

初心者向け！

第4.1版

TypeScript で始める つらくない React開発

大岡由佳

① 言語・環境編

最新ツールVite、ES2023に対応

仕事で使えるReact本は
これ！ モダンFEをやさしく解説

シリーズ累計

3.4万部
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
技術書部門
1位

りあクト！ TypeScriptで始めるつらくない React開発 第4.1版 【① 言語・環境編】

まえがき

本書は 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」「②React 基礎編」「③React 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 2024 年 1 月時点ではシリーズ累計 3 万 3 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、各社で自主的に読書会が開かれたり、公式に新人研修用の教材として採用されていました。

本書の対象読者

世の技術書の多くは「わかりやすい入門書」であることを目指して書かれています。React の技術書であれば、それは「読者が React を使って何らかのプロダクトを最短で作れるようになる本」ということでしょう。しかし本書が志向するのはそこではありません。何よりも「**仕事で使える React 本**」にすることを目指して作りました。それはつまり「読者を最短で React エンジニアとして現場で働けるようにする本」ということです。

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。そのような方が React によるアプリケーション開発の現場に参加したとき、できるだけ早く一人前の戦力になってもらうためのブートキャンプ的な内容を提供しています。そしてそれにあたり、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 つの力だと筆者は考えます。

- 既存のコードがどのようにし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。よって本書は何より「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」といった方は本書に向きません。

またプログラミングそのものが初心者の方には、本書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、基本的な説明は省略しています。JavaScript自体がまったくの初心者の方は、他の入門書の『JavaScript Primer』¹などを先に読まれる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ることができるようになっています。

なおReactの中級者以上の方にとっても、本書は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詳細な歴史については、よほど初期から力を入れて追いかけていると知らない情報でしょう。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あると思われるかもしれません、会話文は各章の導入の部分のみというパターンが多く、全編に渡って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。

最初は筆者も一人称で普通に技術解説を書き始めたのですが、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な書き方をするのか、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。筆者はそれには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ることに。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書のスタイルです。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。本書では、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

¹azu・Suguru Inatomi（2020）『JavaScript Primer 迷わないと学ぶ入門書』KADOKAWA
<https://www.kadokawa.co.jp/product/302003000984/>

す。他の技術書と比べると冗長かもしれません、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、これまでの読者の少なくない感想です。実際、最初の第 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 の出番はありません。意図的にこのような構成にしているのですが、それについてこれまで説明不足のようだったので、最初にここで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 ライブライア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。以前の公式ドキュメントでも、JavaScript と同時に習得は難しいので先にそちらを学んできたださいと断っていたほどです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、Web フロントエンド開発者の多くが 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 での型定義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。企業による React アプリケーション開発は、今ではその大半が 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。

以上のような事情から、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。本書の読者には、React のコードを書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。

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

愚者は経験に学び、賢者は歴史に学ぶ

本書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、てつ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かもしれません。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ってしまいます。

思想や歴史を知ることは、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、それを知らないままチームが採用しているからという理由だけで使えば、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 力の不足によるところもありましたが、それ以上にそれらはいわゆる **React らしいコード** からほど遠かったです。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感じられました。

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ダルマ式に増えていき、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、本書執筆にあたっての筆者の願いです。

また歴史およびその背景になっている思想を知ることは、「技術選定の審美眼 (by 和田卓人氏)³」を磨くことにも結びつきます。技術的負債を極力生まないという視点から、React による開発では技術選定を上手に行うことの重要さはいくら強調してもしすぎることはありません。

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

混迷するトレンドを読み解く

ソフトウェア開発の中でも、Web フロントエンドは変化のスピードがもっとも速い分野のひとつです。本書の初版が出版された 2018 年当時は Angular と React、Vue.js が三大フレームワークとして覇を競っていました。今では信じられないかもしれません日本では Vue.js の人気が高く、当時は出版される技術書の数も React の 2~3 倍ほどと圧倒的でした。しかしそこから 3 年もしないうちに逆転し、Web フロントエンドは React 一強ともいえる状況になりました。

業界外の人なら、この状況は React さえ採用していれば失敗はないということだから喜ぶところではないかと考えるでしょうが、ことはそんなに単純ではありません。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いため、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。以前には各カテゴリーに定番的存在のライブラリがありましたが、React 一強となったことで開発リソースが集中し、またさまざまな技術レベルの人が流入したせいもあってか、細かなニーズに合わせた有力なライブラリが乱立して百家争鳴といえる状態に突入してしまいました。

また Next.js を始めとした React ベースのフレームワークが台頭し、UI ライブリヤーである React を素のままアプリケーション開発に使う機会が減ってきています。ユーザーが求める Web アプリケーションが高度になってきたことで、画像の最適化や OGP・サイトマップ生成などの各種 SEO のための機能を備え、SSR やエッジコンピューティングに最初から対応したこれらのフレームワークを企業が採用する例が増えたためです。React の新しい公式ドキュメントでは、プロジェクト作成の選択肢として挙げられているのは Next.js や Remix といったフレームワークです⁴。

このような状況でプロジェクトに最適な技術を選ぶのは、経験豊富なシニアエンジニアでもかなり難しい。トレンドを見極め、適切な技術を選定することに失敗してしまうと、後に大きな技術的負債となってしまいかねません。npm trends⁵のようなサイトを参考にするにしても、一度デファクトスタンダードになったライブラリはダウンロード数がなかなか下がらないため、現場感を持たない人にそれを読み解くのは困難です。

本書では定番にこだわらず、今現在、現場で働くエンジニアに好まれている技術を選定するようにしました。その際には他の競合となるプロダクトも公平に紹介し、その上でそれを採用した理由を明記しています。

本書でそれぞれの技術の背景となる歴史と思想を知り、経験豊富な現場のエンジニアがどのように

⁴ 「Start a New React Project – React」
<https://react.dev/learn/start-a-new-react-project>

⁵<https://npmtrends.com/>

ライブラリやツールを選定しているかを学ぶことで、読者自身にもトレンドを読み解く力を身につけたいと考えています。

サンプルコードについて

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

<https://github.com/klemiitary/Riakuto-StartingReact-ja4.1>

挙動をその場で確認していただくため StackBlitz⁶に置いているサンプルもありますが、本文中に随時 URL を掲載していますのでそちらをブラウザでご覧ください。学習を効率的に行うためにも、本文を読み進めながらこれらのコードを実際に実行することを強くおすすめします。またコードをご自身で変更して挙動のちがいを確かめてみるのも理解の助けになるはずです。

