北京理工大学计算机学院

《Android 技术开发基础》

课程设计开发文档

目录

1 App 的运行与开发环境

（1） **运行环境：** 10.0 以上版本 Android 的 Android 手机
（2） **部署方法：** 直接安装 Academia.apk 即可
（3） **开发环境：** Android Studio 2024.3.2
（4） **手写代码行数：** 手机端约 4300 行（Kotlin）

2 App 功能说明

Academia 是一款针对**科学类英语论文**的便捷的**搜索与管理应用**。其核心功能基于 Arxiv API，并提供多项功能，包括：**简洁的搜索引擎、论文预览、用户及论文管理系统、云端 AI Agent** 等。多数功能需要网络连接。

（1） 系统功能分解图

整个应用的所有界面/状态、功能如以下总览图所示：

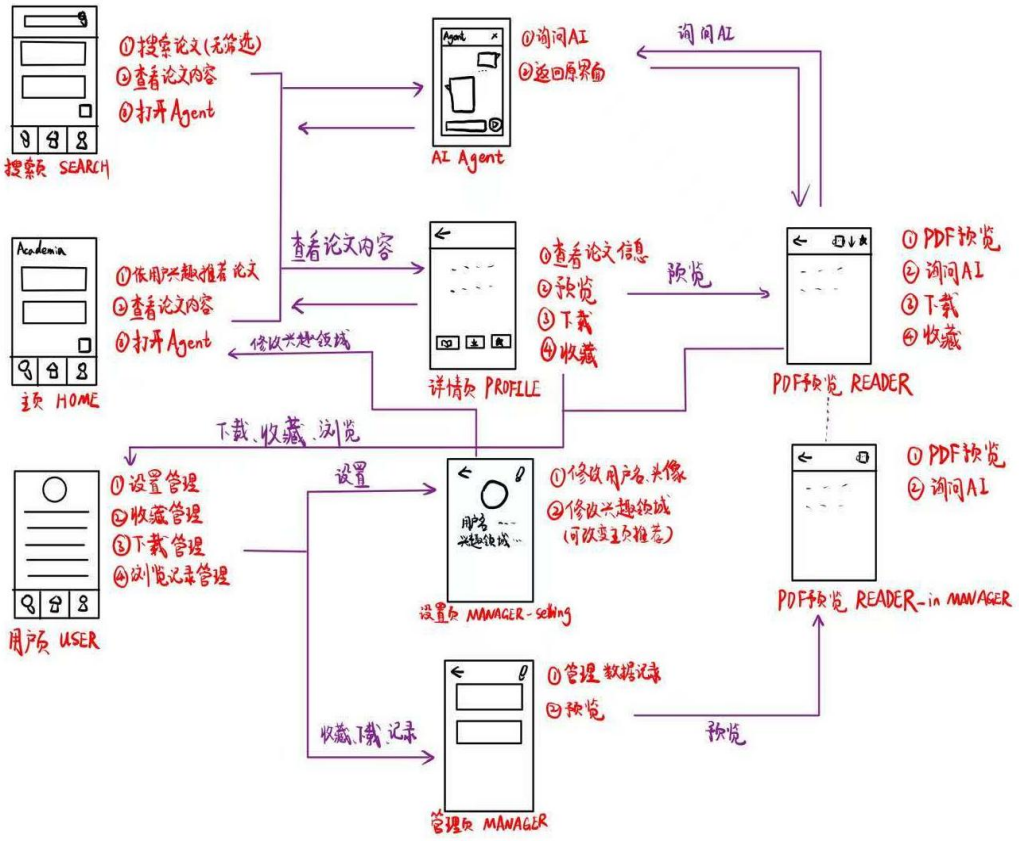

图 2.1 Academia 应用功能总览图

a） 主页 HOME 与搜索页 SEARCH

1. 论文推荐与搜索功能

主页是用户进入应用时看到的首个界面，包含论文推荐的功能，提供了查看论文内容、打开 AI Agent 的入口。搜索页包含了论文搜索的功能，也提供了查看论文内容、打开 AI Agent 的入口。其中**论文查看只需要点击论文卡片即可进入详情页；点击浮动按钮 FAB 进入 Agent 页；点击主页右上角的头**

像可以进入用户页。

通过底部栏，用户可以在主页、搜索页、用户页之间转换。搜索页的核心控件 PaperScreen 和主页一致，但其内容是通过顶部的搜索栏输入控制的。**输入关键词（由于 Arxiv 数据库限制，只支持英语），能够返回相应的信息。**

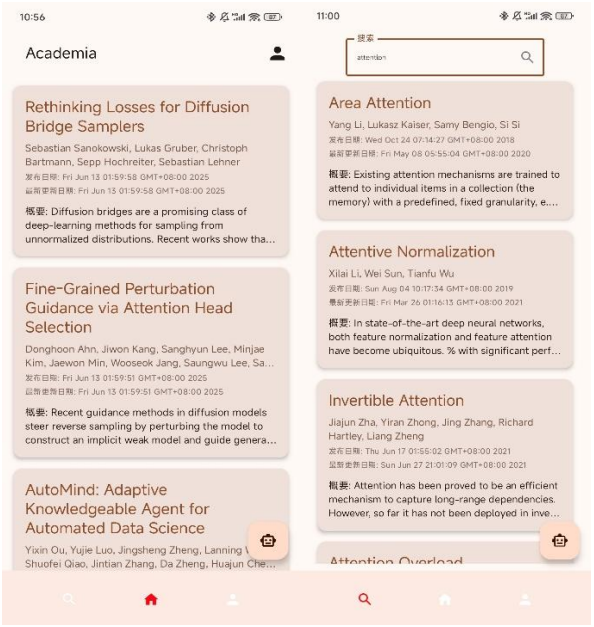

图 2.2 主页、搜索页论文推荐与搜索功能

由于该接口是第三方接口，且受到网络、地域等影响，有时无法稳定地返回结果。为此，Academia 提供了提示：获取论文 10s 后若无结果会弹出**“网络响应缓慢”**的提示；获取论文 30s 后仍无结果会终止请求并弹出**“请求超时，请重试”**的提示；获取论文中途捕捉到由第三方接口返回的**异常**（通常是网络问题）会弹出**“出现异常，请重试”**的提示；若**无连接**会弹出**“网络不可用，请检查连接"**的提示。

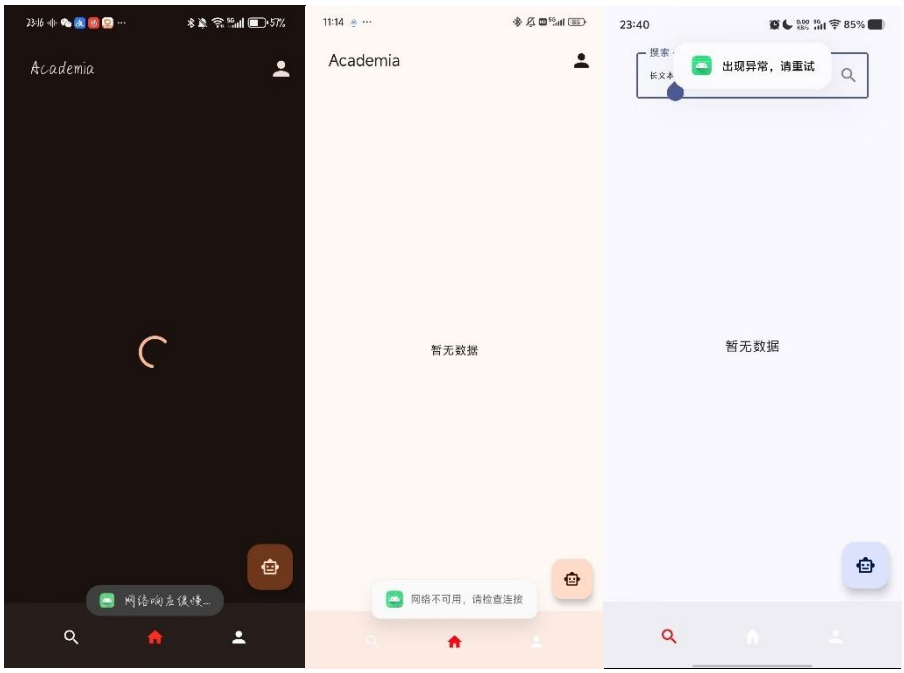

图 2.3 网络检测效果测试图

2. 下拉刷新、上滑更新

为增强用户使用体验，Academia 提供了下拉刷新、上滑更新的功能。

在暂无请求的时候，从顶部下拉组件，出现进度提示，随即进入刷新；面对因网络而导致请求失败的情况，也可以通过刷新来反复获得请求。

在搜索界面，可以通过上滑更新更多结果。每次请求返回 10 个结果并显示在界面上，从而避免一次搜索结果过多导致的显示问题。而**主界面的推荐功能并不支持上滑更新，而是一次推荐 30 篇论文。**

