
仕事ですぐに使える **TypeScript**

Future Corporation

2019 年 11 月 20 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	JavaScript のバージョン	6
1.5	本書の参考文献など	6
1.6	ライセンス	7
1.7	本書で扱わないこと	7
1.8	TypeScript と互換性	7
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	12
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	14
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	16
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	42
5.3	まとめ	47
第 6 章	基本的な構文	49
6.1	制御構文	49
6.2	式	53
6.3	まとめ	54
第 7 章	基本的な型付け	55
7.1	一番手抜きな型付け: any	55
7.2	未知の型: unknown	56
7.3	型に名前をつける	56
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	57
7.5	属性名が可変のオブジェクトを扱う	58
7.6	A かつ B でなければならない	58
7.7	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	59
7.8	型ガード	61
7.9	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	64
7.10	インタフェースを使った型定義	66
7.11	まとめ	66
第 8 章	関数	69
8.1	アロー関数	69
8.2	関数の引数と返り値の型定義	70
8.3	関数を扱う変数の型定義	72
8.4	デフォルト引数	73
8.5	関数を含むオブジェクトの定義方法	74
8.6	this を操作するコードは書かない (1)	75
8.7	即時実行関数はもう使わない	76
8.8	まとめ	76
第 9 章	クラス	79
9.1	用語の整理	79
9.2	基本のクラス宣言	80
9.3	アクセス制御 (public/protected/private)	81
9.4	コンストラクタの引数を使ってプロパティを宣言	82
9.5	static メンバー	83
9.6	インスタンスクラスフィールド	84

9.7	読み込み専用の変数 (readonly)	85
9.8	メンバー定義方法のまとめ	86
9.9	継承/インタフェース実装宣言	86
9.10	クラスとインタフェースの違い・使い分け	88
9.11	デコレータ	89
9.12	まとめ	89
第 10 章	非同期処理	91
10.1	非同期とは何か	91
10.2	コールバックは使わない	92
10.3	非同期と制御構文	95
10.4	Promise の分岐と待ち合わせの制御	96
10.5	ループの中の await に注意	97
10.6	まとめ	98
第 11 章	例外処理	99
11.1	TypeScript の例外処理構文	99
11.2	Error クラス	100
11.3	例外処理とコードの読みやすさ	102
11.4	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	103
11.5	例外処理を使わないエラー処理	105
11.6	非同期と例外処理	105
11.7	例外とエラーの違い	107
11.8	まとめ	108
第 12 章	モジュール	109
12.1	用語の整理	109
12.2	基本文法	112
12.3	中級向けの機能	115
12.4	ちょっと上級の話	116
12.5	まとめ	118
第 13 章	Vue.js でアプリケーションを作成してみる	119
第 14 章	ジェネリクス	121
14.1	ジェネリクスの書き方	121
14.2	ジェネリクスの型パラメータに制約をつける	122
14.3	型パラメータの自動解決	123
14.4	ジェネリクスの文法でできること、できないこと	124
14.5	型変換のためのユーティリティ型	125
14.6	any や unknown、Union Type との違い	126
14.7	まとめ	127

第 15 章	関数型指向のプログラミング	129
15.1	イミュータブル	129
第 16 章	クラス上級編	131
16.1	アクセッサ	131
16.2	抽象クラス	133
16.3	まとめ	133
第 17 章	リアクティブ	135
第 18 章	JavaScript のライブラリに対する型定義ファイルの作成	137
第 19 章	高度なテクニック	139
第 20 章	ソフトウェア開発の環境を考える	141
第 21 章	基本の環境構築	145
21.1	作業フォルダの作成	146
21.2	ビルドのツールのインストールと設定	147
21.3	テスト	148
21.4	Visual Studio Code の設定	149
第 22 章	ライブラリ開発のための環境設定	151
22.1	ディレクトリの作成	152
22.2	ビルド設定	152
22.3	ライブラリコード	154
22.4	まとめ	154
第 23 章	CLI ツール作成のための環境設定	157
23.1	作業フォルダを作る	157
23.2	ビルド設定	157
23.3	CLI ツールのソースコード	159
23.4	バンドラーで 1 つにまとめる	159
23.5	まとめ	161
第 24 章	Next.js (React) の環境構築	163
24.1	作業フォルダを作る	164
24.2	ビルドのツールのインストールと設定	164
24.3	Next.js+TS のソースコード	167
24.4	まとめと、普段の開発	170
第 25 章	CI (継続的インテグレーション) 環境の構築	171
第 26 章	成果物のデプロイ	173

第 27 章	使用ライブラリのバージョン管理	175
27.1	バージョンとは	175
27.2	バージョン選びの作戦	176
27.3	バージョンアップの方法	178
27.4	バージョンアップ時のトラブルを減らす	180
27.5	なぜバージョンを管理する必要があるのか	181
27.6	まとめ	182
第 28 章	おすすめのパッケージ・ツール	185
28.1	TypeScript Playgournd 各種	185
28.2	ビルド補助ツール	186
28.3	コマンドラインツール用ライブラリ	187
28.4	アルゴリズム関連のライブラリ	187
第 29 章	貢献者	189
29.1	Pull Request をくださった方々	189
29.2	フィードバックをくださった方々	190
第 30 章	Indices and tables	193

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れたユーザーが、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よう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。

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、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 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。

本ドキュメントは TypeScript ファーストで説明していきますが、JavaScript との差異があるところは適宜補足します。

1.5 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](#) などが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、今後は参考にする可能性があります。