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

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

⁶<https://stackblitz.com/>

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7年ほど。本格的なフロントエンド開発チームを作るための中核的人材として、今の会社に転職してきた。チームメンバーを集めるため人材採用にも関わり自ら面接も行っていたが、彼女の要求水準の高さもあってなかなか採用に至らない。そこで「自分がReactを教えるから他チームのエンジニアを回してほしい」と上層部に要望を出し、社内公募が行われた。

秋谷香苗（あきや かなえ）

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

前版との差分および正誤表

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

・各版における内容の変更

<https://github.com/klemiitary/Riakuto-StartingReact-ja4.1/blob/main/CHANGELOG.md>

・『りあクト！TypeScriptで始めるつらくないReact開発 第4.1版』正誤表

<https://github.com/klemiitary/Riakuto-StartingReact-ja4.1/blob/main/errata.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
- ReactDOM (`react-dom`) 18.2.0
- Vite (`vite`) 5.0.10
- TypeScript (`typescript`) 5.3.3

目次

まえがき	3
本書について	9
登場人物の紹介	9
前版との差分および正誤表	9
本書内で使用している主なソフトウェアのバージョン	10
プロローグ	16
第1章 こんにちは React	18
1-1. 基本環境の構築	18
1-1-1. Node.js がなぜフロントエンド開発に必要なのか	18
1-1-2. Node.js をインストールする	21
1-1-3. 超絶推奨エディタ Visual Studio Code	26
1-2. Vite でプロジェクトを作成する	29
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	38
1-3-1. pnpm	38
1-3-2. npm スクリプト	44
第2章 ライトでディープな JavaScript の世界	47
2-1. あらためて JavaScript ってどんな言語？	47
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	47
2-1-2. 年々進化していく JavaScript	50
2-2. 変数の宣言	52
2-3. JavaScript のデータ型	55
2-3-1. JavaScript におけるプリミティブ型	55
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	58
2-3-3. オブジェクト型とそのリテラル	61
2-4. 関数の定義	64
2-4-1. 関数宣言と関数式	64
2-4-2. アロー関数式と無名関数	68
2-4-3. さまざまな引数の表現	71
2-5. クラスを表現する	73
2-5-1. クラスのようでクラスでない、JavaScript のクラス構文	73
2-5-2. プロトタイプベースのオブジェクト指向とは	75
2-6. 配列やオブジェクトの便利な構文	80

2-6-1. 分割代入とスプレッド構文	80
2-6-2. オブジェクトのマージとコピー	85
2-7. 式と演算子で短く書く	88
2-7-1. ショートサーキット評価	88
2-7-2. Nullish Coalescing と Optional Chaining	89
2-8. JavaScript の this を理解する	92
2-8-1. JavaScript の this とは何なのか	92
2-8-2. 関数における this の中身 4 つのパターン	96
2-8-3. this の挙動の問題点と対処法	101
2-9. モジュールを読み込む	104
2-9-1. JavaScript モジュール三國志	104
2-9-2. ES Modules でインポート／エクスポート	109
第3章 関数型プログラミングでいこう	113
3-1. 関数型プログラミングは何がうれしい？	113
3-2. コレクションの反復処理	119
3-2-1. 配列の反復処理	119
3-2-2. オブジェクトの反復処理	125
3-3. JavaScript で本格関数型プログラミング	127
3-3-1. あらためて関数型プログラミングとは何か	127
3-3-2. 高階関数	128
3-3-3. カリー化と関数の部分適用	130
3-3-4. 閉じ込められたクロージャの秘密	132
3-4. 非同期処理と例外処理	136
3-4-1. Promise で非同期処理を扱う	136
3-4-2. async と await	139
3-4-3. 複数の Promise をまとめて扱う	142
3-4-4. JavaScript の例外処理	145
第4章 TypeScript で型をご安全に	149
4-1. ナウなヤングに人気の TypeScript	149
4-2. TypeScript の基本的な型	154
4-2-1. 型アノテーションと型推論	154
4-2-2. JavaScript と共通のデータ型	156
4-2-3. Enum 型とリテラル型	159
4-2-4. タプル型	162
4-2-5. any、unknown、never	163
4-3. 関数とクラスの型	166
4-3-1. 関数の型定義	166
4-3-2. TypeScript でのクラスの扱い	170
4-3-3. クラスの 2 つの顔	174

4-4. 型の名前と型合成	177
4-4-1. 型エイリアス VS インターフェース	177
4-4-2. ユニオン型とインターフェクション型	180
4-4-3. 型の Null 安全性を保証する	183
4-5. さらに高度な型表現.....	186
4-5-1. 型表現に使われる演算子	186
4-5-2. 条件付き型とテンプレートリテラル型	189
4-5-3. 組み込みユーティリティ型	193
4-5-4. 関数のオーバーロード	197
4-6. 型アサーションと型ガード	200
4-6-1. as による型アサーション	200
4-6-2. 型ガードでスマートに型安全を保証する	202
4-7. モジュールと型定義.....	205
4-7-1. TypeScript のインポート／エクスポート	205
4-7-2. JavaScript モジュールを TypeScript から読み込む	210
4-7-3. モジュールの型はどのように解決されるか	213
4-8. TypeScript の環境設定	219
4-8-1. コンパイラオプション strict	219
4-8-2. tsconfig.json の設定項目	222
4-8-3. 複数の tsconfig.json を連携させる	227

他巻目次

「②React 基礎編」 目次

第5章 JSXでUIを表現する

- 5-1. なぜReactはJSXを使うのか
- 5-2. JSX構文の書き方

第6章 ビルドツールを理解して有効に使う

- 6-1. コンパイラとモジュールバンドラ
- 6-2. そしてViteが最後に残った
- 6-3. Viteを本格的に使いこなす
- 6-4. ポストNode.jsとしてのDenoとBun

第7章 リンターとフォーマッタでコード美人に

- 7-1. リンターでコードの書き方を矯正する
- 7-2. フォーマッタでコードを一律に整形する
- 7-3. スタイルシートもリンクティングする
- 7-4. さらに進んだ設定