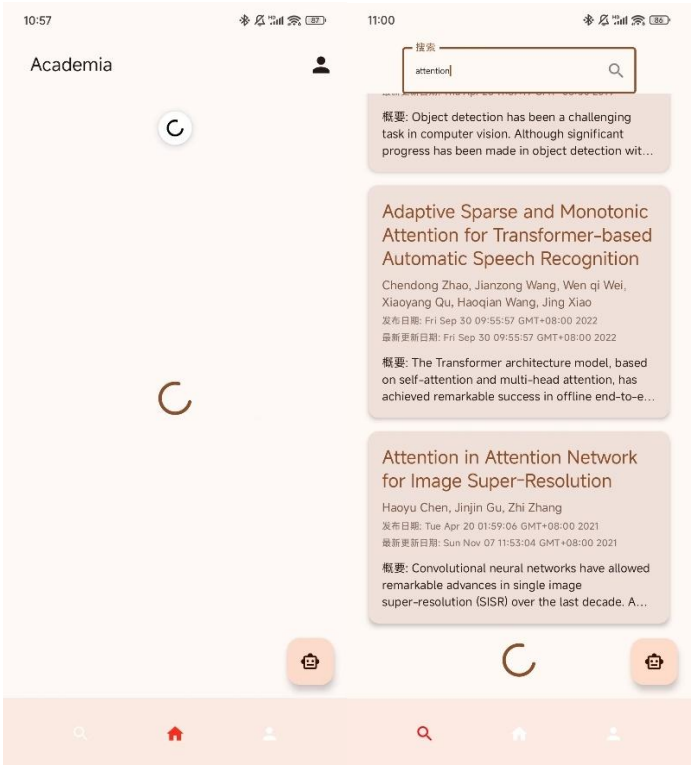

图 2.4 下拉刷新、上滑更新实机效果图

b) 详情页 PROFILE

详情页是展开论文描述的界面，提供了论文描述预览与论文管理的功能。点击论文卡片即可进入详情页。

详情页中，主要提供了预览、下载、收藏的功能。点击预览按钮，会进入 PDF 阅读器界面（见后文）。点击收藏按钮，目标数据加入收藏表，再次点击取消收藏。

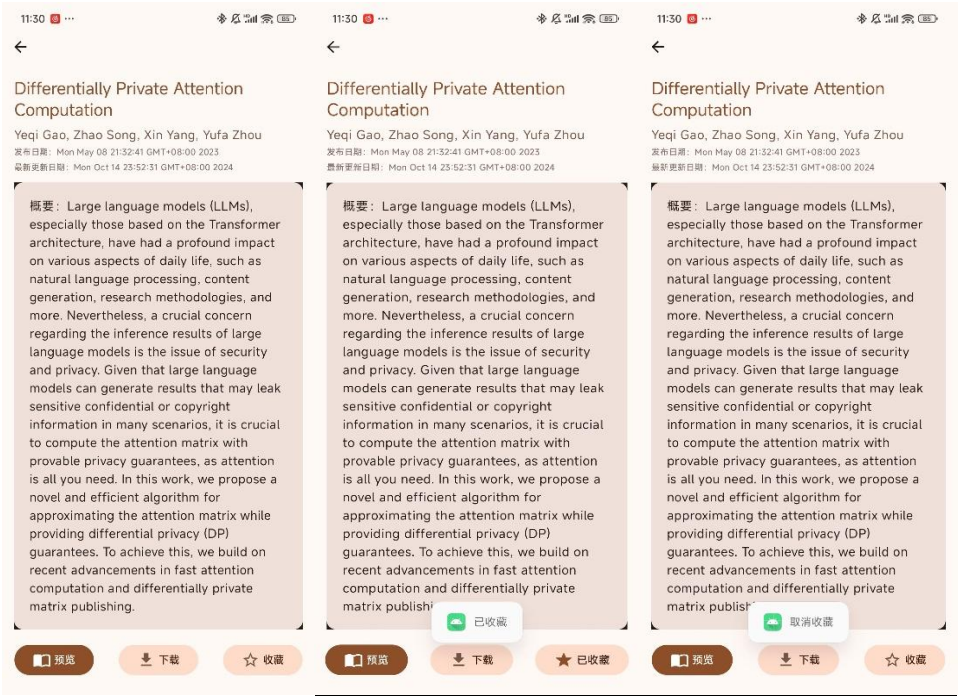

图 2.5 详情页实机界面与收藏功能

点击下载按钮，应用会开始**从网络下载文章，随后按键进入处理中状态且被禁用，直到处理完成**，按键状态变化，目标被下载在本地文件当中；这条数据被加入下载表。若**再次点击，会弹出删除下载确认弹窗，确认即可删除**，下载表也更新。**有时，因为 Arxiv 数据库没有及时更新文章，可能会导致找不到目标文件，这时会弹窗“下载失败”。**

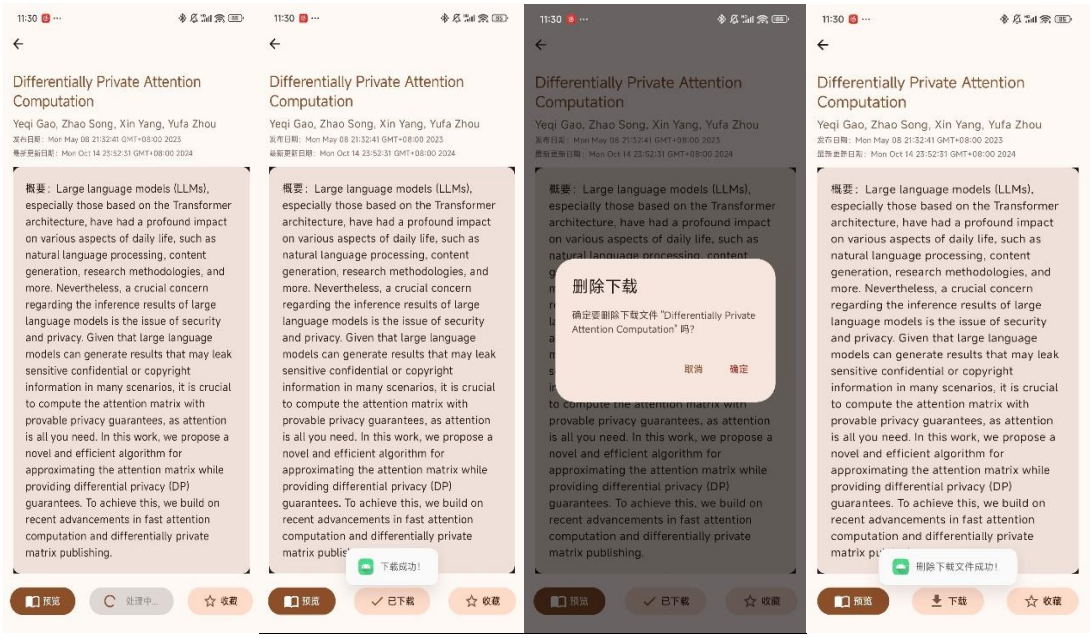

图 2.6 详情页下载功能

c) 用户页 USER 和管理页 MANAGER

用户页提供了简洁的用户系统和论文管理系统。

1. 设置界面

设置界面为用户提供了**修改用户名、头像、兴趣领域**的功能。

本应用并没有独立的用户系统，而是使用 DataStore 来存储本机用户信息。这样设计是为了避免额外的注册、登录功能影响应用的间接性与便利性，即这类用户信息与数据存储无关。用户可以自行修改用户名、头像。

设置界面还支持**设置兴趣领域**，功能如下：点击右上角进入管理状态，可以选择多个感兴趣领域；这些领域会当作搜索的关键字被存储在 DataStore，且**会影响首页推荐的结果**。这个设计有助于用户及时查看其兴趣领域内最新的研究成果。

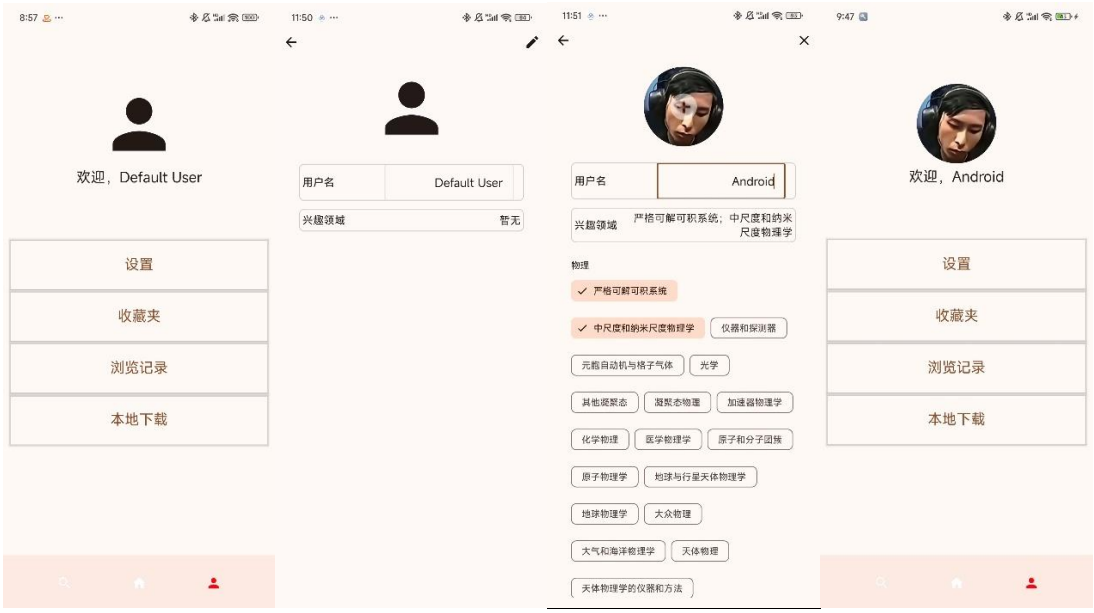

图 2.7 用户系统、设置界面与用户信息修改功能

2. 收藏、下载、浏览记录管理

用户页还提供了收藏、下载、浏览记录管理的功能。这些信息有数据表管理，并支持统一进行管理。

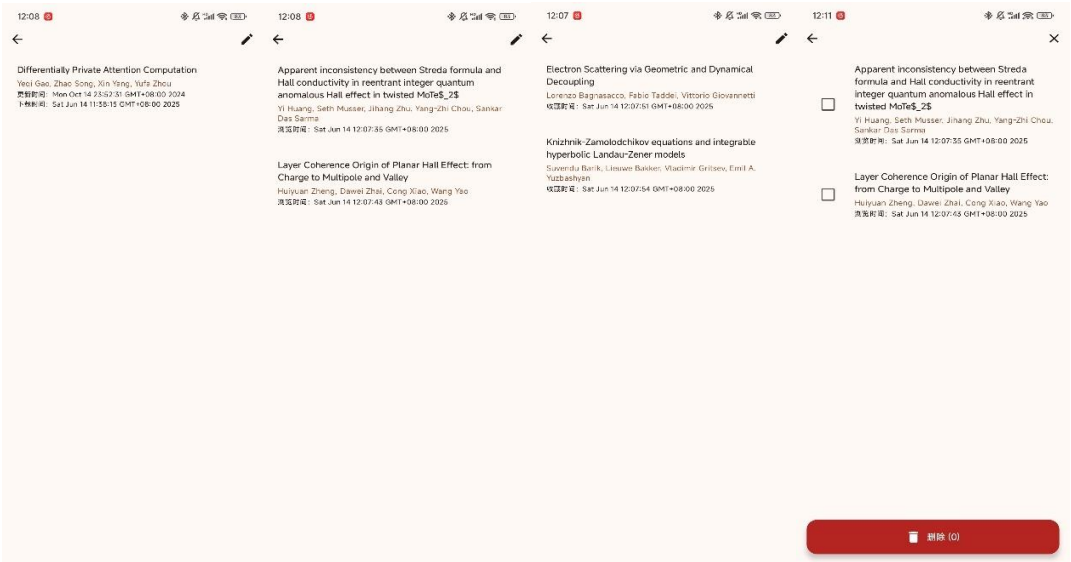

图 2.8 收藏、下载、浏览记录管理功能

收藏、下载可以由详情页、阅读页来添加、删除数据；浏览记录则在每次预览时都会产生，包括从管理页中进入预览。**在管理页点击右上角图标**

进入管理界面，即能够批量删除数据。

点击数据条目，也可以进入预览界面，但不再支持预览页的下载、收藏功能，避免与数据表冲突（见后文）。

d) PDF 阅读器 READER

Academia 包含了简单的 PDF 阅读器，提供**基础的阅览功能，支持上下滑动、缩放**。

顶部栏提供了额外的功能：询问 AI Agent、下载、收藏。下载和收藏的逻辑与详情页完全一致。若在外部预览，三个功能全部可以使用；若在管理页预览，只允许使用 AI Agent 功能。

