调用时才加载
function loadLazy() {
  return import('./lazy.js')
}
```

懒加载对于页面初次加载时的优化帮助极大，它帮助应用暂时略过了那些不是立即需要的功能。在 Vue 应用中，这可以与 Vue 的异步组件搭配使用，为组件树创建分离的代码块：

js
```js
import { defineAsyncComponent } from 'vue'
// 会为 Foo.vue 及其依赖创建单独的一个块
// 它只会按需加载
```

```
//（即该异步组件在页面中被渲染时）
const Foo = defineAsyncComponent(() => import('./Foo.vue'))
```

For applications using Vue Router, it is strongly recommended to use lazy loading for route components. Vue Router has explicit support for lazy loading, separate from `defineAsyncComponent`. See Lazy Loading Routes for more details.

7.2.4 Update Optimizations

Props Stability

In Vue, a child component only updates when at least one of its received props has changed. Consider the following example:

html

```html
<ListItem
  v-for="item in list"
  :id="item.id"
  :active-id="activeId" />
```

Inside the `<ListItem>` component, it uses its `id` and `activeId` props to determine whether it is the currently active item. While this works, the problem is that whenever `activeId` changes, **every** `<ListItem>` in the list has to update!

Ideally, only the items whose active status changed should update. We can achieve that by moving the active status computation into the parent, and make `<ListItem>` directly accept an `active` prop instead:

html

```html
<ListItem
  v-for="item in list"
  :id="item.id"
  :active="item.id === activeId" />
```

Now, for most components the `active` prop will remain the same when `activeId` changes, so they no longer need to update. In general, the idea is keeping the props passed to child components as stable as possible.

```
//（即该异步组件在页面中被渲染时）
const Foo = defineAsyncComponent(() => import('./Foo.vue'))
```

对于使用了 Vue Router 的应用，强烈建议使用异步组件作为路由组件。Vue Router 已经显性地支持了独立于 `defineAsyncComponent` 的懒加载。查看懒加载路由了解更多细节。

7.2.4 更新优化

Props 稳定性

在 Vue 之中，一个子组件只会在其至少一个 props 改变时才会更新。思考以下示例：

html

```html
<ListItem
  v-for="item in list"
  :id="item.id"
  :active-id="activeId" />
```

在 `<ListItem>` 组件中，它使用了 `id` 和 `activeId` 两个 props 来确定它是否是当前活跃的那一项。虽然这是可行的，但问题是每当 `activeId` 更新时，列表中的**每一个** `<ListItem>` 都会跟着更新！

理想情况下，只有活跃状态发生改变的项才应该更新。我们可以将活跃状态比对的逻辑移入父组件来实现这一点，然后让 `<ListItem>` 改为接收一个 `active` prop：

html

```html
<ListItem
  v-for="item in list"
  :id="item.id"
  :active="item.id === activeId" />
```

现在，对于大多数的组件来说，`activeId` 改变时，它们的 `active` prop 都会保持不变，因此它们无需再更新。总结一下，这个技巧的核心思想就是让传给子组件的 props 尽量保持稳定。

v-once

`v-once` is a built-in directive that can be used to render content that relies on runtime data but never needs to update. The entire sub-tree it is used on will be skipped for all future updates. Consult its API reference for more details.

v-memo

`v-memo` is a built-in directive that can be used to conditionally skip the update of large sub-trees or `v-for` lists. Consult its API reference for more details.

7.2.5 General Optimizations

> The following tips affect both page load and update performance.

Virtualize Large Lists

One of the most common performance issues in all frontend applications is rendering large lists. No matter how performant a framework is, rendering a list with thousands of items **will** be slow due to the sheer number of DOM nodes that the browser needs to handle.

However, we don't necessarily have to render all these nodes upfront. In most cases, the user's screen size can display only a small subset of our large list. We can greatly improve the performance with **list virtualization**, the technique of only rendering the items that are currently in or close to the viewport in a large list.

Implementing list virtualization isn't easy, luckily there are existing community libraries that you can directly use:

- vue-virtual-scroller
- vue-virtual-scroll-grid
- vueuc/VVirtualList

Reduce Reactivity Overhead for Large Immutable Structures

Vue's reactivity system is deep by default. While this makes state management intuitive, it does create a certain level of overhead when the data size is large, because every property access triggers

v-once

`v-once` 是一个内置的指令，可以用来渲染依赖运行时数据但无需再更新的内容。它的整个子树都会在未来的更新中被跳过。查看它的 API 参考手册可以了解更多细节。

v-memo

`v-memo` 是一个内置指令，可以用来有条件地跳过某些大型子树或者 `v-for` 列表的更新。查看它的 API 参考手册可以了解更多细节。

7.2.5 通用优化

> 以下技巧能同时改善页面加载和更新性能。

大型虚拟列表

所有的前端应用中最常见的性能问题就是渲染大型列表。无论一个框架性能有多好，渲染成千上万个列表项**都会**变得很慢，因为浏览器需要处理大量的 DOM 节点。

但是，我们并不需要立刻渲染出全部的列表。在大多数场景中，用户的屏幕尺寸只会展示这个巨大列表中的一小部分。我们可以通过**列表虚拟化**来提升性能，这项技术使我们只需要渲染用户视口中能看到的部分。

要实现列表虚拟化并不简单，幸运的是，你可以直接使用现有的社区库：

- vue-virtual-scroller
- vue-virtual-scroll-grid
- vueuc/VVirtualList

减少大型不可变数据的响应性开销

Vue 的响应性系统默认是深度的。虽然这让状态管理变得更直观，但在数据量巨大时，深度响应性也会导致不小的性能负担，因为每个属性访问都将触发代理的依

proxy traps that perform dependency tracking. This typically becomes noticeable when dealing with large arrays of deeply nested objects, where a single render needs to access 100,000+ properties, so it should only affect very specific use cases.

Vue does provide an escape hatch to opt-out of deep reactivity by using `shallowRef()` and `shallowReactive()`. Shallow APIs create state that is reactive only at the root level, and exposes all nested objects untouched. This keeps nested property access fast, with the trade-off being that we must now treat all nested objects as immutable, and updates can only be triggered by replacing the root state:

```js
const shallowArray = shallowRef([
  /* 巨大的列表，里面包含深层的对象 */
])
// 这不会触发更新...
shallowArray.value.push(newObject)
// 这才会触发更新
shallowArray.value = [...shallowArray.value, newObject]
// 这不会触发更新...
shallowArray.value[0].foo = 1
// 这才会触发更新
shallowArray.value = [
  {
    ...shallowArray.value[0],
    foo: 1
  },
  ...shallowArray.value.slice(1)
]
```

Avoid Unnecessary Component Abstractions

Sometimes we may create renderless components or higher-order components (i.e. components that render other components with extra props) for better abstraction or code organization. While there is nothing wrong with this, do keep in mind that component instances are much more expensive than plain DOM nodes, and creating too many of them due to abstraction patterns will incur performance costs.

Note that reducing only a few instances won't have noticeable effect, so don't sweat it if the component is rendered only a few times in the app. The best scenario to consider this optimization is

赖追踪。好在这种性能负担通常只有在处理超大型数组或层级很深的对象时，例如一次渲染需要访问 100,000+ 个属性时，才会变得比较明显。因此，它只会影响少数特定的场景。

Vue 确实也为此提供了一种解决方案，通过使用 `shallowRef()` 和 `shallowReactive()` 来绕开深度响应。浅层式 API 创建的状态只在其顶层是响应式的，对所有深层的对象不会做任何处理。这使得对深层级属性的访问变得更快，但代价是，我们现在必须将所有深层级对象视为不可变的，并且只能通过替换整个根状态来触发更新：

```js
const shallowArray = shallowRef([
  /* 巨大的列表，里面包含深层的对象 */
])
// 这不会触发更新...
shallowArray.value.push(newObject)
// 这才会触发更新
shallowArray.value = [...shallowArray.value, newObject]
// 这不会触发更新...
shallowArray.value[0].foo = 1
// 这才会触发更新
shallowArray.value = [
  {
    ...shallowArray.value[0],
    foo: 1
  },
  ...shallowArray.value.slice(1)
]
```

避免不必要的组件抽象

有些时候我们会去创建无渲染组件或高阶组件 (用来渲染具有额外 props 的其他组件) 来实现更好的抽象或代码组织。虽然这并没有什么问题，但请记住，组件实例比普通 DOM 节点要昂贵得多，而且为了逻辑抽象创建太多组件实例将会导致性能损失。

需要提醒的是，只减少几个组件实例对于性能不会有明显的改善，所以如果一个用于抽象的组件在应用中只会渲染几次，就不用操心去优化它了。考虑这种优化

again in large lists. Imagine a list of 100 items where each item component contains many child components. Removing one unnecessary component abstraction here could result in a reduction of hundreds of component instances.

7.3 Accessibility

Web accessibility (also known as a11y) refers to the practice of creating websites that can be used by anyone — be that a person with a disability, a slow connection, outdated or broken hardware or simply someone in an unfavorable environment. For example, adding subtitles to a video would help both your deaf and hard-of-hearing users and your users who are in a loud environment and can't hear their phone. Similarly, making sure your text isn't too low contrast will help both your low-vision users and your users who are trying to use their phone in bright sunlight.

Ready to start but aren't sure where?

Checkout the Planning and managing web accessibility guide provided by World Wide Web Consortium (W3C)

7.3.1 Skip link

You should add a link at the top of each page that goes directly to the main content area so users can skip content that is repeated on multiple Web pages.

Typically this is done on the top of `App.vue` as it will be the first focusable element on all your pages:

```html
<ul class="skip-links">
  <li>
    <a href="#main" ref="skipLink" class="skip-link">Skip to main content</a>
  </li>
</ul>
```

To hide the link unless it is focused, you can add the following style:

```css
.skip-link {
  white-space: nowrap;
  margin: 1em auto;
```

的最佳场景还是在大型列表中。想象一下一个有 100 项的列表，每项的组件都包含许多子组件。在这里去掉一个不必要的组件抽象，可能会减少数百个组件实例的无谓性能消耗。

7.3 无障碍访问

Web 无障碍访问 (也称为 a11y) 是指创建可供任何人使用的网站的做法——无论是身患某种障碍、通过慢速的网络连接访问、使用老旧或损坏的硬件，还是仅处于某种不方便的环境。例如，在视频中添加字幕可以帮助失聪、有听力障碍或身处嘈杂环境而听不到手机的用户。同样地，确保文字样式没有处于太低的对比度，可以对低视力用户和在明亮的强光下使用手机的用户都有所帮助。

你是否已经准备开始却又无从下手?

请先阅读由万维网联盟 (W3C) 提供的 Web 无障碍访问的规划和管理。

7.3.1 跳过链接

你应该在每个页面的顶部添加一个直接指向主内容区域的链接，这样用户就可以跳过在多个网页上重复的内容。

通常这个链接会放在 `App.vue` 的顶部，这样它就会是所有页面上的第一个可聚焦元素：

```html
<ul class="skip-links">
  <li>
    <a href="#main" ref="skipLink" class="skip-link">Skip to main content</a>
  </li>
</ul>
```

若想在非聚焦状态下隐藏该链接，可以添加以下样式：

```css
.skip-link {
  white-space: nowrap;
  margin: 1em auto;
  top: 0;
  position: fixed;
  left: 50%;
  margin-left: -72px;
  opacity: 0;
}
.skip-link:focus {
  opacity: 1;
  background-color: white;
  padding: 0.5em;
  border: 1px solid black;
}
```

Once a user changes route, bring focus back to the skip link. This can be achieved by calling focus on the skip link's template ref (assuming usage of `vue-router`):

html

```html
<script setup>
import { ref, watch } from 'vue'
import { useRoute } from 'vue-router'
const route = useRoute()
const skipLink = ref()
watch(
  () => route.path,
  () => {
    skipLink.value.focus()
  }
)
</script>
```

Read documentation on skip link to main content

7.3.2 Content Structure

One of the most important pieces of accessibility is making sure that design can support accessible implementation. Design should consider not only color contrast, font selection, text sizing, and language, but also how the content is structured in the application.

```
  top: 0;
  position: fixed;
  left: 50%;
  margin-left: -72px;
  opacity: 0;
}
.skip-link:focus {
  opacity: 1;
  background-color: white;
  padding: 0.5em;
  border: 1px solid black;
}
```

一旦用户改变路由，请将焦点放回到这个“跳过”链接。通过如下方式聚焦“跳过”链接的模板引用 (假设使用了 `vue-router`) 即可实现：

html

```html
<script setup>
import { ref, watch } from 'vue'
import { useRoute } from 'vue-router'
const route = useRoute()
const skipLink = ref()
watch(
  () => route.path,
  () => {
    skipLink.value.focus()
  }
)
</script>
```

阅读关于跳过链接到主要内容的文档

7.3.2 内容结构

确保设计可以支持易于访问的实现是无障碍访问最重要的部分之一。设计不仅要考虑颜色对比度、字体选择、文本大小和语言，还要考虑应用中的内容是如何组织的。

Headings

Users can navigate an application through headings. Having descriptive headings for every section of your application makes it easier for users to predict the content of each section. When it comes to headings, there are a couple of recommended accessibility practices:

- Nest headings in their ranking order: `<h1>` - `<h6>`
- Don't skip headings within a section
- Use actual heading tags instead of styling text to give the visual appearance of headings

Read more about headings

html

```html
<main role="main" aria-labelledby="main-title">
  <h1 id="main-title">Main title</h1>
  <section aria-labelledby="section-title-1">
    <h2 id="section-title-1"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- 内容 -->
  </section>
  <section aria-labelledby="section-title-2">
    <h2 id="section-title-2"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- 内容 -->
    <h3>Section Subtitle</h3>
    <!-- 内容 -->
  </section>
</main>
```

Landmarks

Landmarks provide programmatic access to sections within an application. Users who rely on assistive technology can navigate to each section of the application and skip over content. You can use ARIA roles to help you achieve this.

标题

用户可以通过标题在应用中进行导航。为应用的每个部分设置描述性标题，这可以让用户更容易地预测每个部分的内容。说到标题，有几个推荐的无障碍访问实践：

- 按级别顺序嵌套标题：`<h1>` - `<h6>`
- 不要在一个章节内跳跃标题的级别
- 使用实际的标题标记，而不是通过对文本设置样式以提供视觉上的标题

阅读更多有关标题的信息

html

```html
<main role="main" aria-labelledby="main-title">
  <h1 id="main-title">Main title</h1>
  <section aria-labelledby="section-title-1">
    <h2 id="section-title-1"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- 内容 -->
  </section>
  <section aria-labelledby="section-title-2">
    <h2 id="section-title-2"> Section Title </h2>
    <h3>Section Subtitle</h3>
    <!-- 内容 -->
    <h3>Section Subtitle</h3>
    <!-- 内容 -->
  </section>
</main>
```

Landmarks

Landmark 会为应用中的章节提供访问规划。依赖辅助技术的用户可以跳过内容直接导航到应用的每个部分。你可以使用 ARIA role 帮助你实现这个目标。

HTML	ARIA Role	地标的目的	Landmark Purpose
header	`role="banner"`	主标题：页面的标题	Prime heading: title of the page

nav	`role="navigation"`	适合用作文档或相关文档导航的链接集合	Collection of links suitable for use when navigating the document or related documents
main	`role="main"`	文档的主体或中心内容	The main or central content of the document.
footer	`role="contentinfo"`	关于父级文档的信息：脚注/版权/隐私声明链接	Information about the parent document: footnotes/-copyrights/links to privacy statement
aside	`role="complementary"`	用来支持主内容，同时其自身的内容是相对独立且有意义的	Supports the main content, yet is separated and meaningful on its own content
* 无对应元素 *	`role="search"`	该章节包含整个应用的搜索功能	This section contains the search functionality for the application
hline form	`role="form"`	表单相关元素的集合	Collection of form-associated elements
section	`role="region"`	相关的且用户可能会导航至此的内容。必须为该元素提供 label	Content that is relevant and that users will likely want to navigate to. Label must be provided for this element

TIP
It is recommended to use landmark HTML elements with redundant landmark role attributes in order to maximize compatibility with legacy browsers that don't support HTML5 semantic elements.

Read more about landmarks

7.3.3 Semantic Forms

When creating a form, you can use the following elements: `<form>`, `<label>`, `<input>`, `<textarea>`, and `<button>`

Labels are typically placed on top or to the left of the form fields:

html
```html
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      :type="item.type"
      :id="item.id"
      :name="item.id"
      v-model="item.value"
```

TIP
建议同时使用 landmark HTML 元素和 role 属性，以最大程度地兼容不支持 HTML5 语义元素的传统浏览器。

阅读更多有关标题的细节

7.3.3 语义化表单

当创建一个表单，你可能使用到以下几个元素：`<form>`、`<label>`、`<input>`、`<textarea>` 和 `<button>`。

标签通常放置在表格字段的顶部或左侧：

html
```html
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      :type="item.type"
      :id="item.id"
      :name="item.id"
      v-model="item.value"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

Notice how you can include `autocomplete='on'` on the form element and it will apply to all inputs in your form. You can also set different values for autocomplete attribute for each input.

Labels

Provide labels to describe the purpose of all form control; linking `for` and `id`:

```html
<label for="name">Name</label>
<input type="text" name="name" id="name" v-model="name" />
```

If you inspect this element in your chrome developer tools and open the Accessibility tab inside the Elements tab, you will see how the input gets its name from the label:

```
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

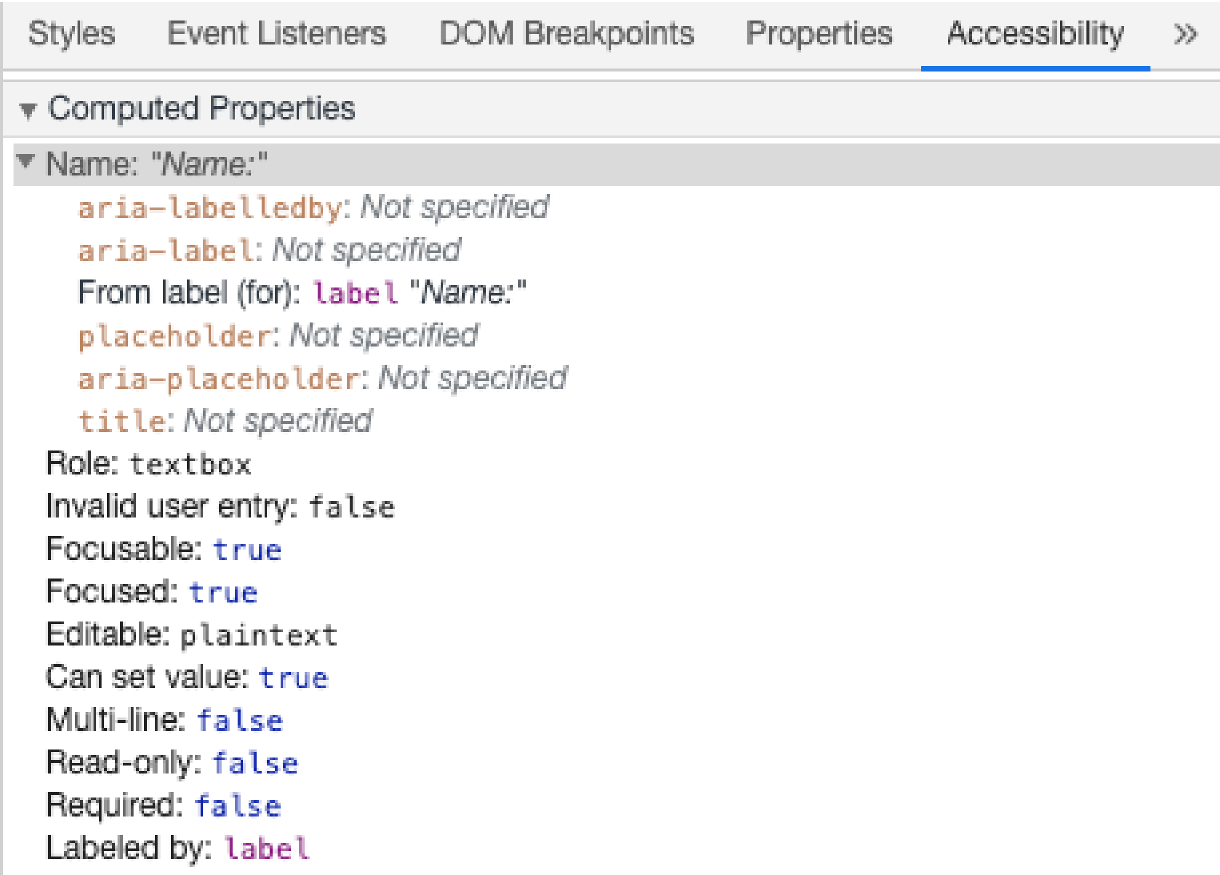
请注意这里我们是如何在表单元素中引入 `autocomplete='on'` 的，它将应用于表单中的所有 input 框。你也可以为每个 input 框都设置不同的 autocomplete attribute 的值。

标签

提供标签来描述所有表单控件的用途；使 `for` 和 `id` 链接起来：

```html
<label for="name">Name</label>
<input type="text" name="name" id="name" v-model="name" />
```

如果你在 chrome 开发工具中检查这个元素，并打开 Elements 选项卡中的 Accessibility 选项卡，你将看到输入是如何从标签中获取其名称的：



▼ Computed Properties

▼ Name: "*Name:*"

aria-labelledby: *Not specified*

aria-label: *Not specified*

From label (for): label "*Name:*"

placeholder: *Not specified*

aria-placeholder: *Not specified*

title: *Not specified*

Role: textbox

Invalid user entry: false

Focusable: true

Focused: true

Editable: plaintext

Can set value: true

Multi-line: false

Read-only: false

Required: false

Labeled by: label

Warning:

Though you might have seen labels wrapping the input fields like this:

html
```html
<label>
  Name:
  <input type="text" name="name" id="name" v-model="name" />
</label>
```

Explicitly setting the labels with a matching id is better supported by assistive technology.

警告：

你可能还见过这样的包装 input 框的标签：

html
```html
<label>
  Name:
  <input type="text" name="name" id="name" v-model="name" />
</label>
```

但我们仍建议你显式地为 input 元素设置 id 相匹配的标签，以更好地实现无障碍访问。

aria-label　You can also give the input an accessible name with `aria-label`.

html
```html
<label for="name">Name</label>
```

你也可以为 input 框配置一个带有 `aria-label` 的无障碍访问名。

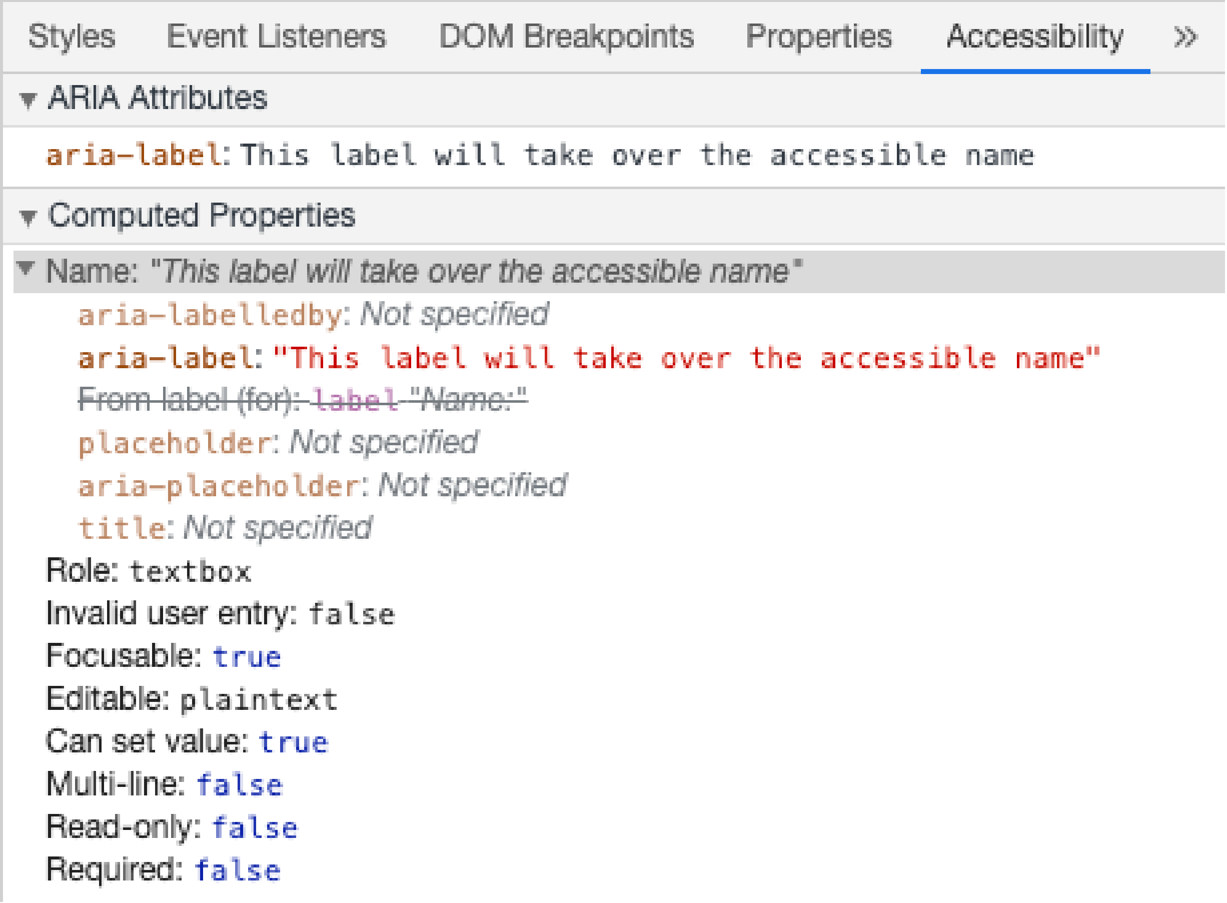
html
```html
<label for="name">Name</label>
```

```
<input
  type="text"
  name="name"
  id="name"
  v-model="name"
  :aria-label="nameLabel"
/>
```

```
<input
  type="text"
  name="name"
  id="name"
  v-model="name"
  :aria-label="nameLabel"
/>
```

Feel free to inspect this element in Chrome DevTools to see how the accessible name has changed:

在 Chrome DevTools 中审查此元素，查看无障碍名称是如何更改的：



▼ ARIA Attributes

aria-label: This label will take over the accessible name

▼ Computed Properties

▼ Name: "This label will take over the accessible name"

aria-labelledby: Not specified

aria-label: "This label will take over the accessible name"

~~From label (for): label "Name:"~~

placeholder: Not specified

aria-placeholder: Not specified

title: Not specified

Role: textbox

Invalid user entry: false

Focusable: true

Editable: plaintext

Can set value: true

Multi-line: false

Read-only: false

Required: false

aria-labelledby Using aria-labelledby is similar to aria-label except it is used if the label text is visible on screen. It is paired to other elements by their id and you can link multiple ids:

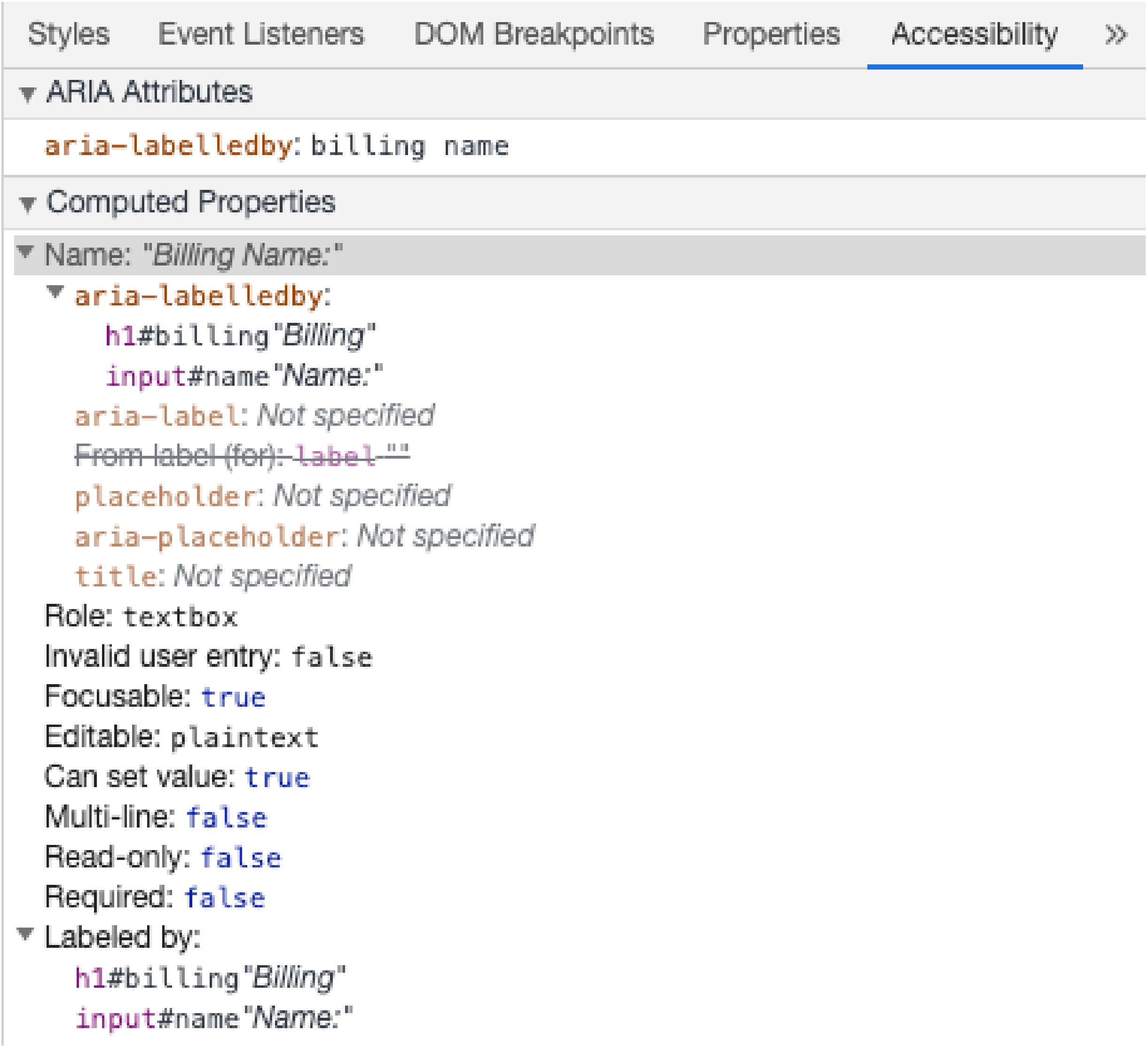
使用 aria-labelledby 类似于 aria-label，除非标签文本在屏幕上可见。它通过 id 与其他元素配对，你可以链接多个 id：

html

```html
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <h1 id="billing">Billing</h1>
  <div class="form-item">
    <label for="name">Name:</label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing name"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

html

```html
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <h1 id="billing">Billing</h1>
  <div class="form-item">
    <label for="name">Name:</label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing name"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```



▼ ARIA Attributes

aria-labelledby: billing name

▼ Computed Properties

▼ Name: "*Billing Name:*"

▼ aria-labelledby:

h1#billing "*Billing*"

input#name "*Name:*"

aria-label: *Not specified*

~~From label (for): label ""~~

placeholder: *Not specified*

aria-placeholder: *Not specified*

title: *Not specified*

Role: textbox

Invalid user entry: false

Focusable: true

Editable: plaintext

Can set value: true

Multi-line: false

Read-only: false

Required: false

▼ Labeled by:

h1#billing "*Billing*"

input#name "*Name:*"

aria-describedby aria-describedby is used the same way as `aria-labelledby` except provides a description with additional information that the user might need. This can be used to describe the criteria for any input:

html
```html
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
```

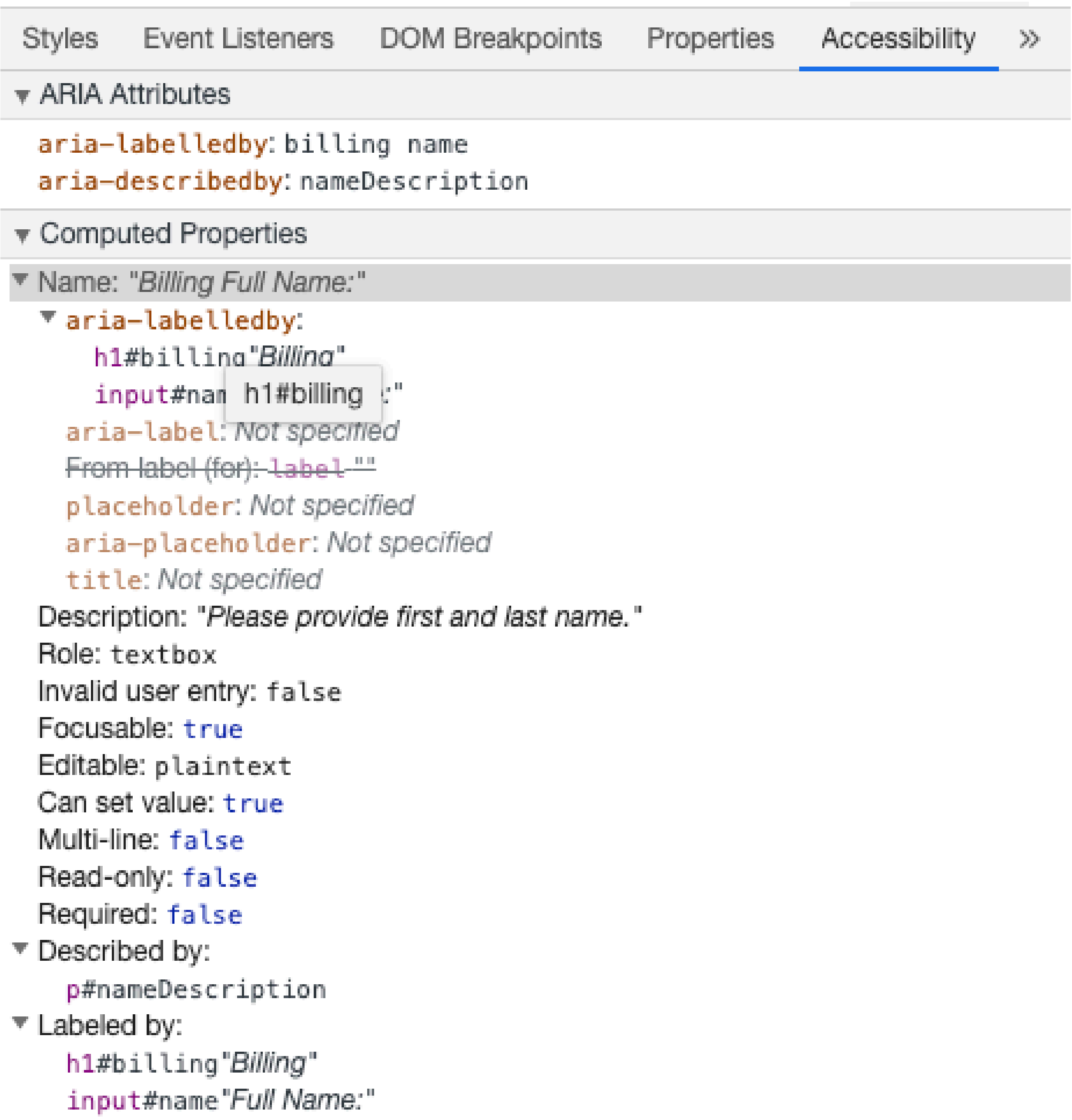
aria-describedby 的用法与 `aria-labelledby` 相同，它提供了一条用户可能需要的附加描述信息。这可用于描述任何输入的标准：

html
```html
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <h1 id="billing">Billing</h1>
  <div class="form-item">
    <label for="name">Full Name:</label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing name"
      aria-describedby="nameDescription"
    />
    <p id="nameDescription">Please provide first and last name.</p>
  </div>
  <button type="submit">Submit</button>
</form>
```

You can see the description by inspecting Chrome DevTools:

