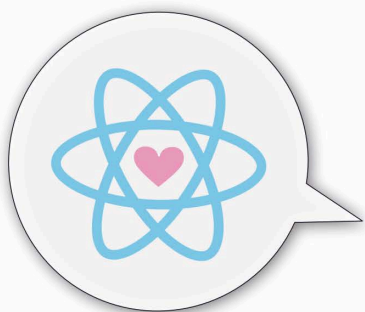


# いあクト!



第3版

Sample

TypeScriptで始める  
つらくないReact開発  
【II. React基礎編】

大岡由佳

最新のReact 17.0に完全対応

歴史から知るReactの本質!  
JSX, Hooksが納得して理解できる

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……」

シリーズ累計  
**8千部**  
突破!

前版  
BOOTH売上  
**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マン指導により、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の基礎が始まるが、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の歴史と思想を語り始める。

### 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**

**【II. React 基礎編】**

大岡由佳

くるみ割り書房



## 第二部まえがき

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。

本書は三部作の第二部「Ⅱ. React 基礎編」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 の知識を得ました。第二部ではまずソースコードの静的解析ツールとフォーマッタを導入した後、React の基礎知識を身につけた上で、その基本的な使い方を学んでいきます。

第二部は本作の中でもっとも opinionated な、つまり思想性の強いタイトルとなっています。React が日本の少なくない開発者に敬遠されるいちばんの原因である JSX について類書に見られないほど詳細に分析し、むしろ JSXこそが React が覇権を握った大きな要因であると結論づけています。第5章は巷にあふれる「JSX キモい」という否定的な感情論に対する筆者なりの答えです。

「第7章 React をめぐるフロントエンドの歴史」は、本作中もっとも調査に時間がかかった章です。この章はいったん最後まで書き終えた後、内容的に浅く感じられ納得がいかなかった部分をふくらませ、あらためて章として独立させたものです。フロントエンド全体の歴史の流れから React を検証するという作業は、フロントエンド途中参入組である筆者自身にとっても React の理解がかなり深まる有意義なものでした。

Hooks が React の新しいスタンダードになった 2020 年の今日において、クラスコンポーネントを今版で引き続き紹介するかどうかは悩みました。しかし第一部のまえがきにも書きましたが、本作は何よりも「仕事で使える React 本」を目指しています。職業エンジニアの読者が現場でクラスコンポーネントに遭遇する可能性は依然高いはずである以上、クラスコンポーネントについては説明を残しています。ただ位置づけとしては歴史的経緯をたどるためのものに留め、全体の内容は関数コンポーネントによる Hooks ファーストの立場をとっています。

## 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



# 目次

第二部まえがき .....	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 5 章 JSX で UI を表現する .....	12
5-1. なぜ React は JSX を使うのか .....	12
JSX の本質を理解する .....	12
なぜ React は見た目とロジックを混在させるのか .....	15
なぜ React はテンプレートを使わないのか .....	19
なぜ React は View をタグツリーで表現するのか .....	25
5-2. JSX の書き方 .....	30
JSX の基本的な文法 .....	30
JSX とコンポーネントの関係 .....	35
React の組み込みコンポーネント .....	39
第 6 章 Linter とフォーマッタでコード美人に .....	46
6-1. ESLint .....	46
JavaScript、TypeScript における Linter の歴史 .....	46
ESLint の環境を作る .....	50
ESLint の適用ルールをカスタマイズする .....	55
6-2. Prettier .....	66
特別なコードフォーマッタ Prettier .....	66
ESLint のプラグインとして Prettier をインストール .....	69
6-3. stylelint .....	73
6-4. さらに進んだ設定 .....	77

<b>第7章 React をめぐるフロントエンドの歴史</b>	<b>81</b>
7-1. React 登場前夜	81
すべては Google マップショックから始まった	81
フロントエンド第2世代技術の興隆	83
7-2. Web Components が夢見たもの	85
7-3. React の誕生	88
7-4. React を読み解く 6 つのキーワード	90
公式サイトトップに掲げられている三大コンセプト	90
Declarative (宣言的)	93
Component-Based (コンポーネントベース)、Just The UI (UI にしか関知しない)	95
Virtual DOM (仮想 DOM)	98
One-Way Dataflow (単方向データフロー)	103
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)	105
7-5. 他のフレームワークとの比較	108
第3世代のフレームワーク情勢	108
Angular	110
Vue.js	113
LitElement、そして Web Components	116
<b>第8章 何はなくともコンポーネント</b>	<b>121</b>
8-1. コンポーネントのメンタルモデル	121
8-2. コンポーネントと Props	122
8-3. クラスコンポーネントで学ぶ State	131
コンポーネントをクラスで表現する	131
クラスコンポーネントに State を持たせる	135
8-4. コンポーネントのライフサイクル	142
8-5. Presentational Component と Container Component	149
<b>第9章 Hooks、関数コンポーネントの合体強化パーツ</b>	<b>153</b>
9-1. Hooks に至るまでの物語	153
手軽ながら壊れやすかった Mixins	153
コミュニティによって普及した HOC	156

HOC の対抗馬 Render Props .....	160
ついに Hooks が登場する .....	164
9-2. Hooks で State を扱う .....	169
9-3. Hooks で副作用を扱う .....	174
Effect Hook の使い方 .....	174
Effect Hook とライフサイクルメソッドの相違点 .....	179
9-4. Hooks におけるメモ化を理解する .....	183
9-5. Custom Hook でロジックを分離・再利用する .....	191

## 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

## 第三部「Ⅲ. React 応用編」 目次

### 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
- 10-2. ルーティングライブラリの選定
- 10-3. React Router (5 系) の API
- 10-4. React Router をアプリケーションで使う
- 10-5. React Router バージョン 5 から 6 への移行

### 第 11 章 Redux でグローバルな状態を扱う

- 11-1. Redux の歴史
- 11-2. Redux の使い方
- 11-3. Redux 公式スタイルガイド
- 11-4. Redux Toolkit を使って楽をしよう
- 11-5. Redux と useReducer

### 第 12 章 React は非同期処理とどう戦ってきたか

- 12-1. 過ぎ去りし Redux ミドルウェアの時代
- 12-2. Effect Hook で非同期処理
- 12-3. 「Redux 不要論」を検証する

### 第 13 章 Suspense でデータ取得の宣言的 UI を実現する

- 13-1. Suspense とは何か
- 13-2. “Suspense Ready”なデータ取得ライブラリ
- 13-3. Suspense の優位性と Concurrent モード
- 13-4. Suspense と Concurrent モードが革新する UX

## 第5章 JSXでUIを表現する

### 5-1. なぜ React は JSX を使うのか

#### JSX の本質を理解する

「モダンフロントエンド開発に必要な JavaScript と TypeScript の知識をひとつとおり得られたので、次は 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。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するね。『JSX』という名前は『JavaScript』と『XML』の組み合わせでできてる。JSX は XML ライクな記述ができるようにした ECMAScript 2015 に対する構文拡張なの」

「『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、後付けで使えるようにしたものだね。実際には JSX は Babel や tsc によってコンパイルされることを前提としたシンタックスシュガーになってる。たとえばこの JSX コードは React をターゲットにしたコンパイルの結果、次のように変換される」

リスト 1: 変換前の JSX コード

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

リスト 2: 変換後の JavaScript コード

```
React.createElement(
  'button',
```

---

<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エンジン。Ruby 処理系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。

```
{ type: 'submit', autoFocus: true },
  'Click Here'
);
```

