React &
State Management

리액트와 상태 관리 라이브러리

2024.03.17

두선아

Contents

1. 상태 관리의 역사

리액트와 상태 관리 since 2014

1. 상태 관리의 역사 State Management?

상태

어떠한 의미를 가진 값

애플리케이션의 시나리오에 따라
지속적으로 변경될 수 있는 값

State

UI: 상호 작용이 가능한 모든 요소
URL: 브라우저에서 관리되고 있는 상태값
form: 폼의 상태
서버 응답: 클라이언트에서 서버로 요청을 통해 가져온 값

Management

- 웹 내부에서 관리해야할 상태들의 관리
- 전역으로 관리해야할 상태의 위치는 어디?
- tearing을 방지

1. 상태 관리의 역사

Flux 패턴이란?

Flux

Redux 이전의 이야기

2014년경, 리액트 등장과 비슷한 시기에
Flux 패턴, 그리고 Flux 라이브러리가 나타났더랬다...

Flux pattern

안정적으로 관리하자

양방향 => 단방향으로 데이터 흐름 변경

- Action -> Dispatcher -> Model -> View

Flux 라이브러리

Flux | alt | RefluxJS | NuclearJS | Fluxible | Fluxxor

목적

MVC 패턴의 복잡도가 증가하던 상황에서
상태 변화의 원인과 위치 추적의 어려움을 해결하기 위한 시도

구분	역할	작동
action	작업을 처리한 액션, 액션 발생 시 포함시킬 데이터	액션 타입과 데이터를 정의해 디스패쳐로 보냄
dispatcher	액션을 스토어에 보냄	콜백 함수 형태로, 앞서 액션이 정의한 타입과 데이터를 스토어에 보냄
store	실제 상태에 따른 값과 상태를 변경할 수 있는 메서드	액션 타입에 따라, 어떻게 변경할 지 정의되어 있음
view	(리액트 컴포넌트에 해당) 스토어에서 만들어진 데이터를 가져와 화면을 랜더링	액션을 호출해서 상태 업데이트를 할 수 있음

1. 상태 관리의 역사

Redux의 두둥등장

React & Context

시장을 지배한다

Redux

Flux 구조 + Elm 아키텍처

Model | View | Update

단점

- 상태 관리를 위한 액션 타입 선언, creator 함수 작성, dispatcher와 selector 사용
- 간단한 상태를 추가하기 위해서 보일러플레이트 작성이 필요 (현재는 많이 간소화됨)

React & Context

컨텍스트 값을 구독하는
모든 컴포넌트가
컨텍스트의 값이 변경될 때
리렌더링 된다.

React 16.3 이전 getChildContext()

- 상위 컴포넌트 랜더링 시, getChildContext가 호출되어 리랜더링
- context를 인수로 받아야해서, 컴포넌트와 결합도가 높아짐

React 16.3 이후 createContext | Context.Provider

- Context API
- 컴포넌트 트리 안에서 데이터를 전역적으로 관리

---

1. 상태 관리의 역사

Hook, React Query, SWR(Vercel)
Recoil(페이스북), Zustand, Jotai, Valtio

훅과 state을 이용한 상태 관리 등장

React 16.8 이후 Hook State 캐싱 API 호출 상태

React Query(Tanstack Query), SWR

- fetch 관리에 특화
- API 호출에 대한 상태를 관리
- HTTP 요청에 특화된 상태 관리 라이브러리

더 범용적인 상태관리 라이브러리

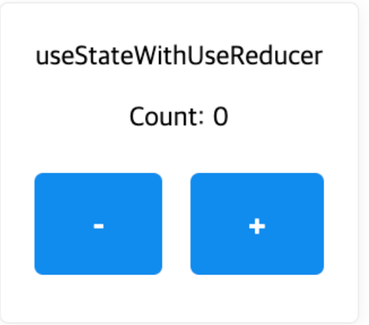
React 16.8 이후 Hook State 작은 크기의 상태

Recoil(페이스북), Zustand, Jotai, Valtio

- peerDependencies는 react 16.8^
- 개발자가 원하는 만큼 상태를 지역적으로 관리

2. State Management

지역 상태 관리-1



useReducer로 useState 만들기

useStateWithUseReducer

커스텀훅

```tsx
// 1.
function useStateWithUseReducer<T>(initState: T) {
  const [state, dispatch] = useReducer(
    (prev: T, action: Initializer<T>) =>
      typeof action === "function" ? action(prev) : action,
    initState
  );

  return [state, dispatch];
}
```

