20. useReducer 를 사용하여 상태 업데이트 로직 분리하기

이 프로젝트에서 사용된 코드는 다음 링크에서 확인 할 수 있습니다.

useReducer 이해하기

우리가 이전에 만든 사용자 리스트 기능에서의 주요 상태 업데이트 로직은 App 컴포넌트 내부에서 이루어졌었습니다. 상태를 업데이트 할 때에는 `useState` 를 사용해서 새로운 상태를 설정해주었는데요, 상태를 관리하게 될 때 `useState` 를 사용하는것 말고도 다른 방법이 있습니다. 바로, `useReducer` 를 사용하는건데요, 이 Hook 함수를 사용하면 컴포넌트의 상태 업데이트 로직을 컴포넌트에서 분리시킬 수 있습니다. 상태 업데이트 로직을 컴포넌트 바깥에 작성 할 수도 있고, 심지어 다른 파일에 작성 후 불러와서 사용 할 수도 있지요.
App 컴포넌트에서 `useReducer` 를 사용해보기전에 우리가 `useState` 를 처음 배울 때 만들었던 Counter.js 컴포넌트에서 `useReducer` 를 사용해보겠습니다.
이전에 작성한 코드를 다시 볼까요?

Counter.js

```
import React, { useState } from 'react';

function Counter() {
  const [number, setNumber] = useState(0);

  const onIncrease = () => {
    setNumber(prevNumber => prevNumber + 1);
  };

  const onDecrease = () => {
    setNumber(prevNumber => prevNumber - 1);
  };

  return (
    <div>
      <h1>{number}</h1>
      <button onClick={onIncrease}>+1</button>
      <button onClick={onDecrease}>-1</button>
    </div>
  );
}

export default Counter;
```

useReducer Hook 함수를 사용해보기전에 우선 reducer 가 무엇인지 알아보겠습니다. reducer 는 현재 상태와 액션 객체를 파라미터로 받아와서 새로운 상태를 반환해주는 함수입니다.

```
function reducer(state, action) {
  // 새로운 상태를 만드는 로직
  // const nextState = ...
  return nextState;
}
```

reducer 에서 반환하는 상태는 곧 컴포넌트가 지닐 새로운 상태가 됩니다.

여기서 action 은 업데이트를 위한 정보를 가지고 있습니다. 주로 type 값을 지닌 객체 형태로 사용하지만, 꼭 따라야 할 규칙은 따로 없습니다.
액션의 예시들을 확인해볼까요?

```
// 카운터에 1을 더하는 액션
{
  type: 'INCREMENT'
}
// 카운터에 1을 빼는 액션
{
  type: 'DECREMENT'
}
// input 값을 바꾸는 액션
{
  type: 'CHANGE_INPUT',
  key: 'email',
  value: 'tester@react.com'
}
// 새 할 일을 등록하는 액션
{
  type: 'ADD_TODO',
  todo: {
    id: 1,
    text: 'useReducer 배우기',
    done: false,
  }
}
```

보신 것 처럼 action 객체의 형태는 자유입니다. type 값을 대문자와 _ 로 구성하는 관습이 존재하기도 하지만, 꼭 따라야 할 필요는 없습니다.
자, 이제 reducer 를 배웠으니 useReducer 의 사용법을 알아봅시다. useReducer 의 사용법은 다음과 같습니다.

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

여기서 state 는 우리가 앞으로 컴포넌트에서 사용 할 수 있는 상태를 가르키게 되고, dispatch 는 액션을 발생시키는 함수라고 이해하시면 됩니다. 이 함수는 다음과 같이 사용합니다: dispatch({ type: 'INCREMENT' }).
그리고 useReducer 에 넣는 첫번째 파라미터는 reducer 함수이고, 두번째 파라미터는 초기 상태입니다.
그럼, Counter 컴포넌트를 만약에 useReducer 로 구현한다면 어떻게 바뀌는지 알아볼까요?
다음 코드를 한번 따라서 작성해보세요.

