react中的前端路由

react本身也是一个数据驱动页面的框架，并且也是可以实现单页面开发的，它的开发模式与我们之前vue当中的单页面开发模式保持一致，都是通过前端路由去完成的

在`react`当中也有前端路由的包【在`vue`里面是通过`vue-router`来进行的】

安装`react`的前端路由包

react的前端路由包是有2个的

1. `react-router`
2. `react-router-dom`

```
$ yarn add react-router react-router-dom
```

配置`react-router`

这一点配置与之前的vue当中是截然不同的，在`react`里面一切皆组件，所以它的路由也是一个组件，同时有很多地方又与vue很像，它也有两种模式`hash`与`history`

```
import React from "react";

import { HashRouter, Route, Switch } from "react-router-dom";
import Login from "./views/Login/Login";
import Register from "./views/Register/Register";

class App extends React.Component {
  render() {
    return <HashRouter>
      <Switch>
        <Route path="/Login" component={Login}></Route>
        <Route path="/Register" render={props => {
          return <Register></Register>
        }}></Route>
      </Switch>
    </HashRouter>
  }
}

export default App;
```

代码分析:

1. `HashRouter`代表当前路由模式以及路由管理对象使用是`hash`模式，同时它还有另一种模式叫`BrowserRouter`，也就是vue-router里面的`history`模式
2. `Switch`的目的是为了保证路由路径加载的唯一性，方便后期配置二级路由【嵌套路由】
3. `Route`则是每一个路由对象，用于根据路径去加载不同的组件
4. 在加载`Register`的时候，我们使用了另一种试去加载，这一种方式称之为动态加载

路由的动态渲染

在上面的代码当中，我们配置了我们的路由及路由管理对象，其实`Login`的配置方式与`Register`的配置方式是不一样的，后面的`Register`称之为动态渲染，在动态渲染里面，我们有很多的优点

优点:

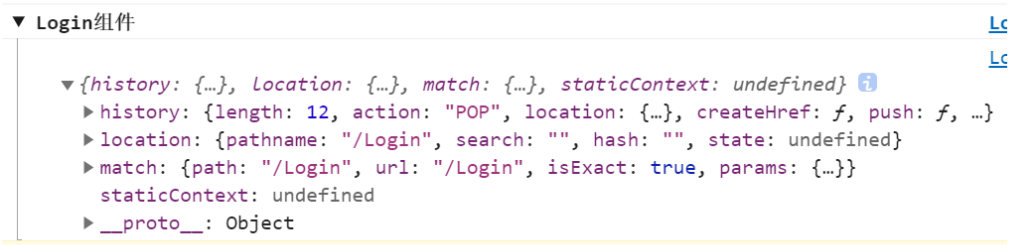
```
import React from "react";

import { HashRouter, Route, Switch, BrowserRouter } from "react-router-dom";
import Login from "./views/Login/Login";
import Register from "./views/Register/Register";

class App extends React.Component {
  render() {
    return <HashRouter>
      <Switch>
        <Route path="/Login" component={Login}></Route>
        <Route path="/Register" render={props => {
          return (new Date().getHours() == 12) ? <Register></Register> : <h2>我是
h2标签</h2>
        }}></Route>
      </Switch>
    </HashRouter>
  }
}

export default App;
```

在上面的代码当中，我们可以明显的感觉到，我们做了一个条件判断，这样就可以根据不同的逻辑去动态加载不同的路由【这一点可以理解为我们之前所说过的vue-router里面的**导航守卫**】

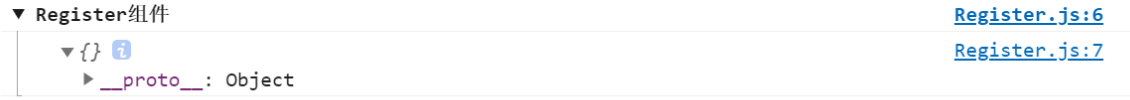
缺点:

它的缺点也很明显，我们分别在两个组件的内部去打印`props`，我们现在分别在`Login`组件与`Register`组件里面去打印`props`，结果如下图所示

