

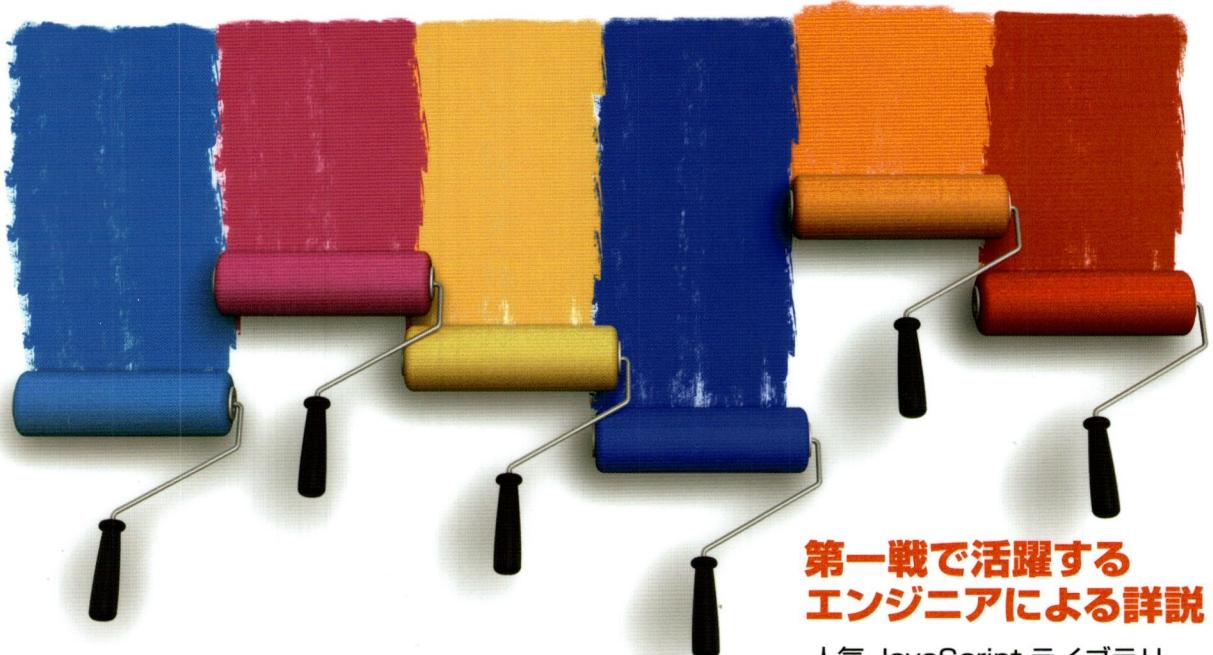
NEXT
ONE

React入門

React・Reduxの導入から
サーバサイドレンダリングによるUXの向上まで



穴井宏幸、石井直矢、柴田和祈、三宮 肇 著



第一戦で活躍する
エンジニアによる詳説

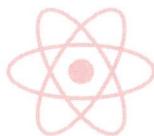
人気 JavaScript ライブライ
React を導入するまでの
道筋を完全解説



