21 | 从 React-Router 切入，系统学习前端路由解决方案

2020/12/21 修言

React-Router 是 React 场景下的路由解决方案，本讲我们将学习 React-Router 的实现机制，并基于此提取和探讨通用的前端路由解决方案。

> 注：没有使用过 React-Router 的同学，可以点击这里完成快速上手。

认识 React-Router

本着尽快进入主题的原则，这里我用一个尽可能简单的 Demo 作为引子，帮助你快速地把握 React-Router 的核心功能。请看下面代码（解析在注释里）：

复制代码

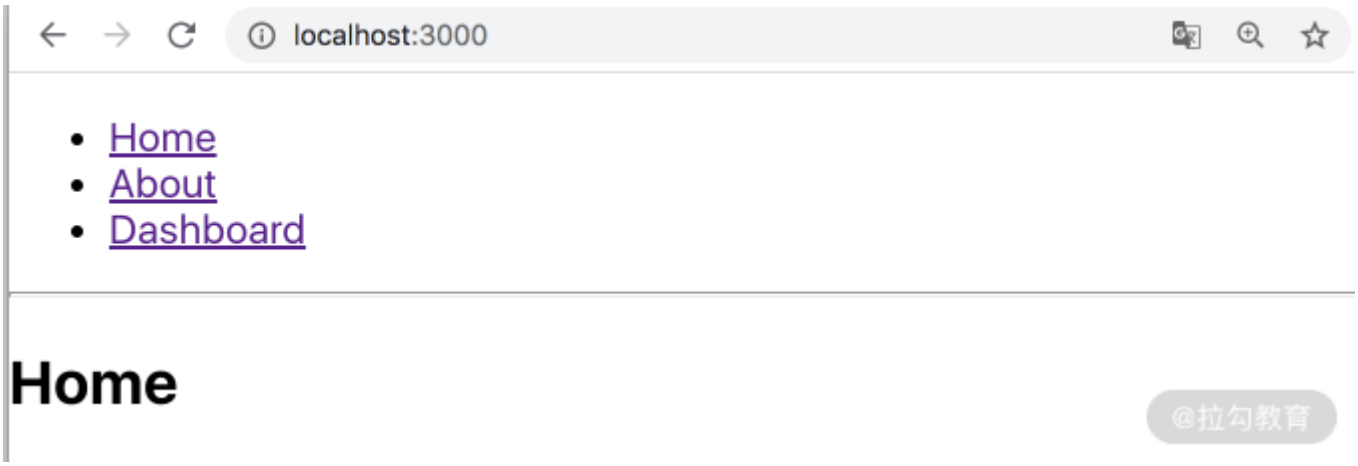
```
import React from "react";
// 引入 React-Router 中的相关组件
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
// 导出目标组件
const BasicExample = () => (
  // 组件最外层用 Router 包裹
  <Router>
    <div>
      <ul>
        <li>
          // 具体的标签用 Link 包裹
          <Link to="/">Home</Link>
        </li>
        <li>
          // 具体的标签用 Link 包裹
          <Link to="/about">About</Link>
        </li>
        <li>
          // 具体的标签用 Link 包裹
          <Link to="/dashboard">Dashboard</Link>
        </li>
      </ul>
      <hr />

      // Route 是用于声明路由映射到应用程序的组件层
      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/dashboard" component={Dashboard} />
    </div>
  </Router>
```