Login组件是正常加载的



Register组件是路由动态加载的



通过上面的图，我们可以很明显的看到结果

通过路由静态加载的组件它的props里面是有数据的，而通过路由动态加载的组件里面的props是没有数据

这个`props`其实就是路由给你的相关操作数据，可以理解为vue-router里面的`$router`路由管理对象

解决办法:

我们通过代码其实可以看到在做动态加载的时候，`render`已经给了我们一个参数，这个参数就是`props`，我们只要把这个`props`往向传就可以了

```
<Route path="/Register" render={props => {
        return <Register {...props}></Register>
    }}></Route>
```

我们通过解构把值向组件内部传递就可以了

路由的跳转方式

在`react`里面路由的跳转方式分为2种

1. 仍然与`vue-router`里面一样，使用`router-link`这一种方式，只是在`react`换了一个个组件名称叫`Link`
2. 与`vue-router`里面的api跳转一样，通过路由管理对象去跳转

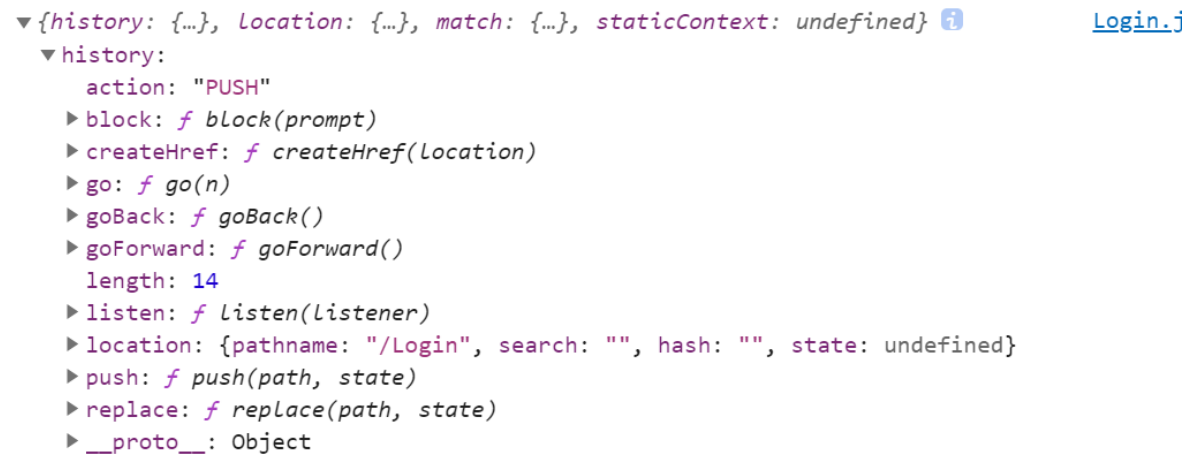
通过内置组件去跳转

在`react-router-dom`有一个专门用于跳转路由的组件，叫`Link`

```
import React from 'react';
import { Link } from "react-router-dom";

class Register extends React.Component {
    constructor() {
        super(...arguments);
        console.group("Register组件");
        console.log(this.props);
        console.groupEnd();
    }
    render() {
        return <div>
            这是注册的页面
            <hr />
            <Link to="/Login">我样去登陆页面</Link>
        </div>
    }
}

export default Register;
```

通过路由管理对象去跳转

在每个通过`Route`去加载的组件里面，我们可以直接的【渲染加载】或间接的【动态加载】拿到这个路由管理对象，这个路由管理对象在`this.props`里面，主要就是里面的`history`这个对象的方法



1. `push`推入一个新的路由【相当于新的路由入栈】
2. `goBack()`返回上一个路由【相当于路由出栈】
3. `goForward()`前进
4. `go(n)`直接前进或后退（如果里面的参数是正数则代表前进，如果是负数则代表后退）
5. `replace()`替换路由

```
<button type="button" onClick={event => {
        this.props.history.push("/Register");
    }}>我要去注册的页面</button>
```

路由之前的传值

在之前的`vue-router`里面我们可以通过路由对象进行值的传递，有query与params及state三种方式，而`react-router`也可以进行传值

query传值

> 前提：必须由其他页面跳过来，参数才会被传递过来
>
> 注：不需要配置路由表。路由表中的内容照常像下面这样配置
>
> ```
> <Route path='/test' component={test}></Route>
> ```

