📌 React Hook: useState

🚀 1. useState란?

📖 개념

- `useState`는 **React 함수형 컴포넌트에서 상태(state)를 관리하기 위해 사용하는 가장 기본적인 Hook**입니다.
- 상태는 사용자 인터랙션, API 응답 등으로 인해 변경될 수 있는 **동적인 데이터**입니다.
- `useState`를 통해 컴포넌트의 상태를 선언하고, **변경될 때마다 컴포넌트를 리렌더링**하게 만들 수 있습니다.

const [state, setState] = useState(initialValue);

- `state`는 현재 상태 값
- `setState`는 상태를 변경하는 함수
- `initialValue`는 상태의 초기값

💡 2. 기본 사용법

📌 숫자 상태 예제

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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>현재 카운트: {count}</p>
      <button onClick={() => setCount(count + 1)}>증가</button>
    </div>
  );
}
```

✅ 상태 값이 변경되면 해당 컴포넌트가 다시 렌더링됩니다.

🛠 3. 다양한 타입의 상태 관리

📌 문자열 상태

```
const [name, setName] = useState('Alice');
```

📌 배열 상태

```
const [items, setItems] = useState([]);
setItems([...items, '새로운 아이템']);
```

📌 객체 상태

```
const [user, setUser] = useState({ name: 'Alice', age: 25 });
setUser({ ...user, age: 26 });
```

✅ 객체나 배열의 상태는 **불변성(immutability)** 을 지키기 위해 **spread 연산자(...)**를 사용합니다.

⚡ 4. 상태 업데이트 시 주의사항

📌 상태 업데이트는 비동기적이다

```
setCount(count + 1);
setCount(count + 1);
```

✅ 위 코드는 실제로는 한 번만 증가할 수 있음 (batching 처리 때문)

📌 이전 상태를 기준으로 업데이트하려면 함수형 업데이트 사용

```
setCount(prev => prev + 1);
setCount(prev => prev + 1);
```

✅ 함수형 업데이트를 사용하면 이전 상태에 안전하게 접근 가능

🎯 5. 조건부 렌더링과 상태 활용

```
const [isVisible, setIsVisible] = useState(true);

return (
  <div>
    <button onClick={() => setIsVisible(!isVisible)}>
      {isVisible ? '숨기기' : '보이기'}
    </button>
    {isVisible && <p>이 내용은 조건부로 보여집니다.</p>}
  </div>
```

);

✅ 상태를 이용해 UI의 조건부 렌더링을 제어할 수 있습니다.

🔁 6. 여러 개의 useState 사용

```
const [email, setEmail] = useState('');
const [password, setPassword] = useState('');
```

✅ 여러 개의 상태를 각자 관리할 수 있으며, **컴포넌트가 복잡해질수록 관리 단위를 나누는 것이 유리**합니다.

또는, 아래와 같이 객체 하나로 통합하여 관리할 수도 있습니다:

```
const [form, setForm] = useState({ email: '', password: '' });

const handleChange = (e) => {
  setForm({ ...form, [e.target.name]: e.target.value });
};
```

🧠 7. 언제 useState를 써야 할까?

- 사용자 입력 값을 추적할 때
- 버튼 클릭 횟수를 셀 때
- 모달, 토글, 드롭다운 등의 열림/닫힘 상태를 관리할 때
- 간단한 카운터, 폼 등의 단순 상태일 때

✅ 복잡한 상태 트리나 여러 상태가 상호작용해야 한다면 `useReducer`를 고려하는 것이 좋습니다.

🏁 8. 마무리

✅ 핵심 요약

- `useState`는 함수형 컴포넌트에서 상태를 선언하고 관리할 수 있게 해주는 기본 훅
- 상태를 변경할 땐 반드시 `setState` 함수를 사용해야 하며, 직접 변경은 무시됨
- 객체나 배열은 항상 **불변성**을 유지하여 업데이트
- 함수형 업데이트 방식은 이전 상태가 필요한 경우 안정적으로 사용 가능

✅ 실무 팁

- 상태가 너무 많아지면 **컴포넌트를 분리하거나 `useReducer`로 전환**하는 것이 유지보수에 유리합니다.
- 초깃값이 복잡한 계산을 포함할 경우, **함수형 초기값**을 사용해 불필요한 연산을 피할 수 있습니다:

const [value, setValue] = useState(() => 복잡한계산());

📌 React Hook: useEffect 완벽 가이드

🚀 1. useEffect란?

📖 개념

- `useEffect`는 **React 함수형 컴포넌트에서 사이드 이펙트(side effect)를 처리하기 위한 Hook**입니다.
- 사이드 이펙트란, 컴포넌트 외부의 작업(ex. API 호출, 이벤트 등록, 타이머 설정 등)으로 인해 발생하는 효과를 말합니다.
- `useEffect`를 사용하면 클래스 컴포넌트의 **`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`** 를 대체할 수 있습니다.

