# 第1 章 Installation

Direct Download / CDN

https://unpkg.com/vue-router/dist/vue-router.js(opens new window)

Unpkg.com (opens new window)provides npm-based CDN links. The above link will always point to the latest release on npm. You can also use a specific version/tag via URLs like https://unpkg.com/vue-router@2.0.0/dist/vue-router.js.

Include vue-router after Vue and it will install itself automatically:

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

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

#npm

npm install vue-router

When used with a module system, you must explicitly install the router via Vue.use():

import Vue from 'vue'

import VueRouter from 'vue-router'

Vue.use(VueRouter)

You don't need to do this when using global script tags.

#Vue CLI

If you have a project using Vue CLI (opens new window)you can add Vue Router as a plugin. You can let the CLI generate the code above for you as well as two sample routes. It will also overwrite your App.vue so make sure to backup the file before running the following command inside your project:

vue add router

#Dev Build

You will have to clone directly from GitHub and build vue-router yourself if you want to use the latest dev build.

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 Getting Started

Note

You are reading the documentation of Vue Router 3 for Vue 2. If you are working with Vue 3, use the Vue Router 4 documentation (opens new window)instead.

We will be using ES2015 (opens new window)in the code samples in the guide.

Also, all examples will be using the full version of Vue to make on-the-fly template compilation possible. See more details here (opens new window).

Watch a free video course about Vue Router on Vue School

Creating a Single-page Application with Vue + Vue Router feels natural: with Vue.js, we are already composing our application with components. When adding Vue Router to the mix, all we need to do is map our components to the routes and let Vue Router know where to render them. Here's a basic example:

#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!

By injecting the router, we get access to it as this.$router as well as the current route as this.$route inside of any component:

// 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('/')

}

}

}

Throughout the docs, we will often use the router instance. Keep in mind that this.$router is exactly the same as using router. The reason we use this.$router is because we don't want to import the router in every single component that needs to manipulate routing.

You can also check out this example live (opens new window).

Notice that a <router-link> automatically gets the .router-link-active class when its target route is matched. You can learn more about it in its API reference.

# 第3章 Dynamic Route Matching

Very often we will need to map routes with the given pattern to the same component. For example we may have a User component which should be rendered for all users but with different user IDs. In vue-router we can use a dynamic segment in the path to achieve that:

const User = {

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

}

const router = new VueRouter({

routes: [

// dynamic segments start with a colon

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

]

})

Now URLs like /user/foo and /user/bar will both map to the same route.

A dynamic segment is denoted by a colon :. When a route is matched, the value of the dynamic segments will be exposed as this.$route.params in every component. Therefore, we can render the current user ID by updating User's template to this:

const User = {

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

}

You can check out a live example here (opens new window).

You can have multiple dynamic segments in the same route, and they will map to corresponding fields on $route.params. Examples:

pattern matched path $route.params

/user/:username /user/evan { username: 'evan' }

/user/:username/post/:post\_id /user/evan/post/123 { username: 'evan', post\_id: '123' }

In addition to $route.params, the $route object also exposes other useful information such as $route.query (if there is a query in the URL), $route.hash, etc. You can check out the full details in the API Reference.

#Reacting to Params Changes

One thing to note when using routes with params is that when the user navigates from /user/foo to /user/bar, the same component instance will be reused. Since both routes render the same component, this is more efficient than destroying the old instance and then creating a new one. However, this also means that the lifecycle hooks of the component will not be called.

To react to params changes in the same component, you can simply watch the $route object:

const User = {

template: '...',

watch: {

$route(to, from) {

// react to route changes...

}

}

}

Or, use the beforeRouteUpdate navigation guard introduced in 2.2:

const User = {

template: '...',

beforeRouteUpdate(to, from, next) {

// react to route changes...

// don't forget to call next()

}

}

#Catch all / 404 Not found Route

Regular params will only match characters in between url fragments, separated by /. If we want to match anything, we can use the asterisk (\*):

{

// will match everything

path: '\*'

}

{

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

path: '/user-\*'

}

When using asterisk routes, make sure to correctly order your routes so that asterisk ones are at the end. The route { path: '\*' } is usually used to 404 client side. If you are using History mode, make sure to correctly configure your server as well.

When using an asterisk, a param named pathMatch is automatically added to $route.params. It contains the rest of the url matched by the asterisk:

// 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'

#Advanced Matching Patterns

vue-router uses path-to-regexp (opens new window)as its path matching engine, so it supports many advanced matching patterns such as optional dynamic segments, zero or more / one or more requirements, and even custom regex patterns. Check out its documentation (opens new window)for these advanced patterns, and this example (opens new window)of using them in vue-router.

