Table of Contents

JavaScript Plugin Architecture 

この書籍はJavaScriptのライブラリやツールにおけるプラグインアーキテクチャについて見ていくことを目的としたものです。

次の形式で読むことができます。

- Web版
- PDF形式
- ePub形式
- Mobi形式

この書籍のソースコードは、次のGitHubリポジトリに公開されています。

- azu/JavaScript-Plugin-Architecture: JavaScriptプラグインアーキテクチャの本

Twitterのハッシュタグは#js_plugin_book

更新情報はRSSやリリースノートから見ることができます。

はじめに

JavaScriptの世界では1つの大きなライブラリよりも小さなライブラリを組み合わせていくようなスタイルが多く見られます。小さなものを組み合わせて作るためには、プラグインと呼ばれる拡張の仕組みが必要となります。またそのようなプラグインがたくさんあるエコシステムの土台を作るには、プラグインアーキテクチャが重要になるといえます。

> ソフトウェアの構造に「プラグイン機構」を設け、ユーザコミュニティから開発者コミュニティへの質的な転換を図るのは、ソフトウェア設計からエコシステム設計へとつながる
> -- OSS開発の活発さの維持と良いソフトウェア設計の間には緊張関係があるのだろうか? - t-wadaのブログ

この書籍では、JavaScriptにおけるプラグインアーキテクチャやそのエコシステムを形成してるライブラリやツールなどの実装を学ぶことが目的となっています。

JavaScriptの基本的な文法などについては解説していないため、次の書籍を参照してください。

- JavaScript Primer - 迷わないための入門書 #jsprimer

この書籍の内容について

jQuery

jQueryのプラグインについて解説しています。 `<script>` タグをベースとしたプラグインアーキテクチャについて解説しています。

ESLint

ESLintのルールを拡張する仕組みについて解説しています。 ESLintではJavaScriptのコードをパースして作成されたASTを元にコードのLintを行います。実際にESLintのルールを解釈できる小さな実装を作りながらプラグインの仕組みについて学びます。

Connect

Connectの middleware と呼ばれるプラグインアーキテクチャについて解説しています。 Node.js以外においてもRackなどHTTPサーバーでよく見られるプラグインを使った階層構造について学びます。

gulp

タスク自動化ツールとして知られるgulpのプラグインアーキテクチャについて解説しています。 gulpではデータの流れとして既存のNode.js Streamを使い、そこで流すデータとしてvinylオブジェクトを利用します。実際にgulpプラグインを書きながら、gulpのプラグインの仕組みについて学びます。

Redux

アプリケーションのStateを管理ライブラリのReduxのプラグインアーキテクチャについて解説しています。 Reduxでは middleware と呼ばれる拡張の仕組みを持っていますが、Connectとの類似点や相違点があります。小さなReduxの実装を作りながら middleware の仕組みについて学びます。

Contributing

この書籍は無料で読むことができ、同時に修正や新しいページを書く権利があります。

CONTRIBUTING.mdに、書籍で扱うべきプラグインアーキテクチャのProposalの書き方や Pull Request、コミットのやりかたなどが書かれています。

間違いやライブラリのアップデートへの追従など何かあれば、IssueやPull Requestをよろしくお願いします。

ソースコードはすべてGitHubに公開されています。

- azu/JavaScript-Plugin-Architecture

Author

- github/azu
- twitter/azu_re

License

jQuery

> この文章はjQuery 2.1.4を元に書かれています。

jQueryでは `$.fn` を拡張することで、 `$()` の返り値となるjQueryオブジェクトにメソッドを追加できます。

次の `greenify` プラグインでは、 `$(document.body).greenify();` というメソッド呼び出しが可能になります。

```
(function ($) {
    $.fn.greenify = function () {
        this.css("color", "green");
        return this;
    };
})(jQuery);
```

実際に利用するため際は、 `jquery.js` を読み込んだ後に `greenify.js` を読み込ませる必要があります。

```
<script src="jquery.js"></script>
<script src="greenify.js"></script>
```

どのような仕組み?

このjQueryプラグインがどのような仕組みで動いているのかを見てみましょう。

jQueryプラグインはprototype拡張のように `$.fn.greenify = function (){}` と拡張するルールでした。

`jQuery.fn` の実装を見てみると、実態は `jQuery.prototype` なので、prototype拡張していることがわかります。

```
// https://github.com/jquery/jquery/blob/2.1.4/src/core.js#L39
jQuery.fn = jQuery.prototype = {
    // prototypeの実装
};
```

`$()` は内部的に `new` をしてjQueryオブジェクトを返すので、このjQueryオブジェクトではprototypeに拡張したメソッドが利用できます。

```
$(document.body); // 返り値はjQueryのインスタンス
```

つまり、jQueryプラグインはJavaScriptのprototypeをそのまま利用しているだけに過ぎないということがわかります。

どのような用途に向いている?

jQueryプラグインの仕組みがわかったのでどのような用途に有効な仕組みなのか考えてみましょう。

単純なprototype拡張なので、利点はJavaScriptのprototypeと同様です。動的にメソッドを追加するだけではなく、既存の実装を上書きするmonkey patchのようなものもプラグインとして追加できます。

どのような用途に向いていない?

これもJavaScriptのprototypeと同様で、prototypeによる拡張は柔軟すぎるため、 jQuery自体がプラグインのコントロールをすることは難しいです。

また、プラグインが拡張するjQueryの実装に依存しやすいため、 jQueryのバージョンによって動かなくなるプラグインが発生しやすいです。

jQueryではドキュメント化されてないAPIを触っていけないというルールを設けていますが、これは必ずしも守られているわけではありません。

実装してみよう

`calculator` という拡張可能な計算機をjQuery Pluginと同じ方法で作ってみたいと思います。

`calculator` は次のような形となります。

```
function calculator(value = 0) {
    if (!(this instanceof calculator)) {
        return new calculator(value);
    }
    this.value = value;
}
// alias
calculator.fn = calculator.prototype;
export default calculator;
```

`$.fn` と同様に `prototype` へのaliasを貼っているだけのただのコンストラクタ関数です。

```
calculator.fn = calculator.prototype;
```

`calculator(初期値)` と書けるようにしているため、少し特殊なコンストラクタとなっていますが、この拡張の仕組みとは関係ないのでとりあえず置いておきましょう。

calculator.jsには何も実装が入ってないので、プラグインで四則演算の実装を追加してみます。

```
import calculator from "./calculator";
calculator.fn.add = function (x) {
    this.value += x;
    return this;
};
calculator.fn.sub = function (x) {
    this.value -= x;
    return this;
};
calculator.fn.multi = function (x) {
    this.value *= x;
    return this;
};
calculator.fn.div = function (x) {
    this.value /= x;
    return this;
};
```

calculator-plugin.jsでは、 `calculator.fn.add` というように `add` というメソッドを追加しています。

また、モジュールで依存関係を示していますがやっていることはjQueryと同じで、calculator.jsを読み込んでからcalculator-plugin.jsを読み込んでいるだけですね。

```
<script src="calculator.js"></script>
<script src="calculator-plugin.js"></script>
```

これを使うと `calculator#add` といったメソッドが利用できるようになるので、次のように書くことができます。

```
import assert from "assert";
import calculator from "./calculator";
import "./calculator-plugin"; // Extend

const resultValue = calculator(0).add(10).multi(10).value;
assert.equal(resultValue, 10 * 10);
```

実装をみてもらうと分かりますが、JavaScriptの `prototype` の仕組みをそのまま利用しています。そのため、「拡張する時は `calculator.prototype` の代わりに `calculator.fn` を拡張する」というルールがあるだけともいえます。

エコシステム

このプラグインの仕組みはあるグローバルオブジェクトに依存しており、これはスクリプトを `<script>` 要素で読み込むだけで拡張することを前提とした作りです。

```
<script src="jquery.js"></script>
<script src="greenify.js"></script>
```

Node.jsで使われているCommonJSやES6 Modulesなどがなかった時代に作られた仕組みなので、それらと組み合わせる際には少し不向きな拡張の仕組みといえるかもしれません。

まとめ

ここではjQueryのプラグインアーキテクチャについて学びました。

- jQueryプラグインは `jQuery.fn` を拡張する
- `jQuery.fn` は `jQuery.prototype` と同じである
- jQueryプラグインとは `jQuery.prototype` を拡張したものといえる
- 何でもできるためプラグインが行うことを制御することのは難しい

参考資料

- Plugins | jQuery Learning Center
- jQuery拡張の仕組み 〜 JSおくのほそ道 #013 - Qiita
- The npm Blog — Using jQuery plugins with npm

ESLint

> この文章はESLint v7.8.1を元に書かれています。

ESLintはJavaScriptのコードをJavaScriptで書かれたルールによって検証するLintツールです。

大まかな動作としては、検証したいJavaScriptのコードをパースしてできたAST(抽象構文木)をルールで検証し、エラーや警告を出力します。

このルールがプラグインとして書くことができ、ESLintのすべてのルールはプラグインとして実装されています。

今回はESLintのプラグインアーキテクチャがどうなっているかを見て行きましょう。

どう書ける?

