りあクト!

第3.1版

TypeScriptで始めるつらくないReact開発

【III. React応用編】

大岡由佳

最新のReact 17、Suspenseにも対応

Reactの難関、副作用処理を徹底的にわかりやすく!

シリーズ累計 **1.2万部** 突破!

読者からの声

「この本を選んだ自分を褒めてあげたい」

「実践的なReactを学びたい人にオススメ」

前版 BOOTH売上 1位

フロントエンド未経験から1週間で戦力になるための秋谷へのReact研修もいよいよ佳境。敏腕リーダー柴崎の指導によりHooksまでなんとかたどりついた秋谷だったが、ついにReact入門者にとって最大の障害であるReduxがその前に立ちふさがる。しかもその先にはまだ新技術が控えているという。まさに今、Reactの副作用処理をめぐる手に汗握るサスペンスが始まろうとしている。

本作の構成（全三部作）

りあクト!
TypeScript で始めるつらくない React 開発
第 3.1 版
【Ⅲ. React 応用編】

大岡由佳

くるみ割り書房

第三部まえがき

本作は TypeScript で React アプリケーションを開発するための技術解説書です。「Ⅰ. 言語・環境編」「Ⅱ. React 基礎編」「Ⅲ. React 応用編」からなる三部作となっており、通して読むことで React によるモダンフロントエンド開発に必要な知識がひととおり身につくようになっています。

本書は三部作の最後を飾る第三部「Ⅲ. React 応用編」です。第一部では、モダンフロントエンド開発に必要な JavaScript および TypeScript の知識を、第二部で React の基礎知識と基本的な使い方を学びました。

第三部では React による副作用処理をテーマとして扱います。なお今版は、第 2 版までとその対応する部分の内容が大幅に変わっています。これは Hooks が React の新しいスタンダードとなったことにより、Redux も含めて React における副作用処理のあり方が大きく変革を求められたためです。よって初版と第 2 版で大きく扱っていた redux-saga にとうとう引導を渡すこととなりました。さらに開発者の間で近年ささやかれだした「Redux 不要論」についても、かなりのページを割いて検証しています。

そして筆者として本作を通じていちばんの目玉だと思っているのは、「第 13 章　Suspense でデータ取得の宣言的 UI を実現する」です。Suspense for data fetching は公式にはまだ裏技扱いであり、Concurrent モードに至っては実験的ビルドでしかリリースされていません。しかし、これらは当初から React が全力で追求してきた「宣言的 UI」を実現するための最後のパズルのピースとなる、集大成の技術だというのが筆者の見解です。ゆえにかなり力を入れて書きました。もし他の章は飛ばしたとしても、13 章だけは何としても読んでいただきたい内容になっています。

「Hooks で大変革があったばかりなのに、もう次の新しい技術が出るのかよ……」という声が聞こえてきそうです。でもその誕生のときから仮想 DOM、単方向データフロー、関数コンポーネント、Hooks と他のフレームワークが追随せざるをえない革新的な新技術を世に送り出し続けてきた React の業（ごう）といいましょうか。公開から 7 年を過ぎてもその圧倒的な地位を他に譲る気配すらないのは、この React の持つダイナミズムによるところが大きいです。React を使いこなすにはその変化へ対応できる開発者になる必要があり、その点において本作は他のどのリソースよりも役に立つも

のであることを筆者として信じています。

サンプルコードについて

本文で紹介しているサンプルコードは、GitHub に用意した下記の専用リポジトリにて、章・節ごとに分けてすべて公開してあります。

https://github.com/oukayuka/Riakuto-StartingReact-ja3.1

CodeSandbox を使用しているものもありますが、本文中に随時 URL を掲載していますのでそちらをブラウザでご覧ください。学習を効率的に行うためにも、本文を読み進めながらこれらのコードを実際に実行することを強くおすすめします。またコードをご自身で変更して挙動のちがいを確かめてみるのも理解の助けになるはずです。

なお、リポジトリ内のコードと本文に記載されているコードの内容にはしばしば差分があります。ESLint のコメントアウト文などは本筋と関係ないため省略することもあり、それをそのままご自身の環境で書き写して実行するとエラーになる場合がありますので、ご注意ください。

本文および上記リポジトリに掲載しているコードは、読者のプログラムやドキュメントに使用してかまいません。コードの大部分を転載する場合を除き、筆者に許可を求める必要はありません。出典を明記する必要はありませんが、そうしていただければありがたいです。

本書について

登場人物

柴崎雪菜（しばさき・ゆきな）

とある都内のインターネットサービスを運営する会社のフロントエンドエンジニアでテックリード。React 歴は 3 年ほど。本格的なフロントエンド開発チームを作るための中核的人材として採用され、今の会社に転職してきた。チームメンバーを集めるため採用にも関わり自ら面接も行っていたが、彼女の要求基準の高さもあってなかなか採用に至らない状態が続く。そこで「自分が React を教えるから他チームのエンジニアを回してほしい」と上層部に要望を出し、社内公募が実行された。

秋谷香苗（あきや・かなえ）

柴崎と同じ会社に勤務する、新卒 2 年目のやる気あふれるエンジニア。入社以来もっぱら Ruby on Rails によるサーバサイドの開発に携わっていたが、柴崎のメンバー募集に志願してフロントエンド開発チームに参加した。そこで柴崎から「1 週間で戦力になって」といわれ、彼女にマンツーマンで教えを受けることになる。

前版との差分および正誤表

過去の版からの差分と、本文内の記述内容の誤りや誤植についての正誤表は以下のページに掲載します。なお、電子書籍版では訂正したものを新バージョンとして随時配信する予定です。

- 各版における内容の変更
  https://github.com/oukayuka/Riakuto-StartingReact-ja3.1/blob/master/CHANGELOG.md

- 『りあクト！ TypeScript で始めるつらくない React 開発 第 3.1 版』正誤表
  https://github.com/oukayuka/Riakuto-StartingReact-ja3.1/blob/master/errata.md

本文中で使用している主なソフトウェアのバージョン

- React（`react`） …… 17.0.1
- React DOM（`react-dom`） …… 17.0.1
- React（`react@experimental`） …… 0.0.0-experimental-3310209d0
- React DOM（`react-dom@experimental`） …… 0.0.0-experimental-3310209d0
- Create React App（`create-react-app`） …… 4.0.1
- TypeScript（`typescript`） …… 4.1.3
- ESLint（`eslint`） …… 7.14.0
- Prettier（`prettier`） …… 2.2.1
- React Router（`react-router`） …… 5.2.0
- React Router（`react-router@next`） …… 6.0.0-beta.0
- Redux（`redux`） …… 4.0.5
- React Redux（`react-redux`） …… 7.2.1
- Redux Toolkit（`@reduxjs/toolkit`） …… 1.5.0
- Ky（`ky`） …… 0.25.1
- React Query（`react-query`） …… 3.5.5
- SWR（`swr`） …… 0.3.11

目次

第一部「Ⅰ. 言語・環境編」目次

第 1 章　こんにちは React

第 2 章　エッジでディープな JavaScript の世界

第 3 章　関数型プログラミングでいこう

第 4 章　TypeScript で型をご安全に

第二部「II. React 基礎編」目次

第 5 章　JSX で UI を表現する

第 6 章　Linter とフォーマッタでコード美人に

第 7 章　React をめぐるフロントエンドの歴史

第 8 章　何はなくともコンポーネント

第 9 章　Hooks、関数コンポーネントの合体強化パーツ

第 10 章　React におけるルーティング

10-1.　SPA におけるルーティングとは

「まず最初に秋谷さん、Web アプリケーションにおける『ルーティング』の定義を述べてください」

「……いきなりですね。えーっと、『リクエストされた URL に対して、それに紐づく適切なページ内容をアプリケーションサーバがクライアントに返すこと』でいいでしょうか？」

「うん、Ruby on Rails で作られたようなサーバサイド Web アプリケーションであれば、それで正解だね。そのしくみを図で示せば次のようになるかな」

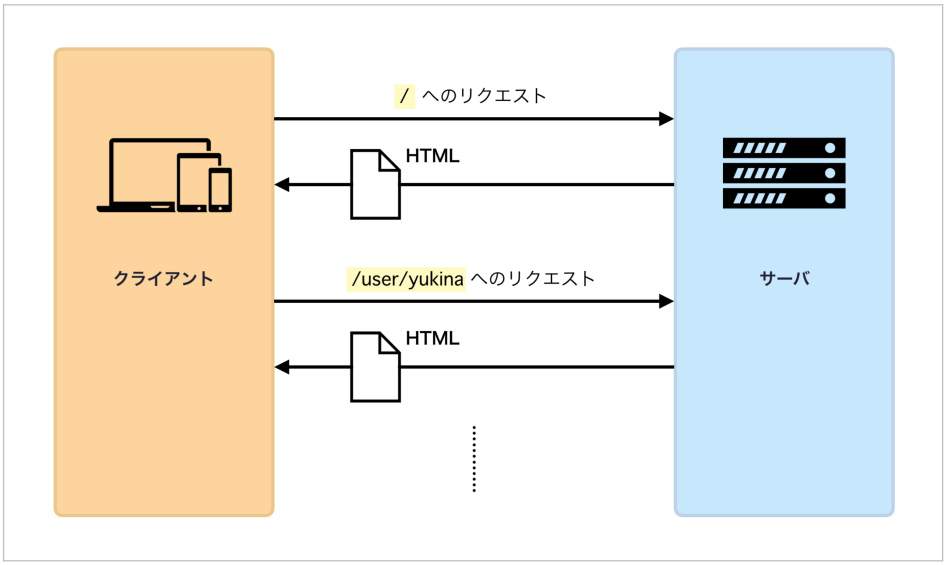

図 1: サーバサイド Web アプリケーションのルーティング

「ただし SPA（Single Page Application）では、その定義内容は適切とはいえない。典型的な SPA では、サーバへの初回のリクエストに対してその URL にかかわらず、アプリケーション全体が記述された JavaScript のコードのかたまりとアセットファイルが返される。それ以降のアプリケーション内部でのページ遷移は、アプリケーションが動的に DOM 要素を書き換えることで移動してるよ

うに見えるだけなのね。そしてそれによってブラウザのアドレスバーの URL が書き換わることがあっても、その処理もブラウザの中で完結していて実際にサーバへリクエストが飛ぶことは原則的にないの[1]」

