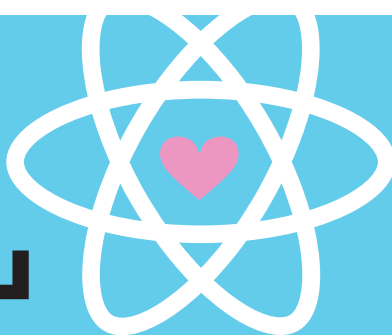
React Beginner's Book

いあクト!

TypeScriptで始めるつらくないReact開発

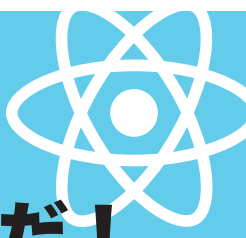
大岡由佳
YUKA O'OKA



「わぁいReact、
私Reactだーい好き」

（※中身はまじめな入門書です）

入社以来2年間、サーバサイド開発に携わっていた秋谷佳苗。彼女は社内の公募に志願し、ほぼ未経験ながらフロントエンドチームに参加、React でのお仕事を始めることになる。しかしその初日、待っていたのはリーダーである柴崎雪菜からの「1週間で戦力になってもらう」という言葉だった。彼女は果たして試練を乗り越え、一人前になることができるのか？



《対話形式》だからわかりやすい。
Reactってそういうことだったんだ！

初心者にも経験者にもオススメ！

りあクト！

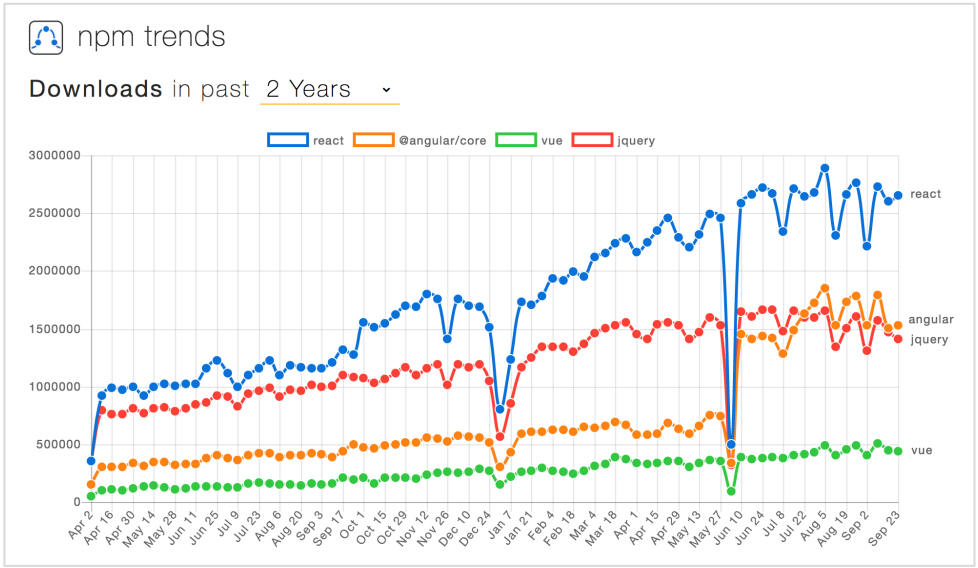
TypeScript で始めるつらくない React 開発

大岡由佳 著

まえがき

『りあクト！ TypeScript で始めるつらくない React 開発』をお買い上げいただき、まことにありがとうございます。一見してふざけたタイトルだと思われたかもしれませんが、内容は React を初めて学ぶ、学び始めたけれどなかなか理解が進まなかった、また途中で挫折してしまった、そんな人たちに向けたマジメな入門書です。

React は Facebook で開発され自社サービスに段階的に導入されていたものが、2015 年にオープンソースソフトとして公開された、Web アプリケーションフロントエンドの UI を構築するための JavaScript ライブラリです。その登場はたちまちフロントエンド開発者たちの注目を集め、あっという間にその分野でのデファクトスタンダードともいえる地位に上り詰めました。



npm のダウンロード数の統計では、翌年の 2016 年には早々に jQuery を抜き、2018 年 10 月の現在にいたるまでライバルたちを寄せ付けずダントツの 1 位となっています。そして Facebook や Instagram にとどまらず Twitter、Airbnb、Netflix、

Uber と軒並み有名どころのネットサービスを始め[1]、変わったところでは Nintendo Switch の eShop インターフェースにも React が使われるようになっています。また React の登場で他の多くのフロントエンドフレームワークが影響を受けて変遷し、そのため中には根本から作り直してほぼ別物になってしまったプロダクトもあるほどです。

その React 登場当時の衝撃と興奮を、残念ながら著者はリアルタイムでは体験していません。そのときはフロントエンドにあまり興味がなかったせいもありますが、まだドキュメントが少なく周辺情報も不足していて、かみくだいて解説してくれる人もあまりいない状況でしたので、著者の頭では「なんだか難しそう。JavaScript でそんなに複雑なことする必要あるの？」と思うだけでそのすごさが理解できなかったのです。

そうこうするうちに SPA（Single Page Application）の重要性の高まりとともに React の採用が拡大していき、さらには iOS や Android のネイティブアプリが開発できる ReactNative までが登場して盛り上がり、そのシーンが無視できなくなってきたため、重い腰を上げて学習を始めました。しかし React は最初の印象通りに著者にとってはかなり難しく、また「なぜこれをするのにこんなに回りくどいことを書く必要があるのだろう？」と疑問に思う点もいくつかあり、なかなか先に進めないでいました。

