第 5 章

检索增强生成

在当今人工智能发展的浪潮中，检索增强生成（Retrieval-Augmented Generation，RAG）作为一种基于大模型的先进应用系统架构，已成为主流趋势。这一架构的核心理念在于将检索技术和大模型巧妙结合，以提升生成式模型的回答质量和准确性。RAG 系统通过检索相关数据源中的信息，并将这些信息融入大模型的上下文中，显著地增强了模型的回答能力，确保生成的答案不仅基于模型自身的知识储备，还能够充分地利用实时检索到的有效信息。RAG 架构的关键组成部分包括嵌入式搜索引擎、向量数据库及专门针对 RAG 设计的开发框架。Faiss 等工具被用于高效实现向量搜索，而诸如 Pinecone 之类的向量数据库则提供开源搜索索引及额外的存储解决方案，极大地促进了 RAG 系统的构建。与此同时，开源社区涌现出了 LangChain 和 LLaMAIndex 这样优秀的开发框架，它们专为基于大模型的应用程序设计，帮助开发者迅速搭建起 RAG 流水线，并在 ChatGPT 引发的热潮中获得了广泛应用。

RAG 系统的实现过程始于对文本文档语料库的操作，主要包括将文档切分为有意义的块、通过 Transformer 编码器模型生成块的嵌入向量、建立向量索引及构造大模型的提示语句。在运行阶段，用户查询会被转换为向量形式，通过索引搜索找到最相关的上下文片段，随后将这些片段提供给大模型，引导模型生成精确答案。另外，RAG 系统还整合了一系列高级技术，如查询路由、数据分块策略、嵌入模型选择及向量索引优化等。通过灵活运用这些技术手段，RAG 能够在不同场景下，无论是面向大规模文档集合的智能问答，还是成千上万用户数据驱动的聊天机器人应用都能展现出卓越的性能表现。值得注意的是，针对 RAG 系统性能的评估涉及多项指标，包括但不限于答案相关性、可信度及检索上下文的相关性，并可通过诸如 Ragas 框架、Truelens 评估框架及 LLaMAIndex 提供的 Rag_evaluator 等工具进行全面评测和持续优化。

RAG 架构的成功应用不仅推动了大模型技术在实际场景下的深化发展，同时也催生出一系列创新性的微调和融合搜索技术，使大模型产品能够更精准地服务于多样化的需求，展现出了广泛的实用价值和未来潜力，接下来进行详细讲解。

5.1 RAG 技术原理

在探索人工智能的前沿领域时，经常会遇到一些令人兴奋的新概念和新技术，它们不断地推动着机器学习和自然语言处理的边界。RAG 便是这样一种引人注目的技术，它不仅代表了当前大模型应用的一种趋势，同时也预示着未来人机交互和信息检索的可能方向。本节将深入探讨 RAG 技术背后的原理，包括它的概念和应用、技术架构及如何通过分块和向量化、搜索索引、重新排序和过滤、查询转换、查询路由、智能体的整合，以及响应合成器等高级技术来实现高效的问答服务。此外，还将讨论如何对 RAG 系统模型进行微调，以及如何进行性能评估以确保系统的准确性和可靠性。通过这些深入的分析和理解，可以更全面地把握 RAG 技术的精髓，并为实际应用打下坚实的基础。

5.1.1 RAG 的概念与应用

RAG 是一种结合了大模型和向量数据库的技术，旨在提高生成答案的质量和准确度。具体来讲，RAG 通过引入一个向量数据库作为外部知识源，使大模型在生成回答时能够参考更多的背景信息，从而提供更加丰富和准确的输出。这种技术的运用，不仅提升了模型的回答质量，还扩展了其在不同场景下的应用能力。RAG 的工作原理可以概括为以下几个步骤：首先，当用户提出一个问题时，系统会利用向量数据库进行检索，找出与问题相关的信息片段；然后，将这些检索到的信息作为上下文，与大模型生成的答案相结合，形成最终的输出。这种方法类似于将传统的搜索引擎与先进的自然语言处理能力结合起来，以提供更智能化的服务。RAG 技术在多个领域都展现出了其潜力，包括问答服务、与数据对话的应用程序等。随着技术的不断发展和完善，RAG 有望成为未来人工智能系统中的一个重要组成部分，为用户带来更加高效和精确的信息获取体验。

RAG 的应用非常广泛，它在多个领域展示了显著的优势。

（1）提升准确性和相关性：通过在生成答案之前检索广泛的文档数据库中的相关信息，引导语言模型的生成过程，从而极大地提升了内容的准确性和相关性。

（2）缓解幻觉问题：传统的大型语言模型可能会产生所谓的“幻觉”，即生成不存在或错误的信息。RAG 通过提供准确的背景信息，有效地减轻了这一问题。

（3）提高知识更新速度：在快速变化的信息时代，实时获取最新知识至关重要。RAG 能够快速检索最新资料，确保生成的内容能反映当前的事实状态。

（4）增强可追溯性：通过明确标记哪些信息是检索得来的，提高了内容生成的透明度和可信度。

（5）实用性和可信度的提升：由于上述优点，RAG 使大型语言模型在实际应用中变得更加实用和可信。

（6）成本效益高：相对于成本高昂的个性化训练或微调，基于 RAG 的技术方案往往具有更高的成本效益。

综上所述，RAG 不仅为大模型应用提供了新的可能性，也推动了人工智能研究领域的发展。接下来深度剖析 RAG 技术架构。

5.1.2 RAG 技术架构

RAG 基础技术架构始于一个庞大的文本文档语料库，其核心在于通过检索技术辅助大模型生成答案，使模型的回答不是依赖于其自身的预测，而是有实际的、可查证的信息作为支撑。RAG 基础技术架构可以分为以下几个关键环节。

（1）数据准备：首先，需要将大量的文本数据加载并准备好，这通常由功能强大的开源数据加载工具来完成，这些工具能够连接到各种数据源，如 YouTube、Notion 等。

（2）分块与向量化：将文本切分为多个段落，并利用 Transformer 编码器模型将这些段落转换成向量形式。这一步骤的目的是将非结构的文本数据转换为计算机可以高效处理的数值型数据。

（3）建立索引：将所有向量汇集到一个索引中，这个索引将作为后续检索操作的基石。

（4）检索过程：当用户提出一个查询时，同样的编码器模型会将这个查询转换成向量，然后系统会在索引中搜索与查询向量最相近的前 K 个结果。

（5）上下文提供：从数据库中提取与查询最相关的文本段落，并将这些段落作为上下文信息提供给大模型。

（6）生成答案：大模型根据所提供的上下文信息和原始查询生成最终的答案。

（7）后处理：在某些情况下，可能需要对检索结果进一步地进行过滤、重新排序或转换，以优化最终输出的答案。

RAG 基础技术架构如图 5-1 所示。

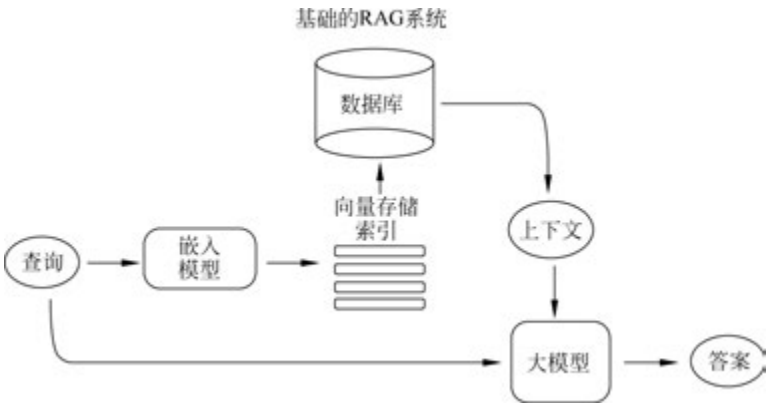

图 5-1 RAG 基础技术架构

RAG 技术的发展不仅限于上述基础架构，还包括了多种高级技术和优化策略，如查询转换、智能体行为、聊天引擎开发、查询路由决策等，这些高级技术使 RAG 系统更加灵活和智能，能够更好地适应用户多样化的需求。RAG 高级技术架构如图 5-2 所示。

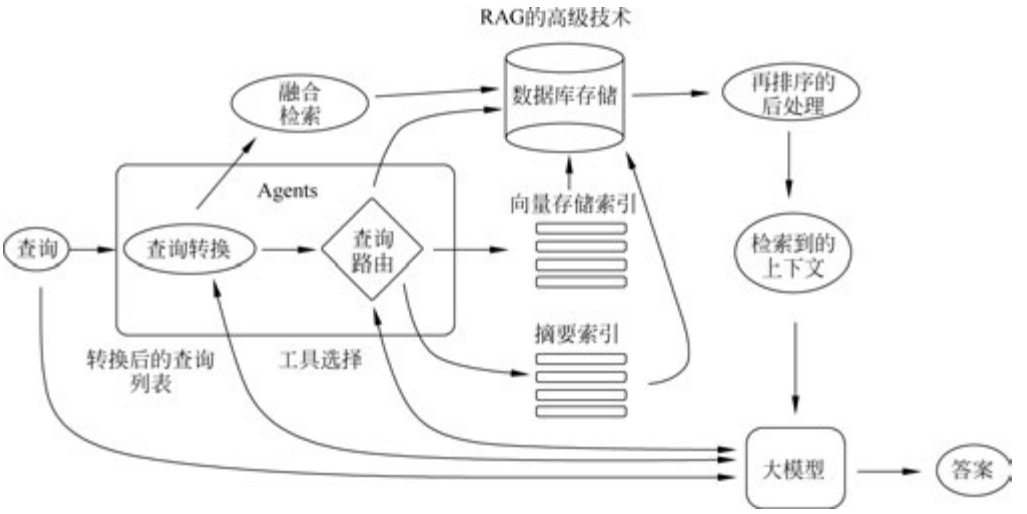

图 5-2　RAG 高级技术架构

RAG 高级技术架构主要包括以下几个方面：

（1）分块和向量化：将文档分割成块，并为每个块生成向量表示，以便后续索引和检索。

（2）搜索的索引：构建索引以存储文档块的向量，并使用向量搜索技术（如 Faiss）来高效检索与查询向量最接近的文档块。

（3）重新排序和过滤：对检索结果进行过滤和重新排序，以提高答案的相关性和准确性。

（4）查询转换：通过大模型修改用户输入以提高检索质量，包括查询分解、后退提示和查询重写等技术。

（5）查询路由：根据用户查询动态决定检索路径，例如选择不同的索引或数据存储。

（6）智能体：利用大模型作为推理引擎，结合一系列工具和任务，形成能够自主决策的智能体。

（7）响应合成：将检索到的上下文与用户查询结合，通过大模型生成最终的答案。

接下来深入讲解每个关键环节。

5.1.3　分块和向量化

为了在 RAG 系统中实现高效的信息检索，首先需要将文档内容转换为向量索引。这一过程涉及将文本数据分块并转换为向量表示，以便在运行时搜索与查询向量最接近的语义匹配。由于 Transformer 模型具有固定的输入序列长度限制，因此对数据进行分块是一种有意义的技术。通过将初始文档分成一定大小的块，可以确保每个块都能代表其所包含的一个或几个句子的语义意义。这样，即使输入上下文的窗口很大，一个或几个句子的向量也比一个在几页文本上取平均值的向量更能代表它们的语义意义。在进行数据分块时，可以采用各种现有的文本分割器实现，例如在 LLaMAIndex 中，NodeParser 提供了一些高级选项，如定义自己的文本分割器、元数据、节点/块关系等。数据块的大小是一个重要的参

数，它取决于所使用的嵌入模型及其 Token 容量。标准的 Transformer 编码模型，如 BERT 的句子转换器，最多只能使用 512 个 Token，而 OpenAI 的 ada-002 模型能够处理更长的序列，如 8191 个 Token。然而，在选择数据块大小时，需要在足够的上下文和特定的足够文本嵌入之间进行权衡，以便有效地执行搜索。接下来，需要选择一个模型来生成所选块的嵌入。有许多方法可供选择，例如搜索优化的模型（如 bge-large-zh 或 E5 系列），MTEB 排行榜可以提供最新的一些方法信息。此外，还可以考虑使用中文 Embedding 模型，如 gte-large-zh、bge-large-zh-v1.5、multilingual-e5-large、m3e-base 和 bce-embedding-base v1 等。总之，在 RAG 系统中，分块和向量化是实现高效信息检索的关键步骤。通过合理地划分数据块并选择合适的模型生成嵌入，可以在保持语义准确性的同时提高搜索效率。

5.1.4 搜索的索引

在面向 RAG 的大模型应用中，关键的部分是用于搜索的索引，它存储了先前得到的向量内容。当然，查询总是首先需要向量化，对于 Top K 分块也是一样的。最简单的实现是使用一个平面索引，在查询向量和所有块向量之间进行距离计算并遍历。为了在一万多个元素的尺度上有效地进行检索，需要一个向量索引库，如 Faiss、Milvus、Pinecone 等，可以使用一些近似最近邻方法实现向量检索，例如聚类、树或 HNSW 算法。根据索引的选择，数据和搜索需求还可以将元数据与向量一起存储，然后使用元数据过滤器在某些日期或数据源中搜索信息。LLaMAIndex 既支持许多向量存储索引，也支持其他更简单的索引实现，如列表索引、树索引和关键字表索引。如果有许多文档，就需要能够有效地在其中进行搜索，找到相关信息，并将其聚合在一个带有源引用的答案中。对于大型数据库，一个有效的方法是创建两个索引，一个由摘要组成，另一个由文档块组成；然后分两个步骤进行搜索，首先通过摘要过滤掉相关文档，然后通过相关组进行搜索。另一种方法是要求大模型为每个块生成一个问题，并将这些问题嵌入向量中，在运行时对这个问题的向量索引执行查询搜索（在索引中用问题向量替换块向量），然后路由到原始文本块并将它们作为大模型获得答案的上下文。这种方法提高了搜索质量，因为与实际块相比，查询和假设问题之间具有更高的语义相似性。还有一种被称为 HyDE 的反向逻辑方法，要求一个大模型生成一个假设的给定查询的响应，然后使用它的向量和查询向量来提高搜索质量。为了获得更好的搜索质量而检索更小的块，就要为大模型添加周围的上下文。有两种选择：一种是句子窗口检索，即在检索到的较小块周围按句子展开上下文；另一种是父文档检索，即递归地将文档分割成若干较大的父块，其中包含较小的子块。在句子窗口检索方案中，文档中的每个句子都是单独嵌入的，这为上下文余弦距离搜索提供了很高的准确性。在获取最相关的单个句子之后，为了更好地进行推理并找到上下文，在检索到的句子之前和之后将上下文窗口扩展为 k 个句子，然后将这个扩展的上下文发送给大模型。父文档检索与句子窗口检索非常相似，都是搜索更细粒度的信息，然后将上下文提供给大模型进行推理之前扩展过的上下文窗口。文档被拆分成引用较大父块中的较小子块。具体而言，文档被分割成块的层次结构，然后最小的叶子块被发送到索引。在检索期间，获取较小的块，然后如果在 top K 检索的块中有

超过 n 个块链接到同一个父节点(较大的块),就用这个父节点替换提供给大模型的上下文。需要注意的是,搜索仅在子节点索引中执行。

还有一个相对较老的思路,可以像 TF-IDF 或 BM25 这样的稀疏检索算法那样从现代语义或向量搜索中获取最佳结果,并将其结合在一个检索结果中。这里唯一的技巧是将检索到的结果与不同的相似度得分恰当地结合起来,这个问题通常借助于倒数排序融合(Reciprocal Rank Fusion,RRF)算法来解决,对检索到的结果重新排序以得到最终的输出。RRF 算法是一种用于融合多个排序结果的技术。在信息检索领域,当有多个排序模型或排序算法对某个查询进行排序时,能够将这些多个排序结果融合为一个最优的排序结果,而 RRF 算法就是一种用于实现这一目标的方法。RRF 算法中的 Reciprocal Rank 表示一个文档在单个排名结果中的相对重要性。它是一个介于 0 和 1 之间的值,取决于文档在排序列表中的位置,越靠前,值越大。在 LangChain 中,这是在集成检索器类中实现的,例如,一个 Faiss 向量索引和一个基于 BM25 的检索器,并使用 RRF 算法进行重新排序。在 LLaMAIndex 中,也是以一种非常类似的方式完成的。

最后,在融合检索或混合搜索方面,目标是结合传统基于关键字的稀疏检索算法(如 TF-IDF 或 BM25)与现代基于语义的向量搜索的优势。要将这两种不同类型的检索结果合理地结合在一起,常常借助 RRF 算法对检索结果进行加权和重新排序,从而产生最终输出。在 LangChain 和 LLaMAIndex 这样的框架中,实现了类似的混合检索器,它可以同时使用 Faiss 向量索引和基于 BM25 的检索器,并通过 RRF 算法对检索结果进行智能融合,以期在兼顾查询与文档间语义相似性和关键字匹配的基础上,提供更为优质的检索结果。

