りあクト！

第4版

TypeScriptで始めるつらくないReact開発

③React応用編

大岡由佳

最新のReact 18、Suspenseにも対応

Reactの状態管理と副作用を圧倒的にわかりやすく！

シリーズ累計 2.5万部 突破！

読者からの声

「この本を選んだ自分を褒めてあげたい」

「実践的なReactを学びたい人にオススメ」

BOOTH 技術書 前版 1位

秋谷が未経験から1週間で戦力になるためのReact研修もいよいよ佳境。コンポーネントがようやく書けるようになったばかりの彼女にレンダリングの最適化、ページのルーティング、グローバルな状態管理と次々に歯ごたえのあるテーマが襲いかかる。それらを乗り越えReactの真髄をつかむことができるか。そしてガードの固い柴崎に心を許してもらえるのか。感動の最終巻。

本作の構成（全3巻）

りあクト! TypeScript で始める
つらくない React 開発 第 4 版
【③ React 応用編】

本書について

登場人物の紹介

柴崎雪菜（しばさき ゆきな）

とある都内のインターネットサービスを運営する会社のテックリードのフロントエンドエンジニア。React 歴は 6 年ほど。本格的なフロントエンド開発チームを作るための中核的人材として、今の会社に転職してきた。チームメンバーを集めるため人材採用にも関わり自ら面接も行っていたが、彼女の要求水準の高さもあってなかなか採用に至らない。そこで「自分が React を教えるから他チームのエンジニアを回してほしい」と上層部に要望を出し、社内公募が行われた。

秋谷香苗（あきや かなえ）

柴崎と同じ会社に勤務する、新卒 2 年目のやる気あふれるエンジニア。入社以来もっぱら Ruby on Rails によるサーバサイドの開発に携わっていたが、柴崎のメンバー募集に志願してフロントエンド開発チームに参加した。そこで柴崎から「1 週間で戦力になって」といわれ、彼女にマンツーマンで教えを受けることになる。

前版との差分および正誤表

過去の版からの変更点と、本文内の記述内容の誤りや誤植についての正誤表は以下のページに掲載しています。なお、電子書籍版では訂正したものを新バージョンとして随時配信していきます。

- 各版における内容の変更
  https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/blob/main/CHANGELOG.md

- 『りあクト！ TypeScript で始めるつらくない React 開発 第 4 版』正誤表
  https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/blob/main/errata.md

サンプルコードについて

本文で紹介しているサンプルコードは、GitHub に用意した下記の専用リポジトリにて、章・節ごとに分けてすべて公開してあります。

https://github.com/klemiwary/Riakuto-StartingReact-ja4.0

挙動をその場で確認していただくため StackBlitz に置いているサンプルもありますが、本文中に随時 URL を掲載していますのでそちらをブラウザでご覧ください。学習を効率的に行うためにも、本文を読み進めながらこれらのコードを実際に実行することを強くおすすめします。またコードをご自身で変更して挙動のちがいを確かめてみるのも理解の助けになるはずです。

なおリポジトリのコードと本文に記載されているコードには、ときに差分が存在します。紙面に適切になるよう改行位置を調整したり、各種コメントアウト文を省略するなどによるものですが、そのため行番号が一致しないことがありますので、ご注意ください。

本文および上記リポジトリに掲載しているコードは、読者自身のプログラムやドキュメントに流用してかまいません。それにあたりコードの大部分を転載する場合を除き、筆者に許可を求める必要はありません。出典を明記する必要はありませんが、そうしていただければありがたいです。

本書内で使用している主なソフトウェアのバージョン

- React（`react`）…… 18.2.0
- React DOM（`react-dom`）…… 18.2.0
- Vite（`vite`）…… 3.0.9
- TypeScript（`typescript`）…… 4.8.2
- ESLint（`eslint`）…… 8.22.0
- Prettier（`prettier`）…… 2.7.1
- React Router DOM（`react-router-dom`）…… 6.3.0
- Redux（`redux`）…… 4.2.0

- React Redux（`react-redux`） …… 8.0.2
- Redux Toolkit（`@reduxjs/toolkit`） …… 1.8.4
- SWR（`swr`） …… 2.0.0-beta.6

目次

「① 言語・環境編」目次

「② React 基礎編」目次

第 10 章 コンポーネント操作の高度な技法

10-1. ロジックを分離、再利用する技術

10-1-1. React におけるロジック分離技術の歴史

「アプリケーションが大規模になってくると、コンポーネントから state を伴ったロジックを切り出して再利用できるようにしたいと考えるようになるのはプログラマとしての本能みたいなもの。この課題をいかにスマートに解決するかという模索を React はずっと続けていて、その末にたどり着いた現時点での最適解が先に紹介した `useState` や `useEffect` を使った手法なの。これらの API を使って関数コンポーネントに状態や副作用といった機能を追加する技術を総称して **Hooks** と呼んでる。そして状態を扱う Hooks は **State Hook**、副作用を扱う Hooks は **Effect Hook** とカテゴライズされる」

「へー、フックスっていうんですか。その 2 つ以外にも Hooks はあるんですか？」

「うん、おいおい紹介していくよ。そして Hooks とは API インターフェースのことでもあるので、アプリケーション開発者が自分で作れるし、最近はサードパーティーのライブラリも Hooks インターフェースによる API を提供するようになってる。React の歴史で、これだけ急速に広く受け入れられた技術もめずらしいだろうね」

「そうなんですか」

「でも Hooks という現状での最適解に至るまでは長い試行錯誤の歴史で、公式が推したものでもコミュニティに受け入れられず廃れていったものもある。ただ Hooks 以前の技術は完全に消えてしまったわけではなく、用途が限定的ながら今でも使われ続けてる。どういう思想でそれらの技術が登場し、後発の技術に道を譲ることになったのか、それを知っておくことは現場の開発者にとって大事だと思う。それらが用いられているコードに行き当たる機会もまだまだあるだろうしね」

「なるほど」

「だからそれらをこれから紹介していくよ。これまで登場した React におけるコンポーネントのロジックを分離・再利用するための代表的な技術は、時系列に沿って並べると次のようになる」

- **Mixins** → 2013 年 ～ 2017 年
- **HOC**（Higher Order Component）→ 2015 年 ～ 現在
- **Render Props** → 2017 年 ～ 現在

•Hooks → 2019 年 ～ 現在

「最初の mixins から紹介していこう。2013 年に公開された当初の React ではコンポーネントはクラスしかなくて、しかもそれを ES2015 のクラス定義ではなく `React.createClass` というメソッドを使って擬似的なクラスを生成するようになってた話はしたね。`createClass()` の引数はオブジェクトになってて、`render` メソッドなんかもそのプロパティとして実装していくの。そして `mixins` というプロパティで、そのコンポーネントに追加したいクラスメンバーを任意のオブジェクトに格納して登録できた。それが mixins の手法ね」

「うーん、ちょっとイメージがわきません……」

「前にカウンターを作ったよね。あれに最大値を props として設定してそれを超えるとリセットされるようにしたものを `React.createClass` と mixins を使って書いてみよう。古いやり方なので TypeScript で書くのは難しいから JavaScript になるけど」

リスト 1: カウンターのロジックを mixins にする

```
import React from 'react';

const CounterMixin = {
  getInitialState: () => ({ count: 0 });
  reset: () => {
    this.setState({ count: 0 });
  },
  increment: () => {
    this.setState((state) => ({ count: state.count + 1 }));
  },
  componentDidUpdate() {
    if (this.state.count > this.props.maxCount) this.reset();
  },
};

const Counter = React.createClass({
  propTypes: {
    maxCount: React.PropTypes.number.isRequired,
  },
  mixins: [CounterMixin],

  render: () => {
    const { count = 0 } = this.state;

    return (
      <div>
```

```
        <div>
          {count}
        </div>
        <button onClick={reset} type="button">
          Reset
        </button>
        <button onClick={increment} type="button">
          +1
        </button>
      </div>
    );
  }
});

export default Counter;
```

「ふむふむ。クラスコンポーネントのメンバーを切り出してオブジェクトにまとめ、それを `mixins` プロパティでコンポーネントに登録できるんですね。値が配列形式になってるから、他にも登録したい mixins があればここに追加していく感じでしょうか。……わかりやすいじゃないですか。どうしてこの方法じゃダメだったんでしょうか？」

「初見でわかりやすい・書きやすいというのと、設計がきれい・継続的にメンテナンスしやすいというのは別だからね。まず mixins はコンポーネントとの間に暗黙的な依存関係をしばしば持ってしまう。ミックス先のコンポーネントに特定の名前の props、特定の名前の state、特定の名前のプロパティやメソッドがあることを前提とした記述になりがちで、それではロジックの再利用には使いづらい」

「あー、そう言われてみればたしかに」

「props、state、プロパティやメソッドの名前が意図せず重複してしまうとバグの原因になる。しかも複数の mixin オブジェクトをフラットに配列として渡すインターフェースなので、mixins 間でも名前の衝突が起きないか常に気を配る必要がある」

「むむむ……」

「またロジックが複雑になってくると、ある mixin が他の mixin に依存するようになりがちで、そうするとどれかひとつを削除すると他の mixin が壊れるような事態が生じる。mixins の間には階層構造がなくフラットな関係なため、データの流れや依存関係がどうなっているのかを把握するのも難しくなる。

mixins はこうした欠点のため時間の経過とともに複雑さが雪だるま式に増えてしまう。コンポーネントが mixins のメソッドを呼びだすこともあれば、mixins がコンポーネントのメソッドを呼びだすこともあり、見えない相互依存性が高まっていく。そうすると安全にコードを変更することが難しくなるし、新しくプロジェクトに参加したメンバーが理解するハードルも上がってしまうわけ」

「……そうなんですね。わかりやすくていいなと思ったんですけど」

「だからフレームワークとして easy を売りにしてる Vue.js はこのまんまの mixins を導入してて、3 系でもまだバリバリの現役で使える [1]。でも React は 2016 年 7 月に、まず公式ブログで『Mixins Considered Harmful（有害とされる mixins）』[2] という記事を出して mixins の使用を控えるようにアナウンスした。そして翌 2017 年 6 月のバージョン 16.6.0 では `React.createClass` が非推奨になるのと同時に mixins も廃止されたの」

「へー、見切りが早いですね」

「React は時代に合わなくなってきたものは早めに非推奨の声明を出した上で、段階的に廃止する姿勢を取ってる。でも急進的すぎず、現実を見据えたコミュニティに受け入れられやすい指針を示してるので、他のプロダクトと比べてもこれまで特に大きな混乱はなく新しい技術への移行が進んできてるの」

10-1-2. コミュニティ主導で普及した HOC

「さっき紹介した React 公式ブログの mixins にダメ出しした記事の中で、代わりに使うよう勧められていたのが **HOC** というパターンだった。正式には『Higher Order Component』で、日本語では『高階コンポーネント』と訳される」

「高階コンポーネント？ 以前、関数型言語について学んだとき、関数を引数にとったり関数を戻り値として返す高階関数 [3] が出てきたのをおぼえてますけど」

「そう、HOC は高階関数の手法をコンポーネントに応用したものだね。HOC ではコンポーネントを引数に取り、戻り値としてコンポーネントを返す。React らしく関数型プログラミングの発想から出てきたテクニックなわけ。React 開発チームの初期からのメンバーである Sebastian Markbåge が個人的にサンプルを示したところ [4]、そのやり方が開発者コミュニティの中で広まっていった。それが公式にあらためて追認された形になったのね。

HOC がどんなふうにコンポーネントに機能を追加するのか、ごく簡単なサンプルで使い方を見てみよう。この例では `withTarget` という関数が HOC に相当する」

```
type Props = { target: string };
const Hello: FC<Props> = ({ target }) => <h1>Hello {target}!</h1>;
```

