第12章 Vue Router 路由

在前端开发中，经常会听说单页面应用程序（Single-Page Application，简称SPA）。这种应用程序只有一个页面，页面的内容是通过JS动态渲染。在切换页面时，SPA应用并不会重新加载新页面，而是通过hash或history API来实现前端路由的切换。

在 React 应用程序中，前端路由可以通过react-router-dom实现，而在Vue.js 3应用程序中，前端路由可通过官方提供的Vue Router来实现。Vue Router不仅可以实现基本的路由功能，还提供了丰富的路由导航守卫、动态路由、嵌套路由等高级功能，使得开发者可以更加灵活地控制前端路由。在学习Vue Router前，先看看本章节源代码的管理方式，目录结构如下：

```
VueCode
├── ......
├── chapter11
|   └── 01_learn_vuerouter
|   └── 02_learn_vuerouter_project # 提供一个空的Vue.js 3项目
```

12.1 邂逅Vue Router

12.1.1 认识前端路由

路由实际上是网络工程中的一个术语，在构建网络时，路由器和交换机是两个非常重要的设备。路由器在我们的日常生活中随处可见。它主要维护一个**映射表**，该映射表决定了数据的流向。在软件开发中，路由概念最初出现在后端路由。随着Web技术的高速发展，路由主要经历了以下三个阶段。

1.后端路由

早期的网站，HTML页面是由服务器端渲染，服务器端将渲染好的HTML页面直接返回给客户端展示。当一个网站存在多个页面时，每个页面都有对应的网址（URL）。当访问网址时，URL会发送到服务器，服务器会通过正则对该URL进行匹配，并且交给Controller进行处理，最终生成HTML或数据返回给前端。这种每个页面都对应一个URL路径，称为后端路由。。通过后端路由实现的应用程序有如下优点。

- 更快的首屏渲染速度：HTML页面是由服务器端渲染，服务器端将渲染好的页面直接返回给客户端展示，客户端不需要单独加载任何的JS和CSS资源，也不需要动态生成页面。
- 更好的SEO：服务器端直接返回静态的HTML，爬虫是最擅长爬取静态的HTML页面，有利于SEO。

然而，后端路由实现的应用程序也有如下的缺点:

- 整个页面的模块需前后端人员共同参与编写和维护。
- 前端开发人员需要通过PHP、Java或Node等语言来编写页面代码。
- HTML代码和数据以及对应的逻辑会混在一起，编写和维护都是非常糟糕。

2.前后端分离

随着Ajax的出现，前后端分离的开发模式逐渐流行。该模式下，前端负责渲染页面，后端负责提供数据。当访问一个网页时，页面的静态资源都会从静态资源服务器获取，包括HTML、CSS、JS等。前端会对这些请求回来的资源进行动态渲染，无需后端提供渲染好的HTML页面，只需提供接口（API）即可。通过前后端分离实现的应用程序具有以下优点。

- 后端只需提供API返回数据，前端通过Ajax获取数据，然后通过JavaScript将页面动态渲染。
- 前后端职责清晰，后端专注于数据，前端专注于交互和可视化。
- 后端编写的一套API多端适用，例如Web端、微信小程序、移动端（iOS/Android）等。

3.单页面应用程序（SPA）

单页面应用程序只有一个页面，当URL发生改变时，并不会从服务器请求新的静态资源。而是通过JavaScript监听URL的改变，并根据URL的不同去渲染新的页面，这也是前端路由实现的原理。

前端路由维护着URL和渲染页面的映射关系，它可根据不同的URL，让框架（如Vue.js 3、React、Angular）去渲染不同的组件。最终，我们在页面上看到的是渲染的一个个页面组件。单页面应用程序的优点如下。

- 只需加载一次：SPA只需要在第一次请求时加载页面，因此页面加载速度快。
- 更好的用户体验：类似于桌面或移动应用程序的体验，页面切换无需重新加载，因此体验更流畅。

12.1.2 前端路由原理

前端路由维护着URL和渲染页面的映射关系，它能够根据不同的URL，让框架去渲染不同的组件。目前，前端路由通常采用两种实现方案：hash模式和history模式。

下面将详细讲解这两种方式的实现。

1.hash模式

在Vue.js 3中，hash模式类似于锚点，本质上是通过在URL中使用#号来拼接路径（也称路由路径）来实现的。因此，我们可以通过修改location.hash的值来修改URL中#号拼接的路径，这种修改hash的方式不会刷新浏览器。接着，我们可以通过监听URL中hash的变化来切换渲染的内容，例如渲染页面组件等。

下面我们通过hash模式来实现前端路由。在`01_learn_vuerouter`项目根目录下新建`hash-demo.html`文件。

在hash-demo.html文件中，使用hash来实现前端路由，代码如下所示：

```
<!DOCTYPE html>
<html lang="en">
......
<body>
  <div id="app">
    <!-- 2.单击a元素修改URL的hash值 -->
    <a href="#/home">home</a>
    <a href="#/about">about</a>
    <div class="content">Default</div>
  </div>
  <script>
    const contentEl = document.querySelector('.content');
```

```
    changePage() // 1.根据当前hash，修改页面显示的内容
    window.addEventListener("hashchange", () => {// 3.监听hash变化，单击a标签会改
变hash
      changePage()
    })
    function changePage(){// 4.根据URL的hash来修改页面显示的内容
      switch(location.hash) {
        case "#/home":
          contentEl.innerHTML = "Home";
          break;
        case "#/about":
          contentEl.innerHTML = "About";
          break;
        default:
          contentEl.innerHTML = "Default";
      }
    }
  </script>
</body>
</html>
```

可以看到，首先分别给两个`<a>`元素的`href`属性添加`#/home`和`#/about`锚点（URL的`hash`也是锚点）。

接着，在`<script>`标签中使用`addEventListener`来监听`hash`改变事件。当单击“”元素时，会改变URL的`hash`值，`hash`改变会触发`changePage`函数回调。`changePage`函数的作用是根据URL的`hash`值来修改`contentEl`元素的内容，这样就可以实现切换`hash`时，切换页面显示的内容。

保存代码，在`hash-demo.html`文件上`右击 ->Open In Default Browser`，在浏览器中显示的效果如图12-1所示。当单击"home"来修改hash时，页面上会显示"Home"；当单击"about"时，页面上会显示"About"。

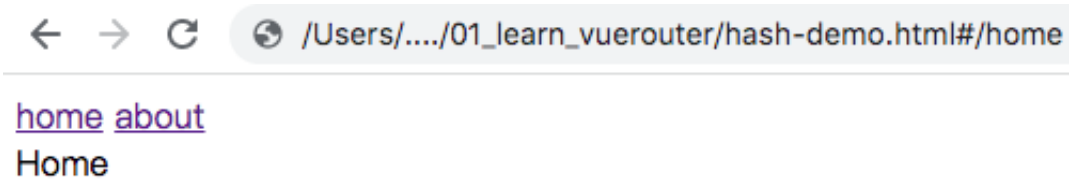

图12-1 前端路由之hash模式

2.history模式

在Vue.js 3中，可以使用history模式来实现前端路由。HTML5新增的history接口提供了以下六种模式来改变URL而不刷新页面。

1. replaceState：替换原来的路径，改变URL但不支持回退。
2. pushState：使用新的路径，改变URL并支持回退。
3. popState：用于路径的回退。
4. go：用于向前或向后改变路径。
5. forward：用于向前改变路径。

6. back：用于向后改变路径。

下面通过history模式来实现前端路由。在`01_learn_vuerouter`项目根目录下新建`hash-demo.html`文件。

在history-demo.html文件中使用history模式来实现前端路由，代码如下所示：

```
<!DOCTYPE html>
<html lang="en">
......
<body>
  <div id="app">
    <a href="/home">home</a>
    <a href="/about">about</a>
    <div class="content">Default</div>
  </div>
  <script>
    const contentEl = document.querySelector('.content');
    const aEls = document.getElementsByTagName("a");
    for (let aEl of aEls) {
      aEl.addEventListener("click", e => { // 1.监听所有a元素的单击事件
        e.preventDefault(); // 2.阻止a元素的默认行为，例如界面跳转
        const href = aEl.getAttribute("href");
        history.pushState({}, "", href); // 3.改变URL路径而不刷新页面，页面会压栈，
支持回退
        // history.replaceState({}, "", href);
        changeContent(); // 4.根据URL来切换显示的内容
      })
    }
    const changeContent = () => {
      switch(location.pathname) { // 根据URL的路径来切换显示的内容
        case "/home":
          contentEl.innerHTML = "Home";
          break;
        case "/about":
          contentEl.innerHTML = "About";
          break;
        default:
          contentEl.innerHTML = "Default";
      }
    }
    // 4.监听页面的回退，即页面出栈操作
    window.addEventListener("popstate", changeContent)
  </script>
</body>
</html>
```