ESLintでは `.eslintrc` という設定ファイルに利用するルールを設定して利用します。そのため、実行方法についてはドキュメントを参照してください。

- Documentation - ESLint - Pluggable JavaScript linter

ESLintにおけるルールとは、次のような `create` メソッドをもつオブジェクトをexportしたモジュールです。
`create` メソッドには `context` オブジェクトが渡されるので、それに対して1つのオブジェクトを返すようにします。

```
module.exports = {
    meta: { /* ルールのメタ情報 */ },
    create: function (context) {
        return {
            "MemberExpression": function (node) {
                if (node.object.name === "console") {
                    context.report({
                        node,
                        message: "Unexpected console statement."
                    });
                }
            }
        };
    }
};
```

ESLintではコードを文字列ではなくASTを元にチェックしていきます。 ASTについてはここでは詳細を省きますが、コードをJavaScriptのオブジェクトで表現した木構造のデータだと思えば問題ないと思います。

たとえば、

```
console.log("Hello!");
```

というコードをパースしてASTにすると次のようなオブジェクトとして取得できます。

```
{
    "type": "Program",
    "body": [
        {
            "type": "ExpressionStatement",
            "expression": {
                "type": "CallExpression",
                "callee": {
```

```
                    "type": "MemberExpression",
                    "computed": false,
                    "object": {
                        "type": "Identifier",
                        "name": "console"
                    },
                    "property": {
                        "type": "Identifier",
                        "name": "log"
                    }
                },
                "arguments": [
                    {
                        "type": "Literal",
                        "value": "Hello!",
                        "raw": "\"Hello!\""
                    }
                ]
            }
        }
    ],
    "sourceType": "script"
}
```

- JavaScript AST explorer

ESLintではこのASTを使って、no-console.jsのように`console.log`などがコードに残ってないかなどをルールを元にチェックできます。

ルールをどう書けるかという話に戻すと、`context`というオブジェクトはただのユーティリティ関数の集合と考えて問題ありません。ルールの本体は`create`メソッドが`return`してるメソッドをもったオブジェクトです。

このオブジェクトはNodeのtypeをキーとしたメソッドを持っています。そして、ASTを探索しながら「`"MemberExpression"` typeのNodeに到達した」と登録したルールに対して通知(メソッド呼び出し)を繰り返しています。

先ほどの`console.log`のASTにおける`MemberExpression` typeのNodeとは次のオブジェクトのことを言います。

```
{
    "type": "MemberExpression",
    "computed": false,
    "object": {
        "type": "Identifier",
        "name": "console"
    },
    "property": {
        "type": "Identifier",
        "name": "log"
    }
}
```

no-console.jsのルールを見ると`MemberExpression` typeのNodeが`node.object.name === "console"`となった場合に、`console`が残ってると判断してエラーレポートすると読めてくると思います。

ASTの探索がイメージしにくい場合は、次のルールで探索の動作を見てみると分かりやすいかもしれません。

- azu.github.io/visualize_estraverse/

```
function debug(string){
    console.log(string);
}
debug("Hello");
```

その他、ESLintのルールの書き方についてはドキュメントや次の記事を見てみるといいでしょう。

- Working with Rules - ESLint - Pluggable JavaScript linter
- コードのバグはコードで見つけよう！ | サイバーエージェント 公式エンジニアブログ

どのような仕組み?

ESLintはコードをパースしてASTにして、そのASTをJavaScriptで書いたルールを使いチェックするという大まかな仕組みは分かりました。

次に、このルールをプラグインとする仕組みがどのように動いているのか見て行きましょう。

ESLintのLintは次のような3つの手順で行われています。

1. ルール毎に使っている `Node.type` をイベント登録する
2. ASTをtraverseしながら、 `Node.type` のイベントを発火する
3. ルールから `context.report()` された内容を集めて表示する

このイベントの登録と発火にはEventEmitterを使い、 ESLint本体に対してルールは複数あるので、典型的なPub/Subパターンとなっています。

擬似的なコードで表現すると次のような流れでLintの処理が行われています。

```
import {parse} from "esprima";
import {traverse} from "estraverse";
import {EventEmitter} from "events";

function lint(code){
    // コードをパースしてASTにする
    const ast = parse(code);
    // イベントの登録場所
    const emitter = new EventEmitter();
    const results = [];
    emitter.on("report", message => {
        // 3. のためのreportされた内容を集める
        results.push(message);
    });
    // 利用するルール一覧
    const ruleList = getAllRules();
    // 1. ルール毎に使っている`Node.type`をイベント登録する
    ruleList.forEach(rule => {
        // それぞれのルールに定義されているメソッド一覧を取得
        // e.g) MemberExpression(node){}
        // => {"MemberExpression" : function(node){}, ... } というオブジェクト
        const methodObject = getDefinedMethod(rule);
        Object.keys(methodObject).forEach(nodeType => {
            emitter.on(nodeType, methodObject[nodeType]);
        });
    });
    // 2. ASTをtraverseしながら、`Node.type`のイベントを発火する
    traverse(ast, {
        // 1.で登録したNode.typeがあるならここで呼ばれる
        enter: (node) => {
```

```
            emitter.emit(node.type, node);
        },
        leave: (node) => {
            emitter.emit(`${node.type}:exit`, node);
        }
    });
    // 3. ルールから`context.report()`された内容を集めて表示する
    console.log(results.join("\n"));
}
```

Pub/Subパターンを上手く使うことで、ASTを走査するのが一度のみで、それぞれのルールに対してどのようなコードかという情報が `emit` で通知できていることがわかります。

もう少し具体的にするため、実装して動かせるようなものを作ってこの仕組みについて見ていきます。

実装してみよう

今回は、ESLintのルールを解釈できるシンプルなLintの処理を書いてみます。

利用するルールは先ほども出てきたno-console.jsをそのまま使い、このルールを使って同じようにJavaScriptのコードを検証できる `MyLinter` を書いてみます。

MyLinter

MyLinterは単純な2つのメソッドをもつクラスとして実装しました。

- `MyLinter#loadRule(rule): void`
  - 利用するルールを登録する処理
  - `rule` はno-console.jsがexportしたもの
- `MyLinter#lint(code): string[]`
  - `code` を受け取りルールによってLintした結果を返す
  - Lint結果はエラーメッセージの配列とする

実装したものが次のようになっています。

```
import { parse } from "esprima";
import { traverse } from "estraverse";
import { EventEmitter } from "events";

class RuleContext extends EventEmitter {
    report({ message }) {
        this.emit("report", message);
    }
}

export default class MyLinter {
    constructor() {
        this._emitter = new EventEmitter();
        this._ruleContext = new RuleContext();
    }

    loadRule(rule) {
        const ruleExports = rule.create(this._ruleContext);
        // on(nodeType, nodeTypeCallback);
        Object.keys(ruleExports).forEach(nodeType => {
            this._emitter.on(nodeType, ruleExports[nodeType]);
        });
    }


    lint(code) {
```

```
        const messages = [];
        const addMessage = (message) => {
            messages.push(message);
        };
        this._ruleContext.on("report", addMessage);
        const ast = parse(code);
        traverse(ast, {
            enter: (node) => {
                this._emitter.emit(node.type, node);
            },
            leave: (node) => {
                this._emitter.emit(`${node.type}:exit`, node);
            }
        });
        this._ruleContext.removeListener("report", addMessage);
        return messages;
    }
}
```

このMyLinterを使って、`MyLinter#load`でno-console.jsを読み込ませて、

```
function add(x, y){
    console.log(x, y);
    return x + y;
}
add(1, 3);
```

というコードをLintしてみます。

```
import assert from "assert";
import MyLinter from "./MyLinter";
import noConsole from "./no-console";

const linter = new MyLinter();
linter.loadRule(noConsole);
const code = `
function add(x, y){
    console.log(x, y);
    return x + y;
}
add(1, 3);
`;
const results = linter.lint(code);
assert(results.length > 0);
assert.equal(results[0], "Unexpected console statement.");
```

コードには`console`という名前のオブジェクトが含まれているので、 "Unexpected console statement." というエラーメッセージが取得できました。

RuleContext

もう一度、MyLinter.jsを見てみると、`RuleContext`というシンプルなクラスがあることに気づくと思います。

この`RuleContext`はルールから使えるユーティリティメソッドをまとめたものです。今回は`RuleContext#report`というエラーメッセージをルールからMyLinterへ通知するものだけを実装しています。

ルールの実装の方を見てみると、直接オブジェクトをexportしないで、`context`として`RuleContext`のインスタンスを受け取っていることが分かると思います。

```
module.exports = {
    meta: { /* ルールのメタ情報 */ },
    create: function (context) {
```

```
        return {
            "MemberExpression": function (node) {
                if (node.object.name === "console") {
                    context.report({
                        node,
                        message: "Unexpected console statement."
                    });
                }
            }
        };
    }
};
```

このようにして、ルールは `context` という与えられたものだけを使うので、ルールがMyLinter本体の実装の詳細を知らなくても良くなります。

どのような用途に向いている?