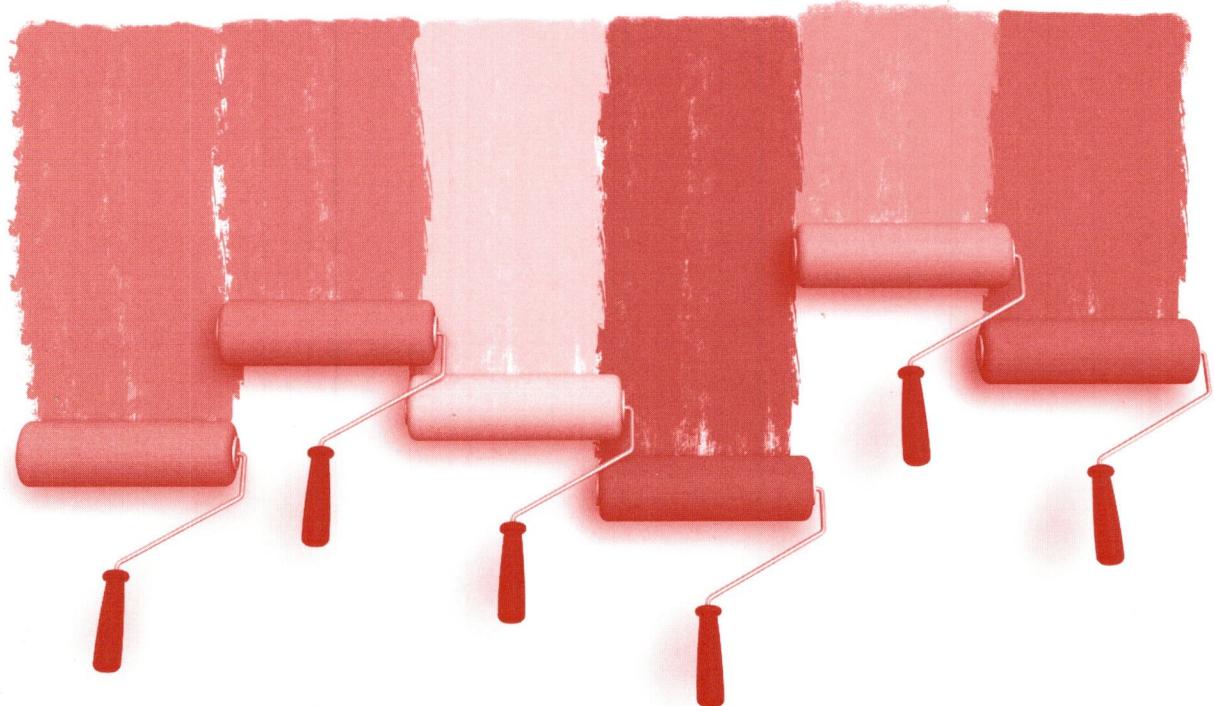
SE
SHOEISHA

React入門

React・Reduxの導入から
サーバサイドレンダリングによるUXの向上まで



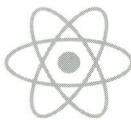
穴井宏幸、石井直矢、柴田和祈、三宮 肇 著



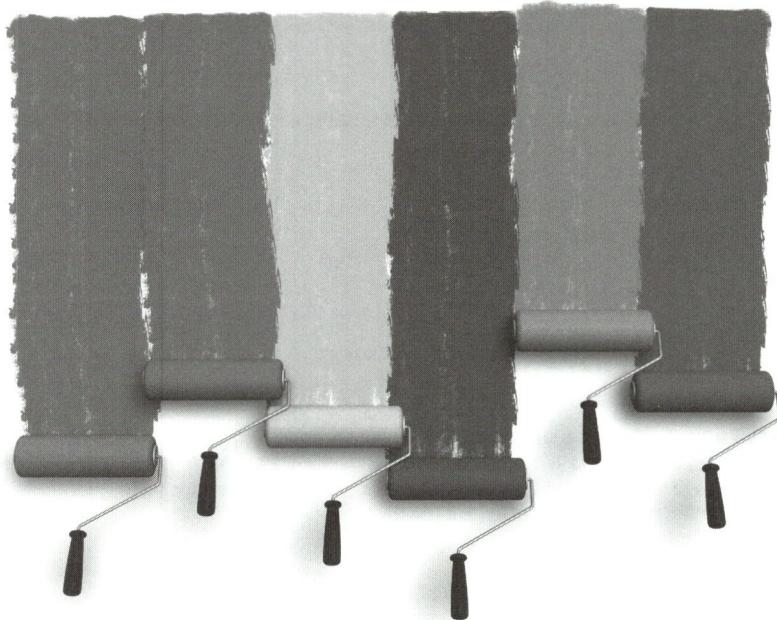


React入門

React・Reduxの導入から
サーバサイドレンダリングによるUXの向上まで



穴井宏幸、柴田和祈、石井直矢、三宮 肇 著



SE
SHOEISHA

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願い致しております。下記項目をお読みいただき、手順に従ってお問い合わせください。

●ご質問される前に

弊社Webサイトの「正誤表」をご参照ください。これまでに判明した正誤や追加情報を掲載しています。

正誤表 <http://www.shoeisha.co.jp/book/errata/>

●ご質問方法

弊社Webサイトの「刊行物Q&A」をご利用ください。

刊行物Q&A <http://www.shoeisha.co.jp/book/qa/>

インターネットをご利用でない場合は、FAXまたは郵便にて、下記“翔泳社 愛読者サービスセンター”までお問い合わせください。

電話でのご質問は、お受けしておりません。

●回答について

回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

●ご質問に際してのご注意

本書の対象を越えるもの、記述個所を特定されないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、あらかじめご了承ください。

●郵便物送付先およびFAX番号

送付先住所 〒160-0006 東京都新宿区舟町5

FAX番号 03-5362-3818

宛先 (株) 翔泳社 愛読者サービスセンター

※本書に記載されたURL等は予告なく変更される場合があります。

※本書の出版にあたっては正確な記述につとめましたが、著者や出版社などのいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関しててもいっさいの責任を負いません。

※本書に掲載されているサンプルプログラムやスクリプト、および実行結果を記した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。

