# 第1章 介绍

Vue Router 是Vue.js的官方路由器。它与 Vue.js 核心深度集成，使使用 Vue.js 构建单页应用程序变得轻而易举。特点包括：

嵌套路由映射

动态路由

模块化、基于组件的路由器配置

路由参数、查询、通配符

查看由 Vue.js 的过渡系统提供支持的过渡效果

细粒度的导航控制

具有自动活动 CSS 类的链接

HTML5 历史模式或哈希模式

可定制的滚动行为

URL 的正确编码

开始或玩游乐场（请参阅README.md运行它们）。

# 第2章 安装

## 2.1 直接下载/CDN

https://unpkg.com/vue-router@4

Unpkg.com基于 npm 的 CDN。上面的链接还将提供指向 npm 上的最新版本。您可以通过其他版本https://unpkg.com/vue-router@4.0.15/dist/vue-router.global.js。

## 2.2 Npm

npm install vue-router@4

## 2.3 yarn

yarn add vue-router@4

**注意：**

<https://github.com/vuejs/router/edit/main/packages/docs/installation.md>

# 第3章 入门

使用 Vue + Vue Router 创建单页应用程序：使用 Vue.js，我们已经在使用我们自然的应用程序。将 Vue Router 添加到中的时候，我们需要做的就是将我们的组件组合映射到路由并让 Vue Router 知道它们在哪里渲染。这是一个基本示例：

## 3.1 HTML

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

<script src="https://unpkg.com/vue-router@4"></script>

<div id="app">

<h1>Hello App!</h1>

<p>

<!-- use the router-link component for navigation. -->

<!-- specify the link by passing the `to` prop. -->

<!-- `<router-link>` will render an `<a>` tag with the correct `href` attribute -->

<router-link to="/">Go to Home</router-link>

<router-link to="/about">Go to About</router-link>

</p>

<!-- route outlet -->

<!-- component matched by the route will render here -->

<router-view></router-view>

</div>

## 3.1.1 router-link

请注意，我们如何使用自定义组件来创建链接，而不是经常使用的a标签。router-link这允许 Vue Router 在不重新加载页面的情况下更改 URL，如何处理 URL 生成和编码。稍后我们将看到从这些功能中圆满。

## 3.1.2 router-view

router-view将与 url 的组件。你可以把它显示在任何地方以适应你的结构。

## 3.2 JavaScript

// 1. Define route components.

// These can be imported from other files

const Home = { template: '<div>Home</div>' }

const About = { template: '<div>About</div>' }

// 2. Define some routes

// Each route should map to a component.

// We'll talk about nested routes later.

const routes = [

{ path: '/', component: Home },

{ path: '/about', component: About },

]

// 3. Create the router instance and pass the `routes` option

// You can pass in additional options here, but let's

// keep it simple for now.

const router = VueRouter.createRouter({

// 4. Provide the history implementation to use. We are using the hash history for simplicity here.

history: VueRouter.createWebHashHistory(),

routes, // short for `routes: routes`

})

// 5. Create and mount the root instance.

const app = Vue.createApp({})

// Make sure to \_use\_ the router instance to make the

// whole app router-aware.

app.use(router)

app.mount('#app')

// Now the app has started!

通过调用app.use(router)，我们可以访问它this.$router以及this.$route任何组件内部的当前路由：

// Home.vue

export default {

computed: {

username() {

// We will see what `params` is shortly

return this.$route.params.username

},

},

methods: {

goToDashboard() {

if (isAuthenticated) {

this.$router.push('/dashboard')

} else {

this.$router.push('/login')

}

},

},

}

要访问路由器或setup事务内部的路由，请调用useRouter或useRoute函数。我们将在 Composition API中了解更多相关信息

在整个文档中，我们将直接使用router实例。请记住，这与使用创建的实例是因为我们使用的原因是因为不想在路由器中操作路径的this.$router组件。routercreateRouterthis.$router

# 第4章 带参数的动态路由匹配

很多时候，我们需要将具有给定模式的路由映射到同一个组件。例如，我们可能有一个User组件应该为所有用户呈现，但具有不同的用户 ID。在 Vue Router 中，我们可以在路径中使用动态段来实现这一点，我们称之为param：

const User = {

template: '<div>User</div>',

}

// these are passed to `createRouter`

const routes = [

// dynamic segments start with a colon

{ path: '/users/:id', component: User },

]

现在 URL 喜欢/users/johnny并且/users/jolyne都将映射到相同的路由。

参数用冒号:表示。当路由匹配时，其参数的值将像this.$route.params在每个组件中一样公开。User因此，我们可以通过更新's 模板来呈现当前用户 ID ：

const User = {

template: '<div>User {{ $route.params.id }}</div>',

}

您可以在同一条路由中有多个参数，它们将映射到$route.params. 例子：

图案 匹配路径 $route.params

/users/:用户名 /用户/爱德华多 { username: 'eduardo' }

/users/:用户名/posts/:postId / users/eduardo/posts/123 { username: 'eduardo', postId: '123' }

除了$route.params，该$route对象还公开了其他有用的信息，例如$route.query（如果 URL 中有查询）等。您可以在API 参考$route.hash中查看完整的详细信息。

可以在此处找到此示例的工作演示。

## 4.1 对参数更改做出反应

使用带参数的路由时要注意的一件事是，当用户从/users/johnnyto导航时/users/jolyne，将重用相同的组件实例。由于两条路由都渲染相同的组件，因此这比销毁旧实例然后创建新实例更有效。但是，这也意味着不会调用组件的生命周期钩子。

要对同一组件中的参数更改做出反应，您可以简单地观察$route对象上的任何内容，在这种情况下，$route.params：

const User = {

template: '...',

created() {

this.$watch(

() => this.$route.params,

(toParams, previousParams) => {

// react to route changes...

}

)

},

}

或者，使用beforeRouteUpdate 导航守卫，它也允许取消导航：

const User = {

template: '...',

async beforeRouteUpdate(to, from) {

// react to route changes...

this.userData = await fetchUser(to.params.id)

},

}

## 4.2 Catch all / 404 Not found Route

常规参数只会匹配 url 片段之间的字符，以 . 分隔/。如果我们想匹配任何东西，我们可以通过在参数后面的括号内添加正则表达式来使用自定义参数正则表达式：

const routes = [

// will match everything and put it under `$route.params.pathMatch`

{ path: '/:pathMatch(.\*)\*', name: 'NotFound', component: NotFound },

// will match anything starting with `/user-` and put it under `$route.params.afterUser`

{ path: '/user-:afterUser(.\*)', component: UserGeneric },

]

在这个特定场景中，我们在括号之间使用自定义正则表达式pathMatch并将参数标记为可选可重复的。如果需要，这使我们可以通过将其拆分path为数组来直接导航到路线：

this.$router.push({

name: 'NotFound',

// preserve current path and remove the first char to avoid the target URL starting with `//`

params: { pathMatch: this.$route.path.substring(1).split('/') },

// preserve existing query and hash if any

query: this.$route.query,

hash: this.$route.hash,

})

在重复参数部分中查看更多信息。

如果您使用的是历史模式，请确保按照说明正确配置您的服务器。

## 4.3 高级匹配模式

Vue Router 使用自己的路径匹配语法，灵感来自于 使用的语法，express因此它支持许多高级匹配模式，例如可选参数、零个或多个/一个或多个要求，甚至自定义正则表达式模式。请查看高级匹配文档以探索它们。

# 第5章 路由的匹配语法

大多数应用程序将使用像我们刚刚在动态路由匹配中看到的那样的静态路由/about和动态路由，但是 Vue Router 可以提供更多！/users/:userId

为简单起见，所有路由记录都省略了component属性以专注于path值。

## 5.1 参数中的自定义正则表达式

在定义类似的参数时:userId，我们在内部使用以下正则表达式([^/]+)（至少一个不是斜杠的字符/）从 URL 中提取参数。除非您需要根据参数内容区分两条路线，否则这很有效。想象一下两条路由/:orderId和/:productName，它们都会匹配完全相同的 URL，所以我们需要一种方法来区分它们。最简单的方法是将静态部分添加到区分它们的路径中：

const routes = [

// matches /o/3549

{ path: '/o/:orderId' },

// matches /p/books

{ path: '/p/:productName' },

]

