

# 初心者向け！

## TypeScriptで始める つらくない React開発

大岡由佳

② React基礎編

最新ツールVite、React 18に対応

思想・歴史からReactを学ぶ  
JSXやHooksが納得して身につく

シリーズ累計

2.5万部  
突破！

読者  
からの声

「説明がめちゃくちゃ丁寧！」  
「この本を最初に読みたかった……」

BOOTH  
技術書 前版  
1位

フロントエンド未経験で飛び込んだ先で、リーダーの柴崎に短期集中研修を受ける秋谷。最初の厳しい言葉とは裏腹に丁寧な彼女のマンツー指導により、関数型プログラミングと TypeScript をなんとか使えるレベルに習得できた。そしてついに React の学習が始まるが、サーバサイドでの考え方からなかなか頭が切り替えられない。そんな秋谷に、柴崎は React の思想と歴史を語り始める。

### 本作の構成 (全3巻)

第1章 こんにちは React

第2章 ライトでディープな 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 進化したビルドツールを活用する

第7章 リンターとフォーマッタでコード美人に

第8章 React をめぐるフロントエンドの歴史

第9章 コンポーネントの基本を学ぶ

第10章 コンポーネント操作の高度な技法

第11章 React アプリケーションの開発手法

第12章 React のページをルーティングする

第13章 Redux でグローバルな状態を扱う

第14章 ポスト Redux 時代の状態管理

第15章 React 18 の新機能を使いこなす

りあクト! TypeScript で始める  
つらくない React 開発 第4版

【② React 基礎編】

# 本書について

## 登場人物の紹介

### 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テックリードのフロントエンドエンジニア。React歴は6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、今の会社に転職してきた。チームメンバーを集めるため人材採用にも関わり自ら面接も行っていたが、彼女の要求水準の高さもあってなかなか採用に至らない。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、社内公募が行われた。

### 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

過去の版からの変更点と、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ています。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していきます。

- 各版における内容の変更

<https://github.com/klemiitary/Riakuto-StartingReact-ja4.0/blob/main/CHANGELOG.md>

- 『りあクト！ TypeScriptで始めるつらくないReact開発 第4版』正誤表

<https://github.com/klemiitary/Riakuto-StartingReact-ja4.0/blob/main/errata.md>

## 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おります。

<https://github.com/klemiitary/Riakuto-StartingReact-ja4.0>

挙動をその場で確認していただくため StackBlitz に置いているサンプルもありますが、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。

なおリポジトリのコードと本文に記載されているコードには、ときに差分が存在します。紙面に適切になるよう改行位置を調整したり、各種コメントアウト文を省略するなどによるものですが、そのため行番号が一致しないことがありますので、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、読者自身のプログラムやドキュメントに流用してかまいません。それにあたり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とうございます。

## 本書内で使用している主なソフトウェアのバージョン

• React ( react )	.....	18.2.0
• React DOM ( react-dom )	.....	18.2.0
• Vite ( vite )	.....	3.0.9
• TypeScript ( typescript )	.....	4.8.2
• ESLint ( eslint )	.....	8.0.1
• Prettier ( prettier )	.....	2.7.1

# 目次

本書について .....	2
登場人物の紹介 .....	2
前版との差分および正誤表 .....	2
サンプルコードについて .....	3
本書内で使用している主なソフトウェアのバージョン .....	3
第 5 章 JSX で UI を表現する .....	9
5-1. なぜ React では JSX を使うのか .....	9
5-1-1. JSX はどのように変換されるか .....	9
5-1-2. なぜ React ではデザインとロジックを混在させるのか .....	13
5-1-3. なぜ React はテンプレートを使わないのか .....	16
5-1-4. JSX は汎用的に UI を表現する .....	21
5-2. JSX 構文の書き方 .....	24
5-2-1. JSX の基本的な文法 .....	24
5-2-2. JSX とコンポーネントの関係 .....	29
5-2-3. React の組み込みコンポーネント .....	35
第 6 章 進化したビルドツールを活用する .....	43
6-1. コンパイラとモジュールバンドラ .....	43
6-1-1. フロントエンド開発に使われるコンパイラ .....	43
6-1-2. モジュールバンドラの登場 .....	44
6-1-3. 新しい波を生んだ次世代モジュールバンドラ .....	47
6-2. Vite Killied the Other Star .....	52
6-2-1. ノーバンドルなビルドツール Snowpack .....	52
6-2-2. Create React App はもう古い？ .....	55
6-2-4. 絶賛ブレイク中の Vite .....	57
6-3. Vite を本格的に使いこなす .....	60
6-4. Deno が JavaScript の世界を塗り替える？ .....	68

6-4-1. Deno とは何か	68
6-4-2. Deno の特徴	70
6-4-3. Node.js との互換性	73
6-4-4. Deno はもう実用に耐えうるか	75
<b>第 7 章 リンターとフォーマッタでコード美人に</b>	<b>79</b>
7-1. リンターでコードの書き方を矯正する	79
7-1-1. JavaScript、TypeScript におけるリンターの歴史	79
7-1-2. ESLint の環境を作る	84
7-1-3. ESLint の環境をカスタマイズする	90
7-2. フォーマッタでコードを一律に整形する	102
7-2-1. コードフォーマッタの地位を確立した Prettier	102
7-2-2. Prettier の環境を作る	105
7-3. スタイルシートもリンティングする	109
7-4. さらに進んだ設定	113
<b>第 8 章 React をめぐるフロントエンドの歴史</b>	<b>119</b>
8-1. React の登場に至る物語	119
8-1-1. すべては Google マップショックから始まった	119
8-1-2. フロントエンド第 2 世代技術の興隆	121
8-1-3. Web Components が夢見たもの	123
8-1-4. そして React が誕生する	126
8-2. React を読み解く 6 つのキーワード	127
8-2-1. 公式サイトトップに掲げられている三大コンセプト	127
8-2-2. Declarative（宣言的）	130
8-2-3. Component-Based（コンポーネントベース）、Just The UI（提供するのは UI だけ）	131
8-2-4. Virtual DOM（仮想 DOM）	134
8-2-5. One-Way Dataflow（単方向データフロー）	139
8-2-6. Learn Once, Write Anywhere（ひとたび習得すれば、あらゆるプラットフォームで開発できる）	141
8-3. 他のフレームワークとの比較	143
8-3-1. 第 3 世代のフレームワーク情勢	143

8-3-2. Lit .....	145
8-3-3. Svelte .....	147
8-3-4. Preact .....	152
8-3-5. Solid .....	156
<b>第9章 コンポーネントの基本を学ぶ .....</b>	<b>160</b>
9-1. コンポーネントのメンタルモデル .....	160
9-2. コンポーネントに <b>Props</b> を受け渡す .....	162
9-3. コンポーネントに <b>State</b> を持たせる .....	170
9-4. コンポーネントと副作用 .....	175
9-5. クラスでコンポーネントを表現する .....	181

# 「① 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. Vite でプロジェクトを作成する
- 1-3. プロジェクトを管理するためのコマンドやスクリプト

## 第2章 ライトでディープな 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. 非同期処理と例外処理

## 第4章 TypeScript で型をご安全に

- 4-1. ナウなヤングに人気の TypeScript
- 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

## 目次

### 「③ React 応用編」 目次

#### 第 10 章 コンポーネント操作の高度な技法

- 10-1. ロジックを分離、再利用する技術
- 10-2. フォームをハンドリングする
- 10-3. コンポーネントのレンダリングを最適化する

#### 第 11 章 React アプリケーションの開発手法

- 11-1. React アプリケーションのデバッグ
- 11-2. コンポーネントの設計
- 11-3. プロジェクトのディレクトリ構成

#### 第 12 章 React のページをルーティングする

- 12-1. ルーティングについて知ろう
- 12-2. React Router の基本的な使い方
- 12-3. React Router をアプリケーションで使う

#### 第 13 章 Redux でグローバルな状態を扱う

- 13-1. Redux をめぐる状態管理の歴史
- 13-2. Redux の使い方
- 13-3. Redux Toolkit を使って楽をしよう
- 13-4. useReducer はローカルに使える Redux

#### 第 14 章 ポスト Redux 時代の状態管理

- 14-1. Context API の登場
- 14-2. React で非同期処理とどう戦うか
- 14-3. Redux オルタナティブな状態管理ライブラリ
- 14-4. Redux はもう不要なのか？

#### 第 15 章 React 18 の新機能を使いこなす

- 15-1. React 18 はなぜ特別なバージョンなのか
- 15-2. React 18 の新機能を有効にする
- 15-3. Concurrent Rendering の具体的なメリット
- 15-4. Concurrent Rendering で UI の質を高める

# 第5章 JSXでUIを表現する

## 5-1. なぜReactではJSXを使うのか

### 5-1-1. JSXはどのように変換されるか

「モダンフロントエンド開発に必要なJavaScriptとTypeScriptの知識をひととおり得られたので、今度はJSXについて学んでいこうか」

「JSXってReactに標準搭載されてるテンプレート言語ですよね。Railsに対するERB<sup>1</sup>のような」

「……うーん、ちょっと誤解があるようだね。じゃあまずその誤解を解消するためにも、JSXとは実際のところ何なのかを説明しておこう。JSXという名前の由来は『JavaScript Syntax Extension』とも『JavaScript XML』ともいわれる。つまりJSXはJavaScriptの中にXMLライクな記述ができるようにした構文拡張なの」

「その『構文拡張』というのは？」

「言語標準には含まれない特殊な用途のための便利な構文を、後づけで使えるようにしたもののこと。実際にはJSXは標準のJavaScriptにコンパイルされることを前提としている。たとえばこのJSXコードは、現在のReactのデフォルト設定では次のようにコンパイルされる」

リスト1: コンパイル前のJSXコード

```
<button type="submit" autoFocus>
  Click Here!
</button>
```

リスト2: コンパイル後のJavaScriptコード

```
import { jsx as _jsx } from 'react/jsx-runtime';

_jsx('button', {
  type: 'submit',
  autoFocus: true,
```

<sup>1</sup> 「Embedded Ruby」の略で、「eRuby」とも。HTMLへRubyスクリプトを埋め込むためのテンプレートシステム。Ruby 1.8以降に標準ライブラリとして組み込まれており、Ruby on Railsのviewを記述するための開発言語として標準採用されている。

## 第5章 JSXでUIを表現する

```
    children: 'Click Here',  
});
```

「ふむふむ、`jsx()` という関数に要素名や各種属性を渡す形になってますね」

「この変換を実際に行ってるのは Babel や tsc などのコンパイラなんだけど、実はコンパイル結果はオプション設定によって複数の様式に分かれる。TypeScript の場合、コンパイラオプション `jsx` がその設定ね。設定値によって出力は次のようになる<sup>2</sup>」

設定値	入力	出力	出力ファイルの拡張子
<code>preserve</code>	<code>&lt;div /&gt;</code>	<code>&lt;div /&gt;</code>	<code>.jsx</code>
<code>react</code>	<code>&lt;div /&gt;</code>	<code>React.createElement("div");</code>	<code>.js</code>
<code>react-native</code>	<code>&lt;div /&gt;</code>	<code>&lt;div /&gt;</code>	<code>.js</code>
<code>react-jsx</code>	<code>&lt;div /&gt;</code>	<code>_jsx("div", {}, void 0);</code>	<code>.js</code>
<code>react-jsxdev</code>	<code>&lt;div /&gt;</code>	<code>_jsx("div", {}, void 0, false, {...}, this);</code>	<code>.js</code>

「こんなに種類があるんですか。さっきのコンパイル結果はこのオプション値が `react-jsx` と設定されていた場合のものなんですね」

「そのとおり。なおこの設定は 2020 年 10 月にリリースされた React 17.0 で導入された比較的新しい変換形式で<sup>3</sup>、TypeScript もバージョン 4.1 でそれに対応したという経緯がある。それ以前はこの設定値のデフォルトは `react` で、コンパイル結果はこうなっていた」