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

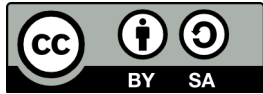
本書のベースとなっているのは、本原稿を執筆した渋谷が Qiita に書いたエントリーの [イマドキの JavaScript の書き方 2018^{*1}](#) と、それを元にして書いた [Software Design 2019 年 3 月号](#) の JavaScript 特集です。それ以外に、状況別の TypeScript の環境構築について書いた [2019 年版: 脱 Babel! フロント/JS 開発を TypeScript に移行するための環境整備マニュアル^{*2}](#) も内包していますし、他のエントリーも細々と引用しています。

^{*1} <https://qiita.com/shibukawa/items/19ab5c381bbb2e09d0d9>

^{*2} <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。

1.6 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/)^{*3} 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)」でお願いします。

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。

1.7 本書で扱わないこと

本書は 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。

また、ジェネリクスや型システムの複雑な機能には深入りはしません。TypeScript も、少しずつ着実にバージョンアップを重ねています。一般的に開発者がよく利用するようなケースにおいては、すでに書きやすくなるようにデザインされています。また、今現在は簡単に定義できないようなものがあっても将来バージョンで改善されて簡単に書けるようになると期待されるからです。

1.8 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^{*4} 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^{*5}。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。ブラウザ以外では Google Apps Script が ECMAScript3 にしか対応していません。

^{*3} <http://creativecommons.org/licenses/by-sa/4.0/deed.ja>

^{*4} Microsoft 社が Edge が Chromium ベースにすることが発表され、現在ベータ版が配布されています。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。

^{*5} 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。

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^{*6}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^{*7}。

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^{*8}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

^{*6} <http://kangax.github.io/compat-table/es6/>

^{*7} 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。

^{*8} <https://www.npmjs.com/package/core-js>

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";
```


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（まだまだアルファ）

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>

- 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、リリースにはいらない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。

npm install	dependencies と devDependencies の両方をインストールする。
npm install --prod	dependencies のみをインストールする。
npm ci	dependencies と devDependencies の両方をインストールする。package-lock.json は更新しない。
npm ci --prod	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 数字でも文字列でも受け取れる変数

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

// 正常
birthYear = 1980;
// これも正常
birthYear = '昭和';
// 答えたくないの 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 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 Union Type
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、ソースコード中に直接記述できるデータのことで、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。

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);

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

(次のページに続く)

```
// 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、式を整理するのもにも使えます。

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。

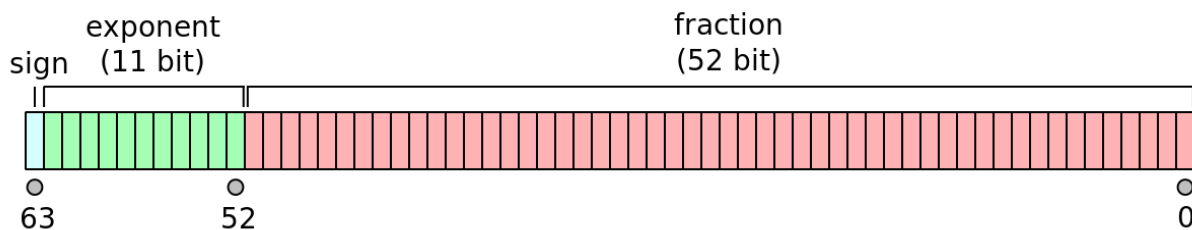


図 1 The memory format of an IEEE 754 double floating point value. by Codekaizen (CC 4.0 BY-SA)

```
// 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(0777); // 0, 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。

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みてください^{*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、改行があっても大丈夫なので、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 boolean
```

(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引っかからない、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、引っかかりそうで引っかからない、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、かっこ悪い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。共用体（union）型定義という、「A もしくは B」という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 か 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` は別のものであるので、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、エラーにはならなくなりますが、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。この場合違う型を入れようとするとエラーになります。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];
// error TS2322: Type 'number' is not assignable to type 'string'.
movie[0] = 2019;
```

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、新しいスプレッド構文 (...) 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。

スプレッド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。スプレッド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) {
```

(次のページに続く)

(前のページからの続き)

```
// 見つけた！
}

// 新: includes を利用
if (places.includes("小淵沢")) {
  // 見つけた！
}
```

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いましたが、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' ]

// 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' ]
```

(次のページに続く)

(前のページからの続き)

```
// 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 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];
  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。

5.1.6 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ば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];
```

5.1.7 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はなく、速度のメリットもあります。

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。キー名に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 4 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 5 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);

// 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 6 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 7 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;
```

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
```

(次のページに続く)

(前のページからの続き)

```
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};  
  
// 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"
};

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次の次の章で紹介します。

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。

5.3 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ーフェースです。このインター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

条件にあっている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えます。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 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
}
```

これを積極的に使う場面はというと、すでに JavaScript として動作していて実績があるコードを TypeScript にまづは持ってくる、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、まづ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。

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 Union Type 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っ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);
}
```

(次のページに続く)

(前のページからの続き)

```

}

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。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。

```

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"
};

```

このように型定義をしておくで、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 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` などでも設定できます。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。

7.6 A かつ B でなければならない

`A | B` という記法を紹介しました。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;
}
```

(次のページに続く)

(前のページからの続き)

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

この場合、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;
```

7.7 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)'
      }}
  ],
});
```

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。

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

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、**Union Type** 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、**Union Type** 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 Union Type 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)'
    }}
]
```

(次のページに続く)

(前のページからの続き)

```
    }}  
  ];
```

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 つだけ作る場所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;  
}
```

7.8 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
- 数字型か、文字列
- オブジェクトの特定の属性 `columnType` 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") {
    // 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()
    });
}
```

7.8.1 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"

null は "object" になりますが、それ以外のほとんどが object なので区別はつきませんので、null の判定には使えません。

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

7.8.2 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タイプ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

