

Vue-router

介绍说明

先安装

基础路由跳转

1. 声明式导航

2. 编程式导航

3. 路径参数（动态路由）

 路径参数 + 正则

 多个重复参数

 不区分大小写

 可选参数

4. 嵌套路由

5. 路由命名name

6. 路由容器router-view

7. 重定向

8. 别名alias

9. 路由组件传参

10. 不同的历史记录模式

进阶版路由配置

1. 导航守卫

 1. 全局前置守卫beforeEach

 2. 后置钩子afterEach

 3. 组件内的守卫

 4. 完整的导航解析流程

2. 路由元信息meta

3. 数据获取

 导航完成后（常用）

 导航完成前

4. 组合式API

路由对象

导航守卫

useLink

5.RouterView插槽

keepAlive

Transition

传递props和插槽

6.过渡动画

强制在复用的视图之间进行过渡

7.滚动效果

8.路由懒加载

9.类型化路由

10.拓展Router-Link

11.导航故障

12.动态路由

介绍说明

vue-router 是 官方出品的 路由管理器， 提供了一系列路由方法：

- 嵌套路由映射
- 动态路由选择
- 模块化、基于组件的路由配置
- 路由参数、查询、通配符
- 展示由 Vue.js 的过渡系统提供的过渡效果
- 细致的导航控制
- 自动激活 CSS 类的链接
- HTML5 history 模式或 hash 模式
- 可定制的滚动行为
- URL 的正确编码

项目中这些api已经能满足全部需求，这次针对各个方法做一个详细的介绍，使用示例。

先安装

```
PS D:\Aurora\study\Vue\vue-router-demo> pnpm add vue-router
● Packages: +2
  ++
  Progress: resolved 162, reused 107, downloaded 0, added 2, done

dependencies:
+ vue-router 4.5.0

Done in 1.4s
```

接着在src目录下新建router文件夹，router下新建index.js，这是路由的配置文件。

默认导出router对象，并在main.js中引入使用：

同时请注意，此处的router对象将会是我们整个项目中需要用到的router对象。

```
1 import router from './router/index'
2
3 const app = createApp(App)
4
5 app.use(router)
6
7 app.mount('#app')
```

请注意，vue-router提供了一个获取到当前路由对象的方法：useRouter，获取到的实例即为router（路由对象）

基础路由跳转

1. 声明式导航

首先准备两个组件，Home和A，将他们导入到路由配置中：

The screenshot shows a code editor with several tabs at the top: vite.config.js, App.vue, index.js, main.js, and index.scss. The current file is index.js. The code defines a routes array:

```
1 import { createMemoryHistory, createRouter } from 'vue-router'
2
3 import HomePage from '../components/home/index.vue'
4 import APage from '../components/A/index.vue'
5
6 const routes = [
7   { path: '/home', component: HomePage },
8   { path: '/a', component: APage },
9 ]
```

Lines 3 and 4 are highlighted with a red box, and lines 6 through 9 are highlighted with another red box.

此处的path参数和component参数一一对应，path不要重复

然后我们在页面上就可以写上跳转的组件RouterLink和对应路由组件渲染区域RouterView

这两个组件是vue-router包内提供的快捷组件，可以直接拿来使用。

The screenshot shows a code editor with a JavaScript file open. The code defines a navigation component:

```
1 <nav>
2   <RouterLink to="/home">Go to Home</RouterLink>
3   <RouterLink to="/a">Go to A</RouterLink>
4 </nav>
5 <main>
6   <RouterView />
7 </main>
```

我们点击对应的标签就可以切换到对应的路由，同时看到对应的内容：

Hello App!

[Go to Home](#) [Go to A](#)

我是A页面



这里只是提供了一个简单的路由示例，真实的项目中肯定不存在这么简单的跳转。

以上方法被称为：声明式导航，即通过一些特定的标签、标签属性来达成路由跳转的目的

2. 编程式导航

上面声明式导航其实只在菜单中比较常见，一般应用页面中应用更多的是点击某个按钮跳转到详情页等类似的操作，这种实现就需要借助编程式导航：

通俗地讲，就是代码里面手动实现跳转。`router.push`

```
1 <script setup>
2
3 import { useRouter } from "vue-router";
4
5 const router = useRouter()
6
7 const gotoHome = () => {
8     router.push("/home")
9 }
10
11 const gotoA = () => {
12     router.push("/a")
13 }
14
15 </script>
```

点击按钮也可以实现跳转：



为什么用的是`router.push()`方法？

因为路由相当于一个数组，你往里面push了一个新的路由地址后，当你返回`goBack`，就会回到数组前一项也就是旧的页面。

编程式导航有很多种写法，上面写的是最常见的，参数就是url: `router.push("/home")`

```
1 // 字符串路径
2 router.push('/users/eduardo')
3
4 // 带有路径的对象
5 router.push({ path: '/users/eduardo' })
6
7 // 命名的路由，并加上参数，让路由建立 url
8 router.push({ name: 'user', params: { username: 'eduardo' } })
9
10 // 带查询参数，结果是 /register?plan=private
11 router.push({ path: '/register', query: { plan: 'private' } })
12
13 // 带 hash，结果是 /about#team
14 router.push({ path: '/about', hash: '#team' })
```

