德国与欧洲资助项目智能发现系统完整教程

第一章：项目导论与需求分析

欢迎来到《AI赋能：德国与欧洲资助项目智能发现系统开发指南》。本章将为您深入剖析项目的背景、目标、核心功能，并概览整个系统的架构和所选用的技术栈。这将帮助您建立对项目的整体认知，为后续的学习和开发打下坚实基础。

1.1 项目背景、目标与价值

1.1.1 项目缘起与挑战

在当今全球化的经济环境中，获取资金支持对于企业、研究机构乃至初创公司的发展至关重要。特别是在德国和欧洲这样经济发达且创新活跃的地区，存在着数量庞大且种类繁多的资助项目。然而，这些资助信息往往分散在各个政府部门、欧盟机构以及行业协会的网站上，其内容和申请条件又在不断动态变化。

正如 Legal-Pythia LLP 项目描述中所指出的，"该领域最大的挑战在于德国和欧洲存在大量的资助计划，并且这些计划在不断变化。即使缩小到特定行业，仍然存在大量相关的资助机会，咨询团队分析和评估这些机会非常耗时且占用大量资源。"

传统的人工筛选方式面临着严峻的挑战：

- 信息过载与筛选困难：面对海量信息，人工难以快速、准确地找到最匹配的资助项目。
- 耗时耗力：研究和评估每个潜在项目都需要投入大量时间和人力成本。
- 信息滞后性：新的资助项目不断涌现，旧的项目可能调整或截止，人工跟踪难以保证信息的时效性。
- 匹配精准度不足：人工判断可能存在主观偏差，导致错失良机或申请不相关的项目。

这些痛点不仅降低了企业和机构获取资助的效率，也给提供资助咨询服务的团队带来了巨大的工作压力。因此，利用人工智能技术来革新这一领域，实现资助信息的智能发现与匹配，具有重要的现实意义。

1.1.2 核心目标：打造AI驱动的资助项目雷达

本项目的核心目标是借鉴 Legal-Pythia LLP 项目的愿景，开发一个由人工智能（AI）支持的搜索工具，旨在自动化识别、筛选并结构化呈现德国和欧洲范围内最新的、最合适的资助

项目。具体目标分解如下：

- 自动化信息获取：通过网络爬虫技术，定期自动从指定的官方数据库和网站（如 ptj.de, eu-startups.com 等）抓取最新的资助项目和投资者相关数据。
- 定制化搜索配置：允许用户根据自身特点（如行业、公司成立年限、营业额、员工数量等）创建个性化的搜索配置文件，以便系统进行精准匹配。
- **AI**智能匹配与筛选：利用自然语言处理（NLP）和机器学习（ML）技术，分析资助项目的文本描述和申请条件，并与用户画像进行智能匹配，筛选出最相关的资助机会。
- 结构化数据呈现：将筛选后的资助项目信息以统一、清晰、易于理解的结构化格式呈现给用户，例如通过用户友好的界面或 API 接口。
- 提升效率与时效性：显著减少人工研究资助项目所需的时间和精力，同时确保用户能够及时获取最新的资助信息，从而提高咨询建议的及时性和准确性。

通过实现这些目标，本项目旨在构建一个智能“雷达”，持续扫描并精准定位潜在的资助机会，为用户提供强大的决策支持。

1.1.3 项目价值与预期成果

本项目的成功实施将带来多方面的价值：

- 对企业和研究机构：提供一个高效、便捷的渠道，帮助它们快速发现并申请到合适的资金支持，加速创新和发展。
- 对咨询团队：赋能咨询顾问，使其从繁琐的信息搜集和初步筛选工作中解放出来，更专注于提供高价值的战略咨询和申请指导，从而提升服务质量和客户满意度。
- 对初学者和开发者：本项目将产出一份详尽的、教科书级别的开发指南。这份指南将从零开始，详细拆解项目开发的每一个步骤，包括环境配置、技术选型、核心模块实现、测试与部署等，为初学者提供一个完整的、可实践的学习案例。

预期成果：

1. 一个功能可用的 AI 资助项目智能发现系统原型。
2. 一套完整的网络爬虫模块，能够从指定数据源获取资助信息。
3. 一个初步的数据处理和特征工程流程。
4. 一个基础的 AI 匹配模型或高级规则引擎。
5. 一个科学的模型评估流程和基线性能报告。
6. 最终交付核心：一份详尽的、教科书级别的项目开发文档（本项目将以 HTML 格式输出，方便在线学习），包含所有理论讲解、代码示例、步骤说明和经验总结，旨在帮助软件开发初学者理解并有能力从头到尾实现类似系统。

1.2 核心功能需求

根据项目目标，本工具将包含以下核心功能：

1. 定制化搜索档案创建：
    - 允许用户输入详细的特征，如行业、公司年龄、营业额和员工数量等。

    - 基于这些特征对资金项目进行定向筛选，确保搜索结果的高度相关性。
2. 来源管理：
    - 能够存储和管理多个相关数据库和网站的列表。
    - 系统将定期检查这些来源，以获取最新的资金项目信息。
3. 自动化搜索：
    - 利用 AI 技术对存储的来源进行评估，自动识别新的、合适的资金项目。
    - 搜索过程将定期执行（例如每周），以确保信息的及时更新。
4. 数据准备与输出：
    - 将搜索到的原始数据进行清洗、处理和结构化。
    - 通过用户友好的界面展示最终结果，方便用户查阅和分析。

此外，在开发任何机器学习（ML）方法之前，建立一个模型评估管线至关重要。这意味着我们需要：

- 定义成功指标：明确哪些指标可以用来衡量系统（特别是未来可能引入的 ML 模型）的成功。例如，搜索结果的准确性、召回率、用户满意度等。
- 数据驱动的评估：根据可用的数据类型，设计方法来估算这些指标。即使是非常简单的启发式方法（例如，随机推荐）也可以作为基线，用于开发和验证评估管线。

思考如何构建一个“好”的系统，这个过程本身也将有助于我们更好地设计和训练未来的 ML 模型。

1.3 系统总体架构概览

为了帮助软件开发初学者更好地理解和实现本项目，我们将采用一个清晰、模块化的三层架构。这种架构不仅易于理解，也便于后续的扩展和维护。每一层都承担着特定的职责，并通过明确的接口进行通信。

本项目的系统架构可以概括为以下三层：

1. 数据爬取层（**Scraping Layer**）
2. 数据存储与管理层（**Data Storage & Management Layer**）
3. 应用逻辑层（**Application Logic Layer**）

下图展示了这三层之间的关系和数据流向：

```
graph TD
    A[外部数据源: 网站/数据库] --> B{数据爬取层};
    B --> C[原始数据];
    C --> D{数据存储与管理层};
    D -- 存储/读取 --> E[SQLite数据库];
    D -- 提供数据 --> F{应用逻辑层: 后端API};
    F -- 提供服务 --> G[应用逻辑层:前端界面];
```

G -- 用户交互 --> H[用户];
H -- 定制搜索档案 --> F;

图**1-1:** 系统架构概览

1.3.1 数据爬取层 **(Scraping Layer)**

职责：

数据爬取层是系统的“眼睛”，负责从互联网上的各种外部数据源（如政府网站、新闻门户、创业平台等）抓取原始的资金项目信息。这一层的主要任务是获取网页内容，并从中提取出我们所需的核心数据。

核心组件与技术：

- **HTTP** 请求库（**requests**）：用于向目标网站发送 HTTP 请求（GET, POST 等），获取网页的 HTML 内容或 JSON 数据。requests 库简单易用，是 Python 中进行网络请求的首选。
- **HTML** 解析库（**BeautifulSoup4, lxml**）：用于解析从网页获取的 HTML 或 XML 文档。BeautifulSoup4 能够将复杂的 HTML 结构转换为易于操作的 Python 对象，从而方便我们通过标签、类名、ID 等选择器来定位和提取所需的数据。lxml 则提供更快的解析速度和对 XPath 的支持。
- 数据清洗与结构化：在提取数据后，通常需要进行初步的清洗，去除不必要的空格、换行符或特殊字符。然后，将非结构化的文本数据转换为结构化的格式，例如 Python 字典或列表，最终整合成 Pandas DataFrame，以便于后续的数据处理和存储。

工作流程：

1. 根据预设的 URL 列表或搜索条件，向目标网站发送 HTTP 请求。
2. 接收网站返回的 HTML 响应。
3. 使用 BeautifulSoup4 或 lxml 解析 HTML，根据预定义的规则（如 CSS 选择器或 XPath）提取资金项目的标题、描述、申请条件、截止日期等信息。
4. 对提取的数据进行初步清洗和格式化。
5. 将结构化后的数据传递给数据存储与管理层。

1.3.2 数据存储与管理层（**Data Storage & Management Layer**）

职责：

数据存储与管理层是系统的“记忆”，负责持久化存储所有爬取到的资金项目数据以及用户创建的定制化搜索档案。这一层还包括数据去重、更新和查询等核心逻辑，确保数据的完整性、一致性和可访问性。

核心组件与技术：

- **SQLite** 数据库：本项目初期选择 SQLite 作为数据库解决方案。SQLite 是一个轻量级的、文件型的关系型数据库，它不需要独立的服务器进程，可以直接将数据存储在文件中。这使得它非常适合初学者项目，因为它易于设置、部署和管理，且功能强大，足以满足本项目前期的需求。
- **Python** 的 **sqlite3** 模块：Python 标准库中内置了 sqlite3 模块，可以直接通过 Python 代码与 SQLite 数据库进行交互，执行 SQL 语句来创建表、插入数据、查询数据等。
- 数据去重（**hashlib**）：为了避免重复存储相同的资金项目信息，我们将使用 hashlib 库为每个爬取到的记录生成一个唯一的哈希值（例如，基于项目标题和描述的组合）。在插入新数据之前，先检查该哈希值是否已存在于数据库中，从而实现数据去重。
- 数据更新逻辑：当爬虫再次运行时，可能会发现已存在的资金项目信息有所更新。我们需要设计逻辑来识别这些更新，并相应地更新数据库中的记录，而不是简单地插入新的重复记录。

工作流程：

1. 接收来自数据爬取层的结构化数据。
2. 为每条数据生成唯一 ID（哈希值 ）。
3. 检查数据库中是否已存在相同 ID 的记录。
4. 如果不存在，则将新数据插入到资金项目表中。
5. 如果存在且数据有更新，则更新现有记录。
6. 管理用户创建的搜索档案，包括存储、读取、更新和删除。
7. 响应应用逻辑层的查询请求，从数据库中检索相关数据。

1.3.3 应用逻辑层 (Application Logic Layer)

应用逻辑层是系统的“大脑”和“面孔”，它包含了项目的核心业务逻辑，并负责与用户进行交互。这一层又可以细分为后端 API 和前端界面两个部分。

1.3.3.1 后端API (Flask)

职责：

后端 API 是前端界面与数据存储层之间的桥梁。它负责接收前端的请求，处理业务逻辑，与数据库进行交互，并将结果以结构化的数据格式（通常是 JSON）返回给前端。Flask 是一个轻量级的 Python Web 框架，非常适合构建 RESTful API。

核心组件与技术：

- **Flask** 框架：一个微型 Web 框架，提供路由、请求处理、模板渲染等基本功能。它的简洁性使得初学者能够快速上手并理解 Web 应用的基本工作原理。
- **RESTful API** 设计原则：遵循 REST (Representational State Transfer) 原则来设计

API 接口，使用 HTTP 方法（GET, POST, PUT, DELETE）来表示对资源的增删改查操作，并使用 URL 来标识资源。
- **JSON** 数据格式：API 之间以及 API 与前端之间通常使用 JSON (JavaScript Object Notation) 格式进行数据交换，因为它轻量、易读且易于解析。

API 接口示例：

- GET /api/profiles：获取所有搜索档案。
- POST /api/profiles：创建新的搜索档案。
- PUT /api/profiles/<id>：更新指定 ID 的搜索档案。
- DELETE /api/profiles/<id>：删除指定 ID 的搜索档案。
- POST /api/scrape：触发一次数据爬取任务（可以模拟定时触发）。
- GET /api/funds：根据查询参数（如搜索档案 ID）筛选和获取资金项目列表。

工作流程：

1. 接收来自前端的 HTTP 请求。
2. 根据请求的 URL 和 HTTP 方法，匹配到相应的处理函数（路由）。
3. 在处理函数中，解析请求参数，调用数据存储与管理层的方法进行数据库操作。
4. 处理业务逻辑，例如根据搜索档案筛选资金项目。
5. 将处理结果封装成 JSON 格式的响应。
6. 将 JSON 响应返回给前端。

1.3.3.2 前端界面 (HTML/CSS/JavaScript)

职责：

前端界面是用户与系统直接交互的界面。它负责展示信息、接收用户输入，并通过 AJAX (Asynchronous JavaScript and XML) 技术与后端 API 进行通信，实现数据的动态加载和页面的无刷新更新。本项目将使用原生的 HTML、CSS 和 JavaScript，避免引入复杂的前端框架，以便初学者专注于 Web 开发的基础知识。

核心组件与技术：

- **HTML (HyperText Markup Language)**：用于构建网页的结构和内容，例如表单、按钮、文本框、表格等。
- **CSS (Cascading Style Sheets)**：用于美化网页的样式和布局，例如颜色、字体、间距、响应式设计等。
- **JavaScript**：用于实现网页的交互逻辑，例如：
  - DOM 操作：动态修改网页内容和结构。
  - 事件处理：响应用户的点击、输入等操作。
  - AJAX 请求：通过 fetch API 或 XMLHttpRequest 对象向后端 API 发送异步请求，获取或提交数据。

用户界面示例：

- 搜索档案管理页面：包含一个表单，用于创建新的搜索档案（输入行业、公司规模等），以及一个列表，展示已有的搜索档案，并提供编辑和删除功能。
- 资金项目展示页面：根据用户选择的搜索档案，展示符合条件的资金项目列表，可能包含标题、描述、来源链接、截止日期等信息，并提供分页或排序功能。

工作流程：

1. 用户通过浏览器访问前端页面。
2. 前端页面加载 HTML、CSS 和 JavaScript 文件。
3. 用户在界面上进行操作（如填写表单、点击按钮）。
4. JavaScript 捕获用户操作，并根据需要向后端 API 发送 AJAX 请求。
5. 接收后端 API 返回的 JSON 数据。
6. JavaScript 解析 JSON 数据，并动态更新前端页面，展示最新的信息。

通过这种分层设计，每个模块都相对独立，职责明确，这对于初学者来说，有助于逐步理解和掌握项目的各个部分，并为未来的复杂项目开发打下坚实的基础。

1.4 技术栈选择

为了实现上述目标，我们将采用一套现代且适合初学者的技术栈。这些技术工具的选择综合考虑了功能需求、社区支持、学习曲线以及项目进展报告中已验证的技术方案。

- 编程语言：Python 3.x（推荐最新稳定版，如 3.9+）。Python 因其简洁的语法、丰富的库生态以及在数据科学领域的广泛应用而成为首选。
- 网络爬虫：
  - Requests：用于发送 HTTP 请求，获取网页内容。
  - Beautiful Soup 4 (BS4)：用于解析 HTML 和 XML 文档，提取数据。
  - lxml：一个功能强大且速度极快的 HTML 和 XML 处理库，BeautifulSoup 可以指定它作为解析器，也可以单独用于 XPath 解析。
  - fake-useragent：用于生成随机的 User-Agent，以应对基本的反爬虫机制。
  - （可选）Selenium / Playwright：更高级的浏览器自动化工具，用于处理 JavaScript 动态加载内容和复杂反爬机制（本项目初期可先不使用，但会介绍概念）。
- 数据处理与分析：
  - Pandas：提供高性能、易用的数据结构（如 DataFrame）和数据分析工具，用于数据清洗、转换、分析和存储。
  - NumPy：Python 科学计算的基础包，为 Pandas 提供底层支持，尤其在处理数值数据方面。
- 机器学习与评估：
  - Scikit-learn：一个全面的机器学习库，包含分类、回归、聚类、降维、模型选择和预处理等工具。本项目将主要使用其模型评估模块，并可能引入简单的分类模型。

- 数据存储：
  - CSV, JSON：轻量级的数据存储格式，适合存储爬取的原始数据和中间结果。
  - SQLite：一个轻量级的磁盘数据库，无需单独的服务器进程，Python 内置支持，非常适合初学者和小型项目。
  - MinIO：一个开源的对象存储服务，与 Amazon S3 API 兼容。在本项目中，如项目进展报告所述，用于模拟 Google Cloud Storage (GCS)，测试数据上传和存储流程。
  - （可选进阶）PostgreSQL, MongoDB：更强大的关系型和 NoSQL 数据库，适用于生产环境（概念性介绍）。
- 容器化与部署：
  - Docker：用于创建、部署和运行应用程序的容器化平台，确保开发、测试和生产环境的一致性。
  - Python schedule：一个轻量级的 Python 库，用于在 Python 脚本中创建简单的定时任务。
  - （可选）Google Cloud Functions (GCF)：一个无服务器计算平台，可用于运行爬虫等后台任务（概念性介绍）。
- 版本控制：
  - Git：分布式版本控制系统，用于跟踪代码变更。
  - GitHub（或 GitLab/Bitbucket）：基于 Git 的代码托管平台，便于协作和项目管理。
- 开发工具（**IDE**）：
  - Visual Studio Code (VS Code)：轻量级且功能强大的代码编辑器，拥有丰富的 Python 和 Docker 插件支持。
  - PyCharm Community Edition：JetBrains 出品的专业 Python IDE，社区版免费。

1.5 如何使用本教程

本指南旨在为初学者提供一份“保姆级”的教程，引领您从零开始，一步步构建一个实用的 AI 赋能资助项目发现系统。为了最大限度地从本指南中获益，请注意以下几点：

- 循序渐进：指南内容按照项目开发的逻辑顺序组织。请按章节顺序学习，确保理解前一阶段的知识后再进入下一阶段。
- 理论与实践并重：指南不仅会讲解相关的理论知识点，更会提供具体的代码示例和操作步骤。请务必亲自动手编写代码、运行程序，通过实践来巩固理解。
- 理解而非复制：对于代码示例，鼓励您先尝试理解其逻辑和每一行代码的作用，而不是简单地复制粘贴。尝试修改代码，观察其行为变化，有助于加深理解。
- 积极思考与提问：在学习过程中，遇到不理解的概念或问题时，首先尝试通过搜索引擎、查阅官方文档或推荐的学习资源自行解决。如果仍有困惑，可以向社区、导师或同学请教。
- 善用参考资料：指南中会引用大量外部链接和参考资料，这些是扩展学习和深入研究的重要资源。

- 版本控制习惯：从项目开始就使用 Git 进行版本控制，勤提交代码，并学习使用分支进行功能开发，这将是您未来职业生涯中的宝贵技能。
- 保持耐心与毅力：软件开发，尤其是涉及多个技术栈的项目，学习曲线可能较为陡峭。遇到困难时不要气馁，分解问题，逐步攻克。完成这个项目将是您技能树上一次显著的成长。

本指南将力求详尽，但技术的海洋浩瀚无垠。我们希望它能成为您探索 AI 与数据科学世界的一块坚实踏脚石。祝您学习愉快，收获满满！

第二章：开发环境搭建与核心知识储备

在正式开始项目开发之前，搭建一个稳定、高效的开发环境并储备相关的基础知识至关重要。这个阶段就像为建造一栋大楼打下坚实的地基。本章将详细指导您完成开发环境的配置，并梳理项目所需的核心知识点及其学习资源。

2.1 Python安装与配置

本项目将使用 Python 作为主要的开发语言。请确保您的系统上已安装 Python 3.6 或更高版本。如果您尚未安装 Python，可以按照以下步骤进行：

2.1.1 下载Python

访问 Python 官方网站下载页面：https://www.python.org/downloads/

根据您的操作系统（Windows, macOS, Linux）选择合适的安装包。通常，建议下载最新稳定版本的 Python 3。

2.1.2 安装Python

- **Windows:**
  - 运行下载的 .exe 安装包。
  - 非常重要：在安装向导的第一步，请务必勾选“Add Python X.Y to PATH”（将 Python 添加到系统路径）选项。这将允许您在命令行中直接运行 Python 命令。
  - 选择“Install Now”进行默认安装，或选择“Customize installation”进行自定义安装（如果您熟悉）。
  - 等待安装完成。
- **macOS:**
  - 运行下载的 .pkg 安装包。
  - 按照安装向导的指示进行操作。macOS 通常会自带 Python 2，但我们需要 Python 3。安装 Python 3 不会覆盖 Python 2。
- **Linux:**
  - 大多数 Linux 发行版都预装了 Python。您可以通过在终端中运行 python3 --version 来检查 Python 3 是否已安装及其版本。
  - 如果未安装或版本过低，可以使用包管理器进行安装。例如，在基于 Debian 的系统（如 Ubuntu）上：
    sudo apt update
    sudo apt install python3 python3-pip python3-venv
  - 在基于 RPM 的系统（如 Fedora, CentOS）上：
    sudo dnf install python3 python3-pip python3-venv

2.1.3 验证安装

安装完成后，打开命令行终端（Windows: Command Prompt 或 PowerShell; macOS/Linux: Terminal），输入以下命令，如果能正确显示 Python 版本号，则表示安装成功：

```
python3 --version
```

同时，验证 pip（Python 的包管理工具）是否也已安装：

```
pip3 --version
```

2.2 虚拟环境的创建与管理

在 Python 开发中，强烈推荐使用虚拟环境（Virtual Environment）。虚拟环境可以为每个项目创建一个独立的 Python 运行环境，使得项目之间所需的库版本互不干扰，避免潜在的冲突。

2.2.1 为什么使用虚拟环境?

想象一下，您有两个 Python 项目：项目 A 需要 requests 库的 1.0 版本，而项目 B 需要 requests 库的 2.0 版本。如果没有虚拟环境，您全局安装的 requests 库只能是其中一个版本，这将导致另一个项目无法正常运行。虚拟环境解决了这个问题，它允许您在每个项目中安装特定版本的库，而不会影响其他项目或系统全局的 Python 环境。

2.2.2 创建虚拟环境

1. 进入项目目录：首先，在您的文件系统中创建一个用于存放本项目的文件夹，并进入该文件夹。例如：
   ```
   mkdir funding_search_tool
   cd funding_search_tool
   ```

2. 创建虚拟环境：在项目目录下，使用 venv 模块（Python 3.3+ 内置）创建虚拟环境。venv 是 virtualenv 的轻量级替代品。
   ```
   python3 -m venv venv
   ```

   这会在当前目录下创建一个名为 venv 的文件夹（您可以选择其他名称，但 venv 是约定俗成的）。这个文件夹包含了 Python 解释器的一个副本以及用于安装包的 pip 工具。

2.2.3 激活虚拟环境

在开始项目开发之前，每次都需要激活虚拟环境。激活后，您在命令行中使用的 python 和 pip 命令都将指向虚拟环境中的解释器和包管理器。

- **Windows (Command Prompt):**
  .\venv\Scripts\activate

- **Windows (PowerShell):**
  .\venv\Scripts\Activate.ps1

  （可能需要先执行 Set-ExecutionPolicy Unrestricted -Scope Process 来允许脚本执行）
- **macOS/Linux:**
  source venv/bin/activate

激活成功后，您的命令行提示符前会显示 (venv)，表示您当前正处于虚拟环境中。

2.2.4 退出虚拟环境

当您完成项目开发或需要切换到其他项目时，可以简单地输入 deactivate 命令来退出虚拟环境：

deactivate

2.3 核心依赖库安装

激活虚拟环境后，我们可以使用 pip 来安装本项目所需的核心 Python 库。这些库包括：

- Flask: 轻量级 Web 框架，用于构建后端 API。
- requests: 用于发送 HTTP 请求，获取网页内容。
- BeautifulSoup4: 用于解析 HTML 和 XML 文档，提取数据。
- lxml: 高性能 XML/HTML 处理库，常与 BeautifulSoup4 配合使用，或单独用于 XPath 解析。
- pandas: 用于数据结构化、清洗和分析。
- scikit-learn: 机器学习库，用于模型评估和可能的智能匹配模型。
- fake-useragent: 用于生成随机的浏览器 User-Agent 字符串。
- schedule: 用于创建简单的定时任务。
- hashlib: Python 内置库，用于生成哈希值，实现数据去重（无需单独安装）。
- MinIO：MinIO 官方 Python 客户端库，用于与 MinIO 对象存储服务交互（用于模拟云存储 ）。

在激活的虚拟环境中，运行以下命令安装这些库：

pip install Flask requests beautifulsoup4 lxml pandas scikit-learn fake-useragent schedule minio

安装完成后，您可以通过 pip freeze 命令查看虚拟环境中已安装的所有库及其版本：

pip freeze > requirements.txt

这会将您虚拟环境中安装的所有 Python 库及其版本信息输出到 requirements.txt 文件中。之后，其他人或在新的环境中，只需运行 pip install -r requirements.txt 即可安装所有依赖。

2.4 集成开发环境（IDE）选择与配置

集成开发环境（IDE）能够提供代码编辑、调试、版本控制、自动补全等一系列功能，极大地提高开发效率。对于 Python 开发，我们强烈推荐使用 Visual Studio Code (VS Code) 或 PyCharm Community Edition。

2.4.1 安装 VS Code

访问 VS Code 官方网站：https://code.visualstudio.com/

根据您的操作系统下载并安装 VS Code。

2.4.2 配置 VS Code 进行 Python 开发

1. 安装 **Python** 扩展：
    - 打开 VS Code。
    - 点击左侧活动栏的“Extensions”图标（或按下 Ctrl+Shift+X）。
    - 在搜索框中输入“Python”，找到由 Microsoft 提供的 Python 扩展并点击“Install”。推荐同时安装 Pylance（Microsoft 官方），它提供更快速、更智能的类型检查和代码补全。
    - 如果您打算使用 Docker，也可以安装 Docker 扩展。
2. 选择 **Python** 解释器：
    - 打开您的项目文件夹（funding_search_tool）。
    - 在 VS Code 的左下角，点击状态栏中的 Python 版本号（如果没有显示，可以按下 Ctrl+Shift+P，然后输入“Python: Select Interpreter”）。
    - 选择您刚刚创建的虚拟环境中的 Python 解释器（通常会显示为 Python 3.x.x ('venv': venv) 或类似路径）。选择正确的解释器是确保 VS Code 使用虚拟环境中安装的库的关键。

3. 创建第一个 **Python** 文件：
    - 在 VS Code 中，点击“File”->“New File”，保存为 hello.py。
    - 输入以下代码：
      print("Hello, Funding Search Tool!")

    - 右键点击编辑器中的代码，选择“Run Python File in Terminal”，或者点击右上角的“Run”按钮。如果一切配置正确，您将在 VS Code 的集成终端中看到输出。

至此，您的 Python 开发环境已成功搭建，并配置了 VS Code，为接下来的项目开发做好了准备。

2.4.3 PyCharm Community Edition（可选）

- 特点：由 JetBrains 开发，是一款功能非常强大的 Python IDE，社区版免费。它提供了智能代码补全、强大的调试器、版本控制集成、数据库工具等。
- 下载地址：https://www.jetbrains.com/pycharm/download/
- 配置：创建新项目或打开现有项目时，PyCharm 会引导您配置 Python 解释器。选择“Existing interpreter”并找到您虚拟环境中的 Python 可执行文件。

选择哪款 IDE 主要取决于个人偏好。两者都能很好地支持本项目开发。

2.5 版本控制工具 Git 入门：代码时光机

Git 是目前最流行的分布式版本控制系统，用于跟踪代码的修改历史、协作开发以及代码备份。GitHub、GitLab 等代码托管平台都是基于 Git 的。

- **Git** 安装：访问 https://git-scm.com/downloads/ 下载并安装适合您操作系统的 Git。
- 基本配置：安装完成后，打开命令行终端，设置您的用户名和邮箱。这些信息会记录在您的每一次提交中。
  git config --global user.name "Your Name"
  git config --global user.email "youremail@example.com"

- 项目仓库初始化与基本操作：
    1. 创建项目目录并进入：
       mkdir funding_radar_project
       cd funding_radar_project

    2. 初始化 **Git** 仓库：
       git init

       这会在项目目录下创建一个 .git 隐藏文件夹，用于存储版本信息。
    3. 基本工作流程：

    - 查看状态：git status（查看哪些文件被修改或未被跟踪）
    - 添加文件到暂存区：git add <file_name>（添加特定文件）或 git add .（添加所有修改和新文件）
    - 提交更改：git commit -m "Your descriptive commit message"（将暂存区的内容提交到本地仓库，并附带一条说明信息）
    - 查看提交历史：git log
4. （可选）关联远程仓库：如果您在 GitHub 等平台创建了一个空仓库，可以将其与本地仓库关联，以便推送代码。
   git remote add origin <远程仓库URL>
   git branch -M main # 确保本地主分支名为 main
   git push -u origin main # 首次推送并设置上游分支

- 学习资源：
    - Pro Git（中文版）- 非常经典的 Git 学习教材。
    - 廖雪峰的 Git 教程 - 适合初学者快速入门。

养成良好的 Git 使用习惯对项目管理和团队协作至关重要。

2.6 必备知识点梳理与学习资源

掌握以下基础知识将对您理解和完成本项目大有裨益。虽然本教程会尽量详细，但提前具备这些知识能让您学习得更快。

2.6.1 Python 编程基础

- 核心概念：变量、数据类型（字符串、列表、字典、元组、集合）、运算符、控制流（if/else, for/while 循环）、函数定义与调用、类与面向对象编程 (OOP) 基础（封装、继承、多态的概念）、模块与包的导入和使用、文件操作（读写文件）、异常处理（try-except-finally）。
- 推荐资源：
    - Python 官方教程（权威且全面）
    - 菜鸟教程 Python3（中文，适合快速入门）

2.6.2 网络基础与 HTTP 协议

- 核心概念：HTTP/HTTPS 基本原理（请求-响应模型），URL 结构（协议、域名、路径、查询参数），HTTP 请求方法 (GET, POST 等及其区别），HTTP 头部（Headers）的作用，特别是 User-Agent, Content-Type, Cookies；HTTP 响应状态码的含义（如 200 OK, 301 Moved Permanently, 403 Forbidden, 404 Not Found, 500 Internal Server Error）；Cookies 与 Session 机制的基本原理（用于维持状态和用户会话）。
- 推荐资源：
    - MDN Web Docs - HTTP（非常权威和详细）

- 《图解 HTTP》(书籍，通俗易懂)

2.6.3 网页结构基础(**HTML, CSS** 选择器**, XPath**)

- 核心概念：HTML 文档的基本结构(<html>, <head>, <body>)，常用 HTML 标签(如 , <p>, <a>, <img>, <table>, , <ul>, <li>, <h1>-<h6> 等)及其作用；标签属性(如 id, class, href, src, style)；DOM(文档对象模型)树的概念。CSS 选择器语法(用于选取 HTML 元素，如标签选择器 p, 类选择器 .classname, ID 选择器 #idname, 属性选择器 [attribute=value], 后代选择器 div p, 子代选择器 div > p)。XPath 路径表达式语法(另一种强大的节点选择语言)。
- 推荐资源：
  - W3Schools HTML Tutorial
  - W3Schools CSS Selectors
  - W3Schools XPath Syntax
  - MDN Web Docs - HTML

2.6.4 数据处理基础(**Pandas**)

- 核心概念：Pandas 的两个核心数据结构 DataFrame(二维表格型数据)和 Series(一维数组型数据)；从文件(CSV, Excel, JSON 等)读取数据到 DataFrame；数据查看与探索常用方法(如 .head(), .tail(), .info(), .describe(), .shape, .dtypes)；数据选择与切片(使用 .loc[] 基于标签索引，.iloc[] 基于位置索引，条件选择/布尔索引)；数据清洗常见操作(处理缺失值如 .isnull(), .fillna(), .dropna()；处理重复值如 duplicated(), drop_duplicates())；数据转换与应用函数(如 .apply(), .map(), 数据类型转换 .astype())。
- 推荐资源：
  - Pandas 官方十分钟入门
  - 《利用 Python 进行数据分析》(Wes McKinney 著，Pandas 库的作者)

2.6.5 机器学习入门与模型评估

- 核心概念：理解什么是机器学习，监督学习(Supervised Learning)的基本概念(如分类 Classification、回归 Regression)；训练集(Training Set)、验证集(Validation Set)、测试集(Test Set)的划分及其作用；特征(Features/Input Variables)与标签(Labels/Target Variable)的概念；模型评估的基本概念，如准确率(Accuracy)、精确率(Precision)、召回率(Recall)、F1 分数 (F1-Score) 的定义和计算场景。
- 推荐资源：
  - Scikit-learn 官方文档用户指南(特别是模型评估部分)
  - 吴恩达(Andrew Ng)的机器学习课程(Coursera 或 B 站有中文字幕版，经典入门)

2.6.6 Docker 基础(概念性了解)

- 核心概念：理解什么是容器化技术；镜像(Image)的概念(一个轻量级、可执行的软件

包，包含运行应用所需的一切：代码、运行时、库、环境变量和配置文件）；容器（Container）的概念（镜像的运行实例）；Dockerfile（用于定义如何构建镜像的文本文件）；Docker Hub（公共的镜像仓库）。了解 Docker 的主要优势（如环境一致性、快速部署、资源隔离、可移植性）。

- 推荐资源：
  - Docker 官方入门指南
  - 菜鸟教程 Docker

以上知识点的掌握程度将直接影响您后续项目的顺利进行。建议在开始具体编码前，花一些时间回顾和学习这些基础内容。

关键要点总结

- 环境搭建：安装 Python 3.x，配置好虚拟环境（venv），选择并熟悉一款 IDE (VS Code 或 PyCharm)。
- 版本控制：安装 Git，进行基本配置，并为项目初始化 Git 仓库。
- 核心库安装：在虚拟环境中通过 pip 安装 Flask, requests, beautifulsoup4, lxml, pandas, scikit-learn, fake-useragent, schedule, minio 等。
- 知识储备：重点学习 Python 编程、HTTP 协议、HTML/CSS 选择器/XPath、Pandas 数据处理、机器学习评估基础以及 Docker 基本概念。

第三章：数据引擎核心：网络爬虫开发实战

网络爬虫是本项目的“数据触手”，负责从互联网上自动抓取资助项目相关信息。本章将深入探讨网络爬虫开发的伦理规范、核心技术栈、反爬策略，并针对目标网站给出具体的爬取方案和代码实现思路。这是将原始信息转化为项目“燃料”的关键一步。

3.1 网络爬虫的“行为准则”：伦理、法规与 robots.txt

在启动任何爬虫项目之前，理解并遵守相关的伦理和法规至关重要。一个负责任的开发者应当像一个有礼貌的访客一样对待目标网站，而不是一个“不速之客”。

3.1.1 robots.txt 协议深度解读

- 什么是 **robots.txt**：它是一个存放在网站根目录下的文本文件（例如 https://www.example.com/robots.txt），用于告知网络爬虫（Web Crawlers/Spiders）哪些页面的内容可以被抓取，哪些不可以。这是一种“君子协定”，主流的搜索引擎爬虫（如 Googlebot, Bingbot）会遵守这些规则，但并非所有爬虫都会。
- 语法解析：
  - User-agent：指定规则适用的爬虫名称。User-agent: * 表示适用于所有爬虫。
  - Disallow：指定禁止访问的 URL 路径。例如，Disallow: /admin/ 禁止爬虫访问所有以 /admin/ 开头的路径。Disallow: / 禁止访问整个网站。
  - Allow：指定允许访问的 URL 路径，通常用于在 Disallow 规则下开特例。例如，Disallow: /private/ 后跟 Allow: /private/public.html。
  - Crawl-delay：（非标准，但部分爬虫支持）指定爬虫两次请求之间的最小时间间隔（秒），用于减轻服务器压力。
  - Sitemap：指向网站的 XML 站点地图文件，帮助爬虫发现网站上的所有重要页面。
- 如何检查和遵守：在编写爬虫代码前，务必手动访问目标网站的 robots.txt 文件（例如，https://www.ptj.de/robots.txt 或 https://www.eu-startups.com/robots.txt）。根据提供的参考资料，直接访问上述网站的 robots.txt 文件结果为空。这意味着网站没有通过 robots.txt 明确禁止通用爬虫。然而，这不代表可以无节制爬取，仍需遵守通用爬虫礼仪。

3.1.2 数据抓取的合法性与道德考量

- 版权问题：抓取的内容（文本、图片、数据）可能受版权保护。确保您的使用方式符合合理使用原则，或已获得授权。本项目主要用于信息聚合和分析，非直接商业复制。
- 隐私问题：避免抓取和存储个人身份信息（PII），如姓名、邮箱、电话号码等，除非这些信息是公开用于联系且与资助项目直接相关。如果抓取到，应进行脱敏处理或不予存储。
- 服务器负载：过于频繁或并发量过大的请求会给目标服务器带来巨大压力，可能导致其服务中断，影响正常用户访问。务必设置合理的请求间隔（time.sleep()），避免在短时间内发送大量请求。

