16.如何解释 React 18 的并发特性（Concurrency）？

面试题与参考答案

第一部分：理解并发特性 (Understanding Concurrency)

Q1: 请用你自己的话解释一下 React 18 的并发特性是什么？它和之前 React 版本的渲染方式有何不同?
A1:
React 18 的并发特性是其渲染机制的一次重大升级。简单来说，它允许 React 在渲染一个大的更新任务时，能够“暂停”下来，去处理更紧急的任务（比如用户的点击或输入），然后再回来继续之前的渲染工作。这就像一个聪明的厨师在做一道复杂的菜（比如“佛跳墙”）时，可以中途停下来，先给客人倒杯可乐（紧急且简单的需求），然后再继续做大菜。

与之前版本的主要区别在于：

- **旧模式（同步渲染）**：一旦开始一个渲染任务，React 必须一口气把它做完，中途不能被打断。如果这个任务很耗时，整个应用就会卡住，无法响应用户操作。
- **新模式（并发渲染）**：React 的渲染过程变得可中断。它可以将渲染工作拆分成小片，在每片工作完成后，会检查是否有更高优先级的任务需要处理。这样就确保了即使在进行复杂渲染时，应用也能及时响应用户。

Q2: React 18 引入并发特性主要是为了解决什么问题?
A2:
并发特性主要为了解决以下由传统同步阻塞式渲染导致的用户体验问题：

1. **UI 卡顿**：当 React 处理一个耗时较长的更新（如加载复杂图表、渲染长列表）时，主线程被长时间占用，导致页面无法响应用户操作，看起来像卡死了。
2. **交互延迟**：即使用户的操作最终被处理，但因为需要等待之前的长任务完成，响应会显得非常缓慢。
3. **动画掉帧**：流畅的动画需要浏览器稳定地绘制帧。如果 React 的渲染任务阻塞了主线程，浏览器就无法及时绘制下一帧，导致动画卡顿。

通过并发，React 能够更智能地调度任务，优先保证用户输入的响应，从而提升应用的整体流畅度和用户体验。

Q3: React 的并发特性是基于多线程实现的吗？如果不是，它大概是如何工作的?
A3:
不是的。JavaScript 本质上仍然是单线程执行的。React 的并发特性并不是指 React 同时在多个线程上运行代码（那叫并行）。

它更多的是一种在单线程环境下的**任务调度策略**和**可中断的渲染机制**。核心思想是：

- **可中断性**：React 可以在渲染过程中暂停，处理其他优先任务，然后再回来继续。
- **时间分片 (Time Slicing)**：React 将大的渲染任务分解成许多小的时间片。在每个时间片执行完毕后，React 会将控制权交还给浏览器，让浏览器有机会处理用户输入、动画等其他任务。
- **优先级调度 (Priority Scheduling)**：React 会为不同的更新分配不同的优先级。例如，用户输入是高优先级的，而后台数据获取后的渲染可能是低优先级的。高优先级的更新可以打断低优先级的更新。

所以，并发更像是一种让 React 在主线程上更智能、更合作地分配工作时间的方式，而不是真正的多线程并行处理。

Q4: 请解释一下并发模式下的几个核心机制，比如“可中断的渲染”和“时间分片”。
A4:
并发模式的核心机制包括：

1. **可中断的渲染 (Interruptible Rendering)**：这是并发的基础。React 可以开始渲染一个更新，但如果中途有更高优先级的任务（如用户输入）进来，它可以暂停当前的渲染工作，先去处理高优先级的任务。处理完毕后，React 可以选择回来继续之前的渲染，或者如果之前的数据已经过时，甚至可以放弃之前的渲染成果，根据最新数据重新开始。
2. **时间分片 (Time Slicing)**：React 不会一次性执行完整个更新渲染，而是将工作分解成许多小块（时间片）。在完成一小块工作后，它会主动把执行控制权交还给浏览器主线程，让浏览器有机会处理其他待处理的任务（如响应用户输入、执行动画的下一帧等）。如果浏览器没有其他紧急任务，并且当前帧还有剩余时间，React 才会继续执行下一块渲染工作。
3. **优先级调度 (Priority Scheduling)**：React 能够为不同的更新任务赋予不同的优先级。例如，用户直接交互（如键盘输入、点击）产生的更新通常具有高优先级，而后台加载数据后触发的UI更新可能具有较低优先级。高优先级的更新可以“插队”并打断正在进行的低优先级更新的渲染过程。

第二部分：表达与应用并发特性 (Explaining and Applying Concurrency)

Q5: 如果面试官问你：“React 18 的并发特性，你是如何理解的？”你会如何组织你的回答?
A5:
我会从以下几个方面来组织我的回答：

1. **定义与核心思想**：首先，我会解释并发是 React 渲染机制的根本性变革，它使得渲染过程可以被打断以响应更紧急的任务。其核心思想是实现可合作（cooperative）和可中断（interruptible）的渲染，避免长时间占用主线程。
2. **解决的问题**：接着，我会说明并发主要解决了同步渲染模型下的 UI 卡顿、交互延迟和动画掉帧等用户体验痛点，通过智能调度优先保证用户输入的响应。
3. **关键机制简介**：然后，我会简要介绍实现并发所依赖的底层机制，如时间分片、优先级调度和可中断渲染，说明 React 如何将大任务拆分并在执行间隙响应高优任务。

4. **开发者如何利用并发（举例）**：之后，我会提到 React 18 提供的新 API，如 `startTransition` 和 `useTransition`，并举例说明如何使用它们来标记非紧急更新（如搜索结果的渲染），从而保持 UI 的流畅性。例如，在搜索框中，用户输入是高优的，而结果展示可以用 `startTransition` 包裹。
5. **与并行的区别（如果需要）**：最后，如果面试官有进一步的疑问，我会强调 React 的并发是在单线程环境中通过巧妙的调度实现的，并非真正的多线程并行。

