[toc]

React Router学习

introduction

安装

首先通过 npm 安装：

```
$ npm install --save react-router
```

然后使用一个支持 CommonJS 或 ES2015 的模块管理器，例如 webpack：

```
// 使用 ES6 的转译器，如 babel
import { Router, Route, Link } from 'react-router'

// 不使用 ES6 的转译器
var ReactRouter = require('react-router')
var Router = ReactRouter.Router
var Route = ReactRouter.Route
var Link = ReactRouter.Link
```

也可以在 unpkg 上构建 UMD 格式：

```
<script src="https://unpkg.com/react-router/umd/ReactRouter.min.js"></script>
```

你可以在 `window.ReactRouter` 找到这个库。

版本控制和稳定性

React Router 遵循语义化版本控制，并很好地诠释了它。我们希望 React Router 是一个稳定的依赖库，这样易于保持流行性。这是我们对你的应用的升级策略。

假设我们目前是 1.0 版本：

1. 2.0 完全向后兼容 1.0，所以你可以放心地升级，然后逐步更新你的代码。
2. 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，并链接到升级指南。
3. 在 3.0 中将会完全移除 1.0 所弃用的东西。
4. 3.0 将发布不早于 2.0 三个月后。最坏的情况是，给你一个新的 API，你需要花费 6 个月的时间去完美升级。
5. 可以用 rackt/rackt-codemod 去自动升级你的代码

> 如果是完全向后兼容的，为什么这不是一个小版本呢？

如果我们不提供向后兼容性，然后你就不会问这个问题 —— 升级后的应用将不可运行。这不是我们想要的结果，我们想要平稳地，逐步地升级。

在实践中，这意味着你可以：

1. 从 1.0 升级到 2.0，你的应用仍可以运行。
2. 逐步更新你的代码到新的 API，在下一个版本发布之前，你有 3 个月的时间去完成。
3. 运行 codemods 去处理自动运行 (2) 部分。
4. 如果您的代码运行没有警告，你可以使用 3.0 版本重复这个列表

这看起来像什么?

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

const App = React.createClass({/*...*/})
const About = React.createClass({/*...*/})
// 等等。

const Users = React.createClass({
  render() {
    return (
      <div>
        <h1>Users</h1>
        <div className="master">
          <ul>
            {/* 在应用中用 Link 去链接路由 */}
            {this.state.users.map(user => (
              <li key={user.id}><Link to={`/user/${user.id}`}>{user.name}</Link>
</li>

            ))}
          </ul>
        </div>
        <div className="detail">
          {this.props.children}
        </div>
      </div>
    )
  }
})

const User = React.createClass({
  componentDidMount() {
    this.setState({
      // 路由应该通过有用的信息来呈现，例如 URL 的参数
      user: findUserById(this.props.params.userId)
    })
  },

  render() {
    return (
      <div>
```

```
        <h2>{this.state.user.name}</h2>
        {/* 等等。 */}
      </div>
    )
  }
})

// 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)
```

简介

React Router 是一个基于 React 之上的强大路由库，它可以让你向应用中快速地添加视图和数据流，同时保持页面与 URL 间的同步。

为了向你说明 React Router 解决的问题，让我们先来创建一个不使用它的应用。所有文档中的示例代码都会使用 ES6/ES2015 语法和语言特性。

不使用 React Router

```
import React from 'react'
import { render } from 'react-dom'

const About = React.createClass({/*...*/})
const Inbox = React.createClass({/*...*/})
const Home = React.createClass({/*...*/})

const App = React.createClass({
  getInitialState() {
    return {
      route: window.location.hash.substr(1)
    }
  },

  componentDidMount() {
    window.addEventListener('hashchange', () => {
      this.setState({
        route: window.location.hash.substr(1)
      })
```

```
    })
  },

  render() {
    let Child
    switch (this.state.route) {
      case '/about': Child = About; break;
      case '/inbox': Child = Inbox; break;
      default:      Child = Home;
    }

    return (
      <div>
        <h1>App</h1>
        <ul>
          <li><a href="#/about">About</a></li>
          <li><a href="#/inbox">Inbox</a></li>
        </ul>
        <Child/>
      </div>
    )
  }
})

React.render(<App />, document.body)
```

当 URL 的 hash 部分（指的是 # 后的部分）变化后，`<App>` 会根据 `this.state.route` 来渲染不同的 `<Child>`。看起来很直接，但它很快就会变得复杂起来。

现在设想一下 `Inbox` 下面嵌套一些分别对应于不同 URL 的 UI 组件，就像下面这样的**列表-详情**视图：

```
path: /inbox/messages/1234

+---------+------------+------------------------+
| About   |    Inbox   |                        |
+---------+            +------------------------+
| Compose    Reply    Reply All    Archive      |
+-----------------------------------------------+
|Movie tomorrow|                                |
+--------------+   Subject: TPS Report          |
|TPS Report        From:    boss@big.co         |
+--------------+                                |
|New Pull Reque|   So ...                       |
+--------------+                                |
|...           |                                |
+--------------+--------------------------------+
```

还可能有一个状态页，用于在没有选择 message 时展示：

```
path: /inbox

+---------+------------+------------------------+
| About   |    Inbox   |                        |
+---------+            +------------------------+
| Compose    Reply    Reply All    Archive      |
+-----------------------------------------------+
|Movie tomorrow|                                |
+--------------+   10 Unread Messages           |
|TPS Report    |   22 drafts                    |
+--------------+                                |
|New Pull Reque|                                |
+--------------+                                |
|...           |                                |
+--------------+--------------------------------+
```

为了让我们的 URL 解析变得更智能，我们需要编写很多代码来实现指定 URL 应该渲染哪一个嵌套的 UI 组件分支：`App -> About`, `App -> Inbox -> Messages -> Message`, `App -> Inbox -> Messages -> Stats`, 等等。

使用 React Router 后

让我们用 React Router 重构这个应用。

```
import React from 'react'
import { render } from 'react-dom'

// 首先我们需要导入一些组件...
import { Router, Route, Link } from 'react-router'

// 然后我们从应用中删除一堆代码和
// 增加一些 <Link> 元素...
const App = React.createClass({
  render() {
    return (
      <div>
        <h1>App</h1>
        {/* 把 <a> 变成 <Link> */}
        <ul>
          <li><Link to="/about">About</Link></li>
          <li><Link to="/inbox">Inbox</Link></li>
        </ul>

