불변성과 useReducer

**학습목표**

불변성 구현

useReducer 훅

**목차**

[1. 렌더링 2](#_Toc131966092)

[1) 개요 2](#_Toc131966093)

[2) 상태 변화와 렌더링 예제 2](#_Toc131966094)

[2. todo 앱 보충 5](#_Toc131966095)

[1) 개요 5](#_Toc131966096)

[2) todo 앱 수정 5](#_Toc131966097)

[3) 개선할 점 9](#_Toc131966098)

[4) 불변성 구현 10](#_Toc131966099)

[3. useReducer 훅 12](#_Toc131966100)

[1) 개요 12](#_Toc131966101)

[2) useReducer 구성 요소 12](#_Toc131966102)

[3) counter1 예제 13](#_Toc131966103)

[4) 연습문제 16](#_Toc131966104)

# 렌더링

## 개요

리액트는 컴포넌트 단위로 렌더링된다.

state나 props가 변경된 컴포넌트와 그 자식들만 다시 렌더링 되고,

그렇지 않은 컴포넌트는 렌더링 되지 않는다.

따라서 화면을 좀 더 작은 단위로 분할해서 컴포넌트로 구현하는 것이 렌더링 효율에 좋다.

그런데, 컴포넌트 하나 당 파일 하나에 구현해야만 하는 것은 아니다.

소스 코드 파일 하나에 여러 컴포넌트를 구현해도 된다.

앞에서 구현했던 예제를 다시 구현해 보자.

## 상태 변화와 렌더링 예제

### src/App.css

|  |
| --- |
| h1 { font-size: 11pt; margin: 0; }  div.box { display: inline-block; margin: 10px; padding: 10px;    border: 1px solid gray; }  button { display: block; padding: 0.2em 2em; margin: 0.5em; } |

### src/App.tsx 수정

|  |
| --- |
| import React, { useState } from "react";  import "./App.css";  function CounterA() {    console.log("CounterA 호출됨");    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>CounterA</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>      </div>    )  }  function CounterB() {    console.log("CounterB 호출됨");    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>CounterB</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>      </div>    )  }  function GroupComponent() {    console.log('GroupComponent 호출됨');    const [value, setValue] = useState<number>(0);    return (      <div className='box'>        <h1>GroupComponent</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>        <CounterA />        <CounterB />      </div>    )  }  function App() {    console.log('App 호출됨');    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>App</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>        <GroupComponent />      </div>    );  }  export default App; |

CounterA, CounterB, GroupComponent, App 컴포넌트들을 전부 App.tsx 파일 하나에 구현하였다.

Graphical user interface

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

처음 실행될 때, 모든 함수형 컴포넌트가 렌더링 되었다.

CounterA 버튼을 클릭하면

Graphical user interface, text, application

Description automatically generated

CounterA 컴포넌트만 렌더링 되는 것을 확인할 수 있다.

# todo 앱 개선

## 개요

할일 완료(done) 상태 토글 기능과 할일 삭제 기능을 구현하자.

TodoContainer에 toggleTodo, deleteTodo 함수를 구현해야 한다.

toggleTodo, deleteTodo 함수는 다음 경로로 TodoItem 컴포넌트까지 전달되어야 한다. (props 전달)

TodoContainer -> TodoRoot -> TodoList -> TodoItem

따라서 위 컴포넌트들을 전부 수정해야 한다.

기능 하나 추가하려면, 모든 컴포넌트들을 다 수정해야 하다니, 바람직하지 않은 구조이다.