---

[1] 「ミックスイン | Vue.js」https://v3.ja.vuejs.org/guide/mixins.html

[2] 「Mixins Considered Harmful–React Blog」https://reactjs.org/blog/2016/07/13/mixins-considered-harmful.html

[3] 「①言語・環境編」の「3-3-2. 高階関数」を参照のこと。

[4] https://gist.github.com/sebmarkbage/ef0bf1f338a7182b6775

```
export default withTarget(Hello);
```

「えーっと、これは何がやりたいんでしょうか？」
「props の `target` に『Hello』というだけのコンポーネントに、HOC で外から `target` の実体を与えたいのね。そこにたとえば `'Patty'` という文字列を与える `withTarget` 側の実装はこうなる」

```
const withTarget = (WrappedComponent: FC<Props>) => WrappedComponent({ target: 'Patty' });
```

「なるほど、`withTarget` はコンポーネントを受け取りその props の `target` の実体を注入したコンポーネントを返す関数なわけですね。これが高階コンポーネントですか」
「そう。HOC は特定の props を用いてコンポーネントに機能を追加する。だから受ける側のコンポーネントは、その名前の props を受け皿として用意しておく必要があるの。さっきのカウンターコンポーネントを、今度は関数コンポーネントに HOC を適用する形で実装してみよう [5]」

リスト 2: HOC によるカウンターの実装

```
import { Component, FC } from 'react';

type InjectedProps = {
  count: number;
  reset: () => void;
  increment: () => void;
};
type Props = { maxCount: number };
type State = { count: number };

const withCounter = (WrappedComponent: FC<Props & Partial<InjectedProps>>) =>
  class Counter extends Component<Props, State> {
    constructor(props: Props) {
      super(props);
      this.state = { count: 0 };
    }

    #increment = () => this.setState((state) => ({ count: state.count + 1 }));
    #reset = () => this.setState({ count: 0 });

    componentDidUpdate(): void {
      if (this.state.count > this.props.maxCount) this.#reset();
```

[5] https://stackblitz.com/edit/react-ts-bjhtfc?file=Counter.tsx

```
    }

    render = (): JSX.Element => (
      <WrappedComponent
        maxCount={this.props.maxCount}
        count={this.state.count}
        reset={this.#reset}
        increment={this.#increment}
      />
    );
  };

const CounterBox: FC<Props & Partial<InjectedProps>> = ({
  maxCount,
  count = 0,
  reset = () => undefined,
  increment = () => undefined,
}) => (
  <div>
    <div>
      {count} / {maxCount}
    </div>
    <button onClick={increment} type="button">
      +1
    </button>
    <button onClick={reset} type="button">
      Reset
    </button>
  </div>
);

export default withCounter(CounterBox);
// SEE: https://stackblitz.com/edit/react-ts-bjhtfc?file=Counter.tsx
```

「う、だいぶ複雑になりましたね。難しそう……」

「順を追って考えればそんなに難しくはないよ。まず下の `CounterBox` を見てみて。これは状態やロジックを持たない、props が同じなら出力される React Elements も同じ純粋関数なコンポーネントでしょ？ HOC である `withCounter` はこの中の `count`、`reset`、`increment` という 3 つの props および包含するクラスコンポーネントの state とライフサイクルメソッドによって、外からロジックを注入してる」

「この `CounterBox` コンポーネントに props 型として渡されてる `Props & Partial<InjectedProps>` というのは？」

「本来 `CounterBox` を JSX からマウントするとき必要な props は `maxCount` だけ。でも HOC によって `count`、`reset`、`increment` にロジックを注入できるようにしておく必要があるので、その 3 つを組み込みユーティリティ型の `Partial`[6] で省略可能な形にして合成してるのよ。

HOC を TypeScript で使う場合、この内側のコンポーネントの props と HOC 適用後の外側のコンポーネントの props をうまく辻褄が合うように型合成してあげる必要がある。加えて内側のコンポーネントはそれ単体でも成立するよう、HOC から注入される予定の props にはデフォルト値を設定しないといけない。このへんはまあ面倒だけどね」

「ふーむ。すべてを props によって取り回すから、内側と外側の props の調整をする必要があるんですね」

「そう。`CounterBox` を引数として受け取り `withCounter` が返すことになるコンポーネント `Counter` の props 型を見て。`maxCount` ひとつだけになってるでしょ？　だから `withCounter(CounterBox)` は props として `maxCount` だけを指定してあげればいいわけ」

「なるほど」

「その `withCounter` がやってることを説明すると、まず必要な state やロジックのためのメンバーメソッドを持つクラスコンポーネントを生成してる。そして `render` メソッドの中で、`withCounter` が引数として受け取ったコンポーネントに props としてそれらを渡してあげてるわけ。そうやって元のコンポーネントが compose されたクラスコンポーネントが戻り値として返される。結果、`<Counter maxCount={100} />` のようなインターフェースのカウンターコンポーネントができあがるの」

「すごい。めっちゃ頭のいい人が考えたテクニックですね。よくこんなの思いつくなあ」

「mixins とちがって状態とロジックをきれいに切り分けられるので、HOC は Redux を始めとする一連のメジャーなライブラリのインターフェースにも採用されていった。

またさっきのサンプルのように自前で包含するクラスコンポーネントを書かなくても、関数コンポーネントに state やライフサイクルメソッドを始めとする便利な機能を追加してくれる HOC ライブラリの Recompose[7] というプロダクトが開発され、意識の高い開発者たちの間で広く使われるようになった」

「そのリコンポーズ？　って柴崎さんも使ってたんですか？」

「うん、かなりのヘビーユーザーだったよ。クラスコンポーネントを隠蔽してくれて、すべてを関数コンポーネントで書くのに便利だったからね。一時期は Recompose なしで React アプリを開発するなんて考えられなくなってたくらい」

「へー。ちなみに Hooks 全盛の今日で HOC を使う場面ってあるんですか？」

「API のインターフェースに HOC を使ってたライブラリはほとんど Hooks に移行したからね。HOC イ

---

[6] 「① 言語・環境編」「4-5-3. 組み込みユーティリティ型」を参照のこと。

[7] https://github.com/acdlite/recompose

ンターフェースを残してるものもあるけど、あえてそれを使う理由はないだろうね。ただ既存のコンポーネントにレイアウトやユーザー認証を適用するのに HOC を使うケースを見かけたりするので、限定的ではあるけどまだ使われてるよ」

「そうなんですね」

「話を当時に戻そう。HOC は軒並みメジャーなライブラリのインターフェースとして採用され、Recompose が普及して一世を風靡したと言っていいほどだった。でも公式はほどなく HOC を推すのをやめて別の新しい手法を推奨するようになったの」

10-1-3. HOC の対抗馬 Render Props

「HOC を置き換えるものとして公式が推すようになったパターン、それが render props だね」

「レンダープロップス？　クラスコンポーネントの `render` メソッドとはちがうんですか？」

「`render` メソッドとは分けて考えて。render props とは React Elements を返す関数を props として受け取って、それを自身のレンダリングに使う特殊なコンポーネントを使った手法なの。レンダリングのための関数を props として受け取るから render props という」

「えーっと、ちょっと整理させてください。HOC が任意のコンポーネントを引数として受け取って、その戻り値にコンポーネントを返す関数。render props は React Elements を返す関数を props として受け取ってそれを自身のレンダリングに利用するコンポーネント、ってことでいいんでしょうか？」

「おおむね正しいよ。ただ正確には render props とはその特殊なコンポーネントのことじゃなく、受け渡されてレンダリングに使われる props のほうを指す言葉ね。まああまりそのへんは厳密じゃなくて、ざっくりとこの手法を render props パターンと呼ぶことのほうが多いけど。

じゃあ実際のコードで理解するために、さっきの HOC で `Hello` コンポーネントに `target` を注入してたサンプルを render props を使って書き直してみようか。まず `Hello` コンポーネントはこういう内容だったよね」

```
type Props = { target: string };
const Hello: FC<Props> = ({ target }) => <h1>Hello {target}!</h1>;
```

「`Hello` はコンポーネントだけど、React Elements を返す関数でもある。ここから render props で `target` に任意の文字列を与える `TargetProvider` というコンポーネントを作ってみるよ。まず JSX からの使い方としてはこうしたいわけね」

```
<TargetProvider render={Hello} />
```

「`render` という props に `Hello` コンポーネントが渡されてる？」
「そう。そしてこの `TargetProvider` の実装はこうなる」

```
const TargetProvider: FC<{ render: FC<Props> }> = ({ render }) => render({ target: 'Patty' });
```

「ええーっと、このコンポーネントは `Hello` コンポーネントを `render` という名前の props として受け取り、その props の `target` に `'Patty'` を設定して返してるわけですか。うーん、HOC よりさらにわかりにくくなってません？　なぜこんなわけがわからないことをやるんでしょうか」
「それを答える前に、さっきのカウンターコンポーネントも render props で実装し直しておこうか。
　今度は `render` という props を使わず `children`、つまり子要素を利用してみる。render props は必ずしも `render` という名前の props を使う必要はなくてね。そのコンポーネントが自身のレンダリングのために使うコンポーネントの props は、何だろうと技術的には render props と呼ぶの[8]」

リスト 3: render props によるカウンターの実装

```
import { Component, FC } from 'react';

type ChildProps = {
  count: number;
  reset: () => void;
  increment: () => void;
};

type Props = {
  maxCount: number;
  children: (props: ChildProps) => JSX.Element;
};
type State = { count: number };

class CounterProvider extends Component<Props, State> {
  constructor(props: Props) {
    super(props);
    this.state = { count: 0 };
  }

  #increment = () => this.setState((state) => ({ count: state.count + 1 }));
```

[8] https://stackblitz.com/edit/react-ts-ztznn3?file=Counter.tsx

```
  #reset = () => this.setState({ count: 0 });

  componentDidUpdate(): void {
    if (this.state.count > this.props.maxCount) this.#reset();
  }

  render = (): JSX.Element =>
    this.props.children({
      count: this.state.count,
      reset: this.#reset,
      increment: this.#increment,
    });
}

const Counter: FC<{ maxCount: number }> = ({ maxCount }) => (
  <CounterProvider maxCount={maxCount}>
    {({ count, reset, increment }) => (
      <div>
        <div>
          {count} / {maxCount}
        </div>
        <button onClick={increment} type="button">
          +1
        </button>
        <button onClick={reset} type="button">
          Reset
        </button>
      </div>
    )}
  </CounterProvider>
);

export default Counter;
// SEE: https://stackblitz.com/edit/react-ts-ztznn3?file=Counter.tsx
```

「render props も根本の発想は HOC と同じで `count`、`reset`、`increment` という props に外側のコンポーネントから状態やロジックを注入したいわけね」

「なんとなくそれはわかります。それで、どうして HOC があるのに公式は render props のほうを推すようになったんでしょうか？」

「それに関して、実は React のコアチーム内でも意見は分かれてたぽいね。開発者コミュニティでも HOC 派と render props 派に分かれてしばらく論争になってたの。その render props 派が、HOC より render props

のほうが優れていると主張する理由は次のようなものだった」

- HOC のように props の名前の衝突が起こりづらく、起こったとしてもコードから一目瞭然
- TypeScript 使用時、props の型合成が必要ない
- どの変数が機能を注入するための props として親コンポーネントに託されたのかがコードから判別しやすい
- コンポーネントの外で結合させる必要がなく、JSX の中で動的に扱うことができる

