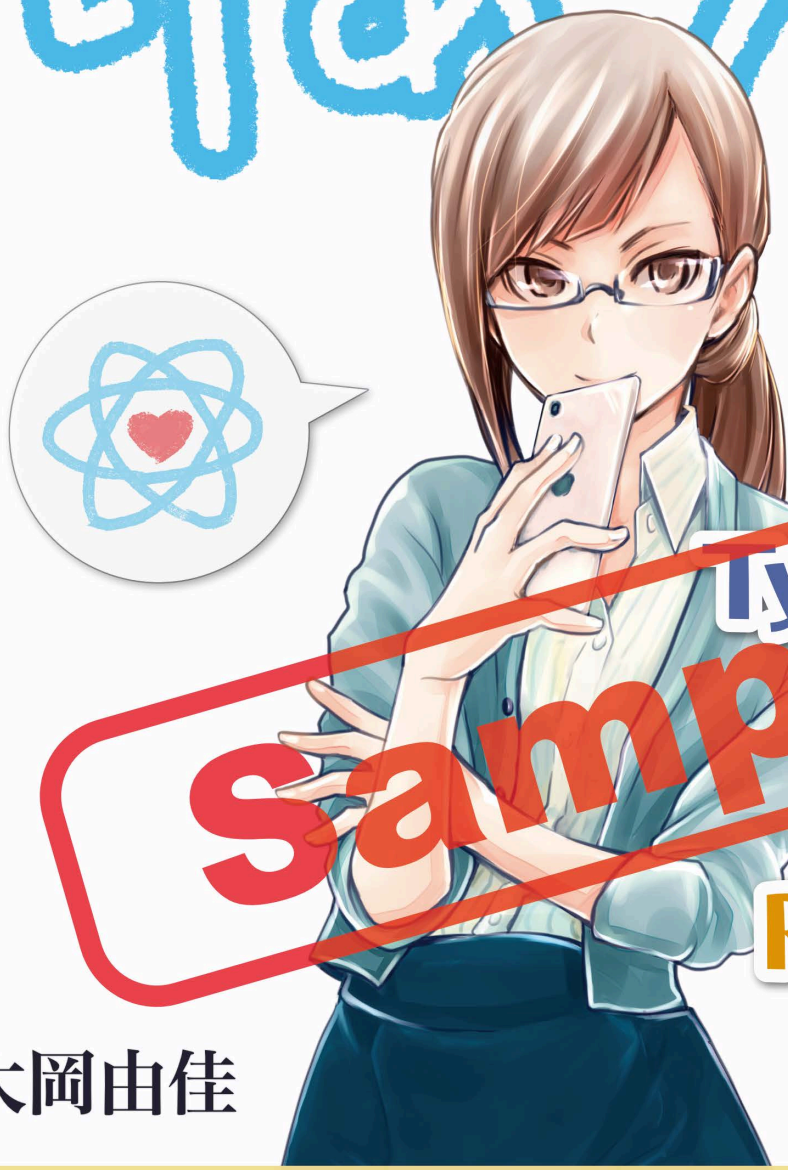
りあクト！



第4版

TypeScriptで始めるつらくないReact開発

① 言語・環境編

大岡由佳

最新ツールVite、ES2022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 2.5万部 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」

BOOTH 技術書 前版 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参加、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3巻）

りあクト! TypeScript で始める
つらくない React 開発 第 4 版
【①言語・環境編】

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。「① 言語・環境編」「② React 基礎編」「③ React 応用編」の全 3 巻構成となっており、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。

『りあクト！ TypeScript で始めるつらくない React 開発』は初版が 2018 年 10 月に発行、国内最大の技術同人誌即売イベント「技術書典 5」にて頒布されました。その後、版を重ねて 2022 年 8 月時点でシリーズ累計 2 万 5 千部以上という異例の売り上げを記録、同人誌の枠を超えて業界内で広く読まれています。特にインターネットサービス企業で働く現場のエンジニアの方々からの反響が大きく、また Chatwork 株式会社様には 2 年連続でインターンシップ用教材としてご採用いただきました。

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、作中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がありながら React にさわるのは初めてという職業エンジニアの方です。そういった方々が React によるアプリケーション開発の現場に参加したとき、可及的速やかに戦力になれるようにすることを目的として、本作は書かれています。それにあたり筆者は、新人が現場の戦力になるために喫緊に必要なのは次の 3 つの力だと考えます。

- 既存のコードがどうやって動いているのかを読み解く力
- 既存のコードがなぜそのように書かれているのか、その意図を理解する力
- 可読性とメンテナンス性を担保したコードを書く力

この 3 つの力を身につけるには React とそれを支える技術の原理原則を理解しておく必要があります。よって本作は何より「理屈」を重視し、かなり丁寧にそれを説明しています。ですので、「理屈よりもとりあえず手を動かしたい」「とにかく動けばいいので、コードがきれいかどうかは気にしない」といった方は本作には向きません。

またプログラミングそのものが初心者の方には、本作は難易度が高いと思われます。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内容に限定しており、基本的な概念や文法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、他の入門書の『JavaScript Primer』[1] を先に読むことをおすすめします。なお TypeScript については初めての方でも、本書「① 言語・環境編」に掲載されている内容で入門から実践まで必要な知識を身につけられます。

なお本作は、React 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 を始めとしたフロントエンド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。また各ライブラリの比較情報や、React 18 における新機能についての解説なども読み応えがあると思います。

文章のスタイル

初版を書き始めたとき、読者がうわべだけの理解に終わらず技術の本質までを理解できるようになるにはどうすればいいか、試行錯誤を繰り返しました。そうして行き着いたのが、シニアエンジニアが新人エンジニアにマンツーマンでを教えていく、すべての解説をその 2 人の会話文の中に収めるスタイルです。似たような技術書は他にもあると思われるかもしれませんが、会話文は各章の導入の部分のみというパターンが多く、ほぼすべてが会話文というのは実はかなりめずらしいスタイルです。

当初は筆者も一人称で普通の技術解説を書いていたのですが、そうしているとついつい無意識の内に自分をごまかしがちになります。なぜそこはそのような仕様になっていてそのように記述するのか、自分で深く理解しないままそれが決まりだからと流してしまう。途中まで書いていて、これではいけないと筆を止めました。そして少しでも引っかかりを感じたなら、そこで自分の中にいたまだ何も知らなかったころの新人ちゃんに、遠慮なく自由に質問させてみたのです。筆者はそれにほとんど答えられず困り果てました。そして実際に現場でコードを書いていたはずの自分の理解度はこんなに浅かったのかと思い知らされました。そこで新人ちゃんの疑問を解消するべく、納得がいくまで情報を調べ、本当にわかったと実感できるまで考え抜きました。

この自問自答を文章に落とし込んだのが本作のスタイルです。React で開発していると「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。初心者ならなおさらでしょう。本作では、そこに新人エンジニアが都度「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ます。他の技

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

術書と比べると冗長かもしれませんが、今版まで続くこのスタイルは読みやすくわかりやすいと読者にとても好評です。

React の本なのに、なぜ React の解説がなかなか始まらないのか

「React の本なのに、React の解説がなかなか始まらない」とは、前版までの読者の少なくない感想です。実際、最初の 1 章でいちおう「Hello, World!」だけはやりますが、その後は JavaScript と TypeScript、さらに関数型プログラミングの解説が続き、「① 言語・環境編」は内容のほとんどがそれらで占められていて React の出番はあまりありません。意図的にこのような構成にしているのですが、これまでは説明不足だったと思うので、今版では最初にその理由を説明しておきます。

今般多くのアプリケーションフレームワークはユーザーの間口を広げるべくかなり敷居が下げられており、その使用言語の知識があまりなくてもなんとなく雰囲気でアプリケーションが作れてしまいます。ところが React についてはその限りではありません。フレームワークではなく UI ライブラリを自称していることもあり、JavaScript の力をフルに活用するようになっていて、JavaScript 初心者には全く歯が立ちません。公式ドキュメントでさえ、JavaScript と同時に習得は難しいので先にそちらを学んできてくださいと断っているほどです[2]。

さらに React は UI を「宣言的」に構築するために作られたプロダクトであり、その実現のために関数型プログラミングが多く用いられます。宣言的であるとはどういうことかや、関数型プログラミングについて基本的なことを知っていなければ、「React らしい」コードを書くことは不可能です。

また現在において、フロントエンド開発者の大半が JavaScript コードをそのまま書くのではなく、TypeScript でアプリケーション開発をしています。しかし React 本体は静的型チェックに同じ Meta 社製の Flow を用いて開発されており、TypeScript での型宣言についてはボランティアにその作業を頼っています。そのような事情の中、React の公式ドキュメントには TypeScript による開発の情報がほとんどなく、また第三者から発信される情報も散発的なものにとどまっています。実際には React によるアプリケーション開発はかなりの割合が TypeScript で行われているため、現場の開発者の多くは情報不足のなか手探りでコードを書くことをしいられています。