图 2.9 PDF 阅读器（外部预览、管理页预览、异常）

若希望对 PDF 使用更多的功能，可以通过下载文件再从手机选择其他应用的方式进行。此外，如果**从下载中打开的 PDF 直接由 URI 读取，通常其加载速度会比 URL（预览）更快。**

若加载失败（通常由于 Arxiv 数据库没有及时更新 PDF 文件或者网络问题），PDF 阅读器也会返回结果告知用户。

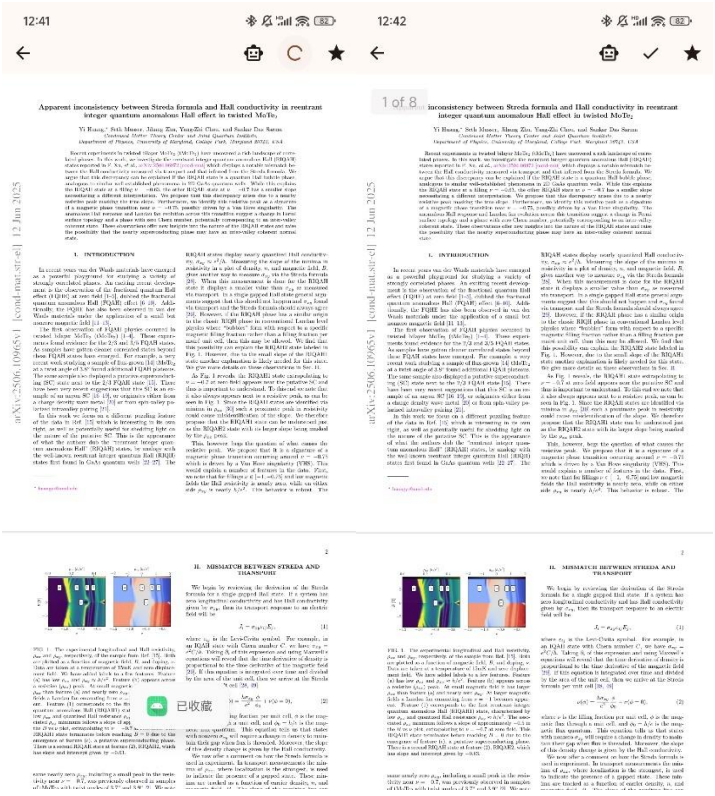

图 2.10 PDF 阅读器内置下载、收藏功能测试

e) AI Agent Chatbox（接入 DeepSeek）

作为智能化应用，Academia 接入了 **DeepSeek**，并提供了聊天界面，能够**向 AI Agent 询问有关领域研究现状、文章内容等多种问题。**聊天内容**采用流式输出，且加入了 Markdown 格式解析**，能够更流畅地展现 AI 输出结果，增强了用户的使用体验。AI Agent 可以从主页、搜索页、阅读器进入。

图 2.11 AI Agent 界面与实机效果

(2) UI 设计方案与人机交互特性

Academia 的 UI 设计以间接明朗、强化体验为原则，旨在为用户提供便捷的论文搜索、管理服务，简洁的论文显示效果，强大的辅助功能。大部分的 UI 实机效果已在前一节展示，下面提供草图并讲述 UI 设计思路。

a) 核心界面 UI 架构设计

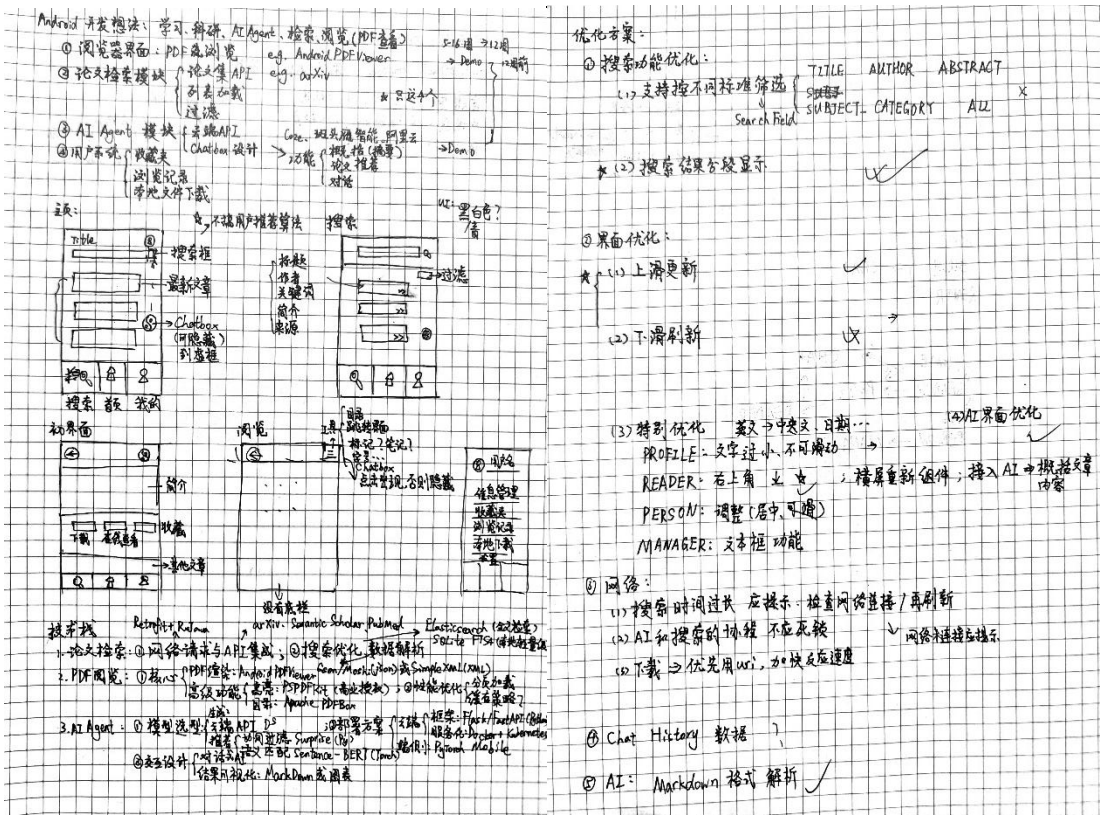

图 2.12 应用整体结构设计与优化方案手稿

在应用开发的整个阶段，从初步设计、测试调试到优化重构，我都对各个部分的功能与 UI 进行了较为详细的构思。**对于核心界面，核心思路是以 Scaffold 框架为中心，向其中添加不同功能的入口。**这样的好处是符合常见应用的设计思路，观感舒适且功能齐全。

头部栏在主页和搜索页中分别为应用头部栏和搜索引擎。主体部分在主页和搜索页中均为自定义组件 PaperScreen，这是一个支持下拉刷新、上滑更新的，基于 LazyColumn 封装并支持懒加载的组件，内部使用卡片来显示论文的信息。底部栏实现页面跳转。浮动按钮 FAB 则被设计为 AI Agent 的入口点，不仅节省了底部栏的空间，而且更易于使用。

b) 管理器与阅读器 UI 设计

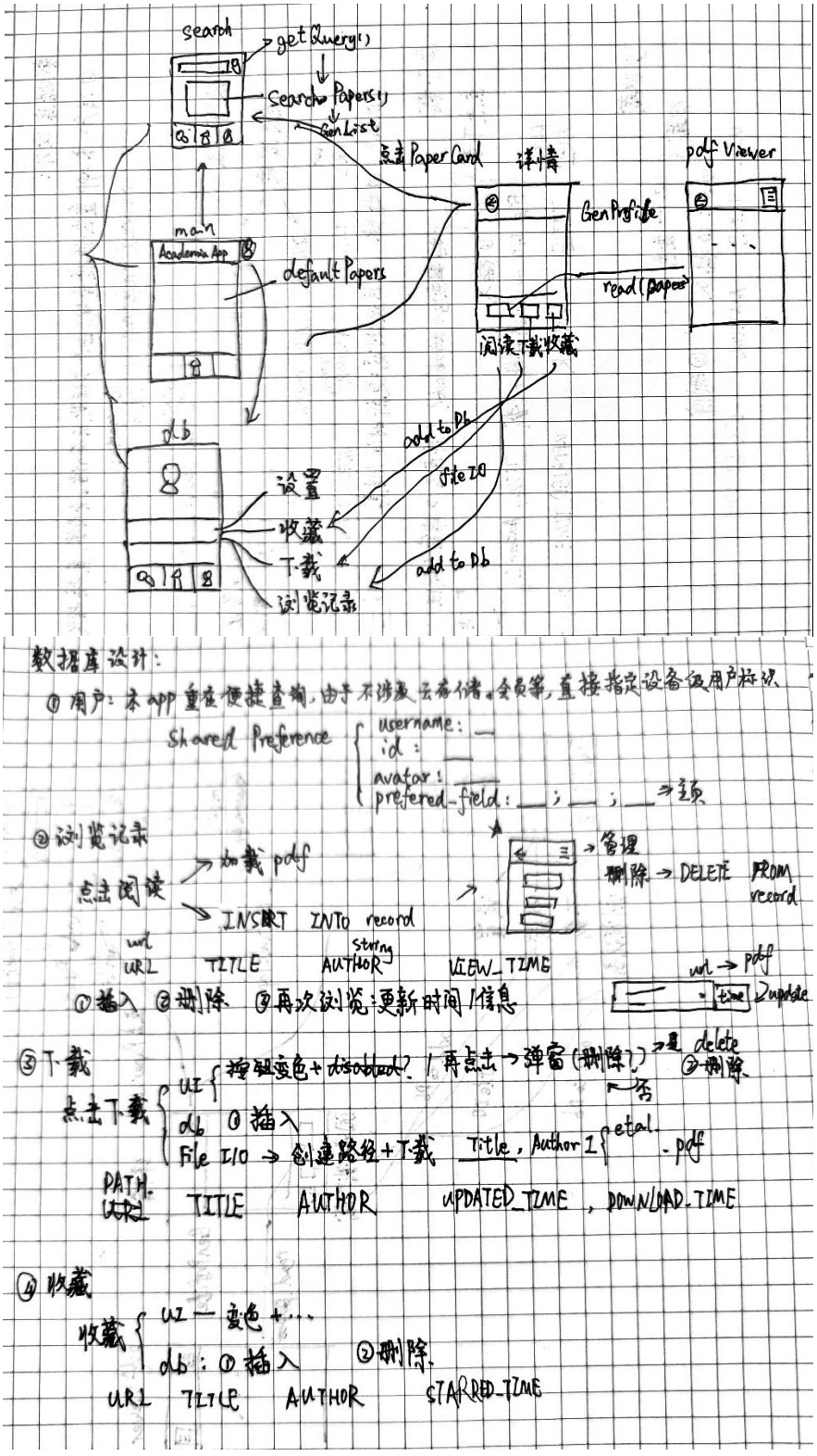

图 2.13 管理器与阅读器 UI 与导航设计、用户数据库设计手稿

由于本应用的定位是轻型的论文搜索和管理应用，管理器与阅读器的设计思路仍然是尽可能地简洁，尤其避免过多的功能填充。

管理器的 UI 设计初衷是为用户系统提供可视化界面，因此**管理器的 UI 是高度模板化且支持重用的**——核心只包括一个返回按键与编辑按键，内部内容可以是数据表的内容，一般被封装进 ManageList（一个由 LazyColumn 封