5.1.5 重新排序和过滤

在 RAG 框架中,重新排序扮演着关键角色,其目标在于提升检索结果的质量和针对性。在基础的 RAG 流程中,系统可能会检索到大量与用户查询相关的上下文,但由于检索阶段可能存在噪声,所以不是所有检索到的上下文都高度相关或能提供准确答案。重新排序机制则能够对这些初步检索出的文档进行筛选和排序,确保相关度最高的文档被优先考虑并用于最终答案的生成,从而显著地提高生成答案的准确性及整体质量。重新排序技术的核心任务是评估每个检索到的上下文与查询间的相关性,并将最有可能提供精确答案的上下文排在前列。这一过程类似于一个智能过滤器,可以根据上下文与查询之间的交互特征精确地衡量它们的相关程度。下面是两种主流的重新排序方法。

(1) 重新排序模型:这类模型特别关注文档与查询之间的交互特性,通过更深入的分析以提高评估相关性的准确性,例如,Cohere 公司提供的在线模型可以直接通过 API 调用来执行重新排序任务,同时也有开源模型(如 Bge-Reranker-Base 和 Bge-Reranker-Large 等)可供选择。这些模型接收查询和上下文作为输入,直接输出反映它们相似性的得分,而非嵌入向量的得分,并且通过交叉熵损失函数进行优化训练,这意味着相关性得分不受限于特定正向数值区间,甚至可以为负值。

(2) 使用大模型进行重新排序:大模型的出现为重新排序提供了全新的解决方案,例

如，RankGPT 作为一种利用大模型（如 ChatGPT、GPT-4 或其他大模型）进行零样本列表式段落重新排序的技术，采用了排列生成法结合滑动窗口策略，以有效的方式对段落进行排序。这种方法不需要外部评分机制，而是利用大模型自身的语义理解和生成能力，对候选段落进行端到端排序。针对大模型输入长度限制的问题，RankGPT 提出了一种滑动窗口策略，逐步对文本片段进行排序，最终综合整个文本集从而得到最优排序结果。

在实际应用中，为了整合重新排序功能，可使用 Bge-Reranker-Base 模型进行基本的检索和排序操作。在实践中，重排序和过滤的步骤通常包括以下几个环节。

（1）环境配置：设置必要的环境变量和全局变量，导入相关库，并准备好检索器所需的资源。

（2）构建检索器：使用 LLaMAIndex 等工具，根据文档集合构建检索器，并设置检索参数，如相似度阈值。

（3）基本检索：执行检索操作，获取与查询相关的上下文节点。

（4）重新排序：应用重排序模型或大模型对检索到的节点进行重新评分和排序。

（5）评估：通过比较重排序前后的结果，使用诸如命中率（Hit Rate）和平均倒数排名（Mean Reciprocal Rank，MRR）等指标来评估重排序的效果。

以 LLaMAIndex 为例，可以使用 FlagEmbeddingReranker 类加载 Bge-Reranker-Base 模型来进行重新排序，或者使用 RankGPTRerank 类结合 OpenAI 的大模型进行重新排序。在实施过程中，可以根据具体需求和场景选择合适的重排序方法，并通过调整参数和尝试不同的模型组合来优化检索效果。总体来讲，在 RAG 框架下，重新排序技术极大地提升了检索与生成系统的效率和精准度，通过对检索结果的精细化处理，使大模型能够更聚焦于最相关的内容，进而生成更高质量的答案。无论是通过专门设计的重新排序模型还是利用大模型的内在理解能力，重新排序都是改进 RAG 性能的重要手段。

5.1.6 查询转换与路由

RAG 已经成为生成式人工智能领域讨论最多的主题之一，因为在减轻大模型中的幻觉方面，没有其他解决方案能与之媲美。RAG 通过可靠的外部来源，如维基百科页面、私人 PDF 等，来增强语言模型的通用知识，因此，RAG 最重要的步骤是确保检索能够找到正确的文档供模型使用。之所以如此需要 RAG，是因为在将完整文档放入上下文窗口时面临一些限制。这些限制包括模型输入的标记长度受限、计算成本的比例增加，以及所谓的“中间丢失”问题。“中间丢失”是指模型难以使用长输入上下文中的中间信息的现象。如果检索到的文档太长或不相关，就像俗话说的垃圾进、垃圾出。增强 RAG 的技术有很多，这就带来了一个额外的挑战，即知道何时应用每种技术。下面分析一些查询转换技术，看一看在每种情况下哪种效果最好。

1. 假设性文档嵌入

假设性文档嵌入（Hypothetical Document Embeddings，HyDE）是一种生成文档嵌入以检索相关文档而不需要实际训练数据的技术。首先，大模型针对查询创建一个假设性答案。

虽然该答案反映了与查询相关的模式，但这个答案包含的信息可能与事实不符。接下来，查询和生成的答案都被嵌入，然后系统会从预定义的数据库中识别并检索出在向量空间中与这些嵌入最接近的实际文档。

2. 子问题

子问题(Sub Questions，SQ)技术采用分而治之的方法来处理复杂的问题。它首先分析问题并将其分解为更简单的子问题。每个子问题针对不同的相关文件，这些文件可以提供部分答案，然后引擎收集中间回复，并将所有部分结果合成最终回复。

3. 多步查询转换

多步查询转换(Multi-Step Query Transformation，MQT)方法以自我提问法为基础，即语言模型在回答原始问题之前向自己提出并回答后续问题。这有助于模型将其在预训练期间分别学到的事实和见解结合起来。原始论文表明，大模型往往无法将两个事实组合在一起，即使它们知道每个独立的事实，例如，一个模型可能知道事实 A 和事实 B，但却无法推导出 A 和 B 在一起的蕴涵。自我提问法旨在克服这一局限性。测试时，只需向模型提供提示和问题，然后它就会自动生成任何必要的后续问题，将事实联系起来，组成推理步骤，并决定何时停止。

4. 路由查询引擎

路由查询引擎(Router Query Engine，RQE)是一个用于在多种查询转换策略之间进行选择的组件，它可以根据输入提示动态地决定应用哪一种转换工具。这种设计允许模型根据不同查询的独特需求灵活选择合适的处理方法，无论是无转换、子问题分解还是多步转换。简而言之，RQE 充当了一个决策者角色，它根据输入提示的特点和复杂性，智能地选择最合适的查询转换策略，从而优化整个查询流程的性能和效率。

事实证明，每种查询转换都适用于不同的情况。对于可以分解成更简单子问题的问题，子问题分解最有效。多步转换最适合需要反复探索上下文的查询，如连接多个信息面。简单的查询可能根本不需要任何转换，应用转换会浪费资源。为了在所有这些情况中做出选择，可以使用路由器查询引擎为大模型提供一系列查询转换工具，让它根据输入提示来决定应用哪种最佳工具。通过高级查询转换增强 RAG 可以显著地提高模型的性能。虽然查询转换只是改进检索的众多技术之一，但它显示了将检索与大模型固有的推理能力相结合的定制分析的潜力和需求。

5.1.7 RAG 中的智能体

智能体的概念在 RAG 系统中扮演着至关重要的角色，更详细的智能体的内容会在第 6 章中展开讨论。自从首个大模型 API 发布以来，智能体的概念就一直被广泛探讨和应用。智能体的设计初衷是为具备推理能力的大模型提供一个工具集，以及一系列需要完成的任务。这些工具不仅包括确定性函数，如代码函数或外部 API 调用，甚至还包括其他代理。这种将大模型与各种工具相连接的思想正是 LangChain 的来源。OpenAI 助手实际上已经整合了许多围绕大模型所需的功能，其中最核心的是函数调用 API。该 API 能够将自然语

言指令转换为对外的工具或数据库查询的 API 调用，极大地提升了大模型的实用性和灵活性。在 LLaMAIndex 中，OpenAIAgent 类将这种高级逻辑与 ChatEngine 和 QueryEngine 相结合，提供了一个基于知识和上下文感知能力的聊天功能，并且能够在一次对话中调用多个 OpenAI 函数，真正实现了智能代理的行为。下面以多文档的智能体为例，如图 5-3 所示。

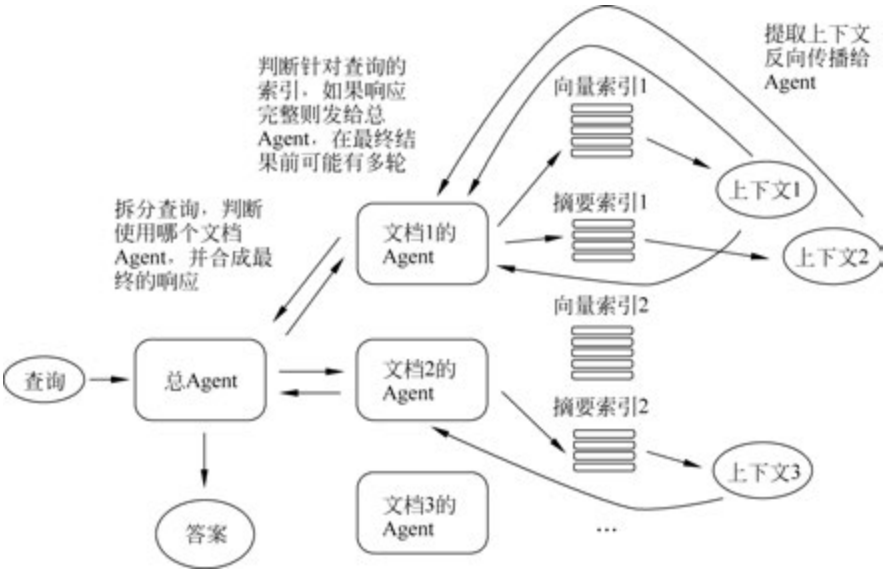

图 5-3　RAG 多文档的智能体

多文档智能体会在每个文档上初始化一个代理(OpenAI Agent)，这个代理能够进行文档摘要和经典问答机制的处理。此外，还有一个顶层总代理负责将查询路由到各个文档代理，并最终合成答案。每个文档代理配备了两个工具：向量存储索引和摘要索引，并根据路由查询的需求决定使用哪一个工具。这种体系结构涉及了大量的路由决策，由各个相关的代理共同完成。这种方案的一个优点在于它能够比较来自不同文档及其摘要的不同解决方案或实体，覆盖了与文档集进行交互的最常见场景。然而，由于它在内部使用了大模型进行多次来回迭代，因此在速度上可能会稍显缓慢。为了提高效率，特别是在处理大型多文档存储的情况下，可以对这一方案进行一定的简化，以实现更好的可扩展性。

5.1.8　响应合成

响应合成是任何 RAG 流水线的最后一步，根据检索的所有上下文和初始用户查询生成一个答案。一种简单直接的合成方法是，将所有超过一定相关性阈值的上下文片段与原始查询语句直接拼接起来，然后将这个组合后的长字符串提供给大模型。然而，这种做法可能过于简单，有时并不能充分考虑到上下文之间的关联性和层次结构。为了获得更加精确和高质量的答案，研究人员开发了更为复杂的响应合成策略。这些方法包括但不限于以下几种。

(1) 迭代细化：首先将检索到的上下文分成若干块，然后逐一将这些块送入大模型进

行理解和处理，逐步构建起完整的答案。每块上下文都可能触发大模型生成一部分答案，随着更多上下文信息的加入，答案也会相应地进行调整和优化。

(2) 上下文总结：在对检索到的信息进行深入分析的基础上，提炼关键信息，形成一个紧凑的总结，以便更好地符合用户的查询提示。这个过程可能需要大模型展现出高度的概括能力和理解力，以确保总结既全面又精准。

(3) 多路径生成：依据不同的上下文块，大模型可能会生成多个潜在的答案。随后，这些答案可以被合并、排序或者进一步加工，形成最终的输出。这可能涉及对多个候选答案进行评估和选择，以确保最终输出的质量和一致性。

在实际操作中，响应合成策略的选择和应用需要根据具体的应用场景、用户需求和系统配置等因素综合考虑。有效的响应合成不仅可以显著地提升回答的质量和相关性，还能增强用户体验，提高整体的信息检索效能。

5.1.9 大模型微调和 RAG 的优劣势对比

在更新大模型的知识方面，微调模型和使用 RAG 这两种方法各有千秋。微调模型的优势在于能够通过有监督学习的方式，通过对任务相关数据的反复迭代调整，使模型更好地适应特定领域的知识和要求。这种方法允许模型在保留原有预训练知识的基础上，有针对性地优化其在特定任务上的表现。然而，它的缺点在于知识更新的成本较高，需要大量的时间和计算资源来重新训练模型，并且对新信息的适应能力相对较弱。另外，RAG 技术结合了检索系统和生成模型的优点，能够从外部知识库中检索最新、最准确的信息，从而提高了答案的质量和时效性。RAG 的优势在于它可以高效地利用外部资源，特别适合处理各类数据库和动态更新的数据源。这种方法使模型能够即时获取并利用最新的外部信息，从而更好地适应当前的事件和知识变化。不过，RAG 也可能面临检索质量问题，如检索到的信息可能不够准确或与查询不匹配，以及增加额外的计算资源需求。在知识更新方面，RAG 支持实时更新检索库，非常适合处理动态数据和快速变化的信息环境，而无须频繁地重新训练模型。相比之下，微调模型通常用于存储静态信息，更新知识需要重新训练，这对于快速变化的领域来讲可能不够灵活。下面通过一张表格对此更加详细地进行对比，见表 5-1。

表 5-1 大模型微调和 RAG 对比

特 性	RAG 技术	大模型微调
知识更新	实时更新检索库，适合处理动态数据，无须频繁地重新训练	存储静态信息，更新知识需要重新训练
外部知识	高效利用外部资源，适合各类数据库，尤其适合处理动态更新的数据源	可对齐外部知识，但对于动态数据源不够灵活
数据处理	数据处理需求较低，便于快速响应	需要构建高质量的数据集，数据限制可能影响模型性能
模型定制化	专注于信息检索和整合，定制化程度相对较低	可定制行为、风格及领域知识，定制化程度更高

续表

特　性	RAG 技术	大模型微调
可解释性	答案可追溯，具有较高的解释性	解释性相对较低，可能需要额外的解释机制
计算资源	需要支持检索的计算资源，以及维护外部数据源的资源	需要训练数据集和微调资源，以及可能的持续训练资源
延迟要求	数据检索可能会导致延迟，尤其是在处理大量数据时	微调后的模型反应更快，延迟较低
减少幻觉	基于实际数据，有助于减少幻觉	通过特定域训练可以减少幻觉，但仍有限制
道德和隐私	处理外部文本数据时需考虑隐私和道德问题	训练数据的敏感内容可能引发隐私问题，需谨慎处理

总体来讲，大模型微调和 RAG 技术在处理外部知识方面各有特点。微调模型侧重于通过训练数据的有监督学习来优化模型，而 RAG 则侧重于通过高效的检索机制来集成最新的外部信息。在选择合适的方法时，需要考虑任务的性质、数据的可获得性及对模型更新频率的要求等因素。在深入探讨了 RAG 技术原理后，接下来详细讲解文本向量模型，在上面讲解分块与向量化、搜索索引、重新排序内容时都涉及了文本向量模型，它能够将文本数据转换为数值向量，从而为各种下游任务提供支持。

5.2　文本向量模型

RAG 是解决大模型幻觉的一个非常有用的技术手段，而 RAG 有效的关键是可以检索到相关的知识。一般来讲，检索主要分为两个阶段。第一阶段是用嵌入(Embedding)模型来召回相关的多个文本，然后用排序(Reranker)模型来进行排序，最后将检索到的知识和用户的查询输入构造 Prompt 提示词，输入大模型，进而返回更加精准的回答。目前，许多研究机构已经开源了一系列模型，以支持这一过程，例如，在 Embedding 模型方面有 GTE、BGE、M3E、BCE 等，而在排序模型方面，则有 bge-rerank、bce-rerank 等；此外，还有结合 Embedding 模型和 Reranker 模型特点的 ColBERT 模型，如 bge-m3-colbert，接下来详细讲解。

5.2.1　Embedding 模型、Reranker 模型及 ColBERT 模型

在自然语言处理和信息检索领域，Embedding 模型、ColBERT 模型及 Reranker 模型都是重要的技术，它们各自有不同的特点和用途。

(1) Embedding 模型：这类模型主要用于将文本数据转换为数值向量，以便计算机能够理解和处理。这些向量捕捉了文本中的语义和句法信息，使相似的文本在向量空间中彼此靠近。传统的 Embedding 模型有 Word2Vec、GloVe 和 FastText 等。这些模型通过学习词汇在大量文本数据中的共现模式来生成词嵌入，被广泛地应用于文本分类、情感分析、机器翻译等任务，近年来，随着深度学习技术的发展，基于 Transformer 架构的 Embedding 模型逐渐崭露头角。这些模型不仅在传统 Embedding 模型的基础上进行了改进，还引入了更

