资金项目搜索工具开发教程

第一章：项目概述与需求分析

1.1 项目背景

在德国和欧洲，资金项目数量庞大且不断变化，这给咨询团队带来了巨大的挑战。即使将范围缩小到特定行业，仍然存在大量相关的资金机会，但分析和评估这些机会非常耗时且资源密集。为了解决这一痛点，本项目旨在开发一个AI辅助的搜索工具，以自动化识别新的合适资金项目，并以结构化的方式呈现，从而显著减少研究工作量，同时显著提高咨询的时效性。

1.2 项目目标

本项目的核心目标是构建一个智能化的资金项目搜索工具，能够：

1. **自动化识别**：利用AI技术自动从各种数据源中识别出符合特定条件的新资金项目。
2. **结构化呈现**：将识别出的资金项目信息进行清洗、整理和结构化，以便用户能够快速理解和利用。
3. **提高效率**：大幅减少人工研究的时间和精力投入，提高咨询团队的工作效率。
4. **增强时效性**：通过定期（例如每周）的自动化搜索，确保提供的信息是最新的，从而提高咨询的时效性和准确性。

1.3 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。

1.4 模型评估管线（补充需求）

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：

1. **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。
2. **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。

1.5 数据来源与爬取考虑

本项目将从以下网站获取相关数据：

- `https://www.ptj.de/`
- `https://www.eu-startups.com/`
- `https://www.foerderdatenbank.de/FDB/DE/Home/home.html`
- `https://www.deutsche-digitale-bibliothek.de/`

在进行网络爬取时，我们将特别关注以下几点：

1. **`robots.txt` 遵守**：在爬取任何网站之前，我们都将检查其 `robots.txt` 文件（例如 `https://www.foerderdatenbank.de/robots.txt`），并严格遵守其中规定的爬取规则，以避免对网站造成不必要的负担或违反其使用政策。`robots.txt` 文件是一个标准，用于告知网络爬虫哪些页面可以访问，哪些页面不应访问。它通常包含 `User-agent` 和 `Disallow` 等指令，指导爬虫的行为。
2. **爬行限制与反爬机制**：某些网站可能存在更复杂的爬行限制或反爬机制（如动态加载内容、IP限制、验证码等）。对于初学者教程，我们将主要关注静态页面的爬取，并介绍如何处理简单的动态内容。对于复杂的反爬机制，我们将提供概念性介绍，并建议在实际项目中寻求更专业的解决方案。
3. **数据结构多样性**：不同网站的数据格式和结构可能差异很大。我们将设计灵活的爬虫模块，能够适应不同网站的数据特点，并将其统一结构化。

1.6 潜在合作机会

用户提到 `https://granter.ai/` 是一家提供类似服务的公司。虽然本次教程的重点是项目实现，但了解行业内的类似解决方案有助于我们拓宽视野，并在未来考虑潜在的合作或学习机会。在实际商业环境中，与类似公司进行交流和合作，可能会带来互利共赢的局面。

第二章：系统架构设计

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。

2.1 整体架构概览

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2-1: 系统架构概览

2.2 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`)**：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。
- **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` 解析HTML，根据预定义的规则（如CSS选择器）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。

2.3 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。

2.4 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。

2.4.1 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。

2.4.2 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。

第三章：开发环境搭建

一个良好的开发环境是项目顺利进行的基石。本章将详细指导您如何搭建Python开发环境，安装必要的库，并推荐使用集成开发环境（IDE），帮助您高效地编写和管理代码。

3.1 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：

3.1.1 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。

3.1.2 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： `bash sudo apt update sudo apt install python3 python3-pip`
    - 在基于RPM的系统（如Fedora, CentOS）上： `bash sudo dnf install python3 python3-pip`

3.1.3 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
```

3.2 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。

3.2.1 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。

3.2.2 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： `bash 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。 `bash 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。

3.2.3 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**： `bash .\venv\Scripts\activate` 或者，如果您使用的是PowerShell： `powershell .\venv\Scripts\Activate.ps1`
- **macOS/Linux**： `bash 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。

3.2.4 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
```

3.3 安装必要的Python库

激活虚拟环境后，我们可以使用 `pip` 来安装本项目所需的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。
- **`pandas`**：用于数据结构化、清洗和分析。
- **`hashlib`**：Python内置库，用于生成哈希值，实现数据去重（无需单独安装）。

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 pandas
```

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
```

3.4 集成开发环境（IDE）推荐与配置

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)**。

3.4.1 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。

3.4.2 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”。
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： `python 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。

第四章：数据爬取模块开发

数据爬取是本项目获取资金项目信息的核心环节。本章将深入讲解如何构建高效、稳定的网络爬虫，从目标网站提取所需数据。我们将涵盖HTTP请求、HTML解析、数据结构化以及数据去重等关键技术点，并特别关注 `robots.txt` 文件的处理。

4.1 网络爬虫基础

网络爬虫（Web Crawler），也称为网络蜘蛛（Web Spider），是一种自动化程序，它能够模拟人类浏览网页的行为，自动地在互联网上抓取信息。爬虫的工作流程通常包括：

1. **发送请求**：向目标网站的服务器发送HTTP请求，获取网页内容。
2. **解析响应**：对服务器返回的HTML、XML或JSON等内容进行解析，提取出有用的数据。
3. **数据存储**：将提取到的数据保存到本地文件或数据库中。
4. **循环抓取**：根据网页中的链接，继续访问其他页面，重复上述过程。

4.1.1 HTTP请求与响应

当您在浏览器中输入一个网址并按下回车键时，您的浏览器会向该网址对应的服务器发送一个HTTP请求。服务器接收到请求后，会返回一个HTTP响应，其中包含了网页的HTML内容、图片、CSS、JavaScript等资源。网络爬虫正是通过编程的方式模拟这一过程。

- **请求方法**：最常用的HTTP请求方法是 `GET`（用于获取资源）和 `POST`（用于提交数据）。在爬虫中，我们主要使用 `GET` 请求来获取网页内容。
- **状态码**：服务器响应会包含一个状态码，用于表示请求的处理结果。常见的状态码有：
  - `200 OK`：请求成功，服务器已成功处理请求。
  - `301 Moved Permanently` / `302 Found`：重定向，资源已永久或临时移动到新的URL。
  - `403 Forbidden`：服务器拒绝访问，通常是由于权限不足或被网站识别为爬虫而拒绝。
  - `404 Not Found`：请求的资源不存在。
  - `500 Internal Server Error`：服务器内部错误。

4.1.2 HTML结构与解析

网页内容通常以HTML（HyperText Markup Language）格式呈现。HTML文档由一系列标签（Tag）组成，这些标签定义了网页的结构和内容。例如，`<p>` 标签表示段落，`<a>` 标签表示链接，`<div>` 标签表示一个内容区块。

为了从HTML中提取数据，我们需要一个HTML解析器。解析器能够将HTML文本转换为一个可编程操作的树状结构（DOM树），然后我们就可以通过各种选择器（如标签名、ID、类名、属性等）来定位和提取所需的元素。

- **CSS选择器**：与CSS样式表中使用的选择器类似，可以非常方便地选择HTML元素。例如，`div.content` 选择所有class为 `content` 的 `div` 元素，`#title` 选择id为 `title` 的元素。
- **XPath**：一种XML路径语言，也可以用于HTML文档。XPath提供了更强大的路径表达式，可以更灵活地定位元素，尤其是在处理复杂或嵌套结构时。

4.2 `robots.txt` 文件与爬虫礼仪

在开始爬取任何网站之前，作为负责任的爬虫开发者，我们必须检查并遵守目标网站的 `robots.txt` 文件。`robots.txt` 是一个文本文件，位于网站的根目录下（例如 `https://www.example.com/robots.txt`），它告诉搜索引擎爬虫和其他自动化程序哪些页面可以访问，哪些页面不应访问。

4.2.1 为什么需要遵守 `robots.txt`？

1. **避免法律风险**：某些网站可能明确禁止爬取其内容，违反规定可能导致法律纠纷。
2. **减轻服务器负担**：频繁或无限制的爬取可能会给网站服务器带来巨大压力，导致网站响应变慢甚至崩溃。
3. **维护良好关系**：遵守 `robots.txt` 是网络爬虫的基本礼仪，有助于维护与网站所有者的良好关系。

4.2.2 `robots.txt` 文件的结构

`robots.txt` 文件通常包含 `User-agent` 和 `Disallow` 等指令：

- **`User-agent`**：指定该规则适用于哪种爬虫。`*` 表示所有爬虫。
- **`Disallow`**：指定不允许爬虫访问的路径。例如，`Disallow: /admin/` 表示不允许访问 `/admin/` 目录及其子目录。
- **`Allow`**：指定允许爬虫访问的路径，通常用于覆盖 `Disallow` 规则。
- **`Sitemap`**：指定网站的XML站点地图的URL，有助于爬虫发现网站的所有页面。