但在某些情况下，我们不想添加那个静态部分/o/ p。然而，orderId总是一个数字，而productName可以是任何东西，所以我们可以为括号中的参数指定一个自定义正则表达式：

const routes = [

// /:orderId -> matches only numbers

{ path: '/:orderId(\\d+)' },

// /:productName -> matches anything else

{ path: '/:productName' },

]

现在，去/25会匹配/:orderId，而去其他任何东西都会匹配/:productName。数组的顺序routes甚至无关紧要！

确保像我们对(becomes ) 所做的那样转义反斜杠 ( \)，以便在 JavaScript 中实际传递字符串中的反斜杠字符。\d\\d

## 5.2 可重复参数

如果您需要将路由与多个部分（如）匹配/first/second/third，则应使用\*（0 或更多）和+（1 或更多）将参数标记为可重复：

const routes = [

// /:chapters -> matches /one, /one/two, /one/two/three, etc

{ path: '/:chapters+' },

// /:chapters -> matches /, /one, /one/two, /one/two/three, etc

{ path: '/:chapters\*' },

]

这将为您提供一个参数数组而不是字符串，并且还需要您在使用命名路由时传递一个数组：

// given { path: '/:chapters\*', name: 'chapters' },

router.resolve({ name: 'chapters', params: { chapters: [] } }).href

// produces /

router.resolve({ name: 'chapters', params: { chapters: ['a', 'b'] } }).href

// produces /a/b

// given { path: '/:chapters+', name: 'chapters' },

router.resolve({ name: 'chapters', params: { chapters: [] } }).href

// throws an Error because `chapters` is empty

这些也可以通过在右括号后添加它们与自定义正则表达式组合：

const routes = [

// only match numbers

// matches /1, /1/2, etc

{ path: '/:chapters(\\d+)+' },

// matches /, /1, /1/2, etc

{ path: '/:chapters(\\d+)\*' },

]

## 5.3 敏感和严格的路线选择

默认情况下，所有路由都不区分大小写，并且匹配带有或不带有斜杠的路由。例如，路线/users匹配/users, /users/，甚至/Users/。strict可以使用and选项配置此行为sensitive，它们可以在路由器和路由级别进行设置：

const router = createRouter({

history: createWebHistory(),

routes: [

// will match /users/posva but not:

// - /users/posva/ because of strict: true

// - /Users/posva because of sensitive: true

{ path: '/users/:id', sensitive: true },

// will match /users, /Users, and /users/42 but not /users/ or /users/42/

{ path: '/users/:id?' },

],

strict: true, // applies to all routes

})

## 5.4 可选参数

?您还可以使用修饰符（0 或 1）将参数标记为可选：

const routes = [

// will match /users and /users/posva

{ path: '/users/:userId?' },

// will match /users and /users/42

{ path: '/users/:userId(\\d+)?' },

]

请注意，\*技术上也将参数标记为可选，但?参数不能重复。

## 5.5 调试

如果您需要深入了解您的路线如何转换为正则表达式以了解路线未匹配的原因，或者要报告错误，您可以使用路径排名工具。它支持通过 URL 共享您的路线。

# 第6章 嵌套路由

一些应用程序的 UI 由嵌套多个级别的组件组成。在这种情况下，很常见的是 URL 的段对应于嵌套组件的某种结构，例如：

/user/johnny/profile /user/johnny/posts

+------------------+ +-----------------+

| User | | User |

| +--------------+ | | +-------------+ |

| | Profile | | +------------> | | Posts | |

| | | | | | | |

| +--------------+ | | +-------------+ |

+------------------+ +-----------------+

使用 Vue Router，您可以使用嵌套路由配置来表达这种关系。

鉴于我们在上一章中创建的应用程序：

<div id="app">

<router-view></router-view>

</div>

const User = {

template: '<div>User {{ $route.params.id }}</div>',

}

// these are passed to `createRouter`

const routes = [{ path: '/user/:id', component: User }]

这<router-view>是一个顶级的router-view. 它呈现与顶级路由匹配的组件。类似地，渲染组件也可以包含它自己的嵌套<router-view>. 例如，如果我们在User组件的模板中添加一个：

const User = {

template: `

<div class="user">

<h2>User {{ $route.params.id }}</h2>

<router-view></router-view>

</div>

`,

}

要将组件渲染到这个 nestedrouter-view中，我们需要children在任何路由中使用该选项：

const routes = [

{

path: '/user/:id',

component: User,

children: [

{

// UserProfile will be rendered inside User's <router-view>

// when /user/:id/profile is matched

path: 'profile',

component: UserProfile,

},

{

// UserPosts will be rendered inside User's <router-view>

// when /user/:id/posts is matched

path: 'posts',

component: UserPosts,

},

],

},

]

请注意，以 开头的嵌套路径/将被视为根路径。这允许您利用组件嵌套，而无需使用嵌套 URL。

正如您所看到的，该children选项只是像routes它自己一样的另一个路由数组。因此，您可以根据需要尽可能多地保留嵌套视图。

此时，通过上面的配置，当你访问's/user/eduardo内部时不会渲染任何内容，因为没有匹配到嵌套路由。也许你确实想在那里渲染一些东西。在这种情况下，您可以提供一个空的嵌套路径：Userrouter-view

const routes = [

{

path: '/user/:id',

component: User,

children: [

// UserHome will be rendered inside User's <router-view>

// when /user/:id is matched

{ path: '', component: UserHome },

// ...other sub routes

],

},

]

可以在此处找到此示例的工作演示。

## 6.1 嵌套命名路由

在处理命名路由时，您通常命名子路由：

const routes = [

{

path: '/user/:id',

component: User,

// notice how only the child route has a name

children: [{ path: '', name: 'user', component: UserHome }],

},

]

这将确保导航到/user/:id将始终显示嵌套路线。

在某些情况下，您可能希望导航到命名路由而不导航到嵌套路由。例如，如果您想导航到/user/:id而不显示嵌套路线。在这种情况下，您也可以命名父路由，但请注意，重新加载页面将始终显示嵌套的子路由，因为它被视为路径导航/users/:id而不是命名路由：

const routes = [

{

path: '/user/:id',

name: 'user-parent'

component: User,

children: [{ path: '', name: 'user', component: UserHome }],

},

]

# 第7章 程序化导航

除了使用<router-link>为声明式导航创建锚标记外，我们还可以使用路由器的实例方法以编程方式完成此操作。

## 7.1 导航到其他位置

注意：在 Vue 实例内部，您可以以$router. 因此，您可以调用this.$router.push。

要导航到不同的 URL，请使用router.push. 此方法将新条目推送到历史堆栈中，因此当用户单击浏览器后退按钮时，他们将被带到上一个 URL。

这是单击 a 时内部调用的方法<router-link>，因此单击<router-link :to="...">相当于调用router.push(...)。

声明式 程序化

<router-link :to="..."> router.push(...)

参数可以是字符串路径或位置描述符对象。例子：

// literal string path

router.push('/users/eduardo')

// object with path

router.push({ path: '/users/eduardo' })

// named route with params to let the router build the url

router.push({ name: 'user', params: { username: 'eduardo' } })

// with query, resulting in /register?plan=private

router.push({ path: '/register', query: { plan: 'private' } })

// with hash, resulting in /about#team

router.push({ path: '/about', hash: '#team' })

注意:params如果提供了 a 则被忽略path，而 , 则不是这种情况query，如上面的示例所示。相反，您需要提供路由或使用任何参数name手动指定整体：path

const username = 'eduardo'

// we can manually build the url but we will have to handle the encoding ourselves

router.push(`/user/${username}`) // -> /user/eduardo

// same as

router.push({ path: `/user/${username}` }) // -> /user/eduardo

// if possible use `name` and `params` to benefit from automatic URL encoding

router.push({ name: 'user', params: { username } }) // -> /user/eduardo

// `params` cannot be used alongside `path`

router.push({ path: '/user', params: { username } }) // -> /user

指定 时params，请确保提供stringor number（或这些数组用于可重复的 params）。任何其他类型（如undefined,false等）都将被自动字符串化。对于可选参数，您可以提供一个空字符串 ( "") 作为跳过它的值。

由于 propto接受与 相同类型的对象router.push，因此完全相同的规则适用于它们。

router.push所有其他导航方法都返回一个Promise，它允许我们等到导航完成并知道它是成功还是失败。我们将在导航处理中详细讨论这一点。

## 7.2 替换当前位置

它的行为就像router.push，唯一的区别是它在导航时不推送新的历史条目，正如它的名字所暗示的那样 - 它替换了当前条目。

声明式 程序化

<router-link :to="..." replace> router.replace(...)