```
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

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。

7.8.3 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ますが、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;
```

7.9 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
```

(次のページに続く)

(前のページからの続き)

```
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"
```

7.10 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;  
  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"  
}
```

7.11 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。

また、型を実行時にあつかう方法

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

関数の定義、使い方などいろいろ変わりました。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 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 1 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 2 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  
// ただし型を書くとエラーになる  
arg1 => { /* 式 */ };  
  
// 引数が 0 の場合はカッコが必要  
( ) => { /* 式 */ };
```

(次のページに続く)

(前のページからの続き)

```
// 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 3 `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");
});

// 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));
```

8.2 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 4 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";
}

// 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 5 関数への型付け

```
// 文字列の toLowerCase() メソッドの戻り値は文字列なので
// 省略しても string が設定されたと思なされる
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 6 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'` という戻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、このように戻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、戻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。

```
// 戻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'.
```

8.3 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、戻り値だけは => 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戻り値は => 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。

```
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 7 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]) {
```

(次のページに続く)

(前のページからの続き)

```

    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"]

```

8.4 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、完全省略時でもデフォルト値が設定されるし、部分的に設定も可能みたい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、同じ型の引数があっ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、可変長引数があっ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あります。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 8 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;
    }
});

// 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"
    }
}
```

(次のページに続く)

(前のページからの続き)

```

    },
    _favorite: "小笠原",
    get favorite() {
        return this._favorite;
    },
    set favorite(favorite) {
        this._favorite = favorite;
    }
};

```

8.6 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ず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。関数の中では arguments という名前のちょっと配列っぽいオブジェクトです。ちょっと使いにくいので、一旦本物の配列にする時に 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スプレッド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);
}

```

(次のページに続く)

(前のページからの続き)

```
f(1, 2, 3, 4, 5, 6);
// 1, 2, [3, 4, 5, 6];

// 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。

8.7 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;
})();
```

8.8 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。

- アロー関数
- 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
- `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

クラス

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クラスとしてしまいます。なお、Java などのような書き味を求めて、こ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が、今の JavaScript と TypeScript では、より良い書き方が提供されています。

9.1 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。

9.2 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。今時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() {
    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");
};
```

9.3 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;

    dig(): string {
        return this.secretPlace;
    }

    // 埋める
    bury(treasure: string) {
        this.secretPlace = treasure;
    }
}

const miniatureDachshund = new SmallDog();
// 埋めた
miniatureDachshund.bury("骨");
```

(次のページに続く)

(前のページからの続き)

```
// 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。

9.4 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;
  }

  // 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
```

9.5 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);
    // 通常メソッドから静的プロパティを ``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
```

(次のページに続く)

(前のページからの続き)

```

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。

9.6 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。

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}が好きです`);
  };
}

```

^{*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

^{*3} <https://medium.freecodecamp.org/why-arrow-functions-and-bind-in-reacts-render-are-problematic-f1c08b060e36>

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。

9.7 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。それ以外のところでは、

```
class SimLockPhone {
  readonly carrier: string;
  constructor(carrier: string) {
    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) {
  }
}
```

9.8 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。

9.9 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 {
```

(次のページに続く)

(前のページからの続き)

```
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 () メソッドには戻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 インタフェースの戻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 返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返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。

9.10 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なるとか、拡張性のあるクラス設計が難しいとかもあり、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。

9.11 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();
```

9.12 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ど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 10 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。

10.1 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みます。

```
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。

10.2 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;
```

(次のページに続く)

(前のページからの続き)

```
func3(引数, function(err, value) {
    // 最後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同値になります。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 を返す関数の戻り値は必ず Promise になります。ジェネリクスのパラメータとして、戻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、関数の戻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);
  });
};

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
```

(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)

```
import { promisify } from "util";
import { readFile } from "fs";
const readFileAsync = promisify(readFile);

const content = await readFileAsync("package.json", "utf8");
```

10.3 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 finally() {
}

async function main(){
  if (Date.now() % 2 === 1) {
    await randomRun();
  }
  await finally();
}

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");
```

10.4 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。

10.5 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)  
  )  
);
```

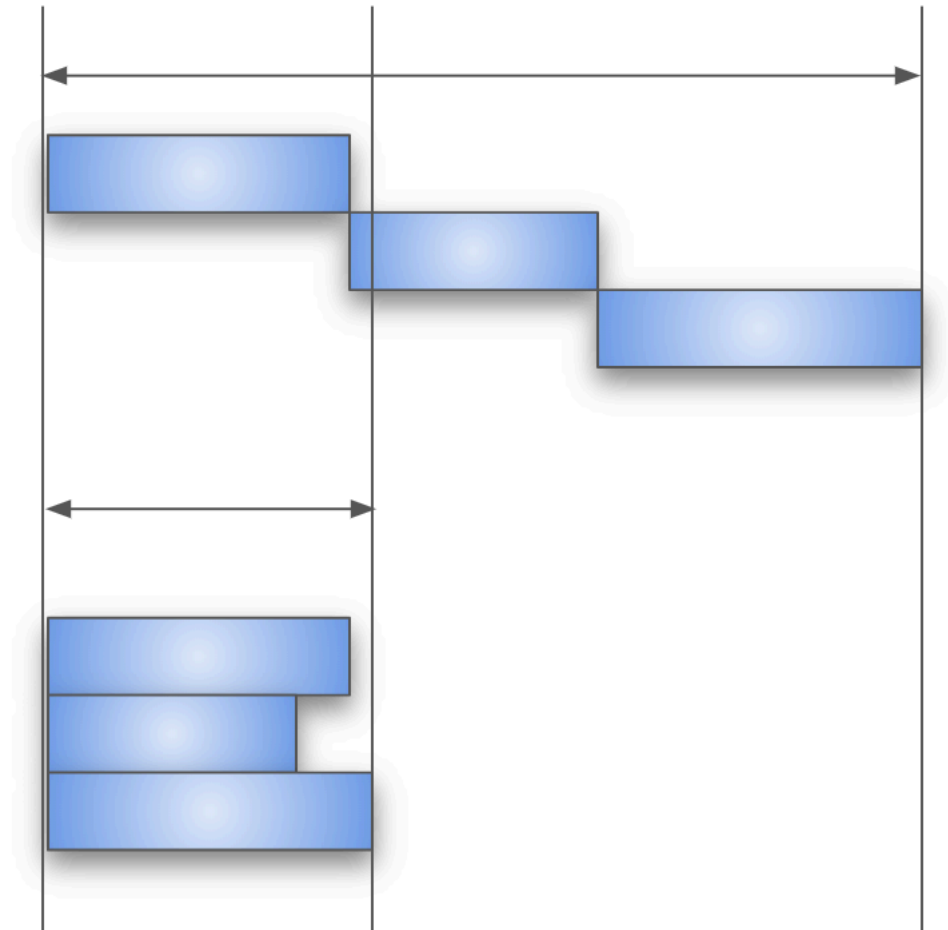
図で見て見ると、この違いは一目瞭然でしょう。

`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 `p-max*2` といった、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()` といったライブラリを活用すると良いでしょう。

`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

^{*2} <https://www.npmjs.com/package/p-map>



けませんし、この用途で使うところも見ただけありませんので、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。

10.6 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 11 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。

11.1 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: Error) {
```

