# 第1章 Introduction

## 1.1 What is Vue?

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

Here is a minimal example:

import { createApp } from 'vue'

createApp({

data() {

return {

count: 0

}

}

}).mount('#app')

<div id="app">

<button @click="count++">

Count is: {{ count }}

</button>

</div>

The above example demonstrates the two core features of Vue:

Declarative Rendering: Vue extends standard HTML with a template syntax that allows us to declaratively describe HTML output based on JavaScript state.

Reactivity: Vue automatically tracks JavaScript state changes and efficiently updates the DOM when changes happen.

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

Prerequisites

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

## 1.2 The Progressive Framework

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

Enhancing static HTML without a build step

Embedding as Web Components on any page

Single-Page Application (SPA)

Fullstack / Server-Side Rendering (SSR)

Jamstack / Static Site Generation (SSG)

Targeting desktop, mobile, WebGL, and even the terminal

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

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

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

## 1.3 Single-File Components

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

<script>

export default {

data() {

return {

count: 0

}

}

}

</script>

<template>

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

</template>

<style scoped>

button {

font-weight: bold;

}

</style>

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

## 1.4 API Styles

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

### 1.4.1 Options API

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

<script>

export default {

// Properties returned from data() become reactive state

// and will be exposed on `this`.

data() {

return {

count: 0

}

},

// Methods are functions that mutate state and trigger updates.

// They can be bound as event listeners in templates.

methods: {

increment() {

this.count++

}

},

// Lifecycle hooks are called at different stages

// of a component's lifecycle.

// This function will be called when the component is mounted.

mounted() {

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

}

}

</script>

<template>

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

</template>

### 1.4.2 Composition API

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

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

<script setup>

import { ref, onMounted } from 'vue'

// reactive state

const count = ref(0)

// functions that mutate state and trigger updates

function increment() {

count.value++

}

// lifecycle hooks

onMounted(() => {

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

})

</script>

<template>

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

</template>

### 1.4.3 Which to Choose?

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

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

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

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

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

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

For production use:

Go with Options API if you are not using build tools, or plan to use Vue primarily in low-complexity scenarios, e.g. progressive enhancement.

Go with Composition API + Single-File Components if you plan to build full applications with Vue.

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

Still Got Questions?

# 第2章 Quick Start

Depending on your use case and preference, you can use Vue with or without a build step.

## 2.1 With Build Tools

A build setup allows us to use Vue Single-File Components (SFCs). The official Vue build setup is based on Vite, a frontend build tool that is modern, lightweight and extremely fast.

### 2.1.1 Online

You can try Vue with SFCs online on StackBlitz. StackBlitz runs the Vite-based build setup directly in the browser, so it is almost identical to the local setup but doesn't require installing anything on your machine.

### 2.1.2 Local

Pre-requisites

Familiarity with the command line

Install Node.js version 15.0 or higher

To create a build-tool-enabled Vue project on your machine, run the following command in your command line (without the > sign):

> npm init vue@latest

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

✔ Project name: … <your-project-name>

✔ Add TypeScript? … No / Yes

✔ Add JSX Support? … No / Yes

✔ Add Vue Router for Single Page Application development? … No / Yes

✔ Add Pinia for state management? … No / Yes

✔ Add Vitest for Unit testing? … No / Yes

✔ Add Cypress for both Unit and End-to-End testing? … No / Yes

✔ Add ESLint for code quality? … No / Yes

✔ Add Prettier for code formatting? … No / Yes

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

Done.

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

> cd <your-project-name>

> npm install

> npm run dev

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

The recommended IDE setup is Visual Studio Code + Volar extension. If you use other editors, check out the IDE support section.

More tooling details, including integration with backend frameworks, are discussed in the Tooling Guide.

To learn more about the underlying build tool Vite, check out the Vite docs.

If you chose to use TypeScript, check out the TypeScript Usage Guide.

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

> npm run build

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

## 2.2 Without Build Tools

To get started with Vue without a build step, simply copy the following code into an HTML file and open it in your browser:

<script src="https://unpkg.com/vue@3"></script>

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

<script>

const { createApp } = Vue

createApp({

data() {

return {

message: 'Hello Vue!'

}

}

}).mount('#app')

</script>

The above example uses the global build of Vue where all APIs are exposed under the global Vue variable. For example, to also use the ref API, you can do:

const { createApp, ref } = Vue

While the global build works, we will be primarily using ES modules syntax throughout the rest of the documentation for consistency. In order to use Vue over native ES modules, use the following HTML instead:

<script type="importmap">

{

"imports": {

"vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"

}

}

</script>

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

<script type="module">

import { createApp } from 'vue'

createApp({

data() {

return {

message: 'Hello Vue!'

}

}

}).mount('#app')

</script>

Notice how we can import directly from 'vue' in our code - this is made possible by the <script type="importmap"> block, leveraging a native browser feature called Import Maps.

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

Import Maps Browser Support

Import maps are supported by default in Chromium-based browsers, so we recommend using Chrome or Edge during the learning process.

If using Firefox, it is only supported in version 102+ and currently needs to be enabled via the dom.importMaps.enabled option in about:config.

If your preferred browser does not support import maps yet, you can polyfill it with es-module-shims.

Not for production

The import-maps-based setup is meant for learning only - if you intend to use Vue without build tools in production, make sure to check out the Production Deployment Guide.

### 2.2.1 Serving over HTTP

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

<!-- index.html -->

<script type="module">

import { createApp } from 'vue'

import MyComponent from './my-component.js'

createApp(MyComponent).mount('#app')

</script>

// my-component.js

export default {

data() {

return { count: 0 }

},

template: `<div>count is {{ count }}</div>`

}

In order for this to work, you need to serve your HTML over the http:// protocol instead of file:// protocol. To start a local HTTP server, first install Node.js, and then run npx serve from the command line in the same directory where your HTML file is. You can also use any other HTTP server that can serve static files with correct MIME types.

You may have noticed that the imported component's template is inlined as a JavaScript string. If you are using VSCode, you can install the es6-string-html extension and prefix the strings with a /\*html\*/ comment to get syntax highlighting for them.As we dive deeper into the guide, we may need to split our code into separate JavaScript files so that they are easier to manage. For example:

<!-- index.html -->

<script type="module">

import { createApp } from 'vue'

import MyComponent from './my-component.js'

createApp(MyComponent).mount('#app')

</script>

// my-component.js

export default {

data() {

return { count: 0 }

},

template: `<div>count is {{ count }}</div>`

}

In order for this to work, you need to serve your HTML over the http:// protocol instead of file:// protocol. To start a local HTTP server, first install Node.js, and then run npx serve from the command line in the same directory where your HTML file is. You can also use any other HTTP server that can serve static files with correct MIME types.

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

# 第3章 Creating a Vue Application

## 3.1 The application instance

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

import { createApp } from 'vue'

const app = createApp({

/\* root component options \*/

})

## 3.2 The Root Component

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

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

import { createApp } from 'vue'

// import the root component App from a single-file component.

import App from './App.vue'

const app = createApp(App)

While many examples in this guide only need a single component, most real applications are organized into a tree of nested, reusable components. For example, a Todo application's component tree might look like this:

App (root component)

├─ TodoList

│ └─ TodoItem

│ ├─ TodoDeleteButton

│ └─ TodoEditButton

└─ TodoFooter

├─ TodoClearButton

└─ TodoStatistics

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

## 3.3 Mounting the App

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

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

app.mount('#app')

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

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

### 3.3.1 In-DOM Root Component Template

When using Vue without a build step, we can write our root component's template directly inside the mount container:

<div id="app">

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

</div>

import { createApp } from 'vue'

const app = createApp({

data() {

return {

count: 0

}

}

})

app.mount('#app')

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

## 3.4 App Configurations

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

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

/\* handle error \*/

}

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

app.component('TodoDeleteButton', TodoDeleteButton)

This makes the TodoDeleteButton available for use anywhere in our app. We will discuss registration for components and other types of assets in later sections of the guide. You can also browse the full list of application instance APIs in its API reference.

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

## 3.5 Multiple application instances

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

const app1 = createApp({

/\* ... \*/

})

app1.mount('#container-1')

const app2 = createApp({

/\* ... \*/

})

app2.mount('#container-2')

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

# 第4章 Template Syntax

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

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

If you are familiar with Virtual DOM concepts and prefer the raw power of JavaScript, you can also directly write render functions instead of templates, with optional JSX support. However, do note that they do not enjoy the same level of compile-time optimizations as templates.

## 4.1 Text Interpolation

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

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

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

## 4.2 Raw HTML

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

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

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

Using text interpolation: <span style="color: red">This should be red.</span>

Using v-html directive: This should be red.

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

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

Security Warning

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

## 4.3 Attribute Bindings

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

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

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

## 4.4 Shorthand

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

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

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

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

## 4.5 Boolean Attributes

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

v-bind works a bit differently in this case:

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

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

## 4.6 Dynamically Binding Multiple Attributes

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

data() {

return {

objectOfAttrs: {

id: 'container',

class: 'wrapper'

}

}

}

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

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

## 4.7 Using JavaScript Expressions

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

{{ number + 1 }}

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

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

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

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

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

1.Inside text interpolations (mustaches)

2.In the attribute value of any Vue directives (special attributes that start with v-)

### 4.7.1 Expressions Only

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

Therefore, the following will NOT work:

<!-- this is a statement, not an expression: -->

{{ var a = 1 }}

<!-- flow control won't work either, use ternary expressions -->

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

### 4.7.2 Calling Functions

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

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

{{ formatDate(date) }}

</span>

TIP

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

### 4.7.3 Restricted Globals Access

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

Globals not explicitly included in the list, for example user-attached properties on window, will not be accessible in template expressions. You can, however, explicitly define additional globals for all Vue expressions by adding them to app.config.globalProperties.

## 4.8 Directives

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

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

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

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

### 4.8.1 Arguments

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

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

<!-- shorthand -->

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

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

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

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

<!-- shorthand -->

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

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

### 4.8.2 Dynamic Arguments

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

<!--

Note that there are some constraints to the argument expression,

as explained in the "Dynamic Argument Value Constraints" and "Dynamic Argument Syntax Constraints" sections below.

-->

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

<!-- shorthand -->

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

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

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

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

<!-- shorthand -->

<a @[eventName]="doSomething">

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

### 4.8.3 Dynamic Argument Value Constraints

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

Dynamic Argument Syntax Constraints#

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

<!-- This will trigger a compiler warning. -->

<a :['foo' + bar]="value"> ... </a>

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

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

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

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

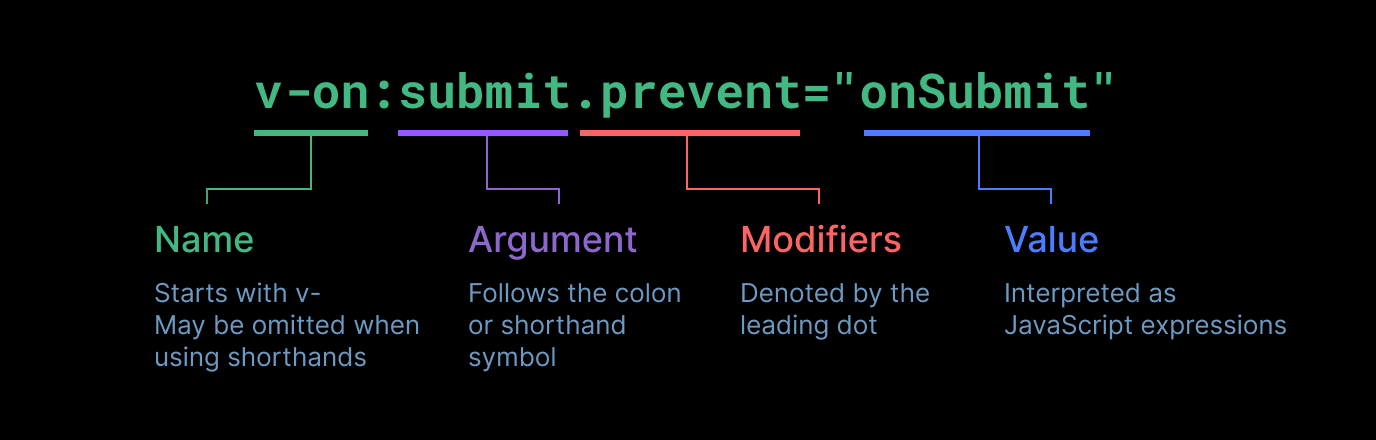
### 4.8.4 Modifiers

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

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

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

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



# 第5章 Reactivity Fundamentals

API Preference

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

## 5.1 Declaring Reactive State

With Options API, we use the data option to declare reactive state of a component. The option value should be a function that returns an object. Vue will call the function when creating a new component instance, and wrap the returned object in its reactivity system. Any top-level properties of this object are proxied on the component instance (this in methods and lifecycle hooks):

export default {

data() {

return {

count: 1

}

},

// `mounted` is a lifecycle hook which we will explain later

mounted() {

// `this` refers to the component instance.

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

// data can be mutated as well

this.count = 2

}

}

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 this without including it in data. However, properties added this way will not be able to trigger reactive updates.

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.

## 5.2 Reactive Proxy vs. Original

In Vue 3, data is made reactive by leveraging JavaScript Proxies. Users coming from Vue 2 should be aware of the following edge case:

export default {

data() {

return {

someObject: {}

}

},

mounted() {

const newObject = {}

this.someObject = newObject

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

}

}

When you access this.someObject after assigning it, the value is a reactive proxy of the original newObject. Unlike in Vue 2, the original newObject is left intact and will not be made reactive: make sure to always access reactive state as a property of this.

## 5.3 Declaring Methods

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

export default {

data() {

return {

count: 0

}

},

methods: {

increment() {

this.count++

}

},

mounted() {

// methods can be called in lifecycle hooks, or other methods!

this.increment()

}

}

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:

export default {

methods: {

increment: () => {

// BAD: no `this` access here!

}

}

}

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:

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

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

### 5.3.1 DOM Update Timing

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

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

import { nextTick } from 'vue'

export default {

methods: {

increment() {

this.count++

nextTick(() => {

// access updated DOM

})

}

}

}

### 5.3.2 Deep Reactivity

In Vue, state is deeply reactive by default. This means you can expect changes to be detected even when you mutate nested objects or arrays:

export default {

data() {

return {

obj: {

nested: { count: 0 },

arr: ['foo', 'bar']

}

}

},

methods: {

mutateDeeply() {

// these will work as expected.

this.obj.nested.count++

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

}

}

}

It is also possible to explicitly create shallow reactive objects where the reactivity is only tracked at the root-level, however they are typically only needed in advanced use cases.

### 5.3.3 Stateful Methods

In some cases, we may need to dynamically create a method function, for example creating a debounced event handler:

import { debounce } from 'lodash-es'

export default {

methods: {

// Debouncing with Lodash

click: debounce(function () {

// ... respond to click ...

}, 500)

}

}

However, this approach is problematic for components that are reused because a debounced function is stateful: it maintains some internal state on the elapsed time. If multiple component instances share the same debounced function, they will interfere with one another.

To keep each component instance's debounced function independent of the others, we can create the debounced version in the created lifecycle hook:

export default {

created() {

// each instance now has its own copy of debounced handler

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

},

unmounted() {

// also a good idea to cancel the timer

// when the component is removed

this.debouncedClick.cancel()

},

methods: {

click() {

// ... respond to click ...

}

}

}

# 第6章 Computed Properties

## 6.1 Basic Example

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

export default {

data() {

return {

author: {

name: 'John Doe',

books: [

'Vue 2 - Advanced Guide',

'Vue 3 - Basic Guide',

'Vue 4 - The Mystery'

]

}

}

}

}

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

<p>Has published books:</p>

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

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

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

export default {

data() {

return {

author: {

name: 'John Doe',

books: [

'Vue 2 - Advanced Guide',

'Vue 3 - Basic Guide',

'Vue 4 - The Mystery'

]

}

}

},

computed: {

// a computed getter

publishedBooksMessage() {

// `this` points to the component instance

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

}

}

}

<p>Has published books:</p>

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

Here we have declared a computed property publishedBooksMessage.

Try to change the value of the books array in the application data and you will see how publishedBooksMessage is changing accordingly.

You can data-bind to computed properties in templates just like a normal property. Vue is aware that this.publishedBooksMessage depends on this.author.books, so it will update any bindings that depend on this.publishedBooksMessage when this.author.books changes.

## 6.2 Computed Caching vs Methods

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

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

// in component

methods: {

calculateBooksMessage() {

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

}

}

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

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

computed: {

now() {

return Date.now()

}

}

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

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

## 6.3 Writable Computed

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

export default {

data() {

return {

firstName: 'John',

lastName: 'Doe'

}

},

computed: {

fullName: {

// getter

get() {

return this.firstName + ' ' + this.lastName

},

// setter

set(newValue) {

// Note: we are using destructuring assignment syntax here.

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

}

}

}

}

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

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

export default {

data() {

return {

firstName: 'John',

lastName: 'Doe'

}

},

computed: {

fullName: {

// getter

get() {

return this.firstName + ' ' + this.lastName

},

// setter

set(newValue) {

// Note: we are using destructuring assignment syntax here.

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

}

}

}

}

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

## 6.4 Best Practices

### 6.4.1 Getters should be side-effect free

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

### 6.4.2 Avoid mutating computed value

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

# 第7章 Class and Style Bindings

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

## 7.1 Binding HTML Classes

### 7.1.1 Binding to Objects

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

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

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

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

data() {

return {

isActive: true,

hasError: false

}

}

And the following template:

<div

class="static"

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

></div>

It will render:

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

When isActive or hasError changes, the class list will be updated accordingly. For example, if hasError becomes true, the class list will become "static active text-danger".

The bound object doesn't have to be inline:

data() {

return {

classObject: {

active: true,

'text-danger': false

}

}

}

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

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

data() {

return {

isActive: true,

error: null

}

},

computed: {

classObject() {

return {

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

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

}

}

}

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

### 7.1.2 Binding to Arrays

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

data() {

return {

activeClass: 'active',

errorClass: 'text-danger'

}

}

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

Which will render:

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

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

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

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

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

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

### 7.1.3 With Components

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

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

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

<!-- child component template -->

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

Then add some classes when using it:

<!-- when using the component -->

<MyComponent class="baz boo" />

The rendered HTML will be:

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

The same is true for class bindings:

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

When isActive is truthy, the rendered HTML will be:

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

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

<!-- MyComponent template using $attrs -->

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

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

<MyComponent class="baz" />

Will render:

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

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

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

## 7.2 Binding Inline Styles

### 7.2.1 Binding to Objects

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

data() {

return {

activeColor: 'red',

fontSize: 30

}

}

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

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

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

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

data() {

return {

styleObject: {

color: 'red',

fontSize: '13px'

}

}

}

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

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

### 7.2.2 Binding to Arrays

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

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

### 7.2.3 Auto-prefixing

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

### 7.2.4 Multiple Values

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

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

This will only render the last value in the array which the browser supports. In this example, it will render display: flex for browsers that support the unprefixed version of flexbox.

# 第8章 Conditional Rendering

## 8.1 v-if

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

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

## 8.2 v-else

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

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

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

<h1 v-else>Oh no 😢</h1>

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

## 8.3 v-else-if

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

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

A

</div>

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

B

</div>

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

C

</div>

<div v-else>

Not A/B/C

</div>

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

## 8.4 v-if on <template>

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

<template v-if="ok">

<h1>Title</h1>

<p>Paragraph 1</p>

<p>Paragraph 2</p>

</template>

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

## 8.5 v-show

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

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

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

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

## 8.6 v-if vs v-show

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

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

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

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

## 8.7 v-if with v-for

Note

It's not recommended to use v-if and v-for on the same element due to implicit precedence. Refer to style guide for details.

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

# 第9章 List Rendering

## 9.1 v-for

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

data() {

return {

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

}

}

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

{{ item.message }}

</li>

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

data() {

return {

parentMessage: 'Parent',

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

}

}

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

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

</li>

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

const parentMessage = 'Parent'

const items = [

/\* ... \*/

]

items.forEach((item, index) => {

// has access to outer scope `parentMessage`

// but `item` and `index` are only available in here

console.log(parentMessage, item.message, index)

})

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

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

{{ message }}

</li>

<!-- with index alias -->

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

{{ message }} {{ index }}

</li>

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

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

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

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

</span>

</li>

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

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

## 9.2 v-for with an Object

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

data() {

return {

myObject: {

title: 'How to do lists in Vue',

author: 'Jane Doe',

publishedAt: '2016-04-10'

}

}

}

<ul>

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

{{ value }}

</li>

</ul>

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

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

{{ key }}: {{ value }}

</li>

And another for the index:

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

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

</li>

## 9.3 v-for with a Range

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

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

Note here n starts with an initial value of 1 instead of 0.

