[React-deep-dive]-5장.리액트와 상태 관리 라이브러리

useState와 Context 동시에 사용하기

useState와 Context의 동시 사용

- 앞선 방법의 경우, **하나의 스토어만 가지게 되는 구조**라 마치 전역 변수처럼 작동하게 되어 여러 개의 스토어를 가질 수 없다.
- Context를 활용하여 해당 스토어를 하위 컴포넌트에 주입한다면 컴포넌트에서는 자신이 주입된 스토어에 대해서만 접근할 수 있다.
- 이 Context에서 내려주는 값을 사용하기 위해서는 useContext를 사용해 스토어에 접근할 수 있는 새로운 훅이 필요하다.

결론: Context를 사용해서 특정 스토어를 특정 컴포넌트에만 주입하고 싶다 근데 이럴려면 주입받은 스토어에 접근하는 훅이 필요

스토어와 Context 함께 사용하기

결론

- **Context로 컴포넌트 트리 내 상태 격리가 필요한 부분에 Provider생성 ⇒ 각기 다른 초기값 설정후 개별 관리**
- **스토어를 사용하는 컴포넌트는 해당 상태가 어느 스토어에서 온건지 신경쓰지 않아도 됨**
- **Context와 Provider를 관리하는 부모 컴포넌트의 입장에서, 보여주고 싶은 데이터를 Context로 잘 격리 하기만 하면 됨 → 각 컴포넌트의 책임과 역할이 명시적인 코드로 구**

분

결론: 리액트 ecosystem에는 많은 상태 관리 라이브러리가 있지만 이것들의 핵심은 결국 아래와 같음

- **useState, useReducer가 가지는 지역 상태라는 점을 극복하기 위해 외부 어딘가에 상태를 둔다.**
- **이는 컴포넌트의 최상단 내지는 상태가 필요한 부모, 혹은 격리된 자바스크립트 스코프 등등 어딘가이다.**
- **이 외부의 상태 변경을 각자의 방식으로 감지해 컴포넌트의 렌더링을 일으킨다.**

상태 관리 라이브러리 살펴보기

Recoil

> *Recoil을 사용하면 atoms (공유 상태)에서 selectors (순수 함수)를 거쳐 React 컴포넌트로 내려가는 data-flow graph를 만들 수 있다. Atoms는 컴포넌트가 구독할 수 있는 상태의 단위다. Selectors는 atoms 상태 값을 동기 또는 비동기 방식을 통해 변환한다.*

RecoilRoot

- 애플리케이션 최상단에 RecoilRoot 생성
- 값의 변경이 발생하면 참조하고 있는 하위 컴포넌트에 알림 (notifyComponents)

Atom : 업데이트와 구독이 가능한 상태의 단위

- Atom이 업데이트 되면 각각 구독된 컴포넌트는 새로운 값을 반영하여 다시 렌더링 된다.
- Atom은 React의 로컬 컴포넌트의 상태 대신 사용할 수 있다. 동일한 Atom이 여러 컴포넌트에서 사용되는 경우 모든 컴포넌트는 상태를 공유한다.

```
const fontSizeState = atom({
  key: 'fontSizeState',//atom을 구별하는 식별자가 되는 필수 값 key
  default: 14,//atom의 초기값
});
```

```
function FontButton() {
  const [fontSize, setFontSize] = useRecoilState(fontSizeState);
  return (
    <button onClick={() => setFontSize((size) => size + 1)} style={{fontSize}}>
      Click to Enlarge
    </button>
  );
}

function Text() {
  const [fontSize, setFontSize] = useRecoilState(fontSizeState);
  return <p style={{fontSize}}>This text will increase in size too.</p>;
}
```

Selectors : atoms나 다른 selectors를 입력으로 받아들이는 순수 함수.

- 종속관계의 atoms 또는 selectors가 업데이트되면 하위의 selector 함수도 다시 실행
- 컴포넌트들은 selectors를 atoms처럼 구독할 수 있으며 selectors가 변경되면 컴포넌트들도 다시 렌더링 된다.

