モジュール

以前のJavaScriptは、公式には複数のファイルに分割してコーディングする方法を提供していませんでした。 当初はClosure Compilerや、その他のYahoo! UIやjQueryなどのライブラリごとの固有のファイル結合ツール、require.jsなどを使っていました。 その後Node.jsが登場して人気になると、CommonJSというサーバーサイドJavaScriptのための仕様から取り込まれたモジュールシステムがデファクトスタンダードとなりました。 これはブラウザからは利用しにくい仕様だったため、ブラウザからも利用できるES2015 modulesが仕様化されました。 今後の開発ではES2015 modulesの理解が不可欠になるでしょう。

本章では、ES2015 modulesを中心に、CommonJSとの互換性などを取り上げます。 アプリケーション開発では、まず基本文法の節だけ理解できていれば十分でしょう。中級、上級のネタはライブラリ作成、環境構築を行う人向けです。

用語の整理

モジュールを説明するまえに、パッケージも含めて、用語の整理をしておきましょう。なお、コンピュータの世界では、同じ用語だけど、人とかコンテキストによって全然違う意味を持つので、雑に語るのが危険なワードがあります。例えば、コンポーネント、モジュール、パッケージなどがそれにあたります。言語やツールによっても違うし、言語の一部のフレームワークによっても違うし、それらを束ねる抽象的な概念としても扱われます。TypeScriptでコードを書くときに他の人が「パッケージ」「モジュール」といったときにどれを指すのか、整理しておきます。

パッケージ

ここで扱うパッケージは、Node.jsを核とする、JavaScriptやTypeScript共栄圏の言葉の定義です。パッケージは、Node.jsが配布するソフトウェアの塊の最小単位です。 `npm` （Nodeパッケージマネージャ）や、 `yarn` といったツールを使って、npmjs.orgなどのリポジトリからダウンロード、社内のファイルサーバーで.tgzの配布物として提供されるものです。Gitリポジトリをそのままパッケージとすることもできます（ダウンロードや更新チェックが遅くなるデメリットがあるのであまり使わない方が良いでしょう）。元はNode.js用として始まりましたが、ブラウザ向けのフレームワークなども今はほとんど多くがnpmjs.orgで配布されています。

パッケージのソースは、 `package.json` というパッケージ定義のファイルが含まれているフォルダです。このファイルには、プロジェクトの名前やバージョン、説明、著者名などのメタデータ以外に、開発時に使うコマンド、依存ライブラリなどの情報が含まれます。github.comでJavaScriptとかTypeScriptのプロジェクトを見ると、トップページ、あるいは `projects` や `packages` という名前のフォルダの下のサブフォルダ内に `package.json` があることがわかるでしょう。このフォルダの中で、 `npm pack` とやれば `.tgz` ファイルができますし、 `npm publish` とタイプすると、npmjs.orgで全世界に向けて公開できます。

パッケージのダウンロードはNode.jsと一緒に配布される `npm` コマンドを使って行います。以下のコマンドでは、Vue.jsのプロジェクトを作成するCLIコマンドと、Vue.jsのライブラリの2つをダウンロードしています。

```
$ npm install @vue/cli vue
```

npmjs.orgで配布しない、自作のアプリケーションやライブラリ、サービスなんかもパッケージとして作成します。Node.js用ではない、ウェブのフロントエンドのコードでも、Node.jsのパッケージ形式でプロジェクトを作成します。公開しない場合でも、パッケージにしておけば、開発用コマンドのランチャーとして使ったり、依存パッケージの管理ができます。 `package.json` がある場合、インストールすると、 `package.json` に `dependencies` のところに情報が保存されます。 `--save-dev` （もしくは `-D` ）をつけると、 `devDependencies` に保存されます。本番環境ではいらない、開発用のツールなどはこのオプションをつけます。

devDependenciesに登録

```
npm install --save-dev [パッケージ名]
```

新しいコンピュータや他人のコンピュータ上で作業フォルダを作る必要があるときは、その以前インストールしたファイル一覧を元に `npm install` コマンドだけですべての必要なパッケージのダウンロードが完了します。 `dependencies` と `devDependencies` のパッケージがダウンロードされます[1]。そのパッケージが他のパッケージに依存していたら、それらもすべてダウンロードしてインストールされます。 `npm install --prod` だと、 `dependencies` のみがダウンロードされます。初回にダウンロードされて `package.json` に登録されるときは、サブのパッケージも含めて `package-lock.json` というファイルに全バージョン情報が保存されます。 `npm ci` コマンドを使って、これに記録されたバージョンと厳密に一致したバージョンのみをダウンロードすることもできます。