第8章 Reactをめぐるフロントエンドの歴史

- 8-1. Reactの登場に至る物語
- 8-2. Reactを読み解く4つのキーワード
- 8-3. 他のフレームワークとの比較
- 8-4. Reactベースのフレームワーク

第9章 コンポーネントの基本を学ぶ

- 9-1. コンポーネントのメンタルモデル
- 9-2. コンポーネントにPropsを受け渡す
- 9-3. コンポーネントにStateを持たせる
- 9-4. コンポーネントと副作用
- 9-5. クラスでコンポーネントを表現する
- 9-6. ロジックを分離、再利用する

「③React 応用編」目次

第 10 章 コンポーネントの高度なハンドリング

- 10-1. フォームを扱う
- 10-2. コンポーネントのレンダリングを最適化する

第 11 章 React アプリケーションの開発テクニック

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

第 12 章 React のページをルーティングする

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

第 13 章 Redux でグローバルな状態を扱う

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

第 14 章 ポスト Redux 時代の状態管理

- 14-1. 若者の Redux 離れ？
- 14-2. Context API の登場
- 14-3. React で非同期処理とどう戦うか
- 14-4. 次世代の状態管理ライブラリ

第 15 章 並列レンダリングを使いこなす

- 15-1. React 18 はなぜ特別なバージョンなのか
- 15-2. 並列レンダリングの機能を有効化する
- 15-3. 並列レンダリングの具体的なメリット
- 15-4. 並列レンダリングで UI の質を高める

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、聞いてますよ。秋谷さんね。こちらこそよろしくお願ひします。私はこのフロントエンド開発チームの……といつ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く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。ありがとうございます。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、よろしくお願ひします！」

「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 Ruby on 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。Railsと考え方が全然ちがっていて、テンプレートとロジックを混ぜて書く感じなのも違和感があって、よくわからなかつたんですよね。すみません……」

「いや、謝ることはないけれども。そうだね、秋谷さんのようにサーバサイドでよくあるMVC⁷フレームワークになじんでる人は、逆にそれが学ぶ障害になってしまうかもしれない」

「……そうなんですか？」

「Reactのアプリケーション設計思想はそもそもMVCと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」

⁷Model-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。

「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について、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、不安だなあ」

「ふふ、だーいじょうぶ。Rails は使いこなしてたんだしよう？ なら原理さえ理解できれば、そこで 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やいけないのが Node.js⁸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。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、知っています」

「Node.js とは簡単にいうと、JavaScript を Ruby や Python と同じように秋谷さんの PC のターミナル上で実行するための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 V8⁹を組み込んで、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も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 Web 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>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>React Test</title>
```

⁸<https://nodejs.org/ja/>

⁹Google 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イルを行う）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/>

```

</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む必要があるて、しかも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。それらをただこうやって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 gem¹⁰や Bundler¹¹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 npm¹²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ほど。昔は Bower¹³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。^{むか}だから今の

¹⁰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/>

¹¹gem 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/>

¹²Node.js のために作られたパッケージ管理システム。パッケージをローカルにインストールしたり管理したりする機能に加え、リポジトリ機能も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。2024 年の時点において提供するパッケージ数は 200 万以上と世界最大で、RubyGems や Maven Central (Java) など他言語のものを圧倒している。<https://www.npmjs.com/>

¹³<https://bower.io/>

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いる

「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AltJS¹⁴のコードを古いバージョンのJavaScriptにコンパイルして、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¹⁵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DX¹⁶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！」

「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」

¹⁴ 「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。

¹⁵ 「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。

¹⁶ 昨今は「Digital Transformation」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 Homebrew¹⁷、Windows なら winget¹⁸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」

「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 をインストールするの。この分野で現在メジャーなのは古くからある nvm¹⁹、新興では fnm²⁰や Volta²¹あたりかな」

「Ruby でも同じようなものに RVM²²と rbenv²³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asdf²⁴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隨も早い。似たようなツールに anyenv²⁵というのがあって私も前はこれを使ってたんだけど、開発が止まっちゃったので後発でより洗練されてる asdf に乗り換えたの」

「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？」

「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を読んでみよう」

¹⁷Max 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。homebrew 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/>

¹⁸Microsoft 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。2021 年 5 月にバージョン 1.0 がリリースされた。

<https://learn.microsoft.com/ja-jp/windows/package-manager/winget>

¹⁹<https://github.com/nvm-sh/nvm>

²⁰<https://github.com/Schniz/fnm>

²¹<https://volta.sh/>

²²<https://rvm.io/>

²³<https://github.com/rbenv/rbenv>

²⁴<https://asdf-vm.com/>

²⁵<https://anyenv.github.io/>

「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のかもね。じゃあ実際にインストールしていこう。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 (10.15) 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まりおすすめできない。その理由は 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？」

「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 WSL²⁶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 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」

「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けど。ハードの選定からできるなら、初心者は周りの先輩が多く使ってる Mac を使ってほしい」

²⁶ 「Windows Subsystem 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、VirtualBox 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://learn.microsoft.com/ja-jp/windows/wsl/>

「まあ、そうなるでしょうね」

「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²⁷、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！」

「話を本題に戻すね。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。`--all` オプションを指定すると追加済みのプラグインすべてが対象になる
- `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アンインストール
- `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

「ここでいうプラグインとは asdf がサポートする言語やツールを追加するためのものね。たとえば Node.js を asdf でバージョン管理するためには『nodejs』という名前のプラグインを追加する必要がある。気をつけないといけないのは、そのプラグインのインストール可能なバージョンのリストはプラグイン自体をアップデートしないと更新されないこと。

「asdf のインストール直後ならいいけど、その後は Node.js の最新バージョンをインストールしたい場合はまず `asdf plugin update nodejs` を実行しないといけない」

「それは気をつけてないと忘れそうですね。ところで任意の名前のプラグインを探すコマンドはないんでしょうか？」

²⁷<https://github.com/klemiitary/Riakuto-StartingReact-ja4.1/blob/main/extrabuild-win-env.md>

「それがないのよ。だから `asdf plugin list all | grep node` のように grep で抽出するしかないね。で、今から実際に 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つて次の内容を書き込んでくれる？」

リスト2: `.default-npm-packages`

```
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` というファイルに相当する。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。次のようにコマンドを実行して」

