# Getting Started

**教程中的案例代码将使用**[**ES2015**](https://github.com/lukehoban/es6features)**来编写。**

用 Vue.js + vue-router 创建单页应用，是非常简单的。使用 Vue.js ，我们已经可以通过组合组件来组成应用程序，当你要把 vue-router 添加进来，我们需要做的是，将组件(components)映射到路由(routes)，然后告诉 vue-router 在哪里渲染它们。下面是个基本例子：

**所有的例子都将使用完整版的 Vue 以解析模板。更多细节请**[**移步这里**](https://cn.vuejs.org/v2/guide/installation.html#运行时-编译器-vs-只包含运行时)**。**

### 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>

<!-- 使用 router-link 组件来导航. -->

<!-- 通过传入 `to` 属性指定链接. -->

<!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->

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

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

</p>

<!-- 路由出口 -->

<!-- 路由匹配到的组件将渲染在这里 -->

<router-view></router-view>

</div>

### JavaScript

// 0. 如果使用模块化机制编程，導入Vue和VueRouter，要调用 Vue.use(VueRouter)

// 1. 定义（路由）组件。

// 可以从其他文件 import 进来

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

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

// 2. 定义路由

// 每个路由应该映射一个组件。 其中"component" 可以是

// 通过 Vue.extend() 创建的组件构造器，

// 或者，只是一个组件配置对象。

// 我们晚点再讨论嵌套路由。

const routes = [

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

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

]

// 3. 创建 router 实例，然后传 `routes` 配置

// 你还可以传别的配置参数, 不过先这么简单着吧。

const router = new VueRouter({

routes // （缩写）相当于 routes: routes

})

// 4. 创建和挂载根实例。

// 记得要通过 router 配置参数注入路由，

// 从而让整个应用都有路由功能

const app = new Vue({

router

}).$mount('#app')

// 现在，应用已经启动了！

你可以看看这个例子 [**live**](https://jsfiddle.net/yyx990803/xgrjzsup/)。

要注意，当 <router-link> 对应的路由匹配成功，将自动设置 class 属性值 .router-link-active。查看 [**API 文档**](https://router.vuejs.org/zh-cn/api/router-link.html)学习更多相关内容。

# 动态路由匹配

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件。例如，我们有一个 User 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 vue-router 的路由路径中使用『动态路径参数』（dynamic segment）来达到这个效果：

const User = {

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

}

const router = new VueRouter({

routes: [

// 动态路径参数 以冒号开头

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

]

})

现在呢，像 /user/foo 和 /user/bar 都将映射到相同的路由。

一个『路径参数』使用冒号 : 标记。当匹配到一个路由时，参数值会被设置到 this.$route.params，可以在每个组件内使用。于是，我们可以更新 User 的模板，输出当前用户的 ID：

const User = {

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

}

你可以看看这个[**在线例子**](https://jsfiddle.net/yyx990803/4xfa2f19/)。

你可以在一个路由中设置多段『路径参数』，对应的值都会设置到 $route.params 中。例如：

| **模式** | **匹配路径** | **$route.params** |
| --- | --- | --- |
| /user/:username | /user/evan | { username: 'evan' } |
| /user/:username/post/:post\_id | /user/evan/post/123 | { username: 'evan', post\_id: 123 } |

除了 $route.params 外，$route 对象还提供了其它有用的信息，例如，$route.query（如果 URL 中有查询参数）、$route.hash 等等。你可以查看 [**API 文档**](https://router.vuejs.org/zh-cn/api/route-object.html) 的详细说明。

### 响应路由参数的变化

提醒一下，当使用路由参数时，例如从 /user/foo 导航到 user/bar，**原来的组件实例会被复用**。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。**不过，这也意味着组件的生命周期钩子不会再被调用**。

复用组件时，想对路由参数的变化作出响应的话，你可以简单地 watch（监测变化） $route 对象：

const User = {

template: '...',

watch: {

'$route' (to, from) {

// 对路由变化作出响应...

}

}

}

或者使用 2.2 中引入的 beforeRouteUpdate 守卫：

const User = {

template: '...',

beforeRouteUpdate (to, from, next) {

// react to route changes...

// don't forget to call next()

}

}

### 高级匹配模式

vue-router 使用 [**path-to-regexp**](https://github.com/pillarjs/path-to-regexp) 作为路径匹配引擎，所以支持很多高级的匹配模式，例如：可选的动态路径参数、匹配零个或多个、一个或多个，甚至是自定义正则匹配。查看它的 [**文档**](https://github.com/pillarjs/path-to-regexp#parameters) 学习高阶的路径匹配，还有 [**这个例子**](https://github.com/vuejs/vue-router/blob/next/examples/route-matching/app.js)展示 vue-router 怎么使用这类匹配。

### 匹配优先级

有时候，同一个路径可以匹配多个路由，此时，匹配的优先级就按照路由的定义顺序：谁先定义的，谁的优先级就最高。

# 嵌套路由

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各层组件，例如：

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

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

| User | | User |

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

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

| | | | | | | |

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

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

借助 vue-router，使用嵌套路由配置，就可以很简单地表达这种关系。

接着上节创建的 app：

<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 }

]

