

# 리액트 스터디 5강

캡스 39기 송윤석

# 강의 전체 로드맵

- 1강: 스터디 OT, 개발환경 구축, HTML/CSS ✓
- 2강: Javascript, Typescript ✓
- 3강: React의 세계관, Electron 개념, 프로젝트 생성 ✓
- 4강: 컴포넌트, 상태관리 ✓
- 5강: 훅, 전역상태관리 <- 오늘
- 6강: 라우팅, 라이브러리, 외부 API, 배포

# 목차 및 오늘의 목표

- 5.1 useEffect 사용하기
- 5.2 fetch로 외부 API 호출
- 5.3 커스텀 훅으로 로딩/에러 상태 다루기 (isLoading, error)
- 5.4 useRef로 DOM 참조하기
- 5.5 전역 상태 관리의 필요성
- 5.6 Zustand 소개 및 설치
- 5.7 Zustand로 데이터 전역 관리

# 5.1 useEffect 사용하기

- 리액트의 리렌더링
  - 리액트 컴포넌트의 상태(state)나 props가 바뀔 때마다 다시 그려지는(리렌더링) 과정
- 그런데 리렌더링 될 때마다 무조건 모든 코드가 다시 실행돼요.
  - 반복해서 실행되면 안 되는 작업(API 호출, 이벤트 리스너 등록 등)은 따로 분리해야 해요.

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

function Child() {
  console.log('Child 컴포넌트 리렌더링!');
  return <div>나는 프롭 없는 자식 컴포넌트예요.</div>;
}

function ChildWithProp({ message }: { message: string }) {
  console.log('ChildWithProp 컴포넌트 리렌더링!');
  return <div>부모가 준 메시지: {message}</div>;
}

export default function App() {
  const [count, setCount] = useState(0);
  console.log('Parent 컴포넌트 리렌더링! count:', count);
  return (
    <div>
      <h1>카운터: {count}</h1>
      <button onClick={() => setCount(count + 1)}>+1</button>
      <Child />
      <ChildWithProp message={`현재 카운트는 ${count}입니다`} />
    </div>
  );
}
```

# 5.1 useEffect 사용하기

- 리액트의 useEffect

- useEffect를 사용하면 원하는 시점에만 코드를 실행하도록 분리할 수 있습니다.
- 불필요한 반복 실행에서 자유로워지고, 효율적인 코드 관리가 가능해집니다.

```
useEffect(() => {  
  // 원하는 로직  
}, [/** 의존성 배열 */]);
```

# 5.1 useEffect 사용하기

## • 사용 방법

- useEffect 안에는 함수를 넣을 수 있습니다.
- 화살표 함수 안에 원하는 로직을 작성하면 됩니다.
- 연관된 **상태나 Props**가 있다면 두 번째 인자인 **의존성 배열**에 넣어 실행 시점을 제어할 수 있습니다.

```
useEffect(() => {  
  console.log(  
    'Parent 컴포넌트의 useEffect 안!'  
  );  
}, []);  
  
useEffect(() => {  
  console.log(  
    'Parent 컴포넌트의 useEffect 안! count:',  
    count  
  );  
}, [count]);
```

# 5.1 useEffect 사용하기

- **useEffect 안 코드는 언제 실행될까?**
  - 컴포넌트가 처음 화면에 나타날 때 (마운트 시점)
  - 의존성 배열의 상태나 Props가 바뀔 때마다

```
useEffect(() => {  
  console.log(  
    'Parent 컴포넌트의 useEffect 안!'  
  );  
}, []);  
  
useEffect(() => {  
  console.log(  
    'Parent 컴포넌트의 useEffect 안! count:',  
    count  
  );  
}, [count]);
```

# 5.1 useEffect 사용하기

- 클린업 문법이란?
  - useEffect 안에서 return 문으로 함수를 반환하는 문법.
  - 이 반환 함수는 **컴포넌트가 사라질 때(언마운트)** 또는 다음 effect가 실행되기 전에 **이전 작업을 정리(cleanup) 하는 역할**