- 遵守网站服务条款 **(Terms of Service/Use)**：许多网站的服务条款中会明确说明是否允许自动化抓取。在爬取前应尽量查阅并遵守。
- 透明性与身份标识：在请求头中设置一个明确的、能代表您爬虫身份的 User-Agent 字符串（例如，包含项目名称或联系方式），而不是伪装成常用浏览器（除非目标网站有严格的浏览器 UA 检测 ）。这有助于网站管理员识别您的爬虫并在必要时联系您。

3.2 API 优先原则

在动手编写爬虫从网页解析数据之前，务必优先调研目标网站是否提供官方 API (Application Programming Interface)。

API 是网站提供方主动开放的数据接口，通常返回结构化的数据（如 JSON 格式），比解析 HTML 更稳定、高效、准确，并且是合法合规获取数据的最佳途径。

例如，项目描述中提到的数据源之一 deutsche-digitale-bibliothek.de 就提供了 API。如果 API 能满足数据需求，应优先使用 API。

3.3 爬虫通用技术栈与核心策略

掌握以下技术和策略是构建高效、稳定爬虫的基础。

3.3.1 HTTP 请求库 Requests：与服务器对话的使者

- 发送 **GET** 请求：用于获取网页内容或通过 URL 参数查询数据。

```
import requests
import time
import random
from fake_useragent import UserAgent

# 初始化 UserAgent
ua = UserAgent()

def fetch_page_content(url, params=None, retries=3, delay=5):
    """
    发送 GET 请求获取网页内容，并处理超时和错误。
    :param url: 目标 URL
    :param params: GET 请求参数字典
    :param retries: 重试次数
    :param delay: 重试间隔秒数
    :return: 网页 HTML 内容或 None
    """
    headers = {'User-Agent': ua.random} # 使用随机 User-Agent
```

```
    for attempt in range(retries):
        try:
            # 设置超时，避免长时间等待
            response = requests.get(url, params=params, headers=headers,
timeout=20)
            response.raise_for_status() # 如果状态码不是 2xx，则抛出异常
            print(f"成功获取: {response.url}")
            return response.text
        except requests.exceptions.Timeout:
            print(f"请求超时: {url}, 尝试次数 {attempt + 1}/{retries}")
            time.sleep(delay * (attempt + 1) + random.uniform(0, 2)) # 递增延时并增加
随机性
        except requests.exceptions.RequestException as e:
            print(f"请求错误 {url}: {e}, 尝试次数 {attempt + 1}/{retries}")
            time.sleep(delay + random.uniform(0, 2)) # 增加随机延时
    return None

# 示例使用
# html_content =
fetch_page_content('https://www.ptj.de/suche-foerderinitiativen')
# if html_content:
#     print(f"获取到的 HTML 长度: {len(html_content)}")
```

- 发送 **POST** 请求：通常用于提交表单数据（如登录、搜索）。

```
def post_form_data(url, data_payload, retries=3, delay=5):
    """
    发送 POST 请求提交表单数据。
    :param url: 目标 URL
    :param data_payload: 提交的数据字典
    :param retries: 重试次数
    :param delay: 重试间隔秒数
    :return: 响应 HTML 内容或 None
    """
    headers = {'User-Agent': ua.random, 'Content-Type':
'application/x-www-form-urlencoded'}
    for attempt in range(retries):
        try:
            response = requests.post(url, data=data_payload, headers=headers,
timeout=20)
```

```
            response.raise_for_status()
            print(f"成功 POST 到: {response.url}")
            return response.text
        except requests.exceptions.Timeout:
            print(f"POST 请求超时: {url}, 尝试次数 {attempt + 1}/{retries}")
            time.sleep(delay * (attempt + 1) + random.uniform(0, 2))
        except requests.exceptions.RequestException as e:
            print(f"POST 请求错误 {url}: {e}, 尝试次数 {attempt + 1}/{retries}")
            time.sleep(delay + random.uniform(0, 2))
    return None

# 示例 POST 请求 (假设有一个登录页面)
# login_url = "https://example.com/login"
# login_payload = {'username': 'testuser', 'password': 'testpassword'}
# login_html = post_form_data(login_url, login_payload)
# if login_html:
#     print(f"登录页面响应 HTML 长度: {len(login_html)}")
```

- 处理 **Cookies** 与 **Session**：对于需要登录或维持会话状态的网站，使用 requests.Session() 对象可以自动管理 Cookies。

```
def login_and_fetch(login_url, login_payload, protected_url):
    """
    使用 session 登录后访问受保护页面。
    :param login_url: 登录 URL
    :param login_payload: 登录数据
    :param protected_url: 受保护页面的 URL
    :return: 受保护页面的 HTML 内容或 None
    """
    session = requests.Session()
    session.headers.update({'User-Agent': ua.random}) # session 级别的 User-Agent

    try:
        # 登录请求，session 会自动管理 cookies
        login_response = session.post(login_url, data=login_payload, timeout=20)
        login_response.raise_for_status()
        print(f"登录成功: {login_url}")

        # 后续使用该 session 对象的请求会自动带上登录后的 cookies
```

```
        protected_page_response = session.get(protected_url, timeout=20)
        protected_page_response.raise_for_status()
        print(f"成功访问受保护页面: {protected_url}")
        return protected_page_response.text
    except requests.exceptions.RequestException as e:
        print(f"登录或访问受保护页面失败: {e}")
        return None

# 示例登录和抓取
# protected_html = login_and_fetch('https://example.com/login', {'user': 'myuser',
'pass': 'mypass'}, 'https://example.com/dashboard')
# if protected_html:
#     print(f"受保护页面 HTML 长度: {len(protected_html)}")
```

- 处理响应：检查 response.status_code（如 200 表示成功），获取响应内容（response.text 用于文本，response.content 用于二进制数据如图片，response.json() 用于解析 JSON 响应）。
- 超时设置与异常处理：通过 timeout 参数设置请求超时时间，并使用 try-except 块捕获可能发生的网络异常（如 requests.exceptions.Timeout, requests.exceptions.ConnectionError 等）。在上面的 fetch_page_content 函数中已经包含了这些。

3.3.2 HTML/XML 解析库 BeautifulSoup4 与 lxml：网页内容的解剖刀

- 初始化：将获取到的 HTML 内容传递给 BeautifulSoup 进行解析。推荐使用 lxml 作为解析器，因为它速度快且容错性好。

```
from bs4 import BeautifulSoup
from lxml import etree # 用于 lxml 的 XPath 功能

def parse_html_with_bs4(html_content):
    """
    使用 BeautifulSoup4 和 lxml 解析器解析 HTML 内容。
    :param html_content: 网页 HTML 字符串
    :return: BeautifulSoup 对象
    """
    if not html_content:
        return None
    # 推荐使用 'lxml' 解析器，因为它速度快且功能强大
    soup = BeautifulSoup(html_content, 'lxml')
    return soup
```

```
# 示例使用
# html_doc = "<html><body><div class='container'><h1 id='title'>Hello</h1><p
class='text'>Some text</p></div></body></html>"
# soup = parse_html_with_bs4(html_doc)
# if soup:
#     print(soup.prettify()) # 格式化输出 HTML
```

- 标签选择（**BeautifulSoup**）：
  - soup.find('tag_name', attrs={'attribute_name': 'value'})：查找第一个符合条件的标签。
  - soup.find_all('tag_name', class_='class_name', id='id_value')：查找所有符合条件的标签，返回一个列表。
  - soup.select('CSS_SELECTOR')：使用类似 CSS 的选择器语法选取元素，非常灵活。

```
def extract_data_with_bs4(soup):
    """
    从 BeautifulSoup 对象中提取数据。
    :param soup: BeautifulSoup 对象
    :return: 提取到的数据字典
    """
    data = {}
    if not soup:
        return data

    # 1. 通过标签名和属性查找单个元素
    title_tag = soup.find('h1', class_='item-title') # 假设网页中存在 <h1
class="item-title">
    if title_tag:
        data['title'] = title_tag.get_text(strip=True) # 获取文本内容并去除首尾空白

    # 2. 通过 CSS 选择器查找所有元素
    # 假设网页中有很多 <div class="item">，每个 item 下面有 <a class="item-link">
    item_links = soup.select('div.item a.item-link')
    data['links'] = [link['href'] for link in item_links if 'href' in link.attrs] # 获取 href 属
性

    # 3. 查找特定 ID 的元素
    # description_div = soup.find(id='project-description')
    # if description_div:
    #     data['description'] = description_div.get_text(strip=True)
```

```
    return data

# 示例 HTML
# sample_html = """
# <html><body>
# <h1 class="item-title">资助项目 A</h1>
# <div id="project-description"><p>这是一个关于创新项目的描述。</p></div>
# <div class="item"><a class="item-link" href="/link1">链接1</a></div>
# <div class="item"><a class="item-link" href="/link2">链接2</a></div>
# </body></html>
# """
# sample_soup = parse_html_with_bs4(sample_html)
# extracted_data = extract_data_with_bs4(sample_soup)
# print(extracted_data)
```

- 获取文本内容：element.text 或 element.get_text(strip=True) (strip=True 可以去除首尾多余空白)。
- 获取标签属性：element['href'] 或 element.get('src')。
- 遍历 **DOM** 树：通过 .children, .parent, find_next_sibling() 等方法在 DOM 树中导航。

3.3.3 数据提取进阶：**XPath** 与 **lxml** 库

- **XPath** 简介：一种用于在 XML（也适用于 HTML）文档中选取节点的语言。其路径表达式提供了比 CSS 选择器更强大的定位能力，尤其在处理复杂或不规则的 HTML 结构时。
- **lxml** 库使用：lxml 是一个高性能的 XML/HTML 处理库，原生支持 XPath。

```
from lxml import etree

def parse_html_with_lxml(html_content):
    """
    使用 lxml 的 etree 解析 HTML 内容。
    :param html_content: 网页 HTML 字符串
    :return: lxml HTML 树对象
    """
    if not html_content:
        return None
    # etree.HTML 自动处理 HTML 结构中的错误
    tree = etree.HTML(html_content)
    return tree

def extract_data_with_xpath(tree):
```

```
    """
    从 lxml 树对象中提取数据。
    :param tree: lxml HTML 树对象
    :return: 提取到的数据字典
    """
    data = {}
    if not tree:
        return data

    # 1. 选取所有 class 为 "item" 的 div 下，具有 href 属性的 a 标签的文本内容
    # XPath 常用语法:
    # /: 从根节点选择
    # //: 从当前节点选择文档中的节点，不考虑它们的位置
    # .: 选取当前节点
    # ..: 选取当前节点的父节点
    # @: 选取属性
    # node_name: 选取此节点名的所有子节点
    # [predicate]: 谓语，用于查找满足特定条件的节点。如 [@class="value"],
[position()=1], [contains(text(), "keyword")]
    # text(): 获取文本节点
    # contains(), starts-with(), ends-with() 等函数
    titles = tree.xpath('//div[@class="item"]/a[@href]/text()')
    data['titles_from_xpath'] = titles

    # 2. 选取所有 href 属性以 "https://example.com/details/" 开头的 a 标签的 href 值
    links = tree.xpath('//a[starts-with(@href,
"https://example.com/details/")]/@href')
    data['links_from_xpath'] = links

    # 3. 获取 ID 为 'title' 的 h1 标签的文本
    # text() 函数用于获取元素的直接文本内容
    title_from_id = tree.xpath('//h1[@id="title"]/text()')
    if title_from_id:
        data['title_from_id'] = title_from_id[0].strip()

    return data

# 示例 HTML
# xpath_sample_html = """
```

```
# <html><body>
# <h1 id="title">项目名称 XYZ</h1>
# <div class="item"><a href="https://example.com/details/1">项目详情 1</a></div>
# <div class="item"><a href="https://example.com/details/2">项目详情 2</a></div>
# <a href="https://another.com/page">其他链接</a>
# </body></html>
# """
# xpath_tree = parse_html_with_lxml(xpath_sample_html)
# xpath_extracted_data = extract_data_with_xpath(xpath_tree)
# print(xpath_extracted_data)
```

3.4 反爬机制识别与应对基础

网站为了保护其数据和服务器资源，常会采用一些反爬虫机制。理解并适当应对这些机制是爬虫成功的关键。

- **User-Agent** 检测：服务器检查请求头中的 User-Agent 字段，如果为空或是典型的爬虫 UA（如 python-requests/2.28.1），可能会拒绝服务。
  - 应对：使用 fake-useragent 库生成随机的、看起来像真实浏览器的 User-Agent。在上面的 fetch_page_content 函数中已经包含了这个功能。
- **IP** 限制/封禁：如果来自同一 IP 地址的请求过于频繁或行为异常，服务器可能会暂时或永久封禁该 IP。
  - 应对：
    1. 降低请求频率：在每次请求后加入随机延时，例如 time.sleep(random.uniform(1, 3))。在 fetch_page_content 中也已加入。
    2. 使用 **IP** 代理池（概念）：通过购买商业 IP 代理服务或自行搭建（较复杂）IP 代理池，轮换使用不同的 IP 地址发送请求。这是应对 IP 封禁的有效手段，但增加了项目复杂度和成本。

       ```
       # proxies = {
       #     'http': 'http://user:password@proxyserver:port',
       #     'https': 'https://user:password@proxyserver:port',
       # }
       # response = requests.get(url, headers=headers, proxies=proxies)
       ```

- 请求频率限制：严格遵守 robots.txt 中声明的 Crawl-delay。如果没有明确声明，也应根据经验设置合理的请求间隔，避免对服务器造成过大压力。
- 动态加载内容（**JavaScript** 渲染）：
  - 识别：如果目标数据在浏览器中可见，但在爬虫获取的 HTML 源码（response.text）中不存在，很可能是通过 JavaScript 动态加载的。

    - 初步应对：
        1. 分析网络请求：打开浏览器开发者工具（通常按 F12），切换到"Network”（网络）标签页，筛选 XHR（XMLHttpRequest）或 Fetch 请求。刷新页面或与页面交互，观察是否有请求直接返回了包含目标数据的 JSON 或 XML。如果找到这样的 API 接口，直接请求该接口通常比渲染整个页面更高效。
        2. 本项目初期尽量避免需要完整 JS 渲染的场景，如项目进展报告中提到“selenium was avoided... as the sites are static”，说明目标网站当时可能主要是静态内容或可通过分析 XHR 获取。
    - 进阶应对（本项目可选，概念性）：使用如 Selenium、Playwright 等浏览器自动化工具，它们可以模拟真实浏览器行为，执行 JavaScript 并获取渲染后的页面内容。但这会增加系统复杂度和资源消耗。
- 验证码：图形验证码、滑动验证码等对爬虫是较大挑战。
    - 应对：本项目初期可先跳过需要验证码的页面，或手动处理（例如，登录时手动输入验证码获取 cookies，然后让爬虫使用这些 cookies）。自动化识别验证码通常需要 OCR 技术和机器学习模型，较为复杂。
- 登录限制：部分数据需要用户登录后才能访问。
    - 应对：使用 requests.Session() 模拟登录过程。首先分析登录请求的 URL、请求方法（通常是 POST）、表单数据（用户名、密码、可能的隐藏字段如 CSRF token），然后发送登录请求。成功登录后，Session 对象会自动保存服务器返回的 Cookies，后续使用该 Session 对象发出的请求就会携带这些 Cookies，从而维持登录状态。

3.5 目标网站爬虫开发实例：逐个攻破

接下来，我们将针对项目需求中提到的几个主要数据源，分析其特点并制定爬取策略。在实际操作前，请务必亲自访问这些网站并使用浏览器开发者工具进行详细分析，因为网站结构可能会发生变化。

通用分析步骤：

1. 目标明确：确定需要从该网站抓取哪些具体信息字段。例如：项目名称、资助机构、资助金额范围、申请条件（行业、公司规模、成立年限、营业额等）、项目描述、申请截止日期、官方链接等。
2. 网站探索：手动浏览网站，熟悉其信息组织结构。关注资助项目通常在哪些板块发布（如“Förderprogramme", "Funding Opportunities", "News", "Database”等）。尝试使用网站的搜索功能，观察 URL 如何变化，是否有分页。
3. 开发者工具侦查 **(F12)**：
    - Elements（元素）面板：检查目标数据在 HTML 代码中的具体位置，记录其标签名、class 属性、id 属性等，这些是编写 CSS 选择器或 XPath 的基础。
    - Network（网络）面板：刷新页面或进行搜索操作时，监控所有网络请求。特别关注 XHR/Fetch 类型的请求，这些请求可能是异步加载数据的 API 调用。记录关键请

求的 URL、请求方法（GET/POST）、请求头（Headers）、请求体（Payload/Form Data）和响应内容（Response）。

4. **robots.txt 与 API 检查**：
    - 访问网站根目录下的 robots.txt 文件（如 https://www.example.com/robots.txt），仔细阅读其规则。
    - 在网站的页脚、“关于我们”、“开发者”等区域寻找是否有官方 API 的链接或说明。也可以通过搜索引擎搜索"[website name] API "。
5. 爬取策略制定：
    - 数据来源：是直接解析 HTML 页面，还是调用分析到的 API 接口？
    - **URL** 规律：如何构造列表页的 URL（例如，通过查询参数控制分类、页码）？如何从列表页获取详情页的 URL？
    - 数据提取规则：针对 HTML，编写精确的 CSS 选择器或 XPath 表达式。针对 API，明确 JSON/XML 的解析路径。
    - 反爬预案：根据网站特点，预估可能遇到的反爬机制，并准备相应的应对策略（如 User-Agent 轮换、请求延时、IP 代理等）。

我们将为每个网站创建一个独立的爬虫文件，例如 scraper_ptj.py, scraper_eustartups.py 等。

3.5.1 爬取 ptj.de 示例

ptj.de 是一个提供资金项目信息的网站。我们将以其为例，演示如何爬取页面内容。

首先，我们来检查 https://www.ptj.de/robots.txt。假设我们发现该网站允许爬取其公开的资金项目页面（实际验证时该文件可能为空，意味着没有明确禁止）。

创建一个名为 scraper_ptj.py 的文件，并添加以下代码：

```
# scraper_ptj.py

import requests
from bs4 import BeautifulSoup
import pandas as pd
import hashlib
import time
import random
from fake_useragent import UserAgent

# 初始化 UserAgent
ua = UserAgent()
```

```
def scrape_ptj():
    """
    爬取 ptj.de 网站的资金项目信息。
    请注意：HTML 选择器是示例性的，实际开发中需要根据网站的实时结构进行调整。
    """
    base_url = "https://www.ptj.de/"
    search_url = "https://www.ptj.de/suche-foerderinitiativen" # 示例搜索页面 URL

    headers = {
        'User-Agent': ua.random # 模拟浏览器请求头，避免被识别为爬虫
    }

    print(f"正在爬取: {search_url}")
    html_content = None
    try:
        response = requests.get(search_url, headers=headers, timeout=10)
        response.raise_for_status() # 检查 HTTP 请求是否成功
        html_content = response.text
    except requests.exceptions.RequestException as e:
        print(f"请求失败: {e}")
        return pd.DataFrame()

    if not html_content:
        return pd.DataFrame()

    soup = BeautifulSoup(html_content, 'lxml')

    # 查找资金项目列表的容器
    # 这里的选择器需要根据实际网页结构进行调整。
    # 假设资金项目列表在一个 class 为 'search-results' 的 div 中，每个项目是一个 class
为 'result-item' 的 div
    # **请注意：这只是一个示例，实际的 HTML 结构可能不同，您需要通过浏览器开发者工
具检查实际的 HTML 结构**
    results_container = soup.find('div', class_='search-results')
    if not results_container:
        print("未找到资金项目列表容器, 请检查选择器或网页结构。")
        return pd.DataFrame()

    fund_items = results_container.find_all('div', class_='result-item')
```

```
    if not fund_items:
        print("未找到任何资金项目，请检查选择器或网页结构。")
        return pd.DataFrame()

    data = []
    for item in fund_items:
        title_tag = item.find('h3', class_='item-title')
        description_tag = item.find('p', class_='item-description')
        link_tag = item.find('a', class_='item-link')

        title = title_tag.get_text(strip=True) if title_tag else 'N/A'
        description = description_tag.get_text(strip=True) if description_tag else 'N/A'
        link = base_url + link_tag['href'] if link_tag and 'href' in link_tag.attrs else 'N/A'

        # 生成唯一 ID
        unique_id = hashlib.md5(f"{title}{description}{link}".encode('utf-8')).hexdigest()

        data.append({
            'id': unique_id,
            'title': title,
            'description': description,
            'link': link,
            'source': 'ptj.de'
        })

    df = pd.DataFrame(data)
    print(f"从 {search_url} 爬取到 {len(df)} 条数据。")
    return df

if __name__ == "__main__":
    ptj_df = scrape_ptj()
    if not ptj_df.empty:
        print("\n爬取到的数据示例 (ptj.de):")
        print(ptj_df.head())
        # 您可以将数据保存到 CSV 或数据库
        # ptj_df.to_csv('ptj_funds.csv', index=False, encoding='utf-8-sig') # 建议保存为
utf-8-sig 避免 Excel 乱码
```

代码解释：

1. 导入必要的库：requests 用于发送 HTTP 请求，BeautifulSoup 用于解析 HTML，pandas 用于数据处理，hashlib 用于生成唯一 ID，time 和 random 用于控制请求延时，fake_useragent 用于生成随机 User-Agent。
2. scrape_ptj() 函数：封装了爬取 ptj.de 的逻辑。
3. headers：设置 User-Agent 请求头，模拟浏览器访问，这有助于避免被网站识别为爬虫并拒绝访问。
4. requests.get()：发送 GET 请求获取网页内容。timeout 参数设置了请求超时时间，raise_for_status() 会在请求失败时抛出异常。
5. BeautifulSoup(html_content, 'lxml')：使用 BeautifulSoup 解析获取到的 HTML 文本。'lxml' 是一个高性能的 HTML 解析器。
6. soup.find() 和 soup.find_all() / soup.select()：这是 BeautifulSoup 中最常用的查找元素的方法。它们允许您通过标签名、属性（如 class 或 id）来定位 HTML 元素。请注意：示例中的 **CSS** 选择器（**div.search-results, div.result-item, h3.item-title** 等）是假设的，您需要根据 **ptj.de** 网站的实际 **HTML** 结构，使用浏览器的开发者工具（通常按 **F12** 打开）来检查并确定正确的选择器。
7. 数据提取：通过 .get_text(strip=True) 获取元素的文本内容，['href'] 获取链接的 href 属性。
8. 生成唯一 **ID**：使用 hashlib.md5() 对项目标题、描述和链接的组合进行哈希，生成一个唯一的 MD5 值。这可以作为数据的唯一标识，用于后续的去重操作。
9. 构建 **DataFrame**：将提取到的数据存储在一个字典列表中，然后使用 pd.DataFrame() 转换为 Pandas DataFrame，便于后续的数据处理和分析。

3.5.2 爬取 **eu-startups.com** 示例

eu-startups.com 提供了投资者信息。我们将以其为例，演示如何爬取其数据。同样，在编写爬虫前，请检查 https://www.eu-startups.com/robots.txt（实际验证可能为空）。

创建一个名为 scraper_eustartups.py 的文件，并添加以下代码：

```
# scraper_eustartups.py

import requests
from bs4 import BeautifulSoup
import pandas as pd
import hashlib
import time
import random
from fake_useragent import UserAgent
```

```
ua = UserAgent()

def scrape_eustartups():
    """
    爬取 eu-startups.com 网站的投资者信息。
    请注意：HTML 选择器是示例性的，实际开发中需要根据网站的实时结构进行调整。
    """
    base_url = "https://www.eu-startups.com/"
    investor_url = "https://www.eu-startups.com/investor-location/?country=DE" # 示例：德国投资者页面

    headers = {
        'User-Agent': ua.random
    }

    print(f"正在爬取: {investor_url}")
    html_content = None
    try:
        response = requests.get(investor_url, headers=headers, timeout=10)
        response.raise_for_status()
        html_content = response.text
    except requests.exceptions.RequestException as e:
        print(f"请求失败: {e}")
        return pd.DataFrame()

    if not html_content:
        return pd.DataFrame()

    soup = BeautifulSoup(html_content, 'lxml')

    # 假设投资者信息在一个 class 为 'investor-list' 的 div 中，每个投资者是一个 class 为 'investor-card' 的 div
    # 同样，这只是一个示例，您需要检查实际的 HTML 结构
    investor_items = soup.find_all('div', class_='investor-card')
    if not investor_items:
        print("未找到任何投资者信息，请检查选择器或网页结构。")
        return pd.DataFrame()

    data = []
```

```
    for item in investor_items:
        name_tag = item.find('h3', class_='investor-name')
        location_tag = item.find('span', class_='investor-location')
        focus_tag = item.find('p', class_='investor-focus')
        link_tag = item.find('a', class_='investor-link')

        name = name_tag.get_text(strip=True) if name_tag else 'N/A'
        location = location_tag.get_text(strip=True) if location_tag else 'N/A'
        focus = focus_tag.get_text(strip=True) if focus_tag else 'N/A'
        link = link_tag['href'] if link_tag and 'href' in link_tag.attrs else 'N/A'

        unique_id =
hashlib.md5(f"{name}{location}{focus}{link}".encode('utf-8')).hexdigest()

        data.append({
            'id': unique_id,
            'name': name,
            'location': location,
            'focus': focus,
            'link': link,
            'source': 'eu-startups.com'
        })

    df = pd.DataFrame(data)
    print(f"从 {investor_url} 爬取到 {len(df)} 条数据。")
    return df

if __name__ == "__main__":
    eustartups_df = scrape_eustartups()
    if not eustartups_df.empty:
        print("\n爬取到的数据示例 (eu-startups.com):")
        print(eustartups_df.head())
        # eustartups_df.to_csv('eustartups_investors.csv', index=False,
encoding='utf-8-sig')
```

代码解释：

与 ptj.de 的爬虫类似，主要区别在于目标网站的 URL 和 HTML 结构。您需要根据

eu-startups.com 的实际网页结构来调整 find() 和 find_all() 方法中的选择器。

3.5.3 爬取 **foerderdatenbank.de** 示例

foerderdatenbank.de 是德国的一个资金数据库。这个网站可能包含表单提交和分页等更复杂的交互。我们将以其搜索结果页面为例进行爬取。

请注意，像 https://www.foerderdatenbank.de/SiteGlobals/FDB/Forms/Suche/Foederprogrammsuche_Formular.html?... 这样的 URL 通常是搜索结果页，其内容可能是通过表单提交生成的。对于初学者教程，我们建议先从简单的静态页面爬取开始，或者直接爬取搜索结果页的 URL（如果它是可直接访问的）。

检查 https://www.foerderdatenbank.de/robots.txt（实际验证可能为空）。

创建一个名为 scraper_foerderdatenbank.py 的文件，并添加以下代码：

```
# scraper_foerderdatenbank.py

import requests
from bs4 import BeautifulSoup
import pandas as pd
import hashlib
import time
import random
from fake_useragent import UserAgent

ua = UserAgent()

def scrape_foerderdatenbank():
    """
    爬取 foerderdatenbank.de 网站的资金项目信息。
    请注意：这个网站可能需要 POST 请求才能获取搜索结果，以下示例仅为 GET 请求演示。
    实际开发中需要根据网站的实时结构和请求方式进行调整。
    """
    base_url = "https://www.foerderdatenbank.de/"
    # 这是一个示例搜索结果页 URL，实际可能需要通过 POST 请求或更复杂的 URL 构建
    # 为了简化，我们假设可以直接访问这个 URL 获取数据
    search_result_url = "https://www.foerderdatenbank.de/FDB/DE/Home/home.html"
```

```
headers = {
    'User-Agent': ua.random
}

print(f"正在爬取: {search_result_url}")
html_content = None
try:
    response = requests.get(search_result_url, headers=headers, timeout=10)
    response.raise_for_status()
    html_content = response.text
except requests.exceptions.RequestException as e:
    print(f"请求失败: {e}")
    return pd.DataFrame()

if not html_content:
    return pd.DataFrame()

soup = BeautifulSoup(html_content, 'lxml')

# 假设资金项目列表在一个 class 为 'search-results-list' 的 ul 中,
# 每个项目是一个 class 为 'list-item' 的 li
# 同样，这只是一个示例，您需要检查实际的 HTML 结构
fund_items = soup.find_all('li', class_='list-item')
if not fund_items:
    print("未找到任何资金项目，请检查选择器或网页结构。")
    return pd.DataFrame()

data = []
for item in fund_items:
    title_tag = item.find('h4', class_='item-title')
    description_tag = item.find('p', class_='item-description')
    link_tag = item.find('a', class_='item-link')

    title = title_tag.get_text(strip=True) if title_tag else 'N/A'
    description = description_tag.get_text(strip=True) if description_tag else 'N/A'
    link = base_url + link_tag['href'] if link_tag and 'href' in link_tag.attrs else 'N/A'

    unique_id = hashlib.md5(f"{title}{description}{link}".encode('utf-8')).hexdigest()
```

```
            data.append({
                'id': unique_id,
                'title': title,
                'description': description,
                'link': link,
                'source': 'foerderdatenbank.de'
            })

        df = pd.DataFrame(data)
        print(f"从 {search_result_url} 爬取到 {len(df)} 条数据。")
        return df

if __name__ == "__main__":
    foerderdatenbank_df = scrape_foerderdatenbank()
    if not foerderdatenbank_df.empty:
        print("\n爬取到的数据示例 (foerderdatenbank.de):")
        print(foerderdatenbank_df.head())
        # foerderdatenbank_df.to_csv('foerderdatenbank_funds.csv', index=False,
encoding='utf-8-sig')
```

重要提示：对于 foerderdatenbank.de 这样的网站，其搜索结果可能需要通过表单提交（POST 请求）才能获取。如果直接访问 GET 请求的 URL 无法获取到数据，您可能需要：

1. 分析表单：使用浏览器开发者工具检查搜索表单的 HTML 结构，找到表单的 action URL 和 method（通常是 POST），以及所有输入字段的 name 属性。
2. 构造 **POST** 请求：使用 requests.post() 方法，将表单数据作为 data 参数传递。例如：

   ```
   # payload = {
   #     'input_field_name1': 'value1',
   #     'input_field_name2': 'value2'
   # }
   # response = requests.post(form_action_url, data=payload, headers=headers)
   ```

   这超出了本章初学者教程的范围，但了解其原理对您未来处理更复杂的网站爬取非常重要。

3.5.4 爬取 deutsche-digitale-bibliothek.de 示例

deutsche-digitale-bibliothek.de 是一个数字图书馆，其数据结构可能非常复杂。 该网站

明确提供了 **API**，因此我们应优先使用 **API** 进行数据获取。

API 使用策略：

1. 仔细阅读 **API** 文档：了解 API 的认证方式（通常需要申请 API Key）、请求频率限制、可用的数据端点 (endpoints)、查询参数、返回的数据格式（通常是 JSON）。
2. 获取 **API Key**：根据文档说明申请 API Key。
3. 构造 **API** 请求：使用 requests 库，根据 API 文档构造请求 URL 和参数。
4. 处理 **API** 响应：解析返回的 JSON 数据，提取所需字段。
5. 目标字段：根据 API 返回的数据结构确定。对于资助项目，可能不直接提供，但可以搜索与“Förderung", "Stipendium", "Projektfinanzierung”等相关的文献或元数据，间接获取线索。

创建一个名为 scraper_ddb.py 的文件，并添加以下代码（注意：这是一个基于通用 API 设计模式的推测，实际端点、参数和响应结构需要严格参照官方 API 文档）：

```
# scraper_ddb.py

import requests
import pandas as pd
import hashlib
import time
import random
import json
from fake_useragent import UserAgent

ua = UserAgent()

# 假设的 DDB API 配置，需要根据实际 API 文档和您的 API Key 进行替换
DDB_API_BASE_URL = "https://api.deutsche-digitale-bibliothek.de/v3" # 假设的 API 基地址
API_KEY = "YOUR_DDB_API_KEY" # !!! 需要替换为真实的 API Key !!!

def search_ddb_api(query_term, rows=10, offset=0):
    """
    通过 DDB API 搜索相关内容。
    :param query_term: 查询关键词
    :param rows: 每页返回结果数
    :param offset: 偏移量（用于分页 ）
    :return: JSON 格式的 API 响应数据或 None
    """
```

```
    endpoint = f"{DDB_API_BASE_URL}/search/index/search" # 假设的搜索端点
    params = {
        'query': query_term,
        'rows': rows,
        'offset': offset,
        'oauth_consumer_key': API_KEY # API Key 通常作为参数或在 Header 中传递
    }
    headers = {'Accept': 'application/json', 'User-Agent': ua.random}

    print(f"正在通过 DDB API 搜索: '{query_term}', offset: {offset}")
    try:
        response = requests.get(endpoint, params=params, headers=headers,
timeout=30)
        response.raise_for_status()
        print(f"DDB API 请求成功: {response.url}")
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"DDB API 请求失败: {e}")
        return None

def scrape_ddb(query_term="funding", max_results=100):
    """
    通过 DDB API 爬取数据。
    :param query_term: 搜索关键词，例如 'funding' 或 'künstliche intelligenz förderung'
    :param max_results: 最大爬取结果数
    :return: 包含爬取数据的 DataFrame
    """
    all_data = []
    current_offset = 0
    page_size = 50 # API 允许每页返回的最大数量 (假设)

    while len(all_data) < max_results:
        data = search_ddb_api(query_term, rows=page_size, offset=current_offset)

        if data and data.get('results') and data['results'][0].get('docs'):
            results_on_page = data['results'][0]['docs']
            if not results_on_page: # 当前页没有数据了
                break
```

```
            for doc in results_on_page:
                # 根据 DDB API 返回的实际 JSON 结构提取数据
                # 这是一个示例，您需要根据实际返回的 doc 结构进行调整
                unique_id = hashlib.md5(json.dumps(doc,
sort_keys=True).encode('utf-8')).hexdigest()
                title = doc.get('title', ['N/A'])[0] if isinstance(doc.get('title'), list) else
doc.get('title', 'N/A')
                description = doc.get('description', ['N/A'])[0] if
isinstance(doc.get('description'), list) else doc.get('description', 'N/A')
                link = doc.get('url', 'N/A') # 假设 'url' 字段包含链接

                all_data.append({
                    'id': unique_id,
                    'title': title,
                    'description': description,
                    'link': link,
                    'source': 'deutsche-digitale-bibliothek.de',
                    'original_data': json.dumps(doc) # 保存原始数据以供调试或未来分析
                })

            print(f"获取到 {len(results_on_page)} 条结果，总计 {len(all_data)} 条。")

            # 简化的分页逻辑：如果返回结果少于请求数量，可能意味着是最后一页
            if len(results_on_page) < page_size:
                break
            current_offset += page_size
            time.sleep(random.uniform(1, 3)) # 遵守 API 的请求频率限制
        else:
            print("未获取到更多结果或 API 响应格式不符。")
            break

    df = pd.DataFrame(all_data)
    print(f"从 DDB 爬取到 {len(df)} 条数据。")
    return df

if __name__ == "__main__":
    # 在运行前，请务必替换上面的 API_KEY 为您真实的 DDB API Key
    if API_KEY == "YOUR_DDB_API_KEY":
        print("警告：请在 scraper_ddb.py 文件中替换 YOUR_DDB_API_KEY 为您真实的
```

DDB API Key。")

```
    ddb_df = scrape_ddb(query_term="funding programs germany", max_results=20)
    if not ddb_df.empty:
        print("\n爬取到的数据示例 (deutsche-digitale-bibliothek.de):")
        print(ddb_df.head())
        # ddb_df.to_csv('ddb_results.csv', index=False, encoding='utf-8-sig')
```

重要提示：

- **HTML** 结构检查：上述所有爬虫代码中的 HTML 元素选择器（如 div.search-results, h3.item-title 等）都是基于假设的通用结构。在实际操作中，您必须使用浏览器的开发者工具（通常按 F12 打开）来检查每个目标网站的实际 HTML 结构，并相应地调整这些选择器。网页结构经常变化，因此爬虫需要定期维护和更新。
- 动态加载内容：如果网站内容是通过 JavaScript 动态加载的（例如，滚动到底部才加载更多内容，或者数据通过 AJAX 请求获取），那么简单的 requests + BeautifulSoup 可能无法获取到所有内容。对于这种情况，您可能需要使用更高级的工具，如 Selenium（模拟浏览器行为）或分析网站的 API 请求。
- 反爬机制：一些网站会采取更复杂的反爬机制，如 IP 限制、验证码、登录验证、请求频率限制等。对于这些情况，您可能需要引入代理 IP 池、打码平台、Cookie 管理、延迟请求等策略。这些超出了本初学者教程的范围，但了解它们的存在对您未来的爬虫开发至关重要。