也可以直接将属性添加replace: true到routeLocation传递给router.push：

router.push({ path: '/home', replace: true })

// equivalent to

router.replace({ path: '/home' })

## 7.3 遍历历史

此方法采用单个整数作为参数，指示在历史堆栈中前进或后退的步数，类似于window.history.go(n).

例子

// go forward by one record, the same as router.forward()

router.go(1)

// go back by one record, the same as router.back()

router.go(-1)

// go forward by 3 records

router.go(3)

// fails silently if there aren't that many records

router.go(-100)

router.go(100)

## 7.4 历史操纵

您可能已经注意到router.push和是router.replace和router.go的对应物，window.history.pushState它们window.history.replaceStatewindow.history.go确实模仿了window.historyAPI。

因此，如果您已经熟悉Browser History APIs，那么在使用 Vue Router 时操作历史记录会感觉很熟悉。

值得一提的是，无论在创建路由器实例时传递哪种选项push，Vue 路由器导航方法 ( , replace, ) 都能始终如一地工作。gohistory

# 第8章 命名路线

除了path，您还可以提供name任何路线。这具有以下优点：

没有硬编码的 URL

自动编码/解码params

防止您在网址中出现拼写错误

绕过路径排名（例如显示 a ）

const routes = [

{

path: '/user/:username',

name: 'user',

component: User

}

]

要链接到命名路由，您可以将对象传递给router-link组件的toprop：

<router-link :to="{ name: 'user', params: { username: 'erina' }}">

User

</router-link>

这是以编程方式使用的完全相同的对象router.push()：

router.push({ name: 'user', params: { username: 'erina' } })

在这两种情况下，路由器都会导航到路径/user/erina。

完整的例子在这里。

https://github.com/vuejs/vue-router/blob/dev/examples/named-routes/app.js

# 第9章 命名视图

有时您需要同时显示多个视图而不是嵌套它们，例如创建一个带有sidebar视图和main视图的布局。这就是命名视图派上用场的地方。您可以拥有多个并为每个插座命名，而不是在您的视图中只有一个插座。没有名字的 Arouter-view将default作为它的名字。

<router-view class="view left-sidebar" name="LeftSidebar"></router-view>

<router-view class="view main-content"></router-view>

<router-view class="view right-sidebar" name="RightSidebar"></router-view>

一个视图是使用一个组件来渲染的，因此多个视图需要多个组件用于同一条路线。确保使用components（带有s）选项：

const router = createRouter({

history: createWebHashHistory(),

routes: [

{

path: '/',

components: {

default: Home,

// short for LeftSidebar: LeftSidebar

LeftSidebar,

// they match the `name` attribute on `<router-view>`

RightSidebar,

},

},

],

})

可以在此处找到此示例的工作演示。

## 9.1 嵌套命名视图

可以使用带有嵌套视图的命名视图来创建复杂的布局。这样做时，您还需要为嵌套router-view命名。让我们以设置面板为例：

/settings/emails /settings/profile

+-----------------------------------+ +------------------------------+

| UserSettings | | UserSettings |

| +-----+-------------------------+ | | +-----+--------------------+ |

| | Nav | UserEmailsSubscriptions | | +------------> | | Nav | UserProfile | |

| | +-------------------------+ | | | +--------------------+ |

| | | | | | | | UserProfilePreview | |

| +-----+-------------------------+ | | +-----+--------------------+ |

+-----------------------------------+ +------------------------------+

Nav只是一个常规组件

UserSettings是父视图组件

UserEmailsSubscriptions, UserProfile,UserProfilePreview是嵌套视图组件

注意：让我们忘记 HTML/CSS 应该如何表示这种布局并关注所使用的组件。

上述布局中的组件<template>部分UserSettings如下所示：

<!-- UserSettings.vue -->

<div>

<h1>User Settings</h1>

<NavBar />

<router-view />

<router-view name="helper" />

</div>

然后你可以用这个路由配置来实现上面的布局：

{

path: '/settings',

// You could also have named views at the top

component: UserSettings,

children: [{

path: 'emails',

component: UserEmailsSubscriptions

}, {

path: 'profile',

components: {

default: UserProfile,

helper: UserProfilePreview

}

}]

}

可以在此处找到此示例的工作演示。

# 第10章 重定向和别名

## 10.1 重定向

重定向也在routes配置中完成。从重定向/home到/：

const routes = [{ path: '/home', redirect: '/' }]

重定向也可以针对命名路由：

const routes = [{ path: '/home', redirect: { name: 'homepage' } }]

甚至使用函数进行动态重定向：

const routes = [

{

// /search/screens -> /search?q=screens

path: '/search/:searchText',

redirect: to => {

// the function receives the target route as the argument

// we return a redirect path/location here.

return { path: '/search', query: { q: to.params.searchText } }

},

},

{

path: '/search',

// ...

},

]

请注意，导航守卫不适用于重定向的路由，仅适用于其目标。例如在上面的例子中，beforeEnter给路由添加一个守卫/home不会有任何效果。

在编写 aredirect时，您可以省略该component选项，因为它永远不会直接到达，因此没有要渲染的组件。唯一的例外是嵌套路由：如果一个路由记录有children一个redirect属性，它也应该有一个component属性。

## 10.2 相对重定向

也可以重定向到相对位置：

const routes = [

{

// will always redirect /users/123/posts to /users/123/profile

path: '/users/:id/posts',

redirect: to => {

// the function receives the target route as the argument

// a relative location doesn't start with `/`

// or { path: 'profile'}

return 'profile'

},

},

]

## 10.3 别名

重定向意味着当用户访问时/home，URL 将被替换为/，然后匹配为/。但什么是别名？

/as的别名/home意味着当用户访问时/home，URL 仍然存在/home，但它会被匹配，就好像用户正在访问一样/。

以上可以在路由配置中表示为：

const routes = [{ path: '/', component: Homepage, alias: '/home' }]

别名使您可以自由地将 UI 结构映射到任意 URL，而不受配置的嵌套结构的约束。使别名以 a 开头/以使路径在嵌套路由中成为绝对路径。您甚至可以将两者结合起来并为数组提供多个别名：

const routes = [

{

path: '/users',

component: UsersLayout,

children: [

// this will render the UserList for these 3 URLs

// - /users

// - /users/list

// - /people

{ path: '', component: UserList, alias: ['/people', 'list'] },

],

},

]

如果您的路线有参数，请确保将它们包含在任何绝对别名中：

const routes = [

{

path: '/users/:id',

component: UsersByIdLayout,

children: [

// this will render the UserDetails for these 3 URLs

// - /users/24

// - /users/24/profile

// - /24

{ path: 'profile', component: UserDetails, alias: ['/:id', ''] },

],

},

]

关于 SEO 的注意事项：使用别名时，请确保定义规范链接。

<https://support.google.com/webmasters/answer/139066?hl=en>

# 第11章 将 Props 传递给路由组件

在组件中使用$route会与路由产生紧密耦合，这限制了组件的灵活性，因为它只能在某些 URL 上使用。虽然这不一定是一件坏事，但我们可以通过一个props选项来解耦这种行为：

我们可以换

const User = {

template: '<div>User {{ $route.params.id }}</div>'

}

const routes = [{ path: '/user/:id', component: User }]

和

const User = {

// make sure to add a prop named exactly like the route param

props: ['id'],

template: '<div>User {{ id }}</div>'

}

const routes = [{ path: '/user/:id', component: User, props: true }]

这允许您在任何地方使用该组件，这使得该组件更易于重用和测试。

## 11.1 布尔模式

当props设置为true时，route.params将设置为组件道具。

## 11.2 命名视图

对于具有命名视图的路由，您必须props为每个命名视图定义选项：

const routes = [

{

path: '/user/:id',

components: { default: User, sidebar: Sidebar },

props: { default: true, sidebar: false }

}

]

## 11.3 对象模式

当props是一个对象时，它将按原样设置为组件道具。当道具是静态的时很有用。

const routes = [

{

path: '/promotion/from-newsletter',

component: Promotion,

props: { newsletterPopup: false }

}

]

## 11.4 功能模式

您可以创建一个返回道具的函数。这允许您将参数转换为其他类型，将静态值与基于路由的值组合等。

const routes = [

{

path: '/search',

component: SearchUser,

props: route => ({ query: route.query.q })

}

]

URL/search?q=vue将{query: 'vue'}作为道具传递给SearchUser组件。

尽量保持props函数无状态，因为它只在路由更改时进行评估。如果您需要状态来定义道具，请使用包装器组件，这样 vue 可以对状态更改做出反应。

# 第12章 不同的历史模式

history创建路由器实例时的选项允许我们在不同的历史模式中进行选择。

## 12.1 哈希模式

哈希历史模式是通过以下方式创建的createWebHashHistory()：

import { createRouter, createWebHashHistory } from 'vue-router'

const router = createRouter({

history: createWebHashHistory(),

routes: [

//...