多的创新元素，使它们在处理复杂的自然语言任务时更加高效和准确，例如，BGE 模型就是一种基于 Transformer 架构的 Embedding 模型，它通过多层自注意力机制来捕捉文本中的长距离依赖关系，从而生成更加丰富和准确的词嵌入。M3E 模型则是一种多语言的 Embedding 模型，它能够同时处理多种语言的数据，为跨语言的信息检索和翻译任务提供了强大的支持。BCE 模型则专注于提供更加精细的文本表示，通过引入上下文信息和注意力机制，使生成的词嵌入更加符合实际的语言使用情况。这些基于 Transformer 架构的 Embedding 模型不仅在学术界取得了显著的研究成果，也在工业界得到了广泛应用。它们为自然语言处理领域带来了新的发展机遇，也为未来的研究提供了新的方向和思路。

(2) Reranker 模型：这类模型通常在检索阶段之后使用，其目的是对已检索到的文档列表进行二次排序，以提高检索结果的准确性。Reranker 模型通常基于机器学习算法，如支持向量机、随机森林或神经网络，它们在训练过程中学习如何根据查询和文档的特征来评估它们的相关性。Reranker 模型能够考虑更多的特征，如文档的长度、查询和文档之间的语义距离等，从而有可能提升检索结果的质量。

(3) ColBERT 模型：Embedding 模型和 Reranker 模型在信息检索系统中扮演着重要角色，而 ColBERT 模型则是结合这两种方法的一种尝试，旨在提高文本匹配的相关性得分精度。ColBERT 模型采用了一种被称为“延迟交互”的策略，先分别对查询和文档进行编码，然后使用轻量且有效的模块对两者之间的相关性进行细粒度建模。这种方法允许 ColBERT 模型离线预先计算好文档的表示，从而加速在线查询的处理速度。在实际应用中，这意味着可以先建立好文档的索引，当有新的查询到来时，只需对查询进行编码，并通过近似近邻检索等技术快速找到与查询相似的文档。总体来讲，ColBERT 模型通过结合 Embedding 模型的编码能力和 Reranker 模型的排序能力，实现了在保持 BERT 编码能力的同时，提升了在线处理的效率和检索结果的准确性。这种模型特别适合于大规模的信息检索系统，能够在保证检索质量的同时，显著降低延迟，提高用户体验。

综上所述，Embedding 模型、Reranker 模型和 ColBERT 模型在自然语言处理和信息检索中都发挥着重要作用，它们各自针对不同的问题和挑战提供了有效的解决方案。接下来深入讲解主流的向量模型。

5.2.2 阿里巴巴 GTE 向量模型

通用文本嵌入模型(General Text Embeddings，GTE)模型，是一种基于多阶段对比学习的通用句向量模型，由阿里巴巴团队提出。GTE 模型在大规模文本嵌入基准(Massive Text Embedding Benchmark，MTEB)评测排行榜上取得了优异的成绩，超过了其他竞争模型如 Instructor 和 E5。MTEB 是衡量文本嵌入模型(Embedding 模型)的评估指标的合集，是目前业内评测文本向量模型性能的重要参考。对应的 C-MTEB 则是专门针对中文文本向量的评测基准，被公认为目前业界最全面、最权威的中文语义向量评测基准之一，为深度测试中文语义向量的全面性和可靠性提供了可靠的实验平台。阿里巴巴、腾讯、商汤、百川等多家厂商在此榜单测评发布模型。GTE 模型采用了 Transformer 编码器，具有 3 种不同

的模型尺寸，分别以 MiniLM-small、BERT-BASE 和 BERT-LARGE 作为模型初始化。在训练过程中，GTE 使用双塔模型结构，将语言模型生成的所有位置的最后一层隐状态的平均值作为句子表征。与传统的对比学习不同，GTE 采用了一个改进的对比损失函数，对负样本进行了更多的扩充。GTE 的训练过程分为预训练和微调两个阶段，两个阶段都使用了对比学习和改进的对比损失函数。在预训练阶段，研究人员收集了各种领域的开源数据，包括网页搜索、科学文献、社区问答、社交媒体、维基百科和代码仓库等，总共接近 8 亿的文本对（Query，Positive Document）。为了维持数据平衡并防止模型学习到不同任务的特性，采用了特定的抽样策略，并确保同一批次的数据来自同一任务。在微调阶段，研究人员基于少量的人工标注数据集，利用额外的检索器获得 Hard Negative 数据，构造出文本相关性三元组（Query，Positive Document，Negative Document）数据，约 300 万条，使模型在高质量数据上进一步微调。实验结果显示，GTE 模型的性能与训练数据的数量（无论是预训练还是微调）、模型本身的容量存在正相关关系。预训练和微调对于句向量模型的性能都是必要的，缺一不可。此外，与传统的只进行微调的方法相比，GTE 等当前的 SOTA 模型在性能上有了显著提升。这可能是因为基底模型本身的能力有限，而针对性的预训练可以进一步提高基底模型的上限，从而提升模型的整体表现。

总体来讲，GTE 模型通过显著增加训练数据，实现了性能的大幅提升。下面讲解 GET 工具如何使用，以中文 nlp_gte_sentence-embedding_chinese-large 模型为例。

1. 使用模型

文本表示是自然语言处理领域的核心问题，其在很多自然语言处理、信息检索的下游任务中发挥着非常重要的作用。近几年，随着深度学习的发展，尤其是预训练语言模型的出现极大地推动了文本表示技术的效果，基于预训练语言模型的文本表示模型在学术研究数据及工业实际应用中都明显优于传统的基于统计模型或者浅层神经网络的文本表示模型。这里主要关注基于预训练语言模型的文本表示。

文本表示示例，输入一个句子，输出一个固定维度的连续向量。

输入：

吃完海鲜可以喝牛奶吗？

输出：

[0.27162，－0.66159，0.33031，0.24121，0.46122，…]

文本的向量表示通常可以用于文本聚类、文本相似度计算、文本向量召回等下游任务中。GTE 模型可以使用在通用领域的文本向量表示及其下游应用场景，包括双句文本相似度计算、查询和多文档候选的相似度排序。

在 ModelScope 框架上，提供输入文本（默认最大文本长度为 128），即可通过简单的 Pipeline 调用来使用 GTE 文本向量表示模型。ModelScope 封装了统一的接口对外提供单句向量表示、双句文本相似度、多候选相似度计算功能。使用 GTE 模型的代码如下：

```
#第 5 章/GTESentenceEmbeddingChineseLarge.py
from modelscope.models import Model
```

```
from modelscope.pipelines import pipeline
from modelscope.utils.constant import Tasks
model_id = "iic/nlp_gte_sentence-embedding_chinese-large"
pipeline_se = pipeline(Tasks.sentence_embedding,
                   model=model_id,
                   sequence_length=512
                   ) #sequence_length 代表最大文本长度,默认值为 128
#当输入包含 source_sentence 与 sentences_to_compare 时会输出 source_sentence 中首个
#句子与 sentences_to_compare 中每个句子的向量表示,以及 source_sentence 中首个句子与
#sentences_to_compare 中每个句子的相似度。
inputs = {
        "source_sentence": ["吃完海鲜可以喝牛奶吗?"],
        "sentences_to_compare": [
            "不可以,早晨喝牛奶不科学",
            "吃了海鲜后是不能再喝牛奶的,因为牛奶中含有维生素 C,如果海鲜和牛奶一起服
用,则会对人体造成一定的伤害",
            "吃海鲜是不能同时喝牛奶和吃水果的,这个至少间隔 6h 以上才可以。",
            "吃海鲜是不可以吃柠檬的,因为其中的维生素 C 会和海鲜中的矿物质形成砷"
        ]
    }
result = pipeline_se(input=inputs)
print (result)
#当输入仅含有 source_sentence 时会输出 source_sentence 中每个句子的向量表示及首个句
#子与其他句子的相似度
inputs2 = {
        "source_sentence": [
            "不可以,早晨喝牛奶不科学",
            "吃了海鲜后是不能再喝牛奶的,因为牛奶中含有维生素 C,如果海鲜和牛奶一起服
用,则会对人体造成一定的伤害",
            "吃海鲜是不能同时喝牛奶和吃水果的,这个至少间隔 6h 以上才可以。",
            "吃海鲜是不可以吃柠檬的,因为其中的维生素 C 会和海鲜中的矿物质形成砷"
        ]
}
result = pipeline_se(input=inputs2)
print (result)
```

2. 训练微调模型

本模型基于中文通用领域数据上训练,在垂类领域效果无法保证,可以在垂类领域数据下通过训练微调来提高性能,代码如下:

```
#第 5 章/TrainGTE.py
#此脚本需要在支持 GPU 的环境中运行
#注意:加载数据集的过程可能会因为网络问题而失败,如果发生这种情况,则可尝试重新运行代码
#导入必要的模块和类
from modelscope.metainfo import Trainers          #用于获取训练器元信息
from modelscope.msdatasets import MsDataset       #用于加载和管理数据集
from modelscope.trainers import build_trainer     #用于构建训练器
import tempfile                                   #用于创建临时目录
```

```
import os                                          #提供与操作系统交互的功能
#创建一个临时目录,用来存储训练过程中的文件
tmp_dir = tempfile.TemporaryDirectory().name
if not os.path.exists(tmp_dir):
    os.makedirs(tmp_dir)                           #如果目录不存在,则创建它
#加载数据集
ds = MsDataset.load('dureader-retrieval-ranking', 'zyznull')
                                                   #从 ModelScope 平台加载指定的数据集
train_ds = ds['train'].to_hf_dataset()             #将训练集转换为 Hugging Face 的 Dataset 格式
dev_ds = ds['dev'].to_hf_dataset()                 #将验证集转换为 Hugging Face 的 Dataset 格式
#定义模型 ID
model_id = 'iic/nlp_gte_sentence-embedding_chinese-large'
#指定要使用的预训练模型 ID
#定义配置修改函数
def cfg_modify_fn(cfg):
    """
    修改模型训练的配置参数。
    :param cfg: 原始的配置字典
    :return: 修改后的配置字典
    """
    cfg.task = 'sentence-embedding'                #将任务类型设置为句子嵌入
    cfg['preprocessor'] = {'type': 'sentence-embedding', 'max_length': 256}
                                                   #设置预处理器类型和最大长度
    cfg['dataset'] = {
        'train': {
            'type': 'bert',                        #使用 BERT 模型处理训练数据
            'query_sequence': 'query',             #查询序列字段名
            'pos_sequence': 'positive_passages',   #正例序列字段名
            'neg_sequence': 'negative_passages',   #负例序列字段名
            'text_fields': ['text'],               #文本字段列表
            'qid_field': 'query_id'                #查询 ID 字段名
        },
        'val': {
            'type': 'bert',                        #使用 BERT 模型处理验证数据
            'query_sequence': 'query',             #查询序列字段名
            'pos_sequence': 'positive_passages',   #正例序列字段名
            'neg_sequence': 'negative_passages',   #负例序列字段名
            'text_fields': ['text'],               #文本字段列表
            'qid_field': 'query_id'                #查询 ID 字段名
        },
    }
    cfg['train']['neg_samples'] = 4                #每个正例对应的负例样本数
    cfg['evaluation']['dataloader']['batch_size_per_gpu'] = 30
    #每个 GPU 的评估批次大小
    cfg.train.max_epochs = 1                       #最大训练轮数
    cfg.train.train_batch_size = 4                 #训练批次大小
    return cfg                                     #返回修改后的配置
#构建训练器的参数
kwargs = dict(
```

```
    model=model_id,                    #指定使用的模型 ID
    train_dataset=train_ds,            #指定训练数据集
    work_dir=tmp_dir,                  #指定工作目录(临时目录)
    eval_dataset=dev_ds,               #指定验证数据集
    cfg_modify_fn=cfg_modify_fn        #指定配置修改函数
)
#构建并初始化训练器
trainer = build_trainer(name=Trainers.nlp_sentence_embedding_trainer, default_
args=kwargs)
#开始训练模型
trainer.train()                        #调用训练器的 train 方法进行模型训练
```

这段代码的主要功能是利用 ModelScope 平台的 API 来加载数据集、配置模型训练参数，并启动训练过程。在代码中首先创建了临时目录以存放在训练过程中产生的文件，然后加载了特定的数据集，并设置了模型训练的相关参数。最后，通过 build_trainer 函数构建了训练器对象，并调用了其 train 方法来执行训练。

5.2.3 中文 acge_text_embedding 模型

中文文本向量化 acge_text_embedding 模型是由国内自主研发的文本向量化模型，简称 acge 模型，其在 C-MTEB 这一权威的中文语义向量评测基准中荣获第一，标志着我国在文本向量化技术领域取得了重大突破。C-MTEB 集合了 35 个公开数据集，覆盖了分类、聚类、检索、排序、文本相似度等六大类任务，为中文向量化模型的评估提供了统一标准和有力支持。

1. acge 模型的工作原理

acge 模型的核心在于采用了俄罗斯套娃表征学习(Matryoshka Representation Learning，MRL)框架。这一框架借鉴了俄罗斯套娃层层嵌套的理念，旨在构建一个多粒度、嵌套式的向量表示体系。在 MRL 中，生成的向量嵌套结构允许不同大小的向量从同一模型中输出，每个向量都对应着不同级别的语义表达，并且能独立地应用于不同的任务场景。在训练过程中，MRL 针对不同维度的向量计算损失函数，确保模型能够根据实际需求，仅通过调整输入参数即可输出指定维度的向量，大大地增强了模型的灵活性和实用性。

2. acge 模型的性能与应用

acge 模型在多个维度展现了卓越的性能，例如，对于烹饪相关文本的相似度计算，模型给出了高度相关的相似度值，表明其在捕捉特定领域内文本间语义关联方面的能力。这不仅对文本分类、推荐系统等任务至关重要，也证明了 acge 模型能够准确地捕获文本之间的语义相似性，实现有效的文本区分。在 C-MTEB 的综合评测中，该模型在 9 个数据集的分类任务平均得分达到 72.75，在全部 35 个数据集的平均得分更是高达 69.07，远超同类模型，如 Baichuan-text-embedding 和阿里云的 OpenSearch-text-hybrid。这表明 acge 模型在处理多样化任务时具有更高的性能和灵活性，同时也体现了其在模型大小和计算效率上的平衡。

3. 技术创新与应用价值

acge 模型在技术创新上实现了策略学习训练方式，显著地提升了在检索、聚类、排序等

任务上的性能。此外，它还引入了持续学习训练方式，有效地解决了神经网络的灾难性遗忘问题，确保了模型在迭代训练过程中的良好收敛性。这一系列技术手段不仅提升了模型的性能表现，也为模型在不同场景下的快速部署和应用创造了条件。acge 模型能够帮助构建通用分类模型，提高长文档信息抽取的精确度，同时保持较低的应用成本，加速了大模型在金融、咨询、教育等多行业的价值创造。

4. acge 模型使用代码实践

acge 模型提供了预训练好的模型以供使用，首先安装 sentence_transformers 依赖，命令如下：

```
pip install --upgrade sentence_transformers
```

安装完成后，可将源文本 source_text 设置为“家常菜烹饪指南”，将想要计算相似度的目标文本 target_text 设置为[“西红柿炒鸡蛋做法”，“农家小炒肉做法”，“上海本帮菜肴传统烹饪技艺”，“汽车维修指南——检测、维修、拆装与保养”]进行测试，代码如下：

```
#第 5 章/acgeTextEmbedding.py
from sentence_transformers import SentenceTransformer
#初始化 SentenceTransformer 模型，使用'acge_text_embedding'模型
model = SentenceTransformer('acge_text_embedding')
#源文本列表，这里只有一个文本"家常菜烹饪指南"
source_text = ["家常菜烹饪指南"]
#目标文本列表，包含多个文本
target_text = [
    "西红柿炒鸡蛋做法", "农家小炒肉做法", "上海本帮菜肴传统烹饪技艺",
    "汽车维修指南——检测、维修、拆装与保养"
]
#使用 model.encode 方法将源文本和目标文本转换为向量表示
#将源文本转换为向量
embs1 = model.encode(source_text, normalize_embeddings=True)
#将目标文本转换为向量
embs2 = model.encode(target_text, normalize_embeddings=True)
#计算源文本向量和目标文本向量之间的相似度
#这里使用了矩阵乘法来计算相似度，即 embs1 的转置与 embs2 的乘积
similarity = embs1 @embs2.T
#打印相似度结果
print(similarity)
```

