React の流儀

巨大で軽快な Web アプリを開発する場合に、React は最高の手段であると、私たちは考えています。Facebook や Instagram といった私たちのサービスにおいても、とてもよくスケールしています。

React のすばらしい特長がいくつもありますが、あなたがどんなアプリを作ろうかと考えたことが、そのままアプリの作り方になる、というのはそのひとつです。本ドキュメントでは、検索可能な商品データ表を React で作っていく様子をお見せしましょう。

モックから始めよう

すでに、JSON API が実装済みで、デザイナーからもデザインモックがもらえているとしましょう。モックは次のような見た目だったとします。

また、JSON API は次のようなデータを返してくるとしましょう。

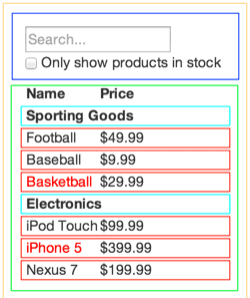
```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

Step 1: UI をコンポーネントの階層構造に落とし込む

まず最初に行うのは、モックを形作っている各コンポーネント（構成要素）を四角で囲んで、それぞれに名前をつけていくことです。もしあなたがデザイナーと一緒に仕事をしている場合は、彼らがすでにこれに相当する作業を終えている可能性がありますので、話をしに行きましょう。彼らが Photoshop でレイヤ名にしていた名前が、最終的にはあなたの React コンポーネントの名前になりうるのです！

しかし、どうやって単一のコンポーネントに括るべき範囲を見つけられるのでしょうか。ちょうど、新しい関数やオブジェクトをいつ作るのかを決めるときと、同じ手法が使えます。このような手法のひとつに、単一責任の原則 (single responsibility principle) があり、これはすなわち、ひとつのコンポーネントは理想的にはひとつのことだけをするべきだということです。将来、コンポーネントが肥大化してしまった場合には、小さなコンポーネントに分割するべきです。

JSON のデータモデルをユーザーに向けて表示することはよくありますので、モデルを正しく構築できているか、UI（つまりコンポーネントの構造）へのマッピングがうまくいっているか、といったことを、皆さんはよく判断できます。これは、UI とデータモデルが同じ **情報の構造** を持つ傾向があるためです。このおかげで、UI をコンポーネントに切り分ける作業は自明なものになりがちです。モックを分割して、データモデルの厳密に一部分だけを表現するコンポーネントへと落とし込みましょう。



5 種類のコンポーネントがこのシンプルなアプリの中にあることが見て取れます。それぞれの解説の中で、データを表すものについては**太字**にしました。

1. **`FilterableProductTable`（オレンジ色）**： このサンプル全体を含む
2. **`SearchBar`（青色）**： すべての**ユーザー入力**を受け付ける
3. **`ProductTable`（緑色）**： **ユーザー入力**に基づく**データの集合**を表示・フィルタする
4. **`ProductCategoryRow`（水色）**： **カテゴリ**を見出しとして表示する

5. **`ProductRow`（赤色）：** 各**商品**を 1 行で表示する

`ProductTable` を見てみると、表のヘッダー（「Name」や「Price」のラベルを含む）が単独のコンポーネントになっていないことがわかります。これは好みの問題で、コンポーネントにするかしないかは両論あります。今回の例でいえば、ヘッダーを `ProductTable` の一部にしたのは、**データの集合**を描画するという `ProductTable` の責務の一環として適切だったからです。しかしながら、将来ヘッダーが肥大化して複雑になった場合（例えばソート機能を追加した場合など）は、`ProductTableHeader` のようなコンポーネントにするのが適切になるでしょう。

さて、モック内にコンポーネントを特定できましたので、階層構造に並べてみましょう。簡単なことです。モックで他のコンポーネントの中にあるコンポーネントを、階層構造でも子要素として配置すればいいのです。次のようになります。

- `FilterableProductTable`
  - `SearchBar`
  - `ProductTable`
    - `ProductCategoryRow`
    - `ProductRow`

Step 2: Reactで静的なバージョンを作成する

HTML CSS Babel Result EDIT ON

```
class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}

class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;

