React-Router路由详解

王红元 coderwhy

目录
content

认识前端路由

- **路由其实是网络工程中的一个术语：**
  - 在架构一个网络时，非常重要的两个设备就是路由器和交换机。
  - 当然，目前在我们生活中路由器也是越来越被大家所熟知，因为我们生活中都会用到路由器：
  - 事实上，路由器主要维护的是一个映射表；
  - 映射表会决定数据的流向；

- **路由的概念在软件工程中出现，最早是在后端路由中实现的，原因是web的发展主要经历了这样一些阶段：**
  - 后端路由阶段；
  - 前后端分离阶段；
  - 单页面富应用（SPA）；

后端路由阶段

- 早期的网站开发整个HTML页面是由**服务器来渲染**的.
  - 服务器直接生产渲染好对应的HTML页面, 返回给客户端进行展示.
- 但是, 一个网站, **这么多页面服务器如何处理呢?**
  - 一个页面有自己对应的网址, 也就是URL;
  - URL会发送到服务器, 服务器会通过正则对该URL进行匹配, 并且最后交给一个Controller进行处理;
  - Controller进行各种处理, 最终生成HTML或者数据, 返回给前端.
- 上面的这种操作, 就是**后端路由**:
  - 当我们页面中需要请求不同的**路径**内容时, 交给服务器来进行处理, 服务器渲染好整个页面, 并且将页面返回给客户端.
  - 这种情况下渲染好的页面, 不需要单独加载任何的js和css, 可以直接交给浏览器展示, 这样也有利于SEO的优化.
- **后端路由的缺点:**
  - 一种情况是整个页面的模块由后端人员来编写和维护的;
  - 另一种情况是前端开发人员如果要开发页面, 需要通过PHP和Java等语言来编写页面代码;
  - 而且通常情况下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的事情;

前后端分离阶段

- **前端渲染的理解:**
  - 每次请求涉及到的静态资源都会从**静态资源服务器获取**，这些资源**包括HTML+CSS+JS**，然后在前端对这些请求回来的资源进行渲染;
  - 需要注意的是，客户端的每一次请求，都会从静态资源服务器请求文件;
  - 同时可以看到，和之前的后端路由不同，这时后端只是负责提供API了;
- **前后端分离阶段:**
  - 随着Ajax的出现, 有了前后端分离的开发模式;
  - 后端只提供API来返回数据，前端通过Ajax获取数据，并且可以通过JavaScript将数据渲染到页面中;
  - 这样做最大的优点就是前后端责任的清晰，后端专注于数据上，前端专注于交互和可视化上;
  - 并且当移动端(iOS/Android)出现后，后端不需要进行任何处理，依然使用之前的一套API即可;
  - 目前比较少的网站采用这种模式开发;
- **单页面富应用阶段:**
  - 其实SPA最主要的特点就是在前后端分离的基础上加了一层前端路由.
  - 也就是前端来维护一套路由规则.
- **前端路由的核心是什么呢？改变URL，但是页面不进行整体的刷新。**

URL的hash

- **前端路由是如何做到URL和内容进行映射呢？监听URL的改变。**
- **URL的hash**
  - URL的hash也就是锚点(#), 本质上是改变window.location的href属性;
  - 我们可以通过直接赋值location.hash来改变href, 但是页面不发生刷新;

```html
<div id="app">
  <a href="#/home">home</a>
  <a href="#/about">about</a>
  <div class="router-view"></div>
</div>
```

```js
// 1.获取router-view
const routerViewEl = document.querySelector(".router-view");

// 2.监听hashchange
window.addEventListener("hashchange", () => {
  switch(location.hash) {
    case "#/home":
      routerViewEl.innerHTML = "home";
      break;
    case "#/about":
      routerViewEl.innerHTML = "about";
      break;
    default:
      routerViewEl.innerHTML = "default";
  }
})
```

- **hash的优势就是兼容性更好，在老版IE中都可以运行，但是缺陷是有一个#，显得不像一个真实的路径。**

HTML5的History

- history接口是HTML5新增的，它有六种模式改变URL而不刷新页面：
  - replaceState：替换原来的路径；
  - pushState：使用新的路径；
  - popState：路径的回退；
  - go：向前或向后改变路径；
  - forward：向前改变路径；
  - back：向后改变路径；

```
// 1.获取router-view
const routerViewEl = document.querySelector(".router-view");

// 2.监听所有的a元素
const aEls = document.getElementsByTagName("a");
for (let aEl of aEls) {
  aEl.addEventListener("click", (e) => {
    e.preventDefault();
    const href = aEl.getAttribute("href");
    console.log(href);
    history.pushState({}, "", href);
    historyChange();
  })
}
```

```
// 3.监听popstate和go操作
window.addEventListener("popstate", historyChange);
window.addEventListener("go", historyChange);

// 4.执行设置页面操作
function historyChange() {
  switch(location.pathname) {
    case "/home":
      routerViewEl.innerHTML = "home";
      break;
    case "/about":
      routerViewEl.innerHTML = "about";
      break;
    default:
      routerViewEl.innerHTML = "default";
  }
}
```

