

# 第8章 RAG检索增强生成

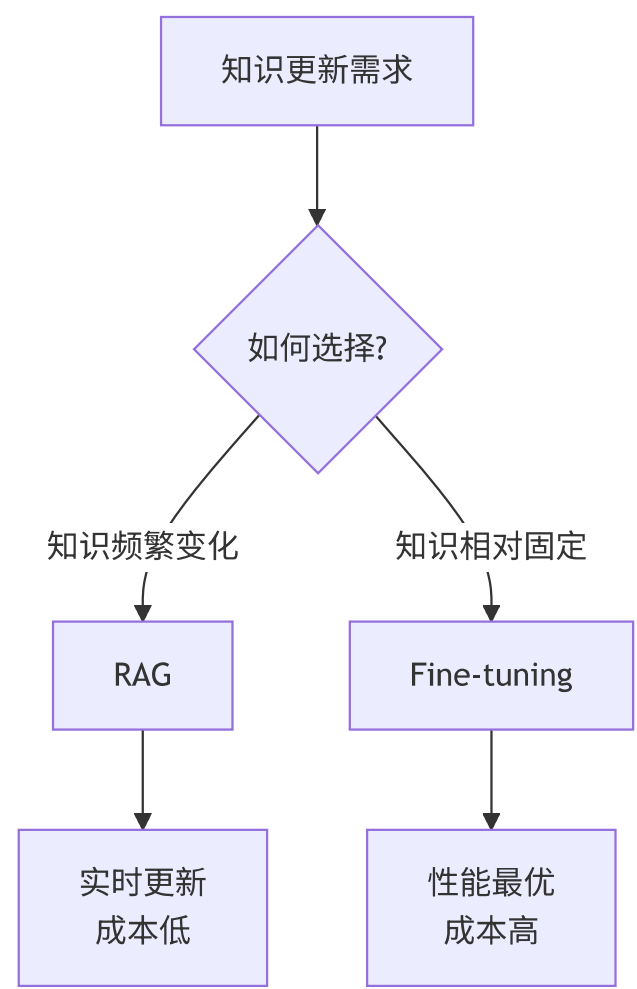
让大模型拥有外部知识库，解决幻觉和知识时效性问题

## 8.1 为什么需要RAG？

### 8.1.1 大模型的局限性

问题	表现	解决方案
知识截止	只知道训练数据中的知识	RAG
幻觉	编造不存在的事实	RAG + 事实验证
领域知识	缺乏企业私有知识	RAG
上下文限制	无法处理超长文档	RAG + 分块

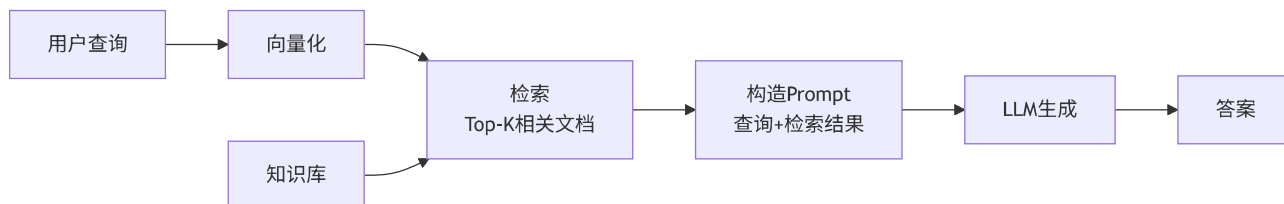
### 8.1.2 RAG vs Fine-tuning



维度	RAG	Fine-tuning
知识更新	随时更新（修改知识库）	需要重新训练
成本	低（无需训练）	高

维度	RAG	Fine-tuning
可解释性	高（可追溯来源）	低
响应时间	稍慢（需要检索）	快
准确性	依赖检索质量	高

## 8.2 RAG基本流程



### 核心步骤：

1. **文档索引**：将知识库文档向量化并存储
2. **查询向量化**：将用户问题转换为向量
3. **相似度检索**：找到最相关的K个文档片段
4. **Prompt构造**：将检索结果和问题组合成prompt
5. **生成答案**：LLM基于上下文生成回答

