# 상태관리-Zustand

# 1. Zustand 소개

## ❖Zustand란?

- 독일어로 '상태' 라는 뜻
- 작고 빠른 확장가능한 상태관리 라이브러리

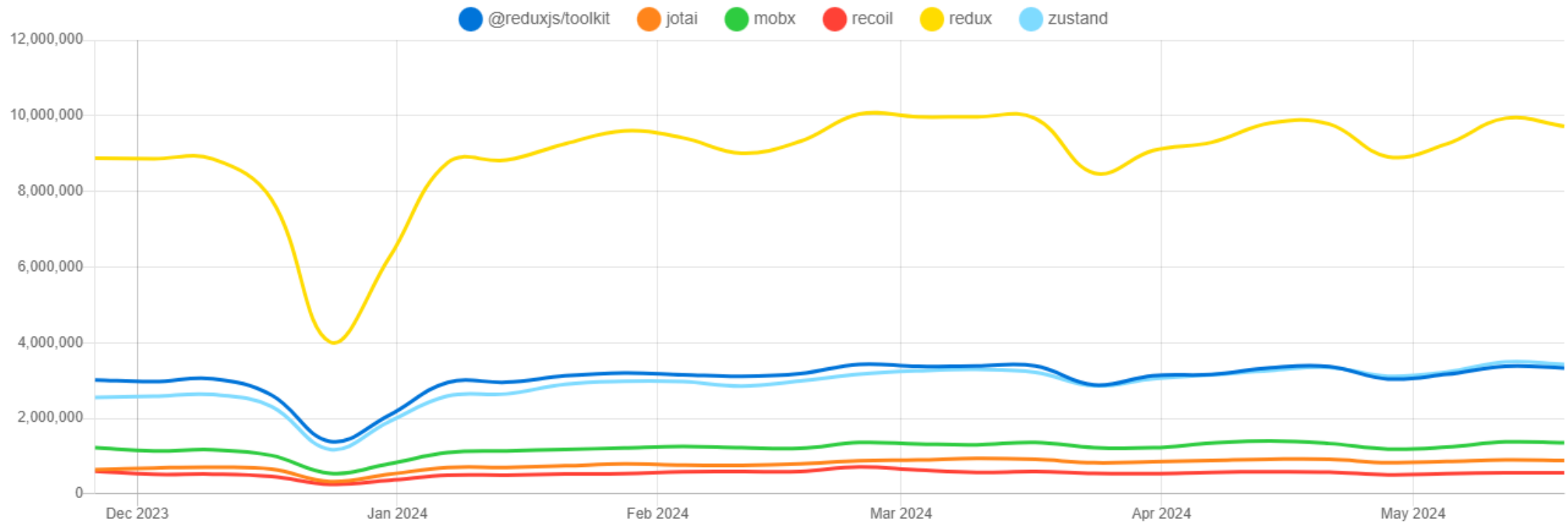
## ❖특징

- 직관적이고 간단한 사용방법
- 불변상태 모델 사용
  - 상태를 변경할 때 불변성을 가진 변경을 수행해야 함. 즉 새로운 상태를 생성하여 리턴함
- ContextProvider 필요 없음
  - Redux, Recoil 등은 Provider, Root 요소가 필요하지만 zustand는 필요하지 않음
- 다중 스토어 지원
  - 필요에 따라  Store를 여러개 만들 수 있음
- 다중 디스패처 지원
  - Redux는 Store가 지원하는 dispatch() 함수를 이용하지만 zustand는 여러개의 dispatch를 만들 수 있으므로 reducer가 필수가 아님
- 디스패처도 상태의 일부로 간주하여 처리

# 1. Zustand 소개

❖최근 트렌드

- redux 〉 zustand 〉 @reduxjs/toolkit 〉 ...



# 2. Zustand 사용법

❖설치

- npm install zustand

❖기초 사용 방법

- Store를 생성하고 Store에 접근하기 위한 훅 생성
  - create〈T〉() 함수 이용

```
//인터페이스 또는 타입을 이용해 상태의 타입을 선언함
interface StateType {
  count: number;
  increment: (num: number) => void;
}
// create<T>()에 의해 생성된 stateCreater 함수에 initializer(slice) 함수를 인자로 전달해 store를 이용할 수 있는 훅 생성
// initializer 함수는 초기화된 상태 객체를 리턴하도록 작성함
// initializer 함수 : (set)=> ({...})
export const useCountStore = create<StateType>()((set) => ({
  count: 0,
  increment: () => {
    set((state) => ({ count: state.count + 1 }));
  },
}));
```

