React Router V6.4 （React Router 组件-钩子函数篇）

React Router `v6.4` 进入 钩子函数-组件篇，开始封装 React 函数钩子和 React 组件。此文章讨论 react-router-dom （web 平台）。下面分析梳洗从 context 和类型开始，最好配合 react-router clone 源码 一起查看。

上下文 context

> context api 为跨组件，提供数据支持。

ts 复制代码

```ts
// 静态 data routers 的上下文
export const DataStaticRouterContext =
  React.createContext<StaticHandlerContext | null>(null);

// data routers 的上下文
export const DataStaticRouterContext =
  React.createContext<StaticHandlerContext | null>(null);

// data router 状态上下文
export const DataRouterStateContext = React.createContext<
  Router["state"] | null
>(null);

// 等待上下文
export const AwaitContext = React.createContext<TrackedPromise | null>(null);

// route 上下文
export const RouteContext = React.createContext<RouteContextObject>({
  outlet: null,
  matches: [],
});

// route 错误上下文
export const RouteErrorContext = React.createContext<any>(null);
```

Index 类型路由与非 Index 类型路由

```ts 复制代码
export type RouteObject = IndexRouteObject | NonIndexRouteObject;

// index 路由
export interface IndexRouteObject {
  caseSensitive?: AgnosticIndexRouteObject["caseSensitive"];
  path?: AgnosticIndexRouteObject["path"];
  id?: AgnosticIndexRouteObject["id"];
  loader?: AgnosticIndexRouteObject["loader"];
  action?: AgnosticIndexRouteObject["action"];
  hasErrorBoundary?: AgnosticIndexRouteObject["hasErrorBoundary"];
  shouldRevalidate?: AgnosticIndexRouteObject["shouldRevalidate"];
  handle?: AgnosticIndexRouteObject["handle"];
  index: true;
  children?: undefined;
  element?: React.ReactNode | null;
  errorElement?: React.ReactNode | null;
}

// 非 index 路由
export interface NonIndexRouteObject {
  caseSensitive?: AgnosticNonIndexRouteObject["caseSensitive"];
  path?: AgnosticNonIndexRouteObject["path"];
  id?: AgnosticNonIndexRouteObject["id"];
  loader?: AgnosticNonIndexRouteObject["loader"];
  action?: AgnosticNonIndexRouteObject["action"];
  hasErrorBoundary?: AgnosticNonIndexRouteObject["hasErrorBoundary"];
  shouldRevalidate?: AgnosticNonIndexRouteObject["shouldRevalidate"];
  handle?: AgnosticNonIndexRouteObject["handle"];
  index?: false;
  children?: RouteObject[];
  element?: React.ReactNode | null;
  errorElement?: React.ReactNode | null;
}
```

我们看到在路由类中有 `element/errorElement` 两种属性，还有熟悉的 `path` 属性，和不太熟悉的 `action` 和 `loader` 两个函数

Navigator 导航跳转相关的类型

```ts 复制代码
export type RelativeRoutingType = "route" | "path";

export interface NavigateOptions {
  replace?: boolean;
  state?: any;
  preventScrollReset?: boolean;
```

```ts
  relative?: RelativeRoutingType;
}

export interface Navigator {
  createHref: History["createHref"];
  encodeLocation?: History["encodeLocation"];
  go: History["go"];
  push(to: To, state?: any, opts?: NavigateOptions): void;
  replace(to: To, state?: any, opts?: NavigateOptions): void;
}

interface NavigationContextObject {
  basename: string;
  navigator: Navigator;
  static: boolean;
}
```

location 上下文类型

ts 复制代码

```ts
interface LocationContextObject {
  location: Location;
  navigationType: NavigationType;
}
```

route 上下文类型

ts 复制代码

```ts
export interface RouteContextObject {
  outlet: React.ReactElement | null;
  matches: RouteMatch[];
}
```

