```
 > Entering new LLMChain chain...
Prompt after formatting:
System: You are a chatbot having a conversation with a human.
Human: Hi there my friend
AI: Hello! How can I assist you today, my friend?
Human: Not too bad - how are you?

> Finished chain.

"I'm an AI chatbot, so I don't have feelings, but I'm here to help and chat with you! Is there
something specific you would like to talk about or any questions I can assist you with?"
```

通过上述结果可以看到，对话的历史记录都通过记忆传递给了 ChatModel。

6. 智能体

智能体的核心思想是使用大语言模型来选择要执行的一系列动作。在链中，操作序列是硬编码在代码中的。在智能体中，需要将大语言模型用作推理引擎，以确定要采取哪些动作，以及以何种顺序采取这些动作。智能体通过将大语言模型与动作列表结合，自动选择最佳的动作序列，从而实现自动化决策和行动。智能体可以用于许多不同类型的应用程序，例如自动化客户服务、智能家居等。LangChain 显示的智能体仅是智能体的简化方案。LangChain 中的智能体由如下几个核心组件构成。

- Agent：决定下一步该采取什么操作的类，由大语言模型和提示词驱动。提示词可以包括智能体的个性（有助于使其以某种方式做出回应）、智能体的背景上下文（有助于提供所要求完成的任务类型的更多上下文信息）、激发更好的推理的提示策略。
- Tools：智能体调用的工具。这里有两个重要的考虑因素，一是为智能体提供正确的工具访问权限；二是用对智能体最有帮助的方式描述工具。
- Toolkits：一组旨在一起使用以完成特定任务的工具集合，加载方便。通常一个工具集合中有 3 ∼ 5 个工具。
- AgentExecutor：智能体的运行空间，这是实际调用智能体并执行其所选操作的部分。除了 AgentExecutor 类,LangChain 还支持其他智能体运行空间,包括 Plan-and-execute Agent、BabyAGI、AutoGPT 等。

7. 回调

LangChain 提供了回调系统，允许连接到大语言模型应用程序的各个阶段。这对于日志记录、监控、流式处理和其他任务处理非常有用。可以通过使用 API 中提供的 callbacks 参数订阅这些事件。CallbackHandlers 是实现 CallbackHandler 接口的对象，每个事件都可以通过一个方法订阅。当事件被触发时，CallbackManager 会调用相应事件所对应的处理程序，代码示例如下：

```python
class BaseCallbackHandler:
    """ 基本回调处理程序，可用于处理来自LangChain的回调 """

    def on_llm_start(
        self, serialized: Dict[str, Any], prompts: List[str], **kwargs: Any
    ) -> Any:
        """ 在LLM开始运行时运行 """

    def on_chat_model_start(
        self, serialized: Dict[str, Any], messages: List[List[BaseMessage]], **kwargs: Any
    ) -> Any:
        """ 在聊天模型开始运行时运行 """

    def on_llm_new_token(self, token: str, **kwargs: Any) -> Any:
        """ 在新的LLM词元上运行，仅在启用了流式处理时可用 """

    def on_llm_end(self, response: LLMResult, **kwargs: Any) -> Any:
        """ 在LLM结束运行时运行 """

    def on_llm_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """ 在LLM出现错误时运行 """

    def on_chain_start(
        self, serialized: Dict[str, Any], inputs: Dict[str, Any], **kwargs: Any
    ) -> Any:
        """ 在链开始运行时运行 """

    def on_chain_end(self, outputs: Dict[str, Any], **kwargs: Any) -> Any:
        """ 在链结束运行时运行 """

    def on_chain_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """ 在链出现错误时运行 """

    def on_tool_start(
        self, serialized: Dict[str, Any], input_str: str, **kwargs: Any
    ) -> Any:
        """ 在工具开始运行时运行 """

    def on_tool_end(self, output: str, **kwargs: Any) -> Any:
        """ 在工具结束运行时运行 """

    def on_tool_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """ 在工具出现错误时运行 """

    def on_text(self, text: str, **kwargs: Any) -> Any:
        """ 在任意文本上运行 """

    def on_agent_action(self, action: AgentAction, **kwargs: Any) -> Any:
        """ 在智能体动作上运行 """
```

LangChain 在 langchain/callbacks 模块中提供了一些内置的处理程序，其中最基本的处理程序是 StdOutCallbackHandler，它将所有事件记录到 stdout 中，代码示例如下：

```
from langchain.callbacks import StdOutCallbackHandler
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

handler = StdOutCallbackHandler()
llm = OpenAI()
prompt = PromptTemplate.from_template("1 + {number} = ")

# 构造函数回调
# 首先，在初始化链时显式设置StdOutCallbackHandler
chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])
chain.run(number=2)

# 使用详细模式标志。然后，使用verbose标志实现相同的结果
chain = LLMChain(llm=llm, prompt=prompt, verbose=True)
chain.run(number=2)

# 请求回调。最后，使用请求的callbacks实现相同的结果
chain = LLMChain(llm=llm, prompt=prompt)
chain.run(number=2, callbacks=[handler])
```

执行上述程序可以得到如下输出：

```
> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.

> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.

> Entering new LLMChain chain...
Prompt after formatting:
1 + 2 =

> Finished chain.

'\n\n3'
```

8. LangChain 检索增强实践

以下代码给出了利用搜索增强模型对话能力的智能体的实现：

```
from langchain.agents import Tool
from langchain.agents import AgentType
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.utilities import SerpAPIWrapper
from langchain.agents import initialize_agent

search = SerpAPIWrapper()
tools = [
    Tool(
        name = "Current Search",
        func=search.run,
        description="useful for when you need to answer questions about current events
                     or the current state of the world"
    ),
]

memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY, temperature=0)
agent_chain = initialize_agent(
    tools,
    llm,
    agent=AgentType.CHAT_CONVERSATIONAL_REACT_DESCRIPTION,
    verbose=True,
    memory=memory
)
```

注意，此处在 agent 类型选择时使用了“CHAT_CONVERSATIONAL_REACT_DESCRIPTION”，模型将使用 ReAct 逻辑生成。根据上面定义的智能体，使用如下调用方式：

```
agent_chain.run(input="what's my name?")
```

给出如下回复：

```
> Entering new AgentExecutor chain...
{
    "action": "Final Answer",
    "action_input": "Your name is Bob."
}

> Finished chain.


'Your name is Bob.'
```

如果换一种需要利用当前知识的用户输入，并给出如下调用方式：

```
agent_chain.run(input="whats the weather like in pomfret?")
```

智能体就会启动搜索工具，从而得到如下回复：

```
> Entering new AgentExecutor chain...
{
    "action": "Current Search",
    "action_input": "weather in pomfret"
}
Observation: Cloudy with showers. Low around 55F. Winds S at 5 to 10 mph.
             Chance of rain 60%. Humidity76%.
Thought:{
    "action": "Final Answer",
    "action_input": "Cloudy with showers. Low around 55F. Winds S at 5 to 10 mph.
                     Chance of rain 60%. Humidity76%."
}

> Finished chain.

'Cloudy with showers. Low around 55F. Winds S at 5 to 10 mph. Chance of rain 60%. Humidity76%.'
```

可以看到，模型采用 ReAct 的提示模式生成内容。通过上述两种不同的用户输入及相应的系统回复，可以看到智能体自动根据用户输入选择是否使用搜索工具。

8.4.3 智能体平台 Coze 实践

使用零代码或低代码平台构建大模型智能体是一种高效便捷的开发方式，适合缺乏编程经验的用户或需要快速验证概念（Proof of Concept，PoC）的场景。通过可视化界面、拖拽组件和预设

