

Vue.js v3.0 教程 (Vue3教程)



Vue (读音 /vju:/, 类似于 view) 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是, Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层, 不仅易于上手, 还便于与第三方库或既有项目整合。...



下载手机APP
畅享精彩阅读

目 录

致谢

基础

安装

介绍

应用 & 组件实例

模板语法

Data Properties and Methods

计算属性和侦听器

Class 与 Style 绑定

条件渲染

列表渲染

事件处理

表单输入绑定

组件基础

深入组件

组件注册

Props

非 Prop 的 Attribute

自定义事件

插槽

提供 / 注入

动态组件 & 异步组件

模板引用

处理边界情况

过渡&动画

过渡 & 动画概述

进入过渡 & 离开过渡

列表过渡

状态过渡

可复用性&组合

混入

自定义指令

传入

渲染函数

插件

高阶指南

响应性

[深入响应性原理](#)

[响应式原理](#)

[响应式计算和侦听](#)

组合 API

[介绍](#)

[Setup](#)

[生命周期钩子](#)

[提供/注入](#)

[模板引用](#)

渲染机制和优化

[Vue 2 中的更改检测警告](#)

工具

[单文件组件](#)

[测试](#)

[TypeScript 支持](#)

[Mobile](#)

规模化

[路由](#)

[状态管理](#)

[服务端渲染](#)

无障碍

[基础](#)

[语义学](#)

[标准](#)

[资源](#)

从 Vue 2 迁移

[介绍](#)

[v-for 中的 Ref 数组](#)

[异步组件](#)

[attribute 强制行为](#)

[自定义指令](#)

[自定义元素交互](#)

[Data 选项](#)

[事件 API](#)

[过滤器](#)

[片段](#)

[函数式组件](#)

[全局 API](#)

[全局 API Treeshaking](#)

[内联模板 Attribute](#)

[key attribute](#)

[按键修饰符](#)

[在 prop 的默认函数中访问 this](#)

[渲染函数 API](#)

[Slot 统一](#)

[过渡的 class 名更改](#)

[v-model](#)

[v-if 与 v-for 的优先级对比](#)

[v-bind 合并行为](#)

[贡献文档](#)

[Vue 文档编写指南](#)

[文档风格指南](#)

[翻译](#)

致谢

当前文档《Vue.js v3.0 教程 (Vue3教程)》由 进击的皇虫 使用 书栈网 (BookStack.CN) 进行构建, 生成于 2020-10-21。

书栈网仅提供文档编写、整理、归类等功能, 以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理, 书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候, 发现文档内容有不恰当的地方, 请向我们反馈, 让我们共同携手, 将知识准确、高效且有效地传递给每一个人。

同时, 如果您在日常工作、生活和学习中遇到有价值有营养的知识文档, 欢迎分享到书栈网, 为知识的传承献上您的一份力量!

如果当前文档生成时间太久, 请到书栈网获取最新的文档, 以跟上知识更新换代的步伐。

内容来源: [Vue](https://github.com/vuejs/docs-next-zh-cn) <https://github.com/vuejs/docs-next-zh-cn>

文档地址: <http://www.bookstack.cn/books/vue-3.0-zh>

书栈官网: <https://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享, 让知识传承更久远! 感谢知识的创造者, 感谢知识的分享者, 也感谢每一位阅读到此处的读者, 因为我们都将成为知识的传承者。

- 安装
- 介绍
- 应用 & 组件实例
- 模板语法
- Data Properties and Methods
- 计算属性和侦听器
- Class 与 Style 绑定
- 条件渲染
- 列表渲染
- 事件处理
- 表单输入绑定
- 组件基础

安装

Vue.js 在设计上是可以逐步采纳的。这意味着它可以根据需求以多种方式集成到一个项目中。

将 Vue.js 添加到项目中有三种主要方式。

1. 在页面上以 [CDN package](#) 的形式导入。
2. 使用 [npm](#) 安装它。
3. 使用官方的 [CLI](#) 来构建一个项目，它为现代前端工作流程提供了功能齐备的构建设置（例如，热重载、保存时的提示等等）。

发布版本说明

最新版本：`npm@next v3.0.2`

每个版本的详细发行说明可在 [GitHub](#) [↗] ([opens new window](#)) 上找到。

Vue Devtools

当前是 Beta 版—Vuex 和 Router 的集成仍然是 WIP

在使用 Vue 时，我们推荐在你的浏览器上安装 [Vue Devtools](#) [↗] ([opens new window](#))，它允许你在一个更友好的界面中审查和调试 Vue 应用。

[获取 Chrome Extension](#) [↗] ([opens new window](#))

[获取 Firefox 插件](#) [↗] ([opens new window](#))

[获取标准的 Electron app](#) [↗] ([opens new window](#))

CDN

对于制作原型或学习，你可以这样使用最新版本

```
1. <script src="https://unpkg.com/vue@next"></script>
```

对于生产环境，我们推荐链接到一个明确的版本号和构建文件，以避免新版本造成的不可预期的破坏：

npm

在用 Vue 构建大型应用时推荐使用 npm 安装[\[1\]](#)。NPM 能很好地和诸如 [Webpack](#) (opens new window) 或 [Browserify](#) (opens new window) 模块打包器配合使用。同时 Vue 也提供配套工具来开发单文件组件。

1. # 最新稳定版
2. \$ npm install vue@next

命令行工具 (CLI)

Vue 提供了一个官方的 CLI (opens new window)，为单页面应用 (SPA) 快速搭建繁杂的脚手架。它为现代前端工作流提供了 batteries-included 的构建设置。只需要几分钟的时间就可以运行起来并带有热重载、保存时 lint 校验，以及生产环境可用的构建版本。更多详情可查阅 [Vue CLI 的文档](#) (opens new window)。

TIP

CLI 工具假定用户对 Node.js 和相关构建工具有一定程度的了解。如果你是新手，我们强烈建议先在不用构建工具的情况下通读[指南](#)，在熟悉 Vue 本身之后再使用 CLI。

对于 Vue 3，你应该使用 `npm` 上可用的 Vue CLI v4.5 作为 `@vue/cli@next`。要升级，你应该需要全局重新安装最新版本的 `@vue/cli`：

1. yarn global add @vue/cli@next
2. # OR
3. npm install -g @vue/cli@next

然后在 Vue 项目运行：

1. vue upgrade --next

Vite

[Vite](#) (opens new window) 是一个 web 开发构建工具，由于其原生 ES 模块导入方法，它允许快速提供代码。

通过在终端中运行以下命令，可以使用 Vite 快速构建 Vue 项目。

使用 npm:

```
1. $ npm init vite-app <project-name>
2. $ cd <project-name>
3. $ npm install
4. $ npm run dev
```

或者 yarn:

```
1. $ yarn create vite-app <project-name>
2. $ cd <project-name>
3. $ yarn
4. $ yarn dev
```

对不同构建版本的解释

在 npm 包的 [dist/ 目录](#) (opens new window) 你将会找到很多不同的 Vue.js 构建版本。这里列出了它们之间的差别:

使用 CDN 或没有构建工具

`vue(.runtime).global(.prod).js` :

- 若要通过浏览器中的 `<script src="...">` 直接使用，则暴露 Vue 全局;
- 浏览器内模板编译：
 - `vue.global.js` 是包含编译器和运行时的“完整”构建，因此它支持动态编译模板。
 - `vue.runtime.global.js` 只包含运行时，并且需要在构建步骤期间预编译模板。
- 内联所有 Vue 核心内部包—即：它是一个单独的文件，不依赖于其他文件，这意味着你必须导入此文件和此文件中的所有内容，以确保获得相同的代码实例。
- 包含硬编码的 prod/dev 分支，并且 prod 构建是预先缩小的。使用 `*.prod.js` 用于生产的文件。

提示

全局打包不是 [UMD](#) (opens new window) 构建的，它们被打包成 [IIFEs](#) (opens new window)，并且仅用于通过 `<script src="...">` 直接使用。

`vue(.runtime).esm-browser(.prod).js` :

- 用于通过原生 ES 模块导入使用（在浏览器中通过 `<script type="module">`）；
- 与全局构建共享相同的运行时编译、依赖内联和硬编码的 prod/dev 行为。

使用构建工具

`vue(.runtime).esm-bundler.js` :

- 使用构建工具像 `webpack` , `rollup` 和 `parcel` 。
- TODO: 将 prod/dev 分支留给 `process.env.NODE_ENV guards` (需要更换构建工具)
- 不提供最小化版本 (捆绑后与其余代码一起完成)
- import 依赖 (例如: `@vue/runtime-core` , `@vue/runtime-compiler`)
 - 导入的依赖项也是 esm bundler 构建, 并将依次导入其依赖项 (例如: `@vue/runtime-core imports @vue/reactivity`)
 - 这意味着你可以单独安装/导入这些依赖, 而不会导致这些依赖项的不同实例, 但你必须确保它们都为同一版本。
- 浏览器内模板编译:
- `vue.runtime.esm-bundler.js` (默认) 仅运行时, 并要求所有模板都要预先编译。这是打包工具的默认入口 (通过 `package.json` 中的 `module` 字段), 因为在使用 bundler 时, 模板通常是预先编译的 (例如: 在 `*.vue` 文件中), 你需要将打包工具配置 `vue` 别名到这个文件

对于服务端渲染

- `vue.cjs(.prod).js` :
 - 或用于 Node.js 通过 `require()` 进行服务器端渲染。
 - 如果你将应用程序与带有 `target: 'node'` 的 `webpack` 打包在一起, 并正确地将 `vue` 外部化, 则将加载此构建。
 - dev/prod 文件是预构建的, 但是根据 `process.env.NODE_env` 会自动需要相应的文件。

运行时 + 编译器 vs. 仅运行时

如果需要在客户端上编译模板 (即: 将字符串传递给 `template` 选项, 或使用其在 DOM 中 HTML 作为模板挂载到元素), 你需要编译器, 因此需要完整的版本:

```

1. // 需要编译器
2. Vue.createApp({
3.   template: '<div>{{ hi }}</div>'
4. })
5.
```

```
6. // 不需要
7. Vue.createApp({
8.   render() {
9.     return Vue.h('div', {}, this.hi)
10.  }
11. })
```

当使用 `vue-loader` 时，`*.vue` 文件中的模板在生成时预编译为 JavaScript，在最终的打包器中并不需要编译器，因此可以只使用运行时构建。

译者注[1] 对于中国大陆用户，建议将 NPM 源设置为[国内的镜像](#)[↗] (`opens new window`)，可以大幅提升安装速度。

介绍

提示

已经了解 Vue 2，只想了解 Vue 3 的新功能可以参阅[迁移指南](#)！

Vue.js 是什么

Vue (读音 /vju:/, 类似于 **view**) 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是, Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层, 不仅易于上手, 还便于与第三方库或既有项目整合。另一方面, 当与[现代化的工具链](#)以及各种[支持类库](#)[↗] ([opens new window](#)) 结合使用时, Vue 也完全能够为复杂的单页应用提供驱动。

如果你想在深入学习 Vue 之前对它有更多了解, 我们[制作了一个视频](#), 带你了解其核心概念和一个示例工程。

如果你已经是有经验的前端开发者, 想知道 Vue 与其它库/框架有哪些区别, 请查看[对比其它框架](#)。

起步

安装

TIP

官方指南假设你已了解关于 HTML、CSS 和 JavaScript 的中级知识。如果你刚开始学习前端开发, 将框架作为你的第一步可能不是最好的主意——掌握好基础知识再来吧! 之前有其它框架的使用经验会有帮助, 但这不是必需的

尝试 Vue.js 最简单的方法是使用 [Hello World 例子](#)[↗] ([opens new window](#)), 你可以在浏览器新标签页中打开它, 跟着例子学习一些基础用法。

[安装教程](#)给出了更多安装 Vue 的方式。请注意我们不推荐新手直接使用 `vue-cli`, 尤其是在你还不熟悉基于 Node.js 的构建工具时。

声明式渲染

Vue.js 的核心是一个允许采用简洁的模板语法来声明式地将数据渲染进 DOM 的系统:

```
1. <div id="counter">
```

```

2.   Counter: {{ counter }}
3. </div>

```

```

1. const Counter = {
2.   data() {
3.     return {
4.       counter: 0
5.     }
6.   }
7. }
8.
9. Vue.createApp(Counter).mount('#counter')

```

我们已经成功创建了第一个 Vue 应用！看起来这跟渲染一个字符串模板非常类似，但是 Vue 在背后做了大量工作。现在数据和 DOM 已经被建立了关联，所有东西都是响应式的。我们要怎么确认呢？请看下面的示例，其中 `counter` property 每秒递增，你将看到渲染的 DOM 是如何变化的：

```

1. const CounterApp = {
2.   data() {
3.     return {
4.       counter: 0
5.     }
6.   },
7.   mounted() {
8.     setInterval(() => {
9.       this.counter++
10.     }, 1000)
11.   }
12. }

```

Counter: 2

Stop timer

除了文本插值，我们还可以像这样绑定元素的 attribute：

```

1. <div id="bind-attribute">
2.   <span v-bind:title="message">
3.     鼠标悬停几秒钟查看此处动态绑定的提示信息！

```

```
4.   </span>
5. </div>
```

```
1. const AttributeBinding = {
2.   data() {
3.     return {
4.       message: 'You loaded this page on ' + new Date().toLocaleString()
5.     }
6.   }
7. }
8.
9. Vue.createApp(AttributeBinding).mount('#bind-attribute')
```

这里我们遇到了一点新东西。你看到的 `v-bind` attribute 被称为指令。指令带有前缀 `v-`，以表示它们是 Vue 提供的特殊 attribute。可能你已经猜到了，它们会在渲染的 DOM 上应用特殊的响应式行为。在这里，该指令的意思是：“将这个元素节点的 `title` attribute 和当前活跃实例的 `message` property 保持一致”。

处理用户输入

为了让用户和应用进行交互，我们可以用 `v-on` 指令添加一个事件监听器，通过它调用在实例中定义的方法：

```
1. <div id="event-handling">
2.   <p>{{ message }}</p>
3.   <button v-on:click="reverseMessage">反转 Message</button>
4. </div>
```

```
1. const EventHandlering = {
2.   data() {
3.     return {
4.       message: 'Hello Vue.js!'
5.     }
6.   },
7.   methods: {
8.     reverseMessage() {
9.       this.message = this.message
10.        .split('')
11.        .reverse()
12.        .join('')
```

```

13.     }
14.   }
15. }
16.
17. Vue.createApp(EventHandling).mount('#event-handling')

```

注意在这个方法中，我们更新了应用的状态，但没有触碰 DOM—所有的 DOM 操作都由 Vue 来处理，你编写的代码只需要关注逻辑层面即可。

Vue 还提供了 `v-model` 指令，它能轻松实现表单输入和应用状态之间的双向绑定。

```

1. <div id="two-way-binding">
2.   <p>{{ message }}</p>
3.   <input v-model="message" />
4. </div>

```

```

1. const TwoWayBinding = {
2.   data() {
3.     return {
4.       message: 'Hello Vue!'
5.     }
6.   }
7. }
8.
9. Vue.createApp(TwoWayBinding).mount('#two-way-binding')

```

条件与循环

控制切换一个元素是否显示也相当简单：

```

1. <div id="conditional-rendering">
2.   <span v-if="seen">现在你看到我了</span>
3. </div>

```

```

1. const ConditionalRendering = {
2.   data() {
3.     return {
4.       seen: true
5.     }
6.   }

```

```
7. }
8.
9. Vue.createApp(ConditionalRendering).mount('#conditional-rendering')
```

这个例子演示了我们不仅可以把数据绑定到 DOM 文本或 attribute，还可以绑定到 DOM 的结构。此外，Vue 也提供一个强大的过渡效果系统，可以在 Vue 插入/更新/移除元素时自动应用[过渡效果](#)。

你可以在下面的沙盒中将 `seen` 从 `true` 更改为 `false`，以检查效果：

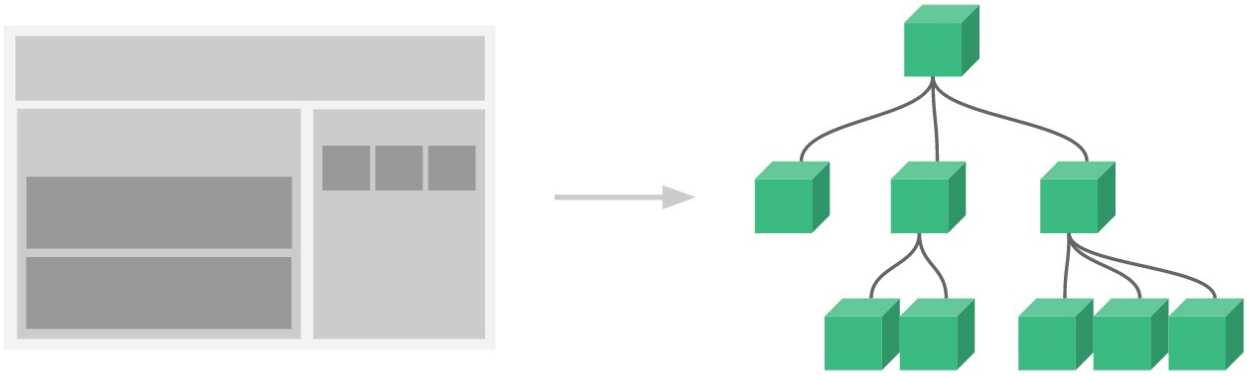
还有其它很多指令，每个都有特殊的功能。例如，`v-for` 指令可以绑定数组的数据来渲染一个项目列表：

```
1. <div id="list-rendering">
2.   <ol>
3.     <li v-for="todo in todos">
4.       {{ todo.text }}
5.     </li>
6.   </ol>
7. </div>
```

```
1. const ListRendering = {
2.   data() {
3.     return {
4.       todos: [
5.         { text: 'Learn JavaScript' },
6.         { text: 'Learn Vue' },
7.         { text: 'Build something awesome' }
8.       ]
9.     }
10.  }
11. }
12.
13. Vue.createApp(ListRendering).mount('#list-rendering')
```

组件化应用构建

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。仔细想想，几乎任意类型的应用界面都可以抽象为一个组件树：



在 Vue 中，组件本质上是一个具有预定义选项的实例。在 Vue 中注册组件很简单：如对象所做的那样创建一个组件对象，并将其定义在父级组件的 `components` 选项中： `App`

```

1. // 创建 Vue 应用
2. const app = Vue.createApp(...)
3.
4. // 定义名为 todo-item 的新组件
5. app.component('todo-item', {
6.   template: `<li>This is a todo</li>`
7. })
8.
9. // 挂载 Vue 应用
10. app.mount(...)
```

现在，你可以将其放到到另一个组件的模板中：

```

1. <ol>
2.   <!-- 创建一个 todo-item 组件实例 -->
3.   <todo-item></todo-item>
4. </ol>
```

但是这样会为每个待办项渲染同样的文本，这看起来并不炫酷。我们应该能将数据从父组件传入子组件才对。让我们来修改一下组件的定义，使之能够接受一个 `prop`：

```

1. app.component('todo-item', {
2.   props: ['todo'],
3.   template: `<li>{{ todo.text }}</li>`
4. })
```

现在，我们可以使用 `v-bind` 指令将待办项传到循环输出的每个组件中：

```

1. <div id="todo-list-app">
2.   <ol>
3.     <!--
4.       现在我们为每个 todo-item 提供 todo 对象
5.       todo 对象是变量，即其内容可以是动态的。
6.       我们也需要为每个组件提供一个“key”，稍后再
7.       作详细解释。
8.     -->
9.     <todo-item
10.      v-for="item in groceryList"
11.      v-bind:todo="item"
12.      v-bind:key="item.id"
13.    ></todo-item>
14.   </ol>
15. </div>

```

```

1. const TodoList = {
2.   data() {
3.     return {
4.       groceryList: [
5.         { id: 0, text: 'Vegetables' },
6.         { id: 1, text: 'Cheese' },
7.         { id: 2, text: 'Whatever else humans are supposed to eat' }
8.       ]
9.     }
10.  }
11. }
12.
13. const app = Vue.createApp(TodoList)
14.
15. app.component('todo-item', {
16.   props: ['todo'],
17.   template: `<li>{{ todo.text }}</li>`
18. })
19.
20. app.mount('#todo-list-app')

```

尽管这只是一个刻意设计的例子，但是我们已经设法将应用分割成了两个更小的单元。子单元通过 prop 接口与父单元进行了良好的解耦。我们现在可以进一步改进 `<todo-item>` 组件，提供更为

复杂的模板和逻辑，而不会影响到父应用。

在一个大型应用中，有必要将整个应用程序划分为多个组件，以使开发更易管理。在[后续教程](#)中我们将详述组件，不过这里有一个（假想的）例子，以展示使用了组件的应用模板是什么样的：

```
1. <div id="app">
2.   <app-nav></app-nav>
3.   <app-view>
4.     <app-sidebar></app-sidebar>
5.     <app-content></app-content>
6.   </app-view>
7. </div>
```

与自定义元素的关系

你可能已经注意到 Vue 组件非常类似于自定义元素——它是 [Web 组件规范](#) (opens new window) 的一部分，这是因为 Vue 的组件语法部分参考了该规范。例如 Vue 组件实现了 [Slot API](#) (opens new window) 与 `is` attribute。但是，还是有几个关键差别：

1. Web Components 规范已经完成并通过，但未被所有浏览器原生实现。目前 Safari 10.1+、Chrome 54+ 和 Firefox 63+ 原生支持 Web Components。相比之下，Vue 组件不需要任何 polyfill，并且在所有支持的浏览器（IE11 及更高版本）之下表现一致。必要时，Vue 组件也可以包装于原生自定义元素之内。
2. Vue 组件提供了纯自定义元素所不具备的一些重要功能，最突出的是跨组件数据流、自定义事件通信以及构建工具集成。

虽然 Vue 内部没有使用自定义元素，不过在应用使用自定义元素、或以自定义元素形式发布时，[依然有很好的互操作性](#) (opens new window)。Vue CLI 也支持将 Vue 组件构建成为原生的自定义元素。

准备好了吗？

我们刚才简单介绍了 Vue 核心最基本的功能——本教程的其余部分将更加详细地涵盖这些功能以及其它高阶功能，所以请务必读完整个教程！

应用 & 组件实例

创建一个应用实例

每个 Vue 应用都是通过用 `createApp` 函数创建一个新的应用实例开始的：

```
1. const app = Vue.createApp({ /* 选项 */ })
```

该应用实例是用来在应用中注册“全局”组件的。我们将在后面的指南中详细讨论，简单的例子：

```
1. const app = Vue.createApp({})
2. app.component('SearchInput', SearchInputComponent)
3. app.directive('focus', FocusDirective)
4. app.use(LocalePlugin)
```

应用实例暴露的大多数方法都会返回该同一实例，允许链式：

```
1. Vue.createApp({})
2.   .component('SearchInput', SearchInputComponent)
3.   .directive('focus', FocusDirective)
4.   .use(LocalePlugin)
```

你可以在 [API reference](#) 中浏览完整的应用 API。

根组件

传递给 `createApp` 的选项用于配置根组件。当我们挂载应用时，该组件被用作渲染的起点。

一个应用需要被挂载到一个 DOM 元素中。例如，如果我们想把一个 Vue 应用挂载到 `<div id="app"></div>`，我们应该传递 `#app`：

```
1. const RootComponent = { /* 选项 */ }
2. const app = Vue.createApp(RootComponent)
3. const vm = app.mount('#app')
```

与大多数应用方法不同的是，`mount` 不返回应用本身。相反，它返回的是根组件实例。

虽然没有完全遵循 [MVVM 模型](#) (opens new window)，但是 Vue 的设计也受到了它的启发。因

此在文档中经常会使用 `vm` (ViewMode1 的缩写) 这个变量名表示组件实例。

尽管本页面上的所有示例都只需要一个单一的组件就可以，但是大多数的真实应用都是被组织成一个嵌套的、可重用的组件树。举个例子，一个 `todo` 应用组件树可能是这样的：

```
1. Root Component
2.   └─ TodoList
3.     └─ TodoItem
4.       └─ DeleteTodoButton
5.       └─ EditTodoButton
6.     └─ TodoListFooter
7.       └─ ClearTodosButton
8.       └─ TodoListStatistics
```

每个组件将有自己的组件实例 `vm`。对于一些组件，如 `TodoItem`，在任何时候都可能有多实例渲染。这个应用中的所有组件实例将共享同一个应用实例。

我们会在稍后的[组件系统](#)章节具体展开。不过现在，你只需要明白根组件与其他组件没什么不同，配置选项是一样的，所对应的组件实例行为也是一样的。

组件实例 property

在前面的指南中，我们认识了 `data` property。在 `data` 中定义的 property 是通过组件实例暴露的：

```
1. const app = Vue.createApp({
2.   data() {
3.     return { count: 4 }
4.   }
5. })
6.
7. const vm = app.mount('#app')
8.
9. console.log(vm.count) // => 4
```

还有各种其他的组件选项，可以将用户定义的 property 添加到组件实例中，例如 `methods`，`props`，`computed`，`inject` 和 `setup`。我们将在后面的指南中深入讨论它们。组件实例的所有 property，无论如何定义，都可以在组件的模板中访问。

Vue 还通过组件实例暴露了一些内置 property，如 `$attrs` 和 `$emit`。这些 property 都有一个 `$` 前缀，以避免与用户定义的 property 名冲突。

生命周期钩子

每个组件在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，这给了用户在不同阶段添加自己的代码的机会。

比如 `created` 钩子可以用来在一个实例被创建之后执行代码：

```
1. Vue.createApp({
2.   data() {
3.     return { count: 1 }
4.   },
5.   created() {
6.     // `this` 指向 vm 实例
7.     console.log('count is: ' + this.count) // => "count is: 1"
8.   }
9. })
```

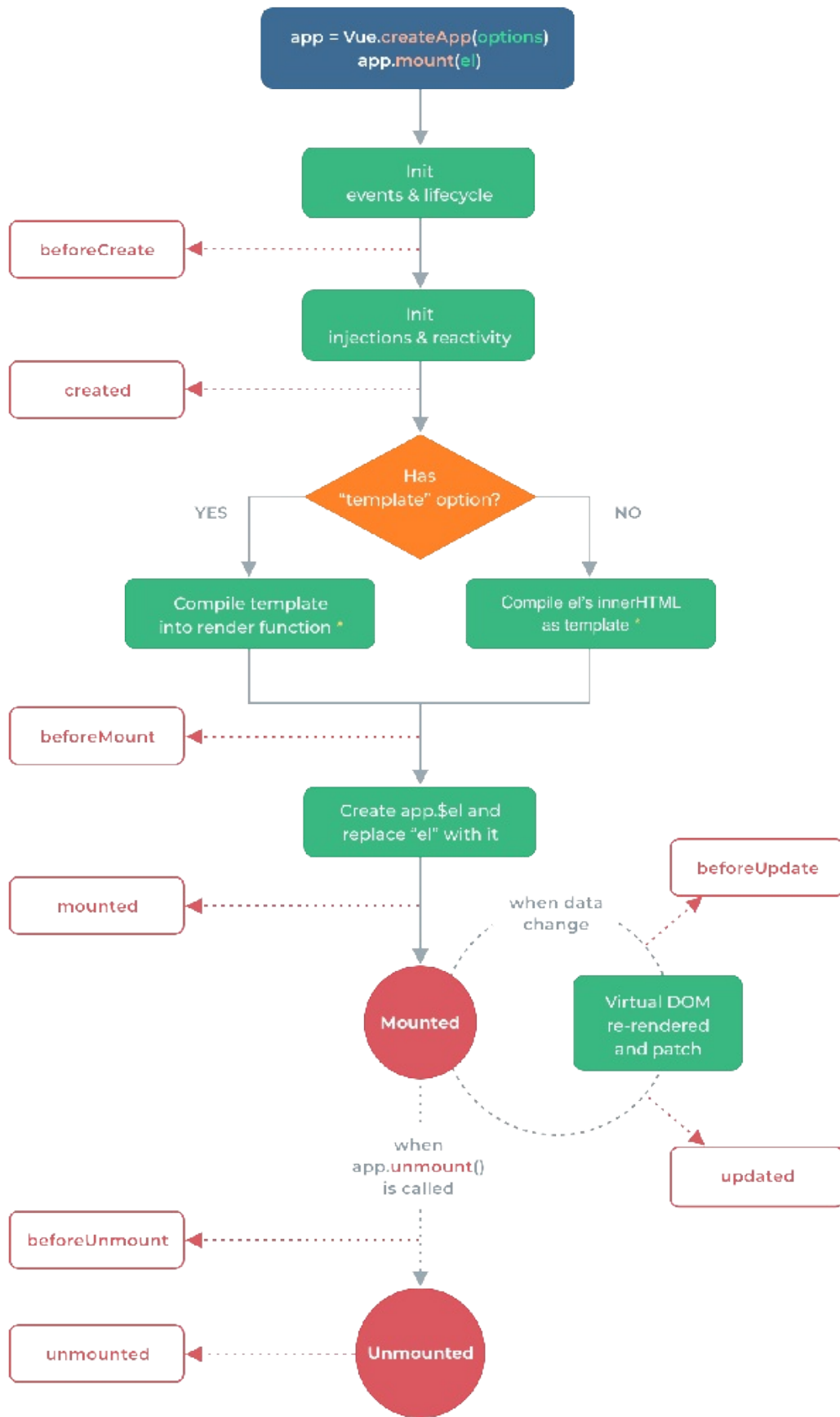
也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 `mounted`、`updated` 和 `unmounted`。生命周期钩子的 `this` 上下文指向调用它的当前活动实例。

TIP

不要在选项 `property` 或回调上使用箭头函数)，比如 `created: () => console.log(this.a)` 或 `vm.$watch('a', newValue => this.myMethod())`。因为箭头函数并没有 `this`，`this` 会作为变量一直向上级词法作用域查找，直至找到为止，经常导致 `Uncaught TypeError: Cannot read property of undefined` 或 `Uncaught TypeError: this.myMethod is not a function` 之类的错误。

生命周期图示

下图展示了实例的生命周期。你不需要立马弄明白所有的东西，不过随着你的不断学习和使用，它的参考价值会越来越高。



* Template compilation is performed ahead-of-time if using a build step, e.g., with single-file components.

模板语法

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层组件实例的数据。所有 Vue.js 的模板都是合法的 HTML，所以能被遵循规范的浏览器和 HTML 解析器解析。

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

如果你熟悉虚拟 DOM 并且偏爱 JavaScript 的原始力量，你也可以不用模板，[直接写渲染 \(render\) 函数](#)，使用可选的 JSX 语法。

插值

文本

数据绑定最常见的形式就是使用“Mustache”语法（双大括号）的文本插值：

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

Mustache 标签将会被替代为对应组件实例中 `msg` property 的值。无论何时，绑定的组件实例上 `msg` property 发生了改变，插值处的内容都会更新。

通过使用 `v-once` 指令，你也能执行一次性地插值，当数据改变时，插值处的内容不会更新。但请留心这会影响到该节点上的其它数据绑定：

```
1. <span v-once>这个将不会改变: {{ msg }}</span>
```

原始 HTML

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 `v-html` 指令：

```
1. <p>Using mustaches: {{ rawHtml }}</p>
2. <p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

这个 `span` 的内容将会被替换成为 property 值 `rawHtml`，直接作为 HTML——会忽略解析 property 值中的数据绑定。注意，你不能使用 `v-html` 来复合局部模板，因为 Vue 不是基于字符串的模板引擎。反之，对于用户界面 (UI)，组件更适合作为可重用和可组合的基本单位。

TIP

在你的站点上动态渲染任意的 HTML 是非常危险的，因为它很容易导致 [XSS 攻击](#) (opens new window)。请只对可信内容使用 HTML 插值，绝不要将用户提供的内容作为插值。

Attribute

Mustache 语法不能在 HTML attribute 中使用，然而，可以使用 `v-bind` 指令：

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

对于布尔 attribute（它们只要存在就意味着值为 `true`），`v-bind` 工作起来略有不同，在这个例子中：

```
1. <button v-bind:disabled="isButtonDisabled">按钮</button>
```

如果 `isButtonDisabled` 的值是 `null` 或 `undefined`，则 `disabled` attribute 甚至不会被包含在渲染出来的 `<button>` 元素中。

使用 JavaScript 表达式

迄今为止，在我们的模板中，我们一直都只绑定简单的 property 键值。但实际上，对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持。

```
1. {{ number + 1 }}
2. {{ ok ? 'YES' : 'NO' }}
3. {{ message.split('').reverse().join('') }}
4.
5. <div v-bind:id="'list-' + id"></div>
```

这些表达式会在当前活动实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。

```
1. <!-- 这是语句，不是表达式：-->
2. {{ var a = 1 }}
3.
4. <!-- 流控制也不会生效，请使用三元表达式 -->
5. {{ if (ok) { return message } }}
```

指令

指令 (Directives) 是带有 `v-` 前缀的特殊 attribute。指令 attribute 的值预期是单个

JavaScript 表达式 (`v-for` 和 `v-on` 是例外情况, 稍后我们再讨论)。指令的职责是, 当表达式的值改变时, 将其产生的连带影响, 响应式地作用于 DOM。回顾我们在介绍中看到的例子:

```
1. <p v-if="seen">现在你看到我了</p>
```

这里, `v-if` 指令将根据表达式 `seen` 的值的真假来插入/移除 `<p>` 元素。

参数

一些指令能够接收一个“参数”, 在指令名称之后以冒号表示。例如, `v-bind` 指令可以用于响应式地更新 HTML attribute:

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

在这里 `href` 是参数, 告知 `v-bind` 指令将该元素的 `href` attribute 与表达式 `url` 的值绑定。

另一个例子是 `v-on` 指令, 它用于监听 DOM 事件:

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

在这里参数是监听的事件名。我们也会更详细地讨论事件处理。

动态参数

也可以在指令参数中使用 JavaScript 表达式, 方法是用方括号括起来:

```
1. <!--
2. 注意, 参数表达式的写法存在一些约束, 如之后的“对动态参数表达式的约束”章节所述。
3. -->
4. <a v-bind:[attributeName]="url"> ... </a>
```

这里的 `attributeName` 会被作为一个 JavaScript 表达式进行动态求值, 求得的值将会作为最终的参数来使用。例如, 如果你的组件实例有一个 data property `attributeName`, 其值为 `"href"`, 那么这个绑定将等价于 `v-bind:href`。

同样地, 你可以使用动态参数为一个动态的事件名绑定处理函数:

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

在这个示例中, 当 `eventName` 的值为 `"focus"` 时, `v-on:[eventName]` 将等价于 `v-`

`on: focus`

修饰符

修饰符 (modifier) 是以半角句号 `.` 指明的特殊后缀, 用于指出一个指令应该以特殊方式绑定。

例如, `.prevent` 修饰符告诉 `v-on` 指令对于触发的事件调用

```
event.preventDefault() :
```

```
1. <form v-on:submit.prevent="onSubmit">...</form>
```

在接下来对 `v-on` 和 `v-for` 等功能的探索中, 你会看到修饰符的其它例子。

缩写

`v-` 前缀作为一种视觉提示, 用来识别模板中 Vue 特定的 attribute。当你在使用 Vue.js 为现有标签添加动态行为 (dynamic behavior) 时, `v-` 前缀很有帮助, 然而, 对于一些频繁用到的指令来说, 就会感到使用繁琐。同时, 在构建由 Vue 管理所有模板的单页面应用程序 (SPA -

single page application) [↗] (opens new window) 时, `v-` 前缀也变得没那么重要了。

因此, Vue 为 `v-bind` 和 `v-on` 这两个最常用的指令, 提供了特定简写:

`v-bind` 缩写

```
1. <!-- 完整语法 -->
2. <a v-bind:href="url"> ... </a>
3.
4. <!-- 缩写 -->
5. <a :href="url"> ... </a>
6.
7. <!-- 动态参数的缩写 -->
8. <a :[key]="url"> ... </a>
```

`v-on` 缩写

```
1. <!-- 完整语法 -->
2. <a v-on:click="doSomething"> ... </a>
3.
4. <!-- 缩写 -->
5. <a @click="doSomething"> ... </a>
6.
7. <!-- 动态参数的缩写 (2.6.0+) -->
```

```
8. <a @[event]="doSomething"> ... </a>
```

它们看起来可能与普通的 HTML 略有不同，但 `:` 与 `@` 对于 attribute 名来说都是合法字符，在所有支持 Vue 的浏览器都能被正确地解析。而且，它们不会出现在最终渲染的标记中。缩写语法是完全可选的，但随着你更深入地了解它们的作用，你会庆幸拥有它们。

从下一页开始，我们将在示例中使用缩写，因为这是 Vue 开发者最常用的用法。

注意事项

对动态参数值约定

动态参数预期会求出一个字符串，异常情况下值为 `null`。这个特殊的 `null` 值可以被显性地用于移除绑定。任何其它非字符串类型的值都将会触发一个警告。

对动态参数表达式约定

动态参数表达式有一些语法约束，因为某些字符，如空格和引号，放在 HTML attribute 名里是无效的。例如：

```
1. <!-- 这会触发一个编译警告 -->
2. <a v-bind:['foo' + bar]="value"> ... </a>
```

变通的办法是使用没有空格或引号的表达式，或用 [计算属性](#) 替代这种复杂表达式。

在 DOM 中使用模板时（直接在一个 HTML 文件里撰写模板），还需要避免使用大写字符来命名键名，因为浏览器会把 attribute 名全部强制转为小写：

```
1. <!--
2. 在 DOM 中使用模板时这段代码会被转换为 `v-bind:[someattr]`。
3. 除非在实例中有一个名为“someattr”的 property，否则代码不会工作。
4. -->
5. <a v-bind:[someAttr]="value"> ... </a>
```

JavaScript 表达式

模板表达式都被放在沙盒中，只能访问 [全局变量的一个白名单](#) [↗] (opens new window)，如 `Math` 和 `Date`。你不应该在模板表达式中试图访问用户定义的全局变量。

Data Properties and Methods

Data Properties

The `data` option for a component is a function. Vue calls this function as part of creating a new component instance. It should return an object, which Vue will then wrap in its reactivity system and store on the component instance as `$data`. For convenience, any top-level properties of that object are also exposed directly via the component instance:

```
1. const app = Vue.createApp({
2.   data() {
3.     return { count: 4 }
4.   }
5. })
6.
7. const vm = app.mount('#app')
8.
9. console.log(vm.$data.count) // => 4
10. console.log(vm.count)      // => 4
11.
12. // Assigning a value to vm.count will also update $data.count
13. vm.count = 5
14. console.log(vm.$data.count) // => 5
15.
16. // ... and vice-versa
17. vm.$data.count = 6
18. console.log(vm.count) // => 6
```

These instance properties are only added when the instance is first created, so you need to ensure they are all present in the object returned by the `data` function. Where necessary, use `null`, `undefined` or some other placeholder value for properties where the desired value isn't yet available.

It is possible to add a new property directly to the component instance without including it in `data`. However, because this property isn't backed by the reactive `$data` object, it won't automatically be tracked by [Vue's reactivity system](#).

Vue uses a `$` prefix when exposing its own built-in APIs via the component instance. It also reserves the prefix `_` for internal properties. You should avoid using names for top-level `data` properties that start with either of these characters.

Methods

To add methods to a component instance we use the `methods` option. This should be an object containing the desired methods:

```
1. const app = Vue.createApp({
2.   data() {
3.     return { count: 4 }
4.   },
5.   methods: {
6.     increment() {
7.       // `this` will refer to the component instance
8.       this.count++
9.     }
10.  }
11. })
12.
13. const vm = app.mount('#app')
14.
15. console.log(vm.count) // => 4
16.
17. vm.increment()
18.
19. console.log(vm.count) // => 5
```

Vue automatically binds the `this` value for `methods` so that it always refers to the component instance. This ensures that a method retains the correct `this` value if it's used as an event listener or callback. You should avoid using arrow functions when defining `methods`, as that prevents Vue from binding the appropriate `this` value.

Just like all other properties of the component instance, the `methods` are accessible from within the component's template. Inside a template they are most commonly used as event listeners:

```
1. <button @click="increment">Up vote</button>
```

In the example above, the method `increment` will be called when the `<button>` is clicked.

It is also possible to call a method directly from a template. As we'll see shortly, it's usually better to use a `computed property` instead. However, using a method can be useful in scenarios where computed properties aren't a viable option. You can call a method anywhere that a template supports JavaScript expressions:

```
1. <span :title="toTitleDate(date)">
2.   {{ formatDate(date) }}
3. </span>
```

If the methods `toTitleDate` or `formatDate` access any reactive data then it will be tracked as a rendering dependency, just as if it had been used in the template directly.

Methods called from a template should not have any side effects, such as changing data or triggering asynchronous processes. If you find yourself tempted to do that you should probably use a `lifecycle hook` instead.

Debouncing and Throttling

Vue doesn't include built-in support for debouncing or throttling but it can be implemented using libraries such as [Lodash](#) (opens new window).

In cases where a component is only used once, the debouncing can be applied directly within `methods` :

```
1. <script src="https://unpkg.com/lodash@4.17.20/lodash.min.js"></script>
2. <script>
3.   Vue.createApp({
4.     methods: {
5.       // Debouncing with Lodash
6.       click: _.debounce(function() {
7.         // ... respond to click ...
8.       }, 500)
9.     }
10.  }).mount('#app')
11. </script>
```

However, this approach is potentially problematic for components that are reused because they'll all share the same debounced function. To keep the component instances independent from each other, we can add the debounced function in the `created` lifecycle hook:

```
1. app.component('save-button', {
2.   created() {
3.     // Debouncing with Lodash
4.     this.debouncedClick = _.debounce(this.click, 500)
5.   },
6.   unmounted() {
7.     // Cancel the timer when the component is removed
8.     this.debouncedClick.cancel()
9.   },
10.  methods: {
11.    click() {
12.      // ... respond to click ...
13.    }
14.  },
15.  template: `
16.    <button @click="debouncedClick">
17.      Save
18.    </button>
19.  `
20. })
```


计算属性和侦听器

计算属性

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。在模板中放入太多的逻辑会让模板过重且难以维护。例如，有一个嵌套数组对象：

```
1. Vue.createApp({
2.   data() {
3.     return {
4.       author: {
5.         name: 'John Doe',
6.         books: [
7.           'Vue 2 - Advanced Guide',
8.           'Vue 3 - Basic Guide',
9.           'Vue 4 - The Mystery'
10.        ]
11.      }
12.    }
13.  })
```

我们想根据 `author` 是否已经有一些书来显示不同的消息

```
1. <div id="computed-basics">
2.   <p>Has published books:</p>
3.   <span>{{ author.books.length > 0 ? 'Yes' : 'No' }}</span>
4. </div>
```

此时，模板不再是简单的和声明性的。你必须先看一下它，然后才能意识到它执行的计算取决于 `author.books`。如果要在模板中多次包含此计算，则问题会变得更糟。

所以，对于任何包含响应式数据的复杂逻辑，你都应该使用计算属性。

基本例子

```
1. <div id="computed-basics">
2.   <p>Has published books:</p>
3.   <span>{{ publishedBooksMessage }}</span>
```

```
4. </div>
```

```
1. Vue.createApp({
2.   data() {
3.     return {
4.       author: {
5.         name: 'John Doe',
6.         books: [
7.           'Vue 2 - Advanced Guide',
8.           'Vue 3 - Basic Guide',
9.           'Vue 4 - The Mystery'
10.        ]
11.      }
12.    }
13.  },
14.  computed: {
15.    // 计算属性的 getter
16.    publishedBooksMessage() {
17.      // `this` points to the vm instance
18.      return this.author.books.length > 0 ? 'Yes' : 'No'
19.    }
20.  }
21. }).mount('#computed-basics')
```

Result:

这里声明了一个计算属性 `publishedBooksMessage`。

尝试更改应用程序 `data` 中 `books` 数组的值，你将看到 `publishedBooksMessage` 如何相应地更改。

你可以像普通属性一样将数据绑定到模板中的计算属性。Vue 知道 `vm.publishedBookMessage` 依赖于 `vm.author.books`，因此当 `vm.author.books` 发生改变时，所有依赖 `vm.publishedBookMessage` 绑定也会更新。而且最妙的是我们已经声明的方式创建了这个依赖关系：计算属性的 `getter` 函数没有副作用，这使得更易于测试和理解。

计算属性缓存 vs 方法

你可能已经注意到我们可以通过在表达式中调用方法来达到同样的效果：

```
1. <p>{{ calculateBooksMessage() }}</p>
```

```

1. // 在组件中
2. methods: {
3.   calculateBooksMessage() {
4.     return this.author.books.length > 0 ? 'Yes' : 'No'
5.   }
6. }

```

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的结果确实是完全相同的。然而，不同的是计算属性是基于它们的反应依赖关系缓存的。计算属性只在相关响应式依赖发生改变时它们才会重新求值。这就意味着只要 `author.books` 还没有发生改变，多次访问 `publishedBookMessage` 计算属性会立即返回之前的计算结果，而不必再次执行函数。

这也同样意味着下面的计算属性将不再更新，因为 `Date.now()` 不是响应式依赖：

```

1. computed: {
2.   now() {
3.     return Date.now()
4.   }
5. }

```

相比之下，每当触发重新渲染时，调用方法将总会再次执行函数。

我们为什么需要缓存？假设我们有一个性能开销比较大的计算属性 `list`，它需要遍历一个巨大的数组并做大量的计算。然后我们可能有其他的计算属性依赖于 `list`。如果没有缓存，我们将不可避免的多次执行 `list` 的 `getter`！如果你不希望有缓存，请用 `method` 来替代。

计算属性的 Setter

计算属性默认只有 `getter`，不过在需要时你也可以提供一个 `setter`：

```

1. // ...
2. computed: {
3.   fullName: {
4.     // getter
5.     get() {
6.       return this.firstName + ' ' + this.lastName
7.     },
8.     // setter
9.     set(newValue) {
10.      const names = newValue.split(' ')
11.      this.firstName = names[0]
12.      this.lastName = names[names.length - 1]

```

```

13.     }
14.   }
15. }
16. // ...

```

现在再运行 `vm.fullName = 'John Doe'` 时, setter 会被调用, `vm.firstName` 和 `vm.lastName` 也会相应地被更新。

侦听器

虽然计算属性在大多数情况下更合适,但有时也需要一个自定义的侦听器。这就是为什么 Vue 通过 `watch` 选项提供了一个更通用的方法,来响应数据的变化。当需要在数据变化时执行异步或开销较大的操作时,这个方式是最有用的。

例如:

```

1. <div id="watch-example">
2.   <p>
3.     Ask a yes/no question:
4.     <input v-model="question" />
5.   </p>
6.   <p>{{ answer }}</p>
7. </div>

```

```

1. <!-- 因为 AJAX 库和通用工具的生态已经相当丰富,Vue 核心代码没有重复 -->
2. <!-- 提供这些功能以保持精简。这也可以让你自由选择自己更熟悉的工具。 -->
3. <script src="https://cdn.jsdelivr.net/npm/axios@0.12.0/dist/axios.min.js">
4. </script>
5. <script>
6.   const watchExampleVM = Vue.createApp({
7.     data() {
8.       return {
9.         question: '',
10.        answer: 'Questions usually contain a question mark. ;-)'
11.      }
12.    },
13.    watch: {
14.      // whenever question changes, this function will run
15.      question(newQuestion, oldQuestion) {
16.        if (newQuestion.indexOf('?') > -1) {

```

```

17.     }
18.     }
19.   },
20.   methods: {
21.     getAnswer() {
22.       this.answer = 'Thinking...'
23.       axios
24.         .get('https://yesno.wtf/api')
25.         .then(response => {
26.           this.answer = response.data.answer
27.         })
28.         .catch(error => {
29.           this.answer = 'Error! Could not reach the API. ' + error
30.         })
31.     }
32.   }
33. }).mount('#watch-example')
34. </script>

```

结果：

在这个示例中，使用 `watch` 选项允许我们执行异步操作（访问一个 API），限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

除了 `watch` 选项之外，你还可以使用命令式的 `vm.$watch API`。

计算属性 vs 侦听器

Vue 提供了一种更通用的方式来观察和响应当前活动的实例上的数据变动：侦听属性。当你有一些数据需要随着其它数据变动而变动时，你很容易滥用 `watch` ——特别是如果你之前使用过 AngularJS。然而，通常更好的做法是使用计算属性而不是命令式的 `watch` 回调。细想一下这个例子：

```
1. <div id="demo">{{ fullName }}</div>
```

```

1. const vm = Vue.createApp({
2.   data() {
3.     return {
4.       firstName: 'Foo',
5.       lastName: 'Bar',
6.       fullName: 'Foo Bar'
7.     }
8.   },

```

```
9.   watch: {
10.     firstName(val) {
11.       this.fullName = val + ' ' + this.lastName
12.     },
13.     lastName(val) {
14.       this.fullName = this.firstName + ' ' + val
15.     }
16.   }
17. }).mount('#demo')
```

上面代码是命令式且重复的。将它与计算属性的版本进行比较：

```
1.  const vm = Vue.createApp({
2.    data() {
3.      return {
4.        firstName: 'Foo',
5.        lastName: 'Bar'
6.      }
7.    },
8.    computed: {
9.      fullName() {
10.        return this.firstName + ' ' + this.lastName
11.      }
12.    }
13.  }).mount('#demo')
```

好得多了，不是吗？

Class 与 Style 绑定

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是 attribute，所以我们可以用 `v-bind` 处理它们：只需要通过表达式计算出字符串结果即可。不过，字符串拼接麻烦且易错。因此，在将 `v-bind` 用于 `class` 和 `style` 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，还可以是对象或数组。

绑定 HTML Class

对象语法

我们可以传给 `:class` (`v-bind:class` 的简写) 一个对象，以动态地切换 class：

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

上面的语法表示 `active` 这个 class 存在与否将取决于数据 property `isActive` 的 truthiness [↗](#) (opens new window)。

你可以在对象中传入更多字段来动态切换多个 class。此外，`:class` 指令也可以与普通的 `class` attribute 共存。当有如下模板：

```
1. <div
2.   class="static"
3.   :class="{ active: isActive, 'text-danger': hasError }"
4. ></div>
```

和如下 data：

```
1. data() {
2.   return {
3.     isActive: true,
4.     hasError: false
5.   }
6. }
```

渲染的结果为：

```
1. <div class="static active"></div>
```

当 `isActive` 或者 `hasError` 变化时, `class` 列表将相应地更新。例如, 如果 `hasError` 的值为 `true`, `class` 列表将变为 `"static active text-danger"`。

绑定的数据对象不必内联定义在模板里:

```
1. <div :class="classObject"></div>
```

```
1. data() {
2.   return {
3.     classObject: {
4.       active: true,
5.       'text-danger': false
6.     }
7.   }
8. }
```

渲染的结果和上面一样。我们也可以在这里绑定一个返回对象的[计算属性](#)。这是一个常用且强大的模式:

```
1. <div :class="classObject"></div>
```

```
1. data() {
2.   return {
3.     isActive: true,
4.     error: null
5.   }
6. },
7. computed: {
8.   classObject() {
9.     return {
10.      active: this.isActive && !this.error,
11.      'text-danger': this.error && this.error.type === 'fatal'
12.    }
13.   }
14. }
```

数组语法

我们可以把一个数组传给 `:class`, 以应用一个 `class` 列表:

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



```
1. data() {
2.   return {
3.     activeClass: 'active',
4.     errorClass: 'text-danger'
5.   }
6. }
```

渲染的结果为：

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

如果你想根据条件切换列表中的 class，可以使用三元表达式：

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

这样写将始终添加 `errorClass`，但是只有在 `isActive` 为 `truthy[1]` 时才添加 `activeClass`。

不过，当有多个条件 class 时这样写有些繁琐。所以在数组语法中也可以使用对象语法：

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

在组件上使用

这个章节假设你已经对 `Vue` 组件有一定的了解。当然你也可以先跳过这里，稍后再回过头来看。

例如，如果你声明了这个组件：

```
1. const app = Vue.createApp({})
2.
3. app.component('my-component', {
4.   template: `<p class="foo bar">Hi!</p>`
5. })
```

然后在使用它的时候添加一些 class：

```
1. <div id="app">
2.   <my-component class="baz boo"></my-component>
3. </div>
```

HTML 将被渲染为：

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

对于带数据绑定 class 也同样适用：

```
1. <my-component :class="{ active: isActive }"></my-component>
```

当 isActive 为 truthy[1] 时，HTML 将被渲染成为：

```
1. <p class="foo bar active">Hi</p>
```

如果你的组件有多个根元素，你需要定义哪些部分将接收这个类。可以使用 `$attrs` 组件属性执行此操作：

```
1. <div id="app">
2.   <my-component class="baz"></my-component>
3. </div>
```

```
1. const app = Vue.createApp({})
2.
3. app.component('my-component', {
4.   template: `
5.     <p :class="$attrs.class">Hi!</p>
6.     <span>This is a child component</span>
7.   `
8. })
```

你可以在[非 prop Attribute](#) 小节了解更多关于组件属性继承的信息。

绑定内联样式

对象语法

`:style` 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS property 名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case, 记得用引号括起来) 来命名：

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

```

1. data() {
2.   return {
3.     activeColor: 'red',
4.     fontSize: 30
5.   }
6. }

```

直接绑定到一个样式对象通常更好，这会让模板更清晰：

```
1. <div :style="styleObject"></div>
```

```

1. data() {
2.   return {
3.     styleObject: {
4.       color: 'red',
5.       fontSize: '13px'
6.     }
7.   }
8. }

```

同样的，对象语法常常结合返回对象的计算属性使用。

数组语法

`:style` 的数组语法可以将多个样式对象应用到同一个元素上：

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

自动添加前缀

在 `:style` 中使用需要（浏览器引擎前缀）[vendor prefixes](#) (opens new window) 的 CSS property 时，如 `transform`，Vue 将自动侦测并添加相应的前缀。

多重值

可以为 `style` 绑定中的 property 提供一个包含多个值的数组，常用于提供多个带前缀的值，例如：

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

这样写只会渲染数组中最后一个被浏览器支持的值。在本例中，如果浏览器支持不带浏览器前缀的 flexbox，那么就只会渲染 `display: flex`。

译者注 [1] `truthy` 不是 `true`，详见 [MDN](#)[↗] (`opens new window`) 的解释。

条件渲染

v-if

`v-if` 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回 `truthy` 值的时候被渲染。

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

也可以用 `v-else` 添加一个“else 块”：

