리액트 상태 관리의 진화 - useReducer와 상태 관리

리액트(React)는 현대 웹 개발에서 가장 널리 사용되는 라이브러리 중 하나이고, 컴포넌트 기반의 구조로, UI를 효율적으로 관리하고 업데이트할 수 있게 해준다.

리액트의 핵심 기능 중 하나인 상태 관리(State Management)는 앱의 복잡성을 줄이고, 사용자 인터페이스(UI)의 일관성을 유지하는 데 중요한 역할을 한다.

리액트 상태 관리란?

리액트에서 상태(State)는 컴포넌트의 렌더링 결과에 영향을 미치는 데이터이다.

이 데이터는 사용자 입력, 네트워크 요청의 결과 등 다양한 소스에서 올 수 있다.

리액트는 상태가 변경될 때마다 UI를 다시 렌더링하여 사용자에게 최신 정보를 보여준다.

useState와 useReducer

리액트에서 상태를 관리하기 위해 가장 자주 사용하는 훅은 useState이다.

하지만 상태 로직이 복잡해지거나 여러 컴포넌트에서 상태를 공유해야 할 때는 useReducer가 더 적합할 수 있다.

**useState:**

useState는 상태 값과 해당 값을 갱신하는 함수를 제공. (상태가 단순할 때 사용하기 좋음)

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**useReducer:**

useReducer는 상태 업데이트 로직을 컴포넌트 외부로 추출하여 더 구조화된 형태로 관리할 수 있다. (복잡한 상태 로직을 처리할 때 유용)

import { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

throw new Error();

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState);

return (

<div>

<p>Count: {state.count}</p>

<button onClick={() => dispatch({ type: 'increment' })}>Increment</button>

<button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>

</div>

);

}

useReducer와 dispatch는 뭐지?

* **useReducer**: 특정 행동이 일어났을 때, 상태(state)를 어떻게 바꿀지 결정하는 리액트의 도구예요.
* **dispatch**: 어떤 행동(action)이 일어났는지 알려주는 도구예요.

2. 비유로 설명해볼게요

상상해보세요. 여러분은 집에서 엄마, 아빠에게 무언가를 요청하는 아이라고 생각해봐요.

* **상태(state)**: 여러분의 방 상태예요. 예를 들어, 방에 장난감이 몇 개 있는지, 책이 몇 권 있는지 등.
* **행동(action)**: 여러분이 엄마, 아빠에게 하는 요청이에요. 예를 들어, "장난감 더 주세요!" 혹은 "책 정리해 주세요!" 같은 요청.
* **reducer**: 엄마, 아빠가 여러분의 요청을 듣고 방의 상태를 어떻게 바꿀지 결정하는 과정이에요.
* **dispatch**: 여러분이 엄마, 아빠에게 하는 요청 그 자체예요.

import { useReducer } from 'react';

*// 1. 상태 변화 로직*

function reducer(state, action) {

switch (action.type) {

case 'ADD\_TOY':

return { ...state, toys: state.toys + 1 };

case 'ADD\_BOOK':

return { ...state, books: state.books + 1 };

default:

return state;

}

}

function App() {

*// 2. 초기 상태*

const initialState = { toys: 0, books: 0 };

*// 3. useReducer 사용*

const [state, dispatch] = useReducer(reducer, initialState);

return (

<div>

<h1>My Room</h1>

<p>Toys: {state.toys}</p>

<p>Books: {state.books}</p>

{/\* 4. dispatch 사용 \*/}

<button onClick={() => dispatch({ type: 'ADD\_TOY' })}>Add Toy</button>

<button onClick={() => dispatch({ type: 'ADD\_BOOK' })}>Add Book</button>

</div>

);

}

export default App;