(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、回復処理で回復仕切れない場合は、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: Error) {
  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: Error) {
  // 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
  }
}
```

11.2 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
```

11.2.1 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: Error) {
```

(次のページに続く)

(前のページからの続き)

```
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
```

11.3 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。

11.3.1 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: Error) {
  // 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: Error) {
  // 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やっているのと、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

11.3.2 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: Error) {
  // 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);
}
```

11.4 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: Error) {
  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。

11.5 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）メリットがあります。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する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
}
```

11.6 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()` 節が重なっていても、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: Error) {
  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。

```
const heavyTask = async (): Promise<number> => {
  return new Promise<number>((resolve, reject) => {
    // 何かしらの処理
    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) {
      // 最後に実行されるコードブロック
    });
  });
});
```

11.7 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かった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、データ入力した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

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。ユーザーには正常に正常に処理が進んだのか、そうじゃ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。

11.8 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の実装は楽になるでしょう。

第 12 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。

12.1 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。

12.1.1 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。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。

12.1.2 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れば良いでしょう。

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。

12.1.3 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

12.2 基本文法

12.2.1 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 {
}
```

12.2.2 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";
```

12.2.3 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";
```

12.2.4 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。

12.2.5 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

12.2.6 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。

12.3 中級向けの機能

12.3.1 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 };
```

12.3.2 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";
```

12.3.3 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。

12.4 ちょっと上級の話

12.4.1 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"
  }
}
```

12.4.2 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 との連携が必要です。

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すると問題ありません。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";
```

12.5 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。

第 13 章

Vue.js でアプリケーションを作成してみる

```
$ npm install -g @vue/cli
```

```
$ vue create vue-example
```

課題: クラスベースの **Vue** のプロジェクトの作成について説明する

第 14 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。

14.1 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);
```

14.1.1 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。

14.2 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 Union Type 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 Union Type だけを受け付ける
function action<T extends string>(actionName: T) {
    :
}

action<keyof ActionList>("register");

```

extends に Union Type を設定すればさらに特定の文字列だけに限定できます。

14.3 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 Union Type 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。

14.4 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。また完全に自由にするのではなく、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で、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。

14.5 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。

14.5.1 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

14.5.2 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 Union Type 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

14.5.3 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、Union Type をパラメータとして受け、新しい Union Type 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 Union Type から、`U` の Union Type の構成要素を除外した Union Type を作る型
- `Extract<T,U>`: `T` の Union Type と、`U` の Union Type の両方に含まれる Union Type を作る型
- `NonNullable<T>`: `T` の Union Type から、`undefined` を抜いた Union Type を作る型

14.5.4 関数のユーティリティ型

関数を渡すと、その返り値の型を返すユーティリティ型です。

- `ReturnType<T>`

14.5.5 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。

14.6 `any` や `unknown`、Union Type 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、Union Type 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 {
```

(次のページに続く)

(前のページからの続き)

```

    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;
    }
}

```

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

```

Union Type 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。

14.7 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 15 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、Schemer という関数型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。

15.1 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渡し、その結果の配列を新たに作って返します。

TypeScript は変数の再代入を防ぐ const はありますが、

<https://www.npmjs.com/package/iterative>

第 16 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。

16.1 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

16.2 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。

16.3 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 17 章

リアクティブ

第 18 章

JavaScript のライブラリに対する型定義ファイルの作成

ライブラリを使おうとしても型定義ファイルが見つからない場合、型定義ファイルを自作してコード補完や型チェックができるようにする必要があるかもしれません。しかし、これには中級以上のスキルが必要です。既存のコードを解析し、TypeScript の文法を理解したうえで、どのように型をあてれば使いやすくなるのか、という設計ができるのは初級レベルを超えています。

しかし、逆に言えば中級以上を名乗るであれば、必要に応じて型定義ファイルを作るスキルがある方が良いでしょう。

課題: CommonJS のモジュールの型づけの仕方について紹介する

TypeScript の書き方 <https://qiita.com/karak/items/b7af3cb2843c39fb3949>

// JSDoc から型定義ファイルを抽出 <https://www.npmjs.com/package/tsd-jsdoc>

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、少し機能が弱いため、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
- Browserify

