TypeScript 勉強会

リードエンジニア 鈴木 慎太郎

1. TypeScript の概要

1.1. 型安全性

1.2. コンパイル

1.1. 型安全性

JavaScript のコード

```
3 + [];
// 3

let obj = {};
obj.foo;
// undefined
```

JavaScript はできるだけエラーを出さずに、コードを活かそうとするため、動作確認をしてみるまでバグに気づくことができません。

1.1. 型安全性

TypeScript のコード

```
3 + [];
// Error: Operator '+' cannot be applied to types 'number' and 'never[]'.

let obj = {};
obj.foo;
// Error: Property 'foo' does not exist on type '{}'.
```

TypeScript は「型」を用いて、不正なコードを実行せずとも "事前に" 教えてくれます。

1.2. コンパイル

JavaScript のコンパイル

1. JavaScript ソース → JavaScript AST
2. JavaScript AST → バイトコード
3. バイトコードがランタイムによって評価される

1.2. コンパイル

TypeScript のコンパイル

1. TypeScript ソース → TypeScript AST
2. **AST が型チェッカーによりチェックされる**
3. TypeScript AST → JavaScript ソース

TypeScript のコンパイルには *tsc (TypeScript Compiler)* を用いる。

```
$ tsc src
```

1.2. コンパイル

Q.型チェックはいつ実行されるか？

A.型チェックはコンパイル時に行われるが、TypeScript にはインクリメンタルコンパイル (コードの変更と同時に再コンパイルが行われる仕組み) が備わっているため、実際には常に同期的に型チェックは行われており、エディタ上でリアルタイムでエラーを確認することができる。

2. TypeScript の初歩

2.1. TypeScript の文法

TypeScript では変数の後ろに型を宣言します。

```
let num: number = 3;
num = "hoge";
// Error: Type 'string' is not assignable to type 'number'.
```

上記の例では、数値型として宣言した `num` に文字列型の `'hoge'` を代入しようとして怒られています。

2.2. 型推論

しかし、冒頭の例の通り、型を明示的に宣言しなくても TypeScript は仕事をしてくれます。

TypeScript はコードから変数の型を "ある程度" 推論することができるため、明示的に型を宣言しなくても、エラーを出力することができます。

```
3 + [];
// Error: Operator '+' cannot be applied to types 'number' and 'never[]'.
```

2.3. オブジェクト

オブジェクトの型宣言は、**オブジェクトリララル表記**というものを用います。

```
let obj: {
    a: number;
    b: string;
} = {
    a: 3,
    b: "x",
};
```

2.4. 配列

配列の型宣言は下記のように行います。

この場合、`arr` は配列の長さを問わず、空配列も許容します。

```
let arr: number[] = [0, 1, 2];
```

2.5. タプル

タプルは長さが固定の配列です。タプルの型宣言は次のように行います。

```
let tuple: [number, number] = [0, 1];
// OK

let tuple: [number, number] = [0, 1, 2];
// Error: Type '[number, number, number]' is not assignable to type '[number, number]'.
//        Source has 3 element(s) but target allows only 2.
```

タプルの中身は、可変長の要素もサポートしており、これを使うと最小の長さについて制約を持った配列を表現できます。

```
let NoEmptyNumber: [number, ...number[]] = [0];
// OK

let NoEmptyNumber: [number, ...number[]] = [];
// Error: Type '[]' is not assignable to type '[number, ...number[]]'.
//        Source has 0 element(s) but target requires 1.
```

2.6. 型エイリアス

型エイリアスを使うと、下記のように型に別名をつけることができます。

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

let tanaka: Person = {
    name: "田中",
    age: 30,
};
```

2.7. オプションパラメータ

前のスライドで `Person` という型エイリアスを定義しましたが、年齢を知られたくない人もいると思うので、`age` プロパティは任意としたいです。

このときに使えるのが **オプションパラメータ(`?`)** です。

```
type Person = {
    name: string;
    age?: number;
};

let tanaka: Person = {
    name: "田中",
    age: 30,
};

let nakano: Person = {
    name: "中野",
};
```

2.8 合併型

囚人番号で管理されている名もなき犯罪者のことも考慮すると、`Person` 型の `name` プロパティは数値も受け付けられるようにしたほうがいいかもしれません。

このような場合に使えるのが **合併型(|)** です。