可以看到，先给两个`<a>`元素的`href`属性分别添加了`/home`和`/about`两个URL路径（注意：不是锚点了）。

接着，在`<script>`标签中使用`for`循环分别给`<a>`元素添加单击事件。当单击""元素时，先拿到当前`<a>`元素`href`属性的值，并将该值传递给`history.pushState`的第三个参数来修改URL路径（其中，第一个参数是一个状态对象，第二个参数是一个标题，这里了解即可）。

然后，调用`changeContent`函数，该函数会根据当前URL路径来切换要显示的内容。最后，监听了`popState`事件，即单击页面的“返回”按钮时会回调`changeContent`函数。

保存代码，在`history-demo.html`文件上`右击 -> Open With Live Server`（注意：history模式不支持本地网页，所以这里使用Live Server插件运行）。在浏览器中显示的效果如图12-2所示。当单击“home”修改URL时，页面上会显示“Home”；当单击“about”时，页面上会显示“About”。

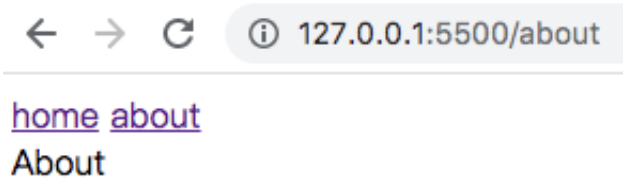

图12-2 前端路由之hash模式

12.1.3 认识Vue Router

上述两个案例仅仅展示了前端路由的核心原理，而在实际开发中，前端路由要复杂得多。为了方便开发和使用前端路由，目前流行的三大框架都有自己的路由实现，它们分别是：

- Angular 的 ngRouter。
- React 的 React Router。
- Vue.js 3 的 Vue Router。

Vue Router 是 Vue.js 官方提供的路由插件，支持 hash 和 history 两种模式。它与 Vue.js 深度集成，让使用 Vue.js 构建单页应用变得非常容易。

目前，Vue Router 的最新版本是 4.x，本书将基于最新版本进行讲解。我们可以在项目的根目录中执行以下命令进行安装。

```
npm install vue-router@4
```

12.2 Vue Router的基本使用

Vue Router是基于路由和组件的。路由用于设定访问路径，需要编写一个路由配置信息将路径和组件映射起来。在单页面应用中，页面路径的改变实际上就是组件的切换。使用Vue Router可以分为以下6个步骤。

1. 在项目根目录执行`npm install vue-router@4`安装 **Vue Router** 插件。
2. 创建路由组件，也可以理解为创建**页面组件**。
3. 配置路由映射，在`routes`数组中配置路由组件和路径之间的映射关系。
4. 通过`createRouter`创建路由对象，并传入`routes`和`history`模式（或`hash`模式）。
5. 使用`app.use`函数将路由插件安装到Vue.js 3框架中。
6. 通过Vue Router内置的`<router-link>`和`<router-view>`组件来使用路由。

12.2.1 路由的基本使用

1.安装Vue Router

打开VS Code终端，在`01_learn_vuerouter`项目的根目录下执行：`npm install vue-router@4.0.14`来安装 Vue Router。安装完成后，在 package.json 文件的`dependencies`属性中会增加`vue-router`的信息。

```
  "dependencies": {
    "core-js": "^3.6.5",
    "vue": "^3.0.0",
    "vue-router": "^4.0.14"
  },
```

2.创建路由组件

在`01_learn_vuerouter`项目的`src`目录下新建`pages`文件夹，然后在该文件夹下分别新建`Home.vue`和`About.vue`路由组件（也称页面组件 或 页面），代码如下所示。

Home.vue页面

```
<template>
  <div class="home">Home Page</div>
</template>
```

About.vue页面

```
<template>
  <div class="about">About Page</div>
</template>
```

3~4.配置路由映射和创建路由对象

在`01_learn_vuerouter`项目的`src`目录下新建router文件夹，然后在该文件夹下新建`index.js`文件，代码如下所示。

```
import { createRouter, createWebHistory, createWebHashHistory } from 'vue-router'
import Home from "../pages/Home.vue";  // 1.导入Home页面，也称路由组件或页面组件
import About from "../pages/About.vue";
const routes = [ // 2.配置路由映射表（路径-->组件）
  {
    path: '/home',
    component: Home
  },
  {
    path: '/about',
    component: About
```

```
  }
]
const router = createRouter({ // 3.导出创建好的路由对象
  routes,
  history: createWebHashHistory() // 4.指定用hash路由
  // history: createWebHistory() //  5.指定用history路由
})
export default router
```

从上面代码可以看到，我们完成了两个核心任务。

1. 在routes数组中配置了路由组件和路径之间的映射关系。例如，Home页面对应的路径是"/home"。
2. 用vue-router提供的createRouter函数来创建路由对象，并将包含routes数组和history属性的对象传递给该函数。其中，history属性用于指定路由的模式。如果值为createWebHashHistory()，则代表用hash模式；如果值为createWebHistory()，则代表用history模式。最后，我们默认导出router路由对象。

5.将路由插件安装到Vue.js 3框架中

修改`01_learn_vuerouter`项目的`src/main.js`入口文件，代码如下所示：

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router/index'
const app=createApp(App)
app.use(router) // 1.安装路由插件
app.mount('#app')
```

可以看到，上面代码比较简单，我们先导入了`router`对象，然后调用了`app.use`函数来安装路由插件。

6.通过`<router-link>`和`<router-view>`来使用路由。

修改`01_learn_vuerouter`项目的`src/App.vue`组件，代码如下所示：

```
<template>
  <div class="nav">
    <!-- 1.切换路由，即切换页面 -->
    <router-link class="tab" to="/home">首页</router-link>
    <router-link class="tab" to="/about">关于</router-link>
  </div>
  <!-- 2.路由组件的占位 -->
  <router-view></router-view>
</template>
......
<style>
.nav{
  margin: 20px 0px;
}
```

```
.tab{
  border: 1px solid #ddd;
  margin-right: 8px;
  padding: 2px 20px;
  text-decoration: none;
}
</style>
```

从上面代码可以看到，我们实现了以下两个功能。

1. 在模板中使用了Vue Router内置的`<router-link>`组件，该组件可以创建`<a>`标签来定义**导航链接**。当用户单击该组件时，便可实现路由切换（即URL路径切换）。该组件的`to`属性用来指定单击后切换到的路径。例如，当单击"首页"时，URL路径会变成"/home"；当单击"关于"时，URL路径会变成"/about"。
2. `<router-view>`组件也是Vue Router的内置组件。这里我们使用该组件的作用是为路由组件提供一个占位。当用户单击“首页”时，`<router-view>`组件处渲染的就是Home.vue页面；当单击"关于"时，`<router-view>`组件处渲染的就是About.vue页面。

保存代码，然后在VS Code终端中执行`npm run serve`，在浏览器中显示的效果如图12-3所示。当单击“首页”时，`<router-view>`组件处渲染的就是Home.vue页面；当单击"关于"时，`<router-view>`组件处渲染的就是About.vue页面。

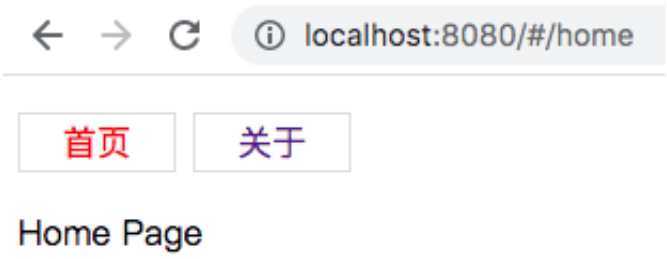

图12-3 Vue Router基本使用

需要注意的是，如果我们在浏览器中直接输入"http://localhost:8080/"并按Enter键，会发现页面中的`<router-view>`组件处并没有显示任何内容。此时，如果我们按F12键打开调试工具查看控制台，会看到如图12-4所示的警告，提示没有匹配到路由的默认路径。为了解决这个问题，下面给大家介绍路由配置的细节。

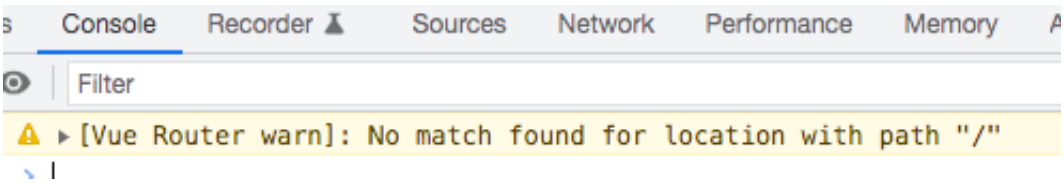

图12-4 提示没匹配到默认路径

12.2.2 路由配置的细节

1.路由的默认路径（`/`）

默认情况下，当进入网站首页时，我们希望`<router-view>`组件处默认渲染首页（Home.vue）的内容。然而，在上述案例中，默认情况下并没有显示首页组件，而是需要单击"首页"按钮才会显示。

为了让访问根路径时默认跳转到首页，我们需要在路由配置中设置默认路径，代码如下所示：

