10-redux-toolkit

9번 예제에서 다루었던 react-redux + redux-thunk 예제에 redux-toolkit을 적용하여 reducer의 작성 과정을 간결화 시킨 예제.

#01. 프로젝트 생성

```
yarn create react-app 08-redux
```

1) 추가 패키지 설치

프로젝트를 VSCode로 열고, `Ctrl` + `~`를 눌러 터미널 실행

지금까지 살펴본 기본 패키지

```
yarn add react-router-dom qs react-helmet node-sass styled-components axios
```

이번 단원에서 다룰 새로운 패키지

```
yarn add redux react-redux redux-actions redux-devtools-extension redux-logger
redux-thunk @reduxjs/toolkit
```

2) 프로젝트 생성 후 기초작업

1. **src폴더** 하위에서 App.css와 index.css, logo.svg 삭제
2. **App.js** 파일에서 App.css와 logo.svg에 대한 참조(import) 구문 제거
3. **index.js** 파일에서 index.css에 대한 참조(import) 구문 제거
4. index.js 파일에서 다음의 구문 추가

```
import { BrowserRouter } from 'react-router-dom';
```

5. index.js 파일에서 `<App />`을 `<BrowserRouter><App /></BrowserRouter>`로 변경
6. App.js 파일에 다음을 추가

```
import { Route, NavLink, Switch } from "react-router-dom";
```

혹은

```
import { Route, Link, Switch } from "react-router-dom";
```

3) 프로젝트 실행

프로젝트를 VSCode로 열고, `Ctrl` + `~`를 눌러 터미널 실행

```
yarn start
```

#02. 리덕스(Redux)

1) 리덕스 개요

리액트 전역 상태 관리 라이브러리.

일반적인 컴포넌트 개발시에는 상태값(변수)을 관리하기 위해 라이프사이클이나 hooks을 사용한다.

이 경우 각각의 컴포넌트가 관리하는 변수값들이 소스파일 여기저기에 흩어져 있기 때문에 코드 유지보수에 좋지 않다.

컴포넌트의 상태 업데이트 관련 로직을 다른 파일로 분리시켜서 더욱 효율적으로 관리할 수 있다.

즉, 여러 개의 컴포넌트가 개별적으로 관리하는 상태값들을 하나의 소스에 모아 놓고 통합 관리하는 것이 목적.

컴포넌트끼리 상태를 공유해야 할 때도 여러 컴포넌트를 거치지 않고 손쉽게 상태 값을 전달하거나 업데이트할 수 있다.

2) 리덕스 기본 요소

> 각 항목에 대한 설명은 예제 소스의 주석을 참고하세요. `/redux-demo/redux.html`

1. 상태값
2. 액션 - 문자열
3. 액션함수 - 리듀서 호출 함수
4. 리듀서 - 상태값 갱신 함수
5. 스토어 - 상태값 저장소
6. 구독 - 상태값이 변경되었음을 감지하는 기능
7. 디스패치 - 액션함수 호출

```
                  ——— 액션 디스패치 ———▶
[ 컴포넌트(HTML) ]                      [ 리덕스 스토어 ]
                  ◀——— 스토어 상태 ———
```

3) 리덕스 미들웨어

액션을 디스패치했을 때 리듀서에서 이를 처리하기에 앞서 실행되는 사전에 지정된 작업들.

미들웨어는 index.js에서 스토어를 생성하는 과정에서 적용한다.

a) 미들웨어로 수행하는 처리들

- 전달받은 액션을 단순히 콘솔에 기록
- 전달받은 액션 정보를 기반으로 액션을 아예 취소
- 다른 종류의 액션을 추가로 디스패치

b) 동작순서

```
[사용자 이벤트] ——▶ [액션] ——▶ [미들웨어] ——▶ [리듀서] ——▶ [스토어]
```

c) 오픈소스 미들웨어 종류

- redux-logger : 브라우저 콘솔에 로그를 기록하는 기능
- redux-thunk : 비동기 작업을 위한 미들웨어 (주로 Timer, Ajax 등)
- redux-saga : 비동기 작업을 위한 미들웨어 (주로 Timer, Ajax 등)

> redux-thunk 와 redux-saga는 서로 경쟁 상태

#03. React에 Redux 적용하기

순정 Redux를 React에 적용하는 것은 매우 복잡한 처리를 요구하기 때문에 최신 버전에서는 Redux-Toolkit이라는 라이브러리를 통하여 Redux 구조를 단순화 하고 있다.

