

おもてなしnpmライブ ラリをつくらう

@Himenon

version 1.0.0

自己紹介

- 大学院まで物理専攻
- 社会人2年目
- 銀座にいるフロントエンドエンジニア
- サポートーズでは何度か話している

好きなもの

- カメラ / バイク（乗りたい）
- ライブラリを探す（最近はnpm系）

今日話すこと

- 開発環境について
- npmライブラリとは
- npmライブラリを作ってみる
- 「おもてなし」を実装していく

サンプルコード

- <https://github.com/Himenon/omotenashi-npm>

sample以下に3つ用意してあります

- get-title
- get-title-babel
- get-title-ts

開発環境について

Editor

- Visual Studio Code
- Settings Sync
 - 設定例(Gistのリンク)

Library

- Node = 9.8.0
- npm = 6.4.1

- Visual Studio Codeは初期設定でも補完が優秀なのでオススメです
- Nodeとnpmが入っていればどのOSでも構いません

npmライブラリとは

JavaScript製のパッケージマネージャー

例えば

```
npm install -g web2pdf
```

参考

- <https://www.npmjs.com/>
- <https://www.npmjs.com/package/web2pdf>

npmのQ&A

npmのQ&A

- よくわからない
 - ほぼ `package.json` に書いてあるので1つずつ読んでいく

npmのQ&A

- よくわからない
 - ほぼ `package.json` に書いてあるので1つずつ読んでいく
- 難しい？
 - ~~凝ったことをしなければ~~ 簡単

npmのQ&A

- よくわからない
 - ほぼ `package.json` に書いてあるので1つずつ読んでいく
- 難しい？
 - ~~凝ったことをしなければ~~ 簡単
- 開発で使うにはどんなメリットがあるの？
 - 責務分離 → 1つのことに、集中できるようになる

お題

RLを入力するとサイトタイトルを取得するCLI

まずは作ってみる

まずは作しましょう

初期化

```
npm init
```

```
{  
  "name": "get-title",  
  "version": "1.0.0",  
  "description": "URLを入力するとサイトタイトルを取得するCLI",  
  "main": "./main.js",  
  "scripts": {  
    "test": "jest"  
  },  
  "author": "Himenon",  
  "license": "MIT",  
}
```

CLI用のエントリーポイントを置く

```
get-title
├─ bin
│   └─ cli.js    // ここ
└─ package.json
```

package.jsonにエンリーポイントを記述する

CLI化するポイント

```
"bin": {  
  "get-title": "./bin/cli.js"  
}
```

`npm install`したときに、`get-title`として使える

シンプルに作る

```
// ./bin/cli.js
#!/usr/bin/env node
function main(url) {
  console.log(url);
}
main(process.argv[2]);
```

試してみる

```
$ node ./bin/cli.js https://github.com
https://github.com
```

```
process.argv=["node", "./bin/cli.js", "https://github.com"]
```

ローカルにインストールしてみる

```
npm i -g .
```

```
$ get-title https://github.com  
https://github.com
```

`npm link`を叩くとどこにインストールされているかわかります。

```
/usr/local/bin/get-title -> /usr/local/lib/node_modules/get-title/bin/cli.js  
/usr/local/lib/node_modules/get-title -> /path/to/get-title
```

- <https://docs.npmjs.com/cli/link>

テストを追加する

jestを使う

```
npm i -D jest  
npx jest init  
# 勝手に初期化してくれうr
```

- ちゃんと使い分けてる? dependenciesいろいろ。 - Qiita

実はこのままだとテストしづらい

```
function main(url) {  
  console.log(url);  
}  
  
main(process.argv[2]); // ここが走ってしまう
```

`main`関数の分離をしておく

ファイル分割

```
// ./main.js
function getTitle(url) {
  console.log(url);
}

exports.getTitle = getTitle;
```

```
// ./bin/cli.js
#!/usr/bin/env node
const main = require('../main');

main.getTitle(process.argv[2]);
```

整理

```
get-title
├─ bin
│   └─ cli.js      # CLIのエントリーポイント
├─ main.js         # メインロジック
└─ package.json
```

