

Deepseek企业级Agent项目开发实战

Part 7. 基于API驱动的Microsoft GraphRAG工程化架构实现

通过前面六节的学习，我们已经详细讲解了 Microsoft GraphRAG 在构建索引和检索两个阶段时其底层运行的原理和各个 workflow 的实现细节。在这个基础上，我们又进一步的对 Microsoft GraphRAG 做了源码级的二次开发，主要实现如下优化策略：

- 1. 对 .csv 文件的索引流程做了优化，不再使用简单的按照 token 数量切分，而是结合 csv 文件的预处理流程，构建分层分级动态切分，从而保证每个 row 可以保留完整的语义；
- 2. 自定义集成了支持并行解析的 MinerU 服务，支持对 PDF 格式、图像（.jpg 及 .png）、word（.doc 及 .docx）、以及 PowerPoint（.ppt 及 .pptx）在内的多种文档格式的文档解析；
- 3. 支持对图像、表格类型数据的分析和处理，接入多模态（视觉）模型对图片和表格进行分析，并生成对应的文本描述，从而保证 GraphRAG 可以处理更多的数据类型；
- 4. 新增 PDF 等格式文件的文档切分器，通过结构感知动态分块策略，有效保留文本和表格/图谱多模式语义组合长文本及图像、表格等单独语义的隔离；

上述各阶段的优化策略，其一是扩展了 Microsoft GraphRAG 对数据类型的支持，针对实际业务中常见的数据类型，如 csv、PDF、图片等，都可以通过我们二次开发后的 Microsoft GraphRAG 索引流程进行处理，并生成对应的图谱；其二是通过优化索引流程，在保证索引质量的前提下，显著提升了索引的效率，从而可以支持更大规模的数据处理。（MinerU 的并行解析能力，并实现了服务分离）。而对检索来说，检索效果的好坏根本原因在于索引阶段构建的图谱质量，如果索引阶段构建的图谱质量不高，那么检索阶段无论使用什么样的检索策略，效果都不会太好。这一点是需要大家明确理解的。

因此，在大家实际基于 Microsoft GraphRAG 作为核心架构构建本地知识库业务时，重心应该放在索引阶段，通过优化索引流程，构建高质量的图谱，从而保证检索阶段的效果。

而对于启动 Microsoft GraphRAG 的索引和检索流程，在之前的实践中我们都是通过 Microsoft GraphRAG 的 CLI 命令行启动或者通过源码的 poetry 命令启动。这两种不同的使用方法本质上的差异如下表所示：

CLI 和 Poe 启动方式对比

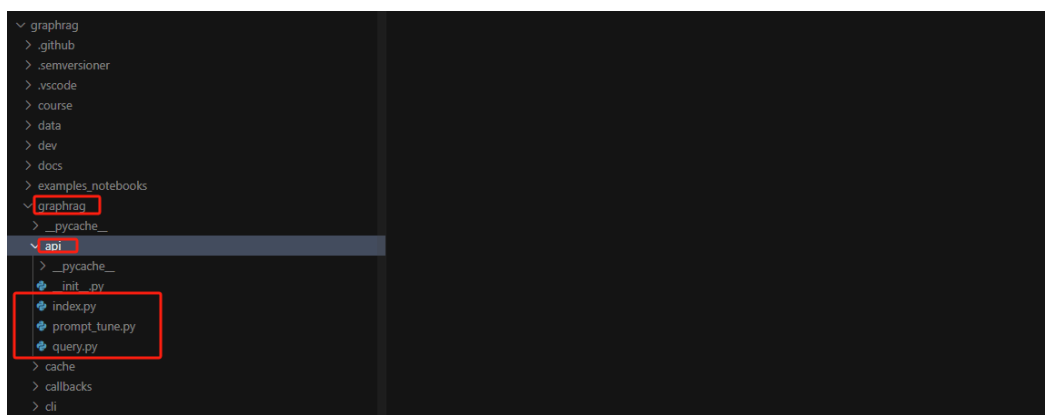
特性	CLI 启动方式	Poe 启动方式
安装方式	pip 安装	源码安装
启动方式	在控制台输入命令启动	在控制台输入命令启动
可用参数	根据 -help 提供的参数指定构建索引/检索流程	可以任意添加参数，满足定制化的需求
灵活度	灵活度低，无法满足一些特殊的需求	灵活度高，可以修改源码，任意添加参数

这里需要注意的是：<两种启动方式的共同点是都需要在控制台输入命令启动。而通过控制台启动的索引构建、查询的流程如果对于离线构建索引和检索来说，是没有任何问题的。因为离线构建索引和检索通常都是一次性操作，不需要频繁调用。但是，对于需要频繁调用索引和检索的场景来说，这种方式就显得有些笨重了，而且是没有办法和业务系统进行融合的。比如用户在前端页面中点击某个按钮，如果每次都需要在控制台输入命令，那么这种应用方式显然是不合适的。

因此，在本节中，我们将介绍如何通过 Python 代码程序启动 Microsoft GraphRAG 的索引和检索流程，并实际开发实现一个 web 服务来展示 Microsoft GraphRAG 的检索效果。同时，针对业务场景中的一些常见需求：如何处理多类型文本、如何做知识图谱更新等实现思路和优化方法，也将一一进行介绍。

我们首先来看如何通过 Python 代码程序启动 Microsoft GraphRAG 的索引构建流程。

在 Microsoft GraphRAG 项目源码中，官方封装并提供了 index、query 和 prompt_tune 三个流程的 API 接口，我们可以直接通过 Python 代码调用这些 API 接口，从而实现索引构建、检索和提示词调优等流程。其源码存储位置为：graphrag\graphrag\api 目录下。



其中 index.py 中定义了 Microsoft GraphRAG 的索引构建的 Pipeline 接口，prompt_tune.py 中定义了提示词调优的 Pipeline 接口，而 query.py 中定义了检索的 Pipeline 接口。根据我们之前的章节，大家应该也能深刻理解到对于使用 Microsoft GraphRAG 构建本地知识库来说，正确的流程就是先通过 prompt_tune 根据实时实际的数据进行提示词自适应调优，然后再构建索引，最后才是通过 query 检索。

因此，我们也按照这个顺序来给大家讲解如何自定义接入这三个 API 接口，从而完成代码层面的应用。首先来看 prompt_tune 提示词调优的 API 接口。

1. Prompt动态编排服务接口设计

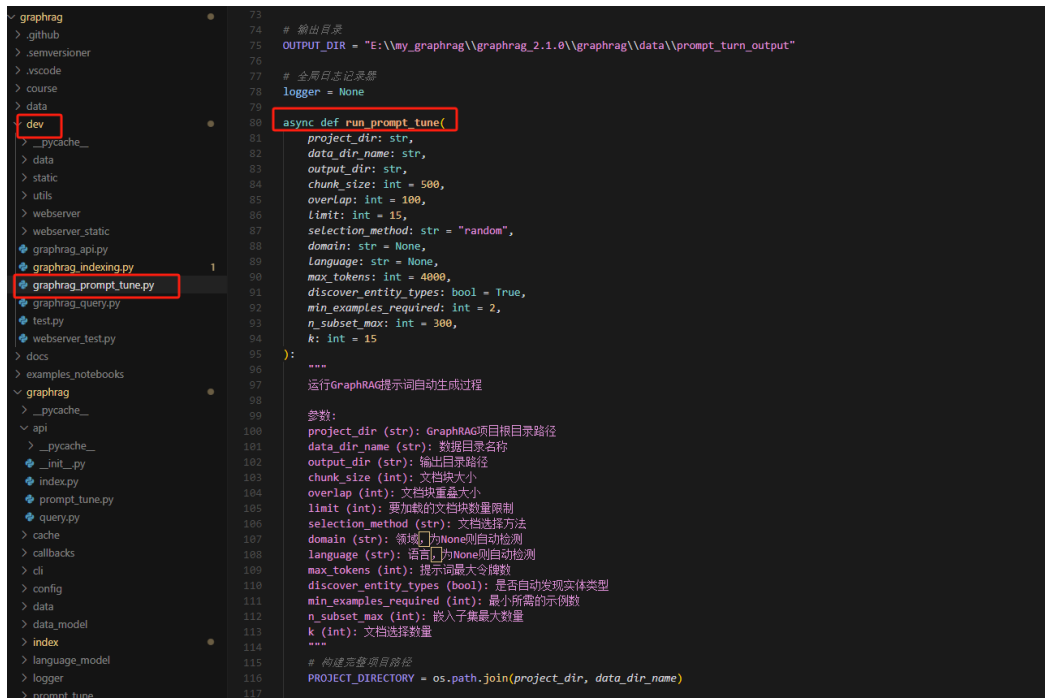
在《MicroSoft GraphRAG 深度实战 - Part 5: Microsoft GraphRAG 多源数据索引构建方案》的 2.4 实体关系提取优化策略 小节中，我们给大家详细讲解了 Microsoft GraphRAG 的提示词调优的实现原理和实现细节。在 CLI 或者 Poetry 命令行中可接入的命令参数在 API 的调用过程中同样适用。主要如下所示：

Auto Prompt Tuning 优化参数

参数名	描述	状态	默认值
--config	配置文件的路径。这是加载数据和模型设置所必需的。	必需	无
--root	数据项目根目录，包括配置文件（YML、JSON 或 .env）。	可选	当前目录
--domain	与输入数据相关的域，例如“空间科学”、“微生物学”或“环境新闻”。如果留空，则将从输入数据中推断出域。	可选	无
--selection-method	选择文档的方法。选项包括 all、random、auto 或 top。	可选	random
--limit	使用随机或顶部选择时要加载的文本单元的限制。	可选	15
--language	用于输入处理的语言。如果它与输入的语言不同，则将进行翻译。默认值为“”，表示将自动从输入中检测。	可选	自动检测
--max-tokens	提示生成的最大令牌数。	可选	2000
--chunk-size	用于从输入文档生成文本单元的标记大小。	可选	200
--n-subset-max	使用自动选择方法时嵌入的文本块数量。	可选	300
--k	使用自动选择方法时要选择的文档数量。	可选	15
--min-examples-required	实体提取提示所需的最小示例数。	可选	2
--discover-entity-types	允许自动发现和提取实体。当数据涵盖大量主题或高度随机化时，建议使用此选项。	可选	无
--output_path	保存生成的提示的文件夹。	可选	prompts

当然，其对应的 API 接口无非就是将这些命令参数转换为 Python 代码，并通过直接调用工具函数的形式构建出一个可以直接运行的 Pipeline。其源码存储位置为：graphrag\graphrag\api\prompt_tune.py

这里不再重复讲解，如不清楚实现细节，请看《Microsoft GraphRAG 深度实战 - Part 5: Microsoft GraphRAG 多源数据索引构建方案》的 2.4 实体关系提取优化策略 小节。我们直接来看如何通过 Python 代码调用 prompt_tune 的 API 接口。根据该接口形式，在源码中新建了一个 dev 文件夹，并在其中创建了 graphrag_prompt_tune.py 文件，其源码存放位置为：graphrag\dev\graphrag_prompt_tune.py，如下所示：



```
73
74 # 输出目录
75 OUTPUT_DIR = "E:\\my_graphrag\\graphrag_2.1.0\\graphrag\\data\\prompt_tune_output"
76
77 # 全局日志记录器
78 logger = None
79
80
81 async def run_prompt_tune(
82     project_dir: str,
83     data_dir_name: str,
84     output_dir: str,
85     chunk_size: int = 500,
86     overlap: int = 100,
87     limit: int = 15,
88     selection_method: str = "random",
89     domain: str = None,
90     language: str = None,
91     max_tokens: int = 4000,
92     discover_entity_types: bool = True,
93     min_examples_required: int = 2,
94     n_subset_max: int = 300,
95     k: int = 15
96 ):
97     """
98     运行GraphRAG提示词自动生成过程
99
100     参数:
101     project_dir (str): GraphRAG项目根目录路径
102     data_dir_name (str): 数据目录名称
103     output_dir (str): 输出目录路径
104     chunk_size (int): 文档块大小
105     overlap (int): 文档块重叠大小
106     limit (int): 要加载的文档块数量限制
107     selection_method (str): 文档选择方法
108     domain (str): 领域(为None则自动检测)
109     language (str): 语言(为None则自动检测)
110     max_tokens (int): 提示词最大令牌数
111     discover_entity_types (bool): 是否自动发现实体类型
112     min_examples_required (int): 最小所需的示例数
113     n_subset_max (int): 嵌入子集最大数量
114     k (int): 文档选择数量
115     """
116     # 构建完整项目路径
117     PROJECT_DIRECTORY = os.path.join(project_dir, data_dir_name)
```