示例 `robots.txt` 文件：

```
 User-agent: *
Disallow: /admin/
Disallow: /private/
Allow: /public/images/

User-agent: Googlebot
Disallow: /temp/

Sitemap: https://www.example.com/sitemap.xml
```

4.2.3 如何检查 `robots.txt`？

在编写爬虫之前，您应该手动访问目标网站的 `robots.txt` 文件。例如，对于 `https://www.foerderdatenbank.de/`，您应该访问 `https://www.foerderdatenbank.de/robots.txt`。阅读其内容，并确保您的爬虫行为符合其规定。

重要提示： `robots.txt` 只是一个“君子协定”，它并不能强制阻止爬虫访问。但作为负责任的开发者，我们应该自觉遵守。对于本教程，我们将假定您已检查并理解了目标网站的 `robots.txt` 文件，并会编写符合其规定的爬虫。

4.3 使用 `requests` 和 `BeautifulSoup4` 进行数据爬取

现在，我们将通过实际代码示例，演示如何使用 `requests` 库获取网页内容，并使用 `BeautifulSoup4` 解析HTML。

4.3.1 安装库

确保您已激活虚拟环境，并安装了 `requests` 和 `beautifulsoup4`：

```
pip install requests beautifulsoup4
```

4.3.2 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。

创建一个名为 `scraper_ptj.py` 的文件，并添加以下代码：

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
import hashlib

def scrape_ptj():
    base_url = "https://www.ptj.de/"
    search_url = "https://www.ptj.de/suche-foerderinitiativen"

    # 模拟浏览器请求头，避免被识别为爬虫
    headers = {
        'User-Agent':
'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
    }

    print(f"正在爬取: {search_url}")
    try:
        response = requests.get(search_url, headers=headers,
timeout=10)
        response.raise_for_status() # 检查HTTP请求是否成功
    except requests.exceptions.RequestException as e:
```

```
        print(f"请求失败: {e}")
        return pd.DataFrame()

    soup = BeautifulSoup(response.text, 'html.parser')

    # 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)
```

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。
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(response.text, 'html.parser')`：** 使用 `BeautifulSoup` 解析获取到的HTML文本。`'html.parser'` 是Python内置的HTML解析器。
6. **`soup.find()` 和 `soup.find_all()`：** 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。

4.3.3 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`。

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

```
import requests
from bs4 import BeautifulSoup
```

```
import pandas as pd
import hashlib

def scrape_eustartups():
    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

    headers = {
        'User-Agent':
'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
    }

    print(f"正在爬取: {investor_url}")
    try:
        response = requests.get(investor_url, headers=headers,
timeout=10)
        response.raise_for_status()
    except requests.exceptions.RequestException as e:
        print(f"请求失败: {e}")
        return pd.DataFrame()

    soup = BeautifulSoup(response.text, 'html.parser')

    # 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)
```

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。

4.3.4 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，`https://www.foerderdatenbank.de/SiteGlobals/FDB/Forms/Suche/Foederprogrammsuche_Formular.html?resourceId=0065e6ec-5c0a-4678-b503-b7e7ec435dfd&input_=23adddb0-dcf7-4e32-96f5-93aec5db2716&pageLocale=de&filterCategories=FundingProgram&te` 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，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`。

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

```
import requests
from bs4 import BeautifulSoup
```

```
import pandas as pd
import hashlib

def scrape_foerderdatenbank():
    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':
'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
    }

    print(f"正在爬取: {search_result_url}")
    try:
        response = requests.get(search_result_url,
headers=headers, timeout=10)
        response.raise_for_status()
    except requests.exceptions.RequestException as e:
        print(f"请求失败: {e}")
        return pd.DataFrame()

    soup = BeautifulSoup(response.text, 'html.parser')

    # 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)
```

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： `python 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。

4.3.5 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。我们将以其搜索结果页为例进行爬取。

检查 `https://www.deutsche-digitale-bibliothek.de/robots.txt`。

创建一个名为 `scraper_ddb.py` 的文件，并添加以下代码：

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
```

```
import hashlib

def scrape_ddb():
    base_url = "https://www.deutsche-digitale-bibliothek.de/"
    # 这是一个示例搜索结果页URL，您可能需要根据实际搜索情况调整
    search_url = "https://www.deutsche-digitale-bibliothek.de/
search?query=funding"

    headers = {
        'User-Agent':
'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
    }

    print(f"正在爬取: {search_url}")
    try:
        response = requests.get(search_url, headers=headers,
timeout=10)
        response.raise_for_status()
    except requests.exceptions.RequestException as e:
        print(f"请求失败: {e}")
        return pd.DataFrame()

    soup = BeautifulSoup(response.text, 'html.parser')

    # 假设搜索结果在一个 class 为 'search-results' 的 div 中，每个结果
是一个 class 为 'result-item' 的 article
    # 同样，这只是一个示例，您需要检查实际的HTML结构
    results_container = soup.find('div', class_='search-
results')

    if not results_container:
        print("未找到搜索结果容器，请检查选择器或网页结构。")
        return pd.DataFrame()

    items = results_container.find_all('article',
class_='result-item')

    if not items:
        print("未找到任何搜索结果，请检查选择器或网页结构。")
        return pd.DataFrame()

    data = []
    for item in 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'

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

        data.append({
            'id': unique_id,
            'title': title,
            'description': description,
            'link': link,
            'source': 'deutsche-digitale-bibliothek.de'
        })

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

if __name__ == "__main__":
    ddb_df = scrape_ddb()
    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)
```

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。

4.4 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：

```python
import pandas as pd
from scraper_ptj import scrape_ptj
from scraper_eustartups import scrape_eustartups
from scraper_foerderdatenbank import scrape_foerderdatenbank
from scraper_ddb import scrape_ddb

def run_all_scrapers():
    all_data = []

    print("\n--- 正在运行 ptj.de 爬虫 ---")
    ptj_df = scrape_ptj()
    if not ptj_df.empty:
        all_data.append(ptj_df)

    print("\n--- 正在运行 eu-startups.com 爬虫 ---")
    eustartups_df = scrape_eustartups()
    if not eustartups_df.empty:
        all_data.append(eustartups_df)

    print("\n--- 正在运行 foerderdatenbank.de 爬虫 ---")
    foerderdatenbank_df = scrape_foerderdatenbank()
    if not foerderdatenbank_df.empty:
        all_data.append(foerderdatenbank_df)

    print("\n--- 正在运行 deutsche-digitale-bibliothek.de 爬虫
---")
    ddb_df = scrape_ddb()
    if not ddb_df.empty:
        all_data.append(ddb_df)

    if all_data:
        # 合并所有爬取到的数据
        final_df = pd.concat(all_data, 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
        final_df.drop_duplicates(subset=['id'], inplace=True)
        print(f"\n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:
        print("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')
        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` 列表中。
3. **`pd.concat()`**：将 `all_data` 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。

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。

第五章：数据存储与管理

在上一章中，我们学习了如何从不同的网站爬取数据。本章将重点介绍如何将这些爬取到的数据进行持久化存储，并实现有效的数据管理，包括数据库设计、数据插入、查询、更新和去重。我们将使用轻量级的SQLite数据库，并通过Python的 `sqlite3` 模块进行操作。

5.1 数据库选择：SQLite

对于本项目，我们选择SQLite作为数据库解决方案。SQLite是一个自给自足的、无服务器的、零配置的事务性SQL数据库引擎。它不需要独立的服务器进程，数据直接存储在单个文件中。这使得它非常适合：

- **初学者**：易于设置和使用，无需复杂的数据库管理知识。
- **轻量级应用**：适用于数据量不大、并发访问不高的场景。
- **嵌入式应用**：可以直接集成到应用程序中。

5.2 数据库设计

为了存储资金项目信息和用户定制的搜索档案，我们需要设计相应的数据库表。我们将创建两个主要的表：

1. **`funds` 表**：用于存储爬取到的资金项目信息。

2. **`search_profiles` 表**：用于存储用户创建的搜索档案。

5.2.1 `funds` 表结构

字段名	数据类型	约束	描述
`id`	TEXT	PRIMARY KEY	资金项目的唯一标识（MD5哈希值）
`title`	TEXT	NOT NULL	资金项目标题
`description`	TEXT		资金项目描述
`link`	TEXT		资金项目原始链接
`source`	TEXT		数据来源网站
`crawled_at`	TEXT	DEFAULT CURRENT_TIMESTAMP	爬取时间（ISO格式字符串）

说明：

- `id` 字段将存储我们之前使用 `hashlib` 生成的MD5哈希值，作为主键，确保每条资金项目记录的唯一性。
- `crawled_at` 字段将自动记录数据插入的时间，方便追踪数据的时效性。

5.2.2 `search_profiles` 表结构

| 字段名 | 数据类型 | 约束 | 描述 | | :---------- | :------- | :---------- | :---------------------------------- |\n| `id` | INTEGER | PRIMARY KEY AUTOINCREMENT | 搜索档案的唯一标识（自增） | | `name` | TEXT | NOT NULL UNIQUE | 搜索档案名称（唯一） | | `industry` | TEXT | | 行业关键词 | | `company_age` | TEXT | | 公司年龄范围 | | `turnover` | TEXT | | 营业额范围 | | `employees` | TEXT | | 员工数量范围 | | `created_at` | TEXT | DEFAULT CURRENT_TIMESTAMP | 创建时间（ISO格式字符串） |

说明：

- `id` 字段是自增主键。
- `name` 字段是搜索档案的名称，必须唯一。
- 其他字段用于存储用户定义的筛选条件。

5.3 数据库操作：Python `sqlite3` 模块

Python内置的 `sqlite3` 模块提供了与SQLite数据库交互的接口。我们将使用它来创建数据库、表，并执行数据的增删改查操作。

5.3.1 数据库连接与游标

在进行任何数据库操作之前，我们需要建立与数据库的连接，并创建一个游标对象。游标用于执行SQL命令。

