






# RAG 技术详解与实战应用

## 第17讲：企业级RAG：权限、共享与内容安全的全链路方案



# 目录

-  1. 上节回顾 & 本节概要
-  2. 权限多样性以及解决方案
-  3. 共享方式多样性以及解决方案
-  4. 对话管理以及解决方案
-  5. 安全需求以及解决方案
-  6. 企业级RAG的总体实现思路



数据准备	依赖模块详解	构建支持统计分析的论文问答系统
<ul style="list-style-type: none"><li>■ <b>数据收集</b>：使用ArXivQA数据集，从Papers-2024.md中抽取前100篇论文。</li><li>■ <b>知识库构建</b>：下载论文文件到指定目录，用于知识问答。</li><li>■ <b>数据库构建</b>：构建SQLite数据库，存储论文的标题、作者和主题信息。</li></ul>	<ul style="list-style-type: none"><li>■ <b>意图识别</b>：使用IntentClassifier分析用户输入，判断其意图是知识问答还是统计分析。</li><li>■ <b>SQL调用</b>：将自然语言查询转化为SQL语句，执行数据库查询。</li><li>■ <b>统计分析Agent</b>：自动完成数据库查询、数据分析、图表生成和结果汇总。<ul style="list-style-type: none"><li>■ 工具定义：定义run_sql_query和run_code工具，支持SQL查询和代码执行。</li><li>■ ReactAgent：使用ReactAgent工具，实现模型自动调用和任务调度。</li></ul></li></ul>	<ul style="list-style-type: none"><li>■ <b>系统架构</b>：定义两个子工作流，chat_ppl用于知识问答，sql_ppl用于统计分析。</li><li>■ <b>意图识别模块</b>：引入意图识别模块，根据用户查询意图选择合适的工作流。</li><li>■ <b>效果展示</b>：系统能够结合统计分析和知识问答功能，支持全局数据观察需求。</li><li>■ <b>信息抽取</b>：利用大模型进行信息抽取，将每个文档的信息分别入库</li></ul>



# 企业级知识库的多样性需求



在企业实际应用中，知识库不再是简单的信息堆叠，而需要面对来自权限、共享方式与安全保障等多个维度的复杂诉求。下面我们从典型场景出发，系统阐述这些多样性需求及其应对策略。

## ✅ 场景一：按部门隔离的知识访问（权限管理多样性）

### 客户背景：大型制造企业

企业下设多个职能部门（如研发、采购、销售），各自负责不同领域的信息收集与管理。文档内容涵盖供应链合作、成本核算、产品规划等，信息敏感度高，内部访问需严格隔离。

- **部门专属知识库：**各业务单元可自主维护产品文档、市场分析、财务报告等专业内容，系统自动隔离非授权访问。
- **智能标签体系：**支持“研发-技术白皮书”、“市场-竞品分析”等专业标签体系，实现精准检索与受控共享。
- **管理驾驶舱：**高管层可通过“战略视图”标签获取跨部门知识摘要，确保决策支持的同时维护数据安全。

## ✅ 场景二：多种共享方式的协同需求（共享方式多样性）

### 客户背景：咨询服务公司

公司经常与不同客户进行联合项目，涉及文档共享、阶段报告、项目材料和算法资源等。客户使用的工具和偏好多样，需要灵活的共享机制以满足业务合作与文档保密的平衡。

- **算法资源共享：**支持llm模型、embedding模型、检索算法等核心组件在企业内部分享复用
- **差异化调用的知识复用：**一个项目知识库可同时服务内部顾问、客户技术团队及第三方分析机构，能够配置不同的召回规则实现召回解耦，满足多角色精准访问。



## ✅ 场景三：多重安全策略保障内容安全（安全保障多样性）

### 客户背景：金融科技公司

知识库包含大量敏感内容，如用户金融行为分析、监管合规方案、审计材料等，对信息安全的要求极高。




### 安全需求：

- **敏感词智能过滤**：内置多级敏感词识别策略，结合上下文进行动态判断，在问答与检索过程中自动提示、替换或阻断输出，防止企业内部黑名单、客户机密、涉密术语等信息泄露。
- **全链路知识加密**：知识文档在上传、解析、入库、传输及生成阶段均可启用对称或非对称加密机制，确保知识在整个生命周期中不被窃取或篡改。
- **私有化部署方案**：平台可在企业内网私有服务器或专属云环境中完成全栈部署，包括知识库、向量引擎、检索模块与模型推理服务，确保知识数据不经公网传输。系统可无缝集成企业认证、权限与日志体系，形成闭环安全防护结构。

企业级应用场景对知识库提出了更多维度的要求，LazyLLM 针对这些需求，在**权限管理**、**共享模式**以及**安全保障**三方面提供了解决方案。



# 目录

-  1. 上节回顾 & 本节概要
-  2. 权限多样性以及解决方案
-  3. 共享方式多样性以及解决方案
-  4. 对话管理以及解决方案
-  5. 安全需求以及解决方案
-  6. 企业级RAG的总体实现思路

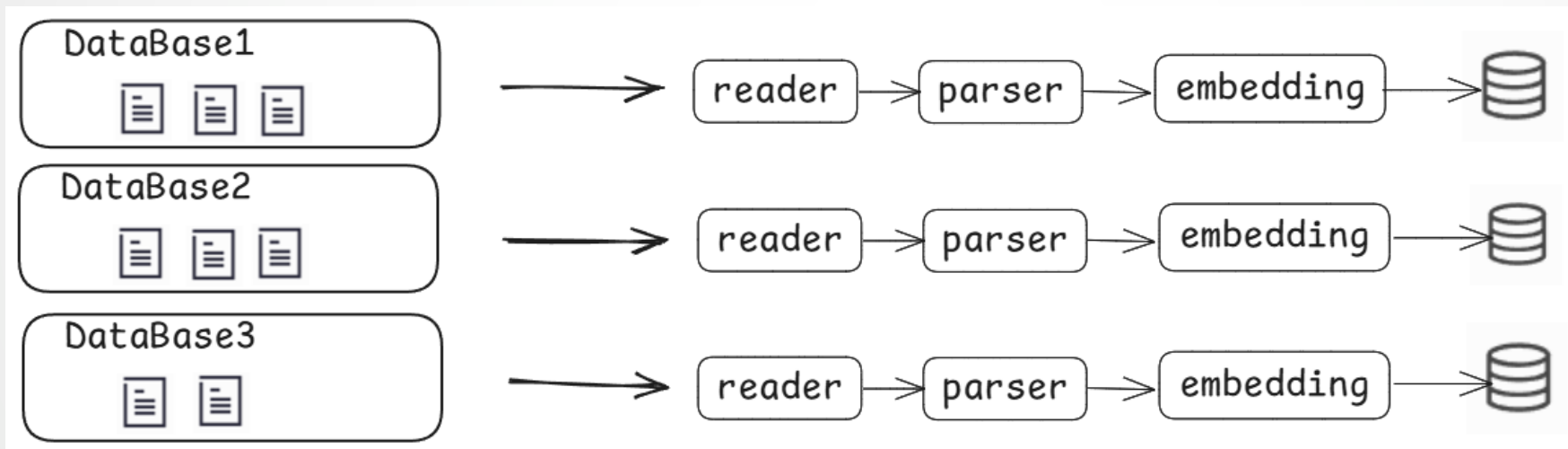


# 权限隔离：支持多部门独立知识运营

**信息隔离与共享：**大型企业中，各部门通常有独立且敏感的文档体系，既要各部门文档相互隔离来确保敏感业务数据的安全性，又支持必要的信息共享，来满足业务协作需求。

？ 如何支持知识库的高频更新与维护？

？ 如果同一篇文档被多个部门使用，需分别入库多次？导致数据冗余和管理困难？



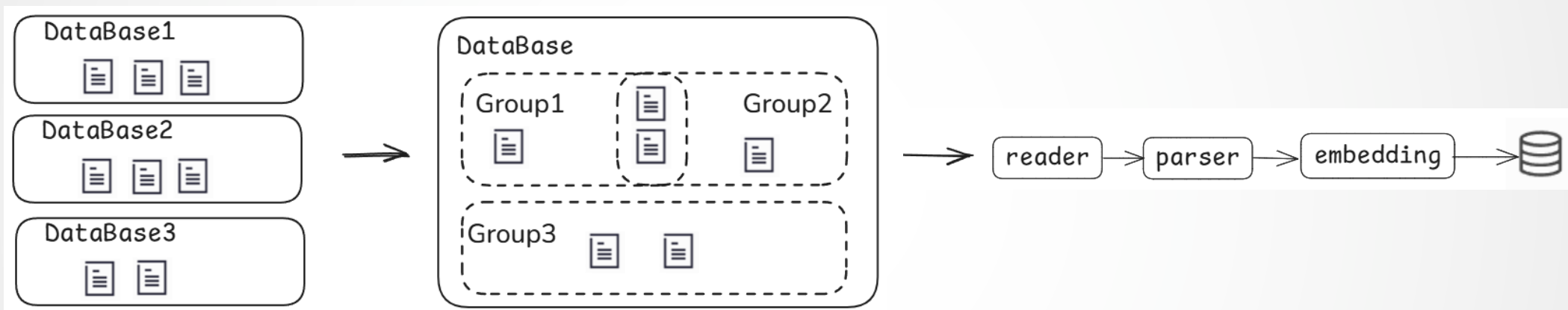
所有共享文档重复解析入库



# 权限隔离：支持多部门独立知识运营

## 可能的解决方案

同一知识库存储内，支持利用**文档管理组**功能进行分组管理，同一文档只需解析一次



传统思路

进阶方案





# 权限多样性：支持更细粒度的访问控制



- 在企业实际运营中，权限控制不仅仅是“哪个部门访问哪些文档”，更涉及**小组、岗位、项目角色**之间对内容的精细隔离与共享控制。
- 企业需要通过标准化鉴权机制（如**基于角色RBAC、基于属性ABAC、基于策略PBAC**）精细控制文档访问。

