useReducer를 사용하여 상태 업데이트 로직 분리하기

**useReducer 이해하기**

이전에 만든 사용자 리스트 기능에서의 주요 상태 업데이트 로직은 App 컴포넌트 내부에서 이루어졌었다. 상태를 업데이트 할때에는 useState를 사용해서 새로운 상태를 설정해주었는데, 또 다른 방법은 useReducer를 사용하는 것이다. 이 Hook 함수를 사용하면 컴포넌트의 상태 업데이트 로직을 컴포넌트에서 분리시킬 수 있다. 상태 업데이트 로직을 컴포넌트 바깥에 작성 할 수도 있고, 심지어 다른 파일에 작성 후 불러와서 사용할 수도 있다.

useReducer Hook함수를 사용해보기 전에 우선 reducer가 무엇인지 알아보자. reducer는 현재 상태와 액션 객체를 파라미터로 받아와서 새로운 상태를 반환해주는 함수이다.

function reducer(state, action) {

    // 새로운 상태를 만드는 로직

    // const nextState = ...

    return nextState;

  }

reducer에서 반환하는 상태는 곧 컴포넌트가 지닐 새로운 상태가 된다. 여기서 action은 업데이트를 위한 정보를 가지고 있다. 주로 type 값을 지닌 객체 형태로 사용하지만, 꼭 따라야 할 규칙은 따로 없다.

action의 예시)

// 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,

}

}

이와 같이 action 객체의 형태는 자유이다.

이제 useReducer의 사용법을 알아보자. useReducer의 사용법은 다음과 같다

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

여기서 state는 우리가 앞으로 컴포넌트에서 사용할 수 있는 상태를 가리키게 되고, dispatch는 액션을 발생시키는 함수라고 이해하면 된다. 이 함수는 다음과 같이 사용한다 dispatch({type:’INCREMENT’})

그리고 useReducer에 넣는ㄴ 첫번째 파라미터는 reucer함수이고, 두 번째 파라미터는 초기 상태이다. 그럼 기존의 Counter.js를 보자.

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;

이를 useReducer로 다시 구현해보자.

import React,{useReducer} from 'react';

function reucer(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;

index.js파일을 열어서 app 대신 counter로 렌더링해서 결과를 확인하고 다시 app으로 렌더링하자

**App 컴포넌트를 useReducer로 구현하기**

이번에는 App컴포넌트에 있던 상태 업데이트 로직들을 useReducer를 사용하여 구현해보자.

import React, {useRef,useReducer,useMemo,useCallback}from 'react';

import UserList from './UserList';

import CreateUser from './CreateUser';

function countActiveUsers(users){

  console.log('활성 사용자 수를 세는 중...');

  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':

      //CHANGE\_INPUT 이라는 액션 객체를 사용하여 inputs 상태를 업데이트 해줌

      //reducer 함수에서 불변성을 지키기 위해서 spread 연산자를 사용

      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>활성 사용자 수: {count}</div>

    </>

  );

}

export default App;

**useReducer vs useState**

상황에 따라 불편할 때도 있고 편할 때도 있기 때문에 정해진 답은 없다.

예를 들어서 컴포넌트에서 관리하는 값이 딱 하나고, 그 값이 단순한 숫자, 문자열 또는 boolean 값이라면 확실히 useState로 관리하는게 편할 것이다.

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

하지만, 컴포넌트에서 관리하는 값이 여러 개가 되어서 상태의 구조가 복잡해진다면 useReducer로 관리하는 것이 편할 수도 있다.

커스텀 Hooks 만들기

컴포넌트를 만들다보면, 반복되는 로직이 자주 발생한다. 이를 커스텀 hooks를 만들어서 쉽게 재사용하는 방법을 알아보자

src디렉터리에 hooks라는 디렉터리를 만들고 그 안에 useInputs.js파일을 만들자

다음과 같이 구현하자

useInput.js

import { useState, useCallback } from 'react';

function useInputs(initialForm) {

  const [form, setForm] = useState(initialForm);

  // change

  const onChange = useCallback(e => {

    const { name, value } = e.target;

    setForm(form => ({ ...form, [name]: value }));

  }, []);

  const reset = useCallback(() => setForm(initialForm), [initialForm]);

  return [form, onChange, reset];

}

export default useInputs;

이제 우리가 만든 useInputs Hook을 App.js에서 사용하자. 그러기 위해서는 먼저 useReducer쪽에서 사용하는 inputs를 없애고 이에 관련된 작업을 useInputs로 대체해주어야 한다. 새로운 항목을 추가할 때 input값을 초기화해야하므로 데이터 등록 후 reset()을 호출한다.