```
useEffect(() => {  
  // 내부 동작  
  return () => {  
    // 정리 함수  
  };  
}, []);
```



# 5.1 useEffect 사용하기

- 예시로 이해하기

- 3초 후 알림을 보여주는 타이머
- count 상태가 바뀌어서 컴포넌트가 여러 번 리렌더링 되어도, useEffect 안 타이머 코드는 다시 실행되지 않음
- 의존성 배열이 빈 배열 [] 이기 때문에 마운트 시점에만 실행
- 타이머가 3초 후에만 실행되고, 컴포넌트가 사라지거나 다시 실행되기 전에 클린업 함수가 타이머를 제거하여 타이머 중복 실행이나 충돌을 방지

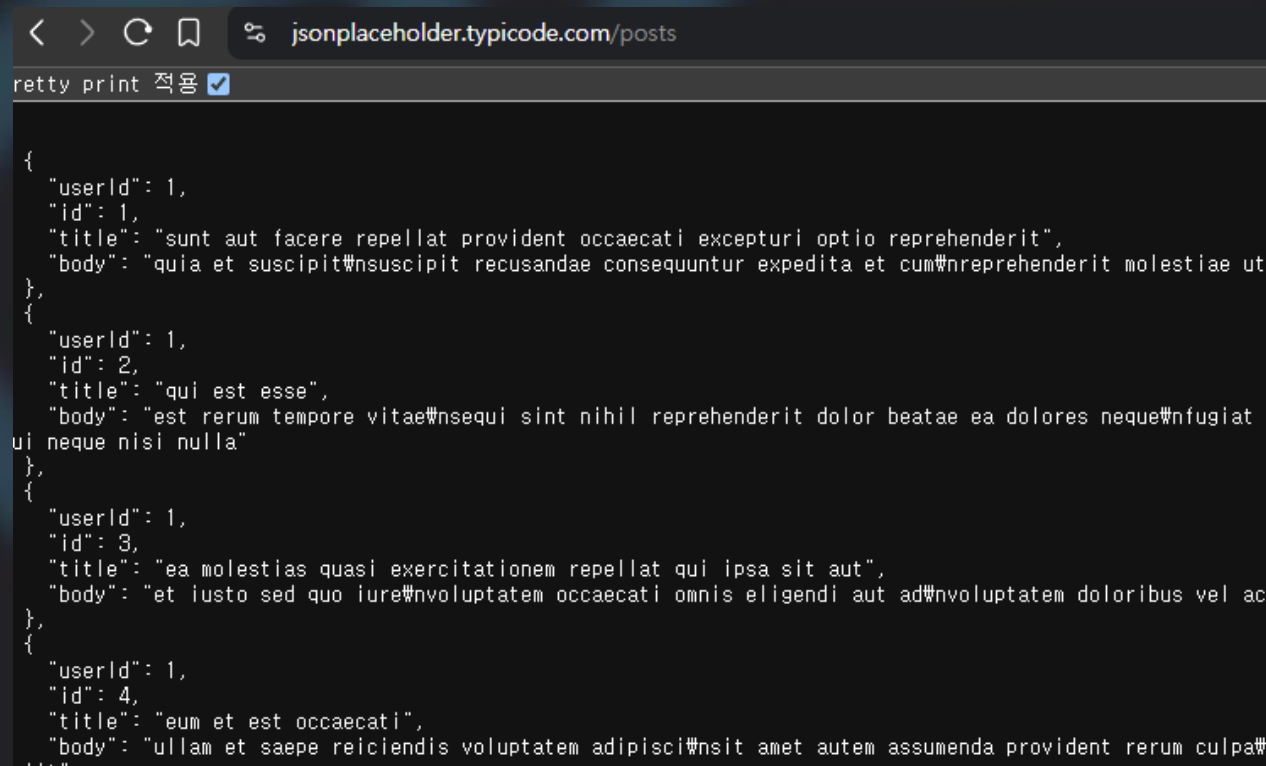
```
useEffect(() => {  
  const timerId = setTimeout(() => {  
    alert('3초가 지났어요!');  
  }, 3000);  
  
  return () => {  
    clearTimeout(timerId);  
    console.log('타이머가 정리되었습니다.');  };  
}, []);
```

## 5.2 fetch로 외부 API 호출

- Fetch 함수를 통해 외부 서버로 요청을 날릴 수 있음
  - <https://jsonplaceholder.typicode.com/posts>
  - 더미 데이터를 제공하는 사이트

```
const [data, setData] = useState(null);

