

適当に教える  
最近のフロントエンド開発  
第一歩👍

# 自己紹介

[@pvcresin](#)

- 明治大学 [宮下研究室](#) M1（修士1年）
- [想隆社](#)のWEBフロントエンドを担当
- 本職は[Kotlin](#)でAndroid書くマン

# はじめに

- 最近のフロントエンド開発の第一歩をめちゃくちゃ雑に紹介します（2017/10 地点）
- 目標は「へえ～こういうのもあるんだ～」程度
- 完全に個人的な考えで進めるので、  
全ての言葉には（**諸説あり**）を付けて下さい
- フロントエンドの開発環境は毎年のように移り変わっていくので、このスライドも 2018/10 にはほぼ使いものにならないでしょう（泣

# ちなみに

- このスライドは [Marp](#) というMarkdownからスライドを生成するツールで作成しました（便利）
- あと今回，最終的にできるファイルはここに置いておきます  
<https://github.com/pvcresin/testMarp>  
適宜見比べて下さい

# Menu

- VSCode
- Node.js
  - npm / Yarn
- live reload
- Pug
- PostCSS
- JavaScript (es6)
  - webpack + Babel

# エディタの有名どころ

- [Sublime Text](#)
  - 「恋に落ちるエディタ」として名高い
  - 有料だが、無料でもフル機能を使える
- [Atom](#) (by Github)
  - Electron製 (JSでデスクトップアプリ作る君)
  - 拡張が多く公開されているが、動作が重い印象
- [Brackets](#) (by Adobe)
  - ライブプレビューが標準搭載

# Visual Studio Code

<https://code.visualstudio.com/>

- Microsoftが作ったWeb開発に特価したエディタ
- Win / Mac / Linux 対応で無料
- Git連携やコマンドラインの機能が標準搭載
- Electron製

今回はこのエディタを使っていく

ちなみに自分の場合: Sublime → Atom → VSCode  
Bracketsは使ったこと無い

# Node.js

<https://nodejs.jp/>

- サーバサイドで動くJavaScript
- フロントエンド開発に無くてはならない
- 偶数バージョンが長期サポート（LTS）になる
- 今回は `v8.x.x` を使う（ただの趣味）



# npm

<https://www.npmjs.com/>

- **Node の Package の Manager 的なやつ**
  - 要はライブラリなどを入れるためのツール
- Node入れたら, だいたいデフォルトで入ってる
- [Yarn](#)というFacebookが作った上位互換に押されつつある (適当)
  - npmより速い (ときもある)
  - 最近はこちらを使う人も増えている

# npmの使い方

- `npm init`で`package.json`（大事なアレ）を作成
  - `npm init -y`でとりあえず空のを作ることも可
- `npm run xxx`で`package.json`の`scripts`に定義したコマンドを起動可能（ = **npm script** ）

```
{
  "name": "testMarp",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "scripts": {
    "test": "echo hello"
  }
}
```

# npmの使い方

- `npm install --save xxx` = `npm i -S xxx`
  - `package.json`の`dependencies`（依存関係）にモジュールを追記し, `/node_modules`にダウンロード
- `npm install --dev xxx` = `npm i -D xxx`
  - 上の`devDependencies`バージョン
- `npm i`
  - `package.json`の依存モジュールを一気に入れる
  - 例 : `git clone`してきたNodeのプロジェクトなど

# Yarnの使い方

<https://yarnpkg.com/lang/en/>

- `yarn` = `yarn install` = `npm i`
- `yarn init` = `npm init`
- `yarn add` = `npm i -S xxx`
- `yarn add -D xxx` = `npm i -D xxx`
- `yarn xxx` = `yarn run xxx` = `npm run xxx`

今回はYarn使います

# live reload

- ファイルを変更して保存したら、ブラウザを自動でリロードする機能
- これを実現する色々なパッケージが存在
  - `live-server`, `browser-sync` など
- ちなみに今どきは、ブラウザをリロードせずに変更した部分の要素だけを入れ替える  
`Hot Module Replacement` という技術を使う
  - が今回は難しいので割愛