リスト 3: コンパイル後の JavaScript コード (`jsx: "react"`)

```
React.createElement(  
  'button',  
  { type: 'submit', autoFocus: true },  
  'Click Here!'  
);
```

「さっきの結果と比べてほしいんだけど、こちらの結果ではインポートしてないはずの `React` オブジェクトがいきなり使われてる。だからこの設定下では、JSX 構文を使うときはファイルの先頭で明示的に `import React from 'react'` を書いてあげないとコンパイルできない」

<sup>2</sup> 「TypeScript: Documentation - JSX」

<https://www.typescriptlang.org/docs/handbook/jsx.html>

<sup>3</sup> 「新しい JSX トランスフォーム -React Blog」

<https://ja.reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html>

「不便だったんですねえ。ちなみに React が JSX の変換形式を刷新した理由って何だったんですか？」  
 「React 開発チームの理想としては、JSX の変換をコンパイラの中で完結させて React に依存しないよう  
 にしたいらしいの<sup>4</sup>。でもそれは現状では難しく妥協するしかない。新しい変換形式は明示的に React  
 をインポートする必要がなく開発者がより意識せず JSX を書ける分、妥協が少ないと言えるだろうね。  
 また内部的にも JSX 部分の実装を `react/jsx-runtime` に切り分けられるようになった。さらに新形式の  
 ほうが処理を最適化しやすくパフォーマンスも多少は向上するみたい」

「なるほど」

「ただ JSX の本質を理解するには従来の変換形式のほうが都合いいので、これ以降の説明は従来形式を  
 使うことにするね。その場合 JSX 構文は基本的に、XML のノードツリーを `React.createElement` という  
 メソッドのコールに変換してる。そしてその結果、次のようなオブジェクトが生成されるの」

```
{
  type: 'button',
  props: {
    type: 'submit',
    autoFocus: true,
    children: 'Click Here',
  },
  key: null,
};
```

「これは TypeScript では `ReactElement` というインターフェースを下敷きにしたオブジェクトになる。つまり React において JSX の構文は、本質的には `React.createElement(...)` のシンタックスシュガーであり、`ReactElement` オブジェクトを生成する特殊な式だと言える」

「シンタックスシュガー……、式……」

「秋谷さんが最初に口にした『JSX はテンプレート言語』という表現がまちがってるのがそこだね。JSX は最終的に JavaScript でオブジェクトを生成する式に還元されるものであって、JavaScript の枠組みを超えた特別なシステムじゃない。だからこそ変数に代入したり、狭義のオブジェクトのプロパティ値にしたり、関数の引数や戻り値にしたりもできるわけ。こんなふうにね」

```
const foo = <span>Hello!</span>;
const obj = { bar: foo };
```

---

<sup>4</sup> 「rfcs/0000-create-element-changes.md at createElement-rfc • reactjs/rfcs」  
<https://github.com/reactjs/rfcs/blob/createElement-rfc/text/0000-create-element-changes.md>

## 第5章 JSXでUIを表現する

```
const fn = (elem: React.ReactElement) => <div>{elem}</div>;
fn(foo);
```

「……あー、なんかわかってきたかも。でもこの書き方、ちょっと気持ち悪いですね。どうしても JSX のとこをクォーテーションで囲いたくなっちゃう」

「その『気持ち悪い』というのは最初に JSX を見た人が抱きがちな感覚なんだけど、本質がわかっていない後は単なる慣れの問題だよ。それに JSX は React 固有のものだと思われがちだけど、React から独立した仕様として公開されていて<sup>5</sup>、他のフレームワークでも使えるようになってる。Vue.js なんか公式ドキュメントに JSX を使うための説明をわざわざ入れてるしね<sup>6</sup>」

「へー、Vue でも JSX が使えるんですね。知らなかった」

「実際の React での開発では JSX はコンポーネントの UI 部分を記述するために使われる。関数コンポーネントであればその関数自身の戻り値として、クラスコンポーネントであれば `render()` メソッドの戻り値として JSX が返されるようにする。雰囲気を知ってもらうために、JSX を使ったコンポーネントの記述をチラ見せするとこんな感じのコードになるかな」

```
const MembersList: React.FC<{ orgId: string }> = ({ orgId }) => {
  const [users, error, isLoading] = useGetMembers(orgId);

  if (isLoading) {
    return <Loader />;
  } else if (error) {
    console.error(error);
  }

  return <div class="error">failed to get users' data</div>;
}

return (
  <CardGroup>
  {users.map((user) => (
    <Card
      header={user.fullName}
      cover={user.profileImage}
      meta={user.birthday?.toLocaleDateString('ja-JP')}
      description={user.bio}
```

<sup>5</sup> <https://facebook.github.io/jsx/>

<sup>6</sup> 「Render Functions & JSX | Vue.js」 <https://vuejs.org/guide/extras/render-function.html>

```

    />
  )}
</CardGroup>
);
};

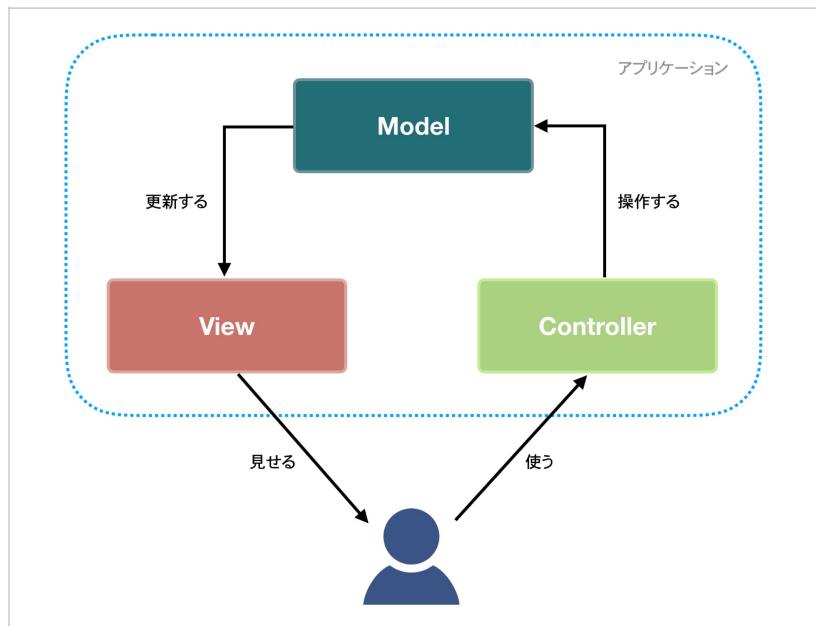
```

「むむむむ……。『Hello, World!』をやったときは単純な処理だったから気がつかなかつたんですが、Reactってデザインとロジックのコードがごちゃまぜになってますよね？ Rails では Model と Controller にロジックを書き、View のテンプレートで見た目を書くというふうにきっちり別れてたので、そこにすごく違和感があります」

「サーバサイドアプリケーションの開発から入ってきた人は、たいていそんなふうに言うね。じゃあなぜ React ではデザインとロジックの記述が混在する形になっているのか、次はそれを説明しようか」

### 5-1-2. なぜ React ではデザインとロジックを混在させるのか

「Ruby on Rails に代表されるサーバサイド Web アプリケーションフレームワークは、そのほとんどがアーキテクチャに MVC のパターンを採用してる」



「MVC とはアプリケーションを Model、View、Controller の 3 つの要素に分割して構築する手法。Web アプリケーションにおいては、ひとつの URL リクエストに対してそれを受けた Controller が起点となり、Model を操作して必要なデータの取得・加工を行って、最後に View でそのデータをテンプレートへ埋め込んでページを出力するというフローになる。視覚的に表すと図 1 のようになるかな」

「はいはい、Rails エンジニアだった私にはなじみのある図ですね」

「ところでソフトウェア工学において、アーキテクチャとかデザインパターンというものは**関心の分離**を行うためにあるものなのね。MVC ではどんな関心によって要素を区分しているかというと、アプリケーション全体で**技術的な役割**によってその 3 つに区分してる」

「ふーむ、まあそのとおりですね」

「この技術的な役割によって関心を分離するというのは一時代を築いた考え方だけど、だからといってそれが普遍的な真理というわけじゃない。特にフロントエンド開発ではその歴史の中で、MVC パターンはうまく機能しないという見方が開発者の間で主流になってるの」

「ええっ？ そうなんですか？」

「Reactにおいては、関心の分離の単位が MVC のように技術的な役割じゃない。アプリケーションにおけるひとつひとつの**機能**こそがその単位になってる。それぞれの機能単位で分割された独立性の高いパートを組み合わせることでアプリケーションを構築しようというのが React の開発思想であり、そのパートに相当する概念こそが**コンポーネント**というわけ。独立した機能単位のパートとして分割するためには、そのパートの中に UI デザインとロジックを閉じ込める必要がある。そうじゃないとパート同士が疎結合にならないでしょ」

「むむむむ……」

「たぶん Rails しか知らない秋谷さんには、React におけるコンポーネントの独立性というのがピンときてないんだと思う。Rails だとそのページの表示に必要な処理を Controller が全部終わらせてから、View でページ全体をまとめてレンダリングする。でも React ではページを構成する部品であるコンポーネント自身が、必要な処理を自律的に行い、レンダリングも非同期的に並列で行われるの」

「……非同期、……並列。そういうえば Twitter とかは、ページ全体のレイアウトはすでに表示されてるのに、部分的にデータの取得待ちでローダーがぐるぐる回ったりしますよね。あれがそうですか？」

「そう、それね。Web 版の Twitter はまさに React 製だから。各コンポーネントが自律的に Twitter API と通信してデータを取得し、個別にレンダリングするからああいう挙動になるの。概念図にするとこんな感じかな」

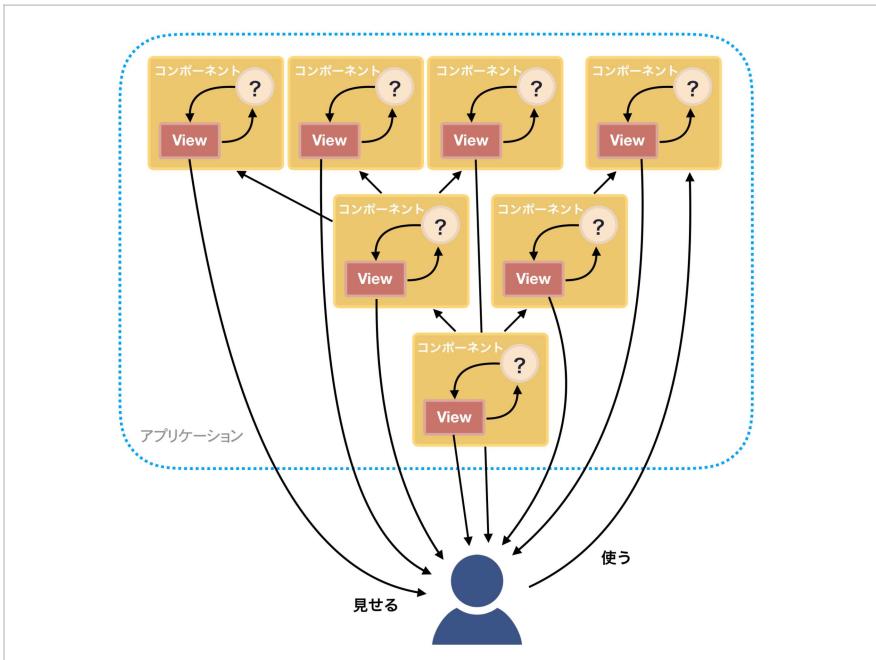


図 2: React のコンポーネントベースなアーキテクチャの概念図

「そしてコンポーネントのロジック、たとえば表示のためのデータをどうやって取得するかとか、起こったイベントをどう処理するかとか、状態の変化が自身の表示にどう影響するかとかなどは、そのコンポーネントのレンダリングと本質的に結合したものであって、それを無理に分割してしまうことのほうが無理があると React では考えるのね。これはまあ、実際に開発してみると実感できないことかもしれないけど」

