react-router的使用

王红元 coderwhy

实力IT教育

路由的由来

- 说起路由你想起了什么？
  - 路由是一个网络工程里面的术语。
  - **路由**（**routing**）就是通过互联的网络把信息从源地址传输到目的地址的活动. --- 维基百科
- 路由器提供了两种机制: 路由和转送.
  - 路由是决定数据包从**来源**到**目的地**的路径.
  - 转送将**输入端**的数据转移到合适的**输出端**.

- 路由中有一个非常重要的概念叫路由表.
  - ✓ 路由表本质上就是一个映射表, 决定了数据包的指向.
- 路由的概念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）；

阶段一：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.
  - 这就完成了一个IO操作.
- 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顿.
  - 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.

阶段二：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，和之前的后断路由不同，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；
  - 目前很多的网站依然采用这种模式开发（jQuery开发模式）；

阶段三：单页面富应用（SPA）

- 单页面富应用的理解：
  - 单页面富应用的英文是single-page application，简称SPA；
  - 整个Web应用只有实际上只有一个页面，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，并且根据URL的不同去渲染新的页面；
- 如何可以应用URL和渲染的页面呢？前端路由
  - 前端路由维护着URL和渲染页面的映射关系；
  - 路由可以根据不同的URL，最终让我们的框架（比如Vue、React、Angular）去渲染不同的组件；
  - 最终我们在页面上看到的实际就是渲染的一个个组件页面；

前端路由的原理

- 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发生变化，同时不引起页面的刷新有两个办法：
  - 通过URL的hash改变URL；
  - 通过HTML5中的history模式修改URL；
- 当监听到URL发生变化时，我们可以通过自己判断当前的URL，决定到底渲染什么样的内容。

URL的hash

- 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；

```
<script>
  // 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";
    }
  })
</script>
```

注意：

- 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l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";
  }
}
```

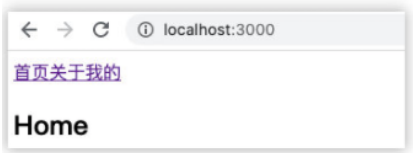
react-router

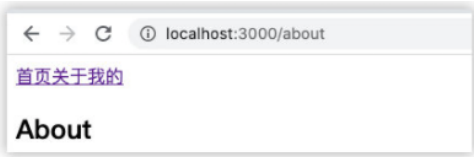
- 目前前端流行的三大框架, 都有自己的路由实现:
  - Angular的ngRouter
  - React的react-router
  - Vue的vue-router
- React Router的版本4开始，路由不再集中在一个包中进行管理了：
  - react-router是router的核心部分代码；
  - react-router-dom是用于浏览器的；
  - react-router-native是用于原生应用的；
- 目前我们使用最新的React Router版本是v5的版本：
  - 实际上v4的版本和v5的版本差异并不大；
- 安装react-router：
  - 安装react-router-dom会自动帮助我们安装react-router的依赖；

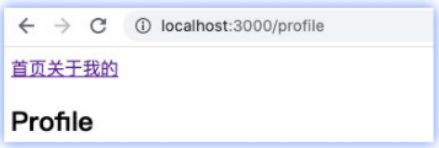
```
yarn add react-router-dom
```

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；
  - 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；
  - Route：
    - ✓ Route用于路径的匹配；
    - ✓ path属性：用于设置匹配到的路径；
    - ✓ component属性：设置匹配到路径后，渲染的组件；
    - ✓ 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；








```jsx
import React, { PureComponent } from 'react';

import { BrowserRouter, Route, Link } from 'react-router-dom';

import Home from './pages/home';
import About from './pages/about';
import Profile from './pages/profile';

export default class App extends PureComponent {
  render() {
    return (
      <BrowserRouter>
        <Link to="/">首页</Link>
        <Link to="/about">关于</Link>
        <Link to="/profile">我的</Link>

        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/profile" component={Profile} />
      </BrowserRouter>
    )
  }
}
```

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：
  - activeStyle：活跃时（匹配时）的样式；
  - activeClassName：活跃时添加的class；
  - exact：是否精准匹配；