```
1. <h1 v-if="awesome">Vue is awesome!</h1>
2. <h1 v-else>Oh no 😞</h1>
```

在 `<template>` 元素上使用 `v-if` 条件渲染分组

因为 `v-if` 是一个指令，所以必须将它添加到一个元素上。但是如果切换多个元素呢？此时可以把一个 `<template>` 元素当做不可见的包裹元素，并在上面使用 `v-if`。最终的渲染结果将不包含 `<template>` 元素。

```
1. <template v-if="ok">
2.   <h1>Title</h1>
3.   <p>Paragraph 1</p>
4.   <p>Paragraph 2</p>
5. </template>
```

v-else

你可以使用 `v-else` 指令来表示 `v-if` 的“else 块”：

```
1. <div v-if="Math.random() > 0.5">
2.   Now you see me
3. </div>
4. <div v-else>
5.   Now you don't
6. </div>
```

`v-else` 元素必须紧跟在带 `v-if` 或者 `v-else-if` 的元素的后面，否则它将不会被识别。

v-else-if

`v-else-if`，顾名思义，充当 `v-if` 的“else-if 块”，可以连续使用：

```
1. <div v-if="type === 'A'">
2.   A
3. </div>
4. <div v-else-if="type === 'B'">
5.   B
6. </div>
7. <div v-else-if="type === 'C'">
8.   C
9. </div>
10. <div v-else>
11.   Not A/B/C
12. </div>
```

类似于 `v-else`，`v-else-if` 也必须紧跟在带 `v-if` 或者 `v-else-if` 的元素之后。

`v-show`

另一个用于根据条件展示元素的选项是 `v-show` 指令。用法大致一样：

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

不同的是带有 `v-show` 的元素始终会被渲染并保留在 DOM 中。`v-show` 只是简单地切换元素的 CSS property `display`。

注意，`v-show` 不支持 `<template>` 元素，也不支持 `v-else`。

`v-if` VS `v-show`

`v-if` 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

`v-if` 也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

相比之下，`v-show` 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

一般来说，`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，如果需要非常频繁地切换，则使用 `v-show` 较好；如果在运行时条件很少改变，则使用 `v-if` 较好。

`v-if` 与 `v-for` 一起使用

提示

不推荐同时使用 `v-if` 和 `v-for`。请查阅[风格指南](#)以获取更多信息。

当 `v-if` 与 `v-for` 一起使用时，`v-if` 具有比 `v-for` 更高的优先级。请查阅[列表渲染指南](#)以获取详细信息。

列表渲染

用 `v-for` 把一个数组对应为一组元素

我们可以用 `v-for` 指令基于一个数组来渲染一个列表。`v-for` 指令需要使用 `item in items` 形式的特殊语法，其中 `items` 是源数据数组，而 `item` 则是被迭代的数组元素的别名。

```
1. <ul id="array-rendering">
2.   <li v-for="item in items">
3.     {{ item.message }}
4.   </li>
5. </ul>
```

```
1. Vue.createApp({
2.   data() {
3.     return {
4.       items: [{ message: 'Foo' }, { message: 'Bar' }]
5.     }
6.   }
7. }).mount('#array-rendering')
```

结果：

See the Pen [v-for with Array](#) by Vue (@Vue) on [CodePen](#).

在 `v-for` 块中，我们可以访问所有父作用域的 `property`。`v-for` 还支持一个可选的第二个参数，即当前项的索引。

```
1. <ul id="array-with-index">
2.   <li v-for="(item, index) in items">
3.     {{ parentMessage }} - {{ index }} - {{ item.message }}
4.   </li>
5. </ul>
```

```
1. Vue.createApp({
2.   data() {
3.     return {
4.       parentMessage: 'Parent',
5.       items: [{ message: 'Foo' }, { message: 'Bar' }]

```



```
6.     }
7.     }
8. }).mount('#array-with-index')
```

结果:

See the Pen [v-for with Array and index](#) by Vue (@Vue) on CodePen.

你也可以用 `of` 替代 `in` 作为分隔符, 因为它更接近 JavaScript 迭代器的语法:

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

在 `v-for` 里使用对象

你也可以用 `v-for` 来遍历一个对象的 property。

```
1. <ul id="v-for-object" class="demo">
2.   <li v-for="value in myObject">
3.     {{ value }}
4.   </li>
5. </ul>
```

```
1. Vue.createApp({
2.   data() {
3.     return {
4.       myObject: {
5.         title: 'How to do lists in Vue',
6.         author: 'Jane Doe',
7.         publishedAt: '2016-04-10'
8.       }
9.     }
10.  }
11. }).mount('#v-for-object')
```

结果:

See the Pen [v-for with Object](#) by Vue (@Vue) on CodePen.

你也可以提供第二个的参数为 property 名称 (也就是键名 key):

```
1. <li v-for="(value, name) in myObject">
```

```
2.   {{ name }}: {{ value }}
3. </li>
```

See the Pen [v-for with Object and key](#) by Vue (@Vue) on [CodePen](#).

还可以用第三个参数作为索引:

```
1. <li v-for="(value, name, index) in myObject">
2.   {{ index }}. {{ name }}: {{ value }}
3. </li>
```

See the Pen [v-for with Object key and index](#) by Vue (@Vue) on [CodePen](#).

提示

在遍历对象时, 会按 `Object.keys()` 的结果遍历, 但是不能保证它在不同 JavaScript 引擎下的结果都一致。

维护状态

当 Vue 正在更新使用 `v-for` 渲染的元素列表时, 它默认使用“就地更新”的策略。如果数据项的顺序被改变, Vue 将不会移动 DOM 元素来匹配数据项的顺序, 而是就地更新每个元素, 并且确保它们在每个索引位置正确渲染。

这个默认的模式是高效的, 但是只适用于不依赖子组件状态或临时 DOM 状态 (例如: 表单输入值) 的列表渲染输出。

为了给 Vue 一个提示, 以便它能跟踪每个节点的身份, 从而重用和重新排序现有元素, 你需要为每项提供一个唯一 `key` attribute:

```
1. <div v-for="item in items" :key="item.id">
2.   <!-- content -->
3. </div>
```

建议尽可能在使用 `v-for` 时提供 `key` attribute, 除非遍历输出的 DOM 内容非常简单, 或者是刻意依赖默认行为以获取性能上的提升。

因为它是 Vue 识别节点的一个通用机制, `key` 并不仅与 `v-for` 特别关联。后面我们将在指南中看到, 它还具有其它用途。

提示

不要使用对象或数组之类的非基本类型值作为 `v-for` 的 `key`。请用字符串或数值类型的值。

更多 `key` attribute 的细节用法请移步至 `key` 的 API 文档。

数组更新检测

变更方法

Vue 将被侦听的数组的变更方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

你可以打开控制台，然后对前面例子的 `items` 数组尝试调用变更方法。比如

```
example1.items.push({ message: 'Baz' })
```

替换数组

变更方法，顾名思义，会变更调用了这些方法的原始数组。相比之下，也有非变更方法，例如 `filter()`、`concat()` 和 `slice()`。它们不会变更原始数组，而总是返回一个新数组。当使用非变更方法时，可以用新数组替换旧数组：

```
1. example1.items = example1.items.filter(item => item.message.match(/Foo/))
```

你可能认为这将导致 Vue 丢弃现有 DOM 并重新渲染整个列表。幸运的是，事实并非如此。Vue 为了使得 DOM 元素得到最大范围的重用而实现了一些智能的启发式方法，所以用一个含有相同元素的数组去替换原来的数组是非常高效的操作。

显示过滤/排序后的结果

有时，我们想要显示一个数组经过过滤或排序后的版本，而不实际变更或重置原始数据。在这种情况下，可以创建一个计算属性，来返回过滤或排序后的数组。

例如：

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

```

1. data() {
2.   return {
3.     numbers: [ 1, 2, 3, 4, 5 ]
4.   }
5. },
6. computed: {
7.   evenNumbers() {
8.     return this.numbers.filter(number => number % 2 === 0)
9.   }
10. }

```

在计算属性不适用的情况下（例如，在嵌套 `v-for` 循环中）你可以使用一个方法：

```

1. <ul v-for="numbers in sets">
2.   <li v-for="n in even(numbers)">{{ n }}</li>
3. </ul>

```

```

1. data() {
2.   return {
3.     sets: [[ 1, 2, 3, 4, 5 ], [6, 7, 8, 9, 10]]
4.   }
5. },
6. methods: {
7.   even(numbers) {
8.     return numbers.filter(number => number % 2 === 0)
9.   }
10. }

```

在 `v-for` 里使用值的范围

`v-for` 也可以接受整数。在这种情况下，它会把模板重复对应次数。

```

1. <div id="range" class="demo">
2.   <span v-for="n in 10">{{ n }} </span>
3. </div>

```

结果：

See the Pen [v-for with a range](#) by Vue (@Vue) on [CodePen](#).

在 `<template>` 中使用 `v-for`

类似于 `v-if`，你也可以利用带有 `v-for` 的 `<template>` 来循环渲染一段包含多个元素的内容。比如：

```
1. <ul>
2.   <template v-for="item in items">
3.     <li>{{ item.msg }}</li>
4.     <li class="divider" role="presentation"></li>
5.   </template>
6. </ul>
```

`v-for` 与 `v-if` 一同使用

TIP

注意我们不推荐在同一元素上使用 `v-if` 和 `v-for`。更多细节可查阅[风格指南](#)。

当它们处于同一节点，`v-if` 的优先级比 `v-for` 更高，这意味着 `v-if` 将没有权限访问 `v-for` 里的变量：

```
  <!-- This will throw an error because property "todo" is not defined on
1. instance. -->
2.
3. <li v-for="todo in todos" v-if="!todo.isComplete">
4.   {{ todo }}
5. </li>
```

可以把 `v-for` 移动到 `<template>` 标签中来修正：

```
1. <template v-for="todo in todos">
2.   <li v-if="!todo.isComplete">
3.     {{ todo }}
4.   </li>
5. </template>
```

在组件上使用 `v-for`

这部分内容假定你已经了解[组件](#)相关知识。你也完全可以先跳过它，以后再回来查看。

在自定义组件上，你可以像在任何普通元素上一样使用 `v-for`：

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

然而，任何数据都不会被自动传递到组件里，因为组件有自己独立的作用域。为了把迭代数据传递到组件里，我们要使用 props：

```
1. <my-component
2.   v-for="(item, index) in items"
3.   :item="item"
4.   :index="index"
5.   :key="item.id"
6. ></my-component>
```

不自动将 `item` 注入到组件里的原因是，这会使得组件与 `v-for` 的运作紧密耦合。明确组件数据的来源能够使组件在其他场合重复使用。

下面是一个简单的 todo 列表的完整例子：

```
1. <div id="todo-list-example">
2.   <form v-on:submit.prevent="addNewTodo">
3.     <label for="new-todo">Add a todo</label>
4.     <input
5.       v-model="newTodoText"
6.       id="new-todo"
7.       placeholder="E.g. Feed the cat"
8.     />
9.     <button>Add</button>
10.  </form>
11.  <ul>
12.    <todo-item
13.      v-for="(todo, index) in todos"
14.      :key="todo.id"
15.      :title="todo.title"
16.      @remove="todos.splice(index, 1)"
17.    ></todo-item>
18.  </ul>
19. </div>
```

```
1. const app = Vue.createApp({
2.   data() {
```

```
3.     return {
4.       newTodoText: '',
5.       todos: [
6.         {
7.           id: 1,
8.           title: 'Do the dishes'
9.         },
10.        {
11.          id: 2,
12.          title: 'Take out the trash'
13.        },
14.        {
15.          id: 3,
16.          title: 'Mow the lawn'
17.        }
18.      ],
19.      nextTodoId: 4
20.    }
21.  },
22.  methods: {
23.    addNewTodo() {
24.      this.todos.push({
25.        id: this.nextTodoId++,
26.        title: this.newTodoText
27.      })
28.      this.newTodoText = ''
29.    }
30.  }
31. })
32.
33. app.component('todo-item', {
34.   template: `
35.     <li>
36.       {{ title }}
37.       <button @click="$emit('remove')">Remove</button>
38.     </li>
39.   `,
40.   props: ['title']
41. })
42.
43. app.mount('#todo-list-example')
```

列表渲染

See the Pen [v-for with components](#) by Vue (@Vue) on [CodePen](#).

事件处理

了解如何处理事件, 一个免费的Vue School课程

监听事件

我们可以使用 `v-on` 指令 (通常缩写为 `@` 符号) 来监听 DOM 事件, 并在触发事件时执行一些 JavaScript。用法为 `v-on:click="methodName"` 或使用快捷方式

```
@click="methodName"
```

例如:

```
1. <div id="basic-event">
2.   <button @click="counter += 1">Add 1</button>
3.   <p>The button above has been clicked {{ counter }} times.</p>
4. </div>
```

```
1. Vue.createApp({
2.   data() {
3.     return {
4.       counter: 1
5.     }
6.   }
7. }).mount('#basic-event')
```

结果:

事件处理方法

然而许多事件处理逻辑会更为复杂, 所以直接把 JavaScript 代码写在 `v-on` 指令中是不可行的。因此 `v-on` 还可以接收一个需要调用的方法名称。

例如:

```
1. <div id="event-with-method">
2.   <!-- `greet` 在下面定义的方法名 -->
3.   <button @click="greet">Greet</button>
4. </div>
```

```

1. Vue.createApp({
2.   data() {
3.     return {
4.       name: 'Vue.js'
5.     }
6.   },
7.   methods: {
8.     greet(event) {
9.       // `this` 内部 `methods` 指向当前活动实例
10.      alert('Hello ' + this.name + '!')
11.      // `event` 是原生 DOM event
12.      if (event) {
13.        alert(event.target.tagName)
14.      }
15.    }
16.  }
17. }).mount('#event-with-method')

```

结果：

内联处理器中的方法

除了直接绑定到一个方法，也可以在内联 JavaScript 语句中调用方法：

```

1. <div id="inline-handler">
2.   <button @click="say('hi')">Say hi</button>
3.   <button @click="say('what')">Say what</button>
4. </div>

```

```

1. Vue.createApp({
2.   methods: {
3.     say(message) {
4.       alert(message)
5.     }
6.   }
7. }).mount('#inline-handler')

```

结果：

有时也需要在内联语句处理器中访问原始的 DOM 事件。可以用特殊变量 `$event` 把它传入方法：

```

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

```

```

1. // ...
2. methods: {
3.   warn(message, event) {
4.     // now we have access to the native event
5.     if (event) {
6.       event.preventDefault()
7.     }
8.     alert(message)
9.   }
10. }

```

多事件处理器

事件处理程序中可以有多个方法，这些方法由逗号运算符分隔：

```

1. <!-- 这两个 one() 和 two() 将执行按钮点击事件 -->
2. <button @click="one($event), two($event)">
3.   Submit
4. </button>

```

```

1. // ...
2. methods: {
3.   one(event) {
4.     // first handler logic...
5.   },
6.   two(event) {
7.     // second handler logic...
8.   }
9. }

```

事件修饰符

在事件处理程序中调用 `event.preventDefault()` 或 `event.stopPropagation()` 是非常常见的需求。尽管我们可以在方法中轻松实现这点，但更好的方式是：方法只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

为了解决这个问题，Vue.js 为 `v-on` 提供了事件修饰符。之前提过，修饰符是由点开头的指令后缀来表示的。

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`
- `.passive`

```

1. <!-- 阻止单击事件继续传播 -->
2. <a @click.stop="doThis"></a>
3.
4. <!-- 提交事件不再重载页面 -->
5. <form @submit.prevent="onSubmit"></form>
6.
7. <!-- 修饰符可以串联 -->
8. <a @click.stop.prevent="doThat"></a>
9.
10. <!-- 只有修饰符 -->
11. <form @submit.prevent></form>
12.
13. <!-- 添加事件监听器时使用事件捕获模式 -->
14. <!-- 即内部元素触发的事件先在此处理，然后才交由内部元素进行处理 -->
15. <div @click.capture="doThis">...</div>
16.
17. <!-- 只当在 event.target 是当前元素自身时触发处理函数 -->
18. <!-- 即事件不是从内部元素触发的 -->
19. <div @click.self="doThat">...</div>

```

TIP

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 `v-on:click.prevent.self` 会阻止所有的点击，而 `v-on:click.self.prevent` 只会阻止对元素自身的点击。

```

1. <!-- 点击事件将只会触发一次 -->
2. <a @click.once="doThis"></a>

```

不像其它只能对原生的 DOM 事件起作用的修饰符，`.once` 修饰符还能被用到自定义的**组件事件**上。如果你还没有阅读关于组件的文档，现在大可不必担心。

Vue 还对应 `addEventListener` 中的 `passive` 选项 [\(opens new window\)](#) 提供了 `.passive` 修饰符。

1. `<!-- 滚动事件的默认行为（即滚动行为）将会立即触发 -->`
2. `<!-- 而不会等待 `onScroll` 完成 -->`
3. `<!-- 这其中包含 `event.preventDefault()` 的情况 -->`
4. `<div @scroll.passive="onScroll">...</div>`

这个 `.passive` 修饰符尤其能够提升移动端的性能。

TIP

不要把 `.passive` 和 `.prevent` 一起使用，因为 `.prevent` 将会被忽略，同时浏览器可能会向你展示一个警告。请记住，`.passive` 会告诉浏览器你不想阻止事件的默认行为。

按键修饰符

在监听键盘事件时，我们经常需要检查详细的按键。Vue 允许为 `v-on` 或者 `@` 在监听键盘事件时添加按键修饰符：

1. `<!-- 只有在 `key` 是 `Enter` 时调用 `vm.submit()` -->`
2. `<input @keyup.enter="submit" />`

你可以直接将 `KeyboardEvent.key` [\(opens new window\)](#) 暴露的任意有效按键名转换为 kebab-case 来作为修饰符。

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

在上述示例中，处理函数只会在 `$event.key` 等于 `'PageDown'` 时被调用。

按键别名

Vue 为最常用的键提供了别名：

- `.enter`
- `.tab`
- `.delete` (捕获“删除”和“退格”键)
- `.esc`
- `.space`
- `.up`

- `.down`
- `.left`
- `.right`

系统修饰键

可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。

- `.ctrl`
- `.alt`
- `.shift`
- `.meta`

提示

注意：在 Mac 系统键盘上，meta 对应 command 键 (⌘)。在 Windows 系统键盘 meta 对应 Windows 徽标键 (⊞)。在 Sun 操作系统键盘上，meta 对应实心宝石键 (◆)。在其他特定键盘上，尤其在 MIT 和 Lisp 机器的键盘、以及其后继产品，比如 Knight 键盘、space-cadet 键盘，meta 被标记为“META”。在 Symbolics 键盘上，meta 被标记为“META”或者“Meta”。

例如：

```
1. <!-- Alt + Enter -->
2. <input @keyup.alt.enter="clear" />
3.
4. <!-- Ctrl + Click -->
5. <div @click.ctrl="doSomething">Do something</div>
```

TIP

请注意修饰键与常规按键不同，在和 `keyup` 事件一起用时，事件触发时修饰键必须处于按下状态。换句话说，只有在按住 `ctrl` 的情况下释放其它按键，才能触发 `keyup.ctrl`。而单单释放 `ctrl` 也不会触发事件。

`.exact` 修饰符

`.exact` 修饰符允许你控制由精确的系统修饰符组合触发的事件。

```
1. <!-- 即使 Alt 或 Shift 被一同按下时也会触发 -->
2. <button @click.ctrl="onClick">A</button>
3.
4. <!-- 有且只有 Ctrl 被按下的时候才触发 -->
```

```
5. <button @click.ctrl.exact="onCtrlClick">A</button>
6.
7. <!-- 没有任何系统修饰符被按下的时候才触发 -->
8. <button @click.exact="onClick">A</button>
```

鼠标按钮修饰符

- `.left`
- `.right`
- `.middle`

这些修饰符会限制处理函数仅响应特定的鼠标按钮。

为什么在 HTML 中监听事件？

你可能注意到这种事件监听的方式违背了关注点分离 (separation of concern) 这个长期以来的优良传统。但不必担心，因为所有的 Vue.js 事件处理方法和表达式都严格绑定在当前视图的 ViewModel 上，它不会导致任何维护上的困难。实际上，使用 `v-on` 或 `@` 有几个好处：

1. 扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。
2. 因为你无须在 JavaScript 里手动绑定事件，你的 ViewModel 代码可以是非常纯粹的逻辑，和 DOM 完全解耦，更易于测试。
3. 当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何清理它们。

表单输入绑定

基础用法

你可以用 `v-model` 指令在表单 `<input>`、`<textarea>` 及 `<select>` 元素上创建双向数据绑定。它会根据控件类型自动选取正确的方法来更新元素。尽管有些神奇，但 `v-model` 本质上不过是语法糖。它负责监听用户的输入事件以更新数据，并对一些极端场景进行一些特殊处理。

提示

`v-model` 会忽略所有表单元素的 `value`、`checked`、`selected` attribute 的初始值而总是将当前活动实例的数据作为数据来源。你应该通过 JavaScript 在组件的 `data` 选项中声明初始值。

`v-model` 在内部为不同的输入元素使用不同的 property 并抛出不同的事件：

- text 和 textarea 元素使用 `value` property 和 `input` 事件；
- checkbox 和 radio 使用 `checked` property 和 `change` 事件；
- select 字段将 `value` 作为 prop 并将 `change` 作为事件。

提示

对于需要使用 [输入法](#) (opens new window) (如中文、日文、韩文等) 的语言，你会发现 `v-model` 不会在输入法组合文字过程中得到更新。如果你也想处理这个过程，请使用 `input` 事件。

文本 (Text)

1. `<input v-model="message" placeholder="edit me" />`
2. `<p>Message is: {{ message }}</p>`

多行文本 (textarea)

1. `Multiline message is:`
2. `<p style="white-space: pre-line;">{{ message }}</p>`
3. `
`
4. `<textarea v-model="message" placeholder="add multiple lines"></textarea>`

在文本区域插值不起作用，应该使用 `v-model` 来代替。


```

1. <!-- bad -->
2. <textarea>{{ text }}</textarea>
3.
4. <!-- good -->
5. <textarea v-model="text"></textarea>

```

复选框 (Checkbox)

单个复选框，绑定到布尔值：

```

1. <input type="checkbox" id="checkbox" v-model="checked" />
2. <label for="checkbox">{{ checked }}</label>

```

多个复选框，绑定到同一个数组：

```

1. <div id="v-model-multiple-checkboxes">
2.   <input type="checkbox" id="jack" value="Jack" v-model="checkedNames" />
3.   <label for="jack">Jack</label>
4.   <input type="checkbox" id="john" value="John" v-model="checkedNames" />
5.   <label for="john">John</label>
6.   <input type="checkbox" id="mike" value="Mike" v-model="checkedNames" />
7.   <label for="mike">Mike</label>
8.   <br />
9.   <span>Checked names: {{ checkedNames }}</span>
10. </div>

```

```

1. Vue.createApp({
2.   data() {
3.     return {
4.       checkedNames: []
5.     }
6.   }
7. }).mount('#v-model-multiple-checkboxes')

```

单选框 (Radio)

```

1. <div id="v-model-radiobutton">
2.   <input type="radio" id="one" value="One" v-model="picked" />
3.   <label for="one">One</label>
4.   <br />

```

```

5.   <input type="radio" id="two" value="Two" v-model="picked" />
6.   <label for="two">Two</label>
7.   <br />
8.   <span>Picked: {{ picked }}</span>
9. </div>

```

```

1.  Vue.createApp({
2.    data() {
3.      return {
4.        picked: ''
5.      }
6.    }
7.  }).mount('#v-model-radiobutton')

```

选择框 (Select)

单选时:

```

1. <div id="v-model-select" class="demo">
2.   <select v-model="selected">
3.     <option disabled value="">Please select one</option>
4.     <option>A</option>
5.     <option>B</option>
6.     <option>C</option>
7.   </select>
8.   <span>Selected: {{ selected }}</span>
9. </div>

```

```

1.  Vue.createApp({
2.    data() {
3.      return {
4.        selected: ''
5.      }
6.    }
7.  }).mount('#v-model-select')

```

Note

如果 `v-model` 表达式的初始值未能匹配任何选项，`<select>` 元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第一个选项。因为这样的情况下，iOS 不会触发 `change` 事件。因此，更推荐像上面这样提供一个值为空的禁用选项。

多选时（绑定到一个数组）：

```

1. <select v-model="selected" multiple>
2.   <option>A</option>
3.   <option>B</option>
4.   <option>C</option>
5. </select>
6. <br />
7. <span>Selected: {{ selected }}</span>

```

用 `v-for` 渲染的动态选项：

```

1. <div id="v-model-select-dynamic" class="demo">
2.   <select v-model="selected">
3.     <option v-for="option in options" :value="option.value">
4.       {{ option.text }}
5.     </option>
6.   </select>
7.   <span>Selected: {{ selected }}</span>
8. </div>

```

```

1. Vue.createApp({
2.   data() {
3.     return {
4.       selected: 'A',
5.       options: [
6.         { text: 'One', value: 'A' },
7.         { text: 'Two', value: 'B' },
8.         { text: 'Three', value: 'C' }
9.       ]
10.    }
11.  }
12. }).mount('#v-model-select-dynamic')

```

值绑定

对于单选按钮，复选框及选择框的选项，`v-model` 绑定的值通常是静态字符串（对于复选框也可以是布尔值）：

```

1. <!-- 当选中时，`picked` 为字符串 "a" -->

```

```

2. <input type="radio" v-model="picked" value="a" />
3.
4. <!-- `toggle` 为 true 或 false -->
5. <input type="checkbox" v-model="toggle" />
6.
7. <!-- 当选中第一个选项时, `selected` 为字符串 "abc" -->
8. <select v-model="selected">
9.   <option value="abc">ABC</option>
10. </select>

```

但是有时我们可能想把值绑定到当前活动实例的一个动态 property 上, 这时可以用 `v-bind` 实现, 此外, 使用 `v-bind` 可以将输入值绑定到非字符串。

复选框 (Checkbox)

```

1. <input type="checkbox" v-model="toggle" true-value="yes" false-value="no" />

```

```

1. // when checked:
2. vm.toggle === 'yes'
3. // when unchecked:
4. vm.toggle === 'no'

```

Tip

这里的 `true-value` 和 `false-value` attribute 并不会影响输入控件的 `value` attribute, 因为浏览器在提交表单时并不会包含未被选中的复选框。如果要确保表单中这两个值中的一个能够被提交, (即“yes”或“no”), 请换用单选按钮。

单选框 (Radio)

```

1. <input type="radio" v-model="pick" v-bind:value="a" />

```

```

1. // 当选中时
2. vm.pick === vm.a

```

Select Options

```

1. <select v-model="selected">
2.   <!-- 内联对象字面量 -->
3.   <option :value="{ number: 123 }">123</option>

```

```
4. </select>
```

1. `// 当被选中时`
2. `typeof vm.selected // => 'object'`
3. `vm.selected.number // => 123`

修饰符

`.lazy`

在默认情况下，`v-model` 在每次 `input` 事件触发后将输入框的值与数据进行同步（除了上述输入法组合文字时）。你可以添加 `lazy` 修饰符，从而转为在 `change` 事件_之后_进行同步：

1. `<!-- 在“change”时而非“input”时更新 -->`
2. `<input v-model.lazy="msg" />`

`.number`

如果想自动将用户的输入值转为数值类型，可以给 `v-model` 添加 `number` 修饰符：

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

这通常很有用，因为即使在 `type="number"` 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 `parseFloat()` 解析，则会返回原始的值。

`.trim`

如果要自动过滤用户输入的首尾空白字符，可以给 `v-model` 添加 `trim` 修饰符：

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

在组件上使用 `v-model`

如果你还不熟悉 Vue 的组件，可以暂且跳过这里。

HTML 原生的输入元素类型并不总能满足需求。幸好，Vue 的组件系统允许你创建具有完全自定义行为且可复用的输入组件。这些输入组件甚至可以和 `v-model` 一起使用！

要了解更多信息，请参阅组件指南中的[自定义输入组件](#)。

组件基础

基本实例

这里有一个 Vue 组件的示例：

```
1. // 创建一个Vue 应用
2. const app = Vue.createApp({})
3.
4. // 定义一个名为 button-counter 的新全局组件
5. app.component('button-counter', {
6.   data() {
7.     return {
8.       count: 0
9.     }
10.  },
11.   template: `
12.     <button @click="count++">
13.       You clicked me {{ count }} times.
14.     </button>`
15. })
```

INFO

在这里演示的是一个简单的示例，但是在典型的 Vue 应用程序中，我们使用单个文件组件而不是字符串模板。你可以在[本节](#)找到有关它们的更多信息。

组件是可复用的组件实例，且带有一个名字：在这个例子中是 `<button-counter>`。我们可以在一个通过 `new Vue` 创建的 Vue 根实例中，把这个组件作为自定义元素来使用：

```
1. <div id="components-demo">
2.   <button-counter></button-counter>
3. </div>
```

```
1. app.mount('#components-demo')
```

因为组件是可复用的组件实例，所以它们与 `new Vue` 接收相同的选项，例如

`data`、`computed`、`watch`、`methods` 以及生命周期钩子等。仅有的例外是像 `el` 这样根实例特有的选项。

组件的复用

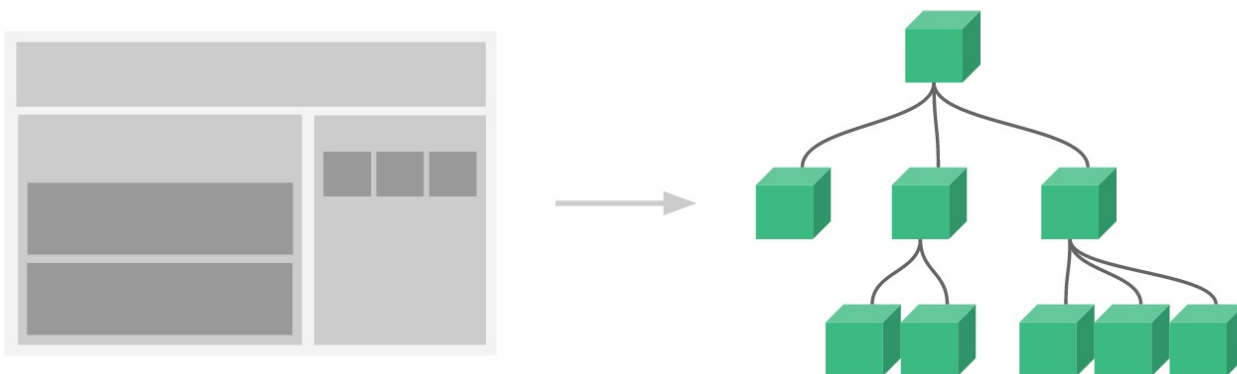
你可以将组件进行任意次数的复用：

```
1. <div id="components-demo">
2.   <button-counter></button-counter>
3.   <button-counter></button-counter>
4.   <button-counter></button-counter>
5. </div>
```

注意当点击按钮时，每个组件都会各自独立维护它的 `count`。因为你每用一次组件，就会有一个它的新实例被创建。

组件的组织

通常一个应用会以一棵嵌套的组件树的形式来组织：



例如，你可能会有页头、侧边栏、内容区等组件，每个组件又包含了其它的像导航链接、博文之类的组件。

为了能在模板中使用，这些组件必须先注册以便 Vue 能够识别。这里有两种组件的注册类型：全局注册和局部注册。至此，我们的组件都只是通过 `component` 全局注册的：

```
1. const app = Vue.createApp({})
2.
3. app.component('my-component-name', {
4.   // ... 选项 ...
5. })
```

全局注册的组件可以在随后创建的 `app` 实例模板中使用，也包括根实例组件树中的所有子组件的模板中。

到目前为止，关于组件注册你需要了解的就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把[组件注册](#)读完。

通过 Prop 向子组件传递数据

早些时候，我们提到了创建一个博文组件的事情。问题是如果你不能向这个组件传递某一篇博文的标题或内容之类的我们想展示的数据的话，它是没有办法使用的。这也正是 `prop` 的由来。

`Prop` 是你可以在组件上注册的一些自定义 `attribute`。当一个值传递给一个 `prop attribute` 的时候，它就变成了那个组件实例的一个 `property`。为了给博文组件传递一个标题，我们可以用一个 `props` 选项将其包含在该组件可接受的 `prop` 列表中：

```
1. const app = Vue.createApp({})
2.
3. app.component('blog-post', {
4.   props: ['title'],
5.   template: `<h4>{{ title }}</h4>`
6. })
7.
8. app.mount('#blog-post-demo')
```

一个组件默认可以拥有任意数量的 `prop`，任何值都可以传递给任何 `prop`。在上述模板中，你会发现我们能够在组件实例中访问这个值，就像访问 `data` 中的值一样。

一个 `prop` 被注册之后，你就可以像这样把数据作为一个自定义 `attribute` 传递进来：

```
1. <div id="blog-post-demo" class="demo">
2.   <blog-post title="My journey with Vue"></blog-post>
3.   <blog-post title="Blogging with Vue"></blog-post>
4.   <blog-post title="Why Vue is so fun"></blog-post>
5. </div>
```

然而在一个典型的应用中，你可能在 `data` 里有一个博文的数组：

```
1. const App = {
2.   data() {
3.     return {
4.       posts: [
5.         { id: 1, title: 'My journey with Vue' },
```



```

6.     { id: 2, title: ' Blogging with Vue' },
7.     { id: 3, title: ' Why Vue is so fun' }
8.   ]
9.   }
10.  }
11. }
12.
13. const app = Vue.createApp({})
14.
15. app.component('blog-post', {
16.   props: ['title'],
17.   template: `<h4>{{ title }}</h4>`
18. })
19.
20. app.mount('#blog-posts-demo')

```

并想要为每篇博文渲染一个组件：

```

1. <div id="blog-posts-demo">
2.   <blog-post
3.     v-for="post in posts"
4.     :key="post.id"
5.     :title="post.title"
6.   ></blog-post>
7. </div>

```

如上所示，你会发现我们可以使用 `v-bind` 来动态传递 prop。这在你一开始不清楚要渲染的具体内容，是非常有用的。

到目前为止，关于 prop 你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把 [prop](#) 读完。

监听子组件事件

在我们开发 `<blog-post>` 组件时，它的一些功能可能要求我们和父级组件进行沟通。例如我们可能会引入一个辅助功能来放大博文的字号，同时让页面的其它部分保持默认的字号。

在其父组件中，我们可以通过添加一个 `postFontSize` 数据 property 来支持这个功能：

```

1. const App = {
2.   data() {
3.     return {

```

```

4.     posts: [
5.         /* ... */
6.     ],
7.     postFontSize: 1
8.   }
9. }
10. }

```

它可以在模板中用来控制所有博文的字号：

```

1. <div id="blog-posts-events-demo">
2.   <div v-bind:style="{ fontSize: postFontSize + 'em' }">
3.     <blog-post v-for="post in posts" :key="post.id" :title="title"></blog-post>
4.   </div>
5. </div>

```

现在我们在每篇博文正文之前添加一个按钮来放大字号：

```

1. app.component('blog-post', {
2.   props: ['title'],
3.   template: `
4.     <div class="blog-post">
5.       <h4>{{ title }}</h4>
6.       <button>
7.         Enlarge text
8.       </button>
9.     </div>
10. `
11. })

```

问题是这个按钮不会做任何事：

```

1. <button>
2.   Enlarge text
3. </button>

```

当点击这个按钮时，我们需要告诉父级组件放大所有博文的文本。幸好组件实例提供了一个自定义事件的系统来解决这个问题。父级组件可以像处理 native DOM 事件一样通过 `v-on` 或 `@` 监听子组件实例的任意事件：

```

1. <blog-post ... @enlarge-text="postFontSize += 0.1"></blog-post>

```

同时子组件可以通过调用内建的 `$emit` 方法并传入事件名称来触发一个事件：

```
1. <button @click="$emit('enlarge-text')">
2.   Enlarge text
3. </button>
```

多亏了 `@enlarge-text="postFontSize += 0.1"` 监听器，父级将接收事件并更新 `postFontSize` 值。

我们可以在组件的 `emits` 选项中列出已抛出的事件。

```
1. app.component('blog-post', {
2.   props: ['title'],
3.   emits: ['enlarge-text']
4. })
```

这将允许你检查组件抛出的所有事件，还可以选择 `validate them`

使用事件抛出一个值

有的时候用一个事件来抛出一个特定的值是非常有用的。例如我们可能想让 `<blog-post>` 组件决定它的文本要放大多少。这时可以使用 `$emit` 的第二个参数来提供这个值：

```
1. <button @click="$emit('enlarge-text', 0.1)">
2.   Enlarge text
3. </button>
```

然后当在父级组件监听这个事件的时候，我们可以通过 `$event` 访问到被抛出的这个值：

```
1. <blog-post ... @enlarge-text="postFontSize += $event"></blog-post>
```

或者，如果这个事件处理函数是一个方法：

```
1. <blog-post ... @enlarge-text="onEnlargeText"></blog-post>
```

那么这个值将会作为第一个参数传入这个方法：

```
1. methods: {
2.   onEnlargeText(enlargeAmount) {
3.     this.postFontSize += enlargeAmount
4.   }
```

```
5. }
```

在组件上使用 v-model

自定义事件也可以用于创建支持 `v-model` 的自定义输入组件。记住：

```
1. <input v-model="searchText" />
```

等价于：

```
1. <input :value="searchText" @input="searchText = $event.target.value" />
```

当用在组件上时，`v-model` 则会这样：

```
1. <custom-input
2.   :model-value="searchText"
3.   @update:model-value="searchText = $event"
4. ></custom-input>
```

WARNING

请注意，我们在这里使用的是 `model value`，因为我们使用的是 DOM 模板中的 kebab-case。你可以在 [DOM Template Parsing Caveats](#) 部分找到关于 kebab cased 和 camelCased 属性的详细说明

为了让它正常工作，这个组件内的 `<input>` 必须：

- 将其 `value` attribute 绑定到一个名叫 `modelValue` 的 prop 上
- 在其 `input` 事件被触发时，将新的值通过自定义的 `update:modelValue` 事件抛出

写成代码之后是这样的：

```
1. app.component('custom-input', {
2.   props: ['modelValue'],
3.   template: `
4.     <input
5.       :value="modelValue"
6.       @input="$emit('update:modelValue', $event.target.value)"
7.     >
8.   `
9. })
```

现在 `v-model` 就应该可以在这个组件上完美地工作起来了：

```
1. <custom-input v-model="searchText"></custom-input>
```

在自定义组件中创建 `v-model` 功能的另一种方法是使用 `computed` property 的功能来定义 getter 和 setter。

在下面的示例中，我们使用计算属性重构 `<custom-input>` 组件。

请记住，`get` 方法应返回 `modelValue` property，或用于绑定的任何 property，`set` 方法应为该 property 触发相应的 `$emit`。

```
1. app.component('custom-input', {
2.   props: ['modelValue'],
3.   template: `
4.     <input v-model="value">
5.   `,
6.   computed: {
7.     value: {
8.       get() {
9.         return this.modelValue
10.      },
11.      set(value) { this.$emit('update:modelValue', value)
12.    }
13.  }
14. }
15. })
```

现在你只需要了解自定义组件事件，但一旦你读完本页并对其内容还觉得不错，我们建议你稍后再阅读有关[自定义事件](#)

通过插槽分发内容

和 HTML 元素一样，我们经常需要向一个组件传递内容，像这样：

```
1. <alert-box>
2.   Something bad happened.
3. </alert-box>
```

可能会渲染出这样的东西：

幸好，Vue 自定义的 `<slot>` 元素让这变得非常简单：

```

1. app.component('alert-box', {
2.   template: `
3.     <div class="demo-alert-box">
4.       <strong>Error!</strong>
5.       <slot></slot>
6.     </div>
7.   `
8. })

```

如你所见，我们只要在需要的地方加入插槽就行了——就这么简单！

到目前为止，关于插槽你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把[插槽](#)读完。

动态组件

有的时候，在不同组件之间进行动态切换是非常有用的，比如在一个多标签的界面里：

上述内容可以通过 Vue 的 `<component>` 元素加一个特殊的 `is` attribute 来实现：

```

1. <!-- 组件会在 `currentTabComponent` 改变时改变 -->
2. <component :is="currentTabComponent"></component>

```

在上述示例中，`currentTabComponent` 可以包括

- 已注册组件的名字，或
- 一个组件的选项对象

你可以在[这里](#) [↗] (opens new window) 查阅并体验完整的代码，或在[这个版本](#) [↗] (opens new window) 了解绑定组件选项对象，而不是已注册组件名的示例。

请注意，这个 attribute 可以用于常规 HTML 元素，但这些元素将被视为组件，这意味着所有的 attribute 都会作为 **DOM attribute** 被绑定。对于像 `value` 这样的 property，若想让其如预期般工作，你需要使用 `.prop` 修饰器。

到目前为止，关于动态组件你需要了解的大概就这些了，如果你阅读完本页内容并掌握了它的内容，我们会推荐你再回来把[动态 & 异步组件](#)读完。

解析 DOM 模板时的注意事项

有些 HTML 元素，诸如 ``、``、`<table>` 和 `<select>`，对于哪些元素可以出

现在其内部是有严格限制的。而有些元素，诸如 ``、`<tr>` 和 `<option>`，只能出现在其它某些特定的元素内部。

这会导致我们使用这些有约束条件的元素时遇到一些问题。例如：

```
1. <table>
2.   <blog-post-row></blog-post-row>
3. </table>
```

这个自定义组件 `<blog-post-row>` 会被作为无效的内容提升到外部，并导致最终渲染结果出错。幸好这个特殊的 `v-is` attribute 给了我们一个变通的办法：

```
1. <table>
2.   <tr v-is="'blog-post-row'"></tr>
3. </table>
```

WARNING

`v-is` 值应为 JavaScript 字符串文本：

```
1. <!-- 错误的，这样不会渲染任何东西 -->
2. <tr v-is="blog-post-row"></tr>
3.
4. <!-- 正确的 -->
5. <tr v-is="'blog-post-row'"></tr>
```

另外，HTML 属性名不区分大小写，因此浏览器将把所有大写字符解释为小写。这意味着当你在 DOM 模板中使用时，驼峰 prop 名称和 event 处理器参数需要使用它们的 kebab-cased (横线字符分隔) 等效值：

```
1. // 在JavaScript中的驼峰
2.
3. app.component('blog-post', {
4.   props: ['postTitle'],
5.   template: `
6.     <h3>{{ postTitle }}</h3>
7.   `
8. })
```

```
1. <!-- 在HTML则是横线字符分割 -->
2.
```

```
3. <blog-post post-title="hello!"></blog-post>
```

需要注意的是如果我们从以下来源使用模板的话，这条限制是不存在的：

- 字符串模板（例如：`template: '...'`）
- [单文件组件](#)
- `<script type="text/x-template">`

到这里，你需要了解的解析 DOM 模板时的注意事项——实际上也是 Vue 的全部必要内容，大概就是这些了。恭喜你！接下来还有很多东西要去学习，不过首先，我们推荐你先休息一下，试用一下 Vue，自己随意做些好玩的东西。

如果你感觉已经掌握了这些知识，我们推荐你再回来把完整的[组件&异步组件指南](#)，包括侧边栏中组件深入章节的所有页面读完。

- [组件注册](#)
- [Props](#)
- [非 Prop 的 Attribute](#)
- [自定义事件](#)
- [插槽](#)
- [提供 / 注入](#)
- [动态组件 & 异步组件](#)
- [模板引用](#)
- [处理边界情况](#)

组件注册

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

组件名

在注册一个组件的时候，我们始终需要给它一个名字。比如在全局注册的时候我们已经看到了：

```
1. const app = Vue.createApp({...})
2.
3. app.component('my-component-name', {
4.   /* ... */
5. })
```

该组件名就是 `app.component` 的第一个参数，在上面的例子中，组件的名称是“my-component-name”。

你给予组件的名字可能依赖于你打算拿它来做什么。当直接在 DOM 中使用一个组件（而不是在字符串模板或单文件组件）的时候，我们强烈推荐遵循 [W3C 规范](#) (opens new window) 中的自定义组件名（字母全小写且必须包含一个连字符）。这会帮助你避免和当前以及未来的 HTML 元素相冲突。

1. 全部小写
2. 包含连字符（及：即有多个单词与连字符符号连接）

这样会帮助你避免与当前以及未来的 HTML 元素发生冲突。

你可以在[风格指南](#)中查阅到关于组件名的其它建议。

组件名大小写

在字符串模板或单个文件组件中定义组件时，定义组件名的方式有两种：

使用 kebab-case

```
1. app.component('my-component-name', {
2.   /* ... */
3. })
```

当使用 kebab-case（短横线分隔命名）定义一个组件时，你也必须在引用这个自定义元素时使用 kebab-case，例如 `<my-component-name>`。

使用 PascalCase

```
1. app.component('MyComponentName', {
2.   /* ... */
3. })
```

当使用 PascalCase（首字母大写命名）定义一个组件时，你在引用这个自定义元素时两种命名法都可以使用。也就是说 `<my-component-name>` 和 `<MyComponentName>` 都是可接受的。注意，尽管如此，直接在 DOM（即非字符串的模板）中使用时只有 kebab-case 是有效的。

全局注册

到目前为止，我们只用过 `app.component` 来创建组件：

```
1. Vue.createApp({...}).component('my-component-name', {
2.   // ... 选项 ...
3. })
```

这些组件是全局注册的。也就是说它们在注册之后可以用在任何新创建的组件实例的模板中。比如：

```
1. const app = Vue.createApp({})
2.
3. app.component('component-a', {
4.   /* ... */
5. })
6. app.component('component-b', {
7.   /* ... */
8. })
9. app.component('component-c', {
10.  /* ... */
11. })
12.
13. app.mount('#app')
```

```
1. <div id="app">
2.   <component-a></component-a>
3.   <component-b></component-b>
4.   <component-c></component-c>
5. </div>
```

在所有子组件中也是如此，也就是说这三个组件在各自内部也都可以相互使用。

局部注册

全局注册往往是不够理想的。比如，如果你使用一个像 webpack 这样的构建系统，全局注册所有的组件意味着即便你已经不再使用一个组件了，它仍然会被包含在你最终的构建结果中。这造成了用户下载的 JavaScript 的无谓的增加。

在这些情况下，你可以通过一个普通的 JavaScript 对象来定义组件：

```
1. const ComponentA = {
2.   /* ... */
3. }
4. const ComponentB = {
5.   /* ... */
6. }
7. const ComponentC = {
8.   /* ... */
9. }
```

然后在 `components` 选项中定义你想要使用的组件：

```
1. const app = Vue.createApp({
2.   components: {
3.     'component-a': ComponentA,
4.     'component-b': ComponentB
5.   }
6. })
```

对于 `components` 对象中的每个 property 来说，其 property 名就是自定义元素的名字，其 property 值就是这个组件的选项对象。

注意局部注册的组件在其子组件中不可用。例如，如果你希望 `ComponentA` 在 `ComponentB` 中可用，则你需要这样写：

```
1. const ComponentA = {
2.   /* ... */
3. }
4.
5. const ComponentB = {
6.   components: {
```

```

7.     'component-a': ComponentA
8.   }
9.   // ...
10. }
```

或者如果你通过 Babel 和 webpack 使用 ES2015 模块，那么代码看起来更像：

```

1. import ComponentA from './ComponentA.vue'
2.
3. export default {
4.   components: {
5.     ComponentA
6.   }
7.   // ...
8. }
```

注意在 ES2015+ 中，在对象中放一个类似 `ComponentA` 的变量名其实是 `ComponentA : ComponentA` 的缩写，即这个变量名同时是：

- 用在模板中的自定义元素的名称
- 包含了这个组件选项的变量名

模块系统

如果你没有通过 `import` / `require` 使用一个模块系统，也许可以暂且跳过这个章节。如果你使用了，那么我们会为你提供一些特殊的使用说明和注意事项。

在模块系统中局部注册

如果你还在阅读，说明你使用了诸如 Babel 和 webpack 的模块系统。在这些情况下，我们推荐创建一个 `components` 目录，并将每个组件放置在其各自的文件中。

然后你需要在局部注册之前导入每个你想使用的组件。例如，在一个假设的 `ComponentB.js` 或 `ComponentB.vue` 文件中：

```

1. import ComponentA from './ComponentA'
2. import ComponentC from './ComponentC'
3.
4. export default {
5.   components: {
6.     ComponentA,
7.     ComponentC
```

组件注册

```
8.   }  
9.   // ...  
10. }
```

现在 `ComponentA` 和 `ComponentC` 都可以在 `ComponentB` 的模板中使用了。

Props

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

Prop 类型

到这里，我们只看到了以字符串数组形式列出的 prop：

```
1. props: ['title', 'likes', 'isPublished', 'commentIds', 'author']
```

但是，通常你希望每个 prop 都有指定的值类型。这时，你可以以对象形式列出 prop，这些 property 的名称和值分别是 prop 各自的名称和类型：

```
1. props: {
2.   title: String,
3.   likes: Number,
4.   isPublished: Boolean,
5.   commentIds: Array,
6.   author: Object,
7.   callback: Function,
8.   contactsPromise: Promise // 或任何其他构造函数
9. }
```

这不仅为你的组件提供了文档，还会在它们遇到错误的类型时从浏览器的 JavaScript 控制台提示用户。你会在这个页面接下来的部分看到[类型检查和其它 prop 验证](#)。

传递静态或动态的 Prop

这样，你已经知道了可以像这样给 prop 传入一个静态的值：

```
1. <blog-post title="My journey with Vue"></blog-post>
```

你也知道 prop 可以通过 `v-bind` 或简写 `:` 动态赋值，例如：

```
1. <!-- 动态赋予一个变量的值 -->
2. <blog-post :title="post.title"></blog-post>
3.
4. <!-- 动态赋予一个复杂表达式的值 -->
```

```
5. <blog-post :title="post.title + ' by ' + post.author.name"></blog-post>
```

在上述两个示例中，我们传入的值都是字符串类型的，但实际上任何类型的值都可以传给一个 prop。

传入一个数字

```
1.
2. <!-- 即便 `42` 是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
3. <!-- 这是一个 JavaScript 表达式而不是一个字符串。 -->
4. <blog-post :likes="42"></blog-post>
5.
6. <!-- 用一个变量进行动态赋值。 -->
7. <blog-post :likes="post.likes"></blog-post>
```

传入一个布尔值

```
1. <!-- 包含该 prop 没有值的情况在内，都意味着 `true`。 -->
2. <blog-post is-published></blog-post>
3.
4. <!-- 即便 `false` 是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
5. <!-- 这是一个 JavaScript 表达式而不是一个字符串。 -->
6. <blog-post :is-published="false"></blog-post>
7.
8. <!-- 用一个变量进行动态赋值。 -->
9. <blog-post :is-published="post.isPublished"></blog-post>
```

传入一个数组

```
1. <!-- 即便数组是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
2. <!-- 这是一个 JavaScript 表达式而不是一个字符串。 -->
3. <blog-post :comment-ids="[234, 266, 273]"></blog-post>
4.
5. <!-- 用一个变量进行动态赋值。 -->
6. <blog-post :comment-ids="post.commentIds"></blog-post>
```

传入一个对象

```
1. <!-- 即便对象是静态的，我们仍然需要 `v-bind` 来告诉 Vue -->
2. <!-- 这是一个 JavaScript 表达式而不是一个字符串。 -->
3. <blog-post
```



```

4.   :author="{
5.     name: 'Veronica',
6.     company: 'Veridian Dynamics'
7.   }"
8. ></blog-post>
9.
10. <!-- 用一个变量进行动态赋值。 -->
11. <blog-post :author="post.author"></blog-post>

```

传入一个对象的所有 property

如果你想要将一个对象的所有 property 都作为 prop 传入，你可以使用不带参数的 `v-bind` (取代 `v-bind : prop-name`)。例如，对于一个给定的对象 `post`：

```

1. post: {
2.   id: 1,
3.   title: 'My Journey with Vue'
4. }

```

下面的模板：

```
1. <blog-post v-bind="post"></blog-post>
```

等价于：

```
1. <blog-post v-bind:id="post.id" v-bind:title="post.title"></blog-post>
```

单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外变更父级组件的状态，从而导致你的应用的数据流向难以理解。

另外，每次父级组件发生变更时，子组件中所有的 prop 都将会刷新为最新的值。这意味着你不应该在一个子组件内部改变 prop。如果你这样做了，Vue 会在浏览器的控制台中发出警告。

这里有两种常见的试图变更一个 prop 的情形：

1. 这个 **prop** 用来传递一个初始值；这个子组件接下来希望将其作为一个本地的 **prop** 数据来使用。在这种情况下，最好定义一个本地的 data property 并将这个 prop 作为其初始值：

```

1. props: ['initialCounter'],
2. data() {
3.   return {
4.     counter: this.initialCounter
5.   }
6. }

```

1. 这个 **prop** 以一种原始的值传入且需要进行转换。在这种情况下，最好使用这个 prop 的值来定义一个计算属性：

```

1. props: ['size'],
2. computed: {
3.   normalizedSize: function () {
4.     return this.size.trim().toLowerCase()
5.   }
6. }

```

提示

注意在 JavaScript 中对象和数组是通过引用传入的，所以对于一个数组或对象类型的 prop 来说，在子组件中改变变更这个对象或数组本身将会影响到父组件的状态。

Prop 验证

我们可以为组件的 prop 指定验证要求，例如你知道的这些类型。如果有一个需求没有被满足，则 Vue 会在浏览器控制台中警告你。这在开发一个会被别人用到的组件时尤其有帮助。

为了定制 prop 的验证方式，你可以为 `props` 中的值提供一个带有验证需求的对象，而不是一个字符串数组。例如：