```
type Person = {
    name: string | number;
    age?: number;
};

let tanaka: Person = {
    name: "田中",
    age: 30,
};

let criminal: Person = {
    name: 334,
};
```

2.9 交差型

`Person` 型にはまだ考慮漏れがあります。それは、`Person` でもあり `Wolf` でもある人狼のことを考慮していない点です。

交差型(`&`) を使って、人狼も考慮してあげましょう。

```
type Wolf = {
    bark: () => void;
};

let miyajima: Person & Wolf = {
    name: "宮島",
    age: 22,
    bark: () => console.log("Awoo"),
};
```

3. TypeScript の実践

3.1. ルックアップ型

OpenAPI.yml から生成した型 `Contract` があります。

このうち、`workHours` の要素の型のみを利用したい場合、どうしたらよいでしょう？

```
type Contract = {
    contractId: number;
    clientName: string;
    workHours: {
        shiftType: string;
        workStartTime: string;
        workEndTime: string;
    }[];
};
```

3.1. ルックアップ型

オブジェクトリテラル表記の型からプロパティを取り出したい場合は **ルックアップ型** が使えます。

オブジェクトのプロパティを取り出したいときは `Object["propaty"]` を、配列の要素を取り出したいときは `Array[number]` とすることで取得することができます。

```
type WorkHour = Contract["workHours"][number];

let workHour: WorkHour = {
    shiftType: "早番",
    workStartTime: "07:00",
    workEndTime: "15:00",
};
```

3.2. ジェネリック型

受け取った引数をそのまま返す `getArg()` という関数を考えます。~~実際にこんなコードがあったらつっぱねます。~~

このとき、① 引数の型と返り値の型は一致していなくてはなりませんが、② 引数にはどんな型もとることができます。

引数と返り値にはどのような型宣言をしたら、上記の制約を与えられるでしょうか？

3.2. ジェネリック型

はじめに考えられるのは `any` ですが、これは ② の制約は満たしていますが、① の制約は満たしていません。

```
const getArg: (arg: any) => any = (arg) => {
    return arg;
};

const a = getArg(3);

a + 1;
// OK 😐
```

3.2. ジェネリック型

このような場合に使えるのが **ジェネリック型** です。型引数 (ジェネリック型パラメータ) を用いて、型制約を提供することができます。

```
type GetArg<T> = (arg: T) => T;

const getArg: GetArg<number> = (arg) => {
    return arg;
};

const a: string = getArg(3);
// Error: Type 'number' is not assignable to type 'string'.
```

演習 1

空を許容しない配列を生成するジェネリック型 `NoEmptyArray<T>` を定義してください。

例)

```ts
const a: NoEmptyArray<number> = [1, 2, 3];
// OK

const b: NoEmptyArray<string> = [1, 2, 3];
// Error

const c: NoEmptyArray<string> = [];
// Error
```

演習 1 <解答>

```ts
type NoEmptyArray<T> = [T, ...T[]];

const a: NoEmptyArray<number> = [1, 2, 3];
// OK

const b: NoEmptyArray<string> = [1, 2, 3];
// Error: Type 'number' is not assignable to type 'string'.

const c: NoEmptyArray<string> = [];
// Error: Type '[]' is not assignable to type 'NoEmptyArray<string>'.
```

3.3. 制約つきのジェネリック型

演習 1 でつくった `NoEmptyArray<T>` を使って、変数を宣言してみます。

```
type Numbers = number[];

const nums: NoEmptyArray<Numbers> = [1, 2, 3];
// Error: Type 'number' is not assignable to type 'Numbers'.
```

上記の実装では、`NoEmptyArray<Numbers>` は `Numbers` の配列を要求しているので、下記のように修正しなければなりません。

```
const nums: NoEmptyArray<Numbers[number]> = [1, 2, 3];
```

3.3. 制約つきのジェネリック型

実用的には、配列を渡して配列を返すジェネリック型を作った方が直感的かもしれません。

`extends` 演算子を用いて作成してみます。