装的组件），也可以是自定义的内容（即设置界面）；状态管理上又共用一个 ViewModel，提高了页面的统一性。

考虑到市面上已有十分成熟的 PDF 阅读器应用，本 App 内阅读器的 UI 设计根本上是为了提供**预览功能**。考虑到手机用户的习惯，阅读器支持**上下滑动与缩放**。实际上，阅读器的导航功能也与管理器颇为相似。

c) AI Agent Chatbox UI 设计

AI Agent Chatbox 主要是为了方便用户及时提问、积极使用 AI 而设计的。起初的设计思路是放在一个悬浮窗口中使用；然而，这一方案不仅难度很大，而且实际上并不符合手机阅读的习惯，容易因为界面过小反而影响操作。对于 Chatbox 的设计，悬浮窗窗口的设计思路由 AI 给出；考虑到其实现难度和最终效果，我在此基础上将悬浮窗窗口修改为全屏的聊天界面。

AI Agent Chatbox UI 设计的亮点在于 Chatbox 采用聊天室的样式，支持流式输出与 Markdown 解析。这既是主流的 AI 服务的 UI 设计，美观且舒适，而且能够支持用户多次提问放入同一个对话，并且在增强了用户与 AI Agent 的互动感，明显地增强了使用体验。

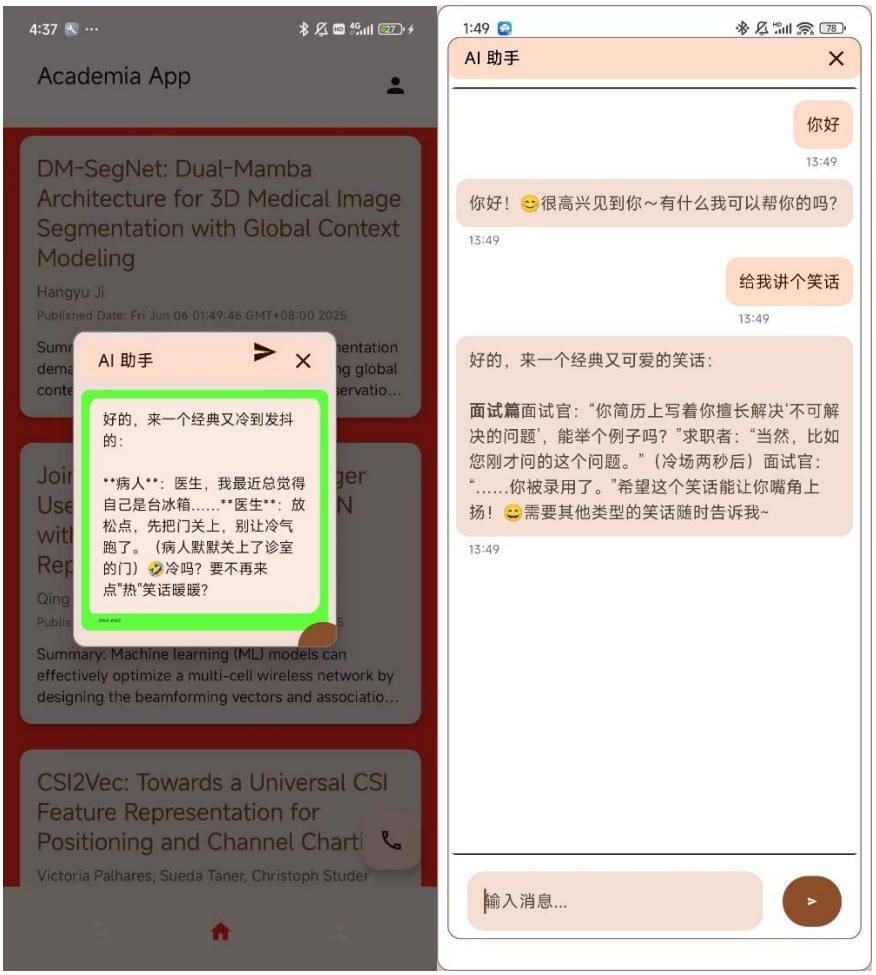

图 2.14 AI Agent Chatbox UI 设计方案对比

3 App 架构设计及技术实现方案

（1） 项目依赖

本应用以 Jetpack Compose 为基础，使用了多个库来编写程序，表格如下：

表 3.1 Academia 项目依赖表

UI 设计	Compose 1.8.0；androidx.compose.material:material-icons-core:1.5.4； androidx.compose.material:material-icons-extended:1.5.4 com.google.accompanist:accompanist-swiperefresh:0.17.0（下拉刷新）
生命周期管理	androidx.lifecycle:lifecycle-viewmodel-compose:2.8.5； androidx.lifecycle:lifecycle-livedata-ktx:2.9.0； androidx.lifecycle:lifecycle-runtime-ktx:2.9.0
数据存取	androidx.datastore:datastore-preferences:1.1.7 io.coil-kt:coil-compose:2.3.0（图片加载框架） androidx.room:room-runtime:2.7.1（数据库）
网络	com.squareup.retrofit2:retrofit:2.9.0； com.squareup.retrofit2:converter-gson:2.9.0 com.squareup.okhttp3:okhttp:4.12.0； com.squareup.okhttp3:logging-interceptor:4.12.0
Arxiv 接口	olegthelilfix:ArxivApiAccess:0.2-RELEASE（第三方库，封装接口） （Github：https://github.com/ResearchPreprintsTools/ArxivApiAccess）
PDF 渲染	io.github.afreakyelf:Pdf-Viewer:2.3.6 （Github：https://github.com/afreakyelf/Pdf-Viewer）
字符解析	javax.xml.stream:stax-api:1.0-2（XML 解析，与 Arxiv 接口相关） com.github.jeziellago:compose-markdown:0.5.7 （Github：https://github.com/jeziellago/compose-markdown）
依赖注入	com.google.dagger:hilt-android:2.56.2 androidx.hilt:hilt-work:1.0.0

（2） 软件包架构/UML 包图

Academia 项目软件包按照功能进行分配，即采取了模块化管理的方式，共分为五个包：core；feature_agent；feature_db；feature_manager；feature_search。软件包 UML 图与顶层架构如下（注有每个包的核心功能）：

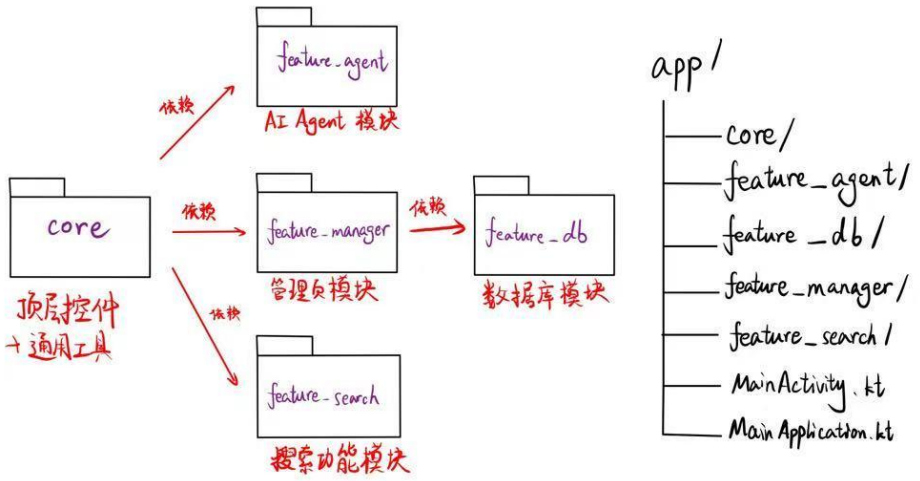

图 3.2 Academia 项目软件包 UML 图与顶层架构图

接下来展示每个软件包的架构以及各自核心类的功能：

core：核心组件包，主要包含顶层应用的状态切换与管理（**本应用的导航就是由状态的切换实现的**）、网络检测与异常处理、顶层 UI 控件、通用工具。其中 AcademiaApp **这个 Composable 函数就是整个项目 UI 的最顶层函数**。

