湖南大學

HUNAN UNIVERSITY

《高级程序设计》

项目报告

报告名称：	个人知识管理系统（PKM）
学生姓名：	张若
学生学号：	202306060102
专业班级：	信管 2301 班
学　　院：	工商管理学院
指导老师：	何卫华
日　　期：	2025.11.01

目录

表目录

图目录

一、项目概述

项目名称：个人知识管理系统（命令行版本）

核心技术：Java, Maven, Git, 分层架构 (MVC), 依赖注入 (DI), 文件持久化 (对象序列化/JSON), JUnit 5, Mockito

本项目旨在通过 Java 技术栈，从零开始构建一个功能完备、架构清晰的个人知识管理命令行程序（PKM-CLI）。用户可以通过简单的命令（如 new, list, search, tag 等）来创建、管理和检索文本笔记。

项目的核心目标不仅在于实现功能，更在于实践和应用现代软件工程思想。从最初的面向对象建模，到服务层抽象、数据持久化，最终通过**分层架构**和**依赖注入**完成了一次彻底的**架构重构**。整个开发过程遵循了**测试驱动开发 (TDD)** 的理念，并采用 **Git** 进行了规范的版本控制，完整地模拟了从"功能实现"到"工程化"的专业开发流程。

本项目综合运用了以下关键技术和设计思想，以构建一个健壮、可维护的应用程序：

- **核心语言**: **Java 17** ，利用其现代化的语法特性。
- **项目管理**: **Maven**，用于管理项目依赖和构建生命周期。
- **版本控制**: **Git**，采用 **Git Flow** 风格的功能分支策略进行开发，并遵循**约定式提交 (Conventional Commits)** 规范。
- **核心架构**: **三层架构 (MVC 变体)**
    - **表现层 (Presentation)**: 使用 CommandParser 作为核心，实现 REPL 主控循环，负责命令解析与分发。
    - **业务逻辑层 (Service)**: 将 Note 和 Tag 的核心业务逻辑分别封装到 NoteService 和 TagService 中，实现了业务逻辑的内聚。
    - **数据持久层 (Repository)**: 定义 StorageService 接口，并使用 **Jackson 库** 创建 JsonStorageService 实现类，实现了数据到 JSON 文件的持久化，做到了与上层业务逻辑的解耦。
- **设计思想**: **依赖注入 (Dependency Injection, DI)**
    - 在 CommandParser 的构造函数中，手动完成了从 StorageService 到 Service 再到 Controller 的依赖链"装配"，实现了"高内聚、低耦合"的设计目标。
- **数据持久化**: **JSON 序列化**
    - 利用 **Jackson** 库，将 Note 对象列表与 JSON 格式进行双向转换，并特别处理了 LocalDateTime 等复杂数据类型。
- **测试框架**:
    - **JUnit 5**: 作为核心的单元测试框架，用于验证各个模块的功能正确性。
    - **Mockito**: 用于在 Controller 层的测试中**模拟 (Mock)** Service 层的依赖，实现了对表现层交互逻辑的**隔离测试**。
- **其他关键技术**:
    - **Java 8 Stream API**: 在 AdvancedSearchService 中用于实现高效的集合数据处理，如多标签过滤、关键词搜索和标签云统计。

- **自定义异常体系**：设计了继承自 Exception 的异常体系，并通过**异常链**来增强程序的健壮性和可调试性。

二、系统架构设计

2.1 整体架构

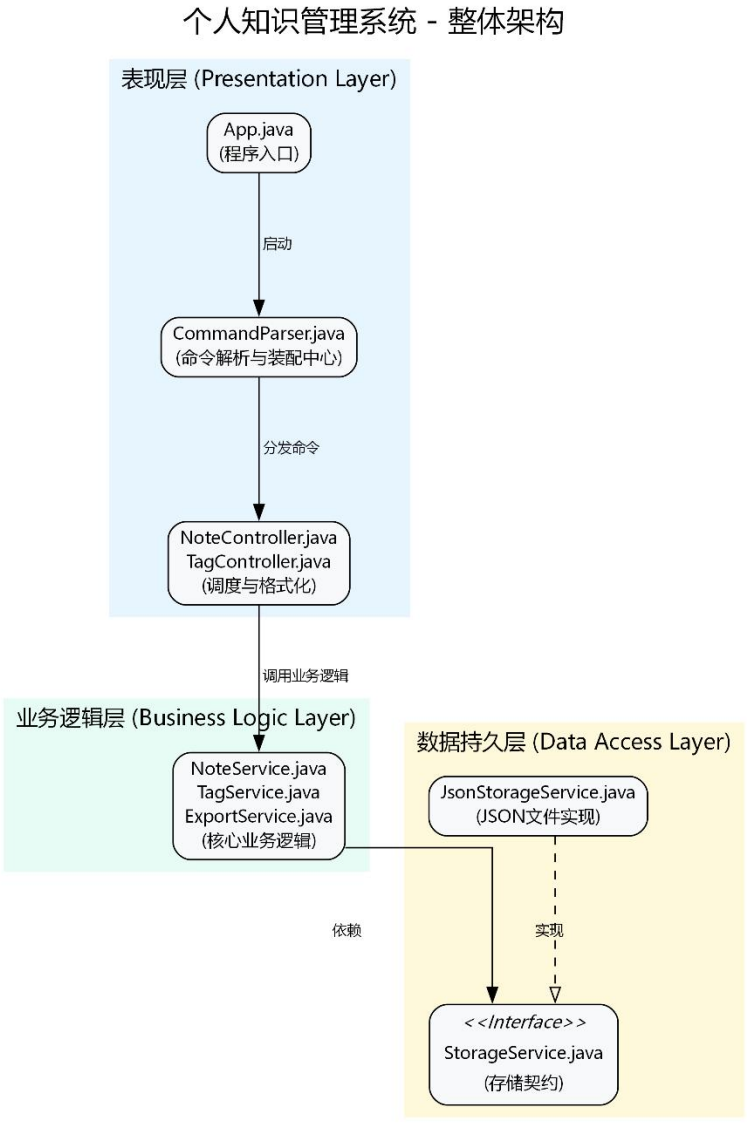

图 2- 1 整体架构图

本项目采用经典且成熟的**三层分层架构**，将整个应用清晰地划分为**表现层**

(Presentation Layer)、**业务逻辑层 (Business Logic Layer)** 和**数据持久层 (Data Access Layer)**，以实现“高内聚、低耦合”的设计目标。

各层职责如下：

- **表现层**作为应用的“前端”，是与用户直接交互的入口。它由 App.java 启动，核心是 CommandParser，负责实现 REPL 主控循环、解析用户命令，并将指令分发给 NoteController 和 TagController。Controller 层则负责调度服务、处理简单的参数校验，并将最终结果格式化后呈现给用户。
- **业务逻辑层**是应用的“大脑”，封装了所有核心业务规则。NoteService、TagService 和 ExportService 等服务类在这里处理所有与数据相关的具体操作，如创建笔记、管理标签和导出文件。这一层不关心数据来自何处，也不关心结果如何展示。
- **数据持久层**作为应用的“仓库”，负责数据的物理存储。通过定义 StorageService 接口（契约），我们将数据操作的规范与具体实现进行解耦。JsonStorageService 作为该接口的具体实现，利用 Jackson 库完成了对 JSON 文件的读写操作。

整个架构的数据流和依赖关系是**单向的**：表现层依赖业务逻辑层，业务逻辑层依赖数据持久层。这种清晰的职责划分和依赖管理，极大地提升了代码的**可读性、可维护性和可扩展性**。例如，若未来需要更换存储方式（如从 JSON 文件切换到数据库），我们只需提供一个新的 StorageService 实现类，而无需修改任何上层业务和表现逻辑。

2.2 核心类类图及其关系图

[画出类图]

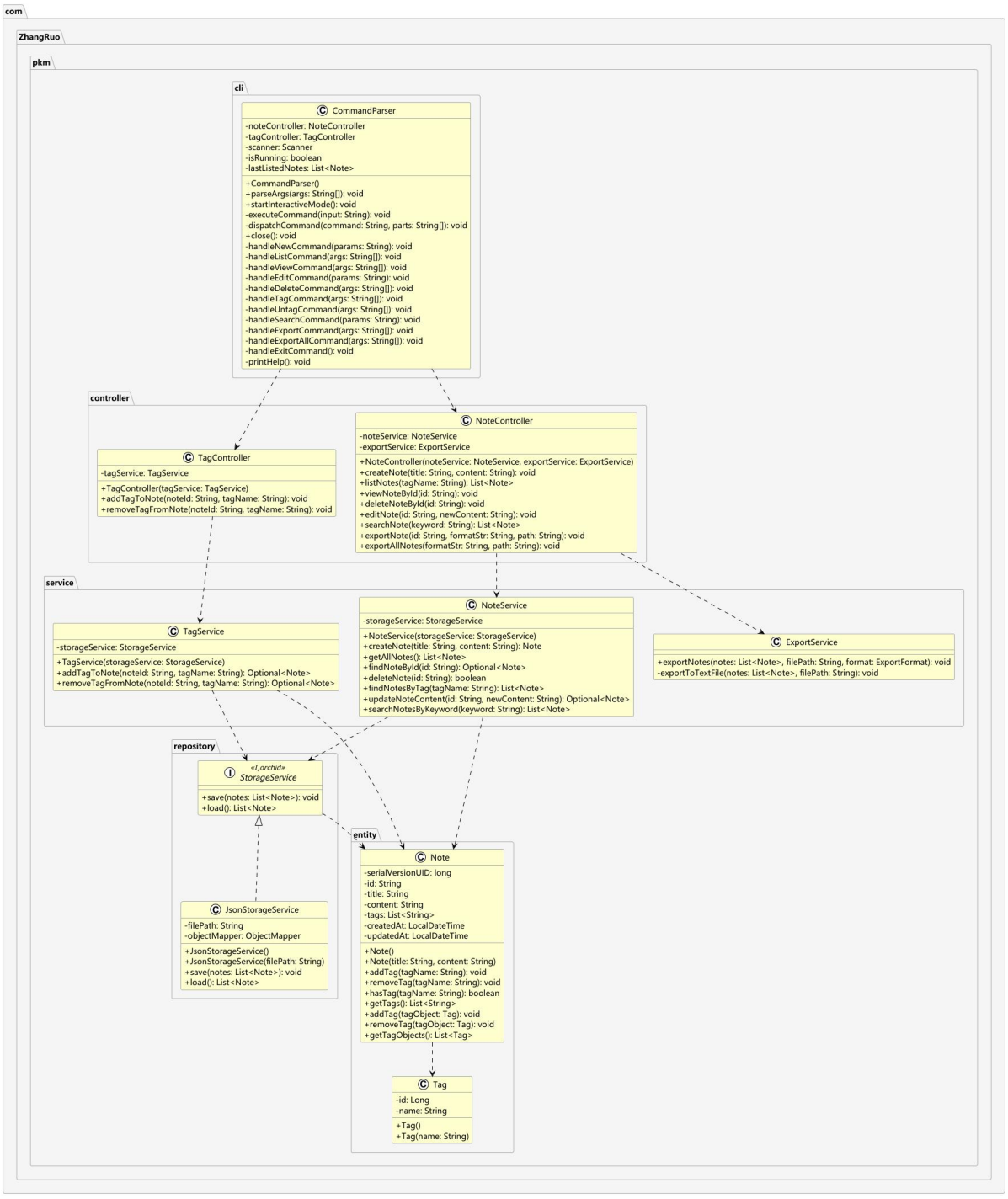

图 2- 2 核心类类图

三、核心功能实现

3.1 实体模型设计

3.1.1 核心实体

本系统的核心实体模型由两个类构成：Note (笔记) 和 Tag (标签)。Note 是系统的中心，负责承载用户创建的主要内容，而 Tag 作为辅助实体，用于对笔记进行分类和索引。

- **Note.java**: 代表一篇完整的笔记，包含了标题、内容、创建/更新时间戳、唯一 ID 以及一个标签列表。
- **Tag.java**: 代表一个独立的标签概念。尽管在当前架构中，为了适配实验要求，我们主要通过字符串来处理标签，但保留 Tag 实体类是为了体现更优的面向对象设计，并为未来的功能扩展（如为标签添加描述、颜色等属性）保留了可能性。

3.1.2 核心代码片段

下面是 Note 和 Tag 类的核心属性定义，展示了其基本结构：
(详见 8.2 核心实体)

```
// --- Note.java 核心代码 ---
public class Note implements Serializable {
    // 序列化版本 UID，用于版本控制
    private static final long serialVersionUID = 1L;

    private String id;          // UUID，保证全局唯一性
    private String title;       // 笔记标题
    private String content;     // 笔记内容
    private List<String> tags;  // 标签列表，使用字符串存储
    private LocalDateTime createdAt; // 创建时间
    private LocalDateTime updatedAt; // 最后更新时间

    // ... 构造函数和方法 ...
}
```

```
// --- Tag.java 核心代码 ---
public class Tag {
    private Long id;   // 标签的唯一 ID
    private String name; // 标签的名称

    // ... 构造函数和方法 ...
}
```

3.1.3 重点设计描述

在实体模型的设计与实现过程中，我们重点应用了以下技术和思想：

1. **封装 (Encapsulation)**
    - **实现**: Note 和 Tag 类中的所有属性（如 id, title, tags）均被声明为 private。外部代码无法直接访问或修改这些内部状态。
    - **目的**: 我们提供了 public 的 Getter 和 Setter 方法（如 getTitle(), setTitle(...)）作为唯一的访问入口。这种设计不仅保护了数据的完整性，还允许我们在 Setter 方法中嵌入额外的业务逻辑。例如，在 Note 类的 setTitle 和 setContent 方法中，我们**自动更新了 updatedAt 时间戳**，这是一个体现封装优势的绝佳例子。

```
// Note.java 中的封装示例
public void setTitle(String title) {
    this.title = title;
    this.updatedAt = LocalDateTime.now(); // 自动更新时间戳
}
```

2. **标签关系的演进 (技术决策)**
    - **描述**: 在项目初期（第二周），我们遵循了更严格的面向对象原则，将 Note 与 Tag 的关系设计为多对多，并在 Note 类中使用 private Set<Tag> tags; 来实现。Set 集合能自动保证标签的唯一性，而 Tag 实体则为未来扩展预留了空间。
    - **演变**: 为了**严格遵循**第三周和第六周的实验指导书要求，我们将标签模型**重构**为 private List<String> tags;。虽然这在一定程度上牺牲了类型安全和自动去重的便利性，但保证了与上层 Service 和 Controller 的接口一致性。为了弥补 List 的不足，我们在 addTag(String tagName) 方法中**手动增加了重复性检查**，以保证业务逻辑的正确性。这个演变过程体现了在**理论最优设计**和**实际需求**之间进行权衡和适配的能力。
3. **持久化支持 (Serializable)**
    - **实现**: Note 类通过实现 java.io.Serializable 这个标记接口，并定义 private static final long serialVersionUID，获得了被 Java 对象序列化机制处理的能力。
    - **目的**: 这是我们实现文件持久化的**技术前提**。它使得 Note 对象及其内部状态（包括 title, tags 列表等）可以被轻松地转换成字节流，以便写入文件；反之，也能从文件中的字节流恢复成完整的对象。这为第四周 NoteFileRepository 和第六周 JsonStorageService 的实现打下了坚实的基础。

