ライブラリ開発のための環境設定

現在、JavaScript系のものは、コマンドラインのNode.js用であっても、Web用であっても、なんでも一切合切npmjsにライブラリとしてアップロードされます。 ここでのゴールは可用性の高いライブラリの実現で、具体的には次の3つの目標を達成します。

- 特別な環境を用意しないと（ES2015 modules構文使っていて、素のNode.jsでnpm installしただけで）エラーになったりするのは困る
- npm installしたら、型定義ファイルも一緒に入って欲しい
- 今流行りのTree Shakingに対応しないなんてありえないよね？

今の時代でも、ライブラリはES5形式で出力はまだまだ必要です。 インターネットの世界で利用するには古いブラウザ対応が必要だったりもします。

ライブラリのユーザーがブラウザ向けにwebpackとBabelを使うとしても、基本的には自分のアプリケーションコードにしかBabelでの変換は適用しないでしょう。 よく見かける設定はBabelの設定は次の通りです。 つまり、配布ライブラリは、古いブラウザなどでも動作する形式でnpmにアップロードしなければならないということです。

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: "babel-loader",
        exclude: /node_modules/
      }
    ]
  }
};
```

なお、ライブラリの場合は特別な場合を除いてBabelもwebpackもいりません。CommonJS形式で出しておけば、Node.jsで実行するだけなら問題ありませんし、基本的にwebpackとかrollup.jsとかのバンドラーを使って1ファイルにまとめるのは、最終的なアプリケーション作成者の責務となります。特別な場合というのは、Babelプラグインで加工が必要なJSS in JSとかですが、ここでは一旦おいておきます。ただし、JSXはTypeScriptの処理系自身で処理できるので不要です。

TypeScriptユーザーがライブラリを使う場合、バンドルされていない場合は `@types/ライブラリ名` で型情報だけを追加ダウンロードすることがありますが、npmパッケージにバンドルされていれば、そのような追加作業がなくてもTypeScriptから使えるようになります。 せっかくTypeScriptでライブラリを書くなら、型情報は自動生成できますし、それをライブラリに添付する方がユーザーの手間を軽減できます。

Tree Shakingというのは最近のバンドラーに備わっている機能で、`import`と`export`を解析して、不要なコードを削除し、コードサイズを小さくする機能です。 webpackやrollup.jsのTree ShakingではES2015 modules形式のライブラリを想定しています。

一方、Node.jsは--experimental-modulesを使わないとES2015 modulesはまだ使えませんし、その場合は拡張子は`.mjs`でなければなりません。 一方で、TypeScriptは出力する拡張子を.mjsにできません。 また、Browserifyを使いたいユーザーもいると思いますのでCommonJS形式も必須です。

そのため、モジュール方式の違いで2種類のライブラリを出力する必要があります。

ディレクトリの作成

前章のディレクトリ作成に追加して、2つのディレクトリを作成します。

```
$ mkdir dist-cjs
$ mkdir dist-esm
```

出力フォルダをCommonJSと、ES2015 modules用に2つ作っています。

これらのファイルはソースコード管理からは外すため、`.gitignore`に入れておきましょう。

ビルド設定

開発したいパッケージがNode.js環境に依存したものであれば、Node.jsの型定義ファイルをインストールして入れておきます。 これでコンパイルのエラーがなくなり、コーディング時にコード補完が行われるようになります。

```
$ npm install --save-dev @types/node
```

共通設定のところで`tsconfig.json`を生成したと思いますが、これをライブラリ用に設定しましょう。 大事、というのは今回の要件の使う側が簡単なように、というのを達成するための.d.tsファイル生成と、出力形式のES5のところと、入力ファイルですね。 ユーザー環境でデバッグしたときにきちんと表示されるようにsource-mapもついでに設定しました。あとはお好みで設定してください。

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",           // 大事
    "declaration": true,       // 大事
    "declarationMap": true,
    "sourceMap": true,         // 大事
    "lib": ["dom", "ES2018"],
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  },
  "include": ["src/**/*"]      // 大事
}
```

ここでは `lib` に `ES2018` を指定しています。 ES5で出力する場合、 `Map` などの新しいクラスや、 `Array.entries()` などのメソッドを使うと、実行時に本当に古いブラウザで起動するとエラーになることがあります。 Polyfillをライブラリレベルで設定しても良いのですが、最終的にこれを使うアプリケーションで設定するライブラリと重複してコード量が増えてしまうかもしれないので、 `README` に書いておくでも良いかもしれません。

package.jsonで手を加えるべきは次のところぐらいですね。

- つくったライブラリを読み込むときのエントリーポイントをmainで設定

  src/index.tsというコードがあって、それがエントリーポイントになるというのを想定しています。

- scriptsにbuildでコンパイルする設定を追加

  今回はCommonJS形式とES2015の両方の出力が必要なため、一度のビルドで両方の形式に出力するようにします。

package.json

```
{
  "main": "dist-cjs/index.js",
  "module": "dist-esm/index.js",
  "types": "dist-cjs/index.d.ts",
  "scripts": {
    "build": "npm-run-all -s build:cjs buid:esm",
    "build:cjs": "tsc --project . --module commonjs --outDir ./dist-cjs",
    "build:esm": "tsc --project . --module es2015 --outDir ./dist-esm"
  }
}
```

課題

// browser/moduleなど // https://qiita.com/shinout/items/4c9854b00977883e0668

ライブラリコード

`import ... from "ライブラリ名"` のようにアプリケーションや他のライブラリから使われる場合、最初に読み込まれるエントリーポイントは `package.json` で指定していました。

拡張子の前のファイル名が、TypeScriptのファイルのファイル名となる部分です。前述の例では、 `index.js` や `index.d.ts` が `main` 、 `module` , `types` で設定されていましたので、 `index.ts` というファイルでファイルを作成します。 `main.ts` に書きたい場合は、 `package.json` の記述を修正します。

src/index.ts

```
export function hello() {
    console.log("Hello from TypeScript Library");
}
```

ここで `export` したものが、ライブラリユーザーが触れられるものになります。

まとめ

アルゴリズムなどのロジックのライブラリの場合、webpackなどのバンドラーを使わずに、TypeScriptだけを使えば良いことがわかりました。 ここにある設定で、次のようなことが達成できました。

- TypeScriptでライブラリのコードを記述する
- 使う人は普段通りrequire/importすれば、特別なツールやライブラリの設定をしなくても適切なファイルがロードされる。
- 使う人は、別途型定義ファイルを自作したり、別パッケージをインストールしなくても、普段通りrequire/importするだけでTypeScriptの処理系やVisual Stuido Codeが型情報を認識する
- Tree Shakingの恩恵も受けられる

`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。

```
# 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
```