1. React 리뷰

React의 핵심 철학

- 단방향 데이터 흐름
  - 부모 컴포넌트에서 자식 컴포넌트로 속성을 이용해 데이터를 전달
  - 애플리케이션 실행에 대한 추적을 통해 어떤 상황에서 어떤 코드가 실행되는지 쉽게 파악할 수 있음
- 단점
  - 컴포넌트들의 다중 중첩 구조
  - 최상위 컴포넌트의 상태가 자식 컴포넌트로 계층적으로 반복적으로 전달되어야 함. 이 과정에서 오류 발생 가능성이 높아짐

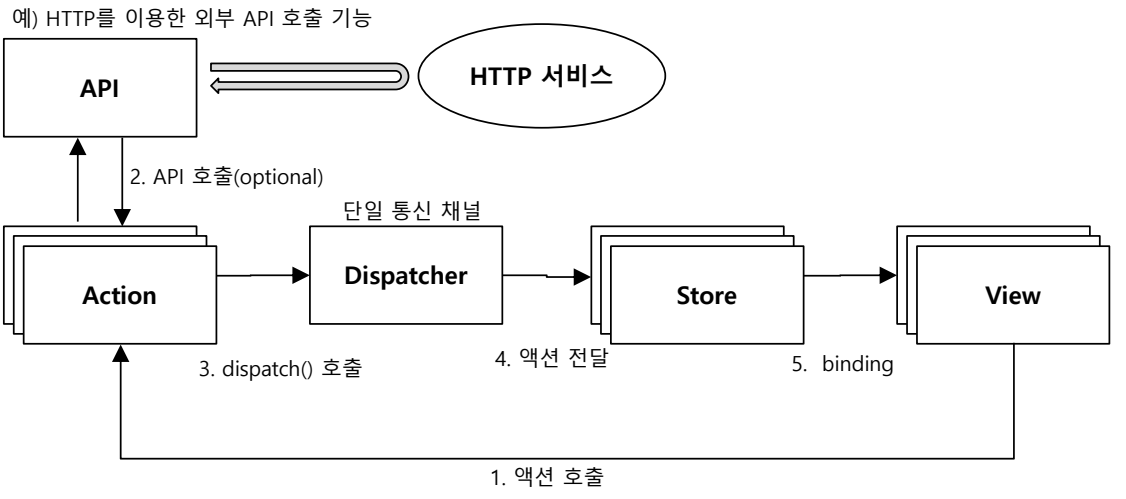
2. Flux란?(1)

플럭스란?

- Facebook에서 클라이언트-사이드 웹 어플리케이션을 만들기 위해 사용하는 어플리케이션 아키텍쳐. 단방향 데이터 흐름을 활용해 View 컴포넌트를 구성하는 React를 보완할 수 있음
- 구성 요소
  - Dispatcher : 단 하나의 디스패처. Actions으로부터 전달받은 메시지를 Store에 전달하는 단일 통신 채널.
  - Stores : 애플리케이션의 상태와 상태를 변경하는 메서드를 보유함. 상태 변경 메서드는 상태의 불변성을 유지할 수 있도록 하는 것이 권장됨.
  - Views : Store의 상태를 UI로 나타내고, Action을 일으킬 수 있는 환경을 제공함.
  - Action : 상태를 변경하는 기능 이외의 비즈니스 로직을 배치함. 비즈니스 로직 실행 후의 결과를 Dispatcher를 거쳐 Store 로 전달하여 상태를 변경함.

2. Flux란?(2)

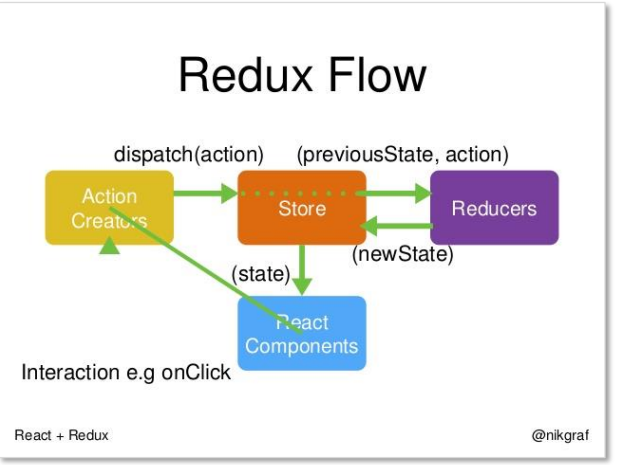
Flux 구조



3. Redux 개요(1)

Redux?

- JS 앱을 위한 예측가능한 상태 컨테이너
  - JS 앱에서 UI상태, 데이터 상태를 관리하기 하기 위한 도구
- Flux의 아키텍처를 발전시키면서 복잡성을 줄임
- React에서만 사용하는 것이 아님.
  - jQuery, Angular, Vue.js 에서도 사용할 수 있음.
- Flux 기능 + hot reloading + time travel debugging



4

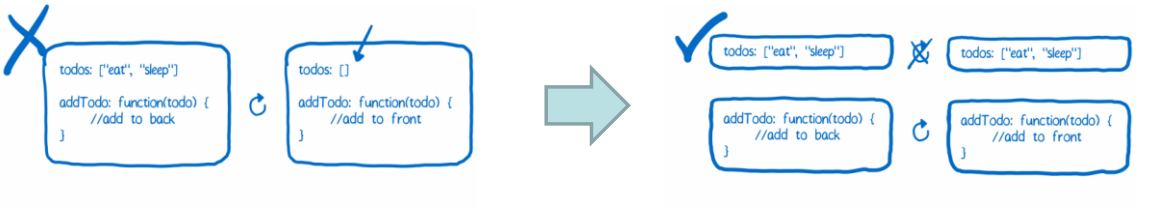
■ 참조

- http://www.slideshare.net/nikgraf/react-redux-introduction
- https://onsen.io/blog/react-state-management-redux-store/

3. Redux 개요(2)

Flux의 문제점

- Store의 코드는 상태를 삭제하지 않고는 reloading이 불가능하다.
  - Flux에서 Store가 관리하는 정보
    - 애플리케이션 상태
    - 상태 변경을 위한 로직
  - Store가 위의 두가지를 모두 가지고 있는 것은 핫 리로딩을 할 때 문제를 일으킴
    - 새로운 상태와 관련한 로직을 위해 Store 객체를 리로딩하면 기존에 저장되어 있는 상태가 날아감. 또한 스토어와 나머지 구성요소와의 이벤트 구독 정보도 유실.
  - 해결 방법 : 두 기능의 분리 -> 상태 + 상태 변경 로직



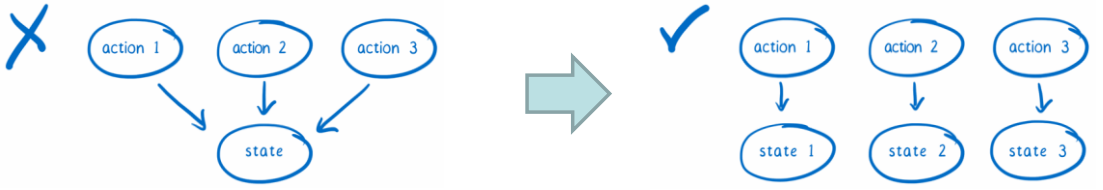
5

■ https://bestalign.github.io/2015/10/26/cartoon-intro-to-redux/

3. Redux 개요(3)

Flux의 문제점(이어서)

- 애플리케이션의 상태는 액션마다 재기록한다.
  - 시간 여행 디버깅(Time Travel Debugging)을 위해서는 이전 애플리케이션 상태를 버전별로 추가할 필요가 있다.
    - 하지만 객체일 때는 단순히 객체의 메모리 참조 정보만 복사할 뿐 deep copy 본을 저장하는 것은 아니다.
    - 제대로 시간 여행 디버깅을 지원하기 위해서는 각각의 버전이 완전히 독립된 객체여야 한다.
  - 해결 방법
    - 액션이 스토어로 전달되었을 때 기존 애플리케이션 상태를 수정하는 것이 아니라 그 상태를 복제한 뒤 복제본을 수정하여 저장한다.(불변성 유지)



■ https://bestalign.github.io/2015/10/26/cartoon-intro-to-redux/

4. Redux 아키텍처(1)

Store

- Redux는 단하나의 Store를 가짐
  - 반면 Flux는 다수의 Store를 가질 수 있다.
- 각 Store는 각자의 범위를 갖고 내부의 상태를 통제한다.
  - 단 하나의 Store를 가지기 때문에 애플리케이션의 전체 상태 트리를 관리하기 힘듬
    - 해야 될 작업을 Reducer에게 위임함.
- Flux의 dispatcher가 Redux에는 존재하지 않음

Action

- Flux의 액션을 그대로 사용
- 애플리케이션의 상태를 변경하고 싶다면 액션을 전송해야 함.(dispatch)
- 액션은 Store를 거쳐 Reducer로 전달되고 Reducer가 상태를 변경한 후 Store로 리턴함.

4. Redux 아키텍처(2)

Reducer

- Reducer는 순수함수(Pure Function)이어야 한다.
  - 순수 함수의 의미는 아래쪽에서 참조.
- 액션이 Store에 전달되면 Store는 이것을 Reducer로 전달함.
- Store는 단 하나이지만 Reducer는 여러개일 수 있음
  - 단 RootReducer를 중심으로 계층 구조를 가짐
  - RootReducer는 상태 데이터의 키를 이용해 처리할 수 있는 Reducer에게 전달함.,
  - 각 Reducer는 불변성을 유지하도록 복제한 뒤 변경사항을 반영해 리턴함.
    - 불변성 관련 라이브러리를 이용해야 함.
    - 이전 상태 객체는 남겨지고 새로운 상태 객체가 생성되어 갱신되므로 변경 사항을 추적할 수 있는 밑바탕이 됨.
- 다중 Reducer
  - 작은 애플리케이션이라면 하나의 리듀서로도 충분하지만 복잡한 애플리케이션은 여러 개의 Reducer를 결합하여 reducer tree를 구성할 수 있음.
    - Flux은 여러개의 Store를 사용할 때 서로 연결될 필요가 없지만 Redux는 Reducer들을 계층적 트리구조로 구성해야 함.

■ 순수 함수의 의미

- 입력인자가 동일하면 리턴값도 동일해야 함
- 부수효과(side effect)가 없어야 함
- 함수에 전달된 인자는 불변성으로 여겨짐. 인자는 변경할 수 없음

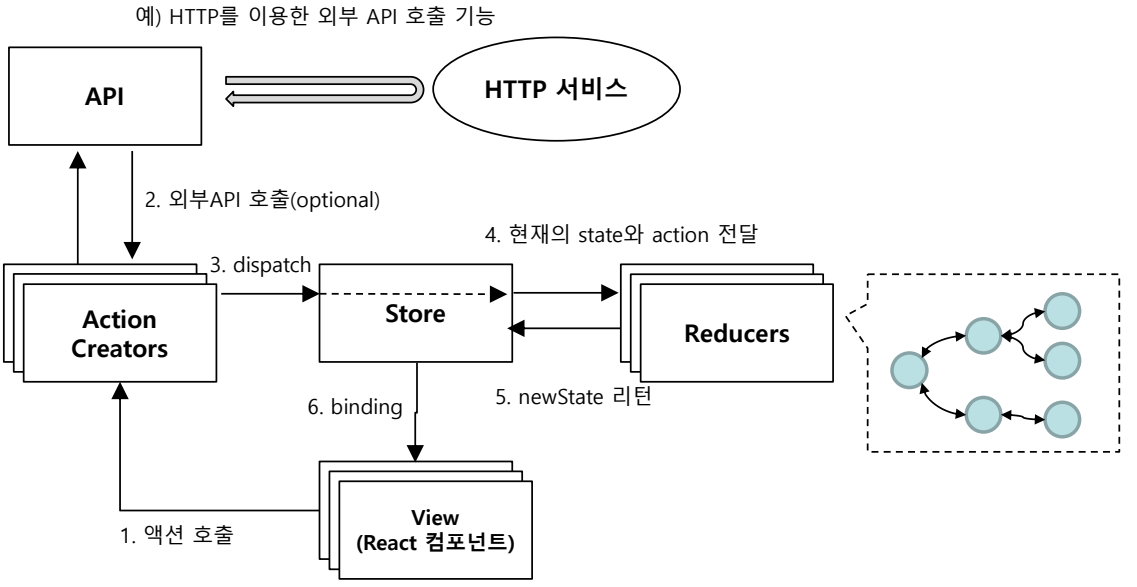
4. Redux 아키텍처(3)

Redux의 3가지 원칙

- Single Source of Truth(진실에 대한 단일 근원)
  - 애플리케이션의 상태 관리를 위해 단 하나의 Store를 사용함
- State is read-only(상태는 읽기 전용)
  - 애플리케이션의 다른 부분에서 상태를 변경할 수 없음
  - 상태를 변경하기 위해서는 action이 보내져야 함.
  - Store객체는 단 4개의 메서드. setter 메서드가 없음
    - dispatch(action), subscribe(listener), getState(), replaceReducer(nextReducer)
- Changes are made width Pure Function
  - "변경은 순수 함수로만 이루어져야 한다."
  - Reducer가 순수 함수임.
    - Reducer는 현재의 상태를 인자로 전달받고 상태를 수정하여 새로운 상태로 리턴하는 함수
  - 순수 함수의 정의는 이전 페이지에서...