※本書に記載されている会社名、製品名はそれぞれ各社の商標および登録商標です。



前書き

この度は本書をお買い上げいただき、誠にありがとうございます。Web フロントエンドは、流行り廃りが他の界隈と比較して早く、「また新しいライブラリを使っているのか」と揶揄されがちです。ですが、その流行り廃りの中にはタスクランナやモジュールバンドラなどの開発ツール、Sass や PostCSS などの CSS 拡張言語など、さまざまな要素が含まれており、React や Angular、Vue.js などの View（見た目の部分）を構築するライブラリ・フレームワークに限ってみるとここ3年くらいはトレンドに大きな変化はありません。React は筆者が注目し始めた3年前から、内部の実装は変われど、外から利用する API に大きな変更はほとんどありません（React が後方互換性をなるべく維持して開発しているということもあります）。

また、React が普及に大きく貢献した仮想 DOM やコンポーネント指向の開発スタイルは、他のフレームワーク・ライブラリにも取り入れられており、Web アプリケーション開発において今や欠かせないものになっています。開発ツールに関しては、「Less Configuration」という考え方方が広まっており、インストールしたデフォルトの状態である程度利用でき、学習コストが低いツールがよいとされています。本書は `create-react-app` を利用しており、モジュールバンドラやトランスペイラの複雑な設定に気をとらわれることなく、React や Redux の学習に集中できるように意識して執筆しました。

「流行り廃りが早い」ことは悪いことだけではなく、そこに身をおいているエンジニアとしては毎日ワクワクする楽しい環境です。ライブラリ・フレームワークだけではなく、最近ではブラウザの API や言語自体の機能も私がエンジニアになった2009年と比べると考えられないスピードで進化を遂げています。iOS や Android のネイティブアプリの方が機能的には制限がありませんが、アイデアを形にするプラットフォームとして Web の価値はこれからも変わらないと思います。本書をきっかけに Web や Web フロントエンドに魅力を感じてくれる方が増えると幸いです。

最後に、個人的な話になりますが、学生時代にバイト前の空いた時間に本屋で技術書を眺めるのが好きで、技術書がたくさん並んだ本棚を見ながら「いつかここに名前が並ぶようなエンジニアになりたいな～」とぼんやりと思っていました。今回、その大きな目標が叶い、感慨深いものがあります。このような素晴らしい機会をご提供いただいた翔泳社様、筆の遅い執筆者陣の怒涛の追い上げに親切にご対応いただいた緑川さんにこの場を借りてお礼申し上げます。



はじめに	iii
------------	-----



1 章 React・Redux とは？

1

1.1 React とは？	2
React の特徴	3
その他のライブラリ・フレームワークとの比較	4
1.2 Flux とは？	6
Flux の特徴	6
Flux の構成要素	6
1.3 Redux とは？	9
Redux の特徴	9
Redux の構成要素	10



2 章 create-react-app で開発をはじめよう

17

2.1 create-react-app とは？	18
開発環境を整える	18
インストール	18
create-react-app のインストール	25
2.2 アプリケーションの作成	26
プロジェクトの構成	26
アプリケーションを起動	27
Hello, World!	28



3 章 JSX

31

3.1 JSX とは？	32
JavaScript を拡張した言語	32

JSX はなぜ必要なのか？	35
JSX の文法	37
⌚ 3.2 Babel を使って JSX を JavaScript に変換する	43
トランスペイラとは？ Babel とは？	43
CLI	43
webpack とは	47



章

React コンポーネント

53

⌚ 4.1 React コンポーネントとは？	54
コンポーネント開発の準備	54
Functional Component と Class Component	54
コンポーネントの再利用	56
React エレメント	57
データの受け渡し (props)	60
⌚ 4.2 state とイベントハンドリング	71
コンポーネントの準備	71
イベントハンドリング	80
State のまとめ	83
⌚ 4.3 ライフサイクル	85
マウントに関するライフサイクルメソッド	85
データのアップデートに関するライフサイクルメソッド	86



章

Redux でアプリケーションの状態を管理しよう

91

⌚ 5.1 Redux でアプリケーションの状態を管理する	92
Redux のみで Todo アプリケーションを実装	92
Redux の構成	93
ActionCreator を定義する	95
Store を生成する	95
React.js と組み合わせよう	102
ファイルを機能ごとに分割する	106



5.2 react-redux	109
react-redux のインストール	109
Container Component と Presentational Component	109
react-redux が行なっていること	110
Todo アプリに react-redux を導入する	115



