React 13: Use Reducer Hook

Yuvaraj S · Follow

3 min read · Jul 22, 2024

> ***PREVIOUS BLOG:***
> *React 12 : Context API with Custom hook*

In this lesson we are going understand about useReducer hook, When building complex React applications, managing state can become challenging. While the `useState` hook is suitable for simple state management, it often falls short when the state logic becomes intricate. This is where the `useReducer` hook shines. In this blog, we'll dive deep into `useReducer`, understand its benefits, and explore practical examples to see it in action.

What is `useReducer` ?

`useReducer` is a hook that is typically used for managing more complex state logic in React. It is inspired by Redux, a popular state management library, and follows a similar pattern of dispatching actions to update the state.

Why Use `useReducer`?

- **Complex State Logic:** When state transitions are complex, `useReducer` helps organize the logic in a predictable manner.
- **Predictability:** State updates are handled by pure functions called reducers, which makes the state transitions predictable.
- **Scalability:** It is easier to manage and scale complex state logic with `useReducer` compared to `useState`.

Basic Usage of `useReducer`

Here's the basic syntax of `useReducer`:

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

- `reducer`: A function that determines how the state should change based on an action.
- `initialState`: The initial value of the state.
- `state`: The current state.
- `dispatch`: A function that you call with an action to trigger a state change.

Example: Counter with `useReducer`

Let's start with a simple example of a counter:

Define the initial state and the reducer function:

```
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 };
    case 'reset':
      return initialState;
    default:
      throw new Error('Unknown action type');
    }
  }
```

Create the component using `useReducer`:

```
import React, { useReducer } from 'react';

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</buttor
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</buttor
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </div>
  );
};
export default Counter;
```

In this example, the `reducer` function handles three action types: `increment`, `decrement`, and `reset`. The `dispatch` function is used to send actions to the reducer.

Example: Todo List with `useReducer`

Now, let’s consider a more complex example: a todo list.

Define the initial state and the reducer function:

```
const initialState = { todos: [] };

function reducer(state, action) {
  switch (action.type) {
    case 'add':
      return { todos: [...state.todos, action.payload] };
    case 'remove':
      return { todos: state.todos.filter(todo => todo.id !== action.payload.id)
    case 'toggle':
      return {
        todos: state.todos.map(todo =>
          todo.id === action.payload.id ? { ...todo, completed: !todo.completec
        ),
      };
    default:
      throw new Error('Unknown action type');
  }
}
```

Create the component using `useReducer`:

```
import React, { useReducer, useState } from 'react';

const TodoApp = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  const [input, setInput] = useState('');
  const addTodo = () => {
    dispatch({ type: 'add', payload: { id: Date.now(), text: input, completed:
    setInput('');
  };
  return (
    <div>
      <input value={input} onChange={(e) => setInput(e.target.value)} />
      <button onClick={addTodo}>Add Todo</button>
      <ul>
        {state.todos.map((todo) => (
          <li key={todo.id}>
            <span style={{ textDecoration: todo.completed ? 'line-through' : 'r
              {todo.text}
            </span>
            <button onClick={() => dispatch({ type: 'toggle', payload: { id: tc
              Toggle
            </button>
            <button onClick={() => dispatch({ type: 'remove', payload: { id: tc
              Remove
            </button>
          </li>
        ))}
      </ul>
    </div>
  );
};
export default TodoApp;
```

In this example, the `reducer` function manages adding, removing, and toggling the completion state of todos. The `TodoApp` component uses `useReducer` to handle these actions and `useState` to manage the input field.

Tips for Using `useReducer`

- **Keep Reducers Pure:** Reducers should be pure functions. They should not have side effects or depend on external state.
- **Use Action Types:** Define action types as constants to avoid typos and improve maintainability.
- **Combine Reducers:** For large applications, consider breaking down reducers into smaller, more manageable functions and combining them.

> *STACKBLITZ LINK:*
> *https://stackblitz.com/edit/react-use-reducer-hooks*

Conclusion

`useReducer` is a powerful hook for managing complex state logic in React applications. By separating state transitions into pure reducer functions and using the `dispatch` function to trigger actions, you can create predictable and scalable state management solutions. Whether you're building a simple counter or a complex todo list, `useReducer` provides a robust alternative to `useState` that can help you manage your state more effectively.

React Hooks | React | Reactjs | Code Yuvaraj | JavaScript

Follow

Written by Yuvaraj S

97 followers · 14 following

Frontend Developer with 6years of experience, I write blogs that are [Easy words + Ground Level Code + Live Working Link]. Angular | React | Javascript | CSS

No responses yet