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



```
< > ↺ 📄 jsonplaceholder.typicode.com/posts
retty print 적용 ☒

{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit#nsuscipit recusandae consequuntur expedita et cum#nreprehenderit molestiae ut
},
{
  "userId": 1,
  "id": 2,
  "title": "qui est esse",
  "body": "est rerum tempore vitae#nsequi sint nihil reprehenderit dolor beatae ea dolores neque#nfugiat
  neque nisi nulla"
},
{
  "userId": 1,
  "id": 3,
  "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut",
  "body": "et iusto sed quo iure#nvoluptatem occaecati omnis eligendi aut ad#nvoluptatem doloribus vel ac
},
{
  "userId": 1,
  "id": 4,
  "title": "eum et est occaecati",
  "body": "ullam et saepe reiciendis voluptatem adipisci#nsit amet autem assumenda provident rerum culpa#
..."
```

## 5.2 fetch로 외부 API 호출

- 일반적인 개발 흐름
  1. API 응답 타입 정의
  2. API 호출
  3. (선택) 데이터를 가공
  4. 화면에 표시

```
interface Post {  
  userId: number;  
  id: number;  
  title: string;  
  body: string;  
}
```

## 5.2 fetch로 외부 API 호출

- 일반적인 개발 흐름

1. API 응답 타입 정의
2. API 호출
3. (선택) 데이터를 가공
4. 화면에 표시

```
const [data, setData] = useState<Post[] | null>(null);

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

## 5.2 fetch로 외부 API 호출

- 일반적인 개발 흐름

1. API 응답 타입 정의
2. API 호출
3. (선택) 데이터를 가공
4. 화면에 표시

fetch 테스트

title: sunt aut facere repellat provident occaecati excepturi optio reprehenderit

body: quia et suscipit suscipit recusandae consequuntur expedita et cum reprehenderit mollit

title: qui est esse

body: est rerum tempore vitae sequi sint nihil reprehenderit dolor beatae ea dolores neque  
qui neque nisi nulla

title: ea molestias quasi exercitationem repellat qui ipsa sit aut

body: et iusto sed quo iure voluptatem occaecati omnis eligendi aut ad voluptatem doloribus

title: cum et est occaecati

```
return (  
  <div>  
    <h1>fetch 테스트</h1>  
    <ul>  
      {data && data.map((item) => (  
        <li key={item.id}>  
          <h2>title: {item.title}</h2>  
          <p>body: {item.body}</p>  
        </li>  
      )}}  
    </ul>  
  </div>  
);
```

## 5.3 커스텀 훅으로 로딩/에러 상태 다루기

- 리액트 훅이란?

- 리액트 컴포넌트에서 상태 관리 (`useState`), 생명주기 관리(`useEffect`) 등 여러 기능을 쉽게 쓸 수 있게 해주는 특별한 함수

- 커스텀 훅이란?

- 리액트 훅을 직접 만들어 재사용할 수 있도록 감싼 함수
- 공통 로직을 분리해서 여러 컴포넌트에서 간편하게 재사용할 수 있게 해줌
- 이름이 `use`로 시작하는 것이 관례이며 그래야 리액트가 훅으로 인식

## 5.3 커스텀 훅으로 로딩/에러 상태 다루기

- 방금 만든 API 요청의 경우
  - useState, useEffect로 동작
  - 이 페이지 뿐만 아니라 다른 페이지에서도 쓰일 수 있음
  - 훅으로 만들면 사용성이 올라감

```
const [data, setData] = useState(null);

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

## 5.3 커스텀 훅으로 로딩/에러 상태 다루기

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

interface Post {
  userId: number;
  id: number;
  title: string;
  body: string;
}

function useFetchPosts(url: string) {
  const [data, setData] = useState<Post[] | null>(null);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);
```

```
  useEffect(() => {
    setLoading(true);
    setError(null);

    fetch(url)
      .then(response => {
        if (!response.ok) {
          throw new Error('네트워크 응답이 좋지 않습니다');
        }
        return response.json();
      })
      .then(data => {
        setData(data);
        setLoading(false);
      })
      .catch((err: Error) => {
        setError(err.message);
        setLoading(false);
      });
  }, [url]);

  return { data, loading, error };
}

export default useFetchPosts;
```



## 5.3 커스텀 훅으로 로딩/에러 상태 다루기

- 커스텀 훅에서 넘겨주는 로딩, 에러 상태도 활용
- 재사용성이 높아짐
- 컴포넌트내부도 깔끔해짐

```
import useFetchPosts from './hooks/useFetchPosts';

