実践課題制作

2019/04/22

Kazuma Sekiguchi
class@cieds.jp

Node.js

- 本来は大量の接続に耐えるWebサーバー
  - C10K問題（同時10,000接続に耐える）への解として出てきた
  - 言語としてJavaScriptを用いたサーバーサイドスクリプト環境を提供している
  - 現在もサーバーサイドJavaScriptとして出てくることもある

  - 中身的にはChromeで使われているv8エンジンを元にしている

Node.js

- フロントの世界では。。。
  - JavaScript実行環境とされる
  - JavaScriptはブラウザーから実行するのが一般的だが、それをブラウザー以外で他の機能を付け加えた上で利用可能とした
    - CommonJS（ファイルの書き込みや読込など）
    - 厳密にはNodeはCommonJSから外れる、、らしい
- JavaScriptでアプリを作成できることからフロントの世界で異様に使われている
  - アプリと言ってもGUIは無いため、全てコマンドで操作する

NPM(Node Package Manager)

- Node上で動作するアプリは大量
  - それぞれが依存関係を有していることがある
  - 依存＝他のファイルが無いと動作しない関係
    - リンクに似ているが、リンクは無くても動作するが、依存関係の場合は無いと動作しない
  - 依存関係をきちんと処理してアプリを入れていく必要があるが面倒
    - 1にあれを入れ、2にあれを入れ、、3にあれを入れ、、、、。
- NPMを使うことで、依存関係を処理してアプリを導入してくれる
  - こういうのをパッケージマネージャーと呼ぶ
  - Linuxなどで良く使われている考え方

NPMで導入

- Nodeは実行環境、NPMはアプリ管理ツールとでも
  - NodeをインストールすればNPMは勝手に付いてくる
- Nodeはインストールが必要
  - 比較的アップデートが早い
  - 導入するならLTS版を入れること（LTS＝長期サポート版）
  - Windowsはそのままインストーラーで導入
  - macはhomebrew経由で入れるのがお奨め
    - バージョンの切替ができる（Windowsでは結構面倒）

macにhomebrew経由で入れる

- homebrewが入っていない場合は入れる
  - ターミナルで以下を入力

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

  - インストールが終わったらnodebrewを入れる

```
$ brew install nodebrew
```

  - パスを通しておきましょう

```
$ echo 'export PATH=$HOME/.nodebrew/current/bin:$PATH' >> ~/.bash_profile
```

  - Node本体を入れる

```
$ nodebrew install-binary latest
```

macにhomebrew経由で入れる

- homebrewが入っていない場合は入れる
  - 入っているNodeを確認

```
$ nodebrew ls
```

  - 使うバージョンを指定する

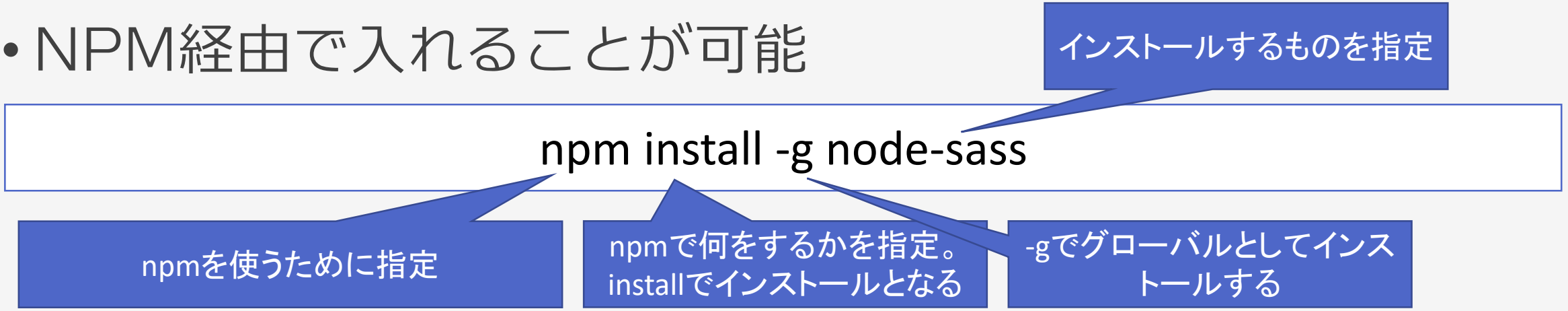
```
$ nodebrew use v7.1.0
```

  - 確認しておきましょう

```
$ node -v
```

NPMでNode-Sassを入れる

- SCSSをCSSにトランスパイルできるのがNode-sass
- NPM経由で入れることが可能



  - npmコマンドでインストールする
    - -g（ハイフンが入るので注意）でグローバルインストールになる
    - ローカルインストールとグローバルインストールがある
    - 通常はローカルインストールを行うが、どのプロジェクトでも利用するようなアプリはグローバルインストールしておくと楽

NPMでNode-Sassを入れる

- ちょっと待っているとインストールが完了する
  - 実体は変なところに入る
  - C:¥Users¥（ユーザー名）¥AppData¥Roaming¥npm
- 何かで指定する場合（WebStormとか）はここを指定する
  - AppDataが隠しフォルダーであるため、直接打たないとダメな場合もある
- 使用する際は、nodeから利用する
  - ダブルクリックしたら使える、というようなものではない