这段代码首先导入了 sentence_transformers 库中的 SentenceTransformer 类，然后创建了一个名为 model 的 SentenceTransformer 实例，该实例使用了名为 acge_text_embedding 的预训练模型。接下来，代码定义了两个文本列表：source_text 和 target_text，然后使用 model. encode 方法将这些文本转换为向量表示，并计算这些向量之间的相似度。最后，代码打印出相似度的结果，最终计算结果如下。

西红柿炒鸡蛋做法：0.495

农家小炒肉做法：0.618

上海本帮菜肴传统烹饪技艺：0.581

汽车维修指南——检测、维修、拆装与保养：0.277

本示例中，观察到数值反映了源文本“家常菜烹饪指南”与目标文本之间的语义亲密度。相似度得分越趋于1，意味着文本间的语义联系越紧密。通过这些结果，可以看出不同领域文本与源文本之间的相似度评估。对于那些与烹饪主题紧密相关的文本，例如“西红柿炒鸡蛋做法”“农家小炒肉做法”和“上海本帮菜肴传统烹饪技艺”，文本向量化模型展现出了较高的相似度得分。这证明了该模型在捕捉烹饪领域文本间语义联系方面的有效性。这种模型能够为具有相似主题或语义的文本提供精确的相似度评估，这对于文本分类、推荐系统等应用场景至关重要。然而，对于与汽车维修相关的文本，相似度得分较低，这是因为这些文本与源文本的语义联系较弱。这凸显了模型的另一项优势，即其能够根据文本内容识别并捕捉不同领域的语义特征，从而实现对文本的有效区分。总之，acge模型能够高效地从文本中提取语义特征，并将其转换为向量形式，同时还能准确地衡量文本间的语义相关性。

acge模型的突破性进展不仅巩固了中国在文本向量化技术领域的领先地位，也为文本处理和自然语言理解技术的发展开辟了新路径。该模型的成功不仅体现在技术层面的创新，更在于在实际应用中展现的广泛适用性和经济性。acge模型不仅为文本检索、分类、推荐等传统任务提供了强有力的工具，也为构建更加智能化、个性化的服务系统奠定了坚实的基础。

5.2.4 智源BGE通用向量模型

随着人工智能技术的飞速发展，向量模型(Embedding Model，EM)已经成为支撑搜索、推荐系统、数据挖掘等多个关键领域发展的核心技术之一。尤其是在大模型广泛应用的背景下，语义向量模型对于解决大模型在实际应用中遇到的幻觉问题、知识时效性问题及超长文本处理限制等问题发挥着至关重要的作用。近期，北京智源人工智能研究院发布了一款名为BGE(BAAI General Embedding)的开源可商用中英文语义向量模型，该模型在多项评估中表现卓越，标志着语义向量技术的一个重要里程碑。

1. BGE模型的突破性成就

BGE模型的发布，解决了中文世界高质量语义向量模型稀缺且开源少的问题。它不仅在中英文语义检索精度和整体语义表征能力上超越了社区内所有已知的同类模型，包括OpenAI的Text Embedding 002，而且在同等参数量级的情况下，保持了最小的向量维度，这意味着使用BGE模型的成本更低，更加高效。BGE模型的开源代码位于FlagEmbedding项目下，这是北京智源人工智能研究院旗下面向Embedding技术和模型的开源体系——FlagOpen大模型技术开源体系的一部分。

2. C-MTEB：填补中文评测基准空白

鉴于中文社区在评测基准上的缺失，智源团队还构建了C-MTEB(Chinese Massive Text Embedding Benchmark)，这是一个大规模、全面的中文语义向量表征能力评测基准，它覆盖了检索、排序、句子相似度、推理、分类、聚类六大评测任务类别，涉及31个数据集，为中文语义向量的综合能力评价奠定了坚实的基础。C-MTEB的发布，使开发者和研究人员

能够更准确、更全面地评估模型性能，推动技术进步。

3. BGE 模型的技术创新点

BGE 模型之所以能取得如此显著的性能提升，主要得益于两个关键因素：针对表征的预训练算法 RetroMAE 和大规模文本对训练。在悟道和 Pile 两大语料库上，BGE 采用了 RetroMAE 算法，通过低掩码率的输入生成语义向量，再与高掩码率的输入进行拼接，以此来重建原始输入，巧妙地利用无标签数据进行预训练，提升了模型对语义表征任务的适应性。此外，BGE 构建了超过 1.2 亿对中文样本和 2.3 亿对英文样本的训练数据，通过负采样扩增和负样例挖掘技术，增强模型的判别能力，实现高达 6.5 万的负样本规模，显著地提高了语义向量的区分度。BGE 还创新性地引入了非对称指令添加方式，即在问题端加入场景描述，进一步地提升了模型在多任务场景下的泛化能力。

4. 应用价值与影响

BGE 模型的发布，对于大模型应用开发者来讲是一个巨大的福音。结合 LangChain 等框架，开发者可以轻松地定制本地知识问答助手，无须投入大量资源重新训练特定领域的大型模型。这一特性对于降低开发成本、加速应用场景落地具有重要意义。BGE 模型不仅提升了大模型在阅读理解、开放域问答、知识型对话等复杂任务中的表现，还通过构建垂直领域知识库索引，帮助大模型实时获取最新的世界知识和本地知识，解决了大模型因知识陈旧而导致的回答不准确问题，增强了大模型的“活知识”获取能力。同时，通过将长文档结构化，BGE 模型还有效地缓解了大模型处理长文本时的内存限制，提高了处理长文本的能力，从而更好地服务于复杂的信息检索和知识管理场景。

5. BGE 模型使用代码实践

BGE 模型将任意文本映射为低维稠密向量，以用于检索、分类、聚类或语义匹配等任务，并可支持为大模型调用外部知识。BGE 模型发布了中英文、small、base、large 多个模型，见表 5-2。

表 5-2 BAG 模型列表

模　　型	语言	描　　述
BAAI/bge-large-en	英文	在 MTEB 榜单上排名第一
BAAI/bge-base-en	英文	在 MTEB 榜单上排名第二
BAAI/bge-small-en	英文	small-scale 模型性能高于很多开源 large-scale 模型，推理更高效
BAAI/bge-large-zh	中文	在 C-MTEB 榜单上排名第一
BAAI/bge-large-zh-noinstruct	中文	在 C-MTEB 榜单上排名第二
BAAI/bge-base-zh	中文	base-scale 模型与 bge-large 性能类似，但推理更快，向量维度更小
BAAI/bge-small-zh	中文	small-scale 模型，推理比 base 模型更快

在模型使用过程中，如果为一个短查询搜索相关的长文档，则需要在查询中添加指令；在其他情况下，不需要指令，直接使用原始查询即可。在任何情况下都不需要为候选文档增加指令。还有一些常见问题如下。

(1) 不相似句子之间的相似度分数很高：当前 BGE 模型的相似度分布并不是[0,1]区

间的均匀分布，其大概处于[0.6,1]区间，因此并不是大于0.6就代表相似。尤其是对于长度较短句子之间的相似度，当前模型的相似度数值会偏高。对于检索任务或者相似度任务，影响结果的是不同句子间相似度的相对大小关系，而不是绝对数值。如果需要根据相似度阈值筛选相似句子，则需要根据实际数据上的相似度分布情况，使用一个合适的相似度阈值（如0.8、0.85，甚至是0.9）。

（2）什么时候需要添加查询指令对于一个使用短查询寻找相关长文档的检索任务，查询与文档之间长度非常不一致，推荐为短查询添加指令。其他任务，推荐不添加指令，例如，像Quora这类用一个较短的问题去搜索其他相关的短问题，推荐不添加指令。具体是否添加指令，可以根据实际情况选择其中表现最好的方式。在所有情况下，文档端都不用添加指令，只是查询端可以选择是否添加指令。

下面通过代码示例演示模型如何使用，BGE模型项目源代码开源地址是https://github.com/FlagOpen/FlagEmbedding。可以通过FlagEmbedding、Sentence-Transformers、LangChain、HuggingFace Transformers等多种方式来使用BGE模型。

1）使用FlagEmbedding

先安装FlagEmbedding库，命令是pip install -U FlagEmbedding，使用此库的代码如下：

```
#第5章/BGEFlagEmbedding.py
#导入FlagEmbedding模块中的FlagModel类
from FlagEmbedding import FlagModel
#定义两个示例句子
sentences = ["样例数据-1", "样例数据-2"]
#初始化FlagModel模型，指定预训练模型'BAAI/bge-large-zh'和查询指令
model = FlagModel('BAAI/bge-large-zh', query_instruction_for_retrieval="为这个
句子生成表示以用于检索相关文章：")
#对样例数据进行编码，获取它们的嵌入向量
embeddings_1 = model.encode(sentences)
embeddings_2 = model.encode(sentences)
#计算两组嵌入向量的相似度矩阵
similarity = embeddings_1 @embeddings_2.T
#打印相似度矩阵
print(similarity)
#对于短查询到长文档的检索任务，需要对查询使用encode_queries()函数，该函数会自动为每
#个查询加上指令
#由于候选文本不需要添加指令，所以检索中的候选集依然使用encode()或encode_corpus()函数
#定义查询语句列表
queries = ['query_1', 'query_2']
#定义候选文档列表
passages = ["样例文档-1", "样例文档-2"]
#对查询语句进行编码，自动添加查询指令
q_embeddings = model.encode_queries(queries)
#对候选文档进行编码，无须添加指令
p_embeddings = model.encode(passages)
#计算查询与文档之间的相关性得分
scores = q_embeddings @p_embeddings.T
```

这段代码首先导入了 FlagEmbedding 模块中的 FlagModel 类，并使用预训练的中文模型 BAAI/bge-large-zh 初始化了一个实例，然后分别对一组示例句子进行了两次编码，得到了它们的嵌入向量，并计算了这些向量之间的相似度矩阵。接下来，代码展示了如何在短查询到长文档的检索任务中使用 encode_queries()函数来处理查询语句，并为它们自动添加指令，同时使用 encode()函数来处理候选文档。最后，通过矩阵乘法计算了查询与文档之间的相关性得分。

FlagModel 既支持 GPU 也支持 CPU 推理。如果 GPU 可用，则默认优先使用 GPU。如果想禁止使用 GPU，则可设置 os. environ["CUDA_VISIBLE_DEVICES"]="" 。为了提高效率，FlagModel 默认会使用所有的 GPU 进行推理。如果想要使用具体的 GPU，则可设置 os. environ["CUDA_VISIBLE_DEVICES"]。

2）使用 Sentence-Transformers

首先需要安装 Sentence-Transformers 库，命令是 pip install -U sentence-transformer，基于 Sentence-Transformers 的使用方法的代码如下：

```
#第 5 章/BGESentenceTransformer.py
#导入 SentenceTransformer 模块
from sentence_transformers import SentenceTransformer
#定义两个示例句子
sentences = ["样例数据-1", "样例数据-2"]
#初始化 SentenceTransformer 模型，指定预训练模型'BAAI/bge-large-zh'
model = SentenceTransformer('BAAI/bge-large-zh')
#对样例数据进行编码，获取它们的嵌入向量，并进行归一化
embeddings_1 = model.encode(sentences, normalize_embeddings=True)
embeddings_2 = model.encode(sentences, normalize_embeddings=True)
#计算两组嵌入向量的相似度矩阵
similarity = embeddings_1 @embeddings_2.T
#打印相似度矩阵
print(similarity)
#对于短查询到长文档的检索任务，每个查询都应该以一条指令开始(指令参考 Model List)
#但对于文档，不需要添加任何指令
#定义查询语句列表
queries = ['query_1', 'query_2']
#定义候选文档列表
passages = ["样例文档-1", "样例文档-2"]
#定义查询指令
instruction = "为这个句子生成表示以用于检索相关文章："
#对查询语句进行编码，自动添加查询指令，并进行归一化
q_embeddings = model.encode([instruction+q for q in queries], normalize_embeddings=
True)
#对候选文档进行编码，无须添加指令，并进行归一化
p_embeddings = model.encode(passages, normalize_embeddings=True)
#计算查询与文档之间的相关性得分
scores = q_embeddings @p_embeddings.T
```

这段代码首先导入了 SentenceTransformer 模块，并使用预训练的中文模型 BAAI/bge-large-zh 初始化了一个实例，然后分别对一组示例句子进行了两次编码，得到了它们的

嵌入向量，并计算了这些向量之间的相似度矩阵。接下来，代码展示了如何在短查询到长文档的检索任务中使用 encode()函数来处理查询语句，并为它们自动添加指令，同时使用 encode()函数来处理候选文档。最后，通过矩阵乘法计算了查询与文档之间的相关性得分。

3）使用 LangChain

BGE 模型整合入 LangChain，可以在 LangChain 中非常简单地使用它，在 LangChain 中使用 BGE 模型的代码如下：

```
#第 5 章/BGELangchain.py
#导入 langchain.embeddings 模块中的 HuggingFaceBgeEmbeddings 类
from langchain.embeddings import HuggingFaceBgeEmbeddings
#定义预训练模型名称
model_name = "BAAI/bge-small-en"
#定义模型参数，这里指定在 CPU 上运行
model_kwargs = {'device': 'cpu'}
#定义编码参数，将 normalize_embeddings 设置为 True 以计算余弦相似度
encode_kwargs = {'normalize_embeddings': True}
#初始化 HuggingFaceBgeEmbeddings 模型，传入预训练模型名称、模型参数和编码参数
model = HuggingFaceBgeEmbeddings(
    model_name=model_name,
    model_kwargs=model_kwargs,
    encode_kwargs=encode_kwargs
)
```

这段代码首先导入了 langchain. embeddings 模块中的 HuggingFaceBgeEmbeddings 类，并使用预训练的英文模型 BAAI/bge-small-en 初始化了一个实例。在初始化过程中，指定了模型参数（在 CPU 上运行）和编码参数（计算余弦相似度）。

4）使用 HuggingFace Transformers

使用 Transformers 库时，可以先将输入传递给 Transformer 模型，然后选择第 1 个标记的最后一个隐藏状态（[CLS]）作为句子嵌入，代码如下：

```
#第 5 章/BGEHuggingFaceTransformers.py
#导入 Transformers 模块中的 AutoTokenizer 和 AutoModel 类
from transformers import AutoTokenizer, AutoModel
import torch
#定义我们想要获取句子嵌入的句子列表
sentences = ["样例数据-1", "样例数据-2"]
#从 HuggingFace Hub 加载模型
tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-large-zh')
model = AutoModel.from_pretrained('BAAI/bge-large-zh')
#对句子进行分词
encoded_input = tokenizer(sentences, padding=True, truncation=True, return_
tensors='pt')
#对于短查询到长文档的检索任务为查询加上指令
#encoded_input = tokenizer([instruction + q for q in queries], padding=True, truncation=
True, return_tensors='pt')
#计算嵌入
```

```
with torch.no_grad():
    model_output = model(encoded_input)
    #执行池化操作。在这个例子中,是 CLS 池化
    sentence_embeddings = model_output[0][:, 0]
#对嵌入进行归一化
sentence_embeddings = torch.nn.functional.normalize(sentence_embeddings, p=2, dim=1)
print("Sentence embeddings:", sentence_embeddings)
```

BGE 模型的推出不仅在技术层面上实现了语义向量模型性能的突破,更重要的是,它为中文语义向量技术的评测、应用与生态建设提供了坚实的基础。通过开源共享,BGE 正引领着语义向量模型技术向更加实用、高效、易用的方向发展,为人工智能技术的普及和深化应用铺平了道路。

5.2.5 Moka 开源文本嵌入模型 M3E

M3E 是一项创新的开源中文 Embedding 技术,是由 Moka(北京希瑞亚斯科技)开源的系列文本嵌入模型。M3E 是 Moka Massive Mixed Embedding 的缩写,每个单词的含义如下。

(1) Moka:此模型由 MokaAI 训练、开源和评测,训练脚本使用 Uniem,评测 Benchmark 使用 MTEB-zh。

(2) Massive:此模型通过千万级(2200 万+)的中文句对数据集进行训练。

(3) Mixed:此模型支持中英双语的同质文本相似度计算,以及异质文本检索等功能,未来还会支持代码检索。

(4) Embedding:此模型是文本嵌入模型,可以将自然语言转换成稠密的向量。

M3E 模型是使用千万级的中文句对数据集进行训练的 Embedding 模型,在文本分类和文本检索的任务上都超越了 openai-ada-002 模型(ChatGPT 官方的模型)。该技术凭借其在多个中文 NLP 任务中的卓越表现,特别是在文本分类、情感分析、命名实体识别等领域取得的 SOTA 性能,成为推动中文 NLP 领域进步的重要力量。M3E 模型不仅在技术层面上实现了多模态融合、多层次表达、上下文建模等关键特性,而且在应用层面展示了广泛的适用性和实用性,如智能客服、内容推荐等场景,预示着其在未来 NLP、计算机视觉、语音识别乃至更多领域有着巨大的应用潜力。

