
仕事ですぐに使える TypeScript

Future Corporation

2021 年 03 月 16 日

TypeScript の世界を知る

第 1 章	前書き	3
1.1	本ドキュメントの位置づけ	3
1.2	TypeScript のウェブ開発における位置づけ	4
1.3	TypeScript を選んで開発すべき理由	5
1.4	ライセンス	6
1.5	本書の構成と学習の準備	6
1.6	JavaScript のバージョン	7
1.7	TypeScript と互換性	8
1.8	本書の参考文献など	9
第 2 章	Node.js エコシステムを体験しよう	11
2.1	Node.js のインストール	12
2.2	package.json の作成と ts-node のインストール	13
2.3	プロジェクトフォルダ共有後の環境構築	14
2.4	インストールしたコマンドの実行	15
2.5	TypeScript の環境設定	16
2.6	エディタ環境	16
2.7	ts-node を使った TypeScript のコードの実行	17
2.8	まとめ	17
第 3 章	変数	19
3.1	三種類の宣言構文	19
3.2	変数の型定義	20
3.3	より柔軟な型定義	21
3.4	変数の巻き上げ	22
3.5	変数のスコープ	23
3.6	まとめ	24
第 4 章	プリミティブ型	25
4.1	boolean リテラル	25
4.2	数値型	27
4.3	string リテラル	32
4.4	undefined と null	35

4.5	まとめ	36
第 5 章	複合型	37
5.1	配列	37
5.2	オブジェクト	46
5.3	まとめ	52
第 6 章	基本的な構文	53
6.1	制御構文	53
6.2	式	57
6.3	まとめ	58
第 7 章	基本的な型付け	59
7.1	一番手抜きな型付け: any	59
7.2	未知の型: unknown	60
7.3	型に名前をつける	61
7.4	関数のレスポンスや引数で使うオブジェクトの定義	61
7.5	オブジェクトの属性の修飾: オプション、読み込み専用	62
7.6	属性名が可変のオブジェクトを扱う	63
7.7	A かつ B でなければならない	64
7.8	タグ付き合併型: パラメータの値によって必要な属性が変わる柔軟な型定義を行う	64
7.9	型ガード	67
7.10	keyof と Mapped Type: オブジェクトのキーの文字列のみを許容する動的な型宣言	70
7.11	インタフェースを使った型定義	71
7.12	もし TypeScript の型を付けるのがコストが高いと感じたら?	72
7.13	まとめ	73
第 8 章	関数	75
8.1	関数の引数と返り値の型定義	76
8.2	関数を扱う変数の型定義	78
8.3	関数を扱う変数に、デフォルトで何もしない関数を設定する	79
8.4	デフォルト引数	80
8.5	関数を含むオブジェクトの定義方法	81
8.6	クロージャと this とアロー関数	82
8.7	this を操作するコードは書かない (1)	84
8.8	即時実行関数はもう使わない	85
8.9	まとめ	85
第 9 章	その他の組み込み型・関数	87
9.1	Date	88
9.2	RegExp	97
9.3	JSON	101

9.4	URL と URLSearchParams	106
9.5	時間のための関数	108
第 10 章	クラス	109
10.1	JavaScript と Java 風オブジェクト指向文法の歴史	109
10.2	用語の整理	110
10.3	基本のクラス宣言	110
10.4	アクセス制御 (public/protected/private)	111
10.5	コンストラクタの引数を使ってプロパティを宣言	112
10.6	static メンバー	113
10.7	インスタンスクラスフィールド	114
10.8	読み込み専用の変数 (readonly)	115
10.9	メンバー定義方法のまとめ	116
10.10	継承/インタフェース実装宣言	117
10.11	クラスとインタフェースの違い・使い分け	119
10.12	デコレータ	120
10.13	まとめ	120
第 11 章	非同期処理	123
11.1	非同期とは何か	124
11.2	コールバックは使わない	125
11.3	非同期と制御構文	127
11.4	Promise の分岐と待ち合わせの制御	128
11.5	ループの中の await に注意	129
11.6	非同期で繰り返し呼ばれる処理	130
11.7	まとめ	131
第 12 章	例外処理	133
12.1	TypeScript の例外処理構文	133
12.2	Error クラス	135
12.3	例外処理とコードの読みやすさ	137
12.4	リカバリー処理の分岐のためにユーザー定義の例外クラスを作る	138
12.5	例外処理を使わないエラー処理	139
12.6	非同期と例外処理	140
12.7	例外とエラーの違い	141
12.8	例外処理のハンドリングの漏れ	142
12.9	例外処理機構以外で例外を扱う	144
12.10	まとめ	145
第 13 章	モジュール	147
13.1	用語の整理	147
13.2	基本文法	150

13.3	中級向けの機能	153
13.4	ちょっと上級の話題	154
13.5	まとめ	158
第 14 章	<code>console.log</code> によるログ出力	159
14.1	<code>console.log/info/warn/error()</code>	159
14.2	<code>console.table/dir()</code>	161
14.3	その他のメソッド	162
14.4	まとめ	162
第 15 章	ジェネリクス	163
15.1	ジェネリクスの書き方	163
15.2	ジェネリクスの型パラメータに制約をつける	164
15.3	型パラメータの自動解決	165
15.4	ジェネリクスの文法でできること、できないこと	166
15.5	型変換のためのユーティリティ型	167
15.6	<code>any</code> や <code>unknown</code> 、合併型との違い	169
15.7	まとめ	170
第 16 章	関数型指向のプログラミング	171
16.1	イミュータブル	171
第 17 章	クラス上級編	173
17.1	アクセッサ	173
17.2	抽象クラス	175
17.3	まとめ	175
第 18 章	リアクティブ	177
第 19 章	高度なテクニック	179
第 20 章	ソフトウェア開発の環境を考える	181
20.1	環境構築できるメンバーがいなかったらどうするか？	183
第 21 章	基本の環境構築	185
21.1	作業フォルダの作成	187
21.2	TypeScript の環境整備	188
21.3	Prettier	189
21.4	ESLint	191
21.5	テスト	196
第 22 章	ライブラリ開発のための環境設定	199
22.1	ディレクトリの作成	200
22.2	ビルド設定	200

22.3	ライブラリコード	202
22.4	まとめ	202
第 23 章	CLI ツール・ウェブサーバー作成のための環境設定	205
23.1	Node.js の環境構築	205
23.2	Deno	209
第 24 章	CI（継続的インテグレーション）環境の構築	211
第 25 章	成果物のデプロイ	213
25.1	npm パッケージとしてデプロイ	213
25.2	サーバーにアプリケーションをデプロイ	214
25.3	Docker イメージの作成	216
25.4	CLI/ウェブアプリケーションのイメージ作成	218
25.5	ウェブフロントエンドの Docker イメージの作成	223
25.6	Kubernetes へのデプロイ	226
25.7	まとめ	230
第 26 章	使用ライブラリのバージョン管理	231
26.1	バージョンとは	231
26.2	バージョン選びの作戦	233
26.3	バージョンアップの方法	234
26.4	バージョンアップ時のトラブルを減らす	236
26.5	なぜバージョンを管理する必要があるのか	237
26.6	まとめ	239
第 27 章	ブラウザ環境	241
27.1	ウェブアプリケーションにおけるビルドツールのゴール	241
第 28 章	ブラウザ関連の組み込み型	243
28.1	Polyfill、Ponyfill	243
28.2	fetch	243
28.3	FormData	243
28.4	EventListener	244
28.5	EventSource と WebSocket	244
28.6	LocalStorage と sessionStorage	244
第 29 章	React の環境構築	245
29.1	create-react-app による環境構築	245
29.2	Next.js	246
29.3	React の周辺ツールのインストールと設定	247
29.4	UI 部品の追加	248
29.5	React+Material UI+TypeScript のサンプル	250

29.6	React と TypeScript	253
29.7	Redux と TypeScript	255
29.8	React と Redux の非同期アクセス	258
29.9	React の新しい書き方	262
29.10	まとめ	268
第 30 章	Vue.js の環境構築	269
30.1	クラスベースのコンポーネント	271
30.2	関数ベースのコンポーネント作成	272
30.3	Vue.js を使った jQuery のリプレイス	273
30.4	まとめ	280
第 31 章	Parcel を使ったウェブ開発	281
31.1	Parcel とは	281
31.2	React 環境	282
31.3	WebComponents 開発環境	285
第 32 章	Electron アプリケーションの作成	287
32.1	React+Electron の環境構築の方法	287
32.2	配布用アプリケーションの構築	290
32.3	デバッグ	291
32.4	レンダープロセスとメインプロセス間の通信	292
32.5	まとめ	293
第 33 章	おすすめのパッケージ・ツール	295
33.1	TypeScript Playground	295
33.2	ビルド補助ツール	295
33.3	コマンドラインツール用ライブラリ	297
33.4	アルゴリズム関連のライブラリ	297
第 34 章	貢献者	299
34.1	Pull Request/Issue をくださった方々	299
34.2	フィードバックをくださった方々	300
第 35 章	Indices and tables	303
	索引	305

注釈: 本ドキュメントは、まだ未完成ですが、ウェブフロントエンドの開発を学ぶときに、JavaScript を経由せずに、最初から TypeScript で学んでいく社内向けコンテンツとして作成されはじめました。基本の文法部分以外はまだ執筆されていない章もいくつもあります。書かれている章もまだまだ内容が追加される可能性がありますし、環境の変化で内容の変更が入る可能性もあります。

書籍の原稿は GitHub 上で管理しております。もし Typo を見つけてくださった方がいらっしゃいましたら、[GitHub 上で連絡](#)をお願いします*¹。reST ファイルだけ修正してもらえれば、HTML/PDF の生成までは不要です。フィードバックなども歓迎しております。

*¹ <https://github.com/future-architect/typescript-guide/pulls>

第 1 章

前書き

1.1 本ドキュメントの位置づけ

本ドキュメントは、まだ未完成ですが、ウェブフロントエンドの開発を学ぶときに、JavaScript を経由せずに、最初から TypeScript で学んでいくコンテンツとして作成されはじめました。TypeScript は基本的に JavaScript の上位互換であり、JavaScript には歴史的経緯で古い書き方も数多くあります。そのため、文法を学ぶだけではなくよりモダンな書き方をきちんと学べることを目指しています。

現在、B2B 企業であっても、B2C 企業であっても、どこの企業もウェブのフロントエンドの求人が足りないという話をしています。

以前は、企業システムのフロントエンドというと、一覧のテーブルがあって、各行の CRUD（Create: 作成、Read: 読み込み、Update: 更新、Delete: 削除）の操作を作る、といったようにハンコを押すように量産するものでした。また、画面はテンプレートを使ってサーバーで作成して返すものでした。

しかし、ユーザーがプライベートで触れるウェブの体験というものもリッチになり、それに呼応するようにフロントエンドのフレームワークやら開発技術も複雑化しました。よりリアルタイムに近い応答を返す、画面の切り替えを高速にする、動的に変化する画面で効率よく情報を提供する、といったことが普通に行われるようになりました。

コンシューマー向けのリッチなウェブになったユーザーが、1 日の大半を過ごす仕事で触れるシステムが時代遅れな UI というのは良いものではありません。UI に関するコモンセンスから外れれば外れるほど、ユーザーの期待から外れれば外れるほど、「使いにくい」システムとなり、ストレスを与えてしまいます。

近年、React、Vue.js、Angular といった最新のフレームワークの導入は、企業システムであっても現在は積極的に行われるようになってきています。しかし、フロントエンド側での実装の手間暇が多くなったり、Bootstrap と jQuery 時代のように、ボタンを押した処理だけ書く、というのと比べると分量がかなり増えます。分量が増える分、モジュール分割の仕方を工夫したり、きちんと階層に分離したアーキテクチャを採用したり、きちんとした設計が求められるようになります。

フューチャーアーキテクトおよびフューチャーでは開発の方法論を定めて効率よく開発を行うことを実現してきており、典型的なサーバー側で処理を多く行うアプリケーションの場合は数多くの成功を納めてきています。一方で、前述のようにフロントエンド側の比重が高まると、当然のことながらプロジェクトに占めるフロントエンドの

開発者の割合を高める必要があります。フューチャーではコンピューターを専門で学んできた学生以外にも多様な学生を受け入れて研修を行なっています。新卒教育の時間は限られているため、あまり多くのことを学んでもらうのは困難です。現在はサーバー側の知識を中心に学んでいます。

サーバーサイド Java や SQL などはほぼすべての社員が身につけていますが、フロントエンド側の開発を行なってもらうには新たな学習の機会の提供が必要です。趣味の時間で勝手に勉強しておいてね、というのは企業活動の中では認められません（趣味でやることを止めるものではありません）。業務遂行のために必要なリソース、知識は業務時間中に得られる体制を整えなければなりません。これまでは、お客様のニーズ的に高度なフロントエンドが必要な案件では、一部の趣味でやっていたメンバーやら、キャリア入社のメンバー、あるいは現場でそれぞれ独自に学んだりといったゲリラ戦で戦ってきましたが、お客様の期待にもっと答えられる体制を築くには、きちんと学べるコンテンツや教育体制の構築が必要ということが、フューチャーの技術リーダー陣の共通見解となっています。

フロントエンド開発の難しい点は、コードを書くのよりも環境構築の難易度が高く、また、その環境構築を行わないとコードが書き始められないという点にあります。また、JavaScript 向けのパッケージで TypeScript 用に型情報をコンパイラに教えてくれる定義ファイルが提供されていないと、開発時にそれも整備しなければなりません。しかし、これも初心者がいきなり手を出すのは厳しいものがあります。近年、コードジェネレータなどが整備されてきているため、Vue.js や React、Angular といった人気のフレームワークと一緒に利用するのはさほど難しくなくなっていますが、その部分はチームでシニアなメンバーが行うものとして、まずは書き方にフォーカスし、次に型定義ファイル作成、最後に環境構築というステップで説明を行い、必要な場所をつまみ食いしてもらえるようにすることを目指しています。

1.2 TypeScript のウェブ開発における位置づけ

フロントエンドの開発がリッチになってきていると同時に、開発におけるムーブメントもあります。それはブラウザで動作する JavaScript を直接書かなくなっているということです。JavaScript は 2015 年より前は保守的なアップデートがおこなわれていました。Netscape 社（現 Mozilla）が開発し、企業間のコンソーシアムで当初仕様が策定されていました。クラスを導入するという大規模アップデートを一度は目指したものの（ECMAScript 4）、その時は頓挫しました。しかし、ECMAScript 2015 時点で大幅なアップデートが加えられ、よりオープンなコミュニティ、TC39 で議論されるようになりました。

しかし、ブラウザの組み込み言語である以上、サーバーアプリケーションの Java や Python のようにランタイムをアプリケーションに合わせて維持したり更新するといったことはできません。そのため、高度な文法を備える言語でコードを書き、トラディショナルな JavaScript に変換するというアプローチが好まれるようになりました。CoffeeScript などが一時広く使われたものの、現在よく利用されているのは Babel と TypeScript です。

Babel（旧名 6to5）はオープンソースで開発されている処理系で、ECMAScript の最新の文法を解釈し、古い JavaScript 環境でも動作するように変換します。プラグインを使うことで、まだ規格に正式に取り入れられていない実験的な文法を有効にすることもできます。実際、言語の仕様策定の間でも参考実装として活発に使われています。TypeScript は Microsoft 社が開発した言語で、ECMAScript を土台にして、型情報を付与できるようにしたものです。一部例外はありますが、TypeScript のコードから型情報を外すとほぼそのまま JavaScript になります。言語仕様の策定においても、参考実装の 1 つとして扱われています。

大規模になってくると、型があるとコーディングが楽になります。型を書く文の手間は多少増えますが、昔の静的型付き言語と異なり、TypeScript は「明示的に型がわかる場面」では型を省略して書くことができます。また、動的なオブジェクトなど、JavaScript でよく登場する型もうまく扱えるように設計されています。そのような使い勝手の良さもあり、現在は採用数が伸びています。有名な OSS も実装を TypeScript に置き換えたり、企業でも積極的な活用が進んでいます。

Babel にも、プラグインを追加して型情報を追加できる flowtype という Facebook 製の拡張文法がありますが、ライブラリに型情報をつけるのは、手作業で整備していかなければならず、シェアが高いほど型情報が手に入りやすく、開発の準備のために型定義ファイルを作る時間を別途かけなくて済みます。現時点で型定義ファイルの充実度が高いのは TypeScript です。

まとめると

- 新しい記法を使うが、ブラウザの互換性を維持するコードを書く手法としてコンパイラを使うのが当たり前になってきた
- 大規模になると（小規模でも）、型情報があるとエラーチェックが実装中に行われるので開発がしやすくなる
- 型を持った JavaScript には TypeScript と flowtype の 2 つがあるが、シェアが高いのが TypeScript

です。

1.3 TypeScript を選んで開発すべき理由

「型情報を省くとほぼ JavaScript」なのに、なぜわざわざ別のツールを導入してまで TypeScript を使うのでしょうか？

型情報が得られることで開発が加速される、というので十分に元はとれます。また、最初から TypeScript を書くメリットとしては、JavaScript のライブラリをコンパイルして作る際に、型定義ファイルも同時生成できて、無駄がない点にあります。最初から書くことで TypeScript 資産がたまり、さらに TypeScript の開発が楽になります。

一方、JavaScript を開発に使う場合も、各ブラウザのエンジンのサポートしている機能が古いことがあり、Babel を利用して、最新の JavaScript から互換性の高い JavaScript への変換を行うのが通例です。そのため、JavaScript を選んでも、TypeScript を選んでも、何かしらのツール導入が必要なことには変わりありません。それであれば、TypeScript を入れた方が付加価値がさらにあります。

また、JavaScript は極めて柔軟な言語です。関数の引数の型もどんなものでも受け付けるとか、引数によって内部の動作が大きく切り替わるようなコード（メソッド・オーバーロード）も書こうと思えば書けます。また TypeScript の機能を駆使して、そのような関数に型情報を付与することもできます。しかし、TypeScript で最初から書けば、そのような型付けに苦勞するような、トリッキーな書き方がしにくくなります。結果として型定義のメンテナンスに時間を取られることが減ります。

それ以外にも、型情報が分かった上で変換を行うため、ループ構文など、いくつかのコードを変換するときに、Babel よりも実行効率の良い JavaScript コードを生成することもわかっています。

本ドキュメントでは、大規模化するフロントエンド開発の難易度を下げ、バグが入り込みにくくなる TypeScript を使いこなせるように、文法や環境構築などを紹介していきます。

1.4 ライセンス



本ドキュメントは [クリエイティブ・コモンズ 4.0 の表示 - 継承 \(CC BY-SA 4.0\)](https://creativecommons.org/licenses/by-sa/4.0/) ^{*1} の元で公開します。修正や足したいコンテンツは Pull Request を出していただけるとうれしいのですが、改変の制約はありませんのでフォークしていただくことも可能です。また、商用利用の制限はありません。

著作権者名は「フューチャー株式会社 (Future Corporation)」をお願いします。

なお、LICENSE ファイルは [Creative Commons Markdown](https://creativecommons.org/licenses/by-sa/4.0/) から引用させていただきました。

1.5 本書の構成と学習の準備

最初に説明した通り、本ドキュメントは TypeScript ファーストで説明していきます。現在でも JavaScript を書く人の多くが TypeScript を採用しており、その数は増えているため今後は TypeScript でフロントエンドなどの開発のキャリアをスタートする人も増えるでしょう。本書は「全員が TypeScript を書くようになった」時代がくることを想定して書いています。JavaScript との差異があるところは適宜補足します。

本ドキュメントではまず TypeScript の基本的な文法を学んでいきます。TypeScript は JavaScript の記法はすべてサポートしていますが、JavaScript も歴史のある言語なので今となっては古い書き方も増えています。よく使われていたが今はよりよい書き方があることもある時はモダンな書き方が学べるようにしています。

他の人が使うライブラリで必要となるような高度な文法は中級編として別のセクションに分けています。こちらは最初は飛ばしてもかまいません。

その後は各環境向けの Tips を紹介します。共通部分をみながら必要な箇所をピックアップして読めるようにしています。ここでは VSCode を用いた開発環境構築やツール整備から始まり、それぞれの環境固有のトピックについて触れていきます。

本ドキュメントは TypeScript のエコシステムまで含めたすべてを説明しようとするものではありません。例えば、既存の JavaScript のライブラリのための型定義ファイルを作成する方法については紹介しません。時間が経てば有名ライブラリについてはほぼ網羅されることを期待していますし、自作していくときはゼロから TypeScript でいけば、型定義ファイルは自動生成されるので不要です。

環境構築まではエディタなどの準備は不要です。文法を学ぶときは、本家が提供している Playground が便利です。

^{*1} <http://creativecommons.org/licenses/by-sa/4.0/deed.ja>

- TypeScript Playground: <https://www.typescriptlang.org/play/>

Playground は徐々に機能が追加されています。最初はちょっとした変換だけだったものが、コンパイルオプションがいろいろ選べるようになったり、処理系のバージョンが選択できるようになりました。2020 年 8 月に公開された V3 では、変換結果以外に、TypeScript が解釈した型情報 (.d.ts)、エラー、実行ログも確認できて、学習ツールとして使いやすくなっています。プラグインも実行できるようになっています。

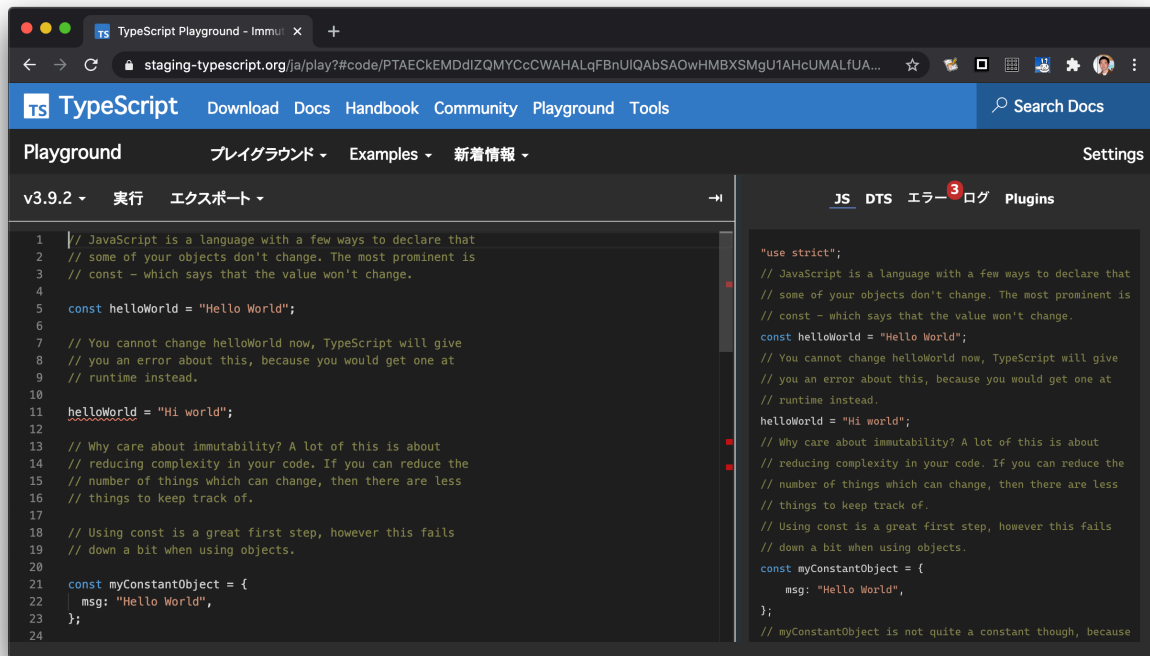


図 1 最新の Playground。

1.6 JavaScript のバージョン

最近では JavaScript の仕様はコミュニティで議論されています。TC39 という ECMA 内部の Technical Committee がそれにあたります。クラスなどの大幅な機能追加が行われた ES6 は、正式リリース時に ECMAScript 2015 という正式名称になり、それ以降は年次でバージョンアップを行なっています。

議論の結果や現在上がっている提案はすべて GitHub 上で見るができます。

- <https://github.com/tc39/proposals>

機能単位で提案が行われます。最初は stage 0 から始まり、stage 1、stage 2 とステップがあがっていきます。最初はアイデアでも、徐々にきちんとした仕様やデモ、参考実装など動くようにすることが求められていきます。stage 1 が提案、stage 2 がドラフト、stage 3 がリリース候補、stage 4 が ECMAScript 標準への組み込みになります。1 月ぐらいに stage 4 へ格上げになる機能が決定され、6 月に新しいバージョンがリリースされます。

TypeScript も基本的には型がついた ECMAScript として、ECMAScript の機能は積極的に取り込んでいます。また、いくつか stage 2 や stage 3 の機能も取り込まれていたりします。

1.7 TypeScript と互換性

インターネット上ですべてのユーザーが見られるサイトを作る場合、現在の機能的な下限は Internet Explorer 11^{*2} です。Google の検索エンジンのボットもこれとほぼ同等機能 (const、let ありのクラスなし) の Chrome 41 で固定されています^{*3}。それ以外には、バージョンアップがもう提供されていない iOS や Android のスマートフォンの場合に最新の機能が使えないことがあります。

100% のブラウザとの互換性を維持するのは開発リソースがいくらあっても足りないため、捻出できる工数と相談しながら、サポート範囲を決めます。ブラウザのバージョンごとにどの機能が対応しているかは ECMAScript Compatibility Table^{*4}のサイトで調べられます。

新しいブラウザのみに限定できるイントラネットのサービスや、Node.js 以外は、Babel なり、TypeScript などのコンパイラを使い、変換後の出力として古いブラウザ向けの JavaScript コードに変換して出力するのが現在では一般的です。Lambda、Cloud Functions、Google App Engine などは、場合によっては少し古いバージョンの Node.js を対象にしなければならないため、これも変換が必要になるかもしれません^{*5}。

TypeScript の場合はほぼ最新の ECMAScript の文法に型をつけて記述できますが、コンパイル時に出力するコードのバージョンを決めることができます。デフォルトでは ES3 ですが、ES5、ES2015 から ES2018、ESNEXT とあわせて、合計 7 通りの選択肢が取れます。一部の記述はターゲットが古い場合にはオプションが必要になることもあります。最低限、ES5 であれば、新旧問わずどのブラウザでも問題になることはないでしょう。

ただし、TypeScript が面倒を見てくれるのは文法の部分だけです。たとえば、Map や Set といったクラスは ES5 にはありませんし、イテレータを伴う Array のメソッドもありません。

TypeScript には tsconfig.json というコンパイラの動作を決定する定義ファイルがあります。ブラウザの可搬性を維持しつつ、これらの新しい要素を使いたい場合には別途そこをサポートするものを入れる必要があります。現在、その足りないクラスやメソッドを追加するもの (Polyfill と呼ばれる) で、一番利用されるのが core-js^{*6}で、Babel から使われているようです。

出力ターゲットを古くすると、利用できるクラスなども一緒に古くなってしまうため、対策が必要です、まずは、ES2017 や ES2018 などのバージョンのうち、必要なクラスを定義しているバージョンがどれかを探してきます。どのバージョンがどの機能をサポートしているかは、前述の compat-table が参考になります。

ターゲットに es5 を選ぶと、lib には ["DOM", "ScriptHost", "ES5"] が定義されます。lib は使える

^{*2} Microsoft 社が Edge を Chromium ベースにすることを発表し、2020 年現在、配布が開始されています。これまでの Edge と異なり、Windows 7 以降のすべての Windows で提供されるようになります。IE モードも搭載されて IE とのリプレースも行えるようになるため、IE 基準で考える必要はなくなっていく予定です。

^{*3} Google I/O 2019 で、これが現時点の最新版と同じ Chrome 74 に更新されることが発表されています。

^{*4} <http://kangax.github.io/compat-table/es6/>

^{*5} Lambda は長らく Node.js 6 というかなり古いバージョンを使っていましたが 10 が提供されて 6 はサポート終了になり、Node.js 6 ベースのタスクの新規作成と更新ができなくなりました。

^{*6} <https://www.npmjs.com/package/core-js>

クラスとかメソッド、その時の型などが定義されているもので、これを増やしたからといってできることが増えたりはしませんが、「これはないよ」というコンパイラがエラーを出力するための情報源として使われます。この "ES5" には、そのバージョンで利用できるクラスとメソッドしかないため、次のように ES2017 に置き換えます。

リスト 1 tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["DOM", "ScriptHost", "ES2017"]
  }
}
```

こうすると、Map などを使っても TypeScript のエラーにはならなくなりますが、変換されるソースコードには Map が最初からあるものとして出力されてしまいます。あとは、その Map を利用している場所に、import を追加すると、その機能がない環境でも動作するようになります。core-js のオプションが知りたい場合は、core-js のサイトの README に詳しく書かれています。

```
import "core-js/es6/map";
```

1.8 本書の参考文献など

ECMAScript の仕様および、MDN、TypeScript の仕様などは一番のリファレンスとしています。

- ECMAScript 規格: <https://www.ecma-international.org/publications/standards/Ecma-262.htm>
- MDN: <https://developer.mozilla.org/ja/docs/Glossary/JavaScript>
- 本家サイト: <http://www.typescriptlang.org/>

下記のサイトは最近までは Compiler Internal などを書いてあるサイトとしてしか思っていなくて、詳しくは見えていませんでしたが、現在ではかなり充実してきています。現時点では参考にはしていませんでしたが、今後は参考にすることがあります。

- TypeScript Deep Dive: <https://basarat.gitbooks.io/typescript/>
- TypeScript Deep Dive 日本語版: <https://typescript-jp.gitbook.io/deep-dive/>

本書のベースとなっているのは、本原稿を執筆した渋川が Qiita に書いたエントリーの [イマドキの JavaScript の書き方 2018^{*7}](#) と、それを元にして書いた [Software Design 2019 年 3 月号の JavaScript 特集](#) です。それ以外に、状況別の TypeScript の環境構築について書いた [2019 年版: 脱 Babel! フロント/JS 開発を TypeScript に移行するための環境整備マニュアル^{*8}](#) も内包していますし、他のエントリーも細々と引用しています。

^{*7} <https://qiita.com/shibukawa/items/19ab5c381bbb2e09d0d9>

^{*8} <https://qiita.com/shibukawa/items/0a1aaf689d5183c6e0f1>

これらの執筆においてもそうですが、本書自体の執筆でも、ウェブ上で多くの議論をしてくれた人たちとの交流によって得られた知識ふんだんに盛り込まれていますので、ここに感謝申し上げたいと思います。

第 2 章

Node.js エコシステムを体験しよう

TypeScript は JavaScript への変換を目的として作られた言語です。公式の処理系がありますが、それに変換すると、JavaScript が生成されます。勉強目的で実行するには、現在のところ、いくつかのオプションがあります。このなかで、とりあえず安定して使えて、比較的簡単なのは `ts-node` です。

- TypeScript のウェブサイトの `playground`^{*1}: 公式のコンパイラで変換してブラウザで実行
- `tsc + Node.js`: 公式のコンパイラで変換してから Node.js で実行
- `babel + ts-loader + Node.js`: Babel 経由で公式のコンパイラで変換してから Node.js で実行
- `babel + @babel/preset-typescript + Node.js`: Babel で型情報だけを落として簡易的に変換して Node.js で実行
- `ts-node`: TypeScript を変換してそのまま Node.js で実行する処理系
- `Deno`: TypeScript をネイティブサポートした処理系（2020-05-13 に 1.0 がリリースされました^{*2}）

Node.js は JavaScript にファイル入出力やウェブサーバー作成に必要な APIなどを足した処理系です。本章では、TypeScript の環境整備をするとともに、Node.js を核としたエコシステムの概要を説明します。プログラミング言語を学んで書き始める場合、言語の知識だけではどうにもなりません。どこにソースコードを書き、どのようにビルドツールを動かし、どのように処理系を起動し、どのようにテストを行うかなど、言語周辺のエコシステムを学ばないと、どこから手をつけて良いかわかりません。本章で紹介するエコシステムの周辺ツールや設定ファイルは以下の通りです。

- Node.js: 処理系
- `npm` コマンド: パッケージマネージャ
- `package.json`: プロジェクトファイル
 - 依存パッケージの管理
 - `scripts` で開発用のタスクのランチャーとして利用

^{*1} <https://www.typescriptlang.org/play/index.html>

^{*2} <https://deno.land/v1>

- npx コマンド: Node.js 用の npm パッケージで提供されているツールの実行
- TypeScript 関連のパッケージ
 - tsc: TypeScript のコンパイラ (プロジェクト用の TypeScript の設定ファイルの作成)
 - ts-node: TypeScript 変換しながら実行する、Node.js のラッパーコマンド

まずは Node.js をインストールして npm コマンドを使えるようにしてください。なお、本章ではゼロから環境を構築していきますが、ハンズオンのチーム教育などで、構築済みの環境をシェアする場合には次の節は飛ばしてください。コードを書くスキルに対して、環境構築に必要なスキルは数倍難易度が高いです。エコシステムを完全に理解してツール間の連携を把握する必要があります。ただし、そのために必要な知識はコードを書いていくうちに学びます。どうしても難易度が高い作業が最初に来てしまい、苦手意識を広げてしまう原因になってしまいます。環境構築は、本章と、5 章以降で取り扱うので、必要になったら戻って来てください。

2.1 Node.js のインストール

Node.js は公式の <https://nodejs.org> からダウンロードしてください。あるいは、chocolatey や Homebrew、macports などのパッケージマネージャなどを使ってインストールすることも可能です。もし、複数のバージョンを切り替えて検証する場合には nvm が利用できます。ただし、フロントエンド開発においては、コードは変換してから他の環境 (ブラウザやクラウドのサービス) 上で実行されますし、基本的に後方互換性は高く、バージョン間の差もそこまでないため、最新の LTS をとりあえず入れておけば問題ないでしょう (ただし、AWS Lambda などの特定の環境での動作を確認したい場合はのぞく)。

課題: あとで書く

Node.js をインストールすると、標準のパッケージマネージャの npm もインストールされます。もしかしたらパッケージマネージャの種類によってはインストールされない場合もあるので、その場合は追加インストールしてください。

npm コマンドはパッケージのダウンロードのためにインターネットアクセスをします。もし、社内プロキシなどがある場合は npm のインストール後に設定しておくのをおすすめします。

リスト 1 プロキシの設定

```
npm config set proxy http://アカウント名:パスワード@プロキシの URL
npm config set https-proxy http://アカウント名:パスワード@プロキシの URL
```

2.2 package.json の作成と ts-node のインストール

最初に作業フォルダを作り、package.json を作成します。

```
$ mkdir try_ts
$ cd try_ts
$ npm init -y
{
  "name": "try_ts",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

package.json は Node.js を使ったプロジェクトの核となるファイルです。次のような情報が入ります。

- プロジェクト自身のさまざまな情報
- プロジェクトが依存する (実行で必要、もしくは開発に必要) ライブラリの情報
- プロジェクトのビルドやテスト実行など、プロジェクト開発に必要なタスクの実行

他の人が行なっているプロジェクトのコードを見るときも、まずは package.json を起点に解析していくと効率よく探すことができます。この npm init コマンドで作成される package.json は、とりあえずフォルダ名から付与された名前、固定のバージョン (1.0.0)、空の説明が入っています。このファイルはパッケージのリポジトリである npmjs.org にアップロードする際に必要な情報もすべて入ります。仕事のコードやハンズオンのプロジェクトを間違えて公開しないように (することもないと思いますが)、"private": true を書き足しておきましょう。

```
{
  "name": "env",
  "version": "1.0.0",
  "description": "",
  "private": true
}
```

(次のページに続く)

(前のページからの続き)

```
:  
}
```

次に必要なツールをインストールします。npm install で、ts-node と typescript を入れます。--save-dev をつけると、開発に必要だが、リリースにはいらんという意味になります。

```
$ npm install --save-dev ts-node typescript
```

もし、本番環境でも ts-node を使ってビルドしたい、ということがあれば --save-dev の代わりに --save をつけます。

```
$ npm install --save ts-node
```

package.json を見ると、項目が追加されているのがわかりますね。また、package-lock.json という、環境を構築したときの全ライブラリのバージョン情報が入ったファイルも生成されます。このファイルを手で修正することはありません。

```
{  
  "dependencies": {  
    "ts-node": "^8.0.2"  
  },  
  "devDependencies": {  
    "typescript": "^3.3.1"  
  }  
}
```

また、node_modules フォルダができて必要なライブラリなどがインストールされていることがわかります。他の言語と異なり、基本的に Node.js は現在いるフォルダ以外のところにインストールすることはありません(キャッシュはありますが)。複数プロジェクト掛け持ちしているときも、プロジェクト間でインストールするライブラリやツールのバージョンがずれることを心配する必要はありません。

プロジェクトをチーム間で共有するときは、この package.json があるフォルダをバージョン管理にシステムに入れます。ただし、node_modules は配布する必要はありません。 .gitignore などに名前を入れておくと良いでしょう。

2.3 プロジェクトフォルダ共有後の環境構築

チーム内では、git などでプロジェクトのソースコードを共有します。JavaScript 系のプロジェクトでは、その中に package.json と package-lock.json があり、デプロイ時に環境を作ったり、共有された人は環境を手元で再現したりするのが簡単にできます。

以下は、環境変数 NODE_ENV が未設定または production 以外の場合の動作です。

<code>npm install</code>	dependencies と devDependencies の両方をインストールする。
<code>npm install --prod</code>	dependencies のみをインストールする。
<code>npm ci</code>	dependencies と devDependencies の両方をインストールする。package-lock.json は更新しない。
<code>npm ci --prod</code>	dependencies のみをインストールする。package-lock.json は更新しない。

2.4 インストールしたコマンドの実行

npm コマンドでインストールするパッケージは、プログラムから使うライブラリ以外に実行できるコマンドを含むものがあります。先ほどインストールした typescript と ts-node は両方ともこれを含みます。コマンドは、node_modules/.bin 以下にインストールされています。これを直接相対パスで指定しても良いのですが、専用のコマンドもあります。

ts-node を気軽に試す REPL（1 行ごとに実行されるインタプリタ）の実行もできます。

```
$ npx ts-node
> console.log('hello world')
hello world
```

package.json の scripts のセクションに登録すると、npm コマンドを使って実行できます。

```
"scripts": {
  "start": "ts-node"
}
```

"scripts"にはオブジェクトを書き、その中にはコマンドが定義できます。ここでは start コマンドを定義しています。コマンドが実行されたときに実行されるコードを書けます。ここでは node_modules/.bin 以下のコードをパスを設定せずに書くことができます。npm run [コマンド名] とシェルで実行すると、この scripts セクションのコマンドが実行されます。

```
$ npm run start
> console.log('hello world')
hello world
```

だいたい、次のようなコマンドを定義することが多いです。

- start / serve: パッケージがウェブアプリケーションを含む場合はこれを起動
- test: テストを実行
- lint: コードの品質チェックを行う
- build: ビルドが必要なライブラリではビルドを実行して配布できるようにする

ビルドツールや処理系、テストフレームワークなどは、プロジェクトによって千差万別ですが、この `scripts` セクションを読むと、どのようにソースコードを処理したり、テストしたりしているかがわかります。これは、プロジェクトのコードを読むための強い武器になります。

また、このコマンド実行までは Windows だろうが、Linux だろうが、macOS だろうが、どれでもポータブルに動作します。Node.js と npm コマンドさえあれば、開発機 (Windows、macOS)、CI サーバー (Linux)、本番環境 (Linux) で動作します。もちろん、中で動作させるプログラムに、Node.js 以外の OS のコマンドを書くとそのポータビリティは下がりますが、それに関してはおすすめパッケージの中でポータブルな `scripts` セクションを書くのに使えるパッケージを紹介します。

2.5 TypeScript の環境設定

TypeScript を使うには、いくつか設定が必要です。JavaScript 系のツールのビルドは大きく分けて、2 つのフェーズがあります。

- コンパイル: TypeScript や最新の JavaScript 文法で書かれたコードを、実行環境にあわせた JavaScript に変換
- バンドル: ソースコードは通常、整理しやすいクラスごと、コンポーネントごとといった単位で分けて記述します。配布時には 1 ファイルにまとめてダウンロードの高速化、無駄な使われてないコードの排除が行われます。

前者のツールとしては、TypeScript や Babel を使います。後者は、webpack、Browserify、Rollup、Parcel などがあります。ただし、後者は大規模なアプリケーションでなければ必要ありませんので、5 章以降で紹介します。

何も設定せずとも、TypeScript のコンパイルは可能ですが、入力フォルダを設定したい、出力形式を調整したい、いくつかのデフォルトでオフになっている新しい機能を使いたいなどの場合は設定ファイル `tsconfig.json` を作成します。このファイルの雛形は TypeScript の処理系を使って生成できます。

```
$ npx tsc --init
message TS6071: Successfully created a tsconfig.json file.
```

あとはこの JSON ファイルを編集すれば、コンパイラの動作を調整できます。TypeScript を Node.js で実行するだけであれば細かい設定は不要ですが、4 章ではオプションを使わないといけない文法にもついても紹介します。

2.6 エディタ環境

現在、一番簡単に設定できて、一番精度の高い補完・コードチェックが自動で行われるのが Visual Studio Code です。Windows ユーザーも、Linux ユーザーも、macOS ユーザーも、これをダウンロードしてインストールしておけば間違いありません。何も拡張を入れなくても動作します。

プロジェクトごとの共通の設定も、`.vscode` フォルダに設定を書いてリポジトリに入れるだけで簡単にシェアできる点も、プロジェクトで使うのに適しています。よりアドバンスな設定やツールに関しては環境構築の章で紹介し

ます。

2.7 ts-node を使った TypeScript のコードの実行

それでは適当なコードを書いて実行してみましょう。本来はこのコードは JavaScript と完全互換で書けるのですが（次章で解説します）、あえて型を定義して、通常の Node.js ではエラーとなるようにしています。

リスト 2 最初のサンプルコード (first.ts)

```
const personName: string = '  小心者  ';\n\nconsole.log(`Hello ${personName}!`);
```

実行するには npx 経由で ts-node コマンドを実行します。

```
$ npx ts-node first.ts\nHello  小心者 !
```

今後のチュートリアルでは基本的にこのスタイルで実行します。

2.8 まとめ

本章では次のようなことを学んで来ました。

- JavaScript のエコシステムと package.json
- サンプルを動かすための最低限の環境設定

次章からはさっそくコーディングの仕方を学んで行きます。

第 3 章

変数

TypeScript と JavaScript の一番の違いは型です。型が登場する場面は主に 3 つです。

- 変数 (プロパティも含む)
- 関数の引数
- 関数の戻り値

本章ではまず変数について触れ、TypeScript の型システムの一部を紹介します。

関数については関数の章で説明します。型システムの他の詳細については、オブジェクトの型付け（インタフェースの章）、クラスの型付け（クラスの章）、既存パッケージへの型付けの各章で説明します。

3.1 三種類の宣言構文

変数宣言には `const`、`let` があります。 `const` をまず使うことを検討してください。変数は全部とりあえず `const` で宣言し、再代入がどうしても必要なところだけ `let` にします。変わる必要がないものは「もう変わらない」と宣言することで、状態の数が減ってコードの複雑さが減り、理解しやすくなります。変数を変更する場合は `let` を使います。なお、この `const` は、多くの C/C++ 経験者を悩ませたオブジェクトの不変性には関与しないため、再代入はできませんが `const` で宣言した変数に格納された配列に要素を追加したり、オブジェクトの属性変更はできます。そのため、使える場所はかなり広いです。

リスト 1 変数の使い方

```
// 何はともあれ const
const name = "小動物";

// 変更がある変数は let
// 三項演算子を使えば const にもできる
let name;
if (mode === "slack") {
  name = "小型犬";
}
```

(次のページに続く)

(前のページからの続き)

```
} else if (mode === "twitter") {  
    name = "小動物";  
}
```

なお、JavaScript と異なり、未定義の変数に代入すると、エラーになります。

```
undefinedVar = 10;  
// error TS2552: Cannot find name 'undefinedVar'.
```

若者であれば記憶力は強いので良いですが、歳をとるとだんだん弱ってくるのです。また、若くても二日酔いの時もあるでしょうし、風邪ひいたり疲れている時もあると思うので、頑張らないで理解できるように常にしておくのは意味があります。

注釈: 昔の JavaScript は変数宣言で使えるのは `var` のみでした。 `var` はスコープが関数の単位とやや広く、影響範囲が必要以上に広がります。また、宣言文の前にアクセスしてもエラーにならなかったりと、他の宣言よりも安全性が劣ります。現在でも使えますが、積極的に使うことはしないでしょ。

リスト 2 変数の使い方

```
// 古い書き方  
var name = "小動物";
```

3.2 変数の型定義

TypeScript は変数に型があります。TypeScript は変数名の後ろに後置で型を書きます。これは Go、Rust、Python3 などで見られます。一度定義すると、別の型のデータを入れると、コンパイラがエラーを出します。それによってプログラムのミスが簡単に見つかります。また、型が固定されると、Visual Studio Code などのエディタでコード補完機能が完璧に利用できます。

リスト 3 変数への型の定義

```
// name は文字列型  
let name: string;  
  
// 違う型のデータを入れるとエラー  
// error TS2322: Type '123' is not assignable to type 'string'  
name = 123;
```

なお、代入の場合には右辺のデータ型が自動で設定されます。これは型推論と呼ばれる機能で、これのおかげで、メソッドの引数や、クラスや構造体のフィールド以外の多くの場所で型を省略できます。

リスト 4 推論

```
// 代入時に代入元のデータから型が類推できる場合は自動設定される
// 右辺から文字列とわかるので文字列型
let title = "小説家";

// 代入もせず、型定義もないと、なんでも入る（推論ができない）any 型になります。
let address;
// 明示的に any を指定することもできる
let address: any;
```

型については型の章で取り上げます。変数以外にも関数の引数でも同様に型を定義できますが、これについては関数の節で紹介します。

3.3 より柔軟な型定義

TypeScript は、Java や C++、Go などの型付き言語を使ったことがある人からすると、少し違和感を感じるかもしれない柔軟な型システムを持っています。これは、型システムが単にプログラミングのサポートの機能しかなく、静的なメモリ配置まで面倒を見るような言語では不正となるようなコードを書いても問題がないからと言えるでしょう。2 つほど柔軟な機能を紹介します。

- A でも B でも良い、という柔軟な型が定義できる
- 値も型システムで扱える

A でも B でも良い、というのは例えば数値と文字列の両方を受け取れる（が、他のデータは拒否する）という指定です。

リスト 5 数値でも文字列でも受け取れる変数

```
// 生まれの年は数値か文字列
let birthYear: number | string;

// 正常
birthYear = 1980;
// これも正常
birthYear = '昭和';
// 答えたくないのに null... はエラー
birthYear = null;
// error TS2322: Type 'null' is not assignable to type 'string | number'.
```

次のコードは、変数に入れられる値を特定の文字列に限定する機能です。型は `|` で複数並べることができる機能を使って、取りうる値を列挙しています。この複数の状態を取る型を合併型（Union Type）と呼びます。ここで書いていない文字列を代入しようとするとエラーになります。数値にも使えます。

リスト 6 変数に特定の文字列しか設定できないようにする

```
let favoriteFood: "北極" | "冷やし味噌";
favoriteFood = "味噌タンメン"
// error TS2322: Type '"味噌タンメン"' is not assignable to
//   type '"北極" | "冷やし味噌"'.

// 数値も設定可能
type PointRate = 8 | 10 | 20;
// これもエラーに
let point: PointRate = 12;
```

型と値を組み合わせてすることもできます。

```
// 値と型の合併型
let birthYear: number | "昭和" | "平成";
birthYear = "昭和";
```

3.4 変数の巻き上げ

var、const、let では変数の巻き上げの挙動が多少異なります。var はスコープ内で宣言文の前では、変数はあるが初期化はされていない（undefined）になりますが、他の2つはコンパイルエラーになります。宣言前に触るのは行儀が良いとは言えないため、const の挙動の方が適切でしょう。

リスト7 変数の巻き上げ（変数の存在するスコープの宣言行前の挙動）

```
// 旧: var は undefined になる
function oldFunction() {
  console.log(`巻き上げのテスト ${v}`);
  var v = "小公女";
  // undefinedが入っている変数がある扱いになり、エラーならず
}
oldFunction();

// 新: let/const
function letFunction() {
  console.log(`巻き上げのテスト ${v}`);
  let v = "小公女";
  // 宣言より前ではエラー
  // error TS2448: Block-scoped variable 'v' used before its declaration.
}
letFunction();
```

3.5 変数のスコープ

以前は{、}は制御構文のためのブロックでしかなく、var 変数は宣言された function のどこからでもアクセスできました。let、const で宣言した変数のスコープは宣言されたブロック（if、for は条件式部分も含む）の中に限定されます。スコープが狭くなると、同時に把握すべき状態が減るため、コードが理解しやすくなります。

```
// 古いコード
for (var i = 0; i < 10; i++) {
  // do something
}
console.log(i); // -> 10

// 新しいコード
for (let i = 0; i < 10; i++) {
  // do something
}
console.log(i);
// error TS2304: Cannot find name 'i'.
```

なお、スコープはかならずしも制御構文である必要はなく、{、}だけを使うこともできます。

```
function code() {
  {
    //この変数はこの中でのみ有効
    const store = "小売店";
  }
}
```

(次のページに続く)

(前のページからの続き)

```
}
```

3.6 まとめ

本章では、TypeScript の入り口となる変数について紹介しました。昔の JavaScript とはやや趣向が変わっているところもありますが、新しい `let`、`const` を使うことで、変数のスコープをせばめ、理解しやすいコードになります。

第 4 章

プリミティブ型

プログラムの解説にはよく、リテラルという言葉がでできます。リテラルというのは、専用の文法を持ち、ソースコード中に直接記述できるデータのことです。TypeScript には何種類かあります。

- boolean 型
- number 型
- string 型
- 配列
- オブジェクト
- 関数
- undefined
- null

このうち、それ以上分解できないシンプルなデータを「プリミティブ型」と呼びます。ここでは、よく出てくるプリミティブ型を紹介します。

配列とオブジェクトは次章の[複合型](#)で、関数は[関数](#)で紹介します。

4.1 boolean リテラル

boolean 型は true/false の 2 つの真偽値を取るデータ型です。if 文、while ループなどの制御構文や、三項演算子などを使ってプログラムの挙動をコントロールするために大切な型です。

```
// 値を表示
console.log(true);
console.log(false);
```

(次のページに続く)

```
// 変数に代入。変数の型名は boolean
const flag: boolean = true;

// 他のデータ型への変換
console.log(flag.toString()); // 'true' / 'false' になる
console.log(String(flag));    // こちらでも変換可能
console.log(Number(flag));    // 1, 0 になる

// 他のデータ型を true/false に変換
const notEmpty = Boolean("test string"); // 変換ルールは後述
const flag = flagStr === 'true';         // 'true' の文字を true にするなら
const str = "not empty string";          // true/false 反転するが演算子一つで変換可能
const isEmpty = !str;                    // 反転すると !Boolean() と同じ
const notEmpty = !!str;                   // もう 1 つ使うと反転せずに boolean 型に
```

TypeScript では、数字のゼロ（負も含む）、空文字列、`null`、`undefined`、`NaN` を変換すると `false`、それ以外を変換すると `true` になります^{*1}。

4.1.1 ド・モルガンの法則

`if` 文の条件式が複雑なときに、それを簡単にするのにごくたまに役立つのがド・モルガンの法則です。次のような法則で NOT と AND と OR の組みを変換できます。

^{*1} この真偽値への変換ルールは言語によって異なります。例えば、Python では空の配列や辞書も偽 (`False`) になります。Ruby の場合は数字のゼロや空文字列も真 (`true`) になります。

リスト 1 ド・モルガンの法則

```
!(P || Q) == !P && !Q
!(P && Q) == !P || !Q
```

特に、右辺から左辺への変換がコードの可読性を高めることが多いと思います。NOT の集合同士の演算というのは普段の生活ではあまり出てきません。集合の AND/OR を考えてから逆転させる方が簡単にイメージできると思いますので、条件を書く時に、想定が漏れてロジックが正しく動作しない、ということが減るでしょう。また、より構成要素が多い論理式のときに、式を整理するのにも使えます。

4.2 数値型

TypeScript には組み込みで 2 種類の数値型があります。ほとんどの場合は `number` だけで済むでしょう。

4.2.1 `number`

TypeScript（というか、その下で動作している JavaScript）は 64 ビットの浮動小数点数で扱います。これはどの CPU を使っても基本的に同じ精度を持ちます^{*2}。整数をロスなく格納できるのは 53 ビット (-1) までなので、± 約 9007 兆までの整数を扱えます。それ以上の数値を入れると、末尾が誤差としてカットされたりして、整数を期待して扱うと問題が生じる可能性があります。正確な上限と下限は `Number.MAX_SAFE_INTEGER`、`Number.MIN_SAFE_INTEGER` という定数で見ることができます。また、`Number.isSafeInteger(数値)` という関数で、その範囲内に収まっているかどうかを確認できます。

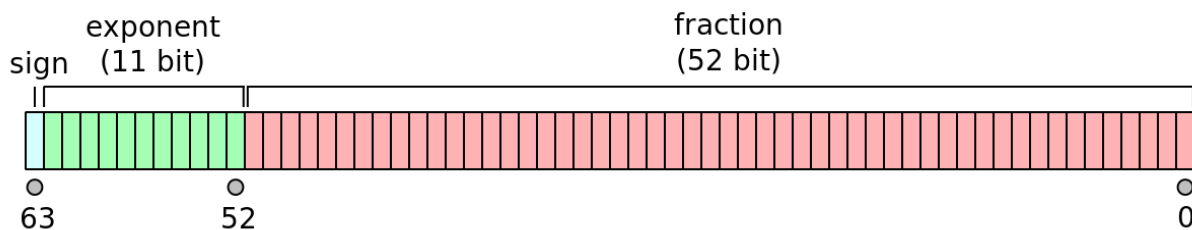


図 1 The memory format of an IEEE 754 double floating point value. by Codekaizen (CC 4.0 BY-SA)

```
// 値を表示
console.log(10.5);
console.log(128);
console.log(0b11); // 0b から始まると 2 進数
console.log(0o777); // 0o から始まると 8 進数
console.log(0xf7); // 0x から始まると 16 進数
```

(次のページに続く)

^{*2} IEEE 754 という規格で決まっています。

(前のページからの続き)

```
// 変数に代入。変数の型名は number
const year: number = 2019;

// 他のデータ型への変換
console.log(year.toString()); // '2019' になる
console.log(year.toString(2)); // toString の引数で 2 進数-36 進数にできる
console.log(Boolean(year)); // 0 以外は true

// 他のデータ型を数値に変換
console.log(parseInt("010")); // →10 文字列は parseInt で 10 進数/16 進数変換
console.log(parseInt("010", 8)); // →8 2 つめの数値で何進数として処理するか決められる
console.log(Number(true)); // boolean 型は Number 関数で 0/1 になる
```

変換の処理は、方法によって結果が変わります。10 進数を期待するものは radix 無しの `parseInt()` で使っておけば間違いありません。

表 1 文字列から数値の変換

	Number (文字列)	parseInt (文字列)	parseInt (文字列, radix)	リテラル
"10"	10	10	radix によって変化	10
"010"	10	10	radix によって変化	8 (8 進数)
"0x10"	16	16	radix が 16 以外は 0	8 (8 進数)
"0o10"	8	0	0	8 (8 進数)
"0b10"	2	0	0	2 (8 進数)

なお、リテラルの 8 進数ですが、ESLint の推奨設定を行うと `no-octal` というオプションが有効になります。このフラグが有効だと、8 進数を使用すると警告になります。

注釈: IE8 以前及びその時代のブラウザは、`parseInt()` に 0 が先頭の文字列を渡すと 8 進数になっているため、かならず radix を省略せずに 10 を設定しろ、というのが以前言われていました。その世代のブラウザは現在市場に出回っていないため、10 は省略しても問題ありません。

また、8 進数リテラルは以前の JavaScript は 0777 のように、ゼロ始まりのものも使えましたが、現在はこちらの記法は ES5 以上で非推奨となっており、TypeScript ではエラーになります。

4.2.2 数値型の使い分け

TypeScript には組み込みで 2 種類の数値型があります。2 つの型を混ぜて計算することはできません。

- `number`: 64 ビットの浮動小数点数
- `bigint`: 桁数制限のない整数（`10n` のように、後ろに `n` をつける）

`number` 型は多くのケースではベストな選択になります。特に浮動小数点数を使うのであればこちらしかありません。それ以外に、 $\pm 2^{53}-1$ までであれば整数として表現されるので情報が減ったりはしません。また、これらの範囲では一番高速に演算できます。

`bigint` 型は整数しか表現できませんが、桁数の制限はありません。ただし、現時点では `"target": "esnext"` と設定しないと使えません。使える場面はかなり限られるでしょう。本ドキュメントでは詳しく扱いません。

`number` は 2 進数で表した数値表現なので、`0.2 + 0.1` などのようなきれいに 2 進数で表現できない数値は誤差が出てしまいます。金額計算など、多少遅くても正確な計算が必要な場合は `decimal.js`^{*3} などの外部ライブラリを使います。

リスト 2 `number` の誤差

```
const a = 0.1;
const b = 0.2;
console.log(a + b);
// 0.30000000000000004
```

4.2.3 演算子

`+`、`-`、`*`、`/`、`%`（剰余）のよくある数値計算用の演算子が使えます。これ以外に、`**` というべき乗の演算子が ES2016 で追加されています^{*4}。

また、`number` は整数としても扱えますのでビット演算も可能です。ビット演算は 2 進数として表現した表を使って、計算するイメージを持ってもらえると良いでしょう。コンピュータの内部はビット単位での処理になるため、高速なロジックの実装で使われることがよくあります^{*5}。

ただし、ビット演算時には精度は 32 ビット整数にまで丸められてから行われるため、その点は要注意です。

^{*3} <https://www.npmjs.com/package/decimal.js>

^{*4} 以前は `Math.pow(x, y)` という関数を使っていました。

^{*5} ビット演算を多用する用途としては、遺伝子情報を高速に計算するのに使う FM-Index といったアルゴリズムの裏で使われる簡潔データ構造と呼ばれるデータ構造があります。

AND	<code>a & b</code>	2つの数値の対応するビットがともに 1 の場合に 1 を返します。
OR	<code>a b</code>	2つの数値の対応するビットのどちらかが 1 の場合に 1 を返します。
XOR	<code>a ^ b</code>	2つの数値の対応するビットのどちらか一方のみが 1 の場合に 1 を返します。
NOT	<code>~a</code>	ビットを反転させます
LSHIFT	<code>a << b</code>	a のビットを、b (32 未満の整数) 分だけ左にずらし、右から 0 をつめます。
RSHIFT	<code>a >> b</code>	a のビットを、b (32 未満の整数) 分だけ右にずらし、左から 0 をつめます。符号は維持されます。
0 埋め RSHIFT	<code>a >>> b</code>	a のビットを、b (32 未満の整数) 分だけ右にずらし、左から 0 をつめます。

なお、トリッキーな方法としては、次のビット演算を利用して、小数値を整数にする方法があります。なぜ整数になるかはぜひ考えて見てください^{*6}。これらの方法、とくに後者の方は小数値を整数にする最速の方法として知られているため、ちょっと込み入った計算ロジックのコードを読むと出てくるかもしれません。

- `~~` を先頭につける
- `| 0` を末尾につける

4.2.4 特殊な数値

数値計算の途中で、正常な数値として扱えない数値が出てくることがあります。業務システムでハンドリングすることはあまりないと思います。もし意図せず登場することがあればロジックの不具合の可能性が高いでしょう。

- 無限大: `Infinity`
- 数字ではない: `NaN (Not a Number)`

4.2.5 Math オブジェクト

TypeScript で数値計算を行う場合、`Math` オブジェクトの関数や定数を使います。

^{*6} ただし、ビット演算なので、本来扱えるよりもかなり小さい数字でしか正常に動作しません。

表 2 数値の最大値、最小値

関数	説明
<code>Math.max(x, y, ...)</code>	複数の値の中で最大の値を返す。配列内の数値の最大値を取得したい場合は <code>Math.max(...array)</code>
<code>Math.min(x, y, ...)</code>	複数の値の中で最小の値を返す。配列内の数値の最小値を取得したい場合は <code>Math.min(...array)</code>

乱数生成の関数もここに含まれます。0 から 1 未満の数値を返します。例えば 0-9 の整数が必要な場合は、10 倍して `Math.floor()` などを使うと良いでしょう。

表 3 乱数

関数	説明
<code>Math.random()</code>	0 以上 1 未満の疑似乱数を返す。暗号的乱数が必要な場合は <code>crypto.randomBytes()</code> を代わりに使う。

整数に変換する関数はたくさんあります。一見、似たような関数が複数あります。例えば、`Math.floor()` と `Math.trunc()` は似ていますが、負の値を入れた時に、前者は数値が低くなる方向に (-1.5 なら -2) 丸めますが、後者は 0 に近く方向に丸めるといった違いがあります。

表 4 整数変換

関数	説明
<code>Math.abs(x)</code>	x の絶対値を返す。
<code>Math.ceil(x)</code>	x 以上の最小の整数を返す。
<code>Math.floor(x)</code>	x 以下の最大の整数を返す。
<code>Math.fround(x)</code>	x に近似の単精度浮動小数点数を返す。ES2015 以上。
<code>Math.round(x)</code>	x を四捨五入して、近似の整数を返す。
<code>Math.sign(x)</code>	x が正なら 1、負なら -1、0 なら 0 を返す。ES2015 以上。
<code>Math.trunc(x)</code>	x の小数点以下を切り捨てた値を返す。ES2015 以上。

整数演算の補助関数もいくつかあります。ビット演算と一緒に使うことが多いと思われます。

表 5 32 ビット整数

関数	説明
<code>Math.clz32(x)</code>	x を 2 進数 32 ビット整数値で表した数の先頭の 0 の個数を返す。ES2015 以上。
<code>Math.imul(x, y)</code>	32 ビット同士の整数の乗算の結果を返す。超えた範囲は切り捨てられる。主にビット演算と一緒に使う。ES2015 以上。

平方根などに関する関数もあります。

表 6 ルート

関数・定数	説明
<code>Math.SQRT1_2</code>	1/2 の平方根の定数。
<code>Math.SQRT2</code>	2 の平方根の定数。
<code>Math.cbrt(x)</code>	x の立方根を返す。ES2015 以上。
<code>Math.hypot(x, y, ...)</code>	引数の数値の二乗和の平方根を返す。ES2015 以上。
<code>Math.sqrt(x)</code>	x の平方根を返す。