## 9.4 v-for on <template>

Similar to template v-if, you can also use a <template> tag with v-for to render a block of multiple elements. For example:

<ul>

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

<li>{{ item.msg }}</li>

<li class="divider" role="presentation"></li>

</template>

</ul>

## 9.5 v-for with v-if

Note

It's not recommended to use v-if and v-for on the same element due to implicit precedence. Refer to style guide for details.

When they exist on the same node, v-if has a higher priority than v-for. That means the v-if condition will not have access to variables from the scope of the v-for:

<!--

This will throw an error because property "todo"

is not defined on instance.

-->

<li v-for="todo in todos" v-if="!todo.isComplete">

{{ todo.name }}

</li>

This can be fixed by moving v-for to a wrapping <template> tag (which is also more explicit):

<template v-for="todo in todos">

<li v-if="!todo.isComplete">

{{ todo.name }}

</li>

</template>

## 9.6 Maintaining State with key

When Vue is updating a list of elements rendered with v-for, by default it uses an "in-place patch" strategy. If the order of the data items has changed, instead of moving the DOM elements to match the order of the items, Vue will patch each element in-place and make sure it reflects what should be rendered at that particular index.

This default mode is efficient, but only suitable when your list render output does not rely on child component state or temporary DOM state (e.g. form input values).

To give Vue a hint so that it can track each node's identity, and thus reuse and reorder existing elements, you need to provide a unique key attribute for each item:

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

<!-- content -->

</div>

When using <template v-for>, the key should be placed on the <template> container:

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

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

</template>

Note

key here is a special attribute being bound with v-bind. It should not be confused with the property key variable when using v-for with an object.

It is recommended to provide a key attribute with v-for whenever possible, unless the iterated DOM content is simple (i.e. contains no components or stateful DOM elements), or you are intentionally relying on the default behavior for performance gains.

The key binding expects primitive values - i.e. strings and numbers. Do not use objects as v-for keys. For detailed usage of the key attribute, please see the key API documentation.

## 9.7 v-for with a Component

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

You can directly use v-for on a component, like any normal element (don't forget to provide a key):

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

However, this won't automatically pass any data to the component, because components have isolated scopes of their own. In order to pass the iterated data into the component, we should also use props:

<MyComponent

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

:item="item"

:index="index"

:key="item.id"

/>

The reason for not automatically injecting item into the component is because that makes the component tightly coupled to how v-for works. Being explicit about where its data comes from makes the component reusable in other situations.

Check out this example of a simple todo list to see how to render a list of components using v-for, passing different data to each instance.

## 9.8 Array Change Detection

### 9.8.1 Mutation Methods

Vue is able to detect when a reactive array's mutation methods are called and trigger necessary updates. These mutation methods are:

push()

pop()

shift()

unshift()

splice()

sort()

reverse()

### 9.8.2 Replacing an Array

Mutation methods, as the name suggests, mutate the original array they are called on. In comparison, there are also non-mutating methods, e.g. filter(), concat() and slice(), which do not mutate the original array but always return a new array. When working with non-mutating methods, we should replace the old array with the new one:

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

You might think this will cause Vue to throw away the existing DOM and re-render the entire list - luckily, that is not the case. Vue implements some smart heuristics to maximize DOM element reuse, so replacing an array with another array containing overlapping objects is a very efficient operation.

## 9.9 Displaying Filtered/Sorted Results

Sometimes we want to display a filtered or sorted version of an array without actually mutating or resetting the original data. In this case, you can create a computed property that returns the filtered or sorted array.

For example:

data() {

return {

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

}

},

computed: {

evenNumbers() {

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

}

}

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

In situations where computed properties are not feasible (e.g. inside nested v-for loops), you can use a method:

data() {

return {

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

}

},

methods: {

even(numbers) {

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

}

}

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

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

</ul>

Be careful with reverse() and sort() in a computed property! These two methods will mutate the original array, which should be avoided in computed getters. Create a copy of the original array before calling these methods:

- return numbers.reverse()

+ return [...numbers].reverse()

# 第10章 Event Handling

## 10.1 Listening to Events

We can use the v-on directive, which we typically shorten to the @ symbol, to listen to DOM events and run some JavaScript when they're triggered. The usage would be v-on:click="handler" or with the shortcut, @click="handler".

The handler value can be one of the following:

Inline handlers: Inline JavaScript to be executed when the event is triggered (similar to the native onclick attribute).

Method handlers: A property name or path that points to a method defined on the component.

## 10.2 Inline Handlers

Inline handlers are typically used in simple cases, for example:

data() {

return {

count: 0

}

}

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

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

## 10.3 Method Handlers

The logic for many event handlers will be more complex though, and likely isn't feasible with inline handlers. That's why v-on can also accept the name or path of a component method you'd like to call.

For example:

data() {

return {

name: 'Vue.js'

}

},

methods: {

greet(event) {

// `this` inside methods points to the current active instance

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

// `event` is the native DOM event

if (event) {

alert(event.target.tagName)

}

}

}

<!-- `greet` is the name of the method defined above -->

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

A method handler automatically receives the native DOM Event object that triggers it - in the example above, we are able to access the element dispatching the event via event.target.tagName.

### 10.3.1 Method vs. Inline Detection

The template compiler detects method handlers by checking whether the v-on value string is a valid JavaScript identifier or property access path. For example, foo, foo.bar and foo['bar'] are treated as method handlers, while foo() and count++ are treated as inline handlers.

## 10.4 Calling Methods in Inline Handlers

Instead of binding directly to a method name, we can also call methods in an inline handler. This allows us to pass the method custom arguments instead of the native event:

methods: {

say(message) {

alert(message)

}

}

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

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

## 10.5 Accessing Event Argument in Inline Handlers

Sometimes we also need to access the original DOM event in an inline handler. You can pass it into a method using the special $event variable, or use an inline arrow function:

<!-- using $event special variable -->

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

Submit

</button>

<!-- using inline arrow function -->

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

Submit

</button>

methods: {

warn(message, event) {

// now we have access to the native event

if (event) {

event.preventDefault()

}

alert(message)

}

}

## 10.6 Event Modifiers

It is a very common need to call event.preventDefault() or event.stopPropagation() inside event handlers. Although we can do this easily inside methods, it would be better if the methods can be purely about data logic rather than having to deal with DOM event details.

To address this problem, Vue provides event modifiers for v-on. Recall that modifiers are directive postfixes denoted by a dot.

.stop

.prevent

.self

.capture

.once

.passive

<!-- the click event's propagation will be stopped -->

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

<!-- the submit event will no longer reload the page -->

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

<!-- modifiers can be chained -->

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

<!-- just the modifier -->

<form @submit.prevent></form>

<!-- only trigger handler if event.target is the element itself -->

<!-- i.e. not from a child element -->

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

TIP

Order matters when using modifiers because the relevant code is generated in the same order. Therefore using @click.prevent.self will prevent clicks default action on the element itself and its children while @click.self.prevent will only prevent clicks default action on the element itself.

The .capture, .once, and .passive modifiers mirror the options of the native addEventListener method:

<!-- use capture mode when adding the event listener -->

<!-- i.e. an event targeting an inner element is handled here before being handled by that element -->

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

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

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

<!-- the scroll event's default behavior (scrolling) will happen -->

<!-- immediately, instead of waiting for `onScroll` to complete -->

<!-- in case it contains `event.preventDefault()` -->

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

The .passive modifier is typically used with touch event listeners for improving performance on mobile devices.

TIP

Do not use .passive and .prevent together, because .passive already indicates to the browser that you do not intend to prevent the event's default behavior, and you will likely see a warning from the browser if you do so.

## 10.7 Key Modifiers

When listening for keyboard events, we often need to check for specific keys. Vue allows adding key modifiers for v-on or @ when listening for key events:

<!-- only call `vm.submit()` when the `key` is `Enter` -->

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

You can directly use any valid key names exposed via KeyboardEvent.key as modifiers by converting them to kebab-case.

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

In the above example, the handler will only be called if $event.key is equal to 'PageDown'.

### 10.7.1 Key Aliases

Vue provides aliases for the most commonly used keys:

.enter

.tab

.delete (captures both "Delete" and "Backspace" keys)

.esc

.space

.up

.down

.left

.right

### 10.7.2 System Modifier Keys

You can use the following modifiers to trigger mouse or keyboard event listeners only when the corresponding modifier key is pressed:

.ctrl

.alt

.shift

.meta

Note

On Macintosh keyboards, meta is the command key (⌘). On Windows keyboards, meta is the Windows key (⊞). On Sun Microsystems keyboards, meta is marked as a solid diamond (◆). On certain keyboards, specifically MIT and Lisp machine keyboards and successors, such as the Knight keyboard, space-cadet keyboard, meta is labeled “META”. On Symbolics keyboards, meta is labeled “META” or “Meta”.

For example:

<!-- Alt + Enter -->

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

<!-- Ctrl + Click -->

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

TIP

Note that modifier keys are different from regular keys and when used with keyup events, they have to be pressed when the event is emitted. In other words, keyup.ctrl will only trigger if you release a key while holding down ctrl. It won't trigger if you release the ctrl key alone.

### 10.7.3 .exact Modifier

The .exact modifier allows control of the exact combination of system modifiers needed to trigger an event.

<!-- this will fire even if Alt or Shift is also pressed -->

<button @click.ctrl="onClick">A</button>

<!-- this will only fire when Ctrl and no other keys are pressed -->

<button @click.ctrl.exact="onCtrlClick">A</button>

<!-- this will only fire when no system modifiers are pressed -->

<button @click.exact="onClick">A</button>

## 10.8 Mouse Button Modifiers

.left

.right

.middle

These modifiers restrict the handler to events triggered by a specific mouse button.

# 第11章 Form Input Bindings

When dealing with forms on the frontend, we often need to sync the state of form input elements with corresponding state in JavaScript. It can be cumbersome to manually wire up value bindings and change event listeners:

<input

:value="text"

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

The v-model directive helps us simplify the above to:

<input v-model="text">

In addition, v-model can be used on inputs of different types, <textarea>, and <select> elements. It automatically expands to different DOM property and event pairs based on the element it is used on:

<input> with text types and <textarea> elements use value property and input event;

<input type="checkbox"> and <input type="radio"> use checked property and change event;

<select> use value as a prop and change as an event.

Note

v-model will ignore the initial value, checked or selected attributes found on any form elements. It will always treat the current bound JavaScript state as the source of truth. You should declare the initial value on the JavaScript side, using the data option.

## 11.1 Basic Usage

### 11.1.1 Text

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

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

Message is:

edit me

Try it in the Playground

Note

For languages that require an IME (Chinese, Japanese, Korean etc.), you'll notice that v-model doesn't get updated during IME composition. If you want to respond to these updates as well, use your own input event listener and value binding instead of using v-model.

### 11.1.2 Multiline text

<span>Multiline message is:</span>

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

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

Note that interpolation inside <textarea> won't work. Use v-model instead.

<!-- bad -->

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

<!-- good -->

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

### 11.1.3 Checkbox

**Single checkbox, boolean value:**

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

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

**We can also bind multiple checkboxes to the same array or Set value:**

export default {

data() {

return {

checkedNames: []

}

}

}

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

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

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

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

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

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

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

In this case, the checkedNames array will always contain the values from the currently checked boxes.

### 11.1.4 Radio

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

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

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

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

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

### 11.1.5 Select

**Single select:**

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

<select v-model="selected">

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

<option>A</option>

<option>B</option>

<option>C</option>

</select>

Note

If the initial value of your v-model expression does not match any of the options, the <select> element will render in an "unselected" state. On iOS this will cause the user not being able to select the first item because iOS does not fire a change event in this case. It is therefore recommended to provide a disabled option with an empty value, as demonstrated in the example above.

**Multiple select (bound to array):**

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

<select v-model="selected" multiple>

<option>A</option>

<option>B</option>

<option>C</option>

</select>

Select options can be dynamically rendered with v-for:

export default {

data() {

return {

selected: 'A',

options: [

{ text: 'One', value: 'A' },

{ text: 'Two', value: 'B' },

{ text: 'Three', value: 'C' }

]

}

}

}

<select v-model="selected">

<option v-for="option in options" :value="option.value">

{{ option.text }}

</option>

</select>

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

## 11.2 Value Bindings

For radio, checkbox and select options, the v-model binding values are usually static strings (or booleans for checkbox):

<!-- `picked` is a string "a" when checked -->

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

<!-- `toggle` is either true or false -->

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

<!-- `selected` is a string "abc" when the first option is selected -->

<select v-model="selected">

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

</select>

But sometimes we may want to bind the value to a dynamic property on the current active instance. We can use v-bind to achieve that. In addition, using v-bind allows us to bind the input value to non-string values.

### 11.2.1 Checkbox

<input

type="checkbox"

v-model="toggle"

true-value="yes"

false-value="no" />

true-value and false-value are Vue-specific attributes that only work with v-model. Here the toggle property's value will be set to 'yes' when the box is checked, and set to 'no' when unchecked. You can also bind them to dynamic values using v-bind:

<input

type="checkbox"

v-model="toggle"

:true-value="dynamicTrueValue"

:false-value="dynamicFalseValue" />

Tip

The true-value and false-value attributes don't affect the input's value attribute, because browsers don't include unchecked boxes in form submissions. To guarantee that one of two values is submitted in a form (e.g. "yes" or "no"), use radio inputs instead.

### 11.2.2 Radio

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

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

pick will be set to the value of first when the first radio input is checked, and set to the value of second when the second one is checked.

### 11.2.3 Select Options

<select v-model="selected">

<!-- inline object literal -->

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

</select>

v-model supports value bindings of non-string values as well! In the above example, when the option is selected, selected will be set to the object literal value of { number: 123 }.

## 11.3 Modifiers

### 11.3.1 .lazy

By default, v-model syncs the input with the data after each input event (with the exception of IME composition as stated above). You can add the lazy modifier to instead sync after change events:

<!-- synced after "change" instead of "input" -->

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

### 11.3.2 .number

If you want user input to be automatically typecast as a number, you can add the number modifier to your v-model managed inputs:

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

If the value cannot be parsed with parseFloat(), then the original value is used instead.

The number modifier is applied automatically if the input has type="number".

### 11.3.3 .trim

If you want whitespace from user input to be trimmed automatically, you can add the trim modifier to your v-model-managed inputs:

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

## 11.4 v-model with Components

If you're not yet familiar with Vue's components, you can skip this for now.

HTML's built-in input types won't always meet your needs. Fortunately, Vue components allow you to build reusable inputs with completely customized behavior. These inputs even work with v-model! To learn more, read about Usage with v-model in the Components guide.

# 第12章 Lifecycle Hooks

Each Vue component instance goes through a series of initialization steps when it's created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called lifecycle hooks, giving users the opportunity to add their own code at specific stages.

## 12.1 Registering Lifecycle Hooks

For example, the mounted hook can be used to run code after the component has finished the initial rendering and created the DOM nodes:

export default {

mounted() {

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

}

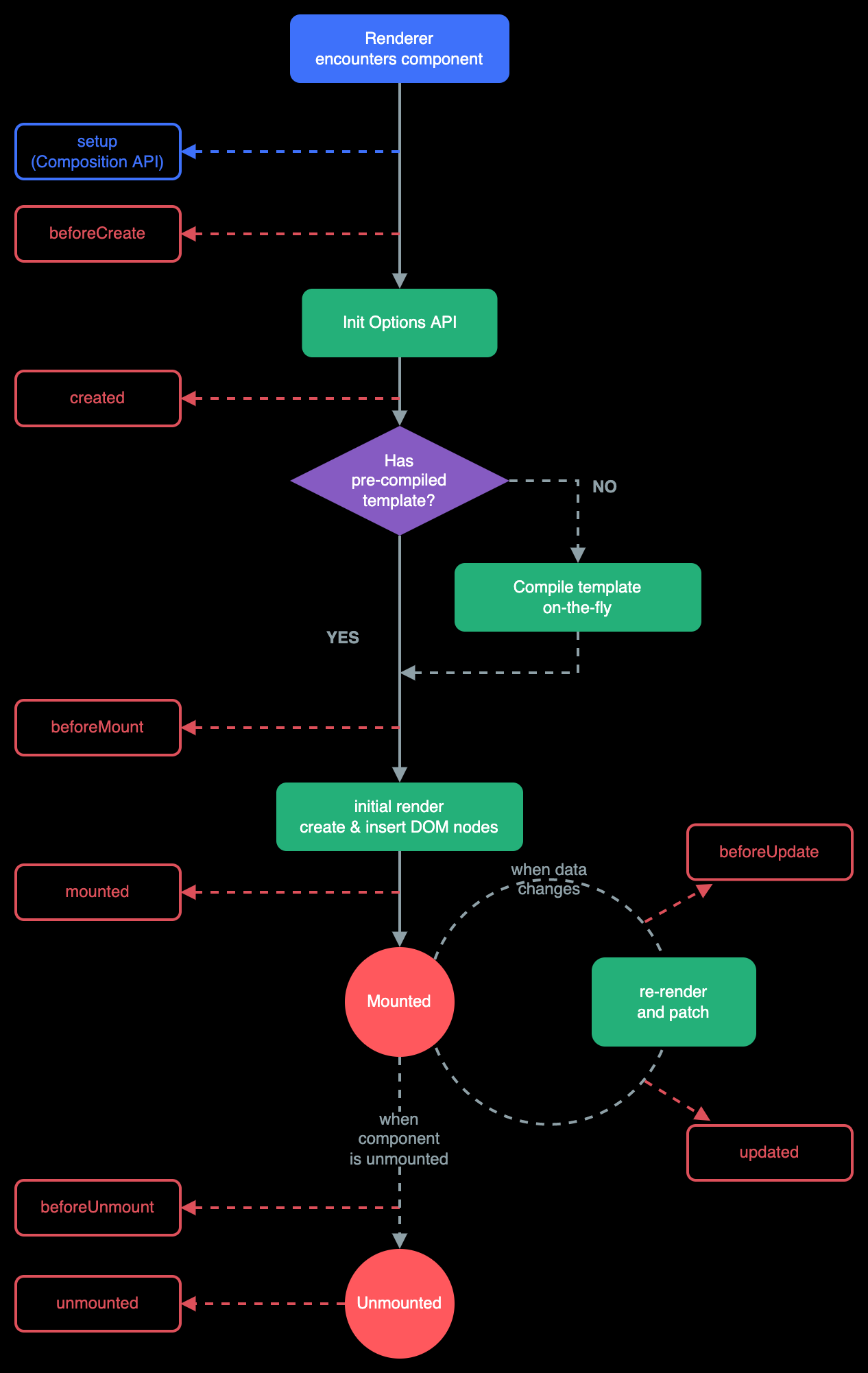
}

There are also other hooks which will be called at different stages of the instance's lifecycle, with the most commonly used being mounted, updated, and unmounted.

All lifecycle hooks are called with their this context pointing to the current active instance invoking it. Note this means you should avoid using arrow functions when declaring lifecycle hooks, as you won't be able to access the component instance via this if you do so.

## 12.2 Lifecycle Diagram

Below is a diagram for the instance lifecycle. You don't need to fully understand everything going on right now, but as you learn and build more, it will be a useful reference.



Consult the Lifecycle Hooks API reference for details on all lifecycle hooks and their respective use cases.

# 第13章 Watchers

## 13.1 Basic Example

Computed properties allow us to declaratively compute derived values. However, there are cases where we need to perform "side effects" in reaction to state changes - for example, mutating the DOM, or changing another piece of state based on the result of an async operation.

With the Options API, we can use the watch option to trigger a function whenever a reactive property changes:

export default {

data() {

return {

question: '',

answer: 'Questions usually contain a question mark. ;-)'

}

},

watch: {

// whenever question changes, this function will run

question(newQuestion, oldQuestion) {

if (newQuestion.includes('?')) {

this.getAnswer()

}

}

},

methods: {

async getAnswer() {

this.answer = 'Thinking...'

try {

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

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

} catch (error) {

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

}

}

}

}

<p>

Ask a yes/no question:

<input v-model="question" />

</p>

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

The watch option also supports a dot-delimited path as the key:

export default {

watch: {

// Note: only simple paths. Expressions are not supported.