1. M3E 模型概述

M3E 模型的核心在于其多模态、多粒度的嵌入技术,即 Multi-Modal Multi-Granularity Embedding。该模型能够将文本、图像、语音等多种模态的数据以一种高维向量的形式表示,从而实现不同媒体数据间的深度融合。通过利用大规模的中文句对数据集,M3E 能够学习到广泛的中文语言知识和语义信息,不仅支持中文,还具备中英双语的同质文本相似度计算能力,未来还计划支持代码检索等功能,其设计思路充分体现了对多模态数据处理的重视,以及对信息全面分析和理解的追求。

2. 技术特点与优势

M3E 模型的技术特点与优势主要有以下几个方面。

(1) 多模态融合：M3E 技术能够整合文本、图像、语音等多种类型的数据，通过嵌入技术提取它们之间的共享特征，使模型在处理复杂任务时能获得更全面的信息分析能力。这一特性在情感分析、图像检索等跨模态任务中尤为重要，能够提升模型的综合理解力。

(2) 多层次表达：M3E 不仅关注单一层面的特征，还构建了多层次的向量表达体系，涵盖了从词语、句子到篇章的多层级信息。这种策略有助于模型更细致地把握数据的细微差别，提升在文本相似度、机器翻译等任务中的表现。

(3) 上下文建模：模型在训练过程中考虑上下文信息，确保了对文本语境的准确把握，这对于准确理解文本含义、进行语义分析等任务至关重要。

(4) 可扩展性与兼容性：M3E 模型因其通用性，能够被广泛地应用于自然语言处理、计算机视觉、语音识别等多个领域。同时，通过 sentence-transformers 库，模型可以被轻松地集成到现有的 NLP 系统中，便于用户进行二次开发和定制化应用。

3. 应用前景与性能表现

在实际应用方面，M3E 模型展示出显著的优越性，例如，在文本分类任务中，M3E-base 模型在 6 个公开数据集上的平均准确率达到了 0.6157，优于 openai-ada-002 的 0.5956；在 T2Ranking 1W 中文数据集的检索排序任务上，M3E-base 在 ndcg@10 指标上达到了 0.8004，同样优于 openai-ada-002 的 0.7786。这些实证结果验证了 M3E 在处理中文 NLP 任务时的高效性与准确性。

4. 实践与部署

1) 使用 M3E 模型

先安装 sentence-transformers 库，命令是 pip install -U sentence-transformers，安装完成后就可以使用 M3E 模型了，代码如下：

```
#第 5 章/M3ESentenceTransformers.py
#导入 SentenceTransformer 模块
from sentence_transformers import SentenceTransformer
#加载预训练的 M3E 基础模型
model = SentenceTransformer('moka-ai/m3e-base')
#我们想要编码的句子列表
sentences = [
    '*Moka 此文本嵌入模型由 MokaAI 训练并开源，训练脚本使用 Uniem',
    '*Massive 此文本嵌入模型通过千万级的中文句对数据集进行训练',
    '*Mixed 此文本嵌入模型支持中英双语的同质文本相似度计算，以及异质文本检索等功能，未来还会支持代码检索，ALL in one'
]
#通过调用 model.encode()方法对句子进行编码
embeddings = model.encode(sentences)
#打印每个句子的嵌入向量
for sentence, embedding in zip(sentences, embeddings):
    print("Sentence:", sentence)
    print("Embedding:", embedding)
    print("")
```

这段代码首先导入了 SentenceTransformer 模块，然后加载了一个预训练的 M3E 基础模型。接着定义了一个包含 3 个句子的列表，这些句子描述了 M3E 模型的一些特性。之后，使用 model.encode()方法对这些句子进行编码，得到它们的嵌入向量。最后，遍历每个句子及其对应的嵌入向量，并将它们打印出来。

2）微调 M3E 模型

Uniem 提供了非常易用的 FineTune 接口，代码如下：

```
#第 5 章/M3EFineTune.py
#导入 load_dataset 函数和 FineTuner 类
from datasets import load_dataset
from uniem.finetuner import FineTuner
#加载名为'shibing624/nli_zh'的数据集，选择'STS-B'任务，并将缓存目录设置为'cache'
dataset = load_dataset('shibing624/nli_zh', 'STS-B', cache_dir='cache')
#从预训练的'moka-ai/m3e-small'模型创建一个 FineTuner 对象，使用上面加载的数据集
finetuner= FineTuner.from_pretrained('moka-ai/m3e-small', dataset=dataset)
#运行微调过程，将训练轮数设置为 3，将批大小设置为 64，将学习率设置为 3e-5
finetuned_model = finetuner.run(epochs=3, batch_size=64, lr=3e-5)
```

这段代码首先从 datasets 模块导入了 load_dataset 函数，并从 uniem.finetuner 模块导入了 FineTuner 类。接着，它加载了一个名为 shibing624/nli_zh 的数据集，选择了其中的 STS-B 任务，并将缓存目录设置为 cache，然后它从一个预训练的 moka-ai/m3e-small 模型创建了一个 FineTuner 对象，并指定了之前加载的数据集作为微调的数据。最后，它运行了微调过程，将训练轮数设置为 3，将批大小设置为 64，将学习率设置为 3e-5。

综上所述，M3E 模型以其强大的多模态融合能力、多层次表达、上下文敏感性及良好的可扩展性，为中文 NLP 领域带来了革新。

5.2.6 OpenAI 的 text-embedding 模型

Embedding 在 NLP 中扮演着至关重要的角色，它是一种将文本数据（如单词、短语或整个文档）转换成数值向量的技术。这些数值向量不仅能捕捉文本的表面意义，还能揭示其深层的语义特征，从而让计算机能够理解和处理人类语言。在没有嵌入技术之前，处理文本数据通常需要将每个单词映射到一个高维稀疏向量，这种表示方法不仅计算复杂度高，而且难以捕捉词汇间的语义关系，而嵌入技术则通过学习一个低维的密集向量空间，使相似意义的词汇在空间中距离较近，从而有效地降低了数据的维度，提升了计算效率，并且增强了模型的语义理解能力。下面介绍 OpenAI 发布的主要的 text-embedding-3 系列模型和 text-embedding-ada-002 模型，其中 text-embedding-3-small 是一个更新、更高效、性价比更高的选项，适合那些寻求高性能文本嵌入解决方案同时又关注成本控制的用户或项目，而 text-embedding-ada-002 作为之前的模型，可能在某些特定场景下仍有其应用价值，但整体而言，text-embedding-3-small 提供了更先进的功能和更好的性能指标。

text-embedding-3 系列模型和 text-embedding-ada-002 可以从多个方面来对比，下面进行详细说明。首先，从性能角度来看，text-embedding-3 系列模型相较于 text-embedding-

ada-002 有了显著的提升。这意味着用户在享受更高性能的同时，也获得了更低的使用成本。其次，text-embedding-3 系列模型还包括了一个更大规模的版本 text-embedding-3-large。这个版本拥有高达 3072 维的嵌入空间，是目前 OpenAI 表现最佳的模型，而 text-embedding-ada-002 的输出维度为 1536。最后，从技术细节上来看，虽然具体的模型设计和训练方法没有详细披露，但可以推断 text-embedding-3 系列模型采用了更先进的深度学习技术和更大的数据集进行训练，从而实现了性能上的飞跃。

综上所述，text-embedding-3 系列模型在性能、成本和规模上都超越了 text-embedding-ada-002，为用户带来了更高效、更经济且更强大的文本嵌入解决方案。接下来分别介绍 text-embedding-3 系列模型和 text-embedding-ada-002。

1. text-embedding-ada-002

text-embedding-ada-002 是 OpenAI 于 2022 年 12 月提供的一个 Embedding 模型，但调用接口需付费。

1) text-embedding-ada-002 的特点

其主要具有以下特点。

(1) 统一能力：OpenAI 通过将 5 个独立的模型(文本相似性、文本搜索-查询、文本搜索-文档、代码搜索-文本和代码搜索-代码)合并为一个新的模型。

在一系列不同的文本搜索、句子相似性和代码搜索基准中，这个单一的表述比以前的嵌入模型表现得更好。

(2) 上下文：上下文长度为 8192，使它在处理长文档时更加方便。

(3) 嵌入尺寸：只有 1536 个维度，是 davinci-001 嵌入尺寸的八分之一，使新的嵌入在处理向量数据库时更具成本效益。

2) text-embedding-ada-002 核心优势

text-embedding-ada-002 是 OpenAI 推出的一个强大的文本嵌入模型，属于大模型系列的一部分。该模型的核心优势如下。

(1) 深度语义理解：该模型不仅能够理解文本的字面意义，还能够深入地理解上下文和同义词等深层含义，生成的嵌入向量能充分地反映这些复杂的语义特征。

(2) 高维数据的有效压缩：它能够将复杂多变的文本信息转换为相对简洁的数值向量，在保留关键信息的同时，极大地简化了数据表示，便于进一步处理。

(3) 广泛的适用性：text-embedding-ada-002 适用于多种自然语言处理任务，包括但不限于文本分类、情感分析、机器翻译等，展现了其高度的通用性和灵活性。

(4) 深度学习驱动：该模型基于深度学习技术构建，经过大规模文本数据集的训练，能够高效地学习和表达语言模式。

(5) 数值向量表示：将文本转换成数值向量后，便于计算机进行高效计算，解决了机器处理文本的难题。

(6) 预训练模型的便利性：作为预训练模型，text-embedding-ada-002 经过大规模数据训练后用户可以直接应用，无须从零开始训练，显著地节省了时间和资源。

(7) 经济高效：相较于之前的模型，如“达芬奇”，text-embedding-ada-002 在价格上便宜了 99.8%，同时在性能上实现了超越，尤其在文本搜索、代码搜索、句子相似性任务上表现出色，并且在文本分类任务中也取得了与最强模型相当的性能。

总之，text-embedding-ada-002 模型凭借其深度语义理解能力、高效的高维数据压缩、多用途的适用性、基于深度学习的训练背景、数值向量的便捷处理、预训练模型的即时可用性及显著的成本效益，成为自然语言处理领域内的一个里程碑式的工具。

2. text-embedding-3 系列模型

2024 年 1 月 26 日，OpenAI 宣布推出两款全新的嵌入式模型，分别是小巧高效的 text-embedding-3-small 及庞大强大的 text-embedding-3-large。这两款模型的发布标志着 OpenAI 在嵌入式技术领域的又一次重大突破，为用户带来了前所未有的性能提升和价格优惠。首先，这款小巧高效的 text-embedding-3-small 模型与前代产品 text-embedding-ada-002 相比，text-embedding-3-small 在性能上有了显著的提升。在多语言检索(MIRACL)的常用基准测试中，其平均得分从 31.4%提高到了 44.0%，而在英语任务(MTEB)的常用基准测试中，平均得分也从 61.0%提高到了 62.3%。更令人惊喜的是，text-embedding-3-small 的价格仅为前代产品的 20%，每千 Token 的价格从 0.0001 美元降至 0.00002 美元，这无疑为广大用户带来了巨大的实惠。接下来，再来看这款更强大的 text-embedding-3-large 模型。作为 OpenAI 目前性能最强的模型，text-embedding-3-large 能够生成高达 3072 维的嵌入向量。在 MIRACL 上的平均得分从 31.4%提升至 54.9%，在 MTEB 上的平均得分从 61.0%提升至 64.6%，这一系列的数字足以证明其在性能上的卓越表现。当然，高性能也意味着更高的价格，text-embedding-3-large 的价格定为每千 Token 0.00013 美元，虽然比小型模型贵一些，但考虑到其强大的性能，这个价格还是非常具有竞争力的。

值得一提的是，这两款新嵌入模型都采用了特殊的训练技术，使开发者可以在嵌入的使用成本和性能之间做出权衡。具体来讲，开发者可以通过设定 dimensions API 参数来有效地缩短嵌入向量的长度，而这样做并不会影响嵌入向量代表概念的核心特性。这种灵活性使开发者可以根据实际需求选择合适的模型和参数，从而在保证性能的同时降低成本。总体来讲，OpenAI 推出的这两款新嵌入模型无疑为用户带来了更多的选择和可能性。无论是追求高性价比的小型模型，还是追求高性能的大型模型，用户都可以根据自己的需求和预算进行选择。同时，这两款模型的推出也为嵌入式技术的发展注入了新的活力。

上面探讨了各种文本向量模型，包括阿里巴巴发布的 GTE 向量模型、acge_text_embedding 模型、智源中英文语义向量模型 BGE、Moka 开源文本嵌入模型 M3E 及 OpenAI 的 text-embedding 模型。这些模型提供了一种将文本转换为数值向量的方法，使计算机能够理解和处理文本数据。然而，仅仅拥有这些向量模型还不足以构建一个完整的文本处理系统。为了有效地存储、检索和管理这些向量，需要一种专门的数据库技术，这就是接下来要介绍的向量数据库。

5.3 向量数据库

RAG是一种结合了检索和生成技术的先进自然语言处理方法。在这种技术中,向量数据库扮演着至关重要的角色。向量数据库是一种专门用于处理向量数据的数据库系统,它将数据转换成数学上的向量形式进行存储和处理。这使RAG能够通过向量数据库快速有效地检索和处理大量的向量数据,从而增强了模型的整体性能和应用范围。

在RAG技术中,向量数据库的作用主要体现在以下几个方面。

(1) 数据预处理和嵌入:向量数据库能够将原始的非结构化文本数据转换为结构化的向量形式(前面讲的文本向量模型已经详细探讨过),便于后续进行处理和分析。这个过程类似于将图书馆里的书籍按照主题和内容进行分类和索引,使检索变得更加高效。

(2) 检索和信息提取:向量数据库支持高效的相似度搜索,可以帮助RAG在海量数据中迅速定位到与查询相关的信息片段。这对于提高模型处理未知或少见信息的能力至关重要,同时也使模型能够更加准确地响应新的挑战。

(3) 生成和优化:向量数据库不仅用于检索,还可以辅助生成过程。通过检索到的相关信息,RAG能够生成更加准确和丰富的回应,从而提升了生成文本的质量。

(4) 多模态融合:向量数据库还能够处理多种类型的数据,如图像、音频等,并将其转换为向量形式,实现多模态信息的融合,进一步拓宽了RAG的应用场景。

综上所述,向量数据库在RAG系统中起到了至关重要的作用,它不仅提高了数据处理的效率和准确性,还为模型的生成能力提供了强有力的支撑。接下来将讲解主流的向量数据库。

5.3.1 Faiss

Faiss的全称是Facebook AI Similarity Search,是FaceBook的AI团队针对大规模相似度检索问题开发的一个工具,使用C++编写,有Python接口,对10亿量级的索引可以做到毫秒级检索。Faiss是免费且无须注册的,最早由Facebook AI Research团队在2017年发布,它在大规模语言模型的基础上发展而来,利用高效的索引结构和搜索算法,可以处理大规模数据集和高维向量。Faiss在发布后迅速受到了广泛的关注和应用,其高效的性能和优秀的扩展性使它成为许多大型公司和研究机构进行相似性搜索和向量检索的首选工具。随着时间的推移,Faiss不断进行更新和优化,增加了更多的功能和特性,为用户提供更好的体验。

1. 安装与环境配置

在开始使用Faiss之前,首先需要确保它已经被正确地安装在系统中。Faiss的安装过程非常简单。建议安装1.7.3及以上版本,因为新版本已经修复了旧版本中存在的诸多问题,安装过程也变得更加顺畅。可以通过pip命令轻松安装CPU版本的Faiss:

```
pip install faiss-cpu==1.7.3
```

如果硬件环境支持 GPU，并且需要利用 GPU 来加速检索过程，则可以安装 GPU 版本的 Faiss：

```
pip install faiss-gpu==1.7.3
```

新版本的 Faiss 还加强了对 Windows 操作系统的支持，这意味着无论是在 Linux、macOS 还是 Windows 环境下工作都可以无障碍地使用 Faiss。