실행

```tsx
const StateWithUseReducerComponent: React.FC = () => {
  const [count, setCount] = useStateWithUseReducer(0);

  return (
    <section>
      <aside>
        <h2>useStateWithUseReducer</h2> <p>Count: {count}</p>
        <div className="flex justify-between">
          <button onClick={() => setCount(count - 1)}>-</button>
          <button onClick={() => setCount(count + 1)}>+</button>
        </div>
      </aside>
    </section>
  );
};
```

2. State Management

지역 상태 관리-2

useReducerWithUseState Example

Count: 0

- +

useState로 useReducer 만들기

useReducerWithUseState

커스텀훅

```tsx
function useReducerWithUseState<T, A>(
  reducer: (state: T, action: A) => T,
  initialState: T,
  initializer?: (initial: T) => T
): [state: T, dispatch: (action: A) => void] {
  const [state, setState] = useState(
    initializer ? () => initializer(initialState) : initialState
  );

  const dispatch = useCallback(
    (action: A) => setState((prev) => reducer(prev, action)),
    [reducer]
  );

  return [state, dispatch];
}
```

실행

```tsx
const ReducerWithUseStateComponent: React.FC = () => {
  const [state, dispatch] = useReducerWithUseState(counterReducer, {
    count: 0,
  });

  return (
    <section>
      <aside>
        <h2>useReducerWithUseState Example</h2>
        <p>Count: {state.count}</p>
        <div className="flex justify-between">
          <button onClick={() => dispatch({ type: "increment" })}>-</button>
          <button onClick={() => dispatch({ type: "increment" })}>+</button>
        </div>
      </aside>
    </section>
  );
};
```

reducer

```tsx
// 2.
type CounterAction = { type: "increment" | "decrement" };

function counterReducer(
  state: { count: number },
  action: CounterAction
): { count: number } {
  switch (action.type) {
    case "increment":
      return { ...state, count: state.count + 1 };
    case "decrement":
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
}
```