4. Redux 아키텍처(4)

전체 흐름도

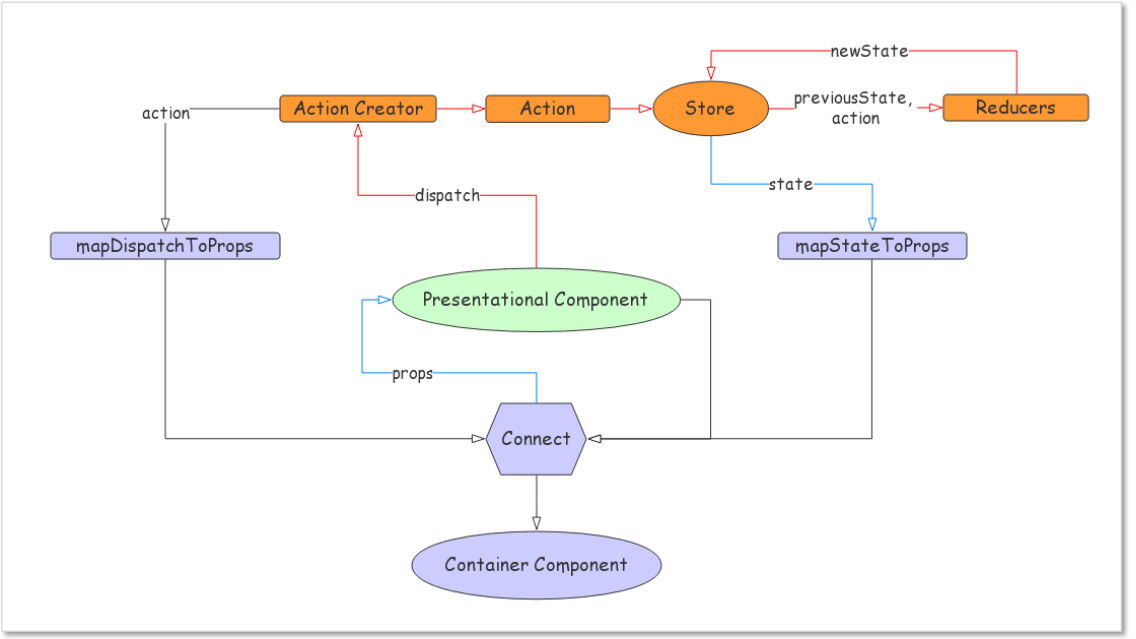


10

- View는 Store의 상태 데이터를 바인딩하여 화면을 구성한다.
- React Component에서 이벤트가 발생하면 Action 메서드를 호출한다.
- Action 메서드에서 외부 API 호출과 같은 비즈니스 로직을 실행하고 그 처리 결과를 액션정보를 담아 dispatch() 메서드를 호출해 Store로 전달한다.
- Store는 액션 정보이외에 현재의 상태 데이터를 Root Reducer로 전달한다.
- Root Reducer는 상태 데이터의 키의 값을 확인해 적절한 Reducer가 처리하도록 계층 구조를 따라 전달한다.
- 각 Reducer에서는 기존 상태의 불변성을 유지하여 새로운 객체를 만들어 리턴한다.
- Store는 Reducer로부터 새로운 상태를 설정한다.
- Store를 구독중인 React Component는 변경된 상태를 바인딩하여 UI를 갱신한다.

4. Redux 아키텍처(5)

React + Redux



11

■ 가능하다면 표현 컴포넌트로 개발함.

- 표현 컴포넌트(Presentational Component)가 테스트가 용이하고 재사용성이 좋음.

■ 표현컴포넌트를 Wrapping한 컨테이너 컴포넌트를 만듬

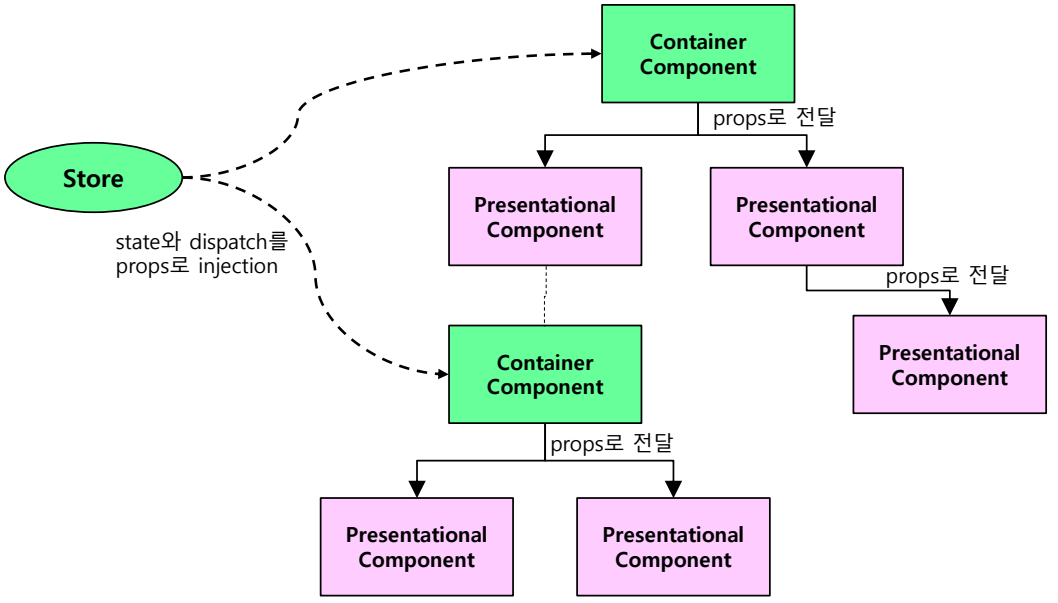
- react-redux 라이브러리가 제공하는 고차 컴포넌트를 이용하면 편리하게 만들 수 있음

■ connect 고차함수 mapStateToProps, mapDispatchToProps 함수를 인자로 전달해서 리턴받은 함수에 다시 표현컴포넌트를 인자로 전달하여 호출한 뒤 컨테이너 컴포넌트를 생성함.

- const AppContainer = connect(mapStateToProps, mapDispatchToProps)(App);

4. Redux 아키텍처(6)

React + Redux(이어서)



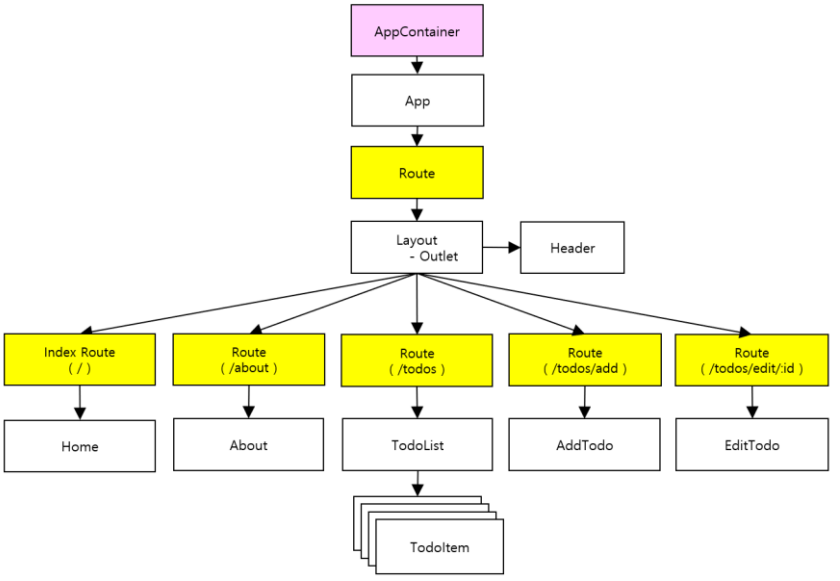
12

■ 모든 표현 컴포넌트를 Wrapping하여 컨테이너 컴포넌트로 만들 필요는 없다.

- 주요 거점 컴포넌트에 대해 Container 컴포넌트를 작성하고 부분적인 컴포넌트 트리에서는 Props로 전달하는 방법을 사용함.
- 표현 컴포넌트만으로 구성된 경우가 컴포넌트의 재사용성이 좋음

5. Redux 기본 적용(1)

8장에서 작성한 Todolist 앱을 Redux를 사용하도록 변경함.



13

■ 기존 예제는 AppContainer 컴포넌트를 직접 작성했다.

- 이 컴포넌트 내부에 상태 데이터와 상태를 변경하는 메서드를 모두 작성했었다.
- 상태와 메서드를 속성(props)로 App으로 전달하고 다시 하위 컴포넌트들로 props를 통해서 전달하는 방식이었다.
- props-props-props .....
  - 새로운 속성이 추가된다면? 거쳐가는 경로상의 모든 컴포넌트에 props 추가해야 함.

5. Redux 기본 적용(2)

상태 트리와 Reducer 설계를 반드시...

- App에서 관리해야 하는 상태를 먼저 설계해야 함.
- 그후 상태를 변경하는 기능을 Reducer로 표현해야 함.

Todolist App의 상태 도출

- todolist 배열!!
  - 특정 컴포넌트에서만 사용되는 상태이거나 컴포넌트의 생명주기가 바뀌더라도 유지되어야 할 필요가 없는 상태는 Redux State로 관리할 필요 없음

Reducer Action 도출

- 오로지 상태데이터가 바뀌는 작업에 국한지어 생각해야 함.
  - addTodo
  - deleteTodo
  - toggleDone
  - updateTodo

14

■ 전달되는 Action(Message)의 형식은 다음과 유사할 것이다.

- { type : "addTodo", payload : { todo:"할일1", desc:"설명1" } }

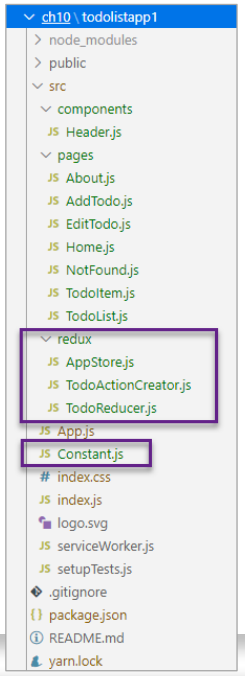
■ Action에서 type은 상수(Constant)를 이용해 오탈자가 발생하지 않도록 전달함

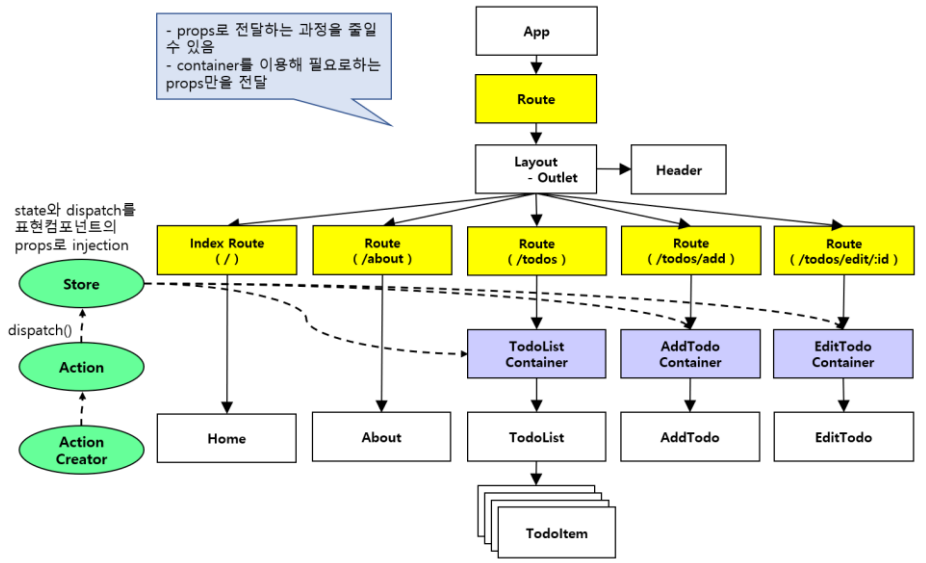
■ Action의 형식도 반드시 미리 정의되어 있어야 함.

5. Redux 기본 적용(3)

- 새롭게 변경할 예제의 구조

yarn add redux react-redux





15

■ 다음 명령어를 실행하여 기존 프로젝트에 패키지를 추가한다.

- yarn add redux react-redux

■ 새롭게 변경할 예제의 컴포넌트에서 표현컴포넌트는 변경할 내용이 없다.

- 이 예제의 표현 컴포넌트는 App, InputTodo, Todolist, TodoItem 이며, 기존 컴포넌트를 그대로 사용할 것이다.

■ AppContainer 파일 삭제

■ App 컴포넌트 내용 변경

- PropTypes를 import하는 구문과 PropTypes 정의하는 코드를 삭제함
- <Route />에서 props를 전달하는 코드를 모두 제거
- 전체 코드는 다음을 확인함

  https://github.com/stepanowon/react_quickstart_v2/blob/master/examples/ch10/todolistapp1/src/App.js

5. Redux 기본 적용(4)

- src/redux/TodoReducer.js

```
import produce from 'immer';
//액션 타입명의 오타를 줄이기 위한 상수
export const TODO_ACTION = {
    ADD_TODO : "addTodo",
    DELETE_TODO : "deleteTodo",
    TOGGLE_DONE : "toggleDone",
    UPDATE_TODO : "updateTodo",
}
const initialState = {
    todolist : [
        { id:1, todo:"ES6학습", desc:"설명1", done:false },
        { id:2, todo:"React학습", desc:"설명2", done:false },
        { id:3, todo:"ContextAPI 학습", desc:"설명3", done:true },
        { id:4, todo:"야구경기 관람", desc:"설명4", done:false },
    ]
}

(다음 페이지로 이어짐)
```

16

■ src/redux/TodoReducer.js 에서는 초기 상태 데이터로 주어질 객체와 Reducer 함수를 작성한다.

5. Redux 기본 적용(5)

```
const TodoReducer = (state=initialState, action) => {
    let index;
    switch(action.type) {
        case TODO_ACTION.ADD_TODO :
            return produce(state, (draft)=> {
                draft.todolist.push({ id:new Date().getTime(),
                    todo:action.payload.todo, desc: action.payload.desc, done:false});
            })
        case TODO_ACTION.DELETE_TODO :
            index = state.todolist.findIndex((item)=>item.id === action.payload.id);
            return produce(state, (draft)=> {
                draft.todolist.splice(index,1);
            })
        case TODO_ACTION.TOGGLE_DONE :
            index = state.todolist.findIndex((item)=>item.id === action.payload.id);
            return produce(state, (draft)=> {
                draft.todolist[index].done = !draft.todolist[index].done;
            })
        case TODO_ACTION.UPDATE_TODO :
            index = state.todolist.findIndex((item)=>item.id === action.payload.id);
            return produce(state, (draft)=> {
                draft.todolist[index] = { ...action.payload };
            })
        default :
            return state;
    }
}
export default TodoReducer;
```

17

■ ES6의 default paramemter 문법을 이용해서 Store의 초기 상태를 부여한다.