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のように行われます。。

課題: あとで、開発サーバーとかもろもろについて語る

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。

課題: ESLint の TypeScript 対応はまだ開発途上で厳しそうなので TSLint に書き戻す？

- 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でこれですね。

- @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。

- 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` とかを使わず、`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

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、すべてに必要なになるため、`npm init` でファイルを作成します。

```
$ 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
Thumbs.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 ビルドのツールのインストールと設定

まず、最低限、インデントとか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くれます。

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
```

ツール群はこんな感じで入れました。

```
$ npm install --save-dev typescript prettier
  eslint @typescript-eslint/eslint-plugin
  eslint-plugin-prettier
  eslint-config-prettier npm-run-all
```

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。詳細は各パッケージの種類の章で取り扱います。

```
$ npx tsc --init
```

ESLint の設定も作ります。Prettier と連携するようにします。

リスト 4 .eslintrc

```
{
  "plugin": [
    "prettier"
  ],
  "extends": [
    "plugin:@typescript-eslint/recommended",
    "plugin:prettier/recommended"
  ],
  "rules": {
    "no-console": [
      false
    ],
    "@typescript-eslint/indent": "ignore",
    "prettier/prettier": "error"
  }
}
```

ESLint を起動するタスクを `package.json` に追加しましょう。これで、`npm run lint` や `npm run fix` でコードチェックをしたり、スタイル修正が行えます。

リスト 5 package.json

```
"scripts": {
  "lint": "eslint .",
  "fix": "eslint --fix ."
}
```

21.3 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
```

`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 6 package.json

```
{
  "scripts": {
    "test": "jest"
  }
}
```

リスト 7 jest.config.js

```
module.exports = {
  transform: {
    "^.+\\.tsx?$": "ts-jest"
  },
  moduleFileExtensions: [
    "ts",
    "tsx",
    "js",
    "json",
    "jsx"
  ]
};
```

21.4 Visual Studio Code の設定

Visual Studio Code でフォルダを開いたときに、eslint の拡張と、editorconfig の拡張がインストールされるようにします。

リスト 8 .vscode/extensions.json

```
{
  "recommendations": [
    "dbaeumer.vscode-eslint",
    "EditorConfig.editorconfig"
  ]
}
```

ファイル保存時に `eslint -fix` が自動実行されるように設定しておきましょう。これで Visual Studio Code を使う限り、誰がプロジェクトを開いてもコードスタイルが保たれます。eslint プラグインの `autoFixOnSave` は、`files.autoSave` が `off` のときにしか効かないので、それも設定しておきます。

リスト 9 .vscode/settings.json

```
{
  "eslint.autoFixOnSave": true,
  "files.autoSave": "off"
}
```

課題: tsdoc とかドキュメントツールを紹介

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おきます。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のように、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 ツール作成のための環境設定

TypeScript を使って Node.js のための CLI ツールを作成するための環境構築方法を紹介していきます。

23.1 作業フォルダを作る

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、出力先も CommonJS だけで十分です。

```
$ mkdir dist # commonJS だけなので 1 つだけ
```

``.gitignore`` も、出力先フォルダを 1 つだけに修正しましょう。

23.2 ビルド設定

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。

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 cli-color command-line-args  
  command-line-usage source-map-support  
$ npm install --save-dev @types/node @types/cli-color  
  @types/command-line-args @types/command-line-usage  
  @types/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,          //
    "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": "tsc --project .",
    "lint": "eslint .",
    "fix": "eslint --fix ."
  }
}
```

テストの設定、VSCode の設定は変わりません。

23.3 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();
```

23.4 バンドラーで 1 つにまとめる

npm で配るだけであればバンドルは不要ですが、ちょっとしたスクリプトを TypeScript で書いて Docker サーバーで実行したいが、コンテナを小さくしたいので `node_modules` は入れたくないということは今後増えていくと思い

ますので、そちらの方法も紹介します。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。

バンドラーでは一番使われているのは間違いなく webpack ですが、昔ながらの Browserify は設定ファイルレスで、ちょっとした小物のビルドには便利です。Browserify は Node.js の CommonJS 形式のコードをバンドルしつつ、Node.js 固有のパッケージは互換ライブラリを代わりに利用するようにして、Node.js のコードをそのままブラウザで使えるようにするものです。オプションで Node.js の互換ライブラリを使わないようにもできます。

webpack でもちょっと設定ファイルを書けばいけるはずですが、TypeScript 対応以外に、shebang 対応とかは別のプラグインが必要だったり、ちょっと手間はかかります。

Browserify をインストールします。tsify は Browserify プラグインで、TypeScript のコードを変換します。バンドルしてしまうので、バイナリのインストールが必要なパッケージ以外の各種ライブラリはすべて `--save-dev` で入れても大丈夫です。

```
$ npm install --save-dev browserify tsify
```

`tsconfig.json` に関してはそのまま問題ありません。この手のバンドラーから使われる時は、`.ts` と `.js` の 1:1 変換ではなくて、複数の `.ts` をメモリ上で `.js` に変換し、そのあとにまとめて 1 ファイルにする `n:1` 変換になります。そのため、個別の変換ファイルを出力しない `noEmit: true` をつけたり `distDir` を消したりする必要がありますが、そのあたりは tsify が勝手にやってくれます。

ただし、ES5 しか動作しない `low.js` を使う場合は、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
  }
}
```

ビルドスクリプトは次の形式になります。--node をつけないと、ブラウザ用のコードを生成しますが、ファイル入出力などが使えなくなりますので、CLI では --node を忘れずにつけます。TypeScript 変換のために、-p で tsify を追加しています。もし minify とかしたくなったら、minifyify などの別のプラグインを利用します。

```
{
  "scripts": {
    "build": "browserify --node -o dist/script.js -p [ tsify -p . ] src/index.ts"
  }
}
```

