Revised

TypeScript in Definitelyland

Revised TypeScript in Definitelyland

vvakame　著

2016-08-14 版　　ひかる黄金わかめ帝国 発行

Revised 型の国の TypeScript

本書について

本書は C87 で頒布した『型の国の TypeScript』の改訂（C90）版です。前回頒布した時は TypeScript 1.3.0 が安定版で、ちらほら 1.4.1 の形が見えている…という時期だったと記憶しています。そこから、本書の扱う TypeScript 2.0.0 までの間にいくつもの魅力的な変更がありました。本書を改訂するにあたり、さまざまな「こういう制限に注意するように」や「この方法は使うべきではない」といった文章を削りました。こうして振り返ってみると TypeScript チームの大きな働き、コードの積み重ねに感謝するばかりです。

本書の内容は Web サイト[*1]にて全文を公開しています。現在は C87 版を提供中です。なるべく夏コミから間をおかず、C90 版に差し替えたいと考えています。しかしながら、Web 公開版では表紙は含まれませんので、ぜひここで入手していってください。内容は筆者のやる気次第で今後も更新されていく可能性がありますが、やる気は今回の売れ行きに左右されます。

誤字や内容の誤り、深く掘り下げてほしい内容などがある場合、本書リポジトリまで Issue[*2]として報告していただけますと幸いです。

対象読者

本書は ECMAScript 2015 について理解しているユーザを対象にしています。 **OOP**（Object Oriented Programming）についても効能や利点をある程度理解していることが望ましいです。もし、これらの理解に不安がある場合、azu さんと laco が執筆中の ECMAScript 2015 時代のための JavaScript 入門書『js-primer』[*3]などを参考にしてください。今のところまだ書き終わっていないようですが 2016 年中くらいには書き終わるのではないでしょうか。

また解説する TypeScript の内容は 2.0.0（Beta）に準拠するため、現時点での最新の安定

[*1] http://typescript.ninja/typescript-in-definitelyland/
[*2] https://github.com/typescript-ninja/typescript-in-definitelyland/issues
[*3] https://github.com/asciidwango/js-primer

版である 1.8.10 では動作しない内容が多く含まれます。本書に登場するすべてのサンプルコードは `npm install typescript@beta` でインストールできる TypeScript 2.0.0 Beta でコンパイルの確認をしています。

本書で利用している TypeScript コンパイラは次のものです。

Version 2.0.0

本書の内容

本書は`--noImplicitAny`、`--strictNullChecks`、`--noImplicitReturns`、`--noImplicitThis` を有効にした状態を基本として解説します。各オプションの詳細については第 5 章「オプションを知り己のコードを知れば百戦危うからず」を参照してください。

また Node.js、ブラウザを問わずモジュールの仕組みを使います。これはブラウザでの実行には Browserify[*4]なり webpack[*5]なりのツールを使う前提があることを意味します。target の指定は Internet Explorer 11 でも es5 指定で十分なため `es5` 以上とし、`es3` については本書では考慮しません。

TypeScript は JSX のサポートを含みますが、筆者が今のところ JSX ないし React に興味がないため、本書では扱いません。興味がある方は公式のドキュメント[*6]を参照してください。

第 1 章「戦闘準備だ！ TypeScript！」では、TypeScript コンパイラのセットアップ方法と Visual Studio Code の設定について言及します。

第 2 章「TypeScript の基本」では、TypeScript の基本構文を簡単に解説し、このあとの章を読み解くための基礎知識を蓄えます。

第 3 章「型は便利だ楽しいな」では、TypeScript による開発を行う上で理解しておきたい型についての知識を蓄えます。

第 4 章「アドバンスド型戦略」では、TypeScript で利用可能な型のちょっと難しいところ、利用頻度は低いが知っておくと嬉しいことについて解説します。

第 5 章「オプションを知り己のコードを知れば百戦危うからず」では、tsc コマンドや

[*4] http://browserify.org/
[*5] https://webpack.github.io/
[*6] http://www.typescriptlang.org/docs/handbook/jsx.html

tsconfig.json で利用可能なオプションについて、重要なオプションを中心に解説します。

第 6 章「JS 資産と型定義ファイル」では、既存の JS 用資産を活かすための型定義ファイルについての解説と書き方、さらに DefinitelyTyped へのコントリビュートの仕方について解説します。

なぜ TypeScript を選ぶべきなのか

TypeScript は Microsoft が主導となって開発している言語で、ECMAScript（JavaScript）に静的な型付けによる検証を導入したものです。現実を見据えた言語仕様で、"未来の JavaScript そのもの"になることを目指しています。

TypeScript は ECMAScript の superset（上位互換）であることを標榜しています。つまり、ECMAScript ＋静的型付け＝ TypeScript です。そして、"TypeScript 独自の実装として表れる仕様"を注意深く避けてきています。

この方針では ECMAScript のクソな仕様の数々は（勝手に闇に葬るわけにはいかないので）残ってしまうため、自力で避ける必要があります。with 構文や try-catch 構文などは、TypeScript 的に使いにくい仕様になっているためそれと分かるでしょう。

ですが、その代わりに TypeScript は将来"正式な"JavaScript になる可能性があります[*7]。…あるんじゃないかな！

稀に TypeScript のリポジトリに「TypeScript に LINQ を導入してほしい」などという ECMAScript 仕様にない独自の要望が上がってくることがありますが、上記のポリシーを考えればそのような要望が取り込まれないのは明らかです。どうしても TypeScript に独自の仕様を入れたい場合、まずは ECMAScript 本体に入れる努力が必要でしょう。

TypeScript を選んだ時のデメリット

一番大きなデメリットは学習コストが追加で発生します。JavaScript の書き方に加え、TypeScript で型注釈を与える記法を学ばねばなりません。基本的な書き方はすぐに習得できると思いますが、既存の JavaScript ライブラリと組み合わせようとしたときに専門知識が必要になります。本書ではそのための専門知識について解説し、TypeScript を自由自在に使い

[*7] 過去に Types の正式な仕様化について TC39 ミーティングで話されたことがあったが Jonathan Turner が Microsoft を離れたため以後の進捗はよくない

こなすお手伝いをします。

　その他の懸念として TypeScript にロックインされてしまうのでは？ という不安を耳にすることがあります。これについては脱出口が用意されています。TypeScript できれいに書いたコードは`--target es6` で出力すると、単に型注釈を取り除いただけの素直な JavaScript が出力されてきます。簡単なコードを自分で書いてみて、それを ECMAScript 2015 な JS に変換して確認してみるとよいでしょう。まずは、恐れずに TypeScript に挑戦してみましょう。

目次

第1章

戦闘準備だ！ TypeScript！

1.1 インストールしてみよう

Node.js のセットアップはすでに完了しているものとします。筆者は Node.js のインストールとバージョン管理に nodebrew[*1] を利用しています。

TypeScript のインストールには、Node.js のパッケージマネージャである **npm**（Node Package Manager）を利用します。TypeScript をインストールすると、tsc というコマンドが利用可能になります。tsc コマンドで TypeScript コードのコンパイルを行います。

```
# -g をつけるとグローバルなコマンドとしてインストールする
$ npm install -g typescript
# 省略
$ tsc -v
Version 2.0.0
$ echo "class Sample {}" > sample.ts
$ tsc sample.ts
$ cat sample.js
var Sample = (function () {
    function Sample() {
    }
    return Sample;
}());
```

なお、本書執筆時点では `npm install -g typescript` で導入できる TypeScript バージョンは 1.8.10 です。2.0.0 以降が使いたい場合はしばらくの間は `npm install -g typescript@beta` とする必要があります。

ともあれ、これで準備は整いました。

[*1] https://github.com/hokaccha/nodebrew#nodebrew

cutting edge な最新版コンパイラを利用したい場合は次の手順で行います。

```
$ npm install -g typescript@next
# 省略
$ tsc -v
Version 2.1.0-dev.20160716
```

実際にはプロジェクトごとに利用する TypeScript のバージョンを変えたい場合がほとんどでしょう。常に最新のバージョンだけを使い続けるのは自分の管理するプロジェクトが増えれば増えるほど難しくなりますからね。その方法を次に示します。

```
$ npm init
# Enter 連打
$ ls package.json
package.json
$ npm install --save-dev typescript
$ ./node_modules/.bin/tsc -v
Version 2.0.0
```

この、node_modules/.bin にはプロジェクトローカルに導入された実行ファイルが集められています。npm bin コマンドを実行するとパスが得られます。macOS や Linux 環境下では$(npm bin)/tsc とするとプロジェクトローカルの tsc コマンドへのパスが取得できます。

さらに npm scripts では$PATH に node_modules/.bin が自動的に追加されます。このため、npm scripts をうまく活用してプロジェクトのビルド環境を構築すると上手かつ自然にプロジェクトローカルなビルド環境が整えられるでしょう。

```
$ npm bin
$PWD/node_modules/.bin 的なパスが表示される
$ $(npm bin)/tsc -v
Version 2.0.0
```

1.2 tsconfig.json を準備する

TypeScript では tsconfig.json という設定ファイルを利用します。必要なコンパイルオプションやコンパイル対象となるファイルはすべて tsconfig.json に記述します。すべてのツールや IDE・エディタ間で共通に利用できる設定ファイルになるため、大変役立ちます。

まずは tsconfig.json を作成してみましょう。`tsc --init` で作成できます。

```
$ tsc --init
message TS6071: Successfully created a tsconfig.json file.
$ cat tsconfig.json
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false
    },
    "exclude": [
        "node_modules"
    ]
}
```

この設定では TypeScript が利用できる制約の多くを ON にしていないため、完全にガバガバな設定です。一番最初に利用する例としてはいいかもしれませんが、運用するにあたって TypeScript が与える堅牢さのすべてを享受することはできません。

本書のサンプルコード用 tsconfig.json をリスト 1.1 に示します。この設定をベースに自分たちにとって都合がよい設定値へ変更したり制限を緩めたりすることをお勧めします。具体的に"include"や"exclude"の値は一般的なフォルダ構成を対象にしたものではないので変更したほうがよいでしょう。

リスト 1.1: 本書サンプルコード用の tsconfig.json

```
{
    "compilerOptions": {
        "module": "commonjs",
        "moduleResolution": "node",
```

```
        "target": "es6",
        "declaration": true,
        "lib": [
            "dom",
            "es2017"
        ],
        "types": [
            "node"
        ],
        "noImplicitAny": true,
        "strictNullChecks": true,
        "noFallthroughCasesInSwitch": true,
        "noImplicitReturns": true,
        "noImplicitThis": true,
        "noUnusedLocals": true,
        "noUnusedParameters": true,
        "noImplicitUseStrict": false,
        "sourceMap": false,
        "emitDecoratorMetadata": true,
        "experimentalDecorators": true,
        "forceConsistentCasingInFileNames": true,
        "traceResolution": false,
        "listFiles": false,
        "stripInternal": true,
        "skipDefaultLibCheck": true,
        "skipLibCheck": false,
        "pretty": false,
        "noEmitOnError": true
    },
    "include": [
        "code/**/*.ts"
    ],
    "exclude": [
        "node_modules",
        "code/definition-file/usage/",
        "code/definition-file/augmentGlobal/",
        "code/definition-file/issue9831/",
        "code/**/*-invalid.ts",
        "code/**/*-invalid.d.ts",
        "code/**/invalid.ts",
        "code/**/invalid.d.ts",
        "code/**/*-ignore.ts",
        "code/**/*-ignore.d.ts",
        "code/**/ignore.ts",
        "code/**/ignore.d.ts"
```

```
    ]
}
```

より詳しい説明は第 5 章「オプションを知り己のコードを知れば百戦危うからず」を参照してください。

1.3 試しにコンパイルしてみる

tsconfig.json ができたら、後はコンパイルするだけです。コンパイルの方法は tsconfig.json があれば、どんな環境でも`--project` オプションを使うだけです。それ以外のオプションについては tsconfig.json にすべて記述されています。具体的に、`tsc -p ./`または `tsc -p ./tsconfig.json` とします。

本節についてもより詳しい説明は第 5 章「オプションを知り己のコードを知れば百戦危うからず」を参照してください。

1.4 エディタ・IDE の環境を整えよう

Atom、Eclipse、Emacs、Vim、Visual Studio、WebStorm などさまざまな TypeScript 対応の IDE・エディタが存在しています[*2]。

TypeScript には language service という仕組みがあるため、IDE を作る時に TypeScript コンパイラ本体から変数の種類やメソッドの有無などの情報を得えられます。そのため多数の IDE・エディタで Visual Studio に勝るとも劣らないサポートを得ることができます。

language service に興味がある場合、https://github.com/Microsoft/TypeScript/wiki/Using-the-Language-Service-API や https://github.com/Microsoft/language-server-protocol などを参照するとよいでしょう。

[*2] https://github.com/Microsoft/TypeScript/wiki/TypeScript-Editor-Support

Visual Studio Code を使ってみる

現時点での筆者のお勧めは Visual Studio Code[*3]です。Visual Studio Code（略称：vscode）は Microsoft が提供している無料のエディタです。Visual Studio の名を冠していますが Electron[*4]を利用して組まれているマルチプラットフォームなエディタで、Windows 以外でも利用できます。

筆者は Mac OS X ユーザですが TypeScript を書く時は vscode 一本です。

tsconfig.json がプロジェクト内に配置されていれば vscode はそこから必要な設定を読み込みます。つまり、設定に手間をかけることなく TypeScript コードを書き始めることができます。

[*3] https://code.visualstudio.com/
[*4] http://electron.atom.io/

第2章

TypeScript の基本

第 1 章「戦闘準備だ！ TypeScript！」で述べたとおり、本書では ECMAScript 2015 の文法・仕様についてすべてを解説することはしません。ECMAScript 2015 の知識はどんどん広まってきていますし、今後は基本的な JavaScript の知識になっていくでしょう。ECMAScript の知識は、TypeScript 固有の知識ではないですからね。

この章では TypeScript での基本的な構文を解説します。まずは、TypeScript を使うのに必要最低限の知識を身につけていきましょう。

型の基本は第 3 章「型は便利だ楽しいな」を、難しいことや便利なことは第 4 章「アドバンスド型戦略」を見てください。既存の JavaScript な資産やライブラリを使いたい場合は第 6 章「JS 資産と型定義ファイル」を見てください。

また、本書は`--noImplicitAny`、`--strictNullChecks`、`--noImplicitReturns`、`--noImplicitThis` を有効にした状態を基本として解説します。各オプションの詳細については第 5 章「オプションを知り己のコードを知れば百戦危うからず」を参照してください。

2.1 変数

TypeScript の変数宣言はおおむね JavaScript と同じです。違うのは、リスト 2.1 のように変数名の後に：　型名という形式でその変数がどういう型の値の入れ物になるのか指定できるところです[*1]。これを **型注釈**（type annotations）と呼びます。

リスト 2.1: 型注釈付きの変数

```
let str: string;
let num: number;
let bool: boolean;

let func: Function;
```

[*1] コンパイルエラーを消すため、今後もサンプルコード中に一見意味のなさそうな export {} などが表れます

```
let obj: any; // なんでも型

str = "文字列";
num = 1;
bool = true;
func = () => { };
obj = {};

export { }
```

　型注釈の何が嬉しいかというと、型に反するようなコードを書くと tsc コマンドを使ってコンパイルしたときにコンパイルエラーになるのです。たとえばリスト 2.2 のように、整合性がとれていない箇所が TypeScript によって明らかにされます。安心安全！

リスト 2.2: 型注釈に反することをやってみる

```
let str: string;
// 文字列は数値と互換性がない！
// error TS2322: Type 'number' is not assignable to type 'string'.
str = 1;

let num: number;
// 数値は真偽値と互換性がない！
// error TS2322: Type 'boolean' is not assignable to type 'number'.
num = true;

let bool: boolean;
// 真偽値は文字列と互換性がない！
// error TS2322: Type 'string' is not assignable to type 'boolean'.
bool = "str";
```

　安心安全なのはよいですが、わかりきったことを書くのは省きたいと思うのはエンジニアの性分でしょう。そんなあなたのために、TypeScript は型推論の機能を備えています。リスト 2.3 のように、型注釈を書かずに変数定義と初期化を同時に行えます。

リスト 2.3: 初期化付き変数 = 最強

```
let str = "string";
let num = 1;
```

```
let bool = true;

let func = () => {
};
let obj = {};

export { str, num, bool, func, obj }
```

これで手で型注釈を与えずに済みます。しかも、書き方が JavaScript と全く同じになりました。楽に書ける上に実行前にコンパイルの段階で不審な臭いのするコードを発見できるようになる、第一歩です。

2.2 クラス

普通のクラス

ECMAScript 2015 より導入されたクラス構文についても各所に型注釈可能な構文が追加されています（リスト 2.4)。

リスト 2.4: さまざまなクラス要素

```
class Base {
  // インスタンス変数
  numA: number;
  strA = "string";
  public numB: number;
  private numC: number;
  protected numD: number;
  regexpA?: RegExp;

  // クラス変数
  static numA: number;
  public static numB: number;
  private static numC: number;
  protected static numD: number;
  static regexpA?: RegExp;

  // コンストラクタ
  constructor(boolA: boolean,
    public boolB: boolean,
```

```
    private boolC: boolean,
    protected boolD: boolean) {
    // エラー消し 一回も使われない可能性があると怒られる
    console.log(boolA, this.numC, this.boolC, Base.numC);
  }

  // メソッド
  hello(word: string): string {
    return "Hello, " + word;
  }

  // get, set アクセサ
  // コンパイル時に --target es5 以上が必要です
  /** @internal **/
  private _date: Date;
  get dateA(): Date {
    return this._date;
  }
  set dateA(value: Date) {
    this._date = value;
  }

  optional() {
    // 省略可能なプロパティは値の存在チェックが必要です
    if (this.regexpA != null) {
      this.regexpA.test("Hi!");
    }
    if (Base.regexpA != null) {
      Base.regexpA.test("Hi!");
    }
  }
}

let obj = new Base(true, false, true, false);
obj.numA;
obj.strA;
obj.numB;
// obj.numC; // private   なメンバにはアクセスできない
// obj.numD; // protected なメンバにもアクセスできない
obj.boolB;
// obj.boolC; // private   なメンバにはアクセスできない
// obj.boolD; // protected なメンバにもアクセスできない
obj.hello("TypeScript");
obj.dateA = new Date();
obj.dateA;
```

```
export { }
```

上から順に見て行きましょう。

まずはクラス変数、インスタンス変数です。クラスそのものやインスタンスに紐づく変数です。JavaScript っぽくいうとプロパティですね。

アクセス修飾子として、private、public、protected などの可視性を制御するアクセス修飾子を利用できます。何も指定していないとき、デフォルトの可視性は public になります。

コンパイル後の JS ファイルを見るとわかりますが `any` にキャストするとそれらの要素にアクセスできてしまうので、アクセス修飾子をつけたから外部からの変更を 100% 防げる！ と考えるのは禁物です。そのため筆者はアクセス修飾子を使うだけではなく、private な要素の prefix に_を使い、ドキュメントコメントに`@internal` をつけるといった工夫をしています。

また、プロパティには省略可能（optional）かを明示する **?**を指定できます。コンストラクタ内の処理が終わるまでの間に値がセットされないプロパティについては、省略可能である旨を指定したほうがよいかもしれません。

次はコンストラクタです。コンストラクタ自体にも前述の private、protected などのアクセス修飾子を利用できます。

引数にアクセス修飾子をあわせて書くと、インスタンス変数としてその値が利用可能になります。これを **引数プロパティ宣言**（parameter property declaration）と呼びます。引数プロパティ宣言は TypeScript 固有の記法です。そもそも、JavaScript にはアクセス修飾子がありませんからね。リスト 2.5 のようなコードを書くとリスト 2.6 のような JavaScript が出てきます。

リスト 2.5: 引数プロパティ宣言！

```
class Sample {
  constructor(public str: string) {
  }
}

let obj = new Sample("TypeScript");
// TypeScript と表示される
console.log(obj.str);

export { }
```

リスト 2.6: コンパイルするとこんなの

```
"use strict";
class Sample {
    constructor(str) {
        this.str = str;
    }
}
let obj = new Sample("TypeScript");
// TypeScript と表示される
console.log(obj.str);
```

リスト 2.4 の解説に戻ります。次はメソッドです。これも特に特筆すべき要素はありませんね。普通です。

最後に get、set アクセサです。これを含んだコードをコンパイルするときは、`--target es5` 以上を指定します。get、set アクセサを使うと、getter しか定義していない場合でもプログラム上は値の代入処理が記述できてしまうので、"use strict"を併用して実行時にエラーを検出するようにしましょう。

次に、クラスの継承も見て行きましょう（リスト 2.7)。super を使い親クラスのメソッドを参照することも普通にできます。

リスト 2.7: 普通に継承もあるよ

```
class Base {
  greeting(name: string) {
    return "Hi! " + name;
  }
}

class Inherit extends Base {
  greeting(name: string) {
    return super.greeting(name) + ". How are you?";
  }
}

let obj = new Inherit();
// Hi! TypeScript. How are you? と出力される
```

```
console.log(obj.greeting("TypeScript"));

export { }
```

TypeScript 以外のオブジェクト指向言語でもいえることですが、なんでもかんでも継承すればいいや！ という考えはよくありません。頑張ってオブジェクト指向に適した設計を行いましょう。

抽象クラス（Abstract Class）

ECMAScript にはない機能として、抽象クラスが作成できます。抽象クラスは単独ではインスタンス化できません。その代わり、抽象クラスを継承したクラスに対して、abstract で指定した要素の実装を強制できます。例を見てみましょう（リスト 2.8）。

リスト 2.8: 抽象クラス

```
abstract class Animal {
  abstract name: string;
  abstract get poo(): string;

  abstract speak(): string;
  sleep(): string {
    return "zzzZZZ...";
  }
}
// もちろん、abstract class はそのままではインスタンス化できない
// error TS2511: Cannot create an instance of the abstract class 'Animal'.
// new Animal();

class Cat extends Animal {
  // プロパティの実装を強制される
  name = "Cat";
  poo = "poo...";

  // メソッドの実装を強制される
  speak(): string {
    return "meow";
  }
}
```

```
new Cat();

export { }
```

便利ですね。private や protected に比べ、よっぽど使い出があります。

コンパイル後の JavaScript を見てみると、単なる普通のクラスに変換されていることがわかります（リスト 2.9)。

リスト 2.9: コンパイルしてしまえばただのクラス

```
"use strict";
class Animal {
    get poo() { }
    sleep() {
        return "zzzZZZ...";
    }
}
// もちろん、abstract class はそのままではインスタンス化できない
// error TS2511: Cannot create an instance of the abstract class 'Animal'.
// new Animal();
class Cat extends Animal {
    constructor(...args) {
        super(...args);
        // プロパティの実装を強制される
        this.name = "Cat";
        this.poo = "poo...";
    }
    // メソッドの実装を強制される
    speak() {
        return "meow";
    }
}
new Cat();
```

2.3 関数

普通の関数

いたって普通です（リスト 2.10）。型注釈の与え方や、引数を省略可能にする方法だけが JavaScript と違いますね。

リスト 2.10: 色々な関数定義

```
function hello(word: string): string {
  return `Hello, ${word}`;
}
hello("TypeScript");

// 返り値の型を省略すると返り値の型から推論される。明記したほうが読みやすい場合もある
function bye(word: string) {
  return `Bye, ${word}`;
}
bye("TypeScript");

// ? をつけると呼び出しときに引数が省略可能になる
function hey(word?: string) {
  // 省略可能にした時は undefined の時の考慮が必要！
  return `Hey, ${word || "TypeScript"}`;
}
hey();

// デフォルト値を指定することもできる（? を付けたのと同じ扱い + α）
function ahoy(word = "TypeScript") {
  return `Ahoy! ${word}`;
}
ahoy();

export { }
```