# live-server を使ってみる

1. `yarn add -D live-server` でモジュールを追加
2. `dist` フォルダを作成し, その中に `index.html` を作成
3. `scripts` に  

```
"watch:browser": "live-server dist --browser=chrome  
--watch=/"
```

を追加
4. `yarn watch:browser` したら, `dist/index.html` を編集して保存するとブラウザがリロードする

出力フォルダ名をよく `dist` にするけど, **distribution** (配布物) や **district** (特定の場所) という説がある

# Pug (旧Jade)

<https://pugjs.org/>

- HTMLを楽に書くための定番プリプロセッサ
- インデントで記述
- パーツごとに分けて記述可能 (`include`)

```
doctype html
html(lang="ja")
  head
    meta(charset="UTF-8")
    title Pug
  body
    h1 Hello
```

# pug-cli を使ってみる

1. `yarn add -D pug-cli` して追加
2. `src/pug/index.pug` を作成
3. `scripts` はこんなの

```
"build:pug": "pug src/pug/index.pug -o dist/ -P",  
"watch:pug": "npm run build:pug -- -w",
```

- `yarn build:pug` で 1 度だけビルド
- `yarn watch:pug` でファイルの変更を監視 (`watch`)  
`src/pug/index.pug` の更新時, `dist/index.html` に出力



# `npm-run-all`を使って並列化

- npm scriptを複数指定し，順番または並列に処理できる
- `run-p build:*`
  - `build:*`にマッチするnpm scriptを **parallel** に **run** するという意味
- `yarn add -D npm-run-all`で追加

# npm-run-allを使って並列化

watch:browser とさっきの watch:pug を組み合わせると

```
"build": "run-p build:*",  
"build:pug": "pug src/pug/index.pug -o dist/ -P",  
"watch": "run-p watch:*",  
"watch:pug": "npm run build:pug -- -w",  
"watch:browser":  
  "live-server dist --browser=chrome --watch=/"
```

- yarn build → build:\* → build:pug
- yarn watch → watch:\*
  - watch:browser
  - watch:pug → build:pug

# 休憩

- これでnpm scriptの基礎は完成
- あとは`build:xxx`と`watch:xxx`を同じように増やしていくことで並列処理を増やしていくことが出来る

# PostCSS

<http://postcss.org/>

- CSSを楽に書くための新しめのプリプロセッサ
  - 他にも[Stylus](#), [LESS](#), [SASS\(SCSS\)](#)などがある
  - SASSがよく用いられていたが, 高機能のため, 変換に時間がかかるのが難点だった
- 欲しい機能をプラグインとして個別に導入が可能
  - 変数を使いたい, ネストしたい...などなど

# PostCSS定番プラグイン

- `postcss-cssnext`
  - まだ導入が進んでいない次世代のCSS記法を先取りして使える
  - 様々なプラグインの集合体でもある
    - `autoprefixer`: ブラウザでの表示差を埋める
    - `postcss-nesting`: ネストしてCSSをかける
- `postcss-simple-vars`
  - SASSのスタイルでCSS内に変数を宣言できる

# postcss-cli を使ってみる

1. `yarn add -D postcss-cli postcss-cssnext postcss-simple-vars`

2. `src/postcss/style.css` を作成

3. `scripts` に

```
"build:postcss": "postcss src/postcss/*.css -d  
dist/css/ --no-map -u postcss-simple-vars postcss-  
cssnext",
```

と

```
"watch:postcss": "npm run build:postcss -- -w",
```

を追記

# PostCSSを触ってみる

- `yarn build`すると, `dist/css/style.css`が出力される
- `src/pug/index.pug`の`head`タグの最後に`link(rel="stylesheet", href="css/style.css")`を追記
- `yarn watch`し, `src/postcss/style.css`を編集すると自動でCSSに変換し, ブラウザをリロード！

# PostCSSのコード例

```
body { // 変換前のpostcss
  $baseColor: cyan;
  background: $baseColor;
  & h1 {
    color: $baseColor;
    background: red;
  }
}
```

```
body { /* 変換後のcss */
  background: cyan;
}
body h1 {
  color: cyan;
  background: red;
}
```



