**1장. 리액트 입문 3/4: Hook**

**<<16. useEffect를 사용하여 마운트/언마운트/업데이트 시 할 작업 설정하기>>**

\* **useEffect Hook**을 사용하여 컴포넌트가 마운트 됐을 때(처음 나타났을 때), 언마운트 됐을 때(사라질 때), 그리고 업데이트 될 때(특정 props가 바뀔 때) 특정 작업을 처리하는 방법에 대해 공부하겠음

**\* 마운트/언마운트**

- UserList.js

import React, { useEffect } from 'react';

function User({ user, onRemove, onToggle }) {

useEffect(() => { //useEffect 사용

console.log('컴포넌트가 화면에 나타남');

return () => {

console.log('컴포넌트가 화면에서 사라짐');

};

}, []);

return (

<div>

<b

style={{

cursor: 'pointer',

color: user.active ? 'green' : 'black'

}}

onClick={() => onToggle(user.id)}

>

{user.username}

</b>

&nbsp;

<span>({user.email})</span>

<button onClick={() => onRemove(user.id)}>삭제</button>

</div>

);

}

function UserList({ users, onRemove, onToggle }) {

return (

<div>

{users.map(user => (

<User

user={user}

key={user.id}

onRemove={onRemove}

onToggle={onToggle}

/>

))}

</div>

);

}

export default UserList;

- useEffect를 사용할 때에는 첫번째 파라미터에 함수, 두번째 파라미터에 의존값이 들어있는 배열(deps)를 넣음. 만약 deps 배열을 비우게 된다면, 컴포넌트가 처음 나타날 때에만 useEffect에 등록한 함수가 호출됨

- useEffect에서는 함수를 반환할 수 있는데 이를 cleanup함수라고 부름. cleanup함수는 useEffect에 대한 뒷정리를 해준다고 이해하면 되는데, deps가 비어있는 경우에는 컴포넌트가 사라질 때 cleanup함수가 호출됨

텍스트이(가) 표시된 사진

자동 생성된 설명

- 주로 마운트 시에 하는 작업들

ㄴ props로 받은 값을 컴포넌트의 로컬 상태로 설정

ㄴ 외부 API 요청(REST API 등)

ㄴ 라이브러리 사용(D3, Video.js 등)

ㄴ setInterval을 통한 반복 작업 혹은 setTimeout을 통한 작업 예약

- 언마운트 시에 하는 작업들

ㄴ setInterval, setTimeout을 사용하여 등록한 작업들 clear하기(clearInterval,clearTimeout)

ㄴ 라이브러리 인스턴스 제거

**\* deps에 특정 값 넣기**

- deps에 특정 값을 넣게 되면 컴포넌트가 처음 마운트 될 때 호출이 되고, 지정한 값이 바뀔 때에도 호출됨. 그리고 deps안에 특정 값이 있다면 언마운트시에도 호출되고, 값이 바꾸기 직전에도 호출됨

- UseList.js

import React, { useEffect } from 'react';

function User({ user, onRemove, onToggle }) {

useEffect(() => {

console.log('user 값이 설정됨');

console.log(user);

return () => {

console.log('user 가 바뀌기 전..');

console.log(user);

};

}, [user]);

return (

<div>

<b

style={{

cursor: 'pointer',

color: user.active ? 'green' : 'black'

}}

onClick={() => onToggle(user.id)}

>

{user.username}

</b>

&nbsp;

<span>({user.email})</span>

<button onClick={() => onRemove(user.id)}>삭제</button>

</div>

);

}

function UserList({ users, onRemove, onToggle }) {

return (

<div>

{users.map(user => (

<User

user={user}

key={user.id}

onRemove={onRemove}

onToggle={onToggle}

/>

))}

</div>

);

}

export default UserList;

텍스트이(가) 표시된 사진

자동 생성된 설명

- useEffect 안에서 사용하는 상태나, props가 있다면, useEffect의 deps에 넣어주어야 함. 그렇게 하는 게 규칙임.