```
  autocomplete="on"
>
  <h1 id="billing">Billing</h1>
  <div class="form-item">
    <label for="name">Full Name:</label>
    <input
      type="text"
      name="name"
      id="name"
      v-model="name"
      aria-labelledby="billing name"
      aria-describedby="nameDescription"
    />
    <p id="nameDescription">Please provide first and last name.</p>
  </div>
  <button type="submit">Submit</button>
</form>
```

你可以通过使用Chrome开发工具来查看说明：



▼ ARIA Attributes

aria-labelledby: billing name
aria-describedby: nameDescription

▼ Computed Properties

▼ Name: "*Billing Full Name:*"

▼ aria-labelledby:

h1#billing "*Billing*"

input#nam h1#billing :"

aria-label: *Not specified*

~~From label (for): label ""~~

placeholder: *Not specified*

aria-placeholder: *Not specified*

title: *Not specified*

Description: "*Please provide first and last name.*"

Role: textbox

Invalid user entry: false

Focusable: true

Editable: plaintext

Can set value: true

Multi-line: false

Read-only: false

Required: false

▼ Described by:

p#nameDescription

▼ Labeled by:

h1#billing "*Billing*"

input#name "*Full Name:*"

Placeholder

Avoid using placeholders as they can confuse many users.

One of the issues with placeholders is that they don’t meet the color contrast criteria by default;

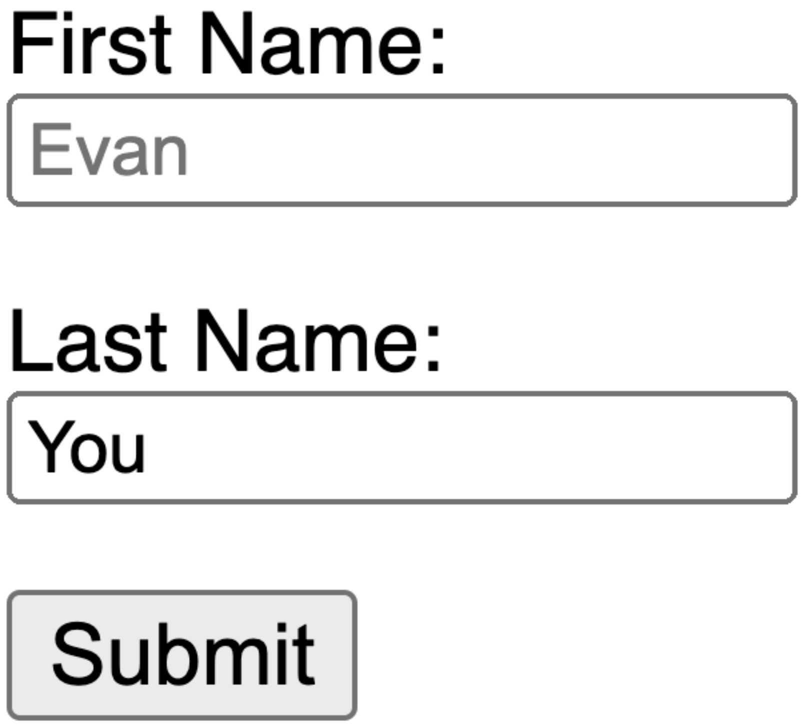
占位符

避免使用占位符，因为它们可能会使许多用户感到困惑。

占位符的缺陷之一是默认情况下它们不符合颜色对比度标准；应当修改其颜色，让

fixing the color contrast makes the placeholder look like pre-populated data in the input fields. Looking at the following example, you can see that the Last Name placeholder which meets the color contrast criteria looks like pre-populated data:

它看起来像是预先填入 input 框中的数据一样。查看以下示例，可以看到满足颜色对比度条件的姓氏占位符看起来像预填充的数据：


First Name:

Evan

Last Name:

You

html
```html
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      type="text"
      :id="item.id"
      :name="item.id"
```

html
```html
<form
  class="demo"
  action="/dataCollectionLocation"
  method="post"
  autocomplete="on"
>
  <div v-for="item in formItems" :key="item.id" class="form-item">
    <label :for="item.id">{{ item.label }}: </label>
    <input
      type="text"
      :id="item.id"
      :name="item.id"
      v-model="item.value"
      :placeholder="item.placeholder"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

css

```
    /* https://www.w3schools.com/howto/howto_css_placeholder.asp */
    #lastName::placeholder {
      /* Chrome, Firefox, Opera, Safari 10.1+ */
      color: black;
      opacity: 1; /* Firefox */
    }
    #lastName:-ms-input-placeholder {
      /* Internet Explorer 10-11 */
      color: black;
    }
    #lastName::-ms-input-placeholder {
      /* Microsoft Edge */
      color: black;
    }
```

It is best to provide all the information the user needs to fill out forms outside any inputs.

Instructions

When adding instructions for your input fields, make sure to link it correctly to the input. You can provide additional instructions and bind multiple ids inside an `aria-labelledby`. This allows for more flexible design.

html

```
<fieldset>
  <legend>Using aria-labelledby</legend>
  <label id="date-label" for="date">Current Date:</label>
  <input
    type="date"
    name="date"
```

```
      v-model="item.value"
      :placeholder="item.placeholder"
    />
  </div>
  <button type="submit">Submit</button>
</form>
```

css

```
/* https://www.w3schools.com/howto/howto_css_placeholder.asp */
#lastName::placeholder {
  /* Chrome, Firefox, Opera, Safari 10.1+ */
  color: black;
  opacity: 1; /* Firefox */
}
#lastName:-ms-input-placeholder {
  /* Internet Explorer 10-11 */
  color: black;
}
#lastName::-ms-input-placeholder {
  /* Microsoft Edge */
  color: black;
}
```

最好在表单外提供所有用户需要填写输入的信息。

用法说明

添加用法说明时，请确保将其正确链接到目标 input 框。你可以提供附加用法说明并在 `aria-labelledby` 内绑定多个 id。这可以使设计更加灵活。

html

```
<fieldset>
  <legend>Using aria-labelledby</legend>
  <label id="date-label" for="date">Current Date:</label>
  <input
    type="date"
    name="date"
    id="date"
    aria-labelledby="date-label date-instructions"
  />
  <p id="date-instructions">MM/DD/YYYY</p>
</fieldset>
```

Alternatively, you can attach the instructions to the input with `aria-describedby`:

html

```
<fieldset>
  <legend>Using aria-describedby</legend>
  <label id="dob" for="dob">Date of Birth:</label>
  <input type="date" name="dob" id="dob" aria-describedby="dob-instructions" />
  <p id="dob-instructions">MM/DD/YYYY</p>
</fieldset>
```

Hiding Content

Usually it is not recommended to visually hide labels, even if the input has an accessible name. However, if the functionality of the input can be understood with surrounding content, then we can hide the visual label.

Let's look at this search field:

html

```
<form role="search">
  <label for="search" class="hidden-visually">Search: </label>
  <input type="text" name="search" id="search" v-model="search" />
  <button type="submit">Search</button>
</form>
```

We can do this because the search button will help visual users identify the purpose of the input field.

We can use CSS to visually hide elements but keep them available for assistive technology:

html

```
.hidden-visually {
  position: absolute;
  overflow: hidden;
```

```
    id="date"
    aria-labelledby="date-label date-instructions"
  />
  <p id="date-instructions">MM/DD/YYYY</p>
</fieldset>
```

或者，你可以通过 `aria-describedby` 将用法说明附加到 input 框上。

html

```
<fieldset>
  <legend>Using aria-describedby</legend>
  <label id="dob" for="dob">Date of Birth:</label>
  <input type="date" name="dob" id="dob" aria-describedby="dob-instructions" />
  <p id="dob-instructions">MM/DD/YYYY</p>
</fieldset>
```

隐藏内容

通常，即使 input 框具有无障碍的名称，也不建议在视觉上隐藏标签。但是，如果可以借助周围的内容来理解输入的功能，那么我们也可以隐藏视觉标签。

让我们看看这个搜索框：

html

```
<form role="search">
  <label for="search" class="hidden-visually">Search: </label>
  <input type="text" name="search" id="search" v-model="search" />
  <button type="submit">Search</button>
</form>
```

现在，只要视力情况良好，用户可以就能通过按钮的内容识别出该 input 框的目的。

此时我们可以使用 CSS 从视觉上隐藏元素，同时也不会影响到无障碍访问：

css

```
.hidden-visually {
  position: absolute;
  overflow: hidden;
```

```css
  white-space: nowrap;
  margin: 0;
  padding: 0;
  height: 1px;
  width: 1px;
  clip: rect(0 0 0 0);
  clip-path: inset(100%);
}
```

aria-hidden=”true” Adding `aria-hidden="true"` will hide the element from assistive technology but leave it visually available for other users. Do not use it on focusable elements, purely on decorative, duplicated or offscreen content.

html

```html
<p>This is not hidden from screen readers.</p>
<p aria-hidden="true">This is hidden from screen readers.</p>
```

Buttons

When using buttons inside a form, you must set the type to prevent submitting the form. You can also use an input to create buttons:

html

```html
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <!-- 按钮 -->
  <button type="button">Cancel</button>
  <button type="submit">Submit</button>
  <!-- 输入按钮 -->
  <input type="button" value="Cancel" />
  <input type="submit" value="Submit" />
</form>
```

Functional Images

You can use this technique to create functional images.

```css
  white-space: nowrap;
  margin: 0;
  padding: 0;
  height: 1px;
  width: 1px;
  clip: rect(0 0 0 0);
  clip-path: inset(100%);
}
```

添加 `aria-hidden="true"` 在无障碍访问时被隐藏，但对其他可视用户仍然是可见的。不要在可聚焦的元素上使用它，请只在装饰性的、重复的或屏幕外的内容上使用它。

html

```html
<p>This is not hidden from screen readers.</p>
<p aria-hidden="true">This is hidden from screen readers.</p>
```

按钮

在表单中使用按钮时，必须设置类型以防止提交表单。你也可以使用一个 input 元素来创建按钮：

html

```html
<form action="/dataCollectionLocation" method="post" autocomplete="on">
  <!-- 按钮 -->
  <button type="button">Cancel</button>
  <button type="submit">Submit</button>
  <!-- 输入按钮 -->
  <input type="button" value="Cancel" />
  <input type="submit" value="Submit" />
</form>
```

功能图片

你可以使用这种方式来创建一个带有功能的图片。

- Input fields
  - These images will act as a submit type button on forms

html

```html
<form role="search">
  <label for="search" class="hidden-visually">Search: </label>
  <input type="text" name="search" id="search" v-model="search" />
  <input
    type="image"
    class="btnImg"
    src="https://img.icons8.com/search"
    alt="Search"
  />
</form>
```

- Icons

html

```html
<form role="search">
  <label for="searchIcon" class="hidden-visually">Search: </label>
  <input type="text" name="searchIcon" id="searchIcon" v-model="searchIcon" />
  <button type="submit">
    <i class="fas fa-search" aria-hidden="true"></i>
    <span class="hidden-visually">Search</span>
  </button>
</form>
```

7.3.4 Standards

The World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI) develops web accessibility standards for the different components:

- User Agent Accessibility Guidelines (UAAG)
  - web browsers and media players, including some aspects of assistive technologies
- Authoring Tool Accessibility Guidelines (ATAG)
  - authoring tools

- input 框
  - 这些图片会像一个类型为 submit 的表单按钮一样

html

```html
<form role="search">
  <label for="search" class="hidden-visually">Search: </label>
  <input type="text" name="search" id="search" v-model="search" />
  <input
    type="image"
    class="btnImg"
    src="https://img.icons8.com/search"
    alt="Search"
  />
</form>
```

- 图标

html

```html
<form role="search">
  <label for="searchIcon" class="hidden-visually">Search: </label>
  <input type="text" name="searchIcon" id="searchIcon" v-model="searchIcon" />
  <button type="submit">
    <i class="fas fa-search" aria-hidden="true"></i>
    <span class="hidden-visually">Search</span>
  </button>
</form>
```

7.3.4 规范

万维网联盟 (W3C) Web 无障碍访问倡议 (WAI) 为不同的组件制定了 Web 无障碍性标准：

- 用户代理无障碍访问指南 (UAAG)
  - 浏览器和媒体查询，包括一些其他方面的辅助技术
- 创作工具无障碍访问指南 (ATAG)
  - 创作工具

- Web Content Accessibility Guidelines (WCAG)
  - web content - used by developers, authoring tools, and accessibility evaluation tools

Web Content Accessibility Guidelines (WCAG)

WCAG 2.1 extends on WCAG 2.0 and allows implementation of new technologies by addressing changes to the web. The W3C encourages use of the most current version of WCAG when developing or updating Web accessibility policies.

WCAG 2.1 四大指导原则 (缩写 POUR)：

- Perceivable
  - Users must be able to perceive the information being presented
- Operable
  - Interface forms, controls, and navigation are operable
- Understandable
  - Information and the operation of user interface must be understandable to all users
- Robust
  - Users must be able to access the content as technologies advance

Web 无障碍倡议 – 无障碍访问丰富的互联网应用（WAI-ARIA） W3C's WAI-ARIA provides guidance on how to build dynamic content and advanced user interface controls.

- Accessible Rich Internet Applications (WAI-ARIA) 1.2
- WAI-ARIA Authoring Practices 1.2

7.3.5 Resources

Documentation

- Web 内容无障碍访问指南 (WCAG)
  - 网站内容 - 由开发者、创作工具和无障碍访问评估工具使用。

网络内容无障碍指南（WCAG）

WCAG 2.1 继承自 WCAG 2.0，接纳 Web 演进过程中的新技术。W3C 鼓励在开发或更新 Web 无障碍访问策略时使用 WCAG 的最新版本。

- 可感知性
  - 用户必须能够感知所渲染的信息
- 可操作性
  - 表单界面，控件和导航是可操作的
- 可理解性
  - 信息和用户界面的操作必须为所有用户所理解
- 健壮性
  - 随着技术的进步，用户必须能够访问内容

W3C 的 WAI-ARIA 为如何构建动态内容和高阶用户界面控件提供了指导。

- 可便捷访问的丰富互联网应用 (WAI-ARIA) 1.2
- WAI-ARIA 实践 1.2

7.3.5 资源

文档

- WCAG 2.0
- WCAG 2.1
- Accessible Rich Internet Applications (WAI-ARIA) 1.2
- WAI-ARIA Authoring Practices 1.2

Assistive Technologies

- Screen Readers
  - NVDA
  - VoiceOver
  - JAWS
  - ChromeVox
- Zooming Tools
  - MAGic
  - ZoomText
  - Magnifier

Testing

- Automated Tools
  - Lighthouse
  - WAVE
  - ARC Toolkit
- Color Tools
  - WebAim Color Contrast
  - WebAim Link Color Contrast

- WCAG 2.0
- WCAG 2.1
- Accessible Rich Internet Applications (WAI-ARIA) 1.2
- WAI-ARIA Authoring Practices 1.2

辅助技术

- 屏幕助读器
  - NVDA
  - VoiceOver
  - JAWS
  - ChromeVox
- 缩放工具
  - MAGic
  - ZoomText
  - Magnifier

测试

- 自动化相关的工具
  - Lighthouse
  - WAVE
  - ARC Toolkit
- 颜色相关的工具
  - WebAim Color Contrast
  - WebAim Link Color Contrast

- Other Helpful Tools
    - HeadingMap
    - Color Oracle
    - NerdeFocus
    - Visual Aria
    - Silktide Website Accessibility Simulator

Users

The World Health Organization estimates that 15% of the world's population has some form of disability, 2-4% of them severely so. That is an estimated 1 billion people worldwide; making people with disabilities the largest minority group in the world.

There are a huge range of disabilities, which can be divided roughly into four categories:

- *Visual* - These users can benefit from the use of screen readers, screen magnification, controlling screen contrast, or braille display.
- *Auditory* - These users can benefit from captioning, transcripts or sign language video.
- *Motor* - These users can benefit from a range of assistive technologies for motor impairments: voice recognition software, eye tracking, single-switch access, head wand, sip and puff switch, oversized trackball mouse, adaptive keyboard or other assistive technologies.
- *Cognitive* - These users can benefit from supplemental media, structural organization of content, clear and simple writing.

Check out the following links from WebAim to understand from users:

- Web Accessibility Perspectives: Explore the Impact and Benefits for Everyone
- Stories of Web Users

Edit this page on GitHub

7.4 Security

- 其他有用的工具
    - HeadingMap
    - Color Oracle
    - NerdeFocus
    - Visual Aria
    - Silktide Website Accessibility Simulator

用户

世界卫生组织估计，全世界 15% 的人口患有某种形式的残疾，其中约 2 - 4% 的人严重残疾。估计全世界有 10 亿残障人士，他们是世界上最大的少数群体。

残疾的种类繁多，大致可分为以下四类：

- *视觉* - 可以为这些用户提供屏幕助读器、屏幕缩放、控制屏幕对比度或盲文显示等帮助。
- *听觉* - 可以为这些用户提供视频字幕、文字记录或手语视频。
- *运动能力* - 可以为这些用户提供一系列运动障碍辅助技术中：比如语音识别软件、眼球跟踪、单刀式开关、超大轨迹球鼠标、自适应键盘等等。
- *认知能力* - 可以为这些用户提供补充媒体、更清晰和简单、更结构化的内容。

你可以查看以下来自 WebAim 的链接，更深入地了解这些用户的需求：

- Web 无障碍愿景：探索改变 & 人人受益
- Web 用户的故事

在 GitHub 上编辑此页

7.4 安全

7.4.1 Reporting Vulnerabilities

When a vulnerability is reported, it immediately becomes our top concern, with a full-time contributor dropping everything to work on it. To report a vulnerability, please email `security@vuejs.org`.

While the discovery of new vulnerabilities is rare, we also recommend always using the latest versions of Vue and its official companion libraries to ensure your application remains as secure as possible.

7.4.2 Rule No.1: Never Use Non-trusted Templates

The most fundamental security rule when using Vue is **never use non-trusted content as your component template**. Doing so is equivalent to allowing arbitrary JavaScript execution in your application - and worse, could lead to server breaches if the code is executed during server-side rendering. An example of such usage:

```js
Vue.createApp({
  template: `<div>` + userProvidedString + `</div>` // 永远不要这样做!
}).mount('#app')
```

Vue templates are compiled into JavaScript, and expressions inside templates will be executed as part of the rendering process. Although the expressions are evaluated against a specific rendering context, due to the complexity of potential global execution environments, it is impractical for a framework like Vue to completely shield you from potential malicious code execution without incurring unrealistic performance overhead. The most straightforward way to avoid this category of problems altogether is to make sure the contents of your Vue templates are always trusted and entirely controlled by you.

7.4.3 What Vue Does to Protect You

HTML content

Whether using templates or render functions, content is automatically escaped. That means in this template:

```html
<h1>{{ userProvidedString }}</h1>
```

7.4.1 报告漏洞

当一个漏洞被上报时，它会立刻成为我们最关心的问题，会有全职的贡献者暂时搁置其他所有任务来解决这个问题。如需报告漏洞，请发送电子邮件至 `security@vuejs.org`。

虽然很少发现新的漏洞，但我们仍建议始终使用最新版本的 Vue 及其官方配套库，以确保你的应用尽可能地安全。

7.4.2 首要规则：不要使用无法信赖的模板

使用 Vue 时最基本的安全规则就是**不要将无法信赖的内容作为你的组件模板**。使用无法信赖的模板相当于允许任意的 JavaScript 在你的应用中执行。更糟糕的是，如果在服务端渲染时执行了这些代码，可能会导致服务器被攻击。举例来说：

```js
Vue.createApp({
  template: `<div>` + userProvidedString + `</div>` // 永远不要这样做!
}).mount('#app')
```

Vue 模板会被编译成 JavaScript，而模板内的表达式将作为渲染过程的一部分被执行。尽管这些表达式在特定的渲染环境中执行，但由于全局执行环境的复杂性，Vue 作为一个开发框架，要在性能开销合理的前提下完全避免潜在的恶意代码执行是不现实的。避免这类问题最直接的方法是确保你的 Vue 模板始终是可信的，并且完全由你控制。

7.4.3 Vue 自身的安全机制

HTML 内容

无论是使用模板还是渲染函数，内容都是自动转义的。这意味着在这个模板中：

```html
<h1>{{ userProvidedString }}</h1>
```

if userProvidedString contained:

```js
'<script>alert("hi")</script>'
```

then it would be escaped to the following HTML:

```html
&lt;script&gt;alert(&quot;hi&quot;)&lt;/script&gt;
```

thus preventing the script injection. This escaping is done using native browser APIs, like textContent so a vulnerability can only exist if the browser itself is vulnerable.

Attribute bindings

Similarly, dynamic attribute bindings are also automatically escaped. That means in this template:

```html
<h1 :title="userProvidedString">
  hello
</h1>
```

if userProvidedString contained:

```js
'" onclick="alert(\'hi\')'
```

then it would be escaped to the following HTML:

```html
&quot; onclick=&quot;alert('hi')
```

thus preventing the close of the title attribute to inject new, arbitrary HTML. This escaping is done using native browser APIs, like setAttribute, so a vulnerability can only exist if the browser itself is vulnerable.

7.4.4 Potential Dangers

In any web application, allowing unsanitized, user-provided content to be executed as HTML, CSS, or JavaScript is potentially dangerous, so it should be avoided wherever possible. There are times when some risk may be acceptable, though.

For example, services like CodePen and JSFiddle allow user-provided content to be executed, but

如果 userProvidedString 包含了：

```js
'<script>alert("hi")</script>'
```

那么它将被转义为如下的 HTML：

```html
&lt;script&gt;alert(&quot;hi&quot;)&lt;/script&gt;
```

从而防止脚本注入。这种转义是使用 textContent 这样的浏览器原生 API 完成的，所以只有当浏览器本身存在漏洞时，才会存在漏洞。

Attribute 绑定

同样地，动态 attribute 的绑定也会被自动转义。这意味着在这个模板中：

```html
<h1 :title="userProvidedString">
  hello
</h1>
```

如果 userProvidedString 包含了：

```js
'" onclick="alert(\'hi\')'
```

那么它将被转义为如下的 HTML：

```html
&quot; onclick=&quot;alert('hi')
```

从而防止在 title attribute 解析时，注入任意的 HTML。这种转义是使用 setAttribute 这样的浏览器原生 API 完成的，所以只有当浏览器本身存在漏洞时，才会存在漏洞。

7.4.4 潜在的危险

在任何 Web 应用中，允许以 HTML、CSS 或 JavaScript 形式执行未经无害化处理的、用户提供的内容都有潜在的安全隐患，因此这应尽可能避免。不过，有时候一些风险或许是可以接受的。

例如，像 CodePen 和 JSFiddle 这样的服务允许执行用户提供的内容，但这是在

it's in a context where this is expected and sandboxed to some extent inside iframes. In the cases when an important feature inherently requires some level of vulnerability, it's up to your team to weigh the importance of the feature against the worst-case scenarios the vulnerability enables.

HTML Injection

As you learned earlier, Vue automatically escapes HTML content, preventing you from accidentally injecting executable HTML into your application. However, **in cases where you know the HTML is safe**, you can explicitly render HTML content:

- Using a template:

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

- Using a render function:

```js
h('div', {
  innerHTML: this.userProvidedHtml
})
```

- Using a render function with JSX:

```html
<div innerHTML={this.userProvidedHtml}></div>
```

> **WARNING**
> User-provided HTML can never be considered 100% safe unless it's in a sandboxed iframe or in a part of the app where only the user who wrote that HTML can ever be exposed to it. Additionally, allowing users to write their own Vue templates brings similar dangers.

URL Injection

In a URL like this:

```html
<a :href="userProvidedUrl">
  click me
```

iframe 这样一个可预期的沙盒环境中。当一个重要的功能本身会伴随某种程度的漏洞时，就需要你自行权衡该功能的重要性和该漏洞所带来的最坏情况。

注入 HTML

我们现在已经知道 Vue 会自动转义 HTML 内容,防止你意外地将可执行的 HTML 注入到你的应用中。然而，**在你知道 HTML 安全的情况下**，你还是可以显式地渲染 HTML 内容。

- 使用模板：

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

- 使用渲染函数：

```js
h('div', {
  innerHTML: this.userProvidedHtml
})
```

- 以 JSX 形式使用渲染函数：

```html
<div innerHTML={this.userProvidedHtml}></div>
```

> **警告**
> 用户提供的 HTML 永远不能被认为是 100% 安全的，除非它在 iframe 这样的沙盒环境中，或者该 HTML 只会被该用户看到。此外，允许用户编写自己的 Vue 模板也会带来类似的危险。

URL 注入

在这样一个使用 URL 的场景中：

```html
<a :href="userProvidedUrl">
  click me
```

```
</a>
```

There's a potential security issue if the URL has not been "sanitized" to prevent JavaScript execution using `javascript:`. There are libraries such as sanitize-url to help with this, but note: if you're ever doing URL sanitization on the frontend, you already have a security issue. **User-provided URLs should always be sanitized by your backend before even being saved to a database.** Then the problem is avoided for *every* client connecting to your API, including native mobile apps. Also note that even with sanitized URLs, Vue cannot help you guarantee that they lead to safe destinations.

Style Injection

Looking at this example:

```html
<a
  :href="sanitizedUrl"
  :style="userProvidedStyles"
>
  click me
</a>
```

Let's assume that `sanitizedUrl` has been sanitized, so that it's definitely a real URL and not JavaScript. With the `userProvidedStyles`, malicious users could still provide CSS to "click jack", e.g. styling the link into a transparent box over the "Log in" button. Then if `https://user-controlled-website.com/` is built to resemble the login page of your application, they might have just captured a user's real login information.

You may be able to imagine how allowing user-provided content for a `<style>` element would create an even greater vulnerability, giving that user full control over how to style the entire page. That's why Vue prevents rendering of style tags inside templates, such as:

```html
<style>{{ userProvidedStyles }}</style>
```

To keep your users fully safe from clickjacking, we recommend only allowing full control over CSS inside a sandboxed iframe. Alternatively, when providing user control through a style binding, we recommend using its object syntax and only allowing users to provide values for specific properties it's safe for them to control, like this:

```
</a>
```

如果这个 URL 允许通过 `javascript:` 执行 JavaScript，即没有进行无害化处理，那么就会有一些潜在的安全问题。可以使用一些库来解决此类问题，比如 sanitize-url，但请注意：如果你发现你需要在前端做 URL 无害化处理，那你的应用已经存在一个更严重的安全问题了。**任何用户提供的 URL 在被保存到数据库之前都应该先在后端做无害化处理。**这样，连接到你 API 的*每一个*客户端都可以避免这个问题，包括原生移动应用。另外，即使是经过无害化处理的 URL，Vue 也不能保证它们指向安全的目的地。

样式注入

我们来看这样一个例子：

```html
<a
  :href="sanitizedUrl"
  :style="userProvidedStyles"
>
  click me
</a>
```

我们假设 `sanitizedUrl` 已进行无害化处理，它是一个正常 URL 而非 JavaScript。然而，由于 `userProvidedStyles` 的存在，恶意用户仍然能利用 CSS 进行“点击劫持”，例如，可以在“登录”按钮上方覆盖一个透明的链接。如果用户控制的页面 `https://user-controlled-website.com/` 专门仿造了你应用的登录页，那么他们就有可能捕获用户的真实登录信息。

你可以想象，如果允许在 `<style>` 元素中插入用户提供的内容，会造成更大的漏洞，因为这使得用户能控制整个页面的样式。因此 Vue 阻止了在模板中像这样渲染 style 标签：

```html
<style>{{ userProvidedStyles }}</style>
```

为了避免用户的点击被劫持，我们建议仅在沙盒环境的 iframe 中允许用户控制 CSS。或者，当用户控制样式绑定时，我们建议使用其对象值形式并仅允许用户提供能够安全控制的、特定的属性，就像这样：

html
```html
<a
  :href="sanitizedUrl"
  :style="{
    color: userProvidedColor,
    background: userProvidedBackground
  }"
>
  click me
</a>
```

JavaScript Injection

We strongly discourage ever rendering a `<script>` element with Vue, since templates and render functions should never have side effects. However, this isn't the only way to include strings that would be evaluated as JavaScript at runtime.

Every HTML element has attributes with values accepting strings of JavaScript, such as `onclick`, `onfocus`, and `onmouseenter`. Binding user-provided JavaScript to any of these event attributes is a potential security risk, so it should be avoided.

WARNING
User-provided JavaScript can never be considered 100% safe unless it's in a sandboxed iframe or in a part of the app where only the user who wrote that JavaScript can ever be exposed to it.

Sometimes we receive vulnerability reports on how it's possible to do cross-site scripting (XSS) in Vue templates. In general, we do not consider such cases to be actual vulnerabilities because there's no practical way to protect developers from the two scenarios that would allow XSS:

1. The developer is explicitly asking Vue to render user-provided, unsanitized content as Vue templates. This is inherently unsafe, and there's no way for Vue to know the origin.

2. The developer is mounting Vue to an entire HTML page which happens to contain server-rendered and user-provided content. This is fundamentally the same problem as #1, but sometimes devs may do it without realizing it. This can lead to possible vulnerabilities where the attacker provides HTML which is safe as plain HTML but unsafe as a Vue template. The

html
```html
<a
  :href="sanitizedUrl"
  :style="{
    color: userProvidedColor,
    background: userProvidedBackground
  }"
>
  click me
</a>
```

JavaScript 注入

我们强烈建议任何时候都不要在 Vue 中渲染 `<script>`，因为模板和渲染函数不应有其他副作用。但是，渲染 `<script>` 并不是插入在运行时执行的 JavaScript 字符串的唯一方法。

每个 HTML 元素都有能接受字符串形式 JavaScript 的 attribute，例如 `onclick`、`onfocus` 和 `onmouseenter`。绑定任何用户提供的 JavaScript 给这些事件 attribute 都具有潜在风险，因此需要避免这么做。

警告
用户提供的 JavaScript 永远不能被认为是 100% 安全的，除非它在 iframe 这样的沙盒环境中，或者该段代码只会在该用户登录的页面上被执行。

有时我们会收到漏洞报告，说在 Vue 模板中可以进行跨站脚本攻击 (XSS)。一般来说，我们不认为这种情况是真正的漏洞，因为没有切实可行的方法，能够在以下两种场景中保护开发者不受 XSS 的影响。

1. 开发者显式地将用户提供的、未经无害化处理的内容作为 Vue 模板渲染。这本身就是不安全的，Vue 也无从溯源。

2. 开发者将 Vue 挂载到可能包含服务端渲染或用户提供内容的 HTML 页面上，这与 #1 的问题基本相同，但有时开发者可能会不知不觉地这样做。攻击者提供的 HTML 可能在普通 HTML 中是安全的，但在 Vue 模板中是不安全的，这就会导致漏洞。最佳实践是：**不要将 Vue 挂载到可能包含服务**

best practice is to **never mount Vue on nodes that may contain server-rendered and user-provided content**.

7.4.5 Best Practices

The general rule is that if you allow unsanitized, user-provided content to be executed (as either HTML, JavaScript, or even CSS), you might open yourself up to attacks. This advice actually holds true whether using Vue, another framework, or even no framework.

Beyond the recommendations made above for Potential Dangers, we also recommend familiarizing yourself with these resources:

- HTML5 Security Cheat Sheet
- OWASP's Cross Site Scripting (XSS) Prevention Cheat Sheet

Then use what you learn to also review the source code of your dependencies for potentially dangerous patterns, if any of them include 3rd-party components or otherwise influence what's rendered to the DOM.

7.4.6 Backend Coordination

HTTP security vulnerabilities, such as cross-site request forgery (CSRF/XSRF) and cross-site script inclusion (XSSI), are primarily addressed on the backend, so they aren't a concern of Vue's. However, it's still a good idea to communicate with your backend team to learn how to best interact with their API, e.g., by submitting CSRF tokens with form submissions.

7.4.7 Server-Side Rendering (SSR)

There are some additional security concerns when using SSR, so make sure to follow the best practices outlined throughout our SSR documentation to avoid vulnerabilities.

端渲染或用户提供内容的 DOM 节点上。

7.4.5 最佳实践

最基本的规则就是只要你允许执行未经无害化处理的、用户提供的内容 (无论是 HTML、JavaScript 还是 CSS)，你就可能面临攻击。无论是使用 Vue、其他框架，或是不使用框架，道理都是一样的。

除了上面为处理潜在危险提供的建议，我们也建议你熟读下面这些资源：

- HTML5 安全手册
- OWASP 的跨站脚本攻击 (XSS) 防护手册

接着你可以利用学到的知识，来审查依赖项的源代码，看看是否有潜在的危险，防止它们中的任何一个以第三方组件或其他方式影响 DOM 渲染的内容。

7.4.6 后端协调

类似跨站请求伪造 (CSRF/XSRF) 和跨站脚本引入 (XSSI) 这样的 HTTP 安全漏洞，主要由后端负责处理，因此它们不是 Vue 职责范围内的问题。但是，你应该与后端团队保持沟通，了解如何更好地与后端 API 进行交互，例如，在提交表单时附带 CSRF 令牌。

7.4.7 服务端渲染 (SSR)

在使用 SSR 时还有一些其他的安全注意事项，因此请确保遵循我们的 SSR 文档给出的最佳实践来避免产生漏洞。

第八章　TypeScript

8.1　Using Vue with TypeScript

A type system like TypeScript can detect many common errors via static analysis at build time. This reduces the chance of runtime errors in production, and also allows us to more confidently refactor code in large-scale applications. TypeScript also improves developer ergonomics via type-based auto-completion in IDEs.