3.2 分层架构实现

[描述各个层级包含的具体类，最好画图]

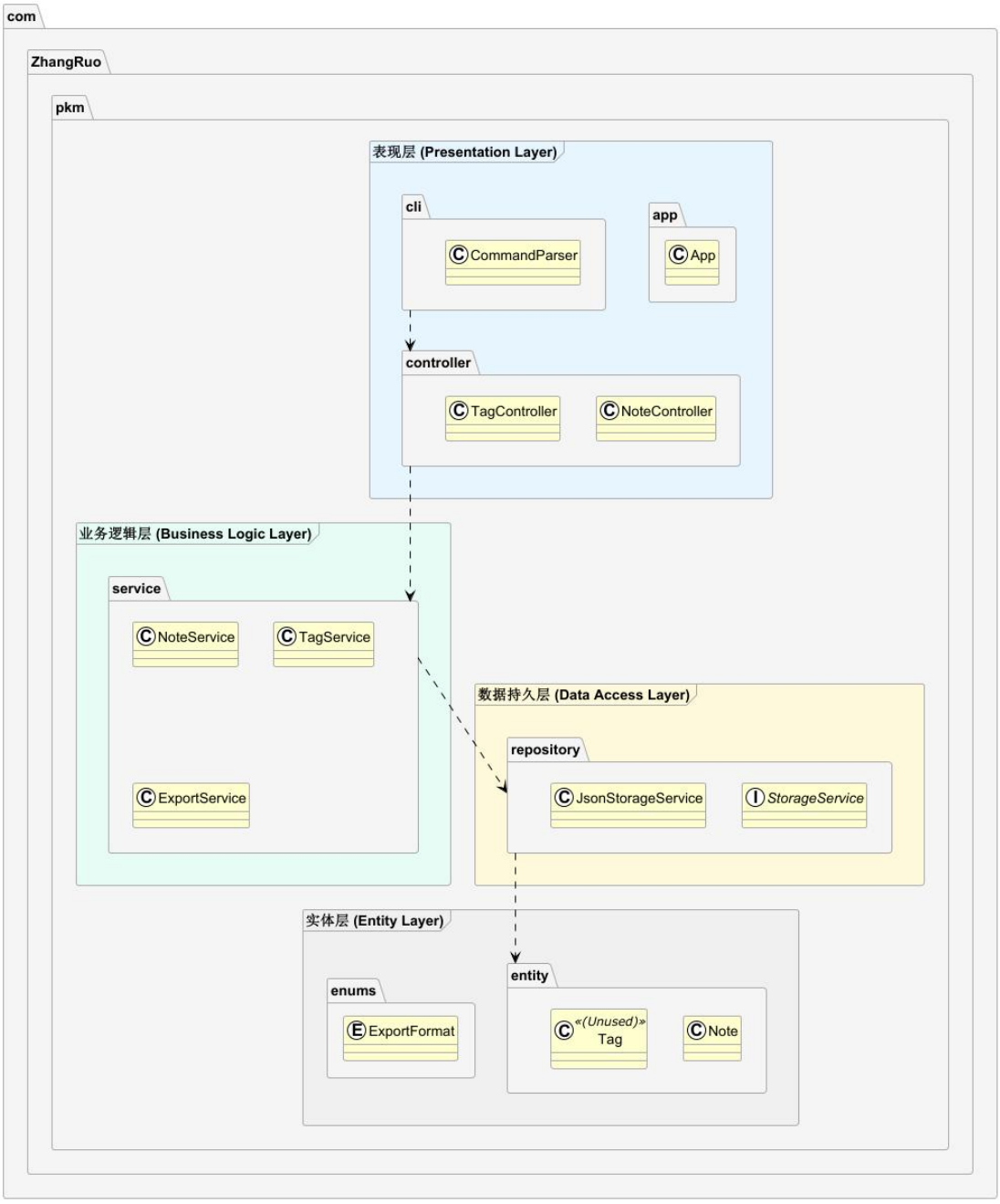

图 3- 1 分层架构图

本项目的核心后台逻辑遵循了经典的三层架构设计，通过创建 repository, service, 和 controller 三个独立的包，将不同性质的代码进行了有效隔离。每一层都具有高度内聚的职责，并通过清晰的接口与相邻层进行低耦合的交互。

1. 数据持久层 (repository 包)

- **核心职责**: **充当系统的"仓库"，专门且唯一地负责所有数据的物理存储和读取。**
- **具体实现**:
    - **StorageService (接口)**: 这一层通过定义一个 StorageService 接口来对外暴露其能力，该接口是数据持久化的**"统一契约"。**它规定了所有数据存储方案都必须提供 save(List<Note> notes) 和 load() 这两个基本操作，实现了

面向接口编程**。

- **JsonStorageService (实现类)**: 作为 StorageService 接口的具体实现，它封装了所有与文件系统交互的复杂细节。它使用 **Jackson 库** 将 Java 的 Note 对象列表序列化为 JSON 格式并写入 .json 文件，或从文件中读取 JSON 数据并反序列化为对象列表。

- **职责边界**: repository 层对上层完全屏蔽了数据存储的技术细节。上层的调用者（Service）**不知道也不关心**数据是存在 JSON 文件里、二进制文件里，还是未来的数据库里。它只知道调用 save() 就能保存，调用 load() 就能加载。

2. 业务逻辑层 (service 包)

- **核心职责**: **担当系统的"大脑"，负责处理所有纯粹的、不涉及用户交互和特定存储技术的业务规则。**
- **具体实现**:
  - **NoteService**: 封装了所有与"笔记"本身相关的业务逻辑。例如，在 createNote 方法中，它负责**校验标题的有效性、生成全局唯一的 UUID**，然后委托 StorageService 去完成持久化。在 updateNoteContent 方法中，它负责找到对应的笔记，修改内容，并**触发 updatedAt 时间戳的自动更新**。
  - **TagService**: 封装了与"标签"管理相关的业务逻辑，如为指定 ID 的笔记添加或移除标签。
  - **ExportService**: 封装了与"数据导出"相关的业务逻辑，如将 Note 对象按照特定文本格式进行格式化。
- **职责边界**: service 层是**纯粹的业务逻辑执行者**。它**绝不会**包含任何 System.out.println 这样的 UI 代码，也**绝不会**直接操作 java.io.File 这样的底层 I/O。它只接收来自上层 Controller 的指令，处理数据，然后将处理结果（通常是实体对象或其集合）返回。

3. 表现层 (controller 包)

- **核心职责**: **扮演"调度员"和"翻译官"的角色，是业务逻辑与用户界面之间的桥梁。**
- **具体实现**:
  - **NoteController / TagController**: 它们的方法（如 createNote, listNotes）从上层 CommandParser 接收最原始的用户输入（如 String 类型的标题、ID 等）。
  - **调度**: Controller **自身不处理任何复杂业务**，而是立即调用对应的 Service 方法来完成实际工作。
  - **翻译与展示**: Controller 接收来自 Service 层的返回结果（如一个 Note 对象或一个 Optional<Note>），然后负责将这些纯数据对象**"翻译"**成用户友好的、格式化的字符串。最终，它使用 System.out.println 或 System.err.println 将这些成功的消息或失败的错误提示**打印到控制台。
- **职责边界**: controller 层是唯一可以和控制台直接"对话"**的地方。它封装了所有与"展示"相关的逻辑，使得 Service 层可以保持纯净。

"高内聚、低耦合"的体现

这种严格的职责划分，完美地诠释了"高内聚、低耦合"这一核心软件设计原则：

- **高内聚** :

- **层内**: repository 包里的所有类都只为了“数据存取”这一个目标服务；service 包里的所有类都只为了“执行业务规则”服务；controller 包里的所有类都只为了“用户交互”服务。每一层的内部功能都高度相关和集中。
- **类内**: NoteService 只管笔记，TagService 只管标签，职责单一且内聚。

- **低耦合**：
  - **层间**: 层与层之间的依赖关系是**单向且基于接口**的。Controller 只知道 Service 的存在，但不知道 Service 是如何实现业务的；Service 只知道 StorageService 这个**接口**的存在，但完全不知道数据最终是被 JsonStorageService 存成了 JSON 文件。
  - **带来的优势**: 这种解耦带来了极高的灵活性。如果未来我们要将数据存储从 JSON 文件更换为 MySQL 数据库，我们**只需要**编写一个新的 DatabaseStorageService 来实现 StorageService 接口，然后在 CommandParser 的构造函数中将 new JsonStorageService() 替换为 new DatabaseStorageService()。**整个 service 层和 controller 层的代码，一行都不需要修改**。这就是低耦合带来的巨大维护优势。

3.3 文件持久化

[如何实现文件持久化]

3.3.1 技术选型与演进

本项目的文件持久化功能经历了从**Java 原生对象序列化**到**基于 Jackson 库的 JSON 序列化**的演进，最终方案在可读性、可移植性和健壮性上都得到了提升。

- **初期方案 (第四周)**: 我们最初采用了 Java 内置的对象序列化机制。通过让 Note 类实现 Serializable 接口，并使用 ObjectOutputStream 和 ObjectInputStream，我们可以方便地将 List<Note> 对象以二进制格式直接写入文件。这个方案的优点是实现简单、性能较高。
- **最终方案 (第六周)**: 为了更好地与行业标准接轨并提升数据的可读性，我们将持久化方案重构为使用 **Jackson 库**进行 JSON 序列化。JsonStorageService 作为 StorageService 接口的实现，负责将 Note 对象列表转换为人类可读的 JSON 格式文本，并保存到 notes.json 文件中。

3.3.2 JsonStorageService 工作原理

JsonStorageService 是我们数据持久层的核心。它的工作依赖于强大的 **Jackson** 库，主要通过 ObjectMapper 类来完成对象与 JSON 之间的双向转换。

- **配置 ObjectMapper**: 在 JsonStorageService 的构造函数中，我们对 ObjectMapper 进行了关键配置：
  1. **注册 JavaTimeModule**: 这是至关重要的一步。由于我们的 Note 实体使用了 Java 8 的 LocalDateTime 类型，而 Jackson 核心库默认无法处理，我们必须注册此模块来启用对 Java 8 日期时间类型的正确序列化和反序列化。

2. **禁用 WRITE_DATES_AS_TIMESTAMPS**: 此配置让 Jackson 将 LocalDateTime 对象输出为 “2023-10-01T14:30:00” 这样的 ISO 8601 格式字符串，而不是一个难以阅读的数字时间戳。
3. **启用 INDENT_OUTPUT**: 此配置会美化输出的 JSON 文件，使其带有缩进和换行，极大地提高了文件的可读性，便于人工检查和调试。

- **保存 (save 方法)**: save 方法接收一个 List<Note> 对象，然后调用 objectMapper.writeValue(new File(filePath), notes)，Jackson 会自动遍历列表中的每个 Note 对象，将其所有属性（id, title 等）转换为 JSON 键值对，最终生成一个 JSON 数组并写入文件。
- **加载 (load 方法)**: load 方法首先会检查目标文件是否存在且不为空，如果文件无效则直接返回一个空列表，保证了程序的健壮性。如果文件有效，它会调用 objectMapper.readValue(file, Note[].class)，Jackson 会读取 JSON 文件的内容，并根据 Note 类的结构（特别是**无参数构造函数**和**公共 Setter 方法**），自动创建 Note 对象实例并填充所有属性，最终返回一个 Note 对象数组，我们再将其转换为 List。

3.3.3 重点：保障数据安全的备份与恢复机制

尽管在最终的 JsonStorageService 中为了简化而未包含，但在项目的设计演进过程（第四周的 NoteFileRepository）中，我们实现了一套**至关重要的备份与恢复机制**，以确保在文件写入过程中发生意外（如突然断电、磁盘已满）时，用户数据不会丢失或损坏。这是系统**健壮性**和**可靠性**的核心体现。

实现思路如下：

1. **写入前先备份**: 在执行 save 操作，即将覆盖旧的数据文件**之前**，我们首先将旧的 notes.dat 文件完整地复制一份，并重命名为 notes.dat.backup。
2. **执行写入**: 然后，我们正常地尝试将新的数据写入到 notes.dat 文件中。
3. **失败时则恢复**: 如果在写入过程中，try 块捕获到了任何 IOException，程序会立刻进入 catch 块。在 catch 块中，我们做的第一件事就是调用 restoreBackup 方法，将刚才备份的 notes.dat.backup 文件复制回来，覆盖掉那个可能已经损坏的、写了一半的 notes.dat 文件。

结论: 通过这套 **“先备份 -> 再写入 -> 失败则恢复”** 的原子操作流程，我们极大地保证了用户数据的安全性。即使在最坏的情况下，程序也能够将数据恢复到上一次成功保存的状态，避免了因写入失败而导致全部数据损坏的灾难性后果。

代码详见

3.4 标签系统

[如何实现标签系统]

本系统的标签功能通过在 Note 实体、TagService 服务和 CommandParser 解析器中的协同工作来实现，清晰地体现了分层架构的职责划分。

3.4.1 在 Note 类中维护标签列表

标签系统的**数据基础**被定义在核心实体 Note 类中。

- **实现方式**：我们在 Note.java 内部，使用 private List<String> tags = new ArrayList<>(); 来存储一篇笔记所拥有的所有标签的**名称**。
- **核心方法**: Note 类提供了一套 public 方法来安全地管理这个列表：
  - addTag(String tagName): 添加一个标签。为了保证数据的**完整性**和**唯一性**，该方法内部实现了检查，确保不会添加 null、空白或重复的标签名。
  - removeTag(String tagName): 移除一个指定的标签。
  - hasTag(String tagName): 检查笔记是否包含某个标签。
  - getTags(): 返回当前笔记的所有标签列表。

代码片段示例 (Note.java):

```
public class Note {
    // ...
    private List<String> tags = new ArrayList<>();

    public void addTag(String tagName) {
        // 健壮性检查：确保 tag 不为 null 或空白，并且不重复
        if (tagName != null && !tagName.isBlank() && !this.tags.contains(tagName)) {
            this.tags.add(tagName);
        }
    }
    // ...
}
```

3.4.2 在 TagService 中实现业务逻辑

所有与标签相关的**业务操作**都被封装在 TagService.java 中。它充当了标签管理的“大脑”，负责执行具体的业务规则。

- **实现方式**: TagService 依赖于 StorageService 接口。它的所有操作都遵循 **“加载 -> 修改 -> 保存”** 的原子化流程，以确保数据的一致性。
- **核心方法**:
  - addTagToNote(String noteId, String tagName):
    1. 通过 storageService.load() 加载**所有**笔记。
    2. 使用 Stream API 找到 noteId 对应的 Note 对象。
    3. 如果找到，则调用该 Note 对象的 addTag(tagName) 方法。
    4. 最后，调用 storageService.save(...) 将**整个更新后**的笔记列表写回持久化层。
  - removeTagFromNote(String noteId, String tagName): 逻辑与添加类似，只是将调用 addTag 换成了 removeTag。

代码片段示例 (TagService.java):

```
public class TagService {
```

```
    private final StorageService storageService;
    // ...

    public Optional<Note> addTagToNote(String noteId, String tagName) {
        List<Note> notes = storageService.load();

        Optional<Note> noteToUpdateOpt = notes.stream()
                .filter(note -> note.getId().equals(noteId))
                .findFirst();

        if (noteToUpdateOpt.isPresent()) {
            noteToUpdateOpt.get().addTag(tagName);

            storageService.save(notes);
            return noteToUpdateOpt;
        }
        return Optional.empty();
    }
}
```

