useCallback, useMemo, useReducer

RUJUAN XING

useCallback

Problem

One reason to use useCallback is to prevent a component from re-rendering unless its props have changed.

In this example, you might think that the Todos component will not re-render unless the todos change:

```
type Props = {
  todos: string[],
  onAddtodo: () => void
}

export default function Todos(props: Props) {
  console.log('inside Todos Components');
  const {todos, onAddtodo} = props;
  return (
    <div>
        {todos.map((todo, index) => <p key={index}>{todo}</p>)}
        <button onClick={onAddtodo}>Add Todo</button>
    </div>
  )
}
```

Demo (Cont.)

```
export default function CallbackHook() {
    const [count, setCount] = useState(0);
    const [todos, setTodos] = useState<string[]>([]);

    const increment = () => {
        setCount(count + 1);
    }

    const addTodo = () => {
        setTodos([...todos, 'New Todo']);
    }

    return (
        <div>
            <p>Count: {count}<button onClick={increment}>+1</button></p>
            <hr />
            <Todos todos={todos} onAddtodo={addTodo}/>
        </div>
    )
}
```

React Memo

memo lets you skip re-rendering a component when its props are unchanged.

```
const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
```

Parameters

Component: The component that you want to memoize. The memo does not modify this component, but returns a new, memoized component instead

optional arePropsEqual: A function that accepts two arguments: the component's previous props, and its new props. It should return true if the old and new props are equal: that is, if the component will render the same output and behave in the same way with the new props as with the old. Otherwise it should return false. Usually, you will not specify this function. By default, React will compare each prop with Object.is.

Returns

memo returns a new React component. It behaves the same as the component provided to memo except that React will not always re-render it when its parent is being re-rendered unless its props have changed.

Still render?

```
import { memo } from "react";

type Props = {
  todos: string[],
  onAddtodo: () => void
}

function Todos(props: Props) {
  console.log('inside Todos Components');
  const {todos, onAddtodo} = props;
  return (
    <div>
        {todos.map((todo, index) => <p key={index}>{todo}</p>)}
        <button onClick={onAddtodo}>Add Todo</button>
    </div>
  )
}

export default memo(Todos);
```

Why Todos componented re-rendered?

This is because of something called "referential equality".

Every time a component re-renders, its functions get recreated. Because of this, the addTodo function has actually changed.

Callback Hook - `useCallback`

The React `useCallback` Hook returns a memoized callback function.

This allows us to isolate resource intensive functions so that they will not automatically run on every render.

The `useCallback` Hook **only runs when one of its dependencies update** which can improve performance.

```
const cachedFn = useCallback(fn, dependencies)
```

- `fn`: The function value that you want to cache.
- `dependencies`: The list of all reactive values referenced inside of the `fn` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the `Object.is` comparison algorithm.

- `Return:`
  - On the initial render, useCallback returns the fn function you have passed.
  - During subsequent renders, it will either return an already stored fn function from the last render (if the dependencies haven't changed), or return the fn function you have passed during this render.

Demo

```
export default function CallbackHook() {
    const [count, setCount] = useState(0);
    const [todos, setTodos] = useState<string[]>([]);

    const increment = () => {
        setCount(count + 1);
    }

    const addTodo = useCallback(() => {
        setTodos([...todos, 'New Todo']);
    }, [todos]);

    return (
        <div>
            <p>Count: {count}<button onClick={increment}>+1</button></p>
            <hr />
            <Todos todos={todos} onAddtodo={addTodo} />
        </div>
    )
}
```

useMemo

Memo Hook - `useMemo`

The React `useMemo` Hook returns a memoized value.

The `useMemo` Hook only runs when one of its dependencies update which can improve performance.

```
const cachedValue = useMemo(calculateValue, dependencies)
```

- `calculateValue`: The function calculating the value that you want to cache. It should be pure, should take no arguments, and should return a value of any type. React will call your function during the initial render. On next renders, React will return the same value again if the dependencies have not changed since the last render. Otherwise, it will call `calculateValue`, return its result, and store it so it can be reused later.
- `dependencies`: The list of all reactive values referenced inside of the `fn` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the `Object.is` comparison algorithm.
- Returns
  - On the initial render, `useMemo` returns the result of calling `calculateValue` with no arguments.
  - During next renders, it will either return an already stored value from the last render (if the dependencies haven't changed), or call `calculateValue` again, and return the result that `calculateValue` has returned.

Demo

The useMemo Hook can be used to keep expensive, resource intensive functions from needlessly running.

Example: we have an expensive function that runs on every render. When changing the count or adding a todo, you will notice a delay in execution.

```
export default function MemoHook() {
  const [todoList, setTodoList] = useState([]);
  const [count, setCount] = useState(0);

  const expensiveCalculation = useCallback(num => {
    console.log("Calculating...");
    for (let i = 0; i < 1000000000; i++) {
      num += 1;
    }
    return num;
  },[]);

  const calculation = useMemo(() => expensiveCalculation(count), [count]);
  // const calculation = expensiveCalculation(count);
  const addTodo = () => {
    setTodoList([...todoList, 'new List']);
  }
```

```
  const increaseByOne = () => {
      setCount(count + 1);
    }

    return (
      <div>
        <h1>Todo List</h1>
        {todoList.map((todo, index) => <p key={index}>{todo}</p>)}
        <button onClick={addTodo}>Add a Todo</button>
        <hr/>
        <h1>Count: {count}
          <button onClick={increaseByOne}>+1</button>
        </h1>
        <hr/>
        <h1>Expensive Calculation</h1>
        Result: {calculation}
      </div>
    );
}
```

useReducer

React Hook: useReducer

The `useReducer` Hook is similar to the `useState` Hook.

It allows for custom state logic.

If you find yourself keeping track of multiple pieces of state that rely on complex logic, `useReducer` may be useful.

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

The `reducer` function contains your custom state logic and
the `initialState`can be a simple value but generally will contain an object.
The `useReducer` Hook returns the current `state`and a `dispatch`method.

How to use useReducer() ?

Start with creating a reducer function.

- A reducer function takes current state and action as input and returns a new state.


```
const initialState = 0;

function reducer(state: number, action: { type: string, data:
number }) {
    const { type, data } = action;
    switch (type) {
        case 'increment':
            return state + data;
        case 'decrement':
            return state - data;
        case 'reset':
            return 0;
        default:
            return initialState;
    }
}
```

How to use useReducer() ? (Cont.)

After reducer function is created, we can use useReducer as follows

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

Whenever `dispatch(<SOME_ACTION>)` is called, it will invoke reducer function to generate a new state and populate variable with new State value.

Demo

```
export default function ReducerHook() {
    const [state, dispatch] = useReducer(reducer, initialState);
    return (
        <div>
            Count: {state}
            <br />
            <button onClick={() => dispatch({type: 'increment', data: 6})}>+6</button>
            <button onClick={() => dispatch({type: 'decrement', data: 2})}>-2</button>
            <button onClick={() => dispatch({type: 'reset', data: 0})}>Reset</button>
        </div>
    )
}
```