8. useEffect 依赖项的作用

面试题与参考答案：`useEffect` 依赖项

一、基础知识与核心概念

Q1: `useEffect` Hook 的主要作用是什么？它的依赖项数组（dependency array）的核心目的是什么？
A1:
`useEffect` Hook 允许我们在函数组件中执行副作用操作，例如数据获取、设置订阅、手动更改 DOM 等。
依赖项数组的核心目的，是精确控制副作用函数的执行时机。它告诉 React 在哪些数据发生变化时才需要重新运行这个 effect，从而帮助我们优化性能，避免不必要的计算或 API 调用，并确保 effect 内部能访问到最新的 props 和 state。

Q2: 如果在调用 `useEffect` 时不传递第二个参数（依赖项数组），副作用函数会在什么时候执行？这种写法的潜在风险是什么？
A2:
如果不传递依赖项数组，副作用函数会在**每一次组件渲染完成之后**执行。这包括了组件的首次渲染以及后续的每一次更新。
潜在风险是：

1. **性能问题**：如果组件频繁更新，副作用会频繁执行，可能导致不必要的计算和资源消耗。
2. **无限循环**：如果副作用内部又触发了组件的更新（例如调用了 `setState`），很容易导致无限循环。
   除非非常明确需要在每次渲染后都执行，否则一般不推荐这种写法。

Q3: 当我们给 `useEffect` 传递一个空数组 `[]` 作为依赖项时，副作用函数及其返回的清理函数（如果存在）分别会在什么时候执行？这种模式适合哪些场景？
A3:
当传递一个空数组 `[]` 作为依赖项时：

- 副作用函数只会在**组件首次渲染（mount）之后执行一次**。
- 如果副作用函数返回了一个清理函数，那么这个清理函数会在**组件卸载（unmount）时执行一次**。
  后续的组件更新将不会再次触发这个 effect。

这种模式非常适合那些只需要执行一次的设置类操作，比如：

- 获取初始数据。
- 设置定时器或者订阅（并在卸载时清除）。
- 添加全局事件监听器（并在卸载时移除）。

Q4: 当 `useEffect` 的依赖项数组包含具体依赖项时（例如 `[dep1, dep2]`），React 是如何判断是否需要重新执行副作用函数的？它使用的是什么比较算法？

A4:

当依赖项数组包含具体依赖项时，副作用函数会在组件首次渲染之后执行。并且，在后续的每一次渲染完成后，React 会检查这个数组中的每一个依赖项，与上一次渲染时的值进行比较。如果数组中**至少有一个依赖项发生了变化**，那么副作用函数就会被重新执行。

React 使用的是 `Object.is()` 算法来比较依赖项是否发生变化。

- 对于原始类型（如数字、字符串、布尔值），`Object.is()` 的行为和严格相等 `===` 基本一致（除了 `Object.is(NaN, NaN)` 为 `true`，以及对 `+0` 和 `-0` 的区分）。
- 对于引用类型（如对象、数组、函数），`Object.is()` 比较的是它们的**引用地址**。

Q5: 什么是“过时闭包”（Stale Closure）问题？它在 `useEffect` 中是如何产生的？请举例说明，并指出如何修正。

A5:

“过时闭包”是指 `useEffect` 的副作用函数“记住”了它首次执行时所依赖的组件作用域中的某个变量（props 或 state）的值。即使那个变量后来更新了，effect 内部拿到的依然是旧值，导致逻辑错误。

产生原因：如果 effect 函数内部使用到了组件作用域中的某个变量，但没有把它正确地添加到依赖项数组中。

举例：

```
function DelayedCounter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      // 此处闭包捕获了首次渲染时的 count（值为 0）
      console.log(`[Stale Closure] Interval tick, current count:
${count}`);
    }, 1000);

    return () => clearInterval(intervalId);
  }, []); // 依赖项为空，effect 和闭包只创建一次

  return (
    <div>
      <p>Actual Count: {count}</p>
      <button onClick={() => setCount(c => c + 1)}>Increment
Count</button>
    </div>
  );
}
```

