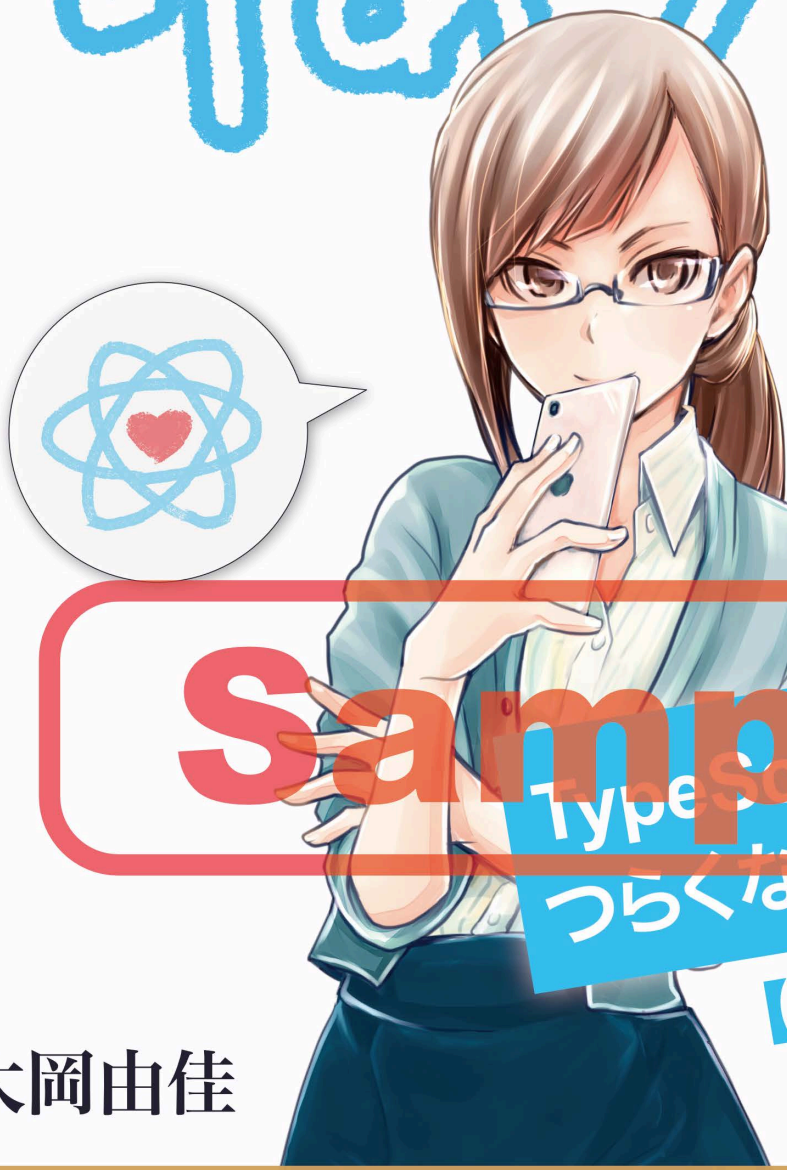
いまクト!

第3.1版



TypeScriptで始めるつらくないReact開発

【I. 言語・環境編】

大岡由佳

最新ES2020、TypeScript4.1に対応

仕事で使えるReact本はこれ！ モダンFE(フロントエンド)をやさしく解説

シリーズ累計 **1.2万部** 突破！

読者からの声

「会話形式で超わかりやすい！」

「この本でReactが怖くなくなりました」

前版 BOOTH売上 1位

入社以来1年半、Railsでの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
志願、Reactでの業務を始めることに。しかしその初日、リーダー
柴崎雪菜の「私がマンツーで教えるので1週間で戦力になってもらう」
と厳しい言葉で歓迎を受ける。秋谷ははたして無事に研修を終え、
一人前のReactエンジニアになることができるのか？

本作の構成（全三部作）

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

大岡由佳

くるみ割り書房

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。

本作は同人誌として初版が 2018 年 10 月に国内最大の技術同人誌即売イベント「技術書典 5」、その半年後の「技術書典 6」にて第 2 版が頒布されました。TypeScript による React の技術書が少ないのもあってか、その後のダウンロード販売も含めると初版が 1,000 部以上、第 2 版が 3,000 部近くと同人誌として異例の売り上げを記録。特に第 2 版は pixiv によるオンラインショップ BOOTH における技術書カテゴリー約 3,500 タイトル中、累計の売り上げ 1 位を獲得するに至りました。特に SIer や Web 系企業で働く現場のエンジニアの方々から大きな支持をいただいており、とある上場企業の新人教育用の教材に採用されたりもしています。

執筆の経緯、動機

React は Web フロントエンド分野において公開当初から技術革新を牽引し続けている特別な存在であり、世界においてそのシェアは並みいる競合たちを圧倒しています。しかし日本では React は多くのエンジニアから敬遠され、むしろ Vue.js が採用されるケースが多い傾向にあるようです。実際に求人サイトで検索しても国内の求人件数は Vue.js のほうが React より多く、同人誌でも商業誌でも Vue.js をテーマとして出版されているタイトル数は React のそれを大きく上回っています。日本は中国と並んで、React より Vue.js のほうが勢いがある世界でもめずらしい国です。

その理由は、日本の Web 開発コミュニティが今なおサーバサイドに偏重しているためだと筆者は考えています。各社の CTO や技術部門の責任者の顔ぶれを見てもサーバサイド畑の方が圧倒的です。だからテンプレートベースで Ruby on Rails のようなサーバサイドフレームワークになじんだ開発者が好みがちな Vue.js が採用されやすい。またフロントエンド開発の業務が十分に確立されておらず、他の分野のエンジニアが兼務で行うか、フロントエンド専任者がいてもその立場が弱く給与水準も低い傾向があります。だから JavaScript 力が低くてもだましだまし書けてしまう Vue.js が選

ばれるのでしょう。

しかし今日の主(おも)だったグローバルな Web サービスは React なしでは立ち行きません。Facebook や Instagram はもちろんのこと Twitter や TikTok、Pinterest のような数億人単位のユーザーを抱える SNS、さらには Slack、Discord、Netflix、Uber Eats、Airbnb といった有名どころのサービス。これらの Web インターフェースはすべて React 製です。これだけ錚々(そうそう)たる顔ぶれの企業に採用されている React が、日本の Web 開発シーンでは正当に評価されていないことに筆者はもどかしさを感じます。

React はたしかに学習曲線の初期の勾配がゆるい、つまり使いこなせるようになるまでに時間がかかる技術です。JavaScript がきちんと理解できていないとその入り口にすら立てませんし、バックボーンになっている関数型プログラミングの知識がなければ途中で挫折するかもしれません。しかしその原理はシンプルかつ一貫しており、React 固有のおぼえなければいけない決まりごとというのは、実は他のフレームワークと比べてもむしろ少ないといえます。ゆえに基礎から正しく順を追って学んでいけばトータルの学習コストはさほど高くなく、しかもそこで身につけた知識は他にも応用が利く汎用的なものが多いのです。

本作の目的のひとつは、頭ごなしに難しいと思われがちの React の学習の敷居を下げることにあります。頭から読んでいけば、正しい順番で必要な知識が身につくようになっています。現在は日本語にも訳されている React の公式ドキュメントはもちろんすばらしいものではありますが、各章を理解するために必要な他の技術知識はすでに身につけてるものとして話が進むので、初心者がいきなり読んですべてを理解するのは難しいでしょう。

本作の目的のもうひとつは、皆が React 開発においてきれいな設計ができ、きれいなコードが書けるようになることです。筆者はフリーランスとしてさまざまな現場へおもむき React アプリケーションの開発に参加してきましたが、そこで出会ったコードたちは正直なところ、そのほとんどがひどいものでした。そんな状況が蔓延していたのには開発メンバーにじゅうぶんな基礎知識がないのもあったと思いますが、それよりも React が持つ思想や哲学への理解が浅いことが問題に感じました。ネットでたまたま見つけた記事のやり方を鵜呑みにしたり、素性のあやしい技術顧問のいうがまま偏った設計になっていたりするのは、それが原因でしょう。ひどいコードは関わる人の魂を濁らせる。一度生まれたひどいコードをきれいにするには、最初からきれいなコードを書くより何倍もの労力が必要になります。「React 開発におけるすべてのひどいコードを、生まれる前に消し去りたい」というのが、本作執筆のもっとも大きな動機のひとつです。

スタイル

読者がうわべだけの理解に終わらず、ちゃんと技術の本質を理解できるようになるにはどうすればいいだろう。執筆にあたってまずそれを考えました。そうして本作は、他の技術書ではなかなか見かけない 2 つの特徴を備えるに至りました。

まずひとつめ。本作はシニアエンジニアが新人エンジニアにマンツーマンで React を教えていくという、2 人の会話文によってすべての解説がなされています。毎回の説明において新人エンジニアに質問させることで、初学者が抱きがちな疑問点を根こそぎ洗い出し、その都度解消させていこうという目論見の下にこのスタイルを採用しました。筆者の経験談ですが、React を学んでいると「なぜこんな回りくどいことをする必要があるんだろう」とか「しれっと出てきたこの技術にどういう必然性があるのか」という思いが頭をよぎることがよくあります。普通の技術ドキュメントや技術書では、それらの疑問には答えてくれずそのまま話を先に進められてしまいます。しかし本作では、そこに新人エンジニアが「待った！」をかけて、納得がいくまでシニアエンジニアに説明を求める。冗長かもしれませんが、実際に前版まではこの形式が読みやすくわかりやすいと好評でした。今版では、さらに遠慮なく新人エンジニアが疑問をぶつけるようになっています。

2 つめの特徴として本作では、各技術の概要と使い方を紹介するだけにとどまらず、その歴史と思想にまで深く踏み込んで解説しています。これは動機のところでもふれたように、その技術を使ってきれいに設計し、きれいなコードを書くために必要なことだからです。なぜその技術が登場するに至ったかという背景や、その技術が他のどの技術からどんな影響を受けたか、作者はどういう動機から何を理想としてそのプロダクトを開発しているのか。それらを理解して初めて正しく使えるようになると筆者は考えます。

React は Angular のようなフルスタックのフレームワークではなく、それなりの規模のアプリケーションを開発するためにはサードパーティのライブラリをいくつも組み合わせる必要があります。それらにも流行り廃(はや すた)りがあり、選定に失敗するとそれが後に大きな技術的負債となってしまいかねません。このことは React のエコシステムを豊かにし、技術革新を推進する原動力にもなっているのですが、初心者には React を採用するハードルになってしまっていることも確かです。「技術選定

の審美眼（by 和田卓人氏）[1]」を磨くには、歴史およびそのバックグラウンドになっている思想のトレンドを知る必要があります。本作を通して読むことで、新しい技術が出てきたときもそれが筋がよくて今後普及していくものなのかどうか、読者自身が判断できるようになっているはずです。

対象読者

本作は何よりも「**仕事で使える React 本**」であることを目指しました。もっともマッチする読者像は、登場人物の新人エンジニアがそうであるように、業務での Web アプリケーション開発の経験はありながら React 自体は初心者という職業エンジニアの方です。そういう方が可及的速やかにチームでの開発に参加できるようにすることを目的として、本作は書かれています。またサーバサイドエンジニアの方、Vue.js など他のフロントエンドフレームワークを経験済みの方などにとっても有用な内容になっています。

プログラミング未経験者や業務経験のない趣味の初心者プログラマの方には、難易度が高かったり内容そのものが向いていないかもしれません。JavaScript のために章をひとつ割いていますが、モダンフロントエンド開発に必要な部分の説明であり、基礎知識については省略しています。JavaScript 自体がまったくの初心者の方は、すぐれた入門書の『JavaScript Primer』[2] を先に読むことをおすすめします。なお TypeScript については知らなくても、本書で紹介されている内容のみで入門できておよそ中級レベルまでは習得可能です。