3.6 整合爬虫模块

为了方便管理和调用，我们可以创建一个主爬虫文件，统一调度各个网站的爬虫函数。

创建一个名为 main_scraper.py 的文件：

```
# main_scraper.py

import pandas as pd
import time
import random
import logging

# 导入各个爬虫模块中的函数
from scraper_ptj import scrape_ptj
from scraper_eustartups import scrape_eustartups
from scraper_foerderdatenbank import scrape_foerderdatenbank
```

```
from scraper_ddb import scrape_ddb # 如果 DDB 网站有 API，请确保已正确配置 API Key

# 配置日志
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def run_all_scrapers():
    """
    运行所有已定义的爬虫，合并并去重数据。
    """
    all_data_frames = []

    # --- 运行 ptj.de 爬虫 ---
    logging.info("--- 正在运行 ptj.de 爬虫 ---")
    ptj_df = scrape_ptj()
    if not ptj_df.empty:
        all_data_frames.append(ptj_df)
        logging.info(f"ptj.de 爬虫完成，爬取到 {len(ptj_df)} 条数据。")
    else:
        logging.warning("ptj.de 爬虫未获取到数据。")
    time.sleep(random.uniform(2, 5)) # 增加随机延时，尊重网站

    # --- 运行 eu-startups.com 爬虫 ---
    logging.info("--- 正在运行 eu-startups.com 爬虫 ---")
    eustartups_df = scrape_eustartups()
    if not eustartups_df.empty:
        all_data_frames.append(eustartups_df)
        logging.info(f"eu-startups.com 爬虫完成，爬取到 {len(eustartups_df)} 条数据。")
    else:
        logging.warning("eu-startups.com 爬虫未获取到数据。")
    time.sleep(random.uniform(2, 5))

    # --- 运行 foerderdatenbank.de 爬虫 ---
    logging.info("--- 正在运行 foerderdatenbank.de 爬虫 ---")
    foerderdatenbank_df = scrape_foerderdatenbank()
    if not foerderdatenbank_df.empty:
        all_data_frames.append(foerderdatenbank_df)
        logging.info(f"foerderdatenbank.de 爬虫完成，爬取到 {len(foerderdatenbank_df)}
```

```
条数据。")
    else:
        logging.warning("foerderdatenbank.de 爬虫未获取到数据。")
    time.sleep(random.uniform(2, 5))

    # --- 运行 deutsche-digitale-bibliothek.de 爬虫 (API 优先) ---
    logging.info("--- 正在运行 deutsche-digitale-bibliothek.de 爬虫 ---")
    # 确保在 scraper_ddb.py 中已配置正确的 API_KEY
    if scraper_ddb.API_KEY == "YOUR_DDB_API_KEY":
        logging.error("DDB API Key 未配置！请检查 scraper_ddb.py 文件。跳过 DDB 爬取。
")
    else:
        ddb_df = scrape_ddb(query_term="funding programs europe", max_results=50)
# 示例关键词和数量
        if not ddb_df.empty:
            all_data_frames.append(ddb_df)
            logging.info(f"deutsche-digitale-bibliothek.de 爬虫完成，爬取到 {len(ddb_df)}
条数据。")
        else:
            logging.warning("deutsche-digitale-bibliothek.de 爬虫未获取到数据。")
    time.sleep(random.uniform(2, 5))

    if all_data_frames:
        # 合并所有爬取到的数据
        final_df = pd.concat(all_data_frames, ignore_index=True)

        # 进一步处理: 去重 (基于 id 列)，如果不同来源有相同 id 的数据，保留第一个
        # 注意：这里的 'id' 是由 hashlib 生成的，如果不同来源有相同标题+描述+链接，则 ID
会相同。
        # 对于不同来源可能内容相似但 ID 不同的情况，需要在数据清洗阶段做更高级的去
重（例如基于文本相似度）
        records_before_dedup = len(final_df)
        final_df.drop_duplicates(subset=['id'], keep='first', inplace=True)
        records_after_dedup = len(final_df)
        logging.info(f"去重前记录数: {records_before_dedup}, 去重后记录数:
{records_after_dedup}")

        logging.info(f"所有爬虫运行完毕，共爬取到 {len(final_df)} 条不重复的数据。")
        return final_df
```

```
    else:
        logging.info("没有爬取到任何数据。")
        return pd.DataFrame()

if __name__ == "__main__":
    combined_df = run_all_scrapers()
    if not combined_df.empty:
        print("\n合并后的数据示例:")
        print(combined_df.head())
        # 将合并后的数据保存到 CSV 文件，或传递给数据存储模块
        combined_df.to_csv('combined_funds_data.csv', index=False,
encoding='utf-8-sig')
        print("数据已保存到 combined_funds_data.csv")
```

代码解释：

1. 导入各个爬虫函数：从之前创建的各个爬虫文件中导入相应的函数。
2. run_all_scrapers() 函数：依次调用每个爬虫函数，并将返回的 DataFrame 添加到 all_data_frames 列表中。
3. pd.concat()：将 all_data_frames 列表中的所有 DataFrame 合并成一个大的 DataFrame。
4. drop_duplicates(subset=['id'], inplace=True)：根据 id 列进行去重操作，确保最终数据集中没有重复的资金项目。inplace=True 表示直接在原 DataFrame 上修改。
5. 数据保存：将最终合并去重后的数据保存到 combined_funds_data.csv 文件中。在实际项目中，这些数据将传递给数据存储与管理层。
6. 日志记录：引入 logging 模块，在关键步骤输出信息，帮助跟踪爬虫运行状态。
7. 延时与随机性：在每次爬虫调用之间增加了随机延时，以避免给网站服务器造成过大压力，并降低被反爬机制识别的风险。

通过本章的学习，您应该已经掌握了使用 Python 进行网络爬取的基本方法，包括 HTTP 请求、HTML/JSON 解析、数据提取和初步的数据结构化。在下一章中，我们将学习如何将这些爬取到的数据存储到数据库中，并进行有效的管理。

第四章：数据炼金：从原始数据到可用特征

上一章我们通过网络爬虫获取了大量的原始资助项目数据。然而，这些数据往往是“粗糙”的，包含噪声、格式不一、甚至有所缺失。本章将聚焦于“数据炼金”的过程，即利用 Pandas 等工具对原始数据进行深度清洗和转换，并进行特征工程，提取出对后续 AI 匹配模型有用

的特征。这是将原始数据转化为机器可理解、可利用的“精金”的关键步骤。

4.1 使用 Pandas 进行深度数据清洗与转换

Pandas 是 Python 中进行数据处理和分析的瑞士军刀。我们将利用它来完成以下关键任务：

4.1.1 加载多源数据

首先，使用 Pandas 的读取函数（如 pd.read_csv(), pd.read_json()）加载之前爬虫阶段保存的、来自不同数据源的标准化数据文件。如果数据分散在多个文件中，可以使用 pd.concat() 将它们合并成一个统一的 DataFrame。

```
# data_processing.py (部分代码)

import pandas as pd
import numpy as np
import re
import hashlib
from datetime import datetime

def load_and_combine_data(file_paths):
    """
    加载并合并来自不同 CSV 文件的爬取数据。
    :param file_paths: 包含各个 CSV 文件路径的列表。
    :return: 合并后的 Pandas DataFrame。
    """
    all_data_frames = []
    for fp in file_paths:
        try:
            df = pd.read_csv(fp, encoding='utf-8-sig') # 确保正确读取编码
            all_data_frames.append(df)
            print(f"成功加载文件: {fp}")
        except FileNotFoundError:
            print(f"警告: 文件未找到: {fp}。跳过此文件。")
        except Exception as e:
            print(f"加载文件 {fp} 时发生错误: {e}")

    if all_data_frames:
        combined_df = pd.concat(all_data_frames, ignore_index=True)
        print(f"合并后的数据共有 {combined_df.shape[0]} 条记录和
```

```
{combined_df.shape[1]} 个字段。")
        print("数据初步概览:")
        print(combined_df.head())
        print("\n数据信息:")
        combined_df.info()
        return combined_df
    else:
        print("没有可加载的数据。")
        return pd.DataFrame()

# 示例使用 (假设 'combined_funds_data.csv' 是上一章爬虫输出的文件)
# file_paths_to_process = ['combined_funds_data.csv'] # 实际项目中可能包含多个来源的单独文件
# combined_df = load_and_combine_data(file_paths_to_process)
```

4.1.2 处理缺失值 (Missing Values)

数据中经常存在缺失值（Pandas 中通常表示为 NaN - Not a Number）。处理缺失值是数据清洗的重要一步。

- 识别缺失值：使用 .isnull().sum() 可以快速查看每列中缺失值的数量。
  ```
  # if not combined_df.empty:
  #     print("\n各字段缺失值数量:")
  #     print(combined_df.isnull().sum())
  ```

- 处理策略：
  - 删除：
    - 如果某条记录的关键字段（如项目标题 title、核心描述 description）缺失，这条记录可能价值不大，可以考虑删除整行：
      ```
      # combined_df.dropna(subset=['title', 'description'], inplace=True)
      ```
    - 如果某一列大部分数据都缺失（例如超过 70%），且该列对分析不关键，可以考虑删除整列：
      ```
      # combined_df.drop(columns=['some_mostly_empty_column'], inplace=True)
      ```
  - 填充 **(Imputation)**：
    - 对于数值型字段（如资助金额 ），可以用 0、平均值、中位数或特定业务逻辑值填充：

```
# combined_df['funding_amount_numeric'].fillna(0, inplace=True)
# 或者
#
combined_df['funding_amount_numeric'].fillna(combined_df['funding_amount_numeric'].median(), inplace=True)
```

- 对于文本型字段（如申请条件），可以用特定字符串（如“未知”、“Not Specified”）或空字符串填充：
```
# combined_df['description'].fillna("无描述", inplace=True)
# combined_df['link'].fillna("无链接", inplace=True)
# combined_df['source'].fillna("未知来源", inplace=True)
```

选择何种策略取决于缺失数据的比例、字段的重要性以及业务需求。填充不当可能引入偏差。

4.1.3 数据类型转换与规范化

确保每个字段的数据类型正确且格式统一，对于后续的分析和模型训练至关重要。

- 日期/时间转换：将字符串格式的日期（如"2024-12-31", "31. Dez. 2024", "12/31/2024"）统一转换为 Pandas 的 datetime 对象。这使得日期比较和计算成为可能。
```
def clean_date_column(df, column_name):
    """
    尝试将指定列转换为日期时间类型，处理多种格式。
    :param df: DataFrame
    :param column_name: 日期列名
    :return: 处理后的 DataFrame
    """
    if column_name in df.columns:
        # 尝试多种常见的日期格式
        date_formats = [
            '%Y-%m-%d',      # 2024-12-31
            '%d.%m.%Y',      # 31.12.2024
            '%m/%d/%Y',      # 12/31/2024
            '%Y/%m/%d',      # 2024/12/31
            '%d %b %Y',      # 31 Dec 2024 (英文月份简写)
            '%d %B %Y'       # 31 December 2024 (英文月份全称)
        ]

        # 先尝试通用的 pd.to_datetime，errors='coerce' 会将无法转换的日期变为
```

```
NaT
        df[f'{column_name}_dt'] = pd.to_datetime(df[column_name], errors='coerce')

        # 对于仍然是 NaT 的值，尝试用特定格式解析
        for fmt in date_formats:
            mask = df[f'{column_name}_dt'].isna()
            df.loc[mask, f'{column_name}_dt'] = pd.to_datetime(df.loc[mask,
column_name], format=fmt, errors='coerce')

        print(f"\n日期转换后 {column_name}_dt 的数据类型:
{df[f'{column_name}_dt'].dtype}")
    return df

# 示例使用
# combined_df = combined_df.copy() # 避免 SettingWithCopyWarning
# combined_df['application_deadline'] = pd.Series([
#     "2025-12-31", "31.03.2026", "01/15/2025", "Invalid Date", np.nan
# ]) # 假设有这样一个列
# combined_df = clean_date_column(combined_df, 'application_deadline')
# print(combined_df[['application_deadline', 'application_deadline_dt']].head())
```

- 金额转换：资助金额通常以文本形式出现，包含货币符号（€, $ 等）、千位分隔符（, 或 . ）、小数点（. 或 ,）。需要将这些文本清洗并转换为数值类型（float 或 integer）。

```
def clean_funding_amount(amount_str):
    """
    清洗和转换资金金额字符串为数值型 (float)。
    处理货币符号、千位分隔符、小数点，并尝试提取数字。
    :param amount_str: 金额字符串
    :return: 浮点数金额或 None
    """
    if pd.isna(amount_str):
        return None
    s = str(amount_str).strip().lower()

    # 移除常见货币符号和文字
    s = s.replace('€', '').replace('eur', '').replace('usd', '').replace('$', '')
    s = s.replace('up to', '').replace('bis zu', '').replace('ca.', '').replace('approx.', '')
    s = s.replace('m', '000000').replace('k', '000') # 简单处理 M 和 K 单位
```

```
    # 处理千位分隔符和小数点
    # 欧洲常用格式: . 为千位分隔符, , 为小数点 (如 1.234.567,89)
    # 英文常用格式: , 为千位分隔符, . 为小数点 (如 1,234,567.89)

    # 尝试匹配数字, 可能包含千位分隔符和小数点
    # 这个正则表达式尝试匹配包含数字、逗号、点的序列
    match = re.search(r'\d[\d\.,]*', s)
    if not match:
        return None

    num_str = match.group(0)

    # 判断是欧洲格式还是英文格式, 并统一为英文格式 (使用 . 作为小数点, 无千位分隔符)
    if ',' in num_str and '.' in num_str:
        # 如果最后一个逗号在最后一个点之后, 认为是欧洲格式 (1.234,56)
        if num_str.rfind(',') > num_str.rfind('.'):
            num_str = num_str.replace('.', '').replace(',', '.')
        else: # 否则认为是英文格式 (1,234.56)
            num_str = num_str.replace(',', '')
    elif ',' in num_str: # 只有逗号, 假设是小数点 (例如 123,45)
        num_str = num_str.replace(',', '.')
    else: # 只有点或没有分隔符
        # 移除所有千位分隔符(点), 如果点是小数点, 则应该只保留一个
        # 这是一个简化的处理, 对于像 "1.000" (一千) 这样的情况可能会误判为 1
        # 更精确的需要结合上下文或模式识别, 这里我们假设点通常是千位分隔符
        if num_str.count('.') > 1: # 多个点通常是千位分隔符
            num_str = num_str.replace('.', '')
        # 如果只有一个点, 可能是小数点, 保留

    try:
        return float(num_str)
    except ValueError:
        return None

# 示例使用
# combined_df['funding_amount_text'] = pd.Series([
#     "€1.234.567,89", "1,000,000 EUR", "500.000", "50k", "2.5M", "No Info",
np.nan
```

```
# ])
# combined_df['funding_amount_numeric'] =
combined_df['funding_amount_text'].apply(clean_funding_amount)
# print("\n金额转换后 funding_amount_numeric 的描述性统计:")
# print(combined_df[['funding_amount_text',
'funding_amount_numeric']].head(7))
# print(combined_df['funding_amount_numeric'].describe())
```

- 文本数据规范化：
  - 统一大小写：通常将文本字段（如描述、标题）转换为全小写，便于后续的文本比较和特征提取。

    ```
    # if 'title' in combined_df.columns:
    #     combined_df['title_processed'] =
    combined_df['title'].astype(str).str.lower()
    # if 'description' in combined_df.columns:
    #     combined_df['description_processed'] =
    combined_df['description'].astype(str).str.lower()
    ```

  - 去除首尾空白：.str.strip()
- 文本内容清洗：文本数据（如项目描述）是信息的重要来源，但也可能包含许多“噪音”。
  - 去除 **HTML** 标签：如果爬取时未能完全清除 HTML 标签，可以使用正则表达式或 BeautifulSoup 再次清洗。

    ```
    def remove_html_tags(text):
        """使用正则表达式去除 HTML 标签。"""
        if pd.isna(text):
            return ""
        clean = re.compile('<.*?>')
        return re.sub(clean, '', str(text))

    # if 'description' in combined_df.columns:
    #     combined_df['description_no_html'] =
    combined_df['description'].apply(remove_html_tags)
    ```

  - 去除多余空白字符：将多个连续空格替换为单个空格，去除换行符、制表符等。

    ```
    def clean_whitespace(text):
        """去除多余空白字符（包括换行符、制表符），并替换为单个空格。"""
        if pd.isna(text):
            return ""
    ```

```
        text = re.sub(r'\s+', ' ', str(text)) # 将一个或多个空白字符替换为单个空格
        return text.strip()

    # if 'description_no_html' in combined_df.columns:
    #     combined_df['description_cleaned'] =
    combined_df['description_no_html'].apply(clean_whitespace)
    #     print("\n文本清洗后 (description_cleaned) 示例:")
    #     print(combined_df[['description', 'description_cleaned']].head())
```

- 处理特殊字符和编码问题：确保文本数据使用统一的编码（如 UTF-8）。替换或移除不必要的特殊字符（如非打印字符）。

4.1.4 数据去重

不同数据源可能包含相同的资助项目信息，或者爬虫可能重复抓取了某些记录。需要基于之前生成的 id 或几个关键字段的组合（如项目标题 + 资助机构 + 截止日期）进行精确去重。

```
def deduplicate_data(df):
    """
    基于 'id' 列进行去重。
    :param df: 需要去重的 DataFrame
    :return: 去重后的 DataFrame
    """
    if df.empty:
        print("DataFrame 为空，无需去重。")
        return df

    records_before_dedup = len(df)
    if 'id' in df.columns:
        df.drop_duplicates(subset=['id'], keep='first', inplace=True)
        records_after_dedup = len(df)
        print(f"\n去重前记录数: {records_before_dedup}, 去重后记录数:
{records_after_dedup}")
    else:
        print("未找到 'id' 列，无法进行基于 ID 的去重。")
    return df

# 示例使用
# combined_df = deduplicate_data(combined_df)
```

4.1.5 异常值检测与处理（初步）

异常值 (Outliers) 是指与数据集中其他观测值显著不同的数据点。它们可能由测量错误、数据输入错误或真实但极端的情况引起。

- 识别：对于数值型字段（如资助金额、员工人数要求），可以使用描述性统计 (describe()) 观察其分布（均值、标准差、最小值、最大值、四分位数）。绘制箱线图（Box Plot）或直方图可以直观地发现异常值。
- 简单处理策略：
  - 设定合理阈值：根据业务知识或统计方法（如 IQR 法则：Q1 - 1.5*IQR, Q3 + 1.5*IQR），将超出阈值的数据视为异常。
  - 处理方法：可以删除包含异常值的记录（如果异常值很少且确定是错误），或者用上限/下限值替换（Winsorization），或者将其标记为特殊值。

本项目初期，可以先进行观察，对于明显错误的极端值（如资助金额为负数或过大天文数字）进行修正或标记。

4.2 文本特征工程：让机器理解文字的奥秘

资助项目的核心信息大多蕴含在文本描述中。为了让机器学习模型能够处理这些文本数据，我们需要将其转换为数值形式的特征向量。这个过程称为文本特征工程。

4.2.1 文本预处理

这是特征提取前的重要步骤，旨在减少噪音，提取文本的核心语义。

- 分词 **(Tokenization)**：将连续的文本切分成有意义的词语单元 (tokens)。
  - 英文分词相对简单，可以按空格和标点符号切分。
  - 德语等语言存在复合词和复杂的屈折变化，需要专门的分词器。NLTK 和 spaCy 库都支持多种语言的分词。
- 去除停用词 **(Stop Words Removal)**：移除文本中频繁出现但对区分文档主题贡献不大的词语，如冠词（"the", "a", "der", "die", "das"）、介词（"in", "on", "mit"）、连词（"and", "or", "und"）等。NLTK 和 spaCy 都提供了常见语言的停用词列表。
- （可选）词干提取 **(Stemming)/**词形还原 **(Lemmatization)**：
  - 词干提取：将单词简化为其词干（如"running", "ran" -> "run"）。方法比较粗暴，可能产生无意义的词干。例如，NLTK 中的 PorterStemmer（英文）或 SnowballStemmer（支持德语等多种语言）。
  - 词形还原：将单词转换为其字典中的基本形式（词元，lemma），如"better" -> "good", "corpora" -> "corpus"。通常比词干提取更精确，但计算量也更大。spaCy 等语言模型通常包含高质量的词形还原器。

import nltk

```
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
# from nltk.stem.snowball import SnowballStemmer # 示例：德语词干提取器

# 首次使用 NLTK 资源需要下载，只运行一次即可
# try:
#     nltk.data.find('tokenizers/punkt')
# except nltk.downloader.DownloadError:
#     nltk.download('punkt')
# try:
#     nltk.data.find('corpora/stopwords')
# except nltk.downloader.DownloadError:
#     nltk.download('stopwords')

# 获取德语停用词
# german_stop_words = set(stopwords.words('german'))

# german_stemmer = SnowballStemmer('german') # 如果需要词干提取

def preprocess_text(text, language='german'):
    """
    对文本进行预处理：转小写，去除标点数字，分词，去除停用词。
    :param text: 原始文本
    :param language: 文本语言 ('german' or 'english')
    :return: 预处理后的文本字符串
    """
    if pd.isna(text):
        return ""
    text = str(text).lower() # 转小写

    # 去除标点符号和数字，仅保留字母词（包括德语特有字母）
    if language == 'german':
        text = re.sub(r'[^a-zA-ZäöüßÄÖÜ\s]', '', text)
        stop_words = set(stopwords.words('german'))
    else: # 默认为英文
        text = re.sub(r'[^a-zA-Z\s]', '', text)
        stop_words = set(stopwords.words('english'))

    tokens = word_tokenize(text, language=language) # 分词

    processed_tokens = []
    for token in tokens:
        if token not in stop_words and len(token) > 1: # 去除停用词和过短的词
            # token_stemmed = german_stemmer.stem(token) # (可选) 词干提取
            # processed_tokens.append(token_stemmed)
```

```
            processed_tokens.append(token) # 不使用词干提取

    return " ".join(processed_tokens)

# 示例使用
# if not combined_df.empty and 'description_cleaned' in combined_df.columns:
#     combined_df['processed_description'] =
combined_df['description_cleaned'].apply(lambda x: preprocess_text(x,
language='german'))
#     print("\n文本预处理后 (processed_description) 示例:")
#     print(combined_df[['description_cleaned', 'processed_description']].head())
```

4.2.2 特征表示方法：将文本向量化

机器学习模型不能直接处理原始文本，需要将文本转换为数值向量。

- **TF-IDF (Term Frequency-Inverse Document Frequency)**：一种经典的统计方法，用于评估一个词语对于一个文档集或一个语料库中的其中一份文档的重要程度。
  - 词频 **(TF)**：一个词在文档中出现的次数（通常会进行归一化）。
  - 逆文档频率 **(IDF)**：衡量一个词的普遍重要性。如果一个词在很多文档中都出现，其 IDF 值较低；反之较高。
  - **TF-IDF** 值：TF * IDF。一个词在特定文档中的 TF-IDF 值高，说明它在该文档中常出现，但在整个文档集中不常见，因此很可能能代表该文档的主题。
  - 实现：使用 sklearn.feature_extraction.text.TfidfVectorizer。

```
from sklearn.feature_extraction.text import TfidfVectorizer

def vectorize_text_tfidf(df, text_column='processed_description',
max_features=5000, min_df=5, max_df=0.7, ngram_range=(1,2)):
    """
    使用 TF-IDF 将文本列向量化。
    :param df: DataFrame
    :param text_column: 包含预处理文本的列名
    :param max_features: 构建词汇表时只考虑 TF-IDF 排序后前 N 个词
    :param min_df: 忽略在少于 N 个文档中出现的词 (可以是整数或比例)
    :param max_df: 忽略在超过 N 个文档 (或比例) 中出现的词 (去除过于普遍的词)
    :param ngram_range: 例如 (1,2) 表示同时考虑单个词和两个词的组合 (bigrams)
    :return: TF-IDF 稀疏矩阵和训练好的 TfidfVectorizer 对象
    """
    if df.empty or text_column not in df.columns or df[text_column].isnull().all():
        print(f"DataFrame 为空或文本列 '{text_column}' 不存在或全部为空，无法进行
TF-IDF 向量化。")
        return None, None

    corpus = df[text_column].fillna('').tolist() # 确保处理 None/NaN 值
```

```
    vectorizer = TfidfVectorizer(
        max_features=max_features,
        min_df=min_df,
        max_df=max_df,
        ngram_range=ngram_range
    )
    try:
        tfidf_matrix = vectorizer.fit_transform(corpus)
        print(f"\nTF-IDF 矩阵维度: {tfidf_matrix.shape}")
        # print("部分特征词 (词汇表):", vectorizer.get_feature_names_out()[:20])
        return tfidf_matrix, vectorizer
    except ValueError as e:
        print(f"TF-IDF 向量化失败: {e}. 可能语料库为空或不满足 min_df/max_df 条件。
")
        return None, None

# 示例使用
# tfidf_matrix, vectorizer = vectorize_text_tfidf(combined_df)
```

- （可选入门）词向量 **(Word Embeddings)**：
  - 概念：将词语映射到一个低维（如 50-300 维）的稠密向量空间中，使得语义相近的词在向量空间中的距离也相近。例如，"king" 和 "queen" 的向量会比较接近。
  - 经典模型：Word2Vec (Skip-gram, CBOW), Glove, FastText。
  - 工具：Gensim 库可以训练这些模型或加载预训练的词向量。
  - 应用：对于每个文档，可以将其所有词的词向量进行平均（Average Word Embeddings）或加权平均，得到文档的向量表示。更高级的方法是使用循环神经网络（RNN）或 Transformer 模型（如 BERT）来生成文档/句子级别的向量表示，这些方法能更好地捕捉词序和上下文信息，但对初学者来说较为复杂。本项目初期以 TF-IDF 为主。

4.3 结构化特征处理：挖掘元数据价值

除了文本描述，资助项目通常还包含一些结构化的元数据，如资助金额、申请截止日期、目标行业、地区等。这些特征对于匹配也非常重要。

4.3.1 从文本中提取结构化信息（如果需要）

有时，一些关键的结构化信息（如特定技术领域、目标公司规模的详细描述）可能仍隐藏在文本描述中，而未被爬虫直接提取为独立字段。这种情况下，可以尝试使用：

- 正则表达式 **(Regular Expressions)**：编写精确的正则表达式来从文本中匹配和提取这些信息。例如，从描述中提取“员工人数少于 XX 人”或“针对 XX 行业”。
- 关键词/规则匹配：定义一些关键词列表或简单规则，如果描述中包含某些关键词，则

赋予相应的结构化标签。

提取后，创建新的特征列存储这些信息。

4.3.2 数值型特征

- 例如：funding_amount_numeric（数值型资助金额），project_duration_months（项目时长，月）。
- 归一化/标准化：不同数值特征的量纲和取值范围可能差异很大（如金额可能从几千到几百万，时长可能从几个月到几年）。为了消除量纲影响，使不同特征在模型中具有可比性（尤其对于基于距离的算法如 SVM、KNN，或梯度下降优化的模型），通常需要进行归一化或标准化。
  - **Min-Max** 归一化 **(MinMaxScaler)**：将数据缩放到一个固定范围，通常是 [0,1]。

```
from sklearn.preprocessing import MinMaxScaler

def normalize_numeric_column(df, column_name):
    """
    对指定数值列进行 Min-Max 归一化。
    :param df: DataFrame
    :param column_name: 数值列名
    :return: 处理后的 DataFrame
    """
    if column_name in df.columns and not df[column_name].isnull().all():
        scaler = MinMaxScaler()
        # 确保处理缺失值，例如先填充
        df[f'{column_name}_scaled'] =
scaler.fit_transform(df[[column_name]].fillna(0))
        print(f"\n列 '{column_name}' 已进行 Min-Max 归一化。")
    else:
        print(f"列 '{column_name}' 不存在或全部为空，跳过归一化。")
    return df

# 示例使用
# if not combined_df.empty and 'funding_amount_numeric' in
combined_df.columns:
#     combined_df = normalize_numeric_column(combined_df,
'funding_amount_numeric')
#     print(combined_df[['funding_amount_numeric',
'funding_amount_numeric_scaled']].head())
```

- 标准化 **(StandardScaler)**：将数据转换为均值为 0，标准差为 1 的分布。对异常值不那么敏感。

```
from sklearn.preprocessing import StandardScaler

def standardize_numeric_column(df, column_name):
    """
    对指定数值列进行标准化 (Z-score normalization)。
    :param df: DataFrame
    :param column_name: 数值列名
    :return: 处理后的 DataFrame
    """
    if column_name in df.columns and not df[column_name].isnull().all():
        scaler = StandardScaler()
        df[f'{column_name}_standardized'] =
scaler.fit_transform(df[[column_name]].fillna(0))
        print(f"\n列 '{column_name}' 已进行标准化。")
    else:
        print(f"列 '{column_name}' 不存在或全部为空，跳过标准化。")
    return df
# 示例使用
# if not combined_df.empty and 'funding_amount_numeric' in
combined_df.columns:
#     combined_df = standardize_numeric_column(combined_df,
'funding_amount_numeric')
#     print(combined_df[['funding_amount_numeric',
'funding_amount_numeric_standardized']].head())
```

4.3.3 类别型特征

- 例如：source（如联邦、州、欧盟），target_industry（如 IT、生物技术、 能源）。
- 机器学习模型通常不能直接处理文本形式的类别特征，需要将其转换为数值表示。
  - 独热编码 **(One-Hot Encoding)**：为每个类别创建一个新的二元（0 或 1）特征列。如果一个类别特征有 K 个不同取值，独热编码后会产生 K 个新特征。适用于名义类别（类别间无序）。

```
def one_hot_encode_column(df, column_name):
    """
    对指定类别列进行独热编码。
    :param df: DataFrame
    :param column_name: 类别列名
```

```
    :return: 包含独热编码特征的 DataFrame
    """
    if column_name in df.columns:
        # 填充缺失值，否则 get_dummies 会为 NaN 也创建一个类别
        df[column_name] = df[column_name].fillna('未知')
        df_encoded = pd.get_dummies(df, columns=[column_name],
prefix=column_name)
        print(f"\n列 '{column_name}' 已进行独热编码。")
        return df_encoded
    else:
        print(f"列 '{column_name}' 不存在，跳过独热编码。")
        return df

# 示例使用
# combined_df_encoded = one_hot_encode_column(combined_df, 'source')
# print(combined_df_encoded.filter(like='source_').head())
```

- 标签编码 **(Label Encoding)**：将每个类别映射为一个整数（如 0,1,2,...）。适用于有序类别（如小、中、大），或者某些树模型（如决策树、随机森林）可以直接处理整数编码的类别特征。

```
from sklearn.preprocessing import LabelEncoder

def label_encode_column(df, column_name):
    """
    对指定类别列进行标签编码。
    :param df: DataFrame
    :param column_name: 类别列名
    :return: 处理后的 DataFrame
    """
    if column_name in df.columns:
        label_encoder = LabelEncoder()
        df[f'{column_name}_encoded'] =
label_encoder.fit_transform(df[column_name].astype(str).fillna(''))
        print(f"\n列 '{column_name}' 已进行标签编码。")
    else:
        print(f"列 '{column_name}' 不存在，跳过标签编码。")
    return df
# 示例使用
# combined_df = label_encode_column(combined_df,
```

```
    'company_age_category') # 假设有类别型公司年龄
    # print(combined_df[['company_age_category',
    'company_age_category_encoded']].head())
```

4.4 构建最终特征集：为模型输送“弹药”

在完成上述所有处理后，我们需要将所有有用的特征整合起来，形成最终的特征集，用于后续的模型评估和 AI 系统。

4.4.1 整合特征

- 将文本特征（如 TF-IDF 矩阵的稀疏表示或其降维后的稠密表示）与处理好的结构化特征（数值型、编码后的类别型）横向拼接起来。
- 如果使用 TF-IDF 稀疏矩阵，可以直接与 Pandas DataFrame 中的其他数值特征拼接（需要先将 DataFrame 转换为 NumPy 数组或稀疏矩阵，并确保行对齐）。
  scipy.sparse.hstack 可用于拼接稀疏矩阵。

```
from scipy.sparse import hstack, csr_matrix
import pickle

def combine_features(tfidf_matrix, df_structured, structured_cols_to_use):
    """
    组合 TF-IDF 文本特征和结构化特征。
    :param tfidf_matrix: TF-IDF 稀疏矩阵
    :param df_structured: 包含结构化特征的 DataFrame (确保其索引与 tfidf_matrix 对应)
    :param structured_cols_to_use: 结构化特征列名列表
    :return: 最终特征矩阵 (稀疏矩阵)
    """
    if tfidf_matrix is None or df_structured.empty:
        print("TF-IDF 矩阵或结构化 DataFrame 为空，无法组合特征。")
        return None

    # 提取结构化特征，并处理可能的 NaN 值 (填充为 0 或其他适当值)
    # 转换为 NumPy 数组
    structured_features_array = df_structured[structured_cols_to_use].fillna(0).values

    # 确保 structured_features_array 也是稀疏矩阵，以便与 tfidf_matrix 拼接
    # 或者直接拼接 NumPy 数组，如果 tfidf_matrix 转换为稠密
    # 但通常建议保持稀疏以节省内存
```

```
    structured_features_sparse = csr_matrix(structured_features_array)

    if structured_features_sparse.shape[0] != tfidf_matrix.shape[0]:
        print("错误: TF-IDF 矩阵和结构化特征的行数不匹配。")
        return None

    final_feature_matrix = hstack([tfidf_matrix, structured_features_sparse])
    print(f"最终特征矩阵维度: {final_feature_matrix.shape}")
    return final_feature_matrix

# 完整的数据处理和特征工程流程
def run_data_pipeline(file_paths):
    print("\n--- 启动数据处理与特征工程管道 ---")
    df = load_and_combine_data(file_paths)
    if df.empty:
        return None, None, None, None

    # 1. 填充关键缺失值
    df['title'] = df['title'].fillna("无标题")
    df['description'] = df['description'].fillna("无描述")
    df['link'] = df['link'].fillna("无链接")
    df['source'] = df['source'].fillna("未知来源")

    # 2. 文本清洗
    df['description_no_html'] = df['description'].apply(remove_html_tags)
    df['description_cleaned'] = df['description_no_html'].apply(clean_whitespace)
    df['processed_description'] = df['description_cleaned'].apply(lambda x:
preprocess_text(x, language='german')) # 假设德语

    # 3. 金额转换 (假设项目数据中存在 'funding_amount_text' 或类似列)
    # 您需要根据实际爬取到的字段名调整或创建该列
    # if 'funding_amount_text' not in df.columns: # 检查是否存在
    #     df['funding_amount_text'] = '' # 如果不存在，创建空列，避免报错

    # df['funding_amount_numeric'] =
df['funding_amount_text'].apply(clean_funding_amount)
    # 暂时跳过金额转换，因为示例爬虫没有生成此列
    df['funding_amount_numeric'] = 0.0 # 默认为0.0
```

```
    # 4. 日期转换 (假设项目数据中存在 'application_deadline' 或类似列)
    # df = clean_date_column(df, 'application_deadline')
    # 暂时跳过日期转换，因为示例爬虫没有生成此列
    df['application_deadline_dt'] = pd.NaT # Not a Time

    # 5. TF-IDF 向量化
    tfidf_matrix, vectorizer = vectorize_text_tfidf(df, 'processed_description')
    if tfidf_matrix is None:
        print("TF-IDF 向量化失败，无法继续构建特征。")
        return None, None, None, None

    # 6. 数值特征归一化 (使用 funding_amount_numeric_scaled 作为示例结构化特征)
    df = normalize_numeric_column(df, 'funding_amount_numeric')
    df['funding_amount_numeric_scaled'] = df['funding_amount_numeric_scaled'].fillna(0) # 填充 NaN

    # 7. 类别特征独热编码 (使用 source 作为示例结构化特征)
    df_final = one_hot_encode_column(df, 'source')

    # 准备用于组合的结构化特征列
    # 注意: 这里需要确保 df_final 包含了所有独热编码后的列
    # 比如 df_final.filter(like='source_').columns 会返回 'source_ptj.de', 'source_eu-startups.com' 等
    structured_cols_to_use = ['funding_amount_numeric_scaled'] + list(df_final.filter(like='source_').columns)

    # 确保所有结构化特征列都存在，如果不存在则创建并填充默认值
    for col in structured_cols_to_use:
        if col not in df_final.columns:
            df_final[col] = 0 # 填充 0 或其他合适默认值

    # 确保 TF-IDF 矩阵的行顺序与 df_final 匹配 (这里假设它们通过 index 隐式匹配)
    # 如果中间有行被删除，需要重新索引或同步处理
    final_feature_matrix = combine_features(tfidf_matrix, df_final, structured_cols_to_use)

    print("\n--- 数据处理与特征工程管道完成 ---")
    return final_feature_matrix, df_final, vectorizer # df_final 包含原始数据和处理后
```

```
的结构化特征

# 示例运行数据处理管道
# if __name__ == "__main__":
#     # 假设 'combined_funds_data.csv' 存在
#     input_files = ['combined_funds_data.csv']
#     final_features, processed_df, tfidf_vectorizer = run_data_pipeline(input_files)

#     if final_features is not None:
#         # 保存特征矩阵和 TF-IDF Vectorizer 以供后续使用
#         # 可以使用 pickle 保存稀疏矩阵和 vectorizer 对象
#         with open('final_features.pkl', 'wb') as f:
#             pickle.dump(final_features, f)
#         with open('tfidf_vectorizer.pkl', 'wb') as f:
#             pickle.dump(tfidf_vectorizer, f)
#
#         print("最终特征矩阵和 TF-IDF 向量化器已保存。")
#     else:
#         print("未能生成最终特征矩阵。")
```