■ action.type 에 따라 분기처리하여 상태를 변경한다.

- 이 때 상태 변경을 위해 불변성 라이브러리 immer를 사용했고, 새로운 todolist 객체를 만들어 리턴하면 새로운 상태로 설정된다.

5. Redux 기본 적용(6)

- src/redux/AppStore.js

```
import { createStore } from 'redux';
import TodoReducer from './TodoReducer';

const AppStore = createStore(TodoReducer);
export default AppStore;
```

- src/redux/TodoActionCreator.js

```
import { TODO_ACTION } from "./TodoReducer"

const TodoActionCreator = {
    addTodo(todo, desc) {
        return { type: TODO_ACTION.ADD_TODO, payload: { todo, desc } }
    },
    deleteTodo(id) {
        return { type: TODO_ACTION.DELETE_TODO, payload: { id } }
    },
    toggleDone(id) {
        return { type: TODO_ACTION.TOGGLE_DONE, payload : { id } }
    },
    updateTodo(id, todo, desc, done) {
        return { type: TODO_ACTION.UPDATE_TODO, payload : { id, todo, desc, done } }
    },
}
export default TodoActionCreator;
```

■ TodoActionCreator.js 내부의 각각의 메서드가 리턴하는 객체가 액션 정보이다.

- type은 어떤 액션이 실행되어야 하는지를 나타내는 문자열이다.
- payload는 액션을 처리할 때 필요한 전달할 값이다.

■ Action은 "Message"이다.

5. Redux 기본 적용(7)

- src/pages/TodoList.js 변경
  - 기존 TodoList 컴포넌트를 이용해 Container를 생성해 export함.

```
……
import TodoActionCreator from '../redux/TodoActionCreator';
import { connect } from 'react-redux';
const TodoList = (props) => {
  ……
};
TodoList.propTypes = { ……(생략) };

const mapStateToProps = (state)=> ({
    states : {
        todolist : state.todolist
    }
})

const mapDispatchToProps = (dispatch)=> ({
    callbacks : {
        deleteTodo : (id) => dispatch(TodoActionCreator.deleteTodo(id)),
        toggleDone : (id) => dispatch(TodoActionCreator.toggleDone(id))
    }
})

const TodoListContainer = connect(mapStateToProps, mapDispatchToProps)(TodoList);
export default TodoListContainer;
```

19

■ react-redux 라이브러리가 제공하는 connect 고차 함수를 이용해 Store의 상태와 Dispatch 메서드를 App 컴포넌트에 속성으로 주입하여 새로운 TodoListContainer 컴포넌트를 생성한다.

■ mapStateToProps 메서드와 mapDispatchToProps 메서드가 리턴하는 객체가 TodoList 컴포넌트에 전달하는 속성이 된다.

- 표현 컴포넌트의 속성에 맞춰서 mapStateToProps, mapDispatchToProps 메서드의 리턴값 객체를 지정해야 함.

5. Redux 기본 적용(8)

- src/pages/AddTodo.js 변경

```
import React, { useState } from 'react';
import PropTypes from 'prop-types';

import TodoActionCreator from '../redux/TodoActionCreator';
import { connect } from 'react-redux';


const AddTodo = props => {
  ......
};

AddTodo.propTypes = { ......(생략) };

const mapDispatchToProps = (dispatch)=> ({
    callbacks : {
        addTodo : (todo, desc) => dispatch(TodoActionCreator.addTodo(todo, desc)),
    }
})

const AddTodoContainer = connect(null, mapDispatchToProps)(AddTodo);
export default AddTodoContainer;
```

20

■ 만일 속성으로 전달할 state가 없거나 dispatch가 없다면 connect() 고차 함수에 전달할 값을 null로 지정할 수도 있음.

- 이 예제에서는 addTodo() 메서드만 전달하면 되기 때문에, mapStateToProps 메서드는 작성하지 않았음.

5. Redux 기본 적용(9)

- src/pages/EditTodo.js 변경

```
……
import TodoActionCreator from '../redux/TodoActionCreator';
import { connect } from 'react-redux';

const EditTodo = ({ todolist, callbacks }) => {
    const navigate = useNavigate();
    const params = useParams();
    //const todoitem = callbacks.getTodoOne(props.match.params.id);
    const todoitem = todolist.find((item)=>item.id === parseInt(params.id,10));
    if (!todoitem) {      navigate('/todos');  }
    const [ todoOne, setTodoOne ] = useState({ ...todoitem });

    const updateContactHandler = ()=> {
        if (todoOne.todo.trim() === "" || todoOne.desc.trim()==="") {
            alert('반드시 할일, 설명을 입력해야 합니다.');
            return;
        }
        let { id, todo, desc, done } = todoOne;
        callbacks.updateTodo(id, todo, desc, done);
        navigate('/todos');
    }
    return (
       ……(생략)
    );
};
```


■ EditTodo 컴포넌트에서 사용자가 편집하려고 하는 한 건의 todo 정보를 알아내야 함.

- 기존 예제는 이것을 처리하기 위해 callbacks 속성으로 전달받는 메서드 중 getTodoOne() 을 이용했었음
- 하지만 이 메서드는 상태의 변경과 관련이 없으므로 ActionCreator로 처리할 수 없음
- 따라서 props로 todolist를 추가하고 상태 데이터를 todolist 속성으로 주입시킴
- 한건의 todo를 찾는 작업은 배열의 find 메서드를 이용하면 됨.

5. Redux 기본 적용(10)

- src/pages/EditTodo.js 변경(이어서)

```
EditTodo.propTypes = {
    todolist : PropTypes.arrayOf(PropTypes.object).isRequired,
    callbacks :PropTypes.object.isRequired,
};

const mapStateToProps = (state)=> ({
    todolist : state.todolist
})

const mapDispatchToProps = (dispatch)=> ({
    callbacks : {
        updateTodo : (id, todo, desc, done) =>
            dispatch(TodoActionCreator.updateTodo(id, todo, desc, done)),
    }
})

const EditTodoContainer = connect(mapStateToProps, mapDispatchToProps)(EditTodo);
export default EditTodoContainer;
```


22

5. Redux 기본 적용(11)

- src/App.js 변경

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes,  } from 'react-router-dom';
//import PropTypes from 'prop-types';
import Layout from './components/Layout';
.....(생략)

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="about" element={<About />} />
          <Route path="todos" element={<TodoList />} />
          <Route path="todos/add" element={<AddTodo />} />
          <Route path="todos/edit/:id" element={<EditTodo />} />
          <Route path="*" element={<NotFound />} />
        </Route>
      </Routes>
    </Router>
  );
};

export default App;
```


■ AppContainer 파일 삭제

■ App은 더이상 AppContainer로부터 속성(props)을 전달받아 자식 컴포넌트로 전달할 필요가 없음

- 위 코드에서 볼드체로 표현된 부분 중심으로 변경

5. Redux 기본 적용(12)

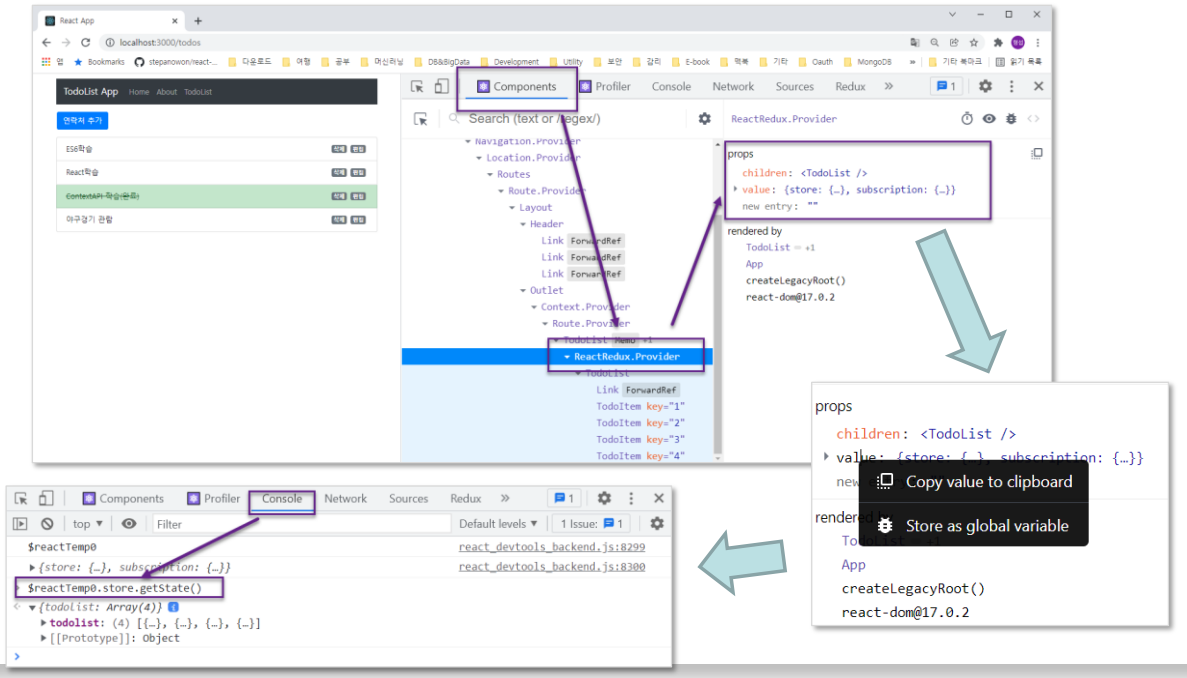
- src/index.js 변경

```
import React from 'react';
import ReactDOM from 'react-dom';
import 'bootstrap/dist/css/bootstrap.css';
import './index.css';
//import AppContainer from './AppContainer';
import App from './App';
import reportWebVitals from './reportWebVitals';

import AppStore from './redux/AppStore';
import { Provider } from 'react-redux';

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

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```


■ App 컴포넌트를 store 객체가 지정된 Provider 컴포넌트로 wrapping한 후 render한다.

5. Redux 기본 적용(13)

- 실행 결과



25

■ react-redux는 내부적으로 Context API를 사용함.

- 화면속의 ReactRedux.Provider를 살펴보면 Context API의 Provider와 같은 구조라는 것을 알 수 있음.
- 따라서 현재 Redux Store 객체는 Provider의 value를 통해서 디버깅을 위해 접근할 수 있음

■ value 속성을 마우스 우클릭하면 나타나는 컨텍스트 메뉴에서 'Store as global variable'을 선택하면 콘솔에서 접근해볼 수 있음

- 화면처럼 생성된 전역 객체를 이용해 [전역객체].store.getState() 로 Store의 상태 데이터를 콘솔에 출력볼 수 있음

6. 다중 리듀서 적용(1)

애플리케이션에서 처리할 액션이 많아진다면?

- 여러 개의 Reducer로 분리해야 할 필요가 있음
  - combineReducers () 함수를 이용해 여러 개의 분리된 Reducer를 단일 Reducer로 만들어낼 수 있음

다중 리듀서 테스트

- 기능을 확인하기 위해 Todolist 예제에 새로운 컴포넌트 추가와 약간의 코드 추가
  - MyTime 컴포넌트
  - TimeReducer
  - RootReducer : TimeReducer와 TodoReducer를 결합한 Root Reducer
  - TimeActionCreator
- 추가할 상태와 Dispatch 메서드
  - currentTime, changeTime()

```
{
  home : { currentTime : xxxxx },
  todos : {
    todolist : [ .... ]
  }
}
```

26

■ 미리 전체 상태 트리를 설계해야 함.

■ 이번 예제를 위한 전체 상태 트리는 다음과 같이 정의하였음

```
{
  home : { currentTime : xxxxx },
  todos : {
    todolist : [ .... ]
  }
}
```

6. 다중 리듀서 적용(2)

예제 작성

- src/redux/TimeReducer.js 추가

```
export const TIME_ACTION = {
    CHANGE_TIME : "changeTime",
}

const initialState = {
    currentTime : new Date()
}
const TimeReducer = (state=initialState, action) => {
    switch(action.type) {
        case TIME_ACTION.CHANGE_TIME:
            return { ...state, currentTime : action.payload.currentTime }
        default:
            return state;
    }
}

export default TimeReducer;
```

27

■ 여러 개의 Reducer를 사용하면 전체 상태의 일부분을 리듀서에서 사용할 수 있도록 초기값을 전달한다.

- RootReducer에서 리듀서들을 조합할 때 상태의 Top Level 속성명을 지정한다.

6. 다중 리듀서 적용(3)

- src/redux/RootReducer.js 추가
  - TodoReducer, TimeReducer를 결합하기 위해 combineReducers 함수를 사용함.
  - Reducer를 결합하면서 각각의 Reducer가 관리하는 상태 데이터의 속성명을 지정함

```
import { combineReducers } from 'redux';
import TimeReducer from './TimeReducer';
import TodoReducer from './TodoReducer';

const RootReducer = combineReducers({ home : TimeReducer, todos: TodoReducer });
export default RootReducer;
```

- src/redux/TimeActionCreator.js 추가

```
import { TIME_ACTION } from './TimeReducer';

const TimeActionCreator = {
    changeTime() {
        return {  type: TIME_ACTION.CHANGE_TIME, payload : { currentTime: new Date() } }
    }
}

export default TimeActionCreator;
```

28

■ combineReducers 메서드를 이용해 각각의 Reducer들이 리턴하는 상태 객체를 조합하며, 이때 home, todos와 같은 객체의 속성명은 전체 상태 트리를 반영해 지정한다.

- 따라서 전체 앱의 상태 트리를 정의하는 것이 대단히 중요함.