认识react-router

- **目前前端流行的三大框架，都有自己的路由实现：**
  - Angular的ngRouter
  - React的ReactRouter
  - Vue的vue-router
- **React Router在最近两年版本更新的较快，并且在最新的React Router6.x版本中发生了较大的变化。**
  - 目前React Router6.x已经非常稳定，我们可以放心的使用；
- **安装React Router：**
  - 安装时，我们选择react-router-dom；
  - react-router会包含一些react-native的内容，web开发并不需要；

```
npm install react-router-dom
```

Router的基本使用

- react-router最主要的API是给我们提供的一些组件：
- BrowserRouter或HashRouter
  - Router中包含了对路径改变的监听，并且会将相应的路径传递给子组件；
  - BrowserRouter使用history模式；
  - HashRouter使用hash模式；

```
<React.StrictMode>
  <HashRouter>
    <App />
  </HashRouter>
</React.StrictMode>
```

路由映射配置

- **Routes：包裹所有的Route，在其中匹配一个路由**
  - Router5.x使用的是Switch组件
- **Route：Route用于路径的匹配；**
  - path属性：用于设置匹配到的路径；
  - element属性：设置匹配到路径后，渲染的组件；
    - ✓ Router5.x使用的是component属性
  - exact：精准匹配，只有精准匹配到完全一致的路径，才会渲染对应的组件；
    - ✓ Router6.x不再支持该属性

```
<Routes>
  <Route path='/' element={<Home/>}/>
  <Route path='/about' element={<About/>}/>
  <Route path='/profile' element={<Profile/>}/>
</Routes>
```

路由配置和跳转

- **Link和NavLink：**
  - 通常路径的跳转是使用Link组件，最终会被渲染成a元素；
  - NavLink是在Link基础之上增加了一些样式属性（后续学习）；
  - to属性：Link中最重要的属性，用于设置跳转到的路径；

```jsx
<div className='header'>
  <Link to="/">首页</Link>
  <Link to="/about">关于</Link>
  <Link to="/profile">我的</Link>
</div>
```

```ts
interface LinkProps
  extends Omit<
    React.AnchorHTMLAttributes<HTMLAnchorElement>,
    "href"
  > {
  replace?: boolean;
  state?: any;
  to: To;
  reloadDocument?: boolean;
}
```

NavLink的使用

- **需求：路径选中时，对应的a元素变为红色**
- **这个时候，我们要使用NavLink组件来替代Link组件：**
  - style：传入函数，函数接受一个对象，包含isActive属性
  - className：传入函数，函数接受一个对象，包含isActive属性

```
<NavLink to="" className={this.getActiveClass}>首页</NavLink>
<NavLink to="about" className={this.getActiveClass}>关于</NavLink>
<NavLink to="profile" className={this.getActiveClass}>我的</NavLink>
```

```
getActiveClass({isActive}) {
  return classNames({"link-active": isActive})
}
```

- **默认的activeClassName：**
  - 事实上在默认匹配成功时，NavLink就会添加上一个动态的active class；
  - 所以我们也可以直接编写样式
- **当然，如果你担心这个class在其他地方被使用了，出现样式的层叠，也可以自定义class**

Navigate导航

- **Navigate用于路由的重定向，当这个组件出现时，就会执行跳转到对应的to路径中：**
- **我们这里使用这个的一个案例：**
  - 用户跳转到Profile界面；
  - 但是在Profile界面有一个isLogin用于记录用户是否登录：
    - ✓ true：那么显示用户的名称；
    - ✓ false：直接重定向到登录界面；

首页关于我的用户

Login Page

首页关于我的用户

User

用户名: coderwhy

- **我们也可以在匹配到/的时候，直接跳转到/home页面**

```
<Route path="/" element={<Navigate to="/home"/>}/>
```

