

Web講習会2021 ワールドワイドウェブ発展

第3回: TypeScript



Arthur

この回の目標

- AltJSの存在とその意義について知る
- TypeScriptによる基本的なプログラミングを体験する

本講習会に期待してほしいこと

- 高級な型システムに関する親切な説明

TypeScript

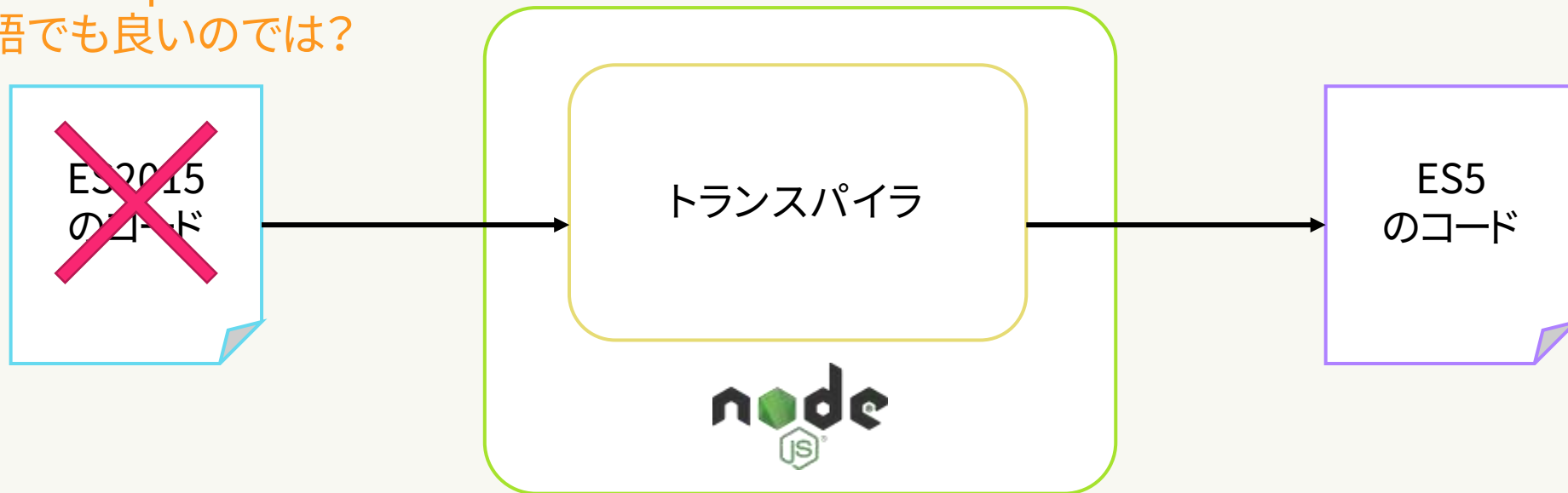


JavaScriptの問題点

JavaScriptはモダンに進化しつつある言語ではあるが...

- エラーをランタイムでしか発見できない(動的型付け)
- 型安全にする機能がない
- (オブジェクト志向における)インタフェースがない

JavaScript以外の
言語でも良いのでは?