注意，path和params不可以一起使用，name和params可以一起使用。

```
1 const username = 'eduardo'
2 // 我们可以手动建立 url，但我们必须自己处理编码
3 router.push(`/user/${username}`) // -> /user/eduardo
4 // 同样
5 router.push({ path: `/user/${username}` }) // -> /user/eduardo
6 // 如果可能的话，使用 `name` 和 `params` 从自动 URL 编码中获益
7 router.push({ name: 'user', params: { username } }) // -> /user/eduardo
8 // `params` 不能与 `path` 一起使用
9 router.push({ path: '/user', params: { username } }) // -> /user
```

我们打印router会发现里面存在很多方法：

```

▼ {currentRoute: RefImpl, listening: true, addRoute: f, removeRoute: f, cle
  arRoutes: f, ...} i
  ▶ addRoute: f addRoute(parentOrRoute, route)
  ▶ afterEach: f add(handler)
  ▶ back: () => go(-1)
  ▶ beforeEach: f add(handler)
  ▶ beforeResolve: f add(handler)
  ▶ clearRoutes: f clearRoutes()
  ▶ currentRoute: RefImpl {dep: Dep, __v_isRef: true, __v_isShallow: true, _}
  ▶ forward: () => go(1)
  ▶ getRoutes: f getRoutes()
  ▶ go: (delta) => routerHistory.go(delta)
  ▶ hasRoute: f hasRoute(name)
  ▶ install: install(app) { const router2 = this; app.component("RouterLink"
  ▶ isReady: f isReady()
  listening: true
  ▶ onError: f add(handler)
  ▶ options: {history: {...}, routes: Array(2)}
  ▶ push: f push(to)
  ▶ removeRoute: f removeRoute(name)
  ▶ replace: f replace(to)
  ▶ resolve: f resolve(rawLocation, currentLocation)
  _hasDevtools: true
  ▶ [[Prototype]]: Object

```

replace方法和push方法用法基本一致，区别无非就是replace不会往路由数组内塞入新的路由地址，所以我们在使用浏览器返回上一级的时候，此功能无法实现。因为上一级被我们replace掉了，没了。

3.路径参数（动态路由）

比方说现在有一个User路由页面：

```

1  <script setup>
2
3  import { onMounted, ref } from 'vue';
4  import { useRouter, useRoute } from 'vue-router';
5
6  const UserList = [
7      { id: 1, name: '小明', age: 18, phone: 132456789 },
8      { id: 2, name: '小王', age: 33, phone: 9999999 },
9      { id: 3, name: '小红', age: 22, phone: 120 },
10     { id: 4, name: '小叶', age: 28, phone: 4800810086 },
11 ]
12
13 const router = useRouter()
14 const route = useRoute()
15 const user_info = ref(null)
16
17 onMounted(() => {
18     console.log(route)
19
20 })
21
22 </script>
23
24 <template>
25     <div>我是User页面</div>
26     <template v-if="user_info">
27         <div>User姓名: {{ user_info.name || '-' }}</div>
28         <div>User年龄: {{ user_info.age || '-' }}</div>
29         <div>User电话: {{ user_info.phone || '-' }}</div>
30     </template>
31 </template>
32
33 <style scoped lang="scss">
34
35 </style>
36

```

针对于每个不同的用户，需要展示不同的信息，那么此时就需要讲用户ID（userId）传入到组件中。我们可以在User页面打印route看一下，这里的route.params即为我们通过路径传递过来的参数。

```
▼ Proxy(Object) {...} ⓘ index.vue:18
  ► [[Handler]]: MutableReactiveHandler
  ▼ [[Target]]: Object
    fullPath: "/user/2"
    hash: ""
    ► matched: Array(1)
    ► meta: Object
      name: undefined
    ▼ params: Object
      id: "2"
      ► [[Prototype]]: Object
      path: "/user/2"
    ► query: Object
      redirectedFrom: undefined
    ► get fullPath: () => currentRoute.value[key]
    ► get hash: () => currentRoute.value[key]
    ► get matched: () => currentRoute.value[key]
    ► get meta: () => currentRoute.value[key]
    ► get name: () => currentRoute.value[key]
    ► get params: () => currentRoute.value[key]
    ► get path: () => currentRoute.value[key]
    ► get query: () => currentRoute.value[key]
    ► get redirectedFrom: () => currentRoute.value[key]
    ► [[Prototype]]: Object
    [[IsRevoked]]: false
```

