35.你用过哪些工具来定位 React 应用的性能瓶颈?

React DevTools Profiler

Q1: 什么是React DevTools Profiler，它主要用来解决什么问题?
A1: React DevTools Profiler是React官方浏览器扩展的一部分。它是一个性能分析工具，主要用于解决React组件级别的性能问题，核心功能包括：

- **可视化渲染耗时**：通过火焰图等形式，直观地展示每个组件的渲染时间。
- **识别不必要的重渲染**：帮助开发者发现哪些组件在不应该更新的时候进行了重渲染，从而造成性能浪费。
- **分析commit阶段性能**：可以分析React在“commit”阶段（即更新DOM）的总耗时。

Q2: 请描述一下React DevTools Profiler中的核心功能或视图，例如火焰图和排行榜。
A2:

- **火焰图 (Flame graph)**: 它以层级结构展示了单次commit中所有组件的渲染情况。图中每个条块代表一个React组件，条块的宽度代表该组件及其所有子组件渲染所花费的总时间；条块的颜色代表该组件自身渲染花费的时间（颜色越偏暖色，如黄色，耗时越长；灰色代表该组件本次未渲染）。
- **排行榜 (Ranked chart)**: 它将单次commit中渲染过的所有组件按“自身渲染耗时”从高到低进行排序。这能让开发者快速定位到本次更新中渲染最耗时的组件。
- **组件图 (Component chart)**: 当你在火焰图或排行榜中选中一个组件后，组件图会展示在多次不同的commit中，该特定组件的渲染耗时情况，便于追踪一个组件在多次交互下的性能表现。

Q3: 在面试中，你会如何描述你使用React DevTools Profiler来定位和优化性能问题的过程?
A3: 在分析React组件性能时，我主要使用React DevTools Profiler。我的步骤通常是：

1. **录制交互**: 在Profiler面板中点击“录制”，然后在应用中执行我怀疑存在性能问题的用户交互操作。
2. **分析数据**: 停止录制后，我会首先查看**排行榜 (Ranked chart)**，快速定位出渲染耗时最长的组件。
3. **深入检查**: 接着，我会切换到**火焰图 (Flame graph)**，观察这些耗时较长组件（通常是较宽的条块）的渲染情况，并分析在特定交互下，哪些组件发生了重渲染。
4. **定位原因并优化**: 一旦定位到可疑组件，我会分析其 `props` 和 `state` 的变化，判断重渲染是否是必要的。如果是不必要的，我会考虑使用 `React.memo` 来包裹组件以阻止因父组件更新导致的无效渲染，或者使用 `useMemo` 和 `useCallback` 来缓存昂贵的计算结果和函数实例，避免不必要的子组件重渲染。

分类：Chrome DevTools Performance Tab

Q4: 什么时候你会选择使用Chrome DevTools的Performance面板，而不是React DevTools Profiler?
A4: 当我怀疑性能瓶颈不仅仅局限于React组件渲染时，我会使用Chrome DevTools的Performance面板。主要有以下几种场景：

- **底层性能分析**：需要分析整个应用的性能，包括原生JavaScript执行、渲染、网络、布局抖动(Layout Thrashing)和绘制风暴(Paint Storms)等。
- **识别长任务**：当页面出现卡顿时，需要识别并分析阻塞主线程的JavaScript长任务（Long Tasks）。
- **宏观指标分析**：需要关注首次内容绘制 (FCP)、最大内容绘制 (LCP)、可交互时间 (TTI)等浏览器层面的核心性能指标。
- **非React瓶颈**：怀疑性能问题可能由第三方库、复杂的DOM操作或非React相关的逻辑引起时。

Q5: Chrome Performance面板中的“Timings”轨道有什么用？它如何与React应用分析关联起来?
A5: 在开发模式下，React 16+版本会利用浏览器的“User Timing API”。这意味着React组件的生命周期事件（如挂载、更新）和commit阶段会被自动标记出来，并显示在Performance面板的“Timings”轨道中。通过查看这些标记，我们可以将浏览器层面捕获到的宏观性能数据（如某个长任务）与具体的React组件行为对应起来，从而更精确地定位到是哪个组件的更新导致了性能问题。

Q6: 请描述一下你使用Chrome Performance面板分析React应用性能的基本流程。
A6:

1. 打开Chrome DevTools并切换到Performance面板。
2. 点击录制按钮，开始记录性能数据。
3. 在应用界面上，复现你想要分析的交互场景（如滚动、点击、输入等）。
4. 停止录制，等待面板处理并生成性能报告。
5. 重点分析 **Main** 主线程的火焰图，寻找是否存在红色三角形标记的 **长任务 (Long Tasks)**，它们是导致页面卡顿的主要原因。
6. 查看 **Timings** 轨道，寻找React自动生成的标记（如 `[App] render start`，`commit`，`mount` 等），将它们与主线程上的任务关联起来，理解是哪个React更新触发了耗时操作。
7. 通过下方的 **Bottom-Up** 或 **Call Tree** 标签页，分析耗时任务的JavaScript调用栈，找到具体耗时的函数。