- 만약 useEffect 안에서 사용하는 상태나 props를 deps에 넣지 않게 된다면 useEffect에 등록한 함수가 실행될 때 최신 props / 상태를 가리키지 않게 됨

**\* deps 파라미터를 생략하기**

- deps 파라미터를 생략하면 컴포넌트가 리렌더링 될 때마다 호출됨

- UseList.js

import React, { useEffect } from 'react';

function User({ user, onRemove, onToggle }) {

useEffect(() => {

console.log(user);

});

return (

<div>

<b

style={{

cursor: 'pointer',

color: user.active ? 'green' : 'black'

}}

onClick={() => onToggle(user.id)}

>

{user.username}

</b>

&nbsp;

<span>({user.email})</span>

<button onClick={() => onRemove(user.id)}>삭제</button>

</div>

);

}

function UserList({ users, onRemove, onToggle }) {

return (

<div>

{users.map(user => (

<User

user={user}

key={user.id}

onRemove={onRemove}

onToggle={onToggle}

/>

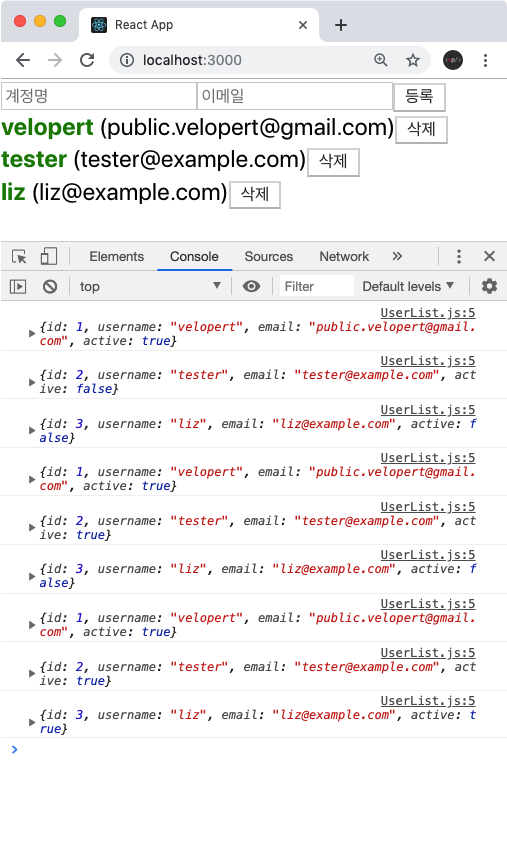
))}

</div>

);

}

export default UserList;



- 리액트 컴포넌트는 기본적으로 부모 컴포넌트가 리렌더링 되면 자식 컴포넌트 또한 리렌더링됨. (바뀐 내용이 없다 할지라도)

- 실제 DOM에 변화가 반영되는 것은 바뀐 내용이 있는 컴포넌트에만 해당됨. 하지만 Virtual DOM에는 모든 걸 다 렌더링하고 있음

- 나중에는 컴포넌트를 최적화 하는 과정에서 기존의 내용을 그대로 사용하면서 Virtual DOM에 렌더링 하는 리소스를 아낄 수도 있음

**<<17. useMemo를 사용하여 연산한 값 재사용하기>>**

**\* useMemo:** 성능 최적화를 위하여 연산된 값을 useMemo Hook을 이용하여 재사용 하는 방법을 알아보자

\* App컴포넌트에서 countActiveUsers라는 함수를 만들어서 active값이 true인 사용자의 수를 세어서 화면에 렌더링 하자

- App.js

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

import UserList from './UserList';

import CreateUser from './CreateUser';

function countActiveUsers(users) { //countActiveUsers 함수 만듦

console.log('활성 사용자 수를 세는중...');

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

}

function App() {

const [inputs, setInputs] = useState({

username: '',

email: ''

});

const { username, email } = inputs;

const onChange = e => {

const { name, value } = e.target;

setInputs({

...inputs,

[name]: value

});