可変長引数もあります！（リスト 2.11）

リスト 2.11: 可変長引数の例

```
function hello(...args: string[]) {
  return `Hello, ${args.join(" & ")}`;
}
// Hello, TS & JS と表示される
console.log(hello("TS", "JS"));

export { }
```

なお、省略可能引数の後に省略不可な引数を配置したり、可変長引数を最後以外に配置したりするのは NG です（リスト 2.12)。

リスト 2.12: こういうパターンは NG

```
// 省略可能な引数の後に省略不可な引数がきてはいけない
// error TS1016: A required parameter cannot follow an optional parameter.
function funcA(arg1?: string, arg2: string) {
  return `Hello, ${arg1}, ${arg2}`;
}

// 可変長引数は必ず最後じゃないといけない
// error TS1014: A rest parameter must be last in a parameter list.
function funcB(...args: string[], rest: string) {
  return `Hello, ${args.join(", ")} and ${rest}`;
}

export { }
```

ここまで見てきた省略可能な引数やデフォルト値付き引数、可変長引数はクラスのコンストラクタやメソッドを記述するときも同様に利用できます。

アロー関数

ECMAScript 2015 で導入された **アロー関数**（Arrow Functions）を見ていきましょう（リスト 2.13)。通常の関数とアロー関数の違いについては ECMAScript 2015 の範囲であるため、本書では解説しません。

リスト 2.13: アロー関数 短くてかっこいい

```
// 次の 2 つは (this が絡まない限り) 等価！
let funcA = () => true;
let funcB = function() {
  return true;
};
funcA();
funcB();

// NOTE ここの callback の型注釈の意味は別の章で解説します
// 引数を 1 つ取って返り値無し の関数を表します
function asyncModoki(callback: (value: string) => void) {
  callback("TypeScript");
}
// ES5 時代の書き方
asyncModoki(function(value: string) {
  console.log(`Hello, ${value}`);
});
// アロー関数だとさらに楽
asyncModoki(value => console.log(`Hello, ${value}`));
// アロー関数に型付をする場合
asyncModoki((value: string): void => console.log(`Hello, ${value}`));

export { }
```

アロー関数も普通の関数同様、型注釈の与え方以外 ECMAScript 2015 との差分は見当たりません。短くてかっこいいですね。

もうひとつの便利な点として、アロー関数は親スコープの this をそのまま受け継ぎます。この仕組みのおかげでクラスのメソッドなどでコールバック関数を使うときに無用な混乱をおこさずに済みます。特別な理由が思いつかない限りアロー関数を使っておけばよいでしょう。

2.4 モジュールのあれこれ

プログラムの規模が大きくなればなるほど、機能ごとに分割して統治し、見通しをよくする必要があります。そのための武器として、ECMAScript 2015 にはモジュールがあります。1 つの JS ファイルを 1 つのモジュールと捉えます。Node.js で使われている CommonJS 形式のモジュールと考え方は一緒です。つまり、別ファイルになれば別モジュールと考え、モジュールから値を export したり import したりして大きなプログラムを分割し統治します。

歴史的経緯により、TypeScript では先に説明した 1 つの JavaScript ファイルを 1 つのモジュールと捉えた形式のことを外部モジュール（External Modules）と呼び、関数を使って 1 つの名前空間を作り出す形式を内部モジュール（Internal Modules）と呼んでいました。しかし、ECMAScript 2015 で本格的に"モジュール"の概念が定義されたため、TypeScript では今後はモジュールといえば外部モジュールを指し、内部モジュールのことは `namespace` と呼ぶようになりました。これにあわせて、内部モジュールの記法も旧来の `module` から `namespace` に変更されました。未だに `module` を使うこともできますが、今後は `namespace` を使ったほうがよいでしょう。

本書でも、これ以降は単にモジュールと書く場合は外部モジュールのことを指し、namespace と書いた時は内部モジュールのことを指します。

仕様としてモジュールが策定され、WHATWG でブラウザでのモジュールの動作について議論が進んでいる現状、namespace のみを使ってプログラムを分割・構成すると将来的にはきっと負債になるでしょう。これから新規にプロジェクトを作成する場合は実行する環境が Node.js であれ、ブラウザであれ、モジュールを使って構成するべきでしょう。

モジュール

モジュールは前述のとおり、1 ファイル＝ 1 モジュールとしてプロジェクトを構成していく方式です。`import * as foo from "./foo";` のように書くと、そのファイルから./foo.ts[*2]を参照できます。ここでは、./foo がひとつのモジュールとして扱われます。

TypeScript では CommonJS、AMD、System（SystemJS）、UMD、ECMAScript 2015 によるモジュールの利用に対応しています。いずれの形式で出力するかについてはコンパイル時に`--module commonjs` などの形式で指定できます。

本書では Node.js でも Browserify や webpack で広く利用しやすい CommonJS 形式についてのみ言及します。対応形式の中では AMD や SystemJS については癖が強く、tsc に与えることができるオプションの数も多いため興味がある人は自分で調べてみてください。筆者は両形式はあまり筋がよいとは今のところ思っていませんけれど。

さて、実際のコード例を見てみましょう。foo.ts（リスト 2.14）、bar.ts（リスト 2.15）、buzz.ts（リスト 2.16）というファイルがあるとき、それぞれがモジュールになるので 3 モ

[*2] Node.js 上の仕様（TypeScript ではない）について細かくいうと、require("./foo") すると最初に./foo.js が探され、次に./foo.json、./foo.node と検索します

ジュールある、という考え方になります。

リスト 2.14: foo.ts

```
// default を bar という名前に hello 関数をそのままの名前で import
import bar, { hello } from "./bar";
// モジュール全体を bar2 に束縛
import * as bar2 from "./bar";
// ECMAScript 2015 形式のモジュールでも CommonJS 形式で import できる
import bar3 = require("./bar");

// Hello, TypeScript! と表示される
console.log(hello());
// Hi!, default と表示される
console.log(bar());
// 上に同じく Hello, TypeScript! と Hi!, default
console.log(bar2.hello());
console.log(bar2.default());
// 上に同じく Hello, TypeScript! と Hi!, default
console.log(bar3.hello());
console.log(bar3.default());

// export = xxx 形式の場合モジュール全体を buzz に束縛
import * as buzz from "./buzz";
// CommonJS 形式のモジュールに対して一番素直で真っ当な書き方 in TypeScript
import buzz2 = require("./buzz");
// 両方 Good bye, TypeScript! と表示される
console.log(buzz());
console.log(buzz2());
```

リスト 2.15: bar.ts

```
export function hello(word = "TypeScript") {
  return `Hello, ${word}`;
}

export default function(word = "default") {
  return `Hi!, ${word}`;
}
```

リスト 2.16: buzz.ts

```
function bye(word = "TypeScript") {
  return `Good bye, ${word}`;
}
// foo.ts で ECMAScript 2015 形式で import する際、
// 次のエラーが出るのを抑制するためのハック
// error TS2497: Module '"略/buzz"' resolves to a non-module entity
//   and cannot be imported using this construct.
namespace bye { }

// CommonJS 向け ECMAScript 2015 では×
export = bye;
```

各モジュールのトップレベルで export したものが別のファイルから import されたときに利用できているのがわかります。コンパイルして結果を確かめてみましょう。Node.js に慣れている人なら、見覚えのある形式のコードが出力されていることが分かるでしょう。

```
$ tsc --module commonjs --target es6 foo.ts
$ cat foo.js
"use strict";
// default を bar という名前に hello 関数をそのままの名前で import
const bar_1 = require("./bar");
// モジュール全体を bar2 に束縛
const bar2 = require("./bar");
// ECMAScript 2015 形式のモジュールでも CommonJS 形式で import できる
const bar3 = require("./bar");
// Hello, TypeScript! と表示される
console.log(bar_1.hello());
// Hi!, default と表示される
console.log(bar_1.default());
// 上に同じく Hello, TypeScript! と Hi!, default
console.log(bar2.hello());
console.log(bar2.default());
// 上に同じく Hello, TypeScript! と Hi!, default
console.log(bar3.hello());
console.log(bar3.default());
// export = xxx 形式の場合モジュール全体を buzz に束縛
const buzz = require("./buzz");
// CommonJS 形式のモジュールに対して一番素直で真っ当な書き方 in TypeScript
const buzz2 = require("./buzz");
// 両方 Good bye, TypeScript! と表示される
```

```
console.log(buzz());
console.log(buzz2());
$ cat bar.js
"use strict";
function hello(word = "TypeScript") {
    return `Hello, ${word}`;
}
exports.hello = hello;
function default_1(word = "default") {
    return `Hi!, ${word}`;
}
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = default_1;
$ cat buzz.js
"use strict";
function bye(word = "TypeScript") {
    return `Good bye, ${word}`;
}
module.exports = bye;
```

namespace

現実的にコードを書く時には namespace を使わないほうがよいのです。ですので、できれば namespace については説明したくないのですが、そうはいかない理由があります。それが、型定義ファイルの存在です。型定義ファイルの中ではインタフェースや関数などをきれいに取りまとめるために namespace の仕組みを活用する場面がでてきます。そのため、TypeScript の習熟度を高めるうえで namespace は避けては通れないのです。

まずは簡単な例を見てみましょう（リスト 2.17)。

リスト 2.17: namespace を使ったコード

```
namespace a {
  // export してないものは外部からは見えない
  class Sample {
    hello(word = "TypeScript") {
      return `Hello, ${word}`;
    }
  }
```

```
  export interface Hello {
    hello(word?: string): string;
  }
  export let obj: Hello = new Sample();
}
namespace a {
  export function bye(word = "JavaScript") {
    return `Bye, ${word}`;
  }

  // 定義を分けてしまうと同名のモジュールでも export されていないものは見えない
  // error TS2304: Cannot find name 'Sample'.
  // let tmp = new Sample();
}

namespace b {
  export namespace c {
    export function hello() {
      return a.obj.hello();
    }
  }
}
namespace d.e {
  export function hello() {
    return a.obj.hello();
  }
}

// Hello, TypeScript と表示される
console.log(b.c.hello());
// Hello, TypeScript と表示される
console.log(d.e.hello());
```

なかなかシンプルです。namespace の内側で定義した要素はクラスであれ、関数であれ、なんであっても export をつけなければ外側から見えないようになります。

これをコンパイルした結果を確認してみます（リスト 2.18)。

リスト 2.18: コンパイルすると関数を使った構文に展開される

```
var a;
(function (a) {
    // export してないものは外部からは見えない
```

```
    class Sample {
        hello(word = "TypeScript") {
            return `Hello, ${word}`;
        }
    }
    a.obj = new Sample();
})(a || (a = {}));
var a;
(function (a) {
    function bye(word = "JavaScript") {
        return `Bye, ${word}`;
    }
    a.bye = bye;
})(a || (a = {}));
var b;
(function (b) {
    var c;
    (function (c) {
        function hello() {
            return a.obj.hello();
        }
        c.hello = hello;
    })(c = b.c || (b.c = {}));
})(b || (b = {}));
var d;
(function (d) {
    var e;
    (function (e) {
        function hello() {
            return a.obj.hello();
        }
        e.hello = hello;
    })(e = d.e || (d.e = {}));
})(d || (d = {}));
// Hello, TypeScript と表示される
console.log(b.c.hello());
// Hello, TypeScript と表示される
console.log(d.e.hello());
```

関数を使って名前空間を擬似的に作っています。モジュールも let や const のようなブロックスコープもなかった頃の名残ですね。

長い名前を使うのが嫌なときはリスト 2.19 のように、import 句を使うこともできます。先

に説明したモジュールではこれとは異なる import 句の使い方が出てきましたが、区別しましょう。

リスト 2.19: import 句で別名を作る

```
namespace a {
  export class Sample { }
}

namespace b {
  // 他のモジュールも普通に参照できる
  let objA: a.Sample;
  objA = new a.Sample();

  // めんどくさいなら import 句 を使えばいい
  import Sample = a.Sample;
  let objB: Sample;
  objB = new Sample();

  // 別に違う名前をつけてもいい (けど混乱しちゃうかも？
  import Test = a.Sample;
  let objC: Test;
  objC = new Test();

  // 別に名前が違っても互換性が失われるわけではないのだ
  objA = new Test();
}
```

第3章

型は便利だ楽しいな

TypeScript の華はやはり型！ 第 2 章「TypeScript の基本」など所詮児戯に過ぎぬわッ！！

この章では TypeScript の型の仕組みのうち、日常的に使う箇所を重点的に解説していきます。TypeScript コードを書く分には使わない範囲（型定義ファイルで主に使う範囲）や、仕様的に少し複雑なものについては第 4 章「アドバンスド型戦略」で紹介します。

まず、TypeScript に熟達していく上で必ず意識しなければいけないのが **型の宣言空間**（type declaration space）と **値の宣言空間**（variable declaration space）の存在です。別の言い方をすると、型定義と実装の差です。

Java や C#だと、おおむね両者は密接に結びついていて、そこの差で困ることは少ないです。筆者が簡単に思いつく範囲では、似たような事例は Java での Generics の型パラメータの type erasure ぐらいでしょうか。Java の Generics の型パラメータは実行時には消されてしまうため、たとえば `new T();` というコードを書くことはできません。TypeScript の型と値の区別は、Java の例に近いかもしれません。

情報として型が存在していても、値として存在していない。そういう場面は TypeScript ではたくさん遭遇します。差を理解し積極的に活用できるようになると TypeScript 上級者といえるでしょう。

この章では、そんな TypeScript の型の宣言空間で活躍する要素を紹介していきます。実用上のテクニックは第 4 章「アドバンスド型戦略」や第 6 章「JS 資産と型定義ファイル」でも言及します。

3.1　オブジェクト型リテラル（Object Type Literals）

オブジェクト型リテラルは、JS のオブジェクトリテラルに似た記法で、匿名の型を作り出す機能です（リスト 3.1）。

リスト 3.1: 基本的な例

```
// オブジェクトリテラルで値を作成！
let objA = {
  x: 1,
  y: 2,
};

// オブジェクト型リテラルで型を作成！（値は無し
// 上記の objA の型は型推論で objB と同一になる
let objB: {
  x: number;
  y: number;
};
objB = { x: 1, y: 2 };

// おんなじ！
objA = objB;
objB = objA;

export { }
```

値の代わりに型名を、要素の終わりに , ではなく ; を書くだけです。簡単ですね。

オブジェクト型リテラルは型を指定する箇所[*1]であればどこでも使えます（リスト 3.2）。

リスト 3.2: でも、正直読みづらい

```
// 関数の仮引数や返り値に対して
function move(
  value: { x: number; y: number; },
  delta: { dx?: number; dy?: number; },
): { x: number; y: number } {
  if (delta.dx) {
    value.x += delta.dx;
  }
  if (delta.dy) {
    value.y += delta.dy;
  }
  return value;
}
```

[*1] interface の extends の後とか typeof の後の識別子とかは厳密にいうと型を指定する箇所ではありません

```
let result = move({ x: 1, y: 2 }, { dx: -2 });
// 次のように表示される
// {
//   "x": -1,
//   "y": 2
// }
console.log(JSON.stringify(result, null, 2));

export { }
```

では、オブジェクト型リテラルで使える書き方を 5 つ見ていきましょう。

プロパティシグニチャ（Property Signatures）

1 つ目は、実はすでに登場しているプロパティを示す記法、プロパティシグニチャです（リスト 3.3）。

リスト 3.3: 大将！ いつものやつ！

```
let obj: {
  property: string;
};
// 当てはまる値はこんな感じ
obj = {
  property: "Hi!",
};

export { }
```

素直でわかりやすいですね。

コールシグニチャ（Call Signatures）

2 つ目は、そのオブジェクトが関数として呼び出し可能であることを示す記法、コールシグニチャです（リスト 3.4）。

リスト 3.4: 関数として利用できる

```
let obj: {
  (word: string): string;
};
// 当てはまる値はこんな感じ
obj = word => `Hello, ${word}`;
obj = (word: string): string => {
  return `Hello, ${word}`;
};
obj = function(word: string): string {
  return `Hello, ${word}`;
};
// 呼び出してみよう！
let str = obj("TypeScript");
console.log(str);

export { }
```

オーバーロードも表現できます（リスト 3.5)。

リスト 3.5: オーバーロードも表現できる

```
let obj: {
  // overload もできるよ
  (word: string): string;
  (): number;
};
// 当てはまる値はこんな感じ
// すべての引数と返り値に矛盾しないようにしなければならない…
obj = (word?: string): any => {
  if (typeof word === "string") {
    return `Hello, ${word}`;
  } else {
    return 42;
  }
};
// 呼び出してみよう！
let str = obj("TypeScript");
// Hello, TypeScript と表示される
console.log(str);

let num = obj();
// 42 と表示される
```

```
console.log(num);

export { }
```

実装が煩雑になるのでなるべくオーバーロードを自分のコード内で利用することは避けたいところです。

コンストラクトシグニチャ（Construct Signatures）

3 つ目は、対象オブジェクトがコンストラクタとして利用可能であることを示す記法、コンストラクトシグニチャです（リスト 3.6）。

リスト 3.6: new できる

```
let clazz: {
  new (): any;
};

// 当てはまる値はこんな感じ
class Sample {
}
clazz = Sample;
let obj = new clazz();

// クラス式だとこんな感じ
clazz = class { };
obj = new clazz();

export { }
```

TypeScript では、クラスを定義しなければコンストラクトシグニチャにマッチするコードを書くことはできません。関数＋型アサーション（要するにキャスト）を使って any に変換し無理やり回避する方法はありますが、使わないほうがよいでしょう。

コンストラクトシグニチャも、コールシグニチャ同様にオーバーロードが可能で、引数毎に別々の型が返るような定義も可能です。しかし、実装するのも正しく利用するのもめんどうなのでほどほどにしましょう。

コンストラクトシグニチャは主に型定義ファイルの作成時にお世話になります。

インデックスシグニチャ（Index Signatures）

4 つ目は、インデックスシグニチャです。添字によるプロパティアクセスに対して、型を当てはめられます（リスト 3.7）。

リスト 3.7: プロパティアクセスの例

```
let objA: {
  [index: number]: string;
} = {};
let objB: {
  [index: string]: string;
} = {};

// どういった使い方ができるの？
let s1 = objA[1];

// --noImplicitAny 付きだとちゃんとエラーになる
// error TS7015: Element implicitly has an 'any' type
//   because index expression is not of type 'number'.
// var s2 = objA["test"];

// インデックスの型指定が string の場合 string でも number でも OK
let s3 = objB[1];
let s4 = objB["test"];

// 当てはまる値はこんな感じ
objA = {
  0: "str",
  // オブジェクトリテラルで直接変数に代入する場合、型に存在しない値があるとエラーになる
  // error TS2322: Type
  //  '{ 0: string; str: string; }'
  //     is not assignable to type '{ [index: number]: string; }'.
  //  Object literal may only specify known properties,
  //     and 'str' does not exist in type '{ [index: number]: string; }'.
  // str: "str",
};
// 変数にオブジェクトリテラル直で代入でなければ余計なパラメータがついていても許される
let tmp = {
  0: "str",
  str: "str",
};
```

```
objA = tmp;

objB = {
  0: "str",
  str: "str",
  // インデックスの型が string の場合、すべてのプロパティの型がインデックスシグニチャに
  // 反しないようにしなければならない
  // error TS2322: Type
  //  '{ 0: string; str: string; num: number; }'
  //     is not assignable to type
  //  '{ [index: string]: string; }'.
  //  Property 'num' is incompatible with index signature.
  //    Type 'number' is not assignable to type 'string'.
  // num: 1,
};

export { s1, s3, s4 }
```

インデックスシグニチャの型と、インデックスシグニチャ以外（たとえばプロパティシグニチャ）の型との間に矛盾が生じないようにする必要があります。

ちなみに、TypeScript はリスト 3.8 のように、文字列リテラルによるアクセスも許可しています。

リスト 3.8: 実はドットによるアクセスと同じ堅牢さを誇る

```
let obj = {
  str: "string",
  num: 1,
};

// 文字列リテラルによるアクセスだと普通に . アクセス同様に型情報が得られる
let str: string = obj["str"];
let num: number = obj["num"];
// ちなみに、存在しない要素にアクセスすると any になる
// --noImplicitAny を使うと暗黙的 any としてちゃんと怒られる 萌え
// error TS7017: Index signature of object type implicitly has an 'any' type.
// let any = obj["notExists"];

// 即値じゃないとダメ！ コンパイラが処理できないのです
let propertyName1 = "str";
// error TS7017: Index signature of object type implicitly has an 'any' type.
// let str2 = obj[propertyName1];
```

```
// なお、string literal types を使っても怒られます
let propertyName2: "str" = "str";
// error TS7017: Index signature of object type implicitly has an 'any' type.
// let str3 = obj[propertyName2];

export { str, num, propertyName1, propertyName2 }
```

　インデックスシグニチャの利用は静的な検証の恩恵からするりと外れる危険性が高いため、安易に使わないようにしましょう。

メソッドシグニチャ（Method Signatures）

　最後の 5 つ目は、メソッドシグニチャです。あるプロパティがメソッドであることを表現できます（リスト 3.9）。

リスト 3.9: メソッドの定義っぽい

```
let obj: {
  hello(word: string): string;
};

// 当てはまる値はこんな感じ
obj = {
  hello(word: string) {
    return `Hello, ${word}`;
  },
};
obj = {
  hello: (word: string) => `Hello, ${word}`,
};
obj = {
  hello: function(word: string) {
    return `Hello, ${word}`;
  },
};

// プロパティシグニチャ + 関数型リテラル と実質同じ意味
let obj2: {
  hello: (word: string) => string;
};
```

```
obj2 = obj;
obj = obj2;

export { }
```

"プロパティシグニチャ + 関数型リテラル（後述）"の組み合わせでも表現できますが、メソッドシグニチャのほうがぱっと見わかりやすいですね。

オブジェクトリテラルと厳密なチェック

オブジェクト型リテラルの話と関わりが深いのでここで説明します。

オブジェクトリテラルを使って値を作る時に、厳密なチェックが行われる場合があります。それは、値を直接何らかの型に当てはめた場合です。例を見てみましょう（リスト 3.10）。

リスト 3.10: 厳密にチェックされる場合、されない場合

```
// OK！ 変数の型に対して、過不足なし
let obj: { name: string; } = {
  name: "TypeScript",
};

// NG… 変数の型に対してプロパティが多すぎる
// error TS2322: Type '{ name: string; version: string; }'
//    is not assignable to type '{ name: string; }'.
//    Object literal may only specify known properties,
//    and 'version' does not exist in type '{ name: string; }'.
obj = {
  name: "JavaScript",
  version: "2016",
};

// オブジェクトリテラルの直接代入じゃなければ OK 互換性はあるのだ
let tmp = {
  name: "JavaScript",
  version: "2016",
};
obj = tmp;

// この制約はオプション名の typo の検出に役立つ
interface FooOptions {
```

```
  fileName?: string;
  checkBar?: boolean;
}
declare function foo(opts: FooOptions): void;

// fileName の大文字小文字を間違えている！
// Object literal may only specify known properties,
// and 'filename' does not exist in type 'FooOptions'.
foo({
  filename: "vvakame.txt",
  checkBar: true,
});

export { }
```