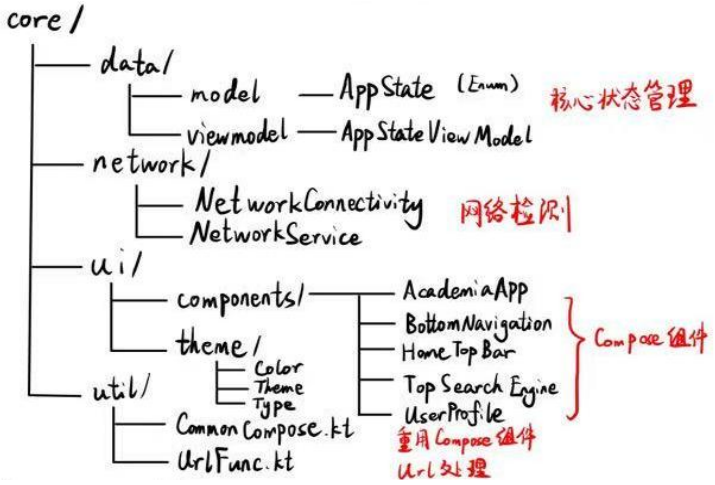

图 3.3 core 包的架构

feature_agent：AI Agent 模块包，主要负责提供 DeepSeek 服务，包含所有涉及 AI Agent 的 UI 设计组件。

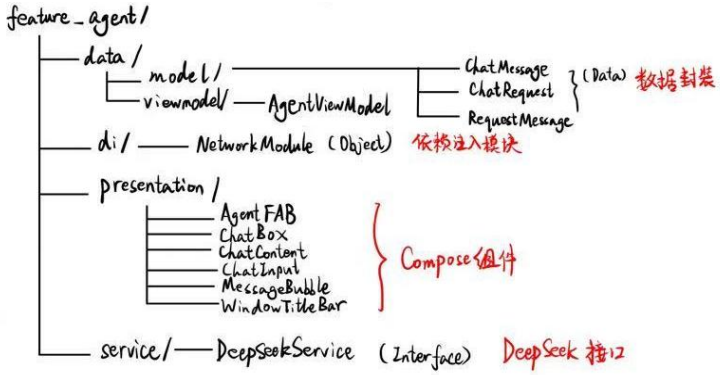

图 3.4 feature_agent 包的架构

feature_db：数据库模块包，负责实现用户系统的数据库存取与管理，包含了收藏、下载、浏览记录的数据封装与数据操纵。

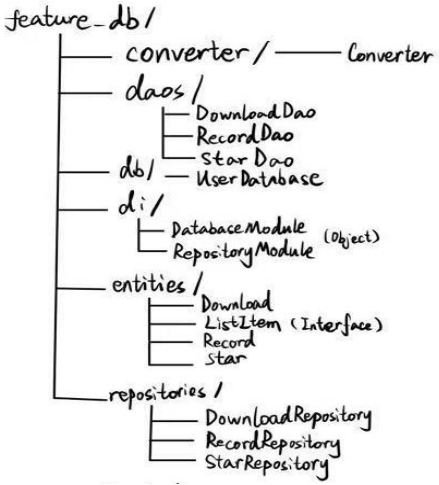

图 3.5 feature_db 包的架构

feature_manager：管理器模块包，主要负责管理页的增删逻辑与用户信息的修改；其中管理器的状态是基于 ManageState 的（即设置、收藏、下载、浏览记录四个状态），下载功能由 PdfDownloadWorker 进行后台操作，SubjectMapper 是一个负责将兴趣领域的字段（API 可访问）与汉语名称进行一一映射的对象。

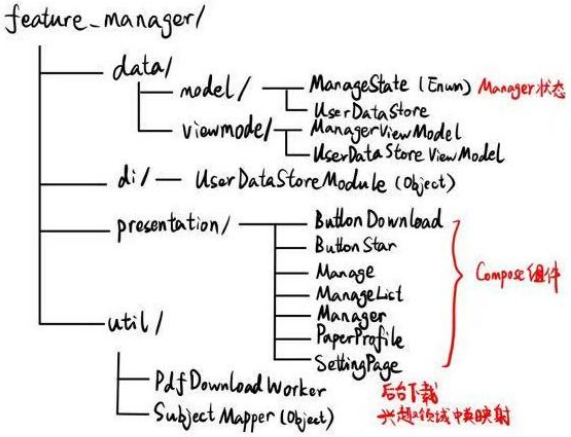

图 3.6 feature_manager 包的架构

feature_search：搜索模块包，主要负责对 Arxiv Api 进行封装并获得结果，包含对返回结果的 UI 设计。

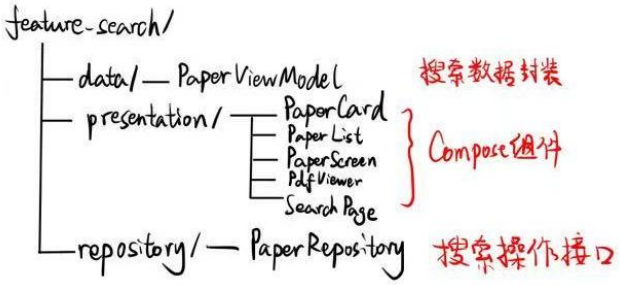

图 3.7 feature_search 包的架构

（3）重要的类与依赖注入关系

a) AppState 与状态转移/导航

前文提到，**Academia 的导航是由状态 AppState 的转移而实现的。**

AppStateViewModel 依赖于枚举类 AppState，并封装了诸多**状态转移的方法**。通常，由 Compose 组件的操作改变 AppStateViewModel 的现有状态和过去状态，来实现状态转移。同时，AppStateViewModel 还存储了当前所选择的论文相关的字段，用于控制 PDF 阅读器所读取的论文、打开方式等，这对于 PDF 阅读器的正确组装和渲染至关重要；每次操作至多选择一篇论文存储进 AppStateViewModel。

下面画出了 UML 类图以及状态转移图：

图 3.8 AppStateViewModel 的 UML 类图与状态转移图

b) **Compose 组件树**

Academia 程序的 UI 全部是基于 Jetpack Compose 编写的，其组件主要是基于 Scaffold 来进行填充的。组件树示意如下图。其中，紫色字迹标注了部分组件所依赖的 ViewModel 状况。

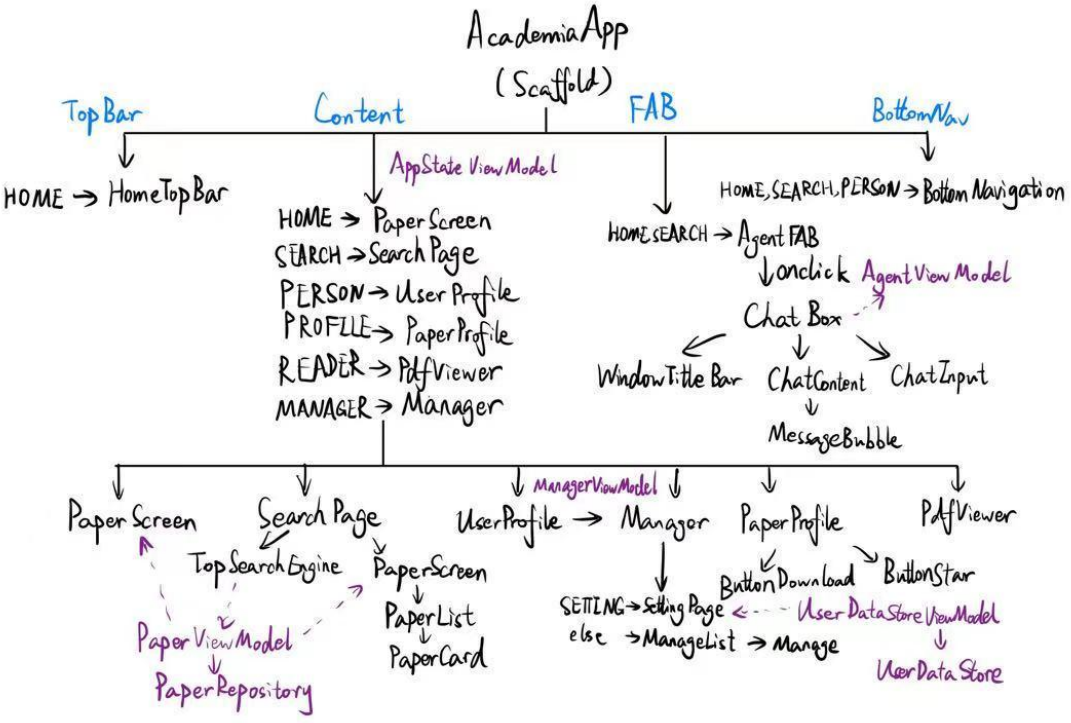

图 3.9 Compose 组件树与 ViewModel 依赖注入图

在这个项目中，一共包含了五个 ViewModel：**AppStateViewModel**（负责应用导航）；**ManagerViewModel**（负责用户数据管理）；**UserDataStoreViewModel**（负责封装用户信息）；**PaperViewModel**（负责搜索并封装结果）；**AgentViewModel**（负责封装 AI Agent 服务与数据）。顶层组件注入了除 UserDataStoreViewModel 其余的 ViewModel 依赖。

本项目的**依赖注入由 Hilt 进行管理**。其他组件视具体功能注入不同依赖，

但为了避免过多依赖注入，通常只在其子组件的顶层组件使用 ViewModel，或者需要操作底层组件改变 ViewModel 时才注入 ViewModel。

c) 数据库实现

在本项目中，针对用户论文数据管理的代码由 Room 进行编写。数据库的实现采用了常用的架构，即实现实体、定义 Dao、创建仓库、封装 ViewModel 的方式。

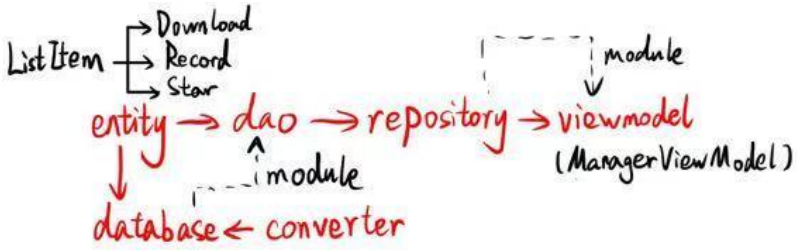

图 3.10 用户论文数据库的实现方式

值得注意的是 entities 中的 ListItem 使用了多态特性，是 Download、Record、Star 实体的共同接口。Converter 是对于特殊类属性的转换器（如 java.util.Date 类）。DatabaseModule 和 RepositoryModule 则是使用 Hilt 管理依赖注入所必需的模块。

d) 管理器与 ManagerViewModel

ManagerViewModel 是管理器的关键。它的主要职能包括：管理 Manager 状态/内部导航；编辑模式开关；接入数据库并存放数据；封装数据库操作，操纵下载、收藏、浏览数据。

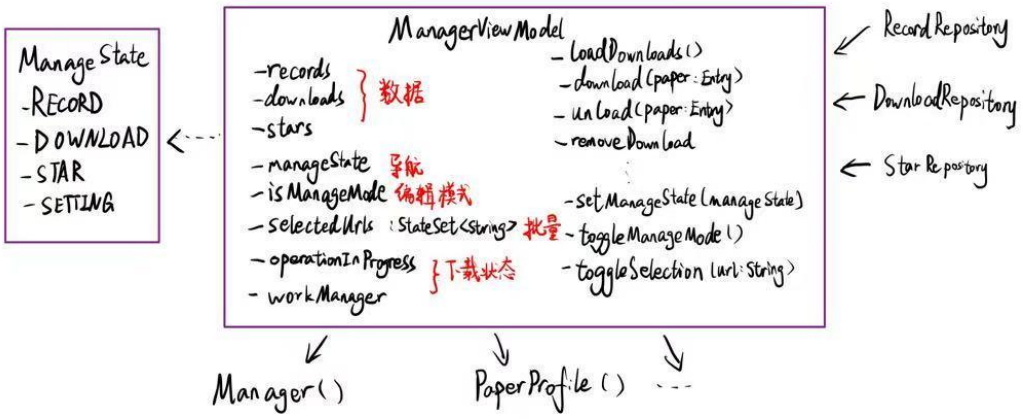

图 3.11 ManagerViewModel 类图与依赖关系

显然，ManagerViewModel 中注入了三个数据库 Repository，体现其在数据库管理上的重要作用。对于 Download、Record、Star 三种数据，每一种都包含了添加、删除、批量删除的操作。而其中下载功能是最难实现的，在第四节会进一步展示。

