**react-router（2.8.1）**

**2.0 完全向后兼容 1.0，所有在 1.0 被弃用的 API 都会在控制台以 warn 的形式打印出来，在 3.0 中将会完全移除 1.0 所弃用的东西，现在最新版本是4.2.0版本；但是4.0版本后，有很多方法不再向前兼容（这里讲解的为2.8.1版本）**

1. **基本用法：**
2. **首先我们需要导入一些组件...**

import { Router, Route, Link } from 'react-router'

1. **配置路由信息：**

* **使用jsx语法**

ReactDom.render((

<Router>

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

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

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

</Route>

</Router>

), document.body)

* **使用js对象进行配置**

const routes = {

path: '/',

component: App,

childRoutes: [

{ path: 'about', component: About },

{ path: 'inbox', component: Inbox },

]

}

React.render(<Router routes={routes} />, document.body)

**注:如果出现如下错误：Cannot read property 'func' of undefined；**

**因为react16版本以后不再兼容react-router的2，3版本**

1. **获取参数**

* **通过动态路由片段**

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

**可以通过如下方式进行获取id：**

this.props.params.id

* **通过query字符串**

/foo?bar=baz；

this.props.location.query.bar

1. **路由的配置规则（Router）**
2. **路径语法：**

* **:paramName – 匹配一段位于 /、? 或 # 之后的 URL。 命中的部分将被作为一个[参数](http://react-guide.github.io/react-router-cn/docs/guides/basics/docs/Glossary.md" \l "params)**
* **() – 在它内部的内容被认为是可选的**
* **\* – 匹配任意字符（非贪婪的）直到命中下一个字符或者整个 URL 的末尾，并创建一个 splat [参数](http://react-guide.github.io/react-router-cn/docs/guides/basics/docs/Glossary.md" \l "params)**

**如下解释：**

// 匹配 /hello/michael 和 /hello/ryan

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

// 匹配 /hello, /hello/michael 和 /hello/ryan

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

// 匹配 /files/hello.jpg 和 /files/path/to/hello.jpg

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

1. **常用路由类型：**

[browserHistory](http://react-guide.github.io/react-router-cn/docs/guides/basics/Histories.html" \l "browserHistory)：react推荐使用（采用的是html5的浏览器路由纪录）

[hashHistory](http://react-guide.github.io/react-router-cn/docs/guides/basics/Histories.html" \l "hashHistory)：默认的（采用的是hash的浏览器路由纪录）

[createMemoryHistory](http://react-guide.github.io/react-router-cn/docs/guides/basics/Histories.html" \l "creatememoryhistory)

使用[browserHistory](http://react-guide.github.io/react-router-cn/docs/guides/basics/Histories.html" \l "browserHistory)如下：

import { browserHistory } from 'react-router'

<Router history={browserHistory}>

1. **路由的跳转配置（Link）：**

**需要动态的引入组件：**

import { Link} from “react-router“

<Link></Link>

**属性：**

**to**: 值为字符串，例:”/list” (路由跳转的路径)；

**query**:值为对象，例：{name:1} (这里相当于/list?name=1,这种路由的书写方式)；

**state**：值为对象，可以通过this.location.state获取到当前的值；

**activeClassName**：当某个 route 是激活状态时，<Link> 可以接收传入的 className。失活状态下是默认的 class。写法如下：activeClassName="current"

**activeStyle：**当某个 route 是激活状态时，可以将样式添加到链接元素上；写法如下：activeStyle={{color: 'red'}}

**onClick**(e)：自定义点击事件的处理方法。如处理 <a> 标签一样 - 调用 e.preventDefault() 来防止过度的点击，同时 e.stopPropagation() 可以阻止冒泡的事件

1. **默认路由的设置：**

**jsx:**

<IndexRoute component={ Index }/>

**js对象：**

const routes = {

path: '/',

component: App,

indexRoute:{

component:Index

},

childRoutes: [

{ path: 'list', component: List },

]

