

React router精讲

1、课程目标

2、课程大纲

3、主要内容

3.1、react-router基本使用

(1) 路由配置

(2) 页面跳转

3.2、搭建基于react-router的应用

3.3、核心源码解析

3.4、react-router和vue-router的差异

(1) 路由类型

(2) 使用方式

(3) 实现差异

1、课程目标

P6:

- 掌握react-router使用方法；
- 能使用react-router设计开发react应用；
-

P6 + - P7:

- 理解react-router关键源码实现。
- 理解react-router和vue-router的实现差异，针对面试提出的问题能举一反三；

2、课程大纲

- react-router使用详解；
- 从0到1搭建一个基于react-router的应用；
- react-router关键源码解析；
- 对比react-router和vue-router的差异；

3、主要内容

3.1、react-router基本使用

(1) 路由配置

jsx用法:

JavaScript | 复制代码

```
1  import { render } from "react-dom";
2  import {
3    BrowserRouter,
4    Routes,
5    Route,
6  } from "react-router-dom";
7  // import your route components too
8
9  render(
10    <BrowserRouter>
11      <Routes>
12        <Route path="/" element={<App />}>
13          <Route index element={<Home />} />
14          <Route path="teams" element={<Teams />}>
15            <Route path=":teamId" element={<Team />} />
16            <Route path="new" element={<NewTeamForm />} />
17            <Route index element={<LeagueStandings />} />
18          </Route>
19        </Route>
20      </Routes>
21    </BrowserRouter>,
22    document.getElementById("root")
23  );
```

config+hooks用法:

```
1  import { render } from "react-dom";
2  import { useRoutes, BrowserRouter } from 'react-router-dom'
3
4  function Routes() {
5    return useRoutes([
6      {
7        path: '/',
8        element: <App>,
9        children: [
10       {
11         path: '/',
12         element: <Home />,
13       },
14       {
15         path: '/teams',
16         element: <Teams />
17         children: [
18           {
19             path: ':tamId',
20             element: <Team />
21           }
22         ]
23       }
24     ],
25   ])
26 }
27
28 render(
29   <BrowserRouter>
30     <Routes />
31   </BowserRouter>,
32   document.getElementById("root")
33 )
34
```

(2) 页面跳转

jsx用法:

```
1  <Link to="/">To</Link>
```

hooks用法：

TypeScript | 复制代码

```
1  const location = useLocation()
2
3  location.push("/")
```

以上是react-hooks的最基本用法，下面我们通过代码演示，来详细了解react-router的使用：

3.2、搭建基于react-router的应用

代码演示：

- 基于cra的基本react-router搭建
- layout和懒加载
- 权限验证

3.3、核心源码解析

我们经常使用的BrowserRouter和HashRouter主要依赖三个包：react-router-dom、react-router、history。

- **react-router** 提供react路由的核心实现，是跨平台的。
- **react-router-dom** 提供了路由在web端的具体实现，与之同级的还有react-router-native，提供react-native端的路由实现。
- **history**是一个对浏览器history操作封装，抹平浏览器history和hash的操作差异，提供统一的location对象给react-router-dom使用。

下面从最简单的例子，进入react-router源码解析：

JavaScript | 复制代码

```
1  ReactDOM.render(
2    <BrowserRouter>
3      <App />
4    </BrowserRouter>,
5    document.getElementById('root')
6  )
```

BrowserRouter实现：

```
1 export function BrowserRouter({
2   basename,
3   children,
4   window,
5 }: BrowserRouterProps) {
6   let historyRef = React.useRef<BrowserHistory>();
7   if (historyRef.current == null) {
8     // 如果之前没有创建过，则创建history对象，createBrowserHistory是history包中
    实现的方法
9     historyRef.current = createBrowserHistory({ window });
10  }
11
12  let history = historyRef.current;
13  let [state, setState] = React.useState({
14    action: history.action,
15    location: history.location,
16  });
17  // history变化时重新渲染
18  React.useLayoutEffect(() => history.listen(setState), [history]);
19
20  return (
21    <Router
22      basename={basename}
23      children={children}
24      location={state.location}
25      navigationType={state.action}
26      navigator={history}
27    />
28  );
29 }
```