？ 如何依据标准化鉴权机制组织内容和设置访问权限，确保信息合规使用？

？ 如何根据权限等级细化访问控制，如同一部门内不同人员拥有不同等级的访问权限？

## LazyLLM 解决方案 —— 基于标签（Tag）的权限控制机制：

- ✓ 每个文档可在上传时绑定预定义标签（如部门、项目、安全等级）
- ✓ 检索时支持基于标签的过滤，仅返回符合条件的内容



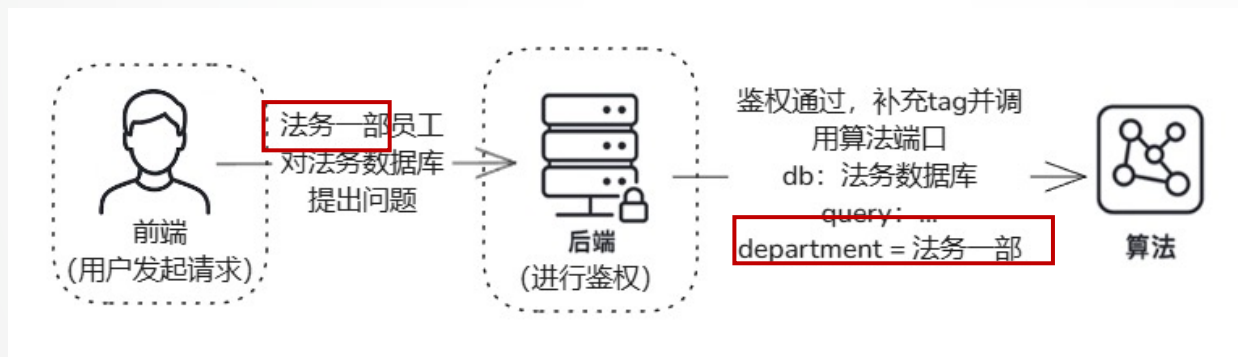
# 权限多样性：支持更细粒度的访问控制

## 🔍 示例：模拟基于角色的权限控制（RBAC）

**目标：**让“法务一部”员工仅能检索本部门的文档

- 定义标签字段：department
- 上传文档时指定：department = 法务一部
- 检索时自动注入过滤条件：filter={"department": "法务一部"}

通过这一机制，实现了**基于角色的隔离访问**，在保障数据安全的同时，也简化了权限策略的实施。



在实际应用中，**鉴权逻辑应由后端统一管理，算法侧不直接处理鉴权**。这样可以确保权限控制的集中化和安全性，避免因算法侧绕过权限而引发的安全漏洞。



- 灵活的增删改查
- 利用**文档管理组**功能进行分组管理

文档管理服务的启用非常简单，只需要在创建 `document` 对象时，将 `manager` 参数设为 `ui`，即可开启带界面的文档管理功能。例如：

设置为 `True` 时仅启动API服务，可基于接口自由开发个性化前端。

```
1. path = "path/to/docs"
2. docs = Document(path, manager='ui')
3. # 注册分组
4. Document(path, name='法务文档管理组', manager=docs.manager)
5. Document(path, name='产品文档管理组', manager=docs.manager)
6. # 启动服务
7. docs.start()
8. time.sleep(3600)
```



- 启动后前端页面如下：

分组列表

上传文件

分组文件列表

删除文件

index	group_name
0	__default__
1	法务文档管理组
2	产品文档管理组

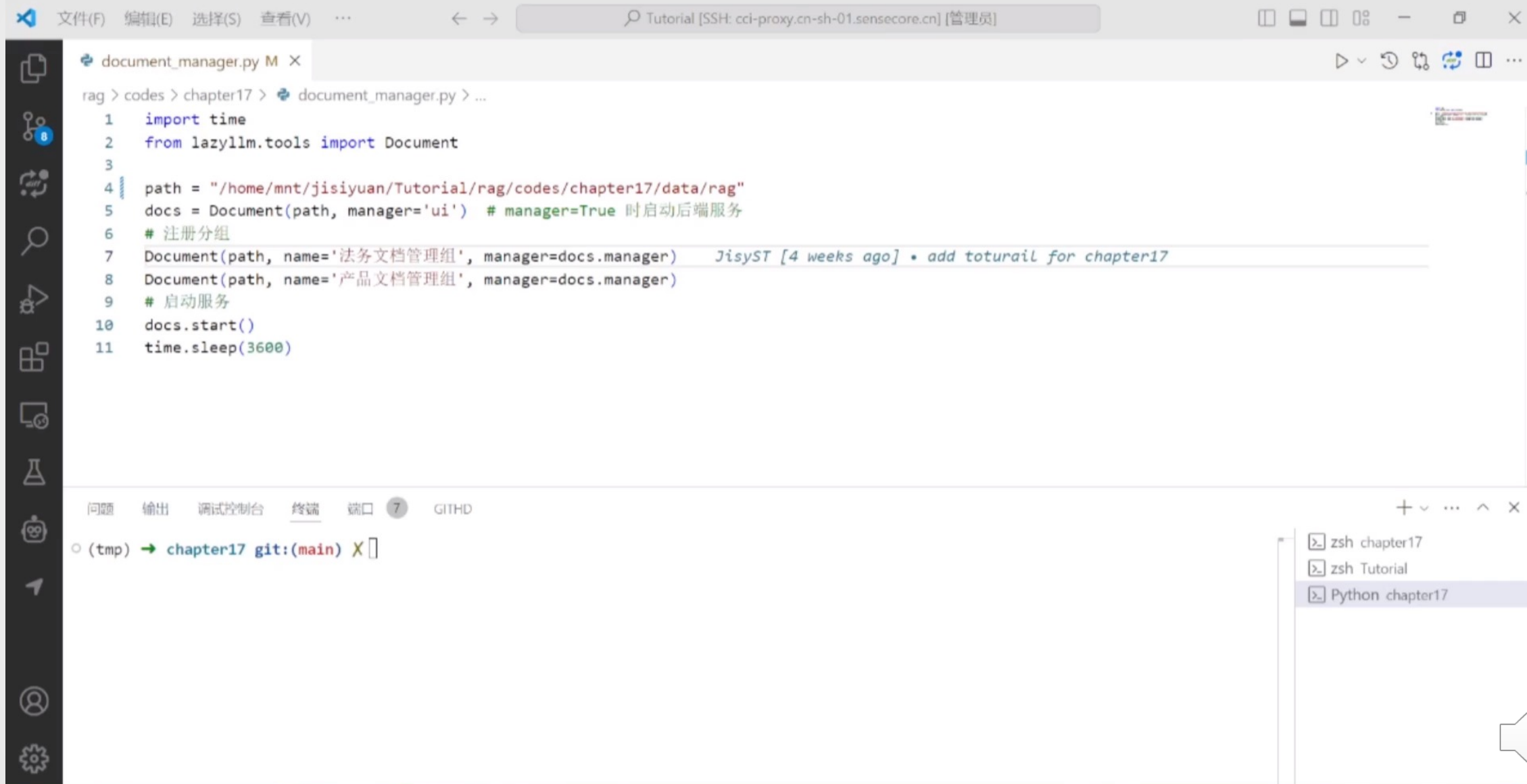
- API接口页面如右：

FastAPI 0.1.0 OAS 3.1

/openapi.json

default	
POST	/generate Generate
GET	/ docs
GET	/list_kb_groups List Kb Groups
POST	/upload_files Upload Files
POST	/add_files Add Files
GET	/list_files List Files
GET	/list_files_in_group List Files In Group
POST	/add_files_to_group_by_id Add Files To Group By Id
POST	/add_files_to_group Add Files To Group
POST	/delete_files Delete Files
POST	/delete_files_from_group Delete Files From Group