「JSX がやってるのは基本、この `React.createElement` というメソッドのコールへの変換を前提に XML のタグとその組み合わせによるノードツリーを JavaScript の中でシームレスに書けるようにしてるだけなのね。そしてこのメソッドコールは、ここでは内容を簡略化するけど、実際には次のようなオブジェクトを生成するの」

```
{
  type: 'button',
  props: {
    type: 'submit',
    autoFocus: true,
    children: 'Click Here',
  },
  key: null,
  ref: 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となる。つまり JSX の構文は、本質的には `ReactElement` オブジェクトを生成するための式だってことね。ゆえに JSX は `ReactElement` オブジェクトを表現するための JavaScript の拡張リテラルだと思っておけば、ニュアンス的にはだいたいあってるかな」

「……式、……リテラル」

「秋谷さんが最初に言った『JSX はテンプレート言語』という認識がまちがってるのがそこだね。JSX は JavaScript においては単なるオブジェクトを表現する式に還元されるものであって、特別な存在じゃない。だから変数に代入したり、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することもできるわけ。こんなふうだね」

```
const element = <span>foo</span>;
const obj = { bar: element };
const wrap = (element) => <div>{element}</div>;
```

「……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」

「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、本質がわかっているれば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>2</sup>、Babel プリセットの設定を変更すれば他のフレームワークでも使えるようになってる。Vue.js や Mithril は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」

「へー、Vue でも JSX が使えるんですか。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 部分の記述のために使われる。コンポーネントが関数であればその関数の戻り値として、クラスであれば `render()` メソッドの戻り値として JSX が返されるようにするの。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<{ deptId: string }> = ({ deptId }) => {
  const [users, error, isLoading] = useFetchMembers({ deptId });
  const formatDate = (d: Date) =>
    `${d.getYear() + 1900}年${d.getMonth() + 1}月${d.getDay()}日`;

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

    return <div>failed to get users data</div>;
  }

  return (
    <CardGroup>
      {users.map((user) => (
        <Card
          header={user.fullName}
          cover={user.profileImage}
          meta={user.birthday ? formatDate(user.birthday) : null}
          description={user.bio}
        />
      ))}
    </CardGroup>
  )
}
```

