React Router V6.4 （Router 对象篇）

history 的不足

在 history 篇中，我们知道了 history 封装三种模式 history:

- browserHistory
- hashHistory
- memoeryHistory

但是实际上它完成的在单个 url 层面的管理。

在实际中项目中，可能有很多的 url 要跳转要管理，一个 history 明显就不够用（或者难以维护）。Router 的概念就被创造了出来，Router 其实就是一个管理 Route 路由的“表”，方便我们书写管理路由代码。

Router 对象使用

可以使用 vite 创建一个项目，然后进行快速 api 测试，使用 vitest 或者 Jest 进行前端单元测试(测试很重要，能快速验证我们猜测，保证代码质量，尤其像 React Router 这种需要高质量

代码的库)

- 安装

```sh 复制代码
pnpm i @remix-run/router
```

Router 类型

```ts 复制代码
export interface Router {
  get basename(): RouterInit["basename"];
  get state(): RouterState;
  get routes(): AgnosticDataRouteObject[];
  initialize(): Router;
  subscribe(fn: RouterSubscriber): () => void;
  enableScrollRestoration(
    savedScrollPositions: Record<string, number>,
    getScrollPosition: GetScrollPositionFunction,
    getKey?: GetScrollRestorationKeyFunction
  ): () => void;
  navigate(to: number): void;
  navigate(to: To, opts?: RouterNavigateOptions): void;
  fetch(
    key: string,
    routeId: string,
    href: string,
    opts?: RouterNavigateOptions
  ): void;
  revalidate(): void;
  createHref(location: Location | URL): string;
  encodeLocation(to: To): Path;
  getFetcher<TData = any>(key?: string): Fetcher<TData>;
  deleteFetcher(key?: string): void;
  dispose(): void;
  _internalFetchControllers: Map<string, AbortController>;
  _internalActiveDeferreds: Map<string, DeferredData>;
}
```

Route 类型

```ts 复制代码
export type AgnosticDataRouteObject =
  | AgnosticDataIndexRouteObject
  | AgnosticDataNonIndexRouteObject;
export type AgnosticDataIndexRouteObject = AgnosticIndexRouteObject & {
```

```ts
  id: string;
};
export type AgnosticIndexRouteObject = AgnosticBaseRouteObject & {
  children?: undefined;
  index: true;
};
type AgnosticBaseRouteObject = {
  caseSensitive?: boolean;
  path?: string;
  id?: string;
  loader?: LoaderFunction;
  action?: ActionFunction;
  hasErrorBoundary?: boolean;
  shouldRevalidate?: ShouldRevalidateFunction;
  handle?: any;
};
```

如果读者使用过 react-router-dom 发现 Route 类型中没有 element, 这并不属于 Router 中的内容， element 是 React 中的特性，属于 React-Router。

创建 Router 工厂函数

ts 复制代码

```ts
export function createRouter(init: RouterInit): Router {}
```

createRouter 接受 init 作为参数，返回一个 Router. 下面是 RouterInit 的类型

ts 复制代码

```ts
export interface RouterInit {
  basename?: string;
  routes: AgnosticRouteObject[];
  history: History;
  hydrationData?: HydrationState;
}
```

创建一个 router

- 使用 browserHistory 和 routes 就可以使用工厂函数初始化一个 router

tsx 复制代码

```tsx
import { createRouter, createBrowserHistory } from "@remix-run/router";

const router = createRouter({
  history: createBrowserHistory(),
  routes: [
```

```
    {
      path: "/",
    },
    {
      path: "/about"
    },
  ],
}).initialize();

router.navigate("/base") // 如果在浏览器环境中，地址就会改变 /base。但是没有刷新页面
```

从 history 函数参数可以得知，router 也是可以支持多种模式: hash/browser/memory

createRouter 内部重要内容

1. 使用 convertRoutesToDataRoutes 将 init.routes 转换成 dataRoutes

在转换之前要明白两个概念：

- indexRoute
- PathOrLayoutRoute

convertRoutesToDataRoutes 主要作用就是加上 id (indexRoute) 或者 id+children(PathOrLayoutRoute)