})

这里的 <router-view> 是最顶层的出口，渲染最高级路由匹配到的组件。同样地，一个被渲染组件同样可以包含自己的嵌套 <router-view>。例如，在 User 组件的模板添加一个 <router-view>：

const User = {

template: `

<div class="user">

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

<router-view></router-view>

</div>

`

}

要在嵌套的出口中渲染组件，需要在 VueRouter 的参数中使用 children 配置：

const router = new VueRouter({

routes: [

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

children: [

{

// 当 /user/:id/profile 匹配成功，

// UserProfile 会被渲染在 User 的 <router-view> 中

path: 'profile',

component: UserProfile

},

{

// 当 /user/:id/posts 匹配成功

// UserPosts 会被渲染在 User 的 <router-view> 中

path: 'posts',

component: UserPosts

}

]

}

]

})

**要注意，以 / 开头的嵌套路径会被当作根路径。 这让你充分的使用嵌套组件而无须设置嵌套的路径。**

你会发现，children 配置就是像 routes 配置一样的路由配置数组，所以呢，你可以嵌套多层路由。

此时，基于上面的配置，当你访问 /user/foo 时，User 的出口是不会渲染任何东西，这是因为没有匹配到合适的子路由。如果你想要渲染点什么，可以提供一个 空的 子路由：

const router = new VueRouter({

routes: [

{

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

children: [

// 当 /user/:id 匹配成功，

// UserHome 会被渲染在 User 的 <router-view> 中

{ path: '', component: UserHome },

// ...其他子路由

]

}

]

})

提供以上案例的可运行代码请[**移步这里**](https://jsfiddle.net/yyx990803/L7hscd8h/)。

# 编程式的导航

除了使用 <router-link> 创建 a 标签来定义导航链接，我们还可以借助 router 的实例方法，通过编写代码来实现。

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

**注意：在 Vue 实例内部，你可以通过 $router 访问路由实例。因此你可以调用 this.$router.push。**

想要导航到不同的 URL，则使用 router.push 方法。这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，则回到之前的 URL。

当你点击 <router-link> 时，这个方法会在内部调用，所以说，点击 <router-link :to="..."> 等同于调用 router.push(...)。

| **声明式** | **编程式** |
| --- | --- |
| <router-link :to="..."> | router.push(...) |

该方法的参数可以是一个字符串路径，或者一个描述地址的对象。例如：

// 字符串

router.push('home')

// 对象

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

// 命名的路由

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

// 带查询参数，变成 /register?plan=private

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

**注意：如果提供了 path，params 会被忽略，上述例子中的 query 并不属于这种情况。取而代之的是下面例子的做法，你需要提供路由的 name 或手写完整的带有参数的 path：**

const userId = 123

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

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

// 这里的 params 不生效

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

同样的规则也适用于 router-link 组件的 to 属性。

在 2.2.0+，可选的在 router.push 或 router.replace 中提供 onComplete 和 onAbort 回调作为第二个和第三个参数。这些回调将会在导航成功完成 (在所有的异步钩子被解析之后) 或终止 (导航到相同的路由、或在当前导航完成之前导航到另一个不同的路由) 的时候进行相应的调用。

**注意：**如果目的地和当前路由相同，只有参数发生了改变 (比如从一个用户资料到另一个 /users/1 -> /users/2)，你需要使用 [**beforeRouteUpdate**](https://router.vuejs.org/zh-cn/essentials/dynamic-matching.html#响应路由参数的变化) 来响应这个变化 (比如抓取用户信息)。

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

跟 router.push 很像，唯一的不同就是，它不会向 history 添加新记录，而是跟它的方法名一样 —— 替换掉当前的 history 记录。

| **声明式** | **编程式** |
| --- | --- |
| <router-link :to="..." replace> | router.replace(...) |

#### router.go(n)

这个方法的参数是一个整数，意思是在 history 记录中向前或者后退多少步，类似 window.history.go(n)。

例子

// 在浏览器记录中前进一步，等同于 history.forward()

router.go(1)

// 后退一步记录，等同于 history.back()

router.go(-1)

// 前进 3 步记录

router.go(3)

// 如果 history 记录不够用，那就默默地失败呗

router.go(-100)

router.go(100)

#### 操作 History

你也许注意到 router.push、 router.replace 和 router.go 跟 [**window.history.pushState、 window.history.replaceState 和 window.history.go**](https://developer.mozilla.org/en-US/docs/Web/API/History)好像， 实际上它们确实是效仿 window.history API 的。

因此，如果你已经熟悉 [**Browser History APIs**](https://developer.mozilla.org/en-US/docs/Web/API/History_API)，那么在 vue-router 中操作 history 就是超级简单的。

还有值得提及的，vue-router 的导航方法 （push、 replace、 go） 在各类路由模式（history、 hash 和 abstract）下表现一致。

# 命名路由

有时候，通过一个名称来标识一个路由显得更方便一些，特别是在链接一个路由，或者是执行一些跳转的时候。你可以在创建 Router 实例的时候，在 routes 配置中给某个路由设置名称。

const router = new VueRouter({

routes: [

{

path: '/user/:userId',

name: 'user',

component: User

}

]