また本作は、React 開発の中級者以上の方にも楽しみながら読んでいただける内容になっています。React や周辺技術のくわしい歴史については、よほど初期から力を入れて追いかけていないとご存じない情報でしょう。Redux Style Guide の各ルールについての解説や、2020 年 12 月現在においてまだ安定版では公式サポートのない Suspense for data fetching や Concurrent モードについての知見などは読み応えがあると思います。

---

[1] 「技術選定の審美眼 / Understanding the Spiral of Technologies. 」
https://speakerdeck.com/twada/understanding-the-spiral-of-technologies

[2] azu・Suguru Inatomi（2020）『JavaScript Primer 迷わないための入門書』KADOKAWA
https://www.kadokawa.co.jp/product/302003000984/

本作の構成

本作は三部作となっており、本書はその第一部「Ⅰ. 言語・環境編」です。第一部では React 公式のプロジェクト作成ツールである Create React App の使い方から始まり、モダンフロントエンド開発に必要な最新の JavaScript、関数型プログラミング、そして TypeScript の知識までを学ぶことができます。第二部「Ⅱ. React 基礎編」では React の歴史と思想を紹介し、コンポーネントとそれを記述する JSX、それに Hooks までをカバー。最後の第三部「Ⅲ. React 応用編」では React における副作用処理を取り扱い、Redux や最新の Suspense について解説しています。

全 13 章構成で、基本的に以前の章で学んだことを前提に話が進むため、TypeScript による React 開発の入門者および初心者の方は第一部から順を追って読み進めることをおすすめします。基礎がじゅうぶん理解できている中級者以上の方は、手っ取り早く知りたい内容の章だけをかいつまんで読んでもいいでしょう。

また本文の内容についてはできるだけ、出典や参考資料へのポインタを脚注としてつけてあります。随時、ご参照ください。

サンプルコードについて

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

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

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

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

本文および上記リポジトリに掲載しているコードは、読者のプログラムやドキュメントに使用し

てかまいません。コードの大部分を転載する場合を除き、筆者に許可を求める必要はありません。出典を明記する必要はありませんが、そうしていただければありがたいです。

本書について

登場人物

柴崎雪菜（しばさき・ゆきな）

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

秋谷香苗（あきや・かなえ）

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

前版との差分および正誤表

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

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

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

本文中で使用している主なソフトウェアのバージョン

- React（`react`） …… 17.0.1
- React DOM（`react-dom`） …… 17.0.1
- Create React App（`create-react-app`） …… 4.0.1
- TypeScript（`typescript`） …… 4.1.3
- ESLint（`eslint`） …… 7.14.0

目次

第二部「II. React 基礎編」目次

第 5 章　JSX で UI を表現する

第 6 章　Linter とフォーマッタでコード美人に

第 7 章　React をめぐるフロントエンドの歴史

第 8 章　何はなくともコンポーネント

第 9 章　Hooks、関数コンポーネントの合体強化パーツ

第三部「III. React 応用編」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

プロローグ

「おはようございます、柴崎さん。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、……といっても今のところは私と秋谷さんの 2 人だけなんだけど、リーダーの柴崎雪菜です」

「柴崎さんのこと私、前から知ってましたよ。柴崎さん、ウチに転職されてきて間もないけど凄腕の女性エンジニアって社内でウワサになってましたし。今回、もちろん前から React に興味があったのもあるんですけど、柴崎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、この社内公募に手を上げたんです！」

「……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、私が秋谷さんに何を期待しているかとか、今後やってもらう予定のこと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！」

「その前に秋谷さんは今、入社何年め？　あと持ってるスキルについても教えてくれるかな」

「新卒で入社して今年で 2 年目です。入社してから 1 年ちょっと 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、それにちょっとだけ JavaScript を。さわってたのは jQuery を使ってちょっとした効果を加えるくらいですけど。あと、業務では使ったことがありませんが、入社前に Java を学んでました」

「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[3] もさわ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、と思っちゃいました。すみません……」

「いや、あやまることはないけれども。そうだね、サーバサイドでよくある MVC[4] フレームワークになじんでる人は Vue.js 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

[3] https://jp.vuejs.org/

[4] Model-View-Controller。UI を持つアプリケーションソフトウェアを実装するためのデザインパターンで、システムを機能別に Model（モデル）、View（ビュー）、Controller（コントローラ）の 3 つの要素に分割して設計する。Ruby on Rails を始めとする多くの Web アプリケーションフレームワークで採用されている。

MVVM だし」

「えむぶいぶいえむ？」

「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略するけど、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、私もそう思いました」

「React の設計パターンはそもそもパラダイムが異なるので、その思想を理解しないまま飛び込んでもなかなか身につかないんだよね。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 開発の基本を叩き込みます。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たいかな」

「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」

「……えええええ？　不安だなぁ」

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

「……わかりました！　柴崎さんがそうおっしゃるなら、覚悟を決めてがんばります！！」

第 1 章　こんにちは React

1-1.　基本環境の構築

Node.js がなぜフロントエンド開発に必要なのか

「ではまず基本的な環境の構築からやってもらうんだけど、いちばん最初に入れなきゃいけないのが Node.js[5] ね」

「Node.js ってよく聞きますけど、それが何なのか実はよくわかってません……。最初に教えておいてもらっていいですか？」

「そっか、ネットで『Node.js とは』って検索しても、あんまり初心者が納得できるような説明がないからね。じゃ、まずそこから始めていこうか。JavaScript って本来はブラウザ上で動かすために作られた言語であって、そのままだとブラウザでしか動かないのは秋谷さんも知ってるよね？」

「はい、もちろん」

「Node.js は簡単にいうと、JavaScript を Ruby や Python と同じように秋谷さんの PC のターミナル上で動かすことができるようにするためのソフトウェアなのね。中の言語処理エンジンは Google Chrome 用に作られた V8[6] を流用してて、そこにローカルマシンで動かすためのファイルやネットワークの入出力機能とかが追加されてる。だから Node.js を使えば、JavaScript を Ruby や Python みたいにサーバサイド言語として使えるようになるわけ」

「……なるほど。でも疑問があります。私たちがやろうとしてるのはフロントエンド開発ですよね。だったらブラウザだけでしか動かないのって、別に困らなくないですか？」

「そうだねえ、じゃあブラウザだけで React を動かしてみようか。この HTML ファイルをブラウザにドラッグ&ドロップして読み込ませてみて」

リスト 1: 01-env/browser-only.html

---

[5] https://nodejs.org/ja/

[6] Google が開発するオープンソースの JavaScript エンジンであり、JIT コンパイル（ソフトウェア実行時にコードのコンパルを行う）を介して動作する仮想マシンの形を取る。Google Chrome や Microsoft Edge といった Chromium ベースのブラウザ、Node.js や Deno などのサーバサイド JavaScript ランタイムに採用されている。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>React Test</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="https://unpkg.com/react@17/umd/react.development.js" crossorigin></script>
        <script  src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
crossorigin></script>
    <script type="text/javascript">
ReactDOM.render(
  React.createElement("h1", null, "React works!"),
  document.getElementById('root')
);
</script>
  </body>
</html>
```

「『React works!』って表示されました。やっぱり React もちゃんとブラウザだけで動くんじゃないですか」

「うん、動くには動くんだけどね。コードをよく見てみて。React のライブラリをただのスクリプトファイルとして読み込んでるでしょ。2 つだけならまだこの書き方でも間に合うよ。でもそれなりの規模のアプリケーションを作ろうとすると、サードパーティのライブラリをいくつも読み込んでくる必要があって、それらは相互に特定のバージョンで依存しあってたりするわけ。それらをただのスクリプトファイルとして読み込むなら、秋谷さんはライブラリ間の依存関係を手で解決して、ライブラリに新しいバージョンが出てアップデートするときにもそれをいちいち最初からやり直す必要がある。それやりたい？」

「パッケージのインストールと整合性の管理の問題ですか……。それって Ruby の gem[7] や Bundler[8] がやってくれてるようなものですよね」

[7] 正確には「RubyGems」で、Ruby 標準のパッケージ管理システム。そのライブラリ群が「gem」と呼ばれる。またそのコマンドも `gem` である。https://rubygems.org/

[8] gem 同士の依存関係とバージョンの整合性を管理してくれるツール。https://bundler.io/

「そうだね。JavaScript でそれを担ってるのが『npm[9]』なの。npm はもともと Node.js のためのパッケージ管理システムだったんだけど、フロントエンド用のパッケージを提供するのにも用いられるようになって、今ではむしろそっちの用途のほうがずっと多いくらい。昔は Web アプリケーション専用のパッケージ管理システム[10]があったんだけど、周辺ツールの充実で npm パッケージの多くがフロントエンドでもそのまま実行可能になったために廃れちゃったのよ。だから JavaScript では、サーバサイド開発でもフロントエンド開発でも、パッケージ管理に npm を使うようになってる」

「へー、そんな過去があったんですね」

「フロントエンド開発に Node が必要な理由は他にもいくつかある。主なものを挙げるとこんな感じかな」

- パフォーマンス最適化のために JavaScript や CSS ファイルを少数のファイルにまとめる（バンドル）
- 最新版の機能を使用している JavaScript のコードを、ブラウザ実行時に polyfill[11] するのではなく最初からコンパイルしておく
- 開発中、ブラウザにローカルファイルを直接読み込ませるのではなく、ローカル環境で開発用の HTTP サーバを起動してそれ経由でアプリケーションを稼働させる
- テストツールを用いてユニットテストや E2E テスト[12]を記述する
- ソフトウェアテストやコードの構文解析をローカル環境で実行する

「後半はまあ、秋谷さんが前のチームの Rails 開発でやってたようなモダンな DX[13] を、フロントエ

---

[9] Node.js のために作られたパッケージ管理システム。名前は「Node Package Manager」から。パッケージのインストール、アップデート、アンインストールだけでなく、リポジトリ機能を備えていて開発者が自身の開発したパッケージを npm で公開することもできる。2020 年現在において、提供するパッケージ数は 120 万以上と世界最大で、RubyGems や Maven Central（Java）など他の言語のものと比べて数倍の差をつけている。https://www.npmjs.com/

[10] https://bower.io/

[11] 最近の機能をサポートしていない古いブラウザでも、その機能を使えるよう古い仕様に合わせてコードを自動的に変換すること。

[12] 「End to End」テストの意。「端から端まで」の言葉通り、Web アプリケーションにおいては実際のユーザーが行うようにブラウザを操作して、期待通りの挙動となるかをシナリオに沿って確認するテストのこと。

[13] 「Developer Experience（開発者体験）」の略。そのシステムや環境、文化などが開発者に与える体験。さらに推し進めて、それがどれだけストレスなく快適に開発・保守できるものかというニュアンスが含まれることも。

ンド開発でも実現するためのものだね。いま挙げたものを実現するためのツールはすべて Node.js 上で動くようになってるの。はい、以上がフロントエンド開発で Node.js をインストールしておくことが必要な理由です」

「なるほど、かなり疑問点が解消されました。ありがとうございます！」

「よかった。じゃあこれから実際に Node.js をインストールしていこうか」

Node.js をインストールする

「Node のインストールのやりかたは色々あって Mac なら Homebrew[14]、Windows なら winget[15] や Chocolatey[16] やといったパッケージ管理システムに Node.js があるので、それらを使って入れるのが簡単で手っ取り早い。でも私たちはアプリケーション開発のプロなので、プロジェクトごとに異なるバージョンの環境を共存させるのが必要になることがあるよね」

「そうですね」

「だからバージョンマネージャを使って Node.js をインストールしましょう。この分野でメジャーなのは **nvm**[17] と **nodenv**[18] あたり。私は nodenv を使ってるので、それに合わせて秋谷さんにも nodenv を入れてもらおうかな」

「Ruby でも同じようなものに RVM[19] と rbenv[20] がありますね。前のチームでは RVM を使ってましたけど」

---

[14] Max Howell が中心となって開発している macOS 用のパッケージ管理システム。Debian 系 Linux に搭載されている APT のようにバイナリを配布するのではなく、都度ビルドを行う。ただしすべてのソースをビルドするのではなく、バイナリがあるものは Bottle というバイナリパッケージをインストールする。home-brew とは英語で自家醸造酒（ビール）を意味し、「ユーザーが自らパッケージをビルドして使用する」ことのメタファーとなっている。https://brew.sh/

[15] Microsoft が開発しているパッケージ管理システム。コマンドラインからアプリケーションのインストールを管理できる。2020 年 12 月現在、プレビュー版であり、登録されているアプリの数もまだ少ない。
https://docs.microsoft.com/ja-jp/windows/package-manager/winget/

[16] Chocolatey Software 社製の Windows 用のパッケージ管理システム。6,000 以上のパッケージが登録されており、GUI でも管理できる。使用には管理者権限が必要。https://chocolatey.org/

[17] https://github.com/nvm-sh/nvm

[18] https://github.com/nodenv/nodenv

[19] https://rvm.io/

[20] https://github.com/rbenv/rbenv

「私が nodenv を使ってる理由は、**anyenv**　経由でインストールすることによって rbenv とか pyenv[21] とか他の言語もまとめて管理できて、それらの設定も共通化できるからだね」
「なるほど。じゃ私もこれを機会に anyenv を入れて、Ruby の管理も RVM から rbenv に移行しちゃいますね」
「うん。私も秋谷さんも Mac だからインストールは簡単だよ。Homebrew がすでに入っていればこんな感じで OK」

