React-router学习

什么是路由？简单来说，路由就是URL到函数的映射。

route就是一条路由，它将一个URL路径和一个函数进行映射，例如：

/users -> getAllUsers()

/users/count -> getUsersCount()

这就是两条路由，当访问 /users 的时候，会执行 getAllUsers() 函数；当访问 /users/count 的时候，会执行 getUsersCount() 函数。

而 router 可以理解为一个容器，或者说一种机制，它管理了一组 route。简单来说，route 只是进行了URL和函数的映射，而在当接收到一个URL之后，去路由映射表中查找相应的函数，这个过程是由 router 来处理的。

**不仅仅是URL**  
在 router 匹配 route 的过程中，不仅会根据URL来匹配，还会根据请求的方法来看是否匹配。例如上面的例子，如果通过 POST 方法来访问 /users，就会找不到正确的路由。

## 动态路由

上面提到的例子都是静态路由，也就是说，路径都是固定的。但是有时候我们需要在路径中传入参数，例如获取某个用户的信息，我们不可能为每个用户创建一条路由，而是在通过捕获路径中的参数（例如用户id）来实现。

一、基本用法

React Router 安装命令如下。

$ npm install -S react-router

路由器Router就是React的一个组件

import { Router } from 'react-router';

render(<Router/>, document.getElementById('app'));

Router组件本身只是一个容器，真正的路由要通过Route组件定义.

import { Router, Route, hashHistory } from 'react-router';

render((

<Router history={hashHistory}>

<Route path="/" component={App}/>

</Router>

), document.getElementById('app'));

上面代码中，用户访问根路由/（比如http://www.example.com/），组件APP就会加载到document.getElementById('app')。

你可能还注意到，Router组件有一个参数history，它的值hashHistory表示，路由的切换由URL的hash变化决定，即URL的#部分发生变化。举例来说，用户访问http://www.example.com/，实际会看到的是http://www.example.com/#/。

Route组件定义了URL路径与组件的对应关系。你可以同时使用多个Route组件。

<Router history={hashHistory}>

<Route path="/" component={App}/>

<Route path="/repos" component={Repos}/>

<Route path="/about" component={About}/>

</Router>

上面代码中，用户访问/repos（比如http://localhost:8080/#/repos）时，加载Repos组件；访问/about（http://localhost:8080/#/about）时，加载About组件。

## 二、嵌套路由

Route组件还可以嵌套。

<Router history={hashHistory}>

<Route path="/" component={App}>

<Route path="/repos" component={Repos}/>

<Route path="/about" component={About}/>

</Route>

</Router>

上面代码中，用户访问/repos时，会先加载App组件，然后在它的内部再加载Repos组件。

<App>

<Repos/>

</App>

App组件要写成下面的样子。

export default React.createClass({

render() {

return <div>

{this.props.children}

</div>

}

})

上面代码中，App组件的this.props.children属性就是子组件。

子路由也可以不写在Router组件里面，单独传入Router组件的routes属性。

let routes = <Route path="/" component={App}>

<Route path="/repos" component={Repos}/>

<Route path="/about" component={About}/>

</Route>;

<Router routes={routes} history={browserHistory}/>

// 路由配置说明（你不用加载整个配置，

// 只需加载一个你想要的根路由，

// 也可以延迟加载这个配置）。

React.render((

<Router>

<Route path="/" component={App}>

<Route path="about" component={About}/>

<Route path="users" component={Users}>

<Route path="/user/:userId" component={User}/>

</Route>

<Route path="\*" component={NoMatch}/>

</Route>

</Router>

), document.body)

React Router 知道如何为我们搭建嵌套的 UI，因此我们不用手动找出需要渲染哪些 <Child> 组件。举个例子，对于一个完整的 /about 路径，React Router 会搭建出 <App><About /></App>。