熟悉这些类型，有助于对后面的钩子函数和组件封装。本质是对导航跳转函数和 state 对象以及路由路径的封装。

react-router 中的 hooks

为什么要在组件前面讲 hooks， 因为 hooks 被组件 依赖 ，没有自己封装过自己 hooks 的小伙伴看看这篇自定 hooks, 其实就是定义一个函数，函数中能使用 react 内置 hook 或者第三方钩子函数（可有返回值）。

hooks 列表

hooks 名	说明
useHref	返回给定"to"值的完整 href。这对于构建自定义链接非常有用，这些链接也可以访问并保留右键单击行为。
useInRouterContext	如果此组件是 `<Router>` 的后代，则返回 true。
useLocation	返回当 location 对象，该对象表示 Web 中的当前 URL浏览器。
useNavigationType	返回当前导航操作，该操作描述路由器如何通过历史堆栈上的弹出、推送或替换到达当前位置。
useMatch	如果给定模式与当前 URL 匹配，则返回 PathMatch 对象。这对于需要知道“活动”状态的组件很有用，例如 `<NavLink>` 。
useNavigate	返回用于更改位置的命令式方法。由 `<Link>` s 使用，但也可以被其他元素用来更改位置。
useOutletContext	返回路由层次结构此级别的子路由的上下文（如果提供）。
useOutlet	返回路由层次结构此级别的子路由的元素。在内部用于 `<Outlet>` 呈现子路由。
useParams	返回当前 URL 中与路由路径匹配的动态参数的键/值对的对象。
useResolvedPath	根据当前位置解析给定"to"值的路径名。
useRoutes	返回与当前位置匹配的路由元素，已准备好使用正确的上下文来呈现路由树的其余部分。树中的路由元素必须呈现 才能 `<Outlet>` 呈现其子路由的元素。
useDataRouterContext	DataRouter 上下文
useDataRouterState	DataRouter 对应 state
useNavigation	返回当前导航，默认为“空闲”导航没有正在进行的导航
useRevalidator	返回用于手动触发重新验证的重新验证函数，以及任何手动重新验证的当前状态
useMatches	返回活动路由匹配项，这对于访问父/子路由的 loaderData 或路由“handle”属性很有用
useLoaderData	返回最近的祖先路由加载器的加载程序数据

hooks 名 说明

useRouteLoaderData	返回给定路由 ID 的加载器数据
useActionData	返回最近祖先路由操作的操作数据
useRouteError	返回最近的祖先路由错误，该错误可能是加载程序/操作错误或呈现错误。 这旨在从您的 errorElement 调用以显示正确的错误消息。
useAsyncValue	返回来自最近祖先的快乐路径数据 `<Await />` 值
useAsyncError	返回来自最接近祖先的错误 `<Await />` 值