'some.nested.key'(newValue) {

// ...

}

}

}

## 13.2 Deep Watchers

watch is shallow by default: the callback will only trigger when the watched property has been assigned a new value - it won't trigger on nested property changes. If you want the callback to fire on all nested mutations, you need to use a deep watcher:

export default {

watch: {

someObject: {

handler(newValue, oldValue) {

// Note: `newValue` will be equal to `oldValue` here

// on nested mutations as long as the object itself

// hasn't been replaced.

},

deep: true

}

}

}

Use with Caution

Deep watch requires traversing all nested properties in the watched object, and can be expensive when used on large data structures. Use it only when necessary and beware of the performance implications.

## 13.3 Eager Watchers

watch is lazy by default: the callback won't be called until the watched source has changed. But in some cases we may want the same callback logic to be run eagerly - for example, we may want to fetch some initial data, and then re-fetch the data whenever relevant state changes.

We can force a watcher's callback to be executed immediately by declaring it using an object with a handler function and the immediate: true option:

export default {

// ...

watch: {

question: {

handler(newQuestion) {

// this will be run immediately on component creation.

},

// force eager callback execution

immediate: true

}

}

// ...

}

## 13.4 Callback Flush Timing

When you mutate reactive state, it may trigger both Vue component updates and watcher callbacks created by you.

By default, user-created watcher callbacks are called before Vue component updates. This means if you attempt to access the DOM inside a watcher callback, the DOM will be in the state before Vue has applied any updates.

If you want to access the DOM in a watcher callback after Vue has updated it, you need to specify the flush: 'post' option:

export default {

// ...

watch: {

key: {

handler() {},

flush: 'post'

}

}

}

## 13.5 this.$watch()

It's also possible to imperatively create watchers using the $watch() instance method:

export default {

created() {

this.$watch('question', (newQuestion) => {

// ...

})

}

}

This is useful when you need to conditionally set up a watcher, or only watch something in response to user interaction. It also allows you to stop the watcher early.

## 13.6 Stopping a Watcher

Watchers declared using the watch option or the $watch() instance method are automatically stopped when the owner component is unmounted, so in most cases you don't need to worry about stopping the watcher yourself.

In the rare case where you need to stop a watcher before the owner component unmounts, the $watch() API returns a function for that:

const unwatch = this.$watch('foo', callback)

// ...when the watcher is no longer needed:

unwatch()

# 第14章 Template Refs

While Vue's declarative rendering model abstracts away most of the direct DOM operations for you, there may still be cases where we need direct access to the underlying DOM elements. To achieve this, we can use the special ref attribute:

<input ref="input">

ref is a special attribute, similar to the key attribute discussed in the v-for chapter. It allows us to obtain a direct reference to a specific DOM element or child component instance after it's mounted. This may be useful when you want to, for example, programmatically focus an input on component mount, or initialize a 3rd party library on an element.

## 14.1 Accessing the Refs

The resulting ref is exposed on this.$refs:

<script>

export default {

mounted() {

this.$refs.input.focus()

}

}

</script>

<template>

<input ref="input" />

</template>

Note that you can only access the ref after the component is mounted. If you try to access $refs.input in a template expression, it will be null on the first render. This is because the element doesn't exist until after the first render!

## 14.2 Refs inside v-for

Requires v3.2.25 or above

When ref is used inside v-for, the resulting ref value will be an array containing the corresponding elements:

<script>

export default {

data() {

return {

list: [

/\* ... \*/

]

}

},

mounted() {

console.log(this.$refs.items)

}

}

</script>

<template>

<ul>

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

{{ item }}

</li>

</ul>

</template>

It should be noted that the ref array does not guarantee the same order as the source array.

## 14.3 Function Refs

Instead of a string key, the ref attribute can also be bound to a function, which will be called on each component update and gives you full flexibility on where to store the element reference. The function receives the element reference as the first argument:

<input :ref="(el) => { /\* assign el to a property or ref \*/ }">

Note we are using a dynamic :ref binding so we can pass it a function instead of a ref name string. When the element is unmounted, the argument will be null. You can, of course, use a method instead of an inline function.

## 14.4 Ref on Component

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

ref can also be used on a child component. In this case the reference will be that of a component instance:

<script>

import Child from './Child.vue'

export default {

components: {

Child

},

mounted() {

// this.$refs.child will hold an instance of <Child />

}

}

</script>

<template>

<Child ref="child" />

</template>

The referenced instance will be identical to the child component's this, which means the parent component will have full access to every property and method of the child component. This makes it easy to create tightly coupled implementation details between the parent and the child, so component refs should be only used when absolutely needed - in most cases, you should try to implement parent / child interactions using the standard props and emit interfaces first.

The expose option can be used to limit the access to a child instance:

export default {

expose: ['publicData', 'publicMethod'],

data() {

return {

publicData: 'foo',

privateData: 'bar'

}

},

methods: {

publicMethod() {

/\* ... \*/

},

privateMethod() {

/\* ... \*/

}

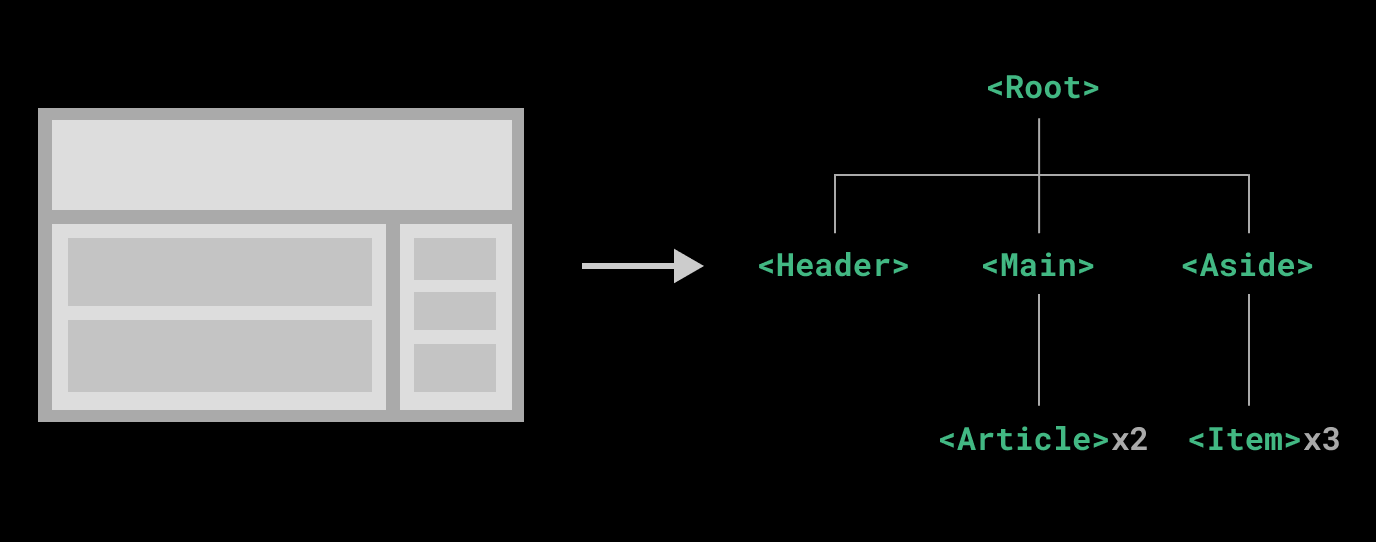
}

}

In the above example, a parent referencing this component via template ref will only be able to access publicData and publicMethod.

# 第15章 Components Basics

Components allow us to split the UI into independent and reusable pieces, and think about each piece in isolation. It's common for an app to be organized into a tree of nested components:



## 15.1 Defining a Component

When using a build step, we typically define each Vue component in a dedicated file using the .vue extension - known as a Single-File Component (SFC for short):

<script>

export default {

data() {

return {

count: 0

}

}

}

</script>

<template>

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

</template>

When not using a build step, a Vue component can be defined as a plain JavaScript object containing Vue-specific options:

export default {

data() {

return {

count: 0

}

},

template: `

<button @click="count++">

You clicked me {{ count }} times.

</button>`

}

The template is inlined as a JavaScript string here, which Vue will compile on the fly. You can also use an ID selector pointing to an element (usually native <template> elements) - Vue will use its content as the template source.

The example above defines a single component and exports it as the default export of a .js file, but you can use named exports to export multiple components from the same file.

## 15.2 Using a Component

TIP

We will be using SFC syntax for the rest of this guide - the concepts around components are the same regardless of whether you are using a build step or not. The Examples section shows component usage in both scenarios.

To use a child component, we need to import it in the parent component. Assuming we placed our counter component inside a file called ButtonCounter.vue, the component will be exposed as the file's default export:

<script>

import ButtonCounter from './ButtonCounter.vue'

export default {

components: {

ButtonCounter

}

}

</script>

<template>

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

<ButtonCounter />

</template>

To expose the imported component to our template, we need to register it with the components option. The component will then be available as a tag using the key it is registered under.

It's also possible to globally register a component, making it available to all components in a given app without having to import it. The pros and cons of global vs. local registration is discussed in the dedicated Component Registration section.

Components can be reused as many times as you want:

<h1>Here are many child components!</h1>

<ButtonCounter />

<ButtonCounter />

<ButtonCounter />

Notice that when clicking on the buttons, each one maintains its own, separate count. That's because each time you use a component, a new instance of it is created.

In SFCs, it's recommended to use PascalCase tag names for child components to differentiate from native HTML elements. Although native HTML tag names are case-insensitive, Vue SFC is a compiled format so we are able to use case-sensitive tag names in it. We are also able to use /> to close a tag.

If you are authoring your templates directly in a DOM (e.g. as the content of a native <template> element), the template will be subject to the browser's native HTML parsing behavior. In such cases, you will need to use kebab-case and explicit closing tags for components:

<!-- if this template is written in the DOM -->

<button-counter></button-counter>

<button-counter></button-counter>

<button-counter></button-counter>

See DOM template parsing caveats for more details.

## 15.3 Passing Props

If we are building a blog, we will likely need a component representing a blog post. We want all the blog posts to share the same visual layout, but with different content. Such a component won't be useful unless you can pass data to it, such as the title and content of the specific post we want to display. That's where props come in.

Props are custom attributes you can register on a component. To pass a title to our blog post component, we must declare it in the list of props this component accepts, using the props option:

<!-- BlogPost.vue -->

<script>

export default {

props: ['title']

}

</script>

<template>

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

</template>

When a value is passed to a prop attribute, it becomes a property on that component instance. The value of that property is accessible within the template and on the component's this context, just like any other component property.

A component can have as many props as you like and, by default, any value can be passed to any prop.

Once a prop is registered, you can pass data to it as a custom attribute, like this:

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

<BlogPost title="Blogging with Vue" />

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

In a typical app, however, you'll likely have an array of posts in your parent component:

export default {

// ...

data() {

return {

posts: [

{ id: 1, title: 'My journey with Vue' },

{ id: 2, title: 'Blogging with Vue' },

{ id: 3, title: 'Why Vue is so fun' }

]

}

}

}

Then want to render a component for each one, using v-for:

<BlogPost

v-for="post in posts"

:key="post.id"

:title="post.title"

/>

Notice how v-bind is used to pass dynamic prop values. This is especially useful when you don't know the exact content you're going to render ahead of time.

That's all you need to know about props for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Props.

## 15.4 Listening to Events

As we develop our <BlogPost> component, some features may require communicating back up to the parent. For example, we may decide to include an accessibility feature to enlarge the text of blog posts, while leaving the rest of the page at its default size.

In the parent, we can support this feature by adding a postFontSize data property:

data() {

return {

posts: [

/\* ... \*/

],

postFontSize: 1

}

}

Which can be used in the template to control the font size of all blog posts:

<div :style="{ fontSize: postFontSize + 'em' }">

<BlogPost

v-for="post in posts"

:key="post.id"

:title="post.title"

/>

</div>

Now let's add a button to the <BlogPost> component's template:

<!-- BlogPost.vue, omitting <script> -->

<template>

<div class="blog-post">

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

<button>Enlarge text</button>

</div>

</template>

The button doesn't do anything yet - we want clicking the button to communicate to the parent that it should enlarge the text of all posts. To solve this problem, components provide a custom events system. The parent can choose to listen to any event on the child component instance with v-on or @, just as we would with a native DOM event:

<BlogPost

...

@enlarge-text="postFontSize += 0.1"

/>

Then the child component can emit an event on itself by calling the built-in $emit method, passing the name of the event:

<!-- BlogPost.vue, omitting <script> -->

<template>

<div class="blog-post">

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

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

</div>

</template>

Thanks to the @enlarge-text="postFontSize += 0.1" listener, the parent will receive the event and update the value of postFontSize.

We can optionally declare emitted events using the emits option:

<!-- BlogPost.vue -->

<script>

export default {

props: ['title'],

emits: ['enlarge-text']

}

</script>

This documents all the events that a component emits and optionally validates them. It also allows Vue to avoid implicitly applying them as native listeners to the child component's root element.

That's all you need to know about custom component events for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Custom Events.

## 15.5 Content Distribution with Slots

Just like with HTML elements, it's often useful to be able to pass content to a component, like this:

<AlertBox>

Something bad happened.

</AlertBox>

Which might render something like:

This is an Error for Demo Purposes

Something bad happened.

This can be achieved using Vue's custom <slot> element:

<template>

<div class="alert-box">

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

<slot />

</div>

</template>

<style scoped>

.alert-box {

/\* ... \*/

}

</style>

As you'll see above, we use the <slot> as a placeholder where we want the content to go – and that's it. We're done!

That's all you need to know about slots for now, but once you've finished reading this page and feel comfortable with its content, we recommend coming back later to read the full guide on Slots.

## 15.6 Dynamic Components

Sometimes, it's useful to dynamically switch between components, like in a tabbed interface:

The above is made possible by Vue's <component> element with the special is attribute:

<!-- Component changes when currentTab changes -->

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

In the example above, the value passed to :is can contain either:

the name string of a registered component, OR

the actual imported component object

You can also use the is attribute to create regular HTML elements.

When switching between multiple components with <component :is="...">, a component will be unmounted when it is switched away from. We can force the inactive components to stay "alive" with the built-in <KeepAlive> component.

## 15.7 DOM Template Parsing Caveats

If you are writing your Vue templates directly in the DOM, Vue will have to retrieve the template string from the DOM. This leads to some caveats due to browsers' native HTML parsing behavior.

**TIP**

It should be noted that the limitations discussed below only apply if you are writing your templates directly in the DOM. They do NOT apply if you are using string templates from the following sources:

Single-File Components

Inlined template strings (e.g. template: '...')

<script type="text/x-template">

### 15.7.1 Case Insensitivity

HTML tags and attribute names are case-insensitive, so browsers will interpret any uppercase characters as lowercase. That means when you’re using in-DOM templates, PascalCase component names and camelCased prop names or v-on event names all need to use their kebab-cased (hyphen-delimited) equivalents:

// camelCase in JavaScript

const BlogPost = {

props: ['postTitle'],

emits: ['updatePost'],

template: `

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

`

}

<!-- kebab-case in HTML -->

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

### 15.7.2 Self Closing Tags

We have been using self-closing tags for components in previous code samples:

<MyComponent />

This is because Vue's template parser respects /> as an indication to end any tag, regardless of its type.

In DOM templates, however, we must always include explicit closing tags:

<my-component></my-component>

This is because the HTML spec only allows a few specific elements to omit closing tags, the most common being <input> and <img>. For all other elements, if you omit the closing tag, the native HTML parser will think you never terminated the opening tag. For example, the following snippet:

<my-component /> <!-- we intend to close the tag here... -->

<span>hello</span>

will be parsed as:

<my-component>

<span>hello</span>

</my-component> <!-- but the browser will close it here. -->

### 15.7.3 Element Placement Restrictions

Some HTML elements, such as <ul>, <ol>, <table> and <select> have restrictions on what elements can appear inside them, and some elements such as <li>, <tr>, and <option> can only appear inside certain other elements.

This will lead to issues when using components with elements that have such restrictions. For example:

<table>

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

</table>

The custom component <blog-post-row> will be hoisted out as invalid content, causing errors in the eventual rendered output. We can use the special is attribute as a workaround:

<table>

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

</table>

TIP

When used on native HTML elements, the value of is must be prefixed with vue: in order to be interpreted as a Vue component. This is required to avoid confusion with native customized built-in elements.

That's all you need to know about DOM template parsing caveats for now - and actually, the end of Vue's Essentials. Congratulations! There's still more to learn, but first, we recommend taking a break to play with Vue yourself - build something fun, or check out some of the Examples if you haven't already.

Once you feel comfortable with the knowledge you've just digested, move on with the guide to learn more about components in depth.

# 第16章 Component Registration

This page assumes you've already read the Components Basics. Read that first if you are new to components.

A Vue component needs to be "registered" so that Vue knows where to locate its implementation when it is encountered in a template. There are two ways to register components: global and local.

## 16.1 Global Registration

We can make components available globally in the current Vue application using the app.component() method:

import { createApp } from 'vue'

const app = createApp({})

app.component(

// the registered name

'MyComponent',

// the implementation

{

/\* ... \*/

}

)

If using SFCs, you will be registering the imported .vue files:

import MyComponent from './App.vue'

app.component('MyComponent', MyComponent)

The app.component() method can be chained:

app

.component('ComponentA', ComponentA)

.component('ComponentB', ComponentB)

.component('ComponentC', ComponentC)

Globally registered components can be used in the template of any component within this application:

<!-- this will work in any component inside the app -->

<ComponentA/>

<ComponentB/>

<ComponentC/>

This even applies to all subcomponents, meaning all three of these components will also be available inside each other.

## 16.2 Local Registration

While convenient, global registration has a few drawbacks:

Global registration prevents build systems from removing unused components (a.k.a "tree-shaking"). If you globally register a component but end up not using it anywhere in your app, it will still be included in the final bundle.

Global registration makes dependency relationships less explicit in large applications. It makes it difficult to locate a child component's implementation from a parent component using it. This can affect long-term maintainability similar to using too many global variables.

Local registration scopes the availability of the registered components to the current component only. It makes the dependency relationship more explicit, and is more tree-shaking friendly.

Local registration is done using the components option:

<script>

import ComponentA from './ComponentA.vue'

export default {

components: {

ComponentA

}

}

</script>

<template>

<ComponentA />

</template>

For each property in the components object, the key will be the registered name of the component, while the value will contain the implementation of the component. The above example is using the ES2015 property shorthand and is equivalent to:

export default {

components: {

ComponentA: ComponentA

}

// ...

}

Note that locally registered components are not also available in descendent components. In this case, ComponentA will be made available to the current component only, not any of its child or descendent components.

## 16.3 Component Name Casing

Throughout the guide, we are using PascalCase names when registering components. This is because:

PascalCase names are valid JavaScript identifiers. This makes it easier to import and register components in JavaScript. It also helps IDEs with auto-completion.

<PascalCase /> makes it more obvious that this is a Vue component instead of a native HTML element in templates. It also differentiates Vue components from custom elements (web components).

This is the recommended style when working with SFC or string templates. However, as discussed in DOM Template Parsing Caveats, PascalCase tags are not usable in DOM templates.

Luckily, Vue supports resolving kebab-case tags to components registered using PascalCase. This means a component registered as MyComponent can be referenced in the template via both <MyComponent> and <my-component>. This allows us to use the same JavaScript component registration code regardless of template source.

# 第 17章 Props

## 17.1 Props Declaration

Vue components require explicit props declaration so that Vue knows what external props passed to the component should be treated as fallthrough attributes (which will be discussed in its dedicated section).

Props are declared using the props option:

export default {

props: ['foo'],

created() {

// props are exposed on `this`

console.log(this.foo)

}

}

In addition to declaring props using an array of strings, we can also use the object syntax:

export default {

props: {

title: String,

likes: Number

}

}

For each property in the object declaration syntax, the key is the name of the prop, while the value should be the constructor function of the expected type.

This not only documents your component, but will also warn other developers using your component in the browser console if they pass the wrong type. We will discuss more details about prop validation further down this page.

See also: Typing Component Props

## 17.2 Prop Passing Details

### 17.2.1 Prop Name Casing

We declare long prop names using camelCase because this avoids having to use quotes when using them as property keys, and allows us to reference them directly in template expressions because they are valid JavaScript identifiers:

export default {

props: {

greetingMessage: String

}

}

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

Technically, you can also use camelCase when passing props to a child component (except in DOM templates). However, the convention is using kebab-case in all cases to align with HTML attributes:

<MyComponent greeting-message="hello" />

We use PascalCase for component tags when possible because it improves template readability by differentiating Vue components from native elements. However, there isn't as much practical benefit in using camelCase when passing props, so we choose to follow each language's conventions.