ManagerViewModel 依赖于枚举类 ManageState，包含四个状态。通过这四个状态的切换，UI 组件完成重组与不同的操作，**实现在 Manager 内部导航（进入设置、下载、浏览记录或收藏）**，其原理与 AppStateViewModel 一致。

ManagerViewModel 还包括了 isManageMode 和 selectedUrls 两个属性，用

于触发编辑状态和实现批量选择/删除。

e) **搜索功能与 PaperViewModel**

PaperViewModel 是搜索功能的关键。它的主要职能包括：封装搜索请求；存储搜索结果；存储加载状态与刷新状态（在下拉刷新功能中很重要）；处理分页逻辑（上滑更新的底层逻辑）。

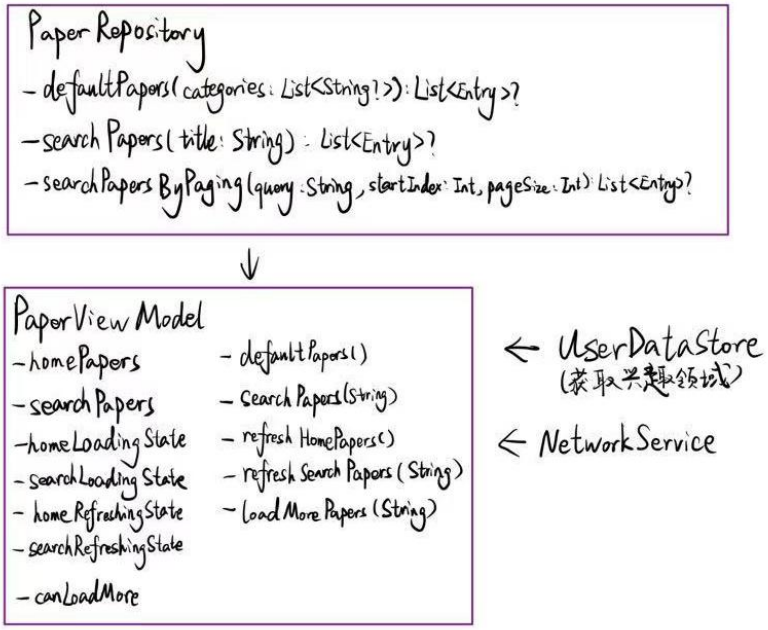

图 3.12 PaperViewModel 类图与依赖关系

PaperRepository 提供了三个方法：

```
suspend fun defaultPapers(categories: List<String?>) :
List<Entry>?
```

——主页推荐搜索，输入为一系列兴趣领域的标识；

```
suspend fun searchPapers(title: String): List<Entry>?
```

——简单搜索，输入为一个查询（不一定要求题目）；

```
suspend fun searchPapersByPaging(query: String, startIndex: Int,
pageSize: Int): List<Entry>?
```

——分页搜索，输入为查询、起始索引、页面大小，用于 loadMorePapers(query: String)函数。

这样，PaperViewModel 就能够通过使用这些方法，来完成搜索、刷新、更新的一系列复杂的操作。PaperViewModel 不仅注入了 PaperRepository，还注入了 UserDataStore、NetworkService，在第四节会提到这些设计的含义。

4 技术亮点、技术难点及其解决方案

（1）同类产品比较：Academia、知网 App、Arxiv 网站在线搜索

进行设计时，我搜集了市面上的论文管理应用，并探索了 Arxiv 网站，分析了设计的目标、用户画像、优势劣势。应用完成后，进行横向比对，比较结果如下：

表 4.1 Academia、知网 App、Arxiv 网站的产品比较与分析

	知网 App	Arxiv 在线搜索	Academia
存储空间	较大（250MB）	不占据	较小（79MB+）
数据来源	广泛（知网）	精确（Arxiv）	精确（Arxiv）
应用功能	很多	仅搜索	搜索与管理
使用体验	复杂	简单但不便捷	简单便捷
用户系统	云端存储，难以实现	无用户系统	本地存储，本机级别
用户画像	国内学术工作者	理工科学生	理工科学生
开发成本	大	不合要求	适中

（2）技术亮点

a） 用户偏好与主页推荐功能

在应用设计中，符合用户习惯的设计往往能够增加交互体验。Academia 为了令用户信息能够与搜索结果联动，支持用户在设置页挑选其兴趣领域，**兴趣领域将直接影响主页的推荐结果**。

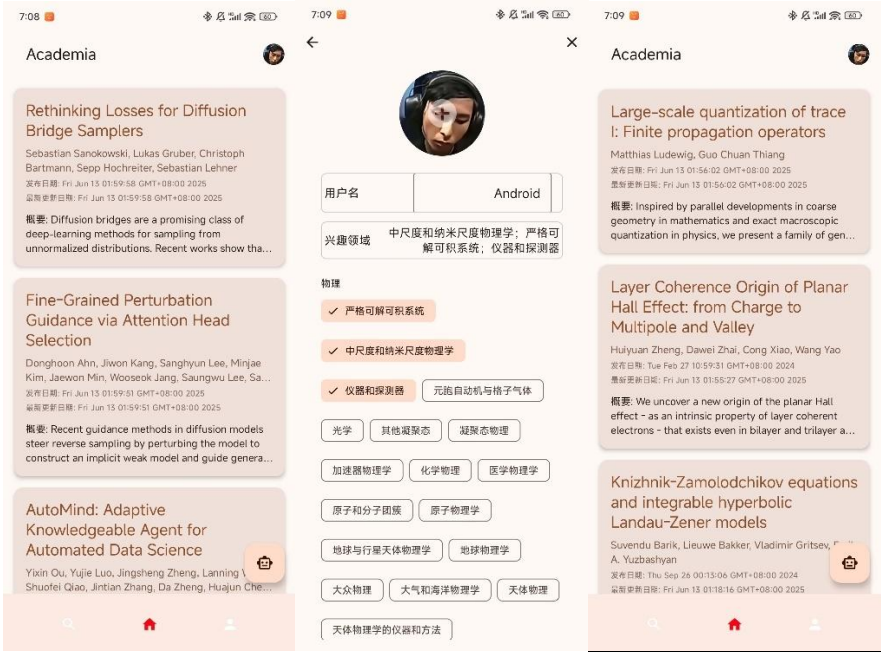

图 4.2 兴趣领域推荐实际效果（前者未选择，后者选择物理的子领域）

实现方案：

UserDataStore 中获取兴趣领域，是将一个 String 转换为一个 Flow：

```
val preferredField: Flow<List<String>> =
    context.userDataStore.data.map { preferences ->
        val field = preferences[UserPreferencesKeys.PREFERRED_FIELD] ?: ""
```

```
        if(field.isEmpty()) emptyList() else field.split(",").map
{ it.trim() }
    }
```

在 PaperViewModel 中，将这个流对象映射并成为 defaultPapers 的参数；defaultPapers 将这些类别拆分并依次封装进搜索请求。在带有网络超时的上下文中，获取用户偏好的兴趣 preferredField，并监听其变化。每当分类变化，就取消之前的请求，并发起新的请求获取默认论文列表。然后取第一个成功完成的请求结果（即某一次分类对应的论文列表），并返回该结果以及网络请求状态。

```
@OptIn(ExperimentalCoroutinesApi::class)
suspend fun defaultPapers() {
    _homeLoadingState.value = true
    val (result, state) = networkService.withNetworkTimeout {
        userDataStore.preferredField
            .flatMapLatest { categories ->
                flowOf(paperRepository.defaultPapers(categories))
            }.first()
    }
    if (result == null) {
        _homePapers.value = emptyList<Entry>()
    } else {
        _homePapers.value = result
    }
    _homeLoadingState.value = false
}
```

b) AI Agent Chatbox 流式返回结果与 UI 设计

在 AI Agent Chatbox 的 UI 设计上，我借鉴了主流 AI 平台的显示方式（如 DeepSeek、ChatGPT）。**流式返回结果**避免因为某些异常导致的阻塞，也可以让用户尽可能快地接收到 AI 返回的信息；从观感上，这种方式也更符合人的偏好，具有趣味性和交互性，降低了网络请求略长带来的不适感。

之所以这样做是可行的，是因为 **DeepSeek 接口的返回结果本身就支持流式传输**。因此，设计这种效果需要将在接口中允许流式传输，并且在 ViewModel 中编写能够及时处理 Json 的解析器和处理 Markdown 的组件。对于前者，AgentViewModel 提供了诸多处理方法，包括：

```
    private fun updateMessage(id: String, newText: String, isError:
Boolean)
    private fun markMessageComplete(id: String)
    private fun parseChunkContent(json: String): String?
```

这些方法能够及时地对既有信息进行处理，具体代码在 AgentViewModel.kt 中。至于 Markdown 组件，我选择使用第三方库进行开发，即一个支持 Compose 显示的 Markdown 解析库，MessageBubble 中组件 MarkdownText 即是库提供的。

图 4.3 AI Agent Chatbox 流式输出效果

实现方案：

将 DeepSeekService 接口按如下形式编写：

```
interface DeepSeekService {
    @Headers("Authorization: Bearer $DEEPSEEK_API_KEY", "Content-Type:
application/json")
    @POST("chat/completions")
    @Streaming
    suspend fun sendRequest(
        @Body request: ChatRequest
    ): Response<ResponseBody>
}
```

然后在 AgentViewModel 中使用流式结果(在方法 sendMessage(String)中)，值得注意的是 delay()语句用来间隔每次输出的结果，否则即使流式输出也难以被人眼所察觉：

```
val source = response.body()?.source()
source?.use { source ->
    val buffer = StringBuilder()
    while (!source.exhausted()) {
        val line = source.readUtf8Line() ?: continue
        when {
            line.startsWith("data:") && line != "data: [DONE]" -> {
                val json = line.substringAfter("data:").trim()
                val content = parseChunkContent(json)
                if (content != null) {
```

```
                buffer.append(content)
                updateMessage(aiMessage.id, buffer.toString(), false)
                delay(25)
            }
        }
        line == "data: [DONE]" -> {
            markMessageComplete(aiMessage.id)
            break
        }
    }
}
}
```

c) PDF 预览功能的优化

在最初的设计中，PDF 阅读器是独立于其他部分的一个模块。但是，考虑到**增强应用中各组件之间的联动性**可以降低使用门槛，增强交互特性，我最终选择在 PDF 阅读器右上角添加了三个按键，分别表示 AI Agent、**下载**、**收藏**。这些组件的效果与在应用其他位置的效果完全一致，因此这里不再赘述其实现。