其他工具 (Lighthouse & `why-did-you-render`)

Q7: Lighthouse是什么？在React性能优化中它扮演什么角色?
A7: Lighthouse是Google提供的一款开源的、自动化的网站质量审计工具。在React性能优化中，它主要扮演**宏观审计和指标监控**的角色：

- **提供高阶性能指标**：它能自动化地测试并生成一份包含核心Web指标（Core Web Vitals）的报告，如最大内容绘制 (LCP)、总阻塞时间 (TBT) 和累积布局偏移 (CLS) 等。
- **提供优化建议**：报告中会包含一系列具体的、可操作的性能优化建议，例如压缩图片、移除未使用的JavaScript、启用文本压缩等。
- **周期性健康检查**：适合用于对应用进行定期的性能“体检”，从一个较高的维度来评估应用的整体性能表现和用户体验，并为优化工作指明大方向。

Q8: `why-did-you-render` 这个库是用来做什么的？它能解决什么具体问题?
A8: `why-did-you-render` 是一个用于React开发的第三方库。它专门用来在开发环境下检测并报告组件的**“不必要”重渲染**。它能解决的非常具体的问题是：当一个组件重渲染时，它会在浏览器的控制台明确地打印出导致这次重渲染的原因，例如：

- 某个 `prop` 的值虽然“相等”，但引用地址发生了变化（如在父组件中每次都内联创建一个新对象或函数）。
- 某个 `state` 发生了变化。
- `props` 或 `state` 的某些属性发生了改变。

通过这些精确的日志，开发者可以快速定位并修复由于不当的数据传递方式而导致的性能浪费。

综合与工具选择

Q9: 请总结一下，在面对一个React性能问题时，你会如何选择使用React Profiler, Chrome Performance, Lighthouse, 和 `why-did-you-render` 这四种工具?
A9: 我会根据问题的性质和分析阶段来选择合适的工具，通常会组合使用它们：

- **首选 React DevTools Profiler**：当问题明确或高度怀疑是与React组件渲染直接相关时，比如想知道“哪个组件慢”或“哪个组件在无效更新”。这是组件级分析的首选。
- **使用 Chrome DevTools Performance**：当问题更复杂、更底层，或怀疑瓶颈在React之外时（如长任务JS、布局问题），我会用它。它提供的是应用级的宏观性能视图。
- **定期运行 Lighthouse**：用于周期性的性能审计，获取整体性能评分和核心Web指标，并发现高层次的优化机会。它更像是一个“健康检查”工具。
- **开发中集成 `why-did-you-render`**：在日常开发和调试阶段，用它来精细化地检测和预防不必要的组件重渲染，是一种辅助开发的利器。

Q10: 你接手了一个大型React项目，被要求进行一次全面的性能评估和优化。请简述你的行动计划，并说明如何结合运用上述工具。
A10: 我的计划会分步进行，并结合使用多种工具：

1. **基线评估 (Lighthouse)**：首先，我会使用Lighthouse对应用的关键页面（如首页、核心业务页）进行一次全面的性能审计。这会给我一个关于当前性能水平的基线数据，特别是核心Web指标（LCP, TBT, CLS），并提供一些初步的、通用的优化建议（如图片优化、代码分割等）。
2. **识别关键交互与瓶颈 (Chrome Performance)**：根据Lighthouse的报告和业务重要性，我会选定几个关键的用户交互流程。然后使用Chrome Performance面板录制这些流程，深入分析主线程活动，寻找长任务（Long Tasks）和渲染瓶颈，并将它们与React的User Timing标记关联，初步定位到可能存在问题的宏观阶段或组件群。
3. **组件级深度分析 (React Profiler)**：对于上一步中定位到的、在交互中耗时较长的React部分，我会使用React DevTools Profiler进行更精细的分析。我会录制相同的交互，然后通过火焰图和排行榜精确找出是哪些组件渲染耗时过长，或者发生了不必要的重渲染。
4. **修复与验证 (why-did-you-render & Profiler)**：在代码层面进行优化时，我会集成 `why-did-you-render` 来帮助我精准定位并修复那些由于props引用变化等原因导致的无效渲染。完成代码修改（如添加 `React.memo`，`useMemo`，`useCallback`）后，我会再次使用React Profiler进行录制，对比优化前后的渲染耗时和重渲染情况，以验证优化的效果。
5. **回归测试 (Lighthouse & Chrome Performance)**：优化完成后，再次使用Lighthouse和Chrome Performance进行测试，确保整体性能指标得到提升，并且没有引入新的性能问题。