```
type NoEmpty<T extends unknown[]> = [T[number], ...T];

const a: NoEmpty<number[]> = [1, 2, 3];
// OK

const b: NoEmpty<string[]> = [];
// Error: Type '[]' is not assignable to type '[string, ...string[]]'.
//        Source has 0 element(s) but target requires 1.

const c: NoEmpty<string> = ["x", "y"];
// Error: Type 'string' does not satisfy the constraint 'unknown[]'.
```

3.3. 制約つきのジェネリック型

ジェネリック型パラメータを `T extends unknown[]` とすることで、`T` に対して `unknown[]` を満たしているという制約を追加しています。

このように、`extends` を用いると、ジェネリック型パラメータに制約をつけることができます。

3.4. 条件型

値レベルにおける三項式と同様のことを、型レベルでも行うことができます。

```ts
type WithoutNullish<T> = T extends null | undefined ? never : T;

type NullableString = string | null | undefined;

const a: NullableString = null;
// OK

const b: WithoutNullish<NullableString> = null;
// Error: Type 'null' is not assignable to type 'string'.
```

3.5. keyof 演算子

keyof 演算子 を使うと、`Contract` 型の例において、`Contract` 型のキーのユニオン型 (合併型) を得られます。

```
type ContractKey = keyof Contract;
// "contractId" | "clientName" | "workHours"

type ContractProperty<T extends ContractKey> = Contract[T];
```

3.6. ユーティリティ型

TypeScript には、ビルトインパッケージで定義されている便利なユーティリティ型があります。

主要なユーティリティ型のみを取り上げるので、他のユーティリティ型に興味があれば公式ドキュメントを参照してください。

https://www.typescriptlang.org/docs/handbook/utility-types.html

3.6. ユーティリティ型

Required<T>

すべてのプロパティを必須にします。

```
type Person = {
    name: string;
    age?: number;
};

let yamazaki: Required<Person> = {
    name: "山﨑",
};
// Error: Property 'age' is missing in type '{ name: string; }' but required in type 'Required<Person>'.
```

3.6. ユーティリティ型

Partial<T>

すべてのプロパティをオプショナルにします。

```
type Contract = {
    contractId: number;
    clientName: string;
    workHours: {
        shiftType: string;
        workStartTime: string;
        workEndTime: string;
    }[];
};

let EmptyContract: Partial<Contract> = {};
// OK 🤯
```

3.6. ユーティリティ型

Pick<T, K>

任意のプロパティだけを持つオブジェクトを取得します。

```
type Contract = {
    contractId: number;
    clientName: string;
    workHours: {
        shiftType: string;
        workStartTime: string;
        workEndTime: string;
    }[];
};

let contract: Pick<Contract, "contractId" | "clientName"> = {
    contractId: 440,
    clientName: "A",
    workHours: [],
};
// Error: Type '{ contractId: number; clientName: string; workHours: never[]; }' is not assignable to type 'Pick<Contract, "contractId" | "clientName">'.
//        Object literal may only specify known properties, and 'workHours' does not exist in type 'Pick<Contract, "contractId" | "clientName">'.
```

3.6. ユーティリティ型

Omit<T, K>

任意のプロパティを除外したオブジェクトを取得します。

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

let tanaka: Omit<Person, "age"> = {
    name: "田中",
    age: 30,
};
// Error: Type '{ name: string; age: number; }' is not assignable to type 'Omit<Person, "age">'.
//        Object literal may only specify known properties, and 'age' does not exist in type 'Omit<Person, "age">'.
```

3.6. ユーティリティ型

NonNullable<T>

null と undefined を型から除外します。

さっき定義した `WithoutNullish<T>` と同様です。

```
type NullableNumber = number | null | undefined;

const a: NonNullable<NullableNumber> = null;
// Error: Type 'null' is not assignable to type 'number'.
```