#Matching Priority

Sometimes the same URL may be matched by multiple routes. In such a case the matching priority is determined by the order of route definition: the earlier a route is defined, the higher priority it gets.

# 第4章 Nested Routes

Real app UIs are usually composed of components that are nested multiple levels deep. It is also very common that the segments of a URL corresponds to a certain structure of nested components, for example:

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

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

| User | | User |

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

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

| | | | | | | |

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

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

With vue-router, it is very simple to express this relationship using nested route configurations.

Given the app we created in the last chapter:

<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 }]

})

The <router-view> here is a top-level outlet. It renders the component matched by a top level route. Similarly, a rendered component can also contain its own, nested <router-view>. For example, if we add one inside the User component's template:

const User = {

template: `

<div class="user">

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

<router-view></router-view>

</div>

`

}

To render components into this nested outlet, we need to use the children option in VueRouter constructor config:

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

}

]

}

]

})

Note that nested paths that start with / will be treated as a root path. This allows you to leverage the component nesting without having to use a nested URL.

As you can see the children option is just another Array of route configuration objects like routes itself. Therefore, you can keep nesting views as much as you need.

At this point, with the above configuration, when you visit /user/foo, nothing will be rendered inside User's outlet, because no sub route is matched. Maybe you do want to render something there. In such case you can provide an empty subroute path:

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

]

}

]

})

A working demo of this example can be found here (opens new window).

# 第5章 Programmatic Navigation

Aside from using <router-link> to create anchor tags for declarative navigation, we can do this programmatically using the router's instance methods.

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

Note: Inside of a Vue instance, you have access to the router instance as $router. You can therefore call this.$router.push.

To navigate to a different URL, use router.push. This method pushes a new entry into the history stack, so when the user clicks the browser back button they will be taken to the previous URL.

This is the method called internally when you click a <router-link>, so clicking <router-link :to="..."> is the equivalent of calling router.push(...).

Declarative Programmatic

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

The argument can be a string path, or a location descriptor object. Examples:

// 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' } })

Note: params are ignored if a path is provided, which is not the case for query, as shown in the example above. Instead, you need to provide the name of the route or manually specify the whole path with any parameter:

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

The same rules apply for the to property of the router-link component.

In 2.2.0+, optionally provide onComplete and onAbort callbacks to router.push or router.replace as the 2nd and 3rd arguments. These callbacks will be called when the navigation either successfully completed (after all async hooks are resolved), or aborted (navigated to the same route, or to a different route before current navigation has finished), respectively. In 3.1.0+, you can omit the 2nd and 3rd parameter and router.push/router.replace will return a promise instead if Promises are supported.

Note: If the destination is the same as the current route and only params are changing (e.g. going from one profile to another /users/1 -> /users/2), you will have to use beforeRouteUpdate to react to changes (e.g. fetching the user information).

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

It acts like router.push, the only difference is that it navigates without pushing a new history entry, as its name suggests - it replaces the current entry.

Declarative Programmatic

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

#router.go(n)

This method takes a single integer as parameter that indicates by how many steps to go forwards or go backwards in the history stack, similar to window.history.go(n).

Examples

// 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)

#History Manipulation

You may have noticed that router.push, router.replace and router.go are counterparts of window.history.pushState, window.history.replaceState and window.history.go (opens new window), and they do imitate the window.history APIs.

Therefore, if you are already familiar with Browser History APIs (opens new window), manipulating history will be super easy with Vue Router.

It is worth mentioning that Vue Router navigation methods (push, replace, go) work consistently in all router modes (history, hash and abstract).

# 第6章 Named Routes

Sometimes it is more convenient to identify a route with a name, especially when linking to a route or performing navigations. You can give a route a name in the routes options while creating the Router instance:

const router = new VueRouter({

routes: [

{

path: '/user/:userId',

name: 'user',

component: User

}

]

})

To link to a named route, you can pass an object to the router-link component's to prop:

<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>

This is the exact same object used programmatically with router.push():

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

In both cases, the router will navigate to the path /user/123.

Full example here (opens new window).

# 第7章 Named Views

Sometimes you need to display multiple views at the same time instead of nesting them, e.g. creating a layout with a sidebar view and a main view. This is where named views come in handy. Instead of having one single outlet in your view, you can have multiple and give each of them a name. A router-view without a name will be given default as its name.