一方で、Qiita の記事などではガチ勢と思われる人たちが「React は簡単！」と書いているのをよく見かけたものでしたが、そんなときは文系出身エンジニアの私とは頭の出来が違うのだろうかと卑屈な考えが頭をもたげたりもしました。それでもなんとか React を使えるようになったのは、以前に Scala を学んでいて関数型プログラミングの考え方になじんでいたおかげも大きいと思います。その後、フロントエンドエンジニアとして 3 つの現場を経験してチームを統括したり、経験の浅い他のメンバーに React を教える立場になったりもしましたが、今ならなぜ当時の著者が「React は難しい」と感じたか、そしてガチ勢が「React は簡単」と自慢（？）のように言っていたのかがよくわかります。

---

[1] https://github.com/facebook/react/wiki/Sites-Using-React

Reactはごくシンプルな思想のもと、恐ろしく複雑なシステムでも破綻せずに信頼性の高いアプリケーションを作ることができる技術です。そのシンプルな思想を貫徹しようとして、ときには回りくどい書き方をユーザーに強要してきますが、それはとっつきやすさを犠牲にしてでもコードの秩序を維持しテスタビリティを確保するために必要なことと Reactの開発者たちが考えているためです。私は当初、その思想を理解しないまま徒手空拳で Reactの世界に飛び込んだために迷走したのでした。

「simple」と「easy」、日本語では混同して使われることもありますが、似て異なる概念です。現実の場においては両者が並び立つのが難しい場面が往々にしてあります。Reactとよく比較される Vueは「easy」なフレームワークと言えるでしょう。入門者のことを考慮して敷居が下げられていて、段階的に複雑なやり方にステップアップできるような作りになっています。しかしそのことは原理の一貫性に欠ける技術という側面を併せ持ちます。

いっぽうの Reactは「simple」なライブラリです。コンポーネント指向、単一方向データフロー、関数型プログラミングといった原理が一貫しており、それを守るために一見回りくどくコードの記述量が増えるやり方を要求されることも多々ありますが、Reactで開発したアプリケーションは非常に複雑なシステムでも見通しがよくて読みやすく、テストやデバッグがしやすく、メンテナンス性や拡張性が高いものになります。

ただその原理こそシンプルですが、高い理解度と応用力が求められるため、Reactは学習コストが低いとは決して言えません。またそれらの原理だけを抜き出して説明されても、初学者にはまったくピンとこないでしょう。でもそれを使う必要にせまられたとき、その背景にある思想やメリットを噛み砕いて説明してくれて、初学者が抱きがちな疑問を解消してくれる、そんな人が隣りにいれば Reactは難しくないはずなのです。現実にはなかなかそんな人に恵まれる機会はありませんが、著者がこの本を 2人のエンジニア、経験者と初学者による対話形式で書こうと思ったのは、こういった経緯があったためです。

あの理解が詰まっていたときにこの説明を誰かがしてくれていたら、すんなりわかったはずなのに。私自身のそんなありえない後悔を晴らすことができれば、これから Reactを学びたいと思っている人たちに喜ばれる本が書けるのではないかと考

えました。「React は難しくて勉強するのつらい」「Redux は回りくどくて必要性がわからない」「JSX がキモいから React に抵抗がある」etc.な方々にぜひ読んでいただきたい。そして React への（著者から見れば）不当な評価を覆してぜひ React を愛するようになってほしい、そんな目論見を抱きつつ本書を執筆しました。

対話ベースのペアプログラミングによる生産性の向上は数々の論文で報告されていますし、Ruby の作者まつもとゆきひろ氏も、テディベアプログラミング[2]のようにあえて誰かに説明する体裁をとることで開発者はより生産的になれると述べています。軽い気持ちから始めた対話体による執筆でしたが、こうして自分で読み直してみても、その試みは成功しているように思えます。（※自画自賛は著者の性格なので、ご容赦ください）

なお本書は入門書の体をとっていますが、基本的に Web アプリケーション開発経験のあるエンジニアだけれども React は未経験といった方が、可及的速やかにチームの開発に参加できるようにすることを目的として構成されており、内容をそのために必要な情報量に絞り込んでいます。ゼロから趣味のアプリを作り上げられるようになりたい、といった方には向いていないかもしれません。また文中に示される、特定技術や思想への見解・感想はあくまで著者個人の考えであることも併せてご理解願います。

サンプルコードが置いてある場所は
https://github.com/oukayuka/ReactBeginnersBook です。本文中で随時、個別のファイルを引用していますが、ページの都合上ところどころ、説明の本筋とは関係ない部分を断りなく省略してあるなど完全に同じではありませんので、ご承知おきください。

それでは React の深遠な森の中へ、以下の 2 人がご案内いたします。

---

[2] https://github.com/facebook/react/wiki/Sites-Using-React

登場人物

柴崎雪菜（しばさき・ゆきな）

とある会社のフロントエンドエンジニア。React 歴は 2 年半ほど。 1 人では手が回らなくなってきたため会社にフロントエンド開発担当の中途採用を促し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ず。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」と会社に要望を出していた。

秋谷香苗（あきや・かなえ）

柴咲と同じ会社の後輩。入社以来もっぱらサーバサイド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チームに参加する。柴崎から「1 週間で戦力になって」と言われ、マンツーマンで教えを請うことになるが……。

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**　16.5.3
- **React Router**　4.3.1
- **Redux**　4.0.0
- **TypeScript**　3.1.1
- **TypeScript FSA**　3.0.0β2
- **Create React App**　1.5.2（※プレリリース版の 2.0.0 でも動作確認済み）
- **react-scripts-ts**　3.1

目次

第 1 章 こんにちは React

1-1. プロローグ

「初めまして。本日からお世話になります、秋谷香苗です」

「はい、秋谷さんね。こちらこそよろしくお願いします。私はこのフロントエンドチームの、っていっても私と秋谷さんとの人だけなんだけど、リーダーの柴崎雪菜です」

「知ってます！ 柴咲さん、できる女性エンジニアって社内で有名ですし。今回、もちろん前から React に興味ああったのもあるんですけど、柴咲さんに教わっていっしょに働けるチャンスだっていうので、社内公募に応募したんです！」