};

const [users, setUsers] = useState([

{

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

}

]);

const nextId = useRef(4);

const onCreate = () => {

const user = {

id: nextId.current,

username,

email

};

setUsers(users.concat(user));

setInputs({

username: '',

email: ''

});

nextId.current += 1;

};

const onRemove = id => {

// user.id 가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

// = user.id 가 id 인 것을 제거함

setUsers(users.filter(user => user.id !== id));

};

const onToggle = id => {

setUsers(

users.map(user =>

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

)

);

};

const count = countActiveUsers(users);

return (

<>

<CreateUser

username={username}

email={email}

onChange={onChange}

onCreate={onCreate}

/>

<UserList users={users} onRemove={onRemove} onToggle={onToggle} />

<div>활성사용자 수 : {count}</div>

</>

);

}

export default App;

텍스트이(가) 표시된 사진

자동 생성된 설명

- (발생하는 성능적 문제) input값을 바꿀 때에도 countActiveUsers함수가 호출됨

텍스트이(가) 표시된 사진

자동 생성된 설명

- 활성 사용자 수를 세는 건 users에 변화가 있을 때만 세야 되는데, input값이 바뀔 때에도 컴포넌트가 리렌더링 되므로 이렇게 불필요할 때에도 호출하여서 자원이 낭비되고 있음

- 이러한 상황에서는 useMemo Hook함수를 사용하면 최적화를 할 수 있음

- Memo는 “memorized”를 의미하는데 이는 이전에 계산한 값을 재사용한다는 의미

- App.js

import React, { useRef, useState, useMemo } from 'react'; //useMomo 불러오기

import UserList from './UserList';

import CreateUser from './CreateUser';

function countActiveUsers(users) {

console.log('활성 사용자 수를 세는중...');

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

}

function App() {

const [inputs, setInputs] = useState({

username: '',

email: ''

});

const { username, email } = inputs;

const onChange = e => {

const { name, value } = e.target;

setInputs({

...inputs,

[name]: value

});

};

const [users, setUsers] = useState([

{

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

}

]);

const nextId = useRef(4);

const onCreate = () => {

const user = {

id: nextId.current,

username,

email

};

setUsers(users.concat(user));

setInputs({

username: '',

email: ''

});

nextId.current += 1;

};

const onRemove = id => {

// user.id 가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

// = user.id 가 id 인 것을 제거함

setUsers(users.filter(user => user.id !== id));

};

const onToggle = id => {

setUsers(

users.map(user =>

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

)

);

};

const count = useMemo(() => countActiveUsers(users), [users]); //useMemo사용

return (

<>

<CreateUser

username={username}

email={email}

onChange={onChange}

onCreate={onCreate}

/>

<UserList users={users} onRemove={onRemove} onToggle={onToggle} />

<div>활성사용자 수 : {count}</div>

</>

);

}

export default App;

텍스트이(가) 표시된 사진

자동 생성된 설명

- **useMemo**의 첫번째 파라미터에는 어떻게 연산할지 정의하는 함수를 넣어주면 되고 두번째 파라미터에는 deps배열을 넣어주면 되는데 이 배열 안에 넣은 내용이 바뀌면 우리가 등록한 함수를 호출해서 값을 연산해주고 만약 내용이 바뀌지 않았다면 연산한 값을 재사용 하게 됨

**<<18. useCallback을 사용하여 함수 재사용하기>>**

**\* useCallback:** UseMemo와 비슷한 Hook임

**\* useMemo:** 특정 결과값을 재사용할 때 사용

**\* useCallback:** 특정 함수를 새로 만들지 않고 재사용하고 싶을 때 사용

- App.js 일부(onCreate, onRemove, onToggle 함수)

const onCreate = () => {

const user = {

id: nextId.current,

username,

email

};

setUsers(users.concat(user));

setInputs({

username: '',

email: ''

});

nextId.current += 1;

