Gulp + webpack勉強会

in ギルコネ

2023.05.07

プロフィール

石本真一（いしもと まさかず）
@masa_kazu5
島根県松江市在住
エンジニア・ディレクター
Web制作歴5年目

■経歴
松江高専卒、卒業後地元のSE系企業に就職
その後、サービス業の会社を経て、
2018年からWeb制作会社に勤める
Web制作に関してはエンジニア歴約5年、
ディレクター歴約1年
現在、35歳。2児の父（5歳、1歳）

この勉強会のメリット

- Gulpを使うことで作業時間の短縮が望める
  - 時給単価のアップ
- 案件の幅が広がる
  - 都内などの製作会社はGulpを使っている確率が高い
- エラー等の対応など、自分で行える幅が広がる

はじめに

この企画のきっかけ

・アウトプットをすることで自分の理解を深めるため
・Gulpの動作環境をアップデートしたから
・Gulpを使える人は多くても、自分で作れる人が少ないと思ったから
・カスタマイズの楽しさを伝えたい

様々なパッケージ

プロジェクトごとに
まとめて管理・共有

Gulp

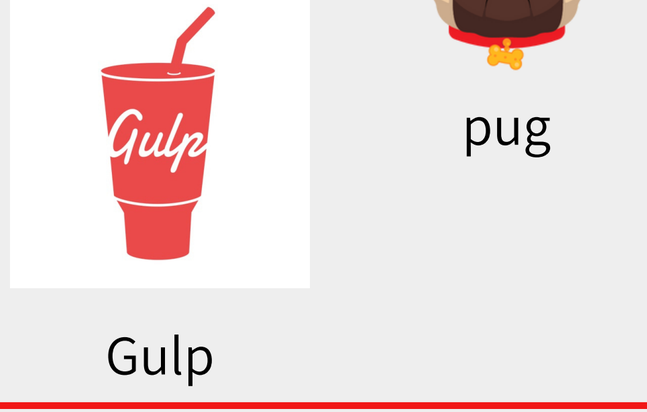
pug

webpack
等々

土台

様々なパッケージ

プロジェクトごとに
まとめて管理・共有



pug

webpack

Gulp

等々

土台

Gulpとは

Gulpは、JavaScriptで書かれたタスクランナー
（開発作業を自動化し効率化するツール）

- 画像の最適化、圧縮
- 画像のWebP化
- SassからCSSへのコンパイル
- CSSやJavaScriptの圧縮・最小化
- ファイルの監視と自動リロード
- ベンダープレフィックスの自動付与（Autoprefixer）
- EJSやPugなどのテンプレートエンジンからHTML生成

webpackとは

モジュールバンドラーとして人気のある
JavaScriptツール

- JavaScriptやアセットファイルをまとめる
- JavaScriptのトランスパイル
  （古いブラウザでも対応できるように変換）
- JavaScript・CSS・画像等の圧縮

主に複数のJavaScriptファイルを１つにまとめる事が多い

Gulp + webpack を利用

- 画像の最適化、圧縮
- 画像のWebP化
- SassからCSSへのコンパイル
- CSSやJavaScriptの圧縮・最小化
- ファイルの監視と自動リロード
- ベンダープレフィックスの自動付与（Autoprefixer）
- EJSやPugなどのテンプレートエンジンからHTML生成

＋

- JavaScriptモジュールを1つのファイルに纏める
- Babelを使用してJavaScriptを変換

様々なパッケージ

プロジェクトごとに
まとめて管理・共有

npm

Gulp

pug

webpack
等々

土台

node.js

・npmを使用するために事前にインストール
・バージョン管理ツールを利用すると便利

Node.jsとは

JavaScriptをサーバーサイドで使えるようにする
実行環境。

通常JavaScriptはブラウザ側の処理しかできなかった。

■環境構築

- 公式サイトからダウンロード
  https://nodejs.org/ja
- Nodebrewやnodenvなどバージョン管理ツール

Node.jsとは

バージョンの確認

```
% node -v
v18.12.1
```

バージョンの確認・切り替え（nodenvの場合）

```
% nodenv versions
16.16.0
18.12.1 (set by /Users/to/path/src/.node-version)

% nodenv global 16.16.0
% node -v
v18.12.1
```