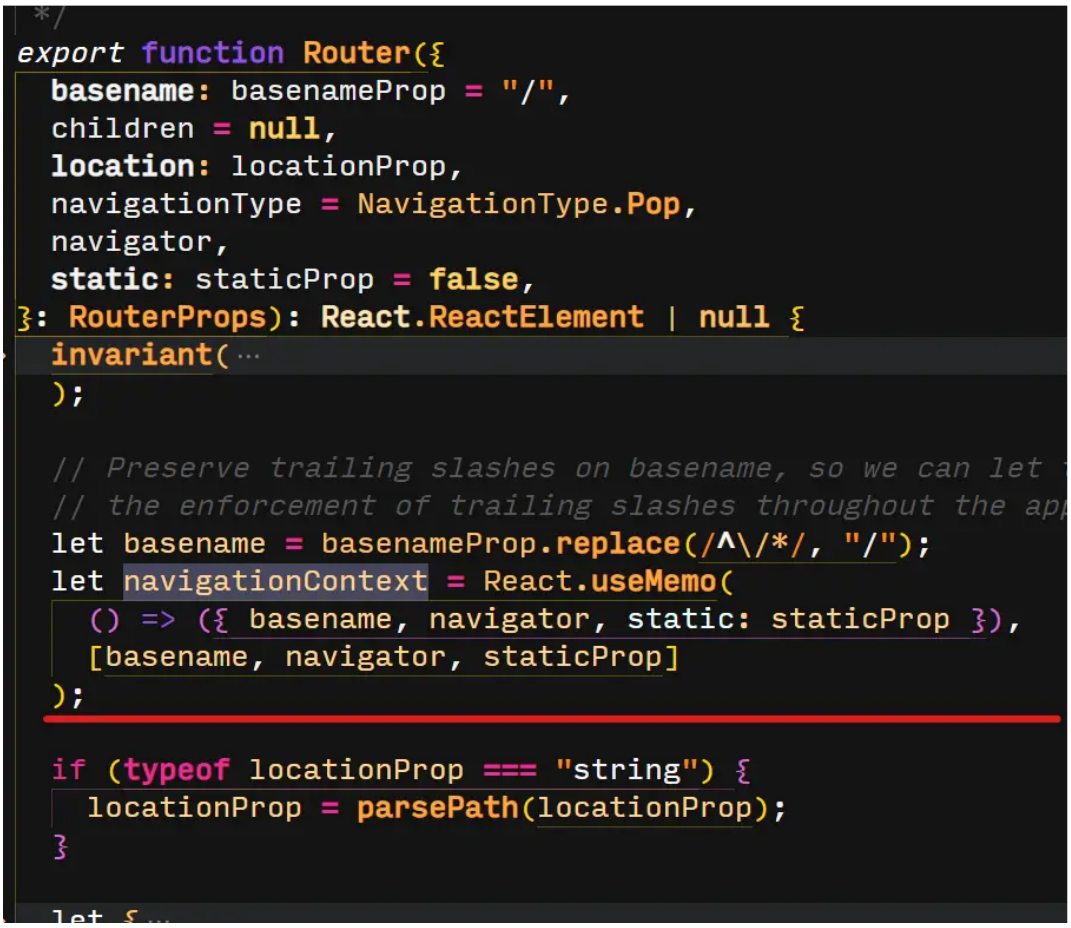
useHref

> 返回给定"to"值的完整 href

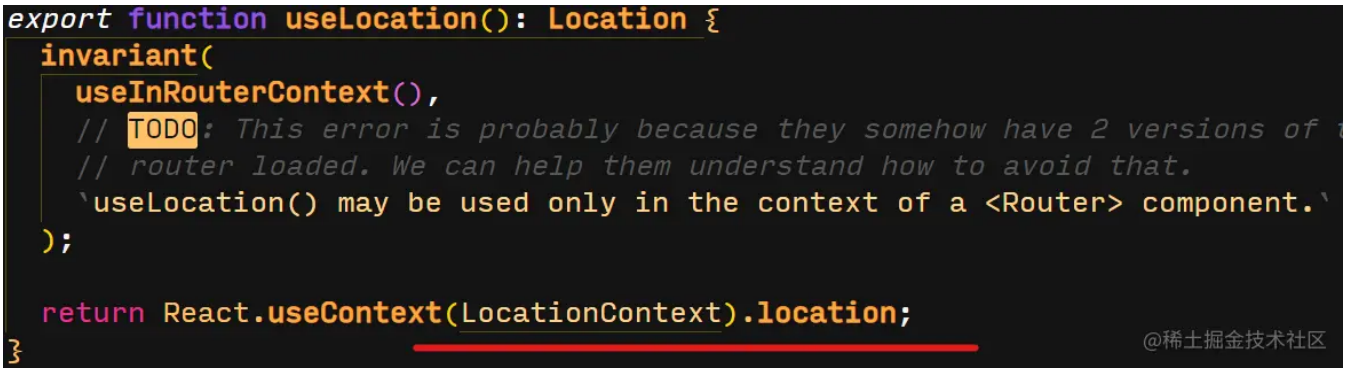
ts 复制代码

```ts
export function useHref(
  to: To,
  { relative }: { relative?: RelativeRoutingType } = {}
): string
  let { basename, navigator } = React.useContext(NavigationContext);
  let { hash, pathname, search } = useResolvedPath(to, { relative });

  let joinedPathname = pathname;
  if (basename !== "/") {
    joinedPathname =
      pathname === "/" ? basename : joinPaths([basename, pathname]);
  }

  return navigator.createHref({ pathname: joinedPathname, search, hash });
}
```