Selectors는 상태를 기반으로 하는 파생 데이터를 계산하는 데 사용된다. 최소한의 상태 집합만 atoms에 저장하고 다른 모든 파생되는 데이터는 selectors에 명시한 함수를 통해 효율적으로 계산함으로써 쓸모없는 상태의 보존을 방지한다.

```
const fontSizeState = atom({
  key: 'fontSizeState',//atom을 구별하는 식별자가 되는 필수 값 key
  default: 14,//atom의 초기값
});

const fontSizeLabelState = selector({
  key: 'fontSizeLabelState',
  get: ({get}) => {
    const fontSize = get(fontSizeState); //atomTest와 종속관계가 생성
    const unit = 'px';

    return `${fontSize}${unit}`;
  },
});

function FontButton() {
  const [fontSize, setFontSize] = useRecoilState(fontSizeState);
  const fontSizeLabel = useRecoilValue(fontSizeLabelState);

  return (
    <>
      <div>Current font size: ${fontSizeLabel}</div>

      <button onClick={setFontSize(fontSize + 1)} style={{fontSize}}>
        Click to Enlarge
      </button>
    </>
  );
}
```

Jotai

> *원자를 결합하여 빌드 상태를 생성하고, 원자 의존성을 기반으로 렌더링을 자동으로 최적화합니다. 이를 통해 React 컨텍스트의 추가 재렌더링 문제를 해결하고, 메모이제이션의 필요성을 없애며, 선언적 프로그래밍 모델을 유지하면서도 시그널과 유사한 개발자 경험을 제공합니다.*

- Recoil의 atom에 영감을 받아 착안 → 작은 단위의 상태를 위로 전달 하는 상향식 접근법
- 상위 클래스에서 틀을 정의하고,세부적인 단계는 하위 클래스가 오버라이딩 해서 구현하는 클래스 기반 상속 패턴
  - → **따라서 클래스 기반 상속 구조 프로젝트에 적합**
- 리액트 Context의 문제점인 불필요한 리렌더링이 일어난다는 문제를 해결 하고자 설계, 추가적으로 개발자들이 메모이제이션이나 최적화를 거치지 않아도 리렌더링이 발생되지 않도록 설계

Jotai vs Recoil

구분	Recoil	Jotai
설계 접근	하향 접근법	상향 접근법
구조 방식	작은 단위(atom, selector)를 먼저 만들고 전체 조합	전체 흐름(알고리즘 틀)을 먼저 만들고 세부 구현
사용 방식	세부 로직(상태 단위)을 구성 → 점진적 조립	전체 알고리즘 구조 먼저 설계 → 세부 로직 하위클래스에서
	atom → selector → 컴포넌트 조립	추상 메서드 정의 → 하위 클래스에서 구체화
유연성	조립형구조	고정, 통제된 흐름

```
const switchAtom = atom(false)

const SetTrueButton = () => {
  const state = useAtomValue(switchAtom)
  const setCount = useSetAtom(switchAtom)
  const setTrue = () => setCount(true)
```

```
  return (
    <div>
      State: <b>{state.toString()}</b>
      <button onClick={setTrue}>Set True</button>
    </div>
  )
}
```

Zustand

- **애플리케이션의 여러 상태를 context가 아니라 스토어를 중앙 집중형으로 활용해 스토어 내부에서 상태 관리**
- **다양한 변경 패턴 존재**
  - **partial(=일부분), replace(=전체 변경) ,reset (=전체를 초기값으로)**
- **스토어의 상태가 변경되면 이상태를 구독하고 있는 컴포넌트에 전파해 리렌더링을 알리는 방식**
- **Redux 에서 영감을 받아, 좀더 가볍고 유연한 라이브러리를 지향점으로 삼아 착안**

create : 상태를 변경, 조회 하는 set, get

```
import { create } from 'zustand'

export const useStore = create((set) => ({
  bears: 0, //상태:초깃값,
  count: 0 ,
  increasePopulation: () => set((state) => ({ bears: state.bears + 1 })), // ← partia
  removeAllBears: () => set({ bears: 0, count: 0 }), // ←reset
  updateBears: (newBears) => set({...newBears }), // ← replace
  //액션 : () => { ...
}))
```

```
function BearCounter() {
  const bears = useStore((state) => state.bears)// ← 이것도 partial임 부분구독

  return <h1>{bears} bears around here...</h1>
}

function Controls() {
  const increasePopulation = useStore((state) => state.increasePopulation)
  return <button onClick={increasePopulation}>one up</button>
}
```

replace 와 reset 차이

전체를 초기값으로 돌리는거나, reset이나 뭐가 다를까?