## 三、 path 属性

Route组件的path属性指定路由的匹配规则。这个属性是可以省略的，这样的话，不管路径是否匹配，总是会加载指定组件。请看下面的例子。

<Route path="inbox" component={Inbox}>

<Route path="messages/:id" component={Message} />

</Route>

上面代码中，当用户访问/inbox/messages/:id时，会加载下面的组件。

<Inbox>

<Message/>

</Inbox>

如果省略外层Route的path参数，写成下面的样子。

<Route component={Inbox}>

<Route path="inbox/messages/:id" component={Message} />

</Route>

现在用户访问/inbox/messages/:id时，组件加载还是原来的样子。

<Inbox>

<Message/>

</Inbox>

## 四、通配符

path属性可以使用通配符。

<Route path="/hello/:name">

// 匹配 /hello/michael

// 匹配 /hello/ryan

<Route path="/hello(/:name)">

// 匹配 /hello

// 匹配 /hello/michael

// 匹配 /hello/ryan

<Route path="/files/\*.\*">

// 匹配 /files/hello.jpg

// 匹配 /files/hello.html

<Route path="/files/\*">

// 匹配 /files/

// 匹配 /files/a

// 匹配 /files/a/b

<Route path="/\*\*/\*.jpg">

// 匹配 /files/hello.jpg

// 匹配 /files/path/to/file.jpg

通配符的规则如下。

**（1）**:paramName

:paramName匹配URL的一个部分，直到遇到下一个/、?、#为止。这个路径参数可以通过this.props.params.paramName取出。

**（2）**()

()表示URL的这个部分是可选的。

**（3）**\*

\*匹配任意字符，直到模式里面的下一个字符为止。匹配方式是非贪婪模式。

**（4）**\*\*

\*\* 匹配任意字符，直到下一个/、?、#为止。匹配方式是贪婪模式。

path属性也可以使用相对路径（不以/开头），匹配时就会相对于父组件的路径，可以参考上一节的例子。嵌套路由如果想摆脱这个规则，可以使用绝对路由。

路由匹配规则是从上到下执行，一旦发现匹配，就不再其余的规则了。

<Route path="/comments" ... />

<Route path="/comments" ... />

上面代码中，路径/comments同时匹配两个规则，第二个规则不会生效。

<Router>

<Route path="/:userName/:id" component={UserPage}/>

<Route path="/about/me" component={About}/>

</Router>

上面代码中，用户访问/about/me时，不会触发第二个路由规则，因为它会匹配/:userName/:id这个规则。因此，带参数的路径一般要写在路由规则的底部。

此外，URL的查询字符串/foo?bar=baz，可以用this.props.location.query.bar获取。

## 五、IndexRoute 组件

下面的例子，你会不会觉得有一点问题？

<Router>

<Route path="/" component={App}>

<Route path="accounts" component={Accounts}/>

<Route path="statements" component={Statements}/>

</Route>

</Router>

上面代码中，访问根路径/，不会加载任何子组件。也就是说，App组件的this.props.children，这时是undefined。

因此，通常会采用{this.props.children || <Home/>}这样的写法。这时，Home明明是Accounts和Statements的同级组件，却没有写在Route中。

IndexRoute就是解决这个问题，显式指定Home是根路由的子组件，即指定默认情况下加载的子组件。你可以把IndexRoute想象成某个路径的index.html。

<Router>

<Route path="/" component={App}>

<IndexRoute component={Home}/>

<Route path="accounts" component={Accounts}/>

<Route path="statements" component={Statements}/>

</Route>

</Router>

现在，用户访问/的时候，加载的组件结构如下。

<App>

<Home/>

</App>

这种组件结构就很清晰了：App只包含下级组件的共有元素，本身的展示内容则由Home组件定义。这样有利于代码分离，也有利于使用React Router提供的各种API。

注意，IndexRoute组件没有路径参数path。

## 六、Redirect 组件

<Redirect>组件用于路由的跳转，即用户访问一个路由，会自动跳转到另一个路由。

<Route path="inbox" component={Inbox}>

{/\* 从 /inbox/messages/:id 跳转到 /messages/:id \*/}

＜Redirect from="messages/:id" to="/messages/:id" />

</Route>

现在访问/inbox/messages/5，会自动跳转到/messages/5。

## 七、IndexRedirect 组件

IndexRedirect组件用于访问根路由的时候，将用户重定向到某个子组件。

<Route path="/" component={App}>

＜IndexRedirect to="/welcome" />

<Route path="welcome" component={Welcome} />

<Route path="about" component={About} />

</Route>

上面代码中，用户访问根路径时，将自动重定向到子组件welcome。

## 八、Link

Link组件用于取代<a>元素，生成一个链接，允许用户点击后跳转到另一个路由。它基本上就是<a>元素的React 版本，可以接收Router的状态。

render() {

return <div>

<ul role="nav">

<li><Link to="/about">About</Link></li>

<li><Link to="/repos">Repos</Link></li>

</ul>

</div>

}

如果希望当前的路由与其他路由有不同样式，这时可以使用Link组件的activeStyle属性。

<Link to="/about" activeStyle={{color: 'red'}}>About</Link>

<Link to="/repos" activeStyle={{color: 'red'}}>Repos</Link>

上面代码中，当前页面的链接会红色显示。

另一种做法是，使用activeClassName指定当前路由的Class。

<Link to="/about" activeClassName="active">About</Link>

<Link to="/repos" activeClassName="active">Repos</Link>

上面代码中，当前页面的链接的class会包含active。

在Router组件之外，导航到路由页面，可以使用浏览器的History API，像下面这样写。

import { browserHistory } from 'react-router';

browserHistory.push('/some/path');