## todo 앱 수정

### src/TodoRoot.css 수정

|  |
| --- |
| .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; } |

.todoRoot input[type=checkbox] { margin-right: 10px; accent-color: gray; }

accent-color 체크박스 색상 서식

.todoRoot tr.done { color: #bbb; text-decoration: line-through; }

<tr class="done"> <tr className="done">

done 속성 값이 true인 todo 항목을 위한 서식

밝은 회색 글자, 텍스트 중간에 가로선이 그어짐

### src/types.ts 수정

|  |
| --- |
| export type Todo = {    id: number,    title: string,    done: boolean  };  export type AddTodoFunc = (text: string) => void;  export type ToogleTodoFunc = (id: number) => void;  export type DeleteTodoFunc = (id: number) => void; |

ToogleTodoFunc

toogleTodo 함수의 타입 선언

DeleteTodoFunc

deleteTodo 함수의 타입 선언

### src/TodoContainer.tsx 수정

|  |
| --- |
| import React, { useState } from 'react'  import type \* as types from './types'  import TodoRoot from './TodoRoot';  const data : types.Todo[] = [    {id: 1, title: '프론트엔드 과제', done: false},    {id: 2, title: '백엔드 과제', done: true},    {id: 3, title: '웹프 시험공부', done: false}  ];  function TodoContainer() {    const [todoList, setTodoList] = useState<types.Todo[]>(data);    const addTodo : types.AddTodoFunc = (title: string) => {      const lastIndex = todoList.length - 1;      const lastId = todoList[lastIndex].id;      const todo = {id: lastId + 1, title: title, done: false};      setTodoList([...todoList, todo]);    }    const toggleTodo : types.ToogleTodoFunc = (id: number) => {      for (let todo of todoList)        if (todo.id === id) {          todo.done = !todo.done;          setTodoList([...todoList]);          return;        }    }    const deleteTodo : types.DeleteTodoFunc = (id: number) => {      setTodoList( todoList.filter(todo => todo.id !== id));    }      return <TodoRoot todoList={todoList}                     addTodo={addTodo}                     toggleTodo={toggleTodo}                     deleteTodo={deleteTodo} />;  }  export default TodoContainer |

for (let todo of todoList)

 if (todo.id === id) {

   todo.done = !todo.done;

   setTodoList([...todoList]);

   return;

 }

전달된 id 값에 해당하는 todo 항목을 todoList에서 찾아서

done 속성값을 변경하고

todoList 배열을 복제해서 setTodoList 상태 변경 setter 함수를 호출한다.

배열을 복제해서 setter 함수를 호출하지 않고, 단지 배열 내부 값만 변경해서는

화면이 렌더링 되지 않는 다는 것을 앞에서 살펴봤다.

setTodoList( todoList.filter(todo => todo.id !== id));

전달된 id 값에 해당하는 todo 항목만 제외하고, 나머지 항목들만 들어있는 새 배열을 만들어서

setTodoList 상태 변경 setter 함수를 호출한다.

### src/TodoRoot.tsx

|  |
| --- |
| import React from 'react'  import './TodoRoot.css'  import type \* as types from './types'  import TodoInput from './TodoInput';  import TodoList from './TodoList';  type Props = {    todoList: types.Todo[],    addTodo: types.AddTodoFunc,    toggleTodo: types.ToogleTodoFunc,    deleteTodo: types.DeleteTodoFunc  }  function TodoRoot({ todoList, addTodo, toggleTodo, deleteTodo }: Props) {    return (      <div className="todoRoot">        <h1>할일</h1>        <TodoInput addTodo={addTodo} />        <TodoList todoList={todoList} toggleTodo={toggleTodo} deleteTodo={deleteTodo} />      </div>    )  }  export default TodoRoot; |

### src/TodoList.tsx

|  |
| --- |
| import React from 'react'  import type \* as types from './types'  import TodoItem from './TodoItem'  type Props = {    todoList: types.Todo[],    toggleTodo: types.ToogleTodoFunc,    deleteTodo: types.DeleteTodoFunc  }  function TodoList({ todoList, toggleTodo, deleteTodo }: Props) {    const trlist = todoList.map(todo =>      <TodoItem todo={todo} key={todo.id} toggleTodo={toggleTodo} deleteTodo={deleteTodo} />);    return (      <table>        <thead>          <tr><td>id</td><td>할일</td></tr>        </thead>        <tbody>          {trlist}        </tbody>      </table>    )  }  export default TodoList |