    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
```

Resources View Compiled

さて、コンポーネントの階層構造が決まったので、アプリの実装に取り掛かりましょう。最初は、データモデルを受け取って UI の描画だけを行い、ユーザーからの操作はできないというバージョンを作るのが、もっとも簡単でしょう。表示の実装とユーザ操作の実装を切り離しておくことは重要です。静的な（操作できない）バージョンを作る際には、タイプ量が多い代わりに考えることが少なく、ユーザ操作を実装するときには、考えることが多い代わりにタイプ量は少ないからです。なぜそうなのかは後で説明します。

データモデルを描画するだけの機能を持った静的なバージョンのアプリを作る際には、他のコンポーネントを再利用しつつそれらに *props* を通じてデータを渡す形で、自分のコンポーネントを組み上げていきましょう。*props* は親から子へとデータを渡すための手段です。もし、あなたが *state* に慣れ親しんでいる場合でも、今回の静的なバージョンを作る上では**一切 state を使わないでください。**state はユーザー操作や時間経過などで動的に変化するデータを扱うために確保されている機能です。今回のアプリは静的なバージョンなので、state は必要ありません。

コンポーネントはトップダウンで作っても、ボトムアップで作っても問題ありません。つまり、高い階層にあるコンポーネント（例えば `FilterableProductTable`）から作り始めても、低い階層にあるコンポーネント（`ProductRow` など）から作り始めても、どちらでもいいのです。シンプルなアプリでは通常トップダウンで作った方が楽ですが、大きなプロジェクトでは開発をしながらテストを書き、ボトムアップで進める方がより簡単です。

ここまでのステップを終えると、データモデルを描画する再利用可能なコンポーネントのライブラリが手に入ります。このアプリは静的なバージョンなので、コンポーネントは `render()` メソッドだけを持つことになります。階層構造の中で最上位のコンポーネント（`FilterableProductTable`）が、データモデルを props として受け取ることになるでしょう。元となるデータモデルを更新して再度 `ReactDOM.render()` を呼び出すと、UI が更新されることになります。このやり方なら、複雑なことをしていないので、UI がどのように更新されて、どこを変更すればよいか、容易に理解できることでしょう。React の**単方向データフロー**（あるいは**単方向バインディング**）により、すべてがモジュール化された高速な状態で保たれます。

このステップを実施する上で助けが必要な場合は、React ドキュメントを参照してください。

幕間：Props vs State

React には 2 種類の「モデル」データが存在します。props と state です。このふたつの相違を理解するのは重要なことです。違いについて自信がない場合は、公式の React ドキュメントに目を通すとよいでしょう。

Step 3: UI 状態を表現する必要かつ十分な state を決定する

UI をインタラクティブなものにするためには元となっているデータモデルを更新できる必要があります。これは React なら **state** を使うことで容易に実現できます。

適切に開発を進めていくにあたり、そのアプリに求められている更新可能な状態の最小構成を、最初に考えておいたほうがよいでしょう。ここで重要なのは、DRY (don't repeat yourself)の原則です。アプリケーションが必要としている最小限の状態を把握しておき、他に必要なものが出てきたら、そのとき計算すればよいのです。例えば、TODO リストを作る場合、TODO の各項目を配列で保持するだけにし、個数のカウント用に別の state 変数を持たないようにします。その代わりに、TODO の項目数を表示したいのであれば、単純に配列の length を使えばよいのです。

今回のサンプルアプリを形作るすべてのデータについて考えてみましょう。次のようなデータがあります。

- 元となる商品のリスト
- ユーザーが入力した検索文字列

- チェックボックスの値
- フィルタ済みの商品のリスト

それぞれについて見ていき、どれが state になりうるのかを考えてみます。各データについて、3 つの質問をするだけです。

1. 親から props を通じて与えられたデータでしょうか？ もしそうなら、それは state ではありません
2. 時間経過で変化しないままでいるデータでしょうか？ もしそうなら、それは state ではありません
3. コンポーネント内にある他の props や state を使って算出可能なデータでしょうか？ もしそうなら、それは state ではありません

元となる商品のリストは props から渡されるので、これは state ではありません。検索文字列とチェックボックスは時間の経過の中で変化し、また、算出することもできないため、state だと思われます。最後に、フィルタ済みの商品のリストは state ではありません。何故ならば、元となる商品のリストと検索文字列とチェックボックスの値を組み合わせることで、フィルタ済みの商品のリストを算出することが可能だからです。
というわけで、state と呼べるのは次の 2 つです。

- ユーザーが入力した検索文字列
- チェックボックスの値

Step 4: state をどこに配置するべきなのかを明確にする

HTML CSS Babel Result EDIT ON

```
class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}

class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;

