**Route组件**用于匹配URL地址，并且在匹配的情况下渲染出对应的组件. 只要URL中的地址匹配<Route>中的path属性，且<Route>组件所在的自定义组件将被渲染，则渲染出<Route>组件对应的内容。

<Route>组件的props:

|  |  |
| --- | --- |
| Props | 效果 |
| path | 表示Route组件应该匹配什么URL.  匹配前缀：  <Route path=”/data”…. />  表示该Route组件将匹配所有开头含有/data的url地址.例如/data/one, /data/two等。如果想要完美匹配，使用exact prop  多种匹配：  <Route path={[“/data/one”, “/data/two”]} …/>  如果向path prop指定数组  使用正则表达式匹配：  <Route path={[“/people/b\*”]} …/>  P.s当url满足path中的匹配时，对应的组件会mountd,触发 组件的componentDidMount().当url切换到其他地址时候，url与path中不匹配，此时之前匹配而产生的组件将会unmounted，触发组件的componentDidUnmount(). |
| exact | 当设置为true时候，只有Route组件中匹配URL与现在URL完全符合时候才会渲染组件 |
| sensitive | 当设置为true时候，匹配URL时候将变为大小写敏感 |
| strict | 当设置为true时候，如果匹配url最后以/结尾，那么URL必须也要以/结果才会成功匹配 |
| component  (与render属性互斥) | 当path匹配成功时候，渲染出什么组件 |
| render  (与component属性互斥) | Render指定一个方法，该方法接受一个对象作为其参数，该参数对象为表示路由系统的状态对象。含有match, location, history属性。***要求我们指定的方法返回渲染内容。***当path匹配成功的时候，使用该方法返回的内容作为渲染内容。当path匹配不成功时候，不渲染任何内容。  我们可以使用该方法为我们组件传入更多props.  例如：  <Route path=”/products” render={(routeProps}=> <ProductDisplay myProps=”myvalue” /> ) />) |
| children | Children用于指定一个方法，这个方法接收一个参数，这个参数是一个代表路由状态的对象。对象中还有match, location, history这三个属性. 这个方法将会返回渲染的内容。  **并且不管当前URL是否匹配<Route>中的path属性，都会渲染显示指定方法的返回内容**。  当匹配时，路由状态对象.match为match对象。  不匹配时，路由状态对象.match为undefined. |

**Switch组件**

React-router提供switch组件，用以包裹<Route>组件，***被包裹的这些<Route>相当于被switch进行归成一组。当需要url匹配时候，这些归组的<Route>逐一进行匹配，并且当有一个<Route>被响应时，匹配结束。***也正因为如此，我们需要将最特殊的匹配放在最前面。

例如：

<Switch>

<Route path="/products" component={ProductDisplay} />

<Route path="/suppliers" component={SupplierDisplay} />

<Route render={()=>this.renderMessage("Fallback Route")} /> //没有path prop的<Route>将匹配所有

</Switch>

如果不被<Switch>归组，则所有的满足匹配的<Route>都将响应，渲染自身的组件。

**Redirect组件**

|  |  |
| --- | --- |
| Props | 效果 |
| to | 表示此redirect组件将重导的位置 |
| from | 表示限制Redirect组件，Redirect组件只有当现在URL地址和from中指定URL地址匹配时重导 |
| exact | 默认为false.当设为true时候，Redirect组件只有当现在URL地址和from中指定URL地址完全相同时才重导 |
| strict | 默认为 false.当设为true时候，Redirect组件只有当现在URL地址以/结尾，并且from指定url地址也是由/结尾时候才重导 |
| push | 当设置为true时候，重导会在浏览器中历史中添加一个新的记录。默认为false时候，不会 |

例如：

<Redirect from=”/ABC” to=”/BC” /> 当满足当前url为/ABC时候，满足该<Redirect>组件的条件。此时<Redirect>的行为不是渲染出需要用户再次点击的一个链接，而是直接让浏览器直接重定向到/BC URL。

