Web Programming

React programming

useMemo

useMemo

- useMemo에서 Memo라는 말은 Memoization을 뜻한다.
- 동일한 값을 리턴하는 함수를 반복적으로 호출할 경우 맨 처음 값을 메모리에 저장해서 필요할 때마다 또다시 계산하지 않고 메모리에서 꺼내서 재사용을 하는 기법이다.
- 간단히 말해서 자주 필요한 값을 맨 처음 계산할 때 캐싱해 놓아서 그 값이 필요할 때마다 다시 계산을 하는 것이 아니라 캐시에서 꺼내서 사용하는 것이다.

```
const value = useMemo(() => {
            return calculate();
}. [item])
```

- useMemo는 두 개의 인자를 받는다. 첫 번째는 콜백 함수, 두 번째는 배열을 받는다.
- 첫 번째 인자인 콜백 함수는 메모이제이션을 해줄 값을 계산해서 리턴해 주는 함수이다.
- 즉 이 콜백 함수가 리턴하는 값이 바로 useMemo가 리턴하는 값이 된다.
- 두 번째 인자인 배열은 의존성 배열이라고 부르는데 useMemo는 배열 안의 요소의 값이 업데이트될 때만 콜백 함수를 다시 호출해서 메모이제이션 된 값을 업데이트해서 다시 메모이제이션을 한다.
- 빈 배열을 넘겨주면 컴포넌트가 처음 마운트 되었을 때만 값을 계산하고 이후에는 항상 메모이제이션된 값을 꺼내와서 사용한다.

useMemo

useMemo

```
function calculate(){
        return 10
}

function Component(){
        const value = calculate();
        return <div>{value}</div>
}
```

컴포넌트가 렌더링되는 조건

1. state가 변경되었을 때
2. props가 변경되었을 때
3. 부모 컴포넌트가 렌더링 되었을 때
4. forceUpdate()가 실행되었을 때

- 함수형 컴포넌트는 함수다. 함수형 컴포넌트가 렌더링이 된다는 것은 그 함수가 호출된다는 것이고 그때 함수 내부에 정의해 놓은 내부 변수가 초기화된다.
- 컴포넌트는 state 와 props의 변화로 인해 수많은 렌더링을 거친다. 컴포넌트가 렌더링이 될 때마다 위 코드의 value 변수는 초기화가 된다. 렌더링이 될 때마다 calculate() 함수는 반복적으로 호출이 될 텐데 이는 효율적이지 못하다. 만약 calculate() 함수가 무거운 일을 하는 함수라면 굉장히 비효율적일 것이다. 왜냐하면 같은 값을 value에 할당하는 무의미한 작업을 계속해서 반복하기 때문이다.

useMemo

useMemo

- 이런 문제점을 useMemo를 사용해서 memoization을 하면 이를 간단하게 해결해 줄 수 있다.
- useMemo는 처음에 계산된 결과값을 메모리에 저장해서 컴포넌트가 반복적으로 렌더링 되어도 계속 calculate() 함수를 호출하지 않고 이전에 이미 계산된 결과값을 메모리에서 꺼내 와서 재사용 할 수 있게 해준다.
- 하지만 useMemo도 무분별하게 남용하면 오히려 성능면에서 좋지 않다.
- useMemo를 사용한다는 것은 값을 재활용하기 위해서 따로 메모리를 소비해서 저장하는 것인데 그렇기 때문에 불필요한 값까지 전부 메모이제이션을 하면 성능에 좋지 않기 때문이다. 그러니 필요할 때만 적절하게 사용하자

useMemo

useMemo

```
import { useEffect, useState } from "react";

export default function Practice() {
  const [number, setNumber] = useState(0);
  const [isKorea, setIsKorea] = useState(true);

  // string 할당
  // const location = isKorea ? "한국" : "외국";

  // 📌Object 할당
  const location = {
    country: isKorea ? "한국" : "외국",
  };
const location = useMemo(() => {
  return { country: isKorea ? "한국" : "외국" };
}, [isKorea]);
```

```
useEffect(() => {
  console.log("useEffect 호출");
}, [location]);

return (
  <div>
    <h2>하루에 몇끼 먹어요?</h2>
    <input
      type="number"
      value={number}
      onChange={(e) => setNumber(e.target.value)}
    />
    <hr />
    <h2>어느 나라에 있어요?</h2>
    {/* 📌 location.country 할당 */}
    <p>나라:{location.country}</p>
    <button onClick={() => setIsKorea(!isKorea)}>비행기 타자</button>
    <p></p>
  </div>
);
}
```

