Reduxハンズオン資料

イントロダクション

目標

- [ ] Fluxアーキテクチャについて理解する。
- [ ] ReactアプリにReduxを適用する。
- [ ] Reactアプリに永続データ構造を適用する。

事前知識

今回は、前回のReactハンズオンで解説した内容をもとに話をすすめます。具体的には、以下の知識がある前提で話を進めます。

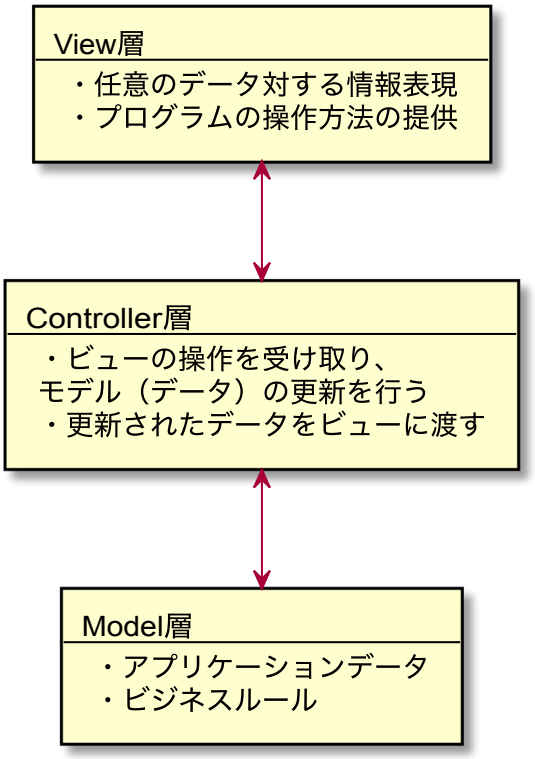
1. Reactについて
2. Visual Studio Codeを使った開発環境の準備
3. create-react-appを使ったプロジェクトを構築
4. JSX記法
5. 関数およびクラスを用いたコンポーネントの作成

Fluxアーキテクチャについて

従来のアーキテクチャ

Fluxアーキテクチャを見ていく前に、MVC/MVP/MVVMなどの従来のアプリケーションアーキテクチャについて確認します。

MVCのアーキテクチャは、プログラムの役割をビュー層・コントローラー層・モデル層に分けて、アプリケーションを設計します。これは現在でも広く使われている一般的な設計手法です。MVCのアプリケーションの構造は一般的に以下のようになります。



MVCは今でも広く使われている設計手法ですが、問題もあります。MVCアーキテクチャでは、Controllerが、ViewとModelの橋渡しをするため、肥大化しやすいです（いわゆるFatController）。また、ModelとViewに対して双方向のデータフロー、ModelとViewが増えると指数関数的に複雑さが増します。

Fluxアーキテクチャとは

Fluxとは

Fluxは、Facebookが提唱しているアーキテクチャですが、同社が提供しているReactのための状態管理ライブラリの名前でもあります。

Fluxはデータフローを **単一の方向に制限する** ことで、データの変更を行う箇所を集約しアプリケーションの状態を制御しやすくします。

Flux自体は、Observerパターンなので理解することはさほど難しくないと思います。

Fluxアーキテクチャの構成要素

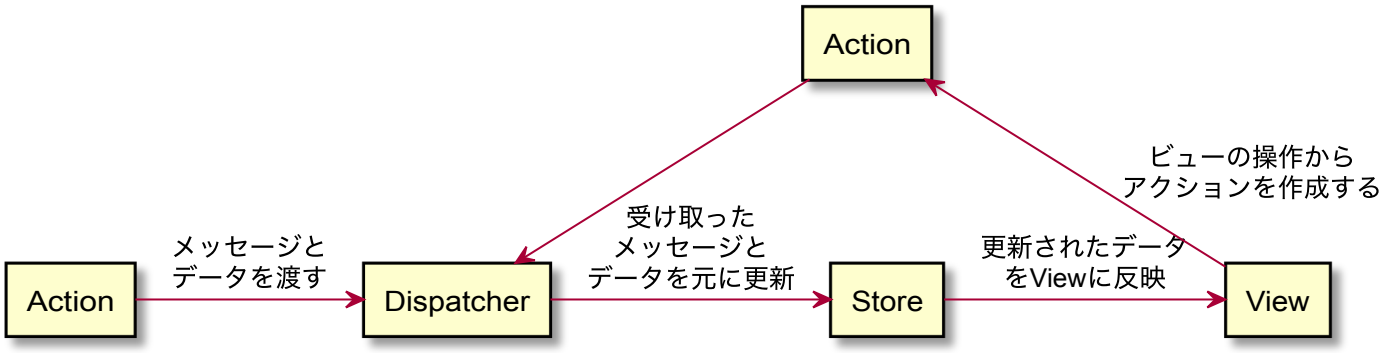
Fluxの構成要素は、 `Action` ・ `Dispatcher` ・ `Store` ・ `View` の4つです。それぞれの意味を以下に示します。