};

const onRemove = id => {

// user.id 가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

// = user.id 가 id 인 것을 제거함

setUsers(users.filter(user => user.id !== id));

};

const onToggle = id => {

setUsers(

users.map(user =>

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

)

);

};

- 해당 함수들은 컴포넌트가 리렌더링 될 때마다 새로 만들어짐. 함수를 선언하는 것 자체는 사실 메모리도 CPU도 리소스를 많이 차지하는 작업은 아니라 그 자체만으로 큰 부하가 생길 일은 없지만 한 번 만든 함수를 필요한 때만 새로 만들고 재사용하는 것은 중요함

- 그 이유는 나중에 컴포넌트에서 props가 바뀌지 않았으면 Virtual DOM에 새로 렌더링 하는 것 조차 하지 않고 컴포넌트의 결과물을 재사용 하는 최적화 작업을 할 건데 이 작업을 하려면 함수를 재사용 하는 것이 필수임

- 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;

}

function App() {

const [inputs, setInputs] = useState({

username: '',

email: ''

});

const { username, email } = inputs;

const onChange = useCallback( //useCallback 사용

e => {

const { name, value } = e.target;

setInputs({

...inputs,

[name]: value

});

},

[inputs]

);

const [users, setUsers] = useState([

{

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

}

]);

const nextId = useRef(4);

const onCreate = useCallback(() => { //useCallback 사용

const user = {

id: nextId.current,

username,

email

};

setUsers(users.concat(user)); //useCallback 사용

setInputs({

username: '',

email: ''

});

nextId.current += 1;

}, [users, username, email]);

const onRemove = useCallback( //UseCallback 사용

id => {

// user.id 가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

// = user.id 가 id 인 것을 제거함

setUsers(users.filter(user => user.id !== id));

},

[users]

);

const onToggle = useCallback( //useCallback 사용

id => {

setUsers(

users.map(user =>

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

)

);

},

[users]

);

const count = useMemo(() => countActiveUsers(users), [users]);

return (

<>

<CreateUser

username={username}

email={email}

onChange={onChange}

onCreate={onCreate}

/>

<UserList users={users} onRemove={onRemove} onToggle={onToggle} />

<div>활성사용자 수 : {count}</div>

</>

);

}

export default App;

- (주의할 점) 함수 안에서 사용하는 상태 혹은 props가 있다면 꼭 deps 배열 안에 포함시켜야 함. 만약 deps 배열 안에 함수에서 사용하는 값을 넣지 않게 된다면, 함수 내에서 해당 값들을 참조할 때 가장 최신 값을 참조할 것이라고 보장할 수 없음. Props로 받아온 함수가 있다면, 이 또한 deps에 넣어주어야 함

- useCallback은 useMemo를 기반으로 만들어짐. 다만 함수를 위해서 사용할 때 더욱 편하게 해준 것뿐임. 이런 식으로도 표현할 수 있음.

const onToggle = useMemo(

() => () => {

/\* ... \*/

},

[users]

);

- useCallback을 사용함으로써 바로 이뤄낼 수 있는 눈에 띄는 최적화는 없음. 컴포넌트 렌더링 최적화 작업을 해주어야만 성능이 최적화됨.

**<<19. React.memo를 사용한 컴포넌트 리렌더링 방지>>**

**\* React.memo:** 컴포넌트의 props가 바뀌지 않았다면 리렌더링을 방지하여 컴포넌트의 리렌더링 성능 최적화를 해줄 수 있는 함수. 컴포넌트에서 리렌더링이 필요한 상황에서만 리렌더링을 하도록 설정해줄 수 있음

- CreateUser.js

import React from 'react';

const CreateUser = ({ username, email, onChange, onCreate }) => {

return (

<div>

<input

name="username"

placeholder="계정명"

onChange={onChange}

value={username}

/>

<input

name="email"

placeholder="이메일"

onChange={onChange}

value={email}

/>

<button onClick={onCreate}>등록</button>

</div>

);