```

1. app.component('my-component', {
2.   props: {
3.     // 基础的类型检查（`null` 和 `undefined` 会通过任何类型验证）
4.     propA: Number,
5.     // 多个可能的类型
6.     propB: [String, Number],
7.     // 必填的字符串
8.     propC: {
9.       type: String,
10.      required: true
11.    },

```

```

12. // 带有默认值的数字
13. propD: {
14.   type: Number,
15.   default: 100
16. },
17. // 带有默认值的对象
18. propE: {
19.   type: Object,
20.   // 对象或数组默认值必须从一个工厂函数获取
21.   default: function() {
22.     return { message: 'hello' }
23.   }
24. },
25. // 自定义验证函数
26. propF: {
27.   validator: function(value) {
28.     // 这个值必须匹配下列字符串中的一个
29.     return ['success', 'warning', 'danger'].indexOf(value) !== -1
30.   }
31. },
32. // 具有默认值的函数
33. propG: {
34.   type: Function,
35.   // 与对象或数组默认值不同，这不是一个工厂函数 — 这是一个用作默认值的函数
36.   default: function() {
37.     return 'Default function'
38.   }
39. }
40. }
41. })

```

当 prop 验证失败的时候，（开发环境构建版本的）Vue 将会产生一个控制台的警告。

提示

注意那些 prop 会在一个组件实例创建之前进行验证，所以实例的 property（如 `data`、`computed` 等）在 `default` 或 `validator` 函数中是不可用的。

类型检查

`type` 可以是下列原生构造函数中的一个：

- String

- Number
- Boolean
- Array
- Object
- Date
- Function
- Symbol

此外，`type` 还可以是一个自定义的构造函数，并且通过 `instanceof` 来进行检查确认。例如，给定下列现成的构造函数：

```
1. function Person(firstName, lastName) {
2.   this.firstName = firstName
3.   this.lastName = lastName
4. }
```

你可以使用：

```
1. app.component('blog-post', {
2.   props: {
3.     author: Person
4.   }
5. })
```

用于验证 `author` prop 的值是否是通过 `new Person` 创建的。

Prop 的大小写命名 (camelCase vs kebab-case)

HTML 中的 attribute 名是大小写不敏感的，所以浏览器会把所有大写字符解释为小写字符。这意味着当你使用 DOM 中的模板时，camelCase (驼峰命名法) 的 prop 名需要使用其等价的 kebab-case (短横线分隔命名) 命名：

```
1. const app = Vue.createApp({})
2.
3. app.component('blog-post', {
4.   // camelCase in JavaScript
5.   props: ['postTitle'],
6.   template: '<h3>{{ postTitle }}</h3>'
7. })
```

1. `<!-- kebab-case in HTML -->`
2. `<blog-post post-title="hello!"></blog-post>`

重申一次，如果你使用字符串模板，那么这个限制就不存在了。

非 Prop 的 Attribute

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

一个非 prop 的 attribute 是指传向一个组件，但是该组件并没有相应 `props` 或 `emits` 定义的 attribute。常见的示例包括 `class`、`style` 和 `id` 属性。

Attribute 继承

当组件返回单个根节点时，非 prop attribute 将自动添加到根节点的 attribute 中。例如，在 `<date-picker>` 组件的实例中：

```

1. app.component('date-picker', {
2.   template: `
3.     <div class="date-picker">
4.       <input type="datetime" />
5.     </div>
6.   `
7. })

```

如果我们需要通过 `data status` property 定义 `<date-picker>` 组件的状态，它将应用于根节点（即 `div.date-picker`）。

```

1. <!-- 具有非prop attribute的Date-picker组件-->
2. <date-picker data-status="activated"></date-picker>
3.
4. <!-- 渲染 date-picker 组件 -->
5. <div class="date-picker" data-status="activated">
6.   <input type="datetime" />
7. </div>

```

同样的规则适用于事件监听器：

```

1. <date-picker @change="submitChange"></date-picker>

```

```

1. app.component('date-picker', {
2.   created() {
3.     console.log(this.$attrs) // { onChange: () => {} }
4.   }

```

```
5. })
```

当有一个 HTML 元素将 `change` 事件作为 `date-picker` 的根元素时，这可能会有帮助。

```
1. app.component('date-picker', {
2.   template: `
3.     <select>
4.       <option value="1">Yesterday</option>
5.       <option value="2">Today</option>
6.       <option value="3">Tomorrow</option>
7.     </select>
8.   `
9. })
```

在这种情况下，`change` 事件监听器从父组件传递到子组件，它将在原生 `select` 的 `change` 事件上触发。我们不需要显式地从 `date-picker` 发出事件：

```
1. <div id="date-picker" class="demo">
2.   <date-picker @change="showChange"></date-picker>
3. </div>
```

```
1. const app = Vue.createApp({
2.   methods: {
3.     showChange(event) {
4.       console.log(event.target.value) // 将记录所选选项的值
5.     }
6.   }
7. })
```

禁用 Attribute 继承

如果你不希望组件的根元素继承 attribute，你可以在组件的选项中设置 `inheritAttrs: false`。例如：

禁用 attribute 继承的常见情况是需要将 attribute 应用于根节点之外的其他元素。

通过将 `inheritAttrs` 选项设置为 `false`，你可以访问组件的 `$attrs` property，该 property 包括组件 `props` 和 `emits` property 中未包含的所有属性（例如，`class`、`style`、`v-on` 监听器等）。

使用上一节中的 `date-picker` 组件示例，如果需要将所有非 prop attribute 应用于 `input`

元素而不是根 `div` 元素，则可以使用 `v-bind` 缩写来完成。

```

1. app.component('date-picker', {
2.   inheritAttrs: false,
3.   template: `
4.     <div class="date-picker">
5.       <input type="datetime" v-bind="$attrs" />
6.     </div>
7.   `
8. })

```

有了这个新配置，`data status` attribute 将应用于 `input` 元素！

```

1. <!-- Date-picker 组件 使用非 prop attribute -->
2. <date-picker data-status="activated"></date-picker>
3.
4. <!-- 渲染 date-picker 组件 -->
5. <div class="date-picker">
6.   <input type="datetime" data-status="activated" />
7. </div>

```

多个根节点上的 Attribute 继承

与单个根节点组件不同，具有多个根节点的组件不具有自动 `attribute` 回退行为。如果未显式绑定 `$attrs`，将发出运行时警告。

```

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

```

```

1. // 这将发出警告
2. app.component('custom-layout', {
3.   template: `
4.     <header>...</header>
5.     <main>...</main>
6.     <footer>...</footer>
7.   `
8. })
9.
10. // 没有警告, $attrs被传递到<main>元素
11. app.component('custom-layout', {
12.   template: `

```



```
13.     <header>...</header>
14.     <main v-bind="$attrs">...</main>
15.     <footer>...</footer>
16.     `
17.  })
```

自定义事件

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

事件名

不同于组件和 prop，事件名不存在任何自动化的大小写转换。而是触发的事件名需要完全匹配监听这个事件所用的名称。举个例子，如果触发一个 camelCase 名字的事件：

```
1. this.$emit('myEvent')
```

则监听这个名字的 kebab-case 版本是不会有任意效果的：

```
1. <!-- 没有效果 -->
2. <my-component @my-event="doSomething"></my-component>
```

不同于组件和 prop，事件名不会被用作一个 JavaScript 变量名或 property 名，所以就没有理由使用 camelCase 或 PascalCase 了。并且 `v-on` 事件监听器在 DOM 模板中会被自动转换为全小写（因为 HTML 是大小写不敏感的），所以 `@myEvent` 将会变成 `@myevent` ——导致 `myEvent` 不可能被监听到。

因此，我们推荐你始终使用 **kebab-case** 的事件名。

定义自定义事件

在 [Vue School](#) 上观看关于定义自定义事件的免费视频。

可以通过 `emits` 选项在组件上定义已发出的事件。

```
1. app.component('custom-form', {
2.   emits: ['in-focus', 'submit']
3. })
```

当在 `emits` 选项中定义了原生事件（如 `click`）时，将使用组件中的事件替代原生事件侦听器。

TIP

建议定义所有发出的事件，以便更好地记录组件应该如何工作。

验证抛出的事件

与 `prop` 类型验证类似，如果使用对象语法而不是数组语法定义发出的事件，则可以验证它。

要添加验证，将为事件分配一个函数，该函数接收传递给 `$emit` 调用的参数，并返回一个布尔值以指示事件是否有效。

```
1. app.component('custom-form', {
2.   emits: {
3.     // 没有验证
4.     click: null,
5.
6.     // 验证submit 事件
7.     submit: ({ email, password }) => {
8.       if (email && password) {
9.         return true
10.      } else {
11.        console.warn('Invalid submit event payload!')
12.        return false
13.      }
14.    }
15.  },
16.  methods: {
17.    submitForm() {
18.      this.$emit('submit', { email, password })
19.    }
20.  }
21. })
```

`v-model` 参数

默认情况下，组件上的 `v-model` 使用 `modelValue` 作为 `prop` 和 `update:modelValue` 作为事件。我们可以通过向 `v-model` 传递参数来修改这些名称：

```
1. <my-component v-model:foo="bar"></my-component>
```

在本例中，子组件将需要一个 `foo` `prop` 并发出 `update:foo` 要同步的事件：

```
1. const app = Vue.createApp({})
2.
3. app.component('my-component', {
```

```
4.   props: {
5.     foo: String
6.   },
7.   template: `
8.     <input
9.       type="text"
10.      :value="foo"
11.      @input="$emit('update:foo', $event.target.value)">
12.   `
13. })
```

```
1. <my-component v-model:foo="bar"></my-component>
```

多个 `v-model` 绑定

通过利用以特定 prop 和事件为目标的能力，正如我们之前在 `v-model` 参数中所学的那样，我们现在可以在单个组件实例上创建多个 `v-model` 绑定。

每个 `v-model` 将同步到不同的 prop，而不需要在组件中添加额外的选项：

```
1. <user-name
2.   v-model:first-name="firstName"
3.   v-model:last-name="lastName"
4. ></user-name>
```

```
1. const app = Vue.createApp({})
2.
3. app.component('user-name', {
4.   props: {
5.     firstName: String,
6.     lastName: String
7.   },
8.   template: `
9.     <input
10.      type="text"
11.      :value="firstName"
12.      @input="$emit('update:firstName', $event.target.value)">
13.
14.     <input
15.      type="text"
```

```

16.     :value="lastName"
17.     @input="$emit('update:lastName', $event.target.value)">
18.   `
19. })

```

处理 `v-model` 修饰符

在 2.x 中，我们对组件 `v-model` 上的 `.trim` 等修饰符提供了硬编码支持。但是，如果组件可以支持自定义修饰符，则会更有用。在 3.x 中，添加到组件 `v-model` 的修饰符将通过 `modelModifiers` prop 提供给组件：

当我们学习表单输入绑定时，我们看到 `v-model` 有内置修饰符——`.trim`、`.number` 和 `.lazy`。但是，在某些情况下，你可能还需要添加自己的自定义修饰符。

让我们创建一个示例自定义修饰符 `capitalize`，它将 `v-model` 绑定提供的字符串的第一个字母大写。

添加到组件 `v-model` 的修饰符将通过 `modelModifiers` prop 提供给组件。在下面的示例中，我们创建了一个组件，其中包含默认为空对象的 `modelModifiers` prop。

请注意，当组件的 `created` 生命周期钩子触发时，`modelModifiers` prop 包含 `capitalize`，其值为 `true`——因为它被设置在 `v-model` 绑定 `v-model.capitalize="bar"`。

```

1. <my-component v-model.capitalize="bar"></my-component>

```

```

1. app.component('my-component', {
2.   props: {
3.     modelValue: String,
4.     modelModifiers: {
5.       default: () => ({}),
6.     }
7.   },
8.   template: `
9.     <input type="text"
10.      :value="modelValue"
11.      @input="$emit('update:modelValue', $event.target.value)">
12.   `
13.   created() {
14.     console.log(this.modelModifiers) // { capitalize: true }
15.   }

```

```
16. })
```

现在我们已经设置了 prop，我们可以检查 `modelModifiers` 对象键并编写一个处理器来更改发出的值。在下面的代码中，每当 `<input/>` 元素触发 `input` 事件时，我们都将字符串大写。

```
1. <div id="app">
2.   <my-component v-model.capitalize="myText"></my-component>
3.   {{ myText }}
4. </div>
```

```
1. const app = Vue.createApp({
2.   data() {
3.     return {
4.       myText: ''
5.     }
6.   }
7. })
8.
9. app.component('my-component', {
10.  props: {
11.    modelValue: String,
12.    modelModifiers: {
13.      default: () => ({}),
14.    },
15.  },
16.  methods: {
17.    emitValue(e) {
18.      let value = e.target.value
19.      if (this.modelModifiers.capitalize) {
20.        value = value.charAt(0).toUpperCase() + value.slice(1)
21.      }
22.      this.$emit('update:modelValue', value)
23.    }
24.  },
25.  template: `<input
26.    type="text"
27.    :value="modelValue"
28.    @input="emitValue">`
29. })
30.
31. app.mount('#app')
```

对于带参数的 `v-model` 绑定, 生成的 prop 名称将为 `arg + "Modifiers"` :

```
1. <my-component v-model:foo.capitalize="bar"></my-component>
```

```
1. app.component('my-component', {
2.   props: ['foo', 'fooModifiers'],
3.   template: `
4.     <input type="text"
5.       :value="foo"
6.       @input="$emit('update:foo', $event.target.value)">
7.   `,
8.   created() {
9.     console.log(this.fooModifiers) // { capitalize: true }
10.  }
11. })
```

插槽

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

插槽内容

Vue 实现了一套内容分发的 API，这套 API 的设计灵感源自 [Web Components 规范草案](#) (opens new window)，将 `<slot>` 元素作为承载分发内容的出口。

它允许你像这样合成组件：

```
1. <todo-button>
2.   Add todo
3. </todo-button>
```

然后在 `<todo-button>` 的模板中，你可能有：

```
1. <!-- todo-button 组件模板 -->
2. <button class="btn-primary">
3.   <slot></slot>
4. </button>
```

当组件渲染的时候，将会被替换为“Add Todo”。

```
1. <!-- 渲染 HTML -->
2. <button class="btn-primary">
3.   Add todo
4. </button>
```

不过，字符串只是开始！插槽还可以包含任何模板代码，包括 HTML：

```
1. <todo-button>
2.   <!-- 添加一个Font Awesome 图标 -->
3.   <i class="fas fa-plus"></i>
4.   Add todo
5. </todo-button>
```

或其他组件


```
1. <todo-button>
2.     <!-- 添加一个图标的组件 -->
3.     <font-awesome-icon name="plus"></font-awesome-icon>
4.     Add todo
5. </todo-button>
```

如果 `<todo-button>` 的 `template` 中没有包含一个 `<slot>` 元素，则该组件起始标签和结束标签之间的任何内容都会被抛弃

```
1. <!-- todo-button 组件模板 -->
2.
3. <button class="btn-primary">
4.     Create a new item
5. </button>
```

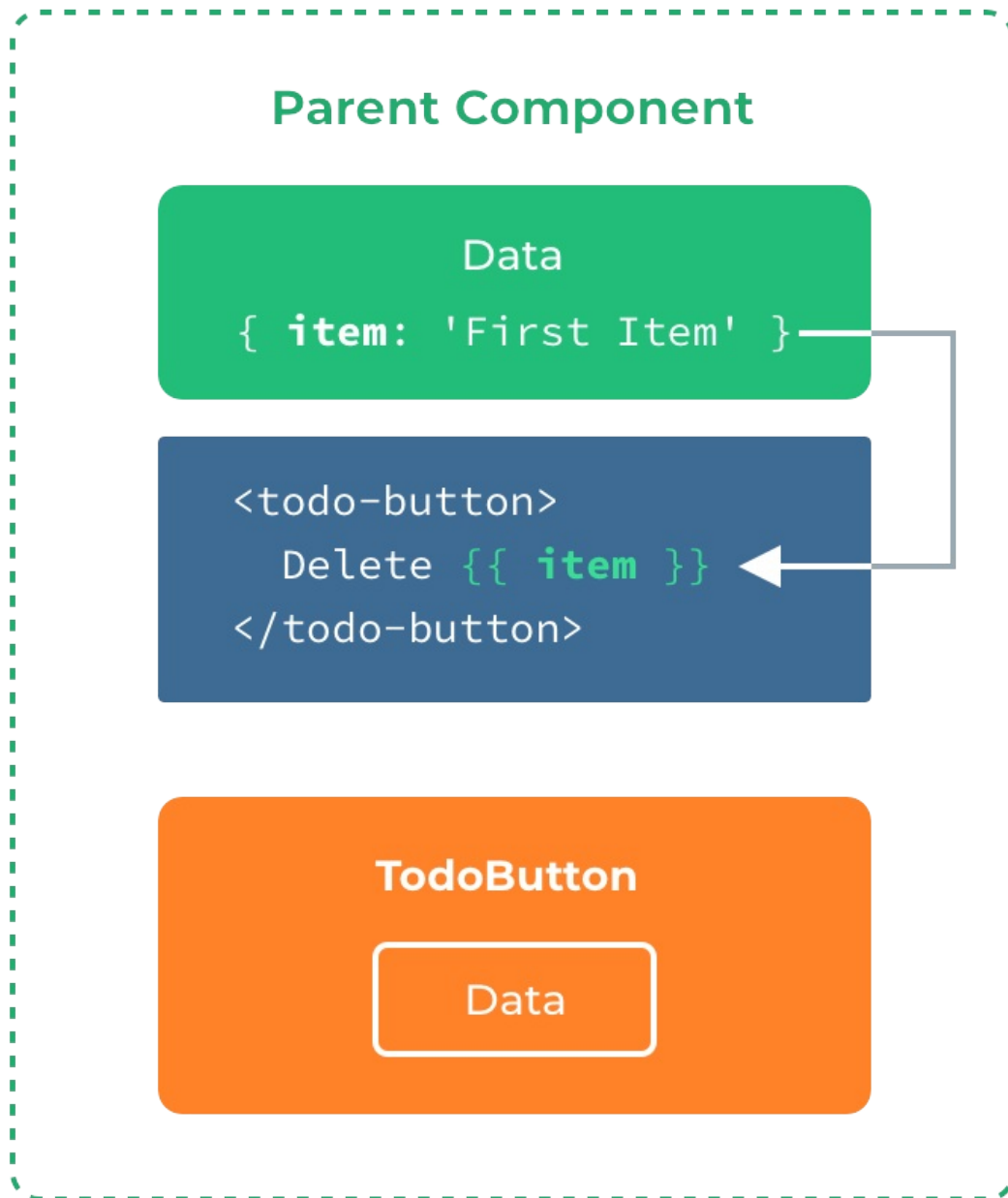
```
1. <todo-button>
2.     <!-- 以下文本不会渲染 -->
3.     Add todo
4. </todo-button>
```

渲染作用域

当你想在一个插槽中使用数据时，例如：

```
1. <todo-button>
2.     Delete a {{ item.name }}
3. </todo-button>
```

该插槽可以访问与模板其余部分相同的实例 `property`（即相同的“作用域”）。



插槽不能访问 `<todo-button>` 的作用域。例如，尝试访问 `action` 将不起作用：

1. `<todo-button action="delete">`
2. Clicking here will `{{ action }}` an item
3. `<!-- `action` 未被定义，因为它的内容是传递*到* <todo-button>，而不是*在* <todo-button>里定义的。 -->`
4. `</todo-button>`

作为一条规则，请记住：

父级模板里的所有内容都是在父级作用域中编译的；子模板里的所有内容都是在子作用域中编译的。

后备内容

有时为一个插槽设置具体的后备（也就是默认的）内容是很有用的，它只会在没有提供内容的时候被渲染。例如在一个 `<submit-button>` 组件中：

```
1. <button type="submit">
2.   <slot></slot>
3. </button>
```

我们可能希望这个 `<button>` 内绝大多数情况下都渲染文本“Submit”。为了将“Submit”作为后备内容，我们可以将它放在 `<slot>` 标签内：

```
1. <button type="submit">
2.   <slot>Submit</slot>
3. </button>
```

现在当我在一个父级组件中使用 `<submit-button >` 并且不提供任何插槽内容时：

```
1. <submit-button></submit-button>
```

后备内容“Submit”将会被渲染：

```
1. <button type="submit">
2.   Submit
3. </button>
```

但是如果我们提供内容：

```
1. <submit-button>
2.   Save
3. </submit-button>
```

则这个提供的内容将会被渲染从而取代后备内容：

```
1. <button type="submit">
2.   Save
3. </button>
```

具名插槽

有时我们需要多个插槽。例如对于一个带有如下模板的 `<base-layout>` 组件：

```
1. <div class="container">
2.   <header>
3.     <!-- 我们希望把页头放这里 -->
4.   </header>
5.   <main>
6.     <!-- 我们希望把主要内容放这里 -->
7.   </main>
8.   <footer>
9.     <!-- 我们希望把页脚放这里 -->
10.  </footer>
11. </div>
```

对于这样的情况，`<slot>` 元素有一个特殊的 attribute: `name`。这个 attribute 可以用来定义额外的插槽：

```
1. <div class="container">
2.   <header>
3.     <slot name="header"></slot>
4.   </header>
5.   <main>
6.     <slot></slot>
7.   </main>
8.   <footer>
9.     <slot name="footer"></slot>
10.  </footer>
11. </div>
```

一个不带 `name` 的 `<slot>` 出口会带有隐含的名字“default”。

在向具名插槽提供内容的时候，我们可以在一个 `<template>` 元素上使用 `v-slot` 指令，并以 `v-slot` 的参数形式提供其名称：

```
1. <base-layout>
2.   <template v-slot:header>
3.     <h1>Here might be a page title</h1>
4.   </template>
5.
```

```
6.   <template v-slot:default>
7.     <p>A paragraph for the main content.</p>
8.     <p>And another one.</p>
9.   </template>
10.
11.  <template v-slot:footer>
12.    <p>Here's some contact info</p>
13.  </template>
14. </base-layout>
```

现在 `<template>` 元素中的所有内容都将会被传入相应的插槽。

渲染的 HTML 将会是：

```
1. <div class="container">
2.   <header>
3.     <h1>Here might be a page title</h1>
4.   </header>
5.   <main>
6.     <p>A paragraph for the main content.</p>
7.     <p>And another one.</p>
8.   </main>
9.   <footer>
10.    <p>Here's some contact info</p>
11.  </footer>
12. </div>
```

注意，`v-slot` 只能添加在 `<template>` 上（只有一种例外情况）

作用域插槽

有时让插槽内容能够访问子组件中才有的数据是很有用的。当一个组件被用来渲染一个项目数组时，这是一个常见的情况，我们希望能够自定义每个项目的渲染方式。

例如，我们有一个组件，包含 `todo-items` 的列表。

```
1. app.component('todo-list', {
2.   data() {
3.     return {
4.       items: ['Feed a cat', 'Buy milk']
5.     }
6.   },
```

```

7.   template: `
8.     <ul>
9.       <li v-for="(item, index) in items">
10.        {{ item }}
11.      </li>
12.    </ul>
13.  `
14. })

```

我们可能需要替换插槽以在父组件上自定义它：

```

1. <todo-list>
2.   <i class="fas fa-check"></i>
3.   <span class="green">{{ item }}</span>
4. </todo-list>

```

但是，这是行不通的，因为只有 `<todo-list>` 组件可以访问 `item`，我们将从其父组件提供槽内容。

要使 `item` 可用于父级提供的 slot 内容，我们可以添加一个 `<slot>` 元素并将其绑定为属性：

```

1. <ul>
2.   <li v-for="( item, index ) in items">
3.     <slot :item="item"></slot>
4.   </li>
5. </ul>

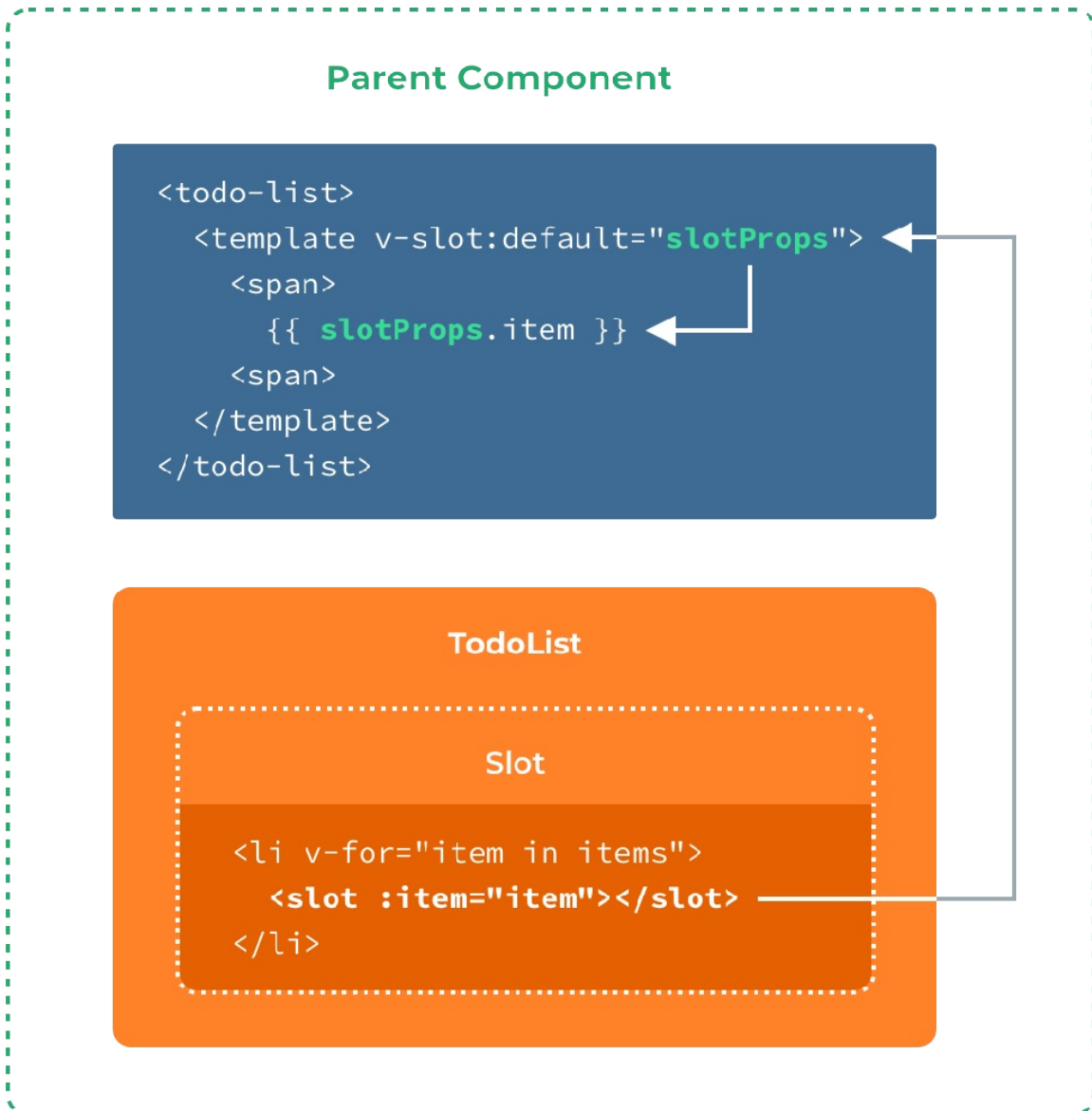
```

绑定在 `<slot>` 元素上的 attribute 被称为插槽 **prop**。现在在父级作用域中，我们可以使用带值的 `v-slot` 来定义我们提供的插槽 prop 的名字：

```

1. <todo-list>
2.   <template v-slot:default="slotProps">
3.     <i class="fas fa-check"></i>
4.     <span class="green">{{ slotProps.item }}</span>
5.   </template>
6. </todo-list>

```



在这个例子中，我们选择将包含所有插槽 prop 的对象命名为 `slotProps`，但你也可以使用任何你喜欢的名字。

独占默认插槽的缩写语法

在上述情况下，当被提供的内容只有默认插槽时，组件的标签才可以被当作插槽的模板来使用。这样我们就可以把 `v-slot` 直接用在组件上：

1. `<todo-list v-slot:default="slotProps">`
2. `<i class="fas fa-check"></i>`
3. `{{ slotProps.item }}`
4. `</todo-list>`

这种写法还可以更简单。就像假定未指定的内容对应默认插槽一样，不带参数的 `v-slot` 被假定对应默认插槽：

```
1. <todo-list v-slot="slotProps">
2.   <i class="fas fa-check"></i>
3.   <span class="green">{{ slotProps.item }}</span>
4. </todo-list>
```

注意默认插槽的缩写语法不能和具名插槽混用，因为它会导致作用域不明确：

```
1. <!-- 无效, 会导致警告 -->
2. <todo-list v-slot="slotProps">
3.   <todo-list v-slot:default="slotProps">
4.     <i class="fas fa-check"></i>
5.     <span class="green">{{ slotProps.item }}</span>
6.   </todo-list>
7.   <template v-slot:other="otherSlotProps">
8.     slotProps is NOT available here
9.   </template>
10. </todo-list>
```

只要出现多个插槽，请始终为所有的插槽使用完整的基于 `<template>` 的语法：

```
1. <todo-list>
2.   <template v-slot:default="slotProps">
3.     <i class="fas fa-check"></i>
4.     <span class="green">{{ slotProps.item }}</span>
5.   </template>
6.
7.   <template v-slot:other="otherSlotProps">
8.     ...
9.   </template>
10. </todo-list>
```

解构插槽 Prop

作用域插槽的内部工作原理是将你的插槽内容包括在一个传入单个参数的函数里：

```
1. function (slotProps) {
2.   // ... 插槽内容 ...
```



```
3. }
```

这意味着 `v-slot` 的值实际上可以是任何能够作为函数定义中的参数的 JavaScript 表达式。你也可以使用 [ES2015](#) ^(opens new window) 解构来传入具体的插槽 prop，如下：

```
1. <todo-list v-slot="{ item }">
2.   <i class="fas fa-check"></i>
3.   <span class="green">{{ item }}</span>
4. </todo-list>
```

这样可以使模板更简洁，尤其是在该插槽提供了多个 prop 的时候。它同样开启了 prop 重命名等其它可能，例如将 `item` 重命名为 `todo`：

```
1. <todo-list v-slot="{ item: todo }">
2.   <i class="fas fa-check"></i>
3.   <span class="green">{{ todo }}</span>
4. </todo-list>
```

你甚至可以定义后备内容，用于插槽 prop 是 undefined 的情形：

```
1. <todo-list v-slot="{ item = 'Placeholder' }">
2.   <i class="fas fa-check"></i>
3.   <span class="green">{{ item }}</span>
4. </todo-list>
```

动态插槽名

[动态指令参数](#) 也可以用在 `v-slot` 上，来定义动态的插槽名：

```
1. <base-layout>
2.   <template v-slot:[dynamicSlotName]>
3.     ...
4.   </template>
5. </base-layout>
```

具名插槽的缩写

跟 `v-on` 和 `v-bind` 一样，`v-slot` 也有缩写，即把参数之前的所有内容（`v-slot:`）替换为字符 `#`。例如 `v-slot:header` 可以被重写为 `#header`：

```
1. <base-layout>
2.   <template #header>
3.     <h1>Here might be a page title</h1>
4.   </template>
5.
6.   <template #default>
7.     <p>A paragraph for the main content.</p>
8.     <p>And another one.</p>
9.   </template>
10.
11.  <template #footer>
12.    <p>Here's some contact info</p>
13.  </template>
14. </base-layout>
```

然而，和其它指令一样，该缩写只在具有参数的时候才可用。这意味着以下语法是无效的：

```
1. <!-- This will trigger a warning -->
2.
3. <todo-list #="{ item }">
4.   <i class="fas fa-check"></i>
5.   <span class="green">{{ item }}</span>
6. </todo-list>
```

如果你希望使用缩写的话，你必须始终以明确插槽名取而代之：

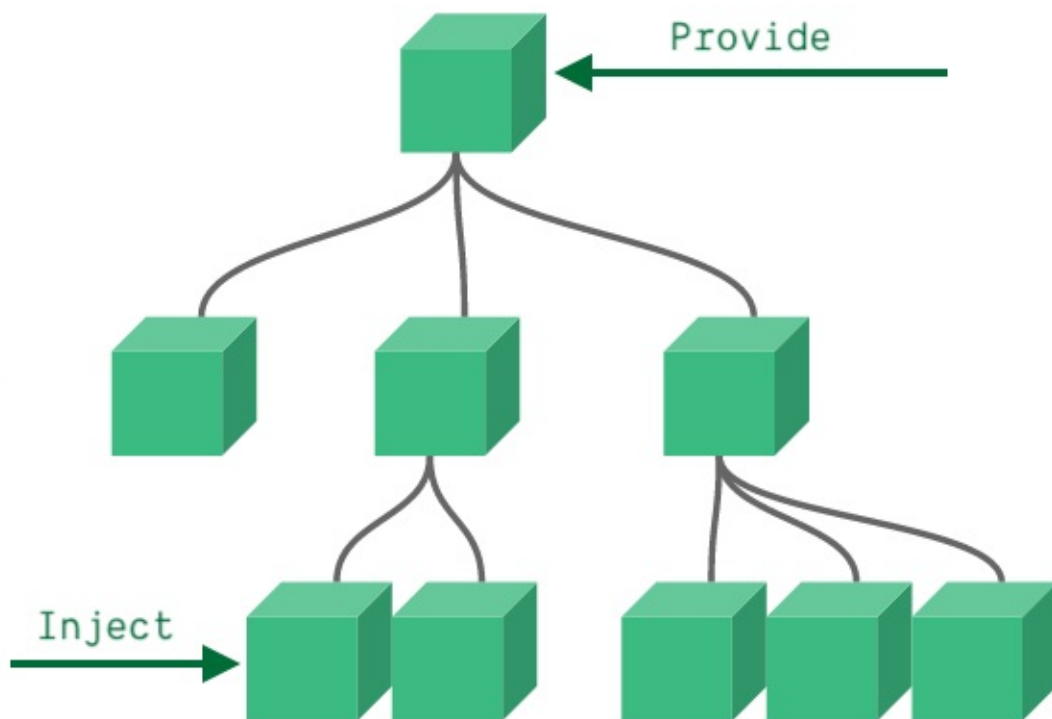
```
1. <todo-list #default="{ item }">
2.   <i class="fas fa-check"></i>
3.   <span class="green">{{ item }}</span>
4. </todo-list>
```

提供 / 注入

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

通常，当我们需要将数据从父组件传递到子组件时，我们使用 `props`。想象一下这样的结构：你有一些深嵌套的组件，而你只需要来自深嵌套子组件中父组件的某些内容。在这种情况下，你仍然需要将 `prop` 传递到整个组件链中，这可能会很烦人。

对于这种情况，我们可以使用 `provide` 和 `inject` 对。父组件可以作为其所有子组件的依赖项提供程序，而不管组件层次结构有多深。这个特性有两个部分：父组件有一个 `provide` 选项来提供数据，子组件有一个 `inject` 选项来开始使用这个数据。



例如，如果我们有这样的层次结构：

1. `Root`
2. └ `TodoList`
3. ├ `TodoItem`
4. └ `TodoListFooter`
5. ├ `ClearTodosButton`
6. └ `TodoListStatistics`

如果要将 `todo-items` 的长度直接传递给 `TodoListStatistics`，我们将把这个属性向下传递到层次结构：`TodoList` -> `TodoListFooter` -> `TodoListStatistics`。通过 `provide/inject` 方法，我们可以直接执行以下操作：

```
1. const app = Vue.createApp({})
2.
3. app.component('todo-list', {
4.   data() {
5.     return {
6.       todos: ['Feed a cat', 'Buy tickets']
7.     }
8.   },
9.   provide: {
10.    user: 'John Doe'
11.  },
12.  template: `
13.    <div>
14.      {{ todos.length }}
15.      <!-- 模板的其余部分 -->
16.    </div>
17.  `
18. })
19.
20. app.component('todo-list-statistics', {
21.   inject: ['user'],
22.   created() {
23.     console.log(`Injected property: ${this.user}`) // > 注入 property: John Doe
24.   }
25. })
```

但是，如果我们尝试在此处提供一些组件实例 `property`，则这将不起作用：

```
1. app.component('todo-list', {
2.   data() {
3.     return {
4.       todos: ['Feed a cat', 'Buy tickets']
5.     }
6.   },
7.   provide: {
8.     todoLength: this.todos.length // 将会导致错误 'Cannot read property 'length'
9.     of undefined`
10.  },
11. })
```

```

10.   template: `
11.     ...
12.   `
13. })

```

要访问组件实例 `property`，我们需要将 `provide` 转换为返回对象的函数

```

1.  app.component('todo-list', {
2.    data() {
3.      return {
4.        todos: ['Feed a cat', 'Buy tickets']
5.      }
6.    },
7.    provide() {
8.      return {
9.        todoLength: this.todos.length
10.     }
11.   },
12.   template: `
13.     ...
14.   `
15. })

```

这使我们能够更安全地继续开发该组件，而不必担心可能会更改/删除子组件所依赖的某些内容。这些组件之间的接口仍然是明确定义的，就像 `prop` 一样。

实际上，你可以将依赖注入看作是“long range props”，除了：

- 父组件不需要知道哪些子组件使用它提供的 `property`
- 子组件不需要知道 `inject` `property` 来自哪里

与响应式一起工作

在上面的例子中，如果我们更改了 `todos` 的列表，这个更改将不会反映在注入的 `todoLength` `property` 中。这是因为默认情况下，`provide/inject` 绑定不是被动绑定。我们可以通过将 `ref` `property` 或 `reactive` 对象传递给 `provide` 来更改此行为。在我们的例子中，如果我们想对祖先组件中的更改做出反应，我们需要为我们提供的 `todoLength` 分配一个组合 API `computed` `property`：

```

1.  app.component('todo-list', {
2.    // ...

```

```
3.   provide() {
4.     return {
5.       todoLength: Vue.computed(() => this.todos.length)
6.     }
7.   }
8. })
```

在这种情况下，对 `todos.length` 将正确反映在组件中，其中“todoLength”被注入。在 [Composition API 部分](#) 中阅读关于 `reactive` `provide/inject` 的更多信息。

动态组件 & 异步组件

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

在动态组件上使用 `keep-alive`

我们之前曾经在一个多标签的界面中使用 `is` attribute 来切换不同的组件：

```
1. <component :is="currentTabComponent"></component>
```

当在这些组件之间切换的时候，你有时会想保持这些组件的状态，以避免反复重渲染导致的性能问题。例如我们来展开说一说这个多标签界面：

你会注意到，如果你选择了一篇文章，切换到 *Archive* 标签，然后再切换回 *Posts*，是不会继续展示你之前选择的文章的。这是因为你每次切换新标签的时候，Vue 都创建了一个新的 `currentTabComponent` 实例。

重新创建动态组件的行为通常是非常有用的，但是在这个案例中，我们更希望那些标签的组件实例能够在它们第一次被创建的时候缓存下来。为了解决这个问题，我们可以用一个 `<keep-alive>` 元素将其动态组件包裹起来。

```
1. <!-- 失活的组件将会被缓存！ -->
2. <keep-alive>
3.   <component :is="currentTabComponent"></component>
4. </keep-alive>
```

来看看修改后的结果：

现在这个 *Posts* 标签保持了它的状态（被选中的文章）甚至当它未被渲染时也是如此。你可以在这个示例查阅到完整的代码。

你可以在 [API 参考文档](#) 查阅更多关于 `<keep-alive>` 的细节。

异步组件

在大型应用中，我们可能需要将应用分割成小一些的代码块，并且只在需要的时候才从服务器加载一个模块。为了简化，Vue 有一个 `defineAsyncComponent` 方法：

```
1. const app = Vue.createApp({})
```

```

2.
3. const AsyncComp = Vue.defineAsyncComponent(
4.   () =>
5.     new Promise((resolve, reject) => {
6.       resolve({
7.         template: '<div>I am async!</div>'
8.       })
9.     })
10. )
11.
12. app.component('async-example', AsyncComp)

```

如你所见，此方法接受返回 `Promise` 的工厂函数。从服务器检索组件定义后，应调用 `Promise` 的 `resolve` 回调。你也可以调用 `reject(reason)`，以指示加载失败。

你也可以在工厂函数中返回一个 `Promise`，所以把 `webpack 2` 和 `ES2015` 语法加在一起，我们可以这样使用动态导入：

```

1. import { defineAsyncComponent } from 'vue'
2.
3. const AsyncComp = defineAsyncComponent(() =>
4.   import('./components/AsyncComponent.vue')
5. )
6.
7. app.component('async-component', AsyncComp)

```

当在本地注册组件时，你也可以使用 `defineAsyncComponent`

```

1. import { createApp, defineAsyncComponent } from 'vue'
2.
3. createApp({
4.   // ...
5.   components: {
6.     AsyncComponent: defineAsyncComponent(() =>
7.       import('./components/AsyncComponent.vue')
8.     )
9.   }
10. })

```

与 Suspense 一起使用

异步组件在默认情况下是可挂起的。这意味着如果它在父链中有一个 `<Suspense>`，它将被视为该

`<Suspense>` 的异步依赖。在这种情况下，加载状态将由 `<Suspense>` 控制，组件自身的加载、错误、延迟和超时选项将被忽略。

异步组件可以选择退出 `Suspense` 控制，并通过在其选项中指定 `suspensible:false`，让组件始终控制自己的加载状态。

你可以在[中查看可用选项的列表](#) [API Reference](#)

模板引用

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

尽管存在 `prop` 和事件，但有时你可能仍然需要直接访问 JavaScript 中的子组件。为此，可以使用 `ref` attribute 为子组件或 HTML 元素指定引用 ID。例如：

```
1. <input ref="input" />
```

例如，你希望以编程的方式 `focus` 这个 `input` 在组件上挂载，这可能有用

```
1. const app = Vue.createApp({})
2.
3. app.component('base-input', {
4.   template: `
5.     <input ref="input" />
6.   `,
7.   methods: {
8.     focusInput() {
9.       this.$refs.input.focus()
10.    }
11.  },
12.  mounted() {
13.    this.focusInput()
14.  }
15. })
```

此外，还可以向组件本身添加另一个 `ref`，并使用它从父组件触发 `focusInput` 事件：

```
1. <base-input ref="usernameInput"></base-input>
```

```
1. this.$refs.usernameInput.focusInput()
```

当 `ref` 与 `v-for` 一起使用时，你得到的 `ref` 将是一个数组，其中包含镜像数据源的子组件。

WARNING

`$refs` 只会在组件渲染完成之后生效。这仅作为一个用于直接操作子元素的“逃生舱”——你应该避免在模板或计算属性中访问 `$refs`。

模板引用

参考: [在 Composition API 中使用 template refs](#)

处理边界情况

该页面假设你已经阅读过了[组件基础](#)。如果你还对组件不太了解，推荐你先阅读它。

提示

这里记录的都是和处理边界情况有关的功能，即一些需要对 Vue 的规则做一些小调整的特殊情况。不过注意这些功能都是有劣势或危险的场景的。我们会在每个案例中注明，所以当你使用每个功能的时候请稍加留意。

控制更新

得益于其响应式系统，Vue 总是知道何时更新（如果你使用正确的话）。但是，在某些边缘情况下，你可能希望强制更新，尽管事实上没有任何反应性数据发生更改。还有一些情况下，你可能希望防止不必要的更新。

强制更新

如果你发现自己需要在 Vue 中强制更新，在 99.99%的情况下，你在某个地方犯了错误。例如，你可能依赖于 Vue 反应性系统未跟踪的状态，例如，在组件创建之后添加了 `data` 属性。

但是，如果你已经排除了上述情况，并且发现自己处于这种非常罕见的情况下，必须手动强制更新，那么你可以使用 `$forceUpdate`。

低级静态组件与 `v-once`

在 Vue 中渲染纯 HTML 元素的速度非常快，但有时你可能有一个包含很多静态内容的组件。在这些情况下，可以通过向根元素添加 `v-once` 指令来确保只对其求值一次，然后进行缓存，如下所示：

```
1. app.component('terms-of-service', {
2.   template: `
3.     <div v-once>
4.       <h1>Terms of Service</h1>
5.       ... a lot of static content ...
6.     </div>
7.   `
8. })
```

TIP

再次提醒，不要过度使用这种模式。虽然在极少数情况下需要渲染大量静态内容时很方便，但除非你注

意到渲染速度—慢，否则就没有必要这样做—另外，这可能会在以后引起很多混乱。例如，假设另一个开发人员不熟悉 `v-once` 或者只是在模板中遗漏了它。他们可能会花上几个小时来弄清楚为什么模板没有正确更新。

过渡

- [过渡 & 动画概述](#)
- [进入过渡 & 离开过渡](#)
- [列表过渡](#)
- [状态过渡](#)

过渡 & 动画概述

Vue 提供了一些抽象概念，可以帮助处理过渡和动画，特别是在响应某些变化时。这些抽象的概念包括：

- 在 CSS 和 JS 中，使用内置的 `<transition>` 组件来钩住组件中进入和离开 DOM。
- 过渡模式，以便你在过渡期间编排顺序。
- 在处理多个元素位置更新时，使用 `<transition-group>` 组件，通过 FLIP 技术来提高性能。
- 使用 `watchers` 来处理应用中不同状态的过渡。

我们将在本指南接下来的三个部分中介绍所有这些以及更多内容。然而，除了提供这些有用的 API 之外，值得一提的是，我们前面介绍的 `class` 和 `style` 声明也可以应用于动画和过渡，用于更简单的用例。

在下一节中，我们将回顾一些 web 动画和过渡的基础知识，并提供一些资源链接以进行进一步的研究。如果你已经熟悉 web 动画，并且了解这些原理如何与 Vue 的某些指令配合使用，可以跳过这一节。对于希望在开始学习之前进一步了解网络动画基础知识的其他人，请继续阅读。

基于 class 的动画和过渡

尽管 `<transition>` 组件对于组件的进入和离开非常有用，但你也可以通过添加一个条件 `class` 来激活动画，而无需挂载组件。

```
1. <div id="demo">
2.   Push this button to do something you shouldn't be doing:<br />
3.
4.   <div :class="{ shake: noActivated }">
5.     <button @click="noActivated = true">Click me</button>
6.     <span v-if="noActivated">Oh no!</span>
7.   </div>
8. </div>
```

```
1. .shake {
2.   animation: shake 0.82s cubic-bezier(0.36, 0.07, 0.19, 0.97) both;
3.   transform: translate3d(0, 0, 0);
4.   backface-visibility: hidden;
5.   perspective: 1000px;
6. }
7.
```

```
8. @keyframes shake {
9.   10%,
10.  90% {
11.    transform: translate3d(-1px, 0, 0);
12.  }
13.
14.  20%,
15.  80% {
16.    transform: translate3d(2px, 0, 0);
17.  }
18.
19.  30%,
20.  50%,
21.  70% {
22.    transform: translate3d(-4px, 0, 0);
23.  }
24.
25.  40%,
26.  60% {
27.    transform: translate3d(4px, 0, 0);
28.  }
29. }
```

```
1. const Demo = {
2.   data() {
3.     return {
4.       noActivated: false
5.     }
6.   }
7. }
8.
9. Vue.createApp(Demo).mount('#demo')
```

过渡与 Style 绑定

一些过渡效果可以通过插值的方式来实现，例如在发生交互时将样式绑定到元素上。以这个例子为例：

```
1. <div id="demo">
2.   <div
3.     @mousemove="xCoordinate"
4.     :style="{ backgroundColor: `hsl(${x}, 80%, 50%)` }">
```



```

5.     class="movearea"
6.   >
7.     <h3>Move your mouse across the screen...</h3>
8.     <p>x: {{x}}</p>
9.   </div>
10. </div>

```

```

1. .movearea {
2.   transition: 0.2s background-color ease;
3. }

```

```

1. const Demo = {
2.   data() {
3.     return {
4.       x: 0
5.     }
6.   },
7.   methods: {
8.     xCoordinate(e) {
9.       this.x = e.clientX
10.    }
11.  }
12. }
13.
14. Vue.createApp(Demo).mount('#demo')

```

在这个例子中，我们是通过使用插值来创建动画，将触发条件添加到鼠标的移动过程上。同时将 CSS 过渡属性应用在元素上，让元素知道在更新时要使用什么过渡效果。

性能

你可能注意到上面显示的动画使用了 `transforms` 之类的东西，并应用了诸如 `perspective` 之类的奇怪的 `property`—为什么它们是这样实现的，而不是仅仅使用 `margin` 和 `top` 等？

我们可以通过对性能的了解，在 web 上创建极其流畅的动画。我们希望尽可能对元素动画进行硬件加速，并使用不触发重绘的 `property`。我们来介绍一下如何实现这个目标。

Transform 和 Opacity

我们可以通过工具，例如 [CSS Triggers](#) [↗] (`opens new window`) 来查看哪些属性会在动画时触

发重绘。在工具中，查看 `transform` 的相关内容，你将看到：

非常好的是，更改 `transform` 不会触发任何几何形状变化或绘制。这意味着该操作可能是由合成器线程在 GPU 的帮助下执行。

`opacity` 属性的行为也类似。因此，他们是在 web 上做元素移动的理想选择。

硬件加速

诸如 `perspective`、`backface-visibility` 和 `transform:translateZ(x)` 等 property 将让浏览器知道你需要硬件加速。

如果要对一个元素进行硬件加速，可以应用以下任何一个 property（并不是需要全部，任意一个就可以）：

1. `perspective: 1000px;`
2. `backface-visibility: hidden;`
3. `transform: translateZ(0);`

许多像 GreenSock 这样的 JS 库都会默认你需要硬件加速，并在默认情况下应用，所以你不需要手动设置它们。

Timing

对于简单 UI 过渡，即从一个状态到另一个没有中间状态的状态，通常使用 0.1s 到 0.4s 之间的计时，大多数人发现 0.25s 是一个最佳选择。你能用这个定时做任何事情吗？并不是。如果你有一些元素需要移动更大的距离，或者有更多的步骤或状态变化，0.25s 并不会有很好的效果，你将不得不有更多的目的性，而且定时也需要更加独特。但这并不意味着你不能在应用中重复使用效果好的默认值。

你也可能会发现，起始动画比结束动画的时间稍长一些，看起来会更好一些。用户通常是在动画开始时被引导的，而在动画结束时没有那么多耐心，因为他们想继续他们的动作。

Easing

Easing 是在动画中表达深度的一个重要方式。动画新手最常犯的一个错误是在起始动画节点使用 `ease-in`，在结束动画节点使用 `ease-out`。实际上你需要的是反过来的。

如果我们将这些状态应用于过渡，它应该像这样：

1. `.button {`
2. `background: #1b8f5a;`

```

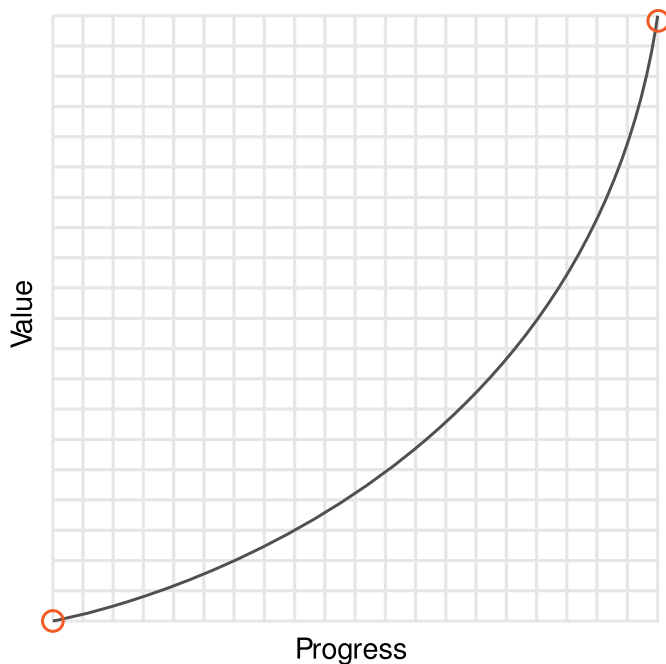
3.  /* 应用于初始状态, 因此此转换将应用于返回状态 */
4.  transition: background 0.25s ease-in;
5.  }
6.
7.  .button:hover {
8.    background: #3eaf7c;
9.    /* 应用于悬停状态, 因此在触发悬停时将应用此过渡 */
10.   transition: background 0.35s ease-out;
11.  }

```

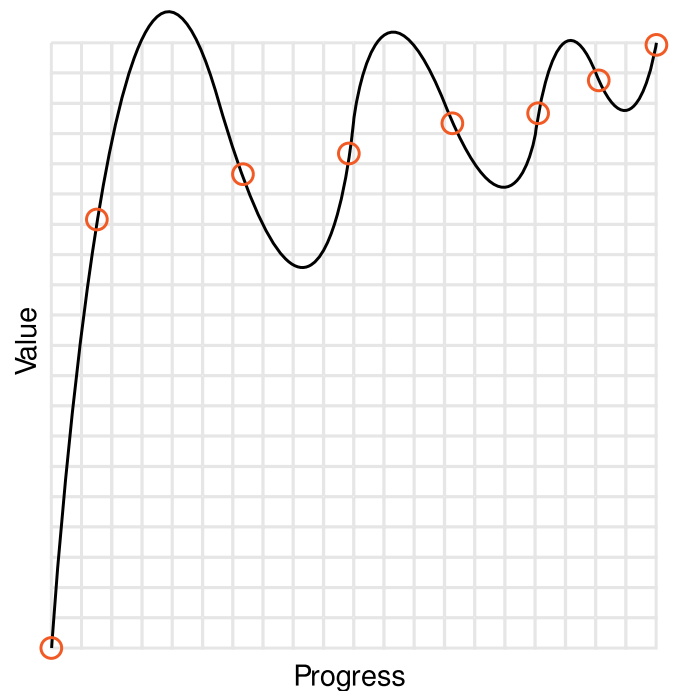
Easing 也可以表达动画元素的质量。以下面的 Pen 为例, 你认为哪个球是硬的, 哪个球是软的?

你可以通过调整你的 Easing 来获得很多独特的效果, 使你的动画非常时尚。CSS 允许你通过调整 cubic bezier 属性来修改 Easing, Lea Verou 开发的[这个 playground](#) (opens new window) 对探索这个问题非常有帮助。

虽然使用 cubic-bezier ease 提供的两个控制柄可以为简单的动画获得很好的效果, 但是 JavaScript 允许多个控制柄, 以此支持更多的变化。



CSS - limited handles



JS - multiple handles


以弹跳为例。在 CSS 中, 我们必须声明向上和向下的每个关键帧。在 JavaScript 中, 我们可以通过 [greensock API \(GSAP\)](#) (opens new window) 中声明 `bounce` 来描述 ease 中所有这些移动 (其他 JS 库有其他类型的 easing 默认值)。

这里是 CSS 中用来实现 bounce 的代码 (来自 animate.css 的例子):

