13.useRef 有哪些常见用途？它和 useState的根本区别是?

面试题与参考答案：`useRef`的原理及使用

考察基础知识和定义

Q1: 请解释一下 React 中的 `useRef` 是什么？它返回什么?
A1:
`useRef` 是 React 提供的一个 Hook 函数。当你调用 `useRef(initialValue)` 时，它会返回一个可变的 ref 对象。这个 ref 对象拥有一个名为 `.current` 的属性，该属性被初始化为传入的 `initialValue`。重要的是，这个 ref 对象在组件的整个生命周期内都会保持不变，即使组件重新渲染，`useRef` 返回的始终是同一个对象引用。

Q2: `useRef` 返回的 ref 对象的 `.current` 属性有什么特点?
A2:
`.current` 属性是 `useRef` 返回的 ref 对象的核心。

1. **可变性**：你可以自由地修改 `ref.current` 的值，例如 `myRef.current = "新的值";`。
2. **不触发渲染**：最关键的特点是，当你修改 `.current` 属性的值时，React **并不会**因此触发组件的重新渲染。
3. **持久性**：`ref.current` 中存储的值在组件的多次渲染之间是持久的，除非你显式地修改它。

考察理解和阐释能力

Q3: `useRef` 和 `useState` 最根本的区别是什么？请详细说明。
A3:
`useRef` 和 `useState` 最根本的区别在于**是否会触发组件的重新渲染**。

- **`useState`**：
  - 用于管理那些当它们发生变化时，需要让 UI 自动更新的数据（即状态）。
  - 通过其 setter 函数（如 `setCount`）更新值时，如果新值与当前值不同，会**触发组件的重新渲染**，从而让界面反映出最新的状态。
- **`useRef`**：
  - 用于存储那些你**不希望**因为它们的变化而触发 UI 更新的可变值。
  - 修改其 `.current` 属性时，**不会触发组件的重新渲染**。
  - `useRef` 返回的 ref 对象本身在组件的整个生命周期内是持久的。

可以这样理解：`useState` 中的数据变化需要“通知”React 并可能导致界面更新；而 `useRef` 中的数据变化则是在组件内部“静默”发生的，不直接引起界面刷新。

Q4: 为什么说 `useRef` 返回的 ref 对象在组件的整个生命周期内是持久的？这对开发者意味着什么？
A4:
说 `useRef` 返回的 ref 对象在组件整个生命周期内是持久的，意味着无论组件因为 props 或 state 的变化重新渲染多少次，`useRef` 在首次调用时创建并返回的那个 JavaScript 对象始终是同一个实例。
这对开发者意味着：

1. **稳定的容器**：你可以依赖这个 ref 对象作为跨渲染周期的稳定“容器”来存储数据。
2. **避免重复创建**：不需要担心每次渲染都会得到一个新的 ref 对象，这有助于维持对特定值（如 DOM 节点引用或定时器 ID）的一致访问。
3. **类似实例变量**：可以将 `useRef` 看作是在函数组件中创建“实例变量”的一种方式，这些变量可以在多次渲染之间共享和修改，而不会直接影响渲染流程。

考察应用和解决问题能力

Q5: `useRef` 有哪些常见的用途？请至少列举两个，并简要说明。
A5:
`useRef` 主要有两大类常见用途：

1. **访问和操作 DOM 元素**：
   - **说明**：这是 `useRef` 最广为人知的用途。通过将 ref 对象附加到 JSX 元素的 `ref` 属性上，React 会在 DOM 元素被挂载后，将该 DOM 节点的引用赋值给 `ref.current`。之后就可以通过 `ref.current` 直接访问和操作这个 DOM 节点，例如调用 `focus()` 方法、获取元素尺寸、触发动画等。
2. **存储可变的、与渲染无关的值**：
   - **说明**：用于存储那些需要在组件多次渲染之间保持一致，但其变化不应触发组件重新渲染的数据。
   - **具体例子**：
     - **存储定时器 ID**：`setInterval` 或 `setTimeout` 返回的 ID 可以存储在 `ref.current` 中，方便在组件卸载或特定条件下清除定时器，而存储 ID 本身不需要触发渲染。
     - **存储上一次的 props 或 state 值**：可以在 `useEffect` 中更新 `ref.current` 为当前的 props 或 state，这样在下一次渲染时，`ref.current` 仍然持有的是上一次渲染时的值，可用于比较或追踪变化。
     - **缓存某些计算结果或对象实例**：如果某个对象的创建成本较高，且不需要每次渲染都重新创建，并且它的变化也不直接驱动UI更新，可以考虑用 `useRef` 缓存。

Q6: 当你需要获取一个 DOM 元素的引用时，比如让一个输入框自动聚焦，你会如何使用 `useRef`？请描述关键步骤。
A6:
使用 `useRef` 让输入框自动聚焦的关键步骤如下：

1. **创建 ref 对象**：在组件内部调用 `useRef(null)` 来创建一个 ref 对象，例如 `const inputRef = useRef(null);`。初始值通常设为 `null`，因为在组件首次渲染完成前，DOM 元素还不存在。
2. **关联 ref 到 DOM 元素**：在 JSX 中，将创建的 ref 对象赋值给目标 DOM 元素的 `ref` 属性，例如 `<input ref={inputRef} type="text" />`。当 React 渲染这个 `input` 元素到 DOM 中后，它会自动将这个 DOM 节点的引用赋值给 `inputRef.current`。
3. **在 `useEffect` 中访问和操作 DOM**：通常在 `useEffect` Hook 中执行 DOM 操作。为了确保 DOM 元素已经挂载，可以将 `useEffect` 的依赖数组设置为空数组 `[]`（表示仅在组件挂载和卸载时运行）。在 `useEffect` 的回调函数内部，通过 `inputRef.current` 访问 DOM 节点，并调用其方法，例如 `inputRef.current.focus();`。需要进行空检查（`if (inputRef.current)`）以确保元素存在。