- Action
  - Actionの名前とActionごとのデータを持つオブジェクト
- Dispatcher

    - Actionオブジェクトを Reducerに割り当てる関数
- Store
    - アプリケーションデータの保持
    - データの更新
- View
    - 画面にデータを表示する
    - Reactのコンポーネント

Fluxアーキテクチャのデータフロー

Fluxアーキテクチャのデータフローは次のようになります。Actionを起点に、一方向のデータフローを実現します。基本的には、Observerパターンです。



Reduxによる状態管理

Reduxとは

`Redux` は、Fluxアーキテクチャのライブラリ実装の1つです。他にも `mobX` や `Flux` などといったライブラリ実装があります。今回は、Amaryllisでも使用しているReduxについて取り上げます。

Redux関連パッケージのインストール

`Redux` 単体でも状態管理ライブラリとして利用可能ですが、 `react-redux` パッケージと `redux-actions` パッケージをインストールしておくと、実装が少し楽になりますのでこれらも合わせてインストールします。

npmの場合

```
$ npm install redux react-redux redux-actions
```

yarnの場合

```
$ yarn add redux react-redux redux-actions
```

react-redux と redux-actions はそれぞれ、 React コンポーネントと Redux の状態管理部分を接続するためのヘルパー、 redux-actions は ActionCreator を実装するためのヘルパーです。

ディレクトリの作成

Redux-way に則った react+redux のディレクトリ設計は、以下のように構造で紹介されていることが多いです。

```
├── src
│   ├── compnents
│   ├── containers
│   ├── actions
│   ├── reducers
│   └── stores
```

一方で、 Flux アーキテクチャの要素のうち action ・ reducer ・ store は互いに依存しています。そのため、 action を編集すれば reducer を編集しなければならず reducer の編集が必要であれば store にも変更が必要になります。

そのため、 action ・ reducer ・ store を個別に管理するのではなく、1つのモジュールとしてみなして管理する手法があります。これを Ducks といいます。

Amaryllisにはこの Duck を採用しているため、今回のハンズオンでも Ducks を採用します。 Ducks を使った場合のディレクトリ構造は以下のようになります。

```
├── src
│   ├── compnents
│   ├── containers
│   └── modules
```

TODOアプリの作成

デモアプリとして、以下の機能を持つTODOアプリを作成を取り上げます。

TODOアプリの仕様

デモで作成するTODOアプリは、以下の機能を持ちます。

- 文字列型のテキストフィールドを複数持つ。
- TODOのテキストフィールドを1つ追加するボタンを持つ。

- TODOのテキストフィールドをを削除ボタンを持つ。

機能的には単純で、テキストフィールドを編集することおよびテキストフィールドの数を増減させるのみです。TODOの期限管理や保留・終了・未着手などのステータス管理やDBへの保存は扱いません。しかし、余力があれば挑戦してみるといいと思います。

TODOコンポーネントの作成

まずは、コンポーネントから追加します。今回作成するTODOアプリはTODOの追加ボタンと編集可能なテキストフィールドの配列、テキストフィールドごとの削除ボタンがあります。
ザクッと書くと、以下のようなコードになるかと思います。

```
export function Todo(props) {
    let key = 0
    return (<div style={{ textAlign: 'left', margin: 12 }}>
        <button
            onClick={props.onClickIncrement} >
            TODOを追加する
        </button><br />
        <table style={{ align: 'center' }}>
            <thead>
                <tr>
                    <th>内容</th>
                    <th>操作</th>
                </tr>
            </thead>
            <tbody>
                {
                    props.todo.map((element, index, array) => {
                        return <tr>
                            <td>
                                <input value={element} onChange={props.onChange.bind(this, index
                            </td>
                            <td>
                                <button onClick={props.onClickDecrement.bind(this, index)} key={
                                    削除する
                    </button>
                            </td>
                        </tr>
                    })
                }
            </tbody>
        </table>
    </div>)

}
```