以上のような事情により、JavaScript および TypeScript、関数型プログラミングについてちゃんとした知識がなければ React でまともなコードを書くことができません。本作の読者には、React のコードを書

---

[2] 「JavaScript 資料 - Getting Started–React」
https://ja.reactjs.org/docs/getting-started.html#javascript-resources

く前にまずそれらについてしっかりと学んでもらいたいと考え、このような構成としました。

愚者は経験に学び、賢者は歴史に学ぶ

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、その思想と歴史にまで深く踏み込んで解説しています。ここまで思想・歴史についてくわしく書かれてる技術書はめずらしいでしょう。「愚者は経験に学び、賢者は歴史に学ぶ」とはドイツ初代宰相のビスマルクの言葉ですが、てっとりばやく使い方だけを身につけ、実践で失敗を重ねながら開発していけば、そのうち正解に行き着くことはできるでしょう。しかしそこに至る道のりは長く、その後には無数の技術的負債が残ります。

思想・歴史を知ることは、その技術を使ってシステムを正しく設計し、可読性とメンテナンス性および拡張性を担保したコードを書くための近道です。どういう経緯でその技術へと続くトレンドが生まれ、どんな問題を解決するためにその技術が生まれたのか、それを知らずチームが採用しているからという理由だけで使おうとすると、しばしば的外れで意図がわかりづらいコードが量産されてしまいます。

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、そこで多くのひどいコードに出会いました。基本的な JavaScript 力の不足による整理のされてなさもありましたが、加えてそれらはいわゆる「React らしい」コードからほど遠かったのです。そのようになる原因は、React の思想および周辺技術のトレンドに対するメンバーの理解が浅いことにあると筆者には思えました。

それらのとりあえずは動くコードを、可読性とメンテナンス性が担保できるように修正し、さらに本人になぜそれがダメだったのかを説明して納得してもらうためのコストは甚大です。そしてそれをしなければ技術的負債が雪だるま式に増えていき、その内にっちもさっちもいかなくなります。本質を理解していなければ切り貼りで間に合わせのコードを書くことはできても、中長期的にはチームに迷惑をかけてしまうのです。

ひどいコードはそれに関わる人の魂を濁らせます。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 アプリケーション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、本作執筆にあたっての筆者の願いです。

また React は最初から必要なものがひととおりそろっている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りがあり、選定に失敗するとそれもまた後に大きな技術的負債となってしまいかねません。そういった状況は 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）[3]」を磨くには、歴史およびその背景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にそれが筋がよく今後伸びそうなものかどうか、読者自身が判断できるようになっているはずです。

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/klemiwary/Riakuto-StartingReact-ja4.0

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

なおリポジトリのコードと本文に記載されているコードには、ときに差分が存在します。紙面に適切になるよう改行位置を調整したり、各種コメントアウト文を省略するなどによるものですが、そのため行番号が一致しないことがありますので、ご注意ください。

本文および上記リポジトリに掲載しているコードは、読者自身のプログラムやドキュメントに流用してかまいません。それにあたりコードの大部分を転載する場合を除き、筆者に許可を求める必要はありません。出典を明記する必要はありませんが、そうしていただければありがたいです。

---

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

本書について

登場人物の紹介

柴崎雪菜（しばさき ゆきな）

とある都内のインターネットサービスを運営する会社のテックリードのフロントエンドエンジニア。React 歴は 6 年ほど。本格的なフロントエンド開発チームを作るための中核的人材として、今の会社に転職してきた。チームメンバーを集めるため人材採用にも関わり自ら面接も行っていたが、彼女の要求水準の高さもあってなかなか採用に至らない。そこで「自分が React を教えるから他チームのエンジニアを回してほしい」と上層部に要望を出し、社内公募が行われた。

秋谷香苗（あきや かなえ）

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

前版との差分および正誤表

過去の版からの変更点と、本文内の記述内容の誤りや誤植についての正誤表は以下のページに掲載しています。なお、電子書籍版では訂正したものを新バージョンとして随時配信していきます。

- 各版における内容の変更
  https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/blob/main/CHANGELOG.md
- 『りあクト！ TypeScript で始めるつらくない React 開発 第 4 版』正誤表
  https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/blob/main/errat.md

本書内で使用している主なソフトウェアのバージョン

- React（`react`）…… 18.2.0
- React DOM（`react-dom`）…… 18.2.0
- Vite（`vite`）…… 3.0.9
- TypeScript（`typescript`）…… 4.6.4

目次

「② React 基礎編」目次

「③ React 応用編」目次

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開発チームの、……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！」

「そうだ、その前に秋谷さんはいま入社何年めだっけ？　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、それに少しだけ JavaScript を。Rails アプリのフロントエンドに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はないですが、入社前に Java を学んでました」

「なるほど、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[4] 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、謝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[5] 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。時期的にたぶん秋谷さんがさわったのはバージョン 2 系だと思うけど、当時の Vue のアプリケーション設計パターンは MVC に近い MVVM だったからね」

---

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

[5] 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。

「……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、データを保持する Model があって、View のテンプレートに Model が出力する値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 でのアプリケーション設計思想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、まず何をやってもらうかだけど。今日から私がマンツーマンでついて、秋谷さんに React 開発の基本を叩き込みます。そうね、1 週間ほど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」

「ええっ、たったの 1 週間でですか！？　無理無理、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」

「……うーん、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！　柴崎さんがそう言われるなら、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

1-1-1. 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**[6] ね」

「あのー、Node.js ってよく聞くんですけど、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、あまり初心者が納得できるような説明が見つからないよね。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、知ってます」

「Node.js とは簡単に言うと、JavaScript を Ruby や Python と同じように秋谷さんの PC のターミナル上で実行できる環境を提供するソフトウェアなのね。核となる言語処理エンジンとして Google Chrome 用に作られた V8[7] を組み込んでて、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、わかりました。……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/static-html-react.html

```
<!DOCTYPE html>
```

---

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

[7] 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。https://v8.dev/

```
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>React Test</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
        <script  src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script type="text/javascript">
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement('h1', null, 'React works!'));
</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。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[8] や Bundler[9] がやってくれてるようなものですよね」
「そうだね。JavaScript の世界でそれを担ってるのが **npm**[10] 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

[8] 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/

[9] 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/

[10] 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 で公開できる。2020 年の時点において提供するパッケージ数は 130 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/

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 Bower[11] のような Web フロントエンド専用のパッケージ管理システムがあったんだけど、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。だから今の JavaScript では、サーバサイド開発にもフロントエンド開発にも、パッケージ管理は npm を使うようになってる」

「へー、そんな経緯があったんですね」

「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。Node を使うことで開発時に可能になることの主なものを挙げてみよう」

- 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）
- 新しいバージョンの JavaScript や AltJS[12] のコードを古いバージョンの JavaScript コンパイルして、古いブラウザでも動作可能にする
- 開発環境において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、ローカルに開発用のアプリケーションサーバを稼働させることで、動作を検証しやすくし開発効率を高める
- テストツールを用いてユニットテストや E2E テスト[13] を記述・実行する
- ソースコードの静的解析や自動整形を行う

「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[14] を、フロントエ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。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。納得できた？」

「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！」

「そう、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」

---

[11] https://bower.io/

[12] 「Alternative JavaScript」の略で、JavaScript を代替することを目的とした言語。可読性や保守性が高められており、一般的に JavaScript にコンパイルして使うことが想定されている。

[13] 「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。

[14] 昨今は「Digital Tranformation」の意味で使われることが多いが、ここでは「Developer Experience」のこと。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。

1-1-2. Node.js をインストールする

「Node.js のインストールのやりかたは色々あって Mac なら Homebrew[15]、Windows なら winget[16] といったパッケージ管理ツールのリストに 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」

「たしかにそうですね」

「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。この分野で現在メジャーなのは古くからある **nvm**[17]、新興では **fnm**[18] や **Volta**[19] あたりかな」

「Ruby でも同じようなものに RVM[20] と rbenv[21] がありますね。前のチームでは RVM を使ってましたけど」

「ただ、今挙げたようなツールは Node.js を単体でしか管理できない。どうせなら Ruby や Python といった他の言語環境も同じツールで管理できると便利だよね。だから私は **asdf**[22] を使ってる。asdf ではプラグインとして管理できる言語環境が 500 近くもあり、それぞれの最新バージョンへの追随も早い。似たようなツールに anyenv[23] というのがあって私も前はこれを使ってたんだけど、プロジェクトがなかなか更新されなくなってしまたので、後発でより洗練されてる asdf に乗り換えちゃったの」

「なるほど。じゃ私もこれを機会に asdf を入れて、Ruby の管理も RVM から移行しちゃいますね。ちなみに asdf って変な名前ですけど、何かの略なんですか？」

---

[15] 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/

[16] Microsoft が開発しているパッケージ管理システム。正式名称は「Windows Package Manager」。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。長らく実験的な扱いだったが、2021 年 5 月にバージョン 1.0 がリリースされた。
https://docs.microsoft.com/ja-jp/windows/package-manager/winget/

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

[18] https://github.com/Schniz/fnm

[19] https://volta.sh/

[20] https://rvm.io/

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

[22] https://asdf-vm.com/

[23] https://anyenv.github.io/

「うん、まず手元のキーボードを見てみて。それでホームポジションで左手の小指を置いてるところからひとつずつキートップの文字にを読んでみよう」
「A、S、D、F……って、ええっ？　そんな適当な意味なんですか？」
「作者がおちゃめさんなんでしょう。じゃあ実際にインストールしていくよ。私も秋谷さんも Mac だからインストールは簡単。Homebrew がすでに入ってるならこんな感じで OK」