「うーむ、そうなんですね。そのへんの理解はあいまいでした」

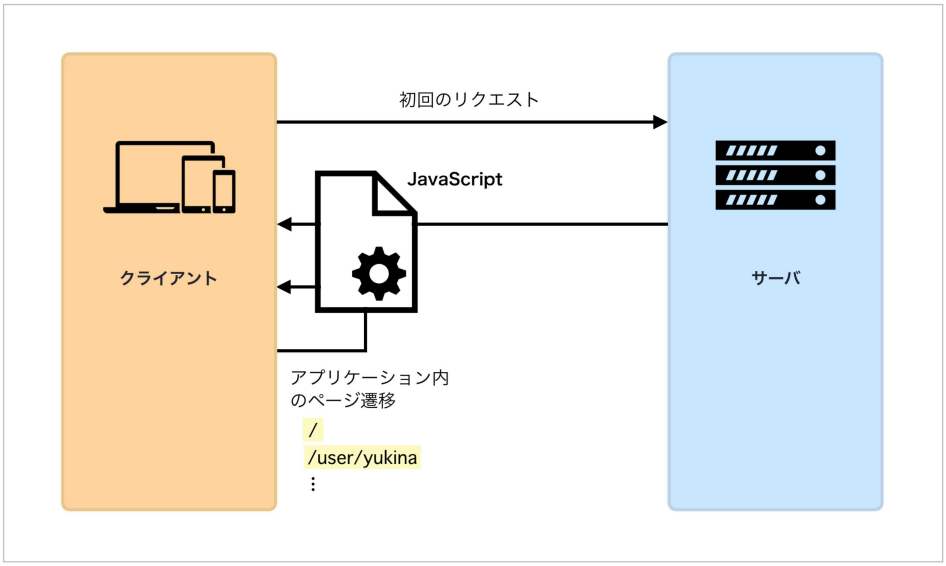

図 2: SPA のルーティング

「ちょっと PC の Chrome で Instagram のトップページにアクセスしてから、デベロッパーツールで『Network』のタブを開いたまま右上のコンパスアイコンをクリックして `instagram.com` から `instagram.com/explore` に遷移してみて。そうすると本来ならリクエストされた要素のリストに `explore` が表示されるはずなのに探しても見当たらないでしょ？」

「ほんとだ。どこにもないですね」

「これが SPA のルーティングの挙動なの。URL の書き換えをともなうページ遷移であっても、サーバにリクエストが行くことなく基本的にクライアントだけで完結する。ただ、歴史的にここに至るまでにはワンクッションあったんだよね。秋谷さんは昔の Gmail や Twitter とかで `#!` みたいな記号が入った URL を見たことはない？」

---

[1] ただしアプリケーションが「コード分割（Code Splitting）」を行っている場合はその限りではない。たとえばルーティング単位でのコード分割を行っていれば、分割単位の URL をまたいだページ遷移ではサーバにリクエストが発生する。https://ja.reactjs.org/docs/code-splitting.html

「うーん、見た記憶はないと思います……」

「む……、最近の若い子は見たことないか。JavaScript にはブラウザ履歴を管理するための History API[2] というものがあるんだけど、それには従来は前のページに戻る `back()` や次のページに進む `forward()`、履歴内の特定の位置に移動する `go()` くらいのメソッドしかなかったの。だから Ajax なアプリでサーバにいちいちリクエストを送ることなくルーティングを実現するために、URL に `#!`（※『ハッシュバン』もしくは略して『シバン』と発音する）を使う一種の裏技がいっとき流行ったんだよ。URL に `#` ハッシュをつけるとページ内リンクアンカーになって、そこ以降が変わってもブラウザの履歴は変わるけどサーバにリクエストは飛ばないでしょ。それを利用したわけ。

だから今なら `twitter.com/kanae_akiya` みたいなユーザーページは、当時は `twitter.com/#!/kanae_akiya` のような URL で表現されてたのよね」

「へえ――、ちょっとした豆知識ですね」

「いや私にとっては実体験なんだけど……、まあいいや。それでその History API が、HTML5 の時代になって `pushState()` および `replaceState()` というメソッドが実装されたの。これはブラウザのセッション履歴に任意の URL を追加したり、特定の履歴を書き換えたりできるもので、これによって JavaScript で履歴の URL が完全に制御可能になったわけ」

「ふーん、それはいつごろの話なんですか？」

「仕様自体はもっと前からあったんだけど、モダンブラウザに実装され始めたのが 2010 年あたり。Twitter がハッシュバン式の URL から移行したのが 2012 年から 2013 年にかけてくらいだったかな。そして AngularJS のようなフロントエンドフレームワークも History API に対応していった。今現在、ほとんどすべてのフロントエンドフレームワークは、軒並みこの History API を使ってルーティングを実現してる」

「なるほどー。フロントエンドのルーティングにはそんな歴史があったんですね」

「話を最初に戻すと、だから SPA におけるルーティングとは『DOM の動的な書き換えによってページ遷移を擬似的に実現するとともに、ブラウザのセッション履歴をそれに同期させること』ってあたりになるかな。かなり玄人向けの説明になるけど」

「たしかに、私もいま柴崎さんがしてくれたような前置きの解説がないと理解できなかったでしょうね……。

ところで質問なんですけど、SPA とサーバサイドアプリケーションのルーティングのちがいって、

[2] https://developer.mozilla.org/ja/docs/Web/API/History_API

サーバへリクエストが飛ばない他に何かありますか？　開発する上で気をつけるべき点とか」
「うん、いい質問だね。サーバにリクエストが行かないということは、サーバ側からはクライアントがどんなページを見てるかとか、ページをどう移動したかとかがわからないわけだよね。これはGoogle Analytics などを使ったアクセス解析を行う上でネックになる。このソリューションとしては、ルーティングが実行されて URL が書き換わる際 Effect Hook で Google Analytics にリクエストを発行する処理を差し込むといった方法がある[3]。
あとサーバが HTTP ステータスコードを返せないことも注意すべき点かな。既存のページを削除したときにサーバが 404 を返すようにすれば、検索エンジンがそれを検知してインデックスから削除してくれるけど、SPA だとそうはいかない。サイトマップや Google Analytics の管理機能を使うことでフォローする必要がある」
「なるほど」
「他には、ルーティングの適用単位がコンポーネントだというのも前提知識として知っておく必要があるね。たとえば Rails だとひとつの URL に対応するのは controller でそれが view を経由してページ全体を描写するわけだけど、React の場合はルートのコンポーネントから階層を下りていってここから先はこのパスのときにはこのコンポーネントがマウントされる、さらにその下の階層でもこのパスのときに別のコンポーネントがマウントされるといった具合に、コンポーネントそのものがルーティングされるのね」
「なるほど。React がコンポーネントベースのアーキテクチャって、そういうことでもあるんですね」

10-2. ルーティングライブラリの選定

「React 向けのルーティングライブラリはいくつかあるけど、現状は React Router[4] がデファクトスタンダードといっていいだろうね。二番手は Reach Router[5]。React Navigation[6] というのがあって React Native の世界ではもっともメジャーなライブラリなんだけど、Web 対応については 2020 年 12 月現在ではまだ実験的段階なのでとりあえずこれは置いておく」

---

[3] https://github.com/react-ga/react-ga/wiki/React-Router-v4-withTracker
[4] https://reactrouter.com/
[5] https://reach.tech/router
[6] https://reactnavigation.org/

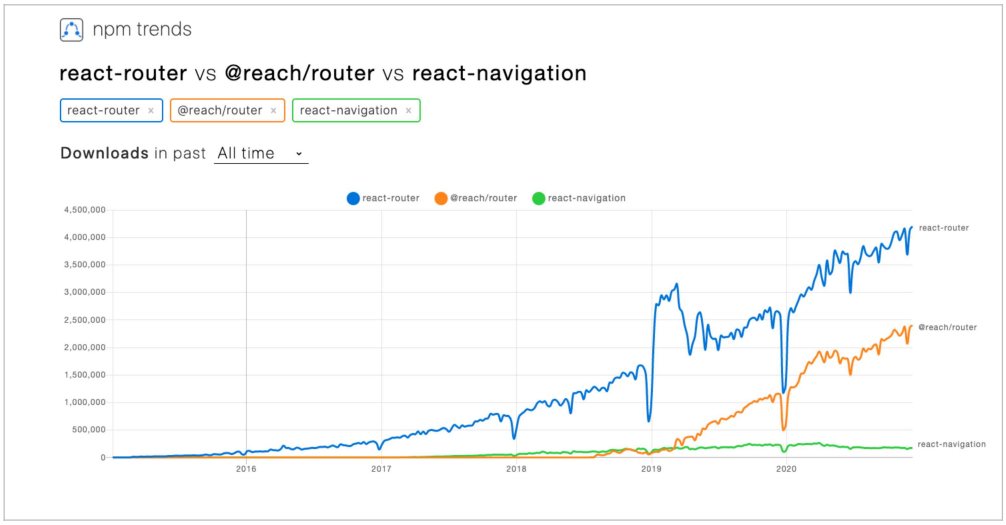

図 3: React 用ルーティングライブラリのトレンド（2020 年 12 月現在）

「グラフを見ると Reach Router がけっこう伸びてきてませんか？　直近のダウンロード数では React Router の半分を超えてますよね」

「うん。Reach Router はシンプルで直感的なインターフェースのルーティングライブラリでね。サーバサイドレンダリングとの相性もよくて、その上 Web アクセシビリティを考慮した各種機能がデフォルトで提供されてるのが特徴。だから静的サイトジェネレータの Gatsby がバージョン 2 で React Router から乗り換えた[7]のも納得できる選択で、これがコミュニティに与えたインパクトも小さくなかった。あと React Router がなかなか Hooks インターフェースを提供できずにいた間に Reach Router がシェアを伸ばした感じかな」

「へーえ」

「ちなみに Reach Router の作者 Ryan Florence は React Router のオリジナル共同作者でもあって、今でもリポジトリのコントリビューションでは 2 位につけてるの[8]。だから API も React Router によく似てて、`Link` コンポーネントなんてほとんど同じ」

「えっ。それはつまり彼が React Router を見限って袂（たもと）を分かった上で、新しく Reach Router を作ったってことですか？」

---

[7] 「How we improved Gatsby's accessibility in v2 by switching to @reach/router」
https://www.gatsbyjs.com/blog/2018-09-27-reach-router

[8] https://github.com/ReactTraining/react-router/graphs/contributors

「んー、このあたりの経緯はちょっと入り組んでるんだよね。Reach Router は React Training という会社が提供してるんだけど、Ryan Florence がその CEO で、いっぽうの React Router の作者の Michael Jackson はその共同創設者なの。そして 2 つとも React Traing 社のプロダクトということになってる」

「同じ会社が競合する 2 つのルーティングライブラリを出してて、しかもそのひとつはもう一方からの分家のようなものなわけですよね。なぜそんなややこしいことをしてるんでしょう？」

「React Training 社としては、用途によって 2 つのライブラリを使い分けてくださいってことだったんじゃないかな。ただやっぱりその姿勢に自らも矛盾を感じてたのか 2019 年 5 月、CEO の Ryan Florence が Reach Router を React Router に将来的に統合するという表明記事を出したんだよね[9]」

「ええー、じゃあ Reach Router はなくなっちゃうんですか？ だとしたらふたたび React Router 一強の世の中になって、どっちを選ぶか悩む必要はなくなるわけですね」

「まあ表面的にはそうかな。ただしそんな単純な話でもなくて、React Router のバージョン 6 の仕様を見てみると、インターフェースを Reach Router に寄せるための破壊的な変更がいくつもあるんだよね[10]。名前的には React Router が残るんだけど、バージョン 6 は下位互換性がないのはもちろんコードベースから作り直されてて、バージョン 4 や 5 とはほぼ別物になるといっていい。既存の React Router の 4 系や 5 系を使ってるプロジェクトで、何も考えずにバージョン 6 へアップグレードしたらエラーが出て動かなくなる」

「ひいいいっ！」

「React Router がメジャーバージョンアップで下位互換性をバッサリ切り捨てるのは、実はこれが初めてじゃないんだよ。だから一部の開発者からは React Router の評判はすこぶる悪かったりする。1 回めの大改変は 2017 年 3 月、バージョン 3 から 4 へのアップグレードのとき。React Router のファーストリリースは 2014 年の 7 月だから、これまでほぼ 3 年に 1 回の頻度で破壊的なバージョンアップを行ってることになる」