```
......
const routes = [
  {path: '/',redirect: '/home'}, // 1.路由默认路径('/')，重定向到/home路径
  {path: '/home',component: Home},
  {path: '/about',component: About}
]
......
```

从上面的代码可以看出，我们在`routes`数组的第一项中又配置了一个映射。

- `path`：配置`/`，表示根路径。
- `redirect`：表示重定向。这里配置`/home`。意思是将根路径`/`重定向到`/home`路径。

这样，在浏览器中输入`http://localhost:8080/`并按Enter键时，页面会默认重定向到首页。

2.路由的history模式

前面案例路由模式用的是hash模式，Vue Router还支持history模式的路由。我们只需要在`src/router/index.js`文件中更改路由模式即可，代码如下所示：

```
import { createRouter, createWebHistory, createWebHashHistory } from 'vue-
router'
......
const router = createRouter({
  routes,
  // history: createWebHashHistory() // 1.指定用hash路由
  history: createWebHistory() // 2.指定用history路由
})
export default router
```

保存代码，在浏览器中输入`http://localhost:8080/`按Enter键，在浏览器中显示的效果如图12-5所示。页面已重定向至首页，并且URL路径已切换至history模式，即路径不再使用#号拼接。

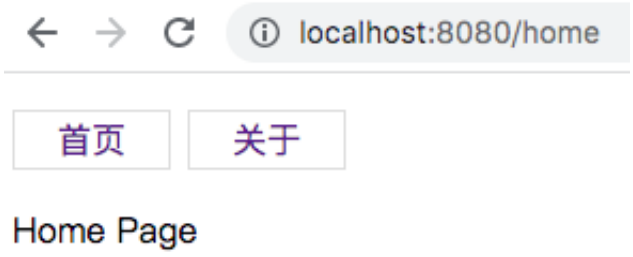

图12-5 配置history模式

3.路由的router-link组件

`<router-link>`是Vue Router的内置组件，我们可以使用它来创建链接并切换URL，从而使Vue Router可以在不重新加载页面的情况下更改URL。该组件还可以对以下属性进行配置。

（1）to属性

表示目标路由的链接。当`<router-link>`组件被单击后，内部会立刻把 "to" 的值传到 "router.push()" API 来实现界面跳转，因此这个值可以是一个字符串或是一个对象，代码如下所示。

```
<router-link class="tab"  to="/home">首页</router-link>
<router-link class="tab" :to="{ path: '/home' }">首页</router-link>
```

（2）replace属性

设置 "replace" 属性后，当`<router-link>`组件被单击时，会调用 "router.replace()" API 来实现页面跳转。这次页面跳转是直接替换当前页面，页面不会被压入浏览器的历史栈中。因此，页面跳转后浏览器无法使用返回功能，代码如下所示。

```
<router-link class="tab"  to="/home" replace>首页</router-link>
<router-link class="tab"  to="/about" replace>关于</router-link>
```

（3）active-class属性

设置`<a>`元素激活后应用的 class，默认的class是`router-link-active`。例如，当切换到首页后，可以按F12键打开调试工具查看元素结构，如图12-6所示。

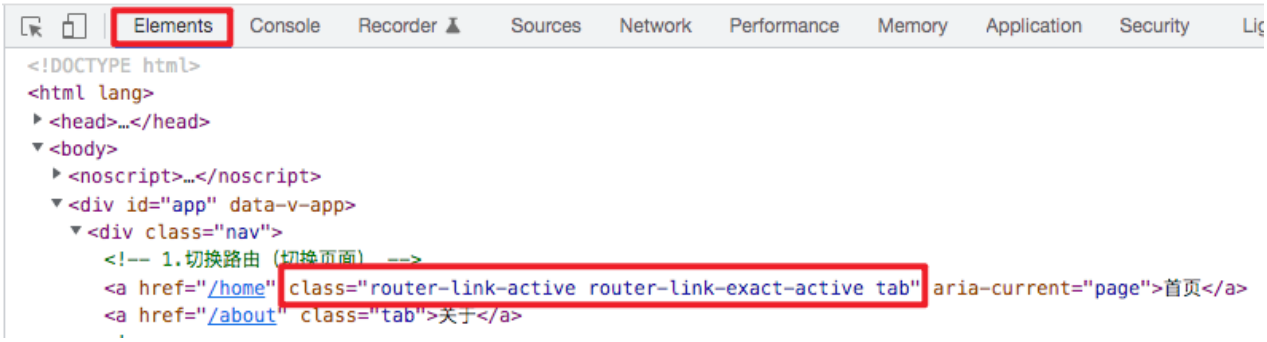

图12-6 激活a元素应用的class

如果想更改`<a>`元素被激活时默认的 class，可以给`<router-link>`组件添加`active-class`属性，代码如下所示。

```
<router-link class="tab" :to="{ path: '/home' }" active-class="why">首
页</router-link>
```

保存代码，按`F12`键打开调试工具，再次查看元素结构，如图12-7所示。被激活的`<a>`元素默认的class已改为`why`。

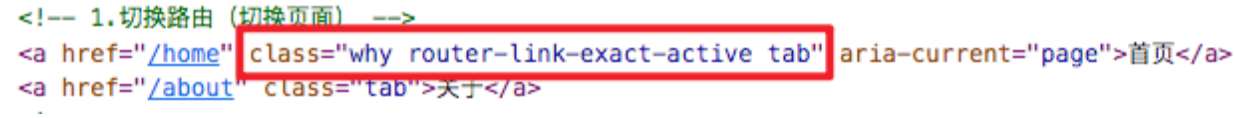

图12-7 自定义激活a元素应用的class

（4）exact-active-class属性：

链接精准激活（即URL和to上配置的路径需完全一样），用于设置`<a>`元素被激活的class，默认是`router-link-exact-active`。该属性的用法和`active-class`属性类似，代码如下所示。

```
<router-link :to="{ path: '/home' }" exact-active-class="coderwhy">首
页</router-link>
```

4.路由的懒加载

在真实开发中，一个项目可能会包含大量页面组件。如果这些组件都没有使用异步加载，那么在打包构建生产项目时，打包的JavaScript包会变得非常大，从而影响页面的首屏加载速度。

为了解决这个问题，我们可以将不同路由对应的组件分割成不同的代码块，然后在访问路由时才加载对应的组件。这种做法不仅更加高效，还可以提高首屏加载速度。实际上，Vue Router 默认就支持异步加载组件，也就是所谓的路由懒加载。

我们修改一下`src/router/index.js`文件，实现路由懒加载，代码如下所示：