「ちなみに render　props 派の急先鋒は React のもっともメジャーなルーティングライブラリである React Router の開発者 Michael Jackson で、いま挙げたのも彼の書いた記事『Use a Render Prop!』[9] の要約ね」
「マイケル・ジャクソン！？」
「あ、『スリラー』や『ブラック・オア・ホワイト』が代表作の超有名なアーティストとは同姓同名の別人だからね。たしかに彼の言うように HOC は依然として props の名前衝突の危険性があったし、props の型合成はめんどくさかった。でも render props のほうがわかりやすいかというのは意見が分かれるところで、私個人は render props のほうがわかりにくいと思うよ。render props は JSX の階層を不用意に深くしてしまう。render　props を複数適用した際のコードは見通しが著しく悪くなり、容易にラッパー地獄（wrapper hell）に陥ってしまいがち」
「そんなたくさんの render props をひとつのコンポーネントで使うことってあったんですか？」
「render　props は公式が推しただけあって、メジャーなライブラリのインターフェースにけっこう採用されたからね。先述した React Router やフォーム管理ライブラリの Formik[10]、GraphQL ライブラリの Apollo Client[11] なんかが代表的だったけど、読みづらいったらないし TypeScript の環境では引数の型指定がわずらわしいし、ずっと早く滅びてくれないかなと思ってた」
「あはははは」
「コミュニティにおける HOC 派の代表格は、Recompose の作者だった Andrew Clark。彼は当時、render props の優位性を主張して HOC を批判する人たちを相手にこんなツイートを残してる」

> **HOCs aren't great, but render props also suck**
> （訳：HOC がすばらしいとはいえないが、render props だって同様にクソだ）

---

[9] 「Use a Render Prop!. Render Props are a powerful way to … | by Michael Jackson | componentDidBlog」 https://medium.com/[@mjackson/use-a-render-prop-50de598f11ce]

[10] https://formik.org/

[11] https://www.apollographql.com/apollo-client/

> **Andrew Clark** @acdlite・ 2018 年 8 月 23 日

「へー。彼、相当 render props が嫌いだったんですね」

「この論争は render props 派がやや優勢に立ちながらも、結局はっきりと決着がつかないまま、突如として公式が提示した画期的な技術の登場によって 2 つとも瞬く間に表舞台から追いやられてしまった。その画期的な技術というのが Hooks だね。これについてはまたドラマチックなエピソードがあるので、そこから話していこう」

10-1-4. ついに Hooks へと至る

「2022 年現在、まだ React の公式ドキュメントには HOC[12] も render props[13] もそれぞれの項目が立派に存在してるので、mixins とちがって完全に過去の技術というわけじゃない。HOC が現在も使われてる例を挙げたけど、Formik のドキュメントでは Hooks より render props を使った例を最初に掲げてる。

でもコミュニティ全体を見渡せばあくまでもそれらは限定的なケースであって、Hooks が今の主役の技術なのは誰が見ても明らかだね。それほどまでに Hooks 登場のインパクトとその後の React エコシステムに与えた影響は甚大だった」

「さっき、Hooks の登場にはドラマチックなエピソードがあるって言ってましたよね。どういう話なんですか？」

「うん。Recompose 作者で HOC 支持派の先鋒だった Andrew Clark なんだけど、実は Meta にスカウトされて 2016 年 9 月に入社し、正式に React 開発コアチームにジョインしたの。そこで彼が大きな役割を果たして作られた技術が Hooks だったのね。Hooks の最初の設計者は Sebastian Markbåge だけど、コントリビュートの量では Andrew Clark もトップクラス[14]。いうなれば、彼は自らの手で Hooks を生み出して render props もろとも HOC を葬り去ったわけ」

「ええっ、そうだったんですか？」

「この、在野の優れたライブラリ作者を公式チームに招き入れ、中から React の改良に尽力してもらうというのは Meta の十八番の OSS 戦略なんだよね。ちなみに Redux のオリジナル作者の Dan Abramov も

---

[12] 「高階 (Higher-Order) コンポーネント」 https://ja.reactjs.org/docs/higher-order-components.html

[13] 「レンダープロップ」 hhttps://ja.reactjs.org/docs/render-props.html

[14] 「フックの先行技術にはどのようなものがありますか？ – フックに関するよくある質問 –React」 https://ja.reactjs.org/docs/hooks-faq.html#what-is-the-prior-art-for-hooks

2015 年 11 月に Meta の React コアチームに参加してる」

「へえ ——。Meta、うまいことやりますねえ」

「Hooks が最初に発表されたのは 2018 年 10 月に開催された ReactConf 2018 での Dan Abramov よる基調講演 [15] の中だったんだけど、そのときはまだ α 版だったというのにコミュニティの反響がすごく大きくてね。直後に Hooks を使ったライブラリが雨後の竹の子のように出てきて、さらにメジャーなライブラリも将来的に Hooks に対応することを次々に表明しだしたの」

「ふむふむ。でもそうなると、Andrew Clark さんが開発してた HOC ライブラリの Recompose はどうなったんですか？」

「React 公式チームから Hooks の発表があった当日、彼は Recompose の GitHub ページにこの文章を掲載したのね」

A Note from the Author (acdlite, Oct 25 2018)

Hi! I created Recompose about three years ago. About a year after that, I joined the React team. Today, we announced a proposal for Hooks. Hooks solves all the problems I attempted to address with Recompose three years ago, and more on top of that. I will be discontinuing active maintenance of this package, and recommending that people use Hooks instead.

（訳： 私は 3 年前に Recompose を開発しました。さらにその翌年、React チームにジョインしました。そして本日、私たちは Hooks をアナウンスする運びとなりました。Hooks は私が 3 年前に Recompose で対処しようとしたすべての問題を解決してくれるものであり、これからは Recompose ではなく Hooks を使うことをおすすめします）

「ええー！？ Hooks の発表と同時にこれですか？」

「まあ彼はその前の 2 年間、Recompose の新規機能の追加にはノータッチだったらしいし、計画的に進めたことなんだろうね。でも当時、私はかなりのショックだったなあ。そのとき携わってたプロジェクトすべてで Recompose を使ってたし」

「それは地味につらいですね……」

「まあ、Web フロントエンドはまだまだ解決するべき問題が山ほどあってイノベーションの余地が多い世界だからね。特に React はその最前線にあって、外部のコミュニティから人や技術を取り入れて常に進化を続けているプロダクトだし。それによって確実に改善されてきてるからこそ React は開発者からの圧

---

[15] 「React Today and Tomorrow and 90% Cleaner React With Hooks」
https://www.youtube.com/watch?v=dpw9EHDh2bM

倒的な支持をずっと得つづけているわけで、その恩恵を受けるために甘んじて引き受ける必要があるコストだと思うしかないかな」

「それでも Hooks の全貌を見て、柴崎さんはそのコストを払ってまで乗り換える必要がある技術だとすぐに判断したんですよね。柴崎さんにそう思わせた Hooks というのはどういう技術だったんですか？」

「そうだね。HOC や render props というのはあくまで既存の技術を利用した**設計パターン**だったわけだけど、Hooks はそうではなくて公式が新たに **React の機能**として提供したものだった。状態やロジックの分離を小手先のテクニックではなく、そのための新しいしくみを仮想 DOM の外に用意したものなのね。だから HOC か render props かといった次元のテーマにとどまらず、既存の React が根本的に抱えてた問題をエレガントに解決してみせたところが衝撃的だったわけ」

「……というと？」

「順を追って説明するね。まず HOC や render props が共通して抱えてた問題とは、ロジックの追加がコンポーネントツリーを汚染してしまうこと。追加するロジックの分だけコンポーネントの階層が深くなってしまう。render props ではそれがコード上でも顕著だったけど、HOC も React Developer Tools で仮想 DOM の階層構造を見れば変な名前のコンポーネントが幾層にも覆いかぶさってるのがわかる。これはシンプルではないし、処理の流れを追いづらくしてしまう。Hooks はコンポーネントにロジックを抱えた別のコンポーネントをかぶせるのではなく、コンポーネントシステムの外に状態やロジックを持つ手段を提供したわけ」

「ふむふむ」

「また HOC も render props も状態を持つロジックを分離はできても、けっきょく積極的に再利用できるほどには抽象化できなかった。でも Hooks を使えば、状態を持ったロジックを完全に任意のコンポーネントから切り離し、それ単独でテストしたり、別のコンポーネントで再利用することが簡単にできるようになる。コンポーネントの階層構造を変えることなしに、状態を伴った再利用可能なロジックを追加できるというのが画期的だったの」

「ふーむ、なるほど」

「そして当初より React が根本的に抱えてた大きな問題が他にもあった。それはクラスコンポーネントの存在。React にとって理想のコンポーネントは純粋関数であるはずなのに、state やライフサイクルメソッドはクラスコンポーネントしか持つことができなかったため、どうしてもクラスコンポーネントを使わざるをえなかった。でも Hooks がコンポーネントシステムの外に状態やロジックを持つしくみを提供したことによって、ほぼ関数コンポーネントだけでアプリケーションが作れるようになったの。というか Hooks は関数コンポーネント内でしか使えない、関数コンポーネントを強化するための機能なんだけどね」

「なるほど。ところで『Hooks』ってどういう意味で名付けられた言葉なんですか？」

「公式には説明されてないんだけど、私の解釈ではコンポーネントに外から状態やそれらのロジックを紐付ける、つまり『引っかけて』おくしくみを提供するというニュアンスがあるんだと思う。引っかけておけば、後は React がしかるべきタイミングでしかるべき処理をしてくれるので、React らしいより宣言的な手法と言える」

「出た、宣言的！」

「実際、Algebraic Effects から発想を得ていたり[16]、メモ化のテクニックが多用されていたりと、関数型プログラミング色のかなり濃い技術だからね。React は UI を宣言的に記述すること、そのためにコンポーネントを純粋関数で表現することを理想としてきた。エコシステムを巻き込んだ試行錯誤の末に行き着いたひとつの解が Hooks なわけ」

「柴崎さん、なんかほぼ絶賛ですね。Hooks が普及した後から React に入門できたのは恵まれてるかも」

「そうだよ。私なんか何回、学んでは捨てを繰り返したことか……」

10-1-5. Custom Hook でロジックを再利用しやすくする

「ここまでコンポーネントからロジックを分離、再利用するための技術の系譜の行き着いた先が Hooks という話をしてきた。でもまだ Hooks を関数コンポーネントに状態や副作用といったロジックを追加するためにしか使ってなくて、それらを分離して再利用できるようにはしてないよね」

「あ、たしかに」

「React では State Hook や Effect Hook を使ったロジックをひとまとめにして独立させ、それをコンポーネントから必要に応じてインポートして利用するようにできる。それを **Custom Hook** という。たとえばサンプルのカウントダウンタイマーのロジックを Custom Hook にして、それをインポートして使うと次のように書ける[17]」

リスト 4: 02-custom-hook/src/components/Timer.tsx

```
import type { FC } from 'react';
import { RepeatClockIcon as ResetIcon } from '@chakra-ui/icons';
import { Box, Button, Stat, StatLabel, StatNumber } from '@chakra-ui/react';
import { useTimer } from 'hooks/useTimer';
```

---

[16] 「我々向けの Algebraic Effects 入門 —Overreacted」
https://overreacted.io/ja/algebraic-effects-for-the-rest-of-us/

[17] https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/tree/main/10-techniques/02-custom-hook

```
type Props = { maxCount?: number };
const MAX_COUNT = 60;

const Timer: FC<Props> = ({ maxCount = MAX_COUNT }) => {
  const [timeLeft, reset] = useTimer(maxCount);

  return (
    <Box p={5} w="sm" borderWidth="1px" borderRadius="lg" boxShadow="base">
      <Stat mb={2}>
        <StatLabel fontSize={18}>Count</StatLabel>
        <StatNumber fontSize={42}>{timeLeft}</StatNumber>
      </Stat>
      <Button
        w="xs" colorScheme="red" variant="solid" leftIcon={<ResetIcon />}
        onClick={reset}
      >
        Reset
      </Button>
    </Box>
  );
};

export default Timer;
```

