Web Programming

React programming

useReducer

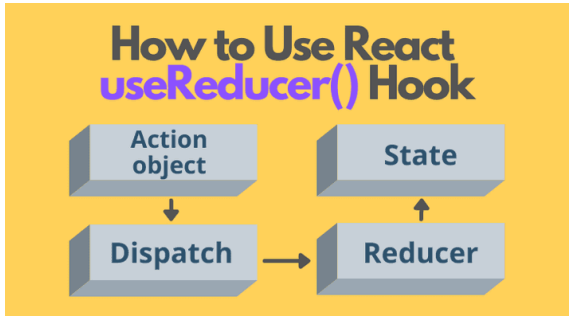
useReducer

- useState를 대체할 수 있는 함수이다.
- React에서 컴포넌트의 상태 관리를 위해 기본적으로 가장 많이 쓰이는 hook은 state이다.
- 좀 더 복잡한 상태 관리가 필요한 경우 reducer를 사용할 수 있다.
- reducer는 이전 상태와 Action을 합쳐, 새로운 state를 만드는 조작을 말한다.

```
import React, { useReducer } from "react";
const [state, dispatch] = useReducer(reducer, initialState, init);
```

① ② ③ ④ ⑤

1. state : 컴포넌트에서 사용할 상태
2. dispatch 함수
   - 첫번째 인자인 reducer 함수를 실행시킨다.
   - 컴포넌트 내에서 state의 업데이트를 일으키기 위해 사용하는 함수
3. reducer 함수
   - 컴포넌트 외부에서 state를 업데이트 하는 함수
   - 현재state, action 객체를 인자로 받아, 기존의 state를 대체하여 새로운 state를 반환하는 함수
4. initialState : 초기 state
5. init : 초기 함수



useReducer

2. dispatch 함수

- reducer 함수를 실행 시킨다.
- action 객체를 인자로 받으며 action 객체는 어떤 행동인지를 나타내는 type 속성과 해당 행동과 관련된 데이터(payload)를 담고 있다.
- action을 이용하여 컴포넌트 내에서 state의 업데이트를 일으킨다.

action

ex1) action type만 정의하여 사용

```
<button onClick={() => dispatch({ type: "INCREMENT" })}>증가</button>
```

ex2) action type과, 데이터를 정의하여 사용

```
<button onClick={() => dispatch({ type: "INCREMENT", payload: 1 })}>증가</button>
```

useReducer

3. reducer 함수

- 상기 dispatch 함수에 의해 실행되며, 컴포넌트 외부에서 state를 업데이트 하는 로직을 담당한다.
- 함수의 인자로는 state와 action을 받게 된다.
- state와 action을 활용하여 새로운 state를 반환한다.

ex1) action type만 정의하여 사용

```
function reducer(state, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      throw new Error("unsupported action type: ", action.type);
  }
}
```

useReducer

3. reducer 함수

ex2) action type과 데이터를 정의하여 사용

```
function reducer(state, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + action.payload };
    case "DECREMENT":
      return { count: state.count - action.payload };
    default:
      throw new Error("unsupported action type: ", action.type);
  }
}
```

payload

useReducer

Counter 예시

ex1) init 함수를 사용하지 않는 counter 예시

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

function reducer(state, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + action.payload };
    case "DECREMENT":
      return { count: state.count - action.payload };
    default:
      throw new Error("unsupported action type: ", action.type);
  }
}
```

```
const Counter = () => {
  const initialState = { count: 0 };
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      <h2>{state.count}</h2>
      <button onClick={() => dispatch({ type: "INCREMENT", payload: 1 })}>
        증가
      </button>
      <button onClick={() => dispatch({ type: "DECREMENT", payload: 1 })}>
        감소
      </button>
      <button onClick={() => dispatch({ type: "kkkkkkkkk", payload: 1 })}>
        에러
      </button>
    </>
  );
};

export default Counter;
```

useReducer

Counter 예시

ex2) init 함수를 사용하여 counter 예시

- 초기 state를 조금 지연해서 생성할 수 있다.
- init 함수를 세 번째 인자로 전달하면, 초기 state는 init(initialArg)에 설정될 것 이다.

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

function init(initialState) {
  //return { count: initialState };
  return { count: 0 };
}

function reducer(state, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + action.payload };
    case "DECREMENT":
      return { count: state.count - action.payload };
    case "RESET":
      return init(action.payload);
    default:
      throw new Error("unsupported action type: ", action.type);
  }
}
```

useReducer

Counter 예시

```
const Counter = ({ initialCount }) => {
  const [state, dispatch] = useReducer(reducer, initialCount, init);

  return (
    <>
      <h2>{state.count}</h2>
      <button onClick={() => dispatch({ type: "RESET", payload: 0 })}>
        초기화
      </button>
      <button onClick={() => dispatch({ type: "INCREMENT", payload: 1 })}>
        증가
      </button>
      <button onClick={() => dispatch({ type: "DECREMENT", payload: 1 })}>
        감소
      </button>
      <button onClick={() => dispatch({ type: "kkkkkkkkk", payload: 1 })}>
        에러
      </button>
    </>
  );
};

export default Counter;
```

useReducer

Counter 예시

ex3) useState 사용

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

const Counter = ({ initialCount }) => {
  const initial = initialCount ? initialCount : 0;
  const [count, setCount] = useState(initial);

  const onIncrease = () => {
    setCount((count) => count + 1);
  };

  const onDecrease = () => {
    setCount((count) => count - 1);
  };
```

```
  return (
    <>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(initial)}>초기화</button>
      <button onClick={onIncrease}>증가</button>
      <button onClick={onDecrease}>감소</button>
    </>
  );
};

export default Counter;
```

useReducer

useState vs useReducer

```jsx
import React, { useState } from "react";

const Counter = ({ initialCount }) => {
  const initial = initialCount ? initialCount : 0;
  const [count, setCount] = useState(initial);

  const onIncrease = () => {
    setCount((count) => count + 1);
  };

  const onDecrease = () => {
    setCount((count) => count - 1);
  };

  return (
    <>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(initial)}>초기화</button>
      <button onClick={onIncrease}>증가</button>
      <button onClick={onDecrease}>감소</button>
    </>
  );
};

export default Counter;
```

state 변경 함수

```jsx
import React, { useReducer } from "react";

function reducer(state, action) {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + action.payload };
    case "DECREMENT":
      return { count: state.count - action.payload };
    default:
      throw new Error("unsupported action type: ", action.type);
  }
}
```

state 변경함수