このプラグインアーキテクチャはPub/Subパターンを上手く使い、 ESLintのように与えられたコードを読み取ってチェックするような使い方に向いています。

つまり、複数のルールで同時にLintをするというread-onlyなプラグインアーキテクチャとしてはパフォーマンスも期待できると思います。

また、ルールは `context` オブジェクトという与えられたものだけを使うようになっているため、ルールと本体が密結合にはなりにくいです。そのため `context` に何を与えるかを決めることで、ルールができる範囲を制御しやすいといえます。

どのような用途に向いていない?

逆に与えられたコード(AST)を書き換える場合には、ルールを同時に処理を行うためルール間で競合するような変更がある場合に上手く整合性を保つ必要があります。

たとえば、あるルールが書き換えるとコードの位置に更新がかかるため、その後のルールはコードの位置更新の影響を受けます。そのため、コードの書き換えをするにはこの仕組みに加えて、もう1つ抽象レイヤーを設けないと対応は難しいです。

つまり、read-writeなプラグインアーキテクチャとしては、このパターンだけでは難しい部分が出てくると思います。

> ESLint 2.0からautofix、つまり書き換えの機能の導入が導入されています。 ESLintでは、各ルールが書き換える位置や文字列を `fixer` オブジェクトという形で報告し、ESLint本体がルールの順番を考慮して最後に実際の書き換えを行うという抽象レイヤーを設けています。 これはCommand パターンを利用してトランザクション的なふるまいを実現しています。
>
> - Implement autofixing · Issue #3134 · eslint/eslint

この仕組みを使っているもの

- stylelint
  - CSSのLintするツール
- textlint
  - テキストやMarkdownをパースしてASTにしてLintするツール

エコシステム

ESLintのルールはただのJavaScriptモジュールなので、ルール自体をnpmで公開できます。

また、ESLintはデフォルトで有効なルールはありません。そのため、利用する際は設定ファイルを作るか、sindresorhus/xoといったESLintのラッパーを利用する形となります。

ESLint公式の設定として `eslint:recommended` が用意されています。これを `extends` することで推奨の設定を継承できます。

```
{
    "extends": "eslint:recommended"
}
```

これらの設定自体もJavaScriptで表現できるため、設定もnpmで公開して利用できるようになっています。

- Shareable Configs - ESLint - Pluggable JavaScript linter

コーディングルールが多種多様なように、ESLintで必要なルールも個人差があると思います。設定なしで使えると一番楽ですが、設定なしだと誰でも使えるツールにするのは難しいです。それを解消するために柔軟な設定のしくみと設定を共有しやすくしています。

これはPluggable JavaScript linterを表現している仕組みといえるかもしれません。

まとめ

ここではESLintのプラグインアーキテクチャについて学びました。

- ESLintはJavaScriptでルールを書ける
- ASTの木構造を走査しながらPub/Subパターンでチェックする
- ルールは `context` を受け取る以外は本体の実装の詳細を知らなくてよい
- ルールがread-onlyだと簡単で効率的
- read-writeとする場合は気を付ける必要がある
- 設定をJavaScriptで表現できる
- 設定をnpmで共有できる作りになっている

Connect

> この文章はConnect 3.4.0を元に書かれています。

ConnectはNode.jsで動くHTTPサーバーフレームワークです。 middleware という拡張する仕組みを持ち、Connectがもつ機能自体はとても少ないです。

この章ではConnectの middleware の仕組みについて見て行きましょう。

どう書ける?

Connectを使い簡単なEchoサーバを書いてみましょう。 Echoサーバとは、送られてきたリクエストの内容をそのままレスポンスとして返すサーバのことです。

```
import connect from "connect";
import http from "http";
import fetch from "node-fetch";
import assert from "assert";
const app = connect();
// add Error handling
app.use(function (err, req, res, next) {
    console.error(err.stack);
    res.status(500).send(err.message);
    next();
});
// request to response
app.use(function (req, res) {
    req.pipe(res);
});
//create node.js http server and listen on port
const server = http.createServer(app).listen(3000, request);

// request => response
function request() {
    const closeServer = server.close.bind(server);
    const requestBody = {
        "key": "value"
    };
    fetch("http://localhost:3000", {
        method: "POST",
        body: JSON.stringify(requestBody)
    })
        .then(res => res.text())
        .then(text => {
            assert.deepEqual(text, requestBody);
        }).then(closeServer, closeServer);
}
```

このEchoサーバに対して、次のようなリクエストBodyを送信すると、レスポンスとして同じ値が返ってきます。

```
{
    "key": "value"
}
```

`app.use(middleware)` という形で、 middleware と呼ばれる関数には `request` や `response` といったオブジェクトが渡されます。この `request` や `response` を middleware で処理することで、ログを取ったり、任意のレスポンスを返すことができます。

Echoサーバでは `req.pipe(res);` という形でリクエストをそのままレスポンスとして流すことで実現されています。

middlewareをモジュールとして実装

もう少し middleware をプラグインらしくモジュールとして実装したものを見てみます。

次のconnect-example.jsは、あらゆるリクエストに対して、 `"response text"` というレスポンスを `"X-Content-Type-Options"` ヘッダを付けて返すだけのサーバです。

それぞれの処理を middleware としてファイルを分けて実装し、 `app.use(middleware)` で処理を追加しています。

```
function setHeaders(res, headers) {
    Object.keys(headers).forEach(key => {
        const value = headers[key];
        if (value !== null) {
            res.setHeader(key, value);
        }
    });
}
export default function () {
    return function nosniff(req, res, next) {
        setHeaders(res, {
            "X-Content-Type-Options": "nosniff"
        });
        next();
    };
}
```

```
export default function (text) {
    return function hello(req, res) {
        res.end(text);
    };
}
```

```
export default function () {
    return function errorHandling(err, req, res, next) {
        res.writeHead(404);
        res.write(err.message);
        res.end();
        next();
    };
}
```

```
import errorHandler from "./errorHandler";
import hello from "./hello";
import nosniff from "./nosniff";
import assert from "assert";
import connect from "connect";
import http from "http";
import fetch from "node-fetch";

const responseText = "response text";
const app = connect();
// add Error handling
app.use(errorHandler());
// add "X-Content-Type-Options" to response
app.use(nosniff());
// respond to all requests
app.use(hello(responseText));

//create node.js http server and listen on port
const server = http.createServer(app).listen(3000, request);
```

```
function request() {
    const closeServer = server.close.bind(server);
    fetch("http://localhost:3000")
        .then(res => res.text())
        .then(text => {
            assert.equal(text, responseText);
            server.close();
        })
        .catch(console.error.bind(console))
        .then(closeServer, closeServer);
}
```

基本的にどの middleware も `app.use(middleware)` という形で拡張でき、モジュールとして実装すれば再利用もしやすい形となっています。

> middleware となる関数の引数が4つであると、それはエラーハンドリングの middleware とするという、Connect独自のルールがあります。

どのような仕組み?

