29.讲讲 RTK Query 如何简化数据获取和缓存逻辑。

Q1: 什么是 RTK Query？它主要解决了前端开发中的哪些痛点?
A1:
RTK Query 是 Redux Toolkit (RTK) 内置的一个强大的数据获取和缓存解决方案。它主要解决以下痛点：

- **手动管理加载状态**：无需手动维护 `isLoading`，`isError`，`isSuccess` 等状态。
- **重复的异步逻辑**：通过声明式API定义端点，无需为每个请求编写重复的 thunks 或异步函数。
- **复杂的缓存策略**：提供了自动化的智能缓存、后台更新和缓存失效机制。
- **组件间数据同步**：由于数据存储在Redux中，可以方便地在不同组件间共享和同步。
- **请求节流与防抖**：内置机制可以避免对相同数据发起重复请求。

Q2: 在使用 RTK Query 时，其核心工作流程是怎样的?
A2:
RTK Query 的核心工作流程主要分为三步：

1. **创建 API Slice**：使用 `createApi` 函数定义服务。这包括配置 `reducerPath` (在Redux state中的路径)、`baseQuery` (通常使用 `fetchBaseQuery` 设置基础URL) 以及最重要的 `endpoints` (定义所有的查询 `query` 和变更 `mutation`)。还可以定义 `tagTypes` 用于缓存管理。
2. **自动生成 Hooks**：一旦 `endpoints` 定义完成，RTK Query 会为每一个 `query` 自动生成一个 `use[EndpointName]Query` Hook，为每一个 `mutation` 自动生成一个 `use[EndpointName]Mutation` Hook。
3. **在组件中使用 Hooks**：在UI组件中直接调用这些自动生成的 Hooks。Query Hooks 会自动触发数据请求并返回 `data`，`isLoading`，`isError` 等状态。Mutation Hooks 返回一个触发函数和相关的状态，用于执行数据变更操作。

Q3: 请解释 RTK Query 中 `Tags` (标签) 的概念以及其工作原理。`providesTags` 和 `invalidatesTags` 是如何协同工作的?
A3:
`Tags` 是 RTK Query 实现精细化和自动化缓存管理的核心机制。

- **概念**：`Tags` 是你用来标记缓存数据的字符串或对象。通过给不同的缓存数据打上特定的标签，你可以在之后精确地使其失效。
- **`providesTags`**：通常在 `query` 类型的 endpoint 中使用。当一个查询成功获取数据后，`providesTags` 会根据返回的数据为这些缓存数据“提供”一个或多个标签。例如，获取所

有文章列表的请求可以提供一个 `{ type: 'Post', id: 'LIST' }` 的标签，获取单篇文章的请求可以提供一个 `{ type: 'Post', id: 1 }` 的标签。

- **`invalidatesTags`**：通常在 `mutation` 类型的 endpoint 中使用。当一个变更操作（如新增、更新、删除）成功后，`invalidatesTags` 会声明哪些标签对应的缓存数据应该“失效”。
- **协同工作原理**：当一个 `mutation` 成功执行并使其 `invalidatesTags` 列表中指定的某个标签（如 `{ type: 'Post', id: 'LIST' }`）失效时，RTK Query 会自动找到所有通过 `providesTags` 提供了该标签的 `query`，并强制它们重新获取数据。这就实现了“当新增一篇文章后，文章列表自动刷新”的效果，保证了数据的一致性。

Q4: 在将一个 `apiSlice` 集成到 Redux store 时，需要进行哪些关键配置？为什么这些配置是必需的?
A4:
将 `apiSlice` 集成到 Redux store 需要两项关键配置：

1. **添加 Reducer**：将 `apiSlice.reducer` 添加到 `configureStore` 的 `reducer` 对象中，并使用 `apiSlice.reducerPath` 作为键。
   - **原因**：这是必需的，因为它创建了 Redux state 的一个分支，用来存储该 API slice 相关的所有数据，包括缓存的响应、加载状态、错误信息等。
2. **添加 Middleware**：将 `apiSlice.middleware` 通过 `.concat()` 方法添加到 `getDefaultMiddleware` 链中。
   - **原因**：这个中间件是 RTK Query 的核心，它负责处理所有的异步请求、管理缓存的生命周期（何时存储、何时失效）、实现轮询、后台重新获取数据等高级功能。没有这个中间件，RTK Query 将无法工作。

```
// 示例代码
export const store = configureStore({
  reducer: {
    // 1. 添加 Reducer
    [postsApi.reducerPath]: postsApi.reducer,
  },
  // 2. 添加 Middleware
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(postsApi.middleware),
});
```