対数関係の関数です。

表 7 対数

関数・定数	説明
<code>Math.E</code>	自然対数の底（ネイピア数）を表す定数。
<code>Math.LN10</code>	10 の自然対数を表す定数。
<code>Math.LN2</code>	2 の自然対数を表す定数。
<code>Math.LOG10E</code>	10 を底とした e の対数を表す定数。
<code>Math.LOG2E</code>	2 を底とした e の対数を表す定数。
<code>Math.exp(x)</code>	<code>Math.E ** x</code> を返す。
<code>Math.expm1(x)</code>	<code>exp(x)</code> から 1 を引いた値を返す。ES2015 以上。
<code>Math.log(x)</code>	x の自然対数を返す。
<code>Math.log1p(x)</code>	1 + x の自然対数を返す。ES2015 以上。
<code>Math.log10(x)</code>	x の 10 を底とした対数を返す。ES2015 以上。
<code>Math.log2(x)</code>	x の 2 を底とした対数を返す。ES2015 以上。

最後は円周率や三角関数です。引数や返り値で角度を取るものはすべてラジアンですので、度（°）で数値を持っている場合は `* Math.PI / 180` でラジアンに変換してください。

4.3 string リテラル

string リテラルは文字列を表現します。シングルクォート、ダブルクォートでくくると表現できます。シングルクォートとダブルクォートは、途中で改行が挟まると「末尾がない」とエラーになってしまいますが、バッククォートでくくると、改行が中にあるても大丈夫なので、複数行あるテキストをそのまま表現できます。

```
// 値を表示
// シングルクォート、ダブルクォート、バッククォートでくくる
console.log('hello world');

// 変数に代入。変数の型名は string
```

(次のページに続く)

(前のページからの続き)

```

const name: string = "future";

// 複数行
// シングルクオート、ダブルクオートだとエラーになる
// error TS1002: Unterminated string literal.
const address = ' 東京都
品川区';
// バッククオートなら OK。ソースコード上の改行はデータ上も改行となる
const address = `東京都
品川区`;

// 他のデータ型への変換
console.log(parseInt('0100', 10)); // 100 になる
console.log(Boolean(name)); // 空文字列は false、それ以外は true になる

// 他のデータ型を string に変換
const year = 2019;
console.log((2019).toString(2)); // number は toString の引数で 2 進数-36 進数にできる
console.log((true).toString()); // boolean 型を 'true'/'false' の文字列に変換
console.log(String(false)); // こちらでも可

```

JavaScript は UTF-16 という文字コードを採用しています。Java と同じです。絵文字など、一部の文字列は 1 文字分のデータでは再現できずに、2 文字使って 1 文字を表現することがあります。これをサロゲートペアと呼びます。範囲アクセスなどで文字列の一部を抜おうとすると、絵文字の一部だけを拾ってしまう可能性がある点には要注意です。何かしらの文字列のロジックのテストをする場合には、絵文字も入れるようにすると良いでしょう。

4.3.1 文字列のメソッド

文字列には、その内部で持っている文字列を加工したり、一部を取り出したりするメソッドがいくつかあります。かなり古い JavaScript の紹介だと、HTML タグをつけるためのメソッドが紹介されていたりもします、TypeScript の型定義ファイルにも未だに存在はしますが、それらのメソッドは言語標準ではないためここでは説明しません。

4.3.2 文字コードの正規化

ユニコードには、同じ意味だけどコードポイントが異なり、字形が似ているけど少し異なる文字というものがあります。たとえば、全角のアルファベットの A と、半角アルファベットの A がこれにあたります。それらを統一してきれいにするのが正規化です。文字列の `normalize("NFKC")` というメソッド呼び出しをすると、これがすべてきれいになります。

```

> "A B C ^ ^ e f ^ ^ b d ^ ^ b l ^ ^ e f ^ ^ b d ^ ^ b 2 ^ ^ e f ^ ^ b d ^ ^ b 3 ^ ^ e f ^ ^ b d ^ ^ b 4 ^ ^ e f ^ ^ b d ^ ^ b 5 ^ ^ e 3 ^ ^ 8 d ^ ^ b b".
↳ normalize("NFKC")
'ABC アイウエオ平成'

```

正規化を行わないと、例えば、「6月6日議事録.md」という全角数字のファイル名を検索しようとして、「6月6日議事録」という検索ワードで検索しようとしたときにひっかからない、ということがおきます。検索対象と検索ワードの両方を正規化しておけば、このような表記のブレがなくなるため、ひっかかりそうでひっかからない、ということが減らせます。

正規形は次の4通りあります。Kがついているものがこのきれいにする方です。また、Dというのは、濁音の記号とベースの文字を分割するときの方法、Cは結合するときの方法になります。macOSの文字コードがNFKDなので、たまにmacOSのChromeでGoogle Spreadsheetを使うと、コピペだかなんだかのタイミングでこのカタカナの濁音が2文字に分割された文字列が挿入されることがあります。NFKCをつかっておけば問題はないでしょう。

- NFC
- NFD
- NFKC
- NFKD

正規化をこのルールに従って行くと、ユーザーに「全角数字で入力する」ことを強いるような、カッコ悪いUIをなくすことができます。ユーザーの入力はすべてバリデーションの手前で正規化すると良いでしょう。

ただし、長音、ハイフンとマイナス、漢数字の1、横罫線など、字形が似ているものの意味としても違うものはこの正規化でも歯が立たないので、別途対処が必要です。

4.3.3 文字列の結合

従来のJavaScriptは他の言語でいうprintf系の関数がなく、文字列を+で結合したり、配列に入れて.join()で結合したりしましたが、いまどきは文字列テンプレートリテラルがあるので、ちょっとした結合は簡単に扱えます。printfのような数値の変換などのフォーマットはなく、あくまでも文字列結合をスマートにやるためのものですが、複数行のテキストを表現できますし、プレースホルダ内には自由に式が書けます。もちろん、数が決まらない配列などは従来どおり.join()を使います。

```
// 古いコード
console.log("[Debug]:" + variable);

// 新しいコード
console.log(`[Debug]: ${variable}`);
```

このバッククォートを使う場合は、関数を前置することで、文字列を加工することができます。国際化でメッセージを置き換える場面などで利用されます。

4.3.4 文字列の事前処理

テンプレートリテラルに関数を指定すると（タグ付きテンプレートリテラル）、文字列を加工する処理を挟めます。よく使われるケースは翻訳などでしょう。テンプレートリテラルの前に置かれた関数は、最初に文字列の配列がきて、その後はプレースホルダの数だけ引数が付く構造になっています。文字列の配列は、プレースホルダに挟まれた部分のテキストになります。自作する機会は多くないかもしれませんが、コード理解のために覚えておいて損はないでしょう。

リスト 3 タグ付きテンプレートリテラル

```
function i18n(texts, ...placeholders) {
  // texts = ['小動物は', 'が好きです']
  // placeholders = ['小旅行']
  return // 翻訳処理
}

const hobby = "小旅行"
console.log(i18n`小動物は${hobby}が好きです`);
```

4.4 undefined と null

JavaScript/TypeScript では、undefined と null があります。他の言語では null（もしくは None、nil と呼ぶことも）だけの場合がほとんどですが、JavaScript/TypeScript では 2 種類登場します。

このうち、undefined は未定義やまだ値が代入されていない変数を参照したり、オブジェクトの未定義の属性に触ると帰ってくる値です。TypeScript はクラスなどで型定義を行い、コーディングがしやすくなるとよく宣伝されますが、「undefined に遭遇するとわかっているコードを事前にチェックしてくれる」ということがその本質だと思われます。

```
let favoriteGame: string; // まだ代入していないので undefined;
console.log(favoriteGame);
```

このコードは、tsconfig.json で strict: true（もしくは strictNullChecks: true）の場合にはコンパイルエラーになります。

JavaScript はメソッドや関数呼び出し時に数が合わなくてもエラーにはならず、指定されなかった引数には undefined が入っていました。TypeScript では数が合わないエラーになりますが、? を変数名の最後に付与すると、省略可能になります。

```
function print(name: string, age?: number) {
  console.log(`name: ${name}, age: ${age || 'empty'}`);
}
```

意図せずうっかりな「未定義」が undefined であれば、意図をもって「これは無効な値だ」と設定するのが

null です。ただし、Java と違って、気軽に null を入れることはできません。変数の章でも紹介した、「A もしくは B のデータが格納できる」という合併型（Union Type）の型宣言ができるので、これをつかって null を代入します。TypeScript では「これは無効な値をとる可能性がありますよ」というのは意識して許可してあげなければなりません。

```
// string が null を入れられるという宣言をして null を入れる
let favoriteGame: string | null = null;
```

undefined と null は別のもので、コンパイラオプションで `compilerOptions.strict: true` もしくは、`compilerOptions.strictNullChecks: true` の場合は、null 型の変数に undefined を入れようとしたり、その逆をするとエラーになります。これらのオプションを両方とも false にすれば、エラーにはなくなりますが、副作用が大きいので、これらのオプションは有効にして、普段から正しくコードを書く方が健全です。

リスト 4 null と undefined は別物

```
const a: string | null = undefined;
// error TS2322: Type 'undefined' is not assignable to type 'string | null'.

const b: string | undefined = null;
// error TS2322: Type 'null' is not assignable to type 'string | undefined'.
```

4.5 まとめ

TypeScript（と JavaScript）で登場する、プリミティブ型を紹介してきました。これらはプログラムを構成する上でのネジやクギとなるデータです。

第 5 章

複合型

他のプリミティブ型、もしくは複合型自身を内部に含み、大きなデータを定義できるデータ型を「複合型」と呼びます。配列、オブジェクトなどがこれにあたります。クラスを定義して作るインスタンスも複合型ですが、リテラルで定義できる配列、およびオブジェクトをここでは取り上げます。

5.1 配列

配列は TypeScript の中でかなり多用されるリテラルですが、スプレッド構文、分割代入などが加わり、また、数々のメソッドを駆使することで、関数型言語のような書き方もできます。配列は、次に紹介するオブジェクトと同様、リテラルで定義できる複合型の 1 つです。

```
// 変数に代入。型名を付けるときは配列に入れる要素の型名の後ろに [] を付与する
// 後ろの型が明確であれば型名は省略可能
const years: number[] = [2019, 2020, 2021];
const divs = ['tig', 'sig', 'saig', 'scig'];

// 配列に要素を追加。複数個も追加可能
years.push(2022);
years.push(2023, 2024);

// 要素から取り出し
const first = years[0];
```

5.1.1 タプル

Java などの配列は要素のすべての型は同じです。TypeScript では、配列の要素ごとに型が違う「タプル」というデータ型も定義できます。裏のデータ型は配列ですが、コンパイラが特殊なモードの配列として扱います。この場合違う型を入れようとするエラーになります。後述の `readonly` を使うことで読み込み専用のタプルを作ることができますが、デフォルトは変更可能です。

配列のインデックスごとに何を入れるか、名前をつけることはできないため、積極的に使うことはないでしょう。

```
const movie: [string, number] = ['Gozilla', 1954];
movie[0] = 2019;
// error TS2322: Type 'number' is not assignable to type 'string'.
```

固定長の配列を表現する手段としても利用できます。

```
const r = 10;
const t = Math.PI * 0.5;
const pos: [number, number] = [r * Math.cos(r), r * Math.sin(r)];
// Tuple type '[number, number]' of length '2' has no element at index '2'.
```

`[string, ...string[]]` と書けば、1 つは必ず要素があり、2 つ以上の要素が格納できるタプル、というのでも表現できますが、「これよりも少ない」は表現できません。

注釈: Python にもタプルがあります。これは要素の変更が不可能で、辞書のキーに使えたりするというので、配列とはかなり性質が異なっていて、言語にとっては重要な要素となっています。

一方、通常データ型として使うにはやはりインデックスアクセスしかできないため、可読性が劣り、インデックスとデータの種類の対応付けを人間が覚えるのはストレスがあるため、かなり初期から `namedtuple`（名前付きタプル）と呼ばれるクラスが提供されています。これは名前要素アクセスができる、タプルと可換なデータ構造です。

TypeScript の場合はリテラルで簡単にオブジェクトが作れますし、多くのタプルはオブジェクトで代替可能でしょう。

5.1.2 配列からのデータの取り出し

以前の JavaScript は、配列やオブジェクトの中身を変数に取り出すには一つずつ取り出すしかありませんでした。現在の JavaScript と TypeScript は、分割代入（=の左に配列を書く記法）を使って複数の要素をまとめて取り出すことができます。`slice()` を使わずに、新しい残余（Rest）構文（`...`）を使って、複数の要素をまとめて取り出すことができます。

残余構文は省略記号のようにピリオドを3つ書く構文で、あたかも複数の要素がそこにあるかのように振る舞いま

す。残余構文は取り出し以外にも、配列やオブジェクトの加工、関数呼び出しの引数リストに対しても使える強力な構文です。ここでは、2 つめ以降のすべての要素を `other` に格納しています。

リスト 1 配列の要素の取り出し

```
const smalls = [
  "小動物",
  "小型車",
  "小論文"
];
// 旧: 一個ずつ取り出す
var smallCar = smalls[1];
var smallAnimal = smalls[0];
// 旧: 2 番目以降の要素の取り出し
var other = smalls.slice(1);

// 新: まとめて取り出し
const [smallAnimal, smallCar, essay] = smalls;
// 新: 2 番目以降の要素の取り出し
const [, ...other] = smalls;
```

5.1.3 配列の要素の存在チェック

以前は、要素のインデックス値を見て判断していましたが、配列に要素が入っているかどうかを `boolean` で返す `includes()` メソッドが入ったので、積極的にこれを使っていきましょう。

リスト 2 要素の存在チェック

```
const places = ["小岩駅", "小浜市", "小倉駅"];

// 旧: indexOf を利用
if (places.indexOf("小淵沢") !== -1) {
  // 見つかった!
}

// 新: includes を利用
if (places.includes("小淵沢")) {
  // 見つかった!
}
```

5.1.4 配列の加工

配列の加工は、他言語の習熟者が JavaScript を学ぶときにつまづくポイントでした。splice() という要素の削除と追加を一度に行う謎のメソッドを使ってパズルのように配列を加工していました。配列のメソッドによっては、配列そのものを変更したり、新しい配列を返したりが統一されていないのも難解さを増やしているポイントです。スプレッド構文を使うと標準文法の範囲内でこのような加工ができます。さきほどのスプレッド構文は左辺用でしたが、こちらは右辺で配列の中身を展開します。

近年の JavaScript では関数型言語のテクニックを借りてきてバグの少ないコードにしよう、という動きがあります。その 1 つが、配列やオブジェクトを加工していくのではなく、値が変更されたコピーを別に作って、最後にリプレースするという方法です。splice() は対象の配列を変更してしましますが、スプレッド構文を使うと、この方針に沿ったコーディングがしやすくなります。配列のコピーも簡単にできます。

リスト 3 配列の加工

```
const smalls = [
  "小動物",
  "小型車",
  "小論文"
];

const others = [
  "小市民",
  "小田急"
];

// 旧: 3 番目の要素を削除して、1 つの要素を追加しつつ、他の配列と結合
smalls.splice(2, 1, "小心者");
// [ '小動物', '小型車', '小心者' ]
var newSmalls = smalls.concat(others);
// [ '小動物', '小型車', '小心者', '小市民', '小田急' ]
```

(次のページに続く)

(前のページからの続き)

```
// 新: スプレッド構文で同じ操作をする
// 先頭要素の削除の場合、分割代入を使えば slice() も消せます
const newSmalls = [...smalls.slice(0, 2), "小心者", ...others]
// [ '小動物', '小型車', '小心者', '小市民', '小田急' ]

// 旧: 配列のコピー
var copy = Array.from(smalls);

// 新: スプレッド構文で配列のコピー
const copy = [...smalls];
```

5.1.5 配列のソート

配列は `sort()` メソッドを使います。これはインプレースで、その配列を変更します。ソートをそのまま実行すると、中の要素をすべて文字列化した上で、辞書順でソートします。

リスト 4 デフォルトでは文字列としてソートする

```
const numbers = [30, 1, 200];

numbers.sort();
// 1, 200, 30
```

数値が入っている場合に、期待と異なる動作をします。比較関数を引数に渡し、0 より小さい数値（左辺を左側に）、0（等価）、0 より大きい数値（左辺を右側に移動）を返すことで要素の並び替えのルールを設定できます。オブジェクトの場合はどのキーを使うかなども比較関数で吸収します。左辺が小さい時に負の数を返せば昇順に、逆を返せば降順になります。

リスト 5 ソート関数を渡す

```
const numbers = [30, 1, 200];
numbers.sort((a, b) => a - b);
// 1, 30, 200

const stations = [
  {name: "池袋", users: 558623},
  {name: "新宿", users: 775386},
  {name: "渋谷", users: 366128},
  {name: "東京", users: 462589}
];
// 駅の利用者数でソート
stations.sort((a, b) => a.users - b.users);
```

複数の条件でソートしたい場合は、if 文を重ねて書いていってもいいのですが、同値条件が抜けたりしがちなので、いったん全て -1, 0, 1 にしておいて、足し合わせて総合スコアを計算する方がミスが減りますし、条件の入れ替

えはしやすいでしょう。

リスト 6 複合条件でソート

```
const stations = [
  {name: "大手町", lines: 5, yomi: "おおてまち"},
  {name: "飯田橋", lines: 7, yomi: "いいだばし"},
  {name: "永田町", lines: 5, yomi: "ながたちょう"},
];

function cmpNum(a: number, b: number) {
  return (a < b) ? -1 : (a === b) ? 0 : 1;
}

function cmpStr(a: string, b: string) {
  return (a < b) ? -1 : (a === b) ? 0 : 1;
}

// 乗入れ本数 → 読みでソート
stations.sort((a, b) => {
  const lineScore = cmpNum(a.lines, b.lines);
  const yomiScore = cmpStr(a.yomi, b.yomi);
  // わかりやすく 10 倍しているが、2 倍でも OK
  return lineScore * 10 + yomiScore;
});
```

非破壊のソートはないので、元の配列を変更せずにソートした結果だけを得たい場合は、前節のスプレッド構文を組み合わせで行います。

リスト 7 非破壊ソート

```
// 駅の利用者数でソート
const sorted = [...stations].sort((a, b) => a.users - b.users);
```

5.1.6 ループは `for ... of` を使う

ループの書き方は大きくわけて 3 通りあります。C 言語由来のループは昔からあるものですがループ変数が必要です。forEach() はその後 ES5 で追加されましたが、その後は言語仕様のアップデートとともに for ... of 構文が追加されました。この構文は Array, Set, Map, String などの繰り返し可能 (iterable) オブジェクトに対してループします。配列の場合で、インデックス値が欲しい場合は、entries() メソッドを使います。関数型主義的なスタイルで統一するために、for ... of を禁止して forEach() のみを使うというコーディング標準を規定している会社もあります (Airbnb)。

```
var iterable = ["小水井", "小淵沢", "小矢部"];

// 旧: C 言語由来のループ
for (var i = 0; i < iterable.length; i++) {
  var value = iterable[i];
}
```

(次のページに続く)

(前のページからの続き)

```

    console.log(value);
}

// 中: forEach() ループ
iterable.forEach(value => {
    console.log(value);
});

// 新: for of ループで配列のインデックスが欲しい
for (const [i, value] of iterable.entries()) {
    console.log(i, value);
}
// 要素のみ欲しいときは for (const value of iterable)

```

注釈: この `entries()` メソッドは、出力ターゲットを ES2015 以上にしないと動作しません。次のようなエラーがでます。

```
// error TS2339: Property 'entries' does not exist on type 'string[]'.
```

Polyfill を使うことで対処もできますが、Polyfill を使わない対処方法としては、`forEach()` を使う（2 つめの引数がインデックス）、旧来のループを使うしかありません。

速度の面で言えば、旧来の `for` ループが最速です。`for ... of` や `forEach()` は、ループ 1 周ごとに関数呼び出しが挟まるため、実行コストが多少上乘せされます。といっても、ゲームの座標計算で 1 フレームごとに数万要素のループを回さなければならない、といったケース以外ではほぼ気にする必要はないでしょう。

`forEach()`、`map()` などのメソッドは関数型プログラミングの章でも紹介します。

5.1.7 iterable とイテレータ

前節の最後に `entries()` メソッドが出てきました。これは、一度のループごとに、インデックスと値のタプルを返すイテレータを返します。配列のループのときに、インデックスと値を一緒に返すときにこのイテレータが登場しています。

```

const a = ["a", "b", "c"];
const b = [[0, "a"], [1, "b"], [2, "c"]];

// この 2 つの結果は同じ
for (const [i, v] of a.entries()) { console.log(i, v); }
for (const [i, v] of b) { console.log(i, v); }

```

この `entries()` は何者なんでしょうか？正解は、`next()` というメソッドを持つイテレータと呼ばれるオブ

ジェクトを返すメソッドです。この `next()` は、配列の要素と、終了したかどうかの `boolean` 値を返します。イテレータ（厳密には外部イテレータと呼ばれる）は `Java` や `Python`、`C++` ではおなじみのものです。

上記の `b` のように全部の要素を持つ二重配列を作ってしまうとこのようなイテレータというものは必要ありませんが、その場合、要素数が多くなればなるほど、コピーに時間がかかってループが回る前の準備が遅くなる、という欠点を抱えることになります。そのため、このイテレータという要素を返すオブジェクトを使い、全コピーを防いでいます。

オブジェクトにループの要素を取り出すメソッド (`@@iterator`) があるオブジェクトは `iterable` なオブジェクトです。繰り返し処理に対する約束事なので「`iterable` プロトコル」と呼ばれます。このメソッドはイテレータを返します。配列は、`@@iterator` 以外にも、`keys()`、`values()`、`entries()` と、イテレータを返すメソッドが合計 4 つあります。

`for...of` ループなどは、このプロトコルにしたがってループを行います。これ以外にも、分割代入や、スプレッド構文など、本特集で紹介した機能がこの `iterable` プロトコルを土台に提供されています。

`Array`、`Set`、`Map`、`String` などのオブジェクトがこのプロトコルを提供していますが、将来的に出てくるデータ構造もこのプロトコルをサポートするでしょう。また、自作することもできます。

イテレータはループするときには問題ありませんが、任意の位置の要素へのアクセスなどは不便です。イテレータから配列に変換したい場合は `Array.from()` メソッドか、スプレッド構文が使えます。

```
// こうする
const names = Array.from(iterable);

// これもできる
const names = [...iterable];
```

注釈: イテレータは `ES2015` 以降にしか存在しないため、スプレッド構文を使ってイテレータを配列に変換するのは、出力ターゲットが `ES2015` 以上でなければなりません。

```
const names = [...iterable];
```

5.1.8 読み込み専用の配列

TypeScript の「`const`」は変数の再代入をさせない、というガードにはなりますが、`C++` のように、「変更不可」にはできません。TypeScript にはこれには別のキーワード、`readonly` が提供されています。型の定義の前に `readonly` を付与するか、リテラルの後ろに `as const` をつけると読み込み専用になります。

```
// 型につける場合は readonly
const a: readonly number[] = [1, 2, 3];
// 値やリテラルに付ける場合は as const
```

(次のページに続く)

(前のページからの続き)

```
const b = [1, 2, 3] as const;
a[0] = 1;
// Index signature in type 'readonly number[]' only permits reading.
```

読み込み専用の配列は普通の変更可能な配列よりは厳しい制約となります。変更可能な配列は、readonly な配列の変数や引数には渡すことができます。逆に読み込み専用の配列を変更可能な配列の変数に格納したり関数の引数に渡したりしようとするとエラーになります。

```
const readonlyArray: readonly number[] = [1, 2, 3];
const mutableArray: number[] = [1, 2, 3];

function acceptReadOnlyArray(a: readonly number[]) {
}

function acceptMutableArray(a: number[]) {
}

// OK
const readonlyVar: readonly number[] = mutableArray;

// NG
const mutableVar: number[] = readonlyArray;
// The type 'readonly number[]' is 'readonly' and cannot be assigned to the mutable_
↪type 'number[]'.

// OK
acceptReadOnlyArray(mutableArray);

// NG
acceptMutableArray(readonlyArray);
// Argument of type 'readonly number[]' is not assignable to parameter of type
↪'number[]'.
// The type 'readonly number[]' is 'readonly' and cannot be assigned to the mutable_
↪type 'number[]'.
```

内部的には同じ配列ではありますので、型アサーションで readonly なしのものにキャストすれば格納したり呼び出し時に渡したりは可能です。しかし、C/C++ ではいわゆる「const 外し」はプログラムの安全性を脅かす邪悪な行為として忌み嫌われます。C/C++ の場合は組み込み機器で、読み込みしかできないメモリ領域にデータがおかれることもあり、動作が未定義で不正な挙動がおきうる、という意味では TypeScript よりもはるかに危険な行為ではありますが、「不変だと思っていた」変数がいつの間にかに書き換わっていたりして、開発者を混乱させる点では同じです。

この readonly を無理やり外したりせずに自然と使うためには、上から下までコード全体で readonly を使うように徹底するか、あるいは、まったく使わないかの二者択一になります。利用しているライブラリが readonly を使っているかということ、使っていないことが多いので、外部ライブラリとの接点では必ず readonly 外しが必要になるかもしれません。ここはプロジェクト全体での意思統一が必要になる場面となります。

```
const mutableVar: number[] = readonlyArray as number[];
acceptMutableArray(readonlyArray as number[]);
```

5.1.9 TypeScript と配列

`for ... of` には速度のペナルティがあるということを紹介しました。しかし、TypeScript を使っている場合には少し恩恵があります。

TypeScript を使っていると、ES5 への出力の場合型情報を見て、Array 型の `for ... of` ループの場合、旧来の最速の `for` ループの JavaScript コードが生成されますので、速度上のペナルティがまったくない状態で、最新の構文が使えるメリットがあります。また、Chrome などの JavaScript エンジンの場合は、同一の型の要素だけを含む配列の場合、特別な最適化を行います。

TypeScript を使うと、型情報がついて実装が簡単になるだけでなく、速度のメリットもあります。

5.1.10 配列のようで配列でない、ちょっと配列なオブジェクト

TypeScript がメインターゲットとしてるブラウザ環境では、配列に似たオブジェクトがあります。HTML の DOM を操作したときに得られる、`HTMLCollection` と、`NodeList` です。前者は `document.forms` などフォームを取得してきたときにも得られます。どちらも `.length` で長さが取得でき、インデックスアクセスができるため、一見配列のようですが、配列よりもメソッドがかなり少なくなっています。`NodeList` は `forEach()` はありますが、`HTMLCollection` にはありません。`map()` や `some()` はどちらにもありません。

どちらもイテレータは利用できますので、次のようなコードは利用できます。

- `for ... of` ループ
- スプレッド構文
- `Array.from()` で配列に変換してから各種配列のメソッドを利用

5.2 オブジェクト

オブジェクトは、JavaScript のコアとなるデータですが、クラスなどを定義しないで、気軽にまとめたデータを扱うときに使います。配列は要素へのアクセス方法がインデックス（数値）でしたが、オブジェクトの場合は文字列です。キー名が変数などで使える文字だけで構成されている場合は、名前をそのまま記述できますが、空白文字やマイナスなどを含む場合にはダブルクォートやシングルクォートでくくります。また、キー名に変数を書く場合は `[]` でくくります。

リスト 8 オブジェクト

```
// 定義はキー、コロン (:)、値を書く。要素間は改行
const key = 'favorite drink';

const smallAnimal = {
  name: "小動物",
  favorite: "小籠包",
  'home town': "神奈川県警のいるところ",
  [key]: "ストロングゼロ"
};

// 参照は `.` + 名前、もしくは [名前]
console.log(smallAnimal.name); // 小動物
console.log(smallAnimal[key]); // ストロングゼロ
```

おおきなプログラムをきちんと書く場合には、次の章で紹介するクラスを使うべきですが、次のようなクラスを定義するまでもない場面で出てきます。

- Web サービスのリクエストやレスポンス
- 関数のオプションな引数
- 複数の情報を返す関数
- 複数の情報を返す非同期処理

5.2.1 JSON (JavaScript Object Notation)

オブジェクトがよく出てくる文脈は「JSON」です。JSON というのはデータ交換用フォーマットで、つまりは文字列です。プレーンテキストであり、書きやすく読みやすい (XML や SOAP と比べて) こともありますし、JavaScript でネイティブで扱えるため、API 通信で使われるデータフォーマットとしてはトップシェアを誇ります。

JSON をパースすると、オブジェクトと配列で階層構造になったデータができあがります。通信用のライブラリでは、パース済みの状態でレスポンスが帰ってきたりするため、正確ではないですが、このオブジェクト/配列も便宜上、JSON と呼ぶこともあります。

リスト 9 JSON とオブジェクト

```
// 最初の引数にオブジェクトや配列、文字列などを入れる
// 2 つめの引数はデータ変換をしたいときの変換関数 (ログ出力からパスワードをマスクしたいなど)
//   省略可能。通常は null
// 3 つめは配列やオブジェクトでインデントするときのインデント幅
//   省略可能。省略すると改行なしの 1 行で出力される
const json = JSON.stringify(smallAnimal, null, 2);
```

(次のページに続く)

(前のページからの続き)

```
// これは複製されて出てくるので、元の smallAnimal とは別物
const smallAnimal2 = JSON.parse(json);
```

JSON は JavaScript/TypeScript のオブジェクト定義よりもルールが厳密です。たとえば、キーは必ずダブルクォートでくくられなければなりませんし、配列やオブジェクトの末尾に不要なカンマがあるとエラーになります。その場合は `JSON.parse()` の中で `SyntaxError` 例外が発生します。特に、JSON を便利だからとマスターデータとして使っていて、非プログラマーの人に、編集してもらったりしたときによく発生します。あとは、JSON レスポンスを期待しているウェブサービスの時に、サーバー側でエラーが発生して、`Forbidden` という文字列が帰ってきた場合（403 エラー時のボディ）にも発生します。

リスト 10 JSON パースのエラー

```
SyntaxError: Unexpected token n in JSON at position 1
```

5.2.2 オブジェクトからのデータの取り出し

オブジェクトの場合も配列同様、分割代入でまとめて取り出せます。また、要素がなかったときにデフォルト値を設定したり、指定された要素以外のオブジェクトを抜き出すことが可能です。注意点としては、まとめて取り出す場合の変数名は、必ずオブジェクトのキー名になります。関数の返回值や、後述の `Promise` では、この記法のおかげで気軽に複数の情報をまとめて返せます。

リスト 11 オブジェクトの要素の取り出し

```
const smallAnimal = {
  name: "小動物",
  favorite: "小籠包"
};

// 旧: 一個ずつ取り出す
var name = smallAnimal.name;
var favorite = smallAnimal.favorite;
// 旧: 存在しない場合はデフォルト値を設定
var age = smallAnimal.age ? smallAnimal.age : 3;

// 新: まとめて取り出し。デフォルト値も設定可能
const {name, favorite, age=3} = smallAnimal;
// 新: name 以外の要素の取り出し
const {name, ...other} = smallAnimal;
```

ES2020 で追加された機能として、オプショナルチェイニングがあります。TypeScript でも 3.7 から導入されました。TypeScript では、変数の型として、文字列だけでなく、場合によっては無効な値として `null` や `undefined` が入る可能性がある、といったバリエーションを持たせることができます。型定義の話は**基本的な型付け**で触れるので、先行した説明になりますが、例えば次の定義は `smallAnimal` 自身がオブジェクト、もしくは `null` となりますし、`favorite` というメンバーも `undefined` になりえるという意味になります。

この場合、深い階層にアクセスする場合は、一つずつ、`null` や `undefined` になりえるところでチェックを行っていました。`&&` 演算子が、一つでも途中で `falsey` な値があると評価を止める、そうでなければ最後の値を返すという挙動を持っているため、それを活用したコーディングが行われていました。

オプショナルチェイニングは同じことを実現する演算子として`?.` が導入されました。途中で `nullish` (`null` か `undefined`) な値があると、式全体の評価結果が `undefined` になります。

```
const smallAnimal: {name: string, favorite?: string} | null = {
  name: "小動物",
  favorite: "小籠包"
};

// 旧: 一個ずつ確認してアクセスし、大文字の好物を取得
var favorite = smallAnimal && smallAnimal.favorite && smallAnimal.favorite.
  .toUpperCase()

// 新: 一個ずつ確認してアクセスし、大文字の好物を取得
const favorite = smallAnimal?.favorite?.toUpperCase()
```

5.2.3 オブジェクトの要素の加工

JavaScript ではオブジェクトがリテラルで作成できるデータ構造として気軽に利用されます。オブジェクトの加工（コピーや結合）も配列同様にスプレッド構文で簡単にできます。

```
const smallAnimal = {
  name: "小動物"
};

const attributes = {
  job: "小説家",
  nearStation: "小岩駅"
}

// 最古: オブジェクトをコピー
var copy = {};
for (var key1 in smallAnimal) {
  if (smallAnimal.hasOwnProperty(key1)) {
    copy[key1] = smallAnimal[key1];
  }
}

// 旧: Object.assign() を使ってコピー
const copy = Object.assign({}, smallAnimal);

// 新: スプレッド構文でコピー
const copy = {...smallAnimal};
```

(次のページに続く)

(前のページからの続き)

```
// 最古: オブジェクトをマージ
var merged = {};
for (var key1 in smallAnimal) {
    if (smallAnimal.hasOwnProperty(key1)) {
        merged[key1] = smallAnimal[key1];
    }
}
for (var key2 in attributes) {
    if (attributes.hasOwnProperty(key2)) {
        merged[key2] = attributes[key2];
    }
}

// 旧: Object.assign() を使ってオブジェクトをマージ
const merged = Object.assign({}, smallAnimal, attributes);

// 新: スプレッド構文でマージ
const merged = {...smallAnimal, ...attributes};
```

5.2.4 辞書用途はオブジェクトではなくて Map を使う

ES2015 では、単なる配列以外にも、Map/Set などが増えました。これらは子供のデータをフラットにたくさん入れられるデータ構造です。これも配列と同じ iterable ですので、同じ流儀でループできます。古のコードはオブジェクトを、他言語の辞書やハッシュのようになっていましたが、今時は Map を使います。他の言語のようにリテラルで簡単に初期化できないのは欠点ですが、キーと値を簡単に取り出してループできるほか、キーだけでループ (for (const key of map.keys()))、値だけでループ (for (const value of map.values())) も使えます。

辞書用途で見た場合の利点は、オブジェクトはキーの型に文字列しか入れることができませんが、Map や Set では number など扱えます。

オブジェクトは、データベースでいうところのレコード (1つのオブジェクトはいつも固定の名前がある) として使い、Map はキーが可変の連想配列で、値の型が常に一定というケースで使うと良いでしょう。

WeakMap や WeakSet という弱参照のキャッシュに使えるコレクションもありますし、ブラウザで使えるウェブアクセスの FetchAPI の Headers クラスも似た API を提供しています。これらのクラスに慣れておくと、コレクションを扱うコードが自在に扱えるようになるでしょう。

```
// 旧: オブジェクトを辞書代わりに
var map = {
    "五反田": "約束の地",
    "戸越銀座": "TGSGNZ"
};
```

(次のページに続く)

(前のページからの続き)

```
for (var key in map) {
    if (map.hasOwnProperty(key)) {
        console.log(key + " : " + map[key]);
    }
}

// 新: Map を利用
// ``<キーの型、 値の型>`` で明示的に型を指定すると
// ``set()`` 時に型違いのデータを入れようとするとチェックできるし、
// ループなどで値を取り出しても型情報が維持されます
const map = new Map<string, string>([
    ["五反田", "約束の地"],
    ["戸越銀座", "TGSGNZ"]
]);

for (const [key, value] of map) {
    console.log(`${key} : ${value}`);
}
```

注釈: Map、Set は ES2015 以降に導入されたクラスであるため、出力ターゲットをこれよりも新しくするか、ライブラリに登録した上で Polyfill を使うしかありません。

5.2.5 TypeScript とオブジェクト

オブジェクトは、プロトタイプ指向という JavaScript の柔軟性をささえる重要な部品です。一方、TypeScript はなるべく静的に型をつけて行く事で、コンパイル時にさまざまなチェックが行えるようになり不具合を見つけることができます。オブジェクトの型の定義については[基本的な型付け](#)の章で紹介します。

型定義をすると、プロパティの名前のスペルミスであったり、違う型を入れてしまうことが減ります。エラーチェックのコードを実装する手間も減るでしょう。

5.2.6 読み込み専用のオブジェクト

配列は `readonly` や `as const` をつけて読み込み専用にできましたが、オブジェクトも同様のことができます。ただし、`readonly` キーワードではできず、型ユーティリティの `Readonly<>` を使います。これには、型を定義しておく必要があります。これ以外にも、フィールドごとに `readonly` を付与することも可能です。あるいは、型ではなく、値の最後に `as const` を付与します。前節でも触れましたが、これも詳しくは[基本的な型付け](#)の章で紹介します。

```
type User = {
  name: string;
  age: number;
};

const u: Readonly<User> = {name: "shibukawa", age: 39};

// こちらでも良い
const u = {name: "shibukawa", age: 39} as const;

// NG
u.age = 17;
// Cannot assign to 'age' because it is a read-only property.
```

5.3 まとめ

JavaScript の 2 大複合型の配列とオブジェクトを紹介しました。また、オブジェクトの関連のデータ構造として Map や Set も紹介しました。

Java と比べると、TypeScript で実装する場合、同じようなものを実装する場合にもクラス定義の数は減るでしょう。ちょっとしたデータを格納するデータ構造などは、これらの型を使って定義なしで使うことが多いからです。Java からやってくると、これらの型を乱用しているように見えて不安になるかもしれません。しかし、TypeScript を使えば、型推論やインラインでの明示的な型定義によって、これらの型でもきちんとしたチェックが行われるようになります。不安はあるかもしれませんが、安全にコーディングができます。

第 6 章

基本的な構文

TypeScript を扱ううえで登場する制御構文です。JavaScript と変わりませんし、Java や C++ とかともほぼ変わりません。すでに知っている方は飛ばしても問題ありません。

6.1 制御構文

6.1.1 if

一番基本的な条件分岐です。Java や C++ を使ったことがあれば一緒です。

- `if` (条件) ブロックです。
- `else if` (条件) を追加することで、最初のケースで外れた場合に追加で条件分岐させることができます。
- `else` をつけると、マッチしなかったケースで処理される節を追加できます。
- ブロックは `{ }` でくくってもいいですし、1 つしか文がないなら `{ }` を省略することもできます。

```
if (task === "休憩中") {  
  console.log("サーフィンに行く");  
} else if (task === "デスマ中") {  
  console.log("睡眠時間を確保する");  
} else {  
  console.log("出勤する");  
}
```

なお、昔、よくバグの原因となると有名だった、条件文の中で比較演算子ではなく、間違って代入を書いてしまうことでプログラムの動きがおかしくなってしまう問題ですが、ESLint の推奨設定で有効になる `no-cond-assign` という項目で検出できます。

6.1.2 switch

条件文の中の値と、case で設定されている値を === 演算子で前から順番に探索し、最初にマッチした節を実行します。一致した値がなく default 節があった場合にはそこが実行されます。

```
switch (task) {
  case "休憩中":
    console.log("サーフィンに行く");
    break;
  case "デスマ中":
    console.log("睡眠時間を確保する");
    break;
  default:
    console.log("出勤する");
}
```

case の条件が重複している case は ESLint の推奨設定でも有効になる no-duplicate-case オプションで検知できます。また、break を忘れると、次の case が実行されてしまいますが、こちらも ESLint の推奨設定で有効になる no-fallthrough オプションで検知できます。

6.1.3 for

一番使うループ構文です。4通りの書き方があります。

C 言語風のループ変数を使う書き方

フラグの数値をインクリメントしながらループする方式です。昔は var を変数宣言に使っていましたが、let が推奨です。let の変数は、この for の条件式とブロックの中だけで有効になります。

リスト 1 C 言語風のループ変数を使う方式

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

for..in

オブジェクトのプロパティを列挙するループです。プロトタイプまで探索しにいくため、想定外の値がループ変数に代入される可能性があります。そのため、hasOwnProperty() メソッドを呼んで、想定外の値が入らないようにブロックする書き方が一般的でした。今では使うことはないでしょう。次に紹介する for..of を使うべきです。

リスト 2 for in

```
for (let key in obj) {  
  if (obj.hasOwnProperty(key))  
    console.log(key, obj[key]);  
}
```

JavaScript 時代との違いは、`let` で定義された変数の範囲です。その `key` の値も含めて、条件文とボディの中以外から見えることはありません。

注釈: 配列のループに `for...in` を使うことも不可能ではありませんが、現在使われている各種ブラウザでは、通常の `for` ループと比べて 50 倍から 100 倍遅くなります。配列のループの手段として使うのはやめましょう。

for...of

`for...in` より新しい記法です。イテレータという各データ構造がループ用に持っている機能を使うため、想定外の値が入ることはありません。

リスト 3 for of

```
for (const value of array) {  
  console.log(value);  
}
```

`of` の右側には、イテレータプロトコルをサポートする、次のような要素が書けます。

- 配列、Map、Set、文字列

また、それ以外に、イテレータを返すメソッドや関数があり、これらの呼び出しを右辺に持ってくることもできます。

- `array.entries()` (配列のインデックスと値がセットで返ってくる)
- `Object.keys(obj)` (オブジェクトのキーが返ってくる)
- `Object.values(obj)` (オブジェクトの値が返ってくる)
- `Object.entries(obj)` (オブジェクトのキーと値が返ってくる)
- `map.keys()` (Map のキーが返ってくる)
- `map.values()` (Map の値が返ってくる)
- `map.entries()` (Map のキーと値が返ってくる)

キーと値の両方が帰ってくるメソッドは、分割代入を用いて変数に入れます。

リスト 4 for of

```
for (const [key, value] of Object.entries(obj)) {  
  console.log(key, value);  
}
```

なお上記に列挙したものの中では、`Object.keys()` が ES5 に入っています。他のものを使うときは、ターゲットバージョンを ES2015 以上にするか、ターゲットバージョンを低くする代わりに Polyfill を設定する必要があります。

このイテレータは、配列以外にも、配列のような複数の値を含むデータ構造（シーケンス）が共通で備えるインタフェースです。このインタフェースを実装することで、ユーザークラスでも `for..of` ループと一緒に使えるようになります。現在はそれほどではないですが、言語標準であるため、何かしらの最適化が行われる可能性があります。

for await of

ES2018 で導入されました。ループごとに非同期の待ち処理を入れます。これに対応するには、`asyncIterator` に対応した要素を条件文の右辺に持ってくる必要がありますが、現在サポートしているのは `ReadableStream` ぐらいしかありません。このクラスは、`fetch()` のレスポンスの `body` プロパティぐらいでしか見かけません。対応するクラスを自作することもできます。

```
for await (const body of response.body) {  
  console.log(body);  
}
```

並行して処理を投げる場合は、非同期の章で紹介するように `Promise.all()` を使い、すべてのリクエストはすべて待たずに投げってしまう方が効率的です。`for await of` は同期的な仕事でのみ利用されることを想定しています。

6.1.4 while、do .. while

条件にあっていて限り回り続けるループです。while はブロックに入る前にチェックが入る方式、do .. while はブロックの後でチェックをします。

以前は、無限ループを実現するために `while (true)` と書くこともありましたが、ESLint では推奨設定で設定される `no-constant-condition` オプションで禁止されます。

6.1.5 try .. catch

例外をキャッチする文法です。Java と違うのは、JavaScript は型を使って複数の catch 節を振り分けることができない、という点です。catch には 1 つだけ入れ条件文を書きます。非同期処理が多い JavaScript では、例外でうまくエラーを捕まえられることはまれでしたが、ES2017 で導入された async 関数は非同期処理の中のエラーを例外として投げるため、再びこの文法の利用価値が高まっています。例外に関しては特別に章を分けて説明します。

```
try {
  // 処理
  throw new Error("例外投げる")
} catch (e) {
  // ここに飛んでくる
  console.log(e);
} finally {
  // エラーがあってもなくてもここにはくる
}
```

6.2 式

基本的な演算子などは、他の言語と変わらないので省略します。他の言語ユーザーが困りそうなポイントは次の 2 つぐらいです。

- 比較の演算子: === と == がある（それぞれ否定は !== と != ）。前者は一致を厳密に見るが、後者は、文字列に変換してから比較する。なお、配列やオブジェクトで厳密な一致（===）は、インスタンスが同一かどうか、で判定されます。
- ** 演算子: x ** y で Math.pow(x, y) と同じ累乗計算を行う

いまどきのウェブフレームワークでコードを書く上で大事な式は 2 つあり、論理積（&&）と、三項演算子ですね。それぞれ、（条件） && 真の時の値、（条件） ? 真の時の値 : 偽の時の値 という、条件分岐を 1 行で書きます。

リスト 5 三項演算子（わかりやすくするためにカッコを入れましたが省略可能です）

```
const result = (day === "金曜日") ? "明日休みなので鳥貴族に行く" : "大人しく帰る";
```

とくに、React は 1 行の一筆書き（1 つの return 文の中で）で、仮想 DOM という巨大な JavaScript のオブジェクトを生成します。このときに条件分岐のコードとして役に立つのが三項演算子というわけです。

リスト 6 React 中の条件分岐

```
render() {  
  return (  
    <div>  
      { this.state.loggedIn ? <p>ようこそ</p> : <p>ログインが必要です</p> }  
    </div>  
  );  
}
```

参考までに、ループは配列の map メソッドを使うことが多いです。

リスト 7 React 中のループ

```
render() {  
  return (  
    <ul>  
      { this.state.users.map(user => {  
        <li>{user.name}</li>  
      }) }  
    </ul>  
  );  
}
```

6.3 まとめ

基本的な部分は他の言語、特に C++ や Java といった傾向の言語を使っている人からすればあまり大きな変化に感じないでしょう。

for ループだけはいくつか拡張がされたりしてましたので紹介しました。また、今時のウェブフレームワークで使う、1 行のコード内で使える条件分岐とループも紹介しました。

第 7 章

基本的な型付け

TypeScript は JavaScript に対して型をつけるという方向で仕様が作られています。JavaScript は動的言語の中でも、いろいろ制約がゆるく、無名関数とオブジェクトを使ってかなり柔軟なプログラミングの手法を提供してきました。そのため、オブジェクトに対して型をつける方法についても、他の Java などの静的型付け言語よりもかなり複雑な機能を持っています。

ただし、ここに説明されている機能を駆使して完璧な型付けを行う必要があるかという点、それは時と場合によります。たとえば、TypeScript を使ってライブラリを作る場合、それを利用するコードも TypeScript であれば型チェックでコンパイル時にチェックが行われます。しかし、利用する側が JavaScript の場合は、型によるチェックができません。エラーを見逃すことがあります。ユーザー数が多くなって、利用者が増えるかどうかで費用対効果を考えて、どこまで詳細に型づけを行うか決めれば良いでしょう。

なお、最初の変数の定義のところで、いくつか型についても紹介しました。それを少し思い出していただければ、と思います。

```
// 型は合併型 (Union Type) で複数列挙できる
let birthYear: number | string;

// 型には文字列や数値の値も設定できる
let favoriteFood: "北極" | "冷やし味噌";
```

7.1 一番手抜きな型付け: any

費用対効果を考えましょう、と言われても、意思決定の幅がわからないと、どこが良いのか決断はできません。最初に、一番費用が少ない方法を紹介します。それが any です。any と書けば、TypeScript のコンパイラは、その変数のチェックをすべて放棄します。

```
function someFunction(opts: any) {
  console.log(opts.debug); // debug があるかどうかチェックしないのでエラーにならない
}
```

ただし、これを使うと、TypeScript が提供する型チェックの恩恵は受けられません。any から型情報付きのデータにするためには後述の型ガードや型アサーションで変換しなければなりません。利用する箇所ですら毎回必要になります。TypeScript の型情報は伝搬するので、なるべく早めに、データが発生する場所で型情報を付ければ、変換が不要になります。そのため、よっぽどの理由がないかぎり any を使わない方がトータルの実装コストは大きく減ります。実際に TypeScript できちんと回っているプロジェクトの場合、ESLint で any を使っていたらエラーにすることになるでしょう。

any を積極的に使う場面は 2 つあります。

1 つは後述するユーザー定義の型ガードの引数です。これは「型がわからないデータの型を診断する」関数ですので、引数は any となります。

それ以外だと、外部からやってくるデータなどはコンパイル時には型情報がわかりません。標準ライブラリのブラウザのサーバーアクセス API の fetch のレスポンスの json() メソッドの戻り値は any となっています。そのため、fetch のレスポンスに関しては何かしらの変換処理が必要になります。ただし、このケースは any が利用されているだけでユーザーコードの中で any とタイプすることはありません。

消極的な利用方法としては、すでに JavaScript として動作していて実績があるコードを TypeScript にまずは持ってくる、というケースが考えられます。あとは、メインの引数ではなくて、挙動をコントロールするオプションの項目がかなり複雑で、型定義が複雑な場合などです。例えば、JSONSchema を受け取るような引数があったら、JSONSchema のすべての仕様を満たす型定義を記述するのはかなり時間を要します。将来やるにしても、まずはコンパイルだけは通したい、というときに使うと良いでしょう。

7.2 未知の型: unknown

unknown は any と似ています。unknown 型の変数にはどのようなデータもチェックなしに入れることができます。違うのは unknown の場合は、その変数を利用する場合には、型アサーションを使ってチェックを行わないとエラーになる点です。型アサーションについてはこの章の最後で扱います。

unknown はもう一箇所出てくる可能性のある場所があります。ジェネリクスを使ったクラスや関数のうち、自動で型推論で設定できなかったものは unknown となります。この型変数の unknown に関してはエラーチェックなどが行われることがなく、any のように振舞います。型推論で自動設定される予定の型変数が unknown になってしまったのであれば、コーディングのミスが発生したものと考えられます。

課題: 事例をつける

7.3 型に名前をつける

`type 名前` という構文を使って、型に名前をつけることができます。名前には、通常の変数や関数名として使える名前が使えます。ここで定義した型は、変数定義や、関数の引数などで使えます。

```
// 型は合併型で複数列挙できる
type BirthYear = number | string;

// 型には値も設定できる
type FoodMenu = "北極" | "冷やし味噌";

// 変数や関数の引数で使える
const birthday: BirthYear = "平成";

function orderFood(food: FoodMenu) {
}
```

使い回しをしないのであれば型名の代わりに、すべての箇所に定義を書いていてもエラーチェックの結果は変わりません。また、TypeScript は型名ではなく、型の内容で比較してチェックを行うため、別名の型でも、片方は型で書いて、片方は直接書き下したケースでも問題なくチェックされます。

```
type FoodMenu = "北極" | "冷やし味噌";
const myOrder: FoodMenu = "北極";

function orderFood(food: "北極" | "冷やし味噌") {
  console.log(food);
}

orderFood(myOrder);
```

7.4 関数のレスポンスや引数で使うオブジェクトの定義

`type` はオブジェクトが持つべき属性の定義にも使えます。属性には型をつけることができます。

```
type Person = {
  name: string;
  favoriteBank: string;
  favoriteGyudon: string;
}

// 変数定義時にインタフェースを指定
const person: Person = {
  name: "Yoichi",
  favoriteBank: "Mizuho",
  favoriteGyudon: "Matsuya"
}
```

(次のページに続く)

(前のページからの続き)

};

このように型定義をしておくと、関数の引数などでもエラーチェックが行われ、関数の呼び出し前後での不具合発生を抑えることができます。

```
// 関数の引数が Person 型の場合
registerPerson({
  name: "Yoichi",
  favoriteBank: "Mizuho",
  favoriteGyudon: "Matsuya"
});

// レスポンスが Person 型の場合
const { name, favoriteBank } = getPerson();
```

もし、必須項目の favoriteBank がなければ代入する場所でエラーが発生します。また、リテラルで書く場合には、不要な項目があってもエラーになります。

```
const person: Person = {
  name: "Yoichi"
};
// error TS2741: Property 'favoriteBank' is missing in
//   type '{ name: string; }' but required in type 'Person'.
```

JavaScript では、多彩な機能を持つ関数を定義する場合に、オプションとなるパラメータをオブジェクトで渡す、という関数が数多くありました。ちょっとタイプミスしてしまっただけで期待通りの結果を返さないでしばらく悩む、といったことがよくありました。TypeScript で型の定義をすると、このようなトラブルを未然に防ぐことができます。

7.5 オブジェクトの属性の修飾: オプション、読み込み専用

```
type Person = {
  name: string;
  readonly favoriteBank: string;
  favoriteGyudon?: string;
}
```

名前の前に readonly を付与すると、属性の値が読み込み専用になり、書き込もうとするとエラーになります。また、名前の後ろに ? をつけることで、省略可能な属性であることを示すことができます。

これらにより、データの有無を柔軟にしたり、意図せぬ変更を抑制してバグを減らしたりする効果があります。

型ユーティリティを使えば、一度定義した型のすべての属性に一括して ? をつけたり、readonly をつけることもできます。

```

type Person = {
  name: string;
  favorite: string;
};

// Partialをつけたので、全ての要素を設定しなくてもよい
const wzz: Partial<Person> = {name: "wzz"};

// Readonlyになったので要素の書き換えが不可に
const bow: Readonly<Person> = {name: "bow", favorite: "よなよなエール"};
bow.favorite = "水曜日の猫";
// Cannot assign to 'favorite' because it is a read-only property.

```

7.6 属性名が可変のオブジェクトを扱う

これまで説明してきたのは、各キーの名前があらかじめ分かっている、他の言語で言うところの構造体のようなオブジェクトです。しかし、このオブジェクトは辞書のようにも使われます。今時であれば Map 型を使う方がイテレータなども使えますし、キーの型も自由に選べて良いのですが、例えば、サーバー API のレスポンスの JSON などのようなところでは、どうしてもオブジェクトが登場します。

その時は、`{ [key: キーの型]: 値の型 }` と書くことで、辞書のように扱われるオブジェクトの宣言ができます。なお、key の部分はなんでもよく、a でも b でもエラーにはなりませんが、key としておいた方がわかりやすいでしょう。

```

const postalCodes: { [key: string]: string } = {
  "602-0000": "京都市上京区",
  "602-0827": "京都市上京区相生町",
  "602-0828": "京都市上京区愛染寺町",
  "602-0054": "京都市上京区飛鳥井町",
};

```

なお、キーの型には string 以外に number など設定できます。その場合、上記の例だとエラーになりますが、`"6020000"`（ダブルクオートがある点に注意）とするとエラーがなくなります。一見数値が入っているように見えますが、JavaScript のオブジェクトのキーは文字列型ですので、`Object.keys()` とか `Object.entries()` で取り出すキーの型まで数字になるわけではなく、あくまでも文字列です。数値としても認識できる文字列を受け取る、という挙動になります。

7.7 A かつ B でなければならない

A | B という記法（合併型）を紹介しました。これは「A もしくは B」という意味です。コンピュータの論理式では「A かつ B」というのがありますよね？ TypeScript の型定義ではこれも表現できます。& の記号を使います。

リスト 1 型を合成する

```
type Twitter = {
  twitterId: string;
}

type Instagram = {
  instagramId: string;
}

const shibukawa: Twitter & Instagram = {
  twitterId: "@shibu_jp",
  instagramId: "shibukawa"
}
```

これは交差型（Intersection Type）と呼ばれ、両方のオブジェクトで定義した属性がすべて含まれないと、変数の代入のところでエラーになります。

もちろん、合成した型に名前をつけることもできます。

```
type PartyPeople = Twitter & Instagram;
```

7.8 タグ付き合併型: パラメータの値によって必要な属性が変わる柔軟な型定義を行う

TypeScript の型は、そのベースとなっている JavaScript の動的な属性を包括的に扱えるように、かなり柔軟な定義もできるようになっています。高速な表描画ライブラリの CheetahGrid^{*1}では、カラムの定義を JSON で行うことができます。

```
const grid = new cheetahGrid.ListGrid({
  parentElement: document.querySelector('#sample2'),
  header: [
    {field: 'number', caption: 'number', columnType: 'number',
      style: {color: 'red'}},
    {field: 'check', caption: 'check', columnType: 'check',
      style: {
        uncheckBgColor: '#FDD',
        checkBgColor: 'rgb(255, 73, 72)'
      }
    }
  ]
});
```

(次のページに続く)

^{*1} <https://github.com/future-architect/cheetah-grid>

(前のページからの続き)

```

    }}
  ],
});

```

columnType の文字によって定義できる style の項目が変わります。今は、number と、check がありますね。check の時は uncheckBgColor と checkBgColor が設定できますが、number はそれらがなく、color があります。本物の CheetahGrid はもっと多くの属性があるのですが、ここでは、このルールだけを設定可能なインタフェースを考えてみます。簡略化のために属性の省略はないものとします（ただ?をつけるだけです）。

TypeScript のインタフェースの定義では「このキーがこの文字列の場合」という指定もできましたね。次の定義は、チェックボックス用の設定になります。columnType: 'check' という項目があります。

リスト 2 チェックボックスのカラム用の設定

```

type CheckStyle = {
  uncheckBgColor: string;
  checkBgColor: string;
}

type CheckColumn = {
  columnType: 'check';
  caption: string;
  field: string;
  style: CheckStyle;
}

```

数値用のカラムも定義しましょう。

リスト 3 数値用のカラム用の設定

```

type NumberStyle = {
  color: string;
}

type NumberColumn = {
  columnType: 'number';
  caption: string;
  field: string;
  style: NumberStyle;
}

```

上記のカラム定義の配列にはチェックボックスと数値のカラムの両方が来ます。どちらかだけの配列ではなくて、両方を含んでも良い配列を作ります。その場合は、合併型を使って、その配列と定義すれば、両方を入れてもエラーにならない配列が定義できます。ここでは type を使って、合併型に名前をつけています。それを配列にしています。

リスト 4 チェックボックス、数値の両方を許容する汎用的な「カラム」型を定義

```
// 両方の型を取り得る合併型を定義
type Column = CheckColumn | NumberColumn;

// 無事、エラーを出さずに過不足なく型付けできた
const header: Column[] = [
  {field: 'number', caption: 'number', columnType: 'number',
    style: {color: 'red'}}},
  {field: 'check', caption: 'check', columnType: 'check',
    style: {
      uncheckBgColor: '#FDD',
      checkBgColor: 'rgb(255, 73, 72)'
    }}
];
```

このように、一部の属性の値によって型が決定され、どちらかの型かが選択されるような合併型を、タグ付き合併型（Tagged Union Type）と呼びます。

注釈: どこまで細かく型をつけるべきか？

これらの機能を駆使すると、かなり細かく型定義が行え、利用者が変な落とし穴に陥いるのを防ぐことができます。

しかし、最初に述べたように、時間は有限です。型をつける作業は楽しい作業ではありますが、利用者数と見比べて、最初から全部を受け入れるような型を 1 つだけ作るどころから始めても良いでしょう。実際には次のような短い定義でも十分なことがほとんどです。

```
type Style = {
  color?: string;
  uncheckBgColor?: string;
  checkBgColor?: string;
}

type Column = {
  columnType: 'number' | 'check';
  caption: string;
  field: string;
  style: Style;
}
```

7.9 型ガード

静的な型付け言語では、どんどん型を厳しく付けていけばすべて幸せになりますよね！というわけにはいかない場面が少しだけあります。

TypeScript では、今まで見て来た通り、少し柔軟な型を許容しています。

- 数値型か、あるいは `null`
- 数値型か、文字列
- オブジェクトの特定の属性 `columnName` が `'check'` という文字列の場合のみ属性が増える

この複数の型を持つ変数を扱うときに、「2通りの選択肢があるうちの、こっちのパターンの場合のみのロジック」を記述したいときに使うのが型ガードです。

一般的な静的型付け言語でも、ダウンキャストなど、場合によってはプログラマーが意思を入れて型の変換を行わせることがあります。場合によっては、うまく変換できなかったときに実行時エラーが発生しうる、実行文です。

例えば、Go の場合、HTTP/2 の時は `http.ResponseWriter` は `http.Pusher` インタフェースを持っています。これにキャストすることで、サーバープッシュが実現できるという API 設計になっています。実行時にはランタイムが型を見て変数に値を代入するなどしてくれます。

リスト 5 Go のキャスト

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    if pusher, ok := w.(http.Pusher); ok {
        // ↑こちらでキャスト、成功すると bool 型の ok 変数に true が入る
        pusher.Push("/application.css", nil);
    }
})
```

しかし、TypeScript のソースコードはあくまでも、JavaScript に変換されてから実行されます。型情報などを消すだけで JavaScript になります。TypeScript のコンパイラが持つインタフェースや `type` などの固有の型情報は実行時にはランタイムには存在しません。そのため、「このオブジェクトがこのインタフェースを持っているとき」という実行文は他の言語のようにそのまま記述する方法はありません。

TypeScript がこれを解決する手段として実装しているのが、型ガードという機能です。型情報を全部抜くと単なる JavaScript としても有効な文ですが、TypeScript はこの実行文の文脈を解析し、型の選択肢を適切に絞り込んでいきます。これにより、正しいメソッドが利用されているかどうかを静的解析したりできますし、開発時においても、コード補完も正常に機能します。

リスト 6 型ガード

```
// userNameOrId は文字列か数値
let userNameOrId: string | number = getUser();

if (typeof userNameOrId === "string") {
```

(次のページに続く)

(前のページからの続き)

```
// この if 文の中では、userNameOrId は文字列型として扱われる
this.setState({
  userName: userNameOrId.toUpperCase()
});
} else {
  // この if 文の中では、userNameOrId は数値型として扱われる
  const user = this.repository.findUserByID(userNameOrId);
  this.setState({
    userName: user.getName()
  });
}
```

7.9.1 組み込みの型ガード

コンパイラは、一部の TypeScript の文を見て、型ガードと判定します。組み込みで使えるのは `typeof` や `instanceof`、`in` や比較です。

`typeof` **変数** は変数の型名を文字列で返します。プリミティブな組込型のいくつかでしか対応できません。

- `undefined`: "undefined"
- `bool` 型: "boolean"
- 数値: "number"
- 文字列: "string"
- シンボル: "symbol"
- 関数: "function"

これ以外のほとんどは `object` になります。 `null` も `object` になりますので、`typeof` は `null` の判定に使えません。

変数 `instanceof` **クラス名** は自作のクラスなどで使えるものになります。

"キー" `in` **オブジェクト** で、オブジェクトに特定の属性が含まれているかどうかの判定ができます。

`type` で型付けを行なったオブジェクトの複合型の場合、属性の有無や特定の属性の値がどうなっているかで判断できます。例えば、前述のカラム型の場合、`field` 属性に文字列が入っていて型の判別ができました。これは、その属性値の比較の `if` 文をかけば TypeScript のコンパイラはきちんと解釈してくれます。

```
type Column = CheckColumn | NumberColumn;

function getValue(column: Column): string {
  if (column.field === 'number') {
```

(次のページに続く)

(前のページからの続き)

```
// ここでは column は NumberColumn 型
} else {
  // ここでは column は CheckColumn 型
}
}
```

7.9.2 ユーザー定義の型ガード

TypeScript のベースになっている JavaScript では、長らくオブジェクトが配列かどうかを判定する明確な手法を提供してきませんでした。文字列にして、その結果をパースするとかも行われていました。ECMAScript 5 の時代によく、`Array.isArray()` というクラスメソッドが提供されるようになりました。

このようなメソッドは組み込みの型ガードとしては使えませんが、ユーザー定義の型ガード関数を作成すると、if 文の中で特定の型とみなすように TypeScript コンパイラに教えることができます。

型ガード関数は、次のような形式で書きます。

リスト 7 ユーザー定義の型ガード