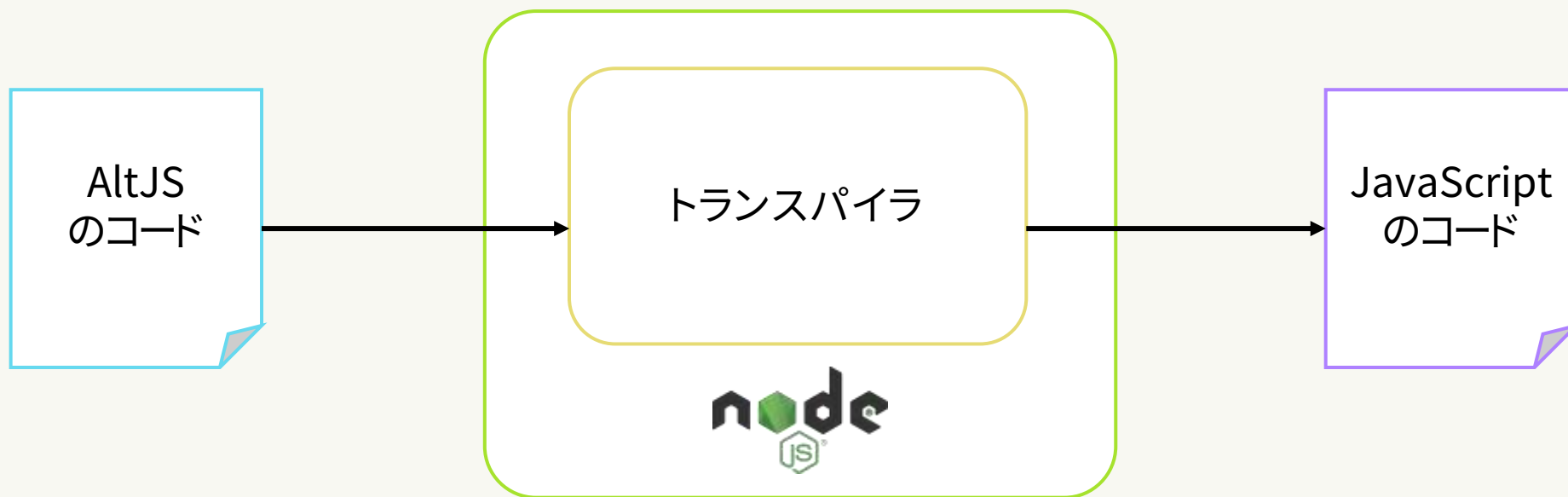


AltJS

AltJS

JavaScriptのアプリケーションを開発するためのより高度な言語
AltJSで書かれたプログラムをトランスパイルしてJavaScriptを生成

※AltJS → JavaScriptへの変換を、一般にはコンパイルと呼ぶので、以降のスライドでは「コンパイル」表記を使用することがある



TypeScript

TypeScript

AltJSの中で最も一般に使われている言語

型の定義やチェックにより保守性や、補完による効率アップが目指せる

ES2015のスーパーセットであることが特徴

npm(Yarn)でインストールできるパッケージの多くがTypeScriptの型ファイルを内包している

型ファイルによる補完の恩恵を受けられる!



本日の説明内容

TypeScriptは、以下両方の実行環境をターゲットに利用できる

- ブラウザのランタイム
- Node.jsのランタイム

本日は、Node.js上で動くプログラムを生成するTypeScriptについて学ぶ

TypeScript入門



導入

ローカルにTypeScriptをコンパイルするためのパッケージを導入

1. 適当なディレクトリを作成
2. Node.jsのバージョンを指定
\$ nodenv local 16.4.0
3. package.jsonを生成
\$ yarn init
4. typescript, @types/node@16を導入
\$ yarn add -D typescript @types/node@16
5. tscコマンドが動くか確認
\$ npx tsc --version

```
~/Documents/titechapp/pr5 >>> nodenv local 14.16.0
~/Documents/titechapp/pr5 >>> yarn init
yarn init v1.22.10
question name (pr5):
question version (1.0.0):
question description:
question entry point (index.js):
question repository url:
question author:
question license (MIT):
question private:
success Saved package.json
🌟 Done in 3.78s.
~/Documents/titechapp/pr5 >>> yarn add -D typescript @types/node@14
yarn add v1.22.10
info No lockfile found.
[1/4] 🔍 Resolving packages...
[2/4] 📦 Fetching packages...
[3/4] 🔗 Linking dependencies...
[4/4] 🏗 Building fresh packages...
success Saved lockfile.
success Saved 2 new dependencies.
info Direct dependencies
├─ @types/node@14.14.34
└─ typescript@4.2.3
info All dependencies
├─ @types/node@14.14.34
└─ typescript@4.2.3
🌟 Done in 1.95s.
~/Documents/titechapp/pr5 >>> npx tsc --version
Version 4.2.3
```

[補足] npx

npmでインストールしたパッケージにコマンドが含まれている場合がある

ex) yarn, tsc...