**<Link>组件以及<NavLink>组件**

<Link>组件用于将页面导向一个新的URL地址，并且不重新加载React App。

|  |  |
| --- | --- |
| Props | 效果 |
| to | 表示<Link>组件将会重新导向的地址 |
| replace | 表示点击此超链接会不会替换此时URL在浏览器中的历史记录。如果替换，则用户不能通过浏览器按钮返回跳转前的页面。如果不替换，则用户可以通过浏览器按钮返回跳转前的页面。默认为false. |
| innerRef |  |

🡪<Link>组件的特性是任何向其传递的props属性，<Link>都会将这些属性进而的传给它生成的超链接元素。基于此，可以通过向<Link>指定className props用于定制其生成的超链接样式。

<NavLink>组件建立在<Link>的基础之上，它拥有<Link>组件的所有功能以及props.除此以外，最特别的是它可以指定当链接处于激活状态（一般情况下，当to prop所指向的url为此时浏览器的URL时候为激活状态）的时候的样式。

<NavLink>组件：

|  |  |
| --- | --- |
| Props | 效果 |
| activeClassName | 当<NavLink>组件处于激活状态时，添加的className |
| activeStyle | 当<NavLink>组件处于激活状态时，添加的CSS Style |
| exact | 默认为false.当设为true时候，<NavLink>组件只有当现在URL地址和to中指定URL地址完全相同时才添加样式 |
| strict | 默认为 false.当设为true时候，<NavLink>组件只有当现在URL地址以/结尾，并且to指定url地址也是由/结尾时候才添加样式 |
| isActive | 这个prop用于指定一个自定义方法，用于判定<NavLink>是否处于激活状态。自定义方法将受到两个参数match, location.  默认情况下也就是非指定自定义方法情况下，逻辑为比较prop to以及浏览器现在的URL地址 |

**Prompt组件**

Prompt组件用于延迟导航，它将在页面显示信息，需要用户确认以后才会进行跳转。在一个react应用中，最多定义一个Prompt控件。在一个React应用中，任何跳转到新的URL地址都将激活Prompt控件。

Prompt控件属性

|  |  |
| --- | --- |
| **Name** | **效果** |
| message | 将会提供给用户的信息。可以为字符串。  也可以一个方法，它将接受一个参数location（使用location对象，我们可以达到提示用户是否确定跳转到${location.path}的地方）.要求该方法返回一个字符串。 |
| when | 如果指定，则prompt组件只会在when所指定的条件为True的时候显示 |

**BrowserRouter组件**

当BrowserRouter组件引入React App中时，通常会重命名为Router.它将包裹所有的需要React-Router支持的部分，提供路由功能。在其包裹的范围内使用<Switch>, <NavLink>, <Route>组件。 它提供的根据URL路由的功能，根据URL的改变提供路由功能，过程中不向Server发送新的HttpRequest.

|  |  |
| --- | --- |
| Props | 效果 |
| basename | 当App不处于URL根节点上时，指定basename |
| getUserConfirmation | 这个prop用于指定一个自定义方法，该自定义方法将服务于自定义prompt组件.  每当react应用中发生url跳转的时候，调用此自定义方法。  使用自定义组件作为导航提示时，也需要在存在原生<Prompt>组件。  自定义方法接受两个参数.  第一个参数是显示给用户的message，他的值就是原生React<Prompt>组件message属性的值。  第二个参数是一个回调函数。  以true调用该回调函数将使浏览器继续跳转。  以false调用该回调函数将使浏览器停止跳转。 |
| forceRefresh | 默认为false.当为True时，当导航过程中发送HTTP Request到服务器时，强制完成一次完整刷新。应只用于测试环境。 |
| keyLength | 每一次改变URL都会为该操作赋值一个唯一的key.这个prop用于指定该key的长度，默认情况下为6长度 |
| History | 指定自定义的history对象 |

**🡪<Route>所渲染的组件，获取路由对象的不同情况。**

**情况1**：当<Route>组件指定component属性,并且当URL match该<Route>path属性，将渲染出对应的组件。该组件会收到由<Route>组件提供的match对象, location对象, history对象这三个props.这些属性用于表达路由系统状态。我们可从这些路由系统状态中获取变量，用于未来导航等等。

例如：

<Route path=”/products” component={ProductDisplay} />

则在ProductDisplay组件中，将可以使用this.props.match, this.props.history, this.props.location.

