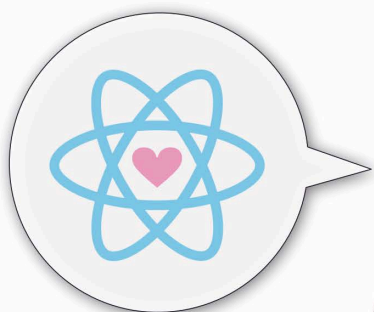


いあぐト!



第3版

Sample

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
をとことんわかりやすく!

シリーズ累計

8千部

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)

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5章 JSX で UI を表現する

第6章 Lint 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

第11章 Redux でグローバルな状態を扱う

第12章 React は非同期処理とどう戦ってきたか

第13章 Suspense でデータ取得の宣言的 UI を実現する

りあクト!

TypeScript で始めるつらくない React 開発

第 3 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 の知識を得、第二部で React の基礎知識と基本的な使い方を身につけました。

第三部では React による副作用処理を大きなテーマとして扱います。なお今版は、前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とみにささやかれた「Redux 不要論」についても、かなりのリソース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。ですので、かなり力を入れて書きました。もし他の章はとばしたとしても、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従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 の業こゝろといきましょうか。公開から 7 年経っ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。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 に用意した以下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。また 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。

<https://github.com/oukayuka/Riakuto-StartingReact-ja3.0>

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 2 年半ほど。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ることになる。

前版との差分および正誤表

初版および第 2 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-ja3.0/blob/master/CHANGELOG.md>
- 『りあクト！TypeScript で始めるつらくない React 開発 第 3 版』正誤表
<https://github.com/oukayuka/Riakuto-StartingReact-ja3.0/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 (<code>react</code>)	16.13.1
• React DOM (<code>react-dom</code>)	16.13.1
• React (<code>react@next</code>)	17.0.0-rc.1
• React DOM (<code>react-dom@next</code>)	17.0.0-rc.1
• React (<code>react@experimental</code>)	0.0.0-experimental-94c0244ba
• React DOM (<code>react-dom@experimental</code>)	0.0.0-experimental-94c0244ba
• Create React App (<code>create-react-app</code>)	3.4.3
• TypeScript (<code>typescript</code>)	4.0.2
• ESLint (<code>eslint</code>)	6.8.0
• Prettier (<code>prettier</code>)	2.1.1
• React Router (<code>react-router</code>)	5.2.0
• React Router (<code>react-router@next</code>)	6.0.0-beta.0
• Redux (<code>redux</code>)	4.0.5
• React Redux (<code>react-redux</code>)	7.2.1
• Redux Toolkit (<code>@reduxjs/toolkit</code>)	1.4.0
• Ky (<code>ky</code>)	0.23.0
• Prettier (<code>prettier</code>)	
• React Query (<code>react-query</code>)	2.15.4

目次

第三部まえがき	3
本書について	5
登場人物	5
前版との差分および正誤表	5
本文中で使用している主なソフトウェアのバージョン	6
第 10 章 React におけるルーティング	13
10-1. SPA におけるルーティングとは	13
10-2. ルーティングライブラリの選定	16
10-3. React Router (5 系) の API	22
React Router のインストールと導入方法	22
React Router のコンポーネント API	24
React Router の Hooks API	29
10-4. React Router をアプリケーションで使う	33
10-5. React Router バージョン 5 から 6 への移行	44
v5 から v6 への変更点	44
v6 でアプリケーションを書き直す	51
第 11 章 Redux でグローバルな状態を扱う	55
11-1. Redux の歴史	55
Flux アーキテクチャ	55
Redux の登場	59
11-2. Redux の使い方	61
Redux の思想	61
Redux をアプリケーションに組み込む	64
11-3. Redux 公式スタイルガイド	75
11-4. Redux Toolkit を使って楽をしよう	84