「……それは、開発者が怒るのもわかる気がしますね。なぜそんな事態になってるんでしょうか？」

「最初の改変はしかたなかった面もあるんだよね。React が公開されてまもないころで、第 3 世代コンポーネントベースのフレームワークにおけるルーティングがどうあるべきかというのはコミュニティの間でも定まってなかった。作者陣は Ember.js のバックグラウンドを持っていたので、Ember

---

[9] 「The Future of React Router and @reach/router」https://reacttraining.com/blog/reach-react-router-future/

[10] 「Migrating React Router v5 to v6」
https://github.com/ReactTraining/react-router/blob/dev/docs/advanced-guides/migrating-5-to-6.md

のルーティング API をインスパイアして React Router を作ったの。当時バージョン 1 系だった Ember は典型的な第 2 世代フレームワークで、サーバサイドアプリケーション・フレームワークの思想を強く引きずっていた。ゆえにそこから派生した当初の React Router はテンプレートベースのルーティングをむりやりコンポーネントに当てはめた、かなりちぐはぐなものになってしまってたの」

「ふむふむ」

```
ReactDOM.render((
  <Router history={history}>
    <Route path="/" component={App}>
      <IndexRoute component={Home} />
      <Route component={SearchBox}>
        <Route path="users" component={UserList} />
        <Route path="widgets" component={WidgetList} />
      </Route>
      <Route path="dashboard" onEnter={authUser} component={Dashboard} />
      <Route path="user/:id" component={User}>
    </Route>
  </Router>
), document.getElementById('root'));
```

「これが React Router バージョン 3 系を使ったときのサンプルコードね。ルーティングは基本的に『静的ルーティング（static routes）』で、アプリケーションの初期化時に一度だけ、まとめて記述したルールによるルーティングを React のトップ階層のレンダリングに差し込む。またルーティングルールはネストされ（nested routes）、マッチングのパスは基本的に相対パスで記述するようになってる」

「けっこうわかりやすそうに見えるんですけど、何が問題だったんでしょうか？」

「バージョン 3 までの React Router では、ルーティングが通常のコンポーネントツリーからなる React World の上位に陣取り、React Router 独自の法原則が支配する隔離空間で行われてた。それによるルーティングはコンポーネントの構造から派生する各種の挙動よりも、ルーティングの構造から来るそれが常に優先する。そのせいで通常の React Way と齟齬が生じるようになってた。典型的なのはコンポーネントのライフサイクルね」

「というと？」

「React Router バージョン 3 には `<Route>` 自身に `onEnter`、`onUpdate`、`onLeave` という独自のライフサ

イクルプロパティがあって、そこに登録されていた関数がページ遷移にともなって発火するようになってた。ライフサイクルメソッドの再実装だね」

「えっ。コンポーネントのライフサイクルメソッドがあるのに、なんで別途そんなものを作っちゃったんですか？」

「React Router を使うとアプリケーションの設計が、コンポーネントの構造よりもルーティングの構造を優先させることになるせいだよ。コンポーネント主体のルーティングでなく、ルーティングにコンポーネントを従属させる形になるから、ルーティング自体にライフサイクルの処理が必要になるわけ。当時の React Router はこうした独自の拡張によって React Way を破壊するものになってしまってた。だから React Router を使うとコードの見通しは悪くなり、コンポーネントベースな React のメンタルモデルからかけ離れたアプリになってしまうということになってたの。

バージョン 4 への破壊的な大変更は、このことに対する反省から生まれたわけ。4 ではオリジナルの React API と競合する要素を一掃し、純粋なコンポーネントベースのルーティングライブラリとして生まれ変わった」

「ふーむ、なるほど。それは必然的で妥当な変革だったと」

「そうだね。アプリケーション初期化時にトップ階層でまとめて記述するしかなかったルーティングルールが、4 になって動的ルーティングが導入されたことでコンポーネントシステムと融合し、どこでも記述できるようになった。`<Route>` はただのコンポーネントになり、ライフサイクルプロパティも削除された。パスのマッチングが厳密になり、nested routes は排除され絶対パスでしか記述できなくなった」

「おおう、劇的な変化ですね……。邪悪な過去をスパッと断ち切ったわけですか」

「ただあまりにもいっぺんに変わりすぎたために、開発者コミュニティからの反発も大きかった。単に互換性がなかっただけじゃなく、アプリの設計を根本から変える必要があったからね。まあ私にいわせれば、バージョン 3 までのやり方が React Way から外れてておかしかっただけなんだけど。

でも乗り換えコストが大きすぎただけに、バージョン 4 が出てからもアップデートせず 3 系を引き続き使う開発者も少なくなかった。だからなのかバージョン 3 系のメンテナンスは継続して行われていて、直近では 2020 年 3 月に 3.2.6 が出てる」

「4.0 が出たのが 2017 年 3 月だから、メジャーバージョン番号が変わって 3 年たってもまだメンテを続けてるんですね」

「容赦なく下方互換性をバッサリ切り捨てるいっぽうで、旧バージョンのメンテナンスもちゃんと

続けるのがこのチームの特色みたいだね」

「なるほど。それで React Router バージョン 4 は成功したんですか？」

「バージョン 3 に未練のある開発者は一定数いたものの、コミュニティの大勢はこの変革を受け入れた。これでルータに設計を歪められることなく、素直な React アプリを作れるようになったわけだからね。React Router 開発チームも React 自体が根本的に変更されない限り API を大きく変えることはないと約束し、2019 年 3 月にリリースされたバージョン 5.0 でも 4 系との互換性は最大限保証された。その半年後にリリースされた 5.1 ではそれまでの HOC や render props に代わる Hooks による API が提供されたけど、それも下位互換性を壊すものではなかった」

「……で、そこにさっきの Reach Router との統合話が出てきたわけですか。さっきの約束では React が根本的に変わらない限り React Router も変わらないという話だったはずですけど、React 本体はそんなに変わったんですか？」

「うーん、Hooks が導入され関数コンポーネントが主体になったことはそれなりに大きな変化だといえるけど、それが React Router にとって下方互換性を捨てなければいけないほどの変化かといえば、私は疑問だね。おそらくバージョン 3 にあった nested routes の復活を求める声や、厳密性より直感性を優先した Reach Router の普及の勢いを見て、React Training 社の気が変わったんだろうね」

「ええー？　何とも移り気な会社ですね……。それで私たちは、どのバージョンの React Router を使えばいいんでしょうか？」

「とりあえず 2020 年 12 月現在において、6 系はまだ β 版で正式版が出るまではまだもう少しかかると思う。現状での最新の安定版は 5 系で、既存のプロジェクトも 4 系または 5 系が使われているものが圧倒的に多いと思う。だから秋谷さんにはまず 5 系をベースに学んでもらって、その後で 6 系への移行のやり方を知ってもらおうかなと」

「なるほど、わかりました。それで柴崎さんが今から新しくプロジェクトを始めるとしたら、5 系と 6 系のどっちを選びますか？」

「そうだねえ、実は React Router は 2020 年 5 月に 5.2.0、同 6 月に 6.0.0-beta.0 が出てから半年ほどずっと開発が停滞して存続そのものが心配されてた。React Training 社がリソースを Remix[11] とかいうプロダクトに全集中させてたみたいで、その間 React Router のリポジトリはほぼ放置状態だったのね」

「えっ。そんなんで採用してもだいじょうぶなんですか、React Router」

---

[11] https://remix.run/

「んー、私も最悪の事態を考えて Wouter[12] とかの代替になりそうなライブラリの検証もしてたのよね。でもこのあいだ、ようやく開発の再開と v6 の安定版がもうすぐ出るというツイートが Ryan Florence からあったので[13]、とりあえずは信じてもいいんじゃないかなと思う。

それを踏まえた上で React Router のどのバージョンを使うべきかというのは、やっぱり 6 系になるかな。5 系のサポートはしばらく継続するとのことだけど、将来的なことを考えれば β 版でも今から導入しておいたほうが、後になっても楽だし。それに Suspense という React が実験的に導入してる新しい機能[14] は 5 系までの React Router と相性が悪いので、Suspense を使いたいなら 6 系を使わざるをえないという事情もある」

「へー、そうなんですね」

「というわけで、今いったようにまずは React Router 5 系の使い方を見ていこうか」

10-3. React Router（5 系）の API

React Router のインストールと導入方法

「React Router の使い方については、React Training が自社のサイトでいくつかのサンプルを含んだドキュメントを提供しているので[15]、基本的にはそれを見といてもらえればいいんだけど……」

「だけど？」

「ルーティングの初心者には難易度が高いし、ボリュームも大きすぎるかな。なのでここでは、まず基本的な API の用法をひとつずつ説明していってから、その後で実際のアプリケーションに適用したものを見ていくようにしよう。とはいっても公式ドキュメントは大事なので、あとで流し見くらいはしておいてね。ユースケースごとのサンプル、たとえばログインの有無でのルーティングの切り分けの仕方とかは参考になるし、開発時のリファレンスとしても使えるし」

「わかりました。時間のあるときに目を通しておきます」

「じゃあライブラリのインストールから始めていこう。なお React Router のインストールには、react-router と react-router-dom の 2 つのパッケージを入れるようになってるので、次のように実行

[12] https://github.com/molefrog/wouter

[13] https://twitter.com/ryanflorence/status/1337512362014752768

[14] 「第 13 章 データ取得の次世代標準 Suspense」を参照のこと。

[15] https://reactrouter.com/web

する。2020年12月現在では最新版が5.2.0なのでそれがインストールされるけど、5系では2つとも公式で型ファイルが提供されてないので、typesyncで `package.json` にエントリが追加された後、あらためて `yarn install` を実行してね」

```
$ yarn add react-router react-router-dom
($ typesync)
$ yarn
```

「React Routerを導入するには通常、ルーティング機能を提供するプロバイダコンポーネントをトップレベルで設定する。`src/index.tsx` へ次のように記述する感じだね。これでReact Routerの各種APIがコンポーネントツリーの下の階層で使えるようになるの」

リスト1: Routerコンポーネントの導入

```
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import App from './App';

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById('root'),
);
```

「この `<BrowserRouter>` っていうのがそれなんですね？」

「うん。react-router-domが提供するルーティングプロバイダ・コンポーネントには次の4つがあって、これらは共通してより低レベルの `<Router>` コンポーネントを下敷きにしてる。通常は `<Router>` をそのまま使うのではなく、ユースケースに合わせてこれらを使い分けるようになってる」

- `<BrowserRouter>`
- `<HashRouter>`
- `<StaticRouter>`
- `<MemoryRouter>`

「`<BrowserRouter>`[16] はHTML5のHistory APIを使ってUIとURLを動的に同期してくれるルータ

[16] 「<BrowserRouter> - React Rouer」https://reactrouter.com/web/api/BrowserRouter

コンポーネント。一般的な SPA の開発ではこれ一択といっていい。その次の `<HashRouter>`[17] は URL に `#` がつくルーティング機能を提供するものね」