Q7: 使用 `useRef` 存储定时器 ID 比使用 `useState` 更好。请解释为什么?
A7:
使用 `useRef` 存储定时器 ID 比 `useState` 更好，主要是因为**避免不必要的组件重新渲染**。

- 定时器 ID（如 `setInterval` 或 `setTimeout` 返回的数字）本身是一个内部管理的值，用户通常不需要在 UI 上直接看到它。
- 如果使用 `useState` 来存储定时器 ID (例如 `const [timerId, setTimerId] = useState(null)`)，那么在 `useEffect` 中启动定时器并调用 `setTimerId(newTimerId)` 时，这个 `setTimerId` 调用会触发一次组件的重新渲染。
- 这次由 `setTimerId` 引起的重新渲染对于用户界面来说通常是多余的，因为它并不改变任何可见的 UI 内容，只是更新了一个内部追踪的 ID。
- 而 `useRef` (例如 `timerIdRef.current = newTimerId;`) 修改 `.current` 属性不会触发渲染，它能静默地保存和更新这个 ID，让开发者可以在需要时（如组件卸载时清除定时器）访问到它，而不会对组件的渲染性能产生额外开销。

Q8: 如何使用 `useRef` 来追踪一个 prop 或 state 的上一个值？请结合 `useEffect` 解释其工作原理。
A8:
可以使用 `useRef` 和 `useEffect` 结合来追踪一个 prop 或 state 的上一个值。工作原理如下：

1. **初始化 Ref**：创建一个 ref 对象，例如 `const prevValueRef = useRef();`。它的 `.current` 属性初始时是 `undefined`。
2. **在 `useEffect` 中更新 Ref**：
   - 设置一个 `useEffect` Hook，其依赖数组包含你想要追踪的那个 prop 或 state 值 (我们称之为 `currentValue`)。
   - 在 `useEffect` 的回调函数内部，将当前的 `currentValue` 赋值给 `prevValueRef.current`。例如：

```
useEffect(() => {
  prevValueRef.current = currentValue;
```

```
  }, [currentValue]);
```

3. **工作流程**：
   - **首次渲染**：组件渲染时，`prevValueRef.current` 是 `undefined` (或者你在 `useRef` 中设置的初始值)。渲染完成后，`useEffect` 执行，将当前的 `currentValue` 存入 `prevValueRef.current`。
   - **后续渲染 (当 `currentValue` 变化时)**：
     - 当 `currentValue` 发生变化，组件触发重新渲染。在这次重新渲染期间，当你访问 `prevValueRef.current` 时，它仍然持有的是 *上一次渲染后* `useEffect` 中存入的值，即变化前的 `currentValue`。这就是“上一个值”。
     - 这次渲染完成后，`useEffect` 会再次执行（因为 `currentValue` 在其依赖数组中并且发生了变化），此时它会将新的 `currentValue` 更新到 `prevValueRef.current`，为下一次可能的追踪做准备。

通过这种方式，`prevValueRef.current` 总是在当前渲染周期中反映上一个渲染周期的值，因为它总是在渲染完成 *之后* 才被更新。

Q9: 你认为 `useRef` 的设计哲学是什么？它在 React 的声明式编程范式中扮演了怎样的角色?
A9:
`useRef` 的设计哲学可以理解为提供了一种在 React 的声明式、函数式组件模型中，处理那些本质上是命令式或者需要持久化引用、但又不直接驱动 UI 状态变化的场景的“出口”或“桥梁”。

在 React 的声明式编程范式中：

- **声明式为主**：开发者主要通过声明期望的 UI 状态 (`props` 和 `state`)，React 负责高效地更新 DOM 以匹配这个状态。
- **`useRef` 的补充角色**：
  1. **逃逸舱口 (Escape Hatch) 到命令式操作**：对于某些无法完全通过声明式状态管理的场景，如直接操作 DOM（聚焦、媒体控制、动画触发）或集成第三方非 React 的库，`useRef` 提供了一个获取底层命令式句柄（如 DOM 节点）的方式。
  2. **组件实例的持久化“字段”**：在类组件中，我们可以使用实例属性（ `this.myValue` ）来存储那些不触发渲染的数据。`useRef` 在函数组件中扮演了类似的角色，提供了一个在多次渲染之间保持不变的容器 (`ref.current`)，用于存储这类数据，而不会因为其变化引起不必要的渲染，这符合 React 追求性能优化的目标。
  3. **保持纯粹性与副作用的分离**：通过将这类可变值或 DOM 引用封装在 `ref` 中，并在 `useEffect` 等副作用钩子中操作它们，有助于保持组件渲染逻辑的纯粹性。`ref` 的更新本身不触发渲染，使得副作用的管理更加可控。

总的来说，`useRef` 增强了函数组件的能力，使其能够处理一些传统上更适合类组件实例属性或命令式编程的场景，同时尽量保持在 React 声明式和响应式更新的核心框架内。它是对

`useState` 驱动的响应式状态系统的一个重要补充，用于管理那些“幕后”的、与 UI渲染解耦的可变数据和引用。