6 章 ルーティングを実装しよう

121

6.1 ルーティングとは	122
ルーティングの実装パターン	122
ルーティングのライブラリ紹介	124



7 章 Redux Middleware

141

7.1 Redux Middleware とは？	142
Redux Middleware の基礎	142
Action のログを表示する Redux Middleware を使う	142
7.2 Action のログを表示する Redux Middleware を作る	147
ミドルウェアの仕組み	147
ログミドルウェアの実装	151
7.3 ミドルウェアのサンプル	152
thunk ミドルウェア	152
localStorage	153



8 章 Redux の非同期処理

155

8.1 非同期処理の基礎	156
非同期処理とは？	156
redux-thunk による非同期処理	157

8.2	thunk ミドルウェアの便利な使い方	162
	複数のアクションをまとめる	162
	getState 関数	165



章

UI をきれいにしよう

167

9.1	UI ライブラリ	168
	React コンポーネントのスタイリング	168
	UI ライブラリとは	170
	Material-UI	171
	Material-UI を使ってみる	172
9.2	アニメーションを実装する	178



章

より実践的なアプリケーションを作ろう

183

10.1	アプリケーション作成の準備	184
	作成するアプリケーション	184
	事前準備	185
	Yahoo! ショッピングのカテゴリランキング API の仕様	188
10.2	アプリケーションを作ろう	192
	アプリケーションの雛形を作成	192
	ファイル・ディレクトリ構成	192
	Redux の導入	193
	ページルーティングの導入	195
	ページルーティングを実装	197
	非同期処理の実装	201
	Reducer の実装	205
	Material-UI の導入	215



11 章 アプリケーションのテストを書こう

225

11.1 テストライブラリ (テストフレームワーク)	226
Jest	226
11.2 React・Redux アプリケーションのテスト	231
ActionCreator のテスト	231
非同期 Action Creator のテスト	232
Reducer のテスト	235
React コンポーネントのユニットテスト	237
React コンポーネントのスナップショットテスト	240



12 章 作ったアプリケーションを公開しよう

245

12.1 アプリケーションを公開する	246
GitHub Pages	246
GitHub Pages のメリット・デメリット	253
12.2 Firebase について	254
Firebase とは?	254



13 章 サーバサイドレンダリング

267

13.1 サーバサイドレンダリングとは?	268
サーバサイドレンダリングは必須ではない	268
React におけるサーバサイドレンダリングの流れ	269
React v15 以前のサーバサイドレンダリング	270
13.2 React v16 以降のサーバサイドレンダリング	287
React v16 でのサーバサイドレンダリングの変更点	287
Redux でのサーバサイドレンダリング	292

索引	297
----------	-----

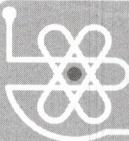
第 1 章

React・Reduxとは？

本章では、React・Reduxについて大筋を紹介します。

それぞれの仕組みを一通り理解した上で、

次章からReactの詳しい説明に入っていきます。



1.1 Reactとは？

ReactはFacebook社が開発しているJavaScriptライブラリであり、WebのUIを作ることに特化しています。UIを細かく分割した場合、そのパーツごとに見た目と機能が存在しています。「見た目」とはHTMLやCSSによって作られた文字通りの見た目のことです。「機能」とは、例えば入力フォーム内の文字数をカウントしたり、あるいは入力した文字列をサーバサイドに送信するなどの振る舞いのことをさします。

従来のクライアントサイドでは、HTMLで見た目を作り、JavaScriptで機能を定義していました。よって汎用的なパーツを作成しても、HTMLとJavaScriptをそれぞれ導入しなくてはならないため管理がしづらいという問題がありました。

