12.什么时候应该使用 `useCallback` 和 `useMemo`？滥用有什么后果?

面试题与参考答案

主题：`useCallback` 和 `useMemo` 的使用时机与滥用后果

一、基础知识与定义

Q1: 请解释一下 React 中的 `useMemo` 是做什么的？它主要解决了什么问题?
A1:
`useMemo` 是 React 提供的一个性能优化 Hook。它的主要作用是**缓存计算结果（值）**。
它解决了以下问题：

1. 避免在每次组件渲染时都进行不必要的、开销较大的重复计算。特别是在处理大数据量的排序、过滤或复杂转换等场景。
2. 当需要向子组件（尤其是用 `React.memo` 优化的子组件）传递对象或数组类型的 props 时，`useMemo` 可以缓存这些引用，确保只有在依赖项真正改变时才创建新的对象/数组实例，从而防止子组件因为 props 引用变化而进行不必要的重新渲染。

Q2: 那么 `useCallback` 呢？它的主要作用是什么？
A2:
`useCallback` 也是 React 的一个性能优化 Hook。它的主要作用是**缓存函数实例（函数的引用）**。
它解决了以下问题：

1. 当一个函数作为 prop 传递给经过优化的子组件（如用 `React.memo` 包裹的子组件）时，`useCallback` 可以防止因为父组件重新渲染导致该函数被重新创建（即使函数体不变），从而避免子组件因 prop 引用变化而不必要地重新渲染。
2. 当回调函数本身是另一个 Hook（如 `useEffect`，`useMemo` 等）的依赖项时，使用 `useCallback` 可以确保这个函数依赖项的稳定性，避免因为函数引用在每次渲染中都变化而导致其他 Hook 不必要地重复执行。

Q3: `useMemo` 和 `useCallback` 都接收一个依赖项数组。这个依赖项数组的作用是什么？如果传入一个空数组 `[]` 作为依赖项，意味着什么?
A3:
依赖项数组（dependency array）的作用是告诉 React 什么时候需要重新计算 `useMemo` 的值或重新创建 `useCallback` 的函数实例。

- **对于 `useMemo`**：只有当依赖项数组中的至少一个值发生变化时，`useMemo` 才会重新执行其“创建”函数来得到新的结果。

- **对于 `useCallback`**：只有当依赖项数组中的某个值发生变化时，`useCallback` 才会返回一个新的函数实例。

如果传入一个**空数组 `[]`** 作为依赖项：

- 对于 `useMemo`：它包裹的计算函数只会在组件初始渲染时执行一次，之后会一直返回第一次计算得到的缓存值，除非组件被卸载。
- 对于 `useCallback`：它包裹的函数实例只会在组件初始渲染时创建一次，之后该函数的引用将保持不变，除非组件被卸载。这通常用于那些不依赖于组件作用域内任何可变值的函数，或者像 `setState` 这种 React 保证其引用稳定的函数。

二、理解与阐释

Q4: 为什么在父组件中定义的函数或对象，即使其内容（逻辑或数据）没有改变，当父组件重新渲染时，它们在传递给用 `React.memo` 优化的子组件时，仍可能导致子组件重新渲染?
A4:
这是因为 JavaScript 中，函数和对象（包括数组）是**引用类型**。
当父组件重新渲染时，即使一个函数或对象的逻辑或内容看起来一模一样，它们通常会在父组件的每次渲染周期中被重新创建。这意味着它们在内存中的引用地址是不同的。
`React.memo` 默认对子组件的 props 进行的是浅比较。浅比较会检查 props 的引用是否相等。由于父组件每次渲染都可能创建新的函数或对象引用，即使内容未变，`React.memo` 也会认为 prop 发生了变化，从而导致子组件重新渲染。