この制約はなかなか強力で、慣れないうちはコンパイルエラーを回避する方法がわからないかもしれません。型定義ファイルを使っていると、型定義ファイルに不足がある場合や時には正規の方法で攻略するのが難しい場合すらあります。そのようなコンパイルエラーは型定義ファイルを修正して対処してほしいところですが、急いでいるのであれば一旦別の変数に代入してから再代入することで回避できます。一旦別変数作戦は、any にキャストするやり方よりは型の不整合の検出などの点で有利なため、いくらかマシなやり方といえます。

readonly 修飾子

TypeScript 固有の機能である readonly 修飾子について紹介します。readonly と指定したプロパティは、読み取り専用となり変更を禁止できます（リスト 3.11）。

リスト 3.11: readonly で読み取り専用にする

```
interface Foo {
  // readonly を使うと直接は書き換えできなくなる
  readonly str: string;
}

let objA: Foo = {
  str: "TypeScript",
};
// 上書きはできない！
// error TS2450: Left-hand side of assignment expression
```

```
//   cannot be a constant or a read-only property.
// objA.str = "JavaScript";

// 別に const ではないので迂回路から変更できてしまう
let objB = {
  str: "Mutable",
};
objA = objB;
// objB 経由で objA.str を書き換える
objB.str = "Modified!";
// Modified! と表示される
console.log(objA.str);

export { }
```

readonly と変更された該当箇所が変更不可になるだけなので迂回路を使うと値は変更できてしまいます。

クラスのプロパティに対して利用すると、コンストラクタでだけ値が変更可能になります（リスト 3.12）。Java の final に似ていますね。

リスト 3.12: readonly の変更は constructor だけ

```
class Foo {
  readonly str: string;

  constructor() {
    // 変更可能
    this.str = "TypeScript";
  }

  modify() {
    // readonly が変更できるのは constructor だけ！
    // error TS2450: Left-hand side of assignment expression
    //   cannot be a constant or a read-only property.
    // this.str = "JavaScript";
  }
}

export { Foo }
```

もちろん、TypeScript 上の制約なのでコンパイル後の JavaScript では普通に変更可能なコードが出力されてきます。使うとある程度 TypeScript コンパイラが身を守るのを助けてくれるヒント、ぐらいに捉えておきましょう。

また、get アクセサのみの実装について型定義ファイルを生成させると、これも readonly 修飾子に変換されます（リスト 3.13、リスト 3.14）。

リスト 3.13: get アクセサのみの実装

```
class Sample {
  get name() {
    return "TypeScript";
  }
}

export { Sample }
```

リスト 3.14: readonly 修飾子に変換される

```
declare class Sample {
    readonly name: string;
}
export { Sample };
```

3.2 関数型リテラル（Function Type Literals）

関数も型として表現できます（リスト 3.15）。

リスト 3.15: 関数も型として表現できる

```
let func: (value: string) => string;
// 当てはまる値はこんな感じ
func = word => `Hello, ${word}`;
func = (word: string) => {
  return `Hello, ${word}`;
};
```

```
func = function(word: string) {
  return `Hello, ${word}`;
};

// 型に対して実装の引数の数が少ないのは OK
func = () => "Hello, TypeScript";

// 型に対して実装の引数の数が多い場合、省略可能かデフォルト値付きでなければならない
func = (v1: string, v2 = "JavaScript") => `Hello, ${v1} & ${v2}`;
```

アロー関数の実装は (word: string): string => "Hello, " + word; という記法なのに対して、関数型リテラルは (word: string) => string という記法で、返り値の型を置く位置が=>の前後という違いがあるので間違えないように気をつけましょう。

3.3 インタフェース（Interfaces）

インタフェースは多くの OOP な言語に存在しているので、ご存知の方も多いでしょう。TypeScript のインタフェースは通常のインタフェース以上に色々な場面で登場します。TypeScript での一番基本的な使い方は名前付きオブジェクト型リテラルを作ることです。インタフェースの中で許される記法はオブジェクト型リテラルそのままです。

TypeScript でのインタフェースの酷使されっぷりをリスト 3.16 で紹介します。

リスト 3.16: 酷使されるインタフェースさん

```
// 一般的な用法
interface A {
  str: string;
}
// クラスに特定の実装を強制する
class AImpl implements A {
  str: string;
}
let objA: A = new AImpl();

// インタフェースは他のインタフェースを拡張できる
interface B1 {
  str: string;
}
interface B2 extends B1 {
```

```
  num: number;
}
// 代入する値はインタフェースを実装したクラスに限らない
let objB: B2 = {
  str: "string",
  num: 42,
};

// interface はクラスすら拡張できる！（実装はなかったことになる
class FooClass {
  constructor(public num: number) {
  }
}
interface C extends FooClass {
  str: string;
}
let objC: C = {
  num: 42,
  str: "string",
};
```

3.4 構造的部分型（Structural Subtyping）

構造的部分型は、乱暴にいうと静的型付け用の duck typing です。TypeScript では、構造が一致するかどうかで型の互換性を判定します（リスト 3.17)。そこに実際の継承関係は必要ありません。

リスト 3.17: 大体一緒ならまぁ一緒ってことでいいよね

```
// クラス Foo は string 型の str という名前のプロパティと
// number 型の num という名前のプロパティを持つ
class Foo {
  str = "string";
  num = 1;
}

// 構造が完全に一致…！！！
// 構造が同じならもう同じってことでいいよね
let obj: Foo = {
  str: "Hi!",
```

```
  num: 42,
};

export { Foo, obj }
```

そのため、リスト 3.18 のようなコードも TypeScript としては正しいです。

リスト 3.18: Point インタフェースが要求されているが？

```
interface Point {
  x: number;
  y: number;
}
// Point の実装を強制する！！！
class PointImpl1 implements Point {
  constructor(public x: number, public y: number) {
  }
}
// Point の実装が強制されないけど互換性はある！
class PointImpl2 {
  constructor(public x: number, public y: number) {
  }
}

// 引数に Point をとる
function double(p: Point): Point {
  return {
    x: p.x * 2,
    y: p.y * 2,
  };
}
// 次のすべては正しいコードとなります
double(new PointImpl1(1, 2));
double(new PointImpl2(3, 4));
double({
  x: 5,
  y: 6,
});
let p = {
  x: 7,
  y: 8,
  z: 9,
};
```

```
double(p); // オブジェクトリテラルを直で渡す場合余計な要素があると怒られる
```

　このコードはコンパイルがちゃんと成功します。Point インタフェースに適合させることが目的のクラスであれば、きちんと implements 節を使ったほうが意図的な仕様であることが明示できるためより好ましいです。

　なお、省略可能なプロパティは存在していなくても同じ型であるものとして扱われます（リスト 3.19）。

リスト 3.19: 省略可能な（?がある）プロパティは値がなくてもよい

```
interface Point {
  x: number;
  y: number;
  color?: string; // なくてもいい
}
function printPoint(p: Point): void {
  let message = `x=${p.x}, y=${p.y}`;
  if (p.color) {
    message = `${p.color}(${message})`;
  }
  console.log(message);
}
// optinal なプロパティはなくても大丈夫！
// x=1, y=2 と表示される
printPoint({
  x: 1,
  y: 2,
});
// red(x=1, y=2) と表示される
printPoint({
  x: 1,
  y: 2,
  color: "red",
});
// とはいえ、型が不一致だと怒られる。
// error TS2345: Argument of type '{ x: number; y: number; color: boolean; }'
//   is not assignable to parameter of type 'Point'.
//   Types of property 'color' are incompatible.
//     Type 'boolean' is not assignable to type 'string'.
// printPoint({
//   x: 1,
```

```
//   y: 2,
//   color: true,
// });
```

3.5　型アサーション（Type Assertions）

　型アサーションは他の言語でいうところのキャストです。キャストの書き方は 2 種類あり、リスト 3.20 のように<変換後型名>値か値 as 変換後型名と書くだけです。

リスト 3.20: 型アサーション 基本例

```
let obj: any = 1;

// お前は今後 number として生きよ…
let num = <number>obj;

let str = "string";
// 非常に外道なコードを書くことができる… 人としての良識を疑う
// string を number な型の変数に代入するだなんてなんて卑劣な…！
num = <any>str;
// 後置のキャストもあります as 型名という形式
num = str as any;

export { }
```

　もちろん、互換性のない型に型アサーションで変換しようとすると怒られます（リスト 3.21）。

リスト 3.21: string は number にはなれない

```
let str = "str";
// any を経由しない場合、整合性の無い型アサーションは成功しない！ 安全！
// error TS2352: Type 'string' cannot be converted to type 'number'.
let num: number = str as number;
```

　ダウンキャストも実行できます（リスト 3.22）。TypeScript コード中で一番よくみる型ア

サーションは、この instanceof との合わせ技のパターンでしょう。

リスト 3.22: 親クラスから子クラスへ変換

```
class Base {
  str: string;
}

class InheritA extends Base {
  num: number;
}
class InheritB extends Base {
  bool: boolean;
}

// とりあえず型は親クラスとして…
let obj: Base = new InheritA();

// obj の中身は InheritA だ！ 人間はそれを知っているので無理やりダウンキャスト
(obj as InheritA).num;

// なお、instanceof で調べると勝手に対象の型にダウンキャストされる（型の narrowing）
// もちろん、キャストするよりこっちのやり方のほうが安全
if (obj instanceof InheritA) {
  obj.num;
} else if (obj instanceof InheritB) {
  obj.bool;
} else {
  obj.str;
}

export { }
```

any に一旦キャストすればなんにでも化けさせられるので、これは一種の切り札です。型定義ファイルを使っているときに、その型定義ファイルに不足や誤りがある場合、型アサーションでとりあえず切り抜けたい場合があります（リスト 3.23）。

リスト 3.23: 無理やりなんとかする例

```
// こういう、型定義があるとする。実装は他のライブラリなどが与える。
declare class Base {
  str: string;
```

```
}

let obj = new Base();
// 本当は、Base クラスが num プロパティも持ってるんだけど…
// 型定義が不足しているので 10 秒で解決するには any で誤魔化すしかない
// キレイごとだけじゃ世の中生きていけないんじゃよ…
// でも、本当は型定義のほうを修正するのが正しいです
let num: number = (obj as any).num;

export { num }
```

3.6　ジェネリクス（Generic Types）

　いよいよ来ました。最後の大ボスです。Java などでは総称型とも呼ばれます。

　ジェネリクスなんて知らんわい！ という人も、実はすでに色々なところでお世話になっています。

ジェネリクスの基本

　TypeScript で一番よく使うジェネリクスを使ったクラスは、Array です。例を見てみましょう（リスト 3.24）。

リスト 3.24: 配列はジェネリクスに支えられております

```
// string[] は実は Array<string> と同じ意味なのだ！（Array だけ特別に！
let strArray: Array<string> = ["a", "b", "c"];
let numArray: Array<number> = [1, 2, 3];

// Array のメソッドとして forEach がある
// forEach で渡される値の型はそれぞれ違う（forEach は汎用的だ！
strArray.forEach(v => v.charAt(0));
numArray.forEach(v => v.toFixed(2));

// 頑張ってこうやって書いてもいいけど、めんどいよね
strArray.forEach((v: string) => v.charAt(0));
numArray.forEach((v: number) => v.toFixed(2));

// あと、間違った型を指定したときにエラーにならないとこわい…
```

```
// error TS2345: Argument of type '(v: RegExp) => boolean'
//     is not assignable to parameter of type
//     '(value: string, index: number, array: string[]) => void'.
//   Types of parameters 'v' and 'value' are incompatible.
//     Type 'string' is not assignable to type 'RegExp'.
// strArray.forEach((v: RegExp) => v.test("str"));
```

実は、string[] という型は Array<string>と同じ意味なのです！Array だけは TypeScript の中で特別扱いされています。

ここで出てくる<string>という部分がジェネリクスの肝です。Array<string>を声に出して読むと、"string の Array"になります。ただの Array ではないのです。"string の"という所が重要です。string を別のものにして"number の Array"とか"RegExp の Array"とすることもできます。つまり、色々な型に対して、"○○の Array"ということができるのです。これをプログラム上で表現すると Array<T>という表現になります。

ここで新しく出てきた T を **型パラメータ**（type parameters）と呼びます。実際、ここで出てくるアルファベットは T ではなくてもかまいせん。Type でもいいですし、なんでもよいです。ただ、慣習として既存の型とかぶらないようにするためにアルファベット大文字 1 文字を使う場合が多いです。代表的な例では Type の頭文字である T、アルファベット的に T の次の文字である U や、Return の頭文字である R などが使われます。

さて、では TypeScript の標準の型定義情報が書かれている lib.d.ts から一部を抜粋したリスト 3.25 を見てみます[*2]。

リスト 3.25: Array<T>が登場する

```
interface ArrayConstructor {
  new <T>(...items: T[]): T[];
};

interface Array<T> {
  length: number;
  push(...items: T[]): number;
  pop(): T | undefined;
  forEach(callbackfn: (value: T) => void, thisArg?: any): void;
  [n: number]: T;
}
```

[*2] 紙面の都合上横幅が辛かったので forEach の定義を大胆に切り詰めてあります…。ごめんなさい！

色々な所で T が使われています。push の定義を見ると、"○○の Array に対して、○○の値いくつかを追加するメソッド push"とか、"○○の Array に対して、末尾の○○の値を 1 つ取得するメソッド pop"、"○○の Array に対して、○○の値それぞれに対して callbackFn を適用するメソッド forEach"などの、汎用化された要素がたくさんあります。

ここで、型パラメータ T を実際に string で具体化します（リスト 3.26）。

リスト 3.26: string 専用 Array に変身

```
interface ArrayConstructor {
  new (...items: string[]): string[];
};

interface Array {
  length: number;
  push(...items: string[]): number;
  pop(): string | undefined;
  forEach(callbackfn: (value: string) => void, thisArg?: any): void;
  [n: number]: string;
}
```

"string の Array に対して、string の値をいくつか追加するメソッド push"や、"string の Array に対して、末尾の string の値を 1 つ取得するメソッド pop"、"string の Array に対して、string の値それぞれに対して callbackFn を適用するメソッド forEach"などになりました。ジェネリクス、使う分にはめっちゃ簡単ですね！

このように、ジェネリクスを使うと柔軟性と堅牢さを両立させることができます。

ジェネリクスの書き方色々

ここでジェネリクスの書き方を確認しておきます（リスト 3.27）。

リスト 3.27: Array<T>が登場する

```
// それぞれの T は独立していて、関連性はない

// クラスとジェネリクス
```

```
class SampleA<T> {
  constructor(public data: T) {
  }
}
// 使い方 <string>は省略しても引数から推論可能
let objA = new SampleA<string>("str");

// インタフェースとジェネリクス
interface SampleB<T> {
  data: T;
}
// 使い方
let objB: SampleB<number> = { data: 1 };

// オブジェクト型リテラル各種
let obj: {
  new <T>(value: T): any;
  <T>(value: T): any;
  methodA<T>(value: T): any;
  // 型パラメータは複数あってもよい
  methodB<T, U>(value: T): U;
};

// 関数型
let func: <T>(array: T[]) => T;
// 使い方
func = <T>(array: T[]) => array[0];
func<number>([1, 2, 3]);

// コンストラクタ型
let ctor: new <T>(value: T) => any;
// 使い方
ctor = SampleA;
new ctor<string>("str");

// type alias
type SampleC<T> = { data: T; };
let objC: SampleC<number> = { data: 1 };

export { SampleA, objA, SampleB, objB, obj, SampleC, objC }
```

　この中でよく使うのは、クラスとインタフェースとメソッドシグニチャの書き方でしょう。まずはこの 3 パターンの書き方を覚えておくべきです。

型パラメータと制約

　型パラメータには満たすべき制約を設けることができます。例を見てみましょう（リスト 3.28）。

リスト 3.28: 型パラメータ T は Base を継承していなければならない

```
class Base {
  str: string;
}
class InheritA extends Base {
  num: number;
}

// T は Base を継承済の型でなければならない制約
interface Sample<T extends Base> {
  method(): T;
}

// これは OK InheritA は Base を継承している
let objA: Sample<InheritA>;

// これはダメ RegExp は Base を継承していない
// error TS2344: Type 'RegExp' does not satisfy the constraint 'Base'.
// Property 'str' is missing in type 'RegExp'.
// let objB: Sample<RegExp>;

// これは OK 指定したオブジェクト型リテラルは Base クラスの要件を満たす
let objC: Sample<{ str: string; }>;

interface Service<T> {
  service(t: T): T;
}

// F-Bounded Polymorphism の例
// 制約の内容に自分自身の参照を含む 極稀に使う
function f<T extends Service<T>>(x: T) {
  return x.service(x);
}

export { Base, InheritA, Sample, objA, objC, Service, f };
```

型パラメータが満たすべき制約を extends の形式で指定できます。これにより、T に何が指定されようとも、Base に存在するプロパティには安全にアクセスできることがわかります。

自分でジェネリクス有りのコードを書く

ジェネリクスを使いこなすにあたり、一番難しいのは使うことではなく、使わせることです。なぜならば、ジェネリクスを提供するコードというのは、何かしらの要素を抽象的なまま扱わねばならないからです。たとえば、"○○の Array"のように、型パラメータ部分が何になっても上手く動くような設計です。

逆にいうと、実際に使うときには具体化しなければならないわけで、ジェネリクス有りのコードは"必ず何かと組み合わせて具体化する"必要があります。これを上手に使いこなすには一段上の設計力が要求されます。

通常の範囲では自分でジェネリクスを提供するコードを作る機会はさほど多くはありません。ですが、そこができるようになったらだいぶ型に慣れ親しんできたといえます。

3.7 "ありえない"型（The Never Type）

ありえないなんてことはありえない！ はずだけれど、"ありえない"ことを示す型があります。到達不可能なコードは never 型となります。具体例を見てみましょう（リスト 3.29）。

リスト 3.29: "ありえない"ことが型として示される

```
let str = "TypeScript";
if (typeof str === "number") {
  // string 型な変数の値が number だったら… ありえん！ never！
  // error TS2339: Property 'toUpperCase' does not exist on type 'never'.
  str.toUpperCase();
}

function test(): never {
  // return ないし関数のおしりに到達できないので返り値の型は never になる
  throw new Error();
}

let obj: never = test();
// ここに到達する事は…ありえん！
```

```
// error TS2339: Property 'test' does not exist on type 'never'.
obj.test();

export { }
```

変数の型が string なのに if 文では number でなければ突入できない…。関数を実行すると必ず例外が発生する…。そんな、到達できないコードでは never 型が利用されます。コードを書いていて never 型が必要になったり、コード上に現れることは少ないです。基本的には never 型を見かけることがあったら、何かミスをしているな…と考えたほうがよいでしょう。

第4章

アドバンスド型戦略

型のうち、難しいけど便利な話や、あまり関わりたくないけど実用上たまにお世話になる内容を解説していきます。タプル型（tuple types）や直和型（union types）についての解説もありますよ！ なお、普段書くコードではこの章で出てくる内容をなるべく使わずに済む設計こそよい設計だと筆者は考えています[*1]。

TypeScript でコードを書く中で、JavaScript で書かれたコードを型定義ファイルを介して扱う場面があります。そういったときに本章の内容が活きてくる場面があるでしょう。しかし、本章で書かれた内容を活かさないと上手く扱えない JavaScript コードは、元々の品質が微妙なコードだと考えてよいでしょう。

4.1 直和型（Union Types）

はい、皆様待望の機能でございます。"名前を言ってはいけないあの界隈"がよく使う用語を使って解説しないといけないのでビクビクですね。

一番最初に書いておくけど **TypeScript のコード書くときに積極的に使うものじゃあないぞ！！** という感じなんですが、`--strictNullChecks` オプションを使う場合に避けて通れない要素であるためしっかり覚えましょう。

では解説していきましょう。union types はいわゆる直和型です。たとえば `string | number | boolean` という型注釈があった場合、この変数の値は string か、number か、boolean かのどれか！ ということを表します。

なんのために直和型が TypeScript に導入されたかというと、まずは既存 JavaScript によりよい型定義を与えるためでしょう。そして null や undefined、string literal types など TypeScript の中でも適用領域が広がっています。JavaScript という現実と安全な世界を構築する TypeScript の橋渡しを上手にしてくれる機能といえます。

[*1] 本章で触れる機能を使うほうがよい場合もあります。たとえば構文木の構築・分解時などです。自分の用途に本当にそれが必要かはよくよく考えてみてください

ちなみに自分で TypeScript コード書いてるときにあまり欲しくなる機能ではありません。

まずは簡単な例から見ていきましょう（リスト 4.1）。

リスト 4.1: 型 A | 型 B で A か B のどちらかを表す

```
let a: string | boolean | undefined;
// string | boolean なので次はオッケー！
a = "str";
a = true;
// number はアカン。
// error TS2322: Type 'number' is not assignable
//   to type 'string | boolean | undefined'.
// a = 1;

// b1 と b2 を合体させてみよう
let b1: string | boolean | undefined;
let b2: boolean | number | undefined;
// c の型は string | number | boolean | undefined となる
let c: typeof b1 | typeof b2;

export { b1, b2, c }
```

型注釈を書く際に複数の型を｜で区切って書けます。既存の JavaScript ライブラリだとこういった、返り値の型が複数ある困った関数がかなりあります。あとは普通に TypeScript を書いているときでも SyntaxTree とかをコードから構築するときにはあったほうが便利かもしれません。

ご覧のとおり、union types 中の型の順番とかは関係ない（交換可能）し、union types の union types などは合体させてひとつの union types にできます。

TypeScript を自然に書いていて、union types を目にする機会は 3 種類あります。|| 演算子を使ったとき、条件（三項）演算子を使ったとき、配列リテラルを使ったときです（リスト 4.2）。

リスト 4.2: こういうときは目にしますね

```
// and の型は string | boolean
let and = "str" || true;
// cond の型は number | string
let cond = true ? 1 : "str";
// array の型は (number | boolean | string)[]
```

```
let array = [1, true, "str"];

export { and, cond, array }
```

一番よくお目にかかるのは配列リテラルでしょうか。TypeScript のベストプラクティスとして 1 つの配列で複数の型の値を扱わないほうが堅牢なコードになるため、きれいなコードを書いている限りはあまり見ないかもしれません。

型注釈として関数を与えるときは記法にちょっと気をつけないとコンパイルエラーになります（リスト 4.3)。

リスト 4.3: 型名をカッコで囲うんです？

```
// 引数無しの返り値 string な関数  な型注釈
let func: () => string;

// 素直に考えるとこう書けてもいいっしょ！ でもダメ！
// let a: () => string | () => boolean;

// 型名をカッコでくくる必要がある。これなら OK
let b: (() => string) | (() => boolean);
// もしくはオブジェクト型リテラル使う
let c: { (): string; } | { (): boolean; };

// union types じゃないときでも使えるけど見づらいな！
let d: (() => string);

export { func, b, c, d }
```

