おまけ

Editor's Note

Table of Contents

JavaScript Promiseの本 を書き始めた理由や、どのように書いていったか、またこの書籍はどのような仕組みで作られているかなどについて紹介していきます。

これは本編とは直接関係のないおまけの内容となってるので気楽に読んで下さい。

1. 書籍の始まり

この書籍を書くことにしたのは、私自身が運営する JSer.info にて、あなたが読むべき JavaScript Promises | JSer.info[1] という記事を書いたことが始まりです。

この記事を書く際にECMAScript 6 Promisesについて色々調べながら書いていましたが、最終的に扱いきれてない情報もありました。

- あなたが読むべきPromises by azu · Pull Request #17 · azu/jser.info[2]

またこのとき、電子書籍はどのようなワークフローで書かれているんだろう?ということに興味を持っていて、それを学ぶには実際に電子書籍を書いてみるのが早いと思ったのも書き始めた理由の一つです。

[1] http://jser.info/post/77696682011/es6-promises
[2] https://github.com/azu/jser.info/pull/17

一番最初に自分用の今後やりたいTodoを書いているリポジトリにIssueを立てることから始めました。

- Promisesの薄い本・Issue #7・azu/azu[3]

電子書籍を書くにはどのフォーマットで書くのかを決めなければなりません。

以前、The little book of Buster.JS[4] という電子書籍を書いた際は、Sphinx(reStructuredText)を使って書いていたため、今回は別の方法を取ってみようと考えました。

最初に候補にあげたのは人気のあるMarkdownでした。しかし、Markdownには外部ファイルを読み込んで埋め込む標準的な方法がありません。

The little book of Buster.JS を書いた時からサンプルコードにテストを書くことにしていたので、外部ファイルを読み込む機能は必要なものでした。

拡張したMarkdownフォーマットを利用するのもよいですが、そのときにAsciiDocというOreillyも使っているフォーマットがあることを思い出しました。また、Asciidocのモダンな処理エンジンとして Asciidoctor[5] という実装があることを知り、これを使ってみようと思いました。

まとめると、主目的は電子書籍を書いてみたいという欲求で、サブ目的がES6 Promisesについて扱いきれなかったものを書くというのがこの書籍を書き始めた理由です。

2. 書き始めのワークフロー

おおまかに書く内容と形式を決めましたが、私は形から入るタイプなので、Asciidocで書くときにどのようなプロジェクト構造がよいのか悩んでいた記憶があります。

そのため、一番最初に書いたのは CONTRIBUTE.md[6] で、コントリビュートガイドを書くという形でプロジェクトの構造がどうあるべきかについて決めていきました。

書籍の内容としては、Promiseの基本的な使い方、パターンをコード例中心に書いていくというおおまかな方針が決まっていたので、それをより明確にするためにアウトラインを固めていきました。

- この書籍で扱う内容について by azu・Pull Request #1・azu/promises-book[7]

[3] https://github.com/azu/azu/issues/7
[4] http://the-little-book-of-busterjs.readthedocs.org/en/latest/
[5] http://asciidoctor.org/
[6] https://github.com/azu/promises-book/commit/bbf23086c5bbaf60bd9991b5b1a4229ce54dfb30
[7] https://github.com/azu/promises-book/pull/1

アウトラインを固めつつ、書籍を書く上で一番コワかったのが飽きることだったので、とりあえずでもいいから気になったことについてをセクションに分けて書いていきました。

そのため、書いた順番はかなりバラバラで、第二章や第四章を最初の頃に書いていました。

多くのセクションは、最初にコンセプトとなるサンプルコードとテストを書いて、それに対する解説として文章を書いていくという方針で行っていたものが多かったと記憶しています。

そのため、いいサンプルコードが思いつかないセクションは後回しにする傾向がありました。

3. 移動中に書く

飽きる前に一度完成の形まで持っていきたいという気持ちがあったため、とりあえずの目標として三ヶ月でひとまずの完成まで持っていくとしていました。

この書籍以外にもやりたいことは山ほどあるので、どう目標を達成するか考える前に、空いてる時間を使っていくしかないという感じがしてました。

空いてる時間として一番ありそうなのは移動中の時間でした。

移動中でもMacBook Proの上にThinkPad Bluetooth keyboardを載せて使えば、普段とあまり変わらないレベルで書けることがわかったので、移動中にコーディングや文章を書き進めていきました。

もちろん、この文章も移動中に書いてます。

移動中に書く短所としては、電波があまり強くないのでオフライン環境になってしまうケースがあることでした。そのため、移動する前に移動中に何を書くのかを計画してから進める癖が自然とついた気がします。