4.4.2 特征选择（可选）

如果特征维度过高，可以考虑使用特征选择技术（如基于统计检验、基于模型的方法）来移除不重要或冗余的特征，以降低模型复杂度、减少过拟合风险、提升训练效率。

4.4.3 存储与管理

- 将最终处理好的特征集（X）与对应的标签（y，用于模型评估）分开存储。
- 可以使用 Pandas 的 to_pickle() 方法保存 DataFrame（包含特征和标签），或者将特征矩阵和标签数组分别保存为 NumPy 的 .npy 或 .npz 文件，或 HDF5 等格式。
- 同时保存用于文本向量化的 TfidfVectorizer 对象（使用 pickle 库），以便在处理新的用户输入或新的项目数据时，能以相同的方式进行转换。

至此，我们已经将原始的、混杂的数据“提炼”成了机器可以学习和利用的结构化特征。这是构建智能系统的关键一步，数据的质量直接决定了后续模型性能的上限。

关键要点总结

- 数据清洗核心：使用 Pandas 处理缺失值、转换数据类型（日期、金额）、清洗文本内容（去 HTML、去多余空白）、数据去重、初步异常值处理。
- 文本特征工程：包括文本预处理（分词、去停用词、可选的词干/词形还原）和特征表示（TF-IDF 是常用起点）。

- 结构化特征处理：对数值型特征进行归一化/标准化，对类别型特征进行独热编码或标签编码。
- 特征集构建：整合文本特征和结构化特征，形成最终的特征矩阵。
- 工具保存：保存用于数据转换的工具（如 TfidfVectorizer），以便在新数据上应用相同的转换。

第五章：核心基石：构建科学的模型评估流程

在投入大量精力开发复杂的 AI 匹配模型之前，建立一个科学、可复现的模型评估流程至关重要。正如用户补充信息中强调的："在探索任何机器学习方法之前，首先要解决的是模型评估流程。"这个流程不仅能帮助我们衡量当前系统的性能，还能指导我们如何设计和训练更好的模型。本章将详细阐述模型评估的重要性、关键指标、数据集准备以及如何搭建一个自动化的评估"裁判系统"，并建立性能基线。

5.1 模型评估的重要性与基本原则

5.1.1 为何评估

- 量化性能：将"好"或"坏"的主观感受转化为客观的、可度量的指标。
- 发现问题：通过评估结果，可以了解模型在哪些方面表现不佳，从而有针对性地进行优化。
- 指导优化：比较不同模型、不同参数设置或不同特征集下的性能，选择最优方案。
- 避免盲目开发：没有评估，开发过程就像在黑暗中摸索，不知道方向是否正确。
- 建立信任：可靠的评估结果是证明系统有效性的基础。

5.1.2 评估什么

- 针对本项目的核心目标——智能发现合适的资助项目，我们需要评估系统在以下方面的能力：
  - 相关性判断：系统推荐的项目是否真的与用户的需求（画像）相关？
  - 覆盖度：系统是否能找到大部分相关的项目，而没有遗漏重要的机会？
  - 效率：系统给出结果的速度如何？（本项目初期更关注匹配质量）

5.1.3 基本原则

- 客观性：使用业界公认的、标准化的评估指标和独立的测试数据集。
- 可复现性：评估流程应该是固定的，对于相同的模型和数据，多次评估应得到相同的结果。
- 对比性：将当前模型的性能与一个或多个基线模型（Baseline）或其他候选模型进行比较，以判断其相对优劣。
- 业务相关性：选择的评估指标应能反映真实的业务价值。例如，对于某些场景，漏掉一个重要机会的代价可能远高于推荐一个不相关项目。

5.2 定义项目成功指标（Metrics for Success）：量化"好"的标准

对于资助项目匹配/搜索这类信息检索和推荐任务，常用的评估指标主要围绕分类的准确性展开。我们将"项目是否匹配用户画像"视为一个二分类问题（匹配/不匹配）。

5.2.1 核心分类指标

假设我们有以下四种情况（通常通过混淆矩阵展示）：

预测 \ 实际	正例（**Relevant**）	反例（**Not Relevant**）
预测为正例	True Positives (TP)	False Positives (FP)
预测为反例	False Negatives (FN)	True Negatives (TN)

- True Positives (TP)：实际相关，系统也判断为相关。（正确命中）
- False Positives (FP)：实际不相关，但系统判断为相关。（误报，Type I Error）
- False Negatives (FN)：实际相关，但系统判断为不相关。（漏报，Type II Error）
- True Negatives (TN)：实际不相关，系统也判断为不相关。（正确排除）

图**5-1:** 混淆矩阵示意图

- 精确率 (Precision)：在所有被系统判断为“相关”的项目中，真正相关的比例。
  Precision=TP+FPTP
  - 业务含义：高精确率意味着系统推荐的结果质量高，用户看到的错误推荐少，减少了用户的筛选成本。如果 FP 的代价很高（例如，给用户推荐大量不相关的项目会严重影响用户体验），则精确率很重要。
- 召回率 (Recall)/真正例率 (True Positive Rate, TPR)/灵敏度 (Sensitivity)：在所有实际“相关”的项目中，被系统成功找回的比例。
  Recall=TP+FNTP
  - 业务含义：高召回率意味着系统能尽可能多地找出所有相关的项目，减少遗漏重要机会的风险。如果 FN 的代价很高（例如，错过一个关键的资助项目可能导致重大损失），则召回率很重要。
- F1 分数 (F1-Score)：精确率和召回率的调和平均数，是一个综合评价指标。当精确率和召回率都较高时，F1 分数也较高。
  F1=2×Precision+RecallPrecision×Recall
  - 业务含义：F1 分数试图在精确率和召回率之间取得平衡。在很多场景下，我们希望两者都表现良好，此时 F1 是一个很好的综合指标。
- 准确率 (Accuracy)：系统正确分类（包括正确判断为相关和正确判断为不相关）的项目占总项目数的比例。
  Accuracy=TP+FP+TN+FNTP+TN
  - 注意：在类别不平衡的数据集（例如，相关的资助项目远少于不相关的项目）中，准确率可能具有误导性。一个总是预测“不相关”的简单模型也可能获得很高的准确率，但它没有任何实际价值。因此，对于本项目，精确率、召回率和 F1 分数通常是更重要的指标。

5.2.2 指标选择依据

在项目的不同阶段或针对不同用户群体，对指标的侧重可能不同。例如：

- 对于一个急需资金的初创企业，可能更看重召回率，希望能看到所有潜在的机会，即使其中混杂一些不太相关的。
- 对于一个时间宝贵的咨询顾问，可能更看重精确率，希望系统推荐的结果都是高度相关的，以减少筛选时间。

F1 分数提供了一个平衡的视角。在实际应用中，通常会同时监控这几个核心指标。

5.3 准备评估数据集（Ground Truth）：评估的“标准答案”

要评估系统的性能，我们需要一个“标准答案”——即一个已经标注好“项目-用户画像”是否匹配的数据集。这个数据集的质量直接影响评估结果的可靠性。

5.3.1 构建“真实”标签数据 (Ground Truth)

- 挑战：获取大规模、高质量的“项目-用户画像”匹配标签是非常困难且耗时的工作。理想情况下，我们需要大量的用户画像和对应的资助项目，并由领域专家判断它们之间的匹配程度。
- 初期可行方法：
  1. 人工标注 **(Human Annotation)**：
     - 选取一部分具有代表性的资助项目（例如，从爬取的数据中随机抽取几百到几千条）。
     - 设计若干典型的用户画像（例如，不同行业、不同规模、不同发展阶段的企业）。
     - 由项目成员或邀请领域专家（如果有条件）对每一对“项目-用户画像”进行标注，判断其是否匹配（例如，标记为“相关”/“不相关”，或者使用一个相关性评分，如 1-5 分）。
     - 标注标准需要提前定义清晰，以保证标注的一致性。
  2. 利用历史数据/用户反馈（未来展望）：如果系统上线后有用户使用，可以收集用户的行为数据作为隐式反馈。例如，用户点击查看了哪些推荐项目、申请了哪些项目，这些都可以作为正向标签的来源。用户跳过或标记为不相关的项目可以作为负向标签。
  3. 启发式规则生成伪标签（谨慎使用）：可以基于一些简单的规则生成初步的标签。例如，如果一个资助项目的描述中明确提到了“生物技术初创公司”，而一个用户画像也正好是“生物技术行业，成立少于2年”，则可以初步标记这对组合为“相关”。这种方法生成的标签质量不高，可能引入偏差，但可以作为冷启动阶段的一种辅助手段。

5.3.2 数据集划分：保证评估的公正性

为了客观评估模型的泛化能力（即在未见过的数据上的表现），需要将标注好的数据集划分为几个互不相交的子集：

- 训练集 **(Training Set)**：用于训练机器学习模型（如果采用机器学习方案）。模型从这

部分数据中学习模式。通常占总数据集的 60%-80%。

- 验证集 **(Validation Set)**：用于调整模型的超参数（例如，决策树的深度、正则化参数等）和进行模型选择（比较不同模型的性能）。通常占 10%-20%。
- 测试集 **(Test Set)**：在模型训练和调优完成后，用于最终评估模型/系统的性能。测试集的数据严禁用于训练或调参过程，否则评估结果会过于乐观，不能反映模型在真实未知数据上的表现。通常占 10%-20%。

划分方法：可以使用 sklearn.model_selection.train_test_split 函数。为了保证划分后各数据集中类别比例与原始数据集相似（尤其在类别不平衡时），可以使用 stratify 参数。

```
# evaluation_data_prep.py

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import pickle

def prepare_evaluation_datasets(features_matrix, labels, test_size=0.2,
validation_size=0.25, random_state=42):
    """
    划分特征矩阵和标签为训练集、验证集和测试集。
    :param features_matrix: 最终特征矩阵 (例如 TF-IDF 稀疏矩阵)
    :param labels: 对应的标签数组 (0 或 1)
    :param test_size: 测试集占总体的比例
    :param validation_size: 验证集占训练验证集的比例 (例如 0.25 表示从 80% 中再分
25%，即总体的 20%)
    :param random_state: 随机种子，用于保证划分结果可复现
    :return: X_train, X_val, X_test, y_train, y_val, y_test
    """
    if features_matrix is None or labels is None or len(labels) == 0:
        print("特征矩阵或标签为空，无法划分数据集。")
        return None, None, None, None, None, None

    print(f"原始数据集大小: {features_matrix.shape[0]}")

    # 第一次划分：分出测试集
    X_train_val, X_test, y_train_val, y_test = train_test_split(
        features_matrix, labels, test_size=test_size, random_state=random_state,
stratify=labels
    )
```

```
    print(f"训练+验证集大小: {X_train_val.shape[0]}, 测试集大小: {X_test.shape[0]}")

    # 第二次划分：从剩余的训练验证集中分出验证集
    X_train, X_val, y_train, y_val = train_test_split(
        X_train_val, y_train_val, test_size=validation_size, random_state=random_state,
stratify=y_train_val
    )
    print(f"训练集大小: {X_train.shape[0]}, 验证集大小: {X_val.shape[0]}")

    return X_train, X_val, X_test, y_train, y_val, y_test

# 示例使用 (假设 final_features.pkl 和 labels.pkl 已经生成)
# if __name__ == "__main__":
#     # 模拟加载特征矩阵和标签 (实际需要从文件加载)
#     try:
#         with open('final_features.pkl', 'rb') as f:
#             final_features = pickle.load(f)
#
#         # 假设标签是手动创建或从某个 CSV/数据库中加载的
#         # 为了演示，我们在这里模拟一个简单的标签数组
#         # 实际项目中，这些标签需要人工标注生成
#         num_samples = final_features.shape[0]
#         # 假设 10% 的项目是相关的 (标签为 1)，90% 是不相关的 (标签为 0)
#         simulated_labels = np.random.choice([0, 1], size=num_samples, p=[0.9, 0.1])
#         labels = simulated_labels
#         print(f"模拟标签分布: {np.bincount(labels)}")

#         X_train, X_val, X_test, y_train, y_val, y_test =
prepare_evaluation_datasets(final_features, labels)

#         if X_train is not None:
#             # 保存划分好的数据集，以便后续模型训练和评估使用
#             with open('X_train.pkl', 'wb') as f: pickle.dump(X_train, f)
#             with open('X_val.pkl', 'wb') as f: pickle.dump(X_val, f)
#             with open('X_test.pkl', 'wb') as f: pickle.dump(X_test, f)
#             with open('y_train.pkl', 'wb') as f: pickle.dump(y_train, f)
#             with open('y_val.pkl', 'wb') as f: pickle.dump(y_val, f)
#             with open('y_test.pkl', 'wb') as f: pickle.dump(y_test, f)
#             print("\n训练集、验证集和测试集已保存。")
```

```
#     except FileNotFoundError:
#         print("请确保 'final_features.pkl' 文件已生成。")
#     except Exception as e:
#         print(f"准备评估数据集时发生错误: {e}")
```

5.3.3 交叉验证 (Cross-Validation) 原理

当数据集规模较小，或者想更稳健地评估模型性能、减少因单次划分带来的偶然性时，可以使用交叉验证。

- **K-Fold** 交叉验证：最常用的交叉验证方法。将训练集（或整个数据集，如果不单独分验证集）平均划分为 K 个互斥的子集（folds）。进行 K 轮评估：每一轮，选择其中一个子集作为验证集，其余 K-1 个子集作为训练集训练模型，然后在该验证集上评估模型。最终的性能指标是 K 轮评估结果的平均值。
- 优点：更充分地利用了数据，评估结果更稳定可靠。
- 实现：sklearn.model_selection.KFold 或更方便的 sklearn.model_selection.cross_val_score。

5.4 搭建模型/系统评估流水线：自动化的“裁判系统”

为了方便、一致地评估不同模型或系统版本的性能，我们需要编写可复用的评估脚本或函数。这个流水线接收系统的预测结果和真实的标签数据作为输入，输出预定义的各项评估指标。

我们将使用 sklearn.metrics 模块中提供的函数来计算指标：

- accuracy_score
- precision_score
- recall_score
- f1_score
- classification_report（一次性输出精确率、召回率、F1 分数和支持数）
- confusion_matrix（生成混淆矩阵 ）

创建一个名为 evaluation.py 的文件：

```
# evaluation.py

from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score,
classification_report, confusion_matrix
import numpy as np
import logging
```

```
# 配置日志
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def evaluate_matching_system(y_true, y_pred_labels, model_name="System"):
    """
    评估匹配系统的性能（假设是二分类: 0 为不匹配, 1 为匹配）。
    :param y_true: 真实标签列表或数组 (0 或 1)
    :param y_pred_labels: 系统预测的标签列表或数组 (0 或 1)
    :param model_name: 用于打印报告的模型/系统名称
    :return: 包含各项评估指标的字典
    """
    if not isinstance(y_true, (list, np.ndarray)) or not isinstance(y_pred_labels, (list, np.ndarray)):
        logging.error("输入 y_true 和 y_pred_labels 必须是列表或 NumPy 数组。")
        return None

    if len(y_true) != len(y_pred_labels):
        logging.error(f"真实标签和预测标签的长度不一致: {len(y_true)} vs {len(y_pred_labels)}")
        return None

    if len(y_true) == 0:
        logging.warning("输入数据为空，无法进行评估。")
        return {"precision": 0, "recall": 0, "f1": 0, "accuracy": 0, "report": "No data to evaluate."}

    # zero_division=0: 当分母为 0 时，指标返回 0 而不是警告或错误。可以设为 1 或 'warn'。
    precision = precision_score(y_true, y_pred_labels, zero_division=0)
    recall = recall_score(y_true, y_pred_labels, zero_division=0)
    f1 = f1_score(y_true, y_pred_labels, zero_division=0)
    accuracy = accuracy_score(y_true, y_pred_labels)

    # target_names 用于在分类报告中显示类别名称
    report = classification_report(y_true, y_pred_labels, zero_division=0, target_names=['Not Relevant (0)', 'Relevant (1)'])
    cm = confusion_matrix(y_true, y_pred_labels)
```

```
    print(f"\n--- 评估报告: {model_name} ---")
    print(f"精确率 (Precision): {precision:.4f}")
    print(f"召回率 (Recall): {recall:.4f}")
    print(f"F1 分数 (F1-Score): {f1:.4f}")
    print(f"准确率 (Accuracy): {accuracy:.4f}")
    print("\n分类报告 (Classification Report):\n", report)
    print("\n混淆矩阵 (Confusion Matrix):\n", cm)
    print(f"                 Predicted Not Relevant | Predicted Relevant")
    print(f"Actual Not Relevant: {cm[0,0]:<20} | {cm[0,1]:<20}")
    print(f"Actual Relevant:     {cm[1,0]:<20} | {cm[1,1]:<20}")
    print("------------------------------------------------")

    return {
        "precision": precision,
        "recall": recall,
        "f1": f1,
        "accuracy": accuracy,
        "report_str": report, # report_str for string representation
        "confusion_matrix": cm.tolist() # cm.tolist() for JSON serializable
    }

# 示例使用:
# if __name__ == "__main__":
#     # 假设这是一些真实的标签和预测结果
#     y_true_example = [0, 1, 0, 1, 0, 1, 0, 0, 1, 1]
#     y_pred_example = [0, 0, 1, 1, 0, 1, 0, 1, 1, 0] # 假设这是某个系统的预测结果

#     evaluation_results = evaluate_matching_system(y_true_example, y_pred_example,
model_name="MyFirstMatchingSystem")
#     if evaluation_results:
#         print(f"\nF1 Score from dict: {evaluation_results['f1']:.4f}")

#     # 测试空数据
#     evaluate_matching_system([], [], model_name="Empty Data Test")

#     # 测试长度不一致
#     evaluate_matching_system([0,1], [0,1,0], model_name="Length Mismatch Test")
```

这个函数可以被反复调用，用于评估不同版本的系统或不同模型的性能，确保评估过程的一致性。

5.5 建立基线模型 (Baseline Models) 与评估：衡量进步的起点

在开发复杂的 AI 系统之前，建立一个或多个简单的基线模型并评估其性能至关重要。基线模型为我们提供了一个性能的"底线"或参照点。如果后续开发的复杂 AI 模型性能还不如简单的基线，那么就需要反思 AI 模型的有效性或实现方式。

5.5.1 为何需要基线

- 衡量进步：任何新模型的性能提升都应该相对于基线来衡量。
- 验证价值：证明复杂模型确实带来了额外的价值，而不是"杀鸡用牛刀"。
- 快速迭代：基线模型实现简单，可以快速得到初步结果，帮助理解数据和问题。

5.5.2 简单启发式方法作为基线

1. 随机推荐/匹配 **(Random Baseline)**：
   - 原理：对于给定的用户画像，随机从项目库中推荐 N 个项目；或者对于每个"项目-用户画像"对，随机判断其是否匹配（例如，以 50% 的概率判断为相关）。
   - 实现：使用 numpy.random.choice 或 random.random。
   - 这是性能的最低下限，任何有意义的系统都应该显著优于它。
2. 基于关键词完全匹配 **(Keyword Matching Baseline)**：
   - 原理：用户输入一些描述其需求的关键词（例如，"biotechnology startup funding"）。系统从项目描述或标题中查找完全包含这些关键词（或其任意一个/所有）的项目，并将其判断为相关。
   - 实现：简单的字符串匹配或正则表达式。
   - 通常精确率可能较高（如果关键词选得好），但召回率可能较低（很多相关的项目可能没有完全匹配这些关键词）。
3. 基于用户画像结构化字段精确匹配 **(Exact Profile Match Baseline)**：
   - 原理：用户画像中包含结构化字段（如行业="IT"，公司规模="小于50人"，地区="柏林"）。系统筛选出项目库中在这些结构化字段上与用户画像完全匹配的项目。
   - 实现：Pandas DataFrame 的条件过滤。
   - 这种方法通常精确率较高，但召回率可能很低，因为它要求所有条件都严格满足。

5.5.3 实现基线方法并评估

为上述基线方法编写简单的 Python 函数，使其能够针对测试集中的用户画像（或"项目-用户画像"对）产生预测标签。然后，使用前面定义的 evaluate_matching_system 函数和测试集的真实标签来评估这些基线模型的性能。

```
# baselines.py

import numpy as np
import pandas as pd
import random
from evaluation import evaluate_matching_system
import pickle
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

def random_baseline_predict(num_samples, positive_proba=0.1):
    """
    随机基线预测器。
    :param num_samples: 预测的样本数量
    :param positive_proba: 预测为正例 (1) 的概率
    :return: 随机预测标签数组
    """
    return np.random.choice([0, 1], size=num_samples, p=[1 - positive_proba,
positive_proba])

def keyword_matching_baseline_predict(project_descriptions, user_keywords_list):
    """
    基于关键词匹配的基线预测器 (简化示例)。
    如果项目描述包含用户关键词列表中的任意一个关键词，则预测为相关。
    :param project_descriptions: 项目描述列表 (或 Series)
    :param user_keywords_list: 用户的关键词列表 (例如 ['创新', '软件'] )
    :return: 预测标签数组
    """
    predictions = []
    if not isinstance(project_descriptions, pd.Series):
        project_descriptions = pd.Series(project_descriptions)

    for desc in project_descriptions:
        is_match = False
        if pd.isna(desc):
            predictions.append(0)
            continue
```

```
        desc_lower = str(desc).lower()
        for keyword in user_keywords_list:
            if keyword.lower() in desc_lower:
                is_match = True
                break
        predictions.append(1 if is_match else 0)
    return np.array(predictions)

def exact_profile_match_baseline_predict(projects_df, user_profile):
    """
    基于用户画像结构化字段精确匹配的基线预测器 (简化示例)。
    如果项目在行业和地区上与用户画像完全匹配，则预测为相关。
    :param projects_df: 项目 DataFrame (包含 'source_target_industry', 'source_region' 等独热编码列)
    :param user_profile: 用户画像字典 (包含 'industry', 'region' 等)
    :return: 预测标签数组
    """
    predictions = np.zeros(len(projects_df), dtype=int)

    # 转换为小写以便匹配
    user_industry = user_profile.get('industry', '').lower()
    user_region = user_profile.get('region', '').lower()

    # 假设 projects_df 已经包含了独热编码的列，例如 'source_IT_Software' 或 'source_Germany'
    # 这个基线需要非常具体地匹配处理后的特征

    # 为了演示，我们模拟一个简化的精确匹配逻辑
    # 实际应用中，您需要根据独热编码后的列名来构建条件

    # 假设 'source_IT_Software' 和 'source_Germany' 这样的列存在
    # is_industry_match = projects_df.filter(like=f'source_{user_industry}').sum(axis=1) > 0 if user_industry else True
    # is_region_match = projects_df.filter(like=f'source_{user_region}').sum(axis=1) > 0 if user_region else True

    # 简化为：如果项目描述包含用户行业关键词，且来源是某个特定国家
    # 实际应使用独热编码后的列进行匹配
```

```
    if user_industry and 'processed_description' in projects_df.columns:
        industry_match =
projects_df['processed_description'].astype(str).str.contains(user_industry,
case=False, na=False)
    else:
        industry_match = pd.Series([True] * len(projects_df)) # 不进行行业匹配

    if user_region and 'source' in projects_df.columns:
        region_match = projects_df['source'].astype(str).str.contains(user_region,
case=False, na=False)
    else:
        region_match = pd.Series([True] * len(projects_df)) # 不进行地区匹配

    predictions = (industry_match & region_match).astype(int)
    return predictions.to_numpy() # 确保返回 numpy 数组

# 示例评估流程 (需要真实的测试数据 X_test, y_test, processed_df)
# if __name__ == "__main__":
#     # 1. 加载数据
#     try:
#         with open('X_test.pkl', 'rb') as f:
#             X_test = pickle.load(f)
#         with open('y_test.pkl', 'rb') as f:
#             y_test = pickle.load(f)
#         with open('processed_df.pkl', 'rb') as f: # 包含原始项目信息和处理后的结构化特
征
#             processed_df = pickle.load(f)
#
#         # 确保 processed_df 的索引与 X_test 和 y_test 对应
#         # 实际使用中，processed_df 应该在划分数据集时就与特征矩阵和标签对齐
#         # 这里我们假设 X_test 和 y_test 对应 processed_df 的子集，并且行数一致
#         if X_test.shape[0] != len(processed_df):
#             logging.warning("X_test 和 processed_df 行数不一致，精确匹配基线可能无法
正确演示。")
#             # 为了演示，创建一个与 X_test 行数一致的模拟 processed_df
#             processed_df_test = pd.DataFrame(index=range(X_test.shape[0]))
#             processed_df_test['processed_description'] = ["这是一个关于创新和软件的项
目。"] * X_test.shape[0]
#             processed_df_test['source'] = ["ptj.de"] * (X_test.shape[0] // 2) +
```

```
["eu-startups.com"] * (X_test.shape[0] - X_test.shape[0] // 2)
#         else:
#             processed_df_test = processed_df # 假设已经对齐

#     except FileNotFoundError:
#         logging.error("请确保 X_test.pkl, y_test.pkl, processed_df.pkl 文件已生成。")
#         X_test, y_test, processed_df_test = None, None, None
#     except Exception as e:
#         logging.error(f"加载评估数据时发生错误: {e}")
#         X_test, y_test, processed_df_test = None, None, None

#     if X_test is not None and y_test is not None:
#         # --- 评估随机基线 ---
#         y_pred_random = random_baseline_predict(len(y_test), positive_proba=0.1) #
假设 10% 真实相关
#         print("\n--- 评估随机基线 ---")
#         random_results = evaluate_matching_system(y_test, y_pred_random,
model_name="Random Baseline")

#         # --- 评估关键词匹配基线 ---
#         # 为了演示，假设用户关键词为 'innovation' 和 'software'
#         user_keywords_example = ['innovation', 'software']
#         y_pred_keyword =
keyword_matching_baseline_predict(processed_df_test['processed_description'],
user_keywords_example)
#         print("\n--- 评估关键词匹配基线 ---")
#         keyword_results = evaluate_matching_system(y_test, y_pred_keyword,
model_name="Keyword Match Baseline")

#         # --- 评估精确画像匹配基线 ---
#         # 为了演示，假设用户画像为行业 'software'，地区 'germany'
#         user_profile_example = {'industry': 'software', 'region': 'germany'}
#         y_pred_profile_match =
exact_profile_match_baseline_predict(processed_df_test, user_profile_example)
#         print("\n--- 评估精确画像匹配基线 ---")
#         profile_match_results = evaluate_matching_system(y_test,
y_pred_profile_match, model_name="Profile Match Baseline")

#         # 记录基线性能
```

```
#         baseline_performances = {
#             "Random": random_results,
#             "KeywordMatch": keyword_results,
#             "ProfileMatch": profile_match_results
#         }
#         print("\n--- 所有基线模型性能摘要 ---")
#         for name, res in baseline_performances.items():
#             print(f"{name} F1: {res['f1']:.4f}, Precision: {res['precision']:.4f}, Recall:
{res['recall']:.4f}")
#     else:
#         print("无法进行基线模型评估，数据未加载。")
```

基线模型性能比较示意图的图片

图**5-2:** 基线模型性能比较（假设数据）

5.5.4 分析基线结果

记录下各个基线模型的精确率、召回率和 F1 分数。这些数值将成为衡量后续 AI 模型改进程度的标杆。例如，如果关键词匹配基线的精确率是 0.7 而召回率是 0.2，那么 AI 模型的一个目标可能是在保持精确率不低于 0.65 的情况下，将召回率提升到 0.5 以上。

通过建立科学的评估流程和性能基线，我们为后续的 AI 系统开发和优化奠定了坚实的基础，确保每一步改进都是有据可依、目标明确的。

关键要点总结

- 评估重要性：量化性能、发现问题、指导优化、避免盲目。
- 核心指标：精确率（Precision）、召回率（Recall）、F1 分数 (F1-Score) 是本项目初期的关键评估指标。
- 评估数据集：需要人工标注或通过其他方法构建包含真实标签的数据集，并划分为训练集、验证集和测试集。
- 评估流水线：编写可复用的评估函数，使用 sklearn.metrics 计算各项指标。
- 基线模型：实现并评估简单启发式方法（如随机匹配、关键词匹配、精确画像匹配），作为衡量 AI 模型性能的参照点。

第六章：AI赋能：实现智能资助项目搜索与匹配

在完成了数据获取、清洗、特征工程以及评估体系构建之后，我们终于来到了项目的核心——利用 AI 技术实现智能的资助项目搜索与匹配。本章将结合 Legal-Pythia LLP 项目描述中的核心功能需求，探讨两种实现方案：一种是基于规则与特征相似度的匹配系统，作为坚实的起点；另一种是（可选进阶）引入简单的机器学习模型进行匹配。目标是根据用户提供的画像，从海量资助项目中筛选并推荐最合适的选项。

6.1 系统核心功能设计：满足用户需求

根据 Legal-Pythia LLP 项目描述，系统应具备以下核心功能：

1. 定制化搜索配置文件创建 **(Customised search profile creation)**：允许用户输入其特征，如行业 (Industry)、公司年龄 (Company age)、营业额 (Turnover)、员工数量 (Number of employees) 等。这些信息构成了用户画像，是后续匹配的基础。
2. **AI** 支持的资助项目自动识别与匹配：这是系统的核心智能所在。根据用户画像和资助项目的详细信息（包括文本描述和结构化条件），智能判断项目是否适合该用户。
3. 结构化数据准备与输出 **(Data preparation and output)**：将匹配到的资助项目以清晰、结构化的方式呈现给用户，方便用户进一步筛选和决策。

此外，项目描述中还提到了“信源管理 (Source management)”和“自动化搜索 (Automated search)”，这些主要与爬虫模块的持续运行和数据更新相关，已在前面章节涉及。

6.2 方案一：基于规则与特征相似度的匹配系统（作为坚实起点）

这个方案不直接使用复杂的机器学习模型，而是通过一系列明确的规则和计算特征间的相似度来进行匹配。它相对容易实现和解释，可以作为项目的第一个可用版本，并为后续的机器学习方案提供有价值的洞见和基线参考。

6.2.1 用户画像输入与解析

设计一个简单的用户输入机制。初期可以是命令行参数，或者一个包含预定义字段的字典/JSON 对象。

```
# matching_engine.py (部分代码)

# 示例用户画像 (实际会从用户输入或数据库加载)
user_profile_example = {
    "industry": "Software Development",  # 行业
    "company_age_years": 2,              # 公司年龄 (年)
    "turnover_eur_min": 50000,           # 最低年营业额 (欧元)
    "turnover_eur_max": 500000,          # 最高年营业额 (欧元)
```

```
    "num_employees_min": 5,            # 最少员工数
    "num_employees_max": 49,           # 最多员工数
    "region": "Germany",               # 地区 (例如 'Berlin', 'Bavaria' 或 'Germany')
    "keywords": ["innovation", "digitalization", "sme support"], # 用户关注的关键词
    "funding_needed_eur": 100000       # 期望资助金额
}

def parse_user_profile(profile_data):
    """
    解析用户画像输入，进行标准化和清理。
    :param profile_data: 原始用户画像字典
    :return: 清理和标准化的用户画像字典
    """
    parsed_profile = {
        "industry": profile_data.get("industry", "").lower() if profile_data.get("industry")
else None,
        "company_age_years": int(profile_data["company_age_years"]) if
profile_data.get("company_age_years") is not None else None,
        "turnover_eur_min": float(profile_data["turnover_eur_min"]) if
profile_data.get("turnover_eur_min") is not None else None,
        "turnover_eur_max": float(profile_data["turnover_eur_max"]) if
profile_data.get("turnover_eur_max") is not None else None,
        "num_employees_min": int(profile_data["num_employees_min"]) if
profile_data.get("num_employees_min") is not None else None,
        "num_employees_max": int(profile_data["num_employees_max"]) if
profile_data.get("num_employees_max") is not None else None,
        "region": profile_data.get("region", "").lower() if profile_data.get("region") else
None,
        "keywords": [kw.lower() for kw in profile_data.get("keywords", [])],
        "funding_needed_eur": float(profile_data["funding_needed_eur"]) if
profile_data.get("funding_needed_eur") is not None else None
    }
    return parsed_profile

# parsed_user_profile = parse_user_profile(user_profile_example)
# print(parsed_user_profile)
```

将用户输入的画像信息转换为与项目中资助项目特征一致的格式，以便进行比较。例如，

行业名称进行标准化处理，金额、年龄等转换为数值。

6.2.2 多维度筛选 (Hard Filtering)

首先，根据用户画像中的结构化信息（如行业、公司规模要求、地区限制、资助金额范围等），对项目库（combined_df 或 processed_df）进行一轮硬性筛选，排除掉明显不符合条件的项目。这可以大大缩小后续进行文本相似度计算的范围，提高效率。

```
# matching_engine.py (部分代码)

def filter_projects_by_criteria(projects_df, user_profile):
    """
    根据用户画像中的结构化条件筛选项目 DataFrame。
    :param projects_df: 包含项目信息的 DataFrame (processed_df)
    :param user_profile: 标准化的用户画像字典
    :return: 筛选后的 DataFrame
    """
    if projects_df.empty:
        print("项目 DataFrame 为空，无法进行筛选。")
        return pd.DataFrame()

    filtered_df = projects_df.copy()

    # 行业筛选 (假设项目 DataFrame 中有 'target_industry' 或 'processed_description' 列)
    if user_profile.get("industry"):
        # 简单包含匹配，实际可能需要更复杂的行业分类匹配逻辑 (例如，独热编码匹配)
        filtered_df = filtered_df[
filtered_df['processed_description'].astype(str).str.contains(user_profile["industry"], case=False, na=False)
        ]
        print(f"行业筛选 ({user_profile['industry']}) 后剩余 {len(filtered_df)} 个项目。")

    # 公司年龄筛选 (假设项目有 'min_company_age_req', 'max_company_age_req' 列或可从描述中提取)
    # 为简化演示，这里假设 projects_df 已经有这些数值列
    if user_profile.get("company_age_years") is not None:
        user_age = user_profile["company_age_years"]
        if 'min_company_age_req' in filtered_df.columns:
```

```
            filtered_df = filtered_df[filtered_df['min_company_age_req'].fillna(0) <=
user_age]
        if 'max_company_age_req' in filtered_df.columns:
            filtered_df = filtered_df[filtered_df['max_company_age_req'].fillna(float('inf'))
>= user_age]
        print(f"公司年龄筛选 ({user_age}年) 后剩余 {len(filtered_df)} 个项目。")

    # 员工数量筛选 (假设项目有 'required_employees_min', 'required_employees_max'
列)
    if user_profile.get("num_employees_min") is not None:
        user_min_employees = user_profile["num_employees_min"]
        if 'required_employees_max' in filtered_df.columns:
            filtered_df = filtered_df[filtered_df['required_employees_max'].fillna(float('inf'))
>= user_min_employees]
        print(f"员工最低数量筛选 ({user_min_employees}) 后剩余 {len(filtered_df)} 个项
目。")
    if user_profile.get("num_employees_max") is not None:
        user_max_employees = user_profile["num_employees_max"]
        if 'required_employees_min' in filtered_df.columns:
            filtered_df = filtered_df[filtered_df['required_employees_min'].fillna(0) <=
user_max_employees]
        print(f"员工最高数量筛选 ({user_max_employees}) 后剩余 {len(filtered_df)} 个项
目。")

    # 地区筛选 (假设项目有 'source' 或 'region' 列)
    if user_profile.get("region"):
        # 简单包含匹配，或者可以根据独热编码后的 region 列进行匹配
        filtered_df = filtered_df[
            filtered_df['source'].astype(str).str.contains(user_profile["region"], case=False,
na=False) |

filtered_df['processed_description'].astype(str).str.contains(user_profile["region"],
case=False, na=False)
        ]
        print(f"地区筛选 ({user_profile['region']}) 后剩余 {len(filtered_df)} 个项目。")

    # 期望资助金额筛选 (假设项目有 'funding_amount_min_numeric',
'funding_amount_max_numeric' 列)
    if user_profile.get("funding_needed_eur") is not None:
```

```
        funding_needed = user_profile["funding_needed_eur"]
        if 'funding_amount_numeric' in filtered_df.columns: # 假设已经清洗并转换成了数值
            # 项目的资助金额范围覆盖用户需求
            filtered_df = filtered_df[
                (filtered_df['funding_amount_numeric'].fillna(0) <= funding_needed) &
                (filtered_df['funding_amount_numeric'].fillna(float('inf')) >= funding_needed)
            ]
        print(f"期望金额筛选 ({funding_needed} EUR) 后剩余 {len(filtered_df)} 个项目。")

    print(f"硬性筛选后最终剩余 {len(filtered_df)} 个项目。")
    return filtered_df

# 示例使用
# if __name__ == "__main__":
#     # 模拟一个 processed_df (实际从数据处理管道获取)
#     mock_projects_data = {
#         'id': ['id1', 'id2', 'id3', 'id4', 'id5'],
#         'title': ['Software Innovation Grant', 'Biotech Startup Fund', 'SME Growth Program', 'Digital Transformation Aid', 'Agricultural Development Grant'],
#         'description': ['Grant for software development in Germany.', 'Fund for biotech startups in Berlin.', 'Support for small and medium enterprises across EU.', 'Aid for digitalization in manufacturing in Germany.', 'Rural development fund.'],
#         'processed_description': ['software innovation grant', 'biotech startup fund', 'sme growth program', 'digital transformation aid', 'agricultural development fund'], # 预处理后的描述
#         'source': ['ptj.de', 'eu-startups.com', 'foerderdatenbank.de', 'ptj.de', 'foerderdatenbank.de'],
#         'funding_amount_numeric': [150000, 750000, 300000, 200000, 100000], # 假设的金额
#         'min_company_age_req': [1, 0, 3, 2, 5], # 假设的年龄要求
#         'max_company_age_req': [5, 10, 15, 8, 20],
#         'required_employees_min': [1, 10, 5, 3, 2], # 假设的员工要求
#         'required_employees_max': [50, 100, 200, 100, 50]
#     }
#     mock_processed_df = pd.DataFrame(mock_projects_data)

#     user_profile_soft_eng = {
#         "industry": "software",
```

```
#         "company_age_years": 3,
#         "region": "germany",
#         "funding_needed_eur": 180000,
#         "num_employees_min": 10,
#         "num_employees_max": 60,
#         "keywords": ["innovation", "digital"]
#     }
#     filtered_projects = filter_projects_by_criteria(mock_processed_df,
user_profile_soft_eng)
#     print("\n筛选后的项目:")
#     print(filtered_projects[['title', 'source', 'funding_amount_numeric']])
```