2. Faiss 的基本使用方法

Faiss 的使用流程可以分为 3 个基本步骤。

(1) 准备向量库：需要准备一个向量库，这些向量将被用于后续的检索过程。向量库中的每个向量都应该具有相同的维度。

(2) 构建索引：使用 Faiss 提供的 API 构建一个索引。索引的类型可以根据具体的需求进行选择，例如暴力检索(Flat)、倒排索引(IVF)等。

(3) 检索相似向量：使用构建好的索引进行相似向量的检索。Faiss 会返回与查询向量最相似的 Top K 向量及其相似度得分。

以下是一个简单的示例代码，展示了如何使用 Faiss 进行相似向量的检索，代码如下：

```
#第 5 章/FaissSimpleUseDemo.py
#导入所需的库
import numpy as np
import faiss
#设置向量的维度和数量
d = 64                                                    #向量维度
nb = 100000                                               #索引向量库的数据量
nq = 10000                                                #待检索 query 的数目
#设置随机种子以确保结果可复现
np.random.seed(1234)
#生成索引向量库的向量
xb = np.random.random((nb, d)).astype('float32')  #索引向量库的向量
xb[:, 0] += np.arange(nb) / 1000.                         #在第一维上加上一个递增的偏移量
#生成待检索的 query 向量
xq = np.random.random((nq, d)).astype('float32')  #待检索的 query 向量
xq[:, 0] += np.arange(nq) / 1000.                         #在第一维上加上一个递增的偏移量
#创建一个 FlatL2 索引，使用 L2 范数(欧氏距离)作为相似度度量方法
index = faiss.IndexFlatL2(d)
#将索引向量库中的向量添加到索引中
index.add(xb)
#检索 Top K 相似 query，这里将 K 设为 4
k = 4
D, I = index.search(xq, k)                #返回每个 query 最相似的 Top K 索引列表及其距离
#打印前 5 个 query 的 Top K 索引和距离
print(I[:5])
print(D[-5:])
```

3. Faiss 常用索引类型分析

Faiss 有以下几种核心索引类型。

(1) Flat：暴力检索，提供最高的召回率，但速度慢、内存占用大，适用于向量量级较小的情况。

(2) IVFx Flat：倒排索引与暴力检索结合，利用 k-means 聚类减少搜索空间，提高检索速度，适用于百万级向量。

(3) PQx：乘积量化索引，将向量分割成多个子空间独立地进行检索，速度快、内存占用小，但牺牲了召回率，适合内存紧张且速度敏感的应用。

(4) IVFxPQy：结合倒排和乘积量化，平衡了速度、内存和召回率，被广泛地用于工业界。

(5) LSH：局部敏感哈希，这是一种特别适用于处理大规模数据集的检索技术。它通过一种巧妙的方式利用哈希技术来加速高维空间中相似向量的查找过程。与传统的哈希技术不同，局部敏感哈希并不试图避免哈希冲突，相反，它依赖于冲突来查找近邻。具体来讲，如果两个向量在高维空间中彼此接近，则通过精心设计的哈希函数对它们进行哈希处理后，它们被分配到同一个哈希桶的概率非常高。相反，如果两个向量相距较远，则它们被分配到相同哈希桶的概率就会非常低。LSH 的优势在于其训练过程非常迅速，支持分批导入数据，并且索引占用的内存非常小，检索速度也相对较快。这些特性使 LSH 非常适合于候选向量库非常庞大、内存资源受限的离线检索场景。尽管如此，LSH 也有其局限性，最主要的是召回率较低。这意味着在检索过程中可能会遗漏一些实际上与查询向量相似的向量，因此，在实际应用中，需要在检索速度、内存占用和召回率之间做出权衡。

(6) HNSWx：HNSWx 中的 x 为构建图时每个点最多连接多少个节点，x 越大，构图越复杂，查询越精确，当然构建 Index 时间也就越长，x 取 4～64 中的任何一个整数。HNSW 是基于图的高效近似最近邻搜索，检索速度快，召回率高，支持分批导入，但构建索引慢且内存消耗大，适用于对速度和召回率有极高要求，并且不介意长时间构建和大内存占用的场景。

4. Faiss 度量方法

度量方法是 Faiss 中用于计算向量间相似度的关键因素。Faiss 支持多达 8 种不同的度量方式，以适应更广泛的相似度计算需求。

(1) METRIC_INNER_PRODUCT：内积，用于计算两个向量的点积，通常用于计算余弦相似度。在使用内积度量时，通常需要先对向量进行归一化处理，以确保结果的准确性。

(2) METRIC_L1：曼哈顿距离，计算两个向量在各个维度上差值的绝对值之和，适用于对向量各维度的差异性敏感的场景。

(3) METRIC_L2：欧氏距离是最常用的度量方法，计算两个向量之间直线距离的平方。

(4) METRIC_Linf：无穷范数，计算两个向量在各个维度上差值的最大值，适用于对向量最大差异敏感的场景。

(5) METRIC_Lp：p 范数，是欧氏距离的一种泛化形式，允许调整距离计算的权重。

(6) METRIC_BrayCurtis：BC 相异度是基于两个向量差值的相对大小的度量方法。

(7) METRIC_Canberra：兰氏距离/堪培拉距离，类似于曼哈顿距离，但是对较小的数值变化更加敏感。

(8) METRIC_JensenShannon：JS 散度，是一种基于信息论的度量方法，用于计算两个概率分布的相似度。

在选择度量方法时，需要根据具体应用场景和数据特性来决定，例如，如果需要计算余弦相似度，则可选择内积度量，并确保向量已经被归一化处理。如果关心的是向量在各个维度上的绝对差异，则曼哈顿距离或兰氏距离可能是更好的选择，而如果数据具有概率分布的特性，则 JS 散度可能更适合。总之，Faiss 提供的多样化的度量方法为处理不同类型的相似度检索问题提供了丰富的工具，使 Faiss 成为一个非常灵活和强大的相似度检索框架。

5. Faiss 的核心原理

Faiss 是一个高效的向量数据库，它的核心功能是实现大规模的向量相似度搜索。在 Faiss 中，向量数据库的基础是由原始向量构成的集合，这些向量是数据库的基本单位。当执行搜索操作时，通常会输入一个查询向量 x，然后 Faiss 会返回与 x 最相似的 k 个向量。在这个过程中，索引(Index 对象)扮演着至关重要的角色。索引是对原始向量集进行预处理和封装的结构，它继承了一组向量库，并提供了一系列的操作，如训练(Train)和添加(Add)。通过训练和添加操作，可以建立一个索引对象并将其缓存在计算机内存中，以便进行快速搜索操作。在构建索引之前，需要明确向量的维度 d。对于大多数索引类型，还需要通过训练阶段来分析向量的分布，这一步骤对于提高搜索效率至关重要。只有当索引被成功建立后，才能进行后续的搜索操作。

训练的目的是生成原向量中心点，以及残差(向量中心点的差值)向量中心点，同时还会进行一部分预计算的距离计算。训练的过程大致如下：

(1) 将原始向量分成 m 个子空间，并为每个子空间训练中心点。如果每个子空间的中心点为 n，则 PQ(Product Quantization，乘积量化)可以表达 n^m 个中心点。

(2) 对于每个向量，找到它所属的子空间，并确定对应的中心点。

(3) 将向量减去对应的中心点，生成残差向量。

(4) 对残差向量生成二级量化器，进一步提高搜索的效率。

搜索操作是索引的重要组成部分，它涉及实际的相似度计算。在执行搜索操作时，Faiss 会返回两个矩阵，分别包含查询向量 $\boldsymbol{xq}$ 中元素与近邻的距离大小，以及近邻向量的索引序号。在使用 Faiss 进行查询向量的相似性搜索之前，需要将原始的向量集构建封装成一个索引文件，并将其缓存在内存中，以便提供实时的查询计算。在第 1 次构建索引文件时，需要经过训练和添加两个过程。如果后续有新的向量需要被添加到索引文件中，则可以再次执行添加操作，从而实现增量构建索引。总体来讲，Faiss 通过高效的索引结构和算法，实现了在大规模向量集中进行快速相似度搜索的能力，这对于许多机器学习、数据挖掘和信息检索等领域的应用来讲是非常有价值的。

6. 实践经验与技巧

在使用 Faiss 的过程中，可能会遇到一些问题和挑战。以下是一些常见问题、经验技巧。

(1) 索引的可分批导入：当向量数据量过大且无法一次性加载到内存中时，可以考虑使用支持分批导入的索引类型，如 HNSW、Flat 和 LSH。

(2) PCA 降维与分批添加：如果既想通过 PCA 降维来减少索引占用的内存，又想分批添加向量，则可以使用 sklearn 中的增量 PCA 方法。

(3) HNSW 的使用经验：HNSW 虽然在检索速度和召回率上有出色的表现，但其构建索引的过程可能非常耗时，并且占用的内存也相当大。

(4) 索引仅支持 float32 格式：Faiss 的所有索引仅支持浮点数为 float32 格式，不支持其他格式。

(5) 索引构建策略：推荐使用 faiss.index_factory 统一构建索引，简化参数配置，提高灵活性。

(6) 性能与权衡：根据实际需求(如向量量级、内存限制、检索速度要求等)选择合适的索引类型，强调综合考量各因素的重要性。

Faiss 作为一个高效的大规模相似度检索工具，其在检索增强的生成模型领域展现了巨大价值，尤其是与大模型结合时，极大地促进了知识密集型任务的表现。RAG 框架通过先检索相关的信息片段，再依据这些片段生成回答，而 Faiss 正是在这一过程中的加速器。具体而言，RAG 模型运作时，首先面临的是如何从庞大的知识库中快速地找到与用户查询最相关的片段。Faiss 凭借其高度优化的索引结构，能够对向量化的文档或信息块进行亚秒级检索，即使面对亿级数据量也能保持高效。利用如 HNSW 这样的高级索引方法，Faiss 在保持高召回率的同时，确保查询速度，这对于实时交互式应用至关重要。为了与 RAG 模型集成，通常会将知识库中的每个文档或关键信息点编码为高维向量，这一步骤可以通过预训练的语义嵌入模型完成。随后，Faiss 建立索引，将这些向量组织成易于搜索的数据结构。当用户进行查询时，查询语句经过相同嵌入模型转换为向量，Faiss 即刻在这海量的向量库中找出最相似的 Top K 向量，对应的知识片段随之被提取出来。这些片段作为额外的上下文，输入语言模型中，引导生成更准确、信息更丰富的回答。简言之，Faiss 为 RAG 模型提供了一个强大的“记忆库”，使模型能够在生成响应前即时访问最相关的外部信息，有效地提升了回答的质量和多样性，尤其在处理那些依赖丰富背景知识的任务上，如问答系统、对话系统或特定领域的文本生成，其效果尤为显著。这种结合不仅优化了资源利用，也为大模型的智能化和实用化开辟了新路径。

在企业数据量巨大的情况下，也就是在进行稠密向量相似度检索时，Faiss 并不太适合，它不支持分布式，不支持弹性伸缩，在这种情况下 Milvus 就特别适合，接下来详细讲解 Milvus。

5.3.2 Milvus

Milvus 是一款云原生向量数据库，具有高可用性、高性能和易扩展性，专门用于处理海量向量数据的实时检索。它基于 Faiss、Annoy、HNSW 等向量搜索库构建，旨在解决稠密向量相似度检索问题。Milvus 支持数据分区、分片、数据持久化、增量数据摄取、标量向量混合查询和 Time Travel 等功能，并对向量检索性能进行了大幅优化，满足各种向量检索场景的需求。建议用户使用 Kubernetes 部署 Milvus，以获得最佳的可用性和弹性。Milvus

采用共享存储架构，实现了存储计算的完全分离，计算节点支持横向扩展。从架构上看，Milvus 遵循数据流和控制流分离的原则，整体分为 4 个层次：接入层、协调服务、执行节点和存储服务，各层次相互独立，并且可以独立扩展和容灾。Milvus 向量数据库能够帮助用户轻松地应对海量非结构化数据（图片/视频/语音/文本）检索。单节点 Milvus 可以在秒级完成十亿级的向量搜索，分布式架构亦能满足用户的水平扩展需求。随着互联网的发展，非结构化数据变得越来越普遍，包括电子邮件、论文、物联网传感数据、社交媒体照片、蛋白质分子结构等。为了让计算机理解和处理非结构化数据，使用 Embedding 技术将这些数据转换为向量。Milvus 存储并索引这些向量。Milvus 能够通过计算两个向量的相似距离来分析它们之间的相关性。如果两个嵌入向量非常相似，则表示原始数据源也非常相似。Milvus 向量数据库专为向量查询与检索设计，能够为万亿级向量数据建立索引。与现有的主要用作处理结构化数据的关系数据库不同，Milvus 在底层设计上就是为了处理由各种非结构化数据转换而来的 Embedding 向量而生。

1. Milvus 基本概念

随着大模型时代的到来，非结构化数据的处理成为一项挑战性的任务。Milvus 作为一款开源的向量数据库，提供了一系列强大的功能来应对这一挑战。下面详细介绍 Milvus 的基本概念，帮助读者更好地理解和使用这个强大的工具。

1）非结构化数据

非结构化数据是指数据结构不规则，没有统一预定义数据模型，不便用数据库二维逻辑表来表现的数据。非结构化数据包括图片、视频、音频、自然语言等，占所有数据总量的 80%。非结构化数据处理可以通过各种人工智能或机器学习模型转换为向量数据后进行处理。

2）特征向量

向量又称为 Embedding Vector，是指由 Embedding 技术从离散变量（如图片、视频、音频、自然语言等各种非结构化数据）转变而来的连续向量。在数学表示上，向量是一个由浮点数或者二值型数据组成的 n 维数组。通过现代的向量转化技术，可以将非结构化数据抽象为 n 维特征向量空间的向量，采用最近邻算法计算非结构化数据之间的相似度。

3）Collection

Collection 包含一组 Entity，可以等价于关系数据库中的表。

4）Entity

Entity 包含一组 Field。Field 与实际对象相对应。Field 既可以是代表对象属性的结构化数据，也可以是代表对象特征的向量。Primary Key 是用于指代一个 Field 的唯一值。可以自定义 Primary Key，否则 Milvus 将会自动生成 Primary Key。需要注意，目前 Milvus 不支持 Primary Key 去重，因此在一个 Collection 内可能出现 Primary Key 相同的 Entity。

5）Field

Field 是 Entity 的组成部分。Field 既可以是结构化数据，例如数字和字符串，也可以是向量。Milvus 2.0 已支持标量字段过滤，并且 Milvus 2.0 在一个集合中只支持一个主键

字段。

6) Partition

分区(Partition)是集合(Collection)的一个分区。Milvus 支持将收集数据划分为物理存储上的多部分,这个过程称为分区,每个分区可以包含多个段。

7) Segment

Milvus 在数据插入时,通过合并数据自动创建数据文件。一个 Collection 可以包含多个 Segment。一个 Segment 可以包含多个 Entity。在搜索中,Milvus 会搜索每个 Segment,并返回合并后的结果。

8) Sharding

Sharding 是指将数据写入操作分散到不同节点上,使 Milvus 能充分利用集群的并行计算能力进行写入。在默认情况下,单个 Collection 包含两个分片(Shard)。目前 Milvus 采用基于主键哈希的分片方式,未来将支持随机分片、自定义分片等更加灵活的分片方式。分区的意义在于通过划定分区减少数据读取,而分片的意义在于在多台机器上并行写入操作。

9) 索引

索引基于原始数据构建,可以提高对 Collection 数据搜索的速度。Milvus 支持多种索引类型。为了提高查询性能,可以为每个向量字段指定一种索引类型。目前,一个向量字段仅支持一种索引类型。切换索引类型时,Milvus 会自动删除之前的索引。相似性搜索引擎的工作原理是将输入的对象与数据库中的对象进行比较,找出与输入最相似的对象。索引是有效组织数据的过程,极大地加速了对大型数据集的查询,在相似性搜索的实现中起着重要作用。对一个大规模向量数据集创建索引后,查询可以被路由到最有可能包含与输入查询相似的向量的集群或数据子集。在实践中,这意味着要牺牲一定程度的准确性来加快对真正的大规模向量数据集的查询。

10) PChannel

PChannel 表示物理信道。每个 PChannel 对应一个日志存储主题。在默认情况下,将分配一组 256 个 PChannel 来存储记录 Milvus 集群启动时数据插入、删除和更新的日志。

11) VChannel

VChannel 表示逻辑通道。每个集合将分配一组 VChannel,用于记录数据的插入、删除和更新。VChannel 在逻辑上是分开的,但在物理上共享资源。

2. 核心特性与优势

Milvus 是一款高性能、高可用、高可靠的国产向量数据库,它采用了独特的数据结构和算法,实现了高效的向量运算和查询,主要有以下特性与优势。

(1) 高性能:Milvus 采用了独特的数据结构和算法,可以实现高效的向量运算和查询,其性能指标在很多情况下优于其他国产向量数据库。

(2) 高可用、高可靠:Milvus 支持在云上扩展,其容灾能力能够保证服务高可用。

(3) 混合查询:Milvus 支持在向量相似度检索过程中进行标量字段过滤,从而实现混合查询。

(4) 开发者友好：支持多语言、多工具的 Milvus 生态系统。

(5) 易用性：Milvus 具有简单的 API 和易于使用的管理工具，用户可以快速上手并进行大规模的向量数据处理和分析。

(6) 兼容性：Milvus 支持多种数据格式和协议，如 JSON、XML、HTTP 等，可以方便地与其他系统和平台进行集成和数据交换。

3. Milvus 系统架构

Milvus 2.0 是一款云原生向量数据库，采用存储与计算分离的架构设计，所有组件均为无状态组件，极大地增强了系统弹性和灵活性。整个系统分为 4 个层次。

1) 接入层

接入层(Access Layer，AL)是系统的门面，由一组无状态 Proxy 组成，对外提供用户连接的 Endpoint。接入层负责验证客户端请求并减少返回结果。Proxy 本身是无状态的，一般通过负载均衡组件(如 Nginx、Kubernetes Ingress、NodePort、LVS)对外提供统一的访问地址并提供服务。由于 Milvus 采用大规模并行处理(Massively Parallel Processing，MPP)架构，所以 Proxy 会先对执行节点返回的中间结果进行全局聚合和后处理，再返回至客户端。

