React Router v4

## 前端路由和后端路由

**后端路由**

多页应用中，一个URL对应一个HTML页面，一个Web应用包含很多HTML页面，在多页应用中，页面路由控制由服务器端负责，这种路由方式称为后端路由。

多页应用中,每次页面切换都需要向服务器发送一次请求，页面使用到的静态资源也需要重新加载，存在一定的浪费。而且，页面的整体刷新对用户体验也有影响，因为不同页面间往往存在共同的部分，例如导航栏、侧边栏等，页面整体刷新也会导致共用部分的刷新。

**前端路由**

在单面应用(SPA)中，URL发生并不会向服务器发送新的请求，所以“逻辑页面”的路由只能由前端负责，这种路由方式称为前端路由。

目前，国内的搜索引擎大多对单页应用的SEO支持的不好，因此，对于 SEO 非常看重的 Web

应用(例如，企业官方网站，电商网站等)，一般还是会选择采用多页面应用。React 也并非只能用于开发单页面应用。

# 安装

React Router 现在已经被划分成了三个包：

react-router，react-router-dom，react-router-native。

你不应该直接安装 react-router，这个包为 React Router 应用提供了核心的路由组件和函数，另外两个包提供了特定环境的组件（浏览器和 react-native 对应的平台），不过他们也是将 react-router 导出的模块再次导出。

你应该选择这两个中适应你开发环境的包，我们需要构建一个网站（在浏览器中运行），所以我们要安装 react-router-dom

路由器

React Router 通过 Router 和 Route 两个组件完成路由功能。

Router 可以理解成路由器，一个应用中需要一个 Router 实例，所有配置组件 Route 都定义为 Router 的子组件。

# Router

当开始一个新项目时，你应该决定要使用哪种 router。对于在浏览器中运行的项目，我们可以选择 <BrowserRouter> 和 <HashRouter> 组件，<BrowserRouter> 应该用在服务器处理动态请求的项目中（知道如何处理任意的URI），<HashRouter> 用来处理静态页面（只能响应请求已知文件的请求）。

通常来说更推荐使用 <BrowserRouter>，可是如果服务器只处理静态页面的请求，那么使用 <HashRouter> 也是一个足够的解决方案。

对于我们的项目，我们假设所有的页面都是由服务器动态生成的，所以我们的 router 组件选择 <BrowserRouter>。

**<BrowserRouter>**

A <Router> that uses the HTML5 history API (pushState, replaceState and the popstate event) to keep your UI in sync with the URL.

BrowserRouter使用 HTML5 的 history API（pushState、replaceState等）实现应用的 UI 和 URL 的同步。

#### **BrowserRouter 创建的 URL 形式如下：**

http://example.com/some/path

**<HashRouter>**

A <Router> that uses the hash portion of the URL (i.e. window.location.hash) to keep your UI in sync with the URL.

HashRouter 使用 URL 的 hash 实现应用的 UI 和 URL 同步。

#### **HashRouter 创建的 URL 形式如下：**

http://example.com/*#/some/path*

使用 BrowserRouter 时，一般还需要对服务器进行配置，让服务器能正确地处理所有可能的URL。例如，当浏览器发生 <http://example.com/some/path> 和 <http://example.com/some/path2> 两个请求时，服务器需要能返回正确的 HTML 页面（也就是单页面应用中唯一的 HTML 页面）

HashRouter 则不存在这个问题，因为 hash 部分的内容会被服务器自动忽略，真正有效的信息是 hash 前端的部分，而对于单页应用来说，这部分是固定的。

Router 的组件只能接受一个子元素，为了遵照这种限制，创建一个 <App> 组件来渲染其他的应用将非常方便（将应用从 router 中分离对服务器端渲染也有重要意义，因为我们在服务器端转换到 <MemoryRouter> 时可以很快复用 <App>）

# History

Router 会创建一个 history 对象，history 用来跟踪 URL, 当URL 发生变化时， Router,的后代组件会重新渲染。React Router 中提供的其他组件可以通过 context 获取 history 对象，这也隐含说明了 React Router 中其他组件必须作为 Router 组件后代使用。

React Router 提供的其他组件依赖在 context 上储存的 history 对象，所以他们必须在 router 对象的内部渲染。

**一个没有 Router 祖先元素的 React Router 对象将无法正常工作。**

# Route

<Route> 组件是 React Router 的主要组成部分，是React Router中用于配置路由信息的组件，如果你想要在路径符合的时候在任何地方渲染什么东西，你就应该创造一个 <Route> 元素。