様々なパッケージ

プロジェクトごとに
まとめて管理・共有

npm

・「npm i」でインストール
・packege.jsonに定義

Gulp

pug

webpack
等々

土台

node.js

・npmを使用するために事前にインストール
・バージョン管理ツールを利用すると便利

npmとは

npm（Node Package Manager）はNode.jsのライブラリやツールを管理してくれるシステム。

Gulpやwebpackを使う時、それぞれのライブラリをダウンロードする必要がある。
１つずつインストールのは手間。

package.json を使って管理・共有できるようにする。

npmとは

ライブラリのインストール

```
% npm i パッケージ名
// npm install パッケージ名 でもOK
% npm I gulp

// 開発環境のみ
% npm i パッケージ名 --save-dev
% npm i gulp --save-dev

→ package.json に追加 + node_modulesにダウンロード
```

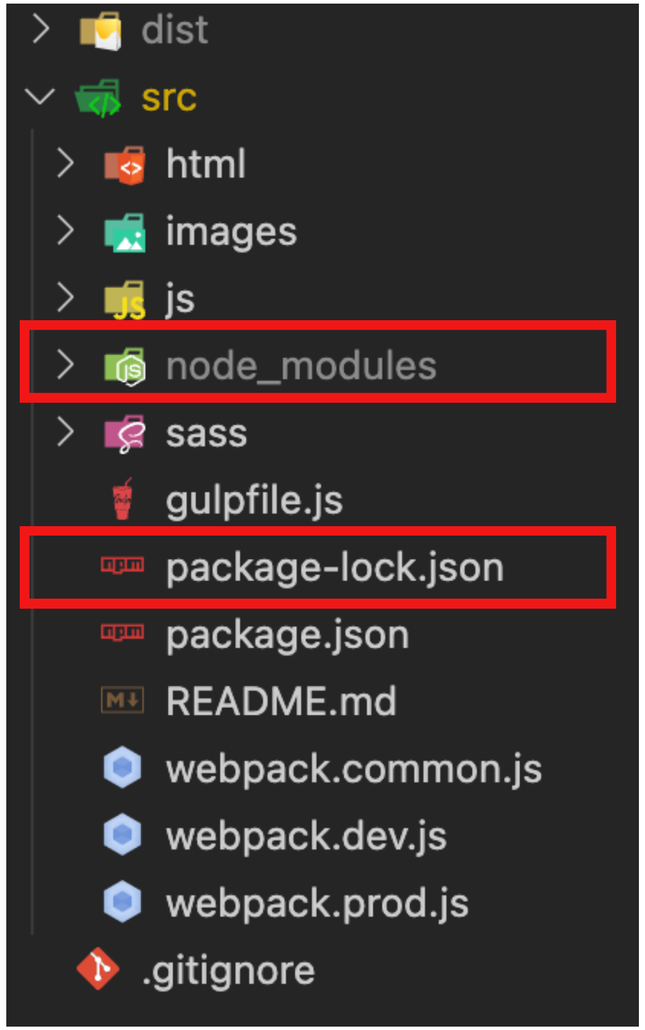
package.jsonとは

プロジェクトで使う様々なパッケージをjson形式で管理。
※パッケージは機能ごとのプログラムのまとまり

- name：ファイルの名前
- version：ファイルのバージョン
- devDependencies：開発時にのみ使うパッケージ（自動リロード、Sassのコンパイルなど）
- dependencies：本番環境で使うパッケージ情報（jQuery、Swiperなど）