```jsx
const Counter = () => {
  const initialState = { count: 0 };
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      <h2>{state.count}</h2>      You, 22 hours ago • first …
      <button onClick={() => dispatch({ type: "INCREMENT", payload: 1 })}>
        증가
      </button>
      <button onClick={() => dispatch({ type: "DECREMENT", payload: 1 })}>
        감소
      </button>
      <button onClick={() => dispatch({ type: "kkkkkkkkk", payload: 1 })}>
        에러
      </button>
    </>
  );
};

export default Counter;
```

- useReducer를 활용 한다면 좀더 복잡한 프로세스를 처리할 수 있을 것이다.
- 그리고 state를 변경하는 부분(Count를 감소, 증가 시키는 부분)이 useState와 같은 경우 내부에 위치하며, useReducer와 같은 경우 외부에 위치 한다.

useContext

useContext

- 리액트의 일반적인 데이터 흐름은 부모 컴포넌트에서 자식 컴포넌트로 props를 통해 '단방향'으로 흐른다. 엄청 큰 컴포넌트 트리가 있다고 가정할 때 공통적으로 필요한 전역적인 데이터가 있을 수 있다.
- 전역 데이터를 일일이 props로 단계별로 전달해야 한다면 정말 비효율 적이다.
- 리액트는 이러한 문제점을 해결해 주는 Context API를 제공한다.
- 하위 컴포넌트에게 context를 제공하는 훅
- Context는 앱 안에서 전역적으로 사용되는 데이터를 여러 컴포넌트끼리 쉽게 공유할 수 있는 방법을 제공한다. Context를 사용하면 Props로 데이터를 일일이 전달해 주지 않아도 해당 데이터를 가지고 있는 상위 컴포넌트에 그 데이터가 필요한 하위 컴포넌트가 접근할 수 있다. 즉 사용자 정보, 테마, 언어 등 전역적인 데이터를 전달하기에 정말 편리하다.
- 상위 컴포넌트의 data가 필요한 하위 컴포넌트들은 useContext 훅을 사용해서 해당 데이터를 받아오기만 하면 된다.
- useContext는 Context로 분류한 데이터를 쉽게 받아올 수 있게 도와주는 역할을 한다.

useContext

Counter 예시

ex1) state와 props만 사용해서 만든 웹사이트

```
// 최상위 컴포넌트
import { useState } from "react";
import "./App.css";
import Page from "./Compopnents/Page";


function App() {
  // 현제 App이 다크모드인지 아닌지 true, false로 정보를 받고 있다.
  const [isDark, setIsDark] = useState(false);

  // Page 자식 컴포넌트에게 해당 데이터를 props로 넘겨 주고 있다.
  return <Page isDark={isDark} setIsDark={setIsDark}/>
}
export default App;
```

```
const contextValue = useContext(ContextObject);
```

context object (i.e. value returned from React.createContext)

current context value as given by the nearest context provider

```
// page 컴포넌트
import React, { useContext } from "react";
import Content from "./Context";
import Header from "./Header";
import Footer from "./Footer";

const Page = ({isDark, setIsDark}) => {
  return (
    <div className="page">
      <Header isDark={isDark}/>
      <Content isDark={isDark}/>
      <Footer isDark={isDark} setIsDark={setIsDark}/>
    </div>
  );
};

export default Page;
```

useContext

Counter 예시

```
import React from "react";

const Header = ({ isDark }) => {
  return (
    <header
      className="header"
      style={{
        backgroundColor: isDark ? "black" : "lightgray",
        color: isDark ? "white" : "black",
      }}
    >
      <h1>Welcome 홍길동</h1>
    </header>
  );
};

export default Header;
```

```
const Content = ({ isDark }) => {
  return (
    <div
      className="content"
      style={{
        backgroundColor: isDark ? "black" : "white",
        color: isDark ? "white" : "black",
      }}
    >
      <p>홍길동님, 좋은 하루 되세요 </p>
    </div>
  );
};

export default Content;
```

useContext

Counter 예시

```
import React from "react";

const Footer = ({ isDark, setIsDark }) => {
  const toggleTheme = () => {
    setIsDark(!isDark);
  };
  return (
    <footer
      className="footer"
      style={{ backgroundColor: isDark ? "black" : "lightgray" }}
    >
      <button className="button" onClick={toggleTheme}>
        Dark Mode
      </button>
    </footer>
  );
};
```

- ✓ App 컴포넌트가 가지고 있는 isDark 는 전체적인 테마에 관련된 data를 담고 있기 때문에 전역적이다. page 컴포넌트에게 isDark를 넘겨준다.
- ✓ 하지만 page 컴포넌트는 isDark를 받아오고 있지만 실질적으로 사용하지 않는다. 단지 자식 컴포넌트에게 전달만 해주고 있다. 그렇기 때문에 page 컴포넌트는 isDark 가 필요하지 않는 중간 컴포넌트라고 할 수 있다.

useContext

Counter 예시

ex2) context를 사용해서 만든 웹사이트

context/ThemeContext.js

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

// 기본값으로는 null을 넣어준다.
export const ThemeContext = createContext(null);
```

```
import { useState } from "react";
import "./App.css";
import Page from "./Compopnents/Page";
import { ThemeContext } from "./context/ThemeContext";

function App() {
  const [isDark, setIsDark] = useState(false);

  return (
    // 📌
    <ThemeContext.Provider value={{ isDark, setIsDark }}>
      <Page />
    </ThemeContext.Provider>
  );
}
export default App;
```

- ✓ App 컴포넌트로 돌아가서 위 context를 import 시켜준다.
- ✓ 그리고 page 컴포넌트를 만들어준 context의 provider로 감싸준다.
- ✓ context의 provider는 value라는 props를 받는데 이 안에는 전달하고자 하는 데이터를 넣어준다.
- ✓ ThemeContext 감싸는 모든 하위 컴포넌트는 props를 사용하지 않고 value로 넣어준 값에 접근할 수 있다.

useContext

Counter 예시

```
import React from "react";
import Content from "./Context";
import Header from "./Header";
import Footer from "./Footer";

// isDark 를 실질적으로 사용하지 않고, 자녀 컴포넌트들에게 전달하는 역할
// data 필요하지 않음 !
const Page = () => {
  return (
    <div className="page">
      <Header/>
      <Content/>
      <Footer/>
    </div>
  );
};

export default Page;
```

```
import { useContext } from "react";
import { ThemeContext } from "../context/ThemeContext";

const Header = () => {
const { isDark } = useContext(ThemeContext);
  return (
    <header
      className="header"
      style={{
        backgroundColor: isDark ? "black" : "lightgray",
        color: isDark ? "white" : "black",
      }}
    >
      <h1>Welcome 홍길동!</h1>
    </header>
  );
};

export default Header;
```