BrowserRouter包装了<Router />，并创建了history对象，监听history变化。

Router实现：

```
1 export function Router({
2   basename: basenameProp = "/",
3   children = null,
4   location: locationProp,
5   navigationType = NavigationType.Pop,
6   navigator,
7   static: staticProp = false,
8 } : RouterProps): React.ReactElement | null {
9
10   let basename = normalizePathname(basenameProp);
11   let navigationContext = React.useMemo(
12     () => ({ basename, navigator, static: staticProp }),
13     [basename, navigator, staticProp]
14   );
15
16   if (typeof locationProp === "string") {
17     locationProp = parsePath(locationProp);
18   }
19
20   let {
21     pathname = "/",
22     search = "",
23     hash = "",
24     state = null,
25     key = "default",
26   } = locationProp;
27
28   let location = React.useMemo(() => {
29     let trailingPathname = stripBasename(pathname, basename);
30
31     if (trailingPathname == null) {
32       return null;
33     }
34
35     return {
36       pathname: trailingPathname,
37       search,
38       hash,
39       state,
40       key,
41     };
42     // 当location中有如下变化时重新生成location对象，触发页面重新渲染
43   }, [basename, pathname, search, hash, state, key]);
44
45
```

```

46 ▾    if (location == null) {
47        return null;
48    }
49    // const LocationContext = React.createContext<LocationContextObject>
    (null!);
50    /**
51     创建了全局的context，用于存放history对象
52    */
53    return (
54        <NavigationContext.Provider value={navigationContext}>
55            <LocationContext.Provider
56                children={children}
57                value={{ location, navigationType }}
58            />
59        </NavigationContext.Provider>
60    );
61 }

```

Router处理了history对象，将其用NavigationContext包裹，使得下层子组件都可以访问到这个history。

Routes实现：

```

1 export function Routes({
2   children,
3   location,
4 } : RoutesProps): React.ReactElement | null {
5   return useRoutes(createRoutesFromChildren(children), location);
6 }
7
8 export function useRoutes(
9   routes: RouteObject[],
10  locationArg?: Partial<Location> | string
11 ): React.ReactElement | null {
12   let { matches: parentMatches } = React.useContext(RouteContext);
13   let routeMatch = parentMatches[parentMatches.length - 1];
14   let parentParams = routeMatch ? routeMatch.params : {};
15   let parentPathname = routeMatch ? routeMatch.pathname : "/";
16   let parentPathnameBase = routeMatch ? routeMatch.pathnameBase : "/";
17   let parentRoute = routeMatch && routeMatch.route;
18   let locationFromContext = useLocation();
19   let location;
20   if (locationArg) {
21     let parsedLocationArg =
22       typeof locationArg === "string" ? parsePath(locationArg) :
23       locationArg;
24     location = parsedLocationArg;
25   } else {
26     location = locationFromContext;
27   }
28   let pathname = location.pathname || "/";
29   let remainingPathname =
30     parentPathnameBase === "/"
31       ? pathname
32       : pathname.slice(parentPathnameBase.length) || "/";
33   let matches = matchRoutes(routes, { pathname: remainingPathname });
34
35   return _renderMatches(
36     matches &&
37     matches.map((match) =>
38       Object.assign({}, match, {
39         params: Object.assign({}, parentParams, match.params),
40         pathname: joinPaths([parentPathnameBase, match.pathname]),
41         pathnameBase:
42           match.pathnameBase === "/"
43             ? parentPathnameBase
44             : joinPaths([parentPathnameBase, match.pathnameBase]),