見た目と機能をひとまとめにしたものを作成する「コンポーネント」と呼びます。Reactを用いるとコンポーネントを容易に作成することができます。

WebのUIはコンポーネントのツリー構造と捉えることが可能ですが、コンポーネントには親子関係があり、その大元はWebページ全体となります（図1.1）。

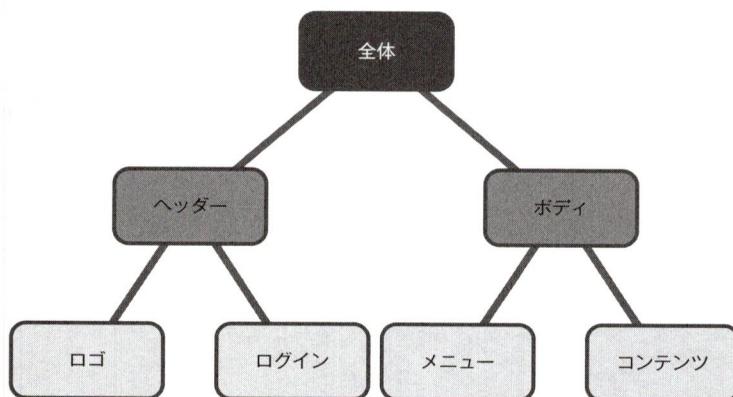


図1.1 コンポーネントツリー構造



Reactの特徴

Reactの特徴としては以下のものがあげられます。それぞれ詳しく解説していきます。

- Virtual DOM
- JSX

Virtual DOM

DOM とは Document Object Model の略で、HTML や SVG といった XML 形式の文書へのインターフェイスです。HTML や SVG の各要素にアクセスするための API ともいえます。DOM は XML 形式の文書をツリーとして表現し、アクセス可能にします。その結果、文書構造、見た目、およびコンテンツを変更することができます。

Virtual DOM はその名の通り、ブラウザが保持している DOM とは別に、React 内で仮想の DOM を管理しているイメージです。Virtual DOM を学ぶ前に、まずブラウザのレンダリングの仕組みを理解する必要があります。レンダリングとは、コードをブラウザへ描画するまでの処理の総称です（図1.2）。



図1.2 レンダリングの仕組み

図1.2のうち、レイアウトとペイントはブラウザにとって負荷が高い処理となります。

DOM操作を行うとレンダーツリーの更新が起こり、再度レイアウトとペイントの処理が走ります（これらをそれぞれリフロー・リペイントと呼びます）。よって、ブラウザのパフォーマ

ソース面において、DOM操作を極力減らすことはよい対策といえます。

ReactではDOMと対構造となっているVirtual DOMを定義し、ページ内を変化させる場合はまずVirtual DOMを変化させます。Virtual DOMはブラウザのレンダリングとは切り離されているため、いくら変更を加えたところで影響はありません。Virtual DOMの変化の差分を算出し、その対応部分のDOMを変化させます。そうすることにより、最小限のDOM操作でページ内を変化させることができます（図1.3）。

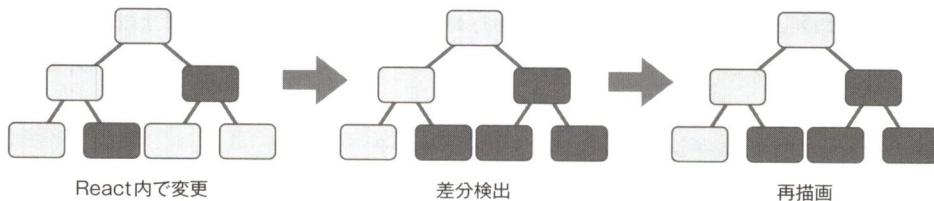


図1.3 Virtual DOMの仕組みを図式化

従来、DOM操作のパフォーマンスチューニングは人の手によって行われており骨が折れる作業でした。しかしReactではそれが自動的に行われるため、ユーザーはコンポーネントツリーの描画に注力することができるのです。

場合によっては、さらなるパフォーマンスチューニングが求められることもありますが、そちらに関しては後々説明します。

JSX

ReactではVirtual DOMを用いるとさきほど述べました。Virtual DOMはReact.createElementというメソッドで生成することができます。しかし、全てのDOMに対してReact.createElementメソッドを行うのは現実的ではありませんし見栄えもよろしくありません。これに対し、JavaScriptを拡張した言語であるJSXを用いることで、開発の効率化を行うことができます。JSXに関しては第3章で詳しく説明するのでここでは割愛します。

● その他のライブラリ・フレームワークとの比較

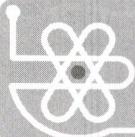
Ajaxが提唱されて以来、サーバサイドでHTMLを構築し、クライアントサイドではそれを表示すれば良いという単純な話ではなくなってきました。ブラウザの処理速度も向上し、サーバサイドで行われていた処理をクライアントサイドで行う例が増えていきました。JavaScript

は従来、Webページに動きをつけるものとして開発された言語ですから、複雑なロジックを組むにはあまりに貧弱です。そこでサーバサイドだけではなく、クライアントサイドにもフレームワークが必要であるという流れがでてきました。

そのような流れの中で誕生した Backbone.js では Model（単一のデータ）、Collection（データの集合）、View（見た目）を分離することで、非同期通信で取得してきたデータを HTML に流し込む作業が簡略化されました。また、クライアントサイドでルーティングを行うための Router や History を持っていました（第6章参照）、シングルページアプリケーションの流布に大きく貢献しました。しかし、View の親子構造であったり、View と Model 間のデータバインディング処理を自分で設定する必要があり、その実装は難易度が高いものでした。薄いライブラリなので、ブラックボックスがなく、全て理解した上で利用できるという点で高い人気を誇っていました。