Connectの middleware がどのような仕組みで動いているのかを見ていきます。

`app` に登録した middleware は、リクエスト時に呼び出されています。そのため、 `app` のどこかに利用する middleware を保持していることは推測できると思います。

Connectでは `app.stack` に middleware を配列として保持しています。次のようにして `app.stack` の中身を表示してみると、 middleware が登録順で保持されていることがわかります。

```
import errorHandler from "./errorHandler";
import hello from "./hello";
import nosniff from "./nosniff";
import connect from "connect";

const responseText = "response text";
const app = connect();
// add Error handling
app.use(errorHandler());
// add "X-Content-Type-Options" to response
app.use(nosniff());
// respond to all requests
app.use(hello(responseText));

// print middleware list
app.stack.map(({handle}) => console.log(handle));
/* =>
 [Function: errorHandling]
 [Function: nosniff]
 [Function: hello]
*/
```

Connectは登録された middleware を、サーバがリクエストを受け取りそれぞれ順番に呼び出しています。

上記の例だと次の順番で middleware が呼び出されることになります。

- nosniff
- hello
- errorHandler

エラーハンドリングの middleware は処理中にエラーが起きた時のみ呼ばれます。

そのため、通常は nosniff.js → hello.js の順で呼び出されます。

```
function setHeaders(res, headers) {
    Object.keys(headers).forEach(key => {
        const value = headers[key];
        if (value !== null) {
            res.setHeader(key, value);
        }
    });
}
export default function () {
    return function nosniff(req, res, next) {
        setHeaders(res, {
            "X-Content-Type-Options": "nosniff"
        });
        next();
    };
}
```

`nosniff.js` は、HTTPヘッダを設定し終わったら `next()` を呼び出し、この `next()` が次の middleware へ行くという意味になります。

次に、 `hello.js` を見てみると、 `next()` がありません。

```
export default function (text) {
    return function hello(req, res) {
        res.end(text);
    };
}
```

`next()` がないということは `hello.js` がこの連続する middleware の最後となっていることがわかります。仮に、これより先に middleware が登録されていたとしても無視されます。

つまり、処理的には次のようにstackを先頭から一個づつ取り出し、処理していくという方法が取られています。

Connectの行っている処理を抽象的なコードで書くと次のような形になっています。

```
const req = "...",
    res = "...";
function next(){
    const middleware = app.stack.shift();
    // nextが呼ばれれば次のmiddleware
    middleware(req, res, next);
}
next();// 初回
```

このような middleware を繋げたものをmiddleware stackと呼ぶことがあります。

middleware stack で構成されるHTTPサーバとして、PythonのWSGI middlewareやRubyのRackなどがあります。ConnectはRackと同じく `use` で middleware を指定することからも分かりますが、 Rackを参考にした実装となっています。

- Ruby - Rack解説 - Rackの構造とRack DSL - Qiita

次は、先ほど抽象的なコードとなっていたものを具体的な実装にしながら見ていきます。

実装してみよう

Connectライクな middleware をサポートしたJunctionというクラスを作成してみます。

Junctionは、`use(middleware)` と `process(value, (error, result) => { });` を持っているシンプルなクラスです。

```
function isErrorHandingMiddleware(middleware) {
    // middleware(error, text, next)
    const arity = middleware.length;
    return arity === 3;
}
function applyMiddleware(error, response, middleware, next) {
    let errorOnMiddleware = null;
    try {
        if (error && isErrorHandingMiddleware(middleware)) {
            middleware(error, response, next);
        } else {
            middleware(response, next);
        }
        return;
    } catch (error) {
        errorOnMiddleware = error;
    }
    // skip the middleware or Error on the middleware
    next(errorOnMiddleware, response);
}

export default class Junction {
    constructor() {
        this.stack = [];
    }

    use(middleware) {
        this.stack.push(middleware);
    }

    process(initialValue, callback) {
        const response = {value: initialValue};
        const next = (error) => {
            const middleware = this.stack.shift();
            if (!middleware) {
                return callback(error, response);
            }
            applyMiddleware(error, response, middleware, next);
        };
        next();
    }
}
```

実装を見てみると、`use` で middleware を登録して、`process` で登録した middleware を順番に実行していきます。そのため、`Junction` 自体は渡されたデータの処理をせずに、 middleware の中継のみをしています。

登録する middleware はConnectと同じもので、処理をしたら `next` を呼んで、次の middleware が処理するというのを繰り返しています。

使い方はConnectと引数の違いはありますが、ほぼ同じような形で利用できます。

```
import Junction from "./junction";
import assert from "assert";
const junction = new Junction();
junction.use(function toUpperCase(res, next) {
    res.value = res.value.toUpperCase();
    next();
});
junction.use(function exclamationMark(res, next) {
    res.value = res.value + "!";
    next();
});
junction.use(function errorHandling(error, res, next) {
    console.error(error.stack);
```

```
    next();
});

const text = "hello world";
junction.process(text, function (error, result) {
    if (error) {
        console.error(error);
    }
    const value = result.value;
    assert.equal(value, "HELLO WORLD!");
});
```

どのような用途に向いている?

ConnectやJunctionの実装を見てみると分かりますが、このアーキテクチャでは機能の詳細を middleware で実装できます。そのため、本体の実装は middleware に提供するインタフェースの決定、エラーハンドリングの手段を提供するだけでとても小さいものとなっています。

今回は紹介していませんが、Connectにはルーティングに関する機能があります。しかし、この機能も「与えられたパスにマッチした場合のみに反応する middleware を登録する」という単純なものです。

```
app.use("/foo", function fooMiddleware(req, res, next) {
    // req.url starts with "/foo"
    next();
});
```

このアーキテクチャは、入力と出力がある場合にコアとなる部分は小さく実装できることが分かります。

そのため、ConnectやRackなどのHTTPサーバでは「リクエストに対してレスポンスを返す」というのが決まっているので、このアーキテクチャは適しています。

どのような用途に向いていない?

このアーキテクチャでは機能の詳細が middleware で実装できます。しかし、多くの機能を middleware で実装していくと、 middleware 間に依存関係を作ってしまうことがあります。

この場合、`use(middleware)` で登録する順番により挙動が変わるため、利用者が middleware 間の依存関係を解決する必要があります。

そのため、プラグイン同士の強い独立性や明確な依存関係を扱いたい場合には不向きといえるでしょう。

これらを解消するためにコアはそのままにして、最初から幾つかのmiddleware stackを作ったものが提供されるケースもあります。

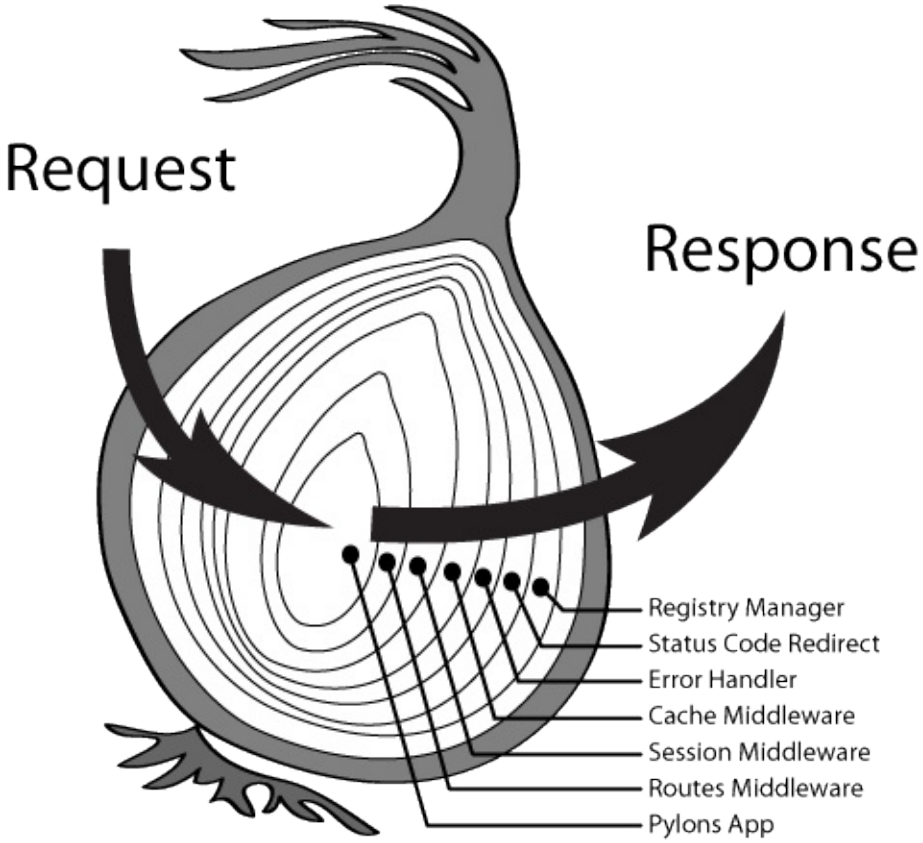
エコシステム

Connect自体の機能は少ないですが、その分 middleware の種類が多くあります。

- github.com/senchalabs/connect#middleware
- Express middleware

また、それぞれの middleware が小さな単機能となっていて、それを組み合わせて使うように作られているケースが多いです。

これは、 middleware が層として重なっている作り、つまり middleware stack の形を取ることが多いためだといえます。



> ミドルウェアでラップするプロセスは、概念的にたまねぎの中の層と同様の構造をもたらします。 WSGI ミドルウェアより引用

この仕組みを使っているもの

- Express
  - Connectと middleware の互換性がある
  - 元々はConnectを利用していたが4.0.0で自前の実装に変更
- wooorm/retext
  - `use` でプラグイン登録していくテキスト処理ライブラリ
- r7kamura/stackable-fetcher
  - `use` でプラグイン登録して処理を追加できるHTTPクライアントライブラリ

まとめ

ここではConnectのプラグインアーキテクチャについて学びました。

- Connectは middleware を使ったHTTPサーバーライブラリである
- Connect自体の機能は少ない
- 複数の middleware を組み合わせることでHTTPサーバを作ることができる

参考資料

- Ruby - Rack解説 - Rackの構造とRack DSL - Qiita
- Pylons のコンセプト — Pylons 0.9.7 documentation

gulp

> この文章はgulp 3.9.0を元に書かれています。

gulpはNode.jsを使ったタスク自動化ツールです。ビルドやテストなどといったタスクを実行するためのツールで、それぞれのタスクをJavaScriptで書くことができるようになっています。

ここでいうタスクとは複数の処理からなる処理の固まりのことです。このタスクを定義するAPIとして `gulp.task` が用意されています。また、それぞれの処理はNode.jsのStreamを使いつなげることで、複数の処理を一時ファイルなしでできるようになっています。

それぞれの処理はgulpのプラグインという形でモジュール化されているため、利用者はモジュールを読み込み、 `pipe()` で繋ぐだけでタスクの定義ができるツールとなっています。

どう書ける?