useContext

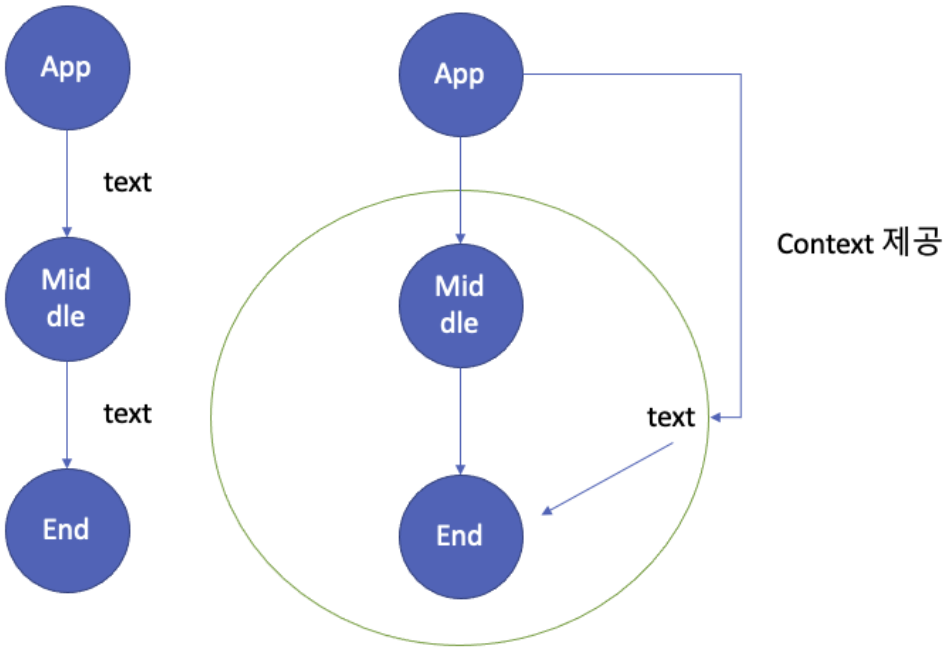
Counter 예시

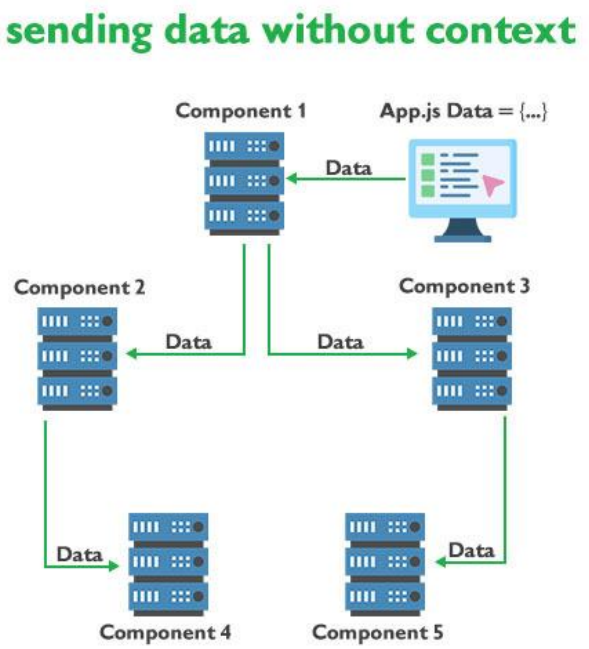
```
import React, { useContext } from "react";
import { ThemeContext } from "../context/ThemeContext";

const Content = () => {
    // 📌
  const { isDark } = useContext(ThemeContext);

  return (
    <div
      className="content"
      style={{
        backgroundColor: isDark ? "black" : "white",
        color: isDark ? "white" : "black",
      }}
    >
      <p>홍길동님, 좋은 하루 되세요 </p>
    </div>
  );
};

export default Content;
```

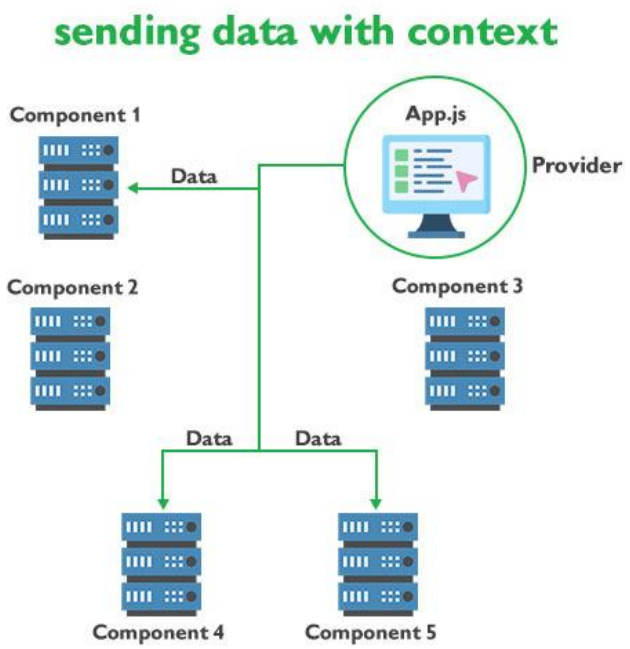
```
import React, { useContext } from "react";
import { ThemeContext } from "../context/ThemeContext";

const Footer = () => {
  // 📌
  const { isDark, setIsDark } = useContext(ThemeContext);
  const toggleTheme = () => {
    setIsDark(!isDark);
  };
  return (
    <footer
      className="footer"
      style={{ backgroundColor: isDark ? "black" : "lightgray" }}
    >
      <button className="button" onClick={toggleTheme}>
        Dark Mode
      </button>
    </footer>
  );
};

export default Footer;
```