package.jsonさえ共有されていれば、「npm i」をすることで同じ環境を構築することができる

package.jsonとは



■node_modules

npm i したときに実際にダウンロードされたファイルが格納される。

- 本番では使用しない
- gitでの管理から外す

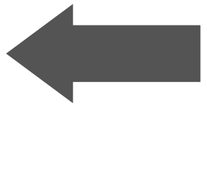
■package-lock.json

実際にダウンロードしたファイルの情報や依存関係が記されている。

- 触らない

ライブラリ

プロジェクトごとに
まとめて管理・共有



・packege.jsonに定義
・「npm i」でインストール

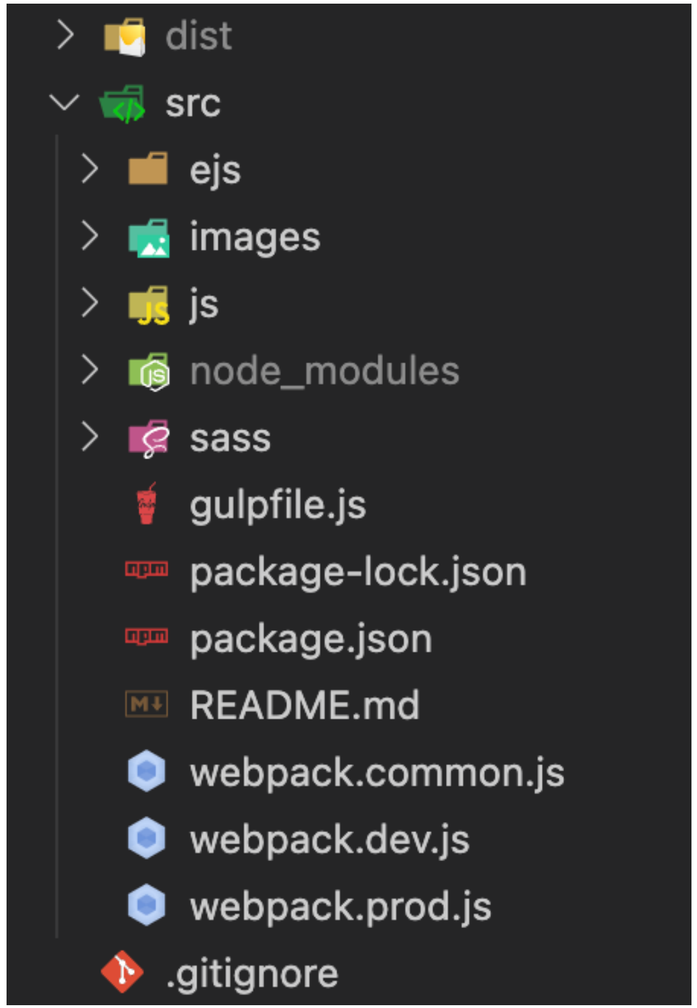
・機能ごとに様々なライブラリを提供
・node_moduledsに実態が格納

土台

・npmを使用するために事前にインストール
・バージョン管理ツールを利用すると便利

実際のGulpファイルについて

ファイル構成



■srcフォルダ
実際に編集するファイルを格納したフォルダ

■distフォルダ
Gulpを使って出力したファイルを格納

■gulpfile.js
Gulpの定義ファイル

■webpack.dev.js / webpack.prod.js
webpackの定義ファイル（開発用/本番用）

「npx gulp」 で開発モード
「npx gulp build」 で本番用コード出力

開発用と本番用の違い

一般的に、開発用と本番用の出力するファイルには違いがあります。

■ファイルの最小化
開発用は最小化せず、コードの読みやすさを重視する事が多いです。
本番用ではファイルの圧縮（改行やスペース等の削除）をしてファイルサイズを圧縮します。

■ソースマップの有無
ソースマップがあることで、SassやJavaScriptなど、分割前のファイルの場所を特定することができます。開発用ではソースマップを使用し、本番用ではソースマップを削除する恵子があります。

・開発用は開発時の効率を重視する
・本番用は読み込み速度やファイルサイズの最小化を重視する

Gulpのコード（全体感）

■プラグインの読み込み
const 変数名 = require("プラグイン名"); で定義する
■各種変数の定義
　読込先・出力先のディレクトリ等
■タスクの定義

- Sass→CSSへの変換
- 画像圧縮
- JavaScriptのバンドル
- ファイル監視
- ブラウザリロード など

■タスクの実行

- exports.タスク名 = 関数名 で定義
- 「exports.default」の内容が「npx gulp」コマンドを叩いた時に実行される
- package.json の scripts でも定義可能