```
// import Home from "../pages/Home.vue";  // 1.注释掉这些同步加载组件的代码
// import About from "../pages/About.vue";
......
const routes = [
  { path: '/', redirect: '/home' },
  // 2.路由懒加载，赖加载Home和About组件)
  { path: '/home', component: () => import('../pages/Home.vue') },
  { path: '/about', component: () => import('../pages/About.vue') }
]
......
```

可以看到，我们先注释之前同步加载组件的代码。这次与之前不同的是，这次component属性接收一个函数，该函数需要返回一个Promise对象。而import函数正好可以动态导入组件并返回一个Promise对象。

注意：component属性可接收一个组件，也可接收一个函数。当接收函数时，该函数需返回Promise对象。

保存代码后，项目依然可以正常运行。为了查看打包后的项目，我们在终端中输入"npm run build"命令来进行打包构建。这里分别演示了配置路由懒加载和未配置路由懒加载的两种情况，如图12-8所示。

- 左图是没有配置路由懒加载打包的结果。
  - `app.857ea648.js`文件存放的是我们编写的所有代码。
  - `chunk-vendors.c51a69b2.js`文件存放的是第三方库的代码。
- 右图是有配置路由懒加载打包的结果。
  - `app.181547e7.js`文件存放的是非异步组件的代码。
  - `chunk-2d21a719.39adf575.js`文件存放的是Home.vue页面组件的代码。
  - `chunk-2d207d33.3a92d3d6.js`文件存放的是About.vue页面组件的代码。
  - `chunk-vendors.708a7330.js`文件存放的是第三方库的代码。

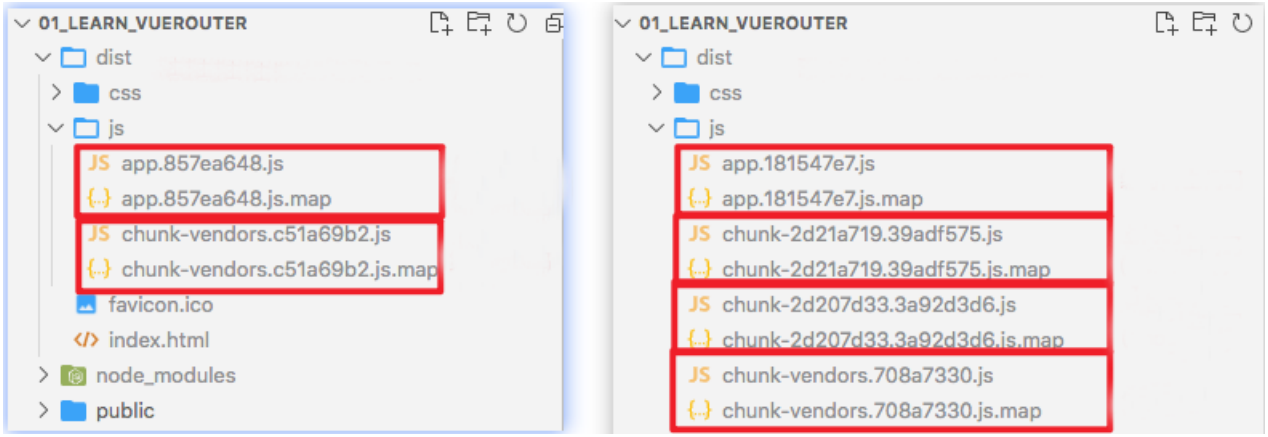

图12-8 路由懒加载打包结果

我们会发现配置了路由懒加载会进行分包，但分出来的包名称都是随机生成的。实际上，自webpack 3.x起，就支持对分包进行命名（chunk name）。只需在import函数的参数中添加**魔法注释**，例如`/* webpackChunkName: "自定义分包名" */`即可对分包进行命名，代码如下所示：

```
......
const routes = [
  { path: '/', redirect: '/home' },
  {
      path: '/home',
      // 1.import函数的参数添加了魔法注释，例如：/* webpackChunkName: "home-chunk"
*/
      component: () => import(/* webpackChunkName: "home-chunk" */
'../pages/Home.vue')
  },
  {
      path: '/about',
      component: () => import(/* webpackChunkName: "about-chunk" */
'../pages/About.vue')
  }
]
......
```

保存代码重新打包，效果如图12-9所示。这次打包后生成的包，已有对应的名称。

- `app.fe0d6d18.js`文件存放的是非异步组件的代码。
- `home-chunk-db823e5c.js`文件存放的是Home.vue页面组件的代码。
- `about-chunk-10edb3d7.js`文件存放的是About.vue页面组件的代码。
- `chunk-vendors.708a7330.js`文件存放的是第三方库的代码。

图12-9 给分包命名

5.路由其他属性

在配置路由映射时，除了可以配置`path`和`component`属性外，实际上路由还支持配置其他的属性，例如。

1. `name`属性：为路由添加一个独一无二的名称。
2. `meta`属性：为路由附加自定义数据。

下面给About.vue页面的路由配置`name`和`meta`属性，代码如下所示：

```
  {
    path: '/about',
    name: 'about', // 1.指定该路由的名称为: about
    component: () => import(/* webpackChunkName: "about-chunk" */
'../pages/About.vue'),
    meta: { // 2.给该路由添加自定义数据
      name: 'why',
      age: 18
    }
  }
```

接下来，可以在About.vue页面的`setup`函数中调用的`useRoute`函数（先了解）来获取对应的路由配置信息，代码如下所示：

```
<script>
import { useRoute } from 'vue-router'
export default {
  setup() {
    // 1.useRoute会返回当前路由信息。相当于在模板中使用$route。useRoute必须在setup中调用。
    const route = useRoute()
    console.log(route.name)  // 2.获取路由名称，打印 about
    console.log(route.meta) // 3.获取路由自定义数据，打印 {name: 'why', age: 18}
  }
}
</script>
```

12.3 Vue Router进阶知识

12.3.1 动态路由的匹配

很多时候，我们需要将符合特定匹配规则的路由映射到同一个组件。例如，我们有一个User.vue页面，它应该对所有用户进行渲染。

- 当访问"http://localhost:8080/user/why"时，应该渲染why用户的信息。
- 当访问"http://localhost:8080/user/kobe"时，应该渲染kobe用户的信息。

为了实现该功能，我们可以使用Vue Router的动态路由，即在路径中使用一个动态路径参数来实现，也称路径参数或路由参数。下面是实现该案例的具体步骤。

1.基本匹配规则

在`pages`文件夹下新建`User.vue`页面，代码如下所示。

```
<template>
  <div class="user">User Page</div>
</template>
```

接着，在`src/router/index.js`文件中添加`User.vue`页面的路由配置，代码如下所示：

```
......
const routes = [ // 1.配置路由映射表（路径-->组件）
  ......
  {
    // 2.动态路径参数以冒号开始，例如，:username 代表是动态路径参数。
    path: "/user/:username",
    component: () => import("../pages/User.vue")
  }
]
......
```

可以看到，我们在routes数组中添加了User.vue页面的路由配置信息，并给`path`属性指定了路由的匹配规则。其中以冒号（:）开头的路径，代表是路径参数。这样的路径，称为动态路由。

接着，在`src/app.vue`中添加一个切换到用户页面的按钮链接，代码如下所示：

```
<template>
  <div class="nav">
    ......
    <router-link class="tab" to="/about">关于</router-link>
    <router-link class="tab" to="/user/why">用户</router-link>
  </div>
  ......
</template>
```

保存代码，在浏览器中显示的效果如图12-10所示。当单击“用户”按钮或手动将URL的路径改为："/user/why"或"/user/kobe"时，页面上都会显示"User Page"。

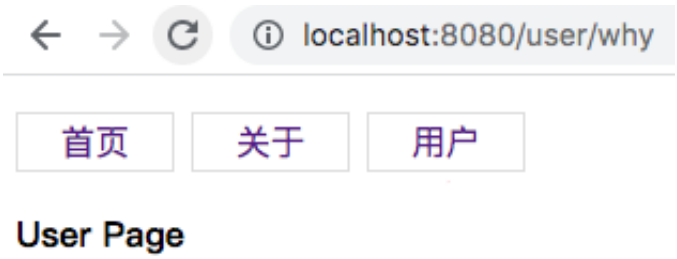

图12-10 动态路由匹配

一个路径参数使用冒号（:）标记，比如 :username。当匹配到一个路由时，路径参数值会被设置到`this.$route.params`中。于是，我们可以通过以下方法在 User.vue 组件中获取该路径参数值。

1. 在 template 中直接通过`$route.params`获取值。
2. 在 created 等生命周期中通过`this.$route.params`获取值。
3. 在 setup 中使用`vue-router`提供的`useRoute()`函数，该函数会返回一个存放当前路由信息的 Route 对象。

修改 User.vue 组件，代码如下所示：