        {/*
          接着用 `this.props.children` 替换 `<Child>`
          router 会帮我们找到这个 children
        */}
        {this.props.children}
      </div>
    )
```

```
  }
})

// 最后，我们用一些 <Route> 来渲染 <Router>。
// 这些就是路由提供的我们想要的东西。
React.render((
  <Router>
    <Route path="/" component={App}>
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox} />
    </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>`。

在内部，router 会将你树级嵌套格式的 `<Route>` 转变成路由配置。但如果你不熟悉 JSX，你也可以用普通对象来替代：

```
const routes = {
  path: '/',
  component: App,
  childRoutes: [
    { path: 'about', component: About },
    { path: 'inbox', component: Inbox },
  ]
}

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

添加更多的 UI

好了，现在我们准备在 inbox UI 内嵌套 inbox messages。

```
// 新建一个组件让其在 Inbox 内部渲染
const Message = React.createClass({
  render() {
    return <h3>Message</h3>
  }
})

const Inbox = React.createClass({
  render() {
    return (
      <div>
        <h2>Inbox</h2>
        {/* 渲染这个 child 路由组件 */}
        {this.props.children || "Welcome to your Inbox"}
```

```
      </div>
    )
  }
})

React.render((
  <Router>
    <Route path="/" component={App}>
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox}>
        {/* 添加一个路由，嵌套进我们想要嵌套的 UI 里 */}
        <Route path="messages/:id" component={Message} />
      </Route>
    </Route>
  </Router>
), document.body)
```

现在访问 URL `inbox/messages/Jkei3c32` 将会匹配到一个新的路由，并且它成功指向了 `App -> Inbox -> Message` 这个 UI 的分支。

```
<App>
  <Inbox>
    <Message params={ {id: 'Jkei3c32'} } />
  </Inbox>
</App>
```

获取 URL 参数

为了从服务器获取 message 数据，我们首先需要知道它的信息。当渲染组件时，React Router 会自动向 Route 组件中注入一些有用的信息，尤其是路径中动态部分的参数。我们的例子中，它指的是 `:id`。

```
const Message = React.createClass({

  componentDidMount() {
    // 来自于路径 `/inbox/messages/:id`
    const id = this.props.params.id

    fetchMessage(id, function (err, message) {
      this.setState({ message: message })
    })
  },

  // ...

})
```

你也可以通过 query 字符串来访问参数。比如你访问 `/foo?bar=baz`，你可以通过访问 `this.props.location.query.bar` 从 Route 组件中获得 `"baz"` 的值。

这就是 React Router 的奥秘。应用的 UI 以盒子中嵌套盒子的方式来表现；然后你可以让这些盒子与 URL 始终保持同步，而且很容易地把它们链接起来。

这个关于 路由配置 的文档深入地描述了 router 的功能。

基础

路由配置

路由配置是一组指令，用来告诉 router 如何匹配 URL以及匹配后如何执行代码。我们来通过一个简单的例子解释一下如何编写路由配置。

```
import React from 'react'
import { render } from 'react-dom'
import { Router, Route, Link } from 'react-router'

const App = React.createClass({
  render() {
    return (
      <div>
        <h1>App</h1>
        <ul>
          <li><Link to="/about">About</Link></li>
          <li><Link to="/inbox">Inbox</Link></li>
        </ul>
        {this.props.children}
      </div>
    )
  }
})

const About = React.createClass({
  render() {
    return <h3>About</h3>
  }
})

const Inbox = React.createClass({
  render() {
    return (
      <div>
        <h2>Inbox</h2>
        {this.props.children || "Welcome to your Inbox"}
      </div>
    )
  }
})

const Message = React.createClass({
```

```
  render() {
    return <h3>Message {this.props.params.id}</h3>
  }
})

render((
  <Router>
    <Route path="/" component={App}>
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox}>
        <Route path="messages/:id" component={Message} />
      </Route>
    </Route>
  </Router>
), document.body)
```

通过上面的配置，这个应用知道如何渲染下面四个 URL：

URL	组件
/	App
/about	App -> About
/inbox	App -> Inbox
/inbox/messages/:id	App -> Inbox -> Message

添加首页

想象一下当 URL 为 / 时，我们想渲染一个在 App 中的组件。不过在此时，App 的 render 中的 this.props.children 还是 undefined。这种情况我们可以使用 IndexRoute 来设置一个默认页面。

```
import { IndexRoute } from 'react-router'

const Dashboard = React.createClass({
  render() {
    return <div>Welcome to the app!</div>
  }
})

render((
  <Router>
    <Route path="/" component={App}>
      {/* 当 url 为/时渲染 Dashboard */}
      <IndexRoute component={Dashboard} />
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox}>
        <Route path="messages/:id" component={Message} />
      </Route>
    </Route>
```

```
    </Router>
  ), document.body)
```

现在，App 的 render 中的 this.props.children 将会是 <Dashboard>这个元素。这个功能类似 Apache 的 DirectoryIndex 以及 nginx的 index指令，上述功能都是在当请求的 URL 匹配某个目录时，允许你制定一个类似index.html的入口文件。

我们的 sitemap 现在看起来如下：

URL	组件
/	App -> Dashboard
/about	App -> About
/inbox	App -> Inbox
/inbox/messages/:id	App -> Inbox -> Message

让 UI 从 URL 中解耦出来

如果我们可以将 /inbox 从 /inbox/messages/:id 中去除，并且还能够让 Message 嵌套在 App -> Inbox 中渲染，那会非常赞。绝对路径可以让我们做到这一点。

```
render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Dashboard} />
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox}>
        {/* 使用 /messages/:id 替换 messages/:id */}
        <Route path="/messages/:id" component={Message} />
      </Route>
    </Route>
  </Router>
), document.body)
```

在多层嵌套路由中使用绝对路径的能力让我们对 URL 拥有绝对的掌控。我们无需在 URL 中添加更多的层级，从而可以使用更简洁的 URL。

我们现在的 URL 对应关系如下：

URL	组件
/	App -> Dashboard
/about	App -> About
/inbox	App -> Inbox
/messages/:id	App -> Inbox -> Message

提醒：绝对路径可能在动态路由中无法使用。

兼容旧的 URL

等一下，我们刚刚改变了一个 URL! 这样不好。 现在任何人访问 `/inbox/messages/5` 都会看到一个错误页面。😦

不要担心。我们可以使用 `` 使这个 URL 重新正常工作。