「えっ、こんなにシンプルになっちゃうんですか？」

「そう、Hooks を使って書いてた部分をごっそり抜き出したからね。なお Custom Hook を作る際にひとつ守らなきゃいけない決まりがある。それは Custom Hook 関数の名前の頭に『use』をつけること」

「なるほど、`useState` や `useEffect` みたいな公式の Hooks API の命名規約を踏襲してるんですね」

「そう。それを守らないと絶対に動かないってわけじゃないんだけど、その関数が Custom Hook なのかをひと目で判断できるよう React の公式チームが課してる規約なのね。`react-hooks/rules-of-hooks` ルールが適用されている環境で Custom Hook の名前を `useXxx` 以外のフォーマットにすると ESLint がエラーにして教えてくれる」

「へー、そこまで eslint-plugin-react-hooks がサポートしてくれるんですか。ところでこの `useTimer()` って、`useState()` みたいに戻り値がタプルで変数や関数を返してるように見えますけど、これも Custom Hook の決まりごとなんですか？」

「ううん。そこは別にルールがあるわけではなくて、どういうフォーマットで返すかは自由に開発者が決められる。たとえばこんなふうに定義して、オブジェクトで返すことだってできるよ」

```
declare function useTimer(maxCount: number): { timeLeft: number, reset: () => void }

const { timeLeft, reset } = useTimer(60);
```

「でもとりあえず今回はタプルで返すようにしておく。次に Custom Hook の本体、`useTimer` の実装を見てみよう。ファイルは `src/hooks/useTimer.tsx` ね」

リスト 5: 02-custom-hook/src/hooks/useTimer.tsx

```
import { useEffect, useState } from 'react';

export const useTimer = (maxCount: number): [number, () => void] => {
  const [timeLeft, setTimeLeft] = useState(maxCount);
  const tick = () => setTimeLeft((t) => t - 1);
  const reset = () => setTimeLeft(maxCount);

  useEffect(() => {
    const timerId = setInterval(tick, 1000);

    return () => clearInterval(timerId);
  }, []);

  useEffect(() => {
    if (timeLeft === 0) reset();
  }, [timeLeft, maxCount]); // eslint-disable-line react-hooks/exhaustive-deps

  return [timeLeft, reset];
};
```

「なるほど。たしかに元々の `Timer` コンポーネントの Hooks 部分を、そのまま丸ごと抜き出した形になってます。エレガントですね！」

「HOC や render props よりずっと読みやすく、無駄なコンポーネント階層もできない、そしてコンポーネントから切り離して再利用しやすい。Hooks の強力さが理解できたでしょ。こうやって Custom Hook でロジックを分離しておくとファイルの数は増えるけど、スッキリと読みやすいし抽出された純粋なロジックの `useTimer` は外でも再利用できる。そしてテストもしやすい」

「納得です。Hooks でコンポーネントからロジックを分離して再利用するってピンとこなかったんですけど、こんなふうに書くんですね」

「作り捨てが最初から決まってるようなアプリケーションならともかく、現代のプロダクトはチームで開発して末永くメンテナンスを続けていくものだからね。こうやってきれいに設計して再利用性やテスタビリティを担保することのほうが、長い目で見れば断然重要になってくるの。秋谷さんも早くこのやり方

に慣れてほしいかな」
「了解です！」

10-2. フォームをハンドリングする

10-2-1. 素の React でフォームを扱うには

「インタラクティブな Web アプリケーションを作るのにフォームは欠かせないよね。次は React でフォームを扱う方法を学ぼう。React でフォームを扱う際に気をつけるべき点は次の 2 つ」

- コンポーネントからイベントを扱う際、React が提供する Synthetic Event が適用される
- フォームの値をコンポーネントの state として持ち、値の反映には state のリフトアップを使う

「ひとつめは Preact の紹介でもちょっとふれたけど[18]、React の仮想 DOM の実装に関係するものね。組み込みコンポーネントの `onClick` や `onChange`、`onSubmit` 属性にコールバック関数を渡すとその引数でイベントが受け取れるんだけど、それがブラウザネイティブの各種 `Event` オブジェクト[19] ではなく、React 独自の `SyntheticEvent` オブジェクト[20] になってる」
「んん？　どういうことですか？」
「たとえば JSX のこういうコードでね」

```
<form onSubmit={(ev) => { ev.preventDefault(); console.log(ev); }>
  <button type="submit">Submit</button>
</form>
```

「このフォーム送信時にダンプされる `ev` が React では `SyntheticEvent` という独自のオブジェクトになってるということ。これはブラウザ間の互換性を吸収するためのもので、仕様的には W3C による標準のイベントに沿ったものになってるので特にクセはないんだけど、TypeScript の環境で引数として受け取るときは `SyntheticEvent` インターフェースを適用するようにしよう」
「よくわからないですけど、わかりました」
「このあとで実際のコードを見れば、すぐピンとくるはずだからだいじょうぶ。なお `SyntheticEvent` は

---

[18] 「8-3-4. Preact」を参照のこと。
[19] 「Event - Web API | MDN」https://developer.mozilla.org/ja/docs/Web/API/Event
[20] 「合成イベント（SyntheticEvent）–React」https://ja.reactjs.org/docs/events.html

各種イベントの基本部分のインターフェースになってて、たとえば change イベント独自のプロパティにアクセスする必要があれば、そこから拡張された `ChangeEvent` を適用する必要があるので随時、適切な型を確認してね。なお公式ドキュメントにはどんなイベントがあるかしか載ってないので、実際の型については @types/react の型定義[21]を見るしかない。他に React TypeScript Cheatsheet というドキュメントにもイベント型の一覧があるので参考になる[22]」

「公式ドキュメント、TypeScript に冷たいですよね……」

「React のフォームの扱いで気をつけるべき 2 つめの点だけど、React が双方向バインディングを用いず、単方向データフローを徹底していることによる制約がある。そろそろ実際のサンプルを見せながら説明したほうがよさそうね[23]」

図 1: React で実装した登録フォーム

「このフォームコンポーネントの実装は次のようになってる」

リスト 6: 03-form/raw-form/src/components/RegistrationForm.tsx

```
import type { FC, ChangeEvent, SyntheticEvent } from 'react';
import { useState } from 'react';
import { Box, Button, ButtonGroup, Checkbox, FormLabel, Input, Select } from '@chakra-ui/react';

const genderCode = { f: '女性', m: '男性', n: 'それ以外' } as const;
```

[21] 「DefinitelyTyped/types/react/index.d.ts・DefinitelyTyped/DefinitelyTyped」
https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/react/index.d.ts

[22] 「Forms and Events | React TypeScript Cheatsheets」
https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/forms_and_events/

[23] https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/tree/main/10-techniques/03-form/raw-form

```
interface FormData {
  username: string;
  zipcode?: string;
  gender?: keyof typeof genderCode;
  isAgreed: boolean;
}

const RegistrationForm: FC = () => {
  const [formData, setFormData] = useState<FormData>({
    username: '',
    isAgreed: false,
  });

  const handleSubmit = (event: SyntheticEvent) => {
    event.preventDefault();
    console.log(formData);
  };

  const handleChange = (event: ChangeEvent<HTMLInputElement | HTMLSelectElement>) => {
    const { name } = event.target;
    const value =
      event.target.type === 'checkbox'
        ? (event.target as HTMLInputElement).checked
        : event.target.value;
    setFormData((state) => ({ ...state, [name]: value }));
  };

  return (
    <Box p={5} w="md" borderWidth="1px" borderRadius="lg" boxShadow="base">
      <form onSubmit={handleSubmit}>
        <FormLabel htmlFor="username" mt={2}>
          ユーザー名
        </FormLabel>
        <Input
          name="username" size="md" value={formData.username} onChange={handleChange}
        />
        <FormLabel htmlFor="zipcode" mt={4}>
          郵便番号
        </FormLabel>
        <Input
          name="zipcode" size="md" maxLength={7}
          value={formData.zipcode ?? ''} onChange={handleChange}
        />
```

```
        <Select
          name="gender" my={6} placeholder="性別を選択…"
          value={formData.gender} onChange={handleChange}
        >
          {Object.entries(genderCode).map(([code, name]) => (
            <option value={code} key={code}>
              {name}
            </option>
          ))}
        </Select>
        <Checkbox
          name="isAgreed" isChecked={formData.isAgreed} onChange={handleChange}
        >
          規約に同意する
        </Checkbox>
        <ButtonGroup my={3} w="xs">
          <Button w="30%" colorScheme="orange" variant="solid" type="reset">
            リセット
          </Button>
          <Button w="70%" colorScheme="blue" variant="solid" type="submit">
            送信
          </Button>
        </ButtonGroup>
      </form>
    </Box>
  );
};

export default RegistrationForm;
```

「なるほど、フォーム全体の値を `FormData` 型のオブジェクトにして state として持ち回るんですね」

「そのとおり。フォームの `onSubmit` 属性に渡してる `handleSubmit` 関数と、それぞれの入力要素の `onChange` 属性に渡してる `handleChange` 関数で定義されてる引数に注目してくれる？」

「あー、引数として受け取ったイベントに `SyntheticEvent` や `ChangeEvent<T>` 型を適用してますね。さっき言ってたのはこれですか」

「そう、お約束なので慣れてね」

「この `handleChange`、受け取ったイベントのターゲットの種類によって抽出する値を振り分け、それを `setFormData` に渡して state を更新してるんですね。ここでも state のリフトアップを使ってるわけですか」

「そのとおり。なおこうやってその値を state で管理されるフォーム要素のことを**『制御されたコンポー**

ネント（Controlled Components）[24]』という。React ではフォームの実装はこの制御されたコンポーネントを使うことが推奨されてるので、特別な理由がない限りはこの手法で実装しましょう」

「そうじゃない特別な理由がある場合は、他の方法が使えるんですか？」

「`ref` というリアル DOM を参照する属性を用いて仮想 DOM を経由せず直接フォーム要素の値にアクセスできるの。この手法が用いられているフォーム要素を『**非制御コンポーネント**（Uncontrolled Components）[25]』という。ユースケースとしてはたとえば、フォーカスが当たったりテキストが選択されたときに何らかのアクションを実行したい場合だったり、サードパーティのフォームヘルパーライブラリを使いたい場合だったりがある」

「ふむふむ、わかりました。ところで `handleSubmit` 関数なんですけど、最初にイベントの `preventDefault` とかいうメソッドを実行してますよね。これって何をやってるんですか？」

「これはね、そのイベント本来の振る舞いを阻止してるの。フォームを submit すると、`action` 属性で指定されている URL、それがなければ現在の URL にフォームの値が POST で送られてページが遷移しちゃうでしょ。SPA でそれをやるとページがトップルートからレンダリングし直しになり、state もすべてクリアされてしまう。それをさせないためにこの 1 行を入れてるわけ」

「あー、なるほど」

「ちなみにこれまでのサンプルでボタン要素の `onClick` 属性に渡してたコールバックも、実は似たような措置が必要なの。安全性を考えれば、こういうふうに書かないといけない」

```
const reset = (event: SyntheticEvent) => {
  event.stopPropagation();
  setTimeLeft(maxCount);
};
```