4. 코드 설명

1. **reducer 함수**: 이 함수는 여러분의 부모님이 어떤 요청을 받았을 때 방의 상태를 어떻게 바꿀지 결정하는 부분이에요.
   * ADD\_TOY: 장난감을 하나 추가해요.
   * ADD\_BOOK: 책을 하나 추가해요.
2. **initialState**: 방의 초기 상태예요. 처음에는 장난감과 책이 0개씩 있어요.
3. **useReducer 사용**: useReducer를 사용하여 상태(state)와 dispatch 함수를 만들어냈어요.
   * state: 현재 방의 상태예요.
   * dispatch: 여러분이 부모님에게 요청하는 역할이에요.
4. **dispatch 사용**: 버튼을 클릭하면 dispatch를 사용하여 특정 행동(action)을 부모님에게 요청해요.
   * dispatch({ type: 'ADD\_TOY' }): "장난감 하나 더 주세요!"라고 요청해요.
   * dispatch({ type: 'ADD\_BOOK' }): "책 하나 더 주세요!"라고 요청해요.

5. 정리

* **useReducer**: 상태(state)를 관리하는 도구예요.
* **dispatch**: 특정 행동(action)을 요청하는 도구예요.
* **reducer 함수**: 요청을 받았을 때 상태(state)를 어떻게 바꿀지 결정해요.

이렇게 useReducer와 dispatch를 사용하면, 어떤 요청에 따라 상태를 체계적으로 관리할 수 있어요.

부모님이 여러분의 요청을 듣고 방의 상태를 바꾸는 것처럼, 리액트가 여러분의 요청을 듣고 컴포넌트의 상태를 바꿔주는 거예요.

useReducer의 작동 방식

1. **초기 상태 설정**: useReducer는 두 번째 인수로 초기 상태(initialState)를 받습니다.
2. **리듀서 함수**: 상태를 업데이트하는 로직을 포함한 함수로, 현재 상태와 액션을 인수로 받습니다.
3. **디스패치 함수**: 액션을 발생시켜 리듀서 함수를 실행합니다.

이 패턴은 리덕스(Redux)와 비슷한 구조를 가지므로, 리덕스를 사용해본 사람이라면 쉽게 이해할 수 있다.

상태 관리의 장점

1. **유지보수 용이**: 상태 업데이트 로직을 한 곳에 모아두면 코드를 더 쉽게 유지보수할 수 있다.
2. **확장성**: 컴포넌트가 커지거나 상태가 복잡해질 때, 상태 관리 로직을 잘 분리해두면 확장하기 쉽다.
3. **테스트 가능성**: 리듀서 함수는 순수 함수로 만들기 쉽기 때문에, 상태 업데이트 로직을 쉽게 테스트할 수 있다.

useReducer를 사용해야 할 때

1. **복잡한 상태 로직**: 상태 업데이트 로직이 복잡하거나 여러 상태를 관리해야 할 때.
2. **다양한 상태 전이**: 다양한 액션에 따라 상태가 여러 가지로 변할 때.
3. **상태 관리의 중앙화**: 여러 컴포넌트에서 상태를 공유해야 할 때.

예제: 간단한 투두 앱

아래는 useReducer를 사용한 간단한 투두(Todo) 앱의 예제입니다.

import { useReducer, useState } from 'react';

const initialState = [];

function reducer(state, action) {

switch (action.type) {

case 'add':

return [...state, { id: Date.now(), text: action.text }];

case 'remove':

return state.filter(todo => todo.id !== action.id);

default:

throw new Error();

}

}

function TodoApp() {

const [todos, dispatch] = useReducer(reducer, initialState);

const [text, setText] = useState('');

return (

<div>

<h1>Todo App</h1>

<input

value={text}

onChange={(e) => setText(e.target.value)}

/>

<button onClick={() => {

dispatch({ type: 'add', text });

setText('');

}}>Add Todo</button>

<ul>

{todos.map(todo => (

<li key={todo.id}>

{todo.text}

<button onClick={() => dispatch({ type: 'remove', id: todo.id })}>

Remove

</button>

</li>

))}

</ul>

</div>

);

}

export default TodoApp;

결론

useReducer는 리액트에서 상태 관리를 보다 구조화하고 효율적으로 할 수 있게 해주는 강력한 훅이다.

상태 로직이 복잡해지거나 여러 컴포넌트에서 공유될 때 useReducer를 활용하면 코드의 유지보수성과 확장성을 높일 수 있다.