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と、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。

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

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

一番手抜きな型付け: `any`

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

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

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

`any`を積極的に使う場面は2つあります。

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

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

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

未知の型: `unknown`

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

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

> **課題**
>
> 事例をつける

型に名前をつける

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

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

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

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

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

使い回しをしないのであれば型名の代わりに、すべての箇所に定義を書いてい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);
}

orderFood(myOrder);
```

関数のレスポンスや引数で使うオブジェクトの定義

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

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

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

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

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

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

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

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

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

オブジェクトの属性の修飾: オプション、読み込み専用

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

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

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

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

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

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

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

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。

AかつBでなければならない

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

型を合成する

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

type Instagram = {
  instagramId: string;
}

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

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

もちろん、合成した型に名前をつけることもできます。

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

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

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

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

`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?をつけるだけですが）。

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。

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。

数値用のカラム用の設定

```
type NumberStyle = {
  color: string;
}

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

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

チェックボックス、数値の両方を許容する汎用的な「カラム」型を定義

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

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

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

注釈

どこまで細かく型をつけるべきか？

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

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

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

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

1 https://github.com/future-architect/cheetah-grid

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えます。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。

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。

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()
    });
}
```

組み込みの型ガード

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

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

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

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

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

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

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

```
type Column = CheckColumn | NumberColumn;

function getValue(column: Column): string {
  if (column.field === 'number') {
    // ここではcolumnはNumberColumn型
  } else {
    // ここではcolumnはCheckColumn型
  }
}
```

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の時代にようやく、`Array.isArray()`というクラスメソッドが提供されるようになりました。

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

型ガード関数は、次のような形式で書きます。

ユーザー定義の型ガード

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

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

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

型アサーション

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

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

`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。それの文法と似た書き方になっています。 Kというのがキー名の変数で、 `in` によるループの要素が1つずつ入るイメージです。

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

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

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

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

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

以下のコードが読み込み専用の型定義になります。

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

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

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

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

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"
}
```

もしTypeScriptの型を付けるのがコストが高いと感じたら？

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

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

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

4. 多様な型を取り扱うためにジェネリクスなどの高度な型定義が必要になるケース。3のケースに付随してそれをカバーするためのコードが複雑になる場合です。

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

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

まとめ

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

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