在这个例子中，即使 `count` 状态通过按钮点击更新，控制台打印的 `count` 始终是 `0`。

修正方法：将 `count` 加入依赖项数组。

```
useEffect(() => {
  const intervalId = setInterval(() => {
    console.log(`Interval tick, current count: ${count}`);
  }, 1000);

  return () => clearInterval(intervalId);
}, [count]); // 将 count 添加到依赖项
```

这样当 `count` 改变时，旧的 interval 会被清除，新的 interval 会带着最新的 `count` 值被创建。

二、理解与阐释能力

Q6: 在面试中，如果要你阐述对 `useEffect` 依赖项数组的理解，你会从哪些方面来组织你的回答?
A6:
我会从以下几个方面来组织回答：

1. **核心作用**：首先点明依赖项数组的目的是精确控制副作用的执行时机，以优化性能和保证数据同步。
2. **三种情况的行为差异**：
   - 不提供依赖项数组：每次渲染后执行，需注意风险。
   - 提供空数组 `[]`：仅在挂载和卸载时执行，适用一次性操作。
   - 提供包含依赖项的数组 `[dep1, dep2]`：挂载后执行，并在任何依赖项变化后重新执行。
3. **关键概念和注意事项**：
   - **`Object.is()` 比较机制**：解释原始类型和引用类型的比较方式及其影响。
   - **过时闭包（Stale Closure）**：解释其成因、危害和如何避免（即正确填写依赖项）。
   - **ESLint 插件 `eslint-plugin-react-hooks`**：提及 `exhaustive-deps` 规则的重要性，帮助开发者避免遗漏依赖项。
4. **（加分项）相关优化手段**：
   - 提及 `useCallback` 配合处理函数依赖项。
   - 提及 `useMemo` 配合处理对象或数组依赖项。
   - 提及 `setState` 或 `dispatch` 函数的引用稳定性。

Q7: 为什么说对于引用类型（如对象、数组、函数）作为依赖项时需要特别注意？如果父组件每次渲染都传递一个新的对象（即使内容一样）给子组件，子组件中依赖此对象的 `useEffect` 会怎样?

A7:
需要特别注意是因为 React 使用 `Object.is()` 比较引用类型的依赖项时，比较的是它们的**引用地址**，而不是内容。
如果父组件在每次渲染时都创建一个新的对象或数组实例（即使其属性或元素与上一次渲染时完全相同），那么这个新的实例在内存中会有不同的引用地址。
当这个新对象作为 prop 传递给子组件，并且子组件的 `useEffect` 依赖于这个对象 prop 时，即使对象的内容没有实质性变化，`useEffect` 也会因为检测到引用地址的变化而重新执行其副作用函数。这可能导致不必要的计算、API 调用或 DOM 操作，影响性能。

三、应用与解决问题能力

Q8: 假设你在一个组件内部定义了一个函数 `fetchData`，并在 `useEffect` 中调用它。如果 `fetchData` 被列为 `useEffect` 的依赖项，可能会有什么问题？请提供至少两种解决方案。
A8:
潜在问题：
如果 `fetchData` 函数是在组件体内部定义的，那么每次组件渲染时，`fetchData` 都会是一个全新的函数实例（即它的引用地址会改变）。由于 `fetchData` 是 `useEffect` 的依赖项，这会导致 `useEffect` 在每次组件渲染后都重新执行，即使实际的获取逻辑或其依赖的数据（如 `query`）并没有改变，从而造成不必要的副作用执行。

解决方案：

1. **使用 `useCallback` 包裹 `fetchData`**：

```
const fetchData = useCallback((currentQuery) => {
  // API call logic using currentQuery
}, [dependenciesOfFetchData]); // 这里的依赖项是 fetchData 函数本身所依赖的 props 或 state

useEffect(() => {
  fetchData(query);
}, [query, fetchData]);
```

   `useCallback` 会返回一个记忆化版本的 `fetchData` 函数。这个函数只有在它自身的依赖项 (`dependenciesOfFetchData`) 改变时才会更新。这样，只要 `fetchData` 的依赖没有变，它的引用就保持稳定，`useEffect` 就不会因此而不必要地重执行。
2. **将函数定义在 `useEffect` 内部**（如果该函数只被这个 effect 使用）：

```
useEffect(() => {
  const localFetchData = (currentQuery) => {
    // API call logic using currentQuery
  };
```

```
  if (query) {
    localFetchData(query);
  }
}, [query]); // 依赖项现在只有 query
```