読みづらいコードになってしまいました。型にも適切な名前をつけることの重要さが偲ばれます。

union types な値を使うときは、型アサーションを使うこともできますがなるべくなら避けましょう（リスト 4.4)。

リスト 4.4: 一応使えるよ こうすれば

```
// 注意！ ここでやってるやり方よりも type guards を使うんだ…！！
// 型アサーションは悪い。常に悪い。なるべく使わないこと。
```

```
let obj: string | number | Date = null as any;

// string 扱いしてみる
(obj as string).charAt(0);

// number 扱いしてみる
(obj as number).toFixed(2);

// Date 扱いしてみる
(obj as Date).getTime();

// 値の集合に含まれない型にしてみると普通に怒られる
// error TS2352: Type 'string | number | Date' cannot be converted to type 'RegExp'.
//   Type 'Date' is not comparable to type 'RegExp'.
//     Property 'exec' is missing in type 'Date'.
// (<RegExp>obj).test("test");

export { }
```

union types を相手にする場合は、次に説明する「4.2 型の番人（Type Guards）」を使いましょう。話はそれからだ！

4.2 型の番人（Type Guards）

type guards は、union types が導入されたことで変数の型が一意ではなくなってしまったため、それを自然に解決するために導入された仕組みです。type guards は"変数 A が○○という条件を満たすとき、変数 A の型は××である"というルールを用いて、ガード（番人となる条件式など）の後の文脈で変数の型を××に狭めることができます。

処理フローに基づく型の解析（Control Flow Based Type Analysis）

さて、トップバッターがいきなり公式に type guards の一員なのか怪しいのですがいってみましょう。名前が長いですが、要するに普通にコードを書いていった時に、値の型を判別するコードは分岐にしたがって変数の型が絞り込まれるというものです。

例を見ていきましょう。TypeScript を書いていて一番対処を迫られる union types のパターンはおそらく `T | undefined` のような、何か＋ `undefined` の形式でしょう。if 文を用

いて undefined の値について対処してみます（リスト 4.5)。

リスト 4.5: undefined の可能性を潰す

```
function upperA(word?: string) {
  // word は省略可能引数なので string | undefined
  // ここで word をいきなり使おうとするとエラーになる
  // Object is possibly 'undefined'.
  // word.toUpperCase();

  if (word == null) { // word が null か undefined の時
    // undefined の可能性を string で上書き！
    word = "TypeScript";
  }

  // undefined の可能性を潰したのでこの時点で word は string 確定！
  console.log(word.toUpperCase());
}

function upperB(word?: string) {
  // 別解：JS で || 演算子は最初に truthy になった値を返す
  // ので、undefined（falsy）な時は "TypeScript" で上書きされる
  word = word || "TypeScript";

  // undefined の可能性を潰したのでこの時点で word は string 確定！
  console.log(word.toUpperCase());
}

function upperC(word = "TypeScript") {
  // TypeScript 的に一番素直なパターン
  console.log(word.toUpperCase());
}

export { upperA, upperB, upperC }
```

もう一例見てみましょう。引数に string と string[] を取り、これを string[] に統一して利用します（リスト 4.6)。

リスト 4.6: 変数の型を統一していく

```
function upperAll(words: string | string[]) {
  if (typeof words === "string") {
    // string なら string[] に変換する
```

```
    words = [words];
  }

  // この時点では words は string[] に揃えられる
  return words.map(word => word.toUpperCase());
}

console.log(upperAll("TypeScript"));
console.log(upperAll(["TypeScript", "JavaScript"]));

export { }
```

変数のプロパティに対しても type guards は利用可能です（リスト 4.7）。コンパイラの実装を想像すると、なにげに大変そうなことをやっていて思わず感心してしまいます。

リスト 4.7: 変数のプロパティも絞り込める

```
interface Foo {
  value: number | string;
}

let foo: Foo = {
  value: "TypeScript",
};

// number | string では toUpperCase があるか確定できない
// error TS2339: Property 'toUpperCase' does not exist on type 'number | string'.
// foo.value.toUpperCase();

// 変数直だけではなくて、変数のプロパティでも type guards が使える
if (typeof foo.value === "string") {
  // ここでは foo.value は string に絞りこまれている！ 一時変数いらない！
  foo.value.toUpperCase();
}

export { }
```

最後に、関数が絡んだ場合の例を見ておきます（リスト 4.8）。関数の内側と外側では、処理フローは別世界です。関数はいつ実行されるかわからないため、変数の再代入が可能な場合、関数の内側で別途絞込みを行う必要があります。一方、const を使うと変数の値を変える

ことができないため、この問題を回避できる場合があります。

リスト 4.8: 関数の外側でのフローは内側では関係ない

```
let v1: string | number;
if (typeof v1 === "string") {
  let f = () => {
    // これはエラーになる！
    // プログラムの字面的には string に確定されていそう…
    // しかし、関数はいつ実行されるかわからない
    // error TS2339: Property 'toUpperCase'
    //   does not exist on type 'string | number'.
    console.log(v1.toUpperCase());
  };
  // ここでは v はまだ string
  f();

  // ここで v が number に！
  v1 = 1;
  f();
}

// let ではなくて const を使うと…
const v2: string | number = null as any;
if (typeof v2 === "string") {
  let f = () => {
    // v2 の中身が入れ替えられる可能性はないのでエラーにならない
    console.log(v2.toUpperCase());
  };
  f();

  // const なので再代入しようとするとエラーになる
  // error TS2450: Left-hand side of assignment expression
  //   cannot be a constant or a read-only property.
  v2 = 1;
}
```

さて、次項意向でどういう処理が絞り込みに繋がるのかの例を見ていきます。

typeof による Type Guards

JavaScript の `typeof` は指定した値がどういう性質のオブジェクトかを調べ、文字列で返す演算子です。ECMAScript 5 の範囲では、変換ルールは次のとおりです。

- string のときは"string"を返す
- boolean のときは"boolean"を返す
- number のときは"number"を返す
- undefined のときは"undefined"を返す
- 関数として呼び出し可能な場合は"function"を返す
- それ以外の場合（null を含む！）は"object"を返す

これを利用して、変数の型を狭めます。

一番簡単な使い方から見ていきましょう（リスト 4.9)。TypeScript の type guards では typeof の結果が string、boolean、number の場合、その型に絞り込むことができます。

リスト 4.9: 実際の型がわからないなら調べるしかないじゃない！

```
let obj: number | string = null as any;
if (typeof obj === "string") {
  // ここでは string と確定されている！
  obj.charAt(0);
} else {
  // ここでは消去法で number と確定されている！
  obj.toFixed(2);
}

export { }
```

変数 obj を typeof で調べたときに値が string だったので、変数 obj の型は string である、という具合に絞りこまれています。

もう一例見てみましょう。リスト 4.10 では、any や number と指定された変数を type guards で string に絞り込んでいます。

リスト 4.10: 変なコードを書くとコンパイラが教えてくれる

```
let objA: any;
if (typeof objA === "string") {
  // ここでは string と確定されている！
  // number にしか存在しないメソッドを呼ぶとコンパイルエラー！
  // error TS2339: Property 'toFixed' does not exist on type 'string'.
  objA.toFixed(0);
}

let objB: number = 1;
if (typeof objB === "string") {
  // "ありえない" パターンだと never 型になり怒られる
  // error TS2339: Property 'toFixed' does not exist on type 'never'.
  objB.toFixed(0);
}
```

この操作を行うと"ありえない"ことを表す never 型になるため、`obj.toFixed(0)` という string には存在しないメソッドの呼び出しはコンパイルエラーとなります。

うーん、便利ですね。変数に指定した型どおりの値が入ってくるのが健全なので、コンパイル時にミスが発見されるのは嬉しいことです。

instanceof による Type Guards

typeof でしか type guards が使えないと辛いので、instanceof を使った type guards も、もちろんあります。

JavaScript における instanceof は、ある値が指定した関数のインスタンスであるかを調べる演算子です。プロトタイプチェーンも遡ってみていくので、親子関係にある場合もインスタンスかどうかを調べることができます。

動作例を確認してみましょう（リスト 4.11）。

リスト 4.11: instanceof の挙動

```
class Base {
}

class InheritA extends Base {
}
class InheritB extends Base {
```

```
}

let obj = new InheritA();

// true と表示される
console.log(obj instanceof Base);
// true と表示される
console.log(obj instanceof InheritA);
// false と表示される
console.log(obj instanceof InheritB);

// 無理矢理親を差し替える！
InheritA.prototype = new InheritB();
obj = new InheritA();
// true と表示される
console.log(obj instanceof InheritB);

export { }
```

オブジェクトの prototype と一致するか順番どおり見ていくだけですね。

instanceof で型を絞り込みます（リスト 4.12)。

リスト 4.12: instanceof の挙動

```
class A {
  str: string;
}
class B {
  num: number;
}
class C extends A {
  bool: boolean;
}

let obj: A | B | C = null as any;
if (obj instanceof A) {
  // ここでは A（含む C）と確定している
  obj.str;
  if (obj instanceof C) {
    // ここでは C と確定している
    obj.bool;
  }
}
```

```
if (obj instanceof C) {
  // ここでは C と確定している
  obj.bool;
} else {
  // ここではまだ A | B
  if (obj instanceof B) {
    // ここでは B と確定している
    obj.num;
  } else {
    // ここでは A と確定している
    obj.str;
  }
}

export { }
```

昔の TypeScript と違って、instanceof の else 句でも型の絞り込みが行われます。挙動として納得感があり大変よいですね。

ユーザ定義の Type Guards（User-defined Type Guards）

ユーザが定義した関数によって、ある値がなんの型なのかを TypeScript コンパイラに教える方法があります（リスト 4.13）。型判別用の関数を作成し、そこで返り値に仮引数名 `is` 型名という形式で判別結果を指定します。この書き方をした場合、返り値は boolean でなければなりません。

リスト 4.13: ユーザ定義の type guards

```
class Sample {
  str: string;
}

// 構造的部分型！
let obj: Sample = {
  str: "Hi!",
};

// 独自に Sample 型である事の判定を実装する
function isSample(s: Sample): s is Sample {
  if (!s) {
```

```
    return false;
  }
  // とりあえず、str プロパティがあって値が string なら
  // Sample 型に互換性あり！ という基準にする
  return typeof s.str === "string";
}

if (isSample(obj)) {
  console.log(obj.str);
}

export { }
```

面白い記法として、is の左辺に this を用いることもできます（リスト 4.14）。

リスト 4.14: is の左辺に this を使う

```
abstract class Node {
  isStringNode(): this is StringNode {
    return this instanceof StringNode;
  }
  isNumberNode(): this is NumberNode {
    return this instanceof NumberNode;
  }
}

class StringNode extends Node {
  constructor(public text: string) {
    super();
  }
}

class NumberNode extends Node {
  constructor(public value: number) {
    super();
  }
}

let nodes: Node[] = [new StringNode("TypeScript"), new NumberNode(8)];
// TypeScript と 8 と表示される
nodes.forEach(n => {
  if (n.isStringNode()) {
    // n is StringNode!
```

```
    console.log(n.text);
  } else if (n.isNumberNode()) {
    // n is NumberNode!
    console.log(n.value);
  }
});

export { }
```

引数として渡された値の型名を明示する代わりに、this の型を指定するわけです。これも利用する機会は少なさそうですが、ツリー状の構造を作るときなどに活躍しそうです。

Type Guards と論理演算子

type guards は&&とか||とか?とか!とかの論理演算子にもちゃんと対応しています（リスト 4.15)。

リスト 4.15: ブール代数みたいな演算に対応してる

```
let obj: number | boolean | string = null as any;

// &&演算子で絞込み
typeof obj === "string" && obj.charAt(0);
// 次のようなコードはエラーになる！
// error TS2339: Property 'charAt' does not exist on type 'number'.
// typeof obj === "number" && obj.charAt(0);

// ||演算子で union types に
if (typeof obj === "string" || typeof obj === "boolean") {
  // string | boolean に絞り込まれる
} else {
  // 消去法で number！
}

// 三項演算子は普通に if 文と一緒の挙動
typeof obj === "string" ? obj.charAt(0) : obj;
// 次と等価
if (typeof obj === "string") {
  obj.charAt(0);
} else {
  obj;
```

```
}

// 一応、否定演算子にも対応している
if (!(typeof obj !== "string")) {
  // 否定の否定は普通にそのまんま string だな！ ちゃんと絞り込まれます
  obj.charAt(0);
}

export { }
```

あんまり使わないかもしれませんが、他の人がこの書き方を使った時に戸惑わぬよう頭の片隅にはとどめておいたほうがよいかもしれません。

Type Guards の弱点

type guards は型システム上の仕組みだということを忘れてはいけません。JavaScript の実行環境とは全く関係がないのです。

TypeScript では構造的部分型の仕組みにより、クラスが要求されている箇所に互換性のある別の値を代入できます。

その仕組みを使って、リスト 4.16 のようなコードが書けてしまいます。

リスト 4.16: 構造的部分型と type guards

```
class Sample {
  str: string;
}

// 構造的部分型！
let obj: Sample = {
  str: "Hi!",
};

if (obj instanceof Sample) {
  // 型は Sample に絞られている しかし、絶対に到達しない
  // 現在の obj は Sample を親に持たない
  console.log(obj.str);
}

export { }
```

obj は Sample を型として持ち、その値として互換性のあるオブジェクトリテラルを持っています。コンパイル後の JavaScript コード（リスト 4.17）を見ると、obj の値が Sample クラスのインスタンスではないことが一目瞭然ですが、TypeScript 上で見ると型を元に判別されていると勘違いしやすいことを頭の片隅においておきましょう。

リスト 4.17: コンパイル後の JS

```
"use strict";
class Sample {
}
// 構造的部分型！
let obj = {
    str: "Hi!",
};
if (obj instanceof Sample) {
    // 型は Sample に絞られている しかし、絶対に到達しない
    // 現在の obj は Sample を親に持たない
    console.log(obj.str);
}
```

これを回避する方法がいくつかあります。

ひとつ目は、ユーザ定義の type guards を使う方法。ふたつ目は private な要素をクラスに突っ込んでしまうことです（リスト 4.18)。

リスト 4.18: private な要素があれば構造的部分型で値を偽造できない

```
class Sample {
  str: string;
  private _tmp: any;
}

// private なインスタンス変数があるクラスのインスタンスは偽造できない！
// error TS2322: Type '{ _tmp: null; str: string; }' is not
//     assignable to type 'Sample'. Property '_tmp' is private
//     in type 'Sample' but not in type '{ _tmp: null; str: string; }'.
let obj: Sample = {
  str: "Hi!",
  _tmp: null,
};
```

色々書きましたが、一番の解決策は union types や any を多用せず、真っ当なコードを書けるよう設計することです。

4.3 交差型（Intersection Types）

union types に似た記法の intersection types（交差型）です。intersection types は 2 つの型を合成し、1 つの型にできます。union types と違って利用頻度は低く、TypeScript 的に使いたくなるシチュエーションもほとんどありません。

まずは例を見てみましょう。ある関数に渡したオブジェクトを拡張し、新しいプロパティやメソッドを生やします（リスト 4.19)。

リスト 4.19: 型を合成する

```
interface Storage {
  $save(): void;
}

function mixinStorage<T>(base: T): T & Storage {
  let modified = base as any;
  modified.$save = () => {
    // めんどいので保存したフリ
    console.log(`データを保存しました！  ${JSON.stringify(base)}`);
  };

  return modified;
}

// 何の変哲もないオブジェクト
let base = {
  name: "TypeScript",
};
// を、Storage を合成する関数に渡す
let obj = mixinStorage(base);

// base に存在しないメソッドが呼べる！
// データを保存しました！  {"name":"TypeScript"} と表示される
obj.$save();

// もちろん、base にあったプロパティにもアクセスできる
```

```
obj.name = "JavaScript";
// データを保存しました！  {"name":"JavaScript"} と表示される
obj.$save();

export { }
```

intersection types を使うと、型定義ファイルが書きやすくなる場合があります。例を見てみます（リスト 4.20）。intersection types を使わない書き方と intersection types を使った書き方、どちらのほうが理解しやすいでしょうか？

リスト 4.20: 型の合成で素直な定義を作る

```
// intersection types を使わない書き方
declare namespace angular.resource1 {
  interface ResourceProvider {
    create<T extends Resource<any>>(): T;
  }

  interface Resource<T> {
    $insert(): T;
  }
  let $resource: ResourceProvider;
}
// 上の定義を使ってみる
namespace sample1 {
  interface Sample {
    str: string;
  }
  // SampleResource という型を 1 つ無駄に作らねばならぬ
  // なぜこれで動くのか、トリックがわかるだろうか？
  interface SampleResource extends Sample, angular.resource1.Resource<Sample> { }

  let $obj = angular.resource1.$resource.create<SampleResource>();
  $obj.str = "test";
  let obj = $obj.$insert();
  console.log(obj.str);
}

// intersection types を使った書き方
declare namespace angular.resource2 {
  interface ResourceProvider {
    create<T>(): T & Resource<T>;
```

```
  }

  interface Resource<T> {
    $insert(): T;
  }
  let $resource: ResourceProvider;
}
// 上の定義を使ってみる
namespace sample2 {
  interface Sample {
    str: string;
  }

  // 超簡単…！！
  let $obj = angular.resource2.$resource.create<Sample>();
  $obj.str = "test";
  let obj = $obj.$insert();
  console.log(obj.str);
}

export { sample1, sample2 }
```

intersection types を使いこなした書き方のほうが、圧倒的に謎が少なく素直に書けています。

4.4 文字列リテラル型（String Literal Types）

文字列リテラルを型として使える機能です。パッと読んだだけでは、意味がわからないですね。まずは例を見てみましょう（リスト 4.21）。

リスト 4.21: カードのスートを型として表す

```
// "文字列" が 型 です。値ではない！
let suit: "Heart" | "Diamond" | "Club" | "Spade";

// OK
suit = "Heart";
// NG suit の型に含まれていない
// error TS2322: Type '"Joker"' is not
//   assignable to type '"Heart" | "Diamond" | "Club" | "Spade"'.
```

```
// suit = "Joker";

export { }
```

文字列が型というのは見慣れないとすごく気持ちが悪いですね。しかし、この機能は TypeScript が JavaScript の現実と折り合いをつける上で重要な役割があります。たとえば、DOM の addEventListener などです。指定するイベント名によって、イベントリスナーの型が変わります（リスト 4.22)。

リスト 4.22: イベント名によって型が変わる

```
// lib.dom.d.ts から抜粋
// 第一引数で指定するイベントによってリスナーで得られるイベントの型が違う
interface HTMLBodyElement extends HTMLElement {
  addEventListener(
    type: "change",
    listener: (this: this, ev: Event) => any,
    useCapture?: boolean): void;
  addEventListener(
    type: "click",
    listener: (this: this, ev: MouseEvent) => any,
    useCapture?: boolean): void;
  addEventListener(
    type: "keypress",
    listener: (this: this, ev: KeyboardEvent) => any,
    useCapture?: boolean): void;
  addEventListener(
    type: string,
    listener: EventListenerOrEventListenerObject,
    useCapture?: boolean): void;
}
```

これにより、自然に TypeScript でコードを書くだけでリスナーで受け取れるイベントの型が自動的に適切なものに絞りこまれます。こんなものが必要になってしまう JavaScript の複雑さよ…。

また union types と文字列リテラル型を組み合わせ、switch で条件分岐ができます（リスト 4.23)。

リスト 4.23: Union Types は switch でえこひいきされている

```
// 足し算
interface Add {
  type: "add";
  left: Tree;
  right: Tree;
}
// 末端の値
interface Leaf {
  type: "leaf";
  value: number;
}

type Tree = Add | Leaf;

// (10 + 3) + 5 を表現する
let node: Tree = {
  type: "add",
  left: {
    type: "add",
    left: { type: "leaf", value: 10 },
    right: { type: "leaf", value: 3 },
  },
  right: {
    type: "leaf",
    value: 5,
  },
};

// 18 と表示される
console.log(calc(node));

function calc(root: Tree): number {
  // プロパティの値で型の絞込ができる！
  switch (root.type) {
    case "leaf":
      // 型は Leaf で決定！
      return root.value;
    case "add":
      // 型は Add で決定！
      return calc(root.left) + calc(root.right);
    default:
      throw new Error("unknown node");
  }
}
```

```
export { }
```

switch 文による type guards（後述）は TypeScript 2.1.0 からのサポートが予定されているので、現時点ではえこひいきされていますね。

なお、執筆時点でアンダース・ヘルスバーグ御大が *Number, enum, and boolean literal types* という pull request を作成、作業しています[*2]。

4.5 型の別名（Type Alias）

最初に書いておきます。 **可能な限り type alias を使うな！ interface 使え！** 筆者は type alias の乱用を恐れています！

type alias も union types の扱いを便利にするために導入された機能です。機能としてはただ単に型をひとまとまりにして、それに名前が付けられるだけです。それだけです。

type alias は仕様上、interface と同じように利用できる場面もあります。ですが、基本的に type alias は interface より機能が貧弱であるため、なるべく避けるべきです。

代表例を見てみましょう（リスト 4.24）。

リスト 4.24: 頻出する union types に名前をつける

```
type FooReturns = string | number | boolean;

interface Foo {
  bar(): FooReturns;
  buzz(): FooReturns;
  barbuzz(): FooReturns;
}
```

わかりやすいですね。1 ヶ所変更すると、関連箇所がすべて更新されるのも便利です。

tuple types に名前をつけることもできます（リスト 4.25）。

リスト 4.25: tuple types に名前をつける

[*2] https://github.com/Microsoft/TypeScript/pull/9407

```
// tuple types に名前をつける
type Point = [number, number];
type Circle = [Point, number];

let c: Circle = [[1, 2], 3];

// でも、こっちのほうが TypeScript としては適切よね
namespace alternative {
  class Point {
    constructor(public x: number, public y: number) {
    }
  }
  class Circle {
    constructor(public p: Point, public r: number) {
    }
  }
  let c2: Circle = new Circle(new Point(1, 2), 3);
  console.log(c2.p, c2.r);
}

export { Point, Circle, c, alternative }
```

こちらは素直にクラスでやればいいのに、という感じです。

type alias は型に別名をつけるだけで、コンパイルされると消えてしまう存在です。そのため、リスト 4.26 のようなコードは書くことができません。

リスト 4.26: type alias は値を作らない

```
// 型の別名を作るだけで何かの値を作るわけではない…！
type StringArray = string[];

// なのでこういうことはできない
// error TS2304: Cannot find name 'StringArray'.
let strArray = new StringArray();
```

TypeScript の仕様書にのっている type alias の利用例について interface での書き換えができるものを示します（リスト 4.27)。

リスト 4.27: interface を使うんだ！

```
// これらは interface で表現不可 type alias で正解
type StringOrNumber = string | number;
type TextObject = string | { text: string };
type Coord = [number, number];
type ObjectStatics = typeof Object;
type Pair<T> = [T, T];
type Coordinates = Pair<number>;
type Tree<T> = T | { left: Tree<T>, right: Tree<T> };

// これらは interface で表現可能
type HolidayLookup = Map<string, Date>;
interface AltHolidayLookup extends Map<string, Date> {
}

type Callback<T> = (data: T) => void;
interface AltCallback<T> {
  (date: T): void;
}

type RecFunc = () => RecFunc;
interface AltRecFunc {
  (): AltRecFunc;
}

export {
  StringOrNumber, TextObject, Coord, ObjectStatics, Pair,
  Coordinates, HolidayLookup, AltHolidayLookup, Callback, AltCallback,
}
```

union types が絡むもの、tuple types が絡むもの、型クエリが絡むものだけが interface で置き換えることができません。

最後に type alias ではなくインタフェースを使ったほうがいい理由を掲げておきます。

- interface のコンパイルエラーには interface 名が表示されてわかりやすい
  - type alias は展開して表示されちゃうので無理
- interface は定義の統合ができるので後から自由に拡張できる
  - type alias は無理

interface でできることを type alias でやるな！

4.6 多態性のある this 型（Polymorphic 'this' Type）

this を型として用いることができます。たとえばリスト 4.28 のようなコードです。

リスト 4.28: this を型として用いる

```
// 自分自身を型として表す時、this を利用する
class A {
  _this: this;
  a(): this {
    return this;
  }

  d(arg: this): this {
    return arg;
  }

  e() { // this を return した場合暗黙的に返り値も this となる
    return this;
  }
}

class B extends A {
  b() {
    console.log("B");
  }
}

interface C extends A {
  c(): void;
}

// a() はクラス A のメソッドだが返り値の型は B 自身だ！
new B().a().e().b();

// d() もクラス A のメソッドだが引数は B でなければならぬ
new B().d(new B()).b();

// d() はクラス A のメソッドだが、B に生えている限り A を渡したら怒られてしまう
// error TS2345: Argument of type 'A' is not assignable to parameter of type 'B'.
//   Property 'b' is missing in type 'A'.
// new B().d(new A()).b();
```

```
// プロパティについても同様に B 自身になる
new B()._this.b();

// インタフェースでも OK C 自身になる
let c: C = null as any;
c.a().c();

export { }
```

this を型として記述するという発想がすごいですね。引数や返り値の型として this を利用しています。fluent な、メソッドチェーンで使う API を組み立てる場合に役立ちそうです。