>

跳转页面即为：

Hello App! /user/4

Go to Home Go to A Go to User 跳转到Home页面 跳转到A页面
我是User页面
User姓名：小叶
User年龄：28
User电话：4800810086

至此为止就实现了一个比较简单的路径参数demo

route的全部属性可以参考：

<https://router.vuejs.org/zh/api/interfaces/RouteLocationNormalizedLoaded.html>

路径参数 + 正则

当应用中同时存在user页面、商品页面...的时候，比如：

```
1  '/1' // 匹配到用户页面
2  '/4663' // 匹配到商品详情页面
3  ...
```

这样就比较难以区分，到底是哪个页面，因为参数都是id，一般这个时候我们会在路径前面加上标识，例如：

```
1  /user/1
2  /product/4663
```

但是如果们不想要这种方式来区分两个页面，还可以通过正则表达式来匹配路径参数的格式：

```
1 ▼ const routes = [
2    // /:orderId -> 仅匹配数字
3    { path: '/:orderId(\d+)', components: A },
4    // /:productName -> 匹配其他任何内容
5    { path: '/:productName', components: B },
6  ]
```

此时如果跳转路径为/6，那么就会匹配到组件A；如果路径是/sxs，那么就会匹配到组件B

不过在真实项目环境中，还是推荐/user/1、/product/4366这种形式。这种更加符合语义化。

多个重复参数

我们注册/films路由以及组件：

```
1  { path: '/films/:id+', component: Films }
```

这里的路由规则表示匹配：/films/id，其中id可以多个（至少一个）



不区分大小写

如果你希望路由不区分大小写，例如/user和/User 匹配的是同一个路由页面，可以设置配置项：

```
sensitive: true
```

如果你希望不管尾部带不带/都匹配，例如/use、/user/ 匹配的是同一个路由页面，可以设置配置项：

```
strict: true
```

当然这两个参数支持单个路由设置，也支持全部统一设置：

```
▼ JavaScript |  
1 const router = createRouter({  
2   history: createWebHistory(),  
3   routes: [  
4     { path: '/users/:id', sensitive: true }, // 单个设置  
5   ],  
6   strict: true, // 全部设置  
7 })
```

可选参数

可以用?来表示参数可选

```

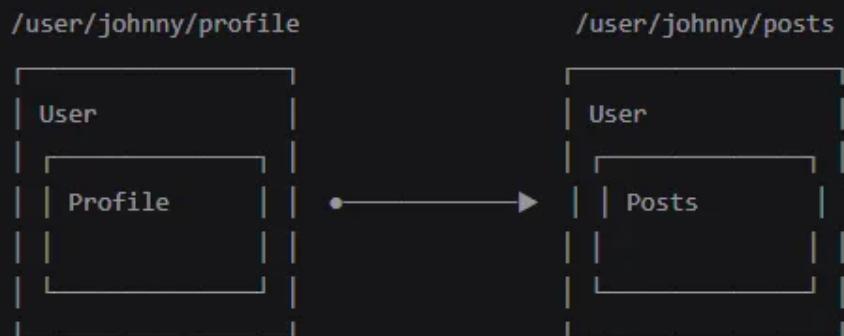
1 const routes = [
2   // 匹配 /users 和 /users/posva
3   { path: '/users/:userId?' },
4   // 匹配 /users 和 /users/42
5   { path: '/users/:userId(\d+)?' },
6 ]

```

4. 嵌套路由

当某一些页面存在公共部分，我们会考虑将其作为一个父路由下的两个子路由：

一些应用程序的 UI 由多层嵌套的组件组成。在这种情况下，URL 的片段通常对应于特定的嵌套路由结构，例如：



例如上图中，父路由就是User，存在两个子路由Profile、Posts，示例如下：

router.js

JavaScript

```

1 import Father from '../components/Father/index.vue'
2 import ChildA from '../components/child/A.vue'
3 import ChildB from '../components/child/B.vue'
4
5 {
6   path: '/father', component: Father, children: [
7     { path: 'childA', component: ChildA },
8     { path: 'childB', component: ChildB },
9   ]
10 }