```
import sqlite3

def get_db_connection():
    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
```

说明：

- `sqlite3.connect("funds.db")`：连接到名为 `funds.db` 的数据库文件。如果该文件不存在，SQLite会自动创建一个新的数据库文件。
- `conn.row_factory = sqlite3.Row`：这是一个非常实用的设置。默认情况下，`sqlite3` 查询结果的每一行都是一个元组。设置 `row_factory` 为 `sqlite3.Row` 后，每一行将表现得像一个字典，您可以通过列名（例如 `row["title"]`）来访问数据，而不是通过索引（例如 `row[1]`），这使得代码更具可读性。

5.3.2 创建数据库和表

我们将编写一个函数来初始化数据库，即创建 `funds` 和 `search_profiles` 表。

```
def init_db():
    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。")

if __name__ == "__main__":
    init_db()
```

说明：

- `CREATE TABLE IF NOT EXISTS`：如果表不存在则创建，避免重复创建表时报错。
- `conn.commit()`：提交当前事务。在执行 `INSERT`，`UPDATE`，`DELETE` 或 `CREATE TABLE` 等修改数据库的操作后，必须调用 `commit()` 才能使更改永久生效。
- `conn.close()`：关闭数据库连接。这是一个好习惯，可以释放资源。

5.4 资金项目数据管理

我们将实现向 `funds` 表插入数据和查询数据的功能。

5.4.1 插入资金项目数据（带去重）

为了避免重复插入相同的资金项目，我们将在插入前检查 `id` 是否已存在。如果存在，则更新现有记录；否则，插入新记录。

```
import sqlite3
import pandas as pd
import hashlib
from datetime import datetime

def get_db_connection():
    conn = sqlite3.connect("funds.db")
    conn.row_factory = sqlite3.Row
    return conn
```

```python
def insert_or_update_fund(fund_data):
    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):
    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。")

# 示例用法 (假设您已经运行了 main_scraper.py 并生成了
combined_funds_data.csv)
if __name__ == "__main__":
    init_db() # 确保数据库和表已初始化

    # 模拟从爬虫模块获取数据
    # 在实际应用中，这里会调用 run_all_scrapers() 或从CSV文件读取
    try:
        combined_df = pd.read_csv("combined_funds_data.csv")
        save_funds_data(combined_df)
    except FileNotFoundError:
        print("combined_funds_data.csv 文件未找到，请先运行
main_scraper.py。")

    # 示例：查询所有资金项目
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM funds")
    funds = cursor.fetchall()
    conn.close()

    print("\n数据库中的资金项目：")
    for fund in funds:
        print(f"ID: {fund["id"][:8]}..., Title:
{fund["title"]}, Source: {fund["source"]}")
```

说明:

- `insert_or_update_fund(fund_data)` 函数:
  - 首先尝试根据 `id` 查询记录是否存在。
  - 如果存在，则执行 `UPDATE` 语句更新所有字段，并更新 `crawled_at` 时间戳。
  - 如果不存在，则执行 `INSERT` 语句插入新记录。
- `save_funds_data(df)` 函数：接收一个Pandas DataFrame，遍历每一行，并调用 `insert_or_update_fund` 函数进行保存。
- `datetime.now().isoformat()`：将当前时间转换为ISO格式的字符串，方便存储到TEXT类型的数据库字段中。

5.4.2 查询资金项目数据

除了上述示例中的查询所有资金项目，我们还可以根据条件进行查询。例如，根据来源或关键词进行查询。

```
def get_funds(source=None, keyword=None):
    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}%")

    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

# 示例用法
if __name__ == "__main__":
    # ... (前面的 init_db 和 save_funds_data 保持不变)

    print("\n查询来自 ptj.de 的资金项目：")
    ptj_funds = get_funds(source="ptj.de")
    for fund in ptj_funds:
        print(f"Title: {fund["title"]}, Link: {fund["link"]}")

    print("\n查询包含 'digital' 关键词的资金项目：")
    digital_funds = get_funds(keyword="digital")
    for fund in digital_funds:
        print(f"Title: {fund["title"]}, Source:
{fund["source"]}")
```

说明:

- `get_funds()` 函数：接受 `source` 和 `keyword` 参数，用于构建动态查询。
- `WHERE 1=1`：这是一个常用的技巧，用于方便地拼接 `AND` 条件，而无需担心第一个条件前是否需要 `AND`。
- `LIKE ?` 和 `%{keyword}%`：用于模糊匹配，`%` 是通配符。
- `tuple(params)`：`execute` 方法的第二个参数必须是元组或列表，用于传递查询参数，这可以有效防止SQL注入攻击。
- `[dict(fund) for fund in funds]`：将 `sqlite3.Row` 对象列表转换为普通的字典列表，方便后续处理。

5.5 搜索档案数据管理

我们将实现对 `search_profiles` 表的增删改查功能。

5.5.1 插入搜索档案

```
def create_search_profile(name, industry=None, company_age=None,
turnover=None, employees=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()

# 示例用法
if __name__ == "__main__":
    # ... (前面的 init_db 保持不变)

    create_search_profile("初创企业资金", industry="科技",
company_age="<3年", turnover="<1M", employees="<10")
    create_search_profile("中小企业发展", industry="制造",
company_age="3-10年", turnover="1M-10M", employees="10-50")
    create_search_profile("初创企业资金", industry="科技") # 尝试创建
重复名称的档案
```

说明:

- `try...except sqlite3.IntegrityError`：捕获 `UNIQUE` 约束冲突，即尝试插入已存在的档案名称时会报错。
- `cursor.lastrowid`：返回最后一次插入操作生成的行ID，对于自增主键非常有用。

5.5.2 查询搜索档案

```
def get_search_profiles(profile_id=None):
    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]

# 示例用法
if __name__ == "__main__":
    # ... (前面的 init_db 和 create_search_profile 保持不变)

    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"]}")

    print("\n查询ID为1的搜索档案：")
    profile_1 = get_search_profiles(profile_id=1)
    if profile_1:
        print(profile_1[0])
```

5.5.3 更新搜索档案

```
def update_search_profile(profile_id, name=None, industry=None,
company_age=None, turnover=None, employees=None):
    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()

# 示例用法
if __name__ == "__main__":
    # ... (前面的代码保持不变)

    update_search_profile(1, name="科技初创企业资金", industry="人工
智能")
    update_search_profile(99, name="不存在的档案") # 尝试更新不存在的
档案
```

说明：

- `update_search_profile()` 函数：根据提供的参数动态构建 `UPDATE` 语句。
- `cursor.rowcount`：返回受上一个 `execute()` 调用影响的行数，可以用来判断更新是否成功。

5.5.4 删除搜索档案

```
def delete_search_profile(profile_id):
    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()

# 示例用法
if __name__ == "__main__":
    # ... (前面的代码保持不变)

    delete_search_profile(1)
    delete_search_profile(99) # 尝试删除不存在的档案
```

5.6 整合数据存储模块

为了方便管理，我们可以将所有数据库操作相关的函数封装到一个单独的文件中，例如 `database.py`。

`database.py` 文件内容：

```
 import sqlite3
from datetime import datetime
import pandas as pd

def get_db_connection():
    conn = sqlite3.connect("funds.db")
    conn.row_factory = sqlite3.Row
    return conn

def init_db():
    conn = get_db_connection()
    cursor = conn.cursor()

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

```
    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()
    conn.close()
    print("数据库初始化完成，表已创建或已存在。")

def insert_or_update_fund(fund_data):
    conn = get_db_connection()
    cursor = conn.cursor()

    cursor.execute("SELECT id FROM funds WHERE id = ?",
(fund_data["id"],))
    existing_fund = cursor.fetchone()

    if existing_fund:
        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(),
            fund_data["id"]
        ))
    else:
        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()
        ))

    conn.commit()
    conn.close()

def save_funds_data(df):
    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):
    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}%")

    cursor.execute(query, tuple(params))
    funds = cursor.fetchall()
    conn.close()

    return [dict(fund) for fund in funds]

def create_search_profile(name, industry=None, company_age=None,
turnover=None, employees=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
    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):
    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):
    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):
    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").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").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"
        }
    ])
    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")

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

    update_search_profile(1, 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"Name: {profile["name"]}, Industry:
{profile["industry"]}")
```