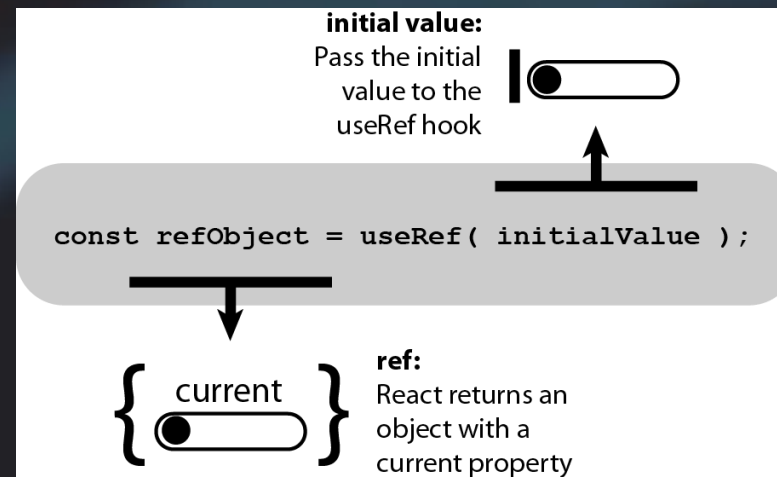
export default function PostList() {
  const { data, loading, error } =
    useFetchPosts('https://jsonplaceholder.typicode.com/posts');

  if (loading) return <p>로딩중...</p>;
  if (error) return <p>에러 발생: {error}</p>;
  if (!data) return <p>데이터가 없습니다.</p>;

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

## 5.4 useRef로 DOM 참조하기

- `useRef`는 리액트 훅 중 하나로, 변경 가능한 값을 저장하는 상자(box) 역할
- 이 값은 컴포넌트가 리렌더링되어도 유지
- 하지만 `useState`와 달리 값을 변경해도 리렌더링이 발생하지 않음



## 5.4 useRef로 DOM 참조하기

- **useRef의 주요 용도**

- 1. DOM 요소 직접 참조하기**

- 특정 DOM에 직접 접근하고 싶을 때 사용
    - Input 요소에 포커스를 주거나, 모달 외부 클릭 감지할 때

- 2. 렌더링과 무관한 값 저장하기**

- 렌더링 과정에 영향을 주지 않는 값을 기억하고 싶을 때 사용
    - 타이머의 ID나 이전 상태값 등

## 5.4 useRef로 DOM 참조하기

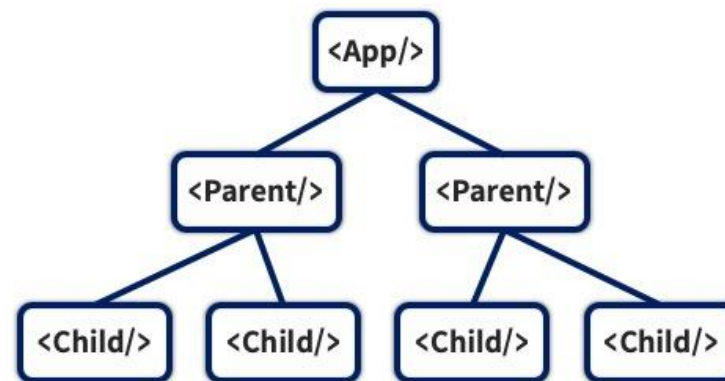
- DOM 조작 용도로 많이 쓰기 때문에...
- 간단한 예시로 알아보기
- 변수로 선언 후 useState처럼 초기값 설정
- 원하는 DOM 요소(HTML 요소)에 ref={변수명} 형태로 매핑

```
export default function InputFocus() {  
  const inputRef = useRef<HTMLInputElement>(null);  
  
  const handleFocus = () => {  
    inputRef.current?.focus();  
  };  
  
  return (  
    <>  
      <input ref={inputRef} type="text" />  
      <button onClick={handleFocus}>포커스 주기</button>  
    </>  
  );  
}
```

## 5.5 전역 상태 관리의 필요성

- 리액트 개발을 하게 되면 컴포넌트 안에 컴포넌트 안에 컴포넌트....
  - 컴포넌트가 중첩이 된다!
  - 이때 상태(state)들 또한 중첩이 된다.
- 만약, App의 어떤 상태를 Child가 필요로 한다면?
  - Parent를 통해 Props로 내려주어야 함(Props Drill)
  - 10단계 아래 컴포넌트라면?
  - 10번 Props로 내려주기?