<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>

A view is rendered by using a component, therefore multiple views require multiple components for the same route. Make sure to use the components (with an s) option:

const router = new VueRouter({

routes: [

{

path: '/',

components: {

default: Foo,

a: Bar,

b: Baz

}

}

]

})

A working demo of this example can be found here (opens new window).

#Nested Named Views

It is possible to create complex layouts using named views with nested views. When doing so, you will also need to name nested router-view components used. Let's take a Settings panel example:

/settings/emails /settings/profile

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

| UserSettings | | UserSettings |

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

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

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

| | | | | | | | UserProfilePreview | |

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

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

Nav is just a regular component

UserSettings is the view component

UserEmailsSubscriptions, UserProfile, UserProfilePreview are nested view components

Note: Let's forget about how the HTML/CSS should look like to represent such layout and focus on the components used.

The <template> section for UserSettings component in the above layout would look something like this:

<!-- UserSettings.vue -->

<div>

<h1>User Settings</h1>

<NavBar/>

<router-view/>

<router-view name="helper"/>

</div>

The nested view components are omitted here but you can find the complete source code for the example above here (opens new window).

Then you can achieve the layout above with this route configuration:

{

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

}

}]

}

A working demo of this example can be found here (opens new window).

# 第8章 Redirect and Alias

Redirect

Redirecting is also done in the routes configuration. To redirect from /a to /b:

const router = new VueRouter({

routes: [

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

]

})

The redirect can also be targeting a named route:

const router = new VueRouter({

routes: [

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

]

})

Or even use a function for dynamic redirecting:

const router = new VueRouter({

routes: [

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

// the function receives the target route as the argument

// return redirect path/location here.

}}

]

})

Note that Navigation Guards are not applied on the route that redirects, only on its target. In the example below, adding a beforeEnter guard to the /a route would not have any effect.

For other advanced usage, checkout the example (opens new window).

#Alias

A redirect means when the user visits /a, the URL will be replaced by /b, and then matched as /b. But what is an alias?

An alias of /a as /b means when the user visits /b, the URL remains /b, but it will be matched as if the user is visiting /a.

The above can be expressed in the route configuration as:

const router = new VueRouter({

routes: [

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

]

})

An alias gives you the freedom to map a UI structure to an arbitrary URL, instead of being constrained by the configuration's nesting structure.

For advanced usage, check out the example (opens new window).

# 第9章 Passing Props to Route Components

Using $route in your component creates a tight coupling with the route which limits the flexibility of the component as it can only be used on certain URLs.

To decouple this component from the router use option props:

Instead of coupling to $route:

const User = {

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

}

const router = new VueRouter({

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

})

Decouple it by using 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 })

}

}

]

})

This allows you to use the component anywhere, which makes the component easier to reuse and test.

#Boolean mode

When props is set to true, the route.params will be set as the component props.

#Object mode

When props is an object, this will be set as the component props as-is. Useful for when the props are static.

const router = new VueRouter({

routes: [

{

path: '/promotion/from-newsletter',

component: Promotion,

props: { newsletterPopup: false }

}

]

})

#Function mode

You can create a function that returns props. This allows you to cast parameters into other types, combine static values with route-based values, etc.

const router = new VueRouter({

routes: [

{

path: '/search',

component: SearchUser,

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

}

]

})

The URL /search?q=vue would pass {query: 'vue'} as props to the SearchUser component.

Try to keep the props function stateless, as it's only evaluated on route changes. Use a wrapper component if you need state to define the props, that way vue can react to state changes.

For advanced usage, check out the example (opens new window).

# 第10章 HTML5 History Mode

The default mode for vue-router is hash mode - it uses the URL hash to simulate a full URL so that the page won't be reloaded when the URL changes.

To get rid of the hash, we can use the router's history mode, which leverages the history.pushState API to achieve URL navigation without a page reload:

const router = new VueRouter({

mode: 'history',

routes: [...]

})

When using history mode, the URL will look "normal," e.g. http://oursite.com/user/id. Beautiful!

Here comes a problem, though: Since our app is a single page client side app, without a proper server configuration, the users will get a 404 error if they access http://oursite.com/user/id directly in their browser. Now that's ugly.

Not to worry: To fix the issue, all you need to do is add a simple catch-all fallback route to your server. If the URL doesn't match any static assets, it should serve the same index.html page that your app lives in. Beautiful, again!