## 8.3 文档处理Pipeline

### 8.3.1 文档加载

```
from langchain.document_loaders import (
    TextLoader,
    PyPDFLoader,
    UnstructuredMarkdownLoader,
    WebBaseLoader
)

# 文本文件
loader = TextLoader("document.txt", encoding="utf-8")
documents = loader.load()

# PDF
pdf_loader = PyPDFLoader("document.pdf")
pdf_docs = pdf_loader.load()

# 网页
web_loader = WebBaseLoader("https://example.com")
web_docs = web_loader.load()
```

*# Markdown*

```
md_loader = UnstructuredMarkdownLoader("document.md")
md_docs = md_loader.load()
```

### 8.3.2 文档分块 (Chunking)

#### 为什么要分块?

- ☒ LLM上下文长度有限
- ☒ 提高检索精度 (细粒度匹配)
- ☒ 减少噪声

#### 分块策略:

##### 1. 固定长度分块

```
from langchain.text_splitter import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
    chunk_size=500,          # 每块500字符
    chunk_overlap=50,        # 重叠50字符 (保持上下文连贯)
    separator="\n"
)

chunks = text_splitter.split_documents(documents)
```

##### 2. 递归分块 (推荐)

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    separators=["\n\n", "\n", "。", "!", "?", " ", ""] # 优先级递减
)

chunks = text_splitter.split_documents(documents)
```

##### 3. 语义分块

```
class SemanticChunker:
    def __init__(self, embedding_model, similarity_threshold=0.7):
        self.embedding_model = embedding_model
        self.threshold = similarity_threshold
```

```

def split(self, text):
    """
    基于语义相似度分块
    """

    sentences = split_into_sentences(text)

    chunks = []
    current_chunk = [sentences[0]]

    for i in range(1, len(sentences)):
        # 计算当前句子与chunk的相似度
        chunk_embedding = self.embedding_model.embed(" ".join(current_chunk))
        sentence_embedding = self.embedding_model.embed(sentences[i])

        similarity = cosine_similarity(chunk_embedding, sentence_embedding)

        if similarity > self.threshold:
            # 相似，添加到当前chunk
            current_chunk.append(sentences[i])
        else:
            # 不相似，开始新chunk
            chunks.append(" ".join(current_chunk))
            current_chunk = [sentences[i]]

    chunks.append(" ".join(current_chunk))
    return chunks

```

分块参数选择：

文档类型	推荐chunk_size	chunk_overlap	说明
代码	300-500	50	保持函数完整性
文章	800-1200	200	段落级别
对话	500-800	100	保持上下文
表格	不分块	-	保持结构完整

### 8.3.3 元数据添加

为chunk添加元数据，提升检索和可追溯性。

```

from langchain.schema import Document

def add_metadata(chunks, source_file, author=None):
    """
    为chunk添加元数据

```

```

"""
enriched_chunks = []

for i, chunk in enumerate(chunks):
    doc = Document(
        page_content=chunk.page_content,
        metadata={
            "source": source_file,
            "chunk_id": i,
            "author": author,
            "created_at": datetime.now().isoformat(),
            "char_count": len(chunk.page_content),
            # 还可以添加: 部门、标签、重要性等
        }
    )
    enriched_chunks.append(doc)

return enriched_chunks

```

## 8.4 Embedding模型选择

### 8.4.1 主流Embedding模型

模型	维度	中文	性能	部署	推荐场景
OpenAI text-embedding-ada-002	1536	✓	优秀	API	快速原型
OpenAI text-embedding-3-small	512/1536	✓	优秀	API	性价比高
bge-large-zh	1024	✓	优秀	本地	中文首选
m3e-base	768	✓	良好	本地	中文、轻量
sentence-transformers	384/768	✗	良好	本地	英文

### 8.4.2 使用Embedding模型

#### OpenAI Embedding:

```

from openai import OpenAI

client = OpenAI()

def get_embedding(text, model="text-embedding-3-small"):
    text = text.replace("\n", " ")
    response = client.embeddings.create(
        input=[text],
        model=model
    )

```