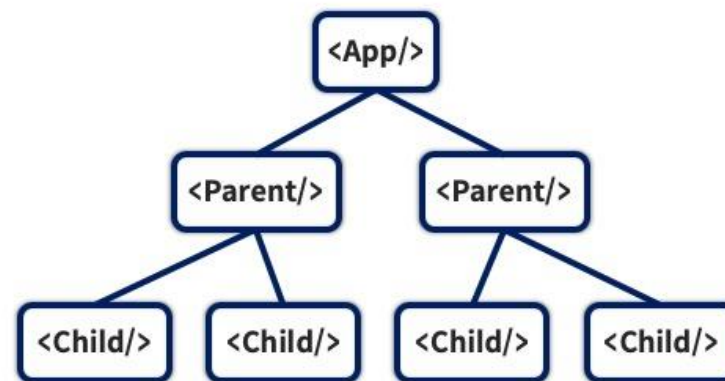
### The React Render Tree



## 5.5 전역 상태 관리의 필요성

- 해결책: 전역 상태 관리
  - 상태를 컴포넌트 바깥, 공용 저장소에 보관해서
  - 필요한 컴포넌트들이 직접 접근하도록 하는 방법입니다.
  - 이렇게 하면 불필요한 props 전달을 줄일 수 있고 코드도 깔끔

### The React Render Tree



## 5.5 전역 상태 관리의 필요성

- 전역 상태 관리가 유용한 경우
- 여러 컴포넌트에서 필요한 공통 정보
- 예) 로그인 정보, 사용자 프로필, 테마 설정(다크 모드 등)
- 페이지 이동이나 컴포넌트 전환 시에도 유지되어야 하는 정보
- 예) 알림 메시지, 장바구니 아이템, 선택한 필터 옵션

## 5.6 Zustand 소개 및 설치

- Zustand란?

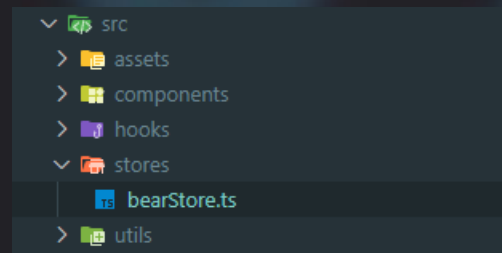
- React용 아주 가볍고 간단한 전역 상태 관리 라이브러리
- 별도 Provider 없이 어디서든 쉽게 상태를 읽고 수정 가능
- 코드가 간결하고 배우기 쉬워 초보자에게도 접근성이 좋음
- 요새 많은 프로젝트가 이 라이브러리를 채택 중





