webpack

ここではバンドルツールについて学習します。

webpack の役割

webpack は指定された JS ファイルを起点として、そこから import 文を頼りに関連している JS ファイルを繋げてゆき、1 つにまとめた JavaScript ファイルを出力します。

このまとめる処理はバンドルと呼ばれています。そしてバンドルを行うツールはバンドラーと呼ばれています。

webpack の基本機能はあくまで 1 つの JavaScript ファイルにまとめるというところです。

イメージとしては A,B,C ファイルがそれぞれあったとします。それをすべて import している JS ファイルがあったとします。

それを import している JS ファイルを対象としてバンドルを行い 1 つの JS ファイルにまとめるイメージです。

なぜ webpack を利用するのか

* 機能ごとにファイルを分割（モジュール化）して開発ができるから

  1 つのサービスには複数の JS で実装された機能があります。

  それを 1 つのファイルに実装すると可読性が悪くなる、変数の競合（重複や機能に相応しくない変数名をつける）が発生など開発する上で非効率、弊害の発生があります。

  それを分割することによって解決し、リリース後にも機能修正しやすくなります。

  この機能ごとに分割することをモジュール化と言います。

* リクエスト数を減らせれる

  開発する上ではモジュール化させた方が良いですが処理する時は 1 つにまとめた方が良いのでバンドル化します。

サービスを運用する際、サーバー間では複数の情報のやりとりが発生しています。

上記のようにモジュール化させたものをそのまま運用するとそれぞれの import 文を読み込む必要があります。

そうすると処理効率が悪くなるのでサービスの利便性にも関わってきます。

webpack のインストールと設定

まずは空のフォルダーを作成し、`npm init`を実行して`package.json`を作成してください。

作成できたら下記のコマンドを実行してください。

`npm i -D webpack webpack-cli`

インストールが完了すると`package.json`に２つのパッケージがインストールされているかと思います。

```JS
"devDependencies": {
  "webpack": "^5.38.1",
  "webpack-cli": "^4.7.0"
}
```

それができたら下記のようなディレクトリ構造を作成してください。

```
ご自身で作成したフォルダー/
├ src/
|    ├ modules/
```

```
|    |    └ test.js
|    └ index.js
└ dist/
```

上記のようにディレクトリー構造をツリー表示させることもあるので覚えておいてください。

作成ができたら`test.js`に下記のコードを貼り付けてください。

```JS
function test() {
  console.log('Hello World!!');
}
// 関数を他のファイルでも使用できるようにエクスポートしている
export { test };
```

次に`index.js`に下記のコードを貼り付けてください。

```JS
import { test } from './modules/test'; //拡張子は省略可能

test();
```

ここまでできたら下記のコマンドを実行してください。(warning エラーが出た場合は無視で大丈夫です。)

`npx webpack`

実行後 dist フォルダーの中に`main.js`が作成されたかと思います。

1 つだけだとイメージが掴み難いと思うので次に`alert.js`と言うファイルを作成してください。

作成ができたら下記のコードを貼り付けてください。

```JS
function alert() {
```

```
  alert('Hello World!!');
}

export { alert };
```

それができたら`index.js`インポートさせ、関数の実行を行うようにしてください。

ここまでできたら再度`npx webpack`を実行してください。

バンドル化されたコードを見ても変化があまり分からないですがこれで 2 つのファイルの処理を 1 つにまとめています。

webpack を使用する

上記の利用方法も間違ってはいませんがあまり融通は効いていません。

`webpack.config.js`を用意することで細い設定を行なって使用できます。

早速`webpack.config.js`を作成して下記のコードをコピペしてください。

```JS
module.exports = {
entry: './src/index.js',
output: {
  path: __dirname + '/dist',
  filename: 'sample.js'
  }
};
```

* entry

バンドルを行う時の起点となるファイルを指定する項目です。（エントリーポイント）

ここを指定することで自分の環境に合わせることができます。

* output

バンドル後の設定を行う項目です。

* path

バンドルしたファイルをどこに出力するかを設定する項目です。

* __dirname

`__dirname`は Node.js であらかじめ用意されている変数です。これは Node.js が自走的に`__dirname`へ`webpack.config.js`の絶対パスを自動的に取得します。

（`__dirname`を使うディレクトリ階層で取得するパスは変わります）

　webpack の仕様上`entry`は相対パスですが、`output`のパスは絶対パスで設定する必要があります。

　複数人で開発する時にここを設定するのは手間がかかる、ディレクトリ階層が人によって違う、作業ミスにより開発環境が変わる可能性があるなどの懸念があります。

　なので`__dirname`を使用することでパス依存することなく利用できます。

* __dirname

　出力後のファイルネームを設定できます、

設定後`npx webpack`を実行しすると`webpack.config.js`の設定にしたがってバンドルを行います。

モード設定

webpack バンドルを行う際、開発用、本番用の 2 つのモードが設定できます。

* development

　開発モード、デバッグしやすい状態にバンドルする

* production

　本番モード、なるべくファイルサイズを小さくする

　`entry`の上に 1 行空けて

　`mode: 'development'`または`mode: 'production'`を追加して使用します。

　追加して実行し、コマンド実行して見てください。

出力するファイル内容にも差が出ています。

Watch モード

gulp にもあった Watch モードは webpack にもあります。

`module.exports = {}`内へ`watch: true,`を追加してください。

そして、`webpack --watch`を実行することで Watch モードになります。

ただこの状態だと`node_modules`など監視が不要なものまで見てしまうので watch の対象外とするファイルも設定する必要があります。

`watchOptions`内に定義して対象外のものを設定できます。

```JS
/* 略 */
watch: true,
watchOptions: {
  ignored: /node_modules/
},
/* 略 */
```