「`stopPropagation` って何ですか？」

「`onClick` イベントはちょっと特殊で、親要素にも `onClick` イベントが仕掛けてあると子要素の `onClick` イベントが発火したとき連動して親のほうも発火してしまうの。ネストしてると子孫から親に向かってすべてのイベントが発火してしまう。イベント発火が下から泡のように浮き上がってくるので、これをイベントの『Bubbling（バブリング）』と呼ぶ。

この bubbling を防ぐために、イベントには `stopPropagation` メソッドが用意されてるの。propagation は電波や音波の伝播とか、思想や習慣が広まることをいう単語ね」

---

[24] 制御されたコンポーネント - フォーム –React
https://ja.reactjs.org/docs/forms.html#controlled-components

[25] 非制御コンポーネント –Reacthttps://ja.reactjs.org/docs/uncontrolled-components.html

「click イベントにはそういう挙動があったんですね。知りませんでした……」

「React でのフォームの扱いはこんなところかな。ただし実際の現場では、これくらい複雑なフォームをこんなふうに state で値を持ち回したりするケースは少ないのよね」

「えっ。じゃあどうして説明したんですか？」

「まず基本を理解してもらいたかったから。何ごとも基本がわかってないと応用も満足にできないでしょ。便利なツールも中で何をしてるかを自分で理解してこそ使いこなせるものだからね」

10-2-2. フォームヘルパーはどれ使うべきか

「React であるていど複雑なフォームを扱う場合、商用レベルのプロジェクトなら普通はフォームヘルパーライブラリを使う。React 用のフォームヘルパーはけっこうたくさんあって、突出して広く使われてるデファクトスタンダードなライブラリがないので、どれを採用するか迷う人も多いと思う。困ったときの npm trends なので、どんなものがどれくらい使われてるかダウンロード数を見てみよう」

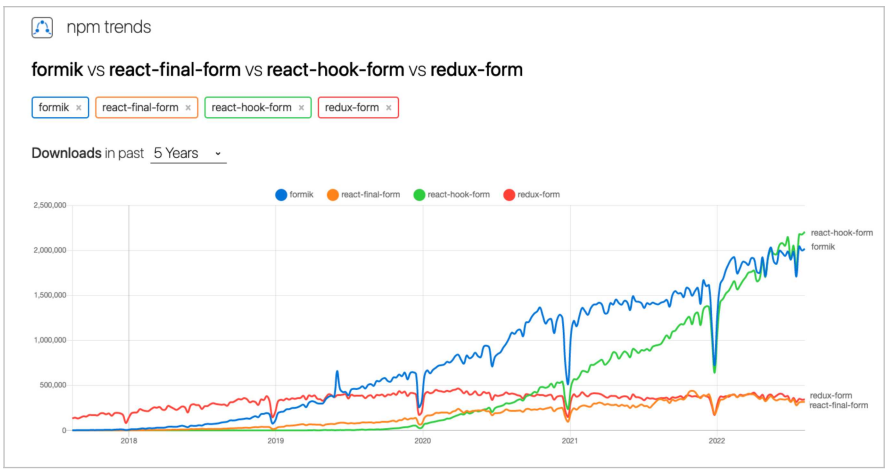

図 2: React 用フォームヘルパーライブラリの DL 数トレンド（2022 年 8 月現在）

「なんか二極化してますね。formik が長らくトップだったのが、直近で react-hook-form に抜かされてます」

「とりあえず古くからあるものから紹介していこうか。最初に出たのが Redux Form[26]。2015 年が初出で

[26] https://github.com/redux-form/redux-form

昔はみんなこれを使ってた。でも別途 Redux[27] を入れてセッティングする必要があるし、またページローカルなフォームの値を Redux のグローバルなストアで管理するのが仰々しく、さらに管理する値が増えると目に見えてパフォーマンスが落ちるため、次第に使われなくなっていった。代わって台頭したのが Formik。フォームの状態管理がローカルに収まり、エラーハンドリングや値のバリデーションもでき、`onChange` や `onSubmit` などのイベントハンドラも提供してくれた。今に続くフォームヘルパーライブラリの基本的な形を作ったのがこの Formik であり、あっというまに Redux Form を抜き去って広く普及した」

「ふむふむ」

「Redux Form の作者が巻き返しを図って出した新しいフォームヘルパーが React Final Form[28]。外部ライブラリへの依存をなくし、TypeScript を意識して型付け機能を強化、バンドルサイズもかなり小さくなった。それらが評価され、React ベースの Ruby on Rails ライクなフルスタックフレームワークの Blitz[29] にも組み込みのフォームヘルパーとして採用された。そしてこの中で最後発のプロダクトが React Hook Form[30]。最初のリリースが 2019 年 3 月だね」

「React Hook Form はそれまでのプロダクトと何がちがったんでしょうか？」

「まずインターフェースが最初から Hooks に対応してた。Hooks の正式リリースは React 16.8 で 2019 年 2 月だったので、早くもその翌月に登場したわけね。Formik も Hooks インターフェースの API を出してはいたんだけど中途半端で、プロダクトとしてはあくまで render　props を用いるのがメインだった。React Final Form に至ってはもっと顕著で、API リファレンスには存在するけど公式ドキュメントのサンプルコードには Hooks による書き方が一切出てこない」

「それは柴崎さん的には大きなマイナスですね……」

「スマホ対応が遅れてモバイルファーストな後発にその王座を奪われたネットサービスってたくさんあったでしょ。あれと同じよね。その名のとおり React Hook Form は Hooks ネイティブなライブラリで、Hooks の急速な普及とともに受け入れられていったわけ」

「なるほど」

「あとはパフォーマンスだね。これまでのフォームヘルパーは制御されたコンポーネントによって成り立っているため、1 文字入力するたびに state が変更されて再レンダリングが走る。React Hook Form はさっきも説明した非制御コンポーネントを使っており、リアル DOM を直接更新するので入力中にコンポーネントの再レンダリングがほとんど起きない。だから React の流儀に反してるのでよくないという意見もあ

---

[27] 「③ React 応用編」の「第 13 章 Redux でグローバルな状態を扱う」で説明します。

[28] https://final-form.org/

[29] https://blitzjs.com/

[30] https://react-hook-form.com/

ったけど、どっちがユーザーに受け入れられたかはさっきのグラフが示してるよね」

「ふーむ、じゃあ柴崎さんのおすすめは断然 React Hook Form なんですね」

「現状、有力な代替候補も見当たらず、他のライブラリを採用する理由がないからね。ただ使うに当たっては React Hook Form が非制御コンポーネントを用いてることは知っておくべき。それによるちょっとしたクセもあるからね」

「わかりました」

「それから、フォーム入力値を検証するために React Hook Form には外部ライブラリのバリデーションメソッドを用いるカスタムリゾルバというしくみがある[31]。本体にもバリデーションの機能はあるけどそれほど使い勝手はよくなくて、専門のスキーマバリデーションライブラリを使ったほうが宣言的に書けて楽なのね。カスタムリゾルバは Yup、Zod、Superstruct、Joi を始め計 11 種類のライブラリ用のものが用意されてる」

「すごい網羅されてますね！　どれを使うのがいいんですか？」

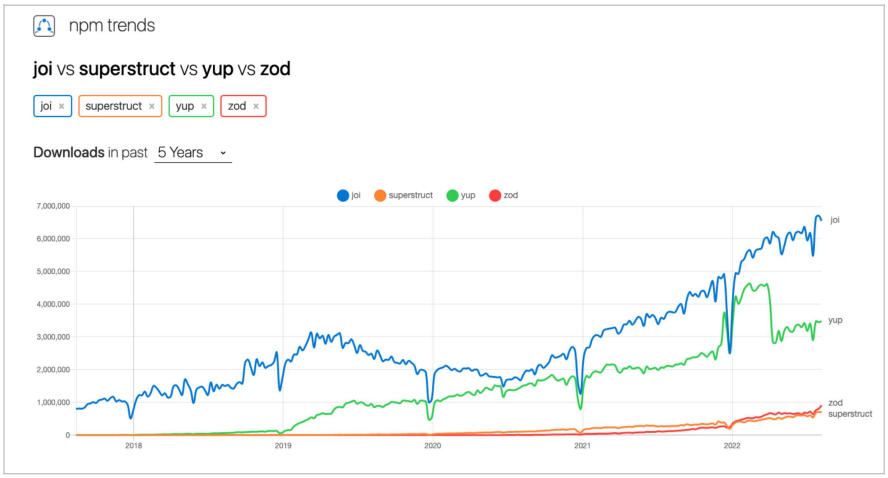

図 3: スキーマバリデーションライブラリの DL 数トレンド（2022 年 8 月現在）

「Yup は Formik 全盛時代から鉄板の組み合わせとして使われてきたので、使い勝手がよく実績もある。Zod はこの中でも最後発のグループにあって Yup の改良版を目指したライブラリ。TypeScript ファーストで型推論も強化されてる。React Hook Form と組み合わせて使われてる事例が多いのはこの 2 つ。比較すると Zod は Yup と逆に項目のデフォルトが required なので、それを optional にする書き方とか、入力値

[31] https://github.com/react-hook-form/resolvers

がなかった場合の判定法やエラーメッセージの与え方がかなり変則的になってしまう。だから Yup かなと」

「変則的って具体的にはどんな感じなんですか？」

「同じスキーマを Yup と Zod で書き分けるとこんな感じになる」

```
// Yup version
const yupSchema = yup.object({
  username: yup.string().required('必須項目です'),
  zipcode: yup.string().max(7).matches(/\d{7}/, '7 桁の数字で入力してください'),
  gender: yup.mixed().oneOf(Object.keys(genderCode)),
  isAgreed: yup.boolean().oneOf([true], '同意が必要です').required(),
});

// Zod version
const zodSchema = z.object({
  username: z.string().min(1, { message: '必須項目です' }),
  zipcode: z.optional(
    z.string().max(7).regex(/\d{7}/, { message: '7 桁の数字で入力してください' })
  ).or(z.literal('')),
  gender: z.optional(
    z.enum(Object.keys(genderCode) as never).or(z.literal(''))
  ),
  isAgreed: z.boolean().refine((val) => val, { message: '同意が必要です' }),
});
```

「たしかに Zod、変態的ですね……」

「だから次から React Hook Form と Yup を組み合わせて使っていくよ」

10-2-3. React Hook Form を使う

「まずは最初に React Hook Form をインストールしておこう」

```
$ yarn add react-hook-form
```

「まずバリデーションを行わず、制御コンポーネントでフォームをハンドリングしていたものを単純に

React Hook Form で置き換えてみよう[32]。先にサンプルとして挙げた登録フォームのコンポーネントを React Hook Form を使って書き換えると次のようになる」

リスト 7: 03-form/hook-form/src/components/RegistrationForm.tsx

```
import type { FC } from 'react';
import { Box, Button, ButtonGroup, Checkbox, FormLabel, Input, Select } from '@chakra-ui/react';
import { useForm } from 'react-hook-form';
import { genderCode } from 'schemas';
import type { SubmitHandler } from 'react-hook-form';

interface FormData {
  username: string;
  zipcode?: string;
  gender?: keyof typeof genderCode;
  isAgreed: boolean;
}

const RegistrationForm: FC = () => {
  const { register, handleSubmit, reset } = useForm<FormData>({
    defaultValues: {
      username: '',
      isAgreed: false,
    },
  });
  const onSubmit: SubmitHandler<FormData> = (data) => console.log(data);
  const onReset = (e: SyntheticEvent) => {
    e.stopPropagation();
    reset();
  };

  return (
    <Box p={5} w="md" borderWidth="1px" borderRadius="lg" boxShadow="base">
      <form onSubmit={handleSubmit(onSubmit)} action="/hoge">
        <FormLabel htmlFor="username" mt={2}>
          ユーザー名
        </FormLabel>
        <Input size="md" {...register('username')} />
        <FormLabel htmlFor="zipcode" mt={4}>
          郵便番号
        </FormLabel>
```

