📊 코드 비교 가이드

세 가지 방식의 **실제 코드를 직접 비교**하여 차이점을 명확히 이해할 수 있습니다.

목차

---

1. 프로젝트 구조 비교

🟣 Props Drilling (basic-props/)

```
basic-props/
├── src/
```

```
|   ├── components/          📦 컴포넌트만 존재
|   |   ├── Header.jsx
|   |   ├── MainContent.jsx
|   |   ├── CounterSection.jsx
|   |   ├── Counter.jsx
|   |   ├── UserSection.jsx
|   |   └── UserInfo.jsx
|   ├── App.jsx              ☆ 모든 상태 여기서 관리
|   └── main.jsx
└── package.json
```

특징:
- 상태 관리 로직이 `App.jsx`에 집중
- 별도의 상태 관리 폴더 없음
- 가장 단순한 구조

🟡 useContext (usecontext-app/)

```
usecontext-app/
├── src/
|   ├── context/             🆕 Context 폴더 추가
|   |   └── AppContext.jsx   ☆ 전역 상태 관리
```

```
|   ├── components/          📦 컴포넌트
|   |   ├── Header.jsx
|   |   ├── MainContent.jsx
|   |   ├── CounterSection.jsx
|   |   ├── Counter.jsx
|   |   ├── UserSection.jsx
|   |   └── UserInfo.jsx
|   ├── App.jsx              ✨ Provider로 감싸기만 함
|   └── main.jsx
└── package.json
```

특징:

- `context/` 폴더가 추가됨
- 상태 관리 로직이 `AppContext.jsx`로 분리
- `App.jsx`는 단순해짐

🔵 Redux (redux-app/)

```
redux-app/
├── src/
|   ├── store/               🆕 Store 폴더 추가
|   |   ├── store.js         ☆ Redux Store 설정
```

```
|   |   ├── counterSlice.js 📊 카운터 상태 + 로직
|   |   └── userSlice.js    📊 사용자 상태 + 로직
|   ├── components/          📦 컴포넌트
|   |   ├── Header.jsx
|   |   ├── MainContent.jsx
|   |   ├── CounterSection.jsx
|   |   ├── Counter.jsx
|   |   ├── UserSection.jsx
|   |   └── UserInfo.jsx
|   ├── App.jsx              💥 매우 단순함
|   └── main.jsx             🔌 Provider 설정
└── package.json             📦 Redux 패키지 추가
```

특징:

- `store/` 폴더로 상태 관리 완전 분리
- 기능별로 Slice 파일 분리
- 가장 구조화된 형태

---

2. 최상위 컴포넌트 비교 (App.jsx)

🟣 Props Drilling

```jsx
```

```
import { useState } from 'react'
import Header from './components/Header'
import MainContent from './components/MainContent'
import './App.css'

function App() {
  // ☆ 모든 상태를 여기서 관리
  const [count, setCount] = useState(0)
  const [user, setUser] = useState({ name: '홍길동', age: 25 })

  // ☆ 모든 상태 변경 함수를 여기서 정의
  const incrementCount = () => setCount(count + 1)
  const decrementCount = () => setCount(count - 1)
  const resetCount = () => setCount(0)
  const updateUserName = (name) => setUser({ ...user, name })
  const updateUserAge = (age) => setUser({ ...user, age })

  return (
    <div className="app">
      {/* ∥ Props로 일일이 전달 */}
      <Header user={user} />
      <MainContent
        count={count}
        user={user}
        incrementCount={incrementCount}
        decrementCount={decrementCount}
```

```
            resetCount={resetCount}
            updateUserName={updateUserName}
            updateUserAge={updateUserAge}
          />
        </div>
    )
}

export default App
```

코드 라인: 31줄
Props 전달: 7개

🟡 useContext

```jsx
import { AppProvider } from './context/AppContext'
import Header from './components/Header'
import MainContent from './components/MainContent'
import './App.css'

function App() {
  return (
```

```
      {/* 💥 Provider로 감싸기만 하면 끝! */}
      <AppProvider>
        <div className="app">
          {/* ✅ Props 전달 없음! */}
          <Header />
          <MainContent />
        </div>
      </AppProvider>
    )
}

export default App
```

코드 라인: 17줄 (거의 절반!)

