43.如何测试一个自定义 Hook?

Q1: 为什么为自定义 Hook 编写单元测试非常重要?
A1:
为自定义 Hook 编写单元测试非常重要，主要有以下几个原因：

1. **封装复杂逻辑**：Hook 往往封装了应用中最复杂的逻辑，如数据请求、状态管理等，是应用稳定性的基石。
2. **避免连锁 Bug**：Hook 的核心价值在于复用。一个有 Bug 的 Hook 会将问题扩散到所有使用它的组件中，造成严重后果。
3. **易于测试**：自定义 Hook 天然地将业务逻辑与 UI 界面分离，使其成为单元测试的理想对象。
4. **确保行为一致**：测试的最终目的是确保 Hook 在任何情况下其行为都和预期完全一致。

Q2: 测试自定义 Hook 时，我们面临的最大挑战是什么？为什么不能像测试普通函数一样直接调用它?
A2:
最大的挑战来自于 React 的核心规则：**Hooks 只能在 React 组件或另一个自定义 Hook 的函数体内调用**。因此，我们无法像测试普通 JavaScript 函数那样，在测试文件中简单地 `import` 并直接调用它。如果这样做，React 会抛出 "Invalid hook call" 的错误，因为它脱离了 React 组件的上下文环境。

Q3: 既然不能直接调用 Hook，那么测试 Hook 的正确思路是什么?
A3:
正确的思路是为 Hook 创建一个“宿主环境”。具体来说，就是在测试过程中，我们创建一个虚拟的测试组件，让这个组件去调用我们需要测试的 Hook。然后，我们通过测试这个虚拟组件的行为（例如它的状态变化、渲染输出等），来间接地验证我们的 Hook 是否工作正常。

Q4: 哪个库专门用于解决 Hook 的测试难题？它提供了什么核心能力?
A4:
`@testing-library/react-hooks` 是专门用于测试 Hook 的库。它的核心能力是**自动创建和管理测试所需的“宿主环境”**，将所有繁琐的设置工作都封装好了。这使得开发者可以摆脱手动创建测试组件的麻烦，专注于测试 Hook 本身的业务逻辑。

Q5: 请解释 `@testing-library/react-hooks` 中的核心 API `renderHook` 的作用和用法。
A5:
`renderHook` 是该库最核心的 API，用于在测试环境中渲染一个 Hook。

- **用法**：调用 `renderHook` 时，需要传入一个回调函数，并在该回调函数内部去调用你想要测试的自定义 Hook，例如 `renderHook(() => useCounter())`。
- **返回值**：`renderHook` 会返回一个对象，其中最重要的属性是 `result`。`result` 对象上又有一个 `current` 属性（即 `result.current`），它指向了 Hook 当前的返回值。无论是状态值还是操作函数，都可以从 `result.current` 上获取。

Q6: 在测试中，当我们调用一个会触发状态更新的函数（如 `increment`）后，为什么通常需要使用 `act` 工具函数?
A6:
这是因为在 React 中，由 `useState` 或 `useReducer` 触发的状态更新是**异步执行**的。如果在调用更新函数后立即进行断言，此时状态可能还没有完成更新，会导致测试失败。`act` 函数可以确保在其回调函数内部触发的所有 React 状态更新都已经处理完毕并同步到 DOM (虽然在 Hook 测试中主要是同步状态)，之后再执行的断言才能获取到最新的状态，从而保证测试的准确性。

Q7: 如何测试一个依赖外部 Props 的 Hook，特别是当这些 Props 发生变化时 Hook 的行为?
A7:
为了测试 Hook 对外部 Props 变化的响应，我们需要使用 `renderHook` 返回的 `rerender` 函数。具体步骤如下：

1. **设置初始 Props**：在调用 `renderHook` 时，通过其第二个参数（配置对象）的 `initialProps` 属性来提供 Hook 的初始属性。
2. **解构 rerender**：从 `renderHook` 的返回值中解构出 `rerender` 函数，例如 `const { result, rerender } = renderHook(...)`。
3. **模拟重渲染**：在测试用例中，调用 `rerender` 函数并传入一组新的 Props 对象，例如 `rerender({ newProp: 'newValue' })`。这个操作会模拟父组件因 Props 变化而发生的重渲染。
4. **断言**：在 `rerender` 调用之后，断言 Hook 的行为是否符合新 Props 下的预期。