};

export default React.memo(CreateUser); //React.memo사용

- UserList.js, User컴포넌트

import React from 'react';

const User = React.memo(function User({ user, onRemove, onToggle }) {

return (

<div>

<b

style={{

cursor: 'pointer',

color: user.active ? 'green' : 'black'

}}

onClick={() => onToggle(user.id)}

>

{user.username}

</b>

&nbsp;

<span>({user.email})</span>

<button onClick={() => onRemove(user.id)}>삭제</button>

</div>

);

});

function UserList({ users, onRemove, onToggle }) {

return (

<div>

{users.map(user => (

<User

user={user}

key={user.id}

onRemove={onRemove}

onToggle={onToggle}

/>

))}

</div>

);

}

export default React.memo(UserList);

텍스트이(가) 표시된 사진

자동 생성된 설명

- User중 하나라도 수정하면 모든 User들이 리렌더링되고, CreateUser도 리렌더링됨

- 그 이유는 users배열이 바뀔 때마다 onCreate도 새로 만들어지고 onToggle, onRemove도 새로 만들어지기 때문

const onCreate = useCallback(() => {

const user = {

id: nextId.current,

username,

email

};

setUsers(users.concat(user));

setInputs({

username: '',

email: ''

});

nextId.current += 1;

}, [users, username, email]);

const onRemove = useCallback(

id => {

// user.id 가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

// = user.id 가 id 인 것을 제거함

setUsers(users.filter(user => user.id !== id));

},

[users]

);

const onToggle = useCallback(

id => {

setUsers(

users.map(user =>

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

)

);

},

[users]

);

- deps에 users가 들어있기 때문에 배열이 바뀔 때마다 함수가 새로 만들어지는 건 당연함

- 이걸 최적화 하고 싶다면: deps에서 users를 지우고 함수들에서 현재 useState로 관리하는 users를 참조하지 않게 하는 것임. => 함수형 업데이트

- 함수형 업데이트를 하게 되면 setUsers에 등록하는 콜백 함수의 파라미터에서 최신 users를 참조할 수 있기 때문에 deps에 users를 넣지 않아도 됨

- 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;

}

function App() {

const [inputs, setInputs] = useState({

username: '',

email: ''

});

const { username, email } = inputs;

const onChange = useCallback(e => {

const { name, value } = e.target;

setInputs(inputs => ({

...inputs,

[name]: value

}));

}, []);

const [users, setUsers] = useState([

{

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

}

]);

const nextId = useRef(4);

const onCreate = useCallback(() => {

const user = {

id: nextId.current,

username,

email

};

setUsers(users => users.concat(user));

setInputs({

username: '',

email: ''

});

nextId.current += 1;

}, [username, email]);

const onRemove = useCallback(id => {

// user.id 가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

// = user.id 가 id 인 것을 제거함

setUsers(users => users.filter(user => user.id !== id));

}, []);

const onToggle = useCallback(id => {

setUsers(users =>

users.map(user =>

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

)

);

}, []);

const count = useMemo(() => countActiveUsers(users), [users]);

return (

<>

<CreateUser

username={username}

email={email}

onChange={onChange}

onCreate={onCreate}

/>

<UserList users={users} onRemove={onRemove} onToggle={onToggle} />

<div>활성사용자 수 : {count}</div>

</>

);

}

export default App;

- 리액트 개발을 할 때 useCallback, useMemo, React.memo는 컴포넌트의 성능을 실제로 개선할 수 있는 상황에서만 사용하셈

- 예를 들어 User컴포넌트에 b와 button에 onClick으로 설정해준 함수들은 해당 함수들을 useCallback으로 재사용한다고 해서 리렌더링을 막을 수 있는 것은 아니므로, 굳이 그렇게 할 필요 없음

- 추가적으로 렌더링 최적화 하지 않을 컴포넌트에 React.memo를 사용하는 것은 불필요한 props비교만 하는 것이기 때문에 실제로 렌더링을 방지할 수 있는 상황이 있는 경우에만 사용하길 바람