})

要链接到一个命名路由，可以给 router-link 的 to 属性传一个对象：

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

这跟代码调用 router.push() 是一回事：

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

这两种方式都会把路由导航到 /user/123 路径。

完整的例子请[**移步这里**](https://github.com/vuejs/vue-router/blob/next/examples/named-routes/app.js)。

# 命名视图

有时候想同时（同级）展示多个视图，而不是嵌套展示，例如创建一个布局，有 sidebar（侧导航） 和 main（主内容） 两个视图，这个时候命名视图就派上用场了。你可以在界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。

<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>

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 components 配置（带上 s）：

const router = new VueRouter({

routes: [

{

path: '/',

components: {

default: Foo,

a: Bar,

b: Baz

}

}

]

})

# 重定向 和 别名

### 重定向

重定向也是通过 routes 配置来完成，下面例子是从 /a 重定向到 /b：

const router = new VueRouter({

routes: [

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

]

})

重定向的目标也可以是一个命名的路由：

const router = new VueRouter({

routes: [

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

]

})

甚至是一个方法，动态返回重定向目标：

const router = new VueRouter({

routes: [

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

// 方法接收 目标路由 作为参数

// return 重定向的 字符串路径/路径对象

}}

]

})

其它高级用法，请参考[**例子**](https://github.com/vuejs/vue-router/blob/next/examples/redirect/app.js)。

### 别名

『重定向』的意思是，当用户访问 /a时，URL 将会被替换成 /b，然后匹配路由为 /b，那么『别名』又是什么呢？

**/a 的别名是 /b，意味着，当用户访问 /b 时，URL 会保持为 /b，但是路由匹配则为 /a，就像用户访问 /a 一样。**

上面对应的路由配置为：

const router = new VueRouter({

routes: [

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

]

})

『别名』的功能让你可以自由地将 UI 结构映射到任意的 URL，而不是受限于配置的嵌套路由结构。

更多高级用法，请查看[**例子**](https://github.com/vuejs/vue-router/blob/next/examples/route-alias/app.js)。

# 路由组件传参

在组件中使用$route会使之与其对应路由形成高度耦合，从而使组件只能在某些特定的url上使用，限制了其灵活性。

使用props将组件和路由解耦：

**与$route耦合**

const User = {

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

}

const router = new VueRouter({

routes: [

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

]

})

**使用props解耦**

const User = {

props: ['id'],

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

}

const router = new VueRouter({

routes: [

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

// 对于包含命名视图的路由，你必须分别为每个命名视图添加props选项：

{

path: '/user/:id',

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

props: { default: true, sidebar: false }

}

]

})

这样你便可以在任何地方使用该组件，使得该组件更易于重用和测试。

### 布尔模式

如果props被设置为true，route.params将会被设置为组件属性。

### 对象模式

如果props是一个对象，它会被按原样设置为组件属性。当props是静态的时候有用。

const router = new VueRouter({

routes: [

{ path: '/promotion/from-newsletter', component: Promotion, props: { newsletterPopup: false } }

]

})

### 函数模式

你可以创建一个函数返回props。这样你便可以将参数转换成另一种类型，将静态值与基于路由的值结合等等。

const router = new VueRouter({

routes: [

{ path: '/search', component: SearchUser, props: (route) => ({ query: route.query.q }) }

]

})

Url: /search?q=vue 会将 {query: "vue"} 作为属性传递给SearchUser组件。

请尽可能保持props函数为无状态的，因为它只会在路由发生变化时起作用。如果你需要状态来定义props，请使用包装组件，这样vue才可以对状态变化做出反应。

更多高级用法，请查看[**例子**](https://github.com/vuejs/vue-router/blob/dev/examples/route-props/app.js)。

# HTML5 History 模式

vue-router 默认 hash 模式 —— 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。

如果不想要很丑的 hash，我们可以用路由的 **history 模式**，这种模式充分利用 history.pushState API 来完成 URL 跳转而无须重新加载页面。

const router = new VueRouter({

mode: 'history',

routes: [...]

})

当你使用 history 模式时，URL 就像正常的 url，例如 http://yoursite.com/user/id，也好看！

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 http://oursite.com/user/id 就会返回 404，这就不好看了。

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

## 后端配置例子

#### Apache

<IfModule mod\_rewrite.c>

RewriteEngine On

RewriteBase /

RewriteRule ^index\.html$ - [L]

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule . /index.html [L]

</IfModule>

#### nginx

location / {

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

}

#### 原生 Node.js

const http = require('http')

const fs = require('fs')

const httpPort = 80

http.createServer((req, res) => {

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

if (err) {

console.log('We cannot open 'index.htm' 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)

})

#### 基于 Node.js 的 Express

对于 Node.js/Express，请考虑使用 [**connect-history-api-fallback 中间件**](https://github.com/bripkens/connect-history-api-fallback)。

#### Internet Information Services (IIS)

1. 安装 [**IIS UrlRewrite**](https://www.iis.net/downloads/microsoft/url-rewrite)
2. 在你的网站根目录中创建一个 web.config 文件，内容如下：