```
<NavLink to="/" activeStyle={{color: "red"}}>首页</NavLink>
<NavLink to="/about" activeStyle={{color: "red"}}>关于</NavLink>
<NavLink to="/profile" activeStyle={{color: "red"}}>我的</NavLink>
```

- 但是，我们会发现在选中about或profile时，第一个也会变成红色：
  - 原因是/路径也匹配到了/about或/profile；
  - 这个时候，我们可以在第一个NavLink中添加上exact属性；

```
<NavLink exact to="/" activeClassName="link-active">首页</NavLink>
<NavLink to="/about" activeClassName="link-active">关于</NavLink>
<NavLink to="/profile" activeClassName="link-active">我的</NavLink>
```

- 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

Switch的作用

- 我们来看下面的路由规则：
  - 当我们匹配到某一个路径时，我们会发现有一些问题；
  - 比如/about路径匹配到的同时，/:userid也被匹配到了，并且最后的一个NoMatch组件总是被匹配到；

```
<Route exact path="/" component={Home} />
<Route path="/about" component={About} />
<Route path="/profile" component={Profile} />
<Route path="/:userid" component={User} />
<Route component={NoMatch} />
```

- 原因是什么呢？默认情况下，react-router中只要是路径被匹配到的Route对应的组件都会被渲染；
  - 但是实际开发中，我们往往希望有一种排他的思想：
  - 只要匹配到了第一个，那么后面的就不应该继续匹配了；
  - 这个时候我们可以使用Switch来将所有的Route进行包裹即可；

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/profile" component={Profile} />
  <Route path="/:userid" component={User} />
  <Route component={NoMatch} />
</Switch>
```

Redirect

- Redirect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User界面；
  - 但是在User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；

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about页面中有两个页面内容：
  - 商品列表和消息列表；
  - 点击不同的链接可以跳转到不同的地方，显示不同的内容；

首页关于我的用户
商品消息

- 商品列表1
- 商品列表2
- 商品列表3

首页关于我的用户
商品消息

- 消息列表1
- 消息列表2
- 消息列表3

手动路由跳转

- 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。
- 但是通过JavaScript代码进行跳转有一个前提：**必须获取到history对象。**
- 如何可以获取到history的对象呢？两种方式
  - 方式一：如果该组件是通过路由直接跳转过来的，那么可以直接获取history、location、match对象；
  - 方式二：如果该组件是一个普通渲染的组件，那么不可以直接获取history、location、match对象；
- 那么如果普通的组件也希望获取对应的对象属性应该怎么做呢？
  - 前面我们学习过高阶组件，可以在组件中添加想要的属性；
  - react-router也是通过高阶组件为我们的组件添加相关的属性的；
- 如果我们希望在App组件中获取到history对象，必须满足一下两个条件：
  - App组件必须包裹在Router组件之内；
  - App组件使用withRouter高阶组件包裹；

- **源码选读：这里的history来自哪里呢？是否和之前使用的window.history一样呢？**

参数传递

- **传递参数有三种方式：**
  - 动态路由的方式；
  - search传递参数；
  - Link中to传入对象；
- **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。

```
<NavLink to="/detail/abc123">详情</NavLink>
```

- **search传递参数**

```
<NavLink to="/detail2?name=why&age=18">详情2</NavLink>
```

- **Link中to可以直接传入一个对象**

```
<NavLink to={{
        pathname: "/detail2",
        query: {name: "kobe", age: 30},
        state: {height: 1.98, address: "洛杉矶"},
        search: "?apikey=123"
      }}>
  详情2
</NavLink>
```

react-router-config

- 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：
  - 这个时候可以使用react-router-config来完成；
- 安装react-router-config

```
yarn add react-router-config
```

- 配置路由映射的关系数组

```
const routes = [
  {
    path: "/",
    exact: true,
    component: Home
  },
  {
    path: "/about",
    component: About,
    routes: [
      {
```

- 使用renderRoutes函数完成配置

```
{renderRoutes(routes)}
```