もし、バイナリを入れる必要のあるライブラリがあると、Browserify がエラーになります。その場合は、そのパッ

*1 <https://www.lowjs.org/>

ケースを `--exclude` 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。

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。

23.5 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
```


第 24 章

Next.js (React) の環境構築

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、Next.js について取り上げます。

注釈: 素の React と Vue.js と Angular

Next.js は React の上に作られたフレームワークですが、それ以外に人気のフレームワークに Vue.js と Angular があります。この 2 つに関しては、手動で環境を作る必要はありません。

Vue.js では vue の CLI コマンドを使ってプロジェクトを作成できますが、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 を選択してから TypeScript を選ぶとインストールと設定が完了します。

コア自体が TypeScript で作成されている Angular は、実装言語は TypeScript 以外が選べません。

また、React そのものも、create-react-app コマンドを使って環境構築を行う場合、--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 myapp --typescript

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!
```

なるべく、いろんなツールとの組み合わせの検証の手間を減らすために、Next.js を使います。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。

その中において、CSS in JS、Router をオールインワンにしてくれている Next.js は大変助かります。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のが一番強いです。Server Side Rendering もありますが、お仕事でやっていて一番ありがたいのはこの設定周りです。

24.1 作業フォルダを作る

Next.js では pages フォルダにおいてあるコンポーネントが Router に自動登録されるので、このフォルダをとりあえず作ります。あとは基本の環境構築と同じです。

```
$ mkdir pages
```

ウェブサービスを npm に公開することはないと思うので、.npmignore は不要ですが、.gitignore の方は、Next.js のファイル生成先の出力先フォルダを設定しておきます。

リスト 1 .gitignore

```
.next
```

24.2 ビルドのツールのインストールと設定

Next.js では next 以外にも、react、react-dom をインストールします。他にも必要なものを入れてしまいましょう。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。

```
$ npm install --save next react react-dom
$ npm install --save-dev @types/node @types/next @types/react @types/react-dom
$ npm install --save-dev typescript prettier
  eslint @typescript-eslint/eslint-plugin eslint-plugin-prettier
  eslint-config-prettier eslint-plugin-react npm-run-all
$ npm install --save-dev jest ts-jest @types/jest
```

Next.js を快適にするために TypeScript と、SCSS を入れます。Next.js では、本家が提供しているプラグインを使います。

```
$ npm install --save-dev @zeit/next-typescript @zeit/next-sass node-sass
```

Next.js だけでは真っ白なシンプルな HTML になってしまうので、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なパッケージが増えますね。

```
$ npm install --save @material-ui/core @material-ui/icons react-jss
```

`.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せず、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"。また、JS で書かれたコードも一部あるので、allowJs も true でなければなりません。

リスト 2 tsconfig.json

```
{
  "compilerOptions": {
    "allowJs": true,
    "allowSyntheticDefaultImports": true,
    "baseUrl": ".",
    "jsx": "preserve",
    "lib": ["dom", "es2017"],
    "module": "esnext",
    "moduleResolution": "node",
    "noEmit": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "preserveConstEnums": true,
    "removeComments": false,
    "skipLibCheck": true,
    "sourceMap": true,
    "strict": true,
    "target": "esnext"
  }
}
```

Babel 側にも設定を足します。

リスト 3 .babelrc

```
{
  "presets": [
    "next/babel",
    "@zeit/next-typescript/babel"
  ]
}
```

TypeScript と、SCSS のプラグインを有効化します。

リスト 4 next.config.js

```
const withTypescript = require("@zeit/next-typescript");
const withSass = require("@zeit/next-sass");

module.exports = withTypescript(
  withSass({
    webpack(config) {
      return config;
    }
  })
)
```

(次のページに続く)

(前のページからの続き)

```
);
```

Next.js の場合は、next コマンドがいろいろやってくれるので、やっていることの分量のわりに scripts がシンプルになります。

リスト 5 package.json

```
{
  "scripts": {
    "dev": "next",
    "build": "next build",
    "export": "next export",
    "start": "next start",
    "lint": "eslint .",
    "fix": "eslint --fix .",
    "test": "jest",
    "watch": "jest --watchAll"
  }
}
```

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 を有効にしないといけないとエラーになるので、そこも配慮します。

リスト 6 .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"]
    }]
  }
}
```

24.3 Next.js+TS のソースコード

まず Material UI を使うときに設定しなければならないコードがあるので、Material UI のサンプルページの `src/getPageContext.js`、`pages/_app.js`、`pages/_document.js` の 3 つのファイルをダウンロードして同じように起きます。Material UI の CSS in JS が Next.js 標準の方法と違うので、それを有効化してやらないと、サーバーサイドレンダリングのときに表示がおかしくなってしまいます。

次にページのコンテンツです。Next.js の規約としては、`pages` 以下のファイルが、`export default` で React コンポーネントを返すと、それがページとなります。ちょっと長いですが、TypeScript でページ作成するための方法を色々埋め込んであります。

リスト 7 pages/index.tsx

```
import Link from "next/link";
import React from "react";

import { Toolbar } from "@material-ui/core";
import AppBar from "@material-ui/core/AppBar";
import Button from "@material-ui/core/Button";
import Dialog from "@material-ui/core/Dialog";
import DialogActions from "@material-ui/core/DialogActions";
import DialogContent from "@material-ui/core/DialogContent";
import DialogContentText from "@material-ui/core/DialogContentText";
```

(次のページに続く)

(前のページからの続き)

