# react-router 路由及状态同构

## 前端路由原理及表现

当我们说前端路由，也就是客户端路由时，更多的是在说当 url 变化时，不会使当前的变化触发到后端而是在前端内部处理。

早期的浏览器我们可以使用 hash 来做路由标识，使用 hash 的好处也非常明显：

- hash 有非常好的兼容性，大部分老浏览器都支持 hash 的形式
- 通过监听 onhashchange 可以判断路由变化，从而判断需要渲染的组件
- hash 默认不会发送到后端

HTML5 之后新增了 history 相关的 API，我们可以通过 pushState 和 replaceState 来控制 url 的变化，同时处理组件的变化

- history 相关的 api 改变的是 url 中的 pathname，刷

新后会将这部分 url 发送到后端去

大部分前端路由项目都是使用这两种方式来实现，这类使用前端路由的应用我们可以统称为 spa 单页应用，因为这类应用使用起来就像是只有一个页面一样，后续的页面都在前端进行内容切换，不会有服务端路由页面闪动的情况。

# react-router 详解

前面我们讲解了路由匹配的原理，对于 react 的 react-router 来说，我们只是需要把前面讲到的内容融合进具体的代码，为了让同学们能更好的理解，在这里把之前讲解的部分和 react 组件融合起来。

对于 react-router 的使用，react router 主要实现了两个组件，一个是 `Route` 组件，另一个是 `Link` 组件。

`Route` 组件主要是定义了规定的路由渲染的组件，`Link` 组件主要是渲染的 a 标签进行跳转。

主要需要以下三点内容：

- Route 组件监听 url 改变，确定渲染的子组件
- Link 组件触发 url 的变化

```
export class Route extends React.Component
{
  componentWillMount() {
    const unlisten =
history.listen((location, action) => {
      console.log('history change listen',
location, action);
      this.forceUpdate();
    });
    this.setState({unlisten});
  }

  componentWillUnmount() {
    const {unlisten} = this.state;
    unlisten();
  }

  render() {
    const {
      render,
      path,
      component: ChildComponent,
    } = this.props;

    const match = matchPath(path);
```

```
    const matchResult =
match(window.location.pathname);

    if (!matchResult) return null;

    if (ChildComponent) {
      return (<ChildComponent match=
{matchResult} />);
    }

    if (render && typeof render ===
'function') {
      return render({matchResult});
    }

    return null;
  }
}
```

Link 组件：

```
export class Link extends React.Component {
  handleClick = (e) => {
    const {
      replace,
      to,
```

```
    } = this.props;
    e.preventDefault();
    replace ? history.replace(to) :
history.push(to);
  }

  render() {
    const {
      to,
      children,
    } = this.props;

    return (
      <a href={to} onClick=
{this.handleClick}>{children}</a>
    );
  }
}
```

```
import React from 'react';
import { createBrowserHistory } from
'history';
import { match as matchPath } from 'path-
to-regexp';

const history = createBrowserHistory();
```

我们主要使用 history 这个库，来对我们的路由进行统一封装，它的内部帮助我们处理了第一小节讲的三种情况：

- hashHistory：hash 的路由
- browserHistory：浏览器 HTML5 中 history 的路由
- memoryHistory：内存里自己记录一下路由情况

同时我们也使用了 path-to-regexp 这个库，来对我们的路径进行处理，由它来确认我们当前的路由是否是符合定义的路由格式。

# 服务端渲染及同构

前面我们学到过，对于服务端渲染和客户端渲染来说，虽然最终显示页面的结果都相同，但是对于不同方式来说，他们的渲染过程不尽相同。

react 的服务端渲染相对来说就简单一些，我们只需要使用 ReactDOM 包中的 server 部分模块进行即可，这里我们启动一个 node.js 服务器来做这件事情。

在渲染的结果中，我们能够看到 react 在标签中增加了一些 hash 值，他的作用是在客户端渲染时，如果 vdom 对比的结果与之相同，则不需要通过客户端方法进行挂载，减少客户端渲染时的压力。

我们可以使用很多框架来实现这种同构效果，例如 next.js 来达到一个同构的效果，需要注意的是它内部会使用一些 server 端的内容。所以在写法上我们需要额外注意一些。

```
import React from 'react'

export default class extends
React.Component {
  static async getInitialProps({ req }) {
    const userAgent = req ?
req.headers['user-agent'] :
navigator.userAgent
    return { userAgent }
  }

  render() {
    return (
      <div>
        Hello World {this.props.userAgent}
      </div>
    )
  }
}
```

比如这就是一个使用 next 运行的例子，next 的应用中会有 `getInitialProps` 这个方法，他的作用就是对应用初始值进行获取。当然这里针对不同的环境，这个 next 应用的生命周期的表现也完全不同。

服务端有的参数内容：

- req: HTTP请求对象
- res: HTTP响应对象
- pathname: URL中的路径部分
- query：URL中的查询字符串部分解析出的对象
- err：错误对象，如果在渲染时发生了错误

客户端渲染有的参数内容：

- xhr：XMLHttpRequest对象（客户端渲染独有）
- pathname: URL中的路径部分
- query：URL中的查询字符串部分解析出的对象
- err：错误对象，如果在渲染时发生了错误

所以我们可以在这个方法中通过判断参数是否存在，来判断出当前执行的环境是服务端还是客户端。

但这样的写法对我们的应用入侵比较大，因为我们需要直接改造我们的 react 应用，后期也不好迁移至其他服务端框架去。

还有一些服务端渲染框架例如 hypernova 等等，他们会使用其他方式达到渲染服务端组件的目的。我们可以不使用 next.js 的写法，而是采用普通组件的写法来达到服务端渲染的目的。当然也可以我们自己维护一个 node.js 服务来使用最单纯的方式做服务端渲染。

但是使用专门的 node.js 也有他的优势，比如默认的 node cluster 加成，部分专门做服务端渲染的框架会启动多个 worker 来执行。

我们用一个非常简单的例子，就能让大家明白同构的意义，以一个普通的需要获取内容的页面为例，曾经我们的代码，是会在客户端 `ComponentDidMount` 发送请求，来获取请求结果，最终将结果渲染到页面中去。我们在服务端把这部分数据直接注入组件达到渲染的目的，最终只需要在客户端进行判断是否需要重新渲染，即可得知。