App.js

import React, { useRef, useReducer, useMemo, useCallback } from 'react';

import UserList from './UserList';

import CreateUser from './CreateUser';

import useInputs from './hooks/useInputs';

function countActiveUsers(users) {

  console.log('활성 사용자 수를 세는중...');

  return users.filter(user => user.active).length;

}

const initialState = {

  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 'CREATE\_USER':

      return {

        users: state.users.concat(action.user)

      };

    case 'TOGGLE\_USER':

      return {

        users: state.users.map(user =>

          user.id === action.id ? { ...user, active: !user.active } : user

        )

      };

    case 'REMOVE\_USER':

      return {

        users: state.users.filter(user => user.id !== action.id)

      };

    default:

      return state;

  }

}

function App() {

  const [{ username, email }, onChange, reset] = useInputs({

    username: '',

    email: ''

  });

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

  const nextId = useRef(4);

  const { users } = state;

  const onCreate = useCallback(() => {

    dispatch({

      type: 'CREATE\_USER',

      user: {

        id: nextId.current,

        username,

        email

      }

    });

    reset();

    nextId.current += 1;

  }, [username, email, reset]);

  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;

\*\*숙제 useinputs 커스텀 Hook을 한번 useReducer를 사용해서 구현해보세요.

Context API를 사용한 전역 값 관리

현재 만들고 있는 프로젝트를 보면, App컴포넌트에서 onToggle, onRemove가 구현이 되어있고 이 함수들은 UserList 컴포넌트를 거쳐서 각 User 컴포넌트들에게 전달이 된다.

여기서 UserList 컴포넌트의 경우에는 onToggle과 onRemove를 전달하기 위하여 중간 다리역할만 하고 있다. UserList에서는 해당 함수들을 직접 사용하는 일도 없다.

지금과 같이 특정 함수를 특정 컴포넌트를 거쳐서 원하는 컴포넌트에게 전달하는 작업은 리액트로 개발을 하다보면 자주 발생할 수 있는 작업이다. 위와 같이 컴포넌트 한 개 정도를 거쳐서 전달하는건 사실 그렇게 큰 불편함도 없지만, 만약 3~4개 이상의 컴포넌트를 거쳐서 전달을 해야하는 일이 발생하게 된다면 이는 매우 번거로울 것이다. 그럴 땐, 리액트의 Context API와 이전 섹션에서 배웠던 dispatch를 함께 사용하면 이러한 복잡한 구조를 해결할 수 있다.

리액트의 Context API를 사용하면, 프로젝트 안에서 전역적으로 사용할 수 있는 값을 관리할 수 있다. 여기서 상태가 아닌 값이라고 언급했는데, 이 값은 꼭 상태를 가리키지 않아도 된다. 이 값은 함수일 수도 있고, 어떤 외부 라이브러리 인스턴스일 수도 있고 심지어 DOM일 수도 있다.

우선 ContextAPI를 사용해서 새로운 context를 만다는 방법을 알아보자.

Context를 만들 땐 다음과 같이 React.createContext()라는 함수를 사용한다

const USerDispatch=React.createContext(null);

createContext의 파라미터에는 Context의 기본값을 설정할 수 있다. 여기서 설정하는 값은 Context를 쓸 때 값을 따로 지정하지 않을 경우 사용되는 기본 값이다. Context를 만들면, Context안에 Provider라는 컴포넌트가 들어있는데 이 컴포넌트를 통하여 Context의 값을 정할 수 있다. 이 컴포넌트를 사용할 때, value라는 값을 설정하면 된다.

<UserDispatch.Provider value={dispatch}>…</UserDispatch.Privder>

이렇게 설정해주고 나면 Provider에 의하여 감싸진 컴포넌트 중 어디서든지 우리가 Context의 값을 다른 곳에서 바로 조회해서 사용할 수 있다.

App 컴포넌트에서 Context를 만들고 사용하고 내보내는 작업을 진행하자

App.js