useContext

useContext







Context

useContext

context의 값 변경하기

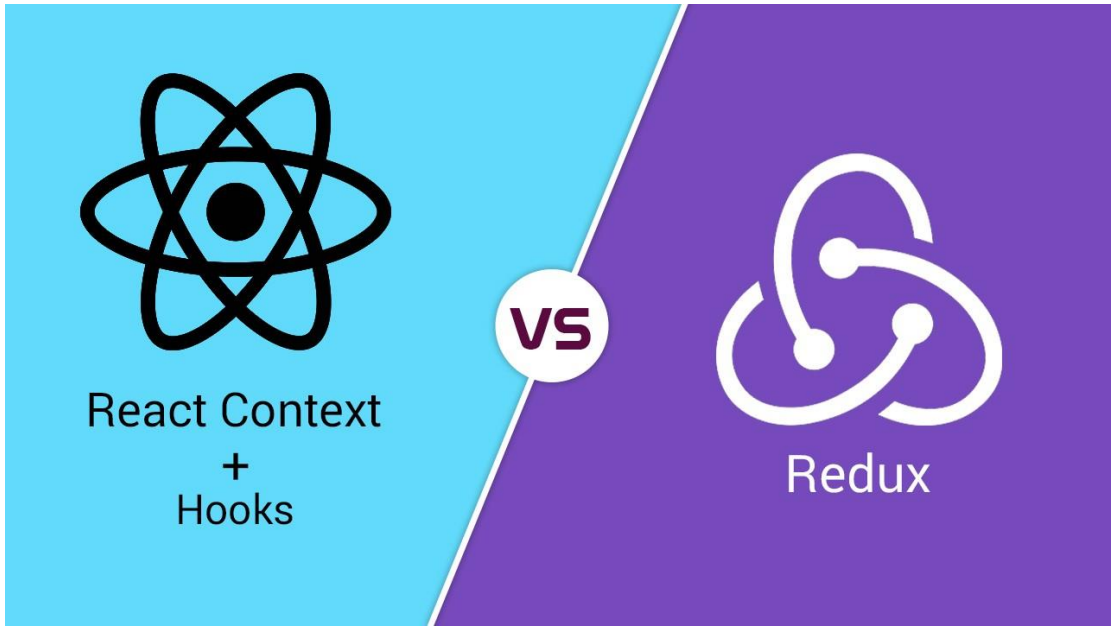
```
import React,{createContext, useContext, useState} from 'react'

const TextContext = createContext("Hello world")

function End(){
    const text = useContext(TextContext);
    return (
        <div>
            {text}
        </div>
    )
}

function Middle(){

    return (
        <div>
            <End/>
        </div>
    )
}
```

"Context" 의 값을 변경하는 것은 매우 간단하다. 바로 "provider"의 value를 업데이트를 하면 이를 구독하는 컴포넌트가 다시 렌더링한다.

```
export default function App() {

    const [text, setText] = useState("Hello world")

    return (
        <div>
            <TextContext.Provider value={text}>
                <Middle/>
            </TextContext.Provider>
            <input type="text" onChange={(e) => setText(e.target.value)} />
        </div>
    )
}
```

useContext

useContext vs Redux

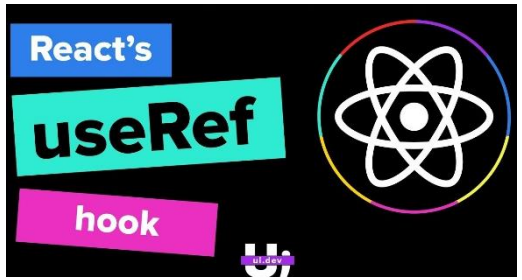


Redux Redux

useRef

useRef

- useRef 는 .current 프로퍼티로 전달된 인자(initialValue)로 초기화된 변경 가능한 ref 객체를 반환한다.
- 반환된 객체는 컴포넌트의 전 생애주기를 통해 유지된다.
- useRef는 저장공간 또는 DOM요소에 접근하기 위해 사용되는 React Hook이다.
- 여기서 Ref는 reference, 즉 참조를 뜻한다.
- 우리가 자바스크립트를 사용 할 때에는, 우리가 특정 DOM 을 선택하기 위해서 querySelector 등의 함수를 사용한다.
- React를 사용하는 프로젝트에서도 가끔씩 DOM 을 직접 선택해야 하는 상황이 필요하다. 그럴 때 우리는 useRef라는 React Hook을 사용한다.



useRef

useRef 예시

1) useRef 사용 예시 - 변수 관리

```
import { useState, useRef } from "react";
import "./styles.css";

function App() {
  const [stateCount, setStateCount] = useState(0);
  const refCount = useRef(0);
  let varCount = 0;

  function upState() {
    setStateCount(stateCount + 1);
    console.log("stateCount : ", stateCount);
  }

  function upRef() {
    ++refCount.current;
    console.log("refCount : ", refCount.current);
  }

  function upVar() {
    ++varCount;
    console.log("varCount : ", varCount);
  }

  return (
    <div>
      <div>stateCount : {stateCount} </div>
      <div>refCount : {refCount.current} </div>
      <div>varCount : {varCount} </div>
      <br />
      <button onClick={upState}>state up</button>
      <button onClick={upRef}>ref up</button>
      <button onClick={upVar}>var up</button>
    </div>
  );
}

export default App;
```

stateCount : 9
refCount : 10
varCount : 0

state up | ref up | var up

```
stateCount : 0
refCount : 1
varCount : 1
stateCount : 1
refCount : 2
varCount : 1
varCount : 2
varCount : 3
refCount : 3
varCount : 4
varCount : 5
stateCount : 2
varCount : 1
```

useRef

useRef 예시

- useState를 사용해 값을 저장한 stateCount, useRef를 사용해 값을 저장한 refCount, 변수를 통해 값을 저장한 varCount가 있다.
- ref up 버튼을 누르면 각각의 값이 올라가고 콘솔창에 출력이 되는 형태인데, ref up을 아무리 눌러도 콘솔창에만 출력되고 화면에 렌더링이 되지 않는 것을 확인할 수 있다.
- 하지만 state up 버튼을 누르면 화면이 렌더링되면서 올려놨던 refCount값도 화면에 출력되게 된다. 이를 통해 우리는 useRef로 관리하는 값은 값이 변해도 화면이 렌더링되지 않음을 알 수 있다.
- 컴포넌트 함수가 다시 호출된다는 것은 함수 내부의 변수들이 모두 다시 초기화가 되고 함수의 모든 로직이 다시 실행된다는 것을 의미한다.
- var up 버튼을 통해 변수 값을 아무리 높여 놓아도 state up 버튼을 통해 렌더링을 한다면, 컴포넌트 내부에 있는 let varCount = 0; 이 계속 실행되기에 값이 초기화된다.

useRef

useRef 예시

2) useRef 사용 예시 - DOM 요소 선택

```
import { useRef, useEffect } from "react";
import "./styles.css";

function DOMFocus() {
  const inputRef = useRef();

  function focus() {
    inputRef.current.focus();
    console.log(inputRef.current);
  }

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="아이디 또는 이메일" />
      <button>Login</button>
      <br />
      <button onClick={focus}>focus</button>
    </div>
  );
}

export default DOMFocus;
```

- ✓ Focus 버튼을 누르면 input 창에 focus되는 것을 확인할 수 있다.
- ✓ useRef를 통해 마운트될 때 focus되게 설정해 놓을 수도 있지만 블로그 포스팅이라는 환경 상, 버튼을 누르면 focus되도록 설정해놓았다.
- ✓ 또 focus 버튼을 누르고 콘솔창에 useRef를 통해 선택된 DOM 요소를 확인할 수 있도록 해놓았으니 한번쯤 확인해보면 좋을 듯 하다.