```
$ brew install anyenv
$ echo 'eval "$(anyenv init -)"' >> ~/.zshrc
$ exec $SHELL -l
$ anyenv install nodenv
$ exec $SHELL -l
```

「ほんとに簡単ですね」
「`>> ~/.zshrc` の部分は使ってるシェルによって適宜変更してね。macOS は Catalina から zsh が標準になったのでここでは zsh を前提にしてるけど」
「私も zsh ユーザーなのでだいじょうぶです。ちなみに Windows の場合はどうするのがいいんですか？」
「nodist[22]　という nodenv にインスパイアされた Windows 用のバージョンマネージャがあるんだけど、あんまりおすすめできないかな。そもそも Web アプリケーション開発は UNIX 環境でやるのが当たり前なのに、初心者ほど Windows 環境で開発しようとしてハマるんだよね」
「うちの会社も開発スタッフが使ってるのは全員 Mac ですし、社外でも Ruby の勉強会とかではやっぱりほとんどの人が Mac で、たまにデスクトップ Linux の人を見かけるくらいですね。私も最初、なんでエンジニアってこんなに Mac ばっかりなんだろうって思ってました。やっぱりフロントエンド開発でも Windows はやめておいたほうがいいんでしょうか？」
「いやそれがね、最近の Windows にはコマンドラインベースの Linux をネイティブ実行できる

[21] https://github.com/pyenv/pyenv
[22] https://github.com/nullivex/nodist

WSL[23] というしくみがあって、それを使えば Mac とも遜色ない環境で Web アプリケーション開発ができるようになってるんだよ。Windows の GUI の上で本物の Ubuntu とかが動くわけだからね。初期の WSL はファイルの読み書きが遅いという問題もあったけど、WSL2 になってそれもほぼ解消されて普通に使えるようになった」

「へー、じゃあ今は Windows でも全然だいじょうぶなんですね」

「WSL2 がリリースされたのが 2019 年なので、ほぼ直近の話だけどね。ただ常に 2 つの環境を意識しないといけないし Linux の知識も必要になるので、おいそれと初心者には勧められないかな。初心者はおとなしくメジャーな Mac を使ってほしい」

「まあ、たしかにそうですね」

「ちなみに秋谷さんが万一 Windows ユーザーだったときのために Windows 用の環境作成手順のドキュメントも作っておいたんだけど[24]、これは必要なかったね」

「えっ、そんな用意もしていただいてたんですか……。じーん。ありがとうございます！」

「じゃあさっきの続きね。anyenv と nodenv 用のプラグインを入れておこう。nodenv を含めた『ほにゃらら env』をまとめてアップデートをしてくれる anyenv プラグインの anyenv-update と、npm インストール時にデフォルトでいっしょにインストールしておくパッケージを指定できる nodenv プラグインの nodenv-default-packages ね」

```
$ mkdir -p $(anyenv root)/plugins
$ git clone https://github.com/znz/anyenv-update.git $(anyenv root)/plugins/anyenv-update
$ mkdir -p "$(nodenv root)"/plugins
$ git clone https://github.com/nodenv/nodenv-default-packages.git "$(nodenv root)/plugins/nodenv-default-packages"
$ touch $(nodenv root)/default-packages
```

「この `default-packages` ファイルの中身はこうしておいて。各パッケージについてはおいおい説明していくから」

リスト 2: default-packages

---

[23] Windows Subsystem for Linux。Microsoft が提供する、Linux のバイナリ実行ファイルを Windows 上でネイティブ実行するための互換レイヤー。WSL2 からは仮想マシン上で本物の Linux カーネルが動作するようになっているが、VirturalBox のように隔離したマシンリソースの割り当てを必要とせず、わずか数秒で起動する。Microsoft Store では WSL で動作する、Ubuntu を始めとする Linux ディストリビューションがいくつも配布されている。
https://docs.microsoft.com/ja-jp/windows/wsl/

[24] https://github.com/oukayuka/Riakuto-StartingReact-ja3.1/extra/build-win-env.md

```
yarn
typescript
ts-node
typesync
```

「これで準備はできたので、いよいよ実際に Node.js をインストールするよ。まずインストール可能なバージョンのリストを取得して、そこから最新のバージョンを見つけ、それを指定してインストールする」

```
$ nodenv install -l
$ nodenv install 14.4.0
$ nodenv global 14.4.0
```

「Node では JavaScript の挙動がバージョン 14.4 以前と 14.5 以降では異なるところがあって、最初に学ぶ段階では以前の挙動のほうがわかりやすいと思う。だから今回は最新版をインストールするときでもそれに加えて 14.4.0 もインストールしておいてね」

「最後の `nodenv global <バージョン番号>` というのは何をやってるんですか？」

「この先、nodenv で複数のバージョンの Node.js をインストールすることになるだろうけど、インストールされているバージョンの中でどれをデフォルトにするかを指定してるんだよ。バージョンを新しくしようとインストールしても、こうやってバージョンを設定し直さないと使ってるのは旧バージョンのままになってしまうので気をつけてね」

「そういえば RVM にも `rvm --default use` って同じようなコマンドがありました。うっかり忘れないようにします！」

「`nodenv local <バージョン番号>` というコマンドもあって、こっちは任意のディレクトリ配下で使う Node のバージョンを指定するものね。これを実行すると `.node-version` というファイルが作られて、そのディレクトリ配下で使われる Node のバージョンがそれに固定されるの。

それから、さっきインストールしたプラグインで `anyenv update` というコマンドが使えるようになってるはずなので、Node を最新版にアップデートする前はこれを実行して、インストール可能なリストを更新するのを忘れないようにね」

「えっ、それを実行しないとリストが古いままなんですね……。うっかりしないようそれも気をつけます。えっとそれから、デフォルトインストールされるようにした npm パッケージについての説明もお願いしていいですか？」

「ああ、そうだったね、じゃ、今の時点で理解できそうなところだけ説明しておこうか。まず **Yarn**[25] から。

前にも説明したように npm は Node.js 公式の、Ruby でいうところの RubyGems と Rails の Bundler の機能をひとつにまとめたようなものだね。npm 公式リポジトリ（npmjs.com）で提供されているパッケージの追加・更新・削除に加えて、各パッケージ間のバージョン整合とかも自動的にやってくれる」

「ふむふむ」

「Yarn というのは Facebook 製の改良版 npm コマンドみたいなものなの。npm より高速だったりサブコマンドのタイピング数が少なかったり色々と使い勝手がいいので、React 界隈では npm 派と Yarn 派が拮抗してるくらいに普及してる。それでウチでは Yarn を使ってるの」

「なるほど」

「あとの 3 つは、TypeScript 本体とそのよく使うツールだね。これについては TypeScript を学ぶときに使いながら教えていくよ」

「わかりました」

「加えて `node` コマンドの使い方を説明しておくね。まず `node <JavaScript ファイル名>` で、その JavaScript ファイルのコードが実行される」

「これは `ruby` コマンドと同じですね」

「`ruby` コマンドとちがうのは、`node` と単体で実行すると REPL[26] が起動することだね。ちょっと実際にやってみようか」

```
$ node
> 4 * 8 + 1
> 33
> const foo = 'foo is string';
> foo
'foo is string'
```

「対話環境で JavaScript が実行できるんですね。Ruby でいう `irb` コマンドだ」

「そう、Node.js ではひとつのコマンドに統一されてるのね。最新バージョンではカーソルキー上下

[25] http://yarnpkg.com/

[26] 「RLead-Eval-Print Loop」の略で、対話型の実行環境を意味する。キーボードから打ち込まれた命令を読み込み（Read）、評価・実行し（Eval）、結果を画面に表示し（Print）、また命令待ちの状態に戻る（Loop）ためこう呼ばれる。

で入力履歴をたどれるのはもちろん、ある程度入力すると候補が表示されて Tab キーで補完できたり、変数の中身や式の結果は改行を入れなくても入力中に値が下に表示されたりと、けっこう高性能だよ」

```
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press ^C to abort current expression, ^D to exit the repl

