36.你还知道哪些 React 性能优化的具体手段?

Q1: 为什么在React开发中性能优化如此重要?
A1: 性能优化至关重要，主要基于两点：

- **用户体验**：应用的流畅度和快速响应是提升用户满意度的关键。卡顿、白屏或响应慢会直接导致用户流失。
- **业务指标**：良好的性能通常与更高的用户转化率和更好的用户留存率等核心业务指标正相关。

Q2: 请列举一些React应用中常见的性能瓶颈。
A2: 常见的性能瓶颈包括：

- **不必要的组件重渲染 (Re-renders)**：当组件的 props 或 state 没有实际变化时，依然触发了渲染。
- **庞大的组件树与深层嵌套**：导致更新和渲染的计算量增大。
- **大量数据处理与渲染**：例如渲染一个包含成百上千项的长列表。
- **首屏加载时间过长**：主要是由过大的初始包体积（Bundle Size）引起的。

Q3: 如果让你负责优化一个性能不佳的React应用，你会遵循怎样的思路和流程?
A3: 我会遵循一个结构化的流程：

- **第一步：发现与定位**：首先使用 React DevTools Profiler 等工具来记录和分析组件的渲染行为，找出渲染次数过多或耗时过长的组件，从而定位性能瓶瓶颈。
- **第二步：分析与策略**：分析瓶颈产生的原因。是无效的重渲染？是昂贵的计算？还是列表渲染问题？然后根据具体原因选择合适的优化手段，例如 `React.memo`、`useMemo`、`useCallback`、列表虚拟化或代码分割。
- **第三步：实施与验证**：小步快跑，逐步应用选定的优化策略，并再次使用 Profiler 工具进行测量，通过对比优化前后的数据来量化和验证优化效果。
- **关键点**：我会特别注意避免过早优化和过度优化，确保每一项优化都是针对已发现的、有数据支撑的性能瓶颈。

核心API与Hooks

Q4: `React.memo` 的作用是什么？它的工作原理是怎样的?
A4: `React.memo` 是一个高阶组件（HOC），用于优化函数组件的性能。它的主要作用是“记住”组件上一次渲染的结果，如果新的 props 与上一次的 props 相同，它将跳过本次渲染，直接复用上一次的结果。其工作原理是通过对组件接收的 props 进行浅比较（shallow compare）来决定是否需要重渲染。

Q5: `React.memo` 和 `React.PureComponent` 有什么异同?
A5:

- **相同点**：它们的核心作用和原理是相同的，都是通过对 props（以及 `PureComponent` 对 state）进行浅比较来避免不必要的渲染。
- **不同点**： `React.memo` 用于函数组件（Functional Components），而 `React.PureComponent` 用于类组件（Class Components）。

Q6: `useCallback` 和 `useMemo` 的作用分别是什么？请说明它们的区别。
A6:

- **`useCallback`**： 用于记忆化（memoize）一个回调函数。它会返回该回调函数的记忆化版本，只有当依赖项数组（deps）中的某个值发生变化时，才会重新创建一个新的函数实例。这在将回调函数传递给经过优化的子组件时非常有用，可以防止因为函数引用变化而导致的子组件不必要的重渲染。
- **`useMemo`**： 用于记忆化一个计算结果。它会执行传入的函数并记住其返回值，只有当依赖项数组（deps）中的某个值发生变化时，才会重新执行函数进行计算。这适用于避免在每次渲染时都进行昂贵的计算。
- **区别**： `useCallback` 记忆化的是函数本身 (`useCallback(fn, deps)`)，而 `useMemo` 记忆化的是函数的执行结果 (`useMemo(() => fn(), deps)`)。可以说， `useCallback(fn, deps)` 等价于 `useMemo(() => fn, deps)` 。

Q7: 在使用 `useCallback` 或 `useMemo` 时，依赖项数组（deps）的作用是什么？如果传入一个空数组 `[]` 会怎样?
A7: 依赖项数组（deps）是 `useCallback` 和 `useMemo` 的第二个参数，它告诉React在何时重新计算被记忆的值或函数。React会比较数组中每一项与上一次渲染时的值，如果任何一项发生变化，就会重新执行计算/创建函数。如果传入一个空数组 `[]`，意味着这个 effect 或记忆化的值不依赖于任何 props 或 state，因此它只会在组件的初始渲染时执行一次，之后将一直返回第一次创建的函数实例或计算值。

特定场景优化

Q8: 当你需要渲染一个包含上千条数据的长列表时，可能会遇到什么性能问题？你会如何解决?
A8:

- **问题**：渲染一个长列表会一次性创建大量的DOM节点，这会导致初始渲染时间过长、内存占用过高，并且在滚动时可能引起卡顿，严重影响用户体验。
- **解决方案**：采用“虚拟化”（Windowing 或 Virtualization）技术。其核心原理是只渲染当前视口（Viewport）内可见的列表项，以及在视口上下方预留少量缓冲区。当用户滚动列表时，动态计算并更新视口内的列表项，从而极大减少了同一时间需要渲染的DOM节点数量，显著提升了性能。可以借助 `react-window` 或 `react-virtualized` 这样的库来实现。

Q9: 什么是代码分割（Code Splitting）？ `React.lazy` 和 `Suspense` 在其中扮演什么角色?
A9:

- **代码分割**：是一种将代码库（bundle）拆分成多个小块的技术，这些小块可以根据需要进行按需加载或并行加载，而不是在应用启动时一次性加载所有代码。主要目的是减小初始加载的包体积，从而加快首屏渲染速度。
- `React.lazy`：是React内置的函数，它允许你像渲染常规组件一样来渲染一个动态导入（dynamic import）的组件。它使得组件的加载可以被延迟到该组件首次需要被渲染时。
- `Suspense`：是一个组件，它可以让你在子组件加载完成前“暂停”渲染，并指定一个加载指示器（如loading spinner）。`React.lazy` 必须与 `Suspense` 组件配合使用，`Suspense` 负责处理懒加载组件在加载过程中显示的“后备”UI。

工具与实践

Q10: 你会使用什么工具来诊断React应用的性能问题？请简述其主要功能。
A10: 我会使用官方的 **React DevTools Profiler** 浏览器扩展工具。它的主要功能包括：

- **Profiler Tab**：可以启动一次性能录制，记录下这段时间内应用发生的每一次组件渲染及其耗时。
- **火焰图 (Flamegraph Chart)**：以图形化的方式展示单次提交（commit）中所有组件的渲染层级、关系以及各自的渲染耗时，可以帮助快速理解哪部分组件树开销最大。
- **排行榜 (Ranked Chart)**：将单次提交中的所有组件按渲染耗时从高到低排序，可以让我一眼就看到哪个组件是性能瓶颈。
  通过这些功能，我可以精确地识别出哪些组件存在不必要的更新，从而进行针对性优化。