```
1. @keyframes bounceInDown {
2.   from,
3.   60%,
4.   75%,
5.   90%,
6.   to {
7.     animation-timing-function: cubic-bezier(0.215, 0.61, 0.355, 1);
8.   }
9.
10.  0% {
11.    opacity: 0;
12.    transform: translate3d(0, -3000px, 0) scaleY(3);
13.  }
14.
15.  60% {
16.    opacity: 1;
17.    transform: translate3d(0, 25px, 0) scaleY(0.9);
18.  }
19.
20.  75% {
21.    transform: translate3d(0, -10px, 0) scaleY(0.95);
22.  }
23.
24.  90% {
25.    transform: translate3d(0, 5px, 0) scaleY(0.985);
26.  }
27.
28.  to {
29.    transform: translate3d(0, 0, 0);
30.  }
31. }
32.
33. .bounceInDown {
34.   animation-name: bounceInDown;
35. }
```

下面是 JS 中使用 GreenSock 实现相同的 bounce:

```
1. gsap.from(element, { duration: 1, ease: 'bounce.out', y: -500 })
```

我们将在之后章节的部分示例中使用 GreenSock。他们有一个很棒的 [ease visualizer](#) 

(opens new window), 帮助你建立精心制作的画架。

进一步阅读

- 界面动画设计: 通过 Val Head 动画改善用户体验 [↗] (opens new window)
- Animation at Work 作者: Rachel Nabors [↗] (opens new window)

进入过渡 & 离开过渡

在插入、更新或从 DOM 中移除项时，Vue 提供了多种应用转换效果的方法。这包括以下工具：

- 自动为 CSS 转换和动画应用 class；
- 集成第三方 CSS 动画库，例如 [animate.css](#) [↗] (opens new window) ；
- 在过渡钩子期间使用 JavaScript 直接操作 DOM；
- 集成第三方 JavaScript 动画库。

在这里，我们只会讲到进入、离开和列表的过渡，你也可以看下一节的[管理过渡状态](#)。

单元素/组件的过渡

Vue 提供了 `transition` 的封装组件，在下列情形中，可以给任何元素和组件添加进入/离开过渡

- 条件渲染（使用 `v-if`）
- 条件展示（使用 `v-show`）
- 动态组件
- 组件根节点

这里是一个典型的例子：

```
1. <div id="demo">
2.   <button @click="show = !show">
3.     Toggle
4.   </button>
5.
6.   <transition name="fade">
7.     <p v-if="show">hello</p>
8.   </transition>
9. </div>
```

```
1. const Demo = {
2.   data() {
3.     return {
4.       show: true
5.     }
6.   }
7. }
8.
```

```
9. Vue.createApp(Demo).mount('#demo')
```

```
1. .fade-enter-active,
2. .fade-leave-active {
3.   transition: opacity 0.5s ease;
4. }
5.
6. .fade-enter-from,
7. .fade-leave-to {
8.   opacity: 0;
9. }
```

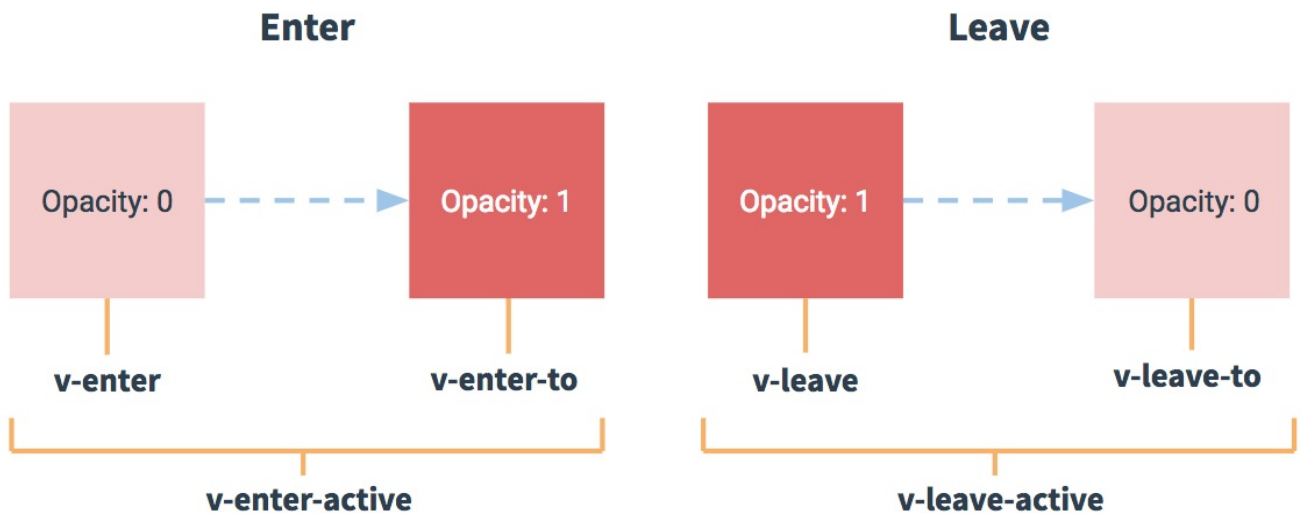
当插入或删除包含在 `transition` 组件中的元素时，Vue 将会做以下处理：

1. 自动嗅探目标元素是否应用了 CSS 过渡或动画，如果是，在恰当的时机添加/删除 CSS 类名。
2. 如果过渡组件提供了 JavaScript 钩子函数，这些钩子函数将在恰当的时机被调用。
3. 如果没有找到 JavaScript 钩子并且也没有检测到 CSS 过渡/动画，DOM 操作（插入/删除）在下一帧中立即执行。（注意：此指浏览器逐帧动画机制，和 Vue 的 `nextTick` 概念不同）

过渡class

在进入/离开的过渡中，会有 6 个 class 切换。

1. `v-enter-from`：定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。
2. `v-enter-active`：定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。
3. `v-enter-to`：定义进入过渡的结束状态。在元素被插入之后下一帧生效（与此同时 `v-enter-from` 被移除），在过渡/动画完成之后移除。
4. `v-leave-from`：定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除。
5. `v-leave-active`：定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数。
6. `v-leave-to`：离开过渡的结束状态。在离开过渡被触发之后下一帧生效（与此同时 `v-leave-from` 被删除），在过渡/动画完成之后移除。



对于这些在过渡中切换的类名来说，如果你使用一个没有名字的 `<transition>`，则 `v-` 是这些class名的默认前缀。如果你使用了 `<transition name="my-transition">`，那么 `v-enter-from` 会替换为 `my-transition-enter-from`。

`v-enter-active` 和 `v-leave-active` 可以控制进入/离开过渡的不同的缓和曲线，在下面章节会有个示例说明。

CSS 过渡

常用的过渡都是使用 CSS 过渡。

```

1. <div id="demo">
2.   <button @click="show = !show">
3.     Toggle render
4.   </button>
5.
6.   <transition name="slide-fade">
7.     <p v-if="show">hello</p>
8.   </transition>
9. </div>

```

```

1. const Demo = {
2.   data() {
3.     return {
4.       show: true
5.     }

```



```

6.   }
7. }
8.
9. Vue.createApp(Demo).mount('#demo')

```

```

1.  /* 可以设置不同的进入和离开动画  */
2.  /* 设置持续时间和动画函数  */
3.  .slide-fade-enter-active {
4.    transition: all 0.3s ease-out;
5.  }
6.
7.  .slide-fade-leave-active {
8.    transition: all 0.8s cubic-bezier(1, 0.5, 0.8, 1);
9.  }
10.
11. .slide-fade-enter-from,
12. .slide-fade-leave-to {
13.   transform: translateX(20px);
14.   opacity: 0;
15. }

```

CSS 动画

CSS 动画用法同 CSS 过渡，区别是在动画中 `v-enter-from` 类名在节点插入 DOM 后不会立即删除，而是在 `animationend` 事件触发时删除。

下面是一个例子，为了简洁起见，省略了带前缀的 CSS 规则：

```

1. <div id="demo">
2.   <button @click="show = !show">Toggle show</button>
3.   <transition name="bounce">
4.     <p v-if="show">
5.       Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris facilisis
6.       enim libero, at lacinia diam fermentum id. Pellentesque habitant morbi
7.       tristique senectus et netus.
8.     </p>
9.   </transition>
10. </div>

```

```

1. const Demo = {
2.   data() {

```

```
3.     return {
4.       show: true
5.     }
6.   }
7. }
8.
9. Vue.createApp(Demo).mount('#demo')
```

```
1. .bounce-enter-active {
2.   animation: bounce-in 0.5s;
3. }
4. .bounce-leave-active {
5.   animation: bounce-in 0.5s reverse;
6. }
7. @keyframes bounce-in {
8.   0% {
9.     transform: scale(0);
10.  }
11.  50% {
12.    transform: scale(1.25);
13.  }
14.  100% {
15.    transform: scale(1);
16.  }
17. }
```

自定义过渡 class 类名

我们可以通过以下 attribute 来自定义过渡类名：

- `enter-from-class`
- `enter-active-class`
- `enter-to-class`
- `leave-from-class`
- `leave-active-class`
- `leave-to-class`

他们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库，如

[Animate.css](#) [↗] (opens new window). 结合使用十分有用。

示例：

```

1. <link
2.   href="https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.1.0/animate.min.css"
3.   rel="stylesheet"
4.   type="text/css"
5. />
6.
7. <div id="demo">
8.   <button @click="show = !show">
9.     Toggle render
10.  </button>
11.
12.  <transition
13.    name="custom-classes-transition"
14.    enter-active-class="animate__animated animate__tada"
15.    leave-active-class="animate__animated animate__bounceOutRight"
16.  >
17.    <p v-if="show">hello</p>
18.  </transition>
19. </div>

```

```

1. const Demo = {
2.   data() {
3.     return {
4.       show: true
5.     }
6.   }
7. }
8.
9. Vue.createApp(Demo).mount('#demo')

```

同时使用过渡和动画

Vue 为了知道过渡的完成，必须设置相应的事件监听器。它可以是 `transitionend` 或 `animationend`，这取决于给元素应用的 CSS 规则。如果你使用其中任何一种，Vue 能自动识别类型并设置监听。

但是，在一些场景中，你需要给同一个元素同时设置两种过渡动效，比如 `animation` 很快的被触发并完成了，而 `transition` 效果还没结束。在这种情况下，你就需要使用 `type` attribute 并设置 `animation` 或 `transition` 来明确声明你需要 Vue 监听的类型。

显性的过渡持续时间

在很多情况下，Vue 可以自动得出过渡效果的完成时机。默认情况下，Vue 会等待其在过渡效果的根元素的第一个 `transitionend` 或 `animationend` 事件。然而也可以不这样设定——比如，我们可以拥有一个精心编排的一系列过渡效果，其中一些嵌套的内部元素相比于过渡效果的根元素有延迟的或更长的过渡效果。

在这种情况下你可以用 `<transition>` 组件上的 `duration` prop 定制一个显性的过渡持续时间（以毫秒计）：

```
1. <transition :duration="1000">...</transition>
```

你也可以定制进入和移出的持续时间：

```
1. <transition :duration="{ enter: 500, leave: 800 }">...</transition>
```

JavaScript 钩子

可以在 attribute 中声明 JavaScript 钩子

```
1. <transition
2.   @before-enter="beforeEnter"
3.   @enter="enter"
4.   @after-enter="afterEnter"
5.   @enter-cancelled="enterCancelled"
6.   @before-leave="beforeLeave"
7.   @leave="leave"
8.   @after-leave="afterLeave"
9.   @leave-cancelled="leaveCancelled"
10.  :css="false"
11. >
12.  <!-- ... -->
13. </transition>
```

```
1. // ...
2. methods: {
3.   // -----
4.   // ENTERING
5.   // -----
6.
7.   beforeEnter(e1) {
```

```
8.     // ...
9.   },
10.  // 当与 CSS 结合使用时
11.  // 回调函数 done 是可选的
12.  enter(e1, done) {
13.    // ...
14.    done()
15.  },
16.  afterEnter(e1) {
17.    // ...
18.  },
19.  enterCancelled(e1) {
20.    // ...
21.  },
22.
23.  // -----
24.  // 离开时
25.  // -----
26.
27.  beforeLeave(e1) {
28.    // ...
29.  },
30.  // 当与 CSS 结合使用时
31.  // 回调函数 done 是可选的
32.  leave(e1, done) {
33.    // ...
34.    done()
35.  },
36.  afterLeave(e1) {
37.    // ...
38.  },
39.  // leaveCancelled 只用于 v-show 中
40.  leaveCancelled(e1) {
41.    // ...
42.  }
43. }
```

这些钩子函数可以结合 CSS transitions/animations 使用，也可以单独使用。

当只用 JavaScript 过渡的时候，在 `enter` 和 `leave` 钩中必须使用 `done` 进行回调。否则，它们将被同步调用，过渡会立即完成。添加 `:css="false"`，也会让 Vue 会跳过 CSS 的检测，除了性能略高之外，这可以避免过渡过程中 CSS 规则的影响。

现在让我们来看一个例子。下面是一个使用 [GreenSock](#) (opens new window) 的 JavaScript 过渡：

```
1. <script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.3.4/gsap.min.js">
2. </script>
3. <div id="demo">
4.   <button @click="show = !show">
5.     Toggle
6.   </button>
7.
8.   <transition
9.     @before-enter="beforeEnter"
10.    @enter="enter"
11.    @leave="leave"
12.    :css="false"
13.  >
14.    <p v-if="show">
15.      Demo
16.    </p>
17.  </transition>
18. </div>
```

```
1. const Demo = {
2.   data() {
3.     return {
4.       show: false
5.     }
6.   },
7.   methods: {
8.     beforeEnter(e1) {
9.       gsap.set(e1, {
10.        scaleX: 0.8,
11.        scaleY: 1.2
12.      })
13.    },
14.    enter(e1, done) {
15.      gsap.to(e1, {
16.        duration: 1,
17.        scaleX: 1.5,
18.        scaleY: 0.7,
```

```
19.     opacity: 1,
20.     x: 150,
21.     ease: 'elastic.inOut(2.5, 1)',
22.     onComplete: done
23.   })
24. },
25.   leave(e1, done) {
26.     gsap.to(e1, {
27.       duration: 0.7,
28.       scaleX: 1,
29.       scaleY: 1,
30.       x: 300,
31.       ease: 'elastic.inOut(2.5, 1)'
32.     })
33.     gsap.to(e1, {
34.       duration: 0.2,
35.       delay: 0.5,
36.       opacity: 0,
37.       onComplete: done
38.     })
39.   }
40. }
41. }
42.
43. Vue.createApp(Demo).mount('#demo')
```

初始渲染的过渡

可以通过 `appear` attribute 设置节点在初始渲染的过渡

```
1. <transition appear>
2.   <!-- ... -->
3. </transition>
```

多个元素的过渡

我们之后讨论 [多个组件的过渡](#)，对于原生标签可以使用 `v-if` / `v-else`。最常见的多标签过渡是一个列表和描述这个列表为空消息的元素：

```
1. <transition>
```

```

2.   <table v-if="items.length > 0">
3.     <!-- ... -->
4.   </table>
5.   <p v-else>Sorry, no items found.</p>
6. </transition>

```

实际上, 通过使用多个 `v-if` 或将单个元素绑定到一个动态 property, 可以在任意数量的元素之间进行过渡。例如:

```

1. <transition>
2.   <button v-if="docState === 'saved'" key="saved">
3.     Edit
4.   </button>
5.   <button v-if="docState === 'edited'" key="edited">
6.     Save
7.   </button>
8.   <button v-if="docState === 'editing'" key="editing">
9.     Cancel
10.  </button>
11. </transition>

```

也可以写为:

```

1. <transition>
2.   <button :key="docState">
3.     {{ buttonMessage }}
4.   </button>
5. </transition>

```

```

1. // ...
2. computed: {
3.   buttonMessage() {
4.     switch (this.docState) {
5.       case 'saved': return 'Edit'
6.       case 'edited': return 'Save'
7.       case 'editing': return 'Cancel'
8.     }
9.   }
10. }

```

过渡模式

这里还有一个问题，试着点击下面的按钮：

在“on”按钮和“off”按钮的过渡中，两个按钮都被重绘了，一个离开过渡的时候另一个开始进入过渡。这是 `<transition>` 的默认行为 — 进入和离开同时发生。

有时这很有效，例如当过渡项绝对位于彼此的 top 时：

同时生效的进入和离开的过渡不能满足所有要求，所以 Vue 提供了过渡模式

- `in-out`：新元素先进行过渡，完成之后当前元素过渡离开。
- `out-in`：当前元素先进行过渡，完成之后新元素过渡进入。

TIP

很快就会发现 `out-in` 是你大多数时候想要的状态 😊

现在让我们用 `out-in` 更新 on/off 按钮的转换：

```
1. <transition name="fade" mode="out-in">
2.   <!-- ... the buttons ... -->
3. </transition>
```

通过添加一个 attribute，我们修复了原来的过渡，而不必添加任何特殊 style。

我们可以用它来协调更具表现力的动作，例如折叠卡片，如下所示。实际上是两个元素在彼此之间转换，但是由于开始状态和结束状态的比例是相同的：水平为0，它看起来就像一个流体运动。这种轻描淡写对于真实的 UI 微交互非常有用：

多个组件之间过渡

组件之间的过渡更简单 — 我们甚至不需要 `key` 属性。相反，我们包装了一个[动态组件](#)：

```
1. <div id="demo">
   <input v-model="view" type="radio" value="v-a" id="a"><label
2.   for="a">A</label>
   <input v-model="view" type="radio" value="v-b" id="b"><label
3.   for="b">B</label>
4.   <transition name="component-fade" mode="out-in">
5.     <component :is="view"></component>
6.   </transition>
7. </div>
```

```
1. const Demo = {
```

```
2.   data() {
3.     return {
4.       view: 'v-a'
5.     }
6.   },
7.   components: {
8.     'v-a': {
9.       template: '<div>Component A</div>'
10.    },
11.    'v-b': {
12.      template: '<div>Component B</div>'
13.    }
14.  }
15. }
16.
17. Vue.createApp(Demo).mount('#demo')
```

```
1. .component-fade-enter-active,
2. .component-fade-leave-active {
3.   transition: opacity 0.3s ease;
4. }
5.
6. .component-fade-enter-from,
7. .component-fade-leave-to {
8.   opacity: 0;
9. }
```

列表过渡

目前为止，关于过渡我们已经讲到：

- 单个节点
- 同一时间渲染多个节点中的一个

那么怎么同时渲染整个列表，比如使用 `v-for` ？在这种场景中，使用 `<transition-group>` 组件。在我们深入例子之前，先了解关于这个组件的几个特点：

- 不同于 `<transition>`，它会以一个真实元素渲染：默认为一个 ``。你也可以通过 `tag` attribute 更换为其他元素。
- 过渡模式不可用，因为我们不再相互切换特有的元素。
- 内部元素总是需要提供唯一的 `key` attribute 值。
- CSS 过渡的类将会应用在内部的元素中，而不是这个组/容器本身。

列表的进入/离开过渡

现在让我们由一个简单的例子深入，进入和离开的过渡使用之前一样的 CSS class 名。

```

1. <div id="list-demo">
2.   <button @click="add">Add</button>
3.   <button @click="remove">Remove</button>
4.   <transition-group name="list" tag="p">
5.     <span v-for="item in items" :key="item" class="list-item">
6.       {{ item }}
7.     </span>
8.   </transition-group>
9. </div>

```

```

1. const Demo = {
2.   data() {
3.     return {
4.       items: [1, 2, 3, 4, 5, 6, 7, 8, 9],
5.       nextNum: 10
6.     }
7.   },
8.   methods: {
9.     randomIndex() {
10.      return Math.floor(Math.random() * this.items.length)
11.    },

```

```

12.     add() {
13.         this.items.splice(this.randomIndex(), 0, this.nextNum++)
14.     },
15.     remove() {
16.         this.items.splice(this.randomIndex(), 1)
17.     }
18. }
19. }
20.
21. Vue.createApp(Demo).mount('#list-demo')

```

```

1. .list-item {
2.     display: inline-block;
3.     margin-right: 10px;
4. }
5. .list-enter-active,
6. .list-leave-active {
7.     transition: all 1s ease;
8. }
9. .list-enter-from,
10. .list-leave-to {
11.     opacity: 0;
12.     transform: translateY(30px);
13. }

```

这个例子有个问题，当添加和移除元素的时候，周围的元素会瞬间移动到他们的新布局的位置，而不是平滑的过渡，我们下面会解决这个问题。

列表的排序过渡

`<transition-group>` 组件还有一个特殊之处。不仅可以进入和离开动画，还可以改变定位。要使用这个新功能只需了解新增的 `v-move` class，它会在元素的改变定位的过程中应用。像之前的类名一样，可以通过 `name` attribute 来自定义前缀，也可以通过 `move-class` attribute 手动设置。

该 class 主要用于指定过渡 timing 和 easing 曲线，如下所示：

```

<script
  src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.15/lodash.min.js">
1. </script>
2.
3. <div id="flip-list-demo">

```

```

4.   <button @click="shuffle">Shuffle</button>
5.   <transition-group name="flip-list" tag="ul">
6.     <li v-for="item in items" :key="item">
7.       {{ item }}
8.     </li>
9.   </transition-group>
10. </div>

```

```

1. const Demo = {
2.   data() {
3.     return {
4.       items: [1, 2, 3, 4, 5, 6, 7, 8, 9]
5.     }
6.   },
7.   methods: {
8.     shuffle() {
9.       this.items = _.shuffle(this.items)
10.    }
11.  }
12. }
13.
14. Vue.createApp(Demo).mount('#flip-list-demo')

```

```

1. .flip-list-move {
2.   transition: transform 0.8s ease;
3. }

```

这个看起来很神奇，内部的实现，Vue 使用了一个叫 [FLIP](#) (opens new window) 简单的动画队列使用 transforms 将元素从之前的位置平滑过渡新的位置。

我们将之前实现的例子和这个技术结合，使我们列表的一切变动都会有动画过渡。

```

<script
  src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.14.1/lodash.min.js">
1. </script>
2.
3. <div id="list-complete-demo" class="demo">
4.   <button @click="shuffle">Shuffle</button>
5.   <button @click="add">Add</button>
6.   <button @click="remove">Remove</button>
7.   <transition-group name="list-complete" tag="p">

```

```

8.     <span v-for="item in items" :key="item" class="list-complete-item">
9.         {{ item }}
10.    </span>
11. </transition-group>
12. </div>

```

```

1.  const Demo = {
2.    data() {
3.      return {
4.        items: [1, 2, 3, 4, 5, 6, 7, 8, 9],
5.        nextNum: 10
6.      }
7.    },
8.    methods: {
9.      randomIndex() {
10.        return Math.floor(Math.random() * this.items.length)
11.      },
12.      add() {
13.        this.items.splice(this.randomIndex(), 0, this.nextNum++)
14.      },
15.      remove() {
16.        this.items.splice(this.randomIndex(), 1)
17.      },
18.      shuffle() {
19.        this.items = _.shuffle(this.items)
20.      }
21.    }
22.  }
23.
24.  Vue.createApp(Demo).mount('#list-complete-demo')

```

```

1.  .list-complete-item {
2.    transition: all 0.8s ease;
3.    display: inline-block;
4.    margin-right: 10px;
5.  }
6.
7.  .list-complete-enter-from,
8.  .list-complete-leave-to {
9.    opacity: 0;
10.   transform: translateY(30px);

```

```

11. }
12.
13. .list-complete-leave-active {
14.   position: absolute;
15. }

```

TIP

需要注意的是使用 FLIP 过渡的元素不能设置为 `display: inline`。作为替代方案，可以设置为 `display: inline-block` 或者放置于 flex 中

FLIP 动画不仅可以实现单列过渡，多维网格也[同样可以过渡](#) (opens new window):

TODO: 示例

列表的交错过渡

通过 data attribute 与 JavaScript 通信，就可以实现列表的交错过渡：

```

    <script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.3.4/gsap.min.js">
1. </script>
2.
3. <div id="demo">
4.   <input v-model="query" />
5.   <transition-group
6.     name="staggered-fade"
7.     tag="ul"
8.     :css="false"
9.     @before-enter="beforeEnter"
10.    @enter="enter"
11.    @leave="leave"
12.  >
13.    <li
14.      v-for="(item, index) in computedList"
15.      :key="item.msg"
16.      :data-index="index"
17.    >
18.      {{ item.msg }}
19.    </li>
20.  </transition-group>
21. </div>

```

```
1. const Demo = {
2.   data() {
3.     return {
4.       query: '',
5.       list: [
6.         { msg: 'Bruce Lee' },
7.         { msg: 'Jackie Chan' },
8.         { msg: 'Chuck Norris' },
9.         { msg: 'Jet Li' },
10.        { msg: 'Kung Fury' }
11.      ]
12.    }
13.  },
14.  computed: {
15.    computedList() {
16.      var vm = this
17.      return this.list.filter(item => {
18.        return item.msg.toLowerCase().indexOf(vm.query.toLowerCase()) !== -1
19.      })
20.    }
21.  },
22.  methods: {
23.    beforeEnter(e1) {
24.      e1.style.opacity = 0
25.      e1.style.height = 0
26.    },
27.    enter(e1, done) {
28.      gsap.to(e1, {
29.        opacity: 1,
30.        height: '1.6em',
31.        delay: e1.dataset.index * 0.15,
32.        onComplete: done
33.      })
34.    },
35.    leave(e1, done) {
36.      gsap.to(e1, {
37.        opacity: 0,
38.        height: 0,
39.        delay: e1.dataset.index * 0.15,
40.        onComplete: done
41.      })
42.    }

```



```

43.   }
44. }
45.
46. Vue.createApp(Demo).mount('#demo')

```

可复用的过渡

过渡可以通过 Vue 的组件系统实现复用。要创建一个可复用过渡组件，你需要做的就是将

`<transition>` 或者 `<transition-group>` 作为根组件，然后将任何子组件放置在其中就可以了。

TODO: 使用 Vue3 重构

使用 `template` 的简单例子：

```

1. Vue.component('my-special-transition', {
2.   template: `
3.     <transition\
4.       name="very-special-transition"\
5.       mode="out-in"\
6.       @before-enter="beforeEnter"\
7.       @after-enter="afterEnter"\
8.     >\
9.     <slot></slot>\
10.   </transition>\
11. `,
12.   methods: {
13.     beforeEnter(e1) {
14.       // ...
15.     },
16.     afterEnter(e1) {
17.       // ...
18.     }
19.   }
20. })

```

[函数式组件](#)更适合完成这个任务：

```

1. Vue.component('my-special-transition', {
2.   functional: true,
3.   render: function(createElement, context) {

```

```

4.     var data = {
5.       props: {
6.         name: 'very-special-transition',
7.         mode: 'out-in'
8.       },
9.       on: {
10.        beforeEnter(e1) {
11.          // ...
12.        },
13.        afterEnter(e1) {
14.          // ...
15.        }
16.      }
17.    }
18.    return createElement('transition', data, context.children)
19.  }
20. })

```

动态过渡

在 Vue 中即使是过渡也是数据驱动的！动态过渡最基本的例子是通过 `name` attribute 来绑定动态值。

```

1. <transition :name="transitionName">
2.   <!-- ... -->
3. </transition>

```

当你想用 Vue 的过渡系统来定义的 CSS 过渡/动画在不同过渡间切换会非常有用。

所有过渡 attribute 都可以动态绑定，但我们不仅仅只有 attribute 可以利用，还可以通过事件钩子获取上下文中的所有数据，因为事件钩子都是方法。这意味着，根据组件的状态不同，你的 JavaScript 过渡会有不同的表现

```

<script
1. src="https://cdnjs.cloudflare.com/ajax/libs/velocity/1.2.3/velocity.min.js">
2. </script>
3. <div id="dynamic-fade-demo" class="demo">
4.   Fade In:
5.   <input type="range" v-model="fadeInDuration" min="0" :max="maxFadeDuration" />

```

```
6.   Fade Out:
7.   <input
8.     type="range"
9.     v-model="fadeOutDuration"
10.    min="0"
11.    :max="maxFadeDuration"
12.  />
13.  <transition
14.    :css="false"
15.    @before-enter="beforeEnter"
16.    @enter="enter"
17.    @leave="leave"
18.  >
19.    <p v-if="show">hello</p>
20.  </transition>
21.  <button v-if="stop" @click="stop = false; show = false">
22.    Start animating
23.  </button>
24.  <button v-else @click="stop = true">Stop it!</button>
25. </div>
```

```
1.  const app = Vue.createApp({
2.    data() {
3.      return {
4.        show: true,
5.        fadeInDuration: 1000,
6.        fadeOutDuration: 1000,
7.        maxFadeDuration: 1500,
8.        stop: true
9.      }
10.    },
11.    mounted() {
12.      this.show = false
13.    },
14.    methods: {
15.      beforeEnter(e1) {
16.        e1.style.opacity = 0
17.      },
18.      enter(e1, done) {
19.        var vm = this
20.        Velocity(
21.          e1,
```

```
22.     { opacity: 1 },
23.     {
24.       duration: this.fadeInDuration,
25.       complete: function() {
26.         done()
27.         if (!vm.stop) vm.show = false
28.       }
29.     }
30.   )
31. },
32. leave(el, done) {
33.   var vm = this
34.   Velocity(
35.     el,
36.     { opacity: 0 },
37.     {
38.       duration: this.fadeOutDuration,
39.       complete: function() {
40.         done()
41.         vm.show = true
42.       }
43.     }
44.   )
45. }
46. }
47. })
48.
49. app.mount( '#dynamic-fade-demo' )
```

TODO: 示例

最后，创建动态过渡的最终方案是组件通过接受 `props` 来动态修改之前的过渡。一句老话，唯一的限制是你的想象力。

状态过渡

Vue 的过渡系统提供了非常多简单的方法设置进入、离开和列表的动效。那么对于数据元素本身的动效呢，比如：

- 数字和运算
- 颜色的显示
- SVG 节点的位置
- 元素的大小和其他的 property

这些数据要么本身就以数值形式存储，要么可以转换为数值。有了这些数值后，我们就可以结合 Vue 的响应式和组件系统，使用第三方库来实现切换元素的过渡状态。

状态动画与侦听器

通过侦听器我们能监听到任何数值 property 的数值更新。可能听起来很抽象，所以让我们先来看看使用 [GreenSock](#) (opens new window) 一个例子：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.2.4/gsap.min.js">
1. </script>
2.
3. <div id="animated-number-demo">
4.   <input v-model.number="number" type="number" step="20" />
5.   <p>{{ animatedNumber }}</p>
6. </div>
```

```
1. const Demo = {
2.   data() {
3.     return {
4.       number: 0,
5.       tweenedNumber: 0
6.     }
7.   },
8.   computed: {
9.     animatedNumber() {
10.      return this.tweenedNumber.toFixed(0)
11.    }
12.  },
13.  watch: {
```

```
14.     number(newValue) {
15.         gsap.to(this.$data, { duration: 0.5, tweenedNumber: newValue })
16.     }
17. }
18. }
19.
20. Vue.createApp(Demo).mount('#animated-number-demo')
```

更新数字时，更改将在输入下方设置动画。

动态状态过渡

就像 Vue 的过渡组件一样，数据背后状态过渡会实时更新，这对于原型设计十分有用。当你修改一些变量，即使是一个简单的 SVG 多边形也可实现很多难以想象的效果。

把过渡放到组件里

管理太多的状态过渡会很快地增加组件实例复杂性，幸好很多的动画可以提取到专用的子组件。我们将之前的示例改写一下：

```
    <script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.2.4/gsap.min.js">
1. </script>
2.
3. <div id="app">
4.   <input v-model.number="firstNumber" type="number" step="20" /> +
   <input v-model.number="secondNumber" type="number" step="20" /> = {{ result
5. }}
6.   <p>
7.     <animated-integer :value="firstNumber"></animated-integer> +
8.     <animated-integer :value="secondNumber"></animated-integer> =
9.     <animated-integer :value="result"></animated-integer>
10.   </p>
11. </div>
```

```
1. const app = Vue.createApp({
2.   data() {
3.     return {
4.       firstNumber: 20,
5.       secondNumber: 40
6.     }
7.   },
```

```
8.   computed: {
9.     result() {
10.      return this.firstNumber + this.secondNumber
11.    }
12.  }
13. })
14.
15. app.component('animated-integer', {
16.   template: '<span>{{ fullValue }}</span>',
17.   props: {
18.     value: {
19.       type: Number,
20.       required: true
21.     }
22.   },
23.   data() {
24.     return {
25.       tweeningValue: 0
26.     }
27.   },
28.   computed: {
29.     fullValue() {
30.       return Math.floor(this.tweeningValue)
31.     }
32.   },
33.   methods: {
34.     tween(newValue, oldValue) {
35.       gsap.to(this.$data, {
36.         duration: 0.5,
37.         tweeningValue: newValue,
38.         ease: 'sine'
39.       })
40.     }
41.   },
42.   watch: {
43.     value(newValue, oldValue) {
44.       this.tween(newValue, oldValue)
45.     }
46.   },
47.   mounted() {
48.     this.tween(this.value, 0)
49.   }

```

```
50. })  
51.  
52. app.mount('#app')
```

我们能在组件中结合使用这一节讲到各种过渡策略和 [Vue 内建的过渡系统](#)。总之，对于完成各种过渡动效几乎没有阻碍。

你可以看到我们如何使用它进行数据可视化，物理效果，角色动画和交互，天空是极限。

赋予设计以生命

只要一个动画，就可以带来生命。不幸的是，当设计师创建图标、logo 和吉祥物的时候，他们交付的通常都是图片或静态的 SVG。所以，虽然 GitHub 的章鱼猫、Twitter 的小鸟以及其它许多 logo 类似于生灵，它们看上去实际上并不是活着的。

Vue 可以帮到你。因为 SVG 的本质是数据，我们只需要这些动物兴奋、思考或警戒的样例。然后 Vue 就可以辅助完成这几种状态之间的过渡动画，来制作你的欢迎页面、加载指示、以及更加带有情感的提示。

Sarah Drasner 展示了下面这个 demo，这个 demo 结合了时间和交互相关的状态改变：

可复用性

- 混入
- 自定义指令
- 传入
- 渲染函数
- 插件

混入

基础

混入 (mixin) 提供了一种非常灵活的方式，来分发 Vue 组件中的可复用功能。一个混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被“混合”进入该组件本身的选项。

例子：

```
1. // define a mixin object
2. const myMixin = {
3.   created() {
4.     this.hello()
5.   },
6.   methods: {
7.     hello() {
8.       console.log('hello from mixin!')
9.     }
10.  }
11. }
12.
13. // define an app that uses this mixin
14. const app = Vue.createApp({
15.   mixins: [myMixin]
16. })
17.
18. app.mount('#mixins-basic') // => "hello from mixin!"
```

选项合并

当组件和混入对象含有同名选项时，这些选项将以恰当的方式进行“合并”。

比如，数据对象在内部会进行递归合并，并在发生冲突时以组件数据优先。

```
1. const myMixin = {
2.   data() {
3.     return {
4.       message: 'hello',
5.       foo: 'abc'
```

```

6.     }
7.   }
8. }
9.
10. const app = Vue.createApp({
11.   mixins: [myMixin],
12.   data() {
13.     return {
14.       message: 'goodbye',
15.       bar: 'def'
16.     }
17.   },
18.   created() {
19.     console.log(this.$data) // => { message: "goodbye", foo: "abc", bar: "def"
20.   }
21. })

```

同名钩子函数将合并为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子之前调用。

```

1. const myMixin = {
2.   created() {
3.     console.log('mixin hook called')
4.   }
5. }
6.
7. const app = Vue.createApp({
8.   mixins: [myMixin],
9.   created() {
10.    console.log('component hook called')
11.  }
12. })
13.
14. // => "混入对象的钩子被调用"
15. // => "组件钩子被调用"

```

值为对象的选项，例如 `methods`、`components` 和 `directives`，将被合并为同一个对象。两个对象键名冲突时，取组件对象的键值对。

```

1. const myMixin = {
2.   methods: {

```

```
3.     foo() {
4.       console.log('foo')
5.     },
6.     conflicting() {
7.       console.log('from mixin')
8.     }
9.   }
10. }
11.
12. const app = Vue.createApp({
13.   mixins: [myMixin],
14.   methods: {
15.     bar() {
16.       console.log('bar')
17.     },
18.     conflicting() {
19.       console.log('from self')
20.     }
21.   }
22. })
23.
24. const vm = app.mount('#mixins-basic')
25.
26. vm.foo() // => "foo"
27. vm.bar() // => "bar"
28. vm.conflicting() // => "from self"
```

全局混入

你还可以为 Vue 应用程序全局应用 mixin：

```
1. const app = Vue.createApp({
2.   myOption: 'hello!'
3. })
4.
5. // 为自定义的选项 'myOption' 注入一个处理器。
6. app.mixin({
7.   created() {
8.     const myOption = this.$options.myOption
9.     if (myOption) {
10.       console.log(myOption)

```

```

11.     }
12.   }
13. })
14.
15. app.mount('#mixins-global') // => "hello!"

```

混入也可以进行全局注册。使用时格外小心！一旦使用全局混入，它将影响每一个之后创建的组件（例如，每个子组件）。

```

1.  const app = Vue.createApp({
2.    myOption: 'hello!'
3.  })
4.
5.  // 为自定义的选项 'myOption' 注入一个处理器。
6.  app.mixin({
7.    created() {
8.      const myOption = this.$options.myOption
9.      if (myOption) {
10.        console.log(myOption)
11.      }
12.    }
13.  })
14.
15.  // 将myOption也添加到子组件
16.  app.component('test-component', {
17.    myOption: 'hello from component!'
18.  })
19.
20.  app.mount('#mixins-global')
21.
22.  // => "hello!"
23.  // => "hello from component!"

```

大多数情况下，只应当应用于自定义选项，就像上面示例一样。推荐将其作为插件发布，以避免重复应用混入。

自定义选项合并策略

自定义选项将使用默认策略，即简单地覆盖已有值。如果想让自定义选项以自定义逻辑合并，可以向

`app.config.optionMergeStrategies` 添加一个函数：

```

1. const app = Vue.createApp({})
2.
3. app.config.optionMergeStrategies.customOption = (toVal, fromVal) => {
4.   // return mergedVal
5. }

```

合并策略接收在父实例和子实例上定义的该选项的值，分别作为第一个和第二个参数。让我们来检查一下使用 `mixin` 时，这些参数有哪些：

```

1. const app = Vue.createApp({
2.   custom: 'hello!'
3. })
4.
5. app.config.optionMergeStrategies.custom = (toVal, fromVal) => {
6.   console.log(fromVal, toVal)
7.   // => "goodbye!", undefined
8.   // => "hello", "goodbye!"
9.   return fromVal || toVal
10. }
11.
12. app.mixin({
13.   custom: 'goodbye!',
14.   created() {
15.     console.log(this.$options.custom) // => "hello!"
16.   }
17. })

```

如你所见，在控制台中，我们先从 `mixin` 打印 `toVal` 和 `fromVal`，然后从 `app` 打印。如果存在，我们总是返回 `fromVal`，这就是为什么 `this.$options.custom` 设置为 `你好！` 最后。让我们尝试将策略更改为始终从子实例返回值：

```

1. const app = Vue.createApp({
2.   custom: 'hello!'
3. })
4.
5. app.config.optionMergeStrategies.custom = (toVal, fromVal) => toVal || fromVal
6.
7. app.mixin({
8.   custom: 'goodbye!',
9.   created() {
10.     console.log(this.$options.custom) // => "goodbye!"

```

```
11.   }  
12. })
```

在 Vue 2 中，mixin 是将部分组件逻辑抽象成可重用块的主要工具。但是，他们有几个问题：

- mixin 很容易发生冲突：因为每个特性的属性都被合并到同一个组件中，所以为了避免 property 名冲突和调试，你仍然需要了解其他每个特性。
- 可重用性是有限的：我们不能向 mixin 传递任何参数来改变它的逻辑，这降低了它们在抽象逻辑方面的灵活性

为了解决这些问题，我们添加了一种通过逻辑关注点组织代码的新方法：[Composition API](#)。

自定义指令

简介

除了核心功能默认内置的指令 (`v-model` 和 `v-show`), Vue 也允许注册自定义指令。注意, 在 Vue2.0 中, 代码复用和抽象的主要形式是组件。然而, 有的情况下, 你仍然需要对普通 DOM 元素进行底层操作, 这时候就会用到自定义指令。举个聚焦输入框的例子, 如下:

当页面加载时, 该元素将获得焦点 (注意: `autofocus` 在移动版 Safari 上不工作)。事实上, 只要你在打开这个页面后还没点击过任何内容, 这个输入框就应当还是处于聚焦状态。此外, 你可以单击 `Rerun` 按钮, 输入将被聚焦。

现在让我们用指令来实现这个功能:

```
1. const app = Vue.createApp({})
2. // 注册一个全局自定义指令 `v-focus`
3. app.directive('focus', {
4.   // 当被绑定的元素插入到 DOM 中时.....
5.   mounted(el) {
6.     // Focus the element
7.     el.focus()
8.   }
9. })
```

如果想注册局部指令, 组件中也接受一个 `directives` 的选项:

```
1. directives: {
2.   focus: {
3.     // 指令的定义
4.     mounted(el) {
5.       el.focus()
6.     }
7.   }
8. }
```

然后你可以在模板中任何元素上使用新的 `v-focus` property, 如下:

```
1. <input v-focus />
```


钩子函数

一个指令定义对象可以提供如下几个钩子函数（均为可选）：

- `beforeMount`：当指令第一次绑定到元素并且在挂载父组件之前调用。在这里你可以做一次性的初始化设置。
- `mounted`：在挂载绑定元素的父组件时调用。
- `beforeUpdate`：在更新包含组件的 `VNode` 之前调用。

提示

我们会在稍后讨论渲染函数时介绍更多 `VNodes` 的细节。

- `updated`：在包含组件的 `VNode` 及其子组件的 `VNode` 更新后调用。
- `beforeUnmount`：在卸载绑定元素的父组件之前调用
- `unmounted`：当指令与元素解除绑定且父组件已卸载时，只调用一次。

接下来我们来看一下在自定义指令 API 钩子函数的参数（即 `el`、`binding`、`vnode` 和 `prevVnode`）

动态指令参数

指令的参数可以是动态的。例如，在 `v-mydirective:[argument]="value"` 中，`argument` 参数可以根据组件实例数据进行更新！这使得自定义指令可以在应用中被灵活使用。

例如你想要创建一个自定义指令，用来通过固定布局将元素固定在页面上。我们可以像这样创建一个通过指令值来更新垂直位置像素值的自定义指令：

```
1. <div id="dynamic-arguments-example" class="demo">
2.   <p>Scroll down the page</p>
3.   <p v-pin="200">Stick me 200px from the top of the page</p>
4. </div>
```

```
1. const app = Vue.createApp({})
2.
3. app.directive('pin', {
4.   mounted(el, binding) {
5.     el.style.position = 'fixed'
6.     // binding.value is the value we pass to directive - in this case, it's 200
7.     el.style.top = binding.value + 'px'
```

```

8.   }
9.  })
10.
11.  app.mount('#dynamic-arguments-example')

```

这会把该元素固定在距离页面顶部 200 像素的位置。但如果场景是我们需要把元素固定在左侧而不是顶部又该怎么办呢？这时使用动态参数就可以非常方便地根据每个组件实例来进行更新。

```

1.  <div id="dynamicexample">
2.    <h3>Scroll down inside this section ↓</h3>
3.    <p v-pin:[direction]="200">I am pinned onto the page at 200px to the left.
4.  </p>
5. </div>

```

```

1.  const app = Vue.createApp({
2.    data() {
3.      return {
4.        direction: 'right'
5.      }
6.    }
7.  })
8.
9.  app.directive('pin', {
10.    mounted(el, binding) {
11.      el.style.position = 'fixed'
12.      // binding.arg is an argument we pass to directive
13.      const s = binding.arg || 'top'
14.      el.style[s] = binding.value + 'px'
15.    }
16.  })
17.
18.  app.mount('#dynamic-arguments-example')

```

结果：

我们的定制指令现在已经足够灵活，可以支持一些不同的用例。为了使其更具动态性，我们还可以允许修改绑定值。让我们创建一个附加属性 `pinPadding`，并将其绑定到 `<input type="range">`。

```

1.  <div id="dynamicexample">
2.    <h2>Scroll down the page</h2>
3.    <input type="range" min="0" max="500" v-model="pinPadding">

```

```

    <p v-pin:[direction]="pinPadding">Stick me {{ pinPadding + 'px' }} from the
4.  {{ direction }} of the page</p>
5. </div>

```

```

1. const app = Vue.createApp({
2.   data() {
3.     return {
4.       direction: 'right',
5.       pinPadding: 200
6.     }
7.   }
8. })

```

让我们扩展我们的指令逻辑来重新计算固定元素更新的距离。

```

1. app.directive('pin', {
2.   mounted(el, binding) {
3.     el.style.position = 'fixed'
4.     const s = binding.arg || 'top'
5.     el.style[s] = binding.value + 'px'
6.   },
7.   updated(el, binding) {
8.     const s = binding.arg || 'top'
9.     el.style[s] = binding.value + 'px'
10.  }
11. })

```

结果：

函数简写

在很多时候，你可能想在 `mounted` 和 `updated` 时触发相同行为，而不关心其它的钩子。比如这样写：

```

1. app.directive('pin', (el, binding) => {
2.   el.style.position = 'fixed'
3.   const s = binding.arg || 'top'
4.   el.style[s] = binding.value + 'px'
5. })

```

对象字面量

如果指令需要多个值，可以传入一个 JavaScript 对象字面量。记住，指令函数能够接受所有合法的 JavaScript 表达式。

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

```
1. app.directive('demo', (el, binding) => {
2.   console.log(binding.value.color) // => "white"
3.   console.log(binding.value.text) // => "hello!"
4. })
```

在组件中使用

在 3.0 中，有了片段支持，组件可能有多个根节点。如果在具有多个根节点的组件上使用自定义指令，则会产生问题。

要解释自定义指令如何在 3.0 中的组件上工作的详细信息，我们首先需要了解自定义指令在 3.0 中是如何编译的。对于这样的指令：

```
1. <div v-demo="test"></div>
```

将大概编译成：

```
1. const vDemo = resolveDirective('demo')
2.
3. return withDirectives(h('div'), [[vDemo, test]])
```

其中 `vDemo` 是用户编写的指令对象，其中包含 `mounted` 和 `updated` 等钩子。

`withDirectives` 返回一个克隆的 `VNode`，其中用户钩子被包装并作为 `VNode` 生命周期钩子注入（请参见[渲染函数](#)更多详情）：

```
1. {
2.   onVnodeMounted(vnode) {
3.     // call vDemo.mounted(...)
4.   }
5. }
```

因此，自定义指令作为 `VNode` 数据的一部分完全包含在内。当在组件上使用自定义指令时，这些

`onVnodeXXX` 钩子作为无关的 `prop` 传递给组件，并以 `this.$attrs` 结束。

这也意味着可以像这样在模板中直接挂接到元素的生命周期中，这在涉及到自定义指令时非常方便：

```
1. <div @vnodeMounted="myHook" />
```

这与 `attribute fallthrough behavior`。因此，组件上自定义指令的规则将与其他无关 `attribute` 相同：由子组件决定在哪里以及是否应用它。当子组件在内部元素上使用 `v-`
`bind="$attrs"` 时，它也将应用对其使用的任何自定义指令。

传入

Vue 鼓励我们通过将 UI 和相关行为封装到组件中来构建 UI。我们可以将它们嵌套在另一个内部，以构建一个组成应用程序 UI 的树。

然而，有时组件模板的一部分逻辑上属于该组件，而从技术角度来看，最好将模板的这一部分移动到 DOM 中 Vue app 之外的其他位置。

一个常见的场景是创建一个包含全屏模式的组件。在大多数情况下，你希望模态的逻辑存在于组件中，但是模态的定位很快就很难通过 CSS 来解决，或者需要更改组件组合。

考虑下面的 HTML 结构。

```
1. <body>
2.   <div style="position: relative;">
3.     <h3>Tooltips with Vue 3 Teleport</h3>
4.     <div>
5.       <modal-button></modal-button>
6.     </div>
7.   </div>
8. </body>
```

让我们来看看 `modal-button` 组件：

该组件将有一个 `button` 元素来触发模态的打开，以及一个具有类 `.modal` 的 `div` 元素，它将包含模态的内容和一个用于自关闭的按钮。

```
1. const app = Vue.createApp({});
2.
3. app.component('modal-button', {
4.   template: `
5.     <button @click="modalOpen = true">
6.       Open full screen modal!
7.     </button>
8.
9.     <div v-if="modalOpen" class="modal">
10.      <div>
11.        I'm a modal!
12.        <button @click="modalOpen = false">
13.          Close
14.        </button>
```

```
15.     </div>
16.   </div>
17. ` ,
18.   data() {
19.     return {
20.       modalOpen: false
21.     }
22.   }
23. })
```

当在初始的 HTML 结构中使用这个组件时，我们可以看到一个问题——模态是在深度嵌套的 `div` 中渲染的，而模态的 `position: absolute` 以父级相对定位的 `div` 作为引用。

Teleport 提供了一种干净的方法，允许我们控制在 DOM 中哪个父节点下呈现 HTML，而不必求助于全局状态或将其拆分为两个组件。

让我们修改 `modal-button` 以使用 `<teleport>`，并告诉 Vue 将这个 HTML 传入至该 `body` 标记。

```
1. app.component('modal-button', {
2.   template: `
3.     <button @click="modalOpen = true">
4.       Open full screen modal! (With teleport!)
5.     </button>
6.
7.     <teleport to="body">
8.       <div v-if="modalOpen" class="modal">
9.         <div>
10.          I'm a teleported modal!
11.          (My parent is "body")
12.          <button @click="modalOpen = false">
13.            Close
14.          </button>
15.        </div>
16.      </div>
17.    </teleport>
18. ` ,
19.   data() {
20.     return {
21.       modalOpen: false
22.     }
23.   }
```

```
24. })
```

因此，一旦我们单击按钮打开模式，Vue 将正确地将模态内容渲染为 `body` 标签的子级。

与 Vue components 一起使用

如果 `<teleport>` 包含 Vue 组件，则它仍将是 `<teleport>` 父组件的逻辑子组件：

```
1. const app = Vue.createApp({
2.   template: `
3.     <h1>Root instance</h1>
4.     <parent-component />
5.   `
6. })
7.
8. app.component('parent-component', {
9.   template: `
10.    <h2>This is a parent component</h2>
11.    <teleport to="#endofbody">
12.      <child-component name="John" />
13.    </teleport>
14.  `
15. })
16.
17. app.component('child-component', {
18.   props: ['name'],
19.   template: `
20.    <div>Hello, {{ name }}</div>
21.  `
22. })
```

在这种情况下，即使在不同的地方渲染 `child-component`，它仍将是 `parent-component` 的子级，并将从中接收 `name` prop。

这也意味着来自父组件的注入按预期工作，并且子组件将嵌套在 Vue Devtools 中的父组件之下，而不是放在实际内容移动到的位置。

在同一目标上使用多个传送

一个常见的用例场景是一个可重用的 `<Modal>` 组件，它可能同时有多个实例处于活动状态。对于这种情况，多个 `<teleport>` 组件可以将其内容挂载到同一个目标元素。顺序将是一个简单的追加

—稍后挂载将位于目标元素中较早的挂载之后。

```
1. <teleport to="#modals">
2.   <div>A</div>
3. </teleport>
4. <teleport to="#modals">
5.   <div>B</div>
6. </teleport>
7.
8. <!-- result-->
9. <div id="modals">
10.  <div>A</div>
11.  <div>B</div>
12. </div>
```

你可以在 [API reference](#) 查看 `teleport` 组件。

渲染函数

Vue 推荐在绝大多数情况下使用模板来创建你的 HTML。然而在一些场景中，你真的需要 JavaScript 的完全编程的能力。这时你可以用渲染函数，它比模板更接近编译器。

让我们深入一个简单的例子，这个例子里 `render` 函数很实用。假设我们要生成一些带锚点的标题：

```
1. <h1>
2.   <a name="hello-world" href="#hello-world">
3.     Hello world!
4.   </a>
5. </h1>
```

锚点标题的使用非常频繁，我们应该创建一个组件：

```
1. <anchored-heading :level="1">Hello world!</anchored-heading>
```