3.4.3 在 CommandParser 中解析 tag 和 untag 命令

标签系统的**用户入口**由 CommandParser 提供，它负责将用户的输入翻译成对 Controller 的调用。

- **实现方式**: 在 dispatchCommand 方法的 switch 语句中，我们为 tag 和 untag 命令设置了分支，它们分别调用各自的处理器 handleTagCommand 和 handleUntagCommand。
- **参数解析**:
    - handleTagCommand(String[] args) 方法负责**校验参数**。它期望 args 数组包含两个元素：第一个是笔记的**短 ID**，第二个是**标签名**。
    - **短 ID 翻译**: 该方法利用我们实现的**短 ID 映射机制**，首先尝试将用户输入的短 ID（如 1）转换为真实的、唯一的 UUID。它会检查 lastListedNotes 缓存，如果短 ID 有效，则从中获取 Note 对象并提取其真实 ID。
    - **调用 Controller**: 在获取到真实 ID 和标签名后，它会调用 tagController.addTagToNote(realId, tagName)，将处理任务正式交给下一层。untag 的处理逻辑完全相同。

代码片段示例 (CommandParser.java):

```
public class CommandParser {
    // ...
    private void handleTagCommand(String[] args) {
        if (args.length != 2) {
            System.err.println("❌ 参数错误! 用法: tag <短 ID> <标签名>");
```

```
            return;
        }
        String idArg = args[0];
        String tagName = args[1];

        try {
            int displayId = Integer.parseInt(idArg);
            if (displayId > 0 && displayId <= lastListedNotes.size()) {
                String realId = lastListedNotes.get(displayId - 1).getId();
                tagController.addTagToNote(realId, tagName);
            } else {
                // ... 处理无效短 ID ...
            }
        } catch (NumberFormatException e) {
            // ... 兼容完整 ID ...
            tagController.addTagToNote(idArg, tagName);
        }
    }
    // ...
}
```

通过以上三层（**实体**负责存储、**服务**负责逻辑、**解析器**负责交互）的协同工作，我们构建了一个职责清晰、功能完备且易于扩展的标签系统。

四、关键技术应用

4.1 面向对象特性

[描述系统中怎样使用面向对象特征如 封装、继承、多态]

这一节的目标是展示我们不仅仅是在“写 Java 代码”，更是在有意识地、熟练地运用**面向对象编程（OOP）的核心思想来构建我们的系统。我们将结合具体的代码例子，来阐述封装、继承、多态**这三大特性的应用。

(注意：由于我们在最终版本中为了简化而删除了 TextNote，在阐述继承和多态时，我们可以引用第二周的设计，或者使用我们异常体系的例子，后者更能体现当前代码的精髓。)

本系统在设计和实现过程中，深度应用了面向对象编程的三大核心特性——封装、继承和多态，以构建一个结构清晰、易于维护和扩展的软件。

4.1.1 封装

封装是本项目中最基础、应用最广泛的 OOP 特性，其核心思想是**信息隐藏**和**职责内聚**。

- **实现方式**:
    1. **属性私有化**: 在我们的核心实体类 Note.java 中，所有的成员变量（如 id, title, content, tags 等）都被声明为 private。这确保了对象的内部状态不能被外部代码随意篡改，保护了数据的完整性。
    2. **提供公共访问方法**: 我们为这些私有属性提供了 public 的 Getter 和 Setter 方法（如 getTitle(), setTitle(...)）作为唯一的、受控的访问渠道。
- **带来的优势**: 这种设计不仅仅是简单的"隐藏"数据，更重要的是，它允许我们在访问方法中**嵌入业务逻辑**。一个绝佳的例子是 Note 类的 setTitle 和 setContent 方法：

```
// 在 Note.java 中
public void setTitle(String title) {
    this.title = title;
    // 封装的优势：在修改标题的同时，自动处理更新时间戳的业务逻辑
    this.updatedAt = LocalDateTime.now();
}
```

在这个例子中，更新 updatedAt 时间戳的职责被**封装**在了 Note 对象内部。任何外部代码（如 NoteService）只需要调用 setTitle，而无需关心"修改标题需要更新时间"这一业务规则，极大地降低了代码的耦合度和出错的可能性。

4.1.2 继承

继承实现了"**is-a**"的关系，允许我们创建更具体的子类来复用和扩展父类的功能。

- **实现方式**: 本项目中，继承最清晰的体现是在我们设计的**自定义异常体系**中。
    - 我们首先定义了一个通用的项目异常基类 PKMException，它继承自 Java 内置的 Exception 类。
    - 然后，我们创建了两个更具体的异常类 FileOperationException 和 SerializationException，它们都通过 extends PKMException 继承了 PKMException。

```
// 异常体系中的继承关系
public class PKMException extends Exception { /* ... */ }

public class FileOperationException extends PKMException { /* ... */ }

public class SerializationException extends PKMException { /* ... */ }
```

- **带来的优势**:

1. **代码复用**: FileOperationException 和 SerializationException 自动获得了作为"可检查异常"的所有特性，无需重复编写。
2. **逻辑分类与多态捕获**: 这种继承结构建立了一个清晰的异常"家族树"。它允许我们在 catch 块中进行多态捕获。例如，我们可以编写一个 catch (PKMException e) 块来捕获**所有**我们项目自定义的异常，而无需为 FileOperationException 和 SerializationException 分别

编写 catch 块，这让异常处理代码更简洁、更具扩展性。

- **早期设计中的体现**: 在项目初期（第二周），我们通过创建 TextNote extends Note 来演示继承。TextNote 自动获得了 Note 的所有通用属性和方法，并可以添加自己独特的 summary 属性，这是对代码复用和功能扩展的直接应用。

4.1.3 多态

多态的核心是"**一个接口，多种实现**"，它允许我们以一种通用的方式来处理不同类型的对象，极大地提升了代码的灵活性和可扩展性。

- **实现方式**: 本项目中，多态最核心、最优雅的应用体现在我们的**数据持久层设计**上。
    1. **定义通用接口**: 我们创建了一个 StorageService **接口**，它定义了所有数据存储方案都必须遵守的"契约"——即必须提供 save() 和 load() 方法。
    2. **提供具体实现**: 我们创建了 JsonStorageService **类**，它 implements StorageService，并提供了基于 JSON 文件的具体实现逻辑。
    3. **上层依赖于抽象**: 在上层的 NoteService 和 TagService 中，我们依赖的是 StorageService 这个**接口**，而不是 JsonStorageService 这个**具体实现**。

```
// 在 NoteService.java 中
public class NoteService {
    // 依赖于抽象的接口，而非具体的实现
    private final StorageService storageService;

    public NoteService(StorageService storageService) {
        this.storageService = storageService;
    }

    public List<Note> getAllNotes() {
        // 调用接口的方法，不关心具体实现是 JSON 还是数据库
        return storageService.load();
    }
}
```

- **带来的优势**: 这就是多态的威力所在。在 CommandParser 的构造函数中，我们创建了一个 JsonStorageService 的实例并注入给了 NoteService。NoteService 将它当作一个通用的 StorageService 来使用。

如果未来我们需要将存储方式更换为数据库，我们只需要创建一个新的 DatabaseStorageService 类也去实现 StorageService 接口。然后，在 CommandParser 中，仅仅需要**修改一行代码**：

```
StorageService storageService = new DatabaseStorageService();
```

整个 NoteService 和 TagService 的代码**一行都不需要改动**！它们能够无缝地处理这个新的、不同类型的 StorageService 实现。这就是多态带来的终极灵活性和"对修改关闭，对扩

展开放"的开闭原则的体现。

4.2 集合框架

[描述系统中怎样使用集合框架]

本系统广泛并深入地应用了 Java 集合框架来管理内存中的数据。根据不同场景下的数据特性和需求，我们策略性地选择了 List, Set, 和 Map 等核心接口及其实现类。

4.2.1 List：有序、可重复的集合

List 接口代表了一个有序的元素序列，允许元素重复。它在我们项目中的应用最为广泛。

- **核心应用场景**: **在 Note 实体中存储标签**
  - **代码**: private List<String> tags = new ArrayList<>();
  - **选择理由**:
    1. **遵循需求**: 为了与实验指导书的要求保持一致，我们最终采用了 List<String> 来维护一篇笔记的标签。
    2. **有序性**: 尽管在标签管理的场景下"顺序"意义不大，但 List 提供了基于索引的快速访问能力。
  - **实现细节**: 我们使用了 ArrayList 作为其具体实现，因为它在添加和遍历操作上具有良好的性能。为了弥补 List 允许重复的"缺点"，我们在 Note 类的 addTag(String tagName) 方法中**手动增加了重复性检查** (!this.tags.contains(tag))，以保证业务逻辑的正确性。
- **其他应用场景**:
  - **方法返回值**: 几乎所有的 Service 方法（如 noteService.getAllNotes(), noteService.searchNotesByKeyword()）都返回 List<Note>。这是因为搜索结果或全量数据通常是需要**按一定顺序**（如创建时间）展示给用户的，List 完美地满足了这一需求。
  - **测试数据准备**: 在所有的单元测试类（如 NoteServiceTest）中，我们都使用 List<Note> 来创建和管理测试数据集，便于构造各种测试场景。

4.2.2 Set：无序、唯一的集合

Set 接口代表了一个不包含重复元素的集合。它在我们项目的设计演进和最终实现中都扮演了重要角色。

- **核心应用场景**: **获取所有不重复的标签**
  - **代码**: 在我们第三周实现的 TagService（内存版）的 getAllTags() 方法中，我们通过 return tagMap.keySet(); 返回了一个 Set<String>。
  - **选择理由**: Map 的 keySet() 方法天然返回一个 Set，这完美地契合了"获取所有**唯一**标签名"的业务需求。Set 自动保证了结果中不会出现重复的标签，无需我们手动去重。
- **设计演进中的应用**:

- 在项目初期（第二周），我们曾将 Note 类中的标签集合设计为 private Set<Tag> tags;。当时选择 Set 的核心理由就是利用其**元素唯一性**的特性，来自动防止为一篇笔记添加重复的 Tag 对象。为了让 Set 能正确识别"重复"，我们还特意重写了 Tag 类的 equals() 和 hashCode() 方法。这个设计决策虽然在后期被重构，但充分体现了我们对不同集合特性的深刻理解和应用能力。

4.2.3 4 Map：键值对映射的集合

Map 接口用于存储键值对（Key-Value）数据，提供了通过 Key 快速查找 Value 的能力。它在我们系统的业务逻辑中扮演了"索引"和"统计"的核心角色。

- **核心应用场景**: **在 TagService (内存版) 中构建标签索引**
  - **代码**: private Map<String, List<Note>> tagMap = new HashMap<>();
  - **选择理由**:
    1. **快速查找**: 这个 Map 构建了一个**"倒排索引"**。它的 Key 是标签名（String），Value 是包含了该标签的所有 Note 对象的列表（List<Note>）。当我们需要实现 findNotesByTag(String tag) 功能时，我们不再需要遍历所有的笔记，而是可以直接通过 tagMap.get(tag)，以 O(1) 的时间复杂度快速定位到所有相关的笔记，**极大地提升了查询性能**。
    2. **动态构建**：我们使用了 HashMap 作为其具体实现，并在 indexNote 方法中巧妙地利用了 Java 8 的 computeIfAbsent 方法来高效地构建这个索引。
- **其他应用场景**:
  - **标签统计**:
    TagService 的 getTagStatistics() 和 AdvancedSearchService 的 generateTagCloud() 方法都返回 Map<String, Integer> 或 Map<String, Long>。在这里，Map 被用作一个**计数器**，Key 是标签名，Value 是该标签出现的次数。这同样是 Map 在数据聚合与统计场景下的经典应用。

通过对 List, Set, Map 的合理运用，我们不仅高效地管理了程序的数据，还通过构建索引等方式优化了核心功能的性能，充分展现了对 Java 集合框架的熟练掌握。

4.3 异常处理

[描述系统中如何使用异常]

健壮的异常处理是构建可靠软件的基石。本项目在第四周的设计中，引入了一套自定义的、层次清晰的**检查时异常体系**，旨在将底层的技术错误转换为带有丰富业务上下文信息的、更易于上层处理的业务异常。

4.3.1 自定义 PKMException 异常体系

为了避免直接向上层抛出模糊的底层异常（如 IOException），我们设计了一套继承自 java.lang.Exception 的自定义异常体系：

- **PKMException (基类)**: 这是我们项目中所有自定义业务异常的顶层父类。通过让所有异常都继承自它，我们未来可以在代码中通过 catch (PKMException e) 来捕获所有源于我们自己系统的、可预见的业务问题。
- **FileOperationException (具体异常)**: 该异常专门用于**文件 I/O 操作失败**的场景。它的设计核心是提供尽可能丰富的错误上下文。
- **SerializationException (具体异常)**: 该异常专门用于**对象序列化或反序列化失败**的场景，将 IOException 或 ClassNotFoundException 等底层错误与业务场景关联起来。

类图关系:

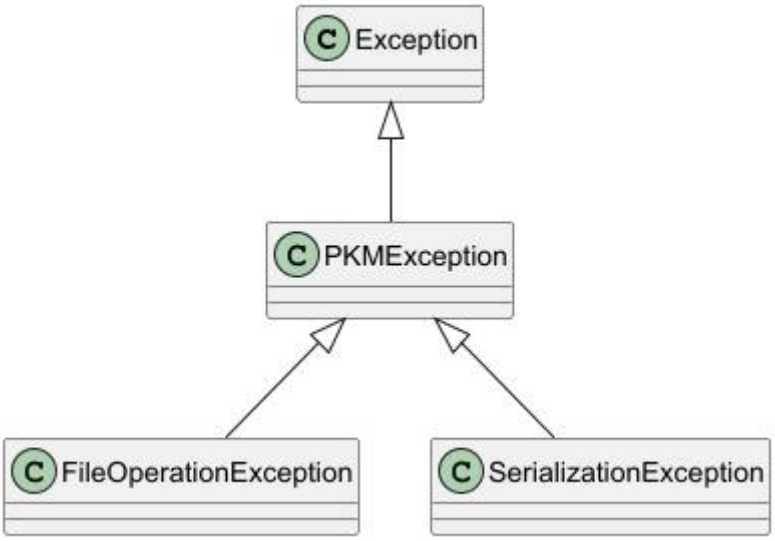

图 4- 1 异常处理类图

4.3.2 通过“异常链”提供丰富的上下文信息

我们的异常体系设计的核心亮点在于**异常链（Exception Chaining）**的有效运用，这完美地满足了“信息详细，支持异常链”的要求。

- **实现方式**: 以 FileOperationException 为例，它的构造函数接收三个参数：operation (正在做什么)、filePath (在哪个文件上做) 和 cause (最根本的原因是什么)。

FileOperationException.java 核心代码片段:

```
public class FileOperationException extends PKMException {
    public FileOperationException(String operation, String filePath, Throwable cause) {
        // 1. 将业务上下文信息格式化成一段易于阅读的 message
        super(
            String.format("文件操作失败: %s [文件: %s]", operation, filePath),
            // 2. 将原始的、底层的 cause 异常对象，“链接”到当前异常上
            cause
        );
    }
}
```