The image shows a VS Code editor window with a file named `document_manager.py` open. The code is a Python script that sets up a document manager. The terminal window at the bottom shows the command `chapter17 git:(main) X` and a list of open terminals: `zsh chapter17`, `zsh Tutorial`, and `Python chapter17`.

```
rag > codes > chapter17 > document_manager.py > ...
1  import time
2  from lazyllm.tools import Document
3
4  path = "/home/mnt/jisiyuan/Tutorial/rag/codes/chapter17/data/rag"
5  docs = Document(path, manager='ui') # manager=True 时启动后端服务
6  # 注册分组
7  Document(path, name='法务文档管理组', manager=docs.manager) JisyST [4 weeks ago] • add toturail for chapter17
8  Document(path, name='产品文档管理组', manager=docs.manager)
9  # 启动服务
10 docs.start()
11 time.sleep(3600)
```

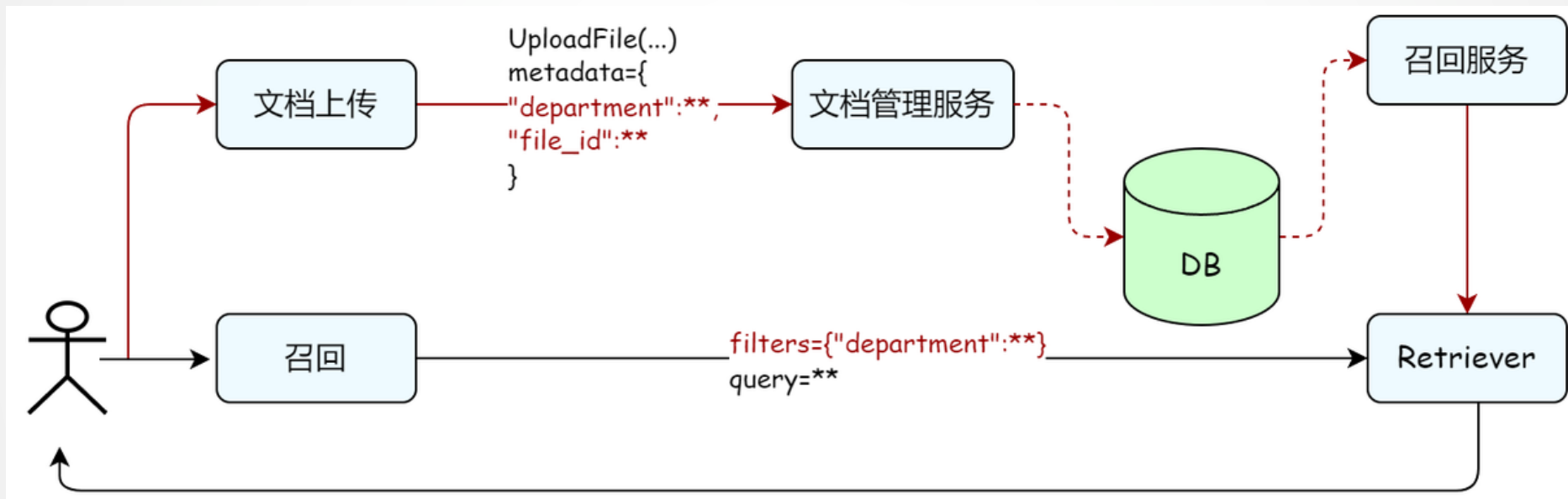
问题 输出 调试控制台 终端 7 GITHD

○ (tmp) → chapter17 git:(main) X

- zsh chapter17
- zsh Tutorial
- Python chapter17

# 基于标签的访问控制

我们可以通过**元数据 (metadata)** 管理和**检索过滤 (filter)** 来实现灵活的分类和查询功能，仅需以下两步

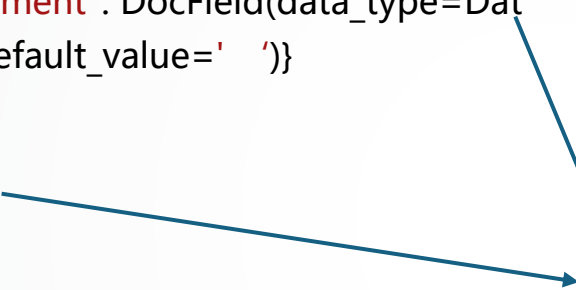


# 基于标签的访问控制

## 第一步：Metadata 添加

使用元数据过滤需**指定milvus数据库**，并且声明**可能用作筛选**的字段，以部门department为例，示例如下：

```
1. CUSTOM_DOC_FIELDS = {"department": DocField(data_type=DataType.VARCHAR, max_size=32, default_value='')}
2. milvus_store_conf = {
3.     'type': 'milvus',
4.     'kwargs': {
5.         'uri': os.path.join(db_path, "milvus.db"),
6.         'index_kwargs': [
7.             {
8.                 'embed_key': 'bge_m3_dense',
9.                 'index_type': 'IVF_FLAT',
10.                'metric_type': 'COSINE',
11.            },
12.            ...
22. law_knowledge_base = Document(
23.     data_path,
24.     name='法务知识库',
25.     manager="ui",
26.     doc_fields=CUSTOM_DOC_FIELDS, # 指定要过滤的字段
27.     store_conf=milvus_store_conf, # 开启milvus数据库
28.     embed=OnlineEmbeddingModule())
```



The diagram consists of two blue arrows. The first arrow originates from the 'CUSTOM\_DOC\_FIELDS' dictionary definition in line 1 and points to its usage as the 'doc\_fields' parameter in line 26. The second arrow originates from the 'milvus\_store\_conf' dictionary definition in line 2 and points to its usage as the 'store\_conf' parameter in line 27.



# 基于标签的访问控制



- 在通过文档管理服务上传文件时，用户可为文件指定需要设定的**元数据 (metadata)** 分类信息。例如：

Parameters

Cancel

Reset

Name	Description
<b>group_name</b> <small>* required</small> string <small>(query)</small>	法务文档管理组
override boolean <small>(query)</small>	false
metadatas string   (string   null) <small>(query)</small>	[{"department": "法务一部", "tag": "case12"}]
user_path string   (string   null) <small>(query)</small>	user_path

Request body required

multipart/form-data

**files** \* required  
array<string>

选择文件 | 合同-2.docx

-

Add string item

Execute

Clear





## 第二步：Metadata 查询

在查询时，用户可以通过 **filter** 机制指定需要过滤的分类信息。可以通过以下方式进行筛选来仅检索来自法务一部和法务二部的文档。

```
1. retriever_support = Retriever(           # 定义召回器
2.     doc = ...,
3.     group_name= ...,
4.     similarity= ...,
5.     topk=2
6. )
7.
8. support_question = "客户关于合同投诉的处理方式"
9. support_res_nodes = retriever_support(
10.     support_question,
11.     filters={ 'department' : [ '法务一部' ] } # 在进行检索时指定已定义的过滤条件
12. )
```