run_prompt_tune 函数是 graphrag_prompt_tune.py 文件中的核心函数，其核心思路就是自定义加载一系列适用于 prompt_tune 参数配置，传入到 prompt_tune 的 API 接口函数中的 generate_indexing_prompts 函数，从而实现 Microsoft GraphRAG 中代码环境下运行提示词调优的需求。支持的配置文件如下所示：

```
# =====
# 用户配置参数 - 直接修改这里的变量
# =====

# GraphRAG项目根目录路径
PROJECT_DIR = "C:\\Users\\Lenovo\\Desktop\\folder\\Agent\\code\\code\\backend\\deepseek_agent\\llm_backend\\app\\graphrag"

# 数据目录名称（相对于项目根目录）
DATA_DIR_NAME = "data"

# 文档块大小
CHUNK_SIZE = 500

# 文档块重叠大小
OVERLAP = 100

# 要加载的文档块数量限制
# 注意：如果文档较少，请设置较小的值
LIMIT = 5

# 文档选择方法: "random", "auto", "all", "top"
# 参考DocSelectionType枚举的有效值
SELECTION_METHOD = "random"

# 领域（为None则自动检测）
DOMAIN = None
```

```

# 语言（为None则自动检测）
LANGUAGE = None

# 提示词最大令牌数
MAX_TOKENS = 4000

# 是否自动发现实体类型
DISCOVER_ENTITY_TYPES = True

# 最小所需的示例数
MIN_EXAMPLES_REQUIRED = 2

# 嵌入子集最大数量（对auto选择方法有效）
N_SUBSET_MAX = 300

# 文档选择数量（对auto选择方法有效）
K = 15

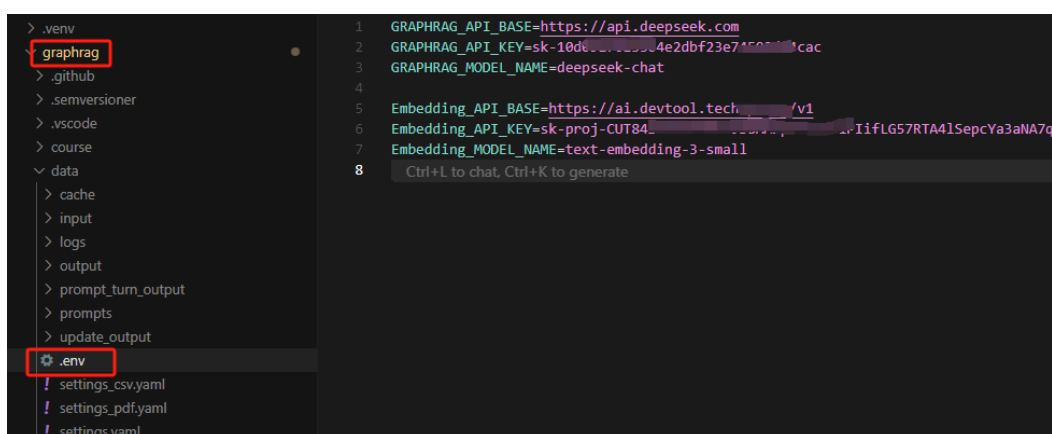
# 输出目录
OUTPUT_DIR = "C:\\Users\\Lenovo\\Desktop\\folder\\Agent\\code\\code\\backend\\
deepseek_agent\\llm_backend\\app\\graphrag\\data\\pdf_prompt_turn_output"

```

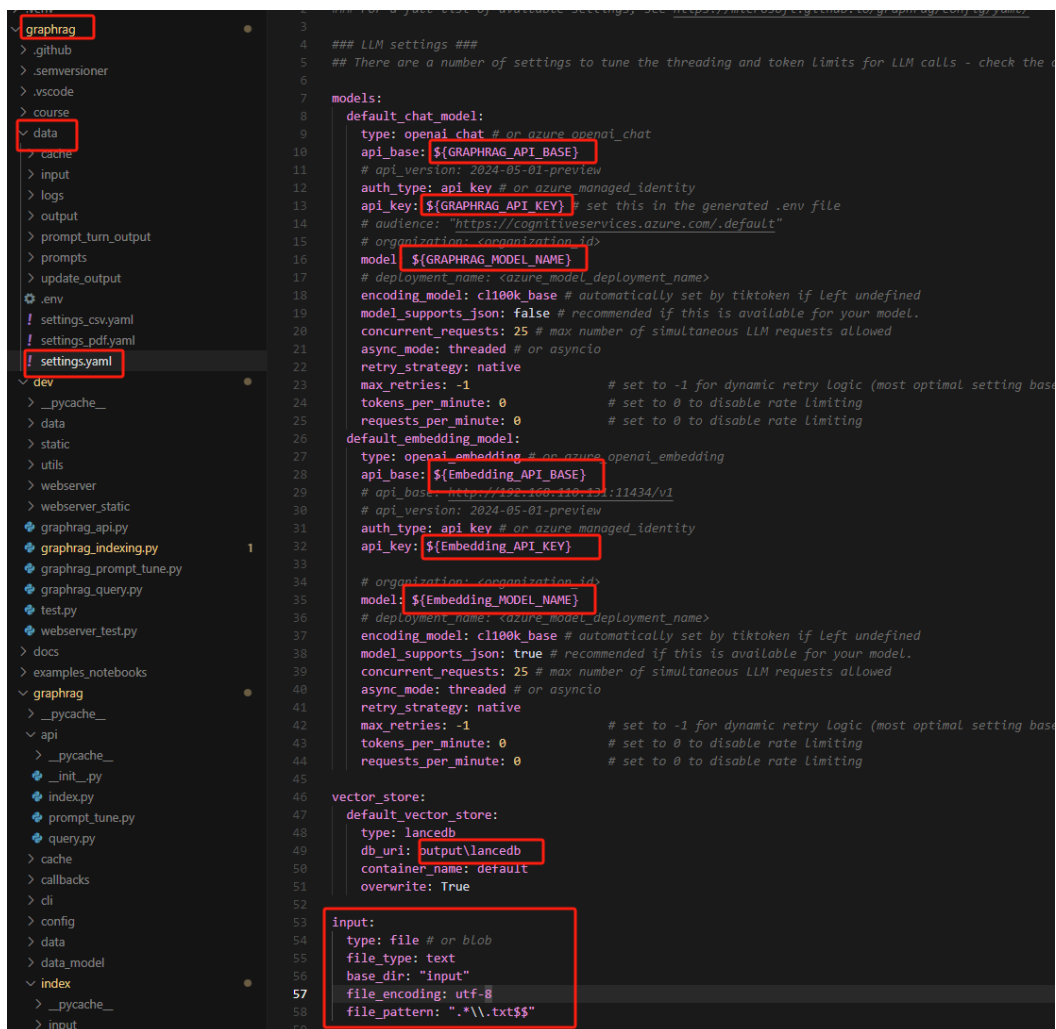
其中需要重点说明的就是：这里 `PROJECT_DIR` 是 Microsoft GraphRAG 项目的根目录，要填写服务器上实际存储的 Microsoft GraphRAG 路径。而 `DATA_DIR_NAME` 是数据目录名称，也就是大家通过 `poetry run poe init --root xxx` 命令初始化时指定的目录。而输出目录 `OUTPUT_DIR`，则是配置用于存放微调后新提示词的文件夹。

同时需要注意的是：按照上述的参数规范自定义修改 `prompt_tune` 的参数配置后，还需要理解的是：`prompt_tune` 无论是命令行调用，还是通过 Python 代码调用，本质上加载的配置文件都是 `settings.yaml` 文件，因此，大家也需要保证 `settings.yaml` 文件中正确配置好了参数，其中比较关键的是：大模型连接信息、Embedding 模型的连接信息，以及要用于提示词调优的原始文档，即通过 `input` 下的参数进行过滤和选择。

大家可以参考如下的配置进行自定义修改。首先是 `.env` 文件。在这个文件中，我们可以添加更多的环境变量以优化 `settings.yaml` 文件中的参数配置。如下所示：



然后是 `settings.yaml` 文件。在这个文件中，我们可以添加更多的参数配置以优化 `prompt_tune` 的参数配置。主要：这里通过 `{{ }}` 的方式，将 `.env` 文件中的环境变量引入到 `settings.yaml` 文件中，因此只要修改了 `.env` 文件中的环境变量，`settings.yaml` 文件中也会自动更新。核心关注的是：`input` 下面要选择哪种格式的文档作为提示词调优的原始文档。



如上所示，我们这里选择是 txt 的文件进行提示词调优的原始文档。同样，因仅用于功能测试，我们这里还是选择 technology_companies.txt 这篇短文档进行快速验证。



当配置好了 settings.yaml 文件，同时也将 graphrag_prompt_tune.py 文件中用于控制 prompt_tune 的参数配置修改好后，则可以直接运行该 Python 脚本，执行代码层面的提示词调优流程。执行如下命令：

```
cd dev
python graphrag_prompt_tune.py
```



```
82 output:
83 type: file # [file, blob, cosmosdb]
84 base_dir: "output"
85
86 ### Workflow settings ###
87 extract_graph:
88 model_id: default_chat_model
89 prompt: "prompt_turn_output/extract_graph_zh.txt"
90 entity_types: [company, person, product, technology, service, location, university, investment, acquisition, operating system]
91 max_gleanings: 1
92
93 summarize_descriptions:
94 model_id: default_chat_model
95 prompt: "prompt_turn_output/summarize_descriptions_zh.txt"
96 max_length: 300
97
98 extract_graph_nlp:
99 text_analyzer:
100 extractor_type: regex_english # [regex_english, syntactic_parser, cfg]
101
102 extract_claims:
103 enabled: false
104 model_id: default_chat_model
105 prompt: "prompts/extract_claims.txt"
106 description: "Any claims or facts that could be relevant to information discovery."
107 max_gleanings: 1
108
109 community_reports:
110 model_id: default_chat_model
111 graph_prompts: "prompt_turn_output/community_report_graph_zh.txt"
112 text_prompts: "prompts/community_report_text_zh.txt"
113 max_length: 2000
114 max_input_length: 8000
```

全部调整并配置好以后，接下来我们来看如何通过 Python 代码执行 Microsoft GraphRAG 的索引构建流程。

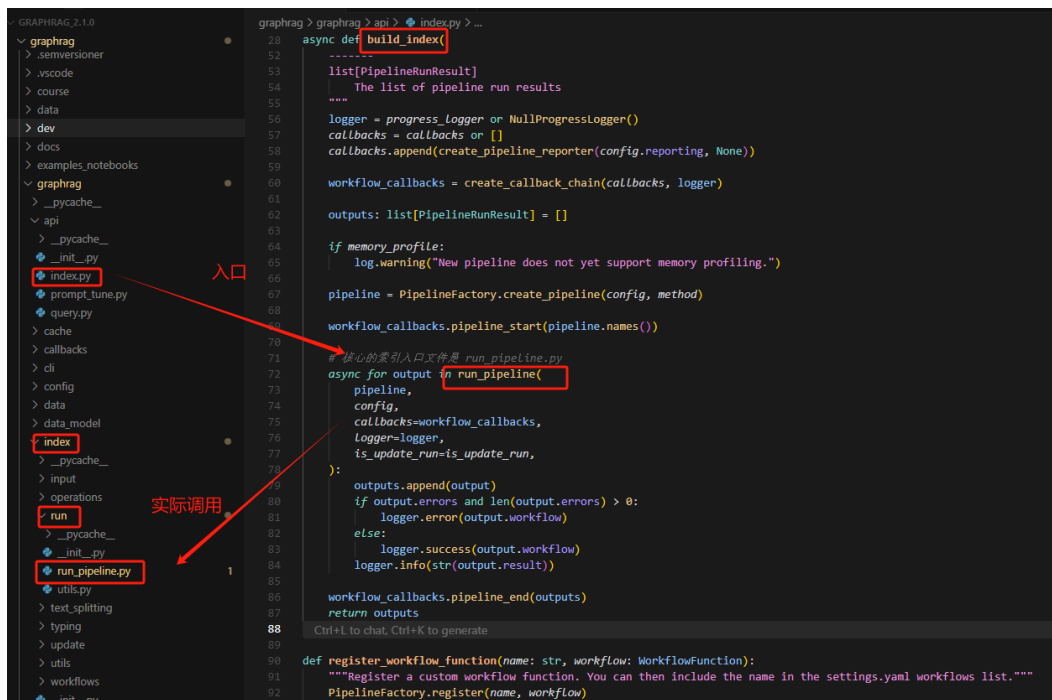
2. 索引构建的工程化接口封装

Indexing 的 API 接口源码文件存放在 graphrag\graphrag\api 目录下的 index.py 文件中，与 prompt_tune 的 API 接口类似，借助 index.py 文件中定义 build_index 函数便可以实现通过 Python 代码执行 Microsoft GraphRAG 的索引构建流程。如下所示：

