



目次

はじめに iii



TypeScript とは何か

1



1-1 なぜ TypeScript が注目されているのか 2

1-2 TypeScript が開発された背景 3

1-3 TypeScript で生産性が上がる理由 4

1-4 TypeScript+JavaScript という二重構造 7

1-5 TypeScript のエコシステム 7



TypeScript コンパイラの基礎

9

2-1	node コマンドをインストールする	10
	COLUMN Node.js とスラウザ以外の JavaScript ランタイム	11
	Windows	12
	macOS - Homebrew	12
2-2	tsc コマンドをインストールする	13
	プロジェクトディレクトリを用意する	13
	tsc コマンドをプロジェクトにインストールする	14
	tsc --init で tsconfig.json を生成する	15
2-3	tsc コマンドで TypeScript のコードを コンパイルする	16
2-4	tsimp コマンドで TypeScript のコードを コンパイルせずに実行する	17
2-5	tsconfig.json について知っておくべきこと	18
	compilerOptions.target	
	- コンパイル先の ES バージョン	18
	compilerOptions.lib - 標準ライブラリの型定義	19
	COLUMN ダウンレベルとポリフィル	20
2-6	Visual Studio Code で TypeScript 言語サービスを利用する	21
2-7	Visual Studio Code から スクリプトを実行できるようにする	22
	COLUMN JSON と JSONC の違い	23
2-8	本書のサンプルコードについて	25




第 3 章

ES2015+ の基本構文

27



- 3-1 変数宣言 28
 - var による変数宣言の復習 28
 - ブロックスコープの変数宣言 let 29
 - ブロックスコープ宣言 const 30
 - TypeScript における declare var 宣言 31
- 3-2 クラス 32
 - クラスの継承 33
 - 第一級オブジェクトとしてのクラス 34
- 3-3 文字列 35
 - JavaScript の文字列リテラルの復習 35
 - テンプレート文字列 36
 - タグ付きテンプレート 36
 - TypeScript における文字列リテラル型 38
- 3-4 プリミティブ値 40
 - 多倍長整数 / bigint 40
 - シンボル / symbol 42
- 3-5 配列とタプル 43
 - コンストラクタ代替メソッド 44
 - タプル 45
 - 配列 46



3-6	オブジェクト	47
	オブジェクトリテラル.....	47
	オブジェクトリテラル型.....	49
	プロパティ速記法.....	50
3-7	グローバルオブジェクト	51
3-8	関数とメソッド	51
	アロー演算子による無名関数.....	52
	ジェネレータ関数.....	53
	デフォルト引数.....	54
	可変長引数.....	56
3-9	スプレッド構文	56
	関数の引数に対するスプレッド構文.....	57
	配列リテラルに対するスプレッド構文 [...a].....	57
	オブジェクトリテラルに対するスプレッド構文 {...o}.....	58
3-10	分割代入	59
3-11	条件分岐	61
	if文.....	61
3-12	for-ofループ文とイテレータ	62
3-13	async/awaitによる非同期処理	64




型演算の基本

67



4-1	JavaScriptの動的型の概要	68
4-2	TypeScriptの静的型の概要	69
	構造型 / Structural Typing	69
	公称型 / Nominal Typing	71
	漸近的型付け / Gradual Typing	72
4-3	any 型	73
4-4	unknown 型	75
4-5	void 型	77
4-6	never 型	77
4-7	オブジェクト型	78
	オブジェクトリテラ的な構文による型定義	79
	interface 宣言による型定義	79
	オブジェクト型におけるメソッドの定義	80
4-8	クラス型	81
	implements 句	82
4-9	型を引数として受け取るジェネリクス	83
	コンストラクタシグネチャ	85



4-10 共用体型 / Union Types	86
4-11 交差型 / Intersection Types	87
4-12 余剰プロパティチェック / Excess Property Checks	87
4-13 ナローイングと型ガード	88
4-14 型アサーションの as 演算子	92
COLUMN 暗黙の型アサーション	95
4-15 as const 演算子	96
4-16 non-null アサーション演算子	98
4-17 ユーザー値技の型ガードを実装する 述語関数	99
4-18 ナローイングを起こすためのアサーション関数	101
4-19 satisfies 演算子	102
implements 句と satisfies 演算子の比較	103