11-5. Redux と useReducer	91
useReducer で Redux の処理を書き直す	91
useReducer と State Hook の正体	97
第 12 章 React は非同期処理とどう戦ってきたか	101
12-1. 過ぎ去りし Redux ミドルウェアの時代	101
コンポーネントの中で非同期処理を行う	101
Redux ミドルウェア黄金時代の始まり	102
公式謹製 Redux Thunk	105
筋はいいがクセの強い redux-saga	110
12-2. Effect Hook で非同期処理	119
Redux ミドルウェアは問題を解決できたのか	119
公式が示した Effect Hook という道	121
12-3. 「Redux 不要論」を検証する	127
オリジナル作者 Dan Abramov の離脱	127
New Context API の登場	129
Redux 周辺を取り巻くトレンドの変化	134
この先 Redux とどうつきあっていくべきか	137
第 13 章 Suspense でデータ取得の宣言的 UI を実現する	142
13-1. Suspense とは何か	142
Effect Hook で副作用を扱う難しさ	142
Suspense for Code Splitting	144
Suspense を非同期的なデータ取得に使う	148
13-2. “Suspense Ready”なデータ取得ライブラリ	150
Relay、Apollo、urql	150
SWR — Next.js で有名な Vercel 社製ライブラリ	155
React Query — 個人開発ながら多機能でプロジェクトが活発	160
React Query with Suspense	164
13-3. Suspense の優位性と Concurrent モード	174
既存アプローチとの比較から見る Suspense の優位性	174
Concurrent モードとは	181

Concurrent モード、もう使うべきか、まだ待つべきか	182
13-4. Suspense と Concurrent モードが革新する UX	184
キーワードは「UX ファースト」	184
Concurrent モードを有効にする	186
Concurrent モード API の使い方	189
Concurrent モードで先進的 UI を実現する	193
Suspense と Concurrent モードは非同期処理の最終解か	204
エピローグ	207
あとがき	209

第一部「Ⅰ. 言語・環境編」目次

第1章 こんにちは React

- 1-1. 基本環境の構築
- 1-2. プロジェクトを作成する
- 1-3. アプリを管理するためのコマンドやスクリプト

第2章 エッジでディープな JavaScript の世界

- 2-1. あらためて JavaScript ってどんな言語？
- 2-2. 変数の宣言
- 2-3. JavaScript のデータ型
- 2-4. 関数の定義
- 2-5. クラスを表現する
- 2-6. 配列やオブジェクトの便利な構文
- 2-7. 式と演算子で短く書く
- 2-8. JavaScript の鬼門、this を理解する
- 2-9. モジュールを読み込む

第3章 関数型プログラミングでいこう

- 3-1. 関数型プログラミングは何がうれしい？
- 3-2. コレクションの反復処理
- 3-3. JavaScript で本格関数型プログラミング
- 3-4. JavaScript での非同期処理

第4章 TypeScript で型をご安全に

- 4-1. TypeScript はイケイケの人気言語？
- 4-2. TypeScript の基本的な型
- 4-3. 関数とクラスの型
- 4-4. 型の名前と型合成
- 4-5. さらに高度な型表現
- 4-6. 型アサーションと型ガード
- 4-7. モジュールと型定義

4-8. TypeScript の環境設定

第二部「II. React 基礎編」目次

第5章 JSX で UI を表現する

- 5-1. なぜ React は JSX を使うのか
- 5-2. JSX の書き方

第6章 Linter とフォーマッタでコード美人に

- 6-1. ESLint
- 6-2. Prettier
- 6-3. stylelint
- 6-4. さらに進んだ設定

第7章 React をめぐるフロントエンドの歴史

- 7-1. React 登場前夜
- 7-2. Web Components が夢見たもの
- 7-3. React の誕生
- 7-4. React を読み解く 6 つのキーワード
- 7-5. 他のフレームワークとの比較

第8章 何はなくともコンポーネント

- 8-1. コンポーネントのメンタルモデル
- 8-2. コンポーネントと props
- 8-3. クラスコンポーネントで学ぶ state
- 8-4. コンポーネントのライフサイクル
- 8-5. Presentational Component と Container Component

第9章 Hooks、関数コンポーネントの合体強化パーツ

- 9-1. Hooks に至るまでの物語

目次

- 9-2. Hooks で state を扱う
- 9-3. Hooks で副作用を扱う
- 9-4. Hooks におけるメモ化を理解する
- 9-5. Custom Hook でロジックを分離・再利用する