```
$ brew install asdf
$ echo -e "\n. $(brew --prefix asdf)/libexec/asdf.sh" >> ~/.zshrc
$ 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 Z shell が標準になったのでここではこうしてるけど」
「私も Z　shell 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？」
「Windows ユーザーは最近ではさっき挙げた Volta を使ってる人が多いみたい。ただ Node を Windows のネイティブ環境で動かすのはあんまりおすすめできない。Web アプリケーションサーバは UNIX 環境で動作させることが圧倒的に多いので、開発環境もそれに合わせたほうがトラブルが少ないから。でも初心者ほど 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、社外の Ruby の勉強会やカンファレンスでも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 WSL[24] というものが用意されていて、それを使えば Mac とも遜色ない UNIX 環境で Web アプリケーション開発ができるんだよ。Windows のデスクトップで本物の 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

[24] 「Windows SWubsystem for Linux」の略。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/

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、けっこう最近の話だけどね。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。ハードの選定からできるなら、初心者は周りの多くの先輩が使ってる Mac を使ってほしい」

「まあ、そうでしょうね」

「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[25]、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！」

「話を asdf に戻すね。まず asdf でよく使うコマンドを紹介しておこう」

- `asdf plugin list` …… インストール済みのプラグインの一覧を表示
- `asdf plugin list all` …… インストール可能なプラグインの一覧を表示
- `asdf list <PLUGIN_NAME>` …… プラグインのインストール済みのバージョンの一覧を表示
- `asdf list all <PLUGIN_NAME>` …… プラグインのインストール可能なバージョンの一覧を表示
- `asdf plugin add <PLUGIN_NAME>` …… プラグインをインストール
- `asdf plugin update <PLUGIN_NAME>` …… プラグインをアップデート
- `asdf plugin remove <PLUGIN_NAME>` …… プラグインを削除
- `asdf install <PLUGIN_NAME> <VERSION>` …… プラグインパッケージの任意のバージョンをインストール。バージョンの代わりに `latest` を指定すると最新版になる
- `asdf uninstall <PLUGIN_NAME> <VERSION>` …… プラグインパッケージの任意のバージョンをインストール
- `asdf global <PLUGIN_NAME> <VERSION>` …… グローバルに使うパッケージのバージョンを設定
- `asdf local <PLUGIN_NAME> <VERSION>` …… そのディレクトリ配下で使うパッケージのバージョンを設定

「これから実際に asdf で Node.js をインストールしていくんだけど、その前にホームディレクトリに `.default-npm-packages` というファイルを作って次の内容を書き込んでくれる？」

リスト 2: .default-npm-packages

[25] https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/blob/main/extra/build-win-env.md

```
yarn
typescript
ts-node
typesync
npm-check-updates
```

「このファイルは何ですか？」
「Node の任意のバージョンをインストールしたとき、デフォルトでいっしょにインストールされる npm パッケージを登録しておけるの。Ruby なら `.default-gems`、Python なら `.default-python-packages` というファイルで同じことができる。ここに書いた各パッケージの内容については、おいおい説明していくから、今は流しておいて」
「わかりました」
「それじゃ、Node.js をインストールするよ。asdf における Node.js のプラグイン名は `nodejs` なので次のように実行する」

```
$ asdf plugin install nodejs
$ asdf install nodejs latest
$ asdf global nodejs latest
```

「最後のコマンドでグローバルに使う Node のバージョンを指定してる。これがないとインストール自体はできてても Node のコマンドが使えないことになるので、忘れないでね。将来、さらに新しいバージョンにアップデートしたときも、これを忘れると使ってるのは古いバージョンのままになったりするし」
「そういえば RVM にも `rvm --default use <バージョン番号>` 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！」
「ちなみに `global` を `local` に置き換えると、そのディレクトリに `.tool-versions` というファイルができて、その配下において asdf が任意のプラグインでどのバージョンを使うかが設定されるよ。じゃあ、コマンド実行の結果がどうなったか `node --version` を実行して確認してみて」
「v18.7.0 って表示されました。インストール成功です！」
「それから `node` コマンドの使い方を簡単に説明しておくね。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 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 Ctrl+C to abort current expression, Ctrl+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 <JavaScript 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」

---

[26] 「Read-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。

「へー、これよさそうですね。活用させてもらいます」

1-1-3. 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、開発には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』って呼ばれてるけど、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 支援による自動補完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 VS Code っていいんですか？」

「フロントエンド界隈では圧倒的だね。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 2020 年の結果では、そのユーザー使用率は 86 % と他をぶっちぎっての 1 位[28]」

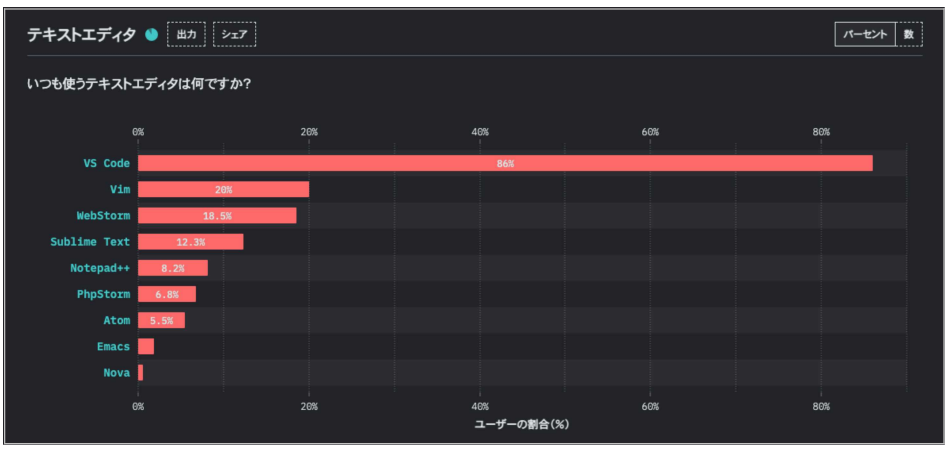


図 1: 2020 年調査の JS 開発者間でのエディタの使用率

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

[28] https://2020.stateofjs.com/ja-JP/other-tools/#text_editors

「さらに React 開発元の Meta[29] でも VS Code がデフォルトの開発環境になってるらしいよ[30]」

「へえええ――」

「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[31] 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[32] 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 ドキュメントで複数人によるドキュメントの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」

「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。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 VSCodeVim[33] とか、もっと軽量でかつマルチカーソル対応の amVim[34] といった拡張を入れると、ほぼ 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」

「わかりました」

「あと VS Code で TypeScript を書いてると、新規に追加したファイルを他から読み込めなくなるとか挙動が怪しくなることがたまにあるので、対処法を教えておくね。

エディタを開き直しても直るんだけど、面倒だよね。そういうときはシステムをリロードすればいい。

---

[29] 正式名称は Meta Platforms。2021 年 10 月に社名を Facebook から変更。本書では以降、略称の「Meta」を用いることにする。

[30] 「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/

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

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

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

[34] https://marketplace.visualstudio.com/items?itemName=auiworks.amvim

⌘ + shift + P（※ Windos では Ctrl + Shift + P）を押してコマンドパレットを開く。そこで `> reload` と入力していくと表示される『Developer: Reload Window（ウィンドウの再読み込み）』を選択して実行する」

「これ、知らないとハマりそうですね……。そんな細かい技まで教えてもらえて助かります！」

■ 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 支援によりコード補完の精度を向上させる
- **Path Intellisense**（`christian-kohler.path-intellisense`）
  …… インポートのパスを自動補完してくれる
- **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
- **Auto Rename Tag**（`formulahendry.auto-rename-tag`）
  …… HTML/XML の開きタグ閉じタグどちらかの要素名を変更すると、対応するタグも同期してくれる
- **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
- **JSON to TS**（`MariusAlchimavicius.json-to-ts`）
  …… JSON オブジェクトを TypeScript のインターフェースに変換してくれる
- **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
- **Remote - WSL**（`ms-vscode-remote.remote-wsl`）
  …… Windows 環境から WSL のファイルシステム上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

1-2. Vite でプロジェクトを作成する

「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 における `rails new` 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存在しない。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する必要があるのね」