```
import { Redirect } from 'react-router'

render((
  <Router>
    <Route path="/" component={App}>
      <IndexRoute component={Dashboard} />
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox}>
        <Route path="/messages/:id" component={Message} />

        {/* 跳转 /inbox/messages/:id 到 /messages/:id */}
        <Redirect from="messages/:id" to="/messages/:id" />
      </Route>
    </Route>
  </Router>
), document.body)
```

现在当有人点击 `/inbox/messages/5` 这个链接，他们会被自动跳转到 `/messages/5`。🙌

进入和离开的Hook

Route 可以定义 `onEnter` 和 `onLeave` 两个 hook ，这些hook会在页面跳转确认时触发一次。这些 hook 对于一些情况非常的有用，例如权限验证或者在路由跳转前将一些数据持久化保存起来。

在路由跳转过程中，`onLeave` hook 会在所有将离开的路由中触发，从最下层的子路由开始直到最外层父路由结束。然后`onEnter` hook会从最外层的父路由开始直到最下层子路由结束。

继续我们上面的例子，如果一个用户点击链接，从 `/messages/5` 跳转到 `/about`，下面是这些 hook 的执行顺序：

- `/messages/:id` 的 `onLeave`
- `/inbox` 的 `onLeave`
- `/about` 的 `onEnter`

替换的配置方式

因为 route 一般被嵌套使用，所以使用 JSX 这种天然具有简洁嵌套型语法的结构来描述它们的关系非常方便。然而，如果你不想使用 JSX，也可以直接使用原生 route 数组对象。

上面我们讨论的路由配置可以被写成下面这个样子：

```
const routeConfig = [
  { path: '/',
    component: App,
    indexRoute: { component: Dashboard },
    childRoutes: [
      { path: 'about', component: About },
      { path: 'inbox',
        component: Inbox,
        childRoutes: [
          { path: '/messages/:id', component: Message },
          { path: 'messages/:id',
            onEnter: function (nextState, replaceState) {
              replaceState(null, '/messages/' + nextState.params.id)
            }
          }
        ]
      }
    ]
  }
]

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

路由匹配原理

路由拥有三个属性来决定是否“匹配”一个 URL：

1. 嵌套关系 和
2. 它的 路径语法
3. 它的 优先级

嵌套关系

React Router 使用路由嵌套的概念来让你定义 view 的嵌套集合，当一个给定的 URL 被调用时，整个集合中（命中的部分）都会被渲染。嵌套路由被描述成一种树形结构。React Router 会深度优先遍历整个路由配置来寻找一个与给定的 URL 相匹配的路由。

路径语法

路由路径是匹配一个（或一部分）URL 的 一个字符串模式。大部分的路由路径都可以直接按照字面量理解，除了以下几个特殊的符号：

- `: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
- `()` – 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
- `**` - 匹配任意字符（贪婪的）直到命中下一个字符 `/`、`?` 或 `#`，并创建一个 `splat` 参数

```
<Route path="/hello/:name">         // 匹配 /hello/michael 和 /hello/ryan
<Route path="/hello(/:name)">       // 匹配 /hello, /hello/michael 和 /hello/ryan
```

```
<Route path="/files/*.*">           // 匹配 /files/hello.jpg 和
/files/path/to/hello.jpg
```

如果一个路由使用了相对路径，那么完整的路径将由它的所有祖先节点的路径和自身指定的相对路径拼接而成。使用绝对路径可以使路由匹配行为忽略嵌套关系。

优先级

最后，路由算法会根据定义的顺序自顶向下匹配路由。因此，当你拥有两个兄弟路由节点配置时，你必须确认前一个路由不会匹配后一个路由中的路径。例如，千万**不要**这么做：

```
<Route path="/comments" ... />
<Redirect from="/comments" ... />
```

Histories

React Router 是建立在 history 之上的。 简而言之，一个 history 知道如何去监听浏览器地址栏的变化， 并解析这个 URL 转化为 `location` 对象， 然后 router 使用它匹配到路由，最后正确地渲染对应的组件。

常用的 history 有三种形式， 但是你也可以使用 React Router 实现自定义的 history。

- `browserHistory`
- `hashHistory`
- `createMemoryHistory`

你可以从 React Router 中引入它们：

```
// JavaScript 模块导入（译者注：ES6 形式）
import { browserHistory } from 'react-router'
```

然后将它们传递给`<Router>`：

```
render(
  <Router history={browserHistory} routes={routes} />,
  document.getElementById('app')
)
```

`browserHistory`

Browser history 是使用 React Router 的应用推荐的 history。它使用浏览器中的 History API 用于处理 URL，创建一个像`example.com/some/path`这样真实的 URL 。

服务器配置

服务器需要做好处理 URL 的准备。处理应用启动最初的 `/` 这样的请求应该没问题，但当用户来回跳转并在 `/accounts/123` 刷新时，服务器就会收到来自 `/accounts/123` 的请求，这时你需要处理这个 URL 并在响应中包含 JavaScript 应用代码。

一个 express 的应用可能看起来像这样的：

```
const express = require('express')
const path = require('path')
const port = process.env.PORT || 8080
const app = express()

// 通常用于加载静态资源
app.use(express.static(__dirname + '/public'))

// 在你应用 JavaScript 文件中包含了一个 script 标签
// 的 index.html 中处理任何一个 route
app.get('*', function (request, response){
  response.sendFile(path.resolve(__dirname, 'public', 'index.html'))
})

app.listen(port)
console.log("server started on port " + port)
```

如果你的服务器是 nginx，请使用 `try_files` 指令：

```
server {
  ...
  location / {
    try_files $uri /index.html
  }
}
```

当在服务器上找不到其他文件时，这可以让 nginx 服务器提供静态文件服务并指向`index.html` 文件。

对于Apache服务器也有类似的方式，创建一个`.htaccess`文件在你的文件根目录下：

```
RewriteBase /
RewriteRule ^index\.html$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.html [L]
```

IE8, IE9 支持情况

如果我们能使用浏览器自带的 `window.history` API，那么我们的特性就可以被浏览器所检测到。如果不能，那么任何调用跳转的应用就会导致 **全页面刷新**，它允许在构建应用和更新浏览器时会有一个更好的用户体验，

但仍然支持的是旧版的。

你可能会想为什么我们不后退到 hash history，问题是这些 URL 是不确定的。如果一个访客在 hash history 和 browser history 上共享一个 URL，然后他们也共享同一个后退功能，最后我们会以产生笛卡尔积数量级的、无限多的 URL 而崩溃。

`hashHistory`

Hash history 使用 URL 中的 hash（`#`）部分去创建形如 `example.com/#/some/path` 的路由。

我应该使用 `createHashHistory`吗?