-추가적으로 React.memo에서 두번째 파라미터에 propsAreEqual이라는 함수를 사용하여 특정 값들만 비교하는 것도 가능함

export default React.memo(

UserList,

(prevProps, nextProps) => prevProps.users === nextProps.users

);

- 하지만 잘못 사용하면 버그가 발생하기 쉬움. 예를 들어 함수형 업데이트로 전환을 안 했는데 이렇게 users만 비교를 하게 된다면 onToggle과 onRemove에서 최신 users배열을 참조하지 않으므로 심각한 오류가 발생할 수 있음

**<<20. useReducer를 사용하여 상태 업데이트 로직 분리하기>>**

**\* useReducer 이해하기**

- 이전에 만든 사용자 리스트 기능에서의 주요 상태 업데이트 로직은 App 컴포넌트 내부에서 이루어졌음. 상태를 업데이트 할 때 useState를 사용해서 새로운 상태를 설정해주었는데 상태를 관리하게 될 때 useState를 사용하는 것 말고도 다른 방법이 있음.

- 바로 **useReducer Hook**함수를 사용하는 건데 사용하면 컴포넌트의 상태 업데이트 로직을 컴포넌트에서 분리시킬 수 있음. 상태 업데이트 로직을 컴포넌트 바깥에 작성할 수도 있고 심지어 다른 파일에 작성 후 불러와서 사용할 수도 있음

- 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;

**\* reducer**는 현재 상태와 액체 객체를 파라미터로 받아와서 새로운 상태를 반환해주는 함수

function reducer(state, action) {

// 새로운 상태를 만드는 로직

// const nextState = ...

return nextState;

}

\* reducer에서 반환하는 상태는 곧 컴포넌트가 지닐 새로운 상태가 됨

\* action은 업데이트를 위한 정보를 가지고 있음. 주로 type값을 지닌 객체 형태로 사용하지만, 꼭 따라야 할 규칙은 따로 없음

// 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,

}

}

- action객체의 형태는 자유임. type값을 대문자와 \_로 구성하는 관습이 존재하기도 하지만 꼭 따라야 할 필요는 없음

**\* useReducer**

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

- state는 우리가 앞으로 컴포넌트에서 사용할 수 있는 상태를 가리키게 되고, dispatch는 액션ㅇ르 발생시키는 함수라고 이해하면 됨. Dispatch({type: ‘INCREMENT’})

- userRenducer에 넣는 첫번째 파라미터는 reducer함수이고 두번째 파라미터는 초기 상태임

- 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;

- 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();

**\* App 컴포넌트를 useReducer로 구현하기**

- APP컴포넌트에 있던 상태 업데이트 로직들을 useState가 아닌 useReducer를 사용하여 구현.

- 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;

**\* useReducer vs useState**

- 컴포넌트에서 관리하는 값이 딱 하나고 그 값이 단순한 숫자, 문자열 또는 boolean값이라면 확실히 useState로 관리하는 게 편할 것

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

- 하지만 만약에 컴포넌트에서 관리하는 값이 여러 개가 되어서 상태의 구조가 복잡해진다면 useReducer로 관리하는 것이 편해질 수도 있음

**<<21. 커스텀 Hooks 만들기>>**

**\* 커스텀 Hooks:** 반복되는 로직을 쉽게 재사용

- 커스텀 Hooks를 만들 때에는 use~라는 키워드로 시작하는 파일을 만들고 그 안에 함수를 작성

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

- 그 안에서 useState, useEffect, useReducer, useCallback등 Hooks를 사용하여 원하는 기능을 구현해주고 컴포넌트에서 사용하고 싶은 값들을 반환

- useInputs.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;

\* useInputs Hook을 App.js에서 사용하기 위해서는 먼저 useReducer쪽에서 사용하는 inputs를 없애고 이에 관련된 작업을 useInputs를 대체해주어야 함. 새로운 항목을 추가할 때 input값을 초기화 해야 하므로 데이터 등록 후 reset()을 호출

- 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;