「うーん、でも開発者はそれでいいかもしれませんけど、じゃあデザイナーとの分業はどうなるんですか？」

Rails による開発ではデザイナーさんに ERB を書いてもらってたんですけど、このロジックが混在した JSX のコードをデザイナーさんが書くのは無理じゃないでしょうか？」

「それは問題の立て方がそもそもおかしいんだよ。デスクトップやモバイルアプリの開発で同じことをデザイナーにしてもらう？ デザイナーが作ってくれるのはサイズやレイアウトが確認できる Figma や Sketch のデザインカンプまででしょ。アプリの開発者はそれを参考にしながら自分で UI を組み上げていく」

「ええっ、それはそうですけど。……うーん」

「実際、モダンなフロントエンド主体のアプリケーション開発の現場ではデザイナーが HTML や CSS のコーディングまで行なうことは稀だね。モバイルアプリのデザイナーと同じく、純粹に UI や UX デザイ

ンを専門にしていて HTML や CSS が書けない人も多い」

「じゃあ、HTML と CSS のコーディングも私たちがやるんですか？」

「そうだよ、それもフロントエンドエンジニアの仕事。まあ完全ではないけど Figma や Sketch から HTML や CSS を書き出すこともできるし、CSS をコンポーネントの名前空間に閉じる技術が普及してて管理もシンプルなので、昔ほど大変じゃないよ」

「ううっ、そうはいってもこれまで自分でまともに CSS を書いたことなかったので……。フロントエンドエンジニアへの道は険しいなあ」

「私たちフロントエンドエンジニアが作るのは、ひとつの URL リクエストに対してひとつの静的な HTML ページを返すだけの単純なアプリじゃないからね。複数の外部 API との並列的な非同期通信、取得データのキャッシュやローカルストレージへの永続化、ユーザーの操作によって即座に変化する UI の状態管理、ときにはカメラや GPS といったデバイスへのアクセスまでをも備えた、インタラクティブでレスポンス性の高いアプリなんだよ。比べるべくはモバイルアプリやデスクトップアプリであって、サーバサイド Web アプリケーションの延長で考えるべきじゃないわけ」

「……そうですか。フロントエンドエンジニアになると決めた以上、意識を切り替えないといけないんですね」

「そうだね。同じ『Web アプリケーション』といっても、React が見据えているものは従来のサーバサイドアプリケーションの開発者が想定しているようなそれとは微妙にちがいがある。次はその話をしようか」

### 5-1-3. なぜ React はテンプレートを使わないのか

「React のアーキテクチャがコンポーネントをベースにしてるという話をしたけど、コンポーネント指向の React の登場がそれまでのフロントエンド開発の問題をあまりにも鮮やかに解決するために、現在のほとんどのフロントエンドアプリケーション用のフレームワークはその影響を受けてコンポーネントベースになってしまった。ただその中には、アプリケーションをコンポーネント単位に分割しながらもその View のレンダリングに HTML テンプレートを用いてるフレームワークも少なくない」

「そういえば Vue はテンプレートを使いますもんね」

「そう。それらは Angular のようにテンプレートを専用ファイルとして分離するか、Vue.js の SFC (Single File Components、單一ファイルコンポーネント) のように特別なタグでくくって記述するかのちがいはあるけど、フレームワークレベルで HTML の記述を特別扱いして通常の JavaScript から隔離し、その中に `*ngIf` や `v-if` といったフレームワーク独自のテンプレート構文を埋め込むという形は共通してる」

「ふむふむ」

「それとは対象的に React では一貫して View のレンダリングも JavaScript の枠組みの中で行う。テンプレートのように見える JSX も、一皮むけば実際にはオブジェクトを生成するための JavaScript の純粋な式であって、フレームワークから特別扱いされることはない。この思想を仮に『JS ファースト』と呼ぶことにしよう」

「JS ファーストか……。かっこいいですね」

「なぜ React が JS ファーストを採用しているのかは、React チームの初期メンバーで Instagram の React 化を推し進めた中心的人物である Pete Hunt による 2013 年の JSConf EU でのセッション『React: Rethinking Best Practices』<sup>7</sup> で語られてる。時間のあるときに見ておくことをおすすめするけど、JS ファーストを採用した理由については次の 2 つの言葉に集約されるかな」

- Templates separate technologies, not concerns.  
(テンプレートは技術を分離するのであって、関心を分離しない)
- Display logic and markup are <sup>いやおう</sup>inevitably tightly coupled.  
(表示のロジックとマークアップは否応なく密接に結びついてる)

「ほー、なるほど。React 開発の初期中心メンバーの言葉だと思うと説得力がありますね」

「このセッションの内容を踏まえて、ここからは私なりの解釈を話していくね。<sup>あまた</sup>数多あるフロントエンド Web アプリケーションフレームワークは、コンポーネントの View レンダリングの方式によってこの『HTML テンプレート派』と『JS ファースト派』の二大派閥に分けられるのね。それぞれの陣営の内容はこんな感じ」

#### • HTML テンプレート派

Angular、Vue.js、Ember.js、Lit、Svelte など

#### • JS ファースト派

React、Preact、Stencil、Solid など

「まさに二分されてますね。それにしても、なぜフロントエンドのフレームワークはこの二大派閥に分かれるんでしょうか？」

「それは作者が Web アプリケーションをどうとらえるかの世界観のちがいだと思う。HTML テンプレート派は Web アプリケーションのことを『動的な Web ページ』だと考える。だからこそ最終出力の HTML の形式に固執する。いっぽうの JS ファースト派は Web アプリケーションもデスクトップやモバ

<sup>7</sup> [Pete Hunt: React: Rethinking best practices – JSConf EU]

<https://www.youtube.com/watch?v=x7cQ3mrcKaY>

イル向けと同じ普通のアプリケーションであり、ただブラウザがプラットフォームになっているだけだと考える。だから他のプラットフォームのアプリケーションと同じように、一貫した言語で開発しようとする」

「ああ、なるほど！ それすごく腑に落ちますね」

「その世界観のちがいが、組版指定と制御構造のどちらが主でどちらを従とするかという実装のちがいとして表象化してるわけだね。Web アプリケーションを動的な Web ページと考えるなら、UI を表現するのに組版指定が主で制御構造は従となる。必然的に HTML に独自の制御構文を埋め込む形になる。

この形式はシンプルなアプリケーションならワークするけど、制御構造が主体にならざるをえない複雑なアプリケーションに適用しようとすると、必然的に色々なところに無理が出てくる。テンプレートは一見とっつきやすいけど、フロントエンドのフレームワークは従来型のテンプレート形式に固執すればするほど、それ以外の DX (開発者体験) が顕著に低下していく。逆説的だけどこれは呪いのようなもので避けることができない」

「HTML テンプレートの呪いですか……。具体的にはどういうことでしょうか？」

「まずフレームワーク独自の制御構文や各種バインディングなどの暗黙の文脈といったものを大量に用意せざるをえなくなる。一見 HTML がそのまま書けるので最初のチュートリアルは簡単に感じられるけど、本格的なアプリケーションを開発しようとすると次から次へと新しい決まりごとが出てきて、それらをおぼえる必要に迫られることになってしまう。Vue.js は『Progressive Framework (漸進的なフレームワーク)』を標榜してるけど苦しい表現じゃないかな」

「たしかに Vue も Angular も公式ドキュメントのページ数がすごいですね」

「そうなんだよね。フレームワークの人気の指標として Google トレンドを参照してる人が多いけど、これはフェアとは言えない。なぜなら独自の決まりごとが多くなるほど開発者は検索することが増えるので、実際の使用数とは関係なく人気があるよう見えるからね」

「……すいません、私もそういう記事を見て納得してました」

「テンプレート形式を維持しながら複雑な挙動に対応しようとすると、フレームワークは独自の決まりごとが増えて複雑化していく。それにどれだけテンプレート構文を洗練させたとしても、本家の JavaScript を表現力で上回ることはできない。だからこそ新しいパラダイムではその呪縛から解き放たれる必要があったと React の開発チームは考えたわけね」

「なるほど」

「テンプレート形式を採用せず JS ファーストを貫いたことで、React はおぼえないといけない独自の決まりごとは少なく、そのコードも素直な JavaScript になった。JavaScript の高い表現力を 100 % フル活用

してより短いコードでコンポーネントを記述できる。テンプレート形式ではどうやっても早期リターン<sup>8</sup>なんてできないでしょ」

「うーむ、たしかに」

「それにひとつのコンポーネントをいったんテンプレートとロジックで分けて書いてしまうと、コンポーネントが肥大化したときに再分割しづらくリファクタリングの障害になる。だからテンプレート形式のフレームワークではどうしてもコンポーネントの粒度が荒くなりがちで、JS ファーストよりも個々のコンポーネントが肥大化する傾向にある」

「そうなんですか」

「たとえばこの前まで私が担当していた React 製のプロジェクトでは、1 コンポーネントファイルの平均行数は 40.74 で、100 行を超えるコンポーネントファイルの割合は全体の 2.78 % だった。次作るときはさらに短くできる自信があるよ」

「フロントエンドのことはわかりませんけど、マークアップを含んだ Web アプリケーションのファイルとしてはかなり少ないのでよね。一般的なプログラミングにおける 1 ファイルは最大 100 行、理想は 50 行以内という目安のほうにむしろ近い感じ。その数字も JS ファーストだからこそなわけですね」

「そう。React は制御構造が主体だからこそリファクタリングしやすく、よってコンポーネントも純粋に機能によって小さく分割しやすい。さらに最近はコンポーネントをほぼ関数だけで記述できるので、副作用が隔離された関数型プログラミングのスタイルを徹底できる。React のコンポーネントが保守性と拡張性に優れているのは、まさにテンプレートの呪縛から開放されたおかげだね」

「組版指定が主体であるがゆえのリファクタリングのしづらさ、というのも開発者体験を悪くするテンプレートの呪いのひとつというわけですか……」

「そう。テンプレート形式が DX を低下させる要因は他にもまだある。テンプレートはフレームワークによるコンパイルをはさむため、エラーが非常にわかりづらくなる。スタックが深くメッセージも冗長になるため、どこでどんなエラーが起きてるのかを読み取るにはしばしば熟練の技が必要になる」

「たしかに Rails でも、Controller や Model のエラーより View で起きたエラーのほうが特定が難しくて直すのが難しかったですね」

「その点 React のエラーは常に素直な JavaScript のエラーだからね。エラー箇所の特定が難しかったことなんて個人的には記憶にない。

そしてもうひとつ JSX が純粋な式だからこそメリットは、静的解析や型推論に適していること。それによって IDE や構文解析ツールなどのサポートが受けやすいのはもちろん、TypeScript との相性がバツ

---

<sup>8</sup> 関数の途中で値を返して処理を終わらせる手法。条件分岐のネストを避け、以降の処理の関心事を減らすためのテクニック。

## 第5章 JSXでUIを表現する

グンによくなる」

「えっ。TypeScriptとの相性だったら、TypeScriptで開発されてるAngularやVueのほうがいいんじゃないですか？」

「フレームワーク自体がその言語で開発されていることと、アプリケーションをその言語で開発しやすいかは別の問題だよ。テンプレートは型推論が難しいので、Angularにおけるテンプレートの型チェック機能は2020年2月リリースのバージョン9になってようやく搭載された。最初のバージョンから実に3年半がかり。Vueもテンプレートで型チェックをしようと思ったら、別途VS Codeに拡張を入れたりする必要がある。サポートが追加されたといつても現状、両者がReact並みにTypeScriptの型の恩恵を受けられているかとユーザーに聞けば、渋い答えが返ってくるだろうね」

