14. forwardRef 和 useImperativeHandle 是用来解决什么问题的？（下）

面试题与参考答案

主题：`forwardRef` 和 `useImperativeHandle`

Q1: `React.forwardRef` 主要解决了什么问题？
A1:
`React.forwardRef` 主要解决了父组件无法直接获取子函数组件内部 DOM 元素或子类组件实例的 `ref` 的问题。默认情况下，`ref` 不能直接作为 `props` 传递给函数组件。`forwardRef` 提供了一种机制，允许父组件创建的 `ref` 可以被“转发”到子组件内部，并附加到子组件的某个 DOM 元素或类组件实例上。

Q2: 什么是 `useImperativeHandle`？它通常与哪个 Hook 一起使用？
A2:
`useImperativeHandle` 是一个 React Hook，它允许你在使用 `ref` 时，自定义暴露给父组件的实例值（句柄）。这意味着子组件可以精确控制父组件通过 `ref.current` 能访问到哪些属性和方法。
它**必须和 `React.forwardRef` 一起使用**。`forwardRef` 负责接收父组件传递过来的 `ref`，然后 `useImperativeHandle` 在子组件内部定义这个 `ref` 最终暴露给父组件的内容。

Q3: 既然 `React.forwardRef` 已经可以将 ref 转发到子组件，为什么我们还需要 `useImperativeHandle`？它带来了哪些核心好处？
A3:
虽然 `React.forwardRef` 可以将 `ref` 转发到子组件的 DOM 元素，但这通常意味着将整个 DOM 节点的控制权都暴露给了父组件。`useImperativeHandle` 的出现是为了解决由此可能引发的问题，它带来的核心好处主要有：

- **增强封装性 (Encapsulation)**：避免直接暴露整个内部 DOM 节点或完整的子组件实例。子组件可以隐藏其内部实现细节，父组件不需要关心子组件的内部 DOM 结构，只通过预定义的接口交互，防止父组件误操作或依赖子组件的内部实现。
- **定义清晰的命令式 API (Clear Imperative API)**：允许子组件为父组件创建一个更明确、更受控、更高级别的命令式 API。例如，暴露一个 `resetForm()` 方法，而不是让父组件

去直接操作表单内的各个 `input` 元素。这样，子组件可以在这些自定义方法内部封装更复杂的逻辑（如状态更新、数据打点等）。

- **解耦 (Decoupling)**：父组件依赖的是子组件通过 `useImperativeHandle` 定义的抽象接口，而不是子组件内部具体的 DOM 实现。这意味着如果子组件内部的 DOM 结构或实现发生变化，只要暴露给父组件的 `ref` 接口（自定义方法）保持不变，父组件的代码就无需修改，提高了代码的可维护性。

Q4: `useImperativeHandle` Hook 接收哪三个参数？请简要说明每个参数的作用。
A4:
`useImperativeHandle` Hook 接收三个参数：

1. **`ref`**： 这是从 `React.forwardRef((props, ref) => ...)` 的第二个参数传进来的 `ref` 。它是父组件创建并希望附加到子组件暴露的句柄上的 `ref` 对象。
2. **`createHandle`**： 一个回调函数。这个函数**需要返回一个对象**，该对象会作为父组件 `ref.current` 的值。这个对象通常包含子组件希望暴露给父组件的方法或少量必要的属性。
3. **`dependencies` (可选)**：一个依赖数组，类似于 `useEffect` 或 `useMemo` 的依赖数组。
   - 只有当 `dependencies` 数组中的某个值发生变化时， `createHandle` 函数才会重新执行，从而更新暴露给父组件的句柄。
   - 如果省略此依赖数组，每次组件重新渲染时， `createHandle` 函数都会执行，句柄也会被重新创建。
   - 通常传入空数组 `[]`，使得句柄只在组件挂载时创建一次，以避免不必要的重复创建。

Q5: 请描述一个场景，说明你为什么会选择使用 `useImperativeHandle` 来自定义暴露给父组件的 ref 句柄，而不是直接暴露 DOM 元素。
A5:
假设我正在开发一个自定义的视频播放器组件 `CustomVideoPlayer` 。父组件可能需要控制视频的播放、暂停，以及获取当前播放时间。

如果我仅仅使用 `forwardRef` 将内部的 `<video>` DOM 元素直接暴露给父组件：

- 父组件可以直接调用 `<video>` 元素的所有原生方法和属性（如 `play()` , `pause()` , `volume` , `playbackRate` , `src` 等）。这可能导致父组件意外地修改了播放器的某些内部状态或行为，破坏了 `CustomVideoPlayer` 组件的封装性。例如，父组件直接修改了 `src` 属性，而 `CustomVideoPlayer` 内部可能有一些围绕 `src` 变化的副作用逻辑没有被触发。
- API 不够清晰。父组件需要知道它操作的是一个原生 `<video>` 元素，并了解其原生 API。

如果我使用 `useImperativeHandle`：
我可以定义一个更受控的 API，比如只暴露 `playVideo()`，`pauseVideo()`，`getCurrentTime()` 和 `setVolume(level)` 这几个方法。

```
// Inside CustomVideoPlayer.js
useImperativeHandle(ref, () => ({
  playVideo: () => {
    // 内部可以添加打点、状态更新等逻辑
    videoRef.current.play();
    console.log('Video playing via custom API');
  },
  pauseVideo: () => {
    videoRef.current.pause();
  },
  getCurrentTime: () => {
    return videoRef.current.currentTime;
  },
  // 可能还想对音量设置做一些限制或记录
  setVolume: (newVolume) => {
    if (newVolume >= 0 && newVolume <= 1) {
        videoRef.current.volume = newVolume;
    }
  }
  // 父组件将无法直接访问 videoRef.current.src 或其他原生属性
}), [/* dependencies */]);
```