# 2. Zustand 사용법

## ❖기초 사용 방법(이어서)

- 컴포넌트에서 사용하는 방법
  - Store 접근하기 위해 생성한 훅을 이용하여 상태 중 컴포넌트에서 필요로 하는 값을 바인딩

```
const Home = () => {
  const count = useCountStore((state) => state.count);
  const increment = useCountStore((state) => state.increment);

  return (
    <div className="card card-body">
      <div>
        <button onClick={() => increment()}>Increment</button>
        <p>count : {count}</p>
      </div>
    </div>
  );
};
```

## ❖주의 사항

- 상태 변경은 set을 이용해 상태 객체를 리턴하도록 해야 함

```
set((state) => ({ count: state.count + 1 }));
```

# 3. todolist-app-router 예제에 zustand 적용

❖준비된 예제를 기반으로 zustand 적용

- todolist-app-router-zustand-1-시작
  - react-router, props drilling 하도록 작성된 예제
  - 모든 상태, 상태 변경 기능은 AppContainer가 관리함

❖ zustand 설치 : npm install zustand

❖src/stores/useTodoStore.ts 작성

```
import { create } from "zustand";
import { produce } from "immer";

export type TodoItemType = { id: number; todo: string; desc: string; done: boolean; };

//상태 데이터와 액션의 타입 선언
type TodoStateType = {
  todoList: TodoItemType[];
};
type TodoActionType = {
  addTodo: ({ todo, desc }: { todo: string; desc: string }) => void;
  deleteTodo: ({ id }: { id: number }) => void;
  toggleDone: ({ id }: { id: number }) => void;
  updateTodo: ({ id, todo, desc, done }: { id: number; todo: string; desc: string; done: boolean }) => void;
};
```

# 3. todolist-app-router 예제에 zustand 적용

## ❖src/stores/useTodoStore.ts 작성(이어서)

```
const useTodoStore = create<TodoStateType & TodoActionType>()(
  (set) => ({
    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 },
    ],
    addTodo: ({ todo, desc }) => {
      set((state) => {
        const newTodoList = produce(state.todoList, (draft) => {
          draft.push({ id: new Date().getTime(), todo, desc, done: false });
        });
        return { todoList: newTodoList };
      });
    },
    deleteTodo: ({ id }) => {
      set((state) => {
        const newTodoList = state.todoList.filter((item) => item.id !== id);
        return { todoList: newTodoList };
      });
    },
```

# 3. todolist-app-router 예제에 zustand 적용

## ❖src/stores/useTodoStore.ts 작성(이어서)

```
    toggleDone: ({ id }) => {
      set((state) => {
        const index = state.todoList.findIndex((item) => item.id === id);
        const newTodoList = produce(state.todoList, (draft: TodoItemType[]) => {
          draft[index].done = !draft[index].done;
        });
        return { todoList: newTodoList };
      });
    },
    updateTodo: ({ id, todo, desc, done }) => {
      set((state) => {
        const index = state.todoList.findIndex((item) => item.id === id);
        const newTodoList = produce(state.todoList, (draft: TodoItemType[]) => {
          draft[index] = { id, todo, desc, done };
        });
        return { todoList: newTodoList };
      });
    },
  })
);

export default useTodoStore;
```

# 3. todolist-app-router 예제에 zustand 적용

❖src/AppContainer.tsx 삭제

❖src/App.tsx 변경

- Props Drilling 코드를 제거함

```
.....(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;
```

# 3. todolist-app-router 예제에 zustand 적용

## ❖src/pages/TodoList.tsx 변경

- 속성과 속성 사용을 위한 타입 삭제
- use~ 훅을 이용해 store의 상태중 필요로 하는 것만 바인딩

```
import { Link } from "react-router-dom";
import TodoItem from "./TodoItem";
import useTodoStore from "../stores/useTodoStore";

const TodoList = () => {
  //store의 상태중 필요로 하는 것만 바인딩
  const todoList = useTodoStore((state) => state.todoList);
  const toggleDone = useTodoStore((state) => state.toggleDone);
  const deleteTodo = useTodoStore((state) => state.deleteTodo);

  ......(생략)

  return (
    ......(생략)
  );
};

export default TodoList;
```