```
useEffect(() => {
  // 실행할 코드
  return () => {
    // 정리(clean-up) 함수
  };
}, [의존성배열]);
```

💡 2. useEffect 기본 사용법

📌 1) 마운트 시 한 번 실행 (componentDidMount 역할)

```
useEffect(() => {
  console.log('컴포넌트가 처음 마운트됨');
}, []);
```

✅ 의존성 배열이 `[]`인 경우, 컴포넌트가 **처음 마운트될 때 딱 한 번만 실행**됩니다.

📌 2) 특정 값이 변경될 때 실행 (componentDidUpdate 역할)

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

✅ `count` 값이 변경될 때마다 이 `useEffect`가 실행됩니다.

📌 3) 언마운트 시 정리 (componentWillUnmount 역할)

```
useEffect(() => {
  const id = setInterval(() => console.log('running...'), 1000);
  return () => clearInterval(id);
}, []);
```

✅ 리턴된 함수는 **컴포넌트가 언마운트되기 직전에 실행**되어 메모리 누수를 방지합니다.

🛠 3. 실전 예제

📌 타이머 예제

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    return () => clearInterval(interval); // 언마운트 시 정리
  }, []);
```

```
  return <p>경과 시간: {seconds}초</p>;
}
```

✅ 마운트 시 타이머 시작, 언마운트 시 정리하는 구조

📌 API 호출 예제

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

function PostList() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(res => res.json())
      .then(data => setPosts(data));
  }, []);

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

✅ 외부 API 호출은 반드시 `useEffect` 내부에서 처리해야 안전합니다.

⚠️ 4. 의존성 배열 주의사항

📌 1) 의존성 배열이 누락된 경우

```
useEffect(() => {
  console.log('실행됨');
});
```

✅ 의존성 배열이 없으면 **렌더링마다 무조건 실행됨** (성능 저하 위험)

📌 2) 의존성 배열에 객체/함수 사용 시 주의

```
useEffect(() => {
  console.log('user 객체가 바뀌었어요');
}, [user]);
```

✅ 객체나 함수는 **매번 새로 생성되기 때문에**, 값이 같아도 다른 것으로 인식됨 → `useMemo`, `useCallback`과 함께 사용 필요

🔄 5. useEffect 실행 순서 이해하기

📌 1) 실행 타이밍

- `useEffect`는 **DOM이 그려진 뒤에 비동기적으로 실행**됨
- 따라서 **브라우저에 영향을 주는 작업(ex. 스크롤 이동, 사이즈 측정 등)** 은 `useLayoutEffect`를 사용해야 함

📌 2) 여러 개의 useEffect

- 한 컴포넌트에 여러 `useEffect`를 선언하면 **작성된 순서대로 실행됨**

```
useEffect(() => console.log('첫 번째'), []);
useEffect(() => console.log('두 번째'), []);
```

✅ 로그 순서: 첫 번째 → 두 번째

🎯 6. useEffect를 사용해야 하는 경우

- API 호출 (GET, POST 등)
- `setTimeout`, `setInterval` 등 비동기 타이머 등록
- 브라우저 이벤트 등록 (scroll, resize 등)
- 외부 라이브러리 초기화 (ex. chart.js, swiper 등)
- 컴포넌트 언마운트 시 정리 필요할 때 (cleanup)