1 ただし環境変数 `NODE_ENV` が `production` に設定されていると、 `--prod` オプションを付けた場合と同じ動作になります。

TypeScriptとパッケージ

TypeScript固有トピックとしては、Aというパッケージ自体にTypeScript固有の型定義ファイルが含まれないことがあり、@types/Aという別のパッケージとして提供されていることがあります。TypeScriptから見ると、この2つで1つのパッケージという感覚でいれば良いでしょう。

インストールされているパッケージの型定義ファイルがあればダウンロードするtypesyncコマンドがnpmjs.orgにあります。明示的に@typesのパッケージをインストールしても良いのですが、typesyncコマンドをインストールして、installコマンド実行時に毎回実行するようにすると便利です。

*typesync*コマンドのインストール

```
$ npm install typesync
```

型定義のパッケージが別にあれば自動ダウンロード

```
{
  "scripts": {
    "postinstall": "typesync"
  }
}
```

これで、`npm install`のたびに、型定義ファイルも（あれば）ダウンロードされるようになります。

モジュール

JavaScriptやTypeScript界隈でモジュールというと、ECMAScript2015で入ったモジュールの機能、およびその文法に準拠しているTypeScript/JavaScriptの1つのソースファイルのことを指します。もっとも、これらの界隈でも、Angularはまたそれ固有のモジュール機構などを持っていたりしますが、それはここでは置いておきます。

簡単にいえば、1つの.ts/jsファイルがモジュールです。モジュール機能を使うと、ファイルを分割して、管理しやすいサイズのソースファイルに区切ってプロジェクトの開発をすすめることができます。モジュールは、外部に提供したい要素を`export`したり、外部のファイルの要素を`import`することができます。同一のフォルダ内の別のファイルを参照する場合にも、`import`が必要です。

パッケージの方がモジュールよりも大きな概念ですが、パッケージとモジュールの言葉が同じような文脈で利用されることがあります。パッケージ内部にたくさんのモジュール（ソースコード）が入ります。パッケージの設計時に、1つの代表となるモジュールに公開要素を集めることができます。パッケージの設定ファイルの中で、デフォルトで参照するモジュールが設定できます（`main`属性）。この場合、他のモジュールを`import`するのと同じように、パッケージの`import`ができるようになります。大抵のnpmjs.orgで公開されているパッケージは、このようにデフォルトで読み込まれるソースファイルにすべての要素を集める（ビルドツールで複数ファイルをまとめて結果として1ファイルになる場合も含む）のが一般的です。

モジュールの理解のやっかいなところは、裏の仕組みがいろいろある点です。モジュール機能はもともとブラウザのための機能としてデザインされましたので、ブラウザでは利用できます。Node.jsはオプションをつけると利用できます（ただし、拡張子は.mjs）。それ以外に、webpackなどのバンドラーと呼ばれるツールが、import/export文を解析して、1つの.jsファイルを生成したりします。Node.jsが旧来よりサポートしていたCommonJS形式のモジュールに、TypeScriptの型定義ファイルを組み合わせて `import` ができるようにしていることもあります。

本ドキュメントではTypeScriptを使いますので、基本的には次の形式のものがモジュールとなります

- TypeScriptの1ファイル
- TypeScript用の型定義ファイル付きのnpmパッケージ
- TypeScript用の型定義ファイルなしのnpmパッケージ+TypeScript用の型定義ファイルパッケージ