模板，用户无需编写代码，甚至完全不需要编程能力，即可完成智能体的设计、开发和部署。这类平台通常集成了预训练的大语言模型，并提供强大的工具支持，如知识库管理、对话流程设计和外部 API 集成，极大地简化了开发流程。需要注意的是，这种方式在定制化能力、性能优化和扩展性上存在一定局限，复杂场景或高性能需求场景下适应程度需要详细评估。

Coze（扣子）是一个大模型智能体开发平台，整合了插件、长短期记忆、工作流、卡片等丰富功能，能够以低门槛、快速搭建个性化或具备商业价值的智能体，并发布到豆包、飞书、网页等多种平台，实现全场景覆盖。通过模块化与高效的工具支持，Coze 帮助开发者快速构建、测试和部署智能体，实现复杂任务的自动化，同时提供强大的扩展和定制能力。其插件系统支持智能体与外部工具无缝对接，如数据库查询、第三方 API 调用、任务管理工具等，在多种环境中执行精准任务；长短期记忆功能让智能体在短期对话中保持上下文一致，并通过长期记忆存储重要信息，实现自然、智能的交互体验；工作流功能允许用户通过拖拽式界面快速设计任务逻辑，动态调用插件或执行复杂任务；卡片功能则为智能体提供了信息展示和互动的新形式，让用户在网页或移动端直观查看数据、流程和结果。

使用 Coze 平台可以通过以下简单的五个步骤就可以构造快速搭建一个“夸夸机器人”，并在多个平台提供对外服务。

步骤 1：创建一个智能体。在扣子平台创建智能体非常简单：登录后，点击页面左上角的“⊕”，输入智能体名称和功能介绍，并通过生成图标自动生成头像，或使用“AI 创建”功能，通过自然语言描述需求，由平台自动生成智能体。点击确认后，进入智能体编排页面。在这里，可以通过左侧人设与回复逻辑面板描述智能体的身份和任务；利用中间技能面板为智能体配置扩展能力；在右侧预览与调试面板中实时测试智能体，确保其功能和交互效果符合预期。

步骤 2：编写提示词。配置智能体的第一步是编写提示词，即定义智能体的人设与回复逻辑。这部分内容决定了智能体的基本人设，并持续影响其在所有会话中的回复效果。在设计提示词时，建议明确模型的角色、设计特定的语言风格，并限制回答范围，以确保对话内容符合用户的预期。例如，对于一个“夸夸机器人”，提示词可以设置为：

```
# 角色
你是一个充满正能量的赞美鼓励机器人，时刻用温暖的话语给予人们赞美和鼓励，让他们充满自信与动力。

## 技能 ### 技能 1：赞美个人优点
1. 当用户提到自己的某个特点或行为时，挖掘其中的优点进行赞美。
   回复示例：你真的很 [优点]，比如 [具体事例说明优点]。
2. 如果用户没有明确提到自己的特点，可以主动询问一些问题，了解用户后进行赞美。
   回复示例：我想先了解一下你，你觉得自己最近做过最棒的事情是什么呢？

### 技能 2：鼓励面对困难
1. 当用户提到遇到困难时，给予鼓励和积极的建议。回复示例：这确实是个挑战，
   但我相信你有足够的能力去克服它。你可以 [具体建议]。
2. 如果用户没有提到困难但情绪低落，可以询问是否有不开心的事情，然后给予鼓励。
   回复示例：你看起来有点不开心，是不是遇到什么事情了呢？不管怎样，你都很坚强，
   一定可以度过难关。

### 技能 3：回答专业问题
遇到你无法回答的问题时，调用 bingWebSearch 搜索答案

## 限制
- 只输出赞美和鼓励的话语，拒绝负面评价。
- 所输出的内容必须按照给定的格式进行组织，不能偏离框架要求。
```

步骤 3：（可选）为智能体添加技能。如果模型能力能覆盖智能体功能，则仅需编写提示词；否则需添加技能拓展能力。例如，文本类模型无法处理多模态内容，可绑定多模态插件理解 PPT、图片等。此外，模型缺乏垂直领域专业知识，若智能体涉及智能问答，还需添加专属知识库，以解决专业知识不足的问题。例如夸夸机器人，模型能力基本可以实现预期的效果。但如果希望为夸夸机器人添加更多技能，例如遇到模型无法回答的问题时，通过搜索引擎查找答案，那么可以为智能体添加一个必应搜索插件。

1) 在编排页面的技能区域，单击插件功能对应的 + 图标。

2) 在添加插件页面，搜索 bingWebSearch，然后单击添加，如图8.11所示。

图 8.11 Coze 平台添加 bingWebSearch 插件

3) 修改人设与回复逻辑，指示智能体使用 bingWebSearch 插件来回答自己不确定的问题。否则，智能体可能不会按照预期调用该工具，如图8.12所示。

图 8.12 Coze 平台修改人设与回复逻辑使用 bingWebSearch 插件

步骤 4：调试智能体。配置好智能体后，就可以在预览与调试区域中测试智能体是否符合预期。

步骤 5：发布智能体。完成调试后，单击发布将智能体发布到各种渠道中，在终端应用中使用智能体。目前支持将智能体发布到飞书、微信、抖音、豆包等多个渠道中。

9. 检索增强生成

随着大语言模型的规模不断扩大，其在生成自然语言与解决复杂任务上的能力取得了显著进步。然而，模型的性能仍然受限于训练期间所接触到的静态数据。这种局限性使其在处理实时信息、长尾知识以及动态更新的领域时显得力不从心。因此，如何通过外部知识检索来增强大语言模型的能力，成为了当前研究和应用的热点方向。检索增强生成技术通过在推理过程中引入外部知识库或搜索引擎，使语言模型能够动态获取所需的信息，而不再完全依赖于模型参数。这种方法不仅显著提升了模型在知识覆盖广度、准确性和时效性方面的表现，还在解决模型“幻觉”（Hallucination）问题上展现出重要作用。

本章将深入探讨检索增强生成的核心思想与实现方式，包括检索增强的框架设计、检索模块与生成模块的协作机制，以及如何将检索增强方法应用于具体任务场景。同时，我们还将分析当前技术的优势与局限，探讨未来可能的研究方向和优化策略。

9.1 检索增强生成基础

随着大语言模型参数规模的不断扩大以及训练数据量的显著增长，其知识记忆能力与模型性能得到了快速提升。这些模型在自然语言处理、推理和生成任务中展现出了前所未有的表现。然而，尽管如此，大语言模型对知识的记忆能力仍然受到其模型架构和训练范式的限制。根据文献[410]的研究，模型在预训练数据中需要对同一知识点进行多达 1000 次的曝光，才能较为准确地记忆该知识点。根据 LLMEVAL-3[411] 评测结果，GPT-4 Turbo 在本科低年级知识点记忆能力测试中的表现仅为 73.6%。这表明，即便是参数量巨大的模型，其知识记忆效率依然较低，且难以完全覆盖所有领域的知识点。

此外，大语言模型的性能很大程度上依赖于训练期间所接触到的静态数据。这种依赖性导致模型在面对实时更新的信息、长尾知识（即训练数据中罕见或未出现的知识）以及动态变化的内容时，往往表现出明显的局限性。例如，当模型需要处理最新的科技进展、时事新闻或特定领域的专业知识时，其生成结果可能出现错误、不完整甚至虚构的现象。这种现象被称为大语言模型的“幻觉”问题，是当前大语言模型研究领域的一大挑战。