Vue is written in TypeScript itself and provides first-class TypeScript support. All official Vue packages come with bundled type declarations that should work out-of-the-box.

8.1.1　Project Setup

`create-vue`, the official project scaffolding tool, offers the options to scaffold a Vite-powered, TypeScript-ready Vue project.

Overview

With a Vite-based setup, the dev server and the bundler are transpilation-only and do not perform any type-checking. This ensures the Vite dev server stays blazing fast even when using TypeScript.

- During development, we recommend relying on a good IDE setup for instant feedback on type errors.
- If using SFCs, use the `vue-tsc` utility for command line type checking and type declaration generation. `vue-tsc` is a wrapper around `tsc`, TypeScript's own command line interface. It

8.1　搭配 TypeScript 使用 Vue

像 TypeScript 这样的类型系统可以在编译时通过静态分析检测出很多常见错误。这减少了生产环境中的运行时错误，也让我们在重构大型项目的时候更有信心。通过 IDE 中基于类型的自动补全，TypeScript 还改善了开发体验和效率。

Vue 本身就是用 TypeScript 编写的，并对 TypeScript 提供了一等公民的支持。所有的 Vue 官方库都自带了类型声明文件，开箱即用。

8.1.1　项目配置

`create-vue`，即官方的项目脚手架工具，提供了搭建基于 Vite 且 TypeScript 就绪的 Vue 项目的选项。

总览

在基于 Vite 的配置中，开发服务器和打包器将只会对 TypeScript 文件执行语法转译，而不会执行任何类型检查，这保证了 Vite 开发服务器在使用 TypeScript 时也能始终保持飞快的速度。

- 在开发阶段，我们推荐你依赖一个好的 IDE 配置来获取即时的类型错误反馈。
- 对于单文件组件，你可以使用工具 `vue-tsc` 在命令行检查类型和生成类型声明文件。`vue-tsc` 是对 TypeScript 自身命令行界面 `tsc` 的一个封装。它的

works largely the same as `tsc` except that it supports Vue SFCs in addition to TypeScript files. You can run `vue-tsc` in watch mode in parallel to the Vite dev server, or use a Vite plugin like vite-plugin-checker which runs the checks in a separate worker thread.

- Vue CLI also provides TypeScript support, but is no longer recommended. See notes below.

IDE Support

- Visual Studio Code (VSCode) is strongly recommended for its great out-of-the-box support for TypeScript.
  - Volar is the official VSCode extension that provides TypeScript support inside Vue SFCs, along with many other great features.

    > **TIP**
    > Volar replaces Vetur, our previous official VSCode extension for Vue 2. If you have Vetur currently installed, make sure to disable it in Vue 3 projects.

  - TypeScript Vue Plugin is also needed to get type support for `*.vue` imports in TS files.
- WebStorm also provides out-of-the-box support for both TypeScript and Vue. Other JetBrains IDEs support them too, either out of the box or via a free plugin. As of version 2023.2, WebStorm and the Vue Plugin come with built-in support for the Vue Language Server. You can set the Vue service to use Volar integration on all TypeScript versions, under Settings > Languages & Frameworks > TypeScript > Vue. By default, Volar will be used for TypeScript versions 5.0 and higher.

Configuring tsconfig.json

Projects scaffolded via `create-vue` include pre-configured `tsconfig.json`. The base config is abstracted in the `@vue/tsconfig` package. Inside the project, we use Project References to ensure correct types for code running in different environments (e.g. app code and test code should have different global variables).

When configuring `tsconfig.json` manually, some notable options include:

- `compilerOptions.isolatedModules` is set to `true` because Vite uses esbuild for transpiling

工作方式基本和 `tsc` 一致。除了 TypeScript 文件，它还支持 Vue 的单文件组件。你可以在开启 Vite 开发服务器的同时以侦听模式运行 `vue-tsc`，或是使用 vite-plugin-checker 这样在另一个 worker 线程里做静态检查的插件。

- Vue CLI 也提供了对 TypeScript 的支持，但是已经不推荐了。详见下方的说明。

IDE 支持

- 强烈推荐 Visual Studio Code (VSCode)，因为它对 TypeScript 有着很好的内置支持。
  - Volar 是官方的 VSCode 扩展，提供了 Vue 单文件组件中的 TypeScript 支持，还伴随着一些其他非常棒的特性。

    > **TIP**
    > Volar 取代了我们之前为 Vue 2 提供的官方 VSCode 扩展 Vetur。如果你之前已经安装了 Vetur，请确保在 Vue 3 的项目中禁用它。

  - TypeScript Vue Plugin 用于支持在 TS 中 import `*.vue` 文件。
- WebStorm 对 TypeScript 和 Vue 也都提供了开箱即用的支持。其他的 JetBrains IDE 也同样可以通过一个免费插件支持。从 2023.2 版开始，WebStorm 和 Vue 插件内置了对 Vue 语言服务器的支持。你可以在设置 > 语言和框架 > TypeScript > Vue 下将 Vue 服务设置为在所有 TypeScript 版本上使用 Volar 集成。默认情况下，Volar 将用于 TypeScript 5.0 及更高版本。

配置 tsconfig.json

通过 `create-vue` 搭建的项目包含了预先配置好的 `tsconfig.json`。其底层配置抽象于 `@vue/tsconfig` 包中。在项目内我们使用 Project References 来确保运行在不同环境下的代码的类型正确（比如应用代码和测试代码应该有不同的全局变量）。

手动配置 `tsconfig.json` 时，请留意以下选项：

- `compilerOptions.isolatedModules` 应当设置为 `true`，因为 Vite 使用 es-

TypeScript and is subject to single-file transpile limitations.

- If you're using Options API, you need to set `compilerOptions.strict` to `true` (or at least enable `compilerOptions.noImplicitThis`, which is a part of the `strict` flag) to leverage type checking of `this` in component options. Otherwise `this` will be treated as `any`.

- If you have configured resolver aliases in your build tool, for example the `@/*` alias configured by default in a `create-vue` project, you need to also configure it for TypeScript via `compilerOptions.paths`.

See also:

- Official TypeScript compiler options docs

- esbuild TypeScript compilation caveats

Volar Takeover Mode

> This section only applies for VSCode + Volar.

To get Vue SFCs and TypeScript working together, Volar creates a separate TS language service instance patched with Vue-specific support, and uses it in Vue SFCs. At the same time, plain TS files are still handled by VSCode's built-in TS language service, which is why we need TypeScript Vue Plugin to support Vue SFC imports in TS files. This default setup works, but for each project we are running two TS language service instances: one from Volar, one from VSCode's built-in service. This is a bit inefficient and can lead to performance issues in large projects.

Volar provides a feature called "Takeover Mode" to improve performance. In takeover mode, Volar provides support for both Vue and TS files using a single TS language service instance.

To enable Takeover Mode, you need to disable VSCode's built-in TS language service in **your project's workspace only** by following these steps:

1. In your project workspace, bring up the command palette with `Ctrl + Shift + P` (macOS: `Cmd + Shift + P`).

2. Type `built` and select "Extensions: Show Built-in Extensions".

3. Type `typescript` in the extension search box (do not remove `@builtin` prefix).

4. Click the little gear icon of "TypeScript and JavaScript Language Features", and select "Dis-

build 来转译 TypeScript，并受限于单文件转译的限制。

- 如果你正在使用选项式 API，需要将 `compilerOptions.strict` 设置为 `true` (或者至少开启 `compilerOptions.noImplicitThis`，它是 `strict` 模式的一部分)，才可以获得对组件选项中 `this` 的类型检查。否则 `this` 会被认为是 `any`。

- 如果你在构建工具中配置了路径解析别名，例如 `@/*` 这个别名被默认配置在了 `create-vue` 项目中，你需要通过 `compilerOptions.paths` 选项为 TypeScript 再配置一遍。

参考：

- 官方 TypeScript 编译选项文档

- esbuild TypeScript 编译注意事项

Volar Takeover 模式

> 这一章节仅针对 VSCode + Volar。

为了让 Vue 单文件组件和 TypeScript 一起工作，Volar 创建了一个针对 Vue 的 TS 语言服务实例，将其用于 Vue 单文件组件。同时，普通的 TS 文件依然由 VSCode 内置的 TS 语言服务来处理。这也是为什么我们需要安装 TypeScript Vue Plugin 来支持在 TS 文件中引入 Vue 单文件组件。这套默认设置能够工作，但在每个项目里我们都运行了两个语言服务实例：一个来自 Volar，一个来自 VSCode 的内置服务。这在大型项目里可能会带来一些性能问题。

为了优化性能，Volar 提供了一个叫做 "Takeover 模式" 的功能。在这个模式下，Volar 能够使用一个 TS 语言服务实例同时为 Vue 和 TS 文件提供支持。

要开启 Takeover 模式，你需要执行以下步骤来**在你的项目的工作空间中**禁用 VSCode 的内置 TS 语言服务：

1. 在当前项目的工作空间下，用 `Ctrl + Shift + P` (macOS：`Cmd + Shift + P`) 唤起命令面板。

2. 输入 `built`，然后选择 "Extensions：Show Built-in Extensions"。

3. 在插件搜索框内输入 `typescript` (不要删除 `@builtin` 前缀)。

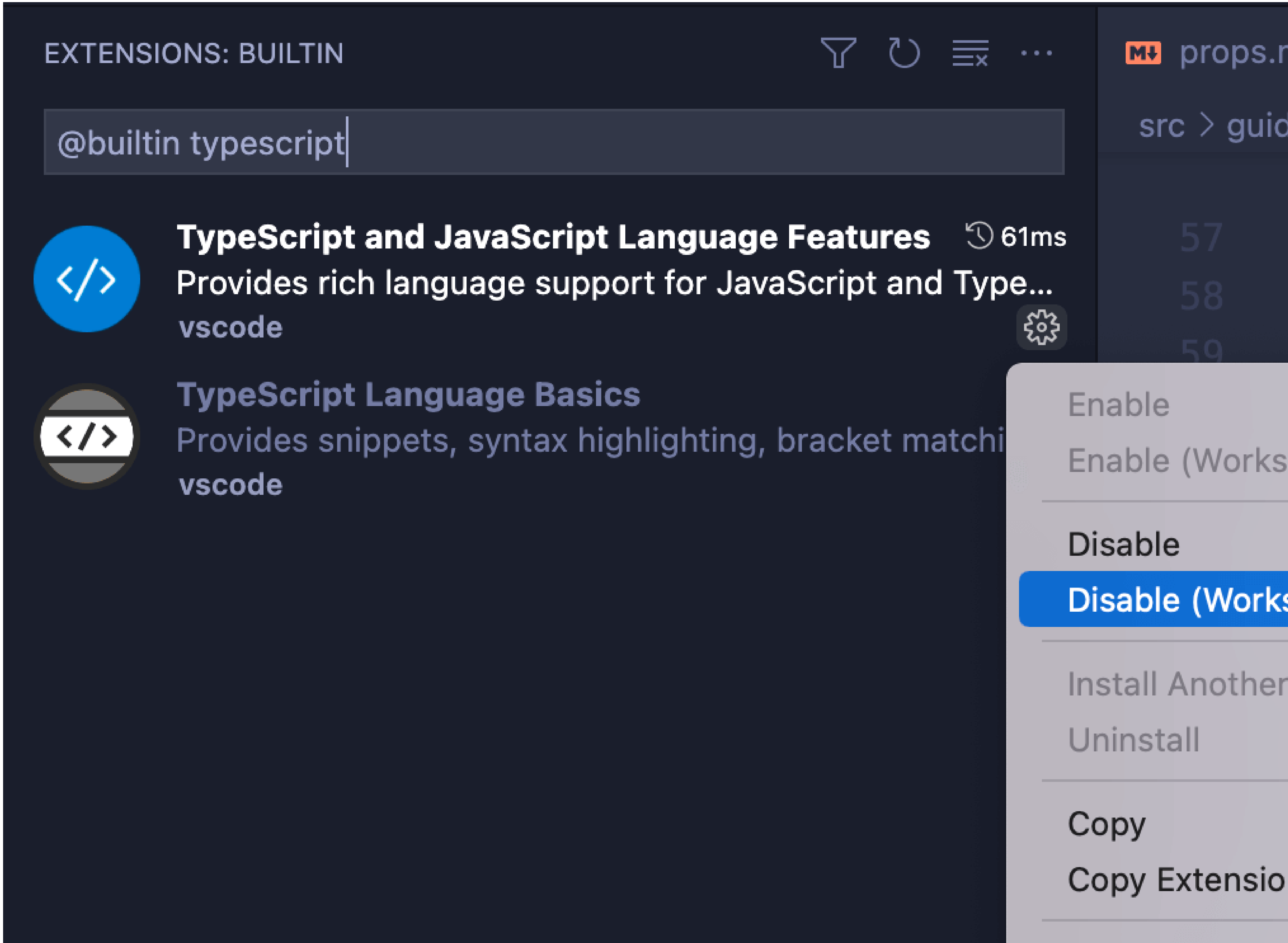
4. 点击 "TypeScript and JavaScript Language Features" 右下角的小齿轮，然

able (Workspace)”.

5. Reload the workspace. Takeover mode will be enabled when you open a Vue or TS file.

后选择 “Disable (Workspace)”。

5. 重新加载工作空间。Takeover 模式将会在你打开一个 Vue 或者 TS 文件时自动启用。



EXTENSIONS: BUILTIN

@builtin typescript

TypeScript and JavaScript Language Features 61ms
Provides rich language support for JavaScript and Type...
vscode

TypeScript Language Basics
Provides snippets, syntax highlighting, bracket matchi
vscode

Note on Vue CLI and ts-loader

In webpack-based setups such as Vue CLI, it is common to perform type checking as part of the module transform pipeline, for example with `ts-loader`. This, however, isn't a clean solution because the type system needs knowledge of the entire module graph to perform type checks. Individual module's transform step simply is not the right place for the task. It leads to the following problems:

- `ts-loader` can only type check post-transform code. This doesn't align with the errors we see in IDEs or from `vue-tsc`, which map directly back to the source code.
- Type checking can be slow. When it is performed in the same thread / process with code transformations, it significantly affects the build speed of the entire application.
- We already have type checking running right in our IDE in a separate process, so the cost of dev experience slow down simply isn't a good trade-off.

If you are currently using Vue 3 + TypeScript via Vue CLI, we strongly recommend migrating over to Vite. We are also working on CLI options to enable transpile-only TS support, so that you can switch to `vue-tsc` for type checking.

8.1.2 General Usage Notes

defineComponent()

To let TypeScript properly infer types inside component options, we need to define components with `defineComponent()`:

```js
import { defineComponent } from 'vue'
export default defineComponent({
  // 启用了类型推导
  props: {
    name: String,
    msg: { type: String, required: true }
  },
  data() {
    return {
      count: 1
```

关于 Vue CLI 和 ts-loader

像 Vue CLI 这样的基于 webpack 搭建的项目，通常是在模块编译的过程中顺道执行类型检查，例如使用 `ts-loader`。然而这并不是一个理想的解决方案，因为类型系统需要了解整个模块关系才能执行类型检查。loader 中只适合单个模块的编译，并不适合做需要全局信息的工作。这导致了下面的问题：

- `ts-loader` 只能对在它之前的 loader 编译转换后的代码执行类型检查，这和我们在 IDE 或 `vue-tsc` 中看到的基于源代码的错误提示并不一致。
- 类型检查可能会很慢。当它和代码转换在相同的线程/进程中执行时，它会显著影响整个应用的构建速度。
- 我们已经在 IDE 中通过单独的进程运行着类型检查了，却还要在构建流程中执行类型检查导致降低开发体验，这似乎不太划算。

如果你正通过 Vue CLI 使用 Vue 3 和 TypeScript，我们强烈建议你迁移到 Vite。我们也在为 CLI 开发仅执行 TS 语法转译的选项，以允许你切换至 `vue-tsc` 来执行类型检查。

8.1.2 常见使用说明

defineComponent()

为了让 TypeScript 正确地推导出组件选项内的类型，我们需要通过 `defineComponent()` 这个全局 API 来定义组件：

```js
import { defineComponent } from 'vue'
export default defineComponent({
  // 启用了类型推导
  props: {
    name: String,
    msg: { type: String, required: true }
  },
  data() {
    return {
      count: 1
    }
  },
  mounted() {
    this.name // 类型: string | undefined
    this.msg // 类型: string
    this.count // 类型: number
  }
})
```

defineComponent() also supports inferring the props passed to setup() when using Composition API without <script setup>:

```js
import { defineComponent } from 'vue'
export default defineComponent({
  // 启用了类型推导
  props: {
    message: String
  },
  setup(props) {
    props.message // 类型: string | undefined
  }
})
```

See also:

- Note on webpack Treeshaking
- type tests for defineComponent

TIP

defineComponent() also enables type inference for components defined in plain JavaScript.

Usage in Single-File Components

To use TypeScript in SFCs, add the lang="ts" attribute to <script> tags. When lang="ts" is present, all template expressions also enjoy stricter type checking.

```
    }
  },
  mounted() {
    this.name // 类型: string | undefined
    this.msg // 类型: string
    this.count // 类型: number
  }
})
```

当没有结合 <script setup> 使用组合式 API 时，defineComponent() 也支持对传递给 setup() 的 prop 的推导：

```js
import { defineComponent } from 'vue'
export default defineComponent({
  // 启用了类型推导
  props: {
    message: String
  },
  setup(props) {
    props.message // 类型: string | undefined
  }
})
```

参考：

- webpack Treeshaking 的注意事项
- 对 defineComponent 的类型测试

TIP

defineComponent() 也支持对纯 JavaScript 编写的组件进行类型推导。

在单文件组件中的用法

要在单文件组件中使用 TypeScript，需要在 <script> 标签上加上 lang="ts" 的 attribute。当 lang="ts" 存在时，所有的模板内表达式都将享受到更严格的类型

html

```html
<script lang="ts">
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      count: 1
    }
  }
})
</script>
<template>
  <!-- 启用了类型检查和自动补全 -->
  {{ count.toFixed(2) }}
</template>
```

lang="ts" can also be used with <script setup>:

html

```html
<script setup lang="ts">
// 启用了 TypeScript
import { ref } from 'vue'
const count = ref(1)
</script>
<template>
  <!-- 启用了类型检查和自动补全 -->
  {{ count.toFixed(2) }}
</template>
```

TypeScript in Templates

The <template> also supports TypeScript in binding expressions when <script lang="ts"> or <script setup lang="ts"> is used. This is useful in cases where you need to perform type casting in template expressions.

Here's a contrived example:

检查。

html

```html
<script lang="ts">
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      count: 1
    }
  }
})
</script>
<template>
  <!-- 启用了类型检查和自动补全 -->
  {{ count.toFixed(2) }}
</template>
```

lang="ts" 也可以用于 <script setup>：

html

```html
<script setup lang="ts">
// 启用了 TypeScript
import { ref } from 'vue'
const count = ref(1)
</script>
<template>
  <!-- 启用了类型检查和自动补全 -->
  {{ count.toFixed(2) }}
</template>
```

模板中的 TypeScript

在使用了 <script lang="ts"> 或 <script setup lang="ts"> 后，<template> 在绑定表达式中也支持 TypeScript。这对需要在模板表达式中执行类型转换的情况下非常有用。

这里有一个假想的例子：

```html
<script setup lang="ts">
let x: string | number = 1
</script>
<template>
  <!-- 出错，因为 x 可能是字符串 -->
  {{ x.toFixed(2) }}
</template>
```

This can be worked around with an inline type cast:

```html
<script setup lang="ts">
let x: string | number = 1
</script>
<template>
  {{ (x as number).toFixed(2) }}
</template>
```

> **TIP**
> If using Vue CLI or a webpack-based setup, TypeScript in template expressions requires `vue-loader@^16.8.0`.

Usage with TSX

Vue also supports authoring components with JSX / TSX. Details are covered in the Render Function & JSX guide.

8.1.3 Generic Components

Generic components are supported in two cases:

- In SFCs: ` `with thegeneric`‘ attribute
- Render function / JSX components: `defineComponent()`'s function signature

```html
<script setup lang="ts">
let x: string | number = 1
</script>
<template>
  <!-- 出错，因为 x 可能是字符串 -->
  {{ x.toFixed(2) }}
</template>
```

可以使用内联类型强制转换解决此问题：

```html
<script setup lang="ts">
let x: string | number = 1
</script>
<template>
  {{ (x as number).toFixed(2) }}
</template>
```

> **TIP**
> 如果正在使用 Vue CLI 或基于 webpack 的配置，支持模板内表达式的 TypeScript 需要 `vue-loader@^16.8.0`。

使用 TSX

Vue 也支持使用 JSX / TSX 编写组件。详情请查阅渲染函数 & JSX。

8.1.3 泛型组件

泛型组件支持两种使用方式：

- 在单文件组件中：在 ` 上使用 generic‘ 属性
- 渲染函数 / JSX 组件：`defineComponent()` 的函数签名

8.1.4 API-Specific Recipes

- TS with Composition API
- TS with Options API

8.2 TypeScript with Composition API

> This page assumes you've already read the overview on Using Vue with TypeScript.

8.2.1 Typing Component Props

Using <script setup>

When using `<script setup>`, the `defineProps()` macro supports inferring the props types based on its argument:

html
```html
<script setup lang="ts">
const props = defineProps({
  foo: { type: String, required: true },
  bar: Number
})
props.foo // string
props.bar // number | undefined
</script>
```

This is called ”runtime declaration”, because the argument passed to `defineProps()` will be used as the runtime `props` option.

However, it is usually more straightforward to define props with pure types via a generic type argument:

html
```html
<script setup lang="ts">
const props = defineProps<{
  foo: string
  bar?: number
```

8.1.4 特定 API 的使用指南

- TS 与组合式 API
- TS 与选项式 API

8.2 TypeScript 与组合式 API

> 这一章假设你已经阅读了搭配 TypeScript 使用 Vue 的概览。

8.2.1 为组件的 props 标注类型

使用 <script setup>

当使用 `<script setup>` 时，`defineProps()` 宏函数支持从它的参数中推导类型：

html
```html
<script setup lang="ts">
const props = defineProps({
  foo: { type: String, required: true },
  bar: Number
})
props.foo // string
props.bar // number | undefined
</script>
```

这被称之为 “运行时声明”，因为传递给 `defineProps()` 的参数会作为运行时的 `props` 选项使用。

然而，通过泛型参数来定义 props 的类型通常更直接：

html
```html
<script setup lang="ts">
const props = defineProps<{
  foo: string
  bar?: number
}>()
</script>
```

This is called "type-based declaration". The compiler will try to do its best to infer the equivalent runtime options based on the type argument. In this case, our second example compiles into the exact same runtime options as the first example.

You can use either type-based declaration OR runtime declaration, but you cannot use both at the same time.

We can also move the props types into a separate interface:

html

```
<script setup lang="ts">
interface Props {
  foo: string
  bar?: number
}
const props = defineProps<Props>()
</script>
```

语法限制 In version 3.2 and below, the generic type parameter for `defineProps()` were limited to a type literal or a reference to a local interface.

This limitation has been resolved in 3.3. The latest version of Vue supports referencing imported and a limited set of complex types in the type parameter position. However, because the type to runtime conversion is still AST-based, some complex types that require actual type analysis, e.g. conditional types, are not supported. You can use conditional types for the type of a single prop, but not the entire props object.

Props Default Values

When using type-based declaration, we lose the ability to declare default values for the props. This can be resolved by the `withDefaults` compiler macro:

ts

```
export interface Props {
  msg?: string
```

```
}>()
</script>
```

这被称之为 "基于类型的声明"。编译器会尽可能地尝试根据类型参数推导出等价的运行时选项。在这种场景下，我们第二个例子中编译出的运行时选项和第一个是完全一致的。

基于类型的声明或者运行时声明可以择一使用，但是不能同时使用。

我们也可以将 props 的类型移入一个单独的接口中：

html

```
<script setup lang="ts">
interface Props {
  foo: string
  bar?: number
}
const props = defineProps<Props>()
</script>
```

在 3.2 及以下版本中，`defineProps()` 的泛型类型参数仅限于类型文字或对本地接口的引用。

这个限制在 3.3 中得到了解决。最新版本的 Vue 支持在类型参数位置引用导入和有限的复杂类型。但是，由于类型到运行时转换仍然基于 AST，一些需要实际类型分析的复杂类型，例如条件类型，还未支持。您可以使用条件类型来指定单个 prop 的类型，但不能用于整个 props 对象的类型。

Props 解构默认值

当使用基于类型的声明时，我们失去了为 props 声明默认值的能力。这可以通过 `withDefaults` 编译器宏解决：

ts

```
    export interface Props {
      msg?: string
      labels?: string[]
```

```
  labels?: string[]
}
const props = withDefaults(defineProps<Props>(), {
  msg: 'hello',
  labels: () => ['one', 'two']
})
```

This will be compiled to equivalent runtime props `default` options. In addition, the `withDefaults` helper provides type checks for the default values, and ensures the returned `props` type has the optional flags removed for properties that do have default values declared.

Without <script setup>

If not using `<script setup>`, it is necessary to use `defineComponent()` to enable props type inference. The type of the props object passed to `setup()` is inferred from the `props` option.

ts

```
import { defineComponent } from 'vue'
export default defineComponent({
  props: {
    message: String
  },
  setup(props) {
    props.message // <-- 类型: string
  }
})
```

Complex prop types

With type-based declaration, a prop can use a complex type much like any other type:

html

```
<script setup lang="ts">
interface Book {
  title: string
  author: string
  year: number
```

```
  }
  const props = withDefaults(defineProps<Props>(), {
    msg: 'hello',
    labels: () => ['one', 'two']
  })
```

这将被编译为等效的运行时 props `default` 选项。此外，`withDefaults` 帮助程序为默认值提供类型检查，并确保返回的 props 类型删除了已声明默认值的属性的可选标志。

非 <script setup> 场景下

如果没有使用`<script setup>`,那么为了开启 props 的类型推导,必须使用`defineComponent()`。传入 `setup()` 的 props 对象类型是从 `props` 选项中推导而来。

ts

```
import { defineComponent } from 'vue'
export default defineComponent({
  props: {
    message: String
  },
  setup(props) {
    props.message // <-- 类型: string
  }
})
```

复杂的 prop 类型

通过基于类型的声明，一个 prop 可以像使用其他任何类型一样使用一个复杂类型：

html

```
<script setup lang="ts">
interface Book {
  title: string
  author: string
  year: number
```

```ts
}
const props = defineProps<{
  book: Book
}>()
</script>
```

For runtime declaration, we can use the PropType utility type:

```ts
import type { PropType } from 'vue'
const props = defineProps({
  book: Object as PropType<Book>
})
```

This works in much the same way if we're specifying the props option directly:

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'
export default defineComponent({
  props: {
    book: Object as PropType<Book>
  }
})
```

The props option is more commonly used with the Options API, so you'll find more detailed examples in the guide to TypeScript with Options API. The techniques shown in those examples also apply to runtime declarations using defineProps().

8.2.2 Typing Component Emits

In <script setup>, the emit function can also be typed using either runtime declaration OR type declaration:

```ts
<script setup lang="ts">
// 运行时
const emit = defineEmits(['change', 'update'])
// 基于类型
const emit = defineEmits<{
```

```ts
}
const props = defineProps<{
  book: Book
}>()
</script>
```

对于运行时声明，我们可以使用 PropType 工具类型：

```ts
import type { PropType } from 'vue'
const props = defineProps({
  book: Object as PropType<Book>
})
```

其工作方式与直接指定 props 选项基本相同：

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'
export default defineComponent({
  props: {
    book: Object as PropType<Book>
  }
})
```

props 选项通常用于 Options API，因此你会在选项式 API 与 TypeScript 指南中找到更详细的例子。这些例子中展示的技术也适用于使用 defineProps() 的运行时声明。

8.2.2 为组件的 emits 标注类型

在 <script setup> 中，emit 函数的类型标注也可以通过运行时声明或是类型声明进行：

```ts
<script setup lang="ts">
// 运行时
const emit = defineEmits(['change', 'update'])
// 基于类型
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()
// 3.3+: 可选的、更简洁的语法
const emit = defineEmits<{
  change: [id: number]
  update: [value: string]
}>()
</script>
```

The type argument can be one of the following:

1. A callable function type, but written as a type literal with Call Signatures. It will be used as the type of the returned `emit` function.

2. A type literal where the keys are the event names, and values are array / tuple types representing the additional accepted parameters for the event. The example above is using named tuples so each argument can have an explicit name.

As we can see, the type declaration gives us much finer-grained control over the type constraints of emitted events.

When not using `<script setup>`, `defineComponent()` is able to infer the allowed events for the `emit` function exposed on the setup context:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  emits: ['change'],
  setup(props, { emit }) {
    emit('change') // <-- 类型检查 / 自动补全
  }
})
\end{
\switchcolumn
\begin{codeTs}
import { defineComponent } from 'vue'
export default defineComponent({
  emits: ['change'],
```

```
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()
// 3.3+: 可选的、更简洁的语法
const emit = defineEmits<{
  change: [id: number]
  update: [value: string]
}>()
</script>
```

类型参数可以是以下的一种：

1. 一个可调用的函数类型，但是写作一个包含调用签名的类型字面量。它将被用作返回的 `emit` 函数的类型。

2. 一个类型字面量，其中键是事件名称，值是数组或元组类型，表示事件的附加接受参数。上面的示例使用了具名元组，因此每个参数都可以有一个显式的名称。

我们可以看到，基于类型的声明使我们可以对所触发事件的类型进行更细粒度的控制。

若没有使用 `<script setup>`，`defineComponent()` 也可以根据 `emits` 选项推导暴露在 setup 上下文中的 `emit` 函数的类型：

```
  setup(props, { emit }) {
    emit('change') // <-- 类型检查 / 自动补全
  }
})
```

8.2.3 Typing ref()

Refs infer the type from the initial value:

```ts
import { ref } from 'vue'
// 推导出的类型: Ref<number>
const year = ref(2020)
// => TS Error: Type 'string' is not assignable to type 'number'.
year.value = '2020'
```

Sometimes we may need to specify complex types for a ref's inner value. We can do that by using the `Ref` type:

```ts
import { ref } from 'vue'
import type { Ref } from 'vue'
const year: Ref<string | number> = ref('2020')
year.value = 2020 // 成功!
```

Or, by passing a generic argument when calling `ref()` to override the default inference:

```ts
// 得到的类型: Ref<string | number>
const year = ref<string | number>('2020')
year.value = 2020 // 成功!
```

If you specify a generic type argument but omit the initial value, the resulting type will be a union type that includes `undefined`:

```ts
// 推导得到的类型: Ref<number | undefined>
const n = ref<number>()
```

8.2.3 为 ref() 标注类型

ref 会根据初始化时的值推导其类型：

```ts
import { ref } from 'vue'
// 推导出的类型: Ref<number>
const year = ref(2020)
// => TS Error: Type 'string' is not assignable to type 'number'.
year.value = '2020'
```

有时我们可能想为 ref 内的值指定一个更复杂的类型，可以通过使用 `Ref` 这个类型：

```ts
import { ref } from 'vue'
import type { Ref } from 'vue'
const year: Ref<string | number> = ref('2020')
year.value = 2020 // 成功!
```

或者，在调用 `ref()` 时传入一个泛型参数，来覆盖默认的推导行为：

```ts
// 得到的类型: Ref<string | number>
const year = ref<string | number>('2020')
year.value = 2020 // 成功!
```

如果你指定了一个泛型参数但没有给出初始值，那么最后得到的就将是一个包含 `undefined` 的联合类型：

```ts
// 推导得到的类型: Ref<number | undefined>
const n = ref<number>()
```

8.2.4 Typing reactive()

`reactive()` also implicitly infers the type from its argument:

```ts
import { reactive } from 'vue'
// 推导得到的类型：{ title: string }
const book = reactive({ title: 'Vue 3 指引' })
```

To explicitly type a `reactive` property, we can use interfaces:

```ts
import { reactive } from 'vue'
interface Book {
  title: string
  year?: number
}
const book: Book = reactive({ title: 'Vue 3 指引' })
```

> **TIP**
>
> It's not recommended to use the generic argument of `reactive()` because the returned type, which handles nested ref unwrapping, is different from the generic argument type.

8.2.5 Typing computed()

`computed()` infers its type based on the getter's return value:

```ts
import { ref, computed } from 'vue'
const count = ref(0)
// 推导得到的类型：ComputedRef<number>
const double = computed(() => count.value * 2)
// => TS Error: Property 'split' does not exist on type 'number'
const result = double.value.split('')
```

You can also specify an explicit type via a generic argument:

```ts
const double = computed<number>(() => {
  // 若返回值不是 number 类型则会报错
```

8.2.4 为 reactive() 标注类型

`reactive()` 也会隐式地从它的参数中推导类型：

```ts
import { reactive } from 'vue'
// 推导得到的类型：{ title: string }
const book = reactive({ title: 'Vue 3 指引' })
```

要显式地标注一个 `reactive` 变量的类型，我们可以使用接口：

```ts
import { reactive } from 'vue'
interface Book {
  title: string
  year?: number
}
const book: Book = reactive({ title: 'Vue 3 指引' })
```

> **TIP**
>
> 不推荐使用 `reactive()` 的泛型参数，因为处理了深层次 ref 解包的返回值与泛型参数的类型不同。

8.2.5 为 computed() 标注类型

`computed()` 会自动从其计算函数的返回值上推导出类型：

```ts
import { ref, computed } from 'vue'
const count = ref(0)
// 推导得到的类型：ComputedRef<number>
const double = computed(() => count.value * 2)
// => TS Error: Property 'split' does not exist on type 'number'
const result = double.value.split('')
```

你还可以通过泛型参数显式指定类型：

```ts
const double = computed<number>(() => {
  // 若返回值不是 number 类型则会报错
})
```

8.2.6 Typing Event Handlers

When dealing with native DOM events, it might be useful to type the argument we pass to the handler correctly. Let's take a look at this example:

```html
<script setup lang="ts">
function handleChange(event) {
  // `event` 隐式地标注为 `any` 类型
  console.log(event.target.value)
}
</script>
<template>
  <input type="text" @change="handleChange" />
</template>
```

Without type annotation, the `event` argument will implicitly have a type of `any`. This will also result in a TS error if `"strict": true` or `"noImplicitAny": true` are used in `tsconfig.json`. It is therefore recommended to explicitly annotate the argument of event handlers. In addition, you may need to use type assertions when accessing the properties of `event`:

```ts
function handleChange(event: Event) {
  console.log((event.target as HTMLInputElement).value)
}
```

8.2.7 Typing Provide / Inject

Provide and inject are usually performed in separate components. To properly type injected values, Vue provides an `InjectionKey` interface, which is a generic type that extends `Symbol`. It can be used to sync the type of the injected value between the provider and the consumer:

```ts
import { provide, inject } from 'vue'
import type { InjectionKey } from 'vue'
const key = Symbol() as InjectionKey<string>
```

```
})
```

8.2.6 为事件处理函数标注类型

在处理原生 DOM 事件时，应该为我们传递给事件处理函数的参数正确地标注类型。让我们看一下这个例子：

```html
<script setup lang="ts">
function handleChange(event) {
  // `event` 隐式地标注为 `any` 类型
  console.log(event.target.value)
}
</script>
<template>
  <input type="text" @change="handleChange" />
</template>
```

没有类型标注时，这个 `event` 参数会隐式地标注为 `any` 类型。这也会在 `tsconfig.json` 中配置了 `"strict": true` 或 `"noImplicitAny": true` 时报出一个 TS 错误。因此，建议显式地为事件处理函数的参数标注类型。此外，你在访问 `event` 上的属性时可能需要使用类型断言：

```ts
function handleChange(event: Event) {
  console.log((event.target as HTMLInputElement).value)
}
```

8.2.7 为 provide / inject 标注类型

provide 和 inject 通常会在不同的组件中运行。要正确地为注入的值标记类型，Vue 提供了一个 `InjectionKey` 接口，它是一个继承自 `Symbol` 的泛型类型，可以用来在提供者和消费者之间同步注入值的类型：

```ts
import { provide, inject } from 'vue'
import type { InjectionKey } from 'vue'
const key = Symbol() as InjectionKey<string>
```

```
provide(key, 'foo') // 若提供的是非字符串值会导致错误
const foo = inject(key) // foo 的类型：string | undefined
```

It's recommended to place the injection key in a separate file so that it can be imported in multiple components.

When using string injection keys, the type of the injected value will be `unknown`, and needs to be explicitly declared via a generic type argument:

```ts
const foo = inject<string>('foo') // 类型：string | undefined
```

Notice the injected value can still be `undefined`, because there is no guarantee that a provider will provide this value at runtime.

The `undefined` type can be removed by providing a default value:

```ts
const foo = inject<string>('foo', 'bar') // 类型：string
```

If you are sure that the value is always provided, you can also force cast the value:

```ts
const foo = inject('foo') as string
```

8.2.8 Typing Template Refs

Template refs should be created with an explicit generic type argument and an initial value of `null`:

```html
<script setup lang="ts">
import { ref, onMounted } from 'vue'
const el = ref<HTMLInputElement | null>(null)
onMounted(() => {
  el.value?.focus()
})
</script>
<template>
  <input ref="el" />
</template>
```

To get the right DOM interface you can check pages like MDN.