ローカルインストール(通常)した場合、パスが通らず実行できない

npx **[コマンド]** を利用することにより、ローカルでも利用可能
もしくは、package.jsonにscriptsを定義する

<https://ginpen.com/2019/12/04/npm-scripts-not-needs-npx-nor-node-modules/>

TypeScriptの設定

設定ファイルを生成

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

```
~/Documents/titechapp/pr5 >>> npx tsc --init  
message TS6071: Successfully created a tsconfig.json file.
```

設定ファイルを編集

"target": "es5", → "es2020",

```
{  
  "compilerOptions": {  
    /* Visit https://aka.ms/tsconfig.json to read more about this file */  
  
    /* Basic Options */  
    // "incremental": true, /* Enable incremental compilation */  
    "target": "es2020", /* Specify ECMAScript target version: 'es5' (default), 'es6', 'es2015', 'es2016', 'es2017', 'es2018', 'es2019', 'es2020', 'esnext'. */  
    "module": "commonjs", /* Specify module code generation: 'none', 'commonjs', 'amd', 'system', 'umd', 'es2015', 'es2020', 'esnext'. */  
    // "lib": [], /* Specify library files to be included in the compilation. */  
    // "allowJs": true, /* Allow javascript files to be compiled. */  
    // "checkJs": true, /* Report errors in .js files. */  
    // "jsx": "preserve", /* Specify JSX code generation: 'preserve', 'react-native', 'react'. */  
    // "declaration": true, /* Generates .d.ts files. */  
    // "declarationMap": true, /* Generates .d.ts.map files. */  
  }  
}
```

TS→JSのコンパイル

右のようなTSファイルをJSにコンパイルして実行する

`npx tsc` : コンパイル (同名の.jsファイルを生成)

```
~/Documents/titechapp/pr5 >>> cat helloworld.ts
namespace helloworld {
  let message: string
  message = 'Hello, World!'

  console.log(message)
}
~/Documents/titechapp/pr5 >>> npx tsc
~/Documents/titechapp/pr5 >>> cat helloworld.js
"use strict";
var helloworld;
(function (helloworld) {
  let message;
  message = 'Hello, World!';
  console.log(message);
})(helloworld || (helloworld = {}));
~/Documents/titechapp/pr5 >>> node helloworld.js
Hello, World!
```

※tsc実行時に、一番外のスコープはファイルが違っても同一視されるため、namespace利用

helloworld.ts

```
namespace helloworld {
  let message: string
  message = 'Hello, World!'

  console.log(message)
}
```

型チェック

string型として宣言した変数にnumber型を入れると…
コンパイルエラーが発生

```
let message: string
message = 'Hello, World!'
message = 0

console.log(message)
```

さあ皆さんと一緒に、

- コンパイルエラーは普通
- コンパイルエラーが出たらありがとう
- コンパイルエラーが出たら大喜び

<https://cpp.rainy.me/004-debug-compile-error.html>

```
~/Documents/titechapp/pr5 >>> npx tsc
helloworld.ts:4:1 - error TS2322: Type 'number' is not assignable to type 'string'

4 message = 0
  ~~~~~

Found 1 error.
```

型推論

宣言時に値を初期化すると、変数はその型になる

型推論により、明示的な型の宣言が不要

```
let message = 'Hello, World!'  
message = 0  
  
console.log(message)
```

初期化時の左辺が文字列リテラルだから、
messageはstring型だ！

tscくん

```
~/Documents/titechapp/pr5 >>> npx tsc  
helloworld.ts:2:1 - error TS2322: Type 'number' is not assignable to type 'string'.  
2 message = 0  
  ~~~~~  
  
Found 1 error.
```

関数の場合の例

引数や返却値でも型宣言ができる

add.ts

```
const add = (a: number, b: number): number => {  
  return a + b  
}
```

```
let result: number = add(3, 4)  
console.log(result)
```

```
const add = (a: number, b: number) => {  
  return a + b  
}
```

```
let result: number = add(3, 4)  
console.log(result)
```

```
~/Documents/titechapp/pr5 >>> npx tsc  
~/Documents/titechapp/pr5 >>> node add.js  
7
```