基本文法

エクスポート

ファイルの中の変数、関数、クラスをエクスポートすると、他のファイルからそれらが利用できるようになります。 エクスポートを行うには `export` キーワードをそれぞれの要素の前に付与します。

エクスポート

```
// 変数、関数、クラスのエクスポート
export const favorite = "小籠包";
export function fortune() {
  const i = Math.floor(Math.random() * 2);
  return ["小吉", "大凶"][i];
}
export class SmallAnimal {
}
```

インポート

エクスポートしたものは `import` を使って取り込みます。 エクスポートされた名前をそのまま使いますが、シンボル名が衝突しそうな場合は `as` を使って別名をつけることができます。 配布用のJavaScriptを作るバンドルツールは、この `import` 文を分析して、不要なコードを最終成果物から落としてファイルサイズを小さくするツリーシェイキングという機能を持っています。

インポート

```
// 名前を指定してimport
import { favorite, fortune, SmallAnimal } from "./smallanimal";

// リネーム
import { favorite as favoriteFood } from "./smallanimal";
```

`default` エクスポートとインポート

他の言語であまりない要素が `default` 指定です。エクスポートする要素の1つを `default` の要素として設定できます。

default

```
// defaultをつけて好きな要素をexport
export default address = "小岩井";

// defaultつきの要素を利用する時は好きな変数名を設定してimport
// ここではlocationという名前でaddressを利用する
import location from "./smallanimal";
```

`default` のエクスポートと、 `default` 以外のエクスポートは両立できます。

default

```
// defaultつきと、それ以外を同時にimport
import location, { SmallAnimal }  from "./smallanimal";
```

パスの書き方 - 相対パスと絶対パス

たいていのプログラミング言語でも同等ですが、パス名には、相対パスと絶対パスの2種類があります。

- 相対パス: ピリオドからはじまる。 `import` 文が書かれたファイルのフォルダを起点にしてファイルを探す
- 絶対パス: ピリオド以外から始まる。TypeScriptなどの処理系が持っているベースのパス、探索アルゴリズムを使って探す

絶対パスの場合、TypeScriptは2箇所を探索します。 ひとつがtsconfig.json の `compilerOptions.baseDir` です。 プロジェクトのフォルダのトップを設定しておけば、絶対パスで記述できます。 プロジェクトのファイルは相対パスでも指定できるので、どちらを使うかは好みの問題ですが、Visual Studio Codeは絶対パスで補完を行うようです。

```
import { ProfileComponent } from "src/app/component/profile.component";
```

もう一箇所は、 `node_modules` 以下です。npmコマンドなどでダウンロードしたパッケージを探索します。 親フォルダを辿っていき、その中に `node_modules` というフォルダがあればその中を探します。なければさらに親のフォルダを探し、その `node_modules` を探索します。

注釈

絶対パスの探索アルゴリズム（ `compilerOptions.moduleResolution` ）は2種類あり、 `"node"` を指定したときの挙動です。 こちらがデフォルトです。 `"classic"` の方は使わないと思うので割愛します。

TypeScript向けの型情報ファイルも一緒に読み込まれます。パッケージの中に含まれている場合は何もしなくても補完機能やコードチェックが利用できます。 そうでない場合は `@types/パッケージ名` というフォルダを探索します。 これは、パッケージとは別に提供されている型情報のみのパッケージです。

注釈

型情報ファイルの置き場は `compilerOptions.typeRoots` オプションで変更できます。既存のパッケージで型情報が提供されておらず、自分のプロジェクトの中で定義する場合に、置き場所を追加するときに使います。詳しくは型定義ファイルの作成の章で紹介します。

動的インポート

`import` / `export` は、コードの実行を開始するときにはすべて解決しており、すべての必要な情報へのアクセスが可能であるという前提で処理されます。 一方で、巨大なウェブサービスで、特定のページでのみ必要とされるスクリプトをあとから読み込ませるようにして、初期ロード時間を減らしたい、ということがあります。 この時に使うのは動的インポートです。

これはPromiseを返す `import()` 関数となっています。 このPromiseはファイルアクセスやネットワークアクセスをしてファイルを読み込み、ロードが完了すると解決します。 なお、この機能は出力ターゲットがES2018以降のみの機能となります。

```
const zipUtil = await import('./utils/create-zip-file');
```