<?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

rewrite {

regexp .\*

to {path} /

}

## 警告

给个警告，因为这么做以后，你的服务器就不再返回 404 错误页面，因为对于所有路径都会返回 index.html 文件。为了避免这种情况，你应该在 Vue 应用里面覆盖所有的路由情况，然后在给出一个 404 页面。

const router = new VueRouter({

mode: 'history',

routes: [

{ path: '\*', component: NotFoundComponent }

]

})

或者，如果你使用 Node.js 服务器，你可以用服务端路由匹配到来的 URL，并在没有匹配到路由的时候返回 404，以实现回退。更多详情请查阅 [**Vue 服务端渲染文档**](https://ssr.vuejs.org/zh/)。

# 导航守卫

**（译者：『导航』表示路由正在发生改变。）**

正如其名，vue-router 提供的导航守卫主要用来通过跳转或取消的方式守卫导航。有多种机会植入路由导航过程中：全局的, 单个路由独享的, 或者组件级的。

记住**参数或查询的改变并不会触发进入/离开的导航守卫**。你可以通过[**观察 $route 对象**](https://router.vuejs.org/zh-cn/essentials/dynamic-matching.html#响应路由参数的变化)来应对这些变化，或使用 beforeRouteUpdate 的组件内守卫。

### 全局守卫

你可以使用 router.beforeEach 注册一个全局前置守卫：

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

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

// ...

})

当一个导航触发时，全局前置守卫按照创建顺序调用。守卫是异步解析执行，此时导航在所有守卫 resolve 完之前一直处于 **等待中**。

每个守卫方法接收三个参数：

* **to: Route**: 即将要进入的目标 [**路由对象**](https://router.vuejs.org/zh-cn/api/route-object.html)
* **from: Route**: 当前导航正要离开的路由
* **next: Function**: 一定要调用该方法来 **resolve** 这个钩子。执行效果依赖 next 方法的调用参数。
  + **next()**: 进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 **confirmed** （确认的）。
  + **next(false)**: 中断当前的导航。如果浏览器的 URL 改变了（可能是用户手动或者浏览器后退按钮），那么 URL 地址会重置到 from 路由对应的地址。
  + **next('/') 或者 next({ path: '/' })**: 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。
  + **next(error)**: (2.4.0+) 如果传入 next 的参数是一个 Error 实例，则导航会被终止且该错误会被传递给 router.onError() 注册过的回调。

**确保要调用 next 方法，否则钩子就不会被 resolved。**

### 全局解析守卫

**2.5.0 新增**

在 2.5.0+ 你可以用 router.beforeResolve 注册一个全局守卫。这和 router.beforeEach 类似，区别是在导航被确认之前，**同时在所有组件内守卫和异步路由组件被解析之后**，解析守卫就被调用。

### 全局后置钩子

你也可以注册全局后置钩子，然而和守卫不同的是，这些钩子不会接受 next 函数也不会改变导航本身：

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

// ...

})

### 路由独享的守卫

你可以在路由配置上直接定义 beforeEnter 守卫：

const router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

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

// ...

}

}

]

})

这些守卫与全局前置守卫的方法参数是一样的。

### 组件内的守卫

最后，你可以在路由组件内直接定义以下路由导航守卫：

* beforeRouteEnter
* beforeRouteUpdate (2.2 新增)
* beforeRouteLeave

const Foo = {

template: `...`,

beforeRouteEnter (to, from, next) {

// 在渲染该组件的对应路由被 confirm 前调用

// 不！能！获取组件实例 `this`

// 因为当守卫执行前，组件实例还没被创建

},

beforeRouteUpdate (to, from, next) {

// 在当前路由改变，但是该组件被复用时调用

// 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，

// 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。

// 可以访问组件实例 `this`

},

beforeRouteLeave (to, from, next) {

// 导航离开该组件的对应路由时调用

// 可以访问组件实例 `this`

}

}

beforeRouteEnter 守卫 **不能** 访问 this，因为守卫在导航确认前被调用,因此即将登场的新组件还没被创建。

不过，你可以通过传一个回调给 next来访问组件实例。在导航被确认的时候执行回调，并且把组件实例作为回调方法的参数。

beforeRouteEnter (to, from, next) {

next(vm => {

// 通过 `vm` 访问组件实例

})

}

你可以 在 beforeRouteLeave 中直接访问 this。这个离开守卫通常用来禁止用户在还未保存修改前突然离开。可以通过 next(false) 来取消导航。

### 完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用离开守卫。
3. 调用全局的 beforeEach 守卫。
4. 在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。
5. 在路由配置里调用 beforeEnter。
6. 解析异步路由组件。
7. 在被激活的组件里调用 beforeRouteEnter。
8. 调用全局的 beforeResolve 守卫 (2.5+)。
9. 导航被确认。
10. 调用全局的 afterEach 钩子。
11. 触发 DOM 更新。
12. 用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

# 路由元信息

定义路由的时候可以配置 meta 字段：

const router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

children: [

{

path: 'bar',

component: Bar,

// a meta field

meta: { requiresAuth: true }

}

]