Hash history 不需要服务器任何配置就可以运行，如果你刚刚入门，那就使用它吧。但是我们不推荐在实际线上环境中用到它，因为每一个 web 应用都应该渴望使用 `browserHistory`。

像这样 `?_k=ckuvup` 没用的在 URL 中是什么?

当一个 history 通过应用程序的 `push` 或 `replace` 跳转时，它可以在新的 location 中存储 “location state” 而不显示在 URL 中，这就像是在一个 HTML 中 post 的表单数据。

在 DOM API 中，这些 hash history 通过 `window.location.hash = newHash` 很简单地被用于跳转，且不用存储它们的location state。但我们想全部的 history 都能够使用location state，因此我们要为每一个 location 创建一个唯一的 key，并把它们的状态存储在 session storage 中。当访客点击“后退”和“前进”时，我们就会有一个机制去恢复这些 location state。

`createMemoryHistory`

Memory history 不会在地址栏被操作或读取。这就解释了我们是如何实现服务器渲染的。同时它也非常适合测试和其他的渲染环境（像 React Native ）。

和另外两种history的一点不同是你必须创建它，这种方式便于测试。

```
const history = createMemoryHistory(location)
```

实现示例

```
import React from 'react'
import { render } from 'react-dom'
import { browserHistory, Router, Route, IndexRoute } from 'react-router'

import App from '../components/App'
import Home from '../components/Home'
import About from '../components/About'
import Features from '../components/Features'

render(
  <Router history={browserHistory}>
    <Route path='/' component={App}>
```

```
        <IndexRoute component={Home} />
        <Route path='about' component={About} />
        <Route path='features' component={Features} />
      </Route>
    </Router>,
    document.getElementById('app')
  )
```

默认路由（IndexRoute）与 IndexLink

默认路由（IndexRoute）

在解释 默认路由(IndexRoute) 的用例之前，我们来设想一下，一个不使用默认路由的路由配置是什么样的：

```
<Router>
  <Route path="/" component={App}>
    <Route path="accounts" component={Accounts}/>
    <Route path="statements" component={Statements}/>
  </Route>
</Router>
```

当用户访问 / 时, App 组件被渲染，但组件内的子元素却没有， App 内部的 this.props.children 为 undefined 。 你可以简单地使用 `{this.props.children ||

}` 来渲染一些默认的 UI 组件。

但现在，Home 无法参与到比如 onEnter hook 这些路由机制中来。 在 Home 的位置，渲染的是 Accounts 和 Statements。 由此，router 允许你使用 IndexRoute ，以使 Home 作为最高层级的路由出现.

```
<Router>
  <Route path="/" component={App}>
    <IndexRoute component={Home}/>
    <Route path="accounts" component={Accounts}/>
    <Route path="statements" component={Statements}/>
  </Route>
</Router>
```

现在 App 能够渲染 {this.props.children} 了， 我们也有了一个最高层级的路由，使 Home 可以参与进来。

Index Links

如果你在这个 app 中使用 <Link to="/">Home</Link> , 它会一直处于激活状态，因为所有的 URL 的开头都是 / 。 这确实是个问题，因为我们仅仅希望在 Home 被渲染后，激活并链接到它。

如果需要在 Home 路由被渲染后才激活的指向 / 的链接，请使用 <IndexLink to="/">Home</IndexLink>

高级用法

动态路由

React Router 适用于小型网站，比如 React.js Training，也可以支持 Facebook 和 Twitter 这类大型网站。

对于大型应用来说，一个首当其冲的问题就是所需加载的 JavaScript 的大小。程序应当只加载当前渲染页所需的 JavaScript。有些开发者将这种方式称之为"代码分拆" —— 将所有的代码分拆成多个小包，在用户浏览过程中按需加载。

对于底层细节的修改不应该需要它上面每一层级都进行修改。举个例子，为一个照片浏览页添加一个路径不应该影响到首页加载的 JavaScript 的大小。也不能因为多个团队共用一个大型的路由配置文件而造成合并时的冲突。

路由是个非常适于做代码分拆的地方：它的责任就是配置好每个 view。

React Router 里的路径匹配以及组件加载都是异步完成的，不仅允许你延迟加载组件，**并且可以延迟加载路由配置**。在首次加载包中你只需要有一个路径定义，路由会自动解析剩下的路径。

Route 可以定义 `getChildRoutes`，`getIndexRoute` 和 `getComponents` 这几个函数。它们都是异步执行，并且只有在需要时才被调用。我们将这种方式称之为 "逐渐匹配"。 React Router 会逐渐的匹配 URL 并只加载该 URL 对应页面所需的路径配置和组件。

如果配合 webpack 这类的代码分拆工具使用的话，一个原本繁琐的构架就会变得更简洁明了。

```
const CourseRoute = {
  path: 'course/:courseId',

  getChildRoutes(location, callback) {
    require.ensure([], function (require) {
      callback(null, [
        require('./routes/Announcements'),
        require('./routes/Assignments'),
        require('./routes/Grades'),
      ])
    })
  },

  getIndexRoute(location, callback) {
    require.ensure([], function (require) {
      callback(null, {
        component: require('./components/Index')
      })
    })
  },

  getComponents(location, callback) {
    require.ensure([], function (require) {
      callback(null, require('./components/Course'))
    })
  }
}
```

现在，可以看一下 webpack 都做了哪些。开玩笑，我现在不想让你伤心。

运行 huge apps 实例，打开浏览器的审查元素选项。你会发现在路由发生改变时，资源按需加载。

跳转前确认

React Router 提供一个 `routerWillLeave` 生命周期钩子，这使得 React 组件可以拦截正在发生的跳转，或在离开 route 前提示用户。`routerWillLeave` 返回值有以下两种：

1. `return false` 取消此次跳转
2. `return` 返回提示信息，在离开 route 前提示用户进行确认。

你可以在 route 组件 中引入 `Lifecycle` mixin 来安装这个钩子。

```
import { Lifecycle } from 'react-router'

const Home = React.createClass({

  // 假设 Home 是一个 route 组件，它可能会使用
  // Lifecycle mixin 去获得一个 routerWillLeave 方法。
  mixins: [ Lifecycle ],

  routerWillLeave(nextLocation) {
    if (!this.state.isSaved)
      return 'Your work is not saved! Are you sure you want to leave?'
  },

  // ...

})
```