useRef 예시

3) useRef 사용 예시 - DOM 요소 선택

```
import React, { useState, useRef } from "react";

const InputSample = () => {
  const [inputs, setInputs] = useState({
    이름: "",
    nickname: "",
  });

  const nameFocus = useRef();

  const { 이름, nickname } = inputs;

  const onChange = (e) => {
    const { value, name } = e.target;
    setInputs({
      ...inputs,
      [name]: value,
    });
  };
```

```
  const onReset = () => {
    setInputs({
      이름: "",
      nickname: "",
    });
    nameFocus.current.focus();
  };
  return (
    <div>
      <input
        name="이름"
        placeholder="이름쓰세요"
        onChange={onChange}
        value={이름}
        ref={nameFocus}
      />
```

```
      <input
        name="nickname"
        placeholder="닉네임쓰세요"
        onChange={onChange}
        value={nickname}
      />
      <button onClick={onReset}>초기화</button>
      <div>
        <b>값:</b>
        {이름}({nickname})
      </div>
    </div>
  );
};

export default InputSample;
```

forwardRef

forwardRef

- (함수형 컴포넌트에서) 부모 컴포넌트에서 자식 컴포넌트 안의 **DOM element**에 접근하고 싶다면, forwardRef를 사용해야 한다.
- forwardRef는 부모 컴포넌트에서 생성된 ref를 자식 컴포넌트에게 전달하며, 이를 통해 부모 컴포넌트가 자식 컴포넌트의 인스턴스에 접근할 수 있도록 한다.

forwardRef 사용 이유

- 함수형 컴포넌트는 인스턴스가 없기 때문에 ref 속성을 사용할 수 없다.
- 따라서, 함수형 컴포넌트를 forwardRef로 감싸주게 되면 ref를 사용할 수 있다.
  (+ 클래스로 선언된 컴포넌트들은 인스턴스를 가지기 때문에 ref 속성 사용 가능)
- 부모 컴포넌트에서 자식 컴포넌트 안의 DOM element에 접근하고 싶을 때 사용한다.

forwardRef

[참고] useRef ?

- forwardRef를 사용하는 방법에 대해 설명하기 전에, useRef라는 Hook에 대한 개념 이해가 필요하다.
- useRef는 DOM element에 직접 접근하기 위해 사용하는 Hook이다.
- 왜 useRef를 통해 DOM element에 접근해야 하냐면, React는 가상돔을 기반으로 작동하는 라이브러리이기 때문이다.
- 컴포넌트가 mount될 때 React는 current 프로퍼티에 DOM element를 대입하고, 컴포넌트가 unmount될 때 프로퍼티를 다시 null로 돌려놓는다.
- 실제 DOM에 React 노드가 렌더될 때까지 ref가 가리키는 DOM element의 주소 값은 확정된 것이 아니다.

forwardRef

forwardRef 사용법

1. 부모 컴포넌트에서 useRef()를 선언하고, 자식 컴포넌트에 보낸다.

```
import { useRef } from 'react'
import Child from './Child'

const Parent = () => {
  const compRef = useRef()

  return (
    <Child ref={compRef} />
  )
}

export default Parent;
```

forwardRef

forwardRef 사용법

2. 자식 컴포넌트를 forwardRef()로 감싸고, 부모에서 사용할 함수를 useImperativeHandle()로 감싼다.

```
import { forwardRef, useImperativeHandle } from 'react'

const Child = forwardRef((props, ref) => {
  // 부모 컴포넌트에서 사용할 함수 설정
  useImperativeHandle(ref, () => ({
    req1,
    req2
  }))

  // 함수 1
  const req1 = () => { }

  // 함수 2
  const req2 = () => { }
}

export default Child
```

forwardRef()
자식 컴포넌트를 forwardRef로 감쌌을 때, 2번째 매개변수인 ref는 부모 컴포넌트가 props로 넘긴 값(=compRef)이다.

useImperativeHandler()
useImperativeHandler의 첫 번째 인수로는 부모 컴포넌트가 props로 넘긴 ref를 전달한다.
두 번째 인수로는 콜백함수를 전달한다(콜백함수의 반환값이 바로 부모 컴포넌트 내 useRef Hook 호출시 반환된 객체의 current 프로퍼티에 바인딩 될 값이 된다).

forwardRef

forwardRef 사용법

3. 부모 컴포넌트에서 current 프로퍼티를 통해 함수를 사용한다.

```
import { useRef } from 'react'
import Child from './Child'

const Parent = () => {
  const compRef = useRef()

  const fnReqBtn1 = e => {
    e.preventDefault();

    compRef.current.req1();
  }
```

```
  const fnReqBtn2 = e => {
    e.preventDefault();

    compRef.current.req2();
  }

  return (
    <>
      <Child ref={compRef} />
      <Link onClick={fnReqBtn1}>가입버튼1</Link>
      <Link onClick={fnReqBtn2}>가입버튼2</Link>
    </>
  )
}

export default Parent;
```

forwardRef

부모 컴포넌트에 DOM 노출하기

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

const MyInput = forwardRef(function MyInput(props, ref) {
  const { label, ...otherProps } = props;
  return (
    <label>
      {label}
       <input {...otherProps} ref={ref} />
    </label>
  );
});
```

```
function Form() {
  const ref = useRef(null);

  function handleClick() {
    ref.current.focus();
  }

  return (
    <form>
      <MyInput label="Enter your name:" ref={ref} />
      <button type="button" onClick={handleClick}>
        Edit
      </button>
    </form>
  );
}
```

Form 컴포넌트는 MyInput 컴포넌트의 ref 속성의 useRef를 사용하여 참조 값을 전달한다.
MyInput 컴포넌트는 전달받은 참조 값을 input 태그의 ref에 전달한다. 이런 과정을 통해 부모 컴포넌트인 Form 컴포넌트는 자식 컴포넌트인 MyInput 컴포넌트의 input 태그에 포커스를 줄 수 있게 된다.

useImperativeHandle

useImperativeHandle

- React 함수형 컴포넌트는 부모 컴포넌트와 자식 컴포넌트 간에 데이터와 함수를 주고받는 일이 일반적이다. 이러한 데이터와 함수 전달은 React의 핵심 원리 중 하나다.
- useImperativeHandle 훅은 자식 컴포넌트가 부모 컴포넌트로부터 함수나 메서드를 노출하고 커스터마이징할 때 사용된다.
- 이를 통해 부모 컴포넌트는 자식 컴포넌트 내부의 **특정 함수나 메서드에 직접 접근**할 수 있으며, 이를 활용하여 자식 컴포넌트의 인터페이스를 더 쉽게 조작하거나 노출할 수 있다.
- child component의 상태 변경을 parent component에서 하거나, child component의 핸들러를 parent component에서 호출해야 하는 경우 사용하는 훅이다.
- ref로 노출할 대상을 커스텀할 때 사용되는 훅이다.

useImperativeHandle

부모 컴포넌트에게 커스텀 함수 노출하기

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

function ChildComponent(props, ref) {
  useImperativeHandle(ref, () => {
    return {
      getText: () => 'useImperativeHandle 테스트'
    };
  }, []);

  return <span>children ref 테스트</span>
}

export default ChildComponent;
```

```
import { useEffect, useRef, forwardRef } from 'react';
import ChildComponent from './ChildComponent';

const ForwardedChild = forwardRef(ChildComponent);

function ParentComponent() {
  const childRef = useRef(null);

  useEffect(() => {
    console.log(childRef.current?.getText()); // 'useImperativeHandle 테스트'
  }, []);

  return (
    <div>
      <ForwardedChild ref={childRef} />
    </div>
  );
}

export default ParentComponent;
```