この書き方がないと、ジェネリクスなどを使ってごまかさなければならないところでしょう。とはいえ、便利になる代わりに仮引数に対して使ったりすると無駄に制約がきつくなったりする場合があるため、乱用は控えましょう。`return this;` を使った時に、メソッドの返り値が暗黙的に `this` になるのを利用する、くらいがよい塩梅かもしれません。

4.7 関数の this の型の指定（Specifying This Types For Functions）

JavaScript では `Function.prototype.bind` や `Function.prototype.call`、`Function.prototype.apply` などの関数により、関数呼び出し時の this の値の型を変更できます。この仕様は悪しき仕様だと筆者は思いますが、jQuery や DOM など、古めの API ではこの仕様を API として組み込んだものが存在しています。TypeScript ではこの変更も頑張ってサポートしようとしています。

まずは簡単な例を見てみます（リスト 4.29)。関数の 1 つ目の仮引数の名前を `this` にするだけです。

リスト 4.29: this の型を指定する

```
// 関数内部での this の型を偽の第一引数で指定
function testA(this: string) {
  console.log(this.toUpperCase());
}

// こういう利用を想定しているはず
```

```
// TYPESCRIPT と表示される
testA.bind("TypeScript")();

// 普通に呼び出すとエラーになる
// error TS2684: The 'this' context of type 'void'
//   is not assignable to method's 'this' of type 'string'.
// testA();

// 1 つ目の仮引数が this の型指定だった場合、それは偽物の仮引数
// 実際に何かを渡すとエラーになってしまう
// error TS2346: Supplied parameters do not match any signature of call target.
// testA("TypeScript");

function testB() {
  // --noImplicitThis オプション利用時、関数内で this にアクセスすると怒られる
  // error TS2683: 'this' implicitly has type 'any'
  //   because it does not have a type annotation.
  // console.log(this.toUpperCase());
}

function testC(this: string, postfix: string) {
  console.log(`${this.toUpperCase()}${postfix}`);
}
// TYPESCRIPT! と表示される
testC.bind("TypeScript")("!");

export { testB }
```

this の値がすり替えられるときの挙動に対応できています。--noImplicitThis オプションを利用すると、this の型指定がない関数内で this へアクセスするとエラーになります。this を使わない限りはエラーにならないため、常用してしまってよいでしょう。

この仕様が現実世界でどう役に立つかを紹介します（リスト 4.30）。

リスト 4.30: this の値が差し替えられる API に対応

```
// lib.dom.d.ts から抜粋
// listener の仮引数の先頭が偽の仮引数で、this の型の指定が行われている
interface HTMLBodyElement extends HTMLElement {
  addEventListener(
    type: "click",
    listener: (this: this, ev: MouseEvent) => any,
```

```
    useCapture?: boolean): void;
  addEventListener(
    type: string,
    listener: EventListenerOrEventListenerObject,
    useCapture?: boolean): void;
}

let el1: HTMLBodyElement = null as any;
el1.addEventListener("click", function() {
  // this の型は HTMLBodyElement
  this.innerText = "Hi!";
});
el1.addEventListener("click", () => {
  // アロー関数の場合 this の値は変えられない
  // error TS2683: 'this' implicitly has type 'any'
  //   because it does not have a type annotation.
  // this.innerText = "Hi!";
});

let el2: HTMLDivElement = null as any;
el2.addEventListener("click", function() {
  // this の型は HTMLDivElement
  this.innerText = "Hi!";
});

export { }
```

イベント発生時のコールバック関数で this が差し替えられる場合に対応できています。自分で TypeScript コードを書く時に必要になる場合は少なくありたいものです。しかし、型定義ファイルを作成する時にはお世話にならざるをえないときがあるでしょう。

4.8 ローカル型（Local Types）

ローカル型は通常より小さい範囲で、クラスやインタフェースや enum や type alias を定義できます（リスト 4.31）。

リスト 4.31: ローカル型を試す

```
{
  type Data = string | boolean;
```

```
  let obj: Data = true;

  console.log(obj);
}
{
  type Data = number | Date;
  let obj: Data = 1;

  console.log(obj);
}

// ブロックスコープの外ではもはや Data 型を参照することはできない
// error TS2304: Cannot find name 'Data'.
// let obj: Data;

{
  // クラス、enum、Buzz なども
  class Foo { }
  enum Bar {
    a,
    b,
  }
  interface Buzz { }

  console.log(Foo, Bar.a, null as any as Buzz); // 警告消し
}
// もちろんブロックスコープの外では上記 3 つは参照できない

export { }
```

使う機会は少ないかもしれませんが、リスト 4.32 のようにメソッドの中で簡易に別名を用意したい場合などに利用できるでしょう。

リスト 4.32: メソッド内でだけ通用する別名

```
// 現実的な活用例
class Foo {
  method() {
    // メソッド内でのみ使える type alias !
    type Data = string | number;
    let obj: Data = 1;

    console.log(obj);
```

```
  }
}

export { Foo }
```

4.9　型クエリ（Type Queries）

　型クエリは指定した変数（やメソッドなど）の型をコピーします。たとえば、リスト 4.33 のようなクラスそのものを型として指定したい場合、それ専用の書き方は用意されていません。そういうときに型クエリを使います。

リスト 4.33: クラスそのものの型だよ！

```
class Sample {
  str: string;
}

// この書き方だと Sample のインスタンスになる Sample クラスそのものではない
let obj: Sample;
// Sample 自体の型をコピー！　つまりこれは Sample クラスそのものです
let clazz: typeof Sample;

// それぞれに当てはまる値は次のとおり なるほど
obj = new Sample();
clazz = Sample;

obj = new clazz();

// clazz を頑張って手で書くと次に等しい
let alterClazz: {
  new (): { str: string; };
};
alterClazz = clazz;
clazz = alterClazz;

export { }
```

　メソッドなどの値も取れますが、this を使うことはできないため、少しトリッキーなコー

ドになる場合もあります。リスト 4.34 の例は、prototype プロパティを使っているため JavaScript 力が多少ないと思いつかないかもしれません。

リスト 4.34: prototype を参照するとメソッドの型が取れる

```
class Sample {
  hello = (word = "TypeScript") => `Hello, ${word}`;
  bye: typeof Sample.prototype.hello;
}

let obj = new Sample();
obj.bye = obj.hello;

export { }
```

　型クエリはわざわざインタフェースを定義するのもめんどくさいけど…というときに使える場合があります。リスト 4.35 では、ひとつ目の引数の型をふたつ目の引数や返り値の型にもコピーして使っています。

リスト 4.35: ここまで複雑にするならインタフェース使って

```
// このコードは（死ぬほど読みにくいけど）正しい
function move(p1: { x1: number; y1: number; x2: number; y2: number; },
  p2: typeof p1,
): typeof p1 {
  return {
    x1: p1.x1 + p2.x1,
    y1: p1.y1 + p2.y1,
    x2: p1.x2 + p2.x2,
    y2: p1.y2 + p2.y2,
  };
}

let rect = move({
  x1: 1, y1: 1, // 無駄に多い
  x2: 2, y2: 2, // プロパティ
}, {
    x1: 3, y1: 3,
    x2: 4, y2: 4,
  });
rect.x1;
rect.x2;
```

```
export { }
```

ここまで来るとさすがに読みにくくなるのでインタフェースをひとつ定義したほうが断然いいですね。

4.10 タプル型（Tuple Types）

tuple（タプル）は、任意の数の要素の組です。JavaScript では tuple はサポートされていないため、TypeScript での tuple はただの Array で表現されます。

既存の JavaScript 資産を使おうとしたときに、配列の形で多値を返してくるライブラリが稀にあります。タプル型はそういったときに使うためのもので、TypeScript でコードを書く際に多用するものではないでしょう。というのも、普通にコードを書いている限りでは型推論の結果としてタプル型が出てこないためです。

タプル型は型（TypeScript）の世界にしか登場せず、コンパイル後の JavaScript コードでは消えてしまいます。記述方法は配列の型指定へ [typeA, typeB] のように配列の要素の代わりに型名を記述していくだけです。例を見てみましょう（リスト 4.36）。

リスト 4.36: 基本的な例

```
// まずは今までどおりの配列から
// これは別の箇所で解説している union types で表現され (number | string | boolean)[]
let array = [1, "str", true];

// {} は charAt を持たないので下記はコンパイルエラーになる
// array[1].charAt(0);

// tuple! 明示的な型の指定が必要
let tuple: [number, string, boolean] = [1, "str", true];

// string は charAt を持つ！
tuple[1].charAt(0);

// TypeScript の tuple types は普通に Array でもあるのだ
tuple.forEach(v => {
  console.log(v);
});
```

```
export { array }
```

各要素の型を指定すると、その要素の index でアクセスしたときに適切な型で扱われます。

もちろん、タプル型は Generics と組み合わせて利用できます（リスト 4.37）。

リスト 4.37: Generics での利用も可

```
// Generics を使って tuple を生成して返す
function zip<T1, T2>(v1: T1, v2: T2): [T1, T2] {
  return [v1, v2];
}

let tuple = zip("str", { hello(): string { return "Hello!"; } });
tuple[0].charAt(0); // おー、静的に検証される！
tuple[1].hello();   // おー、静的に検証される！

export { }
```

Good！ いいですね。

さて、タプル型について重箱の隅をつついていきましょう。要素数が多すぎる場合、指定されていない値の型は union types になります。その例を見てみましょう（リスト 4.38）。

リスト 4.38: 値の要素数が多すぎる場合

```
// 要素が多い分には OK だ！
let tuple: [string, number] = ["str", 1, "test"];

// 範囲外の要素の型はすべての要素の union、つまり string | number になる。
let value = tuple[2];

// 以下はダメ。true は string | number ではないため。
// tuple = ["str", 1, true];
```

お次は要素の順序がズレた場合、どうなるかを見てみましょう（リスト 4.39）。

リスト 4.39: 絶望に身をよじれ…！

```
let tuple: [string, number] = ["str", 1];

// 先頭を number に…
tuple.unshift(1);

// あぁっ！ 実行時エラー！
// Uncaught TypeError: undefined is not a function
tuple[0].charAt(0);

export { }
```

…悲しい結果になりました。[1, true] のような配列のリテラルをタプル型に推論しないのはおそらくこのためでしょう。

unshift や pop など、配列の要素を操作する方法は色々ありますが、後から prototype を拡張することすら可能な JavaScript では TypeScript コンパイラ側ですべてをキャッチアップすることは不可能です。タプル型を扱う場合は要素数を変更するような操作をしないほうがよいでしょう。

なるべくなら、タプルは使いたくないですね。

4.11 非 null 指定演算子（Non-null Assertion Operator）

非 null 指定演算子（!）は、指定した値が null や undefined ではないことを人力でコンパイラに教えてやるための記法です。基本的に、この演算子は使わないにこしたことはありません。新規にコードを書き起こすのであれば非 null 指定演算子は使わないほうがよいでしょう。

しかしながら、昔からメンテしている TypeScript コードについてはこの演算子に頼らざるをえない場合も多いです。--strictNullChecks オプションを有効にしたい場合、省略可能なプロパティでは undefined のチェックが必須になります。警告を低コストに抑制したい場合、非 null 指定演算子は有効な対処法となります。もちろん、将来的には徐々にリファクタリングしこの演算子の利用箇所を消滅させていくべきです。

例を見てみましょう（リスト 4.40）。

リスト 4.40: !演算子を使う

```
import * as fs from "fs";

interface Config {
  filePath?: string | null;
  verbose?: boolean;
}

// 呼び出し元で値をしっかり代入していても...
let config: Config = {};
config.filePath = "settings.json";
config.verbose = false;
processA(config);
function processA(config: Config = {}) {
  // 関数内部では Config のプロパティは undefined の可能性が排除できない…
  // よって、! で無理やりエラーを消す必要がある
  if (fs.existsSync(config.filePath!)) {
    console.log(fs.readFileSync(config.filePath!, "utf8"));
  }
}

function processB(config: Config = {}) {
  // 関数内で初期値を設定してやるとエラーを解消できる（かしこい）
  config.filePath = config.filePath || "settings.json";
  config.verbose = config.verbose || false;

  // 初期値設定済なので ! 不要
  if (fs.existsSync(config.filePath)) {
    console.log(fs.readFileSync(config.filePath, "utf8"));
  }

  // undefined ではなくした結果は関数をまたいで引き継がれない
  // 残念だが当然…
  processA(config);
}

// Config の undefined と null 無し版
interface ConfigFixed {
  filePath: string;
  verbose: boolean;
}

function processC(config: Config = {}) {
  // ? 除去版に値を詰め替える
  const fixed: ConfigFixed = {
```

```
    filePath: config.filePath || "settings.json",
    verbose: config.verbose || false,
  };

  if (fs.existsSync(fixed.filePath)) {
    console.log(fs.readFileSync(fixed.filePath, "utf8"));
  }
}

export { Config, processB, processC }
```

人間が undefined や null ではないと確信できる場合、エラーとなる箇所の末尾に!をつけていきます。非 null 指定演算子をなるべく使わない手段として使う前に初期値を代入する、undefined や null を含まない型の値に詰め直すなどが考えられます。他の方法も見てみます（リスト 4.41）。先に見たリスト 4.40 も併せ、undefined、null フリーな型を用意して処理の途中からそちらに乗り換えるのが王道でしょうか。

リスト 4.41: デフォルト値と付き合う

```
interface Config {
  filePath?: string | null;
  verbose?: boolean;
}

// Config の undefined と null 無し版
interface ConfigFixed {
  filePath: string;
  verbose: boolean;
}

let config: Config = {
  verbose: true,
};
// filled の型は {} & ConfigFixed & Config
// assign の定義が引数 4 つまでは intersection types で定義されているため
// assign<T, U, V>(target: T, source1: U, source2: V): T & U & V; が実際の定義
let defaultConfig: ConfigFixed = { filePath: "settings.json", verbose: false };
let filled = Object.assign({}, defaultConfig, config);

// Config と ConfigFixed には直接の互換性はない！
// error TS2322: Type 'Config' is not assignable to type 'ConfigFixed'.
```

```
//   Types of property 'filePath' are incompatible.
//     Type 'string | undefined' is not assignable to type 'string'.
//       Type 'undefined' is not assignable to type 'string'.
// let fixed: ConfigFixed = config;

// filled は filePath と verbose が存在することが確定しているので ConfigFixed と互換性がある！
let fixed: ConfigFixed = filled;
console.log(fixed);

export { ConfigFixed, fixed }
```

Control flow based type analysis が賢く処理してくれることに賭けるか、`Object.assign` などを使い、intersection types を上手く活用します。

他によい方法が思いついたら、ぜひ筆者にその方法を教えてください。筆者としてはもう少し Control flow based type analysis と構造的部分型の相性がよいと楽だなと考え、TypeScript リポジトリに Issue[*3]を立てています。もし興味があれば覗いてみて、何か意見を書いていってください。

[*3] https://github.com/Microsoft/TypeScript/issues/10065

第5章

オプションを知り己のコードを知れば百戦危うからず

本章では tsc のコマンドラインオプションについて解説していきます。すべてを網羅することはしませんが、いくつかの重要なオプションについては知ることができるでしょう。

本章記載のオプションは tsconfig.json の compilerOptions に記載可能なプロパティ名と同一です。tsconfig.json では短縮形（`-d` や`-p`）は利用できないことに注意してください。

ここに記載されていないオプションで知りたいものがあれば本書の Issue[*1] にお寄せください。

5.1 --init

`--init` オプションについて解説します。このオプションを使うと、TypeScript でコードを始める時に必要な tsconfig.json の雛形を生成します。生成されたファイルは後述の `--project` オプションと組み合わせて使います。TypeScript ではプロジェクトのビルドに必要なコンパイルオプションや、コンパイル対象の指定などを tsconfig.json ファイルにまとめていきます。このファイルはすべてのツールや IDE・エディタ間で共通に利用できる設定ファイルになるため、大変役立ちます。

まずは `tsc --init` コマンドで生成される tsconfig.json を見てみます（リスト 5.1）。

リスト 5.1: 生成された tsconfig.json

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
```

[*1] https://github.com/typescript-ninja/typescript-in-definitelyland/issues

```
        "sourceMap": false
    },
    "exclude": [
        "node_modules"
    ]
}
```

tsconfig.json に記述可能なプロパティは概ね次の 4 つです。

- compilerOptions
- files
- include
- exclude

compilerOptions には、コンパイル時に利用するオプションを指定します。コンパイルオプションの名前と compilerOptions に記載可能なプロパティ名は一致しています。たとえばリスト 5.1 は `tsc --module commonjs --target es5` という意味になります。`--noImplicitAny` と`--sourceMap` の値は false なのでオプションとして指定していない状態を指します。

tsconfig.json で利用可能な compilerOptions については、本章を読むか公式ハンドブックの解説[*2]や JSON Schema の定義[*3]を参照してください。

残る 3 つはコンパイル対象にするファイルを指定するためのプロパティです。3 つすべてに共通の挙動として、コンパイル対象に明示的に含めない場合でも TypeScript コンパイラが自動的に依存関係を解決し必要なファイルを対象に含めてくれる場合があります。この機能は歓迎すべき機能で、余計な設定の手間を減らしてくれます。

files には、コンパイル対象にするファイルを 1 つ 1 つ列挙します。あまりにも面倒くさいため、tsconfig-cli[*4]などのツールを利用する必要がありました。これの反省を踏まえ、次に説明する include、exclude が導入されました。

include、exclude はコンパイル対象とするファイルやフォルダを大まかに指定します。include と exclude を全く指定しない場合、TypeScript コンパイラは処理中のディレクトリ

[*2] http://www.typescriptlang.org/docs/handbook/tsconfig-json.html
[*3] http://json.schemastore.org/tsconfig
[*4] https://www.npmjs.com/package/tsconfig-cli

やサブディレクトリ配下を調べ、すべての.ts ファイルや.tsx ファイルをコンパイルしようとします。

そこで include で調べるディレクトリやファイルを、exclude で除外するディレクトリやファイルを指定し処理対象を限定します。include、exclude の 2 つは組み合わせて使うのが一般的です。ディレクトリを指定すると、そこに含まれるすべての.ts ファイルと.tsx ファイルが対象になります。簡単なワイルドカードも利用できます。例を見てみましょう（リスト 5.2)。

リスト 5.2: 使えるワイルドカードの例

```
{
    "compilerOptions": {
        "listFiles": true, // コンパイルの処理対象を表示する
        "noEmit": true     // コンパイル結果の.js ファイルなどを出力しない
    },
    "include": [
        /// ディレクトリのワイルドカード
        // /**/ で全てのサブフォルダ
        "libA/**/*",
        // /*/ で直下のサブフォルダ
        "libB/*/*",

        /// 文字のワイルドカード
        // * で 0 文字以上にマッチする
        "libC/*.ts",
        // ? で 1 文字にマッチする
        "libD/?b.ts"
    ],
    "exclude": [
        "node_modules",
        // 除外でも同じようにワイルドカードが使える
        "libD/b*.ts"
    ]
}
```

例で示した tsconfig.json を利用してみます。リスト 5.3 はプロジェクト内部に存在する ts 関連ファイルと、ファイルが処理されるかされないかを突き合わせたものです。

リスト 5.3: ファイル一覧とマッチの結果

```
libA/a/index.ts    # 対象になる
libA/a/b/index.ts  # 対象になる
libB/index.ts      # 対象にならない
libB/a/index.ts    # 対象になる
libC/index.ts      # 対象になる
libC/index.tsx     # 対象にならない
libD/ab.ts         # 対象になる
libD/ac.ts         # 対象にならない
libD/bb.ts         # 対象にならない（exclude）
```

なかなか素直な結果です。単にワイルドカードであって、正規表現で記述できるわけではない点に注意しましょう。

5.2 --project

`--project` オプションについて解説します。短縮記法の`-p` も利用できます。

このオプションでプロジェクトのコンパイルを行います。オプションの値として tsconfig.json があるディレクトリか、tsconfig.json のパスを指定します。具体的には `tsc -p ./`または `tsc -p ./tsconfig.json` とします。

tsconfig.json ではない名前のファイルを使って、プロジェクト内に複数のビルド構成を作ることもできます。しかし、その場合 IDE・エディタ側が設定をうまくハンドリングしてくれない場合が多いため、基本的には努力して 1 プロジェクトにつき 1 つの tsconfig.json となるようにしましょう。

gulp や grunt などのタスクランナーを使う場合でも tsconfig.json を用意し`--project` オプションのみでコンパイルを通せる環境を維持するのがよいでしょう。

5.3 --noImplicitAny

TypeScript コンパイラの最重要オプション、`--noImplicitAny` について解説します。このオプションは"暗黙的な any を禁ずる"の名が表すとおり、型推論の結果、暗黙的に変数の型が any になった場合、エラーとしてくれます。

リスト 5.4 のようなメソッドの返り値の型を書き忘れた！ という脇の甘いコードをコンパイルしてみます。

リスト 5.4: メソッドの返り値を書き忘れた！

```
declare class Sample {
  // 返り値の型を指定し忘れている！
  // error TS7010: 'method', which lacks return-type annotation,
  //               implicitly has an 'any' return type.
  method();
}

// 仮引数 word に型注釈がない
// error TS7006: Parameter 'word' implicitly has an 'any' type.
function hi(word) {
  word = word || "TypeScript";
  console.log(`Hello, ${word}`);
}

export { }
```

```
$ tsc --noImplicitAny definition.d.ts
definition.d.ts(3,5): error TS7010: 'method', which lacks return-type
    annotation, implicitly has an 'any' return type.
```

返り値の型を書いていなかったり、関数の仮引数の型が指定されていなかったりしたため暗黙的に any になってしまいました。このようなときに、それはダメだ！ とコンパイラが教えてくれます。any が紛れ込んで型チェックが意味を成さなくなると TypeScript の意義が薄れてしまいます。型定義ファイルを書くときも、通常の開発時も、常に--noImplicitAny を使うようにしましょう。

5.4 --strictNullChecks

--strictNullChecks オプションについて解説します。このオプションは null や undefined の扱いについてより厳格にし、変数の中身についての曖昧さを積極的に排除するよう振る舞います。null や undefined を許容したい場合、union types や省略可能引数を使って明示的に null や undefined である可能性を示さなければなりません。

本書は--strictNullChecks オプションを常に有効にしている前提で書いています。有効

にしている時の挙動は本書のサンプルすべてが該当しますので、この節ではこのオプションを使わないときの挙動について確認します。

まずはオプションありの例です（リスト 5.5）。

リスト 5.5: 危険なコードがいち早く発見される