- 判断 indexRoute

ts 复制代码

```ts
function isIndexRoute(
  route: AgnosticRouteObject
): route is AgnosticIndexRouteObject {
  return route.index === true;
}
```

- convertRoutesToDataRoutes 的内部实现

ts 复制代码

```ts
if (isIndexRoute(route)) {
  let indexRoute: AgnosticDataIndexRouteObject = { ...route, id };
  return indexRoute;
} else {
  let pathOrLayoutRoute: AgnosticDataNonIndexRouteObject = {
    ...route,
    id,
```

```
    children: route.children
      ? convertRoutesToDataRoutes(route.children, treePath, allIds)
      : undefined,
  };
  return pathOrLayoutRoute;
}
```

router 转换完成之前，会初始化一些属性

- unlistenHistory
- subscribers
- savedScrollPositions
- getScrollRestorationKey
- getScrollPosition
- initialScrollRestored

2. 初始化 match 对象 initialMatches

- 先看 match 的类型定义

ts 复制代码

```ts
export interface AgnosticRouteMatch<
  ParamKey extends string = string,
  RouteObjectType extends AgnosticRouteObject = AgnosticRouteObject
> {
  params: Params<ParamKey>;
  pathname: string;
  pathnameBase: string;
  route: RouteObjectType;
}
```

- matchRoutes 匹配 routes 函数

ts 复制代码

```ts
export function matchRoutes(
  routes: RouteObjectType[],
  locationArg: Partial<Location> | string,
  basename = "/"
){return match}

let initialMatches = matchRoutes(
    dataRoutes,
    init.history.location,
    init.basename
);
```

matchRoutes 内部实现

```ts
let branches = flattenRoutes(routes);

let matches = null;
  for (let i = 0; matches == null && i < branches.length; ++i) {
    matches = matchRouteBranch<string, RouteObjectType>(
      branches[i],
      safelyDecodeURI(pathname)
    );
  }

  return matches;
```

- RouteBranch 类型

```ts
interface RouteBranch<
  RouteObjectType extends AgnosticRouteObject = AgnosticRouteObject
> {
  path: string;
  score: number;
  routesMeta: RouteMeta<RouteObjectType>[];
}
```

- matchRouteBranch

```ts
function matchRouteBranch<
  ParamKey extends string = string,
  RouteObjectType extends AgnosticRouteObject = AgnosticRouteObject
>(
  branch: RouteBranch<RouteObjectType>,
  pathname: string
): AgnosticRouteMatch<ParamKey, RouteObjectType>[] | null {
    let matches: AgnosticRouteMatch<ParamKey, RouteObjectType>[] = [];
    let match = matchPath(
      { path: meta.relativePath, caseSensitive: meta.caseSensitive, end },
      remainingPathname
    );

    /*...*/
    matches.push({
      params: matchedParams as Params<ParamKey>,
      pathname: joinPaths([matchedPathname, match.pathname]),
      pathnameBase: normalizePathname(
        joinPaths([matchedPathname, match.pathnameBase])
      ),
```

```
      route,
    });

    eturn matches;
}
```

这里 initialMatches 中 match 对象

定义 router.state 对象（注意：这里的 state 是比较复杂和重要）

ts 复制代码

```ts
let state: RouterState = {
    historyAction: init.history.action, // history 中携带的 action
    location: init.history.location, // history 中携带的 location
    matches: initialMatches, // 上面已经涉及了 initialMatches 对象
    initialized,
    navigation: IDLE_NAVIGATION,
    restoreScrollPosition: null,
    preventScrollReset: false,
    revalidation: "idle",
    loaderData: (init.hydrationData && init.hydrationData.loaderData) || {},
    actionData: (init.hydrationData && init.hydrationData.actionData) || null,
    errors: (init.hydrationData && init.hydrationData.errors) || initialErrors,
    fetchers: new Map(),
  };
```

其中 initizalized

ts 复制代码

```ts
let initialized =
    !initialMatches.some((m) => m.route.loader) || init.hydrationData != null;
```