---

<sup>2</sup> <https://facebook.github.io/jsx/>



```
</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ったんですが、React って見た目とロジックのコードがまぜこ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分かれてたので、そこにすごく違和感があります」

「サーバサイド Web アプリケーションの開発から入ってきた人は、たいていそういうね。じゃあなぜ React では見た目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」

## なぜ React は見た目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。図解で示すと図 1 のようになるかな」

「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 では開発者の何の関心によって要素を分割しているかという、アプリケーション横断的に**技術の役割**によって 3 つに分離してる」

「ふーむ、まあそうですね」

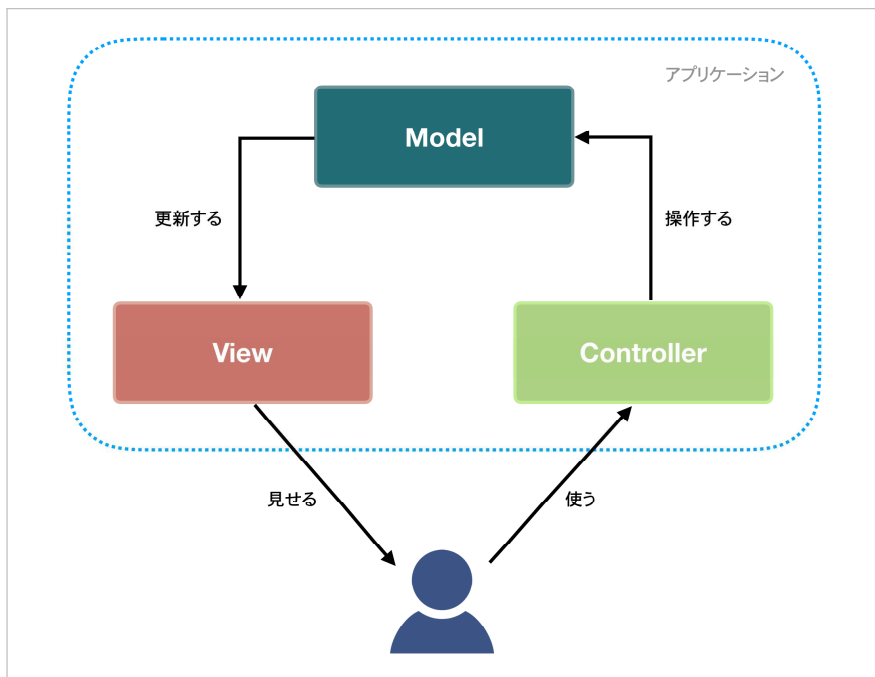


図 1: MVC アーキテクチャの概念図

「この技術の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？ そうなんですか？」

「フロントエンドフレームワークがどんな歴史をたどって React に行き着いたかは、またあらためて説明してあげるよ<sup>3</sup>。そしてその React においては、関心の分離の単位が MVC のような技術の役割じゃないの。アプリケーションの**機能**がその単位になってる。機能単位で分割された独立性の高いパーツ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、そのパーツが**コンポーネント**というわけ。独立した機能単位のパーツとして分割するためには、そのパーツの中に見た目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ツ同士が疎結合にならないでしょ」

「むむむむ……」

---

<sup>3</sup> 「第7章 React をめぐるフロントエンドの歴史」にて説明します。

「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の高さ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しょ。でも React はコンポーネント自身が、自身の描画に必要な処理を自分でやり、レンダリングもコンポーネントごとに個別で行われるの。そしてそれらはコンポーネント単位で並列に非同期で実行される」

「……非同期、……並列。そういえば Twitter とかは、ページ全体のレイアウトはすでにレンダリングされてるのに、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 製だね。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。概念図にすると図 2 のような感じかな」

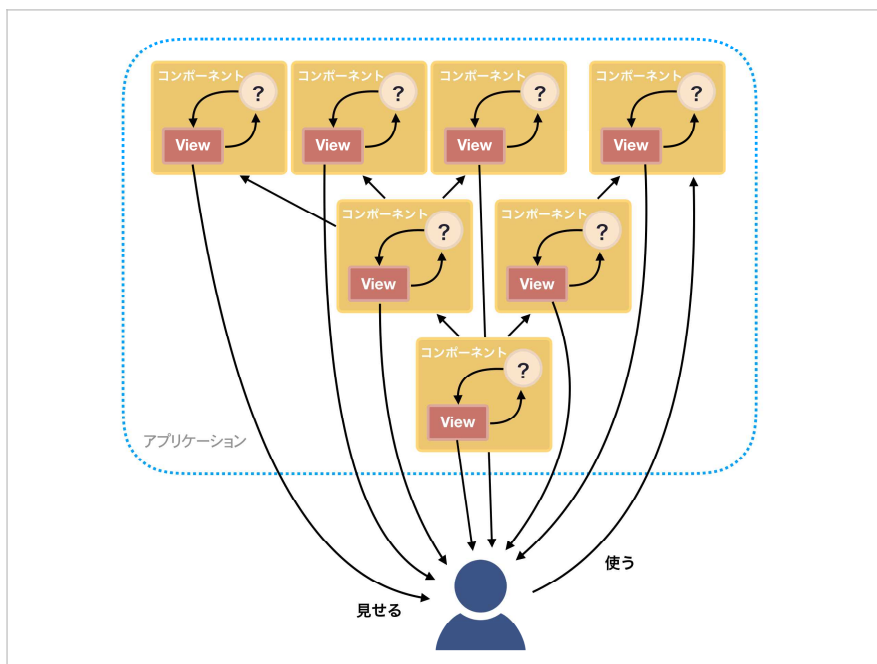


図 2: React のコンポーネントベースなアーキテクチャの概念図

「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、状態の変化がコンポーネントにどう影響するかとかというのは、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非効率だと React では考えるのね。これはまあ、実際に開発してみない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が、じゃあデザイナーとの分業はどうなるんですか？ これま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 をデザイナーさんが書くのは無理じゃないでしょうか？」

「それを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さんにしてもらおう？ デザイナーさんが作ってくれるのはサイズやレイアウトが確認できる Sketch や Figma のデザインカンパまででしょ。アプリの開発者はそれを見て自分で UI を組んでいく」

「ええっ、それはそうですけど。……うーん」

「実際、SPA の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 デザインを専門にっていて HTML を自分で書けない人も多い」

「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。まあ今では Sketch や Figma から HTML や CSS を直接書き出すこともできるし、CSS もコンポーネントの名前空間に閉じられて管理がシンプルなので、昔ほど大変じゃないよ」

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

「私たちが作ろうとしているのは、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ではないの」

「……その話、webpack のときにしてもらったのとつながってますね<sup>4</sup>」

---

<sup>4</sup> 第2章、第9節内の「webpack はフロントエンド標準ビルドツールの夢を見るか？」を参照のこと。

「そうだね。同じ『Web アプリケーション』といっても、React や webpack が対象としてるのは、従来のサーバサイド Web アプリケーション開発者が想定するようなそれとは往々にして異なってる。次はその話をしようか」

## 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ベースだ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してしまったために、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の。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」

「あ、そういえば Vue とかはテンプレートで書きますもんね」

「そう。テンプレートを専用ファイルとして分離するか、Vue.js の単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れども、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ディレクティブを用いた制御構文を埋め込むという形は共通してる」

「ふむふむ」

「それとは対比的に、ここまで見てきたように React では JavaScript で一貫して view のレンダリングも行う。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。この思想は『JS ファースト』と呼ぶことができるね」

「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たのかは、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 は 2013 年の JSConf EU でのセッション『React: Rethinking Best Practices』<sup>5</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、その部分の理由は次の 2 つの言葉に集約されるかな」

---

<sup>5</sup> 「Pete Hunt: React: Rethinking best practices – JSConf EU」

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

- 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 inevitably tightly coupled.

(見た目のロジックとマークアップは必然的に分かちがたく結びついてる)

「ふむふむ」

「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**

AngularJS、Angular、Vue.js、Ember.js、Aurelia、Svelte など

- **JS ファースト派**

React、Preact、Mithril、Cycle.js、hyperapp など

「まさに二分されてますね。それにしてもどうしてフロントエンドのフレームワークは、統一されたり複数の派閥が乱立したりするのではなく、この二大派閥に分かれるのでしょうか？」

「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イルアプリ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、<sup>くみはん</sup>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、必然的に色々なところに無理が出てくる。

## 5-1. なぜReactはJSXを使うのか

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作らざるをえなくなる。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見えるけど、本格的なアプリケーションを開発しようとするとき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。Vue.js は『Progressive Framework（漸進的なフレームワーク）』を自称してるけど、まさに物は言いようだよな（笑）」

ちなみに Vue.js も Angular も npm のダウンロード数は React よりずっと少ないのに、Google トレンドの検索数では長い間 React を上回ってたのは、その決まりごとが多すぎて開発者がしょっちゅう検索しないとコードが書けないからだよ。検索数の多さがフレームワークの人気度の証だと誤解している人が多いようだけど」



図 3: Google トレンドでの各フレームワークの検索数比較（2020 年 9 月現在）

「……すいません、私もそう誤解していました」

「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、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が少なく、そのコードも素直な JavaScript そのもの。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しても早期リターン<sup>6</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再分割しづらくなって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」

「へえ、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が 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」

「へー、フロントエンドのことはわかりませんが、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ですね。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細分化しやすい。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 DX 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まだあるよ。テンプレートはフレームワーク

---

<sup>6</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。



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なため、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정이難しかったことなんて個人的には記憶にない。

それでもうひとつ JSX が純粋な式だからこそのメリットは、静的解析や型推論に適していること。それによって IDE や Lint といったツール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 がいちばんいいんじゃないんですか？」

「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 年半がかりで、それでもまだ完全ではなく型推論の結果が any になってしまうものも残ってる。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」

「うん。そのせいで最新の TypeScript の機能が使えないだけでなく、すでに非推奨になってる TypeScript のライブラリを乗り換えられないという弊害も出てる。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。Vue.js もバージョン 3 から TypeScript 製になるけど、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揺るがないはず。それもテンプレート形式ではなく JSX を採用してるのに依るところが大きい」

「はー、そうだったんですね」

「それにそもそも今の TypeScript は、言語レベルで JSX をサポートしてるからね」

「ええっ？」

「TypeScript のコンパイルオプションについて説明したとき、`jsx` というオプションがあったでしょ

<sup>7</sup>。あれを `react` に設定すると、`tsc` 自身が JSX を `React.createElement()` の関数コールに変換してくれるようになるの。これは 2015 年 9 月リリースのバージョン 1.6 から導入されてる<sup>8</sup>。TypeScript からここまでの特別扱いを受けてるのは JSX だけだね」

「えー、なんか React だけ<sup>ひいき</sup>最<sup>き</sup>優先されてずるくないですか？」

「JSX はいちおうは他のフレームワークでも使うことができる React から独立した仕様だし、純粋な式に還元できるものだからこそ言語エンジンでのサポートが簡単なんだよ。

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、しばしば 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 DX の高さに納得して、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ば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？ それでも私にはテンプレート派の支持のほうが多いように思えるんですけど」

「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図 4 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」

「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、React の半分にも届かない。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の差を見せられると、React の DX が高いというのにすごい説得力を感じてきました。でも他の JS ファースト派のフレームワークはそこまで使われてないですよ。それはどうしてなんでしょうか？」

---

<sup>7</sup> 第 4 章、第 8 節内の「TypeScript のコンパイルオプション」を参照のこと。

<sup>8</sup> [ TypeScript: Handbook - TypeScript 1.6 - JSX support ]  
<https://www.typescriptlang.org/docs/handbook/release-notes/typescript-1-6.html#jsx-support>

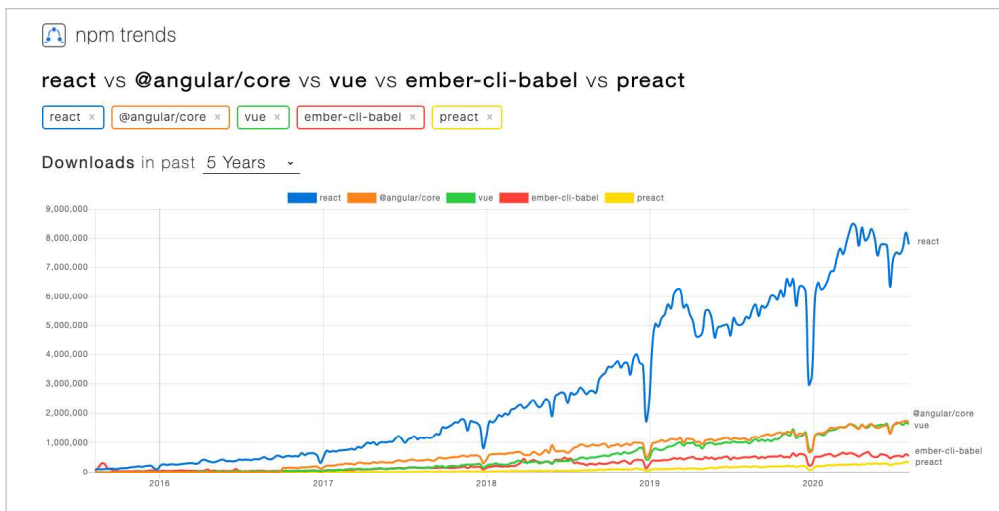


図 4: 各フレームワークの DL 数比較 (2020 年 9 月現在)

「実は JS ファースト派の中でも JSX を標準採用してるのは React と Preact<sup>9</sup> の 2 つ、といっても Preact は React のミニマルなサブセットのフレームワークなので、実質 React だけが JSX を採用しているといいていい。でもこの、初見で多くの開発者に忌避されてしまいがちなこの JSX こそが、React の圧倒的なシェアにつながった最大の要因のひとつだと私は考えてる」

## なぜ React は View をタグツリーで表現するのか

「JSX こそが React の圧倒的シェアの最大要因ってどういうことですか？」

「JS ファースト派に挙げたフレームワークの内、React と Preact 以外は JSX ではなく HyperScript<sup>10</sup> というライブラリを view のレンダリングに使ってるの。HyperScript はこういう記法で HTML を表現する」

