[React Router V6源码] 来聊聊 React Router

React Router V6 源码

路由的基本原理

> 在单页面应用 （SPA）中，笔者会把路由理解为前端组件，路由的切换则可以理解为切换组件的生命周期，事实上，在 React Router 源码当中，就是通过 **url** 来筛选出匹配的 **React** 组件进行渲染。

BroswerRouter

💡 利用 HTML5 规范提供的 **history** 接口实现对前端路由 url 的控制。 **pushState**, **replaceState**, **back**, **forward**, **go**, **onpopState** 对浏览器的会话历史记录进行操作。

javascript 复制代码

```
<a onclick="go('/a')">/a</a>

let histroy = window.history;

function render () {
        root.innerHMLT = root.location.name;
}

const oldPushState = history.pushState;

history.pushState = function (state, title, url) {
        oldPushState.apply(history, [...arguments]);
        render()
}

// 只有当你前进和后退的时候会触发，pushState 不会触发,所以缓存旧方法，派生新方法。
window.onpopstate =  render;

function go(path) {
        history.pushState({}, null, path);
}
```

```
function forward() {
        histroy.go(1);
}

function back() {
        history.go(-1);
}
```

HashRouter

💡 原理是通过监听 **hash** 的变化，来控制路由 url 的改变。由于 **BroswerRouter** 中, 可以根据浏览器 **BOM API history** 会维护一个浏览器会话消息的栈结构，但是 **HashRouter** 不具备这样的栈结构，所以在 **React Router** 源码 中的 **HashRouter** 手动维护了一个栈结构, 实现类似 **history** 同样的 **go, goBack, goForward, push, listen, action**

javascript 复制代码

```
<a onclick="#/a">/a</a>
window.addEventListener("hashChange", () => {})
```

React Router 的 使用和理解

javascript 复制代码

```
<BrowserRouter>
    <div>
      <ul>
        <li>
          <Link to={"/"}>首页</Link>
        </li>
        <li>
          <Link to={"/user"}>用户</Link>
        </li>
        <li>
          <Link to={"profile"}>详情</Link>
        </li>
      </ul>
    </div>

    <Routes>
      <Route path="/" element={<Home />}></Route>
      <Route path="/user/*" element={<User />}>
        <Route path="/list" element={<Profile />}></Route>
      </Route>
      <Route path="/profile" element={<Profile />}></Route>
    </Routes>
  </BrowserRouter>
```

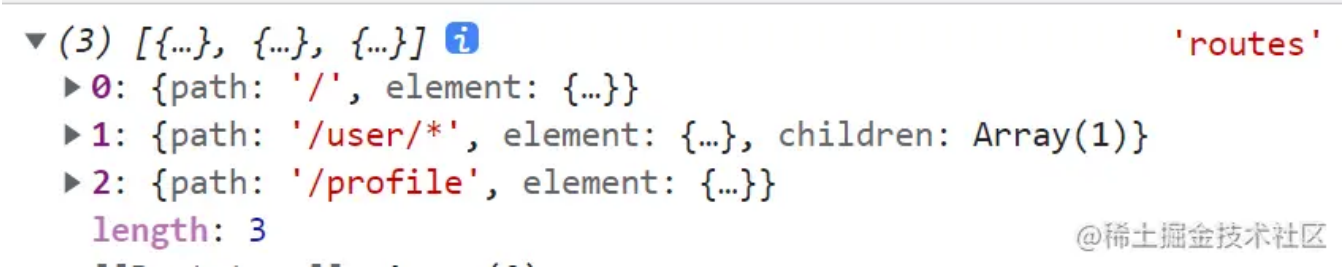
```
import React from "react";
import { Outlet } from "../react-router";

export function User() {
  return (
    <div>
      user
      <Outlet />
    </div>
  );
}
```

浏览器输入 http://localhost:3000/user/list 回车之后，嵌套路由的UI是怎么渲染的的?

