Jestで始める！ ユニットテスト 第1回

環境の準備とテストの実行

JavaScript > テスト　2018年01月04日 発行

JavaScriptのユニットテストのためのツール、Jestの特徴や基本的な使い方を解説します。「そろそろテストを書く習慣を付けたい」、そう思う人のための始めの一歩です。

杉浦 有右嗣
シニアエンジニア

このシリーズの記事

目次

- Jestとは
- Jestを選ぶ理由
- インストールと実行
- 挙動をカスタマイズする
- まとめ

Jestとは

Jestは、Facebook社がOSSとして開発を進めている、JavaScriptのユニットテストのためのツールです。

> Jest・🃏 Delightful JavaScript Testing

日本語のドキュメント

Jestは日本語のドキュメントもあります。日本語で基本的な部分を知るのに役立ちますが、情報が追いついていない場合もありますので、最新情報は英語ドキュメントを参照するようにしてください。

公式ページで「Zero configuration testing platform」と謳っているように、あれこれ設定をすることなく、他の依存ライブラリを追加することもなく、誰でもすぐにユニットテストを書き始められるようになっているのが特徴です。

このシリーズでは、Jestの特徴を紹介しながら、実際にどのようなコードでテストを書くことができるのか、その際にどのような便利な機能が使えるのかを解説していきます。

なお、そもそもテストについてあまり詳しくない・自信がない場合は、先にJavaScript開発のためのテスト入門や、実践、ユニットテストなどのシリーズを一通り読むことをオススメします。

• • •

Jestを選ぶ理由

まずはじめに、筆者が今Jestを選ぶ理由について解説します。

筆者がJestを選ぶ一番の理由は、依存が少なく高機能であることです。プロジェクトの規模が一定以上になってくると、テストのためのライブラリやツールが必要になってきます。その際に必要となるライブラリはいくつかの構成要素に分けられます。

- テストランナー
- アサーション
- テストモック・テストダブル

テストの程度にもよるのですが、テストのありがたみを感じられる規模のプロジェクトでは、この3つが必要になってくるイメージがあります。そしてそれぞれの要素ごとに、いくつものライブラリが存在します。

たとえばテストランナーであれば、次のライブラリのいずれかを選択することが多いように思います。

- Mocha
- Jasmine
- AVA
- tape
- etc...

次にアサーションに使うライブラリの場合です。

- should.js
- Expect
- chai
- power-assert
- etc...

最後に、テストモック・テストダブルによく使われるライブラリです。

- Sinon.js
- testdouble.js
- etc...

これらのライブラリはそれぞれ重複する機能もありますが、特徴として差別化されている点もあります。

さて、これだけの中でベストな組合せはどうやって見つければよいのでしょうか？（もちろん答えはなく、それぞれの好みに応じた選択をしていくことになるのですが）

これだけではなく、たとえばテストのカバレッジを取りたいとなれば、またそのためのライブラリを探し、用意する必要が出てきます。

このように好きなものを好きなように組合せて使うことができるのは利点である反面、それを選ぶ力のない人にとっては不都合でしかありません。

しかしJestを選んだ場合、上述した機能はすべて含まれています。オールインワンなJestだけを依存に追加すれば、すぐにテストコードを書き始められるようになるのです。

「選択肢が多すぎて、テストを書き始める前の段階で足踏みをしてしまう」状況を解消するために、Jestが開発されたのではないかと筆者は考えており、これこそが今Jestを選ぶ理由です。

● ● ●

インストールと実行

Jestを選ぶ理由が理解できたところで、実際にインストールし、テストを書いていくステップに進みます。何はともあれ環境を構築し、テストを書いてみることが一番です。なお、Jestの

使い方は、公式ページに詳しく掲載されています。

- Getting Started · Jest

まずはインストールから始めます。必要なものは`jest`パッケージのみです。

```
# npmなら
npm install --save-dev jest

# yarnなら
yarn add --dev jest
```

他のライブラリと同じく、npmから入手することができます。なお記事執筆時点のJestの最新バージョンは、`22.0.1`です。

後で実行しやすいように、`package.json`の`scripts`に次のように追加するとよいでしょう。

```
{
  "scripts": {
    "test": "jest"
  }
}
```

こうすることで、`npm test`とすることでユニットテストが実行できるようになります。

次に、テストしたい対象のモジュールを適当に用意しましょう。

```
// sum.js
module.exports = function(a, b) {
  return a + b;
}
```

与えられた2つの引数をただ足し合わせるだけの簡単なモジュールです。

そして、このモジュールのテストを用意します。テストコードの書き方については、次回以降詳しく解説しますので、今は流れをつかんでください。

```
// sum.test.js
const sum = require('./sum');
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

この時点のファイル構成は、次のようになるはずです。

```
.
├── package.json
├── sum.js
└── sum.test.js
```

準備はたったこれだけです。

`npm test`として動かしてみると、テストが実行されて結果が表示されます。

