useReducer todo 앱

**학습목표**

useReducer 훅으로 todo 앱 구현

**목차**

[1. todo 앱 2](#_Toc133772067)

[1) 개요 2](#_Toc133772068)

[2) 구현 3](#_Toc133772069)

[3) 평가 8](#_Toc133772070)

[2. React Context DevTool 9](#_Toc133772071)

[1) 개요 9](#_Toc133772072)

[2) 설치 9](#_Toc133772073)

[3) 실행 9](#_Toc133772074)

# todo 앱

## 개요

todoReducer.ts 파일에 다음 사항들을 구현한다.

todo 객체 타입 선언

state 타입 선언

최초 state 값

action 타입 선언  
 reducer 함수 구현

따라서 types.ts 파일이 따로 있을 필요가 없다.

상태 변경 작업들을 reducer 함수에 구현하기 때문에,

TodoContainer.tsx 구현도 필요 없다.

## 구현

### todoReducer.ts

|  |
| --- |
| export type Todo = {      id: number,      title: string,      done: boolean  }  export type State = {      todoList: Todo[]  }  export const initialState = { todoList: [      {id: 1, title: '프론트엔드 과제', done: false},      {id: 2, title: '백엔드 과제', done: true},      {id: 3, title: '웹프 시험공부', done: false}  ] }  let lastId = 3;    export type Action =    { type: "addTodo", payload: { title: string } } |    { type: "toggleTodo", payload: { id: number } } |    { type: "deleteTodo", payload: { id: number } }  export type DispatchFunc = (action: Action) => void;  export type ReducerFunc = (state: State, action: Action) => State;    export const reducer : ReducerFunc = (state: State, action: Action) => {      switch (action.type) {          case "addTodo":              return { todoList: [...state.todoList,                 { id: ++lastId, title: action.payload.title, done: false } ] };          case "deleteTodo":              return { todoList: state.todoList.filter(                  todo => todo.id !== action.payload.id ) };          case "toggleTodo":              return { todoList: state.todoList.map(                  todo => todo.id !== action.payload.id ?                          todo :                          {...todo, done: !todo.done}) };          default:              throw new Error("unknown action type");      }  } |

export type Todo = {

    id: number,

    title: string,

    done: boolean

}

todo 객체의 타입

export type State = {

    todoList: Todo[]

}

todo 앱 상태 타입

export const initialState = { todoList: [

    {id: 1, title: '프론트엔드 과제', done: false},

    {id: 2, title: '백엔드 과제', done: true},

    {id: 3, title: '웹프 시험공부', done: false}

] }

최초 상태 값

let lastId = 3;

마지막 todo 항목의 id 값, 전역 변수

export type Action =

  { type: "addTodo", payload: { title: string } } |

  { type: "toggleTodo", payload: { id: number } } |

  { type: "deleteTodo", payload: { id: number } }

todo 앱 액션 타입

export type DispatchFunc = (action: Action) => void;

todo 앱에서 action을 요청할 때 사용하는 dispatch 함수의 타입

export type ReducerFunc = (state: State, action: Action) => State;

reducer 함수의 타입

reduceer 함수의 파라미터는 현재 상태 값과 요청된 액션.

요청된 액션에 따라 새 상태 값 객체를 생성하여 리턴해야 한다.

export const reducer : ReducerFunc = (state: State, action: Action) => { . . . }

  reducer 함수 구현

화살표 함수 문법으로 구현함.

case "addTodo":

  return { todoList: [...state.todoList,

               { id: ++lastId, title: action.payload.title, done: false } ] };

요청된 action의 type이 "addTodo" 이면, 새 todo 항목이 추가된 새 todo 배열을 만들고

{ todoList: 새\_todo\_배열 } 형태의 새 상태 객체를 만들어 리턴한다.

case "deleteTodo":

return { todoList: state.todoList.filter(todo => todo.id !== action.payload.id ) };

요청된 action의 type이 "deleteTodo" 이면,

action 에 포함되어 전달된 id와 일치하는 todo 항목만 제외된 새 todo 배열을 만들고

{ todoList: 새\_todo\_배열 } 형태의 새 상태 객체를 만들어 리턴한다.

case "toggleTodo":

   return { todoList: state.todoList.map(

                todo => todo.id !== action.payload.id ?

                        todo :

                        {...todo, done: !todo.done}) };

요청된 action의 type이 "toggleTodo" 이면,

action 에 포함되어 전달된 id와 일치하는 todo 항목만 새 todo 객체로 교체된 새 todo 배열을 만들고

{ todoList: 새\_todo\_배열 } 형태의 새 상태 객체를 만들어 리턴한다.

교체한 새 todo 객체의 done 값은, 이전 todo 객체의 done 값을 토글한 값이다.

default:

   throw new Error("unknown action type");

action type 값이 위 3 중 하나가 아니면, 에러를 throw 한다.