```
$ asdf plugin add nodejs
$ asdf install nodejs latest
$ asdf list nodejs
  21.6.0
$ asdf global nodejs 21.6.0
```

「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も Node のコマンドが使えないで忘れないでね。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つたりするので」「そういうえば 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。忘れないよう気をつけます！」

「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、インストールの結果がどうなつたか `node --version` を実行して確認してみて」

「『v21.6.0』って表示されました。インストール成功です！」

「うん、じゃ次に `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、node²⁸と単体で実行すると REPL²⁸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」

「へー、これよさそうですね。活用させてもらいます」

²⁸ 「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。

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 Code²⁹、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、将来的に加わる予定のメンバーも含めて 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『The State of JavaScript』という調査があるんだけど、少し古いながら 2020 年の結果では VS Code のユーザー使用率は 86% と他をぶっちぎっての 1 位³⁰」

²⁹<https://code.visualstudio.com/>

³⁰https://2020.stateofjs.com/ja-JP/other-tools/#text_editors

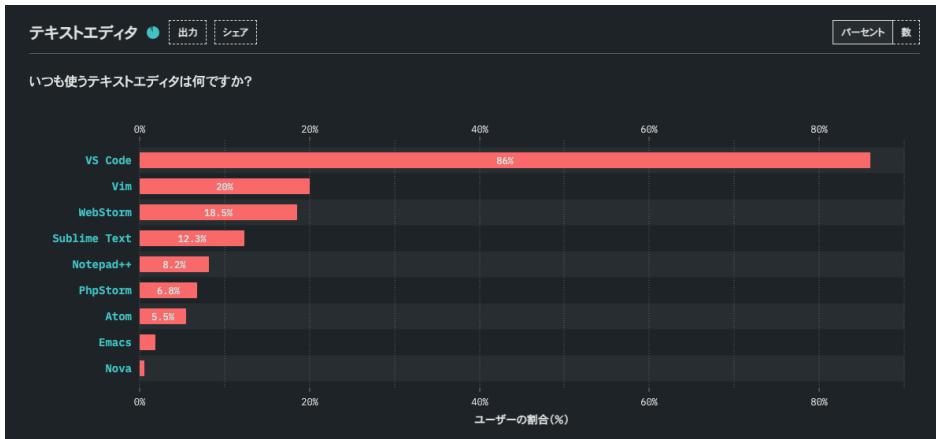


図 1: JS開発者間でのエディタの使用率（State of JS2020 より）

「さらに React 開発元の Meta³¹でも VS Code がデフォルトの開発環境になってるらしいよ³²」

「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いること。Remote Development³³では VS Code をインストールしているリモートのマシンに接続して開発することもできるし、Visual Studio Live Share³⁴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が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」

³¹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。

³² 「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/>

³³<https://code.visualstudio.com/docs/remote/remote-overview>

³⁴<https://visualstudio.microsoft.com/ja/services/live-share/>

「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VSCodeVim³⁵とか、もっと軽量でかつVS Codeのマルチカーソル機能³⁶とも共存可能なamVim³⁷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。`⌘P`(※Windows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
- Path Intellisense (`christian-kohler.path-intellisense`)
..... インポート文のパスを自動補完してくれる

³⁵<https://marketplace.visualstudio.com/items?itemName=vscodevim.vim>

³⁶文書内の複数の場所にカーソルを配置し、同時に編集できる機能。カーソル位置の単語を1件ずつ探索して選択するショートカット`⌘D`(※Windowsでは`Ctrl+D`)が特にVimとバッティングしやすい。
https://code.visualstudio.com/docs/editor/codebasics#_multiple-selections-multicursor

³⁷<https://marketplace.visualstudio.com/items?itemName=auworks.amvim>

- Auto Rename Tag ([formulaahendry.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
- Auto Close Tag ([formulaahendry.auto-close-tag](#))

..... HTML/XML タグを書くと、自動的に閉じタグを追加してくれる
- 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
- Japanese Language Pack for Visual Studio Code ([MS-CEINTL.vscode-language-pack-ja](#))

..... VS Code の UI を日本語にローカライズする。以前は標準機能だったが拡張に切り出された
- 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
- 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 minify³⁸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

³⁸ 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。

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 のプロジェクトを作成・運用するためのツールを使う。以前は公式から Create React App³⁹というツールが提供されていて、それがデファクトスタンダードだったんだけど 2022 年 4 月で開発が止まってしまった。今の主流は Vite⁴⁰だね。以降はこれを使うこと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 の作者ってフランスかぶれなんですね。『ミーはおフランス帰りざんす！』みたいな」

「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」

```
$ npm create vite@latest 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。後ろの --template=react-ts っていうのは？」

「適用するプロジェクトテンプレートの種類を指定してるの。Vite は React 専用のツールってわけじゃなくて、Vue.js や他のフレームワークのプロジェクトも作れる。公式で用意されてるテンプレートは 2024 年 1 月現在で 18 種類あって⁴¹、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」

「わかりました。あとテンプレートオプションの指定の前にもうひとつ -- があるのは何ですか？」

「これは npm コマンド自身のオプションではなく、内部で実行している vite コマンドに渡してるオプションだから、こうやって区別する仕様になってるの。ちなみにこれが必要になったのは npm のバージョン 7 以降」

「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？」

```
Scaffolding project in ***/hello-world...
```

```
Done. Now run:
```

³⁹<https://create-react-app.dev/>

⁴⁰<https://ja.vitejs.dev/>

⁴¹<https://github.com/vitejs/vite/tree/main/packages/create-vite>

```
cd hello-world
npm install
npm run dev
```

「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、`npm install` を実行しよう」

「はい、実行っと……。このコマンドは何をやっているんですか？」

「これは `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 `package-lock.json` というファイルに出力するもののなの」

「たしかに、プロジェクトのディレクトリに `node_modules/` と `package-lock.json` が追加されています。`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、それ相互に特定のバージョンに依存しているので、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ロックファイルが `package-lock.json` ね。」

「だから Git のリポジトリに `node_modules/` は入れないけど、ロックファイルは必ず入れておくようになる」

「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」

「うん。では最後に `npm run dev` を実行」

```
VITE v5.0.10 ready in 305 ms
```

```
►Local:    http://localhost:5173/
►Network:  use --host to expose
►press    h + enter to show help
```

「これでローカル環境に開発サーバが起動した。表示されている 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！」