### **Path**

一个 <Route> 组件需要一个 string 类型的 path prop 来指定路由需要匹配的路径。举例来说，<Route path='/category/'> 将匹配以 /category 开始的路径，当当前的路径和 path 匹配时，route 将会渲染对应的 React 元素。当路径不匹配的时候 ，路由不会渲染任何元素。

example: <https://codesandbox.io/s/react-router-basic-bnpsd>

#### **match**

React Router使用 path-to-regexp 包来判断路径的 path prop 是否匹配当前路径，它将 path 字符串转换成正则表达式与当前的路径进行匹配，关于 path 字符串更多的可选格式，可以查阅 path-to-regexp [文档](https://github.com/pillarjs/path-to-regexp)。

当路由与路径匹配的时候，一个具有以下属性的 **match** 对象将会被作为 prop 传入

* url // 当前路径与路由相匹配的部分
* path // 路由的path
* isExact // path === pathname
* params // 一个包含着 pathname 被 path-to-regexp 捕获的对象 (例如/user/:id，则params是{ id: '123' })

example: <https://codesandbox.io/s/react-router-url-parameters-rrbsb>

### **创建路由**

<Route> 可以在router中的任意位置被创建，不过一般来说将他们放到同一个地方渲染更加合理，你可以使用 <Switch> 组件来组合 <Route>，<Switch>将遍历它的 children 元素（路由），然后只匹配第一个符合的 pathname。

对于我们的网站来说，拿CRTV来举例子，不考虑数据的情况下，我们想要匹配的路径为：

/ - 主页 (HOME PAGE)

/shows/:seoName (SHOWS PAGE)

/show/:seoName (SHOW PAGE)

/season/:seoName (SEASON PAGE)

/video/:seoName (VOD PAGE)

/live-events (Live Event Page)

/event/:seoName (EVENT PAGE)

所以为了匹配路径，我们需要创建带 path prop的 <Route> 元素

<Switch>

<Route exact path='/' component={Home}/>

<Route path='/shows/:seoName component={SHOWS}/>

…...

</Switch>

### <Route> **将会渲染什么**

Routes 可以接受三种 prop 来决定路径匹配时渲染的元素，只能给 <Route> 元素提供一种来定义要渲染的内容。

1. <component> - 一个 React 组件，当一个带有 component prop 的路由匹配的时候，路由将会返回 prop 提供的 component 类型的组件（通过 React.createElement 渲染）。

*<Route path='/page' component={Page} />*

1. render - 一个返回 React 元素的方法，与 component 类似，也是当路径匹配的时候会被调用。写成内联形式渲染和传递参数的时候非常方便。

*const extraProps = { isStaticPage: true }*

*<Route path='/page' render={(props) => (*

*<Page {...props} data={extraProps}/>*

*)}/>*

1. children - 一个返回 React 元素的方法。与前两种不同的是，这种方法**总是**会被渲染，无论路由与当前的路径是否匹配。

<Route path='/page' children={(props) => (

props.match ? <Page {...props}/> : <EmptyPage {...props}/>

)}/>

一般来说，我们一般使用 component 或者 render，children 的使用场景不多，而且一般来说当路由不匹配的时候最好不要渲染任何东西。在我们的例子中，不需要向路由传递任何参数，所有我们使用 <component>。

由 <Route> 渲染的元素将会带有一系列的 props，有 match 对象，当前的 location 对象，还有 history 对象

#### **exact**

如果想让 URL 和 Route 完全匹配时，Route才渲染，那么可以使用 Route 的 exact 属性。Switch 和 exact 常常联合使用，用于应用首页的导航。例如：

*<Router>*

*<Switch>*

*<Route exact path='/' component={Home}/>*

*<Route exact path='/posts' component={Posts} />*

*<Route exact path='/:user' component={User} />*

*…….*

*</Switch>*

*</Router>*

如果不使用 Switch,当 URL 的 pathname 为 "/posts" 时，<Route path='/posts' /> 和 <Route path=':user' /> 都会被匹配，但显然我们并不希望 <Route path=':user' /> 被匹配，实际上也没有用户名为 posts 的用户。如果不使用 exact， "/" "/posts" "/user1"等几乎所有 URL 都会匹配第一个 Route,又因为Switch 的存在，后面的两个 Route永远不会被匹配。使用 exact,保证 只有当 URL 的 pathname 为 '/'时，第一个Route才会匹配。

