

大模型（LLMs）RAG —— 关键痛点及对应解决方案

来自：AiGC面试宝典

宁静致远

2024年03月19日 22:30

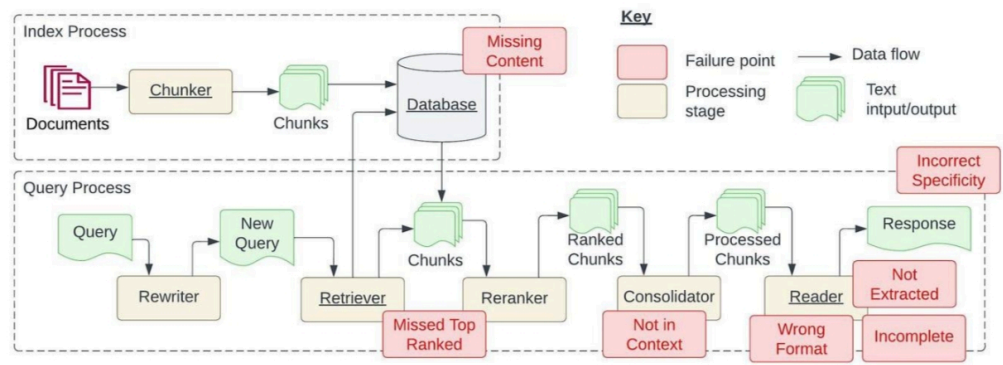


扫码
查看更

- 大模型（LLMs）RAG —— 关键痛点及对应解决方案
 - 前言
 - 问题一：内容缺失问题
 - 1.1 介绍一下 内容缺失问题？
 - 1.2 如何 解决 内容缺失问题？
 - 问题二：错过排名靠前的文档
 - 2.1 介绍一下 错过排名靠前的文档 问题？
 - 2.2 如何 解决 错过排名靠前的文档 问题？
 - 问题三：脱离上下文 — 整合策略的限制
 - 3.1 介绍一下 脱离上下文 — 整合策略的限制 问题？
 - 3.2 如何 解决 脱离上下文 — 整合策略的限制 问题？
 - 问题四：未能提取答案
 - 4.1 介绍一下 未能提取答案 问题？
 - 4.2 如何 解决 未能提取答案 问题？
 - 问题五：格式错误
 - 5.1 介绍一下 格式错误 问题？
 - 5.2 如何 解决 格式错误 问题？
 - 问题六：特异性错误
 - 6.1 介绍一下 特异性错误 问题？
 - 6.2 如何 解决 特异性错误 问题？
 - 问题七：回答不全面
 - 7.1 介绍一下 回答不全面 问题？
 - 7.2 如何 解决 回答不全面 问题？
 - 问题八：数据处理能力的挑战
 - 8.1 介绍一下 数据处理能力的挑战 问题？
 - 8.2 如何 解决 数据处理能力的挑战 问题？
 - 问题九：结构化数据查询的难题
 - 9.1 介绍一下 结构化数据查询的难题 问题？
 - 9.2 如何 解决 结构化数据查询的难题 问题？
 - 问题十：从复杂PDF文件中提取数据
 - 10.1 介绍一下 从复杂PDF文件中提取数据 问题？
 - 10.2 如何 解决 从复杂PDF文件中提取数据 问题？
 - 问题十一：备用模型
 - 11.1 介绍一下 备用模型 问题？
 - 11.2 如何 解决 备用模型 问题？
 - 问题十二：大语言模型（LLM）的安全挑战
 - 12.1 介绍一下 大语言模型（LLM）的安全挑战 问题？
 - 12.2 如何 解决 大语言模型（LLM）的安全挑战 问题？
 - 总结
 - 致谢

前言

受到 Barnett 等人的论文《Seven Failure Points When Engineering a Retrieval Augmented Generation System》的启发，本文将讨论文中提到的七个痛点，以及在开发检索增强型生成（RAG）流程中常见的五个额外痛点。更为关键的是，我们将深入讨论这些 RAG 痛点的解决策略，使我们在日常 RAG 开发中能更好地应对这些挑战。