### 17.2.2 Static vs. Dynamic Props

So far, you've seen props passed as static values, like in:

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

You've also seen props assigned dynamically with v-bind or its : shortcut, such as in:

<!-- Dynamically assign the value of a variable -->

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

<!-- Dynamically assign the value of a complex expression -->

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

### 17.2.3 Passing Different Value Types

In the two examples above, we happen to pass string values, but any type of value can be passed to a prop.

**Number**

<!-- Even though `42` is static, we need v-bind to tell Vue that -->

<!-- this is a JavaScript expression rather than a string. -->

<BlogPost :likes="42" />

<!-- Dynamically assign to the value of a variable. -->

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

**Boolean**

<!-- Including the prop with no value will imply `true`. -->

<BlogPost is-published />

<!-- Even though `false` is static, we need v-bind to tell Vue that -->

<!-- this is a JavaScript expression rather than a string. -->

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

<!-- Dynamically assign to the value of a variable. -->

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

**Array**

<!-- Even though the array is static, we need v-bind to tell Vue that -->

<!-- this is a JavaScript expression rather than a string. -->

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

<!-- Dynamically assign to the value of a variable. -->

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

**Object**

<!-- Even though the object is static, we need v-bind to tell Vue that -->

<!-- this is a JavaScript expression rather than a string. -->

<BlogPost

:author="{

name: 'Veronica',

company: 'Veridian Dynamics'

}"

/>

<!-- Dynamically assign to the value of a variable. -->

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

### 17.2.4 Binding Multiple Properties Using an Object

If you want to pass all the properties of an object as props, you can use v-bind without an argument (v-bind instead of :prop-name). For example, given a post object:

export default {

data() {

return {

post: {

id: 1,

title: 'My Journey with Vue'

}

}

}

}

The following template:

<BlogPost v-bind="post" />

Will be equivalent to:

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

## 17.3 One-Way Data Flow

All props form a one-way-down binding between the child property and the parent one: when the parent property updates, it will flow down to the child, but not the other way around. This prevents child components from accidentally mutating the parent's state, which can make your app's data flow harder to understand.

In addition, every time the parent component is updated, all props in the child component will be refreshed with the latest value. This means you should not attempt to mutate a prop inside a child component. If you do, Vue will warn you in the console:

export default {

props: ['foo'],

created() {

// ❌ warning, props are readonly!

this.foo = 'bar'

}

}

There are usually two cases where it's tempting to mutate a prop:

The prop is used to pass in an initial value; the child component wants to use it as a local data property afterwards. In this case, it's best to define a local data property that uses the prop as its initial value:

export default {

props: ['initialCounter'],

data() {

return {

// counter only uses this.initialCounter as the initial value;

// it is disconnected from future prop updates.

counter: this.initialCounter

}

}

}

The prop is passed in as a raw value that needs to be transformed. In this case, it's best to define a computed property using the prop's value:

export default {

props: ['size'],

computed: {

// computed property that auto-updates when the prop changes

normalizedSize() {

return this.size.trim().toLowerCase()

}

}

}

### 17.3.1 Mutating Object / Array Props

When objects and arrays are passed as props, while the child component cannot mutate the prop binding, it will be able to mutate the object or array's nested properties. This is because in JavaScript objects and arrays are passed by reference, and it is unreasonably expensive for Vue to prevent such mutations.

The main drawback of such mutations is that it allows the child component to affect parent state in a way that isn't obvious to the parent component, potentially making it more difficult to reason about the data flow in the future. As a best practice, you should avoid such mutations unless the parent and child are tightly coupled by design. In most cases, the child should emit an event to let the parent perform the mutation.

## 17.4 Prop Validation

Components can specify requirements for their props, such as the types you've already seen. If a requirement is not met, Vue will warn you in the browser's JavaScript console. This is especially useful when developing a component that is intended to be used by others.

To specify prop validations, you can provide an object with validation requirements to the props option, instead of an array of strings. For example:

export default {

props: {

// Basic type check

// (`null` and `undefined` values will allow any type)

propA: Number,

// Multiple possible types

propB: [String, Number],

// Required string

propC: {

type: String,

required: true

},

// Number with a default value

propD: {

type: Number,

default: 100

},

// Object with a default value

propE: {

type: Object,

// Object or array defaults must be returned from

// a factory function. The function receives the raw

// props received by the component as the argument.

default(rawProps) {

return { message: 'hello' }

}

},

// Custom validator function

propF: {

validator(value) {

// The value must match one of these strings

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

}

},

// Function with a default value

propG: {

type: Function,

// Unlike object or array default, this is not a factory function - this is a function to serve as a default value

default() {

return 'Default function'

}

}

}

}

Additional details:

All props are optional by default, unless required: true is specified.

An absent optional prop other than Boolean will have undefined value.

The Boolean absent props will be cast to false. You should set a default value for it in order to get desired behavior.

If a default value is specified, it will be used if the resolved prop value is undefined - this includes both when the prop is absent, or an explicit undefined value is passed.

When prop validation fails, Vue will produce a console warning (if using the development build).

Note

Note that props are validated before a component instance is created, so instance properties (e.g. data, computed, etc.) will not be available inside default or validator functions.

### 17.4.1 Runtime Type Checks

The type can be one of the following native constructors:

String

Number

Boolean

Array

Object

Date

Function

Symbol

In addition, type can also be a custom class or constructor function and the assertion will be made with an instanceof check. For example, given the following class:

class Person {

constructor(firstName, lastName) {

this.firstName = firstName

this.lastName = lastName

}

}

You could use it as a prop's type:

export default {

props: {

author: Person

}

}

Vue will use instanceof Person to validate whether the value of the author prop is indeed an instance of the Person class.

## 17.5 Boolean Casting

Props with Boolean type has special casting rules to mimic the behavior of native boolean attributes. Given a <MyComponent> with the following declaration:

export default {

props: {

disabled: Boolean

}

}

The component can be used like this:

<!-- equivalent of passing :disabled="true" -->

<MyComponent disabled />

<!-- equivalent of passing :disabled="false" -->

<MyComponent />

When a prop is declared to allow multiple types, e.g.

export default {

props: {

disabled: [Boolean, Number]

}

}

The casting rules for Boolean will apply regardless of type appearance order.

# 第18章 Component Events

## 18.1 Emitting and Listening to Events

A component can emit custom events directly in template expressions (e.g. in a v-on handler) using the built-in $emit method:

<!-- MyComponent -->

<button @click="$emit('someEvent')">click me</button>

The $emit() method is also available on the component instance as this.$emit():

export default {

methods: {

submit() {

this.$emit('submit')

}

}

}

The parent can then listen to it using v-on:

<MyComponent @some-event="callback" />

The .once modifier is also supported on component event listeners:

<MyComponent @some-event.once="callback" />

Like components and props, event names provide an automatic case transformation. Notice we emitted a camelCase event, but can listen for it using a kebab-cased listener in the parent. As with props casing, we recommend using kebab-cased event listeners in templates.

TIP

Unlike native DOM events, component emitted events do not bubble. You can only listen to the events emitted by a direct child component. If there is a need to communicate between sibling or deeply nested components, use an external event bus or a global state management solution.

## 18.2 Event Arguments

It's sometimes useful to emit a specific value with an event. For example, we may want the <BlogPost> component to be in charge of how much to enlarge the text by. In those cases, we can pass extra arguments to $emit to provide this value:

<button @click="$emit('increaseBy', 1)">

Increase by 1

</button>

Then, when we listen to the event in the parent, we can use an inline arrow function as the listener, which allows us to access the event argument:

<MyButton @increase-by="(n) => count += n" />

Or, if the event handler is a method:

<MyButton @increase-by="increaseCount" />

Then the value will be passed as the first parameter of that method:

methods: {

increaseCount(n) {

this.count += n

}

}

TIP

All extra arguments passed to $emit() after the event name will be forwarded to the listener. For example, with $emit('foo', 1, 2, 3) the listener function will receive three arguments.

## 18.3 Declaring Emitted Events

Emitted events can be explicitly declared on the component via the emits option:

export default {

emits: ['inFocus', 'submit']

}

The emits option also supports an object syntax, which allows us to perform runtime validation of the payload of the emitted events:

export default {

emits: {

submit(payload) {

// return `true` or `false` to indicate

// validation pass / fail

}

}

}

See also: Typing Component Emits

Although optional, it is recommended to define all emitted events in order to better document how a component should work. It also allows Vue to exclude known listeners from fallthrough attributes, avoiding edge cases caused by DOM events manually dispatched by 3rd party code.

TIP

If a native event (e.g., click) is defined in the emits option, the listener will now only listen to component-emitted click events and no longer respond to native click events.

## 18.4 Events Validation

Similar to prop type validation, an emitted event can be validated if it is defined with the object syntax instead of the array syntax.

To add validation, the event is assigned a function that receives the arguments passed to the this.$emit call and returns a boolean to indicate whether the event is valid or not.

export default {

emits: {

// No validation

click: null,

// Validate submit event

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

if (email && password) {

return true

} else {

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

return false

}

}

},

methods: {

submitForm(email, password) {

this.$emit('submit', { email, password })

}

}

}

## 18.5 Usage with v-model

Custom events can also be used to create custom inputs that work with v-model. Let's revisit how v-model is used on a native element:

<input v-model="searchText" />

Under the hood, the template compiler expands v-model to the more verbose equivalent for us. So the above code does the same as the following:

<input

:value="searchText"

@input="searchText = $event.target.value"

/>

When used on a component, v-model instead expands to this:

<CustomInput

:modelValue="searchText"

@update:modelValue="newValue => searchText = newValue"

/>

For this to actually work though, the <CustomInput> component must do two things:

Bind the value attribute of a native <input> element to the modelValue prop

When a native input event is triggered, emit an update:modelValue custom event with the new value

Here's that in action:

<!-- CustomInput.vue -->

<script>

export default {

props: ['modelValue'],

emits: ['update:modelValue']

}

</script>

<template>

<input

:value="modelValue"

@input="$emit('update:modelValue', $event.target.value)"

/>

</template>

Now v-model should work perfectly with this component:

<CustomInput v-model="searchText" />

Another way of implementing v-model within this component is to use a writable computed property with both a getter and a setter. The get method should return the modelValue property and the set method should emit the corresponding event:

<!-- CustomInput.vue -->

<script>

export default {

props: ['modelValue'],

emits: ['update:modelValue'],

computed: {

value: {

get() {

return this.modelValue

},

set(value) {

this.$emit('update:modelValue', value)

}

}

}

}

</script>

<template>

<input v-model="value" />

</template>

### 18.5.1 v-model arguments

By default, v-model on a component uses modelValue as the prop and update:modelValue as the event. We can modify these names passing an argument to v-model:

<MyComponent v-model:title="bookTitle" />

In this case, the child component should expect a title prop and emit an update:title event to update the parent value:

<!-- MyComponent.vue -->

<script>

export default {

props: ['title'],

emits: ['update:title']

}

</script>

<template>

<input

type="text"

:value="title"

@input="$emit('update:title', $event.target.value)"

/>

</template>

### 18.5.2 Multiple v-model bindings

By leveraging the ability to target a particular prop and event as we learned before with v-model arguments, we can now create multiple v-model bindings on a single component instance.

Each v-model will sync to a different prop, without the need for extra options in the component:

<UserName

v-model:first-name="first"

v-model:last-name="last"

/>

<script>

export default {

props: {

firstName: String,

lastName: String

},

emits: ['update:firstName', 'update:lastName']

}

</script>

<template>

<input

type="text"

:value="firstName"

@input="$emit('update:firstName', $event.target.value)"

/>

<input

type="text"

:value="lastName"

@input="$emit('update:lastName', $event.target.value)"

/>

</template>

### 18.5.3 Handling v-model modifiers

When we were learning about form input bindings, we saw that v-model has built-in modifiers - .trim, .number and .lazy. In some cases, you might also want the v-model on your custom input component to support custom modifiers.

Let's create an example custom modifier, capitalize, that capitalizes the first letter of the string provided by the v-model binding:

<MyComponent v-model.capitalize="myText" />

Modifiers added to a component v-model will be provided to the component via the modelModifiers prop. In the below example, we have created a component that contains a modelModifiers prop that defaults to an empty object:

<script>

export default {

props: {

modelValue: String,

modelModifiers: {

default: () => ({})

}

},

emits: ['update:modelValue'],

created() {

console.log(this.modelModifiers) // { capitalize: true }

}

}

</script>

<template>

<input

type="text"

:value="modelValue"

@input="$emit('update:modelValue', $event.target.value)"

/>

</template>

Notice the component's modelModifiers prop contains capitalize and its value is true - due to it being set on the v-model binding v-model.capitalize="myText".

Now that we have our prop set up, we can check the modelModifiers object keys and write a handler to change the emitted value. In the code below we will capitalize the string whenever the <input /> element fires an input event.

<script>

export default {

props: {

modelValue: String,

modelModifiers: {

default: () => ({})

}

},

emits: ['update:modelValue'],

methods: {

emitValue(e) {

let value = e.target.value

if (this.modelModifiers.capitalize) {

value = value.charAt(0).toUpperCase() + value.slice(1)

}

this.$emit('update:modelValue', value)

}

}

}

</script>

<template>

<input type="text" :value="modelValue" @input="emitValue" />

</template>

For v-model bindings with both argument and modifiers, the generated prop name will be arg + "Modifiers". For example:

<MyComponent v-model:title.capitalize="myText">

The corresponding declarations should be:

export default {

props: ['title', 'titleModifiers'],

emits: ['update:title'],

created() {

console.log(this.titleModifiers) // { capitalize: true }

}

}

# 第19章 Fallthrough Attributes

## 19.1 Attribute Inheritance

A "fallthrough attribute" is an attribute or v-on event listener that is passed to a component, but is not explicitly declared in the receiving component's props or emits. Common examples of this include class, style, and id attributes.

When a component renders a single root element, fallthrough attributes will be automatically added to the root element's attributes. For example, given a <MyButton> component with the following template:

<!-- template of <MyButton> -->

<button>click me</button>

And a parent using this component with:

<MyButton class="large" />

The final rendered DOM would be:

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

Here, <MyButton> did not declare class as an accepted prop. Therefore, class is treated as a fallthrough attribute and automatically added to <MyButton>'s root element.

### 19.1.1 class and style Merging

If the child component's root element already has existing class or style attributes, it will be merged with the class and style values that are inherited from the parent. Suppose we change the template of <MyButton> in the previous example to:

<!-- template of <MyButton> -->

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

Then the final rendered DOM would now become:

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

### 19.1.2 v-on Listener Inheritance

The same rule applies to v-on event listeners:

<MyButton @click="onClick" />

The click listener will be added to the root element of <MyButton>, i.e. the native <button> element. When the native <button> is clicked, it will trigger the onClick method of the parent component. If the native <button> already has a click listener bound with v-on, then both listeners will trigger.

### 19.1.3 Nested Component Inheritance

If a component renders another component as its root node, for example, we refactored <MyButton> to render a <BaseButton> as its root:

<!-- template of <MyButton/> that simply renders another component -->

<BaseButton />

Then the fallthrough attributes received by <MyButton> will be automatically forwarded to <BaseButton>.

Note that:

Forwarded attributes do not include any attributes that are declared as props, or v-on listeners of declared events by <MyButton> - in other words, the declared props and listeners have been "consumed" by <MyButton>.

Forwarded attributes may be accepted as props by <BaseButton>, if declared by it.

## 19.2 Disabling Attribute Inheritance

If you do not want a component to automatically inherit attributes, you can set inheritAttrs: false in the component's options.

The common scenario for disabling attribute inheritance is when attributes need to be applied to other elements besides the root node. By setting the inheritAttrs option to false, you can take full control over where the fallthrough attributes should be applied.

These fallthrough attributes can be accessed directly in template expressions as $attrs:

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

The $attrs object includes all attributes that are not declared by the component's props or emits options (e.g., class, style, v-on listeners, etc.).

Some notes:

Unlike props, fallthrough attributes preserve their original casing in JavaScript, so an attribute like foo-bar needs to be accessed as $attrs['foo-bar'].

A v-on event listener like @click will be exposed on the object as a function under $attrs.onClick.

Using our <MyButton> component example from the previous section - sometimes we may need to wrap the actual <button> element with an extra <div> for styling purposes:

<div class="btn-wrapper">

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

</div>

We want all fallthrough attributes like class and v-on listeners to be applied to the inner <button>, not the outer <div>. We can achieve this with inheritAttrs: false and v-bind="$attrs":

<div class="btn-wrapper">

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

</div>

Remember that v-bind without an argument binds all the properties of an object as attributes of the target element.

## 19.3 Attribute Inheritance on Multiple Root Nodes

Unlike components with a single root node, components with multiple root nodes do not have an automatic attribute fallthrough behavior. If $attrs are not bound explicitly, a runtime warning will be issued.

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

If <CustomLayout> has the following multi-root template, there will be a warning because Vue cannot be sure where to apply the fallthrough attributes:

<header>...</header>

<main>...</main>

<footer>...</footer>

The warning will be suppressed if $attrs is explicitly bound:

<header>...</header>

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

<footer>...</footer>

## 19.4 Accessing Fallthrough Attributes in JavaScript

If needed, you can access a component's fallthrough attributes via the $attrs instance property:

export default {

created() {

console.log(this.$attrs)

}

}

# 第20章 Slots

## 20.1 Slot Content and Outlet

We have learned that components can accept props, which can be JavaScript values of any type. But how about template content? In some cases, we may want to pass a template fragment to a child component, and let the child component render the fragment within its own template.

For example, we may have a <FancyButton> component that supports usage like this:

<FancyButton>

Click me! <!-- slot content -->

</FancyButton>

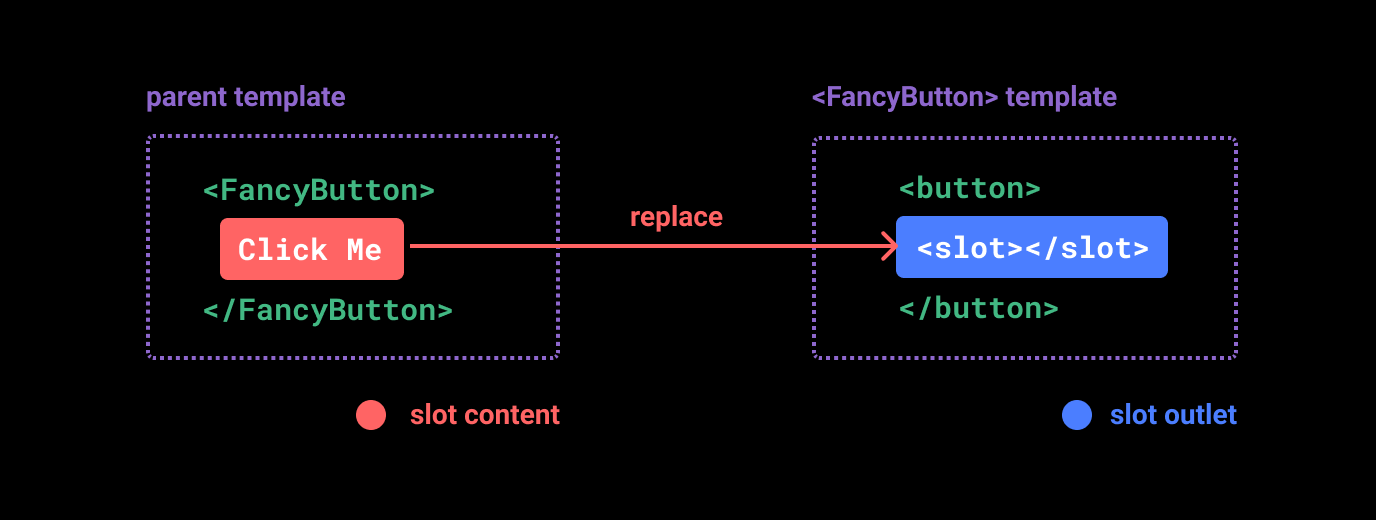
The template of <FancyButton> looks like this:

<button class="fancy-btn">

<slot></slot> <!-- slot outlet -->

</button>

The <slot> element is a slot outlet that indicates where the parent-provided slot content should be rendered.