如果你在组件中使用了 ES6 类，你可以借助 react-mixin 包将 `Lifecycle` mixin 添加到组件中，不过我们推荐使用 `React.createClass` 来创建组件，初始化路由的生命周期钩子函数。

如果你想在一个深层嵌套的组件中使用 `routerWillLeave` 钩子，只需在 route 组件 中引入 `RouteContext` mixin，这样就会把 `route` 放到 context 中。

```
import { Lifecycle, RouteContext } from 'react-router'

const Home = React.createClass({

  // route 会被放到 Home 和它子组件及孙子组件的 context 中，
  // 这样在层级树中 Home 及其所有子组件都可以拿到 route。
  mixins: [ RouteContext ],

  render() {
    return <NestedForm />
  }

})
```

```
const NestedForm = React.createClass({

  // 后代组件使用 Lifecycle mixin 获得
  // 一个 routerWillLeave 的方法。
  mixins: [ Lifecycle ],

  routerWillLeave(nextLocation) {
    if (!this.state.isSaved)
      return 'Your work is not saved! Are you sure you want to leave?'
  },

  // ...

})
```

服务端渲染

服务端渲染与客户端渲染有些许不同，因为你需要：

- 发生错误时发送一个 `500` 的响应
- 需要重定向时发送一个 `30x` 的响应
- 在渲染之前获得数据 (用 router 帮你完成这点)

为了迎合这一需求，你要在 `` API 下一层使用：

- 使用 `match` 在渲染之前根据 location 匹配 route
- 使用 `RoutingContext` 同步渲染 route 组件

它看起来像一个虚拟的 JavaScript 服务器：

```
import { renderToString } from 'react-dom/server'
import { match, RoutingContext } from 'react-router'
import routes from './routes'

serve((req, res) => {
  // 注意! 这里的 req.url 应该是从初始请求中获得的
  // 完整的 URL 路径，包括查询字符串。
  match({ routes, location: req.url }, (error, redirectLocation, renderProps) => {
    if (error) {
      res.send(500, error.message)
    } else if (redirectLocation) {
      res.redirect(302, redirectLocation.pathname + redirectLocation.search)
    } else if (renderProps) {
      res.send(200, renderToString(<RoutingContext {...renderProps} />))
    } else {
      res.send(404, 'Not found')
    }
  })
})
```

至于加载数据，你可以用 `renderProps` 去构建任何你想要的形式——例如在 route 组件中添加一个静态的 `load` 方法，或如在 route 中添加数据加载的方法——由你决定。

组件生命周期

在开发应用时，理解路由组件的生命周期是非常重要的。 后面我们会以获取数据这个最常见的场景为例，介绍一下路由改变时，路由组件生命周期的变化情况。

路由组件的生命周期和 React 组件相比并没有什么不同。 所以让我们先忽略路由部分，只考虑在不同 URL 下，这些组件是如何被渲染的。

路由配置如下：

```
<Route path="/" component={App}>
  <IndexRoute component={Home}/>
  <Route path="invoices/:invoiceId" component={Invoice}/>
  <Route path="accounts/:accountId" component={Account}/>
</Route>
```

路由切换时，组件生命周期的变化情况

1. 当用户打开应用的 '/' 页面

组件	生命周期
App	`componentDidMount`
Home	`componentDidMount`
Invoice	N/A
Account	N/A

2. 当用户从 '/' 跳转到 '/invoice/123'

组件	生命周期
App	`componentWillReceiveProps`, `componentDidUpdate`
Home	`componentWillUnmount`
Invoice	`componentDidMount`
Account	N/A

- `App` 从 router 中接收到新的 props（例如 `children`、`params`、`location` 等数据), 所以 `App` 触发了 `componentWillReceiveProps` 和 `componentDidUpdate` 两个生命周期方法
- `Home` 不再被渲染，所以它将被移除
- `Invoice` 首次被挂载

3. 当用户从 `/invoice/123` 跳转到 `/invoice/789`

组件	生命周期
App	componentWillReceiveProps, componentDidUpdate
Home	N/A
Invoice	componentWillReceiveProps, componentDidUpdate
Account	N/A

所有的组件之前都已经被挂载， 所以只是从 router 更新了 props.

4. 当从 `/invoice/789` 跳转到 `/accounts/123`

组件	生命周期
App	componentWillReceiveProps, componentDidUpdate
Home	N/A
Invoice	componentWillUnmount
Account	componentDidMount

获取数据

虽然还有其他通过 router 获取数据的方法， 但是最简单的方法是通过组件生命周期 Hook 来实现。 前面我们已经理解了当路由改变时组件生命周期的变化， 我们可以在 `Invoice` 组件里实现一个简单的数据获取功能。

```
let Invoice = React.createClass({

  getInitialState () {
    return {
      invoice: null
    }
  },

  componentDidMount () {
    // 上面的步骤2，在此初始化数据
    this.fetchInvoice()
  },

  componentDidUpdate (prevProps) {
    // 上面步骤3，通过参数更新数据
    let oldId = prevProps.params.invoiceId
    let newId = this.props.params.invoiceId
    if (newId !== oldId)
      this.fetchInvoice()
  },

  componentWillUnmount () {
    // 上面步骤四，在组件移除前忽略正在进行中的请求
    this.ignoreLastFetch = true
```

```
  },

  fetchInvoice () {
    let url = `/api/invoices/${this.props.params.invoiceId}`
    this.request = fetch(url, (err, data) => {
      if (!this.ignoreLastFetch)
        this.setState({ invoice: data.invoice })
    })
  },

  render () {
    return <InvoiceView invoice={this.state.invoice}/>
  }

})
```

在组件外部使用导航

虽然在组件内部可以使用 `this.context.router` 来实现导航，但许多应用想要在组件外部使用导航。使用Router组件上被赋予的history可以在组件外部实现导航。

```
// your main file that renders a Router
import { Router, browserHistory } from 'react-router'
import routes from './app/routes'
render(<Router history={browserHistory} routes={routes}/>, el)
// somewhere like a redux/flux action file:
import { browserHistory } from 'react-router'
browserHistory.push('/some/path')
```