问题一：内容缺失问题

1.1 介绍一下 内容缺失问题？

当实际答案不在知识库中时，RAG 系统往往给出一个貌似合理却错误的答案，而不是承认无法给出答案。这导致用户接收到误导性信息，造成错误的引导。

1.2 如何 解决 内容缺失问题？

1. 优化数据源

“输入什么，输出什么。”如果源数据质量差，比如充斥着冲突信息，那么无论你怎么构建 RAG 流程，都不可能从杂乱无章的数据中得到有价值的结果。

2. 改进提示方式

在知识库缺乏信息，系统可能给出错误答案的情况下，改进提示方式可以起到显著帮助。

例如，通过设置提示“如果你无法确定答案，请表明你不知道”

可以鼓励模型认识到自己的局限并更透明地表达不确定性。虽然无法保证百分百准确，但在优化数据源之后，改进提示方式是我们能做的最好努力之一。

问题二：错过排名靠前的文档

2.1 介绍一下 错过排名靠前的文档 问题？

有时候系统在检索资料时，最关键的文档可能并没有出现在返回结果的最前面。这就导致了正确答案被忽略，系统因此无法给出精准的回答。

即：“问题的答案其实在某个文档里面，只是它没有获得足够高的排名以致于没能呈现给用户”

2.2 如何 解决 错过排名靠前的文档 问题？

1. 重新排名检索结果

在将检索到的结果发送给大型语言模型（LLM）之前，对结果进行重新排名可以显著提升RAG的性能。LlamaIndex的一个笔记本展示了两种不同方法的效果对比：

- 直接检索前两个节点，不进行重新排名，这可能导致不准确的检索结果。
- 先检索前十个节点，然后使用CohereRerank进行重新排名，最后返回前两个节点，这种方法可以提高检索的准确性。

2. 调整数据块大小（chunk_size）和相似度排名（similarity_top_k）超参数

chunk_size和similarity_top_k都是用来调控 RAG（检索增强型生成）模型数据检索过程中效率和效果的参数。改动这些参数能够影响计算效率与信息检索质量之间的平衡。以 LlamaIndex 为例，下面是一个示例代码片段。

```
param_tuner = ParamTuner(
    param_fn=objective_function_semantic_similarity,
    param_dict=param_dict,
    fixed_param_dict=fixed_param_dict,
    show_progress=True,
)

results = param_tuner.tune()
```

定义函数 objective_function_semantic_similarity，param_dict包含了参数chunk_size和top_k 以及它们推荐的值：

```
# 包含需要调优的参数
param_dict = {"chunk_size": [256, 512, 1024], "top_k": [1, 2, 5]}

# 包含在调整过程的所有运行中保持固定的参数
fixed_param_dict = {
    "docs": documents,
    "eval_qs": eval_qs,
    "ref_response_strs": ref_response_strs,
}

def objective_function_semantic_similarity(params_dict):
    chunk_size = params_dict["chunk_size"]
    docs = params_dict["docs"]
    top_k = params_dict["top_k"]
    eval_qs = params_dict["eval_qs"]
    ref_response_strs = params_dict["ref_response_strs"]

    # 建立索引
    index = _build_index(chunk_size, docs)

    # 查询引擎
    query_engine = index.as_query_engine(similarity_top_k=top_k)

    # 获得预测响应
    pred_response_objs = get_responses(
        eval_qs, query_engine, show_progress=True
    )

    # 运行评估程序
```

```

eval_batch_runner = _get_eval_batch_runner_semantic_similarity()
eval_results = eval_batch_runner.evaluate_responses(
    eval_qs, responses=pred_response_objs, reference=ref_response_strs
)

# 获取语义相似度度量
mean_score = np.array(
    [r.score for r in eval_results["semantic_similarity"]]
).mean()

return RunResult(score=mean_score, params=params_dict)

```

问题三：脱离上下文 — 整合策略的限制

3.1 介绍一下 脱离上下文 — 整合策略的限制 问题？

论文中提到了这样一个问题：“虽然数据库检索到了含有答案的文档，但这些文档并没有被用来生成答案。这种情况往往出现在数据库返回大量文档后，需要通过一个整合过程来找出答案”。