```
<template>
  <div>
    <!-- 1.template中获取路径参数的值 -->
    <h4>User Page: {{$route.params.username}}</h4>
  </div>
</template>
<script>
  import { useRoute } from "vue-router";
  export default {
    created() {
      console.log(this.$route.params.username); // 2.created中获取路径参数值
    },
    setup() {
```

```
      const route = useRoute();
      console.log(route.params.username); // 3.useRoute函数获取路径参数值
    }
  }
</script>
```

保存代码，在浏览器中显示的效果如图12-11所示。当单击"用户"按钮或手动将URL的路径改为：`/user/why`或`/user/kobe`时，页面上都可以显示"User Page"和对应的路径参数值。

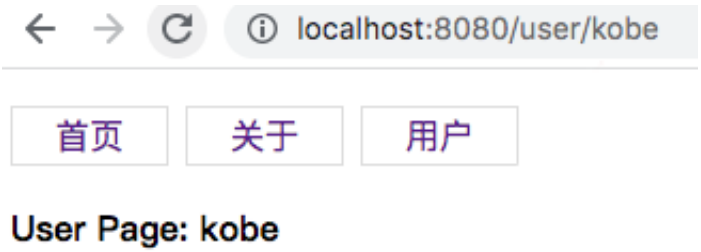

图12-11 动态路径参数

2.匹配多个参数

动态路由支持匹配多个参数，因此我们可以在路径中包括两个动态路径参数。例如，我们在 User.vue 页面的路由配置信息的path属性上，添加了一个新的动态路径参数`:id`，代码如下所示：

```
const routes = [
  ......
  {
    // 1.定义:username和:id两个动态路径参数
    path: "/user/:username/id/:id",
    component: () => import("../pages/User.vue")
  }
]
```

接着，修改`src/app.vue`中`<router-link>`组件to属性的值为`/user/why/id/0001`，代码如下所示：

```
<template>
  <div class="nav">
    ......
    <router-link class="tab" to="/user/why/id/0001">用户</router-link>
  </div>
  ......
</template>
```

修改 User.vue 页面，增加对路径参数（id）值的获取，代码如下所示：

```
<template>
  <div>
```

```
    <h4>User Page: {{$route.params.username}}-{{$route.params.id}}</h4>
  </div>
</template>
<script>
  import { useRoute } from "vue-router";
  export default {
    created() {
      console.log(this.$route.params.username, this.$route.params.id);
    },
    setup() {
      const route = useRoute();
      console.log(route.params.username, route.params.id);
    }
  }
</script>
```

保存代码，在浏览器中显示的然后单击用户按钮，效果如图12-12所示。

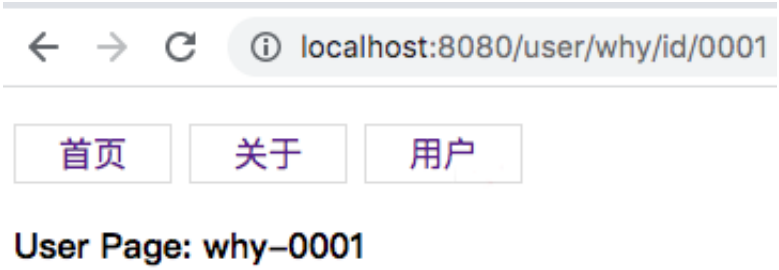

图12-12 匹配多个动态路由参数

最后，总结一下获取动态路径参数值的规则，如图12-13所示。

匹配模式	匹配路径	$route.params
/user/:username	/users/why	`{ username: 'why' }`
/user/:username/id/:id	/users/why/id/0001	`{ username: 'why', id: '0001' }`

图12-13 获取动态路径参数值的规则

3.NotFound页面

对于未匹配到的路由，通常会让其跳转到一个固定的页面，比如NotFound（404）页面。为了实现404页面，我们可以编写一个专门用于匹配所有页面的路由，将其指向404页面。

在`src/router/index.js`文件中，添加`NotFound.vue`页面的路由配置，给path属性指定匹配所有页面的规则。代码如下所示：

```
const routes = [
  ......
  {
    path: "/:patchMatch(.*)", // 1.使用通配符*来匹配任意路径，通配符路由应放在最后
    component: () => import("../pages/NotFound.vue")
  }
]
```

接着，在`pages`文件夹下新建`NotFound.vue`页面，代码如下所示。

```
<template>
  <div>
    <h3>Page Not Found 404</h3>
    <p>您打开的路径页面不存在，请不要使用我们家的应用程序了~</p>
    <h4>{{$route.params.pathMatch}}</h4>
  </div>
</template>
```

可以看到，这里我们通过`$route.params.pathMatch`来获取路径参数值。

保存代码，在浏览器中显示的效果如图12-14所示。这时，当我们手动将URL的路径改为：`/product/1001`（一个没有注册过的路径）时，页面上会显示NotFound.vue页面的内容和对应路径参数值。

图12-14 匹配不存在的页面

匹配所有页面的规则还有另外一种写法，代码如下所示：

```
const routes = [
  {
    path: "/:patchMatch(.*)*",
    component: () => import("../pages/NotFound.vue")
  }
]
```

可以看到，我们在`/:pathMatch(.*)`后面又加了一个`*`。如果省略了最后的`*`，在解析或跳转时，参数中的`/`字符将被编码。如果打算直接使用未匹配的路径名称导航到该路径，这个`*`是必要的。并且多加了一个`*`号那么会把路由参数解析为数组格式。

12.3.2 嵌套路由的使用

目前我们匹配的`Home.vue`、`About.vue`和`User.vue`页面等都属于底层路由（即一级路由），我们可以在它们之间任意来回切换。大部分情况下，像`Home.vue`页面本身也会存在多个组件来回切换，例如：`Home.vue`页面中又包括`Product.vue`、`Message.vue`页面，它们可以在`Home.vue`页面内部来回切换。

为了实现该功能，需要用到嵌套路由，我们可以在`Home.vue`页面中也使用`<router-view>`组件来占位需要渲染的组件。下面演示嵌套路由使用。

在`pages`文件夹下分别新建`HomeMessage.vue`和`HomeShops`页面，代码如下所示。

HomeMessage.vue 页面

```
<template>
  <div class="home-message">
    <h4>Home Message 组件</h4>
    <div>消息通知...</div>
  </div>
</template>
```

HomeShops.vue 页面

```
<template>
  <div class="home-shops">
    <h4>Home Shops 组件</h4>
    <div>商品信息...</div>
  </div>
</template>
```

接着，在`src/router/index.js`文件中添加`HomeMessage.vue`和`HomeShops.vue`页面的路由配置，代码如下所示：

```
......

const routes = [
  ......
  {
    path: '/home',
    component: ()=> import(/* webpackChunkName: "home-chunk" */
'../pages/Home.vue'),
    children: [ // 1.在Home页面下注册二级路由
      {
```

```
        path: "message", // 2.二级路由path不支持/message或/home/message，直接填
message即可
        component: () => import("../pages/HomeMessage.vue")
      },
      {
        path: "shops",
        component: () => import("../pages/HomeShops.vue")
      }
    ]
  }
]
......
```

从上面代码可以看到，我们在`routes`数组中`Home.vue`页面的路由配置中添加了`children`属性，该属性用来给`Home.vue`页面注册子路由。

接着，在`children`属性中分别注册了`HomeMessage`和`HomeShops`页面的路由配置信息，其中`HomeMessage.vue`页面路由路径为：`/home/message`，`HomeShops.vue`页面的路由路径为：`/home/shops`。

注意：二级路由`path`属性无需添加父路由前缀，比如上面二级路由`path`属性值无需添加`/`或`/home`等前缀。

接着，修改`src/pages/Home.vue`页面，代码如下所示：