```
import h from 'hyperscript';

const list = ['one', 'two', 'three'];
```

<sup>9</sup> <https://preactjs.com/>

<sup>10</sup> <https://github.com/hyperhype/hyperscript>

```
h(  
  'div#page',  
  h(  
    'header.title',  
    h('h1.classy', 'h', { style: { 'background-color': '#22f' } })),  
  ),  
  h('nav.menu', { style: { 'background-color': '#2f2' } }},  
    h('ul', list.map((n) => h('li', n))),  
  ),  
  h(  
    'body.article',  
    h('h2', 'content title', { style: { 'background-color': '#f22' } }},  
    h(  
      'p',  
      "so it's just like a templating engine,\n",  
      'but easy to use inline with javascript\n',  
    ),  
  ),  
);
```

「JSX に批判的な人はしばしば『テンプレートじゃないならわざわざタグを使わず、`React.createElement()` で全部書くべきだ』みたいなことをいうけど、HyperScript はまさに React における `React.createElement()` のメソッドコールに相当するものを汎用的に簡略化できるライブラリだね」

「なるほど、JSON 形式で記述することにより冗長なタグを書かなくて済むようにしてるんですか。コードの記述量が減ってシンプルになりますね」

「うん。それで記述量が減るのはいいんだけど、秋谷さんはこれが本当に見やすいと思う？ これを読みやすいと主張する人がいるのは知ってるけど、はたしてその感覚は一般的だろうか。JSON 的なデータを表現するには JavaScript のオブジェクトリテラルは最強だけど、属性を持った要素がその子として複数のテキストまたは別の要素を持ってというノードツリーによるデータ構造を視覚的に表現するにはまったく適してないよね」

「うーん、まあそうかもしれませんね。ただ囲みタグ表記を省略してシンプルに書けるようにするのは、Rails にも代替テンプレートエンジンとして `Haml`<sup>11</sup> や `Slim`<sup>12</sup> とかがあります。私が以

---

<sup>11</sup> <http://haml.info/>

<sup>12</sup> <http://slim-lang.com/>

前いたチームでも Haml を使っていましたし」

「うん、知ってる。でも Rails だってタグを書く形式のテンプレートエンジンである ERB をそのまま使ってる人は別として、タグを省略できる Haml のダウンロード数は高速版 ERB である Erubis の 1/4 にも満たないんだよ<sup>13</sup>。タグ省略型のほうが DX が高いというなら、この数字はおかしいでしょ」

「あ、Haml ってそこまで使われてなかったんですね。チームで採用されてたから使ってたけど、正直なところは、タイピング量こそ減るもののあんまり読みやすいとは私も思ってませんでした…」

「そう、手数は増えるかもしれないけど囲みタグ形式は視認性において優れてるんだよ。<> タグによるブロックがマークアップとそれ以外のコードを視覚的に分けてくれるので、人間の認知にやさしい。実際のエディタではシンタックスハイライトでタグ部分が色分けされるので、視認性はさらに高まるし。

React 向けの HyperScript ライブラリ<sup>14</sup>も存在してるけど、わざわざ JSX から乗り換えようという人はほとんどいないよ。Vue.js 用の JSX の Babel プラグイン<sup>15</sup>が週に 20 万件以上ダウンロードされてるのと比較にならない」

「JS ファーストで制御ロジックにマークアップが混在するからこそ、マークアップ部分がパッと目でわかる視認性が余計に大事だってことですね」

「そう、そのとおり。さらにいうなら JSX が表記にタグを用いるのはテンプレート形式を模倣して HTML を直接書くためというより、単に XML の記法が UI を表現するのに適しているからというニュアンスのほうが強いと思う。アプリケーションを構成するコンポーネントを表現するには各種の属性値を持ったオブジェクトのノードツリーを用いるのが最適で、それをコードで視覚的に表現するのもっとも優れているのが囲みタグ形式だったというだけなんだよ。多分に結果的なものだけだね」

「ふむふむ、なるほど」

「HyperScript がその名のとおりのハイパーテキスト、つまり HTML のみを対象にしていたのに対して、JSX が XML 全般を表現するためのものといううちがいが大きい。React がオープンソースとして

---

<sup>13</sup> 「Category: Template Engines - The Ruby Toolbox」 [https://www.ruby-toolbox.com/categories/template\\_engines](https://www.ruby-toolbox.com/categories/template_engines)

<sup>14</sup> <https://github.com/mlmorg/react-hyperscript>

<sup>15</sup> <https://github.com/vuejs/babel-plugin-transform-vue-jsx>

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>16</sup>』には『HTML is just the 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 はその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、レンダラー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しようとしていたことがわかる。そして現在、React 用の各種レンダラーは活発に開発が行われているものだけでもこれくらい存在してる」

- **React DOM**<sup>17</sup> …… HTML DOM (公式標準パッケージ)
- **react-test-renderer**<sup>18</sup> …… JavaScript オブジェクト (公式標準パッケージ)
- **React ART**<sup>19</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>20</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>21</sup> …… Windows および macOS のネイティブアプリケーション
- **React 360**<sup>22</sup> …… ブラウザ上で動く VR アプリケーション
- **React-pdf**<sup>23</sup> …… PDF ドキュメント
- **react-three-fiber**<sup>24</sup> …… WebGL による 3D グラフィック
- **React Figma**<sup>25</sup> …… Figma プラグイン

---

<sup>16</sup> 「Why did we build React? – React Blog」 <https://reactjs.org/blog/2013/06/05/why-react.html>

<sup>17</sup> <https://github.com/facebook/react/tree/master/packages/react-dom>

<sup>18</sup> <https://github.com/facebook/react/tree/master/packages/react-test-renderer>

<sup>19</sup> <https://github.com/facebook/react/tree/master/packages/react-art>

<sup>20</sup> <https://reactnative.dev/>

<sup>21</sup> <https://microsoft.github.io/react-native-windows/>

<sup>22</sup> <https://facebook.github.io/react-360/>

<sup>23</sup> <https://react-pdf.org/>

<sup>24</sup> <https://github.com/react-spring/react-three-fiber>

<sup>25</sup> <https://react-figma.now.sh/>

• **React Sketch.app**<sup>26</sup> …… Sketch ファイル

「React Native だけは知ってましたが、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！」

「マイナーなものも含めれば、この何倍もあるよ<sup>27</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。Create React App でプロジェクトを生成したとき、src/index.tsx がこうなってたでしょ？」

リスト 3: src/index.tsx

```
import React from 'react';
import ReactDOM from 'react-dom';
:
ReactDOM.render(<App />, document.getElementById('root'));
```

「この 1 行めが React コア本体を、2 行めが React DOM レンダラー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……」

「このレンダラーを react-test-renderer に差し替えれば JavaScript オブジェクトがレンダリングされるし、Canvas API で 2D グラフィックを動的にレンダリングしたい場合は React ART が提供する UI コンポーネントをインポートして使えばいい。

つまり React とは Web にとどまらずアプリケーションやドキュメントを包括的に抽象化するのであり、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 は今やクロスプラットフォーム開発ではいちばん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 躍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きました」

<sup>26</sup> <http://airbnb.io/react-sketchapp/>

<sup>27</sup> <https://github.com/chentsulin/awesome-react-renderer>

「そう、じゃあJSXに対するバイアス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るべき、そのためにはテンプレート形式がベストだと思い込まされてきたので、なかなかそれは抜けないと思いますが、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、そのよさを実感できたらですね」

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

## 5-2. JSX の書き方

### JSX の基本的な文法

「JSXは`React.createElement`のメソッドコールに対するシンタックスシュガーだという話は最初にしたよね。じゃあちょっと『Hello, World!』で作ったプロジェクトをVSCodeで開いて、`src/App.tsx`を見てもらえるかな」

リスト 4: 01-hello/02-hello-world/src/App.tsx

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        :
      </header>
    </div>
  );
}
```