무엇인가 예외 상황이 발생하면, 이렇게 에러가 throw 되도록 구현해야 디버깅 할 때 도움이 된다

### TodoItem.tsx

|  |
| --- |
| import React from 'react'  import type { Todo, DispatchFunc } from './todoReducer'  type Props = {    todo: Todo,    dispatch: DispatchFunc  }  function TodoItem({ todo, dispatch }: Props) {    const onChange = () => dispatch({ type: "toggleTodo", payload: { id: todo.id } });    const onClick = () => {      if (window.confirm('삭제하시겠습니까?'))        dispatch({ type: "deleteTodo", payload: { id: todo.id } });    }    return (      <tr className={ todo.done ? "done" : "" }>        <td>{ todo.id }</td>        <td>          <input type='checkbox' checked={ todo.done } onChange={onChange} />          { todo.title }          <span onClick={onClick}>x</span>        </td>      </tr>    )  }  export default TodoItem; |

dispatch({ type: "toggleTodo", payload: { id: todo.id } }

"toggleTodo" 액션 객체를 생성하여 전달한다.

액션 객체가 전달되면, reducer 함수가 호출되어 새 상태 객체가 생성된다.

### TodoList.tsx

|  |
| --- |
| import React from 'react'  import TodoItem from './TodoItem'  import type { Todo, DispatchFunc } from './todoReducer'  type Props = {    todoList: Todo[],    dispatch: DispatchFunc  }  function TodoList({ todoList, dispatch }: Props) {    const trlist = todoList.map(todo =>      <TodoItem key={todo.id} todo={todo} dispatch={dispatch} />);    return (      <table>        <thead>          <tr><td>id</td><td>할일</td></tr>        </thead>        <tbody>          {trlist}        </tbody>      </table>    )  }  export default TodoList |

### TodoInput.tsx

|  |
| --- |
| import React, { ChangeEvent, useState } from 'react'  import type { DispatchFunc } from './todoReducer'  type Props = {    dispatch : DispatchFunc  }  function TodoInput({ dispatch }: Props) {    const [title, setTitle] = useState<string>('');    const onChange = (e: ChangeEvent<HTMLInputElement>) => setTitle(e.target.value);    const onClick = () => {      dispatch({ type: "addTodo", payload: { title } });      setTitle('');    }    return (      <div>        <input type='text' value={title} onChange={onChange} />        <button onClick={onClick}>추가</button>      </div>    )  }  export default TodoInput |

### TodoRoot.tsx

|  |
| --- |
| import React, { useReducer } from 'react'  import './TodoRoot.css'  import TodoInput from './TodoInput';  import TodoList from './TodoList';  import { reducer, initialState } from './todoReducer'  function TodoRoot() {    const [state, dispatch] = useReducer(reducer, initialState)    return (      <div className="todoRoot">        <h1>할일</h1>        <TodoInput dispatch={dispatch} />        <TodoList todoList={state.todoList} dispatch={dispatch} />      </div>    )  }  export default TodoRoot; |

### TodoRoot.css

|  |
| --- |
| .todoRoot { border: 1px solid gray; width: 400px; margin: 10px auto;    box-shadow: 5px 5px 5px #DDD; padding: 30px; }  .todoRoot h1 { margin: 0  0 20px 0; }  .todoRoot div > input { padding: 6px; width: 280px; }  .todoRoot div button { margin-left: 5px; padding: 3px 10px; }  .todoRoot table { border-collapse: collapse; margin-top: 20px; width: 100%; }  .todoRoot td { border: 1px solid gray; padding: 6px; }  .todoRoot thead { background-color: #eee; text-align: center; }  .todoRoot td:nth-child(1) { text-align: center; width: 30px; }  .todoRoot input[type=checkbox] { margin-right: 10px; accent-color: gray; }  .todoRoot span { float: right; font-weight: bold; cursor: pointer; }  .todoRoot tr.done { color: #bbb; text-decoration: line-through; } |

### App.tsx

|  |
| --- |
| import React from 'react';  import TodoRoot from './TodoRoot';  function App() {    return (      <TodoRoot />    );  }  export default App; |

## 평가

### 장점

(1) todoReducer.ts 모듈에서 전체적인 로직을 구현한다. 따라서 컴포넌트 구현이 단순해 진다.

타입 선언, 액션의 종류 선언, 상태 변경 함수(reducer) 구현

(2) props 전달이 단순해 진다.

필요한 상태 데이터와, dispatch 함수만 props로 전달하기 때문에.

(3) 상태 변경 추적이 쉽다.

상태 변경 과정을 보여주는 개발 도구가 있어서 디버깅에 도움이 된다. (뒤에서 소개)

### 단점

(1) 액션 객체를 생성하여 dispatch 하는 구현이 약간 번거롭다.

# React Context DevTool

## 개요

복잡한 앱을 디버깅할 때, 상태 값 변경 과정을 볼 수 있으면 매우 도움이 된다.

상태 값 변경 과정을 기록하려면, 상태 값 변경 요청이 기록되어야 하고 (요청된 action 객체를 기록),

그 요청에 의해 생성된 새 상태 값 객체도 기록되어야 한다 (reducer 함수가 리턴한 새 생태 각체 기록)

이렇게 기록해서 보여주는 도구가 React Context DevTool 이다.

## 설치

google에서 React Context DevTool을 검색하자.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated with medium confidence

## 실행

Graphical user interface, application

Description automatically generated

F12 개발자 창을 열고 React Context 탭을 열자

화면에서 Actions 아래에서, 지금까지 요청된 action 객체 목록을 볼 수 있다.

이 action 객체들 중 하나를 클릭하면, 화면 오른쪽에 다음과 같은 내용을 볼 수 있다.

State: 이 action에 의해서 만들어진 새 상태 값

Action: 이 action 객체의 내용

Diff: 이전 상태와 새 상태에서 어느 부분이 달라졌나?