「……そ、そう。ありがとう。やる気は十分ってことですね。じゃあ今から、私があなたに何を期待しているか、何をやってもらうかを説明していこうと思うけど、いいですか？」

「はい、よろしくお願いします！」

「ところで秋谷さんは入社何年目？ あと持ってるスキルについても教えてくれるかな」

「新卒で入社して 3 年目です。入社してからはずっと Rails での Web アプリ開発に携わってました。使える言語は Ruby と、ちょっとだけ JavaScript。触ってたのは jQuery を使ってゴニョゴニョするくらいですが。あと、業務では使ったことがありませんけど、入社前に Java を勉強してました」

「うん、わかりました。React については？」

「興味があったので自分で勉強しようとしたんですけど、途中で難しくて挫折しちゃいました……。Vue も触ってみたんですけど、こっちのほうが Rails と考え方が近くてわかりやすいな、と思いました。スイマセン……」

「いや、あやまることはないけども。そうだね、MVC の考え方になじんでる人は Vue のほうがとっつきやすいと思う。Vue の設計パターンは MVC に近い MVVM だし」

「MVVM？」

「Model、View、ViewModel からなる構成、といっても説明が長くなるので今は省くけど、HTML のテンプレートに処理結果を埋め込んでいくやり方が Rails や他のサーバサイド Web フレームワークと共通してるよね」

「はい、そう思いました」

「React の設計パターンはパラダイムが違うから、その思想を理解しないまま飛び込んでもなかなか身にならないんだよね。でもそれらは随時、説明していくので心配しないで」

「はい！ ありがとうございます！」

「あはは、いい返事だね。で、これから何をやってもらうかだけど。今から私がマンツーマンでついて、あなたに React 開発を叩き込みます。そうね、1 週間で私とペアプログラミングで開発に参加してもらえるレベルになってもらいますので」

「えっ、たったの 1 週間でですか？！ ムリムリ、実質 5 日間しかないじゃないですか？！」

「いや、それだけあれば十分でしょう。私、教えるのうまいので」

「……あはははは。不安だなぁ」

「ふふふっ、だーいじょうぶ。Rails は使いこなしてたんでしょう？ なら原理さえ理解してれば React は難しくないから」

「……わかりました！ 柴咲さんがそう言われるなら、覚悟を決めてがんばります！！」

1-2. 環境構築

「では、環境の構築からやってもらおうかな。まずは Node のインストール。やりかたは色々あって、Mac なら Homebrew で入れるのが簡単で手っ取り早いんだけど、私たちはプロだから、プロジェクトごとに違ったバージョンを共存させることが必要になることがあります」

「はい」

「なのでバージョンマネージャを使ってインストールしておくの。メジャーなのは **nvm** と **ndenv**。ウチでは **ndenv** を使ってるので、秋谷さんにもそれを入れてください」

「Ruby でも **rvm** と **rbenv** がありますね。前のチームでは **rvm** を使ってましたけど」

「私が **ndenv** を使うのは、ディレクトリごとに実行バージョンを使い分けられること、**anyenv** 経由でインストールすることで **rbenv**(Ruby)とか **pyenv**(Python)とかと設定をあるていど共通化できることとかがその理由」

「なるほど。じゃ、私もこれを機会に Ruby も **anyenv** から **rbenv** に移行しちゃいますね」

「『anyenv ndenv install』とかで検索すれば、インストール方法は見つかるから」

「だいじょうぶです！ すぐやります！」

ー5 分後 ー

「柴崎さん、できました！」

「おっ、早いね。じゃ、`node -v` と `npm -v` を実行してみて」

「……実行、と。それぞれ『v10.10.0』『6.4.1』が表示されました」

「うん、いいね。次は `npm install -g yarn` ね」

「あの、あまりわかってないんですけど、**Yarn** って何ですか？」

「まず **npm** は Rails で言うところの Gem と Bundler を引っくるめたものだね。Node モジュールパッケージの追加・削除に加えて各パッケージ間のバージョン整合とかを勝手にやってくれる。Yarn は Facebook 製の npm 改良版みたいなもの。高速だったりコマンドのタイピング数が少なかったりいろいろ使い勝手がいいので、React 界隈では npm より Yarn を使う人のほうが多いようで、ウチでも使ってるの。さっき打ってもらったのは、Yarn をグローバル環境にインストールするコマンドだね」

「へー、ほんとだ。どこからでも `yarn` コマンドが使えるようになりました」

「次はエディタね。秋谷さんは今、何のエディタ使ってる？」

「えーと、前のチームでは皆さん Vim の人が多かったので、なんとなく私も Vim を使ってました」

「あー、Rubyist は Vim 大好きだからねえ。でもウチでは **Visual Studio Code**、長いので略して『**VSCode**』って呼ぶけど、それを使ってもらいます」

「えっ、チームでエディタを指定されるんですか？」

「本当は各自好きなものを使っていいよと言いたいんだけど、これから入ってくる人も含めて VSCode を使ってもらおうかなって。理由のひとつは TypeScript と相性がよくて型の整合性チェックや Null 安全性のチェックとかを自動的にやってくれること。あとプラグインが豊富で、コードの自動整形とかも簡単にできる。

もうひとつの理由は、実際の開発は常にペアプログラミングでやってもらうんだけど、ひとつの画面をいっしょに 2 人見ながら開発する従来のやり方じゃなく、**Visual Studio Live Share** を使ってそれぞれ自分のマシンで同時コーディングをする予定だから」

「同時コーディング？」

「Google Docs で複数人のドキュメント同時編集はやったことあるよね？ Live Share を使うとあれと同じようなことがコーディングでできるようになるの」