The 'location' object makes the dependencies of useEffect Hook (at line 17) change on every render.

useCallback

useCallback

- React에서 컴포넌트가 재랜더링될 때, 그 컴포넌트 내에 정의된 모든 함수들은 새로 생성된다.
- 특히, 부모 컴포넌트에서 정의된 함수가 자식 컴포넌트로 props로 전달될 때, 부모 컴포넌트가 재렌더링되면 해당 함수는 새로운 참조를 가지게 된다.
- 그러면 자식 컴포넌트는 props가 변경되었다고 인식하여 불필요한 재랜더링을 하게 된다.
- 이러한 문제는 자바스크립트의 함수가 클로저(closure)라는 특성 때문에 발생하는데, 클로저는 함수가 생성된 시점의 환경을 '기억' 한다.
- 이 문제를 해결하기 위해 react에서는 useCallback이라는 Hook을 사용한다.
- useCallback을 사용하면 useCallback 첫번째 인자로 전달된 함수를 메모이제이션(저장)하고, 컴포넌트가 재렌더링될 때마다 새로 생성되지 않고, 의존성 배열에 명시된 값이 변경될 때만 함수가 새로 생성된다.

useCallback

useCallback

- useMemo는 자주 쓰이는 값을 메모이제이션 즉 캐싱해준다. 그리고 그 값이 필요할 때 다시 계산을 하는 것이 아닌 useMemo를 통해 캐싱을 한 값을 메모리에서 꺼내 와서 재사용한다.
- useMemo에 인자로 콜백함수를 넣어주면 함수가 리턴하는 값을 메모이제이션 하는 것이다.
- useCallback은 인자로 전달한 콜백 함수 그 자체를 메모이제이션 하는 것이다.
- 즉, 이전에 생성된 함수를 재사용하여 동일한 함수 인스턴스를 반환한다.

```
useCallback(() => {
        return value;
}. [item])
```

```
useCallback(callback, dependencies)
```

- callback : 메모이제이션할 함수. 이 함수는 컴포넌트가 렌더링 될 때마다 새로 생성되는 것이 아니라, 동일한 함수 인스턴스를 유지한다.
- dependencies(optional) : 배열 형태로 전달되며, 해당 값들이 변결될 때에만 새로운 콜백을 생성한다. 이 배열을 통해 useCallback이 의존성을 감지하고, 의존성이 변경되지 않으면 이전에 생성된 콜백을 재사용한다. 의존선 배열을 명시하지 않으면 매 렌더링마다 동일한 콜백이 재사용된다.

useCallback

- useCallback은 React.memo와 함께 자식 컴포넌트의 불필요한 렌더링을 최적화할 수 있는데, 침실, 주방, 욕실의 불을 켜고 끄는 예제를 한번 살펴보겠다

```
// App.js
import React, { useState } from "react";
import SmartHome from "./components/SmartHome";

const App = () => {
  return (
    <div style={{ position: "absolute", top: "50%", left: "50%" }}>
      <SmartHome />
    </div>
  );
};

export default App;
```

```
// ./components/Light.jsx

import React from "react";

function Light({ room, on, toggle }) {
  console.log({ room, on });
  return (
    <div>
      <button onClick={toggle}>
        {room}
        {on ? "💡" : "■"}
      </button>
    </div>
  );
}

export default Light;
```

useCallback

useCallback

```
// ./components/SmartHome.jsx

import React, { useState } from "react";
import Light from "./Light";

function SmartHome() {
  const [masterOn, setMasterOn] = useState(false);
  const [kitchenOn, setKitchenOn] = useState(false);
  const [bathOn, setBathOn] = useState(false);

  const toggleMaster = () => {
    setMasterOn(!masterOn);
  };
  const toggleKitchen = () => {
    setKitchenOn(!kitchenOn);
  };
```

```
  const toggleBath = () => {
    setBathOn(!bathOn);
  };

  return (
    <div>
      <Light room="침실" on={masterOn} toggle={toggleMaster}></Light>
      <Light room="주방" on={kitchenOn} toggle={toggleKitchen}></Light>
      <Light room="욕실" on={bathOn} toggle={toggleBath}></Light>
    </div>
  );
}

export default SmartHome;
```

