CLIツール・ウェブサーバー作成のための環境設定

TypeScriptを使ってCLIツールやウェブサーバーなどのNode.jsやDenoがある環境で動作するソフトウェアを作成するための環境構築方法を紹介していきます。2つ方法があります。

- Node.jsがインストールされている環境向け
- Denoががインストールされている環境向け

どちらも、TypeScriptをJavaScript変換して1ファイルにまとめたファイルを作成します。

Node.jsの環境構築

Node.js向けには@zeit/nccを利用します。

@zeit/nccはTypeScriptに最初から対応したコンパイラで、コマンドラインツールやウェブサーバーなどのNode.jsアプリケーションを1ファイルにするために作られています。@zeit/ncc自体も自分自身を使ってバンドル化されており、ごく小さいサイズですばやくインストールできます。Goを参考にしており、`tsconfig.json`以外の設定は不要です。

他に実績のあるツールとしてはBrowserifyがあります。これも、tsifyというプラグインと併用することでTypeScript製アプリケーションをバンドルして1ファイルにできます。このツールは、Node.js製アプリケーションをブラウザでも動くようにすることをメインの目的として作られました。後発の@zeit/nccに比べると設定が多いのと、TypeScriptの設定を変えてcommon.js形式のパッケージにしないと扱えないのがnccと比べると煩雑です。また、`tsc`を使いバンドルせずにTypeScriptをJavaScriptに変換するだけを行って配布する方法もありますが、バンドルをしないとさまざまなファイルが入ってくるため、配布サイズも大きくなりがちです。

作業フォルダを作る

フォルダを作成し、JavaScriptのいつものプロジェクトのように`npm init -y`を実行して`package.json`を作成しましょう。

ライブラリの時は、ES2015 modulesとCommonJSの2通り準備しましたが、CLIの場合はNode.jsだけ動かせば良いので、出力先は一つになります。コンパイルしたファイル置き場は`dist`フォルダになりますが、コンパイル時に自動で作られるので作成しておく必要はありませんが、`.gitignore`には登録しておきましょう。

ビルド設定

まずは@zeit/nccをインストールします。

Node.jsの機能を使うことになるため、Node.jsのAPIの型定義ファイルは入れておきましょう。TypeScriptはnccにバンドルされていますが、バンドルされている処理系のバージョンが古いことがあります。インストールしてあるTypeScriptを優先的に使ってくれるので入れておくと良いでしょう。

コマンドラインで良く使うであろうライブラリを追加しておきます。 カラーでのコンソール出力、コマンドライン引数のパーサ、ヘルプメッセージ表示です。

どれもTypeScriptの型定義があるので、これも落としておきます。 また、ソースマップサポートを入れると、エラーの行番号がソースのTypeScriptの行番号で表示されるようになって便利なので、これも入れておきます。

```
$ npm install --save-dev @zeit/ncc typescript

$ npm install --save-dev @types/node @types/cli-color
   @types/command-line-args @types/command-line-usage
   @types/source-map-support

$ npm install --save cli-color command-line-args
   command-line-usage source-map-support
```

TypeScriptのビルド設定のポイントは、ブラウザからは使わないので、ターゲットのバージョンを高くできる点にあります。 ローカルでは安定版を使ったとしてもNode.js 10が使えるでしょう。

ライブラリのときとは異なり、成果物を利用するのはNode.jsの処理系だけなので、.d.tsファイルを生成する必要はありません。

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2018",        // お好みで変更
    "declaration": false,      // 生成したものを他から使うことはないのでfalseに
    "declarationMap": false,   // 同上
    "sourceMap": true,         //
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",      // Node.jsで使うため
    "outDir": "./dist"         // 出力先を設定
  },
  "include": ["src/**/*"]
}
```

`package.json` 設定時は、他のパッケージから利用されることはないため、 `main` / `modules` / `types` の項目は不要です。 代わりに、 `bin` 項目でエントリーポイント（ビルド結果の方）のファイルを指定します。 このキー名が実行ファイル名になります。

package.js