「あー、あの昔の Twitter が使ってたってやつですね？」

「そう。正確にはこれはハッシュバン `#!` じゃなくて `#` だけど。だからこれは今さら使う機会はないだろうね。`<StaticRouter>`[18] はサーバサイドレンダリングを導入する際にサーバのほうで使うルータコンポーネントね。まあこれも直近で使うことはないから省略。最後の `<MemoryRouter>`[19]、これはブラウザのアドレスバーの URL が一切替わらなくてメモリの中だけで履歴が管理されるもの。React Native と組み合わせてテストの際に用いられたりするらしいけど、まあこれもほぼ使うことはないでしょう」

「ふむふむ」

「これらのルーティングプロバイダ・コンポーネントが具体的に何をやってくれるかというと、下位層の子孫コンポーネントの中で後に説明するような `<Link>` や `<Redirect>` といった機能タグが使えるようにしてくれたり、`match` や `location`、`history` といったオブジェクトへアクセスできるようにしてくれたりするわけね」

React Router のコンポーネント API

「次にルーティングルールを設定する `Route`[20] コンポーネントについて見ていこう。これは任意のパスとレンダリングされるコンポーネントとを結びつける、ルーティングのもっとも基本的な機能を提供してる」

リスト 2: Route コンポーネントの使い方

```
import { FC } from 'react'
import { Route } from 'react-router';

import Home from 'components/pages/Home';
import About from 'components/pages/About';
import Contact from 'components/pages/Contact';
```

---

[17] 「<HashRouter> - React Rouer」https://reactrouter.com/web/api/HashRouter

[18] 「<StaticRouter> - React Rouer」https://reactrouter.com/web/api/StaticRouter

[19] 「<MemoryRouter> - React Rouer」https://reactrouter.com/web/api/MemoryRouter

[20] 「<Route> - React Rouer」https://reactrouter.com/web/api/Route

```
const App: FC = () => (
  <>
    <p>弊社のホームページへようこそ！</p>
    <Route exact path="/" component={Home} />
    <Route path="/about">
      <About />
    </Route>
    <Route path="/contact">
      <Contact destAddress="contact@our-company.com" />
    </Route>
  </>
);

export default App;
```

「これはURLのパスが `path` の値にマッチすると `component` に渡されたコンポーネントがレンダリングされる、で合ってますか？」

「正解。レンダリングされるコンポーネントの指定の方法は `component` 属性として渡してもいいし、子要素にしてもいい。なお子要素にして渡したいときは、階層にする以外にもpropsで `children={<Home />}` のようにも書けるよ」

「`component` 属性に設定するのと子要素にするのとでは、挙動にちがいがあるんですか？」

「まず、子要素にすると `<Home foo={bar} />` のように記述することで任意のpropsが設定可能だね。`component` 属性で設定した場合は、propsを自由に設定できない。その代わりにこういう `RouteComponentProps` 型のオブジェクトが渡される」

```
export interface RouteComponentProps<
  Params extends { [K in keyof Params]?: string } = {},
  C extends StaticContext = StaticContext,
  S = H.LocationState
> {
  history: H.History<S>;
  location: H.Location<S>;
  match: match<Params>;
  staticContext?: C;
}
```

「こういったReact Routerが提供する `history` オブジェクトや `location` オブジェクトをpropsとして受け取れるわけ。またあらためて説明するけど、これらは通常Hooks APIでも取得できるように

なってる。でも直接 props として受け取ればコードの省略になるので、`component` 属性で設定するのはそういうときかな。ただ 2020 年 12 月現在の公式ドキュメントのサンプルコードでは子コンポーネントにしてるのが圧倒的に多いので、基本はそっちを使う感じで」

「わかりました」

「実はこの他に `render` という render props 用の属性があって、それを使えばコンポーネントに対して任意の props に加えて `RouteComponentProps` オブジェクトを渡すことができるのね[21]。でもコードが見づらくなるデメリットのほうが大きいので、これを使うくらいなら普通に子要素にしたうえで、必要なオブジェクトはコンポーネント内部で Hooks API を使って取得したほうがいいから、説明は省略するね」

「柴崎さん、render props 嫌いですもんね……」

「あと `path` 属性値のマッチングについて説明しておこう。まず 4 系と 5 系では常に `/` から始まる絶対パスを使うこと。マッチングの条件はデフォルトでは前方一致だけど、次の boolean 型の props を同時に設定すると挙動が変わるようになってる」

- `exact` …… 完全一致
- `strict` …… 末尾のスラッシュ有無のマッチングを厳密にする
- `sensitive` …… 大文字か小文字かまで正確にマッチングさせる

「なるほど。さっきの例のリスト 2 で最初のルーティングに `Home` コンポーネントを設定してるところは、`/` への前方一致だから `exact` を指定しないとパスが `/about` だったときも `/contact` だったときもマッチしちゃうんですね」

「そのとおり。まあ普通の使い方では `exact` くらいしか出番はなさそうかな。でも文字列のマッチングだけでレンダリングの振り分けをするのは厳しいよね。うっかり複数の Route がマッチしてしまって、1 ページに意図せず複数のコンポーネントが並んでレンダリングされることも出てきかねない」

「たしかに Route が増えてくると管理が大変そうですね」

「だから排他的にマッチさせる `<Switch>`[22] というコンポーネントが用意されてる。これは switch-case

---

[21] 「render: func - React Router」https://reactrouter.com/web/api/Route/render-func

[22] 「<Route> - React Rouer」https://reactrouter.com/web/api/Switch

文を連想してくれればわかりやすいかな」

リスト 3: Switch コンポーネントの使い方

```
import { FC } from 'react'
import { Route } from 'react-router';

import Home from 'components/pages/Home';
import User from 'components/pages/User';
import NotFound from 'components/pages/NotFound';

const App: FC = () => (
  <Switch>
    <Route exact path="/">
      <Home />
    </Route>
    <Route path="/user/:userId">
      <User />
    </Route>
    <Route>
      <NotFound />
    </Route>
  </Switch>
);

export default App;
```

「たしかに switch-case 文っぽいですね。最後に `path` を指定しない `Route` コンポーネントを置くことで default 節のように表現できるところなんかも同じ」

「そうだね。`<Switch>` でくくると、`<Route>` は `path` が初回にマッチしたところでルーチンを抜けるので、以降の `<Route>` の記述で `path` がマッチしたとしても関係なくなって挙動が把握しやすくなる。だからほとんどの場面では `Route` は `Switch` といっしょに使うことになるね」

「2 つめのルーティングで `User` コンポーネントを渡してるところ、`path="/user/:userId"` ってなってますけど、これは？」

「コロンから始まる文字列は URL パラメータといって、ここにマッチした文字列はレンダリングされるコンポーネントで `match` オブジェクトから `userId` の値を抽出できる。たとえば `/user/patty` というパスだったら、`User` コンポーネントで `match.params.userId` に `patty` が格納されることになるわけ。

ちなみにパラメータのマッチングには正規表現が適用できて、`"/user/:userId(patty|rolly)"` と書けばこの二択になるし、8 桁の数字または小文字アルファベットに限定したいなら `"/user/:userId([0-9a-f]{8})"` のように指定する」

「へ――、それは使い勝手がよさそうですね」

「それから、`<Switch>` 構文の中で `<Route>` と並んでよく使われるのが `<Redirect>`[23]。文字通りリダイレクトを実行してくれるコンポーネントね。こういうふうに使う」

リスト 4: Redirect コンポーネントの使い方

```
import { Redirect, Route, Switch } from 'react-router';
  ⋮
  const App: FC = () => (
    <Switch>
      <Route exact path="/" component={Home} />
      <Redirect from="/user/profile/:userId" to="/user/:userId" />
      <Route path="/user/:userId" component={User} />
      <Redirect push to="/" />
    </Switch>
```

「`from` でマッチングして `to` のパスにリダイレクトさせるわけですね。URL パラメータも引き継げるんだ」

「そう。なお `from` が指定されてなければ問答無用でリダイレクトされる。リダイレクトのデフォルトの挙動は HTML5 の History API の `replaceState` が適用されるけど、boolean 型の props である `push` を指定すれば `pushState` になる」

「えーっと、それはつまりどういうちがいがあるんですか？」

「たとえばこの例ならどこにもマッチしないパス `/nowhere` にアクセスしたとき、最後の `<Redirect>` に `push` の指定があれば `/nowhere` → `/` とそこにアクセスした履歴が残って『戻る』ボタンを押せばまた `/nowhere` に来ることになる。その後またすぐ `/` へリダイレクトされるけどね。でも `push` が指定されてなければ `/nowwhere` にアクセスした過去が抹消されて、`/` で『戻る』ボタンを押せば本来ならもうひとつ前にアクセスしていたページまで戻ることになるわけ」

「ふーむ、なるほど。React Router が高度なアプリケーション向けに柔軟なルーティングができるというのはこういうことなんですね」

「うん、まあ `<Redirect>` に限らず Reach Router ではここまで細かく制御できないね。

---

[23] 「<Redirect> - React Rouer」https://reactrouter.com/web/api/Redirect

じゃ次は、リンク機能を提供する `Link` コンポーネント[24] の使い方を見ていくよ。こちらは react-router-dom のパッケージからインポートする」

リスト 5: Link コンポーネントの使い方

```
import { Link } from 'react-router-dom';
  ⋮
    <ul>
      <li>
        <Link to="/">トップページ</Link>
      </li>
      <li>
        <Link to="{{
          pathname: '/contact',
          search: '?from=here'
          hash: '#subject',
          state: { secretCode: '8yUfa9KECH' },
        }}">
          お問い合わせ
        </Link>
      <li>
        <Link to="/anywhere" replace>今ここではないどこか</Link>
      </li>
      </li>
    </ul>
```

「これ素朴な疑問なんですけど、`<a>` リンクをそのまま使っちゃいけないんですか？」

「`<a>` タグを使って書くと、そのリンクを踏んだ時点で React Router の管轄外となって、管理していた履歴がすべて消えてしまうよ。普通に Web サーバにリクエストが飛んで、SPA のコード全体がリロードされることになるわけだから」

「あっ、そうか。そうなっちゃうんですね……」

「うん、いちおうは動くから初心者が気づかずハマりがちな点だけど、アプリケーション内リンクは必ず `<Link>` を使って書くようにしましょう。

あらためてその使い方を説明すると、リンク先を設定する属性の `to` にはパスの文字列または `location` オブジェクト[25] を渡すことができる。`location` オブジェクトならパスの他にクエリパラメ

[24] 「<Link> - React Rouer」https://reactrouter.com/web/api/Link

[25] 「location - React Rouer」https://reactrouter.com/web/api/location

ータやハッシュも設定できるし、ユーザーに見せたくない情報を埋め込んでリンク先に受け渡すことができる」
「アクセス解析とかに使えそうですね」
「そうだね。あと boolean 型の属性である `replace` を指定すれば、クリックした時点でそこにいたページの履歴が消えることになるよ」

React Router の Hooks API

「通常のコンポーネントの中から React Router が提供する `match`, `location`, `history` オブジェクトへアクセスできるようにする Hooks API について説明していこうか。前に示した `<Route path="foo" component={Bar} />` 形式でコールされた場合や、HOC の `withRouter` を使うことでも props としてこれらにアクセスできるんだけど、2019 年 9 月にリリースされたバージョン 5.1 からようやく Hooks 形式のインターフェースでもそれが可能になったの[26]。提供されている API は次の 4 つ」