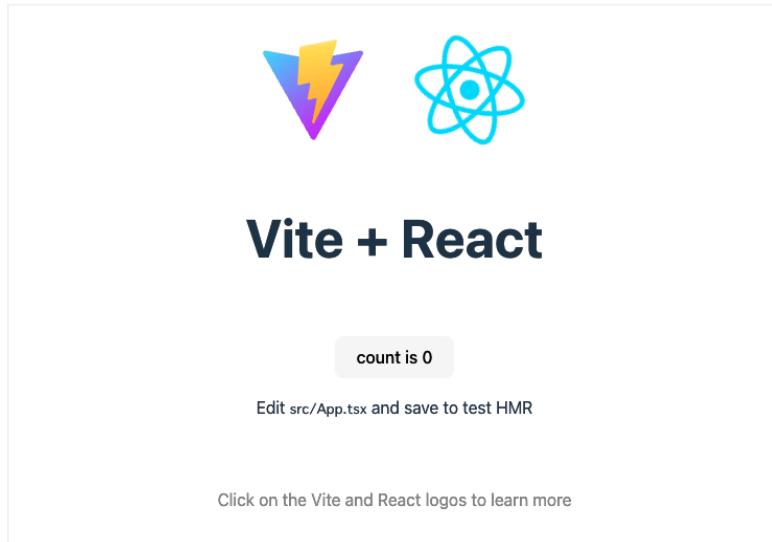


図 2: Vite でプロジェクト作成したデフォルトの画面

「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 `npm run 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『ページのソースを表示』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')).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警告してくれる React の『Strict モード⁴²』という機能を有効にするためのラッパーだね。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。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 パーツを表現するかたまり』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⁴³になってる」

⁴² 「`<StrictMode>` – React」 <https://ja.react.dev/reference/react/StrictMode>

⁴³ 「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。

「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 viteLogo from '/vite.svg'
import './App.css'

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

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

```
export default App
```

「うーむ、何が何やら……」

「カウントボタンを実現するためのロジック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 ということは、コンポーネントを関数で定義してるんでしょうか。あと、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 있습니다. それにしても return 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。『JSX⁴⁴』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 となっているでしょ。これは TypeScript ベースの JSX ファイルってことで、コンパイルを経て最終的に JavaScript コードに変換されるのね。」

「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 のコンパイル後の姿ですか……」

「そう。これで流れはわかっただろから、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 viteLogo from '/vite.svg'
import './App.css'

function App() {
  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
      </div>
    </>
  )
}

export default App
```

⁴⁴<https://facebook.github.io/jsx/>

```
<a href="https://react.dev" target="_blank">
  <img src={reactLogo} className="logo react" alt="React logo" />
</a>
</div>
<h1>Hello, World!</h1>
</>
)
}

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！」

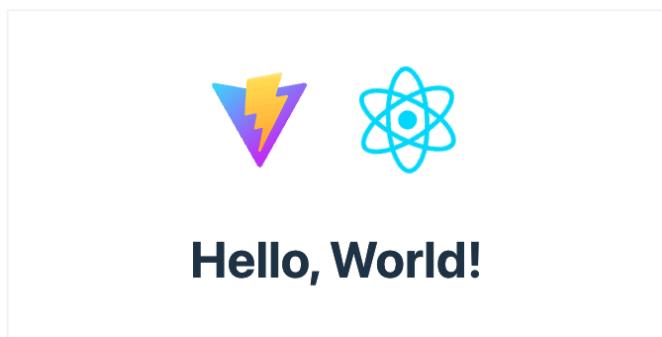


図 3: Hello, World!

「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
<code>src/</code>	アプリケーションのソースコードを置く
<code>node_modules/</code>	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
<code>public/</code>	公開用のアセットファイルを置く
<code>package.json</code>	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
<code>package-lock.json</code>	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
<code>tsconfig.json</code>	TypeScript をコンパイルするための設定ファイル
<code>tsconfig.node.json*</code>	Vite の設定を TypeScript で書くための <code>tsconfig.json</code>
<code>vite.config.ts*</code>	Vite の設定ファイル
<code>.gitignore</code>	Git リポジトリに含めないもののリスト

「プロジェクト作成ツールに Vite、パッケージマネージャに npm、開発言語に TypeScript を使っ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、色々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んだけど」

「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 を使ってるので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のよ。実際、Create React App で不満だった点が Vite ではかなり解消されてて、だからこそ開発者はこぞって乗り換えたんだよね」

「そうなんですか……。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. pnpm

「npm は 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。ただ『npm』といつてもシーンによって意味が複数あるの。ひとつは Node.js のパッケージマネージャとしてのシステム全体を指す言葉として。次にそれによってインストールできるパッケージの公開リポジトリを指す言葉としての意味もある。npmjs.com に登録されているパッケージ群のことね。」

そしてそれらのパッケージをローカルにインストールしたりする管理機能を持つ CLI ツールを指す言葉としても用いられる。パッケージ名もそのまま npm で、これは現在では Node.js をインストールするといっしょに入ってる。だから `npm install` とか `npm run` のようなコマンドが使えたわけ。`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 <foo>` や `npm run <foo>` とかが載ってますね」

「CLI ツールとしてのパッケージマネージャはこの標準の npm の他に、後発のものがいくつかある。後発だけあって npm よりパフォーマンスや使い勝手が改良されてるので、そっちを好む開発者も多いの。有名なところで Yarn⁴⁵ や pnpm⁴⁶ がある」

「ふむふむ、それらはどういう経緯で作られたんでしょうか？」

「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。」

たとえば、依存パッケージのバージョン固定ができるロックファイルのしくみを最初に備えたのはこの Yarn。本家の npm が追随してバージョン 5.0 から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ワークスペースという機能も 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」

「へーえ」

「そして 2017 年にさらに後発の pnpm がリリースされた。『performant npm』という意味で名付けられたように、何よりパフォーマンスの改善に注力されたプロダクトだった。インストールなどの高速化もそうだけど、もっとも特徴的なのはディスク容量を節約するためのしくみがあること」

「そもそも Node.js のパッケージってそんなにディスク容量を食うものなんですか？」

「さっきの Vite によるテンプレートプロジェクトでも、パッケージの数は 210 あってその容量は 64MB ほどもある。これが大きなプロジェクトだと GB 単位になるね」