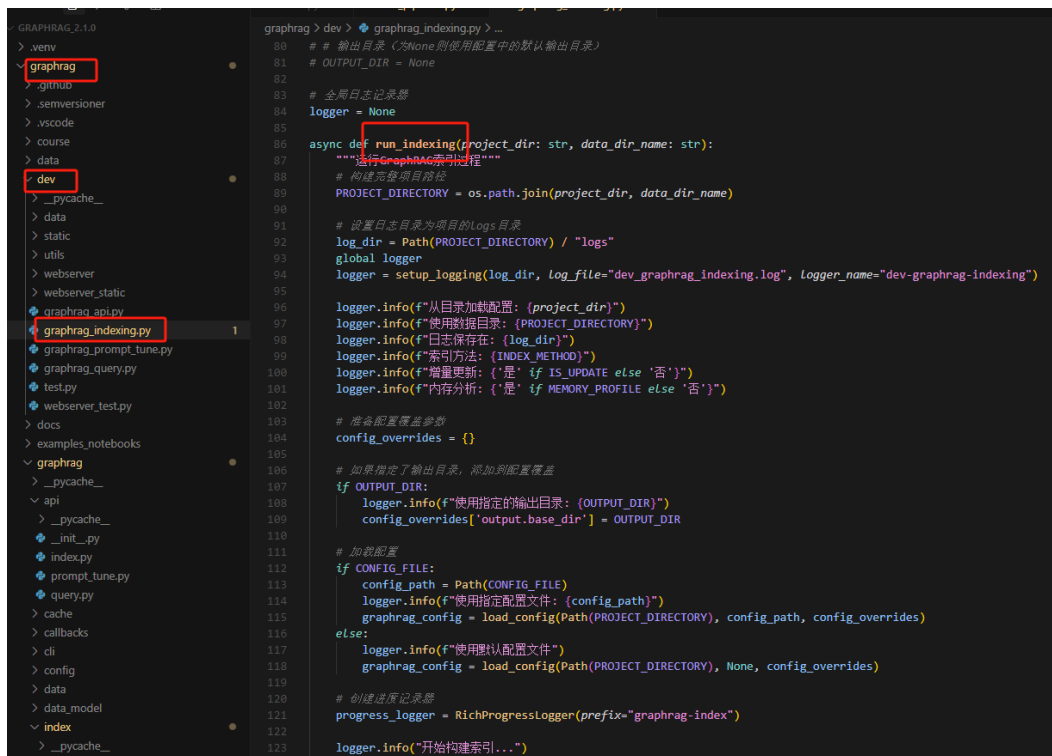
C

```
14 from graphrag.callbacks.workflow_callbacks import WorkflowCallbacks
15 from graphrag.config.enums import IndexingMethod
16 from graphrag.config.models.graph_rag_config import GraphRagConfig
17 from graphrag.index.run_pipeline import run_pipeline
18 from graphrag.index.run_utils import create_callback_chain
19 from graphrag.index.typing.pipeline_run_result import PipelineRunResult
20 from graphrag.index.typing.workflow import WorkflowFunction
21 from graphrag.index.workflows.factory import PipelineFactory
22 from graphrag.logger.base import ProgressLogger
23 from graphrag.logger.null_progress import NullProgressLogger
24
25 log = logging.getLogger(__name__)
26
27
28 async def build_index(
29     config: GraphRagConfig,
30     method: IndexingMethod = IndexingMethod.Standard,
31     is_update_run: bool = False,
32     memory_profile: bool = False,
33     callbacks: list[WorkflowCallbacks] | None = None,
34     progress_logger: ProgressLogger | None = None,
35 ) -> list[PipelineRunResult]:
36     """Run the pipeline with the given configuration.
37
38     Parameters
39     -----
40     config : GraphRagConfig
41         The configuration.
42     method : IndexingMethod default=IndexingMethod.Standard
43         Styling of indexing to perform (full LLM, MLP + LLM, etc.).
44     memory_profile : bool
45         Whether to enable memory profiling.
46     callbacks : list[WorkflowCallbacks] | None default=None
47         A list of callbacks to register.
48     progress_logger : ProgressLogger | None default=None
49         The progress logger.
50
51     Returns
52     -----
53     list[PipelineRunResult]
54         The list of pipeline run results
55     """
```

index.py 中执行构建索引的 API 接口核心就是在调用 index 模块中 run_pipeline 函数，本质上就是在启动 Microsoft GraphRAG 的索引构建 workflow。调用关系如下图所示：



关于 Microsoft GraphRAG 构建索引的完整 workflow 流程，在《Microsoft GraphRAG 深度实战 - Part 2. Microsoft GraphRAG 索引构建细节源码详解》中有非常详细的讲解，这里不在重复说明。我们需要做的是：基于这个 API 接口去构建 Python 的接口调用程序。因此我们在源码文件中创建了一个新的文件夹 dev，并且创建了 graphrag_indexing.py 文件，用于实现 Microsoft GraphRAG 的索引构建流程，其源码存放位置为：graphrag\dev\graphrag_indexing.py



在 graphrag_indexing.py 文件会加载一系列变量以控制索引构建过程的灵活性，其中核心配置项如下：

```
# =====  
# 用户配置参数 - 直接修改这里的变量  
# =====  
  
# GraphRAG项目根目录路径  
PROJECT_DIR = "C:\\Users\\Lenovo\\Desktop\\folder\\Agent\\code\\code\\backend\\  
deepseek_agent\\llm_backend\\app\\graphrag"
```

```
# 数据目录名称（相对于项目根目录）
DATA_DIR_NAME = "data"

# 索引方法: "Standard" 或 "Fast"
INDEX_METHOD = "Standard"

# 是否进行增量更新
IS_UPDATE = False

# 是否进行内存分析
MEMORY_PROFILE = False

# 配置文件路径（为None则使用默认配置，如果是增量更新，则需要指定配置文件，比如：
# "C:\\Users\\Lenovo\\Desktop\\folder\\Agent\\code\\code\\backend\\
# deepseek_agent\\llm_backend\\app\\graphrag\\data\\settings_csv.yaml"）
CONFIG_FILE = None

# 输出目录（为None则使用配置中的默认输出目录，）
OUTPUT_DIR = None
```

其中需要重点说明的就是：这里 `PROJECT_DIR` 是 Microsoft GraphRAG 项目的根目录，要填写服务器上实际存储的 Microsoft GraphRAG 路径。而 `DATA_DIR_NAME` 是数据目录名称，也就是你通过 `poetry run poe init --root xxx` 命令初始化时指定的目录。`IS_UPDATE` 参数为启动增量更新的配置项，大家可以暂时忽略，我们下一小节重点给大家进行介绍。这里先设置为 `False` 表示不进行增量更新。

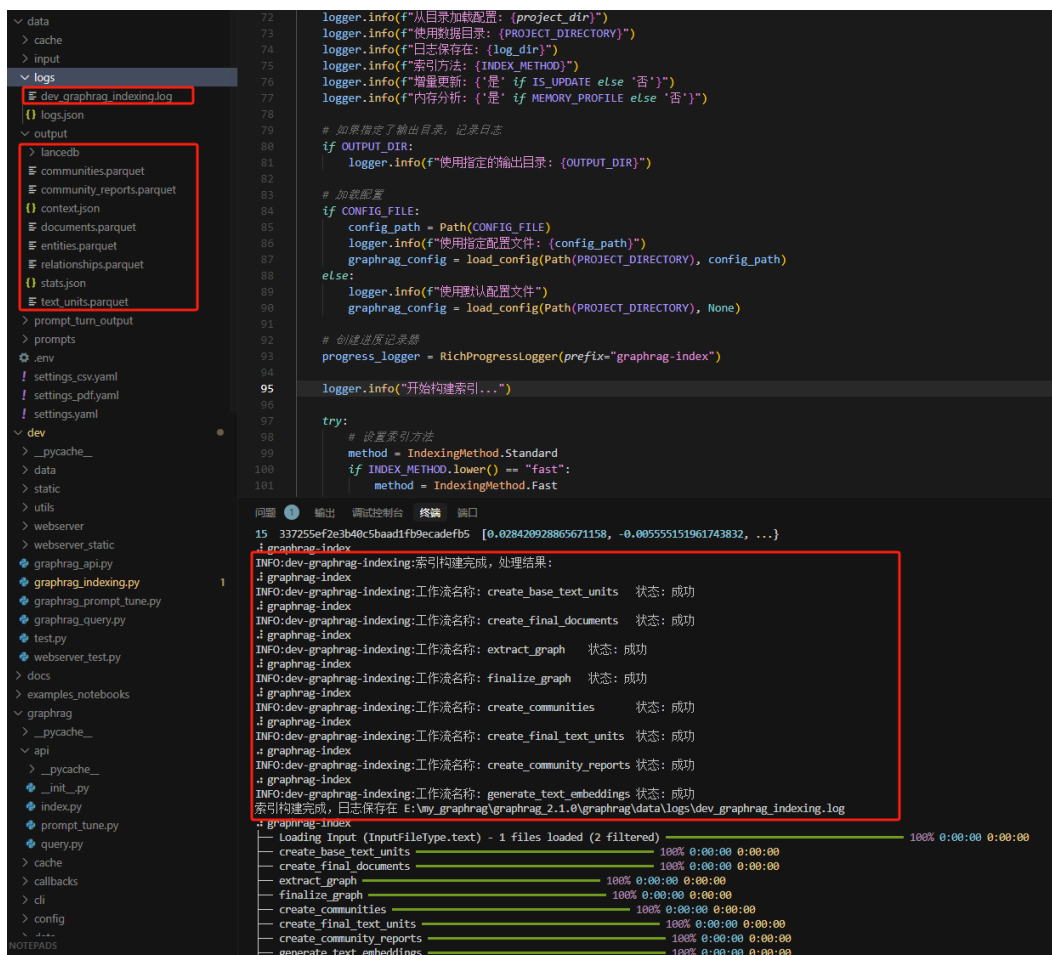
按照上述的参数规范自定义修改后，则可以直接运行该 Python 脚本，同时需要说明的是：这个执行过程加载的配置文件依然是 `settings.yaml` 文件。执行命令如下：

```
python graphrag_indexing.py
```

```
(venv) PS E:\my_graphrag\graphrag_2.1.0\graphrag\dev> python -i graphrag_indexing.py
INFO: 从目录加载配置: E:\my_graphrag\graphrag_2.1.0\graphrag
INFO: 使用数据目录: E:\my_graphrag\graphrag_2.1.0\graphrag\data
INFO: 日志库存在: E:\my_graphrag\graphrag_2.1.0\graphrag\data\logs
INFO: 索引方法: Standard
INFO: 增量更新: 否
INFO: 内存分析: 否
INFO: 使用默认配置文件
# graphrag-indexing: 开始构建索引...
Loading text files from input
Running standard indexing.
  create_base_text_units
    id                                     text                                     document_ids  n_tokens
0  e23c04e9a1fd9e2e471e6f3d6983abcc580b73527a459...  在过去的几十年中，全球科技行业经历了翻天覆地的变化。以硅谷为中心的创新生态系统催生了许多世界...  [a2c186a49102fbffae5c2453442cb5eb74fcdc6d3fe2...
1  1713c3603499bd62e4e97af383ee5f5693e60cb1360d5...  因此成为全球最有价值的公司之一。截至 2023 年，苹果的市值已超过 2 万亿美元。
2  099218da7f5c9194d623a8e4a2276af51b1b423a8d4d...  文化 (Paul Allen) 于 1979 年在英国伦敦创办了微软公司，微软最初以开发 BA...  [a2c186a49102fbffae5c2453442cb5eb74fcdc6d3fe2...  500
3  8eb9d3d07a52f7e6a5d8b8408cf5dffaeb8f394e3dc...  为亚马逊最重要的利润来源之一。此外，亚马逊还通过收购 Whole Foods 和推出 Ale...  [a2c186a49102fbffae5c2453442cb5eb74fcdc6d3fe2...  500
4  ba4c3c148d7b5f8bd7806d4fc1d3a38bcb9102f2a291...  与开源项目和技术标准的制定，推动了整个行业的发展。
  create_final_documents
    id  human_readable_id  title  ...  text_unit_ids  creation_date  metadata
0  a2c186a49102fbffae5c2453442cb5eb74fcdc6d3fe29...  1  technology_companies.txt  ...  [e23c04e9a1fd9e2e471e6f3d6983abcc580b73527a459...  2025-04-01 15:45:35 +0800  NaN

[1 rows x 7 columns]
# graphrag-index
  Loading Input (InputFileType.text) - 1 files loaded (2 filtered) 100% 0:00:00 0:00:00
  create_base_text_units 100% 0:00:00 0:00:00
  create_final_documents 100% 0:00:00 0:00:00
```

等待执行完成后，会与 `CLI` 或者 `Poetry` 命令行执行的流程一样，在输出路径下会依次生成 `.cache`、`logs` 和 `output` 三个文件夹，分别用于存储索引构建的缓存数据、日志信息和索引构建的输出结果。如下图所示：



其中，logs 文件夹中会生成 dev_graphrag_indexing.log 文件，用来记录通过 Python 代码执行索引构建的日志信息。至此，我们就可以通过 Python 代码执行 Microsoft GraphRAG 的索引构建了。正常来说，执行完这一步便可以 query 检索了。但在此之前，我们需要进一步给大家介绍下关于实际构建索引过程中遇到的常见需求及对应的解决方案。

企业常见需求一：如何构建多个索引？

多索引构建对应的开发场景大多存在于企业中存在多个数据源支撑多个业务线，彼此之间相对独立，且每个业务线之间存在一定的差异性，比如：数据格式、数据内容、数据规模等，冗余在一起会加重检索的效率和维护成本。（因为知识图谱越大，检索时需要检验的信息就越多，检索的效率就越低），亦或对不同的部门、人员之前的知识库进行隔离，避免知识库之间的相互污染等常见问题。

这类问题对应的解决方案在当前的流程中非常容易实现：即只需要在构建的时候，通过 init 的 --root 参数指定不同的数据源根目录，制定索引文件与数据源的对应关系（如通过字典构建映射关系），形式如下：

```
# 索引文件与数据源的对应关系
INDEX_FILE_MAP = {
    "index_1": "data_source_1",
    "index_2": "data_source_2",
}
```

各个索引知识库独立构建，在进行问答检索时先通过 INDEX_FILE_MAP 字典找到对应的索引文件根路径，进行加载即可实现多知识库的隔离检索。

企业常见需求二：多个不同类型的数据源，如何共同创建一个索引？

多数据源共建一个知识库的场景，在企业中也比较常见。比如关系型数据库、PDF、Word、Excel、CSV等不同类型的数据库，希望各个数据源中的有效信息可以实现整合，从而构建一个统一的知识库提供用户、或者下游的智能体使用。