# 3. todolist-app-router 예제에 zustand 적용

## ❖src/pages/AddTodo.tsx 변경

```
import { useState } from "react";
import { useNavigate } from "react-router";
import useTodoStore from "../stores/useTodoStore";

const AddTodo = () => {
  const addTodo = useTodoStore((state) => state.addTodo);

  const navigate = useNavigate();

  const [todo, setTodo] = useState<string>("");
  const [desc, setDesc] = useState<string>("");

  const addTodoHandler = () => {
    ......(생략)
  };

  return (
    ......(생략)
  );
};

export default AddTodo;
```

# 3. todolist-app-router 예제에 zustand 적용

## ❖src/pages/EditTodo.tsx 변경

```
import { useState } from "react";
import { useNavigate, useParams } from "react-router-dom";
import { useState } from "react";
import { useNavigate, useParams } from "react-router-dom";
import useTodoStore, { TodoItemType } from "../stores/useTodoStore";

type TodoParam = { id?: string };

const EditTodo = () => {
  const todoList= useTodoStore((state)=>state.todoList);
  const updateTodo = useTodoStore((state)=>state.updateTodo);
  ......(생략)

  const updateTodoHandler = () => {
    ......(생략)
  };

  return (
    ......(생략)
  );
};

export default EditTodo;
```

# 3. todolist-app-router 예제에 zustand 적용

## ❖src/main.tsx 변경

- AppContainer 대신에 App 컴포넌트를 사용하도록 변경

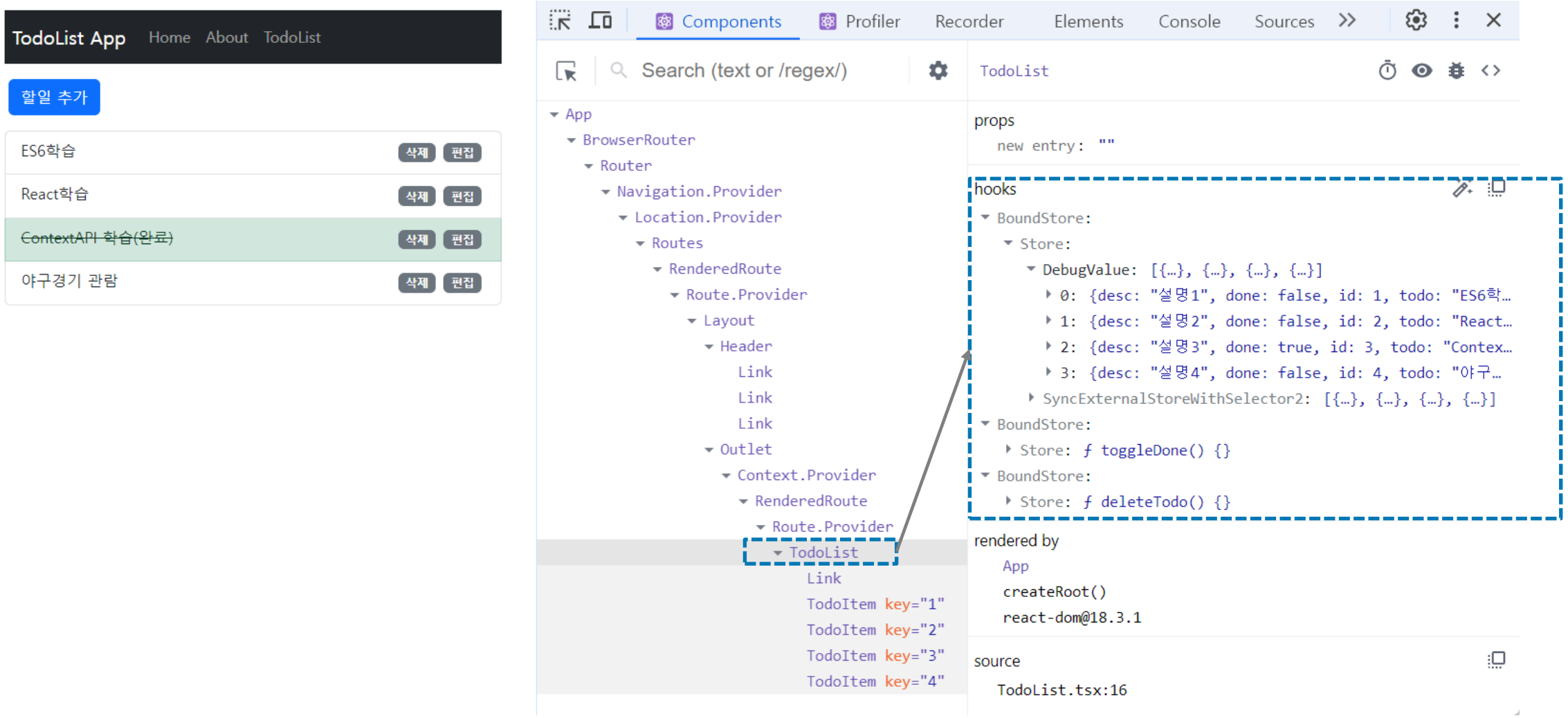
```
import React from "react";
import ReactDOM from "react-dom/client";
import "bootstrap/dist/css/bootstrap.css";
import App from "./App";
import "./index.css";

ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

## ❖ 추가로 사용법을 설명할 때 언급되었던 다음 코드도 적용해보자

- useCountStore.ts
- Home.tsx

# 3. todolist-app-router 예제에 zustand 적용

❖실행 결과



# 4. 몇가지 middleware

## ❖zustand middleware

- 간단한 콘솔 로그를 남기는 미들웨어