```
provide(key, 'foo') // 若提供的是非字符串值会导致错误
const foo = inject(key) // foo 的类型：string | undefined
```

建议将注入 key 的类型放在一个单独的文件中，这样它就可以被多个组件导入。

当使用字符串注入 key 时，注入值的类型是 `unknown`，需要通过泛型参数显式声明：

```ts
const foo = inject<string>('foo') // 类型：string | undefined
```

注意注入的值仍然可以是 `undefined`，因为无法保证提供者一定会在运行时 provide 这个值。

当提供了一个默认值后，这个 `undefined` 类型就可以被移除：

```ts
const foo = inject<string>('foo', 'bar') // 类型：string
```

如果你确定该值将始终被提供，则还可以强制转换该值：

```ts
const foo = inject('foo') as string
```

8.2.8 为模板引用标注类型

模板引用需要通过一个显式指定的泛型参数和一个初始值 `null` 来创建：

```html
<script setup lang="ts">
import { ref, onMounted } from 'vue'
const el = ref<HTMLInputElement | null>(null)
onMounted(() => {
  el.value?.focus()
})
</script>
<template>
  <input ref="el" />
</template>
```

可以通过类似于 MDN 的页面来获取正确的 DOM 接口。

Note that for strict type safety, it is necessary to use optional chaining or type guards when accessing `el.value`. This is because the initial ref value is `null` until the component is mounted, and it can also be set to `null` if the referenced element is unmounted by `v-if`.

8.2.9 Typing Component Template Refs

Sometimes you might need to annotate a template ref for a child component in order to call its public method. For example, we have a `MyModal` child component with a method that opens the modal:

html

```html
<!-- MyModal.vue -->
<script setup lang="ts">
import { ref } from 'vue'
const isContentShown = ref(false)
const open = () => (isContentShown.value = true)
defineExpose({
  open
})
</script>
```

In order to get the instance type of `MyModal`, we need to first get its type via `typeof`, then use TypeScript's built-in `InstanceType` utility to extract its instance type:

html

```html
<!-- App.vue -->
<script setup lang="ts">
import MyModal from './MyModal.vue'
const modal = ref<InstanceType<typeof MyModal> | null>(null)
const openModal = () => {
  modal.value?.open()
}
</script>
```

Note if you want to use this technique in TypeScript files instead of Vue SFCs, you need to enable Volar's Takeover Mode.

In cases where the exact type of the component isn't available or isn't important, `ComponentPublicInstance` can be used instead. This will only include properties that are shared by all components, such as

注意为了严格的类型安全，有必要在访问 `el.value` 时使用可选链或类型守卫。这是因为直到组件被挂载前，这个 ref 的值都是初始的 `null`，并且在由于 `v-if` 的行为将引用的元素卸载时也可以被设置为 `null`。

8.2.9 为组件模板引用标注类型

有时，你可能需要为一个子组件添加一个模板引用，以便调用它公开的方法。举例来说，我们有一个 `MyModal` 子组件，它有一个打开模态框的方法：

html

```html
<!-- MyModal.vue -->
<script setup lang="ts">
import { ref } from 'vue'
const isContentShown = ref(false)
const open = () => (isContentShown.value = true)
defineExpose({
  open
})
</script>
```

为了获取 `MyModal` 的类型，我们首先需要通过 `typeof` 得到其类型，再使用 TypeScript 内置的 `InstanceType` 工具类型来获取其实例类型：

html

```html
<!-- App.vue -->
<script setup lang="ts">
import MyModal from './MyModal.vue'
const modal = ref<InstanceType<typeof MyModal> | null>(null)
const openModal = () => {
  modal.value?.open()
}
</script>
```

注意，如果你想在 TypeScript 文件而不是在 Vue SFC 中使用这种技巧，需要开启 Volar 的 Takeover 模式。

如果组件的具体类型无法获得，或者你并不关心组件的具体类型，那么可以使用 `ComponentPublicInstance`。这只会包含所有组件都共享的属性，比如 `$el`。

$el:

```ts
import { ref } from 'vue'
import type { ComponentPublicInstance } from 'vue'
const child = ref<ComponentPublicInstance | null>(null)
```

8.3 TypeScript with Composition API

> This page assumes you've already read the overview on Using Vue with TypeScript.

TIP
While Vue does support TypeScript usage with Options API, it is recommended to use Vue with TypeScript via Composition API as it offers simpler, more efficient and more robust type inference.

8.3.1 Typing Component Props

Type inference for props in Options API requires wrapping the component with `defineComponent()`. With it, Vue is able to infer the types for the props based on the `props` option, taking additional options such as `required: true` and `default` into account:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  // 启用了类型推导
  props: {
    name: String,
    id: [Number, String],
    msg: { type: String, required: true },
    metadata: null
  },
  mounted() {
    this.name // 类型: string | undefined
    this.id // 类型: number | string | undefined
    this.msg // 类型: string
    this.metadata // 类型: any
```

```ts
import { ref } from 'vue'
import type { ComponentPublicInstance } from 'vue'
const child = ref<ComponentPublicInstance | null>(null)
```

8.3 TypeScript 与选项式 API

> 这一章假设你已经阅读了搭配 TypeScript 使用 Vue 的概览。

TIP
虽然 Vue 的确支持在选项式 API 中使用 TypeScript，但在使用 TypeScript 的前提下更推荐使用组合式 API，因为它提供了更简单、高效和可靠的类型推导。

8.3.1 为组件的 props 标注类型

选项式 API 中对 props 的类型推导需要用 `defineComponent()` 来包装组件。有了它，Vue 才可以通过 `props` 以及一些额外的选项，比如 `required: true` 和 `default` 来推导出 props 的类型：

```ts
import { defineComponent } from 'vue'
export default defineComponent({
    // 启用了类型推导
    props: {
    name: String,
    id: [Number, String],
    msg: { type: String, required: true },
    metadata: null
    },
    mounted() {
    this.name // 类型: string | undefined
    this.id // 类型: number | string | undefined
    this.msg // 类型: string
    this.metadata // 类型: any
```

```
  }
})
```

However, the runtime `props` options only support using constructor functions as a prop's type - there is no way to specify complex types such as objects with nested properties or function call signatures.

To annotate complex props types, we can use the `PropType` utility type:

ts

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'
interface Book {
  title: string
  author: string
  year: number
}
export default defineComponent({
  props: {
    book: {
      // 提供相对 `Object` 更确定的类型
      type: Object as PropType<Book>,
      required: true
    },
    // 也可以标记函数
    callback: Function as PropType<(id: number) => void>
  },
  mounted() {
    this.book.title // string
    this.book.year // number
    // TS Error: argument of type 'string' is not
    // assignable to parameter of type 'number'
    this.callback?.('123')
  }
})
```

```
  }
})
```

然而，这种运行时 `props` 选项仅支持使用构造函数来作为一个 prop 的类型——没有办法指定多层级对象或函数签名之类的复杂类型。

我们可以使用 `PropType` 这个工具类型来标记更复杂的 props 类型：

ts

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'
interface Book {
  title: string
  author: string
  year: number
}
export default defineComponent({
  props: {
    book: {
      // 提供相对 `Object` 更确定的类型
      type: Object as PropType<Book>,
      required: true
    },
    // 也可以标记函数
    callback: Function as PropType<(id: number) => void>
  },
  mounted() {
    this.book.title // string
    this.book.year // number
    // TS Error: argument of type 'string' is not
    // assignable to parameter of type 'number'
    this.callback?.('123')
  }
})
```

Caveats

If your TypeScript version is less than `4.7`, you have to be careful when using function values for `validator` and `default` prop options - make sure to use arrow functions:

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'
interface Book {
  title: string
  year?: number
}
export default defineComponent({
  props: {
    bookA: {
      type: Object as PropType<Book>,
      // 如果你的 TypeScript 版本低于 4.7，确保使用箭头函数
      default: () => ({
        title: 'Arrow Function Expression'
      }),
      validator: (book: Book) => !!book.title
    }
  }
})
```

This prevents TypeScript from having to infer the type of `this` inside these functions, which, unfortunately, can cause the type inference to fail. It was a previous design limitation, and now has been improved in TypeScript 4.7.

8.3.2 Typing Component Emits

We can declare the expected payload type for an emitted event using the object syntax of the `emits` option. Also, all non-declared emitted events will throw a type error when called:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  emits: {
```

注意事项

如果你的 TypeScript 版本低于 `4.7`，在使用函数作为 prop 的 `validator` 和 `default` 选项值时需要格外小心——确保使用箭头函数：

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'
interface Book {
  title: string
  year?: number
}
export default defineComponent({
  props: {
    bookA: {
      type: Object as PropType<Book>,
      // 如果你的 TypeScript 版本低于 4.7，确保使用箭头函数
      default: () => ({
        title: 'Arrow Function Expression'
      }),
      validator: (book: Book) => !!book.title
    }
  }
})
```

这会防止 TypeScript 将 `this` 根据函数内的环境作出不符合我们期望的类型推导。这是之前版本的一个设计限制，不过现在已经在 TypeScript 4.7 中解决了。

8.3.2 为组件的 emits 标注类型

我们可以给 `emits` 选项提供一个对象来声明组件所触发的事件，以及这些事件所期望的参数类型。试图触发未声明的事件会抛出一个类型错误：

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  emits: {
    addBook(payload: { bookName: string }) {
      // 执行运行时校验
      return payload.bookName.length > 0
    }
  },
  methods: {
    onSubmit() {
      this.$emit('addBook', {
        bookName: 123 // 类型错误
      })
      this.$emit('non-declared-event') // 类型错误
    }
  }
})
```

8.3.3 Typing Computed Properties

A computed property infers its type based on its return value:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      message: 'Hello!'
    }
  },
  computed: {
    greeting() {
      return this.message + '!'
    }
  },
  mounted() {
    this.greeting // 类型: string
  }
})
```

```ts
    addBook(payload: { bookName: string }) {
      // 执行运行时校验
      return payload.bookName.length > 0
    }
  },
  methods: {
    onSubmit() {
      this.$emit('addBook', {
        bookName: 123 // 类型错误
      })
      this.$emit('non-declared-event') // 类型错误
    }
  }
})
```

8.3.3 为计算属性标记类型

计算属性会自动根据其返回值来推导其类型：

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      message: 'Hello!'
    }
  },
  computed: {
    greeting() {
      return this.message + '!'
    }
  },
  mounted() {
    this.greeting // 类型: string
  }
})
```

In some cases, you may want to explicitly annotate the type of a computed property to ensure its implementation is correct:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      message: 'Hello!'
    }
  },
  computed: {
    // 显式标注返回类型
    greeting(): string {
      return this.message + '!'
    },
    // 标注一个可写的计算属性
    greetingUppercased: {
      get(): string {
        return this.greeting.toUpperCase()
      },
      set(newValue: string) {
        this.message = newValue.toUpperCase()
      }
    }
  }
})
```

Explicit annotations may also be required in some edge cases where TypeScript fails to infer the type of a computed property due to circular inference loops.

8.3.4 Typing Event Handlers

When dealing with native DOM events, it might be useful to type the argument we pass to the handler correctly. Let’s take a look at this example:

在某些场景中，你可能想要显式地标记出计算属性的类型以确保其实现是正确的：

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  data() {
    return {
      message: 'Hello!'
    }
  },
  computed: {
    // 显式标注返回类型
    greeting(): string {
      return this.message + '!'
    },
    // 标注一个可写的计算属性
    greetingUppercased: {
      get(): string {
        return this.greeting.toUpperCase()
      },
      set(newValue: string) {
        this.message = newValue.toUpperCase()
      }
    }
  }
})
```

在某些 TypeScript 因循环引用而无法推导类型的情况下，可能必须进行显式的类型标注。

8.3.4 为事件处理函数标注类型

在处理原生 DOM 事件时，应该为我们传递给事件处理函数的参数正确地标注类型。让我们看一下这个例子：

```html
<script lang="ts">
import { defineComponent } from 'vue'
export default defineComponent({
  methods: {
    handleChange(event) {
      // `event` 隐式地标注为 `any` 类型
      console.log(event.target.value)
    }
  }
})
</script>
<template>
  <input type="text" @change="handleChange" />
</template>
```

Without type annotation, the `event` argument will implicitly have a type of `any`. This will also result in a TS error if `"strict": true` or `"noImplicitAny": true` are used in `tsconfig.json`. It is therefore recommended to explicitly annotate the argument of event handlers. In addition, you may need to use type assertions when accessing the properties of `event`:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  methods: {
    handleChange(event: Event) {
      console.log((event.target as HTMLInputElement).value)
    }
  }
})
```

8.3.5 Augmenting Global Properties

Some plugins install globally available properties to all component instances via `app.config.globalProperties`. For example, we may install `this.$http` for data-fetching or `this.$translate` for internationalization. To make this play well with TypeScript, Vue exposes a `ComponentCustomProperties` interface

```html
<script lang="ts">
import { defineComponent } from 'vue'
export default defineComponent({
  methods: {
    handleChange(event) {
      // `event` 隐式地标注为 `any` 类型
      console.log(event.target.value)
    }
  }
})
</script>
<template>
  <input type="text" @change="handleChange" />
</template>
```

没有类型标注时，这个 `event` 参数会隐式地标注为 `any` 类型。这也会在 `tsconfig.json` 中配置了 `"strict": true` 或 `"noImplicitAny": true` 时抛出一个 TS 错误。因此，建议显式地为事件处理函数的参数标注类型。此外，在访问 `event` 上的属性时你可能需要使用类型断言：

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  methods: {
    handleChange(event: Event) {
      console.log((event.target as HTMLInputElement).value)
    }
  }
})
```

8.3.5 扩展全局属性

某些插件会通过 `app.config.globalProperties` 为所有组件都安装全局可用的属性。举例来说，我们可能为了请求数据而安装了 `this.$http`，或者为了国际化而安装了 `this.$translate`。为了使 TypeScript 更好地支持这个行为，Vue 暴露了一

designed to be augmented via TypeScript module augmentation:

ts
```ts
import axios from 'axios'
declare module 'vue' {
  interface ComponentCustomProperties {
    $http: typeof axios
    $translate: (key: string) => string
  }
}
```

See also:

- TypeScript unit tests for component type extensions

Type Augmentation Placement

We can put this type augmentation in a `.ts` file, or in a project-wide `*.d.ts` file. Either way, make sure it is included in `tsconfig.json`. For library / plugin authors, this file should be specified in the `types` property in `package.json`.

In order to take advantage of module augmentation, you will need to ensure the augmentation is placed in a TypeScript module. That is to say, the file needs to contain at least one top-level `import` or `export`, even if it is just `export {}`. If the augmentation is placed outside of a module, it will overwrite the original types rather than augmenting them!

ts
```ts
// 不工作，将覆盖原始类型。
declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
  }
}
```

ts
```ts
// 正常工作。
export {}
declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
```

个被设计为可以通过 TypeScript 模块扩展来扩展的 `ComponentCustomProperties` 接口：

ts
```ts
import axios from 'axios'
declare module 'vue' {
  interface ComponentCustomProperties {
    $http: typeof axios
    $translate: (key: string) => string
  }
}
```

参考：

- 对组件类型扩展的 TypeScript 单元测试

类型扩展的位置

我们可以将这些类型扩展放在一个 `.ts` 文件，或是一个影响整个项目的 `*.d.ts` 文件中。无论哪一种，都应确保在 `tsconfig.json` 中包括了此文件。对于库或插件作者，这个文件应该在 `package.json` 的 `types` 属性中被列出。

为了利用模块扩展的优势，你需要确保将扩展的模块放在 TypeScript 模块 中。也就是说，该文件需要包含至少一个顶级的 `import` 或 `export`，即使它只是 `export {}`。如果扩展被放在模块之外，它将覆盖原始类型，而不是扩展!

ts
```ts
// 不工作，将覆盖原始类型。
declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
  }
}
```

ts
```ts
// 正常工作。
export {}
declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
  }
}
```

8.3.6 Augmenting Custom Options

Some plugins, for example vue-router, provide support for custom component options such as beforeRouteEnter:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  beforeRouteEnter(to, from, next) {
    // ...
  }
})
```

Without proper type augmentation, the arguments of this hook will implicitly have any type. We can augment the ComponentCustomOptions interface to support these custom options:

```ts
import { Route } from 'vue-router'
declare module 'vue' {
  interface ComponentCustomOptions {
    beforeRouteEnter?(to: Route, from: Route, next: () => void): void
  }
}
```

Now the beforeRouteEnter option will be properly typed. Note this is just an example - well-typed libraries like vue-router should automatically perform these augmentations in their own type definitions.

The placement of this augmentation is subject the same restrictions as global property augmentations.

See also:

- TypeScript unit tests for component type extensions

```
  }
}
```

8.3.6 扩展自定义选项

某些插件,比如 vue-router,提供了一些自定义的组件选项,比如 beforeRouteEnter:

```ts
import { defineComponent } from 'vue'
export default defineComponent({
  beforeRouteEnter(to, from, next) {
    // ...
  }
})
```

如果没有确切的类型标注，这个钩子函数的参数会隐式地标注为 any 类型。我们可以为 ComponentCustomOptions 接口扩展自定义的选项来支持：

```ts
import { Route } from 'vue-router'
declare module 'vue' {
  interface ComponentCustomOptions {
    beforeRouteEnter?(to: Route, from: Route, next: () => void): void
  }
}
```

现在这个 beforeRouteEnter 选项会被准确地标注类型。注意这只是一个例子——像 vue-router 这种类型完备的库应该在它们自己的类型定义中自动执行这些扩展。

这种类型扩展和全局属性扩展受到相同的限制。

参考：

- 对组件类型扩展的 TypeScript 单元测试

第九章　Extra Topics

9.1 Ways of Using Vue

We believe there is no ”one size fits all” story for the web. This is why Vue is designed to be flexible and incrementally adoptable. Depending on your use case, Vue can be used in different ways to strike the optimal balance between stack complexity, developer experience and end performance.

9.1.1 Standalone Script

Vue can be used as a standalone script file - no build step required! If you have a backend framework already rendering most of the HTML, or your frontend logic isn’t complex enough to justify a build step, this is the easiest way to integrate Vue into your stack. You can think of Vue as a more declarative replacement of jQuery in such cases.

Vue also provides an alternative distribution called petite-vue that is specifically optimized for progressively enhancing existing HTML. It has a smaller feature set, but is extremely lightweight and uses an implementation that is more efficient in no-build-step scenarios.

9.1.2 Embedded Web Components

You can use Vue to build standard Web Components that can be embedded in any HTML page, regardless of how they are rendered. This option allows you to leverage Vue in a completely consumer-agnostic fashion: the resulting web components can be embedded in legacy applications, static HTML, or even applications built with other frameworks.

9.1 使用 Vue 的多种方式

我们相信在 Web 的世界里没有一种方案可以解决所有问题。正因如此，Vue 被设计成一个灵活的、可以渐进式集成的框架。根据使用场景的不同需要，相应地有多种不同的方式来使用 Vue，以此在技术栈复杂度、开发体验和性能表现间取得最佳平衡。

9.1.1 独立脚本

Vue 可以以一个单独 JS 文件的形式使用，无需构建步骤！如果你的后端框架已经渲染了大部分的 HTML，或者你的前端逻辑并不复杂，不需要构建步骤，这是最简单的使用 Vue 的方式。在这些场景中你可以将 Vue 看作一个更加声明式的 jQuery 替代品。

Vue 也提供了另一个更适用于此类无构建步骤场景的版本 petite-vue。它为渐进式增强已有的 HTML 作了特别的优化，功能更加精简，十分轻量。

9.1.2 作为 Web Component 嵌入

你可以用 Vue 来构建标准的 Web Component，这些 Web Component 可以嵌入到任何 HTML 页面中，无论它们是如何被渲染的。这个方式让你能够在不需要顾虑最终使用场景的情况下使用 Vue：因为生成的 Web Component 可以嵌入到旧应用、静态 HTML，甚至用其他框架构建的应用中。

9.1.3 Single-Page Application (SPA)

Some applications require rich interactivity, deep session depth, and non-trivial stateful logic on the frontend. The best way to build such applications is to use an architecture where Vue not only controls the entire page, but also handles data updates and navigation without having to reload the page. This type of application is typically referred to as a Single-Page Application (SPA).

Vue provides core libraries and comprehensive tooling support with amazing developer experience for building modern SPAs, including:

- Client-side router
- Blazing fast build tool chain
- IDE support
- Browser devtools
- TypeScript integrations
- Testing utilities

SPAs typically require the backend to expose API endpoints - but you can also pair Vue with solutions like Inertia.js to get the SPA benefits while retaining a server-centric development model.

9.1.4 Fullstack / SSR

Pure client-side SPAs are problematic when the app is sensitive to SEO and time-to-content. This is because the browser will receive a largely empty HTML page, and has to wait until the JavaScript is loaded before rendering anything.

Vue provides first-class APIs to ”render” a Vue app into HTML strings on the server. This allows the server to send back already-rendered HTML, allowing end users to see the content immediately while the JavaScript is being downloaded. Vue will then ”hydrate” the application on the client side to make it interactive. This is called Server-Side Rendering (SSR) and it greatly improves Core Web Vital metrics such as Largest Contentful Paint (LCP).

There are higher-level Vue-based frameworks built on top of this paradigm, such as Nuxt, which allow you to develop a fullstack application using Vue and JavaScript.

9.1.3 单页面应用 (SPA)

一些应用在前端需要具有丰富的交互性、较深的会话和复杂的状态逻辑。构建这类应用的最佳方法是使用这样一种架构：Vue 不仅控制整个页面，还负责处理抓取新数据，并在无需重新加载的前提下处理页面切换。这种类型的应用通常称为单页应用 (Single-Page application，缩写为 SPA)。

Vue 提供了核心功能库和全面的工具链支持，为现代 SPA 提供了极佳的开发体验，覆盖以下方面：

- 客户端路由
- 极其快速的构建工具
- IDE 支持
- 浏览器开发工具
- TypeScript 支持
- 测试工具

SPA 一般要求后端提供 API 数据接口，但你也可以将 Vue 和如 Inertia.js 之类的解决方案搭配使用，在保留侧重服务端的开发模型的同时获得 SPA 的益处。

9.1.4 全栈 / SSR

纯客户端的 SPA 在首屏加载和 SEO 方面有显著的问题，因为浏览器会收到一个巨大的 HTML 空页面，只有等到 JavaScript 加载完毕才会渲染出内容。

Vue 提供了一系列 API，支持将一个 Vue 应用在服务端渲染成 HTML 字符串。这能让服务器直接返回渲染好的 HTML，让用户在 JavaScript 下载完毕前就看到页面内容。Vue 之后会在客户端对应用进行 “激活 (hydrate)” 使其重获可交互性。这被称为服务端渲染 (SSR)，它能够极大地改善应用在 Web 核心指标上的性能表现，如最大内容绘制 (LCP)。

Vue 生态中有一些针对此类场景的、基于 Vue 的上层框架，比如 NuxtJS，能让你用 Vue 和 JavaScript 开发一个全栈应用。

9.1.5 JAMStack / SSG

Server-side rendering can be done ahead of time if the required data is static. This means we can pre-render an entire application into HTML and serve them as static files. This improves site performance and makes deployment a lot simpler since we no longer need to dynamically render pages on each request. Vue can still hydrate such applications to provide rich interactivity on the client. This technique is commonly referred to as Static-Site Generation (SSG), also known as JAMStack.

There are two flavors of SSG: single-page and multi-page. Both flavors pre-render the site into static HTML, the difference is that:

- After the initial page load, a single-page SSG ”hydrates” the page into an SPA. This requires more upfront JS payload and hydration cost, but subsequent navigations will be faster, since it only needs to partially update the page content instead of reloading the entire page.
- A multi-page SSG loads a new page on every navigation. The upside is that it can ship minimal JS - or no JS at all if the page requires no interaction! Some multi-page SSG frameworks such as Astro also support ”partial hydration” - which allows you to use Vue components to create interactive ”islands” inside static HTML.

Single-page SSGs are better suited if you expect non-trivial interactivity, deep session lengths, or persisted elements / state across navigations. Otherwise, multi-page SSG would be the better choice.

The Vue team also maintains a static-site generator called VitePress, which powers this website you are reading right now! VitePress supports both flavors of SSG. Nuxt also supports SSG. You can even mix SSR and SSG for different routes in the same Nuxt app.

9.1.6 Beyond the Web

Although Vue is primarily designed for building web applications, it is by no means limited to just the browser. You can:

- Build desktop apps with Electron or Tauri
- Build mobile apps with Ionic Vue
- Build desktop and mobile apps from the same codebase with Quasar
- Use Vue’s Custom Renderer API to build custom renderers targeting WebGL or even the

9.1.5 JAMStack / SSG

如果所需的数据是静态的，那么服务端渲染可以提前完成。这意味着我们可以将整个应用预渲染为 HTML，并将其作为静态文件部署。这增强了站点的性能表现，也使部署变得更容易，因为我们无需根据请求动态地渲染页面。Vue 仍可通过激活在客户端提供交互。这一技术通常被称为静态站点生成 (SSG)，也被称为 JAMStack。

SSG 有两种风格：单页和多页。这两种风格都能将站点预渲染为静态 HTML，区别在于：

- 单页 SSG 在初始页面加载后将其 “激活” 为 SPA。这需要更多的前期 JS 加载和激活成本，但后续的导航将更快，因为它只需要部分地更新页面内容，而无需重新加载整个页面。
- 多页 SSG 每次导航都会加载一个新页面。好处是它可以仅需最少的 JS——或者如果页面无需交互则根本不需要 JS！一些多页面 SSG 框架，如 Astro 也支持 “部分激活”——它允许你通过 Vue 组件在静态 HTML 中创建交互式的 “孤岛”。

单页 SSG 更适合于重交互、深会话的场景，或需要在导航之间持久化元素或状态。否则，多页 SSG 将是更好的选择。

Vue 团队也维护了一个名为 VitePress 的静态站点生成器，你正在阅读的文档就是基于它构建的！VitePress 支持两种形式的 SSG。另外，NuxtJS 也支持 SSG。你甚至可以在同一个 Nuxt 应用中通过不同的路由提供 SSR 和 SSG。

9.1.6 Web 之外...

尽管 Vue 主要是为构建 Web 应用而设计的，但它绝不仅仅局限于浏览器。你还可以：

- 配合 Electron 或 Tauri 构建桌面应用
- 配合 Ionic Vue 构建移动端应用
- 使用 Quasar 用同一套代码同时开发桌面端和移动端应用
- 使用 Vue 的自定义渲染 API 来构建不同目标的渲染器，比如 WebGL 甚至

terminal!

9.2 Composition API FAQ

TIP
This FAQ assumes prior experience with Vue - in particular, experience with Vue 2 while primarily using Options API.

9.2.1 What is Composition API?

Composition API is a set of APIs that allows us to author Vue components using imported functions instead of declaring options. It is an umbrella term that covers the following APIs:

- Reactivity API, e.g. `ref()` and `reactive()`, that allows us to directly create reactive state, computed state, and watchers.
- Lifecycle Hooks, e.g. `onMounted()` and `onUnmounted()`, that allow us to programmatically hook into the component lifecycle.
- Dependency Injection, i.e. `provide()` and `inject()`, that allow us to leverage Vue's dependency injection system while using Reactivity APIs.

Composition API is a built-in feature of Vue 3 and Vue 2.7. For older Vue 2 versions, use the officially maintained `@vue/composition-api` plugin. In Vue 3, it is also primarily used together with the “ syntax in Single-File Components. Here's a basic example of a component using Composition API:

```html
<script setup>
import { ref, onMounted } from 'vue'
// 响应式状态
const count = ref(0)
// 更改状态、触发更新的函数
function increment() {
  count.value++
}
// 生命周期钩子
```

是终端命令行！

9.2 组合式 API 常见问答

TIP
这个 FAQ 假定你已经有一些使用 Vue 的经验，特别是用选项式 API 使用 Vue 2 的经验。

9.2.1 什么是组合式 API？

组合式 API (Composition API) 是一系列 API 的集合，使我们可以使用函数而不是声明明选项的方式书写 Vue 组件。它是一个概括性的术语，涵盖了以下方面的 API：

- 响应式 API：例如 `ref()` 和 `reactive()`，使我们可以直接创建响应式状态、计算属性和侦听器。
- 生命周期钩子：例如 `onMounted()` 和 `onUnmounted()`，使我们可以在组件各个生命周期阶段添加逻辑。
- 依赖注入：例如 `provide()` 和 `inject()`，使我们可以在使用响应式 API 时，利用 Vue 的依赖注入系统。

组合式 API 是 Vue 3 及 Vue 2.7 的内置功能。对于更老的 Vue 2 版本，可以使用官方维护的插件 `@vue/composition-api`。在 Vue 3 中，组合式 API 基本上都会配合 “ 语法在单文件组件中使用。下面是一个使用组合式 API 的组件示例：

```html
<script setup>
import { ref, onMounted } from 'vue'
// 响应式状态
const count = ref(0)
// 更改状态、触发更新的函数
function increment() {
  count.value++
}
// 生命周期钩子
```

```
onMounted(() => {
  console.log(`计数器初始值为 ${count.value}。`)
})
</script>
<template>
  <button @click="increment"> 点击了：{{ count }} 次 </button>
</template>
```

Despite an API style based on function composition, **Composition API is NOT functional programming**. Composition API is based on Vue's mutable, fine-grained reactivity paradigm, whereas functional programming emphasizes immutability.

If you are interested in learning how to use Vue with Composition API, you can set the site-wide API preference to Composition API using the toggle at the top of the left sidebar, and then go through the guide from the beginning.

9.2.2 Why Composition API?

Better Logic Reuse

The primary advantage of Composition API is that it enables clean, efficient logic reuse in the form of Composable functions. It solves all the drawbacks of mixins, the primary logic reuse mechanism for Options API.

Composition API's logic reuse capability has given rise to impressive community projects such as VueUse, an ever-growing collection of composable utilities. It also serves as a clean mechanism for easily integrating stateful third-party services or libraries into Vue's reactivity system, for example immutable data, state machines, and RxJS.

More Flexible Code Organization

Many users love that we write organized code by default with Options API: everything has its place based on the option it falls under. However, Options API poses serious limitations when a single component's logic grows beyond a certain complexity threshold. This limitation is particularly prominent in components that need to deal with multiple **logical concerns**, which we have witnessed first hand in many production Vue 2 apps.