注意：上述代码中的列名（如'min_company_age_req', 'required_employees_max'）是假设的，需要替换为实际清洗和特征工程后 DataFrame 中的列名。如果这些信息需要从文本中提取，请在数据处理阶段完成。

6.2.3 文本相似度匹配 (Soft Matching)

对于经过初步筛选的项目，我们需要进一步评估其文本内容（如项目描述、目标、资格要求等）与用户需求描述（可以从用户画像中的关键词 user_profile["keywords"] 生成，或者用户直接输入一段需求描述）之间的相关性。

- 特征表示：使用上一章生成的 TF-IDF 向量（tfidf_matrix）和对应的 TfidfVectorizer 对象（vectorizer）。
- 用户需求向量化：将用户的关键词列表或需求描述，使用已训练好的 vectorizer 转换为 TF-IDF 向量。

```
# matching_engine.py (部分代码)

from sklearn.metrics.pairwise import cosine_similarity
# from data_processing import preprocess_text # 导入预处理函数

def calculate_text_similarity(filtered_projects_df, user_profile, tfidf_vectorizer,
all_projects_tfidf_matrix):
    """
    计算筛选后项目与用户查询的文本相似度。
    :param filtered_projects_df: 经过硬筛选后的项目 DataFrame
    :param user_profile: 标准化的用户画像字典
    :param tfidf_vectorizer: 训练好的 TfidfVectorizer 对象
    :param all_projects_tfidf_matrix: 原始所有项目的 TF-IDF 稀疏矩阵
```

```
    :return: 带有 'text_similarity_score' 列的 DataFrame
    """
    if filtered_projects_df.empty or tfidf_vectorizer is None or all_projects_tfidf_matrix is
None:
        print("无筛选后项目或 TF-IDF 工具未加载，无法计算文本相似度。")
        filtered_projects_df['text_similarity_score'] = 0.0
        return filtered_projects_df

    user_keywords_text = " ".join(user_profile.get("keywords", []))
    if not user_keywords_text:
        print("用户未提供关键词，文本相似度设为 0。")
        filtered_projects_df['text_similarity_score'] = 0.0
        return filtered_projects_df

    # 使用与项目描述相同的预处理函数
    processed_user_query = preprocess_text(user_keywords_text, language='german')
# 假设德语
    if not processed_user_query:
        print("用户查询预处理后为空，文本相似度设为 0。")
        filtered_projects_df['text_similarity_score'] = 0.0
        return filtered_projects_df

    user_query_vector = tfidf_vectorizer.transform([processed_user_query])

    # 获取筛选后项目在原始 TF-IDF 矩阵中的对应行 (通过 ID 或原始索引)
    # 假设 filtered_projects_df 仍然保留了原始 DataFrame 的索引
    # 如果索引不一致，您需要通过 id 列来查找对应的行
    project_indices_in_original_matrix = filtered_projects_df.index

    if len(project_indices_in_original_matrix) == 0:
        print("筛选后项目为空，无法计算相似度。")
        filtered_projects_df['text_similarity_score'] = 0.0
        return filtered_projects_df

    project_tfidf_vectors_for_filtered =
all_projects_tfidf_matrix[project_indices_in_original_matrix]

    # 计算用户查询向量与所有筛选后项目描述向量的余弦相似度
    similarities = cosine_similarity(user_query_vector, project_tfidf_vectors_for_filtered)
```

```
    similarity_scores = similarities[0] # similarities 是一个 (1, num_filtered_projects) 的数组

    filtered_projects_df['text_similarity_score'] = similarity_scores
    print(f"已计算 {len(similarity_scores)} 个项目的文本相似度。")
    return filtered_projects_df

# 示例使用 (需要加载 tfidf_vectorizer 和 all_projects_tfidf_matrix)
# if __name__ == "__main__":
#     import pickle
#     # 假设已经运行了数据处理管道并保存了这些文件
#     try:
#         with open('tfidf_vectorizer.pkl', 'rb') as f:
#             tfidf_vectorizer_loaded = pickle.load(f)
#         with open('final_features.pkl', 'rb') as f: # final_features.pkl 包含了所有项目的 TF-IDF 特征
#             all_projects_tfidf_matrix_loaded = pickle.load(f)
#
#         # 模拟一个 processed_df，其索引与 all_projects_tfidf_matrix_loaded 的行号对应
#         mock_projects_data = {
#             'id': ['id1', 'id2', 'id3', 'id4', 'id5'],
#             'title': ['Software Innovation Grant', 'Biotech Startup Fund', 'SME Growth Program', 'Digital Transformation Aid', 'Agricultural Development Grant'],
#             'description': ['Grant for software development in Germany. This is a great innovation fund.', 'Fund for biotech startups in Berlin.', 'Support for small and medium enterprises across EU. Focus on growth.', 'Aid for digitalization in manufacturing in Germany. Focus on digital solutions.', 'Rural development fund.'],
#             'processed_description': ['software innovation grant germany innovation fund', 'biotech startup fund berlin', 'sme growth program across eu focus growth', 'digital transformation aid manufacturing germany focus digital solutions', 'rural development fund'], # 预处理后的描述
#             'source': ['ptj.de', 'eu-startups.com', 'foerderdatenbank.de', 'ptj.de', 'foerderdatenbank.de'],
#             'funding_amount_numeric': [150000, 750000, 300000, 200000, 100000],
#             # ... 其他结构化特征 ...
#         }
#         mock_processed_df_for_similarity = pd.DataFrame(mock_projects_data)
#         mock_processed_df_for_similarity.index = range(len(mock_processed_df_for_similarity)) # 确保索引是 0 到 N-1
```

```
#
#       user_profile_sim_test = {
#           "industry": "software",
#           "region": "germany",
#           "keywords": ["innovation", "digitalization"] # 用户提供的关键词
#       }
#
#       # 模拟硬筛选 (为了演示，这里直接使用原始 mock_processed_df_for_similarity)
#       filtered_projects_sim = mock_processed_df_for_similarity.copy()
#
#       projects_with_similarity = calculate_text_similarity(
#           filtered_projects_sim,
#           user_profile_sim_test,
#           tfidf_vectorizer_loaded,
#           all_projects_tfidf_matrix_loaded # 这里需要确保这个矩阵的行数和顺序与
mock_processed_df_for_similarity 对应
#       )
#       print("\n计算相似度后的项目 (Top 5):")
#       print(projects_with_similarity[['title', 'processed_description',
'text_similarity_score']].sort_values(by='text_similarity_score',
ascending=False).head())
#
#   except FileNotFoundError:
#       print("请确保 'tfidf_vectorizer.pkl' 和 'final_features.pkl' 文件已生成。")
#   except Exception as e:
#       print(f"计算文本相似度时发生错误: {e}")
```

- 综合打分与排序：
  可以将结构化字段的匹配程度（例如，满足的硬性条件个数或重要性加权）和文本相似度得分结合起来，形成一个综合匹配分。例如，可以设计一个简单的加权公式：
  综合分=w1×(结构化匹配得分)+w2×(文本相似度得分)

  其中 w1 和 w2 是权重，可以根据经验调整。结构化匹配得分可以根据满足用户画像中条件的严格程度来设定（例如，行业完全匹配得 1 分，部分匹配得 0.5 分）。
  然后根据综合得分对项目进行降序排序，推荐得分最高的项目。
  # matching_engine.py (部分代码)

  def rank_projects(projects_df, w_structural=0.4, w_text=0.6):

```
    """
    根据综合得分对项目进行排序。
    :param projects_df: 包含 'text_similarity_score' 的 DataFrame (以及隐含的结构化匹配结果)
    :param w_structural: 结构化匹配得分的权重
    :param w_text: 文本相似度得分的权重
    :return: 按综合得分降序排序的 DataFrame
    """
    if projects_df.empty:
        print("项目 DataFrame 为空，无法进行排序。")
        return pd.DataFrame()

    # 结构化匹配得分：如果通过了硬筛选，则结构化匹配得分至少为 1 (可以设计更复杂的打分机制)
    # 这里简化为如果存在，就得 1 分，否则为 0
    # 假设 filter_projects_by_criteria 已经处理了结构化匹配
    # 我们可以在这里基于通过的硬筛选数量来给分，但为简化，我们假设过了硬筛选的，结构化匹配得分基础分是 1
    projects_df['structural_match_score'] = 1.0 # 假设只要没被硬筛选掉，结构化匹配就得分

    # 确保 text_similarity_score 存在且是数值
    if 'text_similarity_score' not in projects_df.columns:
        projects_df['text_similarity_score'] = 0.0
        print("警告: 缺少 'text_similarity_score' 列，综合得分将主要依赖结构化匹配。")

    projects_df['combined_score'] = (w_structural * projects_df['structural_match_score']) + \
                                    (w_text * projects_df['text_similarity_score'])

    sorted_projects = projects_df.sort_values(by='combined_score', ascending=False)
    print(f"按综合得分排序后，共 {len(sorted_projects)} 个项目。")
    return sorted_projects

# 完整规则与相似度匹配流程
def rule_based_matching_system(all_projects_df, user_profile, tfidf_vectorizer, all_projects_tfidf_matrix):
    """
```

```
    运行基于规则与相似度的匹配系统。
    :param all_projects_df: 所有项目的 DataFrame (processed_df)
    :param user_profile: 标准化的用户画像字典
    :param tfidf_vectorizer: 训练好的 TfidfVectorizer 对象
    :param all_projects_tfidf_matrix: 所有项目的 TF-IDF 稀疏矩阵
    :return: 匹配并排序后的项目 DataFrame
    """
    print("\n--- 启动基于规则与相似度的匹配系统 ---")
    # 1. 硬筛选
    filtered_df = filter_projects_by_criteria(all_projects_df, user_profile)

    if filtered_df.empty:
        print("硬筛选后没有符合条件的项目。")
        return pd.DataFrame()

    # 2. 文本相似度计算
    projects_with_similarity = calculate_text_similarity(
        filtered_df, user_profile, tfidf_vectorizer, all_projects_tfidf_matrix
    )

    # 3. 综合打分与排序
    matched_and_ranked_projects = rank_projects(projects_with_similarity)
    print("--- 规则与相似度匹配系统完成 ---")
    return matched_and_ranked_projects

# 示例运行完整流程
# if __name__ == "__main__":
#     import pickle
#     # 假设加载处理好的数据和工具
#     try:
#         with open('processed_df.pkl', 'rb') as f: # 包含处理后的原始数据
#             processed_df_loaded = pickle.load(f)
#         with open('tfidf_vectorizer.pkl', 'rb') as f:
#             tfidf_vectorizer_loaded = pickle.load(f)
#         with open('final_features.pkl', 'rb') as f: # 所有项目的 TF-IDF 矩阵
#             all_projects_tfidf_matrix_loaded = pickle.load(f)

#         # 模拟用户画像
#         test_user_profile = {
```

```
#             "industry": "software",
#             "company_age_years": 3,
#             "region": "germany",
#             "funding_needed_eur": 180000,
#             "num_employees_min": 10,
#             "num_employees_max": 60,
#             "keywords": ["innovation", "digital"]
#         }
#         parsed_test_user_profile = parse_user_profile(test_user_profile)

#         # 运行匹配系统
#         top_matches = rule_based_matching_system(
#             processed_df_loaded,
#             parsed_test_user_profile,
#             tfidf_vectorizer_loaded,
#             all_projects_tfidf_matrix_loaded
#         )
#         print("\nTop 5 匹配项目:")
#         if not top_matches.empty:
#             print(top_matches[['title', 'source', 'combined_score',
'text_similarity_score']].head())
#         else:
#             print("没有找到匹配项目。")

#     except FileNotFoundError:
#         print("请确保数据和工具文件 (.pkl) 已生成。")
#     except Exception as e:
#         print(f"运行规则与相似度匹配系统时发生错误: {e}")
```

- 评估：
  使用第五部分搭建的评估流水线和准备好的测试集，评估这个基于规则和相似度的匹配系统的性能（精确率、召回率、F1 分数）。这需要将排序后的项目列表（例如，取 Top-K 个）与真实标签进行比较。如果系统输出的是每个项目的匹配概率或得分，需要设定一个阈值来将其转换为二分类标签（匹配/不匹配）才能使用分类评估指标。

6.3 方案二（可选进阶）：引入简单机器学习模型进行匹配

如果基于规则和相似度的系统性能不够理想，或者希望系统能从数据中学习更复杂的匹配模式，可以考虑引入机器学习模型。这里我们介绍一个简单的二分类场景。

6.3.1 场景定义与特征准备

- 场景定义：将“资助项目是否匹配给定的用户画像”视为一个二分类问题。模型需要预测的标签是“匹配(1)”或“不匹配(0)”。
- 特征准备：
  - 项目特征：可以使用项目的 TF-IDF 向量，以及标准化的数值型特征（如资助金额、项目时长）和独热编码后的类别型特征（如资助机构类型、地区）。
  - 用户画像特征：同样需要进行向量化（如用户关键词的 TF-IDF 向量）和结构化处理（如行业独热编码、公司规模数值化）。
  - 组合特征 **(Interaction Features)**：这是关键。模型需要学习项目特征和用户画像特征之间的关系。可以：
    - 拼接：直接将项目特征向量和用户画像特征向量拼接成一个更长的特征向量。
    - 差异/相似度特征：计算项目和用户画像在某些维度上的显式相似度或差异。例如，用户期望资助金额与项目提供金额的差异；项目行业与用户行业的匹配度（如 Jaccard 相似系数）。
      最终的输入特征 X 是针对每一对“（项目，用户画像）”组合生成的。
- 标签准备：需要有标注数据（y_train, y_val, y_test），其中标签为 1 表示该项目与该用户画像匹配，0 表示不匹配。这些标签来自第五章准备的评估数据集。

6.3.2 模型选择（初学者友好）

- 逻辑回归 **(Logistic Regression)**：简单、快速、可解释性较好，输出概率。
- 朴素贝叶斯 **(Naive Bayes)**：基于贝叶斯定理，对特征条件独立性有假设，在文本分类中常用。
- 支持向量机 **(SVM)**：寻找最优超平面进行分类，在中小数据集上表现良好。
- 决策树 **(Decision Tree)**/随机森林 **(Random Forest)**：基于树形结构进行决策，随机森林是多个决策树的集成，通常性能更好，能处理非线性关系，对特征缩放不敏感。

选择这些模型是因为它们相对容易理解和实现，并且 scikit-learn 库提供了完善的接口。

6.3.3 模型训练与调优

```
# ml_matching.py (可选进阶部分)

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, f1_score # 用于 GridSearchCV 的评分
import logging
import pickle

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
```

```
%(message)s')

def train_and_evaluate_model(X_train, y_train, X_val, y_val,
model_type='logistic_regression'):
    """
    训练和评估机器学习模型。
    :param X_train: 训练特征矩阵
    :param y_train: 训练标签
    :param X_val: 验证特征矩阵
    :param y_val: 验证标签
    :param model_type: 模型类型 ('logistic_regression' 或 'random_forest')
    :return: 训练好的模型
    """
    if X_train is None or y_train is None or len(y_train) == 0:
        logging.error("训练数据为空，无法训练模型。")
        return None

    model = None
    if model_type == 'logistic_regression':
        logging.info("正在训练逻辑回归模型...")
        model = LogisticRegression(random_state=42, max_iter=1000) # 增加 max_iter
防止收敛警告
        # 可以添加 GridSearchCV 进行超参数调优
        # param_grid = {'C': [0.1, 1, 10], 'solver': ['liblinear', 'lbfgs']}
        # grid_search = GridSearchCV(model, param_grid, cv=3, scoring='f1', verbose=1)
        # grid_search.fit(X_train, y_train)
        # model = grid_search.best_estimator_
        # logging.info(f"逻辑回归最佳参数: {grid_search.best_params_}")
    elif model_type == 'random_forest':
        logging.info("正在训练随机森林模型...")
        model = RandomForestClassifier(random_state=42)
        # 示例：GridSearchCV 超参数调优
        # param_grid_rf = {
        #     'n_estimators': [50, 100],
        #     'max_depth': [None, 10, 20],
        #     'min_samples_split': [2, 5]
        # }
        # grid_search_rf = GridSearchCV(model, param_grid_rf, cv=3,
scoring=make_scorer(f1_score, zero_division=0), verbose=1)
```

```
        # grid_search_rf.fit(X_train, y_train)
        # model = grid_search_rf.best_estimator_
        # logging.info(f"随机森林最佳参数: {grid_search_rf.best_params_}")
    else:
        logging.error(f"不支持的模型类型: {model_type}")
        return None

    # 实际训练模型
    if model:
        model.fit(X_train, y_train)
        logging.info(f"{model_type} 模型训练完成。")

        # 在验证集上初步评估 (如果提供了验证集)
        if X_val is not None and y_val is not None and len(y_val) > 0:
            y_pred_val = model.predict(X_val)
            val_f1 = f1_score(y_val, y_pred_val, zero_division=0)
            logging.info(f"{model_type} 验证集 F1 分数: {val_f1:.4f}")
        return model
    return None

# 示例使用 (需要加载 X_train, y_train, X_val, y_val)
# if __name__ == "__main__":
#     try:
#         with open('X_train.pkl', 'rb') as f: X_train_loaded = pickle.load(f)
#         with open('y_train.pkl', 'rb') as f: y_train_loaded = pickle.load(f)
#         with open('X_val.pkl', 'rb') as f: X_val_loaded = pickle.load(f)
#         with open('y_val.pkl', 'rb') as f: y_val_loaded = pickle.load(f)

#         # 训练逻辑回归
#         lr_model = train_and_evaluate_model(X_train_loaded, y_train_loaded,
X_val_loaded, y_val_loaded, model_type='logistic_regression')
#         if lr_model:
#             with open('lr_model.pkl', 'wb') as f:
#                 pickle.dump(lr_model, f)
#             print("逻辑回归模型已保存。")

#         # 训练随机森林
#         rf_model = train_and_evaluate_model(X_train_loaded, y_train_loaded,
X_val_loaded, y_val_loaded, model_type='random_forest')
```

```
#         if rf_model:
#             with open('rf_model.pkl', 'wb') as f:
#                 pickle.dump(rf_model, f)
#             print("随机森林模型已保存。")

#     except FileNotFoundError:
#         print("请确保训练集和验证集文件已生成 (X_train.pkl, y_train.pkl, etc.)。")
#     except Exception as e:
#         print(f"训练模型时发生错误: {e}")
```

6.3.4 模型评估

使用在测试集（X_test, y_test）上的预测结果，调用第五章定义的 evaluate_matching_system 函数来评估训练好的模型性能。

```
# ml_matching.py (评估部分)

from evaluation import evaluate_matching_system # 从 evaluation.py 导入
# from ml_matching import train_and_evaluate_model # 从本文件导入
# from baselines import random_baseline_predict, keyword_matching_baseline_predict, exact_profile_match_baseline_predict # 导入基线

# 示例：评估机器学习模型 (在主流程中调用)
def evaluate_ml_model(model, X_test, y_test, model_name="ML Model"):
    """
    使用测试集评估训练好的机器学习模型。
    :param model: 训练好的模型
    :param X_test: 测试集特征
    :param y_test: 测试集真实标签
    :param model_name: 模型名称
    :return: 评估结果字典
    """
    if model is None or X_test is None or y_test is None or len(y_test) == 0:
        logging.error("模型或测试数据为空，无法评估。")
        return None

    logging.info(f"--- 正在评估 {model_name} 模型 (测试集) ---")
    y_pred_test = model.predict(X_test)
```

```
    results = evaluate_matching_system(y_test, y_pred_test, model_name=model_name)
    return results

# 假设在主流程中，您会先运行数据处理，然后划分数据集，然后训练模型，最后评估
# if __name__ == "__main__":
#     # 模拟加载测试集和训练好的模型
#     try:
#         with open('X_test.pkl', 'rb') as f: X_test_loaded = pickle.load(f)
#         with open('y_test.pkl', 'rb') as f: y_test_loaded = pickle.load(f)
#         with open('lr_model.pkl', 'rb') as f: lr_model_loaded = pickle.load(f)
#         with open('rf_model.pkl', 'rb') as f: rf_model_loaded = pickle.load(f)

#         # 评估逻辑回归模型
#         lr_results = evaluate_ml_model(lr_model_loaded, X_test_loaded, y_test_loaded,
model_name="Logistic Regression")

#         # 评估随机森林模型
#         rf_results = evaluate_ml_model(rf_model_loaded, X_test_loaded, y_test_loaded,
model_name="Random Forest")

#         # 假设您也有基线模型的评估结果
#         # baseline_performances_summary = {
#         #     "Random Baseline F1": 0.1000, # 假设值
#         #     "Keyword Match Baseline F1": 0.4500, # 假设值
#         #     "Profile Match Baseline F1": 0.3000, # 假设值
#         # }

#         # print("\n--- 性能对比 ---")
#         # print(f"Logistic Regression F1: {lr_results['f1']:.4f}")
#         # print(f"Random Forest F1: {rf_results['f1']:.4f}")
#         # for k, v in baseline_performances_summary.items():
#         #     print(f"{k}: {v:.4f}")

#     except FileNotFoundError:
#         print("请确保测试集和模型文件 (.pkl) 已生成。")
#     except Exception as e:
#         print(f"评估机器学习模型时发生错误: {e}")
```

6.4 结果呈现与结构化输出

无论采用哪种方案，最终都需要将匹配到的资助项目以清晰、结构化的方式呈现给用户。输出格式可以是：

- **JSON** 数组：每个元素是一个代表资助项目的 JSON 对象，包含关键信息。

```
[
 {
  "id": "md5_hash_of_item1",
  "title": "创新型中小企业扶持计划",
  "source": "ptj.de",
  "link": "https://www.ptj.de/projekt/123",
  "description_summary": "为德国地区创新型中小企业提供资金支持。",
  "match_score": 0.85, // (可选) 匹配得分或概率
  "matched_keywords": ["创新", "中小企业"], // (可选) 与用户画像匹配上的点
  "application_deadline": "2025-12-31",
  "funding_amount_range": "50.000 - 200.000 EUR",
  "region": "Germany",
  "industry": "Software"
 },
 {
  // ...更多匹配的项目...
 }
]
```

- 用户界面初步设想：
  - 初期（命令行界面 **- CLI**）：用户通过命令行输入参数（如行业、关键词等），系统在命令行打印出匹配的项目列表。
  - 进阶（简单 **Web** 界面）：可以使用 Python 的轻量级 Web 框架如 Flask 或 Streamlit 快速搭建一个简单的 Web 应用。用户通过网页表单输入搜索条件，匹配结果以表格或卡片形式在网页上展示。Streamlit 尤其适合快速将数据脚本转换为交互式 Web 应用，对初学者友好。

通过本章的实践，我们将能够构建一个具备初步智能的资助项目发现系统。无论是基于规则和相似度，还是引入简单的机器学习模型，核心都是围绕用户需求，从数据中提取价值，并以有效的方式呈现结果。

关键要点总结

- 核心功能：定制化用户画像输入、AI 智能匹配、结构化结果输出。
- 方案一（规则与相似度）：通过硬性条件筛选，再结合 TF-IDF 和余弦相似度进行文本软

匹配，最后综合打分排序。易于实现和解释。

- 方案二（简单机器学习）：将匹配问题转化为二分类任务，选择如逻辑回归、随机森林等模型，利用标注数据进行训练和评估。可能获得更优性能但需要标注数据和模型调优。
- 特征工程关键：对于机器学习方案，如何组合项目特征和用户画像特征以供模型学习是核心。
- 结果输出：设计清晰的 JSON 输出格式，并考虑未来通过 CLI 或简单 Web 界面与用户交互。

第七章：后端API开发（Flask）

后端 API 是连接前端界面和数据存储层的桥梁，它负责处理业务逻辑、与数据库交互，并向前端提供结构化的数据。本章将指导您如何使用 Python 的轻量级 Web 框架 Flask 来构建 RESTful API。

7.1 Flask 框架简介与 RESTful API 设计原则

7.1.1 Flask 框架简介

Flask 是一个用 Python 编写的微型 Web 框架。它被称为“微型”框架，因为它不包含 ORM（对象关系映射器）或特定的数据库抽象层等工具。相反，它提供了构建 Web 应用程序所需的核心功能，并允许开发者自由选择其他组件。这种灵活性使得 Flask 非常适合小型项目和 API 开发，也易于初学者理解和上手。

Flask 的特点：

- 轻量级：核心功能精简，易于学习和使用。
- 灵活性：不强制使用特定的工具或库，开发者可以根据项目需求自由选择。
- 易于扩展：拥有丰富的扩展库，可以方便地添加各种功能。
- 良好的文档：官方文档详细且易懂。

7.1.2 RESTful API 设计原则

REST (Representational State Transfer) 是一种软件架构风格，用于设计网络应用程序。RESTful API 遵循以下核心原则：

1. 资源 **(Resource)**：API 中的所有事物都被视为资源，例如“资金项目”或“搜索档案”。每个资源都有一个唯一的标识符（URI/URL）。
2. 统一接口 **(Uniform Interface)**：使用标准的 HTTP 方法（GET, POST, PUT, DELETE）来对资源进行操作。
    - GET：从服务器获取资源。
    - POST：在服务器上创建新资源。
    - PUT：更新服务器上的现有资源（通常是完整替换）。
    - DELETE：从服务器上删除资源。
3. 无状态 **(Stateless)**：服务器不保存客户端的任何会话信息。每次请求都必须包含完成该请求所需的所有信息。
4. 可缓存 **(Cacheable)**：客户端可以缓存服务器的响应，以提高性能。
5. 分层系统 **(Layered System)**：客户端无法直接与最终服务器交互，而是通过中间服务器（如代理、负载均衡器）进行通信。

本项目将主要关注资源和统一接口原则，使用 JSON 作为数据交换格式。

7.2 构建 Flask 应用

我们将创建一个名为 app.py 的文件，作为 Flask 应用的主入口。在这个文件中，我们将定义 API 路由，并与之前创建的数据库操作函数进行交互。

7.2.1 项目文件结构

首先，确保您的项目结构如下：

```
funding_search_tool/
├── venv/                  # 虚拟环境
├── scrapers/              # 爬虫模块目录
│   ├── scraper_ptj.py
│   ├── scraper_eustartups.py
│   ├── scraper_foerderdatenbank.py
│   └── scraper_ddb.py
├── main_scraper.py        # 主爬虫调度模块
├── data_processing.py     # 数据处理与特征工程模块 (第四章)
├── evaluation.py          # 模型评估模块 (第五章)
├── matching_engine.py     # 匹配引擎模块 (第六章)
├── database.py            # 数据库操作模块 (我们将在此章补充)
├── funds.db               # SQLite数据库文件 (运行后生成)
├── final_features.pkl     # 最终特征矩阵 (pickle 文件)
├── tfidf_vectorizer.pkl   # TF-IDF 向量化器 (pickle 文件)
├── X_train.pkl            # 训练集特征
├── y_train.pkl            # 训练集标签
├── X_val.pkl              # 验证集特征
├── y_val.pkl              # 验证集标签
├── X_test.pkl             # 测试集特征
├── y_test.pkl             # 测试集标签
├── lr_model.pkl           # 逻辑回归模型 (可选)
├── rf_model.pkl           # 随机森林模型 (可选)
├── app.py                 # Flask 应用主文件
└── static/                # 前端静态文件目录
    ├── index.html
    ├── style.css
    └── script.js
```

7.2.2 数据库操作模块 (database.py)

为了统一管理数据库操作，我们首先创建一个 database.py 文件，包含连接、初始化、数据插入、查询、更新和删除等功能。

```
# database.py

import sqlite3
from datetime import datetime
import pandas as pd
import hashlib # 用于生成 funds 表的 ID

def get_db_connection():
    """
    获取 SQLite 数据库连接。
    :return: 数据库连接对象
    """
    conn = sqlite3.connect("funds.db") # 连接到 funds.db 数据库文件，如果文件不存在则创建
    conn.row_factory = sqlite3.Row # 设置行工厂，使查询结果可以通过列名访问
    return conn

def init_db():
    """
    初始化数据库，创建 funds 和 search_profiles 表（如果不存在）。
    """
    conn = get_db_connection()
    cursor = conn.cursor()

    # 创建 funds 表
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS funds (
            id TEXT PRIMARY KEY,
            title TEXT NOT NULL,
            description TEXT,
            link TEXT,
            source TEXT,
            crawled_at TEXT DEFAULT CURRENT_TIMESTAMP
        )
    """)
```

```
    # 创建 search_profiles 表
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS search_profiles (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL UNIQUE,
            industry TEXT,
            company_age TEXT,
            turnover TEXT,
            employees TEXT,
            created_at TEXT DEFAULT CURRENT_TIMESTAMP
        )
    """)
    conn.commit() # 提交事务，保存更改
    conn.close() # 关闭连接
    print("数据库初始化完成，表已创建或已存在。")

def insert_or_update_fund(fund_data):
    """
    插入或更新资金项目数据。根据 ID 判断是插入新记录还是更新现有记录。
    :param fund_data: 包含资金项目数据的字典，必须包含 'id', 'title', 'description', 'link',
'source'。
    """
    conn = get_db_connection()
    cursor = conn.cursor()

    # 检查是否存在相同 ID 的记录
    cursor.execute("SELECT id FROM funds WHERE id = ?", (fund_data["id"],))
    existing_fund = cursor.fetchone()

    if existing_fund:
        # 如果存在，则更新记录
        print(f"更新资金项目: {fund_data['title']}")
        cursor.execute("""
            UPDATE funds
            SET title = ?, description = ?, link = ?, source = ?, crawled_at = ?
            WHERE id = ?
        """, (
            fund_data["title"],
            fund_data["description"],
```

```
            fund_data["link"],
            fund_data["source"],
            datetime.now().isoformat(), # 更新爬取时间
            fund_data["id"]
        ))
    else:
        # 如果不存在，则插入新记录
        print(f"插入新资金项目: {fund_data['title']}")
        cursor.execute("""
            INSERT INTO funds (id, title, description, link, source, crawled_at)
            VALUES (?, ?, ?, ?, ?, ?)
        """, (
            fund_data["id"],
            fund_data["title"],
            fund_data["description"],
            fund_data["link"],
            fund_data["source"],
            datetime.now().isoformat() # 记录当前时间
        ))
    conn.commit()
    conn.close()

def save_funds_data(df):
    """
    将 Pandas DataFrame 中的资金项目数据保存/更新到数据库。
    :param df: 包含资金项目数据的 DataFrame。
    """
    if df.empty:
        print("没有数据可保存。")
        return
    for index, row in df.iterrows():
        fund_data = row.to_dict()
        insert_or_update_fund(fund_data)
    print("所有资金项目数据已保存/更新到数据库。")

def get_funds(source=None, keyword=None, limit=None, offset=None):
    """
    从 funds 表中查询资金项目数据。
    :param source: 来源网站，可选。
```

```
    :param keyword: 关键词，在标题或描述中模糊匹配，可选。
    :param limit: 返回的最大记录数，可选。
    :param offset: 偏移量，可选。
    :return: 资金项目列表 (字典列表)。
    """
    conn = get_db_connection()
    cursor = conn.cursor()

    query = "SELECT * FROM funds WHERE 1=1"
    params = []

    if source:
        query += " AND source = ?"
        params.append(source)
    if keyword:
        query += " AND (title LIKE ? OR description LIKE ?)"
        params.append(f"%{keyword}%")
        params.append(f"%{keyword}%")

    # 排序 (可选，如果需要)
    query += " ORDER BY crawled_at DESC" # 默认按爬取时间降序

    # 分页 (limit 和 offset)
    if limit is not None:
        query += f" LIMIT {int(limit)}"
    if offset is not None:
        query += f" OFFSET {int(offset)}"

    cursor.execute(query, tuple(params))
    funds = cursor.fetchall()
    conn.close()
    return [dict(fund) for fund in funds] # 将 Row 对象转换为字典列表

def create_search_profile(name, industry=None, company_age=None, turnover=None,
employees=None):
    """
    创建新的搜索档案。
    :return: 新插入记录的 ID，如果名称已存在则返回 None。
    """
```

```
    conn = get_db_connection()
    cursor = conn.cursor()
    try:
        cursor.execute("""
            INSERT INTO search_profiles (name, industry, company_age, turnover,
employees)
            VALUES (?, ?, ?, ?, ?)
        """, (name, industry, company_age, turnover, employees))
        conn.commit()
        print(f"搜索档案 '{name}' 创建成功。")
        return cursor.lastrowid # 返回新插入记录的 ID
    except sqlite3.IntegrityError:
        print(f"错误: 搜索档案 '{name}' 已存在。")
        return None
    finally:
        conn.close()

def get_search_profiles(profile_id=None):
    """
    查询搜索档案。
    :param profile_id: 档案 ID，可选，如果提供则查询指定档案，否则查询所有档案。
    :return: 搜索档案列表 (字典列表)。
    """
    conn = get_db_connection()
    cursor = conn.cursor()
    query = "SELECT * FROM search_profiles WHERE 1=1"
    params = []
    if profile_id:
        query += " AND id = ?"
        params.append(profile_id)
    cursor.execute(query, tuple(params))
    profiles = cursor.fetchall()
    conn.close()
    return [dict(profile) for profile in profiles]

def update_search_profile(profile_id, name=None, industry=None,
company_age=None, turnover=None, employees=None):
    """
    更新指定 ID 的搜索档案。
```

```
:return: True 如果更新成功，False 如果档案未找到或名称已存在。
"""
conn = get_db_connection()
cursor = conn.cursor()
updates = []
params = []
if name is not None:
    updates.append("name = ?")
    params.append(name)
if industry is not None:
    updates.append("industry = ?")
    params.append(industry)
if company_age is not None:
    updates.append("company_age = ?")
    params.append(company_age)
if turnover is not None:
    updates.append("turnover = ?")
    params.append(turnover)
if employees is not None:
    updates.append("employees = ?")
    params.append(employees)

if not updates:
    print("没有提供更新字段。")
    conn.close()
    return False

query = f"UPDATE search_profiles SET {', '.join(updates)} WHERE id = ?"
params.append(profile_id)

try:
    cursor.execute(query, tuple(params))
    conn.commit()
    if cursor.rowcount > 0:
        print(f"搜索档案 ID {profile_id} 更新成功。")
        return True
    else:
        print(f"未找到搜索档案 ID {profile_id}。")
        return False
```

```
    except sqlite3.IntegrityError:
        print(f"错误: 搜索档案名称 '{name}' 已存在。")
        return False
    finally:
        conn.close()

def delete_search_profile(profile_id):
    """
    删除指定 ID 的搜索档案。
    :return: True 如果删除成功，False 如果档案未找到。
    """
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("DELETE FROM search_profiles WHERE id = ?", (profile_id,))
    conn.commit()
    if cursor.rowcount > 0:
        print(f"搜索档案 ID {profile_id} 删除成功。")
        return True
    else:
        print(f"未找到搜索档案 ID {profile_id}。")
        return False
    finally:
        conn.close()

if __name__ == "__main__":
    init_db()

    # 示例数据
    sample_funds_df = pd.DataFrame([
        {
            "id": hashlib.md5(b"fund1_unique_title_ptj").hexdigest(),
            "title": "创新科技资金",
            "description": "支持初创企业的科技创新项目",
            "link": "http://example.com/fund1",
            "source": "ptj.de"
        },
        {
            "id": hashlib.md5(b"fund2_unique_title_eustartups").hexdigest(),
            "title": "欧洲数字转型基金",
```

```
            "description": "帮助中小企业进行数字化转型",
            "link": "http://example.com/fund2",
            "source": "eu-startups.com"
        },
        {
            "id": hashlib.md5(b"fund3_unique_title_foerderdatenbank").hexdigest(),
            "title": "德国科研项目资助",
            "description": "面向大学和研究机构的科研资助",
            "link": "http://example.com/fund3",
            "source": "foerderdatenbank.de"
        }
    ])

    save_funds_data(sample_funds_df)

    print("\n所有资金项目:")
    all_funds = get_funds()
    for fund in all_funds:
        print(f"Title: {fund['title']}, Source: {fund['source']}")

    create_search_profile("初创企业资金", industry="科技", company_age="<3年")
    create_search_profile("中小企业发展", industry="制造")
    create_search_profile("初创企业资金", industry="科技") # 尝试创建重复名称的档案

    print("\n所有搜索档案:")
    all_profiles = get_search_profiles()
    for profile in all_profiles:
        print(f"ID: {profile['id']}, Name: {profile['name']}, Industry: {profile['industry']}")

    update_search_profile(1, name="AI科技初创企业资金", industry="人工智能",
employees=">50")
    delete_search_profile(2)

    print("\n更新和删除后的搜索档案:")
    all_profiles = get_search_profiles()
    for profile in all_profiles:
        print(f"ID: {profile['id']}, Name: {profile['name']}, Industry: {profile['industry']}")
```