```
// eslint-disable-next-line @typescript-eslint/no-explicit-any
function isArray(arg: any): arg is Array {
  return Array.isArray(arg);
}
```

- 名前は `is 型名` だとわかりやすい
- 引数は `arg: any`
- 返り値の型は `arg is Array`
- 関数の返り値は、型ガードの条件が満たされる実行文
- たいていのプロジェクトでは ESLint で `any` はエラーにしていると思うので、ここだけ明示的に使うために警告を抑制

なんども説明している通り、型ガードでは TypeScript のコンパイラだけが知っている情報は扱えません。JavaScript として実行時にアクセスできる情報 (`Array.isArray()` のような関数、`typeof`、`instanceof`、`in`、比較などあらゆる方法を駆使) を使って、`boolean` を返す必要があります。

7.9.3 型アサーション

TypeScript ではキャスト（型アサーション）もいちおうあります（`as` を後置で置く）が、これは実行文ではなくて、あくまでもコンパイラの持つ型情報を上書きするものです。型ガードとは異なり、実行時には情報を一切参照せずに、ただ変数の型だけが変わります。もちろん、`number` から `string` へのキャストなどの無理やりのキャストはエラーになりますが、`any` 型への変換はいつでも可能ですし、`any` から他の型への変換も自由にできます。一旦 `any` を挟むとコンパイラを騙してどんな型にも変換できてしまいますが、コンパイルエラーは抑制できても、実行時エラーになるだけなので、乱用しないようにしましょう。

```
const page: any = { name: "profile page" };
// any 型からは as でどんな型にも変換できる
const name: string = page as string;
```

7.10 keyof と Mapped Type: オブジェクトのキーの文字列のみを許容する動的な型宣言

この項目は中級者向けの項目になります。一般的にはジェネリクスと一緒に使うことが多い機能です。

JavaScript は動的なオブジェクトを駆使してプログラミングをしてきました。そのオブジェクトが他の言語でいう構造体、あるいはレコード型のように特定の属性を持つことが分かっている用途でのみ使われるのであれば今まで説明してきた機能だけで十分に利用できます。

一方、`Map` のように、何かしらの識別子をキーにして子供として要素を持つデータ構造として使われているケースなどもあります。例えばフォームの ID とその値をオブジェクトとして表現する場合は、フォームごとに項目が変わります。そのような用途では、「このキーがある」「このキーのみを対象としたい」「このキーの型情報」みたいな型宣言がしたくなります。`keyof` を使うとこのようなケースでの柔軟性があがります。

```
type Park = {
  name: string;
  hasTako: boolean;
};

// Park のキーである、 "name" | "hasTako" が割り当てられる
type Key = keyof Park;
// 指定されたキー以外はエラーになる
const key: Key = "name";
// 1 行でも書ける
const key: keyof Park = "hasTako";

// 値の方の型も取れる (string になる)
type ParkName = Park["name"];

// 指定されたキー以外はエラーになる
const key: keyof Park = "name";
```

また、オブジェクトのキー全部に対して型定義をすることもできます。構造としては次のように書きます。オブジェクトのキーは [] でくくることで式を書くことができました。その文法と似た書き方になっています。K というのがキー名の変数で、in によるループの要素が1つずつ入るイメージです。

```
// 基本の書き方
{[K in keyof Object]: プロパティの型}

// 入力の Object とキーは同じだがバリデーション結果を返す (値はすべて boolean)
{readonly [K in keyof Object]: boolean}

// 入力の Object とまったく同じものをこの記法で書いたもの
{[K in keyof Object]: Object[K]}

// 入力の Object とまったく同じだが読み込み専用
{readonly [K in keyof Object]: Object[K]}
```

なお、readonly を付与するのはジェネリクスなユーティリティ型 Readonly<T> というものがあるので実際にこのコードを書くことはないでしょう。

以下のコードが読み込み専用の型定義になります。

```
type ParkForm = {
  name: string;
  hasTako: boolean;
};

// 値を全て読み込み専用にした型
type FrozenParkForm = {readonly [K in keyof ParkForm]: ParkForm[K]};

const form: FrozenParkForm = {
  name: "恵比寿東",
  hasTako: true
};

// 読み込み専用なのでエラーになる
form.name = "和布刈公園"
```

7.11 インタフェースを使った型定義

オブジェクトの型をつける方法には、type を使う方法以外に、インタフェース定義を使った方法もあります。インタフェースは基本的には、Java 同様に他の章で紹介するクラスのための機能ですが、ほぼ同じことができますし、世間のコードではこちらの方もよく見かけます。

```
interface Person {
  name: string;
```

(次のページに続く)

(前のページからの続き)

```
favoriteBank: string;
favoriteGyudon?: string;
}
```

前述の型を合成する方法についても、二つのインタフェースの継承でも表現できますが、あまり見かけたことはありません。

```
interface PartyPeople extends Twitter, Instagram {
}

const shibukawa: PartyPeople = {
  twitterId: "@shibu_jp",
  instagramId: "shibukawa"
}
```

7.12 もし TypeScript の型を付けるのがコストが高くと感じたら？

TypeScript は既存の JavaScript の使い方のすべてをカバーできるように機能を拡充しています。例えば、関数ではあるが属性を持つものや同じ関数が引数によって様々な型を返す可能性があるケースなど、他の静的型を最初からもって生まれた言語ではまずみないような物にも型付けができるようになっています。しかし、できることと、少ないコストでもできる、という 2 つには差があります。

通常であれば外部からの入出力や関数の入りの宣言程度で済むはずで、実装コードの中の大部分は明示的に型を指定しなくても、推論で終了することが多いです。それでも済まないのは主に 4 つの理由が考えられます。

1. 利用するライブラリが TypeScript の型定義ファイルを用意していないケース。近年ではどんどん減っていますが、もし用意されていないのであれば型定義を自分で起こしたり、関数のレスポンスに自分で型定義を行う必要があるため、手間が増えます。
2. ライブラリの使用方法が TypeScript フレンドリーではないケース。Redux-Toolkit ではない通常の Redux では、ステートやアクションの型定義を利用者側が細かく用意しなければなりません。React もクラスコンポーネントよりは関数コンポーネントの方が開発者が設定しなければならない型注釈は少なく済みます。JavaScript 時代に作られたライブラリによっては TypeScript の型推論が効きにくいことがあります。例えば、Swagger や OpenAPI で型定義を行っているケースで、required が適切に付与されていないと、すべての属性が | undefined 付きになります。不要な Optional は、型ガードやアサーションが大量に必要なになります。
3. 同一の変数に多様な型のものを入れようとしていたり、関数の引数や戻り値も多様なものを返しているケース。null か何かしらのインスタンスか、ぐらいいれば、その関数の内部の実装や、利用者側のコードで型注釈が必要になることはごくまれです。特に戻り値の型の種類が多様になる場合は要注意です。
4. 多様な型を取り扱うためにジェネリクスなどの高度な型定義が必要になるケース。3 のケースに付随してそれをカバーするためのコードが複雑になる場合です。

TypeScript でわざわざ型情報を付与するのは、コーディングでコード補完によるすばやいコーディングを実現したり、コードを入力中やコンパイルで問題をすばやく解決し、不具合の検出にかかる時間を節約するためだったり、工数の削減が目的です。もし、工数が余計にかかるというのは、高コストなのはコードの設計が「読むときのコンテキスト次第で状態が変わる」ような設計が原因のことがほとんどだと思うので、型付けがシンプルになるように設計を直していくべきという指標になりえます。

あと、TypeScript のコンパイル（型チェック）は通るのに、実行時にエラーになるケースは、`as` などで手動で付けた型情報のミスが原因です。型チェックが信用できないので実行時に自分で `instanceof` などで型を見ざるをえないのであれば、それは上流の型定義を直していきましょう。TypeScript のコンパイラがおかしい、信用できない、と思ったときは 99% 利用者側の責任でしょう。

7.13 まとめ

基本的な型付けの作法、とくにオブジェクトに対する型付けを学びました。JavaScript の世界では、プログラムのロジック以上に、柔軟なデータ構造を活用したコーディングが他の言語以上に行われていました。そのため、ここで紹介した機能は、その JavaScript の世界に型を設定していくうえで必要性の高い知識となります。

これから紹介するクラスの場合は、実装時に自然と型定義もできあがりますが、TypeScript ではクラスに頼らない関数型スタイルのコーディングも増えています。このオブジェクトの型付けは関数の入出力でも力を発揮するため、身につけておいて損はないでしょう。

第 8 章

関数

関数は一連の処理に名前をつけてまとめたものです。他の人の作ったものを利用するだけでは関数などなくても、必要な処理を必要なだけ列挙すれば期待する結果が得られるコードは理論上は実現可能です。しかし、数万行の一連のコードを扱うのは事実上不可能です。

そこで理解できる大きさにグループ化して、名前をつけたものが関数です。関数呼び出しはネストできるので、難しいロジックに名前をつけて関数を作り、それらのロジックを並べたちょっと複雑なタスクを人間の仕事に近い高水準な関数にできます。関数は決まった処理を単純に実行するだけではなく、引数をとって、柔軟に動作させることもできますし、戻り値を返すこともできます。

どの程度の分量が適切かはロジックの複雑さによります。単純な仕事を延々に行っている（React のコンポーネントのレンダリングなど）であれば、数画面分のコードでもなんとかなるでしょうし、帰って細かく分け過ぎてしまうと、全体像の把握が難しくなります。一方で複雑なロジックだと 20 行でも難しいかもしれません。

関数の基本形態は以下の通りです。

- 関数は名前を持ちます。何かに代入したり、引数で渡す場合は省略可能です。
- 引数はカンマ区切りで名前と型を列挙していきます。引数がない場合は省略可能です。引数の型はジェネリクスを使って省略可能にもできますが、基本的に指定が必要です。
- 戻り値は `return` で返します。戻り値がない場合は `return` を省略可能です。
- もし `return` 文の数が一つ、または複数個あっても、毎回同じ型を返しているのであれば `return` の型は省略可能です。

```
function 関数名 (引数リスト): 戻り値の型 {  
  return 戻り値  
}
```

TypeScript の関数には次のような特徴があります。

- 関数そのものを変数に入れて名前をつけられるし、データ構造にも組み込める
- 他の関数の引数に関数を渡せる

- 関数の返り値として返すことができる

関数を受け取る関数があると、プログラムの柔軟性が飛躍的に高まります。従来の JavaScript は関数の使い勝手が極めて便利だった一方で、言語の他の機能は少なく、関数を多用した数々のテクニックが生み出されました。一方で、かなり黒魔術な、一見すると魔法のような使われ方も多数ありました。近年では ECMAScript や TypeScript のバージョンアップで数多くの機能が増え、トリッキーな使い方はだいぶ減っていますが、重要なことには変わりません。

関数にはいくつかのバリエーションがあります。

- 名前がない関数は無名関数、あるいはアノニマス関数と呼びます。変数に代入したり、他の関数の引数に渡す場所で利用されます。とくに、関数の内部で作られる関数を「クロージャ」と呼びます。
- 何かのオブジェクトに属する関数は「メソッド」と呼びます。
- 時間のかかる処理を行い、それが終わるまで他のタスクを途中でできる関数を非同期関数と呼びます（**非同期処理**の章で扱います）

名前のない無名関数にはアロー演算子を使った省略記法も追加されました。ふつうの `function` キーワードを使った宣言とはほぼ同等で置き換えできることが多いのですが、動作については少し異なる部分もあります。こちらについても順をおって説明します。

```
const f = (引数リスト) => {  
  return 返り値  
}
```

関数は TypeScript の中では、プログラムの構成をつくるための重要な部品です。いたるところで使われます。言語のバージョンアップとともに、定義、使い方などいろいろ追加されました。表現したい機能のために、ややこしい直感的でないコードを書く必要性がかなり減っています。

8.1 関数の引数と返り値の型定義

TypeScript では関数やクラスのメソッドでは引数や返り値に型を定義できます。元となる JavaScript で利用できる、すべての書き方に対応しています。

なお、Java などとは異なり、同名のメソッドで、引数違いのバリエーションを定義するオーバーロードは使えません。

リスト 1 関数への型付け

```
// 昔からある function の引数に型付け。書く引数の後ろに型を書く。  
// 返り値は引数リストの () の後に書く。  
function checkFlag(flag: boolean): string {  
  console.log(flag);  
  return "check done";  
}
```

(次のページに続く)

(前のページからの続き)

```
// アロー関数も同様
const normalize = (input: string): string => {
  return input.toLowerCase();
}
```

変数の宣言のときと同じように、型が明確な場合には省略が可能です。

リスト 2 関数への型付け

```
// 文字列の toLowerCase() メソッドの返り値は文字列なので
// 省略しても string が設定されたと見なされる
const normalize = (input: string) => {
  return input.toLowerCase();
}

// 文字配列の降順ソート
// ソートに渡される比較関数の型は、配列の型から明らかなので省略しても OK
// 文字列の toLowerCase() メソッドも、エディタ上で補完が効く
const list: string[] = ["小学生", "小心者", "小判鮫"];
list.sort((a, b) => {
  if (a.toLowerCase() < b.toLowerCase()) {
    return 1;
  } else if (a.toLowerCase() > b.toLowerCase()) {
    return -1;
  }
  return 0;
});
```

関数が何も返さない場合は、`: void` をつけることで明示的に表現できます。実装したコードで何も返していなければ、自動で `: void` がついているとみなされますが、これから先で紹介するインタフェースや抽象クラスなどで、関数の形だけ定義して実装を書かないケースでは、どのように判断すればいいのか材料がありません。`compilerOptions.noImplicitAny` オプションが `true` の場合には、このようなケースで `: void` を書かないとエラーになりますので、忘れずに書くようにしましょう。

リスト 3 何も返さない時は void

```
function hello(): void {
  console.log("ごきげんよう");
}

interface Greeter {
  // noImplicitAny: trueだとエラー
  // error TS7010: 'hello', which lacks return-type annotation,
  //   implicitly has an 'any' return type.
  hello();
}
```

要注意なのは、レスポンスの型が一定しない関数です。次の関数は、2019 が指定された時だけ文字列を返します。この場合、TypeScript が気を利かせて `number | '今年'` という返り値の型を暗黙でつけてくれます。しかしこの場合、単純な `number` ではないため、`number` 型の変数に代入しようとするとエラーになります。

ただ、このように返り値の型がバラバラな関数を書くことは基本的にないでしょう。バグを生み出しやすくなるため、返り値の型は特定の型 1 つに限定すべきです。バリエーションがあるとしても、`| null` をつけるぐらいにしておきます。

内部関数で明らかな場合は省略しても問題ありませんが、公開関数の場合はなるべく省略をやめた方が良いでしょう。

```
// 返り値の型がたくさんある、行儀の悪い関数
function yearLabel(year: number) {
  if (year === 2019) {
    return '今年';
  }
  return year;
}

const label: number = yearLabel(2018);
// error TS2322: Type 'number | "今年"' is not assignable to type 'number'.
//   Type '"今年"' is not assignable to type 'number'.
```

8.2 関数を扱う変数の型定義

関数に型をつけることはできるようになりました。次は、その関数を代入できる変数の型を定義して見ましょう。

例えば、文字列と数値を受け取り、`boolean` を返す関数を扱いたいとしてます。その関数は `check` という変数に入れます。その場合は次のような宣言になります。引数はアロー関数のままですが、返り値だけは `=>` の右につけ、`{ }` は外します。型定義ではなく、実際のアロー関数の定義の返り値は `=>` の左につきます。ここが逆転する点に注意してください。

```
let check: (arg1: string, arg2: number) => boolean;
```

arg2 がもし関数であったら、関数の引数の中に関数が出てくるということで、入れ子の宣言になります。多少わかりにくいのですが、内側から順番に剥がして理解していくのがコツです。ネストが深くなり、理解が難しい場合は type 宣言で型定義を切り出して分解していく方が良いでしょう。

```
let check: (arg1: string, arg2: (arg3: string) => number) => boolean;
```

サンプルとしてカスタマイズ可能なソート関数を作りました。通常のソートだと、すべてのソートを行うためになんども比較関数が呼ばれます。大文字小文字区別なく、A-Z 順でソートしたいとなると、その変換関数が大量に呼ばれます。本来は 1 要素につき 1 回ソートすれば十分なはずですが。それを実装したのが次のコードです。

まず、変換関数を通しながら、**「オリジナル、比較用に変換した文字列」**という配列を作ります。その後、後半の変換済みの文字列を使ってソートを行います。最後に、そのソートされた配列を使い、オリジナルの配列に含まれていた要素だけの配列を再び作成しています。

リスト 4 一度だけ変換するソート

```
function sort(a: string[], conv: (value: string) => string) {
  const entries = a.map((value) => [value, conv(value)])
  entries.sort((a, b) => {
    if (a[1] > b[1]) {
      return 1;
    } else if (a[1] < b[1]) {
      return -1;
    }
    return 0;
  });
  return entries.map(entry => entry[0]);
}

const a: string[] = ["a", "B", "D", "c"];
console.log(sort(a, s => s.toLowerCase()))
// ["a", "B", "c", "D"]
```

8.3 関数を扱う変数に、デフォルトで何もしない関数を設定する

コールバック関数を登録しておく変数に対し、何も代入されないときに呼び出し元が存在チェックをサボっていると、undefined に対して関数呼び出しをしたとエラーが発生します。その場合は、とりあえず何もしない関数を代入してエラーを回避したいと思うでしょう。

JavaScript の世界では型がないため、とりあえず引数を持たず、本体が空の無名関数を入れてしまうと回避はできません。

```
// 何もしない無名関数を入れておく
var callback = function() {};
```

TypeScript では、例え引数を利用しなかったとしても、また実際に実行されないので `return` 文を省略した場合でも、変数の関数の型と合わせる必要があります。わかりやすさのために、変数宣言と代入を分けたコードを提示します。

```
// 変数宣言（代入はなし）
let callback: (name: string) => void;

// ダミー関数を設定
callback = (name: string): void => {};
```

もちろん、1 行にまとめることもできます。JavaScript 的にはどれも違いのない「関数」ですが、引数と戻り値が違う関数は TypeScript の世界では「別の型」として扱われますし、何もしない無名関数は引数も戻り値もない関数の型を持っている、という判断が行われます。実際のロジックが空でも定義が必要な点は要注意です。

```
// 変数宣言（代入で推論で型を設定）
let callback = (name: string): void => {};
```

8.4 デフォルト引数

TypeScript は、他の言語と同じように関数宣言のところに引数のデフォルト値を簡単に書くことができます。また、TypeScript は型定義通りに呼び出さないとエラーになるため、引数不足や引数が過剰になる、というエラーチェックも不要です。

```
// 新しいデフォルト引数
function f(name="小動物", favorite="小豆餅") {
  console.log(`${name}は${favorite}が好きです`);
}
f(); // 省略して呼べる
```

オブジェクトの分割代入を利用すると、デフォルト値付きの柔軟なパラメータも簡単に実現できます。以前は、オプションな引数は `opts` という名前のオブジェクトを渡すこともよくありました。今時であれば、完全省略時にはデフォルト値が設定され、部分的な設定も可能な引数が次のように書けます。

```
// 分割代入を使って配列やオブジェクトを変数に展開&デフォルト値も設定
// 最後の={ }がないとエラーになるので注意
function f({name="小動物", favorite="小豆餅"}={}) {
  :
}
```

JavaScript は同じ動的言語の Python とかよりもはるかにゆるく、引数不足でも呼び出すこともでき、その場合には変数に `undefined` が設定されました。undefined の場合は省略されたとみなして、デフォルト値を設定す

るコードが書かれたりしました。どの引数が省略可能で、省略したら引数を代入しなおしたり・・・とか面倒ですし、同じ型の引数があつたら判別できなかったりもありますし、関数の先頭行付近が引数の処理で 1 画面分埋まる、ということも多くありました。また、可変長引数があつてもコールバック関数がある場合は必ず末尾にあるというスタイルが一般的でしたが、この後に説明する Promise を返す手法が一般的になったので、こちらでも取扱いが簡単になりました。

```
// デフォルト引数の古いコード
function f(name, favorite) {
    if (favorite === undefined) {
        favorite = "小豆餅";
    }
}

// 古くてやっかいな、コールバック関数の扱い
function f(name, favorite, cb) {
    if (typeof favorite === "function") {
        cb = favorite;
        favorite = undefined;
    }
    :
}
```

8.5 関数を含むオブジェクトの定義方法

ES2015 以降、関数や定義の方法が増えました。JavaScript ではクラスを作るまでもない場合は、オブジェクトを作って関数をメンバーとして入れることができますが、それが簡単にできるようになりました。setter/getter の宣言も簡単に行えるようになりました。

リスト 5 関数を含むオブジェクトの定義方法

```
// 旧: オブジェクトの関数
var smallAnimal = {
    getName: function() {
        return "小動物";
    }
};

// 旧: setter/getter 追加
Object.defineProperty(smallAnimal, "favorite", {
    get: function() {
        return this._favorite;
    },
    set: function(favorite) {
        this._favorite = favorite;
    }
});
```

(次のページに続く)

(前のページからの続き)

```
// 新: オブジェクトの関数
//     function を省略
//     setter/getter も簡単に
const smallAnimal = {
  getName() {
    return "小動物"
  },
  _favorite: "小笠原",
  get favorite() {
    return this._favorite;
  },
  set favorite(favorite) {
    this._favorite = favorite;
  }
};
```

8.6 クロージャと this とアロー関数

関数の中で関数を定義したときに、関数は自分の定義の外にある変数を参照できます。

```
function a() {
  const b = 10;
  function c() {
    console.log({b}); // b が表示される
  }
  c();
}
```

実行時の親子関係ではなく、ソースコードという定義時の親子関係を元にしてスコープが決定されます。これをレキシカルスコープと呼びます。また、このように自分が定義された場所の外の変数を束縛した関数を「クロージャ」と呼びます。TypeScript では関数を駆使してロジックを組み上げていきますので、この機能はとても重要です。

以前は Java のようなオブジェクトを実装するために、関数内部の変数をプライベートメンバー変数のように扱うテクニックがかつてありました。クラスの機能が公式のサポートされたので、今では重要度は低くなっているし、そもそも隠す必要性もあまりないので使うことはありませんが、頭の体操にはなるので、興味がある方は調べてみてください。

レキシカルスコープは今では多くの言語が持っている機能なので、わざわざ名前を呼ぶこともありませんが、TypeScript では、知らないと落とし穴に落ちる可能性のあるやや重要な機能となります。前項でオブジェクトの中の関数定義を紹介しました。ここでは、予め定義された変数のように `this` を使っています。しかしこれは変数ではなく、特別な識別子です。レキシカルスコープで束縛できません。クロージャかつ、`this` への束縛ができる新文法としてアロー関数が追加されました。

8.6.1 アロー関数

JavaScript では、やっかいなのが `this` です。無名関数をコールバック関数に渡そうとすると、`this` がわからなくなってしまう問題があります。アロー関数を使うと、その関数が定義された場所の `this` の保持までセットで行いますので、無名関数の `this` 由来の問題をかなり軽減できます。表記も短いため、コードの幅も短くなり、コールバックを多用するところで `function` という長いキーワードが頻出するのを減らすことができます。

リスト 6 アロー関数

```
// アロー関数ならその他の this が維持される。
this.button.addEventListener("click", () => {
  this.smallAnimal.walkTo("タコ公園");
});
```

アロー関数にはいくつかの記法があります。引数が 1 つの場合は引数のカッコを、式の結果をそのまま `return` する場合は式のカッコを省略できます。ただし、引数の場所に型をつけたい場合は省略するとエラーになります。

リスト 7 アロー関数の表記方法のバリエーション

```
// 基本形
(arg1, arg2) => { /* 式 */ };

// 引数が 1 つの場合は引数のカッコを省略できる
// ただし型を書くともエラーになる
arg1 => { /* 式 */ };

// 引数が 0 の場合はカッコが必要
() => { /* 式 */ };

// 式の { } を省略すると、式の結果が return される
arg => arg * 2;

// { } をつける場合は、値を返すときは return を書かなければならない
arg => {
  return arg * 2;
};
```

以前は、`this` がなくなってしまうため、`bind()` を使って束縛したり、別の名前（ここでは `self`）に退避する必要がありました。そのため、`var self = this;` と他の変数に退避するコードがバッドノウハウとして有名でした。

リスト 8 `this` 消失を避ける古い書き方

```
// 旧： 無名関数のイベントハンドラではその関数が宣言されたところの this にアクセスできない
var self=this;
this.button.addEventListener("click", function() {
  self.smallAnimal.walkTo("タコ公園");
});
```

(次のページに続く)

(前のページからの続き)

```
// 旧: bind() で現在の this に強制束縛
this.button.addEventListener("click", (function() {
  this.smallAnimal.walkTo("タコ公園");
}).bind(this));
```

8.7 this を操作するコードは書かない (1)

読者のみなさんは JavaScript の this が何種類あるか説明できるでしょうか? apply() や call() で実行時に外部から差し込み、何も設定しない (グローバル)、bind() で固定、メソッドのピリオドの右辺が実行時に設定、といったバリエーションがあります。これらの this の違いを知り、使いこなせるのがかつての JavaScript 上級者でしたが、このようなコードはなるべく使わないように済ませたいものです。

無名関数で this がグローバル変数になってはズれてしまうのはアロー関数で解決できます。

apply() は、関数に引数セットを配列で引き渡したいときに使っていました。配列展開の文法のスプレッド構文... を使うと、もっと簡単にできます。

```
function f(a, b, c) {
  console.log(a, b, c);
}
const params = [1, 2, 3];

// 旧: a=1, b=2, c=3 として実行される
f.apply(null, params);

// 新: スプレッド構文を使うと同じことが簡単に行える
f(...params);
```

call() は配列の push() メソッドのように、引数を可変長にしたいときに使っていました。関数の中で引数全体は arguments という名前のちょっと配列っぽいオブジェクトで参照されます。そのままではちょっと使いにくいので一旦本物の配列に代入したいという時、call() を使って配列のメソッドを arguments に適用するハックがよく利用されていました。これも引数リスト側に残余 (Rest) 構文を使うことで本体にロジックを書かずに実現できます。

```
// 旧: 可変長配列の古いコード
function f(a, b) {
  // この 2 は固定引数をスキップするためのもの
  var list = Array.prototype.slice.call(arguments, 2);
  console.log(a, b, list);
}
f(1, 2, 3, 4, 5, 6);
// 1, 2, [3, 4, 5, 6];
```

(次のページに続く)

(前のページからの続き)

```
// 新: スプレッド構文。固定属性との共存もラクラク
const f = (a, b, ...c) => {
  console.log(a, b, c);
};
f(1, 2, 3, 4, 5, 6);
// 1, 2, [3, 4, 5, 6];
```

ただし、jQuery などのライブラリでは、this がカレントのオブジェクトを指すのではなく、選択されているカレントノードを表すという別解釈を行います。使っているフレームワークが特定の流儀を期待している場合はそれに従う必要があります。

bind() の排除はクラスの中で紹介します。

8.8 即時実行関数はもう使わない

関数を作ってその場で実行することで、スコープ外に非公開にしたい変数などが見えないようにするテクニックがかつてありました。即時実行関数と呼びます。function() {} をかっこでくくって、その末尾に関数呼び出しのための () がさらに付いています。これで、エクスポートしたい特定の変数だけを return で返して公開をしていました。今時であれば、公開したい要素に明示的に export をつけると、webpack などのツールがそれ以外の変数をファイル単位のスコープで隠してくれます。

リスト 9 古いテクニックである即時実行関数

```
var lib = (function() {
  var libBody = {};
  var localVariable;

  libBody.method = function() {
    console.log(localVariable);
  }
  return libBody;
})();
```

8.9 まとめ

関数についてさまざまなことを紹介してきました。

- 関数の引数と返り値の型定義
- 関数を扱う変数の型定義
- デフォルト引数
- 関数を含むオブジェクトの定義方法

- クロージャと `this` とアロー関数
- `this` を操作するコードは書かない (1)
- 即時実行関数はもう使わない

省略、デフォルト引数など、JavaScript では実現しにくかった機能も簡単に実装できるようになりました。関数は、TypeScript のビルディングブロックのうち、大きな割合をしめています。近年では、関数型言語の設計を一部取り入れ、堅牢性の高いコードを書こうというムーブメントが起きています。ここで紹介した型定義をしっかりと行くと、その関数型スタイルのコードであっても正しく型情報のフィードバックされますので、ぜひ怖がらずに型情報をつけていってください。

関数型志向のプログラミングについては後ろの方の章で紹介します。

第 9 章

その他の組み込み型・関数

これまでの説明の中で、いくつかのデータの種類（ここでは誤解をおそれず、大雑把にクラスとしておきます）について説明してきました。

- プリミティブ型
 - boolean
 - number
 - string と正規表現
 - undefined と null
- 複合型
 - Array
 - Object
 - Map と WeakMap
 - Set と WeakSet
- 関数

TypeScript と JavaScript はブラウザの中の言語として作られたため、数多くのブラウザ環境専用のクラスを備えています。それ以外にも言語の組み込みのクラスがいくつかありますので、本章ではそれらについて説明していきます。なお、Error クラスについては[例外処理](#)で触れます。

本章で扱う組み込み型や関数は次の通りです

- Date: 日付を扱うクラス
- RegExp: 正規表現

- JSON: JSON 形式の文字列をパースしたり、JavaScript のオブジェクトを JSON 形式にシリアライズするクラス。サーバー通信時のボディでよく利用される。
- URL と URLSearchParams: URL をパースしたり、URL を組み立てるクラス。サーバー通信時に URL やクエリーを取り出すのに利用される。
- 時間のための関数

9.1 Date

日付と時間を扱うのが Date 型です。これは最初期から実装されている型で、TypeScript や JavaScript で日付を扱う場合、まず出てくるのがこれです。ただし、即席で作られたこともあって評判がよくなく、大切な機能がいくつか抜け落ちていたり、文字列のパースが柔軟性がなかったりするため、いくつかの追加のモジュールなどが作られています。

以前は `moment.js` が長い間広く使われてきましたが、現在は積極的な開発を中止する声明を出しています。その `moment.js` の声明の中で推奨されているのが次のライブラリです。

- `Luxon` (型定義は別に `npm install @types/luxon` が必要)
- `Day.js` (TypeScript 型定義同梱)
- `date-fns` (TypeScript 型定義同梱)
- `js-Joda` (TypeScript 型定義同梱)

より便利なフォーマットやパース、日時演算などはこちらのパッケージを利用すると簡単に行えます。

JavaScript 本体においても、Date を置き換える Temporal が提案されています。

- <https://github.com/tc39/proposal-temporal>

将来的に、ここの説明は Temporal ベースで書き換えられると思いますが、ひとまずここでは Date のよくある使い方について紹介していきます。

TypeScript の Date 型は数字に毛の生えたようなものですので、それを前提にみていくと良いと思います。

9.1.1 現在時刻の取得・エポック時刻

`new Date()` で簡単に作成できます。コンピュータに保存されているタイムゾーンの情報も含んだ、Date のインスタンスが作成できます。

```
// 現在時刻で Date のインスタンス作成
const now = new Date();
// new を付けないと文字列として帰ってくる
```

(次のページに続く)

(前のページからの続き)

```
const nowStr = Date();  
// 'Sun Sep 06 2020 22:36:08 GMT+0900 (Japan Standard Time)'
```

コンピュータの世界では UNIX 時刻、あるいは UNIX 秒、エポック (Epoch) 秒、エポック時刻と呼ばれるものがよく使われます。これは 1970 年 1 月 1 日 (UTC 基準) からの経過時間で時間を表すものです。JavaScript の中ではミリ秒単位であって秒ではないため、本章ではエポック時刻という名前で統一します。

```
// ミリ秒単位のエポック時刻取得  
const now = Date.now();
```

`console.time()` と `console.timeEnd()` でも時間計測ができますが、何かしらの処理の間の時間を撮りたい場合には、`Date.now()` を複数回呼び出すことで、ミリ秒単位で時間が計測できます。ブラウザでは `performance.now()` という高精度タイマーがありましたが、セキュリティの懸念もあって現在は精度が落とされていますので、`Date.now()` とあまり差はないでしょう。

```
const start = Date.now();  
// :  
// 時間のかかる処理  
// :  
const duration = Date.now() - start;  
// 経過時間 (ミリ秒) の取得
```

このエポック時刻から `Date` のインスタンスにする場合は `new Date()` の引数にミリ秒単位の時間を入れます。逆に、`Date()` のインスタンスからエポック時刻を取得するには `valueOf()` メソッドを使います。

```
// 現在の時刻から 100 秒 (10 万ミリ秒) 前の時刻の取得  
const hundredSecAgo = new Date(Date.now() - 100 * 1000);  
  
// エポック時刻取得  
const epoch = hundredSecAgo.valueOf();
```

さまざまな時間の情報がありますが、TypeScript ではどれを基準に扱うべきでしょうか？ブラウザはユーザーインタフェースであるため、ユーザーの利用環境のタイムゾーン情報を持っています。しかし、多くのユーザーの情報を同時に扱うサーバーではタイムゾーン情報も含めて扱うのは手間隙がかかります。データベースエンジンによってはタイムゾーン込みの時刻も扱いやすいものもあったりはしますが、シンプルに扱うためには以下の指針で大部分のシステムはまかなえるでしょう。

- クライアントで時刻を取得してサーバーに送信するときは、`Date().now` などでエポック時刻にしてからサーバーに送信する (タイムゾーン情報なし)
- サーバーでは常にエポック時刻で扱う (ただし、言語によっては秒単位だったり、ミリ秒単位だったり、マイクロ秒単位だったり違いはあるため、そこはルールを決めておきましょう)
- サーバーからフロントに送った段階で `new Date()` などを使って、ローカル時刻化する

9.1.2 特定の日時の Date インスタンスの作成

特定の日時インスタンスを作成するには、Date () コンストラクタの引数に数値を設定して作成します。月の数値が 1 少なく評価される (1 月は 0) な点に注意が必要です。この日時は現在のタイムゾーンで評価されます。

```
// 2020 年 9 月 21 日 21 時 10 分 5 秒
// 日本で実行すると日本時間 21 時 (UTC では 9 時間前の 12 時) に
const d = new Date(2020, 8, 21, 21, 10, 5)
```

UTC の時刻から生成したい場合には、Date.UTC () 関数を使います。これはエポック秒を返すのでこれを new Date () に渡すことで、UTC 指定の時刻のインスタンスが作成できます。

```
// UTC の 2020 年 9 月 21 日 11 時 10 分 5 秒
// 日本で実行すると日本時間 20 時 (日本時間は UTC は 9 時間進んでいるように見える) に
const d = new Date(Date.UTC(2020, 8, 21, 11, 10, 5))
```

9.1.3 日付のフォーマット出力

RFC 3393 形式にするには、toISOString () メソッドを使います。toString () だと ECMAScript の仕様書で定められたロケール情報も含む文字列で出力を行います。後者の場合、ミリ秒単位のデータは丸められてしまいます。

```
const now = new Date()
now.toISOString()
// '2020-09-21T12:38:15.655Z'

now.toString()
// 'Mon Sep 21 2020 21:38:15 GMT+0900 (Japan Standard Time)'
```

短い形式やオリジナルの形式にするには自分でコードを書く必要があります。短く日時を表現しようとする場合のコードは次のようになります。月のみカレンダーの表記と異なって、0 が 1 月になる点に注意してください。

```
const str = `${
  now.getFullYear()
}/${
  String(now.getMonth() + 1).padStart(2, '0')
}/${
  String(now.getDate()).padStart(2, '0')
} ${
  String(now.getHours()).padStart(2, '0')
}:${
  String(now.getMinutes()).padStart(2, '0')
}:${
  String(now.getSeconds()).padStart(2, '0')
}
```

(次のページに続く)

(前のページからの続き)

```
};  
// "2020/09/06 13:55:43"
```

`padStart()` と、テンプレート文字列のおかげで、以前よりははるかに書きやすくなりましたが、`Day.js` などの提供するフォーマット関数を使った方が短く可読性も高くなるでしょう。

9.1.4 日付データの交換

クライアントとサーバーの間では JSON などを通じてデータをやりとりします。JSON でデータを転送するときは数値か文字列で表現する必要があります。日付データは既に説明した通りに、ミリ秒か秒のどちらかの数値で交換するのがもっともコード的には少ない配慮で実現できます。しかし、一方で出力された数字の列を見ても、即座に現在の日時を暗算できる人はいないでしょう。可読性という点では文字列を使いたいこともあるでしょう。本節ではデータをやりとりする相手ごとのデータ変換の仕方について詳しく説明していきます。

TypeScript (含む JavaScript 同士)

TypeScript (含む JavaScript) であれば、`toISOString()` でも、`toString()` でも、どちらの方式で出力した文字列であってもパースできます。`new Date()` に渡すと `Date` のインスタンスが、`Date.parse()` に渡すと、エポック時刻が帰ってきます。

```
const fromToISOString = new Date(`2020-09-21T12:38:15.655Z`)  
// 2020-09-21T12:38:15.655Z  
  
const fromToString = new Date(`Mon Sep 21 2020 21:38:15 GMT+0900 (Japan Standard_`  
↪Time)`)  
// 2020-09-21T12:38:15.000Z  
  
const fromToISOStringEpoch = Date.parse(`2020-09-21T12:38:15.655Z`)  
// 1600691895655  
  
const fromToStringEpoch = Date.parse(`Mon Sep 21 2020 21:38:15 GMT+0900 (Japan_`  
↪Standard Time)`)  
// 1600691895000
```

Go との交換の場合

Go では日付のフォーマットが何種類か選べますが、このうち、タイムゾーンの時差が数値で入っているフォーマットは TypeScript でパース可能です。ナノ秒の情報がミリ秒に丸められてしまいますが、一番精度良く伝達できるのは `time.RFC3339Nano` の出力です。

- `time.RubyDate`
- `time.RFC822Z`
- `time.RFC1123Z`
- `time.RFC3339`
- `time.RFC3339Nano`

リスト 1 Go で RFC3339Nano で出力

```
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now()
    fmt.Println(now.Format(time.RFC3339Nano))
    // 2020-09-21T21:35:45.057076+09:00
}
```

リスト 2 TypeScript でパース

```
const receivedFromGo = new Date(`2020-09-21T21:35:45.057076+09:00`)
// 2020-09-21T12:35:45.057Z
```

他の言語に出力する場合、`toISOString()` が無難でしょう。Go は `time.RFC3339` か、`time.RFC3339Nano` を使ってパースできます。結果はどちらも同じです。

```
package main

import (
    "fmt"
    "time"
)

func main() {
    t, err := time.Parse(time.RFC3339, "2020-09-21T12:38:15.655Z")
    fmt.Println(t)
```

(次のページに続く)

(前のページからの続き)

```
// 2020-09-21 12:38:15.655 +0000 UTC
}
```

Python との交換の場合

Python で出力する場合は `datetime.datetime.isoformat()` メソッドを使うと良いでしょう。このメソッドはタイムゾーン情報を取り払い、現在のタイムゾーンの表記そのものをフォーマットして出力します。TypeScript の `new Date()` は UTC であることを前提としてパースするため、出力時は UTC として出すように心がける必要があります。この UTC で出力された文字列は TypeScript でパースできます。

```
from datetime import datetime, timezone

# local の場合は astimezone() を呼んで UTC に
localtime = datetime.now()
utctime = localtime.astimezone(timezone.utc)
utctime.isoformat()
# 2020-09-21T13:42:58.279772+00:00

# あるいは、最初から UTC で扱う
utctime = datetime.utcnow()
utctime.isoformat()
# 2020-09-21T13:42:58.279772+00:00
```

パースはやっかいです。Stack Overflow でスレッドが立つぐらいのネタです^{*1}。Python 3.7 からは `fromisoformat()` というクラスメソッドが増えましたが、以前からの `datetime.strptime()` にフォーマット指定を与えた方が高速とのことでした。

```
from datetime import datetime

s = '2020-09-21T12:38:15.655Z'

datetime.fromisoformat(s.replace('Z', '+00:00'))
# datetime.datetime(2020, 9, 21, 12, 38, 15, 655000, tzinfo=datetime.timezone.utc)

datetime.strptime(s, '%Y-%m-%dT%H:%M:%S.%f%Z')
# datetime.datetime(2020, 9, 21, 12, 38, 15, 655000, tzinfo=datetime.timezone.utc)
```

いっそのこと、エポック時刻で扱う方法の方がシンプルでしょう。TypeScript はミリ秒単位で、Python は秒単位なので、1000 で割ってから渡す必要があるのと、UTC の数値であることを明示する必要があります。

```
from datetime import datetime
datetime.fromtimestamp(1600691895655 / 1000.0, timezone.utc)
# datetime.datetime(2020, 9, 21, 12, 38, 15, 655000, tzinfo=datetime.timezone.utc)
```

*1 <https://stackoverflow.com/questions/127803/how-do-i-parse-an-iso-8601-formatted-date>

Java との交換の場合 (8 以降)

Java8 から標準になったクラス群^{*2}を使って TypeScript との交換を行ってみます。ここで紹介するコードは Java8 以降で動作するはずです。このサンプルは Java 11 で検証しています。

`Date.toISOString()` と同等の出力は `DateTimeFormatter` で作成できます。UTC のゾーンになるようにインスタンスを作成してから `format()` メソッドを使って変換します。

```
import java.time.Instant;
import java.time.ZonedDateTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;

class ParseTest {
    public static void main(String[] args) {
        var RFC3339_FORMAT = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'
↪");

        var l = ZonedDateTime.now();
        var isoString = l.withZoneSameInstant(ZoneOffset.UTC).format(RFC3339_FORMAT);
        System.out.println(isoString);
        // "2020-09-23T13:55:53.780Z"
    }
}
```

この文字列は TypeScript でパースできます。

TypeScript で生成した文字列のパースには前述の `DateTimeFormatter` も使えますが、それ以外には `java.time.Instant` が使えます。これはある時点での時刻を表すクラスです。TypeScript の `Date.toISOString()` の出力する文字列をパースできます。実際に日時の操作を行う `LocalDateTime` や `ZonedDateTime` へも、ここから変換できます。次のサンプルは `Instant` でパースし、`ZonedDateTime` に変換しています。

```
import java.time.Instant;
import java.time.ZonedDateTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;

class ParseTest {
    public static void main(String[] args) {
        var i = Instant.parse("2020-09-23T14:06:11.027Z");
        var l = ZonedDateTime.ofInstant(i, ZoneId.systemDefault());

        var f = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        System.out.println(l.format(f));
    }
}
```

(次のページに続く)

^{*2} Java8 の日時 API はとりあえずこれだけ覚えとけ

(前のページからの続き)

```
}  
}
```

9.1.5 1 時間後、1 日後、1 ヶ月後、1 年後の日時の取得

バッチの次回実行時間の計算や、以前のバッチの間に含まれる情報のフィルタリングのために、1 時間後や 1 日後、1 ヶ月後、あるいは 1 時間前や 1 日前といった日時の演算が必要になることがあります。

現在時刻の `Date` インスタンスを作成し、現在の情報を元に、期待する値のインクリメントを行うだけです。1 時間後を計算するときに、12 月 31 日の 23 時 30 分だから年月日をインクリメントしなければならない、今年の 2 月は閏年だから 1 年後の計算の日数が変わる、みたいなことは考慮する必要はなく、そのような演算は JavaScript が行います。

`set` 系メソッドは値を変更してしまいますし、メソッドが返すのは `Date` インスタンスではなく、エポック秒なので、必要に応じて新しいインスタンスを作りましょう。

```
const now = new Date();  
  
// 1 時間後  
const oneHourLater = new Date();  
oneHourLater.setHours(now.getHours() + 1);  
  
// 1 日後  
const oneDayLater = new Date();  
oneDayLater.setDate(now.getDate() + 1);  
  
// 1 ヶ月後  
const oneMonthLater = new Date();  
oneMonthLater.setMonth(now.getMonth() + 1);  
  
// 1 年後  
const oneYearLater = new Date();  
oneYearLater.setFullYear(now.getFullYear() + 1);
```

9.1.6 同一日時かどうかの比較

2 つの日時が同じ日かどうか確認したいことがあります。たとえば、チケットの日時が今日かどうか、といった比較です。JavaScript が提供しているのは、日時がセットになった `Date` 型のみです。他の言語だと、日付だけを扱うクラス、時間だけを扱うクラス、日時を扱うクラスを別に用意しているものもありますが、JavaScript は 1 つだけです。日付だけを扱いたいケースでも `Date` 型を使う必要があります。

`Date` 型で日付だけを扱う方法で簡単なのは時間部分をすべて 0 に正規化してしまう方法です。`setHours()` に 0 を 4 つ設定すると、時、分、秒、ミリ秒のすべてがゼロになります。また、この関数を実行するとエポック時刻

が帰ってくるので、これを比較するのがもっとも簡単でしょう。ただし、このメソッドはその日付を変更してしまうため、変更したくない場合は新しいインスタンスを作ってからこのメソッドを呼ぶと良いでしょう。

この 0 時は現在のタイムゾーンでの日時になります。もし、UTC で計算したい場合は、`setUTCHours()` を使います。

```
// 今日の 0 時 0 分 0 秒のエポック時刻
const today = (new Date()).setHours(0, 0, 0, 0)

// 比較したい日時
const someDate: Date

// 同じ日なら true
const isSameDay = (new Date(someDate)).setHours(0, 0, 0, 0) === today;
```

9.1.7 日時ではなく時間だけを扱う

前節では日付だけを扱いましたが、時間だけを扱いたいケースもあるでしょう。こちら、年月日を同一の日付、例えば、2000 年 1 月 1 日などの特定の日付に固定してしまえば扱えます。例えば、サーバーでは UTC の 0 時 0 分からの経過秒で扱うユースケースを考えてみます。例えば、毎日の会議が日本時間で 10 時であったとします。UTC で朝 1 時、日本時間の 10 時を表現するには、3600 になりますね。UTC の 2000 年 1 月 1 日 0 時のエポック秒は 946,684,800 になりますので、これに足して Date インスタンスを作れば良いことがわかります。

このクラスはローカルタイムゾーンの時間を計算してくれるので、`setHours()` で時間を取得すると、すでにローカルタイムゾーンの時間になります。

```
// 基準日で時間だけオフセットした Date インスタンス
const meetingTime = new Date((946684800 + 3600) * 1000); // JS の時間はミリ秒なので 1000 倍する
console.log(`${meetingTime.getHours()}:${meetingTime.getMinutes()}`);
// 10:00
```

今日のミーティングの日時情報を得るには、今日の日付を設定すれば OK です。

```
// 今日のミーティング時間を計算するには、今日の年月日を設定する
const now = new Date();
const todayMeeting = new Date(meetingTime);
todayMeeting.setFullYear(now.getFullYear(), now.getMonth(), now.getDate());
```


9.1.8 Date 型のタイムゾーンの制約

Date は「タイムゾーンが扱える」ということは何度か説明していますが、任意の日付で自由にタイムゾーンが扱えるわけではありません。標準の国際化の機能を使えば、ニューヨークの今の時間は？というのを数値で取得、みたいなことを計算する能力はあるのですが、それが使いやすいインタフェースを提供していません。一旦文字列にしてからパースするしかありません。

```
const now = new Date();
console.log(now.toLocaleString('en-US', { timeZone: 'America/Los_Angeles' }));
// '12/8/2020, 2:19:55 AM'
```

コンピュータでタイムゾーンで扱う名前は IANA のデータベースが一次情報になります。

- <https://www.iana.org/time-zones>

ただし、これは tar.gz などでの提供のみなので、一覧をみたい場合は Wikipedia などの方が見やすいでしょう。

- https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

Date のみで扱うのは、UTC と現在時刻ぐらいにしておいた方が無難です。最初に紹介した各種ライブラリなどは、タイムゾーンを扱う機能も持っているのも、そちらを利用の方が良いでしょう。ただし、標準の国際化機能を利用したもの以外、タイムゾーンはかなり大きなデータファイルを必要とするので、ビルドした JavaScript のファイルサイズは大きくなる可能性があります。

9.2 RegExp

正規表現のためのクラスです。TypeScript では組み込みで正規表現があります。正規表現は string と一緒に使うクラスで、ちょっと賢い検索を実現します。

正規表現はいくつかの要素を組み合わせた文字列のパターン（ルール）を組み合わせで作ります。このルールを活用することで、「電話番号にマッチする」といった単なる比較以上の比較が実現できます。かなり複雑なパターンも記述できますが、TypeScript においてはあまり活躍の場がありません。ユーザー入力のチェックや設定ファイルぐらいでしょう。

正規表現の評価は、入力文字列とパターンで行われます。文字列を探索し、パターンにマッチしているかどうかをみていきます。たいてい、複数の文字数の文字列を評価することになります。途中でマッチしない文字が出てきたらそこで評価終了となり、マッチしなかった、という結果が返ります。

正規表現はリテラルで記述できます。

```
const input = "03-1234-5678";

if (input.match(/\d{2,3}-\d{3,4}-\d{4}/)) {
  console.log("電話番号です");
}
```

9.2.1 使い方

正規表現は特定の目的を持って実行されます。検索であったり、ルールに従って分割だったり

- 文字列.match (正規表現)
- 文字列.matchAll (正規表現)
- 文字列.search (正規表現)
- 文字列.replace (正規表現, 置き換える文字列)
- 文字列.split (正規表現)

正規表現側にも

- 正規表現.exec (文字列)
- 正規表現.test (文字列)

9.2.2 パターンのルール

正規表現の構成要素は主にこの3つです。

- 文字種
- 繰り返し
- グループ化 (キャプチャ)

文字種は、2つの書き方があります。これは一文字を表します。文字クラスが柔軟な文字のルールが記述できます。このルールに従ったいずれかの文字が来たらマッチします。

- 普通の文字列 (a など)
- 文字クラス ([abc]、[a-zA-Z0-9]、[^x]、\s、. など)

[abc] は、このカッコの中のどれかの文字にマッチします。これだけあればどんなルールも記述できますが、アルファベット全部とかになると正規表現が随分と長くなってしまいます。短く書くためのルールがいくつかあります。[a-z] は、a から z の小文字のアルファベットすべてにマッチします。上記のサンプルの a-zA-Z0-9 は、すべての大文字小文字数字にマッチします。^をつけると、「これら以外の文字」と言うルールになります。また、\s (改行やタブなどのスペースにマッチ)、\w (英数字にマッチ)、\d (数字にマッチ) といったさらなる短縮系もあります。それぞれ、大文字にすると否定 (\S は改行タブスペース以外にマッチ) といったルールになります。また、. はすべての文字にマッチします。

文字ではないですが、先頭の^は文字列の先頭、最後の\$は末尾となります。/abc/は----abc-----にもマッチしますが、余計な文字が前後に付かないことを保証するには/^abc\$/とします。

文字のルールが記述できたので、それを並べれば文章にマッチします。例えば、`/abc/`は、`"abc"`の文字列にマッチします。これを柔軟にしていくなかでメタ文字が何種類か提供されています。こちらでも、文字クラスと同様に、冗長な書き方と、短縮系が提供されています。

- 前のルールを `n` 回繰り返す (`{n}`)
- 前のルールを `n` 回以上繰り返す (`{n, }`)
- 前のルールを `n~m` 回繰り返す (`{n, m}`)
- 前のルールがあってもなくても良い (?)
- 前のルールを 0 回以上繰り返す (*)
- 前のルールを 1 回以上繰り返す (+)

TypeScript の環境設定ではファイルの拡張子のルールを設定することがよくありますが、`.ts` でも `.tsx` でもマッチする書き方は `.tsx?` です。最後の一字を無視しても良い、というルールになります。

日本の郵便番号は数字 3 桁 + 数字 4 桁です。これをルールにすると次のようになります。

```
const postalCode = "141-0032";
if postalCode.match(/\d{3}-\d{4}/) {
  console.log("郵便番号です");
}
```

これはもちろん、次のように書いてもマッチしますが、なるべく抜け漏れのない短い、意図が伝わり安いパターン記述が良いでしょう。

- `/[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/` (上記と等価だが、長すぎる)
- `/\d+-\d+/` (ただし、これでは文字数が間違っても通ってしまうので NG)
- `/\w{3}-\w{4}/` (ただし、これでは数字以外に文字列もマッチしてしまう)
- `/.+ /` (すべての入力にマッチしてしまってテストにならない)

最後にグルーピングとキャプチャです。上記のルールだけでもマッチしているかどうかの条件にはだいたい使えます。しかし、もっと柔軟にしたいことがあります。例えば、固定電話の番号は同じ 10 桁でも、都道府県によってルールがいくつかあります。

- 2 桁-4 桁-4 桁
- 3 桁-3 桁-4 桁
- 4 桁-2 桁-4 桁
- 5 桁-1 桁-4 桁

正規表現ではこれまでのルールで作ったパターンを () でグループ化し、グループ単位で繰り返し (ルールは前述のものと同様) や、どちらかにマッチすれば OK (|) といったことが可能です。

上記の電話番号はそれぞれ、次のようにルール化できます。ついでに、先頭がゼロというのもパターンのルールに入れます。

- `/^0\d-\d{4}-\d{4}$/`
- `/^0\d{2}-\d{3}-\d{4}$/`
- `/^0\d{3}-\d{2}-\d{4}$/`
- `/^0\d{4}-\d-\d{4}$/`

これらをグループ化し、OR 化してあげれば、すべての電話番号にマッチするパターンが作れます。一見複雑に見えますが、グループと OR を分解すると解読しやすくなります。

```
``/(^0\d-\d{4}-\d{4}$)|(^0\d{2}-\d{3}-\d{4}$)|(^0\d{3}-\d{2}-\d{4}$)|(^0\d{4}-\d-\d{4}
↪$)/``
```

2 桁の一番上のものは、2 桁目は 3 か 4 か 5 かなので、正確には `/0[3-5]/` できると思います。正規表現だけでどこまでビジネスロジックを入れ込むかは設計次第です。正規表現だとマッチしたかどうかの 0/1 でしか判断できません。2 桁だが 09 みたいな変則番号が来たときにエラーを通知する場合は、2 桁で広くマッチさせておいて、次に桁をみるエラーチェックを行うなど、2 段構成にする必要があるでしょう。

9.2.3 キャプチャ

グループ化にはもう一つおまけがあります。`match()` の場合、どのカッコごとにマッチした結果を保存して返してくれます。

```
> const tel = /^(^0\d-\d{4}-\d{4}$)|(^0\d{2}-\d{3}-\d{4}$)|(^0\d{3}-\d{2}-\d{4}$)|(^0\d
↪{4}-\d-\d{4}$)/
> "042-234-1234".match(tel)
[
  '042-234-1234',
  undefined,
  '042-234-1234',
  undefined,
  undefined,
  index: 0,
  input: '042-234-1234',
  groups: undefined
]
```

レスポンスは一部フィールドを持った配列です。もしどれにもマッチしなければ `null` が返ります。

- 0 番目: 全体のマッチした結果
- n 番目: n 番目のグループのマッチ結果（マッチしなければ `undefined`）

- `index`: 入力文字列の何番目の文字列からマッチしたか（マッチするまでに何文字読み飛ばしたか）
- `input`: 正規表現と比較した入力値
- `groups`: 名前付きグループの場合、名前ごとのマッチ結果

グループはネストさせることもできますが、正規表現のパターンのスタートの位置で機械的にインデックスが割り振られます。グループは番号でアクセスもできますが、ECMAScript 2018 以降であれば名前をつけることもできます。電話番号は先頭から、それぞれ次のような分類になっています。

- 市外局番: Area Code
- 市内局番: Message Area
- 加入者番号: Subscriber Number

(パターン) を (?<名前>パターン) と書き換えると名前が付きます。

```
``/^ (?<AC>0\d{1}) - (?<MA>\d{4}) - (?<SA>\d{4}) $/``
```

これで、名前でマッチした結果にアクセスできます。

```
> const match = "01-2345-6789".match(/^ (?<AC>0\d{1}) - (?<MA>\d{4}) - (?<SA>\d{4}) $/)
> const { AC, MA, SA } = match.groups
> AC
'01'
> MA
'2345'
> SA
'6789'
```

9.3 JSON

JavaScript で外部の API のやりとりなどで一番使うのはこの JSON でしょう。サーバーから帰ってくる JSON 形式の文字列を、プログラム中で扱いやすい JavaScript や TypeScript のプリミティブなデータ型などに変換します。

基本は `parse()` と `stringify()` を呼ぶだけですなので使い方に迷うことはないでしょう。

9.3.1 JSON のパース

```
// a は any 型
const a = JSON.parse( '{"name": "John Cleese"}' );

// as で型情報を付与できる。
const p = JSON.parse( '{"name": "Terry Gilliam"}' ) as Person;
```

`parse()` は `any` 型になります。 `as` で何かしらの型にキャストする必要がありますが、このパースは当然実行時に行われます。 `as` はコンパイル時の型チェックのためのものなので、実際に実行時にどのような型が来るかは推測でしかありません。この JSON のパース周りは、コンパイルは通ったのに、想定した型と違う情報がきたためにエラーになる、ということが一番起きやすいポイントです。 `as` は一見その型と同等であると保証しているように見せてしまいますが、要素の有無のチェックなどは動的な言語を扱っている意識を忘れないで行いましょう。

型定義をする必要がないかと言えば、明らかなスペルチェックは見つけられますし、補完もされるので、可能なら定義しておく方が良いでしょう。

SyntaxError 例外

この関数は、JSON の文法違反があると `SyntaxError` 例外を投げます。TypeScript で例外を扱うことは稀ですが、キャッチしないと何か操作したのに処理が行われていないように見えるなどの不具合になります。大々的にエラーダイアログが出たりはしないので気づきにくかったりしますが、コンソールには出力されていたりします。パースするときには `try` でくくって、エラーが出たときにはダミーの値を入れるなり、自分でエラーダイアログを表示するなり、対処しましょう。

```
let person: Person
try {
  person = JSON.parse(input);
} catch (e: unknown) {
  // fallback
  person = { name: "Eric Idle" };
}
```

この関数は `fetch()` の中でも使われています。例外の発生についてもまったく同じです。

```
const res = await fetch("/api/person");
// 本当は res.ok で通信が成功したかチェックが必要！
person = { name: "Michael Palin" };
```

HTTP の API の場合、エラーがあると、Forbidden などのステータスコードの文字列がレスポンスとして帰ってくることもあり、これを JSON にパースしようとする次のような例外が発生します。

```
Uncaught SyntaxError: Unexpected token F in JSON at position 0
```

ほとんどの場合はステータスコードのチェックで防げますが、`try` 文を使うとさらに安心です。

JSON とコメント

JSON の厳格な文法 ([json.org](https://www.json.org/)) にはコメントはありません。しかし、入れたくなることがあります。

TypeScript の設定ファイルの `tsconfig.json` や、VSCode の設定ファイルなどはコメントが入れられます。前者は TypeScript パーサーを流用したパーサーを使っていますので、TypeScript と同じコメントが利用できます。後者は JSON with Comments(.jsonc) というモードを持っています。標準の `JSON.parse()` は仕様に従っているの
でコメントがあるとエラーになります。次のようなライブラリを使う必要があるでしょう。

- `strip-json-comments`

設定ファイルとして使って読み手が自分自身だけならいいのですが、サーバーなどとのデータ交換用に JSON を使う場合、読み手がコメントを無視して読んでくれないかぎり
はコメントを使うべきではありません。その他の方法としては、JSON Schema の仕様にある `$comment` キーを使うという妥協案もあります。

```
{
  "$comment": "コメントです",
  "location": "鳥貴族"
}
```

9.3.2 文字列化

文字列化は JavaScript のオブジェクト、配列、数値、文字列、`boolean` 型などの値を渡すと、それを安全に伝送できる文字列にしてくれます。

リスト 3 JSON 形式に文字列化

```
// b は文字列
const b = JSON.stringify({person: "Graham Chapman"})
// '{"person": "Graham Chapman"}'
```

stringify() には 2 つ追加の引数があります。1 つは置換関数、もう 1 つはインデントです。

このうち置換関数は function(key: any, value: any): any な関数で、キーと値を見て、実際に出力する値を決めますが、あまり使い勝手の良いものではありません。階層があったり、配列で同型のオブジェクトがあったりして、仮に同名のキーがあっても、渡される情報だけではどちらの値か区別できなかったりします。bigint などの変換できない型の場合は replacer が実行される前にエラーになってしまうため、この関数で出力できるように文字列にするとといった使い方もできません。事前に出力可能なオブジェクト・配列・プリミティブだけのきれいな情報に変換しておくべきです。そのため、忘れてしまっても構いません。

インデント

デフォルトではインデントがなく、文字数最小で出力されます。インデントに数値、あるいは " " といった文字列を渡すことでインデントが行われて見やすくなります。ただし、Node.js やブラウザの console.log() の場合はインデントを設定しなくても見やすく表示してくれるため、整形してファイル出力したい場合以外は使う必要はないと思います。

リスト 4 インデント

```
const montyPython = {
  members: [
    "John Cleese",
    "Terry Gilliam",
    "Eric Idle",
    "Michael Palin",
    "Graham Chapman",
    "Terry Jones"
  ],
};

console.log(JSON.stringify(montyPython));
// {"members":["John Cleese","Terry Gilliam","Eric Idle","Michael Palin","Graham
↳Chapman","Terry Jones"]}

console.log(JSON.stringify(montyPython, null, 2));
// {
//   "members": [
//     "John Cleese",
//     "Terry Gilliam",
//     "Eric Idle",
//     "Michael Palin",
```

(次のページに続く)

(前のページからの続き)

```
//      "Graham Chapman",  
//      "Terry Jones"  
//    ]  
//  }
```

JSON とデータロス

JSON は言語をまたいで使われるデータのシリアル化のための仕組みです。元は JavaScript のオブジェクト表現をフォーマットにしたものではありませんが（JSON は作成されたのではなく、発見されたと言われています）、いくつか TypeScript と違うところもあります。

JSON は単純な木構造であり、TypeScript のメモリ上の表現のすべてを表現できるわけではありません。例えば、親が子を、子が親を参照しているような循環構造の場合、うまく文字列化できず、エラーになります。事前に変換する関数を使って、子から親方向の参照を切った新しいオブジェクト階層を作るなどして単方向の参照になるようにします。

リスト 5 循環参照があると TypeError

```
const person = {name: "Terry Jones"};  
const group = {name: "Pythons", member: [person]};  
person.group = group; // お互いに参照しあっているため、循環参照になる  
  