### src/TodoItem.tsx

|  |
| --- |
| import React from 'react'  import type \* as types from './types'  type Props = {    todo: types.Todo,    toggleTodo: types.ToogleTodoFunc,    deleteTodo: types.DeleteTodoFunc  }  function TodoItem({ todo, toggleTodo, deleteTodo }: Props) {    const onChange = () => toggleTodo(todo.id)    const onClick = () => {      if (window.confirm('삭제하시겠습니까?'))        deleteTodo(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; |

  const onChange = () => toggleTodo(todo.id)

  <input type='checkbox' checked={ todo.done } onChange={onChange} />

체크 박스의 체크 상태가 달라질 때 마다 toggleTodo 함수가 호출된다.

toggleTodo 함수가 호출되면,

해당 todo의 done 값이 바뀌고

todoList 상태값이 변경되어 TodoContainer와 그 아래 컴포넌트들이 다시 렌더링 된다.

  const onClick = () => {

    if (window.confirm('삭제하시겠습니까?'))

      deleteTodo(todo.id)

  }

  <span onClick={onClick}>x</span>

todo 항목 오른쪽 x 문자를 클릭하면,

deleteTodo 함수가 호출되고,

해당 todo 항목만 제외된 새 todoList 배열이 만들어져

todoList 생타값이 변경되고 TodoContainer와 그 아래 컴포넌트들이 다시 렌더링 된다.

Graphical user interface, text

Description automatically generated

## 개선할 점

### 렌더링 비효율

Todo 항목 한 개만 변경되었는데,

todoList 상태값이 변경되어

TodoContainer부터 전체 컴포넌트들이 다시 렌더링 되는 것은 비효율적이다.

(나중에 개선 함)

### 바람직하지 않은 유지보수

todo 삭제 기능 하나 추가하려면

TodoContainer, TodoRoot, TodoList, TodoItem 모든 컴포넌트들을 다 수정해야 했다.

TodoContainer에 구현된 deleteTodo 함수를

TodoContainer -> TodoRoot -> TodoList -> TodoItem 경로로 전달해야 하기 때문이다.

(나중에 개선 함)

### 불변성(immutable)

앞의 todo 앱 구현은, 항목 하나의 done 값이 변경될 때,

그 항목 뿐만 아니라 전체 todo 목록이 다시 렌더링된다.

done 값이 변경된 항목만 렌더링 되는, 렌더링 최적화가 필요하다.

렌더링 최적화를 구현하려면, 먼저 상태 불변성을 구현해야 한다.

리액트가 상태값 변경을 파악할 때, 배열이나 객체 내부까지 다 비교하지 않고,

배열이나 객체의 참조값만 비교한다는 것을 앞에서 설명했다.

만약 리액트 상태값이 복잡한 구조라면?

예를 들어서 리액트 상태 변수가 배열이고 배열의 원소가 객체이라면?

배열이나 객체의 내부 값을 변경하려면,

원본 배열이나 객체의 내부 값을 변경하지 말고,

복제를 만들어서 복제를 변경해야 한다.

리액트 상태값은 상수처럼 다뤄야 한다. (immutable)

리액트 상태값을 구성하는 배열이나 객체 내부를 수정하면 안된다.

앞의 todo 앱 구현은 상태값 변경을 immutable하게 구현하지 않았다.

꼭 필요한 부분만 렌더링되는, 렌더링 최적화를 구현하려면,

상태값 변경을 immutable하게 구현해야 한다.