And the final rendered DOM:

<button class="fancy-btn">Click me!</button>

With slots, the <FancyButton> is responsible for rendering the outer <button> (and its fancy styling), while the inner content is provided by the parent component.

Another way to understand slots is by comparing them to JavaScript functions:

// parent component passing slot content

FancyButton('Click me!')

// FancyButton renders slot content in its own template

function FancyButton(slotContent) {

return `<button class="fancy-btn">

${slotContent}

</button>`

}

Slot content is not just limited to text. It can be any valid template content. For example, we can pass in multiple elements, or even other components:

<FancyButton>

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

<AwesomeIcon name="plus" />

</FancyButton>

By using slots, our <FancyButton> is more flexible and reusable. We can now use it in different places with different inner content, but all with the same fancy styling.

Vue components' slot mechanism is inspired by the native Web Component <slot> element, but with additional capabilities that we will see later.

## 20.2 Render Scope

Slot content has access to the data scope of the parent component, because it is defined in the parent. For example:

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

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

Here both {{ message }} interpolations will render the same content.

Slot content does not have access to the child component's data. Expressions in Vue templates can only access the scope it is defined in, consistent with JavaScript's lexical scoping. In other words:

Expressions in the parent template only have access to the parent scope; expressions in the child template only have access to the child scope.

## 20.3 Fallback Content

There are cases when it's useful to specify fallback (i.e. default) content for a slot, to be rendered only when no content is provided. For example, in a <SubmitButton> component:

<button type="submit">

<slot></slot>

</button>

We might want the text "Submit" to be rendered inside the <button> if the parent didn't provide any slot content. To make "Submit" the fallback content, we can place it in between the <slot> tags:

<button type="submit">

<slot>

Submit <!-- fallback content -->

</slot>

</button>

Now when we use <SubmitButton> in a parent component, providing no content for the slot:

<SubmitButton />

This will render the fallback content, "Submit":

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

But if we provide content:

<SubmitButton>Save</SubmitButton>

Then the provided content will be rendered instead:

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

## 20.4 Named Slots

There are times when it's useful to have multiple slot outlets in a single component. For example, in a <BaseLayout> component with the following template:

<div class="container">

<header>

<!-- We want header content here -->

</header>

<main>

<!-- We want main content here -->

</main>

<footer>

<!-- We want footer content here -->

</footer>

</div>

For these cases, the <slot> element has a special attribute, name, which can be used to assign a unique ID to different slots so you can determine where content should be rendered:

<div class="container">

<header>

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

</header>

<main>

<slot></slot>

</main>

<footer>

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

</footer>

</div>

A <slot> outlet without name implicitly has the name "default".

In a parent component using <BaseLayout>, we need a way to pass multiple slot content fragments, each targeting a different slot outlet. This is where named slots come in.

To pass a named slot, we need to use a <template> element with the v-slot directive, and then pass the name of the slot as an argument to v-slot:

<BaseLayout>

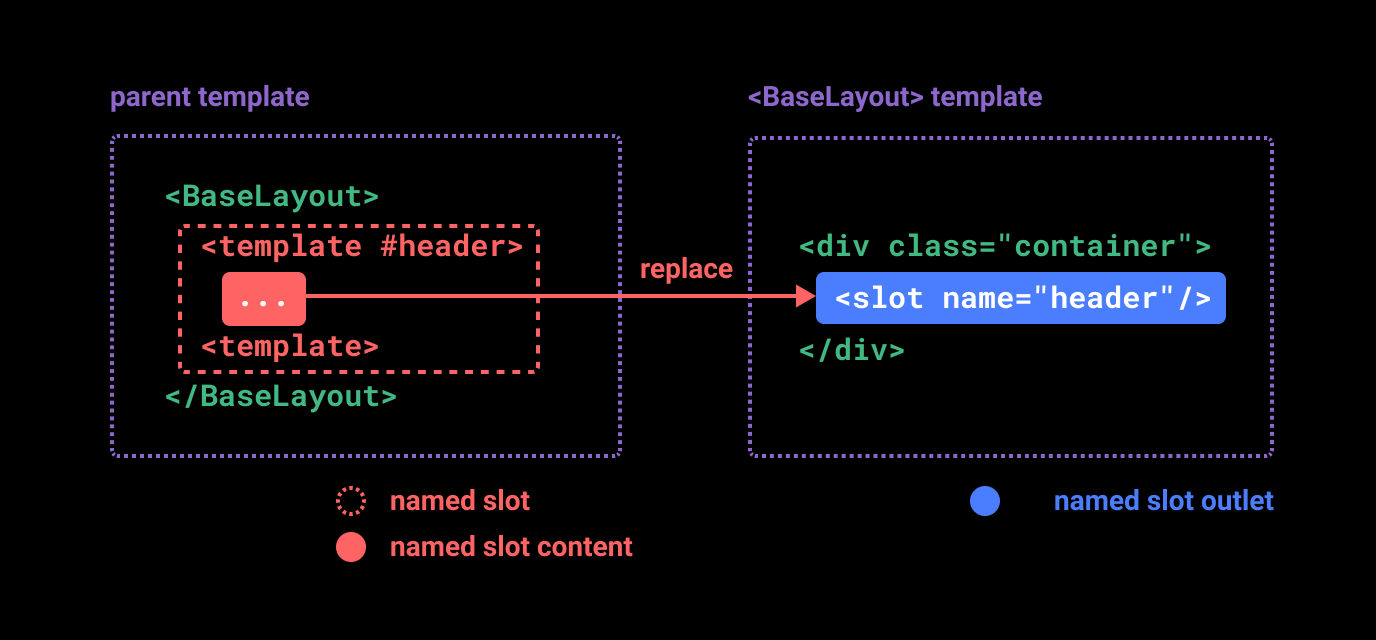
<template v-slot:header>

<!-- content for the header slot -->

</template>

</BaseLayout>

v-slot has a dedicated shorthand #, so <template v-slot:header> can be shortened to just <template #header>. Think of it as "render this template fragment in the child component's 'header' slot".



Here's the code passing content for all three slots to <BaseLayout> using the shorthand syntax:

<BaseLayout>

<template #header>

<h1>Here might be a page title</h1>

</template>

<template #default>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

</template>

<template #footer>

<p>Here's some contact info</p>

</template>

</BaseLayout>

When a component accepts both a default slot and named slots, all top-level non-<template> nodes are implicitly treated as content for the default slot. So the above can also be written as:

<BaseLayout>

<template #header>

<h1>Here might be a page title</h1>

</template>

<!-- implicit default slot -->

<p>A paragraph for the main content.</p>

<p>And another one.</p>

<template #footer>

<p>Here's some contact info</p>

</template>

</BaseLayout>

Now everything inside the <template> elements will be passed to the corresponding slots. The final rendered HTML will be:

<div class="container">

<header>

<h1>Here might be a page title</h1>

</header>

<main>

<p>A paragraph for the main content.</p>

<p>And another one.</p>

</main>

<footer>

<p>Here's some contact info</p>

</footer>

</div>

Again, it may help you understand named slots better using the JavaScript function analogy:

// passing multiple slot fragments with different names

BaseLayout({

header: `...`,

default: `...`,

footer: `...`

})

// <BaseLayout> renders them in different places

function BaseLayout(slots) {

return `<div class="container">

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

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

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

</div>`

}

## 20.5 Dynamic Slot Names

Dynamic directive arguments also work on v-slot, allowing the definition of dynamic slot names:

<base-layout>

<template v-slot:[dynamicSlotName]>

...

</template>

<!-- with shorthand -->

<template #[dynamicSlotName]>

...

</template>

</base-layout>

Do note the expression is subject to the syntax constraints of dynamic directive arguments.

## 20.6 Scoped Slots

As discussed in Render Scope, slot content does not have access to state in the child component.

However, there are cases where it could be useful if a slot's content can make use of data from both the parent scope and the child scope. To achieve that, we need a way for the child to pass data to a slot when rendering it.

In fact, we can do exactly that - we can pass attributes to a slot outlet just like passing props to a component:

<!-- <MyComponent> template -->

<div>

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

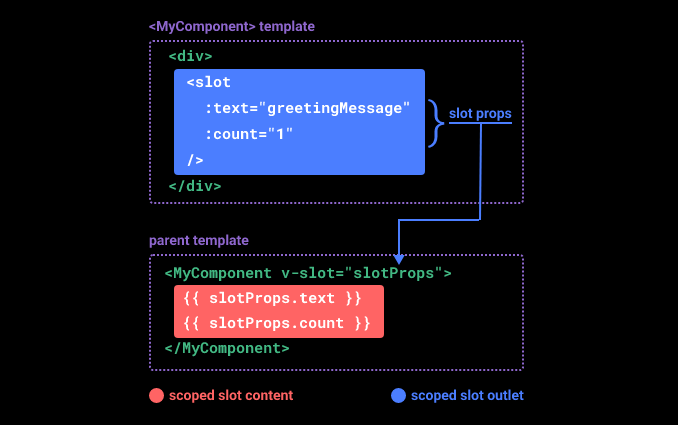
</div>

Receiving the slot props is a bit different when using a single default slot vs. using named slots. We are going to show how to receive props using a single default slot first, by using v-slot directly on the child component tag:

<MyComponent v-slot="slotProps">

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

</MyComponent>



The props passed to the slot by the child are available as the value of the corresponding v-slot directive, which can be accessed by expressions inside the slot.

You can think of a scoped slot as a function being passed into the child component. The child component then calls it, passing props as arguments:

MyComponent({

// passing the default slot, but as a function

default: (slotProps) => {

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

}

})

function MyComponent(slots) {

const greetingMessage = 'hello'

return `<div>${

// call the slot function with props!

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

}</div>`

}

In fact, this is very close to how scoped slots are compiled, and how you would use scoped slots in manual render functions.

Notice how v-slot="slotProps" matches the slot function signature. Just like with function arguments, we can use destructuring in v-slot:

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

{{ text }} {{ count }}

</MyComponent>

### 20.6.1 Named Scoped Slots

Named scoped slots work similarly - slot props are accessible as the value of the v-slot directive: v-slot:name="slotProps". When using the shorthand, it looks like this:

<MyComponent>

<template #header="headerProps">

{{ headerProps }}

</template>

<template #default="defaultProps">

{{ defaultProps }}

</template>

<template #footer="footerProps">

{{ footerProps }}

</template>

</MyComponent>

Passing props to a named slot:

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

Note the name of a slot won't be included in the props because it is reserved - so the resulting headerProps would be { message: 'hello' }.

### 20.6.2 Fancy List Example

You may be wondering what would be a good use case for scoped slots. Here's an example: imagine a <FancyList> component that renders a list of items - it may encapsulate the logic for loading remote data, using the data to display a list, or even advanced features like pagination or infinite scrolling. However, we want it to be flexible with how each item looks and leave the styling of each item to the parent component consuming it. So the desired usage may look like this:

<FancyList :api-url="url" :per-page="10">

<template #item="{ body, username, likes }">

<div class="item">

<p>{{ body }}</p>

<p>by {{ username }} | {{ likes }} likes</p>

</div>

</template>

</FancyList>

Inside <FancyList>, we can render the same <slot> multiple times with different item data (notice we are using v-bind to pass an object as slot props):

<ul>

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

<slot name="item" v-bind="item"></slot>

</li>

</ul>

### 20.6.3 Renderless Components

The <FancyList> use case we discussed above encapsulates both reusable logic (data fetching, pagination etc.) and visual output, while delegating part of the visual output to the consumer component via scoped slots.

If we push this concept a bit further, we can come up with components that only encapsulate logic and do not render anything by themselves - visual output is fully delegated to the consumer component with scoped slots. We call this type of component a Renderless Component.

An example renderless component could be one that encapsulates the logic of tracking the current mouse position:

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

Mouse is at: {{ x }}, {{ y }}

</MouseTracker>

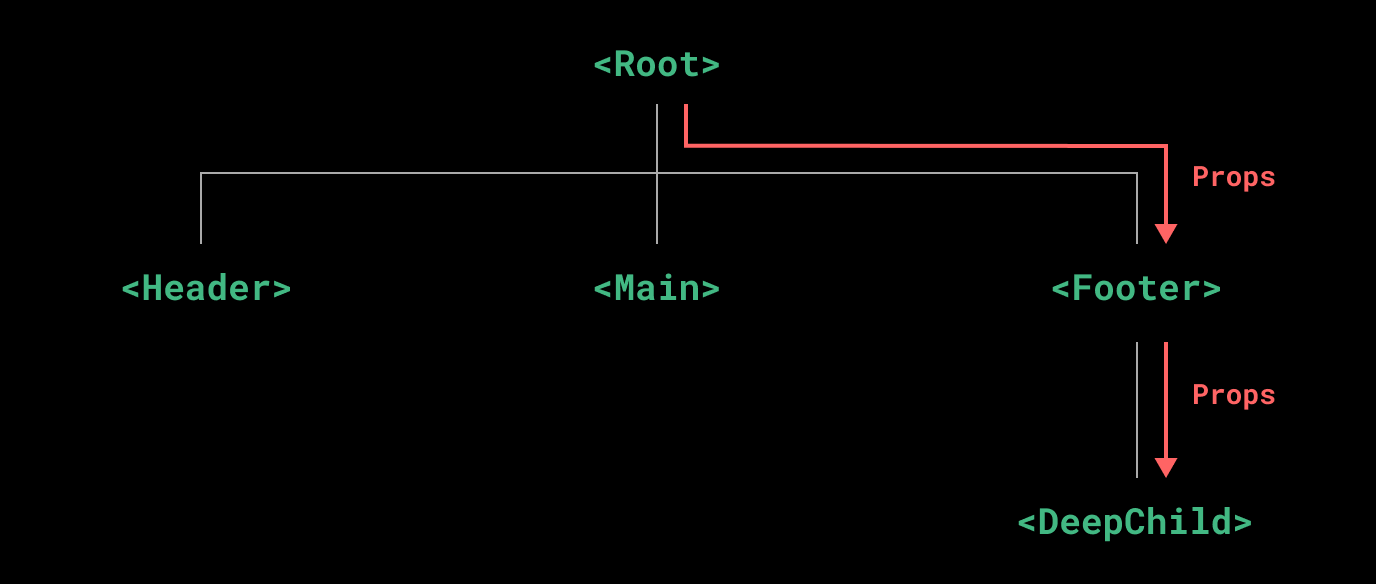
While an interesting pattern, most of what can be achieved with Renderless Components can be achieved in a more efficient fashion with Composition API, without incurring the overhead of extra component nesting. Later, we will see how we can implement the same mouse tracking functionality as a Composable.

That said, scoped slots are still useful in cases where we need to both encapsulate logic and compose visual output, like in the <FancyList> example.

# 第21章 Provide / Inject

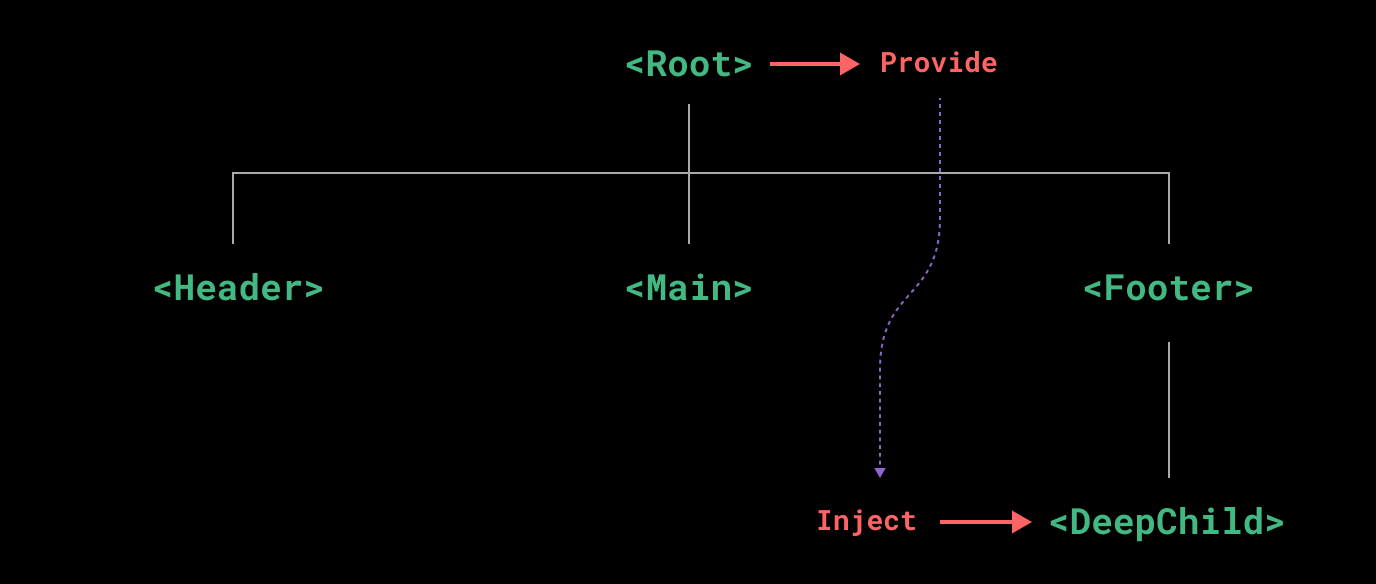
## 21. Prop Drilling

Usually, when we need to pass data from the parent to a child component, we use props. However, imagine the case where we have a large component tree, and a deeply nested component needs something from a distant ancestor component. With only props, we would have to pass the same prop across the entire parent chain:



Notice although the <Footer> component may not care about these props at all, it still needs to declare and pass them along just so <DeepChild> can access them. If there is a longer parent chain, more components would be affected along the way. This is called "props drilling" and definitely isn't fun to deal with.

We can solve props drilling with provide and inject. A parent component can serve as a dependency provider for all its descendants. Any component in the descendant tree, regardless of how deep it is, can inject dependencies provided by components up in its parent chain.



## 21.2 Provide

To provide data to a component's descendants, use the provide option:

export default {

provide: {

message: 'hello!'

}

}

For each property in the provide object, the key is used by child components to locate the correct value to inject, while the value is what ends up being injected.

If we need to provide per-instance state, for example data declared via the data(), then provide must use a function value:

export default {

data() {

return {

message: 'hello!'

}

},

provide() {

// use function syntax so that we can access `this`

return {

message: this.message

}

}

}

However, do note this does not make the injection reactive. We will discuss making injections reactive below.

## 21.3 App-level Provide

In addition to providing data in a component, we can also provide at the app level:

import { createApp } from 'vue'

const app = createApp({})

app.provide(/\* key \*/ 'message', /\* value \*/ 'hello!')

App-level provides are available to all components rendered in the app. This is especially useful when writing plugins, as plugins typically wouldn't be able to provide values using components.

## 21.4 inject

To inject data provided by an ancestor component, use the inject option:

export default {

inject: ['message'],

created() {

console.log(this.message) // injected value

}

}

Injections are resolved before the component's own state, so you can access injected properties in data():

export default {

inject: ['message'],

data() {

return {

// initial data based on injected value

fullMessage: this.message

}

}

}

### 21.4.1 Injection Aliasing

When using the array syntax for inject, the injected properties are exposed on the component instance using the same key. In the example above, the property was provided under the key "message", and injected as this.message. The local key is the same as the injection key.

If we want to inject the property using a different local key, we need to use the object syntax for the inject option:

export default {

inject: {

/\* local key \*/ localMessage: {

from: /\* injection key \*/ 'message'

}

}

}

Here, the component will locate a property provided with the key "message", and then expose it as this.localMessage.

### 21.4.2 Injection Default Values

By default, inject assumes that the injected key is provided somewhere in the parent chain. In the case where the key is not provided, there will be a runtime warning.

If we want to make an injected property work with optional providers, we need to declare a default value, similar to props:

export default {

// object syntax is required

// when declaring default values for injections

inject: {

message: {

from: 'message', // this is optional if using the same key for injection

default: 'default value'

},

user: {

// use a factory function for non-primitive values that are expensive

// to create, or ones that should be unique per component instance.

default: () => ({ name: 'John' })

}

}

}

## 21.5 Working with Reactivity

In order to make injections reactively linked to the provider, we need to provide a computed property using the computed() function:

import { computed } from 'vue'

export default {

data() {

return {

message: 'hello!'

}

},

provide() {

return {

// explicitly provide a computed property

message: computed(() => this.message)

}

}

}

The computed() function is typically used in Composition API components, but can also be used to complement certain use cases in Options API. You can learn more about its usage by reading the Reactivity Fundamentals and Computed Properties with the API Preference set to Composition API.