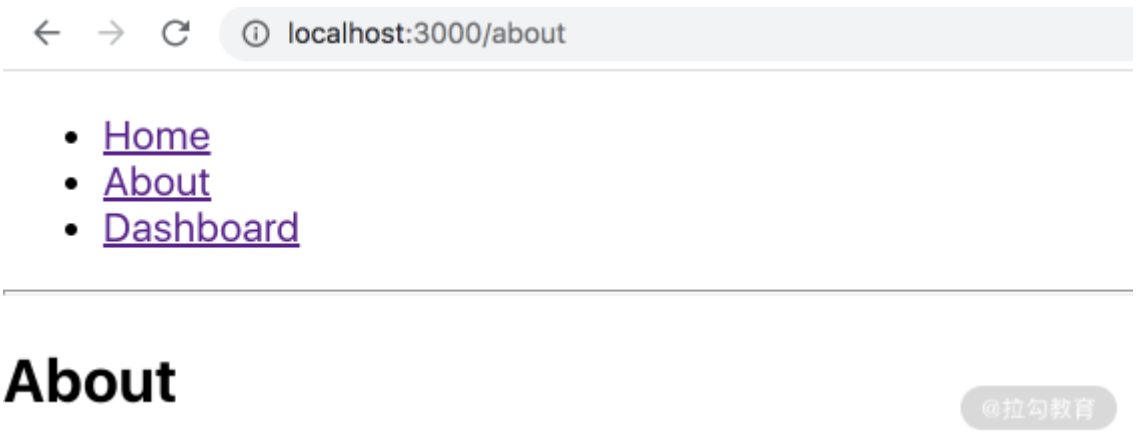
```
);
// Home 组件的定义
const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);
// About 组件的定义
const About = () => (
  <div>
    <h2>About</h2>
  </div>
);
// Dashboard 的定义
const Dashboard = () => (
  <div>
    <h2>Dashboard</h2>
  </div>
);

export default BasicExample;
```

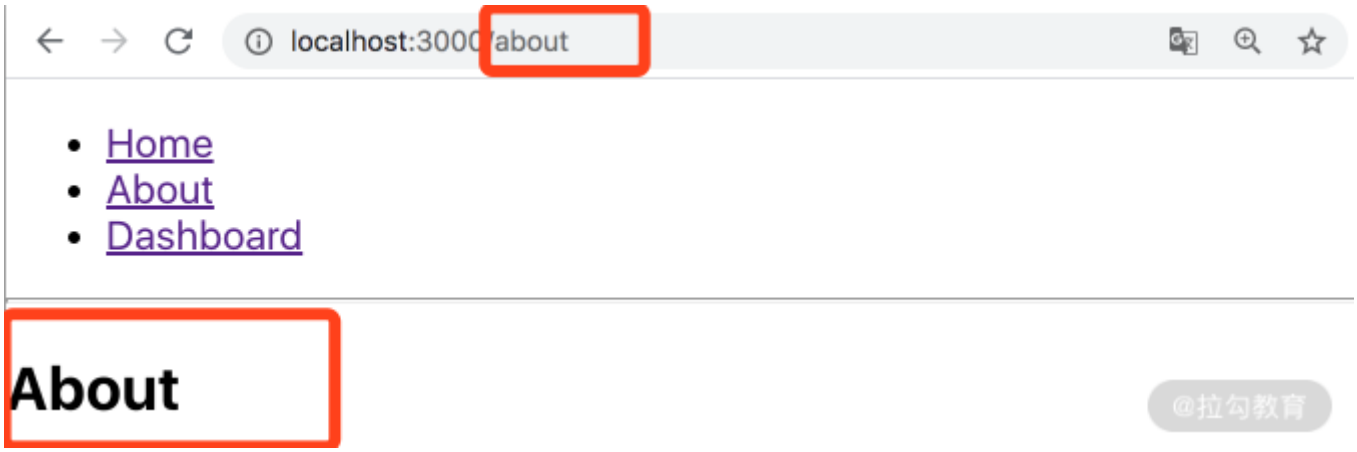
这个 Demo 渲染出的页面效果如下图所示：



当我点击不同的链接时，ul 元素内部就会展示不同的组件内容。比如当我点击“About”链接时，就会展示 About 组件的内容，效果如下图所示：



注意，点击 About 后，界面中发生变化的地方有两处（见下图标红处），除了 ul 元素的内容改变了之外，路由信息也改变了。



在 React-Router 中，各种细碎的功能点有不少，但作为 React 框架的前端路由解决方案，它最基本也是最核心的能力，其实正是你刚刚所见到的这一幕——**路由的跳转**。这也是我们接下来讨论的重点。

接下来我们就结合 React-Router 的源码，一起来看看“跳转”这个动作是如何实现的。

React-Router 是如何实现路由跳转的?