型推論を利用して返却値の型宣言を省略
number + numberはnumber

TypeScriptと型の種類

プリミティブ型

これまで扱って来た型は**プリミティブ型** (primitive data type)
それ以上分解できないようなシンプルなデータ型

TypeScriptにおいて、**プリミティブ型は小文字から始まる**
JavaScriptではプリミティブ型も大文字から始まるものがあった

	JavaScript	TypeScript
真偽値	Boolean	boolean
整数	Number	number
文字列	String	string

他にも、nullやundefinedなど

配列型

配列の要素の型を宣言することが可能

fruits.ts

```
const fruits: string[] = []  
  
fruits.push('apple')  
fruits.push('orange')  
// fruits.push(4)  
  
console.log(fruits)
```

コメントアウトを外すと…

```
~/Documents/titechapp/pr5 >>> npx tsc  
fruits.ts:6:17 - error TS2345: Argument of type 'number' is not assignable to parameter of  
type 'string'.  
  
6     fruits.push(4)  
           ~~~~~  
  
Found 1 error.
```

Generics

Generics

型を抽象化し、特定の型に依存しないように実装できる仕組み

型引数を<>で囲んで渡す

JavaのGenericsやC++のTemplateと同様

先程の配列もArray型+Genericsで宣言可能

```
const fruits: Array<string> = []
```

interface

interface

JavaScriptのオブジェクトに対する型
オブジェクトが持つプロパティと、それぞれの型を定義する

person.ts

```
interface Person {  
  name: string  
  age: number  
  greets(): void  
}  
  
interface Student extends Person {  
  id: string  
}
```

```
const arthur: Student = {  
  name: 'Arthur',  
  age: 24,  
  id: '15_12345',  
  greets() {  
    console.log(`Hello. I'm ${this.name}.`)  
  }  
}  
  
arthur.greets()
```

union型

union型

複数の型のうちどれかとマッチするような型

|で連ねてORを表現

union.ts

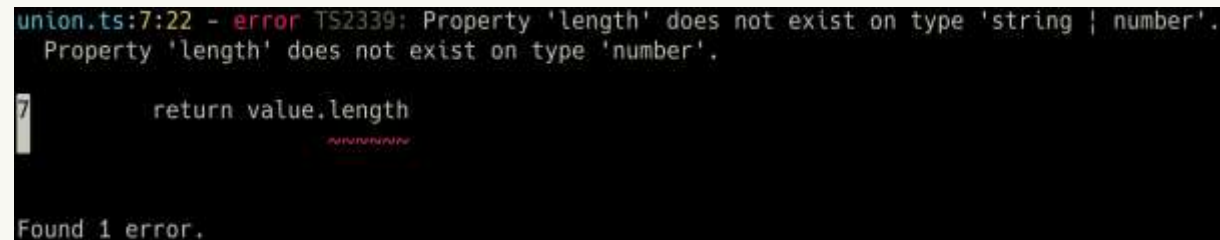
```
const value: string | number = 'hello'

console.log(value)
/*
const getLength = (value: string | number): number => {
  return value.length
}
console.log(getLength(value))
*/
```

コメントアウトを
外すとどうなる？

union型のプロパティ検査

コメントアウトを外すとコンパイルエラー
string型の場合はlengthプロパティがある
number型の場合はない!



The screenshot shows a TypeScript compiler error in a dark-themed editor. The error message is: "union.ts:7:22 - error TS2339: Property 'length' does not exist on type 'string | number'. Property 'length' does not exist on type 'number'." Below the message, the code snippet "return value.length" is visible with a red squiggly line under "length". At the bottom, it says "Found 1 error.".

```
const getLength = (value: string | number): number => {  
  if (typeof value === 'string') {  
    return value.length  
  }  
  return 0  
}  
console.log(getLength(value)) // 0
```

if文で型によって場合分けすることにより
lengthがあるときだけアクセスするように

interfaceとunion型

union型でinterfaceを列挙し、後に場合分けの処理を行う場合、
適当な文字列リテラルをプロパティとしてinterfaceに定義するとよい

```
interface Square {  
  type: 'Square'  
  size: number  
}  
  
interface Rectangle {  
  type: 'Rectangle'  
  width: number  
  height: number  
}  
  
interface Circle {  
  type: 'Circle'  
  radius: number  
}  
  
type Shape = Square | Rectangle | Circle
```

```
const getArea = (s: Shape) => {  
  if (s.type === 'Square') {  
    return s.size * s.size  
  } else if (s.type === 'Rectangle') {  
    return s.width * s.height  
  }  
}  
  
const rect: Rectangle = {  
  type: 'Rectangle',  
  width: 4,  
  height: 3,  
}  
  
console.log(getArea(rect))
```