# JavaScript (es6)

- JSの新しい記法 **es6**
  - ECMA Script6のこと( = es2015)
- イメージ
  - **レガシー: es5, モダン: es6, 次の世代: es7**
- es6で書いて, es5に変換するのが主流
  - **Babel** というツールが有名
    - ブラウザ間の差を埋める機能もついている

# es6 で変わったところ

- 変数文 `let`, `const` の導入
  - `let` (再宣言不可), `const` (再宣言・代入不可)
- アロー関数

```
// 従来の関数
var plus = function(x, y) {
  return x + y;
};

// アロー関数
const plus = (x, y) => {
  return x + y;
};
```

# es6 で変わったところ

- テンプレート構文

```
const name = 'ES6';  
console.log(`My name is ${name}.`); // My name is ES6.
```

- `import` / `export`
  - `import plus from './plus'` のように、他のJSファイルでexportされた関数などを読み込める
  - HTMLにおけるJSの読み込み順からの開放
  - 機能を **モジュール** ごとに分割し、自由に組み合わせることが可能に

# webpack

<https://webpack.js.org/>

- 複数のモジュールを1つのファイルにする **バンドラ**
- JSのみならず, CSSや画像もJSファイルにバンドルすることができ, リクエスト数の削減につながる
- 代替品 : [Rollup](#), [Browserify](#), [Fusebox](#)など
- ちなみに
  - [Grunt](#), [Gulp](#)といったツールは **タスクランナー** と呼ばれ, 先程npm scriptで行ったようなことを中心にサポートするツール (最近使わない)

# webpack と Babel を使ってみる

- Babel で es6 の JS を es5 にし, webpack でバンドル
- モジュールの説明
  - babel-core: Babel 本体
  - babel-preset-es2015: 変換するためのプリセット
  - babel-loader: Babel を webpack 上であつかう君
  - webpack: webpack 本体
- ```
yarn add -D babel-core babel-loader babel-preset-es2015 webpack
```

して追加

# webpack + babel

- `webpack.config.js` をルートに作成

```
const webpack = require('webpack')
const path = require('path')

module.exports = {
  context: path.resolve(__dirname, './src/js'),
  entry: {
    index: './index.js'
  },
  output: {
    path: path.join(__dirname, 'dist/js/'),
    filename: '[name].js'
  },
}
```

```
module: {  
  loaders: [{  
    test: /\.js$/,  
    exclude: /node_modules/,  
    loader: 'babel-loader',  
    query: {  
      presets: ['es2015']  
    }  
  }]  
},  
resolve: {  
  extensions: ['.js']  
}  
}
```

少し癖がある + バージョンによっても書き方が変わる  
ので注意

# JSファイル

- `src/js/`に`plus.js`と`index.js`を作成

```
// plus.js
export default (x, y) => {
  return x + y;
};
```

```
// index.js
import plus from './plus';

console.log(plus(2, 3)); // -> 5
```

- `src/pug/index.pug`の`body`タグの最後に`script(src="js/index.js")`を追加



# JSを変換してみる

- `scripts` に下記2つを追加

```
"build:js": "webpack",  
"watch:js": "npm run build:js -- -w",
```

- `yarn watch` すると、ブラウザのデベロッパーツールのコンソールに数字が表示されている
- webpackのせいで、生成されたJSファイルは読めたもんじゃないが、よく見るとちゃんと1つのファイルにバンドルされているのが確認できる

# 完成

- これでメタな言語や新しい記法のファイルを,  
HTMLとCSSとJSのファイルに変換し, ライブリロードする環境が整いました
- あとはそれぞれをいじって学ぶだけです
- `/node_modules`の中身は**git ignore**しましょう  
(`package.json`さえあれば環境構築はできるので)

# おわりに

- お疲れ様でした！
- ちなみに、この先に `Hot Module Replacement` 機能や `React` などのView用フレームワークとルーター、`Flux/Redux` アーキテクチャ、`CSS modules`、`TypeScript` といったものが見えてくるのですが、書いてる自分の体力が持ちませんでした...
- ここまで来てなんなんですが、どちらかというと、HTML5やCSS3、JSのPromiseとかそこら辺がちゃんと出来たほうが良いと思います