TypeScript製ライブラリと開発体験

はじめに

自己紹介

- 伊藤　瑛（いとう　あきと）
- @Akito0107 Twitter / Github
- TypeScript / Go をよく書いています
- メインはバックエンド

TypeScript とわたし

- 一番最初に触り始めたのは使い始めたのは 2016 年ごろ
  - Node.js 製のサーバサイドアプリケーションの運用をしていた
  - Runtime Error に苦しめられる日々
  - 当時の TypeScript は Third Party 製のライブラリが充実しておらず、余計辛かったので使うのを諦める
- Flow に行ったりしたが、2018 年の春頃から本格的に TypeScript を触り始める
- TypeScript でツール・ライブラリを作るのが趣味
- ライブラリ屋？

今日伝えたいこと

- TypeScriptの型推論を活用すると、どういったAPIが実現できるのか
- 実例ベースで型のトリックとともにいくつか紹介します
- なにか１つでもヒントになれば幸いです！

まえおき

- 型パズルがいくつか出てくる
- こういうこともできるよ、という話
- **すべての TS のコードでこういったことをしよう、という話ではない**

TypeScriptでLibraryを書くメリットはなにか

- Libraryを書く人
  - 型に守られるため堅牢性/メンテナンス性/etcが上がる
- Libraryを使う人
  - (型が正しく提供されていれば)APIの仕様把握のコストが減る
  - IDEの補完による高いDX

TSでしかできないようなDXを実現してみたい

command line parser catacliの紹介

catacli

- github
- TypeScript向けに書いたCommand Line Parser(Node.jsでCLI書くときに使うやつ)
  - commander.js, minimist, yargsなどと同じジャンル
- 型推論がかなり強く効く
- 詳しくはこちら
- (名前変えました)

motivation

- commander.js のAPI(Githubより)

```
const program = require('commander');

program
  .option('-d, --debug', 'output extra debugging')
  .option('-s, --small', 'small pizza size')
  .option('-p, --pizza-type <type>', 'flavour of pizza');

program.parse(process.argv);

if (program.debug) console.log(program.opts());
if (program.small) console.log('- small pizza size');
if (program.pizzaType) console.log(`- ${program.pizzaType}`);
```

- 文字列で flag を定義していくスタイル
- flagが増えたときや、subcommandとか増えだすと辛い

flagを全部型推論させたい！！！

Demo

playground

型のトリック

- ①: String Literalをkeyにしたobjectを返す型
- ②: 可変長引数の型推論

①: String Literalをkeyにしたobjectectを返す型

- ユーザが引数に与えた文字列のpropを持ったobjectの型として推論させる

```
// --test <number> のflagを定義したい
const f = makeNumberFlag("test");
// resは { test: number; }として推論される
const res = f(["test", "123"])
```

- Flag(Parser)の型定義(抜粋)
  - type Flag はコマンドライン引数(process.argv)を受け取り、parseした結果を返す関数(の型定義)

```ts
type Flag<T, N extends string> = (args: string[]) => ParseResult<T, N>;
type ParseResult<T, N extends string> = {
  [key in N]: {
    value?: T;
  };
};
```

- 実際のFlagParserはこの型を利用して実装されている

```ts
export type NumberFlag<N extends string> = Flag<number, N>;

export function makeNumberFlag<N extends string>(name: N): NumberFlag<N> {
  return (args: string[]) => {
      // ~ parserのlogicが入る ~
    return {
      [name]: { value: parseInt(v, 10) }
    };
  };
}
```

- ポイントは**<N extends string>**

TypeScriptのextends

- 特に型パラメータで使われた場合、継承というより、型の制約条件を表すイメージ(参考)
- `N extends string` の `N` は `string` を継承した型というよりも、 `string` の型に包含される型という認識のほうが近い
- `string literal type` (javascriptのstringの値そのものの型)はstringに包含される

```ts
type A<N extends string> = {};

type B = A<"test">; // OK
type C = A<"test1" | "test2">; // OK
type D = A<string>; // OK
type E = A<123>; // NG
```

mapped type

- 参考
- for in に近い動きをする

```
type M<N extends string> = {
    [K in N]: string;
}

type B = M<"test">; // { test: string; }として推論される
type C = M<"test1" | "test2">; // { test1: string; test2: string; }として推論される
type D = M<string>; // { [x: string]: string; }として推論される
```