コード説明(Gulp)

gulpプラグインのインポート

```
// gulpプラグインの読み込み
const { src, dest, lastRun, watch, series, parallel } = require("gulp");
```

src：入力先のパスの指定
dest：出力先のパスの指定
lastRun：変更があったファイルのみ実行
watch：ファイルの変更を監視する
series：順番に処理を実行
parallel：同時に処理を実行

コード説明(Gulp)

```
// 読み込み先のディレクトリ
const srcPath = {
  html: "./html/**/*.html",
  css: "./sass/**/*.scss",
  js: "./js/**/*",
  img: "./images/**/*",
};
const distPath = {
  // 出力先のディレクトリ
  all: "../dist",
  html: "../dist/",
  css: "../dist/assets/css/",
  js: "../dist/assets/js/",
  img: "../dist/assets/images/",
};
```

各種パスの定義

Sassの例

```
// Sassプラグインの読み込み
const sass = require("gulp-sass")(require("sass"));
// Sass タスク
const cssSass = () => {
  return (
    src(srcPath.css, { // ソースファイルを指定
      sourcemaps: true, // ソースマップを有効に
    })
      // エラーがある場合、通知を出す
      .pipe(plumber({ errorHandler: notify.onError("Error:<%= error.message %>") }))
      .pipe(
        sass.sync({
          includePaths: ["node_modules", "src/sass"], // インポートのパスを指定
          outputStyle: "expanded", // コンパイル後のスタイルを指定
        })
      )
      .pipe(dest(distPath.css), { // コンパイル後の出力先を指定
        sourcemaps: ".", // ソースマップを出力
      })
  );
};
```

・pipe
処理をつなげて実行していく
例）plumber
エラーが発生しても強制終了させないプラグイン

コード説明(Gulp)

```
// ファイル監視タスク
const watchFiles = () => {
  watch(srcPath.css, series(cssSassDev, browserSyncReload)); // Sassファイルを監視
};

// 開発時に実行するタスク（npx gulp）
exports.default = cssSass;
```

・watch
変更があったときに処理をする

・seties
順番に処理をする（順次処理）

→SCSSファイルの監視をしつつ、変更があればCSSの変換処理とブラウザリロードを順番に行う

コード説明(Gulp)

```
// ファイル監視タスク
const watchFiles = () => {
  watch(srcPath.css, series(cssSass, browserSyncReload));
  watch(srcPath.js, series(jsBundleDevTask, browserSyncReload));
  watch(srcPath.img, series(imgImagemin, browserSyncReload));
  watch(srcPath.html, series(html, browserSyncReload));
};

// 開発時に実行するタスク（npx gulp）
exports.default = series(
  parallel(html, cssSass, jsBundleDevTask, imgImagemin),
  parallel(watchFiles, browserSyncInit)
);

// 本番用ビルド（npx gulp build）
exports.build = series(
  clean,
  parallel(html, cssSass, jsBundleProdTask, imgImagemin)
);
```

・watch
変更があったときに処理をする

・seties
順番に処理をする（順次処理）

→SCSSファイルの監視をしつつ、
変更があればCSSの変換処理とブラウザリロードを順番に行う

コード説明(Gulp)

```
// ファイル監視タスク
const watchFiles = () => {
  watch(srcPath.css, series(cssSass, browserSyncReload));
  watch(srcPath.js, series(jsBundleDevTask, browserSyncReload));
  watch(srcPath.img, series(imgImagemin, browserSyncReload));
  watch(srcPath.html, series(html, browserSyncReload));
};

// 開発時に実行するタスク（npx gulp）
exports.default = series(
  parallel(html, cssSass, jsBundleDevTask, imgImagemin),
  parallel(watchFiles, browserSyncInit)
);

// 本番用ビルド（npx gulp build）
exports.build = series(
  clean,
  parallel(html, cssSass, jsBundleProdTask, imgImagemin)
);
```