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

「はい。じゃあ秋谷さん、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」

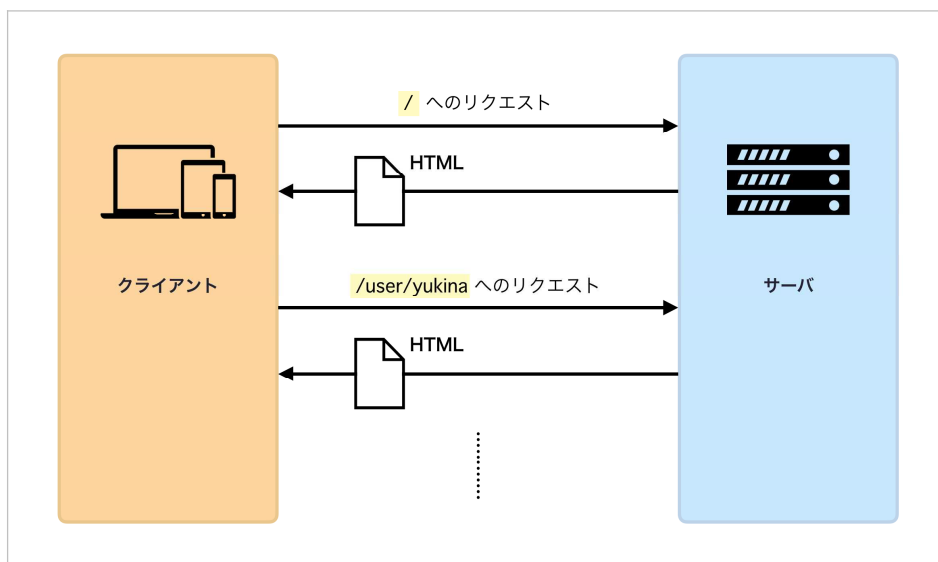


図 1: サーバサイド Web アプリケーションのルーティング

「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 が書き換わるのがあっても、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でないの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」

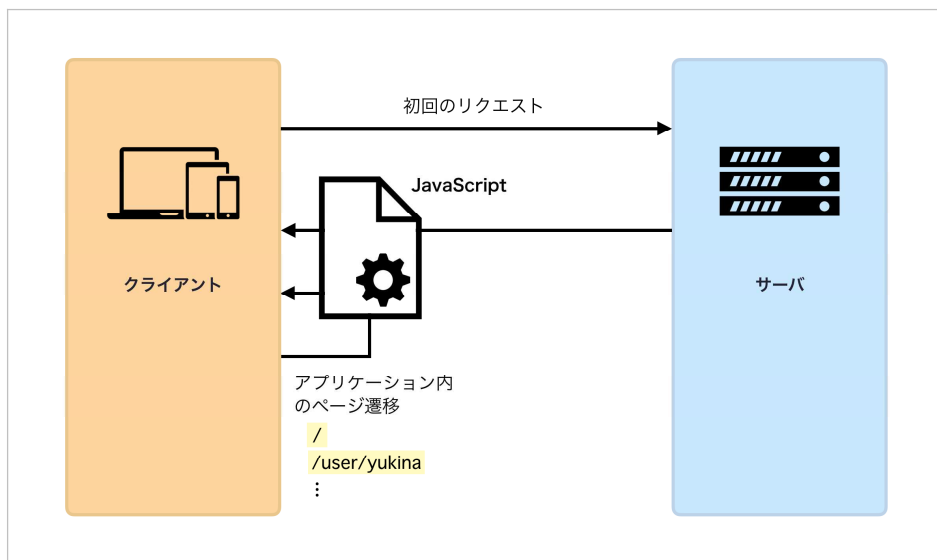


図 2: SPA のルーティング

「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 の書き換えをとま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 API² 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『シバン』と発音する）を使う一種の裏技がいつ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 Router⁴ がデファクトスタンダードとっていいだろうね。二番手は Reach Router⁵。React Navigation⁶ というのがあって React Native の世界ではもっともメジャーなんだけど、Web 対応については 2020 年 9 月現在で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/>

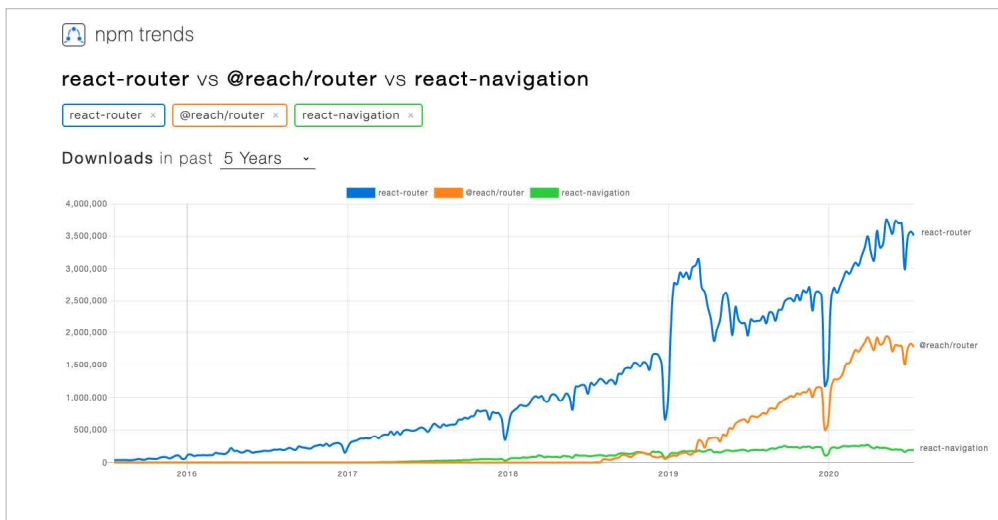


図 3: React 用ルーティングライブラリのトレンド (2020 年 9 月現在)