```
// --strictNullChecks 無しとの比較
// 無しの場合に等しい表現は…
let objA: Date;
// コレです
let objB: Date | null | undefined;

objA = new Date();
// Date 単体の型注釈の場合、エラーとなる
// error TS2322: Type 'null' is not assignable to type 'Date'.
// objA = null;

// …しかし、一回 any を経由すればごまかせてしまう 他のサンプルコードでもたまにやってます
objA = null as any;

// objB は null も undefined も許容するため、ゆるゆる
objB = new Date();
objB = null;
objB = void 0; // undefined

// 処理フロー的に undefined が確定しているのでエラーとなる
// error TS2532: Object is possibly 'undefined'.
// error TS2339: Property 'getTime' does not exist on type 'never'.
// objB.getTime();

// 非 null 指定演算子（!）で無理やりコンパイルを通すこともできる
objB!.getTime();

export { }
```

null や undefined に対するアクセスが多くの場合未然に防がれ、"コンパイルが通ればもう安全"であるコードが書きやすいことがわかります。

非 null 指定演算子（!）については第 4 章「アドバンスド型戦略」の「4.11 非 null 指定演算子（Non-null Assertion Operator）」で触れました。

さて次はオプションなしの例です（リスト 5.6）。

リスト 5.6: 実行時にエラーになるかも

```
// --strictNullChecks 無しだと大変ゆるい
let obj: Date;
// 全部 OK !
obj = new Date();
obj = null;
obj = void 0; // undefined

// 処理フロー的には undefined だけど怒られない
obj.getTime();

export { }
```

ゆるゆるですね。変数の中身を容易に null や undefined にできてしまいます。きっちりコードを書けば、オプション無しでも堅牢なアプリケーションを構築することは不可能ではありません。しかし、それはプログラマの不断の努力の上にしか成り立ちません。そんな苦労をするよりは、コンパイラにしっかりチェックしてもらえたほうがコードの堅牢さが確かなものになりますね。

5.5 --noUnusedLocals

--noUnusedLocals オプションについて解説します。その名のとおり、使っていないローカル変数があったらエラーにしてくれます。本書のサンプルコードでも有効になっているため、エラー消しのために無意味に export している箇所がありました。

例を見てみます（リスト 5.7)。

リスト 5.7: 未使用変数はちゃんと消そう

```
// import した後、一回も使わないのはエラー
// error TS6133: 'readFile' is declared but never used.
import { readFile } from "fs";

// 1 回も参照されていないとエラーになる
// error TS6133: 'objA' is declared but never used.
let objA = {};

// どこかで参照されていれば OK
let objB = {};
```

```
export { objB }

// export していればどこかで使われるかもしれないから OK
export let objC = {};
```

未使用の変数があるとエラーになります。まるで Go 言語のようですね。エラーを削っていくと、import 文自体を削減できるパターンもあるでしょう。コードをきれいに保とう！

5.6 --noUnusedParameters

--noUnusedParameters オプションについて解説します。関数やメソッドの引数に使っていないものがあるとエラーにしてくれます。エラーにせず残しておきたい場合、変数名の頭に _（アンダースコア）をつけることでエラーを抑制できます。

例を見てみます（リスト 5.8)。

リスト 5.8: 使っていない仮引数はできれば削除したい

```
// 仮引数 b は利用されていないのでエラー _c はプリフィクス_なのでエラーにならない
// error TS6133: 'b' is declared but never used.
export function foo(a: string, b: number, _c: boolean) {
  console.log(a);
}

export class Sample {
  // 仮引数 a は利用されていないのでエラー
  // error TS6133: 'a' is declared but never used.
  method(a: string) {
  }
}
```

未使用の仮引数があるとエラーになります。関数の引数の数や型を後から変更するのはめんどくさいので、なるべく早めに検出し修正してしまいたいものです。

5.7 --noImplicitReturns

--noImplicitReturns オプションについて解説します。関数やメソッドの返り値について、return で値を返す場合と return しない場合、エラーになります。

例を見てみます（リスト 5.9）。

リスト 5.9: 暗黙の return を禁じる

```
// return がない（暗黙的に undefined が返る）パターンを検出してくれる
// error TS7030: Not all code paths return a value.
function a(v: number) {
  if (v < 0) {
    return "negative";
  } else if (0 < v) {
    return "positive";
  }

  // return がない！
}

function b() {
  // そもそも常に return がないなら OK
}

export { }
```

プログラミングのスタイルとして、else の漏れや値の返し忘れがあるコードはミスである可能性が高いです。そういったコードを書くとエラーになるのは便利ですね。

5.8 --noImplicitThis

--noImplicitThis オプションについておさらいします。第 4 章「アドバンスド型戦略」の「関数の this の型の指定」で述べたとおり、このオプションを利用すると、this の型指定がない関数内で this へアクセスするとエラーになります。

例を見てみます（リスト 5.10）。

リスト 5.10: 型指定無しの this の利用を禁じる

```
// 関数内部での this の型を偽の第一引数で指定
function testA(this: string) {
  console.log(this.toUpperCase());
}
testA.bind("TypeScript")();

function testB() {
  // --noImplicitThis オプション利用時、関数内で this にアクセスすると怒られる
  // error TS2683: 'this' implicitly has type 'any'
  //   because it does not have a type annotation.
  console.log(this.toUpperCase());
}

export { testB }
```

5.9 --target

`--target` オプションについて解説します。短縮記法で`-t` も利用できます。TypeScript のコンパイルを行う際、ECMAScript 3（超古い！）、ECMAScript 5（古い！）、ECMAScript 2015（最近）のどのバージョンをターゲットとするかを指定します。これは、"TypeScript コードをどのバージョンで書くか"ではなく、"書いた TypeScript をどのバージョンに変換するか"の指定です。TypeScript では基本的に最新の記法で書き、ダウンパイル（古い書き方へ変換）します。

利用可能なオプションの値は次のとおりです。

- es3
- es5
- es6 / es2015

基本的に、IE11 などの少し古いブラウザのサポートを切らないのであれば es5 を選択すればよいでしょう。es3 の利用はもはやお勧めしません。

一部、Generator や async/await などの記法はダウンパイルできません。これらは 2.1.0 でサポートされる予定なので、延期されないようにみんなで祈りましょう。

5.10 --module、--moduleResolution

`--module` オプションについて解説します。短縮記法で`-m` も利用できます。TypeScript はモジュールをコンパイルする際に、どの形式に変換するかを選ぶことができます。

利用可能なオプションの値は次のとおりです。

- none
- commonjs
- system（SystemJS）
- umd
- es6 / es2015

これも明確な事情がない限り、今のところは commonjs でよいでしょう。

`--moduleResolution` オプションについて少し触れておきます。モジュールの名前解決の方法について指定できます。

利用可能なオプションの値は次のとおりです。

- node
- classic（TypeScript 1.6 以前の形式）

基本として node 一択でよいでしょう。

前述の`--target` と自由に組み合わせることができるため、`--target es5` としつつ `--module es6` とすることもできます。この組み合わせが可能になったのは TypeScript 2.0.0 からなので、Rollup.js[*5]との組み合わせての運用はまだ未知数です。TypeScript + Rollup.js をプロジェクトに導入してみてブログ記事などにまとめてみると話題になるかもしれません。お待ちしています！

5.11 --lib

`--lib` オプションについて解説します。TypeScript のコンパイルを行う際、標準の型定義として何を使うかを個別に指定できます。たとえ、`--target es5` としてダウンパイルする

[*5] http://rollupjs.org/

場合でも、利用する型定義は es2015 にできるのです。最近は Promise を使った API は珍しくないですし、かつ IE11 でも動かしたい場合というのはザラにあります。

利用可能なオプションの値は次のとおりです。複数指定したい場合、コマンドラインオプションの場合は , で区切ります。tsconfig.json の場合は素直に配列にしましょう。

- dom
- webworker
- es5
- es6 / es2015
- es2015.core
- es2015.collection
- es2015.iterable
- es2015.promise
- es2015.proxy
- es2015.reflect
- es2015.generator
- es2015.symbol
- es2015.symbol.wellknown
- es2016
- es2016.array.include
- es2017
- es2017.object
- es2017.sharedmemory
- scripthost

自分のプロジェクトの用途を考え、適切なものを選びましょう。たとえば Node.js なプロジェクトであれば HTMLElement などは不要でしょうから dom はいらないです。多くのプロジェクトでは es2017 か、+ dom の指定があれば十分でしょう。

es2017 を利用する場合は es2017 の型定義に es2016 の参照が含まれます。どの標準型定義ファイルが何を参照しているかが気になる場合は直接型定義ファイルを見るか、`--listFiles` オプションをつけてコンパイルしてみたりするとよいでしょう。

5.12 --forceConsistentCasingInFileNames

`--forceConsistentCasingInFileNames` オプションについて解説します。このオプションを有効にすると、ファイル名の参照について大文字小文字の食い違いがあるとエラーにします。macOS のような非ケースセンシティブな環境と、Linux のようなケースセンシティブな環境が混在しているとき、macOS ではエラーにならないけど Linux ではエラーになる…のようなシチュエーションを防止してくれます。チーム内で macOS に統一されていても、外部の人や CI サーバなどは Linux を使っている場合などはかなり多いため、とりあえず有効にしてしまってよいでしょう。

5.13 --noEmitOnError、--noEmit

`--noEmitOnError` オプションと`--noEmit` オプションについて解説します。このオプションは成果物である.js ファイル、.js.map ファイル、.d.ts ファイルを生成するか否かを制御します。

`--noEmitOnError` はコンパイルが成功した時のみファイルを生成します。これは grunt や gulp などのタスクランナーを利用する際の「コンパイル成功したつもりだったけど失敗してた。後続のタスクが続いてしまい失敗を見逃した」というパターンに有効です。前回の生成物を削除してからコンパイルするようにすることで.js ファイルが必要なステップで処理全体が確実に落ちるようにできます。「そんなクソみたいなタスク作らないよ！」と思うかもしれないですが、これが案外やりがちなのです。プロジェクトの健康を保つためにも、`--noEmitOnError` オプションは常に有効でよいでしょう。

`--noEmit` オプションはコンパイルが成功しようが失敗しようが、常に成果物を何も生成しません。`tsc -p ./`でファイルを生成するのとは違う手順でのみビルドを行う場合、例えば webpack で ts-loader を使っているプロジェクトなどで有効です。`tsc -p ./ --noEmit` とすることで TypeScript のコンパイルエラーのみをチェックできます。これはビルドタスク全体を走らせるよりも手短で、作業ディレクトリに不要なファイルを撒き散らすこともありません。

第6章

JS 資産と型定義ファイル

6.1 JavaScript の資産が使いたい

TypeScript は JavaScript の上位互換であり、JavaScript を置き換えるものです。とはいえ、現時点では Web アプリの世界は JavaScript で成り立っていますし、すでに莫大な資産があります。それらを放り出してしまうのはあまりにもったいないので、TypeScript でも活用したいものです。そのために TypeScript は既存の JavaScript 用資産を活用するための仕組みを持っています。それが、型定義ファイルです。

通常の TypeScript コードは拡張子が.ts なのに対して、型定義ファイルは拡張子を.d.ts とします。拡張子を.d.ts としたファイルに実装を含むようなコードを書くと tsc がエラーにするので、ケアレスミス予防のためにも型定義ファイルの拡張子は必ず.d.ts にします。

TypeScript では、JavaScript の自由奔放（かつ、危険がてんこ盛り）の世界に後付で型を与えます。もとから TypeScript で書かれている場合、実装と型定義を同時に書いているためこのふたつがズレて（つまりバグって）しまうことはありません。一方、型定義ファイルはすでに実装がある JavaScript に後付かつ手書きで型をつけていくため、ズレる（バグる）可能性が大いに有ります。そこのところを十分に気をつけないといけません。

6.2 @types を使う

さて、まずは自分で型定義ファイルを作るよりも、既存のものを使ってみましょう。jQuery や lodash などの有名どころはひととおり揃っています。

ライブラリの作者が TypeScript ユーザで、npm package 自体に型定義ファイルがバンドルされていて何も考えずに TypeScript から使える場合もありますが、今のところまだ稀です。基本的には DefinitelyTyped[*1] というコミュニティベースの型定義ファイル集積リポジトリを利用することになるでしょう。

[*1] https://github.com/DefinitelyTyped/DefinitelyTyped

DefinitelyTyped から型定義ファイルをダウンロードしてくるための方法は複数用意されています。TypeScript 2.0.0 からは@types という npm の scoped package[*2]を使って型定義ファイルを利用します。2.0.0 以前では tsd[*3]や dtsm[*4]や typings[*5]というツールを使っていましたが、これらが不要になります。

しばらくは過渡期になるため、混乱があったり利用の仕方がわかりにくかったり型定義ファイルが壊れていたりする場合があるかもしれません。コミュニティの力によって徐々に前進し、やがてはみんなが@types を使うようになるでしょう。もし、ここで紹介する方法でうまくいかない場合、利用事例やブログ記事などが出回っている旧ツール群のいずれかを使ってみるとよいでしょう。

さて、前置きが長くなりましたが実際に型定義ファイルをダウンロードしてきて使ってみましょう。ここではテストで使う便利ライブラリ、power-assert を題材にして型定義ファイルをダウンロードしてみます。

```
# 型定義ファイルを install
$ npm install --save-dev @types/power-assert
└─┬ @types/power-assert@0.0.27
  ├── @types/empower@0.0.28
  └── @types/power-assert-formatter@0.0.26
```

power-assert の型定義ファイルが依存しているモジュールの型定義も芋づる式に取得できています。便利ですね。型定義ファイルのパッケージには残念ながらライブラリの実体は含まれていないため `npm install power-assert` で別途インストールする必要があります。

既存ライブラリに対する型定義ファイルは@types/の下に元ライブラリのパッケージ名と同じ名前で公開される運用です。パッケージの検索は TypeSearch[*6]で行うか、npm search を使うとよいでしょう。

また、@types で導入した型定義ファイルの検索は、モジュールの解決方法（`--moduleResolution`）が node のときのみ行われます[*7]。AMD などを利用したい場合、現

[*2] @xxx/ から始まる名前空間が区切られた npm package のこと https://docs.npmjs.com/misc/scope
[*3] https://www.npmjs.com/package/tsd
[*4] https://www.npmjs.com/package/dtsm
[*5] https://www.npmjs.com/package/typings
[*6] https://microsoft.github.io/TypeSearch/
[*7] https://github.com/Microsoft/TypeScript/issues/9831

時点では`--moduleResolution node`を指定するようにしましょう。

■コラム: @types と DefinitelyTyped の今

@types の対応は Microsoft の TypeScript チームが主体となって始めました。DefinitelyTyped は規模は大きくなっていくもののアクティブにメンテを続けるメンバーが少なく、運用上徐々に無理が生じてきていたと思います。現在、TypeScript 2.0 に向けて Microsoft の TypeScript チームがどんどん参加してきてくれています。彼らは給料を貰い、仕事の時間内にコミュニティを回すための時間を割いてくれているため、今後は今までよりも回転が早くなるでしょう。

執筆時点（2016 年 08 月 01 日）では、DefinitelyTyped リポジトリの types-2.0 ブランチで TypeScript 2.0 対応が行われています。もし、@types へ変更を反映してほしい人がいる場合、現時点では types-2.0 ブランチに pull request を送ってください。また、TypeScript 2.0 リリース付近で、この辺りの運用について TypeScript チームから正式な発表があるでしょう。

参考になる URL を示しておきます。

- types-publisher https://github.com/Microsoft/types-publisher
- TypeSearch https://microsoft.github.io/TypeSearch/
- 上記サイトのリポジトリ https://github.com/Microsoft/TypeSearch

6.3 型定義ファイルを参照してみよう

型定義ファイルを参照するには、tsc コマンドでコンパイルするときにコンパイル対象に含める必要があります。node_modules/@types にある型定義ファイルは特別扱いされ、モジュールを import した時や、tsconfig.json の types に記述したモジュールの解決時に自動的に走査されます。要するに npm install したら、後は何も気にしなくても TypeScript コンパイラが型定義ファイルを探しだしてきてくれるのです。

古くはリファレンスコメントとして、ソースコードの先頭に`/// <reference path="`相対パス or 絶対パス`" />`の形式で書く方法もありましたが tsconfig.json の登場により廃れまし

た。基本として依存性の解決などは tsconfig.json で行うようにします。

mocha + power-assert でテストを書く場合を例に、使い方を解説していきます。

テスト対象のコードは./lib/index.ts です（リスト 6.1）。

リスト 6.1: 至って普通のモジュール

```
export function hello(word = "TypeScript") {
  return `Hello, ${word}`;
}
```

これに対してテストコードとして./test/indexSpec.ts を書いてみましょう（リスト 6.2）。

リスト 6.2: mocha+power-assert でテストを書く

```
import * as assert from "power-assert";

import { hello } from "../lib/";

describe("lib", () => {
  describe("hello function", () => {
    it("generate string with default value", () => {
      let str = hello();
      assert(str === "Hello, TypeScript");
    });
    it("generate string with parameter", () => {
      let str = hello("JavaScript");
      assert(str === "Hello, JavaScript");
    });
  });
});
```

普通ですね。「特定の input を与えると output が得られる」ことを検証するコードです。

ここで問題なのは、TypeScript コンパイラが安全にコードを処理するためには、mocha と power-assert についての情報が必要であることです。たとえば、assert 関数は power-assert が提供するものですし、describe と it は mocha が提供しています。JavaScript の世界では静的な型検査などありませんので問題ありませんが、TypeScript ではそうはいかないため外部ライブラリの型情報をどうにかしてコンパイラに教えてあげる必要があります。そこで使

われるのが型定義ファイルです。

mocha（リスト 6.3）と power-assert（リスト 6.4）の型定義ファイルを抜粋・簡略化したものを見てみましょう。

リスト 6.3: mocha.d.ts 抜粋

```
interface MochaDone {
  (error?: Error): void;
}
declare let describe: {
  (description: string, spec: () => void): any;
};
declare let it: {
  (expectation: string, assertion?: () => void): any;
  (expectation: string, assertion?: (done: MochaDone) => void): any;
};
```

リスト 6.4: power-assert.d.ts 抜粋

```
export = assert;
export as namespace assert;

declare function assert(value: any, message?: string): void;
```

型定義ファイルを見ると mocha と power-assert それぞれの API が表現されています。TypeScript コンパイラがこれらの型定義ファイルを認識できれば、矛盾なくコンパイルを通すことができそうです。そのための package.json（リスト 6.5）と tsconfig.json（リスト 6.6）を確認します。

リスト 6.5: package.json

```
{
  "name": "typescript-in-definitelyland-sample",
  "private": true,
  "version": "1.0.0",
  "main": "lib/index.js",
  "scripts": {
    "build": "tsc -p ./",
    "pretest": "npm run build",
```

```
    "test": "mocha"
  },
  "author": "vvakame",
  "license": "MIT",
  "devDependencies": {
    "@types/mocha": "^2.2.28",
    "@types/power-assert": "0.0.27",
    "mocha": "^2.5.3",
    "power-assert": "^1.4.1"
  }
}
```

リスト 6.6: tsconfig.json

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": true,
        "strictNullChecks": true,
        "types": [
            "mocha"
        ]
    },
    "exclude": [
        "node_modules"
    ]
}
```

power-assert はテストコード中で import しますが、テストランナーである mocha の定義はソースコード中からの参照がありません。そのため、power-assert については TypeScript コンパイラが必要であることを判別し、型定義ファイルを探しにいってくれます。

しかし mocha はそのような機会がないため、TypeScript コンパイラは型定義を探しにいってくれません。このままコンパイルすると mocha がグローバルに値を展開している describe や it などが見つからないと言われてしまいます。これを解決するために tsconfig.json の types プロパティ中で mocha を参照するよう指定します。

あわせて型定義ファイルへの参照が意図どおり処理されずに困った場合のデバッグ方法を紹介しておきます。コンパイルに利用したファイルをリスト表示する`--listFiles` オプショ

ンと、型定義ファイルを見つけるためにコンパイラがどういう探索を行ったかを表示する `--traceResolution` オプションを試してみてください。

6.4　型定義ファイルを書こう

さて型定義ファイルの取得方法、使い方はわかりました。しかし、世の中にある JavaScript ライブラリのうち、型定義ファイルが書かれていないものはまだまだ数多くあります。特に、門外不出の社内ライブラリなどは誰も手をつけていない前人未到の地です。

しからば！ 自分で書くしかあるまいよ！ ぶっちゃけた話、めんどくさいのですが、後々の安心・安全を得るための投資として割りきりましょう。

なお、自分で型定義ファイルを書く覚悟無しに TypeScript をやるのは茨の道だと思いますので頑張ってください。「誰かがやってくれないと自分ではできません」なんて甘えた根性では型サバンナでは到底生きていけないのです[*8]。

型、実体、そして 42。

TypeScript は JavaScript に対して後付で型による制約を付け足した言語です。そのため、Java や C#のような最初から型ありきの言語より少し考え方が複雑です。具体的にいえば型と実体（値）というものが分かれています。

すべてが TypeScript で書かれたプログラムであれば、型と実体は基本的には一致しています。クラスの定義を書いたとき、JavaScript プログラムとしてのクラスと、TypeScript で使う型としてのクラスが一度に誕生します。これは大変素直かつ簡単な動作で、ひとつの記述から型と実体を作成しているためこの 2 つが乖離してしまうことはありません。

一方、JavaScript でコードを書いて TypeScript で型定義ファイルを作成して使う場合、実装と型が個別に定義されることになります。そのため、型と実体が分離してしまい、この 2 つの間に乖離が生じると（つまりバグると）コンパイルが通るのに実行時エラーが多発する、というありさまになるわけです。型定義ファイルを書いて"この変数は、あります！ "と宣言したけれど、実際には存在せず実行時エラーになるというのは広く使われている型定義ファイルですらままある話です。

[*8] DefinitelyTyped メンテナ（＝筆者）の意見です

良い型定義ファイル、悪い型定義ファイル

型定義ファイルにも良し悪しがあります。その基準は至って簡単です。

1. 正しいライブラリの使い方を導くこと
2. 他のコードや型定義ファイルに意図せぬ干渉を引き起こさないこと
3. IDE 上で使いやすいこと

正しいライブラリの使い方を導く、というのは裏を返せば間違った使い方ができないようにする、ということです。これには型と実体の定義に乖離が存在せず、コンパイルが通ったら実行時エラーが簡単には起こらないことも含まれます。

他のコードや型定義ファイルに意図せぬ干渉を引き起こさないこと、というのは意図せぬインタフェースの統合などが起こらないことを指します。このためには汎用的な名前を使うのを避け、可読性が高く理解しやすい型定義を作り、干渉した場合に容易に判断できるようにすることも含まれます。

IDE 上で使いやすいことというのは、Visual Studio Code などでコードを書く上で入力補完の候補が不用意に出過ぎないようにして見通しのよい開発を助けることなどが含まれます。

これら 3 つを守ることが"良い品質であること"に繋がるというのは、TypeScript 自体が型指定を行うことで間違ったコードを書きにくいようにするツールであると考えると納得がいくでしょう。

慣れないうちはどうしても"うまく書けないので仕方なく"悪い型定義を書いてしまうことがあります。DefinitelyTyped に pull request を送ってくれる人にもそういう人は多くいます。