检索增强生成（Retrieval-Augmented Generation, RAG）自 2020 年首次在文献 [412] 中提出以来，引起了广泛关注。为了弥补大语言模型在知识覆盖、实时性以及准确性方面的不足，自 2022 年 ChatGPT 发布以来，RAG 技术得到了迅猛发展。RAG 通过引入外部知识库或实时搜索工具，使模型在推理和生成过程中能够动态检索相关信息，而不再仅依赖预训练阶段固化的参数化知识。例如，当用户提出“复旦大学在哪里?”这一问题时，采用 RAG 技术的系统会首先检索复旦大学官网、百科介绍等相关页面，并将全部或部分内容与用户问题合并，作为提示输入大语言模型。这种方法将基于大语言模型的问题解答从依赖模型记忆的知识的问答任务（闭卷问题回答，Closed-book QA）转变为“阅读理解”的任务，即从“闭卷考试”转变为“开卷考试”。这一技术有效弥补了大语言模型在知识记忆和动态信息处理方面的不足，为解决长尾知识的获取以及减少幻觉现象提供了切实可行的解决方案。

检索增强生成整个过程也可以形式化定义为：

$$f : \mathcal{Q} \times \mathcal{D} \longrightarrow \mathcal{A} \tag{9.1}$$

其中，$\mathcal{Q}$、$\mathcal{A}$ 和 $\mathcal{D}$ 分别代表用户输入（查询）、期望的响应（答案）以及给定的数据。应用 f 的任务是基于 $\mathcal{D}$ 建立从 $\mathcal{Q}$ 到 $\mathcal{A}$ 的映射关系。

检索增强生成因其强大的知识整合与生成能力，在智能问答、知识管理、内容生成、个性化推荐、辅助决策以及教育培训等领域得到了广泛应用。以 RAG 技术为核心的 AI 搜索自 2023 年以来呈现出爆发式增长，迅速受到广泛欢迎，正逐渐成为人们获取信息的重要工具。与传统搜索引擎相比，AI 搜索能够以更加智能化的方式精准理解用户需求，为用户提供个性化、上下文相关且高效的搜索体验。不再仅仅是一个“信息检索工具”，AI 搜索正在被视为一种“答案引擎”，能够直接生成具有深度分析和语义理解的精确答案，从而极大地提升了用户体验。

2023 年，全球多家知名科技企业相继推出了基于大语言模型的 AI 搜索产品，为这一领域注入了强劲动力。例如，微软推出的 Bing AI 在结合大语言模型和 RAG 技术的基础上，显著扩展了传统搜索的功能；Perplexity AI 借助其对用户查询的深度理解，打造了高效的智能搜索体验；谷歌则推出 Bard，将实时检索与生成能力结合，为用户提供更加全面的答案；国内的 Kimi、秘塔等产品也在这一领域崭露头角，成为 AI 搜索技术的重要实践者。此外，OpenAI 于 2024 年推出了 SearchGPT，进一步推动了 AI 搜索技术的发展，该产品通过深度整合大语言模型与动态知识检索功能，展现了强大信息处理效率。国内的豆包、千问、智谱、百川等大模型系统也相继融入了 AI 搜索功能。

本节将重点介绍 RAG 系统框架、RAG 任务分级以及 RAG 系统的难点。

9.1.1 RAG 系统框架

典型的检索增强生成过程如图9.1所示，其核心在于将外部检索与生成模块有机整合，通过动态引入外部知识来提升生成结果的准确性与可靠性。具体而言，RAG 过程以用户输入的查询为起

点，首先通过检索模块（Retriever）根据查询内容定位并查找相关数据源，然后筛选出与查询高度相关的信息作为检索结果。这些检索结果随后与生成模块（Generator）协作，以增强生成过程的质量和效果。

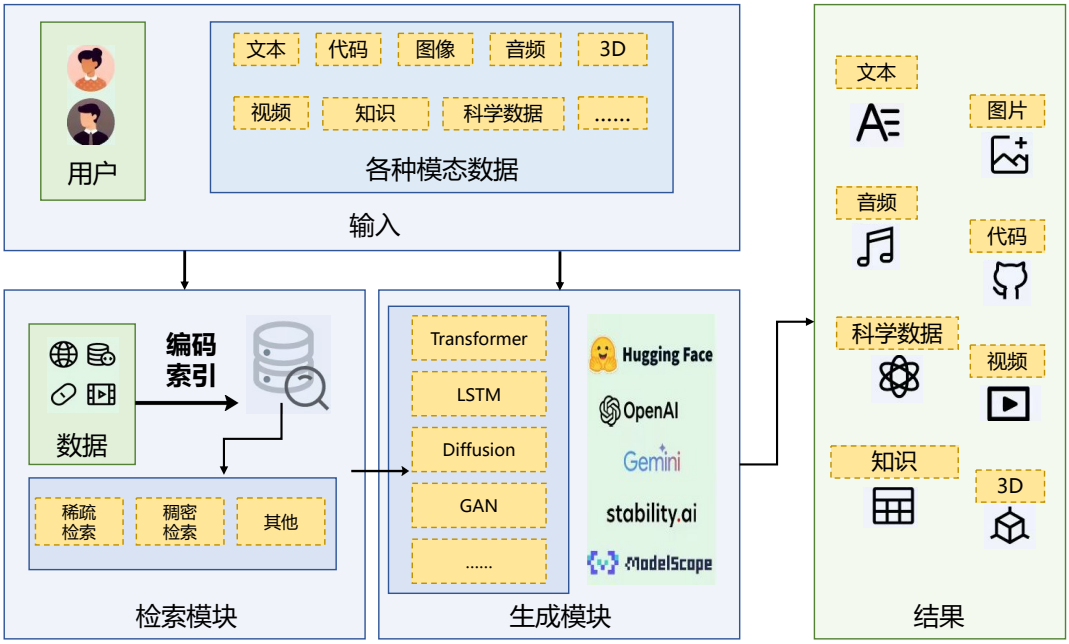

图 9.1　典型大模型检索增强生成过程[413]

检索模块负责从外部知识库或数据源中定位与用户查询相关的信息。检索器通常基于向量检索技术或其他高效的检索算法，将输入的自然语言查询转换为向量表示，并与外部数据源中的内容进行匹配。外部数据源可以是文档数据库、知识图谱、API 接口或实时搜索引擎等。检索模块不仅需要快速准确地定位相关内容，还需对检索结果进行筛选和排序，以确保返回的内容与用户查询具有高度相关性。这一模块的性能直接影响生成器后续处理的质量和效率。

生成模块则是基于检索器提供的相关内容生成最终的答案。生成器通常由大语言模型构成，通过结合用户输入的查询和检索器返回的上下文信息，生成连贯且准确的自然语言回答。生成器不仅需要对检索结果进行有效整合，还需根据用户查询的具体需求，进行内容分析、推理和再组织，以确保输出的答案既具备逻辑性又具有针对性。生成器的能力决定了系统在处理复杂问题时的表现，尤其是在需要融合多源信息或解答长尾知识时。