まずトップに追加ボタンがあって、テーブルレイアウトを使って左揃えにテキストフィールドと対応する削除ボタンを表示しています。

コンポーネントとストアの接続

作成したReactコンポーネントと、 redux のストアを接続するには react-redux の connect 関数を使います。 connect 関数は、 state を props に割り当てる mapStateToProps と、 ActionCreator を作成して dispatch 関数で reducer へ通知する mapDispatchToProps の2つの関数を受け取ります。

定義した mapStateToProps と mapDispatchToProps および React コンポーネントを connect 関数に渡すと Redux のストアと接続されたコンテナコンポーネントを作成できます。

```
import * as React from 'react'
import * as ReactDom from 'react-dom'
import { Todo } from '../component/Todo'
import { connect } from 'react-redux'
import * as TodoModule from '../module/Todo'

function mapStateToProps(state) {
    return {
        ...state.todo
    };
}

function mapDispatchToAciton(dispatch) {
    return {
        onChange(index, e) {
            dispatch(TodoModule.onChange({ index: index, value: e.target.value }))
        },
        onClickIncrement() {
            dispatch(TodoModule.onIncrement())
        },
        onClickDecrement(index) {
            dispatch(TodoModule.onDecliment({ index: index }))
        },
    }
}

export default connect(
    mapStateToProps,
    mapDispatchToAciton
)(Todo);
```

ActionCreatorの実装

ActionCreator は、 Action と呼ばれる名前と Action ごとのデータを持つオブジェクトを作成する関数です。 ActionCreator は redux-actions の createAction 関数を使って実装できます。

createAction 関数には Aciton 名を渡し mapDispatchToProps から渡される dispatch メソッドを適用することで、 Reducer を使ってストアを更新できます。

```
//-----------------------
//Action
//-----------------------
export const OnChangeActionName = "Todo/onChange"
export const OnClickInclimentActionName = "Todo/onClickIncrement"
export const OnClickDeclimentActionName = "Todo/onClickDecliment"


//-----------------------
//ActionCreator
//-----------------------

export const onChange = createAction(OnChangeActionName)
export const onIncrement = createAction(OnClickInclimentActionName)
export const onDecliment = createAction(OnClickDeclimentActionName)
```

Reducerの作成

Reducer は、 ActionCreator から渡されたアクション名とアクションごとのデータを受け取って、ストアを更新する関数です。 Reducer は、 redux-actions の handleActions を使って次のように書くことができます。

```
export const TodoReducer = handleActions({
  [OnChangeActionName]: (state, action) => {
    const todo = state.todo.slice(0)
    const { index, value } = action.payload
    console.log(todo)
    todo[index] = value
    return { todo: todo }

  },

  [OnClickInclimentActionName]: (state, action) => {
    const todo = state.todo.slice(0)
    todo.push('')
    console.log(todo)
    return { todo: todo }
  },

  [OnClickDeclimentActionName]: (state, action) => {
    const todo = state.todo.slice(0)
    const { index } = action.payload
    todo.splice(index, 1)
    console.log(todo)
    return { todo: todo }

  },


}, InitialTodoStore)


export default TodoReducer
```

Storeの実装

最後にストアを実装します。ストアは `combineReducer` と `createStore` を使って実装します。 `combineReducer` は複数の `Reducer` を1つにまとめる関数で、 `createStore` は与えられた `reudcer` からストアを作成する関数です。

```
import { createStore, combineReducers, applyMiddleware, compose } from 'redux';
import { handleActions } from 'redux-actions'

//Reducers
import { TodoReducer } from './module/Todo'


const reducers = Object.assign({},
  { todo: TodoReducer });

const appReducer = combineReducers(reducers)

export const Store = createStore(
  appReducer, {});
```

ReduxストアとRectの接続

react-reduxのProviderコンポーネントを使って、実装したストアとreactのコンポーネントを接続します。

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import { Provider } from 'react-redux'
import * as Store from './Store'

ReactDOM.render(
  <Provider store={Store.Store}>
    <App />
  </Provider>, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

参考資料

1. React
2. Redux
3. Flux
4. Amaryllis