```
onMounted(() => {
  console.log(`计数器初始值为 ${count.value}。`)
})
</script>
<template>
  <button @click="increment"> 点击了：{{ count }} 次 </button>
</template>
```

虽然这套 API 的风格是基于函数的组合，但**组合式 API 并不是函数式编程**。组合式 API 是以 Vue 中数据可变的、细粒度的响应性系统为基础的，而函数式编程通常强调数据不可变。

如果你对如何通过组合式 API 使用 Vue 感兴趣，可以通过页面左侧边栏上方的开关将 API 偏好切换到组合式 API，然后重新从头阅读指引。

9.2.2 为什么要有组合式 API?

更好的逻辑复用

组合式 API 最基本的优势是它使我们能够通过组合函数来实现更加简洁高效的逻辑复用。在选项式 API 中我们主要的逻辑复用机制是 mixins，而组合式 API 解决了 mixins 的所有缺陷。

组合式 API 提供的逻辑复用能力孵化了一些非常棒的社区项目，比如 VueUse，一个不断成长的工具型组合式函数集合。组合式 API 还为其他第三方状态管理库与 Vue 的响应式系统之间的集成提供了一套简洁清晰的机制，例如不可变数据、状态机与 RxJS。

更灵活的代码组织

许多用户喜欢选项式 API 的原因是它在默认情况下就能够让人写出有组织的代码：大部分代码都自然地被放进了对应的选项里。然而，选项式 API 在单个组件的逻辑复杂到一定程度时，会面临一些无法忽视的限制。这些限制主要体现在需要处理多个**逻辑关注点**的组件中，这是我们在许多 Vue 2 的实际案例中所观察到的。

Take the folder explorer component from Vue CLI's GUI as an example: this component is responsible for the following logical concerns:

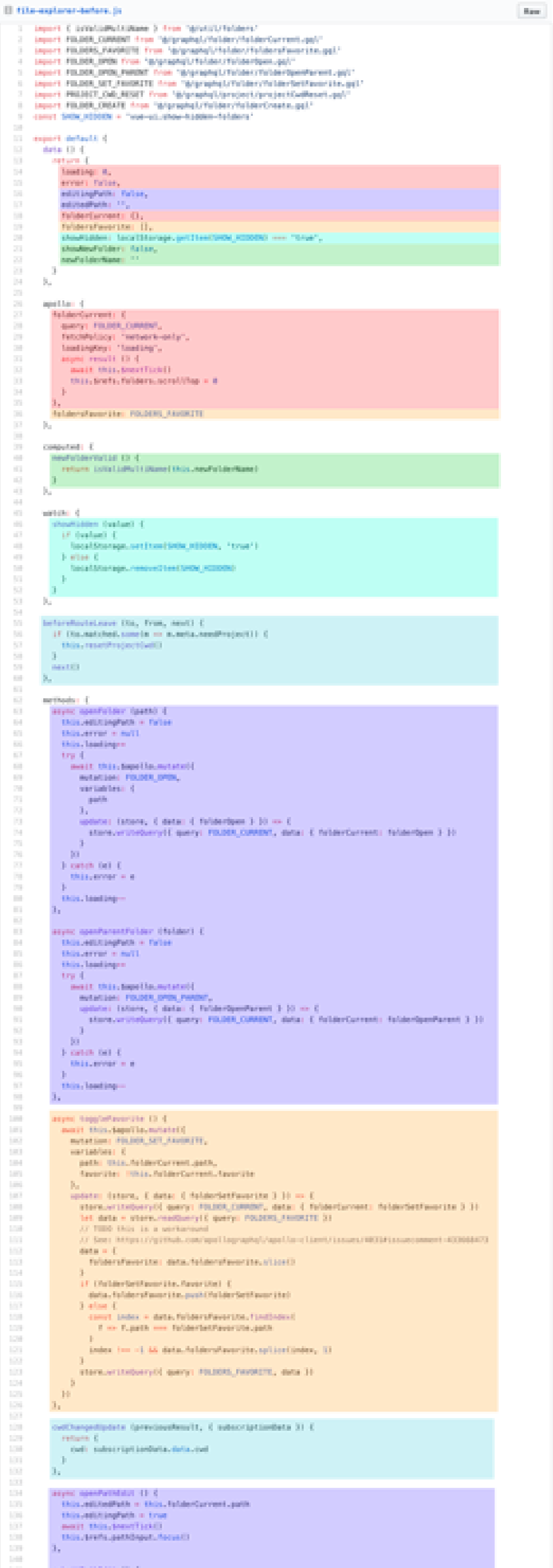
- Tracking current folder state and displaying its content
- Handling folder navigation (opening, closing, refreshing...)
- Handling new folder creation
- Toggling show favorite folders only
- Toggling show hidden folders
- Handling current working directory changes

The original version of the component was written in Options API. If we give each line of code a color based on the logical concern it is dealing with, this is how it looks:

我们以 Vue CLI GUI 中的文件浏览器组件为例：这个组件承担了以下几个逻辑关注点：

- 追踪当前文件夹的状态，展示其内容
- 处理文件夹的相关操作 (打开、关闭和刷新)
- 支持创建新文件夹
- 可以切换到只展示收藏的文件夹
- 可以开启对隐藏文件夹的展示
- 处理当前工作目录中的变更

这个组件最原始的版本是由选项式 API 写成的。如果我们为相同的逻辑关注点标上一种颜色，那将会是这样：



file-explorer-before.js　Raw

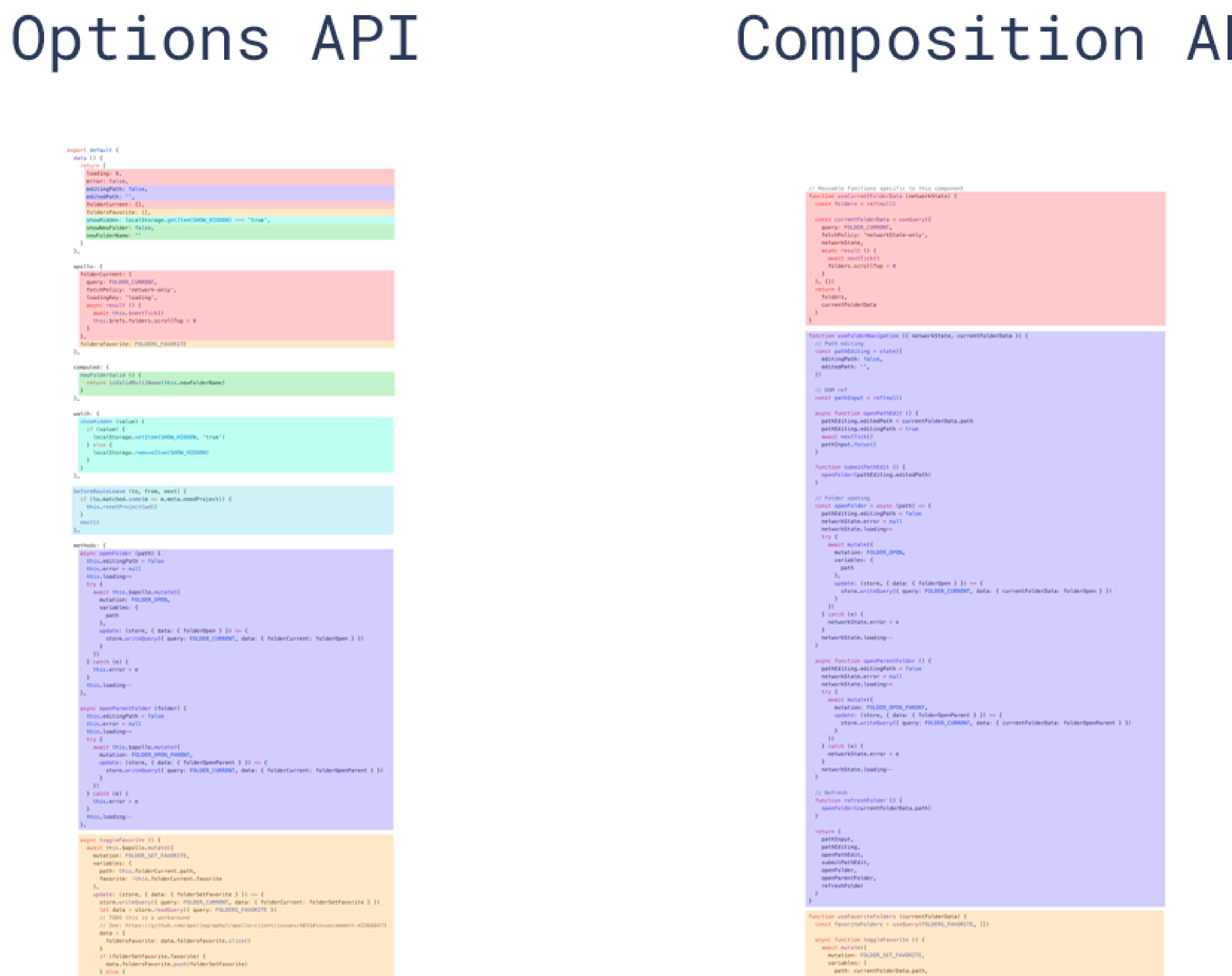
```js
import { isValidMultiName } from '@/util/folders'
import FOLDER_CURRENT from '@/graphql/folder/folderCurrent.gql'
import FOLDERS_FAVORITE from '@/graphql/folder/foldersFavorite.gql'
import FOLDER_OPEN from '@/graphql/folder/folderOpen.gql'
import FOLDER_OPEN_PARENT from '@/graphql/folder/folderOpenParent.gql'
import FOLDER_SET_FAVORITE from '@/graphql/folder/folderSetFavorite.gql'
import PROJECT_CWD_RESET from '@/graphql/project/projectCwdReset.gql'
import FOLDER_CREATE from '@/graphql/folder/folderCreate.gql'
const SHOW_HIDDEN = 'vue-ui.show-hidden-folders'

export default {
  data () {
    return {
      loading: 0,
      error: false,
      editingPath: false,
      editedPath: '',
      folderCurrent: {},
      foldersFavorite: [],
      showHidden: localStorage.getItem(SHOW_HIDDEN) === 'true',
      showNewFolder: false,
      newFolderName: ''
    }
  },

  apollo: {
    folderCurrent: {
      query: FOLDER_CURRENT,
      fetchPolicy: 'network-only',
      loadingKey: 'loading',
      async result () {
        await this.$nextTick()
        this.$refs.folders.scrollTop = 0
      }
    },
    foldersFavorite: FOLDERS_FAVORITE
  },

  computed: {
    newFolderValid () {
      return isValidMultiName(this.newFolderName)
    }
  },

  watch: {
    showHidden (value) {
      if (value) {
        localStorage.setItem(SHOW_HIDDEN, 'true')
      } else {
        localStorage.removeItem(SHOW_HIDDEN)
      }
    }
  },

  beforeRouteLeave (to, from, next) {
    if (to.matched.some(m => m.meta.needProject)) {
      this.resetProjectCwd()
    }
    next()
  },

  methods: {
    async openFolder (path) {
      this.editingPath = false
      this.error = null
      this.loading++
      try {
        await this.$apollo.mutate({
          mutation: FOLDER_OPEN,
          variables: {
            path
          },
          update: (store, { data: { folderOpen } }) => {
            store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderOpen } })
          }
        })
      } catch (e) {
        this.error = e
      }
      this.loading--
    },

    async openParentFolder (folder) {
      this.editingPath = false
      this.error = null
      this.loading++
      try {
        await this.$apollo.mutate({
          mutation: FOLDER_OPEN_PARENT,
          update: (store, { data: { folderOpenParent } }) => {
            store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderOpenParent } })
          }
        })
      } catch (e) {
        this.error = e
      }
      this.loading--
    },

    async toggleFavorite () {
      await this.$apollo.mutate({
        mutation: FOLDER_SET_FAVORITE,
        variables: {
          path: this.folderCurrent.path,
          favorite: !this.folderCurrent.favorite
        },
        update: (store, { data: { folderSetFavorite } }) => {
          store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderSetFavorite } })
          let data = store.readQuery({ query: FOLDERS_FAVORITE })
          // TODO this is a workaround
          // See: https://github.com/apollographql/apollo-client/issues/4031#issuecomment-433668473
          data = {
            foldersFavorite: data.foldersFavorite.slice()
          }
          if (folderSetFavorite.favorite) {
            data.foldersFavorite.push(folderSetFavorite)
          } else {
            const index = data.foldersFavorite.findIndex(
              f => f.path === folderSetFavorite.path
            )
            index !== -1 && data.foldersFavorite.splice(index, 1)
          }
          store.writeQuery({ query: FOLDERS_FAVORITE, data })
        }
      })
    },

    cwdChangedUpdate (previousResult, { subscriptionData }) {
      return {
        cwd: subscriptionData.data.cwd
      }
    },

    async openPathEdit () {
      this.editedPath = this.folderCurrent.path
      this.editingPath = true
      await this.$nextTick()
      this.$refs.pathInput.focus()
    },

```

Notice how code dealing with the same logical concern is forced to be split under different options, located in different parts of the file. In a component that is several hundred lines long, understanding and navigating a single logical concern requires constantly scrolling up and down the file, making it much more difficult than it should be. In addition, if we ever intend to extract a logical concern into a reusable utility, it takes quite a bit of work to find and extract the right pieces of code from different parts of the file.

Here's the same component, before and after the refactor into Composition API:

你可以看到，处理相同逻辑关注点的代码被强制拆分在了不同的选项中，位于文件的不同部分。在一个几百行的大组件中，要读懂代码中的一个逻辑关注点，需要在文件中反复上下滚动，这并不理想。另外，如果我们想要将一个逻辑关注点抽取重构到一个可复用的工具函数中，需要从文件的多个不同部分找到所需的正确片段。

而如果用组合式 API 重构这个组件，将会变成下面右边这样：



Options API

```
export default {
  data () {
    return {
      loading: 0,
      error: false,
      editingPath: false,
      editedPath: '',
      folderCurrent: {},
      foldersFavorite: [],
      showHidden: localStorage.getItem(SHOW_HIDDEN) === 'true',
      showNewFolder: false,
      newFolderName: ''
    }
  },

  apollo: {
    folderCurrent: {
      query: FOLDER_CURRENT,
      fetchPolicy: 'network-only',
      loadingKey: 'loading',
      async result () {
        await this.$nextTick()
        this.$refs.folders.scrollTop = 0
      }
    },
    foldersFavorite: FOLDERS_FAVORITE
  },

  computed: {
    newFolderValid () {
      return isValidMultiName(this.newFolderName)
    }
  },

  watch: {
    showHidden (value) {
      if (value) {
        localStorage.setItem(SHOW_HIDDEN, 'true')
      } else {
        localStorage.removeItem(SHOW_HIDDEN)
      }
    }
  },

  beforeRouteLeave (to, from, next) {
    if (to.matched.some(m => m.meta.needProject)) {
      this.resetProjectCwd()
    }
    next()
  },

  methods: {
    async openFolder (path) {
      this.editingPath = false
      this.error = null
      this.loading++
      try {
        await this.$apollo.mutate({
          mutation: FOLDER_OPEN,
          variables: {
            path
          },
          update: (store, { data: { folderOpen } }) => {
            store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderOpen } })
          }
        })
      } catch (e) {
        this.error = e
      }
      this.loading--
    },

    async openParentFolder (folder) {
      this.editingPath = false
      this.error = null
      this.loading++
      try {
        await this.$apollo.mutate({
          mutation: FOLDER_OPEN_PARENT,
          update: (store, { data: { folderOpenParent } }) => {
            store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderOpenParent } })
          }
        })
      } catch (e) {
        this.error = e
      }
      this.loading--
    },

    async toggleFavorite () {
      await this.$apollo.mutate({
        mutation: FOLDER_SET_FAVORITE,
        variables: {
          path: this.folderCurrent.path,
          favorite: !this.folderCurrent.favorite
        },
        update: (store, { data: { folderSetFavorite } }) => {
          store.writeQuery({ query: FOLDER_CURRENT, data: { folderCurrent: folderSetFavorite } })
          let data = store.readQuery({ query: FOLDERS_FAVORITE })
          // TODO this is a workaround
          // See: https://github.com/apollographql/apollo-client/issues/4031#issuecomment-433668473
          data = {
            foldersFavorite: data.foldersFavorite.slice()
          }
          if (folderSetFavorite.favorite) {
            data.foldersFavorite.push(folderSetFavorite)
          } else {
```

Composition AI

```
// Reusable functions specific to this component
function useCurrentFolderData (networkState) {
  const folders = ref(null)

  const currentFolderData = useQuery({
    query: FOLDER_CURRENT,
    fetchPolicy: 'networkState-only',
    networkState,
    async result () {
      await nextTick()
      folders.scrollTop = 0
    }
  }, {})
  return {
    folders,
    currentFolderData
  }
}

function useFolderNavigation ({ networkState, currentFolderData }) {
  // Path editing
  const pathEditing = state({
    editingPath: false,
    editedPath: '',
  })

  // DOM ref
  const pathInput = ref(null)

  async function openPathEdit () {
    pathEditing.editedPath = currentFolderData.path
    pathEditing.editingPath = true
    await nextTick()
    pathInput.focus()
  }

  function submitPathEdit () {
    openFolder(pathEditing.editedPath)
  }

  // Folder opening
  const openFolder = async (path) => {
    pathEditing.editingPath = false
    networkState.error = null
    networkState.loading++
    try {
      await mutate({
        mutation: FOLDER_OPEN,
        variables: {
          path
        },
        update: (store, { data: { folderOpen } }) => {
          store.writeQuery({ query: FOLDER_CURRENT, data: { currentFolderData: folderOpen } })
        }
      })
    } catch (e) {
      networkState.error = e
    }
    networkState.loading--
  }

  async function openParentFolder () {
    pathEditing.editingPath = false
    networkState.error = null
    networkState.loading++
    try {
      await mutate({
        mutation: FOLDER_OPEN_PARENT,
        update: (store, { data: { folderOpenParent } }) => {
          store.writeQuery({ query: FOLDER_CURRENT, data: { currentFolderData: folderOpenParent } })
        }
      })
    } catch (e) {
      networkState.error = e
    }
    networkState.loading--
  }

  // Refresh
  function refreshFolder () {
    openFolder(currentFolderData.path)
  }

  return {
    pathInput,
    pathEditing,
    openPathEdit,
    submitPathEdit,
    openFolder,
    openParentFolder,
    refreshFolder
  }
}

function useFavoriteFolders (currentFolderData) {
  const favoriteFolders = useQuery(FOLDERS_FAVORITE, [])

  async function toggleFavorite () {
    await mutate({
      mutation: FOLDER_SET_FAVORITE,
      variables: {
        path: currentFolderData.path,
```

Notice how the code related to the same logical concern can now be grouped together: we no longer need to jump between different options blocks while working on a specific logical concern. Moreover, we can now move a group of code into an external file with minimal effort, since we no longer need to shuffle the code around in order to extract them. This reduced friction for refactoring is key to the long-term maintainability in large codebases.

Better Type Inference

In recent years, more and more frontend developers are adopting TypeScript as it helps us write more robust code, make changes with more confidence, and provides a great development experience with IDE support. However, the Options API, originally conceived in 2013, was designed without type inference in mind. We had to implement some absurdly complex type gymnastics to make type inference work with the Options API. Even with all this effort, type inference for Options API can still break down for mixins and dependency injection.

This had led many developers who wanted to use Vue with TS to lean towards Class API powered by `vue-class-component`. However, a class-based API heavily relies on ES decorators, a language feature that was only a stage 2 proposal when Vue 3 was being developed in 2019. We felt it was too risky to base an official API on an unstable proposal. Since then, the decorators proposal has gone through yet another complete overhaul, and finally reached stage 3 in 2022. In addition, class-based API suffers from logic reuse and organization limitations similar to Options API.

In comparison, Composition API utilizes mostly plain variables and functions, which are naturally type friendly. Code written in Composition API can enjoy full type inference with little need for manual type hints. Most of the time, Composition API code will look largely identical in TypeScript and plain JavaScript. This also makes it possible for plain JavaScript users to benefit from partial type inference.

Smaller Production Bundle and Less Overhead

Code written in Composition API and `<script setup>` is also more efficient and minification-friendly than Options API equivalent. This is because the template in a `<script setup>` component is compiled as a function inlined in the same scope of the `<script setup>` code. Unlike property access from `this`, the compiled template code can directly access variables declared inside `<script setup>`, without an instance proxy in between. This also leads to better minification because all

现在与同一个逻辑关注点相关的代码被归为了一组：我们无需再为了一个逻辑关注点在不同的选项块间来回滚动切换。此外，我们现在可以很轻松地将这一组代码移动到一个外部文件中，不再需要为了抽象而重新组织代码，大大降低了重构成本，这在长期维护的大型项目中非常关键。

更好的类型推导

近几年来，越来越多的开发者开始使用 TypeScript 书写更健壮可靠的代码，TypeScript 还提供了非常好的 IDE 开发支持。然而选项式 API 是在 2013 年被设计出来的，那时并没有把类型推导考虑进去，因此我们不得不做了一些复杂到夸张的类型体操才实现了对选项式 API 的类型推导。但尽管做了这么多的努力，选项式 API 的类型推导在处理 mixins 和依赖注入类型时依然不甚理想。

因此，很多想要搭配 TS 使用 Vue 的开发者采用了由 `vue-class-component` 提供的 Class API。然而，基于 Class 的 API 非常依赖 ES 装饰器，在 2019 年我们开始开发 Vue 3 时，它仍是一个仅处于 stage 2 的语言功能。我们认为基于一个不稳定的语言提案去设计框架的核心 API 风险实在太大了，因此没有继续向 Class API 的方向发展。在那之后装饰器提案果然又发生了很大的变动，在 2022 年才终于到达 stage 3。另一个问题是，基于 Class 的 API 和选项式 API 在逻辑复用和代码组织方面存在相同的限制。

相比之下，组合式 API 主要利用基本的变量和函数，它们本身就是类型友好的。用组合式 API 重写的代码可以享受到完整的类型推导，不需要书写太多类型标注。大多数时候，用 TypeScript 书写的组合式 API 代码和用 JavaScript 写都差不太多！这也让许多纯 JavaScript 用户也能从 IDE 中享受到部分类型推导功能。

更小的生产包体积

搭配 `<script setup>` 使用组合式 API 比等价情况下的选项式 API 更高效，对代码压缩也更友好。这是由于 `<script setup>` 形式书写的组件模板被编译为了一个内联函数，和 `<script setup>` 中的代码位于同一作用域。不像选项式 API 需要依赖 `this` 上下文对象访问属性，被编译的模板可以直接访问 `<script setup>` 中定义的变量，无需从实例中代理。这对代码压缩更友好，因为本地变量的名字可

the variable names can be safely shortened.

9.2.3 Relationship with Options API

Trade-offs

Some users moving from Options API found their Composition API code less organized, and concluded that Composition API is ”worse” in terms of code organization. We recommend users with such opinions to look at that problem from a different perspective.

It is true that Composition API no longer provides the ”guard rails” that guide you to put your code into respective buckets. In return, you get to author component code like how you would write normal JavaScript. This means **you can and should apply any code organization best practices to your Composition API code as you would when writing normal JavaScript**. If you can write well-organized JavaScript, you should also be able to write well-organized Composition API code.

Options API does allow you to ”think less” when writing component code, which is why many users love it. However, in reducing the mental overhead, it also locks you into the prescribed code organization pattern with no escape hatch, which can make it difficult to refactor or improve code quality in larger scale projects. In this regard, Composition API provides better long term scalability.

Does Composition API cover all use cases?

Yes in terms of stateful logic. When using Composition API, there are only a few options that may still be needed: `props`, `emits`, `name`, and `inheritAttrs`.

> **TIP**
> Since 3.3 you can directly use `defineOptions` in `<script setup>` to set the component name or `inheritAttrs` property

If you intend to exclusively use Composition API (along with the options listed above), you can shave a few kbs off your production bundle via a compile-time flag that drops Options API related code from Vue. Note this also affects Vue components in your dependencies.

以被压缩，但对象的属性名则不能。

9.2.3 与选项式 API 的关系

取舍

一些从选项式 API 迁移来的用户发现，他们的组合式 API 代码缺乏组织性，并得出了组合式 API 在代码组织方面 “更糟糕” 的结论。我们建议持有这类观点的用户换个角度思考这个问题。

组合式 API 不像选项式 API 那样会手把手教你该把代码放在哪里。但反过来，它却让你可以像编写普通的 JavaScript 那样来编写组件代码。这意味着**你能够，并且应该在写组合式 API 的代码时也运用上所有普通 JavaScript 代码组织的最佳实践**。如果你可以编写组织良好的 JavaScript，你也应该有能力编写组织良好的组合式 API 代码。

选项式 API 确实允许你在编写组件代码时 “少思考”，这是许多用户喜欢它的原因。然而，在减少费神思考的同时，它也将你锁定在规定的代码组织模式中，没有摆脱的余地，这会导致在更大规模的项目中难以进行重构或提高代码质量。在这方面，组合式 API 提供了更好的长期可维护性。

组合式 API 是否覆盖了所有场景？

组合式 API 能够覆盖所有状态逻辑方面的需求。除此之外，只需要用到一小部分选项：`props`，`emits`，`name` 和 `inheritAttrs`。

> **TIP**
> 从 3.3 开始你可以直接通过 `<script setup>` 中的 `defineOptions` 来设置组件名或 `inheritAttrs` 属性。

如果你在代码中只使用了组合式 API (以及上述必需的选项)，那么你可以通过配置编译时标记来去掉 Vue 运行时中针对选项式 API 支持的代码，从而减小生产包大概几 kb 左右的体积。注意这个配置也会影响你依赖中的 Vue 组件。

Can I use both APIs in the same component?

Yes. You can use Composition API via the `setup()` option in an Options API component.

However, we only recommend doing so if you have an existing Options API codebase that needs to integrate with new features / external libraries written with Composition API.

Will Options API be deprecated?

No, we do not have any plan to do so. Options API is an integral part of Vue and the reason many developers love it. We also realize that many of the benefits of Composition API only manifest in larger-scale projects, and Options API remains a solid choice for many low-to-medium-complexity scenarios.

9.2.4 Relationship with Class API

We no longer recommend using Class API with Vue 3, given that Composition API provides great TypeScript integration with additional logic reuse and code organization benefits.

9.2.5 Comparison with React Hooks

Composition API provides the same level of logic composition capabilities as React Hooks, but with some important differences.

React Hooks are invoked repeatedly every time a component updates. This creates a number of caveats that can confuse even seasoned React developers. It also leads to performance optimization issues that can severely affect development experience. Here are some examples:

- Hooks are call-order sensitive and cannot be conditional.
- Variables declared in a React component can be captured by a hook closure and become ”stale” if the developer fails to pass in the correct dependencies array. This leads to React developers relying on ESLint rules to ensure correct dependencies are passed. However, the rule is often not smart enough and over-compensates for correctness, which leads to unnecessary invalidation and headaches when edge cases are encountered.
- Expensive computations require the use of `useMemo`, which again requires manually passing in

可以在同一个组件中使用两种 API 吗？

可以。你可以在一个选项式 API 的组件中通过 `setup()` 选项来使用组合式 API。

然而，我们只推荐你在一个已经基于选项式 API 开发了很久、但又需要和基于组合式 API 的新代码或是第三方库整合的项目中这样做。

选项式 API 会被废弃吗？

不会，我们没有任何计划这样做。选项式 API 也是 Vue 不可分割的一部分，也有很多开发者喜欢它。我们也意识到组合式 API 更适用于大型的项目，而对于中小型项目来说选项式 API 仍然是一个不错的选择。

9.2.4 与 Class API 的关系

我们不再推荐在 Vue 3 中使用 Class API，因为组合式 API 提供了很好的 TypeScript 集成，并具有额外的逻辑重用和代码组织优势。

9.2.5 和 React Hooks 的对比

组合式 API 提供了和 React Hooks 相同级别的逻辑组织能力，但它们之间有着一些重要的区别。

React Hooks 在组件每次更新时都会重新调用。这就产生了一些即使是经验丰富的 React 开发者也会感到困惑的问题。这也带来了一些性能问题，并且相当影响开发体验。例如：

- Hooks 有严格的调用顺序，并不可以写在条件分支中。
- React 组件中定义的变量会被一个钩子函数闭包捕获，若开发者传递了错误的依赖数组，它会变得 “过期”。这导致了 React 开发者非常依赖 ESLint 规则以确保传递了正确的依赖，然而，这些规则往往不够智能，保持正确的代价过高，在一些边缘情况时会遇到令人头疼的、不必要的报错信息。
- 昂贵的计算需要使用 `useMemo`，这也需要传入正确的依赖数组。
- 在默认情况下，传递给子组件的事件处理函数会导致子组件进行不必要的更

the correct dependencies array.

- Event handlers passed to child components cause unnecessary child updates by default, and require explicit `useCallback` as an optimization. This is almost always needed, and again requires a correct dependencies array. Neglecting this leads to over-rendering apps by default and can cause performance issues without realizing it.
- The stale closure problem, combined with Concurrent features, makes it difficult to reason about when a piece of hooks code is run, and makes working with mutable state that should persist across renders (via `useRef`) cumbersome.

In comparison, Vue Composition API:

- Invokes `setup()` or `<script setup>` code only once. This makes the code align better with the intuitions of idiomatic JavaScript usage as there are no stale closures to worry about. Composition API calls are also not sensitive to call order and can be conditional.
- Vue's runtime reactivity system automatically collects reactive dependencies used in computed properties and watchers, so there's no need to manually declare dependencies.
- No need to manually cache callback functions to avoid unnecessary child updates. In general, Vue's fine-grained reactivity system ensures child components only update when they need to. Manual child-update optimizations are rarely a concern for Vue developers.

We acknowledge the creativity of React Hooks, and it is a major source of inspiration for Composition API. However, the issues mentioned above do exist in its design and we noticed Vue's reactivity model happens to provide a way around them.

9.3 Reactivity in Depth

One of Vue's most distinctive features is the unobtrusive reactivity system. Component state consists of reactive JavaScript objects. When you modify them, the view updates. It makes state management simple and intuitive, but it's also important to understand how it works to avoid some common gotchas. In this section, we are going to dig into some of the lower-level details of Vue's reactivity system.

9.3.1 What is Reactivity?

新。子组件默认更新，并需要显式的调用 `useCallback` 作优化。这个优化同样需要正确的依赖数组，并且几乎在任何时候都需要。忽视这一点会导致默认情况下对应用进行过度渲染，并可能在不知不觉中导致性能问题。

- 要解决变量闭包导致的问题，再结合并发功能，使得很难推理出一段钩子代码是什么时候运行的，并且很不好处理需要在多次渲染间保持引用 (通过 `useRef`) 的可变状态。

相比起来，Vue 的组合式 API：

- 仅调用 `setup()` 或 `<script setup>` 的代码一次。这使得代码更符合日常 JavaScript 的直觉，不需要担心闭包变量的问题。组合式 API 也并不限制调用顺序，还可以有条件地进行调用。
- Vue 的响应性系统运行时会自动收集计算属性和侦听器的依赖，因此无需手动声明依赖。
- 无需手动缓存回调函数来避免不必要的组件更新。Vue 细粒度的响应性系统能够确保在绝大部分情况下组件仅执行必要的更新。对 Vue 开发者来说几乎不怎么需要对子组件更新进行手动优化。

我们承认 React Hooks 的创造性，它是组合式 API 的一个主要灵感来源。然而，它的设计也确实存在上面提到的问题，而 Vue 的响应性模型恰好提供了一种解决这些问题的方法。

9.3 深入响应式系统

Vue 最标志性的功能就是其低侵入性的响应式系统。组件状态都是由响应式的 JavaScript 对象组成的。当更改它们时，视图会随即自动更新。这让状态管理更加简单直观，但理解它是如何工作的也是很重要的，这可以帮助我们避免一些常见的陷阱。在本节中，我们将深入研究 Vue 响应性系统的一些底层细节。

9.3.1 什么是响应性

This term comes up in programming quite a bit these days, but what do people mean when they say it? Reactivity is a programming paradigm that allows us to adjust to changes in a declarative manner. The canonical example that people usually show, because it's a great one, is an Excel spreadsheet:

Here cell A2 is defined via a formula of `= A0 + A1` (you can click on A2 to view or edit the formula), so the spreadsheet gives us 3. No surprises there. But if you update A0 or A1, you'll notice that A2 automagically updates too.

JavaScript doesn't usually work like this. If we were to write something comparable in JavaScript:

js

```js
let A0 = 1
let A1 = 2
let A2 = A0 + A1
console.log(A2) // 3
A0 = 2
console.log(A2) // 仍然是 3
```

When we mutate `A0`, `A2` does not change automatically.

So how would we do this in JavaScript? First, in order to re-run the code that updates `A2`, let's wrap it in a function:

js

```js
let A2
function update() {
  A2 = A0 + A1
}
```

Then, we need to define a few terms:

- The `update()` function produces a **side effect**, or **effect** for short, because it modifies the state of the program.
- `A0` and `A1` are considered **dependencies** of the effect, as their values are used to perform the effect. The effect is said to be a **subscriber** to its dependencies.

What we need is a magic function that can invoke `update()` (the **effect**) whenever `A0` or `A1` (the **dependencies**) change:

js

```js
whenDepsChange(update)
```

这个术语在今天的各种编程讨论中经常出现，但人们说它的时候究竟是想表达什么意思呢？本质上，响应性是一种可以使我们声明式地处理变化的编程范式。一个经常被拿来当作典型例子的用例即是 Excel 表格：

这里单元格 A2 中的值是通过公式 `= A0 + A1` 来定义的（你可以在 A2 上点击来查看或编辑该公式），因此最终得到的值为 3，正如所料。但如果你试着更改 A0 或 A1，你会注意到 A2 也随即自动更新了。

而 JavaScript 默认并不是这样的。如果我们用 JavaScript 写类似的逻辑：

js

```js
let A0 = 1
let A1 = 2
let A2 = A0 + A1
console.log(A2) // 3
A0 = 2
console.log(A2) // 仍然是 3
```

当我们更改 `A0` 后，`A2` 不会自动更新。

那么我们如何在 JavaScript 中做到这一点呢？首先，为了能重新运行计算的代码来更新 `A2`，我们需要将其包装为一个函数：

js

```js
let A2
function update() {
  A2 = A0 + A1
}
```

然后，我们需要定义几个术语：

- 这个 `update()` 函数会产生一个**副作用**，或者就简称为**作用** (effect)，因为它会更改程序里的状态。
- `A0` 和 `A1` 被视为这个作用的**依赖** (dependency)，因为它们的值被用来执行这个作用。因此这次作用也可以说是一个它依赖的**订阅者** (subscriber)。

我们需要一个魔法函数，能够在 `A0` 或 `A1` (这两个**依赖**) 变化时调用 `update()` (产生**作用**)。

js

```js
whenDepsChange(update)
```

This `whenDepsChange()` function has the following tasks:

1. Track when a variable is read. E.g. when evaluating the expression `A0 + A1`, both `A0` and `A1` are read.
2. If a variable is read when there is a currently running effect, make that effect a subscriber to that variable. E.g. because `A0` and `A1` are read when `update()` is being executed, `update()` becomes a subscriber to both `A0` and `A1` after the first call.
3. Detect when a variable is mutated. E.g. when `A0` is assigned a new value, notify all its subscriber effects to re-run.

9.3.2 How Reactivity Works in Vue

We can't really track the reading and writing of local variables like in the example. There's just no mechanism for doing that in vanilla JavaScript. What we **can** do though, is intercept the reading and writing of **object properties**.

There are two ways of intercepting property access in JavaScript: getter / setters and Proxies. Vue 2 used getter / setters exclusively due to browser support limitations. In Vue 3, Proxies are used for reactive objects and getter / setters are used for refs. Here's some pseudo-code that illustrates how they work:

js

```
function reactive(obj) {
  return new Proxy(obj, {
    get(target, key) {
      track(target, key)
      return target[key]
    },
    set(target, key, value) {
      target[key] = value
      trigger(target, key)
    }
  })
}
function ref(value) {
  const refObject = {
    get value() {
```

这个 `whenDepsChange()` 函数有如下的任务：

1. 当一个变量被读取时进行追踪。例如我们执行了表达式 `A0 + A1` 的计算，则 `A0` 和 `A1` 都被读取到了。
2. 如果一个变量在当前运行的副作用中被读取了，就将该副作用设为此变量的一个订阅者。例如由于 `A0` 和 `A1` 在 `update()` 执行时被访问到了，则 `update()` 需要在第一次调用之后成为 `A0` 和 `A1` 的订阅者。
3. 探测一个变量的变化。例如当我们给 `A0` 赋了一个新的值后，应该通知其所有订阅了的副作用重新执行。

9.3.2 Vue 中的响应性是如何工作的

我们无法直接追踪对上述示例中局部变量的读写，原生 JavaScript 没有提供任何机制能做到这一点。**但是**，我们是可以追踪**对象属性**的读写的。

在 JavaScript 中有两种劫持 property 访问的方式：getter / setters 和 Proxies。Vue 2 使用 getter / setters 完全是出于支持旧版本浏览器的限制。而在 Vue 3 中则使用了 Proxy 来创建响应式对象，仅将 getter / setter 用于 ref。下面的伪代码将会说明它们是如何工作的：

js