「……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、たぶん力尽きちゃいますね」

「だから React のアプリケーションプロジェクトを作成・運用するためのツールが公式やサードパーティから提供されてる。ウチでは Vite[36] というプロダクトを使ってるので、以降はそれを前提に説明していきます」

「びーと？」

「Vue.js と同じ作者 Evan You によるツールでね。vue は view のフランス語なんだけど、vite もフランス語で『速い（quick, fast）』という意味の言葉らしい」

「へー、Vue の作者っておフランス好きなんですね」

「Vite については後でくわしく説明するとして、今はとりあえずそれでプロジェクトを作ってみよう」

```
$ yarn create vite hello-world --template=react-ts
```

「`hello-world` がプロジェクトの名前で、その名前のディレクトリ配下に必要なファイルが置かれるんですね。オプションに指定してる `--template=react-ts` っていうのは？」

「Vite で作成するプロジェクトのテンプレートの種類だね。Vite は React 専用のツールってわけじゃなく

---

[35] インタ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] https://ja.vitejs.dev/

て、Vue.js や他のフレームワークのプロジェクトも作れるの。公式で用意されてるテンプレートは全部で 12 種類あって[37]、今回はそこから React & TypeScript のテンプレートを適用してる。ちなみに `react` っていうテンプレートもあるけど、こっちは素の JavaScript で React アプリを開発するためのものなのでまちがえないようにね」
「なるほど。じゃ、このコマンドを実行しますね。……えっ、1 秒もかからず瞬時に終わっちゃいましたよ？」

```
Done. Now run:

  cd hello-world
  yarn
  yarn dev

✨  Done in 0.68s.
```

「まあ、まだプロジェクトファイルを置いただけだからね。次に、コンソールで指定された通りにプロジェクトのディレクトリに移動して、`yarn` を実行しよう」
「はい、実行っと……。これも数秒で終わりましたね。これは何をやったんですか？」
「これは略式表記なんだけど、正確には `yarn install` というコマンドを実行したのね。このコマンドは `package.json` に記述されている依存パッケージを `node_modules/` 配下にインストールして、さらにインストールされたパッケージのバージョン情報を、その依存関係も含めて `yarn.lock` というファイルに出力するものなの」
「たしかに、プロジェクトのディレクトリに `node_modules/` と `yarn.lock` が追加されてます。`node_modules/` の中にはさらにすごい数のディレクトリができてますね」
「うん。そこに `react/` とか `react-dom/` とかがあるでしょう？　アプリケーションに必要な npm パッケージの実体がそこに置かれるんだけど、それぞれ相互に特定のバージョンに依存してるので、ちょっとバージョンを変更しただけでアプリケーションが動かなくなることがある。だからいつ誰がインストールしてもすべてのパッケージで完全に同じバージョンがインストールされるよう、いったんインストールしたパッケージの依存情報を保存しておくためのファイルが `yarn.lock` ね。
　だから Git のリポジトリに `node_modules/` は入れないけど、`yarn.lock` は必ず入れておくようにする」
「なるほど。作られたプロジェクトの中の `.gitignore` ファイルにもそう設定されてますね」

---

[37] https://github.com/vitejs/vite/tree/main/packages/create-vite

「では最後に `yarn dev` を実行」

```
  VITE v3.0.9  ready in 279 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
```

「これでローカル環境に開発サーバが起動した。表示されてる URL の http://localhost:5173/ をブラウザで開いてみよう」

「おおっ！　Vite と React のロゴマークですね。React のほうはぐるぐる回ってます。かっこいい！」

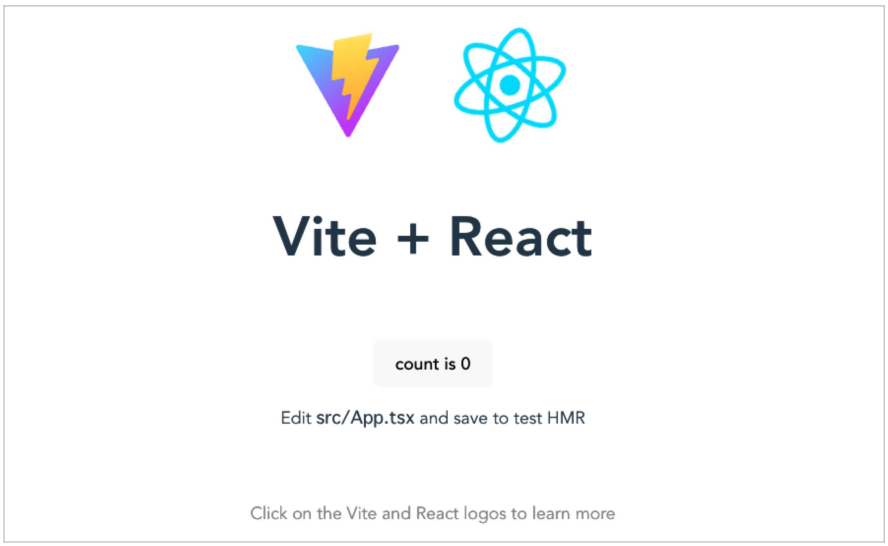


図 2: Vite でプロジェクト作成したデフォルトの画面

「デフォルトだと HTTP サーバが 5173 番ポートで立ち上がる。別のポート番号にしたい場合は `yarn dev -- --port=3000` のように指定してあげるといいよ」

「なんか文章も書いてありますね。『Edit `src/App.tsx` and save to test HMR』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。まずはプロジェクトルートにある `index.html` ファイルを見てみよう」

リスト 3: index.html

```
<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React + TS</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

「ブラウザで画面から『ソースを表示』したものとほぼ同じですね。そっちはヘッダにスクリプトタグで何か追加されてますけど。なるほど、この `index.html` ファイルがアプリケーションの起点になってるわけですか。

でも body タグの中身が `<div id="root"></div>` と空の div タグになってますよね。この実体はどこにあるんでしょうか？」

「その次の行で src/main.tsx ファイルを読み込んでるよね。このファイルの中身を見てみようか」

リスト 4: src/main.tsx

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

ReactDOM.createRoot(document.getElementById('root') as HTMLElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

「`ReactDOM.createRoot()` 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 `<div id="root"></div>` に対応してるわけ。この渡されたドキュメント要素を React を経由して加工した上で DOM に描画し直してるのね」

「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？」

「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[38]』という機能を有効にするためのラッパーなのね。より安全なアプリケーションを構築するためのものなので、今はスルーしてもいいかな」

「なるほど、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、入れ子になってる部品』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[39]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。3 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になってる」

「`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 { useState } from 'react'
import reactLogo from './assets/react.svg'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

  return (
    <div className="App">
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src="/vite.svg" className="logo" alt="Vite logo" />
        </a>
        <a href="https://reactjs.org" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
```

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

[39] 「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。

```
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.tsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </div>
  )
}

export default App
```

「ふーむ、けっこう長いですね」

「カウントボタンを実現するための機能が含まれてるからね。そのあたりは後でじっくり説明するので、今は『Hello, World!』を実現することだけに集中しよう。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。ここでは関数で `App` コンポーネントを定義してるの」

「`function App() {...}` ってとこで、コンポーネントを関数として宣言してるんですね。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 test HMR』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」

「ううん、これは HTML に見えるかもしれないけど、実際には HTML じゃないの。さっき `main.tsx` の `render()` メソッドの引数として渡されてたものも同じ。『JSX[40]』っていう JavaScript の構文を拡張したもので、言ってみれば非公式な JavaScript の一方言みたいなものなの」

「へー、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」

「`main.tsx` も `App.tsx` もファイル名の拡張子が `.tsx` となってるでしょ。これは TypeScript で記述できる JSX のファイルってことで、コンパイルを経て最終的に JavaScript に変換されるのね。

ちょっとブラウザで http://localhost:5173/src/App.tsx にアクセスしてみてくれる？」

「おおー、なんかすごい読みづらい JavaScript のソースが表示されました！　これが `App.tsx` のコンパイ

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

ル後の姿ですか……」

「これで流れはわかっただろうから、画面に表示される内容を『Hello, World!』に書き換えてみよう。`App.tsx` の中身をこんなふうにしてみて」

リスト 6: 書き換え後の App.tsx

```
import reactLogo from './assets/react.svg'
import './App.css'

function App() {
  return (
    <div className="App">
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src="/vite.svg" className="logo" alt="Vite logo" />
        </a>
        <a href="https://reactjs.org" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Hello, World!</h1>
    </div>
  )
}

export default App
```

「おー、バッサリいっちゃうんですね。……はい、書き換え終わりました」

「じゃ、そのままファイルを保存して」

「はい。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!』に変わりました！」

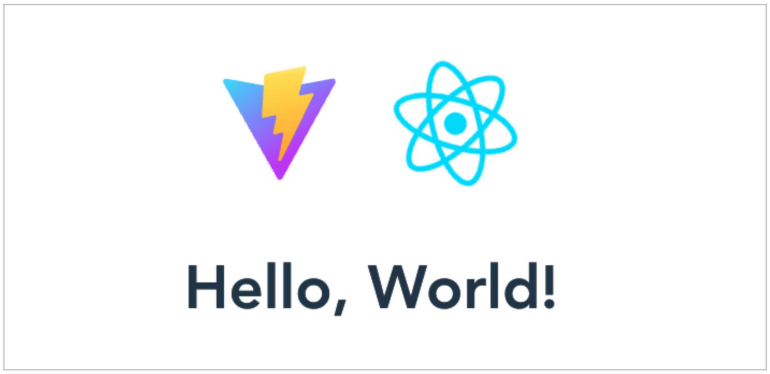


図 3: Hello, World!

