17.useTransition 和 useDeferredValue如何优化用户体验？区别是?

面试题与参考答案：`useTransition` 与 `useDeferredValue`

考察基础知识和定义

Q1: 请解释一下 React 18 中引入 `useTransition` 和 `useDeferredValue` 这两个 Hooks 主要解决了什么样的问题?
A1:
`useTransition` 和 `useDeferredValue` 主要解决了在 React 应用中，当进行一些耗时的 UI 更新（如大数据列表筛选、复杂图表重绘等）时，可能导致页面卡顿、用户输入响应不及时等用户体验不佳的问题。它们通过引入并发特性，允许 React 将某些更新标记为不那么紧急，从而优先处理用户的交互，保证应用的响应性。

Q2: `useTransition` 是什么？它返回什么?
A2:
`useTransition` 是一个 React Hook，它允许我们将某些状态更新标记为“过渡（Transition）”，即低优先级的更新。这意味着 React 会在处理更紧急的更新（如用户输入）之后，再去处理这些被标记为“过渡”的更新。
它返回一个数组，包含两个元素：

1. `isPending` (布尔值)：表示当前的过渡更新是否还在等待执行。可以用来向用户展示加载状态。
2. `startTransition` (函数)：一个函数，用来包裹那些你希望作为低优先级处理的状态更新逻辑。

Q3: `useDeferredValue` 是什么？它的核心作用是什么?
A3:
`useDeferredValue` 是一个 React Hook，它接收一个值，并返回该值的一个“延迟”版本。这个延迟版本的值会在紧急更新（如用户输入引起的UI变化）完成后才会更新。
其核心作用是为你提供一个值的“副本”，这个副本的更新是延迟的，使得依赖这个值的耗时渲染可以推迟执行，从而避免阻塞主渲染流程，保证应用的响应性。

考察理解和阐释能力

Q4: `useTransition` 是如何帮助优化用户体验的？请结合 `startTransition` 和 `isPending` 进行说明。

A4:
`useTransition` 通过允许开发者将某些状态更新标记为低优先级来优化用户体验。

1. 当我们将一个可能引起长时间渲染的状态更新逻辑包裹在 `startTransition` 函数中时，React 会知道这个更新可以被推迟。如果此时有更高优先级的更新（例如用户输入），React 会优先渲染高优先级更新，保持界面的即时响应。
2. `useTransition` 返回的 `isPending` 状态可以用来向用户提供视觉反馈。当被 `startTransition` 包裹的低优先级更新正在进行时， `isPending` 会变为 `true`，开发者可以利用这个状态显示一个加载指示器（如 "加载中..."），告知用户后台正在处理数据，但界面不会因此卡死。
   通过这种方式，即使用户操作触发了耗时的计算或渲染，输入框等交互元素依然能够流畅响应，整体体验更加丝滑。

Q5: `useDeferredValue` 如何实现值的“延迟”？它与原始值之间是什么关系？
A5:
`useDeferredValue` 实现值的“延迟”是通过 React 的并发渲染机制。当你将一个值（如 `value`）传递给 `useDeferredValue(value)` 时，它会返回一个 `deferredValue`。
React 的行为是这样的：

1. 当原始 `value` 发生变化时，React 会优先处理和渲染那些不依赖 `deferredValue` 的、或者依赖原始 `value` 但需要立即响应的部分（比如输入框本身）。
2. `deferredValue` 的更新会“滞后”于原始 `value`。React 会等待浏览器不那么繁忙的时候，或者说等高优先级任务处理完毕后，才会将 `deferredValue` 更新到最新的 `value`。
3. 依赖 `deferredValue` 的组件部分因此会使用一个稍微旧一点的值进行渲染，直到 `deferredValue` 更新。这避免了因原始 `value` 频繁变化导致依赖该值的耗时组件频繁重渲染，从而阻塞用户界面。
   原始值和延迟值之间是“最终一致”的关系：延迟值最终会更新到原始值的最新状态，但这个更新过程是非阻塞的、低优先级的。

Q6: 使用 `useDeferredValue` 时，我们如何知道某个值是否处于“延迟”状态，以便给用户一些反馈？
A6:
`useDeferredValue` 本身并不像 `useTransition` 那样直接返回一个 `isPending` 状态。但是，我们可以通过比较原始值和 `useDeferredValue` 返回的延迟值是否相同，来间接判断是否处于“延迟”状态。
例如，如果有一个 `text` 状态和通过 `const deferredText = useDeferredValue(text)` 得到的延迟值，那么可以通过 `const isLoading = text !== deferredText;` 来判断。当 `text` 已经更新，但 `deferredText` 还没有追上最新的 `text` 时， `isLoading` 就会是 `true`，此时可以显示一个加载提示。

考察应用和解决问题能力

Q7: 请描述一个适合使用 `useTransition` 的具体场景，并说明为什么它比 `useDeferredValue` 更合适。
A7:
一个典型的适合使用 `useTransition` 的场景是**搜索框筛选大型列表**。
在这个场景中：