```
function reactive(obj) {
  return new Proxy(obj, {
    get(target, key) {
      track(target, key)
      return target[key]
    },
    set(target, key, value) {
      target[key] = value
      trigger(target, key)
    }
  })
}
function ref(value) {
  const refObject = {
    get value() {
      track(refObject, 'value')
      return value
    },
    set value(newValue) {
      value = newValue
      trigger(refObject, 'value')
    }
  }
  return refObject
}
```

TIP
Code snippets here and below are meant to explain the core concepts in the simplest form possible, so many details are omitted, and edge cases ignored.

This explains a few limitations of reactive objects that we have discussed in the fundamentals section:

- When you assign or destructure a reactive object's property to a local variable, accessing or assigning to that variable is non-reactive because it no longer triggers the get / set proxy traps on the source object. Note this "disconnect" only affects the variable binding - if the variable points to a non-primitive value such as an object, mutating the object would still be reactive.
- The returned proxy from `reactive()`, although behaving just like the original, has a different identity if we compare it to the original using the `===` operator.

Inside `track()`, we check whether there is a currently running effect. If there is one, we lookup the subscriber effects (stored in a Set) for the property being tracked, and add the effect to the Set:

js
```
// 这会在一个副作用就要运行之前被设置
// 我们会在后面处理它
let activeEffect
function track(target, key) {
  if (activeEffect) {
    const effects = getSubscribersForProperty(target, key)
    effects.add(activeEffect)
```

```
      track(refObject, 'value')
      return value
    },
    set value(newValue) {
      value = newValue
      trigger(refObject, 'value')
    }
  }
  return refObject
}
```

TIP
这里和下面的代码片段皆旨在以最简单的形式解释核心概念，因此省略了许多细节和边界情况。

以上代码解释了我们在基础章节部分讨论过的一些 `reactive()` 的局限性：

- 当你将一个响应式对象的属性赋值或解构到一个本地变量时，访问或赋值该变量是非响应式的，因为它将不再触发源对象上的 get / set 代理。注意这种“断开”只影响变量绑定——如果变量指向一个对象之类的非原始值，那么对该对象的修改仍然是响应式的。
- 从 `reactive()` 返回的代理尽管行为上表现得像原始对象，但我们通过使用 `===` 运算符还是能够比较出它们的不同。

在 `track()` 内部，我们会检查当前是否有正在运行的副作用。如果有，我们会查找到一个存储了所有追踪了该属性的订阅者的 Set，然后将当前这个副作用作为新订阅者添加到该 Set 中。

js

```
// 这会在一个副作用就要运行之前被设置
// 我们会在后面处理它
let activeEffect
function track(target, key) {
  if (activeEffect) {
    const effects = getSubscribersForProperty(target, key)
    effects.add(activeEffect)
  }
}
```

Effect subscriptions are stored in a global `WeakMap<target, Map<key, Set<effect>>>` data structure. If no subscribing effects Set was found for a property (tracked for the first time), it will be created. This is what the `getSubscribersForProperty()` function does, in short. For simplicity, we will skip its details.

Inside `trigger()`, we again lookup the subscriber effects for the property. But this time we invoke them instead:

```js
function trigger(target, key) {
  const effects = getSubscribersForProperty(target, key)
  effects.forEach((effect) => effect())
}
```

Now let's circle back to the `whenDepsChange()` function:

```js
function whenDepsChange(update) {
  const effect = () => {
    activeEffect = effect
    update()
    activeEffect = null
  }
  effect()
}
```

It wraps the raw `update` function in an effect that sets itself as the current active effect before running the actual update. This enables `track()` calls during the update to locate the current active effect.

At this point, we have created an effect that automatically tracks its dependencies, and re-runs whenever a dependency changes. We call this a **Reactive Effect**.

Vue provides an API that allows you to create reactive effects: `watchEffect()`. In fact, you may have noticed that it works pretty similarly to the magical `whenDepsChange()` in the example. We can now rework the original example using actual Vue APIs:

```js
import { ref, watchEffect } from 'vue'
```

```
  }
}
```

副作用订阅将被存储在一个全局的 `WeakMap<target, Map<key, Set<effect>>>` 数据结构中。如果在第一次追踪时没有找到对相应属性订阅的副作用集合，它将会在这里新建。这就是 `getSubscribersForProperty()` 函数所做的事。为了简化描述，我们跳过了它其中的细节。

在 `trigger()` 之中，我们会再查找到该属性的所有订阅副作用。但这一次我们需要执行它们：

```js
function trigger(target, key) {
  const effects = getSubscribersForProperty(target, key)
  effects.forEach((effect) => effect())
}
```

现在让我们回到 `whenDepsChange()` 函数中：

```js
function whenDepsChange(update) {
  const effect = () => {
    activeEffect = effect
    update()
    activeEffect = null
  }
  effect()
}
```

它将原本的 `update` 函数包装在了一个副作用函数中。在运行实际的更新之前，这个外部函数会将自己设为当前活跃的副作用。这使得在更新期间的 `track()` 调用都能定位到这个当前活跃的副作用。

此时，我们已经创建了一个能自动跟踪其依赖的副作用，它会在任意依赖被改动时重新运行。我们称其为**响应式副作用**。

Vue 提供了一个 API 来让你创建响应式副作用 `watchEffect()`。事实上，你会发现它的使用方式和我们上面示例中说的魔法函数 `whenDepsChange()` 非常相似。我们可以用真正的 Vue API 改写上面的例子：

```js
import { ref, watchEffect } from 'vue'
```

```
const A0 = ref(0)
const A1 = ref(1)
const A2 = ref()
watchEffect(() => {
  // 追踪 A0 和 A1
  A2.value = A0.value + A1.value
})
// 将触发副作用
A0.value = 2
```

Using a reactive effect to mutate a ref isn't the most interesting use case - in fact, using a computed property makes it more declarative:

js

```
import { ref, computed } from 'vue'
const A0 = ref(0)
const A1 = ref(1)
const A2 = computed(() => A0.value + A1.value)
A0.value = 2
```

Internally, `computed` manages its invalidation and re-computation using a reactive effect.

So what's an example of a common and useful reactive effect? Well, updating the DOM! We can implement simple ”reactive rendering” like this:

js

```
import { ref, watchEffect } from 'vue'
const count = ref(0)
watchEffect(() => {
  document.body.innerHTML = `计数: ${count.value}`
})
// 更新 DOM
count.value++
```

In fact, this is pretty close to how a Vue component keeps the state and the DOM in sync - each component instance creates a reactive effect to render and update the DOM. Of course, Vue components use much more efficient ways to update the DOM than `innerHTML`. This is discussed in Rendering Mechanism.

```
const A0 = ref(0)
const A1 = ref(1)
const A2 = ref()
watchEffect(() => {
  // 追踪 A0 和 A1
  A2.value = A0.value + A1.value
})
// 将触发副作用
A0.value = 2
```

使用一个响应式副作用来更改一个 ref 并不是最优解，事实上使用计算属性会更直观简洁：

js

```
import { ref, computed } from 'vue'
const A0 = ref(0)
const A1 = ref(1)
const A2 = computed(() => A0.value + A1.value)
A0.value = 2
```

在内部，`computed` 会使用响应式副作用来管理失效与重新计算的过程。

那么，常见的响应式副作用的用例是什么呢？自然是更新 DOM！我们可以像下面这样实现一个简单的 “响应式渲染”：

js

```
import { ref, watchEffect } from 'vue'
const count = ref(0)
watchEffect(() => {
  document.body.innerHTML = `计数: ${count.value}`
})
// 更新 DOM
count.value++
```

实际上，这与 Vue 组件保持状态和 DOM 同步的方式非常接近——每个组件实例创建一个响应式副作用来渲染和更新 DOM。当然，Vue 组件使用了比 `innerHTML` 更高效的方式来更新 DOM。这会在渲染机制一章中详细介绍。

9.3.3 Runtime vs. Compile-time Reactivity

Vue's reactivity system is primarily runtime-based: the tracking and triggering are all performed while the code is running directly in the browser. The pros of runtime reactivity are that it can work without a build step, and there are fewer edge cases. On the other hand, this makes it constrained by the syntax limitations of JavaScript, leading to the need of value containers like Vue refs.

Some frameworks, such as Svelte, choose to overcome such limitations by implementing reactivity during compilation. It analyzes and transforms the code in order to simulate reactivity. The compilation step allows the framework to alter the semantics of JavaScript itself - for example, implicitly injecting code that performs dependency analysis and effect triggering around access to locally defined variables. The downside is that such transforms require a build step, and altering JavaScript semantics is essentially creating a language that looks like JavaScript but compiles into something else.

The Vue team did explore this direction via an experimental feature called Reactivity Transform, but in the end we have decided that it would not be a good fit for the project due to the reasoning here.

9.3.4 Reactivity Debugging

It's great that Vue's reactivity system automatically tracks dependencies, but in some cases we may want to figure out exactly what is being tracked, or what is causing a component to re-render.

Component Debugging Hooks

We can debug what dependencies are used during a component's render and which dependency is triggering an update using the `onRenderTracked` and `onRenderTriggered` lifecycle hooks. Both hooks will receive a debugger event which contains information on the dependency in question. It is recommended to place a `debugger` statement in the callbacks to interactively inspect the dependency:

html

```
<script setup>
import { onRenderTracked, onRenderTriggered } from 'vue'
onRenderTracked((event) => {
  debugger
})
```

9.3.3 运行时 vs. 编译时响应性

Vue 的响应式系统基本是基于运行时的。追踪和触发都是在浏览器中运行时进行的。运行时响应性的优点是，它可以在没有构建步骤的情况下工作，而且边界情况较少。另一方面，这使得它受到了 JavaScript 语法的制约，导致需要使用一些例如 Vue ref 这样的值的容器。

一些框架，如 Svelte，选择通过编译时实现响应性来克服这种限制。它对代码进行分析和转换，以模拟响应性。该编译步骤允许框架改变 JavaScript 本身的语义——例如，隐式地注入执行依赖性分析的代码，以及围绕对本地定义的变量的访问进行作用触发。这样做的缺点是，该转换需要一个构建步骤，而改变 JavaScript 的语义实质上是在创造一种新语言，看起来像 JavaScript 但编译出来的东西是另外一回事。

Vue 团队确实曾通过一个名为响应性语法糖的实验性功能来探索这个方向，但最后由于这个原因，我们认为它不适合这个项目。

9.3.4 响应性调试

Vue 的响应性系统可以自动跟踪依赖关系，但在某些情况下，我们可能希望确切地知道正在跟踪什么，或者是什么导致了组件重新渲染。

组件调试钩子

我们可以在一个组件渲染时使用 `onRenderTracked` 生命周期钩子来调试查看哪些依赖正在被使用，或是用 `onRenderTriggered` 来确定哪个依赖正在触发更新。这些钩子都会收到一个调试事件，其中包含了触发相关事件的依赖的信息。推荐在回调中放置一个 `debugger` 语句，使你可以在开发者工具中交互式地查看依赖：

html

```
<script setup>
import { onRenderTracked, onRenderTriggered } from 'vue'
onRenderTracked((event) => {
  debugger
})
```

```
onRenderTriggered((event) => {
  debugger
})
</script>
```

TIP

Component debug hooks only work in development mode.

The debug event objects have the following type:

```ts
type DebuggerEvent = {
  effect: ReactiveEffect
  target: object
  type:
    | TrackOpTypes /* 'get' | 'has' | 'iterate' */
    | TriggerOpTypes /* 'set' | 'add' | 'delete' | 'clear' */
  key: any
  newValue?: any
  oldValue?: any
  oldTarget?: Map<any, any> | Set<any>
}
```

Computed Debugging

We can debug computed properties by passing `computed()` a second options object with `onTrack` and `onTrigger` callbacks:

- `onTrack` will be called when a reactive property or ref is tracked as a dependency.
- `onTrigger` will be called when the watcher callback is triggered by the mutation of a dependency.

Both callbacks will receive debugger events in the same format as component debug hooks:

```js
const plusOne = computed(() => count.value + 1, {
  onTrack(e) {
    // 当 count.value 被追踪为依赖时触发
```

```
onRenderTriggered((event) => {
  debugger
})
</script>
```

TIP

组件调试钩子仅会在开发模式下工作

调试事件对象有如下的类型定义：

```ts
type DebuggerEvent = {
  effect: ReactiveEffect
  target: object
  type:
    | TrackOpTypes /* 'get' | 'has' | 'iterate' */
    | TriggerOpTypes /* 'set' | 'add' | 'delete' | 'clear' */
  key: any
  newValue?: any
  oldValue?: any
  oldTarget?: Map<any, any> | Set<any>
}
```

计算属性调试

我们可以向 `computed()` 传入第二个参数，是一个包含了 `onTrack` 和 `onTrigger` 两个回调函数的对象：

- `onTrack` 将在响应属性或引用作为依赖项被跟踪时被调用。
- `onTrigger` 将在侦听器回调被依赖项的变更触发时被调用。

这两个回调都会作为组件调试的钩子，接受相同格式的调试事件：

```js
const plusOne = computed(() => count.value + 1, {
  onTrack(e) {
    // 当 count.value 被追踪为依赖时触发
    debugger
  },
  onTrigger(e) {
    // 当 count.value 被更改时触发
    debugger
  }
})
// 访问 plusOne，会触发 onTrack
console.log(plusOne.value)
// 更改 count.value，应该会触发 onTrigger
count.value++
```

TIP

onTrack and onTrigger computed options only work in development mode.

Watcher Debugging

Similar to computed(), watchers also support the onTrack and onTrigger options:

```js
watch(source, callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})
watchEffect(callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})
```

```js
    debugger
  },
  onTrigger(e) {
    // 当 count.value 被更改时触发
    debugger
  }
})
// 访问 plusOne，会触发 onTrack
console.log(plusOne.value)
// 更改 count.value，应该会触发 onTrigger
count.value++
```

TIP

计算属性的 onTrack 和 onTrigger 选项仅会在开发模式下工作。

侦听器调试

和 computed() 类似，侦听器也支持 onTrack 和 onTrigger 选项：

```js
watch(source, callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})
watchEffect(callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})
```

TIP

onTrack and onTrigger watcher options only work in development mode.

9.3.5 Integration with External State Systems

Vue's reactivity system works by deeply converting plain JavaScript objects into reactive proxies. The deep conversion can be unnecessary or sometimes unwanted when integrating with external state management systems (e.g. if an external solution also uses Proxies).

The general idea of integrating Vue's reactivity system with an external state management solution is to hold the external state in a `shallowRef`. A shallow ref is only reactive when its `.value` property is accessed - the inner value is left intact. When the external state changes, replace the ref value to trigger updates.

Immutable Data

If you are implementing an undo / redo feature, you likely want to take a snapshot of the application's state on every user edit. However, Vue's mutable reactivity system isn't best suited for this if the state tree is large, because serializing the entire state object on every update can be expensive in terms of both CPU and memory costs.

Immutable data structures solve this by never mutating the state objects - instead, it creates new objects that share the same, unchanged parts with old ones. There are different ways of using immutable data in JavaScript, but we recommend using Immer with Vue because it allows you to use immutable data while keeping the more ergonomic, mutable syntax.

We can integrate Immer with Vue via a simple composable:

```js
import produce from 'immer'
import { shallowRef } from 'vue'
export function useImmer(baseState) {
  const state = shallowRef(baseState)
  const update = (updater) => {
    state.value = produce(state.value, updater)
  }
```

TIP

侦听器的 `onTrack` 和 `onTrigger` 选项仅会在开发模式下工作。

9.3.5 与外部状态系统集成

Vue 的响应性系统是通过深度转换普通 JavaScript 对象为响应式代理来实现的。这种深度转换在一些情况下是不必要的，在和一些外部状态管理系统集成时，甚至是需要避免的 (例如，当一个外部的解决方案也用了 Proxy 时)。

将 Vue 的响应性系统与外部状态管理方案集成的大致思路是：将外部状态放在一个 `shallowRef` 中。一个浅层的 ref 中只有它的 `.value` 属性本身被访问时才是有响应性的，而不关心它内部的值。当外部状态改变时，替换此 ref 的 `.value` 才会触发更新。

不可变数据

如果你正在实现一个撤销/重做的功能，你可能想要对用户编辑时应用的状态进行快照记录。然而，如果状态树很大的话，Vue 的可变响应性系统没法很好地处理这种情况，因为在每次更新时都序列化整个状态对象对 CPU 和内存开销来说都是非常昂贵的。

不可变数据结构通过永不更改状态对象来解决这个问题。与 Vue 不同的是，它会创建一个新对象，保留旧的对象未发生改变的一部分。在 JavaScript 中有多种不同的方式来使用不可变数据，但我们推荐使用 Immer 搭配 Vue，因为它使你可以在保持原有直观、可变的语法的同时，使用不可变数据。

我们可以通过一个简单的组合式函数来集成 Immer：

```js
import produce from 'immer'
import { shallowRef } from 'vue'
export function useImmer(baseState) {
  const state = shallowRef(baseState)
  const update = (updater) => {
    state.value = produce(state.value, updater)
  }
  return [state, update]
}
```

Try it in the Playground

State Machines

State Machine is a model for describing all the possible states an application can be in, and all the possible ways it can transition from one state to another. While it may be overkill for simple components, it can help make complex state flows more robust and manageable.

One of the most popular state machine implementations in JavaScript is XState. Here's a composable that integrates with it:

```js
import { createMachine, interpret } from 'xstate'
import { shallowRef } from 'vue'
export function useMachine(options) {
  const machine = createMachine(options)
  const state = shallowRef(machine.initialState)
  const service = interpret(machine)
    .onTransition((newState) => (state.value = newState))
    .start()
  const send = (event) => service.send(event)
  return [state, send]
}
```

Try it in the Playground

RxJS

RxJS is a library for working with asynchronous event streams. The VueUse library provides the `@vueuse/rxjs` add-on for connecting RxJS streams with Vue's reactivity system.

9.3.6 Connection to Signals

Quite a few other frameworks have introduced reactivity primitives similar to refs from Vue's Composition API, under the term "signals":

```
  return [state, update]
}
```

在演练场中尝试一下

状态机

状态机是一种数据模型，用于描述应用可能处于的所有可能状态，以及从一种状态转换到另一种状态的所有可能方式。虽然对于简单的组件来说，这可能有些小题大做了，但它的确可以使得复杂的状态流更加健壮和易于管理。

XState 是 JavaScript 中一个比较常用的状态机实现方案。这里是集成它的一个例子：

```js
import { createMachine, interpret } from 'xstate'
import { shallowRef } from 'vue'
export function useMachine(options) {
  const machine = createMachine(options)
  const state = shallowRef(machine.initialState)
  const service = interpret(machine)
    .onTransition((newState) => (state.value = newState))
    .start()
  const send = (event) => service.send(event)
  return [state, send]
}
```

在演练场中尝试一下

RxJS

RxJS 是一个用于处理异步事件流的库。VueUse 库提供了 `@vueuse/rxjs` 扩展来支持连接 RxJS 流与 Vue 的响应性系统。

9.3.6 与信号 (signal) 的联系

很多其他框架已经引入了与 Vue 组合式 API 中的 ref 类似的响应性基础类型，并称之为 "信号"：

- Solid Signals
- Angular Signals
- Preact Signals
- Qwik Signals

Fundamentally, signals are the same kind of reactivity primitive as Vue refs. It's a value container that provides dependency tracking on access, and side-effect triggering on mutation. This reactivity-primitive-based paradigm isn't a particularly new concept in the frontend world: it dates back to implementations like Knockout observables and Meteor Tracker from more than a decade ago. Vue Options API and the React state management library MobX are also based on the same principles, but hide the primitives behind object properties.

Although not a necessary trait for something to qualify as signals, today the concept is often discussed alongside the rendering model where updates are performed through fine-grained subscriptions. Due to the use of Virtual DOM, Vue currently relies on compilers to achieve similar optimizations. However, we are also exploring a new Solid-inspired compilation strategy (Vapor Mode) that does not rely on Virtual DOM and takes more advantage of Vue's built-in reactivity system.

API Design Trade-Offs

The design of Preact and Qwik's signals are very similar to Vue's shallowRef: all three provide a mutable interface via the `.value` property. We will focus the discussion on Solid and Angular signals.

Solid Signals Solid's `createSignal()` API design emphasizes read / write segregation. Signals are exposed as a read-only getter and a separate setter:

js
```js
const [count, setCount] = createSignal(0)
count() // 访问值
setCount(1) // 更新值
```

Notice how the `count` signal can be passed down without the setter. This ensures that the state can never be mutated unless the setter is also explicitly exposed. Whether this safety guarantee

- Solid 信号
- Angular 信号
- Preact 信号
- Qwik 信号

从根本上说，信号是与 Vue 中的 ref 相同的响应性基础类型。它是一个在访问时跟踪依赖、在变更时触发副作用的值容器。这种基于响应性基础类型的范式在前端领域并不是一个特别新的概念：它可以追溯到十多年前的 Knockout observables 和 Meteor Tracker 等实现。Vue 的选项式 API 和 React 的状态管理库 MobX 也是基于同样的原则，只不过将基础类型这部分隐藏在了对象属性背后。

虽然这并不是信号的必要特征，但如今这个概念经常与细粒度订阅和更新的渲染模型一起讨论。由于使用了虚拟 DOM，Vue 目前依靠编译器来实现类似的优化。然而，我们也在探索一种新的受 Solid 启发的编译策略 (Vapor Mode)，它不依赖于虚拟 DOM，而是更多地利用 Vue 的内置响应性系统。

API 设计权衡

Preact 和 Qwik 的信号设计与 Vue 的 shallowRef 非常相似：三者都通过 `.value` 属性提供了一个更改接口。我们将重点讨论 Solid 和 Angular 的信号。

Solid 的 `createSignal()` API 设计强调了读/写隔离。信号通过一个只读的 getter 和另一个单独的 setter 暴露：

js
```js
const [count, setCount] = createSignal(0)
count() // 访问值
setCount(1) // 更新值
```

注意到 `count` 信号在没有 setter 的情况也能传递。这就保证了除非 setter 也被明确暴露，否则状态永远不会被改变。这种更冗长的语法带来的安全保证的合理

justifies the more verbose syntax could be subject to the requirement of the project and personal taste - but in case you prefer this API style, you can easily replicate it in Vue:

```js
import { shallowRef, triggerRef } from 'vue'
export function createSignal(value, options) {
  const r = shallowRef(value)
  const get = () => r.value
  const set = (v) => {
    r.value = typeof v === 'function' ? v(r.value) : v
    if (options?.equals === false) triggerRef(r)
  }
  return [get, set]
}
```

Try it in the Playground

Angular 信号 Angular is undergoing some fundamental changes by foregoing dirty-checking and introducing its own implementation of a reactivity primitive. The Angular Signal API looks like this:

```js
const count = signal(0)
count() // 访问值
count.set(1) //设置值
count.update((v) => v + 1) // 通过前值更新
// 对具有相同身份的深层对象进行更改
const state = signal({ count: 0 })
state.mutate((o) => {
  o.count++
})
```

Again, we can easily replicate the API in Vue:

```js
import { shallowRef, triggerRef } from 'vue'
export function signal(initialValue) {
  const r = shallowRef(initialValue)
  const s = () => r.value
```

性取决于项目的要求和个人品味——但如果你喜欢这种 API 风格，可以轻易地在 Vue 中复制它：

```js
import { shallowRef, triggerRef } from 'vue'
export function createSignal(value, options) {
  const r = shallowRef(value)
  const get = () => r.value
  const set = (v) => {
    r.value = typeof v === 'function' ? v(r.value) : v
    if (options?.equals === false) triggerRef(r)
  }
  return [get, set]
}
```

在演练场中尝试一下

Angular 正在经历一些底层的变化，它放弃了脏检查，并引入了自己的响应性基础类型实现。Angular 的信号 API 看起来像这样：

```js
const count = signal(0)
count() // 访问值
count.set(1) //设置值
count.update((v) => v + 1) // 通过前值更新
// 对具有相同身份的深层对象进行更改
const state = signal({ count: 0 })
state.mutate((o) => {
  o.count++
})
```

同样，我们可以轻易地在 Vue 中复制这个 API：

```js
import { shallowRef, triggerRef } from 'vue'
export function signal(initialValue) {
  const r = shallowRef(initialValue)
  const s = () => r.value
  s.set = (value) => {
    r.value = value
  }
  s.update = (updater) => {
    r.value = updater(r.value)
  }
  s.mutate = (mutator) => {
    mutator(r.value)
    triggerRef(r)
  }
  return s
}
```

Try it in the Playground

Compared to Vue refs, Solid and Angular's getter-based API style provide some interesting trade-offs when used in Vue components:

- `()` is slightly less verbose than `.value`, but updating the value is more verbose.
- There is no ref-unwrapping: accessing values always require `()`. This makes value access consistent everywhere. This also means you can pass raw signals down as component props.

Whether these API styles suit you is to some extent subjective. Our goal here is to demonstrate the underlying similarity and trade-offs between these different API designs. We also want to show that Vue is flexible: you are not really locked into the existing APIs. Should it be necessary, you can create your own reactivity primitive API to suit more specific needs.

9.4 Rendering Mechanism

How does Vue take a template and turn it into actual DOM nodes? How does Vue update those DOM nodes efficiently? We will attempt to shed some light on these questions here by diving into Vue's internal rendering mechanism.

9.4.1 Virtual DOM

You have probably heard about the term "virtual DOM", which Vue's rendering system is based upon.

```
  s.set = (value) => {
    r.value = value
  }
  s.update = (updater) => {
    r.value = updater(r.value)
  }
  s.mutate = (mutator) => {
    mutator(r.value)
    triggerRef(r)
  }
  return s
}
```

在演练场中尝试一下

与 Vue 的 ref 相比，Solid 和 Angular 基于 getter 的 API 风格在 Vue 组件中使用时提供了一些有趣的权衡：

- `()` 比 `.value` 略微省事，但更新值却更冗长；
- 没有 ref 解包：总是需要通过 `()` 来访问值。这使得值的访问在任何地方都是一致的。这也意味着你可以将原始信号作为组件的参数传递下去。

这些 API 风格是否适合你，在某种程度上是主观的。我们在这里的目标是展示这些不同的 API 设计之间的基本相似性和取舍。我们还想说明 Vue 是灵活的：你并没有真正被限定在现有的 API 中。如有必要，你可以创建你自己的响应性基础 API，以满足更多的具体需求。

9.4 渲染机制

Vue 是如何将一份模板转换为真实的 DOM 节点的，又是如何高效地更新这些节点的呢？我们接下来就将尝试通过深入研究 Vue 的内部渲染机制来解释这些问题。

9.4.1 虚拟 DOM

你可能已经听说过“虚拟 DOM”的概念了，Vue 的渲染系统正是基于这个概念构建的。

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM. The concept was pioneered by React, and has been adopted in many other frameworks with different implementations, including Vue.

Virtual DOM is more of a pattern than a specific technology, so there is no one canonical implementation. We can illustrate the idea using a simple example:

```js
const vnode = {
  type: 'div',
  props: {
    id: 'hello'
  },
  children: [
    /* 更多 vnode */
  ]
}
```

Here, `vnode` is a plain JavaScript object (a ”virtual node”) representing a `<div>` element. It contains all the information that we need to create the actual element. It also contains more children vnodes, which makes it the root of a virtual DOM tree.

A runtime renderer can walk a virtual DOM tree and construct a real DOM tree from it. This process is called **mount**.

If we have two copies of virtual DOM trees, the renderer can also walk and compare the two trees, figuring out the differences, and apply those changes to the actual DOM. This process is called **patch**, also known as ”diffing” or ”reconciliation”.

The main benefit of virtual DOM is that it gives the developer the ability to programmatically create, inspect and compose desired UI structures in a declarative way, while leaving the direct DOM manipulation to the renderer.

9.4.2 Render Pipeline

At the high level, this is what happens when a Vue component is mounted:

1. **Compile**: Vue templates are compiled into **render functions**: functions that return virtual

虚拟 DOM (Virtual DOM，简称 VDOM) 是一种编程概念，意为将目标所需的 UI 通过数据结构 “虚拟” 地表示出来，保存在内存中，然后将真实的 DOM 与之保持同步。这个概念是由 React 率先开拓，随后被许多不同的框架采用，当然也包括 Vue。

与其说虚拟 DOM 是一种具体的技术，不如说是一种模式，所以并没有一个标准的实现。我们可以用一个简单的例子来说明：

```js
const vnode = {
  type: 'div',
  props: {
    id: 'hello'
  },
  children: [
    /* 更多 vnode */
  ]
}
```

这里所说的 `vnode` 即一个纯 JavaScript 的对象 (一个 “虚拟节点”)，它代表着一个 `<div>` 元素。它包含我们创建实际元素所需的所有信息。它还包含更多的子节点，这使它成为虚拟 DOM 树的根节点。

一个运行时渲染器将会遍历整个虚拟 DOM 树，并据此构建真实的 DOM 树。这个过程被称为**挂载** (mount)。

如果我们有两份虚拟 DOM 树，渲染器将会有比较地遍历它们，找出它们之间的区别，并应用这其中的变化到真实的 DOM 上。这个过程被称为**更新** (patch)，又被称为 “比对”(diffing) 或 “协调”(reconciliation)。

虚拟 DOM 带来的主要收益是它让开发者能够灵活、声明式地创建、检查和组合所需 UI 的结构，同时只需把具体的 DOM 操作留给渲染器去处理。

9.4.2 渲染管线

从高层面的视角看，Vue 组件挂载时会发生如下几件事：

1. **编译**：Vue 模板被编译为**渲染函数**：即用来返回虚拟 DOM 树的函数。这一

DOM trees. This step can be done either ahead-of-time via a build step, or on-the-fly by using the runtime compiler.

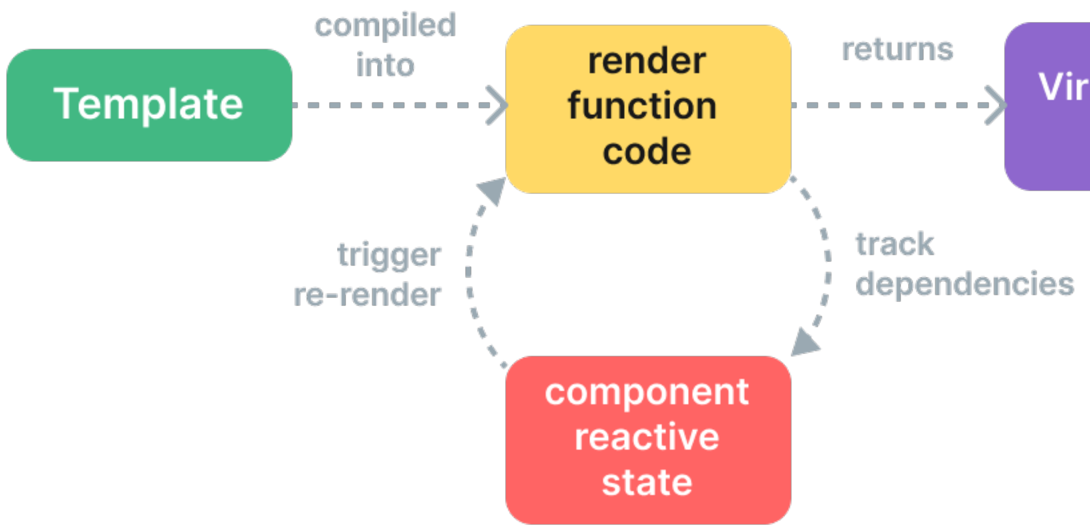
2. **Mount**: The runtime renderer invokes the render functions, walks the returned virtual DOM tree, and creates actual DOM nodes based on it. This step is performed as a reactive effect, so it keeps track of all reactive dependencies that were used.

3. **Patch**: When a dependency used during mount changes, the effect re-runs. This time, a new, updated Virtual DOM tree is created. The runtime renderer walks the new tree, compares it with the old one, and applies necessary updates to the actual DOM.

步骤可以通过构建步骤提前完成，也可以通过使用运行时编译器即时完成。

2. **挂载**：运行时渲染器调用渲染函数，遍历返回的虚拟 DOM 树，并基于它创建实际的 DOM 节点。这一步会作为响应式副作用执行，因此它会追踪其中所用到的所有响应式依赖。

3. **更新**：当一个依赖发生变化后，副作用会重新运行，这时候会创建一个更新后的虚拟 DOM 树。运行时渲染器遍历这棵新树，将它与旧树进行比较，然后将必要的更新应用到真实 DOM 上去。



9.4.3　Templates vs. Render Functions

Vue templates are compiled into virtual DOM render functions. Vue also provides APIs that allow us to skip the template compilation step and directly author render functions. Render functions are more flexible than templates when dealing with highly dynamic logic, because you can work with vnodes using the full power of JavaScript.

So why does Vue recommend templates by default? There are a number of reasons:

1. Templates are closer to actual HTML. This makes it easier to reuse existing HTML snippets,

9.4.3　模板 vs. 渲染函数

Vue 模板会被预编译成虚拟 DOM 渲染函数。Vue 也提供了 API 使我们可以不使用模板编译，直接手写渲染函数。在处理高度动态的逻辑时，渲染函数相比于模板更加灵活，因为你可以完全地使用 JavaScript 来构造你想要的 vnode。

那么为什么 Vue 默认推荐使用模板呢？有以下几点原因：

1. 模板更贴近实际的 HTML。这使得我们能够更方便地重用一些已有的 HTML

apply accessibility best practices, style with CSS, and for designers to understand and modify.

2. Templates are easier to statically analyze due to their more deterministic syntax. This allows Vue's template compiler to apply many compile-time optimizations to improve the performance of the virtual DOM (which we will discuss below).

In practice, templates are sufficient for most use cases in applications. Render functions are typically only used in reusable components that need to deal with highly dynamic rendering logic. Render function usage is discussed in more detail in Render Functions & JSX.

9.4.4 Compiler-Informed Virtual DOM

The virtual DOM implementation in React and most other virtual-DOM implementations are purely runtime: the reconciliation algorithm cannot make any assumptions about the incoming virtual DOM tree, so it has to fully traverse the tree and diff the props of every vnode in order to ensure correctness. In addition, even if a part of the tree never changes, new vnodes are always created for them on each re-render, resulting in unnecessary memory pressure. This is one of the most criticized aspect of virtual DOM: the somewhat brute-force reconciliation process sacrifices efficiency in return for declarativeness and correctness.

But it doesn't have to be that way. In Vue, the framework controls both the compiler and the runtime. This allows us to implement many compile-time optimizations that only a tightly-coupled renderer can take advantage of. The compiler can statically analyze the template and leave hints in the generated code so that the runtime can take shortcuts whenever possible. At the same time, we still preserve the capability for the user to drop down to the render function layer for more direct control in edge cases. We call this hybrid approach **Compiler-Informed Virtual DOM**.

Below, we will discuss a few major optimizations done by the Vue template compiler to improve the virtual DOM's runtime performance.

Static Hoisting

Quite often there will be parts in a template that do not contain any dynamic bindings:

