# 第1章 安装

#直接下载/CDN

https://unpkg.com/vue-router/dist/vue-router.js（打开新窗口）

Unpkg.com （打开新窗口）提供基于 npm 的 CDN 链接。上面的链接将始终指向 npm 上的最新版本。您还可以通过诸如https://unpkg.com/vue-router@2.0.0/dist/vue-router.js.

包含vue-router在 Vue 之后，它会自动安装：

<script src="/path/to/vue.js"></script>

<script src="/path/to/vue-router.js"></script>

#npm

npm install vue-router

当与模块系统一起使用时，您必须通过以下方式显式安装路由器Vue.use()：

import Vue from 'vue'

import VueRouter from 'vue-router'

Vue.use(VueRouter)

使用全局脚本标签时不需要这样做。

#Vue CLI

如果你有一个使用Vue CLI的项目 （打开新窗口）您可以将 Vue Router 添加为插件。您可以让 CLI 为您生成上面的代码以及两个示例路由。它还将覆盖您App.vue的文件，因此请确保在项目中运行以下命令之前备份文件：

vue add router

#开发构建

vue-router如果您想使用最新的开发版本，则必须直接从 GitHub 克隆并自己构建。

git clone https://github.com/vuejs/vue-router.git node\_modules/vue-router

cd node\_modules/vue-router

npm install

npm run build

# 第2章 入门

笔记

您正在阅读 Vue Router 3 for Vue 2的文档。如果您正在使用 Vue 3，请使用Vue Router 4 文档 （打开新窗口）反而。

我们将使用ES2015 （打开新窗口）在指南的代码示例中。

此外，所有示例都将使用完整版本的 Vue 来实现动态模板编译。在此处查看更多详细信息 （打开新窗口）.

在 Vue School 上观看有关 Vue 路由器的免费视频课程

使用 Vue + Vue Router 创建单页应用程序感觉很自然：使用 Vue.js，我们已经在使用组件组合我们的应用程序。将 Vue Router 添加到组合中时，我们需要做的就是将我们的组件映射到路由并让 Vue Router 知道在哪里渲染它们。这是一个基本示例：

#HTML

<script src="https://unpkg.com/vue/dist/vue.js"></script>

<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="app">

<h1>Hello App!</h1>

<p>

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

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

<!-- `<router-link>` will be rendered as an `<a>` tag by default -->

<router-link to="/foo">Go to Foo</router-link>

<router-link to="/bar">Go to Bar</router-link>

</p>

<!-- route outlet -->

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

<router-view></router-view>

</div>

#JavaScript

// 0. If using a module system (e.g. via vue-cli), import Vue and VueRouter

// and then call `Vue.use(VueRouter)`.

// 1. Define route components.

// These can be imported from other files

const Foo = { template: '<div>foo</div>' }

const Bar = { template: '<div>bar</div>' }

// 2. Define some routes

// Each route should map to a component. The "component" can

// either be an actual component constructor created via

// `Vue.extend()`, or just a component options object.

// We'll talk about nested routes later.

const routes = [

{ path: '/foo', component: Foo },

{ path: '/bar', component: Bar }

]

// 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 = new VueRouter({

routes // short for `routes: routes`

})

// 4. Create and mount the root instance.

// Make sure to inject the router with the router option to make the

// whole app router-aware.

const app = new Vue({

router

}).$mount('#app')

// Now the app has started!

通过注入路由器，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: {

goBack() {

window.history.length > 1 ? this.$router.go(-1) : this.$router.push('/')

}

}

}

在整个文档中，我们将经常使用router实例。请记住，this.$router这与使用router. 我们使用的原因this.$router是因为我们不想在需要操作路由的每个组件中都导入路由器。

您还可以实时查看此示例 （打开新窗口）.

请注意，当其目标路由匹配时，a<router-link>会自动获取该类。.router-link-active您可以在其API 参考中了解更多信息。

# 第3章 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。我们可以在vue-router路径中使用动态段来实现：

const User = {

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

}

const router = new VueRouter({

routes: [

// dynamic segments start with a colon

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

]

})

现在 URL 喜欢/user/foo并且/user/bar都将映射到相同的路由。

动态段用冒号表示:。当路由匹配时，动态段的值将this.$route.params在每个组件中公开。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>'

}

你可以在这里查看一个活生生的例子 （打开新窗口）.

您可以在同一条路线中有多个动态段，它们将映射到 上的相应字段$route.params。例子：

图案 匹配路径 $route.params

/user/:用户名 /用户/埃文 { username: 'evan' }

/user/:用户名/post/:post\_id /user/evan/post/123 { username: 'evan', post\_id: '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。

#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/user/footo导航时/user/bar，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：

const User = {

template: '...',

watch: {

$route(to, from) {

// react to route changes...