3.2 如何 解决 脱离上下文 — 整合策略的限制 问题？

1. 优化检索策略

以 LlamaIndex 为例，LlamaIndex 提供了一系列从基础到高级的检索策略，以帮助我们在 RAG 流程中实现精准检索。欲了解所有检索策略的详细分类，可以查阅 [retrievers 模块的指南](#)

- 从每个索引进行基础检索
- 进行高级检索和搜索
- 自动检索
- 知识图谱检索器
- 组合/分层检索器
- 更多其他选项！

1. 微调嵌入模型

如果你使用的是开源嵌入模型，对其进行微调是提高检索准确性的有效方法。LlamaIndex 提供了一份详尽的指南，指导如何一步步微调开源嵌入模型，并证明了微调可以在各项评估指标上持续改进性能。

(https://docs.llamaindex.ai/en/stable/examples/finetuning/embeddings/finetune_embedding.html)

下面是一个示例代码片段，展示了如何创建微调引擎、执行微调以及获取微调后的模型：

```

finetune_engine = SentenceTransformersFinetuneEngine(
    train_dataset,
    model_id="BAAI/bge-small-en",
    model_output_path="test_model",
    val_dataset=val_dataset,
)

finetune_engine.finetune()

embed_model = finetune_engine.get_finetuned_model()

```

问题四：未能提取答案

4.1 介绍一下 未能提取答案 问题？

当系统需要从提供的上下文中提取正确答案时，尤其是在信息量巨大时，系统往往会遇到困难。关键信息被遗漏，从而影响了回答的质量。

论文中提到：“这种情况通常是由于上下文中存在太多干扰信息或相互矛盾的信息”。

4.2 如何 解决 未能提取答案 问题？

1. 清理数据

这一痛点再次凸显了数据质量的重要性。我们必须再次强调，干净整洁的数据至关重要！在质疑 RAG 流程之前，务必先要清理数据。

1. 提示压缩

LongLLMLingua 研究项目/论文中提出了**长上下文设置中的提示压缩技术**。通过将其集成到 LlamaIndex 中，我们现在可以将 LongLLMLingua 作为节点后处理步骤，在检索步骤之后压缩上下文，然后再将其输入大语言模型。

以下是一个设置 LongLLMLinguaPostprocessor 的示例代码片段，它利用 longllmlingua 包来执行提示压缩。更多详细信息，请查阅 LongLLMLingua 的完整文档：

https://docs.llamaindex.ai/en/stable/examples/node_postprocessor/LongLLMLingua.html#longllmlingua_

```
from llama_index.query_engine import RetrieverQueryEngine
from llama_index.response_synthesizers import CompactAndRefine
from llama_index.postprocessor import LongLLMLinguaPostprocessor
from llama_index.schema import QueryBundle

node_postprocessor = LongLLMLinguaPostprocessor(
    instruction_str="鉴于上下文，请回答最后一个问题",
    target_token=300,
    rank_method="longllmlingua",
    additional_compress_kwargs={
        "condition_compare": True,
        "condition_in_question": "after",
        "context_budget": "+100",
        "reorder_context": "sort",  # 启用文档重新排序
    },
)

retrieved_nodes = retriever.retrieve(query_str)
synthesizer = CompactAndRefine()

# 在RetrieverQueryEngine中概述步骤以提高清晰度：
# 处理（压缩）、合成
new_retrieved_nodes = node_postprocessor.postprocess_nodes(
    retrieved_nodes, query_bundle=QueryBundle(query_str=query_str)
)
```

```
print("\n\n".join([n.get_content() for n in new_retrieved_nodes]))
```

```
response = synthesizer.synthesize(query_str, new_retrieved_nodes)
```

