**reactRouter4.0版本**

**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；4.0版本后，有很多方法不再向前兼容，在这里使用react的版本为16.2.0版本（16.2.0版本不再兼容2.0、3.0版本的路由）**

**如果采用4.0版本，只需要将react-router和react-router-dom引入一个即可；**

1. **基本用法：**

**main组件：**

import React from 'react'

import ReactDom from 'react-dom'

import Router from "./router/index"

ReactDom.render(<Router />,document.getElementById("root"))

**Router/Index组件：**

import React from "react"

import {

// HashRouter as Router

BrowserRouter as Router,

Route,

Switch

} from "react-router-dom"

import App from "../components/App"

import Index from "../components/Index"

import parentRender from "./parentRender"

import ChildrenRender from "./childrenRender"

const baseRouter=()=>{

return (

<Router basename="/" >

<App>

<Switch>

<Route exact path="/" component={Index} />

<Route path="/parentrender" render={parentRender} />

<Route path="/children" render={ChildrenRender} />

</Switch>

</App>

</Router>

)

}

// child

// const ParentRouter=()=>{

// return (

// <Parent>

// <Route path="/parent/child" component={Child} />

// </Parent>

// )

// }

export default baseRouter

注：如果使用html5的historyApi时，需要在webpack的devserver中添加如下配置：

**historyApiFallback:true, //任意的 404 响应都可能需要被替为 index.html**

1. **BrowserRouter：**

**basename：**添加基路由，如果所有的路由都在/app下，配置basename:”/app”

如：<BrowserRouter basename="/calendar"/>

<Link to="/today"/> // 渲染为 <a href="/calendar/today">

**getUserConfirmation**：第一次进入路由时的调用的一个函数，无实际意义

**forceRefresh**：当浏览器不支持 HTML5 的 history API 时强制刷新页面（使用history才有效）

**keyLength**：设置它里面路由的 location.key 的长度。默认是6。（key的作用：点击同一个链接时，每次该路由下的 location.key都会改变，可以通过 key 的变化来刷新页面。）

1. **MemoryRouter：在内存中管理history，地址栏不会变化。在**

**reactNative中使用**

<MemoryRouter initialEntries={[ '/prompt', '/children', { pathname: '/three' } ]}>

1. **HashRouter: 使用hash方式进行路由（使用方式跟BrowserRouter类似，不再赘述）**
2. **Route:定义路由**

* **render methods：**
* **component:** router 将使用 React.createElement 根据给定的 component 创建一个新的 React 元素,所以如果直接渲染一个jsx语法的dom结构可以直接使用属性进行渲染；对于内联渲染（inline rendering）, 建议使用 render，component的属性值不要直接在{}中书写函数，然后返回节点，这种属于内联渲染，可以如下将返回的jsx放在一个外部函数中；

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

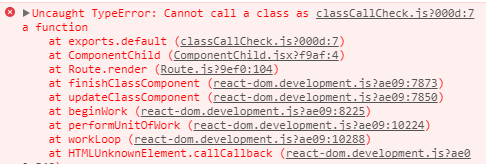
**const User = ({ match }) => {**

**return <h1>Hello {match.params.username}!</h1>**

**}**

* **render：属性值为一个函数，可以进行组合；**

**注：如果在定义子路由或者路由时，出现如下错，将渲染方式改成component，因为render必须为无状态组件：**



**使用render实现路由嵌套，定义子路由：**

**parentRender组件：**

import React from "react"

import {

BrowserRouter as Router,

Route,

Switch

} from "react-router-dom"

import Parent from "../components/test/render/Parent"

import Child from "../components/test/render/Child"

const ParentRouter=()=>{

return (

<Parent>

<Route path="/parent/child" component={Child} />

</Parent>

)