`NavigationContext` 赋值发生在 Router 组件定义内部，所以要使用 useHref 钩子必须包裹在 `<Router />` 后者扩展 Router 的组件内部。如下图



useLocation



LocationContext 赋值发生在 LocationContext.Provider 中：

useNavigate

> 返回是一个使用 useCallback 包裹的 navigate 函数， 基本实现：

ts 复制代码

```ts
let navigate: NavigateFunction = React.useCallback(
    (to: To | number, options: NavigateOptions = {}) => {
      // 实现省略
    },
    [basename, navigator, routePathnamesJson, locationPathname]
  );
```

内部实现： 使用 router 中 navigator 对象进行跳转操作

useOutlet

实现：

```tsx 复制代码
export function useOutlet(context?: unknown): React.ReactElement | null {
  let outlet = React.useContext(RouteContext).outlet;
  if (outlet) {
    return (
      <OutletContext.Provider value={context}>{outlet}</OutletContext.Provider>
    );
  }
  return outlet;
}
```

两个核心：

- 本质就是返回了组件（可以是 null）
- 组件会根据 RouteContext 上下文的 outlet 属性进行变化，这是路由组件切换的原理

useParams

> 可以快速的用钩子函数的形式获取 路径中参数 ，注意不是带有search

实现: RouteContext 上下文中的 matches 对象中 params 属性

```ts 复制代码
export function useParams(){
  let { matches } = React.useContext(RouteContext);
  let routeMatch = matches[matches.length - 1];
  return routeMatch ? (routeMatch.params as any) : {};
}
```

useRoutes

> <Routes /> 组件使用效果一致，实现如何下

```ts 复制代码
export function useRoutes(
  routes: RouteObject[],
  locationArg?: Partial<Location> | string
): React.ReactElement | null {}
```

内部使用 `_renderMatches` 来渲染 match 的组件 renderedMatches， 然后将 renderedMatches 返回出去，如果提供了 location 相关的参数，还需要提供渲染 location 上下文

useNavigation

- 实现: 从 dataRouteState 中获取 state，返回其 navigation 属性

```ts 复制代码
export function useNavigation() {
  let state = useDataRouterState(DataRouterStateHook.UseNavigation);
  return state.navigation;
}
// 下面是 DataRouterStateContext 定义
export const DataRouterStateContext = React.createContext<
  Router["state"] | null
>(null);

// DataRouterStateContext.Provider 组件在使用 Router 组件时初始化 state
```

useRevalidator

- 实现 useRevalidator

```ts 复制代码
export function useRevalidator() {
  let dataRouterContext = useDataRouterContext(DataRouterHook.UseRevalidator);
  let state = useDataRouterState(DataRouterStateHook.UseRevalidator);
  return {
    revalidate: dataRouterContext.router.revalidate,
    state: state.revalidation,
  };
}
```