課題

要検証

誰が `import` を行うのか？

JavaScriptにインポート構文が定義され、ブラウザにも実装は進んでいますが、この機能を使うことはいまのところあまりないです。 ブラウザ向けのTypeScriptのコード開発では、コンパイル時にこの `import`、 `export` をそのまま出力します。 TypeScriptも、この `import` と `export` を解釈して、型情報に誤りがないかは検証しますが、出力時には影響はありません。 それを1つのファイルにまとめるのは、バンドラーと呼ばれるツールが行います。 むしろ、バンドラーからソースコードを変換するフィルターとしてTypeScriptのコンパイラが呼ばれる、といった方が動作としては正確です。 ファイルにまとめるときは、不要な要素を削除するといった処理が行われます。

Node.js向けに出力する場合は、 `import` と `export` を、CommonJSの流儀に変換します。 こうすることで、Node.jsが実行時に `require()` を使って依存関係を解決します。

読み込みが遅く、実行も遅いとなるとそれだけで敬遠されるので、ブラウザの `import` と `export` が将来的には使われるようになるためには、不要なコードを削除する処理などを行って、効率の良いコードへの変換を行うツールが必要とされるでしょう。 しかし、そのようなツールが作られるとして、バンドラーと9割がた同じ処理をして、最後の出力だけは元のばらばらな状態で出力しなおす変換ツールになると思われます。 それであればバンドラーをそのまま使った方が何かと効率的だと思われますので、実際に作られることになるかどうかはわかりません。

中級向けの機能

リネームして `export`

`as` を使って別名でエクスポートも可能です。 たとえば、クラスをそのままエクスポートするのではなく、Reduxのストアと接続したカスタム版をオリジナルの名前でエクスポートしたいときに使います。

リネームをしてエクスポート

```
function MyReactComponent(props: {name: string, dispatch: (act: any) => void}) => {
  return <h1>私は小動物の{props.name}です</h1>
}

// リネームしてエクスポート
export { favorite as favariteFood };
```

複数のファイル内容をまとめてエクスポート

TypeScriptで大規模なライブラリを作成する場合、1ファイルですべて実装することはないでしょう。 アプリケーションから読み込まれるエントリーポイントとなるスクリプトを1つ書き、外部に公開したい要素をそこから再エクスポートすることにより、他の各ファイルに書かれた要素を集約することができます。

記述方法は、`import` 文の先頭のキーワードを `export` に変えるだけです。 他のファイルでデフォルトでなかった要素を、デフォルトとしてエクスポートすることも可能です。

再エクスポート

```
export { favorite, fortune, SmallAnimal } from "./smallanimal";

// リネームもできる
export { favorite as favoriteFood } from "./smallanimal";

// あとからdefaultにすることもできる
export { favorite as default } from "./smallanimal";
```

自動でライブラリを読み込ませる設定

TypeScriptでは、インポートの行を書かなくても、すべてのファイルですでにインポート行が書かれているとみなして読み込ませる機能があります。 JavaScriptの処理系はどれも、標準のECMAScriptの機能だけが提供されているのではありません。 JavaScriptは他のアプリケーション上で動くマクロ言語として使われることが多いので、環境用のクラスやら関数が提供されることがほとんどです。 `compilerOptions.types` を使うとその環境を再現することができます。

といっても、不用意に乱用するのはよくありません。 依存しているのに、依存が見えないということになりがちです。 たいてい必要なのは、Node.js用のライブラリ、特定のテスティングフレームワークの対応ぐらいでしょう。

```
{
  "compilerOptions": {
    "types" : ["node", "jest"]
  }
}
```

なお、ECMAScriptのバージョンアップで増える機能や、ブラウザのための機能は、これとは別に `compilerOptions.lib` で設定します。 こちらについては環境構築のところで紹介します。