useImperativeHandle

부모 컴포넌트에게 커스텀 함수 노출하기

```
import { useRef, useImperativeHandle } from 'react';

function MyInput(props, ref) {
  const inputRef = useRef(null);

  useImperativeHandle(ref, () => {
    return {
      focus() {
        inputRef.current.focus();
      }
    };
  }, []);

  return <input type="text" ref={inputRef} />;
}

export default MyInput;
```

```
import { useRef,forwardRef } from 'react';
import MyInput from './MyInput';

const ForwardedMyInput = forwardRef(MyInput);
function Form() {
  const ref = useRef(null);
  function handleClick() {
    // useImperativeHandle에서 focus만 노출시키고 있다.
    // ref.current에서 input 관련 다른 변수나 함수 접근 시도 시 에러 발생한다.
    ref.current.focus();
  }
  return (
    <form>
      <ForwardedMyInput ref={ref} />
      <button type="button" onClick={handleClick}>
        Edit
      </button>
    </form>
  );
}
export default Form;
```

useImperativeHandle

useImperativeHandle

- React ref와 forwardRef 그리고 useImperativeHandle
- React ref와 forwardRef 그리고 useImperativeHandle 제대로 알기
- 자식 컴포넌트 제어
- createRef와 useRef 그리고 useImperativeHandle
- Ref의 개념과 useImperativeHandle, Uncontrolled form
- useImperativeHandle Hook Ultimate Guide
- React 컴포넌트에 ref props 전달하기
- Mastering Refs in React 18: useRef and useImperativeHandle Explained
- React forwardRef(): How to Pass Refs to Child Components
- 부모 컴포넌트가 자식 컴포넌트의 메서드나 변수에 직접 접근

useEffect

useEffect

- useEffect 함수는 리액트 컴포넌트가 렌더링 될 때마다 특정 작업을 실행할 수 있도록 하는 Hook이다.
- useEffect는 component가 mount 됐을 때, component가 unmount 됐을 때, component가 update 됐을 때 특정 작업을 처리할 수 있다.
- 즉, 클래스형 컴포넌트에서 사용할 수 있었던 생명주기 메소드를 함수형 컴폰넌트에서도 사용할 수 있게 된 것이다.
- 기본 형태

  useEffect(function, deps)

  ✓ function : 수행하고자 하는 작업

  ✓ deps : 배열 형태이며, 배열 안에는 검사하고자 하는 특정 값 or 빈 배열

useEffect

1. Component가 mount 됐을 때(처음 나타났을 때)

- 컴포넌트가 화면에 가장 처음 렌더링
  - ➢ 컴포넌트가 화면에 가장 처음 렌더링 될 때 한 번만 실행하고 싶을 때는 deps에 빈 배열을 넣는다.
  - ➢ 만약에 배열을 생략한다면 리렌더링 될 때마다 실행된다.

```
useEffect(() => {
                console.log("마운트 될 때만 실행된다.");
}, []);
```

useEffect

2. Component가 update될 때(특정 props, state가 바뀔 때)

- 특정 값이 업데이트 될 때마다 실행하고 싶을 때는 deps 배열 안에 검사하고 싶은 값을 넣어준다.
- (의존 값이 들어있는 배열 deps라고 한다. dependency를 의미한다)
- 업데이트 될 때만 실행하는 것이 아니라 마운트 될 때도 실행된다.
- 따라서 업데이트 될 때만 함수를 실행하고 싶다면 아래와 같은 꼼수(?)를 사용하면 된다.

```
//코드 생략

const mounted = useRef(false);

useEffect(() => {
                if(!mounted.current){
                                mounted.current = true;
                } else {
                //특정 작업 수행
                }
}, [바뀌는 값]);

//코드 생략
```

useEffect

3. Component가 unmount될 때(사라질 때) or update 되기 직전에

- cleanup 함수 반환(return 뒤에 나오는 함수이며 useEffect에 대한 뒷정리 함수라고 한다).

❖ 언마운트 될 때만 cleanup 함수를 실행하고 싶을 때

✓ 두 번째 파라미터로 빈 배열을 넣는다.

❖ 특정 값이 업데이트되기 직전에 cleanup 함수를 실행하고 싶을 때

✓ deps 배열 안에 검사하고 싶은 값을 넣어준다.

```
useEffect(() => {
        const fetchData = async() => {
                const response = await fetch('API_URL');
                const data = await response.json();
                setState(data);
        };
        fetchData();
}, [inputValue]);
```

useEffect

useEffect 주의점

- useEffect를 사용할 때 가장 주의해야 할 점은 무한 루프에 빠지지 않도록 하는 것이다. 이를 방지하기 위해, 의존성 배열을 올바르게 관리해야 한다.
- 또한, useEffect 내부에서 상태를 변경할 때는 해당 상태가 의존성 배열에 포함되어 있는지 확인해야 한다. 그렇지 않으면, 상태 변경에 의해 컴포넌트가 다시 렌더링되고, 이는 또 다른 useEffect 호출을 야기할 수 있다.
- 클린업 함수의 중요성도 강조하고 싶다. 메모리 누수를 방지하기 위해, useEffect 내에서 시작된 모든 작업은 컴포넌트가 언마운트 될 때 정리되어야 한다.
- useEffect는 리액트에서 가장 강력하면서도 복잡한 훅 중 하나다. 그러나 이를 올바르게 사용한다면, 컴포넌트의 생명주기를 효과적으로 관리하고, 성능을 최적화할 수 있다.
- 마지막으로, useEffect를 사용할 때는 **리액트 공식 문서**를 참고하는 것이 좋다. 공식 문서에는 useEffect의 사용법과 주의점이 자세히 설명되어 있다.

useEffect

useEffect – 기본 사용법

```
import { useEffect } from "react"; //useEffect를 사용하기 위해 import

export default function UseEffectTest() {
  console.log("useEffect 전");

  // useEffect도 함수기 때문에 함수 호출
  useEffect(() => {
    console.log("메롱으로 바꿀거지롱");
    const hi = document.getElementById("hi");
    hi.innerText = "메롱";
  });

  console.log("useEffect 후");
  return (
    <div className="App">
      <h1 id="hi">안녕하세요.</h1>
    </div>
  );
}
```

useEffect

useEffect – 세가지 사용법

1. 무한반복
   - 위 코드는 맨처음 렌더링, 그리고 매 재렌더링 마다 실행되는 useEffect문이다.
   - 특징으로는 useEffect 함수에 특정 콜백을 전달한다는 정도이다. 이 콜백 함수는 매 렌더링이 이루어지고 나서 실행될 것이다.