> .load 01-hello/01-env/rectangle.js
> const square = new Rectangle(5, 5);
> square.getArea();
25
```

「`.help` で REPL コマンドの一覧が表示される。ざっと見ておいてほしいけど、よく使うのは `.load` かな。指定したファイルの内容を現在のセッションに読み込んでくれるコマンドだね。`node <JavaScrip ファイル名>` で直接実行するより、こっちのほうが色々挙動を試せて便利だから」

「へー、これよさそうですね。活用させてもらいます」

超絶推奨エディタ Visual Studio Code

「開発環境の構築、次はエディタね。秋谷さんは今、どのエディタを使ってる？」

「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**[27]、長いので略して『**VS Code**』って呼ぶけど、それを使ってもらいます」

「えっ、チームで使うエディタを指定されるんですか？」

「本当は各自で好きなものを使っていいよと言ってあげたいんだけど、将来的にジョインするであ

[27] https://code.visualstudio.com/

ろうメンバーも含めて DX を共通化する意味でも全員 VS Code を使ってもらいたいの。

VS Code を全員に使ってもらいたい理由のひとつは、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コーディング中に自動でやってくれること。さらにコードの自動整形や構文チェックツールとの統合、AI 支援による入力サジェスト機能を提供する便利な拡張も提供されてるので、それらをチームで使えばコードの品質や開発効率の向上につながるからね」

「ふーむ、そんなに Visual Studio Code っていいんですか？」

「世界の JavaScript ユーザーを対象に毎年行われている調査『The State of JavaScript』の最新 2019 年の結果では、そのシェアは 56.6 % と他をぶっちぎっての 1 位だね[28]」

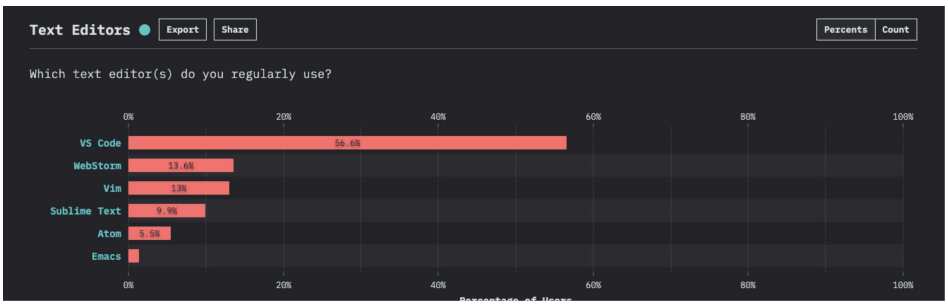


図 1: 2019 年調査の JS 開発者間でのエディタのシェア

「さらに React 開発元の Facebook でも、VS Code がデフォルトの開発環境になってるらしいよ[29]」

「へええええ――」

「VS Code を使ってほしい理由のもうひとつは、リモート開発機能が充実してること。Remote Development[30] でリモートのマシンに SSH 接続してそのまま開発できるし、Visual Studio Live Share[31] による同時コーディングの体験は圧倒的だね」

「同時コーディング？」

「Google Docs を使っての複数人による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んだよ」

「……それはすごいですね」

---

[28] https://2019.stateofjs.com/other-tools/#text_editors

[29] 「Facebook、マイクロソフトの『Visual Studio Code』をデフォルトの開発環境に - ZDNet Japan」 https://japan.zdnet.com/article/35145738/

[30] https://code.visualstudio.com/docs/remote/remote-overview

[31] https://visualstudio.microsoft.com/ja/services/live-share/

「ウチのチームでは実際の開発は常にペアプログラミングでやる予定なんだけど、ひとつの画面をいっしょにふたりで見ながら開発する従来のやり方じゃなく、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していきたいと考えてるのね。もちろん対面じゃなくリモート勤務時でも、ボイスチャットを併用してペアプロ開発するつもり」

「へー、もうそんな時代なんですね。近未来 SF の世界だ……。わかりました、がんばって私も VS Code 使いになります！」

「Vim のキー操作に慣れてるなら VS　CodeVim[32]　とか、もっと軽量でかつマルチカーソル対応の amVim[33] といった拡張を入れると 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」

「ちなみにプロジェクトのルートに `.vscode/` というディレクトリを作って、その中に `settings.json` ファイルを置くと VS Code の設定が、同じく `extensions.json` ファイルを置くと推奨の拡張リストがチームで共有できるようになるよ。これ以降のサンプルコードにも置いておくので、一度は気にして見ておいて」

「了解です！！」

■ 柴崎さんオススメの VS Code 拡張リスト

- **ESLint**（`dbaeumer.vscode-eslint`）
  …… JavaScript の静的コード解析ツール ESLint を VS Code に統合する
- **stylelint**（`stylelint.vscode-stylelint`）
  …… CSS 用のリンター stylelint を VS Code に統合する
- **Prettier**（`esbenp.prettier-vscode`）
  …… コード自動整形ツール Prettier を VS Code に統合する
- **Visual Studio IntelliCode**（`VisualStudioExptTeam.vscodeintellicode`）
  …… AI 支援により API サジェスト一覧の精度を向上させる

[32] https://marketplace.visualstudio.com/items?itemName=vscodevim.vim
[33] https://marketplace.visualstudio.com/items?itemName=auiworks.amvim

- **Bracket Pair Colorizer**（`coenraads.bracket-pair-colorizer`）

  …… マッチする括弧を色分けして教えてくれる

- **indent-rainbow**（`oderwat.indent-rainbow`）

  …… インデントの階層を色分けして見やすくしてくれる

- **vscode-icons**（`vscode-icons-team.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

- **Import Cost**（`wix.vscode-import-cost`）

  …… モジュールをインポートしている文の横に、算出したバンドルサイズを表示してくれる

- **Git History**（`donjayamanne.githistory`）

  …… Git のコミット履歴を見やすく表示してくれる

- **Debugger for Chrome**（`msjsdiag.debugger-for-chrome`）

  …… VS Code 上でブレークポイントの設定や変数の監視といったデバッガの機能が使えるようになる [34]

- **VS CodeVim**（`vscodevim.vim`）

  …… VS Code 上で走る Vim エミュレータ。キーバインディングを Vim 形式に変更するだけでなく、Undo-Redo 履歴や単語検索などの管理空間を独立させたり VS Code 本体とマージしたりもできる

- **Remote - WSL**（`ms-vscode-remote.remote-wsl`）

  …… Windows 環境から WSL のファイルシステムにあるプロジェクトを開けるようにする

- **Live Share**（`ms-vsliveshare.vsliveshare`）

  …… 複数人によるリアルタイムのコーディングコラボレーションを実現する

---

[34] 本書のステップアップ編となる『りあクト！ TypeScript で極める現場の React 開発』の「1-1. VS Code でらくらくデバッグ」で使い方をくわしく説明しています。

1-2.　プロジェクトを作成する

Create React App は何をしてくれているのか

「React はフルスタックなフレームワークではなく UI 構築のための必要最小限のライブラリなので、Ruby on Rails の `rails new` や Vue.js の `vue create` にあたる新規プロジェクトのスケルトンを生成してくれるようなコマンドは長らく存在しなかった。でも React でそれなりの規模のアプリを作ろうとすると、最新仕様の JavaScript や JSX を古いブラウザでも実行可能なコードに変換するためのコンパイラや、JavaScript および CSS のファイル群をひとつにまとめ minify [35] するためのバンドラなどを導入した上で、それらが連携して動作するよう複雑な設定を行う必要があったのね」

「……えええ、たいへんそう。私ならアプリ開発までたどりつく前に、たぶん根尽きちゃいますね」

「それは当時の開発者たちも同じで、DX を改善するはずのツールの導入だったはずがそれらが高度になるにつれ、その学習コストが増大するというジレンマに悩まされてた。ところが Facebook 社内では React を使うための環境が最初から整備されてたので、この問題はずっと放置されてたの」

「ひどい！　自分たちだけよければいいのか！」

「そうこうする内、いわゆる『JavaScript Fatigue（JavaScript 疲れ）[36]』が取り沙汰され、その問題が開発者たちの間で広く認識されるようになってきた。さすがにそこで態度を改めた Facebook が 2016 年 7 月にリリースしたのが Create React App[37] だったの。コマンドひとつでプロジェクトのスケルトンが作られるという、まあ他のライバルのフレームワークたちには当たり前にあったものが、公開から 4 年めにしてようやく提供されたわけね。

Create React App（CRA）は React 本体とは別のプロダクトなんだけど、React のコアチームのメンバーが開発に参加してるし、React の公式ドキュメントでも推奨されてるので [38]、実質的に React の公式コマンドみたいなものかな」

「なるほど。ではウチのチームでもこれを使うわけですね？」

---

[35] インタープリタ型プログラミング言語やマークアップ言語において、その機能を変更することなくソースコードから意味的に不要な文字（空白や改行、コメントやブロックなど）をすべて削除するプロセスのこと。ファイルを軽量化することによるパフォーマンスの改善を目的とする。

[36] 「JavaScript Fatigue - Eric Clemmons - Medium」
https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4

[37] https://create-react-app.dev/

[38] 「新しい React アプリを作る –React」https://ja.reactjs.org/docs/create-a-new-react-app.html

「うん。でも実際に使ってみる前に、CRA が何をしてくれるかをちゃんと理解しておこう。`create-react-app` コマンドを実行すると、指定したテンプレートを用いてアプリケーションのスケルトンが生成されるのと同時に、次の 3 つのパッケージの最新バージョンがインストールされる」

- react
- react-dom
- react-scripts

「react は React 本体のパッケージで、react-dom は DOM を抽象化して React から操作できるようにするレンダラーのパッケージ。最後の react-scripts は CRA の魔法を一手に引き受けてるパッケージとでもいえるかな」
「……『魔法』ですか」
「react-scripts の後ろには 50 個以上のパッケージが隠蔽されてる。react-scripts によって、その後ろに存在してる多数のパッケージ群の存在を意識することも、それらの複雑な設定を自分で行うこともなく次のようなことが可能になるの」

- 新しい仕様の JavaScript や JSX、TypeScript のコードを古いブラウザでも実行可能なレガシーな JavaScript にコンパイルする（Babel[39]）
- コンパイラと連携しつつ大量のソースコードファイルをひとつにまとめ、種々の最適化を施す（webpack[40]）
- ファイルの変更を自動検知して再ビルド、リロードしてくれる開発用の HTTP サーバでのアプリの稼働（webpack-dev-server[41]）
- オールインワンのテスティングフレームワークによるユニットテストの記述（Jest[42]）
- テスト記述の存在および、ファイルの差分を検知して自動で必要最小限のテストを走らせる
- `.env(.*)` ファイルによる環境ごとに異なった環境変数の設定（dotenv[43]）

---

[39] https://babeljs.io/
[40] https://webpack.js.org/
[41] https://github.com/webpack/webpack-dev-server
[42] https://jestjs.io/ja/
[43] https://github.com/motdotla/dotenv

「へえー、意外と多機能！　Create React App って、ただプロジェクトを作ってくれるだけじゃないんですね」

「Facebook が DX を重要視するようになった表れだろうね。開発者の中には、これは初心者専用のツールだとして意地でも使わずにすべて自前でセッティングしようとする人もいるけど、CRA は初級者はもちろん上級者の使用にも耐える完成度の高いツールだよ。また最初のセッティングだけ CRA に行わせた上で react-scripts の庇護から脱け出して、設定を自前でカスタマイズすることもできる」

「ふむふむ。ウチには柴崎さんがいるから、それでもよさそうですけどね」

「でもね、それをちゃんと運用していくのは『Frontend DevOps』と呼ばれる専任の職種があるくらいたいへんな仕事なんだよ。特に webpack の設定は『職人技』と揶揄されるくらい高度な知識と細かい調整が要求される。ウチのような少人数のチームがやるべきことじゃないのよ」

「――*webpack 職人の朝は早い* ――。……すいません、言ってみたかっただけです」

「そういうわけで、ウチでは React のプロジェクトは Create React App を使って作成して、運用もずっとそのレールの上に乗っかってやっていきます」

「了解です！」

Create React App で「Hello, World ！」

「Create React App を使うにはそのパッケージをインストールする必要はなくて、`npx` コマンド[44] 経由で実行するほうがいいの。公式ドキュメントでも CRA をグローバルインストールすることは推奨されてない[45]」

「どうしてですか？」

「プロジェクト生成コマンドってそんなしょっちゅう使うものじゃないからね。だから毎回アップデートしながら使うより、ネット経由で最新版を直接実行したほうが効率がいい。インストールしてしまうとそのアップデートも忘れがちになるしね。

　npx というのは npm パッケージで提供されてるコマンドを実行するためのものなんだけど、該当パッケージがマシンにインストールされていればそれを、なければ最新版をダウンロードしてきて

[44] https://github.com/npm/npx

[45] 「Getting Started - Create React App」https://create-react-app.dev/docs/getting-started

それを実行する。そしてダウンロード実行の場合は、実行後、そのコマンドのパッケージをきれいに削除してくれるの。1 回限りのコマンド実行にうってつけでしょ」

「なるほど」

「`npx` コマンドは npm と同時にインストールされているのでそのまま使えるはずだから、適当なディレクトリで次のように実行してみてくれる？」

```
$ npx create-react-app hello-world --template typescript
```

「このオプションに指定してる `--template typescript` って何ですか？」

「作成プロジェクトの下敷きにするテンプレートの指定ができるんだけど、公式が用意してくれてる cra-template-typescript という TypeScript のためのテンプレートをこれで指定してるの。ちなみにこのオプションを指定しなくても、cra-template っていうテンプレートがデフォルトで適用されてる。こっちは素の JavaScript で開発するためのものね」

「なるほど。じゃ、そのコマンド実行しますね。……おおおお、たくさんパッケージがインストールされていきます」

「npm パッケージだけでも 1,000 個以上はインストールされるからね。全部で 30 〜 40 秒くらいはかかるはず」

「1,000 個！！　そんなにあるんですか。……あ、終わりましたね。最後になんかメッセージが表示されてます」