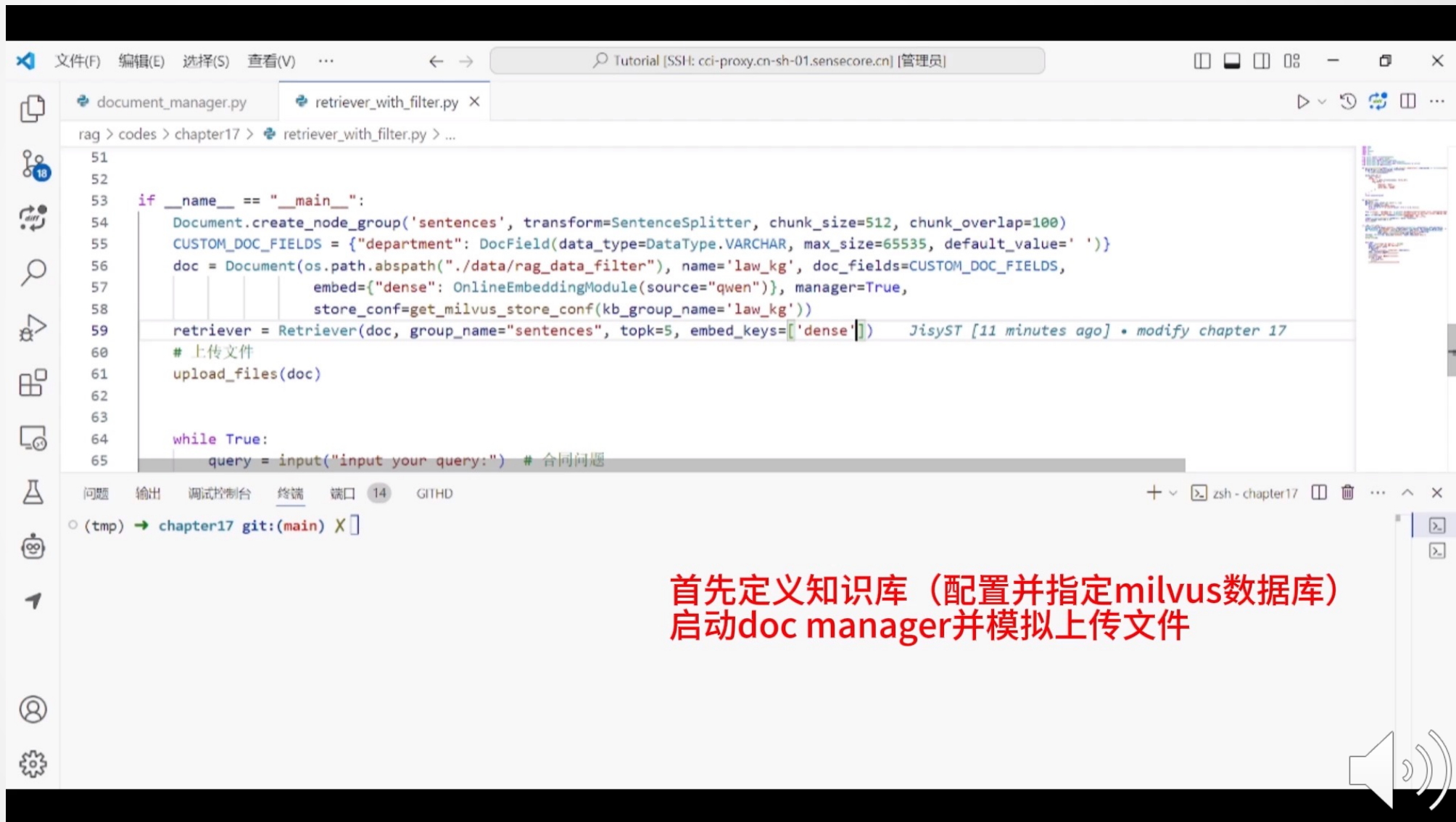
# 基于标签的访问控制

以下示例同时展示了两种检索方式——

- 使用 filter (仅检索“法务一部”文档)
- 不使用 filter (检索所有文档)

该对比仅用于功能展示目的，以便理解系统的过滤机制。

在实际应用中，系统可实现强制绑定过滤条件，确保用户只能检索其所属部门的文档，从而实现文档隔离与权限控制的统一。



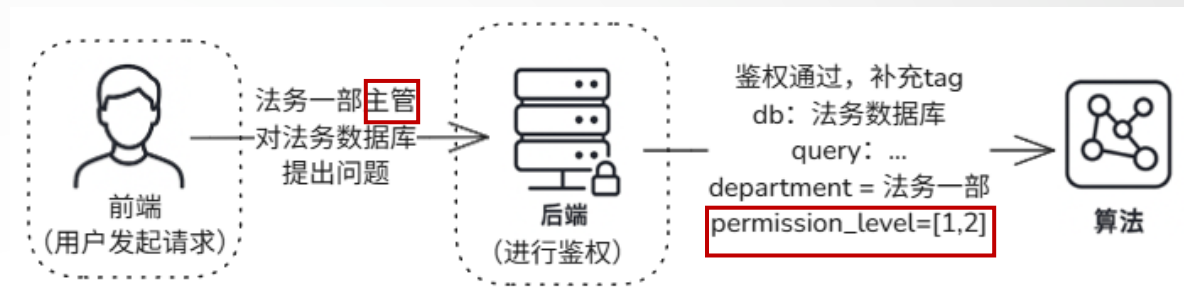
```
rag > codes > chapter17 > retriever_with_filter.py > ...
51
52
53 if __name__ == "__main__":
54     Document.create_node_group('sentences', transform=SentenceSplitter, chunk_size=512, chunk_overlap=100)
55     CUSTOM_DOC_FIELDS = {"department": DocField(data_type=DataType.VARCHAR, max_size=65535, default_value='')}
56     doc = Document(os.path.abspath("../data/rag_data_filter"), name='law_kg', doc_fields=CUSTOM_DOC_FIELDS,
57                     embed={"dense": OnlineEmbeddingModule(source="qwen")}, manager=True,
58                     store_conf=get_milvus_store_conf(kb_group_name='law_kg'))
59     retriever = Retriever(doc, group_name="sentences", topk=5, embed_keys=['dense']) JisyST [11 minutes ago] • modify chapter 17
60     # 上传文件
61     upload_files(doc)
62
63
64 while True:
65     query = input("input your query:") # 合同问题
```

首先定义知识库（配置并指定milvus数据库）  
启动doc manager并模拟上传文件

# 基于标签的访问控制

## 进阶：细化权限等级的权限控制

- **等级 1**：普通员工，仅能查看基础财务报表。
- **等级 2**：主管，能查看部门预算和项目支出。
- **等级 3**：经理及以上，能够访问财务决策和敏感报表。



### 1. 注册 权限等级- permission\_level 字段:

```
CUSTOM_DOC_FIELDS = {"department": DocField(data_type=DataType.VARCHAR, max_size=32, default_value=""),  
                    "permission_level": DocField(data_type=DataType.INT32, default_value=1)}
```

### 2. 上传文档同时标记权限等级 (如permission\_level = 1)







```
files = [('files', ('普通文档.pdf', io.BytesIO(...)),  
         ('files', ('敏感文档.pdf', io.BytesIO(...)))]  
metadatas=[{"department": "法务一部", "permission_level": 1},  
            {"department": "法务一部", "permission_level": 2}]]
```

### 3. 检索时指定权限等级

```
nodes = retriever(query, filters={'department': ['法务一部'], 'permission_level': [1,2]})
```



# 目录

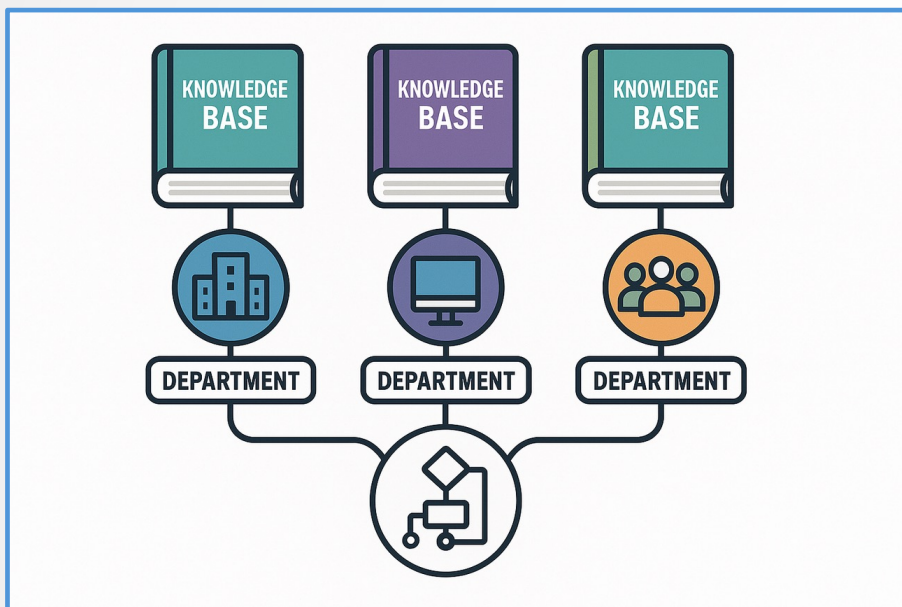
-  1. 上节回顾& 本节概要
-  2. 权限多样性以及解决方案
-  3. 共享方式多样性以及解决方案
-  4. 对话管理以及解决方案
-  5. 安全需求以及解决方案
-  6. 企业级RAG的总体实现思路



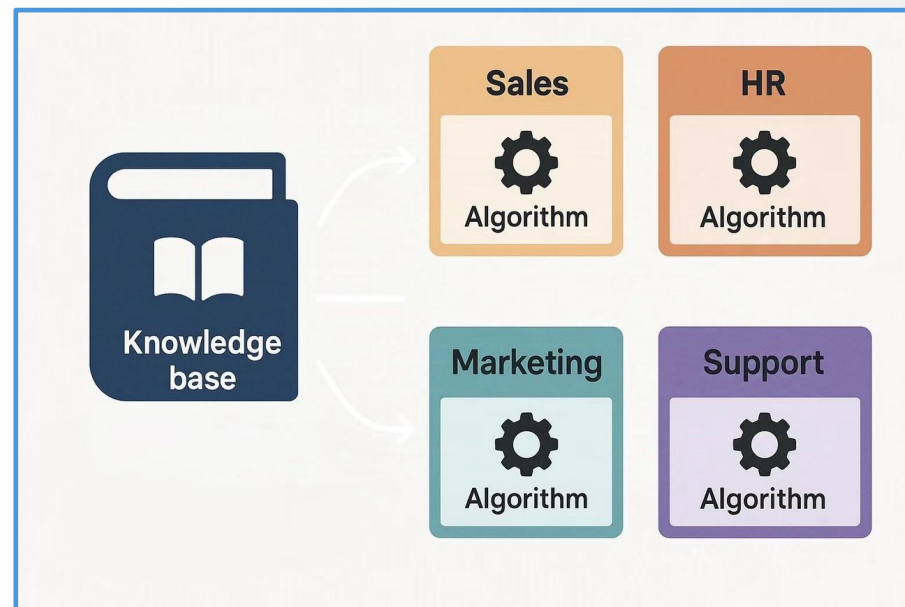
# 共享灵活性：支持多源知识与算法自由适配



在企业中，各部门业务领域不同，存在两种情况。即 “一算法对多知识库” 和 “多算法对一知识库” 两种需求，系统需同时支持以适配业务。



以金融公司为例，风控与市场分析部门或共用文本解析和嵌入算法预处理数据，前者知识库含历史交易与客户信用记录，后者含市场动态与竞品情报，系统需支持算法在不同知识库复用。