その後、双方向データバインディングを備えた Angular.js が流行しました。Model と View 間のデータバインディングを自動で行ってくれるため、今までのデータを View に反映したり、ユーザーからの入力をデータに反映したりしていた処理がほぼ必要なくなり、コード数がグッと減りました。

そして、React が Virtual DOM と JSX という概念を持って登場しました。React は Backbone.js や Angular.js とは違い、View のみのライブラリです。その特徴はさきほど紹介した通りです。React では細かいコンポーネントの組み合わせで Web アプリケーションを形作っていくという考え方から、大規模なアプリケーション制作に向いています。



1.2 Fluxとは？



Fluxの特徴

クライアントサイドが複雑化し、MVC（Model、View、Controller）やMVVM（Model、View、ViewModel）といった考え方方が広まっていきました。その大きな特徴は、見た目とデータを分離することにありました。jQueryを用いてアプリケーションを作っていた時代には、DOMにデータを持たせていることがほとんどでした。例えば、チェックボックスのオンオフはチェックボックス自体のDOMを確認することでしか取得できませんでした。そこで、チェックボックスのオンオフをJSONで管理し、チェックボックスは常にそのJSONを見ながらオンオフを切り替えることで、見た目とデータを分離することができるようになりました。しかし、アプリケーションが巨大化し、ロジックが複雑になってくると、ModelとView、またはModel同士の結びつきも複雑化し、データの反映時に予期せぬ副作用が起きてしまう可能性があります。

そこで考案されたのがFluxという考え方です。Fluxとはアーキテクチャの一種であり、その実装にはいくつかの種類があります。Fluxではユーザーの入力からActionを作成し、そのActionをdispatchすることでStoreにデータを保存し、Viewに反映させるといった流れを取ります。データが一方向のフローで流れるため、複雑なアプリケーションになっても不整合が起きづらい仕組みになっています。



Fluxの構成要素

Fluxはアーキテクチャの一種ですが、Facebook社によるフレームワークとしての実装としても存在します。ややこしいのですが、その実装を元にFluxの構成要素に関して紹介していきます（図1.4）。

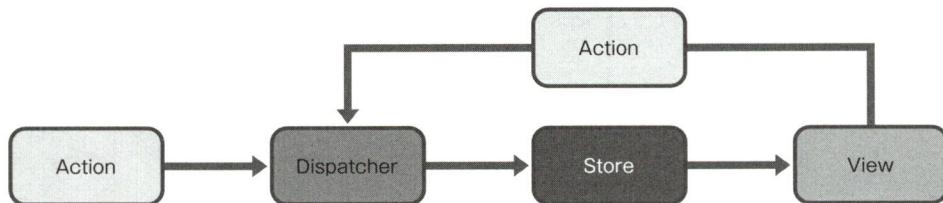


図1.4 Fluxの構成要素

View

ViewはReactコンポーネントと同義と考えて良いでしょう。Viewに対してユーザーから何らかの入力や操作があった場合に、その内容を示すActionを作成します。

Action

Actionは単なるオブジェクトで、非常にシンプルなものです。ActionをdispatchすることでActionがStoreに渡されます。「商品をカートに追加する」であったり、「商品を購入する」といったユーザーの操作はもちろんこの対象です。それだけではなく「モーダルを表示する」や「キャッシュしていたデータをリセットする」などのシステムチックなものも含みます。

Dispatcher

Dispatcherが全てのデータの流れを管理しています。しかし、中身は単なるEventEmitterであり、DispatchされたデータはStoreが受け取ることができます。Dispatcherを通じて、全てのActionがStoreに送られるイメージです。

Store

Storeはアプリケーションの状態(state)とロジックを保持している場所です。StoreはMVCでいうModelに似ていますが別物です。アプリケーション独自のドメインで状態(state)を管理することができるのが特徴です。Dispatcherによって渡されてくるActionを受け取り、アプリケーションの状態を変化させます。例えば、本を読んでいるかどうかの状態をStoreが持っている場合、リスト1.1のような形になるでしょう。

リスト1.1 Storeの例

```
{
  status : 0 // 0: 手をつけていない, 1: 読み途中, 2: 読了
}
```

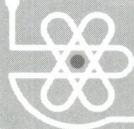
これに対し、リスト1.2のようなActionをDispatchします。

リスト1.2 ActionをDispatchする例

```
{  
  type: 'START_READING'  
}
```

StoreはこのActionを受け取り、「START_READING」というtypeから、読書ステータスを1（読み途中）に変更します。

もし、受け取ったtypeが「FINISH_READING」だったとしたら、Storeにある読書ステータスは2（読み終了）に変わるでしょう。Storeの内容が変更されるとそれをViewが感知して再描画が走ります。