1) Redux Store 준비하기

상태값, 액션, 액션함수, 리듀서, 스토어가 통합된 형태

a) `/src/store.js`

폴더와 파일을 직접 생성해야 함.

```
import { configureStore, getDefaultMiddleware } from '@reduxjs/toolkit';
import { createLogger } from 'redux-logger';

// Slice 오브젝트 참조 구문 명시 위치

const logger = createLogger();

const store = configureStore({
    // 개발자가 직접 작성한  Slice 오브젝트들이 명시되어야 한다.
    reducer: {
        name: object,
        name: object
        ...
    },
```

```
    // 미들웨어를 사용하지 않을 경우 이 라인 생략 가능
    middleware: [...getDefaultMiddleware(), logger],
    // redux-devtools-extension을 사용하지 않을 경우 false 혹은 이 라인 명시 안함
    devTools: true
});

export default store;
```

2) Redux Store를 React에 구독시키기

`/src/index.js`

리덕스를 위한 참조 추가

```
/** 리덕스 구성을 위한 참조 */
import { Provider } from 'react-redux';
import store from './store';
```

렌더링 처리

렌더링 처리를 `<Provider store={store}>` 태그로 감싼다.

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

3) Slice 모듈 작성

/src/slices/MySlice.js

```
import { createSlice } from '@reduxjs/toolkit'

export const slice이름 = createSlice({
    name: 'slice별칭',
    initialState: {
        // 이 모듈이 관리하고자하는 상태값들을 명시
        변수1: 100,
        변수2: 200
    },
```

```
    reducers: {
        // 상태값을 갱신하기 위한 함수들을 구현
        // 컴포넌트에서 이 함수들을 호출할 때 전달되는 파라미터는 action.payload로 전달
된다.
        // initialState와 동일한 구조의 JSON을 리턴한다.
        액션함수1: (state, action) => {...state},
        액션함수2: (state, action) => {...state}
    },
});

// 액션함수들 내보내기
export const { 액션함수1, 액션함수2 } = slice이름.actions;

// 리듀서 객체 내보내기
export default slice이름.reducer;
```

/src/slice/store.js

정의한 Slice 모듈 명시

```
import { configureStore, getDefaultMiddleware } from '@reduxjs/toolkit';
import { createLogger } from 'redux-logger';

import { slice이름 } from './slices/MySlice';

const logger = createLogger();

const store = configureStore({
    // 개발자가 직접 작성한  Slice 오브젝트들이 명시되어야 한다.
    reducer: {
        slice별칭: slice이름.reducer,
        ...
    },
    // 미들웨어를 사용하지 않을 경우 이 라인 생략 가능
    middleware: [...getDefaultMiddleware(), logger],
    // redux-devtools-extension을 사용하지 않을 경우 false 혹은 이 라인 명시 안함
    devTools: true
});

export default store;
```

4) 컴포넌트에서 사용하기

a) 필요한 기능 참조하기

```
// 상태값을 로드하기 위한 hook과 action함수를 dispatch할 hook 참조
import { useSelector, useDispatch } from 'react-redux'
```

```
// Slice에 정의된 액션함수들 참조
import { 함수1, 함수2 } from '../slices/MySlice';
```

b) 컴포넌트 내부에서 hook을 통해 필요한 Object 생성

```
// hook을 통해 slice가 관리하는 상태값 가져오기
const {변수1, 변수2} = useSelector((state) => state.리듀서이름);

// dispatch 함수 생성
const dispatch = useDispatch();
```

3) 필요한 이벤트 핸들러 안에서 액션함수 디스패치하기

Slice에서 정의한 액션함수의 `action.payload` 파라미터로 전달된다.