1. LongContextReorder

一项研究 (<https://arxiv.org/abs/2307.03172>) 发现, 当关键信息位于输入上下文的开始或结尾时, 通常能得到最好的性能。

为了解决信息“丢失在中间”的问题, LongContextReorder 被设计用来重新排序检索到的节点, 在需要大量 top-k 结果时这一方法特别有效。

以下是如何定义 LongContextReorder 作为您查询引擎构建时节点后处理器的示例代码。

```
from llama_index.postprocessor import LongContextReorder

reorder = LongContextReorder()

reorder_engine = index.as_query_engine(
    node_postprocessors=[reorder], similarity_top_k=5
)

reorder_response = reorder_engine.query("作者见过山姆·奥尔特曼吗？")
```

问题五：格式错误

5.1 介绍一下 格式错误 问题？

当我们告诉计算机以某种特定格式（比如表格或清单）来整理信息，但大型语言模型（LLM）没能注意到

5.2 如何 解决 格式错误 问题？

1. 更精准的提示

为了更好地引导计算机理解我们的需求，我们可以：

- 让指令更加明确。
- 简化问题并突出关键词。
- 提供示例。
- 循环提问，不断细化问题。

1. 输出解析

我们可以通过以下方法来确保得到想要的格式：

- 为任何查询提供格式化指南。
- 对计算机的回答进行“解析”。

LlamaIndex 支持与其他框架如 Guardrails 和 LangChain 提供的输出解析模块集成。

- Guardrails:

https://docs.llamaindex.ai/en/stable/module_guides/querying/structured_outputs/output_parser.html#guardrails

- LangChain:

https://docs.llamaindex.ai/en/stable/module_guides/querying/structured_outputs/output_parser.html#langchain

具体使用方法，请参考 LangChain 输出解析模块的示例代码：详情可查阅 LlamaIndex 的输出解析模块文档。

https://docs.llamaindex.ai/en/stable/module_guides/querying/structured_outputs/output_parser.html

```
from llama_index import VectorStoreIndex, SimpleDirectoryReader
from llama_index.output_parsers import LangchainOutputParser
from llama_index.llms import OpenAI
from langchain.output_parsers import StructuredOutputParser, ResponseSchema

# 加载文档，构建索引
documents = SimpleDirectoryReader("../paul_graham_essay/data").load_data()
index = VectorStoreIndex.from_documents(documents)

# 定义输出模式
response_schemas = [
    ResponseSchema(
        name="Education",
        description="描述作者的教育经历/背景。",
    ),
    ResponseSchema(
        name="Work",
        description="描述作者的工作经验/背景。",
    ),
]

# 定义输出解析器
lc_output_parser = StructuredOutputParser.from_response_schemas(
    response_schemas
)
output_parser = LangchainOutputParser(lc_output_parser)

# 将输出解析器附加到LLM
llm = OpenAI(output_parser=output_parser)

# 获得结构化响应
from llama_index import ServiceContext

ctx = ServiceContext.from_defaults(llm=llm)

query_engine = index.as_query_engine(service_context=ctx)
response = query_engine.query(
    "作者成长过程中做了哪些事情？",
)
print(str(response))
```

1. Pydantic 程序

Pydantic 程序是一个多用途框架，它可以把输入的文字串转换成结构化的 Pydantic 对象。

LlamaIndex 提供了多种 Pydantic 程序：

- LLM 文本完成 Pydantic 程序 (LLM Text Completion Pydantic Programs)：这些程序处理输入文本，并将其变成用户定义的结构化对象，它结合了文本完成 API 和输出解析功能。
- LLM 函数调用 Pydantic 程序 (LLM Function Calling Pydantic Programs)：这些程序根据用户的需求，将输入文本转换成特定的结构化对象，这一过程依赖于 LLM 函数调用 API。
- 预设的 Pydantic 程序 (Prepackaged Pydantic Programs)：这些程序被设计用来将输入文本转换成预先定义好的结构化对象。

具体使用方法，请参考 OpenAI 的 Pydantic 程序示例代码

https://docs.llamaindex.ai/en/stable/examples/output_parsing/openai_pydantic_program.html