- 定义 initialize 函数

ts 复制代码

```ts
function initialize() {
    // If history informs us of a POP navigation, start the navigation but do not update
    // state.  We'll update our own state once the navigation completes
    unlistenHistory = init.history.listen(
      ({ action: historyAction, location }) =>
        startNavigation(historyAction, location)
    );

    // Kick off initial data load if needed.  Use Pop to avoid modifying history
    if (!state.initialized) {
      startNavigation(HistoryAction.Pop, state.location);
```

```
    }

    return router;
}
```

返回 router 对象，监听 并开始导航

定义订阅函数 subscribe

实现很简单，在 subscribe Map 数据结构中添加函数，然后返回一个包含删除订阅的函数

ts 复制代码

```ts
// Subscribe to state updates for the router
  function subscribe(fn: RouterSubscriber) {
    subscribers.add(fn);
    return () => subscribers.delete(fn);
  }
```

定义 enableScrollRestoration 函数

ts 复制代码

```ts
function enableScrollRestoration(
    positions: Record<string, number>,
    getPosition: GetScrollPositionFunction,
    getKey?: GetScrollRestorationKeyFunction
) {
    savedScrollPositions = positions;
    getScrollPosition = getPosition;
    getScrollRestorationKey = getKey || ((location) => location.key);

    if (!initialScrollRestored && state.navigation === IDLE_NAVIGATION) {
      initialScrollRestored = true;
      let y = getSavedScrollPosition(state.location, state.matches);
      if (y != null) {
        updateState({ restoreScrollPosition: y });
      }
    }

    return () => {
      savedScrollPositions = null;
      getScrollPosition = null;
      getScrollRestorationKey = null;
    };
 }
```

```
 滚动本质是修改 history 的 state
function updateState(newState: Partial<RouterState>): void {
    state = {
      ...state,
      ...newState,
    };
    subscribers.forEach((subscriber) => subscriber(state));
}
```

定义 navigate 函数（用于修改 url，和后续跳转）

- number 模式，本质是 go 方法
- 对象模式

ts 复制代码

```ts
async function navigate(
    to: number | To,
    opts?: RouterNavigateOptions
  ): Promise<void> {
    if (typeof to === "number") {
      init.history.go(to);
      return;
    }

    let { path, submission, error } = normalizeNavigateOptions(to, opts);

    let location = createLocation(state.location, path, opts && opts.state);
    location = {
      ...location,
      ...init.history.encodeLocation(location),
    };

    let historyAction =
      (opts && opts.replace) === true || submission != null
        ? HistoryAction.Replace
        : HistoryAction.Push;
    let preventScrollReset =
      opts && "preventScrollReset" in opts
        ? opts.preventScrollReset === true
        : undefined;

    return await startNavigation(historyAction, location, {
      submission,
      pendingError: error,
      preventScrollReset,
      replace: opts && opts.replace,
    });
  }
```

对象模式下开始导航 startNavigation 函数/completeNavigation 函数

startNavigation 为 completeNavigation 函数所需要的属性：

- location
- options
  - matches
  - loaderData
  - errors

导航的本质根据不同的条件，调用 push/replace

```ts
function completeNavigation(
    location: Location,
    newState: Partial<Omit<RouterState, "action" | "location" | "navigation">>
  ): void {
      // 更新
      updateState({/**/})
      // 本质
      if (isUninterruptedRevalidation) {
    } else if (pendingAction === HistoryAction.Pop) {
    } else if (pendingAction === HistoryAction.Push) {
      init.history.push(location, location.state);
    } else if (pendingAction === HistoryAction.Replace) {
      init.history.replace(location, location.state);
    }
    // 后面是重置数据
  }
```

navigate 的用法:

```ts
router.navigate("/about-page");
router.navigate("/home", { replace: true });
router.navigate(-1);

let formData = new FormData();
formData.append("name", "m");
router.navigate("/about", {
  formMethod: "post",
  formData,
});
```