3.6. ユーティリティ型

ReturnType<T>

型引数に渡した関数型の返り値の型を取得します。

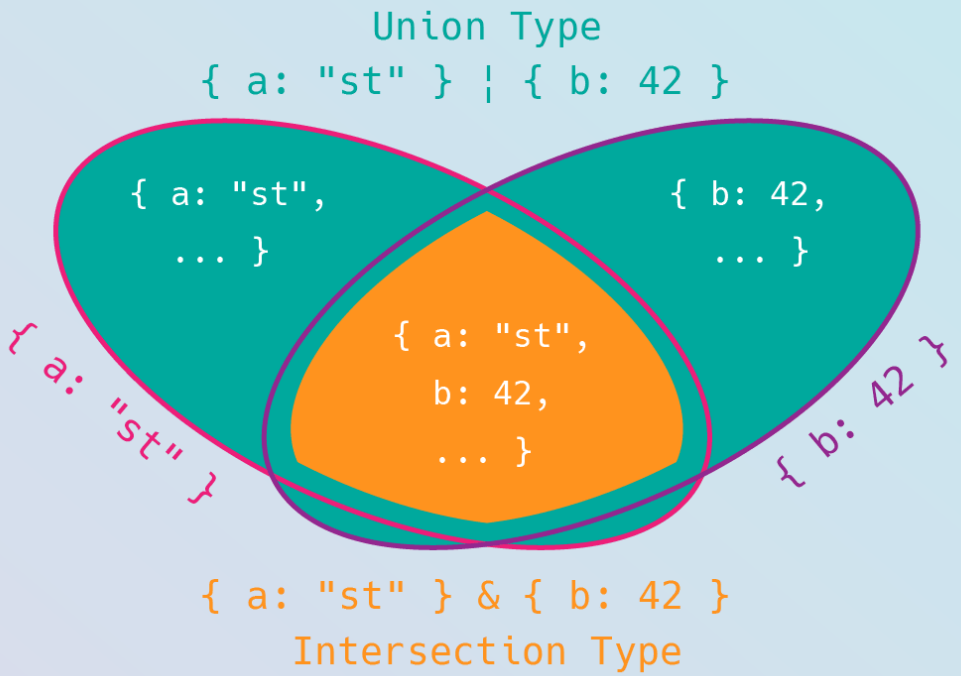
```
type GetString = () => string;

const getString: GetString = () => "x";

const a: ReturnType<GetString> = getString();
// OK 🤩
```

3.7. <発展> 集合としての型

TypeScript の型は本質的には値の集合です。`{ a: "st" }` は、『プロパティ a に "st" という値が入っているという条件を満たしたオブジェクトの集合』になります。

型 A と型 B の合併型 A | B は、集合 A と集合 B の和集合 A ∨ B と考えることができます。

A ∨ B はもちろん A と B を部分集合に持ちますが、A ∧ B もまた、A ∨ B の部分集合です。



3.7. <発展> 集合としての型

タッカーさんは娘のニーナと犬のアレキサンダーと暮らしているため、`MyFamily` という `Person` または `Dog` を指す型を作ります。

```
type Person = {
    speak: () => void;
};

type Dog = {
    bark: () => void;
};

type MyFamily = Person | Dog;
```

3.7. <発展> 集合としての型

この実装だと、人と犬のキメラを作れてしまいます。

```
const nina: MyFamily = {
    speak: () => speck(),
};

const alexander: MyFamily = {
    bark: () => bark(),
};

const chimera: MyFamily = {
    speak: () => speak(),
    bark: () => bark(),
};
// OK! 😮
```

3.7. <発展> 集合としての型

A ∧ B を空集合とすることで、これを回避できます。

人間ならば決して吠えない、犬ならば決してしゃべらないと定義することによって、人間であり犬であるという集合はなくなります。

```
type Person = {
    speak: () => void;
    bark?: never;
};

type Dog = {
    speak?: never;
    bark: () => void;
};

type MyFamily = Person | Dog;
```

3.7. <発展> 集合としての型

無事、ニーナとアレキサンダーは錬金術でキメラにされずに済みそうです。

```
const nina: MyFamily = {
    speak: () => speck(),
};

const alexander: MyFamily = {
    bark: () => bark(),
};

const chimera: MyFamily = {
    speak: () => speak(),
    bark: () => bark(),
};
// Error: Type '{ speak: () => void; bark: () => void; }' is not assignable to type 'MyFamily'.
//        Type '{ speak: () => void; bark: () => void; }' is not assignable to type 'Dog'.
//        Types of property 'speak' are incompatible.
//        Type '() => void' is not assignable to type 'undefined'.
```