```

    )
    return response.data[0].embedding

# 批量处理
def get_embeddings_batch(texts, batch_size=100):
    embeddings = []
    for i in range(0, len(texts), batch_size):
        batch = texts[i:i+batch_size]
        response = client.embeddings.create(
            input=batch,
            model="text-embedding-3-small"
        )
        embeddings.extend([item.embedding for item in response.data])
    return embeddings

```

### 本地Embedding模型:

```

from sentence_transformers import SentenceTransformer

# 中文模型
model = SentenceTransformer('BAAI/bge-large-zh-v1.5')

# 生成embedding
texts = ["这是一个测试文本", "另一个示例"]
embeddings = model.encode(texts)

print(embeddings.shape)  # (2, 1024)

```

## 8.4.3 Embedding优化技巧

### 1. 加入查询指令 (Instruction)

```

# bge模型推荐的格式
query_instruction = "为这个句子生成表示以用于检索相关文章: "

query = "什么是Transformer? "
query_with_instruction = query_instruction + query

query_embedding = model.encode(query_with_instruction)

```

### 2. 非对称检索优化

```

# 查询和文档使用不同的prompt
query_prompt = "查询: "

```

```
doc_prompt = "文档："  
  
query_embedding = model.encode(query_prompt + query)  
doc_embeddings = model.encode([doc_prompt + doc for doc in documents])
```

## 8.5 向量数据库

### 8.5.1 向量数据库对比

数据库	类型	性能	特点	适用规模
Faiss	内存	极快	单机、高性能	百万级
Chroma	内存/持久化	快	易用、轻量	中小型
Milvus	分布式	快	企业级、可扩展	大规模
Pinecone	云服务	快	托管、免运维	任意规模
Qdrant	持久化	快	Rust编写、高效	中大型
Weaviate	分布式	中	丰富功能	大规模

### 8.5.2 使用Chroma（推荐入门）

```
from langchain.vectorstores import Chroma  
from langchain.embeddings import OpenAIEmbeddings  
  
# 初始化embedding模型  
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")  
  
# 创建向量数据库  
vectorstore = Chroma.from_documents(  
    documents=chunks,  
    embedding=embeddings,  
    persist_directory="./chroma_db" # 持久化目录  
)  
  
# 检索  
query = "什么是Transformer?"  
results = vectorstore.similarity_search(query, k=3)  
  
for doc in results:  
    print(doc.page_content)  
    print(doc.metadata)  
    print("---")
```

### 8.5.3 使用Faiss（高性能）

```
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings

# 本地embedding模型
embeddings = HuggingFaceEmbeddings(
    model_name="BAAI/bge-large-zh-v1.5"
)

# 创建FAISS索引
vectorstore = FAISS.from_documents(chunks, embeddings)

# 保存索引
vectorstore.save_local("faiss_index")

# 加载索引
vectorstore = FAISS.load_local("faiss_index", embeddings)

# 检索
docs = vectorstore.similarity_search(query, k=5)
```

### 8.5.4 混合检索（Hybrid Search）

结合关键词检索和向量检索。

```
from langchain.retrievers import BM25Retriever, EnsembleRetriever

# 1. BM25关键词检索
bm25_retriever = BM25Retriever.from_documents(chunks)
bm25_retriever.k = 3

# 2. 向量检索
vector_retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

# 3. 混合检索（加权组合）
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, vector_retriever],
    weights=[0.4, 0.6] # BM25权重0.4, 向量检索权重0.6
)

# 使用
results = ensemble_retriever.get_relevant_documents(query)
```

**混合检索的优势：**



- ☒ BM25擅长精确匹配（专有名词、数字）
- ☒ 向量检索擅长语义理解
- ☒ 互补，提升召回率

## 8.6 检索策略优化

### 8.6.1 重排序 (Reranking)

检索后再用更强的模型重新排序。



实现：

```

from sentence_transformers import CrossEncoder

class Reranker:
    def __init__(self, model_name='cross-encoder/ms-marco-MiniLM-L-6-v2'):
        self.model = CrossEncoder(model_name)

    def rerank(self, query, documents, top_k=3):
        """
        重排序检索结果

        Args:
            query: 查询文本
            documents: 候选文档列表
            top_k: 返回前k个
        """
        # 构造query-document对
        pairs = [[query, doc.page_content] for doc in documents]