当开始写一个只能通过 `level` prop 动态生成标题 (heading) 的组件时，我们很快就可以得出这样的结论：

```
1. const app = Vue.createApp({})
2.
3. app.component('anchored-heading', {
4.   template: `
5.     <h1 v-if="level === 1">
6.       <slot></slot>
7.     </h1>
8.     <h2 v-else-if="level === 2">
9.       <slot></slot>
10.    </h2>
11.    <h3 v-else-if="level === 3">
12.      <slot></slot>
13.    </h3>
14.    <h4 v-else-if="level === 4">
15.      <slot></slot>
16.    </h4>
17.    <h5 v-else-if="level === 5">
18.      <slot></slot>
19.    </h5>
```

```

20.     <h6 v-else-if="level === 6">
21.         <slot></slot>
22.     </h6>
23. ` ,
24. props: {
25.     level: {
26.         type: Number,
27.         required: true
28.     }
29. }
30. })

```

这个模板感觉不太好。它不仅冗长，而且我们为每个级别标题重复书写了 `<slot></slot>`。当我们添加锚元素时，我们必须在每个 `v-if/v-else-if` 分支中再次重复它。

虽然模板在大多数组件中都非常好用，但是显然在这里它就不合适了。那么，我们来尝试使用

`render` 函数重写上面的例子：

```

1. const app = Vue.createApp({})
2.
3. app.component('anchored-heading', {
4.   render() {
5.     const { h } = Vue
6.
7.     return h(
8.       'h' + this.level, // tag name
9.       {}, // props/attributes
10.      this.$slots.default() // array of children
11.    )
12.  },
13.  props: {
14.    level: {
15.      type: Number,
16.      required: true
17.    }
18.  }
19. })

```

`render()` 函数的实现要精简得多，但是需要非常熟悉组件的实例 `property`。在这个例子中，你需要知道，向组件中传递不带 `v-slot` 指令的子节点时，比如 `anchored-heading` 中的 `Hello world!`，这些子节点被存储在组件实例中的 `$slots.default` 中。如果你还不了解，在深入渲染函数之前推荐阅读[实例 property API](#)。

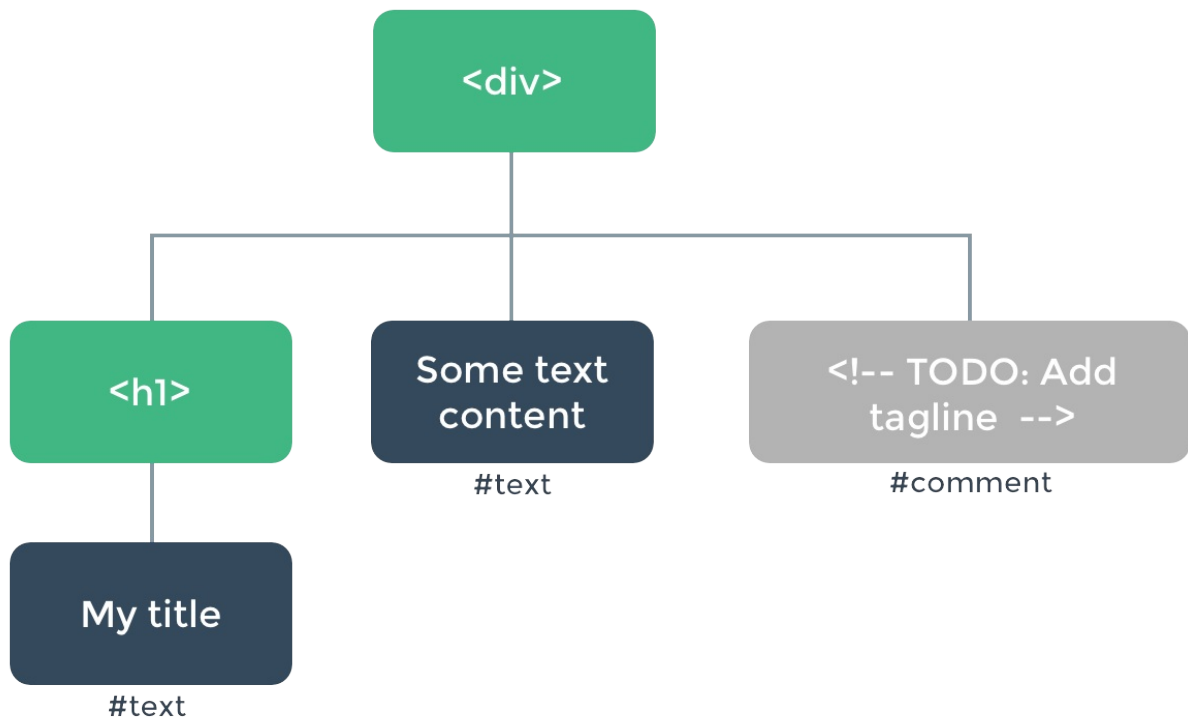
DOM 树

在深入渲染函数之前，了解一些浏览器的工作原理是很重要的。以下面这段 HTML 为例：

```
1. <div>
2.   <h1>My title</h1>
3.   Some text content
4.   <!-- TODO: Add tagline -->
5. </div>
```

当浏览器读到这些代码时，它会建立一个“DOM 节点”树 [\(opens new window\)](#) 来保持追踪所有内容，如同你会画一张家谱树来追踪家庭成员的发展一样。

上述 HTML 对应的 DOM 节点树如下图所示



每个元素都是一个节点。每段文字也是一个节点。甚至注释也都是节点。一个节点就是页面的一个部分。就像家谱树一样，每个节点都可以有孩子节点（也就是说每个部分可以包含其它的一些部分）。

高效地更新所有这些节点会比较困难的，不过所幸你不必手动完成这个工作。你只需要告诉 Vue 你希望页面上的 HTML 是什么，这可以是在一个模板里：

```
1. <h1>{{ blogTitle }}</h1>
```

或者一个渲染函数里：

```
1. render() {
2.   return Vue.h('h1', {}, this.blogTitle)
3. }
```

在这两种情况下，Vue 都会自动保持页面的更新，即便 `blogTitle` 发生了改变。

虚拟 DOM 树

Vue 通过建立一个虚拟 **DOM** 来追踪自己要如何改变真实 DOM。请仔细看这行代码：

```
1. return Vue.h('h1', {}, this.blogTitle)
```

`h()` 到底会返回什么呢？其实不是一个实际的 DOM 元素。它更准确的名字可能是 `createNodeDescription`，因为它所包含的信息会告诉 Vue 页面上需要渲染什么样的节点，包括及其子节点的描述信息。我们把这样的节点描述为“虚拟节点 (virtual node)”，也常简写它为 **VNode**。“虚拟 DOM”是我们对由 Vue 组件树建立起来的整个 VNode 树的称呼。

`h()` 参数

`h()` 函数是一个用于创建 vnode 的实用程序。也许可以更准确地将其命名为 `createVNode()`，但由于频繁使用和简洁，它被称为 `h()`。它接受三个参数：

```
1. // @returns {VNode}
2. h(
3.   // {String | Object | Function | null} tag
4.   // 一个 HTML 标签名、一个组件、一个异步组件，或者 null。
5.   // 使用 null 将会渲染一个注释。
6.   //
7.   // 必需的。
8.   'div',
9.
10.  // {Object} props
11.  // 与 attribute、prop 和事件相对应的对象。
12.  // 我们会在模板中使用。
13.  //
14.  // 可选的。
15.  {},
16.
```

```

17. // {String | Array | Object} children
18. // 子 VNodes, 使用 `h()` 构建,
19. // 或使用字符串获取 "文本 Vnode" 或者
20. // 有 slot 的对象。
21. //
22. // 可选的。
23. [
24.   'Some text comes first.',
25.   h('h1', 'A headline'),
26.   h(MyComponent, {
27.     someProp: 'foobar'
28.   })
29. ]
30. )

```

完整实例

有了这些知识，我们现在可以完成我们最开始想实现的组件：

```

1. const app = Vue.createApp({})
2.
3. /** Recursively get text from children nodes */
4. function getChildrenTextContent(children) {
5.   return children
6.     .map(node => {
7.       return typeof node.children === 'string'
8.         ? node.children
9.         : Array.isArray(node.children)
10.          ? getChildrenTextContent(node.children)
11.          : ''
12.     })
13.   .join('')
14. }
15.
16. app.component('anchored-heading', {
17.   render() {
18.     // create kebab-case id from the text contents of the children
19.     const headingId = getChildrenTextContent(this.$slots.default())
20.       .toLowerCase()
21.       .replace(/\W+/g, '-') // replace non-word characters with dash
22.       .replace(/^(-|-)$/g, '') // remove leading and trailing dashes

```

```

23.
24.     return Vue.h('h' + this.level, [
25.         Vue.h(
26.             'a',
27.             {
28.                 name: headingId,
29.                 href: '#' + headingId
30.             },
31.             this.$slots.default()
32.         )
33.     ])
34. },
35. props: {
36.     level: {
37.         type: Number,
38.         required: true
39.     }
40. }
41. })

```

约束

VNodes 必须唯一

组件树中的所有 VNode 必须是唯一的。这意味着，下面的渲染函数是不合法的：

```

1. render() {
2.     const myParagraphVNode = Vue.h('p', 'hi')
3.     return Vue.h('div', [
4.         // 错误 - 重复的Vnode!
5.         myParagraphVNode, myParagraphVNode
6.     ])
7. }

```

如果你真的需要重复很多次的元素/组件，你可以使用工厂函数来实现。例如，下面这渲染函数用完全合法的方式渲染了 20 个相同的段落：

```

1. render() {
2.     return Vue.h('div',
3.         Array.apply(null, { length: 20 }).map(() => {
4.             return Vue.h('p', 'hi')

```

```

5.     })
6.   )
7. }

```

使用 JavaScript 代替模板功能

`v-if` 和 `v-for`

只要在原生的 JavaScript 中可以轻松完成的操作，Vue 的渲染函数就不会提供专有的替代方法。比如，在模板中使用的 `v-if` 和 `v-for`：

```

1. <ul v-if="items.length">
2.   <li v-for="item in items">{{ item.name }}</li>
3. </ul>
4. <p v-else>No items found.</p>

```

这些都可以在渲染函数中用 JavaScript 的 `if / else` 和 `map()` 来重写：

```

1. props: ['items'],
2. render() {
3.   if (this.items.length) {
4.     return Vue.h('ul', this.items.map((item) => {
5.       return Vue.h('li', item.name)
6.     }))
7.   } else {
8.     return Vue.h('p', 'No items found.')
9.   }
10. }

```

`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prop：

```

1. props: ['modelValue'],
2. render() {
3.   return Vue.h(SomeComponent, {
4.     modelValue: this.modelValue,
5.     'onUpdate:modelValue': value => this.$emit('update:modelValue', value)
6.   })
7. }

```


v-on

我们必须为事件处理程序提供一个正确的prop名称，例如，要处理 `click` 事件，prop名称应该是 `onClick`。

```

1. render() {
2.   return Vue.h('div', {
3.     onClick: $event => console.log('clicked', $event.target)
4.   })
5. }

```

事件修饰符

对于 `.passive`、`.capture` 和 `.once` 事件修饰符，Vue提供了处理程序的对象语法：

实例：

```

1. render() {
2.   return Vue.h('input', {
3.     onClick: {
4.       handler: this.doThisInCapturingMode,
5.       capture: true
6.     },
7.     onKeyUp: {
8.       handler: this.doThisOnce,
9.       once: true
10.    },
11.    onMouseOver: {
12.      handler: this.doThisOnceInCapturingMode,
13.      once: true,
14.      capture: true
15.    },
16.  })
17. }

```

对于所有其它的修饰符，私有前缀都不是必须的，因为你可以事件处理函数中使用事件方法：

修饰符	处理函数中的等价操作
<code>.stop</code>	<code>event.stopPropagation()</code>
<code>.prevent</code>	<code>event.preventDefault()</code>
<code>.self</code>	<code>if (event.target !== event.currentTarget) return</code>

按键： <code>.enter</code> , <code>.13</code>	<code>if (event.keyCode !== 13) return</code> (对于别的按键修饰符来说，可将 <code>13</code> 改为另一个按键码 ↗ (opens new window))
修饰键： <code>.ctrl</code> , <code>.alt</code> , <code>.shift</code> , <code>.meta</code>	<code>if (!event.ctrlKey) return</code> (将 <code>ctrlKey</code> 分别修改为 <code>altKey</code> , <code>shiftKey</code> , 或 <code>metaKey</code>)

这里是一个使用所有修饰符的例子：

```

1. render() {
2.   return Vue.h('input', {
3.     onKeyUp: event => {
4.       // 如果触发事件的元素不是事件绑定的元素
5.       // 则返回
6.       if (event.target !== event.currentTarget) return
7.       // 如果向上键不是回车键，则中止
8.       // 没有同时按下按键 (13) 和 shift 键
9.       if (!event.shiftKey || event.keyCode !== 13) return
10.      // 停止事件传播
11.      event.stopPropagation()
12.      // 阻止该元素默认的 keyup 事件
13.      event.preventDefault()
14.      // ...
15.    }
16.  })
17. }

```

插槽

你可以通过 `this.$slots` 访问静态插槽的内容，每个插槽都是一个 VNode 数组：

```

1. render() {
2.   // `

<slot></slot></div>`
3.   return Vue.h('div', {}, this.$slots.default())
4. }


```

```

1. props: ['message'],
2. render() {
3.   // `

<slot :text="message"></slot></div>`
4.   return Vue.h('div', {}, this.$slots.default({
5.     text: this.message
6.   })))
7. }


```

要使用渲染函数将插槽传递给子组件，请执行以下操作：

```
1. render() {
2.   // `

<child v-slot="props"><span>{{ props.text }}</span></child></div>`
3.   return Vue.h('div', [
4.     Vue.h('child', {}, {
5.       // pass `slots` as the children object
6.       // in the form of { name: props => VNode | Array<VNode> }
7.       default: (props) => Vue.h('span', props.text)
8.     })
9.   ])
10. }


```

JSX

如果你写了很多渲染函数，可能会觉得下面这样的代码写起来很痛苦：

```
1. Vue.h(
2.   'anchored-heading',
3.   {
4.     level: 1
5.   },
6.   [Vue.h('span', 'Hello'), ' world!']
7. )
```

特别是对应的模板如此简单的情况下：

```
1. <anchored-heading :level="1"> <span>Hello</span> world! </anchored-heading>
```

这就是为什么会有一个 [Babel 插件](#) (opens new window)，用于在 Vue 中使用 JSX 语法，它可以让我们回到更接近于模板的语法上。

```
1. import AnchoredHeading from './AnchoredHeading.vue'
2.
3. new Vue({
4.   el: '#demo',
5.   render() {
6.     return (
7.       <AnchoredHeading level={1}>
```

```
8.         <span>Hello</span> world!  
9.         </AnchoredHeading>  
10.    )  
11.  }  
12. })
```

有关 JSX 如何映射到 JavaScript 的更多信息，请参阅[使用文档](#) [↗] (opens new window)。

模板编译

你可能会有兴趣知道，Vue 的模板实际上被编译成了渲染函数。这是一个实现细节，通常不需要关心。但如果你想看看模板的功能具体是怎样被编译的，可能会发现会非常有意思。下面是一个使用

`Vue.compile` 来实时编译模板字符串的简单示例：

插件

插件是自包含的代码，通常向 Vue 添加全局级功能。它可以是公开 `install()` 方法的 `object`，也可以是 `function`

插件的功能范围没有严格的限制——一般有下面几种：

1. 添加全局方法或者 property。如：[vue-custom-element](#) [↗] (opens new window)
2. 添加全局资源：指令/过滤器/过渡等。如：[vue-touch](#) [↗] (opens new window)
3. 通过全局混入来添加一些组件选项。（如[vue-router](#) [↗] (opens new window)）
4. 添加全局实例方法，通过把它们添加到 `config.globalProperties` 上实现。
5. 一个库，提供自己的 API，同时提供上面提到的一个或多个功能。如 [vue-router](#) [↗] (opens new window)

编写插件

为了更好地理解如何创建自己的 Vue.js 版插件，我们将创建一个非常简化的插件版本，它显示 `i18n` 准备好的字符串。

每当这个插件被添加到应用程序中时，如果它是一个对象，就会调用 `install` 方法。如果它是一个 `function`，则函数本身将被调用。在这两种情况下——它都会收到两个参数：由 Vue 的 `createApp` 生成的 `app` 对象和用户传入的选项。

让我们从设置插件对象开始。建议在单独的文件中创建它并将其导出，如下所示，以保持包含的逻辑和分离的逻辑。

```
1. // plugins/i18n.js
2. export default {
3.   install: (app, options) => {
4.     // Plugin code goes here
5.   }
6. }
```

我们想要一个函数来翻译整个应用程序可用的键，因此我们将使用 `app.config.globalProperties` 暴露它。

该函数将接收一个 `key` 字符串，我们将使用它在用户提供的选项中查找转换后的字符串。

```

1. // plugins/i18n.js
2. export default {
3.   install: (app, options) => {
4.     app.config.globalProperties.$translate = key => {
5.       return key.split('.').reduce((o, i) => {
6.         if (o) return o[i]
7.       }, i18n)
8.     }
9.   }
10. }

```

我们假设用户使用插件时，将在 `options` 参数中传递一个包含翻译后的键的对象。我们的 `$translate` 函数将使用诸如 `greetings.hello` 之类的字符串，查看用户提供的配置内部并返回转换后的值 - 在这种情况下为 `Bonjour!`。

例如：

```

1. greetings: {
2.   hello: 'Bonjour!'
3. }

```

插件还允许我们使用 `inject` 为插件的用户提供功能或 `attribute`。例如，我们可以允许应用程序访问 `options` 参数以能够使用翻译对象。

```

1. // plugins/i18n.js
2. export default {
3.   install: (app, options) => {
4.     app.config.globalProperties.$translate = key => {
5.       return key.split('.').reduce((o, i) => {
6.         if (o) return o[i]
7.       }, i18n)
8.     }
9.
10.    app.provide('i18n', options)
11.  }
12. }

```

插件用户现在可以将 `inject[i18n]` 到他们的组件并访问该对象。

另外，由于我们可以访问 `app` 对象，因此插件可以使用所有其他功能，例如使用 `mixin` 和

`directive`。要了解有关 `createApp` 和应用程序实例的更多信息，请查看 [Application API 文档](#)。

```
1. // plugins/i18n.js
2. export default {
3.   install: (app, options) => {
4.     app.config.globalProperties.$translate = (key) => {
5.       return key.split('.')
6.         .reduce((o, i) => { if (o) return o[i] }, i18n)
7.     }
8.
9.     app.provide('i18n', options)
10.
11.    app.directive('my-directive', {
12.      bind (el, binding, vnode, oldVnode) {
13.        // some logic ...
14.      }
15.      ...
16.    })
17.
18.    app.mixin({
19.      created() {
20.        // some logic ...
21.      }
22.      ...
23.    })
24.  }
25. }
```

使用插件

在使用 `createApp()` 初始化 Vue 应用程序后，你可以通过调用 `use()` 方法将插件添加到你的应用程序中。

我们将使用在[编写插件](#)部分中创建的 `i18nPlugin` 进行演示。

`use()` 方法有两个参数。第一个是要安装的插件，在这种情况下为 `i18nPlugin`。

它还会自动阻止你多次使用同一插件，因此在同一插件上多次调用只会安装一次该插件。

第二个参数是可选的，并且取决于每个特定的插件。在演示 `i18nPlugin` 的情况下，它是带有转换后的字符串的对象。

INFO

如果你使用的是第三方插件（例如 `Vuex` 或 `Vue Router` ），请始终查看文档以了解特定插件期望作为第二个参数接收的内容。

```
1. import { createApp } from 'vue'
2. import Root from './App.vue'
3. import i18nPlugin from './plugins/i18n'
4.
5. const app = createApp(Root)
6. const i18nStrings = {
7.   greetings: {
8.     hi: 'Hallo!'
9.   }
10. }
11.
12. app.use(i18nPlugin, i18nStrings)
13. app.mount('#app')
```

[awesome-vue](#)  (opens new window) 集合了大量由社区贡献的插件和库。

- [响应性](#)
- [组合 API](#)
- [渲染机制和优化](#)
- [Vue 2 中的更改检测警告](#)

- [深入响应性原理](#)
- [响应式原理](#)
- [响应式计算和侦听](#)

深入响应性原理

现在是时候深入了！Vue 最独特的特性之一，是其非侵入性的响应性系统。数据模型是被代理的 JavaScript 对象。而当你修改它们时，视图会进行更新。这让状态管理非常简单直观，不过理解其工作原理同样重要，这样你可以避开一些常见的问题。在这个章节，我们将研究一下 Vue 响应性系统的底层的细节。

在 [Vue Mastery](#) 上免费观看关于深入响应性原理的视频。

什么是响应性

这个术语在程序设计中经常被提及，但这是什么意思呢？响应性是一种允许我们以声明式的方式去适应变化的一种编程范例。人们通常展示的典型例子，是一份 excel 电子表格（一个非常好的例子）。

Your browser does not support the video tag.

如果将数字 2 放在第一个单元格中，将数字 3 放在第二个单元格中并要求提供 SUM，则电子表格会将其计算出来给你。不要惊奇，同时，如果你更新第一个数字，SUM 也会自动更新。

JavaScript 通常不是这样工作的——如果我们想用 JavaScript 编写类似的内容：

```
1. var val1 = 2
2. var val2 = 3
3. var sum = val1 + val2
4.
5. // sum
6. // 5
7.
8. val1 = 3
9.
10. // sum
11. // 5
```

如果我们更新第一个值，sum 不会被修改。

那么我们如何用 JavaScript 实现这一点呢？

- 检测其中某一个值是否发生变化
- 用跟踪（track）函数修改值
- 用触发（trigger）函数更新为最新的值

Vue 如何追踪变化？

当把一个普通的 JavaScript 对象作为 `data` 选项传给应用或组件实例的时候，Vue 会使用带有 `getter` 和 `setter` 的处理程序遍历其所有 `property` 并将其转换为 `Proxy` [\(opens new window\)](#)。这是 ES6 仅有的特性，但是我们在 Vue 3 版本也使用了 `Object.defineProperty` 来支持 IE 浏览器。两者具有相同的 Surface API，但是 `Proxy` 版本更精简，同时提升了性能。

这部分需要稍微地了解下 `Proxy` [\(opens new window\)](#) 的某些知识！所以，让我们深入了解一下。关于 `Proxy` 的文献很多，但是你真正需要知道的是 `Proxy` 是一个包含另一个对象或函数并允许你对其进行拦截的对象。

我们是这样使用它的：`new Proxy(target, handler)`

```
1. const dinner = {
2.   meal: 'tacos'
3. }
4.
5. const handler = {
6.   get(target, prop) {
7.     return target[prop]
8.   }
9. }
10.
11. const proxy = new Proxy(dinner, handler)
12. console.log(proxy.meal)
13.
14. // tacos
```

好的，到目前为止，我们只是包装这个对象并返回它。很酷，但还不是那么有用。请注意，我们把对象包装在 `Proxy` 里的同时可以对其进行拦截。这种拦截被称为陷阱。

```
1. const dinner = {
2.   meal: 'tacos'
3. }
4.
5. const handler = {
6.   get(target, prop) {
7.     console.log('intercepted!')
8.     return target[prop]
9.   }
10. }
```

```

11.
12. const proxy = new Proxy(dinner, handler)
13. console.log(proxy.meal)
14.
15. // tacos

```

除了控制台日志，我们可以在这里做任何我们想做的事情。如果我们愿意，我们甚至可以不返回实际值。这就是为什么 Proxy 对于创建 API 如此强大。

此外，Proxy 还提供了另一个特性。我们不必像这样返回值：`target[prop]`，而是可以进一步使用一个名为 `Reflect` 的方法，它允许我们正确地执行 `this` 绑定，就像这样：

```

1. const dinner = {
2.   meal: 'tacos'
3. }
4.
5. const handler = {
6.   get(target, prop, receiver) {
7.     return Reflect.get(...arguments)
8.   }
9. }
10.
11. const proxy = new Proxy(dinner, handler)
12. console.log(proxy.meal)
13.
14. // tacos

```

我们之前提到过，为了有一个 API 能够在某些内容发生变化时更新最终值，我们必须在内容发生变化时设置新的值。我们在处理器，一个名为 `track` 的函数中执行此操作，该函数可以传入 `target` 和 `key` 两个参数。

```

1. const dinner = {
2.   meal: 'tacos'
3. }
4.
5. const handler = {
6.   get(target, prop, receiver) {
7.     track(target, prop)
8.     return Reflect.get(...arguments)
9.   }
10. }
11.

```

```
12. const proxy = new Proxy(dinner, handler)
13. console.log(proxy.meal)
14.
15. // tacos
```

最后，当某些内容发生改变时我们会设置新的值。为此，我们将通过触发这些更改来设置新 Proxy 的更改：

```
1. const dinner = {
2.   meal: 'tacos'
3. }
4.
5. const handler = {
6.   get(target, prop, receiver) {
7.     track(target, prop)
8.     return Reflect.get(...arguments)
9.   },
10.  set(target, key, value, receiver) {
11.    trigger(target, key)
12.    return Reflect.set(...arguments)
13.  }
14. }
15.
16. const proxy = new Proxy(dinner, handler)
17. console.log(proxy.meal)
18.
19. // tacos
```

还记得几段前的列表吗？现在我们有了一些关于 Vue 如何处理这些更改的答案：

- 当某个值发生变化时进行检测：我们不再需要这样做，因为 Proxy 允许我们拦截它
- 跟踪更改它的函数：我们在 Proxy 中的 getter 中执行此操作，称为 `effect`
- 触发函数以便它可以更新最终值：我们在 Proxy 中的 setter 中进行该操作，名为 `trigger`

Proxy 对象对于用户来说是不可见的，但是在内部，它们使 Vue 能够在 property 的值被访问或修改的情况下进行依赖跟踪和变更通知。从 Vue 3 开始，我们的响应性现在可以在[独立的包](#) (opens new window) 中使用。需要注意的是，记录转换后的数据对象时，浏览器控制台输出的格式会有所不同，因此你可能需要安装 `vue-devtools` (opens new window)，以提供一种更易于检查的界面。

Proxy 对象

Vue 在内部跟踪所有已被设置为响应式的对象，因此它始终会返回同一个对象的 Proxy 版本。

从响应式 Proxy 访问嵌套对象时，该对象在返回之前也被转换为 Proxy：

```
1. const handler = {
2.   get(target, prop, receiver) {
3.     track(target, prop)
4.     const value = Reflect.get(...arguments)
5.     if (isObject(value)) {
6.       return reactive(value)
7.     } else {
8.       return value
9.     }
10.  }
11.  // ...
12. }
```

Proxy vs 原始标识

Proxy 的使用确实引入了一个需要注意的新警告：在身份比较方面，被代理对象与原始对象不相等 (`===`)。例如：

```
1. const obj = {}
2. const wrapped = new Proxy(obj, handlers)
3.
4. console.log(obj === wrapped) // false
```

在大多数情况下，原始版本和包装版本的行为相同，但请注意，它们在依赖严格比对的操作下将是失败的，例如 `.filter()` 或 `.map()`。使用选项 API 时，这种警告不太可能出现，因为所有响应式都是从 `this` 访问的，并保证已经是 Proxy。

但是，当使用合成 API 显式创建响应式对象时，最佳做法是不要保留对原始对象的引用，而只使用响应式版本：

```
1. const obj = reactive({
2.   count: 0
3. }) // no reference to original
```

侦听器

每个组件实例都有一个相应的侦听器实例，该实例将在组件渲染期间把“触碰”的所有 property 记录为依赖项。之后，当触发依赖项的 setter 时，它会通知侦听器，从而使得组件重新渲染。

将对象作为数据传递给组件实例时，Vue 会将其转换为 Proxy。这个 Proxy 使 Vue 能够在 property 被访问或修改时执行依赖项跟踪和更改通知。每个 property 都被视为一个依赖项。

首次渲染后，组件将跟踪一组依赖列表——即在渲染过程中被访问的 property。反过来，组件就成为了其每个 property 的订阅者。当 Proxy 拦截到 set 操作时，该 property 将通知其所有订阅的组件重新渲染。

如果你使用的是 Vue2.x 及以下版本，你可能会对这些版本中存在的一些更改检测警告感兴趣，[在这里进行更详细的探讨](#)。

响应式原理

声明响应式状态

要为 JavaScript 对象创建响应式状态，可以使用 `reactive` 方法：

```
1. import { reactive } from 'vue'
2.
3. // 响应式状态
4. const state = reactive({
5.   count: 0
6. })
```

`reactive` 相当于 Vue 2.x 中的 `Vue.observable()` API，为避免与 RxJS 中的 `observables` 混淆因此对其重命名。该 API 返回一个响应式的对象状态。该响应式转换是“深度转换”——它会影响到嵌套对象传递的所有 `property`。

Vue 中响应式状态的基本用例是我们可以渲染期间使用它。因为依赖跟踪的关系，当响应式状态改变时视图会自动更新。

这就是 Vue 响应式系统的本质。当从组件中的 `data()` 返回一个对象时，它在内部交由 `reactive()` 使其成为响应式对象。模板会被编译成能够使用这些响应式 `property` 的渲染函数。

在[响应式基础 API](#) 章节你可以学习更多关于 `reactive` 的内容。

创建独立的响应式值作为 `refs`

想象一下，我们有一个独立的原始值（例如，一个字符串），我们想让它变成响应式的。当然，我们可以创建一个拥有相同字符串 `property` 的对象，并将其传递给 `reactive`。Vue 为我们提供了一个可以做相同事情的方法 — `ref`：

```
1. import { ref } from 'vue'
2.
3. const count = ref(0)
```

`ref` 会返回一个可变的响应式对象，该对象作为它的内部值——一个响应式的引用，这就是名称的来源。此对象只包含一个名为 `value` 的 `property`：

```
1. import { ref } from 'vue'
2.
3. const count = ref(0)
4. console.log(count.value) // 0
5.
6. count.value++
7. console.log(count.value) // 1
```

Ref 展开

当 `ref` 作为渲染上下文（从 `setup()` 中返回的对象）上的 `property` 返回并可以在模板中被访问时，它将自动展开为内部值。不需要在模板中追加 `.value`：

```
1. <template>
2.   <div>
3.     <span>{{ count }}</span>
4.     <button @click="count ++">Increment count</button>
5.   </div>
6. </template>
7.
8. <script>
9.   import { ref } from 'vue'
10.  export default {
11.    setup() {
12.      const count = ref(0)
13.      return {
14.        count
15.      }
16.    }
17.  }
18. </script>
```

访问响应式对象

当 `ref` 作为响应式对象的 `property` 被访问或更改时，为使其行为类似于普通 `property`，它会自动展开内部值：

```
1. const count = ref(0)
2. const state = reactive({
3.   count
4. })
```

```

5.
6. console.log(state.count) // 0
7.
8. state.count = 1
9. console.log(count.value) // 1

```

如果将新的 ref 赋值给现有 ref 的 property, 将会替换旧的 ref:

```

1. const otherCount = ref(2)
2.
3. state.count = otherCount
4. console.log(state.count) // 2
5. console.log(count.value) // 1

```

Ref 展开仅发生在被响应式 `Object` 嵌套的时候。当从 `Array` 或原生集合类型如 `Map` [\(opens new window\)](#) 访问 ref 时, 不会进行展开:

```

1. const books = reactive([ref('Vue 3 Guide')])
2. // 这里需要 .value
3. console.log(books[0].value)
4.
5. const map = reactive(new Map([[ 'count', ref(0) ]]))
6. // 这里需要 .value
7. console.log(map.get('count').value)

```

响应式状态解构

当我们想使用大型响应式对象的一些 property 时, 可能很想使用 `ES6 解构` [\(opens new window\)](#) 来获取我们想要的 property:

```

1. import { reactive } from 'vue'
2.
3. const book = reactive({
4.   author: 'Vue Team',
5.   year: '2020',
6.   title: 'Vue 3 Guide',
7.   description: 'You are reading this book right now ;)',
8.   price: 'free'
9. })
10.

```

```
11. let { author, title } = book
```

遗憾的是，使用解构的两个 property 的响应式都会丢失。对于这种情况，我们需要将我们的响应式对象转换为的一组 ref。这些 ref 将保留与源对象的响应式关联：

```
1. import { reactive, toRefs } from 'vue'
2.
3. const book = reactive({
4.   author: 'Vue Team',
5.   year: '2020',
6.   title: 'Vue 3 Guide',
7.   description: 'You are reading this book right now ;)',
8.   price: 'free'
9. })
10.
11. let { author, title } = toRefs(book)
12.
13. title.value = 'Vue 3 Detailed Guide' // 我们需要使用 .value 作为标题, 现在是 ref
14. console.log(book.title) // 'Vue 3 Detailed Guide'
```

你可以在 [Refs API](#) 部分中了解更多有关 `refs` 的信息

使用 `readonly` 防止更改响应式对象

有时我们想跟踪响应式对象（`ref` 或 `reactive`）的变化，但我们也希望防止在应用程序的某个位置更改它。例如，当我们有一个被 `provide` 的响应式对象时，我们不想让它在注入的时候被改变。为此，我们可以基于原始对象创建一个只读的 Proxy 对象：

```
1. import { reactive, readonly } from 'vue'
2.
3. const original = reactive({ count: 0 })
4.
5. const copy = readonly(original)
6.
7. // 在copy上转换original 会触发侦听器依赖
8.
9. original.count++
10.
11. // 转换copy 将导致失败并导致警告
    copy.count++ // 警告: "Set operation on key 'count' failed: target is
12. readonly."
```

响应式计算和侦听

本节使用单文件组件语法作为代码示例

计算值

有时我们需要依赖于其他状态的状态——在 Vue 中，这是用组件计算属性处理的，以直接创建计算值，我们可以使用 `computed` 方法：它接受 getter 函数并为 getter 返回的值返回一个不可变的响应式 `ref` 对象。

```
1. const count = ref(1)
2. const plusOne = computed(() => count.value++)
3.
4. console.log(plusOne.value) // 2
5.
6. plusOne.value++ // error
```

或者，它可以使用一个带有 `get` 和 `set` 函数的对象来创建一个可写的 `ref` 对象。

```
1. const count = ref(1)
2. const plusOne = computed({
3.   get: () => count.value + 1,
4.   set: val => {
5.     count.value = val - 1
6.   }
7. })
8.
9. plusOne.value = 1
10. console.log(count.value) // 0
```

watchEffect

为了根据反应状态自动应用和重新应用副作用，我们可以使用 `watchEffect` 方法。它立即执行传入的一个函数，同时响应式追踪其依赖，并在其依赖变更时重新运行该函数。

```
1. const count = ref(0)
2.
3. watchEffect(() => console.log(count.value))
4. // -> logs 0
```

```

5.
6.  setTimeout(() => {
7.    count.value++
8.    // -> logs 1
9.  }, 100)

```

停止侦听

当 `watchEffect` 在组件的 `setup()` 函数或生命周期钩子被调用时，侦听器会被链接到该组件的生命周期，并在组件卸载时自动停止。

在一些情况下，也可以显式调用返回值以停止侦听：

```

1.  const stop = watchEffect(() => {
2.    /* ... */
3.  })
4.
5.  // later
6.  stop()

```

清除副作用

有时副作用函数会执行一些异步的副作用，这些响应需要在其失效时清除（即完成之前状态已改变了）。所以侦听副作用传入的函数可以接收一个 `onInvalidate` 函数作入参，用来注册清理失效时的回调。当以下情况发生时，这个失效回调会被触发：

- 副作用即将重新执行时
- 侦听器被停止（如果在 `setup()` 或生命周期钩子函数中使用了 `watchEffect`，则在组件卸载时）

```

1.  watchEffect(onInvalidate => {
2.    const token = performAsyncOperation(id.value)
3.    onInvalidate(() => {
4.      // id has changed or watcher is stopped.
5.      // invalidate previously pending async operation
6.      token.cancel()
7.    })
8.  })

```

我们之所以是通过传入一个函数去注册失效回调，而不是从回调返回它，是因为返回值对于异步错误处理很重要。

在执行数据请求时，副作用函数往往是一个异步函数：

```
1. const data = ref(null)
2. watchEffect(async onInvalidate => {
3.   onInvalidate(() => {...}) // 我们在Promise解析之前注册清除函数
4.   data.value = await fetchData(props.id)
5. })
```

我们知道异步函数都会隐式地返回一个 Promise，但是清理函数必须要在 Promise 被 resolve 之前被注册。另外，Vue 依赖这个返回的 Promise 来自动处理 Promise 链上的潜在错误。

副作用刷新时机

Vue 的响应式系统会缓存副作用函数，并异步地刷新它们，这样可以避免同一个“tick”中多个状态改变导致的不必要的重复调用。在核心的具体实现中，组件的 `update` 函数也是一个被侦听的副作用。当一个用户定义的副作用函数进入队列时，默认情况下，会在所有的组件 `update` 前执行：

```
1. <template>
2.   <div>{{ count }}</div>
3. </template>
4.
5. <script>
6.   export default {
7.     setup() {
8.       const count = ref(0)
9.
10.      watchEffect(() => {
11.        console.log(count.value)
12.      })
13.
14.      return {
15.        count
16.      }
17.    }
18.  }
19. </script>
```

在这个例子中：

- `count` 会在初始运行时同步打印出来
- 更改 `count` 时，将在组件更新前执行副作用。

如果需要在组件更新后重新运行侦听器副作用，我们可以传递带有 `flush` 选项的附加 `options` 对象（默认为 `'pre'`）：

```

1.
2. // fire before component updates
3. watchEffect(
4.   () => {
5.     /* ... */
6.   },
7.   {
8.     flush: 'post'
9.   }
10. )

```

`flush` 选项还接受 `sync`，这将强制效果始终同步触发。然而，这是低效的，应该很少需要。

侦听器调试

`onTrack` 和 `onTrigger` 选项可用于调试侦听器的行为。

- 当响应式 property 或 ref 作为依赖项被追踪时，将调用 `onTrack`
- 当依赖项变更导致副作用被触发时，将调用 `onTrigger`

这两个回调都将接收到一个包含有关所依赖项信息的调试器事件。建议在以下回调中编写

`debugger` 语句来检查依赖关系：

```

1. watchEffect(
2.   () => {
3.     /* 副作用 */
4.   },
5.   {
6.     onTrigger(e) {
7.       debugger
8.     }
9.   }
10. )

```

`onTrack` 和 `onTrigger` 只能在开发模式下工作。

`watch`

`watch` API 完全等同于组件侦听器 property。`watch` 需要侦听特定的数据源，并在回调函

数中执行副作用。默认情况下，它也是惰性的，即只有当被侦听的源发生变化时才执行回调。

- 与 `watchEffect` 比较，`watch` 允许我们：
 - 懒执行副作用；
 - 更具体地说明什么状态应该触发侦听器重新运行；
 - 访问侦听状态变化前后的值。

侦听单个数据源

侦听器数据源可以是返回值的 `getter` 函数，也可以直接是 `ref`：

```

1. // 侦听一个 getter
2. const state = reactive({ count: 0 })
3. watch(
4.   () => state.count,
5.   (count, prevCount) => {
6.     /* ... */
7.   }
8. )
9.
10. // 直接侦听ref
11. const count = ref(0)
12. watch(count, (count, prevCount) => {
13.   /* ... */
14. })

```

侦听多个数据源

侦听器还可以使用数组同时侦听多个源：

```

1. watch([fooRef, barRef], ([foo, bar], [prevFoo, prevBar]) => {
2.   /* ... */
3. })

```

与 `watchEffect` 共享的行为

`watch` 与 `watchEffect` 共享停止侦听，清除副作用（相应地 `onInvalidate` 会作为回调的第三个参数传入）、副作用刷新时机和侦听器调试行为。

- [介绍](#)
- [Setup](#)
- [生命周期钩子](#)
- [提供/注入](#)
- [模板引用](#)

介绍

什么是 Composition API？

提示

在阅读文档之前，你应该已经熟悉了这两个 [Vue 基础](#)和[创建组件](#)。

在 [Vue Mastery](#) 上观看关于组合 API 的免费视频。

通过创建 Vue 组件，我们可以将接口的可重复部分及其功能提取到可重用的代码段中。仅此一项就可以使我们的应用程序在可维护性和灵活性方面走得更远。然而，我们的经验已经证明，光靠这一点可能是不够的，尤其是当你的应用程序变得非常大的时候——想想几百个组件。在处理如此大的应用程序时，共享和重用代码变得尤为重要。

假设在我们的应用程序中，我们有一个视图来显示某个用户的仓库列表。除此之外，我们还希望应用搜索和筛选功能。处理此视图的组件可能如下所示：

```
1. // src/components/UserRepositories.vue
2.
3. export default {
4.   components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
5.   props: {
6.     user: { type: String }
7.   },
8.   data () {
9.     return {
10.      repositories: [], // 1
11.      filters: { ... }, // 3
12.      searchQuery: '' // 2
13.    }
14.  },
15.  computed: {
16.    filteredRepositories () { ... }, // 3
17.    repositoriesMatchingSearchQuery () { ... }, // 2
18.  },
19.  watch: {
20.    user: 'getUserRepositories' // 1
21.  },
22.  methods: {
23.    getUserRepositories () {
```

```
24.     // 使用 `this.user` 获取用户仓库
25.     }, // 1
26.     updateFilters () { ... }, // 3
27.   },
28.   mounted () {
29.     this.getUserRepositories() // 1
30.   }
31. }
```

该组件有以下几个职责：

1. 从假定的外部 API 获取该用户名的仓库，并在用户更改时刷新它
2. 使用 `searchQuery` 字符串搜索存储库
3. 使用 `filters` 对象筛选仓库

用组件的选项（`data`、`computed`、`methods`、`watch`）组织逻辑在大多数情况下都有效。然而，当我们的组件变得更大时，逻辑关注点的列表也会增长。这可能会导致组件难以阅读和理解，尤其是对于那些一开始就没有编写这些组件的人来说。

一个大型组件的示例，其中逻辑关注点是按颜色分组。

这种碎片化使得理解和维护复杂组件变得困难。选项的分离掩盖了潜在的逻辑问题。此外，在处理单个逻辑关注点时，我们必须不断地“跳转”相关代码的选项块。

如果我们能够将与同一个逻辑关注点相关的代码配置在一起会更好。而这正是 Composition API 使我们能够做到的。

Composition API 基础

既然我们知道了为什么，我们就可以知道怎么做。为了开始使用 Composition api，我们首先需要 一个可以实际使用它的地方。在 Vue 组件中，我们将此位置称为 `setup`。

`setup` 组件选项

观看 [Vue Mastery 上的免费 setup 视频](#)。

新的 `setup` 组件选项在创建组件之前执行，一旦 `props` 被解析，并充当合成 API 的入口点。

WARNING

由于在执行 `setup` 时尚未创建组件实例，因此在 `setup` 选项中没有 `this`。这意味着，除了 `props` 之外，你将无法访问组件中声明的任何属性—本地状态、计算属性或方法。

`setup` 选项应该是一个接受 `props` 和 `context` 的函数，我们将在[稍后](#)讨论。此外，我们从 `setup` 返回的所有内容都将暴露给组件的其余部分（计算属性、方法、生命周期钩子等等）以及组件的模板。

让我们添加 `setup` 到我们的组件中：

```
1. // src/components/UserRepositories.vue
2.
3. export default {
4.   components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
5.   props: {
6.     user: { type: String }
7.   },
8.   setup(props) {
9.     console.log(props) // { user: '' }
10.
11.     return {} // 这里返回的任何内容都可以用于组件的其余部分
12.   }
```

```

13. // 组件的“其余部分”
14. }

```

现在让我们从提取第一个逻辑关注点开始（在原始代码段中标记为“1”）。

1. 从假定的外部 API 获取该用户名的仓库，并在用户更改时刷新它

我们将从最明显的部分开始：

- 仓库列表
- 更新仓库列表的函数
- 返回列表和函数，以便其他组件选项可以访问它们

```

1. // src/components/UserRepositories.vue `setup` function
2. import { fetchUserRepositories } from '@api/repositories'
3.
4. // 在我们的组件内
5. setup (props) {
6.   let repositories = []
7.   const getUserRepositories = async () => {
8.     repositories = await fetchUserRepositories(props.user)
9.   }
10.
11.   return {
12.     repositories,
13.     getUserRepositories // 返回的函数与方法的行为相同
14.   }
15. }

```

这是我们的出发点，但它还不能工作，因为我们的 `repositories` 变量不是被动的。这意味着从用户的角度来看，仓库列表将保持为空。我们来解决这个问题！

带 `ref` 的响应式变量

在 Vue 3.0 中，我们可以通过一个新的 `ref` 函数使任何响应式变量在任何地方起作用，如下所示：

```

1. import { ref } from 'vue'
2.
3. const counter = ref(0)

```

`ref` 接受参数并返回它包装在具有 `value` property 的对象中，然后可以使用该 property

访问或更改响应式变量的值：

```

1. import { ref } from 'vue'
2.
3. const counter = ref(0)
4.
5. console.log(counter) // { value: 0 }
6. console.log(counter.value) // 0
7.
8. counter.value++
9. console.log(counter.value) // 1

```

在对象中包装值似乎不必要，但在 JavaScript 中保持不同数据类型的行为统一是必需的。这是因为在 JavaScript 中，`Number` 或 `String` 等基本类型是通过值传递的，而不是通过引用传递的：



在任何值周围都有一个包装器对象，这样我们就可以在整个应用程序中安全地传递它，而不必担心在某个地方失去它的响应式。

提示

换句话说，`ref` 对我们的值创建了一个响应式引用。使用引用的概念将在整个 Composition API 中经常使用。

回到我们的例子，让我们创建一个响应式的 `repositories` 变量：

```

1. // src/components/UserRepositories.vue `setup` function
2. import { fetchUserRepositories } from '@api/repositories'
3. import { ref } from 'vue'
4.
5. // in our component

```



```
6. setup (props) {
7.   const repositories = ref([])
8.   const getUserRepositories = async () => {
9.     repositories.value = await fetchUserRepositories(props.user)
10.  }
11.
12.  return {
13.    repositories,
14.    getUserRepositories
15.  }
16. }
```

完成！现在，每当我们调用 `getUserRepositories` 时，`repositories` 都将发生变化，视图将更新以反映更改。我们的组件现在应该如下所示：

```
1. // src/components/UserRepositories.vue
2. import { fetchUserRepositories } from '@api/repositories'
3. import { ref } from 'vue'
4.
5. export default {
6.   components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
7.   props: {
8.     user: { type: String }
9.   },
10.  setup (props) {
11.    const repositories = ref([])
12.    const getUserRepositories = async () => {
13.      repositories.value = await fetchUserRepositories(props.user)
14.    }
15.
16.    return {
17.      repositories,
18.      getUserRepositories
19.    }
20.  },
21.  data () {
22.    return {
23.      filters: { ... }, // 3
24.      searchQuery: '' // 2
25.    }
26.  },
27.  computed: {
```

```

28.   filteredRepositories () { ... }, // 3
29.   repositoriesMatchingSearchQuery () { ... }, // 2
30. },
31. watch: {
32.   user: 'getUserRepositories' // 1
33. },
34. methods: {
35.   updateFilters () { ... }, // 3
36. },
37. mounted () {
38.   this.getUserRepositories() // 1
39. }
40. }

```

我们已经将第一个逻辑关注点中的几个部分移到了 `setup` 方法中，它们彼此非常接近。剩下的就是在 `mounted` 钩子中调用 `getUserRepositories`，并设置一个监听器，以便在 `user` prop 发生变化时执行此操作。

我们将从生命周期钩子开始。

生命周期钩子注册内部 `setup`

为了使 Composition API 的特性与选项 API 相比更加完整，我们还需要一种在 `setup` 中注册生命周期钩子的方法。这要归功于从 Vue 导出的几个新函数。Composition API 上的生命周期钩子与选项 API 的名称相同，但前缀为 `on`：即 `mounted` 看起来像 `onMounted`。

这些函数接受在组件调用钩子时将执行的回调。

让我们将其添加到 `setup` 函数中：

```

1. // src/components/UserRepositories.vue `setup` function
2. import { fetchUserRepositories } from '@api/repositories'
3. import { ref, onMounted } from 'vue'
4.
5. // in our component
6. setup (props) {
7.   const repositories = ref([])
8.   const getUserRepositories = async () => {
9.     repositories.value = await fetchUserRepositories(props.user)
10.  }
11.
12.   onMounted(getUserRepositories) // on `mounted` call `getUserRepositories`
13.