Counter.js

```
import React, { useReducer } from 'react';

function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

function Counter() {
  const [number, dispatch] = useReducer(reducer, 0);

  const onIncrease = () => {
    dispatch({ type: 'INCREMENT' });
  };

  const onDecrease = () => {
    dispatch({ type: 'DECREMENT' });
  };

  return (
    <div>
      <h1>{number}</h1>
      <button onClick={onIncrease}>+1</button>
      <button onClick={onDecrease}>-1</button>
    </div>
  );
}

export default Counter;
```

잘 작동하는지도 확인해볼까요?

index.js 파일을 열어서 App 대신 Counter 를 렌더링해보세요.

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import Counter from './Counter';

ReactDOM.render(<Counter />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

그리고 카운터가 잘 작동하는지 확인해보세요.

잘 작동하는것을 확인했다면 App 컴포넌트를 렌더링하도록 다시 복구하세요.

App 컴포넌트를 useReducer 로 구현하기

이번에는, App 컴포넌트에 있던 상태 업데이트 로직들을 `useState` 가 아닌 `useReducer` 를 사용하여 구현해보겠습니다. 우선, App 에서 사용 할 초기 상태를 컴포넌트 바깥으로 분리해주고, App 내부의 로직들을 모두 제거해주세요. 우리가 앞으로 차근차근 구현 할 것입니다.

App.js

```
import React, { useRef, useState, useMemo, useCallback } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

const initialState = {
  inputs: {
    username: '',
    email: ''
  },
  users: [
    {
      id: 1,
      username: 'velopert',
      email: 'public.velopert@gmail.com',
      active: true
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
      email: 'liz@example.com',
      active: false
    }
  ]
};

function App() {
  return (
    <>
      <CreateUser />
      <UserList users={[]} />
      <div>활성사용자 수 : 0</div>
```

```
    </>
  );
}

export default App;
```

먼저, `reducer` 함수의 틀만 만들어주고, `useReducer` 를 컴포넌트에서 사용해보세요.

App.js

```
import React, { useRef, useReducer, useMemo, useCallback } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

const initialState = {
  inputs: {
    username: '',
    email: ''
  },
  users: [
    {
      id: 1,
      username: 'velopert',
      email: 'public.velopert@gmail.com',
      active: true
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
      email: 'liz@example.com',
      active: false
    }
  ]
};

function reducer(state, action) {
  return state;
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      <CreateUser />
      <UserList users={[]} />
      <div>활성사용자 수 : 0</div>
    </>
```

```
  );
}

export default App;
```

그 다음엔, state 에서 필요한 값들을 비구조화 할당 문법을 사용하여 추출하여 각 컴포넌트에게 전달해주세요.

App.js

```
import React, { useRef, useReducer, useMemo, useCallback } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

const initialState = {
  inputs: {
    username: '',
    email: ''
  },
  users: [
    {
      id: 1,
      username: 'velopert',
      email: 'public.velopert@gmail.com',
      active: true
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
      email: 'liz@example.com',
      active: false
    }
  ]
};

function reducer(state, action) {
  return state;
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const { users } = state;
  const { username, email } = state.inputs;

  return (
    <>
      <CreateUser username={username} email={email} />
```

```
      <UserList users={users} />
      <div>활성사용자 수 : 0</div>
    </>
  );
}

export default App;
```

이제 onChange 부터 구현을 해줍시다.

App.js

```
import React, { useRef, useReducer, useMemo, useCallback } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

const initialState = {
  inputs: {
    username: '',
    email: ''
  },
  users: [
    {
      id: 1,
      username: 'velopert',
      email: 'public.velopert@gmail.com',
      active: true
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
      email: 'liz@example.com',
      active: false
    }
  ]
};

function reducer(state, action) {
  switch (action.type) {
    case 'CHANGE_INPUT':
      return {
        ...state,
        inputs: {
          ...state.inputs,
          [action.name]: action.value
        }
      };
```

```
    default:
      return state;
  }
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const { users } = state;
  const { username, email } = state.inputs;

  const onChange = useCallback(e => {
    const { name, value } = e.target;
    dispatch({
      type: 'CHANGE_INPUT',
      name,
      value
    });
  }, []);

  return (
    <>
      <CreateUser username={username} email={email} onChange={onChange} />
      <UserList users={users} />
      <div>활성사용자 수 : 0</div>
    </>
  );
}

export default App;
```

`CHANGE_INPUT` 이라는 액션 객체를 사용하여 `inputs` 상태를 업데이트해주었습니다. `reducer` 함수에서 새로운 상태를 만들 때에는 불변성을 지켜주어야 하기 때문에 위 형태와 같이 spread 연산자를 사용해주었습니다. 이번엔 onCreate 를 만들어볼까요?