JSON.stringify(group)  
// Uncaught TypeError: Converting circular structure to JSON
```

また、JSON が扱えるデータ型はそれほど多くありません。ネイティブで扱える型は以下の 6 つです。他のものはうまく文字列にならなかったり、なったとしても再度パースしたときにもとの型が復元できないことがあります。

- オブジェクト
- 配列
- 文字列
- 数値
- boolean 型
- null

例えば undefined の場合はそのキーがなかったことになります。クラスの場合はメンバーフィールドのみのオブジェクトになります。一応データとしては復元できますが、戻すときには単なるオブジェクトで、クラスのインスタンスではなくなります。各クラスに、オブジェクトからインスタンスを復元するファクトリーメソッドを用意してあげる必要があるでしょう。日付は文字列になります。これも、事前に `valueOf()` で数値化しても良いでしょう。Map() などは値が完全に失われたインスタンスになるため、注意が必要です。

リスト 6 クラス、日付

```
> class C { constructor() { this.a = 1; this.b = "hello"; } }
> const i = new C();
> JSON.stringify(i);
// '{"a":1,"b":"hello"}'

> JSON.stringify(new Date())
// '"2020-09-15T14:41:37.173Z"'

> const m = new Map([[1, 2], [2, 4], [3, 8]])
// Map(3) { 1 => 2, 2 => 4, 3 => 8 }
> JSON.stringify(m);
// '{}'
```

9.4 URL と URLSearchParams

正規表現と JSON を紹介しました。正規表現はユーザーの入力のチェックにたまに使います。JSON はサーバーとのデータのやりとりのフォーマットとしてよく使います。もう一つ、サーバーとのやり取りで利用するのが URL や URLSearchParams です。

サーバーにリクエストを送るときにデータを詰め込む箱としては次の 3 つがあります。

- パス
- メソッド
- ボディ

このうち、主にパスに使うのが本節で紹介する URL と URLSearchParams です。TypeScript の元になっている ECMAScript には含まれないものですが、ブラウザには備わっていますし、Node.js にも追加されました。Node.js はもともと別の URL 解析関数を持っていましたが、そちらは非推奨になり、現在はこちらのブラウザ互換のクラスが推奨になっています。

9.4.1 URL

使い方は簡単で、コンストラクタにパスを入れると、パスのそれぞれの構成要素（プロトコルやホスト名、パス）などに分解します。

```
> u = new URL("https://developer.mozilla.org/en-US/docs/Web/API/URL#Methods")
URL {
  href: 'https://developer.mozilla.org/en-US/docs/Web/API/URL#Methods',
  origin: 'https://developer.mozilla.org',
```

(次のページに続く)

(前のページからの続き)

```

protocol: 'https:',
username: '',
password: '',
host: 'developer.mozilla.org',
hostname: 'developer.mozilla.org',
port: '',
pathname: '/en-US/docs/Web/API/URL',
search: '',
searchParams: URLSearchParams {},
hash: '#Methods'
}

```

一部を書き換えて `toString()` を呼ぶことで URL が作成できます。

```

> u.pathname = "/en-US/docs/Web/API/URL/createObjectURL"
> u.toString()
'https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL#Methods'

```

この程度であればテンプレート文字列を使うなどしても簡単ですが、この手の文字列を組み立てるクラスは積極的に使うべきです。SQL インジェクションなどのセキュリティホールは自分で命令を組み立てたときに、エスケープし忘れて想定しない命令が差し込まれてしまうことで発生します。URL も、何かを呼び出すときの鍵ですので、想定外の URL ができてしまうのは問題があります。

URL にはパスの最後にクエリーパラメータが付きますが、そこを `s` よりするのが `URLSearchParams` です。上記の例ではクエリーがなかったので空となっていますが、URL にクエリーがあれば、ここの部分にパラメータが保存されます。

`caniuse.com` で検索した URL を渡すと次のようになります。

```

> c = new URL("https://caniuse.com/?search=url%20search%20params")
URL {
  href: 'https://caniuse.com/?search=url%20search%20params',
  origin: 'https://caniuse.com',
  searchParams: URLSearchParams { 'search' => 'url search params' },
}

```

9.4.2 URLSearchParams

`URLSearchParams` は URL の一部としても利用されますが、クエリー部分だけを渡すことで、`URLSearchParams` を単独で利用できます。

```

> q = new URLSearchParams("?search=url%20search%20params")
URLSearchParams { 'search' => 'url search params' }

```

このクラスは、Map に似ていますが、同一のキーに複数の値を持たせることができます。

メソッド	説明
<code>get(key: string): string</code>	値を取得。複数ある場合は先頭のみ。
<code>getAll(key: string): string[]</code>	値を配列で取得。
<code>append(key: string, value: string)</code>	値を設定（複数共存可能）
<code>set(key: value, value: string)</code>	値を設定（上書きして1つだけ残す）
<code>delete(key: value)</code>	値を削除
<code>has(key: value): boolean</code>	キーが存在するか確認
<code>toString(): string</code>	文字列を生成

イテレータプロトコルをサポートしており、キーだけ (`keys()`)、値だけ (`values()`)、キーと値のペア (本体を渡す) の 3 通りの `for of` ループが可能です。

`URLSearchParams` は基本的に URL のクエリーパラメータ部分の組み立てに使用しますが、HTML のフォームで、デフォルトの `enctype` (`application/x-www-form-urlencoded`) と互換性があります。

HTML のフォームを使って送信する代わりに `fetch()` を使って TypeScript でリクエストを送るときに組み立てにも使えますし、Node.js などですerverを実装した場合に、ポストされたボディをパースするのにも利用できます。こちらも URL と同様に、セキュリティホールを作らないためにも、なるべく利用するようにしましょう。

なお、この `URLSearchParams` がキーと値をそれぞれエスケープするのと同じロジックは `encodeURIComponent()` と `decodeURIComponent()` という関数でも利用できます。

厳密には URL のクエリーとフォームは少し異なり、フォームとクエリーの場合はスペースはプラス (+) で、パスの一部は %20 だったりしますが、どちらも入れ替えてデコードすると正しく戻するため、そこまで厳密にみなくても大丈夫です。`URLSearchParams` はプラスにしますが、`encodeURIComponent()` は %20 にします。

9.5 時間のための関数

`setTimeout()`

`setInterval()`

`requestAnimationFrame()`

第 10 章

クラス

10.1 JavaScript と Java 風オブジェクト指向文法の歴史

クラスはプログラミング言語によってはとても重要な位置付けにあります。たとえば、Java はすべての要素がクラスに属し、そのクラスの組み合わせでプログラムを作成していきます。クラスの利用方法を学び、クラスの実装方法を学び、よりよいクラスの設計方法を知ることが Java においてはとても重要です。また、20 年ほど前は Java はプログラミングのパラダイムにおいては最先端であり、クラスを元にした設計手法、デザインパターン、アジャイルソフトウェア開発のプログラミング系のプラクティス（テスト駆動開発やリファクタリング）を通じて、多くのプログラミング言語に Java 流の設計が輸出されていきました。クラスを使いこなすことで、次のことが実現可能になるとさかんに喧伝されていました。

- 大規模なコード
- 再利用性の高いコード

例えば Python や Ruby などの今は古参扱いのプログラミング言語の多くも、当時は機能拡張を積極的に行っており Java やその周辺のベストプラクティスに影響を受けたと思われるライブラリなどがいくつか組み込まれています。クラスを持ってその先端の設計技法が利用できることは、当時の一級言語のステータスでした。

JavaScript はクラスをダイレクトに表現する文法はなかったものの、昔は関数と prototype という属性をいじくり回してクラスを表現していました。正確には処理系的にはクラスではないのですが、コードのユーザー視点では他の言語のクラスと同等なのでここではこの当時の記法で作るものもクラスとして扱います。とはいえ、それでも JavaScript はクラスがないことを理由に「大規模開発に向かないおもちゃ言語」と言われることもありました。

その欠点をカバーして Java などのような書き味を求めて、ひと昔前の JavaScript 界限では、この prototype の仕組みをラップした自前の extends 関数みたいなのを作ってクラスっぽいことを表現しようという一派も一時期いました。また、5 年ほど前までは、クラスを使うために CoffeeScript に救いを求める人も多数いました。今の JavaScript と TypeScript では、言語の標準機能として良い書き方が提供されています。

クラスをようやくサポートしたものの、近年ではクラスを使わない書き方、関数型言語のエッセンスを取り入れた書き方も流行ってきています。大規模といえばオブジェクト指向でクラス、という短絡的な言い方をする人はもはや絶滅危惧種です。

しかし、大規模なアプリケーションの部品としてではなく、末端の部品としては便利です。みなさんも、文字列や配列などの組み込み型を通じて、自然とオブジェクト指向には触れています。あのような部品を自分で作るための手段です。関数型のスタイルとも組み合わせて利用できますし、書き方を学んでおきましょう。

10.2 用語の整理

オブジェクト指向言語は、それぞれの言語ごとに使っている言葉が違うので、それを一旦整理します。本書では次の用語で呼びます。TypeScript の公式ドキュメント準拠です。

- クラス (class)

他の言語のクラスと一緒にです。ES2015 以前の JavaScript にはかつてなかったものです（似たようなものはありました）。

- インスタンス (instance)

クラスを元にして new を呼び出して作ったオブジェクトです。

- メソッド (method)

他の言語では、メンバー関数と呼んだり、フィールドと呼んでいたりします。名前を持ち、ロジックを書く場所です。自分が属しているインスタンスのプロパティやメソッドにアクセスできます。

- プロパティ (property)

他の言語では、メンバー変数と呼んだり、フィールドと呼んでいたりします。名前を持ち、指定された型のデータを保持します。インスタンスごとに別の名前空間を持ちます。

10.3 基本のクラス宣言

最初はコンストラクタ関数を作り、その prototype 属性を操作してクラスのようなものを作っていました。今の書き方は次のような class を使った書き方になり、他の言語を使っている人からも親しみやすくなりました。

なお、JavaScript では不要ですが、TypeScript ではプロパティの定義をクラス宣言の中で行う必要があります。定義していないプロパティアクセスはエラーになります。

リスト 1 クラスの表現

```
// 新しいクラス表現
class SmallAnimal {
  // プロパティは名前: 型
  animaltype: string;

  // コストラクタ (省略可能)
  constructor() {
```

(次のページに続く)

(前のページからの続き)

```

        this.animaltype = "ポメラニアン";
    }

    say() {
        console.log(`$${this.animaltype}だけどMSの中に永らく居たBOM信者の全身の毛をむしりたい`);
    }
}

const smallAnimal = new SmallAnimal();
smallAnimal.say();
// ポメラニアンだけどMSの中に永らく居たBOM信者の全身の毛をむしりたい

```

以前の書き方は次の通りです。

リスト 2 旧来のクラスのようなものの表現

```

// 古いクラスの表現
// 関数だけどコンストラクタ
function SmallAnimal() {
    this.animaltype = "ポメラニアン";
}
// こうやって継承
SmallAnimal.prototype = new Parent();

// こうやってメソッド
SmallAnimal.prototype.say = function() {
    console.log(this.animalType + "だけどMSの中に永らく居たBOM信者の全身の毛をむしりたい");
};

```

10.4 アクセス制御 (public/protected/private)

TypeScript には C++ や Java のような `private` と `protected`、`public` 装飾子があります。メンバー定義の時の `public` 装飾子は基本的につけてもつけなくても結果は変わりませんので、コメントのようなものです。権限の考え方も同じで、`private` は定義があるクラス以外からの操作を禁止、`protected` は定義のあるクラスと子クラス以外からの操作を禁止、`public` は内外問わず、すべての操作を許可、です。オブジェクト指向言語だと Ruby がやや特殊で、`private` は「同一インスタンスからの操作のみを許可」ですが、これとは違う動作になります。

リスト 3 アクセス制御

```

// 小型犬
class SmallDog {
    // 小型犬は宝物を秘密の場所に埋める
    private secretPlace: string;
}

```

(次のページに続く)

(前のページからの続き)

```

    dig(): string {
        return this.secretPlace;
    }

    // 埋める
    bury(treasure: string) {
        this.secretPlace = treasure;
    }
}

const miniatureDachshund = new SmallDog();
// 埋めた
miniatureDachshund.bury("骨");

// 秘密の場所を知っているのは小型犬のみ
// アクセスするとエラー
// error TS2341: Property 'secretPlace' is private and
// only accessible within class 'SmallDog'.
miniatureDachshund.secretPlace;

// 掘り出した
console.log(miniatureDachshund.dig()); // 骨

```

古くは JavaScript ではさまざまなトリックを使って private 宣言を再現しようといろいろなテクニックが作られました。もはや使わない、と前章で紹介した即時実行関数も、すべて private のようなものを実現するためのものでした。それ以外だと、簡易的に `_` をメンバー名の前につけて「仕組み上はアクセスできるけど、使わないでね」とコーディング規約でカバーする方法もありました。

また `protected` は継承して使うことを前提としたスコープですが、Java はともかく TypeScript では階層が深くなる継承をすることはまずないので、使うことはないでしょう。

10.5 コンストラクタの引数を使ってプロパティを宣言

TypeScript 固有の書き方になりますが、コンストラクタ関数にアクセス制御の装飾子をつけると、それがそのままプロパティになります。コンストラクタの引数をそのまま同名のプロパティに代入します。

リスト 4 プロパティ定義をコンストラクタ変数に

```

// 小型犬
class SmallDog {
    constructor(private secretPlace: string) {
    }

    dig(): string {
        return this.secretPlace;
    }
}

```

(次のページに続く)

(前のページからの続き)

```

    }

    // 埋める
    bury(treasure: string) {
        this.secretPlace = treasure;
    }
}

```

これはコンストラクターの引数になったので、初期化時に渡してあげると初期化が完了します。

```

const miniatureDachshund = new SmallDog("フリスビー");

// 掘り出した
console.log(miniatureDachshund.dig()); // フリスビー

```

10.6 static メンバー

オブジェクトの要素はみな、基本的に new をして作られるインスタンスごとにデータを保持します。メソッドも this は現在実行中のインスタンスを指します。static をつけたプロパティは、インスタンスではなくてクラスという 1 つだけの要素に保存されます。static メソッドも、インスタンスではなくてクラス側に属します。

リスト 5 プロパティ定義をコンストラクタ変数に

```

class StaticSample {
    // 静的なプロパティ
    static staticVariable: number;
    // 通常のプロパティ
    variable: number;

    // 静的なメソッド
    static classMethod() {
        // 静的なメソッドから静的プロパティは ``this`` もしくは、 ``クラス名`` で参照可能
        console.log(this.staticVariable);
        console.log(StaticSample.staticVariable);
        // 通常のプロパティは参照不可
        console.log(this.variable);
        // error TS2339: Property 'variable' does not exist on
        //      type 'typeof StaticSample'.
    }

    method() {
        // 通常メソッドから通常のプロパティは ``this`` で参照可能
        console.log(this.variable);
        // 通常メソッドから静的プロパティは ``クラス名`` で参照可能
        console.log(StaticSample.staticVariable);
    }
}

```

(次のページに続く)

(前のページからの続き)

```
// 通常のメソッドから静的なプロパティを ``this`` では参照不可
console.log(this.staticVariable);
// error TS2576: Property 'staticVariable' is a static
//      member of type 'StaticSample'
}
}
```

Java と違って、すべての要素をクラスで包む必要はないため、static メンバーを使わずにふつうの関数や変数を使って実装することもできます。静的メソッドが便利そうな唯一のケースとしては、インスタンスを作る特別なファクトリーメソッドを実装するぐらいでしょうか。次のクラスは図形の点を表現するクラスですが、polar() メソッドは極座標を使って作成するファクトリーメソッドになっています。

```
class Point {
  // 通常のコンストラクタ
  constructor(public x: number, public y: number) {}

  // 極座標のファクトリーメソッド
  static polar(length: number, angle: number): Point {
    return new Point(
      length * Math.cos(angle),
      length * Math.sin(angle));
  }
}

console.log(new Point(10, 20));
console.log(Point.polar(10, Math.PI * 0.25));
```

静的なプロパティを使いすぎると、複製できないクラスになってしまい、テストなどがしにくくなります。あまり多用することはないでしょう。

10.7 インスタンスクラスフィールド

JavaScript ではまだ Stage 3 の機能ですが、TypeScript ですでに使える文法として導入されているがインスタンスクラスフィールド^{*1}^{*2} という文法です。この提案にはいくつかの文法が含まれていますが、public メンバーのみをここで紹介します。

イベントハンドラにメソッドを渡す時は、メソッド単体を渡すと、オブジェクト引き剥がされてしまって this が行方不明になってしまうため、これまでは bind() を使って回避していたことはすでに紹介しました。インスタンスクラスフィールドを使うと、クラス宣言の中にプロパティ宣言を書くことができ、オブジェクトがインスタンス化されるときに設定されます。このときにアロー関数が利用できるため、イベントハンドラにメソッドをそのまま渡しても問題なく動作するようになります。

^{*1} <https://github.com/tc39/proposal-class-fields>

^{*2} Babel では @babel/plugin-proposal-class-properties プラグインを導入すると使えます

アロー関数を単体で使っても便利ですが、React の `render()` の中で使うと、表示のたびに別の関数オブジェクトが作られたと判断されて、表示のキャッシュがうまく行われずにパフォーマンスが悪化する欠点があります^{*3}。インスタンスクラスフィールドとして定義すると、コンストラクタの中で一回だけ設定されるだけなので、この問題を避けることができます。

```
// 新: インスタンスクラスフィールドを使う場合
class SmallAnimal {
  // プロパティを作成
  fav = "小田原";
  // メソッドを作成
  say = () => {
    console.log(`私は${this.fav}が大好きです`);
  };
}
```

以前は `bind()` を使ってコンストラクタの中で設定していました。インスタンスクラスフィールドもコンストラクタ実行のときに実行されるので、実行結果は変わりません。

```
// 旧: bindを使う場合
class SmallAnimal {
  constructor() {
    this._fav = "小春日";
    this.say = this.say.bind(this);
  }

  say() {
    console.log(`私は${this._fav}が大好きです`);
  };
}
```

注釈: ECMAScript 側のインスタンスクラスフィールドの仕様では `private` の定義は `private` キーワードではなくて `#` を名前の前につける記法が提案されています。

10.8 読み込み専用の変数 (`readonly`)

変数には `const` がありましたが、プロパティにも `readonly` があります。 `readonly` を付与したプロパティは、プロパティ定義時および、コンストラクタの中身でのみ書き換えることができます。

```
class SimLockPhone {
  readonly carrier: string;
  constructor(carrier: string) {
```

(次のページに続く)

^{*3} <https://medium.freecodecamp.org/why-arrow-functions-and-bind-in-reacts-render-are-problematic-f1c08b060e36>

(前のページからの続き)

```
    this.carrier = carrier;
  }
}

// キャリア変更できない!
const myPhone = new SimLockPhone("Docomo");
myPhone.carrier = "au";
// error TS2540: Cannot assign to 'carrier' because it is a read-only property.
```

なお、通常のプロパティ定義以外にも、コンストラクタを使ったプロパティ定義、インスタンスクラスフィールドの定義で使うことができます。また、アクセス制御と一緒に使う場合は、`readonly` をあとにしてください。

```
class BankAccount {
  constructor(private readonly accountNumber) {
  }
}
```

10.9 メンバー定義方法のまとめ

外からプロパティ、メソッドに見えるものの定義の種類がたくさんありました。それぞれ、メリットがありますので、用途に応じて使い分けると良いでしょう。また既存のコードを読むときに、メンバーの定義のコードを確認する場合はこれのどの方法で定義されているのかを確認する必要があります。

これ以外にも、アクセッサがあります。これについては [クラス上級編](#) で紹介します。

表1 メンバーの定義方法

サンプル	メソッド	変数	JS 互換	メリット
<pre>// プロパティ secretPlace: string; // メンバーメソッド dig(): string { return this.secretPlace; }</pre>	○	○	○	一番シンプルで、継承やインタフェース機能との相性が良い。
<pre>// コンストラクタ引数 constructor(private ↵ secretPlace: string);</pre>		○		コンストラクタで外から定義する口とメンバーの宣言が1箇所で済む。初期値の設定が可能
<pre>// インスタンスクラスフィールド private secretPlace = "フリスビー";</pre>		○	△	初期値の設定が可能で、右辺から型が明確にわかる場合は型宣言を省略できる。アロー関数を代入することでbind()を使わずに、イベントハンドラに安全に渡せるメソッドが定義できる。

10.10 継承/インタフェース実装宣言

作られたクラスを元に機能拡張する方法がいくつかあります。そのうちの1つが継承です。

```
class SmallAnimal {
  eat() {
    console.log("中本を食べに行きました");
  }
}

class Pomeranian extends SmallAnimal {
  eat() {
    console.log("シュークリームを食べに行きました");
  }
}
```

もう1つ、インタフェースについては前章で説明しました。前章ではオブジェクトの要素の型定義として紹介しましたが、クラスとも連携します。むしろ Java で導入された経緯を考えると、こちらの用途の方が出自が先でしょう。

```
interface Animal {
  eat();
}

class SmallAnimal implements Animal {
  eat() {
    console.log("中本を食べに行きました");
  }
}
```

インタフェースは、クラスが実装すべきメソッドやプロパティを定義することができ、足りないメソッドなどがあるとエラーが出力されます。

```
// インタフェースで定義されたメソッドを実装しなかった
class SmallAnimal implements Animal {
}
// error TS2420: Class 'SmallAnimal' incorrectly implements interface 'Animal'.
//   Property 'eat' is missing in type 'SmallAnimal' but required in type 'Animal'.
```

今、この `eat()` メソッドには返り値が定義されていません。もしコンパイルオプションが `compilerOptions.noImplicitAny` の場合、ここでエラーが発生します。

リスト 6 インタフェースの返り値の型を省略すると・・・

```
interface Animal {
  eat();
}
// error TS7010: 'eat', which lacks return-type annotation,
//   implicitly has an 'any' return type.
```

明示的に `void` をつけたり、型情報をつけるとエラーは解消されます。

リスト 7 戻り値を返さない関数には void をつける

```
interface Animal {  
  eat(): void;  
}
```

関数のところの型定義で紹介したように、TypeScript は実際のコードの情報を元に、ソースコードを解析して戻り値の型を推測します。しかし、このインタフェースには実装がないため、推測ができず、常に any（なにかを返す）という型になってしまいます。これは型チェックを厳密に行っていくには穴が空きすぎてしまいエディタの補助が聞かなくなってしまうので開発効率向上が得にくくなります。noImplicitAny というオプションを使うとこの穴を塞げます。そのため、「何も返さない」という型も含め、手動で型をつける必要があります。

10.11 クラスとインタフェースの違い・使い分け

クラスとインタフェースは宣言は似ています。

違いがある点は以下の通りです。

- クラスをもとに new を使ってインスタンスを作ることができるが、インタフェースはできない
- インタフェースはインスタンスが作れないので、コンストラクタを定義できない
- インタフェースは public メンバーしか定義できないが、クラスは他のアクセス制御も可能

継承とかオブジェクト指向設計とか方法論とかメソッドはメッセージで云々とか語り出すと大抵炎上するのがオブジェクト指向とかクラスの説明の難しいところです。これらの機能は、言語の文化とか、他の代替文法の有無とかで使われ方が大きく変わってきます。

TypeScript 界限では、Angular などのフレームワークではインタフェースが多用されています。ユーザーが実装するコンポーネントなどのクラスにおいて、Angular が提供するサービスを受けるためのメソッドの形式が決まっています。実装部分の中身をライブラリユーザーが実装するといった使われ方をしています。OnInit を implements すると、初期化時に呼び出されるといった具合です。

継承が必要となるのは実装も提供する必要がある場合ですが、コードが追いかけていくと、拡張性のあるクラス設計が難しいとこともあり、引き継ぐべきメソッドが大量にあるクラス以外で積極的に使うケースはあまり多くないかもしれません。

しかし、TypeScript は JavaScript エコシステムと密接に関わっており、JavaScript の世界にはインタフェースを提供することはできず、実装の保証をする機能が確実に動くとは限りません。TypeScript のように、フレームワーク側も TypeScript で、実装コードも TypeScript というケースでなければ利用しにくいことが多々あります。特に、ライブラリ側が JavaScript で実装されている場合はクラスを使って継承、という使い方になります。

10.12 デコレータ

これも Stage 2 の機能^{*4}ですが、これもすでに多くのライブラリやフレームワークで利用されています。TypeScript では `tsconfig.json` の `compilerOptions.experimentalDecorators` に `true` 設定すると使えます。使い方から内部の動きまで Python 2.5 で導入されたデコレータと似ています。決まった引数とレスポンスを持つ関数を作り、`@` の記号をつけて、クラスなどの前に付与すると、宣言が完了したオブジェクトなどが引数に入ってこの関数が呼ばれます。他の言語でアトリビュートと呼ばれる機能と似ていますが、動的言語なので型情報の追加情報として設定されるのではなく、関数を通じてそれが付与されている対象のクラスやメソッド、属性を受け取り、それを加工する、変更する、記録するといった動作をします。たとえば、ウェブアプリケーションで URL とメソッドのマッピングをデコレータで宣言したり、関数実行時にログを出すようにする、権限チェックやバリデーションを追加する、メソッドを追加するなど、用途はかなり広いです。また、複数のデコレータを設定したりもできます。

次のコードは引数のないクラスデコレータの例です。クラスに付与するもの、属性に付与するもの、それぞれ引数を持つものと持たないものがあるので、書き方が 4 通りありますが、詳細は割愛します。

リスト 8 デコレータでクラスにメソッドを追加する

```
function StrongZero(target) {
  target.prototype.drink = function() {
    console.log("ストロングゼロを飲んだ");
  };
  return target;
}

@StrongZero
class SmallAnimal {
}

const sa = new SmallAnimal();
sa.drink();
```

10.13 まとめ

クラスにまつわる数々の機能を取り上げて来ました。昔の JavaScript をやっていたプログラマーから見ると、一番変化と進歩を感じるころがこのクラスでしょう。一般的なクラスの機能を備えた上で、型チェックも行われ、さらにデコレータなど追加機能なども含まれました。TypeScript の場合は、エディタによるコード補完の正答率が大幅に上がったりしてリターンが大きいので、生産性の高まりを感じられるでしょう。

いろいろと機能は多いですが、TypeScript では、あまりクラスの細かい機能を多用するコーディングは行われていません。そのため、本章で取り上げた機能のうち、使わない機能も多いはずで。ちょっとしたロジックが書ける（バリデーションなど）構造体、といった感じで使われることがほとんどでしょう。最重要なところをピックアップするとしたら次のあたりです。

^{*4} Babel では `@babel/plugin-proposal-decorators` プラグインが必要です。

- 基本のクラス宣言
- アクセス制御（`public/private`）
- インスタンスクラスフィールド
- インタフェース実装宣言

次のものは覚えておいても損はないでしょう。

- `static` メンバー
- コンストラクタの引数を使ってプロパティを宣言
- 読み込み専用の変数（`readonly`）

次の機能はライブラリを提供する側が覚えておくとおしゃれな機能です。

- デコレータ

次の機能を TypeScript で駆使するようになったら警戒しましょう。まず、2 段、3 段、4 段と続くような深い継承になるようなコードを書くことはないでしょう。`private` はともかく継承を前提とする `protected`、抽象クラスを多用するような複雑なクラス設計がでてきたら、アプリケーションコードレベルではほぼ間違いだと思います。もしかしたら、DOM に匹敵するような大規模なクラスライブラリを作るのであれば、抽象クラスだとか `protected` も活躍するかもしれませんが、ほぼ稀でしょう。せいぜいインタフェースを定義して、特定のメソッドを持っていたら仲間とみなす、ぐらいのダックタイピングとクラス指向の中間ぐらいが TypeScript のスイートスポットだと思います。

- アクセス制御（`protected`）
- 継承

アプリケーション開発者は使わないが、ライブラリ・フレームワーク実装者は使うかもしれない機能は、上級編として、[クラス上級編](#) の章で紹介します。次の要素について紹介します。

- アクセッサ
- 抽象クラス

第 11 章

非同期処理

JavaScript のエコシステムは伝統的にはスレッドを使わない計算モデルを使い、その効率をあげる方向で進化してきました。例えば、スリープのような、実行を行の途中で止めるような処理は基本的に持っていませんでした。10 秒間待つ、というタスクがあった場合には、10 秒後に実行される関数を登録する、といった具合の処理が提供され、その場で「10 秒止める」という処理を書く機能は提供されませんでした。

JavaScript は伝統的に、ホストとなる環境（ブラウザ）の中で実行される、アプリケーション言語として使われることが多く、ホスト側のアプリケーションから見て、長時間ブロックされるなどの行儀の悪い動きをすることが忌避されてきたからではないかと思います。そのせいかどうかわかりませんが、他の言語とは多少異なる進化を遂げてきました。

ブラウザでは、数々の HTML 側のインタラクション、あるいはタイマーなどのイベントに対して、あらかじめ登録しておいたイベントハンドラの関数が呼ばれる、というモデルを採用しています。JavaScript がメインの処理系となる Node.js でも、OS が行う、時間のかかる処理を受けるイベントループがあり、OS 側の待ち処理に対してコールバック関数をあらかじめ登録しておきます。そして、結果の準備ができたなら、それが呼ばれるというモデルです。

ES2015 以降、このコーディングスタイルにも手が入り、土台の仕組みはコールバックではありますが、多数の非同期を効率よく扱う方法が整備されてきました。現在、見かける非同期処理の書き方は大きく 3 種類あります。

- コールバック
- Promise
- async / await

本章ではそれらを紹介していきます。なお、非同期処理の例外処理については、例外処理の章で扱います。

11.1 非同期とは何か

JavaScript の処理系には、現在のシステムの UI を担うレイヤーとしてかなりの開発資金が投入されてきました。ブラウザ戦争と呼ばれる時期には、各ブラウザが競うように JavaScript やウェブブラウザの画面描画の速度向上を喧伝し、他社製のブラウザよりも優れていると比較のベンチマークを出していたりしました。その結果としては、スクリプト言語としては JavaScript はトップクラスの速度になりました。Just In Time コンパイラという実行時の最適化が効くと、コンパイル言語に匹敵する速度を出すことすらあります。

CPU 速度が問題になることはあまりないとはいえ、コードで処理するタスクの中には長い時間の待ちを生じさせるものがいくつかあります。例えば、タイマーなどもそうですし、外部のサーバーやデータベースへのネットワークアクセス、ローカルのファイルの読み書きなどは往復でミリ秒、場合によっては秒に近い遅延を生じさせます。JavaScript は、そのような時間のかかる処理は基本的に「非同期」という仕組みで処理を行います。タイマー呼び出しをする次のコードを見て見ます。

```
console.log("タイマー呼び出し前");
setTimeout(() => {
  console.log("時間が来た");
}, 1000);
console.log("タイマー呼び出し後");
```

このコードを実行すると次の順序でログが出力されます。

```
タイマー呼び出し前
タイマー呼び出し後 // 上の行と同時に表示
時間が来た         // 1 秒後に表示
```

JavaScript では時間のかかる処理を実行する場合、完了した後に呼び出す処理を処理系に渡すことはあっても、そこで処理を止めることはありません。タイマーを設定する `setTimeout()` 関数の実行自体は即座に完了し、その次の行がすぐ呼ばれます。そして時間のかかるタイマーの待ちが完了すると、渡してあった関数が実行されます。処理が終わるのをじっくり待つ（同期）のではなく、完了したら後から連絡してもらおう（非同期）のが JavaScript のスタイルです。

昔の JavaScript のコードでは、時間のかかる処理を行う関数は、かならず引数の最後がコールバック関数でした。このコールバック関数の中にコードを書くことで、時間のかかる処理が終わったあとに実行する、というのが表現できました。

11.2 コールバックは使わない

以前は JavaScript で数多くの非同期処理を実装しようとする、多数のコールバック関数を扱う必要があり、以前はコールバック地獄と揶揄されていました。

リスト 1 非同期の書き方

```
// 旧: Promise 以前
func1(引数, function(err, value) {
  if (err) return err;
  func2(引数, function(err, value) {
    if (err) return err;
    func3(引数, function(err, value) {
      // 最後に実行されるコードブロック
    });
  });
});
```

その後、Promise が登場し、ネストが 1 段になり、書きやすく、読みやすくなりました。Promise はその名の通り「重たい仕事が終わったら、あとで呼びに来るからね」という約束です。これにより、上記のような、深いネストがされたコードに触れる必要が減ってきました。何階層もの待ちが発生しても、1 段階のネストで済むようになりました。

この Promise の実装は、文法の進化に頼ることなく、既存の JavaScript の文法の上で実装されたトリックで実現できました。コミュニティベースで実現されたソリューションです。この Promise は現在も生き続けている方法です。直接書く機会は減ると思いますが、Promise について学んだことは無駄にはなりません。

リスト 2 非同期の書き方

```
// 中: Promise 以後
fetch(url).then(resp => {
  return resp.json();
}).then(json => {
  console.log(json);
}).catch(e => {
  // エラー発生時にここを通過する
}).finally(() => {
  // エラーが発生しても、正常終了時もここを通過する
});
```

Promise の `then()` 節の中に、前の処理が終わった時に呼び出して欲しいコードを書きます。また、その `then()` のレスポンスもまた Promise なので、連続して書けるというわけです。また、この `then()` の中で `return` で返されたものが次の `then()` の入力になります。また、この `then()` の中で Promise を返すと、その返された Promise が解決すると、その結果が次の `then()` の入力になります。遅い処理を割り込ませるイメージです。`catch()` と `finally()` は通常の例外処理と同じです。`finally()` は ES2018 で取り込まれた機能です。

コールバック地獄では、コードの呼び出し順が上から下ではなく上 → 下 → 中と分断されてしまいましたが、

Promise の `then()` 節だけをみれば、上から下に順序良く流れているように見えます。初めて見ると面食らうかもしれませんが、慣れてくるとコールバックよりも流れは追いやすいでしょう。

この Promise が JavaScript 標準の方法として決定されると、さらなる改善のために `await` という新しいキーワードが導入されました。これは Promise を使ったコードの、`then()` 節の中だけを並べたのとはほぼ等価になります。それにより、さらにフラットに書けるようになりましたし、行数も半分になります。内部的には、`await` はまったく新しい機構というわけではなく、Promise を扱いやすくする糖衣構文で、`then()` を呼び出し、その引数で渡される値が関数の返り値となるように動作します。Promise 対応のコードを書くのと、`await` 対応のコードを書くのは差がありません。Promise でない返り値の関数の前に `await` を書いても処理が止まることはありません（エラーになることはありません）。

リスト 3 非同期の書き方

```
// 新: 非同期処理を await で待つ (ただし、await は async 関数の中でのみ有効)
const resp = await fetch(url);
const json = await resp.json();
console.log(json);
```

`await` を扱うには、`async` をつけて定義された関数でなければなりません。TypeScript では、`async` を返す関数の返り値は必ず Promise になります。ジェネリクスのパラメータとして、返り値の型を設定します。

```
async function(): Promise<number> {
  await 時間のかかる処理 ();
  return 10;
}
```

なお、Promise を返す関数は、関数の宣言文を見たときに動作が理解しやすくなるので `async` をつけておく方が良いでしょう。ESLint の TypeScript プラグインでも、推奨設定でこのように書くことを推奨しています^{*1}。

TypeScript の処理系は、この Promise の種類と、関数の返り値の型が同一かどうかを判断し、マッチしなければエラーを出してくれます。非同期処理の場合、実際に動かしてデバッグしようにも、送る側の値と、受ける側に渡ってくる値が期待通りかどうかを確認するのが簡単ではありません。ログを出して見ても、実際に実行されるタイミングがかなりずれていることがあります。TypeScript を使うメリットには、このように実際に動かすデバッグが難しいケースでも、型情報を使って「失敗するとわかっている実装」を見つけてくれる点にあります。

比較的新しく作られたライブラリなどは最初から Promise を返す実装になっていると思いますが、そうでないコールバック関数方式のコードを扱う時は `new Promise` を使って Promise 化します。

```
// setTimeout は最初がコールバックという変態仕様なので仕方なく new Promise
const sleep = async (time: number): Promise<number> => {
  return new Promise<number>(resolve => {
    setTimeout(() => {
      resolve(time);
    }, time);
  });
}
```

(次のページに続く)

^{*1} @typescript-eslint/promise-function-async という設定が該当します。

(前のページからの続き)

```
});  
};  
  
await sleep(100);
```

末尾がコールバック、コールバックの先頭の引数は Error という、2010 年代の行儀の良い API であれば、Promise 化してくれるライブラリがあります。Node.js 標準にもありますし、npm で調べてもたくさんあります。

```
// Node.js 標準ライブラリの promisify を使う  
  
import { promisify } from "util";  
import { readFile } from "fs";  
const readFileAsync = promisify(readFile);  
  
const content = await readFileAsync("package.json", "utf8");
```

11.3 非同期と制御構文

TypeScript で提供されている if や for、while などは関数呼び出しを伴わないフラットなコードなので await とも一緒に使えます。Promise やコールバックを使ったコードで、条件によって非同期処理を 1 つ追加する、というコードを書くのは大変です。試しに、TypeScript の PlayGround で下記のコードを変換してみるとどうなるか見て見ると複雑さにひっくり返るでしょう。

```
// たまに実行される  
async function randomRun() {  
}  
  
// 必ず実行される  
async function finallyFunc() {  
}  
  
async function main(){  
  if (Date.now() % 2 === 1) {  
    await randomRun();  
  }  
  await finallyFunc();  
}  
  
main();
```

これを見ると、await は条件が複雑なケースでも簡単に非同期を含むコードを扱えるのがメリットであることが理解できるでしょう。

await を使うと、ループを一回回るたびに重い処理が完了するのを待つことができます。同じループでも、配列

の `forEach()` を使うと、1 要素ごとに `await` で待つことはできませんし、すべてのループの処理が終わったあとに、何かを行わせることもできません。

```
// for of, if, while, switch は await との相性も良い
for (const value of iterable) {
  await doSomething(value);
}
console.log("この行は全部のループが終わったら実行される");
```

```
// この await では待たずにループが終わってしまう
iterable.forEach(async value => {
  await doSomething(value);
});
console.log("この行はループ内の各処理が回る前に即座に実行される");
```

11.4 Promise の分岐と待ち合わせの制御

Promise は「時間がかかる仕事が終わった時に通知するという約束」という説明をしました。みなさんは普段の生活で、時間がかかるタスクというのを行ったことがありますよね？味噌汁をガスレンジあたためつつ、ご飯を電子レンジで温め、両方終わったらいただきます、という具合です。Promise および、その完了を待つ `await` を使えば、そのようなタスクも簡単に実装できます。

```
async function 味噌汁温め(): Promise<味噌汁> {
  await ガスレンジ();
  return new 味噌汁();
}

async function ご飯温め(): Promise<ご飯> {
  await 電子レンジ();
  return new ご飯();
}

const [a 味噌汁, a ご飯] = await Promise.all([味噌汁温め(), ご飯温め()]);
いただきます(a 味噌汁, a ご飯);
```

`味噌汁温め()` と `ご飯温め()` は `async` がついた関数です。省略可能ですがあえて返り値に `Promise` をつけています。これまでの例では、`async` 関数を呼ぶ時には `await` をつけていました。`await` をつけると、待った後の結果（ここでは味噌汁とご飯のインスタンス）が帰ってきます。`await` をつけないと、`Promise` そのものが帰ってきます。

この `Promise` の配列を受け取り、全部の `Promise` が完了するのを待つのが `Promise.all()` です。`Promise.all()` は、引数のすべての結果が得られると、解決して結果をリストで返す `Promise` を返します。`Promise.all()` の結果を `await` すると、すべての結果がまとめて得られます。

この `Promise.all()` は、複数のウェブリクエストを同時に並行で行い、全てが出揃ったら画面を描画する、な

ど多くの場面で使えます。ループで複数の要素を扱う場合も使えます。

なお、`Promise.all()` の引数の配列に、`Promise` 以外の要素があると、即座に完了する `Promise` として扱われます。

類似の関数で `Promise.race()` というものがあります。これは `all()` と似ていますが、全部で揃うと実行されるわけではなく、どれか一つでも完了すると呼ばれます。レスポンスの値は、引数のうちのどれか、ということで、結果を受け取る場合は処理が少し複雑になります。結果を扱わずに、5 秒のアニメーションが完了するか、途中でクリックした場合には画面を更新する、みたいな処理には適しているかもしれません。

11.5 ループの中の `await` に注意

`for` ループと `await` が併用できることはすでに紹介しました。しかし、このコード自体は問題があります。

```
for (const value of iterable) {  
  await doSomething(value);  
}
```

この `doSomething()` の中で外部 API を呼び出しているとなると、要素数×アクセスにかかる時間だけ、処理時間がかかります。要素数が多い場合、要素数に比例して処理時間が伸びます。この `await` を内部にもつループがボトルネックとなり、ユーザーレスポンスが遅れることもありえるかもしれません。上記のような例を紹介はしましたが、基本的にループ内の `await` は警戒すべきコードです。

この場合、`Promise.all()` を使うと、全部の重い処理を同時に投げ、一番遅い最後の処理が終わるまで待つことができます。配列の `map()` は、配列の中のすべての要素を、指定の関数に通し、その結果を格納する新しい配列（元の配列と同じ長さ）を作り出して返します。詳しくは関数型スタイルのコーディングの紹介で触れますが、このメソッドを使うと、上記の例のような、`Promise` の配列を作ることができます。`Promise.all()` の引数は、`Promise` の配列ですので、これをそのまま渡すと、全部の処理が終わるのを待つ、という処理が完成します。

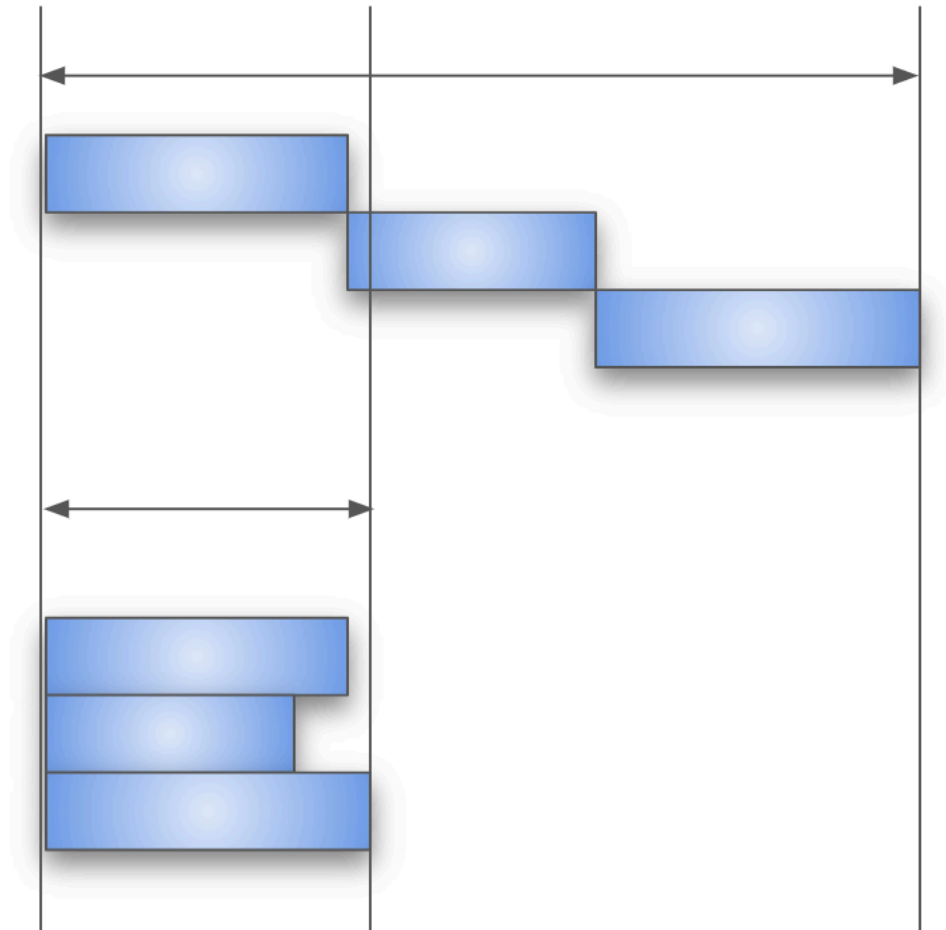
```
await Promise.all(  
  iterable.map(  
    async (value) => doSomething(value)  
  )  
);
```

図で見て見ると、この違いは一目瞭然でしょう。

`Promise.all()` が適切ではない場面もいくつかあります。

例えば、外部の API 呼び出しをする場合、たいてい、秒間あたりのアクセス数が制限されています。配列に 100 個の要素があるからといって 100 並列でリクエストを投げるとエラーが帰って来て正常に処理が終了しないこともありえます。その場合は、並列数を制御しつつ `map()` と同等のことを実現してくれる `p-map`^{*2} といったライブラリを活用すると良いでしょう。

^{*2} <https://www.npmjs.com/package/p-map>



for ループ内部の `await` のように、順番に処理をするための専用構文もあります。 `asyncIterator` というプロトコルを実装したオブジェクトでは、 `for await (const element of obj)` という ES2018 で導入された構文も使えるようになります。 `fetch` のレスポンスのボディがそれにあたります。普段は `json()` メソッドなどで一括で変換結果を受け取ると思いますが、細切れのブロック単位で受信することもできます。この構文を使うと、それぞれのブロックごとにループを回す、という処理が行えます。ただし、それ以外の用途は今のところ見かけませんし、この用途で使うところも見つかりませんので、基本的にはループの中の `await` は要注意であることは変わりありません。

11.6 非同期で繰り返し呼ばれる処理

`async/await` は便利なものですが、ワンショットで終わるイベント向けです。繰り返し行われるイベント (`addEventListener()` を使うようなスクロールイベントとか、画面のリサイズ、 `setInterval()` の繰り返しタイマー) に対しては引き続きコールバック関数を登録して使います。

そこをモダンにしようという動きには [RxJS](#) があります。

11.7 まとめ

Promise と await について紹介しました。非同期は本質的に、難しい処理です。その難しい処理をなるべく簡単に表現しよう、という試みがむかしから試行錯誤されてきました。その 1 つの成果がこの TypeScript で扱えるこの 2 つの要素です。

上から順番に実行されるわけではありませんし、なかなかイメージが掴みにくいかもしれません。最終的には、頭の中で、どの部分が並行で実行されて、どこで待ち合わせをするか、それがイメージができれば、非同期処理の記述に強い TypeScript のパフォーマンスを引き出せるでしょう。

非同期処理を扱うライブラリとして、より高度な処理を実現するための rxJS というものがあります。これはリアクティブの章で紹介します。

第 12 章

例外処理

TypeScript は Java と似たような例外処理機構を備えています。ただし、ベースとなっている JavaScript の言語の制約から、使い勝手などは多少異なります。

12.1 TypeScript の例外処理構文

$A \rightarrow B \rightarrow C$ と順番にタスクをこなすプログラムがあったとします。例えば、データを取得してきて、それを加工して、他のサーバーに送信するバッチ処理のプログラムとかを想像してください。これが順番通りうまくいけば何も問題はありませんが、例えば、データ取得時や送信時にネットワークにうまく繋がらない、加工しようと思ったが、サーバーから送られてきたデータが想定と違ったなど、うまくいかないこともありえます。その場合に、処理を中断する（例外を投げる）、中断したことを察知して何かしらの対処をする（回復処理）を行います。これらの機構をまとめて例外処理と呼んだりします。

throw を使って例外を投げます。throw すると、その行でその関数やメソッド内部の処理は中断し、呼び出し元、そのさらに呼び出し元、と処理がどんどん巻き戻っていきます。最終的に回復処理を行う try 節/ catch 節のペアにあたるまで巻き戻ります。

```
throw new Error("ネットワークアクセス失敗");

console.log("この行は実行されない");
```

例外を投げうる処理の周りは try 節で囲みます。catch 節は例外が飛んできたときに呼ばれるコードブロックです。例外が発生してもしなくても、必ず通るのが finally 節です。後片付けの処理を書いたりします。finally は省略できます。

```
try {
  const data = await getData();
  const modified = modify(data);
  await sendData(modified);
} catch (e) {
```

(次のページに続く)

(前のページからの続き)

```
console.log(`エラー発生 ${e}`);  
} finally {  
  // 最後に必ず呼ばれる  
}
```

もし、回復処理で回復し切れない場合は、再度例外を投げることもできます。

リスト 1 例外の再送

```
try {  
  //  
} catch (e) {  
  throw e; // 再度投げる  
}
```

Java の例外と似ていると最初に紹介しましたが、Java と異なるのが、ベースの JavaScript のコードは型情報をソースコード上に持っていないという点があります。Java の場合は、例外の `catch` 節を複数持つことができ、それぞれの節に例外のクラスの種類を書いておくと、飛んできた例外の種類に応じて適切な節が選択されます。JavaScript では 1 つしか書くことができません。型の種類による分岐というのも、`catch` 節の中で `if` 文を使って行う必要があります。

例外クラスを自分で作る必要はありますが、Java と同じことを実現するには、以下のようなコードになります。

リスト 2 Java と似たような例外の分岐

```
try {
  // 何かしらの処理
} catch (e) {
  // instanceof を使ってエラーの種類を判別していく
  if (e instanceof NoNetworkError) {
    // NoNetworkError の場合
  } else if (e instanceof NetworkAccessError) {
    // NetworkAccessError の場合
  } else {
    // その他の場合
  }
}
```

12.2 Error クラス

例外処理で「問題が発生した」ときに情報伝達に使うのが Error クラスです。さきほどの構文は new と同時に throw していましたが、ふつうのオブジェクトです。

Error クラスは作成時にメッセージの文字列を受け取れます。name 属性にはクラス名、message には作成時にコンストラクタに渡した文字列が格納されます。

JavaScript の言語の標準に含まれていない、処理系独自の機能（といっても、今のところ全ブラウザで使える）のが、stack プロパティに格納されるスタックトレースです。throw した関数が今までどのように呼ばれてきたかの履歴です。ファイル名や行数も書かれていたりします。これがなければ TypeScript のデバッグは数万倍困難だったでしょう。

なお、この行数は TypeScript をコンパイルした後の JavaScript のファイル名と行番号だったりしますが、ソースマップというファイルをコンパイル時に出力しておき、実行時にそれがうまく読み込めると（ブラウザなら一緒にアップロード、Node.js は npm の source-map-support パッケージを利用すれば、もともとの TypeScript のファイル名と行番号で出力されるようになります）。

```
const e = new Error('エラー発生');
console.log(`name: ${e.name}`);
// name: Error
console.log(`message: ${e.message}`);
// message: エラー発生
console.log(`stack: ${e.stack}`);
// stack: Error: test
//   at new MyError (<anonymous>:17:23)
//   at <anonymous>:23:9
```

12.2.1 標準の例外クラス

それ以外にも、いろいろな例外のためのクラスがあります。TypeScript を使っているとコンパイル前に多くの問題を潰せるため、遭遇する回数は JavaScript よりも減ります。

- EvalError
- RangeError
- ReferenceError
- SyntaxError
- TypeError
- URIError

例外を受け取って何もしない（俗称：例外を握りつぶす）は行儀がよくないコードとされますが、JSON パース時には文法がおかしい場合に `SyntaxError` が発生します。`JSON.parse()` だけは拾って無効値で初期化という処理は頻繁に行うでしょう。

```
let json: any;
try {
  json = JSON.parse(jsonString);
} catch (e) {
  json = null;
}
```

あとはブラウザの `fetch()` 関数でサーバー側の API にアクセスするときに、ネットワークエラー（cors での権限がない場合も）は `TypeError` が発生します。`fetch` は JSON をパースする場合に `SyntaxError` も発生します。

```
try {
  const res = await fetch("/api/users"); // ここで TypeError 発生の可能性
  if (res.ok) {
    const json = await res.json(); // ここで SyntaxError 発生の可能性
  }
}
```

よくやりがちなのが、`ok` の確認をしない（ステータスコードが 200 以外で JSON 以外が帰ってきているときに）JSON をパースしようとしてエラーになることです。404 Not Found のときは、ボディが Not Found というテキストになるので、未知のトークン N というエラーになります。あとは 403 Forbidden のときには、未知のトークン F のエラーが発生します。

```
SyntaxError: Unexpected token N in JSON at position 0
```


12.3 例外処理とコードの読みやすさ

例外処理も、コードを読む人の理解を手助けするための、ちょっとしたコツがあります。

12.3.1 try 節はなるべく狭くする

「この関数を呼ぶと、A と B の例外が飛んでくる可能性がある」というのはできあがったソースコードを見ても情報はわかりません。次のコード例を見ても、A から E のどこでどんな例外が飛んでくるかわからないでしょう。

リスト 3 広すぎる try は例外の出どころをわかりにくくする

```
try {  
  logicA();  
  logicB();  
  logicC();  
  logicD();  
  logicE();  
} catch (e) {  
  // エラー処理  
}
```

なるべく狭くすることで、どの処理がどの例外を投げるのかが明確になります。

リスト 4 try の範囲を狭めると、どこで何がおきるのかがわかりやすくなる

```
logicA();  
logicB();  
try {  
  logicC();  
} catch (e) {  
  // エラー処理  
}  
logicD();  
logicE();
```

実際に実行時に例外が起きうる（どんなにデバッグしても例外を抑制できない）ポイントは、外部の通信とかごく一部のはずです。あまりたくさん例外処理を書く必要もないと思いますし、書く場合もどこに書いたかがわかりやすくなります。

広くする問題としては、原因の違う例外が混ざってしまう点もあります。例えば、JSON のパースを何箇所かでなっていると、それぞれの箇所で `SyntaxError` が投げられる可能性が出てきます。原因が違ってリカバリー処理が別の例外が同じ `catch` 節に入ってきてしまうと、正確に例外を仕分けを行って漏れなく対応する必要が出てきます。しかし、どの例外が投げられるかを明示できる文法がなく、実装のコードを見ないとどのような例外が投げられてくるか分かりません。そのため、「間違いなく対処できているか？」を判断するコストが極めて高くなります。

12.3.2 Error 以外を throw しない

前述の catch 文のサンプルでは、e の型が Error という前提で書いていました。これにより、catch 節の中でコード補完がきくので、開発はしやすくなります。しかし、実際には、どの型がくるかは実行時の throw 次第です。throw には Error 関連のクラス以外にも、文字列とか数値とかなんでも投げることができるからです。

基本的には Error 関連のオブジェクトだけを throw するようにしましょう。

```
try {
  :
} catch (e) {
  // e. とタイプすると、name, message などがサジェストされる
  console.log(e.name);
}
```

12.4 リカバリー処理の分岐のためにユーザー定義の例外クラスを作る

例外処理のためにクラスを作ってみましょう。Error を継承することで、例外クラスを作ることができます。ただし、少し Error クラスは特殊なので、いくつかの追加処理をコンストラクタで行う必要があります。5 個例外クラスを作るとして、全部のクラスで同じ処理を書くこともできます。しかし、これが 10 個とか 20 個になると大変です。1 つのベースのクラスを作り、実際にコード中で扱うクラスはこれから継承して作るようにします。

```
// 共通エラークラス
class BaseError extends Error {
  constructor(e?: string) {
    super(e);
    this.name = new.target.name;
    // 下記の行は TypeScript の出力ターゲットが ES2015 より古い場合 (ES3, ES5) のみ必要
    Object.setPrototypeOf(this, new.target.prototype);
  }
}

// BaseError を継承して、新しいエラーを作る
// statusCode 属性に HTTP のステータスコードが格納できるように
class NetworkAccessError extends BaseError {
  constructor(public statusCode: number, e?: string) {
    super(e);
  }
}

// 追加の属性がなければ、コンストラクタも定義不要
class NoNetworkError extends BaseError {}
```

このようにクラスをいくつも作ると、例外を受け取った catch 節で、リカバリーの方法を「選ぶ」ことが可能になります。投げられたクラスごとに instanceof と組み合わせて条件分岐に使えます。また、この instanceof は

型ガードになっていますので、各ブロックの中でコード補完も正しく行われます。上記のクラスの `statusCode` も正しく補完されます。

```
try {
  await getUser();
} catch (e) {
  if (e instanceof NoNetworkError) {
    alert("ネットワークがありません");
  } else if (e instanceof NetworkAccessError) {
    // この節では、e は NetworkAccessError のインスタンスなので、
    // ↓の e. をタイプすると、statusCode がサジェストされる
    if (e.statusCode < 500) {
      alert("プログラムにバグがあります");
    } else {
      alert("サーバーエラー");
    }
  }
}
```

なお、TypeScript は、昔の Java のように継承を前提とした処理を書くことはほとんどありませんので、コードの中で継承を使うことも極めてまれです。Java の場合は、`IOException` クラスを継承したクラスがあって、入出力系のエラーなど継承階層を前提としたコードが書かれたりもしました。しかし、これは「A は B の子クラスである」という知識を持っていないと読めないコードになってしまうため、プロジェクトに入ってきたばかりの人には混乱を与えがちです。例外クラスを作る場合も、`BaseClass` からの直系の子供クラスだけで作れば問題ありません。立派な継承ツリーの設計は不要です。あまり例外クラスが多くても使い分けに迷ったりします。

注釈: ターゲットが ES3/ES5 のときに `Object.setPrototypeOf(this, new.target.prototype);` の行を書き忘れると、`instanceof` が `false` を返してくるようになります。

12.5 例外処理を使わないエラー処理

正常に実行できなかったからといって、なんでも例外として処理しなければならないわけではありません。例えば、ブラウザ標準の `fetch` API の場合、通信ができたが、正常に終わらなかった場合は `ok` 属性を使って判断できます。例外には深い階層から一発で離脱できる（途中の関数では、エラーがあったかどうかを判定不要）メソッドがあります。しかし、階層が深くなく、呼び出し元と例外処理を行うコードがすごく近い場合には、この `ok` のような属性を用意の方が管理もしやすいでしょう。

```
const res = await fetch("/users");
if (res.ok) {
  // ステータスコードが 200/300 番台
} else {
  // 400 番以降
```

(次のページに続く)

(前のページからの続き)

}

12.6 非同期と例外処理

非同期処理で難しいのがエラー処理でした。async と await のおかげで例外処理もだいぶ書きやすくなりました。

Promise では then() の 2 つめのコールバック関数でエラー処理が書けるようになりました。また、エラー処理の節だけを書く catch() 節もあります。複数の then() 節が連なっている、1 箇所だけエラー処理を書けば大丈夫です。なお、一箇所もエラー処理を書かずにいて、エラーが発生すると unhandledRejection というエラーが Node.js のコンソールに表示されることになります。

リスト 5 Promise のエラー書き方

```
fetch(url).then(resp => {
  return resp.json();
}).then(json => {
  console.log(json);
}).catch(e => {
  console.log("エラー発生!");
  console.log(e);
});
```

async 関数の場合はもっとシンプルで、何かしらの非同期処理を実行する場合、await していれば、通常の try 文でエラーを捕まえることができます。

リスト 6 async 関数内部のエラー処理の書き方

```
try {
  const resp = await fetch(url);
  const json = await resp.json();
  console.log(json);
} catch (e) {
  console.log("エラー発生!");
  console.log(e);
}
```

エラーを発生させるには、Promise 作成時のコールバック関数の 2 つめの引数の reject() コールバック関数に Error オブジェクトを渡しても良いですし、then() 節の中で例外をスローしても発生させることができます。

```
async function heavyTask() {
  return new Promise<number>((resolve, reject) => {
    // 何かしらの処理
  });
}
```

(次のページに続く)

(前のページからの続き)

```

    reject(error);
    // こちらでも Promise のエラーを発生可能
    throw new Error();
  });
};

```

Promise 以前は非同期処理の場合は、コールバック関数の先頭の引数がエラー、という暗黙のルールで実装されていました。ただし、1つのコールバックでも `return` を忘れると動作しませんし、通常の例外が発生して `return` されなかったりすると、コールバックの伝搬が中断されてしまいます。

リスト 7 原始時代の非同期のエラー処理の書き方