Q6: 请解释一下 `startTransition` 是什么，以及它如何帮助我们优化用户体验?
A6:
`startTransition` 是 React 18 提供的一个新的 API，它允许开发者将某些状态更新标记为“过渡（Transition）”，即非紧急更新。

它的主要作用和优化体验的方式如下：

1. **区分更新优先级**：`startTransition` 帮助 React 区分哪些 UI 更新是紧急的（例如，用户在输入框中输入文本，输入框需要立即显示反馈），哪些是可以稍微推迟或者在渲染过程中被其他更紧急更新打断的（例如，根据输入文本去获取数据并渲染一个复杂的列表）。
2. **保持 UI 响应**：当一个状态更新被 `startTransition` 包裹后，React 会以较低的优先级来处理这个更新。如果在执行这个低优先级更新的过程中，有更高优先级的更新（如用户再次输入或点击），React 会暂停当前的低优先级渲染，优先处理高优先级更新，从而确保用户界面始终保持对用户操作的即时响应，避免卡顿。
3. **避免不必要的加载状态**：虽然过渡更新可能会被推迟，但 `useTransition` Hook（通常与 `startTransition` 一起使用）会返回一个 `isPending` 状态。开发者可以使用这个状态来告知用户后台正在进行一些工作（例如显示一个加载指示器），但这个加载指示器本身不会阻塞用户与页面其他部分的交互。

例如，在一个搜索功能中，用户在输入框输入字符时，`setInputValue`（更新输入框显示）是紧急的。而 `setSearchQuery`（触发搜索结果列表渲染）可以用 `startTransition` 包裹。这样，即使用户快速连续输入，输入框的响应依然流畅，搜索结果的渲染会在后台进行，不会阻塞输入。

Q7: `useTransition` Hook 返回什么？在实际应用中，我们如何使用它返回的值?
A7:
`useTransition` Hook 返回一个包含两个元素的数组：

1. `isPending` (boolean): 这是一个布尔值，表示由 `startTransition` 标记的过渡更新当前是否正在进行中。如果为 `true`，则意味着过渡任务尚未完成。
2. `startTransition` (function): 这是一个函数，用来包裹那些你希望标记为“过渡”的状态更新。

在实际应用中，我们可以这样使用它们：

- **`startTransition` 函数**：我们将那些可能导致 UI 卡顿的、非紧急的状态更新逻辑放到 `startTransition` 的回调函数中。例如，更新一个用于过滤大型列表的搜索词：

```
const [isPending, startTransition] = useTransition();
const [searchQuery, setSearchQuery] = useState('');

const handleSearchChange = (value) => {
  // 紧急更新，直接设置，让输入框立即响应
  setInputValue(value);
  // 非紧急更新，用 startTransition 包裹
  startTransition(() => {
    setSearchQuery(value); // 这个更新会被认为是可中断的
  });
};
```

- **`isPending` 状态**：我们可以使用 `isPending` 来给用户提供视觉反馈，告知他们一个后台操作正在进行，但这个反馈不应该阻塞用户的进一步交互。例如，在列表更新期间显示一个加载指示器或禁用某个按钮：
  `javascript return ( <> <input onChange={(e) => handleSearchChange(e.target.value)} /> {isPending && <p>Loading list... </p>} <MySlowList items={filteredItems} /> </> );`
  通过这种方式， `useTransition` 帮助我们将紧急更新和非紧急更新分开处理，从而在执行耗时操作时也能保持应用的响应性。

Q8: 在讲义的搜索框示例中，为什么将 `setInputValue` 视为紧急更新，而将 `setSearchQuery` 包裹在 `startTransition` 中？如果不这样做，可能会有什么问题？
A8:
在这个搜索框示例中：

- `setInputValue(e.target.value);` 被视为**紧急更新**，因为它直接控制用户在输入框中看到的文本。用户期望在输入时立即看到字符出现在输入框中，这是即时反馈，对用户体验至关重要。
- `startTransition(() => { setSearchQuery(e.target.value); });` 中的 `setSearchQuery` 被视为**过渡更新**（非紧急更新），因为它通常会触发一个可能比较耗时的操作，比如根据新的查询词去过滤一个庞大的数据集或从服务器获取数据并渲染结果列表。

如果不这样做（即 `setInputValue` 和 `setSearchQuery` 都作为同步的、高优先级的更新处理）：
每次用户输入一个字符， `setInputValue` 会更新输入框，紧接着 `setSearchQuery` 会更新查询词，并立即触发 `ExpensiveList` 的重新渲染。如果 `ExpensiveList` 的渲染非常耗时（例如列表项很多，或者每个列表项都很复杂），那么在 `ExpensiveList` 渲染完成之前，主线程会被阻塞。
这会导致：

1. **输入框卡顿**：用户会感觉到输入字符后，输入框的显示不是立即更新的，或者输入非常迟钝，因为浏览器忙于渲染列表，无法及时响应输入框的更新。

2. **整体应用无响应**：在列表渲染期间，用户可能无法进行其他操作，比如点击页面上的其他按钮或滚动。

通过使用 `startTransition` 将 `setSearchQuery` 标记为过渡，React 就知道这个更新可以被推迟或中断。当用户输入时， `setInputValue` 会被高优先级处理，保证输入框的流畅；而 `setSearchQuery` 及其引发的列表渲染会在后台以较低优先级进行。如果用户在列表渲染完成前又输入了新的字符，React 可能会中断之前的低优先级渲染，优先处理新的输入和随后的过渡更新，从而始终保持界面的响应性。