#Example Server Configurations

Note: The following examples assume you are serving your app from the root folder. If you deploy to a subfolder, you should use the publicPath option of Vue CLI (opens new window)and the related base property of the router (opens new window). You also need to adjust the examples below to use the subfolder instead of the root folder (e.g. replacing RewriteBase / with RewriteBase /name-of-your-subfolder/).

#Apache

<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>

Instead of mod\_rewrite, you could also use FallbackResource (opens new window).

#nginx

location / {

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

}

#Native 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)

})

#Express with Node.js

For Node.js/Express, consider using connect-history-api-fallback middleware (opens new window).

#Internet Information Services (IIS)

Install IIS UrlRewrite(opens new window)

Create a web.config file in the root directory of your site with the following:

<?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>

#Caddy v2

try\_files {path} /

#Caddy v1

rewrite {

regexp .\*

to {path} /

}

#Firebase hosting

Add this to your firebase.json:

{

"hosting": {

"public": "dist",

"rewrites": [

{

"source": "\*\*",

"destination": "/index.html"

}

]

}

}

#Caveat

There is a caveat to this: Your server will no longer report 404 errors as all not-found paths now serve up your index.html file. To get around the issue, you should implement a catch-all route within your Vue app to show a 404 page:

const router = new VueRouter({

mode: 'history',

routes: [

{

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

component: NotFoundComponent,

name: 'NotFound'

}

]

})

Alternatively, if you are using a Node.js server, you can implement the fallback by using the router on the server side to match the incoming URL and respond with 404 if no route is matched. Check out the Vue server side rendering documentation (opens new window)for more information.

# 第11章 Navigation Guards

As the name suggests, the navigation guards provided by vue-router are primarily used to guard navigations either by redirecting it or canceling it. There are a number of ways to hook into the route navigation process: globally, per-route, or in-component.

Remember that params or query changes won't trigger enter/leave navigation guards. You can either watch the $route object to react to those changes, or use the beforeRouteUpdate in-component guard.

#Global Before Guards

Learn how navigation guards works with a free lesson on Vue School

You can register global before guards using router.beforeEach:

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

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

// ...

})

Global before guards are called in creation order, whenever a navigation is triggered. Guards may be resolved asynchronously, and the navigation is considered pending before all hooks have been resolved.

Every guard function receives three arguments:

to: Route: the target Route Object being navigated to.

from: Route: the current route being navigated away from.

next: Function: this function must be called to resolve the hook. The action depends on the arguments provided to next:

next(): move on to the next hook in the pipeline. If no hooks are left, the navigation is confirmed.

next(false): abort the current navigation. If the browser URL was changed (either manually by the user or via back button), it will be reset to that of the from route.

next('/') or next({ path: '/' }): redirect to a different location. The current navigation will be aborted and a new one will be started. You can pass any location object to next, which allows you to specify options like replace: true, name: 'home' and any option used in router-link's to prop or router.push

next(error): (2.4.0+) if the argument passed to next is an instance of Error, the navigation will be aborted and the error will be passed to callbacks registered via router.onError().

Make sure that the next function is called exactly once in any given pass through the navigation guard. It can appear more than once, but only if the logical paths have no overlap, otherwise the hook will never be resolved or produce errors. Here is an example of redirecting to user to /login if they are not authenticated:

// 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()

})

#Global Resolve Guards

You can register a global guard with router.beforeResolve. This is similar to router.beforeEach, with the difference that resolve guards will be called right before the navigation is confirmed, after all in-component guards and async route components are resolved.

#Global After Hooks

You can also register global after hooks, however unlike guards, these hooks do not get a next function and cannot affect the navigation:

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

// ...

})

#Per-Route Guard

You can define beforeEnter guards directly on a route's configuration object:

const router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

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

// ...

}

}

]

})

These guards have the exact same signature as global before guards.

#In-Component Guards

Finally, you can directly define route navigation guards inside route components (the ones passed to the router configuration) with the following options:

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.

}

}

The beforeRouteEnter guard does NOT have access to this, because the guard is called before the navigation is confirmed, thus the new entering component has not even been created yet.

However, you can access the instance by passing a callback to next. The callback will be called when the navigation is confirmed, and the component instance will be passed to the callback as the argument:

beforeRouteEnter (to, from, next) {

next(vm => {

// access to component instance via `vm`

})

}

Note that beforeRouteEnter is the only guard that supports passing a callback to next. For beforeRouteUpdate and beforeRouteLeave, this is already available, so passing a callback is unnecessary and therefore not supported:

beforeRouteUpdate (to, from, next) {

// just use `this`

this.name = to.params.name

next()

}

The leave guard is usually used to prevent the user from accidentally leaving the route with unsaved edits. The navigation can be canceled by calling 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)

}

}

If you are using mixins that add in-component navigation guards, make sure to add the mixin after installing the router plugin:

Vue.use(Router)

Vue.mixin({

beforeRouteUpdate(to, from, next) {

// ...