「へー、今はそんなことができるようになってるんですね。近未来だー。わかりました、がんばって VSCode 使いになります！」

「Vim の操作に慣れてるなら、**VSCodeVim** っていう拡張を入れると、Vim と同じキーバインディングでコードが書けるようになるよ」

「あ、それ嬉しいです」

「あとで入れておくと役に立つ拡張のリストを送っておくから、必要に応じて入れてね。でも『※』マークがついてるのは今の段階で必須のものなので、必ずインストールしておいて。それから、私の VSCode 設定ファイルの `settings.json` も共有しておくから、自分の設定の参考にしたらいいよ」

「了解です！！」

柴崎さんオススメ VSCode 拡張リスト

- **Prettier※** …… コード自動整形ツール **Prettier** を **VSCode** に統合する。
- **TSLint※** …… TypeScript の静的コード解析ツール **TSLint** を **VSCode** に統合する。
- **stylelint※** …… CSS 用のリンター**stylelint** を **VSCode** に統合する。

- **VSCodeVim** …… VSCode 上で走る Vim エミュレータ。キーバインディングを Vim 形式に変更するにとどまらず、Undo/Redo 履歴や単語検索などの管理空間を独立させたり **VSCode 本体**とマージしたりもできる。
- **Bracket Pair Colorizer** …… マッチする括弧を色分けして教えてくれる。
- **vscode-icons** …… 左ペインの Explorer のファイルアイコンをバリエーション豊かにしてくれる。
- **VS Live Share** …… 複数人によるリアルタイムのコーディングコラボを実現する。

1-3. Hello, World!

「はい、じゃあ次はお約束の『Hellow, World!』をやってもらいましょうか」

「どんな技術もまずはそこからですね。でも React で新規のプロジェクトを作るときはどうするんですか？ Rails なら `rails new` だったり、Vue なら `vue create` ってコマンドが用意されてますけど」

「残念ながら React にはそんな便利なコマンドはありません。そもそも公式が「React はフレームワークではなく、単なる UI ライブラリだ」って言ってるくらいだし。一昔前なら、適当なボイラープレートを落としてきてカスタマイズするやり方が多かったんだけど、今は Facebook が **create-react-app** っていう、そのまんまな名前のコマンドモジュールを出してくれているので、それを使わせてもらいます」

「よかった。ちゃんとあるんですね」

「適当なディレクトリで `npx create-react-app hello-world ―scripts-version=react-scripts-ts` を実行してみて。あ、**npx** は npm のパッケージをいちいちインストールしなくてもそのバイナリが実行できるコマンドね」

「これって Rails の scaffold コマンドみたいなものでしょうか？ アプリの雛形っぽいものができましたけど」

「そう。次は作成されたディレクトリの中で `yarn start` を実行」

「えっ、勝手に Chrome が `localhost:3000` を開いた……」

「デフォルトだと Node サーバが 3000 番ポートで立ち上がって、普段使ってるブラウザがそこにアクセスするようになってるの」

「『To get started, edit 'src/App.tsx' and save to reload.』って画面に書いてあります」

「うん、でもちょっと待って。まずは `public/index.html` を見てみよう」

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initia
l-scale=1, shrink-to-fit=no">
    <meta name="theme-color" content="#000000">
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico"
>
    <title>React App</title>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
```

```
    </noscript>
    <div id="root"></div>
  </body>
</html>
```

「……HTML のガワだけですね。body の中身が `<div id="root"></div>` になってますけど、実体はどこにあるんでしょうか？」

「うん、次は `src/index.tsx` を見てみようか」

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
import App from './App';
import './index.css';

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

「さっきの div タグの ID が『root』でしたけど、
`document.getElementById('root')` があやしそうですね」

「カンがいいね。では最後に、さっき画面で編集しろと言われていた
`src/App.tsx` を開いてみて」

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

class App extends React.Component {
  public render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.tsx</code> and s
ave to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

「？？？？」

「まあ、いきなりじゃわかんないだろうね。順番に説明していくからしっかり聞いてて。

React で作られるアプリケーションは、すべて**コンポーネント**の組み合わせで構成されるの。コンポーネントというのは、今は『任意の HTML タグでくくられる単位』と考えていればいいかな。そして React ではコンポーネントは JavaScript で記述される。でも HTML タグの出力をいちいち JavaScript の関数で記述するのは読み

づらいから、JSX という JavaScript の中に HTML タグがそのまま書ける拡張記法を使うのよ。

`src/App.tsx` は JSX で `<App>` タグの中身を定義したファイルになるの。正確には TypeScript に拡張された JSX、つまり TSX だけど」

「えーっと、`App.tsx` で `<App>` タグの中身を定義してエクスポート、それを `index.tsx` でインポートしてるわけですね」

「そう、`import App from './App';` というのがそこだね。インポート時の拡張子は省略できるから。Create React App でプロジェクトを生成したときには **react-scripts**、TypeScript の場合は **react-scripts-ts** っていうコマンドモジュールがインストールされてるんだけど、これがエントリーファイルとして `src/index.tsx` を `public/index.html` に結びつけてくれてる。まあこのあたりは私もふだん忘れてるところなので、そういうもんなんだってくらいの理解でいいよ。そんなにいじらないとこだし」

「わかりました」

「じゃ、`src/App.tsx` をこんな感じに変更してから保存してみてくれる？」