```

```

14.   return {
15.     repositories,
16.     getUserRepositories
17.   }
18. }

```

现在我们需要对 `user` prop 所做的更改做出反应。为此，我们将使用独立的 `watch` 函数。

`watch` 响应式更改

就像我们如何使用 `watch` 选项在组件内的 `user` property 上设置侦听器一样，我们也可以使用从 Vue 导入的 `watch` 函数执行相同的操作。它接受 3 个参数：

- 一个响应式引用或我们想要侦听的 getter 函数
- 一个回调
- 可选的配置选项

下面让我们快速了解一下它是如何工作的

```

1. import { ref, watch } from 'vue'
2.
3. const counter = ref(0)
4. watch(counter, (newValue, oldValue) => {
5.   console.log('The new counter value is: ' + counter.value)
6. })

```

例如，每当 `counter` 被修改时 `counter.value=5`，`watch` 将触发并执行回调（第二个参数），在本例中，它将把 `'The new counter value is:5'` 记录到我们的控制台中。

以下是等效的选项 API：

```

1. export default {
2.   data() {
3.     return {
4.       counter: 0
5.     }
6.   },
7.   watch: {
8.     counter(newValue, oldValue) {
9.       console.log('The new counter value is: ' + this.counter)
10.    }
11.  }

```

```
12. }
```

有关 `watch` 的详细信息，请参阅我们的[深入指南](#)。

现在我们将其应用到我们的示例中：

```
1. // src/components/UserRepositories.vue `setup` function
2. import { fetchUserRepositories } from '@api/repositories'
3. import { ref, onMounted, watch, toRefs } from 'vue'
4.
5. // 在我们组件中
6. setup (props) {
7.   // 使用 `toRefs` 创建对prop的 `user` property 的响应式引用
8.   const { user } = toRefs(props)
9.
10.  const repositories = ref([])
11.  const getUserRepositories = async () => {
12.    // 更新`prop.user` 到 `user.value`访问引用值
13.    repositories.value = await fetchUserRepositories(user.value)
14.  }
15.
16.  onMounted(getUserRepositories)
17.
18.  // 在用户prop的响应式引用上设置一个侦听器
19.  watch(user, getUserRepositories)
20.
21.  return {
22.    repositories,
23.    getUserRepositories
24.  }
25. }
```

你可能已经注意到在我们的 `setup` 的顶部使用了 `toRefs` 。这是为了确保我们的侦听器能够对 `user` prop 所做的更改做出反应。

有了这些变化，我们就把第一个逻辑关注点移到了一个地方。我们现在可以对第二个关注点执行相同的操作—基于 `searchQuery` 进行过滤，这次是使用计算属性。

独立的 `computed` 属性

与 `ref` 和 `watch` 类似，也可以使用从 `vue` 导入的 `computed` 函数在 `vue` 组件外部创建计算属性。让我们回到我们的 `counter` 例子：

```
1. import { ref, computed } from 'vue'
2.
3. const counter = ref(0)
4. const twiceTheCounter = computed(() => counter.value * 2)
5.
6. counter.value++
7. console.log(counter.value) // 1
8. console.log(twiceTheCounter.value) // 2
```

在这里，`computed` 函数返回一个作为 `computed` 的第一个参数传递的 `getter` 类回调的输出。为了访问新创建的计算变量的 `value`，我们需要像使用 `ref` 一样使用 `.value` property。

让我们将搜索功能移到 `setup` 中：

```
1. // src/components/UserRepositories.vue `setup` function
2. import { fetchUserRepositories } from '@api/repositories'
3. import { ref, onMounted, watch, toRefs, computed } from 'vue'
4.
5. // in our component
6. setup (props) {
7.   // 使用 `toRefs` 创建对props的 `user` property 的响应式引用
8.   const { user } = toRefs(props)
9.
10.  const repositories = ref([])
11.  const getUserRepositories = async () => {
12.    // 更新 `props.user` 到 `user.value` 访问引用值
13.    repositories.value = await fetchUserRepositories(user.value)
14.  }
15.
16.  onMounted(getUserRepositories)
17.
18.  // 在用户prop的响应式引用上设置一个侦听器
19.  watch(user, getUserRepositories)
20.
21.  const searchQuery = ref('')
22.  const repositoriesMatchingSearchQuery = computed(() => {
23.    return repositories.value.filter(
24.      repository => repository.name.includes(searchQuery.value)
25.    )
26.  })
27.
```

```

28.   return {
29.     repositories,
30.     getUserRepositories,
31.     searchQuery,
32.     repositoriesMatchingSearchQuery
33.   }
34. }

```

对于其他的逻辑关注点我们也可以这样做，但是你可能已经在问这个问题了——这不就是把代码移到 `setup` 选项并使它变得非常大吗？嗯，那是真的。这就是为什么在继续其他任务之前，我们将首先将上述代码提取到一个独立的组合函数。让我们从创建 `useUserRepositories` 开始：

```

1. // src/composables/useUserRepositories.js
2.
3. import { fetchUserRepositories } from '@api/repositories'
4. import { ref, onMounted, watch } from 'vue'
5.
6. export default function useUserRepositories(user) {
7.   const repositories = ref([])
8.   const getUserRepositories = async () => {
9.     repositories.value = await fetchUserRepositories(user.value)
10.  }
11.
12.   onMounted(getUserRepositories)
13.   watch(user, getUserRepositories)
14.
15.   return {
16.     repositories,
17.     getUserRepositories
18.   }
19. }

```

然后是搜索功能：

```

1. // src/composables/useRepositoryNameSearch.js
2.
3. import { ref, computed } from 'vue'
4.
5. export default function useRepositoryNameSearch(repositories) {
6.   const searchQuery = ref('')
7.   const repositoriesMatchingSearchQuery = computed(() => {
8.     return repositories.value.filter(repository => {

```

```
9.     return repository.name.includes(searchQuery.value)
10.   })
11. })
12.
13. return {
14.   searchQuery,
15.   repositoriesMatchingSearchQuery
16. }
17. }
```

现在在单独的文件中有了这两个功能，我们就可以开始在组件中使用它们了。以下是如何做到这一点：

```
1. // src/components/UserRepositories.vue
2. import useUserRepositories from '@/composables/useUserRepositories'
3. import useRepositoryNameSearch from '@/composables/useRepositoryNameSearch'
4. import { toRefs } from 'vue'
5.
6. export default {
7.   components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
8.   props: {
9.     user: { type: String }
10.  },
11.  setup (props) {
12.    const { user } = toRefs(props)
13.
14.    const { repositories, getUserRepositories } = useUserRepositories(user)
15.
16.    const {
17.      searchQuery,
18.      repositoriesMatchingSearchQuery
19.    } = useRepositoryNameSearch(repositories)
20.
21.    return {
22.      // 因为我们并不关心未经过滤的仓库
23.      // 我们可以在 `repositories` 名称下暴露过滤后的结果
24.      repositories: repositoriesMatchingSearchQuery,
25.      getUserRepositories,
26.      searchQuery,
27.    }
28.  },
29.  data () {
30.    return {
```

```

31.     filters: { ... }, // 3
32.   }
33. },
34. computed: {
35.   filteredRepositories () { ... }, // 3
36. },
37. methods: {
38.   updateFilters () { ... }, // 3
39. }
40. }

```

此时，你可能已经知道了这个练习，所以让我们跳到最后，迁移剩余的过滤功能。我们不需要深入了解实现细节，因为这不是本指南的重点。

```

1. // src/components/UserRepositories.vue
2. import { toRefs } from 'vue'
3. import useUserRepositories from '@/composables/useUserRepositories'
4. import useRepositoryNameSearch from '@/composables/useRepositoryNameSearch'
5. import useRepositoryFilters from '@/composables/useRepositoryFilters'
6.
7. export default {
8.   components: { RepositoriesFilters, RepositoriesSortBy, RepositoriesList },
9.   props: {
10.    user: { type: String }
11.  },
12.  setup(props) {
13.    const { user } = toRefs(props)
14.
15.    const { repositories, getUserRepositories } = useUserRepositories(user)
16.
17.    const {
18.      searchQuery,
19.      repositoriesMatchingSearchQuery
20.    } = useRepositoryNameSearch(repositories)
21.
22.    const {
23.      filters,
24.      updateFilters,
25.      filteredRepositories
26.    } = useRepositoryFilters(repositoriesMatchingSearchQuery)
27.
28.    return {

```



```
29.      // 因为我们并不关心未经过滤的仓库
30.      // 我们可以在 `repositories` 名称下暴露过滤后的结果
31.      repositories: filteredRepositories,
32.      getUserRepositories,
33.      searchQuery,
34.      filters,
35.      updateFilters
36.    }
37.  }
38. }
```

我们完成了！

请记住，我们只触及了 Composition API 的表面以及它允许我们做什么。要了解更多信息，请参阅深入指南。

Setup

本节使用[单文件组件](#)代码示例的语法

本指南假定你已经阅读了[组合式 API 简介](#)和[响应性原理](#)。如果你不熟悉组合式 API，请先阅读这篇文章。

参数

使用 `setup` 函数时，它将接受两个参数：

1. `props`
2. `context`

让我们更深入地研究如何使用每个参数。

Props

`setup` 函数中的第一个参数是 `props`。正如在一个标准组件中所期望的那样，`setup` 函数中的 `props` 是响应式的，当传入新的 `prop` 时，它将被更新。

```
1. // MyBook.vue
2.
3. export default {
4.   props: {
5.     title: String
6.   },
7.   setup(props) {
8.     console.log(props.title)
9.   }
10. }
```

WARNING

但是，因为 `props` 是响应式的，你不能使用 **ES6** 解构，因为它会消除 `prop` 的响应性。

如果需要解构 `prop`，可以通过使用 `setup` 函数中的 `toRefs` 来安全地完成此操作。

```
1. // MyBook.vue
2.
3. import { toRefs } from 'vue'
4.
```

```

5.  setup(props) {
6.      const { title } = toRefs(props)
7.
8.      console.log(title.value)
9.  }

```

上下文

传递给 `setup` 函数的第二个参数是 `context`。`context` 是一个普通的 JavaScript 对象，它暴露三个组件的 property:

```

1.  // MyBook.vue
2.
3.  export default {
4.      setup(props, context) {
5.          // Attribute (非响应式对象)
6.          console.log(context.attrs)
7.
8.          // 插槽 (非响应式对象)
9.          console.log(context.slots)
10.
11.         // 触发事件 (方法)
12.         console.log(context.emit)
13.     }
14. }

```

`context` 是一个普通的 JavaScript 对象，也就是说，它不是响应式的，这意味着你可以安全地对 `context` 使用 ES6 解构。

```

1.  // MyBook.vue
2.  export default {
3.      setup(props, { attrs, slots, emit }) {
4.          ...
5.      }
6.  }

```

`attrs` 和 `slots` 是有状态的对象，它们总是会随组件本身的更新而更新。这意味着你应该避免对它们进行解构，并始终以 `attrs.x` 或 `slots.x` 的方式引用 property。请注意，与 `props` 不同，`attrs` 和 `slots` 是非响应式的。如果你打算根据 `attrs` 或 `slots` 更改应用副作用，那么应该在 `onUpdated` 生命周期钩子中执行此操作。

访问组件的 property

执行 `setup` 时，组件实例尚未被创建。因此，你只能访问以下 property：

- `props`
- `attrs`
- `slots`
- `emit`

换句话说，你将无法访问以下组件选项：

- `data`
- `computed`
- `methods`

结合模板使用

如果 `setup` 返回一个对象，则可以在组件的模板中像传递给 `setup` 的 `props` property 一样访问该对象的 property：

```
1. <!-- MyBook.vue -->
2. <template>
3.   <div>{{ readersNumber }} {{ book.title }}</div>
4. </template>
5.
6. <script>
7.   import { ref, reactive } from 'vue'
8.
9.   export default {
10.    setup() {
11.      const readersNumber = ref(0)
12.      const book = reactive({ title: 'Vue 3 Guide' })
13.
14.      // expose to template
15.      return {
16.        readersNumber,
17.        book
18.      }
19.    }
20.  }
21. </script>
```

注意，从 `setup` 返回的 `refs` 在模板中访问时是[被自动解开的](#)，因此不应在模板中使用 `.value`。

使用渲染函数

`setup` 还可以返回一个渲染函数，该函数可以直接使用在同一作用域中声明的响应式状态：

```
1. // MyBook.vue
2.
3. import { h, ref, reactive } from 'vue'
4.
5. export default {
6.   setup() {
7.     const readersNumber = ref(0)
8.     const book = reactive({ title: 'Vue 3 Guide' })
9.     // Please note that we need to explicitly expose ref value here
10.    return () => h('div', [readersNumber.value, book.title])
11.  }
12. }
```

使用 `this`

在 `setup()` 内部，`this` 不会是该活跃实例的引用，因为 `setup()` 是在解析其它组件选项之前被调用的，所以 `setup()` 内部的 `this` 的行为与其它选项中的 `this` 完全不同。这在和其它选项式 API 一起使用 `setup()` 时可能会导致混淆。

生命周期钩子

本指南假定你已经阅读了 [Composition API 简介](#) 和 [响应式基础](#)。如果你不熟悉组合 API，请先阅读这篇文章。

在 [Vue Mastery](#) 上观看关于生命周期钩子的免费视频

你可以通过在生命周期钩子前面加上 “on” 来访问组件的生命周期钩子。

下表包含如何在 `setup ()` 内部调用生命周期钩子：

选项 API	Hook inside <code>setup</code>
<code>beforeCreate</code>	Not needed
<code>created</code>	Not needed
<code>beforeMount</code>	<code>onBeforeMount</code>
<code>mounted</code>	<code>onMounted</code>
<code>beforeUpdate</code>	<code>onBeforeUpdate</code>
<code>updated</code>	<code>onUpdated</code>
<code>beforeUnmount</code>	<code>onBeforeUnmount</code>
<code>unmounted</code>	<code>onUnmounted</code>
<code>errorCaptured</code>	<code>onErrorCaptured</code>
<code>renderTracked</code>	<code>onRenderTracked</code>
<code>renderTriggered</code>	<code>onRenderTriggered</code>

TIP

因为 `setup` 是围绕 `beforeCreate` 和 `created` 生命周期钩子运行的，所以不需要显式地定义它们。换句话说，在这些钩子中编写的任何代码都应该直接在 `setup` 函数中编写。

这些函数接受一个回调函数，当钩子被组件调用时将会被执行：

```
1. // MyBook.vue
2.
3. export default {
4.   setup() {
5.     // mounted
6.     onMounted(() => {
7.       console.log('Component is mounted!')
8.     })
9.   }
10. }
```

提供/注入

本指南假定你已经阅读了 [Provide / Inject](#)、[Composition API Introduction](#) 和[响应式基础](#)。如果你不熟悉组合 API，请先阅读这篇文章。

我们也可以在 Composition API 中使用 `provide/inject`。两者都只能在当前活动实例的 `setup()` 期间调用。

设想场景

假设我们要重写以下代码，其中包含一个 `MyMap` 组件，该组件使用 Composition API 为 `MyMarker` 组件提供用户的位置。

```
1. <!-- src/components/MyMap.vue -->
2. <template>
3.   <MyMarker />
4. </template>
5.
6. <script>
7.   import MyMarker from './MyMarker.vue'
8.
9.   export default {
10.     components: {
11.       MyMarker
12.     },
13.     provide: {
14.       location: 'North Pole',
15.       geolocation: {
16.         longitude: 90,
17.         latitude: 135
18.       }
19.     }
20.   }
21. </script>
```

```
1. <!-- src/components/MyMarker.vue -->
2. <script>
3.   export default {
4.     inject: ['location', 'geolocation']
5.   }
```

```
6. </script>
```

使用 Provide

在 `setup()` 中使用 `provide` 时，我们首先从 `vue` 显式导入 `provide` 方法。这使我们能够调用 `provide` 时来定义每个 property。

`provide` 函数允许你通过两个参数定义 property：

1. property 的 name (`<String>` 类型)
2. property 的 value

使用 `MyMap` 组件，我们提供的值可以按如下方式重构：

```
1. <!-- src/components/MyMap.vue -->
2. <template>
3.   <MyMarker />
4. </template>
5.
6. <script>
7. import { provide } from 'vue'
8. import MyMarker from './MyMarker.vue
9.
10. export default {
11.   components: {
12.     MyMarker
13.   },
14.   setup() {
15.     provide('location', 'North Pole')
16.     provide('geolocation', {
17.       longitude: 90,
18.       latitude: 135
19.     })
20.   }
21. }
22. </script>
```

使用注入

在 `setup()` 中使用 `inject` 时，还需要从 `vue` 显式导入它。一旦我们这样做了，我们就可以调用它来定义如何将它暴露给我们的组件。

`inject` 函数有两个参数：

1. 要注入的 property 的名称
2. 一个默认的值（可选）

使用 `MyMarker` 组件，可以使用以下代码对其进行重构：

```
1. <!-- src/components/MyMarker.vue -->
2. <script>
3. import { inject } from 'vue'
4.
5. export default {
6.   setup() {
7.     const userLocation = inject('location', 'The Universe')
8.     const userGeolocation = inject('geolocation')
9.
10.    return {
11.      userLocation,
12.      userGeolocation
13.    }
14.  }
15. }
16. </script>
```

响应性

添加响应性

为了增加提供值和注入值之间的响应性，我们可以在提供值时使用 `ref` 或 `reactive`。

使用 `MyMap` 组件，我们的代码可以更新如下：

```
1. <!-- src/components/MyMap.vue -->
2. <template>
3.   <MyMarker />
4. </template>
5.
6. <script>
7. import { provide, reactive, ref } from 'vue'
8. import MyMarker from './MyMarker.vue'
9.
10. export default {
```

```
11.   components: {
12.     MyMarker
13.   },
14.   setup() {
15.     const location = ref('North Pole')
16.     const geolocation = reactive({
17.       longitude: 90,
18.       latitude: 135
19.     })
20.
21.     provide('location', location)
22.     provide('geolocation', geolocation)
23.   }
24. }
25. </script>
```

现在，如果这两个 property 中有任何更改，`MyMarker` 组件也将自动更新！

修改响应式 property

当使用响应式提供/注入值时，建议尽可能，在提供者内保持响应式 **property** 的任何更改。

例如，在需要更改用户位置的情况下，我们最好在 `MyMap` 组件中执行此操作。

```
1.  <!-- src/components/MyMap.vue -->
2.  <template>
3.    <MyMarker />
4.  </template>
5.
6.  <script>
7.    import { provide, reactive, ref } from 'vue'
8.    import MyMarker from './MyMarker.vue'
9.
10.   export default {
11.     components: {
12.       MyMarker
13.     },
14.     setup() {
15.       const location = ref('North Pole')
16.       const geolocation = reactive({
17.         longitude: 90,
18.         latitude: 135
```

```
19.   })
20.
21.   provide('location', location)
22.   provide('geolocation', geolocation)
23.
24.   return {
25.     location
26.   }
27. },
28. methods: {
29.   updateLocation() {
30.     this.location = 'South Pole'
31.   }
32. }
33. }
34. </script>
```

然而，有时我们需要在注入数据的组件内部更新注入的数据。在这种情况下，我们建议提供一个方法来负责改变响应式 property。

```
1. <!-- src/components/MyMap.vue -->
2. <template>
3.   <MyMarker />
4. </template>
5.
6. <script>
7. import { provide, reactive, ref } from 'vue'
8. import MyMarker from './MyMarker.vue'
9.
10. export default {
11.   components: {
12.     MyMarker
13.   },
14.   setup() {
15.     const location = ref('North Pole')
16.     const geolocation = reactive({
17.       longitude: 90,
18.       latitude: 135
19.     })
20.
21.     const updateLocation = () => {
22.       location.value = 'South Pole'
```

```
23.     }
24.
25.     provide('location', location)
26.     provide('geolocation', geolocation)
27.     provide('updateLocation', updateLocation)
28.   }
29. }
30. </script>
```

```
1. <!-- src/components/MyMarker.vue -->
2. <script>
3. import { inject } from 'vue'
4.
5. export default {
6.   setup() {
7.     const userLocation = inject('location', 'The Universe')
8.     const userGeolocation = inject('geolocation')
9.     const updateUserLocation = inject('updateLocation')
10.
11.     return {
12.       userLocation,
13.       userGeolocation,
14.       updateUserLocation
15.     }
16.   }
17. }
18. </script>
```

最后，如果要确保通过 `provide` 传递的数据不会被注入的组件更改，我们建议对提供者的 property 使用 `readonly`。

```
1. <!-- src/components/MyMap.vue -->
2. <template>
3.   <MyMarker />
4. </template>
5.
6. <script>
7. import { provide, reactive, readonly, ref } from 'vue'
8. import MyMarker from './MyMarker.vue'
9.
10. export default {
11.   components: {
```

```
12.   MyMarker
13. },
14. setup() {
15.   const location = ref('North Pole')
16.   const geolocation = reactive({
17.     longitude: 90,
18.     latitude: 135
19.   })
20.
21.   const updateLocation = () => {
22.     location.value = 'South Pole'
23.   }
24.
25.   provide('location', readonly(location))
26.   provide('geolocation', readonly(geolocation))
27.   provide('updateLocation', updateLocation)
28. }
29. }
30. </script>
```

模板引用

本节代码示例使用[单文件组件](#)的语法

本指南假定你已经阅读了 [Composition API 简介](#)和[响应式基础](#)。如果你不熟悉组合 API，请先阅读此文章。

在使用组合 API 时，[响应式引用](#)和[模板引用](#)的概念是统一的。为了获得对模板内元素或组件实例的引用，我们可以像往常一样声明 `ref` 并从 `setup()` 返回：

```
1. <template>
2.   <div ref="root">This is a root element</div>
3. </template>
4.
5. <script>
6.   import { ref, onMounted } from 'vue'
7.
8.   export default {
9.     setup() {
10.      const root = ref(null)
11.
12.      onMounted(() => {
13.        // DOM元素将在初始渲染后分配给ref
14.        console.log(root.value) // <div>这是根元素</div>
15.      })
16.
17.      return {
18.        root
19.      }
20.    }
21.  }
22. </script>
```

这里我们在渲染上下文中暴露 `root`，并通过 `ref="root"`，将其绑定到 `div` 作为其 `ref`。在虚拟 DOM 补丁算法中，如果 `VNode` 的 `ref` 键对应于渲染上下文中的 `ref`，则 `VNode` 的相应元素或组件实例将被分配给该 `ref` 的值。这是在虚拟 DOM 挂载/打补丁过程中执行的，因此模板引用只会在初始渲染之后获得赋值。

作为模板使用的 `ref` 的行为与任何其他 `ref` 一样：它们是响应式的，可以传递到（或从中返回）复合函数中。

JSX 中的用法

```

1. export default {
2.   setup() {
3.     const root = ref(null)
4.
5.     return () =>
6.       h('div', {
7.         ref: root
8.       })
9.
10.    // with JSX
11.    return () => <div ref={root} />
12.  }
13. }

```

v-for 中的用法

Composition API 模板引用在 `v-for` 内部使用时没有特殊处理。相反，请使用函数引用执行自定义处理：

```

1. <template>
2.   <div v-for="(item, i) in list" :ref="el => { if (el) divs[i] = el }">
3.     {{ item }}
4.   </div>
5. </template>
6.
7. <script>
8.   import { ref, reactive, onBeforeUpdate } from 'vue'
9.
10.  export default {
11.    setup() {
12.      const list = reactive([1, 2, 3])
13.      const divs = ref([])
14.
15.      // 确保在每次更新之前重置ref
16.      onBeforeUpdate(() => {
17.        divs.value = []
18.      })
19.
20.      return {

```

```
21.         list,  
22.         divs  
23.     }  
24. }  
25. }  
26. </script>
```


渲染机制和优化

为了学习如何更好地使用 Vue，不需要阅读本页，但是它提供了更多信息，如果你想知道渲染在背后是如何工作的。

虚拟 DOM

现在我们知道了侦听器是如何更新组件的，你可能会问这些更改最终是如何应用到 DOM 中的！也许你以前听说过虚拟 DOM，包括 Vue 在内的许多框架都使用这种方式来确保我们的接口能够有效地反映我们在 JavaScript 中更新的更改

我们用 JavaScript 复制了一个名为 Virtual Dom 的 DOM，我们这样做是因为用 JavaScript 接触 DOM 的计算成本很高。虽然用 JavaScript 执行更新很廉价，但是找到所需的 DOM 节点并用 JS 更新它们的成本很高。所以我们批处理调用，同时更改 DOM。

虚拟 DOM 是轻量级的 JavaScript 对象，由渲染函数创建。它包含三个参数：元素，带有数据的对象，prop，attr 以及更多，和一个数组。数组是我们传递子级的地方，子级也具有所有这些参数，然后它们可以具有子级，依此类推，直到我们构建完整的元素树为止。

如果需要更新列表项，可以使用前面提到的响应式在 JavaScript 中进行。然后，我们对 JavaScript 副本，虚拟 DOM 进行所有更改，并在此与实际 DOM 之间进行区分。只有这样，我们才能对已更改的内容进行更新。虚拟 DOM 允许我们对 UI 进行高效的更新！

Vue 2 中的更改检测警告

该页面仅适用于 Vue 2.x 及更低版本，并假定你已经阅读了[响应式部分](#)。请先阅读该部分。

由于 JavaScript 的限制，有些 Vue 无法检测的更改类型。但是，有一些方法可以规避它们以维持响应性。

对于对象

Vue 无法检测到 property 的添加或删除。由于 Vue 在实例初始化期间执行 getter/setter 转换过程，因此必须在 `data` 对象中存在一个 property，以便 Vue 对其进行转换并使其具有响应式。例如：

```
1. var vm = new Vue({
2.   data: {
3.     a: 1
4.   }
5. })
6. // `vm.a` 现在是响应式的
7.
8. vm.b = 2
9. // `vm.b` 不是响应式的
```

对于已经创建的实例，Vue 不允许动态添加根级别的响应式 property。但是，可以使用

`Vue.set(object, propertyName, value)` 方法向嵌套对象添加响应式 property：

```
1. Vue.set(vm.someObject, 'b', 2)
```

你还可以使用 `vm.$set` 实例方法，这也是全局 `Vue.set` 方法的别名：

```
1. this.$set(this.someObject, 'b', 2)
```

有时你可能需要为已有对象赋值多个新 property，比如使用 `Object.assign()` 或 `_.extend()`。但是，这样添加到对象上的新 property 不会触发更新。在这种情况下，你应该用原对象与要混合进去的对象的 property 一起创建一个新的对象。

```
1. // 而不是 `Object.assign(this.someObject, { a: 1, b: 2 })`
2. this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })
```

对于数组

Vue 不能检测以下数组的变动：

1. 当你利用索引直接设置一个数组项时，例如：`vm.items[indexOfItem] = newValue`
2. 当你修改数组的长度时，例如：`vm.items.length = newLength`

例如：

```
1. var vm = new Vue({
2.   data: {
3.     items: ['a', 'b', 'c']
4.   }
5. })
6. vm.items[1] = 'x' // 不是响应式的
7. vm.items.length = 2 // 不是响应式的
```

为了解决第一种问题，以下两种方式都可以实现和 `vm.items[indexOfItem] = newValue` 相同的效果，同时也将在响应性系统内触发状态更新：

```
1. // Vue.set
2. Vue.set(vm.items, indexOfItem, newValue)
```

```
1. // Array.prototype.splice
2. vm.items.splice(indexOfItem, 1, newValue)
```

你也可以使用 `vm.$set` [\(opens new window\)](#) 实例方法，该方法是全局方法 `Vue.set` 的一个别名：

```
1. vm.$set(vm.items, indexOfItem, newValue)
```

为了解决第二种问题，你可以使用 `splice`：

```
1. vm.items.splice(newLength)
```

声明响应式 property

由于 Vue 不允许动态添加根级响应式 property，所以你必须要在初始化实例前声明所有根级响应式 property，哪怕只是一个空值：

```
1. var vm = new Vue({
2.   data: {
3.     // 声明 message 为一个空值字符串
4.     message: ''
5.   },
6.   template: '<div>{{ message }}</div>'
7. })
8. // 之后设置 `message`
9. vm.message = 'Hello!'
```

如果你未在 `data` 选项中声明 `message`，Vue 将警告你渲染函数正在试图访问不存在的 `property`。

这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使组件实例能更好地配合类型检查系统工作。但与此同时在代码可维护性方面也有点重要的考虑：`data` 对象就像组件的状态结构（schema）。提前声明所有的响应式 `property`，可以让组件代码在未来修改或给其他开发人员阅读时更易于理解。

异步更新队列

可能你还没有注意到，Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 `watcher` 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际（已去重的）工作。Vue 在内部对异步队列尝试使用原生的 `Promise.then`、`MutationObserver` 和 `setImmediate`，如果执行环境不支持，则会采用 `setTimeout(fn, 0)` 代替。

例如，当你设置 `vm.someData = 'new value'`，该组件不会立即重新渲染。当刷新队列时，组件会在下一个事件循环“tick”中更新。多数情况我们不需要关心这个过程，但是如果你想基于更新后的 DOM 状态来做点什么，这就可能会有些棘手。虽然 `Vue.js` 通常鼓励开发人员使用“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们必须这么做。为了在数据变化之后等待 Vue 完成更新 DOM，可以在数据变化之后立即使用 `Vue.nextTick(callback)`。这样回调函数将在 DOM 更新完成后被调用。例如：

```
1. <div id="example">{{ message }}</div>
```

```
1. var vm = new Vue({
2.   el: '#example',
3.   data: {
4.     message: '123'
```

```

5.   }
6. })
7. vm.message = 'new message'           // 更改数据
8. vm.$el.textContent === 'new message' // false
9. Vue.nextTick(function() {
10.   vm.$el.textContent === 'new message' // true
11. })

```

在组件内使用 `vm.$nextTick()` 实例方法特别方便，因为它不需要全局 `Vue`，并且回调函数中的 `this` 将自动绑定到当前的组件实例上：

```

1. Vue.component('example', {
2.   template: '<span>{{ message }}</span>',
3.   data: function() {
4.     return {
5.       message: 'not updated'
6.     }
7.   },
8.   methods: {
9.     updateMessage: function() {
10.      this.message = 'updated'
11.      console.log(this.$el.textContent) // => 'not updated'
12.      this.$nextTick(function() {
13.        console.log(this.$el.textContent) // => 'updated'
14.      })
15.    }
16.  }
17. })

```

因为 `$nextTick()` 返回一个 Promise 对象，所以你可以使用新的 ES2017 `async/await` [\(opens new window\)](#) 语法完成相同的事情：

```

1.   methods: {
2.     updateMessage: async function () {
3.       this.message = 'updated'
4.       console.log(this.$el.textContent) // => 'not updated'
5.       await this.$nextTick()
6.       console.log(this.$el.textContent) // => 'updated'
7.     }
8.   }

```

工具

- [单文件组件](#)
- [测试](#)
- [TypeScript 支持](#)
- [Mobile](#)

单文件组件

介绍

在很多 Vue 项目中，我们使用 `app.component` 来定义全局组件，紧接着用 `app.mount('#app')` 在每个页面内指定一个容器元素。

这种方式在很多中小规模的项目中运作的很好，在这些项目里 JavaScript 只被用来加强特定的视图。但当在更复杂的项目中，或者你的前端完全由 JavaScript 驱动的时候，下面这些缺点将变得非常明显：

- **全局定义 (Global definitions)** 强制要求每个 component 中的命名不得重复
- **字符串模板 (String templates)** 缺乏语法高亮，在 HTML 有多行的时候，需要用到丑陋的 `\`
- **不支持 CSS (No CSS support)** 意味着当 HTML 和 JavaScript 组件化时，CSS 明显被遗漏
- **没有构建步骤 (No build step)** 限制只能使用 HTML 和 ES5 JavaScript，而不能使用预处理器，如 Pug (formerly Jade) 和 Babel

所有这些都可以通过扩展名为 `.vue` 的 **single-file components** (单文件组件) 来解决，并且还可以使用 webpack 或 Browserify 等构建工具。

这是一个文件名为 `Hello.vue` 的简单实例：

```
<template>
  <p>{{ greeting }} World!</p>
</template>

<script>
module.exports = {
  data: function() {
    return {
      greeting: "Hello"
    }
  }
}
</script>

<style scoped>
p {
  font-size: 2em;
  text-align: center;
}
</style>
```

现在我们获得：

- [完整语法高亮](#)  (opens new window)
- [CommonJS 模块](#)  (opens new window)
- [组件作用域的 CSS](#)  (opens new window)

正如我们说过的，我们可以使用预处理器来构建简洁和功能更丰富的组件，比如 Pug, Babel (with ES2015 modules), 和 Stylus。

```
<template lang="pug">
div
  p {{ greeting }} World!
  other-component
</template>

<script>
import OtherComponent from './OtherComponent.vue'

export default {
  components: {
    OtherComponent
  },

  data () {
    return {
      greeting: 'Hello'
    }
  }
}
</script>

<style lang="stylus" scoped>
p
  font-size 2em
  text-align center
</style>
```

这些特定的语言只是例子，你可以只是简单地使用 Babel, TypeScript, SCSS, PostCSS 或者其他任何能够帮助你提高生产力的预处理器。如果搭配 `vue-loader` 使用 webpack，它也能 CSS Modules 提供头等支持。

怎么看待关注点分离？

一个重要的事情值得注意，关注点分离不等于文件类型分离。在现代 UI 开发中，我们已经发现相比于把代码库分离成三个大的层次并将其相互交织起来，把它们划分为松散耦合的组件再将其组合起来更合理一些。在一个组件里，其模板、逻辑和样式是内部耦合的，并且把他们搭配在一起实际上使得组件更加内聚且更可维护。

即便你不喜欢单文件组件，你仍然可以把 JavaScript、CSS 分离成独立的文件然后做到热重载和预编译。

```
1. <!-- my-component.vue -->
2. <template>
3.   <div>This will be pre-compiled</div>
4. </template>
5. <script src="./my-component.js"></script>
6. <style src="./my-component.css"></style>
```

起步

例子沙箱

如果你希望深入了解并开始使用单文件组件，请来 [CodeSandbox 看看这个简单的 todo 应用](#) (opens new window)。

针对刚接触 JavaScript 模块开发系统的用户

有了 `.vue` 组件，我们就进入了高阶 JavaScript 应用领域。如果你没有准备好的话，意味着还需要学会使用一些附加的工具：

- **Node 包管理器 (npm)**: 阅读 [Getting Started guide](#) (opens new window) 中关于如何从注册地 (registry) 获取包的章节。
- **现代 JavaScript 与 ES2015/16**: 阅读 Babel 的 [Learn ES2015 guide](#) (opens new window)。你不需要立刻记住每一个方法，但是你可以保留这个页面以便后期参考。

在你花一天时间了解这些资源之后，我们建议你参考 [Vue CLI](#) (opens new window)。只要遵循指示，你就能很快地运行一个带有 `.vue` 组件、ES2015、webpack 和热重载 (hot-reloading) 的 Vue 项目！

针对高阶用户

CLI 会为你搞定大多数工具的配置问题，同时也支持细粒度自定义配置项[↗] ([opens new window](#))。

有时你会想从零搭建你自己的构建工具，这时你需要通过 [Vue Loader](#)[↗] ([opens new window](#)) 手动配置 webpack。关于学习更多 webpack 的内容，请查阅[其官方文档](#)[↗] ([opens new window](#))和 [Webpack Academy](#)[↗] ([opens new window](#))。

测试

介绍

当构建可靠的应用时，测试在个人或团队构建新特性、重构代码、修复 bug 等工作中扮演了关键的角色。尽管测试的流派有很多，它们在 web 应用这个领域里主要有三大类：

- 单元测试
- 组件测试
- 端到端 (E2E, end-to-end) 测试

本章节致力于引导大家了解测试的生态系统的并为 Vue 应用或组件库选择适合的工具。

单元测试

介绍

单元测试允许你将独立单元的代码进行隔离测试，其目的是为开发者提供对代码的信心。通过编写细致且有意义的测试，你能够有信心在构建新特性或重构已有代码的同时，保持应用的功能和稳定。

为一个 Vue 应用做单元测试并没有和为其它类型的应用做测试有什么明显的区别。

选择框架

因为单元测试的建议通常是框架无关的，所以下面只是当你在评估应用的单元测试工具时需要的一些基本指引。

一流的错误报告

当测试失败时，提供有用的错误信息对于单元测试框架来说至关重要。这是断言库应尽的职责。一个具有高质量错误信息的断言能够最小化调试问题所需的时间。除了简单地告诉你什么测试失败了，断言库还应额外提供上下文以及测试失败的原因，例如预期结果 vs 实际得到的结果。

一些诸如 Jest 这样的单元测试框架会包含断言库。另一些诸如 Mocha 需要你单独安装断言库（通常会用 Chai）。

活跃的社区和团队

因为主流的单元测试框架都是开源的，所以对于一些旨在长期维护其测试且确保项目本身保持活跃的团队来说，拥有一个活跃的社区是至关重要的。额外的好处是，在任何时候遇到问题时，一个活跃的社区

会为你提供更多的支持。

框架

尽管生态系统里有很多工具，这里我们列出一些在 Vue 生态系统中常用的单元测试工具。

Jest

Jest 是一个专注于简易性的 JavaScript 测试框架。一个其独特的功能是可以为测试生成快照 (snapshot)，以提供另一种验证应用单元的方法。

资料：

- [Jest 官网](#) [↗] (opens new window)
- [Vue CLI 官方插件 - Jest](#) [↗] (opens new window)

Mocha

Mocha 是一个专注于灵活性的 JavaScript 测试框架。因为其灵活性，它允许你选择不同的库来满足诸如侦听（如 Sinon）和断言（如 Chai）等其它常见的功能。另一个 Mocha 独特的功能是它不止可以在 Node.js 里运行测试，还可以在浏览器里运行测试。

资料：

- [Mocha 官网](#) [↗] (opens new window)
- [Vue CLI 官方插件 - Mocha](#) [↗] (opens new window)

组件测试

介绍

测试大多数 Vue 组件时都必须将它们挂载到 DOM（虚拟或真实）上，才能完全断言它们正在工作。这是另一个与框架无关的概念。因此组件测试框架的诞生，是为了让用户能够以可靠的方式完成这项工作，同时还提供了 Vue 特有的诸如对 Vuex、Vue Router 和其他 Vue 插件的集成的便利性。

选择框架

以下章节提供了在评估最适合你的应用的组件测试框架时需要记住的事项。

与 Vue 生态系统的最佳兼容性

毋庸置疑，最重要的标准之一就是组件测试库应该尽可能与 Vue 生态系统兼容。虽然这看起来很全面，但需要记住的一些关键集成领域包括单文件组件（SFC）、Vuex、Vue Router 以及应用所依赖的任何其他特定于 Vue 的插件。

一流的错误报告

当测试失败时，提供有用的错误日志以最小化调试问题所需的时间对于组件测试框架来说至关重要。除了简单地告诉你什么测试失败了，他们还应额外提供上下文以及测试失败的原因，例如预期结果 vs 实际得到的结果。

推荐

Vue Testing Library (@testing-library/vue)

Vue Testing Library 是一组专注于测试组件而不依赖实现细节的工具。由于在设计时就充分考虑了可访问性，它采用的方案也使重构变得轻而易举。

它的指导原则是，与软件使用方式相似的测试越多，它们提供的可信度就越高。

资料：

- [Vue Testing Library 官网](#) [↗] (opens new window)

Vue Test Utils

Vue Test Utils 是官方的偏底层的组件测试库，它是为用户提供对 Vue 特定 API 的访问而编写的。如果你对测试 Vue 应用不熟悉，我们建议你使用 Vue Testing Library，它是 Vue Test Utils 的抽象。

资源：

- [Vue Test Utils 官方文档](#) [↗] (opens new window)
- [Vue 测试指南](#) [↗] (opens new window) by Lachlan Miller

端到端（E2E）测试

介绍

虽然单元测试为开发者提供了一定程度的信心，但是单元测试和组件测试在部署到生产环境时提供应用整体覆盖的能力是有限的。因此，端到端测试可以说从应用最重要的方面进行测试覆盖：当用户实际使用应用时会发生什么。

换句话说，端到端测试验证应用中的所有层。这不仅包括你的前端代码，还包括所有相关的后端服务和基础设施，它们更能代表你的用户所处的环境。通过测试用户操作如何影响应用，端到端测试通常是提高应用是否正常运行的信心的关键。

选择框架

虽然 web 上的端到端测试因不可信赖（片面的）测试和减慢开发过程而得到负面的声誉，但现代端到端工具在创建更可靠的、交互的和实用的测试方面取得了长足进步。在选择端到端测试框架时，以下章节在你为应用选择测试框架时提供了一些指导。

跨浏览器测试

端到端测试的一个主要优点是它能够跨多个浏览器测试应用。尽管 100% 的跨浏览器覆盖看上去很诱人，但需要注意的是，因为持续运行这些跨浏览器测试需要额外的时间和机器消耗，它会降低团队的资源回报。因此，在选择应用需要的跨浏览器测试数量时，必须注意这种权衡。

TIP

针对浏览器特定问题的一个最新进展是，针对不常用的浏览器（如：< IE11、旧版 Safari 等）使用应用监视和错误报告工具（如：Sentry、LogRocket 等）。

更快的反馈路径

端到端测试和开发的主要问题之一是运行整个套件需要很长时间。通常，这只在持续集成和部署（CI/CD）管道中完成。现代的端到端测试框架通过添加类似并行化的特性来帮助解决这个问题，这使得 CI/CD 管道的运行速度通常比以前快。此外，在本地开发时，有选择地为正在处理的页面运行单个测试的能力，同时还提供测试的热重载，将有助于提高开发者的工作流程和工作效率。

一流的调试经验

虽然开发者传统上依赖于在终端窗口中扫描日志来帮助确定测试中出了什么问题，但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。

推荐

虽然生态系统中有许多工具，但以下是一些 Vue.js 生态系统中常用的端到端测试框架。

Cypress.io

Cypress.io 是一个测试框架，旨在通过使开发者能够可靠地测试他们的应用，同时提供一流的开发者体验，来提高开发者的生产率。

资料：

- [Cypress 官网](#) [↗] (opens new window)
- [Vue CLI 官方插件 - Cypress](#) [↗] (opens new window)
- [Cypress Testing Library](#) [↗] (opens new window)

Nightwatch.js

Nightwatch.js 是一个端到端测试框架，可用于测试 web 应用和网站，以及 Node.js 单元测试和集成测试。

资料：

- [Nightwatch 官网](#) [↗] (opens new window)
- [Vue CLI 官方插件 - Nightwatch](#) [↗] (opens new window)

Puppeteer

Puppeteer 是一个 Node.js 库，它提供高阶 API 来控制浏览器，并可以与其他测试运行程序（例如 Jest）配对来测试应用。

资料：

- [Puppeteer 官网](#) [↗] (opens new window)

TestCafe

TestCafe 是一个基于端到端的 Node.js 框架，旨在提供简单的设置，以便开发者能够专注于创建易于编写和可靠的测试。

资料：

- [TestCafe 官网](#) [↗] (opens new window)

TypeScript 支持

[Vue CLI](#) [↗] (opens new window) 提供内置的 TypeScript 工具支持。

NPM 包中的官方声明

随着应用的增长，静态类型系统可以帮助防止许多潜在的运行时错误，这就是为什么 Vue 3 是用 TypeScript 编写的。这意味着在 Vue 中使用 TypeScript 不需要任何其他工具——它具有一流的公民支持。

推荐配置

```
1. // tsconfig.json
2. {
3.   "compilerOptions": {
4.     "target": "esnext",
5.     "module": "esnext",
6.     // 这样就可以对 `this` 上的数据属性进行更严格的推断`
7.     "strict": true,
8.     "jsx": "preserve",
9.     "moduleResolution": "node"
10.  }
11. }
```

请注意，必须包含 `strict: true`（或至少包含 `noImplicitThis: true`，它是 `strict` 标志的一部分）才能在组件方法中利用 `this` 的类型检查，否则它总是被视为 `any` 类型。

参见 [TypeScript 编译选项文档](#) [↗] (opens new window) 查看更多细节。

开发工具

项目创建

[Vue CLI](#) [↗] (opens new window) 可以生成使用 TypeScript 的新项目，开始：

```
1. # 1. Install Vue CLI, 如果尚未安装
```

```
2. npm install --global @vue/cli@next
3.
4. # 2. 创建一个新项目, 选择 "Manually select features" 选项
5. vue create my-project-name
6.
7. # 3. 如果已经有一个不存在TypeScript的 Vue CLI项目, 请添加适当的 Vue CLI插件:
8. vue add typescript
```

请确保组件的 `script` 部分已将语言设置为 TypeScript:

```
1. <script lang="ts">
2.   ...
3. </script>
```

编辑器支持

对于使用 TypeScript 开发 Vue 应用程序, 我们强烈建议使用 [Visual Studio Code](#) (opens new window), 它为 TypeScript 提供了很好的开箱即用支持。如果你使用的是单文件组件 (SFCs), 那么可以使用很棒的 [Vetur extension](#) (opens new window), 它在 SFCs 中提供了 TypeScript 推理和许多其他优秀的特性。

[WebStorm](#) (opens new window) 还为 TypeScript 和 Vue 提供现成的支持。

定义 Vue 组件

要让 TypeScript 正确推断 Vue 组件选项中的类型, 需要使用 `defineComponent` 全局方法定义组件:

```
1. import { defineComponent } from 'vue'
2.
3. const Component = defineComponent({
4.   // 已启用类型推断
5. })
```

与 Options API 一起使用

TypeScript 应该能够在不显式定义类型的情况下推断大多数类型。例如, 如果有一个具有数字 `count` property 的组件, 如果试图对其调用特定于字符串的方法, 则会出现错误:

```

1.  const Component = defineComponent({
2.    data() {
3.      return {
4.        count: 0
5.      }
6.    },
7.    mounted() {
8.      const result = this.count.split('') // => Property 'split' does not exist
9.      on type 'number'
10.    }
11.  })

```

如果你有一个复杂的类型或接口，你可以使用 [type assertion](#) (opens new window) 对其进行强制转换：

```

1.  interface Book {
2.    title: string
3.    author: string
4.    year: number
5.  }
6.
7.  const Component = defineComponent({
8.    data() {
9.      return {
10.        book: {
11.          title: 'Vue 3 Guide',
12.          author: 'Vue Team',
13.          year: 2020
14.        } as Book
15.      }
16.    }
17.  })

```

注释返回类型

由于 Vue 声明文件的循环特性，TypeScript 可能难以推断 computed 的类型。因此，你可能需要注释返回类型的计算属性。

```

1.  import { defineComponent } from 'vue'
2.
3.  const Component = defineComponent({

```

```

4.   data() {
5.     return {
6.       message: 'Hello!'
7.     }
8.   },
9.   computed: {
10.    // 需要注释
11.    greeting(): string {
12.      return this.message + '!'
13.    }
14.
15.    // 在使用setter进行计算时, 需要对getter进行注释
16.    greetingUppercased: {
17.      get(): string {
18.        return this.greeting.toUpperCase();
19.      },
20.      set(newValue: string) {
21.        this.message = newValue.toUpperCase();
22.      },
23.    },
24.  }
25. })

```

注释 Props

Vue 对定义了 `type` 的 prop 执行运行时验证。要将这些类型提供给 TypeScript, 我们需要使用 `PropType` 强制转换构造函数:

```

1. import { defineComponent, PropType } from 'vue'
2.
3. interface ComplexMessage {
4.   title: string
5.   okMessage: string
6.   cancelMessage: string
7. }
8. const Component = defineComponent({
9.   props: {
10.    name: String,
11.    success: { type: String },
12.    callback: {
13.      type: Function as PropType<() => void>
14.    },

```

```

15.     message: {
16.       type: Object as PropType<ComplexMessage>,
17.       required: true,
18.       validator(message: ComplexMessage) {
19.         return !!message.title
20.       }
21.     }
22.   }
23. })

```

如果你发现验证器没有得到类型推断或者成员完成不起作用，那么用期望的类型注释参数可能有助于解决这些问题。

与 Composition API 一起使用

在 `setup()` 函数中，不需要将类型传递给 `props` 参数，因为它将从 `props` 组件选项推断类型。

```

1. import { defineComponent } from 'vue'
2.
3. const Component = defineComponent({
4.   props: {
5.     message: {
6.       type: String,
7.       required: true
8.     }
9.   },
10.
11.   setup(props) {
12.     const result = props.message.split('') // 正确, 'message' 被声明为字符串
13.     const filtered = props.message.filter(p => p.value) // 将引发错误: Property
14.     'filter' does not exist on type 'string'
15.   }
16. })

```

类型声明 `ref`

Refs 根据初始值推断类型:

```

1. import { defineComponent, ref } from 'vue'
2.

```

```

3. const Component = defineComponent({
4.   setup() {
5.     const year = ref(2020)
6.
7.     const result = year.value.split('') // => Property 'split' does not exist
8.   }
9. })

```

有时我们可能需要为 `ref` 的内部值指定复杂类型。我们可以在调用 `ref` 重写默认推理时简单地传递一个泛型参数：

```

1. const year = ref<string | number>('2020') // year's type: Ref<string | number>
2.
3. year.value = 2020 // ok!

```

TIP

如果泛型的类型未知，建议将 `ref` 转换为 `Ref<T>`。

类型声明 `reactive`

当声明类型 `reactive` property，我们可以使用接口：

```

1. import { defineComponent, reactive } from 'vue'
2.
3. interface Book {
4.   title: string
5.   year?: number
6. }
7.
8. export default defineComponent({
9.   name: 'HelloWorld',
10.  setup() {
11.    const book = reactive<Book>({ title: 'Vue 3 Guide' })
12.    // or
13.    const book: Book = reactive({ title: 'Vue 3 Guide' })
14.    // or
15.    const book = reactive({ title: 'Vue 3 Guide' }) as Book
16.  }
17. })

```

类型声明

`computed`

计算值将根据返回值自动推断类型

```
1. import { defineComponent, ref, computed } from 'vue'
2.
3. export default defineComponent({
4.   name: 'CounterButton',
5.   setup() {
6.     let count = ref(0)
7.
8.     // 只读
9.     const doubleCount = computed(() => count.value * 2)
10.
11.     const result = doubleCount.value.split('') // => Property 'split' does not
12.     exist on type 'number'
13.   }
14. })
```


Mobile

介绍

虽然 Vue.js 本身并不支持移动应用开发，但是有很多解决方案可以用 Vue.js 创建原生 iOS 和 Android 应用。

混合应用开发

Capacitor

Capacitor [↗](#) (opens new window) 是一个来自 Ionic Team [↗](#) (opens new window) 的项目，通过提供跨多个平台运行的 API，开发者可以使用单个代码库构建原生 iOS、Android 和 PWA 应用。

资源

- [Capacitor + Vue.js Guide](#) [↗](#) (opens new window)

NativeScript

NativeScript [↗](#) (opens new window) 使用已熟悉的 Web 技能为跨平台（真正的原生）移动应用提供支持。两者结合在一起是开发沉浸式移动体验的绝佳选择。

资源

- [NativeScript + Vue.js Guide](#) [↗](#) (opens new window)

规模化

- [路由](#)
- [状态管理](#)
- [服务端渲染](#)

路由

官方 Router

对于大多数单页面应用，都推荐使用官方支持的 [vue-router 库](#) (opens new window)。更多细节可以移步 [vue-router 文档](#) (opens new window)。

从零开始简单的路由

如果你只需要非常简单的路由而不想引入一个功能完整的路由库，可以像这样动态渲染一个页面级的组件：

```
1. const NotFoundComponent = { template: '<p>Page not found</p>' }
2. const HomeComponent = { template: '<p>Home page</p>' }
3. const AboutComponent = { template: '<p>About page</p>' }
4.
5. const routes = {
6.   '/': HomeComponent,
7.   '/about': AboutComponent
8. }
9.
10. const SimpleRouter = {
11.   data: () => ({
12.     currentRoute: window.location.pathname
13.   }),
14.
15.   computed: {
16.     CurrentComponent() {
17.       return routes[this.currentRoute] || NotFoundComponent
18.     }
19.   },
20.
21.   render() {
22.     return Vue.h(this.CurrentComponent)
23.   }
24. }
25.
26. Vue.createApp(SimpleRouter).mount('#app')
```

结合 [HTML5 History API](#) (opens new window), 你可以建立一个麻雀虽小但是五脏俱全的客户端路由器。可以直接看[实例应用](#) (opens new window)。

整合第三方路由

如果你有更偏爱的第三方路由, 如 [Page.js](#) (opens new window) 或者 [Director](#) (opens new window), 整合起来也一样简单 (opens new window)。这里有一个使用了 [Page.js](#) 的完整示例 (opens new window)。

状态管理

类 Flux 状态管理的官方实现

由于状态零散地分布在许多组件和组件之间的交互中，大型应用复杂度也经常逐渐增长。为了解决这个问题，Vue 提供 [vuex](#) (opens new window)：我们有受到 Elm 启发的状态管理库。vuex 甚至集成到 [vue-devtools](#) (opens new window)，无需配置即可进行[时光旅行调试](#) (time travel debugging) (opens new window)。

React 的开发者请参考以下信息

如果你是来自 React 的开发者，你可能会对 Vuex 和 [Redux](#) (opens new window) 间的差异表示关注，Redux 是 React 生态环境中最流行的 Flux 实现。Redux 事实上无法感知视图层，所以它能够轻松的通过一些[简单绑定](#) (opens new window)和 Vue 一起使用。Vuex 区别在于它是一个专门为 Vue 应用所设计。这使得它能够更好地和 Vue 进行整合，同时提供简洁的 API 和改善过的开发体验。

简单状态管理起步使用

经常被忽略的是，Vue 应用中响应式 `data` 对象的实际来源——当访问数据对象时，一个组件实例只是简单的代理访问。所以，如果你有一处需要被多个实例间共享的状态，你可以使用一个 [reactive](#) 方法让对象作为响应式对象。

```
1. const sourceOfTruth = Vue.reactive({
2.   message: 'Hello'
3. })
4.
5. const appA = Vue.createApp({
6.   data() {
7.     return sourceOfTruth
8.   }
9. }).mount('#app-a')
10.
11. const appB = Vue.createApp({
12.   data() {
13.     return sourceOfTruth
```

```

14.   }
15. }).mount('#app-b')

```

```

1. <div id="app-a">App A: {{ message }}</div>
2.
3. <div id="app-b">App B: {{ message }}</div>

```

现在当 `sourceOfTruth` 发生变更，`appA` 和 `appB` 都将自动地更新它们的视图。我们现在只有一个真实来源，但调试将是一场噩梦。我们应用的任何部分都可以随时更改任何数据，而不会留下变更过的记录。

```

1. const appB = Vue.createApp({
2.   data() {
3.     return sourceOfTruth
4.   },
5.   mounted() {
6.     sourceOfTruth.message = 'Goodbye' // both apps will render 'Goodbye'
7.     message now
8.   }
9. }).mount('#app-b')

```

为了解决这个问题，我们采用一个简单的 `store` 模式：

```

1. const store = {
2.   debug: true,
3.
4.   state: Vue.reactive({
5.     message: 'Hello!'
6.   }),
7.
8.   setMessageAction(newValue) {
9.     if (this.debug) {
10.      console.log('setMessageAction triggered with', newValue)
11.    }
12.
13.    this.state.message = newValue
14.  },
15.
16.  clearMessageAction() {
17.    if (this.debug) {
18.      console.log('clearMessageAction triggered')

```

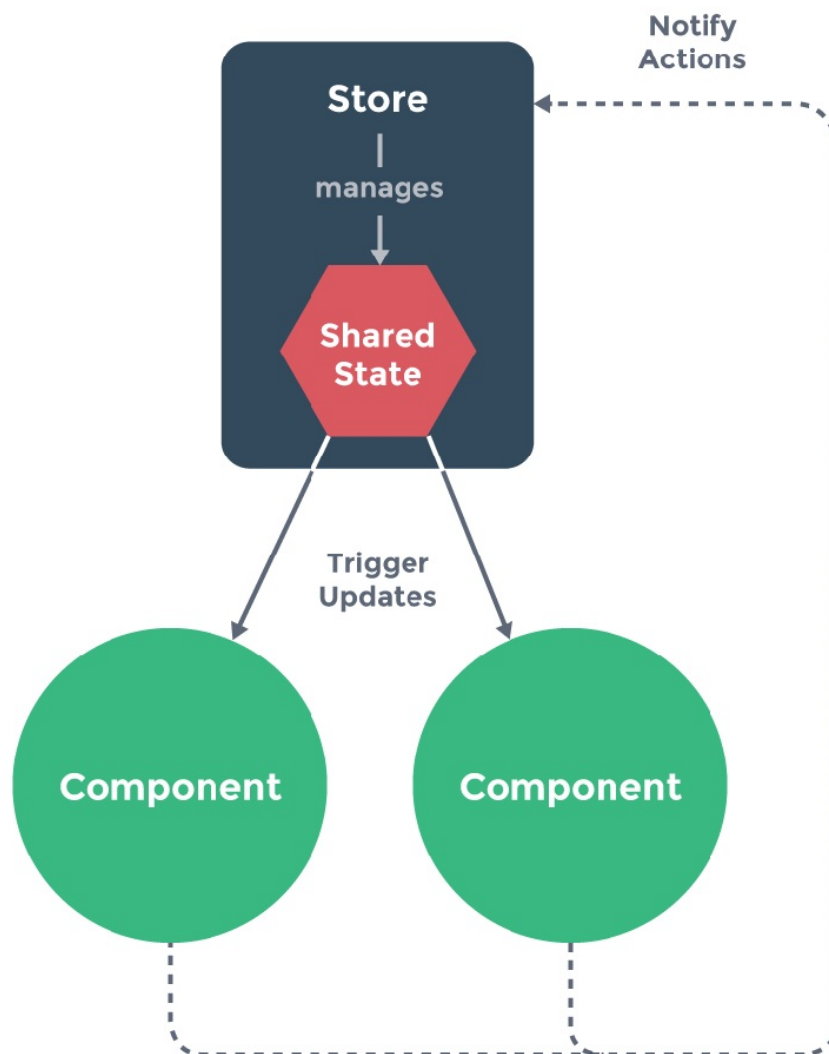
```
19.     }
20.
21.     this.state.message = ''
22.   }
23. }
```

需要注意，所有 store 中 state 的变更，都放置在 store 自身的 action 中去管理。这种集中式状态管理能够被更容易地理解哪种类型的变更将会发生，以及它们是如何被触发。当错误出现时，我们现在也会有一个 log 记录 bug 之前发生了什么。

此外，每个实例/组件仍然可以拥有和管理自己的私有状态：

```
1. <div id="app-a">{{sharedState.message}}</div>
2.
3. <div id="app-b">{{sharedState.message}}</div>
```

```
1. const appA = Vue.createApp({
2.   data() {
3.     return {
4.       privateState: {},
5.       sharedState: store.state
6.     }
7.   },
8.   mounted() {
9.     store.setMessageAction('Goodbye!')
10.  }
11. }).mount('#app-a')
12.
13. const appB = Vue.createApp({
14.   data() {
15.     return {
16.       privateState: {},
17.       sharedState: store.state
18.     }
19.   }
20. }).mount('#app-b')
```