回顾 Microsoft GraphRAG 的索引构建流程，筛选不同文件进入构建索引的 workflow 流程，核心是在 settings.yaml 文件中 input 参数下借助 file_type 参数指定文件类型，以及 file_parttern 通过正则表达式的形式过滤符合标准的文件，比如通过 (*.*.txt\$) 过滤所有 txt 文件。

```
> .vscode
> course
> data
  > cache
  > input
  > logs
  > dev_graphrag_indexing.log
  > logs.json
  > output
    > lancedb
    > communities.parquet
    > community_reports.parquet
    > context.json
    > documents.parquet
    > entities.parquet
    > relationships.parquet
    > stats.json
    > text_units.parquet
    > prompt_turn_output
    > prompts
    > .env
    > settings_csv.yaml
    > settings_pdf.yaml
    > settings.yaml
  > dev
  > __pycache__
  > data
  > static

model_supports_json: true # recommended if this is available for your model.
concurrent_requests: 25 # max number of simultaneous LLM requests allowed
async_mode: threaded # or asyncio
retry_strategy: native
max_retries: -1 # set to -1 for dynamic retry logic (most optimal setting)
tokens_per_minute: 0 # set to 0 to disable rate limiting
requests_per_minute: 0 # set to 0 to disable rate limiting

vector_store:
  default_vector_store:
    type: lancedb
    db_uri: output/lancedb
    container_name: default
    overwrite: True

input:
  type: file # or blob
  file_type: text
  base_dir: "input"
  file_encoding: utf-8
  file_pattern: "*.\\*.txt$"

chunks:
  size: 500
  overlap: 100
  group_by_columns: [id]

embed_text:
  model_id: default_embedding_model
  vector_store_id: default_vector_store
```

那么对于这种过滤文件的方式，大家可能想到的是：既然借助 file_type 和 file_parttern 参数可以过滤文件，那么是否可以借助 file_type 和 file_parttern 参数，将不同类型的数据源进行整合，从而构建一个统一的知识库？实际上确实可以的，比如指定 file_pattern: "*.*(.txt|csv)\$" ,即可同时读取 input 目录下所有 txt 和 csv 文件。

```
> cache
> input
  > all_text.pdf
  > merged_review.csv
  > technology_companies.txt
  > logs
  > output
    > lancedb
    > communities.parquet
    > community_reports.parquet
    > context.json
    > documents.parquet
    > entities.parquet
    > relationships.parquet
    > stats.json
    > text_units.parquet
    > prompt_turn_output
    > prompts
    > .env
    > settings_csv.yaml
    > settings_pdf.yaml
    > settings.yaml
  > dev
  > __pycache__
  > data
  > static

requests_per_minute: 0 # set to 0 to disable rate limiting

vector_store:
  default_vector_store:
    type: lancedb
    db_uri: output/lancedb
    container_name: default
    overwrite: True

input:
  type: file # or blob
  file_type: text
  base_dir: "input"
  file_encoding: utf-8
  file_pattern: "*.\\*(.txt|csv)$"

chunks:
  size: 500
  overlap: 100
  group_by_columns: [id]

embed_text:
  model_id: default_embedding_model
```

INFO: 使用默认配置文件
graphrag.indexing: 开始构建索引...
Loading text files from input
Running standard indexing
create_base_text_units

id	text	document_id	n_tokens
0	e23046a1f8e2471e6f3a683b0c5080732274580...	merged_review.csv	500
1	171138614936024e47af38e5f56993e0e1313085...	technology_companies.txt	500
2	0302184fffc02194023aee43274451b3a324d4d...	merged_review.csv	500
3	8a0d43872ff7e0bca0b0408c3f4faab8f39a3d3c...	technology_companies.txt	500
4	8a0d43872ff7e0bca0b0408c3f4faab8f39a3d3c...	merged_review.csv	500
5	0e0c4c964076807f6ad18c3853d9b0d6f3118134...	technology_companies.txt	304
6	8b3a20c43096c219f084f23844c48b441c75095...	merged_review.csv	500
7	4f0e0a1c1a7796427a043a3b04f4e4a14e4a1225...	technology_companies.txt	500
8	0a0031336c9d3c18c3a057ef4e0a360803d8aa...	merged_review.csv	500
9	760cfc4d9596a037721a0b5b0c49572770c27314...	technology_companies.txt	500
10	0302184fffc02194023aee43274451b3a324d4d...	merged_review.csv	500
11	fac6e6159516216e269914d07b0d9306c538145...	technology_companies.txt	500
12	0e0c4c964076807f6ad18c3853d9b0d6f3118134...	merged_review.csv	500
13	0e0c4c964076807f6ad18c3853d9b0d6f3118134...	technology_companies.txt	500
14	ed5a7f614131120496358f3f1ade435a0a20c1e...	merged_review.csv	500
15	20a4a0b21131020180640e1154a0f14c4d108...	technology_companies.txt	500
16	cd0e0d720c40805d16f210952408d1713474d87...	merged_review.csv	500
17	0a0031336c9d3c18c3a057ef4e0a360803d8aa...	technology_companies.txt	500
18	1c7945d4dd4d7b0c7f2c32608d7240456f92c5c...	merged_review.csv	455
19	4a9_V6181_2.0_2024-01-04_288_智能电视_西门...	merged_review.csv	55

id	human_readable_id	title	test_unit_ids	creation_time	metadata
0	merged_review.csv	merged_review.csv	0e0c4c964076807f6ad18c3853d9b0d6f3118134...	2025-03-20 18:31:14	48800
1	technology_companies.txt	technology_companies.txt	e23046a1f8e2471e6f3a683b0c5080732274580...	2025-03-01 13:43:31	48800

[2 rows x 7 columns]
graphrag.index

从执行流程上看是可以，但是存在一个非常严重的问题是：无法指定不同文件类型加载对应的优化切分策略。也就是说，input 可以通过 file_type 和 file_parttern 参数指定文件类型和文件过滤规则，但是 chunk 下的切分策略，将会用于所有文件的切分，也就是说无法实现不同文件类型加载对应的优化切分策略。比如我们针对不同的文件格式，定义了优化后的切分策略，配置如下所示：

```
input:
  type: file # or blob
  file_type: csv
```

```

base_dir: "input"
file_encoding: utf-8
file_pattern: ".*\\.csv$$"
text_column: "text" # 如果原始数据中没有text字段，则需要指定一个包含文本内容的字段

chunks:
strategy: csv # 自定义参数
size: 500
overlap: 100
group_by_columns: [id]

input:
type: file # or blob
file_type: pdf # or csv
base_dir: "input"
file_encoding: utf-8
file_pattern: ".*\\.pdf$$"
local_output_dir: "./data/pdf_outputs" # 自定义参数：添加PDF输出目录配置
mineru_api_url: "http://192.168.110.131:8000/"
mineru_output_dir: "/home/07_minerU/tmp/"
table_description_api_key: ""
table_description_model: "deepseek-chat"
base_url: "https://api.deepseek.com"

image_description_api_key: ""
image_description_model: "gpt-4o"
image_description_base_url: "https://ai.devtool.tech/proxy/v1"

chunks:
strategy: markdown # 自定义参数
size: 500
overlap: 100
group_by_columns: [id]

```

在现有流程上，是不能实现对文件格式和对应的切分策略进行灵活匹配的。因此，这里能得出的结论是：如果大家对所有格式的文件能够容忍使用同一种最简单的切分策略，那么可以直接用 `file_pattern` 进行文件过滤。而如果希望针对不同文件类型，加载对应的优化切分策略，则需要引入增量更新的解决方案。

增量更新 (Incremental Update) 指的是在已有知识图谱的基础上，快速更新新加入的知识，避免全量重建。一些核心的增量更新的实现方法是：

1. **文档级增量更新**：这类方法会对文本内容生成MD5哈希值，在实现增量插入功能时，先检查是否存在相同内容，如果判定为新内容，则重新计算社区结构并生成新的社区报告。代表项目是 `nano-GraphRAG`。
2. **知识库级增量更新**：这类方法会使用图结构管理实体和关系，每次有新文档加入时，会对其执行相同的图索引步骤，包括实体识别和关系提取。通过将新增的节点和边集与原有图的节点和边进行合并（集合并操作），实现新数据的无缝集成，代表项目是 `LightRAG`。

`Microsoft GraphRAG` 在 v0.4.0 版本中引入了增量索引支持，其实现的方法属于**知识库级增量更新**。在进行原理讲解之前，我们可以通过 `Poetry` 命令直接看一下 `Microsoft GraphRAG` 的增量索引的执行方法，如下所示，对增量更新，需要执行的命令为 `poetry run poe update`，可以通过 `--help` 参数查看具体的执行方法，如下所示：

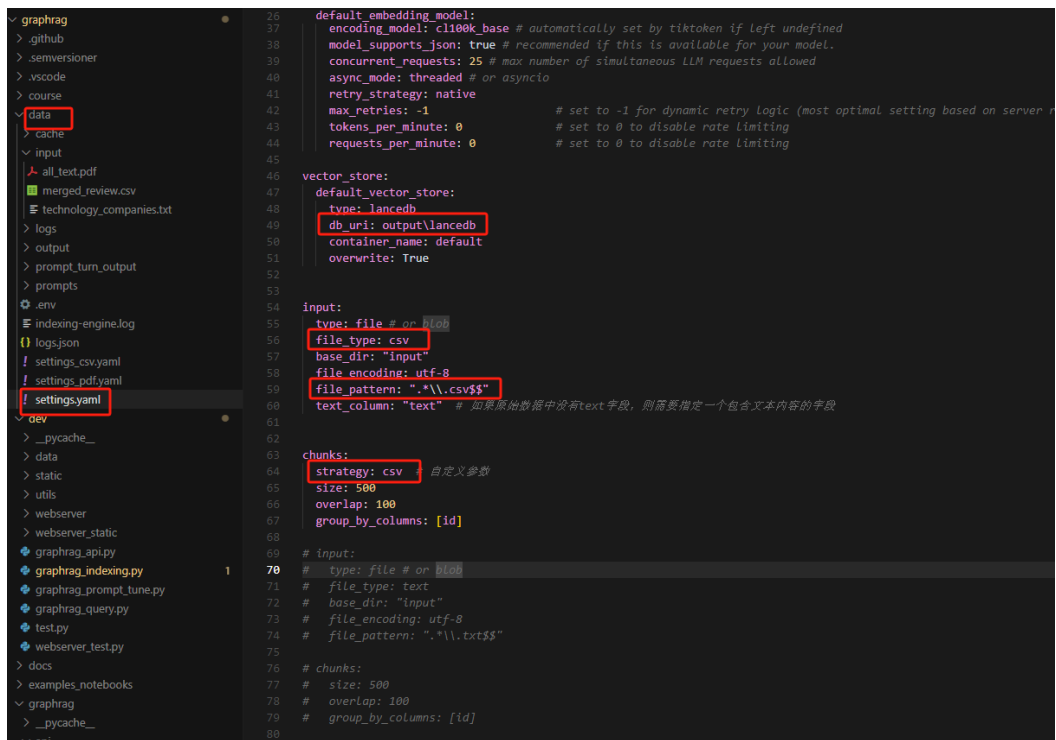
增量更新参数说明

参数名	说明	默认值
<code>--config</code>	要使用的配置文件路径。	<code>None</code>
<code>--root</code>	项目的根目录。	<code>.</code>
<code>--method</code>	要使用的索引方法。可选值为 <code>standard</code> 或 <code>fast</code> 。	<code>standard</code>
<code>--verbose</code>	以详细日志运行索引管道。	<code>no-verbose</code>
<code>--memprofile</code>	运行索引管道时进行内存分析。	<code>no-memprofile</code>
<code>--logger</code>	要使用的进度日志记录器。可选值为 <code>rich</code> 、 <code>print</code> 或 <code>none</code> 。	<code>rich</code>
<code>--cache</code>	使用LLM缓存。	<code>cache</code>
<code>--skip-validation</code>	跳过任何预检验证。适用于不运行LLM步骤时。	<code>no-skip-validation</code>
<code>--output</code>	索引管道输出目录。覆盖配置文件中的 <code>output.base_dir</code> 。	<code>None</code>
<code>--help</code>	显示此帮助信息并退出。	

既然是增量更新，那么就意味着需要将新文件生成的索引文件，与已经存在的某一个图谱进行合并，因此，在执行 `poetry run poe update` 命令时的核心参数为：

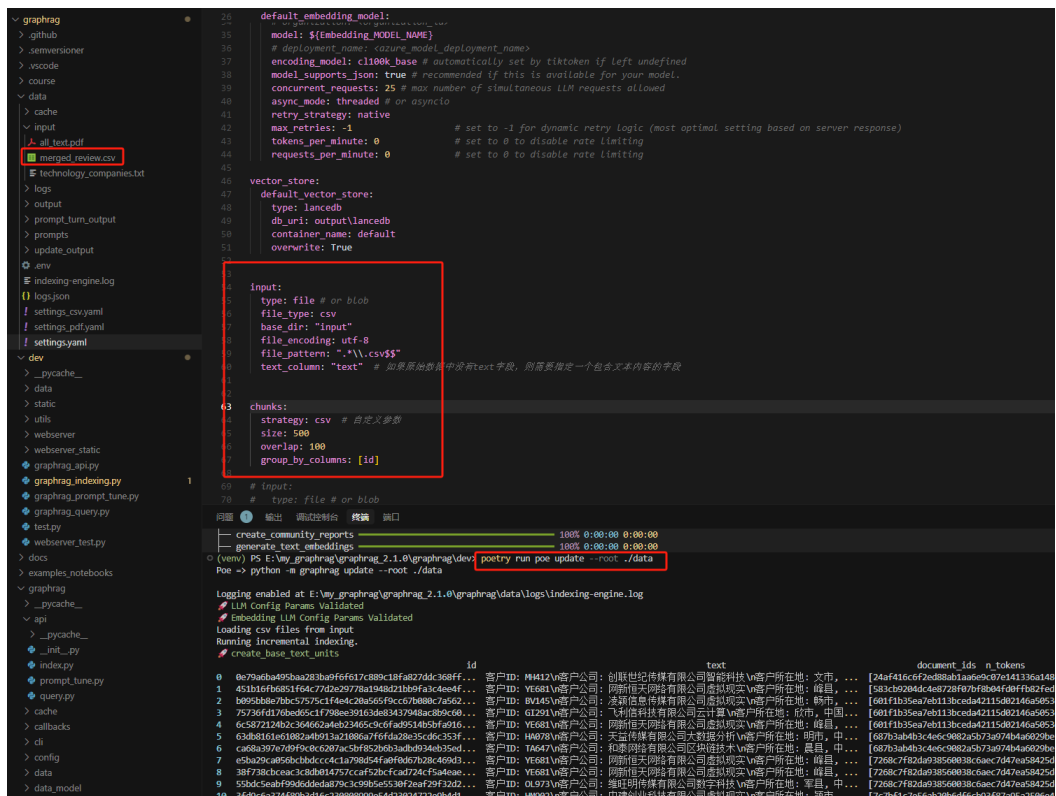
1. `--config`: 指定新文件的默认加载配置，加载这个 `settings.yaml` 做文件切分等策略；
2. `--output`: 指的是要和哪一个已经存在的图谱进行合并，如果不指定，就是 `--root` 文件夹中对应的 `output` 下面的 `.parquet` 文件；
3. `--vector_store`: 这个参数并没有在命令行参数中体现，是实际运行时总结出的一个关键点，即：`db_uri` 要与 `--output` 指定的图谱路径一致，否则会加载不到新合并的实体 `Embedding`，导致合并后检索异常；