「えー、そうなんですか」

「またAngularはフレームワーク自体が特定のバージョンのTypeScriptに完全依存してるので、TypeScriptだけバージョンを上げることができない」

「えっ、Angularって常に最新バージョンのTypeScriptが使えるんじゃないんですね」

「そのせいで最新のTypeScriptの機能が使えないだけじゃなく、すでに非推奨になっているTypeScriptのライブラリを乗り換えられないという弊害も出てた。だからフレームワークがその言語によって書かれているからといってその言語での開発と相性がいいなんてのは幻想だよ。Reactが三大フレームワークの中でもっともTypeScriptでの開発がしやすいという地位はこれからも揺るがないはず。それもReactがテンプレート形式ではなくJSXを採用してるので依るところが大きいわけ」

「はー、そうだったんですね」

「それにJSXは開始時こそReact固有のものだったけど、今では普通に他のフレームワークも採用してるし、tscやesbuildといったコンパイラやモジュールバンドラまでもが公式サポートしてる」

「たしかに、Reactの普及という後押しあつたでしょうけど、JSX自身が優れたものだからこそReactの枠を超えて使われるようになったんでしょうね」

「テンプレート形式は初見はとっつきやすく見えるけど、複雑なアプリケーションを作ろうとすればその呪いのせいでDXが低下していく。いっぽうJSXは一般的な開発者にとっての初見の印象が悪くて、当初はReactを使用したくない主な理由に挙げられるくらいだけど、本格的に使ってみれば開発者の多くはそのよさを実感し、逆にテンプレート形式のほうがクレイジーだったんだと考えを改めるようになる。秋谷さんも実際に開発で使いこなせるようになればわかるよ」

「うーむ、なんかちょっと宗教ぽくてあやしいような……。じゃあ実際の開発者へのウケはテンプレート派とJSファースト派、どっちがいいんでしょうか？ それでも私の肌感覚ではテンプレート派の支持のほうが多いように感じるんですけど」

「純粋に npm パッケージのダウンロード数だけを比較したグラフはこれだね。なおグラフに載せてないフレームワークは、相対的にダウンロード数が少なすぎて線が底に張り付いてしまうので省略してる」

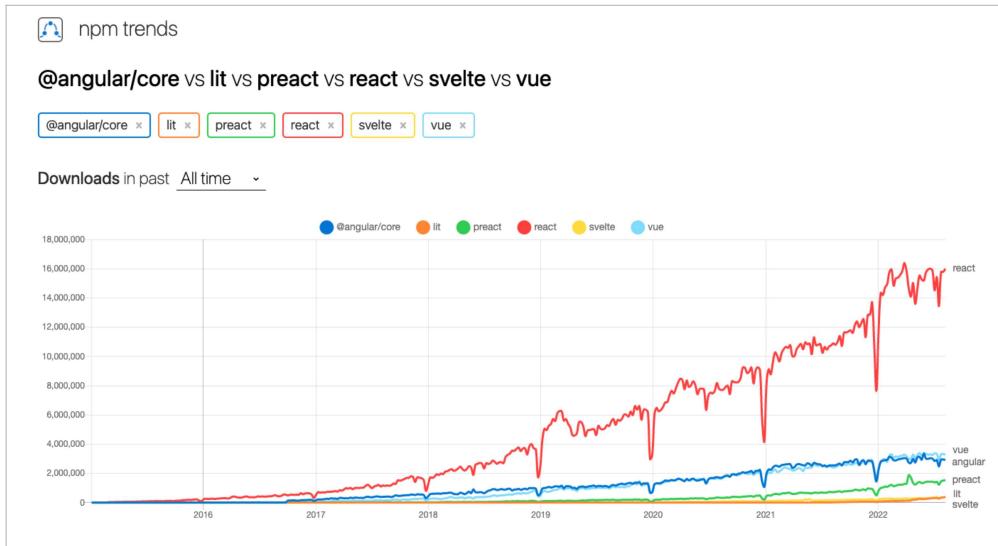


図 3: 各フレームワークの DL 数比較( 2022 年 8 月現在)

「わっ、これ React のひとり勝ちじゃないですか！！　ここまで差がついてるんですか！？」  
 「そうなんだよ。他のすべてのフレームワークのダウンロード数をすべて足し合わせても、やっと React の半分に届くかどうか。JS ファースト派全体というより、まさに React のひとり勝ちなわけ」  
 「……実際の数字でここまで差を見せられると、React の開発者体験がいいというのにすごい説得力を感じてきました」

## 5-1-4. JSX は汎用的に UI を表現する

「JSX が HTML テンプレートと異なるのはその用途にある。JSX の構文はハイパーテキストを表現する HTML ではなく、汎用的な XML がベースになってるの。React がオープンソースとして公開された直後 2013 年 6 月の Pete Hunt による公式ブログの記事『Why did we build React?<sup>9</sup>』には『HTML is just the

<sup>9</sup> 「Why did we build React? -React Blog」 <https://reactjs.org/blog/2013/06/05/why-react.html>

beginning. (HTML はほんの始まりに過ぎない)』というくだりがある。それを読むと React は当初よりレンダラーを本体から分離する設計になっていて、それを入れ替えることでさまざまなプラットフォームのアプリケーションやドキュメントを表現できるようにする意図があったことがわかる。そして現在、React 用の各種レンダラーはめぼしいものだけでこれくらい存在している」

- **React DOM**<sup>10</sup> ..... HTML DOM (公式標準パッケージ)
- **react-test-renderer**<sup>11</sup> ..... JavaScript オブジェクト(公式標準パッケージ)
- **React ART**<sup>12</sup> ..... HTML5 Canvas や SVG などのベクターグラフィック(公式標準パッケージ)
- **React Native**<sup>13</sup> ..... iOS および Android のネイティブアプリケーション
- **React Native for Windows + macOS**<sup>14</sup> ..... Windows および macOS のネイティブアプリケーション
- **React-pdf**<sup>15</sup> ..... PDF ドキュメント
- **react-three-fiber**<sup>16</sup> ..... WebGL による 3D グラフィック
- **React Unity**<sup>17</sup> ..... Unity 3D の UI
- **React Figma**<sup>18</sup> ..... Figma オブジェクト
- **Ink**<sup>19</sup> ..... インタラクティブなコマンドラインアプリ

「React Nativeだけは知っていましたけど、React が対応してるプラットフォームってこんなにあるんですか！」

---

<sup>10</sup> <https://github.com/facebook/react/tree/main/packages/react-dom>

<sup>11</sup> <https://github.com/facebook/react/tree/main/packages/react-test-renderer>

<sup>12</sup> <https://github.com/facebook/react/tree/main/packages/react-art>

<sup>13</sup> <https://reactnative.dev/>

<sup>14</sup> <https://microsoft.github.io/react-native-windows/>

<sup>15</sup> <https://react-pdf.org/>

<sup>16</sup> <https://github.com/react-spring/react-three-fiber>

<sup>17</sup> <https://github.com/ReactUnity/core>

<sup>18</sup> <https://react-figma.dev/>

<sup>19</sup> <https://github.com/vadimdemedes/ink>

「マイナーなものも含めれば、この何倍もあるよ<sup>20</sup>。そしてこれらは別バージョンの React というわけじゃなくて、コア部分は共通でその仮想 DOM を View に反映させるレンダラーのバリエーションということなのね。私たちが Web フロントエンド開発で主に使うのは、HTML DOM のレンダラーである React DOM。Vite でプロジェクトを生成したとき、`src/main.tsx` がこうなってたでしょ？」

リスト 4: `src/index.tsx`

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import './index.css'

ReactDOM.createRoot(document.getElementById('root') as HTMLElement).render(
  :
)
```

「この 1 行めが React コアライブラリを、2 行めが React DOM レンダラーのパッケージからクライアントレンダリングのためのライブラリをインポートしてるってこと」

「へー、これってそういうことだったんですね……」

「記述の仕様はちがうけどレンダラーを React-pdf に差し替えれば PDF がレンダリングされるし、React Native に差し替えれば iOS や Android プラットフォームのネイティブ UI がレンダリングされる。

つまり React コアは Web に限らず包括的にアプリケーションを抽象化するものであり、それを各プラットフォームに合わせて具現化するためのものが各種レンダラーなのね。そしてこれらレンダラーの共通の記述言語が JSX というわけ。JSX の汎用性の高さがわかるでしょ？」

「……なるほど。React は他のフロントエンドフレームワークとは最初からスケールがちがってたんですね。React Native は今や Facebook や Instagram を始めいろんな有名アプリの開発に使われてる技術みたいですし、当初からこの設計が織り込まれていたとはその先見性に恐れ入っちゃいます。それを支えているのが JSX の汎用性の高さだと。JSX が React 跳進の最大要因のひとつっていう柴崎さんの意見も納得です」

「そう、じゃあ JSX に対する先入観はもう払拭されたかな？」

「そうですね。これまでずっとデザインとロジックは分離すべき、そのためにはテンプレート形式がベストだと思い込まれてきたので、なかなかそれは抜けないと思いますけど、少なくとも JSX はキモいからダメだみたいなよくある印象論からは脱却できたと思います。あとは使ってみて、そのよさを実感で

---

<sup>20</sup> <https://github.com/chentsulin/awesome-react-renderer>

きたらですね」

「うん。だいぶ前置きが長くなつたけど、次は実際の JSX 書き方を学んでいこうか」

## 5-2. JSX 構文の書き方

### 5-2-1. JSX の基本的な文法

「ここからは JSX を実際に書いていこう。教材には『Hello, World!』で作ったプロジェクトを使うけど、適宜 TypeScript らしい書き方に変えていくのでそのつもりで<sup>21</sup>。

プロジェクトの `src/` 配下に `main.tsx` と `App.tsx` というファイルがあるよね。これらが JSX ファイルになってる。拡張子が `.tsx` だけど、JSX を JavaScript じゃなく TypeScript をベースにするときはこうする。ただ『TSX』という用語は正式には存在しないので、以降は区別せず JSX と呼びます」

「はい」

「そして JSX は最終的に `ReactElement` オブジェクトの生成式になるわけだけど、式であるがゆえに変数に代入したり、狭義のオブジェクトのプロパティ値にしたり、関数の引数や戻り値にしたりできるというもの、以前に説明したとおり」

リスト 5: `src/App.tsx`

```
import { FC } from 'react';
import logo from './logo.svg';
import './App.css';

const App: FC = () => {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>Hello World!</p>
      </header>
    </div>
  );
};

export default App;
```

<sup>21</sup> <https://github.com/klemiitary/Riakuto-StartingReact-ja4.0/tree/main/05-javascript/02-javascript-hello-world>

「上記の `App` コンポーネントは関数で記述されてるけれども、その関数がJSXを戻り値にしてるのはそういうことだね」

「はい、そこはさっきも説明してもらったので理解します」

「ちなみにテンプレートで生成されたもともとの `App` 関数は型アノテーションなしの関数宣言で定義されてたけど、アロー関数式にした上で呼び出し可能オブジェクトとして `React.FunctionComponent` インターフェースに型付けした。`FC` はそこへのエイリアスね。その型定義はこうなってる」