1. 用户在输入框中输入字符，`inputValue` 状态需要立即更新以保证输入框的响应性（高优先级）。
2. 同时，需要根据输入的内容 `searchTerm` 来筛选一个可能包含成百上千条目的大型列表，这个列表的重新渲染可能非常耗时（低优先级）。

使用 `useTransition` 的原因：

- **控制点明确**：我们能够直接控制设置 `searchTerm` 的 `setState` 操作。我们可以用 `startTransition` 将 `setSearchTerm(newVal)` 这个操作包裹起来，明确告诉 React 这是一个低优先级的更新。

```
// 伪代码
const [inputValue, setInputValue] = useState('');
const [searchTerm, setSearchTerm] = useState('');
const [isPending, startTransition] = useTransition();

const handleInputChange = (e) => {
  setInputValue(e.target.value); // 立即更新
  startTransition(() => {
    setSearchTerm(e.target.value); // 延迟更新，避免列表渲染阻塞输入
  });
};
```

- **反馈机制直接**：`isPending` 状态可以直接用来显示列表正在加载的提示。

为什么 `useDeferredValue` 在此场景中可能不是首选（尽管也能实现类似效果）：
虽然也可以将 `inputValue` 传递给列表组件，并在列表组件内部使用 `useDeferredValue(inputValue)` 来获取一个延迟的搜索词，但 `useTransition` 更直接地作用于“状态更新”这个动作本身，更符合“标记一个更新为过渡”的语义。如果 `inputValue` 和 `searchTerm` 是同一个状态，或者更新 `searchTerm` 的逻辑与 `inputValue` 的更新紧密相连且在同一个组件作用域内，`useTransition` 更为自然。

Q8: 请描述一个适合使用 `useDeferredValue` 的具体场景，并说明为什么它可能比 `useTransition` 更合适。
A8:
一个典型的适合使用 `useDeferredValue` 的场景是**一个图表或数据可视化组件，其数据源作为 prop 从外部传入，并且这个数据源可能频繁更新**。
在这个场景中：

1. 父组件可能频繁地更新传递给图表组件的 `data` prop。

2. 图表组件根据这个 `data` prop 进行复杂的计算和渲染，这个过程非常耗时。
3. 我们可能无法控制父组件更新 `data` prop 的时机或频率。

使用 `useDeferredValue` 的原因：

- **控制点在于值而非更新逻辑**：我们无法直接用 `startTransition` 包裹父组件中更新 `data` 的逻辑（因为它在父组件，或者是一个我们无法修改的自定义 Hook 产生的）。但我们可以在图表组件内部，对接收到的 `data` prop 使用 `useDeferredValue`。

```
// 伪代码
function ChartComponent({ data }) {
  const deferredData = useDeferredValue(data);
  // 使用 deferredData 进行耗时的渲染
  return <ExpensiveChartRender data={deferredData} />;
}
```

- **关注值的延迟**：我们的目标是让图表组件基于一个“不那么紧急更新”的 `data` 版本进行渲染。`useDeferredValue` 正好提供了这个能力，它返回 `data` 的一个延迟版本。

为什么 `useTransition` 在此场景中不适用：
因为 `useTransition` 需要包裹状态更新的函数（`setState`）。如果 `data` 是从 props 传来的，子组件（如图表组件）通常没有权力去用 `startTransition` 包裹父组件中导致 `data` 改变的 `setState` 调用。此时 `useDeferredValue` 更为合适，因为它直接作用于这个传入的值。

Q9: 在一个包含输入框和内容预览（例如 Markdown 编辑器）的场景中，如果输入非常快，预览区域的更新可能会导致卡顿。你会考虑使用 `useTransition` 还是 `useDeferredValue`？为什么？
A9:
在这个场景中，`useTransition` 和 `useDeferredValue` 都有可能适用，具体选择取决于状态管理和组件的组织方式：

1. **使用 `useTransition` 的情况**：
   如果编辑器中的文本内容（`editorText`）和用于预览的内容（`previewText`）是两个独立的状态，并且我们有专门的逻辑（例如一个 `useEffect` 或事件处理函数）在 `editorText` 改变时去更新 `previewText`，那么可以使用 `useTransition`。

```
// 伪代码
const [editorText, setEditorText] = useState('');
const [previewText, setPreviewText] = useState('');
const [isPending, startTransition] = useTransition();

const handleEditorChange = (newText) => {
  setEditorText(newText); // 立即更新编辑器输入
  startTransition(() => {
    setPreviewText(newText); // 延迟更新预览内容
```

```
  });
};
// <PreviewComponent content={previewText} />
// {isPending && <p>Loading preview...</p>}
```