```



```

45         })
46     ),
47     parentMatches
48 );
49 }
50
51 export function createRoutesFromChildren(
52     children: React.ReactNode
53 ): RouteObject[] {
54     let routes: RouteObject[] = [];
55
56     React.Children.forEach(children, (element) => {
57         if (!React.isValidElement(element)) {
58             return;
59         }
60
61         if (element.type === React.Fragment) {
62             routes.push.apply(
63                 routes,
64                 createRoutesFromChildren(element.props.children)
65             );
66             return;
67         }
68         let route: RouteObject = {
69             caseSensitive: element.props.caseSensitive,
70             element: element.props.element,
71             index: element.props.index,
72             path: element.props.path,
73         };
74
75         if (element.props.children) {
76             route.children = createRoutesFromChildren(element.props.children);
77         }
78
79         routes.push(route);
80     });
81
82     return routes;
83 }
84

```

Routes通过Route子组件生成路由列表，通过location中的pathname匹配组件并渲染。

通过以上代码，我们基本理解了react-router如何感知history中的pathname变化，并渲染对应组件。但我们具体是如何操作history变化的呢？

我们在回到最上面的createBrowserHistory 和 history.listen 方法，看看history对象是怎么被创建和改变的：

```
1  export function createBrowserHistory(  
2    options: BrowserHistoryOptions = {}  
3  ): BrowserHistory {  
4    let { window = document.defaultView! } = options;  
5    let globalHistory = window.history;  
6  
7    // 获取当前history中的index和location对象  
8    function getIndexAndLocation(): [number, Location] { }  
9  
10   let blockedPopTx: Transition | null = null;  
11  
12   // 处理返回上一页  
13   function handlePop() {}  
14  
15   // 监听浏览器的popState事件  
16   window.addEventListener(PopStateEventType, handlePop);  
17  
18  
19   // 操作history对象  
20   function applyTx(nextAction: Action) {  
21     action = nextAction;  
22     [index, location] = getIndexAndLocation();  
23     listeners.call({ action, location });  
24   }  
25   // push操作  
26   function push(to: To, state?: any) {}  
27  
28   // replace操作  
29   function replace(to: To, state?: any) {}  
30  
31   // 前进或后退n页  
32   function go(delta: number) {}  
33  
34   let history: BrowserHistory = {  
35     get action() {  
36       return action;  
37     },  
38     get location() {  
39       return location;  
40     },  
41     createHref,  
42     push,  
43     replace,  
44     go,  
45     back() {
```

```

46         go(-1);
47     },
48     forward() {
49         go(1);
50     },
51     listen(listener) {
52         return listeners.push(listener);
53     },
54     block(blocker) {
55         let unblock = blockers.push(blocker);
56
57         if (blockers.length === 1) {
58             window.addEventListener(BeforeUnloadEventType,
promptBeforeUnload);
59         }
60
61         return function () {
62             unblock();
63             if (!blockers.length) {
64                 window.removeEventListener(BeforeUnloadEventType,
promptBeforeUnload);
65             }
66         };
67     },
68 };
69
70 return history;
71 }

```

createBrowserHistory创建了一个标准的history对象，以及对history对象操作的各方法，且操作变更后，通过listen方法将变更结果回调给外部。

getIndexAndLocation实现：

```
1 ▾ function getIndexAndLocation(): [number, Location] {  
2     let { pathname, search, hash } = window.location;  
3     let state = globalHistory.state || {};  
4 ▾     return [  
5         state.idx,  
6 ▾         readOnly<Location>({  
7             pathname,  
8             search,  
9             hash,  
10            state: state.usr || null,  
11            key: state.key || "default",  
12        } ),  
13     ];  
14 }
```

push实现:

```

1
2 ▾ function push(to: To, state?: any) {
3     let nextAction = Action.Push;
4     let nextLocation = getNextLocation(to, state);
5 ▾     function retry() {
6         push(to, state);
7     }
8
9 ▾     if (allowTx(nextAction, nextLocation, retry)) {
10         let [historyState, url] = getHistoryStateAndUrl(nextLocation, index
+ 1);
11
12 ▾         try {
13             // 操作浏览器的history
14             globalHistory.pushState(historyState, "", url);
15 ▾         } catch (error) {
16             window.location.assign(url);
17         }
18         // 处理history对象并回调
19         applyTx(nextAction);
20         /**
21         function applyTx(nextAction: Action) {
22             action = nextAction;
23             [index, location] = getIndexAndLocation();
24             listeners.call({ action, location });
25         }
26         */
27     }
28 }