```
<template>
  <div class="home">
    <h3>Home Page</h3>
    <router-link class="btn" to="/home/message">消息页</router-link>
    <router-link class="btn" to="/home/shops">商品页</router-link>
    <!-- 路由组件的占位 -->
    <router-view></router-view>
  </div>
</template>
<style scoped>
.btn{
  margin: 4px;
  padding: 3px 5px;
  text-decoration: none;
  border: 1px dashed #ddd;
}
</style>
```

从上面的代码可以看到，我们完成了以下两件事情。

1. 使用`<router-link>`组件实现单击切换路由。当单击"消息页"按钮时，URL路径会变成"/home/message"，当单击"商品页"按钮时，URL路径会变成"/home/shops"。
2. 使用`<router-view>`组件为页面提供占位。当单击"消息页"按钮时，`<router-view>`组件处渲染`HomeMessage.vue`页面，当单击"商品页"按钮时，`<router-view>`组件处渲染

`HomeShops.vue`页面。

保存代码，在浏览器中显示的效果如图12-15所示。

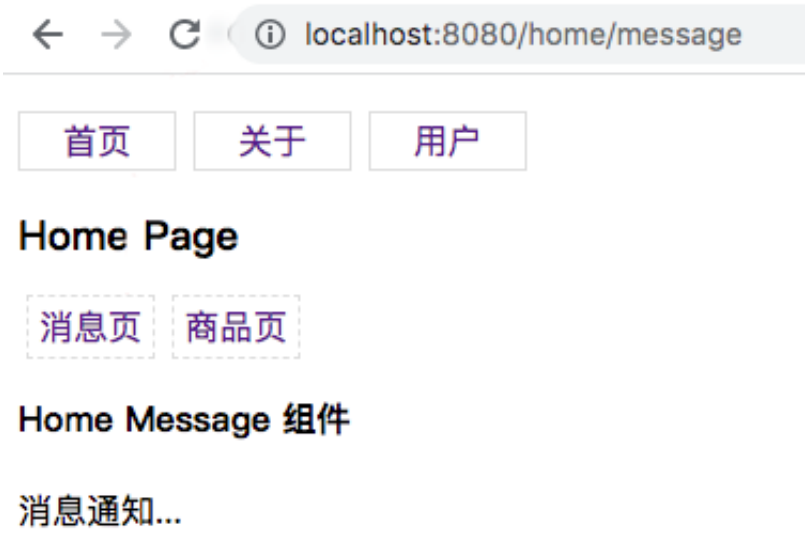

图12-15 路由嵌套

下面进一步完善该案例，例如当用户单击“首页”按钮时，默认显示消息页面的内容。实际上，我们只需要在刚才Home.vue页面的路由配置的children属性中添加路由重定向的配置，即可实现该功能，代码如下所示：

```
const routes = [
  ......
  {
    path: '/home',
    component: ()=> import(/* webpackChunkName: "home-chunk" */
'../pages/Home.vue'),
    children: [ // 1.在Home页面下注册二级路由
      {
        path: '',
        redirect: '/home/message' // 2.访问/home路径时，重定向到/home/message路径
      },
      {
        path: "message",
        component: () => import("../pages/HomeMessage.vue")
      },
      ......
    ]
  }
]
```

保存代码，在浏览器中当单击“首页”按钮，Home.vue页面就默认显示为消息页面。

12.3.3 编程式导航的使用

除了可以使用`<router-link>`组件来实现页面导航，Vue Routerd的实例还提供一些导航相关的方法，比如`router.push`方法。当我们通过调用方法来实现页面导航，这种方式称为编程式导航。

下面我们来看一下如何通过代码实现界面跳转。

1.代码实现界面跳转

我们来实现单击“关于”按钮跳转到关于页面的功能。这次我们不使用`<router-link>`组件，而是通过代码实现界面跳转。修改`src/App.vue`组件，代码如下所示：

```
<template>
  <div class="nav">
    ......
    <router-link class="tab" to="/user/why/id/0001">用户</router-link>
    <button @click="jumpToAbout">关于</button>
    ......
  </div>
</template>
<script>
import { useRouter } from 'vue-router';
export default {
  name: 'App',
  setup() {
    const router = useRouter() // 1.拿到router对象，即创建的路由对象
    const jumpToAbout= ()=>{   // 2.监听button单击事件
      router.push("/about")    // 3.跳转到关于页面
    }
    return { jumpToAbout }
  }
}
</script>
```

从上面代码可以看到，我们首先在`setup`中使用`vue-router`提供`useRouter()`这个 hook 来获取路由对象，该对象可以用于实现页面跳转和返回等功能。然后，我们监听`<button>`元素的单击事件，当用户单击 "关于"按钮 时，会调用`router.push()`方法实现页面跳转。

保存代码，在浏览器中显示的效果如图12-16所示。

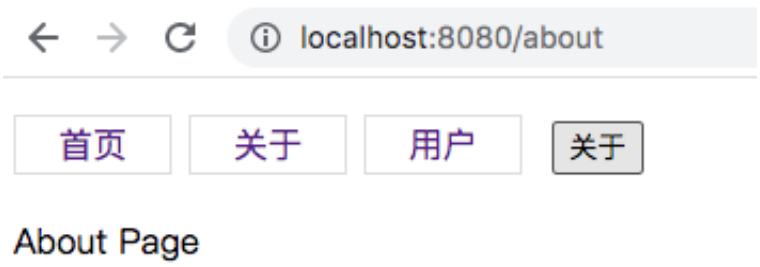

图12-16 编写代码切换页面

其实，`push`方法不仅可以接收一个字符串类型的参数，还支持接收一个对象类型的参数，如下代码所示：

```
const jumpToAbout= ()=>{
  // router.push("/about")  // 1.接收字符串类型参数
  router.push({             // 2.接收对象类型参数，功能和上一样。
    path: '/about'          // 3.指定跳转页面路径
  })
}
```

最后，再看一下Options API的实现（更推荐setup语法），我们修改`src/App.vue`组件，代码如下所示：

```
<template> ..... </template>
<script>
export default {
  name: 'App',
  methods: {
    jumpToAbout() {
      // this.$router.push('/about')
      this.$router.push({
        path: '/about'
      })
    }
  },
  setup() {
    return {}
  }
}
</script>
```

2.query参数

当单击"关于"按钮跳转到"关于"页面时，实际上可以通过查询字符串（query）的方式给目标页面传递参数。以下是使用 setup 和 Options API 两种方式，通过 query 参数向目标页面传递参数的示例。

（1）setup方式（推荐），代码如下所示：

```
const jumpToAbout= ()=>{
  // router.push("/about?name=coder&age=20") // 1.通过URL查询字符串方式给目标页面传递参数
  router.push({
    path: '/about',
    query: { // 2.通过query属性给目标页面传递参数
      name: 'coder',
      age: 20
    }
  })
}
```

（2）Options API 方式，代码如下所示：

```
jumpToAbout() {
  // this.$router.push('/about?name=coder&age=20')
  this.$router.push({
    path: '/about',
    query: {
      name: 'coder',
      age: 20
    }
  })
}
```

接着，我们可以在关于页面通过`route`对象来获取`query`参数。修改`src/pages/About.vue`文件，代码如下所示：