・exports.default
開発用で実行するタスクを定義
npx gulpコマンド

・parallel
同時に処理をする（並列処理）

→「npx gulp」がされたら、
①HTML、CSS、JavaScript(開発用)、画像の処理が並列で行われる
②ファイルの監視処理、ブラウザシンクが開始

コード説明(Gulp)

```
// ファイル監視タスク
const watchFiles = () => {
  watch(srcPath.css, series(cssSass, browserSyncReload));
  watch(srcPath.js, series(jsBundleDevTask, browserSyncReload));
  watch(srcPath.img, series(imgImagemin, browserSyncReload));
  watch(srcPath.html, series(html, browserSyncReload));
};

// 開発時に実行するタスク（npx gulp）
exports.default = series(
  parallel(html, cssSass, jsBundleDevTask, imgImagemin),
  parallel(watchFiles, browserSyncInit)
);

// 本番用ビルド（npx gulp build）
exports.build = series(
  clean,
  parallel(html, cssSass, jsBundleProdTask, imgImagemin)
);
```

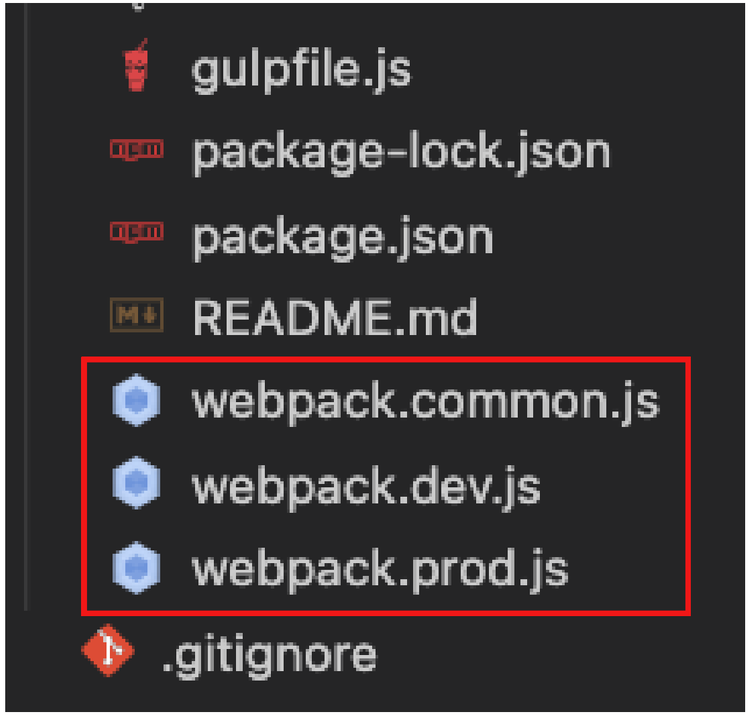
・exports.build
本番用で実行するタスクを定義
npx gulp buildコマンド