```

注意，子路由不需要再加上/

Father文件如下：

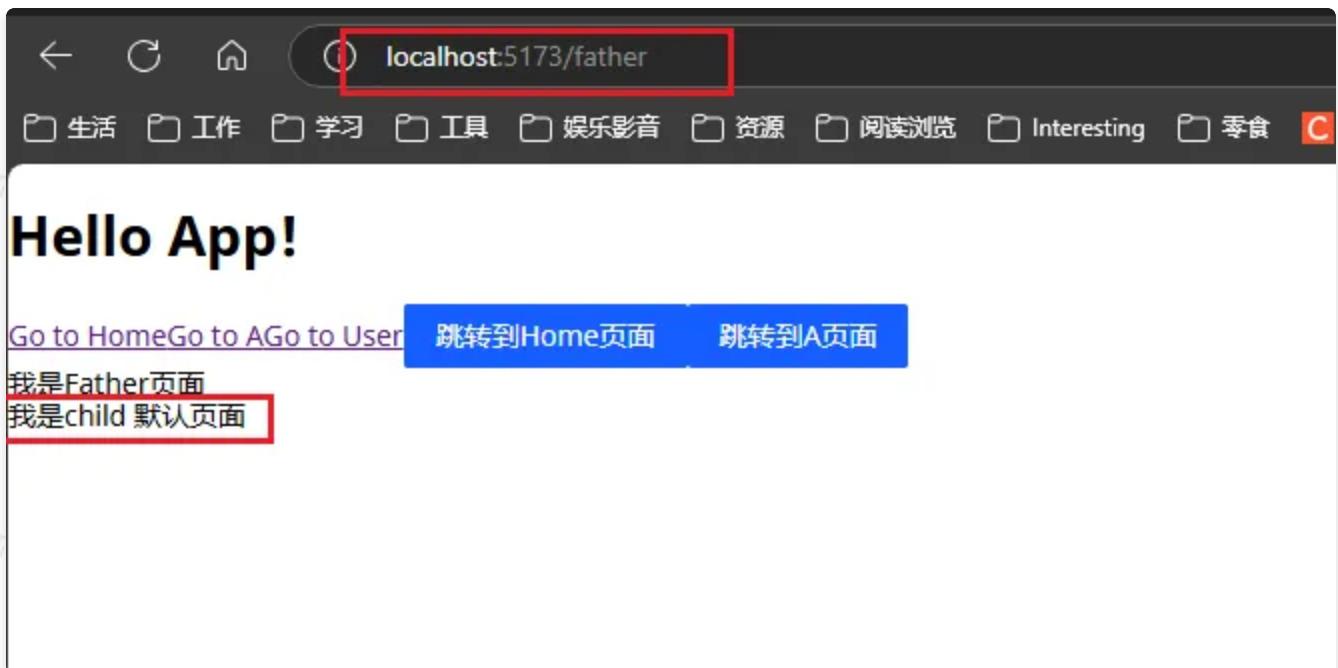
```
▼ Father
1 <template>
2   <div>我是Father页面</div>
3   <RouterView></RouterView> // 这里可以再写一个RouterView
4 </template>
```

最终实现的效果就是嵌套路由：



可以制定一个默认的展示路由，当后面不跟任何嵌套后缀的时候展示：

```
1 {
2   path: '/father', component: Father, children: [
3     { path: '', component: ChildDefault }, // 默认路由
4     { path: 'childA', component: ChildA },
5     { path: 'childB', component: ChildB },
6   ]
7 }
```



另外，子路由也全部支持name属性，此时父路由的name属性会失效。

```
Vue |  
1  {  
2    path: '/father', component: Father, name: '我是父页面', children: [  
3      { path: '', component: ChildDefault },  
4      { path: 'childA', component: ChildA, name: '我是childA' },  
5      { path: 'childB', component: ChildB },  
6    ]  
7  }
```

5. 路由命名name

在创建路由的时候，可以给每一个路由指定name，比如下面例子name为：profile

```
Vue |  
1  const routes = [  
2    {  
3      path: '/user/:username',  
4      name: 'profile',  
5      component: User  
6    }  
7  ]
```

命名后，我们就可以通过name的方式来进行跳转，而不是url：

```
1 <RouterLink :to="{ name: 'main' }">Go to Home</RouterLink>
```