1. `Link`处

   ```
   <Link to={{ pathname: '/test' , query : { type: 'sex' }}}>
   ```

2. JS方式

   ```
   this.props.history.push({ pathname : '/test' ,query : { type: 'sex'} })
   ```

3. 接收值

   ```
   this.props.location.query.type
   ```

   注意：建议不用，刷新页面时丢失

state传值

> 同query差不多，只是属性不一样，而且state传的参数是加密的，query传的参数是公开的，在地址栏

1. `Link`处

```
<Link to={{ pathname : ' /test' , state : { type: 'sex' }}}>
```

1. JS方式

```
this.props.history.push({ pathname:'/test',state:{type: 'sex' } })
```

2. 接收值

```
this.props.location.state.sex
```

params传值

这一种传值方式就解决了刷新以后值丢失的问题

1. 路由表中

```
<Route path=' /test/:id '   component={test}></Route>
```

2. `Link`处

```
<Link to={ ' /test/ ' + ' 2 ' }  activeClassName='active'>XXXX</Link>
```

3. JS方式

```
 this.props.history.push(  '/user/'+'2'  )
```

4. 接收值

```
 通过  this.props.match.params.id       就可以接受到传递过来的参数（id）
```

上面的方式是`react`框架内置的三种方式，各有优点也各有缺点

第一种query与第二种state都不能刷新页面，第三种params的方式虽然可以刷新页面，仍是它必须要先改路由表，要把原来的`/Register`改为`/Register/:userName`

针对上面的情况，我们有一个自己的方叫是其于`react`的`search`来进行的

search传值

1. JS方式跳转

```
this.props.history.push("/Register?userName=biaogege&age=18");
```

在上面的方式跳转以后，最生生成的URL地址如下

http://localhost:3000/#/Register?userName=biaogege&age=18

在上面的地址里面，我们可以看到它有一个`?`，这个东西我们太熟悉不过了

2. 接收值

```
let u = new URLSearchParams(this.props.location.search);
let userName = u.get("userName");
let age = u.get("age");
```

react-router的嵌套路由

在之前的vue-router我们可以通过一个方式实现路由的嵌套，在`react-router`里面，它一样的可以实现路由的嵌套，只是实现起来可以比较难以理解

我们以上图为例子进行开发，分别新建了如下几个页面

1. `Home`页面
2. `ChooseFood`页面，它是一个二级路由页面
3. `Order`页面，它也是一个二级路由页面
4. `Detail`页面
5. `App.js`根组件

App.js的代码

```
import React from "react";
import "./App.css";
import {HashRouter,Switch,Route} from "react-router-dom"
import Home from "./views/Home/Home";
import Detail from "./views/Detail/Detail";
class  App extends  React.Component{
  render() {
    return <HashRouter>
      <Switch>
        <Route path="/Home" component={Home}></Route>
        <Route path="/Detail" component={Detail}></Route>
      </Switch>
    </HashRouter>
  }
}
export default App;
```

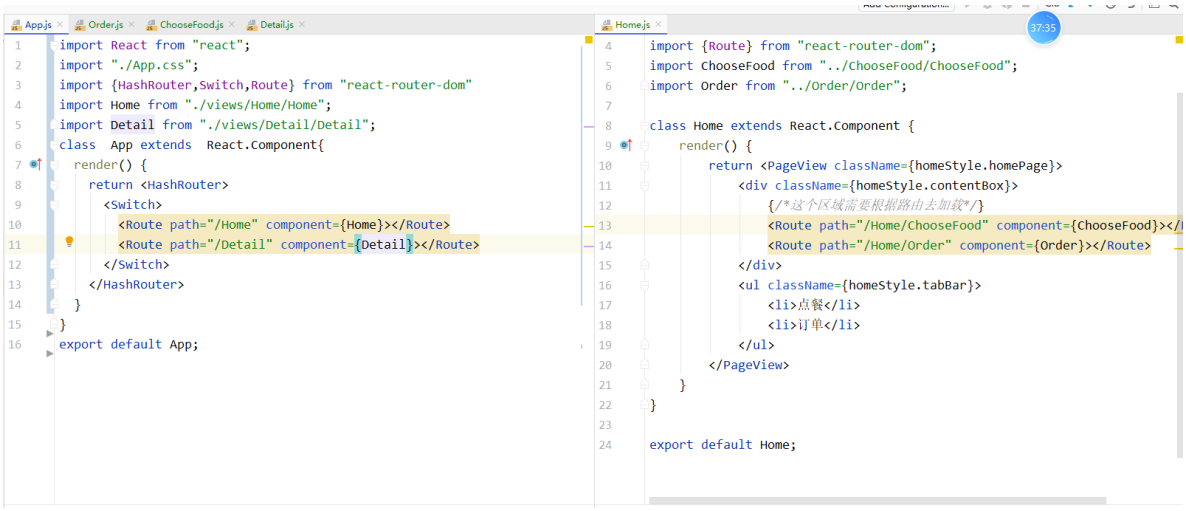
我们现在是在APP.js里面配置了一级路由，同时我们还发现，在Home的页面下面，有二级路由页面，所以我们现在找到这个页面，去书写代码