因此，我们这里可以通过 `poetry run poe update` 命令，实践一下增量更新的执行流程。接下来我们将 `input` 目录下的 `csv` 文件，与刚刚对 `txt` 文件构建的索引进行合并。首先，修改 `settings_csv.yaml` 文件，让其加载 `input` 目录下的 `csv` 文件，并使用 `csv` 的切分策略，与上一小节对 `txt` 文件构建的索引进行合并。配置修改如下所示：



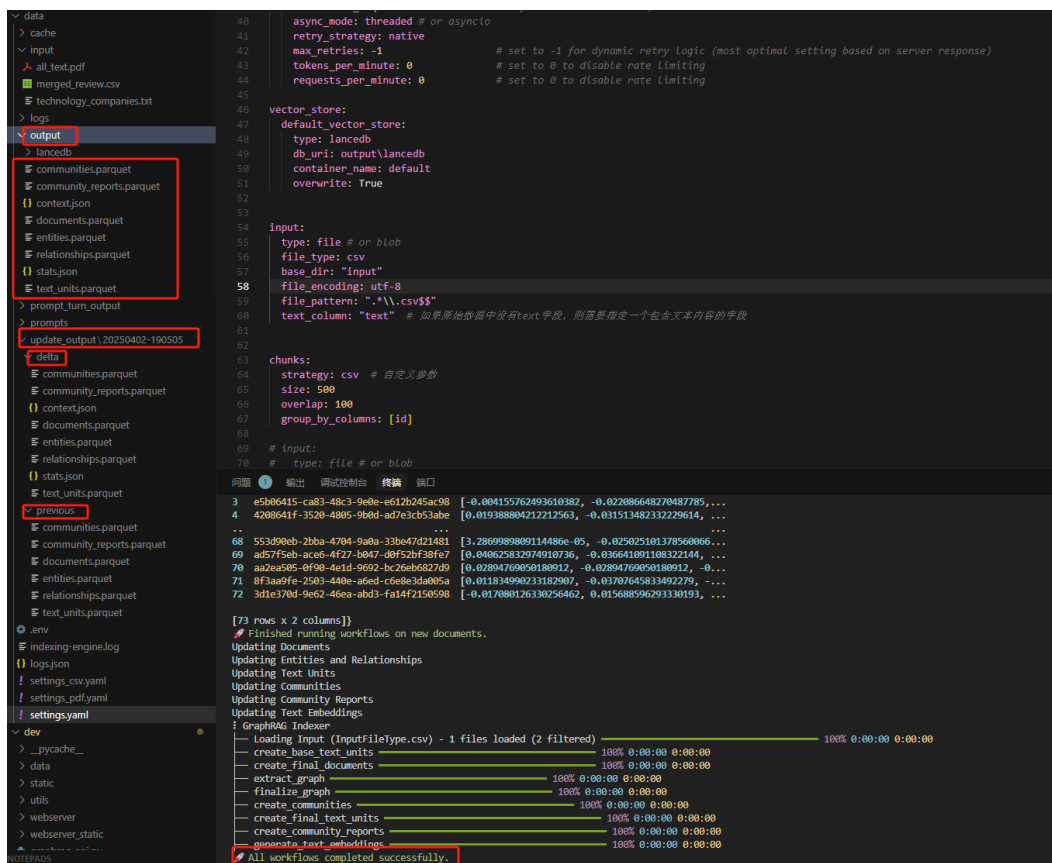
然后执行增量更新流程，执行命令如下，耐心等待执行结束即可。

```
poetry run poe update --root ./data
```



当增量更新执行结束后，发生的改动主要体现在：

1. 旧的索引文件会进行更新，融入新文件产生的实体、关系、描述等信息，因此可以看到就得索引的日期会发生变化，标志着索引文件已经更新。
2. 当指定的根路径下（我们使用的是 ./data），会生成一个 `update_output` 文件夹，同时以时间戳命名的文件夹，该文件夹中里面会包含 `delta` 和 `previous` 两个子文件夹，分别用于存储新文件产生的实体、关系、描述等信息，以及旧文件的备份。

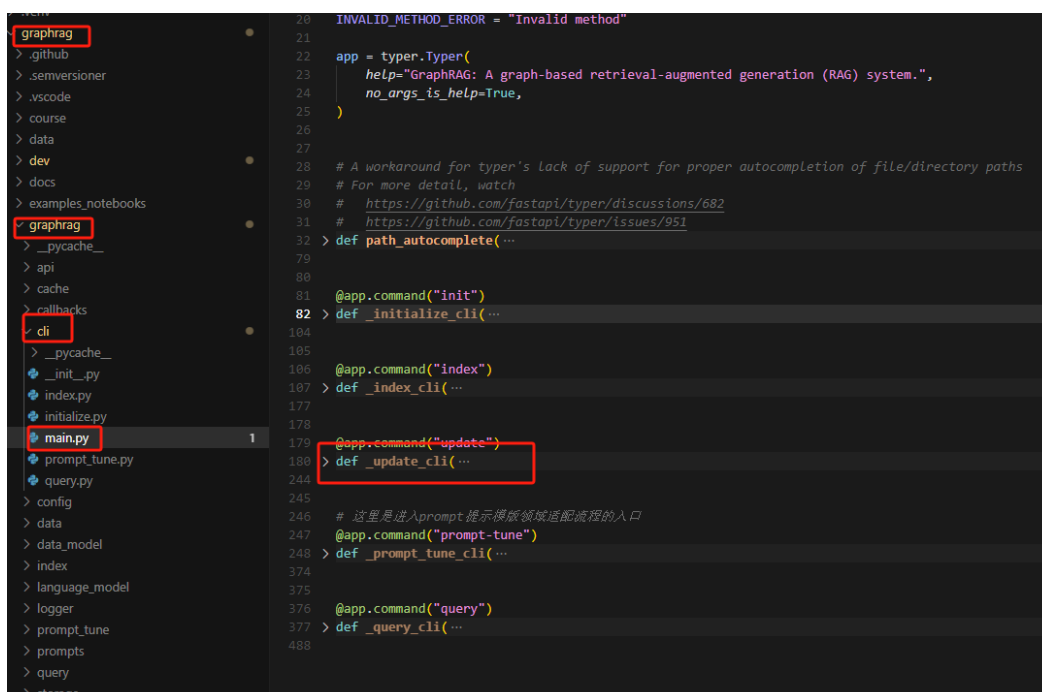


```
40 async_mode: threaded # or asyncio
41 retry_strategy: native
42 max_retries: -1 # set to -1 for dynamic retry logic (most optimal setting based on server response)
43 tokens_per_minute: 0 # set to 0 to disable rate limiting
44 requests_per_minute: 0 # set to 0 to disable rate limiting
45
46
47 vector_store:
48   default_vector_store:
49     type: lancedb
50     db_uri: output/lancedb
51     container_name: default
52     overwrite: True
53
54 input:
55   type: file # or blob
56   file_type: csv
57   base_dir: "input"
58   file_encoding: utf-8
59   file_pattern: ".*\\.csv$"
60   text_column: "text" # 如果原始数据中没有text字段，则需要指定一个包含文本内容的字段
61
62
63 chunks:
64   strategy: csv # 自定义参数
65   size: 500
66   overlap: 100
67   group_by_columns: [id]
68
69 # input:
70 # type: file # or blob
71
72 问题 输出 调试控制台 终端 窗口
73
74 3 e5b66415-ca83-48c3-9e0e-e612b245ac98 [-0.004155762493618382, -0.022086648270487785, ...
75 4 4208641f-3528-4805-9b8d-ad7e3cb53abe [0.019388804212212563, -0.03151348233229614, ...
76 ...
77 68 553d90eb-2bba-4704-9a0a-33be47d21481 [3.2869989809114486e-05, -0.02502510137856006, ...
78 69 ad57f5eb-ace6-4f27-b047-d0f52bf38fe7 [0.040625832974919736, -0.036641991108322144, ...
79 70 aa2ea505-0f90-4e1d-9692-bc26eb6827d9 [0.02894769050180912, -0.02894769050180912, -0...
80 71 8f3aa9fe-2503-440e-a6ed-c6e8e3da005a [0.011834990233182907, -0.03707645833492279, ...
81 72 3d1e378d-9e62-46ea-abd3-fa14f2150598 [-0.017080126330256462, 0.01568859629330193, ...
82
83 [73 rows x 2 columns]]
84
85 Finished running workflows on new documents.
86 Updating Documents
87 Updating Entities and Relationships
88 Updating Text Units
89 Updating Communities
90 Updating Community Reports
91 Updating Text Embeddings
92
93 GraphRAG Indexer
94 Loading Input (InputFileType.csv) - 1 files loaded (2 filtered) 100% 0:00:00 0:00:00
95 create_base_text_units 100% 0:00:00 0:00:00
96 create_final_documents 100% 0:00:00 0:00:00
97 extract_graph 100% 0:00:00 0:00:00
98 finalize_graph 100% 0:00:00 0:00:00
99 create_communities 100% 0:00:00 0:00:00
100 create_final_text_units 100% 0:00:00 0:00:00
101 create_community_reports 100% 0:00:00 0:00:00
102 generate_text_embeddings 100% 0:00:00 0:00:00
103
104 All workflows completed successfully.
```

至此，整个增量更新就已经执行结束了。该过程已经成功的将 `merged_review.csv` 文件中的内容，与第一次构建 `technology_companies.txt` 文件的索引进行了合并，其存储的索引文件依然是 `output` 文件夹下的所有 `.parquet` 文件。整个过程在命令行的操作形式下非常简单，只需要修改 `settings.yaml` 文件，指定新的数据源，然后执行 `poetry run poe update` 命令即可。而对于如果我们想将其封装成 `Python` 接口，则需要进一步了解其背后的实现原理。因为在 `Microsoft GraphRAG` 的 `api` 接口中，仅仅提供了 `prompt_tune.py`、`query_api.py` 和 `index.py` 三个文件，并没有提供增量更新的接口，同时对于增量更新后新生成的索引文件应如何使用，只有通过进一步的源码分析才能够真正理解其应用的方法。

因此接下来，我们就详细解读 `Microsoft GraphRAG` 中实现增量更新的完整底层逻辑，以支撑接下来通过 `Python` 接口的开发实现增量更新功能的实际需求。