以电商企业为例，推荐系统与搜索优化部门分别用协同过滤嵌入、词向量相似度排序算法处理同一用户行为数据集，系统需支持在同一知识库独立运行不同算法以生成针对性结果。





# 共享灵活性：支持多源知识与算法自由适配



✓ 同一套算法在多个知识库中的应用场景已在前面权限的部分讨论过。

✚ 接下来，我们实现在同一知识库中，通过不同文档分组实现算法多样化的场景。

```
docs = Document(path, manager=True, embed=OnlineEmbeddingModule())
# 注册分组
Document(path, name='法务文档管理组', manager=docs.manager)
Document(path, name='产品文档管理组', manager=docs.manager)
# 模拟文档上传
docs.start()
files = [('files', ('产品文档.txt', io.BytesIO("这是关于产品的信息。该文档由产品部编写。\\n来自产品文档管理组".encode("utf-8")), 'text/plain'))]
files = [('files', ('法务文档.txt', io.BytesIO("这是关于法律事务的说明。该文档由法务部整理。\\n来自法务文档管理组".encode("utf-8")), 'text/plain'))]
...
```

为同一知识库注册分组  
并模拟上传两篇文档

```
# 为 产品文档管理组 设置切分方式为按 段落 切分
doc1 = Document(path, name= '产品文档管理组', manager=docs.manager)
doc1.create_node_group(name="block", transform=lambda s: s.split("\\n") if s else "")
retriever1 = Retriever([doc1], group_name="block", similarity="cosine", topk=3)

# 为 法务文档管理组 设置切分方式为按 句子 切分
doc2 = Document(path, name= '法务文档管理组', manager=docs.manager)
doc2.create_node_group(name="line", transform=lambda s: s.split("。") if s else "")
retriever2 = Retriever([doc2], group_name="line", similarity="cosine", topk=3)
```

为同一知识库的不同文档组  
分别定义不同的切分算法

按段落切分

>>> 这是关于产品的信息。该文档由产品部编写。  
>>> 来自产品文档管理组

按句子切分

>>> 这是关于法律事务的说明  
>>> 该文档由法务部整理  
>>> 来自法务文档管理组



# 召回解耦：支持知识库与召回服务灵活协同



## 企业需求背景：

为应对知识共享与复用，企业需要灵活高效的知识组织能力。

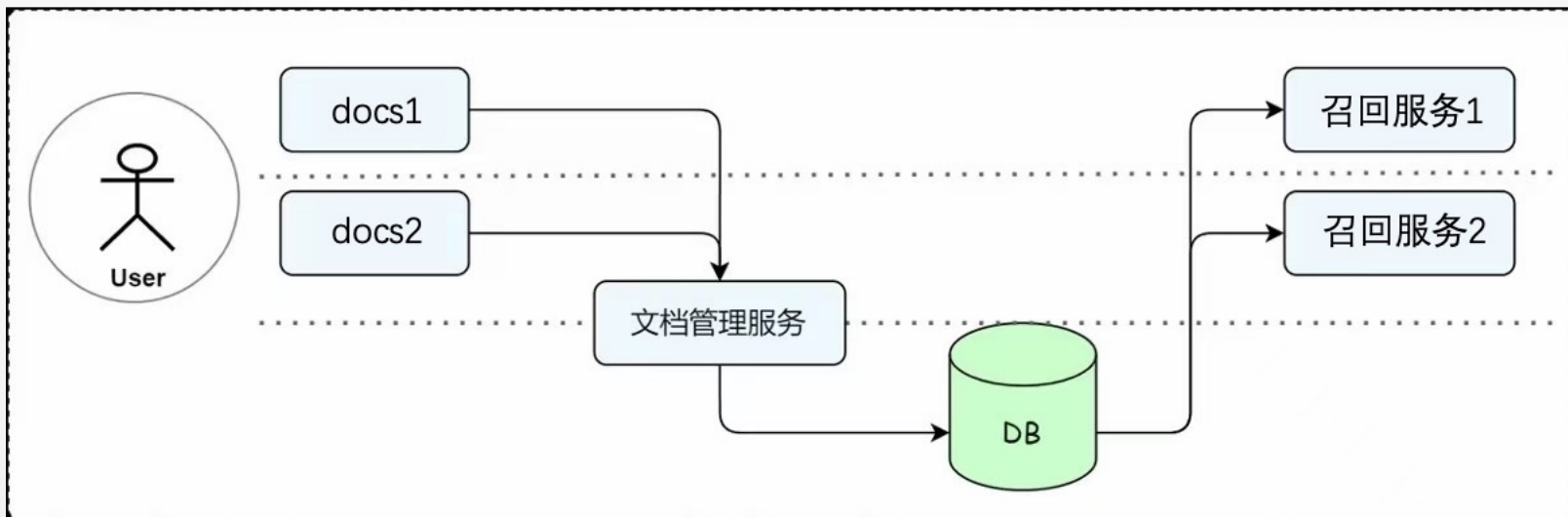
- 单一部门知识具备跨场景应用潜力和需求
- 特定检索或知识问答服务往往依赖多个部门的知识库整合



## LazyLLM解决方案：

——文档管理与RAG召回服务解耦。

- **单知识库多服务**：一个知识库支持多个RAG召回
- **多知识库联合召回**：单个召回可绑定多个知识库



# 多知识库管理和召回



而具体实现起来，仅需以下两步骤，可搭建多知识库管理和召回流程。

## 1. 初始化知识库 —— 可为不同的知识库自定义算法

### 1. # 法务知识库

2. law\_knowledge\_base = Document(law\_data\_path, name='法务知识库', embed=OnlineEmbeddingModule())
3. law\_knowledge\_base.add\_reader("\*\*.pdf", LawFileReader) # 注册专用于法务文档的 PDF 解析器
4. law\_knowledge\_base.create\_node\_group('sentence', transform=LawSentenceSplitter) # 注册适用于法务文档的自定义切分算法

### 5. # 产品知识库

6. product\_knowledge\_base = Document(product\_data\_path, name='产品知识库', embed=OnlineEmbeddingModule())
7. product\_knowledge\_base.add\_reader("\*\*.pdf", TechnicalPDFReader)
8. product\_knowledge\_base.create\_node\_group('sentence', transform=ProductSentenceSplitter)

### 9. # 用户支持知识库

10. support\_knowledge\_base = Document(support\_data\_path, name='用户支持知识库', embed=OnlineEmbeddingModule())
11. support\_knowledge\_base.add\_reader("\*\*.pdf", FAQPDFReader)
12. support\_knowledge\_base.create\_node\_group('sentence', transform=FAQSentenceSplitter)

为不同知识库灵活地注册不同的算法，但需要保持 node group 的 name 一致，以便后续进行联合召回





# 多知识库管理和召回



## 2. 启用 RAG 召回服务

定义知识库后，可为其灵活配置召回服务，将文档管理对象传入 Retriever 即可。

### 组合法务 + 产品知识库，处理与产品相关的法律问题

```
3. retriever_product = Retriever(  
4.     [law_knowledge_base, product_knowledge_base],  
5.     group_name="sentence",  
6.     similarity="cosine",  
7.     topk=1  
8.)  
  
9. product_question = "A产品功能参数和产品合规性声明"  
10. product_res_nodes = retriever_product(product_question)
```



>>> node1(来自法务知识库)：根据《产品质量法》第十八条，产品合规性声明需明确功能参数及其合规性保障...

>>> node2(来自产品知识库)：A产品在功能参数上符合行业合规标准，具体指标详见产品合规性声明...

### 组合法务 + 客户支持知识库，处理客户支持相关问题

```
11. retriever_support = Retriever(  
12.     [law_knowledge_base, support_knowledge_base],  
13.     group_name="sentence",  
14.     similarity="cosine",  
15.     topk=1  
16.)  
  
17. support_question = "客户投诉的处理方式以及会导致的法律问题"  
18. support_res_nodes = retriever_support(support_question)
```









>>> node1(来自用户支持知识库)：针对客户投诉的处理方式，应先安抚客户情绪，再核实事实，提供合理解释，并记录客户反馈...

>>> node2(来自法务知识库)：在客户投诉处理过程中，需遵循《消费者权益保护法》，避免侵权及虚假陈述...