■ 전체 앱의 상태 트리

```
{
   home : { currentTime : xxxxx },
   todos : {
      todolist : [ .... ]
   }
}
```

6. 다중 리듀서 적용(4)

- src/redux/AppStore.js 변경
  - TodoReducer가 아닌 RootReducer를 참조하도록 변경함.

```
import { createStore } from 'redux';
import RootReducer from './RootReducer';

const AppStore = createStore(RootReducer);
export default AppStore;
```

- src/pages/EditTodo.js, TodoList.js 변경
  - 상태 트리가 변경되었기 때문에 mapStateToProps를 변경해주어야 함.

```
……
const mapStateToProps = (state)=> ({
    todolist : state.todos.todolist
})
……
```


29

6. 다중 리듀서 적용(5)

- src/pages/MyTime.js

```
import React from 'react';
import PropTypes from 'prop-types';

const MyTime = ({ currentTime, changeTime }) => {
    return (
        <div className="row">
            <div className="col">
                <button className="btn btn-primary" onClick={()=>changeTime() }>
                    현재 시간 확인</button>
                <h4>
                  <span className="label label-default">
                    { currentTime.toLocaleString() }
                  </span>
                </h4>
            </div>
        </div>
    );
};

MyTime.propTypes = {
    currentTime: PropTypes.object.isRequired,
    changeTime: PropTypes.func.isRequired
};

export default MyTime;
```

6. 다중 리듀서 적용(6)

- src/pages/Home.js

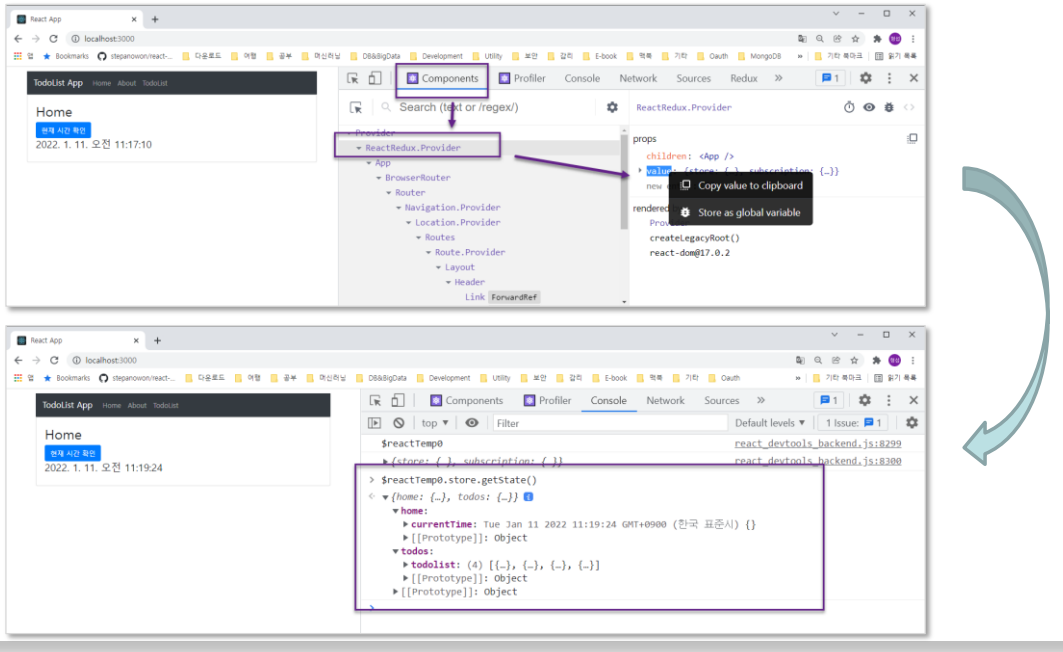
```
import React from 'react';
import MyTime from './MyTime';
import PropTypes from 'prop-types';
import TimeActionCreator from '../redux/TimeActionCreator';
import { connect } from 'react-redux';

const Home = ({ currentTime, changeTime }) => {
    return (
        <div className="card card-body">
            <h2>Home</h2>
            <MyTime currentTime={currentTime} changeTime={changeTime} />
        </div>
    );
};
Home.propTypes = {
    currentTime: PropTypes.object.isRequired,
    changeTime: PropTypes.func.isRequired
};

const mapStateToProps = (state)=> ({ currentTime : state.home.currentTime })
const mapDispatchToProps = (dispatch)=> ({
    changeTime : () => dispatch(TimeActionCreator.changeTime()),
})

const HomeContainer = connect(mapStateToProps, mapDispatchToProps)(Home);
export default HomeContainer;
```

31

■ MyTime 컴포넌트에 대해 Container를 작성할 수도 있지만 이 예제에서는 라우트 경로 단위로 보여지는 화면의 최상위 컴포넌트를 주요 거점 컴포넌트로 설정하였기 때문에 Home 컴포넌트에 대한 Container를 작성하고 Props를 이용해 MyTime 컴포넌트로 전달하도록 하였음.

6. 다중 리듀서 적용(7)

실행 결과



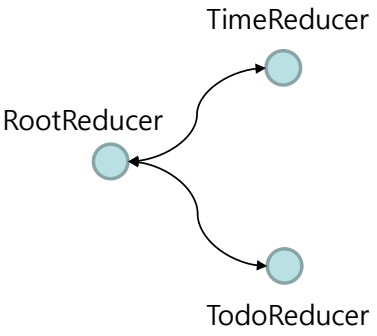
32

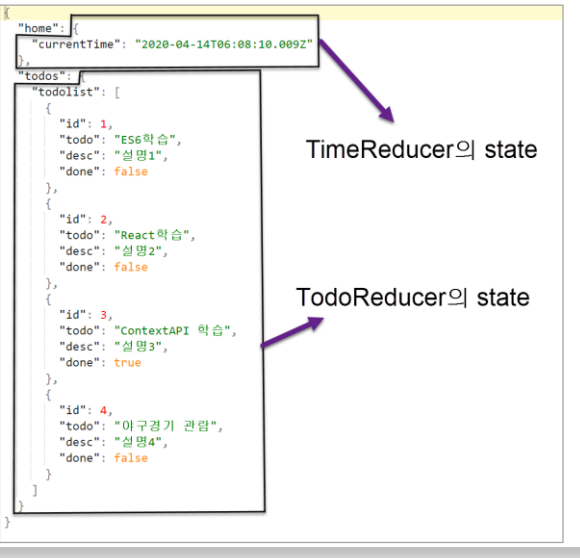
■ React Dev Tools의 기능 중에 객체를 Console에 출력할 수 있는 기능이 포함되어 있음.
위의 그림대로 진행하면 됨.

6. 다중 리듀서 적용(8)

combine!!

```
const RootReducer = combineReducers({
        home : TimeReducer,
        todos: TodoReducer
});
```



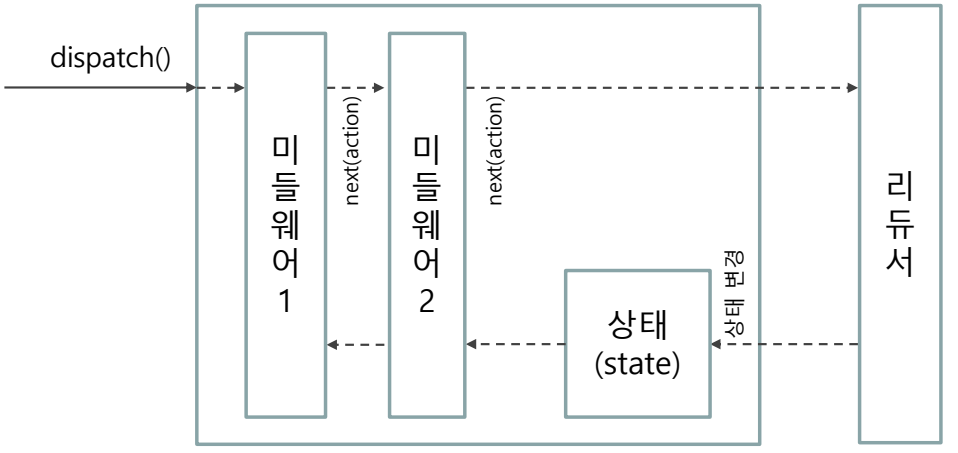


33

7. Redux 미들웨어(1)

미들웨어(Middleware)란?

- 액션 생성자가 액션을 전달한후 Reducer에 도달하기 전과 상태 변경 후에 수행할 작업을 지정할 수 있음
- Store 객체에서 지정함



34

■ next(action)을 실행하여 다음 체인으로 action을 전달함

■ next(action)을 실행하지 않으면?

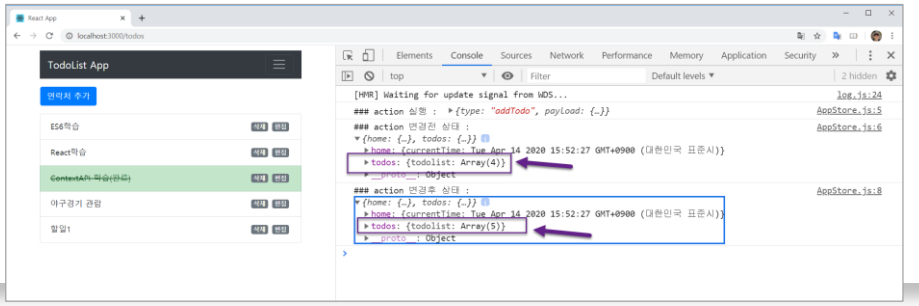
- 리듀서로 action이 전달되지 않으므로 상태가 변경되지 않음.

7. Redux 미들웨어(2)

간단한 로깅 기능을 수행하는 미들웨어 추가

- src/redux/AppStore.js 변경

```
import { createStore, applyMiddleware } from 'redux';
import RootReducer from './RootReducer';

const logger = (store) => (next) => (action) => {
    console.log("### action 실행 : ", action);
    console.log("### action 변경전 상태 : ", store.getState());
    next(action);
    console.log("### action 변경후 상태 : ", store.getState());
}
const AppStore = createStore(RootReducer, applyMiddleware(logger));
export default AppStore;
```



35

8. 비동기 처리(1)

Redux에서의 비동기 처리 방법

- 전통적인 방법
  - Promise 패턴을 이용할 수 있음
    - axios 와 같은 AJAX 라이브러리를 이용할 때 Promise 객체를 리턴함.
  - async~await~ 기법 적용
  - 간단한 경우라면 위의 두 방법을 사용해도 됨
- redux 에서는?
  - 비동기 처리를 할만한 좋은 위치가 없음
    - ActionCreator에서 하고 싶으나... ActionCreator는 action 객체를 리턴해야 하므로 비동기 처리에 적절하지 않음.
    - 그래서 Middleware를 이용하여 ActionCreator 내부에서 비동기 처리를 가능하게 함.
  - redux-thunk
  - redux-saga
  - redux-promise, redux-promise-middleware
- 이 과정에서는 redux-thunk를 다룸

8. 비동기 처리(2)

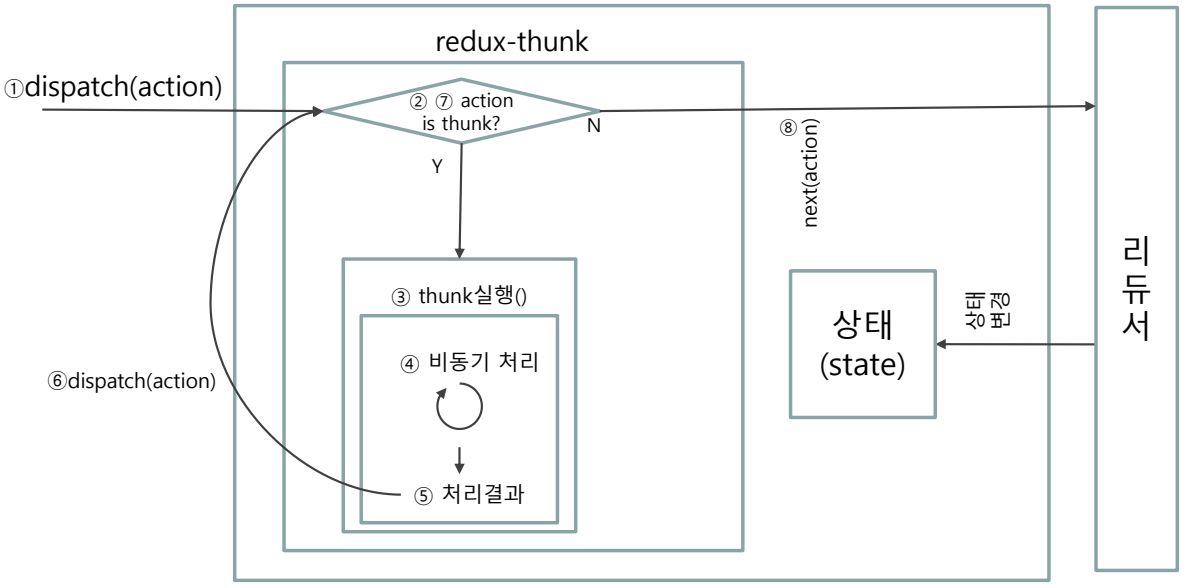
redux-thunk

- 비동기 처리를 위한 redux용 미들웨어
- thunk
  - 실행을 지연시키기 위해 표현식으로 wrap한 함수
  - ActionCreator가 액션정보가 아니라 thunk 함수를 리턴함.
- Redux의 Reducer는 순수함수이므로 Side effect를 일으켜서는 안됨.
  - Reducer 수준에서 dispatch 할 수 없음
  - 따라서 액션 생성자(ActionCreator)에서 처리해야 함.
- 적용 방법
  - store 객체에서 미들웨어 등록

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducer/rootReducer';

const store = createStore(rootReducer, applyMiddleware(thunk) );

export default store;
```

8. 비동기 처리(3)

redux-thunk 실행 구조



■ ActionCreator가 Action(Message)를 리턴하지 않고 thunk 함수를 리턴함

■ redux-thunk 미들웨어를 거칠 때 Action이 thunk 함수인지를 확인하고

- Action이 일반적인 메시지라면 --> next(action)
- Action이 thunk라면 --> Action을 실행(비동기처리)
- 비동기 처리가 완료되면 다시 Action을 dispatch함 --> 비동기 처리 결과를 이용해 상태를 변경하는 작업 진행

8. 비동기 처리(4)

Todolist 앱에 redux-thunk 적용

- 시간 확인시 의도적 1초 지연 시간 발생
- src/redux/TimeReducer.js 변경