- **工作流程**: 当 JsonStorageService 在保存文件时捕获到一个 IOException，它不会直接把这个 IOException 扔给上层。相反，它会创建一个 FileOperationException，将当前的操作信息（如“保存笔记”）和这个 IOException 一起“包裹”进去，然后再向上抛出。
- **带来的优势**: 上层调用者（如 Application.java）捕获到 FileOperationException 后，可以同时获得两个层面的信息：
    1. **业务层面 (发生了什么)**: 通过 e.getMessage() 可以得到 "文件操作失败: 保存笔记 [文件: notes.json]" 这样的友好信息。
    2. **技术层面 (为什么发生)**: 通过 e.getCause() 可以获取到那个原始的 IOException 对象，并查看其完整的堆栈跟踪（Stack Trace），从而精确定位到是“磁盘已满”还是“权限不足”等根本原因。

五、功能演示

5.1 支持命令

个人知识管理系统（PKM-CLI）提供了一套丰富的命令行指令，用于对笔记进行完整的生命周期管理，包括创建、读取、更新、删除（CRUD）以及高级的搜索和导出功能。
所有命令均在程序的交互式提示符 pkm> 后输入。
支持的命令列表

表格 1 支持的命令

命令 (Command)	参数 (Parameters)	描述 (Description)	示例 (Example)
new	"<标题>" "<内容>"	创建一篇新的笔记。标题和内容必须用双引号包围。	pkm> new "Java 学习笔记" "今天学习了分层架构"
list	[--tag <标签名>]	列出所有笔记的摘要。可选的 --tag 参数可以根据标签进行过滤。	pkm> list pkm> list --tag java
view	<短 ID 或 完整 ID>	查看一篇指定 ID 的笔记的详细信息。	pkm> view 1
edit	<短 ID 或 完整 ID> "<新内容>"	编辑一篇已存在笔记的内容。新内容必须用双引号包围。	pkm> edit 1 "更新后的内容..."
delete	<短 ID 或 完整 ID>	删除一篇指定的笔记。	pkm> delete 2
tag	<短 ID 或 完整 ID> <标签名>	为指定的笔记添加一个标签。	pkm> tag 1 programming
untag	<短 ID 或 完整 ID> <标签名>	从指定的笔记中移除一个标签。	pkm> untag 1 old-tag
search	"<关键词>"	搜索标题或内容中包含指定关键词的笔记。关键词必须用双引号包围。	pkm> search "设计模式"

export	<短 ID 或 完整 ID> <格式> <路径>	将单篇指定的笔记导出为特定格式的文件。	pkm> export 1 text note.txt
export-all	<格式> <路径>	将所有笔记一次性导出到指定文件。	pkm> export-all text backup.txt
help	(无)	显示此帮助信息，列出所有可用命令及其用法。	pkm> help
exit	(无)	退出个人知识管理系统程序。	pkm> exit

5.2 交互示例截图

5.2.1 初识与创建 (Help & New)

- **动作**: 了解系统功能，并创建两篇新笔记。
- **命令**:

```
pkm> help
pkm> new "Java 学习" "今天学习了分层架构和依赖注入。"
pkm> new "Python 探索" "Python 的列表推导式非常优雅。"
```

```
D:\app\java\jdk\jdk-17\bin\java.exe "-javaagent:D:\app\java\IDEA\I
> 欢迎使用个人知识管理系统（CLI版）
> 输入 help 查看可用命令
pkm> help
--- 可用命令 ---
  new "<标题>" "<内容>"    - 创建一篇新笔记
  list                      - 列出所有笔记
  view <笔记ID>             - 查看笔记详情
  edit <笔记ID>'<新内容>'   - 编辑一篇笔记的内容
  delete <笔记ID>           - 删除一篇笔记
  tag <笔记ID> <标签名>     - 为笔记添加标签
  untag <笔记ID> <标签名>   - 为笔记移除标签
  search <关键词>           - 搜索标题或内容包含关键词的笔记
  export <笔记ID> <格式> <路径> - 导出单篇笔记
  export-all <格式> <路径> - 导出所有笔记
  exit                      - 退出程序
  help                      - 显示此帮助信息
----------------
pkm> new "Java 学习" "今天学习了分层架构和依赖注入。"
✅ 笔记创建成功!
ID: 08351e9d-19fe-49a0-892e-ffeabd92f36c
标题: Java 学习
pkm> new "Python 探索" "Python的列表推导式非常优雅。"
✅ 笔记创建成功!
ID: 3bb32834-af5d-47f7-91b9-99dee620a1f2
标题: Python 探索
pkm>
```

图 5- 1 Help & New

5.2.2 查看与组织 (List & Tag)

- **动作**: 查看刚刚创建的笔记，并为它们添加标签。
- **命令**:

```
pkm> list
pkm> tag 1 Java
pkm> tag 1 OOP
pkm> tag 2 Python
pkm> list
```

```
pkm> list
--- 所有笔记列表 ---
-----------------------------------
[1] 这是标题 (2025-11-02) [标签]
[2] 标题2 (2025-11-03) []
[3] Java 学习 (2025-11-03) []
[4] Python 探索 (2025-11-03) []
-----------------------------------
pkm> tag 3 Java
✅ 标签 'Java' 已成功添加到笔记 'Java 学习'。
   当前标签: Java
pkm> tag 3 OOP
ℹ️  首次操作ID，正在刷新笔记列表...
--- 所有笔记列表 ---
-----------------------------------
[1] 这是标题 (2025-11-02) [标签]
[2] 标题2 (2025-11-03) []
[3] Java 学习 (2025-11-03) [Java]
[4] Python 探索 (2025-11-03) []
-----------------------------------
✅ 标签 'OOP' 已成功添加到笔记 'Java 学习'。
   当前标签: Java, OOP
pkm> tag 4 Python
ℹ️  首次操作ID，正在刷新笔记列表...
--- 所有笔记列表 ---
-----------------------------------
[1] 这是标题 (2025-11-02) [标签]
[2] 标题2 (2025-11-03) []
[3] Java 学习 (2025-11-03) [Java, OOP]
[4] Python 探索 (2025-11-03) []
-----------------------------------
✅ 标签 'Python' 已成功添加到笔记 'Python 探索'。
   当前标签: Python
pkm> list
--- 所有笔记列表 ---
-----------------------------------
[1] 这是标题 (2025-11-02) [标签]
[2] 标题2 (2025-11-03) []
[3] Java 学习 (2025-11-03) [Java, OOP]
[4] Python 探索 (2025-11-03) [Python]
-----------------------------------
pkm>
```

图 5- 2 List & Tag

5.2.3 深入与过滤 (View & List --tag)

- **动作**: 详细查看一篇笔记，并尝试按标签进行过滤。
- **命令**:

codeCode

```
pkm> view 1
pkm> list --tag Java
pkm> list --tag Python
```

```
pkm> view 3
--- 笔记详情 ---
ID       : 08351e9d-19fe-49a0-892e-ffeabd92f36c
标题     : Java 学习
标签     : Java, OOP
创建时间: 2025-11-03T17:25:43
更新时间: 2025-11-03T17:25:43
----------------
内容:
今天学习了分层架构和依赖注入。"
----------------
pkm> list --tag Java
--- 标签为 'Java' 的笔记列表 ---
-------------------------------------
[1] Java 学习 (2025-11-03) [Java, OOP]
-------------------------------------
pkm> list --tag Python
--- 标签为 'Python' 的笔记列表 ---
-------------------------------------
[1] Python 探索 (2025-11-03) [Python]
-------------------------------------
```

图 5- 3 View & List --tag

5.2.4 搜索与修改 (Search & Edit)

- **动作**: 尝试关键词搜索，并修改一篇笔记的内容。
- **命令**:

codeCode

```
pkm> search "架构"
pkm> edit 1 "补充：高内聚、低耦合是核心思想。"
pkm> view 1
```

```
pkm> search "架构"
--- 关键词为 '架构' 的搜索结果 ---
[08351e9d-19fe-49a0-892e-ffeabd92f36c] Java 学习 (2025-11-03) [Java, OOP]
--------------------

pkm> edit 3 "补充"
✅ 笔记 (ID: 08351e9d-19fe-49a0-892e-ffeabd92f36c) 内容已成功更新。
pkm> view 3
ℹ️ 首次操作，正在刷新笔记列表...
--- 所有笔记列表 ---
-----------------------------------
[1] 这是标题 (2025-11-02) [标签]
[2] 标题2 (2025-11-03) []
[3] Java 学习 (2025-11-03) [Java, OOP]
[4] Python 探索 (2025-11-03) [Python]
-----------------------------------
--- 笔记详情 ---
ID       : 08351e9d-19fe-49a0-892e-ffeabd92f36c
标题     : Java 学习
标签     : Java, OOP
创建时间: 2025-11-03T17:25:43
更新时间: 2025-11-03T17:34:57
----------------
内容:
补充
----------------
pkm>
```

图 5- 4 Search & Edit

5.2.5 数据的导出与备份 (Export & Export-all)

- **动作**: 演示如何将单篇笔记和所有笔记，以不同的格式导出为文件。
- **命令**:

```
pkm> list
pkm> export 3 text D:\exercise\my_single_note.txt
pkm> export-all text D:\exercise\all_my_notes_backup.txt
pkm> exit
```

```
pkm> export 3 text D:\exercise\my_single_note.txt
✅ 笔记 (ID: 08351e9d-19fe-49a0-892e-ffeabd92f36c) 已成功导出到: D:\exercise\my_single_note.txt
pkm> export-all text D:\exercise\all_my_notes_backup.txt
✅ 所有 3 篇笔记已成功导出到: D:\exercise\all_my_notes_backup.txt
```

图 5- 5 Export & Export-all

5.2.6 清理与收尾 (Untag & Delete & Exit)

- **动作**: 移除一个标签，删除一篇笔记，最后退出程序。
- **命令**:

codeCode

```
pkm> untag 1 OOP
pkm> view 1
pkm> delete 2
pkm> list
pkm> exit
```

```
pkm> untag 3 OOP
✅ 标签 'OOP' 已成功从笔记 'Java 学习' 移除。
pkm> view 3
ℹ️  首次操作，正在刷新笔记列表...
--- 所有笔记列表 ---
-----------------------------------
[1] 这是标题 (2025-11-02) [标签]
[2] 标题2 (2025-11-03) []
[3] Java 学习 (2025-11-03) [Java]
[4] Python 探索 (2025-11-03) [Python]
-----------------------------------
--- 笔记详情 ---
ID       : 08351e9d-19fe-49a0-892e-ffeabd92f36c
标题     : Java 学习
标签     : Java
创建时间: 2025-11-03T17:25:43
更新时间: 2025-11-03T17:34:57
----------------
内容:
补充
----------------
pkm> delete 4
✅ 笔记 (ID: 3bb32834-af5d-47f7-91b9-99dee620a1f2) 已被成功删除。
pkm> list
--- 所有笔记列表 ---
-----------------------------------
[1] 这是标题 (2025-11-02) [标签]
[2] 标题2 (2025-11-03) []
[3] Java 学习 (2025-11-03) [Java]
-----------------------------------
pkm> exit
👋 再见！

Process finished with exit code 0
```

图 5- 6 Untag & Delete & Exit

六、工程实践

6.1 版本控制

[描述如何使用版本控制]

本项目在整个开发生命周期中，严格遵循了基于 **Git** 的现代化版本控制实践，以确保代码历史的清晰、可追溯，并为未来的团队协作和维护打下坚实基础。

6.1.1 遵循 Git Flow 简化工作流

我们采纳了业界广泛应用的 **Git Flow** 的简化版本作为我们的分支管理策略。其核心流程如

下：

1. **main 分支**: 作为项目**唯一、稳定**的主分支。main 分支上的每一次提交都代表一个经过测试的、可发布的版本。
2. **feature/... 分支**: 每一个新功能的开发、每一个实验周期的任务，都在一个**独立的功能分支**上进行。这些分支都从 main 分支切出，并以 feature/ 作为统一的前缀，后接具体的功能描述（如 feature/persistence, feature/architecture-refactor）。
3. **独立开发**: 所有的开发和测试工作都在功能分支上进行，期间可以有多次独立的提交。这保证了 main 分支在开发过程中始终保持干净和稳定。
4. **合并回 main**: 当一个功能被完整实现并通过所有测试后，该功能分支将被**合并（Merge）**回 main 分支，标志着该功能的正式发布。合并后，功能分支的使命即告完成，将被删除以保持仓库的整洁。

这种工作流确保了开发过程的**高度隔离**和主分支的**绝对稳定**。

6.1.2 强调原子提交与约定式提交规范

为了让版本历史不仅仅是一串无序的记录，而是成为一份清晰可读的"项目开发日记"，我们严格遵循了以下两大提交原则：

1. **原子提交 (Atomic Commits)**:
   我们坚持**每一次 git commit 只包含一个独立的、完整的逻辑变更**。例如，"实现 NoteService 及其测试"是一次提交，"实现 NoteController 及其测试"是另一次独立的提交。这种做法带来了巨大的好处：
   - **便于代码审查 (Code Review)**: 每次审查的目标都非常集中。
   - **精确定位问题**: 如果某个功能引入了 bug，我们可以快速定位到引入该 bug 的具体提交。
   - **安全的回滚**: 可以安全地撤销某次提交，而不会影响到其他功能。
2. **约定式提交 (Conventional Commits)**:
   我们所有的提交信息都遵循了 type(scope): subject 的规范格式。这使得提交历史极具可读性，并且便于自动化工具进行分析。
   - **type**: 表明提交的性质，如 feat (新功能), fix (修复 bug), docs (文档), test (测试), refactor (重构)。
   - **scope**: (可选) 指明提交影响的范围，如 (cli), (service), (entity)。
   - **subject**: 用简洁的语言描述本次提交做了什么。

6.1.3 提交历史证据

以下是通过 git log --oneline --graph 命令生成的提交历史图谱，它清晰地展示了我们的分支创建、原子化提交以及最终合并的完整流程。

```
(base) PS D:\exercise\java\IdeaProjects\pkm-cli -6 构建优雅的命令行程序—架构设计与实现> git log --oneline --graph --all
* 0a774ec (HEAD -> feature/refactor, gitea/feature/refactor) feat: 命令行项目完成
* dde040e feat(cli): implement full application architecture and command parser
```

```
* 9d2e208 feat(controller): implement NoteController,TagContraller and add unit tests with
Mockito
* 6995ba7 feat(service): implement NoteService,TagService and unit tests
* 679fe8c feat(setup): create package structure and implement JsonStorageService
* 88337cb (origin/main, gitea/main, main) feat: integrate file persistence into application
flow
* 5694e8f feat: add text export functionality
:...skipping...
* 0a774ec (HEAD -> feature/refactor, gitea/feature/refactor) feat: 命令行项目完成
* dde040e feat(cli): implement full application architecture and command parser
* 9d2e208 feat(controller): implement NoteController,TagContraller and add unit tests with
Mockito
* 6995ba7 feat(service): implement NoteService,TagService and unit tests
* 679fe8c feat(setup): create package structure and implement JsonStorageService
* 88337cb (origin/main, gitea/main, main) feat: integrate file persistence into application
flow
* 5694e8f feat: add text export functionality
* cef51dc feat: implement NoteFileRepository with save, load and backup mechanism
* 0483911 feat: create exception classes and tests
* 64d5d95 feat: make Note entity serializable
* a1435b1 feat: implement AdvancedSearchService with Stream API
* 20a69ac feat: complete Task 2 - implement TagService and tests
* b169b6a feat: complete Task 1 - Enhance Note entity and add tests
*   0aff214 Merge branch 'main' of https://github.com/zrlucky/pkm-cli
|\
| * 59917c0 feat: add TextNote subclass and unit tests
* | 29965c2 feat: add TextNote subclass and unit tests
|/
* e301b33 feat: implement base Note entity with tags
* 2be9dee feat: implement Tag entity
* 271b43a docs: add initial UML class diagram
(END)
```