2. State Management

useState 상태를 바깥으로

5장 353p~371p

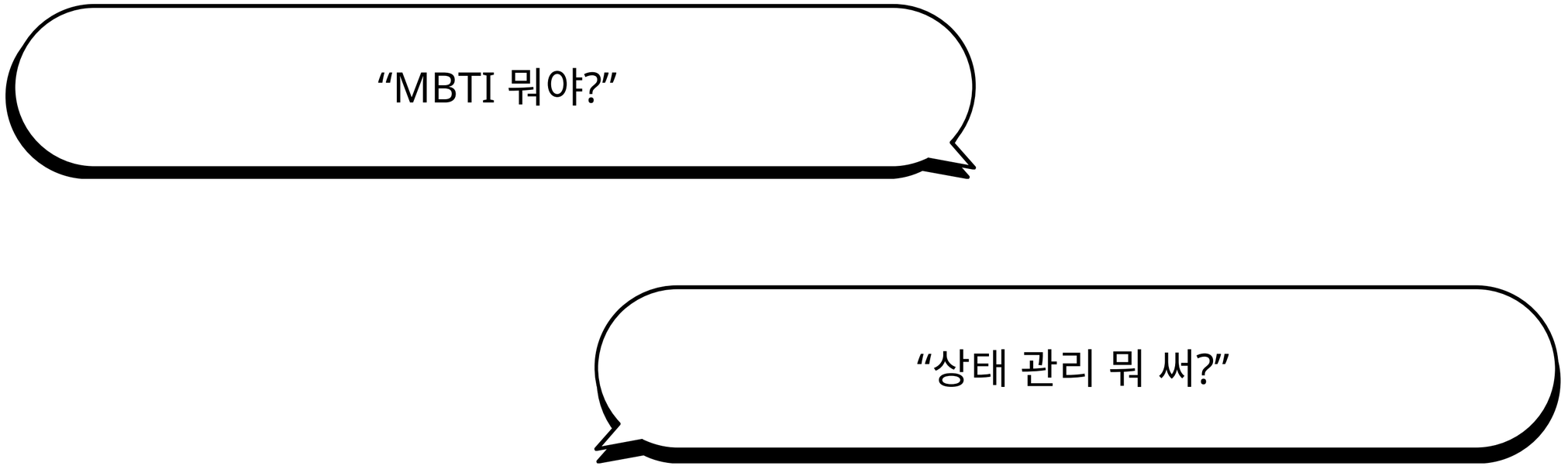
- createStore와 useCreateStore 예제
  - 컴포넌트 외부 어딘가에 상태를 두고, 여러 컴포넌트가 같이 쓸 수 있어야 함
  - 상태의 변화를 알아챌 수 있어야 하고,
    상태 변화 시 리랜더링을 일으켜 컴포넌트를 최신 상태값 기준으로 랜더링해야 함. (모든 컴포넌트에서 동일 작동)
  - 원시값 아닌 객체인 경우, 감지하지 않는 값이 변하면 리랜더링 x
    - obj.a가 바뀌었을 때 obj.b를 의존하는 컴포넌트가 리랜더링되면 x (객체 참조는 그대로)
- useState과 Context를 사용하기
  - 리액트의 useSubscription
  - 더 넓은 스코프의 상태 관리 예제코드
  - 각 스토어를 Context로 격리

3. 상태 관리의 미래

더 많은 선택지와 유연성

3. 상태 관리의 미래

WEB front-end 스몰톡

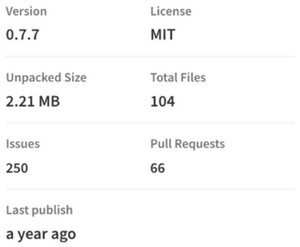


3. 상태 관리의 미래

요즘 상태관리 라이브러리

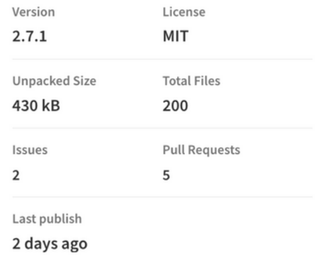
공통점

- React 16.8 이상
- 단방향 데이터 흐름
- 타입스크립트 지원



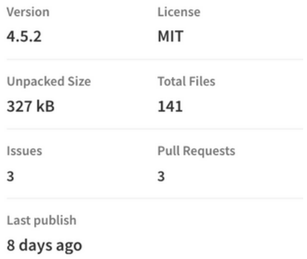
Recoil

- React 생태계에서 시작된 최소 상태 개념인 Atom을 중심으로 구축
- 상태의 정의와 업데이트 방식이 명확



Jotai

- Atom 개념을 사용
- Atom으로 상태를 만들거나, 파생 상태를 만듦



Zustand

- Redux와 유사한 패턴을 제공하지만 훨씬 간결하고 직관적인 API
- 설정이 거의 필요 없어 빠르게 상태 관리 시스템을 구축

3. 상태 관리의 미래

간단하게 쓸 수 있는 전역 상태 관리툴 찾기

recoil TS

0.7.7 • Public • Published a year ago

Node.js CI failing

Version	License
0.7.7	MIT
Unpacked Size	**Total Files**
2.21 MB	104
Issues	**Pull Requests**
250	66
Last publish	
a year ago	

이번 챕터 중요하다

Recoil 쓰고 있는데 바꿔야해요🔥

Jotai와 Recoil는 얼마나 비슷한가?

NPM trend

5장 예제 너무 어려워

Zustand는 정말 간단한가?