⁴⁵<https://yarnpkg.com/>

⁴⁶<https://pnpm.io/>

「210！？ そんなにありましたっけ？」

「`package.json` に記述されているパッケージは 12 個なんだけど、それらが依存しているパッケージを合計すると 200 以上になるのよ。そしてそれらのファイルは `node_modules/` ディレクトリに格納されるんだけど、npm や Yarn では各パッケージが依存する他のパッケージは入れ子になった `node_modules/` に格納される。すると困ったことに、A と B の 2 つのパッケージが C というパッケージに依存していると、C のパッケージファイルがそれぞれ `node_modules/A/node_modules/` と `node_modules/B/node_modules/` の下に置かれることになる」

「うーむ、大いなる無駄ですね」

「pnpm では `node_modules/.pnpm` ディレクトリにパッケージの実体を置き、`node_modules` 直下にはそこからシンボリックリンクを貼るようになってる。だから依存が重複するパッケージがあっても、パッケージはひとつしか置かれない。pnpm はこの方法でディスク容量を節約してるの。また重複インストールがないということは、それだけ全体のインストールの時間も少なくなる。小さいプロジェクトではあまり差はないけど、大きいプロジェクトになるとけっこう変わってくる」

「なるほど」

「また最近は『モノレポ (Monorepo)』といって、ひとつのリポジトリに複数のプロジェクトを含める開発手法が流行ってきてる。たとえば Web フロントエンドとサーバ API と定期実行のサーバスクリプトを同じ TypeScript で開発したら、ライブラリやツールも共有したいよね。そういう場合に 3 つのプロジェクトをリポジトリ内にひっくるめる。そのときに先述したワークスペースという機能を使いんだけど、pnpm は設定がシンプルでパフォーマンスも高い。だからモノレポが多用される組織では pnpm が選ばれることが多いの。

CLI ツール用途での npm のダウンロード数だけの抽出はできないので、Yarn と pnpm の 2 つで見るとトレンドはこんな感じになっている」

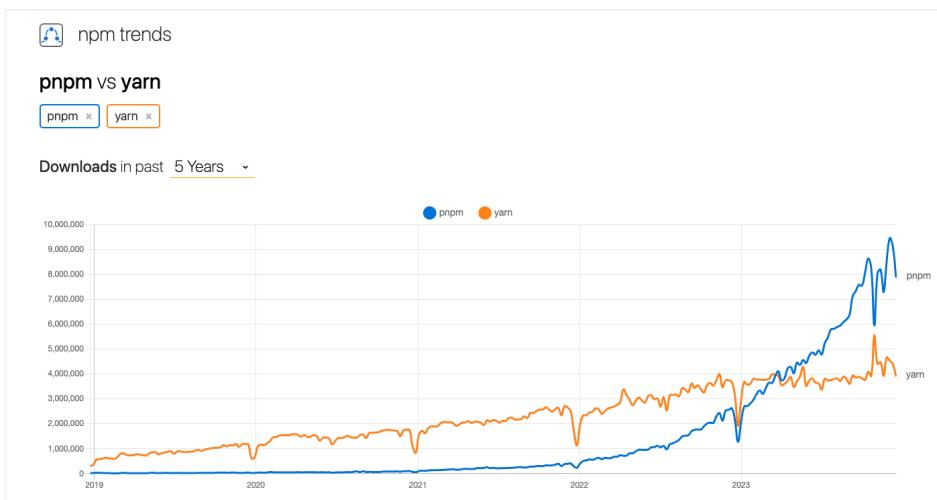


図 4: パッケージマネージャの DL 数比較（2024 年 1 月現在）

「おおー、2023年にpnpmがめっちゃ伸びてYarnを追い抜いてる！」

「最近は有名どころのオープンソースプロジェクトでもpnpmの採用例が多い。ViteやNext.jsもpnpmのワークスペース機能を使って開発されてるよ⁴⁷。」

「後発だけあって、乗り換えやすいようにコマンドもnpmやYarnのいいとこ取りになってておぼえやすい。移行のためにロックファイルのインポートもできるようになってる」

「ふーむ、最初からユーザーを奪う気満々だったんですね」

「というわけで、ウチでもパッケージマネージャにはpnpmを採用します。ここからはpnpmの使い方を学んでいこう。まずpnpmをどのようにインストールするかだけど、Node.jsのバージョンに依存させたくないのasdfを使うことにする」

```
$ asdf plugin add pnpm
$ asdf install pnpm latest
$ asdf list pnpm
  8.14.1
$ asdf global pnpm 8.14.1
```

「次に、pnpmでよく使うコマンドを次にピックアップしてみた」

- `pnpm 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。`i`に省略可。
- `pnpm add [-D|--save-dev] <PACKAGE_NAME>`
..... 指定したパッケージをインストールする。`-D`を指定すると`devDependencies`としてインストール。
- `pnpm remove <PACKAGE_NAME>`
..... 指定したパッケージをアンインストールする。`rm`に省略可。
- `pnpm update [-L|--latest] <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。`up`に省略可。`-L`を指定すると`package.json`で指定されているバージョン範囲を無視して最新にする。
- `pnpm [run] <SCRIPT_NAME>`
..... package.jsonに登録されているスクリプトを実行する。
- `pnpm dlx <PACKAGE_NAME>`
..... 依存関係としてインストールすることなく、パッケージを一時的に読み込んで、そのパッケ

⁴⁷ 「使用例 ワークスペース | pnpm」

<https://pnpm.io/ja/workspaces>

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

一覧が提供するデフォルトのコマンドバイナリを実行する。`pnp` にエイリアスが貼られている。

- `pnpm info <PACKAGE_NAME> [time]`
..... 指定したパッケージについての情報を表示する。`show` でも同じ。パッケージ名の後に `time` でインストール可能なバージョンが公開日時付きでリスト表示される。
- `pnpm list [--depth <n>]`
..... インストールされているパッケージのすべてのバージョンと依存関係を、ツリー構造で出力する。`ls` に省略可。`--depth <n>` でツリーの深さを指定できるが、`n` のデフォルト値は 0。
- `pnpm outdated`
..... プロジェクトの古くなったパッケージを確認する。