```
import DialogTitle from "@material-ui/core/DialogTitle";
import {
  createStyles,
  Theme,
  withStyles,
  WithStyles
} from "@material-ui/core/styles";
import Typography from "@material-ui/core/Typography";

function styles(theme: Theme) {
  return createStyles({
    root: {
      paddingTop: theme.spacing.unit * 20
    }
  });
}

interface Props {
  children?: React.ReactNode;
}

interface State {
  openDialog: boolean;
}

class Index extends React.Component<
  Props & WithStyles<typeof styles>,
  State
> {
  public state = {
    openDialog: false
  };

  constructor(props: Props & WithStyles<typeof styles>) {
    super(props);
  }

  public handleCloseDialog = () => {
    this.setState({
      openDialog: false
    });
  };

  public handleClickShowDialog = () => {
    this.setState({
      openDialog: true
    });
  };
}
```

(次のページに続く)

(前のページからの続き)

```
public render() {
  const { classes } = this.props;
  const { openDialog } = this.state;

  return (
    <div className={classes.root}>
      <Dialog open={openDialog} onClose={this.handleCloseDialog}>
        <DialogTitle>Dialog Sample</DialogTitle>
        <DialogContent>
          <DialogContentText>
            Easy to use Material UI Dialog.
          </DialogContentText>
        </DialogContent>
        <DialogActions>
          <Button
            color="primary"
            onClick={this.handleCloseDialog}
          >
            OK
          </Button>
        </DialogActions>
      </Dialog>
      <AppBar>
        <Toolbar>
          <Typography variant="h6" color="inherit">
            TypeScript + Next.js + Material UI Sample
          </Typography>
        </Toolbar>
      </AppBar>
      <Typography variant="display1" gutterBottom={true}>
        Material-UI
      </Typography>
      <Typography variant="subheading" gutterBottom={true}>
        example project
      </Typography>
      <Typography gutterBottom={true}>
        <Link href="/about">
          <a>Go to the about page</a>
        </Link>
      </Typography>
      <Button
        variant="contained"
        color="secondary"
        onClick={this.handleClickShowDialog}
      >
        Shot Dialog
      </Button>
    </div>
  );
}
```

(次のページに続く)

(前のページからの続き)

```
        <style jsx={true}>{\`
          .root {
            text-align: center;
          }
        \`}</style>
      </div>
    );
  }
}

export default withStyles(styles)(Index);
```

まずは、React のコンポーネントを TypeScript で書くための Props や State の型定義の渡し方ですね。Component のパラメータとして type を設定します。やっかいなのは、Material UI のスタイル用の機能です。テーマを元に少し手を加えればできる、という仕組みが実現されていますが、TypeScript でやるには少々骨が折れます。それが styles 関数と withStyles(styles) の部分です。

24.4 まとめと、普段の開発

これで一通り、React を使う環境ができました。BFF 側に API 機能を持たせたいとか、Redux を使いたい、というのがあればここからまた少し手を加える必要があるでしょう。

開発は `npm run dev` で開発サーバーが起動し、ローカルのファイルの変更を見てホットデプロイとリロードを行ってくれます。

デプロイ時は `npm run build` とすると、`.next` フォルダ内にコンテンツが生成されます。`npm run build` の後に、`npm run export` をすると、静的ファイルを生成することもできます。ただし、いくつか制約があったりしますので、ドキュメントをよくご覧ください。

React も、ここまでくればそんなに難しくないですよ。

第 25 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 26 章

成果物のデプロイ

課題: npm パッケージ to npm npm パッケージ to nexus Docker イメージ

<https://qiita.com/kannkyo/items/5195069c65350b60edd9>

<https://qiita.com/shibukawa/items/fd49f98736045789ffc3#%E3%83%95%E3%83%AD%E3%83%B3%E3%83%88%E3%82%A8%E3%83%B3%E3%83%89%E5%91%A8%E3%82%8A%E3%81%AEdocker%E8%A8%AD%E5%AE%9A>

第 27 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。

27.1 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 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

27.1.1 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** でなくなって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 12 ですが、これはまだ LTS ではなくて、13 が出ると 12 が LTS になる、というのは要注意です。12 はメジャーバージョンアップですが、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。

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 の最終盤が LTS として 18 ヶ月サポートされると宣言されています。

27.2 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です。「そんなの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。

27.2.1 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、いろいろ食べ合わせが悪くて苦労したことがあります。

27.2.2 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、という方法があります。

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。

27.2.3 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。

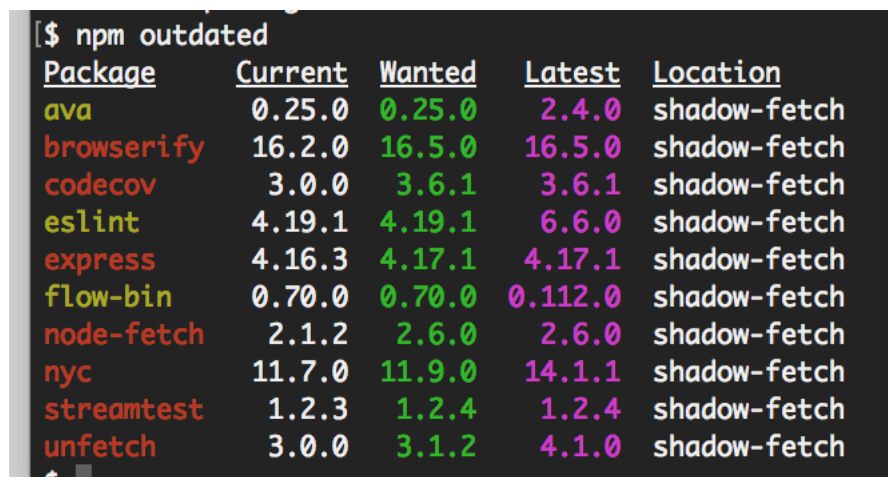
脆弱性のあるパッケージは自分が直接インストールしたもので発生するだけではなく、そのパッケージが利用している別のパッケージのさらに依存しているパッケージが・・・みたいな依存の深いところで起きがちです。特にウェブフロントエンド開発をしていると、依存パッケージ数が簡単に4桁とかいってしまうので、すべてを目視で確認するのは難しいです。自動検知を活用しましょう。

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。

27.3 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 outdated
```