排错

如何获得上一次路径?

```
<Route component={App}>
  {/* ... 其它 route */}
</Route>

const App = React.createClass({
  getInitialState() {
    return { showBackButton: false }
  },

  componentWillReceiveProps(nextProps) {
    const routeChanged = nextProps.location !== this.props.location
    this.setState({ showBackButton: routeChanged })
  }
})
```

API 接口

> TODO: Need proofread https://github.com/rackt/react-router/blob/master/docs/API.md

组件

Router

React Router 的重要组件。它能保持 UI 和 URL 的同步。

Props

`children` **(required)**

一个或多个的 `Route` 或 `PlainRoute`。当 history 改变时， `<Router>` 会匹配出 `Route` 的一个分支，并且渲染这个分支中配置的组件，渲染时保持父 route 组件嵌套子 route 组件。

`routes`

`children` 的别名。

`history`

Router 监听的 history 对象，由 `history` 包提供。

`createElement(Component, props)`

当 route 准备渲染 route 组件的一个分支时，就会用这个函数来创建 element。当你使用某种形式的数据进行抽象时，你可以想要获取创建 element 的控制权，例如在这里设置组件监听 store 的变化，或者使用 props 为每个组件传入一些应用模块。

```
<Router createElement={createElement} />

// 默认行为
function createElement(Component, props) {
  // 确保传入了所有的 props!
  return <Component {...props}/>
}

// 你可能会使用什么，如 Relay
function createElement(Component, props) {
  // 确保传入了所有的 props!
  return <RelayContainer Component={Component} routerProps={props}/>
}
```

`stringifyQuery(queryObject)`

一个用于把 `Link` 或调用 `transitionTo` 函数的对象转化成 URL query 字符串的函数。

`parseQueryString(queryString)`

一个用于把 query 字符串转化成对象，并传递给 route 组件 props 的函数。

`onError(error)`

当路由匹配到时，也有可能会抛出错误，此时你就可以捕获和处理这些错误。通常，它们会来自那些异步的特性，如 `route.getComponents`，`route.getIndexRoute`，和 `route.getChildRoutes`。

`onUpdate()`

当 URL 改变时，需要更新路由的 state 时会被调用。

示例

请看仓库中的示例目录 `examples/`，这些例子都广泛使用了 `Router`。

Link

允许用户浏览应用的主要方式。`<Link>` 以适当的 href 去渲染一个可访问的锚标签。

`<Link>` 可以知道哪个 route 的链接是激活状态的，并可以自动为该链接添加 `activeClassName` 或 `activeStyle`。

Props

`to`

跳转链接的路径，如 `/users/123`。

`query`

已经转化成字符串的键值对的对象。

`hash`

URL 的 hash 值，如 `#a-hash`。

注意：React Router 目前还不能管理滚动条的位置，并且不会自动滚动到 hash 对应的元素上。如果需要管理滚动条位置，可以使用 scroll-behavior 这个库。

`state`

保存在 `location` 中的 state。

`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。

`activeStyle`

当某个 route 是激活状态时，可以将样式添加到链接元素上。

`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。

其他

你也可以在 `<a>` 标签上传入一些你想要的 props，如 `title`，`id`，`className` 等等。

示例

如 `<Route path="/users/:userId" />` 这样的 route：

```
<Link to={`/users/${user.id}`} activeClassName="active">{user.name}</Link>
// 变成它们其中一个依赖在 History 上，当这个 route 是
// 激活状态的
<a href="/users/123" class="active">Michael</a>
<a href="#/users/123">Michael</a>

// 修改 activeClassName
<Link to={`/users/${user.id}`} activeClassName="current">{user.name}</Link>

// 当链接激活时，修改它的样式
<Link to="/users" style={{color: 'white'}} activeStyle={{color:
'red'}}>Users</Link>
```

IndexLink

敬请期待!

RoutingContext

在 context 中给定路由的 state、设置 history 对象和当前的 location，`<RoutingContext>` 就会去渲染组件树。

组件的配置

Route

`Route` 是用于声明路由映射到应用程序的组件层。

Props

`path`

URL 中的路径。

它会组合父 route 的路径，除非它是从 `/` 开始的， 将它变成一个绝对路径。

注意：在动态路由中，绝对路径可能不适用于 route 配置中。

如果它是 undefined，路由会去匹配子 route。

`component`

当匹配到 URL 时，单个的组件会被渲染。它可以 被父 route 组件的 `this.props.children` 渲染。

```
const routes = (
  <Route component={App}>
    <Route path="groups" component={Groups}/>
    <Route path="users" component={Users}/>
  </Route>
)

class App extends React.Component {
  render () {
    return (
      <div>
        {/* 这会是 <Users> 或 <Groups> */}
        {this.props.children}
      </div>
    )
  }
}
```

components

Route 可以定义一个或多个已命名的组件，当路径匹配到 URL 时， 它们作为 `name:component` 对的一个对象去渲染。它们可以被 父 route 组件的 `this.props[name]` 渲染。

```
// 想想路由外部的 context — 如果你可拔插
// `render` 的部分，你可能需要这么做：
// <App main={<Users />} sidebar={<UsersSidebar />} />

const routes = (
  <Route component={App}>
    <Route path="groups" components={{main: Groups, sidebar: GroupsSidebar}}/>
    <Route path="users" components={{main: Users, sidebar: UsersSidebar}}>
      <Route path="users/:userId" component={Profile}/>
    </Route>
  </Route>
)

class App extends React.Component {
  render () {
    const { main, sidebar } = this.props
    return (
      <div>
        <div className="Main">
          {main}
        </div>
        <div className="Sidebar">
          {sidebar}
        </div>
      </div>
    )
  }
}

class Users extends React.Component {
  render () {
    return (
      <div>
        {/* 当路径是 "/users/123" 是 `children` 会是 <Profile> */}
        {/* UsersSidebar 也可以获取作为 this.props.children 的 <Profile> ，
            所以这有点奇怪，但你可以决定哪一个可以
            继续这种嵌套 */}
        {this.props.children}
      </div>
    )
  }
}
```

getComponent(location, callback)

与 `component` 一样，但是是异步的，对于 code-splitting 很有用。

`callback` signature

```
cb(err, component)
<Route path="courses/:courseId" getComponent={(location, cb) => {
  // 做一些异步操作去查找组件
  cb(null, Course)
}}/>
```

`getComponents(location, callback)`

与 `component` 一样，但是是异步的，对于 code-splitting 很有用。

`callback` signature

```
cb(err, components)
<Route path="courses/:courseId" getComponent={(location, cb) => {
  // 做一些异步操作去查找组件
  cb(null, {sidebar: CourseSidebar, content: Course})
}}/>
```