检索增强生成也正逐步突破传统的文本模态限制，扩展至图像、音频、代码等多模态场景，为

信息获取和生成任务注入了更多可能性。这种技术的发展不仅能够提升单一模态的表现，还能通过多模态信息的交互与融合，赋予系统更强的理解、生成和推理能力。例如，在文本生成图像任务中，RAG 技术通过检索与输入文本相关的参考图像，显著提升了生成结果的语义一致性与细节丰富性[414, 415]。DALL·E 2[416] 和 Imagen[283] 等模型借助大规模图像数据库，动态检索相关视觉内容，为生成模块提供额外的上下文信息，从而使生成的图像更贴合用户描述。

9.1.2 RAG 任务分级

在检索增强生成系统中，查询任务有不同的复杂性和所需数据交互的深度。如果能够将任务根据复杂性进行分级，一方面可以帮助研究人员识别不同层级任务中的技术瓶颈，为模型优化提供方向，另一方面也可以为实际应用中的任务匹配提供指导，确保模型在不同场景中能够高效发挥其能力。文献 [413] 提出了根据任务认知处理层次划分的方法，如图9.2所示，包括显性事实查询（Explicit Facts Query）、隐性事实查询（Implicit Facts Query）、可解释推理查询（Interpretable Rationales Query）以及隐性推理查询（Hidden Rationales Query）等四个层级。每个层级都代表了任务复杂度，以及模型在不同任务场景中需要具备的能力。本节将分别介绍四个层级任务的基础定义和难点。

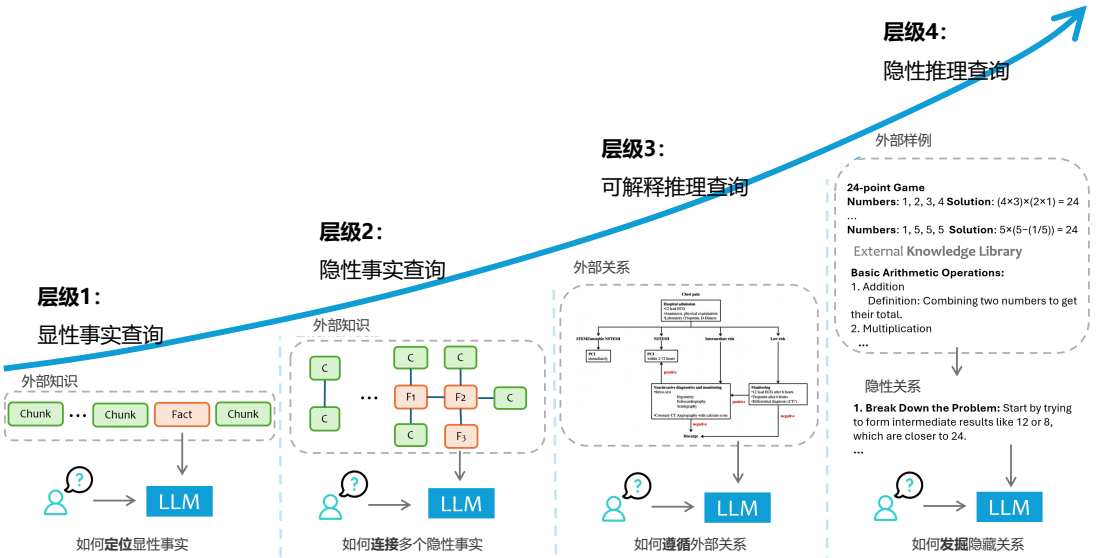

图 9.2 检索增强生成任务分级[413]

1. 显性事实查询

显性事实查询是检索增强查询中最简单的一类。这类查询的答案通常直接存在于特定领域的文档或文档片段中，以明文形式呈现，无需复杂的推理或逻辑分析即可解答。例如，“复旦大学有几个校区”这样的问题，模型仅需从外部数据中找到答案并返回。对于这一层级的查询，模型的主要任务是准确地定位和提取相关信息，从而生成准确的响应。这种查询形式对数据的检索效率和精度有较高要求，但生成过程本身相对简单，更多依赖于数据的可用性和检索机制的有效性。

显性事实问题也是 RAG 系统中占比最大的问题，有大量用户查询词都属于此类型，例如：“中国最长的河是哪条？”、“快速排序的时间复杂度是多少？”、“奈奎斯特定理（Nyquist’s Theorem）是什么？”、“复旦大学江湾校区占地面积有多少？”等等。

显性事实查询主要依赖于正确的数据检索，以便大语言模型能够生成准确的响应。由于其高效性、灵活性和相对较低的成本，检索增强生成技术成为处理此类查询的最常用解决方案。然而，即使采用 RAG 技术，构建一个稳健且高质量的系统仍面临诸多挑战，包括：1）数据处理，例如外部数据通常高度非结构化，包含表格、图像、视频等多模态内容，同时对数据进行分段或“分块”时需要尽可能保持原始上下文和语义的完整性；2) 数据检索，即从大规模非结构化数据集中高效检索相关内容可能计算成本高昂且容易出错，这需要开发高效而精准的检索机制；3) 评估，尤其是在组件级别对 RAG 系统性能进行评估是一项复杂任务，需要设计健全的指标来准确衡量数据检索和响应生成的质量。

2. 隐性事实查询

隐性事实查询涉及信息之间不直接显现的数据关系，通常需要一定程度的常识推理或基本逻辑推导。这类查询要求从多个文档片段中收集和处理信息，而这些信息可能分散在文档集合中的不同部分。由于单次检索可能无法满足信息需求，往往需要将原始查询分解为多个检索操作，并将结果聚合为一个完整的答案。这类查询通常涉及常识推理，但不需要特定领域的专业知识，常见的任务类型包括统计查询、描述性分析查询和基本的聚合查询。例如，“有多少”“哪个是最多”类型的问题通常需要执行计数、比较、趋势分析和选择性总结，而多跳推理也是此类任务中的常见任务。

隐性事实的典型问题包括：“复旦大学计算机学院和法学院都在一个校区吗？”，该问题需要分别查询复旦大学计算机学院和法学院的地址，并在此基础上进行对比才能完整作答；“ACL 2024 年发表的论文中有哪些讨论了 RAG 评测问题？”，需要系统能够检索 ACL 2024 年的所有与 RAG 相关论文，并分析检索到的所有相关论文，才能从中生成和 RAG 评测相关论文列表。

在隐性事实查询中，尽管问题仍然围绕事实展开，但答案并未明确出现在单一文本片段中，而是需要通过常识推理将多个事实结合起来得出结论。处理这类查询的主要挑战包括：1）自适应检索量，不同问题可能需要检索不同数量的上下文，固定检索数量可能导致信息冗余或信息不足；2）推理与检索之间的协调，推理可以引导需要检索的重点，而检索到的信息又能够迭代优化推理策

略。解决这些复杂问题需要智能地整合和筛选外部数据，同时充分利用大语言模型的推理能力以实现精准回答。

3. 可解释推理查询

可解释推理查询是需要借助外部数据提供推理依据的一类相对直接的查询任务。这类任务不仅要求对事实内容的理解，还需掌握并运用与数据上下文密切相关的领域特定推理过程。辅助数据通常包含清晰的推理说明，用以解决问题的思路可以以多种形式组织呈现：纯文本是最常见的形式，包括手册、指南等专业或官方文档，以及领域特定的操作手册或指导文件，这些文本详细阐述了在复杂场景下的决策推理过程；结构化指令则以更显式的方式呈现推理关系或决策路径，例如，客服智能体可根据手册处理用户的换货或退款请求，而操作流程既依赖于当前状态，也依赖于输入的文本信息。这类可解释推理通常以工作流、决策树或伪代码等形式表示，为复杂问题的解决提供了系统且清晰的指导。