✓ 컴포넌트들을 생성하고 침실 토글 버튼을 클릭하고 콘솔 창을 확인해보니 침실뿐만 아니라 다른 모든 방에 대한 Light 컴포넌트 함수가 호출되는 것이 확인되었다.

useCallback

- 조명을 켜고 끄는 방에 대해서만 Light 컴포넌트를 호출되게 하기 위해 Light 컴포넌트 부분에 React.memo를 사용해보겠다.

```
// ./components/Light.jsx

import React from "react";

function Light({ room, on, toggle }) {
  console.log({ room, on });
  return (
    <div>
      <button onClick={toggle}>
        {room}
        {on ? "🔆" : "■"}
      </button>
    </div>
  );
}

export default React.memo(Light);
```

✓ React.memo를 사용했음에도 불구하고 props가 변경되지 않았다고 생각된 침실이나 욕실 또한 컴포넌트가 렌더링 되었다. 그 이유는 SmartHome 컴포넌트가 렌더링 될 때마다 toggleMaster(), toggleKitchen(), toggleBath() 함수의 참조값이 바뀌어버리기 때문에 props의 변경을 감지하고 Lgith컴포넌트가 렌더링 되기 때문이다.

useCallback

useCallback

- 따라서 해당 toggle 함수들에게 useCallback 함수를 적용하면!

```
// ./components/SmartHome.jsx

import React, { useState, useCallback } from "react";
import Light from "./Light";

function SmartHome() {
  const [masterOn, setMasterOn] = useState(false);
  const [kitchenOn, setKitchenOn] = useState(false);
  const [bathOn, setBathOn] = useState(false);

  const toggleMaster = useCallback(() => {
    setMasterOn(!masterOn);
  }, [masterOn]);
  const toggleKitchen = useCallback(() => {
    setKitchenOn(!kitchenOn);
  }, [kitchenOn]);
```

```
  const toggleBath = useCallback(() => {
    setBathOn(!bathOn);
  }, [bathOn]);

  return (
    <div>
      <Light room="침실" on={masterOn} toggle={toggleMaster}></Light>
      <Light room="주방" on={kitchenOn} toggle={toggleKitchen}></Light>
      <Light room="욕실" on={bathOn} toggle={toggleBath}></Light>
    </div>
  );
}

export default SmartHome;
```

✓ 욕실에 대한 조명을 켜보면 욕실에 대한 Light 컴포넌트만 리 렌더링 된 것을 확인할 수 있다.

useTransition

useTransition

- useTransition은 UI를 차단하지 않고 상태를 업데이트 할 수 있는 리액트 훅이다.
- 일반적으로 오래 걸리는 상태 업데이트(setState)가 존재할 경우, 해당 업데이트가 완료된 이후에 렌더링이 일어나기 때문에 그 시간만큼 렌더 트리가 '블락(Block)'된다. 이 때문에 유저는 아무런 동작을 할 수 없는 상태에 빠지게 되므로 UX에 좋지 않은 영향을 준다.
- useTransition은 컴포넌트 최상위 수준에서 호출되어 startTransition을 통해 우선순위가 낮은 상태 업데이트들을 transition이라고 표시해 리액트가 UI 렌더링시 우선순위에 따라 업데이트 할 수 있도록 한다. 이로써 렌더링이 오래 걸리는 컴포넌트의 블락을 피할수 있게 된다.
- transition으로 표시된 상태 업데이트(A라 호칭)는 다른 일반적인 상태 업데이트(B)가 호출될때 중단되고 B의 상태 업데이트가 완료된 다음 다시 A를 렌더링 시작한다. 이를 통해 특정 컴포넌트의 렌더링이 오래 걸리더라도 다른 우선순위 높은 상태의 변경을 통해 User Interaction을 블로킹하지 않고 자연스럽게 동작할 수 있도록 한다.

useTransition

useTransition

- 렌더링시간이 매우 오래 걸리는 컴포넌트가 있다고 가정하자. 버튼 클릭, 타이핑할 때 마다 그 컴포넌트를 렌더링해야한다면 이상하게 버튼 클릭, 타이핑 반응 속도도 느려진다.
- 사람들은 원래 클릭, 타이핑을 했을 때 0.3초 이상 반응이 없으면 불편함을 느끼기 때문에 개선방법을 알아보자.
- 당연히 그 컴포넌트 안의 html 개수를 줄이면 대부분 해결되지만 그렇지 않을 경우 useTransition 기능을 쓰면 된다.

```
const[isPending, startTransition] = useTransition();
```