Q5: 你是如何理解 `useCallback(fn, deps)` 和 `useMemo(() => fn, deps)` 之间的关系的?
A5:
从功能上讲，`useCallback(fn, deps)` 是等价于 `useMemo(() => fn, deps)` 的。
`useCallback(fn, deps)` 可以看作是 `useMemo(() => fn, deps)` 的一个语法糖，它专门用于更清晰地表达我们正在记忆化一个函数本身（即函数的引用）。React 团队提供 `useCallback` 主要是为了语义上的清晰，让开发者一眼就能看出这里优化的目标是一个回调函数。

Q6: 在面试中，你会如何向面试官简明扼要地解释 `useMemo` 的核心价值和使用场景?
A6:
“面试官你好，`useMemo` 是 React 提供的一个性能优化 Hook。它的主要作用是**缓存计算结果**，避免在每次渲染时都进行不必要的、开销较大的重复计算。

它接收一个函数和一个依赖项数组。只有当依赖数组中的项发生变化时，它才会重新执行该函数并返回新的值；否则，它会返回上一次缓存的值。

`useMemo` 在两种场景下特别有用：

1. **进行昂贵的计算**：比如对大数据进行复杂处理，可以避免每次渲染都重新计算。
2. **缓存对象或数组的引用**：当我们想传递一个引用稳定的对象或数组给用 `React.memo` 优化的子组件时，可以用 `useMemo` 来缓存这个对象或数组的实例，从而防止子组件因为父组件不必要的 prop 引用变化而重新渲染。”

Q7: 类似地，你会如何向面试官简明扼要地解释 `useCallback` 的核心价值和使用场景?
A7:
“面试官你好，`useCallback` 也是 React 的一个性能优化 Hook。它主要用于**缓存函数实例**。

它接收一个内联回调函数和一个依赖项数组。`useCallback` 会返回该回调函数的 memoized (记忆化) 版本，这个回调函数仅在某个依赖项改变时才会更新它的引用。

这在以下场景中非常有用：

1. **将回调传递给经过优化的子组件**：特别是那些用 `React.memo` 包裹的子组件，它们依赖于 props 的引用相等性来防止不必要的渲染。
2. **当回调函数本身是另一个 Hook (如 `useEffect`，`useMemo` 等) 的依赖项时**：使用 `useCallback` 可以确保这个函数依赖项的稳定性，避免因为函数引用在每次渲染中都变化而导致其他 Hook 不必要地重复执行。”

三、应用与解决问题能力

Q8: 假设你有一个列表组件，它接收一个大的 `items` 数组和一个 `filterTerm` 字符串作为 props，然后需要根据 `filterTerm` 对 `items` 进行过滤并显示。如果这个过滤操作比较耗时，你会如何使用 `useMemo` 进行优化？请描述关键的实现思路。
A8:
我会使用 `useMemo` 来缓存过滤后的列表结果。关键思路如下：

1. 将过滤逻辑封装在一个函数中，例如 `filterItems(allItems, term)`。
2. 在组件内部，使用 `useMemo` 来调用这个过滤函数并存储其结果。

```
const visibleItems = useMemo(() => {
  console.log('Filtering items...'); // 用于观察执行频率
  return filterItems(items, filterTerm);
}, [items, filterTerm]); // 依赖项是 items 和 filterTerm
```

3. `useMemo` 的第一个参数是执行过滤操作的函数，它返回过滤后的列表。
4. `useMemo` 的第二个参数是依赖项数组 `[items, filterTerm]`。
   这样，只有当 `items` 数组或 `filterTerm` 字符串发生变化时，过滤操作才会重新执行。如果组件因为其他不相关的 state 更新而重新渲染，但 `items` 和 `filterTerm` 没变，那么 `visibleItems` 会直接使用上次缓存的结果，避免了昂贵的重复过滤。