```

// 旧: Promise 以前
func1(引数, function(err, value) {
  if (err) return err;
  func2(引数, function(err, value) {
    if (err) return err;
    func3(引数, function(err, value) {
      // 最後に行われるコードブロック
    });
  });
});
});

```

12.7 例外とエラーの違い

この手の話になると、エラーと例外の違いとか、こっちはハンドリングするもの、こっちは OS にそのまま流すものとかいろんな議論が出てきます。例外とエラーの違いについても、コンセンサスは取れておらず、人によって意味が違ったりします。一例としては、回復可能なものがエラーで、そうじゃないものが例外といったことが言われたりします。このエントリーではエラーも例外も差をつけずに、全部例外とひっくるめて説明します。

例外というのはすべて、何かしらのリカバリーを考える必要があります。

- ちょっとしたネットワークのエラーなので、3回ぐらいはリトライしてみる
 - 原因: ネットワークエラー
 - リカバリー: リトライ
- サーバーにリクエストを送ってみたら 400 エラーが帰ってきた
 - 原因: リクエストが不正
 - リカバリー (開発時): 本来のクライアントのロジックであればバリデーションで弾いていないといけ
ないのでこれは潰さないといけない実装バグ。とりあえずスタックトレースとかありったけの情報を
`console.log` に出しておく。

- リカバリー (本番): ありえないバグが出た、とりあえず中途半端に継続するのではなくて、システムエラー、開発者に連絡してくれ、というメッセージをユーザーに出す (人力リカバリー)
- JSON をパースしたら `SyntaxError`
 - 原因: ユーザーの入力が不正
 - リカバリー: フォームにエラーメッセージを出す

最終的には、実装ミスなのか、ユーザーが間違っただけなのかという実行時の値の不正なのか、ネットワークの接続がおかしい、クラウドサービスの秘密鍵が合わないみたいな環境の問題なのか、どれであったとしても、システムが自力でリカバリーする、ユーザーに通知して入力修正や WiFi のある環境で再実行などの人力リカバリーしてもらい、開発者に通知してプログラム修正するといった人力リカバリーなど、何かしらのリカバリーは絶対必要になります。

Node.js で `async/await` やら `Promise` を一切使っていないコードの場合、エラーを無視すると、Node.js 自体が最後に `catch` して、エラー詳細を表示してプログラムが終了します。これはある意味プログラムとしては作戦放棄ではありますが、「プログラムの進行が不可能なので、OS に処理を返す」というリカバリーと言えなくもないでしょう。開発者にスタックトレースを表示して後を託す、というのも立派なリカバリーの戦術の 1 つです。

ブラウザの場合、誰もキャッチしないと、開発者ツールのコンソールに表示されますが、開発者ツールを開いていない限りエラーを見ることはできません。普段から開発者コンソールを開いている人はまれだと思いますし、大部分のユーザーには正常に正常に処理が進んだのか、そうでなかったのかわかりませんので、かならずキャッチして画面に表示してあげる必要があるでしょう。

どちらにしても何かしらのリカバリー処理が必要となりますので、本書ではエラーと例外の区別といったことはしません。

12.8 例外処理のハンドリングの漏れ

例外処理が漏れた場合、ロジックが中断します。例えば、サーバー通信結果の JSON のパースでエラーが発生すると、`SyntaxError` が発生し、その後の処理が行われなくなります。その JSON をパースして画面に表示しようとしていた場合は、通信は行われるものの、画面表示が更新されずに何も発生していないように見えます。この場合、開発者ツールをみると、`Uncaught Error` が記録されています。なお、将来の Node.js ではエラーコード 0 以外で、プログラムが終了することになっています。

一方、やっかいなのが、サーバー通信のエラーです。`XMLHttpRequest` の場合は `onerror()` コールバックでエラーを取得してハンドリングする必要がありますが、うまくハンドリングしてもコンソールにはエラーが残ります。`fetch` の場合は `Uncaught (in promise) Error` が出ますが、これを適切に処理しても、やはり同様のエラーが出力されます。

これは [console.log によるログ出力](#) の章で紹介した `console.error()` 出力と同じです。うまくハンドリングしても、キャッチしなかったときと同じように赤くエラーが表示されます。XHR を直接使う場合は `onerror` が設定されていなくて例外が握り潰されている可能性もゼロではないため注意しなければなりません、XHR をラッ

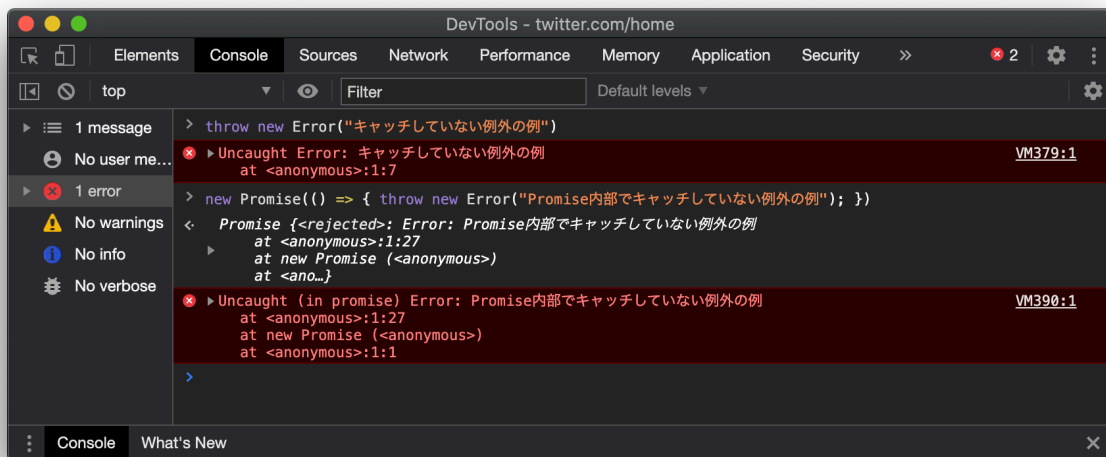


図1 キャッチしていない例外

プした `axios` や、`fetch()` を使った場合には `Uncaught Error` の文字がなければ実装上は気にする必要はありません。



図2 うまく処理できていた場合にも同様に表示されるエラー

12.9 例外処理機構以外で例外を扱う

これまで TypeScript における例外処理の方法や作法などを説明してきました。しかし、ベースとなっている JavaScript の制約により、お世辞にも使いやすい機能とは言えません。理由は以下の 3 つです。

- Java の `throws` のように、メソッドがなげうる例外の種類がわからず、ソースの関数の実態やドキュメント（整備されていれば）を確認する必要がある
- Java や C++ のように、`catch` 節を複数書いて、型ごとの後処理を書くことができず、`instanceof` を駆使してエラーの種類を見分けるコードを書く必要がある
- `Promise` や `async` 関数で、何が `reject` に渡されたり、どんな例外を投げるのかを型定義に書く方法がない

例外に関しては、補完も効かないし、型のチェックも行えません。いっそのこと、例外処理機構を忘れてしまうのも手です。例外処理のない Go では、関数の返り値の最後がエラーという規約でコードを書きます。TypeScript でも、いくつか方法が考えられます。

- タプルを使ってエラーを返す（Go 式）
- オブジェクトを使ってエラーを返す
- 合併型を使ってエラーを返す

```
type User = {
  name: string;
  age: number;
}

// タプル
function create(name: string, age: number): [User?, Error?] {
  if (age < 0) {
    return [undefined, new Error("before born")]
  }
  return [{name, age}];
}

// オブジェクト
function create2(name: string, age: number): {user?: User, error?: Error} {
  if (age < 0) {
    return {error: new Error("before born")}
  }
  return {user: {name, age}};
}

// 合併型
function create3(name: string, age: number): User | Error {
  if (age < 0) {
```

(次のページに続く)

(前のページからの続き)

```
    return new Error("before born");
  }
  return {name, age};
}
```

この中でどれが良いかは好みの問題ですが、個人的なおすすめはオブジェクトです。タプルよりかは返り値の意味に名前をつけられるのと、合併と異なり、オプショナルチェイニングを使ってエラーチェックを簡単に書ける可能性があります。instanceof と毎回タイプする必要性もありません。ただ、どの書き方もマジョリティではなく、好みの問題ではあります。

12.10 まとめ

例外についての文法の説明、組み込みのエラー型、エラー型を自作する方法、非同期処理の例外処理などを説明してきました。例外の設計も、一種のアーキテクチャ設計であるため、ちょっとした経験が必要になるかもしれません。

TypeScript、特にフロントエンドの場合、例外を無視することはユーザーの使い勝手を悪くします。どのようなことが発生し、どのケースではどのようにリカバリーするか、というのをあらかじめ決めておくと実装は楽になるでしょう。

第 13 章

モジュール

以前の JavaScript は、公式には複数のファイルに分割してコーディングする方法を提供していませんでした。当初は Closure Compiler や、その他の Yahoo! UI や jQuery などのライブラリごとの固有のファイル結合ツール、require.js などを使っていました。その後 Node.js が登場して人気になると、CommonJS というサーバーサイド JavaScript のための仕様から取り込まれたモジュールシステムがデファクトスタンダードとなりました。これはブラウザからは利用しにくい仕様だったため、ブラウザからも利用できる ES2015 modules が仕様化されました。今後の開発では ES2015 modules の理解が不可欠になるでしょう。

本章では、ES2015 modules を中心に、CommonJS との互換性などを取り上げます。アプリケーション開発では、まず基本文法の節だけ理解できていれば十分でしょう。中級、上級のネタはライブラリ作成、環境構築を行う人向けです。

13.1 用語の整理

モジュールを説明するまえに、パッケージも含めて、用語の整理をしておきましょう。なお、コンピュータの世界では、同じ用語だけど、人とかコンテキストによって全然違う意味を持つので、雑に語るのが危険なワードがあります。例えば、コンポーネント、モジュール、パッケージなどがそれにあたります。言語やツールによっても違うし、言語の一部のフレームワークによっても違うし、それらを束ねる抽象的な概念としても扱われます。TypeScript でコードを書くときに他の人が「パッケージ」「モジュール」といったときにどれを指すのか、整理しておきます。

13.1.1 パッケージ

ここで扱うパッケージは、Node.js を核とする、JavaScript や TypeScript 共栄圏の言葉の定義です。パッケージは、Node.js が配布するソフトウェアの塊の最小単位です。npm (Node パッケージマネージャ) や、yarn といったツールを使って、npmjs.org などのリポジトリからダウンロード、社内のファイルサーバーで.tgz の配布物として提供されるものです。Git リポジトリをそのままパッケージとすることもできます (ダウンロードや更新チェックが遅くなるデメリットがあるのであまり使わない方が良いでしょう)。元は Node.js 用として始まりましたが、ブ

ラウザ向けのフレームワークなども今はほとんど多くが `npmjs.org` で配布されています。

パッケージのソースは、`package.json` というパッケージ定義のファイルが含まれているフォルダです。このファイルには、プロジェクトの名前やバージョン、説明、著者名などのメタデータ以外に、開発時に使うコマンド、依存ライブラリなどの情報が含まれます。`github.com` で JavaScript とか TypeScript のプロジェクトを見ると、トップページ、あるいは `projects` や `packages` という名前のフォルダの下の子フォルダ内に `package.json` があることがわかるでしょう。このフォルダの中で、`npm pack` とやれば `.tgz` ファイルができますし、`npm publish` とタイプすると、`npmjs.org` で全世界に向けて公開できます。

パッケージのダウンロードは Node.js と一緒に配布される `npm` コマンドを使って行います。以下のコマンドでは、Vue.js のプロジェクトを作成する CLI コマンドと、Vue.js のライブラリの 2 つをダウンロードしています。

```
$ npm install @vue/cli vue
```

`npmjs.org` で配布しない、自作のアプリケーションやライブラリ、サービスなんかもパッケージとして作成します。Node.js 用ではない、ウェブのフロントエンドのコードでも、Node.js のパッケージ形式でプロジェクトを作成します。公開しない場合でも、パッケージにしておけば、開発用コマンドのランチャーとして使ったり、依存パッケージの管理ができます。`package.json` がある場合、インストールすると、`package.json` に `dependencies` のところに情報が保存されます。`--save-dev` (もしくは `-D`) をつけると、`devDependencies` に保存されます。本番環境ではいらない、開発用のツールなどはこのオプションをつけます。

リスト 1 devDependencies に登録

```
npm install --save-dev [パッケージ名]
```

新しいコンピュータや他人のコンピュータ上で作業フォルダを作る必要があるときは、その以前インストールしたファイル一覧を元に `npm install` コマンドだけですべての必要なパッケージのダウンロードが完了します。`dependencies` と `devDependencies` のパッケージがダウンロードされます^{*1}。そのパッケージが他のパッケージに依存していたら、それらもすべてダウンロードしてインストールされます。`npm install --prod` だと、`dependencies` のみがダウンロードされます。初回にダウンロードされて `package.json` に登録されるときは、サブのパッケージも含めて `package-lock.json` というファイルに全バージョン情報が保存されます。`npm ci` コマンドを使って、これに記録されたバージョンと厳密に一致したバージョンのみをダウンロードすることもできます。

13.1.2 TypeScript とパッケージ

TypeScript 固有トピックとしては、`A` というパッケージ自体に TypeScript 固有の型定義ファイルが含まれないことがあり、`@types/A` という別のパッケージとして提供されていることがあります。TypeScript から見ると、この 2 つで 1 つのパッケージという感覚でいけば良いでしょう。

インストールされているパッケージの型定義ファイルがあればダウンロードする `typesync` コマンドが `npmjs.org` にあります。明示的に `@types` のパッケージをインストールしても良いのですが、`typesync` コマンドをインストー

^{*1} ただし環境変数 `NODE_ENV` が `production` に設定されていると、`--prod` オプションを付けた場合と同じ動作になります。

ルして、install コマンド実行時に毎回実行するようにすると便利です。

リスト 2 typesync コマンドのインストール

```
$ npm install typesync
```

リスト 3 型定義のパッケージが別であれば自動ダウンロード

```
{
  "scripts": {
    "postinstall": "typesync"
  }
}
```

これで、npm install のたびに、型定義ファイルも（あれば）ダウンロードされるようになります。

13.1.3 モジュール

JavaScript や TypeScript 界限でモジュールというと、ECMAScript2015 で入ったモジュールの機能、およびその文法に準拠している TypeScript/JavaScript の 1 つのソースファイルのことを指します。もっとも、これらの界限でも、Angular はまたそれ固有のモジュール機構などを持っていたりしますが、それはここでは置いておきます。

簡単にいえば、1 つの .ts/js ファイルがモジュールです。モジュール機能を使うと、ファイルを分割して、管理しやすいサイズのソースファイルに区切ってプロジェクトの開発をすすめることができます。モジュールは、外部に提供したい要素を export したり、外部のファイルの要素を import することができます。同一のフォルダ内の別のファイルを参照する場合にも、import が必要です。

パッケージの方がモジュールよりも大きな概念ですが、パッケージとモジュールの言葉が同じような文脈で利用されることがあります。パッケージ内部にたくさんのモジュール（ソースコード）が入ります。パッケージの設計時に、1 つの代表となるモジュールに公開要素を集めることができます。パッケージの設定ファイルの中で、デフォルトで参照するモジュールが設定できます（main 属性）。この場合、他のモジュールを import するのと同じように、パッケージの import ができるようになります。大抵の npmjs.org で公開されているパッケージは、このようにデフォルトで読み込まれるソースファイルにすべての要素を集める（ビルドツールで複数ファイルをまとめて結果として 1 ファイルになる場合も含む）のが一般的です。

モジュールの理解のやっかいなところは、裏の仕組みがいろいろある点です。モジュール機能はもともとブラウザのための機能としてデザインされたので、ブラウザでは利用できます。Node.js はオプションをつけると利用できます（ただし、拡張子は .mjs）。それ以外に、webpack などのバンドラーと呼ばれるツールが、import/export 文を解析して、1 つの .js ファイルを生成したりします。Node.js が旧来よりサポートしていた CommonJS 形式のモジュールに、TypeScript の型定義ファイルを組み合わせて import ができるようにしていることもあります。

本ドキュメントでは TypeScript を使いますので、基本的には次の形式のものがモジュールとなります

- TypeScript の 1 ファイル
- TypeScript 用の型定義ファイル付きの npm パッケージ

- TypeScript 用の型定義ファイルなしの npm パッケージ +TypeScript 用の型定義ファイルパッケージ

13.2 基本文法

13.2.1 エクスポート

ファイルの中の変数、関数、クラスをエクスポートすると、他のファイルからそれらが利用できるようになります。エクスポートを行うには `export` キーワードをそれぞれの要素の前に付与します。

リスト 4 エクスポート

```
// 変数、関数、クラスのエクスポート
export const favorite = "小籠包";
export function fortune() {
  const i = Math.floor(Math.random() * 2);
  return ["小吉", "大凶"][i];
}
export class SmallAnimal {
}
```

13.2.2 インポート

エクスポートしたものは `import` を使って取り込みます。エクスポートされた名前をそのまま使いますが、シンボル名が衝突しそうな場合は `as` を使って別名をつけることができます。配布用の JavaScript を作るバンドルツールは、この `import` 文を分析して、不要なコードを最終成果物から落としてファイルサイズを小さくするツリーシェイキングという機能を持っています。

リスト 5 インポート

```
// 名前を指定して import
import { favorite, fortune, SmallAnimal } from "./smallanimal";

// リネーム
import { favorite as favoriteFood } from "./smallanimal";
```

13.2.3 default エクスポートとインポート

他の言語であまりない要素が `default` 指定です。エクスポートする要素の 1 つを `default` の要素として設定できます。

リスト 6 default

```
// default をつけて好きな要素を export
export default address = "小岩井";

// default つきの要素を利用する時は好きな変数名を設定して import
// ここでは location という名前で address を利用する
import location from "./smallanimal";
```

default のエクスポートと、default 以外のエクスポートは両立できます。

リスト 7 default

```
// default つきと、それ以外を同時に import
import location, { SmallAnimal } from "./smallanimal";
```

13.2.4 パスの書き方 - 相対パスと絶対パス

たいていのプログラミング言語でも同等ですが、パス名には、相対パスと絶対パスの 2 種類があります。

- 相対パス: ピリオドからはじまる。import 文が書かれたファイルのフォルダを起点にしてファイルを探す
- 絶対パス: ピリオド以外から始まる。TypeScript などの処理系が持っているベースのパス、探索アルゴリズムを使って探す

絶対パスの場合、TypeScript は 2 箇所を探索します。ひとつが `tsconfig.json` の `compilerOptions.baseDir` です。プロジェクトのフォルダのトップを設定しておけば、絶対パスで記述できます。プロジェクトのファイルは相対パスでも指定できるので、どちらを使うかは好みの問題ですが、Visual Studio Code は絶対パスで補完を行うようです。

```
import { ProfileComponent } from "src/app/component/profile.component";
```

もう一箇所は、`node_modules` 以下です。npm コマンドなどでダウンロードしたパッケージを探索します。親フォルダを辿っていき、その中に `node_modules` というフォルダがあればその中を探します。なければさらに親のフォルダを探し、その `node_modules` を探索します。

注釈: 絶対パスの探索アルゴリズム (`compilerOptions.moduleResolution`) は 2 種類あり、`"node"` を指定したときの挙動です。こちらがデフォルトです。`"classic"` の方は使わないと思うので割愛します。

TypeScript 向けの型情報ファイルも一緒に読み込まれます。パッケージの中に含まれている場合は何もしなくても補完機能やコードチェックが利用できます。そうでない場合は `@types/パッケージ名` というフォルダを探索します。これは、パッケージとは別に提供されている型情報のみのパッケージです。

注釈: 型情報ファイルの置き場は `compilerOptions.typeRoots` オプションで変更できます。既存のパッケージで型情報が提供されておらず、自分のプロジェクトの中で定義する場合に、置き場所を追加するときに使います。詳しくは型定義ファイルの作成の章で紹介します。

13.2.5 動的インポート

`import/export` は、コードの実行を開始するときにはすべて解決しており、すべての必要な情報へのアクセスが可能であるという前提で処理されます。一方で、巨大なウェブサービスで、特定のページでのみ必要とされるスクリプトをあとから読み込ませるようにして、初期ロード時間を減らしたい、ということがあります。この時に使うのは動的インポートです。

これは `Promise` を返す `import()` 関数となっています。この `Promise` はファイルアクセスやネットワークアクセスをしてファイルを読み込み、ロードが完了すると解決します。なお、この機能は出力ターゲットが ES2018 以降のみの機能となります。

```
const zipUtil = await import('./utils/create-zip-file');
```

課題: 要検証

13.2.6 誰が `import` を行うのか?

JavaScript にインポート構文が定義され、ブラウザにも実装は進んでいますが、この機能を使うことは今のところあまりないです。ブラウザ向けの TypeScript のコード開発では、コンパイル時にこの `import`、`export` をそのまま出力します。TypeScript も、この `import` と `export` を解釈して、型情報に誤りがないかは検証しますが、出力時には影響はありません。それを 1 つのファイルにまとめるのは、バンドラーと呼ばれるツールが行います。むしろ、バンドラーからソースコードを変換するフィルターとして TypeScript のコンパイラが呼ばれる、といった方が動作としては正確です。ファイルにまとめるときは、不要な要素を削除するといった処理が行われます。

Node.js 向けに出力する場合は、`import` と `export` を、CommonJS の流儀に変換します。こうすることで、Node.js が実行時に `require()` を使って依存関係を解決します。

読み込みが遅く、実行も遅いとなるとそれだけで敬遠されるので、ブラウザの `import` と `export` が将来的には使われるようになるためには、不要なコードを削除する処理などを行って、効率の良いコードへの変換を行うツールが必要とされるでしょう。しかし、そのようなツールが作られるとして、バンドラーと 9 割がた同じ処理をして、最後の出力だけは元のばらばらな状態で出力しなおす変換ツールになると思われます。それであればバンドラーをそのまま使った方が何かと効率的だと思われるので、実際に作られることになるかどうかはわかりません。

13.3 中級向けの機能

13.3.1 リネームして export

as を使って別名でエクスポートも可能です。たとえば、クラスをそのままエクスポートするのではなく、Redux のストアと接続したカスタム版をオリジナルの名前でエクスポートしたいときに使います。

リスト 8 リネームをしてエクスポート

```
function MyReactComponent(props: {name: string, dispatch: (act: any) => void}) => {
  return <h1>私は小動物の{props.name}です</h1>
}

// リネームしてエクスポート
export { favorite as favariteFood };
```

13.3.2 複数のファイル内容をまとめてエクスポート

TypeScript で大規模なライブラリを作成する場合、1 ファイルですべて実装することはないでしょう。アプリケーションから読み込まれるエントリーポイントとなるスクリプトを 1 つ書き、外部に公開したい要素をそこから再エクスポートすることにより、他の各ファイルに書かれた要素を集約することができます。

記述方法は、import 文の先頭のキーワードを export に変えるだけです。他のファイルでデフォルトでなかった要素を、デフォルトとしてエクスポートすることも可能です。

リスト 9 再エクスポート

```
export { favorite, fortune, SmallAnimal } from "./smallanimal";

// リネームもできる
export { favorite as favoriteFood } from "./smallanimal";

// あとから default にすることもできる
export { favorite as default } from "./smallanimal";
```

13.3.3 自動でライブラリを読み込ませる設定

TypeScript では、インポートの行を書かなくても、すべてのファイルですでにインポート行が書かれているとみなして読み込ませる機能があります。JavaScript の処理系はどれも、標準の ECMAScript の機能だけが提供されているわけではありません。JavaScript は他のアプリケーション上で動くマクロ言語として使われることが多いので、環境用のクラスや関数が提供されることがほとんどです。compilerOptions.types を使うとその環境を再現することができます。

といっても、不用意に乱用するのはよくありません。依存しているのに、依存が見えないということになりがちで

す。たいてい必要なのは、Node.js 用のライブラリ、特定のテストフレームワークの対応ぐらいでしょう。

```
{
  "compilerOptions": {
    "types" : ["node", "jest"]
  }
}
```

なお、ECMAScript のバージョンアップで増える機能や、ブラウザのための機能は、これとは別に `compilerOptions.lib` で設定します。こちらについては環境構築のところで紹介します。

13.4 ちょっと上級の話

13.4.1 パス名の読み替え

ひとつのリポジトリに 1 つのパッケージだけを置いて開発するのではなく、関連するライブラリもすべて一緒のリポジトリに置いてしまう、というモノリポジトリという管理方法があります。この名前を提唱して、積極的に使い出したのは Babel で、コア機能と、それをサポートする大量のプラグインが 1 つのリポジトリに収まっています。この考え方自体は昔からあり、Java の世界ではマルチプロジェクトと呼んでいました。

モノリポジトリのメリットは、依存ライブラリを `publish` しなくても使えるため、依存ライブラリと一緒に機能修正する場合に、同時に編集できます。コア側を `publish` して、それにあわせて依存している方を直して、やっぱりだめだったのでコアを再 `publish` ... みたいなことはやりたくないでしょう。関連パッケージ間のバージョンをきちんとそろえて、歩調を合わせたいというときには便利です。

JavaScript 界限のモノリポジトリでは、`packages` や `projects` といったフォルダを作り、その中にプロジェクトフォルダを並べます。`paths` を使ったパスの読み替えを設定すると、各パッケージでは絶対パスで関連パッケージがインポートできます。

この場合によく使われるのが、ルートに共通設定を書いたファイルを作り、各パッケージではこれを継承しつつ、差分だけを記述する方法です。

リスト 10 `tsconfig.base.json`

```
"compilerOptions": {
  "baseUrl": "./packages",
  "paths": {
    "mylibroot": ["mylibroot/dist/index.d.ts"]
  }
}
```

リスト 11 packages/app/tsconfig.json

```
{
  "extends": "../../tsconfig.base",
  "compilerOptions": {
    "outDir": "dist"
  },
  "include": [".src/**/*.ts"]
}
```

なお、テストフレームワークの Jest の場合は、TypeScript の設定と別途名前のマッピングルールの設定が必要です。次のように書けば大丈夫なはずが、テストコードの場合は相対パスで使ってしまうても問題ないでしょう。

課題: ちょっとうまく動いていないので、要調査

リスト 12 jest.config.js

```
module.exports = {
  moduleNameMapper: {
    "mylibroot": "<rootDir>/../mylibroot/src/index.ts"
  }
}
```

13.4.2 CommonJS との違い

ES2015 modules が仕様化されたとはいえ、残念ながら現在の開発ではこれだけで完結はしません。通常はダウンロードをまとめて行うために事前にバンドラーツールで 1 ファイルにまとめつつ最適化を行います。ライブラリの流通の仕組みが Node.js のエコシステムである npmjs.org で行われることもあって、ライブラリの多くが CommonJS 形式で提供されているため、CommonJS とも連携が必要です。

ES2015 modules を利用して開発されたライブラリも、トランスパイラなどを通じて CommonJS 形式にビルドされてパッケージ化されることがほとんどですが、これを利用する場合は特別な配慮をしなくても import できます。それ以外の CommonJS 形式で手書きで書かれたコードの読み込みではいくつか考慮点があります。

1 つだけエクスポートした場合は、default でそのオブジェクトがエクスポートされたのと同じ動作になります。オブジェクトを使って複数エクスポートする場合は明示的なインポートをしないと問題ありません。default 形式と同様の動作をサポートするには、オブジェクトに default という名前の項目を追加し、なおかつ __esModule: true 属性を付与すれば行えます。

これらの動作は Babel と TypeScript のデフォルト設定で確認しましたが、これらの挙動はオプションでも変更される場合があります。また、Rollup や Parcel などの別のバンドラーツールではまた動作が変わることがあります。

リスト 13 CommonJS のライブラリを ES2015 modules でインポート

```
// 1 つだけ CommonJS 形式でエクスポート
module.exports = "小豆島";

// place=="小豆島";
import place from "./cjs-lib";

// オブジェクト形式でエクスポート (1)
module.exports = {
  place: "小豆島"
};

// place=="小豆島";
import { place } from "./cjs-lib";

// オブジェクト形式でエクスポート (2)
module.exports = {
  place: "小豆島",
  default: "小笠原",
  __esModule: true
};

// place=="小笠原";
import place from "./cjs-lib";
```

13.4.3 型のための import/export

TypeScript 3.8 から、型のみをインポートしたりエクスポートできるようになりました。

リスト 14 型のためのインポート

```
import type { AwesomeType } from "./type";
```

読み込まれるファイルの中に何か副作用のある式があったとします。次のファイルはグローバルなところで `console.log()` があります。このファイルがインポートされるだけで、今日の運勢が出力されるという迷惑なライブラリです^{*2}。

リスト 15 fortunes.ts

```
export const fortunes = ["大吉", "吉", "中吉", "小吉", "末吉", "凶", "大凶"];
console.log(`あなたの今日の運勢は${fortunes[Math.floor(Math.random() * 7)]}`);

export type Fortunes = "大吉" | "吉" | "中吉" | "小吉" | "末吉" | "凶" | "大凶";
```

^{*2} 他に迷惑な有名なライブラリとしては、Python の `this` があります。 `import this` をすると Python の設計思想を表す詩が表示されます。

TypeScript はインポートしたものが型だけの場合に、出力からはそのインポート文を丸ごと排除します。排除されると、`ncc` や `Babel` などのバンドラーの出力結果に、そのインポート先のファイルが含まれなくなるため、副作用はおきません。副作用を起こしたい場合は型ではなく値を読み込むか、読み込み対象を指定せずにインポートします。

```
// 値を読み込むと副作用発生
import { fortunes } from "./fortunes";

// 型だけなら発生せず
import { Fortunes } from "./fortunes";

// 対象を絞らなくても副作用発生
import "./fortunes";
```

そもそも副作用があるモジュールはあまりないとは思いますが、この副作用が発生しないことを明示的に指定するのが型のみのインポートです。

この挙動を制御するオプションが 3.8 から増えました。`tsc --init` しても出力されない、レアなオプションです。

```
compilerOptions.importsNotUsedAsValues: "remove" | "preserve" | "error"
```

- "remove": 削除する（現行のデフォルトとおなじ）
- "preserve": 型だけであってもインポートを残し、副作用が必ず発生するようになる
- "error": 型だけを通常の名前束縛のインポート構文で読み込むとエラーにする

この最後の `preserve` や `error` の時に、副作用なく型のインポートのみを許容する構文があります。それが次の書き方です。

```
// 型だけなら発生せず
import type { Fortunes } from "./fortunes";

// compilerOptions.importsNotUsedAsValues: "error"だとエラーに
import { Fortunes } from "./fortunes";
```

なお、この構文は読み込めるのは型だけなので、コロンの左側に来る要素で使うとエラーになります。

型だけインポートを使うと今までも現在もインポート自体がなかったことにされますが、このオプションにより副作用の有無が明示的なコードを書くことができるようになります。`strict` を限界まで設定しているユーザーは `compilerOptions.importsNotUsedAsValues: "error"` も追加すると良いでしょう。

インポートだけではなく、`export type { A, B } from "./modules";` といった、インポートして即エクスポートする文においては、`export` にも利用できます。

現在リリースされている `typescript-eslint` の 4.0 以降においても、`import type` を使うことを強制するルールが

追加されました^{*3}。TypeScript 3.7 以前ではすべてがエラーになってしまうため、デフォルトでは有効化されていません。

リスト 16 eslintrc.json

```
{
  "rules": {
    "@typescript-eslint/consistent-type-imports": "error"
  }
}
```

13.5 まとめ

インポートとエクスポートのための構文自体は難しくありません。ファイル名を間違ったりしても、Visual Studio Code などのエディタがすばやくエラーを見つけてくれるため、問題の発見と解決は素早く行えるでしょう。

JavaScript には当初モジュール機構がなく、後から追加されたりしたため、過去の経緯、CommonJS などの他の仕組みも考慮したうえで設定を行う必要があったりします。しかし、最終的には ES2015 形式のモジュール記法に統一されていくため、基本的にはこちらですべて記述していけば良いでしょう。

やっかいなのはモノリポジトリなどの複雑な環境です。こちらは環境構築を行うメンバーが気合を入れて取り組む必要があるでしょう。

^{*3} <https://github.com/typescript-eslint/typescript-eslint/blob/v4.1.1/packages/eslint-plugin/docs/rules/consistent-type-imports.md>

第 14 章

console.log によるログ出力

基本編のおまけでログ出力について紹介します。言語を学ぶときやデバッグ時には変数を出力することで状態を知ろうとするでしょう。その時に使えるのが Console API です。これらは特に TypeScript 特有のものではありませんが、覚えておいて損はないので紹介します。

```
console.log("ログ出力");
```

これが一番使われるものだと思いますが、他にもいろいろあります。これを実装すべきといった仕様はないのですが、現状で各環境で実装されている仕様は Living Standard として WHATWG にあります。しかし、どの環境でも使えるとは限りません。特に、TypeScript Playground で使えないものが多数ありますが、これらは何もしないだけでエラーになるわけではありません。

- <https://console.spec.whatwg.org/>

本章ではいくつかをピックアップして紹介します。

14.1 console.log/info/warn/error()

API	Chrome 開発ツール	Safari 開発ツール	Node.js	TS PG
console.log()	✓	✓	✓	✓
console.info()	log() と同じ	✓	log() と同じ	✓
console.warn()	✓	✓	log() と同じ	✓
console.error()	✓	✓	log() と同じ	✓

それぞれ、見た目が大きく異なります。

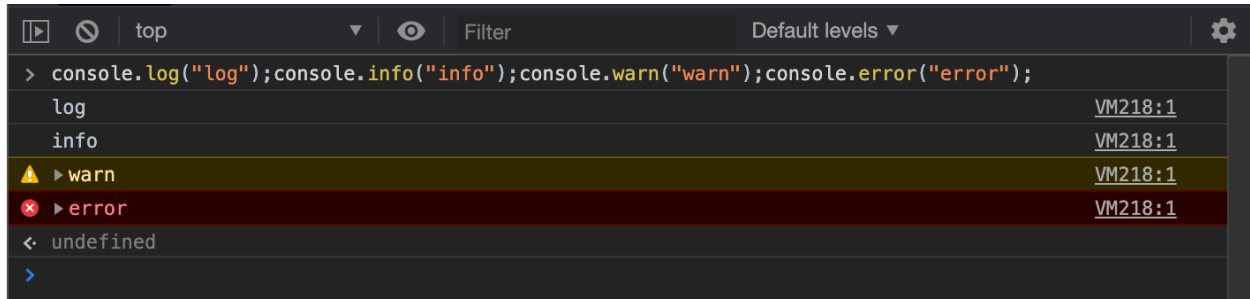


図 1 Chrome の開発ツールでの表示

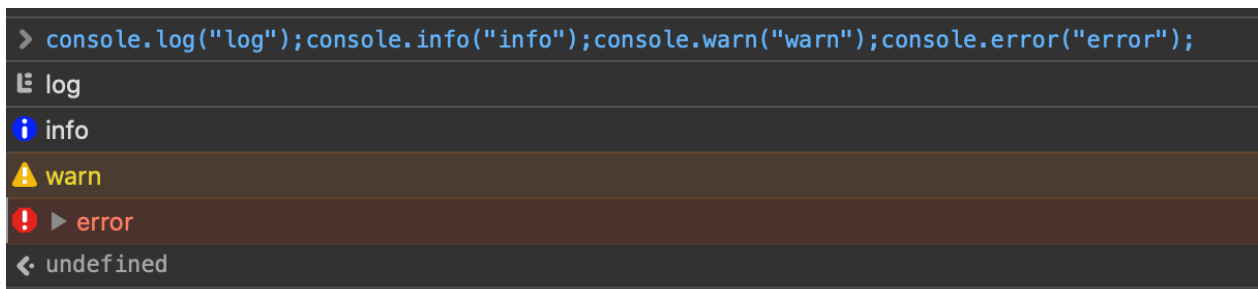


図 2 Safari の開発ツールでの表示



図 3 Node.js での表示

どの環境でも使えるテクニックですが、何かしらの変数を出力したい場合、`{ }`でくくると、変数名と値がペアになって出力されます。これはオブジェクトの短縮記法によるもので、オブジェクトの中に変数だけを書くと、その変数と同じ名前のキーに、その変数の値が格納されます。オブジェクトはキーと値をセットで出力されるため、変数名と値が両方出力されます。

```
console.log({name});  
>>> { "name": "wozozo" }
```

注釈: 2020 年 9 月リリースの Chrome 85 から挙動が変わり、コンソールの左にあるログレベルを選択した場合、選択したものだけが表示されます。エラーがあった場合にも `error` を選択しなければ表示されません。Chrome 84 以前、および 2020 年 10 月現在の Safari では選択したログレベル以上のものが表示されますので、`info` や `warn` を選択した場合にもエラーが表示されます。

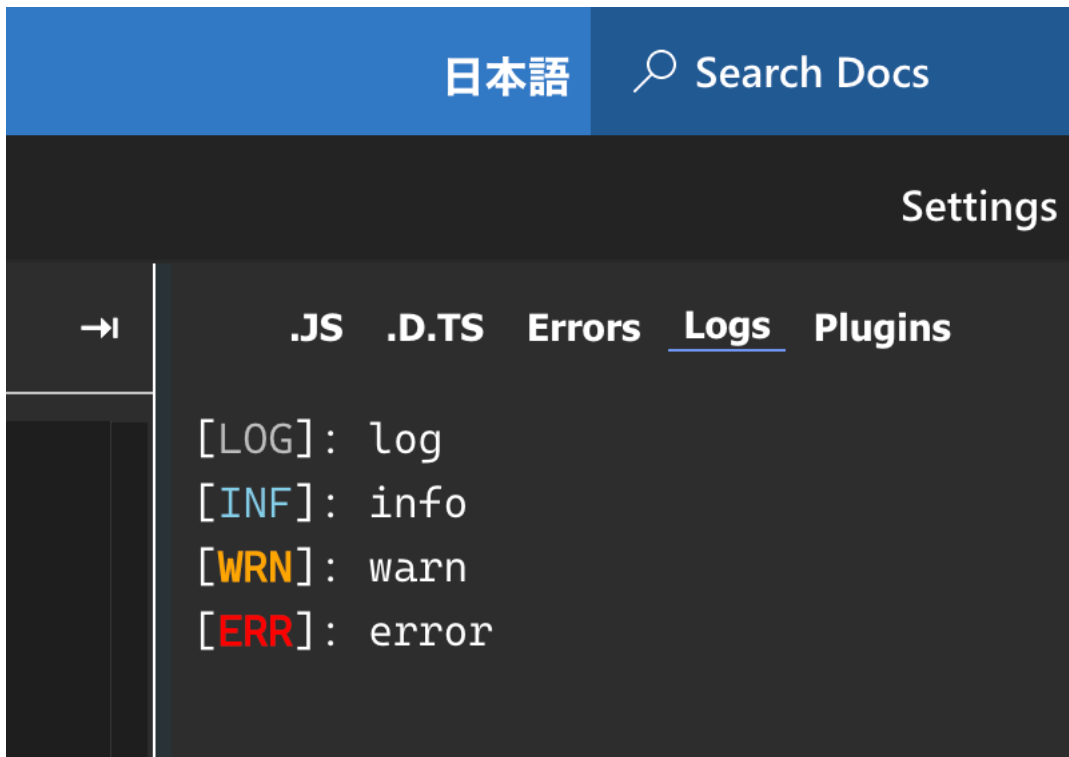


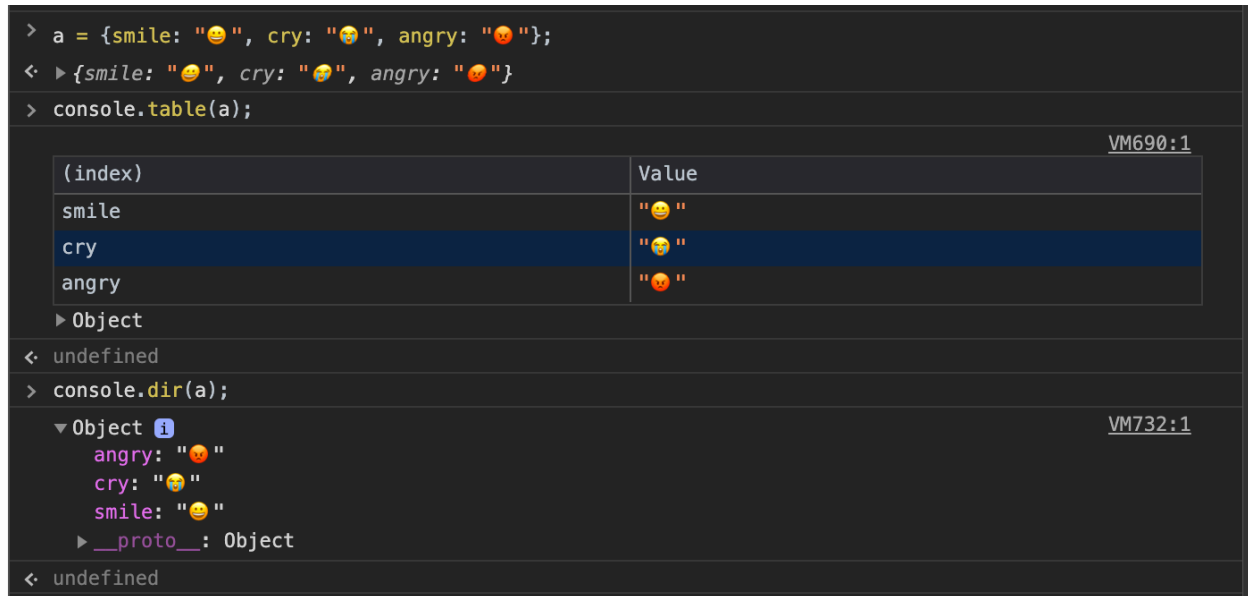
図 4 TypeScript Playground での表示

14.2 console.table/dir()

API	Chrome 開発ツール	Safari 開発ツール	Node.js	TS PG
console.table()	✓	✓	✓	
console.dir()	✓	✓	log() と同じ	

複雑なオブジェクトを見やすく表示してくれる機能です。console.table() は表形式で、console.dir() はツリー形式でデータを表示します。

ただし、この両方がフルでサポートされているのはブラウザの開発者ツールのみです。Node.js は console.table() はサポートしています。TypeScript Playground はエラーにはなりませんが、何も表示されません。

図5 `console.table()` と `console.dir()`

14.3 その他のメソッド

他にもたくさん機能はありますが、特に使う必要がないと思われるものです。

`console.assert()` は、他の言語のライブラリや組み込み機能で提供されているのと同じく、条件式が開発者の想定通りかどうかを確認する関数です。最初の引数が `true` でなければエラーとしてログに出力されます。しかし、開発者ツールでも Node.js でも、エラーが発生したことしかわかりません。実行時にエラーにするよりは、ユニットテストでカバーすべき内容でしょう。

```
console.assert(a === 1);
```

14.4 まとめ

デバッグログで使える機能を紹介しました。いろいろ紹介しましたが、基本的にはテキスト出力がユーザーインタフェースの一部となっている Node.js 以外は本番コードには残すべきものではありません。あくまでもデバッグの補助です。環境ごとに使えたり使えなかったり結果が異なる点にも注意が必要です。

第 15 章

ジェネリクス

ジェネリクスは、使われるまで型が決まらないようないろいろな型の値を受け入れられる機能を作るときに使います。ジェネリクスは日本語で総称型と呼ばれることもあります。

ジェネリクスは、ライブラリを作る人のための機能です。画面を量産する時とかには基本的には出てこないでしょう。実装していて「これはどんな子要素の型が来ても利用できる汎用的な処理だ」といったことがあればそこで初めて登場します。

15.1 ジェネリクスの書き方

ジェネリクスは、関数、インタフェース、クラスなどと一緒に利用できます。

次の関数は指定された第一引数の値を、第二引数の数だけ含む配列を作って返すコードのサンプルです。

ジェネリクスの場合は名前の直後、関数の場合は引数リストの直前に、ジェネリクスの型パラメータを関数の引数のように（ただし、対になる不等号でくる）記述します。下記のコードでは `T` がそれにあたります。関数の宣言の場合は入出力の引数や関数本体の定義時に、`T` がなんらかの型であるかのように利用できます。インタフェースやクラスの場合はメンバーのメソッドの宣言、クラスであればメンバーのフィールドやメソッドの実装の中で利用できます。

どの型が入ってくるかどうかは利用されるまではわかりませんが、`T` は実際に使うときに、全て同じ型名がここに入ります。

リスト 1 ジェネリクスの関数宣言

```
function multiply<T>(value: T, n: number): Array<T> {  
    const result: Array<T> = [];  
    result.length = n;  
    result.fill(value);  
    return result;  
}
```

`T` には `string` など、利用時に自由に型を入れることができます。宣言文と同じように `< >` で括られている中に

型名を明示的に書くことで指定できます。また、型推論も可能なので、引数の型から明示的に導き出せる場合には、型パラメータを省略することができます。

リスト 2 ジェネリクスの利用

```
// -1 が 10 個入った配列を作る
const values = multiply<number>(-1, 10);

// ジェネリクスの型も推論ができるので、引数から明示的にわかる場合は省略可能
const values = multiply("すごい!", 10);
```

15.1.1 ジェネリクスの引数名

ジェネリクスでは、型のパラメータとしては T、U、V などの大文字の文字が一般的に使われます。あるいは T1、T2 などでもいいでしょう。一般に、インスタンス名（変数名）は小文字スタートの識別子が、クラスやインタフェースは大文字の識別子が使われてきたので、呼んだ時にも直感的に理解しやすいでしょう。これは C++ や Java などでも使われてきた慣習ですので、他の言語のユーザーも慣れた方法であります。

基本的な型付けの最後の方で触れた Mapped Type の場合は、オブジェクトのキーをパラメータのように扱っていました。これは K が使われることが多いようです。

15.2 ジェネリクスの型パラメータに制約をつける

ジェネリクスの型パラメータは列挙するだけの場合はどんな型の引数も受け入れるという意味になります。しかし、何かしらの特別な型のみを受け入れたいということがあるでしょう。ジェネリクスは動的に型が決まるといっても、デフォルトでは unknown と同じように解釈されます。関数の本体の中で型パラメータのプロパティにアクセスするとエラーになります。

リスト 3 型パラメータもコンパイル時にチェックされる

```
function isTodayBirthday<T>(person: T): boolean {
    const today = new Date();
    // person の型は未知なので getBirthday() メソッドがあるかどうか未定でエラーになる
    const birthday = person.getBirthday();
    return today.getMonth() === birthday.getMonth() && today.getDate() === birthday.
    ↪ getDate();
}
```

ここでは、getBirthday() メソッドを持っている型ならなんでも受け入れられるようにしたいですね？そのようなときは、extends を使って T はこのインタフェースを満たす型でなければならないということを指定できます。

リスト 4 extends で型パラメータに制約を与える

```

type Person = {
    getBirthDay(): Date;
}

function isTodayBirthDay<T extends Person>(person: T): boolean {
    const today = new Date();
    // person の型は少なくとも Person を満たす型なので getBirthDay() メソッドが利用可能
    const birthDay = person.getBirthDay();
    return today.getMonth() === birthDay.getMonth() && today.getDate() === birthDay.
    ↪ getDate();
}

```

このように書くことで、関数定義の実装時にエラーとなることはありません。また、利用時にも、この制約を満たさない場合にはエラーになります。

文字列などの合併型も extends string で設定できます。これで何かしらの文字列のみを型パラメータに指定できます。number にすれば数値も扱えます。

```

// 何かしらの文字列とその合併型だけを受け付ける
function action<T extends string>(actionName: T) {
    :
}

action<keyof ActionList>("register");

```

extends に合併型を設定すればさらに特定の文字列だけに限定できます。

15.3 型パラメータの自動解決

TypeScript の処理系は入力値の型などから型パラメータを推論しようとします。すべての型が解決可能であれば、型パラメータの指定を省略できます。また、型パラメータ同士で影響を与え合う（制約を与え合う）ような型パラメータの制約も書くことができます。その場合も、お互いの情報や引数の情報を元に、お互いに推論できることから推論していった、自動解決できるものを解決していきます。

次の setValue は何やら不思議な型定義になっています。このうち、T はオブジェクトの型、K はオブジェクトのプロパティ名の合併型で、U はオブジェクトのプロパティの方の型を表しています。やっていることは、オブジェクトの型にマッチした代入をするだけのなんの変哲も無い（役に立たない）コードです。

リスト 5 値の設定を大げさに書く

```

function setValue<T, K extends keyof T, U extends T[K]>(obj: T, key: K, value: U) {
    obj[key] = value;
}

```

Visual Studio Code や TypeScript の Playground のページで次の setValue 呼び出しを書いてみてください。まず、最初の引数に park をタイプすると、型 T が決まります。そうすると、ポップアップする引数 key の型は "name" | "hasTako" に、value の型は string | boolean になります。次に、二つ目の引数に "name" をタイプすると、value の型は string となります。このように連鎖的にパズルを解くように TypeScript の処理系は型の制約を解決していきます。

リスト 6 エディタの補完を試してみよう

```
const park: ParkForm = {
  name: "恵比寿東",
  hasTako: true
};

setValue(park, "name", "神明児童遊園");
```

ただし、型パラメータで設定することを期待しているのか、それとも引数だけからすべてを解決していけるように設計されているのかは一目見て理解するのは難しいので、こういった意図のコードになっているのかはドキュメントやサンプルコードで伝えるようにしたほうが良いでしょう。

15.4 ジェネリクスのできることで、できないこと

ジェネリクスのできることを一言で言えば、利用する側の手間を減らしつつ、型チェックをより厳しくすることで。ジェネリクスを使うと、引数の型によって返り値の型が変わるとか、最初の引数の型によって、別の引数の型が変わるとか、そういったことが実現できます。また完全に自由にではなく、特定の条件を満たす型パラメータのみを受け取ることも指定できましたよね。

一方でできないこともあります。C++ のテンプレートのように、指定された型によってロジックを切り替えるといったことはできません。例えば、要素の型とで、要素数が型パラメータで設定できる固定長配列などはジェネリクスやテンプレートで簡単に実現できます。C++ の場合は、例えば要素が 32 ビットの数値で要素数が 4 の場合だけ SIMD を使って足し算を高速化するという「特殊化」ができますが、ジェネリクスではそのようなことはできません。

また、即値の数値を型パラメータに入れることも C++ ではできましたし、その演算もできます。C++ では特殊化と組み合わせて、次のような数学の漸化式のような型定義もできます。これにより、4 次元配列でも 5 次元配列でも簡単に作り出すことが C++ では可能ですし、これを駆使したテンプレートメタプログラミングという技法も編み出されましたが、これも TypeScript には不可能です。

- n 次元配列は n-1 次元配列の配列
- 1 次元配列は普通の配列（特殊化）

TypeScript の文法のうち、型宣言などの JavaScript から追加されたものは、基本、そのまま切り落とせば単なる JavaScript になる、というのが原則としてありました。ジェネリクスについても同様ですので、型で実装を分岐という JavaScript にはないことはできません。

15.5 型変換のためのユーティリティ型

TypeScript では組み込みの型変換のためのジェネリクスユーティリティ型を提供しています。詳細なリファレンスは [本家のハンドブック](#) 中の [Utility Types](#) にあります。

15.5.1 オブジェクトに対するユーティリティ型

T に定義済みのオブジェクトを指定することで、特定の変更を加えた新しいオブジェクトの型が定義されます。

リスト 7 オブジェクトに対するユーティリティ型の使い方。

```
const userDiff: Partial<User> = {  
  organization: "Future Corporation"  
};
```

- `Partial<T>`: 要素が省略可能になった型
- `Readonly<T>`: 要素が読み込み専用になった型
- `Required<T>`: `Partial<T>` とは逆に、すべての省略可能な要素を必須に直した型

15.5.2 オブジェクトと属性名に対するユーティリティ型

次の 3 つの型は T 以外に、K としてプロパティの文字列の合併型を持ち、新しいオブジェクトの型を作ります。

リスト 8 オブジェクトと属性名に対するユーティリティ型の使い方。

```
const viewItems: Pick<User, "name" | "gender"> = {
  name: "Yoshiki Shibukawa",
  gender: "male"
};
```

- Record<K, T>: T を子供の要素に持つ Map 型のようなデータ型 (K がキー) を作成。
- Pick<T, K>: T の中の特定のキー K だけを持つ型を作成
- Omit<T, K>: T の中の特定のキー K だけを持たない型を作成

15.5.3 型の集合演算のユーティリティ型

次の3つの型は、T と U (NonNullable<T> 以外) として、合併型をパラメータとして受け、新しい合併型を作り出します。

リスト 9 型の集合演算のユーティリティ型の使い方。

```
const year: NonNullable<string | number | undefined> = "昭和";
```

- Exclude<T, U>: T の合併型から、U の合併型の構成要素を除外した合併型を作る型
- Extract<T, U>: T の合併型と、U の合併型の両方に含まれる合併型を作る型
- NonNullable<T>: T の合併型から、undefined を抜いた合併型を作る型

15.5.4 関数のユーティリティ型

関数を渡すと、その戻り値の型を返すユーティリティ型です。

- ReturnType<T>

15.5.5 クラスに対するユーティリティ型

クラスに対するユーティリティ型です。あまり使うことはないと思われます。

- ThisType<T>: JavaScript 時代のコードは this が何を表すのかを外挿できましたのでそれを表現するユーティリティ型です。新しい型は作りません。--noImplicitThis がないと動かないとのこと。
- InstanceType<T>: InstanceType<typeof C> が C を返すとドキュメントに書かれていますが用途はよくわかりません。

15.6 any や unknown 、合併型との違い

未知の型というと、any や unknown が思いつくでしょう。また、複数の型を受け付けるというと、合併型もあります。これらとジェネリクスの違いについて説明します。

any や unknown の変数に値を設定してしまうと、型情報がリセットされます。取り出すときに、適切な型を宣言してあげないと、その後のエラーチェックが無効になったり、エディタの補完ができません。

次の関数は、初回だけ指定の関数を読んで値を取って来るが、2 回目以降は保存した値をそのまま返す関数です。初回アクセスまで初期化を遅延させます。

リスト 10 any 版の遅延初期化関数

```
function lazyInit(init: () => any): () => any {
  let cache: any;
  let isInit = false;
  return function(): any {
    if (!isInit) {
      cache = init();
    }
    return cache;
  }
}
```

any 版を使って見たのが次のコードです。

リスト 11 非ジェネリック版の使い方

```
const getter = lazyInit(() => "initialized");
const value = getter();
// value は any 型なので、上記の value の後ろで . をタイプしてもメソッド候補はでてこない
```

この場合、cache ローカル変数に入っているのは文字列ですし、value にも文字列が格納されます。しかし、TypeScript の処理系は any に入るだけで補完をあきらめてしまいます。

次のジェネリクス版を紹介します。ジェネリクス版は入力された引数の情報から返り値の型が正しく推論されるため、返り値の型を使うときに正しく補完できます。

リスト 12 any 版の遅延初期化関数

```
function lazyInit<T>(init: () => T): () => T {
  let cache: T;
  let isInit = false;
  return function(): T {
    if (!isInit) {
      cache = init();
    }
    return cache;
  }
}
```

(次のページに続く)

(前のページからの続き)

```
}
```

リスト 13 ジェネリック版の使い方

```
const getter = lazyInit(() => "initialized");
const value = getter();
// value は string 型なので、上記の value の後ろで . をタイプするとメソッド候補が出てくる
```

合併型についても、型の補完時に余計な型情報がまざってしまうため、型ガードで必要な型である保証が必要です。また、ジェネリクスには2つの引数があって両方の型が同じ、という保証もしやすいメリットがあります。

15.7 まとめ

ジェネリクスについて紹介しました。

基本的に、画面を量産するという仕事ではなく、共通ライブラリを作り出すとか、そういったタスクで活躍する中級向けの機能です。作り込めば作り込むほど、使う人にやさしく、間違った情報が入れにくい関数やクラス、インタフェースが作れます。

一方で、型情報の作り込みは読みにくいコードに直結します。書いているときには良いのですが数日後にいじるのが少し難しいコードになりがちです。凝った正規表現に近いものがあると思います。

第 16 章

関数型指向のプログラミング

JavaScript の世界には長らく、クラスはありませんでした。プロトタイプを使ったクラスのようなものはありましたが、Java などに慣れた人からは不満を持たれていました。プロトタイプが分かっていたら、インスタンス作成能力には問題はなく、親を継承したオブジェクトも作れるため、能力が劣っていたわけではありましたが、オブジェクト指向としては使いにくい。そんなふうと言われることも多々ありました。

一方で昔から JavaScript で関数型プログラミングを行おう、という一派はそれなりにいました。

JavaScript の出自からして、Scheme という関数型言語が好きだったブレンダン・アイクです。開発時の会社の方針で Java 風の文法を備え、Java に影響を受けた言語にはなっていますが、Java のような静的型付けのオブジェクト指向のコンパイル言語とはやや遠い、プロトタイプ指向のインタプリタになっています。関数もオブジェクトとして、変数に入れたり自由に呼び出したりできる一級関数ですし、無名関数を気軽に作ったり、その関数が作られた場所の変数を束縛するクロージャとしても使えたり、関数型言語の要素も数多く備えています。

おそらく、jQuery も関数型の思想で作られたのではないかと思います。オライリーからも、2013 年に『JavaScript で学ぶ関数型プログラミング』の原著が出ています。

関数型の言語を触ったことがない人も、特に怖がる必要はありません。JavaScript で実現できる関数型プログラミングのテクニックはかなり限られています。ループを再帰で書くと、すぐにスタックを使い果たしてエラーになります。関数型でコードを書くと言っても、あまり極端なことはできずに、オブジェクト指向と組み合わせたハイブリッドな実装方法になりますし、バグを産みにくいコードを書くための指針集といった趣になります。

16.1 イミュータブル

イミュータブル (immutable) は「変更できない」という意味です。データを加工するのではなく、元のデータはそのままに、複製しつつ変化させたバージョンを作ります。ここで活躍するのが、配列のメソッドの、`map()`、`forEach()`、`filter()` です。

たとえば、1 から 10 までの数値が入った配列があったとして、それぞれの値を 2 倍にした配列を作ります。

```
// 元のデータ
const source = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
// 2倍の要素を作る
const doubles = source.map(value => {
  return value * 2;
});
```

変換処理の関数を書き、`map()` に渡すと、要素 1 つずつをその関数にわたし、その結果の配列を新たに作って返します。

- `map()`: 元のデータの各要素に何かしらの加工を行った新しい配列を作る
- `forEach()`: 元のデータの各要素に対するループを行う
- `filter`: 元のデータの書く要素を評価し、選択されたものだけを保持する新しい配列を作る

これは配列の例ですが、複合型の場合に値そのものを書き換えるのではなく、書き換えた別のバージョンを作っていくのがポイントです。オブジェクト指向はデータと操作が結びついていますが、関数型はデータとそれに対する操作を切り離すと共に、データを複製するという異なったアプローチを取ります。実行効率的にもったいないのでは、と思われる方もいると思いますが、JavaScript エンジン開発競争のおかげで、JavaScript エンジンも動的言語の中ではトップクラスのパフォーマンスを持ちます。そこそこ大きなデータであっても高速に処理ができます。多少ペナルティがあったとしても、管理のしやすさや安全を重視してこの方法を取ります。

<https://www.npmjs.com/package/iterative>

第 17 章

クラス上級編

本章では、アプリケーション開発者は使わないかもしれないが、ライブラリやフレームワーク開発者が使うかもしれない機能を紹介します。

17.1 アクセッサ

プロパティのように見えるけど、実際には裏でメソッド呼び出しが行われ、ちょっとした気の利いたをできるようにすることをするのがアクセッサです。メンバーのプロパティへの直接操作はさせないが、その読み込み、変更時に処理を挟むといったことが可能です。登場する概念としては基本的には次の3つです。get だけを設定すれば取得のみができる読み込み専用とかもできます。

- 外部からは見えない `private` なプロパティ
- 値を返すゲッター (getter)
- 値を設定するセッター (setter)

どちらかというと、ライブラリ実装者が使うかもしれない文法です。アプリケーションユーザーの場合、Vue.js の TypeScript のクラス用の実装方法では、ゲッターを `computed property` として扱いますので、Vue.js ユーザーに関しては積極的に使うことになるでしょう。

例えば、金額を入れたら、入り口と出口でビット演算で難読化（と言えないような雑な処理ですが）をする銀行口座クラスは次のようになります。

リスト 1 アクセッサ

```
class BankAccount {  
  private _money: number;  
  
  get money(): number {  
    return this._money ^ 0x4567;  
  }  
}
```

(次のページに続く)

```
set money(money: number) {
    this._money = money ^ 0x4567;
}

const account = new BankAccount()

// 1000 円入れた！
account.money = 1000;

// 表示すると...
console.log(account);
//   BankAccount { _money: 18063 }

// 金額を参照すると正しく出力
console.log(account.money);
// 1000
```

Java とかでよく使われるユースケースは、`private` でメンバー変数を用意し、それに対する `public` なアクセッサを用意するというものです。Java にはこのような TypeScript と同等のアクセッサはなく、`getField()` `setField()` という命名規則のメソッドがアクセッサと呼ばれています。Java はバイトコードエンハンサーを使って処理を挟むなどをすることもあって、重宝されています。

TypeScript (の元になっている JavaScript) にはプライベートフィールドは ES2019 まではありませんでした。メンバーフィールドもメソッドも、同じ名前空間に定義されます。 `name` というプロパティを設定する場合、プライベートなメンバーの名前が `name` だとすると、アクセッサ定義時に衝突してしまいます。メンバーフィールド名には `_` を先頭につけるなどして、名前の衝突を防ぐ必要があります。

ですが、JavaScript の世界では「すべてを変更しない、読み込み専用オブジェクトとみなして実装していく」という流れが強くなっていますし、もともと昔の JavaScript では定義するのが面倒だったり、IDE サポートがなかったためか、Java のオブジェクト指向的なこの手のアクセッサを逐一実装する、ということはあまり行われません。属性が作られた時から変更がないことが確実に分かっているなら `readonly` の方が良いでしょう。

アクセッサとして書く場合、ユーザーは単なるプロパティと同等の使い勝手を想定して利用してきます。そのため、あまりに変な動作を実装することはしないほうが良いでしょう。また、TypeScript では普通にプロパティを直接操作させるのを良しとする文化なので、アクセッサを使うべき場面はかなり限られます。筆者が考える用途は次の用途ぐらいでしょう。

- 通常の型システムではカバーできない入力値のチェック (例えば、正の整数のみで、負の数はエラーにする)
- 出力時の正規化 (テキストはすべて半角の小文字にまとめるなど)
- 過去に実装されたコードとプロパティ名が変わってしまい、別名 (エイリアス) を定義したい

17.2 抽象クラス

インタフェースとクラスの間ぐらいの特性を持つのが抽象クラスです。インタフェースとは異なり実装を持つことができます。メソッドに `abstract` というキーワードをつけることで、子クラスで継承しなければならないメソッドを決めることができます。子クラスで、このメソッドを実装しないとエラーになります。`abstract` メソッドを定義するには、クラスの宣言の前にも `abstract` が必要です。

リスト 2 抽象クラスは実装も渡せるインタフェース