- useTransition은 두개의 항목이 있는 배열을 반환한다.
- 1. isPending 플래그는 대기중인 transition이 있는지 알려준다.
- 2. startTransition 함수는 상태 업데이트(setState)를 transition으로 표시해주는 함수이다.

useTransition

useTransition

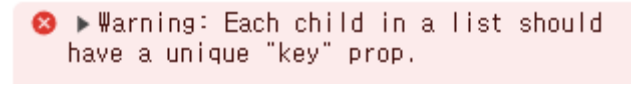
```
import {useState} from 'react'

let a = new Array(10000).fill(0)

function App(){
  let [name, setName] = useState('')

  return (
    <div>
      <input onChange={ (e)=>{ setName(e.target.value) }}/>
      {
        a.map(()=>{
          return <div>{name}</div>
        })
      }
    </div>
  )
}
```

- ✓ 데이터가 10000개 들어있는 array자료를 하나 만들고 그 갯수만큼 를 생성하라고 명령을 주고 유저가 타이핑할 수 있는 <input>도 만들어봤다.
- ✓ 유저가 <input>에 타이핑하면 그 글자를 1만개 안에 넣어줘야하는데 1만개 렌더링해주느라 <input>도 많은 지연시간이 발생한다. 타이핑한 결과가 바로바로 화면에 나타나지 않으니 상당히 답답하여 유저가 다 떠날것이다.
- ✓ "useTransition"을 사용하면 된다.



useTransition

useTransition

```
import {useState, useTransition} from 'react'

let a = new Array(10000).fill(0)

function App(){
  let [name, setName] = useState('')
  let [isPending, startTransition] = useTransition()

  return (
    <div>
      <input onChange={ (e)=>{
        startTransition(()=>{
          setName(e.target.value)
        })
      }}/>
      {
        isPending ? "로딩 중" :
        a.map(()=>{
          return <div>{name}</div>
        })
      }
    </div>
  )
}
```

- ✓ useTransition() 쓰면 그 자리에 [변수, 함수]가 남는다.
- ✓ 그 중 우측에 있는 startTransition() 함수로 state변경 함수 같은걸 묶으면 그걸 다른 코드들보다 나중에 처리해준다.
- ✓ 그래서 <input> 타이핑같이 즉각 반응해야하는걸 우선적으로 처리해줄 수 있다.
- ✓ 타이핑해보면 아까보다 반응속도가 훨씬 낫다.
- ✓ 물론 근본적인 성능개선이라기보단 특정코드의 실행 시점을 뒤로 옮겨주는 것일 뿐이다.
- ✓ html이 많으면 여러페이지로 쪼개는게 좋다.

useTransition

useTransition

- useDeferredValue()는 특정 값의 처리 우선순위를 낮추어 렉 현상을 줄여 사용자경험(UX)을 향상시키는 목적으로 사용된다.
- useTransition()과 차이점이라면 useTransition()은 useCallback()처럼 특정 함수를 마킹하여 처리 우선순위를 낮추지만 useDeferredValue()는 useMemo()처럼 특정 값을 마킹하여 우선순위를 낮추는 식으로 사용한다.
- useMemo(), useCallback()과 같이 앱의 성능을 최적화하는 용도로 사용되는 훅입니다. 남용해서 사용하는 경우 작업에 불필요한 오버헤드가 추가되므로 적절한 경우에만 사용하도록 합시다.
- useDeferredValue의 사용

```
const [state, setState] = useState('');
const deferredState = useDeferredValue(state);
```

useTransition

useTransition

- useDeferredValue()는 보통 useState() 훅과 함께 사용합니다. useState()로 생성된 state 값을 인자로 사용하여 상태 업데이트의 처리 우선순위를 낮추고 다른 작업을 동시적으로 처리하도록 허용한다.
- 앞의 코드처럼 useDeferredValue()로 마킹된 state는 상태 업데이트(setState())가 오래 걸리더라도 다른 작업을 지연 시키거나 렉 현상을 발생시키지 않게끔 다른 작업을 먼저 우선해서 처리하도록 합니다. deferred value라는 뜻 그대로, 값 변경이 일어날 때 다른 작업을 우선 처리하도록 해서 지연된 결과 값을 사용하게 되는 것이다.