}

})

#The Full Navigation Resolution Flow

Navigation triggered.

Call beforeRouteLeave guards in deactivated components.

Call global beforeEach guards.

Call beforeRouteUpdate guards in reused components.

Call beforeEnter in route configs.

Resolve async route components.

Call beforeRouteEnter in activated components.

Call global beforeResolve guards.

Navigation confirmed.

Call global afterEach hooks.

DOM updates triggered.

Call callbacks passed to next in beforeRouteEnter guards with instantiated instances.

# 第12章 Route Meta Fields

Sometimes, you might want to attach arbitrary information to routes like transition names, who can access the route, etc. This can be achieved through the meta property which accepts an object of properties and can be accessed on the route location and navigation guards. You can define meta properties like this:

const router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

children: [

{

path: 'bar',

component: Bar,

// a meta field

meta: { requiresAuth: true }

}

]

}

]

})

So how do we access this meta field?

First, each route object in the routes configuration is called a route record. Route records may be nested. Therefore when a route is matched, it can potentially match more than one route record.

For example, with the above route config, the URL /foo/bar will match both the parent route record and the child route record.

All route records matched by a route are exposed on the $route object (and also route objects in navigation guards) as the $route.matched Array. Therefore, we will need to iterate over $route.matched to check for meta fields in route records.

An example use case is checking for a meta field in the global navigation guard:

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 Transitions

Since the <router-view> is essentially a dynamic component, we can apply transition effects to it the same way using the <transition> component:

<transition>

<router-view></router-view>

</transition>

All transition APIs (opens new window)work the same here.

#Per-Route Transition

The above usage will apply the same transition for all routes. If you want each route's component to have different transitions, you can instead use <transition> with different names inside each route component:

const Foo = {

template: `

<transition name="slide">

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

</transition>

`

}

const Bar = {

template: `

<transition name="fade">

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

</transition>

`

}

#Route-Based Dynamic Transition

It is also possible to determine the transition to use dynamically based on the relationship between the target route and current route:

<!-- 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'

}

}

See full example here (opens new window).

# 第14章 Data Fetching

Sometimes you need to fetch data from the server when a route is activated. For example, before rendering a user profile, you need to fetch the user's data from the server. We can achieve this in two different ways:

Fetching After Navigation: perform the navigation first, and fetch data in the incoming component's lifecycle hook. Display a loading state while data is being fetched.

Fetching Before Navigation: Fetch data before navigation in the route enter guard, and perform the navigation after data has been fetched.

Technically, both are valid choices - it ultimately depends on the user experience you are aiming for.

#Fetching After Navigation

When using this approach, we navigate and render the incoming component immediately, and fetch data in the component's created hook. It gives us the opportunity to display a loading state while the data is being fetched over the network, and we can also handle loading differently for each view.

Let's assume we have a Post component that needs to fetch the data for a post based on $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

}

})

}

}

}

#Fetching Before Navigation

With this approach we fetch the data before actually navigating to the new route. We can perform the data fetching in the beforeRouteEnter guard in the incoming component, and only call next when the fetch is complete:

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

}

}

}

}

The user will stay on the previous view while the resource is being fetched for the incoming view. It is therefore recommended to display a progress bar or some kind of indicator while the data is being fetched. If the data fetch fails, it's also necessary to display some kind of global warning message.

# 第15章 Scroll Behavior

When using client-side routing, we may want to scroll to top when navigating to a new route, or preserve the scrolling position of history entries just like real page reload does. vue-router allows you to achieve these and even better, allows you to completely customize the scroll behavior on route navigation.

Note: this feature only works if the browser supports history.pushState.

When creating the router instance, you can provide the scrollBehavior function:

const router = new VueRouter({

routes: [...],

scrollBehavior (to, from, savedPosition) {

// return desired position

}

})

The scrollBehavior function receives the to and from route objects. The third argument, savedPosition, is only available if this is a popstate navigation (triggered by the browser's back/forward buttons).

The function can return a scroll position object. The object could be in the form of:

{ x: number, y: number }

{ selector: string, offset? : { x: number, y: number }} (offset only supported in 2.6.0+)

If a falsy value or an empty object is returned, no scrolling will happen.