```
interface FunctionComponent<P = {}> {
  (props: P, context?: any): ReactElement<any, any> | null;
  propTypes?: WeakValidationMap<P> | undefined;
  contextTypes?: ValidationMap<any> | undefined;
  defaultProps?: Partial<P> | undefined;
  displayName?: string | undefined;
}

type FC<P = {}> = FunctionComponent<P>;
```

「`ReactElement` を戻り値にした呼び出し可能オブジェクトで、その他にも `propTypes` とか `displayName` といったプロパティがありますね」

「そのへんのプロパティは歴史的経緯によって残されてるだけで、現在の React × TypeScriptでのアプリケーション開発ではあまり用いられないで、そう気にしなくていいよ。話を JSX に戻すけど、自分が式である JSX にはその中に別の式を埋め込むこともできる。そして式の埋め込みには `{}` を使う。次の内容をさっきのファイルに適当に追加してみよう」

```
const name = "Patty";
const greet = (name: string) => <p>Hello, {name || "Guest"}!</p>;
:
<div>{greet(name)}</div>
```

「`{}` の中で変数の展開や関数コールができますね。これって `{}` の中なら JavaScript や TypeScript のコードがそのまま書けるってことですか？」

「正確には、書いたコードが返す値がそこに埋め込まれる。だから通常は `{}` の中に書くのは式だけね。`if` とか `for` とかの値を返さない制御文は書いても意味を持たないよ。」

そして返す値が `null` と `undefined`、および Boolean 値の `true` と `false` の場合には、そこには何も出

## 第5章 JSXでUIを表現する

力されない。だから JSX では次の記述はすべて同じ出力になる」

```
<div />
<div></div>
<div>''</div>
<div>{undefined}</div>
<div>{null}</div>
<div>{true}</div>
<div>{false}</div>
```

「へ——、単純に JavaScript で falsy な値が表示されないわけじゃないんですね。でもそうか、`n = 0` のとき `{n}` で何も表示されなかったら困りますもんね」

「そうだね。それから JSX の中でも任意の条件によってレンダリングする内容を出し分けたいことがあるよね。その場合はこの Boolean 値が何もレンダリングされない性質を利用する」

```
const n = Math.floor(Math.random() * 10);    // 0 ~ 9 のランダムな整数を生成
const threshold = 5;
:
{n > threshold && (<p>{n} は {threshold} よりも大きい値です</p>)}
{n > threshold || (<p>{n} は {threshold} 以下の値です</p>)}
<p>{n} は {n % 2 === 0 ? '偶数' : '奇数'}です</p>
```

「JSX の `{}` の中では値を返さない `if` などの制御構文が使えないで、こういうときはショートサーキット評価<sup>22</sup> を用いる。どういう働きをするものかおぼえてる？」

「もちろんです！ `&&` では左項が真、`||` では左項が偽のときに右項の式が評価されるんでしたよね。基準値が 5 なのでたとえば `n = 7` だった場合、最初のものだけが評価されて『7 は 5 よりも大きい値です』が表示されるはずです」

「はい、よくできました。これらは `if` 文の代用だったわけだけど、最後の行でやってるのは三項演算子による `if-else` 文の代用ね」

「なるほど。このときも左項の `n % 2 === 0` の部分は Boolean 値を返す条件式なので、JSX の出力には影響しないわけですか」

「そう、原理がわかれば簡単でしょ。また JSX 自体も式だから、JSX に `{}` で埋め込んだ式の中にさらに

---

<sup>22</sup> 「2-7-1. ショートサーキット評価」を参照のこと。

{ }を入れ子にして JSX の記述ができる。この例でもそうしてるけど」

「たしかに、最初の 2つはよく見たら外側の { } 中の評価式で JSX の <p>...</p> を返してますね。…  
…うーむ、テンプレート形式のフレームワークではむしろ v-if みたいな制御構文が積極的に使われるのに、JSX では常に値を返す式のみによって書くとは、関数型プログラミングの風味がここでも強いですね」  
「お、そこに気づいたのはえらいね。だから JSX では繰り返し処理もこんな感じになるの」

```
const list = ['Patty', 'Rolley', 'Bobby'];

return (
  <ul>
    {list.map((name) => (
      <li>Hello, {name}!</li>
    )))
  </ul>
);
```

「おおお……、まさしく関数型ですね……」

「そうだね。値を返す式だから .filter(...).map(...) のようにメソッドチェーンや演算子を用いてさらに高度な表現もできる。それじゃ、次は JSX におけるコメントの書き方を学ぼう」

```
<div>
{
  // インラインコメント
}
/*
  複数行に
  渡るコメント
*/
</div>
```

「JSX はあくまで JavaScript で HTML ではないので、<!-- コメント --&gt; のような HTML 形式のコメントは書けない。{ } による式埋め込みの中で // コメント や /\* コメント \*/ といった JavaScript のコメント記法を使う」</p>

「なるほど、そこもあくまで JavaScript なんですね」

「インラインコメントを使うときは、{ // コメント } のように埋め込み閉じの } の前に // を置くことがないように気をつけてね。} がコメントアウトされて埋め込みブロックが壊れるので。VS Code なら

## 第5章 JSXでUIを表現する

⌘ + / (※ Windows では Ctrl + /) がコメントアウトのショートカットになっているけど、複数行を選択してこのショートカットを使うと、適切な形式が自動的に選択されて正しくコメントアウトされるのでおすすめ】

「へえ、VS Code は賢いな」

「他に JSX を記述する上で気をつけるべき点は、複数の要素が含まれるときにトップレベルがひとつの要素じゃないといけないこと。だからこれはコンパイルエラーになる」

```
const elems = (
  <div>foo</div>
  <div>bar</div>
  <div>baz</div>
);
```

「え、 そうなんですか？」

「この問題を解決する方法は、これをさらに <div> で囲むこと」

```
const elems = (
  <div>
    <div>foo</div>
    <div>bar</div>
    <div>baz</div>
  </div>
);
```

「ただこれだと HTML にレンダリングされたとき意味のないノード階層が余分にできてしまうよね。そこで『フラグメント (Fragment)』を使えば、出力時に不要なノードを追加せずに済む」

```
const elems = (
  <>
    <div>foo</div>
    <div>bar</div>
    <div>baz</div>
  </>
);
```

「空のタグ？ HTML じゃ見たことないですね」

「本当は `<React.Fragment>` というコンポーネントなんだけど、省略記法として空タグが使えるようになってるのね。知らないと初見でびっくりするかもしれないけど、JSX 特有の記法でよく用いられるのでおぼえておいて」

## 5-2-2. JSX とコンポーネントの関係

「おさらいとして JSX 構文は `React.createElement` のメソッドコールに変換され、最終的に `ReactElement` オブジェクトを生成するのはこれまでに説明したとおり。例を実際のコードで書くとこんな感じになるかな」

```
<MyComponent foo="bar">baz</MyComponent>
  ↓
React.createElement('MyComponent', { foo: 'bar' }, 'baz');
  ↓
{
  type: 'MyComponent',
  props: {
    foo: 'bar',
    children: 'baz'
  },
  key: null,
}
```

「ここでは関数によるコンポーネントのみに話を限定すると、`ReactElement` オブジェクトとはそのコンポーネント関数を特定の引数でコールするための実行リンクのようなものだと言える。OS でたとえるなら、アプリケーションの実体に対して任意のオプションを与えて実行するエイリアスやショートカットみたいなもの。その実行リンクが他の実行リンクを子要素として持つツリー構造になってて、レンダリングが走ったときに連鎖して起爆していくイメージ」

「ほおー」

「その実行リンクがコンポーネント関数をコールする際に渡す特定の引数というのは、さっきのサンプルでいうと `ReactElement` オブジェクトの `props` プロパティがそれにあたる。JSX のタグ内で設定した属性値をプロパティとしてオブジェクトにしたもの。この `props` オブジェクトを引数としてコンポーネント関

## 第5章 JSXでUIを表現する

数がレンダリング時にコールされるのね。とりあえず実際に用意したコード<sup>23</sup> でその流れを見てみようか」

リスト 6: 02-jsx-usage/jsx-demo/src/App.tsx

```
import Greet from './components/Greet';
;

const App: React.FC = () => (
  <div className="App">
    <Greet name="Patty" times={4} />
    ;
  </div>
);

export default App;
```

「ここでは他ファイルで定義してある `Greet` コンポーネントをインポートして JSX の中で使ってる。タグ内で属性として定義されてる `name` と `times` が `Greet` コンポーネントに `props` として受け渡されてるのね」  
「すいません、さっきから何度も登場してる『プロップス』って何ですか？」

「`props` とは『properties』から来てて、コンポーネントを関数として考えたときにその引数に相当するものだね。インターフェース `FunctionComponent<P>` は `(props: P, context?: any) => ReactElement<any, any>` | `null` という型の関数として定義されてるので、`props` は第1引数にオブジェクトとして渡される。上記の例なら、`{ name: 'Patty', times: 4 }` が `props` となる。`Greet` コンポーネント本体のコードを見ればわかりやすい」

リスト 7: 02-jsx-usage/jsx-demo/src/components/Greet.tsx

```
type Props = { name: string; times?: number };

const Greet: React.FC<Props> = (props) => {
  const { name, times = 1 } = props;

  return (
    <>
    {[...Array(times)].map(() => (
      <p>Hello, {name}!</p>
    )));
    </>
  );
};
```

<sup>23</sup> <https://github.com/klemiitary/Riakuto-StartingReact-ja4.0/tree/main/05-javascript/02-javascript/jstx-demo>

```
};

export default Greet;
```

「Greet を関数コンポーネントとして型付けするのに、FunctionComponent<P> インターフェースを用いてる。この型引数 P が props の型に適用されるのね。この例では Props という型エイリアスで定義してる。この Props 型が JSX で Greet コンポーネントをタグで書くときの属性値と対応してるわけ。ここでためしにちょっと VS Code で App.tsx の times の属性値を 4 から "foo" とかに書き換えてみて」

「あっ、times に赤線で『型 'string' を型 'number' に割り当てることはできません』と型エラーが出た！」

「Props 型で times は省略可能な数値として定義してるから、文字列値では型が合わないってコンパイラに怒られるわけ。コンポーネントの props と JSX での属性との対応関係がこれでわかったでしょ？」

「はい！ コンポーネントタグの属性として設定した name と times がコンポーネント関数に props として渡されてるんですよね。そして times が省略されたときのために、分割代入で値を取りだすときデフォルト値に 1 を設定してると」

「そうだね。ちなみにここではいったん受け取った props オブジェクトからあらためて分割代入で各属性値を取り出してるけど、引数の受け取り時に直接代入することもできるよ」

```
type Props = { name: string; times?: number };

- const Greet: React.FC<Props> = (props) => {
-   const { name, times = 1 } = props;
+ const Greet: React.FC<Props> = ({ name, times = 1 }) => {
  :
```

「へー、コンパクトにまとまっていいですね！」

「うん、だからこれからはこっちの書き方でいこう。では次に暗黙の props である children について話そうか。JSX で入れ子になってる子要素をそのコンポーネント関数で受け取りたいときがあるんだけど、そういう場合どうすればいいか」

リスト 8: 02-javascript-usage/jsx-demo/src/App.tsx

```
import Greet from './components/Greet';
import Summary from './components/Summary';
import './App.css';
```

## 第5章 JSXでUIを表現する

```
const App: React.FC = () => (
  <div className="App">
    <Greet name="Patty" times={4} />
    <Summary title="Maple Town">
      <p>
        Patty Hope-rabbit, along with her family, arrives in Maple Town, a
        smalltown inhabited by friendly animals.
      </p>
      <p>
        Soon, the Rabbit Family settles in Maple Town as mail carriers and the
        bitter, yet sweet friendship of Patty and Bobby begins to blossom.
      </p>
    </Summary>
  </div>
);

export default App;;
```

「`Summary` コンポーネントの子要素として、`<p>` 要素が 2 つあるよね。これを `Summary` 関数で受け取りたい場合、このように書くの」

リスト 9: 02-jsx-usage/jsx-demo/src/components/Summary.tsx

