地址：http://blog.csdn.net/sinat\_17775997/article/details/77411324

https://juejin.im/post/5995a2506fb9a0249975a1a4?utm\_source=tuicool&utm\_medium=referral

我在 React Rally 2016 大会上第一次遇到了 [Michael Jackson](https://link.juejin.im/?target=https%3A%2F%2Ftwitter.com%2Fmjackson)，不久之后便写了一篇 [an article on React Router 3](https://link.juejin.im/?target=https%3A%2F%2Fcss-tricks.com%2Flearning-react-router%2F)。Michael 与 [Ryan Florence](https://link.juejin.im/?target=https%3A%2F%2Ftwitter.com%2Fryanflorence) 都是 React Router 的主要作者。遇到一位我非常喜欢的工具的创建者是激动人心的，但当他这么说的时候，我感到很震惊。“让我向你们展示我们在 React Router 4 的想法，它的**方式**是截然不同的！”。老实说，我真的不明白新的方向以及为什么它需要如此大的改变。由于路由是应用程序架构的重要组成部分，因此这可能会改变一些我喜欢的模式。这些改变的想法让我很焦虑。考虑到社区凝聚力以及 React Router 在这么多的 React 应用程序中扮演着重要的角色，我不知道社区将如何接受这些改变。

几个月后，[React Router 4](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2F) 发布了，仅仅从 Twitter 的嗡嗡声中我便得知，大家对于这个重大的重写存在着不同的想法。这让我想起了第一个版本的 React Router 针对其渐进概念的推回。在某些方面，早期版本的 React Router 符合我们传统的思维模式，即一个应用的路由“应该”将所有的路由规则放在一个地方。然而，并不是每个人都接受使用嵌套的 JSX 路由。但就像 JSX 自身说服了批评者一样（至少是大多数），许多人转而相信嵌套的 JSX 路由是很酷的想法。

如是，我学习了 React Router 4。无可否认，第一天是挣扎的。挣扎的倒不是其 API，而更多的是使用它的模式和策略。我使用 React Router 3 的思维模式并没有很好地迁移到 v4。如果要成功，我将不得不改变我对路由和布局组件之间的关系的看法。最终，出现了对我有意义的新模式，我对路由的新方向感到非常高兴。React Router 4 不仅包含 v3 的所有功能，而且还有新的功能。此外，起初我对 v4 的使用过于复杂。一旦我获得了一个新的思维模式，我就意识到这个新的方向是惊人的！

本文的意图并不是重复 React Router 4 [已经写得很好的文档](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2F)。我将介绍最常见的 API，但真正的重点是我发现的成功模式和策略。

对于本文，以下是一些你需要熟悉的 JavaScript 概念:

* React [（无状态）函数组件](https://link.juejin.im/?target=https%3A%2F%2Ffacebook.github.io%2Freact%2Fdocs%2Fcomponents-and-props.html)
* ES2015 [箭头函数](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FJavaScript%2FReference%2FFunctions%2FArrow_functions) 以及它们的“隐式返回”
* ES2015 [解构](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FJavaScript%2FReference%2FOperators%2FDestructuring_assignment)
* ES2015 [模板字符串](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FJavaScript%2FReference%2FTemplate_literals)

如果你喜欢跳转到演示区的话，请点这里：

[查看演示](https://link.juejin.im/?target=https%3A%2F%2Fcodepen.io%2Fbradwestfall%2Fproject%2Feditor%2FXWNWge%2F%3Fpreview_height%3D50%26amp%3Bopen_file%3Dsrc%2Fapp.js)

**新的 API 和新的思维模式**

React Router 的早期版本将路由规则集中在一个位置，使它们与布局组件分离。当然，路由可以被划分成多个文件，但从概念上讲，路由是一个单元，基本上是一个美化的配置文件。

或许了解 v4 不同之处的最好方法是用每个版本编写一个简单的两页应用程序并进行比较。示例应用程序只有两个路由，对应首页和用户页面。

这里是 v3 的：

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

const PrimaryLayout = props => (

<div className="primary-layout">

<header>

Our React Router 3 App

</header>

<main>

{props.children}

</main>

</div>

)

const HomePage =() => <div>Home Page</div>

const UsersPage = () => <div>Users Page</div>

const App = () => (

<Router history={browserHistory}>

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

<IndexRoute component={HomePage} />

<Route path="/users" component={UsersPage} />

</Route>

</Router>

)

render(<App />, document.getElementById('root'))