- `useHistory`
- `useLocation`
- `useParams`
- `useRouteMatch`

「あれ？ 3 つじゃなくて 4 つなんですね」
「`useParams` は `match` オブジェクトから URL パラメータだけを抽出して使い勝手をよくしたもの。`useRouteMatch` なら `match` オブジェクトをまるごと取得できるけど、マッチした URL パラメータにしか用がないときは `useParams` を使ったほうが便利だね。とりあえず、最初からひとつずつ使い方を見ていこう」

リスト 6: useHistory の使い方

```
import { FC } from 'react'
import { useHistory } from 'react-router-dom'

const historyButtons: FC = () => {
  const history = useHistory();
```

[26] 「React Router v5.1: React Training Blog」https://reacttraining.com/blog/react-router-v5-1/

```
  return (
    <button type="button" onClick={() => history.goBack()}>
      戻る
    </button>
    <button type="button" onClick={() => history.goForward()}>
      進む
    </button>
    <button type="button" onClick={() => history.push("/")}>
      トップページへ
    </button>
  );
};

export default historyButtons;
```

「`useHistory` が返すのは HTML5 の Hisotry API が提供する生の `History` オブジェクト[27] じゃなく、React Router が独自に定義している同じ名前の `history` オブジェク[28] トね。その中で提供されている主な要素は次のとおり」

- `length` …… スタックされている履歴の数
- `action` …… 直近に実行されたアクションの種類（`"PUSH"`, `"REPLACE"`, `"POP"`）
- `push(PATH)` …… 引数 PATH で指定したパスに移動するメソッド
- `replace(PATH)` …… 引数 PATH で指定したパスにリダイレクトするメソッド（現在いるページの履歴は消える）
- `goBack()` …… ひとつ前の履歴のページに戻るメソッド
- `goForward()` …… ひとつ先の履歴のページに進むメソッド
- `go(N)` …… 引数 N で指定した番号の履歴に移動するメソッド

「なるほど。ブラウザ履歴系の機能を使いたいときは `useHistory` と。わかりました」
「じゃ次は `useLocation`」

---

[27] 「History - Web API | MDN」https://developer.mozilla.org/ja/docs/Web/API/History

[28] 「history- React Rouer」https://reactrouter.com/web/api/history

リスト 7: useLocation の使い方

```
import { FC } from 'react';
import { Switch, useLocation } from 'react-router-dom'
import ReactGA from 'react-ga';

import Home from 'components/pages/Home';
import User from 'components/pages/User';
import NotFound from 'components/pages/NotFound';

const App: FC = () => {
  const location = useLocation();

  useEffect(() => {
    ReactGA.pageview(location.pathname + location.search);
  }, [location.key]);

  return (
    <Switch>
      <Route exact path="/" component={Home} />
      <Route path="/user/:userId" component={User} />
      <Route component={NotFound} />
    </Switch>
  );
}

export default usePageViews;
```

「……これはアレですよね。SPA だと URL が変わってもサーバにリクエストが行かないから、そのままでは Google Analytics が使えないという問題に対応したやつですよね？」

「えらいえらい、よくおぼえてたね。React Router が提供する `location` オブジェクトにはその時点での URL 情報が格納されてるの。だからその内容が変わったときに Google Analytics のサーバへページ情報を送信する処理を追加したのが上記のサンプルね。

`location` オブジェクトの構造については `Link` コンポーネントの説明のときにもふれたけど、その内容はたとえば URL `https://exampleapp.com/user/patty?from=user-list#friends` だったときには次のようになってる」

```
{
  pathname: '/user/patty',
  search: '?from=user-list',
```

```
  hash: '#friends',
  state: {
    [secretKey]: '9qWV408Zyr',
  },
  key: '1j3qup',
}
```

「この `key` っていうのは何ですか？」

「`location` オブジェクトごとに生成されるユニークな文字列だね。`useLocation` で取得した `location` オブジェクトには必ず格納されてる。だからここでは `useEffect` の依存配列には、`location` オブジェクトそのものよりも意味的にふさわしいこっちを渡してあげるようにしてる。挙動は変わらないはずだけどね」

「あ、ほんとだ」

「じゃ次、`useParams` と `useRouteMatch` ね。2つとも React Router が提供する `match` オブジェクトをハンドリングするための API。これらを理解するには `src/App.tsx` から `<Route path="/user/:userId" component={User} />` でルーティングされた User コンポーネントの実装がどうなるかを見ればてっとり早いかな」

リスト 8: useParams の使い方

```
import { FC } from 'react';
import { useParams, useRouteMatch } from 'react-router-dom'

const User: FC = () => {
  const { userId } = useParams();
  const match = useRouteMatch();

  // for debug
  console.log(userId);
  console.log(match);
  ⋮
```

「これでアクセスしたパスが `/user/patty` だったとき、コンソールには次のように表示されてるはず」

```
patty

{
  path: "/user/:userId",
```

```
  url: "/user/patty",
  isExact: true,
  params: {
    userId: "patty",
  }
}
```

「おおー、なるほど。`useRouteMatch` が `match` オブジェクト[29]をまるごと取得してるのに対して、`useParams` がそこから URL パラメータだけを抽出してるのがよくわかりますね」
「うん。パラメータの `userId` を `useRouteMatch` で取得しようとすると次のようなめんどうな記述になるけど、これを 1 行のシンプルなコードで書けるのが `useParams` なわけ」

```
const match = useRouteMatch();
const { userId } = match.params as { userId: string };
```

「型注釈もつける必要があって、たしかにめんどうですね……。この API が別途作られた理由が納得できました」
「じゃ React Router の使い方は以上ね。今度は実際のアプリケーションの中でルーティング機能を使ってみよう」

10-4. React Router をアプリケーションで使う

「今回はコンポーネントの props について学んだときに使った『SLAM DUNK』のキャラ一覧のサンプルコードをベースに、複数のページを作って React Router でルーティング機能を追加していこう」

[29] 「match - React Rouer」https://reactrouter.com/web/api/match

図 4: SLAM DUNK キャラ一覧トップ

図 5: 湘北高校の選手

「高校別に登場人物の紹介ページができたんですね。トップから高校へのページに遷移できて、『ホームへ』のボタンを押すとトップに戻ってきますね」

「うん。今回はトップページと全キャラ一覧、高校別キャラ一覧の 3 種類のページを用意してる。それぞれのパスは次のとおり」

- `/`
- `/characters`
- `/characters/:schoolCode`

「個別のファイルの説明へ入る前に、ディレクトリ構造を確認しておこうか。そろそろ実際の開発を見据えて、コンポーネントの切り出し方やファイルの置き方をちゃんと考えていきたいからね。このサンプルアプリにおけるファイルのレイアウトはこんなふうになってる」

リスト 9: ディレクトリ構造

```
src/
  components/
    molecules/
      HomeButton.tsx
      Spinner.tsx
    organisms/
      CharacterList.tsx
      SchoolList.tsx
    pages/
      Characters.tsx
      Home.tsx
    templates/
      AllCharacters.tsx
      SchoolCharacters.tsx
  containers/
    molecules/
      HomeButton.tsx
    organisms/
      SchoolList.tsx
    templates/
      AllCharacters.tsx
      SchoolCharacters.tsx
  data/
    characters.ts
  App.tsx
  index.tsx
```

「presentational components を `components/` に、container components を `containers/` に置くのは変わ

ってませんよね。でも今回はそれらの中でさらに階層が分かれてます。この `molecules/` とか `organisms/` って何ですか？」

「これは **Atomic Design**[30] という UI デザイン手法を適用したものでね。粒度の小さいものから atoms（原子）、molecules（分子）、organisms（有機体）、templates（テンプレート）、pages（ページ）という 5 つのカテゴリーに UI パーツを分けて設計しようというものなの。

atoms はそれ以上分割できないデザインとして最小単位のコンポーネント、具体的には UI キットが提供する `<Button>` や `<Icon>` のようなパーツね。もっとも大きい単位の pages はルーティングの対象となるページ全体を表現するコンポーネントに相当する」

「なるほど、うまくハマればきれいに設計できそうですね。ただ 5 つもカテゴリーがあると、どれがどれに相当するか悩みそうですけど明確な基準ってあるんでしょうか？」

「鋭い質問だね。最小単位の atoms と最大単位の pages を除いて絶対的な基準といえるものはなくて、ただ molecules は atoms を組み合わせたもの、organisms は atoms と molecules を組み合わせたもの、といった具合に自分のカテゴリーより小さい単位を組み合わせて作られるもの、というふうに考えていけばいいかな」

「なるほど」

「デザインについてはそれくらいにして、次はデータについて。今回は `data/characters.ts` のファイルに型情報といっしょに格納してある」

リスト 10: data/characters.ts

```
export type Character = {
  id: number;
  name: string;
  grade: number;
  height?: number;
};

type SchoolPlayers = {
  school: string;
  players: Character[];
};

export type CharactersData = { [schoolCode: string]: SchoolPlayers };
```

[30] 「Atomic Design | Brad Frost」https://bradfrost.com/blog/post/atomic-web-design/

```
export const charactersData: CharactersData = {
  shohoku: {
    school: '湘北高校',
    players: [
      {
        id: 1,
        name: '桜木花道',
        grade: 1,
        height: 189.2,
      },
      ⋮
```

「高校別のページでパスの特定に使われるのが `schoolCode` として設定されている `shohoku` や `ryonan` ね。これをURLパラメータとして用いるわけ。まず `App.tsx` で次のようにルーティングルールを設定する」

リスト 11: App.tsx

```
⋮
import Home from 'components/pages/Home';
import Characters from 'containers/pages/Characters';
import './App.css';

const App: FC = () => {
  ⋮
  return (
    <div className="container">
      <Switch>
        <Route exact path="/">
          <Home />
        </Route>
        <Route path="/characters" component={Characters} />
        <Redirect to="/" />;
      </Switch>
    </div>
  );
};

export default App;
```

「そして `/characters` に前方一致でマッチングした先のルーティングは `components/pages/Characters.tsx` で行ってる」

リスト 12: components/pages/Characters.tsx

```
⋮
import AllCharacters from 'containers/templates/AllCharacters';
import SchoolCharacters from 'containers/templates/SchoolCharacters';
import HomeButton from 'containers/molecules/HomeButton';

const Characters: FC<RouteComponentProps> = ({ match }) => (
  <>
    <header>
      <h1>『SLAM DUNK』登場人物</h1>
    </header>
    <Switch>
      <Route exact path={`${match.path}`}>
        <AllCharacters />
      </Route>
      <Route path={`${match.path}/:schoolCode`}>
        <SchoolCharacters />
      </Route>
      <Redirect to="/" />
    </Switch>
    <Divider hidden />
    <HomeButton />
  </>
);

export default Characters;
```

「このコンポーネントは `Route` コンポーネントから `component` 属性によってコールされてるので props の中に `history` オブジェクト、`location` オブジェクト、`match` オブジェクトが格納されてる。そこから `match` オブジェクトだけを抽出してるのね。

`App.tsx` で `path` にマッチングしてるのは `'/characters'` なので、ここでの `match.path` は `'/characters'` になる。だからパスが `/characters` なら `AllCharacters` コンポーネントが、`/characters/:schoolCode` なら `SchoolCharacters` コンポーネントがレンダリングされることになるわけ」