1.3 Reduxとは？

Reduxは数あるFluxアーキテクチャの一種で、Dan Abramov氏によって作成されたものです。おもにReactと組み合わせることが多いですが、ReduxはあくまでFluxアーキテクチャの一種であるため、Reactと依存関係にあるものではありません。Angular.jsやVue.jsと組み合わせることも可能です。

まずはその特徴について詳しく見ていきましょう。



Reduxの特徴

Reduxには次のような「Reduxの3原則」と呼ばれるものがあります。

- Single source of truth
- State in read-only
- Changes are made with pure functions

これらは主にアプリケーションの状態管理に関わる部分で、Reduxの思想として非常に重要なものとなります。それでは1つずつ説明していきます。

Single source of truth

アプリケーション内の全ての状態を一枚岩の大きなオブジェクトとして管理します。これにより、アプリケーションのデバッグやテストが容易になります。必要な状態をどこからでも取り出すことができるので、アプリケーションの実装自体もシンプルにすることができます。

State in read-only

アプリケーションの状態はコンポーネントから参照することができますが、直接変更することはできません。Action（どんな動作を行ったかを示す単なるオブジェクト）を dispatch（発行）することが、アプリケーションの状態を変更する唯一の方法です。これにより、データの流れが完全に一方向となり、余計な副作用が生まれることがなくなります。

Changes are made with pure functions

状態の変更は副作用のない純粋関数によって行われます。純粋関数とはなんでしょうか。簡単な言い方をすると、「同じ入力値を渡すたび、決まって同じ出力値が得られる関数」をさします。Reduxでは、Actionを入力として受け取り、状態を変化させ、それを出力します。その部分を担う存在としてReducerがあります。Reducerに関して詳しくは後述します。



Reduxの構成要素

さて、Reduxはどのような要素で構成されているのでしょうか。基本的にはFluxと似ています。特にView、Action、Store、Dispatcherの概念は一緒と考えて良いでしょう。Reduxではそれらに加えて少し登場人物が増えることになります。

Reducer

Reduxの特徴でも述べましたが、Reducerは状態を変化させるための関数です。どこからともなく送られてきたActionの内容によって既存の状態を変化させます。先ほどの「本を読んでいるかどうか」という状態を変化させる例をもって説明します（リスト1.3）。

リスト1.3 Reducerの例

```
function books(state = null, action) { —————①
  switch (action.type) { —————②
    case 'START_READING': —————③
      return {
        ...state,
        status : 1,
      }; —————④

    case 'FINISH_READING': —————④
      return {
        ...state,
        status : 2,
      }; —————④

    default:
      return state;
  }
}
```

リスト1.3のコードではまずbooksという関数が定義されています❶。これがReducer本体となります。第一引数にstate、第二引数にはactionが渡ってきます。stateとは文字通り、今の状態を表すオブジェクトで、Reducerではこれに変更を加えることになります。actionとは、どんな動作を行ったかを示すオブジェクトです❷。actionには必ずtypeというプロパティが生えており、行った動作を示す文字列が渡ってきます。ちなみに文字列以外でも動作可能です。typeを一意にするためにSymbol要素を使う方法もあります。今回は、本を読み始めたことを示すSTART_READINGというactionタイプと❸、本を読了したことを示すFINISH_READINGというactionタイプがあります❹。もしSTART_READINGというactionタイプが渡ってきた場合は読書ステータスを1に変更し、FINISH_READINGというactionタイプが渡ってきた場合は読書ステータスを2に変更します。今回の例では、switch文を用いてactionタイプによって処理を振り分けています。

リスト1.4 Spread Operator

```
return {
  ...state,
  status : 1,
}
```

リスト1.4の部分で頭にはてながついてしまった方がいると思うので説明します。この構文はES2015で定義されたSpread Operatorというものです。

Spread Operator

Spread Operatorは、ES2015から導入された仕様です。これを用いることで、配列やオブジェクト、さらには関数に与える引数を展開することができます。

例えばリスト1.5のように配列を展開することができます。

リスト1.5 配列の展開

```
const hoge = [2, 3];
console.log([1, ...hoge, 4, 5]);
```

実行結果

```
[1, 2, 3, 4, 5]
```

オブジェクトの例も挙げてみます。

リスト1.6 オブジェクトの展開

```
const fuga = { name: 'Taro', age: 25 };
const piyo = { name: 'Jiro', location: 'Tokyo' };
console.log({...fuga, ...piyo});
```