useTransition

useTransition

```
import { useDeferredValue, useState } from "react";

export default function Home() {
  const [count1, setCount1] = useState(0);
  const [count2, setCount2] = useState(0);
  const [count3, setCount3] = useState(0);
  const [count4, setCount4] = useState(0);

  const deferredValue = useDeferredValue(count2);

  const onIncrease = () => {
    setCount1(count1 + 1);
    setCount2(count2 + 1);
    setCount3(count3 + 1);
    setCount4(count4 + 1);
  };

  console.log({ count1 });
  console.log({ count2 });
  console.log({ count3 });
  console.log({ count4 });
  console.log({ deferredValue });

  return <button onClick={onIncrease}>클릭</button>;
}
```

- ✓ 먼저 useDeferredValue는 상태 값 변화에 낮은 우선순위를 지정하기 위한 훅이다.
- ✓ 코드에서 count2의 값을 가장 낮은 우선순위로 상태를 변경하고 싶을 경우에는 useDeferredValue로 count2를 래핑하면 된다.
- ✓ onIncrease 함수를 실행하게 되면 count2의 값은 바뀌게 되었는데 deferredValue의 값은 다른 상태변화가 다 발생하고 가장 나중에 변경되게 된다.

use

use

- Promise를 정의하는 데 필요한 상용구 코드의 양을 줄여 데이터 가져오기 프로세스를 단순화하도록 설계된 솔루션이다.

```
// `use` inside of a React component or Hook...
const data = use(promise);

// ...roughly equates to `await` in an async function
const data = await promise;
```

- 다른 Hook과 달리 useHook은 조건, 블록, 루프 내부에서 호출할 수 있는 특별한 특성을 가지고 있다.
- use를 사용하면 promise을 인수로 전달하여 데이터를 비동기적으로 가져올 수 있다. 또한 if 문 내에서 "use"를 사용하여 특정 조건에 따라 조건부로 데이터를 일시 중지할 수 있다.

use

use

```
import { use } from "react";

function Note({id, shouldIncludeAuthor}) {
  // fetching data asynchronously
  const note = use(fetchNote(id));

  let byline = null;
  if (shouldIncludeAuthor) {
    // can be used inside if statements
    // Because `use` is inside a conditional block, we avoid blocking
    // unncessarily when `shouldIncludeAuthor` is false.
    const author = use(fetchNoteAuthor(note.authorId));
    byline = <h2>{author.displayName}</h2>;
  }

  return (
    <div>
      <h1>{note.title}</h1>
      {byline}
      <section>{note.body}</section>
    </div>
  );
}
```

use

useDebugValue

useDebugValue

- React DevTools에서 사용자 지정 hook에 label을 추가할 수 있는 React Hook이다. 이 hook은 아무것도 return 하지 않는다.

```
useDebugValue(value, format?)
```

- value
  - ➢ React DevTools에 보여주고 싶은 값으로, 어떤 타입이든 가능하다.
- optional format
  - ➢ formatting 함수이다. 컴포넌트를 검사할 때, React DevTools는 value를 매개변수로 받아 이 formatting 함수를 호출한다. 그리고 formatting된 value를 return 해준다. 만약 해당 함수를 작성하지 않은 경우에는 value 그 자체가 보여질 것이다.

useDebugValue

사용법

- Adding a label to a custom Hook
  - ➢ React DevTools에서 debug value를 읽기쉽게 하기 위해서, 커스텀 훅의 최상위 레벨에서 useDebugValue를 호출한다. 아래의 예시에서 만약 useDebugValue를 사용하지 않으면 OnlineStatus: "Online" 대신 OnlineStatus: true가 보여질 것이다.

```
import { useDebugValue } from 'react';

function useOnlineStatus() {
  // ...
  useDebugValue(isOnline ? 'Online' : 'Offline');
  // ...
}
```

  - ➢ 모든 커스텀 훅에 debug value를 추가할 필요는 없다. 검사하기 어렵고 복잡한 데이터 구조를 가진 경우에 가장 유용하다고 할 수 있다.