다수의 파라미터가 필요한 경우 JSON객체로 묶어서 전달한다.

```
dispatch(액션함수1(파라미터));
dispatch(액션함수2(파라미터));
```

#04. Redux-Thunk를 활용한 비동기 처리

Redux는 기본적으로 동기 처리만 지원하기 때문에 Redux에 Ajax등의 비동기 처리를 추가하기 위해서는 확장 패키지가 필요하다. Redux-Thunk나 Redux-Saga등의 미들웨어가 Redux를 통한 비동기 처리를 가능하게 해 준다.

1) `/src/store.js`

미들웨어 사용 설정에서 동기 처리 체크(serializableCheck)를 하지 않도록 옵션을 추가한다.

```
const store = configureStore({
    // 개발자가 직접 작성한  Slice 오브젝트들이 명시되어야 한다.
    reducer: {
        slice별칭: slice이름.reducer,
        ...
    },
    // 미들웨어를 사용하지 않을 경우 이 라인 생략 가능
    middleware: [...getDefaultMiddleware({serializableCheck: false}), logger],
    // redux-devtools-extension을 사용하지 않을 경우 false 혹은 이 라인 명시 안함
    devTools: true
});
```

2) `/src/slices/MyAsyncSlice.js`

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
import axios from 'axios';

/** 비동기 처리 함수 구현 */
// payload는 이 함수를 호출할 때 전달되는 파라미터.
export const 액션함수 = createAsyncThunk("액션함수별칭", async (payload, {
rejectWithValue }) => {
    let result = null;

    try {
        result = await axios.get(URL및 파라미터);
    } catch (err) {
        // 에러 발생시 `rejectWithValue()`함수에 에러 데이터를 전달하면 extraReducer
의 rejected 함수가 호출된다.
        result = rejectWithValue(err.response);
    }

    return result;
});

/** Slice 정의 (Action함수 + Reducer의 개념) */
export const slice이름 = createSlice({
    name: 'slice별칭',
    initialState: {
        /** 상태값 구조 정의 (자유롭게 구성 가능함) */
        rt: null,           // HTTP 상태 코드(200,404,500 등)
        rtmsg: null,        // 에러메시지
        item: [],           // Ajax 처리를 통해 수신된 데이터
        loading: false      // 로딩 여부
    },
    // 내부 action 및 동기 action (Ajax처리시에는 사용하지 않음)
    reducers: {},
    // 외부 action 및 비동기 action
    extraReducers: {
        /** Ajax요청 준비 */
        [액션함수.pending]: (state, { payload }) => {
            // state값을 적절히 수정하여 리턴한다.
            return { ...state, loading: true }
        },
        /** Ajax 요청 성공 */
        [액션함수.fulfilled]: (state, { payload }) => {
            // state값을 적절히 수정하여 리턴한다.
            return {
                ...state,
                rt: payload.status,
                rtmsg: payload.statusText,
                item: payload.data,
                loading: false
            }
        },
        /** Ajax 요청 실패 */
        [액션함수.rejected]: (state, { payload }) => {
            // state값을 적절히 수정하여 리턴한다.
```

```
            return {
                ...state,
                rt: payload.status,
                rtmsg: payload.statusText,
                item: payload.data,
                loading: false,
            }
        }
    },
});

// 리듀서 객체 내보내기
export default slice이름.reducer;
```

3) `/src/store.js`

추가된 slice의 정보를 기입한다.

```
... 생략 ...

import { slice이름 } from '파일경로';

... 생략 ...

const store = configureStore({
    reducer: {
        slice별칭: slice이름.reducer
    },
    ... 생략 ...
});

... 생략 ...
```

4) 필요한 이벤트 핸들러 안에서 액션함수 디스패치하기

Slice에서 정의한 액션함수의 `action.payload` 파라미터로 전달된다.

다수의 파라미터가 필요한 경우 JSON객체로 묶어서 전달한다.

```
dispatch(액션함수1(파라미터));
dispatch(액션함수2(파라미터));
```