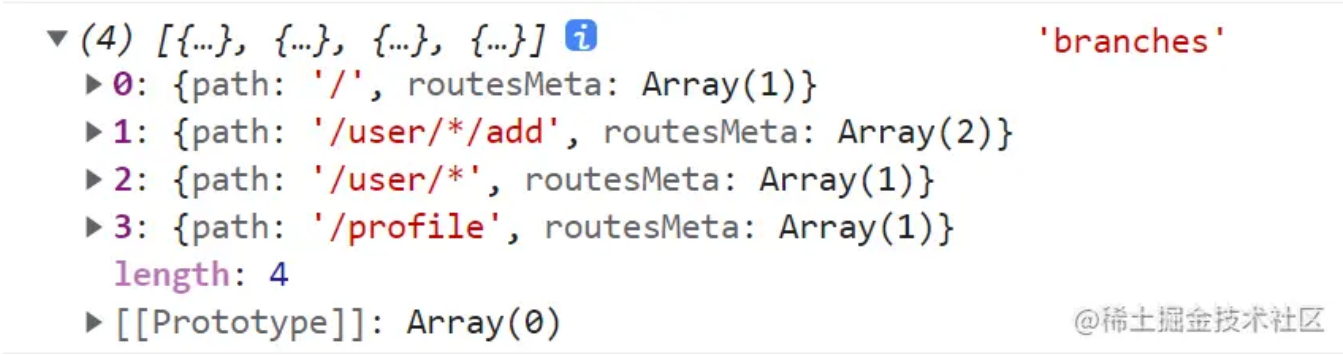
首先 遍历到 **BroserRouter** 组件，**BrowerRouter 组件** 会提供和创建两个上下文 **Context**，一个是 **HistroyContext** （通过 **history-lib** 创建的 **history**），另一个是 **LocationContext**.(**window.location**)。由此看来，我们每次在最外层 包裹上 **BroserRouter/HashRouter** 就是为了 消费这两个上下文 **Context**。源码当中的 **useNaviagete useParams …… Hook**，也同样是消费了这两个上下文来完成功能。

然后遍历到 **Routes** 中的 **Route** 对象，将这些 **Route** 对象，第一步：通过 **createRouteChildren 函数** 转换成配置式路由，**{path: "", element: "" children: []}**



图一：第一步之后生成的 route 对象

第二步：通过 **flatenRoutes** 扁平化路由，生成路由源数据 **routeMeta**, 然后通过 **computedScpore** 给每个路由计算分数，实现匹配优先级（*（通配符） -2 分，"" + 1， 静态字段：user + 6, index + 6) 分数的主要作用就是为 **索引路由，动态路由，通配符路由 提供路由优先级和默认路由服务。**



图二：第二步之后生成的 branch 对象

第三步：开始循环匹配路由对象，这里运用了大量的正则表达式来进行精准的分组和匹配，想知道细节的朋友可以往后看源码，匹配成功后 存放 到 **matches 数组** 当中。

比如 路由为 **/user**，匹配到的就是一个 **match** 对象， 路由为 **/user/list** 就是两个 **match** 对象 父 **match**对象 的 **element** 是 **<User/>** 子 **match** 对象的 **element** 是 **<List/>**。如果三级，四级路由以此类推。

图三：/user 生成的 匹配成功的 matches 对象

图四：/user/list 生成的 匹配成功的 matches 对象

最后的环节： **renderOutlet** 过程，看第一遍可能不大好理解，**我们可以带着问题去理解，为什么每次在父组件 当中 使用 Outlet 渲染的是子组件，子组件当中使用 Outlet 渲染的是子组件的子组件?**

带着最后一个问题，我们来看最后一个过程，如果拿到了两个 **match** 对象，这两个 **match** 对象，会通过一个 **reduce** 方法（reduceRight）， 从嵌套最深的子元素开始，创建一个