```
import { StateCreator, StoreApi } from "zustand";

export declare type LoggerType<S> = (state: S) => S;

export const logger = <S>(stateCreater: StateCreator<S>) => {
  return (set: StoreApi<S>["setState"], get: StoreApi<S>["getState"], api: StoreApi<S>): S => {
    const state = stateCreater(
      (...args) => {
        set(...args);
        console.log("## 변경된 상태 : ", get());
      },
      get,
      api
    );
    return { ...state };
  };
};
```

```
//미들웨어 사용은 다음과 같이
const useTodoStore = create<TodoStateType & TodoActionType>()(
  logger((set) => ({  ……(생략)……}))
);
```

# 4. 몇가지 middleware

❖zustand에 포함된 유용한 미들웨어

- immer 미들웨어
  - 상태를 변경할 때 새로운 객체를 만들어서 리턴할 필요없이 상태를 직접 변경할 수 있도록 함
  - 내부적으로 immer 라이브러리 사용
- devtools 미들웨어
  - redux devtools를 이용할 수 있는 기능 제공
- persist 미들웨어
  - zustand 상태를 localStorage, indexedDB, AsyncStorage와 같은 저장소에 저장할 수 있는 기능 제공

- 여러 개의 미들웨어를 적용하고 싶다면 중첩시켜 사용

```
//미들웨어 사용은 다음과 같이
const useTodoStore = create<TodoStateType & TodoActionType>()(
  devtools(
    immer(
      (set) => ({  ......(생략)......})
    )
  )
);
```

# 4. 몇가지 middleware

❖immer 미들웨어 적용전과 적용 후 액션 메서드 비교

```
//immer 미들웨어 적용 전
addTodo: ({ todo, desc }) => {
  set((state) => {
    const newTodoList = produce(state.todoList, (draft) => {
      draft.push({ id: new Date().getTime(), todo, desc, done: false });
    });
    return { todoList: newTodoList };
  });
},

//immer 미들웨어 적용 후
//직접 상태를 변경하면 됨
addTodo: ({ todo, desc }) => {
  set((state) => {
    state.todoList.push({ id: new Date().getTime(), todo, desc, done: false });
  });
},
```

# 5. 비동기 처리

❖zustand를 사용할 때 비동기 처리는 어떻게?

- 액션에서 직접 처리. redux와 같이 복잡한 미들웨어를 사용할 필요 없음
- zustand의 액션에서는 async/await 지원

```
//전체 코드는 contacts-app-zustand 예제 참조
const useContactStore = create<ContactStateType>()(
  immer((set) => ({
    contacts: [],
    isLoading: false,
    status: "",
    searchContacts: async (name: string) => {
      set((state) => {
        state.isLoading = true;
        state.status = "pending";
      });
      const url = "http://localhost:3000/contacts_long/search/" + name;
      const response = await axios.get(url);
      set((state) => {
        state.isLoading = false;
        state.contacts = response.data;
        state.status = "completed";
      });
    },
  }))
);
```

# 6. Slice 패턴

## ❖Slice란?

- Zustand에서 스토어의 상태관리 기능를 작게 분할한 것
  - Slice는 Initializer 형태로 작성함
- 여러 슬라이스를 조합하여 하나의 스토어를 만들 수 있음

## ❖Slice를 사용하는 이유

- 다중 스토어 vs 단일 스토어+다중 슬라이스
- 서로 연관성을 가진 상태들이라면 스토어를 여러개 만드는 것보다 하나의 스토어에 슬라이스를 분리시키는 것이 바람직함

# 6. Slice 패턴

❖완성된 예제 검토

- todolist-app-router-zustand

❖Slice 만들기

- 각 Slice별 타입 선언
  - 상태, 액션 뿐만 아니라 기존 상태를 이용하는 다른 액션도 정의할 수 있음
- Slice 작성
  - StateCreator 타입으로 작성