## 불변성 구현

### immutable 하지 않은 구현

|  |
| --- |
| function TodoContainer() {    const [todoList, setTodoList] = useState<types.Todo[]>(data);    const addTodo : types.AddTodoFunc = (title: string) => {      const lastIndex = todoList.length - 1;      const lastId = todoList[lastIndex].id;      const todo = {id: lastId + 1, title: title, done: false};      setTodoList([...todoList, todo]);    }    const toggleTodo : types.ToogleTodoFunc = (id: number) => {      for (let todo of todoList)        if (todo.id === id) {          todo.done = !todo.done;          setTodoList([...todoList]);          return;        }    }    const deleteTodo : types.DeleteTodoFunc = (id: number) => {      setTodoList( todoList.filter(todo => todo.id !== id) );    } |

### addTodo

setTodoList([...todoList, todo]);

새 배열을 만들어 setTodoList 함수를 호출한다

이 새 배열에는, 기존의 todoList 배열의 모든 원소가 전개되어 들어가고,

새 todo 객체가 끝에 추가된다.

immutable한 구현이다.

### toggleTodo

for (let todo of todoList)

 if (todo.id === id) {

todo.done = !todo.done;

setTodoList([...todoList]);

todoList 배열의 원소인 todo 객체의 done 값이 변경되었으니 immutable한 구현이 아니다.

let newTodoList = todoList.map(

 todo => todo.id != id ? todo : {...todo, done: !todo.done }

);

setTodoList(newTodoList);

todoList.map 메소드에 의해서 새 배열이 만들어진다 (newTodoList)

todo => todo.id != id ? todo : {...todo, done: !todo.done }

todoList 배열의 각각의 원소에 대해서, 위 화살표 함수가 호출된다.

이 화살표 함수가 리턴하는 값들이 새 배열의 원소가 된다.

id가 일치하지 않으면, 기존의 todo 객체가, 새 배열의 원소가 된다.

id가 일치하면, 그 todo 객체를 복제하고, done 값만 토글하여만들 새 객체가, 새 배열의 원소가 된다.

위 녹새 코드에 의해서 새 배열과 새 객체가 만들어졌을 뿐, 수정된 것이 아무 것도 없다.

immutable한 구현이다.

### deleteTodo

setTodoList( todoList.filter(todo => todo.id !== id) );

todoList 배열에서 id가 일치하는 todo 항목만 제외된 새 배열을 만들어서

setter 함수를 호출한다. 기존 배열이 수정된 것이 아니므로 immutable한 구현이다.

### src/TodoContainer.tsx - immutable한 구현

|  |
| --- |
| import React, { useState } from 'react'  import type \* as types from './types'  import TodoRoot from './todoRoot';  const data : types.Todo[] = [    {id: 1, title: '프론트엔드 과제', done: false},    {id: 2, title: '백엔드 과제', done: true},    {id: 3, title: '웹프 시험공부', done: false}  ];  function TodoContainer() {    const [todoList, setTodoList] = useState<types.Todo[]>(data);    const addTodo : types.AddTodoFunc = (title: string) => {      const lastIndex = todoList.length - 1;      const lastId = todoList[lastIndex].id;      const todo = {id: lastId + 1, title: title, done: false};      setTodoList([...todoList, todo]);    }    const toggleTodo : types.ToogleTodoFunc = (id: number) => {      let newTodoList = todoList.map(        todo => todo.id !== id ? todo : {...todo, done: !todo.done }      );      setTodoList(newTodoList);    }    const deleteTodo : types.DeleteTodoFunc = (id: number) => {      setTodoList( todoList.filter(todo => todo.id !== id) );    }    return <TodoRoot todoList={todoList}                     addTodo={addTodo}                     toggleTodo={toggleTodo}                     deleteTodo={deleteTodo} />;  }  export default TodoContainer |

# useReducer 훅

## 개요

useState를 이용하여 리액트 상태를 만들고 상태값을 변경하는 기능을 구현해봤다.

리액트에서는 useState 같은 것을 훅(hook)이라고 부른다.

useState 훅이 제공하는 setter 함수를 이용해서 상태값을 변경하는 것은

간단한 구조의 컴포넌트에 적당하다.