```
We suggest that you begin by typing:

  cd hello-world
  yarn start

Happy hacking!
```

「じゃ、ここに書いてあるとおりにやってみよう。`hello-world/` ディレクトリに移動してから `yarn start` を実行」

「おおっ！　勝手に Chrome のタブが新しく開いて、でっかい React のロゴマークがぐるぐる回ってます。かっこいい！」

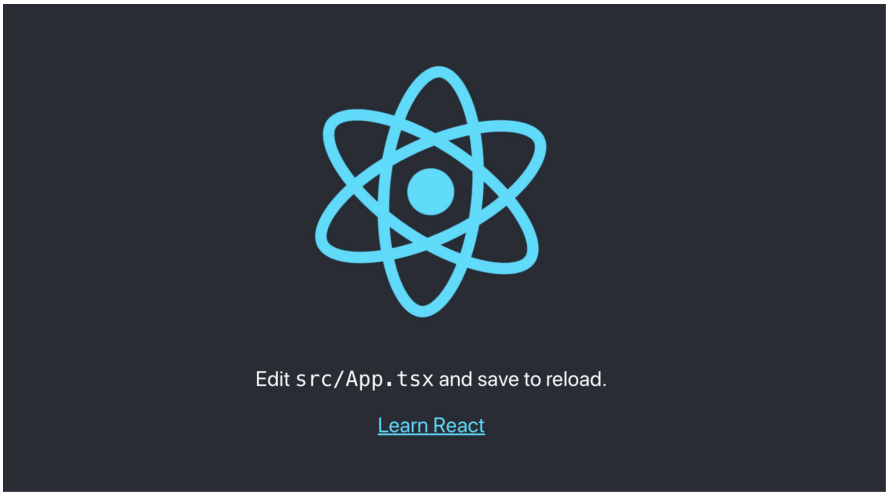


図 2: CRA でプロジェクト作成したデフォルトの画面

「デフォルトだと HTTP サーバが 3000 番ポートで立ち上がってそこでアプリが稼働され、同時にユーザーがふだん使ってるブラウザが開いてその画面を表示するようになってるのね」

「なんか文章も書いてありますね。『Edit `src/App.tsx` and save to reload.』だそうです。このファイルを開きますか？」

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

リスト 3: public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta name="description" content="Web site created using create-react-app" />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

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

「うん。それじゃ、次は `src/index.tsx` ファイルを見てみようか」

「あれっ、なんかいっぱいエラーみたいなのが出てますけど？」

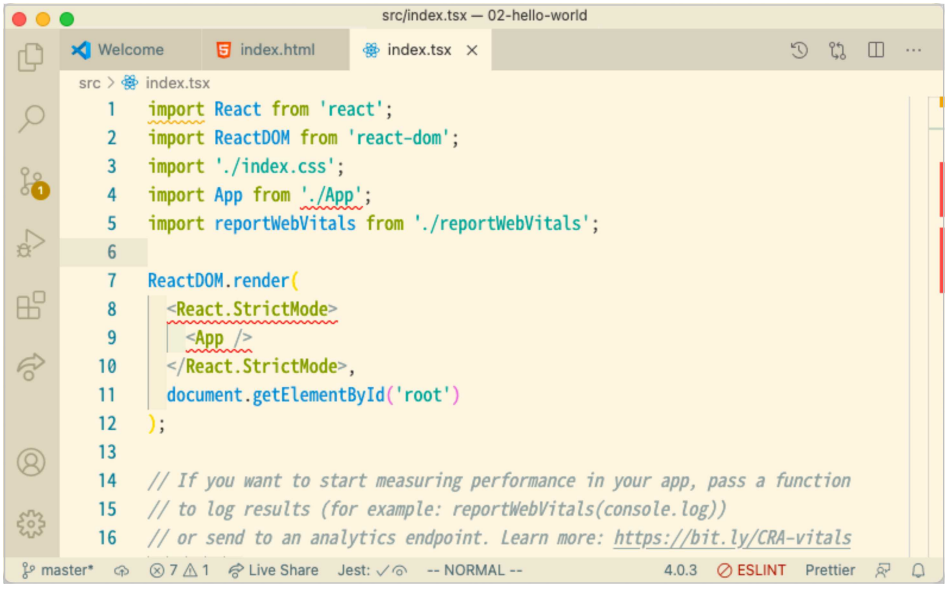


図 3: VS Code で index.tsx を開いた画面

「うん。まず Create　React　App で生成したプロジェクトには JavaScript の静的解析ツールである ESLint[46] が組み込まれてるんだけど、VS Code で動作させるには設定が必要なのね。ウィンドウの右下に赤文字で『ESLINT』って表示されてるでしょ。そこをクリックして表示される選択肢の中から『Allow Everywhere』をクリックして」

「はい。あ、赤文字だった『ESLINT』のところが通常色の表示になって、1 行目の `import` にあったオレンジの下線が消えました！」

「これで VS Code で ESLint が使えるようになった。それでまだ残ってる赤の下線なんだけど、2020 年 12 月現在 Create　React　App の提供している TypeScript 用のテンプレートの設定が不適切なのよ。`tsconfig.json` を次のように修正して保存してくれる？」

---

[46] 「6-1. ESLint」で説明します。

```
      ⋮
      "noEmit": true,
-     "jsx": "react-jsx"
+     "jsx": "react"
    },
    "include": [
      "src"
    ]
  }
```

「index.tsx の赤線エラーが消えました！　これってどういうことなんですか？」

「react-jsx オプションは TypeScript 4.1 以降に導入されたものなんだけど、今のテンプレートでは TypeScript 4.0.3 がインストールされるようになってるのでエラーになってしまうわけ。オプション値の意味についてはまた後でくわしく説明する予定なので、ここではとりあえずこうしておいて」

「わかりました」

「じゃ、index.tsx の中身を見ていこう」

リスト 4: src/index.tsx

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

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

reportWebVitals(console.log);
```

「ReactDOM.render() の 2 つめの引数が document.getElementById('root') ってなってるでしょ。これがさっきの index.html ファイルの中の に対応してるわけ。この render メソッドは、第 1 引数に渡された React のコンポーネントを DOM に描画しなおして第 2 引数で指定された HTML 要素に上書きしてくれるのね」

「ふむふむ、なるほど。ところでこの `<React.StrictMode>` というタグは何ですか？」

「ああ、これはバージョンが進んで非推奨になった API の使用や意図しない副作用といった、アプリケーションの潜在的な問題点を見つけて warning で教えてくれる React の『Strict モード[47]』という機能を有効にするためのラッパーなのね。アプリケーションをより安全なものにするために Create React App がバージョン 3.4.1 からデフォルトで追加してくれるようになったの」

「じゃ、描画する内容には特に影響もないってことですね。なら本体はその中の `<App />` タグですか。

……でも、そもそもこれらのタグって何なんですか？　HTML にはこんなタグはありませんし、純粋な HTML ではタグ名は全部小文字で書くはずですよね」

「そうね、じゃあ基本概念から説明していくのでしっかり聞いててね。まず 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**（Component）』の組み合わせで構成される。コンポーネントというのは、今の段階では『任意の UI を表現するパーツの単位』くらいに考えておけばいいかな。そしてその命名規則として、コンポーネント名は必ず大文字で始まるパスカルケース[48]になってる」

「ということは `App` も、そのコンポーネントなんですね」

「そのとおり。4 行めに `import App from './App';` とあるでしょ。これは `App.tsx` ファイルから `App` コンポーネントを読み込んできてるわけ。ちなみにインポート時のファイル拡張子は省略可能ね」

「`App.tsx` って、ブラウザの画面に表示されてたメッセージで編集しろっていわれてたファイルですよね」

「そう。次はそのファイルの中を見てみよう」

リスト 5: 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">
```

[47] 「strict モード –React」https://ja.reactjs.org/docs/strict-mode.html

[48] 「PascalCase」のように複合語をひとくくりにして、各単語の最初を大文字で書き表す記法。プログラミング言語 Pascal で最初に使われたことからこう呼ばれる。「アッパーキャメルケース」とも。

```
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</code> and save to reload.
        </p>
           <a  className="App-link"  href="https://reactjs.org"  target="_blank"  rel="noopener
noreferrer">
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

「React ではコンポーネントの実装は、関数またはクラスで定義することになってるのね。ここでは関数で `App` コンポーネントを定義してる」

「ふむふむ、たしかに画面に表示されてたメッセージの『Edit `src/App.tsx` and save to reload』が中にありますね。それにしても `retrun` 文で直接 HTML が返されてるのが、なんていうかすごいワイルドですね」

「ちがうんだよ。さっきの `ReactDOM.render()` の第 1 引数として渡されていたものもそうだったけど、これらは HTML のように見えても実際は HTML じゃない。『**JSX**[49]』っていう JavaScript の構文拡張で、最終的には JavaScript にコンパイルされるものなのね」

「これがウワサに聞く『JSX』ですか……。HTML に見えて実は JavaScript っていうのが不思議な感じですね」

「ファイル名が `.tsx` となってるでしょ。これは TypeScript が記述できる JSX のファイルってことで、その中で react パッケージから `React` オブジェクトをインポートすることで JSX の記法が使用可能になってるの」

「ふーむ」

「これで流れはわかっただろうから、今度は『Hello, World!』をやってみよう。`App.tsx` の中身をこんなふうに書き換えてみて」

```
  function App() {
    return (
```

[49] https://facebook.github.io/jsx/

```
      <div className="App">
       <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
-           Edit <code>src/App.tsx</code> and save to reload.
+           Hello, World!
          </p>
          ⋮
```

「……はい、書き換え終わりました」

「じゃ、そのファイルを保存して」

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

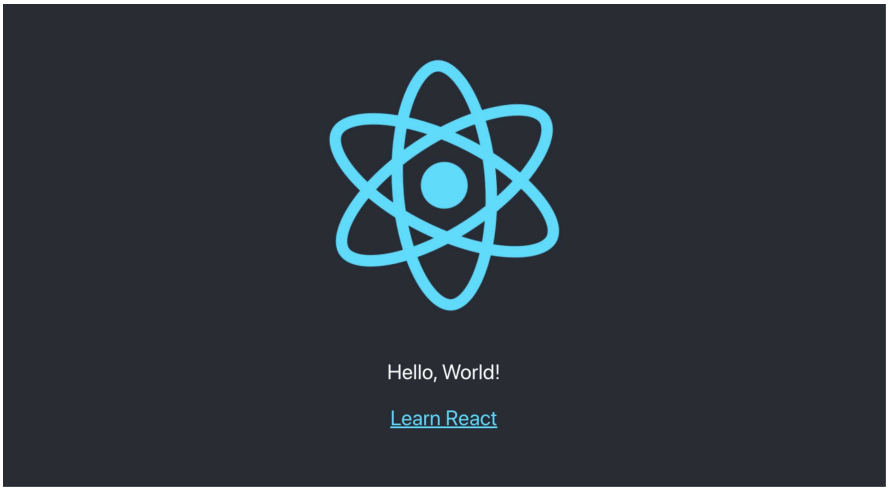


図 4: Hello, World!