「なるほど。そしてそのどちらにもマッチングしなければトップへリダイレクトされると」

「そう。じゃ高校別のキャラ一覧ページの実装を見てみよう。`containers/templates/SchoolCharacters.tsx` がまず呼ばれてるよね」

リスト 13: containers/templates/SchoolCharacters.tsx

```
import { FC } from 'react';
import { Redirect, useLocation, useParams } from 'react-router-dom';

import SchoolCharacters from 'components/templates/SchoolCharacters';
import { charactersData } from 'data/characters';

const EnhancedSchoolCharacters: FC = () => {
  const { schoolCode } = useParams<{ schoolCode: string }>();
  const schoolCodeList = Object.keys(charactersData);

  const { search } = useLocation();
  const queryParams = new URLSearchParams(search);
  const isLoading = !!queryParams.get('loading');

  if (schoolCodeList.includes(schoolCode)) {
    const { school, players } = charactersData[schoolCode];

    return (
      <SchoolCharacters
        school={school}
        characters={players}
        isLoading={isLoading}
      />
    );
  }

  return <Redirect to="/" />;
};

export default EnhancedSchoolCharacters;
```

「通常、データは外部 API などを叩いて取得するものなので、ローカルファイル `data/characters.ts` からインポートするのも今回は副作用処理とみなして container component に入れてる。インラインコメントでマークしてる部分、これが任意の高校の学校名とキャラクター一覧データを抽出してる肝心の処理ね。そこに至るまでに何をしてるかわかる？」

「えーっと、まずこのコンポーネントの冒頭で `useParams` でパスから `:schoolCode` の値を抽出してるんですよね。その 3 つ下で定義してる `schoolCodeList` というのは `charactersData` から抽出した `schoolCode` の一覧。

だからこのコンポーネントは、URL パラメータの `:schoolCode` が `schoolCodeList` の中にあれば、presentational component に学校名とキャラクター一覧のデータを属性値に設定してレンダリングしてる。ただし `:schoolCode` が適切でなければ、トップにリダイレクトする、というものじゃないでしょうか？　ただ `isLoading` 周りが何をしてるかはわかりませんでした……」

「説明してくれた部分については正解だね。それでこの `isLoading` というのは、外部 API にリクエストしてデータが返ってくるまでのローダー表示をシミュレートしたくて、URL に `?loading=true` のようなクエリを追加するとそれが表示されるようにしてみたのね」

「へー？」

「ちょっと URL ボックスの中を `http://localhost:3000/characters/shohoku?loading=true` のように書き換えてみてくれる？」

「あっ、キャラ一覧が表示されてたところがローダーに変わりました！」

「まず `useLocation` を使って `location` オブジェクトを取得し、そこから URL クエリパラメータ文字列（e.g. `'?foo=bar&baz=qux'`）を抽出する。それを元に `URLSearchParams` オブジェクト[31] を作成し、その `get` メソッドで `loading` に設定されてる値を抽出してるわけ」

「抽出した値につけてる `!!` というのは何ですか？」

「二重否定（double nagation）という技法で値の型を boolean にキャストしてるんだよ。`!` で否定すれば truthy な値は `false` になるよね。そこから再度否定すれば `true` になる。こうすることで `?loading=true` でも `?loading=1` でも、とにかく何か値が設定されていれば有効になるようにしてるの」

「なるほど、かっこいい書き方ですね！」

「じゃ次に、`SchoolCharacters` の presentational component を確認しよう」

リスト 14: components/templates/SchoolCharacters.tsx

```
import { FC } from 'react';
import { Helmet } from 'react-helmet';
import { Header } from 'semantic-ui-react';

import CharactersList from 'components/organisms/CharactersList';
import { Character } from 'data/characters';

type Props = {
```

[31] 「URLSearchParams - Web API | MDN」https://developer.mozilla.org/ja/docs/Web/API/URLSearchParams

```
  school: string;
  characters: Character[];
  isLoading?: boolean;
};

const SchoolCharacters: FC<Props> = ({ school, characters, isLoading = false }) => (
  <>
    <Helmet>
      <title>登場人物一覧 | {school}</title>
    </Helmet>
    <Header as="h2">{school}</Header>
    <CharactersList characters={characters} isLoading={isLoading} />
  </>
);

export default SchoolCharacters;
```

「最初のほうにある `<Helmet>` って何ですか？」

「これは React Helmet[32] ってライブラリで、どこからでも HTML ドキュメントヘッダを動的に上書きできるようにしてくれるものなの。SPA では意図して書き換えないと常に `public/index.html` に書かれた `<title>` の中身がどこのページでもそのページタイトルになってしまうからね」

「なるほど。SPA で気をつけないといけない点がここにも……」

「そこ以外はだいじょうぶそうだね。`CharactersList` コンポーネントの中身も、とりたてて特別なことはしてないけどいちおう見ておこう」

リスト 15: components/organisms/CharactersList.tsx

```
import { FC } from 'react';
import { Icon, Item } from 'semantic-ui-react';

import Spinner from 'components/molecules/Spinner';
import { Character } from 'data/characters';

type Props = {
  characters: Character[];
  isLoading?: boolean;
};

const CharactersList: FC<Props> = ({ characters = [], isLoading = false }) => (
```

[32] https://github.com/nfl/react-helmet

```
  <>
    {isLoading ? (
      <Spinner />
    ) : (
      <Item.Group>
        {characters.map((character) => (
          <Item key={character.id}>
            <Icon name="user circle" size="huge" />
            <Item.Content>
              <Item.Header>{character.name}</Item.Header>
              <Item.Meta>{character.grade}年生</Item.Meta>
              <Item.Meta>
                {character.height ?? '???'}
                cm
              </Item.Meta>
            </Item.Content>
          </Item>
        ))}
      </Item.Group>
    )}
  </>
);

export default CharactersList;
```

「props の `isLoading` の値によって、ローダーとコンテンツの出し分けをしてるんですね。なるほどなるほど」

「そのとおり。ここまでわかれば、`AllCharacters` や `SchoolList` とか他のコンポーネントの処理も自分で追っていけそうだね」

「はい。React Router の使い方、だいたい理解できたと思います。アプリケーションがコンポーネントをベースにしているところは変わらなくて、ツリーの子孫コンポーネントが URL パスによって振り分けられてるだけですよね。サーバサイドフレームワークのルーティングが一箇所にまとまってるのとちがって、任意のコンポーネントでいつでもルーティングできるのがややこしいですけど」

「うん。React Router はバージョン 4 で React Way に素直に則（のっと）ったライブラリになったんだけど、その反面、ルーティングルールを一箇所にまとめられなくなってしまったのね。厳密にいうと、`App.tsx` の中でこう書いておけばまとめられなくはないんだけど」

```
  <Switch>
    <Route exact path="/"><Home /></Route>
```

```
    <Route exact path="/characters" /><AllCharacters /></Route>
    <Route exact path="/characters/:schoolCode" /><SchoolCharacters /></Route>
    <Redirect to="/" />;
  </Switch>
```

「ただこうしてしまうと、最初の書き方では `Characters` コンポーネントに記述していたUIコンテンツを `AllCharacters` と `SchoolCharacters` に重複して書かざるをえなくなるというジレンマがある。どっちをとるかは、まあ書く人の好みかな」

「私としては、こっちのほうがわかりやすくていいですね。子コンポーネントで `match` オブジェクトを取り回す必要もないですし」

「うん。最初の書き方はこの後で説明するバージョン6との対比で、nested routesを強引に模したものだからね。その話はあらためてするとして、SPAのルーティングではサーバサイドで考慮する必要がなかった注意点がもうひとつ残ってるので、そこの説明をさせてね。

ブラウザのスクロール位置なんだけど、React Routerではルーティング遷移時にスクロール位置が変わらないの。というかReact Routerに限らずたいていのSPA用のルーティングライブラリでも挙動は同じ」

「？？？　ちょっとピンとこないです」

「たとえばこのサンプルで、高校別のページの末尾に『前の高校』『次の高校』というリンクをつけたとする。今は高校別に5人ずつしか登録してないけど、これが30人とかになれば下のほうを見るためにはスクロールが必要になるよね」

「はい」

「その状態で『次の高校』というリンクを踏むと、ページが変わってるのに一番下へスクロールしたままになってしまうの」

「えっ、それユーザーはびっくりしますよね」

「まあそういうこと。サーバサイドアプリケーションなら、ページを遷移すると当然のように毎回トップ位置から始まるわけだけど、History APIの `pushState` を使ったルーティングでは履歴が変わって新しいページがレンダリングされても、スクロール位置は遷移前のままになる。これは一般ユーザーの期待する挙動じゃなくて、違和感を持たれるよね」

「それはもう、そうですね」

「だから今回、`App.tsx` にその問題を対処するルーチンを入れてあるの。もう一度そこのファイルを見てみて」

リスト 16: スクロール位置をクリアする

```
import { FC, useEffect } from 'react';
import { Redirect, Route, Switch, useHistory, useLocation } from 'react-router';
⋮
const App: FC = () => {
  const { hash, pathname } = useLocation();
  const { action } = useHistory();

  useEffect(() => {
    if (!hash || action !== 'POP') {
      window.scrollTo(0, 0);
    }
  }, [action, hash, pathname]);

  return (
   ⋮
```

「ふーむ、`useEffect` を使って、コンポーネントの初回レンダリング時に強制的にトップにスクロールさせてるんですか」
「そう。ページ遷移したときに URL の中に `#` があったり、『戻る』『進む』ボタンで移動したんじゃなければ、その処理を実行してる。まあめんどうだけど SPA の特性としてそういうのがあるということをおぼえておいて」
「は――、やっぱりサーバサイドアプリケーションとは勝手がけっこうちがうんですねー」

10-5.　React Router バージョン 5 から 6 への移行

v5 から v6 への変更点

「前にもふれたように React Router はメジャーバージョン 6 で、5 までのとの互換性を切り捨てた上で Reach Router と統合され新しく作り直される。2020 年 12 月現在、6 系の最新版は 6 月にリリースされた 6.0.0-beta.0 であり、前にもいったようにまだ安定版は出てない。β 版なのでこの先いくらか仕様が変わることはありえるんだけど、公式からすでに移行ガイドのドキュメントが提供さ

れてる[33]ので、それを参考にしながら移行の要点をまとめてみるね」

「はい、お願いします」

「その前にまずバージョン 6 の仕様以外の変更点を紹介しておこうか。大きなところではコードベースが TypeScript によって書き直されてる。バージョン 5 までは純粋な JavaScript 製で、かつ型も公式が提供してなかったので TypeScript から使う際は DefinitelyTyped のものをインストールする必要があったんだけど、その必要もなくなるね」

「おおー、TypeScript 化の波がこんなとこまで……」

「また書き直しによってコードの中身がかなり整理されたおかげで、react-router と react-router-dom のパッケージの容量がバージョン 5.2.0 では minify & gzip 後で合わせて 16.9KB だったのが、6.0.0-beta.0 では 7.2KB と大幅に削減されてる。正式版までにもう少し増えるかもしれないけど」

「それでもすごいですね。半分以下じゃないですか」

「過去をバッサリ切り捨ててるからね。IE はバージョン 11 より前のサポートは打ち切り、React も Hooks が有効になった 16.8 より前のバージョンはサポートしない」