```
abstract class Living {
  abstract doMorningTask(): void;
  doNightTask() {
    console.log("寝る");
  }
}

class SalaryMan extends Living {
  doMorningTask() {
    console.log("山手線に乗って出勤する");
  }
}

class Dog extends Living {
  doMorningTask() {
    console.log("散歩する");
  }
}
```

Java ではおなじみの機能ですが、TypeScript で使うことはほぼないでしょう。

17.3 まとめ

クラスの文法のうち、アプリケーション開発者が触れる機会がすくないと思うものを取り上げました。

- アクセッサ
- 抽象クラス

第 18 章

リアクティブ

第 19 章

高度なテクニック

課題: デコレータを使って DI。他にある？

<https://github.com/Microsoft/tsyringe>

第 20 章

ソフトウェア開発の環境を考える

すでに開始しているフロントエンドの開発に加わる場合、まず文法ぐらいが分かって入れれば開発に入ることができるでしょう。しかし、立ち上げる場合は最低 1 人は環境構築ができるメンバーが必要になります。本章以降はその最低 1 人を育てるための内容になります。

近年では JavaScript は変換をしてデプロイするのが当たり前になってきている話は紹介しました。フロントエンド用途の場合、規模が大きくなってきているため数多くのツールのサポートが必要になってきています。

- コンパイラ

TypeScript で書かれたコードを JavaScript に変換するのがコンパイラです。基本的には TypeScript 純正のものを使うことになるでしょう。一部の Babel を前提としたシステムでは Babel プラグインが使われることがあります。これは型定義部分を除外するだけで、変換は Babel 本体で行います。ネイティブコードにしてから WebAssembly に変換する AssemblyScript というものもあったりします。

- TypeScript
- @babel/plugin-transform-typescript
- AssemblyScript

- テスティングフレームワーク

ソフトウェアを、入力と出力を持つ小さい単位に分けて、一つずつ検証するために使います。ブラウザの画面の操作をエミュレートし、結果が正しくなるかを検証する end-to-end テストもあります。

- Jest
- Ava
- Mocha
- Jasmine

- 静的チェックツール

近年のプログラミング言語は、他の言語の良いところを積極的に取り入れたりして機能拡張を積極的に行なっています。それにより、古い書き方と、より良い書き方のいくつかの選択肢がある場合があります。間違いやすい古い書き方をしている箇所を見つけて、警告を出すのが静的チェックツールです。TSLint が今まで多く使われていましたが、Microsoft が、TSLint では構造的にパフォーマンスが出にくいとのことで、ESLint をバックアップしていくという発表しました。そのため、選択肢としては現在はこれ一択です。

- eslint
- コードフォーマッター

JavaScript の世界では、以前は静的チェックツールの 1 機能として行われることが多かったのですが、最近では、役割を分けて別のツールとなっています。

- Prettier
- TypeScript Formatter
- gts
- タスクランナー

ソフトウェア開発ではたくさんのツールを組み合わせる必要がありますが、その組み合わせを定義したり、効率よく実行するのがタスクランナーです。色々なツールがでてきましたが、現在は実行自体は npm で行い、変更があったファイルだけを効率よくビルドするのはバンドラー側で行うと、役割分担が変わってきています。npm scripts は標準機能ですが、少し機能が弱いので、npm-run-all を組み合わせて少し強化して使うことがあります。本ドキュメントでもそのようにします。

- npm scripts
- gulp
- grunt
- バンドラー

コンパイラが変換したファイルを最終的に結合したり、動的ロードを有効にするのがバンドラーです。次のようなものがあります。なお、ライブラリなど、バンドラーは行わず、コンパイラだけ実行した状態で配布することもできます。

- Webpack
- parcel
- rollup

さらに規模が大きくなり、ビルドしてリロードして起動、というステップに時間がかかるようになってくると、ビルドサーバーを前面にたてて開発にかかる待ち時間を減らすことも当たり前に行われます。

ビルドサーバーはコンパイル済みのファイルを結果をメモリにキャッシュします。ローカルのファイルを監視し、変更があったらそのファイルだけを更新し、JavaScript にコンパイルします。ビルドサーバーによっては、開発ビ

ルドではサーバーに変更があったかを問い合わせるコードを埋め込んでおくものもあります。その機能を使うと、エディタでファイルを保存すると、コンパイルが自動で走り、ブラウザが開発サーバーから変更された情報を受け取り、ブラウザに変更されたコードをロードし直すまで自動で行われたりします。

20.1 環境構築できるメンバーがいなかったときはどうするか？

一時的に構築できるメンバーにヘルプに入ってもらって環境整備をやってもらう、ということも方法としてはありますが、可能であれば常に質問できる人が望ましいです。もちろん、Webpack や Babel を自分で設定する覚悟があるのであれば問題はありますが、一時的に誰かに高度な環境を作ってもらうと、やはりそこがタコツボ化してしまい、メンテナンスができなくなりがちです。あまり背伸びをしてしまうと、そこが負債になりがちです。

環境構築をしても、一度きりでは終わりません。ライブラリが定期的にバージョンアップしていく作業も発生しますし、何かしらの新しいツールやライブラリを付け加えるという作業も発生します。環境周りの面倒を見れるメンバーが一人はプロジェクトにはいることが望ましいです。

もしも誰もいないかつ React であれば Next.js を使い、てっとり速く作れる環境を利用すべきです。Vue や Angular はデフォルトのプロジェクト作成である程度揃うので、それを活用しましょう。そして、環境設定はなるべくがんばらず、デフォルトのままで頑張れるだけ頑張るのが良いでしょう。バージョンアップ時は、新しくプロジェクトを作り、そこに既存のコードを持ってきて式年遷宮をする、というのがトラブルが少ないと思われます。

Next.js であれば、ある程度のベストプラクティスに従って設定はされているし、そこからの機能追加も情報が得やすいので、無難な落としどころとなります。

第 21 章

基本の環境構築

環境構築の共通部分を紹介しておきます。

プロジェクトでのコーディングであれば、誰が書いても同じスタイルになるなど、コード品質の統一が大切になりますので、単なる個人用の設定ではなく、それをシェアできるというのも目的として説明していきます。

ここでは、基本的にすべてのプロジェクトで Jest、ESLint、Prettier などを選択しています。まあ、どれも相性問題が出にくい、数年前から安定して存在している、公式で推奨といった保守的な理由ですね。きちんと選べば、「JS はいつも変わっている」とは距離を置くことができます。

- Jest

テストフレームワークはたくさんありますが、ava と Jest がテスト並列実行などで抜きん出ています。Jest は TypeScript 用のアダプタが完備されています。ava は Babel/webpack に強く依存しており、単体で使うなら快適ですが、他の Babel Config と相性が厳しくなるので Jest にしています。

- ESLint

公式が押しているのがこれですね。かつては TSLint というツールがデファクトスタンダードでしたが、最適化の限界があることなどを理由に、TypeScript が公式に ESLint をメインの Linter とすることが宣言されました。よほどの理由がないかぎり ESLint を使うべきです。

ESLint はプラグインを使うことでさまざまなルールを足せますし、プラグインを通じて設定を一括で変更できます。これはツール同士の干渉を避けるためだったり、推奨のルールセットを配布する手段として利用されています。

- @typescript-eslint/eslint-plugin

ESLint に TypeScript の設定を追加するプラグインです

- Prettier

TypeScript 以外の SCSS とかにも対応していたりします。現在はシェアが伸びています。EditorConfig というエディタ向けの共通の設定ファイル（ただし、それぞれのエディタでプラグインが必要）を読み込んで利用できます。

– eslint-config-prettier

eslint 側で、Prettier と衝突する設定をオフにするプラグインです

- npm scripts

ビルドは基本的に Makefile とか gulp とか grunt とかを問わず、npm scripts で完結するようにします。ただし、複数タスクをうまく並列・直列に実行する、ファイルコピーなど、Windows と他の環境で両対応の npm scripts を書くのは大変なので、mysticatea さんの Qiita のエントリーの npm-scripts で使える便利モジュールたちを参考に、いくつかツールを利用します。

- Visual Studio Code

TypeScript 対応の環境で、最小設定ですぐに使い始められるのは Visual Studio Code です。しかも、必要な拡張機能をプロジェクトファイルで指定して、チーム内で統一した環境を用意しやすいので、推奨環境として最適です。Eclipse などの IDE の時代とは異なり、フォーマッターなどはコマンドラインでも使えるものを起動するケースが多いため、腕に覚えのある人は Vim でも Emacs でもなんでも利用は可能です。

課題: lynt の TypeScript 対応状況を注視する

これらのツールを組み合わせ環境を作っていきます。昔の JavaScript の開発環境は、ビルドツール（Grunt や Gulp）を使ってこれらのツールの組み合わせを手作業で行う必要がありましたが、ツールが複雑になってプロジェクトのテンプレート化が進んだことや、TypeScript の人気が上がるにつれて TypeScript を考慮したツールが増えてきました。そのため、現在は比較的簡単に導入できます。

ツールは日々改良されているため、どの目的に対してどのツールを利用するのがベストかは一概には言えませんが、比較的広く使われていて、安定していて、かつ利便性が高い物はいくつかピックアップできます。次の表はその対応表になります。フォルダ作成は package.json を含むプロジェクトのフォルダを作成することを意味します。

最終的に TypeScript の処理系が何かしらのツール経由で呼ばれ、それをそのツールが結合し、ブラウザや Node.js で直接実行できる JavaScript ファイルになる部分は変わりません。これらのツールを使ってプロジェクトを作ると、TypeScript コンパイラを直接呼び出さずにビルドできます。

表 1 TypeScript

対象	ツール	フォルダ作成	TS 設定	Lint 等の設定
Node.js (CLI/Server)	ts-node	•	○	•
Node.js (CLI/Server)	@vercel/ncc	•	○	•
React	create-react-app	○	(オプション)	•
React + SSR	Next.js	○	(オプション)	•
Vue.js	@vue/cli	○	(オプション)	(オプション)
Angular	@angular/cli	○	○	○
ウェブ全般	Parcel	•	○	•

本章では、これらのツール間で共通となる情報を紹介します。

21.1 作業フォルダの作成

出力先フォルダの作成はプロジェクト構成ごとによって変わってくるため、入力側だけをここでは説明します。プロジェクトごとにフォルダを作成します。ウェブだろうがライブラリだろうが、`package.json` が必要なツールのインストールなど、すべてに必要なため、`npm init` でファイルを作成します。ツールによってはこのフォルダの作成と `package.json` の作成を勝手にやるものもあります。

```
$ mkdir projectdir
$ cd projectdir
$ npm init -y
$ mkdir src
$ mkdir __tests__
```

外部に公開しないパッケージの場合には、`"private": true` という設定を忘れずにいれましょう。

`src` フォルダ以下に `.ts` ファイルを入れて、出力先のフォルダ以下にビルド済みファイルが入るイメージです。仮にこれを `dist` とすると、これは `Git` では管理しませんので `.gitignore` に入れておきます。

リスト 1 .gitignore

```
dist
.DS_Store
Thumbds.db
```

もし成果物を配布したい場合は、それとは逆に、配布対象は `dist` とルートの `README` とかだけですので、不要なファイルは配布物に入らないように除外しておきましょう。これから作る TypeScript の設定ファイル類も外して起きましょう。

リスト 2 .npmignore

```
dist
.DS_Store
Thumbds.db
__tests__/
src/
tsconfig.json
jest.config.json
.eslintrc
.travis.yml
.editorconfig
.vscode
```

21.2 TypeScript の環境整備

まずは第一歩として TypeScript のコンパイラを入れます。これも環境によっては最初から入っているものもあります。

```
$ npm install --save-dev typescript
```

設定ファイルは以下のコマンドを起動すると雛形を作ってくれます。これを対象の成果物ごとに編集していきます。

```
$ npx tsc --init
```

設定ファイルの詳細や、TypeScript コンパイラを呼び出す部分は各開発環境の章で取り扱います。TypeScript の処理系は、上記のサンプルの通り、`tsc` コマンドですが、これを直接使うことはありません。大抵は `webpack` などのバンドラー経由で使います。コンパイラは単体のファイルの変換機能しかありませんが、TypeScript が利用される環境のほとんどは、1 ファイルにバンドルして配布します。返還後のファイルを塊のファイルにまとめたり、コンパイル結果をメモリ上にキャッシュし、変更のあったファイルだけを素早く変換してプレビューしたりと、コンパイルを行う部分をラップして、よりスマートにビルドします。

21.3 Prettier

コードフォーマッターはコードを自動整形するツールです。チーム内で書き方が統一されるため、レビューアの負担は減ります。また、全員が同一のフォーマットで編集するため、コンフリクトが減ったり、コンフリクトの修正作業も楽になります。一方で、後から導入しようとする、かなりの差分とコンフリクトが発生することもあります。そのため、このフォーマッターはプロジェクト開始時に忘れずに設定しておきましょう。TypeScript で一番人気は Prettier です。

コードフォーマッターは2つのツールを使います。といっても、Prettier の処理系が2つのフォーマッターを利用できるため、インストールするツールは Prettier のみです。Prettier は自身のルール以外に、editorconfig というフォーマットのためのルール集も利用できるため、この2つを整備します。

まず、最低限、文字コード、インデントとか改行コードの統一はしたいので、editorconfig の設定をします。editorconfig を使えば Visual Studio、Vim など複数の環境があってもコードの最低限のスタイルが統一されます(ただし、各環境で拡張機能は必要)。また、これから設定する Prettier もこのファイルを読んできます。

```
$ npm install --save-dev prettier
```

リスト 3 .editorconfig

```
root = true

[*]
indent_style = space
indent_size = 4
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true
```

Prettier の設定ファイルも作成します。シングルクォートの有無、末尾のカンマ、セミコロンの有無などが設定できます。詳しくは [Prettier のドキュメント](#) に説明があります。この説明では EditorConfig の設定との併用を紹介しましたが、Prettier 単体の設定ファイルにも同様の機能があるので1つにまとめることも可能です。

リスト 4 .prettierrc

```
{
  "trailingComma": "es5",
  "semi": true,
  "singleQuote": true
}
```

次に package.json のスクリプトに組み込んで実行してみましょう。エディタへの組み込みを行う場合も、必ず単体で実行できることを確認してから設定すると、設定ミスで時間を溶かすことが減るでしょう。あとから ESLint と組み合わせて実行するため、ここでは修飾子付きのコマンド名をあえて設定しています。検証対象のファイルはすべて src フォルダにあるものとします。

リスト 5 package.json

```
{
  "scripts": {
    "fix:prettier": "prettier --write src",
    "lint:prettier": "prettier --check src"
  }
}
```

コマンドラインで実行してみて、わざとクオート記号を違うのを設定して、正しく問題が発見できるかみてみましょう。OK なら、次は修正も試してみましょう。なければ次のステップに進みましょう。

```
$ npm run lint:prettier
Checking formatting...
[warn] src/main.ts
[warn] Code style issues found in the above file(s). Forgot to run Prettier?

$ npm run fix:prettier
src/main.ts 184ms
```

21.3.1 Visual Studio Code の設定

Visual Studio Code から利用する場合は、拡張機能と、その設定をファイルに記述しておきます。まずは拡張機能です。

Visual Studio Code でフォルダを開いたときに、必要な拡張機能がインストールされるようにします。`.vscode` フォルダにファイルを作ることで、プロジェクトのソースコードと一緒に、プロジェクトの共有設定を共有できます。同じ拡張機能を入れてもらって、コードチェックなどのクオリティを統一し、コードインテグレーション時に無駄な調整をしなくて済むようにできます。ここではついでにコードのスペルチェックの拡張機能も入れておきます。

この設定はこの JSON を書いても良いですし、拡張機能のページで該当する拡張機能を開いてから、コードパレットで `Extensions: Add to Recommended Extensions (Workspace Folder)` を選択すると追加されます。

リスト 6 .vscode/extensions.json

```
{
  "recommendations": [
    "esbenp.prettier-vscode",
    "streetsidesoftware.code-spell-checker"
  ],
  "unwantedRecommendations": []
}
```

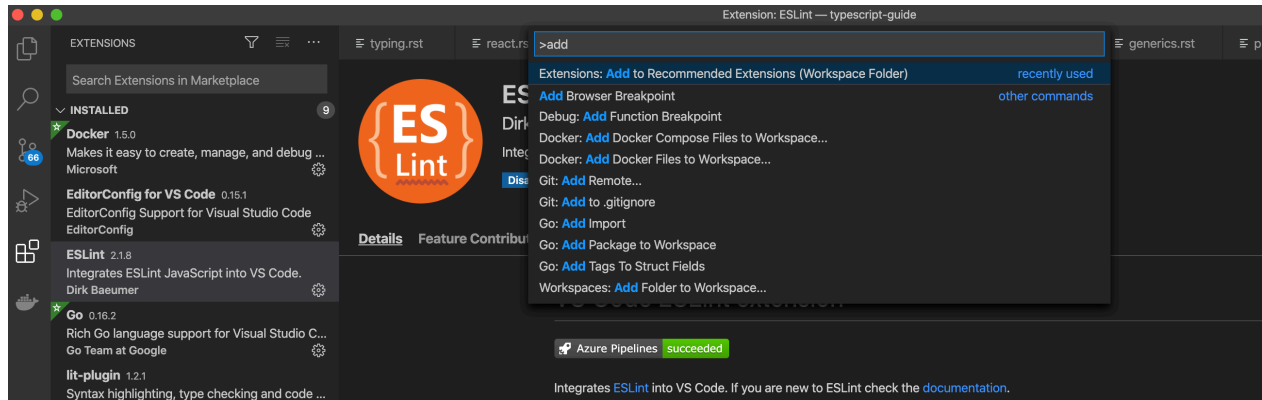


図 1 拡張機能をプロジェクト推奨に設定

インストールができれば、次はその拡張機能の設定をします。こちらでもプロジェクトのリポジトリにファイルを入れておくことでプロジェクトメンバー間で共通の設定をシェアできます。

Prettier を標準のフォーマッターに指定し、VSCode 自身の実行メカニズムを利用してファイル保存時にフォーマットがかかるようにします。

リスト 7 .vscode/settings.json

```
{
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.formatOnSave": true
}
```

注釈: 古い説明の中には、この `editor.formatOnSave` を `false` にして、ESLint の実行時に同時にフォーマッターが稼働するように設定する人がいます。しかし、可能ならタイピングのたびに稼働して欲しい Linter と同時に毎回フォーマットをかけるのは実行効率や応答性の面でよくありません。本節のように、ESLint はなるべく軽量にしておいて、ファイル保存時のみフォーマッターが稼働するようにすると、応答性が上がります。

21.4 ESLint

次に ESLint 関連ツールをインストールして設定します。ESLint は開発するプロジェクトに応じて、さまざまな環境向けにプラグインや事前に設定されたコンフィグをロードして整備していきます。ここで入れるのは TypeScript をチェックできるようにするとともに、Prettier と喧嘩しないようにするための最低限のプラグインです。テストフレームワーク向けの設定などはそれぞれ追加のインストールや設定が必要となります。なお、Vue.js の場合はそこから ESLint を有効化できますので、そちらを利用してください。

ESLint のインストールと設定はウィザードで作ります。

```
$ npx eslint --init
```

最初に聞かれる三択の質問は **To check syntax and find problems** を選びましょう。最後のコードスタイルは AirBnB スタイルとか Google スタイルなどを選んでプロジェクトに適用するコードスタイルを決定できますが、すでに Prettier を使って設定済みですので不要です。

モジュール形式は Common.js か ES6 modules か、使う場合は React か Vue か、Node.js なのかブラウザなのか、TypeScript を使うのかあたりを聞かれます。設定ファイルをどの形式で出力するか、最後に必要なパッケージを npm でインストールするかも聞かれます。モジュール形式は ES6 modules を、TypeScript の利用は Y を、設定ファイルの形式は JavaScript を、ツールのインストールは Y を選択します。ウェブのフロントエンド、ブラウザ向けか Node.js か向けかは環境に応じて選択してください。これインストールと設定は 8 割がた完了です。

ESLint の設定は、機能を追加するプラグインと、設定をまとめて変更する extends、プロジェクト内部で個別に機能を切り替えるのは rules に書きます。次のサンプルはブラウザ & React、TypeScript で生成したものに、Prettier 関連の extends を 2 つ追加したのと（必ず末尾におくこと）、個別ルールで、開発時のみ `console.log()` を許可するように、返り値の型推論を許可しています。また、コールバック関数の利用でよくあるのですが、未使用引数で出る警告はライブラリ側の都合で避けようがなかったりするため、アンダースコアで始まる名前の変数に関しては未使用でも警告が出ないようにしています。

ESLint と Prettier でオーバーラップしている領域があり、ここで追加した extends はそれらの設定が喧嘩しないようにするためのもので、ESLint 側の重複機能をオフにします。React 拡張を作成する場合は、React バージョンの設定をしないと警告を毎回見ることになるでしょう。

先ほどの初期化でほとんどのツールはインストール済みですが、Prettier との連携用設定のパッケージは入っていないので追加します。

```
$ npm install --save-dev eslint-config-prettier
```

リスト 8 .eslintrc.js

```
module.exports = {
  env: {
    browser: true,
    es2021: true,
  },
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:@typescript-eslint/recommended',
    'prettier',
  ],
  parser: '@typescript-eslint/parser',
  parserOptions: {
    ecmaFeatures: {
      jsx: true,
    },
  },
}
```

(次のページに続く)

(前のページからの続き)

```

    ecmaVersion: 12,
    sourceType: 'module',
  },
  plugins: ['react', '@typescript-eslint'],
  rules: {
    'no-console': process.env.NODE_ENV === 'production' ? 2 : 0,
    '@typescript-eslint/explicit-module-boundary-types': 0,
    'no-unused-vars': ['error', { argsIgnorePattern: '^_' } ]
  },
  settings: {
    react: {
      version: "detect",
    }
  }
};

```

"env"はソースコードが対象している環境です。使えるクラスや関数の種類がここで変わります。ここではES2020の宣言が利用できるようにしています。これ以外に設定する可能性があるのは"browser"か、"node"、テストフレームワークなどです。必要な方を追加しましょう。

- <https://eslint.org/docs/2.0.0/user-guide/configuring#specifying-environments>

コマンドラインは、Prettier の項目に追加して、4つ追加しました。2つはESLintのチェックと修正。のこりの2つはPrettierとESLintの一括実行です。

リスト 9 package.json

```

{
  "scripts": {
    "fix": "run-s fix:prettier fix:eslint",
    "fix:eslint": "eslint src --ext .ts --fix",
    "lint": "run-p lint:prettier lint:eslint",
    "lint:eslint": "eslint src --ext .ts",
  }
}

```

修正は直列、チェックは並列実行するようにしています。複数のタスクを並列や並行で実行するには次のコマンドをインストールします。

```
$ npm install --save-dev npm-run-all
```

再び、試しに実行して、正しくインストールされたか確認します。ここでは一度きりの代入しかないのに、constではなく、letを使うコードで試したものです。次に、修正コマンドも確認してみましょう。問題なければ、ESLintの設定の基本は完了です。

```
$ npm run lint
/examples/console/src/main.ts
   4:9  error  'content' is never reassigned. Use 'const' instead

$ npm run fix
```

21.4.1 Visual Studio Code の設定

こちらも、Prettier 同様に VSCode に追加しましょう。recommendations に以下の拡張機能を追加します。コードの品質向上を目的として、スペルチェッカーも入れておきましょう。これを入れると英語の単語として存在しないものに青線が引かれるようになります。固有名詞やプロジェクトのキーワードは

リスト 10 .vscode/extensions.json

```
{
  "recommendations": [
    "streetsidesoftware.code-spell-checker",
    "dbaeumer.vscode-eslint"
  ],
  "unwantedRecommendations": []
}
```

次にプロジェクトで ESLint を使うように設定します。

リスト 11 .vscode/settings.json

```
{
  "eslint.lintTask.enable": true,
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  }
}
```

わざと引っかかるようなコードを書いてみて、赤線が表示され、保存時に let が const に書き換わるようになれば完了です。

注釈: 以前は次に紹介する Prettier を ESLint の一部として組み込んで利用することがデファクトスタンダードでした。Lint のエラーもしかし、その場合、チェックのたびにコードをフォーマットしなおし、それからパースして文法のチェックが実行されます。ESLint はコーディングの中でなるべくリアルタイムに結果をプログラマーに提示の方が開発の流れが途切れずに品質の高いコードが量産できます。現在はフォーマッターとこの ESLint は同期させないで個別に実行させるのが推奨となっています。

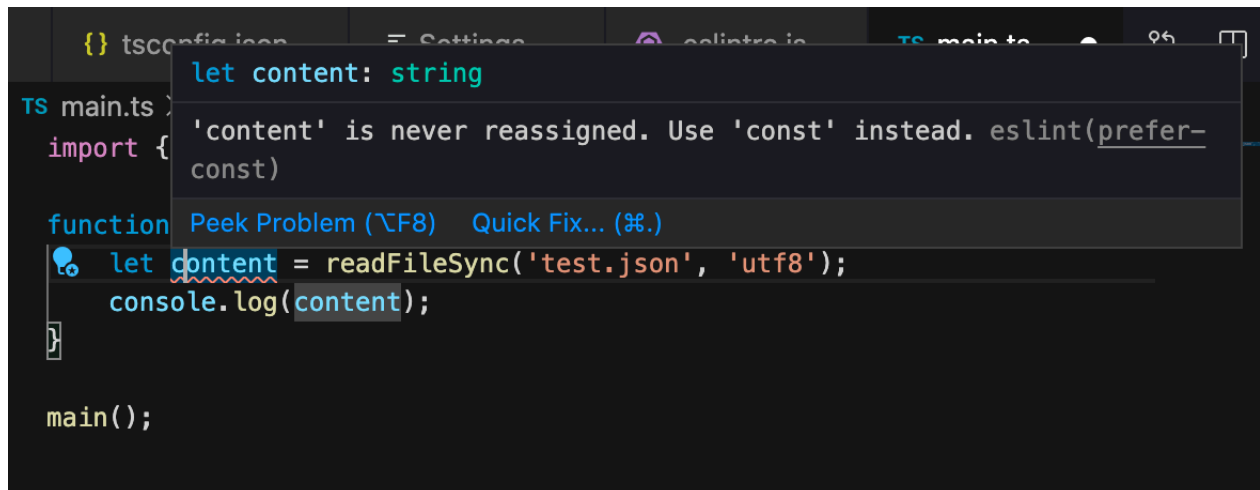


図 2 赤線が引かれるようになった

21.4.2 ESLint の警告と特定の行だけ無効化する

ESLint の警告はなるべく適用したいが、特別なコードだけ除外したいことがあります。逆をやることは基本的になく、なるべく厳しくして、特別な箇所だけ緩めてあげるのが一番やりやすい方法でしょう。例えば、コードジェネレータで生成したコードの警告を無効化したり、変数名の規則は camelCase だが、サーバーのレスポンスのみ snake_case を許容したい場合などがあります。

リスト 12 特定の行のみ無効化

```
const { status_code } = await res.json(); // eslint-disable-line camelcase

// eslint-disable-next-line camelcase
const { status_code } = await res.json();
```

リスト 13 特定のブロック内のみ無効化

```
/* eslint-disable camelcase */  
  
const { status_code } = await res.json();  
  
/* eslint-enable camelcase */
```

これ以外に、`.eslintignore` でファイルごと無効化する方法など、さまざまな方法があります。

21.5 テスト

ユニットテスト環境も作ります。TypeScript を事前に全部ビルドしてから Jasmine とかも見かけますが、公式で TypeScript を説明している Jest にしてみます。

```
$ npm install --save-dev jest ts-jest @types/jest eslint-plugin-jest
```

`scripts/test` と、`jest` の設定を追加します。古い資料だと、`transform` の値が `node_modules/ts-jest` 等になっているのがありますが、今は `ts-jest` だけでいけます。

リスト 14 package.json

```
{  
  "scripts": {  
    "test": "jest"  
  }  
}
```

リスト 15 jest.config.js

```
module.exports = {
  transform: {
    "^.+\\.tsx?$": "ts-jest"
  },
  moduleFileExtensions: [
    "ts",
    "tsx",
    "js",
    "json",
    "jsx"
  ]
};
```

Jest でも Mocha でも、人気のフレームワークはテスト専用の関数などが定義されているものとしてテストコードを記述していきますが、これらの関数があるかどうかは、ESLint からは見えません。ESLint にさまざまな設定を追加することで、Jest 固有のキーワードでもエラーがでなくなります。

リスト 16 .eslintrc.js

```
{
  env: {
    :
    'jest/globals': true,
  }
  extends: [
    :
    'plugin:jest/recommended'
    :
  ],
  plugins: [
    "jest"
    :
  ]
}
```

課題: tsdoc とかドキュメントツールを紹介

課題: eslint やテストの書き方の紹介

第 22 章

ライブラリ開発のための環境設定

現在、JavaScript 系のものは、コマンドラインの Node.js 用であっても、Web 用であっても、なんでも一切合切 npmjs にライブラリとしてアップロードされます。ここでのゴールは可用性の高いライブラリの実現で、具体的には次の 3 つの目標を達成します。

- 特別な環境を用意しないと（ES2015 modules 構文使っていて、素の Node.js で npm install しただけで）エラーになったりするのは困る
- npm install したら、型定義ファイルも一緒に入って欲しい
- 今流行りの Tree Shaking に対応しないなんてありえないよね？

今の時代でも、ライブラリは ES5 形式で出力はまだ必要です。インターネットの世界で利用するには古いブラウザ対応が必要だったりもします。

ライブラリのユーザーがブラウザ向けに webpack と Babel を使うとしても、基本的には自分のアプリケーションコードにしか Babel での変換は適用しないでしょう。よく見かける設定は Babel の設定は次の通りです。つまり、配布ライブラリは、古いブラウザなどでも動作する形式で npm にアップロードしなければならないということです。

リスト 1 webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: "babel-loader",
        exclude: /node_modules/
      }
    ]
  }
};
```

なお、ライブラリの場合は特別な場合を除いて Babel も webpack もいりません。CommonJS 形式で出しておけば、

Node.js で実行するだけなら問題ありませんし、基本的に webpack とか rollup.js とかのバンドラーを使って 1 ファイルにまとめるのは、最終的なアプリケーション作成者の責務となります。特別な場合というのは、Babel プラグインで加工が必要な JSS in JS とかですが、ここでは一旦おいておきます。ただし、JSX は TypeScript の処理系自身で処理できるので不要です。

TypeScript ユーザーがライブラリを使う場合、バンドルされていない場合は `@types/ライブラリ名` で型情報だけを追加ダウンロードすることがありますが、npm パッケージにバンドルされていれば、そのような追加作業がなくても TypeScript から使えるようになります。せっかく TypeScript でライブラリを書くなら、型情報は自動生成できますし、それをライブラリに添付する方がユーザーの手間を軽減できます。

Tree Shaking というのは最近のバンドラーに備わっている機能で、import と export を解析して、不要なコードを削除し、コードサイズを小さくする機能です。webpack や rollup.js の Tree Shaking では ES2015 modules 形式のライブラリを想定しています。

一方、Node.js は `--experimental-modules` を使わないと ES2015 modules はまだ使えませんし、その場合は拡張子は `.mjs` でなければなりません。一方で、TypeScript は出力する拡張子を `.mjs` にできません。また、Browserify を使いたいユーザーもいると思いますので CommonJS 形式も必須です。

そのため、モジュール方式の違いで 2 種類のライブラリを出力する必要があります。

22.1 ディレクトリの作成

前章のディレクトリ作成に追加して、2 つのディレクトリを作成します。

```
$ mkdir dist-cjs
$ mkdir dist-esm
```

出力フォルダを CommonJS と、ES2015 modules 用に 2 つ作っています。

これらのファイルはソースコード管理からは外すため、`.gitignore` に入れておきましょう。

22.2 ビルド設定

開発したいパッケージが Node.js 環境に依存したものであれば、Node.js の型定義ファイルをインストールして入れています。これでコンパイルのエラーがなくなり、コーディング時にコード補完が行われるようになります。

```
$ npm install --save-dev @types/node
```

共通設定のところで `tsconfig.json` を生成したと思いますが、これをライブラリ用に設定しましょう。大事、というのは今回の要件の使う側が簡単ように、というのを達成するための `.d.ts` ファイル生成と、出力形式の ES5 のところと、入力ファイルですね。ユーザー環境でデバッグしたときにきちんと表示されるように `source-map` もついでに設定しました。あとはお好みで設定してください。

リスト 2 tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",           // 大事
    "declaration": true,      // 大事
    "declarationMap": true,
    "sourceMap": true,        // 大事
    "lib": ["dom", "ES2018"],
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  },
  "include": ["src/**/*.ts"] // 大事
}
```

ここでは lib に ES2018 を指定しています。ES5 で出力する場合、Map などの新しいクラスや、Array.entries() などのメソッドを使うと、実行時に本当に古いブラウザで起動するとエラーになることがあります。Polyfill をライブラリレベルで設定しても良いのですが、最終的にこれを使うアプリケーションで設定するライブラリと重複してコード量が増えてしまうかもしれないので、README に書いておくでも良いかもしれません。

package.json で手を加えるべきは次のところぐらいですね。

- つくったライブラリを読み込むときのエントリーポイントを main で設定

src/index.ts というコードがあって、それがエントリーポイントになるというのを想定しています。

- scripts に build でコンパイルする設定を追加

今回は CommonJS 形式と ES2015 の両方の出力が必要なため、一度のビルドで両方の形式に出力するようにします。

リスト 3 package.json

```
{
  "main": "dist-cjs/index.js",
  "module": "dist-esm/index.js",
  "types": "dist-cjs/index.d.ts",
  "scripts": {
    "build": "npm-run-all -s build:cjs build:esm",
    "build:cjs": "tsc --project . --module commonjs --outDir ./dist-cjs",
    "build:esm": "tsc --project . --module es2015 --outDir ./dist-esm"
  }
}
```

(次のページに続く)

(前のページからの続き)

```
}
```

課題: // browser/module など// <https://qiita.com/shinout/items/4c9854b00977883e0668>

22.3 ライブラリコード

`import ... from "ライブラリ名"` のようにアプリケーションや他のライブラリから使われる場合、最初に読み込まれるエントリーポイントは `package.json` で指定していました。

拡張子の前のファイル名が、TypeScript のファイルのファイル名となる部分です。前述の例では、`index.js` や `index.d.ts` が `main`、`module`、`types` で設定されていたので、`index.ts` というファイルでファイルを作成します。`main.ts` に書きたい場合は、`package.json` の記述を修正します。

リスト 4 src/index.ts

```
export function hello() {  
  console.log("Hello from TypeScript Library");  
}
```

ここで `export` したものが、ライブラリユーザーが触れられるものになります。

22.4 まとめ

アルゴリズムなどのロジックのライブラリの場合、`webpack` などのバンドラーを使わずに、TypeScript だけを使えば良いことがわかりました。ここにある設定で、次のようなことが達成できました。

- TypeScript でライブラリのコードを記述する
- 使う人は普段通り `require/import` すれば、特別なツールやライブラリの設定をしなくても適切なファイルがロードされる。
- 使う人は、別途型定義ファイルを自作したり、別パッケージをインストールしなくても、普段通り `require/import` するだけで TypeScript の処理系や Visual Studio Code が型情報を認識する
- Tree Shaking の恩恵も受けられる

`package.json` の `scripts` のところに、開発に必要なタスクがコマンドとして定義されています。`npm` コマンドを使って行うことができます。

```
# ビルドしてパッケージを作成  
$ npm run build
```

(次のページに続く)

(前のページからの続き)

```
$ npm pack

# テスト実行 (VSCode だと、⌘ R T でいける)
$ npm test

# 文法チェック
$ npm run lint

# フォーマッター実行
$ npm run fix
```


第 23 章

CLI ツール・ウェブサーバー作成のための環境設定

TypeScript を使って CLI ツールやウェブサーバーなどの Node.js や Deno がある環境で動作するソフトウェアを作成するための環境構築方法を紹介していきます。2 つ方法があります。

- Node.js がインストールされている環境向け
- Deno がインストールされている環境向け

どちらも、TypeScript を JavaScript 変換して 1 ファイルにまとめたファイルを作成します。

23.1 Node.js の環境構築

Node.js 向けには@zeit/ncc を利用します。

@zeit/ncc は TypeScript に最初から対応したコンパイラで、コマンドラインツールやウェブサーバーなどの Node.js アプリケーションを 1 ファイルにするために作られています。@zeit/ncc 自体も自分自身を使ってバンドル化されており、ごく小さいサイズですばやくインストールできます。Go を参考にしており、tsconfig.json 以外の設定は不要です。もちろん、

他に実績のあるツールとしては Browserify があります。これも、tsify というプラグインと併用することで TypeScript 製アプリケーションをバンドルして 1 ファイルにできます。このツールはメインの目的としては Node.js 製アプリケーションをブラウザでも動くようにするために作られました。後発の@zeit/ncc に比べると設定が多いのと、TypeScript の設定を変えて common.js 形式のパッケージにしないと扱えないのが ncc と比べると煩雑です。また、tsc を使いバンドルせずに TypeScript を JavaScript に変換するだけを行って配布する方法もありますが、バンドルをしないとさまざまなファイルが入ってくるため、配布サイズも大きくなります。

23.1.1 作業フォルダを作る

フォルダを作成し、JavaScript のいつものプロジェクトのように `npm init -y` を実行して `package.json` を作成しましょう。

ライブラリの時は、ES2015 modules と CommonJS の 2 通り準備しましたが、CLI の場合は Node.js だけ動かせば良いので、出力先は一つになります。コンパイルしたファイル置き場は `dist` フォルダになりますが、コンパイル時に自動で作られるので作成しておく必要はありませんが、`.gitignore` には登録しておきましょう。

23.1.2 ビルド設定

まずは `@zeit/ncc` をインストールします。

Node.js の機能を使うことになるため、Node.js の API の型定義ファイルは入れておきましょう。TypeScript は `ncc` にバンドルされていますが、バンドルされている処理系のバージョンが古いことがあります。インストールしてある TypeScript を優先的に使ってくれるので入れておくと良いでしょう。

コマンドラインで良く使うであろうライブラリを追加しておきます。カラーでのコンソール出力、コマンドライン引数のパーサ、ヘルプメッセージ表示です。

どれも TypeScript の型定義があるので、これも落としておきます。また、ソースマップサポートを入れると、エラーの行番号がソースの TypeScript の行番号で表示されるようになって便利なので、これも入れておきます。

```
$ npm install --save-dev @zeit/ncc typescript

$ npm install --save-dev @types/node @types/cli-color
  @types/command-line-args @types/command-line-usage
  @types/source-map-support

$ npm install --save cli-color command-line-args
  command-line-usage source-map-support
```

TypeScript のビルド設定のポイントは、ブラウザからは使わないので、ターゲットのバージョンを高くできる点にあります。ローカルでは安定版を使ったとしても Node.js 10 が使えるでしょう。

ライブラリのときとは異なり、成果物を利用するのは Node.js の処理系だけなので、`.d.ts` ファイルを生成する必要はありません。

リスト 1 tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2018",          // お好みで変更
    "declaration": false,       // 生成したものを他から使うことはないので false に
    "declarationMap": false,    // 同上
    "sourceMap": true,          //
```

(次のページに続く)

(前のページからの続き)

```

    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",          // Node.js で使うため
    "outDir": "./dist"             // 出力先を設定
  },
  "include": ["src/**/*.ts"]
}

```

package.json 設定時は、他のパッケージから利用されることはないため、main/modules/types の項目は不要です。代わりに、bin 項目でエントリーポイント（ビルド結果の方）のファイルを指定します。このキー名が実行ファイル名になります。

リスト 2 package.js

```

{
  "bin": {
    "awesome-cmd": "dist/index.js"
  },
  "scripts": {
    "build": "ncc build main.ts --minify --v8-cache --source-map",
    "watch": "ncc build main.ts --watch",
    "start": "ncc run main.ts",
    "lint": "eslint .",
    "fix": "eslint --fix ."
  }
}

```

ncc build でビルドします。ncc run でビルドして即座に実行します。設定ファイルはありません。必要十分な機能が揃っています。オプションで指定できるものは以下の通りです。

- --minify: ミニファイしてファイルを小さくする
- --source-map: ソースマップを出力
- -e, --external [mod]: バンドルをスキップするモジュール
- --watch: 変更を検知してビルド
- --v8-cache: V8 のコンパイルキャッシュを生成

テストの設定、VSCode の設定は他の環境の設定と変わりません。

もし、バイナリを入れる必要のあるライブラリがあると、ビルド時にエラーになります。その場合は、そのパッ

ケースを `--external` パッケージ名 で指定してバンドルされないようにします。ただし、この場合は配布環境でこのライブラリだけは `npm install` しなければなりません。

注釈: Node.js/Deno 以外の処理系

また、`low.js`^{*1} という、ES5 しか動かないものの Node.js と一部互換性があるモジュールを提供し、ファイルサイズがごく小さいインタプリタがありますが、これと一緒に使うこともできます。

`low.js` は ES5 までしか対応しないため、出力ターゲットを ES5 にする必要があります。

リスト 3 tsconfig.json (low.js を使う場合)

```
{
  "compilerOptions": {
    "target": "es5",           // もし low.js を使うなら
    "lib": ["dom", "es2017"]  // もし low.js で新しいクラスなどを使うなら
  }
}
```

これで、TypeScript 製かつ、必要なライブラリが全部バンドルされたシングルファイルなスクリプトができあがります。

23.1.3 CLI ツールのソースコード

TypeScript はシェバング (`#!`) があると特別扱いしてくれます。必ず入れておきましょう。ここで紹介した `command-line-args` と `command-line-usage` は Wiki で用例などが定義されているので、実装イメージに近いものをベースに加工していけば良いでしょう。

```
index.ts
#!/usr/bin/env node

import * as clc from "cli-color";
import * as commandLineArgs from "command-line-args";
import * as commandLineUsage from "command-line-usage";

// あとで治す
require('source-map-support').install();

async function main() {
  // 内部実装
}

main();
```

*1 <https://www.lowjs.org/>

23.1.4 Node.js のまとめ

コマンドラインツールの場合は、npm で配布する場合はライブラリ同様、バンドラーを使わずに、TypeScript だけを使えば大丈夫です。ここにある設定で、次のようなことが達成できました。

- TypeScript で CLI ツールのコードを記述する
- 使う人は普段通り `npm install` すれば実行形式がインストールされ、特別なツールやライブラリの設定をしなくても利用できる。

また、おまけで 1 ファイルにビルドする方法も紹介しました。

`package.json` の `scripts` のところに、開発に必要なタスクがコマンドとして定義されています。npm コマンドを使って行うことができます。すべてライブラリと同じです。

```
# ビルドして実行
$ npm start

# ビルドしてパッケージを作成
$ npm run build
$ npm pack

# テスト実行 (VSCode だと、⌘ R Tでいける)
$ npm test

# 文法チェック
$ npm run lint

# フォーマッター実行
$ npm run fix
```

23.2 Deno

Deno は新しい処理系です。Node.js と同じく V8 をベースとしている兄弟処理系ですが、TypeScript にデフォルトで対応していたり、ネイティブコード部分は Rust を使って実装されています。サードパーティのパッケージは npm コマンドではなく deno コマンドを使ってダウンロードします。

Node.js を置き換えるものかという点、現時点ではまだパッケージやツールが足りていません。特に Node.js の npm はウェブフロントエンドのライブラリなどの配布にも使われており、ウェブフロントエンドの開発のためのプラットフォームとしても活用されています。Deno はコマンドラインツールやウェブサービスの開発に特化しています。

こちらにも Go を参考にしたコマンドを持っています。それ単体でビルドして配布できます。

第 24 章

CI（継続的インテグレーション）環境の構築

課題: travis、circle.ci、gitlab-ci の設定を紹介。あとは Jenkins？

<https://qiita.com/nju33/items/72992bd4941b96bc4ce5>

<https://qiita.com/naokikimura/items/f1c8903eec86ec1de655>

第 25 章

成果物のデプロイ

TypeScript で作ったアプリケーションの開発環境の作り方を、バリエーションごとに紹介してきました。それぞれの環境でビルド方法についても紹介しました。本章ではデプロイについて紹介します。

課題: npm パッケージ to npm npm パッケージ to nexus

<https://qiita.com/kannkyo/items/5195069c65350b60edd9>

25.1 npm パッケージとしてデプロイ

この方法でデプロイする対象は以下の通りです。

- Node.js 用のライブラリ
- Node.js 用の CLI ツール
- Node.js 用のウェブサービス

ビルドしたらアーカイブファイルを作ってみましょう。これで、`package.tgz` ファイルができます。`npm install` にこのファイルのパスを渡すとインストールできます。アップロードする前に、サンプルのプロジェクトを作ってみて、このパッケージをインストールしてみて、必要なファイルが抜けていないか、必要な依存パッケージが足りているかといったことを確認してみましょう。また、展開してみて、余計なファイルが含まれていないことも確認すると良いでしょう。

```
$ npm pack
```

- ビルドには必要だが、配布する必要のないファイルが含まれている
 - `.npmignore` ファイルにそのファイル名を列挙します。パッケージを作る時に無視されます。
- ビルドには必要だが、利用環境でインストール不要なパッケージがある

package.json の dependencies から、devDependencies に移動します。

npm のサイトにアップロードしてみましょう。

パッケージリポジトリは npm 以外にもあります。例えば、Nexus を使えばローカルにパッケージリポジトリが建てられます。GitHub もパッケージリポジトリを提供しています。

25.2 サーバーにアプリケーションをデプロイ

Node.js はシングルコアを効率よく使う処理系です。サーバーアプリケーションでマルチコアを効率よく使うには、プロセスマネージャを利用します。本書では pm2 を利用します。

- <https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/>

サンプルとしては次のコードを使います。

リスト 1 src/main.ts

```
import express, { Request, Response } from "express";
import compression from "compression";
import bodyParser from "body-parser";
import gracefulShutdown from "http-graceful-shutdown";

const app = express();
app.use(compression());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.get("/", (req: Request, res: Response) => {
  res.json({
    message: `hello ${req.headers["user-agent"]} `,
  });
});

const host = process.env.HOST || "0.0.0.0";
const port = process.env.PORT || 3000;

const server = app.listen(port, () => {
  console.log("Server is running at http://%s:%d", host, port);
  console.log(" Press CTRL-C to stop\n");
});

gracefulShutdown(server, {
  signals: "SIGINT SIGTERM",
  timeout: 30000,
  development: false,
  onShutdown: async (signal: string) => {
    console.log("... called signal: " + signal);
```

(次のページに続く)

(前のページからの続き)

```

    console.log("... in cleanup");
    // shutdown DB or something
  },
  finally: () => {
    console.log("Server gracefully shutdown.....");
  },
});

```

tsconfig.json は `npx tsc --init` で生成したものをひとまず使います。package.json は以下のものを利用します。ncc を使ってビルドする前提となっています。

リスト 2 package.json

```

{
  "name": "webserver",
  "version": "1.0.0",
  "scripts": {
    "build": "ncc build src/main.ts"
  },
  "author": "Yoshiki Shibukawa",
  "license": "ISC",
  "dependencies": {
    "pm2": "^4.4.0"
  },
  "devDependencies": {
    "@types/body-parser": "^1.19.0",
    "@types/compression": "^1.7.0",
    "@types/express": "^4.17.7",
    "@zeit/ncc": "^0.22.3",
    "body-parser": "^1.19.0",
    "compression": "^1.7.4",
    "express": "^4.17.1",
    "http-graceful-shutdown": "^2.3.2",
    "typescript": "^3.9.7"
  }
}

```

ncc でビルドすると、`dist/index.js` という一つの js ファイルが生成されます。実行は `node dist/index.js` の代わりに次のコマンドを利用します。これで、CPU コア数分 Node.js のインスタンスを起動し、クラスタで動作します。

```
pm2 start dist/index.js -i max
```

pm2 で起動すると、デーモン化されてアプリケーションが起動します。pm2 logs コマンドでログをみたり、pm2 status や pm2 list で起動しているプロセスの状態を知ることができます。

デーモン化させないでフォアグラウンドで動作させる場合は `--no-daemon` をつけて起動します。

25.3 Docker イメージの作成

この方法でデプロイする対象は以下の通りです。

- Node.js 用の CLI ツール
- Node.js 用のウェブサービス
- ウェブフロントエンド

なお、本章のサンプルはベースイメージのバージョンは細かく指定していませんが、突然メジャーバージョンが上がってビルドできなくなることもあります。特に業務開発では Docker Hub のイメージ情報のタグを見て、適宜バージョンを固定することをお勧めします。

25.3.1 コンテナとは何か

コンテナは 1 アプリケーションだけが格納されたミニ OS 環境です。Linux 上でも macOS 上でも Windows 上でもクラウド環境でも、アプリケーションからは同じ OS 環境のように見えます。ポータブルなアプリケーション配布・実行環境としてますます地位が高まっています。コンテナは動いている環境のことを指します。コンテナは実行時にイメージを読み込んで環境を構築します。これは実行に必要なファイルと起動時のコマンドなどがセットになったものです。開発者が作るのはイメージです。

コンテナ関係のシステムは、実行のランタイムやビルド方法など、それぞれにいくつか選択肢がありますが、開発時の環境として一番ポピュラーなのが Docker です。本書ではコンテナ=Docker コンテナとして説明をします。

ローカルではコンテナは Docker for Desktop を使ってイメージの作成や動作のテストができます。運用環境として、どのクラウド事業者も Kubernetes を使ってコンテナベースで本番運用環境の維持管理できるサービスを提供しています。1 つのノードにリソースが許すかぎり多数のコンテナを配置することができ、実行時の効率も上がります。それ以外にも、AWS ECS や AWS Fargate、GCP Cloud Run など、単体の Docker イメージや Docker イメージ群を起動できるサービスもあります。コンテナはウェブアプリケーションのような起動し続けるサービスにも使えますし、一度実行して終了するバッチ処理にも活用できます。

Docker コンテナ内のアプリケーションは外部の環境と切り離されて実行されますが、Docker の実行時のオプションで外界と接点を設定できます。複雑な設定が必要なアプリケーションの場合は、設定ファイルをコンテナ内の特定のパスに置くこともできますが、推奨されるのは環境変数のみによって制御されるアプリケーションです。

- 環境変数
- ネットワークの設定
 - 特定のポートを localhost に公開
 - localhost とコンテナ内部のを同一ネットワークにするかどうか
- ファイルやフォルダのマウント

- 最後に実行するコマンドのオプション

コンテナは上記のように、クラウドサービスに直接デプロイして実行できます。

複数のコンテナに必要な設定を与えてまとめて起動するツール（コンテナオーケストレーションツール）もあります。それが docker-compose や Kubernetes です。

25.3.2 Docker のベースイメージの選択

Docker イメージを作成するには Dockerfile という設定ファイルを作成し、docker build コマンドを使ってイメージを作成します。ベースイメージと呼ばれる土台となるイメージを選択して、それに対して必要なファイルを追加します。ビルド済みのアプリケーションを単に置く、という構築方法もありますが（公式イメージの多くはそれに近いことをしている）、アプリケーション開発の場合はソースコードを Docker 内部に送り、それを Docker 内部でビルドして、実行用イメージを作成します。できあがったイメージをコンパクトにするために、ビルド用イメージと、実行用イメージを分ける（マルチステージビルド）が今の主流です。

Node.js の公式のイメージは以下のサイトにあります。

- https://hub.docker.com/_/node/

バージョンと、OS の組み合わせだけイメージがあります。その中でおすすめの組み合わせが次の 3 つです。

用途	バリエーション	ビルド用イメージ	実行用イメージ	解説
Node.js (鉄板ウェブアプリ)		node の Debian 系	node の Debian-slim 系	ネイティブ拡張があっても利用可能
Node.js (CLI 型ウェブアプリ)	ネイティブ拡張なし	node の Debian-slim 系	node の Debian-slim 系	ビルド環境もコンパクトに
Node.js (セキュリティ重視ウェブアプリ)		node の Debian-slim 系	distroless の node.js	コンテナへのログインを許さないセキュアな実行イメージ
ウェブフロントエンド配信		node の Debian-slim 系	nginx:alpine	

課題: Deno はこちらのスレッドを見守る <https://github.com/denoland/deno/issues/3356>

Debian は Linux ディストリビューションの名前です。buster (Debian 10)、stretch (Debian 9)、jessie (Debian 8) が執筆時点ではコンテナリポジトリにあります。それぞれ、無印がフル版で、gcc や各種開発用ライブラリを含みます。いろいろ入っていて便利ですが、イメージサイズは大きめです。slim がつくバージョンがそれぞれにあります。これは Node.js は入っているが、gcc などがいないバージョンです。例えば、最新 LTS (執筆時点で 12) の Debian の開発環境込みのイメージであれば、`node:12-buster` を選びます。

もう一つ、GCP のコンテナレジストリで提供されているのが `distroless` です。こちらはシェルもなく、セキュリティパッチも積極的に当てていくという、セキュリティにフォーカスした Debian ベースのイメージです。シェルがないということはリモートログインができませんので、踏み台にされる心配がないイメージです。これは GCP のコンテナレジストリに登録されており、`gcr.io/distroless/nodejs` という名前で利用可能です。

Alpine というサイズ重視の OS イメージはありますが、あとから追加インストールしなければならないパッケージが増えがちなのと、パッケージのバージョン固定がしにくい (古いパッケージが削除されてしまってインストールできなくなる) などの問題がありますし、他のイメージがだいたい Debian ベースなので、Debian ベースのもので揃えておいた方がトラブルは少ないでしょう。

Docker イメージはサイズが重視されますが、ビルド時間や再ビルド時間も大切な要素です。開発ツールなしのイメージ (`slim` や `alpine` など) を選び、必要な開発ツールだけをダウンロードするのはサイズの上では有利ですが、すでにできあがったイメージをただダウンロードするのよりも、依存関係を計算しながら各パッケージをダウンロードの方が時間がかかります。

25.4 CLI/ウェブアプリケーションのイメージ作成

CLI とウェブアプリケーションの場合の手順はあまり変わらないので一緒に説明します。ベースイメージの選択では 3 種類の組み合わせがありました。

- C 拡張あり (Debian 系でビルド)
- C 拡張なし (Debian-slim 系でビルド)
- セキュリティ重視 (`distroless` に配信)

前 2 つ目はベースイメージが `node:12-buster` から `node:12-buster-slim` に変わるだけですので、まとめて紹介します。

なお、Node.js はシングルコアで動作する処理系ですので、マルチコアを生かしたい場合はインスタンスを複数起動し、ロードバランスをする仕組みを外部に起動する必要があります。

25.4.1 Debian ベースのイメージ作成と Docker の基礎

まず、イメージにするアプリケーションを作成します。コンテナの中のアプリケーションは終了時にシグナルが呼ばれますので、シグナルに応答して終了するように実装する必要があります。

Dockerfile はコンテナのイメージを作成するためのレシピです。行志向のスクリプトになっています。Windows や macOS では、Linux が動作している仮想 PC の中で Docker のサーバーが動作しており（Windows の場合は Windows も動作しますが、ここでは無視します）、ビルドを実行すると、実行されているフォルダの配下のファイル（コンテキスト）と Dockerfile が、まとめてサーバーに送られます。サーバーの中で、Dockerfile に書かれた命令に従ってベースとなるディスクイメージに手を加えていきます。実行されるのはローカルコンピュータではないので、記述できるコマンドも、そのベースイメージの Linux で使えるものに限られます。

そのため、Dockerfile に問題があれば、Windows 上でビルドや実行をしても、macOS 上でビルドや実行をしても、まったく同じエラーが発生するはずです。バージョン番号や内部で使われるパッケージのバージョンはその時の最新を使うこともできるため、いつでもまったく同じにはならないかもしれませんが、どこで誰がどの OS で実行しても、同じイメージが作られます。この冪等性が Docker が重宝されるポイントです。

次のファイルが、実際に動作する Dockerfile です。順を追って見ていきます。

リスト 3 Dockerfile

```
# ここから下がビルド用イメージ

FROM node:12-buster AS builder

WORKDIR app
COPY package.json package-lock.json ./
RUN npm ci
COPY tsconfig.json ./
COPY src ./src
RUN npm run build

# ここから下が実行用イメージ

FROM node:12-buster-slim AS runner
WORKDIR /opt/app
COPY --from=builder /app/dist ./
USER node
EXPOSE 3000
CMD ["node", "/opt/app/index.js"]
```

FROM はベースとなるイメージを選択する命令です。ここではビルド用のベースイメージと、実行用のベースイメージと 2 箇所 FROM を使用しています。FROM から次の FROM、あるいはファイルの終行までがイメージになります。ここでは 2 つイメージが作られていますが、最後のイメージが、この Dockerfile の成果物のイメージとなります。わざわざ 2 つに分けるのは、アクロバティックなことをしないで最終的なイメージサイズを小さくするためです。

それぞれのイメージの中ではいくつかの命令を使ってイメージを完成させていきます。

COPY は実行場所（コンテキスト）や別のイメージ（`--from`が必要）からファイルを取得してきて、イメージ内部に配置する命令です。このサンプルでは使っていませんが、ADD 命令もあり、こちらは COPY の高性能バージョンです。ネットワーク越しにファイルを取得できますし、アーカイブファイルを展開してフォルダに配置もできます。RUN は何かしらの命令を実行します。

重要なポイントが、このイメージ作成のステップ（行）ごとに内部的にはイメージが作成されている点です。このステップごとのファイルシステムの状態は「レイヤー」と呼ばれます。このレイヤーはキャッシュされて、コンテキストとファイルシステムの状態に差分がなければキャッシュを利用します。イメージの内部にはこのレイヤーがすべて保存されています。実行用イメージはこのレイヤーとサイズの問題は心の片隅に置いておく方が良いですが（優先度としては 10 番目ぐらいです）、ビルド用のイメージはサイズが大きくなっても弊害とかはないので、なるべくステップを分けてキャッシュされるようにすべきです。また、キャッシュ効率をあげるために、なるべく変更が少ない大物を先にインストールすることが大切です。

上記のサンプルではパッケージ情報ファイル（`package.json` と `package-lock.json`）取得してきてサードパーティのライブラリのダウンロード（`npm install`）だけを先に実行しています。利用パッケージの変更はソースコードの変更よりもレアケースです。一方、ソースコードの変更は大量に行われます。そのためにソースコードのコピーを後に行っています。もし逆であれば、ソースコード変更のたびにパッケージのダウンロードが走り、キャッシュがほとんど有効になりません。このようにすれば、ソースコードを変更して再ビルドするときは `COPY src` の行より以前はスキップされてそこから先だけが実行されます。

実行用イメージの最後は CMD 命令を使います。シェルスクリプトで実行したいプログラムを記述するのと同じように記述すれば問題ありません。最後の CMD は、常時起動しつづけるウェブサーバーであっても、一通り処理を実行して終了するバッチコマンドであっても、どちらでもかまいません。

注釈: Docker とイメージサイズ削減

今回紹介している、ビルド用イメージと実行用イメージなど、複数イメージを利用してイメージを作成する方法は「マルチステージビルド」と呼ばれます。

すべてのステップがレイヤーとして保存されると紹介しました。例えば、ファイルを追加、そして削除をそれぞれ 1 ステップずつ実行すると、消したはずのファイルもレイヤーには残ってしまい、イメージサイズは大きなままとなります。

マルチステージビルドがなかった時代は、なるべくレイヤーを作らないことでイメージサイズを減らそうとしていました。例えば、C++ コンパイラをパッケージマネージャを使ってインストールし、ビルドを実行し、コンパイラを削除するというところまで、一つの RUN コマンドで行うこともありました。これであれば、結果のレイヤーは一つですし、ビルド済みのバイナリだけが格納されるのでサイズは増えません。次のコードは Redis の実際のリポジトリから取得してきたものです。Redis 3.0 当時のもので、今はもっと複雑ですが、この場合は試行錯誤すると、毎回パッケージのダウンロードが始まります。もはやこのようなことは不要です。

```

RUN buildDeps='gcc libc6-dev make' \
    && set -x \
    && apt-get update && apt-get install -y $buildDeps --no-install-recommends \
    && rm -rf /var/lib/apt/lists/* \
    && mkdir -p /usr/src/redis \
    && curl -sSL "$REDIS_DOWNLOAD_URL" -o redis.tar.gz \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | shasum -c - \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps

```

注釈: ビルド時間を確実に短くするテクニック

最初に、ローカルのファイルを一式 Docker のサーバーに送信すると説明しました。`.dockerignore` ファイルがあれば、そのサーバーに送るファイルを減らし、ビルドが始まるまでの時間が短縮されます。また、余計なファイルが変更されることでキャッシュが破棄されることを減らします。

明らかに巨大になるのが次の 2 ファイルです。この 2 つは最低限列举しましょう。`.git` を指定しないと、ブランチを切り替えただけで再ビルドになります。

リスト 4 .dockerignore

```

node_modules
.git

```

25.4.2 Docker イメージのビルドと実行

Dockerfile ができたらイメージをビルドして実行してみましょう。名前をつけなくても、最後に作成したものなので実行直後は迷子にはならないのですが、何かしら名前をつけておく方が何かと良いです。ハッシュな識別子はかならず生成されるので、削除や実行にはこの識別子も使えます。

```

# ビルド。作成したイメージに webserver と名前をつける
$ docker build -t [イメージ名] .

# イメージ一覧
$ docker images

# イメージ削除
$ docker rmi [イメージ名]

```

実行は次の通りです。

```
# 実行
$ docker run -it --rm --name [コンテナ名] -p "3000:3000" [イメージ名]

# 実行中コンテナの表示 (-a をつけると停止中のものも表示)
$ docker ps -a

# コンテナ停止 (-d で実行していなければ、Ctrl+C で停止可能)
$ docker stop [コンテナ名]

# 停止済みコンテナの削除
$ docker rm [コンテナ名]
```

説明によってはデーモン化のための `-d` オプションを紹介しているものもありますが、ログを見たりもあると思いますし、簡単に停止ができるようにこのオプションは付けずに実行する方が便利です。代わりに、実行しているターミナルに接続して情報を出力するために `-it` (ハイフンは一つ) を付けます。また、Docker を停止すると、停止状態になり、その後削除は `docker rm` コマンドを実行しなければなりません、`--rm` をつけると、停止時に削除まで行ます。`-p` はあけたいポート番号です。左がホスト、右が Docker 内部のプロセスのポートです。今回は同じなので、`-p "3000:3000"` を指定します。もし、コンテナ内部が 80 であれば、`-p "3000:80"` になります。

25.4.3 distroless ベースの Docker イメージの作成

distroless はシェルが入っておらず、外部からログインされることもなく安全という Google 製の Docker イメージです。標準 Linux に入っているようなツールも含めて、最小限にカットされています。Node.js、Java、Python、.net など言語のランタイムだけが入ったバージョン、llvm ベースのコンパイラで作成したコードを動かすだけのバージョン、何もないバージョンなど、いくつかのバリエーションが用意されています。今回は Node.js を使います。

現在、8 種類タグが定義されています。latest は LTS が終わるまでは 10 のままです。debug がついているものはデバッグ用のシェルが内蔵されています。

- `gcr.io/distroless/nodejs:latest: 10` と同じ
- `gcr.io/distroless/nodejs:10`
- `gcr.io/distroless/nodejs:12`
- `gcr.io/distroless/nodejs:14`
- `gcr.io/distroless/nodejs:10: 10-debug` と同じ
- `gcr.io/distroless/nodejs:10-debug`
- `gcr.io/distroless/nodejs:12-debug`
- `gcr.io/distroless/nodejs:14-debug`

一般的な Dockerfile は、ENTRYPOINT がシェル、CMD がそのシェルから呼び出されるプログラムです。distroless はシェルがなく、ENTRYPOINT に Node.js が設定されているので、CMD には JavaScript のスクリプトを設定します。拡張を使わないコードなら簡単に動作します。先ほどの Dockerfile と、ビルド部分はまったく同じです。

リスト 5 Dockerfile

```
# ここから下がビルド用イメージ

FROM node:12-buster AS builder

WORKDIR app
COPY package.json package-lock.json ./
RUN npm ci
COPY tsconfig.json ./
COPY src ./src
RUN npm run build

# ここから下が実行用イメージ

FROM gcr.io/distroless/nodejs AS runner
WORKDIR /opt/app
COPY --from=builder /app/dist ./
EXPOSE 3000
CMD ["/opt/app/index.js"]
```

25.5 ウェブフロントエンドの Docker イメージの作成

本節ではウェブフロントエンドが一式格納された Docker イメージを作成します。もし、Next.js でサーバーサイドレンダリングを行う場合、それは単なる Node.js のサーバーですので、前節の内容に従って Node.js のコンテナを作成してください。本節で扱うのは、サーバーを伴わない HTML/JavaScript などのフロントエンドのファイルを配信する方法です。

シングルページアプリケーションをビルドすると静的な HTML や JS、CSS のファイル群ができます。これらのファイルを利用する方法はいくつかあります。

- CDN やオブジェクトストレージにアップする
- Docker コンテナとしてデプロイする

このうち、CDN やオブジェクトストレージへのアップロードはそれぞれのサービスごとの作法に従って行ます。ここでは Docker コンテナとしてデプロイする方法を紹介します。Docker コンテナにするメリットはいくつかあります。主にテスト環境の構築がしやすい点です。

デプロイ用のバックエンドサーバーを Docker 化する流れは今後も加速していくでしょう。しかし、フロントエンドが特定のマネージドサービスにアップロードする形態の場合、デプロイ手段がバックエンドと異なるため、別の

デプロイ方式を取る必要が出てきます。ちょっとしたステージング環境や、開発環境を構築する際に、フロントエンドも Docker イメージになっていて本番環境の CDN をエミュレートできると、ローカルでもサーバーでも、簡単に一式のサービスが起動できます。PostgreSQL のイメージや、Redis のイメージ、クラウドサービスのエミュレータなどと一緒に、docker-compose で一度に起動するとテストが簡単に行えます。

サンプルとして、次のコマンドで作った React のアプリケーションを使います。もし、E2E テストの Cypress のような、ダウンロードが極めて重い超巨大パッケージがある場合は、optionalDependencies に移動しておくことをお勧めします。この Dockerfile では optional な依存はインストールしないようにしています。

```
$ npx create-react-app --template typescript webfront
```

