React router精讲

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

```
import { render } from "react-dom";
import {
  BrowserRouter,
  Routes,
  Route,
} from "react-router-dom";
// import your route components too

render(
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<App />}>
        <Route index element={<Home />} />
        <Route path="teams" element={<Teams />}>
          <Route path=":teamId" element={<Team />} />
          <Route path="new" element={<NewTeamForm />} />
          <Route index element={<LeagueStandings />} />
        </Route>
      </Route>
    </Routes>
  </BrowserRouter>,
  document.getElementById("root")
);
```

config+hooks用法：

```typescript
import { render } from "react-dom";
import { useRoutes, BrowserRouter } from 'react-router-dom'

function Routes() {
  return useRoutes([
    {
      path: '/',
      element: <App>,
      children: [
      {
        path: '/,
        element: <Home />,
      },
      {
        path: '/teams',
        element: <Teams />
        children: [
          {
            path: ':tamId',
            element: <Team />
          }
        ]
      }]
    },
   ])
}

render(
  <BrowserRouter>
    <Routes />
  </BowserRouter>,
  document.getElementById("root")
)

```

（2）页面跳转

jsx用法：

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

hooks用法：

```typescript
const location = useLocation()

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
ReactDOM.render(
    <BrowserRouter>
      <App />
    </BrowserRouter>,
  document.getElementById('root')
)
```

BrowserRouter实现：

react-router-dom/index.tsx : 142 JavaScript 复制代码

```
export function BrowserRouter({
  basename,
  children,
  window,
}: BrowserRouterProps) {
  let historyRef = React.useRef<BrowserHistory>();
  if (historyRef.current == null) {
    // 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
    historyRef.current = createBrowserHistory({ window });
  }

  let history = historyRef.current;
  let [state, setState] = React.useState({
    action: history.action,
    location: history.location,
  });
  // history变化时重新渲染
  React.useLayoutEffect(() => history.listen(setState), [history]);

  return (
    <Router
      basename={basename}
      children={children}
      location={state.location}
      navigationType={state.action}
      navigator={history}
    />
  );
}
```

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

Router实现：

JavaScript | 复制代码

```javascript
export function Router({
  basename: basenameProp = "/",
  children = null,
  location: locationProp,
  navigationType = NavigationType.Pop,
  navigator,
  static: staticProp = false,
}: RouterProps): React.ReactElement | null {

  let basename = normalizePathname(basenameProp);
  let navigationContext = React.useMemo(
    () => ({ basename, navigator, static: staticProp }),
    [basename, navigator, staticProp]
  );

  if (typeof locationProp === "string") {
    locationProp = parsePath(locationProp);
  }

  let {
    pathname = "/",
    search = "",
    hash = "",
    state = null,
    key = "default",
  } = locationProp;

  let location = React.useMemo(() => {
    let trailingPathname = stripBasename(pathname, basename);

    if (trailingPathname == null) {
      return null;
    }

    return {
      pathname: trailingPathname,
      search,
      hash,
      state,
      key,
    };
    // 当location中有如下变化时重新生成location对象，触发页面重新渲染
  }, [basename, pathname, search, hash, state, key]);


```

```
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：

JavaScript | 复制代码

```
export function Routes({
  children,
  location,
}: RoutesProps): React.ReactElement | null {
  return useRoutes(createRoutesFromChildren(children), location);
}

export function useRoutes(
  routes: RouteObject[],
  locationArg?: Partial<Location> | string
): React.ReactElement | null {
  let { matches: parentMatches } = React.useContext(RouteContext);
  let routeMatch = parentMatches[parentMatches.length - 1];
  let parentParams = routeMatch ? routeMatch.params : {};
  let parentPathname = routeMatch ? routeMatch.pathname : "/";
  let parentPathnameBase = routeMatch ? routeMatch.pathnameBase : "/";
  let parentRoute = routeMatch && routeMatch.route;
  let locationFromContext = useLocation();
  let location;
  if (locationArg) {
    let parsedLocationArg =
      typeof locationArg === "string" ? parsePath(locationArg) :
locationArg;

    location = parsedLocationArg;
  } else {
    location = locationFromContext;
  }
  let pathname = location.pathname || "/";
  let remainingPathname =
    parentPathnameBase === "/"
      ? pathname
      : pathname.slice(parentPathnameBase.length) || "/";
  let matches = matchRoutes(routes, { pathname: remainingPathname });

  return _renderMatches(
    matches &&
      matches.map((match) =>
        Object.assign({}, match, {
          params: Object.assign({}, parentParams, match.params),
          pathname: joinPaths([parentPathnameBase, match.pathname]),
          pathnameBase:
            match.pathnameBase === "/"
              ? parentPathnameBase
              : 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：

history/packages/history/index.ts: 364 JavaScript 复制代码

```
export function createBrowserHistory(
  options: BrowserHistoryOptions = {}
): BrowserHistory {
  let { window = document.defaultView! } = options;
  let globalHistory = window.history;

  // 获取当前history中的index和location对象
  function getIndexAndLocation(): [number, Location] { }

  let blockedPopTx: Transition | null = null;

  // 处理返回上一页
  function handlePop() {}

  // 监听浏览器的popState事件
  window.addEventListener(PopStateEventType, handlePop);


  // 操作history对象
  function applyTx(nextAction: Action) {
    action = nextAction;
    [index, location] = getIndexAndLocation();
    listeners.call({ action, location });
  }
  // push操作
  function push(to: To, state?: any) {}

  // replace操作
  function replace(to: To, state?: any) {}

  // 前进或后退n页
  function go(delta: number) {}

  let history: BrowserHistory = {
    get action() {
      return action;
    },
    get location() {
      return location;
    },
    createHref,
    push,
    replace,
    go,
    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：

JavaScript | 复制代码

```javascript
function getIndexAndLocation(): [number, Location] {
    let { pathname, search, hash } = window.location;
    let state = globalHistory.state || {};
    return [
      state.idx,
      readOnly<Location>({
        pathname,
        search,
        hash,
        state: state.usr || null,
        key: state.key || "default",
      }),
    ];
  }
```

push实现：

JavaScript 复制代码

```
  function push(to: To, state?: any) {
    let nextAction = Action.Push;
    let nextLocation = getNextLocation(to, state);
    function retry() {
      push(to, state);
    }

    if (allowTx(nextAction, nextLocation, retry)) {
      let [historyState, url] = getHistoryStateAndUrl(nextLocation, index + 1);

      try {
        // 操作浏览器的history
        globalHistory.pushState(historyState, "", url);
      } catch (error) {
        window.location.assign(url);
      }
      // 处理history对象并回调
      applyTx(nextAction);
      /**
        function applyTx(nextAction: Action) {
          action = nextAction;
          [index, location] = getIndexAndLocation();
          listeners.call({ action, location });
        }
      */
    }
  }
```

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?

JavaScript 复制代码

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

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：

- borwserRouter
- 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。