通过本章的学习，您应该已经掌握了如何使用SQLite数据库来存储和管理爬取到的资金项目数据以及用户定制的搜索档案。在下一章中，我们将开始构建后端API，实现前端与数据库的交互。

第六章：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。

6.1 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。

6.2 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。

6.3 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。

6.3.1 `app.py` 文件结构

首先，确保您的项目结构如下：

```
funding_search_tool/
├── venv/                       # 虚拟环境
├── scraper_ptj.py              # ptj.de 爬虫模块
├── scraper_eustartups.py       # eu-startups.com 爬虫模块
├── scraper_foerderdatenbank.py # foerderdatenbank.de 爬虫模块
├── scraper_ddb.py              # deutsche-digitale-bibliothek.de 爬虫模块
```

```
├── main_scraper.py          # 主爬虫调度模块
├── database.py              # 数据库操作模块
├── funds.db                 # SQLite数据库文件（运行后生成）
└── app.py                   # Flask应用主文件
```

现在，创建 `app.py` 文件，并添加以下基本代码：

```python
from flask import Flask, request, jsonify
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
import pandas as pd

app = Flask(__name__)

# 初始化数据库
init_db()

@app.route("/", methods=["GET"])
def home():
    return "欢迎来到资金项目搜索工具API！"

#
# 资金项目 API
#

@app.route("/api/funds", methods=["GET"])
def api_get_funds():
    source = request.args.get("source")
    keyword = request.args.get("keyword")

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

#
# 搜索档案 API
#

@app.route("/api/profiles", methods=["POST"])
def api_create_profile():
    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():
    profiles = get_search_profiles()
    return jsonify(profiles)

@app.route("/api/profiles/<int:profile_id>", methods=["GET"])
def api_get_profile_by_id(profile_id):
    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):
    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"})
    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):
    success = delete_search_profile(profile_id)
    if success:
        return jsonify({"message": "搜索档案删除成功"})
    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():
    # 运行所有爬虫并保存数据到数据库
    try:
        combined_df = run_all_scrapers()
        if not combined_df.empty:
            save_funds_data(combined_df)
            return jsonify({"message": f"成功爬取并保存
{len(combined_df)} 条资金项目数据"}), 200
        else:
            return jsonify({"message": "没有爬取到新数据"}), 200
    except Exception as e:
        return jsonify({"error": f"爬虫运行失败: {str(e)}"}), 500

if __name__ == "__main__":
    app.run(debug=True, host=\


'0.0.0.0', port=5000)
```

代码解释：

1. **导入必要的库：**
   - `Flask, request, jsonify`：Flask框架的核心组件，用于创建应用、处理请求和返回JSON响应。
   - `database` 模块中的函数：用于与SQLite数据库进行交互。
   - `main_scraper` 模块中的 `run_all_scrapers`：用于触发所有爬虫。
   - `pandas`：用于处理爬虫返回的DataFrame。
2. **`app = Flask(__name__)`**：创建一个Flask应用实例。
3. **`init_db()`**：在应用启动时调用，确保数据库和表已初始化。
4. **`@app.route()` 装饰器**：用于将URL路径映射到Python函数。`methods` 参数指定了该路由支持的HTTP方法。
5. **`request.args.get()`**：用于获取GET请求中的查询参数（URL中 `?` 后面的参数）。
6. **`request.get_json()`**：用于获取POST、PUT请求中发送的JSON格式的数据。
7. **`jsonify()`**：将Python字典或列表转换为JSON格式的响应。
8. **HTTP状态码**：在 `jsonify` 的第二个参数中指定HTTP状态码，例如 `200 OK`（成功）、`201 Created`（资源创建成功）、`400 Bad Request`（请求错误）、`404 Not Found`（资源未找到）、`409 Conflict`（冲突，如资源已存在）、`500 Internal Server Error`（服务器内部错误）。
9. **`@app.route("/api/profiles/<int:profile_id>", methods=["GET"])`**：`<int:profile_id>` 表示URL路径中的一个变量，它将被Flask自动转换为整数类型的 `profile_id` 参数传递给函数。

10. `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。

6.4 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/`，您应该看到“欢迎来到资金项目搜索工具API！”。
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：
      `json { "name": "我的第一个档案", "industry": "IT", "company_age": "<5年" }`
    - 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)。
    - 这将触发 `main_scraper.py` 中的所有爬虫运行，并将数据保存到数据库。您可以在终端中看到爬虫的输出。
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`。

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的触发。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。

第七章：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。

7.1 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）。

7.2 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
├── scraper_ptj.py              # ptj.de 爬虫模块
├── scraper_eustartups.py       # eu-startups.com 爬虫模块
├── scraper_foerderdatenbank.py # foerderdatenbank.de 爬虫模块
├── scraper_ddb.py              # deutsche-digitale-bibliothek.de 爬虫模块
├── main_scraper.py             # 主爬虫调度模块
├── database.py                 # 数据库操作模块
├── funds.db                    # SQLite数据库文件
├── 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
```

7.3 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>

        <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">

                <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>

            <h3>现有搜索档案</h3>
            <ul id="profile-list"></ul>
        </section>

        <section class="scraper-control">
            <h2>数据爬取控制</h2>
            <button id="trigger-scrape-btn">立即触发爬取</button>
            <p id="scrape-status"></p>
        </section>

        <section class="fund-search">
            <h2>资金项目搜索</h2>
            <label for="search-profile-select">选择搜索档案:</
label>
            <select id="search-profile-select">
                <option value="">-- 请选择 --</option>
            </select>
            <button id="search-funds-btn">搜索资金项目</button>

            <div id="fund-results">
                <h3>搜索结果</h3>
                <table id="funds-table">
                    <thead>
                        <tr>
                            <th>标题</th>
                            <th>描述</th>
                            <th>来源</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。
- `<div class="container">`：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。
- `scraper-control` 区块：
  - 一个按钮（`trigger-scrape-btn`），用于手动触发数据爬取。
  - 一个段落（`scrape-status`），用于显示爬取状态信息。
- `fund-search` 区块：
  - 一个下拉选择框（`search-profile-select`），用于选择一个已有的搜索档案进行资金项目搜索。
  - 一个按钮（`search-funds-btn`），用于触发搜索。
  - 一个表格（`funds-table`），用于动态显示搜索到的资金项目结果。初始时表格体为空。
  - 一个提示信息（`no-funds-message`），当没有搜索结果时显示。
- `<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。

7.4 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：

```
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: 20px;
    border-radius: 8px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h1, h2, h3 {
    color: #0056b3;
}

section {
    margin-bottom: 30px;
    padding-bottom: 20px;
    border-bottom: 1px solid #eee;
}

form label {
    display: block;
    margin-bottom: 5px;
    font-weight: bold;
}

form input[type="text"],
form select {
    width: calc(100% - 22px);
    padding: 10px;
    margin-bottom: 15px;
    border: 1px solid #ddd;
    border-radius: 4px;
}

form button {
    background-color: #007bff;
    color: white;
    padding: 10px 15px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 16px;
    margin-right: 10px;
}

form button:hover {
    background-color: #0056b3;
}
```

```css
form button#clear-form-btn {
    background-color: #6c757d;
}

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

ul#profile-list {
    list-style: none;
    padding: 0;
}

ul#profile-list li {
    background-color: #e9ecef;
    margin-bottom: 10px;
    padding: 10px;
    border-radius: 4px;
    display: flex;
    justify-content: space-between;
    align-items: center;
}

ul#profile-list li button {
    background-color: #ffc107;
    color: #333;
    padding: 5px 10px;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    margin-left: 5px;
}

ul#profile-list li button: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: 10px 15px;
    border: none;
    border-radius: 4px;
```

```
    cursor: pointer;
    font-size: 16px;
}

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

#scrape-status {
    margin-top: 10px;
    font-style: italic;
    color: #666;
}

#funds-table {
    width: 100%;
    border-collapse: collapse;
    margin-top: 20px;
}

#funds-table th,
#funds-table td {
    border: 1px solid #ddd;
    padding: 8px;
    text-align: left;
}

#funds-table th {
    background-color: #f2f2f2;
}

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

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

#funds-table td a {
    color: #007bff;
    text-decoration: none;
}

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

#no-funds-message {
    color: #666;
    text-align: center;
```

```
    margin-top: 20px;
}
```

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。
- **`.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。
- **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。
- **`profile-list`**：美化了搜索档案列表的显示，包括背景色、内边距和按钮样式。
- **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。

7.5 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。