```
export const TIME_ACTION = {
    CHANGE_TIME_REQUEST : "changeTimeRequest",
    CHANGE_TIME_COMPLETED : "changeTimeCompleted",
}

const initialState = {
    currentTime : new Date(),
    isChanging : false,
}
const TimeReducer = (state=initialState, action) => {
    switch(action.type) {
        case TIME_ACTION.CHANGE_TIME_REQUEST:
            return { ...state, isChanging: true  }
        case TIME_ACTION.CHANGE_TIME_COMPLETED:
            return { ...state, currentTime : action.payload.currentTime, isChanging: false  }
        default:
            return state;
    }
}

export default TimeReducer;
```

39

■ 비동기 처리 시점별로 Action Type을 추가하였음

- ~_REQUEST : 비동기 처리 시작 시점
  - 비동기 처리가 진행중임을 알리기 위해 isChanging 값을 true로 설정
- ~_COMPLETED : 비동기 처리 완료 시점
  - 비동기 처리가 완료되었음을 알리기 위해 isChanging 값을 false로 설정

8. 비동기 처리(5)

- src/redux/TimeActionCreator.js 변경

```
import { TIME_ACTION } from './TimeReducer';

const TimeActionCreator = {
    changeTimeRequest() {
        return {  type: TIME_ACTION.CHANGE_TIME_REQUEST }
    },
    changeTimeCompleted() {
        return { type: TIME_ACTION.CHANGE_TIME_COMPLETED, payload : { currentTime: new Date() }  }
    },
    //thunk 함수를 리턴함
    asyncChangeTime() {
        //의도적 지연시간 1초
        return (dispatch, getState)=> {
            dispatch(this.changeTimeRequest());
            setTimeout(()=>{
                dispatch(this.changeTimeCompleted());
            }, 1000)
        }
    }
}

export default TimeActionCreator;
```


40

8. 비동기 처리(6)

- src/pages/Home.js 변경

```
.....
const Home = ({ currentTime, changeTime, isChanging }) => {
    return (
        <div className="card card-body">
            <h2>Home</h2>
            <hr />
            {   isChanging ? <h4>시간 변경중</h4> :
                <MyTime currentTime={currentTime} changeTime={changeTime} /> }
        </div>
    );
};
Home.propTypes = {
    currentTime: PropTypes.object.isRequired,
    isChanging : PropTypes.bool.isRequired,
    changeTime: PropTypes.func.isRequired
};
const mapStateToProps = (state)=>({
    currentTime : state.home.currentTime,
    isChanging: state.home.isChanging
})
const mapDispatchToProps = (dispatch)=> ({
    changeTime : () => dispatch(TimeActionCreator.asyncChangeTime()),
})

const HomeContainer = connect(mapStateToProps, mapDispatchToProps)(Home);
export default HomeContainer;
```


41

8. 비동기 처리(7)

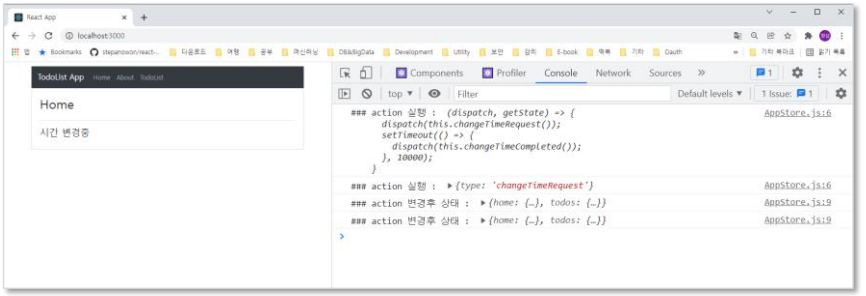
- src/redux/AppStore.js 변경

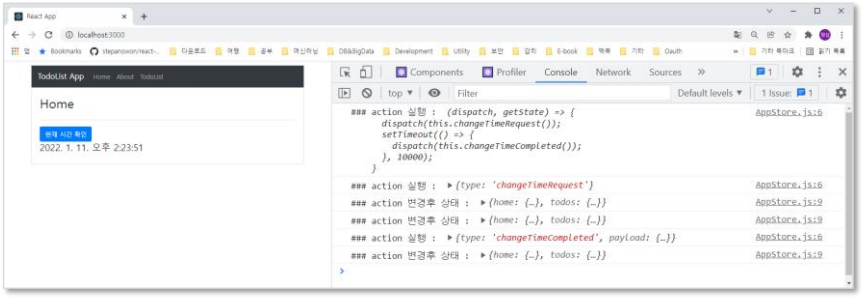
```
import { createStore, applyMiddleware } from 'redux';
import RootReducer from './RootReducer';
import thunk from 'redux-thunk';

const logger = (store) => (next) => (action) => {
    console.log("### action 실행 : ", action);
    //console.log("### action 변경전 상태 : ", store.getState());
    next(action);
    console.log("### action 변경후 상태 : ", store.getState());
}

const AppStore = createStore(RootReducer, applyMiddleware(logger, thunk));
export default AppStore;
```


42

8. 비동기 처리(8)

- 실행 결과

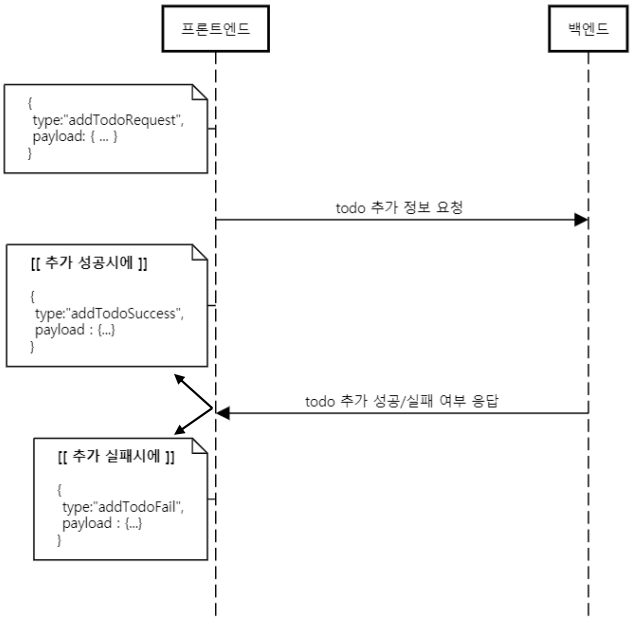




43

8. 비동기 처리(9)

백엔드 + Redux 에서의 비동기 처리



44

■ axios로 요청하기 전에 addTodoRequest 액션을 dispatch함

- 이 액션을 이용해 처리중을 나타내는 Spinner UI를 나타냄

■ 요청 후 응답이 수신되면 다음 두가지 경우로 처리함

- 응답이 성공이라면 addTodoSuccess 액션을 dispatch하여 상태를 변경하도록 함. 더불어 Spinner UI를 사라지도록 함
- 응답이 실패라면 addTodoFail 액션을 dispatch 함. 에러 메시지를 사용자에게 알리도록 UI를 변경함. Spinner UI를 사라지도록 함.

■ 정리하자면 비동기 처리를 위해 세가지 액션을 처리하는 Reducer 기능을 준비해야 함

- ~Request
- ~Success
- ~Fail

■ 액션의 type명은 직접 지정하는 것이므로 다른 이름으로 변경할 수 있음

8. 비동기 처리(10)

redux-saga

- 직관적인 Side Effect 관리자
  - 애플리케이션 내부의 사이드 이펙트(side effects: 예-데이터 처리와 관련된 비동기 처리)를 손쉽게 관리해주는 Redux 미들웨어.
  - https://redux-saga.js.org/
- 특징
  - ES6 Generator와 Effect를 이용함.
    - Generator 개념을 필수적으로 이해해야 함.
    - 러닝커브가 존재함
  - ActionCreator는 상태 변경 기능만을 가짐
    - SIde Effect(비동기 처리 코드)는 worker sage로 분리
- saga란?
  - 분산 트랜잭션을 처리하는 패턴(특히 MSA 환경에서)

45

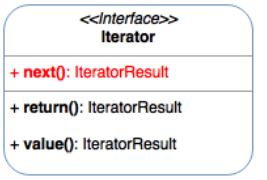
8. 비동기 처리(11)

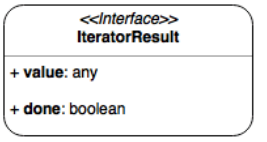
Iteration이란?

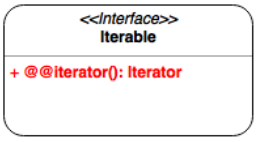
- 데이터 컬렉션(Array, Map등) 을 순회하기 위해 만들어진 규칙
- 규칙만 준수하면 어떤 객체에서도 구현 가능

두가지 규칙

- Iterator
  - 컬렉션을 순회하기 위한 next() 메서드를 구현하고 있음
  - next() 메서드의 호출결과 Iterator result를 리턴받음
    - { value : xxxxx, done : true/false }
  - 예) Array, Map 등
- Iterable
  - iterator를 이용한 순회가능한 데이터 구조
  - [Symbol.iterator]라는 메서드를 구현해야 함.
  - for ~ of ~ 문과 Spread 연산자를 사용할 수 있음.







46

■ Symbol은 ES6에 새롭게 추가된 타입이며 변경할 수 없는 타입의 값.

- 주로 이름 충돌이 발생하지 않는 유일한 객체의 속성 키를 만들기 위해 사용

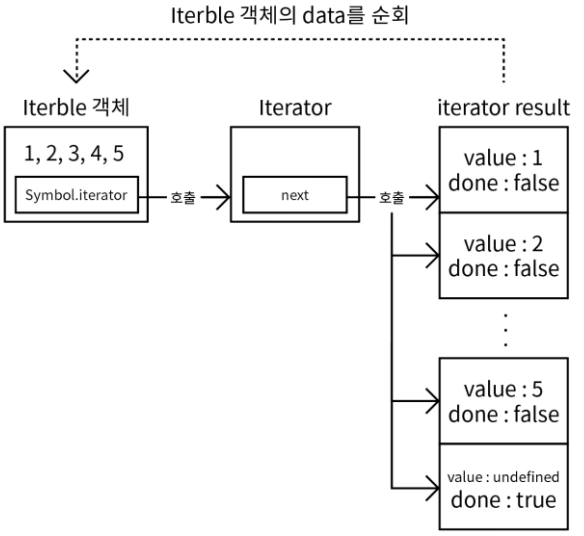
■ 그림 참조

- https://dzone.com/articles/es6-iterator-innbspdepth

8. 비동기 처리(12)

간단한 예제

```
const array = [1, 2, 3, 4, 5];

const iterator = array[Symbol.iterator]()

let iteratorResult = itResult = iterator.next();
while (iteratorResult.done === false) {
  console.log(iteratorResult);
  iteratorResult = iterator.next();
}

console.log("마지막 : ", iteratorResult);
```



47

■ 예제, 그림 참조

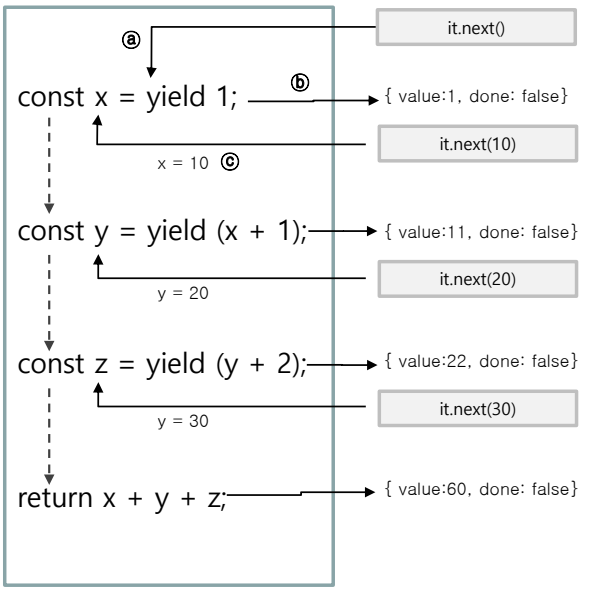
- https://velog.io/@kimjeongwonn/이터러블이터레이터제네레이터-복습

8. 비동기 처리(13)

Generator

- 일반적인 문법 확인
  - https://hacks.mozilla.org/2015/05/es6-in-depth-generators/
- 간단한 예제

```
//아래 코드를 콘솔에서 실행해보자
const numGenerator = function*() {
    const x = yield 1;
    const y = yield (x + 1);
    const z = yield (y + 2);
    return x + y + z;
}

//아래코드는 한줄씩 실행해보자.
const it = numGenerator();
console.log(it.next());
console.log(it.next(10));
console.log(it.next(20));
console.log(it.next(30));
```



48

■ Generator는 iterable 객체를 리턴하는 함수임

- 리턴된 iterable 객체의 next() 메서드를 호출할 때마다 generator함수의 중간 결과물을 여러 시간대별로 리턴받을 수 있음.

■ 일반 함수와의 비교

- 일반함수는 함수를 호출하면 함수의 끝까지 실행이 완료되어 버림
- Generator는 다음번 next()를 호출할 때까지 함수 내부의 실행을 정지시킬 수 있음.

8. 비동기 처리(14)

Generator를 이용한 비동기 처리 예