実行結果

```
{ name: 'Jiro', age: 25, location: 'Tokyo' }
```

このようにオブジェクトの展開が可能です。プロパティが被っている場合は、後ろのものが優先されます。リスト1.6の例では連結後のnameは'Jiro'となります。次に、関数に与える引数の展開を行ってみます。配列を関数の引数に直接与えたい場合に、Spread Operatorを用いるとリスト1.7のように記述することができます。

リスト1.7 関数に与える引数の展開

```
const myFunc = (x, y, z) => [ x, y, z ];
const args = [0, 1, 2];
myFunc(...args);
```

実行結果

```
[ 0, 1, 2 ]
```

また、引数の一部にのみ適用することも可能です（リスト1.8）。

リスト1.8 引数の一部への適用

```
const myFunc = (v, w, x, y, z) => [v, w, x, y, z];
const args = [0, 1, 2];
myFunc(3, ...args, 4);
```

実行結果

```
[3, 0, 1, 2, 4]
```

さて、話を戻しましょう。Reducerの説明の途中でしたね。switch文を用いてactionタイプによって処理を振り分けるという話をしていました。処理の振り分けさえできればswitch文である必要もありません。リスト1.9のような書き方も可能です。

リスト1.9 様式の振り分け

```

const reducers = {
  START_READING : (state, action) => (
    return {
      ...state,
      status : 1,
    };
  ),
  FINISH_READING : (state, action) => (
    return {
      ...state,
      status : 2,
    };
  ),
};

function books(state = null, action) {
  if (!reducers[action.type]) {
    return state;
  }
  return reducers[action.type](state, action);
}

```

switch文では一致するactionタイプがない場合、最後のcaseまで見に行ってしまいます、上記のようにオブジェクトのKeyを用いた判定であれば早期リターンをすることができます。

Reducerにおける注意点

Reducerでは状態を変化させますが、Component側ではその変化を元にViewを描画します。ここで注意していただきたいのが、JavaScriptの参照渡しについてです。変数への代入はリスト1.10のように値渡しになるイメージはつきやすいと思います。

リスト1.10 Reducerにおける値渡し

```

let x = 10;
let y = x;
y = 5;
console.log(y); // 5
console.log(x); // 10

```

x、yという変数の格納場所はそれぞれ存在しており、yを変更したところでxには関係がありません。

次に参照渡しの例です（リスト1.11）。

リスト1.11 Reducerにおける参照渡し

```
let x = [1, 2, 3];
let y = x;
y[1] = 4;
console.log(y); // [1, 4, 3]
console.log(x); // [1, 4, 3]
```

2行目でyにxを代入する際、xの値のコピーがyに渡されるのではなく、xの値が格納されているメモリ番地がyに渡されます。つまり、この時点でxとyは同じ箱を見ているのです。よって3行目で配列の中身を書き変えると、xとyともに変更されます。

さて、Reducerにおいて参照渡しをしてしまうと何が起きるでしょうか。Reducerで状態を変化させた後、View側でその変化の比較を行いたい場合があります。例えばリスト1.12のようなコレクションがあったとします。

リスト1.12 Reducerの参照渡し

```
const member = [
  {
    id: 1,
    name: 'Taro'
  },
  {
    id: 2,
    name: 'Jiro'
  }
];
```

これに対し、リスト1.13のオブジェクトを加えたいと思います。

リスト1.13 オブジェクトの追加

```
member.push({
  id: 3,
  name: 'Saburo'
});
```

これによって member の中身はリスト 1.14 のように変化します。

リスト 1.14 オブジェクトの変化

```
const member = [
  {
    id: 1,
    name: 'Taro'
  },
  {
    id: 2,
    name: 'Jiro'
  },
  {
    id: 3,
    name: 'Saburo'
  }
];
```

今回の例、実は参照渡しになっていることにお気付きでしょうか。変更前と変更後で比較すると配列の中身は変わっているはずなのにイコール判定されてしまいます。そこで、オブジェクトや配列の操作を行う場合は基本的には参照渡しではなく、値のコピーを渡してあげることが大切です。さきほど説明した Spread Operator は、オブジェクトや配列のコピーを行ってくれるので元の値には影響を与えません。また、参照を切る方法として ES2015 で標準化された Object.assign() というメソッドを用いることが多いです。Object.assign() を用いるとリスト 1.15 のような書き方となります。

リスト 1.15 Object.assign メソッドを用いた例

```
function books(state = null, action) {
  switch (action.type) {
    case 'START_READING':
      return Object.assign({}, state, {
        status: 1
      });

    case 'FINISH_READING':
      return Object.assign({}, state, {
        status: 2
      });
  }
}
```