[32] https://github.com/klemiwary/Riakuto-StartingReact-ja4.0/tree/main/10-techniques/03-form/hook-form

```
        <Input size="md" maxLength={7} {...register('zipcode')} />
        <Select my={6} placeholder="性別を選択…" {...register('gender')}>
          {Object.entries(genderCode).map(([code, name]) => (
            <option value={code} key={code}>
              {name}
            </option>
          ))}
        </Select>
        <Checkbox {...register('isAgreed')}>規約に同意する</Checkbox>
        <ButtonGroup my={3} w="xs">
          <Button w="30%" colorScheme="orange" variant="solid" onClick={onReset}>
            リセット
          </Button>
          <Button w="70%" colorScheme="blue" variant="solid" type="submit">
            送信
          </Button>
        </ButtonGroup>
      </form>
    </Box>
  );
};

export default RegistrationForm;
```

「一見して `useForm` っていう Hooks 関数が肝になってるのがわかりますね。これはどういう API なんですか？」

「`useForm` は React Hook Form の心臓部ともいえる API で、引数も戻り値もかなり複雑になってる。メインの目的はフォーム要素を React Hook Form の管理下に置くための関数などを生成することで、引数としてひとつのオブジェクトを取り、そのプロパティで各種オプションを設定するようになってる。主なプロパティは次のとおりで、デフォルト値には『*』をつけておいた」

- **`mode`**：`onChange` | `onBlur` | `onSubmit*` | `onTouched` | `all`
  …… どのタイミングでバリデーションがトリガーされるか。
- **`reValidateMode`**：`onChange*` | `onBlur` | `onSubmit`
  …… エラーのある入力が再度バリデーションされるタイミング。
- **`defaultValues`**：`{ elementName: elementValue }`
  …… 各フォーム要素のデフォルト値。この値を適切に設定していれば型推論が効くため型引数は省略可能。
- **`resolver`**

…… 外部バリデーションライブラリを利用するためのカスタムリゾルバを設定する。

- **shouldUnregister**：`true` | `false`*
  …… 登録していたフォーム要素がアンマウントされると同時に React Hook Form もその入力値を破棄する。
- **shouldFocusError**：`true`* | `false`
  …… フォームが送信されエラーが含まれている場合に、エラーのある最初のフィールドにフォーカスする。

「ちなみにさっきのサンプルでは `defaultValues` プロパティを使ってフォームの初期値を設定してる」
「ふむふむ」
「`useForm` の戻り値はさらに複雑で、そのオブジェクトの中に 15 個のプロパティが含まれてる。よく使うものだけを説明しておくね」

- **register**
  …… フォーム要素を React Hook Form の管理下に置くよう登録するための関数。`ref`、`name`、`onChange`、`onBlur` の属性に対応したプロパティを含むオブジェクトが返される。
- **formState**
  …… フォームの各種状態を検知するためのオブジェクト。以下のプロパティが含まれる。
- `isDarty` …… フォーム内容が初期値から変更されているか
- `dartyFields` …… 初期値から変更されたフォーム要素
- `touchedFields` …… 一度でもユーザーによる操作のあったフォーム要素
- `isSubmitted` …… フォームが送信されたかどうか
- `isSubmitSuccessful` …… フォームの送信が成功したかどうか
- `isSubmitting` …… フォームが送信中かどうか
- `submitCount` …… フォームが送信された回数
- `isValid` …… フォームの内容にバリデーションエラーがないかどうか
- `isValidating` …… フォームの内容をバリデーション中かどうか
- `errors` …… 各フォーム要素に対応したバリデーションエラーの内容が格納されているオブジェクト
- **watch**

…… 各フォーム要素の値を監視し、変更が即時反映された値を返す関数。コンポーネントの再レンダリングを引き起こすため、パフォーマンス悪化につながらないよう使用には注意が必要。

- **`handleSubmit`**
  …… 引数として関数を受け取り、フォームが送信されたときにフォームデータをその関数に渡して実行する高階関数。
- **`reset`**
  …… フォームの状態をすべてリセットする関数。引数のオプション指定で部分的かつ条件をカスタマイズしたリセットが可能。
- **`setError`**
  …… バリデーションの結果に関わらず、手動で任意のエラーを設定できる関数。
- **`clearErrors`**
  …… エラーをクリアする関数。引数で指定した要素のエラーのみクリアすることもできる。

「ここではすべてを説明しきれないので、くわしくは公式サイトの API リファレンスを参照してね [33]」
「……うーん。孫要素まで持ったオブジェクトがあったり、オブジェクトを返す関数があったりと複雑すぎて目が回ります」
「それだけフォーム操作って真面目にやろうとすると、考慮するべきことがたくさんあるってことだよ。直感的で使いやすいインタラクティブなフォームは、裏でこれだけ高度なことをやってるの」
「優雅にスイスイ泳いでいるように見える白鳥も、水面下では必死に足をもがいてるってやつですか……」
「さて、さっきのサンプルではこれらの戻り値の中から `register` と `handleSubmit`、`reset` の関数をピックアップして受け取ってたね。
まずフォーム送信時にその内容をコンソールにダンプする処理を `onSubmit` 関数として用意しているので、受け取った `handleSubmit` 関数にその `onSubmit` 関数を実行させる処理をフォームの `onSubmit` イベントに仕込んでおく。そしてリセットボタンの `onClick` イベントでも同様に `reset` 関数を実行するようにしておく」
「ふむふむ」
「そしていちばん大事なのが `register` 関数によるフォーム要素の登録。この関数の戻り値には `ref` のプロパティが含まれてる。それを展開してフォーム要素の属性として与えることで、非制御コンポーネントとして対応するリアル DOM を React Hook Form が管理下に置くわけ。その部分だけ抜き出して挙げて

[33] https://react-hook-form.com/api/

みると、次の 4 箇所がそうだね」

```
<Input {...register('username')} />
<Input maxLength={7} {...register('zipcode')} />
<Select placeholder="性別を選択…" {...register('gender')}>
<Checkbox {...register('isAgreed')}>
```

「なるほど、これらはそういう意味だったんですね。試しに `register()` に渡してる引数の文字列をでたらめなものに変えてみたら型エラーになりました。これならスペルミスで知らない内に登録失敗することもないですね！」

「なおフォームの構造は `useForm` 実行時に型引数として与えるか、それがない場合は引数オプションの `defaultValues` の設定値から型推論される。今回は前者の方法を使ってるね」

「素朴な疑問なんですけど、`register` 関数の API リファレンスを見ると第 2 引数のオプションに `maxLength` があるじゃないですか[34]。なぜここではそれを使わずに `Input` 要素に直接その属性値を設定してるんですか？」

「その API リファレンスをよく見てみて。`register` 関数の戻り値のオブジェクトに含まれてるプロパティは `onChange`、`onBlur`、`name`、`ref` の 4 つだけでしょ。これら以外の属性は `register` からは設定されない。ここで設定できる `maxLength` はあくまでバリデーションのためのものなのね。HTML ネイティブなフォームデータ検証[35] を実装しないのは React Hook Form の方針なのかもね」

「なるほど」

「入力値のバリデーションはそうやって `register` 関数の引数オプションでも可能だけど、それらをすべて手で設定していくのは煩雑すぎてやりたくない。だからスキーマを用意して宣言的にバリデーションを行うの」

10-2-4. Yup でスキーマバリデーション

「次は Yup を React Hook Form と併用してスキーマによるバリデーションをしてみよう。React Hook Form のバリデーションを Yup によって行う場合、Yup 本体に加えてカスタムリゾルバのパッケージをインストールする必要がある」

---

[34] https://react-hook-form.com/api/useform/register

[35] 「クライアント側のフォームデータ検証 - ウェブ開発を学ぶ | MDN」
https://developer.mozilla.org/ja/docs/Learn/Forms/Form_validation

```
$ yarn add yup @hookform/resolvers
```

「@hookform/resolvers のパッケージの中には、先述した 11 種類のスキーマバリデーションライブラリ用のカスタムリゾルバが含まれてる。じゃあ、まず最初に Yup を使ってスキーマを記述してみる。元々の `FormData` インターフェースはこうだったよね」

```
interface FormData {
  username: string;
  zipcode?: string;
  gender?: keyof typeof genderCode; // ['f', 'm', 'n"]
  isAgreed: boolean;
}
```

「これに `zipcode` は 7 桁の数字のみ、`isAgreed` は `true` しか許容しないといった制限やエラーメッセージ含めて Yup で記述したスキーマはこうなる」

リスト 8: 03-form/hook-form/src/schamas/registrationForm.ts

```
import * as yup from 'yup';
import { genderCode } from './constants';
import type { InferType } from 'yup';

export const regFormSchema = yup.object({
  username: yup.string().required('必須項目です'),
  zipcode: yup.string().max(7).matches(/\d{7}/, '7 桁の数字で入力してください'),
  gender: yup.mixed().oneOf(Object.keys(genderCode)),
  isAgreed: yup.boolean().oneOf([true], '同意が必要です').required(),
});

export type RegFormSchema = InferType<typeof regFormSchema>;
```

「ふむふむ。ぱっと見で何が設定してあるか、だいたいわかりますね」

「Yup が広く使われてるのもそういう理由だろうね。そして Yup が提供する `InferType` という型は、その名のとおり定義したスキーマオブジェクトから型を推論してくれる。VS Code でマウスホバーして型推論の結果を見てみよう」

図 4: Yup の InferType による型推論の結果

「gender が any になってしまってるのは残念だけど、他はおおむね適切だよね。そしてこの Yup で定義したスキーマオブジェクトとその型を用いて、React Hook Form にフォームのバリデーションをさせるためのコードが次のようになる」

リスト 9: 03-form/hook-form/src/components/ValidRegistrationForm.tsx

```
import type { FC, SyntheticEvent } from 'react';
import {
  Box, Button, ButtonGroup, Checkbox, FormControl, FormErrorMessage, FormLabel, Input, Select
} from '@chakra-ui/react';
import { yupResolver } from '@hookform/resolvers/yup';
import { useForm } from 'react-hook-form';
import { genderCode, regFormSchema } from 'schemas';
import type { SubmitHandler } from 'react-hook-form';
import type { RegFormSchema } from 'schemas';

const RegistrationForm: FC = () => {
  const {
    register,
    handleSubmit,
    reset,
    formState: { errors },
  } = useForm<RegFormSchema>({
    defaultValues: {
      username: '',
      isAgreed: false,
    },
    resolver: yupResolver(regFormSchema),
```

```
  });
  const onSubmit: SubmitHandler<RegFormSchema> = (data) => console.log(data);
  const onReset = (e: SyntheticEvent) => {
    e.stopPropagation();
    reset();
  };

  return (
    <Box p={5} w="md" borderWidth="1px" borderRadius="lg" boxShadow="base">
      <form onSubmit={handleSubmit(onSubmit)} action="/hoge">
        <FormControl isInvalid={errors.username !== undefined} isRequired>
          <FormLabel htmlFor="username" mt={2}>
            ユーザー名
          </FormLabel>
          <Input size="md" {...register('username')} />
          <FormErrorMessage>{errors.username?.message}</FormErrorMessage>
        </FormControl>
        <FormControl isInvalid={errors.zipcode !== undefined}>
          <FormLabel htmlFor="zipcode" mt={4}>
            郵便番号
          </FormLabel>
          <Input size="md" maxLength={7} {...register('zipcode')} />
          <FormErrorMessage>{errors.zipcode?.message}</FormErrorMessage>
        </FormControl>
        <Select my={6} placeholder="性別を選択…" {...register('gender')}>
          {Object.entries(genderCode).map(([code, name]) => (
            <option value={code} key={code}>
              {name}
            </option>
          ))}
        </Select>
        <FormControl isInvalid={errors.isAgreed !== undefined}>
          <Checkbox {...register('isAgreed')}>規約に同意する</Checkbox>
          <FormErrorMessage justifyContent="center">
            {errors.isAgreed?.message}
          </FormErrorMessage>
        </FormControl>
        <ButtonGroup my={3} w="xs">
          <Button w="30%" colorScheme="orange" variant="solid" onClick={onReset} >
            リセット
          </Button>
          <Button w="70%" colorScheme="blue" variant="solid" type="submit">
            送信
          </Button>
```

```
        </ButtonGroup>
      </form>
    </Box>
  );
};

export default RegistrationForm;
```