这样做的好处是：

1. **封装性**：父组件只能调用我明确提供的这几个方法，无法直接操作底层的 `<video>` DOM 元素，保护了组件的内部状态和逻辑。
2. **API 清晰性**：父组件通过 `customVideoPlayerRef.current.playVideo()` 调用，意图明确。
3. **可维护性**：如果将来我想在 `playVideo` 方法内部增加一些额外的打点统计或者状态管理逻辑，我只需要修改 `CustomVideoPlayer` 组件内部，而父组件的调用方式完全不需要改变。

因此，对于这种需要父组件命令式调用子组件特定行为，同时又想保持子组件封装性和API清晰性的场景，我会选择使用 `useImperativeHandle`。

Q6: 在提供的 `CustomInput` 示例中，如果父组件通过 `ref` 访问子组件，它实际得到的是什么？为什么这样做比直接暴露内部的 `input` DOM 元素更好?
A6:
在提供的 `CustomInput` 示例中，当父组件通过 `ref` (即 `customInputApiRef.current`) 访

问子组件时，它实际得到的是在 `useImperativeHandle` 的 `createHandle` 回调函数中返回的那个对象。这个对象包含了三个方法：`focusInput`，`clearInput`，和 `getInputValue`。

```
// 父组件 ref.current 指向这个对象
{
  focusInput: () => { /* ... */ },
  clearInput: () => { /* ... */ },
  getInputValue: () => { /* ... */ }
}
```

这样做比直接暴露内部的 `input` DOM 元素更好，原因如下：

1. **封装性**：父组件无法直接访问 `<input>` DOM 元素的其他原生属性和方法（如 `select()`，`setSelectionRange()`，或者直接修改 `value` 属性而不通过我们定义的方法等）。这保护了 `CustomInput` 组件的内部实现不被外部随意篡改，子组件对其内部 DOM 的控制权更强。
2. **API 清晰可控**：父组件通过调用 `customInputApiRef.current.focusInput()` 而不是 `customInputApiRef.current.focus()`（如果直接暴露 DOM 的话）。这使得API更加语义化，并且子组件可以在这些暴露的方法内部添加额外的逻辑（如示例中的 `console.log`，实际应用中可能是状态更新、校验逻辑等）。
3. **解耦和维护性**：如果 `CustomInput` 的内部结构发生变化（例如，`input` 元素被包裹在一个 `div` 中），只要暴露的 `focusInput`，`clearInput`，`getInputValue` 方法的签名和行为不变，父组件的代码就不需要任何修改。如果直接暴露 DOM，父组件的代码可能需要因为子组件内部结构的改变而调整（例如，从 `ref.current.focus()` 改为 `ref.current.querySelector('input').focus()`）。

Q7: `useImperativeHandle` 的第三个参数（依赖数组）有什么作用？如果省略它或传入空数组 `[]` 会有什么不同？
A7:
`useImperativeHandle` 的第三个参数是一个可选的**依赖数组 (dependencies array)**。它的作用与 `useEffect`，`useMemo`，`useCallback` 中的依赖数组类似：

- **作用**：它控制 `createHandle` 回调函数（即定义暴露给父组件的句柄的那个函数）何时重新执行。只有当依赖数组中的某个值与上一次渲染时相比发生了变化，`createHandle` 函数才会重新执行，从而更新父组件通过 `ref.current` 访问到的句柄。
- **如果省略依赖数组**：`createHandle` 函数会在**每次子组件重新渲染时**都执行。这意味着父组件 `ref.current` 指向的那个包含自定义方法的对象会在每次子组件渲染后都被重新创建。这通常是不必要的，并且可能导致性能问题，尤其如果父组件依赖于这个 `ref` 对象的引用稳定性（虽然通常命令式句柄不应如此）。
- **如果传入空数组 `[]`**：`createHandle` 函数只会在组件**首次挂载时执行一次**，之后就不会再执行了（除非组件被卸载后重新挂载）。这意味着暴露给父组件的句柄（包含那些自定义方法的对象）只会被创建一次，并且在组件的整个生命周期中保持不变（除非组件自身

状态或 props 导致了内部逻辑变化，而这些变化又通过闭包被句柄方法捕获）。这是最常见的用法，因为它确保了暴露的 API 的稳定性，并避免了不必要的重复创建。
- **如果传入包含特定值的数组 `[dep1, dep2, ...]`**： `createHandle` 函数会在组件首次挂载时执行，并且在后续的渲染中，只有当数组中任何一个依赖项 `dep1` 或 `dep2` 等的值发生变化时，才会再次执行以更新句柄。这在暴露的句柄需要根据子组件内部的某些状态或属性动态改变其行为或包含的方法时可能有用，但这种情况相对较少。

总结来说：

- 省略：每次渲染都更新句柄。
- `[]`：仅在挂载时创建一次句柄。
- `[deps]`：仅在依赖项变化时更新句柄。

通常推荐使用空数组 `[]`，除非有明确的理由需要在依赖项变化时更新暴露的语句。