```

问：我们在代码中都是 `const location = useLocation(); location.push("/")` 这样的方式使用push的，那上面这个push方法到底是怎么跟useLocation关联的呢？

还记得Router中有这么一段代码吗？

```

1 const LocationContext = React.createContext<LocationContextObject>
  (null!);

```

我们的history对象创建后会被Router注入进一个LocationContext的全局上下文中。

useLocation实际就是包裹了这个上下文对象。

JavaScript | 复制代码

```
1 export function useLocation(): Location {  
2   return React.useContext(LocationContext).location;  
3 }
```

总结一下，BrowserRouter核心实现包含三部分：

- 创建history对象，提供对浏览器history对象的操作。
- 创建Router组件，将创建好的history对象注入全局上下文。
- Routes组件，遍历子组件生成路由表，根据当前全局上下文history对象中的pathname匹配当前激活的组件并渲染。

HashRouter和BrowserRouter原理类似，只是监听的浏览器原生history从pathname变为hash，这里不再赘述。

3.4、react-router和vue-router的差异

（1）路由类型

React：

- browserRouter
- hashRouter
- memoryRouter

Vue：

- history
- hash
- abstract

memoryRouter和abstract作用类似，都是在不支持浏览器的环境中充当fallback。

（2）使用方式

路由拦截的实现不同

vue router 提供了 全局守卫、路由守卫、组件守卫 供我们实现 路由拦截。

react router 没有提供类似 vue router 的 守卫 供我们使用，不过我们可以在组件渲染过程中自己实现路由拦截。如果是 类组件，我们可以在 componentWillMount 或者 getDerivedStateFromProps

中通过 `props.history` 实现 路由拦截；如果是 函数式组件，在函数方法中通过 `props.history` 或者 `useHistory` 返回的 `history` 对象 实现 路由拦截。

(3) 实现差异

hash 模式的实现不同

react router 的 hash 模式，是基于 `window.location.hash(window.location.replace)` 和 `hashchange` 实现的。当通过 `push` 方式 跳转页面时，直接修改 `window.location.hash`，然后渲染页面；当通过 `replace` 方式 跳转页面时，会先构建一个 修改 hash 以后的临时 url，然后使用这个临时 url 通过 `window.location.replace` 的方式 替换当前 url，然后渲染页面；当 激活历史记录导致 hash 发生变化时，触发 `hashchange` 事件，重新 渲染页面。

vue router 的 hash 模式，是先通过 `pushState(replaceState)` 在浏览器中 新增(修改)历史记录，然后渲染页面。当 激活某个历史记录 时，触发 `popstate` 事件，重新 渲染页面。如果 浏览器不支持 `pushState`，才会使用 `window.location.hash(window.location.replace)` 和 `hashchange` 实现。

history 模式不支持 `pushState` 的处理方式不同

使用 react router 时，如果 history 模式下 不支持 `pushState`，会通过 重新加载页面 (`window.location.href = href`)的方式实现页面跳转。

使用 vue router 时，如果 history 模式下不支持 `pushState`，会根据 `fallback` 配置项 来进行下一步处理。如果 `fallback` 为 `true`，回退到 hash 模式；如果 `fallback` 为 `false`，通过重新加载页面的方式实现页面跳转。

懒加载实现过程不同

vue router在路由懒加载 过程中，会先去获取懒加载页面对应的js文件。等 懒加载页面对应的js文件加载并执行完毕，才会开始渲染懒加载页面。

react router在 路由懒加载 过程中，会先去获取懒加载页面对应的js文件，然后渲染 loading 页面。等懒加载页面对应的js文件加载并执行完毕，触发更新，渲染懒加载页面。