### **Link**

我们的网站需要在页面之间导航，如果我们使用 <a> 标签导航的话，将会载入一整个新的页面。React Router 提供了一个 <Link> 组件来避免这种情况，当点击 <Link> 时，URL 将会更新，页面也会在不载入整个新页面的情况下渲染内容。

<Link> 使用 to prop 来决定导航的目标，可以是一个字符串，或者是一个 location 对象（包含 pathname, search, hash 和 state 属性）。当只是一个字符串的时候，将会被转化为一个 location 对象

to 可以是 string 或 object 类型，当 to 为 object 类型时，可以包含 pathname、search、hash、state 四个属性，例如:

*<Link*

*to={{*

***pathname****: '/posts',*

***search****: '?sort=name',*

***hash****:'#the-hash',*

***state****: { fromHome: true}*

*}}>*

</Link>

除了使用Link外，我们还可以使用 history 对象手动实现导航。

history 中最常用的两个方法是 push(path,[state]) 和 replace(path,[state]),push会向浏览器记录中新增一条记录，replace 会用新记录替换记录。例如：

***history.push('/posts');***

***history.replace('/posts');***

### **NavLink**

<NavLink>是<Link>的特殊版本，当它与当前URL匹配时，它将为呈现的元素添加样式属性。导航链接应该使用 NavLink 而不再是 Link

1. activeClassName: string (选中时的className，默认是active)

<NavLink to="/faq" activeClassName="selected">

FAQs

</NavLink>

1. activeStyle: object (选中时的样式)

<NavLink

to="/faq"

activeStyle={{

fontWeight: "bold",

color: "red"

}}

>

FAQs

</NavLink>

1. exact: bool (完全匹配才选中)

<NavLink exact to="/profile">

Profile

</NavLink>

# 新的 API 和新的思维模式

React Router 的早期版本将路由规则集中在一个位置，使它们与布局组件分离。当然，路由可以被划分成多个文件，但从概念上讲，路由是一个单元，基本上是一个美化的配置文件。

V3:

const PrimaryLayout = props => (

<div className="primary-layout">

<header>

Our React Router 3 App Header

</header>

<main>

{props.children}

</main>

<footer>

Our React Router 3 App Header

</footer>

</div>

)

<Router history={browserHistory}>

<Route path="/" component={PrimaryLayout}>

<IndexRoute component={HomePage} />

<Route path="/users" component={UsersPage} />

</Route>

</Router>

以下是 v3 中的一些核心思想，但在 v4 中是不正确的:

* 路由集中在一个地方。
* 布局和页面嵌套是通过 <Route> 组件的嵌套而来的。
* 布局和页面组件是完全纯粹的，它们是路由的一部分。

V4:

const PrimaryLayout = () => (

<div className="primary-layout">

<header>

Our React Router 4 App

</header>

<main>

<Route path="/" exact component={HomePage} />

<Route path="/users" component={UsersPage} />

</main>

</div>

)

const App = () => (

<BrowserRouter>

<PrimaryLayout />

</BrowserRouter>

)

**React Router 4并不需要在一个地方集中声明应用需要的所有 Route, Route实际上也是一个普通的 React 组件，可以在任意地方使用它（前提是，Route必须是 Router 的子节点）。当然，这样的灵活性也一定程度上增加了组织 Route 结构层次的难度。**

**新的 API 概念：**

由于我们的应用程序是用于浏览器的，所以我们需要将它封装在来自 v4 的 BrowserRouter 中。还要注意的是我们现在从 react-router-dom 中导入它（这意味着我们安装的是 react-router-dom 而不是 react-router）。

对于使用 React Router v4 构建的应用程序，首先看到的是“路由”似乎丢失了。在 v3 中，路由是我们的应用程序直接呈现给 DOM 的最巨大的东西。 现在，除了 <BrowserRouter> 外，我们首先抛给 DOM 的是我们的应用程序本身。

另一个在 v3 的例子中有而在 v4 中没有的是，使用 {props.children} 来嵌套组件。这是因为在 v4 中，<Route> 组件在何处编写，如果路由匹配，子组件将在那里渲染。

### **包容性路由**

V4 的路由默认为“包含”的，这意味着多个 <Route> 可以同时进行匹配和渲染。

### **排他性路由**

如果你只需要在路由列表里匹配一个路由，则使用 <Switch> 来启用排他路由：

<Switch>

<Route exact path='/' component={Home}/>

<Route path='/shows/:seoName component={SHOWS}/>

…...