5-1	型関数と型演算子	106
	型関数のためのユーティリティ	106
	条件付き型 / Conditional Types	108
	条件付き型の分配 / Distributive Conditional Types	109
	T[P] - 型のプロパティ参照	111
	infer 型演算子	113
5-2	共用体型と交差型	113
5-3	テンプレートリテラル型	115
	テンプレート文字列に従って一定の規則を持つ文字列型を作る	116
	テンプレートリテラル型とinfer 型演算子で 文字列リテラル型の中身を解析する	117
5-4	組み込み型関数	118
	Record<KeyType, ValueType>	
	- 連想配列として使うオブジェクト型を生成する	118
	ReturnType<Fn> - 関数の戻り値を取り出す	119
	Pick<T, K> - オブジェクト型から 一部のプロパティを取り出す	120
	Omit<T, K> - オブジェクト型から一部のプロパティを除外する	121
	Partial<T> - オブジェクト型のプロパティをすべて省略可能にする	122
5-5	型演算活用事例 - ルーティングパスの文字列型から パラメータを取り出す型関数 ParamsOf<S>	124

6-1	import で拡張子なし	128
6-2	import で拡張子に .mjs	129
6-3	import で拡張子に .mts	130

この章では、TypeScriptコンパイラおよびその主要な実装であるtscコマンドについて、基本的な使い方を解説します。実際のWebフロントエンドやバックエンドの開発ではtscを直接呼び出すことはまれですが、それでもtscはTypeScriptで開発するときの基本です。tscがどのようなソフトウェアなのかは、知っておいたほうがよいでしょう。

ところで「TypeScriptコンパイラ」と「tsc」は必ずしも同じ意味ではありません。現在のところTypeScriptコンパイラはいくつか異なる実装があります。

tscはTypeScriptチームによる代表的な実装です。参照実装としての側面もあるため、TypeScriptの仕様はtscが実装しているものと考えてよいでしょう。しかしそれ以外にも、BabelのTypeScriptプラグインやesbuildもTypeScriptをコンパイルできます。これらはtscのすべての機能を実装しているわけではありません。tscと併用することが前提とされていますが、それでも十分に実用的です。

ただし実際には、TypeScriptコンパイラはtscとほぼ同じ意味で使われることも多いです。本書ではtscそのものについて触れるとき以外は「TypeScriptコンパイラ」で用語を統一しています。



2-1

node コマンドを インストールする



TypeScriptコンパイラの準備をする前に、コマンドラインのJavaScript処理系であるNode.jsが必要です。Node.jsのインストール方法はいくつかあります。

まず、公式サイト^{注1)}からビルド済みパッケージをダウンロードして手動でインストールするパターンです。この方法はOSによらず簡単にそのときの最新バージョンをインストールできます。しかし、この方法で継続的に最新バージョンをインストールするのは手間がかかります。

次に、OSごとの汎用的なパッケージマネージャで入れる方法があります。macOSであればHomebrew、Windows (WSL) であればaptやLinuxbrewなどです。HomebrewやLinuxbrewであれば最新版を入れられますし、その後継続して新しいバージョンに追従するのも簡単です。aptで入るNode (nodejsパッ

注1) <https://nodejs.org/en/download/>

ページ)のバージョンはディストリビューションにもよりますが、少し古いことが多いでしょう。

また、`nvm`や`nodebrew`のようなNode.js専用のバージョンマネージャを使うこともできます。このような専用のバージョンマネージャを使うと、複数バージョンのNode.jsをインストールしてプロジェクトごとに使い分けることができます。

ところでNode.jsのメジャーバージョンは、開発版である「Current」バージョンと安定版である「LTS (Long Term Support : 長期サポート)」バージョンがあります。Currentはその時点での開発版でいずれ安定版になる予定のバージョンですが、処理系自体が不安定なことがあります。実際の運用ではLTSを使い、Currentはテストで使うにとどめるのがよいでしょう。ただし、macOSのパッケージマネージャであるHomebrewではデフォルトでCurrentの最新版が入るようです。本書で扱う範囲においてはCurrentでも問題ありません。

またいずれの場合もNode.js用のパッケージマネージャである`npm`コマンドも同時にインストールされます。`npm`はもともとNode.js用プロジェクトのパッケージマネージャですが、現在はWebフロントエンド用のライブラリも管理できます。