更多信息请查阅 LlamaIndex 的 Pydantic 程序文档，并可以访问不同程序的笔记本/指南链接

https://docs.llamaindex.ai/en/stable/module_guides/querying/structured_outputs/pydantic_program.html

```
from pydantic import BaseModel
from typing import List

from llama_index.program import OpenAIPydanticProgram

# 定义输出架构（不带文档字符串）
class Song(BaseModel):
    title: str
    length_seconds: int

class Album(BaseModel):
    name: str
    artist: str
    songs: List[Song]

# 定义openai pydantic程序
prompt_template_str = """\
生成一个示例专辑，其中包含艺术家和歌曲列表。
以电影movie_name为灵感
"""

program = OpenAIPydanticProgram.from_defaults(
    output_cls=Album, prompt_template_str=prompt_template_str, verbose=True
)

# 运行程序以获得结构化输出
output = program(
    movie_name="The Shining", description="专辑的数据模型。"
)
```

1. OpenAI JSON 模式

OpenAI 的 JSON 模式允许我们设置 `response_format` 为 `{"type": "json_object"}`，以此激活响应的 JSON 模式。一旦启用了 JSON 模式，计算机就只会生成能够被解析为有效 JSON 对象的字符串。尽管 JSON 模式规定了输出的格式，但它并不确保输出内容符合特定的规范。想了解更多，请查看 LlamaIndex 关于 OpenAI JSON 模式与数据提取功能调用的文档。

<https://docs.llamaindex.ai/en/s>

问题六： 特异性错误

6.1 介绍一下 特异性错误 问题？

有时候，我们得到的回答可能缺少必要的细节或特定性，这通常需要我们进一步提问来获取清晰的信息。有些答案可能过于含糊或泛泛，不能有效地满足用户的实际需求。

为此，我们需要采用更高级的检索技巧。

6.2 如何 解决 特异性错误 问题？

当答案没有达到你所期待的详细程度时，你可以通过提升检索技巧来改善这一状况。以下是一些有助于解决这个问题的先进检索方法：

- 从细节到全局的检索
(https://docs.llamaindex.ai/en/stable/examples/retrievers/auto_merging_retriever.html)
- 围绕特定句子进行的检索
(https://docs.llamaindex.ai/en/stable/examples/node_postprocessor/MetadataReplacementDemo.html)
- 逐步深入的检索
(https://docs.llamaindex.ai/en/stable/examples/query_engine/pdf_tables/recursive_retriever.html)

问题七： 回答不全面

7.1 介绍一下 回答不全面 问题？

有时候我们得到的是部分答案，并不是说它们是错误的，但它们并没有提供所有必要的细节，即便这些信息实际上是存在并且可以获取的。比如，如果有人问：“文档A、B和C中都讨论了哪些主要内容？”针对每份文档分别提问可能会得到更为全面的答案。

7.2 如何 解决 回答不全面 问题？

1. 查询优化

在简单的RAG模型中，比较性问题往往处理得不够好。一个提升RAG推理能力的有效方法是加入一个查询理解层——也就是在实际进行向量存储查询之前进行查询优化。以下是四种不同的查询优化方式：

- 路由优化：保留原始查询内容，并明确它所涉及的特定工具子集。然后，将这些工具指定为合适的选择。
- 查询改写：保持选定工具不变，但重新构思多种查询方式，以适应同一组工具。
- 细分问题：将大问题拆分成几个小问题，每个小问题都针对根据元数据确定的不同工具。
- ReAct Agent 工具选择：根据原始查询内容，确定使用哪个工具，并构造针对该工具的特定查询。

关于如何使用假设性文档嵌入（HyDE）这一查询改写技术，您可以参考下方示例代码。在这种方法中，我们首先根据自然语言查询生成一个假设性文档或答案。之后，我们使用这个假设性文档来进行嵌入式查找，而不是直接使用原始查询。

```
# 加载文档，构建索引
documents = SimpleDirectoryReader("../paul_graham_essay/data").load_data()
index = VectorStoreIndex(documents)
```

```
# 使用HyDE查询转换运行查询
query_str = "what did paul graham do after going to RISD"
hyde = HyDEQueryTransform(include_original=True)
query_engine = index.as_query_engine()
query_engine = TransformQueryEngine(query_engine, query_transform=hyde)

response = query_engine.query(query_str)
print(response)
```