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

export default function UseEffectTest() {
  const [count, setCounter] = useState(0);

  useEffect(() => {
    console.log(`useEffect: ${Date()}`);
  });

  const countHandler = (e) => {
    setCounter((s) => s + 1);
  };
```

```
  return (
    <div className="App">
      <h1 id="hi">{count}</h1>
      <button onClick={countHandler}>카운터 증가</button>
    </div>
  );
}
```

useEffect

useEffect – 세가지 사용법

2. 처음에만 실행
   - 이전 예제와 동일하지만, useEffect 부분에 두번째 인자가 생겼다.
   - 두번째 인자를 빈 배열로 전달하게 되면, 리액트는 최초 렌더링될 때만 useEffect를 실행한다.
   - 초기에 무언가를 할 때 유용할 것이다.

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

export default function App() {
  const [count, setCounter] = useState(0);

  useEffect(() => {
    console.log(`useEffect: ${Date()}`);
  }, []);

  const countHandler = (e) => {
    setCounter((s) => s + 1);
  };
```

```
  return (
    <div className="App">
      <h1 id="hi">{count}</h1>
      <button onClick={countHandler}>카운터 증가</button>
    </div>
  );
}
```

useEffect – 세가지 사용법

3. 의존성 배열 사용

- state를 2개 만들었다. 그리고 useEffect의 두번째 인자인 배열을 확인해 보자. firstCount가 배열에 들어있다. 이를 의존성 배열이라고 부르며, 이것이 의미하는 것은 단순히 해당 컴포넌트가 재렌더링될 때 useEffect를 실행하는 것이 아닌, 의존성 배열에 존재하는 데이터가 변경될 때만 실행하라는 것을 나타낸다.
- 즉, 위 코드의 useEffect는 firstcount가 변경되는 firstCountHandler 이벤트 핸들러가 호출될 때만 실행될 것이다(처음 렌더링 때도 실행됨).

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

export default function UseEffectTest() {
  const [firstCount, setFirstCounter] = useState(0);
  const [secondCount, setSecondCounter] = useState(0);

  useEffect(() => {
    console.log(`useEffect: ${Date()}`);
  }, [firstCount]);

  const firstCountHandler = (e) => {
    setFirstCounter((s) => s + 1);
  };
```

```
  const secondCountHandler = (e) => {
    setSecondCounter((s) => s + 1);
  };

  return (
    <div className="App">
      <h1>{firstCount}</h1>
      <button onClick={firstCountHandler}>카운터 증가</button>
      <h1 id="hi">{secondCount}</h1>
      <button onClick={secondCountHandler}>카운터 증가</button>
    </div>
  );
}
```

useEffect

useEffect – cleanup

- useEffect는 기본적으로 컴포넌트가 재렌더링될 때 실행된다고 했다. API 요청으로 인한 데이터 획득, 로깅, DOM 조작 등의 행위는 일반적으로 문제가 되지 않지만, 특정 작업의 경우에는 메모리 누스 등의 문제를 해결하기 위한 방법이 필요하다.
- 예를 들어 setInterval이나 setTimeout 등과 같은 이벤트는 등록되고 나서 clearInterval, clearTimeout 등이 호출되지 않으면 사라지지 않는다. 따라서 이를 처리하는 방법을 필요로 하게 된다.

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

export default function UseEffectTest() {
  const [count, setCount] = useState(1000);

  useEffect(() => {
    setInterval(() => console.log(count), count);
  }, [count]);
```

```
  const countHandler = (e) => {
    setCount((c) => c + 1000);
  };

  return (
    <div className="App">
      <h1>{count}</h1>
      <button onClick={countHandler}>카운트 증가</button>
    </div>
  );
}
```

useEffect

useEffect – cleanup

- 앞의 코드는 사용자가 버튼을 클릭할 때마다 해당 count를 기준으로 setInterval을 지정한다. 그러나 매번 useEffect가 실행될 때마다 기존의 seInterval이 사라지지 않는다. 이는 새로운 setInterval을 실행할 때마다 clearInterval을 호출하지 않았기 때문이다.
- 그렇다면 언제 clearInterval을 호출해야 할까?

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

export default function UseEffectTest() {
  const [count, setCount] = useState(1000);

  useEffect(() => {
    console.log("useEffect");
    const interval = setInterval(() => console.log(count), count);

    return () => {
      clearInterval(interval);
      console.log("clearInterval");
    };
  }, [count]);
```

```
  const countHandler = (e) => {
    setCount((c) => c + 1000);
  };