```
          <header className="App-header">
            <img src={logo} classNam="App-logo" alt="logo" /
>
-           <h1 className="App-title">Welcome to React</h1>
+           <h1 className="App-title">こんにちは React</h1>
          </header>
          <p className="App-intro">
-           To get started, edit <code>src/App.tsx</code> and
 save to reload.
+           Hello, World !
          </p>
```

「あっ、勝手にブラウザがリロードされて、ページの内容が『Hellow, World!』に書き換わりました」

「Create React App で生成されたプロジェクトは、ホットリロードが有効になってるからね。まあ、まだいろいろ納得はいってないだろうけど、いちおう『Hello, World!』はこれで完成」

「そうですね。あまり自信はないですけど、とりあえず手順はおぼえましたし、雰囲気はわかったのでよしとします」

「うん。細かいところについては、おいおい理解していけばいいよ」

1-4. Yarn コマンド

「さっき生成したコードをリポジトリに上げたものがあるので[3]、ちょっと別のディレクトリに落としてきてくれる？」

「……はい、ローカルに展開しました」

「じゃ、このアプリを起動してみて」

「えっと `yarn start` でしたよね。実行……と。あれ？ エラーになりましたよ」

「うん、ディレクトリを見てみて。`node_modules/` がないでしょ。パッケージモジュールのインストールが必要なの。`yarn install` ってコマンドを実行しましょう」

「おおっ、なんかいっぱいダウンロードしてますね。 `yarn start` を実行したら、今度は `localhost:3000` にアプリが立ち上がりました」

「ちなみに `install` は省略できるので、ただ `yarn` と打っただけでも結果は同じね。Yarn は多機能なコマンドだけど、メインの役割はパッケージモジュールの管理なので、そこから説明していこう。とりあえずよく使うのはこれくらいかな」

- `yarn add PACKAGE_NAME` …… 任意のパッケージをインストールする
- `yarn remove PACKAGE_NAME` …… 任意のパッケージをアンインストールする
- `yarn upgrade PACKAGE_NAME` …… 任意のパッケージを最新バージョンに更新する

---

[3] https://github.com/oukayuka/ReactBeginnersBook/tree/master/01-hello/03-hello

- `yarn info PACKAGE_NAME` …… 任意のパッケージについての情報を表示する

「`yarn add` なんだけど、インストールする環境を認識する必要がある。開発のワークフローに必要だけどプロダクションに含む必要がないものは `yarn add -D PACKAGE_NAME` のように `-D` オプションを指定してあげるの。ちょっと `package.json` を開いてみて」

```
{
  "name": "hello-world",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "react": "^16.5.2",
    "react-dom": "^16.5.2",
    "react-scripts-ts": "3.1.0"
  },
  "scripts": {
    "start": "react-scripts-ts start",
    "build": "react-scripts-ts build",
    "test": "react-scripts-ts test ——env=jsdom",
    "eject": "react-scripts-ts eject"
  },
  "devDependencies": {
    "@types/jest": "^23.3.2",
    "@types/node": "^10.11.3",
    "@types/react": "^16.4.14",
    "@types/react-dom": "^16.0.8",
    "typescript": "^3.1.1"
```

```
    }
  }
```

「この devDependencies にエントリがあるパッケージが -D オプションでインストールされたものに相当するの。TypeScript のコンパイラや型情報は、ビルドしてすでに JavaScript にトランスパイルされたプロダクションには必要ないでしょ？ あとコードの静的構文チェッカーや自動整形ツール、デバッグのために必要なツールとかもここに入るわけ」

「なるほど」

「たとえばここではすでにインストールされてるけど、React のパッケージには TypeScript の方が含まれてなくて別で提供されてるから、一からインストールするときは yarn add react react-dom yarn add -D @types/react @types/react-dom のように別にコマンド実行する必要があるわけ。

ついでに **npm scripts** についても説明しておこうか。yarn help でコマンドの一覧が見られるけど、その中には start なんてコマンドはない。なのに実行できるのは package.json に "scripts" としてエントリが記述されてるからなの。yarn start を実行すると ./node_modules/.bin/react-scripts-ts start が実行される。パスの解決を自動で行ってくれるのでコマンドは直に書けるけどね」

「えー、そうだったんですか。じゃ、ここに自分でエントリを追加すれば、yarn コマンドで実行できるんですか？」

「その通り。後のほうのサンプルコードでは構文チェックのエントリとかが追加されてるよ。その他、ドキュメントやスタイルガイドを生成させたりとか、そうやって作った npm scripts を git の任意コマンドの実行に割り込んで起動させたりもできる」

「へえーー、便利ですね」

「それから、アプリを起動するときは専用のターミナルアプリじゃなく VSCode にターミナルの機能があるので、そっちを使ったほうがいいよ。エラーがあったとき、エラー表示を cmd ＋左クリックで該当ファイルが開くからね」

「なるほど、ありがとうございます！」

第 2 章 ナウでモダンな JavaScript

2-1. ECMAScript

「とりあえず『Hello, World!』はできたけど、React についてくわしく見ていく前に、言語について学んでおいてもらう必要がありそう。JavaScrpt はさわったことはあるんだよね？」

「あ、はい。Rails アプリの View 部分で、ちょっと凝った UI を実現するために jQuery を使ったりすることがありましたので」

「そっか。じゃあ ES5 以前の JavaScript しか知らないんだ」

「ES5?」

「正式には『ECMAScript 5』。ECMAScript は JavaScript の標準仕様で、最近はほぼ毎年のように更新されてるの。2018 年現在で策定中の最新版は ES2018、つまり『ECMAScript 2018』」

「ES5 は通し番号ぽいのに、ES2018 は年号なんですね。Windows や Office みたい」

「2015 年に公開された ES6 がイコール ES2015 で、以降はずっと年号でバージョンが表現されてるようね。で、ECMAScript には大きなパラダイム転換の時期があって、ES6 でモダンな仕様が一気に盛り込まれたわけ。だから ES5 までの JavaScript しか知らない人には、今の JavaScript は大げさに言って別の言語と思ってもらってもいいくらい」

「……うっ、そうなんですね」

「実際に使うのは TypeScript だけど、TypeScript は基本的に JavaScript に型リテラルを拡張しただけの言語であとの仕様は共通なので、まずは最新仕様の JavaScript を学んでもらいます。私が担当しているプロジェクトでは ES2018 をすでに使っているので、ES2018 ベースで話を進めていきます」

「はい、お願いします！」

2-2. 変数の宣言

「まずは変数の宣言から。これまで変数の宣言には var を使っていたと思うけど、これはもう金輪際、使ってはいけません」

「え、絶対にダメなんですか？」

「はい、私が許さないので。まず再代入の必要がない場合は const を、どうしても再代入の必要がある場合だけ let を使いましょう」

「わかりました。ところで var を使っちゃダメな理由を教えてください」

「うん。じゃ、このコードを見て。結果はどうなると思う？」

