useMemo를 사용하여 연산한 값 재사용하기

성능 최적화를 위하여 연산된 값을 useMemo라는 Hook을 사용하여 재사용하는 방법을 알아보도록 하자. **App 컴포넌트에서 다음과 같이 countActiveUsers라는 함수를 만들어서, active 값이 true인 사용자의 수를 세어서 화면에 렌더링 해보자.**

App.js를 다음과 같이 수정해보자. 또한 UserList.js의 useEffect 부분은 console을 보기 편하게 잠시 주석처리 해두도록 하자.

countActiveUsers 함수에서 콘솔에 메시지를 출력하도록 한 이유는, 이 함수가 호출될 때 마다 우리가 알 수 있게 하기 위함이다.

import React, {useRef,useState} 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;

}

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,user]);

  setInputs({

    username:'',

    email:''

  });

  nextId.current+=1;

};

const onRemove = id => {

  //user.id가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

  //=user.id가 id인 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>활성 사용자 수 : {count}</div>

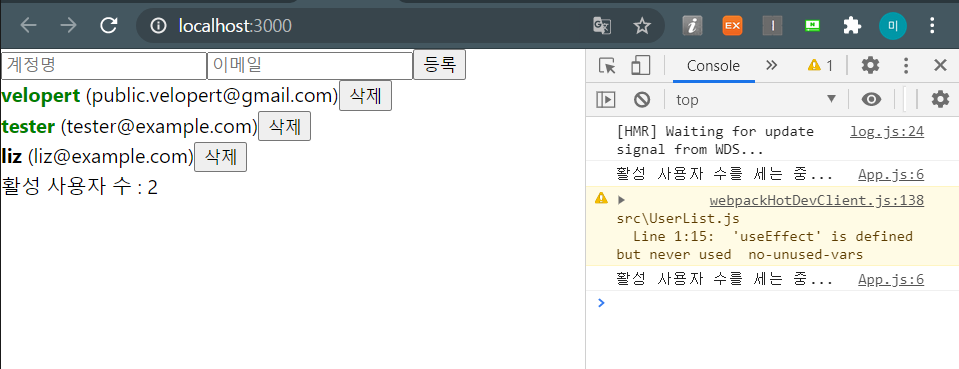
    </>

  );

}

export default App;

구현을 마치면 다음과 같은 결과가 나타난다. 다른 계정명을 눌러서 초록색으로 만들면 활성 사용자 수를 업데이트할 수 있다.



그런데 여기서 성능적 문제가 한가지 있다. 바로 input의 값을 바꿀때에도 countActiveUsers 함수가 호출된다는 것이다. 활성 사용자 수를 세는건, users에 변화가 있을때만 세야되는건데, input 값이 바뀔 때에도 컴포넌트가 리렌더링 되므로 불필요할 때에도 호출되어 자원이 낭비된다. 이러한 상황에는 **useMemo라는 Hook 함수를 사용하면 성능을 최적화 할 수 있다.**

Memo는 memoized를 의미하는데 이는 이전에 계산한 값을 재사용한다는 의미를 가지고 있다.

App.js를 다음과 같이 수정해보자

import React, {useRef,useState,useMemo} 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;

}

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,user]);

  setInputs({

    username:'',

    email:''

  });

  nextId.current+=1;

};

const onRemove = id => {

  //user.id가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

  //=user.id가 id인 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]);

  return (

    <>

    <CreateUser

      username={username}

      email={email}

      onChange={onChange}

      onCreate={onCreate}/>

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

   <div>활성 사용자 수 : {count}</div>

    </>

  );

}

export default App;

useMemo의 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, 만약에 내용이 바뀌지 않았다면 이전에 연산한 값을 재사용하게 된다.

이제 input인 username 과 email의 값을 바꾸어도 countActiveUsers 함수가 호출되지 않는다.

useMemo는 의존성이 변경되었을 때에만 메모이제이션된 값만 다시 계산 할 것입니다. 이 최적화는 모든 렌더링 시의 고비용 계산을 방지하게 해 줍니다.

useMemo로 전달된 함수는 렌더링 중에 실행된다는 것을 기억하세요. 통상적으로 렌더링 중에는 하지 않는 것을 이 함수 내에서 하지 마세요. 예를 들어, 사이드 이펙트(side effects)는 useEffect에서 하는 일이지 useMemo에서 하는 일이 아닙니다.