此钩子函数返回 router 对象中的 revalidate 和 state 对象的 reavaliation 属性。

useMatches

> 返回活动路由匹配项

```ts 复制代码
export function useMatches() {
  let { matches, loaderData } = useDataRouterState(
    DataRouterStateHook.UseMatches
  );
  return React.useMemo(
    () =>
      matches.map((match) => {
        let { pathname, params } = match;
        return {
          id: match.route.id,
          pathname,
          params,
          data: loaderData[match.route.id] as unknown,
          handle: match.route.handle as unknown,
        };
      }),
    [matches, loaderData]
  );
}
```

useLoaderData

> 返回最近祖先路由加载器的加载器数据

```ts 复制代码
export function useLoaderData(): unknown {
  let state = useDataRouterState(DataRouterStateHook.UseLoaderData);
  let route = React.useContext(RouteContext);
  let thisRoute = route.matches[route.matches.length - 1];
  return state.loaderData[thisRoute.route.id];
}
```

useRouteLoaderData

> 返回给定routeId的loaderData

```tsx 复制代码
export function useRouteLoaderData(routeId: string): unknown {
  let state = useDataRouterState(DataRouterStateHook.UseRouteLoaderData);
  return state.loaderData[routeId];
}
```

useActionData

> 返回最近祖先路由操作的操作数据

ts 复制代码

```ts
export function useActionData(): unknown {
  let state = useDataRouterState(DataRouterStateHook.UseActionData);

  let route = React.useContext(RouteContext);

  return Object.values(state?.actionData || {})[0];
}
```

useRouteError

> 返回最近的祖先路由错误，这可能是 loader/action 错误或呈现错误

ts 复制代码

```ts
export function useRouteError(): unknown {
  let error = React.useContext(RouteErrorContext);
  let state = useDataRouterState(DataRouterStateHook.UseRouteError);
  let route = React.useContext(RouteContext);
  let thisRoute = route.matches[route.matches.length - 1];

  if (error) {
    return error;
  }

  return state.errors?.[thisRoute.route.id];
}
```

useAsyncValue

> 返回来自最近祖先的快乐路径数据 `<Await />` 值

ts 复制代码

```ts
export function useAsyncValue(): unknown {
  let value = React.useContext(AwaitContext);
  return value?._data;
}
```

useAsyncError

> 返回来自最接近祖先的错误 `<Await />` 值

```ts
export function useAsyncError(): unknown {
  let value = React.useContext(AwaitContext);
  return value?._error;
}
```

---

react-router 中的组件

组件	说明
`<RouterProvider />`	给定一个Remix Router实例，呈现适当的UI
`<MemoryRouter />`	将所有 `<Router>` 条目存储在内存中
`<Navigate />`	更改当前位置
`<Outlet />`	渲染子路由的元素（如果有）。
`<Route />`	声明应在特定URL路径处呈现的元素。
`<Router />`	为应用程序的其余部分提供位置上下文。
`<Routes />`	呈现分支的 `<Route>` 元素嵌套树的容器最符合当前位置的。
`<Await />`	用于在加载器函数中从返回 `defer()` 中呈现延迟加载的数据的组件
`<AwaitErrorBoundary />`	`<Await />` 的错误边界
`<ResolveAwait />`	间接利用 `<Await>` 上的渲染道具API的useAsyncValue

RouterProvider

> 要渲染 `<Routes />` 组件，同时要提供好上下文数据：DataRouterContext/DataRouterStateContext

tsx 复制代码

```tsx
<DataRouterContext.Provider
  value={{
    router,
    navigator,
    static: false,
    // Do we need this?
    basename,
  }}
>
  <DataRouterStateContext.Provider value={state}>
    <Router
      basename={router.basename}
      location={router.state.location}
      navigationType={router.state.historyAction}
      navigator={navigator}
    >
      {router.state.initialized ? <Routes /> : fallbackElement}
    </Router>
  </DataRouterStateContext.Provider>
</DataRouterContext.Provider>
```