  return (
    <div className="App">
      <h1>{count}</h1>
      <button onClick={countHandler}>카운트 증가</button>
    </div>
  );
}
```

useEffect

useEffect – cleanup

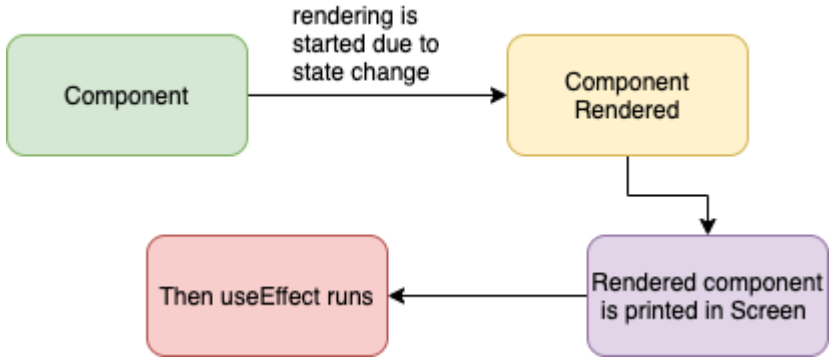
- 이전코드와 달라진 부분은 useEffect 함수에 return문이 추가되었다는 점이다. 위 코드는 useEffect가 다시 호출되기 전에 useEffect의 return에 작성한 콜백이 실행된다. 이러한 콜백을 cleanup 함수라고 부른다. 결과를 확인해 보면 useEffect가 실행되고 버튼을 클릭하면 다음 useEffect가 실행되기 전에 clearInterval이 실행되는 것을 확인할 수 있다.
- (처음 useEffect가 두번 실행되는 것은 react의 StrictMode 때문)

- return을 사용하지 않으면 새로운 useEffect를 수행하기 전에 아무런 작업을 하지 않는 것으로 인식한다. 또한 의존성 배열도 주의해야 하는데, 만약 의존성 배열이 []로 되어있으면 해당 useEffect는 첫 렌더링 시에만 동작할 것이다. 따라서 return에 작성된 내용은 컴포넌트가 최종적으로 DOM에서 unmount되는 시점에만 수행될 것이다.
- 이에 반해 의존성 배열 내에 특정 변수(의존성)이 존재하면 해당 변수들의 변경이 감지될 때마다 useEffect가 재수행될 것이기 때문에 cleanup 함수 또한 계속 실행될 것이다.

useLayoutEffect

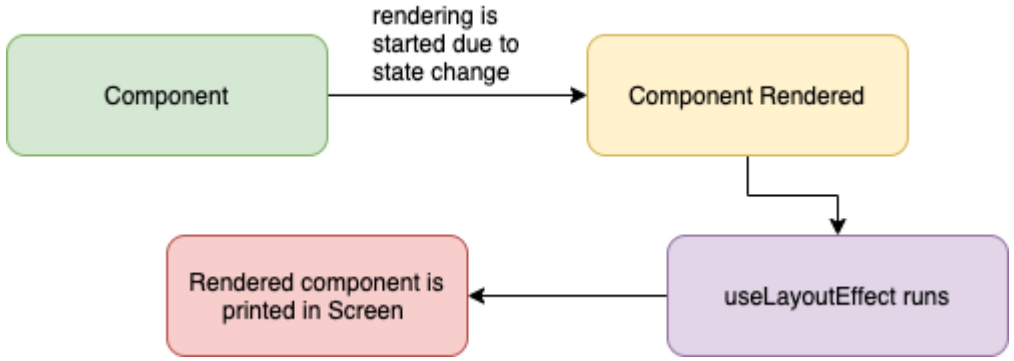
useEffect

- useEffect 는 컴포넌트들이 render 와 paint 된 후 실행된다. 비동기적(asynchronous) 으로 실행된다. Paint 된 후 실행되기 때문에, useEffect 내부에 DOM에 영향을 주는 코드가 있을 경우 사용자 입장에서는 화면의 깜빡임을 보게 된다.



useLayoutEffect

- useLayoutEffect 는 컴포넌트들이 render 된 후 실행되며, 그 이후에 paint 가 된다. 이 작업은 동기적(synchronous) 으로 실행된다. Paint 가 되기 전에 실행되기 때문에 DOM을 조작하는 코드가 존재하더라도 사용자는 깜빡임을 경험하지 않는다.



useLayoutEffect

useLayoutEffect

- useLayoutEffect는 useEffect와 동일하지만, 렌더링 후 layout과 paint 전에 동기적으로 실행된다.
- 때문에 설령 DOM을 조작하는 코드가 존재하더라도, 깜빡임 현상이 없다.

- useLayoutEffect는 DOM이 그려지기 이전 시점에 동기적으로 수행된다.
- 즉 컴포넌트들이 render 된 후 실행되며, 그 이후에 paint 된다. paint가 되기 전에 실행되기 때문에 DOM을 조작하는 코드가 존재하더라도 사용자는 깜빡임을 경험하지 않는다.

- useEffect와 useLayoutEffect 훅의 형태는 완전히 동일하다.

```
useEffect(() => {
   first

   return () => {
     second
   }
 }, [third])
```

```
useLayoutEffect(() => {
   first

   return () => {
     second
   };
 }, [third])
```

- useEffect의 이펙트는 DOM이 화면에 그려진 이후에 호출된다.
- useLayoutEffect의 이펙트는 DOM이 화면에 그려지기 전에 호출된다.
- 따라서 렌더링할 상태가 이펙트 내에서 초기화되어야 할 경우, 사용자 경험을 위해 useLayoutEffect를 활용하자!

useLayoutEffect

useLayoutEffect

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

export default function App() {
  const [age, setAge] = useState(0);
  const [name, setName] = useState("");

  useEffect(() => {
    setAge(25);
    setName("찬민");
  }, []);

  return (
    <>
      <div className="App">{`그의 이름은 ${name} 이며,
나이는 ${age}살 입니다.`}</div>
    </>
  );
}
```

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

export default function App() {
  const [age, setAge] = useState(0);
  const [name, setName] = useState("");

  useLayoutEffect(() => {
    setAge(25);
    setName("찬민");
  }, []);

  return (
    <>
      <div className="App">{`그의 이름은 ${name} 이며,
나이는 ${age}살 입니다.`}</div>
    </>
  );
}
```

useLayoutEffect

useLayoutEffect

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

const Practice = () => {
  const [logoUrl, setLogoUrl] = useState('');

  useEffect(() => {
    setLogoUrl('logo192.png');
  }, []);

  return (
    <>
      <img alt='test' src={logoUrl} />
    </>
  );
};

export default Practice;
```

useEffect를 useLayoutEffect로 변경해 보기

새로 고침하면 깜박거림이 발생, 바꾸면 발생하지 않음

- 스크롤 위치를 찾거나 어떤 element의 스타일 요소를 변경하는 등 직접적으로 DOM을 조작하는 곳을 제외하고는 useEffect를 사용하는 것을 추천한다.
- 공식 문서에서도 useEffect를 먼저 사용한 후에, 문제가 생긴다면 그때 useLayoutEffect를 사용하는 것을 추천하고 있다.

useInsertionEffect

useInsertionEffect

- React 18에서 추가된 Hook
- CSS-in-JS 방식을 사용하는 라이브러리를 위한 Hook
- 이 경우가 아니라면, useEffect나 useLayoutEffect를 사용하는 것이 권장됨.
- DOM 업데이트 이전에 호출되는 useEffect의 한 형태이다.
- 일반적으로 DOM 변경 전 동기적으로 스타일을 주입하는 데 사용된다.
- useEffect와 마찬가지로, 서버사이드에선 실행되지 않고 클라이언트사이드에서만 실행된다.

왜 쓰는가?

- 렌더링 중에 스타일을 주입하고 React가 non-blocking update를 처리하는 경우, 브라우저는 컴포넌트 트리를 렌더링하는 동안 매 프레임마다 스타일을 다시 계산하므로 느릴 수 있음.
- useInsertionEffect는 컴포넌트에서 다른 Effect가 실행될 때 이미 실행되어 있음이 보장되므로 style이 주입된 상태에서 Effect 실행 가능

useInsertionEffect

사용하는 경우

- CSS-in-JS 라이브러리를 사용하는 경우
- 렌더링 전 다크모드 여부를 결정해야 할 경우

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

// 컴포넌트
function MyButton() {

        function useCSS(rule) {
        useInsertionEffect(() => {
        // ... <style> 태그를 여기에서 주입하세요 ...
        });
                return rule;
        }

  const className = useCSS('...');
  return <div className={className} />;
}
```

Report

Github 정리

발표주제

Memoization
CSS-in-JS vs CSS
일급객체

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