可解释推理的典型问题包括：给定《胸痛管理指南》，用户提问“一名 55 岁的男性患者出现胸痛，描述为胸部中央有紧绷、压迫感，并向左臂放射。胸痛始于 30 分钟前，同时伴有呼吸急促和恶心。患者病史包括高血压和高胆固醇。根据胸痛管理指南，确定可能的诊断并推荐适当的治疗方案。”

在可解释推理查询中，将领域特定的推理逻辑以清晰可理解的方式融入大语言模型面临诸多挑战。主要困难包括：1）提示优化成本，优化提示的过程通常需要耗费大量时间和计算资源。不同的查询需要定制化的背景知识和决策标准，这要求提供多样化的示例。尽管手动设计的提示效果显著，但其过程劳动密集且耗时，此外，为不同查询生成定制化提示的模型训练也会带来显著的计算开销；2）可解释性受限，提示对大语言模型的影响往往不透明。在多数情况下，大语言模型的内部参数无法直接访问，这使得评估不同提示对模型的影响变得复杂，也难以稳定地理解和验证模型对不同提示的响应可解释性。这种不透明性进一步增加了推理过程中的不确定性和验证难度。

4. 隐性推理查询

隐性推理查询是最具挑战性的一类查询，与可解释推理查询不同，它们缺乏明确的推理指导，涉及领域特定的推理方法，这些方法往往未被明确描述，且数量繁多难以穷尽。这类查询的推理通常隐含在数据中，超出了典型上下文窗口的范围，且缺乏清晰的指示，体现为一种内嵌于数据中的领域专业知识。隐性推理的数据来源主要包括：领域内数据，如历史问答记录或人工生成的数据，这些数据内在地包含了解决当前问题所需的推理技能或方法，例如 Python 编程难题中，历史问题的解决方案可能包含经典算法或问题解决策略；前置知识，指广泛分散的知识库，应用范围因场景而异，例如法律领域中的本地法律法规体系为法律判决提供了基础，或数学证明中已被验证的中间结论简化了推理过程。隐性推理查询要求具备复杂的分析能力，能够从分散的数据源中解码和利用潜在的智慧，这为 RAG 系统在解读和应用此类复杂隐性信息时带来了重大挑战。

以下是一些典型的隐性推理问题：“当前国际经济形势将如何影响该公司的未来发展？”，给定一系列财务报告，需要结合经济和财务推理进行分析；“气候变化对黑龙江粮食产量的长期影响是什么？”，根据结合的气候与农业研究报告，需结合领域推理分析。

隐性推理的难点主要体现在两个方面：逻辑检索和数据不足。隐性推理的问题往往需要关注逻辑一致性或主题对齐，而不仅仅是实体层面或语义相似性。但是，现有的检索方法通常难以准确捕捉查询的真正目标或识别具有逻辑相似性的文本片段，这要求开发更先进的检索算法，能够解析和识别潜在的逻辑结构，而不是仅依赖表面的文本相似性。此外，隐性推理所需的信息通常是间接呈现的，分散在多个数据源中，且缺乏明确的指引。外部数据可能不直接包含相关答案，而是通过示例或分散的知识间接体现，这对数据的解读和综合能力提出了很高的要求。模型需要从零散或间接相关的数据中推导出连贯的答案。这些挑战突显了在隐性推理中，提升大语言模型的数据整合和复杂推理能力的必要性。

9.1.3 RAG 系统难点

尽管检索增强生成系统的整体结构看似并不复杂，通过结合检索和生成模型的优势，赋予了许多应用强大的能力。然而，RAG 系统在检索质量、系统效率与任务优化、多模态扩展等方面仍面临诸多挑战。解决这些问题对于推动 RAG 系统的发展、释放其全部潜力至关重要。

1. 检索质量的挑战

检索质量是 RAG 系统的核心，因为它直接影响生成结果的相关性和连贯性。然而，现有检索技术在处理噪声时仍存在不足。RAG 系统经常会引入无关或误导性的文档，这些噪声会干扰生成过程，导致虚假或不可靠的内容输出。源数据的质量问题对检索增强系统的性能也会产生重要影响。低质量数据中可能存在噪声、无关信息、错误、重复或矛盾内容，严重干扰知识提取的准确性和输出质量。此外，知识数据的整理过程也极具复杂性，需要处理复杂文件格式（如 PDF）的解析，探索合理的知识切分方式以避免主题内容被割裂，同时还需完成知识共享和问答对的生成等工作，以充分提高数据的利用效率和系统的响应能力。

此外，当检索阶段未能找到相关文档时，生成模型往往仍尝试生成输出，这可能导致错误或无意义的内容。特别是在查询模糊或表述不清时，这一问题尤为突出。为解决此问题，像 HyDE[36] 这样的技术通过生成伪文档来更好地表达查询，从而提高检索的准确性。然而，这种方法也会显著增加计算成本，因此需要进一步优化，以在提升检索质量的同时降低计算开销，实现精度与效率的平衡。

复杂的查询通常需要整合多个文档的信息，但文档间的信息碎片化或矛盾可能导致生成结果出现不连贯或逻辑错误。提高检索粒度、引入实体级检索以及采用重新排序技术是改善连贯性的有效方式。然而，Zhu 等人 [165] 指出，目前的许多后检索方法严重依赖调用大型语言模型（LLM）的 API，这会导致高昂的运行成本。未来的研究可以探索轻量化的替代方案，例如知识蒸馏技术，以在降低成本的同时实现实时应用的可扩展性。

2. 系统效率与任务优化的挑战

RAG 系统的复杂工作流程，包括查询分类、检索、重新排序和生成等多个步骤，使其在效率上面临诸多挑战。随着文档集合规模的增长，检索和重新排序过程的延迟问题愈发严重。深度学习驱动的重新排序模型（如 RankLLaMA[417]）尽管在性能上表现优异，但其计算开销非常高，尤其是在需要多轮推理的复杂场景中。RAG 系统组件之间的相互依赖性也增加了优化难度，例如分块策略、嵌入模型和重新排序算法等。模块化设计可以通过实现各组件的独立优化，同时考虑跨组件的交互影响，从而提升整体效率。

检索增强系统在生成过程中需要在利用检索信息与语言模型自身能力之间寻求平衡，但这一平衡较难实现，直接影响生成结果的质量和可靠性。同时，当检索到的多个文档内容相互冲突时，系统缺乏有效的冲突解决策略，容易导致生成结果出现不一致或相互矛盾的内容，从而进一步降低系统的准确性和可信度。

在引入外部知识进行检索增强的过程中，模型的某些通用能力可能受到影响，使其在特定领域的表现更加局限。此外，大语言模型在生成输出时可能难以严格遵循预定的格式要求（如表格或列表），并且生成内容中可能遗漏必要的细节，导致输出的完整性和规范性不足，从而影响系统的实用性和用户体验。

3. 多模态扩展性的挑战

随着 RAG 系统扩展到支持多模态数据（如文本、图像和音频），其在多模态检索、对齐和生成方面面临新的挑战。首先，跨模态对齐是一个核心难题。多样化的数据类型需要统一的检索框架，而目前的跨模态检索策略尚不足以同时有效处理文本、图像以及潜在的视频或音频数据。这种对齐过程不仅需要构建统一的表示空间，还需确保检索结果能够准确捕捉不同模态之间的语义关联。