    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
```

Resources View Compiled

さて、state の最小構成が明確になりました。次は、どのコンポーネントが state を変化させるのか、つまり state を**所有**するのかを明確にしましょう。
復習：React は、コンポーネントの階層構造をデータが流れ落ちていく、単方向データフローで成り立っています。もしかすると、どのコンポーネントがどんな state を持つべきなのかが、すぐにはわからないかもしれません。**これは、初学者が React への理解を深める上で、最も難しい問題になりがちなところ** なので、ステップを踏みながら理解していきましょう。
アプリの各 state について、次の各項目を確認していきます。

- その state を使って表示を行う、すべてのコンポーネントを確認する
- 共通の親コンポーネントを見つける（その階層構造の中で、ある state を必要としているすべてのコンポーネントの上位にある単一のコンポーネントのことです）
- 共通の親コンポーネントか、その階層構造でさらに上位の別のコンポーネントが state を持っているべきである
- もし state を持つにふさわしいコンポーネントを見つけられなかった場合は、state を保持するためだけの新しいコンポーネントを作り、階層構造の中ですでに見つけておいた共通の親コンポーネントの上に配置する

それでは、この戦術をサンプルアプリにも適用してみましょう。

- `ProductTable` は商品リストをフィルタする必要があり、`SearchBar` は検索文字列とチェック状態を表示する必要がある
- 共通の親コンポーネントは `FilterableProductTable` である
- 概念的にも、検索文字列とチェック状態が `FilterableProductTable` に配置されることは妥当である

いいですね。state を `FilterableProductTable` の中に配置することが決まりました。では早速、インスタンス変数として `this.state = {filterText: '', inStockOnly: false}` を `FilterableProductTable` の `constructor` に追加して、初期状態をアプリに反映しましょう。その次は、`filterText` と `inStockOnly` を `ProductTable` と `SearchBar` に props として渡します。最後に、これらの props を使って `ProductTable` のフィルタ処理を行い、`SearchBar` のフォームにも値を埋めます。
これで、このアプリがどんな振る舞いをするのか見られるようになってきました。`filterText` に `"ball"` と入力した状態でアプリを更新すると、データの表が正しく更新されたことが確認できるはずです。

Step 5: 逆方向のデータフローを追加する

HTML CSS Babel Result EDIT ON

```
class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}

class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;

    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
```

Resources View Compiled

ここまでで、props と state が階層構造を流れ落ちていく関数として、正しく描画を行うアプリを作ることができました。それでは、別の方向のデータフローもサポートしてみましょう。階層構造の奥深くにあるフォームのコンポーネントが、 `FilterableProductTable` にある state を更新できるようにするのです。

React ではデータフローが明示的になりプログラムの動作が理解しやすくなりますが、従来の双方向データバインディングよりも少しタイプ量が増えてはしまいます。

試しに、現在のバージョンのサンプルで文字を打ち込んだり、チェックボックスを切り替えてみると、React がその入力を無視することがわかります。これは意図的な挙動で、 `input` の `value` props が、常に `FilterableProductTable` から渡された `state` と同じ値になるようにセットしてあるのです。

それでは、どんな挙動になってほしいのかを考えてみましょう。ユーザーがフォームを変更するたびに、ユーザー入力を反映するように state を更新したいですね。コンポーネントの state を更新できるのは自分自身だけであるべきなので、 `FilterableProductTable` は `SearchBar` にコールバックを渡しておいて、state を更新したいときに実行してもらうようにします。入力のたびに呼び出される `onChange` イベントを利用するとよいでしょう。このコールバックを実行された `FilterableProductTable` は、 `setState()` を呼び出し、その結果としてアプリが更新されます。

これは複雑に思えるかもしれませんが、ほんの数行のコードです。そして、これらはデータがアプリの中をどのように流れているのかを、明確に示しているのです。

終わりに

これで React を使ってコンポーネントやアプリケーションを構築するときの考え方が身に付いたのではないでしょうか。それはもしかしたら、あなたのこれまでのやり方よりも、多くのコードを書くことになるかもしれません。しかしながら、コードは書くよりも読むことのほうが多いこと、そしてモジュール化されていて明示的であるコードは非常に読みやすいということを思い出してください。大規模なコンポーネントのライブラリを構築し始めると、この明示性やモジュール化しやすさのありがたみが分かり始めます。そしてコードを再利用できるようになるにつれて、あなたが書くコードの行数は減っていくのです。:)

このページを編集する