27.如何用 useContext+ useReducer实现一个轻量级的状态管理器?

Q1: 请解释一下 React 中的 `useContext` 是什么，它主要解决了什么问题?

A1:
`useContext` 是 React 官方提供的一个 Hook，它能让我们在组件树中实现跨层级的依赖注入和数据传递。它主要解决了 **"Prop Drilling"（属性钻探）** 的问题。所谓 Prop Drilling，就是指当组件层级很深时，一些全局或上层的状态需要通过 props 一层一层地手动传递给底层的子组件，即使中间的很多组件本身并不需要这些数据。这个过程非常繁琐，且难以维护。
`useContext` 通过创建一个“上下文（Context）”，允许上层组件（Provider）提供数据，任何层级的下层组件都可以直接“消费”或获取这些数据，从而避免了逐层传递的麻烦。

Q2: 什么是 `useReducer`？在什么情况下你会选择使用 `useReducer` 而不是 `useState`？

A2:
`useReducer` 是 React 提供的另一个内置 Hook，它可以被看作是 `useState` 的替代方案，尤其适用于管理更复杂的状态逻辑。它通过一个 `reducer` 函数来集中处理所有的状态更新。

根据讲义内容，我会在以下情况下选择使用 `useReducer`：

1. **状态逻辑复杂**：当一个状态值是包含多个子值的对象或数组，并且更新逻辑比较复杂时，使用 `useReducer` 可以将更新逻辑抽离到 `reducer` 函数中，使组件代码更清晰。
2. **下一个状态依赖于前一个状态**：当新的状态需要基于前一个状态进行计算时，`useReducer` 能提供更可靠和可预测的状态更新。
3. **集中管理状态更新**：`useReducer` 将相关的状态更新逻辑统一放在一个 `reducer` 函数中，通过派发（dispatch）不同类型的 `action` 来执行不同的更新，这让状态管理更有组织性，也更易于调试和维护。

Q3: 请阐述一下为什么将 `useContext` 和 `useReducer` 结合使用是一种有效的轻量级状态管理方案，并说明其主要优势。

A3:
将 `useContext` 和 `useReducer` 结合使用是一种强强联合的模式：

- **`useReducer` 负责管理状态逻辑**：它提供了一个集中的地方（`reducer` 函数）来处理所有复杂的状态变更。通过 `dispatch` 一个 `action` 来触发更新，使得状态的变更过程变得清晰、可预测。

- **`useContext` 负责全局注入**：它扮演了“依赖注入”的角色，将 `useReducer` 创建的 `state`（状态）和 `dispatch`（派发函数）无缝地、跨层级地传递给所有需要的子组件，避免了 props 的逐层传递。

这种组合方案的主要优势在于：

1. **轻量级**：完全基于 React 内置的 API，无需引入任何第三方状态管理库（如 Redux），减小了项目的体积和依赖。
2. **职责清晰**：成功地将状态管理逻辑（`useReducer`）与 UI 视图组件分离开来，降低了耦合度。
3. **可维护性高**：对于中小型应用，它的代码结构清晰，逻辑集中，比 Redux 等更复杂的库更容易上手和后期维护。

Q4: 在讲义的代码示例中，实现计数器的 `store.js` 包含了几个核心部分。请分别解释 `reducer` 函数、`CounterProvider` 组件和 `useCounter` 自定义 Hook 的作用。