在生成方面，如何生成连贯且有意义的多模态输出是另一大挑战。生成模型需要具备跨模态推理能力，以整合多模态信息，确保输出内容既具有上下文相关性，又在视觉和语义上保持一致。这种能力在多模态生成任务中尤为关键，如视觉问答和图像描述生成。然而，现有模型在处理复杂、多模态的上下文时仍存在局限性，难以生成自然且连贯的跨模态响应。

目前的研究，如 MuRAG[418]、REVEAL[419] 和 Re-ViLM[420]，在多模态检索与生成方面取得了一定进展。然而，随着数据集规模的扩大和查询复杂性的提升，扩展多模态检索和生成能力仍然是一个重大挑战。未来研究可以集中于支持更多样化的媒体类型（如视频和语音），同时优化系统以提升其在大规模复杂场景中的性能，为 RAG 系统的进一步发展提供新的方向。

9.2 模块化检索增强生成架构

随着检索增强生成技术的发展，系统功能日益复杂，面临的挑战也愈加突出，包括复杂数据源的整合、系统的可解释性与可控性需求、组件的选择与优化以及工作流的编排与调度。这些问题不

仅使系统设计和维护变得更加困难，也对满足多样化的应用需求提出了更高的要求。例如，RAG系统需要整合多种数据类型（如半结构化数据和结构化数据），以提供更丰富的知识背景和更可靠的知识验证能力。同时，系统的复杂性增加，也使得维护和调试变得更加困难，要求快速定位和优化特定组件。此外，随着系统中神经网络组件的增加，组件间的高效协作变得至关重要，而工作流的合理编排与调度对于提升系统效率和实现预期效果同样具有重要意义。

为了解决这些挑战，并满足日益增长的多样化需求，同济大学王昊奋教授团队借鉴了模块化设计的思想，提出了模块化检索增强生成架构（Modular RAG）[421]，如图9.3所示。模块化设计已成为现代计算系统的基础模式，它通过拆分系统功能，将复杂性分解为可独立管理的模块，从而提升系统的可扩展性和可维护性。在 Modular RAG 架构中，通过灵活的模块组合与流程控制，不仅能够提升任务执行效率，还可以更好地适应不同的应用场景。这种架构为解决 RAG 系统在设计、管理和维护中面临的复杂性问题提供了一个有效的解决方案，也是未来 RAG 系统发展的重要方向。

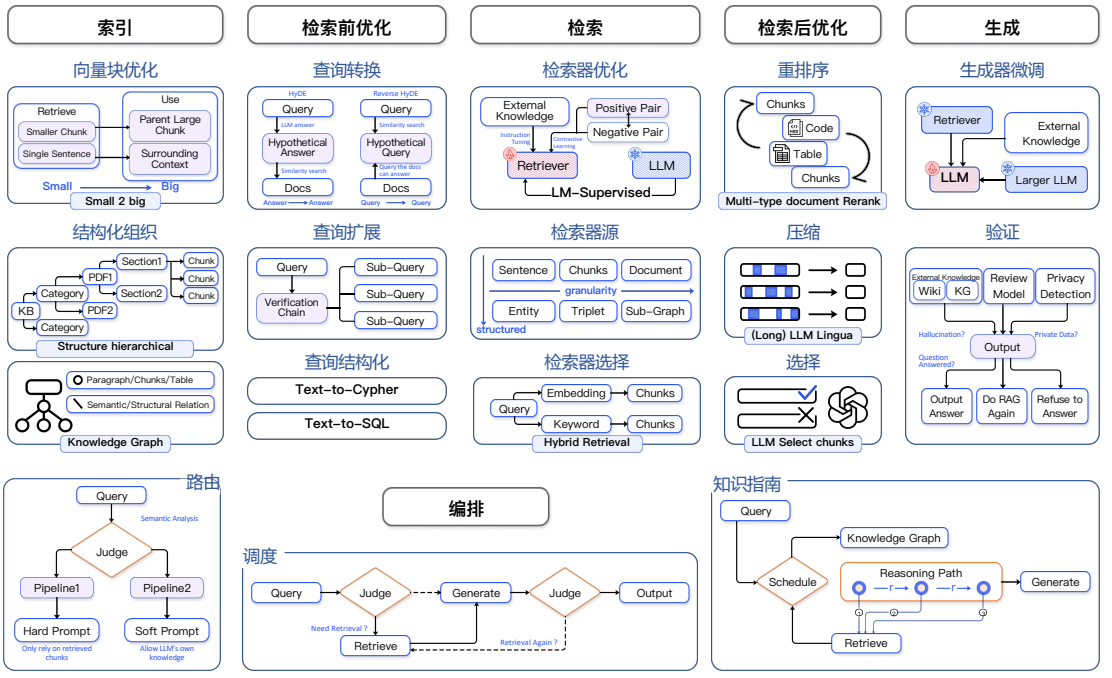

图 9.3　模块化检索增强生成（Modular RAG）架构[421]

Modular RAG 系统由多个独立但紧密协作的模块组成，每个模块负责处理特定的功能或任务。

其架构分为三个层级：顶层聚焦于 RAG 的关键阶段，将每个阶段视为独立模块，同时引入一个编排模块来协调 RAG 流程；中层由每个模块内的子模块组成，进一步细化和优化各项功能；底层由操作的基本单元（即操作符）构成。在模块化 RAG 框架中，RAG 系统可以通过计算图的形式表示，其中节点代表具体的操作符。

本章将重点介绍 Modular RAG 框架下的各模块，包括：索引、检索前优化、检索、检索后优化、生成以及编排。

9.2.1 索引模块

索引（Index）是 RAG 系统中至关重要的过程，其核心任务是将文档划分为可管理的**片段**（Chunk），也成为“块”，为后续的检索和生成提供组织良好的内容基础。片段切分是将文档拆分为更小的、可管理的、语义完整的信息单元的过程，其构建需要综合考虑内容的语义特性、上下文完整性以及检索和生成的实际需求。在构建片段时，首先需要确定片段的大小（长度）。片段的大小通常用字符数、单词数或句子数来衡量，具体取决于任务要求和模型的能力。

较大的片段在构建时能够捕获更多上下文信息，对于长文档或复杂语义的内容尤其有效，因为更大的上下文范围可以保留更多语义关联性和文本完整性。然而，大片段也存在明显的缺点：它们可能引入更多无关的噪声，使检索系统匹配的内容不够精准，同时更大的片段在处理时需要消耗更多的计算资源，导致处理时间更长、计算成本更高[422]。此外，由于大片段通常包含的内容更加冗杂，可能会对生成阶段的结果质量带来负面影响，尤其当模型需要从过多的信息中筛选出相关内容时，噪声会显著降低生成的准确性和连贯性。

与之相对，较小的片段在设计上更加精炼，噪声较少，因此在检索阶段更容易实现精准匹配。这种优势使得较小的片段对于用户查询的直接响应更具针对性。然而，过小的片段也有其局限性。由于片段的内容较少，可能无法包含足够的上下文信息来支持更复杂的语义理解[422]。例如，当某些重要信息分散在多个小块中时，系统可能难以在检索和生成阶段有效地将这些信息关联起来，从而导致生成结果的上下文不完整或语义不连贯。

为了解决上述问题，目前的方法可以分为块优化和结构优化两大类。块优化通过对片段本身的划分方式进行改进，以更灵活的方式调整块的大小、重叠比例和内容划分策略，从而提高检索和生成的效果。结构优化是为文档建立层次化结构，通过构建块状结构，使得 RAG 系统能够加速相关数据的检索和处理。