定义 fetch 函数

什么是 fetch

fetch 是一种不触发导航情况下调用 loader加载器/action 操作的机制

```ts
// Execute the loader for /page
router.fetch("key", "/page");

// Submit to the action for /page
let formData = new FormData();
formData.append(key, value);
router.fetch("key", "/page", {
  formMethod: "post",
  formData,
});
```

fetch 函数将 createRouter 中 state 对象 fetcher Map 数据结构保存到 history.state 中

fetcher 类型

```ts
type FetcherStates<TData = any> = {
  Idle: {
    state: "idle";
    formMethod: undefined;
    formAction: undefined;
    formEncType: undefined;
    formData: undefined;
    data: TData | undefined;
  };
  Loading: {
    state: "loading";
    formMethod: FormMethod | undefined;
    formAction: string | undefined;
    formEncType: FormEncType | undefined;
    formData: FormData | undefined;
    data: TData | undefined;
  };
  Submitting: {
    state: "submitting";
    formMethod: FormMethod;
    formAction: string;
    formEncType: FormEncType;
    formData: FormData;
```

```
    data: TData | undefined;
  };
};

export type Fetcher<TData = any> =
  FetcherStates<TData>[keyof FetcherStates<TData>];
```

获取 fetcher

ts 复制代码

```
function getFetcher<TData = any>(key: string): Fetcher<TData> {
    return state.fetchers.get(key) || IDLE_FETCHER;
}
```

- loadingFetcher

ts 复制代码

```
let loadingFetcher: FetcherStates["Loading"] = {
  state: "loading",
  formMethod: undefined,
  formAction: undefined,
  formEncType: undefined,
  formData: undefined,
  data: existingFetcher && existingFetcher.data,
};
state.fetchers.set(key, loadingFetcher);
updateState({ fetchers: new Map(state.fetchers) });
```

- doneFetcher

ts 复制代码

```
 let doneFetcher: FetcherStates["Idle"] = {
  state: "idle",
  data: result.data,
  formMethod: undefined,
  formAction: undefined,
  formEncType: undefined,
  formData: undefined,
};
state.fetchers.set(key, doneFetcher);
updateState({ fetchers: new Map(state.fetchers) });
```

定义 getFetcher

```ts
function getFetcher<TData = any>(key: string): Fetcher<TData> {
    return state.fetchers.get(key) || IDLE_FETCHER;
}
```

定义 deleteFetcher

```ts
function deleteFetcher(key: string): void {
    if (fetchControllers.has(key)) abortFetcher(key);
    fetchLoadMatches.delete(key);
    fetchReloadIds.delete(key);
    fetchRedirectIds.delete(key);
    state.fetchers.delete(key);
}
```

定义 revalidate

重新验证所有当前的所有 loader

```ts
function revalidate() {
    interruptActiveLoads();
    updateState({ revalidation: "loading" });

    if (state.navigation.state === "submitting") {
      return;
    }

    if (state.navigation.state === "idle") {
      startNavigation(state.historyAction, state.location, {
        startUninterruptedRevalidation: true,
      });
      return;
    }

    startNavigation(
      pendingAction || state.historyAction,
      state.navigation.location,
      { overrideNavigation: state.navigation }
    );
  }
```

定义 createHref

```ts 复制代码
(to: To) => init.history.createHref(to)
```

定义 encodeLocation

```ts 复制代码
(to: To) => init.history.encodeLocation(to)
```

定义 dispose

清除

```ts 复制代码
function dispose() {
    if (unlistenHistory) {
      unlistenHistory();
    }
    subscribers.clear();
    pendingNavigationController && pendingNavigationController.abort();
    state.fetchers.forEach((_, key) => deleteFetcher(key));
}
```

Jest 测试

Router 包使用了 Jest 进行测试，在前面的文章里面已经说明了测试最好使用 memoryHistory 模式 history 和 router 进行测试，或者集成 dom 环境。

如果想要更加深入理解 Router 对象，不妨阅读测试文件。可以自己写这些测试用例，提供自己测试能力和学习源码的理解能力。

小结

Router 是一个复杂的对象，Router 建立在 history 对象之上支持多种模式）。重要的概念包括：

- routes （路由表）
- state (state 对象的操作，包括做滚动存储)
- navigate(跳转的功能)

- fetch (fetch 无跳转情况下操作 loaders/actions)

有了 Router 的基础下就是深入理解 Router 和测试用例（这一章要自己动手实践，充分理解才能进入 React-Router 阶段）和 React 层面的封装（组件，hooks）