복잡한 구조의 컴포넌트에서 상태를 구현할 때는 useReducer 훅이 좋다.

useReducer 상태값이 변경된 과정 전체를 볼 수 있도록 상태값들을 관리한다.

이것은 디버깅할 때 매우 유용하다.

따라서 상태값 변경은 반드시 immutable하게 구현해야 한다.

|  |
| --- |
| 참고:  useReducer 훅이 필요할 정도로 복잡한 상태 관리가 필요한 앱은,  겨우 todo 앱 정도 보다 훨씬 더 기능이 많고 복잡하고 디버깅이 힘든 앱이다.  예: facebook 앱  플젝할 땐, 아마도 useState 훅의 기능 정도면 충분할 것이다.  그렇지만 useReducer 훅의 기능을 이해해야 한다. |

## useReducer 구성 요소

useState 훅의 구성 요소는 다음과 같다.

const [state, setter] = useState(initialState)

useReducer 훅의 구성 요소는 다음과 같다.

const [state, dispatch] = useReducer(reducer, initialState)

useState 훅의 setter로 구현했던 것을

dispatch와 reducer로 나뉜다.

### state

현재 상태

### action

현재 상태를 변경하기 위한 명령에 해당하는 객체

이 객체에는 다음 정보가 들어있어야 한다.

type: 어떤 상태 변경을 요구하는가? (type 값은 문자열 상수이어야 함)

payload: 그 상태 변경에 필요한 데이터는?

(payload 값은 객체. action을 수행하기 위해 필요한 데이터)

### function reducer(state, action) 함수

현재 상태와, 상태 변경 요구 명령을 전달 받아서

새 상태값을 만들어서 리턴해야 한다.

immutable 하게 구현해야 한다.

### function dispatch(action) 함수

reducer 함수를 직접 호출할 수는 없고,

dispatch 함수를 통해 간접 호출해야 한다.

dispatch(action) 함수를 호출하면

reducer(현재상태,action) 함수가 호출된다.

## counter1 예제

### 프로젝트 생성

|  |
| --- |
| npx create-react-app counter1 --template typescript |

### src/counterReducer.ts

|  |
| --- |
| type StateType = {    count: number,    step: number  }  export type ActionType =    { type: "increaseCount" } |    { type: "decreaseCount" } |    { type: "setStep", payload: number }  export const initialState: StateType = { count: 0, step: 1 }  export function reducer(state: StateType, action: ActionType) {    switch (action.type) {      case "increaseCount":        return { ...state, count: state.count + state.step }      case "decreaseCount":        return { ...state, count: state.count - state.step }      case "setStep":        return { ...state, step: action.payload }    }  } |

type StateType = {

  count: number,

  step: number

}

상태값의 타입 선언이다.

export type ActionType =

  { type: "increaseCount" } |

  { type: "decreaseCount" } |

  { type: "setStep", payload: { step: number } }

상태값을 변경하기 위한 액션들의 타입 선언이다.

위 코드는 타입스크립트 타입 선언 문법으로 구현되었다.

ActionType 타입의 액션은 다음 3 타입 중 하나 이어야 한다.

3 타입 | or 연산자로 결합

  { type: "increaseCount" } |

  { type: "decreaseCount" } |

  { type: "setStep", payload: number }

{ type: "increaseCount" } 타입

type 속성만 가지고 있는 객체이어야하고, 그 객체의 값은 "increaseCount" 이어야 한다.

{ type: "decreaseCount" } 타입

type 속성만 가지고 있는 객체이어야하고, 그 객체의 값은 " decreaseCount" 이어야 한다.

{ type: "setStep", payload: number }

type 속성과 payload 속성을 가진 객체이어야 하고,

type 속성값은 "setStep" 이어야 하고

payload 속성값은 number 이어야 한다.