For example:

scrollBehavior (to, from, savedPosition) {

return { x: 0, y: 0 }

}

This will simply make the page scroll to top for all route navigations.

Returning the savedPosition will result in a native-like behavior when navigating with back/forward buttons:

scrollBehavior (to, from, savedPosition) {

if (savedPosition) {

return savedPosition

} else {

return { x: 0, y: 0 }

}

}

If you want to simulate the "scroll to anchor" behavior:

scrollBehavior (to, from, savedPosition) {

if (to.hash) {

return {

selector: to.hash

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

}

}

}

We can also use route meta fields to implement fine-grained scroll behavior control. Check out a full example here (opens new window).

#Async Scrolling

New in 2.8.0

You can also return a Promise that resolves to the desired position descriptor:

scrollBehavior (to, from, savedPosition) {

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

setTimeout(() => {

resolve({ x: 0, y: 0 })

}, 500)

})

}

It's possible to hook this up with events from a page-level transition component to make the scroll behavior play nicely with your page transitions, but due to the possible variance and complexity in use cases, we simply provide this primitive to enable specific userland implementations.

#Smooth Scrolling

You can enable native smooth scrolling for browsers supporting it (opens new window)by simply adding the behavior option to the object returned inside scrollBehavior:

scrollBehavior (to, from, savedPosition) {

if (to.hash) {

return {

selector: to.hash,

behavior: 'smooth',

}

}

}

# 第16章 Lazy Loading Routes

When building apps with a bundler, the JavaScript bundle can become quite large, and thus affect the page load time. It would be more efficient if we can split each route's components into a separate chunk, and only load them when the route is visited.

Combining Vue's async component feature (opens new window)and webpack's code splitting feature (opens new window), it's trivially easy to lazy-load route components.

First, an async component can be defined as a factory function that returns a Promise (which should resolve to the component itself):

const Foo = () =>

Promise.resolve({

/\* component definition \*/

})

Second, in webpack 2, we can use the dynamic import (opens new window)syntax to indicate a code-split point:

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

Note

if you are using Babel, you will need to add the syntax-dynamic-import (opens new window)plugin so that Babel can properly parse the syntax.

Combining the two, this is how to define an async component that will be automatically code-split by webpack:

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

Nothing needs to change in the route config, just use Foo as usual:

const router = new VueRouter({

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

})

#Grouping Components in the Same Chunk

Sometimes we may want to group all the components nested under the same route into the same async chunk. To achieve that we need to use named chunks (opens new window)by providing a chunk name using a special comment syntax (requires 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 will group any async module with the same chunk name into the same async chunk.

# 第17章 Navigation Failures

When using router-link, Vue Router calls router.push to trigger a navigation. While the expected behavior for most links is to navigate a user to a new page, there are a few situations where users will remain on the same page:

Users are already on the page that they are trying to navigate to

A navigation guard aborts the navigation by calling next(false)

A navigation guard throws an error or calls next(new Error())

When using a router-link component, none of these failures will log an error. However, if you are using router.push or router.replace, you might come across an "Uncaught (in promise) Error" message followed by a more specific message in your console. Let's understand how to differentiate Navigation Failures.

Background story

In v3.2.0, Navigation Failures were exposed through the two optional callbacks of router.push: onComplete and onAbort. Since version 3.1.0, router.push and router.replace return a Promise if no onComplete/onAbort callback is provided. This Promise resolves instead of invoking onComplete and rejects instead of invoking onAbort.

#Detecting Navigation Failures

Navigation Failures are Error instances with a few extra properties. To check if an error comes from the Router, use the isNavigationFailure function:

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')

}

})

TIP

If you omit the second parameter: isNavigationFailure(failure), it will only check if the error is a Navigation Failure.

#NavigationFailureType

NavigationFailureType help developers to differentiate between the various types of Navigation Failures. There are four different types:

redirected: next(newLocation) was called inside of a navigation guard to redirect somewhere else.

aborted: next(false) was called inside of a navigation guard to the navigation.

cancelled: A new navigation completely took place before the current navigation could finish. e.g. router.push was called while waiting inside of a navigation guard.

duplicated: The navigation was prevented because we are already at the target location.

#Navigation Failures's properties

All navigation failures expose to and from properties to reflect the target and current location respectively for the navigation that failed:

// trying to access the admin page

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

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

failure.to.path // '/admin'

failure.from.path // '/'

}

})

In all cases, to and from are normalized route locations.