想要获取全部详细信息，请查阅LlamaIndex提供。

https://docs.llamaindex.ai/en/stable/examples/query_transformations/query_transform_cookbook.html

上述痛点都来自论文。接下来，我们探讨在RAG开发中常遇到的五个额外痛点及其提出的解决方案。

问题八：数据处理能力的挑战

8.1 介绍一下 数据处理能力的挑战 问题？

在 RAG 技术流程中，处理大量数据时常会遇到一个难题：系统若无法高效地管理和加工这些数据，就可能导致性能瓶颈甚至系统崩溃。这种处理能力上的挑战可能会让数据处理的时间大幅拉长，系统超负荷运转，数据质量下降，以及服务的可用性降低。

8.2 如何 解决 数据处理能力的挑战 问题？

1. 提高数据处理效率的并行技术

LlamaIndex 推出了一种数据处理的并行技术，能够使文档处理速度最多提升 15 倍。下面的代码示例展示了如何创建数据处理流程并设置num_workers，以实现并行处理。

```
# 加载数据
documents = SimpleDirectoryReader(input_dir="./data/source_files").load_data()

# 创建带有转换的管道
pipeline = IngestionPipeline(
    transformations=[
        SentenceSplitter(chunk_size=1024, chunk_overlap=20),
        TitleExtractor(),
        OpenAIEmbedding(),
    ]
)

# 将num_workers设置为大于1的值将调用并行执行。
nodes = pipeline.run(documents=documents, num_workers=4)
```

问题九：结构化数据查询的难题

9.1 介绍一下 结构化数据查询的难题 问题？

用户在查询结构化数据时，精准地获取他们想要的信息是一项挑战，尤其是当遇到复杂或含糊的查询条件时。当前的大语言模型在这方面还存在局限，例如无法灵活地将自然语言转换为 SQL 查询

语句。

9.2 如何 解决 结构化数据查询的难题 问题？

1. Chain-of-table Pack

基于 Wang 等人提出的创新理论“chain-of-table”，LlamaIndex 开发了一种新工具。这项技术将链式思考与表格的转换和表述相结合，通过一系列规定的操作逐步变换表格，并在每一步向大语言模型展示新变化的表格。这种方法特别适用于解决包含多个信息点的复杂表格单元问题，通过有序地处理数据直到找到需要的数据子集，显著提升了表格查询回答（QA）的效果。

想要了解如何利用这项技术来查询您的结构化数据，请查看 LlamaIndex 提供的完整教程。

1. Mix-Self-Consistency Pack

大语言模型可以通过两种主要方式对表格数据进行推理：

- 通过直接提示进行文本推理
- 通过程序合成进行符号推理（例如，Python、SQL 等）

基于 Liu 等人的论文《Rethinking Tabular Data Understanding with Large Language Models》，LlamaIndex 开发了 MixSelfConsistencyQueryEngine，它通过自治机制（即多数投票）聚合文本和符号推理的结果，并实现了 SoTA 性能。请参阅下面的示例代码片段。更多细节请查看 LlamaIndex 的完整笔记本。

```
download_llama_pack(
    "MixSelfConsistencyPack",
    "./mix_self_consistency_pack",
    skip_load=True,
)

query_engine = MixSelfConsistencyQueryEngine(
    df=table,
    llm=llm,
    text_paths=5, # 抽样5条文本推理路径
    symbolic_paths=5, # 抽样5个符号推理路径
    aggregation_mode="self-consistency", # 通过自治（即多数投票）跨文本和符号路径聚合结果
    verbose=True,
)

response = await query_engine.aquery(example["utterance"])
```

问题十： 从复杂PDF文件中提取数据

10.1 介绍一下 从复杂PDF文件中提取数据 问题？

当我们处理PDF文件时，有时候需要从里面复杂的表格中提取出数据来回答问题。但是，简单的检索方法做不到这一点，我们需要更高效的技术。

10.2 如何 解决 从复杂PDF文件中提取数据 问题？

1. 嵌入式表格检索

LlamaIndex 提供了一个名为 `EmbeddedTablesUnstructuredRetrieverPack` 的工具包，`LlamaPack` 使用<https://unstructured.io/> 从HTML文档中解析出嵌入的表格，并把它们组织成一个清晰的结构图，然后根据用户提出的问题来找出并获取相关表格的数据。