```
//Generator를 이용한 비동기 처리
const getContacts = (name) => {
  axios
    .get(`https://contactsvc.herokuapp.com/contacts_long/search/${name}`)
    .then((response) => gen.next(response.data));
};

const contactsGenerator = function* () {
  console.log("### Start!!");
  let contacts;
  contacts = yield getContacts("ja");
  console.log("ja 키워드 : ", contacts);
  contacts = yield getContacts("an");
  console.log("an 키워드 : ", contacts);
  contacts = yield getContacts("se");
  console.log("se 키워드 : ", contacts);
  return;
};
const gen = contactsGenerator();
gen.next();
```

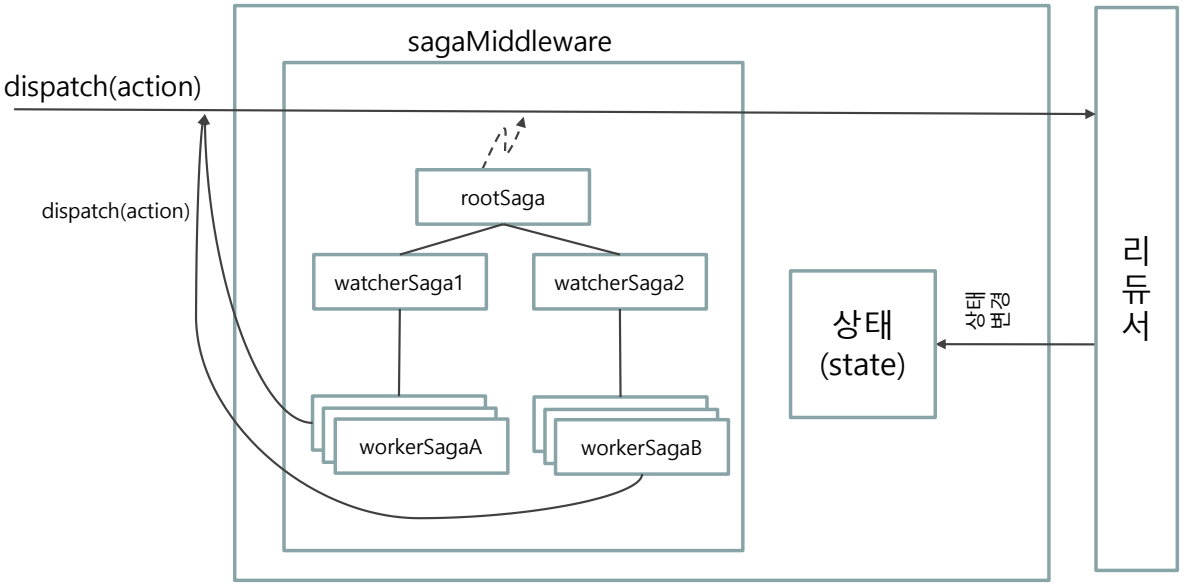
49

■ https://codesandbox.io/s/contactsapp-redux-saga-thnfq?file=/src/App2.js

- index.js 에서 App2를 참조하도록 주석을 변경하고 화면 오른쪽에서 콘솔을 살펴보자

8. 비동기 처리(15)

redux-saga 실행 구조



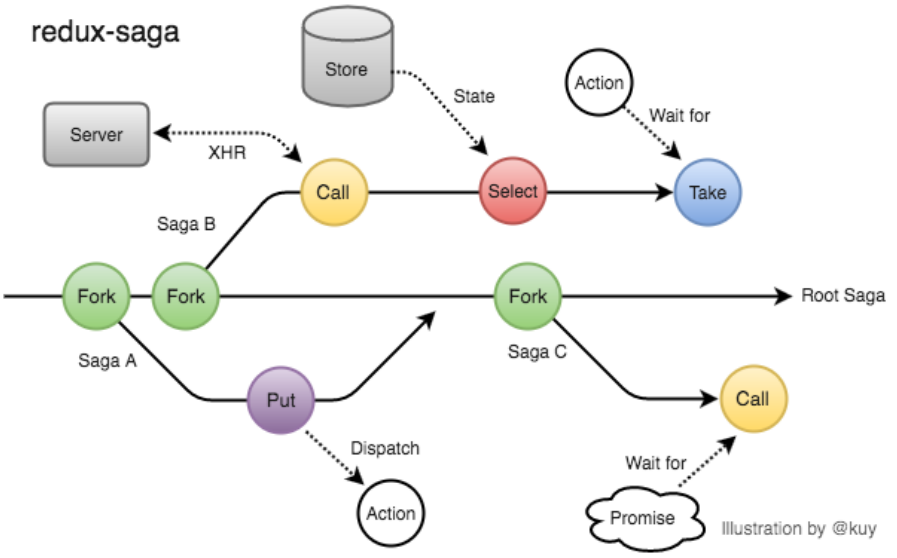
50

■ watcherSaga에 의해 하나의 workerSaga만이 실행되는 것은 아니며 다양한 effect에 의해 조합되고 처리의 순서를 지정할 수 있음.

■ saga는 ES6의 Generator를 이용함.

8. 비동기 처리(16)

redux-saga의 effect



■ https://ideveloper2.tistory.com/53 참조

■ effect 종류

- take: 특정 action을 감시함.
- put: action을 dispatch함.
- fork : 새로운 saga 태스크를 시작함.
- call : 블럭되는 fork이며 promise가 완료될 때까지 블럭상태가 됨.
- select: 상태(state)로부터 필요한 데이터를 읽어옴
- join: 다른 Task의 종료를 기다림.

8. 비동기 처리(17)

redux-promise-middleware

- ActionCreator가 리턴하는 Action이 다음과 같은 형식
  - { type: "ACTION_TYPE" , payload : [Promise객체] }
- 이 미들웨어에서 Action의 payload가 Promise인지 확인
  - promise가 아니라면 즉시 next(action)
  - promise라면?
    - next(action)하지 않고 비동기처리가 완료될 때까지 기다림
    - 미리 지정된 Action Type을 이용해 비동기 처리의 각 시점별로 정해진 Action을 Reducer로 전달함
  - 미리 지정된 Action Type 기본 설정
    - 비동기 처리 시작이 SEARCH_CONTACTS인 경우의 예시
    - SEARCH_CONTACTS_PENDING
    - SEARCH_CONTACTS_ FULFILLED
    - SEARCH_CONTACTS_REJECTED
  - 물론 지정된 Action Type은 변경할 수 있음

52

8. 비동기 처리(18)

- **redux-saga VS redux-thunk VS redux-promise-middleware**
  - 동일한 기능을 수행하는 코드를 세가지 버전으로 작성
  - redux-thunk :
    - https://bit.ly/react-redux-thunk
  - redux-midldeware
    - https://bit.ly/react-redux-promise
  - redux-saga
    - https://bit.ly/react-redux-saga

53

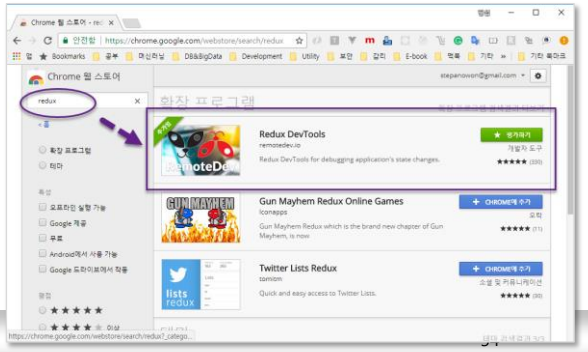
9. Redux Devtools(1)

Redux Devtools

- Redux를 이용한 앱을 개발할 때 개발을 강력하게 지원하는 개발 패키지 도구
  - Redux의 상태와 액션 정보를 시각화하며, 상태 변경을 추적할 수 있도록 함.

사용 방법

- https://github.com/zalmoxisus/redux-devtools-extension
- 크롬 확장 프로그램 설치 + 약간의 코드 추가



■ Redux Devtools를 이용하면 시간 여행 디버깅(Time Travel Debugging)이 가능하다.

- 상태의 변경을 시각적으로 추적할 수 있다.
- 특정 시점의 상태로 돌아가서 다시 테스트해볼 수 있다.

9. Redux Devtools(2)

TodoList 앱에 Redux Devtools 기능 추가

- 관련 패키지 참조
  - yarn add -D redux-devtools-extension
- src/redux/AppStore.js 변경

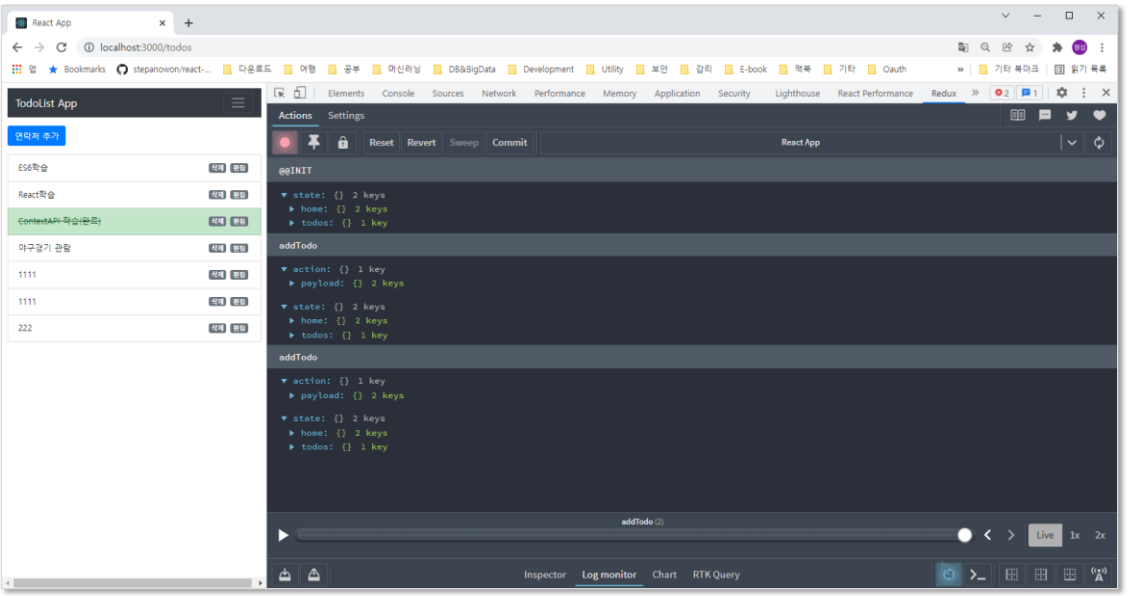
```
import { createStore, applyMiddleware } from 'redux';
import RootReducer from './RootReducer';
import thunk from 'redux-thunk';
import TimeActionCreator from './TimeActionCreator';
import TodoActionCreator from './TodoActionCreator';
import { composeWithDevTools } from 'redux-devtools-extension';

let AppStore;
if (process.env.NODE_ENV === "development") {
    const composeEnhancers = composeWithDevTools({ ...TimeActionCreator, ...TodoActionCreator });
    AppStore = createStore(RootReducer, composeEnhancers(
        applyMiddleware(thunk)
    ));
} else {
    AppStore = createStore(RootReducer, applyMiddleware(thunk));
}