たとえば、Sassで書いたファイルを次のように処理したいとします。

1. `sass/*.scss` のファイルを読み込む
2. 読み込んだsassファイルを `sass` でコンパイル
3. CSSとなったファイルに `autoprefixture` で接頭辞をつける
4. CSSファイルをそれぞれ `minify` で圧縮する
5. 圧縮したCSSファイルをそれぞれ `css` ディレクトリに出力する

この一連の処理は次のようなタスクとして定義できます。

```
import gulp from "gulp";
import sass from "gulp-sass";
import autoprefixer from "gulp-autoprefixer";
import minify from "gulp-minify-css";

gulp.task("sass", function() {
    return gulp.src("sass/*.scss")
        .pipe(sass())
        .pipe(autoprefixer())
        .pipe(minify())
        .pipe(gulp.dest("css"));
});
```

ここでは、gulpプラグインの仕組みについて扱うので、gulpの使い方について詳しくは以下を参照してください。

- gulp/docs at master · gulpjs/gulp
- 現場で使えるgulp入門 - gulpとは何か | CodeGrid
- gulp入門 (全12回) - プログラミングならドットインストール

どのような仕組み?

実際にgulpプラグインを書きながら、どのような仕組みで処理同士が連携を取り動作しているのかを見ていきましょう。

先ほどのgulpタスクの例では、すでにモジュール化された処理を `pipe` で繋げただけで、それぞれの処理がどのように実装されているかはよく分かりませんでした。

ここでは `gulp-prefixer` というgulpプラグインを書いていきます。 `gulp-prefixer` は与えられたそれぞれのファイルに対して先頭に特定の文字列の追加するプラグインです。

同様の名前のプラグインが公式のドキュメントで「プラグインの書き方」の例として紹介されているので合わせて見るとよいでしょう。

- gulp/docs/writing-a-plugin
- gulp/dealing-with-streams.md

多くのgulpプラグインはオプションを受け取り、NodeのStreamを返す関数として実装されます。

```
import {Transform} from "stream";
export function prefixBuffer(buffer, prefix) {
    return Buffer.concat([Buffer(prefix), buffer]);
}

export function prefixStream(prefix) {
    return new Transform({
        transform: function (chunk, encoding, next) {
            const buffer = prefixBuffer(chunk, prefix);
            this.push(buffer);
            next();
        }
    });
}

const gulpPrefixer = function (prefix) {
    // enable `objectMode` of the stream for vinyl File objects.
    return new Transform({
        // Takes in vinyl File objects
        writableObjectMode: true,
        // Outputs vinyl File objects
        readableObjectMode: true,
        transform: function (file, encoding, next) {
            if (file.isBuffer()) {
                file.contents = prefixBuffer(file.contents, prefix);
            }

            if (file.isStream()) {
                file.contents = file.contents.pipe(prefixStream(prefix));
            }
            this.push(file);
            next();
        }
    });
};

export default gulpPrefixer;
```

ここで実装した `gulp-prefixer` は、次のようにしてタスクに組み込むことができます。

```
import gulp from "gulp";
import gulpPrefixer from "./gulp-prefixer";
gulp.task("default", function () {
    return gulp.src("./*.*")
        .pipe(gulpPrefixer("prefix text"))
        .pipe(gulp.dest("build"))
        .on("error", (error) => {
            console.error(error);
        });
});
```

この `default` タスクは次のような処理が行われます。

1. `./*.*` にマッチするファイルを取得(すべてのファイル)
2. 取得したファイルの先頭に"prefix text"という文字列を追加する
3. 変更したファイルを `build/` ディレクトリに出力する

Stream

gulp-prefixer.jsを見てみると、 `gulpPrefixer` というTransform Streamのインスタンスを返していることが分かります。

```
const gulpPrefixer = function (prefix) {
    // enable `objectMode` of the stream for vinyl File objects.
    return new Transform({
        // Takes in vinyl File objects
        writableObjectMode: true,
        // Outputs vinyl File objects
        readableObjectMode: true,
        transform: function (file, encoding, next) {
            if (file.isBuffer()) {
                file.contents = prefixBuffer(file.contents, prefix);
            }

            if (file.isStream()) {
                file.contents = file.contents.pipe(prefixStream(prefix));
            }
            this.push(file);
            next();
        }
    });
};

export default gulpPrefixer;
```

Transform Streamというものが出てきましたが、Node.jsのStreamは次の4種類があります。

- Readable Stream
- Transform Stream
- Writable Stream
- Duplex Stream

今回の `default` タスクの処理をそれぞれ当てはめると次のようになっています。

1. `./*.*` にマッチするファイルを取得 = Readable Stream
2. 取得したファイルの先頭に"prefix text"という文字列を追加する = Transform Stream
3. 変更したファイルを `build/` ディレクトリに出力する = Writable Stream

あるファイルを Read して、 Transform したものを、別のところに Write としているというよくあるデータの流れといえます。

gulp-prefixer.jsでは、gulpから流れてきたデータをStreamで受け取り、そのデータを変更したもの次へ渡すTransform Streamとなっています。

「gulpから流れてきたデータ」を扱うために `readableObjectMode` と `writableObjectMode` をそれぞれ `true` にしています。この ObjectMode というのは名前のとおり、Streamでオブジェクトを流すための設定です。

通常のNode.js StreamはBufferというバイナリーデータを扱います。このBufferはStringと相互変換が可能できます。しかし、一方で複数の値を持ったオブジェクトのようなものは扱えません。

そのため、Node.js StreamにはObject Modeがあり、有効の場合はBufferやString以外のJavaScriptオブジェクトをStreamで流せるようになっています。

Node.js Streamについては以下を合わせて参照するといいでしょう。

- Stream Node.js Manual & Documentation
- substack/stream-handbook

vinyl

gulpではvinylオブジェクトがStreamで流れてきます。このvinylは Virtual file format という呼ばれているもので、ファイル情報と中身をラップしたgulp用に作成された抽象フォーマットです。

なぜこういった抽象フォーマットが必要なのかは次のことを考えてみると分かりやすいです。

- Streamで流れてきたデータの拡張子を知りたい
- Streamで流れてきたデータの読み取り属性をチェックしたい
- Streamで流れてきたデータと同じ場所にファイルを書き出したい

ファイルの中身だけがStreamで流れた場合は、ファイルのパスや読み取り属性などの詳細な情報を知ることができません。そのため、`gulp.src`で読み込んだファイルはvinylでラップされ、ファイルの中身は`contents`として参照できるようになっています。

vinylの中身を処理する

次はTransform Streamの具体的な処理を見てみましょう。

```
// file は `vinyl` オブジェクト
if (file.isBuffer()) {
    file.contents = prefixBuffer(file.contents, prefix);
}

if (file.isStream()) {
    file.contents = file.contents.pipe(prefixStream(prefix));
}
```

`vinyl`抽象フォーマットの`contents`プロパティには、読み込んだファイルのBufferまたはStreamが格納されています。そのため両方のパターンに対応したコードする場合はどちらが来ても問題ないように書く必要があります。

> : gulp pluginは必ずしも両方のパターンに対応しないといけないのではなく、Bufferだけに対応したものも多いです。しかし、その場合にStreamが来た時のErrorイベントを通知することがガイドラインで推奨されています。 - gulp/guidelines.md at master · gulpjs/gulp

`contents`にどちらのタイプが格納されているかは、ひとつ前のStreamで決定されます。

```
gulp.src("./*.*")
    .pipe(gulpPrefixer("prefix text"))
    .pipe(gulp.dest("build"));
```

この場合は、`gulp.src`により決定されます。`gulp.src`はデフォルトでは、`contents`にBufferを格納するので、この場合はBufferで処理されることになります。

`gulp.src`はオプションに`{ buffer: false }`を渡すことで`contents`にStreamを流すことも可能です。

```
gulp.src("./*.*", { buffer: false })
    .pipe(gulpPrefixer("prefix text"))
    .pipe(gulp.dest("build"));
```