7.2.3 app.py 文件结构

现在，创建 app.py 文件，并添加以下基本代码。它将定义 API 路由，并与之前创建的数据库操作函数以及爬虫调度、数据处理、匹配引擎等模块进行交互。

```
# app.py

from flask import Flask, request, jsonify, send_from_directory
import pandas as pd
import pickle # 用于加载特征和 TF-IDF vectorizer
import logging

# 导入自定义模块
from database import init_db, get_funds, create_search_profile, \
    get_search_profiles, update_search_profile, delete_search_profile, save_funds_data
from main_scraper import run_all_scrapers
from data_processing import run_data_pipeline, preprocess_text # 从
data_processing 导入 run_data_pipeline 和 preprocess_text
from matching_engine import parse_user_profile, rule_based_matching_system # 从
matching_engine 导入匹配系统

# 配置日志
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

app = Flask(__name__, static_folder='static')

# 全局变量，用于存储加载的数据和模型
global_data = {
    "processed_df": None,
    "tfidf_vectorizer": None,
    "final_features_matrix": None
}

def load_ml_assets():
    """
    加载机器学习相关的资产 (处理后的数据, TF-IDF Vectorizer, 特征矩阵)。
    这些文件应在数据处理管道运行后生成。
    """
    try:
```

```
        with open('processed_df.pkl', 'rb') as f:
            global_data["processed_df"] = pickle.load(f)
            logging.info("已加载 processed_df.pkl")
        with open('tfidf_vectorizer.pkl', 'rb') as f:
            global_data["tfidf_vectorizer"] = pickle.load(f)
            logging.info("已加载 tfidf_vectorizer.pkl")
        with open('final_features.pkl', 'rb') as f:
            global_data["final_features_matrix"] = pickle.load(f)
            logging.info("已加载 final_features.pkl (特征矩阵)")
        return True
    except FileNotFoundError as e:
        logging.error(f"加载机器学习资产失败: {e}. 请确保已运行数据处理管道。")
        return False
    except Exception as e:
        logging.error(f"加载机器学习资产时发生未知错误: {e}")
        return False

# 初始化数据库
init_db()

# 在应用启动时尝试加载机器学习资产
# 可以在 app.before_first_request 或直接在这里调用
with app.app_context(): # 确保在应用上下文中执行
    if not load_ml_assets():
        logging.warning("机器学习资产未完全加载，某些匹配功能可能受限。")

@app.route("/")
def index():
    """
    提供前端主页面。
    """
    return send_from_directory(app.static_folder, "index.html")

# --- 资金项目 API ---
@app.route("/api/funds", methods=["GET"])
def api_get_funds():
    """
    获取资金项目列表。支持按来源和关键词筛选，以及分页。
    """
```

```
    source = request.args.get("source")
    keyword = request.args.get("keyword")
    limit = request.args.get("limit", type=int)
    offset = request.args.get("offset", type=int)

    funds = get_funds(source=source, keyword=keyword, limit=limit, offset=offset)
    return jsonify(funds)

# --- 搜索档案 API ---
@app.route("/api/profiles", methods=["POST"])
def api_create_profile():
    """
    创建新的搜索档案。
    """
    data = request.get_json()
    if not data or "name" not in data:
        return jsonify({"error": "缺少搜索档案名称"}), 400

    name = data["name"]
    industry = data.get("industry")
    company_age = data.get("company_age")
    turnover = data.get("turnover")
    employees = data.get("employees")

    profile_id = create_search_profile(name, industry, company_age, turnover,
employees)
    if profile_id:
        return jsonify({"message": "搜索档案创建成功", "id": profile_id}), 201
    else:
        return jsonify({"error": "搜索档案名称已存在或创建失败"}), 409 # 409 Conflict

@app.route("/api/profiles", methods=["GET"])
def api_get_profiles():
    """
    获取所有搜索档案。
    """
    profiles = get_search_profiles()
    return jsonify(profiles)
```

```
@app.route("/api/profiles/<int:profile_id>", methods=["GET"])
def api_get_profile_by_id(profile_id):
    """
    根据 ID 获取指定搜索档案。
    """
    profile = get_search_profiles(profile_id=profile_id)
    if profile:
        return jsonify(profile[0])
    else:
        return jsonify({"error": "搜索档案未找到"}), 404

@app.route("/api/profiles/<int:profile_id>", methods=["PUT"])
def api_update_profile(profile_id):
    """
    更新指定 ID 的搜索档案。
    """
    data = request.get_json()
    if not data:
        return jsonify({"error": "缺少更新数据"}), 400

    success = update_search_profile(profile_id, **data)
    if success:
        return jsonify({"message": "搜索档案更新成功"}), 200
    else:
        return jsonify({"error": "搜索档案更新失败或未找到"}), 404

@app.route("/api/profiles/<int:profile_id>", methods=["DELETE"])
def api_delete_profile(profile_id):
    """
    删除指定 ID 的搜索档案。
    """
    success = delete_search_profile(profile_id)
    if success:
        return jsonify({"message": "搜索档案删除成功"}), 200
    else:
        return jsonify({"error": "搜索档案删除失败或未找到"}), 404

# --- 爬虫触发 API ---
@app.route("/api/scrape", methods=["POST"])
```

```
def api_trigger_scrape():
    """
    触发一次数据爬取任务，并运行数据处理管道，将数据保存到数据库和生成特征文件。
    """
    logging.info("API 触发爬取任务开始...")
    try:
        # 1. 运行所有爬虫
        combined_df = run_all_scrapers()
        if combined_df.empty:
            logging.info("没有爬取到新数据。")
            return jsonify({"message": "没有爬取到新数据"}), 200

        # 2. 将原始爬取数据保存到数据库 (funds 表)
        save_funds_data(combined_df)
        logging.info(f"成功保存 {len(combined_df)} 条原始资金项目数据到数据库。")

        # 3. 运行数据处理管道，生成特征文件和 processed_df
        # 注意: file_paths_to_process 应该指向 main_scraper.py 输出的 CSV 文件
        # 为了简化，我们假设 run_data_pipeline 可以直接处理 combined_df
        # 实际更严谨的做法是先将 combined_df 保存为 CSV，再由 data_pipeline 读取
        # 或者调整 data_pipeline 接收 DataFrame

        # 暂时将 combined_df 保存为 CSV，然后调用 data_pipeline
        temp_csv_path = 'temp_combined_funds_data.csv'
        combined_df.to_csv(temp_csv_path, index=False, encoding='utf-8-sig')

        final_features_matrix, processed_df_output, tfidf_vectorizer_output =
run_data_pipeline([temp_csv_path])

        # 保存处理后的资产
        if final_features_matrix is not None and processed_df_output is not None and
tfidf_vectorizer_output is not None:
            with open('final_features.pkl', 'wb') as f:
                pickle.dump(final_features_matrix, f)
            with open('tfidf_vectorizer.pkl', 'wb') as f:
                pickle.dump(tfidf_vectorizer_output, f)
            with open('processed_df.pkl', 'wb') as f:
                pickle.dump(processed_df_output, f)
            logging.info("数据处理与特征工程管道完成，资产已保存。")
```

```
            # 更新全局数据
            global_data["processed_df"] = processed_df_output
            global_data["tfidf_vectorizer"] = tfidf_vectorizer_output
            global_data["final_features_matrix"] = final_features_matrix
            logging.info("全局机器学习资产已更新。")

            return jsonify({"message": f"成功爬取、处理并保存 {len(combined_df)} 条资金项目数据，并更新了机器学习资产。"}), 200
        else:
            logging.error("数据处理管道未能成功生成所有资产。")
            return jsonify({"error": "数据处理失败，未能更新机器学习资产。"}), 500

    except Exception as e:
        logging.exception("爬虫或数据处理管道运行失败。") # 使用 exception 会打印详细堆栈信息
        return jsonify({"error":f"爬虫或数据处理管道运行失败: {str(e)}"}), 500

# --- 资金项目匹配 API ---
@app.route("/api/match_funds", methods=["POST"])
def api_match_funds():
    """
    根据用户画像匹配资金项目。
    """
    user_profile_raw = request.get_json()
    if not user_profile_raw:
        return jsonify({"error": "缺少用户画像数据"}), 400

    if global_data["processed_df"] is None or \
       global_data["tfidf_vectorizer"] is None or \
       global_data["final_features_matrix"] is None:
        return jsonify({"error": "机器学习资产未加载，请先运行数据爬取和处理管道。"}), 503 # Service Unavailable

    logging.info(f"接收到用户画像进行匹配: {user_profile_raw}")
    try:
        # 1. 解析用户画像
        parsed_user_profile = parse_user_profile(user_profile_raw)
```

```
        # 2. 调用基于规则与相似度的匹配系统
        matched_projects_df = rule_based_matching_system(
            global_data["processed_df"],
            parsed_user_profile,
            global_data["tfidf_vectorizer"],
            global_data["final_features_matrix"]
        )

        if matched_projects_df.empty:
            return jsonify({"message": "没有找到符合条件的项目。"})

        # 3. 格式化输出结果 (只返回必要字段)
        results = []
        for index, row in matched_projects_df.iterrows():
            results.append({
                "id": row['id'],
                "title": row['title'],
                "description_summary": row['description'], # 使用原始描述或截断后的
                "link": row['link'],
                "source": row['source'],
                "crawled_at": row['crawled_at'] if 'crawled_at' in row else 'N/A', # 确保字段存
在
                "match_score": float(row['combined_score']) if 'combined_score' in row else
None,
                "text_similarity_score": float(row['text_similarity_score']) if
'text_similarity_score' in row else None
            })

        # 可选: 对结果进行排序 (在 rule_based_matching_system 中已经排序)
        # results_sorted = sorted(results, key=lambda x: x['match_score'] or 0,
reverse=True)

        return jsonify(results), 200

    except Exception as e:
        logging.exception("资金项目匹配失败。")
        return jsonify({"error":f"资金项目匹配失败: {str(e)}"}), 500
```

```
if __name__ == "__main__":
    # 在生产环境中，请将 debug=True 替换为 debug=False
    # 并且使用 Gunicorn 等生产级 WSGI 服务器来运行应用
    app.run(debug=True, host='0.0.0.0', port=5000)
```

代码解释：

1. 导入必要的库和自定义模块：
    - Flask, request, jsonify, send_from_directory：Flask 框架的核心组件，用于创建应用、处理请求、返回 JSON 响应以及提供静态文件。
    - database 模块中的函数：用于与 SQLite 数据库进行交互。
    - main_scraper.run_all_scrapers：用于触发所有爬虫。
    - data_processing.run_data_pipeline：用于运行数据处理管道，生成特征文件。
    - matching_engine.parse_user_profile, rule_based_matching_system：用于解析用户画像和调用匹配系统。
    - pandas, pickle：用于数据处理和加载/保存 .pkl 文件。
    - logging：用于日志记录。
2. app = Flask(__name__, static_folder='static')：创建一个 Flask 应用实例，并指定静态文件目录为 static。
3. 全局变量 **global_data**：用于在应用内存中存储加载的 processed_df、tfidf_vectorizer 和 final_features_matrix。这样做是为了避免每次请求时都重新加载这些大文件，提高性能。
4. load_ml_assets()：一个辅助函数，在应用启动时加载预先生成好的机器学习资产（数据和模型）。如果文件不存在，会打印错误信息并返回 False，这会影响 api_match_funds 的功能。
5. init_db()：在应用启动时调用，确保数据库和表已初始化。
6. @app.route("/")：将根路径 / 映射到 index() 函数，该函数会返回 static 文件夹中的 index.html 文件。
7. @app.route("/api/funds", methods=["GET"])：获取资金项目的 API 接口。支持 source、keyword 筛选和 limit、offset 分页。
8. @app.route("/api/profiles", ...)：一系列用于搜索档案的 CRUD（创建、读取、更新、删除）API 接口。遵循 RESTful 原则，使用 POST、GET、PUT、DELETE 方法。
9. @app.route("/api/scrape", methods=["POST"])：核心 **API** 之一，用于触发完整的爬取和数据处理管道。
    - 首先调用 run_all_scrapers() 获取最新数据。
    - 然后调用 save_funds_data() 将原始爬取数据保存到 SQLite 数据库的 funds 表。
    - 接着调用 run_data_pipeline() 对爬取到的数据进行清洗、预处理和特征工程，生成 final_features.pkl、tfidf_vectorizer.pkl 和 processed_df.pkl 文件。
    - 成功后，更新全局变量 global_data 中的资产，使匹配 API 能够使用最新的数据和

模型。
    - 处理过程中可能出现的任何错误都会被捕获并返回 500 状态码和错误信息。
10. @app.route("/api/match_funds", methods=["POST"])：核心 **API** 之一，用于根据用户画像匹配资金项目。
    - 接收前端发送的 JSON 格式用户画像数据。
    - 首先检查 global_data 中所需的机器学习资产是否已加载。如果未加载，返回 503 Service Unavailable。
    - 调用 parse_user_profile() 对用户画像进行解析和标准化。
    - 调用 rule_based_matching_system()（在第六章定义）执行匹配逻辑，返回匹配并排序后的 DataFrame。
    - 将匹配结果格式化为 JSON 并返回。
11. app.run(debug=True, host='0.0.0.0', port=5000)：启动 Flask 开发服务器。
    - debug=True：开启调试模式。在开发过程中，当代码发生变化时，服务器会自动重启，并且会提供详细的错误信息。在生产环境中，请务必关闭调试模式。
    - host='0.0.0.0'：使 Flask 应用可以从任何 IP 地址访问，而不仅仅是本地主机。这在 Docker 容器或虚拟机中运行应用时非常有用。
    - port=5000：指定应用监听的端口号。

7.3 运行后端 API

在命令行终端中，进入您的项目根目录（funding_search_tool），激活虚拟环境，然后运行 app.py：

```
cd funding_search_tool
source venv/bin/activate # macOS/Linux
# .\venv\Scripts\activate # Windows
python app.py
```

如果一切正常，您将看到类似以下的输出：

```
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: XXX-XXX-XXX
```

现在，您的后端 API 已经在 http://127.0.0.1:5000 上运行。您可以使用浏览器或像 Postman、Insomnia 这样的 API 测试工具来测试这些 API 接口。

示例 **API** 测试：

1. 访问根路径：在浏览器中打开 http://127.0.0.1:5000/，您应该能看到 index.html 页面（如果前端文件已创建）。
2. 创建搜索档案 **(POST /api/profiles)**：
   - 使用 Postman 或 Insomnia 发送 POST 请求到 http://127.0.0.1:5000/api/profiles。
   - 请求体选择 raw 和 JSON，内容如下：

     ```
     {
       "name": "我的第一个档案",
       "industry": "IT",
       "company_age": "<5年",
       "turnover": "<1M",
       "employees": "<10"
     }
     ```

   - 您应该收到 201 Created 状态码和成功消息。
3. 获取所有搜索档案 **(GET /api/profiles)**：
   - 在浏览器中打开 http://127.0.0.1:5000/api/profiles，您应该看到刚刚创建的档案。
4. 触发爬虫 **(POST /api/scrape)**：
   - 发送 POST 请求到 http://127.0.0.1:5000/api/scrape（请求体可以为空 JSON {}）。
   - 这将触发 main_scraper.py 中的所有爬虫运行，并调用数据处理管道，将数据保存到 SQLite 数据库和生成 .pkl 特征文件。您可以在终端中看到爬虫和数据处理的输出。这一步可能耗时较长，请耐心等待。
5. 获取资金项目 **(GET /api/funds)**：
   - 在浏览器中打开 http://127.0.0.1:5000/api/funds，您应该看到爬取到的资金项目列表。
   - 您也可以尝试带参数查询：
     http://127.0.0.1:5000/api/funds?source=ptj.de&keyword=创新。
6. 匹配资金项目 **(POST /api/match_funds)**：
   - 发送 POST 请求到 http://127.0.0.1:5000/api/match_funds。
   - 请求体选择 raw 和 JSON，内容如下（根据您的测试档案和期望匹配的项目来调整）：

     ```
     {
       "industry": "software",
     ```

```
  "company_age_years": 3,
  "region": "germany",
  "funding_needed_eur": 180000,
  "num_employees_min": 10,
  "num_employees_max": 60,
  "keywords": ["innovation", "digital"]
}
```

- 您应该收到 200 OK 状态码和匹配到的资金项目列表（如果存在）。

通过本章的学习，您已经成功构建并运行了项目的后端 API，实现了与数据库的交互以及爬虫和数据处理管道的触发，并初步实现了匹配功能。在下一章中，我们将着手开发前端界面，让用户能够通过友好的界面与您的 API 进行交互。

第八章：前端界面开发（HTML/CSS/JavaScript）

前端界面是用户与资金项目搜索工具直接交互的窗口。本章将指导您如何使用原生的 HTML、CSS 和 JavaScript 来构建一个简洁、实用的前端页面，实现搜索档案的管理和资金项目的展示。我们将重点讲解如何通过 JavaScript 发送 AJAX 请求与后端 Flask API 进行通信，并动态更新页面内容。

8.1 前端技术栈概述与项目文件结构

8.1.1 前端技术栈概述

为了保持教程的初学者友好性，我们将避免使用复杂的前端框架（如 React, Vue, Angular），而是专注于 Web 开发的基础三剑客：

- **HTML (HyperText Markup Language)**：负责网页的结构和内容。它定义了页面上的各种元素，如标题、段落、表单、按钮、表格等。
- **CSS (Cascading Style Sheets)**：负责网页的样式和布局。它控制着元素的颜色、字体、大小、位置等视觉表现，使页面看起来美观且易于阅读。
- **JavaScript**：负责网页的交互和动态行为。它能够响应用户的操作（如点击、输入），修改页面内容，以及与后端服务器进行异步通信 (AJAX)。

8.1.2 项目文件结构

为了更好地组织前端代码，我们将在项目根目录下创建一个 static 文件夹，用于存放前端相关的 HTML、CSS 和 JavaScript 文件。最终的项目结构将如下所示：

```
funding_search_tool/
├── venv/                  # 虚拟环境
├── scrapers/              # 爬虫模块目录
│   └── ...
├── main_scraper.py        # 主爬虫调度模块
├── data_processing.py     # 数据处理与特征工程模块
├── evaluation.py          # 模型评估模块
├── matching_engine.py     # 匹配引擎模块
├── database.py            # 数据库操作模块
├── funds.db               # SQLite数据库文件
├── final_features.pkl     # 最终特征矩阵 (pickle 文件)
├── tfidf_vectorizer.pkl   # TF-IDF 向量化器 (pickle 文件)
├── processed_df.pkl       # 处理后的原始数据 DataFrame (pickle 文件)
├── X_train.pkl            # 训练集特征 (可选)
├── y_train.pkl            # 训练集标签 (可选)
├── ...                    # 其他模型和数据文件
├── app.py                 # Flask 应用主文件
```

```
└── static/                # 前端静态文件目录
    ├── index.html          # 主页面 HTML
    ├── style.css           # 页面样式 CSS
    └── script.js           # 页面交互 JavaScript
```

8.2 构建 HTML 页面 (index.html)

index.html 将是我们的主页面，它将包含搜索档案的管理界面、数据爬取控制区域和资金项目的展示区域。

在 static 文件夹下创建 index.html 文件，并添加以下内容：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>资金项目搜索工具</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div class="container">
        <h1>资金项目搜索工具</h1>

        <!-- 搜索档案管理模块 -->
        <section class="profile-management">
            <h2>搜索档案管理</h2>
            <form id="profile-form">
                <input type="hidden" id="profile-id">
                <label for="profile-name">档案名称:</label>
                <input type="text" id="profile-name" required>

                <label for="profile-industry">行业:</label>
                <input type="text" id="profile-industry" placeholder="例如: Software,
Biotech">

                <label for="profile-company-age">公司年龄:</label>
                <input type="text" id="profile-company-age" placeholder="例如: <3年, 3-10
年">
```

```
            <label for="profile-turnover">营业额:</label>
            <input type="text" id="profile-turnover" placeholder="例如: <1M, 1M-10M">

            <label for="profile-employees">员工数量:</label>
            <input type="text" id="profile-employees" placeholder="例如: <10, 10-50">

            <button type="submit" id="save-profile-btn">保存档案</button>
            <button type="button" id="clear-form-btn">清空表单</button>
        </form>
        <p id="profile-status-message" class="status-message"></p>

        <h3>现有搜索档案</h3>
        <ul id="profile-list">
            <!-- 搜索档案将在这里动态加载 -->
        </ul>
    </section>

    <!-- 数据爬取控制模块 -->
    <section class="scraper-control">
        <h2>数据爬取与处理</h2>
        <button id="trigger-scrape-btn">立即触发爬取与处理</button>
        <p id="scrape-status-message" class="status-message"></p>
        <p class="info-text">(此操作会获取最新数据，并重新处理数据生成机器学习资产，耗时较长，请耐心等待。)</p>
    </section>

    <!-- 资金项目搜索/匹配模块 -->
    <section class="fund-search">
        <h2>资金项目匹配</h2>
        <p class="info-text">请填写您的需求，系统将为您匹配最相关的资金项目。</p>
        <form id="match-form">
            <label for="match-industry">行业:</label>
            <input type="text" id="match-industry" placeholder="例如: Software Development">

            <label for="match-company-age-years">公司年龄 (年):</label>
            <input type="number" id="match-company-age-years" placeholder="例如: 2">
```

```
            <label for="match-turnover-min">最低年营业额 (欧元):</label>
            <input type="number" id="match-turnover-min" placeholder="例如: 50000">

            <label for="match-turnover-max">最高年营业额 (欧元):</label>
            <input type="number" id="match-turnover-max" placeholder="例如: 500000">

            <label for="match-employees-min">最少员工数:</label>
            <input type="number" id="match-employees-min" placeholder="例如: 5">

            <label for="match-employees-max">最多员工数:</label>
            <input type="number" id="match-employees-max" placeholder="例如: 49">

            <label for="match-region">地区 (德国、柏林等):</label>
            <input type="text" id="match-region" placeholder="例如: Germany, Berlin">

            <label for="match-keywords">关键词 (逗号分隔):</label>
            <input type="text" id="match-keywords" placeholder="例如: innovation, digitalization">

            <label for="match-funding-needed">期望资助金额 (欧元):</label>
            <input type="number" id="match-funding-needed" placeholder="例如: 100000">

            <button type="submit" id="search-funds-btn">匹配资金项目</button>
        </form>
        <p id="match-status-message" class="status-message"></p>

        <div id="fund-results">
            <h3>匹配结果</h3>
            <table id="funds-table">
                <thead>
                    <tr>
                        <th>标题</th>
                        <th>描述摘要</th>
                        <th>来源</th>
                        <th>匹配得分</th>
```

```
                    <th>文本相似度</th>
                    <th>链接</th>
                    <th>爬取时间</th>
                </tr>
            </thead>
            <tbody>
                <!-- 资金项目数据将在这里动态加载 -->
            </tbody>
        </table>
        <p id="no-funds-message" style="display: none; ">没有找到符合条件的资金项目。</p>
    </div>
</section>
</div>

<script src="script.js"></script>
</body>
</html>
```

HTML 结构解释：

- <head>：包含页面的元信息，如字符集、视口设置、页面标题和 CSS 样式表的链接。
- ：一个容器，用于包裹所有页面内容，方便 CSS 布局。
- profile-management 区块：
  - 包含一个表单 (profile-form)，用于创建或编辑搜索档案。表单中包含各种输入字段和保存/清空按钮。
  - 一个无序列表 (profile-list)，用于动态显示已保存的搜索档案。
  - 一个状态信息段落 (profile-status-message)。
- scraper-control 区块：
  - 一个按钮 (trigger-scrape-btn)，用于手动触发数据爬取和处理。
  - 一个状态信息段落 (scrape-status-message)。
  - 一个提示信息 (info-text)。
- fund-search 区块：
  - 一个表单 (match-form)，用于用户输入详细需求进行匹配。
  - 一个按钮 (search-funds-btn)，用于触发匹配。
  - 一个状态信息段落 (match-status-message)。
  - 一个表格 (funds-table)，用于动态显示匹配到的资金项目结果。初始时表格体为空。
  - 一个提示信息 (no-funds-message)，当没有匹配结果时显示。

- <script src="script.js"></script>：在 <body> 结束标签之前引入 JavaScript 文件，确保 HTML 元素加载完成后再执行脚本。

8.3 添加 CSS 样式 (style.css)

在 static 文件夹下创建 style.css 文件，并添加以下内容，为页面添加基本样式：

```
/* style.css */

body {
    font-family: 'Arial', sans-serif;
    line-height: 1.6;
    margin: 0;
    padding: 20px;
    background-color: #f4f4f4;
    color: #333;
}

.container {
    max-width: 1200px;
    margin: 0 auto;
    background: #fff;
    padding: 30px;
    border-radius: 12px;
    box-shadow: 0 4px 20px rgba(0, 0, 0, 0.1);
}

h1 {
    color: #0056b3;
    text-align: center;
    margin-bottom: 40px;
    font-size: 2.5em;
    border-bottom: 2px solid #0056b3;
    padding-bottom: 15px;
}

h2 {
    color: #007bff;
    margin-top: 30px;
    margin-bottom: 20px;
```

```
    border-left: 5px solid #007bff;
    padding-left: 10px;
    font-size: 1.8em;
}

h3 {
    color: #333;
    margin-top: 25px;
    margin-bottom: 15px;
    font-size: 1.4em;
}

section {
    margin-bottom: 40px;
    padding-bottom: 30px;
    border-bottom: 1px dashed #e0e0e0;
}

section:last-child {
    border-bottom: none;
    margin-bottom: 0;
    padding-bottom: 0;
}

form label {
    display: block;
    margin-bottom: 8px;
    font-weight: bold;
    color: #555;
}

form input[type="text"],
form input[type="number"],
form select {
    width: calc(100% - 22px);
    padding: 12px;
    margin-bottom: 18px;
    border: 1px solid #ddd;
    border-radius: 6px;
```

```
    box-sizing: border-box; /* 确保 padding 不会增加宽度 */
    font-size: 1em;
}

form button {
    background-color: #007bff;
    color: white;
    padding: 12px 20px;
    border: none;
    border-radius: 6px;
    cursor: pointer;
    font-size: 1.1em;
    margin-right: 15px;
    transition: background-color 0.3s ease, transform 0.2s ease;
    box-shadow: 0 2px 5px rgba(0, 123, 255, 0.3);
}

form button:hover {
    background-color: #0056b3;
    transform: translateY(-2px);
}

form button:active {
    transform: translateY(0);
}

form button#clear-form-btn {
    background-color: #6c757d;
    box-shadow: 0 2px 5px rgba(108, 117, 125, 0.3);
}

form button#clear-form-btn:hover {
    background-color: #5a6268;
}

ul#profile-list {
    list-style: none;
    padding: 0;
    margin-top: 20px;
```

```
}

ul#profile-list li {
    background-color: #e9ecef;
    margin-bottom: 12px;
    padding: 15px;
    border-radius: 8px;
    display: flex;
    justify-content: space-between;
    align-items: center;
    box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1);
    transition: transform 0.2s ease;
}

ul#profile-list li:hover {
    transform: translateX(5px);
}

ul#profile-list li span {
    flex-grow: 1;
    font-weight: 500;
}

ul#profile-list li div {
    display: flex;
    gap: 8px;
}

ul#profile-list li button {
    padding: 8px 12px;
    font-size: 0.9em;
    margin-right: 0; /* Override default margin */
    box-shadow: none; /* Override default shadow */
}

ul#profile-list li button.edit-btn {
    background-color: #ffc107;
    color: #333;
}
```

```
ul#profile-list li button.edit-btn:hover {
    background-color: #e0a800;
}

ul#profile-list li button.delete-btn {
    background-color: #dc3545;
    color: white;
}

ul#profile-list li button.delete-btn:hover {
    background-color: #c82333;
}

#trigger-scrape-btn {
    background-color: #28a745;
    color: white;
    padding: 15px 25px;
    font-size: 1.2em;
    display: block;
    width: fit-content;
    margin: 20px auto; /* Center the button */
    box-shadow: 0 3px 8px rgba(40, 167, 69, 0.4);
}

#trigger-scrape-btn:hover {
    background-color: #218838;
}

#trigger-scrape-btn:disabled {
    background-color: #a0a0a0;
    cursor: not-allowed;
    box-shadow: none;
    transform: none;
}

.status-message {
    margin-top: 15px;
    font-style: italic;
```

```
    text-align: center;
    min-height: 1.5em; /* Prevent layout shift */
}

.status-message.success {
    color: #28a745;
}

.status-message.error {
    color: #dc3545;
}

.status-message.info {
    color: #6c757d;
}

.info-text {
    font-size: 0.9em;
    color: #777;
    text-align: center;
    margin-top: -10px;
    margin-bottom: 20px;
}

#funds-table {
    width: 100%;
    border-collapse: collapse;
    margin-top: 25px;
    background-color: #fff;
    border-radius: 8px;
    overflow: hidden; /* For rounded corners on table */
    box-shadow: 0 2px 10px rgba(0, 0, 0, 0.08);
}

#funds-table th,
#funds-table td {
    border: 1px solid #e0e0e0;
    padding: 12px;
    text-align: left;
```

```
    font-size: 0.95em;
}

#funds-table th {
    background-color: #f2f2f2;
    color: #555;
    font-weight: bold;
    text-transform: uppercase;
    letter-spacing: 0.05em;
}

#funds-table tbody tr:nth-child(even) {
    background-color: #f9f9f9;
}

#funds-table tbody tr:hover {
    background-color: #f1f1f1;
}

#funds-table td a {
    color: #007bff;
    text-decoration: none;
    word-break: break-all; /* 链接过长时换行 */
}

#funds-table td a:hover {
    text-decoration: underline;
}

#no-funds-message {
    color: #666;
    text-align: center;
    margin-top: 20px;
    padding: 15px;
    background-color: #f0f0f0;
    border-radius: 8px;
}

/* Responsive adjustments */
```

```
@media (max-width: 768px) {
    .container {
        padding: 15px;
    }

    h1 {
        font-size: 2em;
        margin-bottom: 20px;
    }

    h2 {
        font-size: 1.5em;
    }

    form input[type="text"],
    form input[type="number"],
    form select,
    form button {
        width: 100%;
        margin-right: 0;
        margin-bottom: 10px;
    }

    form button {
        display: block;
        box-sizing: border-box;
    }

    ul#profile-list li {
        flex-direction: column;
        align-items: flex-start;
        gap: 10px;
    }

    ul#profile-list li div {
        width: 100%;
        justify-content: flex-end;
    }
```

```
    #funds-table thead {
        display: none; /* Hide header on small screens */
    }

    #funds-table, #funds-table tbody, #funds-table tr, #funds-table td {
        display: block;
        width: 100%;
    }

    #funds-table tr {
        margin-bottom: 15px;
        border: 1px solid #e0e0e0;
        border-radius: 8px;
        box-shadow: 0 2px 5px rgba(0,0,0,0.05);
    }

    #funds-table td {
        text-align: right;
        padding-left: 50%;
        position: relative;
    }

    #funds-table td::before {
        content: attr(data-label);
        position: absolute;
        left: 10px;
        width: calc(50% - 20px);
        padding-right: 10px;
        white-space: nowrap;
        text-align: left;
        font-weight: bold;
        color: #777;
    }
}
```

CSS 样式解释：

- 通用样式：设置了字体、行高、背景色和基本文本颜色，使页面整体美观。
- .container：定义了页面的最大宽度、居中、背景、内边距、圆角和阴影，使页面看起来

更整洁和专业。

- 标题样式：h1, h2, h3 均设置了统一的颜色、字体大小和边框，增加页面层次感。
- 表单元素：对 label, input, select 和 button 进行了样式设置，使其具有统一的外观、适当的内边距和圆角，并添加了阴影和过渡效果，提升用户体验。
- profile-list：美化了搜索档案列表的显示，包括背景色、内边距、圆角和按钮样式，并使用 flexbox 布局。
- #trigger-scrape-btn：特别设计了爬取按钮的样式，使其更醒目，并添加了 disabled 状态样式。
- .status-message：统一的状态消息样式，根据消息类型显示不同颜色。
- 表格样式：为 funds-table 添加了边框、内边距、背景色和悬停效果，使其更易读，并加入了圆角和阴影。
- 响应式设计 **(@media 查询)**：针对小屏幕设备（最大宽度 768px），调整了布局和元素样式，使页面在移动设备上也能良好显示。表格在小屏幕上会转换为堆叠式布局，每个单元格前显示对应的标题 (data-label)，提供更好的可读性。

8.4 编写 JavaScript 交互逻辑 (script.js)

在 static 文件夹下创建 script.js 文件，并添加以下内容。这将是前端的核心部分，负责处理用户交互、发送 AJAX 请求和动态更新页面。

```
// script.js

document.addEventListener("DOMContentLoaded", () => {
    // --- 获取 DOM 元素 ---
    const profileForm = document.getElementById("profile-form");
    const profileIdInput = document.getElementById("profile-id");
    const profileNameInput = document.getElementById("profile-name");
    const profileIndustryInput = document.getElementById("profile-industry");
    const profileCompanyAgeInput =
document.getElementById("profile-company-age");
    const profileTurnoverInput = document.getElementById("profile-turnover");
    const profileEmployeesInput = document.getElementById("profile-employees");
    const saveProfileBtn = document.getElementById("save-profile-btn");
    const clearFormBtn = document.getElementById("clear-form-btn");
    const profileList = document.getElementById("profile-list");
    const profileStatusMessage =
document.getElementById("profile-status-message");

    const triggerScrapeBtn = document.getElementById("trigger-scrape-btn");
    const scrapeStatusMessage =
```

```
document.getElementById("scrape-status-message");

    const matchForm = document.getElementById("match-form");
    const matchIndustryInput = document.getElementById("match-industry");
    const matchCompanyAgeInput =
document.getElementById("match-company-age-years");
    const matchTurnoverMinInput = document.getElementById("match-turnover-min");
    const matchTurnoverMaxInput =
document.getElementById("match-turnover-max");
    const matchEmployeesMinInput =
document.getElementById("match-employees-min");
    const matchEmployeesMaxInput =
document.getElementById("match-employees-max");
    const matchRegionInput = document.getElementById("match-region");
    const matchKeywordsInput = document.getElementById("match-keywords");
    const matchFundingNeededInput =
document.getElementById("match-funding-needed");
    const matchStatusMessage = document.getElementById("match-status-message");

    const fundsTableBody = document.querySelector("#funds-table tbody");
    const noFundsMessage = document.getElementById("no-funds-message");

    const API_BASE_URL = "/api"; // 后端 API 的基础 URL

    // --- 辅助函数 ---
    // 显示消息
    const showMessage = (element, message, type = "info") => {
        element.textContent = message;
        element.className = `status-message ${type}`; // Add status type class
        setTimeout(() => {
            element.textContent = "";
            element.className = "status-message";
        }, 5000);
    };