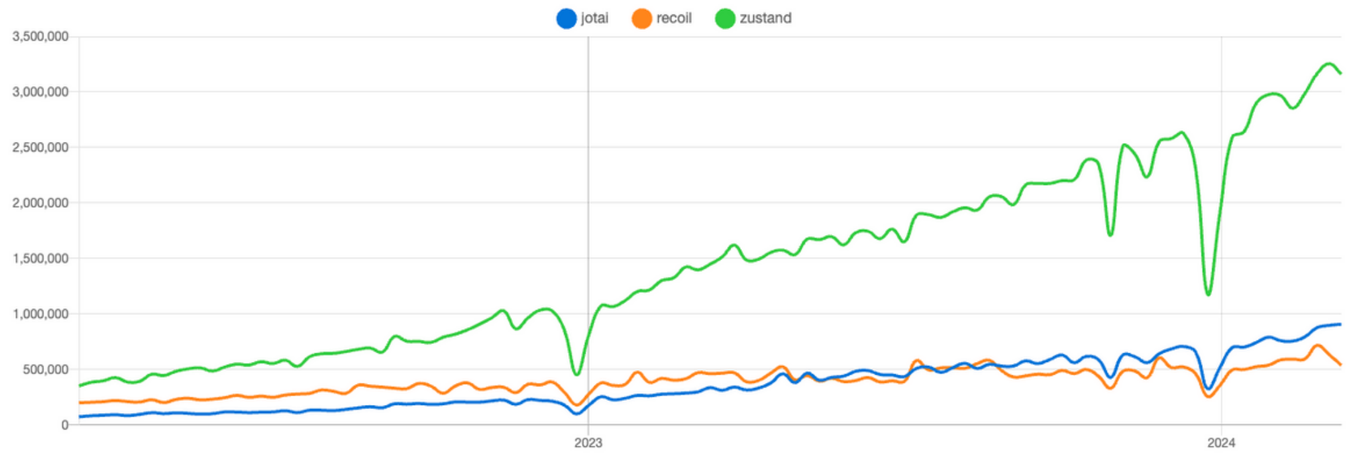
안정성 있는 라이브러리

일단 써보자

3. 상태 관리의 미래 🤔

NPM Trend

Zustand의 인기 확인

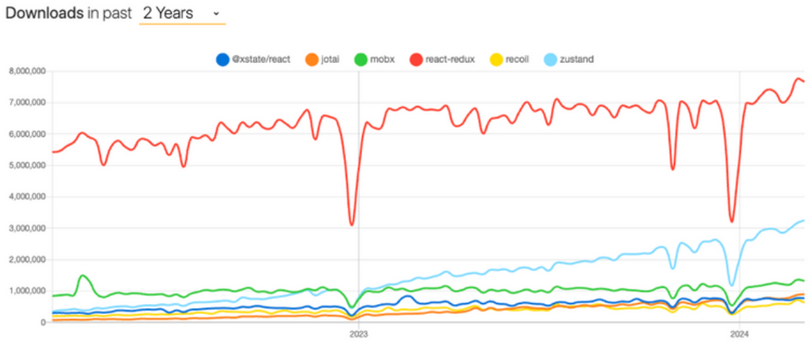


인기 있는 라이브러리는 이유가 있다!

많이 쓴다고 좋은 라이브러리는 아니지만

좋은 라이브러리는 많이 쓰기 때문

리덕스가 넘사벽이긴 해요



Recoil

state

```
<RecoilRoot>
  <Recoil />
</RecoilRoot>
```

```ts
import { atom } from "recoil";

export const countState = atom({
  key: "countState",
  default: 0,
});
```

TS countAtom.ts

get value

```ts
export default function Recoil() {
  const count = useRecoilValue(countState);

  return (
    <section>
      <aside>
        <label>Recoil</label>

        {/* Recoil */}
        <div>{count}</div>
        <ChildComponent />
      </aside>
    </section>
  );
}
```

action

```ts
const ChildComponent = () => {
  const [count, setState] = useRecoilState(countState);

  return (
    <ConterButton
      onClick={() => setState((prev) => prev + 1)}
      disabled={count > 3}
    />
  );
};
```