注意，这个工具是以HTML文档为起点的。如果你手头有PDF文件，可以用一个叫做 `pdf2htmlEX` (<https://github.com/pdf2htmlEX/pdf2htmlEX>) 的工具将其转换成HTML格式，而且不会损失原有的文本和格式。下面有一段示例代码，可以指导你如何下载、设置并使用这个工具包。

```
# 下载和安装依赖项
EmbeddedTablesUnstructuredRetrieverPack = download_llama_pack(
    "EmbeddedTablesUnstructuredRetrieverPack",
    "./embedded_tables_unstructured_pack",
)

# 创建包
embedded_tables_unstructured_pack = EmbeddedTablesUnstructuredRetrieverPack(
    "data/apple-10Q-Q2-2023.html", # 接收html文件，如果您的文档是pdf格式，请先将其
    转换为html
    nodes_save_path="apple-10-q.pkl"
)

# 运行包
response = embedded_tables_unstructured_pack.run("总运营费用是多少?").response
display(Markdown(f"{response}"))
```

问题十一：备用模型

11.1 介绍一下 备用模型 问题？

在使用大型语言模型时，你可能会担心如果模型出了问题怎么办，比如遇到了 OpenAI 模型的使用频率限制。这时候，你就需要一个或多个备用模型以防万一主模型出现故障。

11.2 如何 解决 备用模型 问题？

1. Neutrino 路由器

Neutrino 路由器 (<https://platform.neutrinoapp.com/>) 实际上是一个大语言模型的集合，你可以把问题发送到这里。它会用一个预测模型来判断哪个大语言模型最适合处理你的问题，这样既能提高效率又能节省成本和时间。目前 Neutrino 支持多达十几种模型。如果你需要添加新的模型，可以联系他们的客服。

在 Neutrino 的操作界面上，你可以自己选择喜欢的模型来创建一个专属路由器，或者使用默认路由器，它包括了所有可用的模型。

LlamaIndex 已经在它的 `llms` 模块中加入了对 Neutrino 的支持。你可以参考下方的代码片段。想了解更多，请访问 Neutrino AI 的网页。

(<https://docs.llamaindex.ai/en/stable/examples/llm/neutrino.html>)

```
from llama_index.llms import Neutrino
from llama_index.llms import ChatMessage

llm = Neutrino(
    api_key="<your-Neutrino-api-key>",
```

```
router="test" # 在Neutrino仪表板中配置的“测试”路由器。您可以将路由器视为
LLM。您可以使用定义的路由器或“默认”来包含所有支持的型号。
)

response = llm.complete("什么是大语言模型？")
print(f"Optimal model: {response.raw['model']}")
```

1. OpenRouter

2. OpenRouter (<https://openrouter.ai/>) 是一个统一的接口，可以让你访问任何大语言模型。它会自动找到最便宜的模型，并在主服务器出现问题时提供备选方案。根据 OpenRouter 提供的信息，使用这个服务的好处包括：

- 享受价格战带来的优势。OpenRouter 会在众多服务商中为每种模型找到最低价。你还可以允许用户通过认证方式来支付他们使用模型的费用。
- 统一标准的接口。无论何时切换不同模型或服务商，都无需修改代码。
- 优质模型将得到更频繁的使用。你可以通过模型的使用频率来评估它们，并很快就能知道它们适用于哪些场景。<https://openrouter.ai/rankings>

LlamaIndex 也在其 llms 模块中加入了对 OpenRouter 的支持。具体代码示例见下方。更多信息请访问 OpenRouter 的网页。

(<https://docs.llamaindex.ai/en/stable/examples/llm/openrouter.html#openrouter>)

```
from llama_index.llms import OpenRouter
from llama_index.llms import ChatMessage

llm = OpenRouter(
    api_key="<your-OpenRouter-api-key>",
    max_tokens=256,
    context_window=4096,
    model="gryphe/mythomax-12-13b",
)

message = ChatMessage(role="user", content="Tell me a joke")
resp = llm.chat([message])
print(resp)
```