ちょっと上級の話題

パス名の読み替え

ひとつのリポジトリに1つのパッケージだけを置いて開発するのではなく、関連するライブラリもすべて一緒のリポジトリに置いてしまう、というモノリポジトリという管理方法があります。 この名前を提唱して、積極的に使い出したのはBabelで、コア機能と、それをサポートする大量のプラグインが1つのリポジトリに収まっています。 この考え方自体は昔からあり、Javaの世界ではマルチプロジェクトと呼んでいました。

モノリポジトリのメリットは、依存ライブラリをpublishしなくても使えるため、依存ライブラリと一緒に機能修正する場合に、同時に編集できます。 コア側をpublishして、それにあわせて依存している方を直して、やっぱりだめだったのでコアを再publish・・・みたいなことはやりたくないでしょう。 関連パッケージ間のバージョンをきちんとそろえて、歩調を合わせたいというときには便利です。

JavaScript界隈のモノリポジトリでは、packagesやprojectsといったフォルダを作り、その中にプロジェクトフォルダを並べます。 pathsを使ったパスの読み替えを設定すると、各パッケージでは絶対パスで関連パッケージがインポートできます。

この場合によく使われるのが、ルートに共通設定を書いたファイルを作り、各パッケージではこれを継承しつつ、差分だけを記述する方法です。

tsconfig.base.json

```
"compilerOptions": {
  "baseUrl": "./packages",
  "paths": {
    "mylibroot": ["mylibroot/dist/index.d.ts"]
  }
}
```

packages/app/tsconfig.json

```
{
  "extends": "../../tsconfig.base",
  "compilerOptions": {
    "outDir": "dist"
  },
  "include": ["./src/**/*"]
}
```

なお、テスティングフレームワークのJestの場合は、TypeScriptの設定と別途名前のマッピングルールの設定が必要です。次のように書けば大丈夫なはずが、テストコードの場合は相対パスで使ってしまっても問題ないでしょう。

課題

ちょっとうまく動いていないので、要調査

jest.config.js

```
module.exports = {
  moduleNameMapper: {
    "mylibroot": "<rootDir>/../mylibroot/src/index.ts"
  }
}
```

CommonJSとの違い

ES2015 modulesが仕様化されたとはいえ、残念ながら現在の開発ではこれだけで完結はしません。 通常はダウンロードをまとめて行うために事前にバンドラーツールで1ファイルにまとめつつ最適化を行います。 ライブラリの流通の仕組みがNode.jsのエコシステムであるnpmjs.orgで行われることもあって、ライブラリの多くがCommonJS形式で提供されているため、CommonJSとも連携が必要です。

ES2015 modulesを利用して開発されたライブラリも、トランスパイラなどを通じてCommonJS形式にビルドされてパッケージ化されることがほとんどですが、これを利用する場合は特別な配慮をしなくてもimportできます。 それ以外のCommonJS形式で手書きで書かれたコードの読み込みではいくつか考慮点があります。

1つだけエクスポートした場合は、`default`でそのオブジェクトがエクスポートされたのと同じ動作になります。 オブジェクトを使って複数エクスポートする場合は明示的なインポートをすると問題ありません。 `default`形式と同様の動作をサポートするには、オブジェクトに`default`という名前の項目を追加し、なおかつ`__esModule: true`属性を付与すれば行えます。

これらの動作はBabelとTypeScriptのデフォルト設定で確認しましたが、これらの挙動はオプションでも変更される場合があります。また、RollupやParcelなどの別のバンドラーツールではまた動作が変わることがあります。

CommonJSのライブラリをES2015 modulesでインポート

```
// 1つだけCommonJS形式でエクスポート
module.exports = "小豆島";

// place==="小豆島";
import place from "./cjs-lib";

// オブジェクト形式でエクスポート(1)
module.exports = {
  place: "小豆島"
};

// place==="小豆島";
import { place } from "./cjs-lib";

// オブジェクト形式でエクスポート(2)
module.exports = {
  place: "小豆島",
  default: "小笠原",
  __esModule: true
};

// place==="小笠原";
import place from "./cjs-lib";
```

