31.为什么 Immer.js 在 Redux 生态中如此重要?

Q1: 请简要介绍一下 Immer.js 是什么，以及它主要解决了 Redux 生态中的什么问题?
A1: Immer.js 是一个用于处理不可变数据状态的库。它主要解决了在 Redux 中进行不可变更新时的核心痛点：当状态对象（尤其是深层嵌套的对象）需要更新时，传统方式需要编写大量复杂的、易出错的展开运算符（spread operator）样板代码。Immer 允许开发者用看似“可变”的、直观的方式来修改状态，而它会在背后自动生成一个正确、高效的不可变的新状态，从而极大地简化了代码，提高了开发效率和可读性。

Q2: 为什么 Redux 强调状态的“不可变性”（Immutability）？它有哪些具体的好处?
A2: Redux 强调不可变性主要有三个好处：

1. **时间旅行调试**：由于每个状态都是一个独立的对象，不会被修改，因此可以轻松地在应用的不同历史状态之间切换，这为调试（如 Redux DevTools）提供了可能。
2. **性能优化**：当状态没有被直接修改时，可以通过高效的浅比较（shallow comparison）快速检测到状态是否发生了变化。这在 React 等视图库中可以用来避免不必要的组件重渲染。
3. **可预测性**：状态的更新逻辑变得清晰和可预测。因为状态的引用地址在更新时一定会改变，所以很容易追踪状态是在何时、何处、因为什么 action 而改变的。

Q3: 请解释一下 Immer.js 的核心工作原理，特别是它如何做到“用可变的方式更新不可变数据”?
A3: Immer.js 的核心工作原理基于 **Copy-on-Write（写时复制）**机制，并通过 JavaScript 的 **Proxy** 对象实现。

1. **创建草稿状态（Draft State）**：当你调用 `produce(currentState, recipe)` 时，Immer 不会直接让你操作原始状态 `currentState`。相反，它会为 `currentState` 创建一个 Proxy 代理对象，这个代理对象就是所谓的“草稿状态”（draft）。
2. **拦截修改操作**：你可以在 `recipe` 函数中像操作普通 JavaScript 对象一样直接修改这个 `draft` 对象（例如 `draft.user.name = 'new name'`）。所有这些修改操作都会被 Proxy 拦截。
3. **生成最终状态**：当 `recipe` 函数执行完毕后，Immer 会检查你对 `draft` 做了哪些修改。它会根据这些修改，创建一个新的不可变对象。对于未被修改的部分，Immer 会进行**结构共享**，即直接复用原始状态中的对应部分，从而优化性能和内存使用。

Q4: 在使用 Immer 的 `produce` 函数时，我们可以在 recipe (处理函数) 中直接使用像 `array.push()` 或 `array.splice()` 这样会修改原数组的方法。为什么这是安全的?
A4: 这是安全的，因为我们操作的不是原始的状态数组，而是 Immer 创建的草稿状态（draft）的代理。当我们对这个 draft 数组调用 `push()` 或 `splice()` 等方法时，Immer 的 Proxy 会拦截这些操作，并记录下这些“突变”意图。在函数执行结束后，Immer 会根据这些记录，生成一个全新的、包含了这些更新的数组，而原始的状态数组完全不会受到影响，从而保证了不可变性。

Q5: 在 Redux Toolkit 中，Immer 是如何被集成的？我们在使用 `createSlice` 时还需要手动调用 `produce` 函数吗?

A5: 在 Redux Toolkit 中，Immer 是被**内置集成**的，尤其是在 `createSlice` 和 `createReducer` 这两个核心 API 中。

因此，当你在 `createSlice` 的 `reducers` 选项中定义 case reducer 时，你**不需要**手动调用 `produce` 函数。你可以直接在 reducer 函数内部修改 `state` 参数，就好像它是一个可变对象一样。Redux Toolkit 会在底层自动为你调用 Immer，处理所有的不可变更新逻辑。

Q6: 假设有以下 Redux 状态，请分别用“传统方式”和“Immer 方式”编写一个 reducer case，用于更新特定用户的街道地址。

```
const initialState = {
  users: {
    '123': {
      name: 'John',
      profile: {
        address: {
          street: 'Old Street',
          city: 'Beijing'
        }
      }
    }
  }
};
const action = { type: 'UPDATE_STREET', userId: '123', newStreet: 'New
Street' };
```

A6:

传统方式 (使用展开运算符):

```
function traditionalReducer(state = initialState, action) {
  if (action.type === 'UPDATE_STREET') {
    return {
      ...state,
      users: {
        ...state.users,
        [action.userId]: {
          ...state.users[action.userId],
          profile: {
            ...state.users[action.userId].profile,
            address: {
              ...state.users[action.userId].profile.address,
              street: action.newStreet
            }
          }
        }
      }
```

```
    };
  }
  return state;
}
```

Immer 方式 (使用 `produce`):

```
import produce from 'immer';

const immerReducer = (state = initialState, action) =>
  produce(state, draft => {
    if (action.type === 'UPDATE_STREET') {
      draft.users[action.userId].profile.address.street =
action.newStreet;
    }
  });
```

Q7: 在 Immer 的 producer 函数中，关于修改 `draft` 和返回值，有一个重要的使用规则是什么？请举例说明正确和错误的用法。
A7: 规则是：**要么修改 `draft` 对象（此时不要有返回值，或者说返回 `undefined`），要么返回一个全新的状态对象（此时不要修改 `draft`）。绝不能同时修改 `draft` 又返回一个新的对象。**

- ✅ **正确用法1 (修改 draft):**

```
const correctReducer1 = produce((draft, action) => {
  draft.value += 1;
  // 无返回值 (隐式返回 undefined)
});
```

- ✅ **正确用法2 (返回新状态):**

```
const correctReducer2 = produce((draft, action) => {
  if (action.type === 'RESET') {
    return { value: 0 }; // 返回一个全新的状态对象
  }
});
```

- ❌ **错误用法 (既修改又返回):**

```
const wrongReducer = produce((draft, action) => {
  draft.value += 1; // 修改了 draft
  return { ...draft, otherProp: true }; // 又返回了一个新对象，这会导致
```

```
  Immer 抛出错误。
});
```

Q8: 除了简化代码，Immer 的“结构共享”机制还能带来什么好处?
A8: Immer 的“结构共享”（Structural Sharing）机制主要带来**性能上的好处**。当状态树的某一部分没有被修改时，Immer 不会创建新的对象，而是会复用原始状态中对应的对象引用。这意味着在进行状态比较时（例如在 React 组件的 `shouldComponentUpdate` 或 `React.memo` 中），对于未改变的状态分支，可以通过 `===` 进行快速的浅比较，发现引用没有变化，从而避免了不必要的深度比较和组件重渲染，提升了应用性能。