```
import type { PropsWithChildren } from 'react';
import './Summary.css';

type Props = { title: string; folded?: boolean } & PropsWithChildren;

const Summary: React.FC<Props> = ({ title, folded = false, children }) => {
  console.log(children);

  return (
    <details className="story" open={!folded}>
      <summary>{title}</summary>
      {children}
    </details>
  );
};

export default Summary;
```

「Props がインターフェクション型<sup>24</sup>になってますね。これは何をやってるんですか？」

「React では純粋な JavaScript の世界において子要素 `children` は実際には `props` のプロパティの中に暗黙的に含まれてる。ただし TypeScript の型安全な世界ではこの暗黙の `children` は危険な存在であると `@types/react` のスタッフは考えていて、バージョン 18 以降は `FunctionComponent` インターフェースの `props` の型からこれを削除してしまったの<sup>25</sup>。だから `children` を `props` で受け取りたいときは、上記のように明示的に `PropsWithChildren` の型を交錯する必要があるわけ。この型は次のように定義されてる」

```
type PropsWithChildren<P = unknown> = P & { children?: ReactNode | undefined };
```

「この `ReactNode` とは `string`、`number`、`boolean`、`ReactElement`、`ReactFragment` なんかを包括する型ね。子要素として受け取れるものの型をすべて含んでる。こうして受け取った `children` をコンソールにダンプしてるよね。`yarn dev` で実行して、ブラウザのコンソールで中身を確認してみよう」



図 4: Summary コンポーネントが受け取った `children` の中身

「`type: 'p'` の `ReactElement` が 2 つ渡ってきてるのがわかるでしょ？ これは `App.tsx` で `Summary` コンポーネントの子要素になってた 2 つの `<p>` 要素に対応してる」

<sup>24</sup> 「4-4-2. ユニオン型とインターフェクション型」を参照のこと

<sup>25</sup> 「Removal of implicit children」 <https://solverfox.dev/writing/no-implicit-children/>

## 第5章 JSXでUIを表現する

「なるほど。子要素は `children` プロパティとして `props` の中に含まれてるけど、TypeScript で扱うにはちょっと工夫がいるんですね」

「ちなみに JSX で子要素として文字列を記述するときの挙動として、行の先頭と末尾の空白文字が削除され、空白行も削除される。そして改行はひとつの空白文字に置き換えられる。上記の `Summary` コンポーネントの子要素の `<p>` の中身も、ソースコードの中では読みやすくインデントされてるけど、前後の空白や改行は実際にレンダリングされた HTML には影響してない。ダンプされたデータでもそれは確認できる」

```
▼ 0:  
  :  
  props: {children: 'Patty Hope-rabbit, along with her family, arrives in Maple Town, a smalltown  
inhabited by friendly animals.'}  
  :  
▼ 1:  
  :  
  props: {children: 'Soon, the Rabbit Family settles in Maple Town as mail carriers and the bitter,  
yet sweet friendship of Patty and Bobby begins to blossom.'}  
  :
```

「まあこのへんの挙動は通常の HTML と共通してるので違和感はないはず。またコンポーネントへの属性値の設定の方法も、`{}` による値の埋め込みだけでなく、より HTML っぽい書き方ができる。値が文字列の場合はわざわざ `name={'patty'}` とせずとも `name="patty"` のように書けるの」

「へ——」

「また属性値が `true` の場合は値の記述を省略できる。さっきの `App.tsx` で `Summary` コンポーネントを呼んでるところをこう書き換えてみて。保存したらブラウザで確認」

```
  <Greet name="Patty" times={4} />  
-  <Summary title="Maple Town">  
+  <Summary title="Maple Town" folded>  
    <p>
```

「あっ、説明文のところが折りたたまれた！」

「`props` の `folded` はデフォルトでは `false` だけど、こう書くことで `true` が渡るようになったのね。これも HTML っぽい書き方だよね。JSX によるコンポーネントの記述についての説明はだいたいこんなと

ころかな」

「ありがとうございます。JSX で書いたタグと、その実体となるコンポーネント関数との対応関係がよくわかりました。

ところで疑問に思ったんですけど、JSX によるタグ記述が実際にはコンポーネントの呼び出しになってるというのなら、`<div>` とか `<p>` といった標準の HTML タグに見えるものも `<Greet>` や `<Summary>` と同じコンポーネントということになりますよね？ 自分で定義したコンポーネントと、標準タグに見えるコンポーネントの間には何かちがいがあるんでしょうか？」

「うん、いいところに気がついたね。じゃあそれを説明して JSX の学習の仕上げにしようか」

### 5-2-3. React の組み込みコンポーネント

「実は React のコンポーネントには、ユーザー定義コンポーネントと組み込みコンポーネントの 2 種類があるの。先のサンプルに出てきた `Greet` や `Summary` が前者のユーザー定義コンポーネントに当たり、標準 HTML タグに見える `div` や `p` が組み込みコンポーネントに当たる。

そしてユーザー定義コンポーネントには命名規則があり、コンポーネントの名前を必ず大文字から始めるといけない。`Greet` や `Summary` もそうなってるよね」

「もし小文字から始まる名前でコンポーネントを定義したらどうなるんですか？」

「JSX では小文字から始まる名前のタグ記述は、すべて組み込みコンポーネントだと解釈される。そして組み込みコンポーネントにはそれに該当する名前のものが存在しないので、JSX からはコンポーネントとして呼ぶことができなくなる。

TypeScript の環境では `JSX.IntrinsicElements` インターフェースにおいて、キーがタグ名として登録されている。組み込みコンポーネントとしては現在のところ、HTML 要素と SVG 要素に対応する合計 175 種類が登録されている<sup>26</sup>」

---

<sup>26</sup> 「DefinitelyTyped/types/react/index.d.ts • DefinitelyTyped/DefinitelyTyped」  
<https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/index.d.ts>

## 第5章 JSXでUIを表現する

```
interface IntrinsicElements {
  // HTML
  a: React.DetailedHTMLProps<React.AnchorHTMLAttributes<HTMLAnchorElement>, HTMLAnchorElement>;
  abbr: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  address: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  area: React.DetailedHTMLProps<React.AreaHTMLAttributes<HTMLAreaElement>, HTMLAreaElement>;
  article: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  aside: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  audio: React.DetailedHTMLProps<React.AudioHTMLAttributes<HTMLAudioElement>, HTMLAudioElement>;
  b: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  base: React.DetailedHTMLProps<React.BaseHTMLAttributes<HTMLBaseElement>, HTMLBaseElement>;
  bdi: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  bdo: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  big: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  blockquote: React.DetailedHTMLProps<React.BlockquoteHTMLAttributes<HTMLElement>, HTMLElement>;
  body: React.DetailedHTMLProps<React.HTMLAttributes<HTMLBodyElement>, HTMLBodyElement>;
  br: React.DetailedHTMLProps<React.HTMLAttributes<HTMLBRElement>, HTMLBRElement>;
  button: React.DetailedHTMLProps<React.ButtonHTMLAttributes<HTMLButtonElement>, HTMLButtonElement>;
  canvas: React.DetailedHTMLProps<React.CanvasHTMLAttributes<HTMLCanvasElement>, HTMLCanvasElement>;
  caption: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  cite: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
  code: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>;
```

図 5: JSX.IntrinsicElements インターフェースの中身

「a とか div とか img とか、見たことある HTML 要素がずらーっと並んでますね。なるほど、こんなふうに組み込みコンポーネントとして定義されてるんだ」

「ちなみに『intrinsic』とは『本来備わっている、固有の』という意味ね。このインターフェースからたどって、各要素に props として渡せる固有の属性も確認できるよ。たとえば img コンポーネントの props の型はこうなってる」

リスト 10: @types/react で定義されている ImgHTMLAttributes 型

```
interface ImgHTMLAttributes<T> extends HTMLAttributes<T> {
  alt?: string | undefined;
  crossOrigin?: "anonymous" | "use-credentials" | "" | undefined;
  decoding?: "async" | "auto" | "sync" | undefined;
  height?: number | string | undefined;
  loading?: "eager" | "lazy" | undefined;
  referrerPolicy?: HTMLAttributeReferrerPolicy | undefined;
  sizes?: string | undefined;
  src?: string | undefined;
  srcSet?: string | undefined;
  useMap?: string | undefined;
  width?: number | string | undefined;
}
```

「HTML 標準の `<img>` 要素に指定できる属性は MDN Web Docs の該当ページ<sup>27</sup> とかを参照してもらえばわかるけど、主要な属性を網羅していて属性名もほぼ 1 対 1 で対応してる」

「あれ？ でも微妙にちがいますよね？ `crossorigin` が JSX では `crossOrigin` になってたり、`srcset` が `srsSet` になってたりします」

「そこは JavaScript の命名規則に合わせて、複合語は全般的にローワーキャメルケース<sup>28</sup> に定義し直されてるよ。そして HTML だとブラウザは属性名に対して大文字と小文字を区別しないケース・インセンシティブだけど、JSX は普通の JavaScript だから変数名は当然ケース・センシティブになるので、`crossorigin` や `srcset` と書いたりすると適切な属性として認識されなくなるから気をつけて」

「そうなんですね、わかりました」

「ただしこの命名規則にも例外があって、`aria-*` と `data-*` 属性だけは HTML と同じケバブケース<sup>29</sup> が適用されてる」

「『ARIA』？ ……って、未来のテラフォーミングされた火星で一人前のウンディーネを目指す少女たちの物語ですか？」

「いやその ARIA じゃなくてね。『Accessible Rich Internet Applications (ARIA)<sup>30</sup>』っていう Web アクセシビリティの標準を定めた規格があって、`aria-` で始まる HTML 属性がいくつも定義されてるよ。主に視覚障害者用の読み上げブラウザのために意味づけられたものなんだけど」

「へえ——、そんな規格があるんですね」

「React の型定義ではそれらは `AriaAttributes` インターフェースとして定義されてる。それでもういっぽうの `data-*` はカスタムデータ属性<sup>31</sup> といって、HTML5 の仕様で追加された、開発者がオリジナルの属性を作ることができるものね。たとえば E2E テストで DOM を特定するための ID を埋め込む用途に使われたりする。Twitter とかでブラウザのデベロッパーツールから DOM 要素を確認すると、`data-testid` のようなカスタム属性が使われてるのがわかるよ」

「へ——、知らなかった」

<sup>27</sup> 「`<img>`: 画像埋め込み要素 - HTML: HyperText Markup Language | MDN」  
<https://developer.mozilla.org/ja/docs/Web/HTML/Element/img>

<sup>28</sup> 「lowerCamelCase」のように複合語をひとくくりにして、頭文字を小文字で始め、以降の各単語の最初を大文字で書き表す記法。単に「キャメルケース」とも。

<sup>29</sup> 「kebab-case」のように複合語をひとくくりにして、各単語を小文字で記述し、ハイフン（-）で連結する記法。「ハイフンケース」「リスプケース（Lisp Case）」とも。

<sup>30</sup> 「ARIA - アクセシビリティ | MDN」  
<https://developer.mozilla.org/ja/docs/Web/Accessibility/ARIA>

<sup>31</sup> 「`data-*` - HTML: HyperText Markup Language | MDN」  
[https://developer.mozilla.org/ja/docs/Web/HTML/Global\\_attributes/data-\\*](https://developer.mozilla.org/ja/docs/Web/HTML/Global_attributes/data-*)

## 第5章 JSXでUIを表現する

「さらに React の組み込みコンポーネントの `props` と標準 HTML 要素の属性とではいくつか挙動が異なる点があるので、それらを紹介しておこう。まず JavaScript の文字ケースの挙動によるもののに、JavaScript の予約語とかぶってしまったせいで名前自体を変更せざるをえなかったものがある。それがこの 2つね」

- `class` → `className`

- `for` → `htmlFor`

「知ってると思うけど、HTML における `class` は要素にスタイルを適用するための CSS クラス名を設定するもの。`for` は `<input>` や `<select>` といったラベル付け可能なフォーム要素への入力の関連付けを `<label>` 要素に設定するためのものね」

「はい、知っています。ちょっと変に見えますけど、JavaScript の予約語とかぶっちゃったんじゃしかたないですね。でも 2つだけだし、ちゃんとおぼえておきます」

「それから HTML での挙動と異なり、その値が Boolean になってる属性が次の 3つね」

- `checked`

- `disabled`

- `selected`

「HTML だと `checked="checked"` とか書いてましたよね。それが `checked={true}` のようになるわけですか」

「そう。そしてさっきも説明したけど値が `true` のときは `<input type="checkbox" checked>` のように省略記法が使える。あと説明すべきは `value` 属性かな。React では `<select>` と `<textarea>` も `value` 属性が持てるようになってるのが HTML と異なってる」

リスト 11: `select` と `textarea` コンポーネントの `value` 属性