「これが HMR（Hot Module Replacement）と呼ばれる機能で、ソースコードを変更・保存するとアプリケーションに即座に反映されるの。これも Vite が実現してくれてるものね。

実際のアプリケーション開発では、こうやってプログラムを追加・変更しながらブラウザで確認していくわけだけど、何か質問ある？」

「はい、プロジェクトにあるファイルがそれぞれどういう役割を担ってるのか知っておきたいです」

「わかった。ざっと説明しておこうか」

表 1: プロジェクトファイルの説明（* は Vite 特有のファイル）

ディレクトリ名／ファイル名	内容
`src/`	アプリケーションのソースコードを置く
`node_modules/`	アプリケーションに必要な npm パッケージが保存されている
`public/`	公開用のアセットファイルを置く
`package.json`	インストールするパッケージの情報などが書かれた設定ファイル
`yarn.lock`	インストールしたパッケージの依存情報が保存されたファイル
`tsconfig.json`	TypeScript をコンパイルするための設定ファイル
`tsconfig.node.json*`	Vite の設定を TypeScript で書くための `tsconfig.json`
`vite.config.ts*`	Vite の設定ファイル
`gitignore`	Git リポジトリに含めないものリスト

「プロジェクト作成ツールに Vite、パッケージ管理に Yarn、開発言語に TypeScript を使っててそれに特有の形式になっているし設定内容によっても変わってくるので、React のアプリケーションが必ずしもこのファイル構成になるわけじゃないことをおぼえておいてね」

「へー、Rails みたいに規約でかっちり決まってるわけじゃないんですね」

「React は Rails みたいにフルスタックのフレームワークじゃないからね。このへんは Vue や Angular といったフロントエンドの他の競合と比べても自由度が高い。ただこれについては一長一短で、色んなやり方が乱立してしまって初心者を混乱させてしまうことにもつながってしまう」

「たしかにチームごとで使ってるツールや開発の思想もちがうでしょうし、チームを移ったら同じ React を使ってるのにファイル構成が別物なのは困りますね」

「でも公式・非公式が入り混じって自由競争の中で多くのユーザーを獲得したものが勝ち残っていく、というのは Web フロントエンドのような変化が激しく未来の不確実性が高い領域ではメリットも大きい文化なんだよね。実際、ウチが Vite を採用したのもつい最近だし」

「えっ、そうだったんですか……。じゃあ私もこれからはそういうカルチャーに慣れていかないといけませんね」

1-3. プロジェクトを管理するためのコマンドやスクリプト

1-3-1. Yarn

「npm は名前の由来である『Node Package Manager』のとおり、Node.js のパッケージ管理をするためのシステムだということは前に話したよね。その npm の公式リポジトリに収納されているパッケージを管理するためのコマンドパッケージが同名の `npm` なんだけど、これは Node.js をインストールするといっしょに入れられる。`npm --help` を実行すると使えるコマンドの一覧が表示されるよ」

「ほんとだ。`npm install` とかが使えるようになってる。あれ？　でもここまで `npm` コマンドは使ってませんよね？」

「うん。JavaScript のパッケージマネージャは npm が先行してたんだけど、その後しばらくして Yarn[41] や pnpm[42] といった後発のプロダクトが出てきて、一定の支持を集めてるの。ウチで使ってるのもこの Yarn ね」

「これまで `yarn create` とか `yarn install` とかのコマンドを実行してましたね。どうして Node 標準の npm じゃなく Yarn を使ってるんですか？」

「npm は Node.js 登場の翌年の 2010 年とかなり早い時期にリリースされたんだけど、Yarn はそれより後の 2016 年に Meta（当時は Facebook）によってリリースされた。開発者たちが npm に感じていた不満を解消するためにベター npm を提供すべく Yarn は作られたの」

「へー、npm と比べて Yarn は何が優れていたんですか？」

「たとえば依存パッケージのバージョン固定ができる `yarn.lock` によるロックファイルのしくみが最初から備わっていた。これは npm がバージョン 5.0.0 で追随して同じことができるようになった。また複数のプロジェクトを単一のリポジトリで管理するための Workspace という機能も Yarn が先行して実装し、後年になって npm が追随した。こんなふうに Yarn が先に実装して普及した機能を npm が後追いするという構図があったの」

「へー」

「またパフォーマンスのよさもウリだね。各プロダクトによるパフォーマンスの比較については、pnpm

---

[41] https://yarnpkg.com/

[42] https://pnpm.io/

の公式サイトに掲載されてるベンチマーク [43] が参考になる。純粋に速度だけで比較すると総合的にはpnpm > Yarn >> npm のようになる。いっぽう、どれだけ開発者に使われているかというトレンドはこんな感じ」

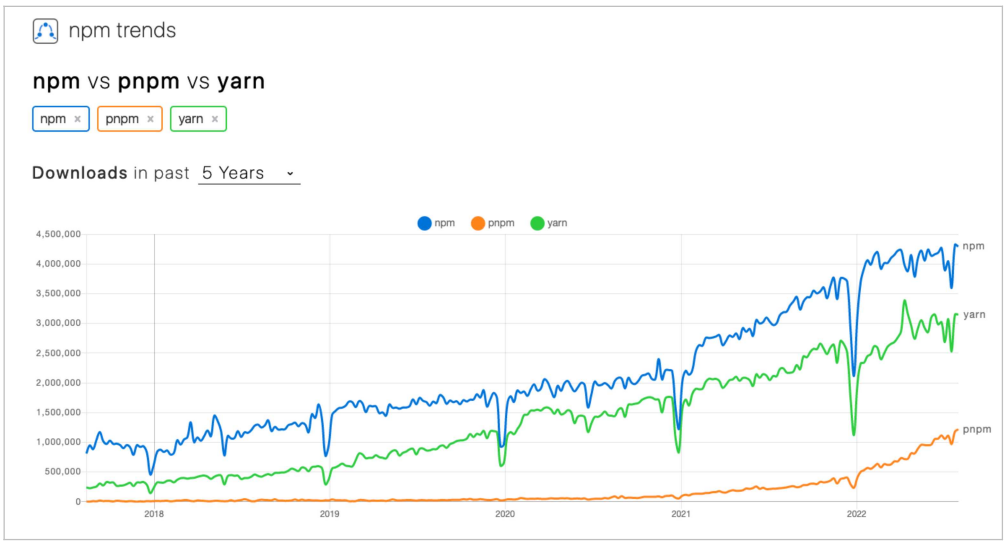


図 4: パッケージマネージャの DL 数比較（2022 年 8 月現在）

「おー、Yarn がけっこう npm に肉薄してますね！」

「Meta がプロデュースしているので React のコミュニティでよく使われてるというのもある。その他にも個人的には `info` コマンドでの情報量が多いこと、場面に応じて `install` や `run` コマンドが省略できることなんかが好みかな。まあ共通するコマンドも多いし、使い勝手もそこまで変わらない。乗り換えコストも高くないので、チームごとに話し合って好きなほうを採用すればいいと思う」

「なるほど」

「ここからは Yarn の使い方を学んでいこう。よく使うコマンドは次のものになる」

- `yarn [install]` …… `package.json` に登録されているパッケージ、およびそれらと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 [run] <SCRIPT_NAME>` …… `package.json` に登録されているスクリプトを実行する

---

[43] 「Benchmarks of JavaScript Package Managers | pnpm」
https://pnpm.io/benchmarks

- `yarn add [-D|--dev] <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新しい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

「`install` と `run` コマンドについては動作が `package.json` の記述に依存する。さっき作成したプロジェクトで実際のファイルを見てみよう」

リスト 7: package.json

```
{
  "name": "hello-world",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "devDependencies": {
    "@types/react": "^18.0.17",
    "@types/react-dom": "^18.0.6",
    "@vitejs/plugin-react": "^2.0.1",
    "typescript": "^4.6.4",
    "vite": "^3.0.9"
  }
}
```

「`scripts` エントリは `run` コマンドに関係してるんだけど、この説明は後回しにさせてね。`dependencies` と `devDependencies` 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るでしょう？　これが `yarn install` を実行したときにインストールされるようになってる」