],

})

#它在内部传递的实际 URL 之前使用哈希字符 ( )。因为这部分 URL 永远不会发送到服务器，所以它不需要在服务器级别进行任何特殊处理。然而，它确实对 SEO 产生了不良影响。如果您对此感到担忧，请使用 HTML5 历史记录模式。

## 12.2 HTML5 模式

HTML5 模式是通过以下方式创建的，createWebHistory()并且是推荐的模式：

import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({

history: createWebHistory(),

routes: [

//...

],

})

使用createWebHistory()时，URL 看起来“正常”，例如https://example.com/user/id. 美丽的！

但是问题来了：由于我们的应用程序是单页客户端应用程序，没有适当的服务器配置，如果用户https://example.com/user/id直接在浏览器中访问，将会收到 404 错误。现在这很难看。

不用担心：要解决此问题，您需要做的就是向您的服务器添加一个简单的包罗万象的后备路由。如果 URL 与任何静态资源不匹配，它应该为index.html您的应用程序所在的同一页面提供服务。再次美丽！

## 12.3 记忆模式

内存历史模式不假设浏览器环境，因此不与 URL 交互，也不自动触发初始导航。这使其非常适合 Node 环境和 SSR。它是使用创建的，createMemoryHistory()并且需要您在调用后推送初始导航app.use(router)。

import { createRouter, createMemoryHistory } from 'vue-router'

const router = createRouter({

history: createMemoryHistory(),

routes: [

//...

],

})

虽然不推荐，但您可以在浏览器应用程序中使用此模式，但请注意不会有历史记录，这意味着您将无法返回或前进。

## 12.4 示例服务器配置

注意：以下示例假设您从根文件夹提供应用程序。如果部署到子文件夹，则应使用Vue CLI的选项和publicPathbaserouter的相关属性。您还需要调整以下示例以使用子文件夹而不是根文件夹（例如，替换RewriteBase /为RewriteBase /name-of-your-subfolder/）。

## 12.5 阿帕奇

<IfModule mod\_negotiation.c>

Options -MultiViews

</IfModule>

<IfModule mod\_rewrite.c>

RewriteEngine On

RewriteBase /

RewriteRule ^index\.html$ - [L]

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule . /index.html [L]

</IfModule>

代替mod\_rewrite，您也可以使用FallbackResource。

## 12.6 nginx

location / {

try\_files $uri $uri/ /index.html;

}

## 12.7 本机 Node.js

const http = require('http')

const fs = require('fs')

const httpPort = 80

http

.createServer((req, res) => {

fs.readFile('index.html', 'utf-8', (err, content) => {

if (err) {

console.log('We cannot open "index.html" file.')

}

res.writeHead(200, {

'Content-Type': 'text/html; charset=utf-8',

})

res.end(content)

})

})

.listen(httpPort, () => {

console.log('Server listening on: http://localhost:%s', httpPort)

})

## 12.8 使用 Node.js 表达

对于 Node.js/Express，考虑使用connect-history-api-fallback 中间件。

## 12.9 互联网信息服务 (IIS)

安装IIS UrlRewrite

web.config使用以下内容在站点的根目录中创建一个文件：

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<system.webServer>

<rewrite>

<rules>

<rule name="Handle History Mode and custom 404/500" stopProcessing="true">

<match url="(.\*)" />

<conditions logicalGrouping="MatchAll">

<add input="{REQUEST\_FILENAME}" matchType="IsFile" negate="true" />

<add input="{REQUEST\_FILENAME}" matchType="IsDirectory" negate="true" />

</conditions>

<action type="Rewrite" url="/" />

</rule>

</rules>

</rewrite>

</system.webServer>

</configuration>

## 12.10 球童 v2

try\_files {path} /

## 12.11 球童 v1

rewrite {

regexp .\*

to {path} /

}

## 12.12 Firebase 托管

将此添加到您的firebase.json：

{

"hosting": {

"public": "dist",

"rewrites": [

{

"source": "\*\*",

"destination": "/index.html"

}

]

}

}

## 12.13 网络化

创建一个\_redirects包含在部署文件中的文件：

/\* /index.html 200

在 vue-cli、nuxt 和 vite 项目中，此文件通常位于名为staticor的文件夹下public。

您可以在Netlify 文档中了解更多关于语法的信息。您还可以创建一个netlify.toml将重定向与其他 Netlify 功能相结合。

## 12.14 韦尔塞尔

vercel.json在项目的根目录下创建一个文件，内容如下：

{

"rewrites": [{ "source": "/:path\*", "destination": "/index.html" }]

}

## 12.15 警告

有一个警告：您的服务器将不再报告 404 错误，因为所有未找到的路径现在都为您的index.html文件提供服务。为了解决这个问题，你应该在你的 Vue 应用中实现一个包罗万象的路由来显示一个 404 页面：

const router = createRouter({

history: createWebHistory(),

routes: [{ path: '/:pathMatch(.\*)', component: NotFoundComponent }],

})

或者，如果您使用的是 Node.js 服务器，则可以通过使用服务器端的路由器来匹配传入的 URL，如果没有匹配的路由，则使用 404 响应来实现回退。查看Vue 服务器端渲染文档以获取更多信息。

# 第13章 导航卫士

顾名思义，Vue 路由器提供的导航守卫主要用于通过重定向或取消导航来保护导航。有许多方法可以连接到路由导航过程：全局、每个路由或组件内。

## 13.1 全球前卫

您可以使用以下方式在警卫之前注册全局router.beforeEach：

const router = createRouter({ ... })

router.beforeEach((to, from) => {

// ...

// explicitly return false to cancel the navigation

return false

})

每当触发导航时，按创建顺序调用守卫之前的全局。守卫可以异步解决，在所有钩子都解决之前，导航被认为是挂起的。

每个保护函数都接收两个参数：

to：导航到的标准化格式的目标路线位置。

from：以标准化格式导航的当前路线位置。

并且可以选择返回以下任何值：

false：取消当前导航。如果浏览器 URL 被更改（由用户手动或通过后退按钮），它将被重置为from路由的 URL。

A Route Location：通过传递一个路径位置来重定向到不同的位置，就像您正在调用一样router.push()，这允许您传递诸如replace: trueor之类的选项name: 'home'。当前导航被删除，并使用相同的from.

router.beforeEach(async (to, from) => {

if (

// make sure the user is authenticated

!isAuthenticated &&

// ❗️ Avoid an infinite redirect

to.name !== 'Login'

) {

// redirect the user to the login page

return { name: 'Login' }

}

})

Error如果遇到意外情况，也可以抛出一个。这也将取消导航并调用通过 注册的任何回调router.onError()。

如果什么都没有，undefined或者true返回，则导航被验证，并调用下一个导航守卫。

上面所有的事情都与async函数和 Promises 的工作方式相同：

router.beforeEach(async (to, from) => {

// canUserAccess() returns `true` or `false`

const canAccess = await canUserAccess(to)

if (!canAccess) return '/login'

})

## 13.2 可选的第三个参数next

在以前版本的 Vue Router 中，也可以使用第三个参数 next，这是一个常见的错误来源，并通过RFC将其删除。但是，它仍然受支持，这意味着您可以将第三个参数传递给任何导航守卫。在这种情况下，您必须next在通过导航守卫的任何给定通道中只调用一次。它可以出现不止一次，但前提是逻辑路径没有重叠，否则钩子永远不会被解析或产生错误。如果用户未通过身份验证，以下是重定向到用户的错误示例：/login