首先需要回顾下 Demo 中的第一行代码：

```
1. import { BrowserRouter as Router, Route, Link } from "react-router-dom";
```

复制代码

这行代码告诉我们，为了实现一个简单的路由跳转效果，一共从 React-Router 中引入了以下 3 个组件：

- BrowserRouter
- Route
- Link

这 3 个组件也就代表了 React-Router 中的 3 个核心角色：

- **路由器**，比如 BrowserRouter 和 HashRouter
- **路由**，比如 Route 和 Switch
- **导航**，比如 Link、NavLink、Redirect

路由（以 Route 为代表）负责定义路径与组件之间的映射关系，而导航（以 Link 为代表）负责触发路径的改变，路由器（包括 BrowserRouter 和 HashRouter）则会根据 Route 定义出来的映射关系，为新的路径匹配它对应的逻辑。

以上便是 3 个角色“打配合”的过程。这其中，最需要你注意的是路由器这个角色，React Router 曾在说明文档中官宣它是“React Router 应用程序的核心”。因此学习 React Router，最要紧的是搞明白路由器的工作机制。

路由器：BrowserRouter 和 HashRouter

路由器负责感知路由的变化并作出反应，它是整个路由系统中最为重要的一环。React-Router 支持我们使用 hash（对应 HashRouter）和 browser（对应 BrowserRouter） 两种路由规则，这里我们把两种规则都讲一下。

HashRouter、BrowserRouter，这俩人名字这么像，该不会底层逻辑区别也不大吧？别说，还真是如此。我们首先来瞟一眼 HashRouter 的源码：

```
1  import React from "react";
2  import { Router } from "react-router";
3  import { createHashHistory as createHistory } from "history";
4  import PropTypes from "prop-types";
5  import warning from "tiny-warning";
6
7  /**
8   * The public API for a <Router> that uses window.location.hash.
9   */
10 class HashRouter extends React.Component {
11   history = createHistory(this.props);
12
13   render() {
14     return <Router history={this.history} children={this.props.children} />;
15   }
16 }
```


再瞟一眼 BrowserRouter 的源码：

```
import React from "react";
import { Router } from "react-router";
import { createBrowserHistory as createHistory } from "history";
import PropTypes from "prop-types";
import warning from "tiny-warning";

/**
 * The public API for a <Router> that uses HTML5 history.
 */
class BrowserRouter extends React.Component {
  history = createHistory(this.props);

  render() {
    return <Router history={this.history} children={this.props.children} />;
  }
}
```


我们会发现这两个文件惊人的相似，而最关键的区别我也已经在图中分别标出，即它们调用的 history 实例化方法不同：HashRouter 调用了 createHashHistory，BrowserRouter 调用了 createBrowserHistory。

这两个 history 的实例化方法均来源于 history 这个独立的代码库，关于它的实现细节，你倒不必纠结。对于 createHashHistory 和 createBrowserHistory 这两个 API，我们最要紧的是掌握它们各自的特征。

- `createBrowserHistory`：它将在浏览器中使用 HTML5 history API 来处理 URL（见下图标红处的说明），它能够处理形如这样的 URL，example.com/some/path。由此可得，**BrowserRouter 是使用 HTML 5 的 history API 来控制路由跳转的。**

```
/**
 * Creates a history object that uses the HTML5 history API including
 * pushState, replaceState, and the popstate event.
 */
const createBrowserHistory = (props = {}) => {
  invariant(
    canUseDOM,
    'Browser history needs a DOM'
  )

  const globalHistory = window.history
  const canUseHistory = supportsHistory()
  const needsHashChangeListener = !supportsPopStateOnHashChange()

  const {
    forceRefresh = false,
    getUserConfirmation = getConfirmation,
    keyLength = 6
  } = props
  const basename = props.basename ? stripTrailingSlash(addLeadingSlash(props.basename)) : ''
```


- `createHashHistory`：它是使用 hash tag (#) 处理 URL 的方法，能够处理形如这样的 URL，example.com/#/some/path。我们可以看到它的源码中对各种方法的定义基本都围绕 hash 展开（如

下图所示），由此可得，**HashRouter 是通过 URL 的 hash 属性来控制路由跳转的。**

```
const getHashPath = () => {
  // We can't use window.location.hash here because it's not
  // consistent across browsers - Firefox will pre-decode it!
  const href = window.location.href
  const hashIndex = href.indexOf('#')
  return hashIndex === -1 ? '' : href.substring(hashIndex + 1)
}

const pushHashPath = (path) =>
  window.location.hash = path

const replaceHashPath = (path) => {
  const hashIndex = window.location.href.indexOf('#')

  window.location.replace(
    window.location.href.slice(0, hashIndex >= 0 ? hashIndex : 0) + '#' + path
  )
}

const createHashHistory = (props = {}) => {
  invariant(
    canUseDOM,
    'Hash history needs a DOM'
  )
```


> 注：关于 hash 和 history 这两种模式，我们在下文中还会持续探讨。

现在，见识了表面现象，了解了背后机制。我们不妨回到故事的原点，再多问自己一个问题：为什么我们需要 React-Router?