Props 전달: 0개 ✨

🔵 Redux

```jsx
import Header from './components/Header'
import MainContent from './components/MainContent'
import './App.css'
```

```
function App() {
  return (
    <div className="app">
      {/* ✔️ Props 전달 없음! */}
      <Header />
      <MainContent />
    </div>
  )
}

export default App
```

코드 라인: 14줄 (가장 짧음!)

Props 전달: 0개 ✨

참고: Provider는 `main.jsx`에서 설정

---

3. 중간 컴포넌트 비교 (MainContent.jsx)

이 컴포넌트는 실제로 상태를 **사용하지 않고** 단순히 하위 컴포넌트를 **레이아웃**합니다.

🟣 Props Drilling

```jsx
```

```
import CounterSection from './CounterSection'
import UserSection from './UserSection'

function MainContent({
  count,                // 📦 받아서
  user,                 // 📦 받아서
  incrementCount,       // 📦 받아서
  decrementCount,       // 📦 받아서
  resetCount,           // 📦 받아서
  updateUserName,       // 📦 받아서
  updateUserAge         // 📦 받아서
}) {
  return (
    <main className="main-content">
      {/* 📦 그대로 전달 (Props Drilling!) */}
      <CounterSection
        count={count}
        incrementCount={incrementCount}
        decrementCount={decrementCount}
        resetCount={resetCount}
      />
      <UserSection
        user={user}
        updateUserName={updateUserName}
        updateUserAge={updateUserAge}
      />
```

```
      </main>
    )
}

export default MainContent
```

문제점:
- 😰 7개의 props를 받음
- 😰 자신은 사용하지 않음
- 😰 단지 전달만 함
- 😰 Props 이름 하나만 틀려도 버그

🟡 useContext

```jsx
import CounterSection from './CounterSection'
import UserSection from './UserSection'

function MainContent() {
  // ✅ Props 없음!
  return (
    <main className="main-content">
      {/* ✅ 깔끔한 전달 */}
```

```
        <CounterSection />
        <UserSection />
      </main>
    )
}

export default MainContent
```

장점:
- ✨ Props 0개
- ✨ 레이아웃 구조만 명확히 표현
- ✨ 코드가 매우 깔끔

🔵 Redux

```jsx
import CounterSection from './CounterSection'
import UserSection from './UserSection'

function MainContent() {
  // ✔️ Props 없음!
  return (
    <main className="main-content">
```

```
        {/* ✅ 깔끔한 전달 */}
        <CounterSection />
        <UserSection />
      </main>
    )
}

export default MainContent
```

장점:
- ✨ useContext와 동일하게 깔끔
- ✨ Props Drilling 문제 완전 해결

---

4. 최하위 컴포넌트 비교 (Counter.jsx)

실제로 카운터 **상태를 사용하고 변경**하는 컴포넌트입니다.

🟣 Props Drilling

```jsx
function Counter({ count, incrementCount, decrementCount, resetCount }) {
  // Props로 받은 값과 함수를 사용
  return (
```

```
      <div className="counter">
        <div className="count-display">{count}</div>
        <div className="button-group">
          <button onClick={decrementCount}>-</button>
          <button onClick={resetCount}>Reset</button>
          <button onClick={incrementCount}>+</button>
        </div>
      </div>
    )
}

export default Counter
```

특징:
- 4개의 props 필요
- 컴포넌트 시그니처가 복잡함

🟡 useContext

```jsx
import { useAppContext } from '../context/AppContext'

function Counter() {
```

```
  // 💥 Hook으로 필요한 것만 가져오기
  const { count, incrementCount, decrementCount, resetCount } = useAppContext()

  return (
    <div className="counter">
      <div className="count-display">{count}</div>
      <div className="button-group">
        <button onClick={decrementCount}>-</button>
        <button onClick={resetCount}>Reset</button>
        <button onClick={incrementCount}>+</button>
      </div>
    </div>
  )
}

export default Counter
```

특징:

- ✅ Props 없이 Hook으로 가져옴
- ✅ 필요한 것만 선택적으로 가져오기
- ✅ 컴포넌트 시그니처가 깔끔

🔵 Redux

```jsx
import { useSelector, useDispatch } from 'react-redux'
import { increment, decrement, reset } from '../store/counterSlice'

function Counter() {
  // 🌟 useSelector로 상태 읽기
  const count = useSelector((state) => state.counter.value)

  // 🌟 useDispatch로 액션 발송 준비
  const dispatch = useDispatch()

  return (
    <div className="counter">
      <div className="count-display">{count}</div>
      <div className="button-group">
        {/* 🌟 dispatch로 액션 발송 */}
        <button onClick={() => dispatch(decrement())}>-</button>
        <button onClick={() => dispatch(reset())}>Reset</button>
        <button onClick={() => dispatch(increment())}>+</button>
      </div>
    </div>
  )
}

export default Counter
```

```

**특징:**

- ✅ Props 없음
- ✅ `useSelector`로 필요한 상태만 선택
- ✅ `dispatch`로 명확한 액션 발송
- ℹ️ 약간 더 많은 코드 (하지만 명확함)

---

## 5. 상태 관리 파일 비교

### 🟣 Props Drilling

```
없음 ❌

모든 상태 관리가 App.jsx 안에 있음
```

**장점:** 별도 파일 불필요

**단점:** App.jsx가 복잡해짐

---

### 🟡 useContext
```

📄 context/AppContext.jsx (42줄)

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

//① Context 생성
const AppContext = createContext()

//② Provider 컴포넌트
export function AppProvider({ children }) {
  const [count, setCount] = useState(0)
  const [user, setUser] = useState({ name: '홍길동', age: 25 })

  const incrementCount = () => setCount(count + 1)
  const decrementCount = () => setCount(count - 1)
  const resetCount = () => setCount(0)
  const updateUserName = (name) => setUser({ ...user, name })
  const updateUserAge = (age) => setUser({ ...user, age })

  // 모든 것을 value 객체에 담기
  const value = {
    count,
    user,
    incrementCount,
    decrementCount,
```

```
    resetCount,
    updateUserName,
    updateUserAge,
  }

  return <AppContext.Provider value={value}>{children}</AppContext.Provider>
}

//③ Custom Hook
export function useAppContext() {
  const context = useContext(AppContext)
  if (!context) {
    throw new Error('useAppContext must be used within AppProvider')
  }
  return context
}
```

장점:
- ✅ 상태 관리가 분리됨
- ✅ Custom Hook으로 편리한 사용
- ✅ 에러 처리 포함

🔵 Redux

📄 store/store.js (11줄)

```jsx
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from './counterSlice'
import userReducer from './userSlice'

export const store = configureStore({
  reducer: {
    counter: counterReducer,
    user: userReducer,
  },
})
```

📄 store/counterSlice.js (25줄)

```jsx
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  value: 0,
}

export const counterSlice = createSlice({
```

```
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      state.value += 1  // Immer가 불변성 관리
    },
    decrement: (state) => {
      state.value -= 1
    },
    reset: (state) => {
      state.value = 0
    },
  },
})

export const { increment, decrement, reset } = counterSlice.actions
export default counterSlice.reducer
```

📄 store/userSlice.js (23줄)

```jsx
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  name: '홍길동',
```

```
  age: 25,
}

export const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {
    updateName: (state, action) => {
      state.name = action.payload
    },
    updateAge: (state, action) => {
      state.age = action.payload
    },
  },
})

export const { updateName, updateAge } = userSlice.actions
export default userSlice.reducer
```

장점:

- ✅ 기능별로 완전히 분리 (Slice)
- ✅ 확장성 뛰어남
- ✅ 각 파일이 단일 책임만 가짐

단점:

- ❌ 파일이 많음
- ❌ 초기 설정이 복잡

---

6. 사용자 정보 컴포넌트 비교

🔵 Props Drilling - UserInfo.jsx

```jsx
function UserInfo({ user, updateUserName, updateUserAge }) {
  return (
    <div className="user-info">
      <div className="info-item">
        <label>이름:</label>
        <input
          type="text"
          value={user.name}
          onChange={(e) => updateUserName(e.target.value)}
        />
      </div>
      <div className="info-item">
        <label>나이:</label>
        <input
          type="number"
          value={user.age}
```