TIP

重要的是，注意你不应该在 `action` 中替换原始的状态对象——组件和 `store` 需要引用同一个共享对象，变更才能够被观察到。

接着我们继续延伸约定，组件不允许直接变更属于 `store` 实例的 `state`，而应执行 `action` 来分发 (`dispatch`) 事件通知 `store` 去改变，我们最终达成了 [Flux](#) [🔗] (`opens new window`) 架构。这样约定的好处是，我们能够记录所有 `store` 中发生的 `state` 变更，同时实现能做到记录变更、保存状态快照、历史回滚/时光旅行的先进的调试工具。

说了一圈其实又回到了 [Vuex](#) [🔗] (`opens new window`)，如果你已经读到这儿，或许可以去尝试一

下！

服务端渲染

SSR 完全指南

我们创建了一份完整的构建 Vue 服务端渲染应用的指南。这份指南非常深入，适合已经熟悉 Vue、webpack 和 Node.js 开发的开发者阅读。请移步 ssr.vuejs.org [↗] (opens new window)。

Nuxt.js

从头搭建一个服务端渲染的应用是相当复杂的。幸运的是，我们有一个优秀的社区项目 [Nuxt.js](https://nuxt.js.org) [↗] (opens new window) 让这一切变得非常简单。Nuxt 是一个基于 Vue 生态的更高层的框架，为开发服务端渲染的 Vue 应用提供了极其便利的开发体验。更酷的是，你甚至可以用它来做为静态站生成器。推荐尝试。

Quasar Framework SSR + PWA

[Quasar Framework](https://quasar.dev) [↗] (opens new window) 可以通过其一流的构建系统、合理的配置和开发者扩展性生成（可选地和 PWA 互通的）SSR 应用，让你的想法的设计和构建变得轻而易举。你可以在服务端挑选执行超过上百款遵循“Material Design 2.0”的组件，并在浏览器端可用。你甚至可以管理网站的 `<meta>` 标签。Quasar 是一个基于 Node.js 和 webpack 的开发环境，它可以通过一套代码完成 SPA、PWA、SSR、Electron、Capacitor 和 Cordova 应用的快速开发。

无障碍

- [基础](#)
- [语义学](#)
- [标准](#)
- [资源](#)

基础

Web 可访问性（也称为 a11y）是指创建可供任何人使用的网站的实践方式——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，还是仅仅是处于不利环境中的人。例如，在视频中添加字幕可以帮助失聪、重听或在嘈杂的环境中听不到手机的用户。同样，请确保文字对比度不要太低，这对低视力用户和那些试图在强光下使用手机的用户都有帮助。

你是否已经准备开始却又无从下手？

可以先看看由[万维网联盟 \(W3C\)](#) [↗] ([opens new window](#)) 提供的[规划和管理 web 可访问性](#) [↗] ([opens new window](#))。

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：

```
1. <ul class="skip-links">
2.   <li>
3.     <a href="#main" ref="skipLink">跳到主内容</a>
4.   </li>
5. </ul>
```

若想在非聚焦状态下隐藏该链接，可以添加以下样式：

```
1. .skipLink {
2.   white-space: nowrap;
3.   margin: 1em auto;
4.   top: 0;
5.   position: fixed;
6.   left: 50%;
7.   margin-left: -72px;
8.   opacity: 0;
9. }
10. .skipLink:focus {
11.   opacity: 1;
12.   background-color: white;
13.   padding: .5em;
```

```
14.   border: 1px solid black;
15. }
```

一旦用户改变路由，请将焦点放回到这个跳过链接。通过用如下方式聚焦 `ref` 即可实现：

```
1. <script>
2.   export default {
3.     watch: {
4.       $route() {
5.         this.$refs.skipLink.focus();
6.       }
7.     }
8.   };
9. </script>
```

[阅读关于跳跃到主体内容的链接的文档](#) [↗] (opens new window)

组织内容

可访问性最重要的部分之一是确保设计本身是可访问的。设计不仅要考虑颜色对比度、字体选择、文本大小和语言，还要考虑应用程序中内容的结构。

标题

用户可以通过标题在应用程序中进行导航。为应用程序的每个部分设置描述性标题可以让用户更容易地预测每个部分的内容。说到标题，有几个推荐的可访问性实践：

- 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

[关于标题可进一步阅读](#) [↗] (opens new window)

```
1. <main role="main" aria-labelledby="main-title">
2.   <h1 id="main-title">Main title</h1>
3.   <section aria-labelledby="section-title">
4.     <h2 id="section-title"> Section Title </h2>
5.     <h3>Section Subtitle</h3>
6.     <!-- 内容 -->
7.   </section>
8.   <section aria-labelledby="section-title">
```

```

9.     <h2 id="section-title"> Section Title </h2>
10.    <h3>Section Subtitle</h3>
11.    <!-- 内容 -->
12.    <h3>Section Subtitle</h3>
13.    <!-- 内容 -->
14.    </section>
15. </main>

```

地标

地标 (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程序的每个部分。你可以使用 [ARIA role \(opens new window\)](#) 帮助你实现这个目标。

HTML	ARIA Role	地标的目的
header	role="banner"	主标题：页面的标题
nav	role="navigation"	适合用作文档或相关文档导航的链接集合
main	role="main"	文档的主体或中心内容
footer	role="contentinfo"	关于父级文档的信息：脚注/版权/隐私声明链接
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
无对应元素	role="search"	该章节包含整个应用的搜索功能
form	role="form"	表单相关元素的集合
section	role="region"	相关的且用户可能会导航到的内容。必须为该元素提供 label

Tip:

在使用地标 HTML 元素时，建议加上冗余的地标 role attribute，以最大限度地与传统[不支持 HTML5 语义元素的浏览器 \(opens new window\)](#)兼容。

[关于地标可进一步阅读 \(opens new window\)](#)

语义学

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>`。

标签通常放置在表单字段的顶部或左侧：

```
1. <form action="/dataCollectionLocation" method="post" autocomplete="on">
2.   <div v-for="item in formItems" :key="item.id" class="form-item">
3.     <label :for="item.id">{{ item.label }}: </label>
4.     <input
5.       :type="item.type"
6.       :id="item.id"
7.       :name="item.id"
8.       v-model="item.value"
9.     />
10.   </div>
11.   <button type="submit">Submit</button>
12. </form>
```

注意如何在表单元素中包含 `autocomplete='on'`，它将应用于表单中的所有输入。你也可以为每个输入设置不同的[自动完成属性的值](#) (opens new window)。

标签

提供标签以描述所有表单控件的用途；链接 `for` 和 `id`：

```
1. <label for="name">Name</label>
2. <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：

Styles Event Listeners DOM Breakpoints Properties Accessibility >>

▼ 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

警告:

虽然你可能已经看到这样包装输入字段的标签:

```
1. <label>
2.   Name:
3.   <input type="text" name="name" id="name" v-model="name" />
4. </label>
```

辅助技术更好地支持用匹配的 `id` 显式设置标签。

aria-label

你也可以给输入一个带有 `aria-label` [\(opens new window\)](#) 的可访问名称。

```
1. <label for="name">Name</label>
2. <input
3.   type="text"
4.   name="name"
5.   id="name"
6.   v-model="name"
7.   :aria-label="nameLabel"
8. />
```

请随意在 Chrome DevTools 中检查此元素，以查看可访问名称是如何更改的:

Styles Event Listeners DOM Breakpoints Properties Accessibility >>

▼ 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

使用 `aria-labelledby` [\(opens new window\)](#) 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`：

```

1. <form
2.   class="demo"
3.   action="/dataCollectionLocation"
4.   method="post"
5.   autocomplete="on"
6. >
7.   <h1 id="billing">Billing</h1>
8.   <div class="form-item">
9.     <label for="name">Name:</label>
10.    <input
11.      type="text"
12.      name="name"
13.      id="name"
14.      v-model="name"
15.      aria-labelledby="billing name"
16.    />
17.  </div>
18.  <button type="submit">Submit</button>
19. </form>

```

Styles Event Listeners DOM Breakpoints Properties Accessibility >>

▼ 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](#) (opens new window) 的用法与 `aria-labelledby` 相同，预期提供了用户可能需要的附加信息的描述。这可用于描述任何输入的标准：

```

1. <form
2.   class="demo"
3.   action="/dataCollectionLocation"
4.   method="post"
5.   autocomplete="on"
6. >
7.   <h1 id="billing">Billing</h1>
8.   <div class="form-item">
9.     <label for="name">Full Name:</label>
10.    <input
11.      type="text"
12.      name="name"
13.      id="name"
14.      v-model="name"
15.      aria-labelledby="billing name"
16.      aria-describedby="nameDescription"
17.    />
18.    <p id="nameDescription">Please provide first and last name.</p>
19.  </div>

```

```
20.   <button type="submit">Submit</button>
21. </form>
```

你可以通过使用 Chrome 开发工具来查看说明：

The screenshot shows the Accessibility panel in Chrome DevTools. The 'ARIA Attributes' section lists:

- aria-labelledby: billing name
- aria-describedby: nameDescription

The 'Computed Properties' section shows the 'Name' property as "Billing Full Name:". Underneath, the 'aria-labelledby' property is expanded to show a list of elements: h1#billina"Billina", input#name h1#billing :", aria-label: Not specified, From-label (for): Label "", placeholder: Not specified, aria-placeholder: Not specified, and title: Not specified. Below this, the 'Description' is "Please provide first and last name.", the 'Role' is 'textbox', and various other accessibility-related properties like 'Invalid user entry', 'Focusable', 'Editable', 'Can set value', 'Multi-line', 'Read-only', and 'Required' are listed as false or true. The 'Described by' section points to p#nameDescription, and the 'Labeled by' section points to h1#billing"Billing" and input#name"Full Name:".

占位符

避免使用占位符，因为它们可能会混淆许多用户。

占位符的一个问题是默认情况下它们不符合[颜色对比标准](#) (opens new window)；修复颜色对比度会使占位符看起来像输入字段中预填充的数据。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：

最好提供用户在任何输入之外填写表单所需的所有信息。

操作指南

为输入字段添加说明时，请确保将其正确链接到输入。你可以提供附加指令并在 `aria-labelledby`

(opens new window) 内绑定多个 id。这使得设计更加灵活。

```
1. <fieldset>
```

```

2.   <legend>Using aria-labelledby</legend>
3.   <label id="date-label" for="date">Current Date:</label>
4.   <input
5.     type="date"
6.     name="date"
7.     id="date"
8.     aria-labelledby="date-label date-instructions"
9.   />
10.  <p id="date-instructions">MM/DD/YYYY</p>
11. </fieldset>

```

或者，你可以用 `aria-describedby` [\(opens new window\)](#) 将指令附加到输入。

```

1. <fieldset>
2.   <legend>Using aria-describedby</legend>
3.   <label id="dob" for="dob">Date of Birth:</label>
4.   <input type="date" name="dob" id="dob" aria-describedby="dob-instructions" />
5.   <p id="dob-instructions">MM/DD/YYYY</p>
6. </fieldset>

```

隐藏内容

通常不建议直观地隐藏标签，即使输入具有可访问的名称。但是，如果输入的功能可以与周围的内容一起理解，那么我们可以隐藏视觉标签。

让我们看看这个搜索字段：

```

1. <form role="search">
2.   <label for="search" class="hidden-visually">Search: </label>
3.   <input type="text" name="search" id="search" v-model="search" />
4.   <button type="submit">Search</button>
5. </form>

```

我们可以这样做，因为搜索按钮将帮助可视化用户识别输入字段的用途。

我们可以使用 CSS 直观地隐藏元素，但可以将它们用于辅助技术：

```

1. .hidden-visually {
2.   position: absolute;
3.   overflow: hidden;
4.   white-space: nowrap;

```

```

5.   margin: 0;
6.   padding: 0;
7.   height: 1px;
8.   width: 1px;
9.   clip: rect(0 0 0 0);
10.  clip-path: inset(100%);
11. }

```

aria-hidden="true"

添加 `aria-hidden="true"` 将隐藏辅助技术中的元素，但使其在视觉上对其他用户可用。不要把它用在可聚焦的元素上，纯粹用于装饰性的、复制的或屏幕外的内容上。

```

1. <p>This is not hidden from screen readers.</p>
2. <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。

也可以使用输入创建按钮：

```

1. <form action="/dataCollectionLocation" method="post" autocomplete="on">
2.   <!-- Buttons -->
3.   <button type="button">Cancel</button>
4.   <button type="submit">Submit</button>
5.
6.   <!-- Input buttons -->
7.   <input type="button" value="Cancel" />
8.   <input type="submit" value="Submit" />
9. </form>

```

功能图像

你可以使用此技术创建功能图像。

- Input 字段
 - 这些图像将作为表单上的提交类型按钮

```

1. <form role="search">
2.   <label for="search" class="hidden-visually">Search: </label>
3.   <input type="text" name="search" id="search" v-model="search" />

```

```
4.   <input
5.     type="image"
6.     class="btnImg"
7.     src="https://img.icons8.com/search"
8.     alt="Search"
9.   />
10. </form>
```

- 图标

```
1. <form role="search">
2.   <label for="searchIcon" class="hidden-visually">Search: </label>
3.   <input type="text" name="searchIcon" id="searchIcon" v-model="searchIcon" />
4.   <button type="submit">
5.     <i class="fas fa-search" aria-hidden="true"></i>
6.     <span class="hidden-visually">Search</span>
7.   </button>
8. </form>
```

标准

万维网联盟 (W3C) 网络可访问性倡议 (WAI) 为不同的组件制定了 Web 可访问性标准:

- [用户代理无障碍指南 \(UAAG\)](#) [↗] (opens new window)
 - 网络浏览器和媒体播放器, 包括一些辅助技术
- [创作工具辅助功能指南 \(ATAG\)](#) [↗] (opens new window)
 - 创作工具
- [网络内容无障碍指南 \(WCAG\)](#) [↗] (opens new window)
 - web 内容—由开发人员、创作工具和可访问性评估工具使用

网络内容无障碍指南 (WCAG)

[WCAG 2.1](#) [↗] (opens new window) 在 [WCAG 2.0](#) [↗] (opens new window) 上进行了扩展, 允许通过处理 web 的变化来实现新技术。W3C 鼓励在开发或更新 Web 可访问性策略时使用 WCAG 的最新版本。

WCAG 2.1 四大指导原则 (缩写 POUR):

- [可感知性](#) [↗] (opens new window)
 - 用户必须能够感知所渲染的信息
- [可操作性](#) [↗] (opens new window)
 - 表单界面, 控件和导航是可操作的
- [可理解性](#) [↗] (opens new window)
 - 信息和用户界面的操作必须为所有用户所理解
- [稳健性](#) [↗] (opens new window)
 - 随着技术的进步, 用户必须能够访问内容




Web 可访问性倡议—可访问的富互联网应用程序 (WAI-ARIA)

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。

- [可访问的富 Internet 应用程序 \(WAI-ARIA\) 1.2](#) [↗] (opens new window)
- [WAI-ARIA 创造实践 1.2](#) [↗] (opens new window)

资源

文档

- [WCAG 2.0](#)  (opens new window)
- [WCAG 2.1](#)  (opens new window)
- [可访问的富 Internet 应用程序 \(WAI-ARIA\) 1.2](#)  (opens new window)
- [WAI-ARIA 创作实践 1.2](#)  (opens new window)

辅助技术

- 屏幕阅读器
 - [NVDA](#)  (opens new window)
 - [VoiceOver](#)  (opens new window)
 - [JAWS](#)  (opens new window)
 - [ChromeVox](#)  (opens new window)
- 缩放工具
 - [MAGic](#)  (opens new window)
 - [ZoomText](#)  (opens new window)
 - [Magnifier](#)  (opens new window)

测试

- 自动化工具
 - [Lighthouse](#)  (opens new window)
 - [WAVE](#)  (opens new window)
- 颜色工具
 - [WebAim Color Contrast](#)  (opens new window)

- [WebAim Link Color Contrast](#) [↗] (opens new window)
- 其他辅助性工具
 - [HeadingMap](#) [↗] (opens new window)
 - [Color Oracle](#) [↗] (opens new window)
 - [Focus Indicator](#) [↗] (opens new window)
 - [NerdeFocus](#) [↗] (opens new window)

用户

世界卫生组织估计，世界 15%的人口患有某种形式的残疾，其中 2 - 4%的人严重残疾。估计全世界有 10 亿人，使残疾人成为世界上最大的少数群体。

残疾的种类繁多，大致可分为四类：

- [视觉的](#) [↗] (*opens new window*) - 这些用户可以受益于使用屏幕阅读器、屏幕放大、控制屏幕对比度或盲文显示。
- [听觉的](#) [↗] (*opens new window*) - 这些用户可以从字幕、文字记录或手语视频中获益。
- [运动的](#) [↗] (*opens new window*) - 这些用户可以从一系列[运动障碍辅助技术](#) [↗] (*opens new window*)中受益：语音识别软件、眼球跟踪、单开关接入、头棒、单开关接入、sip 和 puff 开关、超大轨迹球鼠标、自适应键盘或其他辅助技术。
- [认知](#) [↗] (*opens new window*) - 这些用户可以从补充媒体、内容的结构化组织、清晰和简单的写作中获益。

请从 [WebAim](#) 查看以下链接，以使用户了解：

- [web 可达性透视：探索对每个人的影响和益处](#) [↗] (opens new window)
- [网络用户的故事](#) [↗] (opens new window)

- [介绍](#)
- [v-for 中的 Ref 数组](#)
- [异步组件](#)
- [attribute 强制行为](#)
- [自定义指令](#)
- [自定义元素交互](#)
- [Data 选项](#)
- [事件 API](#)
- [过滤器](#)
- [片段](#)
- [函数式组件](#)
- [全局 API](#)
- [全局 API Treeshaking](#)
- [内联模板 Attribute](#)
- [key attribute](#)
- [按键修饰符](#)
- [在 prop 的默认函数中访问 this](#)
- [渲染函数 API](#)
- [Slot 统一](#)
- [过渡的 class 名更改](#)
- [v-model](#)
- [v-if 与 v-for 的优先级对比](#)
- [v-bind 合并行为](#)

介绍

INFO

刚接触 Vue.js? 先从[基础指南](#)开始吧。

本指南主要是为有 Vue 2 经验的用户希望了解 Vue 3 的新功能和更改而提供的。在试用 **Vue 3** 之前, 你不必从头阅读这些内容。虽然看起来有很多变化, 但很多你已经了解和喜欢 Vue 的部分仍是一样的。不过我们希望尽可能全面, 并为每处变化提供详细的例子。

- [快速开始](#)
- [值得注意的新特性](#)
- [重大改变](#)
- [支持的库](#)

概览

开始学习 Vue 3 [Vue Mastery](#) [↗] ([opens new window](#))。

快速开始

- 通过 CDN:

```
<script src="https://unpkg.com/vue@next"></script>
```
- 通过 [Codepen](#) [↗] ([opens new window](#)) 的浏览器 playground
- 脚手架 [Vite](#) [↗] ([opens new window](#)):

```
1. npm init vite-app hello-vue3 # OR yarn create vite-app hello-vue3
```

- 脚手架 [vue-cli](#) [↗] ([opens new window](#)):

```
1. npm install -g @vue/cli # OR yarn global add @vue/cli
2. vue create hello-vue3
3. # select vue 3 preset
```

值得注意的新特性

Vue 3 中需要关注的一些新功能包括：

- `createRenderer` API 来自 `@vue/runtime-core` [\(opens new window\)](#) 创建自定义渲染器
- 单文件组件 Composition API 语法糖 (`<script setup>`) [\(opens new window\)](#) 实验性
- 单文件组件状态驱动的 CSS 变量 (`<style vars>`) [\(opens new window\)](#) 实验性
- 单文件组件 `<style scoped>` 现在可以包含全局规则或只针对插槽内容的规则 [\(opens new window\)](#)

重大改变

提示

我们仍在开发 Vue 3 的专用迁移版本，该版本的行为与 Vue 2 兼容，运行时警告不兼容。如果你计划迁移一个非常重要的 Vue 2 应用程序，我们强烈建议你等待迁移版本完成以获得更流畅的体验。

下面列出了从 2.x 开始的重大更改：

Global API

- 全局 Vue API 已更改为使用应用程序实例
- 全局和内部 API 已经被重构为可 tree-shakable

模板指令

- 组件上 `v-model` 用法已更改
- `<template v-for>` 和非 `v-for` 节点上 `key` 用法已更改
- 在同一元素上使用的 `v-if` 和 `v-for` 优先级已更改
- `v-bind="object"` 现在排序敏感
- `v-for` 中的 `ref` 不再注册 `ref` 数组

组件

- 只能使用普通函数创建功能组件
- `functional` 属性在单文件组件 (SFC) `<template>` 和 `functional` 组件选项被抛弃
- 异步组件现在需要 `defineAsyncComponent` 方法来创建

渲染函数

- 渲染函数 API 改变
- `$scopedSlots` property 已删除, 所有插槽都通过 `$slots` 作为函数暴露
- 自定义指令 API 已更改为与组件生命周期一致
- 一些转换 class 被重命名了:
 - `v-enter` -> `v-enter-from`
 - `v-leave` -> `v-leave-from`
- 组件 watch 选项和实例方法 `$watch` 不再支持点分隔字符串路径, 请改用计算函数作为参数
- 在 Vue 2.x 中, 应用根容器的 `outerHTML` 将替换为根组件模板 (如果根组件没有模板/渲染选项, 则最终编译为模板)。VUE3.x 现在使用应用程序容器的 `innerHTML` 。

其他小改变

- `destroyed` 生命周期选项被重命名为 `unmounted`
- `beforeDestroy` 生命周期选项被重命名为 `beforeUnmount`
- `prop default` 工厂函数不再有权访问 `this` 是上下文
- 自定义指令 API 已更改为与组件生命周期一致
- `data` 应始终声明为函数
- 来自 mixin 的 `data` 选项现在可简单地合并
- `attribute` 强制策略已更改
- 一些过渡 class 被重命名
- 组建 watch 选项和实例方法 `$watch` 不再支持以点分隔的字符串路径。请改用计算属性函数作为参数。
- `<template>` 没有特殊指令的标记 (`v-if/else-if/else` 、 `v-for` 或 `v-slot`) 现在被视为普通元素, 并将生成原生的 `<template>` 元素, 而不是渲染其内部内容。
- 在 Vue 2.x 中, 应用根容器的 `outerHTML` 将替换为根组件模板 (如果根组件没有模板/渲染选项, 则最终编译为模板)。Vue 3.x 现在使用应用容器的 `innerHTML` , 这意味着容器本身不再被视为模板的一部分。

移除 API

- `keyCode` 支持作为 `v-on` 的修饰符
- `$on`, `$off` 和 `$once` 实例方法
- 过滤
- 内联模板 attribute
- `$destroy` 实例方法。用户不应再手动管理单个 Vue 组件的生命周期。

支持的库

我们所有的官方库和工具现在都支持 Vue 3，但大多数仍然处于 beta 状态，并在 npm 的 `next` dist 标签下发布。我们正计划在 2020 年底前稳定所有项目，并将其转换为使用 `latest` 的 `dist` 标签。

Vue CLI

从 v4.5.0 开始，`vue-cli` 现在提供了内置选项，可在创建新项目时选择 Vue 3 预设。现在可以升级 `vue-cli` 并运行 `vue create` 来创建 Vue 3 项目。

Vue Router

Vue Router 4.0 提供了 Vue 3 支持，并有许多突破性的变化，查看 [README](#) [↗] (opens new window) 中完整的细节，

- `npm@next v4.0.0-beta.13` [↗] (opens new window)
- [Github](#) [↗] (opens new window)
- [RFCs](#) [↗] (opens new window)

Vuex

Vuex 4.0 提供了 Vue 3 支持，其 API 与 3.x 基本相同。唯一的突破性变化是 [插件的安装方式](#) [↗] (opens new window)。

- `npm@next v4.0.0-beta.4` [↗] (opens new window)
- [Github](#) [↗] (opens new window)

Devtools Extension

我们正在开发一个新版本的 Devtools，它有一个新的 UI 和经过重构的内部结构，以支持多个 Vue 版本。新版本目前处于测试阶段，目前只支持 Vue 3。Vuex 和路由器的集成也在进行中。

- Chrome: [从 Chrome web 商店中安装](#) [↗] (opens new window)
 - 提示: beta 版本可能与 devtools 的稳定版本冲突，因此你可能需要暂时禁用稳定版本，以便 beta 版本正常工作。

- Firefox: [下载签名扩展](#) [↗] (opens new window) (assets 下的 `.xpi` 文件)

IDE 支持

推荐使用 [VSCode](#) [↗] (opens new window) 和我们官方拓展 [Vetur](#) [↗] (opens new window), 它为 Vue 3 提供了全面的 IDE 支持

其他项目

项目	npm	仓库
<code>@vue/babel-plugin-jsx</code>	<code>npm v1.0.0-rc.3</code> [↗] (opens new window)	[Github [↗] (opens new window)]
<code>eslint-plugin-vue</code>	<code>npm@next v7.0.0-beta.4</code> [↗] (opens new window)	[Github [↗] (opens new window)]
<code>@vue/test-utils</code>	<code>npm@next v2.0.0-beta.7</code> [↗] (opens new window)	[Github [↗] (opens new window)]
<code>vue-class-component</code>	<code>npm@next v8.0.0-beta.4</code> [↗] (opens new window)	[Github [↗] (opens new window)]
<code>vue-loader</code>	<code>npm@next v16.0.0-beta.8</code> [↗] (opens new window)	[Github [↗] (opens new window)]
<code>rollup-plugin-vue</code>	<code>npm@next v6.0.0-beta.8</code> [↗] (opens new window)	[Github [↗] (opens new window)]

v-for 中的 Ref 数组

breaking

在 Vue 2 中，在 `v-for` 里使用的 `ref` attribute 会用 `ref` 数组填充相应的 `$refs` property。当存在嵌套的 `v-for` 时，这种行为会变得不明确且效率低下。

在 Vue 3 中，这样的用法将不再在 `$ref` 中自动创建数组。要从单个绑定获取多个 `ref`，请将 `ref` 绑定到一个更灵活的函数上（这是一个新特性）：

```
1. <div v-for="item in list" :ref="setItemRef"></div>
```

结合选项式 API：

```
1. export default {
2.   data() {
3.     return {
4.       itemRefs: []
5.     }
6.   },
7.   methods: {
8.     setItemRef(el) {
9.       this.itemRefs.push(el)
10.    }
11.  },
12.  beforeUpdate() {
13.    this.itemRefs = []
14.  },
15.  updated() {
16.    console.log(this.itemRefs)
17.  }
18. }
```

结合组合式 API：

```
1. import { ref, onBeforeUpdate, onUpdated } from 'vue'
2.
3. export default {
4.   setup() {
5.     let itemRefs = []
6.     const setItemRef = el => {
```



```
7.     itemRefs.push(e1)
8.   }
9.   onBeforeUpdate(() => {
10.    itemRefs = []
11.  })
12.  onUpdated(() => {
13.    console.log(itemRefs)
14.  })
15.  return {
16.    itemRefs,
17.    setItemRef
18.  }
19. }
20. }
```

注意：

- `itemRefs` 不必是数组：它也可以是一个对象，其 `ref` 会通过迭代的 `key` 被设置。
- 如果需要，`itemRef` 也可以是响应式的且可以被监听。

异步组件

new

概览

以下是对变化的高层次概述：

- 新的 `defineAsyncComponent` 助手方法，用于显式地定义异步组件
- `component` 选项重命名为 `loader`
- Loader 函数本身不再接收 `resolve` 和 `reject` 参数，且必须返回一个 Promise

如需更深入的解释，请继续阅读！

介绍

以前，异步组件是通过将组件定义为返回 Promise 的函数来创建的，例如：

```
1. const asyncPage = () => import('./NextPage.vue')
```

或者，对于带有选项的更高阶的组件语法：

```
1. const asyncPage = {  
2.   component: () => import('./NextPage.vue'),  
3.   delay: 200,  
4.   timeout: 3000,  
5.   error: ErrorComponent,  
6.   loading: LoadingComponent  
7. }
```

3.x 语法

现在，在 Vue 3 中，由于函数式组件被定义为纯函数，因此异步组件的定义需要通过将其包装在新的

`defineAsyncComponent` 助手方法中来显式地定义：

```
1. import { defineAsyncComponent } from 'vue'  
2. import ErrorComponent from './components/ErrorComponent.vue'  
3. import LoadingComponent from './components/LoadingComponent.vue'  
4.
```

```

5. // 不带选项的异步组件
6. const asyncPage = defineAsyncComponent(() => import('./NextPage.vue'))
7.
8. // 带选项的异步组件
9. const asyncPageWithOptions = defineAsyncComponent({
10.   loader: () => import('./NextPage.vue'),
11.   delay: 200,
12.   timeout: 3000,
13.   errorComponent: ErrorComponent,
14.   loadingComponent: LoadingComponent
15. })

```

对 2.x 所做的另一个更改是，`component` 选项现在被重命名为 `loader`，以便准确地传达不能直接提供组件定义的信息。

```

1. import { defineAsyncComponent } from 'vue'
2.
3. const asyncPageWithOptions = defineAsyncComponent({
4.   loader: () => import('./NextPage.vue'),
5.   delay: 200,
6.   timeout: 3000,
7.   error: ErrorComponent,
8.   loading: LoadingComponent
9. })

```

此外，与 2.x 不同，`loader` 函数不再接收 `resolve` 和 `reject` 参数，且必须始终返回 Promise。

```

1. // 2.x 版本
2. const oldAsyncComponent = (resolve, reject) => {
3.   /* ... */
4. }
5.
6. // 3.x 版本
7. const asyncComponent = defineAsyncComponent(
8.   () =>
9.     new Promise((resolve, reject) => {
10.      /* ... */
11.    })
12. )

```

有关异步组件用法的详细信息，请参阅：

- [指南：动态 & 异步组件](#)

attribute 强制行为

breaking

信息

这是一个低级的内部 API 更改，不会影响大多数开发人员。

概览

下面是对这些变化的高层次总结：

- 删除枚举 attribute 的内部概念，并将这些 attribute 视为普通的非布尔 attribute
- 重大改变：如果值为布尔值，则不再删除 attribute `false`。相反，它被设置为 `attr="false"`。移除 attribute，使用 `null` 或者 `undefined`。

如需更深入的解释，请继续阅读！

2.x 语法

在 2.x，我们有以下策略来强制 `v-bind` 的值：

- 对于某些 attribute/元素对，Vue 始终使用相应的 IDL attribute(property)：比如 `value` 的 `<input>`，`<select>`，`<progress>`，等等 [\(opens new window\)](#)。
- 对于“布尔 attribute [\(opens new window\)](#)”和 `xlinks` [\(opens new window\)](#)，如果它们是 `falsy` 的，Vue 会移除它们 (`undefined`，`null` or `false` [\(opens new window\)](#)) 另外加上它们 (见[这里](#) [\(opens new window\)](#))和 [这里](#) [\(opens new window\)](#))。
- 对于“枚举 attribute [\(opens new window\)](#)” (目前 `contenteditable`，`draggable` 和 `spellcheck`)，Vue 会尝试强制 [\(opens new window\)](#)将它们串起来 (目前对 `contenteditable` 做了特殊处理，修复 [vuejs/vue#9397](#) [\(opens new window\)](#))。
- 对于其他 attribute，我们移除了 `falsy` 值 (`undefined`，`null`，or `false`)

并按原样设置其他值（见[这里](#) [↗] (opens new window)）。

下表描述了 Vue 如何使用普通非布尔 attribute 强制“枚举 attribute”：

绑定表达式	foo 正常	draggable 枚举
<code>:attr="null"</code>	/	<code>draggable="false"</code>
<code>:attr="undefined"</code>	/	/
<code>:attr="true"</code>	<code>foo="true"</code>	<code>draggable="true"</code>
<code>:attr="false"</code>	/	<code>draggable="false"</code>
<code>:attr="0"</code>	<code>foo="0"</code>	<code>draggable="true"</code>
<code>attr=""</code>	<code>foo=""</code>	<code>draggable="true"</code>
<code>attr="foo"</code>	<code>foo="foo"</code>	<code>draggable="true"</code>
<code>attr</code>	<code>foo=""</code>	<code>draggable="true"</code>

从上表可以看出，当前实现 `true` 强制为 `'true'` 但如果 attribute 为 `false`，则移除该 attribute。这也导致了不一致性，并要求用户在非常常见的用例中手动强制布尔值为字符串，例如 `aria-*` attribute 像 `aria-selected`，`aria-hidden`，等等。

3.x 语法

我们打算放弃“枚举 attribute”的内部概念，并将它们视为普通的非布尔 HTML attribute。

- 这解决了普通非布尔 attribute 和“枚举 attribute”之间的不一致性
- 它还可以使用 `'true'` 和 `'false'` 以外的值，甚至可以使用 `contenteditable` 等 attribute 的关键字`

对于非布尔 attribute，如果 attribute 为 `false`，Vue 将停止删除它们，相反强制它们为 `'false'`。

- 这解决了 `true` 和 `false` 之间的不一致性，并使输出 `aria-*` attributes 更容易

下表描述了新行为：

绑定表达式	foo 正常	draggable 枚举
<code>:attr="null"</code>	/	/ †
<code>:attr="undefined"</code>	/	/
<code>:attr="true"</code>	<code>foo="true"</code>	<code>draggable="true"</code>
<code>:attr="false"</code>	<code>foo="false"</code> †	<code>draggable="false"</code>
<code>:attr="0"</code>	<code>foo="0"</code>	<code>draggable="0"</code> †
<code>attr=""</code>	<code>foo=""</code>	<code>draggable=""</code> †

<code>attr="foo"</code>	<code>foo="foo"</code>	<code>draggable="foo"</code> †
<code>attr</code>	<code>foo=""</code>	<code>draggable=""</code> †

†: 变更

布尔 attributes 的强制保持不变。

迁移策略

枚举 attribute

缺少枚举 attribute 和 `attr="false"` 可能会产生不同的 IDL attribute 值 (将反映实际状态), 描述如下:

缺少枚举attr	IDL attr & 值
<code>contenteditable</code>	<code>contentEditable</code> → <code>'inherit'</code>
<code>draggable</code>	<code>draggable</code> → <code>false</code>
<code>spellcheck</code>	<code>spellcheck</code> → <code>true</code>

为了保持原有的行为, 并且我们将强制使用 `false` 为 `'false'`, 在 3.x Vue 中, 开发人员需要将 `v-bind` 表达式解析为 `false` 或 `'false'`, 表示 `contenteditable` 和 `spellcheck`。

在 2.x 中, 枚举 attribute 的无效值被强制为 `'true'`。这通常是无意的, 不太可能大规模依赖。在 3.x 中, 应显式指定 `true` 或 `'true'`。

将 `false` 强制为 `'false'` 而不是删除 attribute

在 3.x, `null` 或 `undefined` 应用于显式删除 attribute。

2.x 和 3.x 行为的比较

Attributes	v-bind value 2.x	v-bind value 3.x	HTML 输出
2.x “枚举attribute” i.e. <code>contenteditable</code> , <code>draggable</code> and <code>spellcheck</code> .	<code>undefined</code> , <code>false</code>	<code>undefined</code> , <code>null</code>	<i>removed</i>
	<code>true</code> , <code>'true'</code> , <code>''</code> , <code>1</code> , <code>'foo'</code>	<code>true</code> , <code>'true'</code>	<code>"true"</code>
	<code>null</code> , <code>'false'</code>	<code>false</code> , <code>'false'</code>	<code>"false"</code>
其他非布尔attribute eg. <code>aria-checked</code> , <code>tabindex</code> ,	<code>undefined</code> , <code>null</code> , <code>false</code>	<code>undefined</code> , <code>null</code>	<i>removed</i>

		'false'	
--	--	---------	--

自定义指令

breaking

概览

下面是对变更的简要总结：

- API 已重命名，以便更好地与组件生命周期保持一致
- 自定义指令将由子组件通过 `v-bind="$attrs"`

更多信息，请继续阅读！

2.x 语法

在 Vue 2，自定义指令是通过使用下面列出的钩子来创建的，这些钩子都是可选的

- **bind** - 指令绑定到元素后发生。只发生一次。
- **inserted** - 元素插入父 DOM 后发生。
- **update** - 当元素更新，但子元素尚未更新时，将调用此钩子。
- **componentUpdated** - 一旦组件和子级被更新，就会调用这个钩子。
- **unbind** - 一旦指令被移除，就会调用这个钩子。也只调用一次。

下面是一个例子：

```
1. <p v-highlight="yellow">高亮显示此文本亮黄色</p>
```

```
1. Vue.directive('highlight', {
2.   bind(el, binding, vnode) {
3.     el.style.background = binding.value
4.   }
5. })
```

在这里，在这个元素的初始设置中，指令通过传递一个值来绑定样式，该值可以通过应用程序更新为不同的值。

3.x 语法

然而，在 Vue 3 中，我们为自定义指令创建了一个更具凝聚力的 API。正如你所看到的，它们与我

们的组件生命周期方法有很大的不同，即使我们正与类似的事件钩子，我们现在把它们统一起来了：

- `bind` → **`beforeMount`**
- `inserted` → **`mounted`**
- **`beforeUpdate`**：新的！这是在元素本身更新之前调用的，很像组件生命周期钩子。
- `update` → 移除！有太多的相似之处要更新，所以这是多余的，请改用 `updated`。
- `componentUpdated` → **`updated`**
- **`beforeUnmount`**：新的！与组件生命周期钩子类似，它将在卸载元素之前调用。
- `unbind` → **`unmounted`**

最终 API 如下：

```

1. const MyDirective = {
2.   beforeMount(el, binding, vnode, prevVnode) {},
3.   mounted() {},
4.   beforeUpdate() {},
5.   updated() {},
6.   beforeUnmount() {}, // 新
7.   unmounted() {}
8. }
```

生成的 API 可以这样使用，与前面的示例相同：

```
1. <p v-highlight="yellow">高亮显示此文本亮黄色</p>
```

```

1. const app = Vue.createApp({})
2.
3. app.directive('highlight', {
4.   beforeMount(el, binding, vnode) {
5.     el.style.background = binding.value
6.   }
7. })
```

既然定制指令生命周期钩子映射了组件本身的那些，那么它们就更容易推理和记住了！

Edge Case: Accessing the component instance

It's generally recommended to keep directives independent of the component instance they are used in. Accessing the instance from within a custom directive is often a sign that the directive should rather be a component itself. However, there are situations where this actually makes sense.

In Vue 2, the component instance had to be accessed through the `vnode` argument:

```
1. bind(el, binding, vnode) {
2.   const vm = vnode.context
3. }
```

In Vue 3, the instance is now part of the `binding` :

```
1. mounted(el, binding, vnode) {
2.   const vm = binding.instance
3. }
```

实施细节

在 Vue 3 中, 我们现在支持片段, 这允许我们为每个组件返回多个 DOM 节点。你可以想象, 对于具有多个 `` 的组件或一个表的子元素这样的组件有多方便:

```
1. <template>
2.   <li>Hello</li>
3.   <li>Vue</li>
4.   <li>Devs!</li>
5. </template>
```

如此灵活, 我们可能会遇到一个定制指令的问题, 它可能有多个根节点。

因此, 自定义指令现在作为虚拟 DOM 节点数据的一部分包含在内。当在组件上使用自定义指令时, 钩子作为无关的 prop 传递到组件, 并以 `this.$attrs` 结束。

这也意味着可以像这样在模板中直接挂接到元素的生命周期中, 这在涉及到自定义指令时非常方便:

```
1. <div @vnodeMounted="myHook" />
```

这与属性 fallthrough 行为是一致的, 因此, 当子组件在内部元素上使用 `v-bind="$attrs"` 时, 它也将应用对其使用的任何自定义指令。

自定义元素交互

breaking

概览

- 非兼容：自定义元素白名单现在在模板编译期间执行，应该通过编译器选项而不是运行时配置来配置。
- 非兼容：特定 `is` prop 用法仅限于保留的 `<component>` 标记。
- 新：有了新的 `v-is` 指令来支持 2.x 用例，其中在原生元素上使用了 `v-is` 来处理原生 HTML 解析限制。

自主定制元素

如果我们想添加在 Vue 外部定义的自定义元素（例如使用 Web 组件 API），我们需要“指示”Vue 将其视为自定义元素。让我们以下面的模板为例。

```
1. <plastic-button></plastic-button>
```

2.x 语法

在 Vue 2.x 中，将标记作为自定义元素白名单是通过 `Vue.config.ignoredElements`：

```
1. // 这将使Vue忽略在Vue外部定义的自定义元素
2. // （例如：使用 Web Components API）
3.
4. Vue.config.ignoredElements = ['plastic-button']
```

3.x 语法

在 Vue 3.0 中，此检查在模板编译期间执行指示编译器将 `<plastic-button>` 视为自定义元素：

- 如果使用生成步骤：将 `isCustomElement` 传递给 Vue 模板编译器，如果使用 `vue-loader`，则应通过 `vue-loader` 的 `compilerOptions` 选项传递：

```
1. // webpack 中的配置
2. rules: [
```

```

3.   {
4.     test: /\.vue$/,
5.     use: 'vue-loader',
6.     options: {
7.       compilerOptions: {
8.         isCustomElement: tag => tag === 'plastic-button'
9.       }
10.    }
11.  }
12.  // ...
13. ]

```

- 如果使用动态模板编译，请通过 `app.config.isCustomElement` 传递：

```

1. const app = Vue.createApp({})
2. app.config.isCustomElement = tag => tag === 'plastic-button'

```

需要注意的是，运行时配置只会影响运行时模板编译——它不会影响预编译的模板。

定制内置元素

自定义元素规范提供了一种将自定义元素用作[自定义内置模板](#) [↗] (opens new window)的方法，方法是向内置元素添加 `is` 属性：

```
1. <button is="plastic-button">点击我!</button>
```

Vue 对 `is` 特殊 prop 的使用是在模拟 native attribute 在浏览器中普遍可用之前的作用。但是，在 2.x 中，它被解释为渲染一个名为 `plastic-button` 的 Vue 组件，这将阻止上面提到的自定义内置元素的原生使用。

在 3.0 中，我们仅将 Vue 对 `is` 属性的特殊处理限制到 `<component>` tag。

- 在保留的 `<component>` tag 上使用时，它的行为将与 2.x 中完全相同；
- 在普通组件上使用时，它的行为将类似于普通 prop：

```
1. <foo is="bar" />
```

- 2.x 行为：渲染 `bar` 组件。
- 3.x 行为：通过 `is` prop 渲染 `foo` 组件。

- 在普通元素上使用时，它将作为 `is` 选项传递给 `createElement` 调用，并作为原生 `attribute` 渲染，这支持使用自定义的内置元素。

```
1. <button is="plastic-button">点击我！</button>
```

- 2.x 行为：渲染 `plastic-button` 组件。
- 3.x 行为：通过回调渲染原生的 `button`。

```
1. document.createElement('button', { is: 'plastic-button' })
```

v-is 用于 DOM 内模板解析解决方案

提示：本节仅影响直接在页面的 HTML 中写入 Vue 模板的情况。在 DOM 模板中使用时，模板受原生 HTML 解析规则的约束。一些 HTML 元素，例如 ``，``，`<table>` 和 `<select>` 对它们内部可以出现的元素有限制，和一些像 ``，`<tr>`，和 `<option>` 只能出现在某些其他元素中。

2x 语法

在 Vue 2 中，我们建议通过在原生 tag 上使用 `is` prop 来解决这些限制：

```
1. <table>
2.   <tr is="blog-post-row"></tr>
3. </table>
```

3.x 语法

随着 `is` 的行为变化，我们引入了一个新的指令 `v-is`，用于解决这些情况：

```
1. <table>
2.   <tr v-is="'blog-post-row'"></tr>
3. </table>
```

WARNING

`v-is` 函数像一个动态的 2.x `:is` 绑定—因此，要按注册名称渲染组件，其值应为 JavaScript 字符串文本：

```
1. <!-- 不正确，不会渲染任何内容 -->
```

```
2. <tr v-is="blog-post-row"></tr>
3.
4. <!-- 正确 -->
5. <tr v-is="'blog-post-row'"></tr>
```

迁移策略

- 替换 `config.ignoredElements` 与 `vue-loader` 的 `compilerOptions` (使用 `build` 步骤) 或 `app.config.isCustomElement` (使用动态模板编译)
- 将所有非 `<component>` tags 与 `is` 用法更改为 `<component is="...">` (对于 SFC 模板) 或 `v-is` (对于 DOM 模板)。

Data 选项

breaking

概览

- **breaking:** `data` 组件选项声明不再接收纯 JavaScript `object`，而需要 `function` 声明。

当合并来自 `mixin` 或 `extend` 的多个 `data` 返回值时，现在是浅层次合并的而不是深层次合并的(只合并根级属性)。

2.x Syntax

在 2.x 中，开发者可以定义 `data` 选项是 `object` 或者是 `function`。

例如：

```
1. <!-- Object 声明 -->
2. <script>
3.   const app = new Vue({
4.     data: {
5.       apiKey: 'a1b2c3'
6.     }
7.   })
8. </script>
9.
10. <!-- Function 声明 -->
11. <script>
12.   const app = new Vue({
13.     data() {
14.       return {
15.         apiKey: 'a1b2c3'
16.       }
17.     }
18.   })
19. </script>
```

虽然这对于具有共享状态的根实例提供了一些便利，但是由于只有在根实例上才有可能，这导致了混乱。

3.x Update

在 3.x, `data` 选项已标准化为只接受返回 `object` 的 `function`。

使用上面的示例, 代码只有一个可能的实现:

```
1. <script>
2.   import { createApp } from 'vue'
3.
4.   createApp({
5.     data() {
6.       return {
7.         apiKey: 'a1b2c3'
8.       }
9.     }
10.  }).mount('#app')
11. </script>
```

Mixin 合并行为变更

此外, 当来自组件的 `data()` 及其 `mixin` 或 `extends` 基类被合并时, 现在将浅层次执行合并:

```
1. const Mixin = {
2.   data() {
3.     return {
4.       user: {
5.         name: 'Jack',
6.         id: 1
7.       }
8.     }
9.   }
10. }
11. const CompA = {
12.   mixins: [Mixin],
13.   data() {
14.     return {
15.       user: {
16.         id: 2
17.       }
18.     }
19.   }
20. }
```

```
19.   }  
20. }
```

在 Vue 2.x 中，生成的 `$data` 是：

```
1. {  
2.   user: {  
3.     id: 2,  
4.     name: 'Jack'  
5.   }  
6. }
```

在 3.0 中，其结果将会是：

```
1. {  
2.   user: {  
3.     id: 2  
4.   }  
5. }
```

迁移策略

对于依赖对象声明的用户，我们建议：

- 将共享数据提取到外部对象并将其用作 `data` 中的 property
- 重写对共享数据的引用以指向新的共享对象

对于依赖 `mixin` 的深度合并行为的用户，我们建议重构代码以完全避免这种依赖，因为 `mixin` 的深度合并非常隐式，这让代码逻辑更难理解和调试。

事件 API

breaking

概览

`$on` , `$off` 和 `$once` 实例方法已被移除, 应用实例不再实现事件触发接口。

2.x 语法

在 2.x 中, Vue 实例可用于触发通过事件触发 API 强制附加的处理程序 (`$on` , `$off` 和 `$once`), 这用于创建 event hub, 以创建在整个应用程序中使用的全局事件侦听器:

```
1. // eventHub.js
2.
3. const eventHub = new Vue()
4.
5. export default eventHub
```

```
1. // ChildComponent.vue
2. import eventHub from './eventHub'
3.
4. export default {
5.   mounted() {
6.     // 添加 eventHub listener
7.     eventHub.$on('custom-event', () => {
8.       console.log('Custom event triggered!')
9.     })
10.  },
11.  beforeDestroy() {
12.    // 移除 eventHub listener
13.    eventHub.$off('custom-event')
14.  }
15. }
```

```
1. // ParentComponent.vue
2. import eventHub from './eventHub'
3.
4. export default {
```

```
5.   methods: {
6.     callGlobalCustomEvent() {
7.       eventHub.$emit('custom-event') // 如果ChildComponent mounted, 控制台中将显示
8.       一条消息
9.     }
10.  }
```

3.x 更新

我们整个从实例中移除了 `$on` , `$off` 和 `$once` 方法, `$emit` 仍然是现有 API 的一部分, 因为它用于触发由父组件以声明方式附加的事件处理程序

迁移策略

例如, 可以通过使用实现事件发射器接口的外部库来替换现有的 event hub `mitt` [\(opens new window\)](#)。

在兼容性构建中也可以支持这些方法。

过滤器

removed

概览

从 Vue 3.0 开始，过滤器已删除，不再支持。

2.x 语法

在 2.x，开发者可以使用过滤器来处理通用文本格式。

例如：

```
1. <template>
2.   <h1>Bank Account Balance</h1>
3.   <p>{{ accountBalance | currencyUSD }}</p>
4. </template>
5.
6. <script>
7.   export default {
8.     props: {
9.       accountBalance: {
10.        type: Number,
11.        required: true
12.      }
13.    },
14.    filters: {
15.      currencyUSD(value) {
16.        return '$' + value
17.      }
18.    }
19.  }
20. </script>
```

虽然这看起来很方便，但它需要一个自定义语法，打破大括号内表达式是“只是 JavaScript”的假设，这不仅有学习成本，而且有实现成本。

3.x 更新

在 3.x 中，过滤器已删除，不再支持。相反地，我们建议用方法调用或计算属性替换它们。

使用上面的例子，这里是一个如何实现它的例子。

```
1. <template>
2.   <h1>Bank Account Balance</h1>
3.   <p>{{ accountInUSD }}</p>
4. </template>
5.
6. <script>
7.   export default {
8.     props: {
9.       accountBalance: {
10.        type: Number,
11.        required: true
12.      }
13.    },
14.    computed: {
15.      accountInUSD() {
16.        return '$' + this.accountBalance
17.      }
18.    }
19.  }
20. </script>
```

迁移策略

我们建议用计算属性或方法代替过滤器，而不是使用过滤器。

全局过滤器

如果在应用中全局注册了过滤器，那么在每个组件中用计算属性或方法调用来替换它可能就没那么方便了。

相反地，你可以通过[全局属性](#)在所有组件中使用它：

```
1. // main.js
2. const app = createApp(App)
3.
4. app.config.globalProperties.$filters = {
5.   currencyUSD(value) {
6.     return '$' + value
```

```
7.   }  
8. }
```

然后，你可以通过 `$filters` 对象修改所有的模板，像下面这样：

```
1. <template>  
2.   <h1>Bank Account Balance</h1>  
3.   <p>{{ $filters.currencyUSD(accountBalance) }}</p>  
4. </template>
```

注意，这种方式只能用于方法中，不可以在计算属性中使用，因为后者只有在单个组件的上下文中定义时才有意义。

片段

new

概览

在 Vue 3 中，组件现在正式支持多根节点组件，即片段！

2.x 语法

在 2.x 中，不支持多根组件，当用户意外创建多根组件时会发出警告，因此，为了修复此错误，许多组件被包装在一个 `<div>` 中。

```
1. <!-- Layout.vue -->
2. <template>
3.   <div>
4.     <header>...</header>
5.     <main>...</main>
6.     <footer>...</footer>
7.   </div>
8. </template>
```

3.x 语法

在 3.x 中，组件现在可以有多个根节点！但是，这确实要求开发者明确定义属性应该分布在哪里。

```
1. <!-- Layout.vue -->
2. <template>
3.   <header>...</header>
4.   <main v-bind="$attrs">...</main>
5.   <footer>...</footer>
6. </template>
```

有关 `attribute` 继承如何工作的详细信息，见[非 Prop Attributes](#)。

函数式组件

breaking

概览

就变化而言，属于高等级内容：

- 在 3.x 中，函数式组件 2.x 的性能提升可以忽略不计，因此我们建议只使用有状态的组件
- 函数式组件只能使用接收 `props` 和 `context` 的普通函数创建（即：`slots`，`attrs`，`emit`）。
- 非兼容变更：`functional` attribute 在单文件组件（SFC）`<template>` 已被移除
- 非兼容变更：`{ functional: true }` 选项在通过函数创建组件已被移除

更多信息，请继续阅读！

介绍

在 Vue 2 中，函数式组件有两个主要用例：

- 作为性能优化，因为它们的初始化速度比有状态组件快得多
- 返回多个根节点

然而，在 Vue 3 中，有状态组件的性能已经提高到可以忽略不计的程度。此外，有状态组件现在还包包括返回多个根节点的能力。

因此，函数式组件剩下的唯一用例就是简单组件，比如创建动态标题的组件。否则，建议你像平常一样使用有状态组件。

2.x 语法

使用 `<dynamic-heading>` 组件，负责提供适当的标题（即：`h1`，`h2`，`h3`，等等），在 2.x 中，这可能是作为单个文件组件编写的：

```
1. // Vue 2 函数式组件示例
2. export default {
3.   functional: true,
4.   props: ['level'],
5.   render(h, { props, data, children }) {
6.     return h(`h${props.level}`, data, children)
```

```

7.   }
8. }

```

或者，对于喜欢在单个文件组件中使用 `<template>` 的用户：

```

1. // Vue 2 函数式组件示例使用 <template>
2. <template functional>
3.   <component
4.     :is="`h${props.level}`"
5.     v-bind="attrs"
6.     v-on="listeners"
7.   />
8. </template>
9.
10. <script>
11. export default {
12.   props: ['level']
13. }
14. </script>

```

3.x 语法

通过函数创建组件

现在在 Vue 3 中，所有的函数式组件都是用普通函数创建的，换句话说，不需要定义 `{ functional: true }` 组件选项。

他们将接收两个参数：`props` 和 `context`。`context` 参数是一个对象，包含组件的 `attrs`，`slots`，和 `emit` property。

此外，现在不是在 `render` 函数中隐式提供 `h`，而是全局导入 `h`。

使用前面提到的 `<dynamic-heading>` 组件的示例，下面是它现在的样子。

```

1. import { h } from 'vue'
2.
3. const DynamicHeading = (props, context) => {
4.   return h(`h${props.level}`, context.attrs, context.slots)
5. }
6.
7. DynamicHeading.props = ['level']
8.