}

]

})

那么如何访问这个 meta 字段呢？

首先，我们称呼 routes 配置中的每个路由对象为 **路由记录**。路由记录可以是嵌套的，因此，当一个路由匹配成功后，他可能匹配多个路由记录

例如，根据上面的路由配置，/foo/bar 这个 URL 将会匹配父路由记录以及子路由记录。

一个路由匹配到的所有路由记录会暴露为 $route 对象（还有在导航守卫中的路有对象）的 $route.matched 数组。因此，我们需要遍历 $route.matched 来检查路由记录中的 meta 字段。

下面例子展示在全局导航守卫中检查元字段：

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() // 确保一定要调用 next()

}

})

# 过渡动效

<router-view> 是基本的动态组件，所以我们可以用 <transition> 组件给它添加一些过渡效果：

<transition>

<router-view></router-view>

</transition>

[**<transition> 的所有功能**](https://cn.vuejs.org/guide/transitions.html) 在这里同样适用。

### 单个路由的过渡

上面的用法会给所有路由设置一样的过渡效果，如果你想让每个路由组件有各自的过渡效果，可以在各路由组件内使用 <transition> 并设置不同的 name。

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

还可以基于当前路由与目标路由的变化关系，动态设置过渡效果：

<!-- 使用动态的 transition name -->

<transition :name="transitionName">

<router-view></router-view>

</transition>

// 接着在父组件内

// watch $route 决定使用哪种过渡

watch: {

'$route' (to, from) {

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

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

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

}

}

查看完整例子请[**移步这里**](https://github.com/vuejs/vue-router/blob/next/examples/transitions/app.js)。

# 数据获取

有时候，进入某个路由后，需要从服务器获取数据。例如，在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现：

* **导航完成之后获取**：先完成导航，然后在接下来的组件生命周期钩子中获取数据。在数据获取期间显示『加载中』之类的指示。
* **导航完成之前获取**：导航完成前，在路由进入的守卫中获取数据，在数据获取成功后执行导航。

从技术角度讲，两种方式都不错 —— 就看你想要的用户体验是哪种。

## 导航完成后获取数据

当你使用这种方式时，我们会马上导航和渲染组件，然后在组件的 created 钩子中获取数据。这让我们有机会在数据获取期间展示一个 loading 状态，还可以在不同视图间展示不同的 loading 状态。

假设我们有一个 Post 组件，需要基于 $route.params.id 获取文章数据：

<template>

<div class="post">

<div class="loading" v-if="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 () {

// 组件创建完后获取数据，

// 此时 data 已经被 observed 了

this.fetchData()

},

watch: {

// 如果路由有变化，会再次执行该方法

'$route': 'fetchData'

},

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

}

})

}

}

}

## 在导航完成前获取数据

通过这种方式，我们在导航转入新的路由前获取数据。我们可以在接下来的组件的 beforeRouteEnter 守卫中获取数据，当数据获取成功后只调用 next 方法。

export default {

data () {

return {

post: null,

error: null

}

},

beforeRouteEnter (to, from, next) {

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

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

})

},

// 路由改变前，组件就已经渲染完了

// 逻辑稍稍不同

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

}

}

}

}

在为后面的视图获取数据时，用户会停留在当前的界面，因此建议在数据获取期间，显示一些进度条或者别的指示。如果数据获取失败，同样有必要展示一些全局的错误提醒。

**滚动行为**

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。 vue-router 能做到，而且更好，它让你可以自定义路由切换时页面如何滚动。

**注意: 这个功能只在 HTML5 history 模式下可用。**

当创建一个 Router 实例，你可以提供一个 scrollBehavior 方法：

const router = new VueRouter({

routes: [...],

scrollBehavior (to, from, savedPosition) {

// return 期望滚动到哪个的位置

}

})

scrollBehavior 方法接收 to 和 from 路由对象。第三个参数 savedPosition 当且仅当 popstate 导航 (通过浏览器的 前进/后退 按钮触发) 时才可用。

这个方法返回滚动位置的对象信息，长这样：

* { x: number, y: number }
* { selector: string, offset? : { x: number, y: number }} (offset 只在 2.6.0+ 支持)

如果返回一个 falsy (译者注：falsy 不是 false，[**参考这里**](https://developer.mozilla.org/zh-CN/docs/Glossary/Falsy))的值，或者是一个空对象，那么不会发生滚动。

举例：

scrollBehavior (to, from, savedPosition) {

return { x: 0, y: 0 }

}

对于所有路由导航，简单地让页面滚动到顶部。

返回 savedPosition，在按下 后退/前进 按钮时，就会像浏览器的原生表现那样：

scrollBehavior (to, from, savedPosition) {

if (savedPosition) {

return savedPosition

} else {

return { x: 0, y: 0 }

}

}

如果你要模拟『滚动到锚点』的行为：

scrollBehavior (to, from, savedPosition) {

if (to.hash) {

return {

selector: to.hash

}

}

}

我们还可以利用[**路由元信息**](https://router.vuejs.org/zh-cn/advanced/meta.html)更细颗粒度地控制滚动。查看完整例子请[**移步这里**](https://github.com/vuejs/vue-router/blob/next/examples/scroll-behavior/app.js)。