MemoryRouter

> 不需要浏览器环境，它将所有 `<Router>` 条目存储在内存中运行，方便测试用例和native环境

ts 复制代码

```ts
export function MemoryRouter({
  basename,
  children,
  initialEntries,
  initialIndex,
}: MemoryRouterProps): React.ReactElement {
  let historyRef = React.useRef<MemoryHistory>();
  if (historyRef.current == null) {
    historyRef.current = createMemoryHistory({
      initialEntries,
      initialIndex,
      v5Compat: true,
    });
```

```
  }

  let history = historyRef.current;
  let [state, setState] = React.useState({
    action: history.action,
    location: history.location,
  });

  React.useLayoutEffect(() => history.listen(setState), [history]);

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

`<Navigate />` 导航组件

> Navigate 没有渲染组件，直接返回 null, 以组件形式进行跳转(组件跳转（非命令式跳转）)

ts 复制代码

```
export function Navigate({
  to,
  replace,
  state,
  relative,
}: NavigateProps): null {
  let dataRouterState = React.useContext(DataRouterStateContext);
  let navigate = useNavigate();

  React.useEffect(() => {
    if (dataRouterState && dataRouterState.navigation.state !== "idle") {
      return;
    }
    navigate(to, { replace, state, relative });
  });

  return null;
}
```

`<Outlet />` 组件

> 路由需要渲染的位置组件

ts 复制代码

```ts
export function Outlet(props: OutletProps): React.ReactElement | null {
  return useOutlet(props.context);
}
```

`<Route />` 路由器组件

`<Route />` 组件不能单独使用必须放在 `<Routes/>` 组件里面，给 `<Routes />` 组件提供数据

`<Router />` 路由组件

tsx 复制代码

```tsx
export function Router({
  basename: basenameProp = "/",
  children = null,
  location: locationProp,
  navigationType = NavigationType.Pop,
  navigator,
  static: staticProp = false,
}: RouterProps): React.ReactElement | null {
  let basename = basenameProp.replace(/^\/*/, "/");
  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,
    };
  }, [basename, pathname, search, hash, state, key]);

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

  return (
    <NavigationContext.Provider value={navigationContext}>
      <LocationContext.Provider
        children={children}
        value={{ location, navigationType }}
      />
    </NavigationContext.Provider>
  );
}
```

`<Routes />` 路由器容器组件

ts 复制代码

```
export function Routes({
  children,
  location,
}: RoutesProps): React.ReactElement | null {
  let dataRouterContext = React.useContext(DataRouterContext);
  let routes =
    dataRouterContext && !children
      ? (dataRouterContext.router.routes as DataRouteObject[])
      : createRoutesFromChildren(children);
  return useRoutes(routes, location);
}
```

`<Await />`

> 用于在加载器函数中从返回 `defer()` 中呈现延迟加载的数据的组件

tsx 复制代码

```tsx
export function Await({ children, errorElement, resolve }: AwaitProps) {
  return (
    <AwaitErrorBoundary resolve={resolve} errorElement={errorElement}>
      <ResolveAwait>{children}</ResolveAwait>
    </AwaitErrorBoundary>
  );
}
```

- `<AwaitErrorBoundary />` 组件是一个 Promise， 捕获的时 `<Await />` 组件的错误边界

ResolveAwait

> 间接利用 `<Await>` 上的渲染道具 API 的 useAsyncValue

tsx 复制代码

```tsx
function ResolveAwait({
  children,
}: {
  children: React.ReactNode | AwaitResolveRenderFunction;
}) {
  let data = useAsyncValue();
  if (typeof children === "function") {
    return children(data);
  }
  return <>{children}</>;
}
```

工具函数

- createRoutesFromChildren 用于传创建 routes

ts 复制代码

```ts
const router = createBrowserRouter(
  createRoutesFromElements(
    <Route path="/" element={<Root />}>
      <Route path="dashboard" element={<Dashboard />} />
      {/* ... etc. */}
    </Route>
  )
);
```

下面时 createRoutesFromChildren 函数的实现

ts 复制代码

```ts
// 递归的创建 routes
export function createRoutesFromChildren(
  children: React.ReactNode,
  parentPath: number[] = []
): RouteObject[] {
  let routes: RouteObject[] = [];

