# 政企问答 RAG实现流程

政企问答RAG系统提供了知识库管理，文档管理，数据库存储和问答系统的解决方案。

## 1.知识数据存储

**关系数据库层（SQLite）**

利用关系数据库构建知识库；

* knowledge\_database 表：存储知识库基本信息，knowledge\_id作为该库的主键；

* knowledge\_document 表：存储文档基本信息，将knowledge\_id作为外键；

**向量检索层（Elasticsearch）**

* document\_meta 索引：存储文档摘要和全文内容

* chunk\_info 索引：存储文档分块、向量表示和页面信息

## 2. 问答核心实现流程

### 2.1 文档处理与入库流程

**步骤1：文档上传接口**

@app.post("/v1/document")  
async def add\_document(knowledge\_id, title, category, file, background\_tasks)

**主要处理逻辑：**

1. 接收用户上传的文档文件

1. 在对应的知识数据库中创建文档记录，获得document\_id

1. 将文件保存到本地存储目录

1. 启动后台任务进行文档内容提取和向量化

**步骤2：文档内容提取（PDF处理）**

def \_extract\_pdf\_content(self, knowledge\_id, document\_id, title, file\_path)

**处理过程：**

1. 使用pdfplumber打开PDF文件

1. 逐页提取文本内容

1. 前4页内容作为文档摘要

1. 对每页内容进行向量化编码

1. 将页级内容存储到ES的chunk\_info索引

**步骤3：文档分块处理**

def split\_text\_with\_overlap(text, chunk\_size, chunk\_overlap)

**分块策略：**

* 固定长度分块：默认256字符

* 重叠处理：默认20字符重叠，保证语义连续性

* 每个chunk独立编码并存储向量表示

**步骤4：向量化存储**

对每个文本块进行embedding：

def get\_embedding(self, text) -> np.ndarray:  
 return EMBEDDING\_MODEL\_PARAMS["embedding\_model"].encode(text, normalize\_embeddings=True)

### 2.2 问答检索流程

**步骤1：问答接口调用**

@app.post("/chat")  
def chat(req: RAGRequest) -> RAGResponse

**步骤2：混合检索策略**

def query\_document(self, query: str, knowledge\_id: int) -> List[str]

**检索包含两个并行过程：**

**全文检索（BM25）：**

* 基于Elasticsearch的match查询

* 使用ik分词器进行中文分词

* 返回相关性最高的50个候选块

**语义检索（KNN）：**

* 对查询语句进行向量编码

* 在ES中执行KNN向量相似性搜索

* 返回语义最相似的50个候选块

#### 步骤3：结果融合（RRF - Reciprocal Rank Fusion）

# RRF融合算法实现  
k = 60  
fusion\_score = {}  
for idx, record in enumerate(search\_results):  
 \_id = record["\_id"]  
 fusion\_score[\_id] = fusion\_score.get(\_id, 0) + 1 / (idx + k)

**融合策略：**

* 使用RRF算法融合两种检索结果

* 每个文档的最终得分 = BM25得分 + 向量检索得分

* 选择Top-K个最相关的文档块

#### 步骤4：重排序（可选）

def get\_rank(self, text\_pair) -> np.ndarray:  
 # 使用BGE-reranker模型对查询-文档对进行精确打分

如果启用重排序功能：

* 构建查询-文档对

* 使用专门的rerank模型进行精确相关性评分

* 按照新的得分重新排序

### 2.3 答案生成流程

#### 步骤1：构建RAG提示词

BASIC\_QA\_TEMPLATE = '''现在的时间是{#TIME#}。你是一个专家，你擅长回答用户提问，帮我结合给定的资料，回答下面的问题。  
如果问题无法从资料中获得，或无法从资料中进行回答，请回答无法回答。如果提问不符合逻辑，请回答无法回答。  
如果问题可以从资料中获得，则请逐步回答。  
  
资料：  
{#RELATED\_DOCUMENT#}  
  
问题：{#QUESTION#}  
'''

**提示词构建过程：**

1. 将检索到的相关文档拼接作为背景资料

1. 插入当前时间信息

1. 插入用户的原始问题

1. 设置明确的回答规则和格式要求

#### 步骤2：LLM生成回答

def chat(self, messages: List[Dict], top\_p: float, temperature: float):  
 completion = self.client.chat.completions.create(  
 model=self.llm\_model,  
 messages=messages,  
 top\_p=top\_p,  
 temperature=temperature  
 )  
 return completion.choices[0].message

**生成参数配置：**

* 模型：GLM-4-air

* temperature：0.1（低随机性，确保答案准确性）

* top\_p：1.0

* max\_tokens：1024

## 3. 关键配置参数

### 3.1 模型配置

models:  
 embedding\_model:  
 bge-small-zh-v1.5:  
 dims: 512  
 rerank\_model:  
 bge-reranker-base  
 llm:  
 glm-4-air

### 3.2 RAG参数优化

rag:  
 chunk\_size: 256 # 文档分块大小  
 chunk\_overlap: 20 # 分块重叠长度  
 chunk\_candidate: 10 # 候选文档块数量  
 use\_embedding: true # 启用向量检索  
 use\_rerank: false # 重排序开关  
 use\_rrf: true # 启用结果融合

## 4. 系统特点分析

### 4.1 优势

1. **混合检索**：结合了关键词匹配和语义理解

1. **分层存储**：关系数据库+向量数据库的合理分工

1. **可扩展性**：支持多知识库管理

1. **模块化设计**：各组件职责明确，便于维护

### 4.2 可改进空间

1. **重排序功能**：当前未启用，可能影响检索精度

1. **查询优化**：缺少查询改写和意图理解

1. **多模态支持**：目前仅支持文本，可扩展图表处理

1. **缓存机制**：可添加检索结果缓存提升性能