**情况2**： 当<Route>组件指定render属性，并且当URL匹配该<Route>path属性，使用render属性返回的内容作为渲染出的内容。该方法将获取一个路由对象，该路由对象的属性含有match, location, history对象。

例如：

<Route path=”/products” render = {(routeProps) => 可访问routeProps.match, routeProps.location, routeProps.history }

**情况3：**

当<Route>组件指定children属性，并且无论URL是否匹配该<Route>path属性，使用children属性返回的内容作为渲染出的内容。该方法将获取一个路由对象，该路由对象的属性含有match, location, history对象。

当前URL匹配该<Route>path属性时候，路由对象的match属性有值。

当前URL不匹配该<Route>path属性时候，路由对象的match属性为undefined。

例如

<Route path={this.props.to} exact={this.props.exact}

children = {routeProps => {

const baseClasses = this.props.className || "m-2 btn btn-block";

const activeClass = this.props.activeClass || "btn-primary";

const inActiveClass = this.props.inActiveClass || "btn-secondary";

const combinedClasses = `${baseClasses}${routeProps.match?activeClass : inActiveClass}`;

return <Link to={this.props.to} className={combinedClasses}>{this.props.children}</Link>

// {this.props.children>为以后使用<ToggleLink>时中间所存在的所有内容

|  |  |
| --- | --- |
| **Props** | **描述** |
| Match对象 | 表示<Route>组件是如何匹配当前浏览器URL |
| location对象 | 表示当前的URL地址信息 |
| History对象 | 提供导航API的对象 |

Match对象的属性：

|  |  |
| --- | --- |
| **Name** | **Description** |
| url | 返回被<Route>组件所匹配的Url |
| path | 返回<Route>中用于匹配url的path值 |
| params | 返回url中所带的参数，常用于将部分url映射成变量 |
| isExact | 返回<Route>组件的path是否完全匹配了当前URL地址 |

Location对象的属性：

|  |  |
| --- | --- |
| **Name** | **Description** |
| key | 返回一个key用于识别该location |
| pathname | 返回location的路径 |
| search | 返回在URL地址中?后面的搜索项 |
| hash | 返回URL地址中#后面搜索项 |
| state |  |

并且Location对象还可以直接放到<Link>,<NavLink>的To prop中用以导航。

History对象的属性：

|  |  |
| --- | --- |
| **Name** | **Description** |
| push(path) | 导航到指定path所在URL,并且把跳转前URL添加到浏览器的历史记录 |
| replace(path) | 导航到指定path所在的URL,并且把跳转前URL替换浏览器当前最新的历史记录 |
| goBack() | 导航到浏览器历史记录上一条记录 |
| goForward() | 导航到浏览器记录下一条记录 |
| go(n) | 从当前位置，去到浏览器第n位。如果n位正数，则向前走，n位负数，向后走 |
| block(prompt) | 阻塞导航，直到用户responds to a prompt |

**🡪使用Match对象的params属性获取URL中的变量**

例如：

<NavLink className=” …” activeClassName=”…” to=”/info/location”>Location</NavLink>

<Route path=”info/: datatype” component={RouteInfo} />

点击<NavLink>组件时候浏览器跳转到”/info/location”, <Route path=”info/:datatype” component={RouteInfo} />组件匹配该url并且渲染出RouteInfo组件，并且将为RouteInfo组件添加this.props. match, this.props.location, this.props.history属性。由于在<Route>组件中通过: datatype申明URL变量，所以在渲染出的组件RouteInfo中，可以通过this.props.match.params.datatype获取。

P.s: 参数名 表示此参数必须存在于URL中，如果<Route>中path没有此参数则无法匹配。

P.s: 参数名? 在最后加上?, 表示此参数为可选参数。可以存在或不存在于URL中，如果存在则获取值，不存在则也匹配但值为undefined.