图 4.4 PDF 预览联动优化示意

PDF 预览功能还有一个优化：当在**管理页打开下载文件时，PDF 阅读器直接从下载文件的 URI 读取 PDF，加载速度大幅度加快**，无需网络连接，并且契合了文件下载的目的。

实现方案：

调用第三方库的 PDF 渲染函数，根据实际填入的参数，如下面的 source

属性（在 PdfViewer.kt 中）：

```
PdfRendererViewCompose(
    source =
        when (selectedPaperType) {
            "url" -> PdfSource.Remote(selectedPaperUrl)
            "uri" -> PdfSource.LocalUri(selectedPaperUrl.toUri())
            else -> PdfSource.LocalUri(selectedPaperUrl.toUri())
        },
    lifecycleOwner = LocalLifecycleOwner.current,
    headers = HeaderData(mapOf("Authorization" to "123456789")),
    zoomListener = object : PdfRendererView.ZoomListener {
        override fun onZoomChanged(isZoomedIn: Boolean, scale: Float) {
            Log.i("PDF Zoom", "Zoomed in: $isZoomedIn, Scale: $scale")
        }
    },
    statusCallBack = pdfStatusCallBack
)
```

（3）技术难点与解决方案

a）搜索页面下拉刷新、上滑更新功能

搜索页面下拉刷新、上滑更新，既是为了强化交互体验，也是为了允许在没有及时返回搜索结果时可以提供重试的方法、允许返回更全面的结果。

最大困难在于代码重构：起初只有 PaperList 的代码，要想**实现下拉刷新、上滑更新，不仅需要添加若干参数——刷新状态、更多数据标识、刷新方法、加载方法；还需要特殊的判断与动画。**

解决方案：

针对下拉刷新的功能，我经过调查资料，发现一个经常被使用的库——SwipeRefresh，只要将其他代码封装进 SwipeRefresh 组件，即可实现下拉刷新。为此，我又设计了 PaperScreen，加 SwipeRefresh 和 PaperList 进一步包装。

PaperScreen 的参数如下，其中 isRefreshing、hasMore 都由 PaperViewModel 控制，hasMore 的更新在 loadMorePapers(String) 中：

```
fun PaperScreen(
    papers: List<Entry>,
    isLoading: Boolean,
    isRefreshing: Boolean,
    hasMore: Boolean, // 是否有更多数据的标志
    onRefresh: () -> Unit,
    onLoadMore: () -> Unit = {},
    listState: LazyListState,
    onClick: (Entry) -> Unit = {}
)
```

其中实现下拉刷新的方法比较简单，如下所示：

```
SwipeRefresh(
    state = swipeRefreshState,
    onRefresh = onRefresh,
    modifier = Modifier.fillMaxSize()
) {
    PaperList(
        papers = papers,
        isLoading = isLoading,
        hasMore = hasMore,
        isRefreshing = isRefreshing,
        listState = listState,
        onClick = onClick
    )
}
```

但是值得注意的是：由于 onRefresh 一般使用了 PaperViewModel 的其他函数，不当的处理可能会导致 isRefreshing 没有正常更新。比如下面这段代码：

```
fun refreshHomePapers() {
    if (homeRefreshingState.value) return
    _homeRefreshingState.value = true
    viewModelScope.launch {
        Log.i("Refresh", _homeRefreshingState.value.toString())
        try {
            defaultPapers()
        } catch (e: Exception) {
            Log.e("Paper View Model", "Refresh Exception")
        }
        Log.i("Paper ViewModel", "Home Refreshed")
        _homeRefreshingState.value = false
    }
}
```

在初期的代码，defaultPapers()并不是一个挂起函数，而是一个内部又由 viewModelScope 包装的函数。在这种情况下，refreshHomePapers()更新 _homeRefreshingState 和 defaultPapers()是一系列异步的操作，也就是说处理主页推荐的操作和改变刷新状态并不同步，导致逻辑错误。最终的解决方案是将 defaultPapers()变成挂起函数，并将刷新状态更新放入同一个协程作用域，从而可以使这部分代码顺序执行。

至于上滑更新的代码，关键是 PaperScreen 中的一段代码：

```
val shouldLoadMore by remember(listState, papers, isLoading, hasMore,
isRefreshing) {
```

```
    derivedStateOf {
        val layoutInfo = listState.layoutInfo
        // 基本条件检查
        if (papers.isEmpty() || isLoading || isRefreshing || !hasMore) {
            Log.i("LoadMoreCheck", "条件不满足: papersEmpty=${papers.isEmpty()},
isLoading=$isLoading, isRefreshing=$isRefreshing, hasMore=$hasMore")
            return@derivedStateOf false
        }
        val totalItems = layoutInfo.totalItemsCount
        if (totalItems == 0) {
            Log.i("LoadMoreCheck", "总项目数为 0")
            return@derivedStateOf false
        }
        val lastVisibleItem = layoutInfo.visibleItemsInfo.lastOrNull() ?:
return@derivedStateOf false
        // 检查是否滚动到最后一个项目
        val shouldTrigger = lastVisibleItem.index >= totalItems - 1
        Log.i("LoadMoreCheck", "检查触发: lastIndex=${lastVisibleItem.index},
totalItems=$totalItems, threshold=${totalItems},
shouldTrigger=$shouldTrigger")
        shouldTrigger
    }
}

// 监听加载更多触发状态
LaunchedEffect(shouldLoadMore) {
    if (shouldLoadMore) {
        val currentTime = System.currentTimeMillis()
        // 添加 1500ms 防抖，防止连续触发
        if (currentTime - lastLoadTime > 1500L) {
            Log.i("Paper List", "Triggering load more")
            lastLoadTime = currentTime
            onLoadMore()
        }
    }
}
```

shouldLoadMore 是一个联合的 Boolean 状态，它的含义是：在当前界面拥有搜索结果、不处于加载与刷新状态、且保证有足够的结果返回时，检查当前 LazyList 所处于的索引，当索引接近最后一条时（为此 PaperList 也做了一些修改,列表下方加入了额外的空行与进度提示),启动刷新流程。PaperScreen 监听这一状态，当发生时执行 onLoadMore 操作。通过这种方法，实现了一个相对简易的上滑更新操作。

最终的细节效果如下：

图 4.5 下拉刷新、上滑更新的最终效果

b) **网络检测与超时提示**

Academia 应用依赖于 Arxiv API 第三方接口，而这个接口稳定性并不理想（通常在某些时段响应很快，某些时段响应很慢甚至异常）。这并非是 Academia 应用的问题，然而面对这种情况，**必须要及时告知用户，并且能够允许用户再次尝试**（这由刷新、更新功能控制）。

解决方案：

解决这个问题的方案是一个额外的网络检测类 NetworkService，其中最核心的代码是 withNetworkTimeout，如下：

```
// 带超时处理的网络请求
suspend fun <T> withNetworkTimeout(
    block: suspend () -> T
): Pair<T?, NetworkState> {
    Log.i("Network Service", "Available: " + checkNetwork())
    if (!checkNetwork()) {
        showToast(context, "网络不可用，请检查连接")
        return Pair(null, NetworkState.Unavailable)
    }
    return try {
        // 使用 coroutineScope 来管理子协程
        coroutineScope {
            // 创建 10 秒超时警告的 Channel
            val warningChannel = Channel<Unit>(Channel.RENDEZVOUS)
            // 启动 10 秒警告任务
            val warningJob = launch {
                delay(10000L)
```

```
                if (isActive) {
                    warningChannel.send(Unit)
                    Log.i("Network Service", "Warning: 10s")
                }
            }
            val deferredResult = async { block() }
            launch {
                for (unit in warningChannel) {
                    Log.i("Network Service", "Received 10s warning
notification.")
                    showToast(context, "网络响应缓慢...")
                }
            }
            // 使用 withTimeoutOrNull 来处理 30 秒超时，等待 deferredResult 完成
            val timeoutResult = withTimeoutOrNull(30000L) { // 30 秒最终超时
                val result = deferredResult.await() // 等待实际的网络请求完成
                // 如果请求在 30 秒内完成，取消警告任务和通知通道
                warningJob.cancel()
                warningChannel.close() // 关闭通道，停止监听
                Log.i("Network Service", "Job Finished")
                Pair(result, NetworkState.Success)
            }
            // 任务完成或超时
            warningJob.cancel() // 确保警告任务被取消
            warningChannel.close() // 确保通知通道被关闭
            when (timeoutResult) {
                null -> {
                    // 只有当 withTimeoutOrNull 自身超时，而 select 还没有返回时，
timeoutResult 才是 null
                    deferredResult.cancel()
                    Log.i("Network Service", "TimeOut: 30s")
                    showToast(context, "请求超时，请重试")
                    Pair(null, NetworkState.TimeoutCancel)
                }
                else -> {
                    // select 表达式返回了一个 Pair
                    Log.i("Network Service", "Timeout Result != null")
                    timeoutResult
                }
            }
        }
    } catch (e: Exception) {
        Log.e("Network Service", e.message.toString())
        showToast(context, "出现异常，请重试")
```

```
        Pair(null, NetworkState.Error)
    }
}
```

这段代码注释较为详实，它的含义是：在网络连接状态下，获取一个任务，在执行这个任务同时执行一个 10s 的报警任务，对于这两个任务，若前者先执行完，直接返回结果，若后者先执行完（即处理时间超过 10s），先提示“网络响应缓慢…”；随后，若 30s 内前者还没有执行完，提示“请求超时，请重试”；若中途捕获其他异常，提示“出现异常，请重试”。

效果图已在图 2.3 中展示。

因此，在 PaperViewModel 中需要注入 NetworkService 来调用这个检测方法。使用方法比较简单，以下代码为例：

```
val (papers, state) = networkService.withNetworkTimeout {
    paperRepository.searchPapers(query)
}
```

c) **后台下载功能与下载按钮的状态同步**

下载功能可以异步处理，这样可以避免线程阻塞并支持其他 UI 操作。后台下载的代码在 PdfDownloadWorker 中，这是一个由 CoroutineWorker 派生的类，支持后台处理的工作。这段代码主要由 AI 进行生成。

然而，实现后台下载功能还需要与下载按钮状态实现同步。在初期的设计中，下载按钮只有两个状态：下载和已下载，且 UI 更新与后台工作独立。这样很有可能导致诸多异常，如重复下载、指定删除未下载的文件。