3.7. <発展> 集合としての型

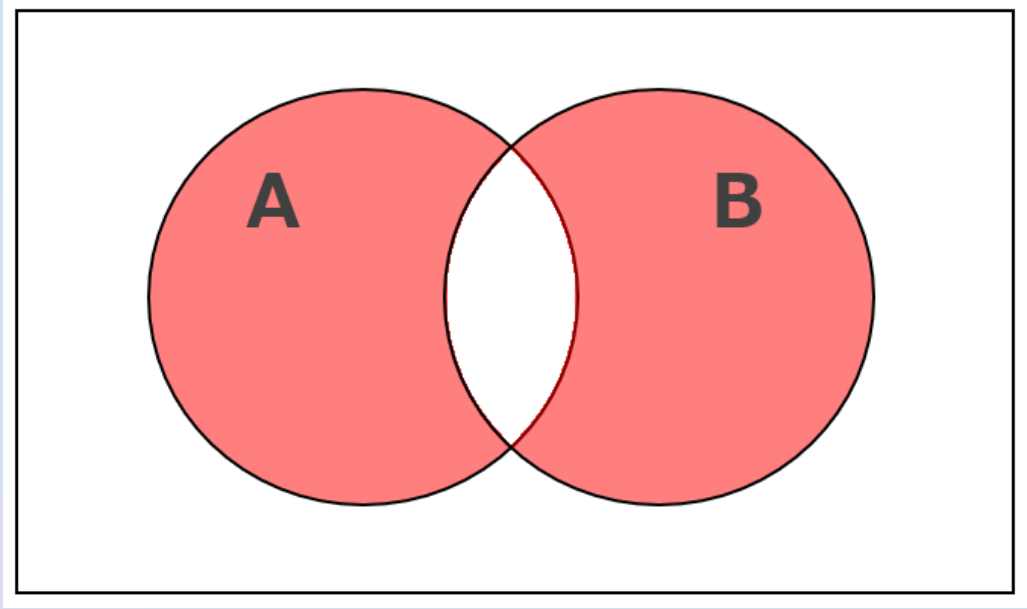
fetchAPI という fetch のラッパー関数を作ろうとしており、method が GET のときは body に値を渡したくありません。

この場合、次の実装でこれを実現できます。

```
type GetMethod = {
    method: "GET";
    body?: never;
};

type OtherMethod = {
    method: "POST" | "PUT";
    body?: object;
};

type FetchAPIProps = GetMethod | OtherMethod;
```

3.7. <発展> 集合としての型

XOR (右図) を表現する演算子が TypeScript に用意されていれば、オブジェクトどうしの合併型に関する問題に悩まされずに済むのですが、組み込みでそのような機能はありません。



3.7. <発展> 集合としての型

XOR を表現するジェネリック型を用意しました。

```
type Without<T, U> = { [K in Exclude<keyof T, keyof U>]?: never };

export type XOR<T, U> = T | U extends object
    ? (Without<T, U> & U) | (Without<U, T> & T)
    : T | U;
```

関連キーワード

- マップ型
- ユーティリティ型 (Exclude)

演習 2

ユーティリティ型を使って任意のプロパティのみをオプショナルにする型を作ってください。

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

let kimura: SomePartial<Person, "name" | "age"> = {};
// OK

let suzuki: SomePartial<Person, "age"> = {
    name: "鈴木",
};
// OK
```

演習 2 <解答>

制約つきジェネリック型、keyof 演算子、ユーティリティ型、交差型 を使った勉強会の最後を飾るにふさわしい問題だったと思います。

特に、`K` に制約をつけるということを忘れがちだと思いますが、extends 演算子もフル活用して TypeScript を用いた型安全な開発をしていきましょう。

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

type SomePartial<T, K extends keyof T> = Omit<T, K> & Partial<Pick<T, K>>;
```