「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 はシンプルなインターフェースでかつフットプリントも小さいルーティングライブラリだね。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 Contribution 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 ^{たもと} 袂 を分かった上で、新しく Reach Router を作ったってことですか？」

「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。React Router は React Training という

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>

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 Training 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。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ツ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回めの大変更は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いなかった。作者陣は Ember.js のバックグラウンドを持っていたので、Ember 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

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>

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いた」

「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、ルーティングは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でよ。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た。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。

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の。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、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でなく、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、最近では 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ナンスしてるんですね」

「容赦なく下方互換性をバツサリ切り捨ててるいっぽうで、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 9 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何か月かかかるはず。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 系のどちらを選びますか？」

「まあ 6 系かな。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¹¹ 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——」

「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」

¹¹ 「第 13 章 データ取得の次世代標準 Suspense」を参照のこと。

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¹²、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たから、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。余裕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、次のように実行する。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 React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import App from './App';
```

¹² <https://reactrouter.com/web>


```
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いる。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>` 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コンポーネント。一般的な SPA の開発ではこれ一択っていい。その次の `<HashRouter>` 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>` 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>`、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し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 `<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` 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 React, { 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';

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、挙動にちがいはあるんですか？」

「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 9 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¹³。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良いから、説明は省略するね」

「柴崎さん、`render props` 嫌いでもんね……」

「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

¹³ 「`render: func` - React Router」<https://reactrouter.com/web/api/Route/render-func>

- `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>` というコンポーネントが用意されてる。これは `switch-case` 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 React, { 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>。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ひがあるんですか？」

「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 `/nowhere` 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。

じゃ次は、リンク機能を提供する `Link` コンポーネント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>
  <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。そうなんじゃないですか……」

「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¹⁴。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

¹⁴ 「React Router v5.1: React Training Blog」 <https://reacttraining.com/blog/react-router-v5-1/>

を見ていこう」

リスト 6: useHistory の使い方

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

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

  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 History API が提供する生の History オブジェクト¹⁵ 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

¹⁵ 「History - Web API | MDN」 <https://developer.mozilla.org/ja/docs/Web/API/History>

- `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、`useLocation` ね」

リスト 7: `useLocation` の使い方

```
import React, { 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。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。」

「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いる」

```
{
  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ってっとり早いかな」

リスト 8: `useParams` の使い方

```
import React, { 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なめんなめんな記述になるけど、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、たしかにめんどいですね……。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』のキャラクター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」



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クター一覧、高校別キャラクター一覧の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います。この `molecules/` とか `organisms/` って何ですか？」

「これは『Atomic Design¹⁶』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が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ではなくて、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;
```

¹⁶ 「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>
  );
};

```

「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 React, { FC } from 'react';
import { Redirect, Route, Switch, RouteComponentProps } from 'react-router';
import { Divider } from 'semantic-ui-react';

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ヘ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 React, { FC } from 'react';
import { Redirect, useLocation, useParams } from 'react-router-dom';
import { parse } from 'query-string';

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

const EnhancedSchoolCharacters: FC = () => {
  const { schoolCode } = useParams<{ schoolCode: string }>();
  const { search } = useLocation();
  const isLoading = !!parse(search)?.loading;
  const schoolCodeList = Object.keys(charactersData);

  if (schoolCodeList.includes(schoolCode)) {
    const { school, players } = charactersData[schoolCode]; // ★データを抽出

    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 に入ってる。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。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、キャラクター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` を使ってるのは URL クエリパラメータを抽出してるのね。`query-string`¹⁷ は URL クエリ文字列をパースしてオブジェクトにしてくれるライブラリね。その `parse` 関数によってクエリパラメータから `loading` に設定されてる値をさらに抽出してるわけ」

「`?.` は optional chaining¹⁸ で、`loading` パラメータがなかったときは `undefined` を返してるんですよ。その前の `!!` というのは？」

「『二重否定 (Double Negation)』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 React, { 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 = {
  school: string;
```