これから説明するベストプラクティスを踏まえて、より良い型定義ファイルを作成できるように鍛錬していきましょう。

6.5　型定義ファイルを書くための心得

型定義ファイルを書く上でのベストプラクティスを解説していきます。基本的には公式 Handbook の Writing Declaration Files[*9] と DefinitelyTyped の best practices[*10] にしたがっておけばよいです。本章では、そこに書かれていることや筆者の経験則などを説明していき

[*9] http://www.typescriptlang.org/docs/handbook/writing-declaration-files.html
[*10] http://definitelytyped.org/guides/best-practices.html

ます。

テキトーに、やろー！

一番最初にコレを書くのもどうかと思うのですが、まずは"使える"ようにするのが一番大切です。

型定義ファイルの品質の良さにこだわるあまり、完成しない、使いたいライブラリが使えない、というのがもっともよくない状態です。型定義ファイルの良し悪しを判断する力は、TypeScript 自体への理解度に大きく依存します。TypeScript を書き始めの頃は、品質を気にした所で後々粗が見えてくるのは避けられません。まずは"使える"状態にすることを目指しましょう。

品質や"ライブラリ全体をカバーしている"かは気になるところではあります。しかし、まずは使いたいところが使えればいいのです。スゴいものになると、1 万行を超える型定義ファイルがあります。また 3000 行程度のものはわりとごろごろしています…。しかし、そんなにも頑張って書いてると余裕で日が暮れてしまいます[*11]。

まずは、使いたいところが、使える！ それでよいのです。ドラゴン・ゲンドーソー＝先生のインストラクション・ワンを思い出しましょう。

> 百発のスリケンで倒せぬ相手だからといって、一発の力に頼ってはならぬ。一千発のスリケンを投げるのだ！

最初はうまくできなくても数をこなし学習を重ねれば、そのうち立派な型定義ファイルを書けるようになるでしょう。

最高に雑な型定義ファイルを作る

テキトーにやるためにまずは最高に雑な、とりあえず動く型定義ファイルを作ってみます(リスト 6.7)。モジュール名しか指定しなかったり、any な変数を用意したりしてコンパイルエラーを回避します。

リスト 6.7: 雑な型定義ファイルの例

[*11] なお筆者は GitHub の作っているエディタ、Atom の型定義ファイルを 3 日かけて書いたことがあります。アレがジゴクだ

```
// 名前だけ定義すると全て any でとりあえず使える
declare module "lodash";

// 必要な変数をとりあえず any で生やす
declare let $: any;

// 特定のパッケージ配下をとりあえず全部 any で
declare module "sample/*";


// webpack など特殊なローダー用
declare module "json!*";

// 同上
// モジュール読んだらモジュールは文字列
declare module "*!text" {
  const _: string;
  export = _;
}
```

この例だと、--noImplicitAny オプションを有効にするとエラーになってしまいます。そのため、コンパイルエラーを無くしたらなるべく早く--noImplicitAny を有効にできるように頑張りたいところです。

この型定義ファイルの利用例を見てみます（wildcard/basicUsage-ignore）。

リスト 6.8: 型定義ファイルの利用例

```
import * as _ from "lodash";
import * as sample from "sample/foobar";
import * as data from "json!./bar.json";
import * as text from "./foo.txt!text";

// _は any
_.map([1, 2, 3], n => n * 3);
// $は any
$("#id");
// sample は any
sample;
// data も any
data;
// text は string
```

```
text.toUpperCase();
```

any ばっかりですね。しかし、コンパイルはとおります。

インタフェースを活用する

インタフェースは大変使いやすいパーツです。というのも、インタフェースには **後から定義を拡張できる**という特性があるからです（リスト 6.9、リスト 6.10）[*12]。

リスト 6.9: 定義を分割して書く

```
interface Foo {
  hello(): string;
}

// 同名のインタフェースを定義すると、合成される！
interface Foo {
  bye(): string;
}
```

リスト 6.10: 定義が統合される！

```
/// <reference path="./declarationMerging.d.ts" />
// ↑ 昔はこのように reference comment を使ってファイル間の依存関係を明示していましたが、
//    最近は tsconfig.json に依存関係を書くため見かけることが大変少なくなりました

let foo: Foo = null as any;

foo.hello();
foo.bye();

export { }
```

このとおり別々に定義したインタフェースがひとつに統合されています。これを利用することで、既存の型であろうとも拡張が可能になるのです。

[*12] ちなみに、class の定義も後から拡張可能になりました https://github.com/Microsoft/TypeScript/issues/3332

例をひとつ見てみましょう。String#trimStart は、文字列の先頭にある空白文字を取り除く機能です。本章執筆時点（2016 年 08 月 01 日）では、この提案[*13]は TC39 のプロポーザルで stage 2[*14]で、TypeScript にはまだ入ってきていません。そのため String インタフェースを拡張する形でコンパイルを通せるようにしてみましょう（リスト 6.11）

リスト 6.11: String#trimStart を生やす

```
interface String {
  trimStart(): string;
}

let str = "  TypeScript  ";

// 文字列先頭の空白文字を削る
console.log(str.trimStart());
```

あとは、実行時に String.prototype.trimStart を適当な実装で補ってやれば未サポートのブラウザでも利用可能になるでしょう。

この手法は、他人が作った型定義ファイルを拡張する場合にも活用できます。相乗りできるのであれば遠慮なく乗っかっていってしまいましょう。

幽霊 namespace

幽霊 namespace[*15]という考え方があります。

namespace を作ったとしても、即座に実体が生成されるとは限りません。namespace が抱えるのがインタフェースのみである場合、実体がある扱いにはならないのです（リスト 6.12)。

リスト 6.12: 幽霊 namespace

```
declare namespace ghost {
  interface Test {
```

*13 https://github.com/sebmarkbage/ecmascript-string-left-right-trim

*14 https://tc39.github.io/process-document/

*15 TypeScript リファレンスでは非インスタンス化モジュールという名前で紹介しました。その後、Definitely-Typed の best practices で ghost module と表記された

```
    str: string;
  }
}

// 型としては普通にアクセスできる
let test: ghost.Test;
test.str;

// 実体としては存在していない！
// invalid.ts(13,17): error TS2304: Cannot find name 'ghost'.
let notExists = ghost;

export { }
```

これを活用して大量のインタフェースをもつようなライブラリの定義をひとまとまりにできます。

実際の例を見てみましょう。リスト 6.13 は jQuery の型定義ファイルからの抜粋（&一部改変）です。

リスト 6.13: 実際の jQuery の型定義の例

```
interface JQuery {
  addClass(className: string): JQuery;
  html(htmlString: string): JQuery;
  val(): any;
  empty(): JQuery;
  append(content1: JQuery, ...content2: any[]): JQuery;
  appendTo(target: JQuery): JQuery;
}

interface JQueryStatic {
  ajax(settings: JQueryAjaxSettings): any;
  (selector: string, context?: Element): JQuery;
  (element: Element): JQuery;
}

interface JQueryAjaxSettings {
  data?: any;
  type?: string;
  url?: string;
}
```

```
interface JQueryPromise<T> {
  state(): string;
  then<U>(
    fullfill: (value: T) => U,
    reject?: (...reasons: any[]) => U
  ): JQueryPromise<U>;
}

interface JQueryDeferred<T> extends JQueryPromise<T> {
  reject(...args: any[]): JQueryDeferred<T>;
  resolve(value?: T, ...args: any[]): JQueryDeferred<T>;
}

declare var $: JQueryStatic;
```

トップレベルに複数の型がいくつも散乱してしまうのがよくありません。それに JQuery という prefix が乱舞していて目を惑わせます。ライブラリ内部で API 同士が参照する場合でも引数や返り値にプリフィクスが必要なのはめんどうくさいです。IDE 上で型注釈を手書きするときも候補がたくさんサジェストされてしまうことでしょう。

これを幽霊 namespace を使って書きなおしてみます（リスト 6.14）。

リスト 6.14: 幽霊 namespace を使ってみた

```
declare namespace jquery {
  interface Element {
    addClass(className: string): Element;
    html(htmlString: string): Element;
    val(): any;
    empty(): Element;
    append(content1: Element, ...content2: any[]): Element;
    appendTo(target: Element): Element;
  }

  interface Static {
    ajax(settings: AjaxSettings): any;
    (selector: string, context?: Element): Element;
    (element: Element): Element;
  }

  interface AjaxSettings {
    data?: any;
```

```
    type?: string;
    url?: string;
  }

  interface Promise<T> {
    state(): string;
    then<U>(
      fullfill: (value: T) => U,
      reject?: (...reasons: any[]) => U
    ): Promise<U>;
  }

  interface Deferred<T> extends Promise<T> {
    reject(...args: any[]): Deferred<T>;
    resolve(value?: T, ...args: any[]): Deferred<T>;
  }
}

declare var $: jquery.Static;
```

インタフェース名が短く、かつわかりやすくなりました。やっぱり、こういうのがいいですね。

もちろん、無理に幽霊 namespace を使う必要はありません。クラスや変数や関数などを持ち、通常の実体をもつ namespace が存在している場合は、その namespace に相乗りしてしまったほうが楽でしょう。

…どうして DefinitelyTyped 上にある型定義ファイルでそうなってないものが多いのかって？ よい質問です。ひとつは幽霊 namespace の認知度が低いこと、もうひとつは型定義ファイルの大幅な書き換えは互換性の破壊を生み出すからです。先で説明しましたが、インタフェースは定義の統合ができます。この性質を利用して定義の拡張を行っているので、うかつに JQueryStatic から jquery.Static に型名を変更すると jQuery の型定義に依存しているさまざまなライブラリの色々なところが壊れてしまうのです。特に jQuery プラグインとかはインタフェースを拡張する形で型定義するのでその量たるや…。

ともあれ、過去の定義との互換性を壊すことに繋がるため、途中から幽霊 namespace に切り替えるのは難しい場合があります。可能であれば最初から幽霊 namespace を使うようにしましょう。将来的には、このパターンの検出は tslint などで機械的に行えるようにしたいところですね。

なんでもかんでもインタフェースにしてはならない

少し前の文章であんだけインタフェースを持ち上げといてこれかぁ！？ と思われたかもしれませんが、なんでもかんでも乱用すればいいってものではありません。

具体的には namespace 様の構造をインタフェースを使って作ってはいけません（リスト 6.15）。

リスト 6.15: インタフェースで namespace を表現してしまう。何故なのか…

```
interface Foo {
  bar: FooBar;
}

interface FooBar {
  buzz: FooBarBuzz;
}

interface FooBarBuzz {
  str: string;
}

declare var foo: Foo;

// foo.bar.buzz.str という使い方ができる。わかりにくくてユーザは死ぬ。
```

この型定義ファイルを読み解いて一瞬で使えるのは、元の JavaScript コードを熟知している人だけでしょう。少なくとも、この型定義ファイルをヒントに実際のコードを書くことには大いなる苦痛を伴います。筆者は絶対に使いません。絶対です。普通にリスト 6.16 のように書きましょう。

リスト 6.16: 素直にこうしよう

```
// 普通にコレでいいだろ！！
declare namespace foo.bar.buzz {
  let str: string;
}
```

さて次です。通常リスト 6.17 のような型定義ファイルを書こうとは思わないと思いますが、こういうコードが必要になる場合が稀にあります。関数としても呼べるし、namespace のようにも振る舞うオブジェクトの型定義を作成したいときです。

リスト 6.17: 関数・namespace どっちなの？

```
// assert は関数としても呼べるし namespace のようにも見える
assert(foo === "foo");
assert.ok(value);
```

呼び出し可能で、プロパティをもつ。この場合、すぐに考えつく型定義はリスト 6.18 か、リスト 6.19 でしょう。

リスト 6.18: こうしてしまいたい、気持ち

```
declare var assert: {
  (value: any): void;
  ok(value: any): void;
};
```

リスト 6.19: 匿名型注釈よりはマシ

```
declare var assert: Assert;

interface Assert {
  (value: any): void;
  ok(value: any): void;
}
```

たしかに、この定義でも動きます（正直、assert 関数だけの定義だとこのままでもいい気がしますが…）。

しかし、これには別のよいやり方があるのです（リスト 6.20）。

リスト 6.20: 関数と namespace 両方やらなきゃいけないのが辛いところだ

```
declare function assert(value: any): void;
declare namespace assert {
  function ok(value: any): void;
}
```

関数と namespace を同名で宣言できるのです。メリットは階層構造を素直に表現できることと、前項で説明した幽霊 namespace の書き方を併用できるところです。

この手法は、実際に power-assert の型定義ファイル[*16]でも利用されています。リスト 6.21 に抜粋＆改変したものを示します。

リスト 6.21: 関数 +namespace の実例

```
declare function assert(value: any, message?: string): void;
declare namespace assert {

  export function deepEqual(actual: any, expected: any): void;
  export function notDeepEqual(acutal: any, expected: any): void;

  export interface Options {
    assertion?: any;
    output?: any;
  }

  export function customize(options: Options): typeof assert;
}
```

外部に公開されている関数は `assert` のみで、そこに追加でプロパティが生えている形式です。namespace に Options インタフェースがうまく取り込まれています。余計な名前を階層の浅いところにバラ撒かず、厳密さも損なっていません。この書き方は、意外とよく登場するパターンなので覚えておきましょう。

実は、このやり方は型定義ファイルだけではなく通常の TypeScript コードでも使えます（リスト 6.22）。

リスト 6.22: 関数が先、namespace は後！ 絶対！

[*16] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/power-assert/

```
function test() {
  return "test!";
}
namespace test {
  export function func() {
    return "function!";
  }
}
```

コンパイル結果のリスト 6.23 を見ると、なぜ関数が先で namespace が後、という決まりになっているかがわかります。

リスト 6.23: JS として正しい構造だ

```
function test() {
    return "test!";
}
var test;
(function (test) {
    function func() {
        return "function!";
    }
    test.func = func;
})(test || (test = {}));
```

クラスはクラスとして定義する

クラスを型定義として起こす方法について解説します。歴史的経緯により、TypeScript ではクラスの型定義を行う時に 2 つの代表的なやり方が存在しています。まずはその 2 つのやり方を見てみましょう（リスト 6.24)。

リスト 6.24: 素直にクラス定義 vs インタフェース + 変数

```
// A. 普通にクラスを定義する
declare class TestA {
}
```

```
// B. クラスの分解定義 変数 + インタフェース 2 つ
declare let TestB: TestBConstructor;
interface TestBConstructor {
  new (): TestB;
}
interface TestB {
}
```

こんな感じです。普通にクラス定義をするほうが素直ですね。

過去にはこの 2 つのやり方にそれぞれメリット・デメリットがありました。しかし、現在の TypeScript では大幅に制限が緩和されたためメリット・デメリットの面で考える必要はなくなってきました（リスト 6.25)。よい時代になったものです。

リスト 6.25: 相互運用性がある！

```
// class は open-ended になったため同名の interface で拡張可能に
class Person {
  name: string;
}
interface Person {
  age: number;
}
let p: Person = new Person();
// 両方アクセス可能！
console.log(p.name, p.age);

// interface を使ったクラスの構成でも
interface AnimalConstructor {
  new (): Animal;
}
interface Animal {
  speak(): string;
}
/* tslint:disable:variable-name */
let Animal: AnimalConstructor = class {
  speak() {
    return "???";
  }
};
/* tslint:enable:variable-name */
// Animal はただの変数だが普通に継承できる！
class Cat extends Animal {
```

```
  speak() {
    return "meow";
  }
}
let cat: Cat = new Cat();
console.log(cat.speak());

// ECMAScript 2015 だと次のような文すら書けるのでまぁできて然るべきだった
let cat2: Cat = new class extends class {
  speak() {
    return "???";
  }
} {
  speak() {
    return "meow";
  }
}();
console.log(cat2.speak());

export { }
```

オーバーロードを上手く使おう！

正しいライブラリの使い方を導くこと。を心に秘めて、リスト 6.26 を見てください。

質問：どれが一番、元々の関数の仕様がわかりやすいですか？

リスト 6.26: 普通に使えます

```
// 同じ実装に対して、どの型定義が一番便利かな？
// 1 関数で get, set 両方の役目を果たす場合…

// get のとき set のとき 仕様が違うことがよく分かる
declare function valueA(value: any): void;
declare function valueA(): any;

// set のときも値が取れる気がする…？
declare function valueB(value?: any): any;

// 詳細が不明だ…！
declare let valueC: Function;
```

答え：一番最初のやつ。

JavaScript のライブラリは 1 つの関数にさまざまな使い方をさせようとする場合がままあります。つまり、1 つの関数が複数の顔をもつということです。その顔ひとつひとつに個別の型定義を割り振ってやるテクニックをオーバーロードと呼びます。

なお TypeScript コードを書くときは普通はオーバーロードをあまり使わないのがよいスタイルです。実装が煩雑になってしまいますからね。素直にメソッドを分けましょう。

union types を使うとリスト 6.27 のように書くこともできます。簡単な例だと union types のほうがよいと思いますが、見た目が煩雑になるケースではどっちがいいかは判断が分かれるところです。

リスト 6.27: うーん、どっちがいいかは難しい

```
// union types 未使用
declare function hello(word: string): string;
declare function hello(callback: () => string): string;

hello("TypeScript");
hello(() => "function");

// union types あり
declare function bye(word: string | { (): string; }): string;

bye("JavaScript");
bye(() => "function");
```

もう一例見てみます（リスト 6.28）。union types と overload の両方が選択肢に入る場合、現時点では union types を選んだほうがよい場合があります。

リスト 6.28: overload と union types は相性がよくない

```
declare function funcA(word: string): string;
declare function funcA(num: number): string;

let obj: string | number = null as any;

// string か number を渡さなければならない場合 string | number はコンパイルエラーになる
// 本来であれば、受け入れてほしいのだけど…
```

```
// error TS2345: Argument of type 'string | number'
//   is not assignable to parameter of type 'number'.
//  Type 'string' is not assignable to type 'number'.
funcA(obj);

// 元の定義が union types ならもちろん OK
declare function funcB(word: string | number): string;
funcB(obj);
```

この問題は Issue[*17]として管理されています。"Accepting PRs"ラベルがついているため、TypeScript チームが積極的に直す候補にはなっていないけれどコミュニティの誰かがやる気を出せば修正される、という状態です。

モジュールの定義の統合

あまり言及されることがないのでここで触れておきます。モジュールの型定義は open ended ですのでリスト 6.29 とリスト 6.30 のようなコードが書けます。めでたい。

リスト 6.29: モジュール定義を後から拡張可能

```
// モジュールの定義の統合ができます
declare module "foo" {
  let str: string;
}

declare module "foo" {
  let num: number;
}
```

リスト 6.30: 普通に使えます

```
import * as foo from "foo";
foo.str;
foo.num;
```

[*17] https://github.com/Microsoft/TypeScript/issues/5766

DefinitelyTyped ではモジュールの型定義の外側に namespace を使った定義を掃き出し、モジュールの型定義の外側に拡張ポイントを設ける例がありました。モジュールを利用しない namespace だけの構成です。たとえば、lodash や jQuery のようなグローバルな名前空間に変数を生やすような場合に、いまだに有効です。

any と{}と Object

もしも型定義ファイルを書いていて具体的な型がわからないとき、頭を使わずにとりあえずコンパイルを通したいときは、素直に any を使いましょう。こういったシチュエーションで、稀に Object を指定する人がいます。これは JavaScript の仕様として、プロトタイプチェーンの頂点にいる Object を使おう！ と思ったのでしょう。

関数の引数に Object や{}を指定するのは、どういう性質の値がほしいのかを述べていません。本当にどのような値でも受け入れるのであれば、any にするべきです。

関数の返り値に Object や{}を指定しても有用なプロパティが存在しないため型アサーションでもって適切な型にキャストするしかありません。これは any を指定するのと同程度に危険で、なおかつ any より検出しにくいです。素直に any を使いましょう。

筆者は今のところ、Object や{}が型注釈として適切な場面を見たことがありません。大抵の場合は、適切な型を定義してそちらを参照するほうが優れています。

そして any を使うことに気後れするのであれば、よくよく調べて適切な型定義を与えるようにしましょう。

ドキュメントから書き起こす

もしライブラリにしっかりしたドキュメントがあるのであれば、実装コードから型定義ファイルを起こすのではなく、ドキュメントをベースに作成しましょう。Visual Studio などの IDE では、型定義ファイル上に書かれた JSDoc コメントも利用時に表示してくれる場合があります。そのため、型定義を起こしつつ、あわせて JSDoc を記述していくとよいでしょう。

サンプルをテスト用コードとして TypeScript コードに移植し、ドキュメントどおりの記述が可能かも確かめるとよいです。型定義ファイルは書き起こしたけれどもドキュメント中に書かれている利用例のコードをコンパイルしてみて失敗するようであれば、それは悪い型定義だといえます。たまにドキュメントのほうが間違っている場合があるのでその場合は修正の

pull request を送るチャンスです。

世の中、ドキュメントにコストをあまり掛けることのできないプロジェクトも多くあるため絶対的なルールではありません。この場合、コードから型定義ファイルを起こすことになるのは仕方のないことです。

現在、DefinitelyTyped にある jQuery の型定義ファイルを熱心に面倒みてくれている John Reilly は特にドキュメントとの整合性を熱心に見ます。そのため、もし jQuery のドキュメント自体が間違っている場合は jQuery のドキュメントを直すところから始めるとよいでしょう。コントリビュートの輪！

コールバック関数の引数を無闇に省略可能（optional）にしない

optional とは、値が渡されるかどうかの指標であって、コールバックを受け取った側が使うかどうかではありません。ここを勘違いすると、"コールバックに値が渡されるが別に使わなくてもいいよ"マークとして optional を使ってしまうのです。

例を見てみましょう（リスト 6.31）。

リスト 6.31: optional はもしかしたら値がないことを表す

```
// 良い例
declare function readFileA(
  filePath: string,
  listener: (data: string) => void): void;
// 悪い例
declare function readFileB(
  filePath: string,
  listener: (data?: string) => void): void;

// 使ってみよう！
readFileA("./test.txt", data => {
  // ここでの data は必ず実体がある
  console.log(data.toUpperCase());
});
readFileB("./test.txt", data => {
  // ここでの data は undefined かもしれない… チェックしなければダメ
  if (!data) {
    data = "not found";
  }
  console.log(data.toUpperCase());
});
```

```
// 引数を無視するのは自由 optional にする理由にはならない
readFileA("./test.txt", () => {
  console.log("done");
});
readFileB("./test.txt", () => {
  console.log("done");
});
```

両方とも、ファイルの読み取りを行うための関数を型定義として書き起こしたものです。readFile は data が省略不可、readFileOpt は data が省略可能（optional）になっています。これは readFileOpt では data が undefined になるかもしれないことを表します。data が undefined かもしれないため、if 文などで中身をチェックし、undefined だった場合の対応を入れなければなりません。本当に undefined になりうるのであれば省略可能にするか、union types で undefined を与える必要があります。しかし、そうではなく必ず data の値が渡されてくる場合は、無用なチェック処理が発生することになります。

間違えないよう、留意しましょう。

インタフェースのプリフィクスとして I をつけるのはやめよう！

と TypeScript の公式ドキュメントで明記[*18]されました。

C#や Java よりも、広い範囲でインタフェースが利用されるので"実装を強制させるパーツ"扱いしてはいけないからだそうです。

古くは TypeScript コンパイラ本体のコードも C#の伝統に倣い I プリフィクスを使っていましたが、現在では取り除かれています。また DefinitelyTyped でも公式の記述に従い新しい型定義ファイルについては I プリフィクスを使わぬようレビューしています。