</Switch>

### **授权路由**

在应用程序中，通常会根据用户的登录状态来限制用户访问某些路由。对于未经授权的页面（如“登录”和“忘记密码”）与已授权的页面（应用程序的主要部分）看起来不一样也是常见的。

例子:

<https://codesandbox.io/s/react-router-redirects-auth-668he>

**嵌套路由**

由于路由是常规的React组件，因此可以在应用程序中的任何位置（包括子元素中）呈现它们。

例子：

[https://codesandbox.io/s/react-router-nesting-qzy12](https://codesandbox.io/s/react-router-nesting-qzy12?from-embed)

**递归路由**

有时，您可能不知道应用程序的所有可能路由。 例如，在构建文件系统浏览UI或基于数据动态确定URL时。 在这些情况下，拥有能够在运行时根据需要生成路由的动态路由器会有所帮助。

[https://codesandbox.io/s/react-router-recursive-paths-07d4i](https://codesandbox.io/s/react-router-recursive-paths-07d4i?from-embed)

## **代码分片**

默认情况下，当在项目根路径下执行 npm run build 时 ,create-react-app内部使用 webpack将 src路径下的所有代码打包成一个 JS 文件和一个 Css 文件。

当项目代码量不多时，把所有代码打包到一个文件的做法并不会有什么影响。但是，对于一个大型应用，如果还把所有的代码都打包到一个文件中，显然就不合适了。

create-react-app 支持通过动态 import() 的方式实现代码分片。

import()接收一个模块的路径作为参数，然后返回一个 Promise 对象， Promise 对象的值就是待导入的模块对象。例如

*// moduleA.js*

*const moduleA = 'Hello'*

*export { moduleA };*

*// App.js*

*import React, { Component } from 'react';*

*class App extends Component {*

*handleClick = () => {*

*// 使用import 动态导入 moduleA.js*

***import('./moduleA')***

*.then(({moduleA}) => {*

*// 使用moduleA*

*})*

*.catch(err=> {*

*//处理错误*

*})*

*};*

*render() {*

*return(*

*<div>*

*<button onClick={this.handleClick}>加载 moduleA</button>*

*</div>*

*)*

*}*

*}*

*export default App;*

上面代码会将 **moduleA.js** 和它所有依赖的其他模块单独打包到一个chunk文件中，只有当用户点击加载按钮，才开始加载这个 chunk 文件。

当项目中使用 React Router 是，一般会根据路由信息将项目代码分片，每个路由依赖的代码单独打包成一个chunk文件。我们创建一个函数统一处理这个逻辑：

*import React, { Component } from 'react';*

*// importComponent 是使用 import()的函数*

*export default function asyncComponent(importComponent) {*

*class AsyncComponent extends Component {*

*constructor(props) {*

*super(props);*

***this.state = {***

***component: null //动态加载的组件***

***}***

*}*

*componentDidMount() {*

*importComponent().then((mod) => {*

***this.setState({***

***// 同时兼容 ES6 和 CommonJS 的模块***

***component: mod.default ? mod.default : mod;***

***});***

*})*

*}*

*render() {*

*// 渲染动态加载组件*

*const C = this.state.component;*

*return C ? <C {...this.props}></C> : null*

*}*

*}*

*return AsyncComponent;*

*}*

asyncComponent接收一个函数参数 importComponent, importComponent 内通过import()语法动态导入模块。在AsyncComponent被挂载后，importComponent就会调用，进而触发动态导入模块的动作。

下面利用 asyncComponent 对上面的例子进行改造，代码如下:

*import React, { Component } from 'react';*

*import { ReactDOM, BrowserRouter as Router, Switch, Route } from 'react-dom';*

*import asyncComponent from './asyncComponent'*

*//通过asyncComponent 导入组件，创建代码分片点*

*const AsyncHome = asyncComponent(() => import("./components/Home"))*

*const AsyncLogin = asyncComponent(() => import("./components/Login"))*

*class App extends component {*

*render() {*

*return(*

*<Router>*

*<Switch>*

*<Route exact path="/" component={AsyncHome}></Route>*

*<Route exact path="/login" component={AsyncLogin}></Route>*

*<Route exact path="/posts" component={AsyncHome}></Route>*

*</Switch>*

*</Router>*

*)*

*}*

*}*

*export default App;*

这样，只有当路由匹配时，对应的组件才会被导入，实现按需加载的效果。

## 代码实例

**Custom Link**

我们可以根据需要定义定制化Link，来实现Link中的公共逻辑

例子:

<https://codesandbox.io/s/react-router-custom-link-sxec1>

**Preventing Transitions**

当我们页面上有form表单时，我们可以用react-router里的Prompt组件，该组件主要作用是,在用户准备离开该页面时, 弹出提示, 返回true或者false, 如果为true, 则离开页面, 如果为false, 则停留在该页面

例子:

<https://codesandbox.io/s/react-router-preventing-transitions-1yvzb?from-embed>

**Not match**

您可以将<Switch>中的最后一个<Route>用作一种“后备”路由，以捕获404错误。

Note:

<Switch>呈现与之匹配的第一个子元素<Route>

<Redirect>可用于将旧的URL重定向到新的URL

<Route path =”\*”>始终匹配

例子:

<https://codesandbox.io/s/react-router-no-match-404-es806>

**Route Config**

有时候我们可以使用集中式路由配置来管理路由，把路由配置作为data，讲路径映射到对应的components

例子:

<https://codesandbox.io/s/react-router-route-config-5stco>

**Modal Gallery**

Gallery Modal可以理解为相册的预览界面。可以显示网络图片，也可以显示base64Image。这里是同一url的预览界面和正常界面

本示例说明如何在同一URL上呈现两个不同的屏幕（或在不同上下文中的同一屏幕）

例子:

<https://codesandbox.io/s/react-router-modal-gallery-1coo0>

**StaticRouter Context**

当一个 <Route> 匹配时，它将把 context 对象传递给呈现为 staticContext 的组件。

渲染之后，可以使用这些属性来配置服务器的响应。

Note: 一般用于ReactDOMServer 服务器端渲染

例子:

<https://codesandbox.io/s/react-router-staticrouter-context-8der0>

扩展：

<https://nextjs.frontendx.cn/docs/#%E6%8A%80%E5%B7%A7>

**Query Parameters**

如果您使用简单的key = value查询字符串，并且不需要支持IE 11，则可以使用浏览器的内置URLSearchParams API

例子:

<https://codesandbox.io/s/react-router-query-parameters-mfh8p>

## 可用Hooks（16.8+）

**useHistory**

useHistory hook可以访问可用于导航的history实例**。**

**useLocation**

useLocation挂钩返回代表当前URL的位置对象。 您可以将其想像为useState，它会在URL发生更改时返回一个 location 对象

**useParams**

useParams返回URL参数的 key/value 对象, 即当前<Route>的match.params

**useRouteMatch**

useRouteMatch尝试在无需实际呈现<Route>的情况下以与<Route>相同的方式匹配当前URL

## withRouter

**定义:**

您可以通过withRouter高阶组件访问历史对象的属性和最接近的<Route>匹配项。 每当呈现时，withRouter都会将更新的匹配，位置和历史道具传递给包装的组件。

**作用:**

高阶组件中的withRouter, 作用是将一个组件包裹进Route里面, 然后react-router的三个对象history, location, match就会被放进这个组件的props属性中. 所以withRouter的作用就是, 如果我们某个东西不是一个Router, 但是我们要依靠它去跳转一个页面, 比如点击页面的logo, 返回首页, 这时候就可以使用withRouter来做。

**使用方法:**

1. 引入 import {withRouter} from "react-router-dom";
2. 讲需要封装React compontent包裹一下 export default withRouter(App)

**注意:**

withRouter不像React Redux的connect那样观察location更改时进行state更改。 而是在location更改后从<Router>组件传播出去后重新渲染, 这意味着withRouter不会在路由转换时re-render，除非其父组件re-render。

**适用场景:**

避免更新受阻

因为react-redux的connect高阶组件会为传入的参数组件实现shouldComponentUpdate 这个钩子函数，导致只有prop发生变化时才触发更新相关的生命周期函数(含render)而很显然，我们的location对象并没有作为prop传入该参数组件, 因此使用withRouter可以避免更新受阻

官方文档

<https://reacttraining.com/react-router/web/guides/quick-start>