「さっきのサンプルの `App.tsx` からこちらのコンポーネントを呼ぶようにするとバリデーションが有効になる。書き換えて実行してみよう」

図 5: バリデーションエラーが起きた登録フォーム

「あ、すごい。ちゃんと入力エラーになりますね。エラーになった項目に自動的にフォーカスが当たったり、一度送信するとリアルタイムでエラーが消えたり現われたりします」

「そのへんの挙動は先に説明したように `useForm` の引数オプションの `mode` や `reValidateMode`、`shouldFocusError` で調整できるよ。コードに戻って、前とちがうのはまず `useForm` の引数オプションに `resolver` として Yup のカスタムリゾルバを設定してるところ。それから戻り値で `formState` を受け取り、さらにその中の `errors` をピックアップしてる」

「その中にスキーマで検証した入力値のバリデーションエラー情報が入るんですね」

「そのとおり。その `errors` を JSX の中で取り回すんだけど、Chakra UI を併用するときに気をつけないといけないのはフォーム項目を `<FormControl>` で囲みエラーがあるときはその `isInvalid` 属性を `true`

にしておくこと[36]。そうしないとエラーメッセージの表示や自動フォーカスの機能がうまく働かない」
「そうなんですね。でもたったこれだけのコードでこんなインタラクティブなフォームが作れるってすごい！　さわってて楽しいですね」
「これが React Hook Form × Yup の威力だね。これらの機能をライブラリを使わず自前で実装しようとすると大変だよ。フォームは項目が多くなってくるとコードがカオスになってメンテするのも大変になってくるので、ヘルパーやスキーマバリデーションのライブラリを賢く使って見通しのいいコードを書くようにしよう」
「わかりました！」

10-3. コンポーネントのレンダリングを最適化する

10-3-1. コンポーネントのライフサイクル

「クラスコンポーネントの説明をしたとき、コンポーネントにはマウントして初期化され、次にレンダリングされた後、何らかのきっかけで再レンダリングされ、最後にアンマウントされるまでのライフサイクルがあるという話をしたね。そしてそれらの各タイミングに介入して任意の処理を行うために、ライフサイクルメソッドというものが用意されてるというのも[37]」
「はい、おぼえてます」
「併せてライフサイクルの中で各メソッドがいつ実行されるかという図も示したけど、クラスコンポーネントのライフサイクルメソッドは**時間を主眼**に作られているため、視覚化しやすく概念を理解しやすかった。いっぽう Effect Hook を始め各 Hooks 関数は**機能を主眼**にしているため整理して書きやすいいっぽうで、概念を理解するには苦労する。だから自分で書いた Hooks 関数が全体のどのタイミングで実行されるのかイメージしにくい」
「そうですね。私もなんとなくしかわかってない自信があります……」
「React は処理が複雑になってくるとパフォーマンスが落ちやすいので、大規模アプリケーションを開発する際にはそのしくみを深く理解した上でレンダリングを最適化してあげる必要があるの。そのためにはどんなときに再レンダリングが発生し、各 Hooks 関数が実行され、パフォーマンスを劣化させる要因がどこで生じやすいのかをちゃんと知っておかないといけない」

---

[36] 「Form Control - Chakra UI」https://chakra-ui.com/docs/components/form-control
[37] 「② React 基礎編」の「9-5. クラスでコンポーネントを表現する」を参照のこと。

「うーむ、難しそうな話ですね……」
「整理して理解すればそう難しいことはないよ。ところでコンポーネントの再レンダリングが発生するのはどんなときだったかおぼえてる？」
「えーっと、props か state の値が変わったときでしたよね。コンポーネントの二大関心事の」
「おおまかにはそうだね。でも正確には次の 4 つになる」

- props が更新されたとき
- state が更新されたとき
- 親コンポーネントが再レンダリングされたとき
- 参照している context が更新されたとき

「親コンポーネントが再レンダリングされたとき……。そうか、上の階層でコンポーネントの再レンダリングが起きるとぷよぷよの連鎖のように下の階層に向かって再レンダリングが発火するって説明してくれましたもんね。最後の context というのは何ですか？」
「アプリケーションでグローバルな値を保持するために React が用意してるしくみなんだけど、これについては後でくわしく説明するから、今はそんなのがあるんだ程度に考えておいて。それを踏まえた上で、関数コンポーネントのライフサイクルを強引に図にしてみたので見てくれる？」

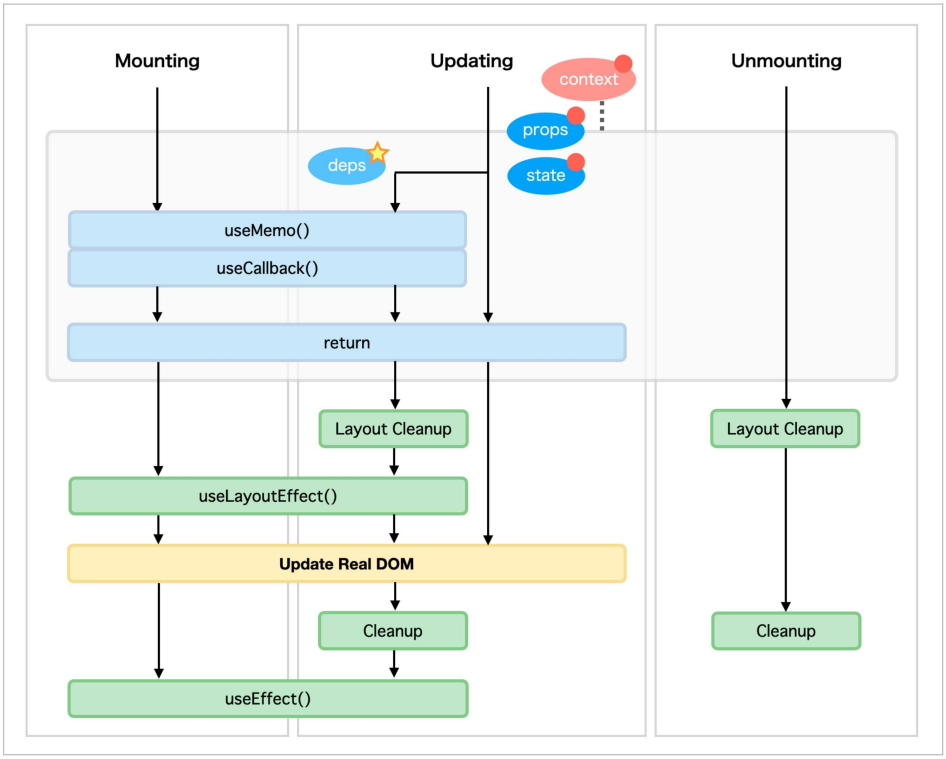

図 6: 関数コンポーネントのライフサイクル図

「まず縦に区分けしたのが 3 つのフェーズ。**Mounting フェーズ**はコンポーネントが初期化され、実行された出力が仮想 DOM にマウントされるまでのフェーズ。このフェーズで初めてコンポーネントがブラウザに表示される。その次が **Updating フェーズ**。さっき挙げた 4 つのケースを React が検知してコンポーネントが再実行される。その出力結果が前回のものと比較され、差分があればその部分のブラウザ表示を差し替える。最後が **Unmounting フェーズ**。コンポーネントが仮想 DOM から削除され、ブラウザ表示からも消える」

「ふむふむ」

「まず Mounting フェーズで何が起こるかから説明していこう。関数コンポーネントが頭から実行され、state が定義されていればその領域が確保される。`useMemo` と `useCallback` という Hooks API についてはこの後すぐ説明するのでいまはスルーさせて。そして実行の結果、React Elements が `return` される。Effect Hook で副作用処理を設定していれば、その後にそれが実行される」

「あれ？ `useEffect` の中に記述された処理って関数コンポーネントの中で実行されるんじゃないんですか？」

「そう考えてる人が多いみたいだけど、実はちがうのよね。`useEffect` に記述された処理はコンポーネン

トが値を返した後、ブラウザへの出力プロセスとは非同期で実行される。つまり先に初期値でブラウザ表示された後、副作用処理によって state が書き換わるとそこから Updating フェーズに移行するの」

「へ――、そうだったんですね」

「だから `useEffect` に記述した副作用処理が中途半端に短時間で終わってしまう場合、UI をちゃんと考慮してないと画面がちらついてユーザーに違和感を与えてしまうことがある。なお Effect Hook にはもうひとつ `useLayoutEffect` という API が用意されてる[38]。こっちはブラウザへの出力プロセスに対して同期的に実行され、副作用処理が終わるまでブラウザへの反映はブロックされる」

「さっきの図にも載ってますね。ちらつきを抑えるためにはこっちを使ったほうがいいんですか？」

「いや画面に何も表示されない時間が長く続くのは UX を大きく損なうので、安易に `useLayoutEffect` を使うのは NG。Web パフォーマンスの計測指標に『FCP（First Contentful Paint）[39]』という視覚コンテンツの初期表示までの時間を表すものがあるけど、`useEffect` の仕様はこの FCP を最小化するためのものなの。

副作用の処理がコンポーネントの表示をブロックしない、というのがレスポンス性を高めてアプリケーション全体での UX の向上につながる。まず `useEffect` を使ってユーザーに違和感を与えない UI を構築して、他に方法がないときだけ例外的に `useLayoutEffect` を用いるようにするべき」

「むー、それって具体的にはどうしたらいいんでしょう？」

「たとえば最近の Web アプリではデータのロード中、部分的にこういうコンテンツのプレースホルダーが表示されるのを見かけない？」

図 7: ローディング中のプレースホルダー

「ああー、たしかに！　これ近ごろよく見るようになりましたね」

「意味のあるコンテンツの表示に時間がかかるのは同じだとして、それが終わるまで何も表示しないの

---

[38] 「useLayoutEffect - フック API リファレンス –React」
https://ja.reactjs.org/docs/hooks-reference.html#uselayouteffect

[39] https://web.dev/first-contentful-paint/

とそれまでとりあえず何かが表示されてるのとでは、後者のほうがマシでしょこの種の UI は処理中だということを待ってるユーザーへ伝えられるのに加えて、レイアウト崩れを極力防げるという利点がある。`useEffect` を使えばこういう UI が簡単に実現できるわけ」

「なるほどー、わかりました」

「Updating フェーズに説明を移そう。図にある『deps』は依存配列に格納した変数のことで、Hooks 関数にこれが渡されその値の更新があった場合となかった場合で挙動が変わる。deps が更新された場合、もしくは依存配列が設定されてなかった場合、`return` 後に副作用処理が実行される。なお Effect Hook にクリーンアップ関数が設定されていれば、副作用処理の前にそれが実行される。

deps が更新されなかった場合、もしくは依存配列が設定されてなかった場合、`return` 後はそのままブラウザ出力につながる」

「クリーンアップ関数って、コンポーネントのアンマウント時だけに実行されるものと思ってました……」

「その誤解も多いけど、たとえばカウントダウンタイマーのインターバル実行で最大カウントを変更して新しいインターバル実行が始まったとき、前のをクリアしとかないとゾンビプロセスとして残っちゃうでしょ」