export default AppStore;
```

55

■ 기존 Logger 미들웨어를 삭제하고 devTool 관련 미들웨어를 추가한다.

9. Redux Devtools(3)

- 실행 결과 1



■ Revert

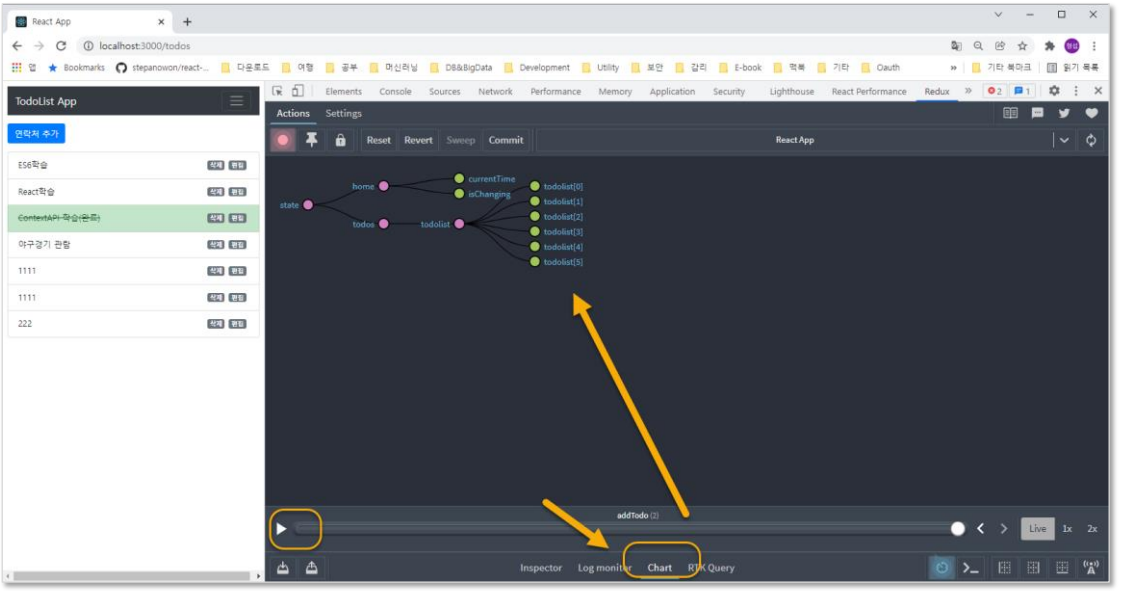
- 개발자 도구의 Redux 탭을 열어 놓고 Todo를 몇개 추가해본다.
- 그후 Redux Devtools 화면에서 Revert 버튼을 클릭하면 초기 상태로 돌아간다.

■ Commit - Revert

- 개발자 도구의 Redux 탭을 열어 놓고 Todo를 몇개 추가하고 Commit 버튼을 클릭한다.
- 그후 몇개의 Todo를 더 추가한다.
- 이제 Revert 버튼을 클릭하면 Commit된 시점의 상태로 돌아간다.

9. Redux Devtools(4)

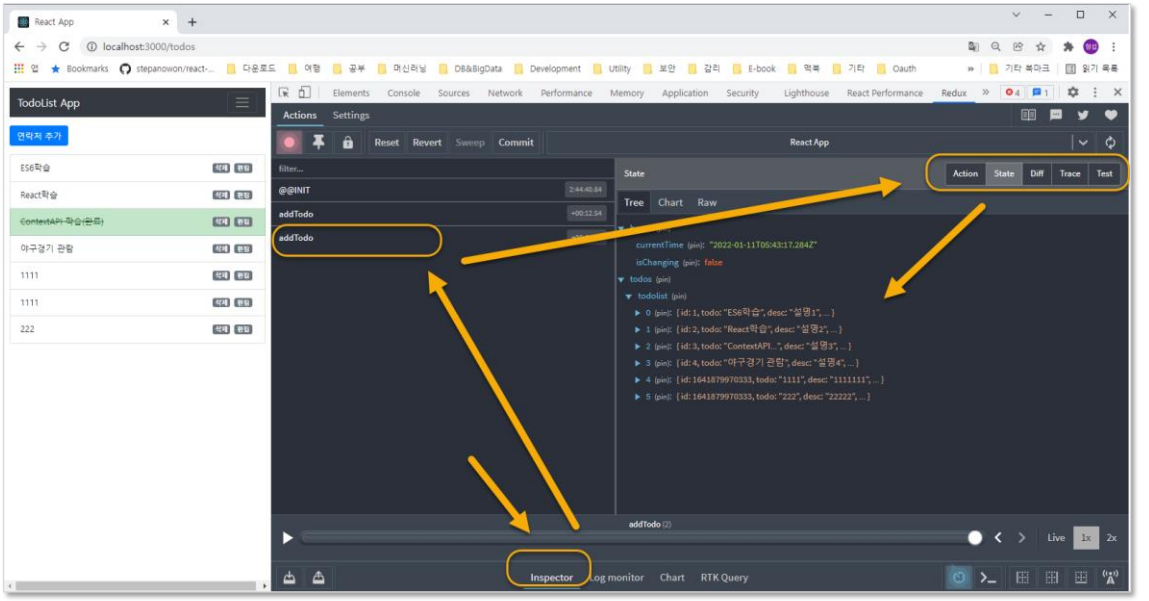
- 실행 결과 2



■ Redux Devtools 화면 왼쪽 상단에서 Chart를 선택하면 Redux가 관리하고 상태 정보를 트리 구조로 살펴볼 수 있다.

9. Redux Devtools(5)

- 실행 결과 3



58

■ Redux Devtools 화면 왼쪽 상단에서 Inspector를 선택하면 다음을 손쉽게 확인할 수 있다.

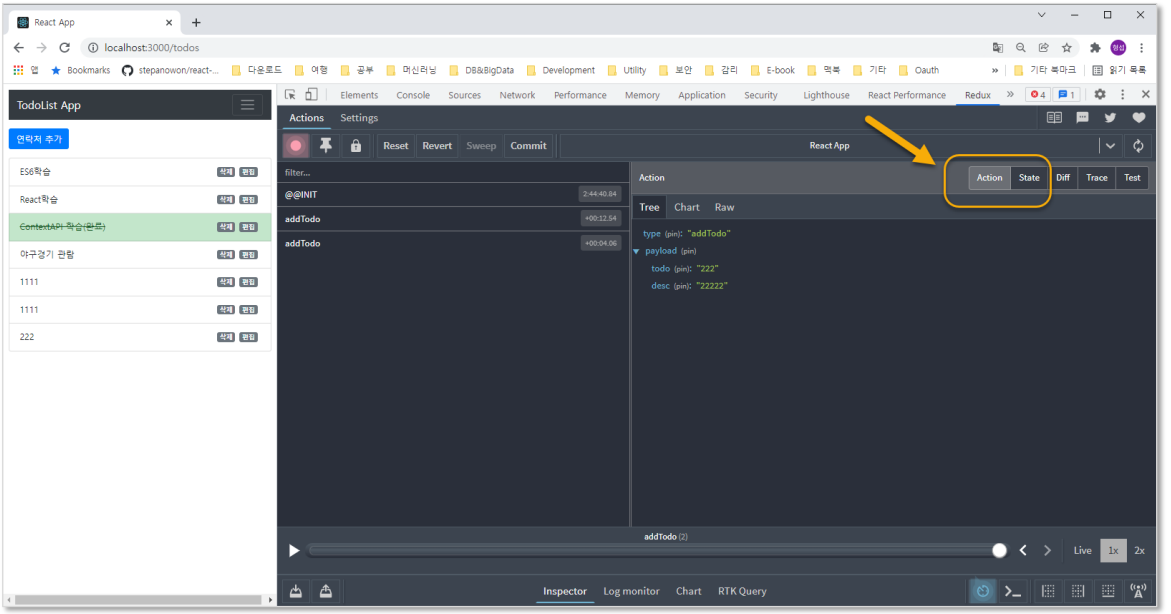
- 액션이 일어난 시점의 상태(State)
- 액션이 일어난 시점의 이전 상태와의 차이(Diff)
- 액션이 일어난 시점의 액션 정보(type, payload)
- 액션을 테스트할 수 있는 Jest 테스트 코드 생성

■ 좀더 자세한 설명은 공식 문서를 참조한다.

- https://github.com/gaearon/redux-devtools

9. Redux Devtools(6)

실행 결과 4



59

10. Hook + react-redux(1)

react-redux 7.1부터 hook 제공

- hook 제공 이전은?
  - connect() 고차 함수!!

```
......
import TodoActionCreator from '../redux/TodoActionCreator';
import { connect } from 'react-redux';
......
const mapStateToProps = (state)=> {
    return {
        states : {
            todolist : state.todos.todolist
        }
    }
}
const mapDispatchToProps = (dispatch)=> {
    return {
        callbacks : {
            deleteTodo : (id) => dispatch(TodoActionCreator.deleteTodo(id)),
            toggleDone : (id) => dispatch(TodoActionCreator.toggleDone(id))
        }
    }
}
const TodoListContainer = connect(mapStateToProps, mapDispatchToProps)(TodoList);
export default TodoListContainer;
```

10. Hook + react-redux(2)

react-redux 에서 제공하는 hook

- useSelector()
  - Redux Store의 State 값을 선택으로 리턴함
- useDispatch()
  - Store의 dispatch 함수를 리턴함
  - const dispatch = useDispatch();
- useStore()
  - Store 객체를 리턴함
  - const store = useStore();

61

10. Hook + react-redux(3)

hook을 적용한 Container

- 더 간단하게...

```
.....
import TodoActionCreator from '../redux/TodoActionCreator';
import { useDispatch, useSelector } from 'react-redux';
.....
const TodoListContainer = () => {
    const dispatch = useDispatch()

    var propsObject = {
        states : {
            todolist : useSelector(state => state.todos.todolist),
        },
        callbacks : {
            deleteTodo : (id) => dispatch(TodoActionCreator.deleteTodo(id)),
            toggleDone : (id) => dispatch(TodoActionCreator.toggleDone(id))
        }
    }
    return (
        <TodoList {...propsObject} />
    );
};
export default TodoListContainer;
```


62

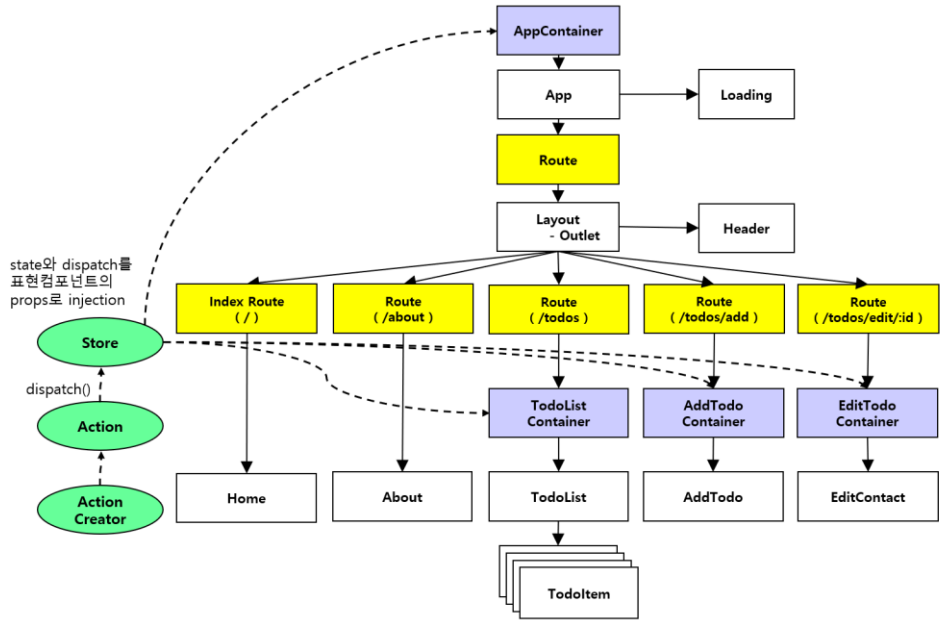
11. todolist + axios + redux 실습(1)

9장에서 작성한 실습 예제(todolistapp)에 redux 적용함.

- 9장까지 작성한 실습 예제 구조와 특징
  - react-router 적용
  - axios 요청
  - AppContainer가 Container 컴포넌트 역할 : 상태와 로직이 집중됨.
- 변경할 내용
  - AppContainer에 집중된 상태와 로직을 Redux 구성요소로 이전
  - AppContainer는 connect() 고차함수 또는 redux 관련 훅을 이용해 생성함.
    - 이와 더불어 각 Route가 렌더링할 컴포넌트에 속성을 전달하는 Container를 직접 생성함.
    - 이를 통해 props - props - props - .... 를 적절하게 최소화할 것임.
- 환경 설정
  - yarn add redux react-redux redux-thunk
  - yarn add -D redux-devtools-extension
- todosvc 백엔드 API 서비스를 미리 실행해둘 것

11. todolist + axios + redux 실습(2)

전체 변경 구조



64

■ Store의 상태는 connect() 고차 함수 또는 redux hook(useSelector)을 이용해 필요한 Container 컴포넌트를 작성함.

■ ActionCreator의 메서드들은 connect() 고차 함수 또는 redux hook(useDispatch)를 이용해 필요한 컴포넌트로 전달함.

■ 위와 같은 작업을 통해 Route 컴포넌트를 거쳐 props로 전달하지 않아도 됨

- 짧은 경로만 props로 전달함
  - 예) TodoList -> TodoItem, App -> Loading

11. todolist + axios + redux 실습(3)

src/redux/TodoReducer.js 작성

```
import produce from 'immer';

export const TODO_ACTION = {
    FETCH_TODOLIST_REQUEST : "fetchTodoList_request",
    FETCH_TODOLIST_SUCCESS : "fetchTodoList_success",
    FETCH_TODOLIST_FAIL : "fetchTodoList_fail",
    ADD_TODO_REQUEST : "addTodo_request",
    ADD_TODO_SUCCESS : "addTodo_success",
    ADD_TODO_FAIL : "addTodo_fail",
    DELETE_TODO_REQUEST : "deleteTodo_request",
    DELETE_TODO_SUCCESS : "deleteTodo_success",
    DELETE_TODO_FAIL : "deleteTodo_fail",
    TOGGLE_DONE_REQUEST : "toggleDone_request",
    TOGGLE_DONE_SUCCESS : "toggleDone_success",
    TOGGLE_DONE_FAIL : "toggleDone_fail",
    UPDATE_TODO_REQUEST : "updateTodo_request",
    UPDATE_TODO_SUCCESS : "updateTodo_success",
    UPDATE_TODO_FAIL : "updateTodo_fail",
}

const initialState = {
    todolist : [],
    isLoading : false
}

(다음 페이지로 이어짐)
```

11. todolist + axios + redux 실습(4)

src/redux/TodoReducer.js 작성

```
const TodoReducer = (state=initialState, action) => {
    switch(action.type) {
        case TODO_ACTION.ADD_TODO_REQUEST :
        case TODO_ACTION.DELETE_TODO_REQUEST :
        case TODO_ACTION.TOGGLE_DONE_REQUEST :
        case TODO_ACTION.UPDATE_TODO_REQUEST :
            return { ...state, isLoading : true };
        case TODO_ACTION.FETCH_TODOLIST_REQUEST :
            return { ...state, isLoading : true, todolist: [] };
        case TODO_ACTION.FETCH_TODOLIST_FAIL :
        case TODO_ACTION.ADD_TODO_FAIL :
        case TODO_ACTION.DELETE_TODO_FAIL :
        case TODO_ACTION.TOGGLE_DONE_FAIL :
        case TODO_ACTION.UPDATE_TODO_FAIL :
            return { ...state, isLoading: false };
        case TODO_ACTION.FETCH_TODOLIST_SUCCESS :
            return { ...state, isLoading:false, todolist: action.payload.todolist };
        case TODO_ACTION.ADD_TODO_SUCCESS :
            return produce(state, draft=>{
                draft.todolist.push(action.payload.todoitem);
                draft.isLoading = false;
            })

(다음 페이지로 이어짐)
```


66

11. todolist + axios + redux 실습(5)

src/redux/TodoReducer.js 작성

```
        case TODO_ACTION.DELETE_TODO_SUCCESS :
            return produce(state, draft=>{
                let index = draft.todolist.findIndex((todo)=>todo.id === parseInt(action.payload.id,10))
                draft.todolist.splice(index, 1);
                draft.isLoading = false;
            })
        case TODO_ACTION.TOGGLE_DONE_SUCCESS :
            return produce(state, draft=>{
                let index = draft.todolist.findIndex((todo)=>todo.id === parseInt(action.payload.id,10))
                draft.todolist[index].done = !draft.todolist[index].done;
                draft.isLoading = false;
            })
        case TODO_ACTION.UPDATE_TODO_SUCCESS :
            return produce(state, draft=>{
                let index = draft.todolist.findIndex((todo)=>todo.id === parseInt(action.payload.todoitem.id,10))
                draft.todolist[index] = action.payload.todoitem;
                draft.isLoading = false;
            })
        default :
            return state;
    }
}

export default TodoReducer;
```

11. todolist + axios + redux 실습(6)

src/redux/TodoActionCreator.js