以下是 v3 中的一些核心思想，但在 v4 中是不正确的:

* 路由集中在一个地方。
* 布局和页面嵌套是通过 <Route> 组件的嵌套而来的。
* 布局和页面组件是完全纯粹的，它们是路由的一部分。

React Router 4 不再主张集中式路由了。相反，路由规则位于布局和 UI 本身之间。例如，以下是 v4 中的相同的应用程序：

**import** { BrowserRouter, Route } **from** 'react-router-dom'

**const** PrimaryLayout = () => (

<div className="primary-layout">

<header>

Our React Router 4 App

</header>

<main>

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

<Route path="/users" component={UsersPage} />

</main>

</div>

)

const HomePage =() => <div>Home Page</div>

const UsersPage = () => <div>Users Page</div>

const App = () => (

<BrowserRouter>

<PrimaryLayout />

</BrowserRouter>

)

render(<App />, document.getElementById('root'))

**新的 API 概念**：由于我们的应用程序是用于浏览器的，所以我们需要将它封装在来自 v4 的 BrowserRouter 中。还要注意的是我们现在从 react-router-dom 中导入它（这意味着我们安装的是 react-router-dom 而不是 react-router）。提示！现在叫做 react-router-dom 是因为还有一个 [native 版本](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fnative)。

对于使用 React Router v4 构建的应用程序，首先看到的是“路由”似乎丢失了。在 v3 中，路由是我们的应用程序直接呈现给 DOM 的最巨大的东西。 现在，除了 <BrowserRouter>外，我们首先抛给 DOM 的是我们的应用程序本身。

另一个在 v3 的例子中有而在 v4 中没有的是，使用 {props.children} 来嵌套组件。这是因为在 v4 中，<Route> 组件在何处编写，如果路由匹配，子组件将在那里渲染。

**包容性路由**

在前面的例子中，你可能已经注意到了 exact 这个属性。那么它是什么呢？V3 的路由规则是“排他性”的，这意味着只有一条路由将获胜。V4 的路由默认为“包含”的，这意味着多个 <Route> 可以同时进行匹配和渲染。

在上一个例子中，我们试图根据路径渲染 HomePage 或者 UsersPage。如果从示例中删除了 exact 属性，那么在浏览器中访问 /users 时，HomePage 和 UsersPage 组件将同时被渲染。

要更好地了解匹配逻辑，请查看 [path-to-regexp](https://link.juejin.im/?target=https%3A%2F%2Fwww.npmjs.com%2Fpackage%2Fpath-to-regexp)，这是 v4 现在正在使用的，以确定路由是否匹配 URL。

为了演示包容性路由是有帮助的，我们在标题中包含一个 UserMenu，但前提是我们在应用程序的用户部分：

**const** PrimaryLayout = () => (

<div className="primary-layout">

<header>

Our React Router 4 App

<Route path="/users" component={UsersMenu} />

</header>

<main>

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

<Route path="/users" component={UsersPage} />

</main>

</div>

)

现在，当用户访问 /users 时，两个组件都会渲染。类似这样的事情在 v3 中通过特定的匹配模式也是可行的，但它更复杂。得益于 v4 的包容性路由，现在能够很轻松地实现。

**排他性路由**

如果你只需要在路由列表里匹配一个路由，则使用 <Switch> 来启用排他路由：

**const** PrimaryLayout = () => (

<div className="primary-layout">

<PrimaryHeader />

<main>

<Switch>

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

<Route path="/users/add" component={UserAddPage} />

<Route path="/users" component={UsersPage} />

<Redirect to="/" />

</Switch>

</main>

</div>

)

在给定的 <Switch> 路由中只有一条将渲染。在 HomePage 路由上，我们仍然需要 exact属性，尽管我们会先把它列出来。否则，当访问诸如 /users 或 /users/add 的路径时，主页路由也将匹配。事实上，战略布局是使用排他路由策略（因为它总是像传统路由那样使用）时的关键。请注意，我们在 /users 之前策略性地放置了 /users/add 的路由，以确保正确匹配。由于路径 /users/add 将匹配 /users 和 /users/add，所以最好先把 /users/add 放在前面。