  React.Children.forEach(children, (element, index) => {
    if (!React.isValidElement(element)) {
      return;
    }

    if (element.type === React.Fragment) {
      routes.push.apply(
        routes,
        createRoutesFromChildren(element.props.children, parentPath)
      );
      return;
    }

    let treePath = [...parentPath, index];
    let route: RouteObject = {
      id: element.props.id || treePath.join("-"),
      caseSensitive: element.props.caseSensitive,
      element: element.props.element,
      index: element.props.index,
      path: element.props.path,
      loader: element.props.loader,
      action: element.props.action,
      errorElement: element.props.errorElement,
      hasErrorBoundary: element.props.errorElement != null,
      shouldRevalidate: element.props.shouldRevalidate,
      handle: element.props.handle,
    };

    if (element.props.children) {
      route.children = createRoutesFromChildren(
        element.props.children,
        treePath
      );
    }

    routes.push(route);
  });
```

```
  return routes;
}
```

测试

与 测试篇 相同的测试方法， 因为要选出 dom 所以引入 react-test-renderer 工具包将 react 组件渲染成 dom 结构

测试流程归纳：

- 准备组件（一般使用 MemoryRouter 作为顶层 router）
- 在组件中调用合适钩子函数
- TestRenderer.create 方法将组件渲染得到 rendererr
- renderer.JSON 获取 json 数据结构，然后进行 toMatchInlineSnapshot 获取行内快照测试
- renderer 其他的测试（如查找等测试）

测试示例

tsx 复制代码

```tsx
it("matches when the location is not overridden", () => {
    let renderer: TestRenderer.ReactTestRenderer;
    TestRenderer.act(() => {
      renderer = TestRenderer.create(
        <MemoryRouter initialEntries={["/users/michael"]}>
          <Routes>
            <Route path="home" element={<Home />} />
            <Route path="users/:userId" element={<User />} />
          </Routes>
        </MemoryRouter>
      );
    });

    expect(renderer.toJSON()).toMatchInlineSnapshot(`
      <div>
        <h1>
          User:
          michael
        </h1>
      </div>
```

```
    `);
  });
```

测试用例在于自己探索，在于自己实践。

---

react-router-dom 浏览器平台

工具函数

> 基于 createRouter 函数封装的创建路由的工具函数。

- createBrowserRouter
- createHashRouter
- createFetcherForm

hooks

钩子函数	说明
useDataRouterContext	获取 DataRouter 上下文
useDataRouterState	获取 DataRouter 中state 对象
useLinkClickHandler	使用 Link 组件中的点击事件
useSearchParams	获取 search 参数
useSubmit	使用 sumbmit 提交
useSubmitImpl	实现 Submit 调教
useFormAction	使用 form action
useFetcher	与 loader 和 acttion 交互，而不会导致导航。非常适合停留在同一页面上的任何交互。
useFetchers	提供页面上当前的所有提取进程。对于需要提供有关获取的 `"pending"/"optimistic"` UI 的布局和父路由很有用。
useScrollRestoration	使用滚动存储
useBeforeUnload	使用之前 beforeUnload

useSubmit 的实现

tsx 复制代码

```tsx
function useSubmitImpl(fetcherKey?: string, routeId?: string): SubmitFunction {
  let { router } = useDataRouterContext(DataRouterHook.UseSubmitImpl);
  let defaultAction = useFormAction();

  return React.useCallback(
    (target, options = {}) => {
      // 👇
      if (fetcherKey) {
        invariant(routeId != null, "No routeId available for useFetcher()");
        router.fetch(fetcherKey, routeId, href, opts);
      } else {
        router.navigate(href, opts);
```

```
      }
    },
    [defaultAction, router, fetcherKey, routeId]
  );
}
```

核心内容是：

- router.navigate 跳转还是使用 router.fetch api

useFetcher

useFetcher 返回值类型：包含 Form 组件/提交函数/load函数

ts 复制代码