```
import axios from 'axios';
import { TODO_ACTION } from './TodoReducer';
const USER = "gdhong";
const BASEURI = "/api/todolist_long/" + USER;

const TodoActionCreator = {
    fetchTodoListRequest : ()=> ({ type :  TODO_ACTION.FETCH_TODOLIST_REQUEST }),
    addTodoRequest : ()=> ({ type :  TODO_ACTION.ADD_TODO_REQUEST }),
    toggleDoneRequest : ()=> ({ type :  TODO_ACTION.TOGGLE_DONE_REQUEST }),
    updateTodoRequest : ()=> ({ type :  TODO_ACTION.UPDATE_TODO_REQUEST }),
    deleteTodoRequest : ()=> ({ type :  TODO_ACTION.DELETE_TODO_REQUEST }),
    fetchTodoListSuccess : (todolist)=> ({ type :  TODO_ACTION.FETCH_TODOLIST_SUCCESS, payload: { todolist } }),
    addTodoSuccess : (todoitem)=>({ type :  TODO_ACTION.ADD_TODO_SUCCESS, payload: { todoitem } }),
    deleteTodoSuccess : (id)=>({ type:  TODO_ACTION.DELETE_TODO_SUCCESS, payload: { id } }),
    toggleDoneSuccess : (id)=>({ type:  TODO_ACTION.TOGGLE_DONE_SUCCESS, payload: { id } }),
    updateTodoSuccess : (todoitem)=>({ type:  TODO_ACTION.UPDATE_TODO_SUCCESS, payload: { todoitem } }),
    fetchTodoListFail : ()=> ({ type :  TODO_ACTION.FETCH_TODOLIST_FAIL }),
    addTodoFail : ()=> ({ type :  TODO_ACTION.ADD_TODO_FAIL }),
    deleteTodoFail : ()=> ({ type :  TODO_ACTION.DELETE_TODO_FAIL }),
    toggleDoneFail : ()=> ({ type :  TODO_ACTION.TOGGLE_DONE_FAIL }),
    updateTodoFail : ()=> ({ type :  TODO_ACTION.UPDATE_TODO_FAIL }),

(다음 페이지로 이어짐)
```

11. todolist + axios + redux 실습(7)

```
asyncFetchTodoList : (failCallback)=> {
    return (dispatch, getState)=> {
        dispatch({ type:  TODO_ACTION.FETCH_TODOLIST_REQUEST });
        axios.get(BASEURI)
        .then((response)=> {
            dispatch({ type: TODO_ACTION.FETCH_TODOLIST_SUCCESS, payload : { todolist: response.data } });
        })
        .catch((error)=>{
            failCallback("할일 조회 실패 : " + error);      dispatch({ type: TODO_ACTION.FETCH_TODOLIST_FAIL });
        })
    }
},
asyncAddTodo : (todo, desc, successCallback, failCallback) => {
    return (dispatch, getState)=> {
        dispatch({ type:  TODO_ACTION.ADD_TODO_REQUEST });
        axios.post(BASEURI, { todo, desc })
        .then((response)=>{
            if (response.data.status === "success") {
                dispatch({ type: TODO_ACTION.ADD_TODO_SUCCESS,
                        payload : { todoitem: { ...response.data.item, done:false } } });
                successCallback();
            } else {
                dispatch({ type: TODO_ACTION.ADD_TODO_FAIL });
                failCallback("할일 추가 실패 : " + response.data.message);
            }
        })
        .catch((error)=>{
            dispatch({ type: TODO_ACTION.ADD_TODO_FAIL });     failCallback("할일 추가 실패 : " + error);
        })
    }
},
```

[제공되는 예제 참조]
- asyncUpdateTodo
- asyncDeleteTodo
- asyncToggleDone

■ 다음 3개의 메서드는 제공되는 예제를 확인한다. (todolistapp-axios-redux-router11)
- asyncUpdateTodo
- asyncDeleteTodo
- asyncToggleDone

■ TodoActionCreator.js에서 비동기 처리가 완료되거나 실패했을 때를 위해서 successCallback, failCallback 파라미터를 사용했음
- successCallback : asyncAddTodo, asyncUpdateTodo 메서드에서 사용함. 이 경우는 할일 추가, 수정이 성공했으므로 /todos로 이동하도록 하는 기능을 수행함.
- failCallback : 에러메시지를 메시지박스로 보여주는 기능 실행

■ TodoActionCreator에서 직접 /todos로 이동시키거나 메시지 박스를 보여주지 않는 이유
- 핵심 비즈니스 로직이 아니라 UI와 관련된 기능이므로 UI(Component) 측에서 기능을 정의하는 것이 올바른 접근법이다.

11. todolist + axios + redux 실습(8)

src/redux/AppStore.js

```
import { createStore, applyMiddleware } from 'redux';
import TodoReducer from './TodoReducer';
import thunk from 'redux-thunk';
import TodoActionCreator from './TodoActionCreator';
import { composeWithDevTools } from 'redux-devtools-extension';

let AppStore;
if (process.env.NODE_ENV === "development") {
    const composeEnhancers = composeWithDevTools(TodoActionCreator);
    AppStore = createStore(TodoReducer, composeEnhancers(
        applyMiddleware(thunk)
    ));
} else {
    AppStore = createStore(TodoReducer, applyMiddleware(thunk));
}

export default AppStore;
```

11. todolist + axios + redux 실습(9)

src/App.js 변경

```
import React, { useEffect } from 'react';
......
const App = ({ states, callbacks }) => {
  useEffect(()=> {
    callbacks.fetchTodoList();
  }, [])

  return (
    <Router>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="about" element={<About />} />
          <Route path="todos" element={<TodoList states={states} callbacks={callbacks} />} />
          <Route path="todos/add" element={<AddTodo callbacks={callbacks} />} />
          <Route path="todos/edit/:id" element={<EditTodo callbacks={callbacks} />} />
          <Route path="*" element={<NotFound />} />
        </Route>
      </Routes>
      { states.isLoading ? <Loading /> : "" }
    </Router>
  );
};
......(생략)
export default App;
```

71

■ Route를 통해서 props를 전달할 필요가 없음

- 대신 TodoList, AddTodo, EditTodo 컴포넌트에 상태와 액션생성자를 전달할 수 있는 Container를 작성해야 함.

11. todolist + axios + redux 실습(10)

src/AppContainer.js 변경

- 기존 코드를 삭제한 후 다시 작성

```
import React  from 'react';
import App from './App';
import { useSelector, useDispatch } from 'react-redux';
import TodoActionCreator from './redux/TodoActionCreator';

const AppContainer = ()=> {
    const dispatch = useDispatch();
    const failCallback = (message) => {
      alert(message)
    }
    var propsObject = {
        states : { isLoading : useSelector(state => state.isLoading)    },
        callbacks : { fetchTodoList : () => dispatch(TodoActionCreator.asyncFetchTodoList(failCallback)) }
    }

    return (
        <App {...propsObject} />
    );
}

export default AppContainer;
```


72

11. todolist + axios + redux 실습(11)

src/pages/AddTodo.js 변경

```
……
import { useDispatch } from 'react-redux';
import TodoActionCreator from '../redux/TodoActionCreator';

const AddTodo = ({callbacks}) => {
    const navigate = useNavigate();
    let [ todo, setTodo ] = useState('');
    let [ desc, setDesc ] = useState('');

    const addContactHandler = ()=> {
        if (todo.trim() === "" || desc.trim()==="") {
            alert('반드시 할일, 설명을 입력해야 합니다.');
            return;
        }
        callbacks.addTodo(todo, desc, ()=> {
            navigate('/todos');
        });
    }
    return (
        ……(생략)
    );
};

(다음 페이지로 이어짐)
```


■ AddTodo는 AddTodoContainer의 자식 컴포넌트가 되면 Route에 의해 직접 렌더링되는 컴포넌트가 아닌 것이 되므로 history 속성이 전달되지 않는다. 그렇기 때문에 useHistory 훅을 이용해 전달하도록 코드를 변경해야 한다.

11. todolist + axios + redux 실습(12)

src/pages/AddTodo.js 변경

```
AddTodo.propTypes = {
    callbacks :PropTypes.object.isRequired,
};

const AddTodoContainer = (props) => {
    const navigate = useNavigate();
    const dispatch = useDispatch()
    const failCallback = (message)=> {
        navigate('/todos');
        alert(message);
    }
    var propsObject = {
        callbacks : {
            addTodo : (todo, desc, successCallback) =>
                    dispatch(TodoActionCreator.asyncAddTodo(todo, desc, successCallback, failCallback)),
        }
    }
    return (<AddTodo {...propsObject} />);
};

export default AddTodoContainer;
```


74

11. todolist + axios + redux 실습(13)

src/pages/EditTodo.js 변경

```
……
import { useDispatch, useSelector } from 'react-redux';
import TodoActionCreator from '../redux/TodoActionCreator';

const EditTodo = ({ todolist, callbacks }) => {
    const navigate = useNavigate();
    const params = useParams();
    const todoitem = todolist.find((item)=>item.id === parseInt(params.id,10));
    if (!todoitem) {
        navigate('/todos');
    }
    const [ todoOne, setTodoOne ] = useState({ ...todoitem });
    const updateContactHandler = ()=> {
        if (todoOne.todo.trim() === "" || todoOne.desc.trim()==="") {
            alert('반드시 할일, 설명을 입력해야 합니다.');
            return;
        }
        let { id, todo, desc, done } = todoOne;
        callbacks.updateTodo(id, todo, desc, done, ()=> {
            navigate('/todos');
        });
    }
    return (
      ……(생략)
    );
};
```

11. todolist + axios + redux 실습(14)

src/pages/EditTodo.js 변경

```
EditTodo.propTypes = {
    todolist : PropTypes.arrayOf(PropTypes.object).isRequired,
    callbacks :PropTypes.object.isRequired,
};

const EditTodoContainer = () => {
    const dispatch = useDispatch()
    const failCallback = (message)=> {
        alert(message);
    }
    var propsObject = {
        todolist : useSelector(state => state.todolist),
        callbacks : {
            updateTodo : (id, todo, desc, done, successCallback) =>
                    dispatch(TodoActionCreator.asyncUpdateTodo(id, todo, desc, done, successCallback, failCallback)),
        }
    }
    return (<EditTodo {...propsObject} />);
};

export default EditTodoContainer;
```


76

11. todolist + axios + redux 실습(15)

src/pages/TodoList.js 변경

```
……
import TodoActionCreator from '../redux/TodoActionCreator';
import { useDispatch, useSelector } from 'react-redux';

……(TodoLIst 컴포넌트 생략)

const TodoListContainer = () => {
    const dispatch = useDispatch()
    const failCallback = (message)=> {
        alert(message);
    }
    var propsObject = {
        states : {    todolist : useSelector(state => state.todolist)  },
        callbacks : {
            fetchTodoList : (id) => dispatch(TodoActionCreator.asyncFetchTodoList(failCallback)),
            deleteTodo : (id) => dispatch(TodoActionCreator.asyncDeleteTodo(id, failCallback)),
            toggleDone : (id) => dispatch(TodoActionCreator.asyncToggleDone(id, failCallback))
        }
    }
    return (
        <TodoList {...propsObject} />
    );
};

export default TodoListContainer;
```


■ 마지막으로 src/index.js의 다음 코드를 변경

- AppStore와 react-redux가 제공하는 Provider 컴포넌트를 import 한 후
- AppContainer를 Provider가 Wrapping할 수 있도록 코드를 작성함.
- Provider를 통해서 AppStore가 제공되어야 함. --> src/index.js 참조

11. todolist + axios + redux 실습(16)

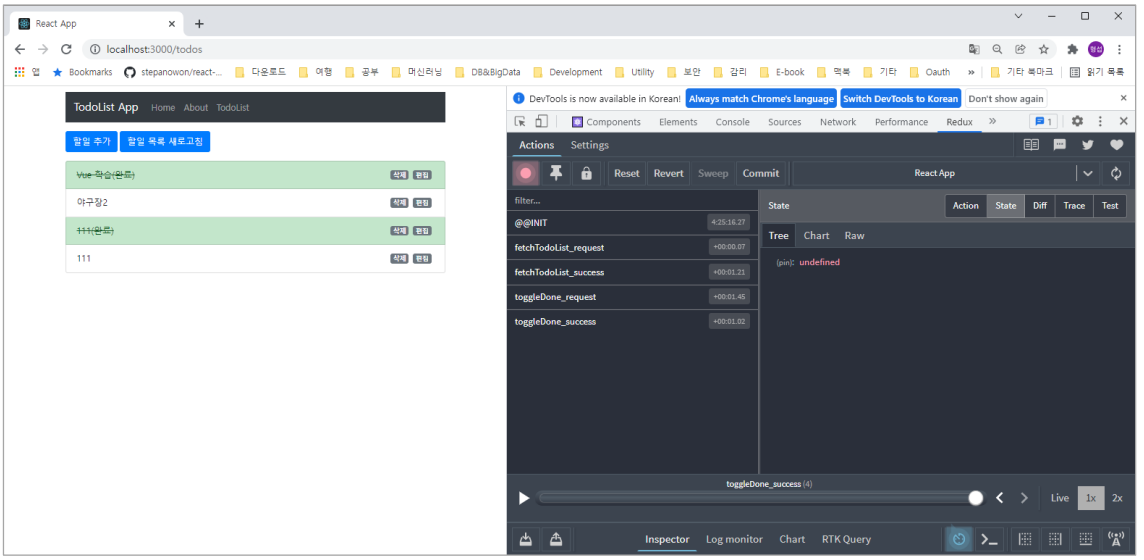
src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import 'bootstrap/dist/css/bootstrap.css';
import './index.css';
import AppContainer from './AppContainer';
import reportWebVitals from './reportWebVitals';

import AppStore from './redux/AppStore';
import { Provider } from 'react-redux';

ReactDOM.render(
  <React.StrictMode>
    <Provider store={AppStore}>
      <AppContainer />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);

reportWebVitals();
```


78

11. todolist + axios + redux 실습(17)

실행 결과



79

12. Context API와 Redux

- **반드시 Redux를 사용해야 하는가?**
  - 그렇지 않음. 간단한 앱이라면 Context API 만으로도 충분함.
  - 오히려 Redux를 적용하기 위해서 더 많은 노력이 들어감
- **그렇다면 언제 Redux를 사용하는가?**
  - 앱의 규모가 커지면 컴포넌트가 많아지고 그로 인해 상태 트리가 복잡해지거나 상태가 여러 컴포넌트에 분산 배치됨
    - 이 경우에 상태와 관련 로직의 관리가 어려워짐.
  - 이런 경우에 Redux를 사용함!!
    - 하지만 Redux는 처음 배우는 개발자에게 어려움.
- **Context API와 Redux이외에 다른 대안은?**
  - mobx : 부록 참조
    - https://mobx.js.org/README.html
  - recoil
    - https://recoiljs.org/