Not Found页面配置

- 如果用户随意输入一个地址，该地址无法匹配，那么在路由匹配的位置将什么内容都不显示。
- 很多时候，我们希望在这种情况下，让用户看到一个Not Found的页面。
- 这个过程非常简单：
  - 开发一个Not Found页面；
  - 配置对应的Route，并且设置path为*即可；

```
<Route path='*' element={<NotFound/>}/>
```

路由的嵌套

- **在开发中，路由之间是存在嵌套关系的。**
- **这里我们假设Home页面中有两个页面内容：**
  - 推荐列表和排行榜列表；
  - 点击不同的链接可以跳转到不同的地方，显示不同的内容；
- <Outlet>组件用于在父路由元素中作为子路由的占位元素。


```
<Routes>
  <Route path="/" element={<Navigate to="/home"/>}/>
  <Route path='/home' element={<Home/>}>
    <Route path="/home" element={<Navigate to="/home/recommend"/>}/>
    <Route path="/home/recommend" element={<HomeRecommend/>}/>
    <Route path="/home/ranking" element={<HomeRanking/>}/>
  </Route>
  <Route path='/about' element={<About/>}/>
  <Route path='/profile' element={<Profile/>}/>
  <Route path='/login' element={<Login/>}/>
  <Route/>
</Routes>
```

```
<h1>Home Page</h1>
<Link to="/home/recommend">推荐</Link>
<Link to="/home/ranking">排行榜</Link>
<Outlet/>
```

手动路由的跳转

- **目前我们实现的跳转主要是通过Link或者NavLink进行跳转的，实际上我们也可以通过JavaScript代码进行跳转。**
  - 我们知道Navigate组件是可以进行路由的跳转的，但是依然是组件的方式。
  - 如果我们希望通过JavaScript代码逻辑进行跳转（比如点击了一个button），那么就需要获取到navigate对象。
- **在Router6.x版本之后，代码类的API都迁移到了hooks的写法：**
  - 如果我们希望进行代码跳转，需要通过useNavigate的Hook获取到navigate对象进行操作；
  - 那么如果是一个函数式组件，我们可以直接调用，但是如果是一个类组件呢？

```
import { useNavigate } from "react-router-dom"

export default function withRouter(WrapperComponent) {
  return props => {
    const navigate = useNavigate()

    return <WrapperComponent {...props} router={{navigate}}/>
  }
}
```

```
interface NavigateFunction {
  (
    to: To,
    options?: { replace?: boolean; state?: any }
  ): void;
  (delta: number): void;
}
```

路由参数传递

- **传递参数有二种方式：**
  - 动态路由的方式；
  - search传递参数；
- **动态路由的概念指的是路由中的路径并不会固定：**
  - 比如/detail的path对应一个组件Detail；
  - 如果我们将path在Route匹配时写成/detail/:id，那么 /detail/abc、/detail/123都可以匹配到该Route，并且进行显示；
  - 这个匹配规则，我们就称之为动态路由；
  - 通常情况下，使用动态路由可以为路由传递参数。

```
<Link to="detail/123">详情:123</Link>
<Link to="detail/321">详情:321</Link>
```

- **search传递参数**

```
<Link to="user?name=why&age=18">用户信息</Link>
```

```
const [searchParams] = useSearchParams()
const query = Object.fromEntries(searchParams)
```

路由的配置文件

- 目前我们所有的路由定义都是直接使用Route组件，并且添加属性来完成的。
- 但是这样的方式会让路由变得非常混乱，我们希望将所有的路由配置放到一个地方进行集中管理：
  - 在早期的时候，Router并且没有提供相关的API，我们需要借助于react-router-config完成；
  - 在Router6.x中，为我们提供了useRoutes API可以完成相关的配置；

```
<div>{useRoutes(routes)}</div>
```

- 如果我们对某些组件进行了异步加载（懒加载），那么需要使用Suspense进行包裹：

```
const HomeRecommend = React.lazy(() => import("../pages/HomeRecommend"))
const HomeRanking = React.lazy(() => import("../pages/HomeRanking"))
```

```
<Suspense fallback={<div>Loading</div>}>
  <HashRouter>
    <App />
  </HashRouter>
</Suspense>
```

```
const routes = [
  {
    path: "/",
    element: <Navigate to="/home"/>
  },
  {
    path: "/home",
    element: <Home/>,
    children: [
      {
        path: "/home/recommend",
        element: <HomeRecommend/>
      },
      {
        path: "/home/ranking",
        element: <HomeRanking/>
      },
    ]
  },
  {
    path: "/about",
    element: <About/>
  },
  {
    path: "profile",
    element: <Profile/>
  },
```