テストを書く

```
get-title
├─ bin
│   └─ cli.js          # CLIのエントリーポイント
├─ main.js             # メインロジック
├─ __tests__
│   └─ main.test.js    # メインロジックのテスト
└─ package.json
```

`jest`のデフォルト設定で今回は行います。

テストの動作確認

```
// __tests__/main.test.js
describe('メインロジックのテスト', () => {
  test('動作確認', () => {
    expect(1).toEqual(1);
  });
});
```

```
// package.json
"scripts": {
  "test": "jest"
},
```

テストの動作確認

```
npm test
```

```
PASS   __tests__/main.test.js
メインロジックのテスト
  ✓ 動作確認 (4ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.135s
```

動作確認が終わってから、ロジックのテストを書くようにすると、気が楽です。

テストできるコードにする

`getTitle`の返り値を指定する

```
// main.js
function getTitle(url) {
-   console.log(url);
+   return 'The world's leading software development platform . GitHub';
}

exports.getTitle = getTitle;
```


テストする

```
const main = require('../main');

describe('getTitleのテスト', () => {
  test('https://github.com/', () => {
    const title = main.getTitle('https://github.com/');
    expect(title).toEqual('The world's leading software development platform · GitHub');
  });
});
```

`npm test`で確認する



```
PASS  __tests__/main.test.js
メインロジックのテスト
✓ 動作確認 (4ms)

Test Suites: 1 passed, 1 total
Tests:      1 passed, 1 total
Snapshots:  0 total
Time:       1.135s
```

テストを追加する

```
test('http://example.com/', () => {  
  const title = main.getTitle('http://example.com/');  
  expect(title).toEqual("Example Domain");  
});
```

- 今のままだと落ちます



The screenshot shows the output of a Jest test run. At the top, it says "FAIL" in red. Below that, it lists two tests: "https://github.com/" which passed (green checkmark) and "http://example.com/" which failed (red X). The failed test is expanded, showing the error message: "Expected value to equal: 'Example Domain' Received: 'Github'". Below this, the test code is shown with line numbers 10 to 15. Line 12 has an arrow pointing to the "toEqual" method. At the bottom, the summary shows "Test Suites: 1 failed, 1 total" and "Tests: 1 failed, 1 passed, 2 total".

```
FAIL __tests__/main.test.js  
getTitleのテスト  
✓ https://github.com/ (5ms)  
✗ http://example.com/ (8ms)  
  
● getTitleのテスト > http://example.com/  
  expect(received).toEqual(expected)  
  
  Expected value to equal:  
    "Example Domain"  
  Received:  
    "Github"  
  
   10 |   test('http://example.com/', () => {  
   11 |     let title = main.getTitle('http://example.com/');  
 >  12 |     expect(title).toEqual("Example Domain");  
      |                       ^  
   13 |   });  
   14 |  
   15 | });  
  
   at Object.toEqual (__tests__/main.test.js:12:19)  
  
Test Suites: 1 failed, 1 total  
Tests:      1 failed, 1 passed, 2 total
```

テストが通過するように実際のロジックを書く

Headless Chromeのpuppeteerを使う

```
npm i puppeteer
```

- <https://github.com/GoogleChrome/puppeteer>
- <https://pptr.dev/>

タイトルを取るロジックを書く

```
const puppeteer = require('puppeteer');

async function getTitle(url) {
  const browser = await puppeteer.launch(); // puppeteerの起動
  const page = await browser.newPage();    // ブラウザタブの用意
  await page.goto(url);                    // 指定したURLを開く
  const title = await page.title();         // ページタイトルの取得
  await browser.close();                    // ブラウザを閉じる
  return title;                            // タイトルを返す
}

exports.getTitle = getTitle;
```

- GithubのUsageをほぼコピペ

テストを実行する

```
npm test
```

- テストが通過します

```
> jest
PASS __tests__/main.test.js
  getTitleのテスト
    ✓ https://github.com/ (3026ms)
    ✓ http://example.com/ (709ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        4.984s, estimated 5s
Ran all test suites.
```