「こうやってアプリを拡張していくわけね。何か質問ある？」

「はい。いちばん最初に 1 ファイルで React を動かしてもらったサンプルのリスト 1 では、`id="root"` の HTML 要素と `ReactDom.render()` の第 2 引数 `document.getElementById('root')` との結びつきが同じファイル内なのですぐわかったんですが、CRA で生成されたプロジェクトの `public/index.html` と `src/index.tsx` はどうやって結びつけられるんですか？」

「そうだね。アプリのページが表示されてる Chrome のタブをアクティブにした状態で command + option + U（Windows の場合は ctrl + u）を押して、ブラウザに実際に返されてる HTML のソースコードを見てみて」

```
    ⋮
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <script src="/static/js/bundle.js"></script>
    <script src="/static/js/0.chunk.js"></script>
    <script src="/static/js/main.chunk.js"></script>
  </body>
</html>
```

「あれ？　`public/index.html` にはなかったスクリプトファイルの読み込みタグが最後に挿入されてる」

「さらにその読み込まれてる 3 つめの `/static/js/main.chunk.js` というのを開いてみようか。これがレガシーな JavaScript にコンパイルされたアプリの中身ね。このページ内を『`function App()`』という語句で検索してみて」

「あっ、これがコンパイル後の `App` コンポーネントなんですね。`"Hello, World!"` のテキストもちゃんとあります」

リスト 6: /static/js/main.chunk.js

```
function App() {
  return /*#__PURE__*/react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement("div", {
    className: "App",
    ⋮
  }, /*#__PURE__*/react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement("header", {
    className: "App-header",
    ⋮
  }, /*#__PURE__*/react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement("img", {
    src: _logo_svg__WEBPACK_IMPORTED_MODULE_1___default.a,
    className: "App-logo",
    ⋮
  }), /*#__PURE__*/react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement("p", {
    __self: this,
    __source: {
      fileName: _jsxFileName,
      lineNumber: 10,
      columnNumber: 9
    }
                                  }, "Hello, World!"),
/*#__PURE__*/react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement("a", {
    className: "App-link",
```

```
    ⋮
  }, "Learn React")));
}

/* harmony default export */ __webpack_exports__["default"] = (App);
```

「CRA で作成したプロジェクトでは、こうやってソースコードファイルはコンパイラである Babel によってコンパイルされ、それがバンドラである webpack によって適切な形にまとめられ、それらが相互に関連付けられるの。開発中は意識することがない部分だけど、本番へのデプロイでトラブルに行きあたったときとかに備えて知っておいたほうがいいよね。ちゃんとそこを疑問に思ったのは偉いよ」

「えへへ。でもただのとおり一遍（いっぺん）な『Hello, World!』じゃなく、ここまで踏み込んで教えてもらえたので、普通にチュートリアルをやるよりだんぜん理解度が深まったと思います。ありがとうございます！」

1-3. アプリを管理するためのコマンドやスクリプト

Yarn コマンド

「Yarn は npm コマンドの Facebook による改良版という話はしたよね。Create React App も Facebook 製だから、Yarn がグローバルインストールされていた場合は、生成したプロジェクトがデフォルトで Yarn を使うようになってる。もし Yarn じゃなく npm を使いたいなら、`create-react-app` コマンドを実行する際に `--use-npm` オプションを指定してあげる必要がある」

「そこは切り替えができるんですね」

「実行速度については npm も最新のバージョンではかなり Yarn に近づいてきてるけど、コマンドの簡易さとかも考えるとわざわざデフォルトから乗り換える必要性を感じないので、ウチのチームではそのまま Yarn を使います」

「はい」

「ところでさっきのサンプルコードをリモートのリポジトリに上げておいたので [50]、ちょっと別のディレクトリに落としてくれる？」

---

[50] https://github.com/oukayuka/Riakuto-StartingReact-ja3.1/tree/master/01-hello/02-hello-world

「了解です。……はい、ローカル環境に展開しました」
「じゃ、そのアプリを起動してみて」
「えっと `yarn start` でしたよね。実行……っと。あれ？　エラーになりましたよ」
「うん、そのディレクトリ内を見てみて。`node_modules/` ディレクトリがないでしょ。アプリが参照してる npm のパッケージモジュールのインストールが必要なの。だからまず `yarn install` 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。あらためて `yarn start` を実行したら、今度は http://localhost:3000 でアプリが立ち上がりました」
「ちなみに Yarn では `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」

- `yarn (install)` …… プロジェクトルートに存在する `package.json` の記述にしたがって、依存関係のあるパッケージをすべてインストールする
- `yarn add <PACKAGE_NAME>` …… 指定したパッケージをインストールする
- `yarn remove <PACKAGE_NAME>` …… 指定したパッケージをアンインストールする
- `yarn upgrade <PACKAGE_NAME>` …… 指定したパッケージを最新バージョンに更新する
- `yarn info <PACKAGE_NAME>` …… 指定したパッケージについての情報を表示する

「プロジェクトにインストールされるパッケージの情報は、そのルートディレクトリの `package.json` に記述される。ちょっとそのファイルを開いてみて」

リスト 7: package.json

```
{
  "name": "hello-world",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "@types/jest": "^26.0.15",
```

```
    "@types/node": "^12.0.0",
    "@types/react": "^16.9.53",
    "@types/react-dom": "^16.9.8",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-scripts": "4.0.1",
    "typescript": "^4.0.3",
    "web-vitals": "^0.2.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

「`dependencies` エントリの中にパッケージらしき名前とバージョン番号がありますね。これのことですか？」

「そう。ちなみにさっき説明したYarnのコマンドを使わずに、このJSONファイルの `dependencies` エントリを編集してから `yarn⏎` を再実行することでも `add` や `remove` コマンド相当のことができるよ」

「へー、なるほど」

「この `package.json` に各パッケージのインストールするべきバージョン番号が指定されてるわけだけど、各パッケージが依存しているパッケージ群がどのバージョンでインストールされるかというのは、これだけでは固定されない。それぞれのパッケージは日々開発が進んでバージョンが更新されているので、`package.json` の内容が同じであってもまっさらな状態から `yarn install` でインストールされた内容は、今日実行するのと１ヶ月後に実行するのとではおそらく異なってくる」
「それって何か困るんですか？」
「同じアプリを開発してるはずのメンバーそれぞれで、インストールされているパッケージのバージョンがちがうと挙動が変わることもあるし、ひいては開発環境では動いてるのに本番環境では動かないなんてことがザラに起こってくるよね」
「そっか、そうですよね」
「だからこそ Yarn ではデフォルトで、パッケージの各依存関係のどのバージョンがインストールされたのかを、正確に記録し再現するために `yarn.lock` というファイルがプロジェクトのルートに作られるようになってるの」
「はい、`yarn.lock` ファイルがたしかにありますね」
「ちなみに npm コマンドを使ってる場合は `package-lock.json` というファイルね。通常ではこれらのロックファイルは Git のリポジトリに含まれるようになってるので、その内容が同じである限りインストールされる npm パッケージ群のバージョンはすべて同じになるはずなのね。だからうっかりこのファイルを削除しないようにね」
「わかりました！」
「『何もしてないのに壊れた！』ってなる場合、パッケージの依存関係がおかしくなってて `yarn.lock` の内容を以前のものに戻すと直ることがたまにある。ライブラリの作者も万能じゃないので、最新版にバグがあってそれが他のライブラリから呼ばれてた、みたいなことだったりするの。それゆえにちゃんと動いてたときのロックファイルは大事に取っておかなきゃいけない」
「なるほど」
「今から説明するコマンドを実行する際も同じね。パッケージ更新ですごく便利なコマンドなのでよく使うんだけど、実行後の動作は保証されないから」

```
yarn upgrade-interactive [--latest]
```

「このコマンドは？」
「`package.json` に記述してある各パッケージのバージョン範囲にしたがって、すべてのインストール

してあるパッケージの更新情報をチェック、対話形式で一括アップデートできるコマンドなの。`--latest` オプションを指定すると、すべてのパッケージが強制的に最新の安定版バージョンに一括アップデートされる。このときもちろん `package.json` も上書きされる」

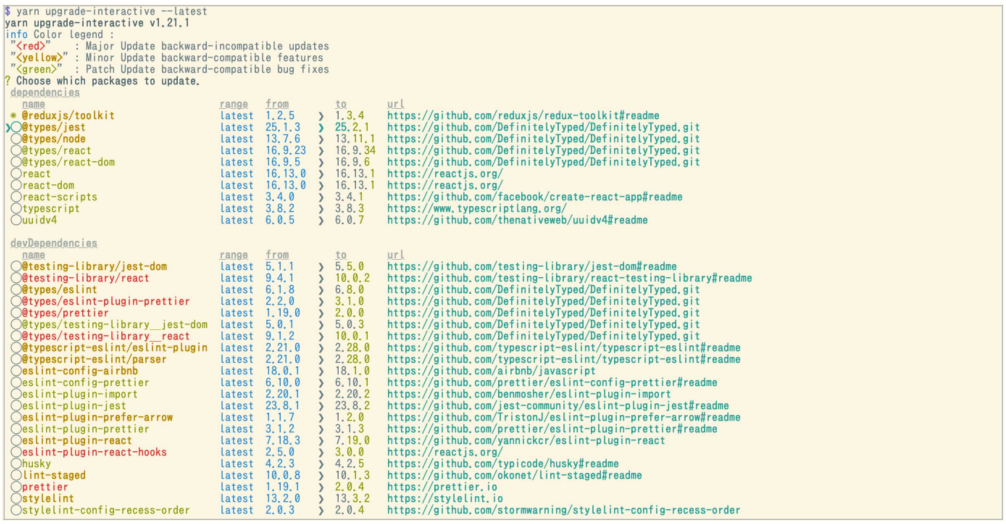


図 5: yarn upgrade-interactive 実行時の対話型インターフェース画面

「へー、これは便利ですね！」

「これに限らず `yarn upgrade` はバージョン整合を取りつつパッケージをアップグレードしてくれるとはいえ動作を保証してくれるものではないので、実行後のテストや動作確認は必要。`upgrade-interactive --latest` を実行した場合は特にね。更新後に動かなくなったときは、いったん `package.json` や `yarn.lock` を元に戻しましょう」

「了解です！」

npm-scripts

「それじゃ `package.json` ファイルに戻って、`dependencies` の次にある `scripts` エントリを見てくれる？」

「はい」

```
  ⋮
```

```
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
   ⋮
```

「`yarn help` で Yarn のサブ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 `yarn start` が実行できるのはなぜか。これは『**npm-scripts**[51]』と呼ばれるものを使ってるからなの。`package.json` の中の `scripts` エントリに実行させたい処理コマンドを記述してリストに入れておくと、あたかも npm コマンドや Yarn コマンドのように実行できる」

「ふむふむ」

「npm だと npm-scripts を実行するためには `npm run start` のように `run` を入れないといけないんだけど、Yarn なら `yarn start` と短くて済むので、私は npm-scripts の実行にも Yarn を使ってる。

　ここで定義されてる `"start"` のエントリに注目して。プロパティが `"react-scripts start"` ってなってるでしょ。こう定義することで、`yarn start` を実行したときに `react-scripts start` が実行されるのね。そして npm-scripts 実行時には `node_modules/.bin/` ディレクトリにパスが自動的に通されるので、そのディレクトリ内にある `react-scripts` という実行権限のついたスクリプトファイルが実行されるわけ」

「なるほど。じゃ、ここに自分でエントリを追加すれば、それを npm-scripts として実行できるんですか？」

「そのとおり。後々、構文チェックのためのコマンドのエントリとかを追加していく予定だよ。

　ただこのエントリには特別な挙動が約束されている予約キーワードが存在してるので、気をつける必要がある。予約キーワードには意味付けだけがなされたものと、他の npm-scripts の実行をフックとしてその前後に実行されるものの 2 種類があるの。まず意味付けだけがなされた予約キーワードは次の 4 つ」

- `start` …… 開発用アプリケーションサーバの起動コマンドの登録用
- `restart` …… 開発用アプリケーションサーバの再起動コマンドの登録用

[51] https://docs.npmjs.com/misc/scripts

- `stop` …… 開発用アプリケーションサーバの停止コマンドの登録用
- `test` …… テスト実行開始コマンドの登録用

「そして他の npm-scripts の実行をフックに実行される予約キーワードについてはたくさんあって紹介しきれないけど、アプリケーションの開発で使うことがありそうなのは次に示すこれくらいかな」

- `preinstall` …… パッケージがインストールされる前に実行される
- `install`, `postinstall` …… パッケージがインストールされた後に実行される
- `preuninstall`, `uninstall` …… パッケージがアンインストールされる前に実行される
- `postuninstall` …… パッケージがアンインストールされた後に実行される
- `prestart`, `start`, `poststart` …… `start` コマンドをフックに実行される
- `pretest`, `test`, `posttest` …… `test` コマンドをフックに実行される

「npm-scripts、基本はコマンドのエイリアスとしての用途がほとんどなんだけど、こういう予約キーワードがあるってことはおぼえておいてね」
「わかりました」
「それから、`yarn start` で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 VS Code 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かな。エラーがあったとき、指摘部分を command + 左クリックで該当ファイルが開いたりと色々と便利だからね」
「そうなんですね、了解です！」

react-scripts

「`package.json` 内に定義されてる npm-scripts で実行されてた `node_modules/.bin/react-scripts` って、Creat React App で生成したプロジェクトの中で Babel や webpack を隠蔽しつつ色々奮闘してくれてるっていう react-scripts ですよね？」
「その react-scripts だね。それぞれの実体は `node_modules/react-scripts/scripts/` ディレクトリの中

にある。ちなみに、`create-react-app` コマンド実行完了時に、react-scripts のコマンド一覧が表示されてたんだよ。見逃してたかもしれないけど」