```
{
  "bin": {
    "awesome-cmd": "dist/index.js"
  },
  "scripts": {
    "build": "ncc build main.ts --minify --v8-cache --source-map",
    ”watch": "ncc build main.ts --watch",
    "start": "ncc run main.ts",
    "lint": "eslint .",
    "fix": "eslint --fix ."
  }
}
```

`ncc build` でビルドします。 `ncc run` でビルドして即座に実行します。設定ファイルはありません。必要十分な機能が揃っています。オプションで指定できるものは以下の通りです。

- `--minify` : ミニファイしてファイルを小さくする
- `--source-map` : ソースマップを出力
- `-e, --external [mod]` : バンドルをスキップするモジュール
- `--watch` : 変更を検知してビルド
- `--v8-cache` : V8のコンパイルキャッシュを生成

テストの設定、VSCodeの設定は他の環境の設定と変わりません。

もし、バイナリを入れる必要のあるライブラリがあると、ビルド時にエラーになります。その場合は、そのパッケージを `--external パッケージ名` で指定してバンドルされないようにします。 ただし、この場合は配布環境でこのライブラリだけはnpm installしなければなりません。

注釈

Node.js/Deno以外の処理系

また、low.js [1] という、ES5しか動かないもののNode.jsと一部互換性があるモジュールを提供し、ファイルサイズがごく小さいインタプリタがありますが、これと一緒に使うこともできます。

low.jsはES5までしか対応しないため、出力ターゲットをES5にする必要があります。

tsconfig.json (low.jsを使う場合)

```
{
  "compilerOptions": {
    "target": "es5",              // もしlow.jsを使うなら
    "lib": ["dom", "es2017"]      // もしlow.jsで新しいクラスなどを使うなら
  }
}
```

これで、TypeScript製かつ、必要なライブラリが全部バンドルされたシングルファイルなスクリプトができあがります。

1 https://www.lowjs.org/

CLIツールのソースコード

TypeScriptはシェバング(#!)があると特別扱いしてくれます。 必ず入れておきましょう。 ここで紹介したcommand-line-argsとcommand-line-usageはWikiで用例などが定義されているので、実装イメージに近いものをベースに加工していけば良いでしょう。

```
index.ts
#!/usr/bin/env node

import * as clc from "cli-color";
import * as commandLineArgs from "command-line-args";
import * as commandLineUsage from "command-line-usage";

// あとで治す
require('source-map-support').install();

async function main() {
    // 内部実装
}

main();
```

Node.jsのまとめ

コマンドラインツールの場合は、npmで配布する場合はライブラリ同様、バンドラーを使わずに、TypeScriptだけを使えば大丈夫です。 ここにある設定で、次のようなことが達成できました。

- TypeScriptでCLIツールのコードを記述する
- 使う人は普段通りnpm installすれば実行形式がインストールされ、特別なツールやライブラリの設定をしなくても利用できる。

また、おまけで1ファイルにビルドする方法も紹介しました。

`package.json` の `scripts` のところに、開発に必要なタスクがコマンドとして定義されています。npmコマンドを使って行うことができます。すべてライブラリと同じです。

```
# ビルドして実行
$ npm start

# ビルドしてパッケージを作成
$ npm run build
$ npm pack

# テスト実行 (VSCodeだと、⌘ R Tでいける)
$ npm test

# 文法チェック
$ npm run lint

# フォーマッター実行
$ npm run fix
```

Deno

Denoは新しい処理系です。Node.jsと同じくV8をベースとしている兄弟処理系ですが、TypeScriptにデフォルトで対応していたり、ネイティブコード部分はRustを使って実装されています。 サードパーティのパッケージはnpmコマンドではなくdenoコマンドを使ってダウンロードします。

Node.jsを置き換えるものかというと、現時点ではまだパッケージやツールが足りていません。特にNode.jsのnpmはウェブフロントエンドのライブラリなどの配布にも使われており、ウェブフロントエンドの開発のためのプラットフォームとしても活用されています。Denoはコマンドラインツールやウェブサービスの開発に特化しています。

こちらもGoを参考にしたコマンドを持っています。それ単体でビルドして配布できます。