その影響はGitHub Issueの使い方に強く出ていたと思います。

実際にセクションを書く前に、GitHub Issueにそれぞれのセクションでやりたいことや、コンセプトとなるサンプルコードのアイデアを大量にメモるようにしていました。

メモやアイデアを元に、移動中に実装などを進めることで結構集中してできたと思います。それぞれのセクションはGitのブランチを切って進めて、`[WIP]` のpull-requestを出した状態で進めていました。

`WIP` のpull-requestを作って置くことで、どこまで進めたのかや別のアイデアが思いついた時は書くことができるため、書いてる文章やコードに対する意見等も記録して残しやすくなりました。

一度の移動中でセクションが完成することは少ないので、どこまでやったかを簡単に見られるようにすることは結構大事でした。

2014年3月2日に書き始めて、2014年6月2日に全ての章が書き終わり、リファクタリングに入ったので、大体目標を達成できたように思えます。

4. GitHub Issueを使ったワークフローの変遷

先ほども書いたように、WIP pull-requestを使ったワークフローを取っていましたが、最初からこのワークフローで書いていた訳ではありません。

最初の頃はGitHub Issueにアイデアをメモるだけで、殆どmasterブランチで作業していましたが、第四章の応用の内容を書いていくあたりからワークフローが変わってきました。

この書籍の第四章では、第二章の基礎を発展させた応用的な使い方について書いています。しかし、それまで好き勝手書いていたため、応用するにも基礎の部分の説明をまだ書いてないというセクションがでてきました。

セクション毎にIssueは立てていたので、どのIssue同士が関係あるのかをIssue References(Issue番号書くだけ) で整理したり、文章全体としての流れを汲み取っていく必要が出てきました。

大抵は一つのセクションが一つのIssueと対応していたので、作業も一つのセクションと一つのpull-requestを対応させて行った方が管理しやすいと気付きました。

これが自分自身にWIP pull-requestをするようになった理由だと思います。

GitHub Issueを使った執筆のワークフローについては 一人で使える Github Issue[8] にまとめています。

5. 文章のレビュー

物理的な書籍と違っていつでも更新できるとはいえ、最低限おかしなところは直さないと行けないため、文章をひととおり書き終わってから誤字などチェックを始めました。

また、最初に好き勝手書いてた影響もあり、セクションの順番を変えたほうがよい箇所もあるなど、全体的な流れを直す作業もレビューと一緒にやっていました。

全体的な流れを見るために、あるセクションがどのセクションに依存してるか、逆にどのセクションから参照されてるかを見るためのツールを書いて、依存関係がおかしくないかを確認していきました。

[8] https://azu.github.io/slide/udonjs/github-issue.html

この影響で第二章のセクションの一部が第四章に移動したものもあり、第二章はPromiseのメソッドの解説に集中した感じに変わったと思います。

誤字脱字などは @vzvu3k6k[9] さんにたくさんのpull-requestを送ってもらったり、自分もiPhone等のモバイル端末から直接GitHub Issueを立てられるようにして一文字のtypoのIssue等を大量に立ててチェックしていきました。

HTMLで見られるようにするとモバイルでも十分文章のレビューはできるので、作っておいたIssueをTiDD(チケット駆動開発)の要領で処理していくと文章の修正もテンポよく進められました。

ここでもGitHub Issueを活用していましたが、typoのような小さな修正と新規セクションを書くような大きな変更では、使い方の違いがでてきた気がします。

どちらもブランチを切ってコミットする所までは同じですが、小さな修正はコミットメッセージに `fix #108` というように書いてマージするだけで、わざわざpull-requestはしてませんでした。

コミットメッセージのルールは Angular.jsで使われている Git Commit Guidelines[10] をベースにしています

逆に大きな修正はpull-requestを使って進めることで、マージする前にもう一度確認しやすかったりやTravis CIによる自動テストが走るため、ミスが減った気がします。

一人で書いてる書籍だったので、機械的にチェック出来るところを出来るだけ多くして間違いを減らそうとしていました。自分自身にpull-requestsを送るやり方は機械的なチェックを挟みやすかったので、このワークフローを体感出来たのは良かったと思います。

しかし、レビュー時に立ったIssueの7割ぐらいは日本語的な問題だったので、日本語は難しいなーと思いました。

6. おわりに

最初の目的にあったように電子書籍をどうやって書いていくのかやGitHub Issueの使い方について ある程度学べるところはあったかなーという感じがします。

文章を書いていくだけじゃなくて、上手くサイクルを回すために色々なツールを自作していて、これは車輪の再発明じゃないかなと思うことがありました。

[9] https://github.com/vzvu3k6k
[10] https://github.com/ajoslin/conventional-changelog/blob/master/CONVENTIONS.md