(참고: payload 단어의 뜻을 찾아보라)

export const initialState: StateType = { count: 0, step: 1 }

최초 상태값 상수

export function reducer(state: StateType, action: ActionType) {

  switch (action.type) {

    case "increaseCount":

      return { ...state, count: state.count + state.step }

    case "decreaseCount":

      return { ...state, count: state.count - state.step }

    case "setStep":

      return { ...state, step: action.payload }

  }

}

상태값을 변경하기 위한 함수

state 파라미터: 현재 상태값

action 파라미터: 상태값 변경을 위해 요청된 action

이 함수는 반드시 immutable하게 구현되어야 한다.

즉 현재 상태값의 내용을 전혀 변경하지 않고, 새 상태값을 생성해서 리턴해야 한다.

case "increaseCount":

 return { ...state, count: state.count + state.step }

요청된 action의 type값이 "increaseCount" 이면

현재 상태 객체를 복제해서 새 상태 객체를 만들고,

새 상태 객체의 count 값을 현재 상태의 count 보다 step 만큼 증가하고

새 상태 객체를 리턴한다.

case "setStep":

 return { ...state, step: action.payload }

요청된 action의 type값이 "setStep" 이면

현재 상태 객체를 복제해서 새 상태 객체를 만들고,

새 상태 객체의 step 값을 action에 포함된 payload 값으로 변경하고

새 상태 객체를 리턴한다.

### src/Counter.css

|  |
| --- |
| .Counter { padding: 20px; }  .Counter button { padding: 0.5em 1.5em; margin-right: 1em; }  .Counter input { padding: 0.4em; } |

### src/Counter.tsx

|  |
| --- |
| import React, { useReducer, ChangeEvent } from 'react'  import { reducer, initialState } from './counterReducer'  import type { ActionType } from './counterReducer'  import './Counter.css'  function Counter() {    const [state, dispatch] = useReducer(reducer, initialState)    const onIncrease = () => {      const action: ActionType = { type: 'increaseCount' }      dispatch(action);    }    const onDecrease = () => {      const action: ActionType = { type: 'decreaseCount' }      dispatch(action);    }    const onChange = (e: ChangeEvent<HTMLInputElement>) => {      const action: ActionType = { type: 'setStep',                                   payload: parseInt(e.target.value) }      dispatch(action);    }    return (      <div className="Counter">        <h1>Counter</h1>        <p>{ state.count }</p>        <button onClick={onIncrease}>increase</button>        <button onClick={onDecrease}>decrease</button>        <hr />        <input type='number' onChange={onChange} value={state.step} />      </div>    )  }  export default Counter |

const [state, dispatch] = useReducer(reducer, initialState)

useReducer 훅 호출

counterReducer.ts 파일에서 구현한 reducer 함수와 initialState를 파라미터로 전달

리턴된 state는 현재 상태값,

dispatch는 상태 변경을 위해 호출할 함수 (useState 훅의 setter에 해당하는 함수)

dispatch 함수 호출에 의해 reducer 함수가 호출되어 상태가 변경된다.

const onIncrease = () => {

  const action: ActionType = { type: 'increaseCount' }

  dispatch(action);

}

increase 버튼이 클릭되었을 때 호출될 함수.

상태값을 변경하기 위한 action 객체를 생성하여 dispatch 함수를 호출해야 한다.

## src/App.tsc 수정

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

## 연습문제

### state

{ message: string, count: number }

message:

input 태그에 입력된 값을 입력하고 ok 버튼을 클릭하면

state의 message에 값이 대입된다.

count:

increase 버튼을 클릭하면 state의 count 값이 1 증가한다.

### action

type ActionType =

{ type: "changeMessage", payload: string } |

{ type: "increaseCount }

위 기능을 useReducer 훅으로 구현하기.