「そっか、そうですよね」

「そして最後の Unmounting フェーズ。これは Effect Hook に登録していたクリーンアップ関数だけが実行される。React の公式は Hooks リリース後はライフサイクルのメンタルモデルから頭を切り替えるべきとしてこういう図を示さなくなったけど、100 % 正確ではなくても流れを整理して理解するには有効だったでしょ？」

「はい！　おかげさまで、あいまいだった Hooks の実行フローがだいぶわかるようになりました！」

10-3-2. メモ化で計算リソースを節約する

「さっきのライフサイクル図（図 6）に `useMemo`[40] と `useCallback` という見たことのない Hooks API があったよね。あれは不要な再レンダリングを防ぐために提供されてるものなの。これには『**メモ化**（Memoization）』という概念をまず理解しておく必要がある」

「メモ化……。初耳な言葉ですね」

「これも関数型プログラミングの文脈でよく用いられる手法なんだけど、関数内における任意のサブルーチンを呼び出した結果を後で再利用するために保持しておき、その関数が呼び出されるたびに再計算さ

[40] 「useMemo - フック API リファレンス –React」
https://ja.reactjs.org/docs/hooks-reference.html#usememo

れることを防ぐ、プログラム高速化の手法のこと。関数コンポーネントの中に計算リソースを多大に消費する処理が内包されていたとして、結果が同じなのにそれが毎回のレンダリングで実行されるのは無駄だよね。だからパフォーマンス最適化のために、必要なときにだけ計算しその結果を『メモ』しておいて再利用するわけ」

「計算結果をメモしておくから『メモ化』というわけですか」

「そういうこと。具体的なコードがあったほうがわかりやすいよね。Custom Hook で作ったカウントダウンタイマーに、残りカウントが素数のときだけその文字色をピンクにする機能をつけ加えてみよう。まず `src/utils/prime.ts` というファイルを次の内容で作る」

リスト 10: 04-optimize/memoize/src/utils/prime.ts

```
export const getPrimes = (maxRange: number): number[] =>
  [...Array(maxRange + 1).keys()].slice(2).filter((n) => {
    for (let i = 2; i < n; i += 1) {
      if (n % i === 0) return false;
    }

    return true;
  });
```

「`getPrimes` というのが引数で与えた最大値までの素数が格納された配列を生成する関数ね。たとえば `getPrimes(10)` を実行すれば `[2, 3, 5, 7]` が返ってくる。まあそんなに計算リソースを食うアルゴリズムじゃないけど、わかりやすくするためのサンプルなので今回はとりあえずこれを使うよ」

「はい」

「この `getPrimes` を使ってさっき言った仕様になるように、まず Custom Hook の `useTimer` を改造して戻り値に `timeLeft` が素数かどうかという Boolean 値を付け加えてみよう。とりあえずはこんなふうになるはず」

リスト 11: 04-optimize/memoize/src/hooks/useTimer.pre.ts

```
import { useEffect, useState } from 'react';
import type { SyntheticEvent } from 'react';
import { getPrimes } from 'utils/prime';

export const useTimer = (maxCount: number): [number, boolean, () => void] => {
  const [timeLeft, setTimeLeft] = useState(maxCount);
  const primes = getPrimes(maxCount);
  const tick = () => setTimeLeft((t) => t - 1);
  const reset = (event?: SyntheticEvent) => {
    event?.stopPropagation();
    setTimeLeft(maxCount);
```

```
  };

  useEffect(() => {
    const timerId = setInterval(tick, 1000);

    return () => clearInterval(timerId);
  }, []);

  useEffect(() => {
    if (timeLeft === 0) reset();
  }, [timeLeft, maxCount]);

  return [timeLeft, primes.includes(timeLeft), reset];
};
```

「コンポーネント内の 2 行めで変数 `primes` を定義するために `getPrimes` 関数をコールしてるよね。もし `getPrimes()` が非常に計算コストの高い処理を行っていたとすれば、再レンダリングのたびにその処理を行って `primes` を定義し直すのはパフォーマンスに悪影響を及ぼしてしまう。それを防ぐためにメモ化を行うの」

```
- const primes = getPrimes(maxCount);
+ const primes = useMemo(() => getPrimes(maxCount), [maxCount]);
```

「えーっと、`useMemo` がメモ化のための API で、第 1 引数で渡した関数がメモ化されるんですね。第 2 引数って依存配列ですか？」

「そのとおり。この場合は `maxCount` が変わったら計算し直すようにしてる。`useEffect` の場合とちがうのは、依存配列を省略できないこと。再レンダリングのたびに計算されるならメモ化する意味がないからね」

「なるほど」

「この `useTimer` を使って、目的の仕様のタイマーコンポーネントを実装するとこうなる」

リスト 12: 04-optimize/memoize/src/components/Timer.ts

```
import type { FC } from 'react';
import { RepeatClockIcon as ResetIcon } from '@chakra-ui/icons';
import { Box, Button, Stat, StatLabel, StatNumber } from '@chakra-ui/react';
import { useTimer } from 'hooks/useTimer';

type Props = { maxCount?: number };
const MAX_COUNT = 60;
```

```
const Timer: FC<Props> = ({ maxCount = MAX_COUNT }) => {
  const [timeLeft, isPrime, reset] = useTimer(maxCount);

  return (
    <Box p={5} w="sm" borderWidth="1px" borderRadius="lg" boxShadow="base">
      <Stat mb={2}>
        <StatLabel fontSize={18}>Count</StatLabel>
        <StatNumber fontSize={42} color={isPrime ? 'pink.300' : 'black'}>
          {timeLeft}
        </StatNumber>
      </Stat>
      <Button
        w="xs" colorScheme="red" variant="solid" leftIcon={<ResetIcon />} onClick={reset}
      >
        Reset
      </Button>
    </Box>
  );
};

export default Timer;
```

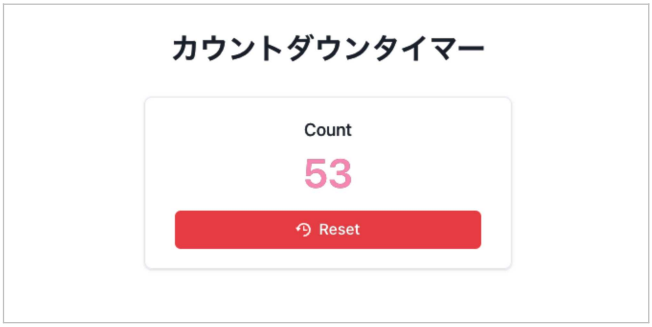

図 8: 素数でピンクになるカウントダウンタイマー

「あ、ちゃんと素数のときだけアホ……じゃなくてピンクになりますね！」

10-3-3. メモ化で不要な再レンダリングを防ぐ

「ところでこのタイマーアプリ、まだ不完全なところがある。数字が切り替わってから、0.9 秒後くらい

著者紹介

大岡由佳（おおおか・ゆか）

技術同人作家。React 専門のフリーランスエンジニアとして複数の現場を渡り歩いた経験を元に執筆した「りあクト！」シリーズが評判を呼びヒット作となる。なお現在は常駐も顧問も請け負っていないため、事実上おそらく国内（世界でも？）に唯一生息する、専業の技術同人作家。
趣味はホームシアターでの映画・ドラマ鑑賞と、3 年以上続けながらちっとも上達しないクラシックバレエ。
Twitter アカウントは @oukayuka。ブログは klemiwary.com。

黒木めぐみ（くろき・めぐみ）

漫画家、イラストレーター。
「りあクト！」シリーズの表紙イラストを一貫して担当。

メールマガジン登録のご案内

くるみ割り書房 BOOTH ページ

「りあクト！」シリーズを刊行している「**くるみ割り書房**」では、新刊や紙の本の再版予定、その他読者の方へのお知らせなどをメールマガジン形式で不定期に配信しています。

購読をご希望の方は、「くるみ割り書房 BOOTH」で検索して最初に表示された上記画像のページにて、「フォロー」ボタンからサークルのフォロー登録をお願いします。BOOTH の一斉送信メッセージにてメルマガの内容をお届けします。

（※ BOOTH のアカウントが必要になります。BOOTH はピクシブ株式会社が運営するオンラインショップです）

りあクト！ TypeScript で始めるつらくない React 開発 第 4 版【①言語・環境編】

現場のエンジニアから多大な支持を受ける『りあクト！ TypeScript で始めるつらくない React 開発』の最新 4 版、全 3 巻構成の第 1 巻「言語・環境編」。

フロントエンド開発において JavaScript および TypeScript のスキルは大事な基礎体力。その基礎を固めつつ、関数型プログラミングの考え方など実際の React 開発において押さえておきたい技術やさらに進んだ書き方までを学んでいきます。

初〜中級者や前の版をお持ちの方にもオススメです。BOOTH にて絶賛販売中！

（2022 年 9 月 10 日発行 ／ B5 版・231 p ／ 電子版 1,200 円、紙＋電子セット 1,380 円）

《第 1 巻 目次》

りあクト！ TypeScript で始めるつらくない React 開発 第 4 版【② React 基礎編】

現場のエンジニアから多大な支持を受ける『りあクト！ TypeScript で始めるつらくない React 開発』の最新 4 版、全 3 巻構成の第 2 巻「React 基礎編」。

本質を理解するため歴史を深堀りして、なぜ React が今のような形になったかを最初に解き明かします。そのうえで JSX やコンポーネント、Hooks を学ぶため、理解度が格段にちがってくるはず。

初〜中級者や前の版をお持ちの方にもオススメです。BOOTH にて絶賛販売中！

（2022 年 9 月 10 日発行 ／ B5 版・188 p ／ 電子版 1,200 円、紙＋電子セット 1,380 円）

《第 2 巻 目次》

りあクト! TypeScript で極める現場の React 開発

『りあクト！ TypeScript で始めるつらくない React 開発』の続編となる本書では、プロの開発者が実際の業務において必要となる事項にフォーカスを当ててました。

React におけるソフトウェアテストのやり方やスタイルガイドの作り方、その他開発を便利にするライブラリやツールを紹介しています。また公式推奨の「React の流儀」を紹介、きれいな設計・きれいなコードを書くために必要な知識が身につきます。BOOTH にて絶賛販売中！

（2019 年 4 月 14 日発行 ／ B5 版・92p ／ 電子版 1,000 円、紙＋電子セット 1,100 円）

《目次》

りあクト! Fiebase で始めるサーバーレス React 開発

個人開発にとどまらず、企業プロダクトへの採用も広まりつつある Firebase を React で扱うための実践的な情報が満載！

Firebase の知識ゼロの状態からコミックス発売情報アプリを完成させるまで、ステップアップで学んでいきます。シードデータ投入、スクレイピング、全文検索、ユーザー認証といった機能を実装していき、実際に使えるアプリが Firebase で作れます。BOOTH にて絶賛販売中！

（2019 年 9 月 22 日発行／ B5 版・136p ／ 電子版 1,500 円、紙＋電子セット 1,600 円）

《目次》

りあクト! TypeScript で始めるつらくない React 開発 第 4 版
【③ React 応用編】

2022 年 9 月 10 日　初版第 1 刷発行
2022 年 9 月 10 日　電子版バージョン 1.0.1

著　者　　大岡由佳
発行者　　大岡由佳
発行所　　くるみ割り書房
連絡先　　oukayuka@gmail.com