}

在嵌套路由中使用{this.props.children}来渲染动态路由，这里相当于vue-router中的router-view

1. **路由对应组件的渲染**
2. **单一组件的渲染：（this.props.children）**

**路由：**

<Route path='center' component={Center}>

<Route path="left" component={Left}></Route>

</Route>

**组件：（Center）**

render(){

return (

<div>

<p>这里是中心也</p>

{this.props.children}

</div>

)

}

1. **多组件的渲染**

**路由：**

<Route path='center' component={Center}>

<Route path="left" components={{left:Left,right:Right}}></Route>

</Route>

**组件：**

render(){

const { left, right } = this.props

return (

<div>

<p>这里是中心也</p>

{left}-{right}

</div>

)

}

1. **异步组件（组件的懒加载）**

* **单组件路由的定义：（getComponent()方法使用）**

**注：这里采用的是webpack的require.ensure()进行的组件懒加载，接受三个参数，第一个参数为加载当前模块时的依赖，第二个参数为加载的当前的模块的回掉函数，第三个参数为加载完毕后模块的名称；**

**这里的getComponent函数第一个参数为当前路由的信息，第二个参数为一个回调函数，这里的回掉函数遵循node.js的回掉函数风格(回调函数的第一个参数为错误的捕捉，一般无错误，所以一般默认填入null，第二个参数为异步装载成功的组件；)**

<Route path="list" getComponent={(location, cb) => {

require.ensure([],(require)=>{

cb(null, require('./components/routerPage/list.jsx').default)

},'List')

}} />

* **多组件路由的定义：（getComponents方法）**

<Route path="center" getComponents={(location, cb) => {

require.ensure([],function(require){

cb(null, {center:require('./components/routerPage/center.jsx').default,list:require('./components/routerPage/center.jsx').default})

},'Center')

* **用js对象的方式配置路由组件的懒加载**

const routes = {

path: '/',

getComponent(nextState, cb) {

require.ensure([], (require) => {

cb(null, require('./components/app.jsx').default)

}, 'App')

},

indexRoute:{

getComponent(nextState, cb) {

require.ensure([], (require) => {

cb(null, require('./components/index.jsx').default)

}, 'Index')

},

},

childRoutes: [

// require("./routes/listR.js").default

{

path: 'list',

getComponent(nextState, cb) {

require.ensure([], (require) => {

cb(null, require('./components/routerPage/list.jsx').default)

}, 'List')

}

}

]

}

1. **Route的钩子函数：（Route组件上书写）**

##### onEnter(nextState, replaceState, callback?)

当 route 即将进入时调用。它提供了下一个路由的 state，一个函数重定向到另一个路径。this 会触发钩子去创建 route 实例。

当 callback 作为函数的第三个参数传入时，这个钩子将是异步执行的，并且跳转会阻塞直到 callback 被调用。

##### onLeave()

当 route 即将离开时调用

1. **路由注入到组件的props上的属性和参数：**

* **history：路由的前端历史纪录和路由跳转方法**

**go():**

**goBack():**

**goForward():**

**push():**

**replace()**

* **location:当前路由的参数信息**

**state:通过在link上的state属性设置的对象值**

**action:当前路由的纪录形式**

**pathname：当前hash路由**

**query：通过路由的传参，路径？后的路径，或者是query属性传递的**

* **params：通过动态路由传递的值**
* **routeParams:（小心有坑）**

this.props.params 是直接在组件中指定 route 的一个子集。例如，如果 route 的路径是 users/:userId 而 URL 是 /users/123/portfolios/345，那么 this.props.routeParams 会是 {userId: '123'}，并且 this.props.params 会是 {userId: '123', portfolioId: 345}。

* **route:当前路由的配置信息**
* **routes:当前路由所包含的所有路由片段**

1. **在路由跳转前阻止路由的跳转：**

**在组件内部如下引入：**

import { withRouter } from 'react-router'

**组件抛出时：**

export default withRouter(List)

**组件回调函数的注册：**

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

return '确认要离开？';

}

1. **数据的请求**

**在constructor和componentDidUpdate中（在进入路由或者进入路由的相应组件内部进行数据请求）；**

**withRouter进入目标路由或组件前，进行数据的请求；**