A4:
这三个核心部分的作用如下：

1. **`reducer` 函数**：这是状态管理的核心逻辑所在。它是一个纯函数，接收当前的 `state` 和一个 `action` 对象作为参数。函数内部通过 `switch` 语句判断 `action.type` 的类型（如 'INCREMENT', 'DECREMENT', 'RESET'），然后根据类型计算并返回一个全新的状态对象（`newState`）。它定义了状态“如何”变化。
2. **`CounterProvider` 组件**：这是一个关键的封装组件。它的内部通过调用 `useReducer(reducer, initialState)` 来初始化状态并获取 `state` 和 `dispatch` 函数。然后，它利用 `CounterContext.Provider` 将一个包含 `{ state, dispatch }` 的对象作为 `value` 属性，提供给它的所有子组件（`children`）。它的作用是连接状态逻辑和需要使用状态的组件树。
3. **`useCounter` 自定义 Hook**：这是一个推荐使用的辅助 Hook，目的是为了简化在业务组件中的使用。它内部封装了 `useContext(CounterContext)` 的调用，直接返回 `Provider` 提供的 `{ state, dispatch }` 对象。同时，它还增加了一个检查，确保该 Hook 必须在 `CounterProvider` 的包裹下使用，否则会抛出错误。这让业务组件获取状态和派发函数变得非常简单（只需一行 `const { state, dispatch } = useCounter();`），提升了代码的简洁性和健壮性。

Q5: 如果让你设计一个全局的主题切换功能（如暗黑/明亮模式），你会如何使用 `useContext` + `useReducer` 模式来实现？请简要描述你的实现思路。

A5:
我会遵循 `useContext` + `useReducer` 模式的核心步骤来设计这个功能：

1. **定义 `initialState` 和 `reducer`**：
   - `initialState` 会是 `{ theme: 'light' }`，表示默认是明亮模式。
   - `reducer` 函数会处理一个 `action`，例如 `{ type: 'TOGGLE_THEME' }`。当收到这个 action 时，它会判断当前 `state.theme` 是 'light' 还是 'dark'，然后返回一个新状态，将 `theme` 切换到另一个值。也可以设计一个 `{ type: 'SET_THEME', payload: 'dark' }` 这样的 action 来直接设置主题。
2. **创建 `ThemeContext`**：
   - 使用 `const ThemeContext = React.createContext()` 创建一个用于共享主题状态的上下文。
3. **创建 `ThemeProvider` 组件**：
   - 在这个组件内部，使用 `const [state, dispatch] = useReducer(reducer, initialState)` 获取主题状态和派发函数。
   - 通过 `<ThemeContext.Provider value={{ state, dispatch }}>` 将它们提供给所有子组件。
4. **创建自定义 Hook `useTheme`**：
   - 为了方便使用，创建一个 `useTheme` Hook，它内部调用 `useContext(ThemeContext)` 并返回 `state` 和 `dispatch`。
5. **在应用中使用**：
   - 在应用的根组件（如 `App.js`）用 `<ThemeProvider>` 将整个应用包裹起来。
   - 在需要根据主题改变样式的组件中，通过 `const { state } = useTheme()` 获取当前主题 `state.theme` (e.g., 'dark' 或 'light')，并应用对应的 CSS 类或样式。
   - 在切换按钮组件中，通过 `const { dispatch } = useTheme()` 获取派发函数，在按钮点击时调用 `dispatch({ type: 'TOGGLE_THEME' })` 来更新全局主题。

Q6: 讲义中提到，`useContext` 的一个潜在问题是，当 `value` 变化时，所有消费该 Context 的组件都会重新渲染。针对这个性能问题，你可以采取哪些优化策略?

A6:
针对 `useContext` 可能导致的性能问题，可以采取以下几种优化策略：

1. **使用 `React.memo`**： 对于消费 Context 但其渲染结果并不依赖于本次 Context 变化的子组件，可以使用 `React.memo` 将其包裹起来。`React.memo` 会对 props 进行浅比较，如果 props 没有变化，它会复用上一次的渲染结果，从而避免不必要的重渲染。当 `dispatch` 函数是稳定的时候（React 保证 `useReducer` 返回的 `dispatch` 函数是稳定的），这对于只消费 `dispatch` 而不消费 `state` 的组件尤其有效。
2. **拆分 Context**：如果一个 Context 中包含了多个相互独立或不常一起变化的状态值，可以考虑将其拆分成多个更细粒度的 Context。例如，一个 Context 用于提供不常变化的 `dispatch` 函数，另一个 Context 用于提供会频繁变化的 `state`。这样，只依赖 `dispatch` 的组件就只会订阅 `DispatchContext`，当 `state` 变化时，这些组件就不会因为 `StateContext` 的 `value` 变化而重新渲染。