✅ 외부와의 연결/영향이 있는 모든 작업은 useEffect로 감싸는 것이 React의 best practice

---

🧠 7. useEffect 성능 최적화 팁

✅ 불필요한 실행 방지

- 의존성 배열에 꼭 필요한 값만 넣기
- `useCallback`, `useMemo`와 함께 사용하여 불필요한 재실행 방지

✅ 네트워크 요청 시 로딩/에러 처리 함께 구현하기

```
useEffect(() => {
  const fetchData = async () => {
    try {
      setLoading(true);
      const res = await fetch(...);
      const data = await res.json();
      setData(data);
    } catch (e) {
      setError(e);
    } finally {
      setLoading(false);
    }
  };
  fetchData();
```

}, []);

🏁 8. 마무리

✅ 핵심 요약

- `useEffect`는 **사이드 이펙트를 안전하게 다루기 위한 핵심 훅**
- 마운트/업데이트/언마운트 시점을 자유롭게 제어할 수 있음
- 의존성 배열을 통해 **언제 실행할지 명확히 제어**할 수 있음
- **정리(clean-up)** 작업이 필요한 경우 꼭 리턴 함수 작성

📌 React Hook: useRef 완벽 가이드

🚀 1. useRef란?

📖 개념

- `useRef`는 **React에서 DOM 요소에 직접 접근하거나, 리렌더링 없이 값을 저장할 수 있는 Hook**입니다.
- 렌더링 사이에 **값을 유지(persistent)** 하지만, 값이 변경되어도 **컴포넌트를 리렌더링하지 않음**.
- `useRef`는 두 가지 주요 용도로 사용됩니다:
    1. **DOM 참조 접근** (ex. input focus)
    2. **값 기억 저장소로 활용** (ex. 이전 값 저장, 타이머 ID 저장 등)

const myRef = useRef(initialValue);

- `myRef.current`에 접근하여 값을 읽거나 수정합니다.

💡 2. DOM 요소에 접근하기

📌 예제: input 자동 포커스

import { useRef, useEffect } from 'react';

function AutoFocusInput() {

```
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus(); // DOM 요소에 직접 접근
  }, []);

  return <input ref={inputRef} placeholder="자동 포커스" />;
}
```

✅ `ref` 속성을 통해 해당 DOM 요소에 직접 연결하고 `.current`로 접근합니다.

🔁 3. useRef로 값 저장하기

📌 리렌더링 없이 값 변경

```
function Counter() {
  const countRef = useRef(0);

  const increase = () => {
    countRef.current += 1;
    console.log('현재 값:', countRef.current);
  };

  return <button onClick={increase}>증가</button>;
}
```

✅ `useRef`는 값이 바뀌어도 화면에 영향을 주지 않기 때문에 **렌더링 성능에 부담 없이 상태처럼 활용** 가능

🧠 4. useRef vs useState

구분	useRef	useState
렌더링 트리거	❌ 변경해도 렌더링 없음	✅ 변경 시 컴포넌트 리렌더링
값 유지	✅ 유지됨	✅ 유지됨
DOM 접근	✅ 가능 (`ref={...}` 사용)	❌ 불가능

✅ 리렌더링이 필요 없는 값을 저장하거나, DOM 접근이 필요할 때는 `useRef` 사용이 적합합니다.

🛠️ 5. 실전 활용 예제

📌 이전 값 추적하기

```
import { useEffect, useRef, useState } from 'react';


function PreviousValueTracker() {
  const [count, setCount] = useState(0);
  const prevCount = useRef(count);

  useEffect(() => {
    prevCount.current = count;
  }, [count]);

  return (
    <div>
```

```
      <p>현재 값: {count}</p>
      <p>이전 값: {prevCount.current}</p>
      <button onClick={() => setCount(count + 1)}>증가</button>
    </div>
  );
}
```

✅ `useRef`를 사용하면 **이전 값을 상태처럼 추적**할 수 있음 (하지만 렌더링에 영향을 주지 않음)

📌 디바운스 타이머 ID 저장

```
function Search() {
  const timerRef = useRef(null);

  const handleChange = (e) => {
    if (timerRef.current) clearTimeout(timerRef.current);
    timerRef.current = setTimeout(() => {
      console.log('검색 요청:', e.target.value);
    }, 500);
  };

  return <input onChange={handleChange} placeholder="검색어 입력" />;
}
```