请注意，name属性也要保持唯一性。有些人习惯在路由中写url跳转，建议可以统一使用name来跳转。

而且，当url过长：/xxx.../xxx.../xxx...的时候，使用name也会更加方便

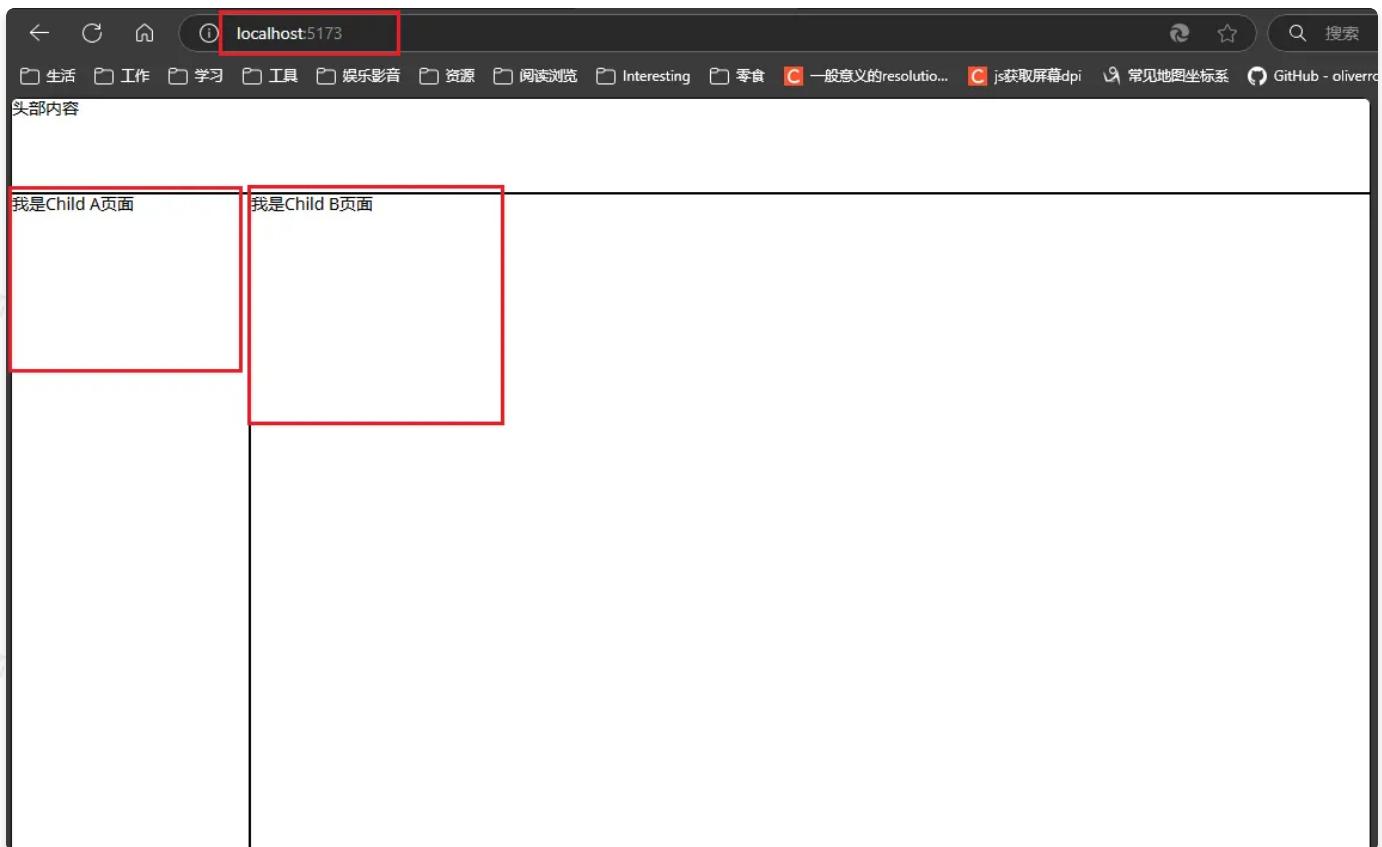
6. 路由容器router-view

当某个容器中需要多个路由视图，此时可以在页面中放置多个router-view容器，并且用

```
1 <template>
2   <a-layout class="main_container">
3     <a-layout-header class="header border">头部内容</a-layout-header>
4     <a-layout class="center">
5       <a-layout-sider class="sider border" :collapsible="true">
6         <router-view name="LeftSidebar" /> // 这里有一个 LeftSidebar 容器
7       </a-layout-sider>
8       <a-layout-content class="content border">
9         <router-view name="Content" /> // 这里有一个 Content 容器
10      </a-layout-content>
11    </a-layout>
12    <a-layout-footer class="footer border">Footer</a-layout-footer>
13  </a-layout>
14 </template>
```

然后当我们在申明router的时候，不能只申明component，而改成components（对象），里面根据router-view的name属性来指定路由组件即可。

```
1 {
2   path: '/',
3   components: {
4     default: Films,
5     LeftSidebar: ChildA,
6     Content: ChildB
7   }
8 }
```



简单来说就是如果在某一个路由下需要放置多个路由容器，那么就要区分各个router-view的name属性并在components中对应申明。

7.重定向

重定向的意思就是当我们打开/A路由的时候会自动跳转到/B，一般应用场景是：比如

- 某个页面没有权限查看，当我们进入后需要重定向到403页面

```
1  {
2    path: '/',
3    component: HomePage,
4    redirect: '/A'
5  },
6  { path: '/A', component: APage },
```

那么此时，redirect中还可以写上重定向的逻辑（便于用来判断是否具有权限）：

```

1 const routes = [
2   {
3     // /search/screens -> /search?q=screens
4     path: '/search/:searchText',
5     redirect: to => {
6       // 方法接收目标路由作为参数
7       // return 重定向的字符串路径/路径对象
8       return { path: '/search', query: { q: to.params.searchText } }
9     },
10   },
11   {
12     path: '/search',
13     // ...
14   },
15 ]