# 目录

-  1. 上节回顾 & 本节概要
-  2. 权限多样性以及解决方案
-  3. 共享方式多样性以及解决方案
-  4. 对话管理以及解决方案
-  5. 安全需求以及解决方案
-  6. 企业级RAG的总体实现思路



在企业应用中，对话系统需记忆历史、支持多用户并发与流式输出。LazyLLM 提供 `globals` 配置中心，集中管理对话历史与上下文，隔离会话、自动清理，确保数据一致与高效流转。

🎯 接下来将介绍如何用 `globals` 实现历史对话和多用户并发管理。

- 对话前，`init_session_config` 初始化独立会话配置，并整理用户历史和预定义对话到 `globals["global_parameters"]["history"]`。
- 推理时，`respond_stream` 将当前输入和历史对话一起提交至模型异步处理，边生成边流式输出，并自动追加新对话到历史中，保持上下文连贯。

```
1. from lazyllm import globals
2. llm = lazyllm.OnlineChatModule(stream=True)
3. DEFAULT_FEW_SHOTS = []
4. def init_session_config(session_id, user_history=None):
5.     """初始化会话配置，包含用户历史"""
6.     globals.init_sid(session_id)
7.     for shot in DEFAULT_FEW_SHOTS:
8.         history.append({
9.             "role": "user",
10.            "content": shot["user"]
11.        })
12.     history.append({
13.         "role": "assistant",
14.         "content": shot["assistant"]
15.     })
```

```
16. # 添加用户历史
17. history = []
18. if user_history:
19.     for h in user_history:
20.         history.append({
21.             "role": "user",
22.             "content": h.user
23.         })
24.         history.append({
25.             "role": "assistant",
26.             "content": h.assistant
27.         })
28.     globals["global_parameters"] = {
29.         "history": history
30.     }
```



# 历史对话管理

```
31. def respond_stream(session_id, model_in, user_history=None):
32.     # 初始化会话配置
33.     init_session_config(session_id, user_history)
34.     # 提交对话任务
35.     ctx = contextvars.copy_context()
36.     func_future = ThreadPool.submit(lambda: ctx.run(llm, model_in, llm_chat_history=history))
37.     response = ""
38.     while True:
39.         if message := FileSystemQueue().dequeue():
40.             msg = "".join(message)
41.             response += msg
42.             yield msg
43.             elif func_future.done():
44.                 break
45.     # 获取完整响应
46.     model_out = func_future.result()
47.     # 更新历史记录
48.     globals["global_parameters"]["history"].append({
49.         "role": "user",
50.         "content": model_in
51.     })
52.     globals["global_parameters"]["history"].append({
53.         "role": "assistant",
54.         "content": model_out
55.     })
56.     return model_out
```

💡 整体来看，这段代码依托 globals 可实现：

- 灵活加载和隔离管理不同用户的历史对话；
- 支持系统内部预置 few-shot 示例，引导模型更好地理解任务；
- 在结合历史上下文的基础上，总结、改写并生成新的对话内容或问题。



# 历史对话管理

The screenshot shows a VS Code editor window with the following components:

- File Explorer:** Shows the file structure: `rag > codes > chapter17 > chat_with_his.py > ...`
- Editor:** Displays the file `chat_with_his.py` with the following code:

```
111  
112 # 指定有历史的请求  
113 history = [  
114     ChatHistory(user="你好", assistant="你好呀!"),  
115     ChatHistory(user="你能帮我翻译成英文吗?", assistant="当然可以, 请告诉我你需要翻译的内容。")  
116 ]  
117 handle_request("user123", "香蕉", user_history=history)  
118 print("\n\n")  
119  
120 handle_request("user123", "总结这段对话")  
121
```
- Terminal:** Shows the command `python chat_with_his.py` being executed in a terminal window titled `Python - chapter17`. The prompt is `(tmp) → chapter17 git:(dev-chapter16) X`.
- Output Panel:** On the right side, there are several warning icons (yellow triangles) indicating errors or warnings.



- 让我们来看一下执行效果：

## 首先指定预定义对话：

```
DEFAULT_FEW_SHOTS = [  
    {"role": "user", "content": "你是谁? "},  
    {"role": "assistant", "content": "我是你的智能助手。"}  
]
```

初始化时，为用户 注入两轮历史对话。对话中指定了一个翻译任务。

```
history = [  
    ChatHistory(user="你好", assistant="你好呀! "),  
    ChatHistory(user="你能帮我翻译成英文吗?", assistant="当然可以，请告诉我你需要翻译的内容。")  
]
```

## 执行效果：

- 第一次请求只输入 “香蕉”，系统按历史上下文正常生成与香蕉相关的回答 “banana”。
- 第二次请求输入 “总结这段对话”，系统基于完整历史成功输出对话总结。
- 并且预先指定的对话也在chat history中。



# 多用户并发对话管理

基于上一小节代码，再进一步实现多用户并发对话管理。

- 每次对话开始时，系统通过 `init_session_config` 初始化会话环境，并用 `globals._init_sid(session_id)` 设置当前协程的 `session_id`，保证不同会话配置隔离。
- 推理过程中，系统利用基于 `session_id` 隔离的 `FileSystemQueue` 实现流式输出，模型生成的新内容实时写入队列，前端可同步展示，多会话并发时也能避免数据串流和冲突。

```
1. def init_session_config(session_id, user_history=None):
2.     """初始化会话配置，包含用户历史"""
3.     globals._init_sid(session_id)
4.     for shot in DEFAULT_FEW_SHOTS:
5.         history.append({
6.             "role": "user",
7.             "content": shot["user"]
8.         })
9.         history.append({
10.            "role": "assistant",
11.            "content": shot["assistant"]
12.        })
```

这行代码用于将当前上下文与传入的 **session\_id** 绑定，使得后续对 `globals["global_parameters"]` 等全局对象的访问都是在当前 session 下进行的。





# 多用户并发对话管理

```
31. def respond_stream(session_id, model_in, user_history=None):
32.     # 初始化会话配置
33.     init_session_config(session_id, user_history)
34.     # 提交对话任务
35.     ctx = contextvars.copy_context()
36.     func_future = ThreadPool.submit(lambda: ctx.run(llm, model_in, llm_chat_history=history))
37.     response = ""
38.     while True:
39.         if message := FileSystemQueue().dequeue():
40.             msg = "".join(message)
41.             response += msg
42.             yield msg
43.         elif func_future.done():
44.             break
45.     # 获取完整响应
46.     model_out = func_future.result()
47.     # 更新历史记录
48.     globals["global_parameters"]["history"].append({
49.         "role": "user",
50.         "content": model_in
51.     })
52.     globals["global_parameters"]["history"].append({
53.         "role": "assistant",
54.         "content": model_out
55.     })
56.     return model_out
```

系统利用基于 `session_id` 隔离的 `FileSystemQueue` 实现流式输出，模型生成的新内容实时写入队列，多会话并发时也能避免数据串流和冲突。