```
<template>
  <div class="about">
    About Page
    <!-- 1.tempalte直接获取query参数-->
    <p>{{$route.query.name}}-{{$route.query.age}}</p>
  </div>
</template>
<script>
import { useRoute } from 'vue-router'
export default {
  name: 'About',
  setup() {
    const route = useRoute();
    console.log(route.name, route.meta) // 其他参数
    console.log(route.query) // 2.通过route获取query参数，打印{name: 'coder', age: '20'}
    return {}
  }
}
</script>
```

从上面代码可以看到，我们分别从`<template>`和`setup()`函数中获取了`query`参数。

- 在`<template>`中，直接通过`$route.query`获取。
- 在`setup()`函数中，需要使用`vue-router`提供的`useRoute()`这个hook获取。

保存代码，在浏览器中显示的效果如图12-17所示。

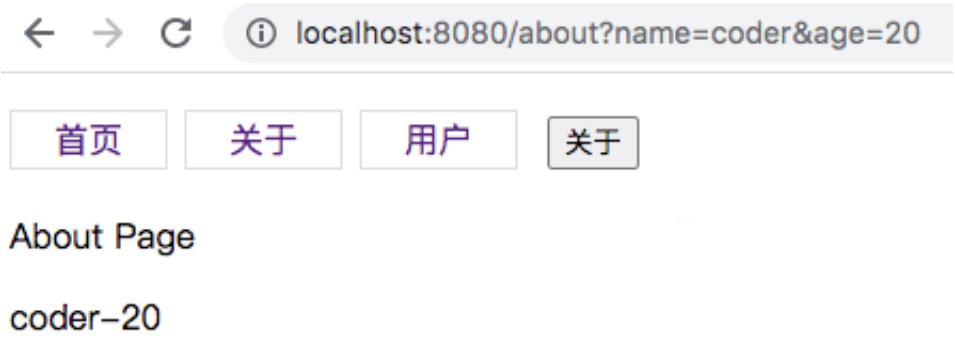

图12-17 query方式传递参数

3.替换当前的位置

使用`push`方法进行界面跳转时，默认会进行压栈操作。当用户单击页面上的返回时，可以回退到上一个页面。但是，如果希望当前页面进行替换操作，不具备回退功能，可以使用`router`对象的`replace`方法。

`replace`方法的使用语法，如图12-18所示。

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

图12-18 replace的使用语法

声明式 replace 的用法，代码如下所示：

```
<router-link to="/about" replace>关于</router-link>
```

编程式 replace 方法的用法，与 push 方法使用方式一样，代码如下所示：

```
const jumpToAbout= ()=>{
  // router.push("/about")
  router.replace("/about")
}
```

4.页面的前进后退

想要实现页面的前进后退功能，可以使用`router.go`方法，代码如下所示：

```
// 1.向前移动一条记录，与router.forward()相同
router.go(1)
// 2.返回一条记录，与router.back() 相同
router.go(-1)
// 3.前进3条记录(记录可理解为页面，即前进3个页面)
router.go(3)
// 4.如果没有那么多记录，静默失败
router.go(-100)
router.go(100)
```

另外，router对象还提供 了back 和 forward 方法。

- back方法：通过调用history.back()方法回溯历史，相当于router.go(-1)。
- forward方法：通过调用history.forward()方法在历史中前进，相当于router.go(1)。

12.3.4 路由内置组件的插槽

前面我们已经学习了Vue Router内置的`<router-link>`和`<router-view>`组件。接下来，我们将继续学习它们的高阶API：作用域插槽（slot）。

1.`<router-link>`组件的作用域插槽

`<router-link>`组件通过一个作用域插槽暴露底层的定制能力。这是一个更高阶的 API，主要面向库作者，但也可以为开发者提供便利。在`vue-router3.x`的时候，`<router-link>`有一个`tag`属性，可以决定`<router-link>`到底渲染成什么元素，但是从`vue-router4.x`开始，该属性被移除了，而给我们提供了更加具有灵活性的`v-slot`的方式来定制渲染的内容。而 v-slot 的使用主要分为两个步骤。

1. 需要添加custom属性，表示整个元素要自定义。如果不添加该属性，自定义的内容会被包裹在一个`<a>`元素中。
2. 使用 v-slot 作用域插槽来获取内部传递给我们的值，例如：
    - href：解析后的URL。
    - route：解析后规范化的route对象。
    - navigate：触发导航的函数。
    - isActive：匹配状态。
    - isExactActive：精准匹配状态。

下面使用`<router-link>`组件的作用域插槽来自定义之前的**首页**按钮。我们继续修改`/src/App.vue`组件，代码如下所示：

```
<template>
  <div class="nav">
    <!-- 1.给router-link添加了custom属性和v-slot指令 -->
    <router-link class="tab" to="/home" custom v-slot="props">
      <strong @click="props.navigate">首页：</strong>
      <span>{{props.href}}</span>
      <span> - {{props.isActive}}</span>
```

```
      <!-- todo ...除了以上的元素，还支持插入自定义组件 -->
    </router-link>
    <router-link class="tab" to="/about">关于</router-link>
  ......
  </div>
  <!-- 2.路由组件占位 -->
  <router-view></router-view>
</template>
```

从上面代码可以看到，这里主要实现了以下两个功能。

1. 在`<router-link>`组件上添加了`custom`属性（表示整个元素要自定义）和`v-slot`指令。该指令通过`props`来接收插槽内部提供的参数，例如`props.navigate`、`props.href`和`props.isActive`等。
2. 给`<router-link>`组件的默认插槽中插入了`<strong>`和两个`<span>`元素。接着，给`<strong>`元素的`click`事件绑定了作用域插槽提供的`navigate`函数。当单击""**元素时，会触发导航函数进行页面跳转。然后，给两个`<span>`元素分别绑定了作用域插槽提供的`href`和`isActive`值，这里仅作为显示使用。**

保存代码，在浏览器中显示的效果如图12-19所示。

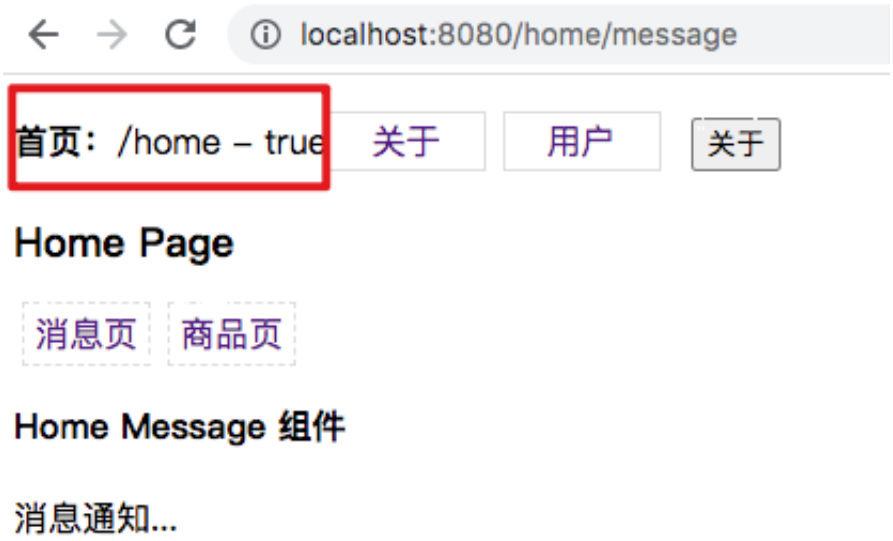

图12-19 rouer-link插槽的使用

2.router-view组件的v-slot

`<router-view>`组件也提供了一个`v-slot` API，可以使用`<transition>`和`<keep-alive>`组件来包裹路由组件。下面使用`<router-view>`作用域插槽来给页面添加过度动画和缓存的功能。

我们继续修改`/src/App.vue`组件，代码如下所示：

```
<template>
  <div class="nav">......</div>
  <!-- 2.路由组件占位，给router-view添加了v-slot指令 -->
  <router-view v-slot="props">
    <transition name="why">
```

```
      <component :is="props.Component"></component>
    </transition>
  </router-view>
</template>
......
<style>
.why-enter-from,
.why-leave-to {
  opacity: 0;
}
.why-enter-active,
.why-leave-active {
  transition: opacity 1s ease;
}
</style>
```

从上面代码可以看到，主要实现了以下两个功能。

1. 在`<router-view>`组件上添加了`v-slot`指令，通过props来接收内部提供的插槽参数。
2. 在`<router-view>`组件的默认插槽中插入`<transition>`组件。其中`<transition>`组件用于为页面组件添加过渡动画，指定的过渡动画类名为why，并在style标签中编写了相应的动画样式。接着，将插槽提供的`props.Component`属性动态绑定到`<component>`组件的`is`属性上。

保存代码，运行在浏览器后，再次切换页面就会有过度效果。

除了可以添加过渡效果，我们还可以使用`<keep-alive>`组件为页面添加缓存功能，代码如下所示：