```
            onChange={(e) => updateUserAge(Number(e.target.value))}
          />
        </div>
      </div>
    )
}
```

Props: 3개 (user, updateUserName, updateUserAge)

🟡 useContext - UserInfo.jsx

```jsx
import { useAppContext } from '../context/AppContext'

function UserInfo() {
  // Hook으로 필요한 것만 가져오기
  const { user, updateUserName, updateUserAge } = useAppContext()

  return (
    <div className="user-info">
      <div className="info-item">
        <label>이름:</label>
        <input
```

```
            type="text"
            value={user.name}
            onChange={(e) => updateUserName(e.target.value)}
          />
        </div>
        <div className="info-item">
          <label>나이:</label>
          <input
            type="number"
            value={user.age}
            onChange={(e) => updateUserAge(Number(e.target.value))}
          />
        </div>
      </div>
    )
}
```

Props: 0개 ✨

Hook 사용: useAppContext

🔵 Redux - UserInfo.jsx

```jsx
```

```
import { useSelector, useDispatch } from 'react-redux'
import { updateName, updateAge } from '../store/userSlice'

function UserInfo() {
  // Store에서 user 상태 읽기
  const user = useSelector((state) => state.user)
  const dispatch = useDispatch()

  return (
    <div className="user-info">
      <div className="info-item">
        <label>이름:</label>
        <input
          type="text"
          value={user.name}
          // Action에 payload를 담아서 dispatch
          onChange={(e) => dispatch(updateName(e.target.value))}
        />
      </div>
      <div className="info-item">
        <label>나이:</label>
        <input
          type="number"
          value={user.age}
          // Number로 변환해서 dispatch
          onChange={(e) => dispatch(updateAge(Number(e.target.value)))}
```

```
          />
        </div>
      </div>
    )
}
```

Props: 0개 ✨

Hooks 사용: useSelector, useDispatch

---

7. 코드 라인 수 비교

파일	Props Drilling	useContext	Redux
App.jsx	31줄	17줄	14줄
MainContent.jsx	31줄	14줄	14줄
Counter.jsx	15줄	18줄	21줄
UserInfo.jsx	25줄	28줄	30줄
상태 관리 파일	0줄	42줄	59줄 (3파일)
총계	~100줄	~119줄	~138줄

📊 분석

- **Props Drilling**: 총 코드는 적지만, 반복 코드가 많음

- **useContext**: 중간 정도, 가장 균형잡힌 코드량

- **Redux**: 가장 많지만, 가장 체계적이고 확장성 좋음

---

8. 실행 흐름 비교

🔵 Props Drilling - 카운터 증가 버튼 클릭 시

```
1. Counter.jsx - 버튼 클릭
    ↓
2. incrementCount() 함수 실행 (props로 받은 함수)
    ↓
3. App.jsx의 incrementCount 실행
    ↓
4. setCount(count + 1) 호출
    ↓
5. App.jsx 리렌더링
    ↓
6. MainContent에 새로운 count props 전달
    ↓
7. CounterSection에 새로운 count props 전달
    ↓
8. Counter에 새로운 count props 전달
    ↓
```

9. Counter 리렌더링 → 화면 업데이트 ✅

```

**단계:** 9단계

**리렌더링:** App, MainContent, CounterSection, Counter (4개)

---

### 🟡 useContext - 카운터 증가 버튼 클릭 시

```
1. Counter.jsx - 버튼 클릭
   ↓
2. incrementCount() 함수 실행 (Context에서 가져온 함수)
   ↓
3. AppProvider의 incrementCount 실행
   ↓
4. setCount(count + 1) 호출
   ↓
5. Context value 업데이트
   ↓
6. Context를 구독하는 모든 컴포넌트 리렌더링
   ↓
7. Counter 리렌더링 → 화면 업데이트 ✅
```
```

단계: 7단계

리렌더링: Context를 사용하는 모든 컴포넌트 ⚠️

(성능 이슈 가능)

🔵 Redux - 카운터 증가 버튼 클릭 시