Temporary Config Required

The above usage requires setting app.config.unwrapInjectedRef = true to make injections automatically unwrap computed refs. This will become the default behavior in Vue 3.3 and this config is introduced temporarily to avoid breakage. It will no longer be required after 3.3.

## 21.6 Working with Symbol Keys

So far, we have been using string injection keys in the examples. If you are working in a large application with many dependency providers, or you are authoring components that are going to be used by other developers, it is best to use Symbol injection keys to avoid potential collisions.

It's recommended to export the Symbols in a dedicated file:

// keys.js

export const myInjectionKey = Symbol()

// in provider component

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

export default {

provide() {

return {

[myInjectionKey]: {

/\* data to provide \*/

}

}

}

}

// in injector component

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

export default {

inject: {

injected: { from: myInjectionKey }

}

}

# 第22章 Async Components

## 22.1 Basic Usage

In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it's needed. To make that possible, Vue has a defineAsyncComponent function:

import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() => {

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

// ...load component from server

resolve(/\* loaded component \*/)

})

})

// ... use `AsyncComp` like a normal component

As you can see, defineAsyncComponent accepts a loader function that returns a Promise. The Promise's resolve callback should be called when you have retrieved your component definition from the server. You can also call reject(reason) to indicate the load has failed.

ES module dynamic import also returns a Promise, so most of the time we will use it in combination with defineAsyncComponent. Bundlers like Vite and webpack also support the syntax (and will use it as bundle split points), so we can use it to import Vue SFCs:

import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent(() =>

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

)

The resulting AsyncComp is a wrapper component that only calls the loader function when it is actually rendered on the page. In addition, it will pass along any props and slots to the inner component, so you can use the async wrapper to seamlessly replace the original component while achieving lazy loading.

As with normal components, async components can be registered globally using app.component():

app.component('MyComponent', defineAsyncComponent(() =>

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

))

You can also use defineAsyncComponent when registering a component locally:

<script>

import { defineAsyncComponent } from 'vue'

export default {

components: {

AdminPage: defineAsyncComponent(() =>

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

)

}

}

</script>

<template>

<AdminPage />

</template>

## 22.2 Loading and Error States

Asynchronous operations inevitably involve loading and error states - defineAsyncComponent() supports handling these states via advanced options:

const AsyncComp = defineAsyncComponent({

// the loader function

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

// A component to use while the async component is loading

loadingComponent: LoadingComponent,

// Delay before showing the loading component. Default: 200ms.

delay: 200,

// A component to use if the load fails

errorComponent: ErrorComponent,

// The error component will be displayed if a timeout is

// provided and exceeded. Default: Infinity.

timeout: 3000

})

If a loading component is provided, it will be displayed first while the inner component is being loaded. There is a default 200ms delay before the loading component is shown - this is because on fast networks, an instant loading state may get replaced too fast and end up looking like a flicker.

If an error component is provided, it will be displayed when the Promise returned by the loader function is rejected. You can also specify a timeout to show the error component when the request is taking too long.

## 22.3 Using with Suspense

Async components can be used with the <Suspense> built-in component. The interaction between <Suspense> and async components are documented in the dedicated chapter for <Suspense>.

# 第23章 Composables

## 23.1 What is a "Composable"?

In the context of Vue applications, a "composable" is a function that leverages Vue's Composition API to encapsulate and reuse stateful logic.

When building frontend applications, we often need to reuse logic for common tasks. For example, we may need to format dates in many places, so we extract a reusable function for that. This formatter function encapsulates stateless logic: it takes some input and immediately returns expected output. There are many libraries out there for reusing stateless logic - for example lodash and date-fns, which you may have heard of.

By contrast, stateful logic involves managing state that changes over time. A simple example would be tracking the current position of the mouse on a page. In real world scenarios, it could also be more complex logic such as touch gestures or connection status to a database.

## 23.2 Mouse Tracker Example

If we were to implement the mouse tracking functionality using the Composition API directly inside a component, it would look like this:

<script setup>

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

const x = ref(0)

const y = ref(0)

function update(event) {

x.value = event.pageX

y.value = event.pageY

}

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

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

</script>

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

But what if we want to reuse the same logic in multiple components? We can extract the logic into an external file, as a composable function:

// mouse.js

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

// by convention, composable function names start with "use"

export function useMouse() {

// state encapsulated and managed by the composable

const x = ref(0)

const y = ref(0)

// a composable can update its managed state over time.

function update(event) {

x.value = event.pageX

y.value = event.pageY

}

// a composable can also hook into its owner component's

// lifecycle to setup and teardown side effects.

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

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

// expose managed state as return value

return { x, y }

}

And this is how it can be used in components:

<script setup>

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

const { x, y } = useMouse()

</script>

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

As we can see, the core logic remains identical - all we had to do was move it into an external function and return the state that should be exposed. Just like inside a component, you can use the full range of Composition API functions in composables. The same useMouse() functionality can now be used in any component.

The cooler part about composables though, is that you can also nest them: one composable function can call one or more other composable functions. This enables us to compose complex logic using small, isolated units, similar to how we compose an entire application using components. In fact, this is why we decided to call the collection of APIs that make this pattern possible Composition API.

For example, we can extract the logic of adding and removing a DOM event listener into its own composable:

// event.js

import { onMounted, onUnmounted } from 'vue'

export function useEventListener(target, event, callback) {

// if you want, you can also make this

// support selector strings as target

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

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

}

And now our useMouse() composable can be simplified to:

// mouse.js

import { ref } from 'vue'

import { useEventListener } from './event'

export function useMouse() {

const x = ref(0)

const y = ref(0)

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

x.value = event.pageX

y.value = event.pageY

})

return { x, y }

}

**TIP**

Each component instance calling useMouse() will create its own copies of x and y state so they won't interfere with one another. If you want to manage shared state between components, read the State Management chapter.

## 23.3 Async State Example

The useMouse() composable doesn't take any arguments, so let's take a look at another example that makes use of one. When doing async data fetching, we often need to handle different states: loading, success, and error:

<script setup>

import { ref } from 'vue'

const data = ref(null)

const error = ref(null)

fetch('...')

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

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

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

</script>

<template>

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

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

Data loaded:

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

</div>

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

</template>

It would be tedious to have to repeat this pattern in every component that needs to fetch data. Let's extract it into a composable:

// fetch.js

import { ref } from 'vue'

export function useFetch(url) {

const data = ref(null)

const error = ref(null)

fetch(url)

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

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

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

return { data, error }

}

Now in our component we can just do:

<script setup>

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

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

</script>

useFetch() takes a static URL string as input - so it performs the fetch only once and is then done. What if we want it to re-fetch whenever the URL changes? We can achieve that by also accepting refs as an argument:

// fetch.js

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

export function useFetch(url) {

const data = ref(null)

const error = ref(null)

function doFetch() {

// reset state before fetching..

data.value = null

error.value = null

// unref() unwraps potential refs

fetch(unref(url))

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

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

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

}

if (isRef(url)) {

// setup reactive re-fetch if input URL is a ref

watchEffect(doFetch)

} else {

// otherwise, just fetch once

// and avoid the overhead of a watcher

doFetch()

}

return { data, error }

}

This version of useFetch() now accepts both static URL strings and refs of URL strings. When it detects that the URL is a dynamic ref using isRef(), it sets up a reactive effect using watchEffect(). The effect will run immediately and will also track the URL ref as a dependency. Whenever the URL ref changes, the data will be reset and fetched again.

Here's the updated version of useFetch(), with an artificial delay and randomized error for demo purposes.

## 23.4 Conventions and Best Practices

### 23.4.1 Naming

It is a convention to name composable functions with camelCase names that start with "use".

### 23.4.2 Input Arguments

A composable can accept ref arguments even if it doesn't rely on them for reactivity. If you are writing a composable that may be used by other developers, it's a good idea to handle the case of input arguments being refs instead of raw values. The unref() utility function will come in handy for this purpose:

import { unref } from 'vue'

function useFeature(maybeRef) {

// if maybeRef is indeed a ref, its .value will be returned

// otherwise, maybeRef is returned as-is

const value = unref(maybeRef)

}

If your composable creates reactive effects when the input is a ref, make sure to either explicitly watch the ref with watch(), or call unref() inside a watchEffect() so that it is properly tracked.

### 23.4.3 Return Values

You have probably noticed that we have been exclusively using ref() instead of reactive() in composables. The recommended convention is for composables to always return a plain, non-reactive object containing multiple refs. This allows it to be destructured in components while retaining reactivity:

// x and y are refs

const { x, y } = useMouse()

Returning a reactive object from a composable will cause such destructures to lose the reactivity connection to the state inside the composable, while the refs will retain that connection.

If you prefer to use returned state from composables as object properties, you can wrap the returned object with reactive() so that the refs are unwrapped. For example:

const mouse = reactive(useMouse())

// mouse.x is linked to original ref

console.log(mouse.x)

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

### 23.4.4 Side Effects

It is OK to perform side effects (e.g. adding DOM event listeners or fetching data) in composables, but pay attention to the following rules:

If you are working on an application that uses Server-Side Rendering (SSR), make sure to perform DOM-specific side effects in post-mount lifecycle hooks, e.g. onMounted(). These hooks are only called in the browser, so you can be sure that code inside them has access to the DOM.

Remember to clean up side effects in onUnmounted(). For example, if a composable sets up a DOM event listener, it should remove that listener in onUnmounted() as we have seen in the useMouse() example. It can be a good idea to use a composable that automatically does this for you, like the useEventListener() example.

### 23.4.5 Usage Restrictions

Composables should only be called synchronously in <script setup> or the setup() hook. In some cases, you can also call them in lifecycle hooks like onMounted().

These are the contexts where Vue is able to determine the current active component instance. Access to an active component instance is necessary so that:

Lifecycle hooks can be registered to it.

Computed properties and watchers can be linked to it, so that they can be disposed when the instance is unmounted to prevent memory leaks.

TIP

<script setup> is the only place where you can call composables after using await. The compiler automatically restores the active instance context for you after the async operation.

## 23.5 Extracting Composables for Code Organization

Composables can be extracted not only for reuse, but also for code organization. As the complexity of your components grow, you may end up with components that are too large to navigate and reason about. Composition API gives you the full flexibility to organize your component code into smaller functions based on logical concerns:

<script setup>

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

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

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

const { foo, bar } = useFeatureA()

const { baz } = useFeatureB(foo)

const { qux } = useFeatureC(baz)

</script>

To some extent, you can think of these extracted composables as component-scoped services that can talk to one another.

## 23.6 Using Composables in Options API

If you are using Options API, composables must be called inside setup(), and the returned bindings must be returned from setup() so that they are exposed to this and the template:

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

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

export default {

setup() {

const { x, y } = useMouse()

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

return { x, y, data, error }

},

mounted() {

// setup() exposed properties can be accessed on `this`

console.log(this.x)

}

// ...other options

}

## 23.7 Comparisons with Other Techniques

### 23.7.1 vs. Mixins

Users coming from Vue 2 may be familiar with the mixins option, which also allows us to extract component logic into reusable units. There are three primary drawbacks to mixins:

Unclear source of properties: when using many mixins, it becomes unclear which instance property is injected by which mixin, making it difficult to trace the implementation and understand the component's behavior. This is also why we recommend using the refs + destructure pattern for composables: it makes the property source clear in consuming components.

Namespace collisions: multiple mixins from different authors can potentially register the same property keys, causing namespace collisions. With composables, you can rename the destructured variables if there are conflicting keys from different composables.

Implicit cross-mixin communication: multiple mixins that need to interact with one another have to rely on shared property keys, making them implicitly coupled. With composables, values returned from one composable can be passed into another as arguments, just like normal functions.

For the above reasons, we no longer recommend using mixins in Vue 3. The feature is kept only for migration and familiarity reasons.

### 23.7.2 vs. Renderless Components

In the component slots chapter, we discussed the Renderless Component pattern based on scoped slots. We even implemented the same mouse tracking demo using renderless components.

The main advantage of composables over renderless components is that composables do not incur the extra component instance overhead. When used across an entire application, the amount of extra component instances created by the renderless component pattern can become a noticeable performance overhead.

The recommendation is to use composables when reusing pure logic, and use components when reusing both logic and visual layout.

### 23.7.3 vs. React Hooks

If you have experience with React, you may notice that this looks very similar to custom React hooks. Composition API was in part inspired by React hooks, and Vue composables are indeed similar to React hooks in terms of logic composition capabilities. However, Vue composables are based on Vue's fine-grained reactivity system, which is fundamentally different from React hooks' execution model. This is discussed in more details in the Composition API FAQ.

## 23.8 Further Reading

Reactivity In Depth: for a low-level understanding of how Vue's reactivity system works.

State Management: for patterns of managing state shared by multiple components.

Testing Composables: tips on unit testing composables.

VueUse: an ever-growing collection of Vue composables. The source code is also a great learning resource.

# 第24章 Custom Directives

## 24.1 Introduction

In addition to the default set of directives shipped in core (like v-model or v-show), Vue also allows you to register your own custom directives.

We have introduced two forms of code reuse in Vue: components and composables. Components are the main building blocks, while composables are focused on reusing stateful logic. Custom directives, on the other hand, are mainly intended for reusing logic that involves low-level DOM access on plain elements.

A custom directive is defined as an object containing lifecycle hooks similar to those of a component. The hooks receive the element the directive is bound to. Here is an example of a directive that focuses an input when the element is inserted into the DOM by Vue:

const focus = {

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

}

export default {

directives: {

// enables v-focus in template

focus

}

}

<input v-focus />

This should be focused

Assuming you haven't clicked elsewhere on the page, the input above should be auto-focused. This directive is more useful than the autofocus attribute because it works not just on page load - it also works when the element is dynamically inserted by Vue.

Similar to components, custom directives must be registered so that they can be used in templates. In the example above, we are using local registration via the directives option.

It is also common to globally register custom directives at the app level:

const app = createApp({})

// make v-focus usable in all components

app.directive('focus', {

/\* ... \*/

})

TIP

Custom directives should only be used when the desired functionality can only be achieved via direct DOM manipulation. Prefer declarative templating using built-in directives such as v-bind when possible because they are more efficient and server-rendering friendly.

### 24.1.1 Directive Hooks

A directive definition object can provide several hook functions (all optional):

const myDirective = {

// called before bound element's attributes

// or event listeners are applied

created(el, binding, vnode, prevVnode) {

// see below for details on arguments

},

// called right before the element is inserted into the DOM.

beforeMount(el, binding, vnode, prevVnode) {},

// called when the bound element's parent component

// and all its children are mounted.

mounted(el, binding, vnode, prevVnode) {},

// called before the parent component is updated

beforeUpdate(el, binding, vnode, prevVnode) {},

// called after the parent component and

// all of its children have updated

updated(el, binding, vnode, prevVnode) {},

// called before the parent component is unmounted

beforeUnmount(el, binding, vnode, prevVnode) {},

// called when the parent component is unmounted

unmounted(el, binding, vnode, prevVnode) {}

}

### 24.1.2 Hook Arguments

Directive hooks are passed these arguments:

el: the element the directive is bound to. This can be used to directly manipulate the DOM.

binding: an object containing the following properties.

value: The value passed to the directive. For example in v-my-directive="1 + 1", the value would be 2.

oldValue: The previous value, only available in beforeUpdate and updated. It is available whether or not the value has changed.

arg: The argument passed to the directive, if any. For example in v-my-directive:foo, the arg would be "foo".

modifiers: An object containing modifiers, if any. For example in v-my-directive.foo.bar, the modifiers object would be { foo: true, bar: true }.

instance: The instance of the component where the directive is used.

dir: the directive definition object.

vnode: the underlying VNode representing the bound element.

prevNode: the VNode representing the bound element from the previous render. Only available in the beforeUpdate and updated hooks.

As an example, consider the following directive usage:

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

The binding argument would be an object in the shape of:

{

arg: 'foo',

modifiers: { bar: true },

value: /\* value of `baz` \*/,

oldValue: /\* value of `baz` from previous update \*/

}

Similar to built-in directives, custom directive arguments can be dynamic. For example:

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

Here the directive argument will be reactively updated based on arg property in our component state.

Note

Apart from el, you should treat these arguments as read-only and never modify them. If you need to share information across hooks, it is recommended to do so through element's dataset.

## 24.2 Function Shorthand

It's common for a custom directive to have the same behavior for mounted and updated, with no need for the other hooks. In such cases we can define the directive as a function:

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

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

// this will be called for both `mounted` and `updated`

el.style.color = binding.value

})

## 24.3 Object Literals

If your directive needs multiple values, you can also pass in a JavaScript object literal. Remember, directives can take any valid JavaScript expression.

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

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

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

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

})

## 24.4 Usage on Components

When used on components, custom directives will always apply to a component's root node, similar to Fallthrough Attributes.

<MyComponent v-demo="test" />

<!-- template of MyComponent -->

<div> <!-- v-demo directive will be applied here -->

<span>My component content</span>

</div>

Note that components can potentially have more than one root node. When applied to a multi-root component, a directive will be ignored and a warning will be thrown. Unlike attributes, directives can't be passed to a different element with v-bind="$attrs". In general, it is not recommended to use custom directives on components.

# 第25章 Plugins

## 25.1 Introduction

Plugins are self-contained code that usually add app-level functionality to Vue. This is how we install a plugin:

import { createApp } from 'vue'

const app = createApp({})

app.use(myPlugin, {

/\* optional options \*/

})

A plugin is defined as either an object that exposes an install() method, or simply a function that acts as the install function itself. The install function receives the app instance along with additional options passed to app.use(), if any:

const myPlugin = {

install(app, options) {

// configure the app

}

}

There is no strictly defined scope for a plugin, but common scenarios where plugins are useful include:

Register one or more global components or custom directives with app.component() and app.directive().

Make a resource injectable throughout the app by calling app.provide().

Add some global instance properties or methods by attaching them to app.config.globalProperties.

A library that needs to perform some combination of the above (e.g. vue-router).

## 25.2 Writing a Plugin

In order to better understand how to create your own Vue.js plugins, we will create a very simplified version of a plugin that displays i18n (short for Internationalization) strings.

Let's begin by setting up the plugin object. It is recommended to create it in a separate file and export it, as shown below to keep the logic contained and separate.

// plugins/i18n.js

export default {

install: (app, options) => {

// Plugin code goes here

}

}

We want to create a translation function. This function will receive a dot-delimited key string, which we will use to look up the translated string in the user-provided options. This is the intended usage in templates:

<h1>{{ $translate('greetings.hello') }}</h1>

Since this function should be globally available in all templates, we will make it so by attaching it to app.config.globalProperties in our plugin:

// plugins/i18n.js

export default {

install: (app, options) => {

// inject a globally available $translate() method

app.config.globalProperties.$translate = (key) => {

// retrieve a nested property in `options`

// using `key` as the path

return key.split('.').reduce((o, i) => {

if (o) return o[i]

}, options)

}

}

}

Our $translate function will take a string such as greetings.hello, look inside the user provided configuration and return the translated value.

The object containing the translated keys should be passed to the plugin during installation via additional parameters to app.use():

import i18nPlugin from './plugins/i18n'

app.use(i18nPlugin, {

greetings: {

hello: 'Bonjour!'

}

})

Now, our initial expression $translate('greetings.hello') will be replaced by Bonjour! at runtime.

See also: Augmenting Global Properties

TIP

Use global properties scarcely, since it can quickly become confusing if too many global properties injected by different plugins are used throughout an app.

## 25.3 Provide / Inject with Plugins

Plugins also allow us to use inject to provide a function or attribute to the plugin's users. For example, we can allow the application to have access to the options parameter to be able to use the translations object.

// plugins/i18n.js

export default {

install: (app, options) => {

app.config.globalProperties.$translate = (key) => {

return key.split('.').reduce((o, i) => {

if (o) return o[i]

}, options)

}

app.provide('i18n', options)

}

}

Plugin users will now be able to inject the plugin options into their components using the i18n key:

export default {

inject: ['i18n'],

created() {

console.log(this.i18n.greetings.hello)

}

}

# 第26章 Transition

Vue offers two built-in components that can help work with transitions and animations in response to changing state:

<Transition> for applying animations when an element or component is entering and leaving the DOM. This is covered on this page.

<TransitionGroup> for applying animations when an element or component is inserted into, removed from, or moved within a v-for list. This is covered in the next chapter.