当然，如果我们以某种方式使用 exact，我们可以把它们放在任何顺序上，但至少我们有选择。

如果遇到，<Redirect> 组件将会始终执行浏览器重定向，但是当它位于 <Switch> 语句中时，只有在其他路由不匹配的情况下，才会渲染重定向组件。想了解在非切换环境下如何使用 <Redirect>，请参阅下面的**授权路由**。

**“默认路由”和“未找到”**

尽管在 v4 中已经没有 <IndexRoute> 了，但可以使用 <Route exact> 来达到同样的效果。如果没有路由解析，则可以使用 <Switch> 与 <Redirect> 重定向到具有有效路径的默认页面（如同我对本示例中的 HomePage 所做的），甚至可以是一个“未找到页面”。

**嵌套布局**

你可能开始期待嵌套子布局，以及如何实现它们。我原本不认为我会纠结这个概念，但我确实纠结了。React Router v4 给了我们很多选择，这使它变得很强大。但是，选择意味着有选择不理想策略的自由。表面上看，嵌套布局很简单，但根据你的选择，可能会因为你组织路由的方式而遇到阻碍。

为了演示，假设我们想扩展我们的用户部分，所以我们会有一个“用户列表”页面和一个“用户详情”页面。我们也希望产品也有类似的页面。用户和产品都需要其个性化的子布局。例如，每个可能都有不同的导航选项卡。有几种方法可以解决这个问题，有的好，有的不好。第一种方法不是很好，但我想告诉你，这样你就不会掉入这个陷阱。第二种方法要好很多。

第一种方法，我们修改 PrimaryLayout，以适应用户和产品对应的列表及详情页面：

**const** PrimaryLayout = props => {

**return** (

<div className="primary-layout">

<PrimaryHeader />

<main>

<Switch>

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

<Route path="/users" exact component={BrowseUsersPage} />

<Route path="/users/:userId" component={UserProfilePage} />

<Route path="/products" exact component={BrowseProductsPage} />

<Route path="/products/:productId" component={ProductProfilePage} />

<Redirect to="/" />

</Switch>

</main>

</div>

)

}

虽然这在技术上可行的，但仔细观察这两个用户页面就会发现问题：

**const** BrowseUsersPage = () => (

<div className="user-sub-layout">

<aside>

<UserNav />

</aside>

<div className="primary-content">

<BrowseUserTable />

</div>

</div>

)

**const** UserProfilePage = props => (

<div className="user-sub-layout">

<aside>

<UserNav />

</aside>

<div className="primary-content">

<UserProfile userId={props.match.params.userId} />

</div>

</div>

)

**新 API 概念**：props.match 被赋到由 <Route> 渲染的任何组件。你可以看到，userId 是由 props.match.params 提供的，了解更多请参阅 [v4 文档](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fweb%2Fexample%2Furl-params)。或者，如果任何组件需要访问 props.match，而这个组件没有由 <Route> 直接渲染，那么我们可以使用 [withRouter()](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fweb%2Fapi%2FwithRouter) 高阶组件。

每个用户页面不仅要渲染其各自的内容，而且还必须关注子布局本身（并且每个子布局都是重复的）。虽然这个例子很小，可能看起来微不足道，但重复的代码在一个真正的应用程序中可能是一个问题。更不用说，每次 BrowseUsersPage 或 UserProfilePage 被渲染时，它将创建一个新的 UserNav 实例，这意味着所有的生命周期方法都将重新开始。如果导航标签需要初始网络流量，这将导致不必要的请求 —— 这都是我们决定使用路由的方式造成的。

这里有另一种更好的方法：

**const** PrimaryLayout = props => {

**return** (

<div className="primary-layout">

<PrimaryHeader />

<main>

<Switch>

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

<Route path="/users" component={UserSubLayout} />

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

<Redirect to="/" />

</Switch>

</main>

</div>

)