「エントリが 2 つあるのはどうしてですか？」

「`devDependencies` のほうは、インストールはされるんだけど開発環境でしか有効にならず、本番用にビ

ルドしたファイルには含まれない。主に開発ツールのパッケージなんかがここに入れられる。またTypeScript のコードはビルド時に JavaScript にコンパイルされるので、TypeScript 本体や TypeScript 用の型定義パッケージなんかも `devDependencies` の対象だね」

「ほんとだ。ちなみに `devDependencies` にパッケージをインストールするにはどうするんですか？」

「`yarn add` コマンドでインストールする際、`-D` または `--dev` オプションを指定するの」

「ふむふむ。ところで `remove` や `upgrade` コマンドの際は、そのオプション指定は必要ないんですか？」

「すでにインストールされてるものなら、その指定は意味がないからね。ちなみにパッケージのインストールやアップグレードは、`package.json` のエントリ内容を編集してから `yarn install` 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の範囲が指定されてるわけだけど、これらのパッケージが依存しているパッケージ群が実際にどのバージョンでインストールされるかというのは、その記述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異なることがしばしばある」

「それって何か困るんですか？」

「任意のパッケージのバグや破壊的変更によってバージョンが異なると挙動が変わることもあるし、他の依存しているパッケージのバージョンが変わったことで相性によって挙動がおかしくなることもある。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起こってくるよね」

「たしかにそれは困りますね」

「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」

「それがさっき説明してもらったロックファイルですね」

「そう。これは npm を使ってる場合には `package-lock.json` という名前のファイルになる。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」

「`upgrade` コマンド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。アップグレード後に動かなくな

ったときは、`git restore` でいったん `package.json` や `yarn.lock` の内容を元に戻してから原因を特定しましょう」
「了解です！」

1-3-2. npm スクリプト

「それじゃ `package.json` ファイルに戻って、さっき飛ばした `scripts` エントリを見てくれる？」

```
  ⋮
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview"
  },
  ⋮
```

「開発サーバを立ち上げたときに実行したコマンドをおぼえてる？ `yarn help` で Yarn のサブコマンド一覧が見られるけど、その中には `dev` なんてコマンドはない。なのに `yarn dev` が実行できるのはなぜか。これは npm の **scripts**[44] という機能を使ってるから。`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 Yarn なら `yarn run <COMMAND_NAME>`、npm なら `npm run <COMMAND_NAME>` でそれを実行できるの。Yarn は `install` と `run` を省略できるので、あたかも Yarn コマンドのように `dev` が実行できたわけ」
「なるほど」
「定義されてる `dev` のエントリに注目して。プロパティが `vite` ってなってるでしょ。npm スクリプトを実行する際には `<PROJECT_ROOT>/node_modules/.bin/` にパスが通った状態になっているので、実際にはそこにインストールされている Vite のパッケージコマンド `vite` が実行される」
「へー。じゃ、`scripts` に自分で適当にエントリを追加していけば、それを npm スクリプトとして実行できるってことですか？」
「そのとおり。後々、ここに構文チェックやフォーマッティングのためのエントリを追加していく予定。ただし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あるので、気をつけないといけ

[44] 「scripts | npm Docs」 （※以前は「npm-scripts」と呼ばれていたが現在、公式はこの呼称を使っていない）
https://docs.npmjs.com/cli/v8/using-npm/scripts

ない。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 コマンドや 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。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
- `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

「あれ？　開発用サーバの起動って `dev` で登録されてましたよね。`start` じゃないんですか？」
「そうなんだよね。Vite のテンプレートはなぜかこの作法を無視してる。だからこの後で説明する `start` の前後で自動的に実行させたいスクリプトを登録したいときは、そのために `dev` を `start` に書き換える必要が出てくるの」
「ふーん、変な話ですねえ」
「そして npm のコマンドおよびスクリプト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ほどだけど、法則としてはコマンド名やスクリプト名の前に `pre` か `post` がつくものが大半。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のものくらいかな」

- `preinstall`, `postinstall` …… パッケージがインストールされる前後に実行される
- `preuninstall`, `postuninstall` …… パッケージがアンインストールされる前後に実行される
- `prestart`, `poststart` …… `start` スクリプトが走る前後に実行される
- `pretest`, `posttest` …… `test` スクリプトが走る前後に実行される
- `prepare` …… `npm install` コマンドの場合には、インストール系の処理が終わった一番最後に実行される

「`prepare` はちょっと特殊で、`npm install` や `npm publish` など複数のコマンドのライフサイクルに組み込まれてるの。さらにこの前後の `preprepare` や `postprepare` なんてのがあったりする。わからなかったら公式ドキュメント[45] を参照してね」
「はい」
「それから、`yarn dev` 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

[45] 「Life Cycle Operation Order - scripts | npm Docs」
https://docs.npmjs.com/cli/v8/using-npm/scripts#life-cycle-operation-order

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、今度からそうします！」

第 2 章 ライトでディープな JavaScript の世界

2-1. あらためて JavaScript ってどんな言語？

2-1-1. 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。Rails を使いながら Ruby のことをあまり知らなかったり、Vue を使いながら JavaScript の知識がおろそかだったりする人がたまにいるけど、React ではそうはいかない。

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 JavaScript を書くことがあったので。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簡単な 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。クラスもなんか簡易的だし、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今もいるみたいだからね。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 を悪く言う開発者は今でも多いけど、世界的にはずっと前から再評価されていて好きな言語として挙げる人も多い」

「そうなんですか？」

「うん。GitHub での利用状況や Stack Overflow で話題になった件数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 は最初の 2012 年からほぼずっとトップにあり続けてて、

その座を譲ったのはわずか 1 回しかない[46]。

使われてる数だけじゃ納得できかもしれないだろうから、次は開発者による好き嫌いの調査を見てみよう。Stack Overflow による Developer Survey 2022[47] での『Most Wanted Languages（もっとも求められている言語)』ランキングでは、JavaScript は 5 位に入ってる。同調査の『Loved vs. Dreaded（愛してる VS 嫌ってる)』グラフでも JavaScript は 61.5 % 対 38.5 % で Loved 派のほうが大きく上回ってる」

「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
- 構造体ともクラスインスタンスとも異なる、シンプルで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なじ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 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

---

[46] 「RedMonk Top 20 Languages Over Time: June 2021」
https://redmonk.com/rstephens/2021/08/05/top-20-june-2021/

[47] 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。2022 年は約 70,000 人からの回答があった。
https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-language-want

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 1995 年ごろ当時 Netscape の社員で、静的スコープと第一級関数をサポートする Scheme[48] とプロトタイプベースのオブジェクト指向言語 Self[49] 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』と改名された。その過程で 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。そして彼にその改変を指示して、ついには言語の名前も『JavaScript』に変えさせたの[50]」

「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。ただそれが改名と構文改変のおかげだったのかは、今となってはわからない。名前については個人的に 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』[51] だという人もいるけど、言い得て妙だと思う」

「ええ〜、JavaScript って Java じゃなくて 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？　どうりで関数周りの扱いが意味不明だと思ってました」

「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だよね。それでちょっと複雑なことをしようとすると、なじみのない概念に突き当たって困惑する。JavaScript を学ぶ上で、C や Java の延長にある言語という先入観はむしろ障害になるかもね」

「……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……」

---

[48] 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。

[49] 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。

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

[51] 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

2-1-2. 年々進化していく JavaScript

「Rails での JavaScript の使い方をちょっと見てみたけど、最近の Rails ではモダンな JavaScript がそのまま普通に書けるようになってるみたいだね」

「モダンな JavaScript ？　JavaScript にモダンもクラシックもあるんですか？」

「なるほど、そこの区別からなわけね。JavaScript の言語仕様にもバージョンがあって、おおまかに ES2015 以降がモダン JavaScript といわれることが多いの」

「いーえすにーまるいちご？」

「正式名は『ECMAScript 2015』ね。『JavaScript』というのはさっきも言ったように元々 Netscape 社が開発し、それを現在は 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[52] という業界団体が定めているの標準仕様で、1997 年に初版が公開された。実は ECMAScript の仕様に準拠している言語は JavaScript だけじゃなくて、Adobe の Flash　Player 上で動作するアプリケーションを開発するための ActionScript という言語もその実装だった。Flash 自体がもうなくなっちゃったけどね」

「Flash、なくなっちゃいましたねえ……」

「そしてさっき言及した ES2015 というのは、ECMAScript 仕様の第 6 版で 2015 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 もコンパイルターゲットをこの ES5 にしてる」

「おるとじぇーえす？」

「『Alternative　JavaScript』の略で、JavaScript の代替言語のこと。JavaScript を使いやすくするための仕様を追加・変更したもので、最終的に JavaScript コードにコンパイルされる形式のものが多い。知ってるかどうかわからないけど、昔の Rails は CoffeeScript[53] っていう JavaScript に Ruby と Python を足して 3 で割ったような言語がデフォルトで組み込まれてたんだよ」