}

export default ParentRouter

* **children：检测当前路径是否被匹配，或者想要改变点击dom的样式以及其他东西；**

**childrenRender组件：**

import React from "react"

import {

BrowserRouter as Router,

Route,

Switch

} from "react-router-dom"

import Parent from "../components/test/children/Parent"

import Child1 from "../components/test/children/Child1"

import Child2 from "../components/test/children/Child2"

const ParentRouter=()=>{

return (

<Parent>

<Switch>

<Route path={`/children/child1`} render={()=><h5>子组件child1</h5> } />

<Route path="/children/child2" render={()=><h5>子组件child2</h5> } />

</Switch>

</Parent>

)

}

export default ParentRouter

**Parent组件：**

import React, { Component } from 'react'

import {

Route,

Link,

Switch

} from "react-router-dom"

class Parent extends Component{

constructor(props){

super(props)

}

render(){

const active={background:'green'}

const ListItemLink = (matchs) => {

console.log(matchs)

const {to,...rest}=matchs

return (

<Route path={to} children={(matchD) => {

console.log(matchD)

const {match,history}=matchD

return(

<li style={match?active:{}}>

<Link to={to} {...rest} />

</li>

)

}

} />

)

}

return (<div>

这里是children属性进行渲染页

<ul>

<ListItemLink to="/children/child1">children1</ListItemLink>

<ListItemLink to="/children/child2" >children2</ListItemLink>

</ul>

<hr/>

{this.props.children}

</div>)

}

}

export default Parent

* **route的props的三个属性（参数）：**

**match**

**location**

**history**

* **path：设置匹配的路径**
* e**xact: bool**

如果为 true，path 为 '/one' 的路由将不能匹配 '/one/two'，反之，亦然。

* **strict: bool**

对路径末尾斜杠的匹配。如果为 true。path 为 '/one/' 将不能匹配 '/one' 但可以匹配 '/one/two'。

1. **Link: 路由的跳转**

**import { Link } from 'react-router-dom'**

* **to：注意一下几种写法：string、object**
* <Link to="/about">关于</Link>
* <Link to="/courses"/>
* <Link to={{

pathname: '/courses',

search: '?sort=name',

state: { fromDashboard: true }

}}/>

* **replace:替换路径，不会再history中留下记录；**
* <Link to="/courses" replace />
* 组件内部获取参数：
* 在组件类上添加contextTypes的props的校验

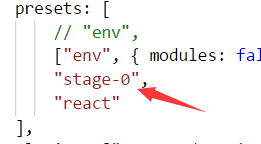
// 这一步是重点

static contextTypes = {

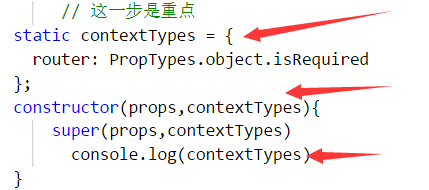
router: PropTypes.object.isRequired

};

如果使用es6的静态方法，需要：



在组建内部获取参数：



1. **NavLink：针对导航类的路由跳转提供的组件**

* **activeClassName: string**  
  导航选中激活时候应用的样式名，默认样式名为 active
* **activeStyle: object**如果不想使用样式名就直接写style
* **exact: bool**若为 true，只有当访问地址严格匹配时激活样式才会应用
* **strict: bool**若为 true，只有当访问地址后缀斜杠严格匹配（有或无）时激活样式才会应用
* **isActive: func**决定导航是否激活，或者在导航激活时候做点别的事情。不管怎样，它不能决定对应页面是否可以渲染。

1. **Switch：只渲染出第一个与当前访问地址匹配的 <Route> 或 <Redirect>**

<Switch>

<Route exact path="/" component={Index} />

<Route path="/parentrender" render={parentRender} />

<Route path="/children" render={ChildrenRender} />

</Switch>

1. **Redirect：渲染时将导航到一个新地址，这个新地址覆盖在访问历史**

**信息里面的本该访问的那个地址**

<Route path='/redirect' render={()=>{

return <Redirect to="/parentrender" />

}}/>

* to: string **重定向的 URL 字符串**
* to: object **重定向的 location 对象**
* push: bool **若为真，重定向操作将会把新地址加入到访问历史记录里面，并且无法回退到前面的页面。**
* from: string **需要匹配的将要被重定向路径。**