}

与每个用户和产品页面相对应的四条路由不同，我们为每个部分的布局提供了两条路由。

请注意，上述示例没有使用 exact 属性，因为我们希望 /users 匹配任何以 /users 开头的路由，同样适用于产品。

通过这种策略，渲染其它路由将成为子布局的任务。UserSubLayout 可能如下所示：

**const** UserSubLayout = () => (

<div className="user-sub-layout">

<aside>

<UserNav />

</aside>

<div className="primary-content">

<Switch>

<Route path="/users" exact component={BrowseUsersPage} />

<Route path="/users/:userId" component={UserProfilePage} />

</Switch>

</div>

</div>

)

新策略中最明显的胜出在于所有用户页面之间的不重复布局。这是一个双赢，因为它不会像第一个示例那样具有相同生命周期的问题。

有一点需要注意的是，即使我们在布局结构中深入嵌套，路由仍然需要识别它们的完整路径才能匹配。为了节省重复输入（以防你决定将“用户”改为其他内容），请改用 props.match.path：

**const** UserSubLayout = props => (

<div className="user-sub-layout">

<aside>

<UserNav />

</aside>

<div className="primary-content">

<Switch>

<Route path={props.match.path} exact component={BrowseUsersPage} />

<Route path={`${props.match.path}/:userId`} component={UserProfilePage} />

</Switch>

</div>

</div>

)

**匹配**

到目前为止，props.match 对于知道详情页面渲染的 userId 以及如何编写我们的路由是很有用的。match 对象给我们提供了几个属性，包括 match.params、match.path、match.url 和[其他几个](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fweb%2Fapi%2Fmatch)。

**match.path vs match.url**

起初这两者之间的区别似乎并不清楚。控制台日志有时会显示相同的输出，这使得它们之间的差异更加模糊。例如，当浏览器路径为 /users 时，它们在控制台日志将输出相同的值：

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

console.log(match.url) // 输出："/users"

console.log(match.path) // 输出："/users"

**return** (

<div className="user-sub-layout">

<aside>

<UserNav />

</aside>

<div className="primary-content">

<Switch>

<Route path={match.path} exact component={BrowseUsersPage} />

<Route path={`${match.path}/:userId`} component={UserProfilePage} />

</Switch>

</div>

</div>

)

}

**ES2015 概念：** match 在组件函数的参数级别将被[解构](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FJavaScript%2FReference%2FOperators%2FDestructuring_assignment)。

虽然我们看不到差异，但 match.url 是浏览器 URL 中的实际路径，而 match.path 是为路由编写的路径。这就是为什么它们是一样的，至少到目前为止。但是，如果我们更进一步，在 UserProfilePage 中进行同样的控制台日志操作，并在浏览器中访问 /users/5，那么 match.url 将是 "/users/5" 而 match.path 将是 "/users/:userId"。

**选择哪一个？**

如果你要使用其中一个来帮助你构建路由路径，我建议你选择 match.path。使用 match.url 来构建路由路径最终会导致你不想看到的场景。下面是我遇到的一个情景。在一个像 UserProfilePage（当用户访问 /users/5 时渲染）的组件中，我渲染了如下这些子组件：

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

<div>UserId: {match.params.userId}</div>

)

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

<div>UserId: {match.params.userId}</div>

)

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

<div>

User Profile:

<Route path={`${match.url}/comments`} component={UserComments} />

<Route path={`${match.path}/settings`} component={UserSettings} />

</div>

)

为了说明问题，我渲染了两个子组件，一个路由路径来自于 match.url，另一个来自 match.path。以下是在浏览器中访问这些页面时所发生的事情:

* 访问 /users/5/comments 渲染 "UserId: undefined"。
* 访问 /users/5/settings 渲染 "UserId: 5"。

那么为什么 match.path 可以帮助我们构建路径 而 match.url 则不可以呢？答案就是这样一个事实：{${match.url}/comments} 基本上就像和硬编码的 {'/users/5/comments'} 一样。这样做意味着后续组件将无法正确地填充 match.params，因为路径中没有参数，只有硬编码的 5。

直到后来我看到[文档的这一部分](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fcore%2Fapi%2Fmatch)，才意识到它有多重要：

match:

* path - (string) 用于匹配路径模式。**用于构建嵌套的 <Route>**
* url - (string) URL 匹配的部分。 **用于构建嵌套的 <Link>**