✅ `useRef`는 비동기 타이머 ID 저장에도 효과적이며, 메모리 누수를 방지하는 데도 유용

⚠️ 6. 주의사항

- `.current` 값은 변경해도 렌더링이 일어나지 않음 → 상태처럼 사용할 수는 있지만 **UI에 직접 연결되면 반영되지 않음**
- DOM 조작이 많은 코드는 React 철학과 맞지 않음 → 되도록 최소한으로만 DOM 직접 조작

🧩 7. 함께 쓰면 좋은 Hooks

- `useEffect`와 함께 사용 시 **컴포넌트가 마운트된 후 DOM 접근이 안전하게 가능**
- `useCallback`, `useMemo`와 조합하여 **참조값 재사용 또는 타이머 관리 등에서 성능 최적화**

🏁 8. 마무리

✅ 핵심 요약

- `useRef`는 리렌더링 없이 값을 저장하거나 DOM에 직접 접근할 수 있는 Hook입니다.
- 값은 `.current`를 통해 접근하며, 렌더링과는 무관하게 유지됩니다.
- 이전 값 추적, 타이머 ID 저장, input 포커싱 등 다양한 상황에 유용하게 사용됩니다.
- 상태 업데이트가 필요 없는 경우에는 `useRef`가 더 가볍고 효율적인 대안이 됩니다.

📌 React Hook: useReducer 완벽 가이드

🚀 1. useReducer란?

📖 개념

- `useReducer`는 **복잡한 상태 관리 로직을 구조적으로 다루기 위한 Hook**입니다.
- 상태를 업데이트하는 로직을 **reducer 함수**로 분리하여, **액션 기반의 상태 변경 패턴**을 따릅니다.
- `useState`보다 상태 변경이 명확하고 예측 가능하며, 특히 **여러 상태가 연관되어 있거나 조건이 복잡한 경우에 유리**합니다.

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

- `state`: 현재 상태 값

- `dispatch`: 액션을 전달하는 함수
- `reducer`: 상태 업데이트를 정의하는 함수
- `initialState`: 초기 상태 값

💡 2. 기본 구조 및 사용법

📌 기본 예제: 숫자 증가/감소

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

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

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

  return (
    <div>
      <p>카운트: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}
```

✅ `dispatch` 함수에 `type`과 필요 시 `payload`를 포함한 액션 객체를 전달 ✅ 상태 업데이트 로직을 reducer로 외부화하여 **코드가 명확하고 관리가 쉬움**

🛠 3. 복잡한 상태 로직 처리 예시

📌 폼 입력 상태를 객체로 관리하기

```
const formReducer = (state, action) => {
```

```
  switch (action.type) {
    case 'CHANGE_INPUT':
      return {
        ...state,
        [action.name]: action.value,
      };
    case 'RESET':
      return initialForm;
    default:
      return state;
  }
};

const initialForm = { name: '', email: '' };

function MyForm() {
  const [form, dispatch] = useReducer(formReducer, initialForm);

  const onChange = (e) => {
    dispatch({
      type: 'CHANGE_INPUT',
      name: e.target.name,
      value: e.target.value,
    });
  };

  return (
    <form>
      <input name="name" value={form.name} onChange={onChange} />
      <input name="email" value={form.email} onChange={onChange} />
    </form>
  );
}
```

✅ 복잡한 폼 입력 처리 시, 상태를 객체 형태로 관리하고 액션을 통해 구조화된 업데이트가 가능

🔍 4. useState와의 차이점

항목	`useState`	`useReducer`
상태의 복잡도	단순한 상태에 적합	복잡한 로직과 상태에 적합
상태 업데이트 방식	setState로 직접 변경	dispatch와 reducer로 변경

코드 구조	짧고 직관적	명확하고 구조적
추천 케이스	단일 값 또는 간단한 객체	여러 상태 간 관계가 있는 경우, 액션 기반 업데이트

⚙️ 5. useReducer + Context 조합 (전역 상태 관리)

- `useReducer`는 `useContext`와 함께 쓰면 **Redux를 대체할 수 있는 구조로 발전 가능**

```
const StateContext = createContext();
const DispatchContext = createContext();

function AppProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <StateContext.Provider value={state}>
      <DispatchContext.Provider value={dispatch}>
        {children}
      </DispatchContext.Provider>
    </StateContext.Provider>
  );
}
```

✅ 전역 상태 공유 및 변경을 커스터마이징된 Redux처럼 구현 가능

🧠 6. 실무 활용 팁

✅ action에 payload 전달하기