```
document.addEventListener("DOMContentLoaded", () => {
    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 triggerScrapeBtn = document.getElementById("trigger-
scrape-btn");
    const scrapeStatus = document.getElementById("scrape-
status");

    const searchProfileSelect = document.getElementById("search-
profile-select");
    const searchFundsBtn = document.getElementById("search-
funds-btn");
```

```
    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 --- //

    // 清空表单
    const clearProfileForm = () => {
        profileIdInput.value = "";
        profileNameInput.value = "";
        profileIndustryInput.value = "";
        profileCompanyAgeInput.value = "";
        profileTurnoverInput.value = "";
        profileEmployeesInput.value = "";
        saveProfileBtn.textContent = "保存档案";
    };

    // 显示消息
    const showMessage = (element, message, type = "info") => {
        element.textContent = message;
        element.style.color = type === "error" ? "red" :
"green";
        setTimeout(() => {
            element.textContent = "";
            element.style.color = "";
        }, 5000);
    };

    // --- 搜索档案管理 --- //

    // 加载所有搜索档案
    const loadProfiles = async () => {
        try {
            const response = await fetch(`${API_BASE_URL}/
profiles`);
            const profiles = await response.json();
            profileList.innerHTML = ""; // 清空现有列表
            searchProfileSelect.innerHTML = "<option
value=\"\">-- 请选择 --</option>"; // 清空并添加默认选项

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

            profiles.forEach(profile => {
                // 更新档案列表
                const li = document.createElement("li");
                li.innerHTML = `
```

```
                    <span>${profile.name} (${profile.industry
|| '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);

                // 更新搜索档案选择框
                const option = document.createElement("option");
                option.value = profile.id;
                option.textContent = profile.name;
                searchProfileSelect.appendChild(option);
            });
        } catch (error) {
            console.error("加载搜索档案失败:", error);
            profileList.innerHTML = "<li>加载搜索档案失败。</li>";
        }
    };

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

        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(profileForm, result.message || "操作
成功！", "success");
                clearProfileForm();
                loadProfiles(); // 重新加载档案列表
            } else {
                showMessage(profileForm, result.error || "操作失
败！", "error");
            }
        } catch (error) {
            console.error("保存搜索档案失败:", error);
            showMessage(profileForm, "保存搜索档案时发生网络错误。",
"error");
        }
    });

    // 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}`);
                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";
```

```
            } catch (error) {
                console.error("加载档案进行编辑失败:", error);
                showMessage(profileForm, "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(profileForm, result.message
|| "删除成功！", "success");
                        loadProfiles(); // 重新加载档案列表
                    } else {
                        showMessage(profileForm, result.error
|| "删除失败！", "error");
                    }
                } catch (error) {
                    console.error("删除档案失败:", error);
                    showMessage(profileForm, "删除档案时发生网络错
误。", "error");
                }
            }
        }
    });

    // --- 数据爬取控制 --- //

    triggerScrapeBtn.addEventListener("click", async () => {
        triggerScrapeBtn.disabled = true;
        scrapeStatus.textContent = "正在触发爬取，请稍候...";
        scrapeStatus.style.color = "orange";

        try {
            const response = await fetch(`${API_BASE_URL}/
scrape`, {
                method: "POST",
            });
            const result = await response.json();
            if (response.ok) {
                showMessage(scrapeStatus, result.message || "爬取
成功！", "success");
            } else {
                showMessage(scrapeStatus, result.error || "爬取失
败！", "error");
            }
        } catch (error) {
```

```
            console.error("触发爬取失败:", error);
            showMessage(scrapeStatus, "触发爬取时发生网络错误。",
"error");
        } finally {
            triggerScrapeBtn.disabled = false;
        }
    });

    // --- 资金项目搜索 --- //

    searchFundsBtn.addEventListener("click", async () => {
        const selectedProfileId = searchProfileSelect.value;
        let queryParams = "";

        if (selectedProfileId) {
            // 如果选择了档案，则获取档案详情并构建查询参数
            try {
                const profileResponse = await fetch(`$
{API_BASE_URL}/profiles/${selectedProfileId}`);
                const profile = await profileResponse.json();

                if (profile.industry) queryParams +=
`&industry=${encodeURIComponent(profile.industry)}`;
                if (profile.company_age) queryParams +=
`&company_age=${encodeURIComponent(profile.company_age)}`;
                if (profile.turnover) queryParams +=
`&turnover=${encodeURIComponent(profile.turnover)}`;
                if (profile.employees) queryParams +=
`&employees=${encodeURIComponent(profile.employees)}`;

                // 注意：后端API的get_funds函数目前只支持source和
keyword,
                // 这里需要根据实际后端实现来调整queryParams的构建方式。
                // 为了演示，我们暂时只用一个通用的keyword来模拟搜索
                // 实际应用中，您可能需要修改后端get_funds函数来支持更多
筛选条件
                // 或者在前端进行更复杂的过滤
                queryParams = `?keyword=$
{encodeURIComponent(profile.name)}`; // 暂时用档案名称作为关键词

            } catch (error) {
                console.error("获取搜索档案详情失败:", error);
                showMessage(searchFundsBtn, "获取搜索档案详情失
败。", "error");
                return;
            }
        }

        try {
            const response = await fetch(`${API_BASE_URL}/funds$
{queryParams}`);
            const funds = await response.json();
```

```
            fundsTableBody.innerHTML = ""; // 清空现有表格内容
            if (funds.length === 0) {
                noFundsMessage.style.display = "block";
                fundsTableBody.style.display = "none";
            } else {
                noFundsMessage.style.display = "none";
                fundsTableBody.style.display = "table-row-
group";
                funds.forEach(fund => {
                    const tr = document.createElement("tr");
                    tr.innerHTML = `
                        <td>${fund.title}</td>
                        <td>${fund.description || 'N/A'}</td>
                        <td>${fund.source}</td>
                        <td><a href="${fund.link}"
target="_blank">链接</a></td>
                        <td>${new
Date(fund.crawled_at).toLocaleString()}</td>
                    `;
                    fundsTableBody.appendChild(tr);
                });
            }
        } catch (error) {
            console.error("搜索资金项目失败:", error);
            showMessage(searchFundsBtn, "搜索资金项目时发生网络错
误。", "error");
        }
    });

    // 页面加载时初始化
    loadProfiles();
});
```

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. **`clearProfileForm()`**：一个辅助函数，用于清空搜索档案表单的内容。
5. **`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。
6. **`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和选择框，然后遍历数据，动态创建 `<li>` 元素和 `<option>` 元素，并添加到页面中。
7. **`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。
8. **`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。
9. **`triggerScrapeBtn.addEventListener("click", ...)`**：
    - 监听“立即触发爬取”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。
10. **`searchFundsBtn.addEventListener("click", ...)`**：
    - 监听“搜索资金项目”按钮的点击事件。
    - 获取用户选择的搜索档案ID。如果选择了档案，则发送GET请求获取档案详情，并根据档案内容构建查询参数（目前简化为只用档案名称作为关键词）。
    - 发送GET请求到后端 `/api/funds` 接口，获取资金项目数据。
    - 清空现有表格内容，遍历获取到的资金项目数据，动态创建 `<tr>` 元素并添加到表格中。
    - 如果没有任何资金项目，则显示“没有找到符合条件的资金项目。”的提示信息。
11. **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。

7.6 配置Flask以提供静态文件

为了让Flask应用能够提供 `static` 文件夹中的HTML、CSS和JavaScript文件，我们需要在 `app.py` 中进行简单的配置。Flask默认会从名为 `static` 的文件夹中查找静态文件，并从名为 `templates` 的文件夹中查找模板文件。对于我们的简单应用，我们只需要确保 `static` 文件夹存在即可。

在 `app.py` 中，您无需额外配置，因为Flask默认就会查找 `static` 文件夹。我们只需要确保当用户访问根路径 `/` 时，返回 `index.html` 文件。

修改 `app.py` 中的 `home()` 函数，使其返回 `index.html`：

```
 from flask import Flask, request, jsonify, send_from_directory
# 导入 send_from_directory
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
import pandas as pd

app = Flask(__name__)

# 初始化数据库
init_db()

@app.route("/")
def index():
    return send_from_directory("static", "index.html")

# 其他API路由保持不变...

#
----------------------------------------------------------------------
# 资金项目 API
#
----------------------------------------------------------------------

@app.route("/api/funds", methods=["GET"])
def api_get_funds():
    source = request.args.get("source")
    keyword = request.args.get("keyword")