useDebugValue

사용법

- Deferring formatting of a debug value
  - useDebugValue의 두번째 매개변수로 formatting 함수를 넘길 수 있다. 컴포넌트를 검사할 때, React DevTools는 해당 함수를 호출하여 formatting 된 value를 반환해 보여준다.

```
useDebugValue(date, date => date.toDateString());
```

  - 이렇게 하면 컴포넌트가 실제로 검사되지 않는 한 잠재적으로 비용이 많이 드는 로직을 실행하지 않아도 된다.

useDebugValue

useDebugValue

- React Hooks 중 하나로 개발자 도구 Dev Tools 에서 컴포넌트의 상태를 디버깅할 때 사용한다.
- useDebugValue 는 두개의 인자를 받음
  - 첫번째 인자는 디버깅에 사용할 값(value)
  - 두번째 인자는 옵션 객체이다.
- useDebugValue 함수를 사용해서 count 값을 Count: ${count} 형태로 출력하도록함.
- 개발자 도구에서 해당 컴포넌트를 선택하면 확인할 수 있음.
- 이는 주로 커스텀 hooks에서 사용

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

function useCounter(initialValue) {
  const [count, setCount] = useState(initialValue);

  useDebugValue(count, count => `Count: ${count}`);

  function handleIncrement() {
    setCount(count + 1);
  }
```

```
  function handleDecrement() {
    setCount(count - 1);
  }

  return {
    count,
    handleIncrement,
    handleDecrement,
  };
}
```

UseId

useId

- react에서는 컴포넌트의 상태를 변경할 때마다 렌더링이 발생하므로, ID를 생성하기 위해 일반적인 JS 변수를 사용하면 ID가 매번 재생성될 수 있다.
- 이러한 문제점을 해결하기 위해 고유한 ID를 생성할 수 있는 useId 훅이 도입되었다.
- 물론 유니크한 id가 있는 경우에는 굳이 useId를 사용할 필요는 없다.
- 유니크한 id가 없을 때만 해당 훅을 사용하면 된다.
- 아래와 같이 컴포넌트 최상단에서 useId를 호출하여 고유 ID 값을 생성하고 생성된 ID를 다른 속성에 전달할 수 있다.

```
import { useId } from 'react';

function PasswordField() {
  const passwordHintId = useId();
  // ...
<>
  <input type="password" aria-describedby={passwordHintId} />
  <p id={passwordHintId}>
</>
```

useId

- label 태그를 사용할때 ID 값을 하드 코딩하는 것보다 useId를 통해서 id값을 부여한다.
- 이렇게 하면 이 컴포넌트를 재사용해도 label에 사용된 id 값이 중복되지 않는다.

```
import { useId } from 'react';

function PasswordField() {
  const passwordHintId = useId();
  return (
    <>
      <label>
        Password:
        <input
          type="password"
          aria-describedby={passwordHintId}
        />
      </label>
      <p id={passwordHintId}>
        The password should contain at least 18 characters
      </p>
    </>
  );
}
```

useSyncExternalStore

useSyncExternalStore

- 리엑트에서 내부적으로 제공하는 useState, useReducer와 같은 상태관리 api가 아니라 자체적으로 상태관리 툴을 만들어 리엑트 훅과 연동시킨 상태관리 라이브러리들을 external store라고 한다. 대표적으로 mobx, redux, recoil, jotai, xtsate, zustand, rect query등이 있다.
- 이들의 상태 관리 흐름은 리엑트에서 관찰하지 않는다.
- 리액트가 외부 저장소의 변경사항을 "구독"할 수 있도록 도와주는 훅이다.
- 18버전에 들어오면서 리액트는 useTransition, useDeferredValue와 같이 렌더링을 일시중지하거나, 뒤로 미루는 등의 동시성 최적화를 도와줄 수 있는 훅들이 사용가능해졌다.
- 하지만 리액트에서 이렇게 동시성 렌더링이 가능해지면서, 외부 저장소의 데이터를 참조하는 컴포넌트를 렌더링할때 같은 시기에 렌더링을 했지만, 서로 다른 시점의 데이터를 참조할 수도 있는 Data Tearing 문제가 발생할 수 있게됐다.
- 이 문제를 해결해주는 훅이 바로 useSyncExternalStore 훅이다.
- 직관적으로 단어들을 분해해서 살펴보면 use + sync external store, 외부 스토어(external store)와 싱크(sync)를 맞추는 훅(use) 이다.

useSyncExternalStore

useSyncExternalStore

- 훅 시그니쳐