Message 컴포넌트

Counter 컴포넌트

------

위 기능을 useState 훅만으로 구현하기

action 함수들 구현

function changeMessage(message: string)

functoin increaseCount()

## 연습문제 useState 구현

### types.ts

|  |
| --- |
| export type State = {      message: string,      count: number  };  export type SetMessageFunc = (message: string) => void;  export type IncreaseFunc = () => void; |

### Counter.tsx

|  |
| --- |
| import React from 'react'  import type { IncreaseFunc } from './types'  type Props = {      count: number,      increase: IncreaseFunc  }  function Counter({ count, increase }: Props) {    return (      <div>          <p>count: { count }</p>          <button onClick={increase}>increase</button>      </div>    )  }  export default Counter |

### Message.tsx

|  |
| --- |
| import React, { ChangeEvent, useState } from 'react'  import type { SetMessageFunc } from './types'  type Props = {      message: string,      setMessage: SetMessageFunc  }  function Message({ message, setMessage}: Props) {    const [msg, setMsg] = useState(message);    const onChange = (e: ChangeEvent<HTMLInputElement>) => setMsg(e.target.value);    const onClick = () => setMessage(msg);    return (      <div>          <input type="text" onChange={onChange} value={msg} />          <button onClick={onClick}>ok</button>      </div>    )  }  export default Message |

### App.tsx

|  |
| --- |
| import React, { useState } from 'react';  import Counter from './Counter';  import Message from './Message';  import { State, IncreaseFunc, SetMessageFunc } from './types';  function App() {    const initialState: State = { message: 'hello', count: 0 };    const [state, setState] = useState<State>(initialState);    const increase: IncreaseFunc =              () => setState({...state, count: state.count + 1});    const setMessage: SetMessageFunc =              (msg: string) => setState({...state, message: msg })    return (      <div style={{ textAlign: "center" }}>        <p>message: { state.message }</p>        <Message message={state.message} setMessage={setMessage} />        <hr />        <Counter count={state.count} increase={increase} />      </div>    );  }  export default App; |

## 연습문제 useReducer 구현

### myReducer.ts

|  |
| --- |
| export type State = {      message: string,      count: number  };  export type Action =    { type: "setMessage", payload: string } |    { type: "increase" }  export const initialState = { message: 'hello', count: 0 };  export type DispatchFunc = (action: Action) => void;  export function reducer(state: State, action: Action) {      switch (action.type) {          case "increase":            return {...state, count: state.count + 1};          case "setMessage":            return {...state, message: action.payload };      }  } |

### Counter.tsx

|  |
| --- |
| import React from 'react'  import { DispatchFunc } from './myReducer'  type Props = {      count: number,      dispatch: DispatchFunc  }  function Counter({ count, dispatch }: Props) {    const onClick = () => dispatch({type: 'increase'})    return (      <div>          <p>count: { count }</p>          <button onClick={onClick}>increase</button>      </div>    )  }  export default Counter |

### Message.tsx

|  |
| --- |
| import React, { ChangeEvent, useState } from 'react'  import { DispatchFunc } from './myReducer'  type Props = {      message: string,      dispatch: DispatchFunc  }  function Message({ message, dispatch}: Props) {    const [msg, setMsg] = useState(message);    const onChange = (e: ChangeEvent<HTMLInputElement>) => setMsg(e.target.value);    const onClick = () => dispatch({type:'setMessage', payload: msg});    return (      <div>          <input type="text" onChange={onChange} value={msg} />          <button onClick={onClick}>ok</button>      </div>    )  }  export default Message |

### App.tsx

|  |
| --- |
| import React, { useReducer } from 'react';  import Counter from './Counter';  import Message from './Message';  import { initialState, reducer } from './myReducer';  function App() {    const [state, dispatch] = useReducer(reducer, initialState);    return (      <div style={{ textAlign: "center" }}>        <p>message: { state.message }</p>        <Message message={state.message} dispatch={dispatch} />        <hr />        <Counter count={state.count} dispatch={dispatch} />      </div>    );  }  export default App; |