```
<form>
  <label htmlFor="favChar">好きなキャラクターは？</label>
  <select id="favChar" value="fox">
    <option value="rabbit">/ティ</option>
    <option value="bear">ボビー</option>
    <option value="fox">ダイアナ</option> /* selected */
    <option value="pig">プリプリン</option>
    <option value="squirrel">ジュディ</option>
  </select>
  <label htmlFor="favReason">そのキャラクターのどこが好き？</label>
```

```
<textarea id="favReason" value="【例】見た目が好き" />
</form>
```

「HTML では `<textarea>` は内容はその子要素として持つようになってましたっけ。`<select>` については、そもそも値を持てないんでしたよね。それが子要素の `option` と同じ `value` を設定すると、その要素が選択された状態になるんですね」

「そのとおり。これらは React が JavaScript でフォームを操作しやすいように挙動を改変したわけだね。ただしこれをこのまま実行すると、ブラウザのコンソールで warning が出てしまう。React では `<input>`、`<textarea>`、`<select>` といったフォーム要素の値はそれらを利用しているユーザー定義コンポーネントの中で状態を持ってそれを反映するように作らなければいけなくなってる。だから `value` 属性に固定値を設定してしまうと React が warning を出すの。こういう場合は `defaultValue` を使うようにしよう<sup>32</sup>」

```
<form>
  <label htmlFor="favChar">好きなキャラクターは？</label>
  - <select id="favChar" value="fox">
  + <select id="favChar" defaultValue="fox">
    :
  </select>
  <label htmlFor="favReason">そのキャラクターのどこが好き？</label>
  - <textarea id="favReason" value="【例】見た目が好き" />
  + <textarea id="favReason" defaultValue="【例】見た目が好き" />
</form>
```

「これも HTML にはなくて React の組み込み要素にしか存在しない属性ですね」

「そうね。静的なページを表現する HTML とちがって、動的なアプリケーションを構築するためには JSX で改変しないといけない部分はどうしても出てくる。まちがった書き方をしてる場合は React がブラウザのコンソールに warning を表示してくれる所以、常に注意しておこう」

「わかりました。……って言われて見てみましたけど、`value` を `defaultValue` に直してもまだ何か warning が出てますよ？」

<sup>32</sup> 「デフォルト値 - 非制御コンポーネント -React」

<https://ja.reactjs.org/docs/uncontrolled-components.html#default-values>

## 第5章 JSXでUIを表現する

```
✖ Warning: Each child in a list should have a unique "key" prop. react-jsx-dev-runtime.development.js:87
Check the render method of `Greet`. See https://reactjs.org/link/warning-keys for more information.
  at p
  at Greet (http://localhost:3000/src/components/Greet.tsx:20:5)
  at div
  at App
```

図 6: リストには “key” props が必要という警告メッセージ

「そうそう。React 開発でよく怒られるのが、このリスト要素における `key` 属性のつけ忘れね。繰り返し処理で同階層に同じ要素のリストを表示させる際、React は個々の要素にユニークな `key` 属性値を必要とする。React では差分のあったコンポーネントを検知してそこだけを再レンダリングするようになってるので、同列に並んでいるコンポーネントをユニークに区別するためのこの値を用いるの<sup>33</sup>」

「へー。その `key` にはどんな値を入れるのがいいんですか？」

「そのリストの中でユニークになる文字列や数値だね。`key` としての理想的な値は、データベースの primary key のような、そのデータコレクションが持つユニーク ID。……なんだけど、それがない場合もあるよね。そういう場合は仕方ないので、繰り返しのインデックスを流用することになる」

リスト 12: Greet.tsx の差分

```
return (
  <>
-   {[...Array(times)].map(() => (
-     <p>Hello, {name}!</p>
+   {[...Array(times).keys].map((i) => (
+     <p key={i}>Hello, {name}!</p>
    )));
  </>
);
```

「あ、warning が消えましたね！」

「最後にもうひとつだけ HTML には存在しないけど React の組み込みコンポーネントにある属性を紹介しておこう。`ref` は組み込みコンポーネントを実際にレンダリングされるリアル DOM へ結びつける参照のための属性。ちょっと今の時点では理解は難しいと思うけど、雰囲気的にはこんなふうに使われる」

<sup>33</sup> 「リストと key=React」 <https://ja.reactjs.org/docs/lists-and-keys.html>

リスト 13: 02-jsx-usage/src/components/TextInput.tsx

```

import { useRef } from 'react';

const TextInput: React.FC = () => {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleClick = () => {
    inputRef.current?.focus();
  };

  return (
    <div>
      <input type="text" ref={inputRef} />
      <input type="button" value="フォーカス" onClick={handleClick} />
    </div>
  );
};