Docker コンテナにする場合、ウェブサーバーの Nginx のコンテナイメージを基に、ビルド済みの JavaScript/HTML/その他のリソースを格納したコンテナイメージを作成します。次の内容が、フロントエンドアプリケーションの静的ファイルを配信するサーバーです。実行用イメージでは nginx:alpine イメージをベースに使っています。このイメージは最後に Nginx を立ち上げる設定がされているため、実行用コンテナに必要なのは生成されたファイルと、nginx.conf をコピーするだけです。

リスト 6 Dockerfile

```
# ここから下がビルド用イメージ

FROM node:12-buster AS builder

WORKDIR app
COPY package.json package-lock.json ./
RUN npm ci --no-optional
COPY tsconfig.json ./
COPY public ./public
COPY src ./src
RUN npm run build

# ここから下が実行用イメージ

FROM nginx:alpine AS runner
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=builder /app/build public
EXPOSE 80
```

配信用の nginx の設定です。シングルページアプリケーションにとって大切なパートが try_files です。シングルページアプリケーションでは 1 つの HTML/JS があらゆるページを作り上げます。そしてその時に URL を書き換えます。しかし、そこでブラウザリロードをすると、JavaScript によって作られた仮想的な URL を読みにこうとします。この try_files を有効にすると、一度アクセスしに行って見つからなかった場合にオプションで設定したファイルを返せます。ここでは index.html を返すので、そこで React Router などのフロントで動作しているウェブサービスが仕分けを行い、もしどこにもマッチしなければフロント側の Router がエラーをハンドリングできます。

リスト 7 nginx.conf

```
worker_processes 2;

events {
    worker_connections 1024;
    multi_accept on;
    use epoll;
}

http {
    include /etc/nginx/mime.types;
    server {
        listen 80;
        server_name 127.0.0.1;

        access_log /dev/stdout;
        error_log stderr;

        location / {
            root /public;
            index index.html;
            try_files $uri $uri/ /index.html =404;
            gzip on;
            gzip_types text/css application/javascript application/json image/svg+xml;
            gzip_comp_level 9;
        }
    }
}
```

それではビルドして実行してみましょう。正しく動作していることが確認できます。

```
$ docker run -it --rm --name [コンテナ名] -p "3000:80" [イメージ名]
```

注釈: <https://qiita.com/shibukawa/items/6a3b4d4b0cbd13041e53>

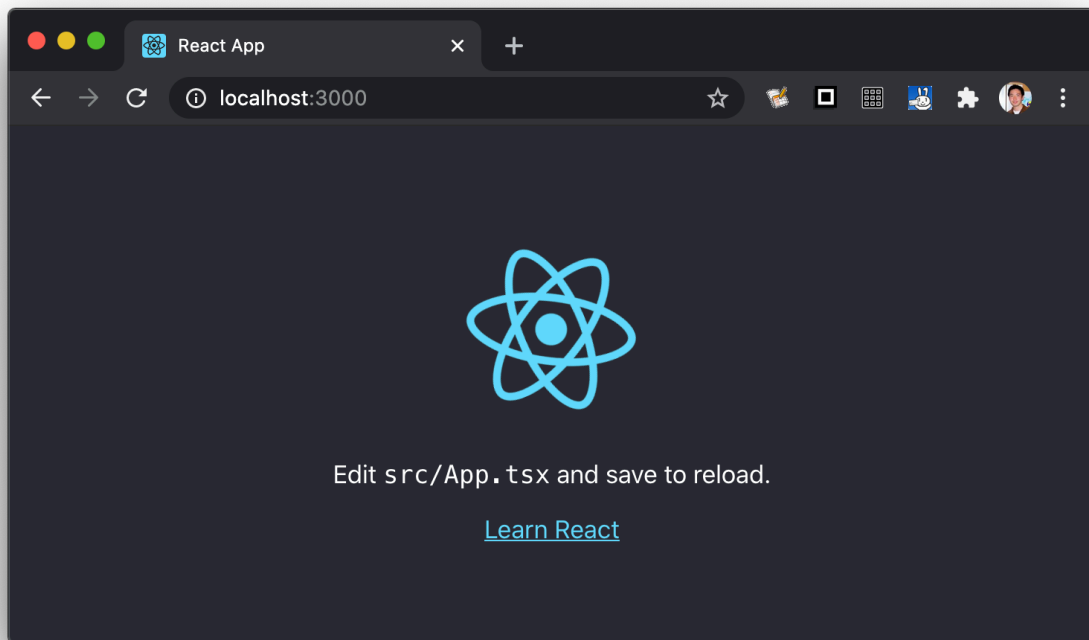


図 1 ビルドした結果を Nginx で配信

25.6 Kubernetes へのデプロイ

このセクションでは、作成したアプリケーションの Docker イメージを Kubernetes の上で動かすための基本について説明します。

25.6.1 Kubernetes の概要

Kubernetes は Google が自社の基盤を OSS として 1 から再実装したソフトウェアです。アプリケーションのデプロイ単位として Docker コンテナを用いており、開発者はコンテナイメージを作成して配信することによって、Kubernetes の上でアプリケーションを動かすことができます。

コンテナアプリケーションを本番で稼働するにあたっては、サービスを停止せずにアップデートする方法や、複数のコンテナを水平にスケールしながら負荷分散する機能など、さまざまな運用面での課題を解決する必要がありますが、Kubernetes ではそれらの機能を一貫して提供してくれるメリットがあります。

25.6.2 Kubernetes をローカルで実行する

Kubernetes をローカルで実行するには以下のようなツールを使う方法があります。

- minikube
- kind
- microk8s

このうち、minikube と kind では手元の Docker 上で Kubernetes を動かすことができるため、Linux 上で Docker を動かさずとも手元の macOS や Windows 上で Docker Desktop を使って簡単に Kubernetes を立ち上げることができます。microk8s に関しては Linux 上でのサポートに限られます (特に Ubuntu が推奨されます) が、依存するパッケージが少ないためインストールがシンプルであるメリットがあります。ここでは minikube を使って Kubernetes を動かしてみましょう。

minikube のインストール方法は以下の公式ドキュメントにあります。

- <https://minikube.sigs.k8s.io/docs/start/>

```
# minikube を起動
$ minikube start
^^f0^^9f^^98^^84  Ubuntu 20.04 上の minikube v1.12.1
^^e2^^9c^^a8  Automatically selected the docker driver

^^e2^^9d^^97  'docker' driver reported a issue that could affect the performance.
^^f0^^9f^^92^^a1  Suggestion: enable overlayfs kernel module on your Linux

^^f0^^9f^^91^^8d  Starting control plane node minikube in cluster minikube
^^f0^^9f^^94^^a5  Creating docker container (CPUs=2, Memory=2200MB) ...
^^f0^^9f^^90^^b3  Docker 19.03.2 で Kubernetes v1.18.3 を準備しています...
^^f0^^9f^^94^^8e  Verifying Kubernetes components...
^^f0^^9f^^8c^^9f  Enabled addons: default-storageclass, storage-provisioner
^^f0^^9f^^8f^^84  Done! kubectl is now configured to use "minikube"
```

minikube のセットアップが終わったら、kubectl をインストールします。kubectl は Kubernetes の CLI ツールで、Kubernetes そのものとは別途インストールする必要があります。各環境でのセットアップは以下のサイトを参考に行ってみてください。

- <https://kubernetes.io/ja/docs/tasks/tools/install-kubectl/>

これで Kubernetes を触るための準備が整いました。次に、アプリケーションの準備を行います。

あらかじめ、手元の Docker で任意のタグを付けてアプリケーションをビルドしておきます。例では typescript-kubernetes:1.0.0 とします。

```
$ docker build -t typescript-kubernetes:1.0.0 .
$ docker images
# イメージ一覧が返ってくることを確認
```

そうしたら、以下の YAML ファイルを作成します。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: typescript-kubernetes-deployment
spec:
  selector:
    matchLabels:
      app: typescript-kubernetes
  replicas: 3
  template:
    metadata:
      labels:
        app: typescript-kubernetes
    spec:
      containers:
        - name: typescript-kubernetes
          image: typescript-kubernetes:1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
---
kind: Service
apiVersion: v1
metadata:
  name: typescript-kubernetes-service
labels:
  app: typescript-kubernetes
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: typescript-kubernetes
type: ClusterIP
```

作成した YAML を Kubernetes に適用します。

```
$ kubectl apply -f app.yaml
deployment.apps/typescript-kubernetes-deployment created
service/typescript-kubernetes-service created
$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

(次のページに続く)

(前のページからの続き)

```

typescript-kubernetes-deployment-8bfd76d4c-2tsl6    1/1    Running    0        3m16s
typescript-kubernetes-deployment-8bfd76d4c-h6sdz    1/1    Running    0        3m13s
typescript-kubernetes-deployment-8bfd76d4c-sg2jz    1/1    Running    0        3m12s
$ kubectl get service
NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kubernetes                         ClusterIP           10.96.0.1       <none>           443/TCP          35m
typescript-kubernetes-service      ClusterIP           10.107.196.16   <none>           80/TCP           ↵
↵ 7m10s

```

作成した Deployment(複数のコンテナアプリケーションをまとめて管理できるリソース) と Service(複数のコンテナアプリケーションをロードバランスしてくれるネットワークリソース) が稼働していることを確認したら、今度は動作を確認します。手元のシェルで `kubectl port-forward` を実行し、Kubernetes 上のアプリケーションを手元のブラウザで接続できるようにします。

```

$ kubectl port-forward service/typescript-kubernetes-service 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80

```

ブラウザで `localhost:8080` にアクセスすると、作成されたアプリケーションが `nginx` の上で動いていることが確認できます。なお、本番などでマネージドサービスを利用する場合、ClusterIP + port-forward を利用しなくとも、以下のように LoadBalancer サービスを使用することで、パブリッククラウドのロードバランサーと簡単に連携させることができます。

```

kind: Service
apiVersion: v1
metadata:
name: typescript-kubernetes-service
labels:
  app: typescript-kubernetes
spec:
ports:
- port: 80
  targetPort: 80
selector:
  app: typescript-kubernetes
type: LoadBalancer

```

最後に、作成した minikube の環境を削除します。

```

$ minikube delete
^^f0^^9f^^94^^a5  docker の「minikube」を削除しています...
^^f0^^9f^^94^^a5  Deleting container "minikube" ...
^^f0^^9f^^94^^a5  /home/kela/.minikube/machines/minikube を削除しています...
^^f0^^9f^^92^^80  Removed all traces of the "minikube" cluster.

```

25.7 まとめ

TypeScript で作成したコードが価値を生むのは、何かしらのデプロイ作業を通じてになります。本章ではそのデプロイの方法について、さまざまな方法を説明しています。

Docker は強力な武器です。ぜひ使いこなせるようになってください。

第 26 章

使用ライブラリのバージョン管理

現代のソフトウェア開発では多くのライブラリに依存して開発を行います。大抵、システムの開発開始時には、その時点での新しいライブラリを使うと思いますが、長期的な運用を考えると、使用するライブラリやパッケージの更新というのは避けて通れない話です。本章ではそのあたりについて紹介します。

前半では技術的な説明ですが、後半はソフトウェア開発に詳しくない人向けに説明する時に参考用の啓蒙的な内容になっています。コストをかけてバージョン更新をしなければならない理由がわかっている方は前半だけ読んでおけば良いです。

26.1 バージョンとは

現在提供されているシステムの多くは 3 つの数字を並べたバージョンを使っています。

- x.y.z

x をメジャーバージョン、y をマイナーバージョン、z をパッチバージョンと呼んだりします。例えば、12.0.4 とか、3.7.3 とかそういうやつです。

Windows は商品名としては 95 とか 2000 とか 10 とかつけたりもしますが、内部的には 2 つの数字の列になっています。18362.175 とかそういうやつです。

数字付けのルールは各システムが勝手につけることが多いので、全部のシステムで統一的なルールというのは、大きい数字ほど新しい、ぐらいのものです。昔は x が偶数が安定板、y が奇数が開発版みたいなのがよく使われたりもしていましたが、マーケティングの都合でいきなり x が大幅にジャンプしたりとかあります。x が上がると後方互換性がないバージョンアップだが、y の更新は後方互換性があるとかもよく見かけます（セマンティックバージョンング）が、気分で x をあげるシステムもあります（Linux とか）。

フロントエンド開発でよく出てくるルールがセマンティックバージョンングです。この 3 桁でバージョン間の大小を一意に定めます。これにより、「古い」「新しい」が判断できるようになります。

細かいルールはもっといろいろあり、例えば、1.0.0 よりも 1.0.0-alpha の方が古い、というルールもあります。詳しくは次のページを参照してください。

- セマンティック バージョニング 2.0.0

26.1.1 バージョンのサポートの考え方

サポートの考え方は大きく 3 種類ぐらいですね。

- 最新メジャーバージョンのみサポート
- 最新のいくつかのメジャーバージョンのみサポート
- 最新のメジャーバージョンと、特定の不連続なメジャーバージョン (LTS) のみサポート

たいてい、メジャーバージョンごとにサポート期間を設定することがほとんどです。開発リソースの多いプロジェクトでは、複数メジャーバージョンを同時サポートします。小さいプロジェクトや個人プロジェクトでは最新バージョンのみサポートというケースがほとんどです。また、変化の早いブラウザも最新バージョンのみです。

最新のいくつかのメジャーバージョンというのは、例えば Oracle 社製のデータベースは最新 2 バージョンのみサポートとかそういうやつです。ただ、ウェブのフロントエンド開発ではあまりみないかもしれません。

よく見るのが LTS (ロングタームサポート) という長期サポートバージョンを定めているライブラリとかツールです。

Node.js は、現在の半年ごとにメジャーバージョンアップします。最新のメジャーバージョンのものは current 扱いです。奇数バージョンは current でなくなるとすぐにサポートが終わりますが、1 年に一回出る偶数バージョンは、current でなくなると (次のバージョンが出ると)LTS になり、2 年半サポートされます。現在の最新は 14 で 14 はメジャーバージョンアップ対象ですが、現在も活発に機能追加が行われていますので、LTS にはなっていません。15 が出ると 14 が LTS になります。

例えば、12 系は、2019 年 10 月 21 日に 12.13 と 13.0 がリリースされ、12.13 が LTS になりました。執筆時点では 12.18 がリリースされています。しかし、12.12 以前は LTS ではなく、新機能もどんどん追加されます。このあたりは他のソフトウェアと異なるルールになっていますが、アルファ版やベータ版では利用する人がおらず、数値が大きく上がったリリースのタイミングで使い始める人が多く、そこで初めて不具合が出て報告されたりするので、多くの人に使ってもらってバグ修正をするという OSS のエコシステムを考慮すると合理的な考え方であると思います。

ライブラリでは Angular がすでに LTS を含む運用をしており、現在最新の 8 は、次の 9 が出ると LTS になって、その後 1 年サポートされます。Vue.js も、3.x が出たら 2.x の最終版が LTS として 18 ヶ月サポートされると宣言されています。

26.2 バージョン選びの作戦

Node.js やアプリケーションで使うパッケージのバージョン選びの戦略は主に3つあります。バージョンアップ作業には時間がかかります。バージョン更新そのものに加えて、確認の工数もかかります。その分、新機能開発の工数は削減されます。時間がかかるということはそれに対して費用も発生します。どこの費用を使ってやるか、どこに請求するか、稟議をどう投げるかの考慮が必要です。なので、それをどこで消化するかを決めるのがバージョン選びの大切なところ。「そんな決めなくてもなんとかなるよ」というのは、チーム内の誰かの善意（やる気）に甘えているだけなので要注意です。

いくつか考えられる作戦を列挙してみます。どれか一つを選ぶというよりは、状況に応じて複数のパターンを利用すると良いでしょう。

26.2.1 1. 最新バージョンを積極的に選ぶ

常に最新バージョンを取り込んでいくスタイルです。バージョンアップタスクの分散化です。日々最新のものを取り込むスタイルです。例外としては iOS のモバイル開発で、最新 iOS の GM が出てから、ユーザーの手に最新の OS が渡り始める 1-2 週間で動くものを作って提出しなければなりません。あと、React とか、安定版などなく、最新版のみが更新されていくライブラリがコアとなると必然的にこうなります。

新しいバージョンを試してその知見を公開するだけでもありがたいと言われるというメリットもあります。バグ報告とかでそのソフトウェアに貢献もできるかもしれません。類似パターンとしては、ベータ版も試すパターン、最新の master のバージョンも試すパターンもあります。

ただ、現在は複数のライブラリを組み合わせる行うため、新しすぎるバージョンでは他の連動して動くライブラリの準備ができていないケースもあり、予想外に時間が取られることがあります。特に、Babel などの影響の大きなライブラリのバージョンには要注意です。以前、Next.js が、Babel 7 beta42 だかの中途半端なバージョンに依存して、いろいろ食い合わせが悪くて苦労したことがあります。

26.2.2 2. LTS を中心に組み立てる

メインの部分（Node.js とか Angular とか）を LTS で固め、その周りをそれに準拠する形で固めていきます。メリットとしては、スケジュールが見えているので、あらかじめバージョンアップのタイミングを計画に折り込みやすい（年間予算の計画が立てやすい、稟議にかけやすい）というものがあります。LTS のリリースの前には安定化のための期間が置かれており、比較的問題が起きにくいでしょう。

ただ、LTS が提供されているものならこれでいけますが、現状、LTS を提供しているコアのライブラリはあんまりないので、現状は Angular を使っている場合のみしか適用できません。Vue はそのうち始まりますね。React の場合は、LTS はないが、きちんと検証された組み合わせであることを期待して、Next.js のメジャーバージョンアップに淡々とついていく、という方法はあります。

LTS を使う場合も、LTS の範囲内で最新のものを積極的に使うか、固定化するかみたいな細かい作戦の差があります。

Angular で LTS 固定運用をしてみた感想でいうと、TypeScript のバージョンが古く、lit-element が利用できない、LTS の範囲内だと次に説明するセキュリティ脆弱性の更新が得られないことがある、といったデメリットがあるのは感じました。

26.2.3 3. セキュリティの脆弱性が検知されたものを更新する

ライブラリのセキュリティの診断はここ数年でいくつもの手法が利用可能になっています。数年ぐらい前は `snyk` にユーザー登録をして `snyk` コマンドを実行して検知していました。現在は `npm` コマンドを使って `npm install` するだけでも脆弱性のあるパッケージが検知できます。また、GitHub にソースコードをアップロードすると、GitHub が検知してくれます。

脆弱性のあるパッケージは自分が直接インストールしたもので発生するだけではなく、そのパッケージが利用している別のパッケージのさらに依存しているパッケージが・・・みたいな依存の深いところで起きがちです。特にウェブフロントエンド開発をしていると、依存パッケージ数が簡単に4桁とかいってしまうので、すべてを目視で確認するのは難しいです。自動検知を活用しましょう。

なお、検知されたすべてを修正しないといけないかというと、そんなことはありません。例えば `node-sass` は `request` という外部ネットアクセスのライブラリに依存しており、これが更新されなくて脆弱性が検知されたことがあります。node-sass は CSS を書きやすくしてくれるユーティリティで、実行時には動作しません。ビルド前の CSS の中で外部リソースに依存していないのであればこのライブラリは使われないはずで、「これは検知されたが影響はありません」というように、説明がつけば OK です。

26.3 バージョンアップの方法

`npm` コマンドにはバージョンアップを支援するサブコマンドがいくつかあります。

まずは、現在のバージョンを知るための、`npm outdated` コマンドです。これを実行すると、現在インストールされているバージョン、現在のバージョン指定でインストールされる最新バージョン、リリースされている最新バージョンが表示されます。

`npm` でインストールするときは、最新のバージョンがインストールされますが、`package.json` 上はその時のバージョンが固定されているわけではありません。`"browserify": "^16.2.0"` のように、`^` や `~` が先頭に付与されています。`^` であれば 16.3.0 があればそれも利用する、`~` であれば 16.2.1 であれば利用するなど、マイナーバージョンやパッチバージョンの変更は吸収する意思がありますよ、という表示になっています。「現在のバージョン指定でインストールされる最新バージョン」というのは、変更可能な範囲での最新という意味です。

更新する場合は、`npm update` コマンドを使用します。

```
# まとめて更新
$ npm update
```

(次のページに続く)

```
$ npm outdated
```

<u>Package</u>	<u>Current</u>	<u>Wanted</u>	<u>Latest</u>	<u>Location</u>
ava	0.25.0	0.25.0	2.4.0	shadow-fetch
browserify	16.2.0	16.5.0	16.5.0	shadow-fetch
codecov	3.0.0	3.6.1	3.6.1	shadow-fetch
eslint	4.19.1	4.19.1	6.6.0	shadow-fetch
express	4.16.3	4.17.1	4.17.1	shadow-fetch
flow-bin	0.70.0	0.70.0	0.112.0	shadow-fetch
node-fetch	2.1.2	2.6.0	2.6.0	shadow-fetch
nyc	11.7.0	11.9.0	14.1.1	shadow-fetch
streamtest	1.2.3	1.2.4	1.2.4	shadow-fetch
unfetch	3.0.0	3.1.2	4.1.0	shadow-fetch

図1 npm outdated の実行結果

(前のページからの続き)

```
# 一部だけ更新
$ npm update express
```

```
$ npm outdated
```

<u>Package</u>	<u>Current</u>	<u>Wanted</u>	<u>Latest</u>	<u>Location</u>
ava	0.25.0	0.25.0	2.4.0	shadow-fetch
eslint	4.19.1	4.19.1	6.6.0	shadow-fetch
flow-bin	0.70.0	0.70.0	0.112.0	shadow-fetch
nyc	11.9.0	11.9.0	14.1.1	shadow-fetch
unfetch	3.1.2	3.1.2	4.1.0	shadow-fetch

図2 npm update を実行し、Current が Wanted になった

この場合、メジャーバージョンアップしたライブラリは更新されません。その場合は手動でインストールします。

```
$ npm install ava@2.4.0
```

26.3.1 セキュリティ目的の自動バージョンアップ

GitHub 上、あるいは開発環境での snyk コマンドや、npm install 時に脆弱性診断が行われます。また、インストールを行わなくても、npm audit コマンドを実行しても脆弱性が報告されたパッケージがあると検知されます。何かしらを検知したら次のコマンドで可能なものを修正します。

```
$ npm audit fix
```

これだけでちょっとした修正は完了できるはずですが、メンテナンスがあまりされていないパッケージの場合は、脆弱性ありで修正がないまま放置されていたりします。あるいは、脆弱性ありのバージョンに依存したまま、という

こともあります。こうなると `npm audit fix` をしても脆弱性があるモジュールでも修正できなくなってきました。それをトリガーにして一部のパッケージのメジャーバージョンアップが必要となります。もちろん、他の戦略をとっていても重大なセキュリティの場合には応急処置せざるを得ない可能性があります。まあ、セキュリティの緊急度が高いほど、いろんなメジャーバージョンに対してパッチが発行されることもあり、逆に簡単かもしれません。

あまりにも脆弱性が放置されているライブラリがある場合は、バージョンアップではなくて、類似の別のライブラリに置き換える、というのも選択に入ります。

26.4 バージョンアップ時のトラブルを減らす

バージョンアップでのトラブルを完全にゼロにはできません。ただ、日頃からの心がけで少し楽にすることはできます。

26.4.1 CI をしておく

普段から CI をしておくことで、いざバージョンアップ時の確認の補助に使えますし、最近では、利用されているモジュールの中に脆弱性のある古いバージョンが紛れていないかの自動検知が行えるようになってきています。

JavaScript と比べた TypeScript の場合、一番有利なのがここですね。ライブラリの API が変わってビルドができない、というのが検知できるのがメリットです。もちろん、ロジックなどが正しく動くかどうかというテストもあるに越したことはありません。

26.4.2 こまめにバージョンアップ

セキュリティのバージョンアップ、パッチバージョンアップなど、小さい修正はこまめにやっておけば、いざというときにあげるバージョンの差が小さくなります。例えば、1.6.5 から 1.6.6 で、0.0.1 だけあげたら問題が起きた、と分かれば、エラーの原因の追求、問題の報告が極めて簡単になります。

26.4.3 人気のある安定しているライブラリを利用する

身もふたもないのですが、API の `breaking change` が頻繁に行われないライブラリを選べば楽になります。後方互換性やサポートポリシーについて言及があるライブラリが良いです。あと、人気があるライブラリであれば、バージョンアップで困ったときに情報が入手しやすくなります。

26.4.4 ライブラリやフレームワークを浅く使う

ライブラリやフレームワークを使う場合、メジャーな一般的な使い方からなるべく外さないようにします。ライブラリをラップして完全なオレオレフレームワークを作るとかすると、バージョンアップ時の作業が多くなります。また、メジャーな使い方に近づけておけば、ネットで情報を調べるときにも問題が発見しやすくなりますし、チームメンバーが途中から入ったとしても、実は最初から使い方を知っている、ということも期待できるかもしれません。

26.4.5 （参考）式年遷宮

近年のウェブフロントエンドは、たくさんの小さなツールやライブラリを組み合わせることで多いことが多くです。ライブラリの数が多ければ多いほど組み合わせの数は爆発していきます。世界であまり多くの人が試していないバージョンの組み合わせでやらざるを得なくて・・・ということも起きるかもしれません。

複雑化するにつれて、プロジェクトの新規作成を手助けするツールが提供されることが増えてきました。特に Vue.js の CLI はきちんと作り込まれています。いっそのこと、バージョンアップ作業をするのではなく、CLI ツールを最新化して、それでプロジェクトを新規に作り、それに既存のコードを持ってくるという方法もありかもしれません。

26.5 なぜバージョンを管理する必要があるのか

なぜライブラリのバージョンの管理が必要なのでしょう？バージョン固定じゃダメなのでしょう？

プログラムを開発するときは、他のツールやライブラリを当たり前に使います。これは今に始まったことではなく、はるか昔からそうですね。OS 組み込みの機能だけで開発するとしても、OS ベンダーの提供する開発ツール、OS の機能を使うライブラリ（API）は最低限使います。

例えば、Java で開発する場合、Java の言語、言語組み込みのライブラリは使いますし、Gradle みたいな別なビルドツールやらも使います。SpringBoot みたいなライブラリも使います。それぞれ、どのバージョンを使うかというのをスタート時に決めますし、メンテ期間等で見直しをする必要がでてきます。

なぜ固定ではダメかということ、主に 2 つの理由があります。

26.5.1 機能的な問題

それぞれのツールやライブラリは、それぞれの開発元が考えるライフサイクルで更新されていきます。そのタイミングで、機能が追加されることもあれば、過去のバージョンで提供されていた機能が削られたり、挙動が変わったり、というのがあります。その過去のバージョンがもう手に入らない、ということもあります。

ウェブの場合だと、アプリケーション側でコントロールできないものにブラウザバージョンがあります。ある程度は使用バージョンを固定するなども業務システムではありますが、古いブラウザでしか動かないとかはダサいです

よね。

例えば Flash を使っていると、もう動かすことはできません。実装していた機能を取り除いて、その互換実装に置き換える、という作業が発生します。

もっと小さい例でいえば非推奨になっている React の特定のライフサイクルメソッドの関数（`componentWillMount`）を使っていたら、React 17 が出るとそのアプリケーションは動かなくなってしまいます。これは React のバージョンを固定してしまえばなんとかなるのかもしれませんが、追加の機能を入れようとして別のライブラリを入れようとしたときに、それが React 16 では動かなくて、React 17 しかサポートしていないと、そのライブラリが使えないということになります。4K ブルーレイを見たいけど、うちの古いブラウン管テレビには HDMI 端子がなくてプレイヤー繋げられないわー、みたいな感じのことが起きます。

時間が経てば経つほどそのようなものが増えてきます。

26.5.2 セキュリティ的な問題

インターネットがなかった時代・接続しない時代は良かったんですが、今ではネットワーク前提のシステムが大幅に増えています。それにより、今までよりもセキュリティのリスクにさらされる機会は増えています。また、ネットワークに直接アクセスしないシステムであっても、USB メモリ経由でやってきたワームの攻撃を受けるなどがあります。

セキュリティに関しては存在（& 攻撃方法）が報告されているセキュリティホールを放置して、システムを危険にさらされると、システムの提供元や開発元が責任を追求されることになります。「無能で説明できることに悪意を見出してはいけない」という格言があります。ただ、これらは「悪意」を持っていると誤解する人が多いからこそこういう言葉が生まれたのだと思います。あと、僕個人としては「時間不足で説明できることに無能を見出してはいけない」という持論があります。組み合わせると、忙しくて直せなかったとしても、「悪意があってユーザーを危険にさらしたのだ」と批判される恐れがあるということです。加害者になってしまうのです。

現代のシステムは数多くの部品で組み上げられています。ゼロからすべてのコードを自分で書くことはありません（ほとんど）。脆弱性に対する防御は社会的な仕組みが構築されています。特定のライブラリやツールに脆弱性があると、その攻撃手法などを報告する窓口があります。また、そこから開発元にこっそり連絡がいき（対策されていない時点での存在発表はそれ自体が加害行為になる）、脆弱性が修正されたバージョンのリリースと同時に公表、という流れです。

同時といっても、大きな問題は発表されたら即座に対策を取らないと、加害者になりかねません。そのためには、最新の修正済みのバージョンを入れる必要が出てきます。

問題はすごく古いバージョンのサポートまでは行われえない点です。だいたい、大きめの OSS や商用のミドルウェアやライブラリを出しているベンダーであれば、きちんとサポートポリシーを定義して、バージョンごとのサポート期限を定めています。ただし、そこから外れてしまうと、よっぽど大きな問題でない限りは更新が提供されないことがあります。そのため、普段から更新を心がけていないと、必要な修正の入ったバージョンへの更新が遠すぎて、なかなか適用できないということもあります。

26.5.3 バージョン管理をしなかった場合のデメリットまとめ

- 既存機能が動かなくなる
- 世間一般では普通の新規機能の追加が困難になっていく
- セキュリティの修正が提供されずに、システムに穴が開いたままになりかねない
- いざ、おおきなインシデントが発生したときに、その変更を取り込むのが困難になる

26.6 まとめ

なぜバージョンアップが必要なのか、そのための方法などについて紹介してきました。

常に更新しつづけるシステムであっても、バージョンの更新がおろそかになってしまうことがよくあります。バージョンアップは自社サービスであってもそうでなくても、保守として工数を確保して行う必要があります。影響を考慮してタイミングを見極めて行ったり、セキュリティ上必要であれば他の作業を止めてでも更新してデプロイなどスケジュールにも影響がありえる話になってきます。

適切にコントロールすれば痛みを減らせる分野でもありますので、本章の内容を頭の片隅に置いてもらえると幸いです。

第 27 章

ブラウザ環境

今も昔も、ウェブフロントエンドが今の JavaScript/TypeScript の主戦場です。本章ではそのウェブフロントエンドの環境について紹介します。

現在、メジャーなウェブフロントエンドのフレームワークというと、React、Vue.js、Angular です。本書で取り上げるのは React と Vue.js です。React 以外に React をベースにした統合フロントエンドフレームワークとなっている Next.js も取り上げます。Angular は 2 以降から TypeScript に書き直されて、TypeScript 以外では書けなくなり、最初から TypeScript が有効な状態でプロジェクトが作成されるため、説明は割愛します。

本章では、それぞれの環境の共通部分について紹介します。

27.1 ウェブアプリケーションにおけるビルドツールのゴール

JavaScript でビルドといっても、いろいろなステップがあります。

1. TypeScript や Babel を使って、ターゲットとなるバージョンの JavaScript に変換
2. SCSS とか PostCSS を使ってブラウザにない機能を使って書かれた CSS を素の CSS に変換
3. webpack などを使って、1 つの JavaScript ファイル、もしくは遅延ロードをする JS ファイル群を生成

実際には綺麗にステップが分かれることなく、バンドラーが import 文を追跡しつつファイルを探し、.ts を見つけては TypeScript で処理して（コンパイル）、コード中に SCSS を見つけては SCSS の処理系に投げて、一つのファイルにまとめる（バンドル）・・・みたいな工程を行ったりきたりしながらビルドします。以前は、これに Jake、Gulp、Grunt などのタスクランナーも組み合わせやってましたが、今は webpack 単体に ts-loader などを組み合わせる感じで一通りできます。webpack がシェア 80% で一強です。

しかし、ファイルを集めてきてコンパイルしつつ 1 ファイルにバンドルし、必要に応じて minify 化してサイズを小さくするといったタスクは、比較的重い処理です。そのため、ウェブのフロントエンドのビルド環境は、開発者のストレスを軽減するために、さまざまな機能を備えてきました。

- バンドルツールは毎回ファイルを読み込むのではなく、メモリに常駐し、変更されたファイルだけすばやく読み込んでバンドルを完成させる
- ブラウザに対して、変更検知を伝えるコードを差し込んでおくことで、すばやくブラウザをリロードして最新のファイルを読み込む仕組み

これらを実現するのが開発サーバです。開発サーバは HTTP サーバとして起動し、JavaScript や HTML、CSS を配信するサーバとしてブラウザからは見えます。その裏では、ファイルシステムのソースコードを監視し、変更があったら即座にビルドをして動作確認までのリードタイムを短くします。それだけではなく、その開発中のウェブサイトを見ているブラウザに対して強制リセットをしかけたり（ホットリロード）、起動中に JavaScript のコードを差し替えたり（ホットモジュールリプレースメント）といったことを実現します。また、TypeScript だけではなく、CSS でも、事前コンパイルでコーディングの効率をあげる方法が一般化しているため、この設定も必要でしょう。

この分野では数多くのツールがあります。スキルのある人は自分の好みや要件に合わせてツールを選択すると良いでしょう。TypeScript の対応についても、最初から対応していたり、後からプラグインで拡張など、いろいろなものがあります。

webpack は細かくカスタマイズできますし、豊富な開発リソースで頻繁にリリースされています。ツリーシェイキングといった不要なコードを除外してサイズを小さくする機能にいち早く取り組んだり、業界をリードしています。困った時に検索すると情報も多く出てきます。一方で、TypeScript 対応で開発サーバや CSS 対応など、機能を足していこうとすると設定やプラグインが増えていきます。特に、React の JSX 構文を利用する場合は、バンドラーの処理の前段で TypeScript を JavaScript に変換したあとに Babel を使い、最後にバンドラーで 1 ファイルや複数ファイルにまとめるなど、ビルドのパイプラインが多段になりがちです。React や Vue の環境構築ツールや Next.js、Nuxt.js などは webpack を内包して、少ない設定の量で一定の機能を備えたビルド環境を整えてくれます。本書でも、webpack そのものを紹介することはしませんが、これらのツールの紹介をします。

他にも数多くのバンドラーがあります。webpack 以降に出てきたものの多くは設定が少ない、あるいは設定ファイルが不要（ゼロコンフィギュレーション）を売りにしたものが数多くあります。Rollup は人気のあるバンドラーで、TypeScript を使うにはプラグインが必要ですが、そうでないのであれば設定がほとんど必要ありません。Rollup をベースに TypeScript サポートを最初から組み込んだ microbundle もあります。HTML や CSS のビルドもできて開発サーバも全てついてくるオールインワンでビルド速度を重視した Parcel や、Go 製でビルド速度に特化した esbuild もあります。一方で、カスタマイズが必要なので最初からカスタマイズ前提で CLI を提供しない Fusebox などもあります。

第 28 章

ブラウザ関連の組み込み型

- `fetch`
- `FormData`
- `EventListener`
- `EventSource` と `WebSocket`
- `LocalStorage` と `SessionStorage`

28.1 Polyfill、Ponyfill

28.2 `fetch`

確実に Node.js だけでしか使われないコードであれば `node-fetch` <<https://www.npmjs.com/package/node-fetch>> をインポートして利用すれば十分です。

もし、ブラウザでもサーバーでも実行されるコードの場合 <https://www.npmjs.com/package/isomorphic-unfetch>
<https://www.npmjs.com/package/isomorphic-fetch> <https://www.npmjs.com/package/fetch-ponyfill>

28.3 `FormData`

<https://www.npmjs.com/package/form-data>

28.4 EventListener

28.5 EventSource と WebSocket

<https://www.npmjs.com/package/eventsourcing> <https://www.npmjs.com/package/ws> <https://www.npmjs.com/package/isomorphic-ws>

28.6 LocalStorage と SessionStorage

<https://www.npmjs.com/package/localstorage-memory> <https://www.npmjs.com/package/node-localstorage>

第 29 章

React の環境構築

React は Facebook 製のウェブフロントエンドのライブラリで、宣言的 UI、仮想 DOM による高速描画などの機能を備え、現在のフロントエンドで利用されるライブラリの中では世界シェアが一位となっています。React によって広まった宣言的 UI はいまやウェブを超え、iOS の Swift UI や Android の Jetpack Compose、Flutter など、モバイルアプリの世界にも波及しており、このコミュニティが近年のムーブメントの発信源となることも増えています。

JavaScript は組み合わせが多くて流行がすぐに移り変わっていつも環境構築させられる（ように見える）とよく言われますが、組み合わせが増えても検証されていないものを一緒に使うのはなかなか骨の折れる作業で、結局中のコードまで読まないといけなかったりとか、環境構築の難易度ばかりが上がってしまいます。特に Router とかすべてにおいて標準が定まっていない React はそれが顕著です。それでも、もう 2013 年のリリースから長い期間が経ち、周辺ライブラリも自由競争の中で淘汰されたりして、定番と言われるものもかなり定まってきています。環境構築においてもほぼ全自動で済むツール create-react-app が登場しましたし、オールインワンな Next.js も利用できます。そろそろ「枯れつつあるフレームワーク」として React を選ぶこともできるようになると感じています。

29.1 create-react-app による環境構築

React は標準で環境構築ツール create-react-app コマンドを提供しています。これを使って環境構築を行う場合、`--template typescript` オプションをつけると TypeScript 用のプロジェクトが作成できます。

```
$ npx create-react-app --template typescript myapp

Creating a new React app in /Users/shibukawa/work/myapp.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...
:
Happy hacking!
```

これで開発サーバーも含めて設定は終わりです。次のコマンドが使えます。

- `npm start`: 開発サーバー起動
- `npm run build`: ビルドして HTML/CSS/JS ファイルを生成
- `npm test: jest` によるテストの実行

構築した環境は、すべてをバンドルした `react-scripts` というコマンドが開発サーバー、コンパイル、バンドルなどすべてを行います。このスクリプトはカスタマイズのポイントがなく、`tsconfig.json` があるだけです。このスクリプトはときに厄介な動きをすることもあります。例えば、`tsconfig.json` の変更を勝手に戻したりします。`npm run eject` を実行すると、このスクリプトが分解されて `config` フォルダに出力されます。これを変更することで、出力先から `webpack` の設定まで、細かい内容が変更できるようになります。また、どのような動作が設定されていたのかも確認できます。

29.2 Next.js

Next.js は Vercel 社が開発している React のオールインワンパッケージです。`next` コマンドでプロジェクトを作成すると、`webpack` によるビルドサーバーやコンパイルに必要な設定だけではなく、フロントエンド側の便利機能の CSS in JS、シングルページアプリケーションのための Router などの開発環境整備が完了した環境が一発で作成できます。バージョン 9 からは TypeScript にも対応済みのプロジェクトが作成されるようになりました。よく使う部品が最初から設定されているため、ツールやライブラリの調整に手間がかからないのが良い点です。

デフォルトではサーバーサイドレンダリングを行うフロントエンド機能のみですが、カスタムサーバー機能を使えば、Express.js などの Node.js の API サーバーにサーバーサイドレンダリング機能などを乗せることができます。Express.js への薄いラッパーになりますので、Express.js の知識を利用して、API サーバー機能も同一のサーバー上に追加できます。また、サーバーサイドレンダリングを使わずに静的な HTML と JavaScript コードを生成することも可能です。

Next.js の良いところは、よく使うツールやライブラリー式が検証された状態で最初からテストされている点にくわえ、issue のところでもアクティブな中の人々がガンガン回答してくれていますし、何よりも多種多様なライブラリとの組み合わせを `example` として公開してくれている^{*1}のが一番強いです。お仕事でやっていて一番ありがたいのはこの相性問題の調査に取られる時間が少なく済む点です。

それにプラスして、自分で設定すると相当難易度の高いサーバーサイドレンダリング、静的コンテンツの生成など、さまざまなパフォーマンス改善のための機能に取り組んでいます。

本書執筆時点のバージョンは 9.4 です。バージョンが変わると、方法が変わる可能性があります。

次のようにコマンドをタイプし、質問に答えると（プロジェクト名、標準構成で作るかサンプルを作るか）、プロジェクトフォルダが作成されます。

```
$ npx create-next-app
```

^{*1} <https://github.com/zeit/next.js/tree/canary/examples>

TypeScript には対応していますが、設定ファイルを置いて拡張子を変える必要があります。作成されたプロジェクトフォルダの中で次のコマンドをタイプします。

```
$ touch tsconfig.json
$ npm install --save-dev typescript @types/react @types/node
```

次のコマンドが使えます。

- `npm run dev`: 開発サーバー起動
- `npm run build`: ビルドして本番モードの HTML/CSS/JS ファイルを生成
- `npm start`: ビルドしたアプリを本番モードのアプリケーションを起動

一度、開発サーバーを起動すると、`tsconfig.json` を認知して、それに初期値を設定したり、`next-env.d.ts` というアンビエント型を書くファイルを作成します。あとは手動で、`.js` ファイルをリネームしていけば設定完了です。JSX が含まれるファイルは `.tsx` に、それ以外のファイルは `.ts` にします。

`tsconfig.json` は今までと少し異なります。後段で Babel が処理してくれる、ということもあって、モジュールタイプは ES6 modules 形式、ファイルを生成することはず、Babel に投げるので `noEmit: true`。React も JSX 構文をそのまま残す必要があるので `"preserve"` となっています。JS で書かれたコードも一部あるので、`allowJs: true` でなければなりません。

Next.js は [CSS Modules](#) に対応しているため、`button.tsx` の場合、`button.module.css` といった名前にすることで、そのファイル専用の CSS を作成できます。もし、SCSS を使う場合は次のコマンドをタイプすると `.module.scss` が使えるようになります。

```
$ npm install sass
```

詳しくは [Next.js の組み込み CSS サポートページ（英語）](#) を参照してください。

29.3 React の周辺ツールのインストールと設定

`create-react-app` の方はすでに設定済みですが、Next.js は ESLint やテストの設定が行われませんので、品質が高いコードを実装するために環境整備をしましょう。ESLint を入れる場合は、React の JSX に対応させるために、`eslint-plugin-react` を忘れないようにしましょう。

```
# テスト関係
$ npm install --save-dev jest ts-jest @types/jest

# ESLint 一式
$ npm install --save-dev prettier eslint
  @typescript-eslint/eslint-plugin eslint-plugin-prettier
  eslint-config-prettier eslint-plugin-react npm-run-all
```

ESLint は JSX 関連の設定や、.tsx や .jsx のコードがあったら JSX として処理する必要があるため、これも設定に含めます。あと、next.config.js とかで一部 Node.js の機能をそのまま使うところがあって、CommonJS の require を有効にしてあげないとエラーになるので、そこも配慮します。

リスト 1 .eslintrc

```
{
  "plugins": [
    "prettier"
  ],
  "extends": [
    "plugin:@typescript-eslint/recommended",
    "plugin:prettier/recommended",
    "plugin:react/recommended"
  ],
  "rules": {
    "no-console": 0,
    "prettier/prettier": "error",
    "@typescript-eslint/no-var-requires": false,
    "@typescript-eslint/indent": "ignore",
    "react/jsx-filename-extension": [1, {
      "extensions": [".ts", ".tsx", ".js", ".jsx"]
    }]
  }
}
```

最後に npm から実行できるように設定します。

リスト 2 package.json

```
{
  "scripts": {
    "test": "jest",
    "watch": "jest --watchAll",
    "lint": "eslint .",
    "fix": "eslint --fix ."
  }
}
```

29.4 UI 部品の追加

React や Next.js にはカッコいい UI 部品などはついておらず、自分で CSS を書かないかぎりには真っ白なシンプルな HTML になってしまいます。React 向けによくメンテナンスされている Material Design のライブラリである、Material UI を入れましょう。ウェブ開発になると急に必要のパッケージが増えますね。

- <https://material-ui.com/>


```
$ npm install --save @material-ui/core @material-ui/icons
```

create-react-app で作成したアプリケーションの場合の設定方法は以下にサンプルがあります。

- <https://github.com/mui-org/material-ui/tree/master/examples/create-react-app-with-typescript>

まずは `src/theme.tsx` をダウンロードしてきて同じパスに配置します。これがテーマ設定を行うスクリプトなので色のカスタマイズなどはこのファイルを操作することで行ます。次に `src/index.tsx` のルート直下に `ThemeProvider` コンポーネントを起き、テーマを設定します。すべての UI はこのルートの下に作られることになりますが、このコンポーネントが先祖にいと、すべての部品が同一テーマで描画されるようになります。

リスト 3 `src/index.tsx`

```
import React, { StrictMode } from 'react';
import { render } from 'react-dom';
import CssBaseline from '@material-ui/core/CssBaseline';
import { ThemeProvider } from '@material-ui/core/styles';
import App from './App';
import * as serviceWorker from './serviceWorker';
import theme from './theme';

render(
  <StrictMode>
    <ThemeProvider theme={theme}>
      <CssBaseline />
      <App />
    </ThemeProvider>
  </StrictMode>,
  document.getElementById('root')
);
```

Next.js も同じようなことをする必要がありますが、サーバーサイドレンダリングをする都合上、Next.js では少し別の設定が必要になります。下記のサイトにサンプルのプロジェクトがあります。

- <https://github.com/mui-org/material-ui/tree/master/examples/nextjs-with-typescript>

行うべき作業は3つです。

- `pages/_app.tsx` をダウンロードしてきて同じパスに配置
- `pages/_document.tsx` をダウンロードしてきて同じパスに配置
- `src/theme.tsx` をダウンロードしてきて同じパスに配置（必要に応じてカスタマイズ）

以上により、ページ内部で自由に Material UI の豊富な UI 部品が使えるようになります。

Material UI 以外の選択肢としては、React 専用でない Web Components 製の UI 部品もあります。

- Material Web Components: <https://github.com/material-components/material-components-web-components>

- Ionic: <https://ionicframework.com/>
- Fast: <https://github.com/microsoft/fast>

29.5 React+Material UI+TypeScript のサンプル

ページ作成のサンプルです。Next.js ベースになっていますが、このサンプルに関しては create-react-app との差はごく一部です。

- Next.js は pages 以下の.tsx ファイルがページになります。このファイルは pages/index.tsx なので、<http://localhost:3000> でアクセスできます。このファイルは export default で React コンポーネントを返す必要があります。create-react-app 製のコードは src/index.tsx がルートになっていますが、そこからインポートされている src/App.tsx がアプリケーションとしてはトップページなので、ここに書くと良いでしょう。
- next/head は<head>タグを生成するコンポーネントになりますが、create-react-app の場合は [react-helmet](#) などの別パッケージが必要でしょう。
- next/link はシングルページアプリケーションのページ間遷移を実現する特殊なリンクを生成するコンポーネントです。create-react-app でシングルページアプリケーションを実現する場合は [React Router](#) などの別パッケージが必要となります。

TypeScript だからといって特殊なことはほとんどなく、世間の JavaScript のコードのほとんどそのままコピーでも動くでしょう。唯一補完が聞かない any が設定されていたのが makeStyle でした。これは CSS を生成する時にパラメータとして任意の情報を設定できるのですが、今回は Material UI のテーマをそのまま渡すことにしたので、Theme を型として設定しています。

リスト 4 pages/index.tsx

```
import { useState } from 'react';
import Head from 'next/head';
import Link from 'next/link';

import { useTheme, makeStyles, Theme } from "@material-ui/core/styles";
import {
  Toolbar,
  Typography,
  AppBar,
  Button,
  Dialog,
  DialogActions,
  DialogContent,
  DialogContentText,
  DialogTitle,
} from "@material-ui/core";
```

(次のページに続く)

(前のページからの続き)

```

const useStyles = makeStyles({
  root: (props: Theme) => ({
    paddingTop: props.spacing(10),
    paddingLeft: props.spacing(5),
    paddingRight: props.spacing(5),
  })
});

export default function Home() {
  const [ dialogOpen, setDialogOpen ] = useState(true);
  const classes = useStyles(useTheme());
  return (
    <div className={classes.root}>
      <Head>
        <title>My page title</title>
        <meta name="viewport" content="initial-scale=1.0, width=device-width" />
        <link rel="stylesheet" href="https://fonts.googleapis.com/css?
↪family=Roboto:300,400,500,700&display=swap" />
      </Head>
      <Dialog open={dialogOpen} onClose={() => {setDialogOpen(false)}}>
        <DialogTitle>Dialog Sample</DialogTitle>
        <DialogContent>
          <DialogContentText>
            Easy to use Material UI Dialog.
          </DialogContentText>
        </DialogContent>
        <DialogActions>
          <Button
            color="primary"
            onClick={() => {setDialogOpen(false)}}
            >OK</Button>
        </DialogActions>
      </Dialog>
      <AppBar>
        <Toolbar>
          <Typography variant="h6" color="inherit">
            TypeScript + Next.js + Material UI Sample
          </Typography>
        </Toolbar>
      </AppBar>
      <Typography variant="h1" gutterBottom={true}>
        Material-UI
      </Typography>
      <Typography variant="subtitle1" gutterBottom={true}>
        example project
      </Typography>
      <Typography gutterBottom={true}>
        <Link href="/about">

```

(次のページに続く)

(前のページからの続き)

```
    <a>Go to the about page</a>
  </Link>
</Typography>
<Button
  variant="contained"
  color="secondary"
  onClick={() => { setDialogOpen(true)}}
>Shot Dialog</Button>
<style jsx={true}>{`
  .root {
    text-align: center;
  }
`}</style>
</div>
);
}
```

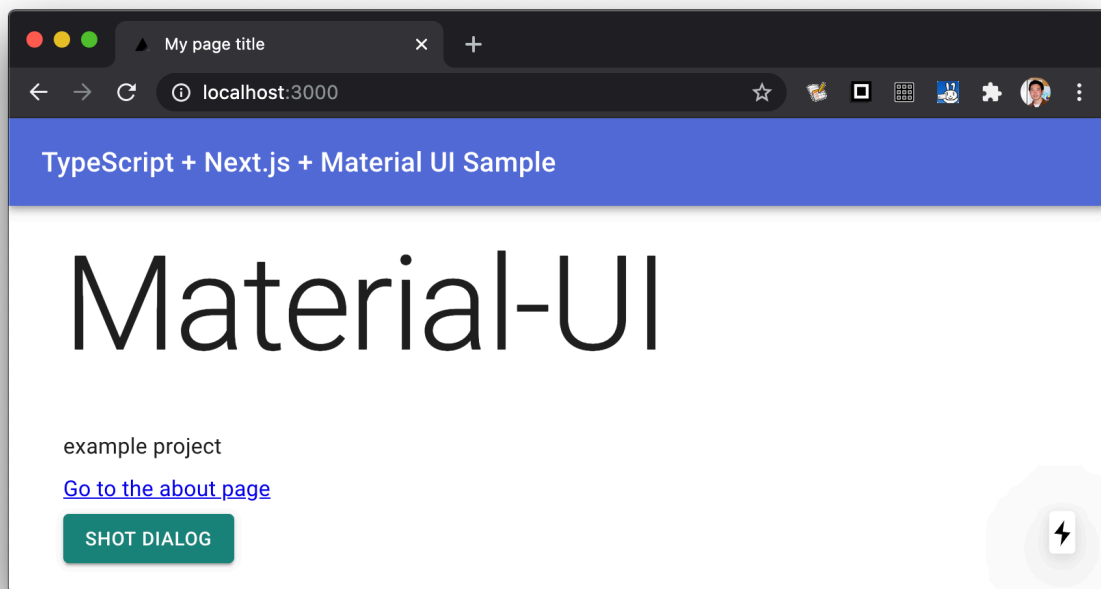


図 1 Next.js + Material UI + TypeScript のサンプル

29.6 React と TypeScript

React に限らず、近年の流行のウェブフロントエンドの実装スタイルは「コンポーネント」と呼ばれる独自タグを実装していく方法です。コンポーネントの中にもコンポーネントを書くことができます。そのコンポーネントが集まってアプリケーションになります。React もその例に漏れず、コンポーネントを実装していきます。

前節のサンプルの中にある、大文字始まりの名前のタグがそのコンポーネントです。それぞれのタグは表示されるときには分解されて、最終的には HTML5 のいつものタグに還元されます。タグなので、引数があり、子要素があります。

React は TypeScript を使って適切に型のチェックができるようになっています。React は jsx 構文を使って書きますが、これは TypeScript や Babel といった処理系によって JavaScript の普通の関数呼び出しに変換されます。React 以前のライブラリなどは、テンプレートをフロントで効率よく実行するための、動的に関数呼び出しのコードを生成し、eval などを使って関数に変換したりしていました。近年ではこれらは Content-Security-Policy でエラーになる可能性があるなどの問題もあり、ビルド時にプログラム化するようになってきました。

React はこのテンプレートの変換を処理系が直々に行うので、別途変換のプリプロセッサを入れる必要がないというメリットもあるのですが、それ以上に入力パラメータの間違いなどを、普通の関数の型チェックと同様に行えるという、他のフロントエンドのフレームワークにはないメリットがあります。このチェックを最大限に生かすのも、それほど手間をかけずに行えます。次のコードは TypeScript を用いて React コンポーネントを作るときによく使う要素を詰め込んだものです。

- 外部からの引数 (props)
- コンポーネント内で管理するステート (useState())
- 初期化コード、終了コード (useEffect())
- デフォルト値

```
import React, { useState, useEffect } from 'react'

// コンポーネントのプロパティ (タグの属性)
type Props = {
  title: string;
  description?: string;
  defaultValue: string;
};

// コンポーネントは関数
// 返り値が最終的に描画される HTML タグ
export function MyComponent(props: Props) {
  // props を参照
  const { title, description } = props;
  // コンポーネント内のステート
  const [count, setCount] = useState(0);
  // ライフサイクルメソッド
```

(次のページに続く)

(前のページからの続き)

```

useEffect(() => {
  // 作成時（初回レンダリング直後）に呼ばれる箇所
  return () => {
    // 終了時に呼ばれる箇所
  };
}, []);

return (
  <div>
    <h1>{title}</div>
    { description ? <section>{description}</section> : null }
    <button onClick={() => setCount(count + 1)}>{count}</button>
  </div>
);
};

// 省略時のデフォルトのプロパティ
MyComponent.defaultProps = {
  defaultValue: 'default';
};

```

これらのうち、引数の関数のプロパティにきちんと型をつけ、defaultProps に値を設定すれば、利用時にエラーチェックが行われるようになります。また、関数内部の useState() は初期値に設定した値を元に型推論が行われます。2つの要素のタプルを返しますが、前者は初期値と同じ型の即値が入った変数、後者はステートを更新するための関数（初期値と同じ型の1の引数のみを持つ）です。これらのおかげで、自分のコンポーネントを作成するときも、その作成したコンポーネントを利用するときも、型の恩恵が受けられます。

コンポーネントを外部公開する場合に、Props を export する必要はありません。コンポーネントから ComponentProps<> を使って導出が可能です。もし継承拡張するニーズがあったとしても、コンポーネントだけ export しておけば利用側でアクセスできます。余計なものを export しない方がプログラムの依存関係はよりシンプルになります。

```

import React, { ComponentProps } from 'react';
import { MyComponent } from './my-component';

type MyComponentProps = ComponentProps<typeof MyComponent>;

```

ただし、defaultProps で初期値を与えていてもそれは反映されません。そのため、利用側の印象と一致させるために、defaultProps を与える場合は、Props 定義に?を追加しておきましょう。

```

type Props = {
  title: string;
  description?: string;
  defaultValue?: string; // こうしておくべきだった
};

```

29.7 Redux と TypeScript

React の周辺のライブラリの作者の中には TypeScript を使わない人が多くいました。React は元々 Facebook が開発していた flowtype を使うことが多かったり、React の JavaScript の記述法がかなりトリッキーだったり、と理由はいろいろ考えられます。その後、Microsoft が React を大々的に利用するように宣言し、TypeScript の機能もかなり充実しました。近年では TypeScript の型定義ファイルが最初から付属するようになったり、TypeScript で再実装されたり、TypeScript との親和性がどんどん上がっています。

Redux はアプリケーション内部で横断的に利用したいデータを保持したり、その更新を行うための補助ライブラリです。データの更新にともない、必要な画面更新だけを効率的に行えるようにもしてくれます。大規模なアプリケーションではよく活用されていました。その Redux 本体も、TypeScript で適切に型をつけていこうとするとかなり頭と手を使う必要がありましたが、公式サポートライブラリの Redux-Toolkit は、TypeScript との親和性が極めて高くなりました。素の Redux をこれから扱う理由は特にないので、本書では Redux-Toolkit 経由での Redux の操作について紹介します。

Redux はストアと呼ばれる中央のデータ庫を持ちます。データに変更を加えるための reducer と呼ばれる変換ロジックを実装します。標準の Redux を使う場合は、reducer のみを実装します。この関数の戻り値がステートになります。データの保管そのものは Redux が行い、開発者が触ることはできません。必要に応じて reducer を Redux が実行し、その結果を Redux が管理するという構成です。

この reducer をトリガーするのに必要なのが、アクションと呼ばれるデータでした。これを `dispatch()` という関数に投げ込むことで reducer が起動され、そのアクションに応じてデータを書き換えていました。

しかし、まず JavaScript の文化で、アクションクリエイターというアクションを作る関数を作っていました。この場合、型をつけるには reducer の引数にはすべてのアクションの型（アクションクリエイターの戻り値の型）の合併型を作る必要がありました。この「すべての」というのが大きなアプリケーションになると依存関係が循環しないように気をつけたり、漏れなく型を合成してあげないといけなかったりと、型のために人間が行う作業が膨大でした。多くの人が「Redux に型をつけるには？」という文章を書いたりしましたが、その後、Redux が公式で出してきた解答が Redux-Toolkit でした。

Redux-Toolkit は次のような実装になります。スライスというステートと reducer、アクションクリエイターがセットになったオブジェクトを作成します。Reducer の引数の state は `Readonly<>` をつけておくと、デバッグで問題の追跡が難しい不測の事態が発生するのを未然に防げます。

リスト 5 スライスを作成

```
import { createSlice, configureStore, PayloadAction } from '@reduxjs/toolkit';

// state の型定義
export type State = {
  count: number;
};

// 初期状態。インラインで書いても良いですが・・・
const initialState: State = {
```

(次のページに続く)

(前のページからの続き)

```

    count: 0
  };

  // createSlice で reducer と action を同時に定義
  const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
      incrementCounter: (state: Readonly<State>, action: PayloadAction<number>) => ({
        ...state,
        count: state.count + action.payload,
      }),
      decrementCounter: (state: Readonly<State>, action: PayloadAction<number>) => ({
        ...state,
        count: state.count - action.payload,
      }),
    },
  });

```

スライス自体は Redux のストアを作る材料ではありますが、もうひとつ、アクションクリエーターのオブジェクトも結果に格納されています。これをエクスポートしてコンポーネントから利用できるようにします。

```

// action creator をスライスから取り出して公開可能
// dispatch 経由でコンポーネントのコードから呼び出せる
export const { incrementCounter, decrementCounter } = counterSlice.actions;

```

スライスからストアを作るには `configureStore()` を使います。管理対象が少なければ、スライス作成からストア作成まで1ファイルでやりきってもいいでしょう。複雑になる場合は、スライス作成部分をファイルに切り出しましょう。

リスト 6 スライスからストアを作成

```

// slice から store を作る
export const store = configureStore({
  reducer: counterSlice.reducer,
});

// 複数の slice から store を作るには reducer にオブジェクトを渡せば OK
export const store = configureStore({
  reducer: {
    counter: counterSlice.reducer,
    primenumber: primenumberSlice.reducer,
  }
});

```

さらに型チェックを強固にするために、コンポーネントとのインターフェースとなる関数群にもきちんと型をつけておきます。上記の `store` を作るファイルと一緒にやってしまうと良いでしょう。

リスト 7 利用側との接点となる型情報付き関数を生成

```
import {
  useSelector as useReduxSelector,
  TypedUseSelectorHook,
} from 'react-redux';

export type RootState = ReturnType<typeof store.getState>;
export const useSelector: TypedUseSelectorHook<RootState> = useReduxSelector;
export type AppDispatch = typeof store.dispatch;
```

Redux との大きな違いは、内部で管理するステートの初期値とその型を明示的に宣言できるようになったことです。Redux では reducer の引数とそのデフォルト値が初期値でした。いろいろなところで活用しますし、ステートの加工にあたってチェックや補完が欲しいところなので、補完も期待通りに行われますし、エラーメッセージもわかりやすくなります。

reducers の中身が実際に値を加工する操作が入っています。この関数では変更前のステートを受け取り、それに値を設定して関数の返り値として返します。Redux と違い、1 つの関数の中に自分で switch 文を書くのではなく、このオブジェクトのキー単位で操作の単位として独立しています。

アプリケーション側との接点は 2 か所です。アプリケーション全体の設定と、値を利用したいコンポーネントです。

リスト 8 アプリ全体で一カ所、store を設定

```
import { store } from '../redux/store';
import { Provider } from 'react-redux';

function App() {
  return (
    <Provider store={store}>
      <Router>
        <Switch>
          <Route exact path="/"><RootPage /></Route>
          <Route path="/edit"><EditPage /></Route>
        </Switch>
      </Router>
    </Provider>
  )
}
```

各コンポーネントでは `useSelector()` と `useDispatch()` を使ってストアへの読み書きを行います。

リスト 9 Redux のステートを利用する関数側

```
// Redux の提供の useDispatch
import { useDispatch } from 'react-redux';
// スライス側からアクションクリエーター
import { incrementCounter } from '../redux/counterslice';
// ストア側からは型をつけた useSelector と Dispatch 用の型定義
import { useSelector, AppDispatch } from '../redux/store';

export function MyComponent() {
  const dispatch = useDispatch<AppDispatch>();
  const counter = useSelector(state => state.counter);
  return (
    <div>
      <!-- ストアのステートを利用 -->
      <h1>count: {counter.count}</h1>
      <!-- dispatch でストアに変更を加える -->
      <button onClick={() => dispatch(incrementCounter(10))} />
    </div>
  )
}
```

要注意なポイントは、スライスの名前です。複数のスライスをまとめて Redux の最終的なステートを作り上げますが、この名前がかぶっていると、変更していないはずなのにいつのまにか値が変更されていたりといったトラブルが発生します。

29.8 React と Redux の非同期アクセス

React の基本の書き方と Redux-Toolkit を使って型チェックが完全な形で行われるようになりました。サーバーサイドレンダリングの仕組みや styled-component を使ったスタイリング、Router によるシングルページアプリケーションのページ切り替えなど、追加の情報や便利ライブラリは別にありますが、React に関する最新の基礎知識はほぼこれでカバーできると言えます。