배열이 없는 경우 매 렌더링 때마다 새 값을 계산하게 될 것입니다.

**useMemo는 성능 최적화를 위해 사용할 수는 있지만 의미상으로 보장이 있다고 생각하지는 마세요.** 가까운 미래에 React에서는, 이전 메모이제이션된 값들의 일부를 “잊어버리고” 다음 렌더링 시에 그것들을 재계산하는 방향을 택할지도 모르겠습니다. 예를 들면, 오프스크린 컴포넌트의 메모리를 해제하는 등이 있을 수 있습니다. useMemo를 사용하지 않고도 동작할 수 있도록 코드를 작성하고 그것을 추가하여 성능을 최적화하세요.

(https://ko.reactjs.org/docs/hooks-reference.html#usememo)

useCallback을 사용하여 함수 재사용하기

useCallback은 useMemo와 비슷한 Hook이다. useMemo는 특정 결과값을 재사용할 때 사용하는 반면, useCallback은 특정 함수를 새로 만들지 않고 재사용하고 싶을 때 사용한다.

App.js에서 구현했었던 onCreate, onRemove, onToggle 함수를 확인해보자. 이 함수들은 컴포넌트가 리렌더링 될 때 마다 새로 만들어진다. 함수를 선언하는 것 자체는 사실 메모리도, CPU도 리소스를 많이 차지하는 작업은 아니기 때문에 함수를 새로 선언한다고 해서 그 자체만으로 큰 문제가 생길일은 없지만, 한번 만든 함수를 필요할 때만 새로 만들고 재사용하는 것은 여전히 중요하다.

다음과 같이 App.js를 수정해보자.

import React, {useRef,useState,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;

}

function App() {

  const [inputs,setInputs]=useState({

    username: '',

    email:''

  });

  const {username,email}=inputs;

  const onChange = useCallback(

    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(()=>{

  const user={

    id:nextId.current,

    username,

    email

  };

  setUsers([...users,user]);

  setInputs({

    username:'',

    email:''

  });

  nextId.current+=1;

},[users,username,email]);

const onRemove = useCallback(

  id=> {

  //user.id가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

  //=user.id가 id인 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]);

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

  return (

    <>

    <CreateUser

      username={username}

      email={email}

      onChange={onChange}

      onCreate={onCreate}/>

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

   <div>활성 사용자 수 : {count}</div>

    </>

  );

}

export default App;

함수 안에서 사용하는 상태 혹은 props가 있다면 꼭, deps 배열안에 포함시켜야 된다. 만약에 deps 배열 안에 함수에서 사용하는 값을 넣지 않게 된다면, 함수 내에서 해당 값들을 참조할 때 가장 최신 값을 참조할 것이라고 보장할 수 없다. props로 받아온 함수가 있다면, 이 또한 deps에 넣어주어야 한다. 사실 useCallback은 useMemo를 기반으로 만들어졌다. 다만, 함수를 위해서 사용할 때 더욱 편하게 해준 것 뿐이다. 다음과 같이 표현 할 수 있다.

const onToggle = useMemo(

  ()=>()=>{

    /\* \*/

  },[users]

);

useCallback을 사용함으로써, 바로 이뤄낼 수 있는 눈에 띄는 최적화는 없다. 컴포넌트 렌더링 최적화 작업을 해주어야만 성능이 최적화 된다. 그 전에, 어떤 컴포넌트가 렌더링 되고 있는지 확인하기 위해서 React DevTools라는 도구를 깔면 된다.

(<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=ko>)

[메모이제이션된](https://ko.wikipedia.org/wiki/%EB%A9%94%EB%AA%A8%EC%9D%B4%EC%A0%9C%EC%9D%B4%EC%85%98) 콜백을 반환합니다.