CLIの修正をする

bin/cli.js

```
#!/usr/bin/env node
const main = require('../main');

- main.getTitle(process.argv[2]);
+ (async() => {
+   const title = await main.getTitle(process.argv[2]);
+   console.log(title);
+ })();
```

動作確認を忘れずに。

公開する

ざっくりといえば、次の手順を踏めば公開できます。

1. <https://www.npmjs.com/>に行き、ユーザー登録をする
2. ローカルで`npm login`をして認証を通す
3. 公開したいライブラリの`package.json`がある位置で`npm run publish`を実行

※ 今回は行いません。

おもてなしを実装

これで終わりではない。

ここからが本番

作ったライブラリに「おもてなし」を実装しよう

ユーザーに対して

- インストール方法
- 使い方
- サンプルコードを用意する
- ドキュメントやpackage.jsonが充実していることが極めて重要
- 一度作ってしまえば、使い回しが効くので最初だけ頑張ればよい

開発者に対して

- 開発環境の構築方法
- テストの方法を書く
- **package.jsonを詳しく書く**

ユーザーに対するおもてなし

最高のUX

インストール

```
npm install [package]
```

動くサンプルコード

```
const hoge = require('hoge')  
hoge();
```

開発者に対するおもてなし

最高のUX

環境構築

```
git clone [repository]  
npm install  
npm start
```

テスト

```
npm test
```

ビルド

```
npm run build
```

何を・どうやって実装するか？

- package.jsonの仕組みをフルに使う
- vscodeの仕組みをフルに使う
- 世の中にある便利なライブラリをフルに使う

```
npm WARN get-title@1.0.0 No repository field.  
audited 17651 packages in 4.483s  
found 0 vulnerabilities
```

npm iをしたとき

```
├─ get-title  
├─ __tests__  
│   └─ JS main.test.js 9+, M  
├─ bin  
│   └─ JS cli.js 9+  
└─ JS main.js 9+  
{ } package-lock.json  
{ } package.json
```

真っ赤

おもてなしリスト

- package.jsonの充実化
- サンプルコードを設置する
- .npmrc / .npmignoreを使う
- babelによるトランスパイル
- Linter / Prettier
- 型 TypeScript / FlowType
- ドキュメントを用意する
- 依存関係を監視する

package.jsonの充実化 その1

```
{
  "scripts": {
    "start": "run-p *:watch",
    "clean": "rimraf lib",
    "prebuild": "npm run clean",
    "build": "babel src --out-dir lib --ignore 'src/__tests__/*.test.js'",
    "build:watch": "npm run build -- --watch",
    "doc": "npx esdoc",
    "lint": "npx eslint src/**/*.js",
    "lint:fix": "npx eslint --fix \"src/**/*.js\"",
    "test": "jest",
    "test:jest": "jest",
    "test:watch": "npm run test:jest -- --watch",
    "prettier": "prettier --write \"**/*.{js,jsx,ts,tsx,css}\"
  }
}
```

package.jsonの充実化 その2

```
{
  "repository": "git@github.com:Himenon/omotenashi-npm.git",
  "engines" : {
    "node" : "<=v6.4.0",
    "npm" : "<=6.4.0"
  },
  "os" : [ "darwin", "linux", "!win32" ],
  "directories": {
    "lib": "lib",
    "example": "sample"
  },
  "typing": "./lib/index.d.ts"
}
```

<https://docs.npmjs.com/files/package.json>

サンプルコード

動くコードが真

```
get-title
├─ package.json
├─ src
│   ├─ cli.ts
│   └─ main.ts
└─ sample
    ├─ package.json // ここから親のpackageを
    ├─ src
    └─ tsconfig.json
```

```
// package.json
"dependencies": {
  "get-title": "file:.."
}
```