# 路由懒加载

当打包构建应用时，Javascript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了。

结合 Vue 的[**异步组件**](https://cn.vuejs.org/guide/components.html#异步组件)和 Webpack 的[**代码分割功能**](https://doc.webpack-china.org/guides/code-splitting-async/#require-ensure-/)，轻松实现路由组件的懒加载。

首先，可以将异步组件定义为返回一个 Promise 的工厂函数 (该函数返回的 Promise 应该 resolve 组件本身)：

const Foo = () => Promise.resolve({ /\* 组件定义对象 \*/ })

第二，在 Webpack 2 中，我们可以使用[**动态 import**](https://github.com/tc39/proposal-dynamic-import)语法来定义代码分块点 (split point)：

import('./Foo.vue') // 返回 Promise

**注意：如果您使用的是 Babel，你将需要添加**[**syntax-dynamic-import**](https://babeljs.io/docs/plugins/syntax-dynamic-import/)**插件，才能使 Babel 可以正确地解析语法。**

结合这两者，这就是如何定义一个能够被 Webpack 自动代码分割的异步组件。

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

在路由配置中什么都不需要改变，只需要像往常一样使用 Foo：

const router = new VueRouter({

routes: [

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

]

})

### 把组件按组分块

有时候我们想把某个路由下的所有组件都打包在同个异步块 (chunk) 中。只需要使用 [**命名 chunk**](https://webpack.js.org/guides/code-splitting-require/#chunkname)，一个特殊的注释语法来提供 chunk name (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 会将任何一个异步模块与相同的块名称组合到相同的异步块中。

# Router 构造配置

### routes

* 类型: Array<RouteConfig>

RouteConfig 的类型定义：

declare type RouteConfig = {

path: string;

component?: Component;

name?: string; // 命名路由

components?: { [name: string]: Component }; // 命名视图组件

redirect?: string | Location | Function;

props?: boolean | string | Function;

alias?: string | Array<string>;

children?: Array<RouteConfig>; // 嵌套路由

beforeEnter?: (to: Route, from: Route, next: Function) => void;

meta?: any;

// 2.6.0+

caseSensitive?: boolean; // 匹配规则是否大小写敏感？(默认值：false)

pathToRegexpOptions?: Object; // 编译正则的选项

}

### mode

* 类型: string
* 默认值: "hash" (浏览器环境) | "abstract" (Node.js 环境)
* 可选值: "hash" | "history" | "abstract"

配置路由模式:

* + hash: 使用 URL hash 值来作路由。支持所有浏览器，包括不支持 HTML5 History Api 的浏览器。
  + history: 依赖 HTML5 History API 和服务器配置。查看 [**HTML5 History 模式**](https://router.vuejs.org/zh-cn/essentials/history-mode.html)。
  + abstract: 支持所有 JavaScript 运行环境，如 Node.js 服务器端。**如果发现没有浏览器的 API，路由会自动强制进入这个模式。**

### base

* 类型: string
* 默认值: "/"

应用的基路径。例如，如果整个单页应用服务在 /app/ 下，然后 base 就应该设为 "/app/"。

### linkActiveClass

* 类型: string
* 默认值: "router-link-active"

全局配置 <router-link> 的默认『激活 class 类名』。参考 [**router-link**](https://router.vuejs.org/zh-cn/api/router-link.html)。

### linkExactActiveClass

**2.5.0+**

* 类型: string
* 默认值: "router-link-exact-active"

全局配置 <router-link> 精确激活的默认的 class。可同时翻阅 [**router-link**](https://router.vuejs.org/zh-cn/api/router-link.html)。

### scrollBehavior

* 类型: Function

签名:

(

to: Route,

from: Route,

savedPosition?: { x: number, y: number }

) => { x: number, y: number } | { selector: string } | ?{}

更多详情参考[**滚动行为**](https://router.vuejs.org/zh-cn/advanced/scroll-behavior.html)。

### parseQuery / stringifyQuery

**2.4.0+**

* 类型: Function

提供自定义查询字符串的解析/反解析函数。覆盖默认行为。

### fallback

**2.6.0+**

* 类型: boolean

当浏览器不支持 history.pushState 控制路由是否应该回退到 hash 模式。默认值为 true。

在 IE9 中，设置为 false 会使得每个 router-link 导航都触发整页刷新。它可用于工作在 IE9 下的服务端渲染应用，因为一个 hash 模式的 URL 并不支持服务端渲染。

# Router 实例

### 属性

#### router.app

* 类型: Vue instance

配置了 router 的 Vue 根实例。

#### router.mode

* 类型: string

路由使用的[**模式**](https://router.vuejs.org/zh-cn/api/options.html#mode)。

#### router.currentRoute

* 类型: Route

当前路由对应的[**路由信息对象**](https://router.vuejs.org/zh-cn/api/route-object.html)。

### 方法

* **router.beforeEach(guard)**
* **router.beforeResolve(guard) (2.5.0+)**: 此时异步组件已经加载完成
* **router.afterEach(hook)**