Node-Sassの使い方

- コマンドで入力
  - 対象となるフォルダーまで移動する
  - cdコマンドを利用する

```
$ cd フォルダーパス
```

    - ls（Windowsではdir）でフォルダー内のファイルを表示

  - 移動したらコマンド

```
node-sass style.scss result.css --output-style expanded
```

    - これで書き出される
    - --output-styleで書き出し方を指定可能

Node-Sassの使い方

- オプションで-wまたは-watchを付けると、ファイルを保存すると勝手にCSSを作成してくれる

```
node-sass style.scss result.css --output-style compressed -w
```

  - あとは勝手に保存すれば書き出されるので、楽
  - 止めたいときは、nodeを落とす

NPM Script

- NPMで導入した各パッケージを組み合わせて一連の動作をさせることができるもの
  - Nodeと各パッケージだけで運用できるため比較的楽
  - 生成されるファイルを渡してnpm initコマンドを打って貰えば、勝手に同じパッケージが導入されるため、複数人で同じ環境を
    容易に作ることが可能
    - 1人ががんばってscriptを記述すれば、みんなで使える
    - 各社秘伝のソースがあったりする
- 編集したSCSSをトランスパイルして、プレフィックスを付与して、自動的にブラウザーをリロードして読込直す
  - みたいなことが可能

NPM Scriptのめんどうさ

- 全てテキストでコマンドを記入していく必要がある
  - 意外にエラーが起きやすい
  - エラーは動かしてみて初めて分かることも
  - それぞれのパッケージで使用できるスイッチ（オプション）が違うため、調べるのが結構面倒
  - 一部は設定ファイルを別ファイルに記述する

NPMコマンドでScript導入

- 何らかのパッケージを導入

```
$ npm install –D node-sass
```

  - -Dを付けることで、このフォルダーにだけインストールされる
  - -gだと全体になるが、できれば、-D推奨

- initコマンドでpackage.jsonを作成

```
$ npm init
```

  - packagename:区別できるような英数字を記入（括弧内はデフォルト値。そのままEnterすればそれが入る）
  - packagename以外はデフォルトで大体平気

package.json

- npmを管理しているJSONファイル
  - 中身を編集して利用する
  - dependenciesまたは、devDependenciesにインストールしたパッケージが書き込まれていく
  - 他者にこのファイルを渡して、npm initして貰えば、勝手にインストールしてくれる仕組み
  - npm script自体は、scriptsに記述する

postcssを入れる

- CSSに後処理をしてくれるパッケージ
  - 良く使われるのは、autoprefixer
  - 自動的にベンダープレフィックスを付与してくれる
  - autoprefixerはpostcssのプラグイン扱いなので、postcssを導入し、プラグインとして導入する

```
$ npm install –D postcss-cli
$npm install –D autoprefixer
```

  - postcss.config.jsファイルを作成し、そこに設定を記述する
    - どこまでのベンダープレフィックスを補うかなどを指定可能

postcss.config.js

```
module.exports = {
  plugins: [
    require('autoprefixer')({
      "browsers": [
        "last 3 versions"
      ]
    })
  ]
}
```

```
module.exports = {
  plugins: [
    require('autoprefixer’)({
        grid: true
        })
  ],
};
```

- browsersでいろいろと設定が可能
  - last 3 versionsだと現行ブラウザーから3つ分のベンダープレフィックスが付与される
  - 右の記述だと、CSSGridのベンダープレフィックスが付与される

npm scriptでscssからcssにしてベンダープレフィックス

- scssを記述してcssにする

```
"node-sass --include-path scss scss/style.scss dist/style.css --output-style compressed"
```

- postcssでベンダープレフィックスを付与する

```
"postcss dist/style.css -o dist/style.css --use autoprefixer --no-map"
```

  - 出力先とかは任意に設定変更する
  - この場合、scssはscss/style.scssに書き込み、dist/style.cssとして変換したものを書き込み、ベンダープレフィックスを付与して上書きする

- NPM scriptでこの2つを繋げる

npm scriptでscssからcssにしてベンダープレフィックス

- package.jsonのscriptに記述

```
"build:scss": "node-sass --include-path scss scss/style.scss dist/style.css --output-style compressed",
"build:postcss": "postcss dist/style.css -o dist/style.css --use autoprefixer --no-map",
```

- コロンの左側にコマンド名（任意）を付ける
  - 右側にコマンドを記述する
  - 末尾に「,」を付与する

```
"all": "npm run build:scss && npm run build:postcss",
```

  - 2つをまとめるような記述を行う
  - ＆＆で繋げる

保存時に自動的に実行

- onchangeかnodemonパッケージを導入

```
$ npm install –D onchange
```

- package.jsonのscriptに記述

```
"watch": "onchange scss/style.scss -- npm run all",
```

- onchangeでファイルの改変を検知する
  - onchangeの後でファイル名を指定するとそれが改変されたときにnpmコマンドを実行させる

browsersync

- 変更時にブラウザーを呼び出して自動的にリロードするパッケージ

```
$ npm install –D browser-sync
```

- package.jsonのscriptに記述

```
"serve": "browser-sync start --server --files ¥"**/*¥" ",
```

- --filesで更新を検知するファイルを指定
  - **/*を指定しておけば、大体のファイルを検知できる
  - 検知したら自動的にリロードする
  - localhost:3000で表示
    - localhost:3001で管理画面を表示する

変更検知したらブラウザーをリロード

- ビルドしつつ、ブラウザーリロードを行うなど、複数の仕組みを動作させるときに使用

```
$ npm install –D concurrently
```

- package.jsonのscriptに記述

```
"dev":"concurrently ¥"npm run watch¥" ¥"npm run serve¥""
```

- npm run watchとnpm run serveが動く
  - SCSSやhtmlの変更したときに検知してサーバーがリロードされる