ECMAScript 2015 と CommonJS でのモジュールの互換性について

最初にまとめを書いておきます。まとめ： **元の JavaScript コード中に default の文字がないなら import の default は使うな。**

現在 JavaScript のモジュールの仕様は過渡期にあります。ECMAScript 2015 でモジュー

[*18] https://www.typescriptlang.org/docs/handbook/writing-declaration-files.html#naming-conventions

ルの記法や考え方は定義されましたが、実際ブラウザにはまだ実装されていません。ブラウザ上でのスクリプトの読み込みは煩雑で、まだ実装のための仕様も固まっていない段階です。さらに CommonJS 形式のモジュールとの互換性なんて、ECMAScript の仕様には含まれていません。

そのために TypeScript や Babel など、各種トランスパイラ毎に ECMAScript 2015 と CommonJS 間の変換方法は食い違っています。TypeScript が正しいのか Babel が正しいのかという議論は、そもそも仕様が不明なのだから成立しません。TypeScript も Babel も ECMAScript 2015 なモジュール記法から CommonJS 形式などへの変換ルールを定めているため、我々はその特徴を知り、正しく使いこなす必要があります。

まずは TypeScript で書いたコードがどのような CommonJS 形式のコードに変換されるかを見てみます（リスト 6.32、リスト 6.33）。

リスト 6.32: 関数などを素直に export する

```
export function hello(word = "TypeScript") {
  console.log(`Hello, ${word}`);
}

export function bye(word = "JavaScript") {
  console.log(`Bye, ${word}`);
}
```

リスト 6.33: CommonJS 形式では exports.xxx = となる

```
"use strict";
function hello(word = "TypeScript") {
    console.log(`Hello, ${word}`);
}
exports.hello = hello;
function bye(word = "JavaScript") {
    console.log(`Bye, ${word}`);
}
exports.bye = bye;
```

単純でわかりやすいですね。

次に CommonJS での `exports.module = ...;` 形式（export.module への代入）の記法

を見てみます（リスト 6.34、リスト 6.35）。

リスト 6.34: export = ... と書く

```
function hello(word = "TypeScript") {
  console.log(`Hello, ${word}`);
}

// CommonJS の exports.module = hello; 相当
// 外からこのモジュールを参照した時のオブジェクト自体を差し替える
export = hello;
```

リスト 6.35: exports.module = ... となる

```
"use strict";
function hello(word = "TypeScript") {
    console.log(`Hello, ${word}`);
}
module.exports = hello;
```

この変換は重要です。変換結果から逆に考えると JavaScript で `exports.module = ...;` の形式を見たら TypeScript では `export = ...;` という型定義に書き起こす必要があります。

理解を深めるため Node.js での CommonJS の実現方法について該当のコードを抜粋[*19]します（リスト 6.36）。

リスト 6.36: Node.js のモジュールの実現方法

```
NativeModule.wrap = function(script) {
  return NativeModule.wrapper[0] + script + NativeModule.wrapper[1];
};

NativeModule.wrapper = [
  '(function (exports, require, module, __filename, __dirname) { ',
  '\n});'
];
```

[*19] https://github.com/nodejs/node/blob/v6.3.1/lib/internal/bootstrap_node.js#L434-L441

大変シンプルなコードが出てきました。Node.js において、モジュール固有の変数というのはモジュールのオリジナルのコードの前後にあの 2 行を付け足して、eval しているだけなのです。なので、Node.js 初心者がたまにやりがちな `exports = ...;` というコードは間違いです。単に変数の値を差し替えているだけなので当然ですね。外部に変更を露出させるには、何かのプロパティの変更（つまり `module.exports = ...;`）でなければなりません。

互換性の話に戻ります。この `export = ...;` の記法に対応した"正規の"import の書き方は先ほど見た `import xxx = require("...");` 形式です。これを無理やり ECMAScript 2015 形式の import 文に書き直すとリスト 6.37 になります。

リスト 6.37: import モジュール全体 as 名前

```
// モジュール全体を util に割当て
import * as util from "./util";

// この書き方は誤り util.ts に default エクスポートはない
// error TS1192: Module '"略/util"' has no default export.
// import util from "./util";

// Hello, CommonJS と表示される
util("CommonJS");
```

このやり方は若干良くなく、`export =`する対象が変数ではない場合、エラーになるためワークアラウンドが必要です（リスト 6.38)。

リスト 6.38: 同名の namespace を被せてごまかす

```
function hello(word = "TypeScript") {
  console.log(`Hello, ${word}`);
}
// 呼び出し元でエラーになるのを防ぐ 同名の namespace を被せてごまかす
// error TS2497: Module '"略/util"' resolves to a non-module entity
//   and cannot be imported using this construct.
namespace hello { }

export = hello;
```

いまいち優雅ではありませんね。この場合は無理に ECMAScript 2015 のモジュール記法を使わないほうが無難かもしれません。世間的にも意見が分かれるところです。

さて、ここで問題になるのが TypeScript と Babel で `module.exports = ...;` 形式のモジュールを利用する際、どう ECMAScript 2015 形式にマッピングするかの解釈が異なる点です。Babel の変換結果を見てみます。リスト 6.39 をコンパイルするとリスト 6.40（リスト 6.41）となります。

リスト 6.39: Babel で変換する前のコード

```
import util from "./util";
util();
```

リスト 6.40: Babel で変換した結果のコード

```
"use strict";

var _util = require("./util");

var _util2 = _interopRequireDefault(_util);

function _interopRequireDefault(obj) {
  return obj && obj.__esModule ? obj : { default: obj };
}

(0, _util2.default)();
```

リスト 6.41: Babel で変換した結果をわかりやすく書き直す

```
"use strict";

var util = require("./util");
if (!util || !util.__esModule) {
  util = { default: util };
}

util.default();
```

Babel は、`module.exports = ...;` 形式のコードに対して特別な配慮を行い、`import util from "./util";` 形式でも動作します。TypeScript が `import * as util from "./util";` 形式しか許していないため、ここに齟齬があります。

ECMAScript 2015 形式＋ Babel のコードを TypeScript から参照したり、ECMAScript 2015 ＋ TypeScript のコードを Babel から参照したりすることには大きな問題はありません。しかし `module.exports = ...;` なコードの取り扱いには注意が必要なのです。

この話題は DefinitelyTyped でよくあるトラブルの 1 つで、TypeScript ＋ Babel の両方を組み合わせて使うユーザからこのあたりがごっちゃになったコードや修正が来ます。レビューする側としては「いやお前の環境では動くかもしれんが大抵のビルド手順では動かんのじゃ」となり、修正してくれるまで取り込むことはありません。TypeScript では `exports.default = ...` とされているコードのみ `export default ...` という型定義を与えてよいのです。**元の JavaScript コード中に default の文字がないなら import の default は使うな**。ということです。

CommonJS 形式でちょっと小難しい export 句の使い方

インタフェースやクラスのインスタンス単体をモジュールの外側に見せたい場合、リスト 6.42 のように書きます。

リスト 6.42: 実はインタフェース Bar も外から見えない

```
declare module "bar" {
  interface Bar {
    num: number;
  }

  // この_は外部からは参照できない。export してないので。
  let _: Bar;
  export = _;
}
```

呼び出し側ではリスト 6.43 のように使います。import した値がインタフェース Foo のインスタンスになっていることがわかります。

リスト 6.43: 使うとき。インタフェース Bar のインスタンスが得られる

```
// b は "bar" の Bar のインスタンス だよ！
import * as b from "bar";
b.num;
```

よくやりがちな誤りはリスト 6.44 のような書き方をしてしまうことです。インタフェースのインスタンスを export したつもりが型が export されてしまうのです。

リスト 6.44: それは値ではなくて型だけ export しているぞ！

```
declare module "buzz" {
  interface Buzz {
    num: number;
  }

  // よくやりがちな過ち
  export = Buzz;
}
```

こういう悲しい目を回避するには、型定義ファイルのテストが有効です。型定義ファイルを書いたら適当なユースケースに当てはめて意図どおりコンパイルできるか確かめてみましょう。

グローバルに展開される型定義とモジュールの両立

グローバルに変数が展開されるタイプとモジュールとしての利用が両立しているタイプのライブラリについて考えます。具体的に **UMD**（Universal Module Definition）と呼ばれる形式[*20]です。ライブラリ内部でモジュールとしての使い方が想定されているのか、そうではないのかを判断し展開の方法を変えます。

TypeScript ではこういうパターンのときに使いやすい型定義ファイルの記述方法があります。しかし、TypeScript 2.0.0 までは任意の場所においてある型定義ファイルを特定の名前のモジュールだと認識させる方法がなかったため、役に立ってはいませんでした。この形

[*20] https://github.com/umdjs/umd

式が使われているのは DefinitelyTyped の@types パッケージシリーズ（本書執筆時点では types-2.0 ブランチ）だけではないでしょうか。

説明のために strutil と strutil-extra という架空のライブラリについて考えてみます。strutil は randomizeString 関数を提供します。strutil-extra は happy 関数を提供し、strutil を拡張します。

まずは型定義ファイルを見てみましょう（リスト 6.45、リスト 6.46)。ちょっと見慣れない書き方ですね。

リスト 6.45: typings/strutil/index.d.ts

```
// import されなかった場合、global に strutil という名前で展開する
export as namespace strutil;

// 普通の型定義 declare module "..." の中と同じ書き味でよい
export interface Options {
  i?: number;
}
export declare function randomizeString(str: string, opts?: Options): string;

// グローバルな要素の拡張
declare global {
  // 既存の string 型にメソッドを生やす
  interface String {
    randomizeString(opts?: Options): string;
  }
}
```

リスト 6.46: typings/strutil-extra/index.d.ts

```
// 他のモジュールの型定義を参照する
import * as strutil from "strutil";

export as namespace strutilExtra;

export declare function happy(str: string): string;

// 他のモジュールの拡張
declare module "strutil" {
  // 既存の要素を拡張できる
  interface Options {
```

```
    reverse?: boolean;
  }

  // 自分ではないモジュールに勝手に新規の変数や関数を生やしたりはできない
  // 定義の拡張のみ可能
  // error TS1038: A 'declare' modifier cannot be used
  //   in an already ambient context.
  // export declare let test: any;
}

declare global {
  interface String {
    happy(): string;
  }
}
```

既存モジュールの定義の拡張もできています。この形式だと、どのライブラリを拡張しているのか明示するところが利点です。

これらを `import ... from "strutil";` したりするための tsconfig.json を確認しておきます（リスト 6.47)。

リスト 6.47: tsconfig.json の例

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": true,
        "baseUrl": "./",
        "paths": {
            "strutil": ["./typings/strutil/"],
            "strutil-extra": ["./typings/strutil-extra/"]
        }
    },
    "exclude": [
        "node_modules"
    ]
}
```

baseUrl と paths の指定があります。TypeScript 2.0.0 からこうして任意の場所の型定義

ファイルを任意の名前に紐付けられるようになったため、ローカル環境でも利用しやすくなりました。

次に前述の型定義ファイルを利用する例を見てみます。まずはグローバルに展開される例です（リスト 6.48)。

リスト 6.48: lib/bare.ts

```
// UMD 形式のライブラリが global に展開されたときの動作に相当する
// import, export 句がない場合、global の strutil が参照できる
strutil.randomizeString("TypeScript");
strutilExtra.happy("TypeScript");

// global の String も拡張されている
"TypeScript".randomizeString();
"TypeScript".happy();

// import、export が存在すると、ちゃんと読み込め！ と怒られる
// error TS2686: Identifier 'strutil' must be imported from a module
// error TS2686: Identifier 'strutilExtra' must be imported from a module
```

なるほど。`export as namespace ...` 形式を使わない UMD 形式の対応方法もありますが、import と混ぜるとエラーになるところがよいですね。

モジュール形式も見てみましょう（リスト 6.49)。普通にモジュールであるかのように利用できますね。

リスト 6.49: lib/module.ts

```
// UMD 形式のライブラリが global に展開されたときの動作に相当する
// import した時、普通のモジュールとして振る舞う
import { randomizeString } from "strutil";
import { happy } from "strutil-extra";

randomizeString("TypeScript");
happy("TypeScript");

// strutil-extra で追加したパラメータも反映されている
randomizeString("TypeScript", {
  i: 11,
  reverse: true, // これ
});
```

```
// global の String も拡張されている
"TypeScript".randomizeString();
"TypeScript".happy();
```

この形式がどこまで普及するかはわかりませんが、時とともに DefinitelyTyped 内部でも見かける頻度が増えていくでしょう。ファイル名を見ただけではどういう名前に解決されるかがわかりにくいところだけ、注意が必要です。

最終チェック！

やった！ 型定義ファイルが書けたぞ！ 出来高に満足する前に、もう少しだけやっておきたいことがあります。それが、--noImplicitAny や--strictNullChecks をつけての試しコンパイルと tslint によるチェックです。

tslint

lint という種類のプログラムがあります。ざっくり、プログラムを静的に解析してバグになりそうな箇所や悪いコードスタイルを見つけてくるツールを指します。

TypeScript では tslint[*21] というプログラムが一般的に使われています。

tslint はコンパイルだけでは見つけきれない、悪いにおいのするコードを検出してくれます。tslint では頻繁に新しいルールが追加されるため、本書では詳しくは取り上げません。その時々の最適な設定を突き詰めてみてください。

tslint は設定ファイルを必要とします。今のところ、TypeScript における統一見解は存在していないので tslint が使ってる設定ファイル[*22]か TypeScript 本体の tslint.json[*23]を参照するとよいでしょう。

6.6 Let’s contribute!

ようこそ！ DefinitelyTyped[*24]へ！ メンテナの vvakame です。

[*21] https://github.com/palantir/tslint
[*22] https://github.com/palantir/tslint/blob/master/tslint.json
[*23] https://github.com/Microsoft/TypeScript/blob/master/tslint.json
[*24] https://github.com/DefinitelyTyped/DefinitelyTyped

DefinitelyTyped ではさまざまな型定義ファイルを取り揃えてございます！ 世界中の人々が作った型定義ファイルは集積され、@types などを介して広く利用されています。

貴方が作った型定義ファイルも世界中の人々に使ってほしいとは思いませんか？ もしくは、あなたがいつも使っている型定義ファイルのバグを治したい…そんな気持ちになることもあるでしょう。その思い、すべて DefinitelyTyped にぶつけてみましょう！

本書を読んでいただいた紳士淑女の皆様は、感じのよい型定義ファイルが書けるようになっています。品質と時間のトレードオフを考えつつ、上品な型定義ファイルを提供していただきたいです。

DefinitelyTyped は GitHub 上のリポジトリなので追加、修正については pull request をご利用ください。

この節では、筆者が DefinitelyTyped に送られてきたときにどういう考えで pull request のレビューをしているかを示します。あくまで、ここに書いてあることは筆者固有の観点なので、レビュワーによって別のことを言われる場合もあるでしょう。実際に pull request を送ってみて、ここに書いてある以外の理由で拒否されたとしても、そこは実際のレビュワーを尊重して議論していただきたいと思います。

とはいえ、メンテナは全員 DefinitelyTyped の Contribution guide[*25]は読んでいるはずなので、こちらには目を通しておくとよいでしょう。

新規型定義ファイルの追加のレビューの観点

まずは今までなかった、新しいライブラリに対する型定義ファイルのレビューの観点を解説していきます。

1. CI が通っているか
2. npm または bower に公開されている名前どおりか。公開されていない場合は競合が発生しないか
3. テストが存在しているか
4. 幽霊 namespace を使ったほうが構造がきれいになるか

だいたいこんな感じです。

[*25] http://definitelytyped.org/guides/contributing.html

CI が通っているか。これは、ヘッダが定められた形式で書かれているか、--noImplicitAny 付きで型定義ファイルやテストがコンパイルできるか、を主に見ています。

npm, または bower に公開されている名前どおりか。npm に公開されているライブラリは npm で公開されている名前と同一のディレクトリ名にあわせます。もし、npm に公開されていない場合は適当に名前を選ぶしかありませんが、同名の別のライブラリが npm 上に存在していないかなどをチェックしています。

逆に、ここに書かれていないことはあまり見ていません。たとえば、ライブラリの実装全体に対する型定義ファイルのカバー率や any の多さなどはあまり見ていません。それらは後から別の人が補ってくれる可能性があるからです。一人でやりきらなくてもいいよな！ という発想ですね。もちろん最初に高品質高カバー率のものが出てきたほうが「やりおる！」と感心はします。

なおレビュワー次第ですが JSDoc がきっちり書かれているか、というのを見る人もいます。きちんとドキュメントから転記などしてあるものが送られてきたときはやはり感心しますね。

既存型定義ファイルの修正のレビューの観点

1. CI が通っているか
2. 破壊的変更が含まれていないか
3. ライブラリ本体のドキュメントまたは実装と照らして変更内容が妥当であるか

これだけです。新規追加の場合は比較的レビューがガバガバなのですが、既存のものの変更はすでに利用している人に影響があるため、勢い慎重になってしまいます。そのため結構レビューに時間が取られて辛いので、ドキュメントや実装が確認できる URL を添えてくれると大変嬉しいです。

いくつか補足しましょう。

破壊的変更が含まれていないか。たとえばコードスタイルの変更（インタフェースのプリフィクスに I をつける、つけない など）や、幽霊 namespace を使っていないスタイルから使っているスタイルへの変更など。または、ある型定義を別の表現へと書き換える場合。これらはレビュアーが妥当かどうかを判断します。たいてい判断できないのでヘッダに書いてある Definitions by:に名前が書いてある人達に GitHub 上で mention が飛ばされ、レビューしてもらって決めます。型定義ファイルを作った人は、たいていの場合、実際に使っている人、

つまり破壊的変更を加えられると困る人達です。変な変更が入りそうな場合、きっと事前に相談してほしいに違いないと考えるわけです。

次に、変更内容が妥当であるかの検証です。多くの場合は筆者が当該ライブラリのドキュメントまたは実装を読んで正しいかどうかを verify します。しかし、小規模でドキュメントの URL がちゃんと添付されている場合以外は、前述のとおり Definitions by:の人たちにレビューの依頼を投げます。

では、皆様の pull request、お待ちしています！

6.7 自分のライブラリを npm で公開するときのベストプラクティス

自分の作ったライブラリを npm に公開する時のベストプラクティスについて説明します。ここで説明するのは TypeScript によってコードが書かれているライブラリを前提とします。また、npm にパッケージを公開するための基本的な説明はここでは行いません。

ポイントは.ts ファイルをリリースに含めないこと、.d.ts ファイルを TypeScript コンパイラに生成させること、.d.ts ファイルを TypeScript コンパイラが自動的に見つけられるようにすることです。

まずは **.ts ファイルをリリースに含めない**理由について説明します。これは、TypeScript コンパイラの探索順序が.ts ファイル、.tsx ファイル、.d.ts ファイルだからです。.d.ts ファイルも公開していたとしても、.ts ファイルが存在しているとそちらが先に発見され、コンパイル処理が走ってしまいます。TypeScript コンパイラのバージョンが上がった時にソースコード（.ts）の修正が必要になるケースは多いですが、型定義ファイル（.d.ts）が影響を受けるケースは稀です。つまり、自分のライブラリをより安定したものとするためには、.ts ファイルをリリースに含めないほうがよいわけです。そのために.npmignore ファイルにリスト 6.50 の記述を追加します。

リスト 6.50: .npmignore で.ts コードを排除し.d.ts はパッケージング対象へ

```
# lib ディレクトリ配下でコードが管理されている場合
lib/**/*.ts
!lib/**/*.d.ts
```

.d.ts ファイルを TypeScript コンパイラに生成させるのは、.ts コードをコンパイルするときに`--declaration`オプションを利用するだけなので簡単です。

つぎに **.d.ts ファイルを TypeScript コンパイラが自動的に見つけられるようにする**理由ですが、これは単純にそのほうが使いやすいからです。実現するためには TypeScript コンパイラの検索パスに自身の型定義ファイルが入るようにします。

そのための方法はいくつかあります。

1. パッケージの root に index.d.ts を置く
2. package.json に typings プロパティを作成し、最初に参照するべき型定義ファイルの相対パスを書く
3. package.json に types プロパティを作成し、最初に参照するべき型定義ファイルの相対パスを書く

1 つ目は Node.js が実行時にパッケージの root にある index.js を最初に読み込もうとする挙動に似せた動作です。2 つ目と 3 つ目はほぼおなじやり方ですが、3 つ目のほうが最近追加されたやり方です。typings と types プロパティの両方が存在する場合は typings プロパティが優先されます。

筆者はもっぱら、1 つ目の方法を使い index.d.ts と index.js を手書きしています。これは package.json に色々と書くよりも一般的なルールに従うのを良しとしているためです。

実例については筆者の typescript-formatter[*26]リポジトリを参照してください。

[*26] https://github.com/vvakame/typescript-formatter

しめくくり

編集・校正の@mhidaka が「書籍の終わりが唐突すぎる！」というのであとがきです。

筆者（@vvakame）は商業出版として 2 年前に TypeScript 1.0.0 の本を、コミックマーケット 87 で 1.3.0〜1.4.1 の本を書いています。そこから 1 年以上、TypeScript についてまとめたものを出していませんでした。最近は TypeScript 2.0.0 の影も濃く感じます。さまざまな feature が追加され、昔よりも断然開発しやすくなったことを喧伝すべく、ここに筆を取った次第です。あと、技術書典は準備するばかりで 1 章も書かなかったら@mhidaka がぷんぷん！ とか言い始めたので。最新の TypeScript について、日本語で読めるまとまった資料として本書が皆様の役に立てば幸いです。

表紙のイラストを描いていただいたのは shati 氏、デザインは siosio 氏です。ありがとうございます！ 聞けば終盤極道スケジュールになったらしく、なんかすみませんでした…

さて、今回は「ひかる黄金わかめ帝国」としての参加です。C89 までは@mhidaka 率いる「TechBooster」と@consomme72 率いる「くずかごのーと」の合体参加でした。しかし、@consomme72 がドール者として覚醒したためジャンルを変更し分離しました（なお、落選した模様）。代わりに我が帝国です。命名について、雑に「わかめ帝国」にしよう。それだけだと弱そうだし、ひかるといいと思うし、ひかるなら黄金なんだろうなー、という気持ちで「ひかる黄金わかめ帝国」になりました。まぁ、当日のブース運営は@_mochicon_が全部やってくれるでしょう。うほほほい。

本文中では紹介していなかったのですが筆者が作っている typescript-formatter について思い出話を書きます。2 年前の執筆時、サンプルコードのフォーマットを統一するのに何かツールがほしいなーと思って作ったのが typescript-formatter でした。おかげ様で予想外に息の長いツールになりまして、2016 年 8 月 3 日時点で 4,596 ダウンロード/日、23,535 ダウンロード/週、116,432 ダウンロード/月されています。使い方はとても簡単で、`tsfmt -r` とするだけで tsconfig.json やらを読んでコンパイル対象になる.ts ファイルらをフォーマットしてくれます。TypeScript はスーパーエクセレントなコンパイラなので、コードをフォーマットする機能を内蔵しています。なので、typescript-formatter はその機能に対して CLI やら設定の処理などをしているだけの薄いツールです。

本書のまとめ：TypeScript ってスゴイ。

Revised TypeScript in Definitelyland

2016 年 8 月 14 日　C90 版 v2.0.0

2016 年 8 月 14 日　電子書籍版 v2.0.0

著　者　vvakame

編　集　mhidaka

発行所　ひかる黄金わかめ帝国

印刷所　ひかる黄金わかめ帝国