    // 清空搜索档案表单
    const clearProfileForm = () => {
        profileIdInput.value = "";
        profileNameInput.value = "";
```

```
    profileIndustryInput.value = "";
    profileCompanyAgeInput.value = "";
    profileTurnoverInput.value = "";
    profileEmployeesInput.value = "";
    saveProfileBtn.textContent = "保存档案";
    profileStatusMessage.textContent = ""; // Clear status message
    profileStatusMessage.className = "status-message";
};

// 清空匹配表单
const clearMatchForm = () => {
    matchIndustryInput.value = "";
    matchCompanyAgeInput.value = "";
    matchTurnoverMinInput.value = "";
    matchTurnoverMaxInput.value = "";
    matchEmployeesMinInput.value = "";
    matchEmployeesMaxInput.value = "";
    matchRegionInput.value = "";
    matchKeywordsInput.value = "";
    matchFundingNeededInput.value = "";
    matchStatusMessage.textContent = "";
    matchStatusMessage.className = "status-message";
    fundsTableBody.innerHTML = ""; // Clear search results
    noFundsMessage.style.display = "none";
};



// --- 搜索档案管理 ---
// 加载所有搜索档案
const loadProfiles = async () => {
    profileList.innerHTML = "<li>正在加载档案...</li>";
    try {
        const response = await fetch(`${API_BASE_URL}/profiles`);
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const profiles = await response.json();
        profileList.innerHTML = ""; // 清空现有列表
```

```
        if (profiles.length === 0) {
            profileList.innerHTML = "<li>暂无搜索档案。</li>";
            return;
        }

        profiles.forEach(profile => {
            const li = document.createElement("li");
            li.innerHTML = `
                <span>
                    ${profile.name}
                    (${profile.industry || 'N/A'},
                    ${profile.company_age || 'N/A'},
                    ${profile.turnover || 'N/A'},
                    ${profile.employees || 'N/A'})
                </span>
                <div>
                    <button class="edit-btn" data-id="${profile.id}">编辑</button>
                    <button class="delete-btn" data-id="${profile.id}">删除</button>
                </div>
            `;
            profileList.appendChild(li);
        });
    } catch (error) {
        console.error("加载搜索档案失败:", error);
        profileList.innerHTML = "<li>加载搜索档案失败。请检查后端API。</li>";
        showMessage(profileStatusMessage, "加载搜索档案失败。", "error");
    }
};

// 提交搜索档案表单 (创建或更新)
profileForm.addEventListener("submit", async (e) => {
    e.preventDefault();
    saveProfileBtn.disabled = true;

    const id = profileIdInput.value;
    const name = profileNameInput.value.trim();
    const industry = profileIndustryInput.value.trim();
    const company_age = profileCompanyAgeInput.value.trim();
    const turnover = profileTurnoverInput.value.trim();
```

```
    const employees = profileEmployeesInput.value.trim();

    const profileData = {
        name,
        industry: industry || null,
        company_age: company_age || null,
        turnover: turnover || null,
        employees: employees || null,
    };

    try {
        let response;
        if (id) {
            // 更新现有档案
            response = await fetch(`${API_BASE_URL}/profiles/${id}`, {
                method: "PUT",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(profileData),
            });
        } else {
            // 创建新档案
            response = await fetch(`${API_BASE_URL}/profiles`, {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(profileData),
            });
        }

        const result = await response.json();
        if (response.ok) {
            showMessage(profileStatusMessage, result.message || "操作成功!",
"success");
            clearProfileForm();
            loadProfiles(); // 重新加载档案列表
        } else {
            showMessage(profileStatusMessage, result.error || "操作失败!", "error");
        }
    } catch (error) {
        console.error("保存搜索档案失败:", error);
```

```
            showMessage(profileStatusMessage, "保存搜索档案时发生网络错误。",
"error");
        } finally {
            saveProfileBtn.disabled = false;
        }
    });

    // 清空表单按钮事件
    clearFormBtn.addEventListener("click", clearProfileForm);

    // 编辑和删除按钮事件委托
    profileList.addEventListener("click", async (e) => {
        if (e.target.classList.contains("edit-btn")) {
            const profileId = e.target.dataset.id;
            try {
                const response = await fetch(`${API_BASE_URL}/profiles/${profileId}`);
                if (!response.ok) {
                    throw new Error(`HTTP error! status: ${response.status}`);
                }
                const profile = await response.json();
                profileIdInput.value = profile.id;
                profileNameInput.value = profile.name;
                profileIndustryInput.value = profile.industry || "";
                profileCompanyAgeInput.value = profile.company_age || "";
                profileTurnoverInput.value = profile.turnover || "";
                profileEmployeesInput.value = profile.employees || "";
                saveProfileBtn.textContent = "更新档案";
                showMessage(profileStatusMessage, `正在编辑档案: ${profile.name}`,
"info");
            } catch (error) {
                console.error("加载档案进行编辑失败:", error);
                showMessage(profileStatusMessage, "加载档案进行编辑失败。", "error");
            }
        } else if (e.target.classList.contains("delete-btn")) {
            const profileId = e.target.dataset.id;
            if (confirm("确定要删除此搜索档案吗?")) {
                try {
                    const response = await fetch(`${API_BASE_URL}/profiles/${profileId}`, {
                        method: "DELETE",
```

```
                });
                const result = await response.json();
                if (response.ok) {
                    showMessage(profileStatusMessage, result.message || "删除成功!",
"success");
                    loadProfiles(); // 重新加载档案列表
                } else {
                    showMessage(profileStatusMessage, result.error || "删除失败!", "error");
                }
            } catch (error) {
                console.error("删除档案失败:", error);
                showMessage(profileStatusMessage, "删除档案时发生网络错误。",
"error");
            }
        }
    }
  });

  // --- 数据爬取与处理控制 ---
  triggerScrapeBtn.addEventListener("click", async () => {
    triggerScrapeBtn.disabled = true;
    scrapeStatusMessage.textContent = "正在触发爬取与处理，请稍候...";
    scrapeStatusMessage.className = "status-message info";

    try {
        const response = await fetch(`${API_BASE_URL}/scrape`, {
            method: "POST",
            headers: { "Content-Type": "application/json" }, // POST 请求通常需要
Content-Type
            body: JSON.stringify({}) // 空 JSON 对象作为请求体
        });

        const result = await response.json();
        if (response.ok) {
            showMessage(scrapeStatusMessage, result.message || "爬取与处理成功!",
"success");
        } else {
            showMessage(scrapeStatusMessage, result.error || "爬取与处理失败!",
"error");
```

```
            }
        } catch (error) {
            console.error("触发爬取与处理失败:", error);
            showMessage(scrapeStatusMessage, "触发爬取与处理时发生网络错误。",
"error");
        } finally {
            triggerScrapeBtn.disabled = false;
        }
    });

    // --- 资金项目匹配 ---
    matchForm.addEventListener("submit", async (e) => {
        e.preventDefault();
        searchFundsBtn.disabled = true;
        fundsTableBody.innerHTML = ""; // 清空现有表格内容
        noFundsMessage.style.display = "none";
        matchStatusMessage.textContent = "正在匹配资金项目，请稍候...";
        matchStatusMessage.className = "status-message info";

        const userProfileData = {
            industry: matchIndustryInput.value.trim() || null,
            company_age_years: matchCompanyAgeInput.value ?
parseInt(matchCompanyAgeInput.value) : null,
            turnover_eur_min: matchTurnoverMinInput.value ?
parseFloat(matchTurnoverMinInput.value) : null,
            turnover_eur_max: matchTurnoverMaxInput.value ?
parseFloat(matchTurnoverMaxInput.value) : null,
            num_employees_min: matchEmployeesMinInput.value ?
parseInt(matchEmployeesMinInput.value) : null,
            num_employees_max: matchEmployeesMaxInput.value ?
parseInt(matchEmployeesMaxInput.value) : null,
            region: matchRegionInput.value.trim() || null,
            keywords: matchKeywordsInput.value.trim().split(',').map(kw =>
kw.trim()).filter(kw => kw) || [],
            funding_needed_eur: matchFundingNeededInput.value ?
parseFloat(matchFundingNeededInput.value) : null
        };

        try {
```

```
        const response = await fetch(`${API_BASE_URL}/match_funds`, {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify(userProfileData),
        });

        const result = await response.json();
        if (response.ok) {
            if (result.message && result.message === "没有找到符合条件的项目。") {
                noFundsMessage.style.display = "block";
                showMessage(matchStatusMessage, "没有找到符合条件的资金项目。",
"info");
            } else if (Array.isArray(result) && result.length > 0) {
                result.forEach(fund => {
                    const tr = document.createElement("tr");
                    tr.innerHTML = `
                        <td data-label="标题">${fund.title}</td>
                        <td data-label="描述摘要">${fund.description_summary || '无'}</td>
                        <td data-label="来源">${fund.source}</td>
                        <td data-label="匹配得分">${fund.match_score !== null ?
fund.match_score.toFixed(4) : 'N/A'}</td>
                        <td data-label="文本相似度">${fund.text_similarity_score !== null ?
fund.text_similarity_score.toFixed(4) : 'N/A'}</td>
                        <td data-label="链接"><a href="${fund.link}" target="_blank">查看
</a></td>
                        <td data-label="爬取时间">${new
Date(fund.crawled_at).toLocaleString()}</td>
                    `;
                    fundsTableBody.appendChild(tr);
                });
                showMessage(matchStatusMessage, `找到 ${result.length} 个匹配项目。`,
"success");
            } else {
                noFundsMessage.style.display = "block"; // Fallback if result is not an array
or empty
                showMessage(matchStatusMessage, "没有找到符合条件的资金项目。",
"info");
            }
        } else {
```

```
            showMessage(matchStatusMessage, result.error || "匹配资金项目失败!",
"error");
        }
    } catch (error) {
        console.error("匹配资金项目失败:", error);
        showMessage(matchStatusMessage, "匹配资金项目时发生网络错误。",
"error");
    } finally {
        searchFundsBtn.disabled = false;
    }
  });

  // --- 页面加载时初始化 ---
  loadProfiles();
  // clearMatchForm(); // 页面加载时清空匹配表单 (可选)
});
```

JavaScript 代码解释：

1. **DOMContentLoaded** 事件监听：确保在 HTML 文档完全加载和解析后才执行 JavaScript 代码，避免操作不存在的 DOM 元素。
2. 获取 **DOM** 元素：通过 document.getElementById() 和 document.querySelector() 获取页面上需要操作的 HTML 元素。
3. API_BASE_URL：定义后端 API 的基础 URL，方便管理和修改。
4. showMessage()：一个辅助函数，用于在页面上显示临时的成功或错误消息，并根据类型添加 CSS 类（success, error, info）。
5. clearProfileForm()：用于清空搜索档案表单的内容。
6. clearMatchForm()：用于清空资金项目匹配表单的内容和结果表格。
7. loadProfiles()：
    - 异步函数，使用 fetch API 向后端 /api/profiles 接口发送 GET 请求，获取所有搜索档案。
    - 获取到数据后，清空现有的档案列表，然后遍历数据，动态创建 <li> 元素并添加到页面中。
8. profileForm.addEventListener("submit", ...)：
    - 监听表单的提交事件。e.preventDefault() 阻止表单的默认提交行为（页面刷新）。
    - 根据 profileIdInput.value 是否存在来判断是创建新档案（POST 请求）还是更新现有档案（PUT 请求）。
    - 使用 fetch API 发送 POST 或 PUT 请求，请求体为 JSON 格式的档案数据。
    - 根据后端响应的状态码和内容，显示成功或失败消息，并重新加载档案列表。

9. profileList.addEventListener("click", ...)：
    - 使用事件委托，监听档案列表中的编辑和删除按钮的点击事件。
    - 编辑按钮：发送 GET 请求获取指定 ID 的档案详情，然后将数据填充到表单中，并将保存按钮的文本改为“更新档案”。
    - 删除按钮：弹出确认框，如果用户确认，则发送 DELETE 请求删除指定 ID 的档案，并重新加载档案列表。
10. triggerScrapeBtn.addEventListener("click", ...)：
    - 监听“立即触发爬取与处理”按钮的点击事件。
    - 禁用按钮并显示状态信息，防止重复点击。
    - 发送 POST 请求到后端 /api/scrape 接口，触发爬虫和数据处理管道运行。
    - 根据后端响应显示爬取结果，并在操作完成后重新启用按钮。
11. matchForm.addEventListener("submit", ...)：
    - 监听“匹配资金项目”按钮的点击事件。
    - 禁用按钮，清空旧结果，显示加载状态。
    - 从表单中收集用户输入的画像数据，将其格式化为 JSON 对象。
    - 发送 POST 请求到后端 /api/match_funds 接口，进行资金项目匹配。
    - 接收后端 API 返回的 JSON 数据。如果匹配到项目，则遍历数据，动态创建 <tr> 元素并添加到表格中。如果没有任何匹配项目，则显示“没有找到符合条件的资金项目。”的提示信息。
    - 在 td 元素中添加 data-label 属性，用于响应式表格在小屏幕上显示列标题。
    - 在操作完成后重新启用按钮。
12. 初始化：在脚本加载完成后，立即调用 loadProfiles() 函数，加载并显示所有已保存的搜索档案。

8.5 运行前端应用

1. 确保文件结构正确：请再次检查您的项目文件结构，确保 index.html, style.css, script.js 都位于 static 文件夹内。
2. 启动 **Flask** 后端：在终端中，进入项目根目录，激活虚拟环境，然后运行 app.py：
   ```
   cd funding_search_tool
   source venv/bin/activate # macOS/Linux
   # .\venv\Scripts\activate # Windows
   python app.py
   ```
   注意：如果这是您第一次运行 app.py，它会尝试初始化数据库。当您触发爬取时，它还会运行完整的爬虫和数据处理管道，这可能需要一些时间来完成并生成 .pkl 文件。
3. 访问前端页面：在浏览器中打开 http://127.0.0.1:5000/。您应该能看到我们刚刚创建的前端界面。

现在，您可以尝试在前端页面上进行操作：

- 创建新的搜索档案：填写表单并点击“保存档案”按钮。您应该能看到档案列表更新。
- 编辑/删除搜索档案：点击档案旁边的“编辑”或“删除”按钮。
- 触发爬取与处理：点击“立即触发爬取与处理”按钮，等待爬取和数据处理完成（可能需要一些时间，状态消息会提示）。
- 匹配资金项目：在“资金项目匹配”区域填写您的需求（例如，行业、地区、关键词等），然后点击“匹配资金项目”按钮。您应该能看到表格中显示匹配到的资金项目（如果有的话）。

通过本章的学习，您已经掌握了如何使用 HTML 构建页面结构，使用 CSS 美化页面，以及使用 JavaScript 实现页面交互和与后端 API 通信。至此，我们已经完成了资金项目搜索工具的核心功能开发。

第九章：走向实用：项目部署、存储与运维初步

一个项目从原型到实用，需要考虑数据的持久化存储、运行环境的封装与部署，以及日常的维护。本章将介绍如何为我们的 AI 资助项目发现系统选择合适的数据存储方案，如何使用 Docker 进行容器化以确保环境一致性，并初步探讨（可选的）云函数部署和代码版本管理策略。这些是项目能够稳定运行并持续迭代的基础。

9.1 数据存储方案：为数据安个家

在项目的不同阶段和针对不同类型的数据，我们需要选择合适的存储方案。

9.1.1 爬取数据的存储

- 原始数据 **(Raw Data)**：爬虫直接抓取到的 HTML 页面、API 返回的原始 JSON 等。这些数据应尽量完整保存，便于问题追溯、重新解析或验证。
  - 存储方式：可以直接存为文本文件（如 .html, .json），或者打包（如 .zip, .tar.gz）。如果数据量大，可以考虑按日期或来源分目录存储。
- 初步提取的结构化数据：从原始数据中解析提取出的、尚未深度清洗的结构化信息（如上一章爬虫模块输出的 CSV 或 JSON Lines 文件）。
  - 存储方式：CSV 文件、JSON Lines 文件。这些格式易于被 Pandas 等工具读取和处理。
- 清洗后和特征工程后的数据：经过数据清洗、转换、特征提取后的数据集，是模型训练和匹配系统直接使用的数据。
  - 存储方式：
    - .pkl (Pickle)：Python 特有的序列化格式，能完整保存 DataFrame 对象及其数据类型，读写速度较快，但不跨语言。
    - Parquet：高效的列式存储格式，压缩比高，读写速度快，尤其适合大数据分析，被 Spark、Hadoop 等生态广泛支持。需要 pyarrow 或 fastparquet 库。
    - .h5 (HDF5)：层次化数据格式，适合存储大规模、多类型数据集。
    - 如果数据量非常大或需要频繁查询，可以存入数据库。

9.1.2 结构化数据存储选型（用于存储最终的项目库和用户信息等）

- **SQLite**：
  - 优点：轻量级，服务器无关（数据存储在单个 .db 文件中），Python 内置 sqlite3 模块直接支持，无需额外安装数据库服务，配置简单。非常适合小型项目、本地开发、原型验证和初学者入门。
  - 缺点：并发写入性能较低，不适合高并发、多用户同时写入的场景。功能相对大型数据库较少。
  - 使用：在第七章的 database.py 中已经详细演示了 SQLite 的使用。
- （可选）**PostgreSQL / MySQL**：
  - 优点：功能完善、性能强大、稳定可靠的关系型数据库管理系统（RDBMS）。支持复

杂 SQL 查询、事务处理、高并发访问、用户权限管理等。适合生产环境和需要处理大量结构化数据的应用。
    - 缺点：需要单独安装、配置和维护数据库服务器。学习曲线比 SQLite 陡峭。
    - **Python** 交互：需要安装相应的数据库驱动库（如 psycopg2 for PostgreSQL, mysql-connector-python for MySQL）。通常会配合 ORM (Object-Relational Mapper) 工具如 SQLAlchemy 来简化数据库操作，将 Python 对象映射到数据库表。
- （可选）**MongoDB (NoSQL** 数据库**)**：
    - 优点：面向文档的 NoSQL 数据库，以类 JSON 的 BSON 格式存储数据。模式灵活（schemaless），适合存储结构多变或嵌套的数据。水平扩展性好。
    - 缺点：事务支持不如关系型数据库强。数据一致性模型与关系型数据库不同。
    - **Python** 交互：使用 pymongo 库。
- （可选）对象存储 **MinIO**：模拟云端存储
    - 用途：适合存储非结构化或半结构化的大型文件，如爬虫下载的原始网页副本、图片、PDF 文档附件，以及处理后的大型数据集（如 Parquet 文件）、机器学习模型文件等。
    - **MinIO** 简介：一个开源的高性能对象存储服务，与 Amazon S3 API 完全兼容。可以在本地服务器、虚拟机或 Kubernetes 集群中部署。
    - 参考项目进展报告：该报告中提到使用 MinIO 来模拟 GCS (Google Cloud Storage) 的功能，用于存储爬取的数据。这是一个很好的实践，可以在本地开发和测试云存储相关的流程。
    - 安装与配置（使用 **Docker** 快速启动）：
        1. 拉取 **MinIO** 镜像并运行容器：
           ```
           docker run -d \
             -p 9000:9000 \
             -p 9001:9001 \
             --name minio-server \
             -e "MINIO_ROOT_USER=YOUR_ACCESS_KEY" \
             -e "MINIO_ROOT_PASSWORD=YOUR_SECRET_KEY" \
             -v /path/to/your/minio/data:/data \
             minio/minio server /data --console-address ":9001"
           ```
           （将 YOUR_ACCESS_KEY, YOUR_SECRET_KEY 替换为您自己的凭证，/path/to/your/minio/data 替换为本地数据存储路径）。
        2. 访问 **MinIO** 控制台：在浏览器中打开 http://localhost:9001，使用设置的 Access Key 和 Secret Key 登录。在控制台中可以创建存储桶（Buckets）。
    - **Python MinIO** 客户端使用：
      ```
      import os
      from minio import Minio
      ```

```
from minio.error import S3Error
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# MinIO 配置，建议从环境变量或配置文件读取，这里仅为演示
MINIO_ENDPOINT = os.getenv("MINIO_ENDPOINT", "localhost:9000")
MINIO_ACCESS_KEY = os.getenv("MINIO_ACCESS_KEY", "minioadmin") # 替换为您的 Access Key
MINIO_SECRET_KEY = os.getenv("MINIO_SECRET_KEY", "minioadmin") # 替换为您的 Secret Key
BUCKET_NAME = "funding-data-bucket" # 替换为您的 bucket 名称

def upload_to_minio(file_path, object_name_in_bucket):
    """
    将文件上传到 MinIO。
    :param file_path: 本地文件路径
    :param object_name_in_bucket: 在 MinIO 中存储的对象名称（包括路径）
    """
    try:
        client = Minio(
            MINIO_ENDPOINT,
            access_key=MINIO_ACCESS_KEY,
            secret_key=MINIO_SECRET_KEY,
            secure=False # 如果 MinIO 未使用 HTTPS，则设为 False
        )

        # 检查存储桶是否存在，如果不存在则创建
        found = client.bucket_exists(BUCKET_NAME)
        if not found:
            client.make_bucket(BUCKET_NAME)
            logging.info(f"存储桶 '{BUCKET_NAME}' 已创建。")
        else:
            logging.info(f"存储桶 '{BUCKET_NAME}' 已存在。")

        # 上传文件
        if os.path.exists(file_path):
            client.fput_object(BUCKET_NAME, object_name_in_bucket, file_path)
```

```
            logging.info(f"文件 '{file_path}' 已成功上传为 '{object_name_in_bucket}'
到存储桶 '{BUCKET_NAME}'。")
        else:
            logging.error(f"本地文件 '{file_path}' 不存在，无法上传。")
    except S3Error as exc:
        logging.error(f"MinIO 操作发生 S3 错误: {exc}")
    except Exception as e:
        logging.error(f"连接 MinIO 或进行操作时发生未知错误: {e}")

def download_from_minio(object_name_in_bucket, download_file_path):
    """
    从 MinIO 下载文件。
    """
    try:
        client = Minio(
            MINIO_ENDPOINT,
            access_key=MINIO_ACCESS_KEY,
            secret_key=MINIO_SECRET_KEY,
            secure=False
        )
        client.fget_object(BUCKET_NAME, object_name_in_bucket,
download_file_path)
        logging.info(f"对象 '{object_name_in_bucket}' 已从存储桶
'{BUCKET_NAME}' 下载到 '{download_file_path}'。")
    except S3Error as exc:
        logging.error(f"MinIO 下载发生 S3 错误: {exc}")
    except Exception as e:
        logging.error(f"连接 MinIO 或进行下载时发生未知错误: {e}")

# 示例用法
if __name__ == "__main__":
    test_file_content = "This is a test file for MinIO upload."
    test_local_path = "test_minio_upload.txt"
    test_object_name = "test_folder/my_test_object.txt"
    download_target_path = "downloaded_minio_file.txt"

    with open(test_local_path, "w") as f:
        f.write(test_file_content)
```

```
upload_to_minio(test_local_path, test_object_name)
download_from_minio(test_object_name, download_target_path)

if os.path.exists(test_local_path):
    os.remove(test_local_path)
if os.path.exists(download_target_path):
    print(f"Downloaded content: {open(download_target_path, 'r').read()}")
    os.remove(download_target_path)
```

9.2 项目容器化与环境一致性（Docker）：打包你的应用

Docker 是一种容器化技术，可以将应用程序及其所有依赖（库、运行时、系统工具等）打包到一个轻量级、可移植的容器中。这确保了应用在不同环境（开发、测试、生产 ）中都能以相同的方式运行，解决了“在我机器上能跑”的问题。

9.2.1 Dockerfile 编写详解

Dockerfile 是一个文本文件，包含了一系列指令，用于告诉 Docker 如何构建镜像。以下是一个针对本项目 Python 应用的示例 Dockerfile：

```
# Dockerfile

# 使用官方 Python 3.9 slim 版本作为基础镜像 (slim 版本更小)
FROM python:3.9-slim-buster

# 设置环境变量, 确保 Python 输出不被缓冲, 日志能及时显示
ENV PYTHONUNBUFFERED=1

# 在容器内创建并设置工作目录
WORKDIR /app

# 复制依赖文件 (requirements.txt) 到工作目录
# 这一步单独复制是为了利用 Docker 的层缓存机制:
# 只有当 requirements.txt 文件发生变化时, 才会重新执行后续的 RUN pip install 指令
COPY requirements.txt .

# 安装 Python 依赖
# --no-cache-dir: 不缓存下载的包, 减小镜像体积
RUN pip install --no-cache-dir -r requirements.txt \
    --trusted-host pypi.python.org \
```

```
    --trusted-host pypi.org \
    --trusted-host files.pythonhosted.org

# (可选) 如果需要下载 NLTK 等数据包, 可以在这里添加 RUN 指令
# RUN python -m nltk.downloader punkt stopwords wordnet omw-1.4

# 复制项目的所有代码到工作目录
# 注意: 建议使用 .dockerignore 文件排除不需要复制的文件 (如 .git, __pycache__, venv/, *.db, .DS_Store 等)
COPY . .

# 暴露 Flask 应用运行的端口
EXPOSE 5000

# 容器启动时执行的默认命令
# 这里我们运行 Flask 应用的主文件 app.py
CMD ["python", "app.py"]
```

Dockerfile 解释：

- FROM python:3.9-slim-buster：指定基础镜像。我们选择了一个轻量级的 Python 3.9 镜像，基于 Debian Buster 系统。slim 版本不包含完整的 Python 开发工具，使得镜像更小。
- ENV PYTHONUNBUFFERED=1：设置环境变量，确保 Python 的标准输出和标准错误流不会被缓冲，使得日志能够实时显示在 Docker 容器的日志中。
- WORKDIR /app：设置容器内的工作目录为 /app。后续的 COPY 和 CMD 指令都将在这个目录下执行。
- COPY requirements.txt .：将宿主机当前目录下的 requirements.txt 文件复制到容器的 /app 目录下。
- RUN pip install --no-cache-dir -r requirements.txt ...：在容器内执行命令安装 requirements.txt 中列出的所有 Python 依赖。--no-cache-dir 选项可以减少镜像大小。--trusted-host 参数用于解决某些网络环境下 pip 下载包可能遇到的证书问题。
- COPY . .：将宿主机当前目录下的所有文件（包括 app.py, database.py, main_scraper.py, static/ 等）复制到容器的 /app 目录下。这一步放在安装依赖之后，可以利用 Docker 的层缓存机制，当代码文件变化时，不需要重新安装依赖。
- EXPOSE 5000：声明容器会监听 5000 端口。这只是一个文档声明，并不会实际发布端口，需要在运行容器时进行端口映射。
- CMD ["python", "app.py"]：定义容器启动时执行的默认命令。这里是运行我们的

Flask 应用。

9.2.2 构建 Docker 镜像

在项目根目录(funding_search_tool)，打开终端，运行以下命令构建 Docker 镜像：

```
docker build -t funding-search-tool .
```

- docker build：构建 Docker 镜像的命令。
- -t funding-search-tool：为镜像指定一个标签(tag)，这里是 funding-search-tool。您可以根据需要命名。
- .：表示 Dockerfile 所在的路径，这里是当前目录。

构建过程可能需要一些时间，具体取决于您的网络速度和机器性能。成功构建后，您可以通过 docker images 命令查看已构建的镜像。

9.2.3 运行 Docker 容器

镜像构建完成后，您就可以运行容器了：

```
docker run -p 5000:5000 --name my-funding-app funding-search-tool
```

- docker run：运行 Docker 容器的命令。
- -p 5000:5000：端口映射。将宿主机的 5000 端口映射到容器的 5000 端口。这样，您就可以通过访问宿主机的 5000 端口来访问容器中运行的 Flask 应用。
- --name my-funding-app：为容器指定一个名称，方便管理。
- funding-search-tool：指定要运行的镜像名称。

现在，您可以通过浏览器访问 http://localhost:5000/ 来访问在 Docker 容器中运行的应用程序了。您会发现，即使在不同的机器上，只要安装了 Docker，应用程序都能以相同的方式运行。

常用 Docker 命令：

- docker ps：查看正在运行的容器。
- docker ps -a：查看所有容器(包括已停止的)。
- docker stop my-funding-app：停止名为 my-funding-app 的容器。
- docker start my-funding-app：启动名为 my-funding-app 的容器。
- docker rm my-funding-app：删除名为 my-funding-app 的容器(需要先停止)。
- docker rmi funding-search-tool：删除名为 funding-search-tool 的镜像(需要先删除所有基于该镜像的容器)。
- docker logs my-funding-app：查看容器的日志输出。

9.2.4 挂载卷 (Volumes)

在开发过程中，为了避免每次代码修改都重新构建镜像，可以使用卷挂载将宿主机的项目代码目录映射到容器内的工作目录。这样，在宿主机上修改代码会立即反映到容器内。

```
# 对于 Linux/macOS
docker run -p 5000:5000 --name my-funding-app -v "$(pwd):/app"
funding-search-tool

# 对于 Windows (PowerShell)
docker run -p 5000:5000 --name my-funding-app -v "${PWD}:/app"
funding-search-tool

# 对于 Windows (Command Prompt)
docker run -p 5000:5000 --name my-funding-app -v "%cd%:/app"
funding-search-tool
```

`-v "$(pwd):/app"`：将宿主机的当前目录（`$(pwd)` 或 `${PWD}` 或 `%cd%`）挂载到容器内的 `/app` 目录。

如果应用需要访问 `MinIO` 等在宿主机或其他容器中运行的服务，需要确保网络配置正确（例如，使用 `--network="host"` 或自定义 `Docker` 网络）。

9.3 自动化与定时任务

在实际应用中，数据爬取通常需要定期自动执行，例如每周一次。虽然在云环境中可以使用 `Google Cloud Functions` 或 `AWS Lambda` 等无服务器计算服务来触发定时任务，但对于本地开发和初学者，我们可以使用 `Python` 的 `schedule` 库或操作系统的 `cron (Linux)`/任务计划程序 (`Windows`) 来模拟实现。

9.3.1 使用 `Python schedule` 库

`schedule` 是一个轻量级的 `Python` 库，用于在 `Python` 脚本中创建简单的定时任务。它非常适合在应用程序内部进行简单的调度。

首先，确保您已安装 `schedule` 库：

```bash
pip install schedule
```

然后，创建一个新的文件 scheduler.py：

```
# scheduler.py

import schedule
import time
import logging
import os # 用于检查文件是否存在

# 导入自定义模块
from main_scraper import run_all_scrapers
from database import save_funds_data, init_db
from data_processing import run_data_pipeline

# 配置日志
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

def job():
    """
    定时任务：执行数据爬取和处理管道。
    """
    logging.info("\n--- 定时任务开始执行爬取和数据处理 ---")

    # 确保数据库已初始化
    try:
        init_db()
    except Exception as e:
        logging.error(f"数据库初始化失败: {e}")
        return

    # 1. 运行所有爬虫
    combined_df = run_all_scrapers()
    if combined_df.empty:
        logging.info("定时任务: 没有爬取到新数据。")
        return

    # 2. 将原始爬取数据保存到数据库 (funds 表)
    save_funds_data(combined_df)
```

```
    logging.info(f"定时任务: 成功保存 {len(combined_df)} 条原始资金项目数据到数据库。")

    # 3. 运行数据处理管道，生成特征文件和 processed_df
    temp_csv_path = 'temp_combined_funds_data_scheduled.csv'
    try:
        combined_df.to_csv(temp_csv_path, index=False, encoding='utf-8-sig')
        final_features_matrix, processed_df_output, tfidf_vectorizer_output = run_data_pipeline([temp_csv_path])

        # 保存处理后的资产
        if final_features_matrix is not None and processed_df_output is not None and tfidf_vectorizer_output is not None:
            with open('final_features.pkl', 'wb') as f:
                pickle.dump(final_features_matrix, f)
            with open('tfidf_vectorizer.pkl', 'wb') as f:
                pickle.dump(tfidf_vectorizer_output, f)
            with open('processed_df.pkl', 'wb') as f:
                pickle.dump(processed_df_output, f)
            logging.info("定时任务: 数据处理与特征工程管道完成，资产已保存。")
        else:
            logging.error("定时任务: 数据处理管道未能成功生成所有资产。")

    except Exception as e:
        logging.exception("定时任务: 数据处理管道运行失败。")
    finally:
        if os.path.exists(temp_csv_path):
            os.remove(temp_csv_path) # 清理临时文件

    logging.info("--- 定时任务执行完毕 ---")

# 每周一的 0 点 0 分执行一次
schedule.every().monday.at("00:00").do(job)

# 每天的 10 点 30 分执行一次 (可选)
# schedule.every().day.at("10:30").do(job)

# 每隔 10 分钟执行一次 (测试用)
# schedule.every(10).minutes.do(job)
```

```
logging.info("定时任务调度器已启动...")

if __name__ == "__main__":
    while True:
        schedule.run_pending()
        time.sleep(1) # 每秒检查一次待执行任务
```

scheduler.py 解释：

- job() 函数：包含了我们希望定时执行的任务，即调用 run_all_scrapers() 进行数据爬取，然后调用 save_funds_data() 将原始数据保存到数据库，最后运行 run_data_pipeline() 进行数据处理和特征工程，并保存相关资产。
- schedule.every().monday.at("00:00").do(job)：这是一个示例，表示每周一的 0 点 0 分执行 job 函数。schedule 库提供了非常灵活的调度方式，您可以根据需要调整。
- while True: schedule.run_pending(); time.sleep(1)：这是一个无限循环，它会每秒检查一次是否有待执行的任务，如果有，则执行。
- 注意：为了让 app.py 和 scheduler.py 能够共享 funds.db 以及 .pkl 文件，它们应该在同一个文件系统下运行，或者通过 Docker Volumes 共享。

运行 **scheduler.py**：

在终端中，进入项目根目录，激活虚拟环境，然后运行 scheduler.py：