```

相对重定向的时候（存在父子路由），尽量选择name属性来进行跳转。

```

1  {
2    path: '/father', component: Father, children: [
3      {
4        path: 'childA', component: ChildA, name: 'childA', redirect: t
5        o => {
6          console.log(to)
7          // 使用相对路径
8          return { name: 'childB' };
9        }
10      },
11      { path: 'childB', component: ChildB, name: 'childB' },
12    ]
13  }

```

8.别名alias

别名的意思就是另一个名字，比如path为 /child 的路由匹配到 child页面，如果我们给这个路由设置别名： /c

那么当我们访问/c的时候，尽管路由是 /c，我们在路由配置中没有申明/c对应的页面，但是还是会匹配到child页面，也就是/c和/child所指向的路由页面是相同的。

```
1  {
2    path: '/father', component: Father, children: [
3      {
4        path: 'childA', component: ChildA, name: 'childA', alias: ['childC', 'c']
5      },
6      { path: 'childB', component: ChildB, name: 'childB' },
7    ]
8 }
```



9. 路由组件传参

上面我们说到，路由组件可以通过`route.params.id`的方式来访问参数，对应的页面url为`/user/:id`还有一种快捷方式就是在路由配置中指定 `props: true` 即可

```
1 const routes = [
2   { path: '/user/:id', component: User, props: true }
3 ]
```

此时 `route.params` 将被设置为组件的 `props`

如果是上面的多个命名路由，那么就需要分别对应设置：

```

1 const routes = [
2   {
3     path: '/user/:id',
4     components: { default: User, sidebar: Sidebar },
5     props: { default: true, sidebar: false }
6   }
7 ]

```

同时在官方示例中，props除了可以设置为布尔值还可以设置为Object, Function等等...

<https://router.vuejs.org/zh/guide/essentials/passing-props.html#%E5%AF%B9%E8%B1%A1%E6%A8%A1%E5%BC%8F>

10.不同的历史记录模式

vue-router的历史记录模式主要可以分为以下三种：

- HTML5 模式—— `createWebHistory()` (这是我们最常用的一种)
- Hash模式—— `createWebHashHistory()`
- Memory 模式—— `createMemoryHistory()`

```

1 import { createRouter, createWebHistory } from 'vue-router'
2
3 const router = createRouter({
4   history: createWebHistory(), // 在这里设置历史记录模式
5   routes: [
6     //...
7   ],
8 })

```

进阶版路由配置

1.导航守卫

1.全局前置守卫beforeEach

导航守卫的使用场景一般就是：在进行某些页面的跳转的时候，如果用户无权限则取消跳转。

```
1 const router = createRouter({ ... })
2
3 ▼ router.beforeEach((to, from) => {
4     // ...
5     // 返回 false 以取消导航
6     return false
7 })
```

to: 即将要跳转的路由地址

form: 原来的路由地址

返回结果支持以下几种格式：

- 布尔值：如果是false，路由跳转会被取消
- 同router.push()的参数格式，如同上面的应用场景一样说的，既可以跳转的某些特定页面比如登录页面、403无权限页面等等

beforeEach 可以注册多个，当存在多个导航守卫的时候，会按照创建顺序分别执行每一个。

以上这些都是全局注册，也适用于单个路由注册

2.后置钩子afterEach

```
1 ▼ router.afterEach((to, from) => {
2     sendToAnalytics(to fullPath)
3 })
```

它常用于分析页面，更改页面标题等等

以上这些都是全局注册，也适用于单个路由注册

3.组件内的守卫

```
1  <script>
2  export default {
3    beforeRouteEnter(to, from) {
4      // 在渲染该组件的对应路由被验证前调用
5      // 不能获取组件实例 `this` !
6      // 因为当守卫执行时，组件实例还没被创建!
7    },
8    beforeRouteUpdate(to, from) {
9      // 在当前路由改变，但是该组件被复用时调用
10     // 举例来说，对于一个带有动态参数的路径 `/users/:id`，在 `/users/1` 和 `/users/2` 之间跳转的时候，
11     // 由于会渲染同样的 `UserDetails` 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。
12     // 因为在这种情况下发生的时候，组件已经挂载好了，导航守卫可以访问组件实例 `this`、
13   },
14   beforeRouteLeave(to, from) {
15     // 在导航离开渲染该组件的对应路由时调用
16     // 与 `beforeRouteUpdate` 一样，它可以访问组件实例 `this`、
17   },
18 }
19 </script>
```

在组合式api中，你只能使用 `beforeRouteUpdate` 和 `beforeRouteLeave` 这两项

4.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 `beforeRouteLeave` 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. 调用 `beforeRouteEnter` 守卫中传给 `next` 的回调函数，创建好的组件实例会作为回调函数的参数传入。