`children`

Route 可以被嵌套，`this.props.children` 包含了从子 route 组件创建的元素。由于这是路由设计中非常重要的部分，请参考 Route 配置。

`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 即将退出时调用。

PlainRoute

route 定义的一个普通的 JavaScript 对象。 `Router` 把 JSX 的 `<Route>` 转化到这个对象中，如果你喜欢，你可以直接使用它们。 所有的 props 都和 `<Route>` 的 props 一样，除了那些列在这里的。

Props

`childRoutes`

子 route 的一个数组，与在 JSX route 配置中的 `children` 一样。

`getChildRoutes(location, callback)`

与 `childRoutes` 一样，但是是异步的，并且可以接收 `location`。对于 code-splitting 和动态路由匹配很有用（给定一些 state 或 session 数据会返回不同的子 route）。

`callback` signature

```
cb(err, routesArray)
let myRoute = {
  path: 'course/:courseId',
  childRoutes: [
    announcementsRoute,
    gradesRoute,
    assignmentsRoute
  ]
}

// 异步的子 route
let myRoute = {
  path: 'course/:courseId',
  getChildRoutes(location, cb) {
    // 做一些异步操作去查找子 route
    cb(null, [ announcementsRoute, gradesRoute, assignmentsRoute ])
  }
}

// 可以根据一些 state
// 跳转到依赖的子 route
<Link to="/picture/123" state={{ fromDashboard: true }}/>

let myRoute = {
  path: 'picture/:id',
  getChildRoutes(location, cb) {
    let { state } = location

    if (state && state.fromDashboard) {
      cb(null, [dashboardPictureRoute])
    } else {
      cb(null, [pictureRoute])
    }
  }
}
```

`indexRoute`

index route。这与在使用 JSX route 配置时指定一个 `<IndexRoute>` 子集一样。

`getIndexRoute(location, callback)`

与 `indexRoute` 一样，但是是异步的，并且可以接收 `location`。与 `getChildRoutes` 一样，对于 code-splitting 和动态路由匹配很有用

`callback` **signature**

```
cb(err, route)
// 例如:
let myIndexRoute = {
  component: MyIndex
}

let myRoute = {
  path: 'courses',
  indexRoute: myIndexRoute
}

// 异步的 index route
let myRoute = {
  path: 'courses',
  getIndexRoute(location, cb) {
    // 做一些异步操作
    cb(null, myIndexRoute)
  }
}
```

Redirect

在应用中 `<Redirect>` 可以设置重定向到其他 route 而不改变旧的 URL。

Props

`from`

你想由哪个路径进行重定向，包括动态段。

`to`

你想重定向的路径。

`query`

默认情况下，query 的参数只会经过，但如果你需要你可以指定它们。

```
// 我们需要从 `/profile/123` 改变到 `/about/123`
// 并且由 `/get-in-touch` 重定向到 `/contact`
<Route component={App}>
```

```
    <Route path="about/:userId" component={UserProfile}/>
    {/* /profile/123 -> /about/123 */}
    <Redirect from="profile/:userId" to="about/:userId" />
  </Route>
```

注意，在 route 层 `<Redirect>` 可以被放在任何地方，尽管正常的优先 规则仍适用。如果你喜欢将下一个重定向到它各自的 route 上，`from` 的路径会匹配到一个与 `path` 一样的正常路径。

```
<Route path="course/:courseId">
  <Route path="dashboard" />
  {/* /course/123/home -> /course/123/dashboard */}
  <Redirect from="home" to="dashboard" />
</Route>
```

IndexRoute

当用户在父 route 的 URL 时， Index Routes 允许你为父 route 提供一个默认的 "child"， 并且为使 `<IndexLink>` 能用提供了约定。

请看 Index Routes 指南。

Props

与 Route 的 props 一样，除了 `path`。

IndexRedirect

Index Redirects 允许你从一个父 route 的 URL 重定向到其他 route。 它们被用于允许子 route 作为父 route 的默认 route， 同时保持着不同的 URL。

请看 Index Routes 指南。

Props

与 Redirect 的 props 一样，除了 `from`。

Route Components

当 route 匹配到 URL 时会渲染一个 route 的组件。路由会在渲染时将以下属性注入组件中：

`history`

Router 的 history history。

对于跳转很有用的 `this.props.history.pushState(state, path, query)`

`location`

当前的 location。

`params`

URL 的动态段。

`route`

渲染组件的 route。

`routeParams`

`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}`。

`children`

匹配到子 route 的元素将被渲染。如果 route 有已命名的组件，那么此属性会是 undefined，并且可用的组件会被直接替换到 `this.props` 上。

示例

```
render((
  <Router>
    <Route path="/" component={App}>
      <Route path="groups" component={Groups} />
      <Route path="users" component={Users} />
    </Route>
  </Router>
), node)

class App extends React.Component {
  render() {
    return (
      <div>
        {/* 这可能是 <Users> 或 <Groups> */}
        {this.props.children}
      </div>
    )
  }
}
```

已命名的组件

当一个 route 有一个或多个已命名的组件时，其子元素的可用性是通过 `this.props` 命名的。因此 `this.props.children` 将会是 undefined。那么所有的 route 组件都可以参与嵌套。

示例

```
render((
  <Router>
    <Route path="/" component={App}>
      <Route path="groups" components={{main: Groups, sidebar: GroupsSidebar}} />
      <Route path="users" components={{main: Users, sidebar: UsersSidebar}}>
        <Route path="users/:userId" component={Profile} />
      </Route>
    </Route>
  </Router>
), node)

class App extends React.Component {
  render() {
    // 在父 route 中，被匹配的子 route 变成 props
    return (
      <div>
        <div className="Main">
          {/* 这可能是 <Groups> 或 <Users> */}
          {this.props.main}
        </div>
        <div className="Sidebar">
          {/* 这可能是 <GroupsSidebar> 或 <UsersSidebar> */}
          {this.props.sidebar}
        </div>
      </div>
    )
  }
}

class Users extends React.Component {
  render() {
    return (
      <div>
        {/* 如果在 "/users/123" 路径上这会是 <Profile> */}
        {/* UsersSidebar 也会获取到作为 this.props.children 的 <Profile> 。
            你可以把它放这渲染 */}
        {this.props.children}
      </div>
    )
  }
}
```