shape.ts

網羅チェックの必要性

先程のコードで円の面積を計算すると…

```
const getArea = (s: Shape) => {  
  if (s.type === 'Square') {  
    return s.size * s.size  
  } else if (s.type === 'Rectangle') {  
    return s.width * s.height  
  }  
}  
  
const circle: Circle = {  
  type: 'Circle',  
  radius: 1,  
}  
  
console.log(getArea(circle))
```

```
~/Documents/titechapp/pr5 >>> npx tsc  
~/Documents/titechapp/pr5 >>> node shape.js  
undefined
```

コンパイルエラーにはならないがundefinedが返る

getArea()にCircleの場合の処理がない
ランタイムでバグ発見はTypeScriptの意義が…

網羅チェックの追加

union型として受け取った変数に対して、個別の型マッチングさせる処理を行う場合、引っかけからなかったらnever型の変数に代入させる
こうすると、網羅できているかをコンパイルエラーにより知ることができる

※Linterが不要な代入として警告してくる場合があるので、変数名を_からはじめ、かつ_から始まる変数を無視するようにする

```
const getArea = (s: Shape) => {  
  if (s.type === 'Square') {  
    return s.size * s.size  
  } else if (s.type === 'Rectangle') {  
    return s.width * s.height  
  } else {  
    const _err: never = s  
  }  
}
```

```
~/Documents/titechapp/pr5 >>> npx tsc  
shape.ts:23:19 - error TS2322: Type 'Circle' is not assignable to type 'never'.  
  
23         const _err: never = s  
                        ~~~~~  
  
Found 1 error.
```

網羅するようにコード修正

Circleの場合分けを追加すると、コンパイルエラーが起きなくなる

```
const getArea = (s: Shape) => {  
  if (s.type === 'Square') {  
    return s.size * s.size  
  } else if (s.type === 'Rectangle') {  
    return s.width * s.height  
  } else if (s.type === 'Circle') {  
    return s.radius * s.radius * 3  
  } else {  
    const _err: never = s  
  }  
}
```

```
~/Documents/titechapp/pr5 >>> npx tsc  
~/Documents/titechapp/pr5 >>> node shape.js  
3
```

コンパイルエラーを起こさない方向に逃げない!

コンパイルエラーは、実行してはじめて分かる
エラーを減らすために存在する!

型定義ファイル

any型を使う？

TypeScriptから、素のJavaScriptで作られたライブラリを利用するとき、型はどうなるのだろうか？

any型: なんでもOKな型 (TypeScriptの型システムを無視)

情報量の少ない型から多い型への遷移は基本的にできない

型情報のないライブラリから得た値はすべて型がなくなってしまう

世の中のすべての人がTypeScriptを使っているわけではない

- Node.jsのビルトインモジュール
- jQuery

型定義ファイル

型定義ファイル (**.d.ts)

JavaScriptで開発されたライブラリに後付けで型情報を与える

TypeScriptプログラムからJavaScriptライブラリを呼び出すときに利用可能

ライブラリに型定義ファイルが含まれていない場合、個別に導入

```
yarn add -D @types/hoge
```

p.9の@types/node@14はNode.js v14のビルトインモジュールの型定義ファイル

型定義ファイルが提供されていない場合は自作できる

型定義ファイルの利用

Node.jsのビルトインモジュールであるfs (非TypeScript) でも型情報を利用できることが確認できる



```
TS: readFile.ts > {} readFile > [0] main > [0] result
1  import fs from 'fs/promises'
2
3  namespace readFile {
4      const main = async () => {
5          const result = await fs.readFile('./.node-version',
6              console.log(result)
7          )
8          main()
9      }
10 }
```

function readFile(path: PathLike | fs.FileHandle, options: { encoding: BufferEncoding; flag?: OpenMode | undefined; } | BufferEncoding): Promise<...> (+2 overloads)

Asynchronously reads the entire contents of a file.

@param path

次回予告

フロントエンドから少し離れてサーバサイドよりの話をします

WebAPIを利用する立場として必要な知識