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

# ... (其他API路由)

if __name__ == "__main__":
    app.run(debug=True, host=\'0.0.0.0\', port=5000)
```

修改解释：

- **`send_from_directory("static", "index.html")`**：这个函数告诉Flask从 `static` 目录下查找并发送 `index.html` 文件给客户端。

7.7 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`：
   `bash cd funding_search_tool source venv/bin/activate # macOS/Linux # .\venv\Scripts\activate # Windows python app.py`
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**：点击“立即触发爬取”按钮，等待爬取完成（可能需要一些时间）。
- **搜索资金项目**：选择一个搜索档案，然后点击“搜索资金项目”按钮。您应该能看到表格中显示爬取到的资金项目。

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。在下一章中，我们将讨论如何自动化部署和测试项目。

第八章：自动化与部署

在前面的章节中，我们已经完成了资金项目搜索工具的核心功能开发。现在，我们将探讨如何将应用程序进行容器化，以便于部署和管理，并模拟实现定时任务来自动化数据爬取过程。容器化技术（如Docker）能够确保应用程序在不同环境中运行的一致性，而自动化任务则能大大提高系统的效率。

8.1 容器化：Docker

Docker是一种开源平台，用于开发、交付和运行应用程序。它通过使用容器（Containers）来打包应用程序及其所有依赖项，从而确保应用程序在任何环境中都能以相同的方式运行。容器是轻量级、可移植、自给自足的执行单元。

8.1.1 为什么使用Docker?

- **环境一致性**：解决了“在我的机器上可以运行”的问题。Docker容器包含了应用程序运行所需的一切（代码、运行时、系统工具、库等），无论在开发、测试还是生产环境，都能保证一致的运行。
- **隔离性**：每个容器都是一个独立的运行环境，容器之间相互隔离，互不影响。

- **可移植性**：容器可以在任何支持Docker的机器上运行，无论是本地开发机、虚拟机还是云服务器。
- **快速部署**：容器的启动速度快，部署过程标准化，可以大大缩短部署时间。
- **资源利用率高**：容器共享宿主机的操作系统内核，比虚拟机更轻量级，启动更快，占用资源更少。

8.1.2 Docker核心概念

- **Dockerfile**：一个文本文件，包含了一系列构建Docker镜像的指令。您可以把它看作是构建镜像的“菜谱”。
- **Image（镜像）**：一个只读的模板，包含了创建Docker容器所需的所有文件和配置。镜像可以从Docker Hub（一个公共的镜像仓库）下载，也可以通过Dockerfile构建。
- **Container（容器）**：镜像的运行实例。容器是轻量级、可执行的软件包，包含了运行应用程序所需的一切。
- **Docker Hub**：Docker官方提供的公共镜像仓库，您可以在这里找到各种预构建的镜像，也可以上传自己的镜像。

8.1.3 安装Docker

请访问Docker官方网站（https://www.docker.com/get-started）下载并安装适用于您操作系统的Docker Desktop（Windows和macOS）或Docker Engine（Linux）。安装完成后，您可以在终端中运行以下命令验证Docker是否安装成功：

```
 docker --version
docker compose version # 如果您安装了Docker Compose
```

8.1.4 编写Dockerfile

在项目根目录（`funding_search_tool`）下创建一个名为 `Dockerfile` 的文件（没有文件扩展名），并添加以下内容：

```
 # 使用官方Python基础镜像
FROM python:3.9-slim-buster

# 设置工作目录
WORKDIR /app

# 复制依赖文件
COPY requirements.txt .

# 安装Python依赖
RUN pip install --no-cache-dir -r requirements.txt

# 复制所有项目文件到容器的 /app 目录
COPY . .
```

```
# 暴露Flask应用运行的端口
EXPOSE 5000

# 定义容器启动时执行的命令
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。
- **`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。**注意：** 我们需要先生成`requirements.txt`文件。
- **`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。
- **`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。

8.1.5 生成 `requirements.txt`

在项目根目录，激活您的虚拟环境，然后运行以下命令生成`requirements.txt`文件：

```
pip freeze > requirements.txt
```

这将把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。

8.1.6 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。

8.1.7 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）。

8.2 模拟定时任务

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。

8.2.1 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，安装 `schedule` 库：

```
pip install schedule
```

然后，修改 `main_scraper.py`，添加定时任务的逻辑。为了不影响API的运行，我们通常会将定时任务放在一个独立的脚本中，或者在API启动时以独立线程的方式运行。

这里我们创建一个新的文件 `scheduler.py`：

```
import schedule
import time
from main_scraper import run_all_scrapers
from database import save_funds_data, init_db

def job():
    print("\n--- 定时任务开始执行爬取 ---")
    init_db() # 确保数据库已初始化
    combined_df = run_all_scrapers()
    if not combined_df.empty:
        save_funds_data(combined_df)
        print("定时任务：资金项目数据已更新。")
    else:
        print("定时任务：没有爬取到新数据。")

# 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
# schedule.every().day.at("10:30").do(job)

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

print("定时任务调度器已启动...")
```

```
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，并将结果保存到数据库。
- **`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。

运行 `scheduler.py`：

在终端中，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。

8.2.2 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。

8.3 MinIO模拟 (概念性介绍)

在原始的项目描述中提到了MinIO，它是一个开源的对象存储服务器，与Amazon S3兼容。在开发阶段，MinIO可以用来模拟云存储服务（如Google Cloud Storage或Amazon S3），而无需实际连接到云端。这对于本地测试数据上传和下载流程非常有用。

本项目教程中，我们为了简化，直接将爬取到的数据保存到本地SQLite数据库。如果您希望模拟将数据上传到云存储，可以：

1. **安装MinIO**：根据MinIO官方文档安装并运行MinIO服务器。
2. **修改爬虫模块**：在 `main_scraper.py` 或 `database.py` 中，添加使用MinIO客户端（如 `minio-py` 库）将DataFrame转换为CSV或JSON文件，然后上传到MinIO bucket的逻辑。

这部分内容超出了初学者教程的范围，但了解其概念有助于您在未来进行更复杂的云原生应用开发。

通过本章的学习，您已经掌握了如何使用Docker对应用程序进行容器化，以及如何模拟实现定时任务来自动化数据爬取。这些技能对于构建可部署和可维护的应用程序至关重要。在下一章中，我们将讨论如何对项目进行测试和完善。

第九章：测试与完善

开发一个功能完善的应用程序不仅仅是编写代码，还包括对其进行严格的测试，以确保其按预期工作，并且在出现问题时能够快速定位和修复。本章将介绍单元测试、集成测试的基本概念，并指导您如何使用Python的测试框架以及IDE的调试功能来完善您的资金项目搜索工具。

9.1 测试的重要性

测试是软件开发生命周期中不可或缺的一部分。它有以下几个主要目的：

- **验证功能**：确保应用程序的各个部分都按照需求规格正确实现。
- **发现缺陷**：在软件发布之前找出并修复错误（Bug），避免在生产环境中出现问题。
- **提高质量**：通过持续测试，提高代码的健壮性、可靠性和性能。
- **支持重构**：当您修改或重构代码时，测试可以提供安全网，确保您的更改没有引入新的问题。
- **文档**：良好的测试用例可以作为代码行为的一种文档，帮助其他开发者理解代码的功能。

9.2 单元测试

单元测试（Unit Testing）是对软件的最小可测试单元进行检查和验证。在Python中，一个“单元”通常是指一个函数、一个方法或一个类。单元测试的目的是确保每个独立的单元都能正确执行其预期的功能。

9.2.1 Python的 `unittest` 模块

Python标准库中内置了 `unittest` 模块，它提供了一个丰富的框架来编写和运行单元测试。`unittest` 模块支持测试自动化、测试共享设置和清理代码、将测试聚合到集合中等。

基本结构：

- **测试用例（Test Case）**：继承自 `unittest.TestCase` 的类。每个测试用例类可以包含多个测试方法。
- **测试方法（Test Method）**：以 `test_` 开头的方法。每个测试方法都应该测试一个特定的功能点。
- **断言（Assertions）**：`TestCase` 类提供了各种断言方法（如 `assertEqual()`，`assertTrue()`，`assertRaises()` 等），用于检查测试结果是否符合预期。
- **`setUp()` 和 `tearDown()`**：可选方法。`setUp()` 在每个测试方法运行前执行，用于准备测试环境；`tearDown()` 在每个测试方法运行后执行，用于清理测试环境。

9.2.2 编写爬虫模块的单元测试

我们将为 `main_scraper.py` 中的 `run_all_scrapers` 函数编写一个简单的单元测试。由于 `run_all_scrapers` 依赖于网络请求和数据库，为了进行单元测试，我们需要使用**模拟（Mocking）**技术来隔离外部依赖。

模拟是指用一个模拟对象替换真实对象，模拟对象的行为可以被控制，从而使测试更稳定、可控和快速。

创建一个名为 `test_scraper.py` 的文件：

```
 import unittest
from unittest.mock import patch, MagicMock
import pandas as pd
import os