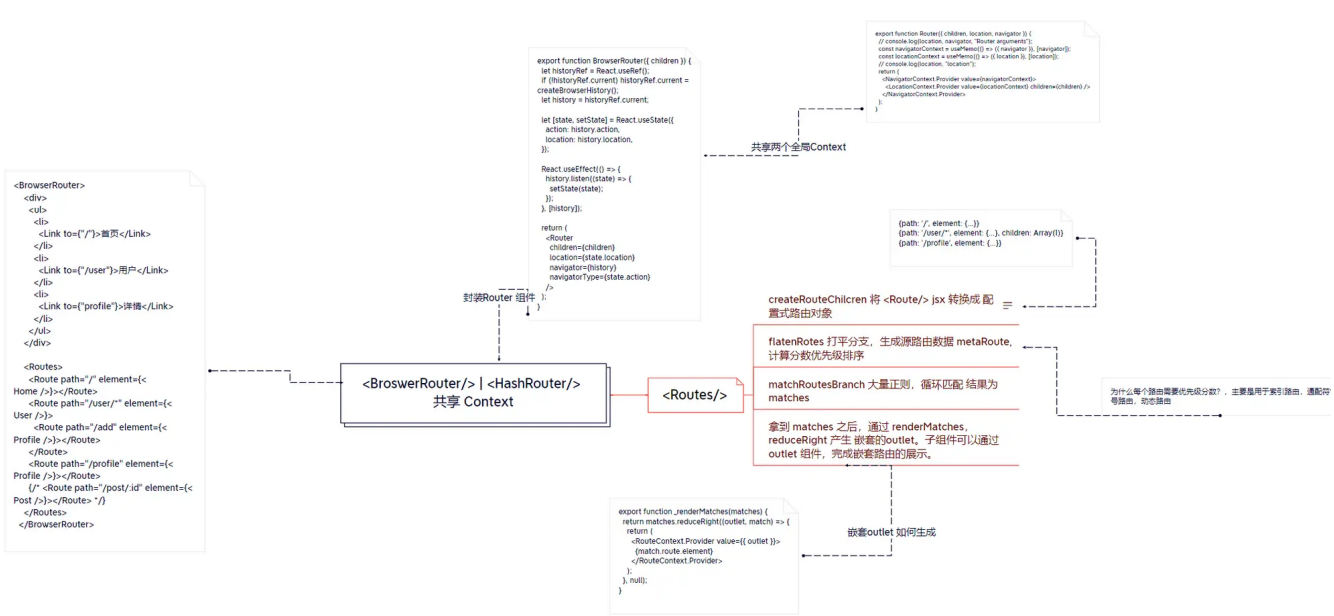
RoutesrContext 共享 **outlet** 对象， **children** 是 子组件， 最开始 **outlet** 是 **null**(因为是层级最深的子组件，所以是，也应该是null)。

```javascript
export function _renderMatches(matches) {
  return matches.reduceRight((outlet, match) => {
    return (
      <RouteContext.Provider value={{ outlet }}>
        {match.route.element}
      </RouteContext.Provider>
    );
  }, null);
}
```

从第二个父组件 的 **outlet** 开始， 通过 **reduce** 方法的特性指向上一个子组件的 **RouteContxt** 对象 **outlet** 和 子组件，**reduce** 遍历完成之后形成一个层层嵌套的 **outlet** 指向。

所以 当路由是 **/user/list** ， 在父组件中放置 **OutLet** 之前，只会显示一个父组件 **<User/>**，不用多说 因为父组件的 **Outlet** 指向了 子组件。没放置 **Oulet** 又怎么能显示子组件呢?