「この1行めのインポート文をよく見てみて。ここでインポートしたはずの`React`はこのファイルではどこからも参照されていない。なのになぜこの文が必要なんだろう。ちょっとこの行を削除してみようか」

「わわっ、JSXの記述部分が軒並み赤のアンダーラインで警告が出ちゃいました。えーっと、メッセージの内容は『JSXを使うときは`React`がスコープの中にいる必要がある』、ですかね？」



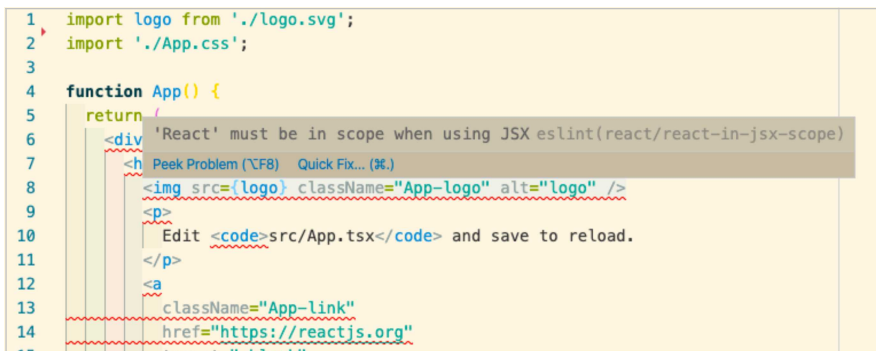


図 5: React のインポート文を削除すると……

「そう。Create React App で作成したプロジェクトには ESLint というコードの静的解析ツールとその共有設定である `eslint-config-react-app`<sup>28</sup> が最初から組み込まれていて、React における構文の簡単なまちがいを指摘してくれるようになってるのね。それが警告を出してくれてるの。これは JSX の構文が `React.createElement()` に変換されるから `createElement` メソッドの上位モジュールである `React` がインポートされてないと変換後、参照エラーになってしまうという警告なのね」

「ああ、そっか。見えなくても JSX では `React` が使われてるわけでもんね。だから JSX の構文を書く前には `React` のインポートが必要なんですね」

「うん、それが JSX を使う際の大前提。ちなみにこのファイルの拡張子は `.tsx` だけど、これは JSX を JavaScript じゃなく TypeScript ベースにするときの拡張子ね。ただ『TSX』という用語は正式にはないので、以降は区別せず JSX と呼びます。

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することもできるというのも、以前に説明したとおり。この `App` コンポーネントは関数だけれども、JSX を戻り値にしてい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てます」

「じゃあ次、自身が式である JSX にはその中に別の式を埋め込むこともできる。式の埋め込みには `{}` を使う」

```
const name = 'Patty';
```

<sup>28</sup> <https://www.npmjs.com/package/eslint-config-react-app>

## 第5章 JSXでUIを表現する

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

return <div>{greet(name)}</div>;
```

「`{}`」の中で変数の展開や関数コールができてますね。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？」

「いや、あくまで書けるのは『式』、つまり値を返す表現だけだよ。だから `if` とか `for` とかの値を返さない制御文は書けないので気をつけて。

それから `null` と `undefined`、および boolean 値の `true` と `false` は `{}` の中では何も出力されない。だから次の6つはすべて同じ出力になる」

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

「へ——、falsy な値だけでなく、`true` も何も出力されないんですか」

「そうだね。それで JSX に埋め込めるのは式だけだから `if` 文などの制御文を中に書くことはできないことはさっきも説明したとおりだけど、任意の条件によってレンダリングするものを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の」

```
const n = Math.floor(Math.random() * 10); // 0 ~ 9 の整数を生成
const threshold = 5;

return (
  <div>
    {n > threshold && <p>'n' is larger than {threshold}</p>}
    <p>'n' is {n % 2 === 0 ? 'even' : 'odd'}</p>
  </div>
);
```

「`<div>` タグのすぐ下に見えるのが `if` 文を代用する JSX での式表現で、`&&` 論理演算子によるシ

ショートサーキット評価<sup>29</sup>を用いてる。その下の行で用いてるのが `if-else` 文の代用で、三項演算子を用いてこのように書くわけ」

「なるほど。`n > threshold` や `n % 2 === 0` といった式は `true` か `false` を返すので、JSX の中に埋め込んでも出力には影響されないわけですか」

「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 に埋め込んだ式の中にさらに JSX を記述することもできる。この例でもそうしてるけど」

「たしかに、よく見たら `{}` の中に `<p>` で囲まれた JSX があって、さらにその中で `{threshold}` という変数の値が展開されてますね。

……うーむ、テンプレートではむしろ制御構文が積極的に使われるのに、JSX では文は書けず常に値を返す式でないといけないって、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。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よ。

それじゃ、次は JSX におけるコメントの書き方ね」

```
<div>
  {
    3 > 1 && 'foo' // インラインコメント
  }
  {/*
```

<sup>29</sup> 第2章、第7節内の「ショートサーキット評価」を参照のこと。

```
    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 -->` 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」

「なんで2～4行めはまとめて1行で書かないんですか？　まとめちゃいましょう……、ってあれっ？　以降がエラーになっちゃいました。なにに、『`} expected`』？」

「インラインコメントの `//` は JSX の式埋め込みの閉じカッコ `}` までもコメントアウトしてしまうんだよ。どうしても1行にまとめたい場合は `/* コメント */` を使うしかないね」

「……なるほど、そういうことですか」

「JSX を記述する上で他に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>  
);
```

「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)』を使うことで、不要なノードを追加することなく複数の要

素をまとめて扱うことができるようになるの」

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

「空<sup>から</sup>のタグ？ HTML じゃ見たことないですね」

「本当は `<React.Fragment>` なんだけど、省略記法で空<sup>から</sup>タグが使えるのね。知らないで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」

## JSX とコンポーネントの関係

「まずおさらいとして 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。サンプル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,
  ref: null,
}
```

「ここではいったんコンポーネントを関数で定義したものとすると、`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いえるね。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」

「ふーむ、なるほど」

「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 `ReactDOM` 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オブジェクト化したものに子要素を `children` プロパティとして追加したオブジェクト。この `props` オブジェクトを引数としてコンポーネント関数がレンダリング時にコールされるのね。とりあえず実際のコード<sup>30</sup>でその流れを見てみようか」

リスト 5: 02-jsx-usage/src/App.tsx

```
import React from 'react';
import Greetings from './components/Greetings';
:
const App: React.FunctionComponent = () => (
  <div className="App">
    <Greetings name="Patty" times={4}>
      <span role="img" aria-label="rabbit">🐇</span>
    </Greetings>
    :
  </div>
);

export default App;
```

「これはすでに定義済みの `Greetings` コンポーネントを JSX で呼んでるコードね。タグ内で属性として定義されてる `name` と `times` が props に相当する」

「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 (プロパティ)』の略で、コンポーネントを関数として考えたとき、その引数に相当するものだね。クラスコンポーネントなら `props` というメンバー変数として設定されるんだけど、基本的にはコンポーネントが呼ばれるときに外から与えられる読み取り専用の変数グループをオブジェクトにまとめたもの。まあ `Greetings` コンポーネント本体の実装コードを見れば何となくわかると思うよ」

---

<sup>30</sup> <https://github.com/oukayuka/Riakuto-StartingReact-ja3.0/tree/master/05-jsx/02-jsx-usage>