Q5: 当你在面试中被问到 “RTK Query 的核心优势是什么” 时，你会如何回答?
A5:
我会这样结构化地回答：

"RTK Query 是一个强大的声明式数据获取库，它极大地简化了与服务器的交互，其核心优势体现在以下几个方面：

- **极致简化与自动化**：它通过自动生成 Hooks，帮我们处理了请求的加载状态（`isLoading`）、错误处理以及最重要的——智能缓存和自动重新获取逻辑，从而大幅减少了数据获取的样板代码。
- **智能的缓存管理**：它内置了高效的自动缓存机制。特别是其基于标签(`Tags`)的缓存失效系统，通过 `providesTags` 和 `invalidatesTags` 的配合，可以非常方便地管理数据一致性。例如，在一个列表页添加新项目后，相关列表可以自动刷新，无需手动干预。
- **与Redux生态无缝集成**：作为 Redux Toolkit 的一部分，它能与 Redux DevTools 完美配合，让数据流和API请求状态的追踪与调试变得非常清晰和便捷。
- **丰富的功能集**：它开箱即用地提供了许多高级功能，如乐观更新、条件性获取（`skip`）、请求轮询（`polling`）以及多种后台静默更新策略（如 `refetchOnFocus`，`refetchOnReconnect`），能满足复杂的业务场景需求。
  总而言之，RTK Query 让开发者能更专注于业务逻辑，而不是数据流管理的细节，显著提升了开发效率和代码质量。"

Q6: RTK Query 与 React Query (TanStack Query) 的主要区别是什么?
A6:
它们的主要区别在于设计哲学和生态集成：

- **状态管理与集成**：这是最核心的区别。RTK Query 是 Redux Toolkit 的一部分，其所有缓存和状态都统一存储在全局的 Redux store 中。这使得它与已在使用 Redux 的项目集成非常自然。而 TanStack Query 是一个独立的、与UI框架无关的库，它在内部自行管理缓存状态，不依赖于任何全局状态管理器，当然也可以与 Redux 等库配合使用。
- **API 设计**：虽然都使用 Hooks，但 API 设计略有不同。RTK Query 通过 `createApi` 集中定义所有 endpoints，更偏向于一种“服务层”的定义方式。TanStack Query 则更为灵活，可以在组件中直接使用 `useQuery` 并传入一个 key 和一个异步函数，无需预先定义。
- **生态系统**：RTK Query 深度集成于 Redux 生态，可以充分利用 Redux 的中间件、DevTools 等工具。TanStack Query 自身就是一个强大的生态，拥有自己的 DevTools，并且可以轻松用于 React、Vue、Svelte 等多种环境。

Q7: RTK Query 适合哪些应用场景？在哪些情况下可能不是最佳选择?
A7:
适合的场景：

- **重度依赖 Redux 的项目**：如果你的应用已经在使用 Redux 或计划使用 Redux 进行复杂的全局状态管理，RTK Query 是一个非常自然和强大的选择。
- **需要标准化数据获取方案的团队/项目**：它提供了一种统一、规范的方式来定义和管理所有API交互，有助于提升大型项目和团队协作的可维护性。

- **需要丰富缓存和自动刷新功能的复杂应用**：对于需要处理复杂数据依赖、缓存失效和实时更新的场景，RTK Query 基于标签的系统提供了强大的支持。

可能不适合的场景：

- **非常简单的应用**：对于只有少数几个API请求、不需要全局状态管理的小型项目或静态网站，引入 Redux 和 RTK Query 可能会显得过于笨重（overkill）。
- **已有成熟数据获取方案的项目**：如果团队已经有了一套运行良好且熟悉的自定义数据获取方案（例如基于 Axios interceptors + SWR/React Query），除非遇到难以解决的痛点，否则没有必要强制迁移。
- **不使用 Redux 的项目**：如果项目完全没有使用或不打算使用 Redux，那么选择像 TanStack Query 这样独立的数据获取库会更加轻量和直接。

Q8: 如何自定义 `fetchBaseQuery` 来处理全局的请求头（如添加认证Token）或统一的错误处理?
A8:
可以通过 `fetchBaseQuery` 提供的参数来进行全局定制：

- **处理全局请求头**：可以使用 `prepareHeaders` 参数。它是一个函数，接收当前的 `headers` 对象和一个包含 `getState` 等方法的 `api` 对象。你可以在这个函数中修改并返回新的 `headers`，非常适合动态添加认证 `Token`。

```
fetchBaseQuery({
  baseUrl: '/',
  prepareHeaders: (headers, { getState }) => {
    const token = (getState()).auth.token; // 从 Redux state 中获取
token
    if (token) {
      headers.set('authorization', `Bearer ${token}`);
    }
    return headers;
  },
})
```

- **统一错误处理**：一种方式是创建一个包装了 `fetchBaseQuery` 的高阶函数。在这个函数内部，你可以 `await` 基础查询的结果，并对可能出现的错误进行捕获和统一处理（例如，全局的 toast 提示、登出操作等），然后再将结果或错误返回。另一种方法是使用 RTK Middleware 监听 `isRejected` action，进行全局错误处理。