        # 计算相关性分数
        scores = self.model.predict(pairs)

        # 排序
        scored_docs = list(zip(documents, scores))
        scored_docs.sort(key=lambda x: x[1], reverse=True)

        # 返回top-k
        return [doc for doc, score in scored_docs[:top_k]]
  
```

# 使用

```

reranker = Reranker()

# 先用向量检索获取候选
candidates = vectorstore.similarity_search(query, k=20)

# 重排序, 选出最相关的3个
top_docs = reranker.rerank(query, candidates, top_k=3)

```

## 8.6.2 查询改写 (Query Rewriting)

改写查询以提升检索效果。

### 策略1: 查询扩展

```

def expand_query(original_query, llm):
    """
    生成查询的多个变体
    """
    prompt = f"""
    请为以下查询生成3个语义相似但表达不同的变体:

    原查询: {original_query}

    变体1:
    变体2:
    变体3:
    """
    variants = llm.generate(prompt)
    return [original_query] + variants

# 使用多个变体检索, 合并结果
query_variants = expand_query("如何优化Transformer模型?", llm)

all_results = []
for variant in query_variants:
    results = vectorstore.similarity_search(variant, k=2)
    all_results.extend(results)

# 去重
unique_results = deduplicate(all_results)

```

### 策略2: 假设性文档嵌入 (HyDE)

```
def hyde_retrieval(query, llm, vectorstore):
    """
    HyDE: 先生成假设性答案，用答案去检索
    """
    # 步骤1: 生成假设性文档
    generate_prompt = f"""
    请为以下问题生成一个详细的答案（假设你知道答案）：

    问题: {query}

    答案:
    """

    hypothetical_answer = llm.generate(generate_prompt)

    # 步骤2: 用假设性答案作为查询
    results = vectorstore.similarity_search(hypothetical_answer, k=5)

    return results
```

### 8.6.3 多跳检索 (Multi-hop Retrieval)

对于复杂问题，多次检索。

```
def multi_hop_retrieval(query, vectorstore, llm, max_hops=3):
    """
    多跳检索
    """
    context = []
    current_query = query

    for hop in range(max_hops):
        # 检索
        docs = vectorstore.similarity_search(current_query, k=3)
        context.extend(docs)

        # 判断是否需要继续检索
        check_prompt = f"""
    问题: {query}

    当前已检索到的信息:
    {format_docs(context)}

    是否需要更多信息？如需要，请提出下一个检索查询；如不需要，回答"不需要"。

    下一个检索查询或"不需要":
```

```

"""
    next_action = llm.generate(check_prompt)

    if "不需要" in next_action:
        break

    current_query = next_action

return context

```

## 8.7 生成阶段优化

### 8.7.1 Prompt模板设计

```

RAG_PROMPT_TEMPLATE = """
你是一个专业的AI助手。请基于以下提供的上下文信息回答用户问题。

```

重要规则：

1. 只使用上下文中的信息回答，不要编造
2. 如果上下文中没有相关信息，明确说"根据提供的信息无法回答"
3. 引用信息时注明来源（[来源X]）

上下文信息：

{context}

用户问题：{question}

请回答：

```

"""

```

```

def generate_answer(query, retrieved_docs, llm):
    """
    基于检索结果生成答案
    """
    # 格式化检索到的文档
    context = ""
    for i, doc in enumerate(retrieved_docs, 1):
        source = doc.metadata.get('source', '未知')
        context += f"[来源{i}]: {source}]\n{doc.page_content}\n\n"

    # 构造prompt
    prompt = RAG_PROMPT_TEMPLATE.format(
        context=context,
        question=query
    )

```

```

# 生成答案
answer = llm.generate(prompt)

return answer, retrieved_docs # 返回答案和来源

```

### 8.7.2 答案验证

```

def verify_answer(question, answer, context, llm):
    """
    验证答案是否基于上下文
    """
    verify_prompt = f"""
    请验证以下答案是否完全基于提供的上下文。