Package	Current	Wanted	Latest	Location
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

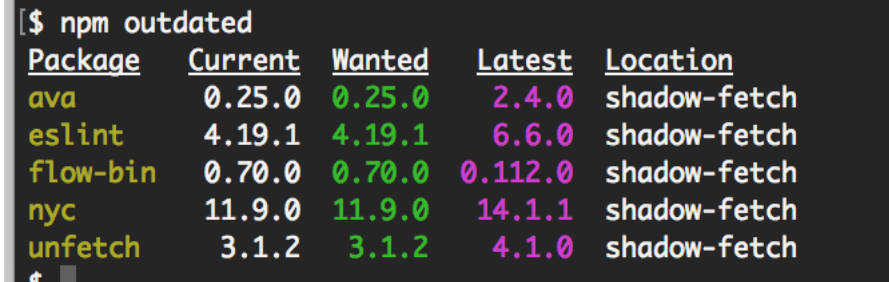
図1 `npm outdated` の実行結果

`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

# 一部だけ更新
$ npm update express
```



```
$ npm outdated
Package   Current  Wanted  Latest  Location
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
```

27.3.1 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。

27.4 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。

27.4.1 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。

27.4.2 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。

27.4.3 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。

27.4.4 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。間違っても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。

27.4.5 （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使い方が多いです。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。

27.5 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と、主に 2 つの理由があります。

27.5.1 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がらないわー、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。

27.5.2 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えます。

27.5.3 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

27.6 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ド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、本性の内容を頭の片隅に置いてもらえると幸いです。

第 28 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。

28.1 TypeScript Playgournd 各種

ブラウザで TypeScript が試せる Playground にも何種類かあります。

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。

28.1.1 標準の Playground

- URL: <https://www.typescriptlang.org/play/>

TypeScript の公式サイトが提供する Playground です。エラーチェックの厳格さのオプションの設定はできます。共有もできますが、URL のクエリーにすべてのソースコードを詰め込むストロングスタイルです。

28.1.2 <https://typescript-play.js.org/>

- URL: <https://typescript-play.js.org/>

標準の Playground よりも少し高機能な Playground です。TypeScript のコンパイラのバージョンを切り替えたり、標準よりも多くのオプションが設定できます。特に、出力先ターゲットの変更は検証には便利です。

共有ボタンはありませんが、入力するたびに URL が変更され、共有が行えます。

28.2 ビルド補助ツール

28.2.1 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

28.2.2 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。

28.2.3 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？

28.2.4 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。

28.2.5 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。

28.3 コマンドラインツール用ライブラリ

28.3.1 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。

28.3.2 @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。

28.4 アルゴリズム関連のライブラリ

28.4.1 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 29 章

貢献者

29.1 Pull Request をくださった方々

- @called-d
- @tkihira
- @kaakaa
- @uult
- @isdh
- @t0yohei
- @shrkw
- @atilol
- @numb86
- @aaaaayako
- @WATANAPEI
- @kensuke
- @ebiiim
- @yinm
- @hidaruma
- @7ma7X
- @isuzuki

- @taiyoooooooooooo
- @yaegassy
- @tmitz
- @ichikawa-hiroki

29.2 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/shibukawa/books/typescript-guide/advance.rst の 4 行目です)

課題: ESLint の TypeScript 対応はまだ開発途上で厳しそうなので TSLint に書き戻す？

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/baseenv.rst の 9 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/shibukawa/books/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/shibukawa/books/typescript-guide/baseenv.rst の 214 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/shibukawa/books/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 Docker イメージ

<https://qiita.com/kannkyo/items/5195069c65350b60edd9>

<https://qiita.com/shibukawa/items/fd49f98736045789ffc3#%E3%83%95%E3%83%AD%E3%83%B3%E3%83%88%E3%82%A8%E3%83%B3%E3%83%89%E5%91%A8%E3%82%8A%E3%81%AEdocker%E8%A8%AD%E5%AE%9A>

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/deploy.rst の 4 行目です)

課題: あとで書く

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/ecosystem.rst の 41 行目です)

課題: // browser/module など// <https://qiita.com/shinout/items/4c9854b00977883e0668>

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/libenv.rst の 131 行目です)

課題: 要検証

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/module.rst の 188 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/shibukawa/books/typescript-guide/module.rst の 311 行目です)

課題: あとで、開発サーバーとかもろもろについて語る

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/prodenv.rst の 66 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/shibukawa/books/typescript-guide/recommended.rst の 69 行目です)

課題: CommonJS のモジュールの型づけの仕方について紹介する

TypeScript の書き方 <https://qiita.com/karak/items/b7af3cb2843c39fb3949>

// JSDoc から型定義ファイルを抽出 <https://www.npmjs.com/package/tsd-jsdoc>

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/typedef.rst の 9 行目です)

課題: 事例をつける

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/typing.rst の 48 行目です)

課題: クラスベースの Vue のプロジェクトの作成について説明する

(元のエントリ は、 /Users/shibukawa/books/typescript-guide/vuesample.rst の 12 行目です)

第 30 章

Indices and tables

- `genindex`
- `modindex`
- `search`