## 九、IndexLink

如果链接到根路由/，不要使用Link组件，而要使用IndexLink组件。

这是因为对于根路由来说，activeStyle和activeClassName会失效，或者说总是生效，因为/会匹配任何子路由。而IndexLink组件会使用路径的精确匹配。

<IndexLink to="/" activeClassName="active">

Home

</IndexLink>

上面代码中，根路由只会在精确匹配时，才具有activeClassName。

另一种方法是使用Link组件的onlyActiveOnIndex属性，也能达到同样效果。

<Link to="/" activeClassName="active" onlyActiveOnIndex={true}>

Home

</Link>

实际上，IndexLink就是对Link组件的onlyActiveOnIndex属性的包装。

## 十、history 属性

Router组件的history属性，用来监听浏览器地址栏的变化，并将URL解析成一个地址对象，供 React Router 匹配。

history属性，一共可以设置三种值。

* browserHistory
* hashHistory
* createMemoryHistory

如果设为hashHistory，路由将通过URL的hash部分（#）切换，URL的形式类似example.com/#/some/path。

import { hashHistory } from 'react-router'

render(

<Router history={hashHistory} routes={routes} />,

document.getElementById('app')

)

如果设为browserHistory，浏览器的路由就不再通过Hash完成了，而显示正常的路径example.com/some/path，背后调用的是浏览器的History API。

import { browserHistory } from 'react-router'

render(

<Router history={browserHistory} routes={routes} />,

document.getElementById('app')

)

但是，这种情况需要对[服务器改造](https://github.com/reactjs/react-router/blob/master/docs/guides/Histories.md#configuring-your-server)。否则用户直接向服务器请求某个子路由，会显示网页找不到的404错误。

如果开发服务器使用的是webpack-dev-server，加上--history-api-fallback参数就可以了。

$ webpack-dev-server --inline --content-base . --history-api-fallback

createMemoryHistory主要用于服务器渲染。它创建一个内存中的history对象，不与浏览器URL互动。

const history = createMemoryHistory(location)

## 十一、表单处理

Link组件用于正常的用户点击跳转，但是有时还需要表单跳转、点击按钮跳转等操作。这些情况怎么跟React Router对接呢？

下面是一个表单。

<form onSubmit={this.handleSubmit}>

<input type="text" placeholder="userName"/>

<input type="text" placeholder="repo"/>

<button type="submit">Go</button>

</form>

第一种方法是使用browserHistory.push

import { browserHistory } from 'react-router'

// ...

handleSubmit(event) {

event.preventDefault()

const userName = event.target.elements[0].value

const repo = event.target.elements[1].value

const path = `/repos/${userName}/${repo}`

browserHistory.push(path)

},

第二种方法是使用context对象。

export default React.createClass({

// ask for `router` from context

contextTypes: {

router: React.PropTypes.object

},

handleSubmit(event) {

// ...

this.context.router.push(path)

},

})

## 十二、路由的钩子

每个路由都有Enter和Leave钩子，用户进入或离开该路由时触发。

<Route path="about" component={About} />

＜Route path="inbox" component={Inbox}>

＜Redirect from="messages/:id" to="/messages/:id" />

</Route>

上面的代码中，如果用户离开/messages/:id，进入/about时，会依次触发以下的钩子。

* /messages/:id的onLeave
* /inbox的onLeave
* /about的onEnter

下面是一个例子，使用onEnter钩子替代<Redirect>组件。

<Route path="inbox" component={Inbox}>

<Route

path="messages/:id"

onEnter={

({params}, replace) => replace(`/messages/${params.id}`)

}

/>

</Route>

onEnter钩子还可以用来做认证。

const requireAuth = (nextState, replace) => {

if (!auth.isAdmin()) {

// Redirect to Home page if not an Admin

replace({ pathname: '/' })

}

}

export const AdminRoutes = () => {

return (

<Route path="/admin" component={Admin} onEnter={requireAuth} />

)

}

下面是一个高级应用，当用户离开一个路径的时候，跳出一个提示框，要求用户确认是否离开。

const Home = withRouter(

React.createClass({

componentDidMount() {

this.props.router.setRouteLeaveHook(

this.props.route,

this.routerWillLeave

)

},

routerWillLeave(nextLocation) {

// 返回 false 会继续停留当前页面，

// 否则，返回一个字符串，会显示给用户，让其自己决定

if (!this.state.isSaved)

return '确认要离开？';

},

})

)

上面代码中，setRouteLeaveHook方法为Leave钩子指定routerWillLeave函数。该方法如果返回false，将阻止路由的切换，否则就返回一个字符串，提示用户决定是否要切换。