Mixins

生命周期 Mixin

在组件中添加一个钩子，当路由要从 route 组件的配置中跳转出来时被调用，并且有机会去取消这次跳转。主要用于表单的部分填写。

在常规的跳转中， `routerWillLeave` 会接收到一个单一的参数：我们正要跳转的 `location`。去取消此次跳转，返回 false。

提示用户确认，返回一个提示信息（字符串）。在 web 浏览器 beforeunload 事件发生时，`routerWillLeave` 不会接收到一个 location 的对象（假设你正在使用 `useBeforeUnload` history 的增强方法）。在此之上，我们是不可能知道要跳转的 location，因此 `routerWillLeave` 必须在用户关闭标签之前返回一个提示信息。

生命周期方法

`routerWillLeave(nextLocation)`

当路由尝试从一个 route 跳转到另一个并且渲染这个组件时被调用。

arguments

- `nextLocation` - 下一个 location

History Mixin

在组件中添加路由的 history 对象。

注意：你的 route components 不需要这个 mixin，它在 `this.props.history` 中已经是可用的了。这是为了组件更深次的渲染树中需要访问路由的 `history` 对象。

Methods

`pushState(state, pathname, query)`

跳转至一个新的 URL。

arguments

- `state` - location 的 state。
- `pathname` - 没有 query 完整的 URL。
- `query` - 通过路由字符串化的一个对象。

`replaceState(state, pathname, query)`

在不影响 history 长度的情况下（如一个重定向），用新的 URL 替换当前这个。

参数

- `state` - location 的 state。
- `pathname` - 没有 query 完整的 URL。
- `query` - 通过路由字符串化的一个对象。

`go(n)`

在 history 中使用 `n` 或 `-n` 进行前进或后退

`goBack()`

在 history 中后退。

`goForward()`

在 history 中前进。

`createPath(pathname, query)`

使用路由配置，将 query 字符串化加到路径名中。

`createHref(pathname, query)`

使用路由配置，创建一个 URL。例如，它会在 `pathname` 的前面加上 `#/` 给 hash history。

`isActive(pathname, query, indexOnly)`

根据当前路径是否激活返回 `true` 或 `false`。通过 `pathname` 匹配到 route 分支下的每个 route 将会是 true（子 route 是激活的情况下，父 route 也是激活的），除非 `indexOnly` 已经指定了，在这种情况下，它只会匹配到具体的路径。

参数

- `pathname` - 没有 query 完整的 URL。
- `query` - 如果没有指定，那会是一个包含键值对的对象，并且在当前的 query 中是激活状态的 - 在当前的 query 中明确是 `undefined` 的值会丢失相应的键或 `undefined`
- `indexOnly` - 一个 boolean（默认：`false`）。

示例

```
import { History } from 'react-router'

React.createClass({
  mixins: [ History ],
  render() {
    return (
      <div>
        <div onClick={() => this.history.pushState(null, '/foo')}>Go to foo</div>
        <div onClick={() => this.history.replaceState(null, 'bar')}>Go to bar
without creating a new history entry</div>
        <div onClick={() => this.history.goBack()}>Go back</div>
      </div>
    )
  }
})
```

假设你正在使用 bootstrap，并且在 Tab 中想让那些 `li` 获得 `active`：

```
import { Link, History } from 'react-router'

const Tab = React.createClass({
  mixins: [ History ],
  render() {
    let isActive = this.history.isActive(this.props.to, this.props.query)
    let className = isActive ? 'active' : ''
    return <li className={className}><Link {...this.props}/></li>
  }
})

// 如 <Link/> 一样使用它，你会得到一个包进 `li` 的锚点
// 并且还有一个自动的 `active` class。
<Tab href="foo">Foo</Tab>
```

但我仍然在使用 Classes

> 注意难道我们没有告诉你要使用 ES6 的 Classes? 😃

https://twitter.com/soprano/status/610867662797807617

在应用中少数组件由于 `History` mixin 的缘故而用得不爽，此时有以下几个选项：

- 让 `this.props.history` 通过 route 组件到达需要它的组件中。
- 使用 context

```
import { PropTypes } from 'react-router'

class MyComponent extends React.Component {
  doStuff() {
    this.context.history.pushState(null, '/some/path')
  }
}

MyComponent.contextTypes = { history: PropTypes.history }
```

- 确保你的 history 是一个 module
- 创建一个高阶的组件，我们可能用它来结束跳转和阻止 history，只是没有时间去思考所有的方法。

```
function connectHistory(Component) {
  return React.createClass({
    mixins: [ History ],
    render() {
      return <Component {...this.props} history={this.history} />
    }
  })
}
```

```
// 其他文件
import connectHistory from './connectHistory'

class MyComponent extends React.Component {
  doStuff() {
    this.props.history.pushState(null, '/some/where')
  }
}

export default connectHistory(MyComponent)
```

RouteContext Mixin

RouteContext mixin 提供了一个将 route 组件设置到 context 中的便捷方法。这对于 route 渲染元素并且希望用生命周期 mixin 来阻止跳转是很有必要的。

简单地将 `this.context.route` 添加到组件中。

Utilities

`useRoutes(createHistory)`

返回一个新的 `createHistory` 函数，它可以用来创建读取 route 的 history 对象。

- listen((error, nextState) => {})
- listenBeforeLeavingRoute(route, (nextLocation) => {})
- match(location, (error, redirectLocation, nextState) => {})
- isActive(pathname, query, indexOnly=false)

`match(location, cb)`

这个函数被用于服务端渲染。它在渲染之前会匹配一组 route 到一个 location，并且在完成时调用 `callback(error, redirectLocation, renderProps)`。

传给回调函数去 `match` 的三个参数如下：

- `error`：如果报错时会出现一个 Javascript 的 `Error` 对象，否则是 `undefined`。
- `redirectLocation`：如果 route 重定向时会有一个 Location 对象，否则是 `undefined`。
- `renderProps`：当匹配到 route 时 props 应该通过路由的 context，否则是 `undefined`。

如果这三个参数都是 `undefined`，这就意味着在给定的 location 中没有 route 被匹配到。

注意：你可能不想在浏览器中用它，除非你做的是异步 route 的服务端渲染。

`createRoutes(routes)`

创建并返回一个从给定对象 route 的数组，它可能是 JSX 的 route，一个普通对象的 route，或是其他的数组。

params

`routes`

一个或多个的 `Route` 或 `PlainRoute`。

PropTypes

敬请期待!