App.js

```
import React, { useRef, useReducer, useMemo, useCallback } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

const initialState = {
  inputs: {
    username: '',
    email: ''
  },
  users: [
    {
      id: 1,
      username: 'velopert',
      email: 'public.velopert@gmail.com',
      active: true
```

```
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
      email: 'liz@example.com',
      active: false
    }
  ]
};

function reducer(state, action) {
  switch (action.type) {
    case 'CHANGE_INPUT':
      return {
        ...state,
        inputs: {
          ...state.inputs,
          [action.name]: action.value
        }
      };
    case 'CREATE_USER':
      return {
        inputs: initialState.inputs,
        users: state.users.concat(action.user)
      };
    default:
      return state;
  }
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const nextId = useRef(4);

  const { users } = state;
  const { username, email } = state.inputs;

  const onChange = useCallback(e => {
    const { name, value } = e.target;
    dispatch({
      type: 'CHANGE_INPUT',
      name,
      value
    });
  }, []);

  const onCreate = useCallback(() => {
    dispatch({
      type: 'CREATE_USER',
      user: {
        id: nextId.current,
        username,
```

```
        email
      }
    });
    nextId.current += 1;
  }, [username, email]);

  return (
    <>
      <CreateUser
        username={username}
        email={email}
        onChange={onChange}
        onCreate={onCreate}
      />
      <UserList users={users} />
      <div>활성사용자 수 : 0</div>
    </>
  );
}

export default App;
```

코드 수정 후 새 데이터 등록이 잘 되는지 확인해주세요.

이제 `onToggle` 과 `onRemove` 도 구현해보겠습니다.

App.js

```
import React, { useRef, useReducer, useMemo, useCallback } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

const initialState = {
  inputs: {
    username: '',
    email: ''
  },
  users: [
    {
      id: 1,
      username: 'velopert',
      email: 'public.velopert@gmail.com',
      active: true
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
```

```
      email: 'liz@example.com',
      active: false
    }
  ]
};

function reducer(state, action) {
  switch (action.type) {
    case 'CHANGE_INPUT':
      return {
        ...state,
        inputs: {
          ...state.inputs,
          [action.name]: action.value
        }
      };
    case 'CREATE_USER':
      return {
        inputs: initialState.inputs,
        users: state.users.concat(action.user)
      };
    case 'TOGGLE_USER':
      return {
        ...state,
        users: state.users.map(user =>
          user.id === action.id ? { ...user, active: !user.active } : user
        )
      };
    case 'REMOVE_USER':
      return {
        ...state,
        users: state.users.filter(user => user.id !== action.id)
      };
    default:
      return state;
  }
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const nextId = useRef(4);

  const { users } = state;
  const { username, email } = state.inputs;

  const onChange = useCallback(e => {
    const { name, value } = e.target;
    dispatch({
      type: 'CHANGE_INPUT',
      name,
      value
    });
  }, []);

  const onCreate = useCallback(() => {
    dispatch({
      type: 'CREATE_USER',
      user: {
```

```
        id: nextId.current,
        username,
        email
      }
    });
    nextId.current += 1;
  }, [username, email]);

  const onToggle = useCallback(id => {
    dispatch({
      type: 'TOGGLE_USER',
      id
    });
  }, []);

  const onRemove = useCallback(id => {
    dispatch({
      type: 'REMOVE_USER',
      id
    });
  }, []);

  return (
    <>
      <CreateUser
        username={username}
        email={email}
        onChange={onChange}
        onCreate={onCreate}
      />
      <UserList users={users} onToggle={onToggle} onRemove={onRemove} />
      <div>활성사용자 수 : 0</div>
    </>
  );
}

export default App;
```

이제 마지막, 활성 사용자수 구하는것도 구현하겠습니다. 사실 이 부분은 바뀌는 코드가 없습니다.