```
> jest

 PASS  ./sum.test.js
  ✓ adds 1 + 2 to equal 3 (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.939s
Ran all test suites.
```

設定ファイルも何も用意せず、ユニットテストを実行することができました。

結果を見てみましょう。検証が成功して問題がなかった場合は上のようのPASSと表示されます。もし検証に失敗する（問題があった場合）とFAILと表示されます。

そのほかの項目は、次のことを示しています。

- Test Suites：テストファイルごとの結果
- Tests：テストケースごとの結果
- Snapshots：取ったSnapshotの数
- Time：実行にかかった時間

●●●

挙動をカスタマイズする

このように、設定ファイルがなくても動くJestですが、設定が変更できないわけではありません。

- 設定ファイルを用意する
  - `package.json`の`jest`フィールドにJSONで設定
  - `jest.config.js`を用意して、JavaScriptで設定
- 実行時にCLIのオプションとして渡す
  - CLIにのみ有効なオプションもある

これらのいずれか、または組合せで挙動のカスタマイズが可能です。

- 指定できるオプションの一覧
- CLIで指定できるオプションの一覧

オプションの一覧を見てわかるように、たくさんのカスタマイズできる内容があります。ただ筆者としてもそれらの設定すべてを把握しているわけではなく、必要になったときに調べて利用する程度です。

ここでは逆引き的に、最低限知っておくとよいカスタマイズ方法や使い方をいくつか解説します。

テスト対象のファイル、ディレクトリを指定する

先ほどのインストールの例で、Jestはテスト対象のファイルを指定することなくテストを実行できました。

そこには、`testRegex`というオプションのデフォルト指定が効いています。デフォルトの`testRegex`は`/__tests__/.*|(\\.|/)(test|spec)\\.jsx?$`となっています

testRegexは、/__tests__/.*|(\\.|/)(test|spec)\\.jsx?$となっています。

- __tests__ディレクトリ配下のすべてのファイル
- プロジェクトルート配下のすべてのディレクトリにある、.spec.js or .test.jsで終わるファイル

先ほどのsum.test.jsが実行されたのはこの正規表現にマッチしていたからです。

テストファイルを配置するディレクトリに、テストコードではないがテストのためのヘルパーなどを用意することがあると思いますが、その際にはこのオプションを利用することでテストファイルだと判別されないようにできます。

testRegexによく似たオプションとして、testMatchがあります。

- testRegex: テスト対象のファイル名にマッチする正規表現文字列
- testMatch: テスト対象のファイル名にマッチする正規表現文字列の配列

基本的には、testRegexで事足りると思いますが、正規表現が複雑になるのを避けたい場合などは、testMatchを使うと良いです。ただし、これらはどちらか片方しか使えないことに留意してください。

また、testPathIgnorePatternsにパスの配列を指定することで、逆にブラックリストとして除外することもできます。こちらのデフォルト値は、["/node_modules/"]となっています。

任意のディレクトリにテストファイルを配置したい場合に使えるオプションとして覚えておいてください。

watchモードで起動する

CLIのオプションで--watchを指定することで、gulpやwebpackのように、テストコードに変更がある度にテスト実行することができるモードで起動できます。

```
# npm-scripts経由の場合
npm test -- --watch
# グローバルにjestがインストールされている場合
jest --watch
```

Jestのwatchモードはとても賢く、GitやMercurial(Hg)で差分となっているファイルのみを対象として実行してくれます。よって、差分がない状態だと何も実行しないため、その場合は--watchAllオプションを利用します。

この差分に影響ある内容だけをテストする機能は、-oオプションとして通常モード時に実行することもできます。

テスト結果のログを調整する

いくつか関連するオプションやCLIの引数があります。

- --verbose: テストファイルごとの結果ではなく、テストファイル内の各テストケースごとの結果を表示
- --silent: console.log()やconsole.error()などを非表示

verboseは、オプションとして設定ファイルに記載することもできます。たとえばソースコード内にconsole.log()が記述されていると、そのコードがテストケースで実行された場合にも、テスト結果のログとして出力されてしまいます。実行結果が見づらくなってしまうのを回避するために、筆者は次の様に

を回避するために、筆者は次の使い方をすることが多いです。

- `npm test`で実行するのは`jest --silent`
- 普段の開発・テストのデバッグ時は`npm test -- --silent=false --verbose`

デフォルトは`--silent`を有効にしておきつつ、普段はそれを引数で上書きして使っています。

テストカバレッジを計測する

CLIのオプションで`--coverage`を指定することで、テストのカバレッジを計測できます。

細かな設定が必要な場合は、`collectCoverage`をはじめとする次の指定を設定ファイルに記載します。

- collectCoverage [boolean]
- collectCoverageFrom [array]
- coverageDirectory [string]
- coveragePathIgnorePatterns [array]
- coverageReporters [array]
- coverageThreshold [object]

オプションの詳細は、先のCLIで指定できるオプションの一覧を参照してください。

• • •

まとめ

今回の記事では、なぜJestが開発されたのか、なぜJestを選ぶのかについて解説しました。また、インストールから最低限のユニットテストを実行するまでの流れについても解説しました。

次回以降の記事では、テストコードそのものにフォーカスして、Jestのさらなる機能を見ていきます。

• • •

杉浦 有右嗣

PixelGrid Inc.
シニアエンジニア

SIerとしてシステム開発の上流工程を経験した後、大手インターネット企業でモバイルブラウザ向けソーシャルゲーム開発を数多く経験した。2015年にピクセルグリッドへ入社し、フロントエンド・エンジニアとして数々のWebアプリ制作を手掛ける。2018年に大手通信会社に転職し、低遅延配信の技術やプロトコルを使ったプラットフォームの開発と運用に携わっていたが、2020年ピクセルグリッドに再び入社。プライベートでのOSS公開やコントリビュート経験を活かしながら、実務ではクライアントにとって、ちょうどいいエンジニアリングを日々探求している。