增加全局的导航守卫。参考[**导航守卫**](https://router.vuejs.org/zh-cn/advanced/navigation-guards.html)。

在 2.5.0+ 这三个方法都返回一个移除已注册的守卫/钩子的函数。

* **router.push(location, onComplete?, onAbort?)**
* **router.replace(location, onComplete?, onAbort?)**
* **router.go(n)**
* **router.back()**
* **router.forward()**

动态的导航到一个新 URL。参考[**编程式导航**](https://router.vuejs.org/zh-cn/essentials/navigation.html)。

* **router.getMatchedComponents(location?)**

返回目标位置或是当前路由匹配的组件数组（是数组的定义/构造类，不是实例）。通常在服务端渲染的数据预加载时时候。

* **router.resolve(location, current?, append?)**

**2.1.0+**

解析目标位置（格式和 <router-link> 的 to prop 一样），返回包含如下属性的对象：

{

location: Location;

route: Route;

href: string;

}

* + current 是当前默认的路由 (通常你不需要改变它)
  + append 允许你在 current 路由上附加路径 (如同 [**router-link**](https://router.vuejs.org/zh-cn/api/router-link.html#props))
* **router.addRoutes(routes)**

**2.2.0+**

动态添加更多的路由规则。参数必须是一个符合 routes 选项要求的数组。

* **router.onReady(callback, [errorCallback])**

**2.2.0+**

该方法把一个回调排队，在路由完成初始导航时调用，这意味着它可以解析所有的异步进入钩子和路由初始化相关联的异步组件。

这可以有效确保服务端渲染时服务端和客户端输出的一致。

第二个参数 errorCallback 只在 2.4+ 支持。它会在初始化路由解析运行出错 (比如解析一个异步组件失败) 时被调用。

* **router.onError(callback)**

**2.4.0+**

注册一个回调，该回调会在路由导航过程中出错时被调用。注意被调用的错误必须是下列情形中的一种：

* + 错误在一个路由守卫函数中被同步抛出；
  + 错误在一个路由守卫函数中通过调用 next(err) 的方式异步捕获并处理；
  + 渲染一个路由的过程中，需要尝试解析一个异步组件时发生错误。

# 路由信息对象

一个 **route object（路由信息对象）** 表示当前激活的路由的状态信息，包含了当前 URL 解析得到的信息，还有 URL 匹配到的 **route records（路由记录）**。

route object 是 immutable（不可变） 的，每次成功的导航后都会产生一个新的对象。

route object 出现在多个地方:

* 在组件内，即 this.$route
* 在 $route 观察者回调内
* router.match(location) 的返回值
* 导航守卫的参数：
* router.beforeEach((to, from, next) => {
* // to 和 from 都是 路由信息对象

})

* + scrollBehavior 方法的参数:

const router = new VueRouter({

scrollBehavior (to, from, savedPosition) {

// to 和 from 都是 路由信息对象

}

})

### 路由信息对象的属性

* **$route.path**
  + 类型: string

字符串，对应当前路由的路径，总是解析为绝对路径，如 "/foo/bar"。

* **$route.params**
  + 类型: Object

一个 key/value 对象，包含了 动态片段 和 全匹配片段，如果没有路由参数，就是一个空对象。

* **$route.query**
  + 类型: Object

一个 key/value 对象，表示 URL 查询参数。例如，对于路径 /foo?user=1，则有 $route.query.user == 1，如果没有查询参数，则是个空对象。

* **$route.hash**
  + 类型: string

当前路由的 hash 值 (带 #) ，如果没有 hash 值，则为空字符串。

* **$route.fullPath**
  + 类型: string

完成解析后的 URL，包含查询参数和 hash 的完整路径。

* **$route.matched**
  + 类型: Array<RouteRecord>

一个数组，包含当前路由的所有嵌套路径片段的 **路由记录** 。路由记录就是 routes 配置数组中的对象副本（还有在 children 数组）。

const router = new VueRouter({

routes: [

// 下面的对象就是 route record

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

children: [

// 这也是个 route record

{ path: 'bar', component: Bar }

]

}

]

})

当 URL 为 /foo/bar，$route.matched 将会是一个包含从上到下的所有对象（副本）。

* **$route.name**

# 对组件注入

### 注入的属性

通过在 Vue 根实例的 router 配置传入 router 实例，下面这些属性成员会被注入到每个子组件。

#### $router

router 实例。

#### $route

当前激活的[**路由信息对象**](https://router.vuejs.org/zh-cn/api/route-object.html)。这个属性是只读的，里面的属性是 immutable（不可变） 的，不过你可以 watch（监测变化）它。

### 允许的额外配置

* **beforeRouteEnter**
* **beforeRouteUpdate** (在 2.2 加入)
* **beforeRouteLeave**

查看[**组件内的守卫**](https://router.vuejs.org/zh-cn/advanced/navigation-guards.html#组件内的守卫)。