```
python scheduler.py
```

这个脚本会一直运行，并在指定的时间自动执行爬取任务。在实际部署中，您可能需要使用 nohup 或 systemd 等工具让它在后台持续运行。

9.3.2 使用操作系统定时任务（**Cron/**任务计划程序）

对于更健壮的定时任务，您可以直接使用操作系统提供的功能：

- **Linux/macOS (Cron)**：
  - 打开终端，输入 crontab -e。
  - 在打开的文件中添加一行，例如：
    0 0 * * 1 /usr/bin/python3 /path/to/your/project/scheduler.py >> /path/to/your/project/cron.log 2>&1

    这表示每周一的 0 点 0 分执行 scheduler.py 脚本，并将输出重定向到 cron.log 文件。请将 /path/to/your/project/ 替换为您的项目实际路径。

- **Windows (任务计划程序)**:
  - 打开“任务计划程序”(Task Scheduler)。
  - 创建基本任务，指定触发器(例如每周、每天)和操作(运行程序)，程序路径指向您的 Python 解释器，参数指向 scheduler.py 脚本。

9.4 代码版本管理与协作(Git & GitHub):团队作业与历史追溯

良好的版本控制实践对于任何规模的项目都至关重要，即使是个人项目。它能帮助您跟踪变更、回溯历史、管理不同功能分支，并在未来进行团队协作。

9.4.1 分支策略 (Branching Strategy)

一个常用的分支模型是 Gitflow 的简化版:

- main(或 master)分支:始终保持稳定、可发布的代码。只接受来自 develop 分支或紧急修复分支的合并。
- develop 分支:作为日常开发的基础分支。所有新功能和常规修复都先合并到此分支。
- 功能分支 **(Feature branches)**: 从 develop 分支创建，用于开发某个特定的新功能(例如，feature/scraper-ptj, feature/similarity-matching)。开发完成后，通过 Pull Request (PR) 或 Merge Request (MR) 合并回 develop 分支。功能分支通常以 feature/ 或开发者名字等作为前缀。
- 修复分支 **(Bugfix branches)**: 用于修复 Bug。如果是修复 main 分支上的紧急 Bug，可以从 main 创建修复分支，修复后同时合并回 main 和 develop。如果是修复 develop 分支上的 Bug，则从 develop 创建。

[简化的Git分支策略流程示意的图片]

9.4.2 Commit 规范

- 原子性提交: 每次提交应尽量只包含一个逻辑上完整的功能或修复。
- 清晰的 **Commit Message**: 编写有意义的提交信息，清晰描述本次提交所做的更改。常用的格式是:
  <type>: <subject>

  (可选) <body>

  (可选) <footer>

  - <type>:如 feat(新功能)，fix(修复 Bug)，docs(文档修改)，style(代码格式调整)，refactor(代码重构)，test(测试相关)，chore(构建过程或辅助工具变动)。
  - <subject>:简明扼要地描述修改内容，通常不超过 50 个字符，祈使句，首字母小写。
  - <body>:更详细的描述，解释修改的原因和具体内容。

- <footer>：可用于记录相关的 Issue 编号等。
- 示例： feat: Add foerderdatenbank scraper module, fix: Correct parsing error for amount field in ptj scraper.

9.4.3 Pull Requests (PRs) / Merge Requests (MRs)

- 当一个功能分支或修复分支开发完成后，不要直接合并到 develop 或 main。应通过代码托管平台（如 GitHub）发起一个 Pull Request。
- PR 提供了一个代码审查 (Code Review) 的机会，团队成员（或自己，如果是个人项目）可以检查代码的质量、逻辑、风格是否符合要求，提出修改建议。
- 代码审查通过后，再将 PR 合并到目标分支。

9.4.4 GitHub/GitLab 等平台使用

- 将本地 Git 仓库与远程仓库（如在 GitHub 上创建的私有或公共仓库 ）关联。
- 定期将本地提交 git push 到远程仓库进行备份和协作。
- 使用平台的 Issue 跟踪功能来管理任务和 Bug。

参考项目进展报告中提到："Current code is available on GitHub, some changes awaiting review and merge"，这表明项目已在使用 GitHub 进行版本控制和协作。

通过合理的存储方案、容器化技术和规范的版本管理，可以大大提升项目的可维护性、可移植性和团队协作效率，为项目的长期发展打下良好基础。

第十章：精益求精：系统集成与全面测试策略

项目开发的后期阶段，重点在于将各个独立开发的模块有机地整合起来，形成一个完整、可运行的系统。同时，通过全面的测试来保证系统的质量和稳定性。本章将讨论如何将各模块集成，并详细讲解系统测试的各个层面。

10.1 模块集成与主流程构建：串珠成链

在前面的章节中，我们分别开发了数据爬取、数据清洗与特征工程、模型评估、AI 匹配、数据库操作以及 Flask API 等模块。现在需要将这些模块有效地组织和调用起来，形成一个完整的数据处理和智能匹配流水线。

10.1.1 设计清晰的模块调用关系和数据流

一个典型的处理流程可能如下：

1. 主程序启动：接收用户输入（如运行模式、用户画像参数等）。
2. 数据获取阶段（如果需要更新数据）：
    - 主程序调用爬虫模块（main_scraper.py）。
    - 爬虫模块针对各个数据源（ptj.de, eu-startups.com, foerderdatenbank.de, deutsche-digitale-bibliothek.de 等）执行爬取任务。
    - 原始数据初步保存为临时文件（如 CSV），或直接传递给下一步。
3. 数据处理与特征工程阶段：
    - 主程序调用数据处理与特征工程模块（data_processing.py），从临时文件或直接从上一步获取数据。
    - 进行深度清洗、格式转换、缺失值处理、去重等操作。
    - 调用特征工程功能，对文本数据进行 TF-IDF 向量化，处理结构化特征。
    - 生成最终的特征数据集 (final_features.pkl)、处理后的原始数据 (processed_df.pkl) 和 TF-IDF 向量化器 (tfidf_vectorizer.pkl)，并持久化存储。
4. 用户交互与匹配阶段（通过 **Flask API** 实现 ）：
    - Flask API 接收前端或定时任务触发的请求。
    - Flask 应用在启动时或请求时加载已保存的特征数据和 TF-IDF 向量化器。
    - Flask API 接收用户输入的画像信息。
    - 调用匹配/搜索模块（matching_engine.py）。
    - 匹配模块根据用户画像，在项目库中进行筛选和匹配计算。
    - 返回匹配结果列表。
5. 结果输出与评估阶段：
    - 前端界面接收 Flask API 返回的匹配结果，并以指定格式（如表格）输出给用户。
    - （可选，在开发/评估模式下）如果有测试集和真实标签，可以调用评估模块（evaluation.py），计算并输出系统的性能指标。

系统主要模块与数据流示意

```
graph TD
    A[用户输入/配置] --> B(主程序/Flask API);
    B -- 定时/手动触发 --> C[数据获取: Crawlers];
    C -- 原始数据 --> D[数据处理与特征工程];
    D -- 特征数据/processed_df/Vectorizer --> E[持久化存储: .pkl 文件];
    E -- 加载 --> F(匹配引擎);
    F -- 用户画像 --> F;
    F -- 匹配结果 --> B;
    B -- 结果输出 --> G[用户界面/API响应];
    B -- (开发/评估模式) --> H[模型评估];
```

10.1.2 编写主程序脚本（**main.py** 或类似）

虽然我们的 Flask 应用（app.py）已经承担了大部分主程序的职责，但为了更清晰地分离命令行工具和 Web 服务，或者在没有 Web 界面时执行特定任务，我们可以创建一个简单的命令行接口（CLI）脚本。

```
# cli.py (命令行接口脚本)

import argparse
import logging
import os
import pickle

# 导入自定义模块
from main_scraper import run_all_scrapers
from data_processing import run_data_pipeline, preprocess_text # preprocess_text 也在 data_processing 中
from database import init_db, save_funds_data, get_funds, get_search_profiles
from matching_engine import parse_user_profile, rule_based_matching_system
from evaluation import evaluate_matching_system
from evaluation_data_prep import prepare_evaluation_datasets # 从 evaluation_data_prep 导入

# 配置日志
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
```

```
# 文件路径定义
PROCESSED_DF_PATH = 'processed_df.pkl'
TFIDF_VECTORIZER_PATH = 'tfidf_vectorizer.pkl'
FINAL_FEATURES_PATH = 'final_features.pkl'
X_TRAIN_PATH = 'X_train.pkl'
Y_TRAIN_PATH = 'y_train.pkl'
X_TEST_PATH = 'X_test.pkl'
Y_TEST_PATH = 'y_test.pkl'
# ... 其他路径

def load_ml_assets_cli():
    """在 CLI 模式下加载机器学习资产。"""
    processed_df = None
    tfidf_vectorizer = None
    final_features_matrix = None
    try:
        if os.path.exists(PROCESSED_DF_PATH):
            with open(PROCESSED_DF_PATH, 'rb') as f:
                processed_df = pickle.load(f)
                logging.info(f"已加载 {PROCESSED_DF_PATH}")
        if os.path.exists(TFIDF_VECTORIZER_PATH):
            with open(TFIDF_VECTORIZER_PATH, 'rb') as f:
                tfidf_vectorizer = pickle.load(f)
                logging.info(f"已加载 {TFIDF_VECTORIZER_PATH}")
        if os.path.exists(FINAL_FEATURES_PATH):
            with open(FINAL_FEATURES_PATH, 'rb') as f:
                final_features_matrix = pickle.load(f)
                logging.info(f"已加载 {FINAL_FEATURES_PATH} (特征矩阵)")
        return processed_df, tfidf_vectorizer, final_features_matrix
    except Exception as e:
        logging.error(f"加载机器学习资产失败: {e}")
        return None, None, None

def main():
    parser = argparse.ArgumentParser(description="AI 资金项目搜索工具 CLI")

    # 子命令 (subparsers)
    subparsers = parser.add_subparsers(dest='mode', help='选择运行模式')
```

```
    # 爬取模式
    parser_crawl = subparsers.add_parser('crawl', help='运行数据爬取和处理管道')

    # 匹配模式
    parser_match = subparsers.add_parser('match', help='根据用户画像匹配资金项目')
    parser_match.add_argument('--industry', type=str, help='用户行业')
    parser_match.add_argument('--company_age_years', type=int, help='公司年龄 (年)')
    parser_match.add_argument('--turnover_eur_min', type=float, help='最低年营业额
(欧元)')
    parser_match.add_argument('--turnover_eur_max', type=float, help='最高年营业额
(欧元)')
    parser_match.add_argument('--num_employees_min', type=int, help='最少员工数')
    parser_match.add_argument('--num_employees_max', type=int, help='最多员工数')
    parser_match.add_argument('--region', type=str, help='地区 (如 Germany, Berlin)')
    parser_match.add_argument('--keywords', type=str, help='关键词 (逗号分隔)')
    parser_match.add_argument('--funding_needed_eur', type=float, help='期望资助金
额 (欧元)')
    parser_match.add_argument('--top_k', type=int, default=5, help='显示前 K 个匹配结
果')

    # 查询模式 (查询 funds 或 profiles)
    parser_query = subparsers.add_parser('query', help='查询数据库中的资金项目或搜
索档案')
    query_subparsers = parser_query.add_subparsers(dest='query_type', help='查询类
型')

    parser_query_funds = query_subparsers.add_parser('funds', help='查询资金项目')
    parser_query_funds.add_argument('--source', type=str, help='按来源筛选')
    parser_query_funds.add_argument('--keyword', type=str, help='按关键词筛选 (标题
或描述)')
    parser_query_funds.add_argument('--limit', type=int, default=10, help='返回最大记
录数')
    parser_query_funds.add_argument('--offset', type=int, default=0, help='偏移量')

    parser_query_profiles = query_subparsers.add_parser('profiles', help='查询搜索档案
')
    parser_query_profiles.add_argument('--id', type=int, help='按 ID 查询特定档案')
```

```
    # 评估模式 (可选，需要标签数据)
    parser_evaluate = subparsers.add_parser('evaluate', help='评估模型性能')

    args = parser.parse_args()

    # 确保数据库已初始化
    init_db()

    if args.mode == 'crawl':
        logging.info("--- 启动数据爬取和处理管道 ---")
        combined_df = run_all_scrapers()
        if not combined_df.empty:
            save_funds_data(combined_df) # 保存原始数据到数据库
            logging.info("原始爬取数据已保存到数据库。")

            # 运行数据处理管道并保存资产
            temp_csv_path = 'temp_combined_funds_data_cli.csv'
            combined_df.to_csv(temp_csv_path, index=False, encoding='utf-8-sig')
            final_features_matrix, processed_df_output, tfidf_vectorizer_output =
run_data_pipeline([temp_csv_path])

            if final_features_matrix is not None and processed_df_output is not None and
tfidf_vectorizer_output is not None:
                with open(FINAL_FEATURES_PATH, 'wb') as f:
pickle.dump(final_features_matrix, f)
                with open(TFIDF_VECTORIZER_PATH, 'wb') as f:
pickle.dump(tfidf_vectorizer_output, f)
                with open(PROCESSED_DF_PATH, 'wb') as f:
pickle.dump(processed_df_output, f)
                logging.info("数据处理管道完成，特征和向量化器已保存。")
            else:
                logging.error("数据处理管道未能成功生成所有资产。")
            os.remove(temp_csv_path) # 清理临时文件

        else:
            logging.info("没有爬取到数据。")
        logging.info("--- 数据爬取和处理完成 ---")

    elif args.mode == 'match':
```

```
        processed_df, tfidf_vectorizer, final_features_matrix = load_ml_assets_cli()
        if processed_df is None or tfidf_vectorizer is None or final_features_matrix is
None:
            logging.error("匹配所需的机器学习资产未加载。请先运行 'python cli.py crawl'。
")
            return

        user_profile_raw = {
            "industry": args.industry,
            "company_age_years": args.company_age_years,
            "turnover_eur_min": args.turnover_eur_min,
            "turnover_eur_max": args.turnover_eur_max,
            "num_employees_min": args.num_employees_min,
            "num_employees_max": args.num_employees_max,
            "region": args.region,
            "keywords": args.keywords.split(',') if args.keywords else [],
            "funding_needed_eur": args.funding_needed_eur
        }
        user_profile = parse_user_profile(user_profile_raw)
        logging.info(f"正在匹配用户画像: {user_profile}")

        matched_projects = rule_based_matching_system(
            processed_df, user_profile, tfidf_vectorizer, final_features_matrix
        )

        if not matched_projects.empty:
            print("\n--- 匹配到的资金项目 (Top %d) ---" % args.top_k)
            for i, row in matched_projects.head(args.top_k).iterrows():
                print(f"标题: {row['title']}")
                print(f"来源: {row['source']}")
                print(f"匹配得分: {row['combined_score']:.4f}")
                print(f"文本相似度: {row['text_similarity_score']:.4f}")
                print(f"链接: {row['link']}")
                print(f"描述: {row['description'][:150]}...") # 截断描述
                print("-" * 30)
        else:
            print("没有找到符合条件的资金项目。")

    elif args.mode == 'query':
```

```
        if args.query_type == 'funds':
            funds = get_funds(source=args.source, keyword=args.keyword, limit=args.limit,
offset=args.offset)
            if funds:
                print("\n--- 资金项目列表 ---")
                for fund in funds:
                    print(f"ID: {fund['id'][:8]}..., Title: {fund['title']}, Source: {fund['source']},
Link: {fund['link']}")
            else:
                print("未找到资金项目。")
        elif args.query_type == 'profiles':
            profiles = get_search_profiles(profile_id=args.id)
            if profiles:
                print("\n--- 搜索档案列表 ---")
                for profile in profiles:
                    print(f"ID: {profile['id']}, Name: {profile['name']}, Industry:
{profile['industry']}, Company Age: {profile['company_age']}")
            else:
                print("未找到搜索档案。")
        else:
            parser.print_help() # 默认打印 query 的帮助

    elif args.mode == 'evaluate':
        # 评估模式需要预先生成 X_test.pkl, y_test.pkl
        try:
            with open(FINAL_FEATURES_PATH, 'rb') as f: final_features_matrix =
pickle.load(f)
            # 在这里我们假设有一个方式来获取真实的标签数据 (例如，从人工标注的 CSV)
            # 为了演示，我们先模拟一个标签数组
            num_samples = final_features_matrix.shape[0]
            simulated_labels = np.random.choice([0, 1], size=num_samples, p=[0.9, 0.1]) #
模拟 10% 相关
            labels = simulated_labels # 这是您实际的 Ground Truth

            X_train, X_val, X_test, y_train, y_val, y_test =
prepare_evaluation_datasets(final_features_matrix, labels)

            if X_test is None or y_test is None:
                logging.error("无法加载测试数据集进行评估。")
```

```
            return

        # 这里可以加载训练好的模型并进行评估
        # 例如：
        # from ml_matching import train_and_evaluate_model # 导入训练函数
        # model = train_and_evaluate_model(X_train, y_train, X_val, y_val,
model_type='logistic_regression')
        # if model:
        #    y_pred = model.predict(X_test)
        #    evaluate_matching_system(y_test, y_pred, model_name="Trained ML
Model")

        # 或者只评估规则基线（简化演示）
        # 注意：rule_based_matching_system 返回的是 DataFrame，需要转换为标签

        # 为了演示，我们假设 y_pred 是通过某种匹配逻辑得到的
        # 这里的 y_pred 应该来自您的匹配系统对测试集项目的预测
        # 由于 rule_based_matching_system 是针对单个用户画像，这里需要更复杂的模
拟
        # 简化为：假设所有项目都针对一个通用用户画像进行匹配，然后转换为二分类标
签
        # 实际需要根据您的测试集中的 (项目, 用户画像) 对来生成预测

        # 为了 CLI 演示，我们使用随机预测作为示例
        y_pred_example = np.random.choice([0, 1], size=len(y_test), p=[0.7, 0.3])
        evaluate_matching_system(y_test, y_pred_example, model_name="CLI
Evaluation Demo")

      except FileNotFoundError:
        logging.error("评估所需的文件 (.pkl) 未找到。请先运行数据爬取和处理管道。")
      except Exception as e:
        logging.exception(f"评估模式运行失败: {e}")

  else:
    parser.print_help() # 如果没有指定模式，打印帮助信息

if __name__ == '__main__':
  main()
```

cli.py 解释：

- **argparse** 模块：用于解析命令行参数，允许用户通过命令行指定不同的运行模式（crawl, match, query, evaluate）和相应的参数。
- 子命令 **(subparsers)**：使得 CLI 工具可以有多个独立的子功能，每个子功能有自己的参数。
- **crawl** 模式：触发完整的爬取和数据处理管道，与 Flask API 中的 /api/scrape 逻辑类似，用于在命令行下独立运行。
- **match** 模式：接收用户画像参数，加载机器学习资产（processed_df, tfidf_vectorizer, final_features_matrix），然后调用 rule_based_matching_system 进行匹配，并在命令行打印结果。
- **query** 模式：允许用户直接查询 funds 表或 search_profiles 表的数据，方便调试和数据查看。
- **evaluate** 模式：用于评估系统或模型的性能。它需要加载测试集数据，并使用 evaluate_matching_system 函数进行评估。注意：这里的评估逻辑只是一个框架，您需要根据实际的标注数据和模型训练情况来填充 y_test 和 y_pred 的获取方式。
- 文件路径：定义了常用 .pkl 文件的路径，方便模块间的共享和管理。
- **load_ml_assets_cli()**：专门为 CLI 模式设计，用于加载 processed_df、tfidf_vectorizer 和 final_features_matrix。

如何运行 cli.py：
在项目根目录，激活虚拟环境：

- 运行爬取和数据处理：
  python cli.py crawl

  这会执行爬虫，并将数据保存到 funds.db 和生成 *.pkl 文件。
- 匹配资金项目：
  python cli.py match --industry "Software" --region "Germany" --keywords "innovation, AI" --top_k 3

  这会根据提供的用户画像进行匹配，并显示前 3 个匹配结果。
- 查询资金项目：
  python cli.py query funds --source "ptj.de" --keyword "创新" --limit 5

- 查询搜索档案：
  python cli.py query profiles --id 1

- 评估（需要 **.pkl** 文件）：
  python cli.py evaluate

（在运行此命令前，请确保您已经通过 cli.py crawl 或 app.py /api/scrape 触发了数据处理管道，生成了 final_features.pkl 等文件。并且，对于真正的评估，您需要替换 cli.py 中 evaluate 模式下的模拟标签 simulated_labels 为您真实的人工标注标签。）

10.2 系统测试：千锤百炼出真金

全面的测试是保证软件质量、发现潜在问题、确保系统按预期工作的关键环节。对于本项目，我们需要考虑以下几个层面的测试：

10.2.1 单元测试 (Unit Testing)

- 目标：针对系统中最小的可测试单元（通常是函数或类的方法）进行测试，验证其逻辑是否正确。
- 工具：Python 内置的 unittest 库，或者更现代、更简洁的第三方库如 pytest。
- 示例：我们在第五章的 test_scraper.py 中已经提供了爬虫模块的单元测试示例。

为了演示 pytest，我们可以为 data_processing.py 中的 clean_funding_amount 函数编写测试用例。
创建一个名为 test_data_processing.py 的文件：

```
# test_data_processing.py (假设使用 pytest)

import pytest
import pandas as pd
import numpy as np
# 导入要测试的函数
from data_processing import clean_funding_amount, remove_html_tags,
clean_whitespace, preprocess_text

# 假设 NLTK 资源已经下载
# from nltk.tokenize import word_tokenize
# from nltk.corpus import stopwords

# 测试 clean_funding_amount 函数
def test_clean_amount_eur_comma_decimal():
    assert clean_funding_amount("€1.234,56") == 1234.56
    assert clean_funding_amount("1.000.000,00 EUR") == 1000000.0

def test_clean_amount_usd_dot_decimal():
    assert clean_funding_amount("$1,234.56") == 1234.56
    assert clean_funding_amount("1,000,000.00 USD") == 1000000.0
```

```
def test_clean_amount_no_symbol_int():
    assert clean_funding_amount("50000") == 50000.0
    assert clean_funding_amount("1000000") == 1000000.0

def test_clean_amount_with_k_m_units():
    assert clean_funding_amount("50k EUR") == 50000.0
    assert clean_funding_amount("2.5M") == 2500000.0
    assert clean_funding_amount("100K") == 100000.0

def test_clean_amount_text_range():
    # clean_funding_amount 倾向于提取第一个数字，这是简化处理的特性
    assert clean_funding_amount("bis zu 50.000 EUR") == 50000.0
    assert clean_funding_amount("up to 1 million") == 1.0 # 1.0M -> 1000000.0
    assert clean_funding_amount("100.000-200.000") == 100000.0

def test_clean_amount_invalid_text():
    assert clean_funding_amount("keine Angabe") is None
    assert clean_funding_amount("Not specified") is None
    assert clean_funding_amount("abc") is None

def test_clean_amount_empty_string():
    assert clean_funding_amount("") is None

def test_clean_amount_nan_none():
    assert clean_funding_amount(np.nan) is None
    assert clean_funding_amount(None) is None

# 测试 remove_html_tags 函数
def test_remove_html_tags():
    assert remove_html_tags("<p>Hello <b>World</b></p>") == "Hello World"
    assert remove_html_tags("No tags here.") == "No tags here."
    assert remove_html_tags("<div>Line<br>Break</div>") == "LineBreak"
    assert remove_html_tags(None) == ""
    assert remove_html_tags(np.nan) == ""

# 测试 clean_whitespace 函数
def test_clean_whitespace():
    assert clean_whitespace("  Hello   World  ") == "Hello World"
    assert clean_whitespace("Line\nBreak\tTest") == "Line Break Test"
```

```
    assert clean_whitespace("Multiple    spaces") == "Multiple spaces"
    assert clean_whitespace(None) == ""
    assert clean_whitespace(np.nan) == ""

# 测试 preprocess_text 函数
# 注意: 运行此测试需要 NLTK 的 'punkt' 和 'stopwords' 资源已下载
# 如果未下载，此测试可能会失败或需要跳过
# @pytest.mark.skipif(not (nltk.data.find('tokenizers/punkt') and
nltk.data.find('corpora/stopwords')), reason="NLTK resources not downloaded")
def test_preprocess_text_german():
    # 模拟 NLTK 资源存在
    # from unittest.mock import patch, MagicMock
    # with patch('nltk.word_tokenize', return_value=['dies', 'ist', 'ein', 'test']) as
mock_tokenize, \
    #      patch('nltk.corpus.stopwords.words', return_value=['ist', 'ein']) as
mock_stopwords:
    #      assert preprocess_text("Dies ist ein Testtext.", language='german') == "dies
testtext"

    # 假设 NLTK 资源已下载，直接测试
    # 实际项目中，您需要在测试环境中确保 NLTK 资源可用，或者 mock NLTK
    assert preprocess_text("Dies ist ein Testtext für Innovationen.", language='german')
== "dies testtext innovationen"
    assert preprocess_text("Eine große Chance für kleine und mittlere Unternehmen.",
language='german') == "große chance kleine mittlere unternehmen"
    assert preprocess_text(None, language='german') == ""
    assert preprocess_text("2024 Test.", language='german') == "test" # 数字和标点被
移除

def test_preprocess_text_english():
    assert preprocess_text("This is a Test Text for Innovation.", language='english') ==
"test text innovation"
    assert preprocess_text("A great Opportunity for Small and Medium Enterprises.",
language='english') == "great opportunity small medium enterprises"
```

运行测试（在命令行）：

```
pip install pytest # 如果尚未安装 pytest
```

```
pytest test_data_processing.py
# 或只运行某个函数
# pytest test_data_processing.py::test_clean_amount_eur_comma_decimal
```

您将看到测试的运行结果，包括通过的测试数量和任何失败或错误的信息。

10.2.2 集成测试 (Integration Testing)

- 目标：测试多个模块组合在一起协同工作时是否正确，检查模块间的接口和数据传递。
- 示例：
  - 测试“爬虫模块获取原始 HTML -> 解析函数提取数据 -> 数据标准化函数处理”的整个流程，验证最终输出的结构化数据是否符合预期。
  - 测试“用户输入画像 -> 特征转换 -> 匹配引擎计算 -> 返回结果”的流程。

由于集成测试通常涉及多个组件，并且可能依赖于外部服务（如真实的网站），因此它们通常比单元测试更复杂，运行时间也更长。对于初学者，我们主要通过手动测试来验证集成，但了解其概念很重要。

10.2.3 端到端测试 (End-to-End Testing)/系统测试

- 目标：从用户的角度出发，模拟真实的使用场景，测试整个系统的功能是否符合需求。
- 示例：
  - 运行 Flask 应用，通过前端界面进行操作。
  - 尝试创建、编辑、删除搜索档案，观察页面和后端数据库的变化。
  - 点击“立即触发爬取与处理”按钮，观察爬虫是否运行，数据处理管道是否成功，以及资金项目数据是否更新到数据库和 pkl 文件。
  - 在匹配表单中输入各种需求，点击“匹配资金项目”，观察是否能正确显示结果，结果的相关性是否合理。

10.2.4 性能测试初步 (Performance Testing)

- 目标：评估系统在特定负载下的响应时间、吞吐量、资源消耗等。
- 示例：
  - 评估爬虫模块的平均抓取速度（例如，每分钟/每小时能抓取多少条项目信息）。
  - 评估匹配引擎对于单个用户画像的平均响应时间。
  - 使用 Python 的 cProfile 或 timeit 模块分析代码瓶颈。

本项目初期，性能测试可以相对简化，重点关注核心功能的正确性和基本效率。

10.2.5 错误处理与日志记录

良好的错误处理和日志记录是构建健壮应用程序的关键。它们可以帮助您在程序运行时捕

获和处理异常，并在出现问题时提供有用的诊断信息。

- 错误处理 (try...except)：
  在 Python 中，使用 try...except 语句来捕获和处理可能发生的异常。例如，在网络请求或数据库操作中，可能会发生连接错误、文件不存在等异常。
  在 app.py、main_scraper.py 和 database.py 中已经使用了 try...except 来处理网络请求和数据库操作可能出现的错误。请确保在关键操作中都包含了适当的错误处理逻辑。
- 日志记录 (logging)：
  logging 模块是 Python 标准库中用于记录日志的强大工具。通过日志，您可以在不中断程序运行的情况下，记录程序的运行状态、警告、错误等信息，这对于调试和监控生产环境中的应用程序至关重要。
  在 app.py、main_scraper.py、database.py、evaluation.py、matching_engine.py 以及 cli.py 等文件中，我们已经引入了 logging 模块并配置了基本的日志输出。
  日志级别：
  - DEBUG：详细的调试信息，通常只在开发阶段使用。
  - INFO：确认程序按预期运行的信息。
  - WARNING：表示发生了意外，但程序仍然可以继续运行。
  - ERROR：由于更严重的问题，程序无法执行某些功能。
  - CRITICAL：严重错误，程序可能无法继续运行。

配置日志：在每个文件的开头，我们通常会像这样配置日志（或使用一个集中的日志配置文件）：import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
# 或者更精细的配置，输出到文件和控制台
# logging.basicConfig(
#   level=logging.INFO,
#   format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
#   handlers=[
#   logging.FileHandler("app.log", encoding='utf-8'), # 输出到文件
#   logging.StreamHandler() # 输出到控制台
#   ]
#)
logger = logging.getLogger(__name__) # 获取当前模块的 logger
在代码中，您可以使用 logging.info(), logging.warning(), logging.error() 等方法来记录信息。对于捕获的异常，使用 logging.exception(e) 可以自动记录完整的堆栈信息，这对于调试至关重要。

通过本章的学习，您应该已经掌握了测试和调试的基本方法，以及错误处理和日志记录的重要性。这些技能将帮助您构建更健壮、更可靠的应用程序。

第十一章：项目完善与未来展望

恭喜您！到目前为止，您已经成功地从零开始构建了一个资金项目搜索工具，涵盖了从数据爬取、存储、后端 API 到前端界面的所有核心功能。本章将讨论如何进一步完善您的项目，使其更加健壮、高效和用户友好，并探讨一些未来可能的扩展方向。

11.1 项目核心成果回顾与价值分析

11.1.1 已完成核心功能总结

通过本教程，您已经实现了以下核心功能：

- 自动化数据获取：成功构建了针对多个德国和欧洲资助信息源的网络爬虫，能够定期或按需抓取最新的项目数据。
- 数据处理与标准化：建立了一套数据清洗、转换和标准化的流程，将来自不同源头的异构数据整合为统一格式，并提取了初步的文本和结构化特征。
- 初步智能匹配能力：实现了基于规则与特征相似度的匹配算法，能够根据用户画像从项目库中筛选和推荐相关的资助机会。
- 可复现的评估流程：搭建了科学的模型/系统评估流水线，定义了明确的评估指标，并建立了性能基线，为后续优化提供了依据。
- 环境封装与初步部署能力：通过 Docker 容器化，保证了项目的环境一致性和可移植性，并探索了 MinIO 等存储方案。
- 完整的 **Web** 应用：实现了 Flask 后端 API 和基于原生 HTML/CSS/JavaScript 的前端界面，使用户能够通过友好的界面进行交互。

11.1.2 项目成果的潜在价值

本项目的成功实施将带来多方面的价值：

- 提高信息获取效率：为企业、研究机构和咨询顾问节省了大量搜寻和筛选资助信息的时间。
- 提升匹配精准度：通过 AI 技术，有望比人工筛选更全面、更精准地匹配用户需求与资助条件。
- 赋能决策：提供结构化、可比较的资助信息，帮助用户做出更明智的申请决策。
- 知识沉淀与教育价值：项目开发过程和最终的“教科书”文档，为初学者提供了宝贵的实战学习材料。

11.2 个人学习与技能提升总结

通过完成这个项目，开发者（尤其是初学者）可以在以下方面获得显著的技能提升和经验积累：

- **Python** 编程能力：熟练运用 Python 进行复杂应用开发，包括模块化设计、面向对象编程、文件操作、异常处理等。
- 网络爬虫技术：掌握 Requests, BeautifulSoup, lxml 等库的使用，理解 HTTP 协议，具

备分析网站结构和应对基本反爬策略的能力。
- 数据处理与分析：精通 Pandas 进行数据清洗、转换、聚合、特征提取等操作。
- 自然语言处理 **(NLP)** 基础：了解文本预处理流程，掌握 TF-IDF 等文本表示方法，并能应用于文本相似度计算。
- 机器学习基础与评估：理解机器学习的基本流程（数据准备、模型训练、评估），掌握常用分类模型的原理和 Scikit-learn 实现，能够科学地评估模型性能。
- **Web** 开发基础：了解 Flask 框架和 RESTful API 的概念，掌握 HTML/CSS/JavaScript 构建前端界面的基础。
- 软件工程实践：体验完整的项目生命周期，学习版本控制 (Git)、环境管理（虚拟环境、Docker）、模块化开发、编写测试用例、撰写技术文档等。
- 问题解决能力：在开发过程中会遇到各种预料之外的技术难题和数据问题，通过查阅资料、调试代码、寻求帮助来解决这些问题，是提升最快的方式。

11.3 潜在合作探讨

用户提及了 granter.ai 这家类似公司，并建议探讨合作可能性。这是一个很好的拓展思路。

11.3.1 分析 granter.ai（此步骤需要实际访问其网站进行调研）

- 业务模式：他们提供什么服务？是 SaaS 平台、数据服务还是咨询？
- 技术特点：他们强调的 AI 能力是什么？使用了哪些技术？
- 目标客户：他们主要服务于哪些类型的客户（初创、中小企业、大型企业、研究机构）？
- 覆盖范围：他们的数据源主要覆盖哪些国家和地区？哪些类型的资助项目？
- 定价策略：他们的服务是如何收费的？

11.3.2 思考合作切入点

- 数据补充/验证：如果本项目的数据源与 granter.ai 存在差异或互补（例如，本项目可能更侧重某些特定类型的德国本土资助，而他们可能更广泛），可以探讨数据共享或交叉验证的可能性。
- 技术方案交流/借鉴：本项目的 AI 匹配思路、特征工程方法或评估体系，如果具有独到之处，可以作为技术交流的内容。反之，也可以从他们的公开信息中学习。
- 市场/区域互补：如果 granter.ai 主要关注北美或全球市场，而本项目深耕德国及欧洲特定区域，可能存在市场合作或渠道共享的机会。
- 功能模块合作：例如，如果 granter.ai 在用户画像分析方面有强项，而本项目在特定数据源的深度挖掘上有优势，可以探讨模块化合作。
- 开源社区贡献：如果本项目的部分组件具有通用性，可以考虑开源，吸引包括 granter.ai 在内的社区关注和贡献。

11.3.3 准备初步沟通材料

- 一份简明扼要的项目介绍 (Project Overview / One-pager)。

- 突出本项目的核心技术、数据覆盖特点、已实现的成果（如评估报告中的关键指标）。
- 清晰阐述潜在的合作价值点和合作模式设想。
- 在接触前，确保自身项目达到一定的成熟度和专业度。

11.4 项目局限性与未来可优化方向：持续迭代的蓝图

任何项目都不可能一蹴而就，本项目在初期版本完成后，必然存在一些局限性，同时也为未来的持续改进和功能增强留下了广阔的空间。

11.4.1 AI 模型深化与智能化提升

- 高级 **NLP** 技术：
  - 使用预训练语言模型（如 BERT, Sentence-BERT, RoBERTa 等，可通过 Hugging Face Transformers 库方便调用）生成文本的上下文感知向量表示，以替代 TF-IDF，有望显著提升语义匹配的精度。
  - 针对德语等特定语言，选择相应的预训练模型。
- 更复杂的匹配/推荐算法：
  - 如果积累了用户行为数据（如用户对推荐项目的点击、收藏、申请等），可以尝试协同过滤算法。
  - 结合内容推荐与知识图谱：构建资助项目、机构、领域、技术、申请条件等实体及其关系的知识图谱，利用图嵌入和图神经网络进行更深层次的关联分析和智能推荐。
  - 引入 Learning to Rank (LTR) 模型，直接优化搜索结果的排序。
- 用户画像的动态学习与完善：根据用户行为和反馈，动态调整和丰富用户画像。

11.4.2 爬虫技术增强与数据源扩展

- 处理复杂反爬机制：
  - 针对 JavaScript 动态渲染的网站，引入 Selenium、Playwright 或 Puppeteer 等浏览器自动化工具（但需权衡性能开销）。
  - 研究更高级的验证码识别技术（如基于深度学习的 OCR，或对接打码平台，需考虑成本和合规性）。
  - 构建更智能、更稳定的 IP 代理池管理与调度系统。
- 增量爬取与实时更新：设计机制监控数据源的变化，实现对新增或更新的资助项目进行增量爬取，保证项目库的时效性。
- 持续接入更多数据源：不断扩展爬取的网站范围，覆盖更多国家、地区和类型的资助信息发布平台（如其他欧盟成员国的国家级资助数据库、行业协会网站、孵化器/加速器项目等）。

11.4.3 用户体验提升 (UX/UI)

- 开发图形用户界面 **(GUI)**：使用 Web 框架（如 Flask, Django, 或更快速的 Streamlit,

Dash）构建用户友好的查询界面和结果展示界面。提供更丰富的筛选条件、排序方式、可视化图表等。

- 个性化推荐与订阅服务：允许用户保存搜索偏好，订阅特定领域或类型的资助项目更新通知（如邮件提醒）。根据用户的历史行为和偏好，进行更个性化的项目推送。
- 智能问答与辅助申请：未来可以集成基于 NLP 的问答系统，解答用户关于特定资助项目的疑问。甚至可以辅助用户准备申请材料的初稿（例如，根据项目要求和用户画像生成部分文本）。

11.4.4 数据质量与治理

- 建立更完善的数据校验规则和自动化清洗流程，持续提升项目库中数据的准确性、完整性和一致性。
- 引入数据版本控制和元数据管理。

11.4.5 推荐进阶学习资源：迈向更高峰

为了实现上述优化方向，开发者需要不断学习和提升技能。以下是一些推荐的进阶学习资源：

- 高级网络爬虫：
  - 《精通 Python 爬虫框架 Scrapy》（书籍）
  - Selenium 官方文档
  - Playwright Python 官方文档
- 自然语言处理 **(NLP)**：
  - 斯坦福大学 CS224n 课程：Natural Language Processing with Deep Learning（经典课程）
  - 《Speech and Language Processing》（Daniel Jurafsky & James H. Martin 著，NLP 领域权威教材）
  - Hugging Face Transformers 库：官方文档（提供了大量预训练模型和便捷的调用接口）
- 机器学习**/**深度学习：
  - 《深度学习》（Ian Goodfellow, Yoshua Bengio, Aaron Courville 著，俗称“花书”）
  - Fast.ai 课程：Practical Deep Learning for Coders（注重实践）
  - PyTorch 官方教程/TensorFlow 官方教程
- **Web** 开发（用于构建 **GUI**）：
  - Flask 官方文档（轻量级 Web 框架）
  - Django 官方文档（全功能 Web 框架）
  - Streamlit 官方文档（快速构建数据应用的利器）
- 数据库与数据工程：
  - 所选数据库（PostgreSQL, MongoDB 等）的官方文档。
  - 学习数据仓库概念、ETL (Extract, Transform, Load) 流程设计、数据管道构建（如 Apache Airflow）等知识。

项目开发是一个持续学习和迭代的过程。通过不断总结经验、拥抱新技术、关注用户需求，这个 AI 资助项目发现系统将能发挥越来越大的价值。

关键要点总结

- 模块集成：设计清晰的数据流和模块调用关系，编写主程序脚本统一调度。
- 全面测试：实施单元测试、集成测试、端到端测试和初步性能测试，确保系统质量。
- 项目总结：回顾已完成功能和价值，总结个人技能提升。
- 合作思考：分析潜在合作对象（如 granter.ai），寻找合作切入点。
- 未来展望：规划 AI 模型深化、爬虫技术增强、用户体验提升、数据源扩展等优化方向，并列出相关进阶学习资源。