或者把这个问题稍微拔高一点：**为什么我们需要前端路由**?

这一切的一切，都要从很久以前说起。

理解前端路由——是什么？解决什么问题?

背景——问题的产生

在前端技术早期，一个 URL 对应一个页面，如果你要从 A 页面切换到 B 页面，那么必然伴随着页面的刷新。这个体验并不好，不过在最初也是无奈之举——毕竟用户只有在刷新页面的情况下，才可以重新去请求数据。

后来，改变发生了——Ajax 出现了，它允许人们在不刷新页面的情况下发起请求；与之共生的，还有“不刷新页面即可更新页面内容”这种需求。在这样的背景下，出现了**SPA（单页面应用）**。

SPA 极大地提升了用户体验，它允许页面在不刷新的情况下更新页面内容，使内容的切换更加流畅。但是在 SPA 诞生之初，人们并没有考虑到“定位”这个问题——在内容切换前后，页面的 URL 都是一样的，这就带来了两个问题：

- SPA 其实并不知道当前的页面“进展到了哪一步”，可能你在一个站点下经过了反复的“前进”才终于唤出了某一块内容，但是此时只要刷新一下页面，一切就会被清零，你必须重复之前的操作才可以重新对内容进行定位——SPA 并不会“记住”你的操作；
- 由于有且仅有一个 URL 给页面做映射，这对 SEO 也不够友好，搜索引擎无法收集全面的信息。

为了解决这个问题，前端路由出现了。

前端路由——SPA“定位”解决方案

前端路由可以帮助我们在仅有一个页面的情况下，“记住”用户当前走到了哪一步——为 SPA 中的各个视图匹配一个唯一标识。这意味着用户前进、后退触发的新内容，都会映射到不同的 URL 上去。此时即便他刷新页面，因为当前的 URL 可以标识出他所处的位置，因此内容也不会丢失。

那么如何实现这个目的呢？首先我们要解决以下两个问题。

- 当用户刷新页面时，浏览器会默认根据当前 URL 对资源进行重新定位（发送请求）。这个动作对 SPA 是不必要的，因为 SPA 作为单页面，无论如何也只会有一个资源与之对应。此时若走正常的请求-刷新流程，反而会使用户的前进后退操作无法被记录。
- 单页面应用对服务端来说，就是一个 URL、一套资源，那么如何做到用“不同的 URL”来映射不同的视图内容呢？

从这两个问题来看，服务端已经救不了 SPA 这个场景了。所以要靠咱们前端自力更生，不然怎么叫“前端路由”呢？作为前端，我们可以提供以下这样的解决思路。

- **拦截用户的刷新操作，避免服务端盲目响应、返回不符合预期的资源内容**，把刷新这个动作完全放到前端逻辑里消化掉；
- **感知 URL 的变化**。这里不是说要改造 URL、凭空制造出 N 个 URL 来。而是说 URL 还是那个 URL，只不过我们可以给它做一些微小的处理，这些处理并不会影响 URL 本身的性质，不会影响服务器对

它的识别，只有我们前端能感知到。一旦我们感知到了，我们就根据这些变化、用 JS 去给它生成不同的内容。

实践思路——hash 与 history

接下来重点就来了，现在前端界对前端路由有哪些实现思路？这里需要掌握的两个实践就是 hash 与 history。

hash 模式

hash 模式是指通过改变 URL 后面以“#”分隔的字符串（这货其实就是 URL 上的哈希值），从而让页面感知到路由变化的一种实现方式。举个例子，比如这样的一个 URL：

```
1. https://www.imooc.com/
```

复制代码

我就可以通过增加和改变哈希值，来让这个 URL 变得有那么一点点不一样：

```
1. // 主页
2. https://www.imooc.com/#index
3. // 活动页
4. https://www.imooc.com/#activePage
```