// BAD

router.beforeEach((to, from, next) => {

if (to.name !== 'Login' && !isAuthenticated) next({ name: 'Login' })

// if the user is not authenticated, `next` is called twice

next()

})

这是正确的版本：

// GOOD

router.beforeEach((to, from, next) => {

if (to.name !== 'Login' && !isAuthenticated) next({ name: 'Login' })

else next()

})

## 13.3 全球决心卫士

您可以使用 注册全局守卫router.beforeResolve。这类似于router.beforeEach因为它在每个导航上触发，但在导航确认之前调用解析防护，在所有组件内防护和异步路由组件都已解析之后。这是一个示例，可确保用户已授予对已定义自定义元属性的路由的相机访问权限requiresCamera：

router.beforeResolve(async to => {

if (to.meta.requiresCamera) {

try {

await askForCameraPermission()

} catch (error) {

if (error instanceof NotAllowedError) {

// ... handle the error and then cancel the navigation

return false

} else {

// unexpected error, cancel the navigation and pass the error to the global handler

throw error

}

}

}

})

router.beforeResolve如果用户无法进入页面，它是获取数据或执行您希望避免执行的任何其他操作的理想位置。

## 13.4 全球后钩

您还可以注册全局后挂钩，但与守卫不同的是，这些挂钩没有next功能并且不会影响导航：

router.afterEach((to, from) => {

sendToAnalytics(to.fullPath)

})

它们对于分析、更改页面标题、发布页面等可访问性功能以及许多其他事情很有用。

它们还将导航失败反映为第三个参数：

router.afterEach((to, from, failure) => {

if (!failure) sendToAnalytics(to.fullPath)

})

在其指南中了解有关导航失败的更多信息。

## 13.5 每路守卫

beforeEnter您可以直接在路由的配置对象上定义守卫：

const routes = [

{

path: '/users/:id',

component: UserDetails,

beforeEnter: (to, from) => {

// reject the navigation

return false

},

},

]

beforeEnter守卫仅在进入路线params时触发，在,query或hash更改（例如从/users/2to/users/3或从/users/2#infoto ）时不会触发/users/2#projects。它们仅在从不同路线导航时触发。

您还可以将一组函数传递给beforeEnter，这在为不同的路由重用警卫时很有用：

function removeQueryParams(to) {

if (Object.keys(to.query).length)

return { path: to.path, query: {}, hash: to.hash }

}

function removeHash(to) {

if (to.hash) return { path: to.path, query: to.query, hash: '' }

}

const routes = [

{

path: '/users/:id',

component: UserDetails,

beforeEnter: [removeQueryParams, removeHash],

},

{

path: '/about',

component: UserDetails,

beforeEnter: [removeQueryParams],

},

]

请注意，可以通过使用路由元字段和全局导航守卫来实现类似的行为。

## 13.6 组件内防护

最后，您可以直接在路由组件（传递给路由器配置的那些）内部定义路由导航防护

### 13.6.1 使用选项 API

您可以添加以下选项来路由组件：

beforeRouteEnter

beforeRouteUpdate

beforeRouteLeave

const UserDetails = {

template: `...`,

beforeRouteEnter(to, from) {

// called before the route that renders this component is confirmed.

// does NOT have access to `this` component instance,

// because it has not been created yet when this guard is called!

},

beforeRouteUpdate(to, from) {

// called when the route that renders this component has changed, but this component is reused in the new route.

// For example, given a route with params `/users/:id`, when we navigate between `/users/1` and `/users/2`,

// the same `UserDetails` component instance will be reused, and this hook will be called when that happens.

// Because the component is mounted while this happens, the navigation guard has access to `this` component instance.

},

beforeRouteLeave(to, from) {

// called when the route that renders this component is about to be navigated away from.

// As with `beforeRouteUpdate`, it has access to `this` component instance.

},

}

beforeRouteEnter守卫无权访问，因为在导航确认之前调用了守卫，因此甚至还没有创建新的进入组件。this

但是，您可以通过将回调传递给next. 确认导航后会调用回调，并将组件实例作为参数传递给回调：

beforeRouteEnter (to, from, next) {

next(vm => {

// access to component public instance via `vm`

})

}

请注意，这beforeRouteEnter是唯一支持将回调传递给next. ForbeforeRouteUpdate和beforeRouteLeave,this已经可用，因此传递回调是不必要的，因此不支持：

beforeRouteUpdate (to, from) {

// just use `this`

this.name = to.params.name

}

离开守卫通常用于防止用户意外离开未保存的编辑的路线。导航可以通过返回来取消false。

beforeRouteLeave (to, from) {

const answer = window.confirm('Do you really want to leave? you have unsaved changes!')

if (!answer) return false

}

### 13.6.2 使用组合 API

如果您正在使用组合 API 和setup函数onBeforeRouteUpdate编写组件，则可以分别通过和来添加更新和离开守卫onBeforeRouteLeave。有关详细信息，请参阅组合 API 部分。

## 13.7 完整的导航解析流程

导航触发。

beforeRouteLeave在停用的组件中调用警卫。

呼叫全球beforeEach警卫。

在重用组件中调用beforeRouteUpdate警卫。

调用beforeEnter路由配置。

解析异步路由组件。

调用beforeRouteEnter激活的组件。

呼叫全球beforeResolve警卫。

导航已确认。

调用全局afterEach挂钩。

触发了 DOM 更新。

使用实例化实例调用传递给守卫的回调next。beforeRouteEnter

# 第14章 路由元字段

有时，您可能希望将任意信息附加到路由，例如转换名称、谁可以访问路由等。这可以通过meta接受属性对象的属性来实现，并且可以在路由位置和导航守卫上访问。您可以像这样定义meta属性：

const routes = [

{

path: '/posts',

component: PostsLayout,

children: [

{

path: 'new',

component: PostsNew,

// only authenticated users can create posts

meta: { requiresAuth: true }

},

{

path: ':id',

component: PostsDetail,

// anybody can read a post

meta: { requiresAuth: false }

}

]

}

]

那么我们如何访问这个meta字段呢？

首先，routes配置中的每个路由对象称为路由记录。路由记录可以嵌套。因此，当一条路由匹配时，它可能会匹配多个路由记录。

例如，使用上述路由配置，URL/posts/new将匹配父路由记录 ( path: '/posts') 和子路由记录 ( path: 'new')。

与路由匹配的所有路由记录都$route作为数组公开在对象（以及导航守卫中的路由对象）上$route.matched。我们可以遍历该数组来检查所有meta字段，但 Vue Router 还为您提供了$route.meta一个非递归合并从父到子的所有字段。meta这意味着你可以简单地写

router.beforeEach((to, from) => {

// instead of having to check every route record with

// to.matched.some(record => record.meta.requiresAuth)

if (to.meta.requiresAuth && !auth.isLoggedIn()) {

// this route requires auth, check if logged in

// if not, redirect to login page.

return {

path: '/login',

// save the location we were at to come back later

query: { redirect: to.fullPath },

}

}

})

## 14.1 打字稿

RouteMeta可以通过从以下扩展接口来键入元字段vue-router：

// This can be directly added to any of your `.ts` files like `router.ts`

// It can also be added to a `.d.ts` file, in which case you will need to add an export

// to ensure it is treated as a module

export {}

import 'vue-router'

declare module 'vue-router' {

interface RouteMeta {

// is optional

isAdmin?: boolean

// must be declared by every route

requiresAuth: boolean

}

}

# 第15章 数据获取

有时您需要在激活路由时从服务器获取数据。例如，在呈现用户配置文件之前，您需要从服务器获取用户数据。我们可以通过两种不同的方式实现这一点：

导航后获取：首先执行导航，并在传入组件的生命周期钩子中获取数据。在获取数据时显示加载状态。

Fetching Before Navigation : 在路由进入守卫中导航前获取数据，获取数据后进行导航。

从技术上讲，两者都是有效的选择 - 它最终取决于您的目标用户体验。‘

## 15.1 导航后获取

使用这种方法时，我们会立即导航和渲染传入的组件，并在组件的created钩子中获取数据。它使我们有机会在通过网络获取数据时显示加载状态，并且我们还可以为每个视图处理不同的加载。