```
1. Counter.jsx - 버튼 클릭
    ↓
2. dispatch(increment()) 실행
    ↓
3. Redux Store에 increment 액션 전달
    ↓
4. counterSlice의 increment reducer 실행
    ↓
5. state.counter.value += 1
    ↓
6. Store 업데이트
    ↓
7. useSelector(state => state.counter.value)를 사용하는 컴포넌트만 리렌더링
    ↓
8. Counter 리렌더링 → 화면 업데이트 ✅
```

단계: 8단계

리렌더링: 해당 state를 구독하는 컴포넌트만 ✅

(성능 최적화 가능)

---

9. 요약 비교표

| 항목 | Props Drilling | useContext | Redux |

|---|---|---|---|

| **파일 수** | 가장 적음 | 중간 (+1) | 가장 많음 (+3) |

| **코드 라인** | ~100줄 | ~119줄 | ~138줄 |

| **학습 난이도** | ☆ 쉬움 | ☆☆ 보통 | ☆☆☆ 어려움 |

| **Props 전달** | 많음 😰 | 없음 ✅ | 없음 ✅ |

| **코드 반복** | 많음 😰 | 적음 ✅ | 없음 ✅ |

| **리렌더링 제어** | 좋음 ✅ | 주의 필요 ⚠️ | 매우 좋음 ✅ |

| **확장성** | 나쁨 ❌ | 보통 🟡 | 뛰어남 ✅ |

| **디버깅** | 쉬움 ✅ | 보통 🟡 | 강력 ✅ |

| **테스트** | 쉬움 ✅ | 보통 🟡 | 쉬움 ✅ |

---

10. 실전 선택 가이드

🟣 Props Drilling을 선택하세요

```jsx
// ✅ 좋은 사례
<Parent>
  <Child data={data} />
</Parent>

// ❌ 나쁜 사례
<Level1>
  <Level2>
    <Level3>
      <Level4>
        <Level5 data={data} />  // 너무 깊음!
      </Level5>
    </Level4>
  </Level3>
</Level2>
</Level1>
```

조건:
- 컴포넌트 깊이 2-3단계
- 전달할 props 3개 이하
- 빠른 프로토타입

🟡 useContext를 선택하세요

```jsx
// ✅ 좋은 사례
<ThemeProvider>
  <App /> {/* 어디서든 theme 접근 */}
</ThemeProvider>

<AuthProvider>
  <App /> {/* 어디서든 user 접근 */}
</AuthProvider>
```

조건:
- 전역 설정 (테마, 언어, 인증)
- 자주 변경되지 않는 데이터
- 중규모 앱

🔵 Redux를 선택하세요

```jsx
// ✅ 좋은 사례
store/
  ├── authSlice.js
```

```
    ├── productsSlice.js
    ├── cartSlice.js
    ├── ordersSlice.js
    ├── reviewsSlice.js
    └── store.js
```

조건:

- 복잡한 상태 로직
- 많은 전역 상태
- 팀 협업
- 대규모 앱

📝 결론

시각적 비교

```
Props Drilling:  [간단함] ██████████▒▒▒▒▒ [복잡함]
useContext:      [간단함] ██████▒▒▒▒▒▒▒▒▒▒ [복잡함]
Redux:           [간단함] ███▒▒▒▒▒▒▒▒▒▒▒▒▒ [복잡함]

확장성:
Props Drilling:  [낮음] ███▒▒▒▒▒▒▒▒▒▒▒▒▒▒ [높음]
```

useContext: [낮음] ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ [높음]

Redux: [낮음] ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ [높음]

성능:

Props Drilling: [나쁨] ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ [좋음]

useContext: [나쁨] ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ [좋음]

Redux: [나쁨] ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ [좋음]

```

### 최종 추천

1. **시작은 Props Drilling으로** 🟣
2. **복잡해지면 useContext로 전환** 🟡
3. **정말 복잡하면 Redux 도입** 🔵

**중요:** 프로젝트 규모와 팀 상황에 맞게 선택하세요!

---

**Happy Coding! 🥕**

직접 세 프로젝트를 실행해보고 코드를 비교하면서 학습하세요!
```