「CoffeeScript って名前だけは聞いたことはありますね。使ったことはないですけど」

---

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

[53] 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 6 系まで、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た。
https://coffeescript.org/

「ECMAScript は近年、ほぼ毎年のように新しい仕様が公表されてる。2022 年 8 月現在での最新版は ES2022、つまり『ECMAScript 2022』で 2022 年の 6 月に正式仕様がリリースされた。ほぼ毎年、6 月には最新版が出てて、今年も順当に出たみたいね」

「ECMAScript って年号をバージョン表記の代わりに使うんですね。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。それまでは『ES5（ECMASCript 5th Edition)』のようにバージョン番号で呼ばれてた。『ES6』のように表記している記事もないわけじゃないけど、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のかというと、最初に言ったように 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ったから。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」

「……そうなんですね、全然区別してなかったです」

「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 の改訂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」

「ふむふむ」

「でもそれらの『悪いパーツ』や『ひどいパーツ』[54] 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 Vite 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 になってる」

「日々、追随していってるんですね」

「それらを踏まえた上で、ここからは JavaScript の基本的な構文は押さえているものとして React と

---

[54] 『JavaScript: The Good Parts —「よいパーツ」によるベストプラクティス』p.117

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 JavaScript を説明していくよ」
「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。Rails のサンプルコードを見ると変数の宣言にカジュアルに `var` キーワードが使われてるけど、これは金輪際もう使ってはいけません。禁止します」
「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 3 つの問題を抱えてるの」

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
> 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
```

	再代入	再宣言
var	☑	☑
let	☑	×
const	×	×

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

「はい、完璧です！」

「ちなみに `var` は『variable（変数）』、`const` は『constant（定数）』が由来。`let` は『Let it be（あるがままに）』ってことだろうね。ところで Rust でも同様に可変変数の宣言に `let` を使うけど、Swift だと `let` は定数の宣言だったりと、このへんは言語によってバラバラなんだよね」

「へー、おもしろいですね」

「`var` < `let` < `const` と後のものを使ったほうがより安全なコードが書ける。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『変数の参照が巻き上げられる』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-vars/hoisting.js

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

var a;



$ node 02-javascript/02-vars/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` なら参照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）[55]』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。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『スコープ単位が関数』。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-vars/scope.js

```
var n = 0;

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

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

「最後から 2 行めの `console.log(n)` の出力は `0` ですよね。次の行は `m` を参照できないんじゃないでしょうか。だから `50`、`0` ときて最後は参照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-vars/scope.js
50
50
100
```

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

「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 JavaScript では `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 `if` のような制御構文のブロックをすり抜けてしまう。いっぽう Ruby を始めとする大多数の言語の変数スコープはブロック単位だよね。最初に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」
「そうです」
「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？」
「……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

2-3-1. 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。異なる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」

```
$ node
> let num = 100;
> num = 'foo';
'foo'

$ ts-node
> let num: number = 100;
> num = 200;
200
> num = 'foo';
<repl>.ts:6:1 - error TS2322: Type 'string' is not assignable to type 'number'.

 num = 'foo'
 ~~~
```

「前者が動的型付け言語である JavaScript の挙動。これは Ruby と同じなのでわかると思うけど、一度宣言した変数に異なるデータ型の値を入れ直すことができる。後者が静的型付け言語の TypeScript での挙動で、変数を `let` で宣言していてもデータ型が異なると値の再代入はコンパイルエラーになって許されない」

「なるほど、理解できました。軽々しく『型がない』とか言っちゃいけないんですね……」

「うん、わかってもらえたようね。そして JavaScript では Java と同様、データ型が**『プリミティブ型』**と**『オブジェクト型』**の 2 つに大別される[56]。値がプリミティブであるというのは、それがオブジェクトではない、 インスタンスメソッドを持たないデータだということ。JavaScript のプリミティブ型は今のところ次の 7 種類がある」

- **Boolean 型**（論理型）
  …… `true` および `false` の 2 つの真偽値を扱うデータ型。
- **Number 型**（数値型）
  …… 数値を扱うためのデータ型。他の多くの言語と異なり、整数も小数も同じ Number 型になる。扱うこと

[56] 「JavaScript のデータ型とデータ構造 - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Data_structures

ができる最大値は 253-1（9,007,199,254,740,991 ≒ 9 千兆）。

- **BigInt 型**（長整数型）
  …… Number 型では扱いきれない大きな数値を扱うためのデータ型。ES2020 で追加された。Number 型と互換性がなく、相互に代入や計算、等値比較などは行えない。

- **String 型**（文字列型）
  …… 文字列（テキストを表す連続した文字）を扱うためのデータ型。

- **Symbol 型**
  ……「シンボル値」という固有の識別子を表現する値。ES2015 から導入された。`Symbol()` 関数を呼びだすことで動的に生成されるが、基本的に同じシンボル値を後から作成できない。オブジェクトのプロパティキーとして使用可能。

- **Null 型**
  …… プリミティブ値 `null` は何のデータも存在しない状態を明示的に表す。

- **Undefined 型**
  …… プリミティブ値 `undefined` は宣言のみが行われた変数や、オブジェクト内の存在しないプロパティへのアクセスに割り当てられる。他の多くの言語と異なり、`null` と明確に区別される。

「これらの内 BigInt 型は扱える環境がまだ限定的だったり、Symbol 型は JSON でのパースができなかったりと取り扱いが不便なのであまり活用の場面がない。だから私たち開発者がよく使うのはこれらを除いた、あとの 5 種類ね」
「なるほど。これらの型で値の真偽がどうなってるか教えてもらえますか？」
「いい質問だね。falsy なのがそれぞれ `false`、`0`、`NaN`、`''`（＝空文字）、`null`、`undefined` で、それ以外は全部 truthy になるよ」
「ちょっと待ってください。`NaN` って？」
「Number 型でありながら、数値ではない（Not a Number）ことを示す値のこと。発生条件としては次のものがある」

- `0` 同士の除算
- `NaN` を含んだ演算
- その他の無効な演算および処理

「最後の『その他の無効な演算および処理』とは具体的には？」

「たとえばこんなものが挙げられる」

```
> Math.sqrt(-1)     // -1 の平方根
NaN
> Infinity * 0      // 無限大と 0 の乗算
NaN
> parseInt('foo')   // 数字以外の文字列を数値としてパース
NaN
```

「ふむふむ、たしかにこれらは数値で表現できませんね」

2-3-2. プリミティブ値のリテラルとラッパーオブジェクト

「ところでさっき柴崎さん、プリミティブ型の定義の中で『インスタンスメソッドを持たない』っていいましたよね。でも手元で確認してみたら、文字列とか普通にメソッドが使えちゃえますよ。本当はJavaScript も Ruby のようにすべてのデータがオブジェクトなんじゃないんですか？」

```
> 'Serval lives in savanna'.replace('savanna', 'jungle');
'Serval lives in jungle'
```

「うーん、その挙動に気づいちゃったか。これを理解するにはちょっと入り組んだ説明が必要になるのでよく聞いててね。まずプリミティブ型の値を定義するのには、通常『**リテラル**（Literal）[57]』を使う。literal は『文字通りの』という意味の言葉で、プログラミング言語においてはソースコードに数値や文字列をベタ書きしてその値を表現する式であることからこう呼ばれるのね。各プリミティブ型に用意されてるリテラルは次のとおり」

- Boolean 型
  …… `true` と `false` の 2 つの**真偽値リテラル**がある。
- Number 型
  …… `36` や `-9` のように数字を記述する**数値リテラル**。先頭に `0x` をつけることで 16 進数、`0o` をつけるこ

[57] 「文法とデータ型 - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Grammar_and_types

とで 8 進数、`0b` をつけることで 2 進数が表現される。他に `3.14` や `2.1e8` などの形式で表現する**浮動小数点リテラル**がある。また `1_000_000` のように途中に挿入された非連続のアンダースコアは無視される。

- BigInt 型
  …… `100n` のように整数の後ろに `n` をつけて表現する**長整数値リテラル**。
- String 型
  …… シングルクォート `'` またはダブルクォート `"` で囲まれた**文字列リテラル**。バッククォート `` ` `` を用いると、改行を含む複数行テキストや `${}` による式の展開が可能な**テンプレートリテラル**[58] になる。
- Null 型
  …… **Null リテラル** である `null` は、プリミティブ値 `null` を返す。

「ちなみに、ややこしいけど `undefined` はリテラルじゃなく、プリミティブ値 `undefined` が格納されている `undefined` という名前のグローバル変数ね。変数といっても書き換え不可だけど」
「へー、ここまで特に意識せずに書いてましたけど、これらはそういう構文として規定されてたんですね」
「うん。それでね、ここに挙げたリテラルを用いて定義された値は、まちがいなくプリミティブ型であってオブジェクトではないの。だからさっき秋谷さんが挙げてくれたサンプルの文字列もプリミティブ型なわけ」
「ならどうしてそれらの値から直接メソッドが生えるんですか？」
「まず `null` と `undefined` を除くすべてのプリミティブ型には、それらの値を抱合する『**ラッパーオブジェクト**（Wrapper Object）』というものが存在してる。String 型なら `String`[59]、Number 型なら `Number`[60] がそれに相当する。これらは JavaScript の言語処理系に標準組み込みオブジェクト [61] として備わってる。

