TypeScript による Web シンタックスハイライト

松田 悠斗 (MATSUDA, Yuto)

2024 年 12 月 8 日

1 概要

本プロジェクトでは，TypeScript で Web ページ用のシンタックスハイライト (コードの色付け) を作成する．ブログにおけるソースコードの自動スタイリングを目的とし，プログラムを作成した．本プロジェクトは React や Next.js への移行を想定したものであり，これを利用すれば効率的な自作ハイライティングを実装できる．

また，プログラムの実装だけでなく，Node.js やその他ツールについての理解を深めることも本プロジェクトの目的である．

本プロジェクトにおける実行環境を以下に示す．

- Ubuntu 20.04.4 LTS
- TypeScript 5.7.2
- Node.js 20.11.1
- npm 10.4.0
- webpack 5.97.0

また，Node.js バージョン管理ツール Volta の利用も検討したが，今回は利用を見送った．

2 実行環境及びツールの概要

2.1 Node.js とは

Node.js とは，JavaScript の実行環境である (Python をインストールすると Python を開発・実行できることと同じ)．従来 Web ブラウザ上でしか実行できなかった JS を，PC やサーバ上でも実行するために開発された．

サーバサイドの JS 実行環境と呼ばれることが多いが，あくまでも「JS の実行環境」なので，クライアントサイドの開発にも利用できる．(HTML にスクリプトを読み込んでブラウザで実行せずとも，クライアントサイドの開発が可能になる．)

Node.js の利点の 1 つとしては，オープンソースのパッケージを `npm` でインストールして効率的な開発が可能になる点である (Python における `pip` と同じ)．これにより，`<script>`タグでライブラリのロードを行わずとも，コマンド 1 つで利用できるようになる．

Node.js のインストール方法は以下の通りである．

```
$ sudo apt install nodejs
```

利用の目的としては，主に以下のものが挙げられる．

(1) **新仕様の** JS/TS **での開発**

JS や TS の最新仕様で開発した際，ブラウザがその仕様に対応していないことがある．この問題を解決するため，旧仕様へのコンパイル (トランスコンパイル) を行う必要があるのだが，主要なトランスコンパイラである `Babel` の実行環境として Node.js を用いることが多い．

(2) Web **アプリケーションの作成**

Node.js は Web サーバの役割も果たすことができるため，Rails のように Web アプリを開発することができる．この場合，実行環境は Node.js，言語は JS/TS，フレームワークは Express や Next.js がよく用いられる．

その他にも，webpack や vite 等のバンドラを利用する際や，Sass のコンパイルのために用いることもある．

2.2 npm とは

npm とは，Node.js におけるパッケージ管理ツールである．コマンド 1 つでパッケージをインストールしたり，バージョンの管理を行うことができる．

2.2.1 主要なコマンド

(1) **プロジェクトの開始** (`package.json` **の生成**)

```
$ npm init
```

`-y` オプションを付けると，プロジェクトの初期設定情報を記した `package.json` が作成される．

(2) **パッケージのインストール**

```
$ npm install [package name]
```

`-g` オプションを付けるとグローバルにインストールでき，どのディレクトリからもパッケージを利用できる．

また，`-D` or `--save-dev` オプションを付けると，開発環境でのローカルインストールになる．この場合，`package.json` の `dependencies` ではなく `devDependencies` に追記される．例えば Git からクローンした後に必要なパッケージをインストールする場合，`npm i --production` とすると，そのパッケージ群はインストールされない．そのため，開発環境依存のパッケージ (webpack 等) は`-D` オプションを付けるべきである．

(3) **パッケージのアンインストール**

```
$ npm uninstall [package name]
```

グローバルにインストールしたパッケージの削除には`-g` オプションが必要になる．

(4) npm **及びパッケージのアップデート**

```
$ npm install -g npm@latest  # Update npm to latest version.
$ npm update [package name]
```

グローバルにインストールしたパッケージの更新には-g オプションが必要になる．

(5) **パッケージの一覧表示**

```
$ npm list
```

グローバルにインストールしたパッケージの表示には-g オプションが必要になる．

2.2.2 package.json

Node.js では，package.json を用いてプロジェクトにインストールされているすべてのパッケージを効率的に管理している．

例えばインストールしたパッケージ及びその依存パッケージは node_modules 以下に格納されるが，Git にプッシュする際はこのディレクトリを除外することが多い．しかし，インストールしたパッケージや依存関係を記した package.json さえダウンロードすれば，同ディレクトリ上で npm install を実行することですべてのパッケージを一括インストールできる．

以下は本プロジェクトにおける package.json である．

```
 1: {
 2:   "name": "syntax_highlight",
 3:   "version": "1.0.0",
 4:   "description": "",
 5:   "main": "index.js",
 6:   "scripts": {
 7:     "dev": "webpack serve --mode development",
 8:     "build": "webpack build --mode production"
 9:   },
10:   "keywords": [],
11:   "author": "Yuto Matsuda",
12:   "license": "ISC",
13:   "devDependencies": {
14:     "css-loader": "^7.1.2",
15:     "html-webpack-plugin": "^5.6.3",
16:     "mini-css-extract-plugin": "^2.9.2",
17:     "sass": "^1.82.0",
18:     "sass-loader": "^16.0.4",
19:     "ts-loader": "^9.5.1",
20:     "typescript": "^5.7.2",
21:     "webpack": "^5.97.0",
22:     "webpack-cli": "^5.1.4",
23:     "webpack-dev-server": "^5.1.0"
24:   }
25: }
```

package.json は，そのプロジェクトを npm パッケージとして公開するという目線で読むと分かりやすい．例えば name はパッケージ名となる．

インストールしたパッケージは dependencies に記述されている．なぜ「依存関係」として記述するかというと，当プロジェクトをパッケージとして公開する場合に，一緒にインストールしてもらう必要があるものだからである．

script は npm-script と呼ばれ, スクリプト名とシェルスクリプトの組で定義する．npm [script name] としてコマンドを実行すると，定義したシェルスクリプトを実行できる．