2) 协调服务

协调服务(Coordinator Service，CS)是系统的大脑，负责向执行节点分配任务。它承担的任务包括集群拓扑节点管理、负载均衡、时间戳生成、数据声明和数据管理等。协调服务共有 4 种角色。

(1) 根协调器：根协调器(Root Coordinator，RC)负责处理数据定义语言(Data Definition Language，DDL)和数据控制语言(Data Control Language，DCL)请求，例如，创建或删除 Collection、Partition、Index 等，同时负责维护中心授时服务(TSO)和时间窗口的推进。

(2) 查询协调器：查询协调器(Query Coordinator，QC)负责管理 Query Node 的拓扑结构和负载均衡及从增长的 Segment 移交切换到密封的 Segment。

(3) 数据协调器：数据协调器(Data Coordinator，DC)负责管理 Data Node 的拓扑结构，维护数据的元信息及触发 Flush、Compact 等后台数据操作。

(4) 索引协调器：索引协调器(Index Coordinator，IC)负责管理 Index Node 的拓扑结构，构建索引和维护索引元信息。

3) 执行节点

执行节点是系统的四肢，负责完成协调服务下发的指令和 Proxy 发起的数据操作语言(Data Manipulation Language，DML)命令。由于采取了存储计算分离，所以执行节点是无状态的，可以配合 Kubernetes 快速实现扩缩容和故障恢复。执行节点分为以下 3 种角色。

(1) Query Node：Query Node 通过订阅消息存储获取增量日志数据并转换为 Growing Segment，基于对象存储加载历史数据，提供标量＋向量的混合查询和搜索功能。

(2) Data Node：Data Node 通过订阅消息存储获取增量日志数据，处理更改请求，并将日志数据打包存储在对象存储上，从而实现日志快照持久化。

(3) Index Node：Index Node 负责执行索引构建任务。Index Node 不需要常驻于内存，可以通过 Serverless 的模式实现。

4) 存储服务

存储服务是系统的骨骼，负责 Milvus 数据的持久化，分为元数据存储(Meta Storage，MS)、消息存储(Log Storage，LS)和对象存储(Object Storage，OS)三部分。

(1) 元数据存储：负责存储元信息的快照，例如集合 Schema 信息、节点状态信息、消息消费的 Checkpoint 等。元数据存储需要极高的可用性、强一致和事务支持，因此，ETCD 是这个场景下的不二选择。除此之外，ETCD 还承担了服务注册和健康检查的职责。ETCD 是一个开源的分布式键值对存储系统，主要用于共享配置和服务发现。

(2) 日志存储：日志存储是一套支持回放的发布订阅系统，用于持久化流式写入的数据，以及可靠的异步执行查询、事件通知和结果返回。当执行节点宕机恢复时，通过回放日志存储保证增量数据的完整性。

(3) 对象存储：负责存储日志的快照文件、标量/向量索引文件及查询的中间处理结果。Milvus 采用 MinIO 作为对象存储，另外也支持部署于 AWS S3 和 Azure Blob 这两大最广泛使用的低成本存储，但是，由于对象存储访问延迟较高，并且需要按照查询计费，因此 Milvus 未来计划支持基于内存或 SSD 的缓存池，通过冷热分离的方式提升性能以降低成本。目前，分布式版 Milvus 依赖 Pulsar 作为消息存储，单机版 Milvus 依赖 RocksDB 作为消息存储。日志存储也可以替换为 Kafka、Pravega 等流式存储。整个 Milvus 以日志为核心来设计，遵循日志即数据的准则，因此在 2.0 版本中没有维护物理上的表，而是通过日志持久化和日志快照来保证数据的可靠性。日志系统作为系统的主干，承担了数据持久化和解耦的作用。通过日志的发布订阅机制，Milvus 将系统的读、写组件解耦。

4. Milvus 部署模式

Milvus 支持两种部署模式，即单机模式(Standalone)和分布式模式(Cluster)。两种模式具备完全相同的能力，用户可以根据数据规模、访问量等因素选择适合自己的模式。Standalone 模式部署的 Milvus 暂时不支持在线升级为 Cluster 模式。

1) 单机版 Milvus

单机版 Milvus 是一个专为人工智能应用设计的开源向量数据库，它由 3 个关键的组件构成。这些组件协同工作，提供了强大的数据处理能力，特别是在处理大量向量数据时表现出色。

(1) Milvus：作为系统的核心，它负责执行向量数据的插入、查询、更新和删除操作。Milvus 支持多种数据类型和索引结构，使它在处理大规模数据集时既高效又灵活。

(2) ETCD：这是一个高性能的键值存储系统，充当元数据引擎的角色。它主要用于管理和同步 Milvus 内部各个组件的状态信息，如代理节点(Proxy)、索引节点(Index Node)等。通过 ETCD，Milvus 能够保证集群内各部分之间的一致性和可靠性。

(3) MinIO：作为存储引擎，MinIO 负责维护 Milvus 数据的持久化。这意味着所有在 Milvus 中存储的数据都会通过 MinIO 进行备份，从而确保数据的安全性和可恢复性。

MinIO 支持多种存储后端，包括本地文件系统和云存储服务，为用户提供了极大的灵活性。

2）分布式版 Milvus

分布式版 Milvus 由 8 个微服务组件和 3 个第三方依赖组成，每个微服务组件可使用 Kubernetes 独立部署。

（1）微服务组件：包括 Root Coordinator、Proxy、Query Coordinator、Query Node、Index Coordinator、Index Node、Data Coordinator、Data Node。

（2）第三方依赖：ETCD 负责存储集群中各组件的元数据信息，MinIO 负责处理集群中大型文件的数据持久化，如索引文件和全二进制日志文件，Pulsar 负责管理近期更改操作的日志，输出流式日志及提供日志订阅服务。

5. Milvus 微服务化及实现特点

Milvus 微服务化架构实现了高度模块化和职责分离，通过 ETCD 管理元数据，利用对象存储保存向量数据，并通过 Proxy 提供统一访问接口。这种设计不仅支持云原生部署，还确保读写和索引操作的进程级隔离，同时允许用户根据业务需求调整一致性级别以优化性能。

1）微服务化

Milvus 将服务拆成多个角色，每个角色的职责划分相对独立，其中 Index Node、Query Node、Data Node 这些角色是实际工作的 Woker 节点，Index Coordinator、Query Coordinator、Data Coordinator 是负责协调 Woker 节点及将任务协调并分派给其他角色的节点，ETCD 负责存储元数据，对象存储负责存储向量数据，Proxy 负责 Milvus 的统一访问，Data Node、Data Coordinator 负责向量的写入，Index Node、Index Coordinator 负责向量索引的构建，Query Node、Query Coordinator 负责向量的查询，Root Coordinator 负责处理 DDL 去协调其他 Coordinator 及全局时间分发并维护当前元数据快照。

2）支持云原生

Milvus 服务本身是没有状态的，数据存储在对象存储，元数据会存放在 ETCD。原生支持 Kubernetes 集群部署，可以根据集群或者个别角色的负载去动态地扩缩资源。

3）向量操作【读/写/建索引】之间进程级别隔离

向量读/写/建索引都是通过不同的节点完成的，这样操作之间都是通过不同进程进行隔离的，既不会抢占资源，也不会相互影响。

4）在查询时指定不同的一致性级别

在真实的业务场景中，一致性要求越强，查询对应的响应时间也会越长。用户可以根据自己的需求选择不同的一致性级别。

6. Milvus 向量执行引擎

Knowhere 是 Milvus 的核心向量执行引擎，它集成了几个向量相似度搜索库，包括 Faiss、HNSWlib 和 Annoy。Knowhere 的定义范畴分为狭义和广义两种：

（1）狭义上的 Knowhere 是下层向量查询库（如 Faiss、HNSW、Annoy）和上层服务调度之间的操作接口。同时，异构计算也由 Knowhere 这一层来控制，用于管理索引的构建和查

询操作在何种硬件上执行，如 CPU 或 GPU，未来还可以支持 DPU/TPU/……这也是 Knowhere 这一命名的源起——Know Where。

(2) 广义上的 Knowhere 还包括 Faiss 及其他所有第三方索引库，因此，可以将 Knowhere 理解为 Milvus 的核心运算引擎。

从上述定义可以得知，Knowhere 只负责处理与数据运算相关的任务，其他系统层面的任务(如数据分片、负载均衡、灾备等)都不在它的功能范畴中。另外，从 Milvus 2.0.1 开始，广义的 Knowhere 已从 Milvus 项目中剥离出来，成为一个单独的项目。

7. Milvus 读写及查询内部机制

Milvus 的读写及查询内部机制都体现了其存算分离的设计理念。

1) 读写过程的内部机制

在 Milvus 的存算分离架构中，读写流程的设计充分体现了其高效性和灵活性。具体来讲，整个过程可以分为以下几个步骤：首先，Proxy 将消息写入消息队列中，这里的消息队列可以是 Kafka、Pulsar 或 RocksMQ 等。这一步骤的目的是解耦生产者和消费者，提高系统的吞吐量和可用性。接着，Data Node 从消息队列中消费消息，生成对应的数据分片，即 Segment，并将其上传到 Object Storage 上。这里的 Object Storage 是一种分布式存储系统，可以提供高可用性和可扩展性，然后协调节点会收集生成的 Segment，并指挥 Query Node 加载对应的 Segment 以供查询。这一步骤的目的是实现数据的实时更新和查询。在整个过程中，生成存储文件的过程发生在 Data Node 上，而使用存储数据进行查询计算则发生在 Query Node 上。这两个过程互不干扰，可以分别进行弹性扩展，例如，在查询计算密集的时段，可以扩展 Query Node 的数量和资源；在写入压力较大时，可以扩展 Data Node 和资源。

2) 查询过程的内部机制

Milvus 的查询过程同样体现了其存算分离的设计理念。具体来讲，这个过程可以分为以下几个步骤：首先，Proxy 向 Query Coordinator 询问 Milvus Delegator 的分布，得知该 Collection 的 Delegator 在 Query Node 2 上。这里的 Delegator 是查询的首领，负责汇总其所统领的 Segment 的查询结果。接着，Proxy 向 Query Node 2 发送查询请求。这一步骤的目的是将查询请求发送到正确的节点上，以便快速获得查询结果，然后当 Delegator 收到请求后，将其转发给 Query Node 1 和 Query Node 3，获取所有 Segment 的查询结果。这一步骤的目的是并行处理查询请求，提高查询效率。最后，Delegator 汇总所有查询结果，返给 Proxy。这一步骤的目的是将查询结果整合在一起，提供给客户端。总体来讲，Milvus 的查询过程充分利用了其存算分离的架构优势，实现了高效、灵活的查询功能。

8. Milvus 提供的工具

Milvus 提供了多种工具来增强用户体验和数据管理，其中包括 Milvus CLI、Milvus Backup、MilvusDM 和 Milvus Attu，它们各自针对不同的需求，提供了从命令行操作到数据迁移再到图形界面的全方位解决方案。

1) Milvus CLI

Milvus 命令行工具(Command-Line Interface，CLI)是一个便捷的数据库客户端命令行

工具，基于 Python SDK，它允许使用交互式命令行提示符通过终端执行命令，支持数据库连接、数据导入和导出及向量间距离计算等多种功能。这款工具可以在 Windows、macOS、Linux 全平台上使用，无须依赖外部包，无论是在线还是离线环境都可以轻松地安装和使用。Milvus CLI 的使用非常简便，用户可以通过交互式命令行提示符在终端执行各种命令。它的安装过程也非常简单，用户只需运行一条 pip 安装命令 pip install milvus-cl 便可完成在线安装，而且它对 Python 版本的要求是 3.8 以上。此外，Milvus CLI 还具备许多实用的特性，例如内置的帮助文档、自动补全功能等，这些都大大地提升了用户的工作效率。总体来讲，Milvus CLI 是一款功能全面、易于使用且跨平台的工具，非常适合那些需要频繁操作 Milvus 向量数据库的用户。Milvus CLI 安装后使用起来非常方便，类似于使用 MySQL 客户端命令行一样，很容易上手操作。常用命令如下。

（1）以默认方式连接数据库：进入 milvus_cli 客户端，输入 connect 即可按默认 IP 地址和端口连接 Milvus 数据库，命令如下：

```
> connect
```

（2）指定 IP 地址和端口号连接数据库，命令如下：

```
> connect -h ip -p 19530
```

（3）创建 collection，命令如下：

```
create collection -c car -f id:INT64:primary_field -f vector:FLOAT_VECTOR:128 -f
color:INT64:color -f brand:INT64:brand -p id -a -d 'car_collection'
```

（4）查看 collection，命令如下：

```
list collections
```

（5）查询表详细信息，命令如下：

```
describe collection -c car
```

（6）删除 collection，命令如下：

```
delete collection -c car
```

2）Milvus Backup

Milvus Backup 是 Zilliz 公司专门为 Milvus 开发的数据备份和恢复工具。它同时提供 CLI 和 API，以适应不同的应用场景。Milvus Backup 具有以下能力：

（1）支持包括命令行和 RESTful API 的多种交互方式。

（2）支持热备份，对 Milvus 集群运行几乎没有影响。

（3）支持集群全量备份或指定 collection 备份。

（4）通过 bulkinsert 实现备份恢复，并可在恢复时重命名 collection。

(5) 支持 S3,支持跨 bucket 备份,可以实现集群间迁移。

(6) 支持 Milvus 2.2.0+版本。

目前只支持备份原数据,不支持备份索引数据。Milvus Backup 编程语言使用 Go 语言实现,部分逻辑直接复制 Milvus 源码,最大限度地保证兼容性和操作习惯一致性。

Milvus Backup 交互方式支持多种使用方式。

(1) 命令行:考虑备份工具的主要用户是对 Milvus 进行运维的开发人员,命令行是最简便直接的使用方式,Milvus Backup 基于 Cobra 框架实现了命令行。

(2) RESTful API:Milvus Backup 提供 RESTful API,便于工具服务化,并提供 Swagger UI。

(3) Go Module:Milvus Backup 还可以作为 Go Module 集成到其他工具之中。

3) MilvusDM

MilvusDM 是一款专门用于 Milvus 数据的导入和导出的开源数据迁移工具。为了大幅提高数据管理效率并降低 DevOps 成本,MilvusDM 支持以下迁移通道。

(1) Milvus 到 Milvus:迁移 Milvus 实例之间的数据。

(2) Faiss 到 Milvus:将未压缩的数据从 Faiss 导入 Milvus。

(3) HDF5 到 Milvus:将 HDF5 文件导入 Milvus。

(4) Milvus 到 HDF5:将 Milvus 中的数据保存为 HDF5 文件。

MilvusDM 托管在 GitHub 上,如果要安装 MilvusDM,则可以运行 pip install pymilvusdm。

4) Milvus Attu

Milvus Attu 是 Milvus 的一个高效的开源管理工具。它具有直观的图形用户界面,允许用户轻松地与数据库进行交互。只需单击几下,就可以可视化集群状态、管理元数据、执行数据查询操作等,接下来展示几个 Attu 管理界面的截图。

(1) 创建集合:在浏览器中访问 Attu 管理系统的地址为 http://{your machine IP}:8000,输入账号和密码后单击 Connect 按钮进入 Attu 管理 Web 平台。单击左侧导航面板上的集合图标,然后单击创建集合,当出现创建集合对话框时输入必要的信息,本示例创建一个名为 test 的集合,其中包含一个主键字段、一个向量字段和一个标量字段,可以根据需要添加标量字段。创建集合的界面如图 5-4 所示。

(2) 导入数据:导入数据会追加数据而不是覆盖数据。在集合页面上,单击导入数据后会显示导入数据对话框,在集合下拉列表中选择要导入数据的集合,在分区下拉列表中选择要导入数据的分区,选择 CSV 文件,列名与模式中指定的字段名相同。导入数据的界面如图 5-5 所示。

如果导入成功,则实体计数列中的行计数状态会更新到集合中。在相应的分区选项卡页面上,导入数据的分区中的实体计数列中的行计数状态会更新。实体计数的更新可能需要一些时间。

(3) 创建索引:以下示例使用欧氏距离作为相似性度量标准,构建了一个具有 1024 个 nlist 值的 IVF_FLAT 索引。在 Collection 页面上单击 Schema 按钮,切换到 Schema 选项