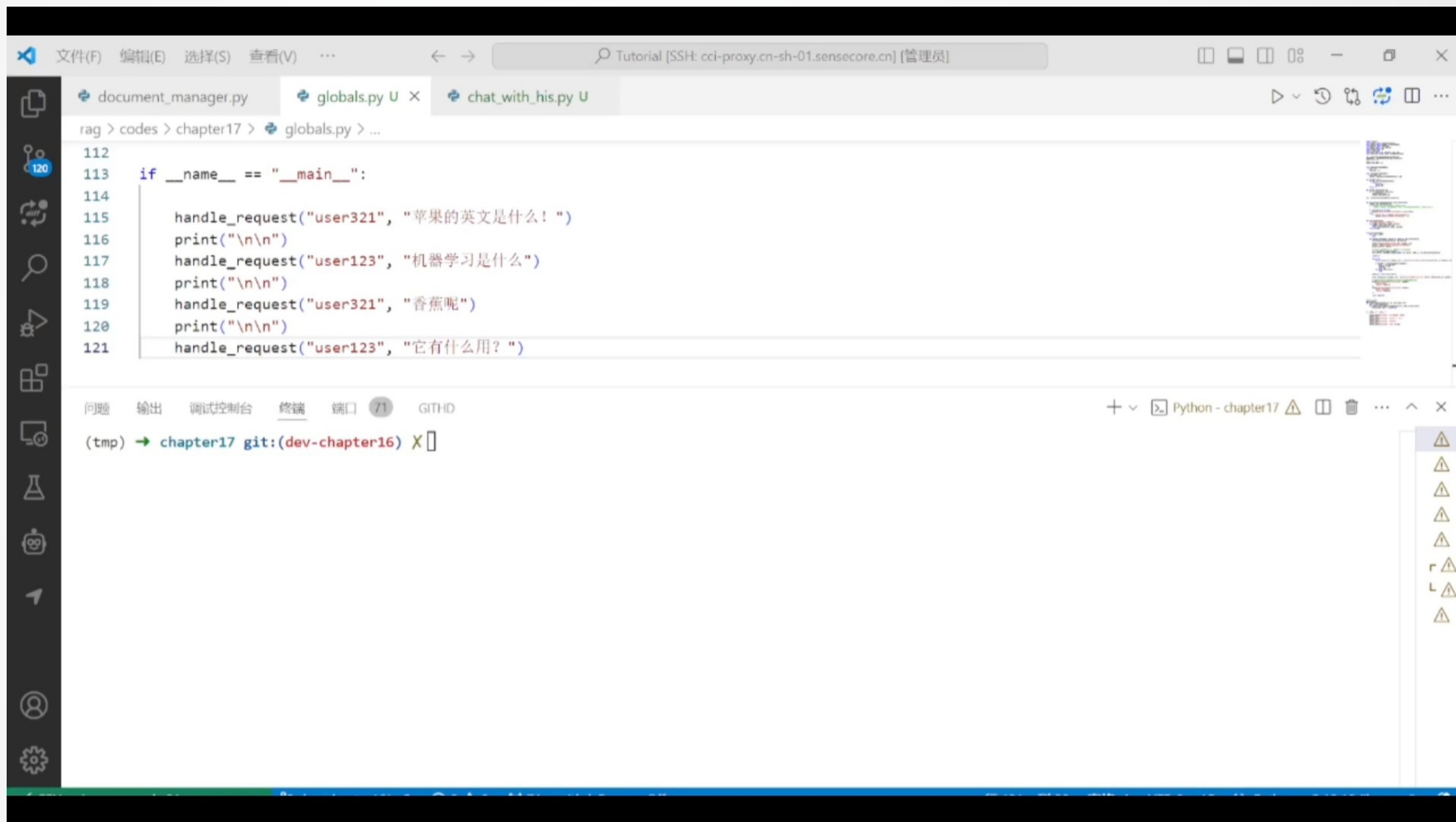




# 多用户并发对话管理

## Chat顺序:

u1:  
"苹果的英文是什么?"  
u2:  
"机器学习是什么"  
u1:  
"香蕉呢?"  
u2:  
"它有什么用"



```
rag > codes > chapter17 > globals.py > ...  
112  
113 if __name__ == "__main__":  
114  
115     handle_request("user321", "苹果的英文是什么!")  
116     print("\n\n")  
117     handle_request("user123", "机器学习是什么")  
118     print("\n\n")  
119     handle_request("user321", "香蕉呢")  
120     print("\n\n")  
121     handle_request("user123", "它有什么用?")
```

问题 输出 调试控制台 终端 窗口 71 GITHD

(tmp) → chapter17 git:(dev-chapter16) X



## 效果示例:

- 用户 1 问 “苹果的英文” ， 再问 “香蕉” 时， 模型能记住当前会话是翻译任务。
- 用户 2 问 “机器学习是什么？” 后， 追问 “它有什么作用？” 时， 模型能关联上下文解释应用场景。

两个用户的历史对话内容相互不影响。






! 注意， 实现上述功能需要用redis数据库实现文件系统输出管理， 设置方法为：

```
export LAZYLLM_DEFAULT_FSQUEUE=REDIS
```

```
export LAZYLLM_FSQREDIS_URL=redis://[user name]:[password]@[host]/[port]
```



# 目录

-  1. 上节回顾 & 本节概要
-  2. 权限多样性以及解决方案
-  3. 共享方式多样性以及解决方案
-  4. 对话管理以及解决方案
-  5. 安全需求以及解决方案
-  6. 企业级RAG的总体实现思路



在企业知识库建设中，安全始终是首要考量，尤其当内容涉及公司政策、财务报表、客户合同等敏感数据时，任何泄露都可能引发严重法律和商业后果。因此，系统需具备全面的保护机制。

## 1. 加密

- **私有数据保护**：通过数据隔离机制，确保不同业务或部门间的数据隔离，防止数据泄露。
- **知识加密**：对文档在存储与传输过程中的全链路加密，确保数据机密性与完整性。
- **模型加密**：支持模型调用过程中的数据加密，避免敏感信息泄露。

## 2. 私有化部署

- **本地化模型推理引擎**：核心组件部署于内网环境，保障数据安全。
- **数据本地处理**：确保知识数据在企业内部完成，避免外泄。
- **强化权限控制**：结合网络隔离和多因子认证，实现安全访问。



在企业知识库建设中，安全始终是首要考量，尤其当内容涉及公司政策、财务报表、客户合同等敏感数据时，任何泄露都可能引发严重法律和商业后果。因此，系统需具备全面的保护机制。

3. 信创

为保障核心技术自主可控，系统全面兼容国家信创名录中的软硬件产品

- 国产CPU**：鲲鹏、龙芯等，提供高性能计算支持。
- 国产操作系统**：麒麟、统信UOS等，确保系统底层安全。
- 国产数据库**：达梦、人大金仓等，敏感数据存得更放心。
- 全链路合规**：从芯片（如鲲鹏/飞腾）到软件均符合信创标准，通过国家信息安全认证。

CPU

序号	产品名称	送测单位
1	鲲鹏920	深圳市海思半导体有限公司
2	龙芯3C5000L	龙芯中科技术股份有限公司
3	申威1621	无锡先进技术研究院
4	龙芯3A4000/3B4000	龙芯中科技术股份有限公司
5	龙芯3A5000/3B5000	龙芯中科技术股份有限公司

操作系统

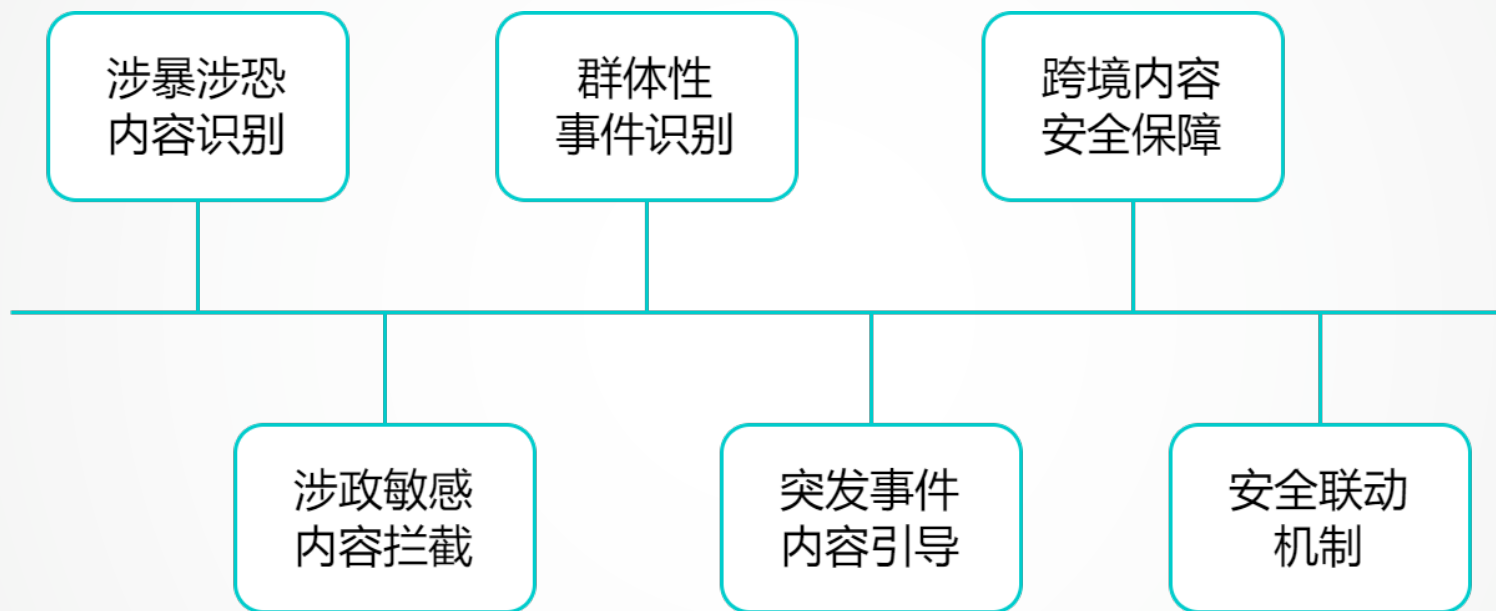
序号	产品名称	送测单位
1	银河麒麟桌面操作系统V10 (内核版本5.4)	麒麟软件有限公司
2	银河麒麟高级服务器操作系统V10 (内核版本4.19)	麒麟软件有限公司
3	统信服务器操作系统V20 (内核版本4.19)	统信软件技术有限公司
4	方德高可信服务器操作系统V4.0 (内核版本4.19)	中科方德软件有限公司

数据库

序号	产品名称	送测单位
1	达梦数据库管理系统V8.4	武汉达梦数据库股份有限公司
2	PolarDB V2.0	阿里云计算有限公司
3	TDSQL关系型数据库管理系统软件V8.0	腾讯云计算（北京）有限责任公司
4	瀚高安全版数据库系统V4.5	瀚高基础软件股份有限公司
5	虚谷数据库管理系统V11.0	成都虚谷伟业科技有限公司



在企业级RAG系统中，公共安全不仅关乎企业自身的声誉与合规风险，更关联到模型输出对社会舆论、信息安全乃至国家安全的影响。系统应具备以下能力，确保模型生成内容不突破公共安全底线：