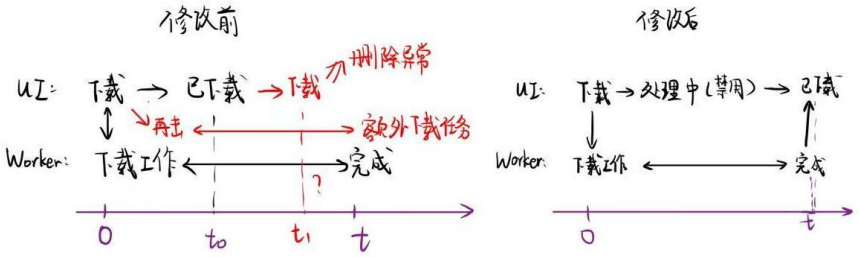

图 4.6 下载异常示意与解决方案示意

解决方案：

为解决这个问题，我选择引入第三个状态——处理中，期间禁用按键。

ManageViewModel 中放置了一个字段用于管理当前正在操作的下载按键。这是一个 StateFlow<Set<String>>类型的变量。它的含义是：若当前正在下载这个文件，将它的 URL（唯一标识）加入集合，此时这个文件对应的按键将处于 isProcessing = true 的状态，禁用按键。若完成下载，由 PdfDownloadWorker 告知，并将其移除集合，isProcessing = false，接触禁用且 UI 能够及时变化。

```
fun download(paper: Entry) {
    val url = convertArxivUrl(paper.id)
    if (_operationsInProgress.value.contains(url)) return
    // 添加到进行中操作
```

```
    _operationsInProgress.update { it + url }
    val title = modifyTitle(paper.title)
    val author = paper.author.joinToString { author -> author.name }
    val updatedTime = paper.updated
    // 检查是否已下载
    Log.i("DOWNLOAD", "Start to download")
    viewModelScope.launch {
        try {
            if (downloadRepository.downloadExists(url)) {
                _downloadStatus.value = DownloadStatus.Error("该论文已下载")
                return@launch
            }
            // 启动下载任务
            startDownload(url, title, author, updatedTime)
        } catch (e: Exception) {
            _operationsInProgress.update { it - url }
        }
    }
}
```

通过这种方式，很容易实现后台工作与 UI 的同步，效果见图 2.6。

5 简要开发过程

5 月 8 号　构思应用主体架构、设计应用核心功能

5 月 14 号　初步完成应用框架的 UI 构建，包括主页、搜索页、用户页

5 月 18 号　搜集 Arxiv API 的接入库，调试论文搜索功能

5 月 20 号　集中学习 Android 开发的技术，包括 Room、Hilt、ViewModel、Coroutine

5 月 22 号　规范化应用包架构，设计管理系统、人机交互功能等

5 月 23 号　完成论文显示 UI 设计,初步建立核心 ViewModel 与 feature_search 包

5 月 25 号　完成从 URL 获取 PDF 与 PDF 预览的功能，接入 PDF 预览库

5 月 28 号　设计用户与论文管理系统，基于 Room 构建论文管理数据库，基于 DataStore 实现本机级别的用户系统，搭建 feature_db 包

5 月 31 号　完成管理界面的 UI 与功能设计，搭建 feature_manager 包

6 月 3 号　实现后台下载 PDF 文件的功能

6 月 5 号　完善 PDF 预览功能，支持从预览界面进行下载、收藏管理

6 月 7 号　接入 DeepSeek，构建 AI Agent 与 feature_agent 包，创建 Chatbox UI；至此应用功能基本完成，进入优化与最终测试阶段

6 月 9 号　搜索、下载等功能的健壮性优化

6 月 10 号　UI 优化

6 月 11 号　增加网络连接、超时等提醒，完善应用健壮性

6 月 12 号　对应用进行最终集成测试

6 月 14 号　程序开发工作完毕，编写及整理文档

6 学习感悟及对本课程的建议

相较于上学期 Java 课的情形，在这次 Android 课程学习上，我认为我有了很大的进步。

首先，我认为我的进步的根本原因是：通过一学年的学习、阅读和实践，我进一步了解了软件工程的特点，积累了许多经验，提升了系统设计的意识，从而能够更好地规划功能。比如上学期我还是跟着文档做了一个 Java FX 文件管理系统，并且由于设计平庸，加之构建一个强健的文件管理系统对技术要求实际相当高，最后在紧迫的时间里做出了一个不那么满意的作业。而这学期 Android 开发中，我选择了更合理的规划方法，即**先花较长的时间确定好一个方向，分析目标用户、市面需求，从其他软件设计借鉴与反思，再动手开始实操。**正因为这样，我认为这次设计不但有较多的亮点，而且在设计的时候也较上学期更加从容（也有一个原因是本学期课业压力没有上学期大）。我还意识到：对于这样的课程设计，一个大而健壮的软件设计几乎不可能由一个人完成，更重要的是**摸清自己的软件的优势，扬长避短，在用户交互、核心功能上做得更精细，而不是纠结于设计的绝对完善。**《人月神话》（The Mythical Man-Month, Frederick P. Brooks, Jr.）提出软件开发的进度分配：1/3 用于计划，1/6 用于编码，1/4 用于构建测试，1/4 用于系统测试。尽管它所说的软件开发和今天常见的软件开发相差甚远，但这段提示还是给人启发——只有精湛的设计才会产生精妙的产品；编码只不过是整个设计流程的一小部分，绝不能将它当作开发的全部。

此外，在这学期，我能够更加熟练地运用 AI 工具进行辅助。得益于 AI 的迅速发展，尤其是国产 AI 的发展（DeepSeek），我能够使用更先进的工具辅助我的开发，这很大程度上缩短了我的开发时间，也使得完成一个相对健全的应用成为可能。我主要在以下几种情况中使用 AI：（1）**项目架构设计**——当我产生一个不成熟的想法，我会先向 AI 询问这个想法的可行性，比如询问市面上是否存在已有的类似的设计，询问完成设计的实现路径和所需的库（从某种意义上来说，AI 比我更有“系统设计经验”）；此外，我还会让 AI 帮我重新组织包、类的结构，从而帮我厘清设计的思路，探索模块化编程的方法。（2）**代码重构**——在这次开发中，我一般是先自己编写一段基础的代码进行测试，等到需要开发一些进阶的功能，我会询问可行的技术方案，AI 也能够给我一个相对简单的实施方案（比如应用中需要一个后台下载功能，AI 告知我可以继承协程中的 CoroutineWorker 来实现）；不仅如此，对于一个只有有限学习时间的初学者来说，AI 能够极大程度地辅助我的学习，向我讲解核心代码的含义（并且由于 Kotlin 更加简洁的特性，软件开发的难度、技术门槛、学习成本也大幅降低了）。（3）**代码测试**——单元测试与系统测试其实是软件开发的重中之重，也是最困难的部分，既因为人的设计一定是存在欠缺的，又因为错误会像多米诺骨牌一样传递给其他组件；面对报错，AI 能够印证我对错误的猜想，还可以帮我找出错误，提出优化方案，使得每个组件都相对地健壮，因此减少了系统最终测试的复杂度。不过，AI 是一个降低开发成本的工具，但对于项目设计其本身还是需要开发者的经验与想法，最终的决策者也是开发者——我；所以，**学会使用 AI 就是“掌握了先进生产力”，就能**

够更好地创造。项目中，我也尝试集成了云端的 AI 服务，尝试接触了智能化应用的开发。

尽管如此，这次大作业的设计还是出现了很多问题，我认为还是应该在这里总结一番。

第一，项目设计上还是存在盲目追求应用功能齐全，又没有足够的时间来开发，导致最终未能实现。这次应用的项目设计的核心功能都能够正常完成，这是比较幸运的。实际上我还设计了许多比较细节的、可以增强用户交互体验的功能，比如：搜索筛选（学科领域、论文作者搜索等）、完善的 PDF 阅读体验（高亮、标注）、自定义的 AI 服务（能够根据 PDF 预览内容总结文章）。然而，因为核心功能相对还是比较多（从项目代码数量也可见一斑），尽管此次项目时间分配相对比较合理，但**由于设计追求功能齐全，许多这些有趣的设计都没有额外的时间来实现了，削减了交互体验，对于软件开发是不利的。**

第二，我在项目设计上仍然缺乏经验，项目开发中发现原先的设计并不是很合适，但是“积重难返”，导致代码重构上出现了困难。实际上，我认为最能体现这个问题的是我使用了变换 AppState 了重组 Compose 组件，但是对于这种相对比较大的应用，我认为更好的方法是使用 Navigation，但等我意识到了这点已经来不及修改了。不过，使用 AppState 也可以实现功能，只是后续增加新的功能会有一些限制，比如在搭建 Agent Chatbox 时就因为其难以重构而没有加入 AppState，而是根据 ViewModel 的字段放在界面之上。再比如上滑更新存在更好的解决方案，如 Paging3，最终也没有来得及重构。此外，包结构的组织也因为这个问题显得有些混乱。这也让我理解了所谓“屎山代码”是如何产生的：**软件开发在不同阶段的工作量成倍提升，不同阶段的代码堆积在这个系统中，牵一发而动全身，在有限时间内重构代码是极其困难的。**

第三，项目开发中，我经常遇到由于设计没有考虑周全而产生的错误。比如我在最后一周测试时发现，Arxiv API 接口有时不太稳定，且有时会直接取消请求；如果我不进行网络检测和网络优化，整个界面会卡住且无法通过刷新来重新获取查询请求，或者因为无网络连接的查询而闪退。因此，最后我加入了网络超时处理；而即便这样，仍然没有解决 Arxiv API 接口不稳的问题，一方面是 Arxiv 提供的接口访问本身就会因为地域等原因产生影响，另一方面是我使用的是第三方的、适合 Kotlin 的第三方接口库，延展性相对较差。开发过程中，这种情况应该是常见的。不过，我认为像我这种**相对常见的设计，还是应当更多地考虑特殊情况，展开边缘测试等等。**再比如开发中还发生了异步的后台下载与 UI 更新不同步导致的错误下载逻辑的异常。在这些问题上，我大多通过补丁的方式增强了程序的健壮性。

总而言之，这次的课程设计令我受益匪浅；与 Java 课程相对基础、强调用特定语言编程不同，Android 开发这门课更强调应用多种不同的方法，博采众长地搭建出自己头脑中理想的应用，或许更接近实际的软件开发的流程。过程中，我不仅了解了 Kotlin 和函数编程的方法，了解了 Gradle 管理项目依赖的方法，更学会了搜集一切有利于项目开发的材料，又快又好地开发一个有益的应用。课程

设置上我认为比较完美：学习难度适当，重视实际开发中常见的方法，强调产业智能化趋势……。希望这门课程继续开下去，始终紧跟时代步伐！最后感谢所有帮助我测试应用的同学，感谢这门课程的讲师金老师！