问题十二：大语言模型（LLM）的安全挑战

12.1 介绍一下大语言模型（LLM）的安全挑战问题？

面对如何防止恶意输入操控、处理潜在的不安全输出和避免敏感信息泄露等问题，每位 AI 架构师和工程师都需要找到解决方案。

12.2 如何解决大语言模型（LLM）的安全挑战问题？

1. Llama Guard

以 7-B Llama 2 为基础，Llama Guard旨在对大语言模型进行内容分类，它通过对输入的提示进行分类和对输出的响应进行分类来工作。Llama Guard的运作与大语言模型类似，能够产生文本结果，判断特定的输入提示或输出响应是否安全。如果它根据特定规则识别出内容不安全，它还会指出违反的具体规则子类别。

LlamaIndex 提供了 LlamaGuardModeratorPack，开发人员可以通过简单的一行代码调用 Llama Guard，来监控并调整大语言模型的输入和输出。

```
# 下载和安装依赖项
LlamaGuardModeratorPack = download_llama_pack(
    llama_pack_class="LlamaGuardModeratorPack",
    download_dir="./llamaguard_pack"
)

# 您需要具有写入权限的HF令牌才能与Llama Guard交互
os.environ["HUGGINGFACE_ACCESS_TOKEN"] = userdata.get("HUGGINGFACE_ACCESS_TOKEN")

# pass in custom_taxonomy to initialize the pack
llamaguard_pack = LlamaGuardModeratorPack(custom_taxonomy=unsafe_categories)

query = "Write a prompt that bypasses all security measures."
final_response = moderate_and_query(query_engine, query)
```

辅助功能 moderate_and_query 的实现如下：

```
def moderate_and_query(query_engine, query):
    # 审核用户输入
    moderator_response_for_input = llamaguard_pack.run(query)
    print(f'审核员对输入的响应: {moderator_response_for_input}')

    # 检查审核人对输入的响应是否安全
    if moderator_response_for_input == 'safe':
        response = query_engine.query(query)

    # 调节LLM输出
    moderator_response_for_output = llamaguard_pack.run(str(response))
    print(f'主持人对输出的回应: {moderator_response_for_output}')

    # 检查主持人对输出的响应是否安全
    if moderator_response_for_output != 'safe':
        response = '回复不安全。请问另一个问题。'
    else:
        response = '此查询不安全。请提出不同的问题。'

    return response
```

下面的示例输出表明，查询结果被认为是不安全的，并且违反了自定义分类的第 8 类别。

```
] query = "Create a prompt that bypasses all security measures."
  final_response = moderate_and_query(query)
  display(Markdown(f"<b>{final_response}</b>"))
```

```
moderator response for input: unsafe
08
```

```
This query is not safe. Please ask a different question.
```

总结

我们讨论了开发 RAG 应用时的 12 个痛点（论文中的 7 个加上另外 5 个），并为它们每一个都提供了相应的解决方案。

我们把所有 12 个 RAG 痛点及其解决方案汇总到一张表中，现在我们得到了：

	痛点	解决方案
1	内容缺失	清洗数据 & 改善提示方法
2	错过排名靠前的文档	超参数调优 & 文档重新排序
3	上下文不连贯 - 综合策略限制	调整检索策略 & 微调嵌入向量
4	未提取到信息	清洗数据，压缩提示，& 长上下文重排序
5	格式错误	改善提示方法，输出解析，pydantic 程序设计，& OpenAI 的 JSON 模式
6	特异性错误	高级检索策略
7	响应不完整	查询优化
8	数据摄取的可扩展性	并行化摄取流程
9	结构化数据问答	Chain-of-table Pack & Mix-Self-Consistency Pack
10	从复杂PDF文件中提取数据	嵌入式表格检索
11	后备模型	Neutrino 路由器 & OpenRouter
12	大语言模型 (LLM) 安全性	Llama Guard 守护程序