```
var n = 0;

if (true) {
  n = 50;
  console.log(n);
}

console.log(n);
```

「n = 50 を代入したところでリファレンスエラーになるんじゃないですか？」

「ふふふ、じゃ node コマンドで実行してみようか」

「えっ、50 が 2 回表示された……？」

「var で定義された変数のスコープってね、関数単位なんだよ。だから制御文のブロックをすり抜けて参照できてしまう。Ruby を始めとする大多数の言語の変数スコープはブロックだよね。その考え方に慣れたプログラマが var でコーディングしてると、うっかりバグを生んでしまいやすいでしょう。だから ES6 で一般的なブロックスコープの const と let が導入されたの」

「はー、なるほど。了解です。もう var は使いません！」

「よろしい。じゃ、次行ってみよう」

2-3. アロー関数

「秋谷さんは CoffeeScript 使ったことある？」

「はい、ちょっと前まで Rails のプロジェクトでは生の JavaScript じゃなく、標準添付の CoffeeScript を使うのが普通でしたから」

「みたいだねー。知ってるなら話が早い。ES6 で導入されたアロー関数リテラルというのがあるんだけど、実際に見てもらったほうが早いね」

```
// 従来の関数宣言
function plusOne(n) {
  return n + 1;
}

// アロー関数の宣言その 1
```

```
const plusOne = (n) => { return n + 1; };

// アロー関数の宣言その 2
const plusOne = n => n + 1;
```

「これ、3 つとも同じことなんですよね？ JavaScript のアロー関数って、本当に CoffeeScript の関数に似てますね」

「アロー関数式のその 1 とその 2 も比べてみて。引数が 1 つだけの場合はカッコが省略できるのと、本文が return 文だけのときは、return が省略できる」

「こっちのほうがスッキリ見やすくていいですね」

「うん、だから省略できるときは省略しましょう。あと、function での関数宣言とアロー関数式での宣言は this の挙動が変わるので気をつけて。次のコードを実行してみてください」

```
const obj1 = {
  num: 444,
  fn: function() {
    console.log(this.num);
  }
};

const obj2 = {
  num: 888,
  fn: () => {
    console.log(this.num);
```

```
  }
};

obj1.fn();    // 444
obj2.fn();    // undefined
```

「えっ……？ これどういうことですか？」

「funciton による関数宣言では、その文脈における直上のオブジェクトが this になる。直上に定義したオブジェクトがない場合はグローバルオブジェクトね。いっぽう、アロー関数式では this はその関数それ自体になるの」

「うーん、わかったようなわからないような……」

「この例だけではピンと来ないかもね。関数に引数として関数を受け渡したりすると、この違いのせいで思った挙動と異なることが出てきたりするんだけど、今の段階では知識として知っておいてくれればいいから」

「はーい」

「それから function での定義とアロー関数での定義、どちらの表現を使うべきかだけど。これは好みかなとも思うんだけど、this の挙動を統一する必要もあるし、どうしても仕方がないとき以外、極力アロー関数を使うようにしましょう。見た目もいかにも関数型っぽくてイケてるし」

「カッコよく書けたほうがモチベも上がりますしね」

「お、秋谷さんもそう思う？ ちなみにあとでコーディングルールをユーザーに強制するツールを紹介するんだけど、私のカスタマイズ設定ではアロー関数で書かないと怒られるようになってます」

「あはは、徹底してますね……」

2-4. クラス構文

「これは Ruby や Java をやってきた秋谷さんには簡単だよね。ES6 から JavaScript にもクラスが使えるようになりました」

「へー、そうなんですね。JavaScript は普通にクラス使えなくて、プロトタイプ？とかいう仕組みを使って擬似的にやる必要があるって聞いてました」

「ES5 まではそうだったね。とりあえずサンプルコードを見てもらおうか」

```
class Bird {
  constructor(name) {
    this.name = name;
  }

  chirp() {
    console.log(`${this.name}が鳴きました`);
  }

  static explain(name) {
    console.log(`${name}は翼があって卵を生みます`);
  }
}

class FlyableBird extends Bird {
  constructor(name) {
    super(name);
  }
```

```
  fly() {
    console.log(`${this.name}が飛びました`);
  }
}

const bd1 = new Bird('ペンギン');
bd1.chirp();              // ペンギンが鳴きました
Bird.explain('カラス');    // カラスは翼があって卵を生みます

const bd2 = new FlyableBird('スズメ');    // スズメが飛びました
bd2.fly();
```

「素直な構文ですね。これといって不明なところはなさそうです！」

「ふふ、さすがだねー。注意点としては、コンストラクタは継承されないので明示的に書く必要があることくらいかな。サンプルコードにはついでに『テンプレートリテラル』をさらっと差し込んでおいたんだけど、わかった？」

「文字列を通常のシングルやダブルのクォートじゃなくバッククォートで囲んだ上で、変数名を ${} で囲うと値が展開されるやつですよね。Ruby でも #{} という書き方があったんでだいじょうぶです」

2-5. 便利なオブジェクトリテラル

「配列やオブジェクトの展開構文。これも口で説明するより、コードで見てもらったほうが早そう」