```

```
9. export default DynamicHeading
```

单文件组件 (SFC)

在 3.x 中，有状态组件和函数式组件之间的性能差异已经大大减少，并且在大多数用例中是微不足道的。因此，在 SFCs 上使用 `functional` 的开发人员的迁移路径是删除该 `attribute`，并将 `props` 的所有引用重命名为 `$props`，将 `attrs` 重命名为 `$attrs`。

使用之前的 `<dynamic-heading>` 示例，下面是它现在的样子。

```
1. <template>
2.   <component
3.     v-bind:is="`h${$props.level}`"
4.     v-bind="$attrs"
5.   />
6. </template>
7.
8. <script>
9. export default {
10.   props: ['level']
11. }
12. </script>
```

主要的区别在于：

1. `functional` `attribute` 在 `<template>` 中移除
2. `listeners` 现在作为 `$attrs` 的一部分传递，可以将其删除

下一步

有关新函数式组件的用法和对渲染函数的更改的详细信息，见：

- [迁移：渲染函数](#)
- [指南：渲染函数](#)

全局 API

breaking

Vue 2.x 有许多全局 API 和配置，这些 API 和配置可以全局改变 Vue 的行为。例如，要创建全局组件，可以使用 `Vue.component` 这样的 API：

```
1. Vue.component('button-counter', {
2.   data: () => ({
3.     count: 0
4.   }),
5.   template: '<button @click="count++">Clicked {{ count }} times.</button>'
6. })
```

类似地，使用全局指令的声明方式如下：

```
1. Vue.directive('focus', {
2.   inserted: el => el.focus()
3. })
```

虽然这种声明方式很方便，但它也会导致一些问题。从技术上讲，Vue 2 没有“app”的概念，我们定义的应用只是通过 `new Vue()` 创建的根 Vue 实例。从同一个 Vue 构造函数创建的每个根实例共享相同的全局配置，因此：

- 在测试期间，全局配置很容易意外地污染其他测试用例。用户需要仔细存储原始全局配置，并在每次测试后恢复（例如重置 `Vue.config.errorHandler`）。有些 API 像 `Vue.use` 以及 `Vue.mixin` 甚至连恢复效果的方法都没有，这使得涉及插件的测试特别棘手。实际上，`vue-test-utils` 必须实现一个特殊的 API `createLocalVue` 来处理此问题：

```
1. import { createLocalVue, mount } from '@vue/test-utils'
2.
3. // 建扩展的 `Vue` 构造函数
4. const localVue = createLocalVue()
5.
6. // 在“local”Vue构造函数上“全局”安装插件
7. localVue.use(MyPlugin)
8.
9. // 通过 `localVue` 来挂载选项
10. mount(Component, { localVue })
```

- 全局配置使得在同一页面上的多个“app”之间共享同一个 Vue 副本非常困难，但全局配置不同。

```

1. // 这会影响两个根实例
2. Vue.mixin({
3.   /* ... */
4. })
5.
6. const app1 = new Vue({ el: '#app-1' })
7. const app2 = new Vue({ el: '#app-2' })

```

为了避免这些问题，在 Vue 3 中我们引入...

一个新的全局 API: `createApp`

调用 `createApp` 返回一个应用实例，这是 Vue 3 中的新概念：

```

1. import { createApp } from 'vue'
2.
3. const app = createApp({})

```

应用实例暴露当前全局 API 的子集，经验法则是，任何全局改变 Vue 行为的 API 现在都会移动到应用实例上，以下是当前全局 API 及其相应实例 API 的表：

2.x 全局 API	3.x 实例 API (<code>app</code>)
<code>Vue.config</code>	<code>app.config</code>
<code>Vue.config.productionTip</code>	<i>removed</i> (见下方)
<code>Vue.config.ignoredElements</code>	<code>app.config.isCustomElement</code> (见下方)
<code>Vue.component</code>	<code>app.component</code>
<code>Vue.directive</code>	<code>app.directive</code>
<code>Vue.mixin</code>	<code>app.mixin</code>
<code>Vue.use</code>	<code>app.use</code> (见下方)

所有其他不全局改变行为的全局 API 现在被命名为 `exports`，文档见[全局 API TreeShaking](#)。

`config.productionTip` 移除

在 Vue 3.x 中，“使用生产版本”提示仅在使用“dev + full build”（包含运行时编译器并有警告的构建）时才会显示。

对于 ES 模块构建，由于它们是与 bundler 一起使用的，而且在大多数情况下，CLI 或样板已经正

确地配置了生产环境，所以本技巧将不再出现。

`config.ignoredElements` 替换为 `config.isCustomElement`

引入此配置选项的目的是支持原生自定义元素，因此重命名可以更好地传达它的功能，新选项还需要一个比旧的 `string/RegExp` 方法提供更多灵活性的函数：

```
1. // before
2. Vue.config.ignoredElements = ['my-el', /^ion-/]
3.
4. // after
5. const app = Vue.createApp({})
6. app.config.isCustomElement = tag => tag.startsWith('ion-')
```

重要

在 3.0 中，元素是否是组件的检查已转移到模板编译阶段，因此只有在使用运行时编译器时才考虑此配置选项。如果你使用的是 `runtime-only` 版本 `isCustomElement` 必须通过 `@vue/compiler-dom` 在构建步骤替换——比如，通过 `compilerOptions` `option in vue-loader` [\(opens new window\)](#)。

- 如果 `config.isCustomElement` 当使用仅运行时构建时时，将发出警告，指示用户在生成设置中传递该选项；
- 这将是 Vue CLI 配置中新的顶层选项。

插件使用者须知

插件开发者通常使用 `Vue.use`。例如，官方的 `vue-router` 插件是如何在浏览器环境中自行安装的：

```
1. var inBrowser = typeof window !== 'undefined'
2. /* ... */
3. if (inBrowser && window.Vue) {
4.   window.Vue.use(VueRouter)
5. }
```

由于 `use` 全局 API 在 Vue 3 中不再使用，此方法将停止工作并停止调用 `Vue.use()` 现在将触发警告，于是，开发者必须在应用程序实例上显式指定使用此插件：

```
1. const app = createApp(MyApp)
2. app.use(VueRouter)
```

挂载 App 实例

使用 `createApp(/* options */)` 初始化后，应用实例 `app` 可用于挂载具有 `app.mount(domTarget)` :

```
1. import { createApp } from 'vue'
2. import MyApp from './MyApp.vue'
3.
4. const app = createApp(MyApp)
5. app.mount('#app')
```

经过所有这些更改，我们在指南开头的组件和指令将被改写为如下内容：

```
1. const app = createApp(MyApp)
2.
3. app.component('button-counter', {
4.   data: () => ({
5.     count: 0
6.   }),
7.   template: '<button @click="count++">Clicked {{ count }} times.</button>'
8. })
9.
10. app.directive('focus', {
11.   mounted: el => el.focus()
12. })
13.
14. // 现在所有应用实例都挂载了，与其组件树一起，将具有相同的 "button-counter" 组件 和
15. // "focus" 指令不污染全局环境
16. app.mount('#app')
```

提供/注入 (Provide / Inject)

与在 2.x 根实例中使用 `provide` 选项类似，Vue 3 应用实例还可以提供可由应用内的任何组件注入的依赖项：

```
1. // 在入口
2. app.provide('guide', 'Vue 3 Guide')
3.
4. // 在子组件
5. export default {
6.   inject: {
```

```
7.     book: {
8.       from: 'guide'
9.     }
10.  },
11.  template: `

{{ book }}

`
12. }
```

在应用之间共享配置

在应用之间共享配置（如组件或指令）的一种方法是创建工厂功能，如下所示：

```
1. import { createApp } from 'vue'
2. import Foo from './Foo.vue'
3. import Bar from './Bar.vue'
4.
5. const createMyApp = options => {
6.   const app = createApp(options)
7.   app.directive('focus' /* ... */)
8.
9.   return app
10. }
11.
12. createMyApp(Foo).mount('#foo')
13. createMyApp(Bar).mount('#bar')
```

现在，Foo 和 Bar 实例及其后代中都可以使用 `focus` 指令。

全局 API Treeshaking

breaking

2.x 语法

如果你曾经在 Vue 中手动操作过 DOM，你可能会遇到以下模式：

```
1. import Vue from 'vue'
2.
3. Vue.nextTick(() => {
4.   // 一些和DOM有关的东西
5. })
```

或者，如果你一直在对涉及 `async components` 的应用程序进行单元测试，那么很可能你编写了以下内容：

```
1. import { shallowMount } from '@vue/test-utils'
2. import { MyComponent } from './MyComponent.vue'
3.
4. test('an async feature', async () => {
5.   const wrapper = shallowMount(MyComponent)
6.
7.   // 执行一些DOM相关的任务
8.
9.   await wrapper.vm.$nextTick()
10.
11.   // 运行你的断言
12. })
```

`Vue.nextTick()` 是一个全局的 API 直接暴露在单个 Vue 对象上——事实上，实例方法 `$nextTick()` 只是一个方便的包装 `Vue.nextTick()` 为方便起见，回调的 `this` 上下文自动绑定到当前实例。

模块捆绑程序，如 [webpack](#) [↗] ([opens new window](#)) 支持 `tree-shaking`，这是“死代码消除”的一个花哨术语。不幸的是，由于代码是如何在以前的 Vue 版本中编写的，全局 API `Vue.nextTick()` 不可摇动，将包含在最终捆绑中不管它们实际在哪里使用。

3.x 语法

在 Vue 3 中，全局和内部 API 都经过了重构，并考虑到了 tree-shaking 的支持。因此，全局 API 现在只能作为 ES 模块构建的命名导出进行访问。例如，我们之前的片段现在应该如下所示：

```
1. import { nextTick } from 'vue'
2.
3. nextTick(() => {
4.   // 一些和DOM有关的东西
5. })
```

and

```
1. import { shallowMount } from '@vue/test-utils'
2. import { MyComponent } from './MyComponent.vue'
3. import { nextTick } from 'vue'
4.
5. test('an async feature', async () => {
6.   const wrapper = shallowMount(MyComponent)
7.
8.   // 执行一些DOM相关的任务
9.
10.  await nextTick()
11.
12.  // 运行你的断言
13. })
```

直接调用 `Vue.nextTick()` 将导致臭名昭著的 `undefined is not a function` 错误。

通过这一更改，如果模块绑定器支持 tree-shaking，则 Vue 应用程序中未使用的全局 api 将从最终捆绑包中消除，从而获得最佳的文件大小。

受影响的 API

Vue 2.x 中的这些全局 API 受此更改的影响：

- `Vue.nextTick`
- `Vue.observable` (用 `Vue.reactive` 替换)
- `Vue.version`
- `Vue.compile` (仅全构建)
- `Vue.set` (仅兼容构建)
- `Vue.delete` (仅兼容构建)

内部帮助器

除了公共 api，许多内部组件/帮助器现在也被导出为命名导出，只有当编译器的输出是这些特性时，才允许编译器导入这些特性，例如以下模板：

```
1. <transition>
2.   <div v-show="ok">hello</div>
3. </transition>
```

被编译为类似于以下的内容：

```
1. import { h, Transition, withDirectives, vShow } from 'vue'
2.
3. export function render() {
4.   return h(Transition, [withDirectives(h('div', 'hello'), [[vShow, this.ok]])])
5. }
```

这实际上意味着只有在应用程序实际使用了 `Transition` 组件时才会导入它。换句话说，如果应用程序没有任何 `Transition` 组件，那么支持此功能的代码将不会出现在最终的捆绑包中。

随着全局 tree-shaking，用户只需为他们实际使用的功能“付费”，更好的是，知道了可选特性不会增加不使用它们的应用程序的捆绑包大小，框架大小在将来已经不再是其他核心功能的考虑因素了，如果有的话。

重要

以上仅适用于 [ES Modules builds](#)，用于支持 tree-shaking 的绑定器—UMD 构建仍然包括所有特性，并暴露 Vue 全局变量上的所有内容（编译器将生成适当的输出，以使用全局外的 api 而不是导入）。

插件中的用法

如果你的插件依赖受影响的 Vue 2.x 全局 API，例如：

```
1. const plugin = {
2.   install: Vue => {
3.     Vue.nextTick(() => {
4.       // ...
5.     })
6.   }
7. }
```

在 Vue 3 中，必须显式导入：

```

1. import { nextTick } from 'vue'
2.
3. const plugin = {
4.   install: app => {
5.     nextTick(() => {
6.       // ...
7.     })
8.   }
9. }

```

如果使用 webpack 这样的模块捆绑包，这可能会导致 Vue 的源代码绑定到插件中，而且通常情况下，这并不是你所期望的。防止这种情况发生的一种常见做法是配置模块绑定器以将 Vue 从最终捆绑中排除。对于 webpack，你可以使用 `externals` [\(opens new window\)](#) 配置选项：

```

1. // webpack.config.js
2. module.exports = {
3.   /*...*/
4.   externals: {
5.     vue: 'Vue'
6.   }
7. }

```

这将告诉 webpack 将 Vue 模块视为一个外部库，而不是捆绑它。

如果你选择的模块绑定器恰好是 Rollup [\(opens new window\)](#)，你基本上可以免费获得相同的效果，因为默认情况下，Rollup 会将绝对模块 id（在我们的例子中为 `'vue'`）作为外部依赖项，而不会将它们包含在最终的 bundle 中。但是在绑定期间，它可能会发出一个“将 vue 作为外部依赖” [\(opens new window\)](#) 警告，可使用 `external` 选项抑制该警告：

```

1. // rollup.config.js
2. export default {
3.   /*...*/
4.   external: ['vue']
5. }

```

内联模板 Attribute

breaking

概览

对[内联特性](#) [↗] (opens new window)的支持已被移除。

2.x 语法

在 2.x 中, Vue 为子组件提供了 `inline-template` attribute, 以便将其内部内容用作模板, 而不是将其作为分发内容。

```
1. <my-component inline-template>
2.   <div>
3.     <p>它们被编译为组件自己的模板</p>
4.     <p>不是父级所包含的内容。</p>
5.   </div>
6. </my-component>
```

3.x 语法

将不再支持此功能。

迁移策略

`inline-template` 的大多数用例都假设没有构建工具设置, 所有模板都直接写在 HTML 页面中

选项 #1: 使用 `<script>` 标签

在这种情况下, 最简单的解决方法是将 `<script>` 与其他类型一起使用:

```
1. <script type="text/html" id="my-comp-template">
2.   <div>{{ hello }}</div>
3. </script>
```

在组件中, 使用选择器将模板作为目标:

```

1. const MyComp = {
2.   template: '#my-comp-template'
3.   // ...
4. }

```

这不需要任何构建设置，可以在所有浏览器中工作，不受任何 DOM HTML 解析警告的约束（例如，你可以使用 camelCase prop 名称），并且在大多数 ide 中提供了正确的语法高亮显示。在传统的服务器端框架中，可以将这些模板拆分为服务器模板部分（包括在主 HTML 模板中），以获得更好的可维护性。

选项 #2：默认 Slot

以前使用 `inline-template` 的组件也可以使用默认 slot——进行重构，这使得数据范围更加明确，同时保留了内联编写子内容的便利性：

```

1. <!-- 2.x 语法 -->
2. <my-comp inline-template :msg="parentMsg">
3.   {{ msg }} {{ childState }}
4. </my-comp>
5.
6. <!-- 默认 slot 版本 -->
7. <my-comp v-slot="{ childState }">
8.   {{ parentMsg }} {{ childState }}
9. </my-comp>

```

子级现在应该渲染默认 slot*，而不是不提供模板：

```

1. <!--
2.   在子模板中，在传递时渲染默认slot
3.   在必要的private状态下。
4. -->
5. <template>
6.   <slot :childState="childState" />
7. </template>

```

- 提示：在 3.x, slot 可以渲染为具有原生 `fragments` 支持的根目录！

key attribute

breaking

概览

- **NEW:** 对于 `v-if` / `v-else` / `v-else-if` 的各分支项 `key` 将不再是必须的，因为现在 Vue 会自动生成唯一的 `key`。
 - **BREAKING:** 如果你手动提供 `key`，那么每个分支必须使用唯一的 `key`。你不能通过故意使用相同的 `key` 来强制重用分支。
- **BREAKING:** `<template v-for>` 的 `key` 应该设置在 `<template>` 标签上（而不是设置在它的子节点上）。

背景

特殊的 `key` attribute 被用于提示 Vue 的虚拟 DOM 算法来保持对节点身份的持续跟踪。这样 Vue 可以知道何时能够重用和修补现有节点，以及何时需要对它们重新排序或重新创建。关于其它更多信息，可以查看以下章节：

- [列表渲染：维护状态](#)
- [API Reference：特殊指令 `key`](#)

在条件分支中

Vue 2.x 建议在 `v-if` / `v-else` / `v-else-if` 的分支中使用 `key`。

```
1. <!-- Vue 2.x -->
2. <div v-if="condition" key="yes">Yes</div>
3. <div v-else key="no">No</div>
```

这个示例在 Vue 3.x 中仍能正常工作。但是我们不再建议在 `v-if` / `v-else` / `v-else-if` 的分支中继续使用 `key` attribute，因为没有为条件分支提供 `key` 时，也会自动生成唯一的 `key`。

```
1. <!-- Vue 3.x -->
2. <div v-if="condition">Yes</div>
3. <div v-else>No</div>
```

非兼容变更体现在如果你手动提供了 `key`，那么每个分支都必须使用一个唯一的 `key`。因此大多数情况下都不需要设置这些 `key`。

```

1. <!-- Vue 2.x -->
2. <div v-if="condition" key="a">Yes</div>
3. <div v-else key="a">No</div>
4.
5. <!-- Vue 3.x (recommended solution: remove keys) -->
6. <div v-if="condition">Yes</div>
7. <div v-else>No</div>
8.
9. <!-- Vue 3.x (alternate solution: make sure the keys are always unique) -->
10. <div v-if="condition" key="a">Yes</div>
11. <div v-else key="b">No</div>

```

结合

`<template v-for>`

在 Vue 2.x 中 `<template>` 标签不能拥有 `key`。不过你可以为其每个子节点分别设置 `key`。

```

1. <!-- Vue 2.x -->
2. <template v-for="item in list">
3.   <div :key="item.id">...</div>
4.   <span :key="item.id">...</span>
5. </template>

```

在 Vue 3.x 中 `key` 则应该被设置在 `<template>` 标签上。

```

1. <!-- Vue 3.x -->
2. <template v-for="item in list" :key="item.id">
3.   <div>...</div>
4.   <span>...</span>
5. </template>

```

类似地，当使用 `<template v-for>` 时存在使用 `v-if` 的子节点，`key` 应改为设置在 `<template>` 标签上。

```

1. <!-- Vue 2.x -->
2. <template v-for="item in list">
3.   <div v-if="item.isVisible" :key="item.id">...</div>

```



```
4.   <span v-else :key="item.id">...</span>
5. </template>
6.
7. <!-- Vue 3.x -->
8. <template v-for="item in list" :key="item.id">
9.   <div v-if="item.isVisible">...</div>
10.  <span v-else>...</span>
11. </template>
```

按键修饰符

breaking

概览

以下是变更的简要总结：

- **BREAKING:** 不再支持使用数字（即键码）作为 `v-on` 修饰符
- **BREAKING:** 不再支持 `config.keyCodes`

2.x 语法

在 Vue 2 中，支持 `keyCodes` 作为修改 `v-on` 方法的方法。

```
1. <!-- 键码版本 -->
2. <input v-on:keyup.13="submit" />
3.
4. <!-- 别名版本 -->
5. <input v-on:keyup.enter="submit" />
```

此外，你可以通过全局 `config.keyCodes` 选项。

```
1. Vue.config.keyCodes = {
2.   f1: 112
3. }
```

```
1. <!-- 键码版本 -->
2. <input v-on:keyup.112="showHelpText" />
3.
4. <!-- 自定义别名版本 -->
5. <input v-on:keyup.f1="showHelpText" />
```

3.x 语法

从 `KeyboardEvent.keyCode` [has been deprecated](#) (opens new window) 开始，Vue 3 继续支持这一点就不再有意义了。因此，现在建议对任何要用作修饰符的键使用 kebab-cased（短横

线) 大小写名称。

1. `<!-- Vue 3 在 v-on 上使用 按键修饰符 -->`
2. `<input v-on:keyup.delete="confirmDelete" />`

因此，这意味着 `config.keyCodes` 现在也已弃用，不再受支持。

迁移策略

对于那些在代码库中使用 `keyCode` 的用户，我们建议将它们转换为它们的 kebab-cased (短横线) 命名对齐。

在 prop 的默认函数中访问 `this`

breaking

生成 prop 默认值的工厂函数不再能访问 `this`。

替代方案：

- 把组件接收到的原始 prop 作为参数传递给默认函数；
- [注入 API](#) 可以在默认函数中使用。

```
1. import { inject } from 'vue'
2.
3. export default {
4.   props: {
5.     theme: {
6.       default (props) {
7.         // `props` 是传递给组件的原始值。
8.         // 在任何类型/默认强制转换之前
9.         // 也可以使用 `inject` 来访问注入的 property
10.        return inject('theme', 'default-theme')
11.      }
12.    }
13.  }
14. }
```

渲染函数 API

breaking

概览

此更改不会影响 `<template>` 用户。

以下是更改的简要总结：

- `h` 现在全局导入，而不是作为参数传递给渲染函数
- 渲染函数参数更改为在有状态组件和函数组件之间更加一致
- `vnode` 现在有一个扁平的 `prop` 结构

更多信息，请继续阅读！

Render 函数参数

2.x 语法

在 2.x 中，`e.render` 函数将自动接收 `h` 函数（它是 `createElement` 的常规别名）作为参数：

```
1. // Vue 2 渲染函数示例
2. export default {
3.   render(h) {
4.     return h('div')
5.   }
6. }
```

3.x 语法

在 3.x 中，`h` 现在是全局导入的，而不是作为参数自动传递。

```
1. // Vue 3 渲染函数示例
2. import { h } from 'vue'
3.
4. export default {
5.   render() {
6.     return h('div')
```

```
7.   }  
8. }
```

渲染函数签名更改

2.x 语法

在 2.x 中，`render` 函数自动接收诸如 `h` 之类的参数。

```
1. // Vue 2 渲染函数示例  
2. export default {  
3.   render(h) {  
4.     return h('div')  
5.   }  
6. }
```

3.x 语法

在 3.x 中，由于 `render` 函数不再接收任何参数，它将主要用于 `setup()` 函数内部。这还有一个好处：可以访问作用域中声明的被动状态和函数，以及传递给 `setup()` 的参数。

```
1. import { h, reactive } from 'vue'  
2.  
3. export default {  
4.   setup(props, { slots, attrs, emit }) {  
5.     const state = reactive({  
6.       count: 0  
7.     })  
8.  
9.     function increment() {  
10.      state.count++  
11.    }  
12.  
13.    // 返回render函数  
14.    return () =>  
15.      h(  
16.        'div',  
17.        {  
18.          onClick: increment  
19.        },  
20.        state.count
```

```
21.     )  
22.   }  
23. }
```

有关 `setup()` 如何工作的详细信息，见 [Composition API Guide](#)。

VNode Props 格式化

2.x 语法

在 2.x 中，`domProps` 包含 VNode props 中的嵌套列表：

```
1. // 2.x  
2. {  
3.   class: ['button', 'is-outlined'],  
4.   style: { color: '#34495E' },  
5.   attrs: { id: 'submit' },  
6.   domProps: { innerHTML: '' },  
7.   on: { click: submitForm },  
8.   key: 'submit-button'  
9. }
```

3.x 语法

在 3.x 中，整个 VNode props 结构是扁平的，使用上面的例子，下面是它现在的样子

```
1. // 3.x 语法  
2. {  
3.   class: ['button', 'is-outlined'],  
4.   style: { color: '#34495E' },  
5.   id: 'submit',  
6.   innerHTML: '',  
7.   onClick: submitForm,  
8.   key: 'submit-button'  
9. }
```

迁移策略

工具库作者

全局导入 `h` 意味着任何包含 Vue 组件的库都将在某处包含 `import { h } from 'vue'`，因此，这会带来一些开销，因为它需要库作者在其构建设置中正确配置 Vue 的外部化：

- Vue 不应绑定到库中
- 对于模块构建，导入应该保持独立，由最终用户绑定器处理
- 对于 UMD/browser 版本，它应该首先尝试全局 `Vue.h`，然后回退以请求调用

下一步

见 [Render 函数指南](#) 更详细的文档！

Slot 统一

breaking

概览

此更改统一了 3.x 中的普通 slot 和作用域 slot。

以下是变化的变更总结：

- `this.$slots` 现在将 slots 作为函数公开
- **BREAKING:** 移除 `this.$scopedSlots`

更多信息，请继续阅读！

2.x 语法

当使用渲染函数时，即 `h`，2.x 用于在内容节点上定义 `slot` data property。

```
1. // 2.x 语法
2. h(LayoutComponent, [
3.   h('div', { slot: 'header' }, this.header),
4.   h('div', { slot: 'content' }, this.content)
5. ])
```

此外，在引用作用域 slot 时，可以使用以下方法引用它们：

```
1. // 2.x 语法
2. this.$scopedSlots.header
```

3.x 语法

在 3.x 中，插槽被定义为当前节点的子对象：

```
1. // 3.x Syntax
2. h(LayoutComponent, {}, {
3.   header: () => h('div', this.header),
4.   content: () => h('div', this.content)
5. })
```

当你需要以编程方式引用作用域 slot 时，它们现在被统一到 `$slots` 选项中。

1. `// 2.x 语法`
2. `this.$scopedSlots.header`
- 3.
4. `// 3.x 语法`
5. `this.$slots.header`

迁移策略

大部分更改已经在 2.6 中发布。因此，迁移可以一步到位：

1. 在 3.x 中，将所有 `this.$scopedSlots` 替换为 `this.$slots`。

过渡的 class 名更改

breaking

概览

过渡类名 `v-enter` 修改为 `v-enter-from`、过渡类名 `v-leave` 修改为 `v-leave-from`。

2.x 语法

在v2.1.8版本之前，为过渡指令提供了两个过渡类名对应初始和激活状态。

在 v2.1.8 版本中，引入 `v-enter-to` 来定义 enter 或 leave 变换之间的过渡动画插帧，为了向下兼容，并没有变动 `v-enter` 类名：

```
1. .v-enter,  
2. .v-leave-to {  
3.   opacity: 0;  
4. }  
5.  
6. .v-leave,  
7. .v-enter-to {  
8.   opacity: 1;  
9. }
```

这样做会带来很多困惑，类似 *enter* 和 *leave* 含义过于宽泛并且没有遵循类名钩子的命名约定。

3.x 语法

为了更加明确易读，我们现在将这些初始状态重命名为：

```
1. .v-enter-from,  
2. .v-leave-to {  
3.   opacity: 0;  
4. }  
5.  
6. .v-leave-from,  
7. .v-enter-to {
```

```
8.   opacity: 1;  
9. }
```

现在，这些状态之间的区别就清晰多了。

`<transition>` 组件相关属性名也发生了变化：

- `leave-class` 已经被重命名为 `leave-from-class`（在渲染函数或 JSX 中可以写为：`leaveFromClass`）
- `enter-class` 已经被重命名为 `enter-from-class`（在渲染函数或 JSX 中可以写为：`enterFromClass`）

迁移策略

1. 将 `.v-enter` 字符串实例替换为 `.v-enter-from`
2. 将 `.v-leave` 字符串实例替换为 `.v-leave-from`
3. 过渡组件相关属性名也需要进行字符串实例替换，规则如上所述。

v-model

breaking

概览

就变化内容而言，此部分属于高阶内容：

- **BREAKING:** 用于自定义组件时，`v-model` prop 和事件默认名称已更改：
 - prop: `value` -> `modelValue` ;
 - event: `input` -> `update:modelValue` ;
- **BREAKING:** `v-bind` 的 `.sync` 修饰符和组件的 `model` 选项已移除，可用 `v-model` 作为代替；
- **NEW:** 现在可以在同一个组件上使用多个 `v-model` 进行双向绑定；
- **NEW:** 现在可以自定义 `v-model` 修饰符。

更多信息，请见下文。

介绍

在 Vue 2.0 发布后，开发者使用 `v-model` 指令必须使用为 `value` 的 prop。如果开发者出于不同的目的需要使用其他的 prop，他们就不得不使用 `v-bind.sync`。此外，由于 `v-model` 和 `value` 之间的这种硬编码关系的原因，产生了如何处理原生元素和自定义元素的问题。

在 Vue 2.2 中，我们引入了 `model` 组件选项，允许组件自定义用于 `v-model` 的 prop 和事件。但是，这仍然只允许在组件上使用一个 `model`。

在 Vue 3 中，双向数据绑定的 API 已经标准化，减少了开发者在使用 `v-model` 指令时的混淆并且在使用 `v-model` 指令时可以更加灵活。

2.x 语法

在 2.x 中，在组件上使用 `v-model` 相当于绑定 `value` prop 和 `input` 事件：

```
1. <ChildComponent v-model="pageTitle" />
2.
3. <!-- 简写: -->
4.
5. <ChildComponent :value="pageTitle" @input="pageTitle = $event" />
```

如果要将属性或事件名称更改为其他名称，则需要要在 `ChildComponent` 组件中添加 `model` 选项：

```
1. <!-- ParentComponent.vue -->
2.
3. <ChildComponent v-model="pageTitle" />
```

```
1. // ChildComponent.vue
2.
3. export default {
4.   model: {
5.     prop: 'title',
6.     event: 'change'
7.   },
8.   props: {
9.     // 这将允许 `value` 属性用于其他用途
10.    value: String,
11.    // 使用 `title` 代替 `value` 作为 model 的 prop
12.    title: {
13.      type: String,
14.      default: 'Default title'
15.    }
16.  }
17. }
```

所以，在这个例子中 `v-model` 的简写如下：

```
1. <ChildComponent :title="pageTitle" @change="pageTitle = $event" />
```

使用 `v-bind.sync`

在某些情况下，我们可能需要对某一个 prop 进行“双向绑定”（除了前面用 `v-model` 绑定 prop 的情况）。为此，我们建议使用 `update:myPropName` 抛出事件。例如，对于在上一个示例中带有 `title` prop 的 `ChildComponent`，我们可以通过下面的方式将分配新 value 的意图传达给父级：

```
1. this.$emit('update:title', newValue)
```

如果需要的话，父级可以监听该事件并更新本地 data property。例如：

```
1. <ChildComponent :title="pageTitle" @update:title="pageTitle = $event" />
```

为了方便起见，我们可以使用 `.sync` 修饰符来缩写，如下所示：

```
1. <ChildComponent :title.sync="pageTitle" />
```

3.x 语法

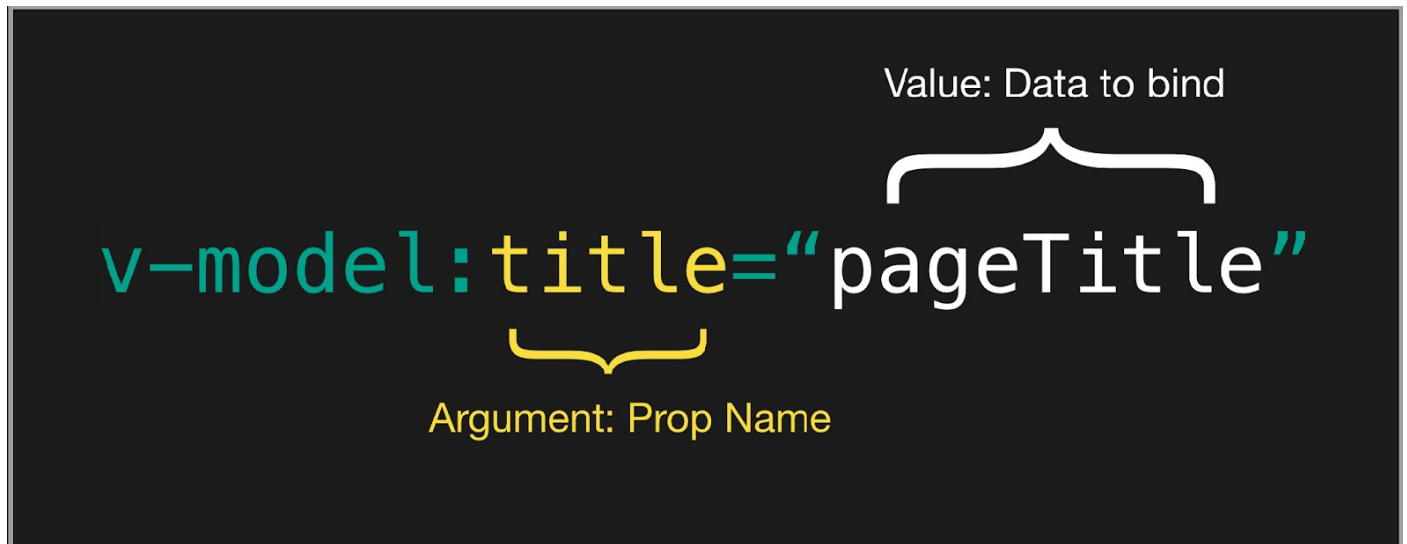
在 3.x 中，自定义组件上的 `v-model` 相当于传递了 `modelValue` prop 并接收抛出的 `update:modelValue` 事件：

```
1. <ChildComponent v-model="pageTitle" />
2.
3. <!-- 简写: -->
4.
5. <ChildComponent
6.   :modelValue="pageTitle"
7.   @update:modelValue="pageTitle = $event"
8. />
```

`v-model` 参数

若需要更改 `model` 名称，而不是更改组件内的 `model` 选项，那么现在我们可以将一个 *argument* 传递给 `model`：

```
1. <ChildComponent v-model:title="pageTitle" />
2.
3. <!-- 简写: -->
4.
5. <ChildComponent :title="pageTitle" @update:title="pageTitle = $event" />
```



这也可以作为 `.sync` 修饰符的替代，而且允许我们在自定义组件上使用多个 `v-model`。

```

1. <ChildComponent v-model:title="pageTitle" v-model:content="pageContent" />
2.
3. <!-- 简写: -->
4.
5. <ChildComponent
6.   :title="pageTitle"
7.   @update:title="pageTitle = $event"
8.   :content="pageContent"
9.   @update:content="pageContent = $event"
10. />

```

`v-model` 修饰符

除了像 `.trim` 这样的 2.x 硬编码的 `v-model` 修饰符外，现在 3.x 还支持自定义修饰符：

```
1. <ChildComponent v-model.capitalize="pageTitle" />
```

我们可以在 [Custom Events](#) 部分中了解有关自定义 `v-model` 修饰符的更多信息。

迁移策略

我们推荐：

- 检查你的代码库中所有使用 `.sync` 的部分并将其替换为 `v-model`：

```

1. <ChildComponent :title.sync="pageTitle" />
2.

```



```
3. <!-- 替换为 -->
4.
5. <ChildComponent v-model:title="pageTitle" />
```

- 对于所有不带参数的 `v-model`，请确保分别将 `prop` 和 `event` 命名更改为 `modelValue` 和 `update:modelValue`

```
1. <ChildComponent v-model="pageTitle" />
```

```
1. // ChildComponent.vue
2.
3. export default {
4.   props: {
5.     modelValue: String // 以前是 `value: String`
6.   },
7.   methods: {
8.     changePageTitle(title) {
9.       this.$emit('update:modelValue', title) // 以前是 `this.$emit('input',
10.      title)`
11.     }
12.   }
13. }
```

下一步

更多新的 `v-model` 语法相关信息，请参考：

- [在组件中使用 `v-model`](#)
- [v-model 参数](#)
- [处理 `v-model` 修饰符](#)

v-if 与 v-for 的优先级对比

breaking

概览

- **BREAKING:** 两者作用于同一个元素上时，`v-if` 会拥有比 `v-for` 更高的优先级。

介绍

Vue.js 中使用最多的两个指令就是 `v-if` 和 `v-for`，因此开发者们可能会想要同时使用它们。虽然不建议这样做，但有时确实是必须的，于是我们想提供有关其工作方式的指南。

2.x 语法

2.x 版本中在一个元素上同时使用 `v-if` 和 `v-for` 时，`v-for` 会优先作用。

3.x 语法

3.x 版本中 `v-if` 总是优先于 `v-for` 生效。

迁移策略

由于语法上存在歧义，建议避免在同一元素上同时使用两者。

比起在模板层面管理相关逻辑，更好的办法是通过创建计算属性筛选出列表，并以此创建可见元素。

v-bind 合并行为

breaking

概览

- 不兼容: v-bind 的绑定顺序会影响渲染结果。

介绍

在元素上动态绑定 attribute 时，常见的场景是在一个元素中同时使用 `v-bind="object"` 语法和单独的 property。然而，这就引出了关于合并的优先级的问題。

2.x 语法

在 2.x，如果一个元素同时定义了 `v-bind="object"` 和一个相同的单独的 property，那么这个单独的 property 总是会覆盖 `object` 中的绑定。

```
1. <!-- template -->
2. <div id="red" v-bind="{ id: 'blue' }"></div>
3. <!-- result -->
4. <div id="red"></div>
```

3.x 语法

在 3.x，如果一个元素同时定义了 `v-bind="object"` 和一个相同的单独的 property，那么声明绑定的顺序决定了它们如何合并。换句话说，相对于假设开发者总是希望单独的 property 覆盖 `object` 中定义的内容，现在开发者对自己所希望的合并行为有了更好的控制。

```
1. <!-- template -->
2. <div id="red" v-bind="{ id: 'blue' }"></div>
3. <!-- result -->
4. <div id="blue"></div>
5.
6. <!-- template -->
7. <div v-bind="{ id: 'blue' }" id="red"></div>
8. <!-- result -->
9. <div id="red"></div>
```

迁移策略

如果你依赖 `v-bind` 的覆盖功能，目前的建议是确保在单独的 `property` 之前定义 `v-bind` `attribute`。

- [Vue 文档编写指南](#)
- [文档风格指南](#)
- [翻译](#)

Vue 文档编写指南

译者：本章节大部分内容是针对母语是英文的读者，中文用户可略读，除非你想以英文文档编写者的身份参与 Vue docs 的编写，

编写文档是一种换位思考的练习。我们并不是在描述客观现实——源代码已经做到了。我们的工作帮助塑造用户与 Vue 生态系统之间的关系。这份不断发展的指南提供了一些规则和建议，说明如何在 Vue 生态系统中始终如一地做到这一点。

原则

- 除非有充分的文档证明，否则功能不存在。
- 尊重用户的认知能力（即脑力）。当用户开始阅读时，他们从一定量的有限脑力开始，而当他们用完时，他们停止学习。
 - 复杂的句子、一次必须学习一个以上的概念，以及与用户的工作没有直接关系的抽象例子，认知能力消耗得更快。
 - 当我们帮助他们持续感到聪明、强大和好奇时，他们的认知能力会慢慢消耗殆尽。把事情分解成可消化的部分并注意文档的流动可以帮助它们保持这种状态。
- 总是试着从用户的角度看问题。当我们彻底理解某件事情时，它就变得显而易见了。这就是所谓的知识诅咒。为了编写好的文档，记住在学习这个概念时首先需要知道什么。你需要学什么行话？你误解了什么？什么花了很长时间才真正掌握？好的文档可以满足用户的需求。这可能有助于练习向人们解释这个概念
- 首先描述问题，然后描述解决方案。在展示功能如何工作之前，解释其存在的原因非常重要。否则，用户将无法知道这些信息对他们是否重要（这是他们遇到的问题吗？）或与之前的知识/经验相联系。
- 在写作时，不要害怕问问题，尤其是如果你害怕他们“蠢”的话。脆弱是很难的，但这是我们更充分地理解我们需要解释的唯一途径。
- 参与特性讨论。最好的 API 来自于文档驱动的开发，我们在开发中构建易于解释的特性，而不是试图在以后解释它们。提前提出问题（尤其是“愚蠢的”问题）通常有助于揭示困惑、不一致和有问题的行为，然后才需要进行破坏性的更改来修复它们。

组织

- 安装/集成：提供有关如何将软件集成到尽可能多的不同项目中的全面概述。

- 介绍/起步：
 - 提供一个不到 10 分钟的项目解决的问题及其存在原因的概述。
 - 提供一个不到 30 分钟的项目解决的问题和如何解决的概述，包括何时和为什么使用项目以及一些简单的代码示例。最后，链接到安装页面和要点指南的开头。
- 指南：让用户感到聪明、强大、好奇，然后保持这种状态，让用户保持不断学习的动力和认知能力。指南页是按顺序阅读的，因此通常应该从最高到最低的功率/工作比排序。
 - 要点：阅读要领的时间不应超过 5 个小时，但越短越好。它的目标是提供 20%的知识来帮助用户处理 80%的用例。Essentials 可以链接到更高阶的指南和 API，不过，在大多数情况下，你应该避免此类链接。当它们被提供时，你还需要提供一个上下文，以使用户知道他们是否应该在第一次阅读时遵循这个链接。否则，许多用户最终会耗尽他们的认知能力，跳转链接，试图在继续之前全面了解一个功能的各个方面，结果是，永远无法完成第一次通读的要领。记住，通顺的阅读比彻底的阅读更重要。我们想给人们提供他们需要的信息，以避免令人沮丧的经历，但他们总是可以回来继续阅读，或者在谷歌遇到一个不太常见的问题。
 - 高阶：虽然要点帮助人们处理大约 80%的用例，但后续的指南帮助用户了解 95%的用例，以及关于非基本特性（例如转换、动画）、更复杂的便利特性（例如 mixin、自定义指令）和开发人员体验改进（例如 JSX、插件）的更详细信息。最后 5%的用例是更基础的、更复杂的和/或更容易被滥用的，将留给烹饪书和 API 参考，它们可以从这些高阶指南链接到。
- 引用/API：提供功能的完整列表，包括类型信息，每个要解决的问题的描述，选项的每种组合的示例以及指向指南，食谱的食谱以及提供更多详细信息的其他内部资源的链接。与其他页面不同，此页面无意自上而下阅读，因此可以提供大量详细信息。这些参考资料还必须比指南更容易浏览，因此格式应比指南的讲故事格式更接近字典条目。
- 迁移：
 - 版本：当进行了重要的更改时，包含一个完整的更改列表是很有用的，包括对为什么进行更改以及如何迁移其项目的详细解释。
 - 从其他项目：这个软件与同类软件相比如何？这对于帮助用户了解我们可能为他们解决或创造的其他问题，以及他们可以在多大程度上转移他们已经拥有的知识，这一点很重要。
- 风格指南：开发中必然有一些关键部分需要决策，但它们不是 API 的核心。风格指南提供了受过教育的、有主见的建议，以帮助指导这些决策。他们不应该盲目遵循，但可以帮助团队节省时间，在较小的细节上保持一致。
- **Cookbook**：Cookbook 中的秘诀是基于对 Vue 及其生态系统的熟悉程度而编写的。每一个文档都是一个高度结构化的文档，它详细介绍了 Vue 开发人员可能遇到的一些常见实现细节。

写作 & 语法

风格

- 标题应该描述问题，不是解决方案。例如，一个不太有效的标题可能是“使用 prop”，因为它描述了一个解决方案。一个更好的标题可能是“通过 Props 将数据传递给子组件”，因为它提供了 Props 解决问题的上下文。用户不会真正开始注意某个功能的解释，直到他们知道为什么/何时使用它。
- 当你假设知识时，就要声明它，在开始时，链接到参考资料，以获得你期望的不太常见的知识。
- 尽可能一次只引入一个新概念（包括文本和代码示例），即使当你介绍不止一个的时候很多人都能理解，也有很多人会迷失方向，即使那些没有迷失方向的人也会耗尽更多的认知能力。
- 尽可能避免使用特殊的内容块来获取提示和注意事项，一般来说，最好将这些内容更自然地融合到主要内容中，例如，通过构建示例来演示边缘案例。
- 每页不要超过两个相互交织的提示和注意事项，如果你发现一个页面需要两个以上的提示，请考虑添加一个警告部分来解决这些问题。本指南的目的是通读，提示和注意事项可能会让试图理解基本概念的人不知所措或分心。
- 避免诉诸权威（例如，“你应该做 X，因为这是一个最佳实践”或“X 是最好的，因为它能让你完全分离关注点”）。相反，用例子来演示由模式引起和/或解决的具体人类问题。
- 当决定先教什么时，想想哪些知识能提供最好的动力/努力比。这意味着教任何能帮助用户解决最大痛苦或最大数量问题的东西，而学习的努力相对较少。这有助于学习者感到聪明、强大和好奇，因此他们的认知能力会慢慢流失。
- 除非上下文采用字符串模板或构建系统，否则只能编写在软件的任何环境中工作的代码（例如 **Vue**、**Vuex** 等）
- 显示，不要说例如，“要在页面上使用 Vue，可以将其添加到 HTML 中”（然后显示脚本标记），而不是“要在页面上使用 Vue，可以添加一个具有 src 属性的脚本元素，该属性的值应为指向 Vue 编译源的链接”。
- 几乎总是避免幽默（对于英文文档），尤其是讽刺和通俗文化的引用，因为它在不同文化之间的翻译并不好。
- 永远不要假设比你必须的更高阶的上下文。
- 在大多数情况下，比起在多个部分中重复相同的内容，更喜欢在文档的各个部分之间建立链接。在内容上有些重复是不可避免的，甚至是学习的必要条件。然而，过多的重复也会使文档更难维护，因为 API 的更改将需要在许多地方进行更改，而且很容易遗漏某些内容。这是一个很难达

到的平衡。

- 具体的比一般的好例如，一个 `<BlogPost>` 组件例子比 `<ComponentA>` 更好。
- 相对胜于晦涩。例如，一个 `<BlogPost>` 组件例子比 `<CurrencyExchangeSettings>` 更好。
- 保持情感相关。与人们有经验并关心的事物相关的解释和示例将永远更加有效。
- 始终喜欢使用简单，简单的语言，而不是复杂或专业的语言。例如：
 - “你可以将 Vue 与脚本元素一起使用”，而不是“为了启动 Vue 的使用，一种可能的选择是通过脚本 HTML 元素实际注入它”
 - “返回函数的函数”而不是“高阶函数”
- 避免使用毫无意义的语言。如“简单”、“公正”、“明显”等，请参阅[教育写作中应避免的词语](#) (opens new window)。

语法

- 避免缩写在编写代码和示例代码中（例如，`attribute` 优于 `attr`，`message` 优于 `msg`），除非你在 API 中明确引用了缩写（例如 `$attrs`）。标准键盘上包含的缩写符号（例如，`@`，`#`，`&`）可以。
- 当引用直接下面的示例时，请使用冒号（`:`）结束句子，而不是句点（`.`）
- 使用牛津逗号（`;` 例如：“a, b, and c”替换“a, b and c”）。！[为什么牛津逗号很重要](#) (opens new window)
 - 来源：[The Serial \(Oxford\) Comma: When and Why To Use It](#) (opens new window)
- 引用项目名称时，请使用项目引用自身的名称。例如，“webpack”和“npm”都应使用小写字母，因为这是它们的文档引用它们的方式。
- 使用标题大小写作为标题 - 至少到目前为止，因为这是我们在其余文档中使用的。有研究表明，句子大小写（仅标题的第一个单词以大写字母开头）实际上在可读性上是优越的，并且还减少了文档作者的认知开销，因为他们不必记住是否要大写“and”，“with”和“about”。
- 请勿使用表情符号（讨论中除外）。Emoji 既可爱又友好，但是它们可能会使文档分散注意力，有些表情符号甚至会在不同文化中传达不同的含义。

迭代 & 沟通

- 卓越源于迭代初稿总是很糟糕，但是编写初稿是该过程的重要组成部分。要避免进度缓慢，很难-不好-> OK->好->好->鼓舞人心->超越。
- 在发布之前，请仅等到某事为“好”为止。社区将帮助你将其推向更深的链。
- 收到反馈时，尽量不要防御我们的写作对我们来说可能是非常私人的，但是如果我们对帮助我们做得更好的人感到不满，他们要么停止提供反馈，要么开始限制他们提供的反馈种类。
- 在向他人展示之前，请先阅读自己的作品。如果你显示某人的拼写/语法错误很多，你将获得有关拼写语法/错误的反馈，而不是获得有关写作是否达到目标的更有价值的注释。
- 当你要求人们提供反馈时，请告诉审阅者以下内容：
 - 你正在尝试做
 - 你的恐惧是
 - 你想要达到的平衡
- 当有人报告问题时，几乎总是有问题，即使他们提出的解决方案不太正确。不断询问后续问题以了解更多信息
- 人们在提交/查看内容时需要放心地提问。这是你可以执行的操作：
 - 即使别人感到脾气暴躁，也要感谢他们的贡献/评价。比如：
 - “Great question!”
 - “感谢你抽出宝贵的时间来解释。”
 - “这实际上是故意的，但感谢你抽出宝贵的时间来贡献自己的力量。 😊”
 - 听别人说什么，如果不确定自己是否正确理解，请照搬。这可以帮助验证人们的感受和经历，同时还可以了解你是否正确理解了他们。
 - 使用大量积极和善解人意的表情符号。显得有些奇怪总比刻薄或急躁好。
 - 请传达规则/边界。如果某人的举止有辱人格/不当行为，请仅以仁慈和成熟来回应，但也要明确表示，这种行为是不可接受的，如果他们继续表现不佳，将会发生什么（根据行为准则）。

提示、标注、警告和行高亮

我们有一些专用的样式来表示需要以特定方式突出显示的内容。这些被捕获为[在这个页面](#)[↗] (opens new window) 请谨慎使用。

滥用这些样式是有一定诱惑力的，因为你可以简单地在标注中添加更改。但是，这会破坏用户的阅读流程，因此，只能在特殊情况下使用。在可能的情况下，我们应该尝试在页面内创建一个叙述和流程，以尊重读者的认知负荷。

在任何情况下都不应该相邻使用两个警告，这表明我们无法很好地解释上下文。

贡献

我们欣赏小型、集中的 PR。如果你想进行非常大的更改，请在发起请求之前与团队成员沟通。这是一份[详细说明为什么这一点如此重要的书面材料](#)[↗] (opens new window) 让我们在这个团队里工作得很

好。请理解，尽管我们总是很感激你的贡献，但最终我们必须优先考虑哪些对整个项目最有效。

资源

软件

- [Grammarly](#) [↗] (opens new window): 用于检查拼写和语法的桌面应用程序和浏览器扩展 (尽管语法检查不能捕获所有内容, 偶尔会显示假阳性)。
- [Code Spell Checker](#) [↗] (opens new window): 一个 VS Code 的扩展, 帮助你在降价和代码示例中检查拼写。

书籍

- [On Writing Well](#) [↗] (opens new window) (参见 [popular quotes](#) [↗] (opens new window))
- [Bird by Bird](#) [↗] (opens new window) (参见 [popular quotes](#) [↗] (opens new window))
- [Cognitive Load Theory](#) [↗] (opens new window)

文档风格指南

本指南将概述可用于创建文档的不同设计元素。

警告

VuePress 提供了一个自定义容器插件来创建警稿框。有四种类型：

- **Info:** 提供中立的信息
- **Tip:** 提供积极和鼓励的信息
- **Warning:** 提供用户应该知道的信息，因为存在低到中等
- **Danger:** 供对用户具有高风险的负面信息

Markdown 范例

```
1. ::: info
2. You can find more information at this site.
3. :::
4.
5. ::: tip
6. This is a great tip to remember!
7. :::
8.
9. ::: warning
10. This is something to be cautious of.
11. :::
12.
13. ::: danger DANGER
14. This is something we do not recommend. Use at your own risk.
15. :::
```

渲染 Markdown

INFO

You can find more information at [this](#) site.

TIP

This is a great tip to remember !

WARNING

This is something to be cautious of.

DANGER

This is something we do not recommend. Use at your own risk.

代码块

VuePress 使用 Prism 提供语言语法高亮显示，方法是将语言附加到代码块的起始反撇号：

Markdown 示例

```
1. ```js
2. export default {
3.   name: 'MyComponent'
4. }
5. ```
```

渲染输出

```
1. export default {
2.   name: 'MyComponent'
3. }
```

行高亮

向代码块添加行高亮显示，需要在大括号中附加行号。

单行

Markdown 示例

```
1. ```js{2}
2. export default {
3.   name: 'MyComponent',
4.   props: {
5.     type: String,
6.     item: Object
7.   }
8. }
9. ```
```

渲染 Markdown

```

1. export default {
2.   name: 'MyComponent',
3.   props: {
4.     type: String,
5.     item: Object
6.   }
7. }

```

行组

```

1. ```js{4-5}
2. export default {
3.   name: 'MyComponent',
4.   props: {
5.     type: String,
6.     item: Object
7.   }
8. }
9. ```

```

```

1. export default {
2.   name: 'MyComponent',
3.   props: {
4.     type: String,
5.     item: Object
6.   }
7. }

```

多个段落

```

1. ```js{2,4-5}
2. export default {
3.   name: 'MyComponent',
4.   props: {
5.     type: String,
6.     item: Object
7.   }
8. }
9. ```

```

```
1. export default {  
2.   name: 'MyComponent',  
3.   props: {  
4.     type: String,  
5.     item: Object  
6.   }  
7. }
```

翻译

Vue 已经遍布全球，核心团队至少在 6 个不同的时区。[论坛](#) (opens new window) 包括 7 种语言和计数，我们的许多文档[积极维护翻译](#) (opens new window)。我们为 Vue 的国际影响力感到骄傲，但我们可以做得更好。

我们可以开始翻译 Vue 3 文档了吗？

目前，VUE3 文档仍处于测试阶段，随时可能更改。因此，我们会谨慎对待任何重要的工作，因为我们仍在收集反馈，并根据需要重新编写。当文档处于发布候选阶段时，我们将确保发布公告，以便你可以开始使用！

我如何参与翻译？

开始的最好方法是检查[此处固定 issues Vuejs.org](#) (opens new window) 其中包含了对社区内各种倡议的积极讨论。