复制代码

这个“不一样”是前端完全可感知的——JS 可以帮我们捕获到哈希值的内容。在 hash 模式下，我们实现路由的思路可以概括如下：

（1）hash 的改变：我们可以通过 location 暴露出来的属性，直接去修改当前 URL 的 hash 值：

```
1. window.location.hash = 'index';
```

复制代码

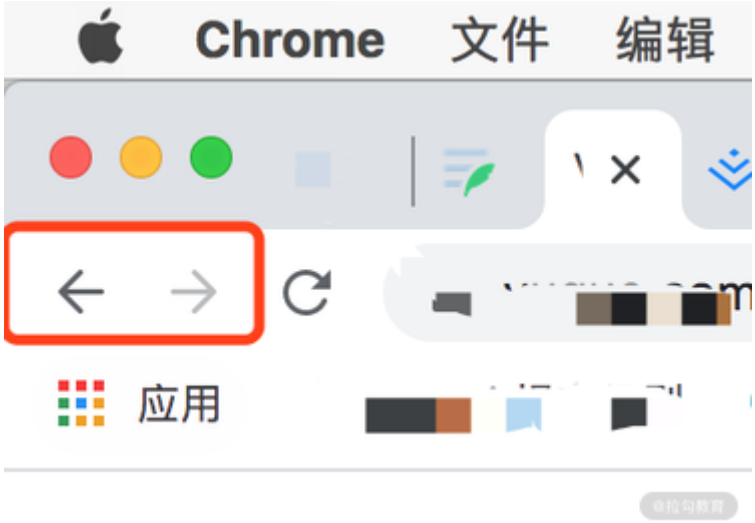
（2）hash 的感知：通过监听 “hashchange”事件，可以用 JS 来捕捉 hash 值的变化，进而决定我们页面内容是否需要更新：

```
1. // 监听hash变化，点击浏览器的前进后退会触发
2. window.addEventListener('hashchange', function(event){
3.     // 根据 hash 的变化更新内容
4. },false)
```

复制代码

history 模式

大家知道，在我们浏览器的左上角，往往有这样的操作点：



通过点击前进后退箭头，就可以实现页面间的跳转。这样的行为，其实是可以通过 API 来实现的。

浏览器的 history API 赋予了我们这样的能力，在 HTML 4 时，就可以通过下面的接口来操作浏览历史、实现跳转动作：

```
window.history.forward()  // 前进到下一页
```

复制代码

```
window.history.back() // 后退到上一页
```

复制代码

```
window.history.go(2) // 前进两页
```

复制代码

```
window.history.go(-2) // 后退两页
```

复制代码

很有趣吧？遗憾的是，在这个阶段，我们能做的只是“切换”，而不能“改变”。好在从 HTML 5 开始，浏览器支持了 pushState 和 replaceState 两个 API，允许我们对浏览历史进行修改和新增：

```
history.pushState(data[,title][,url]); // 向浏览历史中追加一条记录
```

复制代码

```
history.replaceState(data[,title][,url]); // 修改（替换）当前页在浏览历史中的信息
```

复制代码

这样一来，修改动作就齐活了。

有修改，就要有对修改的感知能力。在 history 模式下，我们可以通过监听 popstate 事件来达到我们的目的：

```
window.addEventListener('popstate', function(e) {
  console.log(e)
});
```

复制代码

每当浏览历史发生变化，popstate 事件都会被触发。

注：go、forward 和 back 等方法的调用确实会触发 popstate，但是**pushState 和 replaceState 不会**。不过这一点问题不大，我们可以通过自定义事件和全局事件总线来手动触发事件。

总结

本讲我们以 React-Router 为切入点，结合源码剖析了 React-Router 中“跳转”这一动作的实现原理，由此牵出了针对“前端路由方案”这个知识点相对系统的探讨。行文至此，React 周边生态所涉及的重难点知识，相信已经深深地烙印在了你的脑海里。

下一讲开始，我们将围绕“React 设计模式与最佳实践”以及“React 性能优化”两条主线展开学习。彼时，站在“生产实践”这个全新的视角去认识 React 后，相信各位对它的理解定会更上一层楼。大家加油！