```
dispatch({ type: 'ADD_TODO', payload: { id: 1, text: '공부하기' } });
```

- reducer에서 `action.payload` 사용 가능

✅ 타입 상수화 (enum 활용)

```
const ACTIONS = {
  ADD: 'add',
  REMOVE: 'remove',
};
```

- 매직 문자열 대신 enum처럼 상수 사용으로 오류 방지

🏁 7. 마무리

✅ 핵심 요약

- `useReducer`는 복잡한 상태 업데이트를 함수형으로 분리하여 구조화된 코드 작성이 가능함
- `dispatch`를 통해 액션을 전달하고, `reducer` 함수가 상태를 업데이트함
- `useState`보다 복잡한 상태 로직에서 명확하고 유지보수성이 뛰어남
- `useContext`와 결합하면 전역 상태 관리도 가능

📌 React Hook: useContext 완벽 가이드

🚀 1. useContext란?

📖 개념

- `useContext`는 **컴포넌트 트리 전체에 전역 데이터를 쉽게 전달할 수 있도록 하는 React의 Hook**입니다.
- 부모 컴포넌트에서 하위 컴포넌트로 props를 계속 넘겨주는 **props drilling** 문제를 해결하기 위해 사용합니다.
- React의 **Context API**를 사용하며, 전역 상태, 테마, 언어 설정, 인증 정보 등 다양한 전역 데이터를 관리할 수 있습니다.

const value = useContext(MyContext);

- `MyContext`는 `createContext()`로 생성된 객체여야 합니다.

💡 2. 기본 사용법

📌 1) Context 생성 및 Provider 설정

import { createContext, useContext } from 'react';

// Context 생성

const ThemeContext = createContext('light');

```
// Provider로 감싸기
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

📌 2) 자식 컴포넌트에서 사용하기

```
function Toolbar() {
  return <ThemeButton />;
}

function ThemeButton() {
  const theme = useContext(ThemeContext);
  return <button>현재 테마: {theme}</button>;
}
```

✅ `useContext`를 사용하면 중간 컴포넌트를 거치지 않고 직접 데이터에 접근할 수 있음

🧩 3. 실전 예제: 로그인 정보 전역 상태 공유하기

📌 AuthContext 만들기

```
const AuthContext = createContext();

function AuthProvider({ children }) {
  const user = { name: '홍길동', isLoggedIn: true };
  return (
    <AuthContext.Provider value={user}>
```

```
      {children}
    </AuthContext.Provider>
  );
}
```

📌 전역 상태 사용하기

```
function Profile() {
  const user = useContext(AuthContext);
  return <h2>{user.isLoggedIn ? `${user.name}님 환영합니다!` : '로그인 필요'}</h2>;
}
```

✅ 로그인 정보, 테마, 언어 등 다양한 공통 데이터를 전역에서 간편하게 가져올 수 있음

🔄 4. useContext와 상태 변경

`useContext` 자체는 상태를 변경하는 기능은 없으며, 전역 상태를 단순히 **구독하고 사용하는 역할**입니다.

📌 상태를 함께 관리하려면 `useReducer` 또는 `useState`와 결합

```
const CountContext = createContext();

function CountProvider({ children }) {
  const [count, setCount] = useState(0);
  return (
    <CountContext.Provider value={{ count, setCount }}>
      {children}
    </CountContext.Provider>
  );
}

function Counter() {
  const { count, setCount } = useContext(CountContext);
  return (
    <>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
    </>
  );
```

}

✅ 전역에서 `useState`나 `useReducer`와 함께 쓰면 Context의 확장성이 높아짐

⚙️ 5. useContext의 장점과 단점

✅ 장점

1. **props drilling 제거** → 컴포넌트 간 계층이 깊어도 직접 접근 가능
2. **전역 상태 공유 용이** → 테마, 로그인 상태, 언어 설정 등에 적합
3. **컴포넌트 간 강한 결합 없이 정보 공유** → 유지보수 용이

⚠️ 단점

1. **남용하면 렌더링 성능 저하** → 모든 Context 소비 컴포넌트가 Provider 변경에 반응
2. **복잡한 상태 로직에는 적합하지 않음** → `useReducer`나 Redux를 함께 사용 권장

🏁 6. 마무리

✅ 핵심 요약

- `useContext`는 Context의 소비자 역할을 수행하여 전역 데이터를 쉽게 가져올 수 있도록 해줌
- 컴포넌트 간 데이터 전달을 간결하게 만들어주며, `useState`, `useReducer`와 함께 쓰면 상태 관리까지 가능
- **React 전역 상태 관리의 기본**이자, 프로젝트 규모가 커질수록 필수적인 패턴

📌 React Hook: useMemo 완벽 가이드

🚀 1. useMemo란?