P.s: 参数名(值) 表示该一个命名参数，并且值必须为所指定的值。例如：

<Route key=”AAA” path=”/:dataType(AAA)/:modes? Component={…} />表示第一个参数命名为dataType并且值必须为AAA才匹配。

**非匹配URL组件接受路由状态,以及非路由系统内组件接受路由状态对象**

在使用<Route path=”…” component=”…”/>时候，如果当前URL与<Route>中的path匹配时候，<Route>组件将传递match, location, history给组件。但是如果当前URL与<Route>中的path不匹配，或者控件不由<Route>激活时候，默认情况下这些组件不会收到路由状态对象。也就是不会存在match, history, location对象。有时候我们在这些控件中也要根据路由状态来进行逻辑，例如显示路由状态等等。则我们需要为这些组件注入路由状态.

**🡪非URL匹配下，组件接收路由状态对象**

一般情况下，我们定义<Route path=”…” component=”…” />， 当URL匹配Route的path时，渲染出组件。并且<Route>组件会为这样匹配渲染出的组件添加Match, Location, History的路由属性(Props). 但是对于此时，其他非匹配状态的组件无法获取路由状态。

如果我们的需求是，想为某些组件在非URL匹配的状态下，也要获取路由状态去做某些逻辑。则要求这些组件在URL非匹配状态下也能获取路由状态。此时需要使用<Route>组件的children属性。

例如：

export class ToggleLink extends Component {

render(){

return <Route path={this.props.to} exact={this.props.exact}

children = {routeProps => {

const baseClasses = this.props.className || "m-2 btn btn-block";

const activeClass = this.props.activeClass || "btn-primary";

const inActiveClass = this.props.inActiveClass || "btn-secondary";

const combinedClasses = `${baseClasses}${routeProps.match?activeClass : inActiveClass}`;

return <Link to={this.props.to} className={combinedClasses}>{this.props.children}</Link>

// {this.props.children>为以后使用<ToggleLink>时中间所存在的所有内容

}}

/>

}

}

上面例子中，我们使用<Route>组件并且使用children属性。那么该<Route>组件无论URL是否匹配都会最终渲染出<Link>组件。并且由于children所指定的方法接受一个代表路由状态对象为参数，我们可以根据routeProps.match是否有定义才判断此时该<Link>是否处于激活状态而改变样式。

App.js使用时：

<ToggleLink to="/products">Products</ToggleLink>

<ToggleLink to="/suppliers">Suppliers</ToggleLink>

<ToggleLink to="/info/match">Match</ToggleLink>

从而ToggleLink组件将渲染一个<Link>组件，而在<Route>组件使用children属性，将最终渲染出一个超链接。当浏览器url为/products时，<ToggleLink to="/products">Products</ToggleLink>处于激活状态，routeProps.match存在，表现为激活样式。而其他<Link>组件虽然url不匹配，但是仍然可以获取路由状态，表现为非激活样式。

🡪**为非路由系统的组件接受路由状态对象**

例子：

export class Selector extends Component {

render(){

return <Router>

<RouterInfo />

</Router>

}

}

我们有一组件<RouterInfo />，在该组件中显示路由状态信息，所以需要访问match对象，location对象。但是如果

<RouterInfo />组件不经由<Route>组件的component属性生成，也不经过children属性生成，也不经过render属性生成，也就是不存在路由系统内。则<RouterInfo />组件无法获取路由状态对象。

我们可以使用react-router所带的withRouter方法为一个组件进行包裹，使得该组件在脱离路由系统的情况下也可以获取路由状态对象。

例如:

const RouteInfoHOC = withRouter(RouterInfo); //使用withRouter对RouterInfo组件进行包裹，形成一个HOC组件RouteInfoHOC,渲染RouterInfoHOC。RouterInfoHOC这个组件则可以获取match, location ,history对象。并且由于HOC组件特性，HOC组件将自身props传给其下层组件，也就是RouterInfo,所以在RouterInfo组件中可以获取match, location, history组件。

export class Selector extends Component {

render(){

return <Router>

<RouteInfoHOC />

</Router>

}

}

P.s ***虽然通过withRouter方式，在组件中可以获取match, location, history路由对象。但于<Route component=”” path=””>相比，通过withRouter方式在组件中获取的路由对象的match对象属性值为空。***