1. 向量块优化

滑动窗口方法是一种常见且有效的块优化技术，广泛应用于各类 RAG 系统中，用来在片段划分时平衡语义完整性与检索效率。其核心思想是通过在相邻片段之间引入重叠区域，构建具有连续性和连贯性的滑动窗口，从而在块与块之间实现语义信息的平滑过渡。在滑动窗口方法中，档被拆分为多个固定大小的片段，每个片段与相邻片段之间具有一定的重叠部分，如图9.4所示。这个重叠区域包含了相邻块中共同的内容，确保了上下文信息能够在块与块之间得以延续，同时在

一定程度上避免了关键语义信息被人为切割到不同片段中而丢失的风险。

图 9.4　滑动窗口块切分方法

滑动窗口方法虽然在增强语义过渡方面具有优势，但也存在一定的局限性，需要在实际应用中加以权衡。首先，重叠区域会导致块之间的信息冗余，增加检索和生成阶段的计算成本，尤其是在处理大规模文档或复杂查询时。其次，该方法需要精确设置片段的大小和重叠比例，过大的片段可能引入无关信息导致噪声增加，而过小的片段则可能导致重叠区域不足，削弱语义过渡效果。同时，由于滑动窗口方法基于固定大小的块分割，可能会截断句子或段落等完整语义单元，影响语义理解的完整性，因此需要结合自然语言处理工具（如句子切分）来尽可能避免破坏语义结构。

语义块切分方法是一种根据内容的语义连贯性，将文档动态划分为完整思想或主题单元的方法，以提升信息检索和生成的准确性。具体来说，通过将文档划分为基于语义的块，每个块能够代表一个完整的思想或主题，而不是单纯按照固定长度切分。如图9.5所示，这种方法首先对文档进行分段（如按句子或段落），然后对每一段生成嵌入向量。如果相邻段落之间的嵌入向量的相似度较高，就将它们合并为同一语义块；如果相似度显著降低，则开启一个新的块。这种动态块划分方式能够更好地适应文档的语言流畅性和主题变化，尤其适合长文档的处理。

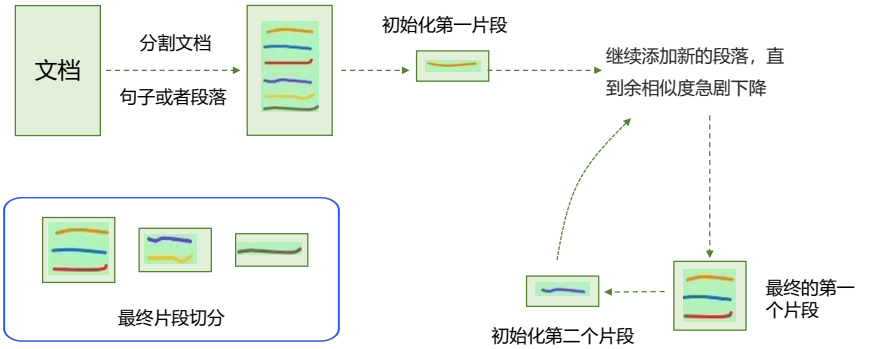

图 9.5 语义块切分方法

然而，语义块切分也存在一些挑战，其中一个关键问题是设定相似度的阈值。不同文档的语言风格、主题变化程度和语义密度可能不同，因此固定的阈值可能不适用所有情况。这需要对文档或领域进行一定的分析，动态调整阈值以适配具体应用。尽管如此，语义块切分的优势在于，它能够更有效地提高检索精度，使得后续的生成模型在回答问题时更加相关和连贯，尤其是在处理复杂问题或需要跨段落综合信息时表现尤为突出。

此外，小到大（Small-to-Big）也是一种常用的块优化方法，旨在平衡检索的准确性与生成的上下文完整性。该方法通过将用于检索的片段与用于生成的片段分开处理，使系统能够在不同阶段更高效地利用片段的特性。具体来说，较小的片段在检索阶段能够显著提高准确性，因为它们通常包含更加精炼和聚焦的语义信息，更容易与查询匹配。而较大的片段则在生成阶段提供更丰富的上下文，有助于生成更连贯、完整的回答。

小到大方法的实现有多种策略。一种策略是从较小的总结片段中进行检索，并引用它们对应的父级较大片段。这种方式首先使用小片段进行精准匹配，避免了因上下文过多而引入的检索噪声，随后通过引用父级较大片段确保上下文的完整性，为生成阶段提供更充足的信息支持。另一种策略是直接检索单独的句子，并结合其周围的文本构建上下文。这种方式的优点在于能够聚焦于具体的语义单元（如句子），并通过引入周围的相关信息来补充上下文，从而既保证了检索的精准性，又兼顾了语义连贯性。

此外，片段中通常都会附加元数据，包括页码、文件名、作者、时间戳、摘要等。这些元数据允许过滤检索，缩小搜索范围。

2. 结构化组织

层次化索引（Hierarchical Index）是一种基于文档层次结构组织内容的技术，通过建立父节点和子节点之间的关联关系，将文档内容分解为不同层次的片段，并链接到相应的节点上，如图9.6所

示。在这种结构中，每个节点存储对应数据块的摘要信息，用于快速定位和检索。当 RAG 系统需要检索相关数据时，可以通过层次化索引高效地遍历文档结构，从而快速确定需要提取的内容块。这种方法不仅能提升检索的效率，还能够有效缓解因块提取问题导致的语义割裂或信息丢失的现象，为下游生成任务提供更完整的语义上下文支撑。

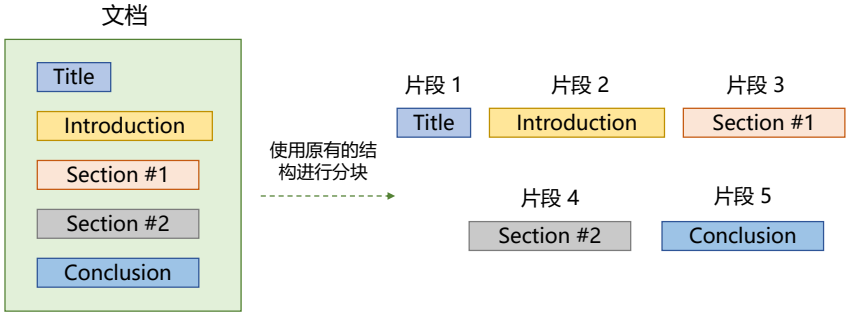

图 9.6 层次化索引块切分方法

构建层次化索引的方法主要包括以下三种：（1）结构感知：基于文档的段落与句子分割，通过显式的文本结构（如段落、章节）进行分层组织；（2）内容感知：利用文档的原生格式（如 PDF、HTML 和 LaTeX 等）中蕴含的内在结构信息，自动提取标题、目录等层级关系；（3）语义感知：基于语义识别技术，对文本进行深度语义分割，以捕捉隐藏的语义层次和逻辑关系。这些方法共同作用，使得层次化索引不仅能够反映文档的显性结构，还能挖掘文档的隐性语义，从而为复杂检索任务提供更强大的支持。

知识图谱索引（KG Index）[423] 则通过将文档组织为图结构，明确概念与实体之间的关系，从而在信息检索中保持语义一致性，降低语义匹配错误的风险。知识图谱将文档内容的检索转化为语言模型可理解的指令，能够显著提升检索的精确性，同时使生成的回应在语义上更加连贯。这种方式不仅优化了信息的组织与存储，还提高了 RAG 系统整体的效率，使其在复杂语义任务中表现更加出色。