型のみのimport/export

TypeScript 3.8から、型のみをインポートしたりエクスポートできるようになりました。

型のみのインポート

```
import type { AwesomeType } from "./type";
```

読み込まれるファイルの中に何か副作用のある式があったとします。次のファイルはグローバルなところで`console.log()`があります。このファイルがインポートされるだけで、今日の運勢が出力されるという迷惑なライブラリです[2]。

fortunes.ts

```
export const fortunes = ["大吉", "吉", "中吉", "小吉", "末吉", "凶", "大凶" ];
console.log(`あなたの今日の運勢は${fortunes[Math.floor(Math.random() * 7)]}`);

export type Fortunes = "大吉" | "吉" | "中吉" | "小吉" | "末吉" | "凶" | "大凶";
```

TypeScriptはインポートしたものが型だけの場合に、出力からはそのインポート文を丸ごと排除します。排除されると、nccやBabelなどのバンドラーの出力結果に、そのインポート先のファイルが含まれなくなるため、副作用はおきません。副作用を起こしたい場合は型ではなく値を読み込むか、読み込み対象を指定せずにインポートします。

```
// 値を読み込むと副作用発生
import { fortunes } from "./fortunes";

// 型だけなら発生せず
import { Fortunes } from "./fortunes";

// 対象を絞らなくても副作用発生
import "./fortunes";
```

そもそも副作用があるモジュールはあまりないとは思いますが、この副作用が発生しないことを明示的に指定するのが型のみのインポートです。

この挙動を制御するオプションが3.8から増えました。`tsc --init`しても出力されない、レアなオプションです。

`compilerOptions.importsNotUsedAsValues: "remove" | "preserve" | "error"`

- "remove": 削除する（現行のデフォルトとおなじ）
- "preserve": 型だけであってもインポートを残し、副作用が必ず発生するようになる
- "error": 型だけを通常の名前束縛のインポート構文で読み込むとエラーにする

この最後の`preserve`や`error`の時に、副作用なく型のインポートのみを許容する構文があります。それが次の書き方です。

```
// 型だけなら発生せず
import type { Fortunes } from "./fortunes";

// compilerOptions.importsNotUsedAsValues: "error"だとエラーに
import { Fortunes } from "./fortunes";
```

なお、この構文は読み込めるのは型だけなので、コロンの左側に来る要素で使うとエラーになります。

型だけインポートを使うと今までも現在もインポート自体がなかったことにされますが、このオプションにより副作用の有無が明示的なコードを書くことができるようになります。strictを限界まで設定しているユーザーは`compilerOptions.importsNotUsedAsValues: "error"`も追加すると良いでしょう。

インポートだけではなく、`export type { A, B } from "./modules";`といった、インポートして即エクスポートする文においては、`export`にも利用できます。

現在リリースされているtypescript-eslintの4.0以降においても、`import type`を使うことを強制するルールが追加されました[3]。TypeScript 3.7以前ではすべてがエラーになってしまうため、デフォルトでは有効化されていません。

eslintrc.json

```
{
  "rules": {
    "@typescript-eslint/consistent-type-imports": "error"
  }
}
```

2 他に迷惑な有名なライブラリとしては、Pythonのthisがあります。`import this`をするとPythonの設計思想を表す詩が表示されます。

3 https://github.com/typescript-eslint/typescript-eslint/blob/v4.1.1/packages/eslint-plugin/docs/rules/consistent-type-imports.md

まとめ

インポートとエクスポートのための構文自体は難しくありません。 ファイル名を間違ったりしても、Visual Studio Codeなどのエディタがすばやくエラーを見つけてくれるため、問題の発見と解決は素早く行えるでしょう。

JavaScriptには当初モジュール機構がなく、後から追加されたりしたため、過去の経緯、CommonJSなどの他の仕組みも考慮したうえで設定を行う必要があったりします。 しかし、最終的にはES2015形式のモジュール記法に統一されていくため、基本的にはこちらですべて記述していけば良いでしょう。

やっかいなのはモノリポジトリなどの複雑な環境です。 こちらは環境構築を行うメンバーが気合を入れて取り組む必要があるでしょう。