6.2 开发工具

[列举用到的开发工具包括 IDE、Maven、GIT 等]

本项目采用了业界主流的现代化 Java 开发工具链，以确保开发效率、代码质量和项目管理的规范性。所使用的核心工具与技术库如下：

- **集成开发环境 (IDE)**: **IntelliJ IDEA**

    - **作用**: 作为项目的主要开发平台，提供了强大的代码编辑、智能提示、重构和调试功能。其无缝集成的 Maven 和 Git 工具，以及对 JUnit 5 和 PlantUML 等插件的良好支持，极大地提升了开发和测试效率。我们利用其强大的调试器（Debugger）完成了对文件操作异常和备份恢复机制的实战验证。
- **项目构建与依赖管理**: **Apache Maven**
    - **作用**: 负责整个项目的构建生命周期和第三方库的依赖管理。通过 pom.xml 配置文件，我们统一管理了项目所需的 JUnit 5, Mockito, Jackson 等所有外部依赖，保证了项目环境的一致性和可复现性。
- **版本控制系统**: **Git**
    - **作用**: 用于对项目源代码进行全面的版本控制。我们利用 Git 实现了功能分支开发、原子化提交，并最终将成果合并与推送到远程仓库，完整地实践了现代化的软件开发工作流。
- **核心编程语言**: **Java (JDK 17)**
    - **作用**: 项目的基础编程语言。我们充分利用了其现代特性，如 Java 8 引入的 Stream API（用于高级查询）、Optional（用于优雅地处理可能为空的结果）和 LocalDateTime（用于精确的时间处理）。
- **单元测试框架**: **JUnit 5 (Jupiter)**
    - **作用**: 作为项目自动化测试的基石。我们使用 JUnit 5 编写了覆盖了从实体层到服务层再到控制器层的单元测试，通过 @Test, @DisplayName, @BeforeEach @AfterEach 等注解组织测试用例，并利用其丰富的断言库 (Assertions) 保证了代码的质量和行为的正确性。
- **模拟测试框架 (Mocking Framework)**: **Mockito**
    - **作用**: 这是我们进行**分层测试**的关键工具。在测试 Controller 层时，我们使用 Mockito 创建了 Service 层的"模拟对象"（Mock Object），从而实现了对 Controller 交互逻辑的**隔离测试**。这确保了单元测试的独立性和快速性，而无需依赖真实的业务逻辑或数据存储。
- **JSON 处理库**: **Jackson**
    - **作用**: 作为 JsonStorageService 的核心引擎，负责 Java 对象与 JSON 格式之间的**序列化**与**反序列化**。我们利用其强大的 ObjectMapper，并特别配置了 JavaTimeModule，以实现对 LocalDateTime 等复杂数据类型的无缝处理。
- **图表绘制工具**: **PlantUML**
    - **作用**: 用于"代码即图表"的架构设计。我们使用 PlantUML 编写了项目的整体三层架构图和核心类图。这种方式使得设计文档可以和源代码一样被 Git 版本控制，保证了设计与实现的同步演进。

七、收获与反思

7.1 技术收获

通过本次个人知识管理系统（PKM-CLI）的完整开发实践，我不仅巩固了 Java 编程的基础，更在软件架构、测试方法和工程化实践等多个方面获得了宝贵的收获。

- **分层架构与依赖注入 (DI)**
  - 我深刻理解了将应用划分为**表现层、业务逻辑层和数据持久层**的必要性和巨大优势。通过亲手将一个“混乱的 main 方法”重构为清晰的三层架构，我体会到这种设计如何使代码职责分明、易于理解和维护。
  - 我掌握了**依赖注入 (DI)** 的核心思想，并在 CommandParser 的构造函数中实践了手动“装配”。我认识到，由外部容器（即 CommandParser）负责创建和组装依赖关系，而不是让类自己去 new 它所依赖的对象，是实现“**高内聚、低耦合**”的关键。特别是通过**面向接口编程**（Service 依赖 StorageService 接口），我理解了如何让上层业务逻辑与底层具体实现技术（JSON 文件）彻底解耦。
- **全面的单元测试与模拟技术 (Mockito)**
  - 本次实验让我真正将**测试驱动开发 (TDD)** 的理念贯穿始终。我学会了使用 **JUnit 5** 框架为每一个独立的逻辑单元（从实体类到服务类）编写单元测试，并特别注重对**边界情况**的覆盖，有效保证了代码质量。
  - 最大的技术突破是学会了使用 **Mockito** 框架。通过创建“模拟对象”（Mock Object），我成功地对 Controller 层进行了**隔离测试**。这让我理解了如何在不依赖真实后端服务（如 NoteService）的情况下，独立地、快速地验证 UI 逻辑（如输出格式、异常捕获）的正确性，这是专业分层测试中的核心技能。
- **健壮的异常处理体系**
  - 我学习并设计了一套继承自 Exception 的**自定义异常体系**。我明白了不能简单地将底层 IOException 抛给上层，而应该将其“包裹”在带有丰富业务上下文信息（如操作类型、文件路径）的自定义异常（如 FileOperationException）中。
  - 我掌握了**异常链 (Exception Chaining)** 的实现方式和价值，它使得错误排查既能看到上层的业务错误，又能追溯到底层的技术根源。同时，在 Application 的顶层 try-catch 块中，我学会了如何向用户展示友好的错误提示，同时保证程序的**容错能力**，不会轻易崩溃。
- **Java 8+ 现代化特性应用**
  - 我熟练运用了 **Stream API** 来实现集合的高效数据处理，例如在 NoteService 中使用 filter 和 collect 实现搜索和过滤，在 AdvancedSearchService 中使用 flatMap 和 groupingBy 实现复杂的标签云统计。这让我体会到函数式编程风格的简洁与强大。
  - 我也在项目中广泛使用了 Optional 来优雅地处理可能为空的查询结果（如 findNoteById），有效避免了 NullPointerException 的风险。
- **专业的工程化实践**
  - 通过本次实验，我将 **Git** 的使用从简单的代码保存，提升到了遵循 **Git Flow** 规范、进行**原子提交**和编写**约定式提交信息**的工程化水平。这让我认识到版本控制在记录项目演进、支持团队协作中的重要作用。

7.2 实践体会

通过本次实验，我最深刻的体会是完成了从“写功能”到“做工程”**的思维转变。在项目初期，我的关注点是如何快速实现 new, list 等具体功能；但随着项目的深入，特别是第六周**

的架构重构，我开始从系统整体的健康度、可维护性和可扩展性角度去思考问题，并真切地体会到了“**高内聚、低耦合**”这一软件设计黄金法则带来的巨大好处。

- **从“混乱的 Main”到“有序的分层”**
  在项目初期，很容易陷入将所有逻辑都堆砌在 main 方法中的陷阱。第四周我们实现的 Application.java 虽然能工作，但加载数据、重建索引、模拟操作、保存数据等所有逻辑都耦合在一起，难以阅读和修改。第六周的重构让我明白了，一个健康的系统应该像一个组织良好的公司：
    - Controller 是“前台”，只负责接待和传话。
    - Service 是“技术部门”，只负责处理核心业务。
    - Repository 是“仓库”，只负责数据的进出。
      这种职责分明的设计，使得当我需要修改某个功能时，我能立刻定位到对应的“部门”（包）和“员工”（类），而不用在一个几百行的 main 方法中大海捞针。

- **切身体会“高内聚、低耦合”的威力**
  在搭建完三层架构后，我再去新增 export-all 这个命令时，其过程的顺滑流畅令我印象深刻。我惊讶地发现：

1. 我**不需要**修改 NoteService，因为它已经提供了 getAllNotes() 的能力。
2. 我**不需要**修改 ExportService，因为它已经具备了导出 List<Note> 的能力。
3. 我**甚至不需要**为它们编写新的单元测试，因为它们的功能已经被充分验证。

我需要做的，仅仅是在 NoteController 中增加一个新的 exportAllNotes 方法来**“编排”**这两个已有的服务，然后在 CommandParser 中为它添加入口。**几乎没有改动任何旧的、稳定的代码，只是在顶层做了一些“胶水”工作**。

这让我深刻体会到，一个“低耦合”的系统，其各个组件就像乐高积木一样，可以被灵活地复用和组合来创造新的功能，极大地提升了开发效率和系统的稳定性。

- **“面向接口编程”带来的自由**
  Service 层依赖于 StorageService 接口而非具体的 JsonStorageService 实现，这个设计在初期看似多此一举，但在我思考“改进方向”时，它的威力便显现出来。如果我想把持久化方案从 JSON 更换为数据库，我确信我只需要创建一个新的 DatabaseStorageService 实现 StorageService 接口，然后在 CommandParser 中替换一行 new 语句即可。这种“可插拔”的设计，让我对系统的未来充满了信心，也让我明白了为什么“面向接口编程”是现代软件设计的基石。

总而言之，本次实验让我认识到，**软件架构并非空中楼阁，而是决定一个项目能否健康成长、从容应对变化的“地基”**。一个良好的架构，能让今天的代码在明天依然易于理解、易于修改、易于扩展。这种从“功能实现者”到“系统设计者”的视角转变，是我最大的收获。

7.3 改进方向

尽管当前版本的个人知识管理系统（PKM-CLI）已经实现了所有核心功能并拥有了清晰

的架构，但在实践和反思中，我们依然识别出多个可以进一步优化和提升的方向。这些改进将使系统在**用户体验、可扩展性**和**健壮性**方面迈上新的台阶。

1. 完善并固化"短 ID"映射机制

- **现状**: 我们已经在 CommandParser 中通过一个临时的内存列表 (lastListedNotes)，**初步实现**了在单次会话中将 list 命令的行号映射为短 ID 的功能，极大地提升了用户体验。
- **不足**: 这种缓存是**易失的**。每次 delete 或 edit 操作后，为了保证数据一致性，我们都清空了缓存，这意味着用户在进行修改操作后，必须**再次执行 list** 才能使用新的短 ID。
- **改进方向**: 可以设计一个更持久的映射方案。例如，在 JsonStorageService 保存数据时，除了保存 notes.json，还可以额外维护一个 id_map.json 文件，里面存储着短 ID（如 1, 2, 3）与真实 UUID 之间的**固定映射关系**。这样，即使程序重启，笔记 [1] 永远都对应着同一篇笔记，直到它被删除。这将提供一种更稳定、更可预测的用户体验。

2. 引入"命令模式"替代 switch 语句，提高扩展性

- **现状与问题**:
  CommandParser 目前使用一个庞大的 switch 语句来分发命令。虽然直观，但这种做法违反了**"开闭原则"。每当我们需要增加一个新的命令（比如未来增加 import 命令），都**必须修改** CommandParser 类的 dispatchCommand 方法，增加了代码的维护成本和引入错误的风险。
- **改进方案**:
  采用**命令模式 (Command Pattern)** 对 CommandParser 进行重构。我们可以定义一个 Command 接口，其中包含一个 execute(String[] args) 方法。然后，为每一个命令（NewCommand, ListCommand 等）创建一个实现了该接口的具体类。在 CommandParser 的构造函数中，我们创建一个 Map<String, Command>，将命令字符串（如 "new"）与对应的命令对象实例映射起来。这样，dispatchCommand 方法就可以被极大地简化：从 Map 中获取命令对象，然后调用其 execute 方法即可。
- **预期收益**:
  - **高度可扩展**: 增加新命令时，只需创建一个新的 Command 实现类并注册到 Map 中，**完全无需修改** CommandParser 的核心分发逻辑。
  - **职责更单一**: 每个命令的参数解析和调用逻辑都被封装在各自的 Command 类中，使得 CommandParser 的职责更纯粹。

3. 增加数据库持久化方案，提升数据管理能力

- **现状与问题**:
  当前系统使用 JsonStorageService 将所有笔记序列化到一个**单一的 JSON 文件**中。这种方案在数据量较少时工作良好，但随着笔记数量的增多，性能会急剧下降，因为每次修改（如删除一篇笔记）都需要**重写整个文件**。此外，它也不支持复杂的查询和事务。
- **改进方案**:
  利用我们已经设计好的 StorageService 接口，创建一个全新的实现类，例如 DatabaseStorageService。这个新类将使用 **JDBC** 或者 **JPA (如 Hibernate)** 技术，将笔记数据持久化到一个关系型数据库（如 H2, SQLite, MySQL）中。Note 实体可

以被映射为数据库中的一张表。

- **预期收益**:
  - **高性能**: 对单篇笔记的增删改查操作，将只涉及数据库中的特定行，性能远超重写整个文件。
  - **强大的查询能力**: 可以利用 SQL 语言实现比 Stream API 更强大、更高效的复杂查询。
  - **数据一致性与事务支持**: 数据库提供了 ACID 事务保证，能更好地确保数据操作的一致性和安全性。
  - **低耦合的验证**: 这将完美验证我们"面向接口编程"的设计优势——更换底层存储方案，上层业务逻辑代码**无需任何改动**。

4. 完善参数解析，支持更复杂的输入

- **现状与问题**:
  我们当前对带引号参数的解析（如 new "title" "content"）依赖于简单的 split("\""），这种方法比较脆弱，无法处理参数内部包含引号等复杂情况。
- **改进方案**:
  可以引入一个简单的状态机，或者使用更专业的命令行解析库（如 **JCommander** 或 **Picocli**）。这些库提供了强大的功能，可以轻松定义命令、选项和参数，并自动处理带引号的字符串、参数校验、类型转换，甚至能自动生成帮助信息。
- **预期收益**: 使命令行程序的参数处理能力达到专业工具的水平，变得更加健壮和可靠。

5. 增强 edit 命令，支持内容追加

- **现状与问题**: 当前的 edit 命令会用新内容完全覆盖旧内容。但在实际使用中，用户更常见的需求可能是在原有笔记的基础上**追加**新的想法或补充。
- **改进方向**: 可以通过两种方式进行改进。**方案一**是增加一个全新的 append <ID> "<内容>" 命令，专门负责追加内容，职责清晰。**方案二**是为 edit 命令增加一个 --append 选项（如 edit --append <ID> "<追加内容>"），使其支持两种模式，更加灵活。这两种方案都需要对 Service, Controller, CommandParser 进行相应的扩展，以提供更贴近用户实际需求的编辑体验。