}

}

}

或者，使用2.2 中引入的beforeRouteUpdate 导航守卫：

const User = {

template: '...',

beforeRouteUpdate(to, from, next) {

// react to route changes...

// don't forget to call next()

}

}

#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，我们可以使用星号 ( \*)：

{

// will match everything

path: '\*'

}

{

// will match anything starting with `/user-`

path: '/user-\*'

}

使用星号路线时，请确保正确排列路线，使星号路线在最后。该路由{ path: '\*' }通常用于 404 客户端。如果您使用的是历史模式，请确保也正确配置您的服务器。

使用星号时，一个名为的参数pathMatch会自动添加到$route.params. 它包含由星号匹配的其余 url ：

// Given a route { path: '/user-\*' }

this.$router.push('/user-admin')

this.$route.params.pathMatch // 'admin'

// Given a route { path: '\*' }

this.$router.push('/non-existing')

this.$route.params.pathMatch // '/non-existing'

#高级匹配模式

vue-router使用正则表达式的路径 （打开新窗口）作为其路径匹配引擎，因此它支持许多高级匹配模式，例如可选动态段、零个或多个/一个或多个要求，甚至自定义正则表达式模式。查看它的文档 （打开新窗口）对于这些高级模式，以及这个例子 （打开新窗口）中使用它们vue-router。

#匹配优先级

有时，同一个 URL 可能会被多个路由匹配。在这种情况下，匹配优先级由路由定义的顺序决定：路由定义越早，优先级越高。

# 第4章 嵌套路由

真正的应用程序 UI 通常由嵌套多个级别的组件组成。一个 URL 的段对应于嵌套组件的某种结构也是很常见的，例如：

/user/foo/profile /user/foo/posts

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

| User | | User |

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

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

| | | | | | | |

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

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

有了vue-router，使用嵌套路由配置来表达这种关系非常简单。

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>'

}

const router = new VueRouter({

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

})

这里<router-view>是顶级出口。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嵌套插座中，我们需要使用构造函数配置children中的选项：VueRouter

const router = new VueRouter({

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选项只是另一个路由配置对象数组，就像routes它自己一样。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，当你访问时/user/foo，'soutlet 内部不会渲染任何东西User，因为没有匹配到子路由。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子路由路径：

const router = new VueRouter({

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 （打开新窗口）.

# 第5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。

#router.push(location, onComplete?, onAbort?)

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('home')

// object

router.push({ path: 'home' })

// named route

router.push({ name: 'user', params: { userId: '123' } })

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

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

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 userId = '123'

router.push({ name: 'user', params: { userId } }) // -> /user/123

router.push({ path: `/user/${userId}` }) // -> /user/123

// This will NOT work

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

相同的规则适用于组件的to属性router-link。

在 2.2.0+ 中，可选地提供onComplete和onAbort回调router.push或router.replace作为第二个和第三个参数。这些回调将分别在导航成功完成（在解决所有异步挂钩之后）或中止（导航到相同的路线，或在当前导航完成之前导航到不同的路线）时调用。在 3.1.0+ 中，您可以省略第二个和第三个参数，如果支持 Promises，则router.push/将返回一个 Promise。router.replace

注意：如果目的地与当前路线相同，并且只有参数发生变化（例如从一个配置文件转到另一个配置文件/users/1-> /users/2），您将不得不使用beforeRouteUpdate来对更改做出反应（例如获取用户信息）。

#router.replace(location, onComplete?, onAbort?)

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(...)

#router.go(n)

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 history.forward()

router.go(1)

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

#历史操纵

您可能已经注意到router.push和是router.replace和的router.go对应window.history.pushState物window.history.replaceStatewindow.history.go （打开新窗口），并且它们确实模仿了window.historyAPI。

因此，如果您已经熟悉浏览器历史 API （打开新窗口），使用 Vue Router 处理历史将变得非常容易。

值得一提的是，Vue 路由器导航方法（push、replace、go）在所有路由器模式（history、hash和abstract）下都能正常工作。

# 第6章 命名路线

有时用名称来识别路线更方便，尤其是在链接到路线或执行导航时。您可以routes在创建 Router 实例时在选项中为路由命名：

const router = new VueRouter({

routes: [

{

path: '/user/:userId',

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: { userId: 123 }}">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: { userId: 123 } })

在这两种情况下，路由器都会导航到路径/user/123。

# 第7 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 one"></router-view>

<router-view class="view two" name="a"></router-view>

<router-view class="view three" name="b"></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(with an s) 选项：

const router = new VueRouter({

routes: [

{

path: '/',

components: {

default: Foo,

a: Bar,

b: Baz

}

}

]

})

可以在此处找到此示例的工作演示 （打开新窗口）.