App.js

```
import React, { useRef, useReducer, useMemo, useCallback } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

const initialState = {
  inputs: {
    username: '',
    email: ''
  },
```

```
  users: [
    {
      id: 1,
      username: 'velopert',
      email: 'public.velopert@gmail.com',
      active: true
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
      email: 'liz@example.com',
      active: false
    }
  ]
};

function reducer(state, action) {
  switch (action.type) {
    case 'CHANGE_INPUT':
      return {
        ...state,
        inputs: {
          ...state.inputs,
          [action.name]: action.value
        }
      };
    case 'CREATE_USER':
      return {
        inputs: initialState.inputs,
        users: state.users.concat(action.user)
      };
    case 'TOGGLE_USER':
      return {
        ...state,
        users: state.users.map(user =>
          user.id === action.id ? { ...user, active: !user.active } : user
        )
      };
    case 'REMOVE_USER':
      return {
        ...state,
        users: state.users.filter(user => user.id !== action.id)
      };
    default:
      return state;
  }
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const nextId = useRef(4);
```

```
  const { users } = state;
  const { username, email } = state.inputs;

  const onChange = useCallback(e => {
    const { name, value } = e.target;
    dispatch({
      type: 'CHANGE_INPUT',
      name,
      value
    });
  }, []);

  const onCreate = useCallback(() => {
    dispatch({
      type: 'CREATE_USER',
      user: {
        id: nextId.current,
        username,
        email
      }
    });
    nextId.current += 1;
  }, [username, email]);

  const onToggle = useCallback(id => {
    dispatch({
      type: 'TOGGLE_USER',
      id
    });
  }, []);

  const onRemove = useCallback(id => {
    dispatch({
      type: 'REMOVE_USER',
      id
    });
  }, []);

  const count = useMemo(() => countActiveUsers(users), [users]);
  return (
    <>
      <CreateUser
        username={username}
        email={email}
        onChange={onChange}
        onCreate={onCreate}
      />
      <UserList users={users} onToggle={onToggle} onRemove={onRemove} />
      <div>활성사용자 수 : {count}</div>
    </>
  );
}

export default App;
```

이전에 구현한 기능들이 이번에도 잘 작동하는지 확인해보세요.

이제 모든 기능들이 `useReducer` 를 사용하여 구현됐습니다.

useReducer vs useState - 뭐 쓸까?

자 이제 궁금해지는 점이 한가지 있을 것입니다. 어떨 때 `useReducer` 를 쓰고 어떨 때 `useState` 를 써야 할까요? 일단, 여기에 있어서는 정해진 답은 없습니다. 상황에 따라 불편할때도 있고 편할 때도 있습니다.
예를 들어서 컴포넌트에서 관리하는 값이 딱 하나고, 그 값이 단순한 숫자, 문자열 또는 boolean 값이라면 확실히 `useState` 로 관리하는게 편할 것입니다.

```
const [value, setValue] = useState(true);
```

하지만, 만약에 컴포넌트에서 관리하는 값이 여러개가 되어서 상태의 구조가 복잡해진다면 `useReducer` 로 관리하는 것이 편해질 수도 있습니다.
이에 대한 결정은, 앞으로 여러분들이 `useState`, `useReducer` 를 자주 사용해보시고 맘에드는 방식을 선택하세요.
저의 경우에는 setter 를 한 함수에서 여러번 사용해야 하는 일이 발생한다면

```
setUsers(users => users.concat(user));
setInputs({
  username: '',
  email: ''
});
```

그 때부터 `useReducer` 를 쓸까? 에 대한 고민을 시작합니다. `useReducer` 를 썼을때 편해질 것 같으면 `useReducer` 를 쓰고, 딱히 그럴것같지 않으면 `useState` 를 유지하면 되지요.