```
// ** Mutator : 미들웨어
//T : 슬라이스 함수 내부로 전달되는 상태의 타입. 일반적으로 스토어의 전체 상태(필수)
//S : 이 슬라이스에서 정의할 상태의 타입, 생략하면 T와 동일값
//Mps : Mutator Parameters. mutator에 전달되는 mutator의 타입을 정의하는 매개변수. 생략하면 []
//Mos : Mutator Outputs. mutator가 적용된 후의 상태 타입을 정의하는 매개변수. 생략하면 []
//Mps, Mos에 지정하는 것은 여러 개의 Mutator를 지정할 수 있기 때문에 Array로 지정함
//Mis, Mos 예시 : [typeof mw1, typeof logger]

StateCreator<T, Mis, Mos, S>
```

- Slice들을 결합하여 Store 객체 생성

# 6. Slice 패턴

## ❖각 Slice 별 타입 선언 : src/stores/index.ts 검토

```
export type TodoSliceType = {
  todoList: TodoItemType[];
  addTodo: ({ todo, desc }: { todo: string; desc: string }) => void;
  deleteTodo: ({ id }: { id: number }) => void;
  toggleDone: ({ id }: { id: number }) => void;
  updateTodo: ({ id, todo, desc, done }: { id: number; todo: string; desc: string; done: boolean }) => void;
};

export type CountSliceType = {
  count: number;
  increment: () => void;
};

export type InitializeSliceType = {
  initialize: () => void;
  getAllStates: () => { count: number; todoList: TodoItemType[] };
};
```

# 6. Slice 패턴

## ❖Slice 작성

```
// src/stores/countSlice.ts 검토
export const countSlice: StateCreator<CountSliceType & TodoSliceType, [], [], CountSliceType> = (set) => ({
  count: 0,
  increment: () => {
    set((state) => {
      return { count: state.count + 1 };
    });
  },
});
// src/stores/initializeSlice.ts 검토. 1. count, todoList 상태 초기화, 2. 상태만 모아서 리턴
export const initializeSlice: StateCreator<TodoSliceType & CountSliceType, [], [], InitializeSliceType> = (set, get) => ({
  initialize: () => {
    set(() => {
      return { count: 0, todoList: [] };
    });
  },
  getAllStates: () => {
    return { count: get().count, todoList: get().todoList };
  },
});
// src/stores/todoSlice.ts는 완성된 예제를 직접 검토
```

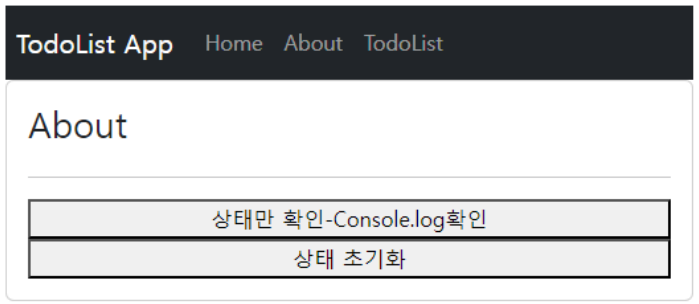
# 6. Slice 패턴

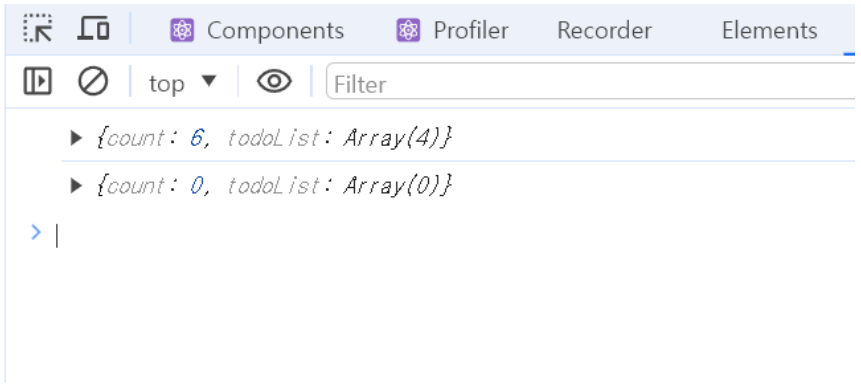
## ❖slice를 결합하여 store 생성