```
<template>
  ......
  <router-view v-slot="props">
      <keep-alive>
        <component :is="props.Component"></component>
      </keep-alive>
  </router-view>
</template>
```

12.3.5 动态添加路由

前面我们都是在`routes`选项中提前配置好路由，但是在某些情况下，我们需要在应用程序运行后再动态添加或删除路由，例如根据登录用户的权限，动态注册不同的路由。这时可以使用`addRoute`方法动态添加路由。

1.addRoute

`addRoute`方法可以动态添加一条新的路由规则。如果该路由规则有`name`属性，并且已经存在一个与之相同的名字，则会覆盖原有的规则。

下面演示`addRoute`方法的使用，修改`src/router/index.js`文件，调用router对象的`addRoute`方法来动态添加`categoryRoute`路由，代码如下所示：

```
......
// 3.商品分类页面的路由配置
const categoryRoute = {
  path: "/category",
  component: ()=> import('../pages/Category.vue')
}
// 4.动态添加顶级路由对象
router.addRoute(categoryRoute)
export default router
```

接着，在`pages`目录下新建`Category.vue`页面，代码如下所示：

```
<template>
  <div class="category"> <h4>Category</h4> </div>
</template>
```

保存代码，运行在浏览器后，在浏览器中输入`http://localhost:8080/category`时，可以正常显示商品分类页面，说明动态注册的路由生效了。

其实，`addRoute`方法不仅可以添加顶级路由，还支持添加二级路由。之前首页已经有了消息通知和商品这两个二级页面，现在我们继续添加一个评论（`HomeComment.vue`）二级页面。

我们在`pages`目录下新建`HomeComment.vue`路由组件，代码如下所示：

```
<template>
  <div class="home-comment">
    <h4>Home Comment 组件</h4>
    <div>评论页面...</div>
  </div>
</template>
```

接着，修改`src/router/index.js`文件，调用`addRoute`方法来动态添加评论页面作为二级路由，该方法需要接收两个参数。参数一为现有路由名称，参数二为现有路由的子路由。代码如下所示：

```
const routes = [
  {
    path: '/home',
    name: 'home', // 路由的name，下面addRoute方法需要用到
    ......
  }
  ......
]
......
// 5.给现有路由（home）增加二级路由
```

```
router.addRoute("home", {
  path: "comment",
  component: () => import("../pages/HomeComment.vue")
})
export default router
```

注意：`router.addRoute`方法在接收一个参数时添加的是一级路由，接收两个参数时添加的是二级路由。

保存代码，在浏览器中显示的效果如图12-20所示。

图12-20 动态添加二级路由

2.动态路由补充

除了可以动态添加路由，Vue Router 还提供了三种方式可以动态删除路由。

方式一：添加一个相同名字（name）的路由，会覆盖之前相同名的路由。

```
router.addRoute({ path: '/about', name: 'about', component: About })
// 添加路由
// 1.添加相同名字路由时，会删除之前相同名字的路由。
router.addRoute({ path: '/other', name: 'about', component: Other })
```

方式二：通过 removeRoute 方法，传入路由的名称。

```
router.addRoute({ path: '/about', name: 'about', component: About })
// 添加路由
router.removeRoute('about') // 2.删除 about 路由
```

方式三：通过调用 `router.addRoute()` 返回的回调函数。

```
const removeRoute = router.addRoute(routeRecord) // 添加路由
removeRoute() // 3.删除所添加的路由
```

除此之外，Vue Router的实例还提供很多常用的方法，比如。

- `router.hasRoute()`：检查路由是否存在。
- `router.getRoutes()`：获取一个包含所有路由记录的数组。

12.3.6 路由守卫

路由导航（界面跳转）在Vue Router中可通过`<router-link>`组件实现，也可通过代码来实现。在某些情况下，我们通常希望能拦截路由导航。比如在进入某个路由之前，先判断用户是否已登录，如登录则放行，否则导航到登录页面。

其实，Vue Router已经给我们提供了该功能，并将拦截路由称为导航守卫（也可称路由守卫）。导航守卫顾名思义就是专门用来守卫导航，可以灵活控制路由的跳转或取消等。导航守卫通常有三种方式来实现：全局路由守卫、单个路由独享的守卫或组件内的守卫。在开发中用的最多的是全局路由守卫中的全局前置守卫。

1.全局前置守卫

在全局路由守卫中，用的最多的就是全局前置守卫。我们可以用`router.beforeEach`方法注册一个路由的全局前置守卫，如下代码所示：

```
const router = createRouter({ ... })
......
router.beforeEach((to, from) => {
/**
 * 返回值的作用：
 *     1.false: 取消当前导航
 *     2.undefined或不返回: 进行默认导航
 *     3.字符串: 一个路由路径
 *     4.对象: 如{path: "/login", query: ....}对象
 */
  return false
})
export default router
```

可以看到，上面的`beforeEach`函数需接收一个回调函数，该回调函数有两个参数：

- `to`：将进入的路由`Route`对象。
- `from`：将离开的路由`Route`对象。

另外，`beforeEach`回调函数支持返回值的类型：

- `false`：取消当前导航。
- `undefined`或不返回：进行默认导航。
- 字符串：一个路由路径，如：`"/about"`
- 对象：如`{path: '/login', query:{}, params: {}}`对象。

下面演示`router.beforeEach`方法的使用，我们继续给`01_learn_vuerouter`项目添加一个新的功能：只有登录后才能看到其他页面，否则跳转到登录页面。

修改`src/router/index.js`文件，添加`router.beforeEach`函数来拦截路由导航，代码如下所示：

```
......
// 6.全局前置守卫
router.beforeEach((to, from) => {
  if (to.path !== '/login') { // 如果不是登录页面
    const token = window.sessionStorage.getItem('token')
    if (!token) { // 通过token判断是否登录，没登录则导航到/login页面
      return {
        path: '/login'
      }
    }
  }
})
export default router
```

可以看到，在`router.beforeEach`方法的回调函数中，我们先判断当前访问的路径是否为登录页。如果不是登录页，则继续判断本地是否有token。如果没有token，代表用户未登录，我们直接导航到登录页；否则进行默认导航。

接着，在`pages`目录下新建一个`Login.vue`登录页面，当用户单击“登录”按钮时，我们先把token存到sessionStorage 中，代表用户已经登录，然后跳转到首页，代码如下所示：

```
<template>
  <div class="login"> <button @click="loginClick">登录</button>
</div>
</template>
<script>
  import { useRouter } from 'vue-router';
  export default {
    setup() {
      const router = useRouter();
      const loginClick = () => { // 1.模拟登录功能
        window.sessionStorage.setItem("token", "why") // 2.存储登录信息
        router.push("/home") // 3.跳转到首页
      }
      return { loginClick }
    }
  }
</script>
```

修改`src/router/index.js`文件，在`routes`数组中注册登录页面，代码如下所示：

```
const routes = [
  ......
  {
    path: "/login",
    component: () => import("../pages/Login.vue")
  },
  ......
]
```

保存代码，在浏览器中显示的效果如图12-21所示。由于页面尚未登录，sessionStorage 中也没有存储 token 信息。因此，导航守卫会直接将用户导航到登录页面。这时如果我们单击了“登录”按钮并将token信息存储到sessionStorage中，就可以成功导航到首页。

图12-21 显示登录页面

2.其他导航守卫

除了上面讲解的 `beforeEach` 导航守卫方法，Vue Router 还提供了许多其他导航守卫方法，旨在在某个时刻给予我们回调，以更好地控制程序的流程或功能。

感兴趣的读者可参考官方文档：https://router.vuejs.org/zh/guide/advanced/navigation-guards.html

12.4 本章小结

本章内容如下。

- 前端路由实现有两种方式：hash模式和history模式。
  - hash模式通过监听URL中的hash值的变化来实现前端路由，而无需新页面。当URL中的hash值发生变化时，Vue Router会根据hash值来切换显示内容。
  - history模式通过HTML5新增的接口来实现前端路由，而无需刷新页面。当URL发生变化时，Vue Router会根据URL来切换显示内容。
- Vue Router基本使用：首先需要通过createRouter函数来创建路由对象，该函数需要接收一个包含routes配置和history路由模式的对象。然后使用app.use函数来安装该路由对象。最后，在页面中添加router-view占位，用于显示路由组件。
- Vue Router进阶使用：除了基本使用外，Vue Router还提供了进阶功能，包括动态路由、嵌套路由、编程式导航、动态添加路由和导航守卫等。这些功能可以帮助我们更好地管理前端路由。