こういう(電子)書籍を書くノウハウについてもっと色々公開されていけばいいなと思いつつ、これでおまけを〆させていただきます。

最後にこの書籍を書くにあたって作成したツールや Travis CIで回してるテストについて紹介して終わりたいと思います。

7. 執筆のために作成したツール

この書籍を書いている最中に開発した生成ツールやTravis CIと連携したテストについて。

7.1. HTML/Asciidoc

表示用JavaScriptの生成

書籍のサンプルコードは外部ファイルにしたかった。しかし、そのまま読み込む使い方だと、サンプルコードのモジュール化が難しくなってしまった。モジュール化が上手くできないとテストを書くことが難しくなる。そこで、サンプルコード(`src`)と表示用コード(`embed`)にわけることにした。

そのために、サンプルコードから表示用コードを生成するモジュールを書いた。

- azu/inlining-node-require[11]
- azu/remove-use-strict[12]

Browserify等の既存のモジュールビルドツールでは `require` のエミュレートをするコードが入るため、結合したコードがあまりキレイではない。

そこで作成したのが inlining-node-require でCommonJSのコードを見た目そのままに結合をすることができる。

> inlining-node-require は見た目はキレイに結合するが、あらゆるパターンに対応することはできないため、ある程度の制限を持った書き方が必要になる。

サンプルコードは `"use strict"` を使ったコードとなっているが、inlining-node-require で結合した際に重複することや表示用コードではスペース的に余計なものとなる。remove-use-strict は不必要な `"use strict"` を取り除くことができるツール。

[11] https://github.com/azu/inlining-node-require
[12] https://github.com/azu/remove-use-strict

動作の詳細については下記の記事で紹介している。

- Node.jsのrequireをインライン化、無駄なuse strictを取り除くモジュールを書いた | Web scratch[13]

サンプルコードの実行エディタ

Webで公開する書籍のメリットとして、その場でコードの実行結果が見られることが大きい。書籍やウェブでの連載でもJSFiddle等のコード公開へのリンクを貼ったものが増えてきている。

そのため、サンプルコードがその場で実行できるのはメリットでありつつ、必要不可欠なものである。

日本語に対応したJavaScriptのコードエディタとしては CodeMirror[14] が良くできているため、CodeMirrorをベースにコードの実行機能を付加するモジュールを作成した。

- azu/codemirror-console[15]
- azu/codemirror-console-ui[16]

codemirror-console はCodeMirrorに書かれているコードを実行できるモジュール。単純にコードをevalするだけでは、グロールスコープを汚染してしまうため、コードを実行する時にiframeを作り、その中でコードを実行している。

NodeのvmモジュールNode.jsのvmモジュール[17]と似た考え方をもつ context-eval[18]を利用した。これにより単純に実行するだけではなく、`console` を独自のものとすり替えて実行することができるようになっている。

codemirror-console-uiはcodemirror-consoleのUIを提供するモジュール。

codemirror-consoleは実行する機能のみであるため、UIは別モジュールとして作成した。

[13] http://efcl.info/2014/0316/res3719/
[14] http://codemirror.net/
[15] https://github.com/azu/codemirror-console
[16] https://github.com/azu/codemirror-console-ui
[17] http://nodejs.org/api/vm.html
[18] https://github.com/amasad/context-eval/

```
var promise = new Promise(function(resolve){
    console.log("inner promise");
    resolve(42);
});
promise.then(function(value){
    console.log(value);
});
console.log("outer promise");
```

実行 ログをクリア 終了

```
inner promise
outer promise
42
```

- https://github.com/azu/promises-book/issues/18
- https://github.com/azu/promises-book/pull/121

7.2. Testing

サンプルコードのテスト

サンプルコードはNode.js上で動くものとして作り、Node.jsで動くテストを書いた。

ES6 PromisesはDOMではなくECMAScriptの仕様であるため、ブラウザが実行環境ではなくてもよいという考えと、実行の手軽さからNode.js上で動くものとして書いていった。

しかし一部を除いて、表示用コードはブラウザでの表示と実行を前提としている。サンプルコードから表示用コードを自動的に生成しているが、これはモジュールに関することのみを解決するため、サンプルコードはどちらの環境でも動く書き方が必要となった。

Node.js上でもブラウザと同様の機能がテストできるようにするためのPolyfillやモックが必要となった。DOM APIが中心となり、具体的には `XMLHttpRequest` や `Notification` 等が該当する。

XHR

Node.jsには `XMLHttpRequest` がないため、`XMLHttpRequest` と同じインターフェイスを持った HTTP通信ライブラリが必要となる。

この書籍では w3c-xmlhttprequest を使い、Node.jsでも `XMLHttpRequest` と同じインターフェイスで通信をするコードを書くことができた。