変換処理

最後にBufferとStreamそれぞれの変換処理を見てみます。

```
export function prefixBuffer(buffer, prefix) {
    return Buffer.concat([Buffer(prefix), buffer]);
}

export function prefixStream(prefix) {
    return new Transform({
        transform: function (chunk, encoding, next) {
            // ObjectMode:falseのTransform Stream
            // StreamのchunkにはBufferが流れてくる
            const buffer = prefixBuffer(chunk, prefix);
            this.push(buffer);
            next();
        }
    });
}
```

やってきたBufferの先頭に`prefix`の文字列をBufferとして結合して返すだけの処理が行われています。

この変換処理はgulpに依存したものではないため、通常のライブラリに渡して処理するということが可能です。BufferはStringと相互変換が可能なので、多くのgulpプラグインと呼ばれるものは、`gulpPrefixer`と`prefixBuffer`にあたる部分だけを実装しています。

つまり、prefixを付けるといった変換処理は、既存のライブラリで行うことができるようになっています。

gulpプラグインはvinylオブジェクトのデータをプラグイン同士でやり取りし、そのインタフェースとして既存のNode.js Streamを使っているといえます。

エコシステム

gulpのプラグインが行う処理は「入力に対して出力を返す」が主となっています。この受け渡すデータとしてvinylオブジェクトを使い、受け渡すAPIのインタフェースとしてNode.js Streamを使っています。

gulpではプラグインがもつ機能は1つ(単機能)とすることを推奨しています。

> Your plugin should only do one thing, and do it well. -- gulp/guidelines.md

gulpは既存のNode.js Streamに乗ることで独自のAPIを使わずに解決しています。

元々、Transform Streamは1つの変換処理を行うことが得意なので、その変換処理を`pipe`を繋げることで複数の処理を行うことができます。

また、gulpはタスク自動化ツールなので、既存のライブラリをそのままタスクとして使いやすくすることが重要だといえます。 Node.js Streamのデフォルトでは流れるデータが`Buffer`となり、そのままでは既存のライブラリでは扱いにくい問題をデータとしてvinylオブジェクトを流すことで緩和しています。

このようにして、gulpはタスクに必要な単機能のプラグインを既存のライブラリで作りやすくしています。これにより再利用できるプラグインが多くできることでエコシステムを構築しているといえます。

どのような用途に向いている?

gulp自体はデータの流れを管理するだけで、タスクを実現するためにはプラグインが重要になります。タスクにはさまざまな処理が想定されるため、必要になるプラグインも種類がさまざまなものとなります。

gulpではvinylオブジェクトを中間フォーマットと決めたことで、既存のライブラリをラップしただけのプラグインが作りやすくなっています。

またgulpは、Gruntとは異なり、タスクをJavaScriptのコードして表現します。これにより、プラグインの組み合わせだけだと実現できない場合に、直接コードを書くことで対応するといった対処法を取ることができます。

そのため、プラグインの行う処理が予測できない場合に、中間フォーマットとデータの流し方だけを決めるというやり方は向いています。

まとめると

- 既存のライブラリをプラグイン化しやすい
- 必要なプラグインがない場合も、設定としてコードを書くことで対応できる

どのような用途に向いていない?

プラグインを複数組み合わせ扱うものに共通することですが、プラグインの組み合わせ問題はgulpでも発生します。

たとえば、BrowserifyはNode.js Streamを扱えますが、変換の開始点としていない場合に問題が発生します。

- gulp/browserify-transforms.md at master · gulpjs/gulp

また、gulpは単機能のプラグインを推奨していますが、これはAPIとしてそういう制限があるわけではないためあくまでルールとなっています。

このような問題に対してgulpはガイドラインやレシピといったドキュメントを充実させることで対処しています。

- gulp/docs at master · gulpjs/gulp

既存のライブラリをプラグイン化しやすい一方、プラグインとライブラリのオプションが異なったり、利用者はプラグイン化したライブラリの扱い方を学ぶ必要があります。

ライブラリとプラグインの作者が異なるケースも多いため、同様の機能をもつプラグインが複数できたり、質もバラバラとなりやすいです。

まとめると

- プラグインの組み合わせ問題は利用者が解決しないといけない
- 同様の機能をもつプラグインが生まれやすい

この仕組みを使っているもの

- sighjs/sigh
  - gulpプラグインそのものをサポートしています。

まとめ

ここではgulpのプラグインアーキテクチャについて学びました。

Redux

> この文章はRedux 3.5.2を元に書かれています。

ReduxはJavaScriptアプリケーションのStateを管理するライブラリで、 Reactなどと組み合わせアプリケーションを作成するために利用されています。

ReduxはFluxアーキテクチャに類似する仕組みです。そのため、事前にFluxについて学習しているとよいです。

ReduxにはThree Principles(以下、三原則)と呼ばれる3つの制約の上で成立しています。

- Single source of truth
  - アプリケーション全体のStateは1つのStateツリーとして保存される
- State is read-only
  - StateはActionを経由しないと書き換えることができない
- Changes are made with pure functions
  - Actionを受け取りStateを書き換えるReducerと呼ばれるpure functionを作る

この三原則についての詳細はドキュメントなどを参照してください。

- Read Me | Redux
- Getting Started with Redux - Course by @dan_abramov @eggheadio

Reduxの使い方についてはここでは解説しませんが、Reduxの拡張機能となる middleware も、この三原則に基づいた仕組みとなっています。

middleware という名前からも分かるように、Connectの仕組みと類似点があります。 Connectの違いを意識しながら、Reduxの middleware の仕組みを見ていきましょう。

どう書ける?

簡潔にReduxの仕組みを書くと次のようになります。

- 操作を表現するオブジェクトをActionと呼ぶ
  - 一般的なコマンドパターンのコマンドと同様のもの
- Actionを受け取りStateを書き換える関数を Reducer と呼ぶ
  - ReducerはStoreへ事前に登録する
- ActionをDispatch(`store.dispatch(action)`)することで、ActionをReducerへ通知する

Reduxの例として次のようなコードを見てみます。

```
import {createStore, applyMiddleware} from "redux";
import createLogger from "./logger";
import timestamp from "./timestamp";
// 4. Actionを受け取り新しいStateを返すReducer関数
const reducer = (state = {}, action) => {
    switch (action.type) {
    case "AddTodo":
        return Object.assign({}, state, {title: action.title});
    default:
        return state;
    }
};
// 1. `logger`と`crashReporter`のmiddlewareを適用した`createStore`関数を作る
const createStoreWithMiddleware = applyMiddleware(createLogger(), timestamp)(createStore);

// 2. Reducerを登録したStoreを作成
```

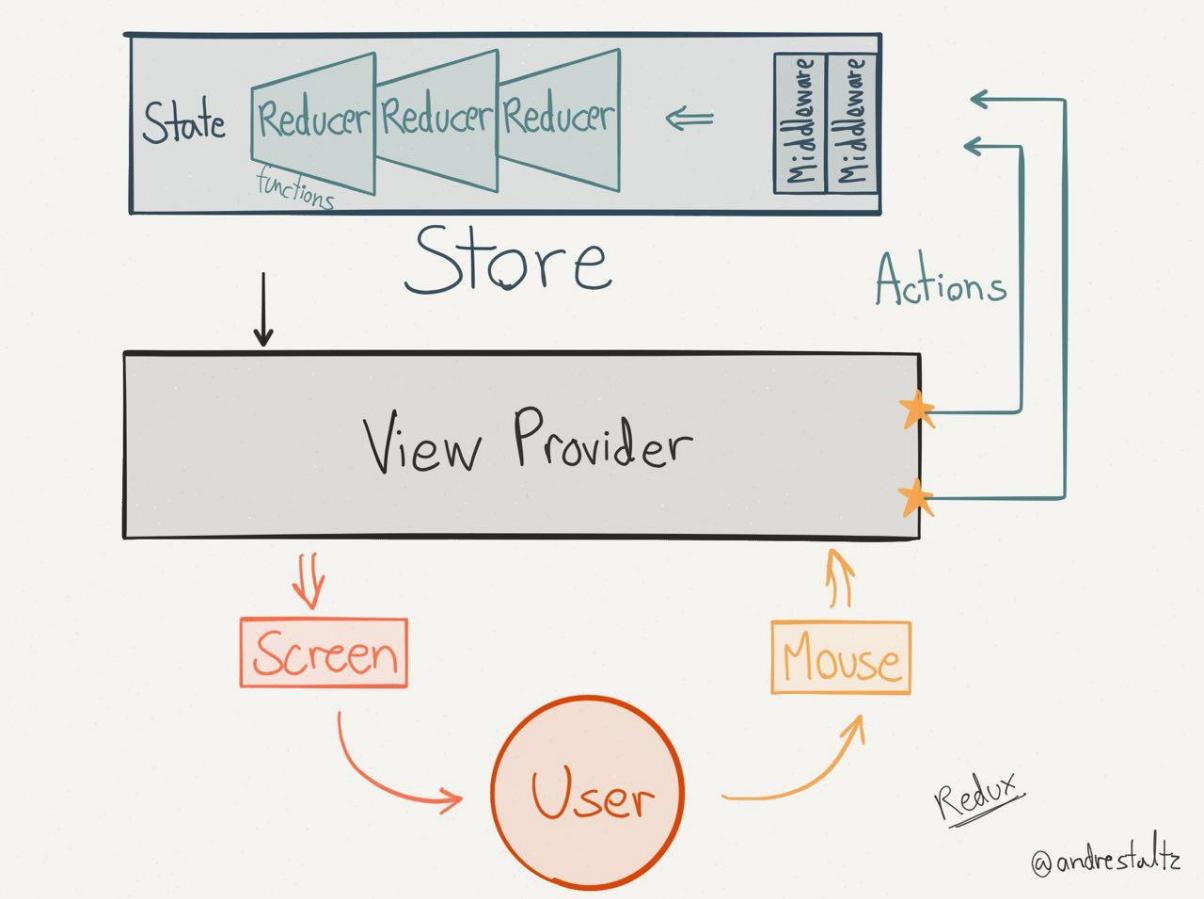
```
const store = createStoreWithMiddleware(reducer);

store.subscribe(() => {
    // 5. Stateが変更されたら呼ばれる
    const state = store.getState();
    // 現在のStateを取得
    console.log(state);
});
// 3. Storeの変更をするActionをdispatch
store.dispatch({
    type: "AddTodo",
    title: "Todo title"
});
```