假设我们有一个Post组件需要根据以下内容获取帖子的数据$route.params.id：

<template>

<div class="post">

<div v-if="loading" class="loading">Loading...</div>

<div v-if="error" class="error">{{ error }}</div>

<div v-if="post" class="content">

<h2>{{ post.title }}</h2>

<p>{{ post.body }}</p>

</div>

</div>

</template>

export default {

data() {

return {

loading: false,

post: null,

error: null,

}

},

created() {

// watch the params of the route to fetch the data again

this.$watch(

() => this.$route.params,

() => {

this.fetchData()

},

// fetch the data when the view is created and the data is

// already being observed

{ immediate: true }

)

},

methods: {

fetchData() {

this.error = this.post = null

this.loading = true

// replace `getPost` with your data fetching util / API wrapper

getPost(this.$route.params.id, (err, post) => {

this.loading = false

if (err) {

this.error = err.toString()

} else {

this.post = post

}

})

},

},

}

## 15.2 导航前获取

通过这种方法，我们在实际导航到新路线之前获取数据。我们可以beforeRouteEnter在传入组件的守卫中执行数据获取，并且仅next在获取完成时调用：

export default {

data() {

return {

post: null,

error: null,

}

},

beforeRouteEnter(to, from, next) {

getPost(to.params.id, (err, post) => {

next(vm => vm.setData(err, post))

})

},

// when route changes and this component is already rendered,

// the logic will be slightly different.

async beforeRouteUpdate(to, from) {

this.post = null

try {

this.post = await getPost(to.params.id)

} catch (error) {

this.error = error.toString()

}

},

}

在为传入视图获取资源时，用户将停留在前一个视图上。因此，建议在获取数据时显示进度条或某种指示器。如果数据获取失败，还需要显示某种全局警告消息。

# 第16章 Vue 路由器和组合 API

setupVue 的Composition API的引入开辟了新的可能性，但为了能够充分发挥 Vue Router 的潜力，我们需要使用一些新功能来替换访问this和组件内导航防护。

## 16.1 访问Router和里面的当前Routesetup

因为我们无法访问thisinside of setup，所以我们无法直接访问this.$routeror this.$route。相反，我们使用以下useRouter功能：

import { useRouter, useRoute } from 'vue-router'

export default {

setup() {

const router = useRouter()

const route = useRoute()

function pushWithQuery(query) {

router.push({

name: 'search',

query: {

...route.query,

},

})

}

},

}

该route对象是一个反应对象，因此它的任何属性都可以被观察，你应该避免观察整个route对象。在大多数情况下，您应该直接观察您期望更改的参数

import { useRoute } from 'vue-router'

import { ref, watch } from 'vue'

export default {

setup() {

const route = useRoute()

const userData = ref()

// fetch the user information when params change

watch(

() => route.params.id,

async newId => {

userData.value = await fetchUser(newId)

}

)

},

}

请注意，我们仍然可以访问$routerand $routein 模板，因此无需返回router或routeinside setup。

## 16.2 导航卫士

虽然您仍然可以将组件内导航守卫与setup函数一起使用，但 Vue Router 将更新和离开守卫公开为组合 API 函数：

import { onBeforeRouteLeave, onBeforeRouteUpdate } from 'vue-router'

import { ref } from 'vue'

export default {

setup() {

// same as beforeRouteLeave option with no access to `this`

onBeforeRouteLeave((to, from) => {

const answer = window.confirm(

'Do you really want to leave? you have unsaved changes!'

)

// cancel the navigation and stay on the same page

if (!answer) return false

})

const userData = ref()

// same as beforeRouteUpdate option with no access to `this`

onBeforeRouteUpdate(async (to, from) => {

// only fetch the user if the id changed as maybe only the query or the hash changed

if (to.params.id !== from.params.id) {

userData.value = await fetchUser(to.params.id)

}

})

},

}

组合 API 防护也可以在任何由 渲染的组件中使用<router-view>，它们不必像组件内防护那样直接在路由组件上使用。

## 16.3 useLink

Vue Router 将 RouterLink 的内部行为公开为一个 Composition API 函数。v-slot它可以访问与API相同的属性：

import { RouterLink, useLink } from 'vue-router'

import { computed } from 'vue'

export default {

name: 'AppLink',

props: {

// add @ts-ignore if using TypeScript

...RouterLink.props,

inactiveClass: String,

},

setup(props) {

const { route, href, isActive, isExactActive, navigate } = useLink(props)

const isExternalLink = computed(

() => typeof props.to === 'string' && props.to.startsWith('http')

)

return { isExternalLink, href, navigate, isActive }

},

}

# 第17章 过渡

为了在你的路由组件上使用转换和动画导航，你需要使用v-slot API：

<router-view v-slot="{ Component }">

<transition name="fade">

<component :is="Component" />

</transition>

</router-view>

所有转换 API在这里的工作方式都是一样的。

## 17.1 每个路由转换

上述用法将对所有路由应用相同的转换。如果您希望每个路由的组件具有不同的转换，您可以结合元字段和动态nameon <transition>：

const routes = [

{

path: '/custom-transition',

component: PanelLeft,

meta: { transition: 'slide-left' },

},

{

path: '/other-transition',

component: PanelRight,

meta: { transition: 'slide-right' },

},

]

<router-view v-slot="{ Component, route }">

<!-- Use any custom transition and fallback to `fade` -->

<transition :name="route.meta.transition || 'fade'">

<component :is="Component" />

</transition>

</router-view>

## 17.2 基于路由的动态过渡

还可以根据目标路由和当前路由之间的关系动态确定要使用的过渡。使用与之前非常相似的片段：

<!-- use a dynamic transition name -->

<router-view v-slot="{ Component, route }">

<transition :name="route.meta.transitionName">

<component :is="Component" />

</transition>

</router-view>

我们可以添加一个after navigation hookmeta ，根据路由的深度动态向字段添加信息

router.afterEach((to, from) => {

const toDepth = to.path.split('/').length

const fromDepth = from.path.split('/').length

to.meta.transitionName = toDepth < fromDepth ? 'slide-right' : 'slide-left'

})

## 17.3 强制重用视图之间的转换

Vue 可能会自动重用看起来相似的组件，避免任何转换。幸运的是，可以添加一个key属性来强制转换。这还允许您在使用不同参数保持在同一路线上的同时触发转换：

<router-view v-slot="{ Component, route }">

<transition name="fade">

<component :is="Component" :key="route.path" />

</transition>

</router-view>

# 第18章 滚动行为

使用客户端路由时，我们可能希望在导航到新路由时滚动到顶部，或者像真正的页面重新加载一样保留历史条目的滚动位置。Vue Router 允许你实现这些，甚至更好，允许你完全自定义路由导航的滚动行为。

注意：此功能仅在浏览器支持history.pushState.

创建路由器实例时，可以提供以下scrollBehavior函数：

const router = createRouter({

history: createWebHashHistory(),

routes: [...],

scrollBehavior (to, from, savedPosition) {

// return desired position

}

})

该scrollBehavior函数接收to和from路由对象，例如Navigation Guards。第三个参数 ,savedPosition仅当这是一个popstate导航（由浏览器的后退/前进按钮触发）时才可用。

该函数可以返回一个ScrollToOptions位置对象：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

// always scroll to top

return { top: 0 }

},

})

您还可以通过 . 传递 CSS 选择器或 DOM 元素el。在这种情况下，top将left被视为该元素的相对偏移量。

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

// always scroll 10px above the element #main

return {

// could also be

// el: document.getElementById('main'),

el: '#main',

top: -10,

}

},

})

如果返回虚假值或空对象，则不会发生滚动。

使用后退/前进按钮导航时，返回savedPosition将导致类似本机的行为：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (savedPosition) {

return savedPosition

} else {

return { top: 0 }

}

},

})

如果要模拟“滚动到锚点”行为：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (to.hash) {

return {

el: to.hash,

}

}

},

})

如果您的浏览器支持滚动行为，您可以使其平滑：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

if (to.hash) {

return {

el: to.hash,

behavior: 'smooth',

}

}

}

})