```
useSyncExternalStore(
  subscribe: (callback) => Unsubscribe,
  getSnapshot: () => State,
  getServerSnapshot?: () => State
)
```

- Subscribe

  콜백함수를 받아 스토어에 등록하는 용도로 사용된다. 스토어에 있는 값이 변경되면 콜백함수가 호출되어야 하며, 콜백이 호출되면 useSyncExternalStore가 이 훅을 사용하는 컴포넌트를 리렌더링한다.

- getSnapshot

  컴포넌트에 필요한 현재 스토어의 데이터를 반환하는 함수로, 스토어가 변경되지 않았다면 매번 함수를 호출할 때마다 동일한 값을 반환해야한다. 스토어에서 값이 변경됐다면, 이 값을 이전 값과 Object.is로 비교해 정말로 값이 변경됐다면 컴포넌트를 리렌더링한다.

- getServerSnapshot

  옵셔널 값으로, 서버 사이드 렌더링 하이드레이션하는 도중에만 사용된다. 서버사이드에서도 사용되는 훅이라면 반드시 이 값을 넘겨줘야 하며, 클라이언트의 값과 불일치가 발생할 경우 오류가 발생한다.

useSyncExternalStore

useSyncExternalStore

```
import { useSyncExternalStore } from "react";

function subscribe(callback) {
  window.addEventListener("resize", callback);
  return () => {
    window.removeEventListener("resize", callback);
  };
}

const useInnerHeight = () => {
  const innerHeight = useSyncExternalStore(
    subscribe,
    () => window.innerHeight
  );

  return innerHeight;
};

export default useInnerHeight;
```

- ✓ 일반적인 애플리케이션을 작성하는 상황에서는 크게 쓸 일이 많지만, 상태관리 라이브러리를 작성하거나, 리액트 외부의 무언가에 대한 값이 필요하고, 값이 변경될때마다 리렌더링이 수행되어야하는 케이스에 사용하면 적절하다.
- ✓ DOM, 글로벌 변수, 외부 상태관리 라이브러리 등을 사용할 때 사용할 수 있으며, 아래는 예시로 작성해본 window.innerHeight 값에 대한 useInnerHeight 훅이다.
- ✓ 첫번째 인수인 subscribe에는 resize 이벤트를 활용해서 innerHeight 값이 변경되는 상황에서 리렌더링이 발생할 수 있도록 해주고, 두번째 인수인 getSnapshot으로 window.innerHeight 값 반환을 통해 최신값을 받아올 수 있도록 커스텀 훅을 작성했다.

Custom Hook

Custom Hook

- 이 Hook은 개발자가 직접 작성하여 새로운 Hook을 생성할 수도 있는데, 이를 Custom Hook이라고 한다.
- Hook도 결국에는 함수 하나이기 때문에 이것이 React의 Hook으로 동작하는 함수인 것이냐, 단순 일반 함수인 것이냐에 대한 구분이 필요하다. 이를 위해React에서는 Custom hook을 작성하기 위한 규칙 몇가지를 정의했다.
- 조건문, 반복문 등에서 호출될 수 없고 컴포넌트 최상단에서만 호출 가능하다.
- React 컴포넌트 함수 내에서만 호출 되어야 한다.
- 함수 이름의 접두어는 반드시 'use' 로 지정해야 한다.

Report

자신의 소개 웹 사이트에 custom hook 추가하기

발표주제

HOC & Hooks
forceUpdate, flushSync
debouncing & throttling
await & async

Reference

- https://ko.legacy.reactjs.org/docs/hooks-reference.html
- https://react-ko.dev/reference/react/hooks
- https://react.dev/reference/react/useState
- https://react.vlpt.us/basic/20-useReducer.html

- 모던 자바스크립트 개발자를 위한 리액트 프로그래밍, 쿠지라히코우즈키에, 윤인성, 위키북스
- 명품 웹 프로그래밍, 황기태, 생능출판사
- 리액트 & 리액트 네이티브 통합 교과서, 아담 보두치, 강경일, 신희철, 에이콘
- Do it! 리액트 모던 웹 개발 with 타입스크립트, 전예홍, 이지스퍼블리싱

TL;DR