```
const arr1 = ['A', 'B', 'C'];
const arr2 = [...arr1, 'D', 'E'];
console.log(arr2);    // [ 'A', 'B', 'C', 'D', 'E', 'F' ]

const obj1 = { a: 1, b: 2, c: 3 };
const obj2 = { ...obj1, d: 4, e: 5 };
console.log(obj2);    // { a: 1, b: 2, c: 3, d: 4, e: 5 }
```

「なるほどー、配列やオブジェクトの名前の前に **...** をつけることで中身が展開されるんですね。便利そう」

「この書き方を『**スプレッド構文**』と言って、**...** を**スプレッド演算子**と呼ぶの。ちなみに配列のスプレッド構文は ES6 から、オブジェクトのスプレッド構文は ES2018 から使えるようになったの。ウチで ES2018 を使ってるのは、この便利な構文が使えるのが大きいね。React でも多用される構文だからおぼえといて。

じゃ、次はこんな構文ね」

```
const foo = 65536;
const obj = { foo, bar: 4096 };
console.log(obj);    // { foo: 65536, bar: 4096 }
```

「obj の最初の要素、キーがないですね。これは変数 foo がオブジェクトの中でその変数名がオブジェクトキーに、変数値がその値になってるわけですか」

「その通り。これは『**プロパティ名のショートハンド**』とかって呼ばれる書き方。ES6 から導入された構文で、こちらも React でよく使われてるのを見かけるね」

「へぇー。こういった書き方、知らずに見てたら混乱したかもしれないですけど、見た目もスッキリするしいいですね。私も使いこなしていきたいです！」

2-6. 非同期処理を扱う

「JavaScript では時間のかかる処理は、ほぼ非同期が前提です。Rails だと API リクエストを行ってそこから得た値を加工して表示、みたいな処理は何の気なしに続けて書いてただろうけど、JavaScript で同じことをやろうとすると、リクエストの結果が返ってくる前に加工処理に進んで表示されてしまいます」

「えっ？ それ困ります」

「ちょっと次のコードを実行してみて」

```
const wakeUp = ms => {
  setTimeout(() => { console.log('起きた'); }, ms);
};

const greet = () => {
  console.log('おやすみ');
  wakeUp(2000);
  console.log('おはよう！');
}

greet();
```

「『おやすみ』と『おはよう！』が即座に表示されて、しばらくしてから「起きた」が表示されました」

「ここではわかりやすく `setTimeout()` を使ったけど、JavaScript では通信とかローカルファイルの読み込みとかの処理は、ほぼ断りなく非同期が前提になってるからね」

「えーっ、そうなんですかー？」

「React コンポーネントのレンダリングだってそうだよ。上の階層からとか関係なく、前処理が終わったものから他を待たずに表示されていくの」

「まず Controller でページ全体の処理を行って、そこからまとめて View に渡してレンダリングされる Rails とは全然ちがいますね」

「いいところに気がついたねー。その話題は後にとっておくとして、非同期処理のプロセスを待ってもらうやり方を見ていこう。まずは ES6 から導入された Promise 構文ね」

```
const delay = ms => new Promise(resolve => setTimeout(resolv
e, ms));

const greet = () => {
  console.log('おやすみ');

  delay(2000)
    .then(() => {
      console.log('起きた');
    })
    .then(() => {
      console.log('おはよう！');
    })
    .catch(err => {
      console.error('睡眠例外です: ', err);
    })
}

greet();
```

「今度は『おやすみ』が表示された後、しばらくして『起きた』『おはよう！』と表示されました」

「`delay()` 関数が **Promise** クラスオブジェクトを返してるよね。これを `then` のメソッドチェーンでつなぐことで、非同期処理をひとつひとつ処理が終わるのを待って順番に実行していくことができるようになるの」

「へえ———。最後の `catch` は例外の捕捉ですよね？」

「その通り。ES2018 からは、例外のあるなしにかかわらず最後にかならず実行される `finally` も使えるようになったよ」

「うーん、でもあまり見やすい構文じゃないですよね」

「JavaScript 独特の書き方だからね。Ruby に慣れた秋谷さんには、というより他の言語を学んできたたいがいのプログラマには直感的じゃないかも。だから ES2017 から **async/await** という構文が導入されたわけなんだけど、こっちのコードも見てくれる？」

```
const sleep = ms => new Promise(resolve => setTimeout(resolv
e, ms));

const greet = async () => {
  console.log('おやすみ');

  try {
    await sleep(2000);
    console.log('起きた');
    console.log('おはよう！');
  } catch (err) {
     console.error('睡眠例外です: ', err);
  }
}