Jotai

state

```ts
const countState = atom(0);
```

get value

```ts
export default function Jotai() {
  const [count] = useAtom(countState);

  return (
    <section>
      <aside>
        <label>Jotai</label>

        {/* Jotai */}
        <div>{count}</div>
        <ChildComponent />
      </aside>
    </section>
  );
}
```

action

```ts
const isBiggerThan3 = atom((get) => get(countState) > 3);

const ChildComponent = () => {
  const [, setCount] = useAtom(countState);
  const biggerThan3 = useAtomValue(isBiggerThan3);

  const onClick = () => setCount((prev) => prev + 1);

  return <ConterButton onClick={onClick} disabled={biggerThan3} />;
};
```

Zustand

state

TS useCountStore.ts

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

interface CountState {
  count: number;
  increase: () => void;
}

export const useCountStore = create<CountState>((set) => ({
  count: 0,
  increase: () => set((state) => ({ count: state.count + 1 })),
}));
```

get value

```ts
export default function Zustand() {
  const count = useCountStore((state) => state.count);

  return (
    <section>
      <aside>
        <label>Zustand</label>

        {/* Zustand */}
        <div>{count}</div>
        <ChildComponent />
      </aside>
    </section>
  );
}
```

action

```ts
const isBiggerThan3 = () => {
  const count = useCountStore((state) => state.count);
  return count > 3;
};

const ChildComponent = () => {
  const increase = useCountStore((state) => state.increase);
  const biggerThan3 = isBiggerThan3();

  return <ConterButton onClick={increase} disabled={biggerThan3} />;
};
```

3. Zustand

1. 스토어 만들기 쉽다 => 정말임
2. 라이브러리 크기가 장점!
3. API가 복잡하지 않고, 사용이 간단한 코드
4. 미들웨어와 플러그인 지원

미들웨어

```
import create from 'zustand';
import { persist } from 'zustand/middleware';
```

```
export const useThemeStore = create<ThemeState>(persist(
  (set) => ({
```

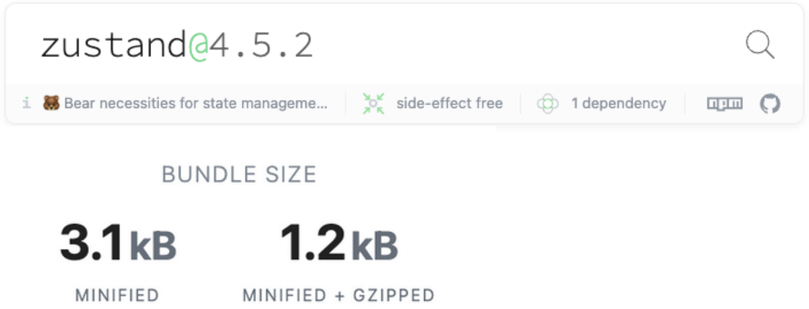
TS useCountStore.ts

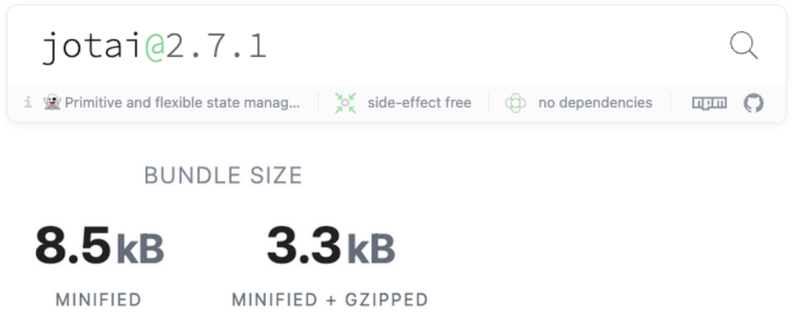
```
import { create } from "zustand";

interface CountState {
  count: number;
  increase: () => void;
}

export const useCountStore = create<CountState>((set) => ({
  count: 0,
  increase: () => set((state) => ({ count: state.count + 1 })),
}));
```

bundlephobia





Thank You

모던 리액트 Deep Dive

5장 리액트와 상태 관리 라이브러리 (336p~399p)