```html
<div>
  <div>foo</div> <!-- 需提升 -->
  <div>bar</div> <!-- 需提升 -->
```

代码片段，能够带来更好的可访问性体验、能更方便地使用 CSS 应用样式，并且更容易使设计师理解和修改。

2. 由于其确定的语法，更容易对模板做静态分析。这使得 Vue 的模板编译器能够应用许多编译时优化来提升虚拟 DOM 的性能表现 (下面我们将展开讨论)。

在实践中，模板对大多数的应用场景都是够用且高效的。渲染函数一般只会在需要处理高度动态渲染逻辑的可重用组件中使用。想了解渲染函数的更多使用细节可以去到渲染函数 & JSX 章节继续阅读。

9.4.4 带编译时信息的虚拟 DOM

虚拟 DOM 在 React 和大多数其他实现中都是纯运行时的：更新算法无法预知新的虚拟 DOM 树会是怎样，因此它总是需要遍历整棵树、比较每个 vnode 上 props 的区别来确保正确性。另外，即使一棵树的某个部分从未改变，还是会在每次重渲染时创建新的 vnode，带来了大量不必要的内存压力。这也是虚拟 DOM 最受诟病的地方之一：这种有点暴力的更新过程通过牺牲效率来换取声明式的写法和最终的正确性。

但实际上我们并不需要这样。在 Vue 中，框架同时控制着编译器和运行时。这使得我们可以为紧密耦合的模板渲染器应用许多编译时优化。编译器可以静态分析模板并在生成的代码中留下标记，使得运行时尽可能地走捷径。与此同时，我们仍旧保留了边界情况时用户想要使用底层渲染函数的能力。我们称这种混合解决方案为**带编译时信息的虚拟 DOM**。

下面，我们将讨论一些 Vue 编译器用来提高虚拟 DOM 运行时性能的主要优化：

静态提升

在模板中常常有部分内容是不带任何动态绑定的：

```html
<div>
  <div>foo</div> <!-- 需提升 -->
  <div>bar</div> <!-- 需提升 -->
  <div>{{ dynamic }}</div>
</div>
```

Inspect in Template Explorer

The foo and bar divs are static - re-creating vnodes and diffing them on each re-render is unnecessary. The Vue compiler automatically hoists their vnode creation calls out of the render function, and reuses the same vnodes on every render. The renderer is also able to completely skip diffing them when it notices the old vnode and the new vnode are the same one.

In addition, when there are enough consecutive static elements, they will be condensed into a single "static vnode" that contains the plain HTML string for all these nodes (Example). These static vnodes are mounted by directly setting innerHTML. They also cache their corresponding DOM nodes on initial mount - if the same piece of content is reused elsewhere in the app, new DOM nodes are created using native cloneNode(), which is extremely efficient.

Patch Flags

For a single element with dynamic bindings, we can also infer a lot of information from it at compile time:

```html
<!-- 仅含 class 绑定 -->
<div :class="{ active }"></div>
<!-- 仅含 id 和 value 绑定 -->
<input :id="id" :value="value">
<!-- 仅含文本子节点 -->
<div>{{ dynamic }}</div>
```

Inspect in Template Explorer

When generating the render function code for these elements, Vue encodes the type of update each of them needs directly in the vnode creation call:

```js
createElementVNode("div", {
  class: _normalizeClass({ active: _ctx.active })
}, null, 2 /* CLASS */)
```

The last argument, 2, is a patch flag. An element can have multiple patch flags, which will be

```
  <div>{{ dynamic }}</div>
</div>
```

在模板编译预览中查看

foo 和 bar 这两个 div 是完全静态的，没有必要在重新渲染时再次创建和比对它们。Vue 编译器自动地会提升这部分 vnode 创建函数到这个模板的渲染函数之外，并在每次渲染时都使用这份相同的 vnode，渲染器知道新旧 vnode 在这部分是完全相同的，所以会完全跳过对它们的差异比对。

此外，当有足够多连续的静态元素时，它们还会再被压缩为一个 “静态 vnode”，其中包含的是这些节点相应的纯 HTML 字符串。(示例)。这些静态节点会直接通过 innerHTML 来挂载。同时还会在初次挂载后缓存相应的 DOM 节点。如果这部分内容在应用中其他地方被重用，那么将会使用原生的 cloneNode() 方法来克隆新的 DOM 节点，这会非常高效。

更新类型标记

对于单个有动态绑定的元素来说，我们可以在编译时推断出大量信息：

```html
<!-- 仅含 class 绑定 -->
<div :class="{ active }"></div>
<!-- 仅含 id 和 value 绑定 -->
<input :id="id" :value="value">
<!-- 仅含文本子节点 -->
<div>{{ dynamic }}</div>
```

在模板编译预览中查看

在为这些元素生成渲染函数时，Vue 在 vnode 创建调用中直接编码了每个元素所需的更新类型：

```js
createElementVNode("div", {
  class: _normalizeClass({ active: _ctx.active })
}, null, 2 /* CLASS */)
```

最后这个参数 2 就是一个更新类型标记 (patch flag)。一个元素可以有多个更新类

merged into a single number. The runtime renderer can then check against the flags using bitwise operations to determine whether it needs to do certain work:

```js
if (vnode.patchFlag & PatchFlags.CLASS /* 2 */) {
  // 更新节点的 CSS class
}
```

Bitwise checks are extremely fast. With the patch flags, Vue is able to do the least amount of work necessary when updating elements with dynamic bindings.

Vue also encodes the type of children a vnode has. For example, a template that has multiple root nodes is represented as a fragment. In most cases, we know for sure that the order of these root nodes will never change, so this information can also be provided to the runtime as a patch flag:

```js
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

The runtime can thus completely skip child-order reconciliation for the root fragment.

Tree Flattening

Taking another look at the generated code from the previous example, you'll notice the root of the returned virtual DOM tree is created using a special `createElementBlock()` call:

```js
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

Conceptually, a ”block” is a part of the template that has stable inner structure. In this case, the entire template has a single block because it does not contain any structural directives like `v-if` and `v-for`.

Each block tracks any descendant nodes (not just direct children) that have patch flags. For example:

型标记，会被合并成一个数字。运行时渲染器也将会使用位运算来检查这些标记，确定相应的更新操作：

```js
if (vnode.patchFlag & PatchFlags.CLASS /* 2 */) {
  // 更新节点的 CSS class
}
```

位运算检查是非常快的。通过这样的更新类型标记，Vue 能够在更新带有动态绑定的元素时做最少的操作。

Vue 也为 vnode 的子节点标记了类型。举例来说，包含多个根节点的模板被表示为一个片段 (fragment)，大多数情况下，我们可以确定其顺序是永远不变的，所以这部分信息就可以提供给运行时作为一个更新类型标记。

```js
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

运行时会完全跳过对这个根片段中子元素顺序的重新协调过程。

树结构打平

再来看看上面这个例子中生成的代码，你会发现所返回的虚拟 DOM 树是经一个特殊的 `createElementBlock()` 调用创建的：

```js
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

这里我们引入一个概念“区块”，内部结构是稳定的一个部分可被称之为一个区块。在这个用例中，整个模板只有一个区块，因为这里没有用到任何结构性指令 (比如 `v-if` 或者 `v-for`)。

每一个块都会追踪其所有带更新类型标记的后代节点 (不只是直接子节点)，举例

```html
<div> <!-- root block -->
  <div>...</div>         <!-- 不会追踪 -->
  <div :id="id"></div>   <!-- 要追踪 -->
  <div>                  <!-- 不会追踪 -->
    <div>{{ bar }}</div> <!-- 要追踪 -->
  </div>
</div>
```

The result is a flattened array that contains only the dynamic descendant nodes:

```html
div (block root)
- div 带有 :id 绑定
- div 带有 {{ bar }} 绑定
```

When this component needs to re-render, it only needs to traverse the flattened tree instead of the full tree. This is called **Tree Flattening**, and it greatly reduces the number of nodes that need to be traversed during virtual DOM reconciliation. Any static parts of the template are effectively skipped.

`v-if` and `v-for` directives will create new block nodes:

```html
<div> <!-- 根区块 -->
  <div>
    <div v-if> <!-- if 区块 -->
      ...
    <div>
  </div>
</div>
```

A child block is tracked inside the parent block's array of dynamic descendants. This retains a stable structure for the parent block.

Impact on SSR Hydration

Both patch flags and tree flattening also greatly improve Vue's SSR Hydration performance:

来说：

```html
<div> <!-- root block -->
  <div>...</div>         <!-- 不会追踪 -->
  <div :id="id"></div>   <!-- 要追踪 -->
  <div>                  <!-- 不会追踪 -->
    <div>{{ bar }}</div> <!-- 要追踪 -->
  </div>
</div>
```

编译的结果会被打平为一个数组，仅包含所有动态的后代节点：

```html
div (block root)
- div 带有 :id 绑定
- div 带有 {{ bar }} 绑定
```

当这个组件需要重渲染时，只需要遍历这个打平的树而非整棵树。这也就是我们所说的**树结构打平**，这大大减少了我们在虚拟 DOM 协调时需要遍历的节点数量。模板中任何的静态部分都会被高效地略过。

`v-if` 和 `v-for` 指令会创建新的区块节点：

```html
<div> <!-- 根区块 -->
  <div>
    <div v-if> <!-- if 区块 -->
      ...
    <div>
  </div>
</div>
```

一个子区块会在父区块的动态子节点数组中被追踪，这为他们的父区块保留了一个稳定的结构。

对 SSR 激活的影响

更新类型标记和树结构打平都大大提升了 Vue SSR 激活的性能表现：

- Single element hydration can take fast paths based on the corresponding vnode's patch flag.
- Only block nodes and their dynamic descendants need to be traversed during hydration, effectively achieving partial hydration at the template level.

9.5 Render Functions & JSX

Vue recommends using templates to build applications in the vast majority of cases. However, there are situations where we need the full programmatic power of JavaScript. That's where we can use the **render function**.

> If you are new to the concept of virtual DOM and render functions, make sure to read the Rendering Mechanism chapter first.

9.5.1 Basic Usage

Creating Vnodes

Vue provides an `h()` function for creating vnodes:

```js
import { h } from 'vue'
const vnode = h(
  'div', // type
  { id: 'foo', class: 'bar' }, // props
  [
    /* children */
  ]
)
```

`h()` is short for **hyperscript** - which means "JavaScript that produces HTML (hypertext markup language)". This name is inherited from conventions shared by many virtual DOM implementations. A more descriptive name could be `createVnode()`, but a shorter name helps when you have to call this function many times in a render function.

The `h()` function is designed to be very flexible:

```js
// 除了类型必填以外，其他的参数都是可选的
```

- 单个元素的激活可以基于相应 vnode 的更新类型标记走更快的捷径。
- 在激活时只有区块节点和其动态子节点需要被遍历，这在模板层面上实现更高效的部分激活。

9.5 渲染函数 & JSX

在绝大多数情况下，Vue 推荐使用模板语法来创建应用。然而在某些使用场景下，我们真的需要用到 JavaScript 完全的编程能力。这时**渲染函数**就派上用场了。

> 如果你还不熟悉虚拟 DOM 和渲染函数的概念的话，请确保先阅读渲染机制章节。

9.5.1 基本用法

创建 Vnodes

Vue 提供了一个 `h()` 函数用于创建 vnodes：

```js
import { h } from 'vue'
const vnode = h(
  'div', // type
  { id: 'foo', class: 'bar' }, // props
  [
    /* children */
  ]
)
```

`h()` 是 **hyperscript** 的简称——意思是“能生成 HTML（超文本标记语言）的 JavaScript”。这个名字来源于许多虚拟 DOM 实现默认形成的约定。一个更准确的名称应该是 `createVnode()`，但当你需要多次使用渲染函数时，一个简短的名字会更省力。

`h()` 函数的使用方式非常的灵活：

```js
// 除了类型必填以外，其他的参数都是可选的
```

```
h('div')
h('div', { id: 'foo' })
// attribute 和 property 都能在 prop 中书写
// Vue 会自动将它们分配到正确的位置
h('div', { class: 'bar', innerHTML: 'hello' })
// 像 `.prop` 和 `.attr` 这样的的属性修饰符
// 可以分别通过 `.` 和 `^` 前缀来添加
h('div', { '.name': 'some-name', '^width': '100' })
// 类与样式可以像在模板中一样
// 用数组或对象的形式书写
h('div', { class: [foo, { bar }], style: { color: 'red' } })
// 事件监听器应以 onXxx 的形式书写
h('div', { onClick: () => {} })
// children 可以是一个字符串
h('div', { id: 'foo' }, 'hello')
// 没有 props 时可以省略不写
h('div', 'hello')
h('div', [h('span', 'hello')])
// children 数组可以同时包含 vnodes 与字符串
h('div', ['hello', h('span', 'hello')])
```

The resulting vnode has the following shape:

```js
const vnode = h('div', { id: 'foo' }, [])
vnode.type // 'div'
vnode.props // { id: 'foo' }
vnode.children // []
vnode.key // null
```

Note

The full VNode interface contains many other internal properties, but it is strongly recommended to avoid relying on any properties other than the ones listed here. This avoids unintended breakage in case the internal properties are changed.

```
h('div')
h('div', { id: 'foo' })
// attribute 和 property 都能在 prop 中书写
// Vue 会自动将它们分配到正确的位置
h('div', { class: 'bar', innerHTML: 'hello' })
// 像 `.prop` 和 `.attr` 这样的的属性修饰符
// 可以分别通过 `.` 和 `^` 前缀来添加
h('div', { '.name': 'some-name', '^width': '100' })
// 类与样式可以像在模板中一样
// 用数组或对象的形式书写
h('div', { class: [foo, { bar }], style: { color: 'red' } })
// 事件监听器应以 onXxx 的形式书写
h('div', { onClick: () => {} })
// children 可以是一个字符串
h('div', { id: 'foo' }, 'hello')
// 没有 props 时可以省略不写
h('div', 'hello')
h('div', [h('span', 'hello')])
// children 数组可以同时包含 vnodes 与字符串
h('div', ['hello', h('span', 'hello')])
```

得到的 vnode 为如下形式：

```js
const vnode = h('div', { id: 'foo' }, [])
vnode.type // 'div'
vnode.props // { id: 'foo' }
vnode.children // []
vnode.key // null
```

注意事项

完整的 VNode 接口包含其他内部属性，但是强烈建议避免使用这些没有在这里列举出的属性。这样能够避免因内部属性变更而导致的不兼容性问题。

声明渲染函数

Declaring Render Functions

When using templates with Composition API, the return value of the `setup()` hook is used to expose data to the template. When using render functions, however, we can directly return the render function instead:

```js
import { ref, h } from 'vue'
export default {
  props: {
    /* ... */
  },
  setup(props) {
    const count = ref(1)
    // 返回渲染函数
    return () => h('div', props.msg + count.value)
  }
}
```

The render function is declared inside `setup()` so it naturally has access to the props and any reactive state declared in the same scope.

In addition to returning a single vnode, you can also return strings or arrays:

```js
export default {
  setup() {
    return () => 'hello world!'
  }
}
```

```js
import { h } from 'vue'
export default {
  setup() {
    // 使用数组返回多个根节点
    return () => [
      h('div'),
      h('div'),
      h('div')
    ]
```

当组合式 API 与模板一起使用时，`setup()` 钩子的返回值是用于暴露数据给模板。然而当我们使用渲染函数时，可以直接把渲染函数返回：

```js
import { ref, h } from 'vue'
export default {
  props: {
    /* ... */
  },
  setup(props) {
    const count = ref(1)
    // 返回渲染函数
    return () => h('div', props.msg + count.value)
  }
}
```

在 `setup()` 内部声明的渲染函数天生能够访问在同一范围内声明的 props 和许多响应式状态。

除了返回一个 vnode，你还可以返回字符串或数组：

```js
export default {
  setup() {
    return () => 'hello world!'
  }
}
```

```js
import { h } from 'vue'
export default {
  setup() {
    // 使用数组返回多个根节点
    return () => [
      h('div'),
      h('div'),
      h('div')
    ]
  }
}
```

TIP

Make sure to return a function instead of directly returning values! The `setup()` function is called only once per component, while the returned render function will be called multiple times.

If a render function component doesn't need any instance state, they can also be declared directly as a function for brevity:

```js
function Hello() {
  return 'hello world!'
}
```

That's right, this is a valid Vue component! See Functional Components for more details on this syntax.

Vnodes Must Be Unique

All vnodes in the component tree must be unique. That means the following render function is invalid:

```js
function render() {
  const p = h('p', 'hi')
  return h('div', [
    // 啊哦，重复的 vnodes 是无效的
    p,
    p
  ])
}
```

If you really want to duplicate the same element/component many times, you can do so with a factory function. For example, the following render function is a perfectly valid way of rendering 20 identical paragraphs:

```js
function render() {
```

```js
  }
}
```

TIP

请确保返回的是一个函数而不是一个值！`setup()` 函数在每个组件中只会被调用一次，而返回的渲染函数将会被调用多次。

如果一个渲染函数组件不需要任何实例状态，为了简洁起见，它们也可以直接被声明为一个函数：

```js
function Hello() {
  return 'hello world!'
}
```

没错，这就是一个合法的 Vue 组件！参阅函数式组件来了解更多语法细节。

Vnodes 必须唯一

组件树中的 vnodes 必须是唯一的。下面是错误示范：

```js
function render() {
  const p = h('p', 'hi')
  return h('div', [
    // 啊哦，重复的 vnodes 是无效的
    p,
    p
  ])
}
```

如果你真的非常想在页面上渲染多个重复的元素或者组件，你可以使用一个工厂函数来做这件事。比如下面的这个渲染函数就可以完美渲染出 20 个相同的段落：

```js
function render() {
  return h(
    'div',
    Array.from({ length: 20 }).map(() => {
      return h('p', 'hi')
    })
  )
}
```

9.5.2 JSX / TSX

JSX is an XML-like extension to JavaScript that allows us to write code like this:

```html
const vnode = <div>hello</div>
```

Inside JSX expressions, use curly braces to embed dynamic values:

```html
const vnode = <div id={dynamicId}>hello, {userName}</div>
```

`create-vue` and Vue CLI both have options for scaffolding projects with pre-configured JSX support. If you are configuring JSX manually, please refer to the documentation of `@vue/babel-plugin-jsx` for details.

Although first introduced by React, JSX actually has no defined runtime semantics and can be compiled into various different outputs. If you have worked with JSX before, do note that **Vue JSX transform is different from React's JSX transform**, so you can't use React's JSX transform in Vue applications. Some notable differences from React JSX include:

- You can use HTML attributes such as `class` and `for` as props - no need to use `className` or `htmlFor`.
- Passing children to components (i.e. slots) works differently.

Vue's type definition also provides type inference for TSX usage. When using TSX, make sure to specify `"jsx": "preserve"` in `tsconfig.json` so that TypeScript leaves the JSX syntax intact for Vue JSX transform to process.

```
  return h(
    'div',
    Array.from({ length: 20 }).map(() => {
      return h('p', 'hi')
    })
  )
}
```

9.5.2 JSX / TSX

JSX 是 JavaScript 的一个类似 XML 的扩展，有了它，我们可以用以下的方式来书写代码：

```html
const vnode = <div>hello</div>
```

在 JSX 表达式中，使用大括号来嵌入动态值：

```html
const vnode = <div id={dynamicId}>hello, {userName}</div>
```

`create-vue` 和 Vue CLI 都有预置的 JSX 语法支持。如果你想手动配置 JSX，请参阅 `@vue/babel-plugin-jsx` 文档获取更多细节。

虽然最早是由 React 引入，但实际上 JSX 语法并没有定义运行时语义，并且能被编译成各种不同的输出形式。如果你之前使用过 JSX 语法，那么请注意 **Vue 的 JSX 转换方式与 React 中 JSX 的转换方式不同**，因此你不能在 Vue 应用中使用 React 的 JSX 转换。与 React JSX 语法的一些明显区别包括：

- 可以使用 HTML attributes 比如 `class` 和 `for` 作为 props - 不需要使用 `className` 或 `htmlFor`。
- 传递子元素给组件 (比如 slots) 的方式不同。

Vue 的类型定义也提供了 TSX 语法的类型推导支持。当使用 TSX 语法时，确保在 `tsconfig.json` 中配置了 `"jsx": "preserve"`，这样的 TypeScript 就能保证 Vue JSX 语法转换过程中的完整性。

JSX Type Inference

Similar to the transform, Vue's JSX also needs different type definitions. Currently, Vue's types automatically registers Vue's JSX types globally. This means TSX will work out of the box when Vue's type is available.

The global JSX types may cause conflict with used together with other libraries that also needs JSX type inference, in particular React. Starting in 3.3, Vue supports specifying JSX namespace via TypeScript's jsxImportSource option. We plan to remove the default global JSX namespace registration in 3.4.

For TSX users, it is suggested to set jsxImportSource to `'vue'` in `tsconfig.json` after upgrading to 3.3, or opt-in per file with `/* @jsxImportSource vue */`. This will allow you to opt-in to the new behavior now and upgrade seamlessly when 3.4 releases.

If there is code that depends on the presence of the global `JSX` namespace, you can retain the exact pre-3.4 global behavior by explicitly referencing `vue/jsx`, which registers the global `JSX` namespace.

9.5.3 Render Function Recipes

Below we will provide some common recipes for implementing template features as their equivalent render functions / JSX.

v-if

Template:

```html
<div>
  <div v-if="ok">yes</div>
  <span v-else>no</span>
</div>
```

Equivalent render function / JSX:

```js
h('div', [ok.value ? h('div', 'yes') : h('span', 'no')])
```

```html
<div>{ok.value ? <div>yes</div> : <span>no</span>}</div>
```

JSX 类型推断

与转换类似，Vue 的 JSX 也需要不同的类型定义。目前，Vue 的类型会在全局范围内自动注册 Vue 的 JSX 类型。这意味着当 Vue 的类型可用时，TSX 将可以开箱即用。

全局的 JSX 类型在与其他同样需要 JSX 类型推断的库一起使用时可能会引起冲突，特别是 React。从 3.3 开始，Vue 支持通过 TypeScript 的 jsxImportSource 选项指定 JSX 命名空间。我们计划在 3.4 中移除默认的全局 JSX 命名空间注册。

对于 TSX 用户，建议在升级到 3.3 之后，在 `tsconfig.json` 中把 jsxImportSource 设置为 `'vue'`，或者针对单个文件加入 `/* @jsxImportSource vue */`。这可以让你现在就选用该新特性，并在 3.4 发布时无痛升级。

如果仍有代码依赖于全局存在的 `JSX` 命名空间，你可以通过显式引用 `vue/jsx` 来保留 3.4 之前的全局行为，它注册了全局 `JSX` 命名空间。

9.5.3 渲染函数案例

下面我们提供了几个常见的用等价的渲染函数 / JSX 语法，实现模板功能的案例：

v-if

模板：

```html
<div>
  <div v-if="ok">yes</div>
  <span v-else>no</span>
</div>
```

等价于使用如下渲染函数 / JSX 语法：

```js
h('div', [ok.value ? h('div', 'yes') : h('span', 'no')])
```

```html
<div>{ok.value ? <div>yes</div> : <span>no</span>}</div>
```

v-for

Template:

```html
<ul>
  <li v-for="{ id, text } in items" :key="id">
    {{ text }}
  </li>
</ul>
```

Equivalent render function / JSX:

```js
h(
  'ul',
  // assuming `items` is a ref with array value
  items.value.map(({ id, text }) => {
    return h('li', { key: id }, text)
  })
)
```

```html
<ul>
  {items.value.map(({ id, text }) => {
    return <li key={id}>{text}</li>
  })}
</ul>
```

v-on

Props with names that start with `on` followed by an uppercase letter are treated as event listeners. For example, `onClick` is the equivalent of `@click` in templates.

```js
h(
  'button',
  {
    onClick(event) {
      /* ... */
    }
```

v-for

模板：

```html
<ul>
  <li v-for="{ id, text } in items" :key="id">
    {{ text }}
  </li>
</ul>
```

等价于使用如下渲染函数 / JSX 语法：

```js
h(
  'ul',
  // assuming `items` is a ref with array value
  items.value.map(({ id, text }) => {
    return h('li', { key: id }, text)
  })
)
```

```html
<ul>
  {items.value.map(({ id, text }) => {
    return <li key={id}>{text}</li>
  })}
</ul>
```

v-on

以 `on` 开头，并跟着大写字母的 props 会被当作事件监听器。比如，`onClick` 与模板中的 `@click` 等价。

```js
h(
  'button',
  {
    onClick(event) {
      /* ... */
    }
  },
  'click me'
)
```

html

```
<button
  onClick={(event) => {
    /* ... */
  }}
>
  click me
</button>
```

Event Modifiers For the `.passive`, `.capture`, and `.once` event modifiers, they can be concatenated after the event name using camelCase.

For example:

js

```
h('input', {
  onClickCapture() {
    /* 捕捉模式中的监听器 */
  },
  onKeyupOnce() {
    /* 只触发一次 */
  },
  onMouseoverOnceCapture() {
    /* 单次 + 捕捉 */
  }
})
```

html

```
<input
  onClickCapture={() => {}}
  onKeyupOnce={() => {}}
  onMouseoverOnceCapture={() => {}}
```

```
  },
  'click me'
)
```

html

```
<button
  onClick={(event) => {
    /* ... */
  }}
>
  click me
</button>
```

事件修饰符

对于 `.passive`、`.capture` 和 `.once` 事件修饰符，可以使用驼峰写法将他们拼接在事件名后面：

实例：

js

```
h('input', {
  onClickCapture() {
    /* 捕捉模式中的监听器 */
  },
  onKeyupOnce() {
    /* 只触发一次 */
  },
  onMouseoverOnceCapture() {
    /* 单次 + 捕捉 */
  }
})
```

html

```
<input
  onClickCapture={() => {}}
  onKeyupOnce={() => {}}
  onMouseoverOnceCapture={() => {}}
```

```
/>
```

For other event and key modifiers, the `withModifiers` helper can be used:

```js
import { withModifiers } from 'vue'
h('div', {
  onClick: withModifiers(() => {}, ['self'])
})
```

```html
<div onClick={withModifiers(() => {}, ['self'])} />
```

Components

To create a vnode for a component, the first argument passed to `h()` should be the component definition. This means when using render functions, it is unnecessary to register components - you can just use the imported components directly:

```js
import Foo from './Foo.vue'
import Bar from './Bar.jsx'
function render() {
  return h('div', [h(Foo), h(Bar)])
}
```

```html
function render() {
  return (
    <div>
      <Foo />
      <Bar />
    </div>
  )
}
```

As we can see, `h` can work with components imported from any file format as long as it's a valid Vue component.

Dynamic components are straightforward with render functions:

```
/>
```

对于事件和按键修饰符，可以使用 `withModifiers` 函数：

```js
import { withModifiers } from 'vue'
h('div', {
  onClick: withModifiers(() => {}, ['self'])
})
```

```html
<div onClick={withModifiers(() => {}, ['self'])} />
```

组件

在给组件创建 vnode 时，传递给 `h()` 函数的第一个参数应当是组件的定义。这意味着使用渲染函数时不再需要注册组件了 —— 可以直接使用导入的组件：

```js
import Foo from './Foo.vue'
import Bar from './Bar.jsx'
function render() {
  return h('div', [h(Foo), h(Bar)])
}
```

```html
function render() {
  return (
    <div>
      <Foo />
      <Bar />
    </div>
  )
}
```

不管是什么类型的文件，只要从中导入的是有效的 Vue 组件，`h` 就能正常运作。

动态组件在渲染函数中也可直接使用：

js
```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'
function render() {
    return ok.value ? h(Foo) : h(Bar)
}
```

html
```
function render() {
  return ok.value ? <Foo /> : <Bar />
}
```

If a component is registered by name and cannot be imported directly (for example, globally registered by a library), it can be programmatically resolved by using the `resolveComponent()` helper.

Rendering Slots

In render functions, slots can be accessed from the `setup()` context. Each slot on the `slots` object is a **function that returns an array of vnodes**:

js
```
export default {
  props: ['message'],
  setup(props, { slots }) {
    return () => [
      // 默认插槽:
      // <div><slot /></div>
      h('div', slots.default()),
      // 具名插槽:
      // <div><slot name="footer" :text="message" /></div>
      h(
        'div',
        slots.footer({
          text: props.message
        })
      )
    ]
  }
```

js
```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'
function render() {
    return ok.value ? h(Foo) : h(Bar)
}
```

html
```
function render() {
  return ok.value ? <Foo /> : <Bar />
}
```

如果一个组件是用名字注册的，不能直接导入 (例如，由一个库全局注册)，可以使用 `resolveComponent()` 来解决这个问题。

渲染插槽

在渲染函数中，插槽可以通过 `setup()` 的上下文来访问。每个 `slots` 对象中的插槽都是一个**返回 vnodes 数组的函数**：

js
```
export default {
  props: ['message'],
  setup(props, { slots }) {
    return () => [
      // 默认插槽:
      // <div><slot /></div>
      h('div', slots.default()),
      // 具名插槽:
      // <div><slot name="footer" :text="message" /></div>
      h(
        'div',
        slots.footer({
          text: props.message
        })
      )
    ]
  }
}
```

JSX equivalent:

```html
// 默认插槽
<div>{slots.default()}</div>
// 具名插槽
<div>{slots.footer({ text: props.message })}</div>
```

Passing Slots

Passing children to components works a bit differently from passing children to elements. Instead of an array, we need to pass either a slot function, or an object of slot functions. Slot functions can return anything a normal render function can return - which will always be normalized to arrays of vnodes when accessed in the child component.