`watchOptions`については他にもいろいろ設定できるので公式ドキュメントを見てください。

[Watch and WatchOptions | webpack](https://webpack.js.org/configuration/watch/)

npm にコマンドを登録する

`npx webpack`でも実行するのも良いですが、モード設定があるように開発用のファイルと本番用のファイルを作って利用することもあります。

その都度ファイルを差し替えるのも面倒なのでnpmにコマンドを登録することで使い分けができます。

`package.json`の`scripts`へオブジェクト型で登録できます。

```JS
"scripts": {
  "start": "webpack -w --config webpack.dev.config",
  "build": "webpack --config webpack.pro.config"
},
```

実行時は`npm run start`、`npm run build`で実行でき、その際に右側のコマンドが実際に実行されています。

これはwebpackに限らず他のパッケージでも可能です、(gulpなども可能)

注意点

startに記述してある`-w`はWatchモードを指定しているのでこの場合は設定ファイル内に記述した`watch: true,`を削除または､コメントアウトしておいてください。

何にもしなくても動作はしますがwarningエラーが表示されます。

Loader

ローダーはCSSや画像ファイルなどJavaScript以外のファイルをJavaScriptで扱えるように変換したり、バンドルする前にモジュールに対して実行する機能です。

たとえばSass使用するのにgulpを用いる必要がありましたが、あるローダーを使用するとgulpが使わずともブラウザーで読み込まれるようにしてくれます。

babel-loader

ES6以降のコードをESに変換する

`npm install babel-loader @babel/core @babel/preset-env --save-dev`

eslint-loader

JavaScriptのコードを検証する

eslintの設定ファイルである「.eslintrc」も追加する。

```JS
{
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 6,
    "sourceType": "module"
  },
  "env": {
    "browser": true
  }
```

```
}
```

`npm install eslint eslint-loader --save-dev`

sass-loader

Sass をコンパイルするコンパイルする為のローダー

`npm install sass-loader sass webpack --save-dev`

ただこれだけだとコンパイルされた CSS を処理してくれないので

style-loader と css-loader もインストールしてください。

`npm install style-loader css-loader --save-dev `

Sass ファイルを作成し、src/index.js にインポートさせればあとは実行時に自動的に処理してくれます。

補足

最終的にでき上がったファルがこちらです。

```JS
module.exports = {
  mode: 'development',
  // mode: 'production',
  entry: './src/index.js',
  // -w コマンドを使用するのでコメントアウト
  // watch: true,
  watchOptions: {
    ignored: /node_modules/
  },
  output: {
```

```
    path: __dirname + '/dist',
    filename: 'sample.js'
  },
  // ローダーの設定
  module: {
    rules: [
      {
        // ローダーの処理対象ファイル
        test: /¥.js$/,
        // ローダーの処理対象から外すディレクトリ
        exclude: /node_modules/,
        use: [
          {
            // 利用するローダー
            loader: 'babel-loader',
            // ローダーのオプション
            // 今回は babel-loader を利用しているため
            // babel のオプションを指定しているという認識で問題ない
            options: {
              presets: [['@babel/preset-env', { modules: false }]]
            }
          },
        ]
      },
      {
        // enforce: 'pre' を指定することによって
        // enforce: 'pre' がついていないローダーより早く処理が実行される
        // 今回は babel-loader で変換する前にコードを検証したいため、指定が
必要
        enforce: 'pre',
        test: /¥.js$/,
        exclude: /node_modules/,
        loader: 'eslint-loader'
      },
      {
        test: /¥.s[ac]ss$/i,
        use: [
          "style-loader",
          "css-loader",
          {
```

```
            loader: "sass-loader",
            options: {
              // Prefer `dart-sass`
              implementation: require("sass"),
            },
          },
        ],
      },
    ]
  },
};
```

コピペして使用して良いので実際にどんな動きをするのか触って確認してください。

この設定だと dist/sample.js ができるのでそれを HTML にインポートして確認してください。

今回は今まで触ったローダーに触れましたがこれ以外にもあるので調べてみてください。

webpack で今まで紹介した機能が使えるのならこれで良いじゃんと思いますが、webpack はあくまでも JS ファイルのバンドルツールであってローダーはメインの機能ではありません。

案件によって作業環境は違うので他ツールの理解度も高めておいてください。