인라인 콜백과 그것의 의존성 값의 배열을 전달하세요. useCallback은 콜백의 메모이제이션된 버전을 반환할 것입니다. 그 메모이제이션된 버전은 콜백의 의존성이 변경되었을 때에만 변경됩니다. 이것은, 불필요한 렌더링을 방지하기 위해 (예로 shouldComponentUpdate를 사용하여) 참조의 동일성에 의존적인 최적화된 자식 컴포넌트에 콜백을 전달할 때 유용합니다.

useCallback(fn, deps)은 useMemo(() => fn, deps)와 같습니다.

useCallback과 useMemo는 다음 링크를 조금 더 읽어보면 이해가 된다.

<https://leehwarang.github.io/2020/05/02/useMemo&useCallback.html>

React.memo를 사용한 컴포넌트 리렌더링 방지

**컴포넌트의 props가 바뀌지 않았다면, 리렌더링을 방지하여 컴포넌트의 리렌더링 성능 최적화를 해줄 수 있는 React.memo 라는 함수**에 대해서 알아보자.

이 함수를 사용한다면, 컴포넌트에서 리렌더링이 필요한 상황에서만 리렌더링을 하도록 설정할 수 있다.

다음과 같이 CreateUser.js를 수정하자

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);

다음과 같이 UserList.js를 수정하자

import React,{useEffect} from 'react';

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

    /\*useEffect(()=>{

        console.log('user 값이 설정됨');

        console.log(user);

        return()=>{

            console.log('user가 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 React.memo(UserList);

수정한 다음, input을 수정할 때 하단의 UserList가 리렌더링이 되지 않음을 확인할 수 있다.

그런데 User중 하나라도 수정하면 모든 User들이 리렌더링 되고, CreateUser도 리렌더링이 된다. 그 이유는 users 배열이 바뀔 때 마다 onCreate도 새로 만들어지고, onToggle, on Remove도 새로 만들어지기 때문이다. deps에 users가 들어있기 대문에 배열이 바뀔 때마다 함수가 새로 만들어지는건 당연하다. 이를 최적화하려면 deps에서 users를 지우고, 함수들에서 현재 useState로 관리하는 users를 참조하지 않게 하는 것이다. **이는 함수형 업데이트를 하면 된다.**

함수형 업데이트를 하게 되면, setUsers에 등록하는 콜백함수의 파라미터에서 최신 users를 참조할 수 있기 때문에 deps에 users를 넣지 않아도 된다.

App.js를 다음과 같이 수정하자

import React, {useRef,useState,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;

}

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.id가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

  //=user.id가 id인 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>활성 사용자 수 : {count}</div>

    </>

  );

}

export default App;

이렇게 해주면, 특정 항목을 수정하게 될 때 해당 항목만 리렌더링 된다. 따라서 최적화가 끝난 것이다.

리액트 개발을 할 때, useCallback, useMemo,React.memo는 컴포넌트의 성능을 실제로 개선할 수 있는 상황에서만 하자. 예를 들어, User컴포넌트에 b와 button에 onClick으로 설정해준 함수들은, 해당 함수들을 useCallback으로 재사용한다고 해서 리렌더링을 막을 수 있는 것이 아니므로 굳이 그렇게 하지 않아도 된다. 추가적으로 렌더링 최적화 하지 않을 컴포넌트에 React.memo를 사용하는 것은 불필요한 props 비교만 하는 것이기 때문에 실제로 렌더링을 방지할 수 있는 상황이 있는 경우에만 사용하길 바란다. 추가적으로 React.memo에서 두번째 파라미터에 propsAreEqual이라는 함수를 사용하여 특정 값들만 비교 하는 것도 가능하다. 이걸 잘못사용한다면 오히려 의도치 않은 버그들이 발생하기 쉽기 때문에 잘 알아보고 사용하자

**최적화는 나중에 필요할 때 다시 공부하자. 지금은 useMemo useCallback React.memo가 어떤 역할을 하는지, 어떤 차이점이 있는지 이정도만 이해하고 넘겨도 될 것 같다.**

ㅇㄹ