八、附录

8.1 项目结构

```
pkm-cli/
├── .gitignore          # Git 忽略文件配置
├── pom.xml             # Maven 项目核心配置文件，定义依赖和构建规则
├── notes.json          # (运行时生成) 持久化的笔记数据文件
├── docs/               #  存放设计文档
```

```
│   └── architecture-design.puml
│   └── pkm-design.puml
│   └── pkm-entity.puml
│   └── 分层架构图.puml
│   └── 异常处理类图.puml
│   └── 异常处理类图.puml
└── src/
    ├── main/
    │   └── java/
    │       └── com/
    │           └── ZhangRuo/
    │               └── pkm/
    │                   ├── app/
    │                   │   └── App.java            # 程序唯一主入口
    │                   ├── cli/
    │                   │   └── CommandParser.java  # 命令解析与分发核心
    │                   ├── controller/
    │                   │   ├── NoteController.java # 笔记相关的交互逻辑
    │                   │   └── TagController.java  # 标签相关的交互逻辑
    │                   ├── entity/
    │                   │   ├── Note.java           # 笔记核心实体
    │                   │   └── Tag.java            # 标签实体 (为未来扩展保留)
    │                   ├── enums/
    │                   │   └── ExportFormat.java   # 导出格式枚举
    │                   ├── exception/
    │                   │   ├── FileOperationException.java
    │                   │   ├── PKMException.java   # 异常体系基类
    │                   │   └── SerializationException.java
    │                   ├── repository/
    │                   │   ├── JsonStorageService.java # JSON 持久化实现
    │                   │   └── StorageService.java     # 持久化接口(契约)
    │                   └── service/
    │                       ├── ExportService.java      # 导出业务逻辑
    │                       ├── NoteService.java        # 笔记核心业务逻辑
    │                       └── TagService.java         # 标签核心业务逻辑
    └── test/
        └── java/
            └── com/
                └── ZhangRuo/
                    └── pkm/
                        ├── controller/
                        │   ├── NoteControllerTest.java
                        │   └── TagControllerTest.java
                        ├── entity/
```

```
│   └── NoteTest.java
├── exception/
│   ├── FileOperationExceptionTest.java
│   └── SerializationExceptionTest.java
├── repository/
│   └── JsonStorageServiceTest.java
└── service/
    ├── ExportServiceTest.java
    ├── NoteServiceTest.java
    └── TagServiceTest.java
```

8.2 核心实体

8.2.1 Note.java

```java
package com.ZhangRuo.pkm.entity;

import java.io.Serializable;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

/**
 * 代表一个笔记的核心实体基类。4 周支持序列化
 */
public class Note implements Serializable {//实现 Serializable 接口
    /**
     * 序列化版本 UID，用于版本控制
     */
    private static final long serialVersionUID = 1L;  //添加 seriaVersionUID

    private String id;
    private String title;
    private String content;

    /**
     * 内部标签集合，根据指导书要求，使用 List<String> 存储标签名。
     */
```

```
    private List<String> tags = new ArrayList<>();

    //LocalDateTime 自身可序列化
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;

    /*
    * 为 Jackson 反序列化提供的无参数构造方法
    * 框架（如 Jackson）需要这个构造方法来创建对象的空实例，然后再填充属性
    * Jackson 默认需要调用 Note 类的无参数构造方法（也叫默认构造方法），也就是 public Note() {}
    * */
    public Note() {
        //构造方法体可以是空的
    }


    public Note(String title, String content) {
        this.title = title;
        this.content = content;
        this.createdAt = LocalDateTime.now();
        this.updatedAt = LocalDateTime.now();
    }


    /**
     * 为笔记添加一个标签（通过标签名）。
     * 添加前会检查标签是否为 null、空白或已存在。
     *
     * @param tagName 要添加的标签名。
     */
    public void addTag(String tagName) {
        if (tagName != null && !tagName.isBlank() && !this.tags.contains(tagName)) {
            this.tags.add(tagName);
        }
    }

    /**
     * 从笔记中移除一个标签（通过标签名）。
     *
     * @param tagName 要移除的标签名。
     */
    public void removeTag(String tagName) {
        this.tags.remove(tagName);
    }
```

```
/**
 * 检查笔记是否包含指定的标签（通过标签名）。
 *
 * @param tagName 要检查的标签名。
 * @return 如果包含则返回 true，否则返回 false。
 */
public boolean hasTag(String tagName) {
    return this.tags.contains(tagName);
}

/**
 * 获取该笔记的所有标签名列表。
 *
 * @return 一个包含所有标签名的列表。
 */
public List<String> getTags() {
    return this.tags;
}

// --- 兼容之前设计的、更健壮的方法（参数为 Tag 对象） ---

/**
 * [兼容方法] 为笔记添加一个标签（通过 Tag 对象）。
 * 内部会提取 Tag 对象的名称并添加到列表中。
 *
 * @param tagObject 要添加的 Tag 对象。
 */
public void addTag(Tag tagObject) {
    if (tagObject != null) {
        // 调用上面已经写好的、更健壮的 addTag(String) 方法
        this.addTag(tagObject.getName());
    }
}

/**
 * [兼容方法] 从笔记中移除一个标签（通过 Tag 对象）。
 *
 * @param tagObject 要移除的 Tag 对象。
 */
public void removeTag(Tag tagObject) {
    if (tagObject != null) {
        this.removeTag(tagObject.getName());
    }
```

```
    }

    /**
     * [兼容方法] 将标签名列表转换为 Tag 对象列表。
     * 便于未来需要使用 Tag 对象集合的场景。
     *
     * @return 一个包含所有标签的 Tag 对象列表。
     */
    public List<Tag> getTagObjects() {
        return this.tags.stream()
                .map(Tag::new) // 对每个 tagName 字符串，调用 Tag 的构造方法 new Tag(tagName)
                .collect(Collectors.toList());
    }

    // --- 其他 Getters and Setters ---

    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; this.updatedAt =
LocalDateTime.now(); }
    public String getContent() { return content; }
    public void setContent(String content) { this.content = content; this.updatedAt =
LocalDateTime.now(); }
    public void setTags(List<String> tags) { this.tags = tags; }
    public LocalDateTime getCreatedAt() { return createdAt; }
    public void setCreatedAt(LocalDateTime createdAt) { this.createdAt = createdAt; }
    public LocalDateTime getUpdatedAt() { return updatedAt; }
    public void setUpdatedAt(LocalDateTime updatedAt) { this.updatedAt = updatedAt; }

    // ... equals, hashCode, toString ...
    @Override
    public boolean equals(Object o) { if (this == o) return true; if (o == null || getClass() !=
o.getClass()) return false; Note note = (Note) o; return Objects.equals(id, note.id); }
    @Override
    public int hashCode() { return Objects.hash(id); }
    @Override
    public String toString() { return "Note{" + "id='" + id + '\'' + ", title='" + title +
'\'' + ", tags=" + tags + '}'; }
}
```

8.2.2 Tag.java

```java
package com.ZhangRuo.pkm.entity;

import java.util.Objects;

/**
 * 代表一个标签的实体类。
 * Tag 对象通过其 'name' 属性来定义，并通过 'id' 进行唯一标识。
 */
public class Tag {

    /**
     * 标签的唯一标识符，通常由数据库生成。
     */
    private Long id;

    /**
     * 标签的名称，例如 "Java", "OOP" 等。
     */
    private String name;

/*
* 为 Jackson 反序列化提供的无参数构造方法。
* */
    public Tag(){
        //构造方法可以是空的
    }

    /**
     * Tag 类的构造方法。
     * 创建一个标签时，必须为其提供一个名称。
     *
     * @param name 标签的名称，不应为 null。
     */
    public Tag(String name) {
        this.name = name;
    }

    // --- Getters and Setters ---

    public Long getId() {
        return id;
    }
```

```
    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    // --- 重写 equals, hashCode, toString (专业实践) ---

    /**
     * 比较两个 Tag 对象是否相等。
     * 如果两个 Tag 的 id 相同，则认为它们是相等的。
     * 这对于在 Set 集合中正确处理 Tag 对象至关重要。
     *
     * @param o 要比较的对象。
     * @return 如果对象相等则返回 true，否则返回 false。
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Tag tag = (Tag) o;
        return Objects.equals(id, tag.id); // 关键：仅基于 id 判断相等性
    }

    /**
     * 根据对象的 id 生成哈希码。
     * 与 equals() 方法保持一致，是使用 HashSet 的必要条件。
     *
     * @return 对象的哈希码。
     */
    @Override
    public int hashCode() {
        return Objects.hash(id); // 关键：仅基于 id 生成哈希码
    }

    /**
     * 返回 Tag 对象的字符串表示形式，方便调试。
     *
```

```
     * @return 对象的字符串表示。
     */
    @Override
    public String toString() {
        return "Tag{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }
}
```

8.3 文件持久化

8.3.1 StorageService.java

```
package com.ZhangRuo.pkm.repository;

import com.ZhangRuo.pkm.entity.Note;
import java.util.List;

/*
*定义了数据存储服务的统一接口（契约）
* 任何具体的存储实现（如 JSON，数据库）都必须实现此接口
* */

public interface StorageService {
    /*
    * 保存笔记列表      * @param notes 要保存的笔记列表      * */      void save(List<Note> notes);

    /*
    * 加载所有笔记      * @return 一个包含所有笔记的列表      * */      List<Note> load();

}
```

8.3.2 JsonStorageService.java

```
package com.ZhangRuo.pkm.repository;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import com.fasterxml.jackson.datatype.jsr310.JavaTimeModule;
import com.ZhangRuo.pkm.entity.Note;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/*
* 使用 JSON 文件实现 StorageService 接口
*负责将笔记对象序列化为 JSON 格式并存入文件，以及从文件中反序列化
*
* StorageService 接口的具体实现者，它负责读写 notes.json 文件
* */

public class JsonStorageService implements StorageService {

    private final String filePath;
    private final ObjectMapper objectMapper; //Jackon 核心对象

    /*
    * 默认构造方法，使用"notes.json"作为文件名     * */
    public JsonStorageService(){
        this("notes.json");
    }

    /*
    * 可指定文件路径的构造方法     * @param filepath JSON 文件的路径     * */
    public JsonStorageService(String filePath){
        this.filePath = filePath;
        //初始化并配置 ObjectMapper
        this.objectMapper = new ObjectMapper();
        //注册 JavaTimeModel 以支持 LocalTimeModel
        this.objectMapper.registerModule(new JavaTimeModule());
        //配置特性：将日期时间格式化为易读的字符串（例："2023-10-27T10:00:00"）
this.objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
        //配置特性：美化输出的 JSON 格式(带缩进)，便于阅读
```

```
this.objectMapper.disable(SerializationFeature.INDENT_OUTPUT);
    }

    @Override
    public void save(List<Note> notes){
        try {
            //使用 ObjectMapper 将 notes 列表写入到指定的文件中
objectMapper.writeValue(new File(filePath),notes);
        }catch (IOException e){
            //在实际应用中，这里应该抛出我们自定义的 FileOperationExpection
            e.printStackTrace();
        }
    }

    @Override
    public List<Note> load(){
        File file = new File(filePath);
        if(!file.exists() || file.length() == 0){
            return new ArrayList<>(); //如果文件不存在或为空，返回空列表          }
        try {
            //使用 objectMpapper 从文件中读取 JSON 数据，并将其转换为 NOTE 对象的数组，再转为列表
Note[] notesArray =objectMapper.readValue(file,Note[].class);
            return new ArrayList<>(Arrays.asList(notesArray));
        }catch (IOException e){
            e.printStackTrace();
            return new ArrayList<>();//加载失败也返回空列表，保证程序健壮性          }
    }

}
```

8.4 业务逻辑层

8.4.1 NoteService.java

```
package com.ZhangRuo.pkm.service;

import com.ZhangRuo.pkm.entity.Note;
import com.ZhangRuo.pkm.repository.StorageService;

import java.util.List;
```

```
import java.util.Optional;
import java.util.UUID;
import java.util.stream.Collectors;

/*
* [业务逻辑层]
* 封装所有与笔记相关的纯业务逻辑
* 它不关心数据具体如何存储，也不关心结果如何展示给用户
* */

public class NoteService {
    //依赖于 StorageService 接口，而不是具体的实现类，这是”面向接口编程“     private final
StorageService storageService;

    /*
    * 构造函数，用于接受外部传入的 StorageService 实例（依赖注入）      * @param storageService 一
个实现了 StorageService 接口的对象      * */
    public NoteService(StorageService storageService) {
        this.storageService = storageService;
    }

    /*
    * [业务逻辑]创建一篇新笔记      * 包含生成 ID、保存到存储等核心逻辑      *
    * @param title 笔记标题      * @param content 笔记内容      * @return 创建好的、带有唯一 ID
的 Note 对象      * */
    public Note createNote(String title, String content) {
        //1.核心业务校验         if (title == null || title.trim().isEmpty()) {
            throw new IllegalArgumentException("标题不能为空");
        }

        //2.加载当前所有笔记，为添加新笔记做准备          List<Note> notes =
storageService.load();

        //3.创建并设置新笔记的核心业务属性          Note newNote = new Note(title, content);
        newNote.setId(UUID.randomUUID().toString());//在 Service 层生成唯一 ID

        //4.将新笔记添加到列表中          notes.add(newNote);

        //5.将更新后的完整列表保存回去          storageService.save(notes);

        return newNote;

    }
```

```
    /*
    * [业务逻辑] 获取所有笔记     * @return 包含所有笔记的列表     * */
    public List<Note> getAllNotes() {
        return storageService.load();
    }

    /*
    * [业务逻辑] 根据 ID 查找一篇笔记     * @param id 要查找的笔记 ID
    * @return 一个包含 Note 的 Optional（如果找到），或一个空的 Optional(如果没找到)
    * */
    public Optional<Note> findNoteById(String id) {
        return storageService.load().stream()
                .filter(note -> note.getId() != null && note.getId().equals(id))
                .findFirst();
    }

    /*
    * [业务逻辑]根据 ID 删除一篇笔记     *@param id 要删除的笔记 ID
    * @return 如果成功删除则返回 true，否则返回 false
    * */
    public boolean deleteNote(String id) {
        List<Note> notes = storageService.load();
        //使用 removeIf 的方式高效删除         boolean removed = notes.removeIf(note ->
note.getId() != null && note.getId().equals(id));

        //如果真的删除了笔记，才需要执行保存操作         if (removed) {
            storageService.save(notes);
        }
        return removed;
    }

    /*
    * [业务逻辑]根据标签名查找所有相关的笔记     * @param tagName 要搜索的标签名     * @return 包
含该标签的所有 Note 对象的列表     * */
    public List<Note> findNotesByTag(String tagName) {
        if (tagName == null || tagName.isBlank()){
            return getAllNotes();//如果标签为空，则返回所有笔记         }
        //使用 Stream API 进行过滤         return storageService.load().stream()
                .filter(note -> note.hasTag(tagName))//只保留包含该标签的笔
记                .collect(Collectors.toList());//将结果收集到列表中

    }

    /*
```

```
    * [业务逻辑] 更新一篇已存在笔记的内容      * 会自动更新笔记的‘update’时间戳      *
    * @param id 要修改的笔记的 ID
    * @param newContent 新的笔记内容      * @return 如果更新成功，返回更新后的 Note 对象；如果笔记未找到，返回空的 Optional
    * */
    public Optional<Note> updateNoteContent(String id, String newContent) {
        List<Note> notes = storageService.load();

        //1.找到需要更新的笔记          Optional<Note> noteToUpdateOpt = notes.stream()
                .filter(note -> note.getId() != null && note.getId().equals(id))
                .findFirst();

        //2.如果找到了，就执行更新          if (noteToUpdateOpt.isPresent()) {
            Note noteToUpdate = noteToUpdateOpt.get();
            //调用 Note 自身的 setter 方法，该方法会自动更新时间戳
noteToUpdate.setContent(newContent);
            storageService.save(notes);//保存整个更新后的列表              return
Optional.of(noteToUpdate);
        }else {
            return Optional.empty();//如果没找到笔记，返回空          }

    }

    /*
    * [业务逻辑] 根据关键词搜索笔记      * 搜索范围包括笔记的标题和内容      *
    * @param keyword 要搜索的关键词      *@return 包含该关键词的笔记列表      * */
    public List<Note> searchNotesByKeyword(String keyword) {
        if (keyword == null || keyword.isBlank()){
            return List.of();//如果关键词为空，返回空列表          }
        String lowerKeyword = keyword.toLowerCase();//转换为小写以进行不区分大小写的搜索

        return storageService.load().stream()
                .filter(note ->                          //检查标题是否包含关键词
(note.getTitle() != null && note.getTitle().toLowerCase().contains(lowerKeyword)) ||
                        //或者检查内容是否包含关键词
(note.getContent() != null && note.getContent().toLowerCase().contains(lowerKeyword))
                )
                .collect(Collectors.toList());
    }



}
```