これらを応用してみる

- TypeScriptの関数の型パラメータは引数の型をcaptureしてくれる 参考
- `type argument inference`

```
function makeNamedProps<N extends string>(propName: N): {[key in N]: string; } {
    // 割愛
}

const res = makeNamedProps("test") // {test: string;}として推論される

const res2 = makeNamedProps<string>("test") // {[x: string]: string;}として推論される
```

②: 可変長引数の型定義

- 任意のflag parserを組み合わせて、複数のflagをparseできるようにしたい
- `reduceFlag` の型定義について

```
const argv = ["--test", "test", "--test2", "test2"]

const f1 = makeStringFlag("test") // (arg: string[]) => ({test: string;});
const f2 = makeStringFlag("test2") // (arg: string[]) => ({test2: string;}):

const reduced = reduceFlag(f1, f2) // (args: string[]) => ({test: string; test2: string;});

const res = reduced(argv); // {test: string; test2: string; }と推論させたい
```

引数が2個の場合

- 任意の関数を2つうけとり、返り値を合成して返す関数
- それぞれの関数の返り値を型パラメータで受け取り, intersection type を用いる

```
function mergeFunction<R1, R2>(f1: (x: any) => R1, f2: (x: any) => R2): (x: any)  => R1 & R
    // 割愛
}

const f1 = () => ({test: "test"});
const f2 = () => ({test2: "test2"});

const fm = mergeFunction(f1, f2); // (x: any) => ({test: "test"} & {test2: "test2"}) として
```

Flagで書いてみる

- T1 / T2をそれぞれの引数のFlag(Parser)の型パラメータに渡す

```
function merge2<T1, T1Name extends string, T2, T2Name extends string>(
  t1: Flag<T1, T1Name>,
  t2: Flag<T2, T2Name>
): Flag<{ [key in T1Name]: T1 } & { [key in T2Name]: T2 }, T1Name | T2Name> {
    // 割愛
}

const f1 = makeStringFlag("test")
const f2 = makeStringFlag("test2")
const merged = merge2(f1, f2);

const res = merged(argv) // {test: string; test2: string;} として推論される
```

引数が3個の場合

- 愚直にやるとこう

```
function merge3<T1, T1Name extends string, T2, T2Name extends string, T3, T3Name extends st
  t1: Flag<T1, T1Name>,
  t2: Flag<T2, T2Name>,
  t3: Flag<T3, T3Name>
): Flag<{ [key in T1Name]: T1 } & { [key in T2Name]: T2 } & { [key in T3Name]: T3 }, T1Name
    // 割愛
}
```

引数がN個の場合

- 想定される引数の数だけ型定義を書いておく（あるいは生成する）のはひとつのベストプラクティス
- 本当に任意の数の引数を取るような関数の型定義を正しく書くのは **すごく大変** (あとで話します)

(参考)yargs

- api styleの違い
- method chain likeなapi

```
import * as yargs from 'yargs';

yargs.command('serve', "Start the server.", (argv) => {
    /* ここの関数の返り値がそのままhandlerで推論されるようになる */
    return argv.option('port', {
        describe: "Port to bind on",
        default: "5000",
    }).option('verbose', {
        alias: 'v',
        default: false,
    });
}, (args) => {
    /* argsの */
    if (args.verbose) {
        console.info("Starting the server...");
    }
    (args.port);
});
```

method chainの型定義

- 関数型チックにやろうとするよりはかなり型は書きやすい
- N個の引数のときに苦しまなくても良い
- (あまり使い勝手も変わらない)

```
class FlagParser<T extends object> {
    opts: T;

    constructor(init: T) {
        this.opts = init;
    }

    addStringFlag<N extends string>(name: N): FlagParser<T & {[key in N]: string;}> {
        // 割愛
        return <any>this
    }

}

const parser = new FlagParser({});
```

tree shakingとmethod chain

- **(今回はNode.jsのCLIライブラリの話なので関係ない)**
- method chain styleのAPIはtree shakingが効きづらい傾向にある
  - それぞれのmethodを独立してexportするのが難しい
    - side effectsの問題
- 副作用のない関数を組み合わせて機能を作っていくほうがtree shaking的には有利

```
export function stringFlag(){}
export function numberFlag(){} // numberFlagは使用されていなければbundle時に消える
```