```
Success! Created hello-world at /Users/kanae/study/hello-world
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can ' t go back!
```

「あ、そうだったんですね。見落としてました……」

「ここにあるコマンドは、`package.json` の `scripts` エントリで設定されてる `react-scripts` の実行に一対一で対応してる。`start` はもうやったけど、ローカルで開発用の HTTP サーバを起動してそこでアプリを稼働させるコマンドね」

「さっきのやつですね」

```
  ⋮
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  ⋮
```

「`build` は本番環境にデプロイするためのファイルを作成するコマンド。これを実行すると、プロジェクトルートに `build/` ディレクトリが作られ、その中に一連のビルド済みファイルが展開される[52]。

[52] 「Creating a Production Build - Create React App」https://create-react-app.dev/docs/production-build/

`test` コマンドは、ソースディレクトリからテストファイルを抽出してテストを走らせる。ちなみにこれを起動したままにしておくと、テストファイルの差分を検知して、変更のあったテストだけが実行されるよ[53]。テストの書き方とかは今回はやらないけど、知識としては知っておいて」

「わかりました、おぼえておきます」

「最後の `eject`。これが前にちょっとだけふれた、react-scripts の庇護(ひご)から抜け出すためのコマンドね。実行すると隠蔽されてた 50 以上のパッケージが `package.json` の `dependencies` エントリに現れる。またプロジェクトルートには `config/` ディレクトリが作られ、そこに webpack などの設定ファイルが置かれる。これは茨(いばら)の道だけど、その代償に webpack の設定をいじらないと使えないようなプラグインが使えるようになったりとか、各種ライブラリが提供する独自の DSL[54] を Babel にコンパイルさせることができたりといった自由が手に入るの」

「庇護の下(もと)の安寧か、荒波に繰りだす自由か、の二択なわけですね」

「そう。ウチのような少人数のチームは Facebook 帝国の庇護の下にいるほうが無難だね。ツールの設定にリソースを奪われることなく、コードを書くことだけに集中できる。

それに `eject` は一方通行の操作で、一度実行したらそのプロジェクトはもう元には戻せなくなる。どうしても仕方がないときにだけ慎重に慎重を重ねて行わないといけないコマンドなわけ。私はこれ実際のプロジェクトで一度だけやったことがあるけど、後になってかなり後悔したので、二度とやりたくない」

「……経験者が語る言葉は重みがありますね」

「最新の Create React App では必要な機能はひととおりそろってるし、今は `eject` せずとも react-scripts の庇護にとどまったまま webpack や Babel の設定を部分的に書き換えられる react-app-rewired[55] や CRACO[56] のようなパッケージもあるしね。ウチは基本、長いものに巻かれろをモットーに `eject` はしない方針で」

「賛成です！」

---

[53] 「Running Tests - Create React App」https://create-react-app.dev/docs/running-tests/

[54] 「ドメイン固有言語（Domain Specific Language）」。特定の一分野のタスク向けに設計されたコンピュータ言語のこと。たとえば正規文法を記述する正規表現、データベースへの問い合わせをおこなう SQL などがこれに当たる。

[55] https://github.com/timarney/react-app-rewired

[56] https://github.com/gsoft-inc/craco

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？

それは世界でもっとも誤解されたプログラミング言語

「とりあえず『Hello，World!』はできたけど React について本格的に学んでいく前に、その記述言語の知識をしっかり身につけておいてほしいの。ウチでは実際には TypeScript で開発していくわけなんだけど、TypeScript は JavaScript に静的型付けによる拡張された型システムを加えた上位互換の言語なので、まずはそのベースとなってる JavaScript から学んでいこう。といっても、秋谷さんは JavaScript での開発経験があるんだよね？」

「あっ、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ったので。ただちゃんと学んだことがあるわけじゃなく、その場その場で必要なとこだけ調べながら書いてました」

「つまり条件分岐や繰り返し構文とかの基本的な文法は知ってて jQuery を使った DOM 操作くらいはできるけど、それ以上に踏み込んだところまでの知識はないと。じゃあそのレベルでの感想でいいんだけど、秋谷さんは JavaScript にどんなイメージを持ってる？」

「うーん、基本的な構文は Java っぽいんだけど、要所要所でクセが強い印象がありますね。クラスはまともに使えないし、たまに出てくる『コールバック』っていうのよくわからないし……。周りの先輩たちも『できるだけ JS は使いたくない』って言ってました」

「……ああ、サーバサイドの Web アプリケーションフレームワークに長年慣れ親しんだ人たちの中には JavaScript のことを、HTML の飾り付けのためだけに使う書き捨て用の貧弱な言語だと思ってる人が少なくないみたいだからね。でもそれは、偏った価値観とアップデートされてない古い知識による不当な評価だと私は抗議したい。日本では JavaScript を悪くいう開発者は今でも多いけど[57]、世界的にはここのところずっと再評価の流れにあるし」

「そうなんですか？」

---

[57] 「JavaScript を悪いプログラミング言語だと考えるプログラマーが多いのは何故ですか？ - Quora」https://bit.ly/2WQ0n55

「うん。GitHub の pull request 数や Stack Overflow のタグ数などを集計してほぼ半年ごとに発表されてる Redmonk の言語ランキングでは、JavaScript は 2012 年からずっとトップにあり続けてて、その座を譲ったのはたったの 1 回だけ[58]。

また純粋に開発者からの人気を測れるものとして Stack Overflow による Developer Survey 2020[59] での『Most Loved Languages（もっとも愛されている言語)』ランキングでは、JavaScript は 10 位とそこそこの位置につけてる。さらに同じ調査の『Most Dreaded Languages（もっとも敬遠されている言語)』ランキングでは 16 位とかなり低く、同 6 位の PHP や 7 位の Ruby、9 位の Java よりもずっと下の順位だったりする」

「へー、知りませんでした。世界的には JavaScript って、けっこう好かれてるんですね」

「JavaScript はたしかに初期設計が甘く、安全性を欠く挙動を生みかねない仕様を多く含んでる。でもそれらは近年のアップデートによってかなりカバーされてきてるし、進歩がめざましい静的解析ツールとの組み合わせで十分フォローできる。必要以上にそこをあげつらって JavaScript を使えない言語だと主張する記事を見かけるけど、それらはフェアな意見じゃないよ。それにいっぽうで JavaScript の基本的な設計の大枠は、あの時代にあって慧眼ともいえるものだったと思う」

「ふむふむ、話を聞いてると柴崎さんの JavaScript への評価ってかなり高いんですね。柴崎さんは JavaScript のどこをそんなに評価してるんですか？」

「そうだね。いくつか挙げるとしたらこんなところかな」

- 第一級関数とクロージャをサポート
- プロパティを随時追加できる柔軟なオブジェクト
- 表現力の高いリテラル記法

「えーっと、ひとつめからして何のことやらさっぱりわからないんですけど……」

「JavaScript には秋谷さんのように Ruby や Java をメインにしている開発者にとって馴染みのない概念が多くて、そこが彼らを困惑させて低い評価につながってる面もあるんだよね。これらを秋谷さんに今の段階で説明して、すぐ納得させてあげるのは無理かな。でもこれからステップを踏んで

---

[58] 「The June 2019 RedMonk Programming Language Rankings: a MonkChat」
https://redmonk.com/kfitzpatrick/2019/07/31/june-2019-redmonk-programming-language-rankings-monkchat/

[59] Stack Overflow が毎年実施している、有志の開発者を対象としたアンケートによる調査結果。2020 年は約 65,000 人からの回答があった。
https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted

JavaScript を学んでいってもらうので、ひととおり終わるころにはこの意味が理解できるようになってるはずだよ」

「わかりました、じゃあ今、理解できそうな部分で質問させてください。そもそもの話なんですけど、JavaScript って Java とどういう関係なんですか？　名前と構文が中途半端に似てますけど、実際のところはまったくの別物だし。だから私も最初、Java の簡易版のようなつもりで使おうとして混乱したんですよね」

「それにはちょっとした歴史的経緯があってね。生みの親である Brendan Eich は、静的スコープと第一級関数をサポートする Scheme[60] とプロトタイプベースのオブジェクト指向言語 Self[61] のいいとこどりをしてブラウザで動く言語を作ろうとした。それは最初『Mocha』と名付けられ、その後『LiveScript』になった。その過程で Eich の雇い主だった Netscape の経営陣は、この言語を普及させるには同じくブラウザで動作し、当時飛ぶ鳥を落とす勢いだった Java と構文を似させる必要があると判断した。そして彼にそう改変するよう指示して、ついには言語の名前も『JavaScript』に変えさせたの[62]」

「うーん。流行ってる言語に名前と構文を似せたら、その人気にあやかれるって発想がわかりませんけど」

「でも結果的には大成功したわけだしね。ただそれが名前のおかげだったのかは、今となってはわからない。個人的には Mocha はアリだけど、LiveScript よりは JavaScript のほうがよかったかな。

　とはいえ中身はまったく別のパラダイムで作られた言語なのに、名前と構文が似てるせいで秋谷さんのように混乱してしまう開発者が多いのも事実だね。JavaScript のことを『C の皮を被った Lisp』[63] だという人もいるけど、言い得て妙だと思う」

「ええ〜、JavaScript って Lisp の仲間だったんですか？　つまりほとんど関数型言語ってこと？　名前と見た目が知ってるものと似てるから油断しちゃいました！」

「基本的な構文が C ライクだから、たいていの開発者はわかってなくてもある程度書けてしまうん

---

[60] Lisp の方言のひとつ。関数を第一級オブジェクトとして扱うことができる（第一級関数）ほか、静的スコープ（「レキシカルスコープ」とも。構文構造のみから決定できるスコープのこと）が特徴。Scheme により Lisp 方言に静的スコープが広められた。

[61] クラスの硬直性を解決するべく生まれた「プロトタイプ」の概念に基づいたオブジェクト指向言語。JavaScript や Lua に概念的な影響を与えた。

[62] 「JavaScript: how it all began」https://2ality.com/2011/03/javascript-how-it-all-began.html

[63] 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 p.3

だよね。それでちょっと複雑なことをしようとすると、馴染みのない概念に突き当たって困惑する。JavaScript を学ぶ上で、C の系譜にある言語という先入観はむしろ障害になると思う」

「……そうだったんですね」

「うん。Java と JavaScript は、マリオブラザーズとスーパーマリオくらいちがうと思ってないと」

「すいません。その喩(たと)え、全然ピンとこないんですが……」

年々進化していく JavaScript

「秋谷さんが前にいた Rails のプロジェクトでは、CoffeeScript[64] は使ってなかったの？」

「はい。けっこう前からあるプロジェクトでコードベースが古かったんですけど、最初の設計者が CoffeeScript を嫌ってたらしく JavaScript を直接書いてました」

「その様子だと Webpacker[65] も使わずに ES5 を直接書いてたみたいだね」

「いーえすご？」

「正式名は『ECMAScript 5th Edition』ね。『JavaScript』というのはもともと Netscape 社が開発し、それを今は Mozilla Foundation が引き継いでるプロダクトとしての名前で、標準仕様の名前は『**ECMAScript**（エクマスクリプト）』っていうの。なしくずし的にその一民間団体による実装の名前のほうがよく使われてるけど」

「ホッチキスとか、ポストイットみたいなもんですか」

「まあそんな感じだね。ECMAScript は Ecma International[66] という業界団体が定めている JavaScript の標準仕様で、1997 年に初版が公開された。さっき言及した ES5 というのは、これの第 5 版で 2009 年に出されたものね。ES5 が現在もっとも広く使われている版で、IE9 上でさえも動作する。だから多くの AltJS[67] もコンパイルターゲットをこの ES5 にしてる」

「ふむふむ」

---

[64] JavaScript をより簡潔に、読みやすくするために開発された言語。コンパイルにより JavaScript のコードに変換される。文法は Ruby や Python から影響を受けている。Ruby on Rails ではバージョン 3.1 以降、フロントエンドを記述するための言語として正式サポートされている。https://coffeescript.org/

[65] JavaScript のモジュールバンドラである webpack を Ruby on Rails で使えるようにする gem パッケージ。https://github.com/rails/webpacker

[66] https://www.ecma-international.org/

[67] 「Alternative JavaScript」の略で、JavaScript の代替言語を指す。AltJS で書かれたプログラムは、コンパイラによって最終的に JavaScript コードに変換される。

「ECMAScript は近年、ほぼ毎年のように新しい仕様が公表されてるのね。2020 年 12 月現在での最新版は ES2020、つまり『ECMAScript 2020』で、この 6 月に正式リリースされたばかり」
「ES5 は通し番号なのに、ES2020 は年号なんですね。Windows や Office みたい」
「2015 年に公開された第 6 版から、エディション名じゃなく年号付きの仕様書名で呼ぶことが推奨されるようになったのね。『ES6』と表記している記事も多いけど、それらはすべて ES2015 のこと。
　なぜそこから呼び方が変わったかという理由は、ES2015 で大きな改変があってそこでモダンな仕様が一気に盛り込まれた記念すべき転換点の年だからなの。ES5 までの JavaScript しか知らない人には、ES2015 以降の JavaScript は別の言語と思ってもらってもいいくらい」
「……ううっ、そうなんですね」
「ただいっぽうで、JavaScript ほど後方互換性を大事にしてる言語は他にないともいえる。古いブラウザで動いてたプログラムが新しいブラウザで動かなくなるという状況を作らないために、基本的に後方互換性を破壊する変更は ECMAScript の改定ではほとんど入らないのね。これはしょっちゅう後方互換性を破壊してる Ruby や、バージョン 2 から 3 への移行が大騒動になった Python とかとはかなり対象的だよね。
　ECMAScript の改訂仕様は古いものから年代を経た地層のように積み重なってる。だから古くて問題のある仕様も、掘り起こせば当時のままたくさん残ってるわけ」
「なるほど」
「でもそれらの『悪いパーツ』や『ひどいパーツ』[68] は最新の開発ツールを併用すれば、うっかり使ってしまわないようにうまく避けることができるので、実際にはそれほど気にする必要はないよ。そして最新の ECMAScript で追加されている仕様を駆使すれば、後発のモダンな言語たちにも引けを取らない快適で効率的な開発ができる」
「その『悪いパーツ』ばかり批判する人たちが見てる JavaScript と、柴崎さんたちが評価してるいいとこ取りした最新の JavaScript は、同じ『JavaScript』といいながら実のところかなり別物なんでしょうね」
「そうだと思うよ。ちなみに今の Create React App で TypeScript テンプレートを指定してプロジェクトを作成すると、デフォルトでコンパイルに含められる ECMAScript のライブラリが ESNext、つまりそのバージョンの TypeScript がサポートするもっとも新しいバージョンの ECMAScript にな

[68] 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 p.117

る。TypeScript 3.7 系なら ES2019、3.8 〜 4.1 系なら ES2020 がそれにあたるね。予定では TypeScript 4.2.0 から ES2021 になるはず[69]」

「ふむふむ」

「なお ES5 の基本的な構文は押さえているものとして、React と TypeScript によるモダンフロントエンド開発で必要な部分にしぼって説明していくつもりだから」

「はい、お願いします！」

2-2. 変数の宣言

「ではまず変数の宣言から。秋谷さんたちはこれまで変数の宣言には `var` キーワードを使っていたのかもしれないけど、これはもう金輪際、使ってはいけません」

「え、JavaScript での変数宣言って `var` を使うっておぼえてたんですけど、ダメなんですか？　じゃあどうすれば？」

「今の JavaScript には `const` と `let` という変数を宣言するためのキーワードが追加されてるの。だからその 2 つを代わりに使って」

「わかりました。でもどうして `var` を使っちゃいけないんでしょうか？」

「従来からある `var` は、安全なプログラミングをする上で次のような問題を抱えてるのよ」

1. 再宣言および再代入が可能
2. 変数の参照が巻き上げられる
3. スコープ単位が関数

「ふーむ。よくわかりませんけど、これらの何がどうまずいんですか？」

「ひとつずつ見ていこうか。`node` コマンドを REPL モードで使いながら説明するね」