8.4.2 TagService.java

```
package com.ZhangRuo.pkm.service;

import com.ZhangRuo.pkm.entity.Note;
import com.ZhangRuo.pkm.repository.StorageService;

import java.util.List;
import java.util.Optional;

/**
 * [业务逻辑层]
 * 封装所有与标签管理相关的纯业务逻辑。
 */ public class TagService {

    private final StorageService storageService;

    /**
     * 构造函数，用于接收外部传入的 StorageService 实例（依赖注入）。      * @param storageService
一个实现了 StorageService 接口的对象。      */     public TagService(StorageService
storageService) {
        this.storageService = storageService;
    }

    /**
     * 为指定的笔记添加一个标签。      *
     * @param noteId  要添加标签的笔记 ID。      * @param tagName 要添加的标签名。      * @return
如果操作成功，返回更新后的 Note 对象；如果笔记未找到，返回空的 Optional。      */     public
Optional<Note> addTagToNote(String noteId, String tagName) {
        List<Note> notes = storageService.load();

        // 1. 找到需要修改的笔记        Optional<Note> noteToUpdateOpt = notes.stream()
                .filter(note -> note.getId() != null && note.getId().equals(noteId))
                .findFirst();

        // 2. 如果找到了，就执行修改        if (noteToUpdateOpt.isPresent()) {
            Note noteToUpdate = noteToUpdateOpt.get();
            noteToUpdate.addTag(tagName); // 调用 Note 自身的 addTag 方法
storageService.save(notes); // 保存整个更新后的列表            return
Optional.of(noteToUpdate);
        } else {
```

```
            return Optional.empty(); // 如果没找到笔记，返回空         }
    }

    /**
     * 为指定的笔记移除一个标签。         *
     * @param noteId   要移除标签的笔记 ID。         * @param tagName 要移除的标签名。         * @return
如果操作成功，返回更新后的 Note 对象；如果笔记未找到，返回空的 Optional。         */
    public Optional<Note> removeTagFromNote(String noteId, String tagName) {
        List<Note> notes = storageService.load();

        Optional<Note> noteToUpdateOpt = notes.stream()
                .filter(note -> note.getId() != null && note.getId().equals(noteId))
                .findFirst();

        if (noteToUpdateOpt.isPresent()) {
            Note noteToUpdate = noteToUpdateOpt.get();
            noteToUpdate.removeTag(tagName); // 调用 Note 自身的 removeTag 方法
storageService.save(notes);
            return Optional.of(noteToUpdate);
        } else {
            return Optional.empty();
        }
    }
}
```

8.4.3 ExportService.java

```
package com.ZhangRuo.pkm.service;

import com.ZhangRuo.pkm.entity.Note;
import com.ZhangRuo.pkm.enums.ExportFormat;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.time.format.DateTimeFormatter;
import java.util.List;

/*
```

```
* [业务逻辑层]
* 封装所有与笔记导出相关的业务逻辑
* */ public class ExportService {

    /*
    * 将笔记列表导出到指定格式的文件     * @param notes 要导出的笔记列表     * @param filePath
导出的文件路径     * @param format 导出的文件格式     * @throws IOException 如果写入文件时发生
错误     * */     public void exportNotes(List<Note> notes,String filePath, ExportFormat
format) throws IOException {
        switch (format) {
            case TEXT:
                exportToTextFile(notes,filePath);
                break;
            //之后可扩展 case JSON 等

        }
    }

    private void exportToTextFile(List<Note> notes,String filePath) throws IOException {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            for (int i = 0;i < notes.size();i++) {
                Note note = notes.get(i);

                writer.write("标题：" + note.getTitle());
                writer.newLine();
                writer.write("创建时间：" + note.getCreatedAt().format(formatter));
                writer.newLine();
                writer.write("最后修改：" + note.getUpdatedAt().format(formatter));
                writer.newLine();
                writer.write("标签：" + String.join(", ", note.getTags()));
                writer.newLine();
                writer.write("内容：");
                writer.newLine();
                writer.write(note.getContent());
                writer.newLine();

                if (i < notes.size() - 1) {
                    writer.write("---");
                    writer.newLine();
                }
            }
        }
```

```
    }
}
```

8.5 表现层

8.5.1 App.java

```
package com.ZhangRuo.pkm.app;

import com.ZhangRuo.pkm.cli.CommandParser;

/**
 * [表现层] 应用程序的唯一主入口。
 * 职责：创建并启动命令解析器。
 */ public class App {

    /**
     * Java 程序的主方法。       * @param args 命令行参数。       */     public static void
main(String[] args) {
        // 1. 创建整个应用的“总指挥” -> CommandParser
        CommandParser parser = new CommandParser();

        // 2. 将命令行参数交给 parser 处理，由它决定启动模式         parser.parseArgs(args);

        // 3. (可选但重要) 在程序结束时关闭资源，例如 CommandParser 中的 Scanner
        parser.close();
    }
}
```

8.5.2 CommandParser.java

```
package com.ZhangRuo.pkm.cli;

import com.ZhangRuo.pkm.controller.NoteController;
import com.ZhangRuo.pkm.controller.TagController;
import com.ZhangRuo.pkm.entity.Note;
```

```
import com.ZhangRuo.pkm.repository.JsonStorageService;
import com.ZhangRuo.pkm.repository.StorageService;
import com.ZhangRuo.pkm.service.ExportService;
import com.ZhangRuo.pkm.service.TagService;
import com.ZhangRuo.pkm.service.NoteService;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;


/*
* [表现层]
* 核心中的核心：命令解析器和分发器
* 负责装配整个应用，并管理主控循环（REPL）
* */

public class CommandParser {

    //--- 依赖的控制器 ---
    private final NoteController noteController;
    private final TagController tagController;

    //--- REPL 相关的状态 ---
    private final Scanner scanner;
    private boolean isRunning;

    private List<Note> lastListedNotes;//用于缓存上一次 list 或 search 的结果


    /**
     * [新增] 程序的总入口方法。       * 根据传入的命令行参数决定启动模式。       * @param args 来
自 main 方法的命令行参数。       */
    public void parseArgs(String[] args) {
        if (args.length == 0) {
            // 如果没有提供任何参数，则启动交互模式             startInteractiveMode();
        } else {
            // 如果提供了参数，则将它们拼接成一个命令字符串并直接执行             String
commandLine = String.join(" ", args);
            executeCommand(commandLine);
        }
    }
```

```
    /*
    * [对应"构造函数正确装配依赖"]
    * 构造函数负责"装配"整个应用程序的依赖关系      * 这是一个典型的依赖注入(DI)容器的简化实现
* */
    public CommandParser(){
        //1.从底层开始创建：数据持久层         StorageService storageService = new
JsonStorageService();

        //2.创建业务逻辑层：并注入其依赖         NoteService noteService = new
NoteService(storageService);
        TagService tagService = new TagService(storageService);

        ExportService exportService = new ExportService();

        //3.创建控制器层，并注入其依赖         //注意：TagController 可能也需要 NoteService 来获
取笔记标题等信息         this.noteController = new NoteController(noteService,exportService);
        this.tagController = new TagController(tagService);

        //4.初始化 REPL 组件         this.scanner = new Scanner(System.in);
        this.isRunning = true;

        this.lastListedNotes = new ArrayList<>();

    }

    /*
    * 启动交互式命令行模式（REPL：Read-Eval-Print_Loop）      * */
    public void startInteractiveMode(){
        System.out.println("> 欢迎使用个人知识管理系统（CLI 版）");
        System.out.println("> 输入 help 查看可用命令");

        while(isRunning){
            System.out.print("pkm> ");
            String input = scanner.nextLine().trim();

            if (!input.isEmpty()){
                executeCommand(input);
            }
        }
    }

    /**
```

```
    * 【解析器】      * 解析单行命令字符串，并将其分发给调度器。      * 职责：只负责解析，不负责执行。      */
    private void executeCommand(String input) {
        // 1. 将输入字符串按照第一个空格分割成 [命令] 和 [可能存在的参数字符串]
        String[] parts = input.split("\\s+", 2);
        String command = parts[0].toLowerCase();

        // 2. 直接调用分发器，把最原始的 parts 数组传过去，让分发器自己决定怎么用
dispatchCommand(command, parts);
    }

    /**
     * 【分发器 (Dispatcher)】      * 根据命令字符串，调用对应的处理方法。      * 职责：负责"switch" 决策，并为不同的 handle 方法准备正确的参数。      * @param command 解析出的小写命令(e.g., "list", "view")
     * @param parts   原始的、包含命令和参数的字符串数组 (e.g., ["view", "1"])
     */
    private void dispatchCommand(String command, String[] parts) {
        // 为那些需要 "arg1 arg2 ..." 格式的 handle 方法准备参数        String[] simpleArgs = (parts.length > 1) ? parts[1].split("\\s+") : new String[0];
        // 为那些需要完整参数字符串的 handle 方法准备参数        String fullParams = (parts.length > 1) ? parts[1] : "";

        switch (command) {
            case "new":
                handleNewCommand(fullParams);
                break;
            case "list":
                handleListCommand(simpleArgs);
                break;
            case "view":
                handleViewCommand(simpleArgs);
                break;
            case "edit":
                handleEditCommand(fullParams);
                break;
            case "delete":
                handleDeleteCommand(simpleArgs);
                break;
            case "tag":
                handleTagCommand(simpleArgs);
                break;
            case "untag":
                handleUntagCommand(simpleArgs);
```

```
                break;
            case "search":
                handleSearchCommand(fullParams);
                break;
            case "export":
                handleExportCommand(simpleArgs);
                break;
            case "export-all":
                handleExportAllCommand(simpleArgs);
                break;
            case "exit":
                handleExitCommand();
                break;
            case "help":
                printHelp();
                break;
            default:
                System.err.println("❌ 未知命令: '" + command + "'。输入 'help' 查看可用命令。
");
                break;
        }
    }

    /**
     * 关闭资源，例如 Scanner。      */
    public void close() {
        scanner.close();
    }




    // --- 私有的 handle...() 方法，负责参数校验和调用 Controller ---
    private void handleNewCommand(String params) {
        // 一个简单的参数解析，假设标题和内容用双引号包围        String[] parts =
params.split("\"", 4);
        if (parts.length < 3) {
            System.err.println("❌ 参数错误！用法: new \"<标题>\" \"<内容>\"");
            return;
        }
        String title = parts[1];
        String content = parts[3];
        noteController.createNote(title, content);
    }
```

```
    /*
    * 处理list命令     * 检查是否存在 --tag 参数     *@param args list 命令后面的所有参数     *
*/
    private void handleListCommand(String[] args) {
        String tagName = null;
        //检查参数是否是”--tag“ 并且后面还跟着一个标签名         if (args.length == 2 &&
"--tag".equals(args[0])) {
            tagName = args[1];
        }else if (args.length > 0){
            //如果有其他无法识别的参数，打印错误信息             System.err.println("❌ 参数错
误! 用法: list 或 list --tag<标签名>");
            return;

        }
        //接受返回值并存入缓存         this.lastListedNotes =
noteController.listNotes(tagName);

    }


    private void handleViewCommand(String[] args) {
        if (args.length != 1) {
            System.err.println("❌ 参数错误! 用法: view <短 ID>");
            return;
        }

        try {
            // 1. 尝试将输入解析为短 ID (数字)
            int displayId = Integer.parseInt(args[0]);

            //如果缓存是空的，就主动执行一次 list 来填充它             if
(lastListedNotes.isEmpty()) {
                System.out.println("ⓘ  首次操作，正在刷新笔记列表...");
                handleListCommand(new String[0]); // 调用 list 命令的处理器             }

            // 2. 检查短 ID 是否有效             if (displayId > 0 && displayId <=
lastListedNotes.size()) {
                // 3. 从缓存中获取真实 ID
                String realId = lastListedNotes.get(displayId - 1).getId();
                noteController.viewNoteById(realId);
            } else {
                System.err.println("❌ 错误: 无效的短 ID '" + displayId + "'。请从下面的列表选
择。");
```

```
            // 如果 ID 无效，再次打印列表，方便用户选择                handleListCommand(new String[0]);
            }
        } catch (NumberFormatException e) {
            // 4. 如果用户输入的不是数字，我们仍然可以尝试把它当作 UUID 来处理 (兼容老用法)
            System.out.println("ℹ️ 尝试将输入作为完整 ID 进行查找...");
            noteController.viewNoteById(args[0]);
        }
    }


    /*
    * 处理 edit 命令     * 解析出笔记 ID 和带引号的新内容     * @param params edit 命令后面的所有参数字符串     * */
    private void handleEditCommand(String params) {
        String[] parts = params.split("\"", 3);
        if (parts.length < 2 || parts[0].trim().isEmpty()) {
            System.err.println("❌ 参数错误！用法: edit <短 ID 或 完整 ID> \"<新内容>\"");
            return;
        }

        String idArg = parts[0].trim();
        String newContent = parts[1];

        try {
            int displayId = Integer.parseInt(idArg);

            // 【智能填充】            if (lastListedNotes.isEmpty()) {
                System.out.println("ℹ️ 首次操作 ID，正在刷新笔记列表...");
                handleListCommand(new String[0]);
            }

            if (displayId > 0 && displayId <= lastListedNotes.size()) {
                String realId = lastListedNotes.get(displayId - 1).getId();
                noteController.editNote(realId, newContent);
                lastListedNotes.clear(); // 清空缓存            } else {
                System.err.println("❌ 错误: 无效的短 ID '" + displayId + "'。");
            }
        } catch (NumberFormatException e) {
            noteController.editNote(idArg, newContent);
        }
    }
```

```
    private void handleDeleteCommand(String[] args) {
        if (args.length != 1) {
            System.err.println("❌ 参数错误！用法：delete <短 ID 或 完整 ID>");
            return;
        }

        try {
            int displayId = Integer.parseInt(args[0]);

            // 【智能填充】            if (lastListedNotes.isEmpty()) {
                System.out.println("ℹ️ 首次操作 ID，正在刷新笔记列表...");
                handleListCommand(new String[0]);
            }

            if (displayId > 0 && displayId <= lastListedNotes.size()) {
                String realId = lastListedNotes.get(displayId - 1).getId();
                noteController.deleteNoteById(realId);
                // 操作成功后，缓存可能已过时，清空它以便下次重新加载
lastListedNotes.clear();
            } else {
                System.err.println("❌ 错误：无效的短 ID '" + displayId + "'。");
            }
        } catch (NumberFormatException e) {
            noteController.deleteNoteById(args[0]);
        }
    }

    private void handleTagCommand(String[] args) {
        if (args.length != 2) {
            System.err.println("❌ 参数错误！用法：tag <短 ID 或 完整 ID> <标签名>");
            return;
        }
        String idArg = args[0];
        String tagName = args[1];

        try {
            int displayId = Integer.parseInt(idArg);

            // 【智能填充】            if (lastListedNotes.isEmpty()) {
                System.out.println("ℹ️ 首次操作 ID，正在刷新笔记列表...");
                handleListCommand(new String[0]);
            }

            if (displayId > 0 && displayId <= lastListedNotes.size()) {
```

```
            String realId = lastListedNotes.get(displayId - 1).getId();
            tagController.addTagToNote(realId, tagName);
            lastListedNotes.clear(); // 清空缓存            } else {
            System.err.println("❌ 错误: 无效的短 ID '" + displayId + "'。");
        }
    } catch (NumberFormatException e) {
        tagController.addTagToNote(idArg, tagName);
    }
}



private void handleUntagCommand(String[] args) {
    if (args.length != 2) {
        System.err.println("❌ 参数错误! 用法: untag <短 ID 或 完整 ID> <标签名>");
        return;
    }
    String idArg = args[0];
    String tagName = args[1];

    try {
        int displayId = Integer.parseInt(idArg);

        // 【智能填充】            if (lastListedNotes.isEmpty()) {
            System.out.println("ℹ️ 首次操作 ID，正在刷新笔记列表...");
            handleListCommand(new String[0]);
        }

        if (displayId > 0 && displayId <= lastListedNotes.size()) {
            String realId = lastListedNotes.get(displayId - 1).getId();
            tagController.removeTagFromNote(realId, tagName);
            lastListedNotes.clear(); // 清空缓存            } else {
            System.err.println("❌ 错误: 无效的短 ID '" + displayId + "'。");
        }
    } catch (NumberFormatException e) {
        tagController.removeTagFromNote(idArg, tagName);
    }
}



/*
 * 处理 search 命令     * 解析出带引号的关键词     * @param params search 命令后面的所有参数
字符串     * */
// --- 这是修改后的 handleSearchCommand 方法 ---
```

```
    private void handleSearchCommand(String params) {
        if (params == null || params.isBlank()) {
            System.err.println("❌ 参数错误！用法：search <关键词> 或 search \"<带空格的关键词>\"");
            return;
        }

        String keyword;
        // 检查参数是否以双引号开头和结尾         if (params.startsWith("\"") && params.endsWith("\"")) {
            // 如果是，就提取引号内部的内容             // 使用 substring 去掉首尾的双引号
            keyword = params.substring(1, params.length() - 1);
        } else {
            // 如果没有引号，就把整个参数作为关键词             // (这种方式只支持不含空格的单个关键词)
            keyword = params;
        }

        if (keyword.isEmpty()) {
            System.err.println("❌ 参数错误！关键词不能为空。");
            return;
        }

        noteController.searchNote(keyword);
    }



    private void handleExportCommand(String[] args) {
        if (args.length != 3) {
            System.err.println("❌ 参数错误！用法：export <短 ID 或 完整 ID> <格式> <路径>");
            return;
        }

        String idArg = args[0];
        String format = args[1];
        String path = args[2];

        try {
            // 1. 尝试将 ID 参数解析为短 ID (数字)
            int displayId = Integer.parseInt(idArg);

            // 2. 【智能填充】如果缓存为空，主动执行 list
```

```
            if (lastListedNotes.isEmpty()) {
                System.out.println("ℹ️ 首次操作 ID，正在刷新笔记列表...");
                handleListCommand(new String[0]); // 模拟执行 "list"
            }

            // 3. 检查短 ID 是否有效            if (displayId > 0 && displayId <=
lastListedNotes.size()) {
                // 4. 从缓存中获取真实 ID
                String realId = lastListedNotes.get(displayId - 1).getId();
                // 5. 调用 Controller 时，传入的是真实 ID
                noteController.exportNote(realId, format, path);
            } else {
                System.err.println("❌ 错误: 无效的短 ID '" + displayId + "'。");
            }
        } catch (NumberFormatException e) {
            // 6. 如果用户输入的不是数字，我们仍然可以尝试把它当作 UUID 来处理 (兼容老用法)
            System.out.println("ℹ️ 尝试将输入作为完整 ID 进行查找...");
            noteController.exportNote(idArg, format, path);
        }
    }


    /**
     * 处理 export-all 命令。      * 解析出格式和路径。      * @param args export-all 命令后面
的所有参数。      */
    private void handleExportAllCommand(String[] args) {
        if (args.length != 2) {
            System.err.println("❌ 参数错误! 用法: export-all <格式> <路径>");
            return;
        }
        String format = args[0];
        String path = args[1];
        noteController.exportAllNotes(format, path);
    }




    private void handleExitCommand() {
        this.isRunning = false;
        System.out.println("👋 再见!");
    }

    private void printHelp() {
```

```
        System.out.println("--- 可用命令 ---");
        System.out.println("  new \"<标题>\" \"<内容>\"   - 创建一篇新笔记");
        System.out.println("  list                    - 列出所有笔记");
        System.out.println("  view <笔记 ID>           - 查看笔记详情");
        System.out.println("  edit <笔记 ID>\'<新内容>\'  - 编辑一篇笔记的内容");
        System.out.println("  delete <笔记 ID>         - 删除一篇笔记");
        System.out.println("  tag <笔记 ID> <标签名>    - 为笔记添加标签");
        System.out.println("  untag <笔记 ID> <标签名>  - 为笔记移除标签");
        System.out.println("  search <关键词>         - 搜索标题或内容包含关键词的笔记");
        System.out.println("  export <笔记 ID> <格式> <路径> - 导出单篇笔记");
        System.out.println("  export-all <格式> <路径> - 导出所有笔记");
        System.out.println("  exit                    - 退出程序");
        System.out.println("  help                    - 显示此帮助信息");
        System.out.println("----------------");
    }
}
```