リスト 6: 02-jsx-usage/src/components/Greets.tsx

```
import React from 'react';

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

const Greetings: React.FunctionComponent<Props> = (props) => {
  const { name, times = 1, children } = props;

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

export default Greetings;
```

「Greetings を関数コンポーネントとして型定義するため、`React.FunctionComponent<P>` インターフェースを適用してる。このジェネリクスになってる `P` が `props` の型になるのね。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 VSCode 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『Type `'string'` is not assignable to type `'number | undefined'`』とコンパイルエラーが出ました！」

「`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いわれてるのね。コンポーネントの `props` と JSX でのタグ属性との対応関係がこれでわかったでしょ？」

「はい！ この `name` と `times` が `Greetings` 関数に `props` として渡されるわけですね。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。でも最後の `children` って何ですか？ `Props` には定義されてないのに引数として受け取れてるのがおかしくないですか？」

「これは `React.createElement()` の第 3 引数に相当するものだよ。JSX においては属性値ではなく子

## 第5章 JSXでUIを表現する

要素として記述され、呼ばれた側のコンポーネントでは暗黙の props として渡されるようになってる。だからここでは `<span role="img" aria-label="rabbit">🐇</span>` が `children` として設定されてるね」

「ふーむ、`children` は暗黙の props ですか。なるほど」

「なお JSX 構文における属性値としての props の渡し方だけど、通常は `times={4}` のように `{}` による式埋め込みで値を渡すのね。ただ値が文字列の場合は `name="patty"` のようにクォーテーションで囲む形式によって渡すこともできる。また HTML エスケープされた文字列を props として渡すと受け取ったコンポーネント側で元の文字列に復元される」

「ふむふむ」

「さらに 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。例を示そうか」

```
<SummaryText>
  &lt;Summary>&gt;<br />
  Patty Hope-rabbit, along with her family, arrives in Maple Town,
  a small town inhabited by friendly animals.

  Soon, the Rabbit Family settles in Maple Town as mail carriers and the bitter,
  yet sweet friendship of Patty and Bobby begins to blossom.
</SummaryText>
```

「このような JSX の記述は、`SummaryText` コンポーネントに対し `children` として次のような配列データが渡されることになるわけね」

```
children = [
  '<Summary>',
  <br />,
  'Patty Hope-rabbit, along with her family, arrives in Maple Town, \
  a small town inhabited by friendly animals. Soon, the Rabbit Family \
  settles in Maple Town as mail carriers and the bitter, yet sweet \
  friendship of Patty and Bobby begins to blossom.',
];
```

「なるほど。たしかに HTML エスケープが復号されて、要素間の前後の空白文字・空白行は削除



されてますね」

「その他の JSX での特殊な属性値の記述だけど、Boolean 値で `true` の場合は値の記述を省略することもできる。よって次の 2 つは同じことになる」

```
<MyButton color="blue" disable={true} />
<MyButton color="blue" disable />
```

「JSX によるコンポーネントの記述についての説明はこんなところかな」

「ありがとうございます。JSX はテンプレートのように見えるけど、あくまで `ReactElement` オブジェクトを生成するためのシンタックスシュガーというの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 タグも `<Greetings>` や `<SummaryText>` と同じコンポーネントじゃないんですか？ これらの間に実際に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 についての学習の仕上げにしようか」

## 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。そしてユーザーが自前のコンポーネントを定義するとき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ないといけない。さっきサンプルで示した `Greetings` や `SummaryText` コンポーネントがそうだったよね」

「小文字の名前でコンポーネントを定義するとどうなるんですか？」

「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ね。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から。TypeScript なら `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ものがそれにあたる。組み込みコンポーネント用のタグとして登録されているのは現在のところ、HTML 要素と SVG 要素の合計 175 個になってる<sup>31)</sup>」

<sup>31)</sup> [DefinitelyTyped/types/react/index.d.ts](https://bit.ly/2ArYGSG) · [DefinitelyTyped/DefinitelyTyped](https://bit.ly/2ArYGSG) | <https://bit.ly/2ArYGSG>

```
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 6: 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いますね。なるほど、こんなふうに定義されてるんだ」

「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>` ならこうなってる」

リスト 7: <https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/index.d.ts>

```
interface ImgHTMLAttributes<T> extends HTMLAttributes<T> {  
  alt?: string;  
  crossOrigin?: "anonymous" | "use-credentials" | "";  
  decoding?: "async" | "auto" | "sync";  
  height?: number | string;  
  loading?: "eager" | "lazy";  
  referrerPolicy?: "no-referrer" | "origin" | "unsafe-url";  
  sizes?: string;  
  src?: string;  
  srcSet?: string;  
  useMap?: string;  
  width?: number | string;  
}
```

「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>32</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 `srcSet` になってたりしてます」

「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>33</sup> に書き直されてるのね。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>34</sup> が適用されてるの」

「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>35</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のよ。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——、そんなのがあるんですね」

「React の型定義では `AriaAttributes` インターフェースとして定義されてる。そしてもういっぽうの `data-*` はカスタムデータ属性<sup>36</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

<sup>32</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>33</sup> 「lowerCamelCase」のように複合語をひとつくりにして、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>34</sup> 「kebab-case」のように複合語をひとつくりにして、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>35</sup> 「ARIA - アクセシビリティ | MDN」  
<https://developer.mozilla.org/ja/docs/Web/Accessibility/ARIA>

<sup>36</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-*)

に使われたりする。Twitter や Facebook 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」

「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。それらを紹介させて。

まず JavaScript の文字ケースの挙動によるものの他に、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に関連付けを `<label>` 要素に設定するためのものね」

「はい、知ってます。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ないかないですね。でも 2 つだけだし、ちゃんとおぼえておきます」

「うん。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」

「そうそう。なお値が `true` のときは `<input type="checkbox" checked />` のように値の設定記述を省略できるよ。

あとは `value` ね。React では `<textarea>` と `<select>` も `value` 属性が持てるようになってるのが HTML と異なる」

```
<form>
  <textarea value="Fixed Text" />
  <select value="uranus">
```

```

    <option value="saturn">Saturn</option>
    <option value="uranus">Uranus</option>  { /* selected */ }
    <option value="neptune">Neptune</option>
    <option value="pluto">Pluto</option>
  </select>
</form>

```

「<textarea> は HTML なら子要素として値を持つようになってましたっけ。<select> はそもそも値を持ってないでしたよね。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」

「そう。これらは JavaScript でフォームを操作しやすいように挙動を改良したわけだね。直感的なので1回どこかで書いてみれば忘れないと思う。」

最後に、HTML には存在しないけど組み込みコンポーネントだけが持つ属性を2つ紹介しておくね。まず ref は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」

リスト 8: 02-jsx-usage/src/components/TextInput.tsx

```

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

  const handleClick = (): void => {
    if (inputRef.current) inputRef.current.focus();
  };

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

```

「……これは、うーん、えーっと……」

「ref 属性の中に任意のオブジェクトを設定しておくと、組み込みコンポーネントがリアル DOM としてレンダリングされた際に、渡されたオブジェクトの .current プロパティにそのリアル DOM への参照値を入れてくれるのね。だから handleClick() 関数の中で .focus() メソッドが実行できて

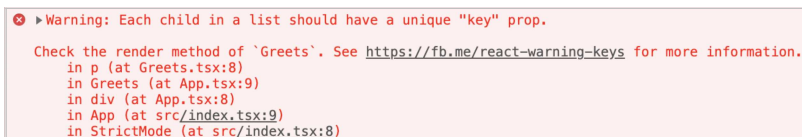
る実体は、レンダリングされた HTML DOM の `HTMLInputElement` オブジェクト<sup>37</sup>になる」

「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` という属性で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」

「なるほど」

「それから最後に `key` 属性。ところでこれまで使ってきたサンプルアプリだけど、ブラウザで実行させている状態から JavaScript コンソールを開いてみてくれる？」