```
$ node
> var a = 1;
> a = 2;          // 再代入 OK
```

[69] https://github.com/microsoft/TypeScript/issues/41238

```
> a
2
> var a = 3;    // 再宣言 OK
> a
3

> let b = 1;
> b = 2;        // 再代入 OK
> b
2
> let b = 3;    // 再宣言は NG
Uncaught SyntaxError: Identifier 'b' has already been declared

> const c = 1;
> c = 2;        // 再代入は NG
Uncaught TypeError: Assignment to constant variable.

> const c = 3;  // 再宣言も NG
Uncaught SyntaxError: Identifier 'c' has already been declared
```

「`var`、`let`、`const` の再宣言と再代入の挙動のちがいがわかった？」

「`var` は再宣言も再代入も可、`let` は再代入のみ可、`const` はどちらも不可というわけですね」

「そのとおり。後のほうにいくほど安全なコードが書けるよね。特に再宣言は他の多くの言語では許されてない書き方なので、潜在的なバグを生みやすい。これが `var` を使ってはいけない理由のひとつめ」

「なるほど」

「`var` の問題の 2 つめについては、まずその挙動から見てもらおうかな。次のコードを実行してみて」

リスト 8: 02-var/hoisting.js

```
a = 100;
console.log(a);

var a;

$ node 02-javascript/02-var/hoisting.js
100
```

「んん？　どこかおかしいですか？」

「ああ、Ruby や PHP は変数の代入が宣言を兼ねるので、それに慣れてると違和感が少ないのかな。よく見てみて。変数 `a` を宣言する前に代入できてしまってるよね。これが `let` や `const` ならリファレンスエラーになるところだけど、`var` では許されるの。これを『巻き上げ（Hoisting）』というんだけど、これも他の言語ではなかなか見られない仕様なので、開発者がミスリードされてうっかりバグを作り込みかねない[70]。Stack Overflow でもよく初心者の質問のネタになってる」

「そうなんですね」

「`var` の問題の最後は変数のスコープの問題。私としてはこれがいちばんやっかいだと思ってる。とりあえずこのサンプルコードを見てみて。結果はどうなると思う？」

リスト 9: 02-var/scope.js

```
var n = 0;

if (true) {
  var n = 50;
  var m = 100;
  console.log(n);
}

console.log(n);
console.log(m);
```

「下の `console.log(n)` の出力は `0` ですよね。それに最後の行は `m` を参照できなくてエラーになるんじゃないでしょうか。だから `50`、`0` ときて最後はリファレンスエラーになると思います」

「ふふふ、じゃあ実行して確かめてみようか」

```
$ node 02-javascript/02-var/scope.js
50
50
100
```

「えっ。`50`、`50`、`100` ……？」

「まあ奇妙に思うよね。なぜこんな挙動になるかというと `var` で定義された変数のスコープって関数単位なんだよ。だから制御構文のブロックをすり抜けてしまう。いっぽう Ruby を始めとする大

[70] 「Hoisting (巻き上げ、ホイスティング) | MDN」
https://developer.mozilla.org/ja/docs/Glossary/Hoisting

多数の言語の変数スコープはブロック単位だよね。秋谷さんが想定した挙動は、ブロックスコープを前提としたものだった。

それじゃ、このコードの `var` をすべて `const` に書き換えて実行し直してみて。最初に秋谷さんが言ってくれた通りの挙動になったでしょう？」

「あ、ほんとだ」

「他の言語に慣れた大半の開発者にはブロックスコープのほうが直感的だよね。だから `var` の関数スコープは評判が悪かった。ES2015 で `const` と `let` が導入されたときブロックスコープになったのは当然の流れだったといえる。

以上の 3 つが `var` を使うことを私が許さない理由。どれもこれもバグの原因になりかねないでしょ。納得してもらえた？」

「はい、納得しました。こういうことでしたらもちろん、もう `var` は使いません！」

「うん、素直でよろしい。ただ `var` だけじゃなく、意図しない値の上書きもバグの原因になるので `let` も気軽に使わないようにね。あくまで `const` が第一選択肢で、どうしても再代入が必要なときだけ `let` にするってことを徹底してほしいの」

「了解です！」

2-3.　JavaScript のデータ型

JavaScript におけるプリミティブ型

「次は JavaScript のデータ型について」

「ん？　JavaScript って Ruby とかと同じく型がない言語なんじゃなかったでしたっけ」

「その『型がある／ない』という言い方は誤解を招くので使ってほしくないんだよね。JavaScript も Ruby も**『静的型付け言語』**ではないけど**『動的型付け言語』**であって、ちゃんと型を持ってるよ」

「……うーん。その『静的型付け』と『動的型付け』って、何がどうちがうんですか？」

「まず両者ともデータ値そのものに型があることは共通してる。ちがうのは静的型付け言語は変数や、関数の引数および戻り値の型がプログラムの実行前にあらかじめ決まってなければいけないのに対して、動的型付け言語ではそれらが実行時の値によって文字通り**動的に**変化するということ。静的型付け言語である TypeScript と挙動のちがいを比べてみるとわかりやすいかな」

著者紹介

大岡由佳（おおおか・ゆか）

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

趣味は漫画読み、ホームシアターでの映画鑑賞、クラシックバレエ。

Twitter アカウントは `@oukayuka`。

黒木めぐみ（くろき・めぐみ）

漫画家、イラストレーター。

「りあクト！」シリーズの表紙イラストを一貫して担当。

メールマガジン登録のご案内

くるみ割り書房 BOOTH ページ

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

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

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

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

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

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

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

（2020 年 12 月 26 日発行／ 202p ／￥ 1,200）

《第二部 目次》

りあクト! TypeScript で始めるつらくない React 開発 第 3.1 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 3.1 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 12 月 26 日発行／ 214p ／￥ 1,200）

《第三部 目次》

りあクト! TypeScript で極める現場の React 開発

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

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

（2019 年 4 月 14 日発行／ 92p ／￥ 1,000）

《目次》

りあクト! Fiebase で始めるサーバーレス React 開発

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

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

（2019 年 9 月 22 日発行／ 136p ／￥ 1,500）

《目次》

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

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

著者　大岡由佳

印刷・製本　日光企画