延迟滚动#

有时我们需要稍等片刻才能滚动页面。例如，在处理转换时，我们希望在滚动之前等待转换完成。为此，您可以返回一个返回所需位置描述符的 Promise。这是一个示例，我们在滚动前等待 500 毫秒：

const router = createRouter({

scrollBehavior(to, from, savedPosition) {

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

setTimeout(() => {

resolve({ left: 0, top: 0 })

}, 500)

})

},

})

可以将其与来自页面级转换组件的事件挂钩，以使滚动行为与您的页面转换很好地配合，但由于用例中可能存在差异和复杂性，我们只是提供此原语来启用特定的用户空间实现。

# 第19 章 延迟加载路线

使用捆绑器构建应用程序时，JavaScript 捆绑包可能会变得非常大，从而影响页面加载时间。如果我们可以将每个路由的组件拆分为单独的块，并且仅在访问路由时加载它们，那会更有效。

Vue Router 支持开箱即用的动态导入，这意味着您可以用动态导入替换静态导入：

// replace

// import UserDetails from './views/UserDetails'

// with

const UserDetails = () => import('./views/UserDetails')

const router = createRouter({

// ...

routes: [{ path: '/users/:id', component: UserDetails }],

})

( componentand components) 选项接受一个返回组件 Promise 的函数，Vue Router只会在第一次进入页面时获取它，然后使用缓存的版本。这意味着你也可以拥有更复杂的函数，只要它们返回一个 Promise：

const UserDetails = () =>

Promise.resolve({

/\* component definition \*/

})

一般来说，始终对所有路由使用动态导入是一个好主意。

笔记

不要将异步组件用于路由。异步组件仍然可以在路由组件中使用，但路由组件本身只是动态导入。

当使用像 webpack 这样的打包工具时，这将自动受益于代码拆分

使用 Babel 时，需要添加syntax-dynamic-import插件，以便 Babel 能够正确解析语法。

在同一块中分组组件#

使用 webpack#

有时我们可能希望将嵌套在同一路由下的所有组件分组到同一个异步块中。为了实现这一点，我们需要通过使用特殊注释语法（需要 webpack > 2.4）提供块名称来使用命名块：

const UserDetails = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserDetails.vue')

const UserDashboard = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserDashboard.vue')

const UserProfileEdit = () =>

import(/\* webpackChunkName: "group-user" \*/ './UserProfileEdit.vue')

webpack 会将任何具有相同块名称的异步模块分组到同一个异步块中。

与Vite#

在 Vite 中，您可以定义以下块rollupOptions：

// vite.config.js

export default defineConfig({

build: {

rollupOptions: {

// https://rollupjs.org/guide/en/#outputmanualchunks

output: {

manualChunks: {

'group-user': [

'./src/UserDetails',

'./src/UserDashboard',

'./src/UserProfileEdit',

],

},

},

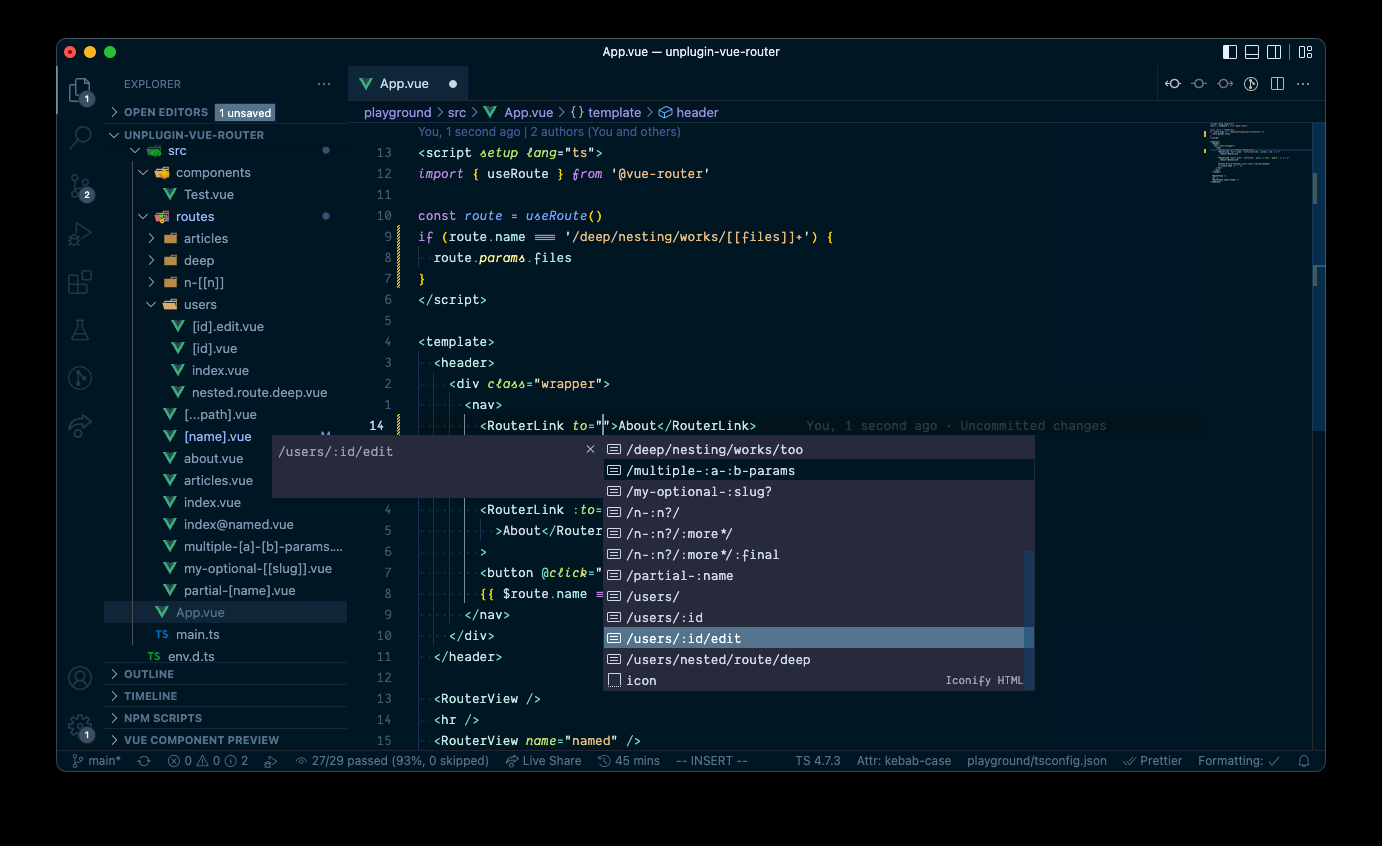
},

})

# 第20章 类型化路由 (v4.1.0+)

‼️ 实验功能

从 v4.1.0 开始，我们引入了一个名为 Typed Routes 的新功能。这个实验性功能是通过 Vite/webpack/rollup 插件启用的。



RouterLink 自动完成

有关此功能的更多信息，请查看 v4.1 发行说明。查看插件GitHub 存储库以获取安装说明和文档。

# 第21章 扩展 RouterLink

RouterLink 组件公开的内容足以props满足大多数基本应用程序的需要，但它并没有尝试涵盖所有可能的用例，您可能会发现自己正在使用v-slot一些高级用例。在大多数中型到大型应用程序中，创建一个（如果不是多个）自定义 RouterLink 组件以在您的应用程序中重用它们是值得的。一些示例是导航菜单中的链接、处理外部链接、添加inactive-class等。

让我们扩展 RouterLink 以处理外部链接并inactive-class在AppLink.vue文件中添加自定义：

<template>

<a v-if="isExternalLink" v-bind="$attrs" :href="to" target="\_blank">

<slot />

</a>

<router-link

v-else

v-bind="$props"

custom

v-slot="{ isActive, href, navigate }"

>

<a

v-bind="$attrs"

:href="href"

@click="navigate"

:class="isActive ? activeClass : inactiveClass"

>

<slot />

</a>

</router-link>

</template>

<script>

import { RouterLink } from 'vue-router'

export default {

name: 'AppLink',

inheritAttrs: false,

props: {

// add @ts-ignore if using TypeScript

...RouterLink.props,

inactiveClass: String,

},

computed: {

isExternalLink() {

return typeof this.to === 'string' && this.to.startsWith('http')