```
Warning: Each child in a list should have a unique "key" prop.
Check the render method of `Greetings`. See https://fb.me/react-warning-keys for more information.
    in p (at Greetings.tsx:8)
    in Greetings (at App.tsx:9)
    in div (at App.tsx:8)
    in App (at src/index.tsx:9)
    in StrictMode (at src/index.tsx:8)
```

図 7: リストには “key” props が必要という警告メッセージ

「あれっ？ warning が出てますね。key が何とかがいわれてますけど」

「うん。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 はユニークな `key` 属性値を必要とするのね。使える値は文字列もしくは数値。key としての理想的な値は、そのコレクションの各要素が持つユニーク ID なんだけど、ここではそれが無いので key には繰り返しのインデックスを使うことにしよう」

リスト 9: Greetings.tsx の差分

```
<>
-   {...Array(times)].map((_) => (
-     <p>Hello, {name}! {children}</p>
+   {...Array(times)].map((_, i) => (
+     <p key={i}>Hello, {name}! {children}</p>
```

<sup>37</sup> 「HTMLInputElement - Web API | MDN」

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

```
    )}}  
  </>
```

「あ、warning が消えましたね！」

「うん。この `key` 属性は React が再レンダリングのための差分検出を効率的に行うのに必要とするもののね。だから実は、`key` にインデックスを使うのはユニークな値がない場合の最終手段。特に並べ替えがあるリストのときは再レンダリングに悪影響があるので、`key` にインデックスは使うべきじゃないと公式にもいわれてる<sup>38</sup>。たとえば外部 API から取得したコレクションデータとかをリスト表示する場合なら、そのデータが持つユニーク ID を `key` に使うのがいいね」

「なるほど、わかりました」

「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あらためて説明するけど、JSX がどういうものかわかってもらえた？」

「はい、だいたい。JSX は `ReactElement` オブジェクトを生成するシンタックスシュガーで、構文のベースはあくまで JavaScript であること。HTML の気分で書いた `<div>` とか `<p>` とかも、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、あたりですかね」

「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こう。JSX の開発効率の高さを実感できれば、デザインとロジックが混在してるのも気にならなくなると思うから」

---

<sup>38</sup> 「リストと `key` - React」 <https://ja.reactjs.org/docs/lists-and-keys.html#keys>

## 第 6 章 Linter とフォーマッタでコード美人に

### 6-1. ESLint

#### JavaScript、TypeScript における Linter の歴史

「React のコードを実際に書いていく前に導入しておきたいツールがあるの。linter とコードフォーマッタね。linter とはコードを静的解析してコンパイルではじかれない潜在的なバグを警告するもの、コードフォーマッタはインデントや改行などのスタイルを一律に自動整形してくれるものね」

「Ruby にも RuboCop<sup>39</sup> というのがあって、前のチームで使ってた」

「なら話は早いね。RuboCop は linter とコードフォーマッタを統合したツールだから。でも JavaScript や TypeScript ではこれらは別々のツールになってるので、まずは linter から導入していこう。

……と、その前に少し linter の歴史を説明しておくね。秋谷さんは『lint』って何か知ってる？」

「え？ だから構文チェックツールじゃないんですか？」

「まあそうなんだけどね。lint はそもそも C 言語のソースコードに対して、コンパイラよりも細かく厳密なチェックを行うプログラムとして開発され、初期の UNIX にコマンドとして装備されていた。その機能が、放っておくと故障の原因となる糸くずを片っぴしから絡めとる乾燥機の『lint trap（糸くずフィルター）』に似ていることから最初は『linttrap』と名付けるつもりだったんだけど、当時の UNIX では 4 文字以上の命令が使えなかったので『lint』と命名したんだって」

「へえ——」

「そこから転じて、C に限らず各種言語で書かれたコードを解析して構文チェックを行うことを『lint』と動詞化して表現されるようになり、さらにそのプログラムは『linter』と呼ばれるようになった。いっぽうオリジナルの lint は C のコンパイラが進化した結果、使われる機会が激減し、その派生である Splint も最後の更新が 2007 年で止まっていて、すでに今の Linux のパッケージには含まれなくなってる」

「豆知識ですね！」

「ところで JavaScript はその歴史的経緯から、容易にバグを生みかねない危険な仕様が当初のままたくさん残ってる話はしたよね。宣言されていない変数への代入がグローバル変数宣言になった

---

<sup>39</sup> <https://rubocop.org/>



り、`===` 等価演算子が暗黙の型変換を行ったりと無数の地雷が埋まってる。ES5 に導入された strict モード<sup>40</sup> を適用することでそれらのうちいくつかはフォローされるけど、それでもすべてをカバーするには全然不足してる」

「ふむふむ」

「そこで JSON 普及の立役者であり、名著『JavaScript: The Good Parts』<sup>41</sup> の著者でもある Douglas Crockford が開発し 2002 年にリリースしたのが JSLint<sup>42</sup> だった。『JavaScript: The Good Parts』に書かれてあるような JavaScript の『悪いパーツ』を使うことを禁じ、『よいパーツ』だけを使うように開発者に縛りをかけるツールで、その制約はかなり厳しいものだった。JavaScript に惚れ込んだ Crockford としては、この厳しい制限をかけることで JavaScript は言語として完成されるんだという固い理念があったんだろうね。その理念が一定の共感呼び、長年 JSLint が JavaScript における唯一の linter だったんだけど、適用ルールの on / off ができず Crockford 流のコーディングスタイルを一律強要されること、そのルールも固定化されて数年単位でアップデートされないことなどから開発者コミュニティとの間に確執が生まれるようになってしまったの」

「あらら。JavaScript への愛と『こうあるべき』という理念が強すぎたがゆえに、<sup>かたぐ</sup>頑 になっちゃったんですね」

「そういう状況にあって、2011 年に JSLint からフォークして JSHint<sup>43</sup> が作られた。JSHint ではどのルールを適用しどのルールを除外するかをカスタマイズするための豊富なオプションが用意され、それを設定ファイルに記述してプロジェクトに含めることで、開発チーム内でルールを共有できた。またデフォルトで適用されるルールも JSLint と比べてかなりゆるく、これらはそれまで開発者が JSLint に抱いていた不満をことごとく解消するものだった。JSHint は着実に支持を増やしていき、2013 年の暮れごろには JSLint を逆転するに至る」

「分家が本家を上回ったんですね。下剋上だ」

「いっぽうそれより少し前の 2013 年 6 月、ESLint<sup>44</sup> という新進気鋭の linter がリリースされてた。ESLint が作られたいちばんの動機は『開発者が独自の lint ルールを作れるようにすること』であり、

<sup>40</sup> 「Strict モード - JavaScript | MDN」

[https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Strict_mode)

<sup>41</sup> Douglas Crockford・水野貴明（訳）（2008）『JavaScript: The Good Parts —— 「よいパーツ」によるベストプラクティス』オライリー・ジャパン

<sup>42</sup> <https://jshint.com/>