    问题: {question}

    上下文: {context}

    答案: {answer}

    验证结果（是/否）：
    如果答案包含上下文中没有的信息，请指出：
    """
    verification = llm.generate(verify_prompt)

    return verification

```

### 8.7.3 引用生成

让模型在答案中标注引用。

```

CITATION_PROMPT = """
请回答以下问题，并在答案中使用[1]、[2]等标注引用的来源。

文档：
[1] {doc1}
[2] {doc2}
[3] {doc3}

问题: {question}

要求：
- 每个事实都要标注来源
- 格式示例: "Transformer架构由Google提出[1]，使用了自注意力机制[2]。"

```

答案:

"""

```
def generate_with_citation(query, docs, llm):
    """
    生成带引用的答案
    """
    doc_dict = {f"doc{i+1}": doc.page_content for i, doc in enumerate(docs)}
    prompt = CITATION_PROMPT.format(question=query, **doc_dict)

    answer = llm.generate(prompt)

    # 后处理: 将[1]替换为实际链接
    for i, doc in enumerate(docs, 1):
        source = doc.metadata.get('source', '')
        answer = answer.replace(f"[{i}]", f"[{i}]({source})")

    return answer
```

## 8.8 高级RAG模式

### 8.8.1 Self-RAG

模型自己决定何时检索。

```
def self_rag(query, vectorstore, llm):
    """
    Self-RAG: 模型决定是否需要检索
    """
    # 步骤1: 判断是否需要检索
    need_retrieval_prompt = f"""
    问题: {query}

    这个问题需要外部知识吗? (是/否)
    """
    need_retrieval = llm.generate(need_retrieval_prompt)

    if "否" in need_retrieval:
        # 直接回答
        answer = llm.generate(query)
    else:
        # 检索后回答
        docs = vectorstore.similarity_search(query, k=3)
        context = format_docs(docs)
```

```

        rag_prompt = f"""
上下文: {context}

问题: {query}

答案:
"""

        answer = llm.generate(rag_prompt)

    return answer

```

## 8.8.2 Corrective RAG

检测到答案不好时，重新检索。

```

def corrective_rag(query, vectorstore, llm, max_iterations=3):
    """
    纠正式RAG: 迭代优化
    """
    for iteration in range(max_iterations):
        # 检索
        docs = vectorstore.similarity_search(query, k=5)

        # 生成答案
        answer = generate_answer(query, docs, llm)

        # 自我评估
        eval_prompt = f"""
问题: {query}
答案: {answer}

请评估答案质量（1-10分）：
如果分数<7，请说明需要什么额外信息：
"""

        evaluation = llm.generate(eval_prompt)
        score = extract_score(evaluation)

        if score >= 7:
            return answer

        # 根据反馈改进查询
        feedback = extract_feedback(evaluation)
        query = improve_query(query, feedback, llm)

```

```
return answer
```

### 8.8.3 Graph RAG

基于知识图谱的RAG。

```
class GraphRAG:
    def __init__(self, knowledge_graph, vectorstore, llm):
        self.kg = knowledge_graph
        self.vectorstore = vectorstore
        self.llm = llm

    def retrieve(self, query):
        """
        结合向量检索和图谱检索
        """

        # 1. 向量检索
        vector_docs = self.vectorstore.similarity_search(query, k=3)

        # 2. 提取实体
        entities = self.extract_entities(query)

        # 3. 图谱扩展（找相关实体）
        related_entities = []
        for entity in entities:
            neighbors = self.kg.get_neighbors(entity, max_depth=2)
            related_entities.extend(neighbors)

        # 4. 将图谱信息转换为文本
        graph_context = self.format_graph_info(related_entities)

        # 5. 合并两种来源的信息
        combined_context = self.combine_contexts(vector_docs, graph_context)

        return combined_context

    def extract_entities(self, text):
        """
        从文本中提取实体
        """

        prompt = f"从以下文本中提取关键实体：\n{text}\n\n实体列表："
        entities = self.llm.generate(prompt)
        return parse_entities(entities)
```



## 8.9 RAG评估

### 8.9.1 评估指标

#### 1. 检索质量:

```
def evaluate_retrieval(query, retrieved_docs, ground_truth_docs):
    """
    评估检索质量
    """

    retrieved_ids = set(doc.metadata['id'] for doc in retrieved_docs)
    ground_truth_ids = set(doc.metadata['id'] for doc in ground_truth_docs)


    # 召回率
    recall = len(retrieved_ids & ground_truth_ids) / len(ground_truth_ids)

    # 精确率
    precision = len(retrieved_ids & ground_truth_ids) / len(retrieved_ids)

    # F1分数
    f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0

    # MRR (Mean Reciprocal Rank)
    for i, doc in enumerate(retrieved_docs, 1):
        if doc.metadata['id'] in ground_truth_ids:
            mrr = 1 / i
            break
    else:
        mrr = 0

    return {
        "recall": recall,
        "precision": precision,
        "f1": f1,
        "mrr": mrr
    }
```



#### 2. 生成质量:

```
def evaluate_generation(query, answer, ground_truth, retrieved_docs):
    """
    评估生成质量
    """

    metrics = {}
```

```

# 1. 忠实度 (Faithfulness) : 答案是否基于检索内容
context = " ".join([doc.page_content for doc in retrieved_docs])
faithfulness_prompt = f"""
上下文: {context}
答案: {answer}

```

答案中的每个事实是否都能在上下文中找到? (是/否)

```

"""

```

```

# 使用LLM或NLI模型评估

```

```

# 2. 相关性 (Relevance) : 答案是否回答了问题

```

```

relevance_prompt = f"""

```

```

问题: {query}

```

```

答案: {answer}

```

答案是否充分回答了问题? (1-5分)

```

"""

```

```

# 3. 准确性 (Accuracy) : 与ground truth对比

```

```

# 使用ROUGE、BLEU等指标

```

```

from rouge import Rouge

```

```

rouge = Rouge()

```

```

rouge_scores = rouge.get_scores(answer, ground_truth)[0]

```

```

metrics['rouge-1'] = rouge_scores['rouge-1']['f']

```

```

return metrics

```

## 8.9.2 端到端评估

```

class RAGEvaluator:
    def __init__(self, test_dataset):
        self.test_dataset = test_dataset # [(query, ground_truth, relevant_docs),

    def evaluate(self, rag_system):
        """
        全面评估RAG系统
        """
        results = {
            "retrieval": [],
            "generation": [],
        }

        for query, ground_truth, relevant_docs in self.test_dataset:

```

```

# 检索
retrieved = rag_system.retrieve(query)

# 评估检索
retrieval_metrics = evaluate_retrieval(query, retrieved, relevant_docs)
results["retrieval"].append(retrieval_metrics)

# 生成
answer = rag_system.generate(query, retrieved)

# 评估生成
generation_metrics = evaluate_generation(query, answer, ground_truth, r
results["generation"].append(generation_metrics)

# 计算平均指标
avg_results = {
    "avg_recall": np.mean([r["recall"] for r in results["retrieval"]]),
    "avg_precision": np.mean([r["precision"] for r in results["retrieval"]]),
    "avg_rouge": np.mean([r["rouge-1"] for r in results["generation"]]),
}

return avg_results

```

## 8.10 RAG实战案例

### 8.10.1 完整的RAG系统

```

class ProductionRAGSystem:
    def __init__(self, embedding_model, llm, vectorstore_path):
        self.embeddings = embedding_model
        self.llm = llm
        self.vectorstore = FAISS.load_local(vectorstore_path, self.embeddings)
        self.reranker = Reranker()

    def query(self, question, top_k=3, use_rerank=True):
        """
        完整的RAG查询流程
        """
        # 1. 查询预处理
        processed_query = self.preprocess_query(question)

        # 2. 检索候选文档
        candidates = self.vectorstore.similarity_search(processed_query, k=20)

        # 3. 重排序（可选）

```

```

if use_rerank:
    docs = self.reranker.rerank(processed_query, candidates, top_k=top_k)
else:
    docs = candidates[:top_k]

# 4. 生成答案
answer = self.generate_answer(question, docs)

# 5. 后处理
answer = self.postprocess_answer(answer)

# 6. 返回结果（包含来源）
return {
    "answer": answer,
    "sources": [
        {
            "content": doc.page_content[:200] + "...",
            "source": doc.metadata.get("source", ""),
            "score": doc.metadata.get("score", 0)
        }
        for doc in docs
    ]
}

```