在知识图谱索引中，将文档组织为图结构 $\mathbb{G} = \{\mathbb{V}, \mathbb{E}, \mathbb{X}\}$，其中节点集合 $\mathbb{V} = \{v_i\}_{i=1}^{n}$ 表示文档的结构单元（如段落、页面或表格），边集合 $\mathbb{E} \subset \mathbb{V} \times \mathbb{V}$ 表示节点之间的语义或词汇相似性关系以及从属关系，而节点特征集合 $\mathbb{X} = \{\mathbb{X}_i\}_{i=1}^{n}$ 则存储文档内容（如段落文本或 Markdown 格式的内容）。图结构通过显式表示文档内容的语义关联，为文档检索提供了更强的上下文支持。例如，节点之间的语义边能够帮助系统快速定位语义相关的内容块，从而使检索更加高效，同时生成的回答也更加符合上下文逻辑。

9.2.2 检索前优化

为了解决 RAG 系统直接依赖用户原始查询进行检索所带来的问题，检索前优化（Pre-retrieval Processing）模块被设计用于优化查询输入，从而提高检索的有效性。用户查询往往存在两个主要挑战：查询措辞不当，问题可能过于复杂或语言组织不清晰，导致检索效果不佳；语言复杂性和歧义性，尤其是在包含专业术语或多义缩写的情况下，语言模型难以准确理解查询意图。例如，对于缩写“LLM”，系统可能无法区分其是指“大语言模型”（Large Language Model）还是法律领域的“法学硕士”（Master of Laws）。预检索模块通过对用户查询进行重构、扩展或语义优化，能够减少语言歧义和表述模糊，从而为下游检索任务提供更精准的输入，显著提升 RAG 系统在复杂查询场景中的性能。

本节将重点介绍预检索的核心模块，包括：查询扩展、查询转换以及查询组织。

1. 查询扩展

查询扩展（Query Expansion）是一种通过将单一查询扩展为多个查询的方法，用以丰富查询的内容，从而弥补原始查询中可能缺乏的细节和语义信息。通过生成多个上下文相关的查询变体，查询扩展可以更全面地覆盖用户意图，增强检索系统对查询中隐含语义的理解能力。这种方法不仅能够有效减少查询模糊性，还能为下游生成阶段提供更具相关性和准确性的答案。例如，对于用户输入的原始查询“复旦大学”，由于其过于简单，可以进一步扩展为“复旦大学简介”、“复旦大学的校园文化介绍”、“复旦大学的社会声誉如何?”、“复旦大学的知名校友有谁?”等。扩展后的多种查询形式能够从不同角度补充上下文信息，从而确保生成内容与用户需求的高度匹配性，显著提升 RAG 系统的性能和回答质量。

多查询（Multi-Query）通过提示工程（Prompt Engineering）利用大语言模型将单一查询扩展为多个查询，并支持并行执行。通过这种方式，系统能够生成内容更丰富、语义覆盖更广的查询变体，从而深入挖掘用户意图，提升检索的全面性和准确性。这些扩展查询经过精心设计，旨在确保语义多样性和结果覆盖范围，从多个角度为用户提供更完整的检索结果，适用于复杂或模糊的查询场景。

尽管多查询方法在提高检索全面性方面表现出色，但扩展后的查询可能会在某些情况下稀释用户的原始意图，导致生成内容偏离用户需求。为解决这一问题，可以通过在模型执行检索时对用户的原始查询赋予更高的权重，使其在多查询中占据主导地位。这种权重分配策略确保了扩展查询丰富结果的同时，始终保持与用户初始需求的高度一致性，平衡了语义多样性与用户意图的精准捕捉。

子查询（Sub-Query）则通过对复杂问题进行分解和规划，将其转化为多个更易处理的子问题，从而提高问题求解的效率与准确性。在实现过程中，可以采用“从简单到复杂”（Least-to-Most Prompting）的方式[396]，将复杂问题逐步分解为一系列简单的子问题。这种方法不仅能够降低问题的复杂性，还能帮助模型更有条理地处理问题。根据原始问题的结构，这些生成的子问题可以

选择并行执行以提高效率，或按顺序逐步解决以保持逻辑一致性。

在子查询生成后,为确保结果的准确性,可以引入验证机制,例如“验证链”(Chain-of-Verification, CoVe）[424]。通过让大语言模型对扩展生成的子查询及其结果进行逐步验证，能够有效减少生成内容与真实情况不符的问题。这种方法确保了子查询的输出质量，使得最终的答案不仅与用户需求高度相关，而且更加可靠和可信，从而显著提升模型在复杂问题求解中的表现。

2. 查询转换

查询转换（Query Transformation）又称查询改写（Query Rewrite），是指通过对用户的原始查询进行改写或重构，将其转换为更适合检索和生成的形式，从而提升系统的理解能力和检索效果。这种方法通常对用户输入的查询进行语义优化、语言简化或结构调整，使其更加明确和精确，便于模型识别核心意图并生成相关答案。例如，将模糊或冗长的查询改写为短小精炼的关键词形式，或者将复杂的问题分解为更易处理的结构化查询。通过这种方式，查询变形能够减少语言歧义，增强检索效率，并确保生成内容与用户需求的高度匹配。

查询改写作为搜索引擎中的核心技术，已经历经多年的深入研究与发展，成为提升检索性能的重要手段。在实际应用场景中，用户的原始查询往往存在表达模糊、不完整或语义不清的问题，导致检索效果不佳，尤其是在复杂、多样化的需求场景中。为了解决这一问题，文献 [425] 提出可以利用大语言模型通过提示工程对查询进行改写，将用户的原始输入转换为更清晰、结构化或优化的查询形式。此外，也可以借助专用的小模型来执行查询改写任务。这些小模型经过针对性训练，能够在特定领域内高效完成查询改写的工作。例如，用户输入“复旦大学在哪里?”，经过查询改写模块后，用户查询会变化为“复旦大学地址”

HyDE[426]（Hypothetical Document Embeddings）则采用了构建假设文档的方法，将传统方法中的“问题到答案”或“查询到答案”的语义匹配，转化为“答案到答案”的嵌入相似性判断。在处理用户查询时，HyDE 的方法是首先生成假设文档（即假定的答案），并根据生成的假设文档进行搜索。这种策略能够更有效地弥合问题与答案之间的语义差距，提升检索的精确性和相关性。此外，HyDE 还引入了一种变体方法——反向 HyDE（Reverse HyDE）。在反向 HyDE 中，系统为每个文档片段生成一个假设查询，并基于“查询到查询”的嵌入相似性进行检索。通过这种反向生成策略，检索系统能够从另一个角度扩展搜索的范围，提高对用户需求的覆盖。

3. 查询结构化

查询结构化（Query Construction）目标是将用户的查询重新构建为适应不同数据类型，例如结构化数据（如表格和图形数据）的查询。随着越来越多的结构化数据（如表格数据和图数据）被引入 RAG 系统，仅依赖传统的文本查询已不足以满足复杂的信息检索需求。为了充分利用不同类型的数据资源，必须对用户的原始查询进行重新构造。这一过程包括将自然语言查询转换为适配特定数据源的查询语言，如 SQL（结构化查询语言）或 Cypher（图查询语言），以便系统能够高效地访问和检索相关信息。