**避免匹配冲突**

假设我们制作的应用程序是一个仪表版，所以我们希望能够通过访问 /users/add 和 /users/5/edit 来新增和编辑用户。但是在前面的例子中，users/:userId 已经指向了 UserProfilePage。那么这是否意味着带有users/:userId 的路由现在需要指向另一个子子布局来容纳编辑页面和详情页面？我不这么认为，因为编辑和详情页面共享相同的用户子布局，所以这个策略是可行的：

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

<div className="user-sub-layout">

<aside>

<UserNav />

</aside>

<div className="primary-content">

<Switch>

<Route exact path={props.match.path} component={BrowseUsersPage} />

<Route path={`${match.path}/add`} component={AddUserPage} />

<Route path={`${match.path}/:userId/edit`} component={EditUserPage} />

<Route path={`${match.path}/:userId`} component={UserProfilePage} />

</Switch>

</div>

</div>

)

请注意，为了确保进行适当的匹配，新增和编辑路由需要战略性地放在详情路由之前。如果详情路径在前面，那么访问 /users/add 时将匹配详情（因为 "add" 将匹配 :userId）。

或者，如果我们这样创建路径 ${match.path}/:userId(\\d+)，来确保 :userId 必须是一个数字，那么我们可以先放置详情路由。然后访问 /users/add 将不会产生冲突。这是我在 [path-to-regexp](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Fpillarjs%2Fpath-to-regexp%23custom-match-parameters) 的文档中学到的技巧。

**授权路由**

在应用程序中，通常会根据用户的登录状态来限制用户访问某些路由。对于未经授权的页面（如“登录”和“忘记密码”）与已授权的页面（应用程序的主要部分）看起来不一样也是常见的。为了解决这些需求，需要考虑一个应用程序的主要入口点：

**class** **App** **extends** **React**.**Component** {

render() {

**return** (

<Provider store={store}>

<BrowserRouter>

<Switch>

<Route path="/auth" component={UnauthorizedLayout} />

<AuthorizedRoute path="/app" component={PrimaryLayout} />

</Switch>

</BrowserRouter>

</Provider>

)

}

}

使用 [react-redux](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2Freactjs%2Freact-redux) 与 React Router v4 非常类似，就像之前一样，只需将 BrowserRouter包在 <Provider> 中即可。

通过这种方法可以得到一些启发。第一个是根据我们所在的应用程序的哪个部分，在两个顶层布局之间进行选择。像访问 /auth/login 或 /auth/forgot-password 这样的路径会使用 UnauthorizedLayout —— 一个看起来适于这种情况的布局。当用户登录时，我们将确保所有路径都有一个 /app 前缀，它使用 AuthorizedRoute 来确定用户是否登录。如果用户在没有登录的情况下，尝试访问以 /app 开头的页面，那么将被重定向到登录页面。

虽然 AuthorizedRoute 不是 v4 的一部分，但是我[在 v4 文档的帮助下](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fweb%2Fexample%2Fauth-workflow)自己写了。v4 中一个惊人的新功能是能够为特定的目的创建你自己的路由。它不是将 component 的属性传递给 <Route>，而是传递一个 render 回调函数：

**class** **AuthorizedRoute** **extends** **React**.**Component** {

componentWillMount() {

getLoggedUser()

}

render() {

**const** { component: Component, pending, logged, ...rest } = **this**.props

**return** (

<Route {...rest} render={props => {

if (pending) return <div>Loading...</div>

return logged

? <Component {...this.props} />

: <Redirect to="/auth/login" />

}} />

)

}

}

const stateToProps = ({ loggedUserState }) => ({

pending: loggedUserState.pending,

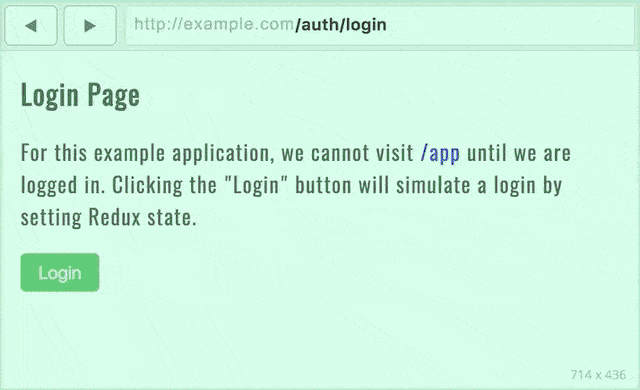
logged: loggedUserState.logged

})

export default connect(stateToProps)(AuthorizedRoute)