Q9: 考虑一个场景：父组件 `App` 有一个状态 `userId`，它将一个 `handleUserAction` 函数传递给一个用 `React.memo` 包裹的子组件 `UserActionsButton`。`handleUserAction` 函数内部逻辑依赖于当前的 `userId`。如果父组件因为其他状态（例如一个计数器 `count`）更新而频繁重新渲染，如何使用 `useCallback` 来优化，以防止 `UserActionsButton` 不必要地重渲染?
A9:
可以使用 `useCallback` 来包裹 `handleUserAction` 函数，并将其依赖的 `userId` 加入到依赖项数组中。

```jsx
const App = () => {
  const [count, setCount] = useState(0);
  const [userId, setUserId] = useState(1);

  const handleUserAction = useCallback(() => {
    console.log(`Handling user action for user: ${userId}`);
    // ... 基于 userId 的操作
  }, [userId]); // 依赖 userId

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(c => c + 1)}>Increment
Count</button>
      <button onClick={() => setUserId(id => id + 1)}>Change User</button>
      <UserActionsButton onAction={handleUserAction} />
    </div>
  );
};
```

这样，`handleUserAction` 函数的引用只有在 `userId` 发生变化时才会更新。如果 `App` 组件因为 `count` 状态变化而重新渲染，但 `userId` 保持不变，那么传递给 `UserActionsButton` 的 `onAction` prop（即 `handleUserAction`）的引用将保持稳定，`UserActionsButton` 就不会因为这个回调函数的变化而重新渲染。

Q10: 过度使用或滥用 `useMemo` 和 `useCallback` 会带来哪些潜在的问题或副作用?
A10:
滥用 `useMemo` 和 `useCallback` 主要会带来两方面的问题：

1. **自身的性能开销**：
   - `useMemo` 和 `useCallback` 本身也需要执行。它们需要比较依赖项数组中的每一项是否发生变化。
   - 它们需要存储缓存的值或函数，这会占用一定的内存。
   - 如果被优化的计算或函数本身开销非常小，或者依赖项非常复杂且频繁变化，那么使用这些 Hook 的成本（包括依赖比较和内存占用）可能比它们带来的性能收益还要大，甚至可能导致轻微的性能下降。
2. **代码复杂度增加和可维护性下降**：
   - 过度使用会让代码变得更复杂，充斥着 `useMemo` 和 `useCallback` 的包裹，使得组件的逻辑不那么直观。
   - 这增加了阅读代码的心智负担，也使得后续的维护和理解更加困难。

Q11: 那么，在哪些情况下你认为可能不需要或者应该避免使用 `useMemo` 或 `useCallback`？你通常如何判断是否真的需要它们?
A11:
应该避免在以下情况下使用 `useMemo` 或 `useCallback`：

1. **对于那些开销本身就很小的计算或函数**：例如返回一个常量，或者一个非常简单的 JSX 结构，重新计算它们的成本几乎可以忽略不计。
2. **如果一个函数或值只在当前组件内部使用**：即没有作为 prop 传递给子组件（特别是 `React.memo` 包裹的子组件），也没有作为其他 Hook (如 `useEffect`) 的依赖项，那么通常也不需要优化它的引用稳定性。
3. **依赖项数组过于复杂或变化频繁**：如果依赖项非常多，或者它们变化得非常频繁，那么 `useMemo` / `useCallback` 每次都要比较这些依赖，其本身的开销可能就抵消了优化的好处，甚至可能比不使用时更差。

判断是否真的需要它们，核心原则是：

1. **性能分析 (Profiling) 是王道**：首先应该使用 React DevTools Profiler 等工具来实际定位性能瓶颈。找到那些渲染耗时较长，或者频繁不必要重新渲染的组件。
2. **经验法则/观察**：
   - 观察子组件是否因为 props (特别是对象或函数类型) 的引用变化而频繁且不必要地重新渲染，如果是，则考虑在父组件对这些 props 使用 `useMemo` 或 `useCallback`。
   - 观察 `useEffect` 是否因为其依赖数组中的函数引用频繁变化而导致不必要的重复执行，如果是，考虑对该函数使用 `useCallback`。
   - 如果一个计算逻辑确实非常耗时（例如，通过 `console.time` 测试发现需要几十毫秒以上），并且它的输入不经常变化时，可以考虑使用 `useMemo`。
3. **不要为了优化而优化**：优化的前提是存在已证实的性能问题，或者有充分理由相信某个部分会成为瓶颈（例如，传递给大量列表项的回调）。