2. 路由元信息meta

所谓的元信息就是你可以加到每一条路由记录route上面的一些自定义属性（包括业务信息）等在访问route和导航守卫的时候都可以被获取到meta的信息：

```

1 const routes = [
2   {
3     path: '/posts',
4     component: PostsLayout,
5     children: [
6       {
7         path: 'new',
8         component: PostsNew,
9         // 只有经过身份验证的用户才能创建帖子
10        meta: { requiresAuth: true },
11      },
12      {
13        path: ':id',
14        component: PostsDetail
15        // 任何人都可以阅读文章
16        meta: { requiresAuth: false },
17      }
18    ]
19  }
20]

```

```

1 router.beforeEach((to, from) => {
2   // 而不是去检查每条路由记录
3   // to.matched.some(record => record.meta.requiresAuth)
4   if (to.meta.requiresAuth && !auth.isLoggedIn()) {
5     // 此路由需要授权, 请检查是否已登录
6     // 如果没有, 则重定向到登录页面
7     return {
8       path: '/login',
9       // 保存我们所在的位置, 以便以后再来
10      query: { redirect: to fullPath },
11    }
12  }
13})

```

TS的支持:

```
1 // 这段可以直接添加到你的任何 `ts` 文件中，例如 `router.ts`  
2 // 也可以添加到一个 `d.ts` 文件中。确保这个文件包含在  
3 // 项目的 `tsconfig.json` 中的 "file" 字段内。  
4 import 'vue-router'  
5  
6 // 为了确保这个文件被当作一个模块，添加至少一个 `export` 声明  
7 export {}  
8  
9 - declare module 'vue-router' {  
10 -   interface RouteMeta {  
11       // 是可选的  
12       isAdmin?: boolean  
13       // 每个路由都必须声明  
14       requiresAuth: boolean  
15   }  
16 }
```

3. 数据获取

一般我们在业务系统中，例如某一个详情页面，获取数据有两种方式：

- 在导航完成后，在组件的生命周期钩子（created、onMounted）中获取数据——导航完成后
- 在导航完成之前，前置守卫（beforeEach）中获取数据——导航完成前

导航完成后（常用）

这种方式一般比较常用，并且可以自定义添加loading效果等等，下面是一段示例代码：

```
1 <template>
2   <div class="post">
3     <div v-if="loading" class="loading">Loading...</div>
4
5     <div v-if="error" class="error">{{ error }}</div>
6
7     <div v-if="post" class="content">
8       <h2>{{ post.title }}</h2>
9       <p>{{ post.body }}</p>
10      </div>
11    </div>
12  </template>
13
14 <script setup>
15 import { ref, watch } from 'vue'
16 import { useRoute } from 'vue-router'
17 import { getPost } from './api.js'
18
19 const route = useRoute()
20
21 const loading = ref(false)
22 const post = ref(null)
23 const error = ref(null)
24
25 // 监听路由的参数，以便再次获取数据
26 watch(() => route.params.id, fetchData, { immediate: true })
27
28 - async function fetchData(id) {
29   error.value = post.value = null
30   loading.value = true
31
32 -   try {
33     // 用获取数据的工具函数 / API 包裹器替换 `getPost`
34     post.value = await getPost(id)
35 -   } catch (err) {
36     error.value = err.toString()
37 -   } finally {
38     loading.value = false
39   }
40 }
41 </script>
```

导航完成前

```

1  export default {
2    data() {
3      return {
4        post: null,
5        error: null,
6      }
7    },
8    beforeRouteEnter(to, from, next) {
9      try {
10        const post = await getPost(to.params.id)
11        // `setPost` 方法定义在下面的代码中
12        next(vm => vm.setPost(post))
13      } catch (err) {
14        // `setError` 方法定义在下面的代码中
15        next(vm => vm.setError(err))
16      }
17    },
18    // 路由改变前，组件就已经渲染完了
19    // 逻辑稍稍不同
20    async beforeRouteUpdate(to, from) {
21      this.post = null
22      getPost(to.params.id).then(this.setPost).catch(this.setError)
23    },
24    methods: {
25      setPost(post) {
26        this.post = post
27      },
28      setError(err) {
29        this.error = err.toString()
30      }
31    }
32  }

```

这种方法下，获取数据出现错误等情况会停留在当前页面，并且最好做一个全局异常报错的提示等等。

4. 组合式API

路由对象

在选项式api中，访问路由管理器router可以通过 `this.$router`，访问当前路由实例可以通过 `this.$route`

但是在组合式api中，需要借助useRouter、useRoute两种方式来对应访问

```
1 <script setup>
2 import { onMounted } from 'vue';
3 import { useRouter, useRoute } from 'vue-router';
4
5 const route = useRoute()
6
7 onMounted(() => {
8     console.log(route)
9 })
10 </script>
```

注意，在模板中还是可以直接借助\$route访问：

```
1 <template>
2     <div>我是Child A页面</div>
3     <div>{{ $route fullPath }}</div>
4 </template>
```