`XMLHttpRequest` と同じインターフェイスをNode.jsに提供するライブラリには以下のようなものがある。

- ykzts/node-xmlhttprequest[19]
- driverdan/node-XMLHttpRequest[20]
- pwnall/node-xhr2[21]

どのライブラリも `XMLHttpRequest` の全ての機能が使えるわけではないため、用途に合わせたものを選ぶ必要がある。

このライブラリをテスト実行前に、`global` に追加することで、ブラウザとNode.jsで同様のコードを動かせるようにした。

```
global.XMLHttpRequest = require('w3c-xmlhttprequest').XMLHttpRequest;
```

詳細は以下で紹介されている

- Test Runner Tips[22]

Web Notifications API

テストできるようにすることが目的であるため、機能まで持ってくる必要性はなかった。そのため、Web Notifications APIではNoticationのモックオブジェクトを作成し、テストを行った。

- mock-notification.js[23]

出力したHTMLのテスト

Asciidocでは `???` という記法で内部リンクを貼ることができるが、この内部リンク先をリファクタリング時に変更してしまうことがあった。

そのため、生成したHTMLから内部リンクを取得して、移動先となる要素が 確かに存在するのかをテストするスクリプトを走らせている。

19 https://github.com/ykzts/node-xmlhttprequest
20 https://github.com/driverdan/node-XMLHttpRequest
21 https://github.com/pwnall/node-xhr2
22 https://azu.github.io/slide/hasakurajs/
23 https://github.com/azu/promises-book/blob/master/Ch4_AdvancedPromises/test/mock/mock-notification.js

- https://github.com/azu/promises-book/issues/25

Asciidoc上のインラインコードテスト

この書籍中のコードには大きく分けて2種類ある。ひとつは外部ファイルとして書いてテストも書いているサンプルコード。もう一つは直接Asciidocのファイルに書いているインラインコードである。

外部ファイルのサンプルコードはテストしているため動作に問題ないことを保証できるが、インラインコードは直接書くため実行して確認せず間違ったコードを書いてしまいがちだった。

そのため、Asciidocのファイルをパースして、インラインコードを抽出し、そのコードがEsprima[24]といったJavaScriptパーサでパースできるかを検証できるようにした。

これによりJavaScriptの文法として間違っているものはパースエラーとなるため、インラインに書いたコードのミスを検出するのに役立った。

- https://github.com/azu/promises-book/issues/52

Asciidoctorのビルドテスト

この書籍はAsciidoc形式で書き、asciidoctorによりビルドしている。

Asciidoctorではリソースが欠損してもエラーではなくWARNINGとなるため、ビルドするときにWARNINGが発生したらCIが落ちるようにした。

- https://github.com/azu/promises-book/issues/54

7.3. Review

プレビュー

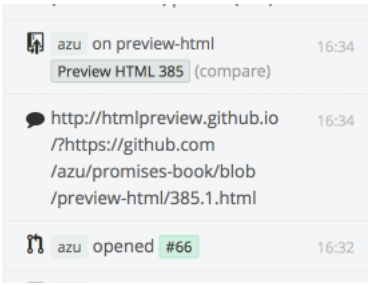
masterへマージされたものは、Travis CIで自動的にビルドして `gh-pages` ブランチにpushする。これによりmasterへのコミットやpull-requestsをマージしたら自動的に https://azu.github.io/promises-book/ にて見られるようにしていた。

- promises-book/_tools/deploy-gh-pages.sh at master · azu/promises-book[25]

[24] http://esprima.org/
[25] https://github.com/azu/promises-book/blob/master/_tools/deploy-gh-pages.sh

pull-requestsのコミットに対しては、そのコミットごとに `preview-html` ブランチに生成済みのHTMLがpushされる。

pushされた一時プレビュー用のURLをGitterに対して通知して、pull-request時のHTMLがプレビューできるようになっている。



- promises-book/_tools/deploy-preview-html.sh at master · azu/promises-book[26]

7.4. 依存関係の可視化

- azu/visualize-promises-book[27]

セクション毎にテーマを分けてることが多いが、それを俯瞰的にどうやってみるかを模索するために、セクション同士の依存関係を可視化するものを作成した。

7.5. Lint

WEB+DB PRESS用語統一ルール[28] の辞書を使うために、辞書のパーサーを書いた。

- azu/wzeditor-word-rules-parser[29]
- WEB+DB PRESS用語統一ルール(WZEditor)のパーサを書いた | Web scratch[30]

[26] https://github.com/azu/promises-book/blob/master/_tools/deploy-preview-html.sh
[27] https://github.com/azu/visualize-promises-book
[28] https://gist.github.com/inao/f55e8232e150aee918b9
[29] https://github.com/azu/wzeditor-word-rules-parser
[30] http://efcl.info/2014/0616/res3931/