#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，您还需要命名router-view使用的嵌套组件。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>

此处省略了嵌套视图组件，但您可以在此处找到上述示例的完整源代码 （打开新窗口）.

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 （打开新窗口）.

# 第8章 重定向和别名

#重定向

重定向也在routes配置中完成。从重定向/a到/b：

const router = new VueRouter({

routes: [

{ path: '/a', redirect: '/b' }

]

})

重定向也可以针对命名路由：

const router = new VueRouter({

routes: [

{ path: '/a', redirect: { name: 'foo' }}

]

})

甚至使用函数进行动态重定向：

const router = new VueRouter({

routes: [

{ path: '/a', redirect: to => {

// the function receives the target route as the argument

// return redirect path/location here.

}}

]

})

请注意，导航守卫不适用于重定向的路由，仅适用于其目标。在下面的示例中，向路由添加beforeEnter警卫/a不会产生任何影响。

对于其他高级用法，请查看示例 （打开新窗口）.

#别名

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

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

以上可以在路由配置中表示为：

const router = new VueRouter({

routes: [

{ path: '/a', component: A, alias: '/b' }

]

})

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。

有关高级用法，请查看示例 （打开新窗口）.

# 第9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。

要将此组件与路由器使用选项分离props：

而不是耦合到$route：

const User = {

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

}

const router = new VueRouter({

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

})

通过使用解耦它props

const User = {

props: ['id'],

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

}

const router = new VueRouter({

routes: [

{ path: '/user/:id', component: User, props: true },

// for routes with named views, you have to define the `props` option for each named view:

{

path: '/user/:id',

components: {

default: User,

sidebar: Sidebar

},

props: {

default: true,

// function mode, more about it below

sidebar: route => ({ search: route.query.q })

}

}

]

})

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。

#布尔模式

当props设置为true时，route.params将设置为组件道具。

#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 router = new VueRouter({

routes: [

{

path: '/promotion/from-newsletter',

component: Promotion,

props: { newsletterPopup: false }

}

]

})

#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 router = new VueRouter({

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。

有关高级用法，请查看示例 （打开新窗口）.

# 第10章 HTML5 历史模式

默认模式vue-router是散列模式- 它使用 URL 散列来模拟完整的 URL，以便在 URL 更改时不会重新加载页面。

为了摆脱散列，我们可以使用路由器的历史模式，它利用history.pushStateAPI 实现 URL 导航而无需重新加载页面：

const router = new VueRouter({

mode: 'history',

routes: [...]

})

使用历史模式时，URL 看起来“正常”，例如http://oursite.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http://oursite.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！

#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publicPath （打开新窗口）以及路由器的相关属性base （打开新窗口）. 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/）。

#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，您还可以使用FallbackResource （打开新窗口）.

#nginx

location / {

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

}

#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)

})

#使用 Node.js 表达

对于 Node.js/Express，考虑使用connect-history-api-fallback 中间件 （打开新窗口）.

#互联网信息服务 (IIS)

安装IIS UrlRewrite（打开新窗口）

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>

#球童 v2

try\_files {path} /

#球童 v1

rewrite {

regexp .\*

to {path} /

}

#Firebase 托管

将此添加到您的firebase.json：

{

"hosting": {

"public": "dist",

"rewrites": [

{

"source": "\*\*",

"destination": "/index.html"

}

]

}

}

#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 = new VueRouter({

mode: 'history',

routes: [

{

path: '/:catchAll(.\*)',

component: NotFoundComponent,

name: 'NotFound'

}

]

})

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 （打开新窗口）了解更多信息。

# 第11章 导航卫士

顾名思义，由 提供的导航守卫vue-router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。

请记住，参数或查询更改不会触发进入/离开导航守卫。您可以观察$route对象以对这些更改做出反应，也可以使用beforeRouteUpdate组件内保护。

#全球前卫

通过 Vue School 上的免费课程了解导航守卫的工作原理

您可以使用以下方式在警卫之前注册全局router.beforeEach：

const router = new VueRouter({ ... })

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

// ...