可能你的登录策略与我的不同，我是使用网络请求来 getLoggedUser()，并将 pending 和 logged 插入 Redux 的状态中。pending 仅表示在路由中请求仍在继续。

点击此处查看 CodePen 上完整的[身份验证示例](https://link.juejin.im/?target=https%3A%2F%2Fcodepen.io%2Fbradwestfall%2Fproject%2Feditor%2FXWNWge%2F%3Fpreview_height%3D50%26amp%3Bopen_file%3Dsrc%2Fapp.js)。

[](https://link.juejin.im/?target=https://codepen.io/bradwestfall/project/editor/XWNWge/?preview_height=50&amp;open_file=src/app.js)

**其他提示**

React Router v4 还有很多其他很酷的方面。最后，一定要提几件小事，以免到时它们让你措手不及。

**<Link> vs <NavLink>**

在 v4 中，有两种方法可以将锚标签与路由集成：[<Link>](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fweb%2Fapi%2FLink) 和 [<NavLink>](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fweb%2Fapi%2FNavLink)

<NavLink> 与 <Link> 一样，但如果 <NavLink> 匹配浏览器的 URL，那么它可以提供一些额外的样式能力。例如，在[示例应用程序](https://link.juejin.im/?target=https%3A%2F%2Fcodepen.io%2Fbradwestfall%2Fproject%2Feditor%2FXWNWge%2F%23)中，有一个<PrimaryHeader> 组件看起来像这样：

**const** PrimaryHeader = () => (

<header className="primary-header">

<h1>Welcome to our app!</h1>

<nav>

<NavLink to="/app" exact activeClassName="active">Home</NavLink>

<NavLink to="/app/users" activeClassName="active">Users</NavLink>

<NavLink to="/app/products" activeClassName="active">Products</NavLink>

</nav>

</header>

)

使用 <NavLink> 可以让我给任何一个激活的链接设置一个 active 样式。而且，需要注意的是，我也可以给它们添加 exact 属性。如果没有 exact，由于 v4 的包容性匹配策略，那么在访问 /app/users 时，主页的链接将处于激活中。就个人经历而言，NavLink 带 exact属性等价于 v3 的 <link>，而且更稳定。

**URL 查询字符串**

再也无法从 React Router v4 中获取 URL 的查询字符串了。在我看来，[做这个决定](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2FReactTraining%2Freact-router%2Fissues%2F4410)是因为没有关于如何处理复杂查询字符串的标准。所以，他们决定让开发者去选择如何处理查询字符串，而不是将其作为一个选项嵌入到 v4 的模块中。这是一件好事。

就个人而言，我使用的是 [query-string](https://link.juejin.im/?target=https%3A%2F%2Fwww.npmjs.com%2Fpackage%2Fquery-string)，它是由 [sindresorhus](https://link.juejin.im/?target=https%3A%2F%2Ftwitter.com%2Fsindresorhus) 大神写的。

**动态路由**

关于 v4 最好的部分之一是几乎所有的东西（包括 <Route>）只是一个 React 组件。路由不再是神奇的东西了。我们可以随时随地渲染它们。想象一下，当满足某些条件时，你的应用程序的整个部分都可以路由到。当这些条件不满足时，我们可以移除路由。甚至我们可以做一些疯狂而且很酷的[递归路由](https://link.juejin.im/?target=https%3A%2F%2Freacttraining.com%2Freact-router%2Fweb%2Fexample%2Frecursive-paths)。

因为它 [Just Components™](https://link.juejin.im/?target=https%3A%2F%2Fyoutu.be%2FMf0Fy8iHp8k%3Ft%3D3m22s)，React Router 4 更简单了。

/\*match 会通过props传到组件中，通过const { history, location, match, match: { params: { environment }}} = this.props可获取到match\*/