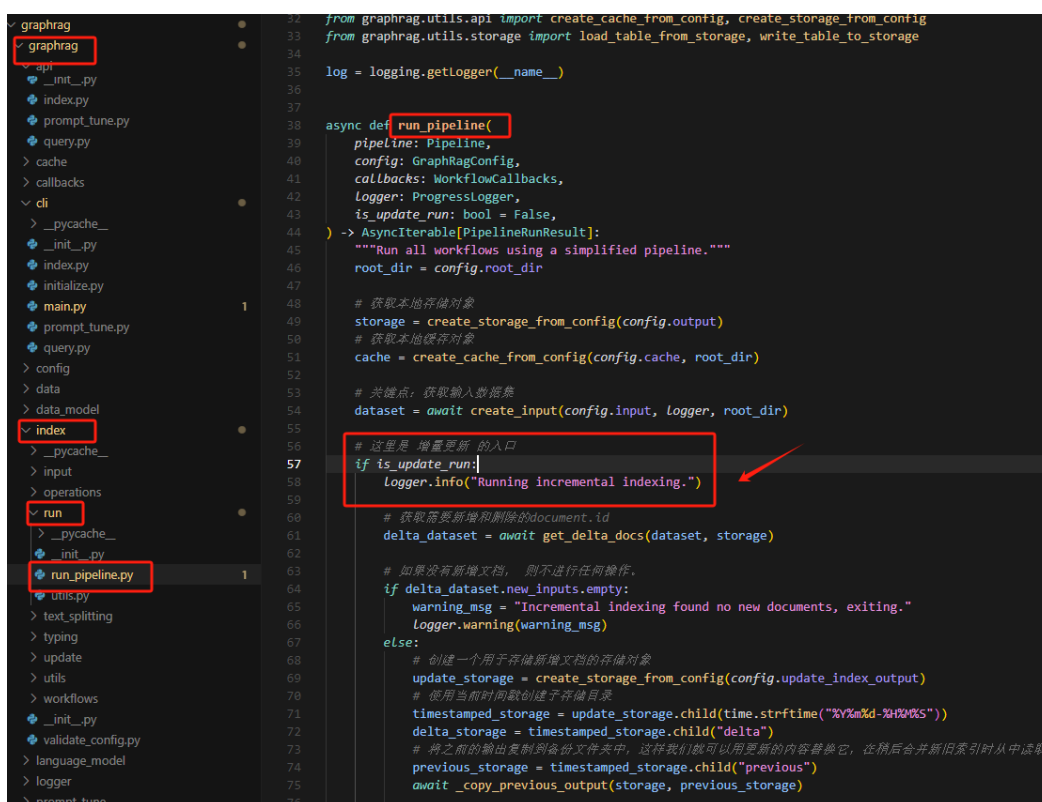
既然可以通过 `Poetry run poe update` 命令行执行增量更新，那么就意味着在 `Microsoft GraphRAG` 的 `CLI` 中，一定存在对应的实现方法。因此接下来，我们需要从 `Microsoft GraphRAG` 的 `CLI` 入手，源码位置为：`graphrag\graphrag\cli\main.py`



```
graphrag
> .github
> .semversioner
> .vscode
> course
> data
> dev
> docs
> examples notebooks
> graphrag
> __pycache__
> api
> cache
> callbacks
> cli
> __pycache__
> __init__.py
> index.py
> initialize.py
> main.py
> prompt_tune.py
> query.py
> config
> data
> data_model
> index
> language_model
> logger
> prompt_tune
> prompts
> query
> storage
```

```
20 INVALID_METHOD_ERROR = "Invalid method"
21
22 app = typer.Typer(
23     help="GraphRAG: A graph-based retrieval-augmented generation (RAG) system.",
24     no_args_is_help=True,
25 )
26
27 # A workaround for typer's lack of support for proper autocompletion of file/directory paths
28 # For more detail, watch
29 # https://github.com/fastapi/typer/discussions/682
30 # https://github.com/fastapi/typer/issues/951
31
32 > def path_autocomplete(...
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

在 `_update_cli` 函数中，`update_cli` 函数是用于执行增量更新的入口函数，根据执行逻辑，其本质上执行的还是 `index` 的流程，只不过在执行的过程中，会根据增量更新的需求，进行一些特殊处理。路由逻辑如下图所示：



```
graphrag
> graphrag
> api
> __init__.py
> index.py
> prompt_tune.py
> query.py
> cache
> callbacks
> cli
> __pycache__
> __init__.py
> index.py
> initialize.py
> main.py
> prompt_tune.py
> query.py
> config
> data
> data_model
> index
> __pycache__
> input
> operations
> run
> __pycache__
> __init__.py
> run_pipeline.py
> utils.py
> text_splitting
> typing
> update
> utils
> workflows
> __init__.py
> validate_config.py
> language_model
> logger
> prompt_tune
```

```
32 from graphrag.utils.api import create_cache_from_config, create_storage_from_config
33 from graphrag.utils.storage import load_table_from_storage, write_table_to_storage
34
35 log = logging.getLogger(__name__)
36
37
38 async def run_pipeline(
39     pipeline: Pipeline,
40     config: GraphRagConfig,
41     callbacks: WorkflowCallbacks,
42     logger: ProgressLogger,
43     is_update_run: bool = False,
44 ) -> AsyncIterable[PipelineRunResult]:
45     """Run all workflows using a simplified pipeline."""
46     root_dir = config.root_dir
47
48     # 获取本地存储对象
49     storage = create_storage_from_config(config.output)
50     # 获取本地缓存对象
51     cache = create_cache_from_config(config.cache, root_dir)
52
53     # 关键点：获取输入数据集
54     dataset = await create_input(config.input, logger, root_dir)
55
56     # 这里是 增量更新 的入口
57     if is_update_run:
58         logger.info("Running incremental indexing.")
59
60     # 获取需要新增和删除的document.id
61     delta_dataset = await get_delta_docs(dataset, storage)
62
63     # 如果没有新增文档，则不进行任何操作。
64     if delta_dataset.new_inputs.empty:
65         warning_msg = "Incremental indexing found no new documents, exiting."
66         logger.warning(warning_msg)
67     else:
68         # 创建一个用于存储新增文档的存储对象
69         update_storage = create_storage_from_config(config.update_index_output)
70         # 使用当前时间戳创建子存储目录
71         timestamped_storage = update_storage.child(time.strftime("%Y%m%d-%H%M%S"))
72         delta_storage = timestamped_storage.child("delta")
73         # 将之前的输出复制到备份文件夹中，这样我们就可以用更新的内容替换它，在随后合并新旧索引时从中读取
74         previous_storage = timestamped_storage.child("previous")
75         await _copy_previous_output(storage, previous_storage)
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

因此增量更新的逻辑是：通过 `is_update_run` 参数，判断是否执行增量更新，如果是 `True` 则执行增量更新，否则执行正常的索引构建流程。其中增量更新的具体流程为：

1. **加载用于增量更新的文件**：根据在 `input` 中存放的文件，通过 `create_input` 函数按照 `settings.yaml` 中配置的文档类型进行加载；
2. **读取已存在的 `documents.parquet` 文件**：获取之前所有已经执行过索引的文件信息，返回一个 `DataFrame` 对象，包括 `id`、`title`、`text` 和 `creation_date` 等字段；
3. **对比增量更新文件与已存在的 `documents.parquet` 文件，具体的检索方法为**：
 - 先根据 `title` 去重 原始文档，得到一个 `Dataframe`；

- 再根据 `title` 去重 新增文档，得到一个 `Dataframe`；
- 检查新增文档中都有哪些是在原始文档中的， 返回一个包含所有新增文档的 `DataFrame`；
- 检查原始索引文件中的每个标题是否在新数据集中， 返回一个包含所有需要删除文档的 `DataFrame`；

这个预处理的流程本质上就是在已有的文档索引中添加新的文档，同时删除不再需要的文档。而根据源码的解析，能够判断出来的一个关键点是：**增量更新对不同文件的识别是根据 `title` 字段进行的，而 `title` 字段对应的是 `input` 目录下文件的名称，所以如果你要增量更新的文档名称与原文档是一样的话，即使内容发生了改变，也不会触发增量更新，因此一定要确保要执行的增量更新文档名称要进行修改，最好的一种方式就是以时间戳命名，比如 `merged_review_20250403.csv`。**

接下来，当识别到了新增的文档后继续执行的流程为：

4. 创建一个用于存储新增文档的存储对象；
5. 创建带时间戳的子存储，（即创建本地文件夹），包含两次操作：
 - 在时间戳目录下创建 `delta` 子目录，用于存储新增的文档数据
 - 在时间戳目录下创建 `previous` 子目录，用于存储原有数据的备份，包含所有的 `.parquet` 文件
6. 将新文档的 `storage` 和 `input`，进入 Microsoft GraphRAG 的 Indexing 的 `Workflow`，先对新文档正常执行索引构建流程，得到新文档对应的索引文件；

```

graphrag
├── api
│   ├── __init__.py
│   ├── index.py
│   ├── prompt_tune.py
│   ├── query.py
│   ├── cache
│   ├── callbacks
│   └── di
│       ├── __pycache__
│       ├── __init__.py
│       ├── index.py
│       ├── initialize.py
│       ├── main.py
│       ├── prompt_tune.py
│       ├── query.py
│       ├── config
│       ├── data
│       └── data_model
├── index
│   ├── __pycache__
│   ├── input
│   ├── operations
│   └── run
│       ├── __pycache__
│       ├── __init__.py
│       ├── run_pipeline.py
│       ├── utils.py
│       ├── text_splitting
│       ├── typing
│       └── update
└── __pycache__

```

```

54 dataset = await create_input(config.input, Logger, root_dir)
55
56 # 这里是增量更新的入口
57 if is_update_run:
58     Logger.info("Running incremental indexing.")
59
60 # 4. 获取需要新增和删除的 document.id
61 delta_dataset = await get_delta_docs(dataset, storage)
62
63 # 如果没有新增文档，则不进行任何操作。
64 if delta_dataset.new_inputs.empty:
65     warning_msg = "Incremental indexing found no new documents, exiting."
66     Logger.warning(warning_msg)
67 else:
68     # 创建一个用于存储新增文档的存储对象
69     update_storage = create_storage_from_config(config.update_index_output)
70     # 使用当前时间戳创建子存储目录
71     timestamped_storage = update_storage.child(time.strftime("%Y%m%d-%H%M%S"))
72     delta_storage = timestamped_storage.child("delta")
73     # 将之前的输出复制到备份文件夹中，这样我们就可以用更新的内容替换它，在随后合并新旧索引时从中读取
74     previous_storage = timestamped_storage.child("previous")
75     await _copy_previous_output(storage, previous_storage)
76
77 Ctrl+L to chat, Ctrl+K to generate
78
79 #
80 async for table in _run_pipeline(
81     pipeline=pipeline,
82     config=config,
83     dataset=delta_dataset.new_inputs,
84     cache=cache,
85     storage=delta_storage,
86     callbacks=callbacks,
87     Logger=Logger,
88 ):
89     yield table
90
91 Logger.success("Finished running workflows on new documents.")

```

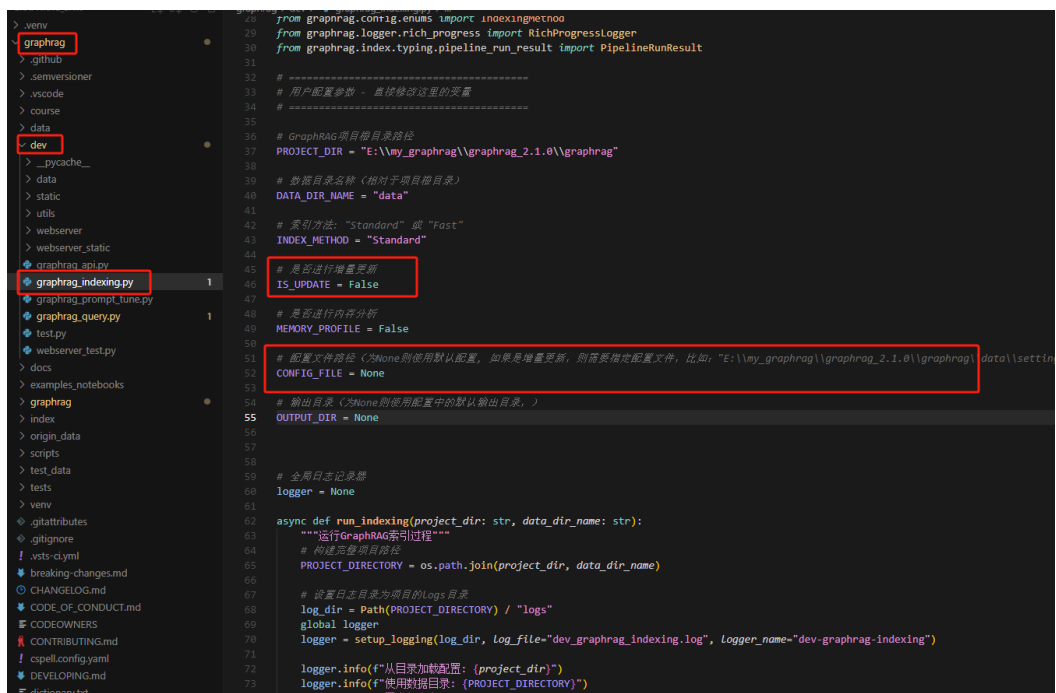
这一步所做的事情就和正常 `poetry run poe index` 构建文件的索引没有任何的区别。我们对 `Index` 在《Microsoft GraphRAG 深度实战 - Part 2. Microsoft GraphRAG 索引构建细节源码详解》中有详细的介绍，这里就不再赘述了。这里需要注意的是，对于增量更新来说，其本质是把新文件的索引文件增加到已有的索引文件中，所以在执行结束后，会进行索引合并，如下图所示：

14. 合并新旧社区及父社区的引用；
15. 合并社区报告；
16. 合并文本嵌入；

综上一共16个步骤进行增量更新，而从其源码的实现来看，Microsoft GraphRAG 的增量更新是属于 knowledge Graph 的增量更新。

至此，我们已经了解了 Microsoft GraphRAG 的增量更新流程，得到的一个重要结论就是：**增量更新只是 index 流程中通过 is_update_run 参数进行控制的一个分支**。因此接下来，我们再来看增量更新在 Python 接口中的实现就能够非常容易理解。

现在回到我们自定义的 generate_index 函数，在 index 流程中，会根据 IS_UPDATE 参数，判断是否执行增量更新，如果执行增量更新，则同时配置新索引对应的配置文件，即 settings_update.yaml 文件，即可快速实现代码调用层面的增量更新流程，源码如下图所示：