しかし、もう 1 つ触れておかなければならないことがあります。それが非同期のデータアクセスです。

画面の表示に必要なデータの取得や結果の格納でサーバーアクセスが必要になることがあります。サーバーアクセスが一切ないウェブフロントエンドはあまりないでしょう。静的サイトジェネレータから呼び出す場合はまたそちらの作法がありますが、今回は通常のウェブアプリケーションのフロントエンドの説明を行います。

まず表示に利用する情報の取得です。コンポーネント単体で取得、あるいは Redux 経由の利用があります。一番簡単なコンポーネント内部で完結する方法を紹介します。コンポーネント内部で呼び出す場合は `useEffect()` を利用します。注意点としては、`useEffect()` には `async` 関数を渡すことができない点です。後始末の処理を `return` で返すという API 設計の制約による気がしますが、利用側としては従わざるをえません。`async` な関数を作り、それを呼び出します。

このコードは、ブラウザ標準 API の `fetch` を使い、最終的に `useState()` 提供の関数 `setData()` で取得してきた値を格納しています。もしエラーがあれば、同様に `setShowErrorDialog()` に格納しています。

```
// サーバーデータ
const [data, setData] = useState({loaded: false} as Data);
// エラーダイアログ表示用ステート
const [showErrorDialog, setShowErrorDialog] = useState('');

useEffect(() => {
  async function getData() {
    const res = await fetch('/api/getdata');
    let data: Data;
    if (res.ok) {
      try {
        data = await res.json() as Data;
      } catch (e) {
        setShowErrorDialog(`parse error ${e}`);
        return;
      }
    } else {
      setShowErrorDialog(`server access error`);
      return;
    }
    setData(data);
  }
  getData();
}, []);
```

Redux-Toolkit の reducers にはそのままでは非同期処理が書けません。 `createAsyncThunk()` を使い、それを `extraReducers` の中で登録します。

リスト 10 非同期の reducer

```
import { createAsyncThunk } import '@reduxjs/toolkit';

type fetchLastCounterReturnType = {
  count: number;
};

export const fetchLastCounter = createAsyncThunk<fetchLastCounterReturnType>(
  'lastcount/fetch',
  async (arg, thunk): Promise<fetchLastCounterReturnType> => {
    const res = await fetch('/api/lastcount', {
      credentials: 'same-origin',
    });
    if (res.ok) {
      return (await res.json()) as fetchLastCounterReturnType;
    }
    throw new Error('fetch count error');
  }
);
```

(次のページに続く)

(前のページからの続き)

```

    }
  );

  const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {},
    extraReducers: builder => {
      builder.addCase(fetchLastCounter.fullfilled,
        (state, action) => {
          return {
            ...state,
            count: action.payload.count
          };
        }
      );
    }
  });
}
})

```

これも、`dispatch(fetchLastCounter())` のように呼び出せます。この非同期アクションに引数を設定したい場合は、`createAsyncThunk` の 2 つ目の型パラメータに引数を設定します。型パラメータに入れずに `async` の関数側の `arg` にだけ型を付けようとしてもエラーになるので注意してください。

```

type fetchLastCounterArgType = {
  counterName: string;
};

const fetchLastCounter = createAsyncThunk<
  fetchLastCounterReturnType,
  fetchLastCounterArgType
> (
  'lastcount/fetch',
  async (arg, thunk) {
    // 略
  }
);

```

この非同期アクションから `Redux` のストアに値を設定する方法が 2 つあります。1 つが上記の登録方法で紹介した `extraReducers` です。pending、fulfilled、error の 3 つの状態に対して reducer が書けます。それぞれ、実行開始直後、完了後、エラー発生の際に呼ばれます。これが一番簡単です。

もう片方が、データ格納用の reducer を別個に作成し、非同期アクションから呼び出す方法です。2 つ目の引数の `thunk` には `getState()` や `dispatch()` といった、`Redux` 本体とアクセスするメソッドがあります。これらを使い、ステートの状態を取得しつつ、`dispatch()` で個別に作成した reducer に呼ぶことで、ステートに結果を書き込むことができます。基本的には前者の方法で済むことが多いでしょう。

これらの非同期アクションを呼び出して結果をコンポーネントから利用する方法は 2 つあります。ひとつはすでに

紹介した `useSelector()` 経由で情報を取得してくる方法です。もう 1 つは、結果を直接受け取る方法です。後者は `dispatch()` の結果を `unwrapResult()` に渡すことで、正常終了したときの結果が得られます。

リスト 11 `unwrapResult` を使った例

```
import { unwrapResult } import '@reduxjs/toolkit';

useEffect(() => {
  async function getData() {
    const ret = unwrapResult(await dispatch(fetchLastCounter()));
    dispatch(fetchUpdateLog(ret.count, username));
  }
  getData();
}, []);
```

React の昔からよく発生するコーディングのミスとして、ステートへ格納した直後に結果を読み出そうとしてもまだ更新されていない、というものがあります。`useState()` のセッターで設定したステートや `Redux` のストアの状態は、次の更新時まで変更されません。更新してしまうと、一つのレンダリング関数の中で、変数の状態が複数存在する可能性が発生してしまい、整合性を保つのが困難になります。しかし、その副作用として、結果を更新したものを扱う場合に、次の更新まで待たなければならなくなります。

しかし、非同期の呼び出しがきちんと期待通りに呼ばれるかどうかというのはうまく動かなかった時の問題追跡が困難です。そのため、サーバーの結果を受けて再度何かサーバーアクセスを行う場合などは、途中で `React` の再描画を待つのではなく、一つの `async` 関数の中で処理を完結させる方がバグが出にくく、コードの行数も短くなり、見通しの良いコードになります。

次のコードはぱっと見たときに `useEffect()` 同士の依存関係が見えません。コードを読み解くと、`fetchLastCounter()` の結果が `Redux` のステートに格納され、その数値が書き換わったことで、2 つ目の `useEffect()` が呼ばれることが分かりますが、お世辞にも分かりやすいとは言えません。`TypeScript` の可視性のすぐれた `async/await` を使うべきです。

リスト 12 unwrapResult を使わない例

```
const counter = useSelector(state => state.counter);

useEffect(() => {
  dispatch(fetchLastCounter());
}, []);

// 分割された useEffect
useEffect(() => {
  dispatch(fetchUpdateLog(counter.count, username));
}, [counter.count]);
```

29.9 React の新しい書き方

React は歴史のあるコンポーネントで、途中でいくつも機能追加が行われたり改善されています。1つのことを実現するのに新旧何通りもやり方が提供されていたりします。新しい書き方が作られるのは、もちろん、そちらの方がミスが少なかったり、コードが短くなったりと改善が見込まれるからです。React の場合は TypeScript 的にも優しい書き方となっているため、もし古いコーディング規約に従っている場合は新しい書き方に整理していくと良いでしょう。

2019 年 2 月にリリースされた React 16.8 の Hooks により、新しい書き方に大々的に移行可能になりました。もしこれ以前から続いているプロジェクトの場合、新しくつくるコンポーネントや、改修を行うコンポーネントから徐々に移行していくと良いでしょう。

29.9.1 クラスコンポーネントではなく、関数コンポーネントにする

まずは古い TypeScript 以前の書き方です。お決まりの書き方だけでもかなりの行数になってしまいます。

リスト 13 古い書き方

```
import React, { Component } from "react";
import propTypes from "prop-types";

class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0;
    }
    this.onClick = this.onClick.bind(this);
  }
}
```

(次のページに続く)

(前のページからの続き)

```

componentDidMount() {
  // サーバーアクセスなどのマウント後に実行したいコードはここ
}

componentWillUnmount() {
  // 削除前に実行したいコードはここ
}

onClick() {
  this.setState({
    count: this.state.count + 1;
  });
  this.props.onUpdated(this.state.count + 1);
}

render() {
  return (
    <div className="panel">
      <div className="message">
        <button onClick={this.onClick}>{this.props.label}</button>
      </div>
    </div>
  );
}
}

MyComponent.propTypes = {
  label: PropTypes.string
  onUpdated: PropTypes.func
}

MyComponent.defaultProps = {
  label: "押して下さい"
};

```

これ以降、JavaScript や TypeScript への機能追加により、何段階か書き方の改善がありました。コンストラクタで `onClick` を `bind` しなおさずに、`class` 定義の中で代入できるようになったので、アロー演算子を使っているかもしれませんし、TypeScript 化で `Component` の型変数で `Props` や `State` の型変数を設定するようになっているかもしれません。

リスト 14 ほどほどに古い書き方

```

interface Props {
  label: string;
  onUpdated: (count: number) => void;
}

```

(次のページに続く)

(前のページからの続き)

```

interface State {
  counter: number;
}

// TypeScript で型定義
export class MyComponent extends Component<Props, State> {
  // コンストラクタではなく、クラス定義の中で代入文
  private state: State = {
    counter: 0,
  };
  // アロー演算子でイベントハンドラ実装
  private onClick = () => {
    this.setState({counter: this.state.counter + 1});
  }
  render() {
    // :ここは同じ
  }
}

```

古い書き方で TypeScript を使わなくても、React レベルでさまざまなチェック機構が提供されており開発は便利ではありました。ただし、state の変更処理（`setState()` 呼び出し）をした直後にはまだインスタンス変数の `this.state` が変更されておらず、状態がおかしくなってしまう、という問題があったり、イベントハンドラを JSX に渡すときに、`this` の束縛を忘れてイベントが発火した後にエラーになるといったミスがおきやすい素地がありました。

現在主流になっているのが関数コンポーネントです。当初は状態を持たないコンポーネントのみだったため、クラスコンポーネントからの完全移行は大変でしたが、Hook という機能が追加されてクラスコンポーネントを完璧に置き換えられるようになりました。関数コンポーネントは状態管理を React 側におまかせして、`render()` のみにしたような書き方です。だいた、縦にも横にも圧縮されたことがわかります。

リスト 15 新しい書き方

```

import React, { useState, useEffect } from "react";

type Props = {
  label?: string;
  onUpdated: (count: number) => void;
};

export function MyComponent(props: Props) {
  const [count, setCount] = useState(0);
  const {label, onUpdate} = props;

  useEffect(() => {
    // サーバアクセスなどのマウント後に実行したいコードはここ
    return () => {
      // 削除前に実行したいコードはここ
    }
  });
}

```

(次のページに続く)

(前のページからの続き)

```

    }
  }, []);

  function onClick() {
    setCount(count + 1);
    onUpdated(count + 1);
  }

  return (
    <div className="panel">
      <div className="message">
        <button onClick={onClick}>{label}</button>
      </div>
    </div>
  );
}

MyComponent.defaultProps = {
  label: "押して下さい"
};

```

一番短くなってミスがおきにくくなったのは state 周りです。useState() に初期値を渡すと、現在の値を保持する変数と、変更する関数がペアで帰ってきます。初期値から型推論で設定されるため、State の型定義を外で行う必要はなくなります。setState() で変更したものが直後に変更されているはず、と誤解されることもなくなりました。もう一度レンダリングが実行されないと変数の値が変更されないのは useState() の宣言を見ればあきらかです。

イベントハンドラの this の束縛もなくなります。もはや単なる関数であって、オブジェクトではないため、this を扱う必要もなくなります。横方向に圧縮されたのは this. がたくさん省略されたからです。

いくつかのライフサイクルメソッドが削除されたり、名前が変わったりはありますが、以前のコードもそのまま動きますので、全部を一度に移行する必要はありません。

29.9.2 サードパーティのライブラリも Hooks を使う

関数コンポーネント自体もコードを短くする効能がありますが、新しい Hooks スタイルにより、サードパーティのライブラリの組み込みも簡単になります。残念ながら、Hooks スタイルの関数は関数コンポーネントでしか利用できませんので、前述の関数コンポーネントへの書き換えがまず必要になります。

例えば、React-Router や Redux との接続は、コンポーネントをラップして props に要素を追加する関数呼び出しが必要でした。ユーザーコード側では、サードパーティのライブラリから何かしら情報をもらったり、サードパーティのライブラリの機能呼び出しするには、props 経由で扱う必要があります、この特殊なラッパーは props に新しい属性を増やす役割を果たしていました。しかし、ユーザーコード側でも propTypes にこれを追加する必要があったりと、たくさんの転記作業が必要でした。コンポーネントの外の状態まで気を配る必要がありました。

リスト 16 古い React-Router のラッパースタイルの書き方

```
import React, { Component } from "react";
import { withRouter } from "react-router-dom";

export class MyComponent extends Component {
  onClick() {
    // ページ遷移
    this.props.history.push("/new-path");
  }
  render() {
    // :
  }
}

// サードパーティを使う側に知識が必要なポイント
MyComponent.propTypes = {
  history: PropTypes.object.isRequired,
};

// ここでラップ!
const MyComponent = withRouter(MyComponent);
```

Hooks に対応した React-Router の v5 移行であればコンポーネントの中で履歴を触るためにコンポーネントの外にまで手を加える（ラップしたり Props を変更する）必要はなくなりました。ここでも、縦にも横にも短くなったことがわかるでしょう。React-Router の機能にアクセスするための壮大な準備コードが不要になりました。

リスト 17 新しい履歴へのアクセス方法

```
import React from "react";
import { useHistory } from "react-router-dom";

export function MyComponent( {
  const history = useHistory();

  onClick() {
    history.push("/new-path");
  }

  return (
    // :
  );
}
```

よく不要論が取り沙汰される Redux も、Redux のストアにアクセスしたり、変更のために dispatch を呼ぶときにその準備コードが多くなる問題がありました。次のコードは、コンポーネント定義自体は全部省略して空ですが、これだけの準備コードが必要でした。

リスト 18 Redux の古い書き方

```
import React, { Component } from "react";
import { connect } from "react-redux"

class MyComponent extends Component {
  ...
}

// PropTypes への追加が必要
MyComponent.propTypes = {
  counter: PropTypes.object,
  onClick: PropTypes.func,
  dispatch: PropTypes.func,
}

// このマッピング関数の定義は必要
function mapStateToProps(state, props) {
  return {
    counter: state.reducer.counter
  };
}

// connect で props に dispatch が増えるので、connect の 2 つめの
// このマッピングは使わずに dispatch をコンポーネント内部で呼び出す
// ことも可能
function mapDispatchToProps(dispatch) {
  return {
    // アクションはオブジェクトそのままではなくアクションクリエータとして切り出されている場合も
    onClick: () => dispatch({
      type: Actions.DISPATCH_EVENT,
      hoge: true,
    }),
  }
};

const MyComponent = connect(mapStateToProps, mapDispatchToProps)(Test);
export MyComponent;
```

`dispatch()` のマッピングはしていませんが、`dispatch()` や `Redux` のストアへのアクセスは 2 つの `Hooks` スタイルの関数で完了します。劇的ですな。Redux のストア定義自体も、本章の中で紹介した `Redux-Toolkit` を使うことで大幅に短く書けるようになりました。

リスト 19 Redux の新しい書き方

```
import { useDispatch, useSelector } from 'react-redux';

export function MyComponent() {
```

(次のページに続く)

(前のページからの続き)

```
const dispatch = useDispatch();
const counter = useSelector(state => state.counter);
}
```

なお、`useDispatch()` と `useSelector()` ですが、本章の中で触れた通りに、`Redux-Toolkit` のストアの定義のついでに型付けをしておくと、コンポーネント内部でも型の恩恵を最大限に得ることができます。

`React-Router` にしても、`Redux` にしても、はたまたスタイル定義のライブラリだったりにしても、一種類だけの適用であれば、探せばサンプルコードや情報も出てきますし、初心者でも調べ物しながらなんとかできる範囲ではありますが、複数のコンポーネントが登場し始めて設定周りのコードが絡みだすと、情報が減り、トラブル発生時のシューティングが難しくなります。コードを読む人も、どこから手を付けて良いのか分かりにくくなってきます。

同じ機能を実装するにしても、コードが縦にも横にも短く、儀式的なコードが減れば、ライブラリや技術へのキャッチアップコストも減りますし、読んで理解するのも簡単になります。また、型の恩恵も受けやすになると、開発がかなり加速するでしょう。

29.10 まとめ

これで一通り、`React` を使う環境ができました。最低限の設定ですが、`TypeScript` を使ったビルドや、開発サーバーの起動などもできるようになりました。

フロントエンドは開発環境を整えるのが大変、すぐ変わる、みたいなことがよく言われますが、ここ 10 年の間、やりたいこと自体は変わっていません。1 ファイルでの開発は大変なので複数ファイルに分けて、デプロイ用にはバンドルして 1 ファイルにまとめる。ブラウザにロードしてデバッグする以前にコード解析で問題をなるべく見つけるようにする。ここ 5 年ぐらいは主要なコンポーネントもだいたい固定されてきているように思います。`State of JavaScript Survey` という調査をみると、シェアが高いライブラリはますますシェアを高めていっており、変化は少なくなっています。一方で、`React` 自体はより良い書き方ができるように進歩しています。

- <https://2019.stateofjs.com/>

`CoffeeScript` や `6to5` に始まり、`Babel`、`TypeScript` と、`AltJS` もいろいろ登場してきましたが、`TypeScript` の人気は現在伸び率がナンバーワンです。それに応じて、各種環境構築ツールも `TypeScript` をオプションの一つに加えており、ドキュメントでも必ず言及があります。デフォルトで `TypeScript` が利用できるというツールも増えてきています。

本章の内容も、最初にしたときよりも、どんどんコンパクトになってきています。もしかしたら、将来みなさんが環境構築をする時になったら本書の内容のほとんどの工程は不要になっているかもしれません。それはそれで望ましいので、早くそのような時代がきて、お詫びと訂正をしたいと思います。

第 30 章

Vue.js の環境構築

Vue.js は日本で人気のあるウェブフロントエンドのフレームワークです。柔軟な設定のできる CLI ツールが特徴です。本書では 3 系についてとりあげます。

```
$ npx @vue/cli@next create myapp
```

作成時に最初に聞かれる質問で default の preset (babel と eslint) ではなく、Manually select features を選択します。

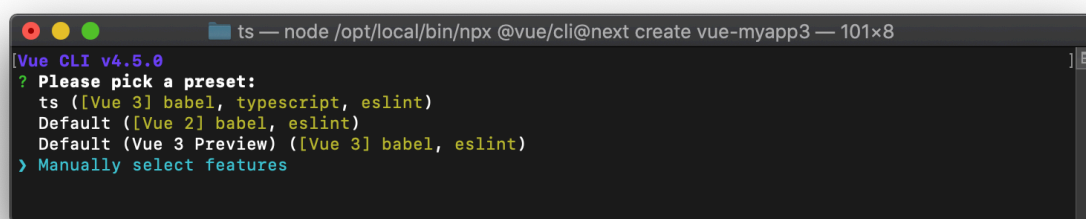
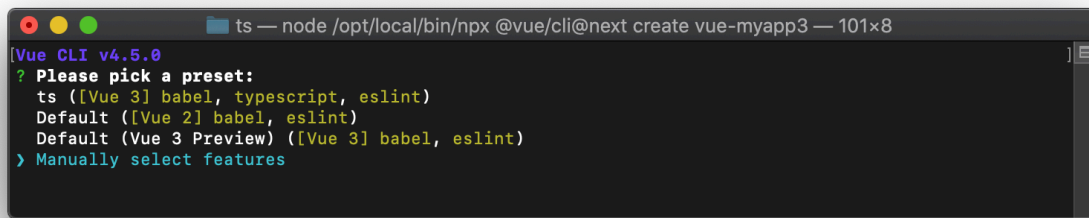


図 1 TypeScript を選択する場合は Manually select features を選択

次のオプションで必要なものをスペースキーで選択して、エンターで次に進みます。選んだ項目によって追加の質問が行われます。Router やステート管理などのアプリケーション側の機能に関する項目以外にも、Linter やユニットテストフレームワークや E2E テストの補助ツールなど、さまざまなものを選択できます。

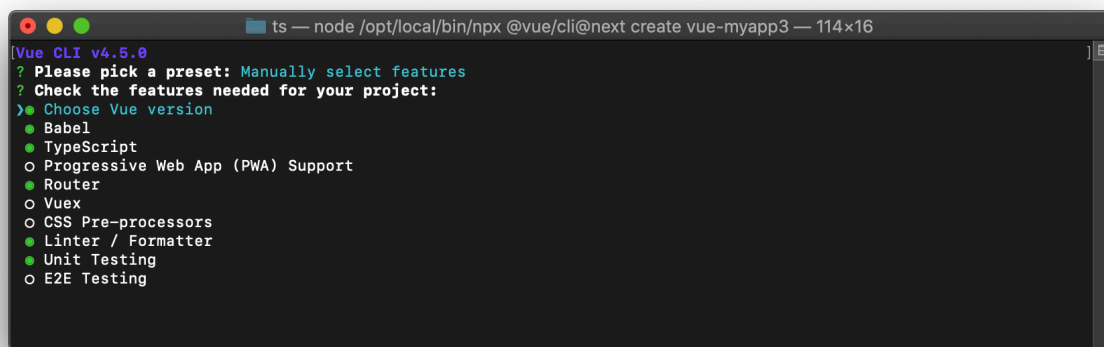
途中で、クラスベースかそうではないか、という質問が出てきます。以前ではクラスベースの API の方が TypeScript との相性がよかったのですが、Vue.js3 からの新しい API で、クラスベースでない時も型チェックなどに優しい Composition API が追加されました。そこはチームで好きな方を選択すれば良いですし、あとから別のスタイルにすることもできます。

現在の Vue.js のプロジェクトのほとんどは、.vue ファイルに記述するシングルファイルコンポーネント (SFC) を使っていると思いますが、TypeScript を使う場合、スクリプトタグの lang 属性を ts になっています。



```
ts — node /opt/local/bin/npm @vue/cli@next create vue-myapp3 — 101x8
[Vue CLI v4.5.0]
? Please pick a preset:
  ts ([Vue 3] babel, typescript, eslint)
  Default ([Vue 2] babel, eslint)
  Default (Vue 3 Preview) ([Vue 3] babel, eslint)
> Manually select features
```

図 2 TypeScript を選択する場合は Manually select features を選択



```
ts — node /opt/local/bin/npm @vue/cli@next create vue-myapp3 — 114x16
[Vue CLI v4.5.0]
? Please pick a preset: Manually select features
? Check the features needed for your project:
> Choose Vue version
  Babel
  TypeScript
  Progressive Web App (PWA) Support
  Router
  Vuex
  CSS Pre-processors
  Linter / Formatter
  Unit Testing
  E2E Testing
```

図 3 必要な機能を選択する

```
<template>
  HTML テンプレート
</template>
<script lang="ts">
  コンポーネントのソースコード (TypeScript)
</script>
<style>
  CSS
</style>
```

30.1 クラスベースのコンポーネント

クラスベースのコンポーネントは `vue-class-component` パッケージを使い、デコレータをつけたクラスとして実装します。クラスのインスタンスのフィールドがデータ、ゲッターが算出プロパティになっているなど、TypeScript のプレーンな文法と Vue の機能がリンクしており、ウェブフロントエンドを最初に学んだのではなく、言語としての TypeScript や JavaScript、他の言語の経験が豊富な人には親しみやすいでしょう。環境構築の CLI のオプションではデフォルトでこちらになるような選択肢になっています。

以下のコードは Vue.js 3 系のクラスベースのコンポーネント実装です。

```
<script lang="ts">
import { Options, Vue } from "vue-class-component";
import HelloWorld from "../components/HelloWorld.vue";

@Options({
  components: { // テンプレート中で利用したい外部のコンポーネント
    HelloWorld
  }
})
export default class Counter extends Vue {
  // フィールドがデータ
  count = 0;

  // 算出プロパティはゲッターとして実装
  get message() {
    return `カウントは${this.count}です`;
  }

  // メソッドを作成するとテンプレートから呼び出せる
  increment() {
    this.count++;
  }

  decrement() {
    this.count--;
  }

  // ライフサイクルメソッドもメソッド定義で OK
  beforeMount() {
  }
}
</script>
```

これをラップしてより多くのデコレータを追加した `vue-property-decorator` というパッケージもあります。こちらの方が、`@Prop` や `@Emit` でプロパティやイベント送信も宣言できて便利でしょう。

- <https://www.npmjs.com/package/vue-property-decorator>

警告: ただし、現時点で 3.0 系で変わった `vue-class-component` の変更にはまだ追従していないように見えます。

30.2 関数ベースのコンポーネント作成

Vue 本体で提供されている `defineComponent()` 関数を使いコンポーネントを定義します。今までのオブジェクトをそのまま公開する方法と違い、この関数の引数のオブジェクトの型は定まっているため、以前よりも TypeScript との相性が改善されています。このオブジェクトの属性で名前や他の依存コンポーネント、`Props`などを定義するとともに、`setup()` メソッドでコンポーネント内部で利用される属性などを定義します。

```
<script lang="ts">
import { defineComponent, SetupContext, reactive } from "vue";
import HelloWorld from "../components/HelloWorld.vue";

type Props = {
  name: string;
}

export default defineComponent({
  name: "App",
  components: {
    HelloWorld
  },
  props: {
    name: {
      type: String,
      default: "hello world"
    }
  },
  setup(props: Props, context: SetupContext) {
    const state = reactive({
      counter: 0,
    });
    const greeting = () => {
      context.emit("greeting", `Hello ${props.name}`);
    };

    return {
      state,
      greeting
    }
  }
});
</script>
```

注釈: Nuxt.js を使ったプロジェクト作成

Vue.js にも、Vue.js をベースにしてサーバーサイドレンダリングなどの自分で設定すると大変な機能がプリセットになっている Nuxt.js があります。Nuxt.js の場合は、通常の設定の後に、いくつか追加のパッケージのインストールや設定が必要です。日本語によるガイドもあります。

- <https://typescript.nuxtjs.org/ja/guide/setup.html>

ただし、現時点では Vue.js 3 対応はまだ計画中でリリースはまだ行われていません。

30.3 Vue.js を使った jQuery のリプレース

jQuery は歴史があるライブラリで、使い勝手の良さから、非フロントエンド開発者にも広く普及しました。一方で、開発が大規模化する場合に整合性をとるのが難しくなってくることが多く、フロントエンドの比重が高まるにつれて、React や Vue.js を使う人が増えています。

jQuery から Vue.js へはパラダイムがかなり違うので、多少コーディングが必要となります。jQuery は、セレクトタでマッチした HTML のタグを直接変更していきます。一方、最近のウェブフロントエンドのフレームワークは TypeScript 内部に状態を持ち、それを画面に反映させる、という形をとります。反映するときはテンプレートエンジンのような記法を用いて表現します。Vue や React は仮想 DOM という仕組みを使っており、ビュー関数を頻繁に実行し、その結果を画面に反映します。

jQuery の方が、極めて簡単なことをする場合に短いコードで済むことがあります。一方、変更が多くなると更新が複雑になります。

- 同じ値を何度も表示する場合、Vue や React の場合、大元の変数を変更するとすべての箇所が変わります。jQuery では利用箇所をすべて自分で見つけて更新しなければなりません。
- テーブル表示など、表示先の階層が深くて場所の特定も大変な場合にはロジックが複雑になります。
- 確認ダイアログを出して OK/Cancel 時に別のダイアログを出してという場合には、次のダイアログを表示にする、前のダイアログを非表示にする、といったように、すべての変更を 1 つずつ適用していかなければなりません。状態遷移が複雑になると、一箇所の修正漏れで画面の遷移がおかしくなります。Vue や React であれば、現在の遷移はどこか、という情報を 1 つもち、それをみるようにすると、扱う状態が少ない分、ミスが減ります。

次のようなシンプルな jQuery のコードを Vue.js にしてみましょう。

```
<div>jQuery test page</div>
<button class="pushbutton">button</button>
<div class="panel" style="display: none; background-color: lightblue;"
  hidden
</div>
```

(次のページに続く)

```
<style scoped>
  .clickedButton {
    background-color: "yellow";
  }
</style>

<script>
$(function() {
  $(".pushbutton").click(function() {
    $(".pushbutton").addClass("clickedButton");
    $(".panel").fadeIn();
  });
});
</script>
```

ここで使っている jQuery の機能は 3 つです。

- `click()` でボタンのクリックのイベントハンドラの設定
- クリック時に `addClass()` で CSS のスタイルの変更
- クリック時に、`fadeIn()` を使って隠された要素の表示

30.3.1 まずはそのまま.vue ファイル化

まずはプロジェクトを作り、既存の HTML ファイルを取り込みます。この説明では次の想定で進めます。

- プロジェクトは `vue-cli` で作成
- 対象バージョンは Vue.js 3 で、クラス形式のコンポーネントではない関数ベースのコンポーネントを利用
- ルーターを利用

jQuery と比較した場合の、Vue.js の大きく違うポイントは次の通りです。

- Vue は特定のノード以下のみをプログラムで変更可能にする
- 1 つの HTML を元にページ切り替えを実現できる (シングルページアプリケーション)

元の jQuery でシングルページアプリケーションを実現しているケースはほとんどないと思うので、いったん、元のプロジェクトは複数ページから構成されているものとして話を進めます。

まず、複数のページであっても、1 つの HTML と TypeScript コードで実現します。今まで共通のヘッダーなどを個別に実装していた場合はベースの HTML 側に書いておけば共通で利用されます。複数のページで違いの発生する部分のみを .vue ファイルにします。

まずは jQuery をライブラリに追加します。

```
$ npm install jquery @types/jquery
```

次に、.vue ファイルを作成します。これは HTML、CSS、スクリプトが 1 ファイルにまとまった、シングルファイルコンポーネントと呼ばれるものです。<template>に HTML を書き、<script lang="ts">には TypeScript のコードを書きます。

DOM 読み込み後に呼ばれるイベントハンドラで jQuery のイベント定義などを行っていました。jQuery の読み込み後のハンドラーは次のどちらかを設定していました。

- \$(document).ready(function () {})
- \$(function() { })

Vue.js では、ページごとにコンポーネントと呼ばれるオブジェクトを作成します。それが作成されるタイミングで setup() メソッドが呼ばれますが、その中で onMounted() 関数に登録したコードが実行時に呼ばれます。まずはそこに jQuery のコードを移植してしまいましょう。完成形は次の通りです。

リスト 1 /src/views/mypage.vue

```
<template>
  <div>jQuery test page</div>
  <button class="pushbutton">button</button>
  <div class="panel" style="display: none; background-color: lightblue;">
    hidden
  </div>
</template>

<style scoped>
  .clickedButton {
    background-color: "yellow";
  }
</style>

<script lang="ts">
import $ from "jquery";

import { defineComponent, onMounted } from "vue";

export default defineComponent({
  name: "MyPage",
  setup() {
    onMounted(() => {
      $(".pushbutton").click(() => {
        $(".pushbutton").css("background-color", "yellow");
        $(".panel").fadeIn();
      });
    });
  }
})
```

(次のページに続く)

(前のページからの続き)

```
});  
</script>
```

なお、この `onMounted` は作成時に一度だけ呼ばれます。その後、属性の変更などがあると、タグの再作成プロセスが走ります。差分のみの更新のはずですが、元の変更によってはイベントハンドラをつけたタグが再作成されたりすることもあり、イベントハンドラがなくなる可能性もあります。このやり方はあくまでも移行のための暫定なので、本番環境にデプロイなどをしてはいけません。

ページを作ったら、そのファイルが表示可能になるように、ルーターに登録します。そのページの URL と、その時に使われるコンポーネントのペアを関連づけるものです。これで表示できるようになりました。

リスト 2 /src/router/router.ts

```
import { createRouter, createWebHistory, RouteRecordRaw } from "vue-router";  
import MyPage from "../views/mypage.vue";  
  
const routes: Array<RouteRecordRaw> = [  
  {  
    path: "/",  
    name: "MyPage",  
    component: MyPage  
  },  
];  
  
const router = createRouter({  
  history: createWebHistory(process.env.BASE_URL),  
  routes  
});  
  
export default router;
```

まずはここまでで動作することを確認しましょう。

30.3.2 クリックイベントの定義

Vue の作法でクリックイベントを設定しましょう。jQuery は HTML の外でタグとイベントを関連付けを `$.click()` を使って行っていました。Vue.js の場合は、テンプレートの中で関数を設定します。呼び出す関数は `setup` の中で作成します。テンプレートで `clicked` という名前呼びたい場合には次のように作成します。

リスト 3 /src/views/mypage.vue

```
import { defineComponent } from "vue";  
  
export default defineComponent({  
  name: "MyPage",
```

(次のページに続く)

(前のページからの続き)

```

setup() {
  // イベントハンドラを作成
  const clicked = () => {
    $(".pushbutton").css("background-color", "yellow");
    $(".panel").fadeIn();
  }
  // テンプレートで使う要素は setup のレスポンスのオブジェクトとして返す
  return {
    clicked
  }
}
});

```

HTML は次のようになります。素の HTML のイベントハンドラの方に近くなります。ここまで動作することを確認しましょう。

リスト 4 /src/views/mypage.vue

```
<button class="pushbutton" @click="clicked">button</button>
```

jQuery との大きな違いとしては、変更対象の HTML タグの発見方法があります。jQuery はすでにある HTML のタグの中から変更対象を見つける必要があります。そのため、セレクトーという機能に磨きをかけて、変更対象をすばやく確実に見つける機能を備えました。一方の Vue.js は、生成時に差し込みます。そのため、タグを探して「セレクトー」という概念がありません。

30.3.3 クリックによるインタラクションを Vue.js 化

Vue では一元管理されたデータを元にビューを更新するとすでに説明しました。その本丸に攻め入ります。その状態管理で利用するのが reactive 関数です。引数にオブジェクトを渡すと、それがコンポーネントのデータ源として利用できます。これもテンプレートから利用するため、setup() の返り値で返します。

今回のサンプルは小さいので問題ありませんが、実用的なコンポーネントの場合、この reactive() の各要素にコメントを書いておいても良いでしょう。

clicked() の中は、単にこの状態を更新するだけになりました。

リスト 5 /src/views/mypage.vue

```

import { defineComponent, reactive } from "vue";

export default defineComponent({
  name: "jQuery",
  setup() {
    const state = reactive({
      show: false // パネルの表示制御に使う
    })
  }
})

```

(次のページに続く)

(前のページからの続き)

```

    });

    const clicked = () => {
      state.show = true;
    };
    return {
      clicked,
      state
    };
  }
});

```

テンプレート側はつぎのようになりました。この状態をみて、ボタンの方はクラスの ON/OFF の切り替えを行い、パネルの方は CSS のスタイルの内容の変更を行っています。

:class は、オリジナルの Vue のルールでは v-bind:class ですが、省略記法として :class も提供されています。実は、@click も v-on:click の省略形になります。

リスト 6 /src/views/mypage.vue

```

<button
  @click="clicked"
  :class="{ clickedButton: state.show }"
>
  button
</button>
<div class="panel" :style="{visibility: state.show ? 'visible' : 'hidden',
  ↪background-color: 'lightblue'}">
  hidden
</div>

```

単に表示の ON/OFF を切り替えるだけであれば、v-if や v-show といったディレクティブもあります^{*1}。

30.3.4 フェードイン効果を実現する

クリックしたタイミングでスタイルシートや表示の ON/OFF の切り替え方法は学びました。本節のラストとして、フェードインを実現します。

jQuery のフェードインはふわっと表示させるときに利用します。デフォルト値は 0.4 秒での表示になります。また、透明度の変化のカーブの選択もできます。このあたりの複雑なパラメータの時間変化は、最新の CSS を使えば造作ありません。まずはフェードインの CSS クラスです。opacity と visibility が非表示状態で開始します。その後、クリックで追加のクラス設定があると最終的には完全表示になりますが、その途中経過で 0.4 秒かけて表示するための transition が含まれています。

^{*1} <https://v1-jp.vuejs.org/guide/conditional.html>

リスト 7 /src/views/mypage.vue

```

.hiddenPanel {
  opacity: 0;
  visibility: hidden;
}

.hiddenPanel.fadeIn {
  opacity: 1;
  visibility: visible;
  transition: opacity 0.4s, visibility 0.4s;
}

```

すでに実装されているロジックで問題ありません。残る修正箇所はテンプレートのみです。

リスト 8 /src/views/mypage.vue

```

<div
  class="panel" :class="{hiddenPanel: true, fadeIn: state.show}"
  :style="{ backgroundColor: 'lightblue' }">
  hidden
</div>

```

これで、クリックされた瞬間に fadeIn クラスが付与されるようになりました。この CSS クラスは 0.4 秒かけて切り替えると実装されているため、jQuery の fadeIn 同等の処理が実現できました。

30.3.5 Vue.js の方が行数が長くなる？

jQuery の移植の基本について説明してきました。イベントのハンドリングとスタイルの切り替えはこれで問題がなくなったでしょう。

これらの処理は jQuery がもっとも得意とするものであり、Vue.js の方がかなり行数が増えてしまっています。ここだけみると Vue.js のメリットが少なく感じます。

しかし、Ajax 的なロジックの実装を開始しはじめると複雑度は逆転します。サーバー通信して、取得してきた JSON を元にリストを表示するようなケースでは、jQuery の場合はプログラムの中で HTML の DOM 要素を作って挿入という処理になります。適切な HTML 片を作って挿入するのは大変ですし、雑に実装すると XSS というセキュリティの穴があく恐れもでてくるため、jQuery でも `mustache` や `underscore.js` の `template`、`handlebars` といったテンプレートエンジンの導入を検討し始めるところです。ロジックと表示の元データと表示のテンプレートが分離しにくいといった問題も出てきますし、コードの行数も爆発し始めます。表示以外に、編集や削除をしたいといった要件が出てきたらもうお手上げでしょう。

Vue.js の場合はスクリプトはデータの保持のみになりますし、テンプレートによる結果のレンダリングは最初から行っていますので、複雑度は変わりません。v-for ディレクティブでリストの表示も簡単です。

それ以外に jQuery の方が短く、コードが長くなってしまうケースとしては、jQuery プラグインを活用している場合があります。jQuery には jQuery UI も含めた高度なコンポーネントの部品が数多くあります。これらはとても便利ですが、やはりデータと表示の分離がしにくく、うまくルールに乗っかれるような最速のパターンでは短かったとしても、例えばカレンダー部品で取得した情報を元にバリデーションを行って結果を画面に反映させたり、インタラクションが増えてくると周辺のコードが複雑化しがちです。

まだ Vue.js 3 には対応していませんが、[bootstrap-vue](#) という人気のある Bootstrap を Vue.js 用にコンポーネント化したものもあります。複雑なフィルタリング機能などを実現するテーブルコンポーネントも [cheetahgrid](#) などのよりよい代替部品がすぐに見つかるでしょう。自分でコンポーネントを実装しなければならない場合を除けば、そこまで行数も変わらず実現できるでしょう。

30.4 まとめ

Vue.js について紹介してきました。まずは Vue の使い勝手の良さの源泉である `vue-cli` によるプロジェクト作成の方法について紹介しました。TypeScript フレンドリーな書き方については 2 系で相性が良いクラス形式の実装方法と、3 系の `composition API` の 2 種類紹介しました。

せめて、2 系にバックポートされた 3 系 API を実現するプラグインの `@vue/composition-api` のバージョンからベータが外れると、プロダクション開発もすべて `composition API` を利用する方針で決定できると思いますが、現時点ではどちらを使うかはプロジェクトの寿命やリリース時期を勘案してどちらか決める必要があるかもしれません。

そのあとで、よくありそうなユースケースとして、jQuery で書かれたウェブフロントエンドを Vue.js で置き換える方法について紹介しました。一見すると行数が増えますが、ロジックが増えても複雑度が増えていかないので、のちのちにありがたみを感じやすくなるでしょう。

第 31 章

Parcel を使ったウェブ開発

React 開発であれば create-react-app や Next.js、Vue であれば @vue/cli や Nuxt.js、Angular も @angular/cli といった、それぞれのフレームワークごとに環境構築をまとめて行う環境整備ツールが近年では充実してきています。

しかし、そこから多少離れるものであったり、あるいはより高速なビルドが欲しいなど、要件によっては他のビルドツールで環境を作る方がプロジェクトにはマッチする可能性もあります。本章では Parcel を使った環境構築について紹介します。

注釈: 現時点ではリリースされていない Parcel 2.0 beta 2 を対象に説明します。なお、このバージョンでは、TypeScript 4 系とは依存のバージョンがコンフリクトして同時にりようできませんが、インストール時に --force をつけることで利用はできます。

31.1 Parcel とは

- <https://v2.parceljs.org/>

Parcel はゼロコンフィグを目指したバンドラーです。ほとんど設定を書くことなく環境を準備できます。たとえばプロダクション開発では webpack を使ったとしても、小さいコードをすばやく書き上げる場合などに知っておいて損はないでしょう。Parcel はビルド設定ファイルを作成することなく、エントリーポイントのファイルを指定するだけでビルドできます。エントリーポイントのファイルはウェブ開発であれば HTML ファイルも使えます。その HTML から参照されているスクリプトファイルや CSS などすべて辿ってビルドしてくれます。

TypeScript も最初からサポートしています。tsconfig.json があればそれを拾って解釈してくれますし、なくても動きます。単に ts ファイルを script タグの src にするだけで、そのまま TypeScript の処理系をインストールしつつビルドしてくれます。最初のビルドも高速ですし、キャッシュもしてくれて 2 回目以降も速いです。TreeShaking とかの生成されたファイルの最適化機構も入っています。

なお、ゼロコンフィグというか、設定ファイルがなかったのが、ちょっと凝ったことをしようとすると、Parcel のビルド機能を API として呼び出すウェブサーバーを書かねばならず、かえって大変になることもありましたが、

Parcel 2 からはちょっとした設定ファイルでデフォルト設定を変更できるようになりました。例えば次のようなことができるようになります。

- TypeScript のビルドでエラーメッセージが出せるようになった
- API サーバーへのプロキシができるようになった

31.2 React 環境

Parcel は環境構築は簡単ですが、全自動の環境構築ツールはありません。まずはプロジェクトフォルダを作り、`package.json` を作成します。

```
$ npm init -y
```

基本の環境構築で紹介したように、Prettier と ESLint を設定しておきましょう。

次に Parcel を追加します。parcel-bundler というパッケージ名は v1 系列です。v2 系列は parcel になりました。まだ v2 は正式リリースされていないため、インストール時は@next（比較的安定板）、あるいは明示的に@2.0.0-beta.2 などのバージョンを指定するようにしてください。

```
$ npm install --save-dev parcel@2.0.0-beta.2
```

次に HTML ファイルを用意します。ポイントは前述のように、`<script>` タグに読み込ませたい TypeScript のコードを書くことです。

リスト 1 src/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Parcel Project</title>
  </head>
  <body>
    <div id="root"></div>
    <script src="./index.tsx"></script>
  </body>
</html>
```

最後にこのスクリプトを書き足します。

リスト 2 src/index.tsx

```
import React, { StrictMode } from 'react';
import { render } from 'react-dom';

render(
  <StrictMode>
    <div>test</div>
  </StrictMode>,
  document.getElementById('root')
);
```

ビルドを実行すればそのタイミングで拡張子を見て TypeScript をインストールして実行はしてくれますが、先にインストールして tsconfig.json を作っておきます。

```
$ npx install --save-dev typescript
$ npx tsc --init
```

デフォルトでは ES2015 modules が有効になっておらず、ビルドターゲットが古いのでそこを修正するのと、今回は React なので JSX も react にしておきます。

リスト 3 tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2018",
    "module": "es2015",
    "jsx": "react"
  }
}
```

リスト 4 package.json

```
{
  "scripts": {
    "start": "parcel serve src/index.html",
    "build": "parcel build src/index.html",
  }
}
```

31.2.1 TypeScript のエラーを表示する

これでビルドと開発はできますが、デフォルトの Parcel は `@babel/preset-typescript` を使って型情報を切り落とすだけで型のチェックは行ません。VSCode で編集すればその場でエラーチェックはしてくれますが、変更したファイルが他のファイルに影響を与えていてエラーになっていたり、警告が出ていた、というのはなかなか気付きにくいです。バリデーションを有効化すると、このようなトラブルは防げます。本体のバージョンと合ったバージョンをインストールします。

```
$ npm install --save-dev @parcel/validator-typescript@2.0.0-beta.2
```

リスト 5 .parcelrc

```
{
  "extends": "@parcel/config-default",
  "validators": {
    "*.ts,tsx": ["@parcel/validator-typescript"]
  }
}
```

31.2.2 API サーバーに対してプロキシする

`.proxyrc` ファイルを作成することで、一部のリクエストを API サーバーに受け流すといったことが可能です。これにより、フロントエンドとバックエンドが同じオリジンで動作するようになり、CORS などのセキュリティの環境整備が簡単になります。もし、本番環境も別ホストで配信するのであれば、元々 CORS の設定などは考慮されていて少ない労力でなんとかなると思われそうですが、そうでない場合、テスト環境のために CORS を設定するといった大仰なことをしなくて済みます。

リスト 6 .proxyrc

```
{
  "/api": {
    "target": "http://localhost:3000/"
  }
}
```

なお、パスのリライトなど、高度なこともできます。しかし、動作しなかったときの問題追跡が面倒になるため、ホスト名の転送だけで済むようにしておくといいでしょう。

31.3 WebComponents 開発環境

あとで書く

第 32 章

Electron アプリケーションの作成

マルチプラットフォームなデスクトップアプリケーションを簡単に作る方法として近年人気なのが GitHub の開発した Electron を使った開発です。Chromium と Node.js が一体となった仕組みになっています。UI はブラウザで「レンダープロセス」が担い、その UI の起動やローカルファイルへのアクセスなどを行うのが「メインプロセス」です。

レンダープロセスとメインプロセスはなるべく疎結合に作ります。プログラムの量はおそらくレンダープロセスが 95% ぐらいの分量になるでしょう。SPA のウェブフロントエンドとウェブサーバーを作る感覚よりも、さらにフロントに荷重が寄った構造になるでしょう。

Electron のランタイムと、ビルドした JavaScript をまとめて、インストーラまで作成してくれるのが Electron-Build です。これを使ってアプリケーションの開発を行っていきます。Vue の場合は Vue CLI 用のプラグイン^{*1}があります。

- <https://www.electron.build/>

32.1 React+Electron の環境構築の方法

Electron の開発は 2 つ作戦が考えられます。一つが、ウェブのフロントエンドとして、そちらのエコシステムを利用して開発します。もう一つが、普段の開発から Electron をランタイムとして開発する方法です。どちらか慣れている方で良いでしょう。

後者については次の Qiita のエントリーが詳しいです。

- <https://qiita.com/yhirose/items/22b0621f0d36d983d8b0>
- <https://github.com/yhirose/react-typescript-electron-sample-with-create-react-app-and-electron-builder>

本書では、前者の方法について紹介します。まず、プロジェクトを作成します。今回は 2 つのエントリーポイントのビルドが必要なため、これに対応しやすい Parcel を利用します。

^{*1} <https://nklayman.github.io/vue-cli-plugin-electron-builder/>

```
# プロジェクトフォルダ作成
$ mkdir electronsample
$ cd electronsample
$ npm init -y

# 必要なツールをインストール
$ npm install --save-dev parcel@next typescript @parcel/validator-typescript@2.0.0-
  ↳ nightly.430
$ npm install --save-dev react @types/react react-dom @types/react-dom
$ npm install --save-dev electron npm-run-all

# tsconfig 作成
$ npx tsc --init
```

tsconfig.json は、いつものように target/module/jsx あたりを修正しておきます。

リスト 1 tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2020",
    "module": "es2015",
    "jsx": "react",
    "strict": true,
    "moduleResolution": "node",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

もう一つ、メインプロセス用の tsconfig も作ります。こちらは Node.js 用に近い形式で出力が必要なため、commonjs 形式のモジュールに設定しています。

リスト 2 tsconfig.main.json

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "dist",
    "module": "commonjs",
    "sourceMap": true,
  },
  "include": [
    "src/main/*"
  ]
}
```

次にファイルを 3 つ作ります。

リスト 3 src/render/index.html

```
<!DOCTYPE html>
<head>
  <meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-inline
  ↳';" />
</head>
<body>
  <div id="app"></div><script src="index.tsx"></script>
</body>
```

リスト 4 src/render/index.tsx

```
import React from "react";
import { render } from "react-dom";

const App = () => <h1>Hello!</h1>;

render(<App />, document.getElementById("app"));
```

リスト 5 src/main/main.ts

```
import { app, BrowserWindow } from 'electron';

let win: BrowserWindow | null = null;

function createWindow() {
  win = new BrowserWindow({ width: 800, height: 600 })
  win.loadURL(`file://${__dirname}/index.html`);
  win.on('closed', () => win = null);
}

app.on('ready', createWindow);

app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

app.on('activate', () => {
  if (win === null) {
    createWindow();
  }
});
```

package.json のスクリプトも追加しておきましょう。レンダープロセス部分は Parcel を使い、メインプロセスには TypeScript の tsc コマンドをダイレクトで使っています。tsc はバンドルをせずに、ソースファイルに対して 1:1 で

変換した結果を出力します。メインプロセスは@vercel/ncc を使っても良いと思いますが、Electron ではレンダープロセス起動時の初期化スクリプト (preload) も設定できるため、生成したいファイルは複数必要になりますが、残念ながら@vercel/ncc は複数のエントリーポイントを扱うのが得意ではないため、ここではバンドルをせずに tsc で処理をしています。外部ライブラリを利用する場合などはメインプロセスも ncc でバンドルを作成する方が良いでしょう。

もう一つのポイントは"browser"と"main"です。生成した JavaScript を元に、ユーザーに配布しやすい形にランタイム込みのバンドルを作成する electron-builder は main の項目を見てビルドを行います。また、Parcel も同じくデフォルトで main を見ますが、electron-builder の main はメインプロセス、Parcel で処理をするのはレンダープロセス側です。そのため、parcel コマンドのオプションで、main じゃない項目（ここでは browser）に書かれたファイル名で出力するように--target オプションを設定しています。

リスト 6 package.json

```
{
  "browser": "dist/index.html",
  "main": "dist/main.js",
  "scripts": {
    "serve": "parcel serve src/render/index.html",
    "build": "run-p build:main build:render",
    "build:main": "tsc -p tsconfig.main.json",
    "build:render": "parcel build --dist-dir=dist --public-url --target=browser \"./\" \"src/render/index.html\",
    "start": "run-s build start:electron",
    "start:electron": "electron dist/main/index.js"
  }
}
```

次のコマンドで開発を行っていきます。

- npm run serve: フロントエンド部分をブラウザ上で実行します。
- npm run build: レンダープロセス、メインプロセス 2 つのコードをビルド
- npm start: ビルドした結果を electron コマンドを使って実行

32.2 配布用アプリケーションの構築

これまで作ってきた環境は開発環境で、Electron 本体を npm からダウンロードして実行します。エンドユーザー環境には npm も Node.js もないことが普通でしょう。Electron の本体も一緒にバンドルしたシングルバイナリの実行可能アプリケーションを作成していきます。ビルドには electron-builder を利用します。

- <https://www.electron.build/>

インストールは npm で行います。

```
npm install --save-dev electron-builder
```

electron-builder の設定は package.json に記述します。output フォルダを設定しないと dist に出力され、Parcel などの出力と最終的なバイナリが混ざり、2 回目以降のビルド時にその前までにビルドした結果のファイルまでバンドルされてしまってファイルサイズがおかしなことになるため、dist と別フォルダを設定します。

リスト 7 package.json

```
{
  "scripts": {
    "electron:build": "run-s build electron:bundle",
    "electron:bundle": "electron-builder"
  },
  "build": {
    "appId": "com.example.electron-app",
    "files": [
      "dist/**/*",
      "package.json"
    ],
    "directories": {
      "buildResources": "resources",
      "output": "electron_dist"
    },
    "publish": null
  }
}
```

次のコマンドで配布用のバイナリが作成できます。

- `npm run electron:build`

これは本当の最小限です。electron-builder を利用すると、アイコンをつけたり、署名をしたりもできますし、クロスビルドも行えます。

32.3 デバッグ

普通のブラウザでは開発者ツールを開かないことには console.log も利用できません。Electron もレンダープロセスのデバッグには開発者ツールが使いたくなるでしょう。開発者ツールを起動するには 1 行書くだけで済みます。環境変数やモードを見て開くようにすると便利でしょう。

```
win.webContents.openDevTools();
```

32.4 レンダープロセスとメインプロセス間の通信

レンダープロセスは通常のブラウザに近いものと紹介しましたが、セキュリティの考え方もほぼ同様です。Electron ではブラウザウィンドウを開くときにどのページを開くかを指定しましたが、ここでは外部のサービスを開くこともできます。普段はローカルのファイルで動くが、リモートのサービスも使えるブラウザです。

```
win.loadURL(`https://google.com`);
```

ただし、このリモートのサービスが使える点が Electron のセキュリティを難しいものにしています。Electron には、レンダープロセスで Node.js の機能が使えるようになる `nodeIntegration` という機能があり、ブラウザウィンドウを開くときのオプションで有効化できます。しかしこれを有効化すると、ローカルのユーザー権限で見られるあらゆる場所のファイルにアクセスできますし、ファイルを書き換えたりできてしまい、クロスサイトスクリプティング脆弱性を入れ込んでしまうときのリスクが極大化されてしまうため、レンダープロセスが外部のリソースをロードする場合はこの機能はオフにすべきです（現在のデフォルトはオフです）。OpenID Connect の認証など、外部のリソースをロードしたいことはよくあるので、この機能はもうなかったものとして考えると良いでしょう。

代わりに提供されているのがコンテキストブリッジになります。歴史的経緯などは次のページにまともっています。

- Electron (v10.1.5 現在) の IPC 通信入門 - よりセキュアな方法への変遷: <https://qiita.com/hibara/items/c59fb6924610fc22a9db>

まず、ウィンドウを開くときのオプションで、`nodeIntegration` をオフに、`contextIsolation` をオンにします。後者は、これからロードするプリロードのスクリプトが直接ブラウザプロセスの情報にアクセスできないようになります。

リスト 8 main.ts

```
const win = new BrowserWindow({
  webPreferences: {
    nodeIntegration: false,
    contextIsolation: true,
    preload: __dirname + '/preload.js'
  }
});
```

次に、レンダープロセスに API を追加します。preload スクリプトを使うことで、レンダープロセスのグローバル変数に関数を追加できます。ここでは、`window.api.writeFile()` という関数を定義しています。このスクリプトは 2 つのプロセスの中間地点です。ブラウザプロセスとは別のコンテキストで実行されます。どちらかというと、レンダープロセス寄りですが、レンダープロセスの内では直接扱えない機能が利用できます。`ipcRenderer` が、メインプロセスとレンダープロセス間の通信を行うオブジェクトです。このコンテキストブ

リッジ内で `ipcRenderer` の `send()` や `on()` を呼び出すことで、メインプロセスに対する送信と受信が実現できます。

リスト 9 preload.ts

```
// eslint-disable-next-line
const { contextBridge, ipcRenderer } = require('electron');
contextBridge.exposeInMainWorld('api', {
  writeFile: (data) => {
    ipcRenderer.send('writeFile', data);
  },
});
```

`ipcRenderer` と対になる `ipcMain` を使って通信を行います。

リスト 10 main.ts

```
import { app, ipcMain } from 'electron';
import { writeFileSync } from 'fs';
import { join } from 'path';

ipcMain.on('writeFile', (_event, data) => {
  const jsonString = JSON.stringify(data, null, 4);
  writeFileSync(join(app.getPath('userData'), jsonString), 'utf8');
});
```

これにより、ブラウザプロセス側には間接的にファイル読み書きを行う API を登録し、それ経由で、実際の危険な操作をとまなうメインプロセス側の処理を呼び出すことが可能です。

表 1 Electron のプロセス間通信

通信方向	レンダープロセス側	メインプロセス側
レンダープロセス → メインプロセス	<code>ipcRenderer.send()</code>	<code>ipcMain.on()</code> に登録したコールバック
メインプロセス → レンダープロセス	<code>ipcRenderer.on()</code> に登録したコールバック	<code>ipcMain.send()</code>

32.5 まとめ

Electron について、環境の構築から配布用バイナリの作成、Electron ならではの開発のトピックを紹介してきました。近年のデスクトップアプリケーションの開発ではかなり人気のある選択肢となっています。TypeScript とブラウザのアプリケーションの知識があればデスクトップアプリケーションが作成できます。フロントエンド系の開発者にとっては福音と言えるでしょう。

Chrome ベースの Edge が利用できるようになって、ブラウザ間の機能差は小さくなりましたが、Electron はすべてのユーザーに同一バージョンの最新ブラウザを提供するようなものでもあるため、社内システム開発でも使いた

いというニーズはあるでしょう。また、ファイルシステムアクセスなど、ブラウザだけでは実現できない機能もいろいろ利用できます。

一方で、ツールバー、トレイなど、デスクトップならではのユーザビリティも考慮する必要は出てきますし、メニュー構成も Windows 標準と mac の違いなどもありそうです。フロントエンドの開発だけではなく、違和感なく使ってもらえるアプリケーションにするには、プラスアルファの手間隙がかかることは忘れないようにしてください。

第 33 章

おすすめのパッケージ・ツール

ばちばち追加。まだ実戦投入していないものも多数あります。

33.1 TypeScript Playground

ブラウザで TypeScript が試せる Playground にも何種類かありますが、現在は公式の Playground が一番おすすめです。以前は本家よりも強力というのを売りにしていたオープンソースのサイトもありましたが、v2 になり、処理系のバージョン設定、細かいビルドオプションの設定など、この別実装にあった機能はすべて本家にも実装されました。また、本書執筆時点では V3 となった最新の Playground が利用できるようになりました。こちらはビルド結果の.d.ts ファイルの結果も確認することができ、書いたコードがどのように TypeScript 処理系に解釈されたのかを確認するのも簡単になりました。

- URL: <https://www.typescriptlang.org/play/>

注釈: これらのサイトを使うということは、ソースコードを外部のサービスに送信することになります。実際にこれらのサービスがサーバーにデータを保存はしていないはずですが、普段からの心がけとして、業務で書いた外部公開できないコードを安易に貼り付けるようなことはしないようにしてください。

33.2 ビルド補助ツール

33.2.1 Rush

- npm パッケージ: `@microsoft/api-extractor`
- TypeScript 型定義: CLI ツールなので不要
- URL: <https://rushjs.io/>

ひとつの Git リポジトリの中に、多数の TypeScript ベースのパッケージを入れて管理するための補助ツール。次のサンプルを見ると、このツールを使った結果がわかります。

<https://github.com/microsoft/web-build-tools>

- apps フォルダ: ウェブアプリケーションが格納される
- libraries フォルダ: npm install で使うライブラリが格納される
- tools フォルダ: npm install で使うコマンドラインツールが格納される

33.2.2 API extractor

- npm パッケージ: [@microsoft/api-extractor](#)
- TypeScript 型定義: CLI ツールなので不要
- URL: https://api-extractor.com/pages/overview/demo_docs/

ドキュメントジェネレータ。パッケージのリファレンスマニュアルが作れる。

33.2.3 cash

- npm パッケージ: [cash](#)
- TypeScript 型定義: [@types/cash](#)

Unix シェルコマンドを Node.js で再実装したもの。Windows の PowerShell が rm などのエイリアスを提供してしまっている（しかも rm -Force -Recurse のようにオプションが違う）が、cash を使うとクロスプラットフォームで動くファイル操作が行える。npm scripts でも利用できるが、プログラム中からも使えるらしい。

課題: cash に export があるので、cross-env はいらないかも？

33.2.4 cross-env

- npm パッケージ: [cross-env](#)
- TypeScript 型定義: CLI ツールなので不要

Windows と Linux/macOS で環境変数変更を統一的に扱えるパッケージ。

33.2.5 typesync

- npm パッケージ: `typesync`
- TypeScript 型定義: CLI ツールなので不要

インストールされているパッケージの型定義パッケージを自動取得してくる CLI ツール。

33.3 コマンドラインツール用ライブラリ

33.3.1 convict

- npm パッケージ: `convict`
- TypeScript 型定義: `@types/convict`

コマンドライン引数、設定ファイル、環境変数などを統合的に扱える設定情報管理ライブラリ。型情報付きで設定を管理できるし、設定内容のバリデーションもできる。TypeScript で使うとさらに便利。

33.3.2 @microsoft/ts-command-line

- npm パッケージ: `@microsoft/ts-command-line`
- TypeScript 型定義: 内蔵

TypeScript 用のコマンドライン引数ライブラリです。Microsoft 社純正。

33.4 アルゴリズム関連のライブラリ

33.4.1 p-map

- npm パッケージ: `p-map`
- TypeScript 型定義: 同梱

並列数を制御しながら多数の仕事を平行で処理できる `Promise.all()`。

第 34 章

貢献者

34.1 Pull Request/Issue をくださった方々

- @called-d
- @tkihira
- @kaakaa
- @uult
- @isdh
- @t0yohei
- @shrkw
- @atlol
- @numb86
- @aaaaayako
- @WATANAPEI
- @kensuke
- @ebiiim
- @yinm
- @hidaruma
- @7ma7X
- @isuzuki

- @taiyoooooooooooo
- @yaegassy
- @tmitz
- @ichikawa-hiroki
- @eduidl
- @inductor
- @tamo
- @ota-meshi
- @ravelll
- @mottox2
- @masayuki0109
- @hnakamur
- @c-bata
- @Yu0614
- @bubusuke
- @mendelssohnbach
- @maito1201
- @aliyome
- @mimosafa
- @irtfrm

34.2 フィードバックをくださった方々

- @satotaichi
- @shun_shushu

課題: デコレータを使って DI。他にある？

<https://github.com/Microsoft/tsyringe>

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/advance.rst の 4 行目です)

課題: travis、circle.ci、gitlab-ci の設定を紹介。あとは Jenkins ?

<https://qiita.com/nju33/items/72992bd4941b96bc4ce5>

<https://qiita.com/naokikimura/items/f1c8903eec86ec1de655>

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/ci.rst の 4 行目です)

課題: npm パッケージ to npm npm パッケージ to nexus

<https://qiita.com/kannkyo/items/5195069c65350b60edd9>

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/deploy.rst の 6 行目です)

課題: Deno はこちらのスレッドを見守る <https://github.com/denoland/deno/issues/3356>

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/deploy.rst の 208 行目です)

課題: あとで書く

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/ecosystem.rst の 42 行目です)

課題: // browser/module など// <https://qiita.com/shinout/items/4c9854b00977883e0668>

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/libenv.rst の 131 行目です)

課題: 要検証

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/module.rst の 190 行目です)

課題: ちょっとうまく動いていないので、要調査

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/module.rst の 313 行目です)

課題: cash に export があるので、cross-env はいらないかも？

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/recommended.rst の 54 行目です)

課題: 事例をつける

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/typing.rst の 59 行目です)

課題: lynt の TypeScript 対応状況を注視する

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/baseenv.rst の 41 行目です)

課題: tsdoc とかドキュメントツールを紹介

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/baseenv.rst の 470 行目です)

課題: eslint やテストの書き方の紹介

(元のエントリ は、 /Users/shibukawayoshiki/book/typescript-guide/baseenv.rst の 472 行目です)

第 35 章

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

索引

RFC

| RFC 3393, 90