并且，`route` 是一个响应式对象，应避免直接监听整个route对象，而是改为监听某一属性值：

```
1 watch(
2     () => route.params.id,
3     async newId => {
4         userData.value = await fetchUser(newId)
5     }
6 )
```

导航守卫

```

1  <script setup>
2  import { onBeforeRouteLeave, onBeforeRouteUpdate } from 'vue-router'
3  import { ref } from 'vue'
4
5  // 与 beforeRouteLeave 相同, 无法访问 `this`
6  onBeforeRouteLeave((to, from) => {
7      const answer = window.confirm(
8          'Do you really want to leave? you have unsaved changes!'
9      )
10     // 取消导航并停留在同一页面上
11     if (!answer) return false
12 })
13
14 const userData = ref()
15
16 // 与 beforeRouteUpdate 相同, 无法访问 `this`
17 onBeforeRouteUpdate(async (to, from) => {
18     //仅当 id 更改时才获取用户, 例如仅 query 或 hash 值已更改
19     if (to.params.id !== from.params.id) {
20         userData.value = await fetchUser(to.params.id)
21     }
22 })
23 </script>

```

useLink

通过useLink可以自定义router-link，例如当地址为外部http链接的时候我们可以选择打开新的标签页跳转等等。

使用场景不多，具体可以参考官方示例：

<https://router.vuejs.org/zh/guide/advanced/composition-api.html#useLink>

5.RouterView插槽

vue-router使用了router-view作为路由组件的插槽（router-view同RouterView）

常规写法为：

```
1 <router-view />
```

也可以写成插槽的形式：

```
1 <router-view v-slot="{ Component }">
2   <component :is="Component" />
3 </router-view>
```

基于插槽的写法，可以带来很多的扩展性，比如：

keepAlive

```
1 <router-view v-slot="{ Component }">
2   <keep-alive>
3     <component :is="Component" />
4   </keep-alive>
5 </router-view>
```

这样可以使得组件具有keep-alive属性的特性，关于keep-alive可参考：

<https://cn.vuejs.org/guide/built-ins/keep-alive>

Transition

同样的我们也可以给路由组件增加Transition组件效果：

```
1 <router-view v-slot="{ Component }">
2   <transition>
3     <component :is="Component" />
4   </transition>
5 </router-view>
```

也可以结合上述两种第三方组件：

```

1 <router-view v-slot="{ Component }">
2   <transition>
3     <keep-alive>
4       <component :is="Component" />
5     </keep-alive>
6   </transition>
7 </router-view>

```

传递props和插槽

```

1 <router-view v-slot="{ Component }">
2   <component :is="Component" some-prop="a value">
3     <p>Some slotted content</p>
4   </component>
5 </router-view>

```

不过这种应用场景不太频繁，路由之间的数据基本通过全局来获取不需要通过props

6.过渡动画

如果想要在路由组件上进行过渡动画的添加，那么可以按照上面说的增加 `<transition><transition />` 组件

如果想要实现单个路由单独过渡动画，可以将过渡动画的信息放置在元数据meta中，如下：

```

1 const routes = [
2   {
3     path: '/custom-transition',
4     component: PanelLeft,
5     meta: { transition: 'slide-left' },
6   },
7   {
8     path: '/other-transition',
9     component: PanelRight,
10    meta: { transition: 'slide-right' },
11  },
12 ]
13
14 <router-view v-slot="{ Component, route }">
15   <!-- 使用任何自定义过渡和回退到 `fade` -->
16   <transition :name="route.meta.transition || 'fade'">
17     <component :is="Component" />
18   </transition>
19 </router-view>

```

强制在复用的视图之间进行过渡

Vue 可能会自动复用看起来相似的组件，从而忽略了任何过渡。幸运的是，可以添加一个 `key` 属性来强制过渡。这也允许你在相同路上使用不同的参数触发过渡：

```

<router-view v-slot="{ Component, route }">
  <transition name="fade">
    <component :is="Component" :key="route.path" />
  </transition>
</router-view>

```

7.滚动效果

在某几个页面中，可能都会存在滚动列表，当你在A页面滚动到底后跳转到B页面，如何实现跳转后重新滚动到顶部或者是滚动到指定位置呢？

注意: 这个功能只在支持 `history.pushState` 的浏览器中可用。

解决方案是在路由配置的地方进行设置:

```
1 const router = createRouter({
2   history: createWebHashHistory(),
3   routes: [...],
4   scrollBehavior (to, from, savedPosition) {
5     // return 期望滚动到哪个的位置
6   }
7 })
```

JavaScript |

具体看文档:

<https://router.vuejs.org/zh/guide/advanced/scroll-behavior.html>

8. 路由懒加载

9. 类型化路由

10. 拓展Router-Link

11. 导航故障

12. 动态路由

至此，vue-router全部结束