可変長引数は諦めきれない

- こうなっちゃう

```
f(d, f(c, f(a, b)))
```

- こう書きたい

```
f(d, c, b, a);
```

可変長引数の型推論

pipeNで考えてみる

- 任意の数の関数を受け取り、関数を返す関数

```
// (a: number) => stringに推論されてほしい
const piped = pipeN(
    (a: number) => (`${a}`),
    (b: string) => ({key: b}),
    (c: {key: string}) => (c.key),
);
```

  - (|> こんな演算子になるかもしれない)

可変長引数のハンドリング第一歩

- 可変長引数は**tuple**として扱える
- TypeScriptには `tuple` という型がある 参考
- Arrayとの違いは、長さが固定であること、それぞれの要素の型が固定されていること

```
let x: [string, number];

x = ["hello", 10]; // OK
x = [10, "hello"]; // Error
```

```
function<A extends Array<any>> tupleTest(...a: A): A {...}

tupleTest(1, 2, 3, 4, 5) // [number, number, number, number, number]として推論される
```

Tuple操作のイディオムを覚えよう

- Tuple操作の型のイディオムは探すと結構出てくる
- 外部ライブラリを頼るのが良さそう
- ts-toolbeltにだいたい揃ってる

```
// Tupleの先頭の要素をとってくる
type Head<T extends Array<any>> = ((...args: T) => any) extends (x: infer Head, ...tail: ar

type H = Head<[1, 2, 3, string]> // H = 1

// Tupleの先頭以外をとってくる
type Tail<T extends Array<any>> = ((...args: T) => any) extends (x: any, ...tail: infer Tai

type T = Tail<[1, 2, 3, string]> // T = [2, 3, string]

// Tupleの一番最後の要素をとってくる
type Last<T extends any[]> = T[Exclude<keyof T, keyof Tail<T>>];

type L = Last<1, 2, 3, string]> // L = string
```

あとはくっつける！

- (conditional typeを使っています)

```
function pipeN<A extends Array<Fn<any, any>>>(...fns: A):
  Head<A> extends (a: infer Arg) => any ?
    Last<A> extends (a: any) => infer R ?  (a: Arg) => R : never : never {
    // 実装は割愛
}

const piped = pipeNT(
    (a: number) => (`${a}`),
    (b: string) => ({test: b})
); // (a: number) => ({test: string}) として推論される
```

ここまではできた

最高のpipeNを目指して

- 引数の関数同士に推論を効かせたい

```
const piped = pipeNT(
    (a: number) => (`${a}`), // 返り値の型はstring
    (b: string) => ({test: b}) // 注釈なくても引数はstringとして推論してほしい
    (c: {test: b}) => 123 // cも推論してほしい...
);
```

これがまだできていない

ramdaのpipe

- ramda.jsの pipe を見てみる
- できてる

```
import {pipe} from 'ramda'

const piped = pipe(
  (a: number) => (`${a}`), // 返り値の型はstrig
  (b) => {
    return  {test: b} // bはstringとして推論
  },
  (c) => c.test // cは{test: b}として推論
);
```

なぜなら

結論

- ある程度は頑張れるものの、やっぱり引数が想定される分だけ型定義を用意したほうが推論的にもよさそう
- ramda / Rxjsはそういうアプローチ

終わりに

まとめ

- ライブラリを作るときに使ったTypeScriptの型推論のトリックをいくつか紹介しました
- method chainと関数の合成
- 可変長引数の取り扱い方

ありがとうございました