¹⁷ <https://github.com/sindresorhus/query-string>

¹⁸ 第2章、第7節内の「Nullish Coalescing と Optional Chaining」を参照のこと。

```

  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 Helmet¹⁹ 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 React, { 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 }) => (
  <>

```

¹⁹ <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。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いるのちがって、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^{のつと}則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、前の書き方では `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 React, { 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いければ、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 において 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 9 月現在、6 系の最新版は 6 月にリリースされた 6.0.0 の β.0 版だけど、従来ならそこから 3～6 か月で正式版が出るはず。まだ β 版なので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²⁰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」

²⁰ 「[Migrating React Router v5 to v6](#)」

<https://github.com/ReactTraining/react-router/blob/dev/docs/advanced-guides/migrating-5-to-6.md>

「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んで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」

「過去をバツ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。β 版である現状、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-router-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> の主な違いは次のとおり」

- <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
 - 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)

- `<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てメンタルモデルを切り替える必要がある。まず 1 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 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>  
  </div>  
)
```



```

    <Switch>
      <Route path={` ${match.path}/me`} ><OwnUserProfile /></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={<OwnUserProfile />} />
    </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={<OwnUserProfile />} />
      <Route path=":id" element={<UserProfile />} />
    </Route>
  </Routes>
);

```

```
    </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 `OwnUserProfile` 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 `OwnUserProfile` がレンダリングされる。`<Outlet>` は子ルートの子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る」

「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、親ルートと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 戻る</button>
      <button onClick={() => navigate(-1)}>戻る</button>
      <button onClick={() => navigate(1)}>進む</button>
      <button onClick={() => navigate(2)}>2 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` 型のオブジェクトの場合はリダイレクト系の操作になる。後者は第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('/', { replace: true })` と書き換えられる」

「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？」

「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 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²¹⁾」

v6 でアプリケーションを書き直す

「じゃ、仕上げて React Router v5 で書いてた『SLAM DUNK』のキャラクター一覧を 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 React, { FC, useEffect } from 'react';
import { Navigate, Route, Routes, useLocation } from 'react-router';
:
const App: FC = () => {
  :
  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>
  );
};
```

²¹⁾ 「第 13 章 Suspense でデータ取得の宣言的 UI を実現する」

```
};  
  
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ヘリダイレクトをし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」

「そのうち使い出しのいい 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 React, { 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 React, { FC } from 'react';
import { Navigate, useLocation, useParams } from 'react-router-dom';
import {
  schoolCodeList,
  charactersData,
  isLoading,
} from '../data';

const EnhancedSchoolCharacters: FC = () => {
  const { schoolCode } = useParams();

  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 React, { 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悪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^{まさ}優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、まだ β 版でも見切りで 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、それらの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』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さっき挙げたような props のバケツリレーをしたり、他のフレームワークを組み合わせを使ってたんだけど、React の急速な普及により大規模なアプリケーション開発に用いられることが増え、その性質に合った新しい設計指針が求められるようになってきた」

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。React をこよなく愛し、フリーランスプログラマとして幾多の現場を渡り歩く。その経験を元に同人誌を書いたところ、かなりの評判を呼びヒットとなる。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、専門のプロ技術同人作家。

本作の執筆にかかりっきりで、せっかく買った 4K プロジェクターと Apple TV が全然稼働できていなかったの、校了したら自宅で映画三昧になる予定。

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9月12日発行／218p／¥1,200)

《第一部 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

りあクト! TypeScript で始めるつらくない React 開発 第3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掘りして、なぜ React が今のよう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9月12日発行／198p／¥1,200)

《第二部 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

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》

- 第 1 章 デバッグをもっとかんたんに
- 第 2 章 コンポーネントのスタイル戦略
- 第 3 章 スタイルガイドを作る
- 第 4 章 ユニットテストを書く
- 第 5 章 E2E テストを自動化する
- 第 6 章 プロフェッショナル React の流儀

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》

- 第 1 章 プロジェクトの作成と環境構築
- 第 2 章 Seed データ投入スクリプトを作る
- 第 3 章 Cloud Functions でバックエンド処理
- 第 4 章 Firestore を本気で使いこなす
- 第 5 章 React でフロントエンドを構築する
- 第 6 章 Firebase Authentication によるユーザー認証

りあクト! TypeScript で始めるつらくない React 開発 第3版 【III. React 応用編】

2020 年 9 月 12 日 初版第 1 刷発行
2020 年 10 月 1 日 電子版バージョン 1.0.4

著者 大岡由佳

印刷・製本 日光企画