8.5.3 NoteController.java

```
package com.ZhangRuo.pkm.controller;

import com.ZhangRuo.pkm.entity.Note;
import com.ZhangRuo.pkm.service.ExportService;
import com.ZhangRuo.pkm.service.NoteService;
import com.ZhangRuo.pkm.enums.ExportFormat;

import java.util.List;
import java.io.IOException;
import java.util.Optional;
import java.time.format.DateTimeFormatter;

/*
* [控制器层]
*负责处理与笔记相关的用户交互逻辑
* 它接受来自 CommandParser 的指令，调用 NoteService 完成工作
* 并将结果格式化后输出到控制台
* */
```

```
public class NoteController {

    private final NoteService noteService;
    private final ExportService exportService;

    /*
    * 构造函数，用于接收外部传入的 NoteService 实例（依赖注入）      * */     public
NoteController(NoteService noteService , ExportService exportService) {
        this.noteService = noteService;
        this.exportService = exportService;
    }

    /*
    * [交互逻辑] 处理创建新笔记的请求      * */

    public void createNote(String title, String content) {
        try {
            Note newNote = noteService.createNote(title, content);
            //交互逻辑：成功信息的展示逻辑             System.out.println("☑ 笔记创建成功！");
            System.out.println("ID: " + newNote.getId());
            System.out.println("标题: " + newNote.getTitle());
        }catch (IllegalArgumentException e){
            //失败信息的展示逻辑             System.out.println("❌ 错误: " + e.getMessage());
        }
    }



    /**
     * [交互逻辑] 处理列出笔记的请求，支持按标签过滤。       * @param tagName 如果不为 null，则只
列出包含该标签的笔记。       * @return 查询到的笔记列表，用于上层缓存。       */
    public List<Note> listNotes(String tagName) { // 1. 返回值从 void 改为 List<Note>
List<Note> notes;
        if (tagName != null) {
            notes = noteService.findNotesByTag(tagName);
            System.out.println("--- 标签为 '" + tagName + "' 的笔记列表 ---");
        } else {
            notes = noteService.getAllNotes();
            System.out.println("--- 所有笔记列表 ---");
        }

        if (notes.isEmpty()) {
            System.out.println("ℹ️  没有找到符合条件的笔记。");
            return notes; // 2. 在这里也要返回 notes 列表         }
```

```
        System.out.println("-----------------------------------");
        for (int i = 0; i < notes.size(); i++) {
            Note note = notes.get(i);
            int displayId = i + 1; // 用户的“短 ID”

            String tags = String.join(", ", note.getTags());
            // 使用短 ID 进行打印            System.out.printf("[%d] %s (%s) [%s]%n",
                    displayId,
                    note.getTitle(),
                    note.getCreatedAt().toLocalDate().toString(),
                    tags);
        }
        System.out.println("-----------------------------------");

        return notes; // 3. 在方法末尾返回最终的列表     }



    /*
    * [交互逻辑] 处理根据 ID 查看笔记详情的请求     * */
    public void viewNoteById(String id) {
        Optional<Note> noteOpt = noteService.findNoteById(id);
        if (noteOpt.isPresent()) {
            Note note = noteOpt.get();

            // 2. 创建一个我们想要的格式化模板            DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss");

            System.out.println("--- 笔记详情 ---");
            System.out.println("ID      : " + note.getId());
            System.out.println("标题    : " + note.getTitle());
            System.out.println("标签    : " + String.join(", ", note.getTags()));

            // 3. 在打印时，使用 .format(formatter) 来应用模板            System.out.println("
创建时间: " + note.getCreatedAt().format(formatter));
            System.out.println("更新时间: " + note.getUpdatedAt().format(formatter));

            System.out.println("----------------");
            System.out.println("内容:\n" + note.getContent());
            System.out.println("----------------");
        } else {
            System.err.println("❌ 错误: 未找到 ID 为 '" + id + "' 的笔记。");
        }
```

```
    }

    /*
    * [交互逻辑] 处理删除笔记的请求     * */
    public void deleteNoteById(String id){
        boolean deleted = noteService.deleteNote(id);
        if (deleted){
            System.out.println("☑ 笔记 (ID: " + id + ") 已被成功删除。");
        }else {
            System.err.println("❌ 错误: 未找到 ID 为 '" + id + "' 的笔记，删除失败。");
        }
    }

    /*
    * [交互逻辑]处理编辑笔记内容的请求     *
    * @Param id  要编辑的笔记 ID
    * @Param newContent  新的笔记内容     * */
    public void editNote(String id, String newContent) {
        Optional<Note> updateNoteOpt = noteService.updateNoteContent(id ,newContent);

        if (updateNoteOpt.isPresent()){
            System.out.println("☑ 笔记 (ID: " + id + ") 内容已成功更新。");
        }else {
            System.err.println("❌ 错误: 未找到 ID 为 '\" + id + \"' 的笔记，编辑失败。");
        }
    }

    /*
    * [交互逻辑] 处理根据关键词搜索笔记的请求     *
    * @param keyword 搜索关键词     * */
    public List<Note> searchNote(String keyword) {
        List<Note>notes = noteService.searchNotesByKeyword(keyword);

        System.out.println("--- 关键词为 ‘"+keyword+"’ 的搜索结果 ---");

        if (notes.isEmpty()){
            System.out.println("ⓘ  没有找到包含该关键词的笔记。");
            return notes;
        }

        //复用 list 命令的输出格式         for (Note note : notes) {
            String tags = String.join(", ", note.getTags());
            System.out.printf("[%s] %s (%s) [%s] %n",
                    note.getId(),
```

```
                note.getTitle(),
                note.getCreatedAt().toLocalDate().toString(),
                tags);
        }
        System.out.println("--------------------");

        return notes;

    }

    /*
    * [交互逻辑] 导出笔记     * */
    public void exportNote(String id,String formatStr,String path) {
        //1.查找笔记         Optional<Note> noteOpt = noteService.findNoteById(id);
        if (noteOpt.isEmpty()){
            System.err.println("❌ 错误: 未找到 ID 为 '" + id + "' 的笔记，导出失败。");
            return;
        }

        //2.解析格式         ExportFormat format;
        try{
            format = ExportFormat.valueOf(formatStr.toUpperCase());
        }catch (IllegalArgumentException e){
            System.err.println("❌ 错误: 不支持的导出格式 '" + formatStr + "' 。目前支持 TEXT");
            return;
        }

        // 3. 调用导出服务         try {
            // 将单篇笔记放入一个列表中进行导出
exportService.exportNotes(List.of(noteOpt.get()), path, format);
            System.out.println("☑ 笔记 (ID: " + id + ") 已成功导出到: " + path);
        } catch (IOException e) {
            System.err.println("❌ 错误: 导出文件时发生错误: " + e.getMessage());
        }

    }

    /*
    * [交互逻辑] 处理导出所有笔记的请求     *
    * @param formatStr 导出的格式字符串（e.g.“text”）     * @param path 导出的文件路径     * */
    public void exportAllNotes(String formatStr, String path) {
        // 1. 获取所有笔记         List<Note> allNotes = noteService.getAllNotes();
        if (allNotes.isEmpty()) {
            System.out.println("ℹ 当前没有任何笔记可以导出。");
```

```
            return;
        }

        // 2. 解析格式 (与 exportNote 方法逻辑相同)
        ExportFormat format;
        try {
            format = ExportFormat.valueOf(formatStr.toUpperCase());
        } catch (IllegalArgumentException e) {
            System.err.println("❌ 错误: 不支持的导出格式 '" + formatStr + "'。目前仅支持
'TEXT'。");
            return;
        }

        // 3. 调用导出服务         try {
            exportService.exportNotes(allNotes, path, format);
            System.out.println("☑ 所有 " + allNotes.size() + " 篇笔记已成功导出到: " + path);
        } catch (IOException e) {
            System.err.println("❌ 错误: 导出文件时发生错误: " + e.getMessage());
        }
    }



}
```

8.5.4 TagController.java

```
package com.ZhangRuo.pkm.controller;

import com.ZhangRuo.pkm.entity.Note;
import com.ZhangRuo.pkm.service.TagService;

import java.util.Optional;

/**
 * [控制器层]
 * 负责处理与标签管理相关的用户交互逻辑。
 */ public class TagController {
```

```
    private final TagService tagService;

    public TagController(TagService tagService) {
        this.tagService = tagService;
    }

    /**
     * [交互逻辑] 处理为笔记添加标签的请求。      */     public void addTagToNote(String noteId,
String tagName) {
        Optional<Note> updatedNoteOpt = tagService.addTagToNote(noteId, tagName);

        if (updatedNoteOpt.isPresent()) {
            Note updatedNote = updatedNoteOpt.get();
            System.out.println("✅ 标签 '" + tagName + "' 已成功添加到笔记 '" +
updatedNote.getTitle() + "'。");
            System.out.println("   当前标签: " + String.join(", ", updatedNote.getTags()));
        } else {
            System.err.println("❌ 错误: 未找到 ID 为 '" + noteId + "' 的笔记，添加标签失败。
");
        }
    }

    /**
     * [交互逻辑] 处理为笔记移除标签的请求。      */     public void removeTagFromNote(String
noteId, String tagName) {
        Optional<Note> updatedNoteOpt = tagService.removeTagFromNote(noteId, tagName);

        if (updatedNoteOpt.isPresent()) {
            Note updatedNote = updatedNoteOpt.get();
            System.out.println("✅ 标签 '" + tagName + "' 已成功从笔记 '" +
updatedNote.getTitle() + "' 移除。");
        } else {
            System.err.println("❌ 错误: 未找到 ID 为 '" + noteId + "' 的笔记，或该笔记不含此标
签，移除失败。");
        }
    }
}
```