ただ同じ値を表現するものであっても、プリミティブ値とそのラッパーオブジェクトは等価とはならない。ラッパーオブジェクトに備わってる `valueOf()` というメソッドが返す値がそのプリミティブ値に対応

---

[58] 「テンプレート文字列 - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Template_literals

[59] 「String - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/String

[60] 「Number - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Number

[61] 実行環境にあらかじめ組み込まれており明示的にインポートすることなく使うことができるオブジェクトのうち、ECMAScript の標準仕様として規定され用意されているオブジェクト。Ruby の組み込みライブラリ、Java のコアライブラリに相当する。なお JavaScript では `parseInt()` や `encodeURI()` といったグローバル関数も標準組み込みオブジェクトに含まれる。
https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects

する。実際の挙動で示すとこうなるね」

```
> const str1 = 'Serval';
> const str2 = new String('Serval');
> str1 === str2
false
> str1 === str2.valueOf()
true
```

「そして JavaScript には、プリミティブ型の値に対してアクセスするとき、その対応するラッパーオブジェクトに自動変換するという仕様があるのね。だからさっき秋谷さんが挙げてくれた例は、内部でこう変換されてる」

```
'Serval lives in savanna'.replace('savanna', 'jungle');
 ↓
(new String('Serval lives in savanna')).replace('savanna', 'jungle');
```

「`replace()` が実行できてるのは、変換された `String` オブジェクトのインスタンスメソッドを実行してたわけね」
「……そういうことだったんですか。なるほど、納得しました。
　ただこの挙動、たしかに便利なんでしょうけどまぎらわしいですよね。開発者としては、常に明示的にこのラッパーオブジェクトのインスタンスを生成するようにしたほうがよくないですか？」
「……秋谷さんって変に几帳面なところがあるんだね。でも JavaScript ではこうやって必要に応じてプリミティブ値が自動的にオブジェクトに変換されるので、わざわざラッパーオブジェクトのインスタンスを生成する必要はないよ。値を取りだすのにいちいち `valueOf()` を呼ぶのも面倒だし、誰もそんなことやってないもの」
「ふーむ、そういうもんですか……」

2-3-3. オブジェクト型とそのリテラル

「プリミティブ型だけじゃなく、オブジェクト型にもリテラルを持つものがいくつかあるので紹介しておくね」

- **配列リテラル**

  …… `[1, 2, 3]` の形式で記述する。`[]` は空の配列を示す。`arr[n]` という構文で n + 1 番めの要素にアクセスできる。`Array` オブジェクト[62] のインスタンスとして生成される。

- **オブジェクトリテラル**

  …… `{ key: value }` の形式で記述する。キーには文字列またはシンボルが用いられる（なお数値を指定すると、自動的に文字列に変換される）。任意のプロパティ値にアクセスするには `obj[key]` または `obj.key` の 2 つの構文が利用できる。`Object` オブジェクト[63] のインスタンスとして生成される。

- **正規表現リテラル**

  …… `/pattern/flags` の形式で記述する。正規表現パターンでの特殊文字の使い方は、他の言語とほぼ共通[64]。`RegExp` オブジェクト[65] のインスタンスとして生成される。

「ここは Ruby とだいたい共通してるので抵抗ないですね」

「Ruby と同じというと、配列だけじゃなく文字列も `str[n]` の形でインデックスを指定して個々の文字にアクセスできるよ。あと Ruby は `arr[n]` で `n` にマイナスの値を与えることで末尾から n + 1 番めの要素にアクセスできるけど、その仕様は JavaScript にはない。でも ES2022 から `Array` と `String` オブジェクトに `at` メソッドが追加されて、`arr.at(-1)` のように書くことで末尾の要素を取得できるようになった」

「へええ。でもそのために要素アクセスの仕様を変えるんじゃなく新しいメソッドを追加するなんて、JavaScript って本当に既存の仕様を上書き変更するのを嫌うんですね」

「そうだねえ。それで、これらオブジェクトのためのリテラルは、それぞれ対応する標準組み込みオブジェクトのインスタンス生成式とほぼ同じ挙動になってるのね」

```
> const arr1 = [1, 2, 3];
> const arr2 = new Array(1, 2, 3);
> console.log(arr1, arr2);
[1, 2, 3] [1, 2, 3]
```

---

[62] 「Array - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Array

[63] 「Object - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Object

[64] 「正規表現 - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular_Expressions

[65] 「RegExp - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/RegExp

```
> const obj1 = {};
> const obj2 = new Object();
> console.log(obj1, obj2);
{} {}

> const regex1 = /pattern/i;
> const regex2 = new RegExp('pattern', 'i');
> console.log(regex1, regex2);
/pattern/i /pattern/i
```

「ふむふむ」

「ただし `Array` のコンストラクタは引数がひとつの数値だった場合、空の要素をその数だけ持った配列が生成されるというおかしな挙動があったりする。また `Object` のコンストラクタは結局、引数にオブジェクトリテラルを使うしかないので使う意味がない。だから動的にマッチングパターンを記述できる `RegExp` 以外は、これらのコンストラクタの出番はほとんどないね」

「なるほど、わかりました。……ところで、柴崎さんが使ってる『オブジェクト』という言葉の意味がブレてる気がします。『オブジェクト型のリテラル』といいながら、その分類の中で配列リテラルとかと同列に並べて『オブジェクトリテラル』があるっていう。それ『ハッシュリテラル』のまちがいじゃないんですか？」

「ああ、混乱させちゃったね。JavaScript では『オブジェクト』っていうと、暗黙の内に**狭義のオブジェクト**と**広義のオブジェクト**の 2 つがあるんだよ。一般的なクラスベースのオブジェクト指向言語では、クラスインスタンスのことを『オブジェクト』っていうよね。

でも JavaScript でいう『オブジェクト』とは通常、キーとそれに対応する値を持ったプロパティの集まりであり、一般的には『連想配列』と呼ばれる、Ruby のハッシュや Java の HashMap に相当するものなのね。これが狭義のオブジェクトで、多くの場合 JSON 形式で表現できる。JSON は『JavaScript Object Notation（JavaScript オブジェクト表記法）』の略だからね」

「え、JSON ってそういう意味だったんですね。知りませんでした」

「そして狭義のオブジェクトに対する広義のオブジェクトとは、プリミティブ値以外のすべてのものを包括して指すの」

「待ってください。さっきはプリミティブ型の定義を『オブジェクト型ではないもの』っていってましたよね。今度は広義のオブジェクトの定義が『プリミティブ値ではないもの』って、それ循環定義[66] じゃ

[66] ある概念を定義するためにその概念自体を用いること。定義条件のみでは定義した概念の本質的な理解ができないため、定義は成立しない。

著者紹介

大岡由佳（おおおか・ゆか）

技術同人作家。React 専門のフリーランスエンジニアとして複数の現場を渡り歩いた経験を元に執筆した「りあクト！」シリーズが評判を呼びヒット作となる。なお現在は常駐も顧問も請け負っていないため、事実上おそらく国内（世界でも？）に唯一生息する、専業の技術同人作家。

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

Twitter アカウントは @oukayuka。ブログは klemiwary.com。

黒木めぐみ（くろき・めぐみ）

漫画家、イラストレーター。

「りあクト！」シリーズの表紙イラストを一貫して担当。

メールマガジン登録のご案内

くるみ割り書房 BOOTH ページ

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

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

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

りあクト！ TypeScript で始めるつらくない React 開発 第 4 版【② React 基礎編】

現場のエンジニアから多大な支持を受ける『りあクト！ TypeScript で始めるつらくない React 開発』の最新 4 版、全 3 巻構成の第 2 巻「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！

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

《第 2 巻 目次》

りあクト! TypeScript で始めるつらくない React 開発 第 4 版【③ 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 4 版、全 3 巻構成の第 3 巻「React 応用編」。

React のルーティングや状態管理、副作用処理を歴史を踏まえつつ包括的に解説しています。Redux や React 18 の新機能 Concurrent Rendering についても詳しく説明。フロントエンドが初めての方はもちろん、初中級者や前の版をお持ちの方にもオススメです。

BOOTH にて絶賛販売中！

（2022 年 9 月 10 日発行 ／ B5 版・229 p ／ 電子版 1,200 円、紙＋電子セット 1,380 円）

《第 3 巻 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 ／ B5 版・92p ／ 電子版 1,000 円、紙＋電子セット 1,100 円）

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

《目次》

りあクト! TypeScript で始めるつらくない React 開発 第 4 版
【①言語・環境編】

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

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