1. `logger` と `crashReporter` のmiddlewareを適用した `createStore` 関数を作る
2. Reducerを登録したStoreを作成
3. (Storeの変更をする)Actionをdispatch
4. Actionを受け取り新しいStateを返すReducer関数
5. Stateが変更されたら呼ばれる

というような流れで動作します。

上記の処理のうち、 3から4の間が middleware が処理する場所となっています。

```
dispatch(action) -> (_middleware_ の処理) -> reducerにより新しいStateの作成 -> (Stateが変わったら) -> subscribeで登録したコールバックを呼ぶ
```



via staltz.com/unidirectional-user-interface-architectures.htmlo

次は middleware によりどのような拡張ができるのかを見ていきます。

middleware

Reduxでは第三者が拡張できる仕組みを middleware と呼んでいます。

- Middleware | Redux

どのような拡張を middleware で書けるのか、実際の例を見てみます。次の middleware はStoreがdispatchしたActionと、その前後でStateにどのような変更があったのかを出力するロガーです。

```
// LICENSE : MIT
const defaultOptions = {
    // default: logger use console API
    logger: console
};
/**
 * create logger middleware
 * @param {{logger: *}} options
 * @returns {Function} middleware function
 */
export default function createLogger(options = defaultOptions) {
    const logger = options.logger || defaultOptions.logger;
    return store => next => action => {
        logger.log(action);
        const value = next(action);
        logger.log(store.getState());
        return value;
    };
}
```

この middleware は次のようにReduxに対して適用できます。

```
import {createStore, applyMiddleware} from "redux";
const createStoreWithMiddleware = applyMiddleware(createLogger())(createStore);
```

このとき、見た目上は `store` に対して middleware が適用されているように見えますが、実際には `store.dispatch` に対して適用され、拡張された `dispatch` メソッドが作成されています。

これにより、 `dispatch` を実行する際に middleware の処理を挟むことができます。これがReduxの middleware による拡張ポイントになっています。

```
store.dispatch({
    type: "AddTodo",
    title: "Todo title"
});
```

先ほどの `logger.js` をもう一度見てみます。

```
export default function createLogger(options = defaultOptions) {
    const logger = options.logger || defaultOptions.logger;
    return store => next => action => {
        logger.log(action);
        const value = next(action);
        logger.log(store.getState());
        return value;
    };
}
```

`createLogger` は、loggerにオプションを渡すためのものなので置いておき、 `return` している高階関数の連なりが middleware の本体となります。

```
const middleware = store => next => action => {};
```

上記のArrowFunctionの連なりが一見すると何をしているのかが分かりにくいですが、これは次のように展開できます。

```
const middleware = (store) => {
    return (next) => {
        return (action) => {
            // Middlewareの処理
        };
    };
};
```

ただ単に関数を返す関数(高階関数)を作っているだけだと分かります。

これを踏まえて `logger.js` をもう一度見てみると、 `next(action)` の前後にログ表示を挟んでいることが分かります。

```
// LICENSE : MIT
const defaultOptions = {
    // default: logger use console API
    logger: console
};
/**
 * create logger middleware
 * @param {{logger: *}} options
 * @returns {Function} middleware function
 */
export default function createLogger(options = defaultOptions) {
    const logger = options.logger || defaultOptions.logger;
    return store => next => action => {
        logger.log(action);
        const value = next(action);
        logger.log(store.getState());
        return value;
    };
}
```

この middleware は次のようなイメージで動作します。

```
// dispatch action
const action = {type: "ActionType"};
dispatch(action);
//        |
//        |
//        v
// logger middleware
//                | "action"
//                v
store => next => action => {
    logger.log(action);
    const value = next(action);
    // next is store.dispatch
    logger.log(store.getState());
    return value;
}
//        |
//        |
//        v
store.dispatch(action);
```

この場合の `next` は `dispatch` と言い換えても問題ありませんが、複数の middleware を適用した場合は、 次の middleware を呼び出すということを表現しています。

Reduxの middleware の仕組みは単純ですが、見慣れないデザインなので複雑に見えます。実際に同じ仕組みを実装しながら、Reduxの middleware について学んでいきましょう。

どのような仕組み?

middleware は `dispatch` をラップする処理ですが、そもそも `dispatch` とはどういうことをしているのでしょうか?

簡潔に書くと、Reduxの `store.dispatch(action)` は `store.subscribe(callback)` で登録した `callback` に `action` を渡し呼び出すだけです。

これはよくみるPub/Subのパターンですが、今回はこのPub/Subパターンの実装からみていきましょう。

Dispatcher

ESLintと同様でEventEmitterを使い、 `dispatch` と `subscribe` をもつ `Dispatcher` を実装すると次のようになります。

```
const EventEmitter = require("events");
export const ON_DISPATCH = "__ON_DISPATCH__";
```

```
/**
 * The action object that must have `type` property.
 * @typedef {Object} Action
 * @property {String} type The event type to dispatch.
 * @public
 */
export default class Dispatcher extends EventEmitter {
    /**
     * subscribe `dispatch` and call handler. it return release function
     * @param {function(Action)} actionHandler
     * @returns {Function} call the function and release handler
     */
    subscribe(actionHandler) {
        this.on(ON_DISPATCH, actionHandler);
        return this.removeListener.bind(this, ON_DISPATCH, actionHandler);
    }

    /**
     * dispatch action object.
     * @param {Action} action
     */
    dispatch(action) {
        this.emit(ON_DISPATCH, action);
    }
}
```

`Dispatcher` はActionオブジェクトを `dispatch` すると、 `subscribe` で登録されていたコールバック関数を呼び出すという単純なものです。

また、この `Dispatcher` の実装はReduxのものとは異なるので、あくまで理解のための参考実装です。

> Unlike Flux, Redux does not have the concept of a Dispatcher. This is because it relies on pure functions instead of event emitters -- Prior Art | Redux

applyMiddleware

次に、 middleware を適用する処理となる `applyMiddleware` を実装していきます。先ほども書いたように、middleware は `dispatch` を拡張する仕組みです。

`applyMiddleware` は `dispatch` と middleware を受け取り、 middleware で拡張した `dispatch` を返す関数です。

```
/*
   => api - middleware api
       => next - next/dispatch function
          => action - action object

 */
const applyMiddleware = (...middlewares) => {
    return middlewareAPI => {
        const originalDispatch = (action) => {
            middlewareAPI.dispatch(action);
        };
        // `api` is middlewareAPI
        const wrapMiddleware = middlewares.map(middleware => {
            return middleware(middlewareAPI);
        });
        // apply middleware order by first
        const last = wrapMiddleware[wrapMiddleware.length - 1];
        const rest = wrapMiddleware.slice(0, -1);
        const roundDispatch = rest.reduceRight((oneMiddle, middleware) => {
            return middleware(oneMiddle);
        }, last);
        return roundDispatch(originalDispatch);
    };
};
```

```
export default applyMiddleware;
```