補足

環境構築

TypeScript 環境のない方向けの、演習問題に取り組むための環境を用意する手順です。

実際のプロジェクトでは、プロジェクトごとに適切な環境を設定してください。

node の環境を構築

```
$ brew install nodenv
$ nodenv install 18.6.0
$ nodenv global 18.6.0
$ npm install -g yarn
```

環境構築

TypeScript のインストール

プロジェクトを作成し、`typescript` と `@types/node` をインストールします。

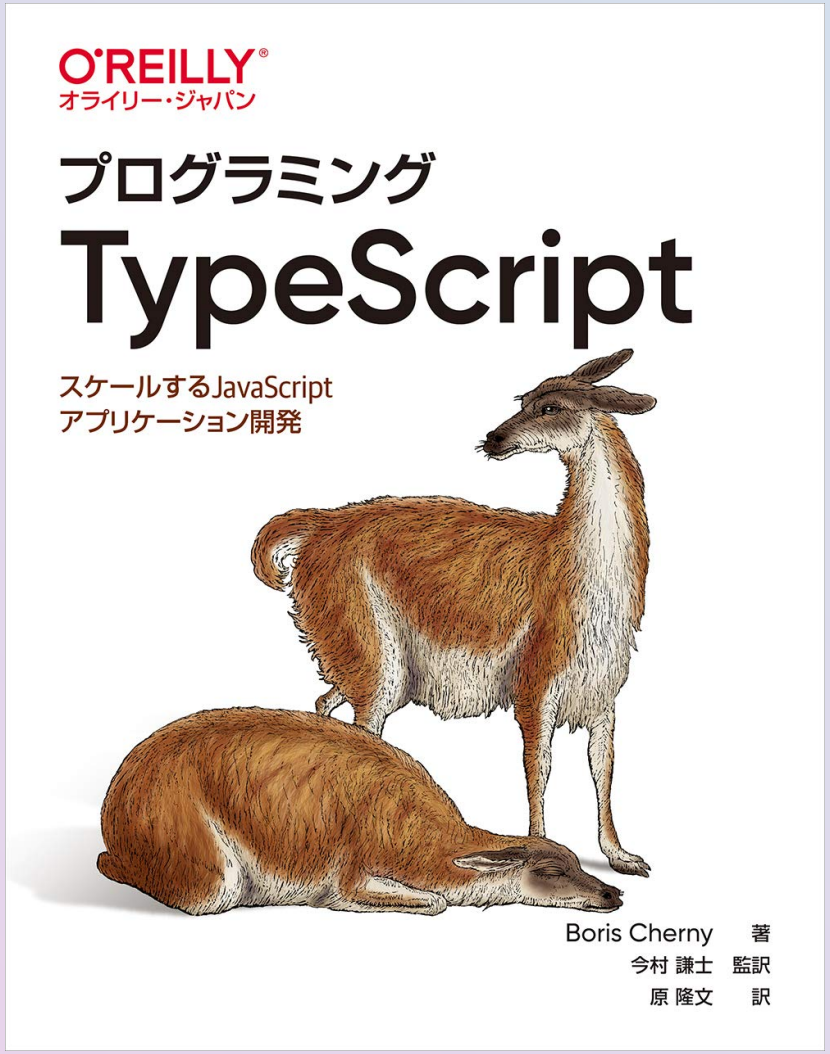
```
$ mkdir lesson_typescript
$ cd lesson_typescript
$ yarn init -y
$ yarn add -D typescript @types/node
```

環境構築

tsconfig.json の作成

`tsconfig.json` は TypeScript の設定ファイルです。

```
{
    "compilerOptions": {
        "target": "es2022",
        "lib": ["es2022"],
        "strict": true,
        "module": "esnext",
        "outDir": "dist"
    }
}
```



『プログラミング TypeScript』

スケールする JavaScript アプリケーション開発

https://www.amazon.co.jp/プログラミングTypeScript-―スケールするJavaScriptアプリケーション開発-Boris-Cherny/dp/4873119049

Marp

Markdown Presentation Ecosystem

https://marp.app/