📖 개념

- `useMemo`는 **비싼 연산 비용이 드는 계산 결과를 메모이제이션(memoization)하여 성능을 최적화하는 Hook**입니다.
- 의존성 배열의 값이 변경되지 않는 한, 이전에 계산된 값을 재사용합니다.
- 컴포넌트가 리렌더링될 때 **매번 다시 계산하는 것을 방지**하고, **불필요한 계산으로 인한 성능 저하를 줄이는 데 유용**합니다.

const memoizedValue = useMemo(() => expensiveFunction(a, b), [a, b]);

- `expensiveFunction`: 연산 비용이 큰 함수
- `[a, b]`: 의존성 배열. 값이 변경될 때만 다시 계산

💡 2. 언제 사용해야 할까?

✅ 다음과 같은 경우 사용을 고려하세요:

1. **복잡한 계산 함수**를 리렌더링마다 반복 호출하는 경우
2. **렌더링 성능이 중요한 컴포넌트**에 데이터를 가공하여 전달할 때
3. **Props로 전달하는 객체, 배열** 등이 자주 새로 생성되어 불필요한 렌더링이 발생할 때

❗ 과도한 사용은 오히려 성능 저하를 유발할 수 있으므로, 연산 비용이 클 때만 사용하는 것이 원칙입니다.

🔍 3. 기본 사용 예제

📌 계산 비용이 큰 함수 메모이제이션

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

function ExpensiveComponent({ number }) {
  const slowFunction = (num) => {
    console.log('무거운 계산 중...');
    for (let i = 0; i < 1e9; i++) {} // 시간 지연
    return num * 2;
  };

  const doubled = useMemo(() => slowFunction(number), [number]);

  return <p>결과: {doubled}</p>;
}
```

✅ `number`가 바뀌지 않으면 `slowFunction`은 다시 실행되지 않음

🛠 4. 객체나 배열을 메모이제이션할 때

📌 객체를 메모이제이션하지 않으면?

const config = { theme: 'dark' }; // 매 렌더링마다 새 객체 생성됨

✅ 이 경우 하위 컴포넌트에 전달 시 매번 변경으로 인식되어 **불필요한 렌더링 유발**

📌 useMemo로 최적화

const config = useMemo(() => ({ theme: 'dark' }), []);

✅ 동일한 객체 참조를 유지하므로, 불필요한 렌더링 방지 가능

📦 5. useMemo vs useCallback

Hook	목적	반환값	사용처
`useMemo`	값을 메모이제이션	**값**	계산 결과, 객체, 배열
`useCallback`	함수를 메모이제이션	**함수**	이벤트 핸들러, 콜백 함수

✅ 공통점: 의존성 배열 기반 메모이제이션 ✅ 차이점: 반환값의 종류 (값 vs 함수)

🧠 6. 실전 예제: 정렬된 리스트 캐싱

```
function SortedList({ items }) {
  const sortedItems = useMemo(() => {
    console.log('정렬 실행');
    return [...items].sort();
  }, [items]);

  return (
    <ul>
      {sortedItems.map((item, index) => <li key={index}>{item}</li>)}
```