「でもまあそれは仕方ないでしょう。Microsoft が IE10 のサポートを終了したのって、たしか 2016 年くらいでしたし」

「インストールの方法なんだけど、内部で使ってる history パッケージがなぜか dependencies から外されてて、別途入れる必要がある。これも正式版までに変わるかもしれないけどね。2020 年 12 月現在、β 版である React Router バージョン 6 系をインストールするためのコマンドはこうなる」

```
$ yarn add react-router@next react-rouer-dom@next history
```

「正式版が出た後は、この `@next` は省いてね」

「了解です」

「じゃ、バージョン 6 の API がどう変わったかを見ていこう。大きいのは排他的ルーティングを行うための `<Switch>` が廃止され、新たに `<Routes>` が導入されたこと。`<Switch>` と `<Routes>` の主なちがいは次のとおり」

i. `<Switch>` はマッチした `<Route>` があり次第、それ以降の評価をせず処理を抜ける。`<Routes>` では最後まで評価した上で、ベストマッチする `<Route>` にルーティングされる

---

[33] 「Migrating React Router v5 to v6」
https://github.com/ReactTraining/react-router/blob/dev/docs/advanced-guides/migrating-5-to-6.md

ii．v4 で消えた nested routes が復活し、それにより一箇所にルーティングルールを集約して記述できるようになった
（※ 従来通り子孫コンポーネントへ分散させて記述することも可能）

iii．`<Routes>` 内のすべての `<Route>` および `<Link>` のパスは、相対パスで記述できる
（※ 従来通り絶対パスでの記述も可能）

「これは……、かなり変わりましたね」
「うん。だからただ `<Switch>` を `<Routes>` に書き換えればいいわけじゃなくて、ちゃんと思想を理解した上でメンタルモデルを切り替える必要がある。まず i は API の名前が変わった理由でもあるよね。switch-case 的な挙動じゃなく、すべての候補を評価した上でもっともマッチするものが選択されるようになった」
「なぜそんなふうに変更されたんでしょうか？」
「`<Switch>` の仕様だと並び順によってマッチするものが変わるので、エントリーの新規追加やリファクタリングのための並べ替えによって簡単にマッチングルールが壊れてしまう。`<Routes>` なら意味合いによる可読性を最優先に並べることができるし、新規追加や書き換えを行っても既存のルールは壊れにくいでしょ」
「なるほど。じゃあ ii の話になりますけど、v3 まではあったのに v4 で廃止されたはずの nested routes が復活したのはどうしてなんですか？」
「もともと作者陣は nested routes がお気に入りの機能だったんだけど、React Way に合わせることを最優先にした結果、v4 では仕方なくそれを手放したみたいなのね。でも Reach Router での知見も加えて、うまく現状の React と nested routes を融和させる目処が立ったので復活させた、という経緯なんじゃないかな。実際、`match` オブジェクトを取り回すことなくシンプルにパスが記述できるので、慣れればメリットは大きいと思うよ。これによってどう変わるかをサンプルで見てみよう」

リスト 17: v5 によるルーティングの例

```
const App: FC = () => (
  <Switch>
    <Route exact path="/"><Home /></Route>
    <Route path="/users" component="Users" />
  </Switch>
);
```

```
const Users: FC<RouteComponentProps> = ({ match }) => (
  <div>
    <nav>
      <Link to={`${match.url}/me`}>My Profile</Link>
    </nav>
    <Switch>
      <Route path={`${match.path}/me`}><SelfProfile /></Route>
      <Route path={`${match.path}/:id`}><UserProfile /></Route>
    </Switch>
  </div>
);
```

「これは v5 による簡単なルーティングのサンプルだね。構造を変えずにそのまま v6 で書き換えると次のようになる」

リスト 18: 構造そのままに v6 でリライト

```
const App: FC = () => (
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="users/*" element={<Users />} />
  </Routes>
);

const Users: FC = () => (
  <div>
    <nav>
      <Link to="me">My Profile</Link>
    </nav>
    <Routes>
      <Route path=":id" element={<UserProfile />} />
      <Route path="me" element={<SelfProfile />} />
    </Routes>
  </div>
);
```

「ルーティングを渡した先のコンポーネントで `match` オブジェクトの取り回しが消えましたね」

「そう。これが相対パスが使えることによるメリットだね。ここからさらに 2 つのコンポーネントに分散してしまってるルーティングルールの記述を nested routes によって集約してみよう」

リスト 19: さらに nested routes を使ってリライト

```
const App: FC = () => (
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="users" element={<Users />}>
      <Route path="me" element={<SelfProfile />} />
      <Route path=":id" element={<UserProfile />} />
    </Route>
  </Routes>
);

const Users: FC = () => (
  <div>
    <nav>
      <Link to="me">My Profile</Link>
    </nav>
    <Outlet />
  </div>
);
```

「おおー、ルールがまとまって見やすくなりましたね！」

「うん。ルーティングをネストすることで、たとえば `/users/me` のパスならまず `Users` コンポーネントがマッチするよね。さらにネストされた中で `SelfProfile` がマッチする。それにより最初に `Users` コンポーネントがレンダリングされ、その `<Outlet />` の中で `SelfProfile` がレンダリングされる。`<Outlet>` は子ルートのコンポーネントをレンダリングするためのプレースホルダーなわけ」

「なるほど、このやり方なら v3 のようにページまるごとルーティングしなくても、コンポーネントベースの設計のまま nested routes が使えますね。頭いい！」

「うん。同じ nested routes でも v3 と v6 のそれはちがってて、v6 では React Way を壊すことなく実現されてる。ルーティングルールが複数のコンポーネントに分散されてると可読性やメンテナンス性が落ちるので、v6 からはこの nested routes を積極的に使ってルールをまとめて書くようにしていこう」

「了解です！」

「次は `<Route>` についてね。v5 と同じ名前だけど、これも使い方がかなり変わってる。まずレンダリング対象のコンポーネントの指定が、従来では `component` や `render` 属性および子要素で行ってたのが、`element` 属性に統一する形で変更された。そして渡す値も `element={<MyComponent />}` のように

JSX タグを記述する形で React Elements を渡すようになった」
「そうか、この形式なら任意の props を渡すこともできますね。でもこれまでの子要素で渡す形式じゃダメだったんですか？」
「`<Route>` の子要素は nested routes で使うようになっちゃったからね。そっちを優先させた結果、こういうインターフェースになったんだと思う。
それからパス `path` 属性値のマッチングなんだけど、これも仕様が大きく変わってる。switch-case 形式じゃなく最後まで評価した上でのベストマッチが選択されることと、相対パスが記述できるようになったのはここまでで説明したけど、その他にもこんな変更点がある」

- ベースが前方一致ではなく完全一致になった。ただし `<Route>` がネストしている場合は、親ルートのマッチングは前方一致になる
- `exact` および `strict` 属性が廃止。末尾のスラッシュはマッチングで無視される
- 正規表現が使えなくなり、現状では末尾の `*` だけがワイルドカードとしてマッチング可能
- 大文字・小文字を区別したい場合は `caseSensitive` 属性を指定する

「……これもけっこう頭の切り替えを必要としますね」
「v4 および v5 は厳密さを優先させたために、書き方は冗長でもロジックは普通のプログラミングと同じだった。v6 ではコードはシンプルになるけど、この独特のロジックを完全に理解しておく必要がある。正規表現が使えなくなったのは痛いけど、将来的にはより高度なパターンマッチングができるようにすると公式が言及してるので、それを待つことにしよう」
「はい」
「次に挙げる大きな変更点は、`History` オブジェクトの廃止。履歴系の操作は Hooks API の `useNavigation` で返される `navigate` 関数に集約された。サンプルで見てみよう」

リスト 20: v5 での History を使った履歴操作

```
import { useHistory, Redirect } from 'react-router-dom';

const Direction: FC<{ shouldGoHome: boolean }> = ({ shouldGoHome }) => {
  const history = useHistory();

  if (shouldGoHome) history.push('/');
```

```
  return (
    <>
      <button onClick={() => history.go(-2)}>2 つ戻る</button>
      <button onClick={history.goBack}>戻る</button>
      <button onClick={history.goForward}>進む</button>
      <button onClick={() => history.go(2)}>2 つ進む</button>
    </>
  );
};
```

「これが v5 での `history` オブジェクトを使った履歴系の操作ね。v6 で書き換えると次のようになる」

リスト 21: v6 での useNavigate を使った履歴操作

```
import { useNavigate, Navigate } from 'react-router-dom';

const Direction: FC<{ shouldGoHome: boolean }> = ({ shouldGoHome }) => {
  const navigate = useNavigate();

  if (shouldGoHome) navigate('/');

  return (
    <>
      <button onClick={() => navigate(-2)}>2 つ戻る</button>
      <button onClick={() => navigate(-1)}>戻る</button>
      <button onClick={() => navigate(1)}>進む</button>
      <button onClick={() => navigate(2)}>2 つ進む</button>
    </>
  );
};
```

「ほんとだ。全部 `navigate` 関数でこと足りてますね」

「`navigate` 関数は引数が数値の場合は『戻る・進む』系の操作、文字列または `PartialPath` 型のオブジェクト（e.g. `{ pathname: '/foo', search: '?bar=baz', hash: '#qux' }`）の場合はリダイレクト系の操作になる。また後者では第 2 引数に `{ replace?: boolean; state?: object | null }` 型のオプションを与えることができる。これによって `history.replace('/')` は `navigate({ path: '/' }, { replace: true })` のように書き換えられる」

「ふむふむ」

「最後に紹介する変更点は `<Redirect>` の廃止。代わりに `<Navigate>` を使うようになった。使い方

は次のようになる」

```
<Navigate to=`/Home` replace />
```

「あ、`<Redirect>` はデフォルトが履歴上書きのリダイレクトで `push` 属性によって変更してたのが、`<Navigate>` では逆になってますね。デフォルトでは履歴を上書きせず、`replace` 属性を与えることで変更するわけですか。

……それにしても疑問なんですけど、どうして従来の `History` や `<Redirect>` からこれらに置き換えられたんでしょうか？」

「これは前にも少しふれたけど、React ではバージョン 16.7 から Suspense という遅延読み込みの機能が導入されてて、それがデータ取得のシーンでも本格的に導入されようとしてるのね。ブラウザの History API を直接使ってる従来の `History` や `<Redirect>` は、その Suspense と相性が悪いのよ。

Suspense に対応させるために、これまでのやり方を変える必要があったわけ。Suspense は近い将来、React でかなり多用されるようになる機能なので、今から対応しておく必要があるの」

「なるほど。ルーティングルールがシンプルに書けるようになるからだけじゃなく、React 本体の新しい機能への対応という理由からも v6 に移行したほうがいいんですね」

「そうだね。ちなみに Suspense については、この研修の一番最後にじっくりと時間をとって教えてあげる予定だから[34]」

v6 でアプリケーションを書き直す

「じゃ、仕上げに React Router v5 で書いてた『SLAM DUNK』のキャラ一覧を v6 に移行させてみよう。まずはトップ階層の `App.tsx` から」

リスト 22: App.tsx

```
import { FC, useEffect } from 'react';
import { Navigate, Route, Routes, useLocation } from 'react-router';
⋮
const App: FC = () => {
  ⋮
```