ビルドした後に、dependenciesを直で書いてインストールする

.npmrc / .npmignore

環境設定を迷わせない

- .npmrc
 - プロジェクトの構成管理
- .npmignore
 - `npm pack`や`npm publish`ときにignoreするファイル
 - .gitignoreと同じ規則
 - 開発で必要だけど、publishに不要なものを除外するときに使う
- <https://docs.npmjs.com/files/npmrc>
- <https://docs.npmjs.com/misc/developers>

babelによるトランスパイル

新しい文法を使いつつ、後方互換性を保つ

- babelを使う
 - 最新の構文が対応していない場合が往々にしてある
- <https://kangax.github.io/compat-table/>

Lint / Prettier

「インデント」や「シングル・ダブルクォート」の議論は不毛

- eslint / tslint
- 構文の整形(矯正)
 - 本質的な議論に集中することができる
- prettier
 - 対象となる形式がいろいろ。eslint、tslintと併用する。

```
eslint --fix src/**/*.js
```

```
tslint -c tslint.json -p tsconfig.json --fix
```

設定で困ったら"`eslint:recommended`"、"`tslint:recommended`"を使う。

型 TypeScript / FlowType

型はカンニングシート

Pros

- 型安全
- vscodeと相性がいい
- JSを含むプロジェクトでも利用可

Cons

- 慣れが必要
 - 環境構築
-
- プロジェクトをどちらに寄せたとしても、OSSを利用しているとどちらも目にする
 - TypeScriptはpackage.jsonに型定義ファイルの参照先（`types`）だけ配置することが可能

ドキュメント

自動生成する

- esdoc / typedoc

```
# esdoc
```

```
esdoc
```

```
# typedoc
```

```
typedoc --out ./docs/ ./src
```

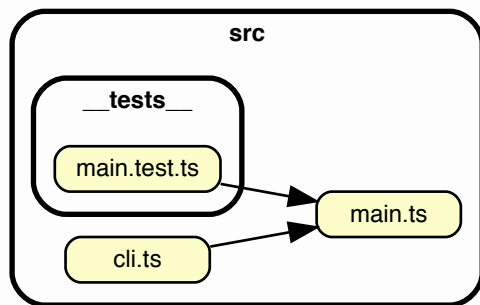
The screenshot shows the npm package page for `get-title-ts`. The page title is `get-title-ts`. Below the title, it says "このライブラリのインストール方法" (How to install this library). A note states: "※ npmには未公開のため、ローカルにcloneしてきてインストールを行う。" (Note: Since it is not published to npm, it must be installed after cloning locally). The installation instructions are: `npm i . -g` and `npm install -g Himenon/omotenashi-npm#master`. The "使い方" (Usage) section shows a TypeScript example: `// TypeScript
import { getTitle } from "get-title";

(async() => {
 const title = await getTitle("https://stackoverflow.com/");
 console.log(`Title = ${title}`);
})`. On the right side, there is a "Globals" section with a table containing the following entries: `"__tests__/_main.test"`, `"cli"`, and `"main"`.

依存関係を確認する

「孤立したコード」や「循環参照」している箇所などをあぶり出す。構造的に不安定なところを検知する

<https://github.com/sverweij/dependency-cruiser>



参考：[dependency-cruiser](#)を使って依存関係を検証し潜在的なバグを潰す - Qiita

まとめ

時間が余ればサンプルコードで解説します

まとめ

- 利用者に対してのおもてなし
 - README
 - サンプルコード
 - ドキュメント
- 開発者に対してのおもてなし
 - テスト
 - npm `start / test / build`

もっとおもてなししたい

- CI/CD
 - travis-ci
- 文法チェック
- フォーマッタ
- コミットフォーマッタ
- TypeScript
- マルチパッケージ管理
 - lerna

開発をサポートするライブラリ

紹介しきれないので、リンクだけ

- [commitlint](#)
- [pre-commit](#)
- [lint-staged](#)
- [cspell](#)
- [dependency-cruiser](#)
- [danger-js](#)
- [node-install-local](#)
- [npm-run-all](#)
- [meow](#)
- [chalk](#)
- [generate-changelog](#)
- [chokidar](#)
- [opn](#)
- [portfinder](#)
- [update-notifier](#)
- [jest-puppeteer](#)

探せばたくさん出てきます。

最後に

- 小さいうちに先手を打つこと
 - 後から導入するものが大変なものもある
- 開発のUXを損ねないように、ドキュメントの整備を行う
 - 暗黙的に使うコマンドに処理をフックさせておくのも手
- コードの保守がもっとも大変。
 - 便利なツールをどんどん使って試していく

おわり