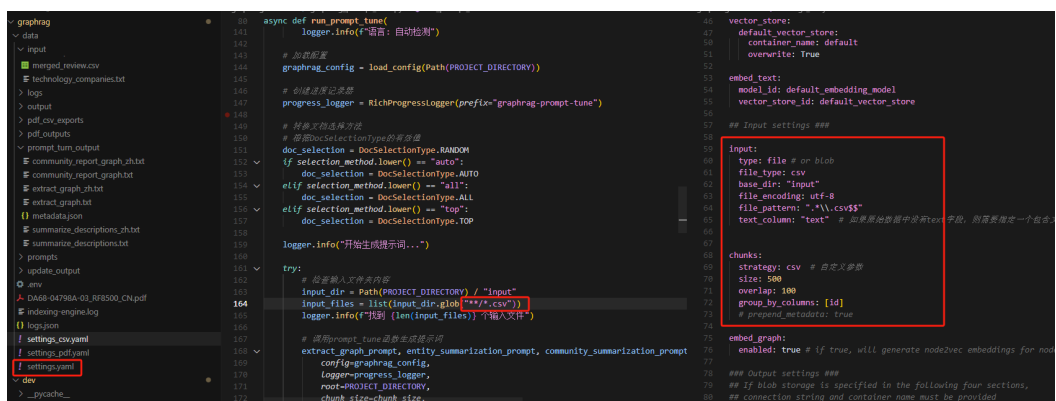


```
26 from graphrag.config.enums import IndexingMethod
27 from graphrag.logger.rich_progress import RichProgressLogger
28 from graphrag.index.typing.pipeline_run_result import PipelineRunResult
29
30 # =====
31 # 用户配置参数 - 直接修改这里的变量
32 # =====
33
34 # GraphRAG 项目根目录路径
35 PROJECT_DIR = "E:\\my_graphrag\\graphrag_2.1.0\\graphrag"
36
37 # 数据目录名称 (相对于项目根目录)
38 DATA_DIR_NAME = "data"
39
40 # 索引方法: "Standard" 或 "Fast"
41 INDEX_METHOD = "Standard"
42
43 # 是否进行增量更新
44 IS_UPDATE = False
45
46 # 是否进行内存分析
47 MEMORY_PROFILE = False
48
49 # 配置文件路径 (<None>则使用默认配置。如果是增量更新，则需要指定配置文件，比如: "E:\\my_graphrag\\graphrag_2.1.0\\graphrag\\data\\settings_update.yaml")
50 CONFIG_FILE = None
51
52 # 输出目录 (<None>则使用配置中的默认输出目录。)
53 OUTPUT_DIR = None
54
55 # 全局日志记录器
56 logger = None
57
58 async def run_indexing(project_dir: str, data_dir_name: str):
59     """运行GraphRAG索引过程"""
60     # 构建索引目录路径
61     PROJECT_DIRECTORY = os.path.join(project_dir, data_dir_name)
62
63     # 设置日志目录为项目的Logs目录
64     log_dir = Path(PROJECT_DIRECTORY) / "logs"
65     global logger
66     logger = setup_logging(log_dir, log_file="dev_graphrag_indexing.log", logger_name="dev-graphrag-indexing")
67
68     logger.info(f"从目录加载配置: {project_dir}")
69     logger.info(f"使用数据目录: {PROJECT_DIRECTORY}")
70     logger.info(f"索引方法: {INDEX_METHOD}")
71
72     vector_store:
73     default_vector_store:
74     container_name: default
75     overwrite: True
76
77     embed_text:
78     model_id: default_embedding_model
79     vector_store_id: default_vector_store
80
81     ## Input settings ##
82     input:
83     type: file # on blob
84     file_type: csv
85     base_dir: "input"
86     file_encoding: utf-8
87     file_pattern: ".\\.*\\.csv$"
88     text_column: "text" # 如果原始数据中没有text字段，则需要指定一个包含...
89
90     chunks:
91     strategy: csv # 自定义策略
92     size: 500
93     overlap: 100
94     group_by_columns: [id]
95     # prepping_metadata: true
96
97     embed_graph:
98     enabled: true # If true, will generate node2vec embeddings for node
99
100     ## Output settings ##
101     # If blob storage is specified in the following four sections,
102     # connection_string and container_name must be provided
```

这里我们可以进行实际的调用测试。其中需要修改的配置参数为：

1. generate_indexing.py 文件中的 IS_UPDATE 参数，设置为 True；
2. generate_indexing.py 文件中的 CONFIG_FILE 参数，设置为新增文件索引对应执行的 settings_update.yaml 配置；（如下我们使用的是 csv 文件，所以配置文件为 settings_csv.yaml）
3. 正常配置 settings_pdf.yaml 文件；

同样，在进行增量更新时，仍然需要使用 Auto Prompt Tuning 进行 prompt 的自动优化，修改配置如下：



```
164 input_dir = Path(PROJECT_DIRECTORY) / "input"
165 input_files = list(input_dir.glob("**/*.csv"))
166 logger.info(f"找到 {len(input_files)} 个输入文件")
167
168 # 使用 prompt_tune 函数进行提示词优化
169 extract_graph_prompt, entity_summarization_prompt, community_summarization_prompt
170 config_graphrag_config,
171 logger_progress_logger,
172 root=PROJECT_DIRECTORY,
173 chunk_size=chunk_size,
```

同时需要注意：如果使用的是 MinerU 服务解析 PDF 文件，需要先进行 MinerU 服务的启动，否则会报错。完成上述修改后，直接在命令行执行 `python generate_prompt_turn.py` 命令，执行 Auto Prompt Tuning 流程，得到优化后的 `prompt` 文件。然后根据实际的情况进行 `settings_csv.yaml` 文件中对应提示模版的修改，这里不再重复演示。

最后，对 `generate_indexing.py` 文件进行修改，配置 `IS_UPDATE` 参数，设置为 `True`，指定 `CONFIG_FILE` 参数，设置为新增文件索引对应执行的 `settings_update.yaml` 配置完成上述修改后，直接在命令行终端执行 `python generate_indexing.py` 命令，即可实现增量更新。如下代码所示：

```
graphrag
├── .github
├── .semversioner
├── .vscode
├── course
├── data
├── cache
├── input
├── .Sail_test.pdf
├── merged_review.csv
├── technology_companies.txt
├── logs
├── output
├── pdf_csv_exports
├── pdf_outputs
├── prompt_turn_output
├── community_report_graph_zh.txt
├── community_report_graph.txt
├── extract_graph_zh.txt
├── extract_graph.txt
├── metadata.json
├── summarize_descriptions_zh.txt
├── summarize_descriptions.txt
├── prompts
├── update_output
├── env
├── DA68-04798A-03_RF8500_CN.pdf
├── indexing-engine.log
├── logs.json
├── settings_csv.yaml
├── settings_pdf.yaml
├── settings.yaml
├── dev
├── .pycache_
├── data
├── INFO:
├── utils
├── webserver
├── webserver_status
├── graphrag_api.py
├── graphrag_indexing.py
├── graphrag_prompt_tune.py
├── graphrag_query.py
├── test.py
├── webserver_test.py
├── docs
├── oacenes_embeddings
├── from graphrag.config.enums import IndexingMethod
├── from graphrag.logger.rich_progress import RichProgressLogger
├── from graphrag.index.typing.pipeline_run_result import PipelineRunResult
├── # =====
├── # 用户配置参数，直接修改这里的变量
├── # =====
├── # GraphRAG项目根目录路径
├── PROJECT_DIR = "E:\\my_graphrag\\graphrag"
├── # 数据目录名称（相对于项目根目录）
├── DATA_DIR_NAME = "data"
├── # 索引方法: "Standard" 或 "Fast"
├── INDEX_METHOD = "Standard"
├── # 是否打印调试信息
├── IS_UPDATE = True
├── # 是否进行内存分析
├── MEMORY_PROFILE = False
├── # 配置日志路径，CHANGE 则需同时配置 日志库配置项，则需同时配置日志库，如用 "E:\\my_graphrag\\graphrag_2.1.0\\graphrag\\data\\set
├── CONFIG_FILE = "E:\\my_graphrag\\graphrag_2.1.0\\graphrag\\data\\settings_csv.yaml"
├── # 输出目录（为None则使用配置中的默认输出目录。）
├── OUTPUT_DIR = None
├── # 全局日志记录器
├── config_overrides = {}
├── 问题 输出 调试控制台 终端 窗口
├── create_final_documents 100% 0:00:00 0:00:00
├── (venv) PS E:\my_graphrag\graphrag_2.1.0\graphrag\dev> python .\graphrag_indexing.py
├── INFO: 从目录加载配置: E:\my_graphrag\graphrag_2.1.0\graphrag
├── INFO: 使用的数据目录: E:\my_graphrag\graphrag_2.1.0\graphrag\data
├── INFO: 日志保存在: E:\my_graphrag\graphrag_2.1.0\graphrag\data\logs
├── INFO: 索引方法: Standard
├── INFO: 配置更新: 是
├── INFO: 内存分析: 否
├── INFO: 使用指定配置文件: E:\my_graphrag\graphrag_2.1.0\graphrag\data\settings_csv.yaml
├── graphrag-index: 开始构建索引...
├── Loading csv files from input
├── Running Incremental Indexing.
├── create_base_text_units
├── id text doc
├── 0 0e793a0a595a283ab9f6f617c89c18fa27d6368ff... 客户ID: M4412v客户公司: 创联世纪传媒有限公司智能科技v客户所在地: 文市, ... [24af416cf62f
├── 1 451b1fe5651f6477d2e2978a1948d21b9fa3c4ee4f... 客户ID: Y6811v客户公司: 网新海天网络科技有限公司虚拟现实v客户所在地: 蚌埠, ... [583c5204dc4d
├── 2 b692b306737547f4c42b055f9c67b8867a5d... 客户ID: BV435v客户公司: 圣泰信息传媒有限公司虚拟现实v客户所在地: 阳市, ... [601f1b35ea7a
├── 3 75726f4172b0e65cf1978ae39163d83437948ac8ba0c... 客户ID: G1291v客户公司: 吉利网络科技有限公司云计算v客户所在地: 阳市, ... [601f1b35ea7a
├── 4 c58721240c23646624eb32465c9c6fad9514b5bf9a16... 客户ID: Y6811v客户公司: 网新海天网络科技有限公司虚拟现实v客户所在地: 蚌埠, ... [601f1b35ea7a
├── 5 63d8181e1082a4b913a21086a76fda28e35dc35f... 客户ID: HA078v客户公司: 天益传媒集团有限公司大数据技术v客户所在地: 阳市, ... [687b3ab4b3c4
├── 6 c683a97f0d9f9c0c207a3c5f852beb34bd934eb35ed... 客户ID: TA647v客户公司: 和泰伊维达有限公司区块链v客户所在地: 蚌埠, ... [687b3ab4b3c4
```

等待执行完成后，会将新文件（.csv）的索引文件，与 output 文件夹下的所有 .parquet 文件进行合并，形成新的索引文件。即完成不同文件类型的增量更新。在进行问答检索时，可以同时获取到 csv 文件和 txt 文件的索引信息，实现知识图谱的跨文件类型检索。

以上就是两种在实际项目中对 Microsoft GraphRAG 离线构建索引针对不同文件类型构建同一索引及构建多文件索引的具体解决思路。当然，如果想在构建多索引的同时也能实现跨文件存储，则把这两种方案结合应用即可。

而接下来，我们进入最后一个环节，Query 流程的 Python 接口实现。

3. 检索服务的工程化接口封装

Microsoft GraphRAG 的 Query 一共实现了四个检索方法，分别是 local_search、global_search、drift 和 basic。其中 local_search 和 global_search 检索的底层原理在《MicroSoft GraphRAG 深度实战 - Part 4. Microsoft GraphRAG Query构建流程详解》中有详细的介绍，其中 Local Search 是基于实体的检索，Global Search 是基于社区的检索。除此以外，我们再给大家介绍两个新的检索方法，分别是 drift 和 basic。

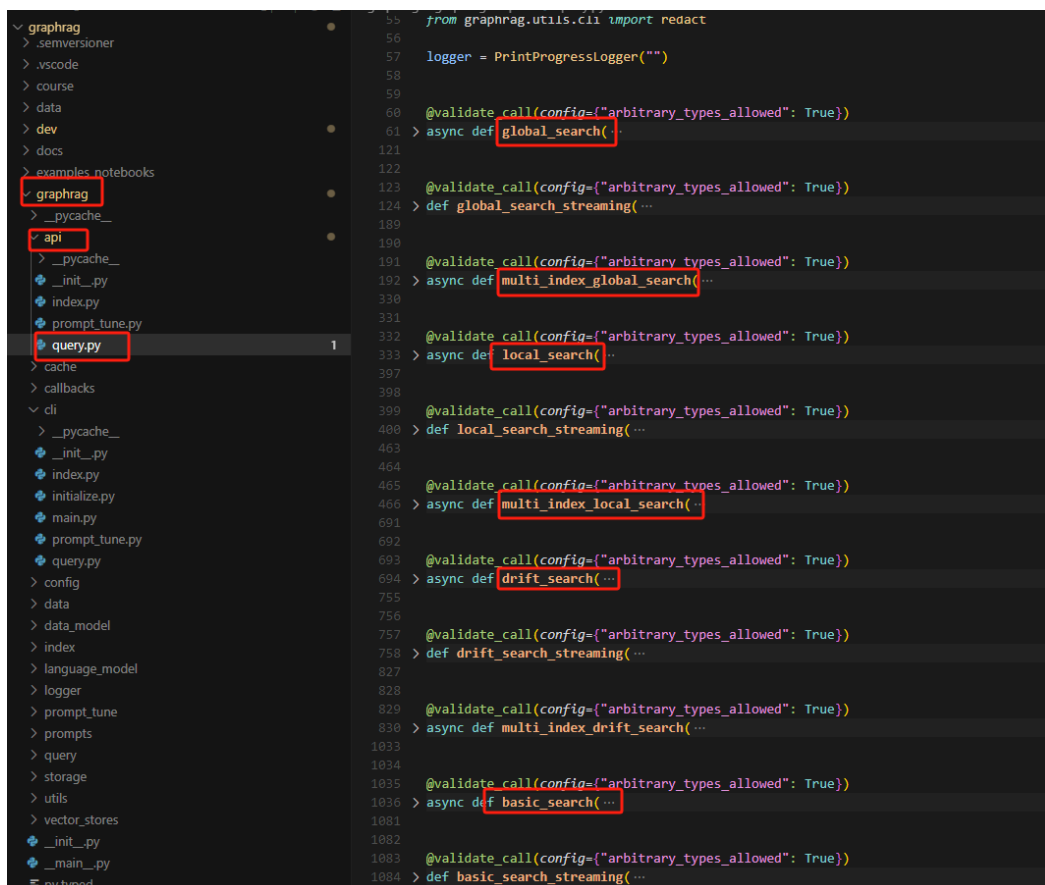
对于 Basic 检索方法，其实现的检索方法非常简单，核心流程如下：

1. 获取到所有的 `text_units`;
2. 提取对应描述的 `Embedding` 向量;
3. 将用户的问题与每一个 `text_units` 的描述进行相似度计算, 得到相似度最高的 `text_unit`;
4. 返回最终的前 `K` 个检索结果。