<sup>43</sup> <https://jshint.com/>

<sup>44</sup> <https://eslint.org/>

その掲げた哲学は『Everything is pluggable』。つまり標準でバンドルされているものも含めてすべてのルールが本体から切り離されたプラグインとして平等に扱われ、それらは読み込んだ後からも個別に適用の可否やエラーレベルの調整が可能になってたの。ESLint 自体は拡張を前提とした中立なツールで特定の流儀でのコードの書き方を強制することがなかった。この思想は JSLint とは正反対とっていいね]

「へー。こうしてみると linter って、ずいぶん作者の思想性が強く出るソフトウェアなんですね」

「うん、そうなんだよね。そして ESLint はこうした拡張性に加え、充実したドキュメント、JSHint よりも読みやすくかつ柔軟に記述できる設定ファイル、出力はメッセージとともに抵触したルールの ID が表示されてしっかり根拠が説明されるという親切設計から、ユーザー数を急激に伸ばしていった。

さらにいっぽうで ESLint と同時期に JSCS<sup>45</sup> というツールも登場していた。こちらは『JavaScript Code Style checker』の名前のとおり、コードスタイルをチェックすることに注力したツールで、またチェックにひっかかった箇所を強制的に修正するコードフォーマッタの機能も備えてた。Airbnb JavaScript Style Guide や Crockford が提唱するスタイルなどを元にしたプリセットが公式から豊富に提供され、手軽に使うことができるのが魅力だった」

「ほうほう」

「JSCS は実際、大きな成功を収めたんだけどプロジェクトが大きくなるにつれて、その運営・保守の負担に耐えられなくなってしまった。そこで部分的に競合しつつも補完しあえるプロダクトであった ESLint に合流することを決め、2016 年 4 月にその声明を出すに至る<sup>46</sup>。コードスタイルチェッカー機能を統合して勢いづいた ESLint は一気に JSHint を追い抜いてシェアトップに躍り出て、そのままライバルたちに圧倒的な差をつけて現在に至るというわけ」

「なるほど。ESLint の拡張性の高さもさることながら、ユーザーとしても linter とコードスタイルチェッカーという機能が微妙に重なるツールはひとつに統合されてたほうが便利ですね」

「ここまでの JavaScript の linter の話。TypeScript の linter が実は別にあって、TSLint<sup>47</sup> というプロダクトが 2013 年にはすでに登場してたのね。最初は JSHint の TypeScript 版を目指してたようなんだけど、ほどなく ESLint を見習ってプラグイン形式でルールを追加できるようにしたりと拡張性を

---

<sup>45</sup> <https://jscs-dev.github.io/>

<sup>46</sup> 「JSCS—end of the line - Oleg Gaidarenko - Medium」  
<https://medium.com/@markelog/jscs-end-of-the-line-bc9bf0b3fdb2>

<sup>47</sup> <https://palantir.github.io/tslint/>

高め、ユーザー数を伸ばしていった。Angular が公式の linter として TSLint をフレームワークに組み込んだりね」

「ふむふむ」

「しかしリソース不足から来るドキュメントの不備や、ESLint にある各種周辺プラグインが TSLint ではなかなか対応されないという状況にユーザーが次第と不満を抱くようになっていった。それに加えてアーキテクチャ上の問題が発覚したことを理由に、Microsoft の TypeScript 開発チームが TSLint を見限って ESLint の TypeScript 対応プロジェクト始動を 2019 年 1 月に発表した<sup>48</sup>。それを受けて翌月、TSLint の作者は TSLint プロジェクトを非推奨にして ESLint への段階的な移行を促すアナウンスを出したの<sup>49</sup>」

「……おおお、ESLint がすべてを飲み込んでいきますね」

「そしていま現在の各種 linter のダウンロード数を比較したグラフがこれね」

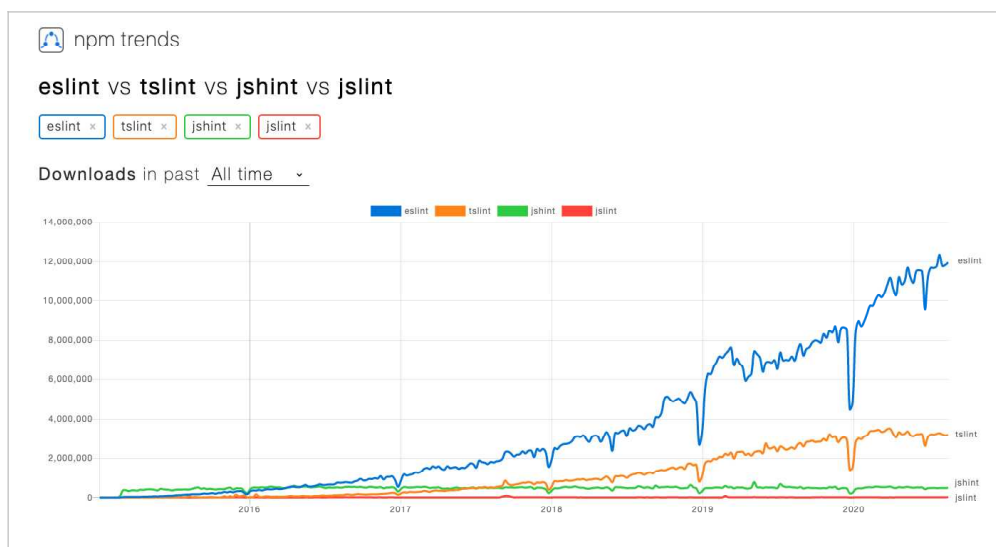


図 8: JS・TS における各種 linter の DL 数比較（2020 年 9 月現在）

「ふーむ、周辺ツールを飲み込んだ ESLint が無双してる感じですね。でも TSLint は非推奨になっ

<sup>48</sup> 「The future of TypeScript on ESLint - ESLint」 <https://eslint.org/blog/2019/01/future-typescript-eslint>

<sup>49</sup> 「TSLint in 2019 - Palantir Blog」 <https://medium.com/palantir/tslint-in-2019-1a144c2317a9>

たはずなのにそれほど減ってないような」

「まあユーザーもすぐには移行できないだろうね。Angular も 2020 年 9 月現在、TSLint から ESLint へ移行できてなくて、その目処も立ってないようだし」

「なるほど」

「はい、じゃあこれで JavaScript と TypeScript における linter の歴史については以上です。ツールの栄枯盛衰はあれども、linting あってこそ JavaScript は言語として完成される、モダン開発における JavaScript にとって linting は切り離せない言語の一部という思想は最初の JSLint から ESLint に至るまで変わらず受け継がれてると思う。それは TypeScript にあっても同様に、TSLint の作者も linting が TypeScript の DX（開発者体験）の中核にあるという認識だった。

ウチのチームでも、linter はモダンフロントエンド開発において必要不可欠なツールとして、各種プラグインを含めて積極的に使っていくから、そのつもりでね」

「わかりました！」

## ESLint の環境を作る

「じゃあ勉強のため、手動で既存のプロジェクトに ESLint を導入していこうか。ESLint の環境構築はけっこう複雑で、ありがちなのはチームの誰かが作ってくれた設定に継ぎ足し、継ぎ足しで秘伝のタレみたいになってしまって、腐臭を放ちつつ肥大化していくパターン。でも、ちゃんと基本から学んで一から自分で環境が作れるようになっておけば困らないから」

「腐臭を放つ秘伝のタレって……（笑）」

「『Hello, World!』のサンプルコード<sup>50</sup>をまるごとコピーきて、その上に構築していこう。ESLint については Create React App で生成したプロジェクトには ESLint パッケージが最初からインストールされてる。

別途最新の ESLint をインストールし直すこともできるんだけど、react-scripts とのバージョンの相性があるね。たとえば 2020 年 9 月現在、ESLint の最新メジャーバージョンである 7 系は react-scripts がサポートしていない<sup>51</sup>。6 系の最新版が入ってるはずなので、それをそのまま使わせて

---

<sup>50</sup> 第 1 章、第 2 節内の「Crate React App で『Hello, World!』」を参照のこと。

<https://github.com/oukayuka/Riakuto-StartingReact-ja3.0/tree/master/01-hello/02-hello-world>

<sup>51</sup> <https://github.com/facebook/create-react-app/issues/8977>

# 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ー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、専業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

## 【Ⅲ. React 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3版、三部作の第三部「React 応用編」。

React のルーティングと副作用処理を、歴史を踏まえつつ包括的に解説しています。Redux の新しい書き方や、宣言的にデータ取得ができる Suspense の情報も掲載。

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／211p／¥ 1,200)

### 《第三部 目次》

- 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 【II. React 基礎編】

---

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 1.0.7

著者 大岡由佳

印刷・製本 日光企画

---