```ts
export type FetcherWithComponents<TData> = Fetcher<TData> & {
  Form: ReturnType<typeof createFetcherForm>;
  submit: (
    target: SubmitTarget,
    // Fetchers cannot replace because they are not navigation events
    options?: Omit<SubmitOptions, "replace">
  ) => void;
  load: (href: string) => void;
};
```

- fetcherKey 在 useFetcher 扮演重要角色，类型中属性生成都需要这个 key

tsx 复制代码

```tsx
export function useFetcher<TData = any>(): FetcherWithComponents<TData> {
  let { router } = useDataRouterContext(DataRouterHook.UseFetcher);

  let route = React.useContext(RouteContext);
  let routeId = route.matches[route.matches.length - 1]?.route.id;
  let [fetcherKey] = React.useState(() => String(++fetcherId));
  let [Form] = React.useState(() => {
    return createFetcherForm(fetcherKey, routeId);
  });
  let [load] = React.useState(() => (href: string) => {
    router.fetch(fetcherKey, routeId, href);
  });
  let submit = useSubmitImpl(fetcherKey, routeId);
  let fetcher = router.getFetcher<TData>(fetcherKey);

  let fetcherWithComponents = React.useMemo(
    () => ({
      Form,
```

```
      submit,
      load,
      ...fetcher,
    }),
    [fetcher, Form, submit, load]
  );

  React.useEffect(() => {
    return () => {
      router.deleteFetcher(fetcherKey);
    };
  }, [router, fetcherKey]);

  return fetcherWithComponents;
}
```

useFetchers

> 提供页面上当前的所有提取进程

ts 复制代码

```
export function useFetchers(): Fetcher[] {
  let state = useDataRouterState(DataRouterStateHook.UseFetchers);
  return [...state.fetchers.values()];
}
```

组件

浏览器路由	说明
BrowserRouter	`<Router>` 用于 Web 浏览器。提供最干净的 URLs。
HashRouter	`<Router>` 用于 Web 浏览器的。将位置存储在 URL 的哈希部分中，以便不会将其发送到服务器。
HistoryRouter	接受 `<Router>` 预实例化的历史对象的。
Link	用标签实现的跳转链接
NavLink	一个 `<Link>` 包装器，它知道它是否"活动"。
Form	Form 组件
FormImpl	Form 组件的实现
ScrollRestoration	滚动存储组件

`<Form />` 组件实现

tsx 复制代码

```tsx
export const Form = React.forwardRef<HTMLFormElement, FormProps>(
  (props, ref) => {
    return <FormImpl {...props} ref={ref} />;
  }
);

const FormImpl = React.forwardRef<HTMLFormElement, FormImplProps>(
  (
    {
      reloadDocument,
      replace,
      method = defaultMethod,
      action,
      onSubmit,
      fetcherKey,
      routeId,
      relative,
      ...props
    },
    forwardedRef
  ) => {
```

```
    let submit = useSubmitImpl(fetcherKey, routeId);
    let formMethod: FormMethod =
      method.toLowerCase() === "get" ? "get" : "post";
    let formAction = useFormAction(action, { relative });
    let submitHandler: React.FormEventHandler<HTMLFormElement> = (event) => {
      onSubmit && onSubmit(event);
      if (event.defaultPrevented) return;
      event.preventDefault();

      let submitter = (event as unknown as HTMLSubmitEvent).nativeEvent
        .submitter as HTMLFormSubmitter | null;

      submit(submitter || event.currentTarget, { method, replace, relative });
    };

    return (
      <form
        ref={forwardedRef}
        method={formMethod}
        action={formAction}
        onSubmit={reloadDocument ? onSubmit : submitHandler}
        {...props}
      />
    );
  }
);
```

底层使用 form 元素实现， form 的属性 `method/action/submit` 提交函数都是通过封装的钩子函数。也就是说路由与 form 表单有绑定关系，好处是能够在路由跳转前后更好的处理数据。

小结