})

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。

每个守卫函数都接收三个参数：

to: Route:导航到的目标路由对象。

from: Route：正在导航的当前路线。

next: Function：必须调用此函数来解决钩子。该操作取决于提供给的参数next：

next(): 转到管道中的下一个钩子。如果没有留下任何钩子，则确认导航。

next(false): 中止当前导航。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。

next('/')或next({ path: '/' })：重定向到不同的位置。当前导航将中止，并开始新的导航。您可以将任何位置对象传递给，它next允许您指定选项replace: true，如name: 'home'router-linktorouter.push

next(error): (2.4.0+) 如果传递给的参数next是 的实例Error，导航将中止，错误将传递给通过 注册的回调router.onError()。

确保next在通过导航守卫的任何给定传递中，该函数仅被调用一次。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。/login如果用户未通过身份验证，以下是重定向到用户的示例：

// 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()

})

// GOOD

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

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

else next()

})

#全球决心卫士

您可以使用 注册全局守卫router.beforeResolve。这类似于，不同之处在于，在所有组件内保护和异步路由组件都已解析之后router.beforeEach，将在导航确认之前调用解析保护。

#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) => {

// ...

})

#每路守卫

beforeEnter您可以直接在路由的配置对象上定义守卫：

const router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

beforeEnter: (to, from, next) => {

// ...

}

}

]

})

这些守卫具有与 global before 守卫完全相同的签名。

#组件内防护

最后，您可以使用以下选项直接在路由组件（传递给路由器配置的组件）内定义路由导航防护：

beforeRouteEnter

beforeRouteUpdate

beforeRouteLeave

const Foo = {

template: `...`,

beforeRouteEnter(to, from, next) {

// 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, next) {

// called when the route that renders this component has changed.

// This component being reused (by using an explicit `key`) in the new route or not doesn't change anything.

// For example, for a route with dynamic params `/foo/:id`, when we

// navigate between `/foo/1` and `/foo/2`, the same `Foo` component instance

// will be reused (unless you provided a `key` to `<router-view>`), and this hook will be called when that happens.

// has access to `this` component instance.

},

beforeRouteLeave(to, from, next) {

// called when the route that renders this component is about to

// be navigated away from.

// 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 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, next) {

// just use `this`

this.name = to.params.name

next()

}

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。可以通过调用取消导航next(false)。

beforeRouteLeave (to, from, next) {

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

if (answer) {

next()

} else {

next(false)

}

}

如果您正在使用添加组件内导航防护的 mixin，请确保在安装路由器插件后添加 mixin ：

Vue.use(Router)

Vue.mixin({

beforeRouteUpdate(to, from, next) {

// ...

}

})

#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12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 router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

children: [

{

path: 'bar',

component: Bar,

// a meta field

meta: { requiresAuth: true }

}

]

}

]

})

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/foo/bar将匹配父路由记录和子路由记录。

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。因此，我们将需要迭代$route.matched以检查路由记录中的元字段。

一个示例用例是检查全局导航守卫中的元字段：

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

if (to.matched.some(record => record.meta.requiresAuth)) {

// this route requires auth, check if logged in

// if not, redirect to login page.

if (!auth.loggedIn()) {

next({

path: '/login',

query: { redirect: to.fullPath }

})

} else {

next()

}

} else {

next() // make sure to always call next()!

}

})

# 第13章 过渡

由于<router-view>本质上是一个动态组件，我们可以使用<transition>组件以相同的方式对其应用过渡效果：

<transition>

<router-view></router-view>

</transition>

所有转换 API （打开新窗口）在这里工作一样。

#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<transition>在每个路由组件内使用不同的名称：

const Foo = {

template: `

<transition name="slide">

<div class="foo">...</div>

</transition>

`

}

const Bar = {

template: `

<transition name="fade">

<div class="bar">...</div>

</transition>

`

}

#基于路由的动态过渡

也可以根据目标路由和当前路由的关系动态确定要使用的过渡：

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

<transition :name="transitionName">

<router-view></router-view>

</transition>

// then, in the parent component,

// watch the `$route` to determine the transition to use

watch: {

'$route' (to, from) {

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

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

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

}

}

# 第14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。

#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 () {

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

// already being observed

this.fetchData()

},

watch: {

// call again the method if the route changes

'$route': 'fetchData'

},

methods: {

fetchData () {

this.error = this.post = null

this.loading = true

const fetchedId = this.$route.params.id

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

getPost(fetchedId, (err, post) => {

// make sure this request is the last one we did, discard otherwise

if (this.$route.params.id !== fetchedId) return

this.loading = false

if (err) {

this.error = err.toString()

} else {

this.post = post

}

})

}

}

}

#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.

beforeRouteUpdate (to, from, next) {

this.post = null

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

this.setData(err, post)

next()

})

},

methods: {

setData (err, post) {

if (err) {

this.error = err.toString()

} else {

this.post = post

}

}

}

}

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15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允许您实现这些甚至更好，允许您完全自定义路线导航上的滚动行为。