将函数定义在 `useEffect` 内部，它就不是来自组件作用域的变量，`useEffect` 的依赖项也就不需要包含它。`useEffect` 只需要依赖那些真正驱动副作用执行的数据（如 `query`）。

3. **将函数移到组件外部**（如果它不依赖组件的 props 或 state）：
   如果 `fetchData` 函数不依赖任何组件内的 props 或 state，可以将它定义在组件外部。这样它的引用就是永久稳定的，可以安全地加入依赖项数组（或如果它不接受任何参数，依赖项数组可能就不需要它了）。

Q9: 如果一个 `useEffect` 的依赖项是一个通过 props 传进来的配置对象（例如 `config = { theme: 'dark', timeout: 5000 }`），你如何优化以避免 `useEffect` 不必要的重执行?
A9:
问题分析：如果父组件在每次渲染时都创建一个新的 `config` 对象实例（即使内容完全一样），子组件的 `useEffect` 会因为 `config` 对象的引用变化而重新执行。

优化方案：

1. **父组件使用 `useMemo` 来稳定对象引用**：
   在父组件中，使用 `useMemo` 来创建和记忆化 `config` 对象。只有当 `useMemo` 自身的依赖项发生变化时，它才会返回一个新的对象实例。

```
// Parent.js
function Parent() {
  const theme = 'dark'; // or some state/prop
  const timeout = 5000; // or some state/prop
  const config = useMemo(() => ({ theme, timeout }), [theme,
timeout]);
  return <ChildComponent config={config} />;
}
```

2. **在子组件中解构对象的原始类型属性作为依赖项**（如果 effect 只关心对象的某些特定属性）：
   如果 `useEffect` 的逻辑只依赖于 `config` 对象中的某些原始类型属性，可以将这些属性解构出来，并把它们作为依赖项。

```
// ChildComponent.js
function ChildComponent({ config }) {
  const { theme, timeout } = config;
  useEffect(() => {
    // logic that uses theme and timeout
    console.log('Effect ran with theme:', theme, 'timeout:', timeout);
  }, [theme, timeout]); // 依赖原始类型
```

```
  return <div>Child</div>;
}
```

这种方式下，只有当 `theme` 或 `timeout` 的值真正改变时，effect 才会重新执行。

Q10: React 官方推荐使用 ESLint 规则 `exhaustive-deps`（来自 `eslint-plugin-react-hooks`）。这条规则有什么作用？如果它提示你需要将 `setState` 函数（例如从 `useState` 返回的 `setCount`）加入依赖项，你应该怎么做？为什么？
A10:
`exhaustive-deps` 规则的作用：
这条规则会分析 `useEffect`（以及 `useCallback`，`useMemo` 等）内部使用到的所有来自组件作用域的变量（props, state, 函数等），并提示你将它们添加到依赖项数组中。其目的是帮助开发者自动检查并提示遗漏的依赖项，从而避免“过时闭包”等难以察觉的 bug。

处理 `setState` 函数的提示：
如果 `exhaustive-deps` 规则提示你将 `setState` 函数（如 `setCount`）或 `useReducer` 返回的 `dispatch` 函数加入依赖项数组，**通常你应该遵循它的建议，将它们添加进去**。

原因：
React **保证** `useState` 返回的 `setState` 函数和 `useReducer` 返回的 `dispatch` 函数的**引用是永久稳定的**。这意味着它们在组件的整个生命周期内都不会改变。因此，将它们加入依赖项数组是完全安全的，并不会导致 `useEffect` 不必要地重新执行。
ESLint 规则要求添加它们是为了保持依赖项列表的完整性和一致性，消除潜在的困惑，并确保即使未来 React 的行为有所改变（尽管对于 `setState` 的稳定性不太可能），代码也能保持健壮。遵循 ESLint 的建议是最佳实践。在极少数特殊情况下可以禁用规则，但必须非常谨慎并充分理解后果。