```
    </ul>
  );
}
```

✅ `items`가 변경되지 않으면 정렬 연산을 다시 하지 않음

⚠️ 7. 주의사항

❗ 과용하지 말 것

- `useMemo` 자체도 오버헤드가 있기 때문에, 단순한 연산에는 사용하지 않는 것이 좋습니다.

❗ 참조 불변성을 활용해야 함

- 리렌더링마다 생성되는 객체/배열은 참조값이 달라짐 → `useMemo`로 관리

🏁 8. 마무리

✅ 핵심 요약

- `useMemo`는 복잡하거나 비용이 큰 계산 결과를 캐싱하여 리렌더링 시 성능을 최적화하는 Hook
- 의존성 배열이 변경될 때만 함수를 실행하고, 그렇지 않으면 이전 결과를 재사용
- React 성능 최적화의 핵심 도구 중 하나이며, 필요한 경우에만 적절히 사용하는 것이 중요

📌 React Hook: useCallback 완벽 가이드

🚀 1. useCallback이란?

📖 개념

- `useCallback`은 **함수를 메모이제이션(memoization)하여 동일한 함수 참조를 유지시키는 Hook**입니다.

- 컴포넌트가 리렌더링될 때마다 **함수가 새로 생성되는 것을 방지**하여, 불필요한 렌더링이나 의도치 않은 동작을 줄일 수 있습니다.
- 주로 **자식 컴포넌트에 콜백 함수를 props로 전달할 때**, 또는 **의존성 배열로 함수를 사용해야 할 때** 유용합니다.

```
const memoizedCallback = useCallback(() => {
  // 함수 내용
}, [dependencies]);
```

- `dependencies` 배열의 값이 변경되지 않으면, 이전에 생성된 함수를 그대로 재사용

💡 2. 왜 필요한가요?

✅ 기본적으로 함수는 렌더링마다 새로 만들어집니다.

```
function Parent() {
  const handleClick = () => {
    console.log('클릭!');
  };
  return <Child onClick={handleClick} />;
}
```

✅ `Parent`가 리렌더링될 때마다 `handleClick`은 새로운 함수로 간주됩니다.

이로 인해:

- `React.memo()`로 감싼 자식 컴포넌트도 불필요하게 렌더링됨
- `useEffect`의 의존성 배열에서 함수가 항상 변경됨으로 감지됨

🛠️ 3. 기본 사용 예제

📌 부모 → 자식으로 콜백 함수 전달

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

const Child = React.memo(({ onClick }) => {
  console.log('Child 렌더링');
  return <button onClick={onClick}>클릭</button>;
});

function Parent() {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
```

```
    console.log('자식 버튼 클릭');
  }, []); // 의존성 없음 → 고정된 함수 유지

  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>부모 증가</button>
      <Child onClick={handleClick} />
    </div>
  );
}
```

✅ `Child`는 props로 받은 함수의 참조가 유지되기 때문에 `React.memo`가 제대로 작동하여 불필요한 렌더링을 방지함

🧠 4. useCallback vs useMemo

Hook	목적	반환 값	주로 사용하는 상황
`useCallback`	**함수 메모이제이션**	함수	이벤트 핸들러, 콜백, props 전달 등
`useMemo`	**값 메모이제이션**	계산된 값	비싼 연산 결과, 객체/배열 유지 등

✅ 핵심 차이: **useMemo는 값, useCallback은 함수**를 반환합니다.

🔍 5. 실전 예제: 의존성 배열 활용

```
function App() {
  const [value, setValue] = useState(0);

  const printValue = useCallback(() => {
    console.log(`현재 값: ${value}`);
  }, [value]); // value가 바뀔 때만 함수 갱신

  useEffect(() => {
    printValue();
  }, [printValue]);

  return <button onClick={() => setValue(v => v + 1)}>+</button>;
}
```

✅ 함수 내부에서 상태나 props를 사용한다면 반드시 의존성 배열에 포함해야 함

⚠️ 6. 주의사항

❗ 과도한 useCallback 사용은 오히려 성능 저하

- 메모이제이션 자체도 비용이 들기 때문에, **불필요하게 사용하지 말고** 진짜 렌더링 최적화가 필요한 경우에만 사용

❗ 의존성 배열 누락 주의

- 내부에서 사용하는 값은 반드시 의존성 배열에 명시해야 함 → 그렇지 않으면 오래된 값을 참조하게 됨

🏁 7. 마무리

✅ 핵심 요약

- `useCallback`은 함수의 재생성을 방지하여 불필요한 렌더링을 줄이는 데 유용한 Hook입니다.
- 자식 컴포넌트에 함수를 props로 넘길 때, 의존성 배열 내에서 함수가 변경되면 안 되는 경우에 사용
- `useMemo`와 함께 React 성능 최적화에 핵심적인 역할을 합니다.