# <router-link>

<router-link> 组件支持用户在具有路由功能的应用中（点击）导航。 通过 to 属性指定目标地址，默认渲染成带有正确链接的 <a> 标签，可以通过配置 tag 属性生成别的标签.。另外，当目标路由成功激活时，链接元素自动设置一个表示激活的 CSS 类名。

<router-link> 比起写死的 <a href="..."> 会好一些，理由如下：

* 无论是 HTML5 history 模式还是 hash 模式，它的表现行为一致，所以，当你要切换路由模式，或者在 IE9 降级使用 hash 模式，无须作任何变动。
* 在 HTML5 history 模式下，router-link 会守卫点击事件，让浏览器不再重新加载页面。
* 当你在 HTML5 history 模式下使用 base 选项之后，所有的 to 属性都不需要写（基路径）了。

### Props

* **to**
  + 类型: string | Location
  + required

表示目标路由的链接。当被点击后，内部会立刻把 to 的值传到 router.push()，所以这个值可以是一个字符串或者是描述目标位置的对象。

<!-- 字符串 -->

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

<!-- 渲染结果 -->

<a href="home">Home</a>

<!-- 使用 v-bind 的 JS 表达式 -->

<router-link v-bind:to="'home'">Home</router-link>

<!-- 不写 v-bind 也可以，就像绑定别的属性一样 -->

<router-link :to="'home'">Home</router-link>

<!-- 同上 -->

<router-link :to="{ path: 'home' }">Home</router-link>

<!-- 命名的路由 -->

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

<!-- 带查询参数，下面的结果为 /register?plan=private -->

<router-link :to="{ path: 'register', query: { plan: 'private' }}">Register</router-link>

* **replace**
* 类型: boolean
* 默认值: false

设置 replace 属性的话，当点击时，会调用 router.replace() 而不是 router.push()，于是导航后不会留下 history 记录。

<router-link :to="{ path: '/abc'}" replace></router-link>

* **append**
* 类型: boolean
* 默认值: false

设置 append 属性后，则在当前（相对）路径前添加基路径。例如，我们从 /a 导航到一个相对路径 b，如果没有配置 append，则路径为 /b，如果配了，则为 /a/b

<router-link :to="{ path: 'relative/path'}" append></router-link>

* **tag**
* 类型: string
* 默认值: "a"

有时候想要 <router-link> 渲染成某种标签，例如 <li>。 于是我们使用 tag prop 类指定何种标签，同样它还是会监听点击，触发导航。

<router-link to="/foo" tag="li">foo</router-link>

<!-- 渲染结果 -->

<li>foo</li>

* **active-class**
* 类型: string
* 默认值: "router-link-active"

设置 链接激活时使用的 CSS 类名。默认值可以通过路由的构造选项 linkActiveClass 来全局配置。

* **exact**
  + 类型: boolean
  + 默认值: false

"是否激活" 默认类名的依据是 **inclusive match** （全包含匹配）。 举个例子，如果当前的路径是 /a 开头的，那么 <router-link to="/a"> 也会被设置 CSS 类名。

按照这个规则，<router-link to="/"> 将会点亮各个路由！想要链接使用 "exact 匹配模式"，则使用 exact 属性：

<!-- 这个链接只会在地址为 / 的时候被激活 -->

<router-link to="/" exact>

查看更多关于激活链接类名的例子[**可运行**](https://jsfiddle.net/8xrk1n9f/)

* **event**

**2.1.0+**

* + 类型: string | Array<string>
  + 默认值: 'click'

声明可以用来触发导航的事件。可以是一个字符串或是一个包含字符串的数组。

* **exact-active-class**

**2.5.0+**

* + 类型: string
  + 默认值: "router-link-exact-active"

配置当链接被精确匹配的时候应该激活的 class。注意默认值也是可以通过路由构造函数选项 linkExactActiveClass进行全局配置的。

### 将激活 class 应用在外层元素

有时候我们要让激活 class 应用在外层元素，而不是 <a> 标签本身，那么可以用 <router-link> 渲染外层元素，包裹着内层的原生 <a> 标签：

<router-link tag="li" to="/foo">

<a>/foo</a>

</router-link>

在这种情况下，<a> 将作为真实的链接（它会获得正确的 href 的），而 "激活时的CSS类名" 则设置到外层的 <li>。

# <router-view>

<router-view> 组件是一个 functional 组件，渲染路径匹配到的视图组件。<router-view> 渲染的组件还可以内嵌自己的 <router-view>，根据嵌套路径，渲染嵌套组件。

### 属性

* **name**
  + 类型: string
  + 默认值: "default"

如果 <router-view>设置了名称，则会渲染对应的路由配置中 components 下的相应组件。查看 [**命名视图**](https://router.vuejs.org/zh-cn/essentials/named-views.html) 中的例子。

### 行为表现

其他属性（非 router-view 使用的属性）都直接传给渲染的组件， 很多时候，每个路由的数据都是包含在路由参数中。

因为它也是个组件，所以可以配合 <transition> 和 <keep-alive> 使用。如果两个结合一起用，要确保在内层使用 <keep-alive>：

<transition>

<keep-alive>

<router-view></router-view>

</keep-alive>

</transition>