・clean
distフォルダの削除処理（別定義）

→「npx gulp build」がされたら、
①distフォルダの削除
②HTML、CSS、JavaScript(本番用)、画像の処理が並列で行われる

※本番用は最終納品用前提なので、ブラウザシンクは不要

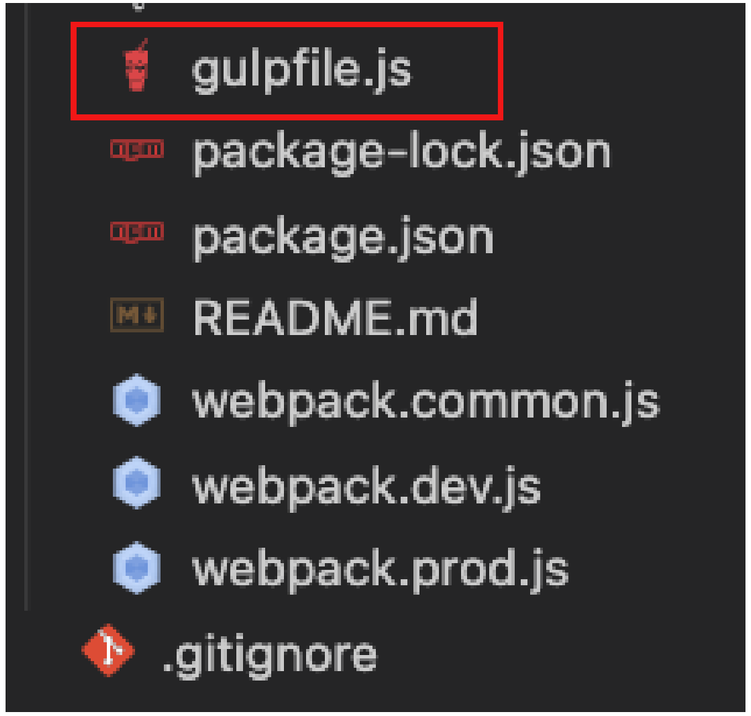
webpackについて



今回の環境は開発用と本番用でファイルを分けている

■理由
・開発用と本番用で利用する機能が異なるためファイルを分ける
・共通にできる機能はwebpack.common.js
・開発用：ソースマップの出力、キャッシュの設定等
・本番用：不要ファイル・コメント等の削除、コードの最小化等

webpackについて




```
.pipe(webpackStream(webpackConfig, webpack))
```

webpackの呼び出しはGulpで行い、実際のJavaScriptの処理はwebpackで実行

webpackについて

webpack.dev.js / webpack.prod.js

```
// 必要なモジュールをインポート
const webpack = require("webpack");
```

必要なモジュールをインポート
webpack：webpackのモジュール

```
module.exports = {
  mode: "development", // 開発モード
  output: {
    filename: "[name].js", // 出力するファイル名
  },
  module: {
    rules: [jsRules], // ルールの設定
  },
};
```

■module.exports
出力するファイルの定義内容
■mode
development：開発用
production：本番用
■output
出力ファイルの定義（pathやfilenameなど）
■module
jsファイルをどのようにまとめるか定義

webpackについて

```
const jsRules = {
  test: /\.js$/, // .jsファイルを対象
  exclude: /node_modules/, // node_modulesディレクトリを除外
  use: {
    loader: "babel-loader", // babel-loaderを使用
    options: {
      presets: [
        [
          "@babel/preset-env",
          {
            useBuiltIns: "usage", // 必要なpolyfillのみをインポート
            corejs: 3, // core-jsのバージョン
          },
        ],
      ],
      // plugin-transform-runtimeとは polyfillをインポートするためのプラグイン
      plugins: ["@babel/plugin-transform-runtime"], // plugin-transform-runtimeを使用
    },
  },
};
```

webpackについて（開発用）

```
module.exports = merge(common, {
  mode: "development", // 開発モード
  devtool: "eval-cheap-module-source-map", // ソースマップの設定
  cache: {
    type: "filesystem", // ファイルシステムをキャッシュタイプとして使用
    buildDependencies: {
      config: [__filename], // このファイルをビルド依存関係として追加
    },
  },
  plugins: [limitChunkCountPlugin], // プラグインの追加
  performance: {
    hints: false, // パフォーマンスの警告を無効化
  },
});
```

■mode
development を指定

■devtool
ソースマップの出力設定

■cache
ビルド時間（実行時間）の高速化

webpackについて（本番用）

```
module.exports = merge(common, {
  mode: "production", // 本番モード
  plugins: [limitChunkCountPlugin], // プラグインの追加
  optimization: {
    minimize: true, // 最小化を有効化
    minimizer: [terserPlugin], // コードの最小化処理。不要な空白やコメントを取り除く。
  },
  performance: {
    hints: false, // パフォーマンスの警告を無効化
  },
});
```

■optimization
コードの最小化処理

→一方で、 cacheやソースマップなどを出力しない。

モジュール形式のJavaScript

■ES modules
ES modulesはJavaScriptファイルを他のJavaScriptファイルで読み込む仕組みのこと。
今回のファイルはES modulesでJavaScriptを読み込んでいます。

・_test.js

```
export function test() {
  console.log("表示テスト");
}
```

- export文を使用して関数を定義

・index.js

```
import { test } from "./_test.js";

test();
```

- import文を使ってファイルを読み込む
- 関数を実行する。

最後に

ご清聴ありがとうございました！

最後に、今回使用したフィアルをご共有いたします。
のちほどURLをDiscord内で共有します。
使い方がわからない場合は、「まさかず」までDMください！

ありがとうございました！