# 导入需要测试的模块
from main_scraper import run_all_scrapers
from database import save_funds_data, init_db, get_db_connection
# 导入数据库相关函数

class TestScraper(unittest.TestCase):
```

```
    def setUp(self):
        # 在每个测试方法运行前执行
        # 创建一个临时的数据库文件用于测试
        self.db_path = "test_funds.db"
        if os.path.exists(self.db_path):
            os.remove(self.db_path)
        # 临时修改 database.py 中的数据库路径
        # 这是一个简单的模拟，实际项目中可能需要更复杂的依赖注入
        self.original_db_connect = sqlite3.connect
        sqlite3.connect = lambda db_name:
self.original_db_connect(self.db_path)
        init_db() # 初始化测试数据库

    def tearDown(self):
        # 在每个测试方法运行后执行
        # 删除临时数据库文件
        if os.path.exists(self.db_path):
            os.remove(self.db_path)
        # 恢复原始的 sqlite3.connect
        sqlite3.connect = self.original_db_connect

    @patch("main_scraper.scrape_ptj")
    @patch("main_scraper.scrape_eustartups")
    @patch("main_scraper.scrape_foerderdatenbank")
    @patch("main_scraper.scrape_ddb")
    def test_run_all_scrapers_success(self, mock_scrape_ddb,
mock_scrape_foerderdatenbank, mock_scrape_eustartups,
mock_scrape_ptj):
        # 模拟各个爬虫函数返回的数据
        mock_scrape_ptj.return_value = pd.DataFrame([
            {"id": "id1", "title": "Fund A", "description":
"Desc A", "link": "linkA", "source": "ptj.de"}
        ])
        mock_scrape_eustartups.return_value = pd.DataFrame([
            {"id": "id2", "name": "Investor B", "location":
"Loc B", "focus": "Focus B", "link": "linkB", "source": "eu-
startups.com"}
        ])
        mock_scrape_foerderdatenbank.return_value =
pd.DataFrame([
            {"id": "id3", "title": "Fund C", "description":
"Desc C", "link": "linkC", "source": "foerderdatenbank.de"}
        ])
        mock_scrape_ddb.return_value = pd.DataFrame([
            {"id": "id4", "title": "Item D", "description":
"Desc D", "link": "linkD", "source": "deutsche-digitale-
bibliothek.de"}
        ])

        # 运行被测试的函数
        combined_df = run_all_scrapers()
```

```
        # 验证结果
        self.assertFalse(combined_df.empty)
        self.assertEqual(len(combined_df), 4)
        self.assertIn("id1", combined_df["id"].values)
        self.assertIn("id2", combined_df["id"].values)
        self.assertIn("id3", combined_df["id"].values)
        self.assertIn("id4", combined_df["id"].values)

        # 验证数据是否保存到数据库
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT COUNT(*) FROM funds")
        count = cursor.fetchone()[0]
        conn.close()
        self.assertEqual(count, 4)

    @patch("main_scraper.scrape_ptj",
return_value=pd.DataFrame())
    @patch("main_scraper.scrape_eustartups",
return_value=pd.DataFrame())
    @patch("main_scraper.scrape_foerderdatenbank",
return_value=pd.DataFrame())
    @patch("main_scraper.scrape_ddb",
return_value=pd.DataFrame())
    def test_run_all_scrapers_no_data(self, *mocks):
        # 模拟所有爬虫函数返回空数据
        combined_df = run_all_scrapers()
        self.assertTrue(combined_df.empty)

        # 验证数据库中没有数据
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT COUNT(*) FROM funds")
        count = cursor.fetchone()[0]
        conn.close()
        self.assertEqual(count, 0)

    @patch("main_scraper.scrape_ptj",
side_effect=Exception("Network error"))
    @patch("main_scraper.scrape_eustartups",
return_value=pd.DataFrame())
    @patch("main_scraper.scrape_foerderdatenbank",
return_value=pd.DataFrame())
    @patch("main_scraper.scrape_ddb",
return_value=pd.DataFrame())
    def test_run_all_scrapers_with_error(self, mock_scrape_ddb,
mock_scrape_foerderdatenbank, mock_scrape_eustartups,
mock_scrape_ptj):
        # 模拟其中一个爬虫函数抛出异常
        combined_df = run_all_scrapers()
        # 即使有错误，也应该返回一个DataFrame（可能是空的或部分数据）
        self.assertIsInstance(combined_df, pd.DataFrame)
```

```
        # 验证数据库中没有数据（因为错误导致保存失败）
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT COUNT(*) FROM funds")
        count = cursor.fetchone()[0]
        conn.close()
        self.assertEqual(count, 0)

if __name__ == "__main__":
    unittest.main()
```

代码解释：

1. **`unittest.mock.patch`**：这是一个装饰器，用于模拟（mock）对象。在这里，我们模拟了 `main_scraper` 模块中调用的各个 `scrape_` 函数。通过 `return_value`，我们可以指定模拟函数返回什么数据；通过 `side_effect`，我们可以模拟函数抛出异常。
2. **`setUp()` 和 `tearDown()`**：
   - `setUp()`：在每个测试方法执行前，创建一个临时的 `test_funds.db` 数据库文件，并调用 `init_db()` 初始化表。为了让 `database.py` 中的 `get_db_connection` 连接到这个临时数据库，我们简单地修改了 `sqlite3.connect` 函数的行为。**注意：这种修改全局函数的方式在大型项目中可能不推荐，更推荐使用依赖注入或更复杂的模拟方式。**
   - `tearDown()`：在每个测试方法执行后，删除临时数据库文件，并恢复 `sqlite3.connect` 的原始行为，确保测试环境的清洁。
3. **`test_run_all_scrapers_success()`**：测试 `run_all_scrapers` 在所有爬虫都成功返回数据时的行为。我们断言返回的DataFrame不为空，长度正确，并且包含预期的ID，同时验证数据已成功保存到数据库。
4. **`test_run_all_scrapers_no_data()`**：测试当所有爬虫都返回空数据时的行为。
5. **`test_run_all_scrapers_with_error()`**：测试当其中一个爬虫抛出异常时的行为。我们期望 `run_all_scrapers` 能够捕获异常并返回一个DataFrame（尽管可能是空的），并且数据不会被保存到数据库。

9.2.3 运行单元测试

在终端中，进入项目根目录，激活虚拟环境，然后运行测试文件：

```
python -m unittest test_scraper.py
```

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。

9.3 集成测试

集成测试（Integration Testing）是对系统中不同模块或组件之间的接口进行测试，以验证它们是否能够协同工作。在本项目中，集成测试可以包括：

- 测试前端与后端API的通信是否正常。
- 测试后端API与数据库的交互是否正确。
- 测试爬虫模块与数据库模块的集成是否顺畅。

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。

9.3.1 手动集成测试

您可以通过以下步骤进行手动集成测试：

1. **启动后端API**：运行 `app.py`。
2. **访问前端页面**：在浏览器中打开 `http://127.0.0.1:5000/`。
3. **测试前端功能**：
    - 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”按钮，观察爬虫是否运行，以及资金项目数据是否更新到数据库。
    - 选择一个搜索档案，点击“搜索资金项目”，观察是否能正确显示结果。

通过这些手动操作，您可以验证前端、后端和数据库之间的交互是否符合预期。

9.4 调试

调试（Debugging）是识别、分析和修复代码中错误的过程。当您的程序没有按预期工作时，调试是找出问题根源的关键技能。

9.4.1 使用IDE的调试功能

现代IDE（如VS Code）提供了强大的调试功能，可以帮助您逐步执行代码、检查变量值、设置断点等。

VS Code调试步骤：

1. **设置断点**：在您想要暂停代码执行的行号旁边点击，会出现一个红点，这就是断点。
2. **启动调试**：
    - 打开您想要调试的Python文件（例如 `app.py` 或 `main_scraper.py`）。
    - 点击VS Code左侧活动栏的“Run and Debug”图标（或按下 `Ctrl+Shift+D`）。

    - 点击顶部的绿色播放按钮（“Start Debugging”），或者选择一个已配置的调试配置。
3. **调试操作**：
    - **步进（Step Over）**：执行当前行代码，如果当前行是函数调用，则跳过函数内部，直接执行函数返回后的下一行。
    - **步入（Step Into）**：执行当前行代码，如果当前行是函数调用，则进入函数内部执行。
    - **步出（Step Out）**：从当前函数中跳出，执行到调用该函数的地方。
    - **继续（Continue）**：继续执行代码，直到下一个断点或程序结束。
    - **重新启动（Restart）**：重新启动调试会话。
    - **停止（Stop）**：停止调试会话。
4. **查看变量**：在调试过程中，您可以在“Variables”面板中查看当前作用域内所有变量的值。您也可以在代码中选择变量，右键点击选择“Add to Watch”来持续观察特定变量的值。
5. **调试控制台**：在“Debug Console”中，您可以执行Python代码，检查表达式的值，或者打印调试信息。

通过熟练使用IDE的调试功能，您可以大大提高排查问题的效率。

9.5 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。

9.5.1 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。