2.3 webpack とは

webpack とは，複数の JS ファイルや CSS，画像等を 1 つの JS ファイルにまとめるモジュールバンドラである．モジュール分割による開発効率向上だけでなく，1 つのファイルにまとめることで HTTP リクエストの数を減らすことにも繋がる．また，JS ファイルの圧縮やローカルサーバの起動等，フロントエンドにおける開発環境が webpack 一つで整う点も特徴である．

インストール方法は以下の通りである．

```
$ npm i -D webpack webpack-cli webpack-dev-server
```

`webpack-cli` は webpack を CUI 操作するためのツールで，ver4.0 以降から必要なものである．`webpack-dev-server` はローカルサーバの起動や，ソースの変更を検知 (watch) しビルドの自動実行とリロードを行うツールである．

エントリポイントと呼ばれるメインの JS ファイル (TS や JSX ファイルも可) を基準に複数ファイルがバンドルされ，1 つの JS ファイルが作成される．それを HTML で読み込むことで，クライアントサイドでの動作を確認できる．

webpack の利用には `webpack.config.js` の利用が一般的であり，詳しくは第 3 章で解説する．

3 環境構築

本プロジェクトにおける環境構築の手順を示す [7].

3.1 プロジェクトの開始

まず，以下のコマンドでプロジェクトの初期化及び開発環境の構築に必要なパッケージのインストールを行う．

```
$ npm init -y
$ npm i -D webpack webpack-cli webpack-dev-server
$ npm i -D typescript ts-loader
$ npm i -D mini-css-extract-plugin css-loader sass sass-loader
$ npm i -D html-webpack-plugin
```

3.2 スクリプトの作成

次に，以下の npm-script を用意する．

```
"dev":   "webpack serve --mode development",
"build": "webpack build --mode production"
```

なお，`--mode` オプションの概要は以下の通りである．

- `development`: ソースマップが作成され，ビルド結果に付加される．
- `production`: ビルド結果を圧縮して生成する．

3.3 webpack の設定

次に，webpack.config.js を以下のように作成する．

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const HtmlWebpackPlugin    = require("html-webpack-plugin");

module.exports = {
    // entry point
    entry: {
        main: `${__dirname}/src/ts/main.ts`  // __dirname: current directory
    },

    // configuration of output files.
    output: {
        path: `${__dirname}/dst`,
        filename: '[name].js'  // In this time, [name] = main.
    },

    resolve: {
        extensions:['.ts','.js'],  // TS and JS are treated as module.
        alias: {
            '@': `${__dirname}/src/`,
        },
    },

    devServer: {
        // Open 'dst/index.html' automatically.
        static: {
            directory: `${__dirname}/dst`,
        },
        open: true,
    },

    module: {
        rules: [
            {
                // Apply TS compiler to files ending with '.ts'.
                test:  /\.ts$/,
                loader:'ts-loader'
            },
            {
                test: /\.(scss|sass|css)$/,
                use:  [
                    MiniCssExtractPlugin.loader,
                    'css-loader',
                    'sass-loader',
                ]
            }
        ]
    },

    plugins: [
        new MiniCssExtractPlugin(),
        new HtmlWebpackPlugin({
            template:      `${__dirname}/src/index.html`,
            inject:        'body',  // insert <script> to just before <body>
            scriptLoading: 'defer'
        }),
    ]
}
```

設定のポイントをいくつかまとめる．

まず，resolve の extensions の指定により，JS ファイルと TS ファイルをモジュールとして扱うことを明示している．例えば，module1.js をインポートする際，import func from 'module1' のように，拡張子を省略することができる．これは，module1 という名前の JS ファイルもしくは TS ファイルをモジュールとして探すことを意味する．

また，alias はインポート時のパスの指定を簡単化するために設定している．これにより，絶対パスによるインポートを import foo from '@/ts/molude1' のように記述できる．

モジュールに対するルール設定においては，test に指定したファイルに対して loader もしくは use 配列にしていしたモジュールを適用するよう指定している．use 配列に指定したモジュールは配列の末尾から順に適用される点がポイントである．例えば今回なら，sass-loader で CSS へのコンパイルを，css-loader で JS へのバンドルを行い，mini-css-extract-plugin で style.css として出力する流れとなる．

html-webpack-plugin は HTML を webpack から出力するためのモジュールで，バンドルされた JS ファイルや CSS ファイルの読み込みを意識することなく HTML を記述することができる．

参考文献

[1] Node.js とはなにか？，https://qiita.com/non_cal/items/a8fee0b7ad96e67713eb，2024/12/5

[2] nodejs とは，https://kinsta.com/jp/knowledgebase/what-is-node-js/，2024/12/5.

[3] npm とは，https://kinsta.com/jp/knowledgebase/what-is-npm/，2024/12/5.

[4] package.json の中身を理解する，
https://qiita.com/dondoko-susumu/items/cf252bd6494412ed7847，2024/12/5.

[5] 最新版で学ぶ webpack 5 入門，https://ics.media/entry/12140/，2024/12/5.

[6] npm install の –save-dev って何？，
https://qiita.com/kohecchi/items/092fcbc490a249a2d05c，2024/12/6.

[7] TypeScript チュートリアル -環境構築編-，
https://qiita.com/ochiochi/items/efdaa0ae7d8c972c8103，2024/12/6.

[8] webpack の苦手意識を無くす，https://zenn.dev/msy/articles/c1f00c55e88358，2024/12/8.

[9] ，，2024/12/.

[10] ，，2024/12/.