[34] 「第 13 章 Suspense でデータ取得の宣言的 UI を実現する」

```
  return (
    <div className="container">
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="characters" element={<Characters />}>
          <Route path="/" element={<AllCharacters />} />
          <Route path=":schoolCode" element={<SchoolCharacters />} />
        </Route>
        <Route path="*" element={<Navigate to="/" replace />} />;
      </Routes>
    </div>
  );
};

export default App;
```

「さっそく nested routes を活用してますね。ん？　最後の `<Route>` エントリーで見慣れないことやってますね。これって何やってるんですか？」

「これはね、`<Switch>` がなくなってどのエントリーにもパスがマッチしなかったときのデフォルトの挙動が設定できなくなったことへの私なりの対処法なの。`<Routes>` の中の `<Route>` エントリーのパスは最後まで評価した上でのベストマッチが選択されるということは説明したよね。だから `<Route path="*" ...>` のようにワイルドカードでパスを指定しておけば、よりマッチする他のエントリーがあるときはそちらが優先され、他のどれにもマッチしなかったときのルーティングが記述できるの」

「なるほど。`<Switch>` でどれにもマッチしなかったときにトップへリダイレクトをしてたのと同じことがこれで実現できるわけですか。でもちょっとわかりにくいですね」

「そのうち使いでのいい API が提供されるかもしれないけどね。じゃあ次、`Characters` コンポーネントを見ようか。`/characters/*` パスの親ルートとしてレンダリングされるコンポーネントだね」

リスト 23: components/pages/Characters.tsx

```
import { FC } from 'react';
import { Outlet } from 'react-router-dom';
import { Divider } from 'semantic-ui-react';

import HomeButton from 'containers/molecules/HomeButton';

const Characters: FC = () => (
  <>
```

```
    <header>
      <h1>『SLAM DUNK』登場人物</h1>
    </header>
    <Outlet />
    <Divider hidden />
    <HomeButton />
  </>
);

export default Characters;
```

「あ、`<Outlet>` がありますね」

「うん。ここにルーティングで定義されてた `AllCharacters` コンポーネントか `SchoolCharacters` コンポーネントがレンダリングされるわけだね。次は `SchoolCharacters` コンポーネントの中身を見てみよう」

リスト 24: containers/templates/SchoolCharacters.tsx

```
import { FC } from 'react';
import { Navigate, useLocation, useParams } from 'react-router-dom';
⋮
const EnhancedSchoolCharacters: FC = () => {
  ⋮
  if (schoolCodeList.includes(schoolCode)) {
    const { school, players } = charactersData[schoolCode];

    return (
      <SchoolCharacters  school={school}  characters={players} isLoading={isLoading} />
    );
  }

  return <Navigate to="/" replace />;
};

export default EnhancedSchoolCharacters;
```

「ここは `<Redirect>` が `<Navigate>` に変わっただけですね」

「そうだね。ルーティングルール以外の部分は、基本的に `History` オブジェクトを直に使ってた箇所と `<Redirect>` を使ってた箇所を書き換えるだけだからね。あとは `HomeButton` コンポーネントくらいかな」

リスト 25: containers/molecules/HomeButton.tsx

```
import { FC } from 'react';
import { useNavigate } from 'react-router-dom';
import HomeButton from 'components/molecules/HomeButton';

const EnhancedHomeButton: FC = () => {
  const navigate = useNavigate();

  return <HomeButton redirectToHome={() => navigate('/')} />;
};

export default EnhancedHomeButton;
```

「`history.push('/')` のところが `navigate('/')` に変わったと。あれ？　これで終わりですか。意外と v5 から v6 への移行って簡単なのでは？」

「v3 から v4 のときのようにアプリケーションの設計まで変更する必要はないからね。ただ `History` オブジェクトから `useNavigation` へのスイッチは機械的に書き換えればいいけど、パスのマッチングのところはそうはいかない。v6 独自のロジックをちゃんと理解してないと、思わぬ誤動作を招いてしまうからね」

「たしかにそうですね。で、最後に聞いておきたいんですけど、柴崎さんとしては React Router v6 をどう評価しますか？」

「nested routes によってルーティングルールを一箇所にまとめて書けるようになったのは大きなメリットだね。`<Routes>` による排他的ルーティングが switch-case 的な挙動からベストマッチに変更されたのも最初は面食らったけど、将来的なメンテナンスを考えればこっちのほうがエントリを追加したときに壊れにくくていいかもと今は思うようになった。v4・v5 が厳密性を重視した仕様になってたのが、v6 ではそれよりも開発者が直感的でシンプルなコードを書けるようにすることを優先したつくりになってる。

　現状、パスのマッチングに正規表現が使えなくなったのと、`History` オブジェクトを直接参照できなくなってしまって、『戻る』『進む』などの直近に実行されたアクションの種類をどうやって判定したらいいのかという課題はあるけど、これはそのうち対応されるのを待つしかないかな。Suspense 対応のことを考えても、v6 に移行しないという選択肢はないと思う」

「移行にデメリットはあるけど、それよりもメリットのほうが優(まさ)ってると考えてるわけですね」

「そうだね。コミュニティとしても、v3 から v4 への移行のときよりはスムーズに進むだろうし。ま

あ過去の対応を考えても、まだしばらくは 5 系のサポートは続くだろうけど。私としては新規プロジェクトを始めるなら、安定版ももうすぐ出るということだし v6 を採用したほうがいいと思うよ」

第 11 章　Redux でグローバルな状態を扱う

11-1.　Redux の歴史

Flux アーキテクチャ

「React ではコンポーネントを組み合わせてアプリケーションを作っていくというのはこれまで何度も説明してきたよね。そしてコンポーネントには自身が状態を持たないステートレスなコンポーネントと、自身の状態を持つステートフルなコンポーネントがある。この『状態（state）』についてだけど、実際のアプリケーション開発では個々のコンポーネントにとどまらず、コンポーネントをまたいで保持させたい状態が存在することがしばしばあるの。たとえばユーザーのログイン非ログイン状態やアカウント情報なんかがその最たるものだね」

「はい、私もそれどうやるんだろうと思ってました。これまでの例では、状態はもっぱら個々のコンポーネントの中に閉じ込められてたので。React ではそういうグローバルに持たせたい状態をどうやって管理してるんでしょうか」

「ひとつ考えられる方法としては、上位のコンポーネントに必要な状態をすべて持たせておいて、それらを子孫のコンポーネントに props でバケツリレーしていく、というのがあるね」

「……いやそれ、めちゃくちゃ取り回し大変じゃないですか？」

「あはは、まあそうだね。でも初期は他に方法がなかったから、必要な値を props として子要素、孫要素、曾孫要素……とひたすら受け渡していくということが実際に行われてた。これは『Prop Drilling[35]』と呼ばれて React アプリのコードを複雑にする問題として忌み嫌われてたんだけど、さすがに今はそんなことをしてる人はいないだろうね」

「むー、先人たちは大変だったんですね……」

「公開当初、React は『Just The UI』というコンセプトを掲げてたわけだけど、本体としてはアプリケーション全体の状態管理の手段を提供することはもちろん、どのように行うべきかという方針を示すこともせず、それらは個々の開発者に委ねられていたのね。最初のほうはみんながんばって

[35] 「Prop Drilling」https://kentcdodds.com/blog/prop-drilling

著者紹介

大岡由佳（おおおか・ゆか）

インディーハッカー、技術同人作家。最初は Ruby on Rails、後に React 専門のフリーランスプログラマとして幾多の現場を渡り歩く。その経験を元に執筆した「りあクト！」シリーズが評判を呼びヒット作となる。現在は常駐も受託も行っていないため、事実上おそらく国内（世界でも？）に唯一生息する、専業のプロ技術同人作家。

趣味は漫画読み、ホームシアターでの映画鑑賞、クラシックバレエ。

Twitter アカウントは `@oukayuka`。

黒木めぐみ（くろき・めぐみ）

漫画家、イラストレーター。

「りあクト！」シリーズの表紙イラストを一貫して担当。

メールマガジン登録のご案内

くるみ割り書房 BOOTH ページ

「りあクト！」シリーズを刊行している技術同人サークル「**くるみ割り書房**」は新刊や紙の本の再版予定、また日々の執筆の様子、その他読者の方へのお知らせなどをメルマガとして毎月 2 回ほど配信しています。

購読をご希望の方は、「くるみ割り書房　BOOTH」で検索して最初に表示された上記画像のページにて、「フォロー」ボタンからサークルのフォロー登録をお願いします。BOOTH の一斉送信メッセージにてメルマガの内容をお届けします。

（※ BOOTH のアカウントが必要になります。BOOTH はピクシブ株式会社が運営するオンラインショップです）

りあクト! TypeScript で始めるつらくない React 開発 第 3.1 版【Ⅰ. 言語・環境編】

現場のエンジニアから多大な支持を受ける『りあクト！ TypeScript で始めるつらくない React 開発』の最新 3.1 版、三部作の第一部「言語・環境編」。

フロントエンド開発において JS および TS のスキルは大事な基礎体力。その基礎を固めつつ、実際の React 開発において押さえておきたい仕様やさらに進んだ書き方までを学んでいきます。

フロントエンドが初めての方はもちろん、初〜中級者や前の版をお持ちの方にもオススメです。BOOTH にて絶賛販売中！

（2020 年 12 月 26 日発行／ 225p ／￥ 1,200）

《第一部 目次》

りあクト! TypeScript で始めるつらくない React 開発 第 3.1 版 【Ⅱ. React 基礎編】

現場のエンジニアから多大な支持を受ける『りあクト！ TypeScript で始めるつらくない React 開発』の最新 3.1 版、三部作の第二部「React 基礎編」。

本質を理解するため歴史を深堀りして、なぜ React が今のような形になったかを最初に解き明かします。そのうえで JSX やコンポーネント、Hooks を学ぶため、理解度が格段にちがってくるはず。

初～中級者や前の版をお持ちの方にもオススメです。BOOTH にて絶賛販売中！

（2020 年 12 月 26 日発行／ 202p ／￥ 1,200）

《第二部 目次》

りあクト! TypeScript で極める現場の React 開発

『りあクト！ TypeScript で始めるつらくない React 開発』の続編となる本書では、プロの開発者が実際の業務において必要となる事項にフォーカスを当ててました。

React におけるソフトウェアテストのやり方やスタイルガイドの作り方、その他開発を便利にするライブラリやツールを紹介しています。また公式推奨の「React の流儀」を紹介、きれいな設計・きれいなコードを書くために必要な知識が身につきます。BOOTH にて絶賛販売中！

（2019 年 4 月 14 日発行／ 92p ／￥ 1,000）

《目次》

りあクト! Fiebase で始めるサーバーレス React 開発

個人開発にとどまらず、企業プロダクトへの採用も広まりつつある Firebase を React で扱うための実践的な情報が満載！

Firebase の知識ゼロの状態からコミックス発売情報アプリを完成させるまで、ステップアップで学んでいきます。シードデータ投入、スクレイピング、全文検索、ユーザー認証といった機能を実装していき、実際に使えるアプリが Firebase で作れます。BOOTH にて絶賛販売中！

（2019 年 9 月 22 日発行／ 136p ／￥ 1,500）

《目次》

りあクト! TypeScript で始めるつらくない React 開発 第3.1 版
【Ⅲ. React 応用編】

2020 年 12 月 26 日　初版第 1 刷発行
2020 年 12 月 26 日　電子版バージョン 1.0.0

著者　大岡由佳

印刷・製本　日光企画