```js
// 单个默认插槽
h(MyComponent, () => 'hello')
// 具名插槽
// 注意 `null` 是必需的
// 以避免 slot 对象被当成 prop 处理
h(MyComponent, null, {
    default: () => 'default slot',
    foo: () => h('div', 'foo'),
    bar: () => [h('span', 'one'), h('span', 'two')]
})
```

JSX equivalent:

```html
// 默认插槽
<MyComponent>{() => 'hello'}</MyComponent>
// 具名插槽
<MyComponent>{{
  default: () => 'default slot',
  foo: () => <div>foo</div>,
  bar: () => [<span>one</span>, <span>two</span>]
```

```
}
```

等价 JSX 语法：

```html
// 默认插槽
<div>{slots.default()}</div>
// 具名插槽
<div>{slots.footer({ text: props.message })}</div>
```

传递插槽

向组件传递子元素的方式与向元素传递子元素的方式有些许不同。我们需要传递一个插槽函数或者是一个包含插槽函数的对象而非是数组，插槽函数的返回值同一个正常的渲染函数的返回值一样——并且在子组件中被访问时总是会被转化为一个 vnodes 数组。

```js
// 单个默认插槽
h(MyComponent, () => 'hello')
// 具名插槽
// 注意 `null` 是必需的
// 以避免 slot 对象被当成 prop 处理
h(MyComponent, null, {
    default: () => 'default slot',
    foo: () => h('div', 'foo'),
    bar: () => [h('span', 'one'), h('span', 'two')]
})
```

等价 JSX 语法：

```html
// 默认插槽
<MyComponent>{() => 'hello'}</MyComponent>
// 具名插槽
<MyComponent>{{
  default: () => 'default slot',
  foo: () => <div>foo</div>,
  bar: () => [<span>one</span>, <span>two</span>]
}}</MyComponent>
```

Passing slots as functions allows them to be invoked lazily by the child component. This leads to the slot's dependencies being tracked by the child instead of the parent, which results in more accurate and efficient updates.

Built-in Components

Built-in components such as `<KeepAlive>`, `<Transition>`, `<TransitionGroup>`, `<Teleport>` and `<Suspense>` must be imported for use in render functions:

```js
import { h, KeepAlive, Teleport, Transition, TransitionGroup } from 'vue'
export default {
  setup () {
    return () => h(Transition, { mode: 'out-in' }, /* ... */)
  }
}
```

v-model

The `v-model` directive is expanded to `modelValue` and `onUpdate:modelValue` props during template compilation—we will have to provide these props ourselves:

```js
export default {
  props: ['modelValue'],
  emits: ['update:modelValue'],
  setup(props, { emit }) {
    return () =>
      h(SomeComponent, {
        modelValue: props.modelValue,
        'onUpdate:modelValue': (value) => emit('update:modelValue', value)
      })
  }
}
```

```
}}</MyComponent>
```

插槽以函数的形式传递使得它们可以被子组件懒调用。这能确保它被注册为子组件的依赖关系，而不是父组件。这使得更新更加准确及有效。

内置组件

诸如 `<KeepAlive>`、`<Transition>`、`<TransitionGroup>`、`<Teleport>` 和 `<Suspense>` 等内置组件在渲染函数中必须导入才能使用：

```js
import { h, KeepAlive, Teleport, Transition, TransitionGroup } from 'vue'
export default {
  setup () {
    return () => h(Transition, { mode: 'out-in' }, /* ... */)
  }
}
```

v-model

`v-model` 指令扩展为 `modelValue` 和 `onUpdate:modelValue` 在模板编译过程中，我们必须自己提供这些 props：

```js
export default {
  props: ['modelValue'],
  emits: ['update:modelValue'],
  setup(props, { emit }) {
    return () =>
      h(SomeComponent, {
        modelValue: props.modelValue,
        'onUpdate:modelValue': (value) => emit('update:modelValue', value)
      })
  }
}
```

Custom Directives

Custom directives can be applied to a vnode using `withDirectives`:

```js
import { h, withDirectives } from 'vue'
// 自定义指令
const pin = {
  mounted() { /* ... */ },
  updated() { /* ... */ }
}
// <div v-pin:top.animate="200"></div>
const vnode = withDirectives(h('div'), [
  [pin, 200, 'top', { animate: true }]
])
```

If the directive is registered by name and cannot be imported directly, it can be resolved using the `resolveDirective` helper.

Template Refs

With the Composition API, template refs are created by passing the `ref()` itself as a prop to the vnode:

```js
import { h, ref } from 'vue'
export default {
  setup() {
    const divEl = ref()
    // <div ref="divEl">
    return () => h('div', { ref: divEl })
  }
}
```

9.5.4 Functional Components

Functional components are an alternative form of component that don't have any state of their own. They act like pure functions: props in, vnodes out. They are rendered without creating a

自定义指令

可以使用 `withDirectives` 将自定义指令应用于 vnode：

```js
import { h, withDirectives } from 'vue'
// 自定义指令
const pin = {
  mounted() { /* ... */ },
  updated() { /* ... */ }
}
// <div v-pin:top.animate="200"></div>
const vnode = withDirectives(h('div'), [
  [pin, 200, 'top', { animate: true }]
])
```

当一个指令是以名称注册并且不能被直接导入时，可以使用 `resolveDirective` 函数来解决这个问题。

模板引用

在组合式 API 中，模板引用通过将 `ref()` 本身作为一个属性传递给 vnode 来创建：

```js
import { h, ref } from 'vue'
export default {
  setup() {
    const divEl = ref()
    // <div ref="divEl">
    return () => h('div', { ref: divEl })
  }
}
```

9.5.4 函数式组件

函数式组件是一种定义自身没有任何状态的组件的方式。它们很像纯函数：接收 props，返回 vnodes。函数式组件在渲染过程中不会创建组件实例 (也就是说，没

component instance (i.e. no this), and without the usual component lifecycle hooks.

To create a functional component we use a plain function, rather than an options object. The function is effectively the render function for the component.

The signature of a functional component is the same as the setup() hook:

```js
function MyComponent(props, { slots, emit, attrs }) {
  // ...
}
```

Most of the usual configuration options for components are not available for functional components. However, it is possible to define props and emits by adding them as properties:

```js
MyComponent.props = ['value']
MyComponent.emits = ['click']
```

If the props option is not specified, then the props object passed to the function will contain all attributes, the same as attrs. The prop names will not be normalized to camelCase unless the props option is specified.

For functional components with explicit props, attribute fallthrough works much the same as with normal components. However, for functional components that don't explicitly specify their props, only the class, style, and onXxx event listeners will be inherited from the attrs by default. In either case, inheritAttrs can be set to false to disable attribute inheritance:

```js
MyComponent.inheritAttrs = false
```

Functional components can be registered and consumed just like normal components. If you pass a function as the first argument to h(), it will be treated as a functional component.

Typing Functional Components

Functional Components can be typed based on whether they are named or anonymous. Volar also supports type checking properly typed functional components when consuming them in SFC templates.

Named Functional Component

有 this)，也不会触发常规的组件生命周期钩子。

我们用一个普通的函数而不是一个选项对象来创建函数式组件。该函数实际上就是该组件的渲染函数。

函数式组件的签名与 setup() 钩子相同：

```js
function MyComponent(props, { slots, emit, attrs }) {
  // ...
}
```

大多数常规组件的配置选项在函数式组件中都不可用，除了 props 和 emits。我们可以给函数式组件添加对应的属性来声明它们：

```js
MyComponent.props = ['value']
MyComponent.emits = ['click']
```

如果这个 props 选项没有被定义，那么被传入函数的 props 对象就会像 attrs 一样会包含所有 attribute。除非指定了 props 选项，否则每个 prop 的名字将不会基于驼峰命名法被一般化处理。

对于有明确 props 的函数式组件，attribute 透传的原理与普通组件基本相同。然而，对于没有明确指定 props 的函数式组件，只有 class、style 和 onXxx 事件监听器将默认从 attrs 中继承。在这两种情况下，可以将 inheritAttrs 设置为 false 来禁用属性继承：

```js
MyComponent.inheritAttrs = false
```

函数式组件可以像普通组件一样被注册和使用。如果你将一个函数作为第一个参数传入 h，它将会被当作一个函数式组件来对待。

为函数式组件标注类型

函数式组件可以根据它们是否有命名来标注类型。在单文件组件模板中，Volar 还支持对正确类型化的函数式组件进行类型检查。

具名函数式组件

html

```
import type { SetupContext } from 'vue'
type FComponentProps = {
  message: string
}
type Events = {
  sendMessage(message: string): void
}
function FComponent(
  props: FComponentProps,
  context: SetupContext<Events>
) {
  return (
    <button onClick={() => context.emit('sendMessage', props.message)}>
        {props.message} {' '}
    </button>
  )
}
FComponent.props = {
  message: {
    type: String,
    required: true
  }
}
FComponent.emits = {
  sendMessage: (value: unknown) => typeof value === 'string'
}
```

Anonymous Functional Component

html

```
import type { FunctionalComponent } from 'vue'
type FComponentProps = {
  message: string
}
type Events = {
  sendMessage(message: string): void
```

html

```
import type { SetupContext } from 'vue'
type FComponentProps = {
  message: string
}
type Events = {
  sendMessage(message: string): void
}
function FComponent(
  props: FComponentProps,
  context: SetupContext<Events>
) {
  return (
    <button onClick={() => context.emit('sendMessage', props.message)}>
        {props.message} {' '}
    </button>
  )
}
FComponent.props = {
  message: {
    type: String,
    required: true
  }
}
FComponent.emits = {
  sendMessage: (value: unknown) => typeof value === 'string'
}
```

匿名函数式组件

html

```
import type { FunctionalComponent } from 'vue'
type FComponentProps = {
  message: string
}
type Events = {
  sendMessage(message: string): void
}
const FComponent: FunctionalComponent<FComponentProps, Events> = (
  props,
  context
) => {
  return (
    <button onClick={() => context.emit('sendMessage', props.message)}>
        {props.message} {' '}
    </button>
  )
}
FComponent.props = {
  message: {
    type: String,
    required: true
  }
}
FComponent.emits = {
  sendMessage: (value) => typeof value === 'string'
}
```

9.6 Vue and Web Components

Web Components is an umbrella term for a set of web native APIs that allows developers to create reusable custom elements.

We consider Vue and Web Components to be primarily complementary technologies. Vue has excellent support for both consuming and creating custom elements. Whether you are integrating custom elements into an existing Vue application, or using Vue to build and distribute custom elements, you are in good company.

9.6.1 Using Custom Elements in Vue

Vue scores a perfect 100% in the Custom Elements Everywhere tests. Consuming custom elements inside a Vue application largely works the same as using native HTML elements, with a few things

```
}
const FComponent: FunctionalComponent<FComponentProps, Events> = (
  props,
  context
) => {
  return (
    <button onClick={() => context.emit('sendMessage', props.message)}>
        {props.message} {' '}
    </button>
  )
}
FComponent.props = {
  message: {
    type: String,
    required: true
  }
}
FComponent.emits = {
  sendMessage: (value) => typeof value === 'string'
}
```

9.6 Vue 与 Web Components

Web Components 是一组 web 原生 API 的统称，允许开发者创建可复用的自定义元素 (custom elements)。

我们认为 Vue 和 Web Components 是互补的技术。Vue 为使用和创建自定义元素提供了出色的支持。无论你是将自定义元素集成到现有的 Vue 应用中，还是使用 Vue 来构建和分发自定义元素都很方便。

9.6.1 在 Vue 中使用自定义元素

Vue 在 Custom Elements Everywhere 测试中取得了 100% 的分数。在 Vue 应用中使用自定义元素基本上与使用原生 HTML 元素的效果相同，但需要留意以下几

to keep in mind:

Skipping Component Resolution

By default, Vue will attempt to resolve a non-native HTML tag as a registered Vue component before falling back to rendering it as a custom element. This will cause Vue to emit a ”failed to resolve component” warning during development. To let Vue know that certain elements should be treated as custom elements and skip component resolution, we can specify the `compilerOptions.isCustomElement` option.

If you are using Vue with a build setup, the option should be passed via build configs since it is a compile-time option.

Example In-Browser Config

```js
// 仅在浏览器内编译时才会工作
// 如果使用了构建工具，请看下面的配置示例
app.config.compilerOptions.isCustomElement = (tag) => tag.includes('-')
```

Example Vite Config

```js
// vite.config.js
import vue from '@vitejs/plugin-vue'
export default {
  plugins: [
    vue({
      template: {
        compilerOptions: {
          // 将所有带短横线的标签名都视为自定义元素
          isCustomElement: (tag) => tag.includes('-')
        }
      }
    })
  ]
}
```

Example Vue CLI Config

点：

跳过组件解析

默认情况下，Vue 会将任何非原生的 HTML 标签优先当作 Vue 组件处理，而将 “渲染一个自定义元素” 作为后备选项。这会在开发时导致 Vue 抛出一个 “解析组件失败” 的警告。要让 Vue 知晓特定元素应该被视为自定义元素并跳过组件解析，我们可以指定 `compilerOptions.isCustomElement` 这个选项。

如果在开发 Vue 应用时进行了构建配置，则应该在构建配置中传递该选项，因为它是一个编译时选项。

浏览器内编译时的示例配置

```js
// 仅在浏览器内编译时才会工作
// 如果使用了构建工具，请看下面的配置示例
app.config.compilerOptions.isCustomElement = (tag) => tag.includes('-')
```

Vite 示例配置

```js
// vite.config.js
import vue from '@vitejs/plugin-vue'
export default {
  plugins: [
    vue({
      template: {
        compilerOptions: {
          // 将所有带短横线的标签名都视为自定义元素
          isCustomElement: (tag) => tag.includes('-')
        }
      }
    })
  ]
}
```

Vue CLI 示例配置

js
```js
// vue.config.js
module.exports = {
  chainWebpack: config => {
    config.module
      .rule('vue')
      .use('vue-loader')
      .tap(options => ({
        ...options,
        compilerOptions: {
          // 将所有以 ion- 开头的标签都视为自定义元素
          isCustomElement: tag => tag.startsWith('ion-')
        }
      }))
  }
}
```

Passing DOM Properties

Since DOM attributes can only be strings, we need to pass complex data to custom elements as DOM properties. When setting props on a custom element, Vue 3 automatically checks DOM-property presence using the `in` operator and will prefer setting the value as a DOM property if the key is present. This means that, in most cases, you won't need to think about this if the custom element follows the recommended best practices.

However, there could be rare cases where the data must be passed as a DOM property, but the custom element does not properly define/reflect the property (causing the `in` check to fail). In this case, you can force a `v-bind` binding to be set as a DOM property using the `.prop` modifier:

html
```html
<my-element :user.prop="{ name: 'jack' }"></my-element>
<!-- 等价简写 -->
<my-element .user="{ name: 'jack' }"></my-element>
```

9.6.2 Building Custom Elements with Vue

js
```js
// vue.config.js
module.exports = {
  chainWebpack: config => {
    config.module
      .rule('vue')
      .use('vue-loader')
      .tap(options => ({
        ...options,
        compilerOptions: {
          // 将所有以 ion- 开头的标签都视为自定义元素
          isCustomElement: tag => tag.startsWith('ion-')
        }
      }))
  }
}
```

传递 DOM 属性

由于 DOM attribute 只能为字符串值，因此我们只能使用 DOM 对象的属性来传递复杂数据。当为自定义元素设置 props 时，Vue 3 将通过 `in` 操作符自动检查该属性是否已经存在于 DOM 对象上，并且在这个 key 存在时，更倾向于将值设置为一个 DOM 对象的属性。这意味着，在大多数情况下，如果自定义元素遵循推荐的最佳实践，你就不需要考虑这个问题。

然而，也会有一些特别的情况：必须将数据以一个 DOM 对象属性的方式传递，但该自定义元素无法正确地定义/反射这个属性 (因为 `in` 检查失败)。在这种情况下，你可以强制使用一个 `v-bind` 绑定、通过 `.prop` 修饰符来设置该 DOM 对象的属性：

html
```html
<my-element :user.prop="{ name: 'jack' }"></my-element>
<!-- 等价简写 -->
<my-element .user="{ name: 'jack' }"></my-element>
```

9.6.2 使用 Vue 构建自定义元素

The primary benefit of custom elements is that they can be used with any framework, or even without a framework. This makes them ideal for distributing components where the end consumer may not be using the same frontend stack, or when you want to insulate the end application from the implementation details of the components it uses.

defineCustomElement

Vue supports creating custom elements using exactly the same Vue component APIs via the `defineCustomElement` method. The method accepts the same argument as `defineComponent`, but instead returns a custom element constructor that extends `HTMLElement`:

html
```html
<my-vue-element></my-vue-element>
```

js
```js
import { defineCustomElement } from 'vue'
const MyVueElement = defineCustomElement({
  // 这里是同平常一样的 Vue 组件选项
  props: {},
  emits: {},
  template: `...`,
  // defineCustomElement 特有的：注入进 shadow root 的 CSS
  styles: [`/* inlined css */`]
})
// 注册自定义元素
// 注册之后，所有此页面中的 `<my-vue-element>` 标签
// 都会被升级
customElements.define('my-vue-element', MyVueElement)
// 你也可以编程式地实例化元素：
// （必须在注册之后）
document.body.appendChild(
  new MyVueElement({
    // 初始化 props（可选）
  })
)
```

Lifecycle

- A Vue custom element will mount an internal Vue component instance inside its shadow root

自定义元素的主要好处是，它们可以在使用任何框架，甚至是在不使用框架的场景下使用。当你面向的最终用户可能使用了不同的前端技术栈，或是当你希望将最终的应用与它使用的组件实现细节解耦时，它们会是理想的选择。

defineCustomElement

Vue 提供了一个和定义一般 Vue 组件几乎完全一致的 `defineCustomElement` 方法来支持创建自定义元素。这个方法接收的参数和 `defineComponent` 完全相同。但它会返回一个继承自 `HTMLElement` 的自定义元素构造器：

html
```html
<my-vue-element></my-vue-element>
```

js
```js
import { defineCustomElement } from 'vue'
const MyVueElement = defineCustomElement({
  // 这里是同平常一样的 Vue 组件选项
  props: {},
  emits: {},
  template: `...`,
  // defineCustomElement 特有的：注入进 shadow root 的 CSS
  styles: [`/* inlined css */`]
})
// 注册自定义元素
// 注册之后，所有此页面中的 `<my-vue-element>` 标签
// 都会被升级
customElements.define('my-vue-element', MyVueElement)
// 你也可以编程式地实例化元素：
// （必须在注册之后）
document.body.appendChild(
  new MyVueElement({
    // 初始化 props（可选）
  })
)
```

生命周期

- 当该元素的 `connectedCallback` 初次调用时，一个 Vue 自定义元素会在内

when the element's `connectedCallback` is called for the first time.

- When the element's `disconnectedCallback` is invoked, Vue will check whether the element is detached from the document after a microtask tick.
  - If the element is still in the document, it's a move and the component instance will be preserved;
  - If the element is detached from the document, it's a removal and the component instance will be unmounted.

Props

- All props declared using the `props` option will be defined on the custom element as properties. Vue will automatically handle the reflection between attributes / properties where appropriate.
  - Attributes are always reflected to corresponding properties.
  - Properties with primitive values (`string`, `boolean` or `number`) are reflected as attributes.
- Vue also automatically casts props declared with `Boolean` or `Number` types into the desired type when they are set as attributes (which are always strings). For example, given the following props declaration:

```html
props: {
  selected: Boolean,
  index: Number
}
```

And the custom element usage:

```html
<my-element selected index="1"></my-element>
```

In the component, `selected` will be cast to `true` (boolean) and `index` will be cast to `1` (number).

Events

Events emitted via `this.$emit` or setup `emit` are dispatched as native CustomEvents on the custom element. Additional event arguments (payload) will be exposed as an array on the CustomEvent object as its `detail` property.

部挂载一个 Vue 组件实例到它的 shadow root 上。

- 当此元素的 `disconnectedCallback` 被调用时，Vue 会在一个微任务后检查元素是否还留在文档中。
  - 如果元素仍然在文档中，那么说明它是一次移动操作，组件实例将被保留；
  - 如果该元素不再存在于文档中，那么说明这是一次移除操作，组件实例将被销毁。

Props

- 所有使用 `props` 选项声明了的 props 都会作为属性定义在该自定义元素上。Vue 会自动地、恰当地处理其作为 attribute 还是属性的反射。
  - attribute 总是根据需要反射为相应的属性类型。
  - 基础类型的属性值 (`string`, `boolean` 或 `number`) 会被反射为 attribute。
- 当它们被设为 attribute 时 (永远是字符串)，Vue 也会自动将以 `Boolean` 或 `Number` 类型声明的 prop 转换为所期望的类型。比如下面这样的 props 声明：

```html
props: {
  selected: Boolean,
  index: Number
}
```

并以下面这样的方式使用自定义元素：

```html
<my-element selected index="1"></my-element>
```

在组件中，`selected` 会被转换为 `true` (boolean 类型值) 而 `index` 会被转换为 `1` (number 类型值)。

事件

通过 `this.$emit` 或者 setup 中的 `emit` 触发的事件都会通过以 CustomEvents 的形式从自定义元素上派发。额外的事件参数 (payload) 将会被暴露为 CustomEvent 对象上的一个 `detail` 数组。

Slots

Inside the component, slots can be rendered using the `<slot/>` element as usual. However, when consuming the resulting element, it only accepts native slots syntax:

- Scoped slots are not supported.
- When passing named slots, use the `slot` attribute instead of the `v-slot` directive:

```html
<my-element>
  <div slot="named">hello</div>
</my-element>
```

Provide / Inject

The Provide / Inject API and its Composition API equivalent also work between Vue-defined custom elements. However, note that this works **only between custom elements**. i.e. a Vue-defined custom element won't be able to inject properties provided by a non-custom-element Vue component.

SFC as Custom Element

`defineCustomElement` also works with Vue Single-File Components (SFCs). However, with the default tooling setup, the `<style>` inside the SFCs will still be extracted and merged into a single CSS file during production build. When using an SFC as a custom element, it is often desirable to inject the `<style>` tags into the custom element's shadow root instead.

The official SFC toolings support importing SFCs in "custom element mode" (requires `@vitejs/plugin-vue@^1.4.0` or `vue-loader@^16.5.0`). An SFC loaded in custom element mode inlines its `<style>` tags as strings of CSS and exposes them under the component's `styles` option. This will be picked up by `defineCustomElement` and injected into the element's shadow root when instantiated.

To opt-in to this mode, simply end your component file name with `.ce.vue`:

```js
import { defineCustomElement } from 'vue'
import Example from './Example.ce.vue'
console.log(Example.styles) // ["/* 内联 css */"]
// 转换为自定义元素构造器
const ExampleElement = defineCustomElement(Example)
```

插槽

在一个组件中，插槽将会照常使用 `<slot/>` 渲染。然而，当使用最终的元素时，它只接受原生插槽的语法：

- 不支持作用域插槽。
- 当传递具名插槽时，应使用 `slot` attribute 而不是 `v-slot` 指令：

```html
<my-element>
  <div slot="named">hello</div>
</my-element>
```

依赖注入

Provide / Inject API 和相应的组合式 API 在 Vue 定义的自定义元素中都可以正常工作。但是请注意，依赖关系**只在自定义元素之间**起作用。例如一个 Vue 定义的自定义元素就无法注入一个由常规 Vue 组件所提供的属性。

将 SFC 编译为自定义元素

`defineCustomElement` 也可以搭配 Vue 单文件组件 (SFC) 使用。但是，根据默认的工具链配置，SFC 中的 `<style>` 在生产环境构建时仍然会被抽取和合并到一个单独的 CSS 文件中。当正在使用 SFC 编写自定义元素时，通常需要改为注入 `<style>` 标签到自定义元素的 shadow root 上。

官方的 SFC 工具链支持以 "自定义元素模式" 导入 SFC (需要 `@vitejs/plugin-vue@^1.4.0` 或 `vue-loader@^16.5.0`)。一个以自定义元素模式加载的 SFC 将会内联其 `<style>` 标签为 CSS 字符串，并将其暴露为组件的 `styles` 选项。这会被 `defineCustomElement` 提取使用，并在初始化时注入到元素的 shadow root 上。

要开启这个模式，只需要将你的组件文件以 `.ce.vue` 结尾即可：

```js
import { defineCustomElement } from 'vue'
import Example from './Example.ce.vue'
console.log(Example.styles) // ["/* 内联 css */"]
// 转换为自定义元素构造器
const ExampleElement = defineCustomElement(Example)
```

```
// 注册
customElements.define('my-example', ExampleElement)
```

If you wish to customize what files should be imported in custom element mode (for example, treating *all* SFCs as custom elements), you can pass the `customElement` option to the respective build plugins:

- @vitejs/plugin-vue
- vue-loader

Tips for a Vue Custom Elements Library

When building custom elements with Vue, the elements will rely on Vue's runtime. There is a ~16kb baseline size cost depending on how many features are being used. This means it is not ideal to use Vue if you are shipping a single custom element - you may want to use vanilla JavaScript, petite-vue, or frameworks that specialize in small runtime size. However, the base size is more than justifiable if you are shipping a collection of custom elements with complex logic, as Vue will allow each component to be authored with much less code. The more elements you are shipping together, the better the trade-off.

If the custom elements will be used in an application that is also using Vue, you can choose to externalize Vue from the built bundle so that the elements will be using the same copy of Vue from the host application.

It is recommended to export the individual element constructors to give your users the flexibility to import them on-demand and register them with desired tag names. You can also export a convenience function to automatically register all elements. Here's an example entry point of a Vue custom element library:

js

```
import { defineCustomElement } from 'vue'
import Foo from './MyFoo.ce.vue'
import Bar from './MyBar.ce.vue'
const MyFoo = defineCustomElement(Foo)
const MyBar = defineCustomElement(Bar)
// 分别导出元素
export { MyFoo, MyBar }
export function register() {
```

```
// 注册
customElements.define('my-example', ExampleElement)
```

如果你想要自定义如何判断是否将文件作为自定义元素导入 (例如将所有的 SFC 都视为用作自定义元素)，你可以通过给构建插件传递相应插件的 `customElement` 选项来实现：

- @vitejs/plugin-vue
- vue-loader

基于 Vue 构建自定义元素库

当使用 Vue 构建自定义元素时，该元素将依赖于 Vue 的运行时。这会有大约 16kb 的基本打包大小，并视功能的使用情况而增长。这意味着如果只编写一个自定义元素，那么使用 Vue 并不是理想的选择。你可能想要使用原生 JavaScript、petite-vue，或其他框架以追求更小的运行时体积。但是，如果你需要编写的是一组具有复杂逻辑的自定义元素，那么这个基本体积是非常合理的，因为 Vue 允许用更少的代码编写每个组件。在一起发布的元素越多，收益就会越高。

如果自定义元素将在同样使用 Vue 的应用中使用，那么你可以选择将构建包中的 Vue 外部化 (externalize)，这样这些自定义元素将与宿主应用使用同一份 Vue。

建议按元素分别导出构造函数，以便用户可以灵活地按需导入它们，并使用期望的标签名称注册它们。你还可以导出一个函数来方便用户自动注册所有元素。下面是一个 Vue 自定义元素库的入口文件示例：

js

```
import { defineCustomElement } from 'vue'
import Foo from './MyFoo.ce.vue'
import Bar from './MyBar.ce.vue'
const MyFoo = defineCustomElement(Foo)
const MyBar = defineCustomElement(Bar)
// 分别导出元素
export { MyFoo, MyBar }
export function register() {
  customElements.define('my-foo', MyFoo)
  customElements.define('my-bar', MyBar)
}
```

If you have many components, you can also leverage build tool features such as Vite's glob import or webpack's `require.context` to load all components from a directory.

Web Components and Typescript

If you are developing an application or a library, you may want to type check your Vue components, including those that are defined as custom elements.

Custom elements are registered globally using native APIs, so by default they won't have type inference when used in Vue templates. To provide type support for Vue components registered as custom elements, we can register global component typings using the the `GlobalComponents` interface in Vue templates and/or in JSX:

html

```
import { defineCustomElement } from 'vue'
// vue 单文件组件
import CounterSFC from './src/components/counter.ce.vue'
// 将组件转换为 web components
export const Counter = defineCustomElement(CounterSFC)
// 注册全局类型
declare module 'vue' {
  export interface GlobalComponents {
    'Counter': typeof Counter,
  }
}
```

9.6.3 Web Components vs. Vue Components

Some developers believe that framework-proprietary component models should be avoided, and that exclusively using Custom Elements makes an application "future-proof". Here we will try to explain why we believe that this is an overly simplistic take on the problem.

There is indeed a certain level of feature overlap between Custom Elements and Vue Components:

```
  customElements.define('my-foo', MyFoo)
  customElements.define('my-bar', MyBar)
}
```

如果你有非常多的组件，你也可以利用构建工具的功能，比如 Vite 的 glob 导入或者 webpack 的 `require.context` 来从一个文件夹加载所有的组件。

Web Components 和 Typescript

如果你正在开发一个应用或者库，你可能想要为你的 Vue 组件添加类型检查，包括那些被定义为自定义元素的组件。

自定义元素是使用原生 API 全局注册的，所以默认情况下，当在 Vue 模板中使用时，它们不会有类型推断。为了给注册为自定义元素的 Vue 组件提供类型支持，我们可以通过 Vue 模板和/或 JSX 中的 `GlobalComponents` 接口 来注册全局组件的类型：

html

```
import { defineCustomElement } from 'vue'
// vue 单文件组件
import CounterSFC from './src/components/counter.ce.vue'
// 将组件转换为 web components
export const Counter = defineCustomElement(CounterSFC)
// 注册全局类型
declare module 'vue' {
  export interface GlobalComponents {
    'Counter': typeof Counter,
  }
}
```

9.6.3 Web Components vs. Vue Components

一些开发者认为应该避免使用框架专有的组件模型，而改为全部使用自定义元素来构建应用，因为这样可以使应用 “永不过时”。在这里，我们将解释为什么我们认为这样的想法过于简单。

自定义元素和 Vue 组件之间确实存在一定程度的功能重叠：它们都允许我们定义

they both allow us to define reusable components with data passing, event emitting, and lifecycle management. However, Web Components APIs are relatively low-level and bare-bones. To build an actual application, we need quite a few additional capabilities which the platform does not cover:

- A declarative and efficient templating system;
- A reactive state management system that facilitates cross-component logic extraction and reuse;
- A performant way to render the components on the server and hydrate them on the client (SSR), which is important for SEO and Web Vitals metrics such as LCP. Native custom elements SSR typically involves simulating the DOM in Node.js and then serializing the mutated DOM, while Vue SSR compiles into string concatenation whenever possible, which is much more efficient.

Vue's component model is designed with these needs in mind as a coherent system.

With a competent engineering team, you could probably build the equivalent on top of native Custom Elements - but this also means you are taking on the long-term maintenance burden of an in-house framework, while losing out on the ecosystem and community benefits of a mature framework like Vue.

There are also frameworks built using Custom Elements as the basis of their component model, but they all inevitably have to introduce their proprietary solutions to the problems listed above. Using these frameworks entails buying into their technical decisions on how to solve these problems - which, despite what may be advertised, doesn't automatically insulate you from potential future churns.

There are also some areas where we find custom elements to be limiting:

- Eager slot evaluation hinders component composition. Vue's scoped slots are a powerful mechanism for component composition, which can't be supported by custom elements due to native slots' eager nature. Eager slots also mean the receiving component cannot control when or whether to render a piece of slot content.
- Shipping custom elements with shadow DOM scoped CSS today requires embedding the CSS inside JavaScript so that they can be injected into shadow roots at runtime. They also result in duplicated styles in markup in SSR scenarios. There are platform features being worked on in this area - but as of now they are not yet universally supported, and there are still production performance / SSR concerns to be addressed. In the meanwhile, Vue SFCs provide CSS

具有数据传递、事件发射和生命周期管理的可重用组件。然而，Web Components 的 API 相对来说是更底层的和更基础的。要构建一个实际的应用，我们需要相当多平台没有涵盖的附加功能：

- 一个声明式的、高效的模板系统；
- 一个响应式的，利于跨组件逻辑提取和重用的状态管理系统；
- 一种在服务器上呈现组件并在客户端“激活”(hydrate) 组件的高性能方法 (SSR)，这对 SEO 和 LCP 这样的 Web 关键指标非常重要。原生自定义元素 SSR 通常需要在 Node.js 中模拟 DOM，然后序列化更改后的 DOM，而 Vue SSR 则尽可能地将其编译为拼接起来的字符串，这会高效得多。

Vue 的组件模型在设计时同时兼顾了这些需求，因此是一个更内聚的系统。

当你的团队有足够的技术水平时，可能可以在原生自定义元素的基础上构建具备同等功能的组件。但这也意味着你将承担长期维护内部框架的负担，同时失去了像 Vue 这样成熟的框架生态社区所带来的收益。

也有一些框架使用自定义元素作为其组件模型的基础，但它们都不可避免地要引入自己的专有解决方案来解决上面列出的问题。使用这些框架便意味着对它们针对这些问题的技术决策买单。不管这类框架怎么宣传它们“永不过时”，它们其实都无法保证你以后永远不需要重构。

除此之外，我们还发现自定义元素存在以下限制：

- 贪婪 (eager) 的插槽求值会阻碍组件之间的可组合性。Vue 的作用域插槽是一套强大的组件组合机制，而由于原生插槽的贪婪求值性质，自定义元素无法支持这样的设计。贪婪求值的插槽也意味着接收组件时不能控制何时或是否创建插槽内容的节点。
- 在当下要想使用 shadow DOM 书写局部作用域的 CSS，必须将样式嵌入到 JavaScript 中才可以在运行时将其注入到 shadow root 上。这也导致了 SSR 场景下需要渲染大量重复的样式标签。虽然有一些平台功能在尝试解决这一领域的问题，但是直到现在还没有达到通用支持的状态，而且仍有生产性能 / SSR 方面的问题需要解决。可与此同时，Vue 的 SFC 本身就提供了 CSS

scoping mechanisms that support extracting the styles into plain CSS files.

Vue will always stay up to date with the latest standards in the web platform, and we will happily leverage whatever the platform provides if it makes our job easier. However, our goal is to provide solutions that work well and work today. That means we have to incorporate new platform features with a critical mindset - and that involves filling the gaps where the standards fall short while that is still the case.

9.7 Animation Techniques

Vue provides the “ and “ components for handling enter / leave and list transitions. However, there are many other ways of using animations on the web, even in a Vue application. Here we will discuss a few additional techniques.

9.7.1 Class-based Animations

For elements that are not entering / leaving the DOM, we can trigger animations by dynamically adding a CSS class:

js

```js
const disabled = ref(false)
function warnDisabled() {
  disabled.value = true
  setTimeout(() => {
    disabled.value = false
  }, 1500)
}
```

html

```html
<div :class="{ shake: disabled }">
  <button @click="warnDisabled">Click me</button>
  <span v-if="disabled">This feature is disabled!</span>
</div>
```

css

```css
.shake {
  animation: shake 0.82s cubic-bezier(0.36, 0.07, 0.19, 0.97) both;
  transform: translate3d(0, 0, 0);
}
```

局域化机制，并支持抽取样式到纯 CSS 文件中。

Vue 将始终紧跟 Web 平台的最新标准，如果平台的新功能能让我们的工作变得更简单，我们将非常乐于利用它们。但是，我们的目标是提供 “好用，且现在就能用” 的解决方案。这意味着我们在采用新的原生功能时需要保持客观、批判性的态度，并在原生功能完成度不足的时候选择更适当的解决方案。

9.7 动画技巧

Vue 提供了 “ 和 “ 组件来处理元素进入、离开和列表顺序变化的过渡效果。但除此之外，还有许多其他制作网页动画的方式在 Vue 应用中也适用。这里我们会探讨一些额外的技巧。

9.7.1 基于 CSS class 的动画

对于那些不是正在进入或离开 DOM 的元素，我们可以通过给它们动态添加 CSS class 来触发动画：

js

```js
const disabled = ref(false)
function warnDisabled() {
  disabled.value = true
  setTimeout(() => {
    disabled.value = false
  }, 1500)
}
```

html

```html
<div :class="{ shake: disabled }">
  <button @click="warnDisabled">Click me</button>
  <span v-if="disabled">This feature is disabled!</span>
</div>
```

css

```css
.shake {
  animation: shake 0.82s cubic-bezier(0.36, 0.07, 0.19, 0.97) both;
  transform: translate3d(0, 0, 0);
}
```

```
@keyframes shake {
  10%,
  90% {
    transform: translate3d(-1px, 0, 0);
  }
  20%,
  80% {
    transform: translate3d(2px, 0, 0);
  }
  30%,
  50%,
  70% {
    transform: translate3d(-4px, 0, 0);
  }
  40%,
  60% {
    transform: translate3d(4px, 0, 0);
  }
}
```

9.7.2 State-driven Animations

Some transition effects can be applied by interpolating values, for instance by binding a style to an element while an interaction occurs. Take this example for instance:

js

```
const x = ref(0)
function onMousemove(e) {
  x.value = e.clientX
}
```

html

```
<div
  @mousemove="onMousemove"
  :style="{ backgroundColor: `hsl(${x}, 80%, 50%)` }"
  class="movearea"
>
```

```
@keyframes shake {
  10%,
  90% {
    transform: translate3d(-1px, 0, 0);
  }
  20%,
  80% {
    transform: translate3d(2px, 0, 0);
  }
  30%,
  50%,
  70% {
    transform: translate3d(-4px, 0, 0);
  }
  40%,
  60% {
    transform: translate3d(4px, 0, 0);
  }
}
```

9.7.2 状态驱动的动画

有些过渡效果可以通过动态插值来实现，比如在交互时动态地给元素绑定样式。看下面这个例子：

js

```
const x = ref(0)
function onMousemove(e) {
  x.value = e.clientX
}
```

html

```
<div
  @mousemove="onMousemove"
  :style="{ backgroundColor: `hsl(${x}, 80%, 50%)` }"
  class="movearea"
>
```

```
  <p>Move your mouse across this div...</p>
  <p>x: {{ x }}</p>
</div>
```

css

```
.movearea {
  transition: 0.3s background-color ease;
}
```

In addition to color, you can also use style bindings to animate transform, width, or height. You can even animate SVG paths using spring physics - after all, they are all attribute data bindings:

Source code

9.7.3 Animating with Watchers

With some creativity, we can use watchers to animate anything based on some numerical state. For example, we can animate the number itself:

js

```
import { ref, reactive, watch } from 'vue'
import gsap from 'gsap'
const number = ref(0)
const tweened = reactive({
  number: 0
})
watch(number, (n) => {
  gsap.to(tweened, { duration: 0.5, number: Number(n) || 0 })
})
```

html

```
Type a number: <input v-model.number="number" />
<p>{{ tweened.number.toFixed(0) }}</p>
```

Try it in the Playground

```
  <p>Move your mouse across this div...</p>
  <p>x: {{ x }}</p>
</div>
```

css

```
.movearea {
  transition: 0.3s background-color ease;
}
```

除了颜色外，你还可以使用样式绑定 CSS transform、宽度或高度。你甚至可以通过运用弹性物理模拟为 SVG 添加动画，毕竟它们也只是 attribute 的数据绑定：

Source code

9.7.3 基于侦听器的动画

通过发挥一些创意，我们可以基于一些数字状态，配合侦听器给任何东西加上动画。例如，我们可以将数字本身变成动画：

js

```
import { ref, reactive, watch } from 'vue'
import gsap from 'gsap'
const number = ref(0)
const tweened = reactive({
  number: 0
})
watch(number, (n) => {
  gsap.to(tweened, { duration: 0.5, number: Number(n) || 0 })
})
```

html

```
Type a number: <input v-model.number="number" />
<p>{{ tweened.number.toFixed(0) }}</p>
```

在演练场中尝试一下