```
try:
    # 可能会引发错误的代码块
    response = requests.get("http://invalid-url-example.com",
timeout=5)
    response.raise_for_status()
    print("请求成功")
except requests.exceptions.RequestException as e:
    # 捕获并处理 RequestException 异常
    print(f"网络请求错误: {e}")
except Exception as e:
    # 捕获其他所有异常
    print(f"发生未知错误: {e}")
finally:
    # 无论是否发生异常，都会执行的代码块
    print("请求处理完成")
```

在您的爬虫和API代码中，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。

9.5.2 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。

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:

```
import logging

# 配置日志输出到文件和控制台
logging.basicConfig(
    level=logging.INFO, # 设置最低日志级别
    format="%(asctime)s - %(name)s - %(levelname)s - %
(message)s",
    handlers=[
        logging.FileHandler("app.log"), # 输出到文件
        logging.StreamHandler()         # 输出到控制台
    ]
)

logger = logging.getLogger(__name__)

# 在代码中使用日志
logger.info("应用程序启动")
logger.debug("这是一个调试信息")
logger.warning("检测到潜在问题")
logger.error("发生了一个错误")
```

您可以在 `app.py`、`main_scraper.py` 和 `database.py` 中引入 `logging` 模块，并在关键位置添加日志记录，以便更好地追踪程序的运行情况和排查问题。

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。

第十章：项目完善与扩展

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。

10.1 错误处理与用户反馈优化

虽然我们已经在代码中加入了 `try...except` 进行错误处理，但用户界面的错误反馈仍然可以做得更好。当后端API返回错误时，前端应该向用户显示清晰、友好的错误消息，而不是仅仅在控制台打印错误。

优化建议：

- **统一错误响应格式**：后端API可以定义一个统一的错误响应格式，例如：`json { "status": "error", "code": 400, "message": "缺少搜索档案名称", "details": "请提供有效的档案名称" }`
- **前端错误消息展示**：前端JavaScript可以解析这种统一的错误响应，并在页面上以更醒目的方式（如弹窗、红色警告文本）显示 `message` 或 `details` 字段。
- **日志记录增强**：在后端，确保所有重要的错误和异常都被详细记录到日志文件中，以便于后续的排查和分析。

10.2 性能优化

随着数据量的增长和用户数量的增加，应用程序的性能可能会成为瓶颈。以下是一些可以考虑的性能优化方向：

- **数据库索引**：为 `funds` 表和 `search_profiles` 表中经常用于查询的字段（如 `funds.source`, `funds.title`, `search_profiles.name`）添加数据库索引，可以显著提高查询速度。
    - **示例SQL（在 `init_db()` 中执行）**：`sql CREATE INDEX IF NOT EXISTS idx_funds_source ON funds (source); CREATE INDEX IF NOT EXISTS idx_funds_title ON funds (title); CREATE INDEX IF NOT EXISTS idx_profiles_name ON search_profiles (name);`
- **分页加载**：当前资金项目搜索结果是全部加载的。当数据量很大时，这会影响页面加载速度。可以修改后端API，使其支持分页参数（如 `page`, `limit`），前端只请求当前页的数据。
- **缓存**：对于不经常变化但频繁访问的数据（如搜索档案列表），可以在后端或前端引入缓存机制，减少数据库查询次数。

- **异步任务**：数据爬取是一个耗时操作。当前它是同步执行的，会阻塞API响应。可以考虑使用异步任务队列（如Celery配合Redis或RabbitMQ）来处理爬取任务，使其在后台运行，API可以立即返回“爬取任务已提交”的响应。

10.3 安全性考虑

对于任何Web应用程序，安全性都是至关重要的。对于初学者项目，我们主要关注以下几点：

- **输入验证**：对所有用户输入（无论是通过前端表单还是直接通过API）进行严格的验证和清理，防止SQL注入、XSS（跨站脚本攻击）等常见漏洞。
- **HTTPS**：在生产环境中，始终使用HTTPS来加密客户端和服务器之间的通信，保护数据传输的安全性。
- **API密钥/认证**：如果您的API需要对外提供服务，考虑引入API密钥或用户认证机制，确保只有授权用户才能访问。
- **`debug=False`**：在生产环境中部署Flask应用时，务必将 `app.run(debug=True)` 中的 `debug` 参数设置为 `False`，以避免泄露敏感信息和安全漏洞。

10.4 部署到生产环境

虽然我们已经使用Docker进行了容器化，但在生产环境中部署Flask应用通常需要更专业的Web服务器（如Gunicorn, uWSGI）和反向代理（如Nginx）。

生产部署栈示例：

- **Web服务器**：Gunicorn (或 uWSGI) 用于运行Flask应用，它是一个WSGI HTTP服务器，比Flask自带的开发服务器更健壮、高效。
- **反向代理**：Nginx (或 Apache) 作为反向代理，负责接收外部请求，并将请求转发给Gunicorn。Nginx还可以处理静态文件、负载均衡、SSL终止等。
- **进程管理**：Supervisor (或 systemd) 用于管理Gunicorn进程，确保应用在崩溃后能自动重启。

简化的Docker Compose部署：

如果您想在生产环境中使用Docker，可以考虑使用 `docker-compose` 来管理多个服务（如Flask应用、数据库、定时任务）。

创建一个 `docker-compose.yml` 文件：

```
version: '3.8'

services:
  web:
    build: .
    ports:
```

```
      - "5000:5000"
    volumes:
      - .:/app # 将宿主机代码挂载到容器，方便开发调试
      - funds_data:/app/funds.db # 将数据库文件挂载为数据卷
    environment:
      FLASK_ENV: production # 设置Flask为生产环境
    command: gunicorn -w 4 -b 0.0.0.0:5000 app:app # 使用Gunicorn
运行Flask应用

  scheduler:
    build: .
    volumes:
      - .:/app
      - funds_data:/app/funds.db
    command: python scheduler.py

volumes:
  funds_data:
```

说明：

- `web` 服务：构建Flask应用镜像，映射端口，挂载代码和数据库数据卷，并使用Gunicorn运行应用。
- `scheduler` 服务：构建定时任务镜像，挂载代码和数据库数据卷，并运行 `scheduler.py`。
- `volumes`：定义一个数据卷 `funds_data`，用于持久化存储 `funds.db` 文件，确保容器重启后数据不会丢失。

运行 `docker-compose up -d` 即可启动所有服务。

10.5 未来扩展方向

这个项目只是一个起点，您可以根据自己的兴趣和需求进行无限的扩展。以下是一些可能的方向：

1. **更智能的资金项目匹配**：
    - **自然语言处理 (NLP)**：使用NLP技术（如文本分类、实体识别）从资金项目描述中提取更多结构化信息。
    - **机器学习推荐**：基于用户的搜索历史和偏好，推荐更相关的资金项目。
    - **语义搜索**：允许用户使用自然语言描述需求，系统能够理解并返回语义相关的资金项目。
2. **更丰富的数据源**：
    - 集成更多政府机构、银行、投资机构的资金项目数据。
    - 考虑爬取非结构化数据（如新闻文章、博客），并从中提取资金信息。

3. **用户管理与认证**：
    - 实现用户注册、登录功能，允许用户拥有自己的搜索档案和收藏夹。
    - 基于用户角色提供不同的权限。
4. **通知系统**：
    - 当有新的符合用户搜索档案的资金项目出现时，通过邮件、短信或站内信进行通知。
5. **数据可视化与报告**：
    - 为用户提供更直观的数据可视化图表，展示资金项目的趋势、分布等。
    - 生成定制化的资金项目报告（PDF、Excel）。
6. **高级爬虫功能**：
    - 处理JavaScript动态加载内容（使用Selenium或Playwright）。
    - 应对更复杂的反爬机制（如验证码识别、IP代理池）。
7. **前端框架升级**：
    - 如果您对前端开发感兴趣，可以尝试使用React、Vue或Angular等现代前端框架来重构前端界面，提供更丰富的交互和更好的用户体验。
8. **云服务集成**：
    - 将SQLite数据库替换为云数据库（如PostgreSQL on Cloud SQL）。
    - 将定时任务部署到云函数（如Google Cloud Functions, AWS Lambda）。
    - 使用云存储（如Google Cloud Storage, AWS S3）存储爬取到的原始数据或文件。

通过不断学习和实践，您可以将这个资金项目搜索工具打造成一个功能强大、性能卓越的应用程序。祝您在软件开发的道路上越走越远！

参考资料：

- Python官方文档: https://docs.python.org/
- Flask官方文档: https://flask.palletsprojects.com/
- Requests库文档: https://docs.python-requests.org/
- BeautifulSoup4文档: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
- Pandas官方文档: https://pandas.pydata.org/docs/
- SQLite官方网站: https://www.sqlite.org/
- Docker官方文档: https://docs.docker.com/
- Schedule库文档: https://schedule.readthedocs.io/
- Visual Studio Code官方文档: https://code.visualstudio.com/docs