```
export const useBoundStore = create<TodoSliceType & CountSliceType & InitializeSliceType>()(
  devtools((...a) => ({
    ...countSlice(...a),
    ...todoSlice(...a),
    ...initializeSlice(...a),
  }))
);
```

## ❖실행 결과

- initializeSlice의 기능을 이용하는 예제 : src/pages/About.tsx
  - Home에서 count를 몇번 증가시킴
  - About으로 이동하여 상태 확인 -〉 상태 초기화 -〉 상태 확인

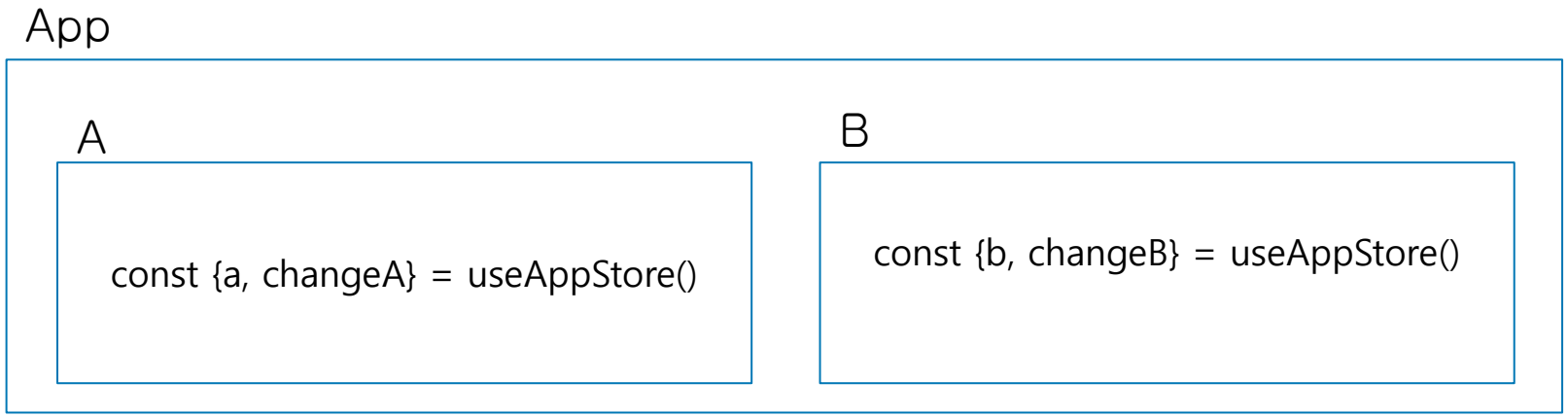


# 7. 사용시 주의사항

❖zustand를 잘못 사용하면 불필요한 렌더링이 발생

- 한 화면에 A, B 컴포넌트가 마운트되어 있다고 가정
- zustand가 관리하는 store의 상태는 a, b, changeA, changeB
  - A 컴포넌트 : a, changeA 사용,  B 컴포넌트 : b, changeB 사용
- 다음 그림과 같이 코드를 작성했다면?
  - 하나씩 바인딩하기 힘드니 구조분해할당으로 한번에 바인딩하면 편해보이긴 하지만 렌더링최적화를 할 수 없음.
  - 예) a가 변경되면 A,B 모두  re-render됨
  - 이유
    - useAppStore() 이 코드는 Store의 상태 전체를 컴포넌트에 바인딩 시도함.
    - 구조분해할당으로 필요한 값을 바인딩하지만 컴포넌트 수준에서는 전체 상태를 바인딩하는 것



# 7. 사용시 주의사항

- 문제 해결
  - 반드시 selector 함수를 이용해 하나씩 필요한 상태만 바인딩함
  - 다음 그림에서 상태 b만 변경된 경우 A 컴포넌트 re-render하지 않음

App

A

```
const a = useAppStore((state)=>state.a)
const changeA =
    useAppStore((state)=>state.changeA)
```

B

```
const b = useAppStore((state)=>state.b)
const changeB =
    useAppStore((state)=>state.changeB)
```

# 7. 사용시 주의사항

❖반드시 불변성을 가진 변경을 수행해야 함

- 새로운 객체를 생성하여 리턴하도록 작성해야 함
  - 상태 변경의 추적과 리액트 앱의 최적화를 위해 반드시 준수해야 함
- 불편하다면 immer 미들웨어를 사용할 것을 권장