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 = new VueRouter({

routes: [...],

scrollBehavior (to, from, savedPosition) {

// return desired position

}

})

该scrollBehavior函数接收to和from路由对象。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滚动位置对象。对象可以是以下形式：

{ x: number, y: number }

{ selector: string, offset? : { x: number, y: number }}（仅在 2.6.0+ 中支持偏移）

如果返回虚假值或空对象，则不会发生滚动。

例如：

scrollBehavior (to, from, savedPosition) {

return { x: 0, y: 0 }

}

这只会使页面滚动到所有路线导航的顶部。

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

scrollBehavior (to, from, savedPosition) {

if (savedPosition) {

return savedPosition

} else {

return { x: 0, y: 0 }

}

}

如果要模拟“滚动到锚点”行为：

scrollBehavior (to, from, savedPosition) {

if (to.hash) {

return {

selector: to.hash

// , offset: { x: 0, y: 10 }

}

}

}

我们还可以使用路由元字段来实现细粒度的滚动行为控制。在此处查看完整示例 （打开新窗口）.

#异步滚动

2.8.0 中的新功能

您还可以返回解析为所需位置描述符的 Promise：

scrollBehavior (to, from, savedPosition) {

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

setTimeout(() => {

resolve({ x: 0, y: 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。

#平滑滚动

您可以为支持它的浏览器启用本机平滑滚动 （打开新窗口）通过简单地将behavior选项添加到内部返回的对象scrollBehavior：

scrollBehavior (to, from, savedPosition) {

if (to.hash) {

return {

selector: to.hash,

behavior: 'smooth',

}

}

}

# 第16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一个单独的块，并且仅在访问该路由时加载它们，那将会更有效率。

结合Vue的异步组件特性 （打开新窗口）和 webpack 的代码拆分功能 （打开新窗口），延迟加载路由组件非常容易。

首先，可以将异步组件定义为返回 Promise 的工厂函数（应解析为组件本身）：

const Foo = () =>

Promise.resolve({

/\* component definition \*/

})

二、在webpack 2中，我们可以使用动态导入 （打开新窗口）表示代码分割点的语法：

import('./Foo.vue') // returns a Promise

笔记

如果你使用 Babel，你需要添加syntax-dynamic-import （打开新窗口）插件，以便 Babel 可以正确解析语法。

结合这两者，这是如何定义一个将被 webpack 自动代码拆分的异步组件：

const Foo = () => import('./Foo.vue')

路由配置中无需更改任何内容，只需Foo照常使用：

const router = new VueRouter({

routes: [{ path: '/foo', component: Foo }]

})

#在同一块中分组组件

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，我们需要使用命名块 （打开新窗口）通过使用特殊的注释语法提供块名称（需要 webpack > 2.4）：

const Foo = () => import(/\* webpackChunkName: "group-foo" \*/ './Foo.vue')

const Bar = () => import(/\* webpackChunkName: "group-foo" \*/ './Bar.vue')

const Baz = () => import(/\* webpackChunkName: "group-foo" \*/ './Baz.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。

# 第17章 导航失败

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

导航守卫通过调用来中止导航next(false)

导航守卫抛出错误或调用next(new Error())

使用router-link组件时，这些故障都不会记录错误。但是，如果您使用router.pushor router.replace，您可能会在控制台中遇到“未捕获（承诺）错误”消息，然后是更具体的消息。让我们了解如何区分Navigation Failures。

背景故事

在 v3.2.0 中，导航失败通过router.push:onComplete和的两个可选回调公开onAbort。从 3.1.0 版本开始，如果没有提供/回调，则router.push返回router.replacePromise 。这个Promise解决而不是调用，并且拒绝而不是调用。onCompleteonAbortonCompleteonAbort

#检测导航故障

导航失败是Error具有一些额外属性的实例。要检查错误是否来自路由器，请使用以下isNavigationFailure函数：

import VueRouter from 'vue-router'

const { isNavigationFailure, NavigationFailureType } = VueRouter

// trying to access the admin page

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

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

// show a small notification to the user

showToast('Login in order to access the admin panel')

}

})

小费

如果省略第二个参数: isNavigationFailure(failure)，它将仅检查错误是否为Navigation Failure。

#NavigationFailureType

NavigationFailureType帮助开发人员区分不同类型的导航失败。有四种不同的类型：

redirected:next(newLocation)在导航守卫内部被调用以重定向到其他地方。

aborted:next(false)在导航的导航守卫内部被调用。

cancelled：在当前导航完成之前完全发生了新的导航。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，以分别反映失败导航的目标和当前位置：

// trying to access the admin page

router.push('/admin').catch(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。