这时通过 在父组件当中使用 **Outlet** 组件 或者 **useOutlet** 组件，可以拿到子组件。如果还有子组件，在子组件里面使用 **Outlet** 得到 子组件的子组件。实现嵌套路由的显示。自此也就结束了。



图五：源码流程图

通过 Link 标签 切换路由至 http://localhost:3000/ 发生了什么？

```javascript
export function Link(props) {
  let navigate = useNavigate();
  let { to, children } = props;

  return <a onClick={() => navigate(to)}>{children}</a>;
}
```

通过代码可以看到，切换路由实际上调用了 navigate 方法。

```javascript
export function useNavigate() {
  const { navigator } = React.useContext(NavigatorContext);
  // debugger;
  const navigate = React.useCallback(
    (to) => {
      navigator.push(to);
    },
    [navigator]
  );
  return navigate;
}
```

navigate 方法，实则是调用了 **navigator.push** 方法，这里的 **navigator** 就是 通过 **history** 这个库，通过 createBrowserHistory 方法，创建的 history。我们来看看 push 方法是什么样子的。

```javascript
  function push(pathname, nextState) {
    if (typeof pathname === "object") {
      state = pathname.state;
      pathname = pathname.pathname;
    } else {
      state = nextState;
    }
    globalHistory.pushState(state, null, pathname);

    let location = {
      state: globalHistory.state,
      pathname: window.location.pathname,
    };
```

```
    notify({ action: "PUSH", location });
  }
```

很清楚了，是通过调用 **window.history.pushState()** 方法，来变更浏览器的 **url**，并且传入状态。然后调用 **notify** 发布通知 调用订阅的 listenr 函数，listner 函数通过 setState 让组件刷新，此时又回到了问题一的整体流程。

```javascript 复制代码
// listner 订阅
React.useLayoutEffect(() => history.listen(setState), [history]); // listener


  /* 监听notify */
  function notify(newState) {
    Object.assign(globalHistory, newState);
    listeners.forEach((listener) =>
      listener({ location: globalHistory.location })
    );
  }
```

部分源码

💡 **React V6** 通过 **history** 这个库， 创建了 **hashHistory** 对象 和 **browserHistory** 对象。解决了我们上面讨论的 **hash** 式路由，内部没有像 浏览器 BOM API **history** 一样的 **api** 来记录浏览器会话消息的问题，同时二次封装 history 支持 action，订阅 listen, 发布通知 notify, 来配合 **React** 刷新组件。

React Router 部分源码 实现

其余源码仓库地址 github.com/Ryan-eng-de...

```javascript 复制代码
export function compilePath(path, end) {
  const pathnames = [];

  let regexpSource =
    "^" +
    path
      .replace(/\/*\*?$/, "")
      .replace(/^\/*/, "/")
      .replace(/:(\w+)/g, (_, key) => {
        pathnames.push(key);
        return "([^\\/]+?)";
```

```
      });
  // debugger;
  if (path.endsWith("*")) {
    pathnames.push("*");
    regexpSource += "(?:\\/(.+)|\\/*)$";
  } else {
    regexpSource += end ? "\\/*$" : "(?:\b|\\/|$)";
  }
  let matcher = new RegExp(regexpSource);
  return [matcher, pathnames];
}

export function matchPath({ path, end }, pathname) {
  //pathname: /id/100/20 || matcher /id/([^\/]+?)/([^\/]+?)
  /* 路径编译为正则 */
  let [matcher, paramNames] = compilePath(path, end);

  let match = pathname.match(matcher);
  if (!match) return null;
  const matchPathname = match[0];
  // debugger;
  let pathnameBase = matchPathname.replace(/(.)\/+$/, "$1");
  let values = match.slice(1);
  let captureGroups = match.slice(1);
  /* 拼出paramsNames对象 */
  let params = paramNames.reduce((memo, paramName, index) => {
    if (paramName === "*") {
      let splitValue = captureGroups[index] || "";
      /* 截取*之前的作为父串 */
      pathnameBase = matchPathname
        .slice(0, matchPathname.length - splitValue.length)
        .replace(/(.)\/+$/, "$1");
    }
    memo[paramName] = values[index];
    return memo;
  }, {});
  return { params, path, pathname: matchPathname, pathnameBase };
}
```