Aside from these two components, we can also apply animations in Vue using other techniques such as toggling CSS classes or state-driven animations via style bindings. These additional techniques are covered in the Animation Techniques chapter.

## 26.1 The <Transition> Component

<Transition> is a built-in component: this means it is available in any component's template without having to register it. It can be used to apply enter and leave animations on elements or components passed to it via its default slot. The enter or leave can be triggered by one of the following:

Conditional rendering via v-if

Conditional display via v-show

Dynamic components toggling via the <component> special element

This is an example of the most basic usage:

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

<Transition>

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

</Transition>

/\* we will explain what these classes do next! \*/

.v-enter-active,

.v-leave-active {

transition: opacity 0.5s ease;

}

.v-enter-from,

.v-leave-to {

opacity: 0;

}

TIP

<Transition> only supports a single element or component as its slot content. If the content is a component, the component must also have only one single root element.

When an element in a <Transition> component is inserted or removed, this is what happens:

Vue will automatically sniff whether the target element has CSS transitions or animations applied. If it does, a number of CSS transition classes will be added / removed at appropriate timings.

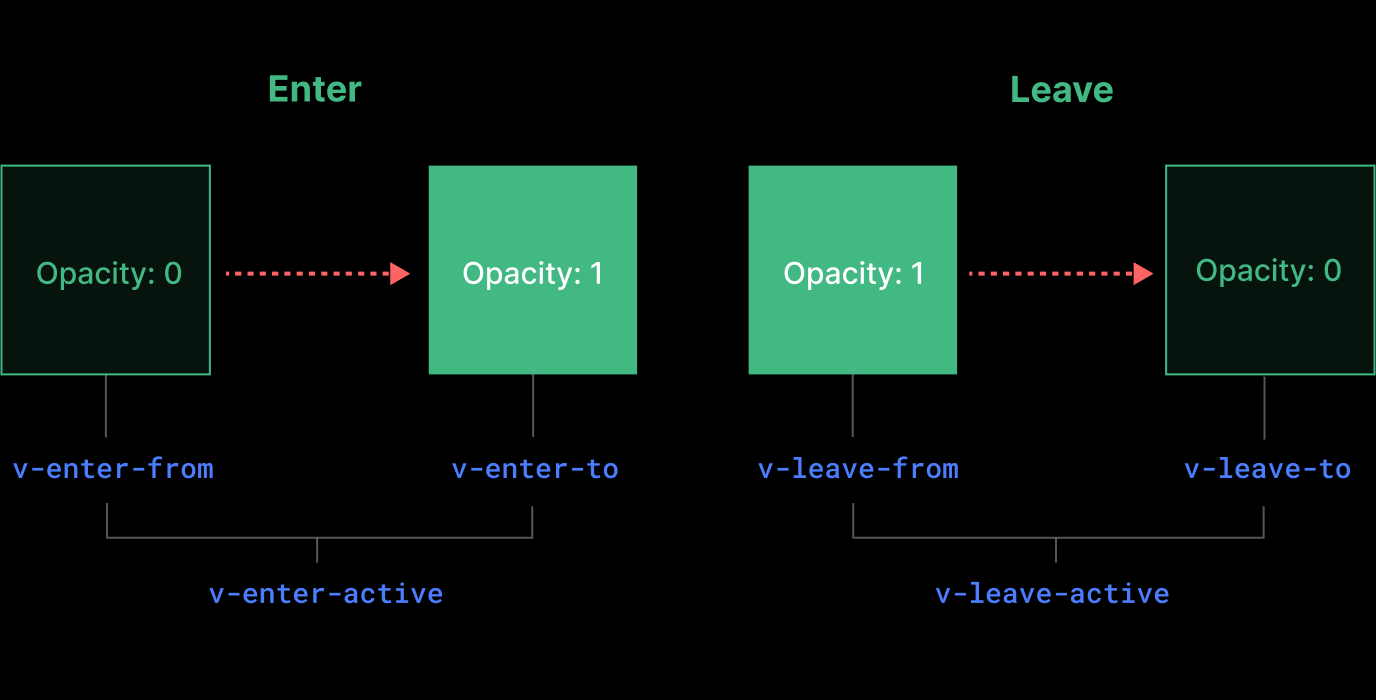
If there are listeners for JavaScript hooks, these hooks will be called at appropriate timings.

If no CSS transitions / animations are detected and no JavaScript hooks are provided, the DOM operations for insertion and/or removal will be executed on the browser's next animation frame.

## 26.2 CSS-Based Transitions

### 26.2.1 Transition Classes

There are six classes applied for enter / leave transitions.



v-enter-from: Starting state for enter. Added before the element is inserted, removed one frame after the element is inserted.

v-enter-active: Active state for enter. Applied during the entire entering phase. Added before the element is inserted, removed when the transition/animation finishes. This class can be used to define the duration, delay and easing curve for the entering transition.

v-enter-to: Ending state for enter. Added one frame after the element is inserted (at the same time v-enter-from is removed), removed when the transition/animation finishes.

v-leave-from: Starting state for leave. Added immediately when a leaving transition is triggered, removed after one frame.

v-leave-active: Active state for leave. Applied during the entire leaving phase. Added immediately when a leave transition is triggered, removed when the transition/animation finishes. This class can be used to define the duration, delay and easing curve for the leaving transition.

v-leave-to: Ending state for leave. Added one frame after a leaving transition is triggered (at the same time v-leave-from is removed), removed when the transition/animation finishes.

v-enter-active and v-leave-active give us the ability to specify different easing curves for enter / leave transitions, which we'll see an example of in the following sections.

### 26.2.2 Named Transitions

A transition can be named via the name prop:

<Transition name="fade">

...

</Transition>

For a named transition, its transition classes will be prefixed with its name instead of v. For example, the applied class for the above transition will be fade-enter-active instead of v-enter-active. The CSS for the fade transition should look like this:

.fade-enter-active,

.fade-leave-active {

transition: opacity 0.5s ease;

}

.fade-enter-from,

.fade-leave-to {

opacity: 0;

}

### 26.2.3 CSS Transitions

<Transition> is most commonly used in combination with native CSS transitions, as seen in the basic example above. The transition CSS property is a shorthand that allows us to specify multiple aspects of a transition, including properties that should be animated, duration of the transition, and easing curves.

Here is a more advanced example that transitions multiple properties, with different durations and easing curves for enter and leave:

<Transition name="slide-fade">

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

</Transition>

/\*

Enter and leave animations can use different

durations and timing functions.

\*/

.slide-fade-enter-active {

transition: all 0.3s ease-out;

}

.slide-fade-leave-active {

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

}

.slide-fade-enter-from,

.slide-fade-leave-to {

transform: translateX(20px);

opacity: 0;

}

### 26.2.4 CSS Animations

Native CSS animations are applied in the same way as CSS transitions, with the difference being that \*-enter-from is not removed immediately after the element is inserted, but on an animationend event.

For most CSS animations, we can simply declare them under the \*-enter-active and \*-leave-active classes. Here's an example:

<Transition name="bounce">

<p v-if="show" style="text-align: center;">

Hello here is some bouncy text!

</p>

</Transition>

.bounce-enter-active {

animation: bounce-in 0.5s;

}

.bounce-leave-active {

animation: bounce-in 0.5s reverse;

}

@keyframes bounce-in {

0% {

transform: scale(0);

}

50% {

transform: scale(1.25);

}

100% {

transform: scale(1);

}

}

### 26.2.5 Custom Transition Classes

You can also specify custom transition classes by passing the following props to <Transition>:

enter-from-class

enter-active-class

enter-to-class

leave-from-class

leave-active-class

leave-to-class

These will override the conventional class names. This is especially useful when you want to combine Vue's transition system with an existing CSS animation library, such as Animate.css:

<!-- assuming Animate.css is included on the page -->

<Transition

name="custom-classes"

enter-active-class="animate\_\_animated animate\_\_tada"

leave-active-class="animate\_\_animated animate\_\_bounceOutRight"

>

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

</Transition>

### 26.2.6 Using Transitions and Animations Together

Vue needs to attach event listeners in order to know when a transition has ended. It can either be transitionend or animationend, depending on the type of CSS rules applied. If you are only using one or the other, Vue can automatically detect the correct type.

However, in some cases you may want to have both on the same element, for example having a CSS animation triggered by Vue, along with a CSS transition effect on hover. In these cases, you will have to explicitly declare the type you want Vue to care about by passing the type prop, with a value of either animation or transition:

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

### 26.2.7 Nested Transitions and Explicit Transition Durations

Although the transition classes are only applied to the direct child element in <Transition>, we can transition nested elements using nested CSS selectors:

<Transition name="nested">

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

<div class="inner">

Hello

</div>

</div>

</Transition>

/\* rules that target nested elements \*/

.nested-enter-active .inner,

.nested-leave-active .inner {

transition: all 0.3s ease-in-out;

}

.nested-enter-from .inner,

.nested-leave-to .inner {

transform: translateX(30px);

opacity: 0;

}

/\* ... other necessary CSS omitted \*/

We can even add a transition delay to the nested element on enter, which creates a staggered enter animation sequence:

/\* delay enter of nested element for staggered effect \*/

.nested-enter-active .inner {

transition-delay: 0.25s;

}

However, this creates a small issue. By default, the <Transition> component attempts to automatically figure out when the transition has finished by listening to the first transitionend or animationend event on the root transition element. With a nested transition, the desired behavior should be waiting until the transitions of all inner elements have finished.

In such cases you can specify an explicit transition duration (in milliseconds) using the duration prop on the <transition> component. The total duration should match the delay plus transition duration of the inner element:

<Transition :duration="550">...</Transition>

If necessary, you can also specify separate values for enter and leave durations using an object:

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

### 26.2.8 Performance Considerations

You may notice that the animations shown above are mostly using properties like transform and opacity. These properties are efficient to animate because:

They do not affect the document layout during the animation, so they do not trigger expensive CSS layout calculation on every animation frame.

Most modern browsers can leverage GPU hardware acceleration when animating transform.

In comparison, properties like height or margin will trigger CSS layout, so they are much more expensive to animate, and should be used with caution. We can check resources like CSS-Triggers to see which properties will trigger layout if we animate them.

## 26.3 JavaScript Hooks

You can hook into the transition process with JavaScript by listening to events on the <Transition> component:

<Transition

@before-enter="onBeforeEnter"

@enter="onEnter"

@after-enter="onAfterEnter"

@enter-cancelled="onEnterCancelled"

@before-leave="onBeforeLeave"

@leave="onLeave"

@after-leave="onAfterLeave"

@leave-cancelled="onLeaveCancelled"

>

<!-- ... -->

</Transition>

export default {

// ...

methods: {

// called before the element is inserted into the DOM.

// use this to set the "enter-from" state of the element

onBeforeEnter(el) {},

// called one frame after the element is inserted.

// use this to start the animation.

onEnter(el, done) {

// call the done callback to indicate transition end

// optional if used in combination with CSS

done()

},

// called when the enter transition has finished.

onAfterEnter(el) {},

onEnterCancelled(el) {},

// called before the leave hook.

// Most of the time, you shoud just use the leave hook.

onBeforeLeave(el) {},

// called when the leave transition starts.

// use this to start the leaving animation.

onLeave(el, done) {

// call the done callback to indicate transition end

// optional if used in combination with CSS

done()

},

// called when the leave transition has finished and the

// element has been removed from the DOM.

onAfterLeave(el) {},

// only available with v-show transitions

onLeaveCancelled(el) {}

}

}

These hooks can be used in combination with CSS transitions / animations or on their own.

When using JavaScript-only transitions, it is usually a good idea to add the :css="false" prop. This explicitly tells Vue to skip auto CSS transition detection. Aside from being slightly more performant, this also prevents CSS rules from accidentally interfering with the transition:

<Transition

...

:css="false"

>

...

</Transition>

With :css="false", we are also fully responsible for controlling when the transition ends. In this case, the done callbacks are required for the @enter and @leave hooks. Otherwise, the hooks will be called synchronously and the transition will finish immediately.

Here's a demo using the GreenSock library to perform the animations. You can, of course, use any other animation library you want, for example Anime.js or Motion One.

## 26.4 Reusable Transitions

Transitions can be reused through Vue's component system. To create a reusable transition, we can create a component that wraps the <Transition> component and passes down the slot content:

<!-- MyTransition.vue -->

<script>

// JavaScript hooks logic...

</script>

<template>

<!-- wrap the built-in Transition component -->

<Transition

name="my-transition"

@enter="onEnter"

@leave="onLeave">

<slot></slot> <!-- pass down slot content -->

</Transition>

</template>

<style>

/\*

Necessary CSS...

Note: avoid using <style scoped> here since it

does not apply to slot content.

\*/

</style>

Now MyTransition can be imported and used just like the built-in version:

<MyTransition>

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

</MyTransition>

## 26.5 Transition on Appear

If you also want to apply a transition on the initial render of a node, you can add the appear prop:

<Transition appear>

...

</Transition>

## 26.6 Transition Between Elements

In addition to toggling an element with v-if / v-show, we can also transition between two elements using v-if / v-else / v-else-if, as long as we make sure that there is only one element being shown at any given moment:

<Transition>

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

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

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

</Transition>

## 26.7 Transition Modes

In the previous example, the entering and leaving elements are animated at the same time, and we had to make them position: absolute to avoid the layout issue when both elements are present in the DOM.

However, in some cases this isn't an option, or simply isn't the desired behavior. We may want the leaving element to be animated out first, and for the entering element to only be inserted after the leaving animation has finished. Orchestrating such animations manually would be very complicated - luckily, we can enable this behavior by passing <Transition> a mode prop:

<Transition mode="out-in">

...

</Transition>

Here's the previous demo with mode="out-in":

<Transition> also supports mode="in-out", although it's much less frequently used.

## 26.8 Transition Between Components

<Transition> can also be used around dynamic components:

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

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

</Transition>

## 26.9 Dynamic Transitions

<Transition> props like name can also be dynamic! It allows us to dynamically apply different transitions based on state change:

<Transition :name="transitionName">

<!-- ... -->

</Transition>

This can be useful when you've defined CSS transitions / animations using Vue's transition class conventions and want to switch between them.

You can also apply different behavior in JavaScript transition hooks based on the current state of your component. Finally, the ultimate way of creating dynamic transitions is through reusable transition components that accept props to change the nature of the transition(s) to be used. It may sound cheesy, but the only limit really is your imagination.

## 26.10 Related

<Transition> API reference

# 第27章 TransitionGroup

<TransitionGroup> is a built-in component designed for animating the insertion, removal, and order change of elements or components that are rendered in a list.

## 27.1 Differences from <Transition>

<TransitionGroup> supports the same props, CSS transition classes, and JavaScript hook listeners as <Transition>, with the following differences:

By default, it doesn't render a wrapper element. But you can specify an element to be rendered with the tag prop.

Transition modes are not available, because we are no longer alternating between mutually exclusive elements.

Elements inside are always required to have a unique key attribute.

CSS transition classes will be applied to individual elements in the list, not to the group / container itself.

TIP

When used in DOM templates, it should be referenced as <transition-group>.

## 27.2 Enter / Leave Transitions

Here is an example of applying enter / leave transitions to a v-for list using <TransitionGroup>:

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

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

{{ item }}

</li>

</TransitionGroup>

.list-enter-active,

.list-leave-active {

transition: all 0.5s ease;

}

.list-enter-from,

.list-leave-to {

opacity: 0;

transform: translateX(30px);

}

## 27.3 Move Transitions

The above demo has some obvious flaws: when an item is inserted or removed, its surrounding items instantly "jump" into place instead of moving smoothly. We can fix this by adding a few additional CSS rules:

.list-move, /\* apply transition to moving elements \*/

.list-enter-active,

.list-leave-active {

transition: all 0.5s ease;

}

.list-enter-from,

.list-leave-to {

opacity: 0;

transform: translateX(30px);

}

/\* ensure leaving items are taken out of layout flow so that moving

animations can be calculated correctly. \*/

.list-leave-active {

position: absolute;

}

## 27.4 Staggering List Transitions

By communicating with JavaScript transitions through data attributes, it's also possible to stagger transitions in a list. First, we render the index of an item as a data attribute on the DOM element:

<TransitionGroup

tag="ul"

:css="false"

@before-enter="onBeforeEnter"

@enter="onEnter"

@leave="onLeave"

>

<li

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

:key="item.msg"

:data-index="index"

>

{{ item.msg }}

</li>

</TransitionGroup>

Then, in JavaScript hooks, we animate the element with a delay based on the data attribute. This example is using the GreenSock library to perform the animation:

function onEnter(el, done) {

gsap.to(el, {

opacity: 1,

height: '1.6em',

delay: el.dataset.index \* 0.15,

onComplete: done

})

}

### 27.5 Related

<TransitionGroup> API reference

# 第28章 KeepAlive

<KeepAlive> is a built-in component that allows us to conditionally cache component instances when dynamically switching between multiple components.

## 28.1 Basic Usage

In the Component Basics chapter, we introduced the syntax for Dynamic Components, using the <component> special element:

<component :is="activeComponent" />

By default, an active component instance will be unmounted when switched away from. This will cause any changed state it holds to be lost. When this component is displayed again, a new instance will be created with only the initial state.

In the example below, we have two stateful components - A contains a counter, while B contains a message synced with an input via v-model. Try updating the state of one of them, switch away, and then switch back to it:

You'll notice that when switched back, the previous changed state would have been reset.

Creating fresh component instance on switch is normally useful behavior, but in this case, we'd really like the two component instances to be preserved even when they are inactive. To solve this problem, we can wrap our dynamic component with the <KeepAlive> built-in component:

<!-- Inactive components will be cached! -->

<KeepAlive>

<component :is="activeComponent" />

</KeepAlive>

Now, the state will be persisted across component switches:

TIP

When used in DOM templates, it should be referenced as <keep-alive>.

## 28.2 Include / Exclude

By default, <KeepAlive> will cache any component instance inside. We can customize this behavior via the include and exclude props. Both props can be a comma-delimited string, a RegExp, or an array containing either types:

<!-- comma-delimited string -->

<KeepAlive include="a,b">

<component :is="view" />

</KeepAlive>

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

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

<component :is="view" />

</KeepAlive>

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

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

<component :is="view" />

</KeepAlive>

The match is checked against the component's name option, so components that need to be conditionally cached by KeepAlive must explicitly declare a name option.

TIP

Since version 3.2.34, a single-file component using <script setup> will automatically infer its name option based on the filename, removing the need to manually declare the name.

## 28.3 Max Cached Instances

We can limit the maximum number of component instances that can be cached via the max prop. When max is specified, <KeepAlive> behaves like an LRU cache: if the number of cached instances is about to exceed the specified max count, the least recently accessed cached instance will be destroyed to make room for the new one.

<KeepAlive :max="10">

<component :is="activeComponent" />

</KeepAlive>

## 28.4 Lifecycle of Cached Instance

When a component instance is removed from the DOM but is part of a component tree cached by <KeepAlive>, it goes into a deactivated state instead of being unmounted. When a component instance is inserted into the DOM as part of a cached tree, it is activated.

A kept-alive component can register lifecycle hooks for these two states using activated and deactivated hooks:

export default {

activated() {

// called on initial mount

// and every time it is re-inserted from the cache

},

deactivated() {

// called when removed from the DOM into the cache

// and also when unmounted

}

}

Note that:

activated is also called on mount, and deactivated on unmount.

Both hooks work for not only the root component cached by <KeepAlive>, but also descendent components in the cached tree.

## 28.5 Related

<KeepAlive> API reference

# 第29 章 Teleport

<Teleport> is a built-in component that allows us to "teleport" a part of a component's template into a DOM node that exists outside the DOM hierarchy of that component.

## 29.1 Basic Usage

Sometimes we may run into the following scenario: a part of a component's template belongs to it logically, but from a visual standpoint, it should be displayed somewhere else in the DOM, outside of the Vue application.

The most common example of this is when building a full-screen modal. Ideally, we want the modal's button and the modal itself to live within the same component, since they are both related to the open / close state of the modal. But that means the modal will be rendered alongside the button, deeply nested in the application's DOM hierarchy. This can create some tricky issues when positioning the modal via CSS.

Consider the following HTML structure.

<div class="outer">

<h3>Vue Teleport Example</h3>

<div>

<MyModal />

</div>

</div>

And here is the implementation of <MyModal>:

<script>

export default {

data() {

return {

open: false

}

}

}

</script>

<template>

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

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

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

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

</div>

</template>

<style scoped>

.modal {

position: fixed;

z-index: 999;

top: 20%;

left: 50%;

width: 300px;

margin-left: -150px;

}

</style>

The component contains a <button> to trigger the opening of the modal, and a <div> with a class of .modal, which will contain the modal's content and a button to self-close.