1. **Prompt：当用户离开当前页面前做出一些提示**

<Route path='/prompt' render={()=>{

return (

<div>

这里是prompt

<Prompt message="Are you sure you want to leave?"/>

<Link to="/">跳转</Link>

</div>

)

}}/>

* **message: string**  
  当用户离开当前页面时，设置的提示信息。
* **message: func**  
  当用户离开当前页面时，设置的回掉函数
* **when: bool**  
  通过设置一定条件要决定是否启用 Prompt

1. **组件懒加载（按需加载）（这里讲解三种方式）（推荐使用第三种）**
2. **采用高阶组件和bundle-loader插件**

**首先安装bundle-loader插件：**

**cnpm install –save-dev bundle-loader**

**lazy组件第一种：**

import React, { Component } from 'react'

class Bundle extends Component {

state = {

mod: null

}

componentWillMount() {

this.load(this.props)

}

componentWillReceiveProps(nextProps) {

if (nextProps.load !== this.props.load) {

this.load(nextProps)

}

}

load(props) {

this.setState({

mod: null

})

props.load((mod) => {

this.setState({

// handle both es imports and cjs

mod: mod.default ? mod.default : mod

})

})

}

render() {

if (!this.state.mod)

return false

return this.props.children(this.state.mod)

}

}

const Lazy = (component) => {

return ()=>{

return (

<Bundle load={component}>

{(List) => <List />}

</Bundle>

)

}

}

export default Lazy

**lazy组件第二种：**

import React, { Component } from 'react'

const Lazy = (load) => {

return class extends Component {

state = {

mod: null

}

componentWillMount() {

console.log(load)

this.load()

}

load() {

this.setState({

mod: null

})

load((mod) => {

console.log(mod)

this.setState({

// handle both es imports and cjs

mod: mod.default ? mod.default : mod

})

})

}

render() {

let Mod=this.state.mod

return Mod?<Mod />:null

}

}

}

export default Lazy

**在index.jsx组件的使用：**

// 组件懒加载

// 使用Lazy高阶组件用来异步加载组件

import Lazy from './tool/bundle';

import ListContainer from 'bundle-loader?lazy&name=[name]!../components/test/lazyloading';

<Route path='/lazyloading' render={Lazy(ListContainer)}/>

1. **使用webpack的import方法进行的加载(注意这里传入的是一个将要加载组件的路径)**

**lazyImport组件：**

import React, { Component } from 'react'

class Bundle extends Component {

state = {

mod: null

}

componentWillMount() {

this.load(this.props)

}

componentWillReceiveProps(nextProps) {

if (nextProps.load !== this.props.load) {

this.load(nextProps)

}

}

load(props) {

this.setState({

mod: null

})

// 这里import抛出的是promise

props.load().then((mod) => {

this.setState({

// handle both es imports and cjs

mod: mod.default ? mod.default : mod

})

})

}

render() {

if (!this.state.mod)

return false

return this.props.children(this.state.mod)

}

}

export default Bundle

**在index.jsx组件的使用：**

import Bundle from "../tool/lazyImport"

const ListContainer=()=>{

return (

<Bundle load={()=>import(("../../components/test/lazyloading"))}>

{(List) => <List />}

</Bundle>

)

}

<Route path='/tests/lazyloading' component={ListContainer}/>

1. **借助webpack和es6的import和async实现异步组件的封装**

**asyncComponent组件：**

import React, { Component } from "react";

export default function asyncComponent(importComponent) {

class AsyncComponent extends Component {

constructor(props) {

super(props);

this.state = {

component: null

};

}

async componentDidMount() {

const { default: component } = await importComponent();

this.setState({

component: component

});

}

render() {

const C = this.state.component;

return C ? <C {...this.props} /> : null;

}

}

return AsyncComponent;

}

**在index.jsx组件的使用：**

import AsyncComponent from "../tool/asyncComponent"

const ListContainer = AsyncComponent(() => import("../../components/test/lazyloading"));

<Route path='/tests/lazyloading' component={ListContainer}/>