「`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。さっそく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",  
    "lint": "eslint . --ext ts,tsx --report-unused-disable-directives --max-warnings 0",  
    "preview": "vite preview"  
  },  
  "dependencies": {  
    "react": "^18.2.0",  
    "react-dom": "^18.2.0"  
  },  
  "devDependencies": {  
    "@types/react": "^18.2.37",  
    "@types/react-dom": "^18.2.15",  
    "@typescript-eslint/eslint-plugin": "^6.10.0",  
    "@typescript-eslint/parser": "^6.10.0",  
    "@vitejs/plugin-react": "^4.2.0",  
    "eslint": "^8.53.0",  
    "eslint-plugin-react-hooks": "^4.6.0",  
    "eslint-plugin-react-refresh": "^0.4.4",  
    "typescript": "^5.2.2",  
    "vite": "^5.0.0"  
  }  
}
```

「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 pnpm 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？」

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

「ふむふむ。ちなみに remove や update コマンドの際は、そのオプション指定は必要ないですか？」

「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 pnpm 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 pnpm 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 pnpm-lock.yaml 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る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りしたときは、`git restore` でいったん `package.json` や `pnpm-lock.yaml` の内容を元に戻してから原因を特定するようにしましょう」

「了解です」

「じゃあ次は、Vite を使ったプロジェクトを `pnpm` で作り直してみようか」

```
$ pnpm create vite hello-pnpm --template=react-ts
$ cd hello-pnpm/
$ pnpm i
$ pnpm dev
```

「`pnpm i` は `pnpm install` の、`pnpm dev` は `pnpm run dev` の省略形なんですよね。それにしても `pnpm` つて打ちづらくないですか？『右小指→右人差し指→右小指→右中指』って、右手つりそう」「それは私もそう思う。公式も自覚はあるようで、適当にエイリアス貼ってねって言ってる⁴⁸。私もホームディレクトリの `.aliases` に次のような設定してるよ」

リスト 8: `.aliases`

```
:
alias pn='pnpm'
```

■ パッケージマネージャのマネージャ Corepack

`package.json` には特定のパッケージマネージャを前提としたスクリプトが記述されていましたが、その特定のバージョンに依存した機能を使用していることがあったりするため、パッケージと同様にその種類やバージョンを `package.json` で管理したいというニーズが以前からありました。そこで 2022 年 2 月にリリースされた Node.js v14.19 より、Corepack という機能が標準でバンドルされるようになりました。

Corepack を使うためには `corepack enable` コマンドを実行して機能を有効にし、`package.json` に次のようなパッケージマネージャ用のフィールドを追加します。

⁴⁸Using a shorter alias - Installation | pnpm
<https://pnpm.io/installation#using-a-shorter-alias>

```
"packageManager": "pnpm@8.14.1",
```

これによりプロジェクトの管理者が、特定のパッケージマネージャの特定のバージョンを開発メンバーに一律に使わせることができます。ただし Corepack 管理下のパッケージマネージャを実行するには、パス解決のために `corepack pnpm` などと毎回記述する必要があります（※公式ではエイリアスを貼ることを推奨）。また 2024 年 1 月現在、まだ実験的扱いのためか Node.js をバージョンアップすると Corepack の設定が無効に戻されるという現象があるようです。

そのため今回は Corepack の使用を見送り、pnpm を asdf でインストールするようにしました。しかし今後 Node.js で正式な機能とされ普及が進んだ暁には採用を前向きに考えてもよさそうです。

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",
  "lint": "eslint . --ext ts,tsx --report-unused-disable-directives --max-warnings 0",
  "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は `pnpm run dev` だったよね。この `run` コマンドで実行できるのが、ここに登録されているエントリの内容なの。これに使っているのは npm の `scripts`⁴⁹ という機能。`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 `pnpm run <COMMAND_NAME>` でそれが実行される。また `run` は省略可能なので、あたかも素のコマンドのように `pnpm dev` と実行できる」

「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

⁴⁹ 「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/v10/using-npm/scripts>

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 `vite50` 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` アプリケーションサーバの起動コマンドの登録用
- `restart` アプリケーションサーバの再起動コマンドの登録用
- `stop` 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 のテンプレートではこうすることで `start` は本番用サーバの起動のために空けてるの。最近はこの形式が増えてるようね」

「ふーん、なるほど」

「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

⁵⁰ 「コマンドラインインターフェイス | Vite」

<https://ja.vitejs.dev/guide/cli.html>

「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。わからなかつたら公式ドキュメント⁵¹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⌘+ 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！」

⁵¹ [Life Cycle Operation Order - scripts | npm Docs]
<https://docs.npmjs.com/cli/v10/using-npm/scripts#life-cycle-operation-order>

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よくいるけど、React ではそれは通用しない。」

ウチのチームでは実際には 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ろ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いる？」

「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、その座を譲ったのはわずか 1 回しかない⁵²。」

⁵²RedMonk Top 20 Languages Over Time: January 2023
<https://redmonk.com/rstephens/2023/05/16/top20-jan2023/>

使われてる数だけじゃ納得できないかもしれないから、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 2022⁵³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。そもそもその話なんですか、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1995年ごろ当時

⁵³Stack 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>

Netscape の社員で、静的スコープと第一級関数をサポートする Scheme⁵⁴とプロトタイプベースのオブジェクト指向言語 Self⁵⁵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⁵⁶」

「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。」

とはいえた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』⁵⁷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き、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がうと思つてないと」

「すいません。その^{なんとう}え、全然ピンとこないんですが……」

⁵⁴Lisp 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。

⁵⁵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。

⁵⁶ 「JavaScript: how it all began」
<https://2ality.com/2011/03/javascript-how-it-all-began.html>

⁵⁷Douglas Crockford／水野貴明（訳）（2008）『JavaScript: The Good Parts ——「よいパート」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

「秋谷さんに言われて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が引き継いでい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 International⁵⁸という業界団体が定めている標準仕様で、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」

「おるとじえーえす?」

「『Alternative JavaScript』の略で、JavaScriptの代替言語のこと。JavaScriptを使いや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CoffeeScript⁵⁹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」

「ECMAScriptは近年、ほぼ毎年のように新しい仕様が公表されてる。2024年1月現在での最新版はES2023、つまり『ECMAScript 2023』で2023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、今回も順当に出たみたいね」

⁵⁸<https://www.ecma-international.org/>

⁵⁹JavaScript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5.1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 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。これはよっし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『悪いパート』や『ひどいパート』⁶⁰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うまく避けることができるの、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取らない快適で効率的な開発ができる」

「その『悪いパート』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 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、お願ひします！」

⁶⁰ 『JavaScript: The Good Parts — 「よいパート」によるベストプラクティス』 p.117

2-2. 変数の宣言

「ではまず変数の宣言から。既存のRailsのプロジェクトコードでは変数の宣言にカジュアルにvarキーワードが使われてたりするけど、これは金輪際もう使ってはいけません。禁止します」

「え、私が前にいたプロジェクトではvarがけっこう使われてたんですけど、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！」

「関数型プログラミングの文脈では、変数などを可変であることをミュータブル（mutable）、不变であることをイミュータブル（immutable）と表現する。`let` はミュータブルな、`const` はイミュータブルな変数を宣言するキーワードってことね」

「でも『イミュータブルな変数』というのは、なんか矛盾を含んだ表現ですよね」

「プログラミングの世界においてはコンパイル言語が先にあって、『定数』というのはコンパイル時に値が決定して最後まで変化しない、任意のデータに固有の名前を与えたもののこと指す。それに対してイミュータブルな変数は宣言時に確保したメモリの領域が書き換え不可というだけ。だから JavaScript なら `const` で宣言したオブジェクトのプロパティは書き換え可能なわけね」

「ふーむ、なるほど」

「ちなみに `var` は『variable（変化しやすい）』、`const` は『constant（恒常的な）』が由来。『～させてやる』という意味の動詞 `let` が最初に変数の宣言に使われるようになったのは Lisp なので、その系列の Scheme を参考にして作られた JavaScript はその流れでこのキーワードが採用されたのかもね。ちなみに Rust や Swift など他の言語でも `let` を使うけど、その場合の変数はイミュータブルになる」

「……まぎらわしいですね」

「まあそのへんは JavaScript が古い歴史を引きずってる所以だよね。

話を戻すと、`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9: 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でかな。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)⁶¹』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10: 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
```