👮 通过公共安全模块的建设，企业可有效防控大模型在生成内容过程中可能引发的社会层面风险，提升企业数字治理能力，践行平台责任。



# 如何维护公共安全?

在知识库管理和检索中，系统需具备敏感信息过滤机制，自动识别和屏蔽个人隐私、法律合规等敏感内容，防止泄露。

**LazyLLM支持自定义规则配置**，管理员可动态维护敏感词列表，结合分词、正则表达式和DFA算法实现精准过滤。接下来将介绍如何使用 DFA 算法在 LazyLLM 中实现敏感词过滤。

## 第一步

定义一个DFA过滤算法，后续作为接入 Lazyllm的组件。

```
1. class DFAFilter:
2.     def __init__(self, sensitive_words):
3.         ...
4.         self.add_word(word)
5.     def add_word(self, word):
6.         ...
```

```
5. def __call__(self, text, replace_char="*"):
6.     ...
7.     while start < length:
8.         ...
9.         while i < length and text[i] in node:
10.            node = node[text[i]]
11.            if self.end_flag in node:
12.                # 匹配到敏感词，替换为指定字符
13.                result.append(replace_char * (i - start + 1))
14.                start = i + 1
15.                break
16.            i += 1
17.        else:
18.            # 未匹配到敏感词，保留原字符
19.            result.append(text[start])
20.            start += 1
21.    return ''.join(result)
```





# 如何维护公共安全?



## 第二步

将定义的DFAFilter 注册为文档服务的node group。

2. # 定义业务敏感词

3. sensitive\_words = ['合同']

4. # 将敏感词过滤算法嵌入到业务逻辑中

5. Document.create\_node\_group(name="dfa\_filter", parent="sentences",  
transform=DFAFilter(sensitive\_words))

6. # 组合法务 + 产品知识库, 处理与产品相关的法律问题

7. retriever\_product = Retriever(

8. [law\_knowledge\_base, product\_knowledge\_base],

9. group\_name="dfa\_filter",

10. similarity="cosine",

11. topk=2

12.)

### 屏蔽前:

#### 3. 合同履行责任

公司在合同中明确约定产品功能、交付标准和服务期限。

若因公司原因未履行合同约定内容, 用户有权根据合同条款要求公司承担违约责任。

若用户在产品中嵌入或调用外挂、脚本工具或未经授权的API, 公司有权终止服务并保留追究法律责任的权利

### 屏蔽后:

#### 3.\*\*履行责任

公司在\*\*中明确约定产品功能、交付标准和服务期限。

若因公司原因未履行\*\*约定内容, 用户有权根据\*\*条款要求公司承担违约责任。

若用户在产品中嵌入或调用外挂、脚本工具或未经授权的API, 公司有权终止服务并保留追究法律责任的权利

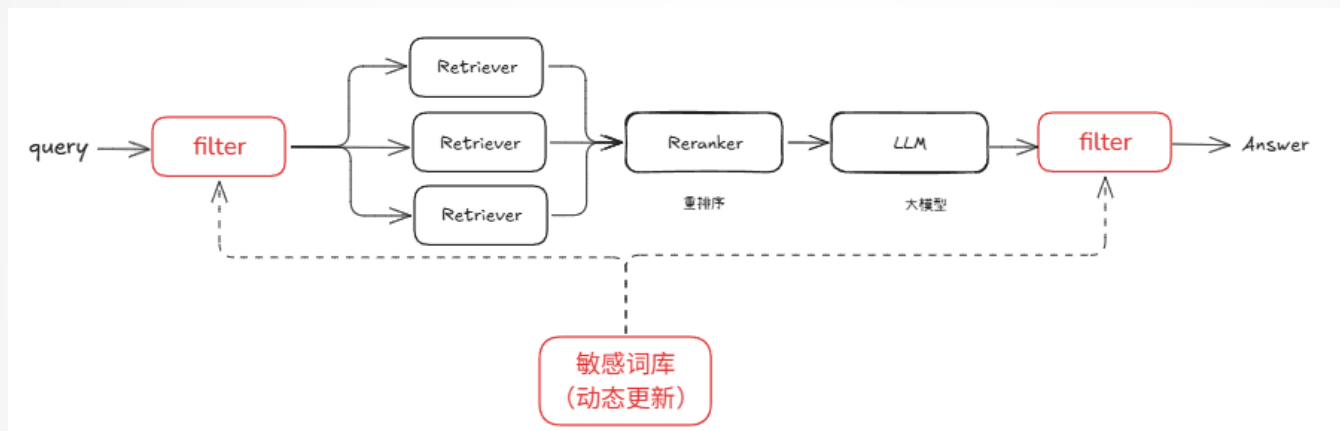




# 如何维护公共安全?

## 全流程敏感词过滤

在实际应用中，除了原文档内容进行敏感词过滤外，我们还需对用户输入和大模型输出进行同样的处理。



with pipeline() as ppl:

```
ppl.query_filter = lambda x: DFAFilter(sensitive_words).filter(x)
```

```
ppl.retriever = Retriever(...)
```





```
ppl.reranker = ...
```

```
ppl.llm = ...
```

```
ppl.output_filter = lambda x: DFAFilter(sensitive_words).filter(x)
```



# 目录

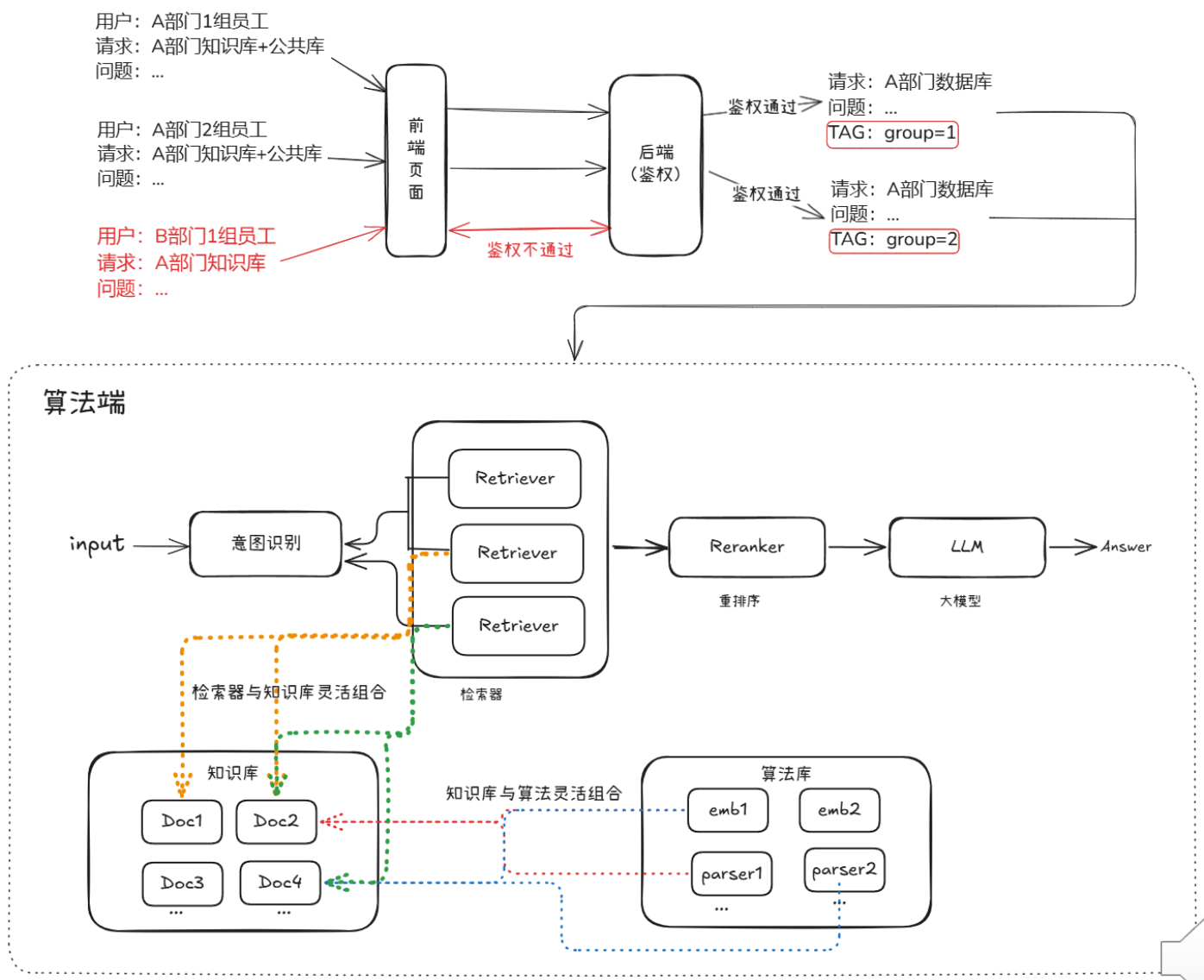
-  1. 上节回顾 & 本节概要
-  2. 权限多样性以及解决方案
-  3. 共享方式多样性以及解决方案
-  4. 对话管理以及解决方案
-  5. 安全需求以及解决方案
-  6. 企业级RAG的总体实现思路



# 企业级RAG的总体实现思路

在前文中，我们从权限控制、共享方式、安全保障等多个维度详细解析了企业级RAG系统在真实落地过程中面临的核心需求与挑战。

接下来，我们将整合上述要素，提出一个功能完善、可落地的企业级RAG搭建思路。



**感谢聆听**  
**Thanks for Listening**