Home.js的代码

```
import React from "react";
import PageView from "../../components/PageView/PageView";
import homeStyle from "./Home.module.css";
import {Route} from "react-router-dom";
import ChooseFood from "../ChooseFood/ChooseFood";
import Order from "../Order/Order";

class Home extends React.Component {
    render() {
        return <PageView className={homeStyle.homePage}>
            <div className={homeStyle.contentBox}>
                {/*这个区域需要根据路由去加载*/}
                <Route path="/Home/ChooseFood" component={ChooseFood}></Route>
                <Route path="/Home/Order" component={Order}></Route>
            </div>
            <ul className={homeStyle.tabBar}>
                <li>点餐</li>
                <li>订单</li>
            </ul>
        </PageView>
    }
}
export default Home;
```

我们在`contentBox`的这个区域又配置了二级路由，这样就完成了二级路由的配置

上面的配置方法是官方文档的配置方式，但是它有很大的缺点



1. 如果我们在`App.js`里面改变了路由的一级路径以后，二级路径也要我们手动去改变，很麻烦
2. 路由没有实现集中的管理，后期管理起来会非常困难

正是因为有上面的问题，所以大多数项目当中都会采用另一种形式去完成嵌套路由

App.js的代码

```
import React from "react";
import "./App.css";
import {HashRouter, Switch, Route} from "react-router-dom"
```

```
import Home from "./views/Home/Home";
import Detail from "./views/Detail/Detail";
import ChooseFood from "./views/ChooseFood/ChooseFood";
import Order from "./views/Order/Order";

class App extends React.Component {
    render() {
        return <HashRouter>
            <Switch>
                <Redirect from="/" to="/Home/ChooseFood" exact></Redirect>
                <Route path="/Home" render={props => {
                    return <Home {...props}>
                        <Route path="/Home/ChooseFood" component={ChooseFood}
exact></Route>
                        <Route path="/Home/Order" component={Order} exact>
</Route>
                    </Home>
                }}></Route>
                <Route path="/Detail" component={Detail} exact></Route>
            </Switch>
        </HashRouter>
    }
}

export default App;
```

> 在路由上面添加`exact`代表的是精确匹配路径，如果这个路由没有子路由了，我们就添加，如果有子路由，一定不能添加

Home.js的代码

```
import React from "react";
import PageView from "../../components/PageView/PageView";
import homeStyle from "./Home.module.css";
import {NavLink} from "react-router-dom";

class Home extends React.Component {
    render() {
        return <PageView className={homeStyle.homePage}>
            <div className={homeStyle.contentBox}>
                {/*这个区域需要根据路由去加载*/}
                {this.props.children}
            </div>
            <div className={homeStyle.tabBar}>
                <NavLink to="/Home/ChooseFood" activeClassName=
{homeStyle.selected}>点餐</NavLink>
                <NavLink to="/Home/Order"  activeClassName={homeStyle.selected}>
订单</NavLink>
            </div>
        </PageView>
    }
}

export default Home;
```