Q8: 请详细描述如何为一个 `useCounter(initialValue)` Hook 编写一个测试用例，来验证当 `initialValue` 改变后，`reset` 函数会重置到这个新的初始值。
A8:
这个测试需要结合 `act` 和 `rerender` 来完成，步骤如下：

1. **初始渲染**：使用 `renderHook` 渲染 `useCounter`，并传入一个初始值，例如 `{ initialValue: 0 }`。
2. **改变状态**：使用 `act()` 包裹一个状态更新操作，例如调用 `increment()`，使计数器的值不再是初始值（变为 1）。
3. **模拟 Props 更新**：调用 `rerender` 函数，并传入一个新的 `initialValue`，例如 `rerender({ initialValue: 10 })`。这模拟了父组件传入了新的初始值。
4. **调用 reset 并断言**：再次使用 `act()` 包裹 `reset()` 函数的调用。在此 `act` 执行完毕后，断言 `result.current.count` 的值是否等于新的初始值 `10`，而不是旧的 `0`。

Q9: 如何测试一个包含异步操作（例如 `fetch` 数据）的自定义 Hook？需要用到哪些关键的测试工具?
A9:
测试异步 Hook，需要结合 `async/await` 和 testing-library 提供的异步工具函数，最常用的是 `waitFor`。

1. **将测试函数标记为 `async`**：测试用例 `test(...)` 或 `it(...)` 的回调函数需要用 `async` 关键字修饰。
2. **初始状态断言**：在 `renderHook` 之后，可以立即断言 Hook 的初始状态，例如 `expect(result.current.loading).toBe(true)`。
3. **使用 `waitFor` 等待异步完成**：使用 `await waitFor(() => ...)` 来包裹一个断言。`waitFor` 会在一定时间内反复执行这个断言，直到它通过为止。通常用它来等待异步操作结束的标志，例如 `await waitFor(() => expect(result.current.loading).toBe(false))`。
4. **最终结果断言**：一旦 `waitFor` 执行完毕，就说明异步流程已经走完。此时，可以安全地对最终的数据或状态进行断言，例如 `expect(result.current.data).toEqual(...)`。
5. **(可选但推荐)** 使用 `jest.mock` 或 `spyOn` 来 Mock 掉真实的 API 请求，使测试独立、快速且可预测。

Q10: 在面试中被问到：“在项目中，你如何保证自定义 Hook 的质量？你会如何对一个封装了异步请求的 Hook 进行测试？” 你会如何回答?
A1:
我会从以下几个层面进行回答，以展示逻辑的清晰性和实践经验的深度：

1. **强调重要性**：首先，我会明确指出自定义 Hook 作为项目中逻辑复用的核心，其稳定性至关重要，因此为其编写单元测试是项目质量保障体系中必不可少的一环。
2. **点明核心工具**：接着，我会说明我们团队主要使用 `@testing-library/react-hooks` 这个库来测试 Hook。并简述其优点：它能自动创建隔离的测试宿主环境，让我们能专注于测试 Hook 逻辑本身。
3. **分场景阐述测试策略**：这是回答的核心，我会分不同场景来阐述具体的测试方法：

- **初始状态**：使用 `renderHook` 渲染后，直接对 `result.current` 的初始值进行断言。
- **状态更新**：所有触发状态更新的操作（如调用返回的函数），都必须包裹在 `act()` 中，以确保在断言时能获取到最新的状态。
- **Props 依赖与更新**：对于依赖外部 Props 的 Hook，会使用 `rerender` 函数来模拟 Props 的变更，并验证 Hook 是否能正确响应。
- **异步逻辑测试（关键点）**：对于封装了异步请求的 Hook，测试函数会使用 `async/await`。我们会使用 `waitFor` 工具函数来等待异步操作（如 API 请求）完成，然后再对最终的状态（如 `loading` 变为 `false`，`data` 被填充）进行断言。同时，会强调使用 `jest.mock` 等工具来 Mock 真实的 API 请求，以保证测试的稳定和高效。