Context API와 상태 관리 라이브러리

상태 관리 라이브러리는 왜 필요한가?

1. 왜 전역 상태 관리가 필요한가?

React의 상태는 기본적으로 로컬(state) + 단방향 흐름이다.

- React의 기본 상태 흐름은 부모 → 자식(props) 이다
- 멀리 떨어진 컴포넌트들이 같은 상태를 공유해야 할 때
- props drilling 발생
- 구조 복잡도 급증
- 따라서 여러 위치에서 접근 가능한 전역 상태가 필요해진다

2. Context API는 무엇이고, 왜 한계가 있는가?

“전역 상태 관리 도구”가 아니라
“전역 값 전달 메커니즘”

Context가 잘하는 것

- props drilling 제거
- 설정 값 전달 (theme, locale, auth client 등)
- 구조상 멀리 있는 컴포넌트에 값 전달

Context는 ‘값 전체’를 구독한다

2. Context API는 무엇이고, 왜 한계가 있는가?

memo, useMemo, useCallback으로 해결되지 않는다

=> Context를 사용하는 컴포넌트는 memo 되어 있어도 Context가 바뀌면 리렌더링 된다.

Context API가 적합한 경우

- 거의 변하지 않는 값 (리렌더링 비용이 문제 되지 않음)
- 설정성 데이터 (theme, locale 등)
- 앱 환경 정보

Context API

count: 1
mode: light
static

```
Provider render
Provider render
Counter render
Counter render
ModeViewer render
ModeViewer render
```

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

const AppContext = createContext(null);

function AppProvider({ children }) {
  const [count, setCount] = useState(0);
  const [mode, setMode] = useState("light");

  console.log("🟡 Provider render");

  return (
    <AppContext.Provider value={{ count, setCount, mode }}>
      {children}
    </AppContext.Provider>
  );
}

function Counter() {
  const { count, setCount } = useContext(AppContext);
  console.log("🔴 Counter render");

  return (
    <button
      onClick={() => setCount((c) => c + 1)}
      className="bg-amber-200 border-1 rounded-4xl p-1"
    >
      count: {count}
    </button>
  );
}
```

```jsx
const ModeViewer = React.memo(() => {
  const { mode } = useContext(AppContext);
  console.log("🔵 ModeViewer render");

  return <div>mode: {mode}</div>;
});

function StaticBox() {
  console.log("⚪ StaticBox render");
  return <div>static</div>;
}

export default function ContextPage() {
  console.log("📄 ContextPage render");

  return (
    <AppProvider>
      <Counter />
      <ModeViewer />
      <StaticBox />
    </AppProvider>
  );
}
```

3. 전역 상태 관리 라이브러리 등장

“전역 상태를 ‘부분 구독’ 가능하게 만들자”

필요한 상태만 구독

필요한 컴포넌트만 리렌더링

상태 변경의 파급 범위 제어

4. Zustand

중앙 Store 기반

selector로 필요한 상태만 구독

```
const count = useStore((s) => s.count);
```

4. Zustand 장단점

구조가 명확

상태 흐름 추적 쉬움

실무 친화적

팀 프로젝트에 유리

selector를 잘못 쓰면 전체 구독 위험

설계 규칙 필요

Zustand

count: 1
mode: light

🔴 Counter render
🔴 Counter render

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

const useStore = create((set) => ({
  count: 0,
  increase: () => set((s) => ({ count: s.count + 1 })),
}));

function Counter() {
  const count = useStore((s) => s.count);
  const increase = useStore((s) => s.increase);

  console.log("🔴 Counter render");
  return (
    <button
      onClick={increase}
      className="🟨 bg-amber-200 border-1 rounded-4xl p-1"
    >
      count: {count}
    </button>
  );
}
```

```tsx
function ModeViewer() {
  console.log("🔵 ModeViewer render");
  return <div>mode: light</div>;
}

export default function ZustandPage() {
  console.log("📄 ZustandPage render");

  return (
    <>
      <Counter />
      <ModeViewer />
    </>
  );
}
```

5. Jotai

상태 = atom

atom 단위로 완전 분리

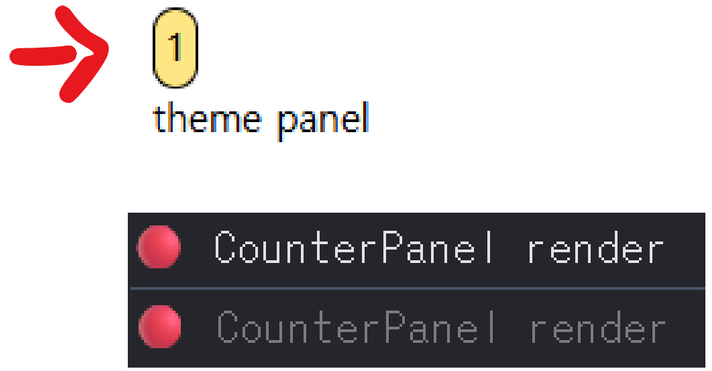
```
const counterAtom = atom(0);
const [count] = useAtom(counterAtom);
```

5. Jotai 장단점

- 리렌더링 안정성 매우 높음
- 상태 분해가 자연스러움
- 파생 상태 선언이 강력
- 개인/소규모 프로젝트에 적합
- atom이 많아지면 관리 어려움
- 상태 흐름 파악이 직관적이지 않을 수 있음

Jotai

```tsx
import { atom, useAtom } from "jotai";

const counterAtom = atom(0);

function CounterPanel() {
  const [count, setCount] = useAtom(counterAtom);
  console.log("🔴 CounterPanel render");

  return (
    <button
      onClick={() => setCount(count + 1)}
      className="🟨 bg-amber-200 border-1 rounded-4xl p-1"
    >
      {count}
    </button>
  );
}

function ThemePanel() {
  console.log("🔵 ThemePanel render");
  return <div>theme panel</div>;
}

export default function JotaiPage() {
  console.log("🟡 JotaiPage render");

  return (
    <>
      <CounterPanel />
      <ThemePanel />
    </>
```



6. Recoil

atom + selector

React 전용 상태 관리

6. Recoil

개발 정체

React 18 이후 업데이트 부족

커뮤니티 감소

Context API는 설정 전달용,
Zustand는 실무용 전역 상태 관리,
Jotai는 정밀한 상태 분해용,
Recoil은 신규 사용 비추천이다.

끝