replace : 전체 상태를 새 객체로 완전히 교체, 외부에서 전체 상태 복원 에 주로 사용

reset : 초기값으로 되돌리는 액션을 직접 정의한 것

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

// 초기 상태
const initialState = {
  count: 0,
  text: '',
};

const useStore = create((set) => ({
  ...initialState,
    ...,
    //replace
  updateText: (newText) => set({ ...newText },),
  // reset
```

```
  reset: () => set(initialState),
    //reset: (newText) => set({ text: newText }),초기화 라는 목적에 부합x, 권장하지 않
}));


//사용시
...
    <button onClick={() => updateText('hello')}>Set Text</button> // => replace
     <button onClick={reset}>Reset</button> // => reset
```

- setState의 replace 플래그가 설정된 경우 더 엄격한 유형으로 취급 (in Typescript)

  `replace: true`

  - 현재 상태를 완전히 덮어쓰는 방식
  - 기존 상태와 새로운 상태가 동일한 구조여야 한다는 규칙 적용

```
...
updateText: (newText) => set({ ...newText },true),
```

그래서 어떤 걸 써야 할까? (개인적인 생각)

https://npmtrends.com/@tanstack/react-query-vs-jotai-vs-recoil-vs-redux-vs-swr-vs-zustand

Zustand와 tanstack 컴바인 → https://chatgpt.com/c/67fc6422-56cc-8008-823b-c40c3c3aeffa

	Zustand	**Redux (Toolkit)**	**TanStack Query**
주요 목적	전역 상태 관리 (간단하고 유연함)	전역 상태 관리 (예측 가능한 상태 관리)	서버 상태 (fetching, caching, syncing) 관리

설치 및 설정	매우 간단 (`zustand` 만 설치)	상대적으로 복잡 (store, slice, middleware 등 설정 필요)	매우 간단 (`@tanstack/react-query` 설치, Provider 만 설정)
기본 상태 타입	클라이언트 상태 (UI 관련, 임시 데이터 등)	클라이언트 상태 (UI, 폼, 인증 등 다양한 용도)	**서버 상태** (API 요청, 비동기 데이터 등)
비동기 처리	상태 내에서 처리 가능 (fetch 함수 정의)	Thunk, Saga, Middleware 사용	내장 `useQuery` , `useMutation` 훅 제공 (자동 상태 관리)
초기 학습 난이도	낮음 (React Hook 기반)	중간 이상 (구조 익숙해져야 함)	낮음~중간 (비동기 로직 이해 필요)
코드량	적음 (함수형 스타일, boilerplate 없음)	많음 (slice, action, reducer 등 필요)	적음 (hook 기반)
미들웨어 지원	자체적으로 제공 (e.g., persist, devtools, immer 등)	다양한 middleware 지원 가능	미들웨어 개념 X (플러그인처럼 확장 가능)
DevTools 지원	`zustand/devtools` 로 지원	Redux DevTools 완벽 연동	React Query Devtools 제공
캐싱/리페칭	수동 구현 필요	수동 구현 필요	**자동 캐싱 / 자동 리페칭 / background sync 제공**
서버 상태 동기화	(직접 fetch & set 필요)	(직접 처리 필요)	(자동 리페칭, polling, stale 관리, optimistic update 등)
Persist (로컬 저장)	`zustand/middleware` 에서 제공	미들웨어 필요 (`redux-persist` 등)	자체 지원 X (직접 구현 필요)
생태계	작지만 실용적 (`persist` , `devtools` , `immer` , `context bridge`)	크고 활발 (많은 middleware와 툴 존재)	매우 활발 (다양한 비동기 상황 커버)

개인적인 결론

간단한 프로젝트 → 굳이 안써도 될것 같다.

아래보다 상대적으로 간단한 프로젝트 → Zustand / Zustand + TanStack Query / TanStack Query

복잡한 전역 상태 (여러 페이지에서 공유되는 폼, 인증 등) → Redux Toolkit