而 **Drift**, 是对本地检索的一种优化, 核心是为基于实体检索的 **Local Search** 赋予局部全局检索的能力。它的具体做法是:

1. 依次获取 **communities**, **community_reports**, **text_units**, **relationships**, **entities** 数据信息;
2. 获取本地存储的对应 **Embedding** 向量描述;
3. **Primer**: 提交查询时, **DRIFT** 会将其与语义最相关的前 **K** 个社区报告进行比较, 生成一个初始答案以及几个后续问题, 这些问题充当全局搜索的简化版本, 具体逻辑如下:
 - 从前 **K** 个社区报告中列表中随机选择一个社区报告;
 - 借助大模型根据模板扩展原始查询, 生成更丰富的查询表示;
 - 通过嵌入模型将扩展后的查询转换为向量表示;
 - 将社区报告分割成多个部分, 实现并行处理;
 - 根据社区报告中的全局知识, 将查询分解成多个子查询;
 - 每个子查询关注不同的维度或方面, 对每个报告分组异步执行查询分解;
 - 合并所有子查询的处理结果;
4. **Follow-Up**: 有了 **Primer** 过程生成子问题与对应的结果, 后, 使用本地搜索执行每个后续行动。产生额外的中间答案和后续问题, 从而创建一个不断完善的循环, 直到搜索引擎满足其终止标准的自动执行过程;
5. **Output Hierarchy**:: 输出层次结构, 最终输出是问题和答案的层次结构, 按其与原始查询的相关性排序。

对这四种不同的检索方法, 在 **Microsoft GraphRAG** 提供的源码中的 **api** 接口中, 都提供了对应的实现方法, 具体位置为: **graphrag\api\query_api.py** 文件中, 如下所示:



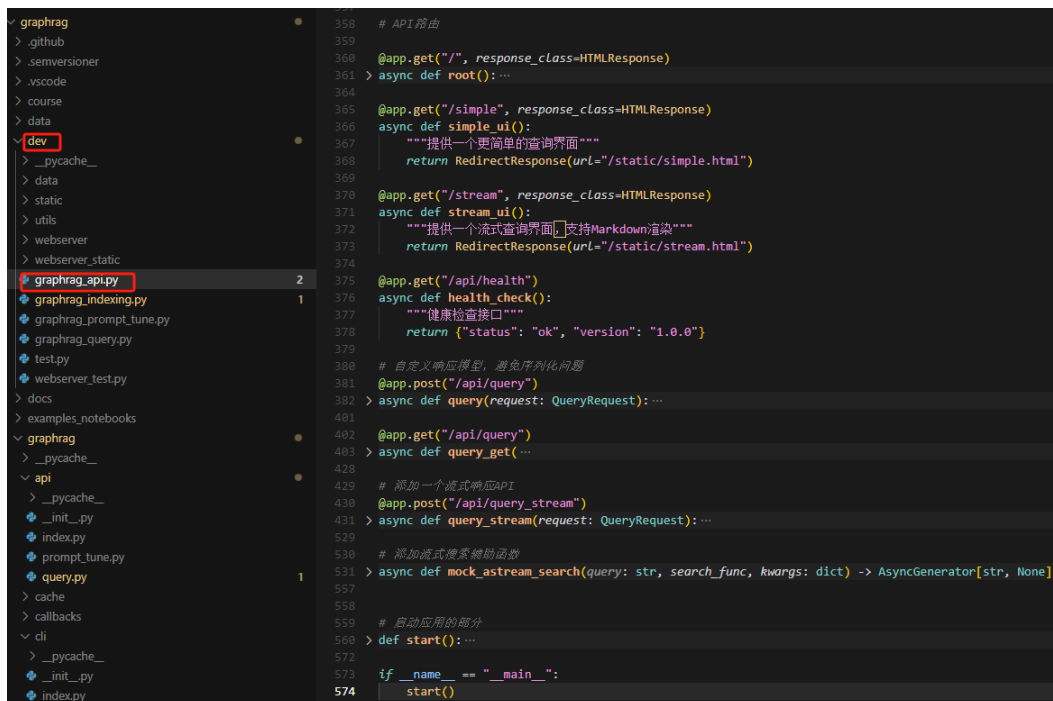
```
graphrag
├── .semversioner
├── .vscode
├── course
├── data
├── dev
├── docs
├── examples_notebooks
├── graphrag
│   ├── __pycache__
│   └── api
│       ├── __pycache__
│       ├── __init__.py
│       ├── index.py
│       ├── prompt_tune.py
│       └── query.py
├── cache
├── callbacks
├── cli
│   ├── __pycache__
│   ├── __init__.py
│   ├── index.py
│   ├── initialize.py
│   ├── main.py
│   ├── prompt_tune.py
│   └── query.py
├── config
├── data
├── data_model
├── index
├── language_model
├── logger
├── prompt_tune
├── prompts
├── query
├── storage
├── utils
├── vector_stores
├── __init__.py
├── __main__.py
└── py.typed
```

```
55 from graphrag.utils.cli import redirect
56
57 logger = PrintProgressLogger("")
58
59
60 @validate_call(config={"arbitrary_types_allowed": True})
61 > async def global_search(...)
121
122
123 @validate_call(config={"arbitrary_types_allowed": True})
124 > def global_search_streaming(...)
189
190
191 @validate_call(config={"arbitrary_types_allowed": True})
192 > async def multi_index_global_search(...)
330
331
332 @validate_call(config={"arbitrary_types_allowed": True})
333 > async def local_search(...)
397
398
399 @validate_call(config={"arbitrary_types_allowed": True})
400 > def local_search_streaming(...)
463
464
465 @validate_call(config={"arbitrary_types_allowed": True})
466 > async def multi_index_local_search(...)
691
692
693 @validate_call(config={"arbitrary_types_allowed": True})
694 > async def drift_search(...)
755
756
757 @validate_call(config={"arbitrary_types_allowed": True})
758 > def drift_search_streaming(...)
827
828
829 @validate_call(config={"arbitrary_types_allowed": True})
830 > async def multi_index_drift_search(...)
1033
1034
1035 @validate_call(config={"arbitrary_types_allowed": True})
1036 > async def basic_search(...)
1081
1082
1083 @validate_call(config={"arbitrary_types_allowed": True})
1084 > def basic_search_streaming(...)
```

因此按照相同的方式, 我们就可以实现 **Query** 流程的 **Python** 接口实现。对应实现的源码文件为: **graphrag\dev\graphrag_query.py** 文件中, 如下所示:

在实现了 Index 和 Query 流程的 Python 接口后，其实就可以基于一些 Python 框架，实现 RESTful API 接口的封装，从而提供后端的服务与前端进行对接。这里我们选择 FastAPI 框架，来介绍如何实现 RESTful API 接口的封装。

在 dev 目录下已经实现了一个 graphrag_api.py 文件，该文件中已经实现了 Query 流程的 Python 的外部接口封装，如下所示：



```
graphrag
├── .github
├── .semversioner
├── .vscode
├── course
├── data
├── dev
│   ├── __pycache__
│   ├── data
│   ├── static
│   ├── utils
│   ├── webservice
│   └── webservice_static
│       └── graphrag_api.py
├── graphrag_indexing.py
├── graphrag_prompt_tune.py
├── graphrag_query.py
├── test.py
├── webservice_test.py
├── docs
├── examples_notebooks
├── graphrag
│   ├── __pycache__
│   └── api
│       ├── __pycache__
│       ├── __init__.py
│       ├── index.py
│       ├── prompt_tune.py
│       └── query.py
├── cache
├── callbacks
├── cli
├── __pycache__
├── __init__.py
└── index.py
```

```
358 # API路由
359
360 @app.get("/", response_class=HTMLResponse)
361 > async def root():...
364
365 @app.get("/simple", response_class=HTMLResponse)
366 async def simple_ui():
367     """提供一个更简单的查询界面"""
368     return RedirectResponse(url="/static/simple.html")
369
370 @app.get("/stream", response_class=HTMLResponse)
371 async def stream_ui():
372     """提供一个流式查询界面，支持Markdown渲染"""
373     return RedirectResponse(url="/static/stream.html")
374
375 @app.get("/api/health")
376 async def health_check():
377     """健康检查接口"""
378     return {"status": "ok", "version": "1.0.0"}
379
380 # 自定义响应模型，避免序列化问题
381 @app.post("/api/query")
382 > async def query(request: QueryRequest):...
401
402 @app.get("/api/query")
403 > async def query_get(...):...
428
429 # 添加一个流式响应API
430 @app.post("/api/query_stream")
431 > async def query_stream(request: QueryRequest):...
520
521 # 添加流式搜索辅助函数
531 > async def mock_astream_search(query: str, search_func, kwargs: dict) -> AsyncGenerator[str, None]:...
557
558
559 # 启动应用的部分
560 > def start():...
572
573 if __name__ == "__main__":
574     start()
```

这个代码中只有一个地方需要大家根据自己的实际情况进行修改，即 start() 函数中的 host 和 port 参数，需要根据实际的网络环境进行修改。



```
559 # 启动应用的部分
560 def start():
561     """启动FastAPI应用"""
562     import uvicorn
563
564     # 预加载数据
565     asyncio.run(load_data())
566
567     # 启动服务
568     host = "0.0.0.0"
569     port = 8000
570     logger.info(f"启动API服务 http://{host}:{port}")
571     uvicorn.run("graphrag_api:app", host=host, port=port, reload=True)
572
573 if __name__ == "__main__":
574     start()
```

修改完成后，首先在命令行终端安装必要的依赖包，再执行 python graphrag_api.py 命令，即可启动 RESTful API 接口服务。如下所示：

```
pip install fastapi uvicorn pydantic_settings
python graphrag_api.py
```

```
(venv) PS E:\my_graphrag\graphrag_2.1.0\graphrag\dev> pip install pydantic_settings
Requirement already satisfied: pydantic_settings in e:\my_graphrag\graphrag_2.1.0\graphrag\venv\lib\site-packages (2.8.1)
Requirement already satisfied: pydantic>=2.7.0 in e:\my_graphrag\graphrag_2.1.0\graphrag\venv\lib\site-packages (from pydantic_settings) (2.10.6)
Requirement already satisfied: python-dotenv>=0.21.0 in e:\my_graphrag\graphrag_2.1.0\graphrag\venv\lib\site-packages (from pydantic_settings) (1.0.1)
Requirement already satisfied: annotated-types>=0.6.0 in e:\my_graphrag\graphrag_2.1.0\graphrag\venv\lib\site-packages (from pydantic>=2.7.0->pydantic_settings) (0.7.0)
Requirement already satisfied: pydantic-core==2.27.2 in e:\my_graphrag\graphrag_2.1.0\graphrag\venv\lib\site-packages (from pydantic>=2.7.0->pydantic_settings) (2.27.2)
Requirement already satisfied: typing-extensions>=4.12.2 in e:\my_graphrag\graphrag_2.1.0\graphrag\venv\lib\site-packages (from pydantic>=2.7.0->pydantic_settings) (4.12.2)
(venv) PS E:\my_graphrag\graphrag_2.1.0\graphrag\dev> python .\graphrag_api.py

INFO: 加载配置...
INFO: 使用输出目录: E:\my_graphrag\graphrag_2.1.0\graphrag\data\output
INFO: 加载索引数据...
INFO: 已加载实体数据, 共 131 条记录
INFO: 已加载文本单元数据, 共 21 条记录
INFO: 已加载社区数据, 共 30 条记录
INFO: 已加载社区报告数据, 共 38 条记录
INFO: 已加载关系数据, 共 147 条记录
INFO: 未找到协变量数据, 将使用None
INFO: 数据加载完成
INFO: 启动API服务 http://0.0.0.0:8000
INFO: Will watch for changes in these directories: ['E:\my_graphrag\graphrag_2.1.0\graphrag\dev']
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO: Started reload process [1956] using StatReload

INFO: Started server process [17556]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

启动服务后, 则可以在 `http://localhost:8000` 地址访问 GraphRAG 的前端页面, 进行网页端的用户交互。如下所示:

GraphRAG 查询界面

基于知识图谱的检索增强生成系统

Q 标准查询

≡ 流式查询 (Markdown)

🔍 请输入您的查询:

例如: GraphRAG系统有哪些核心功能?

▼ 查询类型:

📄 响应类型:

👤 社区级别:

局部查询 (Local)

文本 (Text)

1

☐ 🔄 启用动态社区选择

🚀 提交查询

GraphRAG 查询系统 © 2025 | 基于知识图谱增强的检索式生成