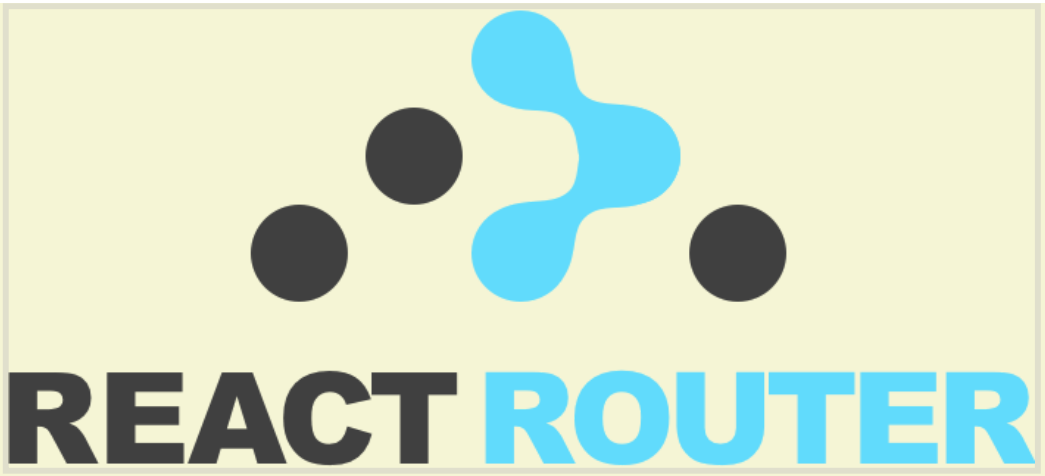
真正学会 React 是一个漫长的过程。

你会发现，它不是一个库，也不是一个框架，而是一个庞大的体系。想要发挥它的威力，整个技术栈都要配合它改造。你要学习一整套解决方案，从后端到前端，都是全新的做法。

举例来说，React 不使用 HTML，而使用 JSX 。它打算抛弃 DOM，要求开发者不要使用任何 DOM 方法。它甚至还抛弃了 SQL ，自己发明了一套查询语言 GraphQL 。当然，这些你都可以不用，React 照样运行，但是就发挥不出它的最大威力。

这样说吧，你只要用了 React，就会发现合理的选择就是，采用它的整个技术栈。

本文介绍 React 体系的一个重要部分：路由库 React-Router 。它是官方维护的，事实上也是唯一可选的路由库。它通过管理 URL，实现组件的切换和状态的变化，开发复杂的应用几乎肯定会用到。



本文针对初学者，尽量写得简洁易懂。预备知识是 React 的基本用法，可以参考我写的《React 入门实例教程》。

另外，我没有准备示例库，因为官方的示例库非常棒，由浅入深，分成14步，每一步都有详细的代码解释。我强烈建议你先跟着做一遍，然后再看下面的API讲解。

一、基本用法

React Router 安装命令如下。

```
$ npm install -S react-router
```

使用时，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}/>
```

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。

设置路径参数时，需要特别小心这一点。

```
<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` 。

{this.props.children ||

因此，通常会采用 <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、histroy 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。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。

（完）