这种情况下，我们主动控制了 `setPreviewText` 这个状态更新的时机。

2. **使用 `useDeferredValue` 的情况**：
   如果预览组件直接接收编辑器中的文本内容作为 prop (`<PreviewComponent content={editorText} />`)，并且预览组件内部的渲染逻辑非常耗时，那么可以在预览组件内部使用 `useDeferredValue`。

```
// 伪代码 - App.js
const [editorText, setEditorText] = useState('');
// ...
// <PreviewComponent content={editorText} />

// 伪代码 - PreviewComponent.js
function PreviewComponent({ content }) {
  const deferredContent = useDeferredValue(content);
  // 使用 deferredContent 进行耗时的 Markdown 解析和渲染
  // const showLoading = content !== deferredContent;
  // ...
}
```

这种情况下，预览组件“响应式”地处理传入的 `content` prop，并获取其延迟版本进行渲染。

选择依据：

- 如果你能直接控制导致性能问题的状态更新代码（例如，你有专门的 `setState` 来更新预览区内容），`useTransition` 通常更直接。
- 如果你无法直接控制值的更新源头（例如，预览组件只是被动接收一个频繁变化的 prop），或者希望在子组件内部处理渲染延迟，`useDeferredValue` 是更好的选择。

考察区别与拓展思考

Q10: 请从控制点、API与反馈机制以及“谁发起”的视角，总结一下 `useTransition` 和 `useDeferredValue` 的主要区别。
A10:
`useTransition` 和 `useDeferredValue` 的主要区别可以从以下几个角度阐述：

1. **控制点不同**：
   - `useTransition`：它允许你**包裹状态更新的逻辑**。你明确知道哪一个 `setState` 操作是耗时的，并且希望将其标记为“过渡”。你有对这个 `setState` 的直接控制权。

- `useDeferredValue`： 它允许你**包裹一个值**（通常是 props，或者是通过其他 hooks 派生出来的状态）。你可能没有直接控制这个值的更新源头（比如它是从父组件传来的），但你希望基于这个值的下游渲染能够延迟。

2. **API 和反馈机制**：
   - `useTransition`： 返回 `isPending` 状态和一个 `startTransition` 函数。`isPending` 状态可以直接用于向用户展示加载中的 UI 反馈。
   - `useDeferredValue`： 只返回一个延迟后的值。它没有直接提供 `isPending` 状态，如果需要加载状态，通常需要开发者自己通过比较原始值和延迟值来实现（例如 `const isLoading = originalValue !== deferredValue;`）。
3. **“谁发起”的视角（主动性 vs. 被动性）**：
   - `useTransition`： 可以看作是**主动的**。开发者主动使用 `startTransition` 函数告诉 React：“这段代码里的状态更新，请作为 transition 处理，可以推迟。”
   - `useDeferredValue`： 可以看作是**被动的**或**响应式的**。开发者只是声明：“我有一个值，我希望它的更新能不那么紧急，给我一个它的延迟版本。” React 会自动处理这个值的延迟更新。

总结来说，**如果你能控制导致性能问题的状态更新代码，并且希望明确标记这些更新为低优先级，`useTransition` 更合适。如果你无法直接控制值的更新源头，但希望基于这个值的组件渲染能够延迟执行，`useDeferredValue` 是更好的选择。**

Q11: 在优化组件性能时，`React.memo` 和 `useTransition` / `useDeferredValue` 有什么不同？它们可以一起使用吗？
A11:
`React.memo` 和 `useTransition` / `useDeferredValue` 是处理不同性能优化方向的工具：

- **`React.memo`**：
  - 主要作用是**避免不必要的重渲染**。它是一个高阶组件，会对组件的 props 进行浅比较（也可以提供自定义比较函数）。如果 props 没有发生变化，`React.memo` 会阻止组件的重渲染，从而节省渲染成本。
  - 它关注的是“组件是否需要重渲染”。
- **`useTransition` / `useDeferredValue`**：
  - 主要作用是处理那些**即使是必要的渲染，但也因为耗时过长而阻塞用户交互**的情况。它们通过将某些更新标记为低优先级，来改善应用的响应性。
  - 它们关注的是渲染的“时机”和“优先级”，而不是“是否渲染”。它们允许一个慢渲染发生，但确保它不会阻塞更重要的用户交互。

它们可以一起使用吗？
是的，它们可以而且经常一起使用，解决不同层面的性能问题。

- 例如，你可能有一个列表组件，列表中的每个列表项（`ListItem`）都用 `React.memo` 包裹，以避免在无关数据变化时单个列表项的无效重渲染。
- 同时，如果整个列表的筛选或排序操作非常耗时（即使每个 `ListItem` 都被 `memo` 优化了，但大量 `ListItem` 的重新排序和渲染仍然可能慢），你可以使用 `useTransition` 来

包裹触发列表筛选/排序的状态更新，或者如果筛选条件来自 props，则在列表组件内部对该筛选条件使用 `useDeferredValue`。

这样，`React.memo` 确保了单个组件只在必要时更新，而 `useTransition` / `useDeferredValue` 则确保了即使是这些必要的更新，如果它们很慢，也不会降低应用的整体响应速度。