## 5.6 Zustand 소개 및 설치

- Zustand 설치하기
- `npm i zustand`

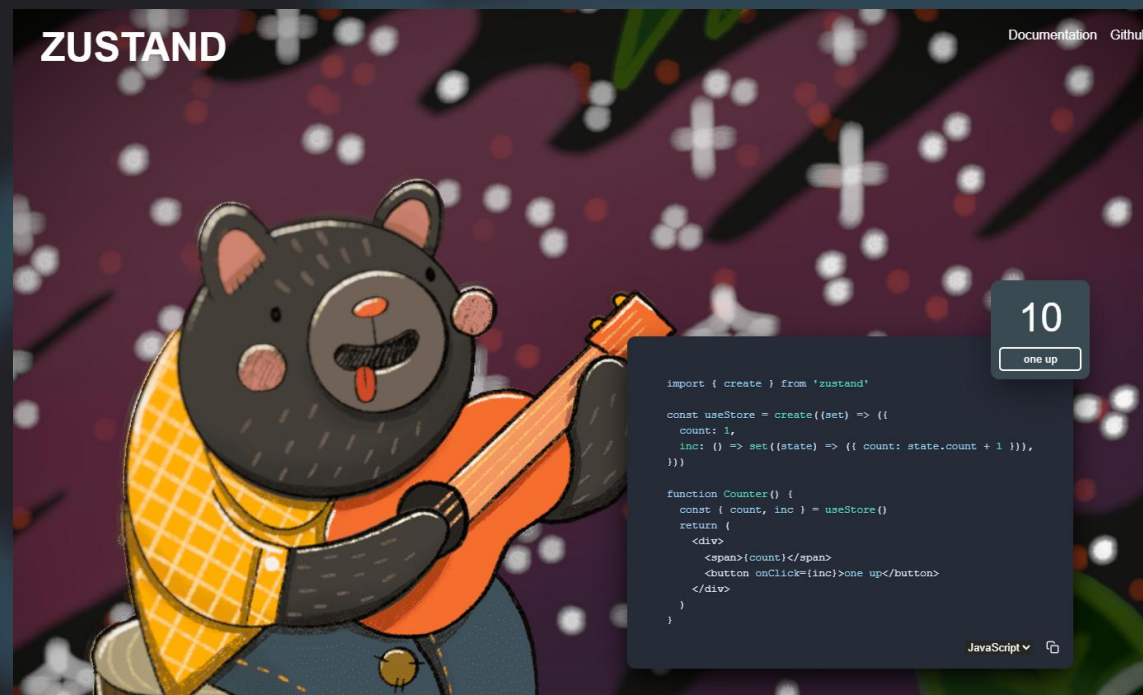


# 5.7 Zustand로 데이터 전역 관리

- Zustand의 기본 문법

- <https://zustand-demo.pmnd.rs/>
- 데모 사이트 코드에서 볼 수 있듯, create로 저장소 생성
- useState와 굉장히 비슷
- 상태 + 설정 함수의 조합

```
const useStore = create((set) => ({
  count: 1,
  inc: () => set((state) => ({
    count: state.count + 1
  })),
})))
```



## 5.7 Zustand로 데이터 전역 관리

- 저장소 만들기 실습

```
import { create } from "zustand";

interface BearState {
  bears: number;
  addBear: () => void;
  removeBear: () => void;
}

export const useBearStore = create<BearState>((set) => ({
  bears: 0,
  addBear: () => set((state) => ({ bears: state.bears + 1 })),
  removeBear: () => set((state) => ({ bears: state.bears - 1 })),
}));
```

- 컴포넌트에서 사용하기

```
import { useBearStore } from "../stores/bearStore";

export default function BearCounter() {
  const bears = useBearStore((state) => state.bears);
  const addBear = useBearStore((state) => state.addBear);
  const removeBear = useBearStore((state) => state.removeBear);

  return (
    <div>
      <h1>곰 수: {bears}</h1>
      <button onClick={addBear}>곰 추가</button>
      <button onClick={removeBear}>곰 제거</button>
    </div>
  );
}
```

## 5.7 Zustand로 데이터 전역 관리

- 구조 분해 할당보다 `state => state.count1` 이 좋은점
- 구조 분해 할당시에는 사용하지 않은 Zustand Store의 상태가 바뀌어도 컴포넌트가 리렌더링
- 하지만 `state => state.count1` 시 리렌더링이 발생하지 않음

## 5.7 Zustand로 데이터 전역 관리

```
import { create } from "zustand";
interface StoreState {
  count1: number;
  count2: number;
  inc1: () => void;
  inc2: () => void;
}
const useStore = create<StoreState>((set) => ({
  count1: 0,
  count2: 0,
  inc1: () => set((state) => ({ count1: state.count1 + 1 })),
  inc2: () => set((state) => ({ count2: state.count2 + 1 })),
}));
// 구조분해 방식 (모든 상태 변경 시 리렌더)
function AllStateButton() {
  console.log("📦 AllStateButton 리렌더");
  const { count1, inc1 } = useStore();
  return (
    <div>
      <span>{count1}</span>
      <button onClick={inc1}>Count1 증가</button>
    </div>
  );
}
```

```
// 선택 구독 방식
function SelectStateButton() {
  console.log("📦 SelectStateButton 리렌더");
  const count2 = useStore((state) => state.count2);
  const inc2 = useStore((state) => state.inc2);
  return (
    <div>
      <span>{count2}</span>
      <button onClick={inc2}>Count2 증가</button>
    </div>
  );
}

export default function App() {
  return (
    <div>
      <AllStateButton />
      <SelectStateButton />
    </div>
  );
}
```

# 오늘은 여기까지

- 오늘 배운 것
  - useEffect로 부수효과 관리
  - fetch로 외부 API 호출하기
  - 커스텀훅 만들기
  - 전역 상태의 필요성
  - 전역 상태 라이브러리 사용
- 실습 코드
  - 깃허브에 업로드 했으니 직접 결과 확인을 원하면 편하게 사용하세요
  - <https://github.com/karpitony/caps-react-study/tree/main/code>
- 수고 많으셨습니다!