export default TextInput;;

```

「……これは、うーん、えーっと……」

「まずここでやりたいことというのは、組み込みコンポーネントの `input` とそれに対応する実際にレンダリングされた後のリアルな DOM とを結びつけ、その要素の `focus()` メソッド<sup>34</sup> を実行したいわけね。組み込みコンポーネントの `ref` 属性の中に React が用意している `RefObject` というオブジェクトを設定しておくと、そのコンポーネントが DOM にレンダリングされた際に、`RefObject` の `.current` プロパティにその DOM への参照値を入れてくれるようになってるの」

「んー、わかったようなわからないような……。とりあえずリアル DOM への参照が `ref` という属性ができるということだけおぼえておきます。それにしてもコンポーネントからリアル DOM を参照することってそんなにあるんですか？」

「そうだね、今やったようなフォーカスだったり、テキストの選択やアニメーションの発火だったり、動画・音声再生の管理みたいなことをやりたいときに `ref` を使うかな。私は無限スクロールを自分で実装したとき、DOM 要素の高さを取得するために `ref` を使ったことがあるよ」

「ふーむ、なるほど」

「じゃあこれで JSX についての説明はいったんおしまい。React のコンポーネントの実装についてはまた

<sup>34</sup> [HTMLElement.focus() - Web API | MDN]

<https://developer.mozilla.org/ja/docs/Web/API/HTMLElement/focus>

## 第5章 JSXでUIを表現する

あらためて説明するけど、JSX がどういうものかわかつてもらえた？」

「はい、だいたいは。JSX は基本的に `ReactElement` オブジェクトを生成する式へのシンタックスシュガーで、しくみとして JavaScript の枠を超えたものではないこと。HTML ライクに書ける `<div>` や `<p>` も、実際には React の組み込みコンポーネントに変換されるということ。属性値や子要素はコンポーネントに `props` という関数に対する引数のようなものとして渡されること。こんな感じでだいじょうぶですか？」

「ちゃんと要点は押さえてるみたいだね。あとはどんどん書いて慣れていく。JSX を使うのが機能単位でコンポーネントを分割・構築するのに最適な方法というのが実感できるはず」

# 第6章 進化したビルドツールを活用する

## 6-1. コンパイラとモジュールバンドラ

### 6-1.1. フロントエンド開発に使われるコンパイラ

「ここまであまり深くふれずに Vite を使ってもらってたんだけど、Vite が実際に何をしてくれてるので、というのをそろそろ説明しておきたいのね。それにはまずフロントエンドアプリケーションのビルドにまつわるツールに、どんなジャンルが存在してるとか知っておいてもらいたい」

「ツールのジャンルってそんなにあるんですか？」

「まずはコンパイラから。現代のフロントエンド開発において、どんなブラウザでも解釈できる古いバージョンの JavaScript やモジュールシステムで直接コーディングすることはほとんどない。私たちがこれまでやったように TypeScript や JSX を使ったり、最低でも ES2015 以降のバージョンの JavaScript で ES Modules を使って開発する」

「まあ、そうですよね」

「ここまで tsc を使って TypeScript のコードを JavaScript にコンパイルしてきた。tsc は『TypeScript Compiler』から来てる名前ね。とろあらためて聞きたいけど『コンパイル (Compile)』ってどういうことだと理解してる？」

「えーっと、C とかだとコンピュータが直接実行できるバイナリを出力することだし、Java なら JVM が実行できるバイトコードに変換することですよね。TypeScript の場合はブラウザが実行できるような JavaScript に変換することだから、高水準言語のコードをコンピュータなりアプリケーションなりが実行できる形のファイルに変換することじゃないでしょうか？」

「うん、近いけど答えとしては 85 点くらいかな。実行できるかどうかは関係なく、ある言語で書かれたプログラムをより低いレベルのコードに変換する行為のことをコンパイルっていうの。似た言葉として、ある言語のプログラムを同水準の言語のコードに変換することを『トランスペイブル (Transpile)』がある。TypeScript → JavaScript の変換がより低水準なのか同水準なのかは微妙なところなので、コンパイルと呼ばれることもあればトランスペイブルと呼ばれることもある」

「へー、そうなんですね」

「Meta 謹製の React 環境構築ツールである Create React App<sup>35</sup> では、TypeScript のコンパイルに Babel<sup>36</sup> というプロダクトが使われてる。Babel は最初 6to5 という名前で、その名のとおり ES2015 (ES6) のコードを ES5 に変換するためのツールだった。そして当初は『トランスペイラ』を自称してたんだけど、後にプラグインシステムを採用して変換元の対象に TypeScript や JSX といったさまざまな言語や文法に対応するようになり<sup>37</sup>、そこから『コンペイラ』を名乗るようになった」

「ツールが進化したことで、プロダクト名だけじゃなくジャンルまで変わったんですね」

「そう。フロントエンド関連のツールは進化が速く、ツールがひとつのジャンルに収まらなくなることがよくある。だからこの先、ジャンルごとにツールを紹介していくけど、そのジャンル分けは便宜上そうしてるけど時には他のものに分類されたりすることもあるというのを理解しておいて。そのことを踏まえた上で、フロントエンド開発のツールで現在、純粋なコンペイラと呼べるのは tsc と Babel くらいかな」

### 6-1-2. モジュールバンドラの登場

「現代のフロントエンド開発において本番に展開されるファイルは往々にして、コンパイルだけじゃなくモジュールバンドルという工程を経たものになってるの。bundle っていうのは『束にする、セット売りにする』といった意味の言葉ね。つまり複数のモジュールに分割された JavaScript のファイルをまとめるここと。そしてそれを行うツールを『モジュールバンドラ (Module Bundler)』と呼ぶ」

「素朴な疑問なんですけど、どうしてモジュールをまとめる必要があるんですか？」

「最近になってフロントエンド開発を始めた人にはピンと来ないかもしれないね。でも歴史の初期には強い必然性があったのよ。まずブラウザそのものがモジュールに対応していなかった。さらに当時用いられてたプロトコルの HTTP/1.0 および 1.1 には通信の同時接続数に大きな制限があった。モダンブラウザがドメインごとに最大 6 つの同時接続が可能になった HTTP/2.0 に対応し始めたのは 2015 年ごろから、ES Modules への対応はさらに遅れて 2017 年ごろから。だからそれ以前はどうにかして ESM で書いたコードを当時のブラウザでも読み込める形に変換すること、およびファイルの数を減らすことへの強いニーズがあったわけ」

「なるほど」

「CommonJS で書いたコードをブラウザでも利用できるようにする Browserify というプロダクトを

---

<sup>35</sup> <https://create-react-app.dev/>

<sup>36</sup> <https://babeljs.io/>

<sup>37</sup> 「Plugins List • Babel」 <https://babeljs.io/docs/en/plugins-list>

JavaScript モジュールの説明をしたときに紹介したよね<sup>38</sup>。そういう働きをするツールのことを『モジュールローダー (Module Loader)』といって、2011 年に登場した Browserify も当時それらの中のひとつだったんだけど、バージョンアップでユーザーのニーズに応えていく内に、モジュール間の依存解決をしつつファイルをまとめるモジュールバンドラとしての機能を備えるようになった』

「そこでもジャンルのブレーカスルーが起きたんですね？」

「そう。そして Web フロントエンドではこの Browserify に gulp<sup>39</sup> や Grunt<sup>40</sup> といったビルトの自動化を行うためのタスクランナーというツールを組み合わせて開発するのが主流となり、その時代が長く続いた。そこに ES2015 の登場で ES Modules が導入され、その普及に伴ってこれを使って書いたコードをブラウザに実行できるように変換したいというニーズが高まっていったの」

「ふむふむ」

「そして大本命の webpack が脚光を浴びることになる。プロダクト自体は 2012 年と他のプロダクトと同じくらい古い歴史があったんだけど、2017 年に正式リリースされたバージョン 2.2 から ES Modules をネイティブサポートするようになって一気に化けた。パッケージのダウンロード数の推移からもそれがうかがえるね」

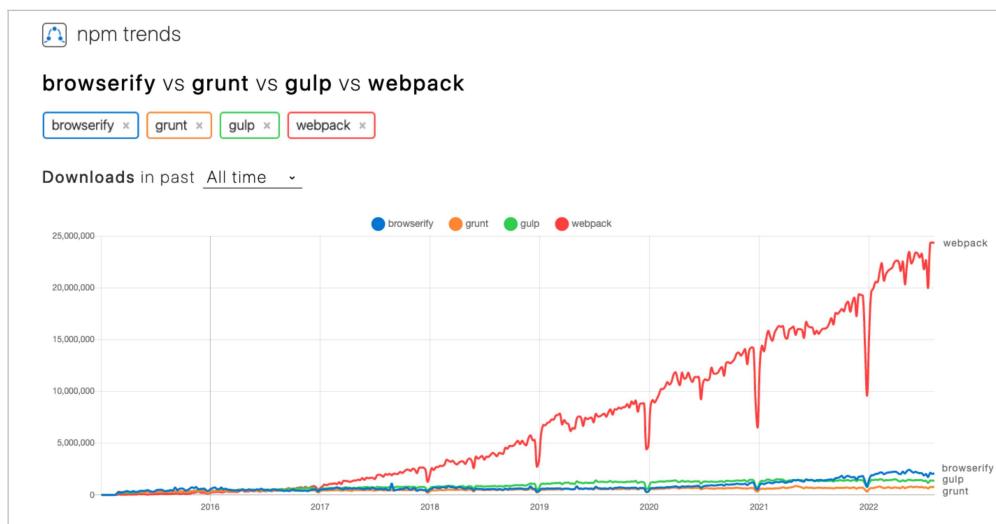


図 7: モジュールローダ、バンドラの DL 数比較(2022 年 8 月現在)

<sup>38</sup> 「①言語・環境編」の「2-9-1. JavaScript モジュール三國志」を参照のこと。

<sup>39</sup> <https://gulpjs.com/>

<sup>40</sup> <https://gruntjs.com/>

## 第6章 進化したビルドツールを活用する

「たしかに 2017 年を境に爆発的に伸びて、他を圧倒してますね」

「この webpack の何がすごかったって ES Modules、CommonJS、AMD を含めたさまざまなモジュール構文をサポートしていて<sup>41</sup>、使われている構文を自動で検出し適切に解釈してくれるので、異なるモジュール構文が混在していてもおかまいなしに依存関係を解決してバンドルしてくれるね。」

npm パッケージには今でも CommonJS 形式で提供されているものが少なくなく、それは React も例外じゃない<sup>42</sup>。ましてや当時はほとんどが CJS 形式だっただろうから、webpack を使えばそれらを ESM 形式の `import` 構文をフルに使って読み込むことができるようになったのは、開発者にとって大きな恩恵だった」

「そうでしょうね。導入するパッケージが ESM なのかとか CJS なのかとか、意識したくないですもん」

「さらに webpack は画像やフォントなどのアセットファイルもバンドルできるようになり、バンドリング工程の前に介入してあらゆるものを JavaScript オブジェクトとして扱えるようにするローダー（Loader）<sup>43</sup> というものが組み込めるようになった。それによって CSS や HTML、さらに Babel や tsc と連携して TypeScript や JSX などバンドルできるファイルの種類を拡大していった」

「コンパイラも飲み込んでいったんですか……。もう webpack があれば他には何もいらないじゃないですか」

「一強状態となった webpack の進化はさらに続き、他にも次のような機能を備えるようになった」

- **Minify** …… 空白文字やコメント、実行されないコードやコンソール出力などを削除、変数名や関数名を短縮、条件文や宣言文を簡略化するなどして、スクリプトファイルの容量を最小限に圧縮する
- **Tree Shaking** …… どこからも参照されていないモジュールを検出してバンドル対象から除外し、出力ファイルの容量を削減する
- **画像データのINLINE埋め込み** …… 画像ファイルをデータ URL<sup>44</sup> としてドキュメントにINLINEで埋め込む
- **コード分割** …… モジュールの共有度合による自動分割に加え、開発者がアプリケーションの UI や規模を考慮しつつ、バンドルファイルを適切な粒度に分割し、それぞれが適切なタイミングで読み込まれるように

---

<sup>41</sup> <https://webpack.js.org/api/module-methods/>

<sup>42</sup> とりあえず現行バージョンの 18 系までの話。次のメジャーバージョン 19 からは ESM になるというウワサです。

<sup>43</sup> <https://webpack.js.org/loaders/>

<sup>44</sup> 「データ URL - HTTP | MDN」

[https://developer.mozilla.org/ja/docs/Web/HTTP/Basics\\_of\\_HTTP/Data\\_URLs](https://developer.mozilla.org/ja/docs/Web/HTTP/Basics_of_HTTP/Data_URLs)

できる

- **キャッシュ管理** …… ビルド時に出力ファイルの名前に差分単位でユニークなハッシュ値<sup>45</sup> を付与することで、ブラウザキャッシュが適切に使われるようになり、かつキャッシュが残ることによる挙動の不具合を予防する

「こんなに！？」

「webpack を始めとする最新のモジュールバンドラがやっているのは、もはやネイティブアプリのビルドと遜色ない作業であって、ビルド先がバイナリコードかテキストコードかのちがいでしかないと見える。そのターゲットが Android アプリなら ARM 系 CPU の上で動く Android OS なのと同様、モダンフロンティア開発ではそれがネットワークの向こうにある V8 や JavaScriptCore<sup>46</sup> や SpiderMonkey<sup>47</sup> が動いてるブラウザというだけの話なのよ」

「だから ESM に対応し、HTTP/2.0 はおろか 3.0 までが動くようになったブラウザが普及している現在でも、モジュールバンドラの役割が終わったなんてことはなく、むしろますます使われるようになってる。将来もずっとそうかはわからないけど、しばらくの間はこの傾向が続くだろうね」

「うーむ、ESM でモジュールシステムが統一されたらバンドラがいらなくなるなんて単純な話じゃないんですね」

### 6-1-3. 新しい波を生んだ次世代モジュールバンドラ

「柴崎さん、webpack が一強状態って言いましたけど、2017 年からけっこう時間が経ってますよね。これまで webpack の地位を脅かすようなモジュールバンドラは出てこなかったんですか？」

「そんなことはないよ。いくつも新しいバンドラが登場して、新しい波を作り出してきた。まず最初に紹介したいのは Parcel<sup>48</sup>。これは公式サイトのトップページにでかでかと掲げているように、『Zero

<sup>45</sup> データを一意に識別する固定長の数値のこと。大量のデータをはるかに小さい数値として表すため、ファイルの改ざん検知や同一性の確認に役立つ。

<sup>46</sup> Safari のブラウザエンジン Webkit に搭載されている JavaScript フレームワーク。Safari の文脈では「Nitro」とも呼ばれる。JIT コンパイラと LLVM（コンパイル時やリンク時、実行時などあらゆる時点でプログラムを最適化するよう設計されたコンパイラ基盤）を採用しており、高速に JavaScript を実行可能。

<sup>47</sup> 1995 年リリースの Netscape Navigator 2.0 の β 版に搭載された世界初の JavaScript エンジン。その後オープンソース化され、現在は Mozilla Foundation がメンテナンスを行っている。継続的に機能の強化が図られ、Firefox の JavaScript エンジンとして受け継がれている。

<sup>48</sup> <https://parceljs.org/>

# 著者紹介

## 大岡由佳（おおか・ゆか）

技術同人作家。React 専門のフリーランスエンジニアとして複数の現場を渡り歩いた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。なお現在は常駐も顧問も請け負っていないため、事実上おそらく国内（世界でも？）に唯一生息する、専業の技術同人作家。

趣味はホームシアターでの映画・ドラマ鑑賞と、3年以上続けながらちっとも上達しないクラシックバレエ。

Twitter アカウントは @oukayuka。ブログは [klemiwary.com](http://klemiwary.com)。

## 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「くるみ割り書房」では、新刊や紙の本の再版予定、その他読者の方へのお知らせなどをメールマガジン形式で不定期に配信しています。

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ピングです)

# りあクト! TypeScript で始めるつらくない React 開発

## 第4版【①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4版、全3巻構成の第1巻「言語・環境編」。

フロントエンド開発において JavaScript および TypeScript のスキルは大事な基礎体力。その基礎を固めつつ、関数型プログラミングの考え方など実際の React 開発において押さえておきたい技術やさらに進んだ書き方までを学んでいきます。

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！

(2022年9月10日発行／B5版・231p／電子版 1,200円、紙+電子セット 1,380円)

### 《第1巻 目次》

- 第1章 こんにちは React
- 第2章 ライトでディープな 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4版【③React応用編】



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4版、全3巻構成の第3巻「React応用編」。

React のルーティングや状態管理、副作用処理を歴史を踏まえつつ包括的に解説しています。Redux や React 18 の新機能 Concurrent Rendering についても詳しく説明。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！

(2022年9月10日発行／B5版・188p／電子版1,200円、紙+電子セット1,380円)

### 《第3巻 目次》

- 第10章 コンポーネント操作の高度な技法
- 第11章 React アプリケーションの開発手法
- 第12章 React のページをルーティングする
- 第13章 Redux でグローバルな状態を扱う
- 第14章 ポスト Redux 時代の状態管理
- 第15章 React 18 の新機能を使いこなす

## 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／B5版・92p／電子版 1,000円、紙+電子セット 1,100円)

### 《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! Firebase 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／B5版・136p／電子版 1,500円、紙+電子セット 1,600円)

### 《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

# りあくト! TypeScript で始めるつらくない React 開発 第4版

## 【② React 基礎編】

---

2022年9月10日 初版第1刷発行  
2022年9月10日 電子版バージョン 1.0.1

著 者 大岡由佳  
発行者 大岡由佳  
発行所 ぐるみ割り書房  
連絡先 oukayuka@gmail.com

---