},

},

}

</script>

如果您更喜欢使用渲染函数或创建computed属性，您可以使用useLink来自Composition API的：

import { RouterLink, useLink } from 'vue-router'

export default {

name: 'AppLink',

props: {

// add @ts-ignore if using TypeScript

...RouterLink.props,

inactiveClass: String,

},

setup(props) {

// `props` contains `to` and any other prop that can be passed to <router-link>

const { navigate, href, route, isActive, isExactActive } = useLink(props)

// profit!

return { isExternalLink }

},

}

在实践中，您可能希望将AppLink组件用于应用程序的不同部分。例如使用Tailwind CSS，您可以创建一个NavLink.vue包含所有类的组件：

<template>

<AppLink

v-bind="$attrs"

class="inline-flex items-center px-1 pt-1 border-b-2 border-transparent text-sm font-medium leading-5 text-gray-500 focus:outline-none transition duration-150 ease-in-out hover:text-gray-700 hover:border-gray-300 focus:outline-none focus:text-gray-700 focus:border-gray-300 transition duration-150 ease-in-out"

active-class="border-indigo-500 text-gray-900 focus:border-indigo-700"

inactive-class="text-gray-500 hover:text-gray-700 hover:border-gray-300 focus:text-gray-700 focus:border-gray-300"

>

<slot />

</AppLink>

</template>

# 第22章 等待导航结果

使用时router-link，Vue Router 调用router.push触发导航。虽然大多数链接的预期行为是将用户导航到新页面，但在某些情况下用户仍会留在同一页面上：

用户已经在他们试图导航到的页面上。

导航守卫通过做 中止导航return false。

一个新的导航守卫发生，而前一个还没有完成。

导航守卫通过返回一个新的位置（例如）重定向到其他地方return '/login'。

导航守卫抛出.Error

如果我们想在导航完成后做某事，我们需要一种在调用后等待的方法router.push。想象一下，我们有一个移动菜单，允许我们转到不同的页面，并且我们只想在导航到新页面后隐藏菜单，我们可能想做这样的事情：

router.push('/my-profile')

this.isMenuOpen = false

但这会立即关闭菜单，因为导航是异步的，我们需要await返回的 promise router.push：

await router.push('/my-profile')

this.isMenuOpen = false

现在菜单将在导航完成后关闭，但如果导航被阻止，它也会关闭。我们需要一种方法来检测我们是否真的改变了我们所在的页面。

检测导航故障#

如果阻止导航，导致用户停留在同一页面上，则Promise返回的解析值router.push将是Navigation Failure。否则，它将是一个虚假值（通常是undefined）。这使我们能够区分我们导航离开我们所在位置的情况：

const navigationResult = await router.push('/my-profile')

if (navigationResult) {

// navigation prevented

} else {

// navigation succeeded (this includes the case of a redirection)

this.isMenuOpen = false

}

导航失败是Error具有一些额外属性的实例，它们为我们提供了足够的信息来了解哪些导航被阻止以及为什么。要检查导航结果的性质，请使用以下isNavigationFailure功能：

import { NavigationFailureType, isNavigationFailure } from 'vue-router'

// trying to leave the editing page of an article without saving

const failure = await router.push('/articles/2')

if (isNavigationFailure(failure, NavigationFailureType.aborted)) {

// show a small notification to the user

showToast('You have unsaved changes, discard and leave anyway?')

}

小费

如果省略第二个参数: isNavigationFailure(failure)，它只会检查是否failure是Navigation Failure。

区分导航故障#

正如我们在开始时所说，有不同的情况会中止导航，所有这些都会导致不同的Navigation Failures。可以使用isNavigationFailure和来区分它们NavigationFailureType。有三种不同的类型：

aborted:false在导航守卫内部返回到导航。

cancelled：在当前导航完成之前发生了新的导航。egrouter.push在导航守卫内等待时被调用。

duplicated：导航被阻止，因为我们已经在目标位置。

Navigation Failures的属性#

所有导航失败都会公开to和from属性以反映当前位置以及失败导航的目标位置：

// trying to access the admin page

router.push('/admin').then(failure => {

if (isNavigationFailure(failure, NavigationFailureType.redirected)) {

failure.to.path // '/admin'

failure.from.path // '/'

}

})

在所有情况下，to和from都是标准化的路线位置。

检测重定向#

当在 Navigation Guard 中返回一个新位置时，我们会触发一个新的导航，它会覆盖正在进行的导航。与其他返回值不同，重定向不会阻止导航，它会创建一个新的。因此，通过读取redirectedFromRoute Location 中的属性进行不同的检查：

await router.push('/my-profile')

if (router.currentRoute.value.redirectedFrom) {

// redirectedFrom is resolved route location like to and from in navigation guards

}

# 第23章 动态路由

向路由器添加路由通常是通过routes选项完成的，但在某些情况下，您可能希望在应用程序已经运行时添加或删除路由。具有可扩展界面的应用程序（如Vue CLI UI）可以使用它来使应用程序增长。

添加路线#

动态路由主要通过两个功能实现：router.addRoute()和router.removeRoute()。他们只注册一条新路线，这意味着如果新添加的路线与当前位置匹配，则需要您手动导航router.push()或router.replace()显示该新路线。我们来看一个例子：

想象一下下面的路由器只有一个路由：

const router = createRouter({

history: createWebHistory(),

routes: [{ path: '/:articleName', component: Article }],

})

转到任何页面、/about、/store或/3-tricks-to-improve-your-routing-code最终呈现Article组件。如果我们在/about并且我们添加一条新路线：

router.addRoute({ path: '/about', component: About })

该页面仍将显示该Article组件，我们需要手动调用router.replace()以更改当前位置并覆盖我们所在的位置（而不是推送一个新条目，在我们的历史记录中两次出现在同一位置）：

router.addRoute({ path: '/about', component: About })

// we could also use this.$route or route = useRoute() (inside a setup)

router.replace(router.currentRoute.value.fullPath)

请记住，await router.replace()如果您需要等待显示新路线，您可以这样做。

在导航守卫中添加路由#

如果您决定在导航守卫内添加或删除路由，则不应调用router.replace()而是通过返回新位置来触发重定向：

router.beforeEach(to => {

if (!hasNecessaryRoute(to)) {

router.addRoute(generateRoute(to))

// trigger a redirection

return to.fullPath

}

})

上面的例子假设了两件事：首先，新添加的路线记录将匹配to位置，有效地导致与我们试图访问的位置不同的位置。其次，添加新路由后hasNecessaryRoute()返回false，以避免无限重定向。

因为我们正在重定向，所以我们正在替换正在进行的导航，有效地表现得像前面显示的示例。在现实世界的场景中，添加更有可能发生在导航守卫之外，例如，当一个视图组件挂载时，它会注册新的路由。

删除路线#

有几种不同的方法可以删除现有路由：

通过添加名称冲突的路线。如果添加与现有路由同名的路由，则会先删除该路由，然后再添加该路由：

router.addRoute({ path: '/about', name: 'about', component: About })

// this will remove the previously added route because they have the same name and names are unique

router.addRoute({ path: '/other', name: 'about', component: Other })

通过调用返回的回调router.addRoute()：

const removeRoute = router.addRoute(routeRecord)

removeRoute() // removes the route if it exists

这在路由没有名称时很有用

通过使用router.removeRoute()按名称删除路由：

router.addRoute({ path: '/about', name: 'about', component: About })

// remove the route

router.removeRoute('about')

请注意Symbol，如果您希望使用此功能但希望避免名称冲突，则可以在路由中使用 s 来表示名称。

每当删除路由时，它的所有别名和子级都会随之删除。

添加嵌套路由#

要将嵌套路由添加到现有路由，您可以将路由名称作为其第一个参数传递给router.addRoute()，这将有效地添加路由，就好像它是通过添加一样children：

router.addRoute({ name: 'admin', path: '/admin', component: Admin })

router.addRoute('admin', { path: 'settings', component: AdminSettings })

这相当于：

router.addRoute({

name: 'admin',

path: '/admin',

component: Admin,

children: [{ path: 'settings', component: AdminSettings }],

})

查看现有路线#

Vue Router 为您提供了两个功能来查看现有路由：

router.hasRoute(): 检查路由是否存在

router.getRoutes()：获取包含所有路线记录的数组。