When using this component inside the initial HTML structure, there are a number of potential issues:

position: fixed only places the element relative to the viewport when no ancestor element has transform, perspective or filter property set. If, for example, we intend to animate the ancestor <div class="outer"> with a CSS transform, it would break the modal layout!

The modal's z-index is constrained by its containing elements. If there is another element that overlaps with <div class="outer"> and has a higher z-index, it would cover our modal.

<Teleport> provides a clean way to work around these, by allowing us to break out of the nested DOM structure. Let's modify <MyModal> to use <Teleport>:

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

<Teleport to="body">

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

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

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

</div>

</Teleport>

The to target of <Teleport> expects a CSS selector string or an actual DOM node. Here, we are essentially telling Vue to "teleport this template fragment to the body tag".

You can click the button below and inspect the <body> tag via your browser's devtools:

You can combine <Teleport> with <Transition> to create animated modals - see Example here.

TIP

The teleport to target must be already in the DOM when the <Teleport> component is mounted. Ideally, this should be an element outside the entire Vue application. If targeting another element rendered by Vue, you need to make sure that element is mounted before the <Teleport>.

## 29.2 Using with Components

<Teleport> only alters the rendered DOM structure - it does not affect the logical hierarchy of the components. That is to say, if <Teleport> contains a component, that component will remain a logical child of the parent component containing the <Teleport>. Props passing and event emitting will continue to work the same way.

This also means that injections from a parent component work as expected, and that the child component will be nested below the parent component in the Vue Devtools, instead of being placed where the actual content moved to.

## 29.3 Disabling Teleport

In some cases, we may want to conditionally disable <Teleport>. For example, we may want to render a component as an overlay for desktop, but inline on mobile. <Teleport> supports the disabled prop which can be dynamically toggled:

<Teleport :disabled="isMobile">

...

</Teleport>

Where the isMobile state can be dynamically updated by detecting media query changes.

## 29.4 Multiple Teleports on the Same Target

A common use case would be a reusable <Modal> component, with the potential for multiple instances to be active at the same time. For this kind of scenario, multiple <Teleport> components can mount their content to the same target element. The order will be a simple append - later mounts will be located after earlier ones within the target element.

Given the following usage:

<Teleport to="#modals">

<div>A</div>

</Teleport>

<Teleport to="#modals">

<div>B</div>

</Teleport>

The rendered result would be:

<div id="modals">

<div>A</div>

<div>B</div>

</div>

## 29.5 Related

<Teleport> API reference

Handling Teleports in SSR

# 第30章 Suspense

Experimental Feature

<Suspense> is an experimental feature. It is not guaranteed to reach stable status and the API may change before it does.

<Suspense> is a built-in component for orchestrating async dependencies in a component tree. It can render a loading state while waiting for multiple nested async dependencies down the component tree to be resolved.

## 30.1 Async Dependencies

To explain the problem <Suspense> is trying to solve and how it interacts with these async dependencies, let's imagine a component hierarchy like the following:

<Suspense>

└─ <Dashboard>

├─ <Profile>

│ └─ <FriendStatus> (component with async setup())

└─ <Content>

├─ <ActivityFeed> (async component)

└─ <Stats> (async component)

In the component tree there are multiple nested components whose rendering depends on some async resource to be resolved first. Without <Suspense>, each of them will need to handle its own loading / error and loaded states. In the worst case scenario, we may see three loading spinners on the page, with content displayed at different times.

The <Suspense> component gives us the ability to display top-level loading / error states while we wait on these nested async dependencies to be resolved.

There are two types of async dependencies that <Suspense> can wait on:

Components with an async setup() hook. This includes components using <script setup> with top-level await expressions.

Async Components.

### 30.1.1 async setup()

A Composition API component's setup() hook can be async:

export default {

async setup() {

const res = await fetch(...)

const posts = await res.json()

return {

posts

}

}

}

If using <script setup>, the presence of top-level await expressions automatically makes the component an async dependency:

<script setup>

const res = await fetch(...)

const posts = await res.json()

</script>

<template>

{{ posts }}

</template>

### 30.1.2 Async Components

Async components are "suspensible" by default. This means that if it has a <Suspense> in the parent chain, it will be treated as an async dependency of that <Suspense>. In this case, the loading state will be controlled by the <Suspense>, and the component's own loading, error, delay and timeout options will be ignored.

The async component can opt-out of Suspense control and let the component always control its own loading state by specifying suspensible: false in its options.

## 30.2 Loading State

The <Suspense> component has two slots: #default and #fallback. Both slots only allow for one immediate child node. The node in the default slot is shown if possible. If not, the node in the fallback slot will be shown instead.

<Suspense>

<!-- component with nested async dependencies -->

<Dashboard />

<!-- loading state via #fallback slot -->

<template #fallback>

Loading...

</template>

</Suspense>

On initial render, <Suspense> will render its default slot content in memory. If any async dependencies are encountered during the process, it will enter a pending state. During the pending state, the fallback content will be displayed. When all encountered async dependencies have been resolved, <Suspense> enters a resolved state and the resolved default slot content is displayed.

If no async dependencies were encountered during the initial render, <Suspense> will directly go into a resolved state.

Once in a resolved state, <Suspense> will only revert to a pending state if the root node of the #default slot is replaced. New async dependencies nested deeper in the tree will not cause the <Suspense> to revert to a pending state.

When a revert happens, fallback content will not be immediately displayed. Instead, <Suspense> will display the previous #default content while waiting for the new content and its async dependencies to be resolved. This behavior can be configured with the timeout prop: <Suspense> will switch to fallback content if it takes longer than timeout to render the new default content. A timeout value of 0 will cause the fallback content to be displayed immediately when default content is replaced.

## 30.3 Events

The <Suspense> component emits 3 events: pending, resolve and fallback. The pending event occurs when entering a pending state. The resolve event is emitted when new content has finished resolving in the default slot. The fallback event is fired when the contents of the fallback slot are shown.

The events could be used, for example, to show a loading indicator in front of the old DOM while new components are loading.

## 30.4 Error Handling

<Suspense> currently does not provide error handling via the component itself - however, you can use the errorCaptured option or the onErrorCaptured() hook to capture and handle async errors in the parent component of <Suspense>.

## 30.5 Combining with Other Components

It is common to want to use <Suspense> in combination with the <Transition> and <KeepAlive> components. The nesting order of these components is important to get them all working correctly.

In addition, these components are often used in conjunction with the <RouterView> component from Vue Router.

The following example shows how to nest these components so that they all behave as expected. For simpler combinations you can remove the components that you don't need:

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

<template v-if="Component">

<Transition mode="out-in">

<KeepAlive>

<Suspense>

<!-- main content -->

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

<!-- loading state -->

<template #fallback>

Loading...

</template>

</Suspense>

</KeepAlive>

</Transition>

</template>

</RouterView>

Vue Router has built-in support for lazily loading components using dynamic imports. These are distinct from async components and currently they will not trigger <Suspense>. However, they can still have async components as descendants and those can trigger <Suspense> in the usual way.

# 第31章 Single-File Components

## 31.1 Introduction

Vue Single-File Components (a.k.a. \*.vue files, abbreviated as SFC) is a special file format that allows us to encapsulate the template, logic, and styling of a Vue component in a single file. Here's an example SFC:

<script>

export default {

data() {

return {

greeting: 'Hello World!'

}

}

}

</script>

<template>

<p class="greeting">{{ greeting }}</p>

</template>

<style>

.greeting {

color: red;

font-weight: bold;

}

</style>

As we can see, Vue SFC is a natural extension of the classic trio of HTML, CSS and JavaScript. The <template>, <script>, and <style> blocks encapsulate and colocate the view, logic and styling of a component in the same file. The full syntax is defined in the SFC Syntax Specification.

## 31.2 Why SFC

While SFCs require a build step, there are numerous benefits in return:

Author modularized components using familiar HTML, CSS and JavaScript syntax

Colocation of inherently coupled concerns

Pre-compiled templates without runtime compilation cost

Component-scoped CSS

More ergonomic syntax when working with Composition API

More compile-time optimizations by cross-analyzing template and script

IDE support with auto-completion and type-checking for template expressions

Out-of-the-box Hot-Module Replacement (HMR) support

SFC is a defining feature of Vue as a framework, and is the recommended approach for using Vue in the following scenarios:

Single-Page Applications (SPA)

Static Site Generation (SSG)

Any non-trivial frontend where a build step can be justified for better development experience (DX).

That said, we do realize there are scenarios where SFCs can feel like overkill. This is why Vue can still be used via plain JavaScript without a build step. If you are just looking for enhancing largely static HTML with light interactions, you can also check out petite-vue, a 6 kB subset of Vue optimized for progressive enhancement.

## 31.3 How It Works

Vue SFC is a framework-specific file format and must be pre-compiled by @vue/compiler-sfc into standard JavaScript and CSS. A compiled SFC is a standard JavaScript (ES) module - which means with proper build setup you can import an SFC like a module:

import MyComponent from './MyComponent.vue'

export default {

components: {

MyComponent

}

}

<style> tags inside SFCs are typically injected as native <style> tags during development to support hot updates. For production they can be extracted and merged into a single CSS file.

In actual projects, we typically integrate the SFC compiler with a build tool such as Vite or Vue CLI (which is based on webpack), and Vue provides official scaffolding tools to get you started with SFCs as fast as possible. Check out more details in the SFC Tooling section.

## 31.4 What About Separation of Concerns?

Some users coming from a traditional web development background may have the concern that SFCs are mixing different concerns in the same place - which HTML/CSS/JS were supposed to separate!

To answer this question, it is important for us to agree that separation of concerns is not equal to the separation of file types. The ultimate goal of engineering principles is to improve the maintainability of codebases. Separation of concerns, when applied dogmatically as separation of file types, does not help us reach that goal in the context of increasingly complex frontend applications.

In modern UI development, we have found that instead of dividing the codebase into three huge layers that interweave with one another, it makes much more sense to divide them into loosely-coupled components and compose them. Inside a component, its template, logic, and styles are inherently coupled, and colocating them actually makes the component more cohesive and maintainable.

Note even if you don't like the idea of Single-File Components, you can still leverage its hot-reloading and pre-compilation features by separating your JavaScript and CSS into separate files using Src Imports.

# 第32章 Tooling

## 32.1 Try It Online

You don't need to install anything on your machine to try out Vue SFCs - there are online playgrounds that allow you to do so right in the browser:

Always deployed from latest commit

Designed for inspecting component compilation results

Vue + Vite on StackBlitz

IDE-like environment running actual Vite dev server in the browser

Closest to local setup

It is also recommended to use these online playgrounds to provide reproductions when reporting bugs.

## 32.2 Project Scaffolding

### 32.2.1 Vite

Vite is a lightweight and fast build tool with first-class Vue SFC support. It is created by Evan You, who is also the author of Vue!

To get started with Vite + Vue, simply run:

$ npm init vue@latest

This command will install and execute create-vue, the official Vue project scaffolding tool.

To learn more about Vite, check out the Vite docs.

To configure Vue-specific behavior in a Vite project, for example passing options to the Vue compiler, check out the docs for @vitejs/plugin-vue.

Both online playgrounds mentioned above also support downloading files as a Vite project.

### 32.2.2 Vue CLI

Vue CLI is the official webpack-based toolchain for Vue. It is now in maintenance mode and we recommend starting new projects with Vite unless you rely on specific webpack-only features. Vite will provide superior developer experience in most cases.

For information on migrating from Vue CLI to Vite:

Vue CLI -> Vite Migration Guide from VueSchool.io

Tools / Plugins that help with auto migration

### 32.2.3 Note on In-Browser Template Compilation

When using Vue without a build step, component templates are written either directly in the page's HTML or as inlined JavaScript strings. In such cases, Vue needs to ship the template compiler to the browser in order to perform on-the-fly template compilation. On the other hand, the compiler would be unnecessary if we pre-compile the templates with a build step. To reduce client bundle size, Vue provides different "builds" optimized for different use cases.

Build files that start with vue.runtime.\* are runtime-only builds: they do not include the compiler. When using these builds, all templates must be pre-compiled via a build step.

Build files that do not include .runtime are full builds: they include the compiler and support compiling templates directly in the browser. However, they will increase the payload by ~14kb.

Our default tooling setups use the runtime-only build since all templates in SFCs are pre-compiled. If, for some reason, you need in-browser template compilation even with a build step, you can do so by configuring the build tool to alias vue to vue/dist/vue.esm-bundler.js instead.

If you are looking for a lighter-weight alternative for no-build-step usage, check out petite-vue.

## 32.3 IDE Support

The recommended IDE setup is VSCode + the Volar extension. Volar provides syntax highlighting, TypeScript support, and intellisense for template expressions and component props.

TIP

Volar replaces Vetur, our previous official VSCode extension for Vue 2. If you have Vetur currently installed, make sure to disable it in Vue 3 projects.

WebStorm also provides great built-in support for Vue SFCs.

Other IDEs that support the Language Service Protocol (LSP) can also leverage Volar's core functionalities via LSP:

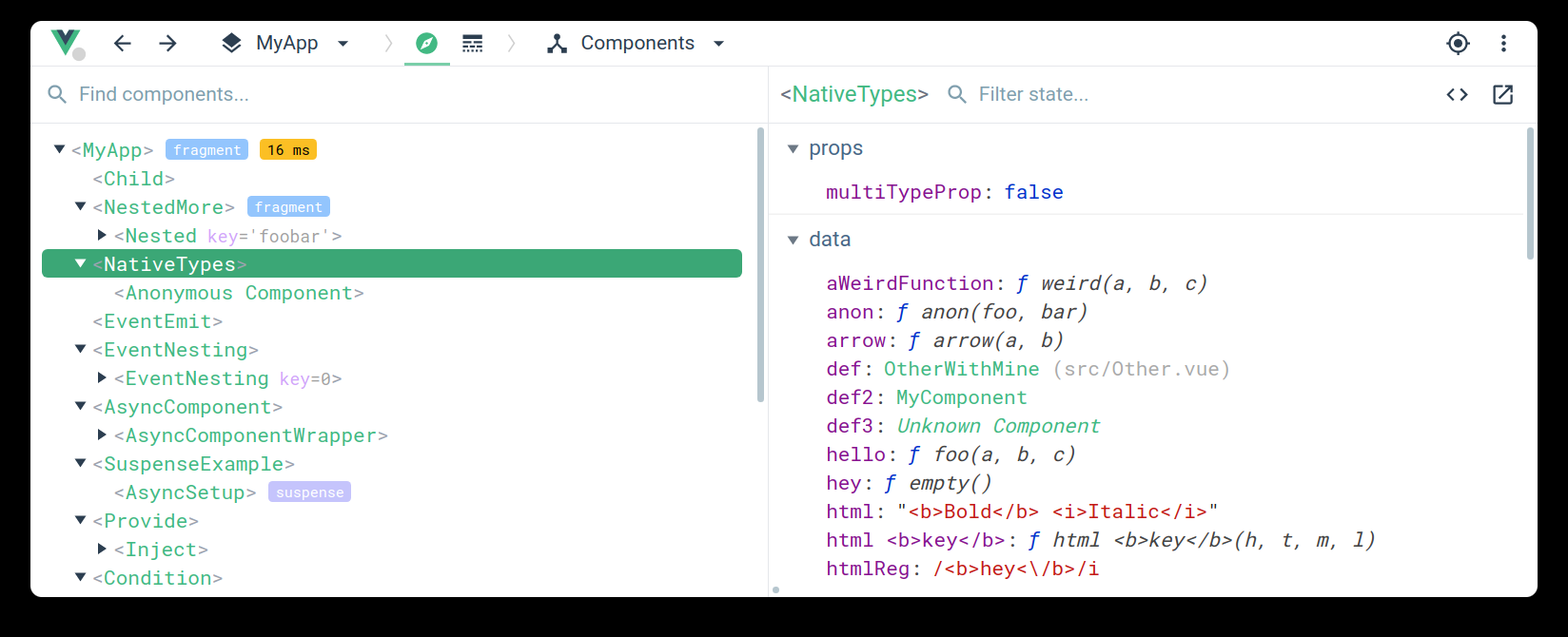
Sublime Text support via LSP-Volar.

vim / Neovim support via coc-volar.

emacs support via lsp-mode

## 32.4 Browser Devtools

The Vue browser devtools extension allows you to explore a Vue app's component tree, inspect the state of individual components, track state management events, and profile performance.



Documentation

Chrome Extension

Firefox Addon

Standalone Electron app

## 32.5 TypeScript

Main article: Using Vue with TypeScript.

Volar provides type checking for SFCs using <script lang="ts"> blocks, including template expressions and cross-component props validation.

Use vue-tsc for performing the same type checking from the command line, or for generating d.ts files for SFCs.

## 32.6 Testing

Main article: Testing Guide.

Cypress is recommended for E2E tests. It can also be used for component testing for Vue SFCs via the Cypress Component Test Runner.

Vitest is a test runner created by Vue / Vite team members that focuses on speed. It is specifically designed for Vite-based applications to provide the same instant feedback loop for unit / component testing.

Jest can be made to work with Vite via vite-jest. However, this is only recommended if you have existing Jest-based test suites that you need to migrate over to a Vite-based setup, as Vitest provides similar functionalities with a much more efficient integration.

## 32.7 Linting

The Vue team maintains eslint-plugin-vue, an ESLint plugin that supports SFC-specific linting rules.

Users previously using Vue CLI may be used to having linters configured via webpack loaders. However when using a Vite-based build setup, our general recommendation is:

npm install -D eslint eslint-plugin-vue, then follow eslint-plugin-vue's configuration guide.

Setup ESLint IDE extensions, for example ESLint for VSCode, so you get linter feedback right in your editor during development. This also avoids unnecessary linting cost when starting the dev server.

Run ESLint as part of the production build command, so you get full linter feedback before shipping to production.

(Optional) Setup tools like lint-staged to automatically lint modified files on git commit.

## 32.8 Formatting

The Volar VSCode extension provides formatting for Vue SFCs out of the box.

Alternatively, Prettier provides built-in Vue SFC formatting support.

## 32.9 SFC Custom Block Integrations

Custom blocks are compiled into imports to the same Vue file with different request queries. It is up to the underlying build tool to handle these import requests.

If using Vite, a custom Vite plugin should be used to transform matched custom blocks into executable JavaScript. Example

If using Vue CLI or plain webpack, a webpack loader should be configured to transform the matched blocks. Example

## 32.10 Lower-Level Packages

### 32.10.1 @vue/compiler-sfc

**Docs https://github.com/vuejs/core/tree/main/packages/compiler-sfc**

This package is part of the Vue core monorepo and is always published with the same version as the main vue package. It is included as a dependency of the main vue package and proxied under vue/compiler-sfc so you don't need to install it individually.

The package itself provides lower-level utilities for processing Vue SFCs and is only meant for tooling authors that need to support Vue SFCs in custom tools.

TIP

Always prefer using this package via the vue/compiler-sfc deep import since this ensures its version is in sync with the Vue runtime.

@vitejs/plugin-vue#

**Docs https://github.com/vitejs/vite/tree/main/packages/plugin-vue**

Official plugin that provides Vue SFC support in Vite.

vue-loader#

**Docs https://vue-loader.vuejs.org/**

The official loader that provides Vue SFC support in webpack. If you are using Vue CLI, also see docs on modifying vue-loader options in Vue CLI.

## 32.11 Other Online Playgrounds

VueUse Playground https://play.vueuse.org/

Vue + Vite on Repl.it <https://replit.com/@replit/VueJS>

Vue on CodeSandbox

Vue on Codepen

Vue on Components.studio

Vue on WebComponents.dev

# 第33 章 Routing

## 33.1 Client-Side vs. Server-Side Routing

Routing on the server side means the server sending a response based on the URL path that the user is visiting. When we click on a link in a traditional server-rendered web app, the browser receives an HTML response from the server and reloads the entire page with the new HTML.

In a Single-Page Application (SPA), however, the client-side JavaScript can intercept the navigation, dynamically fetch new data, and update the current page without full page reloads. This typically results in a more snappy user experience, especially for use cases that are more like actual "applications", where the user is expected to perform many interactions over a long period of time.

In such SPAs, the "routing" is done on the client side, in the browser. A client-side router is responsible for managing the application's rendered view using browser APIs such as History API or the hashchange event.