Node.jsとブラウザ以外のJavaScriptランタイム



Node.js以外のJavaScriptランタイムもここで少し紹介します。「JavaScriptランタイム」というとき、純粋なJavaScript処理系に加えて何らかの実行環境やライブラリが付属しています。JavaScriptランタイムは大きく分けると次の3種類あります。

- ブラウザ用
- コマンドラインアプリケーション用
- エッジコンピューティング用

ブラウザ用のランタイムは、ブラウザに搭載されており、Webサイトを操作するためのランタイムです。もっとも一般的なJavaScriptランタイムともいえます。典型的には、純粋なJavaScript処理系に加えて、Web API (ブラウザAPI) を備えています。ブラウザ用のランタイムを想定したアプリケーションを「Webフロントエンド」と呼ぶこともあります。「Webフロントエンド用のライブラリ」は、ブ

本章では、ES2015+の基本構文を、TypeScriptで使うという文脈で解説します。ESとはECMAScriptの略で、ECMAScriptはJavaScriptの標準規格であるECMA-262で定義される言語の正式名称です。本書では、ECMA-262第6版^{注1)}で定義される「ECMAScript 2015」を「ES2015」、そしてES2015を含むそれ以降の版のESを「ES2015+」と総称しています。本書執筆時点では最新のESはES2024なので、「ES2015+」とは、ここではES2015からES2024までのESすべてという意味です。

なお「ES2015+」としてES2015を特別扱いするのは、ES2015は長いESの歴史の中で特別なバージョンだからです。その前のメジャーバージョンであるECMA-262第5版(ES5)は2009年に策定されたのですが、ES2015はそれから紆余曲折を経て6年越しのメジャーアップデートでした。そしてES2015でクラスやESモジュール、テンプレート文字列など、言語仕様へ追加された新機能が非常に多かったのです。そのあと、ESは毎年バージョンアップされる体制に変化し、ES2015以降も多くの機能が追加されていますが、依然として「新しいJavaScript」として「ES2015+」という総称が使われています。

TypeScriptはES2015+との互換性を維持したまま発展してきました。したがって、TypeScriptを使うためには、ES2015+を学ぶ必要があります。本章はES2015以降のESの新機能、つまりES2015+について、TypeScriptで使うという文脈のもと解説します。

3-1 変数宣言

それでは、ES2015+で導入された新機能を、TypeScriptとの関係を絡めて解説していきます。まず、コードの全域に影響を与える変数宣言です。なお、本章はJavaScriptの機能の説明ですが、サンプルコードはとくに断りのない限りTypeScriptで書いています。



varによる変数宣言の復習

ES2015より前のJavaScriptでは、次のようにvarが唯一の変数宣言でした。

注1) <http://www.ecma-international.org/ecma-262/6.0/>

var.mts

```
var x = "Hello, world!";  
console.log(x); // => Hello, world!
```

`var`は関数の中のどこで宣言しようとも有効範囲が関数全体となり、また変数を複数回宣言してもただ一度だけ宣言したのと変わりません。そしてこの仕様は多くの誤解とバグを生み出すことになりました。

たとえば、次のようなコードは文法的には有効です。この関数 `f()` では複数回 `var x` を宣言していますが、2度目に宣言しても何も効果はありません。その結果、`x`には `"second"` が代入されます。変数宣言をしたのに何も効果がないというのは、わかりにくい仕様だと思います。

var-file-scope.mts

```
function f() {  
  var x = "first";  
  if (true) {  
    var x = "second";  
  }  
  console.log(x); // => second  
}  
f();
```

現代において `var` を使わなければならない状況はありませんが、`var` を使っている古いコードはまだあるため、プロジェクトによっては読む必要はあるかもしれません。しかし、本書で `var` が出てくるのはこの節だけです。



ブロックスコープの変数宣言 `let`

一方ES2015+では、コードブロックごとに新しい変数定義が作られる `let` と `const` を使います。コードブロックごとに定義が作られるため、ブロックスコープの変数宣言と呼ばれます。`let` は再代入可能で、`const` は再代入不可能な変数宣言です。

まず、先ほどの `var` を使ったコードを `let` に置き換えてみると、`"first"` が印字されます。`const` でも結果は同じです。

let.mts

```
function f() {  
  let x = "first";  
  if (true) {
```