greet();
```

「実行結果は同じですけど、なんかスッキリ見やすくなりましたね」

「`async` で定義された **Async 関数**は、本文中に `await` を前置きすることで、他の **Async 関数**の実行結果を待ってくれるようになるの」

「あれ？ でもここの `sleep()` は **Async 関数**じゃないのでは？」

「さすが目のつけどころがちがうねー。察しがよくてお姉さんは助かるよ。そう、**async/await** はじつのところ **Promise** 構文のシンタックスシュガーなの。**Async 関数**は隠されてるだけで暗黙のうちに **Promise** オブジェクトを返してるの」

「えっと、*『シンタックスシュガーとは、プログラミング言語の読み書きのしやすさのために、既に存在する構文に別の記法を与えたもの。糖衣構文などと訳される』*.....。なるほどなるほど」

「一般的なプログラマにはこっちの書き方のほうが直感的だし、コールバック関数が好きな人ってあまりいないんじゃないかな。余計な階層が増えるし。だから非同期処理はどうしても仕方がない場合を除いて **async/await** を使って書きましょう」

「わかりました！ こっちのほうが直感的というのは、私も同感です！」

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？

「はーい、次はお待ちかねの関数型プログラミング講座でーす」

「ううっ、敷居高そう……」

「まあまあ、そう難しく考えないで。React による開発ではオブジェクト指向の出番はあまりなくて、関数型プログラミングの考え方を求められます。これはもともと、HTML を構成する DOM のインタラクティブな操作という入り組んだ超複雑な難問に立ち向かうため、関数型のアプローチが最適だと Facebook の開発者たちが考えたから」

「へー、どうしてですか？」

「クラスから生成されたオブジェクトは内部に状態を抱えていて、それによって振る舞いが変わるでしょう。具体的にはメンバー関数の実行結果がメンバー変数に依存する。つまり副作用が大きくて予測がつきづらい」

「うーん、そういうもんですか？」

「特に非同期処理が関わってくると、とたんに話がややこしくなる。別のプロセスからの処理がそのオブジェクトの状態を変えてしまっていたら、想定したのと違う振る舞いが起きかねない」

「……それはそうですね」

「Rails では非同期処理をやることはあまりなかったかもしれないけど、前にも言ったように JavaScript は通信するのもローカルファイル読むのも基本全て非同期だからね」

「そうでした」

「だからクラスはどうしても使う必要があるところだけに限定しておいたほうがいいね。ところでここまで『関数型』って何気なく言ってきたけど、『関数型プログラミング』ってどういうことかわかる？」

「うーん、`map()` とか `reduce()` とかをよく使うイメージ？」

「へえ、なんとなくわかってるじゃない。関数っていうのは、数学の関数。『y = f(x)』ってアレね。数学の関数は、与えた引数に対して返り値は常に一定でしょう？ そういう同じ入力に対して同じ作用と同じ出力が保証されていることを「参照透過性」っていうんだけど、そんなふうに内部に状態を抱えず、副作用を極力排したプログラミングを行うことで、非同期な DOM の書き換えという難問に対処してるのが React なの」

「なるほどー。でもどうしても状態を抱えることが必要な場面ってありますよね？ たとえばユーザーのログイン状態とか。そういうのって関数型で処理できるんですか？」

「おっ、鋭いところをついてくるねー。それに関してはもっと後で説明することになるんだけど、ちょっとだけ触れておくと、React でよく用いられるのは状態を引数にとって新しい状態を返すやり方ね。そして別の引数に対して、返される状態の差分が常に一定であることが保証されればいいわけ。そうやって状態も外部データとして扱い、振る舞いを分離させて副作用を抑えるのが関数型のやり方と言えるかな」

「ん――――？」

「まあ具体的なコードがないとピンとこないだろうから、先に進みましょう。さっきあなたが挙げてくれた、`map()` や `reduce()` からね」

3-2. コレクションの反復処理

「コレクションの反復処理構文にいってみようか。これもまずサンプルコードを見てもらったほうがいいね」

```
const arr = [1, 2, 3, 4, 5, 6, 7, 8];

console.log(arr.map(n => n * 2));           // [ 2, 4, 6, 8,
10, 12, 14, 16 ]
console.log(arr.filter(n => n % 3 === 0));  // [ 3, 6 ]
console.log(arr.find(n => n > 4));          // 5
console.log(arr.every(n => n !== 0));       // true
console.log(arr.some(n => n > 8));          // false
console.log(arr.includes(5));               // true
console.log(arr.reduce((n, m) => n + m));   // 36
console.log(arr.sort((n, m) => n < m));     // [ 8, 7, 6, 5,
4, 3, 2, 1 ]
```

- `map()` は対象の配列の要素一つ一つを加工した新しい配列を作る
- `filter()` は条件に適合する要素だけを抽出して新しい配列を作る
- `find()` は条件に適合した要素をひとつだけ取り出す。見つからない場合は `undefind` を返す
- `every()` はすべての要素が条件を満たすかを真偽値で返す
- `some()` は条件を満たす要素がひとつでもあるかを真偽値で返す
- `includes()` は指定した要素が含まれるかを真偽値で返す
- `reduce()` は配列の要素を、与えた式で畳み込んだ値を返す

- `sort()` は各要素を、与えた条件によって並び替えた新しい配列を返す

「一気に説明するとこんな感じかな。だいじょうぶそう？」

「ところどころ Ruby と同じなので、だいたいわかります」

「よし。ちなみに `includes()` だけが ES2016 で、あとは ES5 から存在してるものね。これらが関数型プログラミングで多用されるのは、非破壊的な処理を行ってくれるから。`map()` や `filter()` はもとの配列の値を一切いじらずに、新しい配列を生成して返すよね。関数型プログラミングは副作用を嫌うと説明したと思うけど、だから相性がいいの」

3-3. 関数型プログラミングの概要

「ちょっと強引にスルーさせた気もするけど、`arr.map(n => n * 2)` ってちゃんと理解できてる？」

「えっ？」

「だって、これを書き換えるとこうなるんだよ」

```
const double = n => n * 2;
arr.map(double);
```

「関数の引数に他の関数を渡してる。秋谷さんにはなじみのないコードだと思うけど」

「……言われてみればそうですね。なんとなくわかってた気になってただけかも」

「さらに突っ込むと、この `const double = n => n * 2;` だって違和感あるはずだよ。だって変数に関数を代入してるんだから。そんなことしたことないでしょ？」

「ううっ」

「じゃ、そこも含めてあらためて確認していこう。関数型プログラミングでは、だいたい以下のようなことが普通に行われます」

1. 名前のない使い捨ての関数（無名関数）が使える
2. 変数に関数を代入できる（＝変数に無名関数を代入することで名前をつけられる）
3. 関数の引数に関数を渡したり、戻り値として関数を返すことができる（高階関数）
4. 関数に特定の引数を固定した新しい関数を作ることができる（部分適用）
5. 複数の高階関数を合成して 1 つの関数にできる

「さっきの `n => n * 2` が 1 でいう無名関数だね。これを `const double = n => n * 2;` のように変数に代入することができるというのが 2 で言ってること」

「ああ、だんだんわかってきました」

「よしよし。3 と 4 はあらためて次で説明してあげるよ。5 についてはもっと後じゃないと実例が示せないから、とりあえず棚上げで」