⁶¹ 「Hoisting(巻き上げ、ホイスティング)-用語集 | MDN」
<https://developer.mozilla.org/ja/docs/Glossary/Hoisting>

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 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すよ。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ですね。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、型がない言語じゃないんでしたっけ」

「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で

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.....」

「わかつてもらえたようね。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⁶²。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、インスタンスマ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。扱うことができる最大値は $2^{53}-1$ （9,007,199,254,740,991≈9千兆）。

⁶² 「JavaScriptのデータ型とデータ構造 - JavaScript | MDN」
https://developer.mozilla.org/ja/docs/Web/JavaScript/Data_structures

- **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によって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 でのパースができる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、数値ではないことを示す値のこと。『Not a Number』の略ね。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『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」

「ちなみにある値が `NaN` かどうかの判定は `Number.isNaN()` を使う。JavaScriptには他にグローバルな `isNaN()` 関数があるんだけど、これは使っちゃだめ。数値以外の値を渡すと、暗黙の型変換が起こって判定結果がおかしくなるので」

```
> Number.isNaN(Infinity * 0)  
true  
> Number.isNaN(1234)  
false  
> Number.isNaN('foo')  
false  
> isNaN('foo')  
true
```

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『インスタンスマソッドを持たない』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いますよ。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）⁶³』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」

⁶³ 「文法とデータ型 - JavaScript | MDN」

https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Grammar_and_types

- 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
..... シングルクオート ' または ダブルクオート " で囲まれた文字列リテラル。バッククオート \ 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⁶⁴ 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る不可だけだ」

「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、それらの値を抱含する『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`⁶⁵、Number 型なら `Number`⁶⁶ 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⁶⁷として備

⁶⁴ 「テンプレートリテラル (テンプレート文字列) - JavaScript | MDN」

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Template_literals

⁶⁵ 「String - JavaScript | MDN」

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/String

⁶⁶ 「Number - JavaScript | MDN」

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Number

⁶⁷ 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

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するの。実際の挙動で示すところなるね】

```
> 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、内部でこのように変換されてる」

```
'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マ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ろしくないですか？」

「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、そういうものですか.....】

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

著者紹介

大岡由佳（おおか・ゆか）

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

趣味はホームシアターでの映画鑑賞と、5年以上続けながら一向に上達しないクラシックバレエ。最近では某小説に影響されて水墨画も始めた。

Twitterアカウントは@oukayuka。ブログは klemiwary.com（くるみ割り書房公式サイト）。

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。

お知らせフォローのご案内

The screenshot shows a user profile on the BOOTH platform. The profile picture is a cartoon illustration of a young woman with brown hair. The profile name is 'くるみ割り書房 ft. React'. Below the name, it says '大岡由佳'. There is a red 'フォロー' (Follow) button. The bio text reads: '『仕事で使えるReact本』といえばコレ！ 超・実践的なReactが学べると評判の『りあクト！』シリーズ。技術同人誌として異例の累計2.5万部を突破、現在BOOTH技術書カテゴリでも上位を独占しています。現場のエンジニアの方々から絶大な支持をいただき、各社で読書会が開かれたり、新人研修用の教材として使われたりしています。フロントエンド初心者の新人とその先輩の会話によって解説が進むので頭に入りやすく、React学習時に引っかかりやすいポイントや抱きがちな疑問に対しても丁寧にフォローしていきます。どこよりもわかりやすく、どこよりも実践的なReact本の決定版との自負を持って提供しています。' Below the bio, there is a message: '🔔 くるみ割り書房をフォローしていただくと、新刊・更新情報や紙の本の入荷状況などをメッセージで随時お知らせします。' At the bottom, there are social media icons for YouTube, X (Twitter), and a globe icon followed by the text 'くるみ割り書房公式...'. The top of the page has a search bar, login links, and language selection.

「りあクト！」シリーズを刊行している「くるみ割り書房」では、新刊および改版、紙本の増刷時の入荷状況、その他読者の方へのお知らせをBOOTHのダイレクトメッセージで随時お送りしております。

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

りあクト！TypeScriptで始めるつらくないReact開発 第4.1版

【①言語・環境編】

2024年1月22日 初版第1刷発行
2024年1月22日 電子版バージョン 1.0.0

著 者 大岡 由佳
発 行 元 くるみ割り書房
連 絡 先 oukayuka@gmail.com
印 刷 ・ 製 本 株式会社栄光 ブックネクスト事業部

本書を無断で複写・複製、転載、データ配信、オークションやフリマアプリに出品することを固く禁じます。

©2024 Yuka O'oka / Klemiware Books