この `applyMiddleware` はReduxのものと同じなので、次のように middleware を適用した `dispatch` 関数を作成できます。

```
import Dispatcher from "./Dispatcher";
import applyMiddleware from "./apply-middleware";
import timestamp from "./timestamp";
import createLogger from "./logger";
const dispatcher = new Dispatcher();
dispatcher.subscribe(action => {
    console.log(action);
    /*
    { timeStamp: 1463742440479, type: 'FOO' }
     */
});
// Redux compatible middleware API
const state = {};
const middlewareAPI = {
    getState(){
        // shallow-copy state
        return Object.assign({}, state);
    },
    dispatch(action){
        dispatcher.dispatch(action);
    }
};
// create `dispatch` function that wrapped with middleware
const dispatchWithMiddleware = applyMiddleware(createLogger(), timestamp)(middlewareAPI);
dispatchWithMiddleware({type: "FOO"});
```

`applyMiddleware` で `timestamp` をActionに付加する middleware を適用しています。これにより `dispatchWithMiddleware(action)` した `action` には自動的に `timestamp` プロパティが追加されています。

```
const dispatchWithMiddleware = applyMiddleware(createLogger(), timestamp)(middlewareAPI);
dispatchWithMiddleware({type: "FOO"});
```

ここで middleware には `middlewareAPI` として定義した2つのメソッドをもつオブジェクトが渡されています。しかし、 `getState` は読み込みのみで、middlewareにはStateを直接書き換える手段が用意されていません。また、もう1つの `dispatch` もActionオブジェクトを書き換えられますが、結局できることは `dispatch` するだけです。

このことから middleware にも三原則が適用されていることが分かります。

- State is read-only
  - StateはActionを経由しないと書き換えることができない

middleware という仕組み自体はConnectと似ています。しかし、 middleware が直接的に結果(State)を直接書き換えることはできません。

Connectの middleware は最終的な結果(`response`)を書き換えできます。一方、Reduxの middleware は扱える範囲が「 `dispatch` からReducerまで」と線引されている違いといえます。

どういうことに向いている?

Reduxの middleware そのものも三原則に基づいた仕組みとなっています。 middleware はActionオブジェクトを自由に書き換えたり、Actionを無視したりできます。一方、Stateを直接は書き換えることができません。

多くのプラグインの仕組みでは、プラグインに特権的な機能を与えていることが多いですが、 Reduxの middleware は書き込みのような特権的な要素も制限されています。

middleware に与えられている特権的なAPIとしては、`getState()` と `dispatch()` ですが、どちらも書き込みをするようなAPIではありません。

このように、プラグインに対して一定の権限をもつAPIを与えつつ、原則を壊すような特権を与えないことを目的としている場合に向いています。

どういうことに向いていない?

一方、プラグインにも書き込み権限を与えないためには、プラグイン間でやり取りする中間的なデータが必要になります。

ReduxではActionオブジェクトというような命令(コマンド)を表現したオブジェクトに対して、 Reducerという命令を元に新しいStateを作り出す仕組みを設けていました。

つまり、プラグインそのものだけですべての処理が完結するわけではありません。プラグインで処理した結果を受け取り、その結果を処理する実装も同時に必要となっています。 Reduxでは middleware を前提とした処理を実装として書くことも多いです。

そういう意味ではプラグインと実装が密接といえるかもしれません。

そのため、プラグインのみで全処理が完結するような機能を作る仕組みは向いていません。

まとめ

ここではReduxのプラグインアーキテクチャについて学びました。

- Reduxの middleware はActionオブジェクトに対する処理を書ける
- middleware に対しても三原則が適用されている
- middleware に対しても扱える機能の制限を適用しやすい
- middleware のみですべての処理が完結するわけではない

参考

- Middleware | Redux
- 10. Middleware · happypoulp/redux-tutorial Wiki
- Brian Troncone – Redux Middleware: Behind the Scenes
- ReduxのMiddlewareについて理解したいマン | moxt
- Understanding Redux Middleware — Medium

Contribute

Installation

インストールにはNode.jsが必要です

```
git clone https://github.com/azu/JavaScript-Plugin-Architecture.git
cd JavaScript-Plugin-Architecture
npm install
```

Usage

この書籍はHonKitを使い書かれています。

表示の確認

`npm start`でHonKitのローカルサーバを立ち上げて表示を確認できます。

```
npm start
```

テスト

`npm test`でコードや文章の単語チェックを行えます

```
npm test
```

文章カバレッジ

textlintとtextlint-formatter-codecovを使って出してる文章に対するカバレッジ

100%を理想的目標として、それに対する現実的な値をカバレッジの%として表現しています。

- https://codecov.io/github/azu/JavaScript-Plugin-Architecture?branch=master

現在の文章カバレッジは次のコマンドでも確認できます。

```
npm run textlint:coverage
```

Contributeのやりかた

この書籍ではJavaScriptにおけるプラグインアーキテクチャを色々なライブラリやツールを元に紹介する形をとっています。

Contributeは大きく分けて、既存の文章の修正や執筆とProposalの提案などがあります。

文章の修正

typoなどを見つけた場合は、1文字の修正からでも問題無いので、Pull Requestを送っていただけると助かります。

表記揺れを発見した場合は単純にIssueを立ててもらうか、Pull Requestでの修正をいただけると嬉しいです。

また、この書籍ではtest/prh-rule.yamlで定義した辞書を使い表記揺れを辞書でテストできるようにしています。辞書による表記揺れの検知が可能なら、そちらも合わせてご指摘いただけるとありがたいです。

- textlint + prhで表記ゆれを検出する | Web Scratch

新しいプラグインの仕組みを書く

この書籍に載せたいプラグインアーキテクチャがある場合は、Issueを立ててください。

たとえば、XXXというライブラリ/ツールのアーキテクチャについてのIssueを立てる場合、次のようなことが1行とかでもいいので書かれているとよさそうです。

仕組みについて調べるのが大変な場合は、あとで調べれば問題ないため空欄で問題ありません。 JavaScriptはとにかく柔軟な言語なので、こういうプラグインの形式を取ってるというのを知らせるだけでも有用だと思います。

新しいプラグインの仕組みについて書く場合は、次のテンプレートを参照してください。

```
# XXXのアーキテクチャ

## どう書ける?

- 実際のコード例

## どういう仕組み?

- prototypeを拡張しているなど具体的な仕組み
- その機構のコードへのリンク
- その仕組みやプラグインについてドキュメントへのリンク

## どういうことに向いている?

- どういう用途で使われてる(ユースケース)
- 変換する毎にファイルとして吐き出さないので、高速に複数の変換をするのに向いている等

## どういうことに向いていない?

- 変換の仕組み上書き換え等を行うプラグインを扱いにくい等

## この仕組みを使っているもの

- XXX以外にも同様の仕組みを使っているものがあるなら

----

## チェックリスト

- [ ] どう書ける?
- [ ] どういう仕組み?
- [ ] どういうことに向いている?
- [ ] どういうことに向いていない?
- [ ] この仕組みを使っているもの
- [ ] 実装してみよう
- [ ] エコシステム
```

以下からこのテンプレートで使ったIssueを立てることができます。

- 新しいProposalを書く

Proposalの具体例

現在あるProposal一覧を参考にしてみるとよいかもしれません。

- jQuery Plugin · Issue #8 · azu/JavaScript-Plugin-Architecture

テスト

`$ npm test` を実行するとコードや文章に対するテストが実行されます。

```
npm test
```

文章はtextlintによる単語のチェックが行われます。

コミットメッセージ

AngularJSのGit Commit Guidelinesをベースとしています。

- conventional-changelog/angular.md at master · ajoslin/conventional-changelog

次のような形で1行目に概要、3行目から本文、最後に関連するIssue(任意)を書きます。

```
feat(ngInclude): add template url parameter to events

The `src` (i.e. the url of the template to load) is now provided to the
`$includeContentRequested`, `$includeContentLoaded` and `$includeContentError`
events.

Closes #8453
Closes #8454
```

```
                      scope        commit title

        commit type       /               /
                \         |               |
                 feat(ngInclude): add template url parameter to events

        body ->  The 'src` (i.e. the url of the template to load) is now provided to the
                 `$includeContentRequested`, `$includeContentLoaded` and `$includeContentError`
                 events.

 referenced  ->  Closes #8453
 issues          Closes #8454
```

1行の `feat` や `fix` といったcommit typeは、迷ったらとりあえず `chore` と書いて、 `scope` も省略して問題ないので次のような形でも問題ありません。

```
chore: コミットメッセージ
```

このコミットメッセージの規約はconventional-changelogによる自動生成のためでもありますが、取り込むときに調整できるので無視してもらっても問題はありません。

以下を見てみるとよいかもしれません。

- 良いChangeLog、良くないChangeLog | Web Scratch
- われわれは、いかにして変更点を追うか