```

def preprocess_query(self, query):
    """查询预处理"""
    # 拼写纠正、停用词移除等
    return query.strip()

def generate_answer(self, question, docs):
    """生成答案"""
    context = "\n\n".join([
        f"【文档{i+1}】\n{doc.page_content}"
        for i, doc in enumerate(docs)
    ])

```

```

    prompt = f"""

```

基于以下文档回答问题。如果文档中没有相关信息，请明确说明。

```
{context}
```

```
问题: {question}
```

```
答案:
```

```
"""
```

```

    return self.llm.generate(prompt)

```

```
def postprocess_answer(self, answer):  
    """答案后处理"""  
    # 去除多余空格、格式化等  
    return answer.strip()
```

### 8.10.2 流式输出RAG

```
async def stream_rag_response(query, vectorstore, llm):  
    """  
    流式输出RAG响应  
    """  
    # 检索  
    docs = vectorstore.similarity_search(query, k=3)  
    context = format_docs(docs)  
  
    prompt = RAG_PROMPT_TEMPLATE.format(context=context, question=query)  
  
    # 流式生成  
    async for chunk in llm.astream(prompt):  
        yield chunk
```

## 8.11 面试高频问题

Q1: RAG的主要挑战是什么？

答案要点：

#### 1. 检索质量：

- 语义gap（查询和文档表达不一致）
- 长尾查询效果差
- 多跳推理难

#### 2. 上下文利用：

- 噪声文档影响生成
- 上下文长度限制

#### 3. 响应时间：

- 检索耗时
- 需要权衡准确性和速度

#### 4. 成本：

- 向量存储成本
- 嵌入计算成本

Q2: 如何提升RAG的准确性?

策略清单:

阶段	优化方法
索引阶段	- 合理分块
	- 添加元数据
	- 数据清洗
检索阶段	- 混合检索
	- Reranking
	- 查询改写
生成阶段	- 优化prompt
	- 引用机制
	- 答案验证

Q3: Embedding模型如何选择?

决策树:

- 预算充足 && 对延迟不敏感?
- └ 是 → OpenAI API (text-embedding-3-large)

└ 否 → 继续
- 主要处理中文?
- └ 是 → bge-large-zh / m3e

└ 否 → sentence-transformers / bge-large-en

关键考虑:

- 语言支持 (中文vs英文)
- 部署方式 (API vs 本地)
- 性能要求 (准确性vs速度)
- 成本预算

Q4: 如何处理超长文档?

方案对比:

方案	适用场景	优缺点
分块+检索	常规文档	<div><div>✔ 简单</div><div>✖ 可能丢失全局信息</div></div>
层次化检索	结构化文档	<div><div>✔ 保留结构</div><div>✖ 实现复杂</div></div>
摘要+原文	长报告	<div><div>✔ 兼顾全局和细节</div><div>✖ 需要生成摘要</div></div>

方案	适用场景	优缺点
Map-Reduce	极长文档	✅ 可扩展 ❌ 多次调用

Q5: RAG vs Long Context模型？

对比：

维度	RAG	Long Context (128K+)
知识更新	实时更新	需要重新训练
成本	检索成本	高Token成本
准确性	依赖检索	可能迷失在长上下文中
可解释性	高（有引用）	低
适用场景	知识密集型	长文档理解

结论： 两者互补，可以结合使用。

## 8.12 本章小结

本章全面介绍了RAG技术：

✅ **核心流程**：文档处理 → Embedding → 检索 → 生成   ✅ **关键技术**：分块策略、向量数据库、混合检索、Reranking   ✅ **高级模式**：Self-RAG、Corrective RAG、Graph RAG   ✅ **工程实践**：系统设计、评估方法、性能优化

实践要点：

1. 文档分块是基础，需要根据内容类型选择策略
2. Reranking能显著提升准确性
3. 提示工程在RAG中同样重要
4. 建立完善的评估体系

**下一章预告：** 第9章将讲解Agent系统，让模型拥有使用工具和规划能力。