> `<NavLink>`是一个特殊的组件，可以用于配置`tabBar`，它里面有一个`activeClassName`用于设置路由被激活时的样式

通过配置文件渲染路由

我们在刚刚的路由的学习里面，已经可以配置完成了，但是这样有一个问题，如果这个路由页面过多，我们就会写很多个`Route`这样看起来非常麻烦，所以我们需要通过像`vue-router`一样的配置文件去生成

1. 先在当前项目的src目录下面创建一个`router`的目录，然后再创建一个`routes.js`的文件，代码如下

```
import ChooseFood from "../views/ChooseFood/ChooseFood";
import Detail from "../views/Detail/Detail";
import Home from "../views/Home/Home";
import Order from "../views/Order/Order";

const routes = [
    {
        name: "Home",
        path: "/Home",
        component: Home,
        children: [
            {
                name: "ChooseFood",
                path: "/Home/ChooseFood",
                component: ChooseFood
            }, {
                name: "Order",
                path: "/Home/Order",
                component: Order
            }
        ]
    }, {
        name: "Detail",
        path: "/Detail",
        component: Detail
    }
]
export default routes;
```

   这个文件的格式就与我们之前的`vue-router`里面的格式很像，都是使用配置文件的形式去完成路由的书写，接下来，我们只需要根据这个配置文件去生成路由就可以了

2. 在`router`目录下在创建一个`renderRoutes.js`的文件，代码如下

```
import { Route } from "react-router-dom";
//这里定义了一个renderRoutes的方法
const renderRoutes = (routes) => {
    return routes.map((item, index) => {
        return <Route key={item.name} path={item.path} exact={item.children
? false : true} render={props => {
            return <item.component {...props}>
                {item.children && renderRoutes(item.children)}
            </item.component>
        }}></Route>
    });

}
```

```
export default renderRoutes;
```

通过上面的方法，我们可以将一个配置文件去渲染成`<Route>`的组件，并且自动的判断有没有子路由

3. 在`App.js`里面去调用这个方法，生成路由

```
import React from 'react';
import "./App.css";
import { HashRouter, Route, Switch ,Redirect} from "react-router-dom";
import routes from "./router/routes";
import renderRoutes from "./router/renderRoutes"


class App extends React.Component {
  render() {
    return <HashRouter>
      <Switch>
        <Redirect from="/" to="/Home/ChooseFood" exact></Redirect>
        {renderRoutes(routes)}
      </Switch>
    </HashRouter>
  }
}

export default App;
```

通过name来管理路由

之前我们在进行路由跳转的时候，我们发现我们只能通过路径去管理路由，这个时候就很被动，因为路径在开发的时候可能会因为各种原生发生变化，这个时候我们是否能够像`vue-router`一样通过路由的名称去管理呢

刚刚在书写`react`路由的配置文件`route.js`的时候，我们已经在每一个路由对象上面设置了`name`的属性值，也同时设置了`path`的属性值，所以我们可以写一个方法将这个两个进行一个封装，形成一个对象关系即可

在router目录下面创建一个routeName.js的文件，代码如下

```
import routes from "./routes";

/**
 * @param {routes} routes
 */
const getRouteNames = (routes) => {
    const routeNames = {};
    const _temp = (routes) => {
        routes.forEach(item => {
            routeNames[item.name] = item.path;
            if (item.children && Array.isArray(item.children)) {
                _temp(item.children);
            }
        })
    }
    _temp(routes);
```

```
    return routeNames;
}
const routeNames = getRouteNames(routes);

export default routeNames;
```

经过上面的代码运行以后，最终得到的`routeNames`的对象如下

```
{
    Home:"/Home",
    Detail:"/Detail",
    ChooseFood:"/Home/ChooseFood",
    Order:"/Home/Order"
}
```

有一个像这样的对象以后，我们后期管理路由以后就好一些了

```
import routeNames from "../../router/routeNames";

<div className={homeStyle.tabBar}>
    <div>
        {/*可以看到在这里，已经通过这Name的方式在管理路由了*/}
        <NavLink to={routeNames.ChooseFood} activeClassName=
{homeStyle.selected}>点餐</NavLink>
    </div>
    <div>
        <NavLink to={routeNames.Order} activeClassName={homeStyle.selected}>订单
</NavLink>
    </div>
</div>
```