图 5-4　创建集合的界面

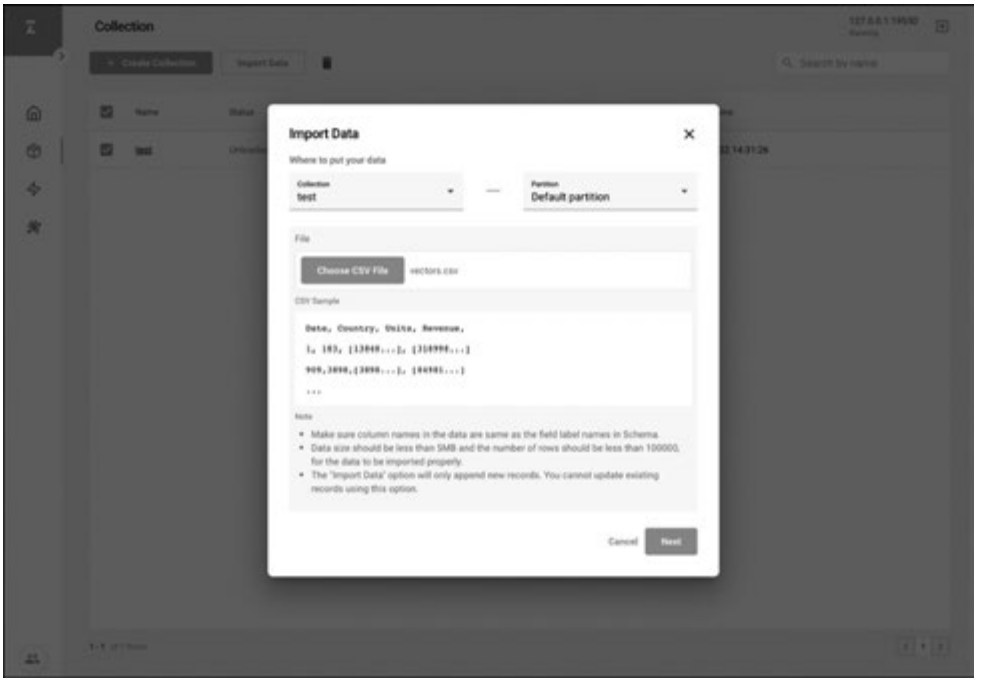

图 5-5　导入数据的界面

卡页面，然后单击 CREATE INDEX 按钮，此时会弹出 Create Index 对话框。在 Create Index 对话框中，从 Index Type 下拉列表中选择 IVF_FLAT，从 Metric Type 下拉列表中选择 L2，并在 nlist 字段中输入 1024。打开 View Code，跳转到 Code View 页面，可以选择以 Python 或 Node.js 查看代码。创建索引的界面如图 5-6 所示。

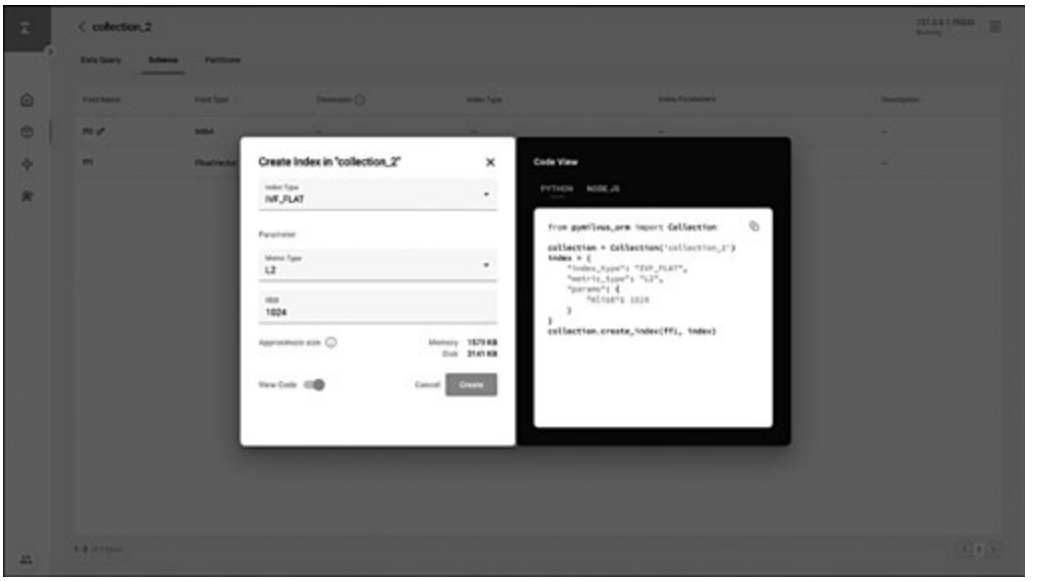

图 5-6　创建索引的界面

单击 Create 按钮来创建索引，如果创建成功，则创建的索引类型将出现在向量字段的 Index Type 列中。

(4) 查询数据：单击要在其中查询数据的集合条目，打开相应的详细页面。在 Data Query 选项卡页面上，单击 Filter 图标，此时会弹出 Advanced Filter 对话框。使用 Field Name 下拉列表、Logic 下拉列表、Value 字段和 AND 运算符指定复杂的查询条件，例如 color＞10 && color＜20，然后单击 Apply Filter 按钮，以应用查询条件。单击 Query 按钮以检索与查询条件匹配的所有查询结果。查询数据的界面如图 5-7 所示。

除了基本的查询，还支持向量相似性搜索及带有时间旅行的搜索的混合搜索。

(5) 系统视图：Attu 具有监控 Milvus 系统的功能。在左侧导航栏中单击系统视图图标，进入系统视图页面。系统视图仪表板会显示 Milvus 实例的拓扑结构。单击 Milvus 节点或协调器节点，相应的信息将显示在右侧的信息栏中。在节点列表视图中，所有由其父协调节点管理的节点都列在表格中，可以按照包括 CPU 内核数、CPU 内核使用率、磁盘使用率和内存使用率在内的指标对节点进行排序。系统视图的界面如图 5-8 所示。

9. Milvus 应用场景

Milvus 在应用中添加相似性搜索很容易，应用场景包括但不限于以下几种。

(1) 图像相似性搜索：使图像可搜索，并即时返回来自大型数据库中最相似的图像。

(2) 视频相似性搜索：首先将关键帧转换为向量，然后将结果输入 Milvus，可以在几乎实时的时间内搜索和推荐数十亿个视频。

(3) 音频相似性搜索：快速查询大量音频数据，如语音、音乐、音效和表面相似声音。

(4) 分子相似性搜索：针对指定分子进行极快相似性搜索、子结构搜索或超结构搜索。

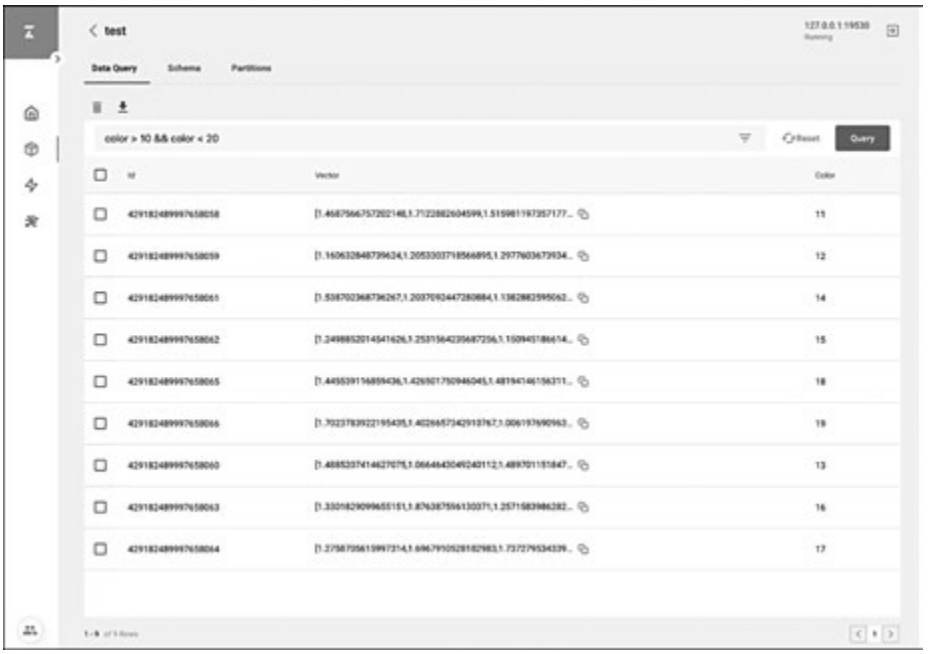

图 5-7　查询数据的界面

图 5-8　系统视图的界面

(5) 推荐系统：根据用户行为和需求推荐信息或产品。

(6) 问答系统：交互式数字问答聊天机器人，自动回答用户的问题。

(7) DNA 序列分类：通过比较相似的 DNA 序列，在毫秒级别准确地分类一个基因。

(8) 文本搜索引擎：通过对关键字与文本数据库进行比较，找到相关信息。

接下来，将探讨另一种向量数据库 Pinecone，它在某些方面与 Milvus 有所不同，但在向量搜索和数据管理方面也提供了强大的功能。

5.3.3 Pinecone

Pinecone 是一个托管式的向量数据库服务，它专注于为机器学习模型和应用程序提供快速、可扩展的向量搜索能力。与关系数据库或键值存储不同，Pinecone 用来高效地处理高维稀疏向量，这些向量常用于表示文本、图像、音频等复杂数据的特征空间。通过优化的索引结构和查询算法，Pinecone 能够实现亚毫秒级的相似性搜索，这对于需要实时反馈的应用场景至关重要。Pinecone 是商业项目，不是开源免费的，可以访问官网 https://www.pinecone.io/注册账号后使用。Pinecone 的核心在于其高效的向量索引机制。它首先接收用户上传的向量数据，并构建一个索引结构，这个结构被优化以支持快速的相似性搜索。当接收到查询向量时，Pinecone 利用空间搜索算法快速定位到最相似的几个向量，而不是逐一比较所有数据，这大大地提升了搜索效率。

1. 核心特性

Pinecone 的核心特性如下。

(1) 高性能搜索：Pinecone 利用先进的索引技术，如近似最近邻搜索算法，能在海量向量集中快速找到最相似的向量，从而支持实时的推荐和检索需求。

(2) 简单易用的 API：无论是通过 Python 客户端库还是 RESTful API，Pinecone 都提供了简洁直观的接口，让开发者能够轻松地将向量搜索功能集成到现有应用中，例如，只需几行代码便可创建索引、插入向量数据，并执行相似性查询。

(3) 自动缩放：随着数据量的增长和查询需求的变化，Pinecone 能够自动调整资源，确保查询性能始终保持高效稳定，无须用户手动配置服务器或管理基础设施。

(4) 多环境部署：支持多种部署环境，包括公有云和私有云部署，确保数据安全性和合规性，满足不同企业的需求。

2. 代码实践

下面通过安装 Pinecone Python 客户端，使用 Python 代码来操作 Pinecone。通过 pip3 install pinecone-client 命令来安装 Pinecone Python 客户端。接下来通过 Python 代码展示如何使用 Pinecone API 及与 OpenAI API 集成，实现向量数据的存储、查询、更新及删除操作，特别是在构建一个基于向量的 AI 知识库场景中的应用，代码如下：

```
#第 5 章/PineconeDemo.py
import pinecone
from openai import OpenAI
#初始化 Pinecone 客户端，使用用户的 API 密钥和环境信息
pinecone.init(
    api_key='你的 Pinecone API 密钥',          #替换为实际的 Pinecone API 密钥
    environment='你的 Pinecone 环境名称'       #替换为实际的 Pinecone 环境名称
)
```

```
#创建一个新的索引,这里将向量维度设置为 1536
pinecone.create_index('ai_knowledge', dimension=1536)
#列出所有现有的索引
print(pinecone.list_indexes())          #应该能看到刚创建的 'ai_knowledge'
#描述指定索引的信息
print(pinecone.describe_index('ai_knowledge'))
#删除索引(演示目的,在实际使用时需要小心)
#pinecone.delete_index('ai_knowledge')
#获取索引实例,用于后续操作
index = pinecone.Index('ai_knowledge')
#获取索引的统计信息
print(index.describe_index_stats())
#定义文本向量化函数,使用 OpenAI 的嵌入模型
def embedding(text):
    response = client.embeddings.create(
        model='text-embedding-ada-002',
        input=[text]
    )
    return response.data[0].embedding
#准备知识库数据,包含 AI 相关的问题与答案
knowledge_base = [
    {"system": "ai_knowledge", "question": "什么是机器学习?", "answer": "机器学习
是一门计算机科学,使系统无须明确编程就能从数据中学习并做出预测或决策。"},
    {"system": "ai_knowledge", "question": "深度学习和机器学习的区别是什么?",
"answer": "深度学习是机器学习的一个子集,专注于使用深度神经网络解决复杂模式识别和学习
问题。"},
]
#向索引中添加或更新向量数据
index.upsert(
    vectors=[
        ('ml_q1', embedding(knowledge_base[0]['question']), knowledge_base[0]),
        ('dl_q1', embedding(knowledge_base[1]['question']), knowledge_base[1]),
    ]
)
#执行相似度搜索
question = '机器学习的基本概念是什么? '
response = index.query(
    top_k=1,
    include_values=False,
    include_metadata=True,
    vector=embedding(question)
)
print(response)
#混合搜索,结合向量查询和元数据过滤
question = '深度学习的特点是什么? '
response = index.query(
    top_k=1,
    include_values=False,
    include_metadata=True,
```

```
        vector=embedding(question),
        filter={'system': 'ai_knowledge'}
    )
    print(response)
    #示例删除数据流程(先获取再删除)
    #result = index.fetch(ids=['ml_q1'])
    #print(result['vectors']['ml_q1']['metadata'])
    #index.delete(ids=['ml_q1'])                          #实际删除操作
    #主函数,整合所有操作
    def main():
        global client                                     #将 client 声明为全局变量
        client = OpenAI(api_key='你的 OpenAI API 密钥')   #初始化 OpenAI 客户端
        #示例查询
        question = '简述人工智能的发展历程.'
        response = index.query(
            top_k=3,
            include_values=False,
            include_metadata=True,
            vector=embedding(question)
        )
        #构建基于知识库的用户提示
        knowledge_summary = '\n'.join([f"问题: {item.metadata['question']} 回答:
    {item.metadata['answer']}" for item in response.matches])
        user_prompt = f"""
        可参考以下知识库内容回答问题:
        {knowledge_summary}
        用户问题: {question}
        """
        print(user_prompt)
        #使用 GPT-4 完成知识问答
        system_prompt = "你是一个 AI 助手,利用提供的知识库信息来准确回答问题。"
        chat_result = chat_completion(system_prompt, user_prompt)
        print("AI 助手的回答:", chat_result.choices[0].message.content)
    #运行主函数
    if __name__ == "__main__":
        main()
```

在实际运行此代码前,需要确保替换所有占位符,例如“你的 Pinecone API 密钥”“你的 OpenAI API 密钥”“你的 Pinecone 环境名称”为实际的认证信息。

5.3.4 Chroma

Chroma 是一个开源的向量数据库,旨在简化大模型应用的构建过程。它的核心功能在于存储文档数据和元数据,以及提供文档和查询的嵌入服务,同时支持高效的向量搜索。Chroma 的设计理念是保持足够的简洁性,以提高开发者的效率,并且在搜索的基础上进行数据分析,追求快速响应的性能。Chroma 允许用户通过 Python 或 JavaScript 代码生成词嵌入,并提供了一个简单的 API,便于开发者在内存中或在客户端/服务器模式下运行的后

端数据库上进行操作。这使开发人员可以先在 Jupyter Notebook 中使用 API 进行原型设计，然后将相同的代码应用于生产环境，后者可以在客户端/服务器模式下运行数据库。在内存中运行时，Chroma 数据库集合可以保存为 Apache Parquet 格式的磁盘文件，这样就能降低生成词嵌入的成本和性能开销。Chroma 作为一个流行的开源项目，它的目标是帮助用户更便捷地将现实世界中的文档（如知识、事实和技能等）整合进大模型中。这有助于提高构建大模型应用的效率和灵活性，让开发者能够更轻松地进行知识集成和应用开发。Chroma 支持用于搜索、过滤等的丰富功能，并能与多种平台和工具（如 LangChain、LLaMAIndex、OpenAI 等）集成。接下来介绍如何用 Python 安装和使用 Chroma，以及一些常见的使用场景和数据结构。如果要安装 Chroma，则需要先安装 Python 3.6 或更高版本，然后用 pip install chromadb 命令安装 chromadb 包，下面用 Python 代码示例演示如何使用 Chroma。

1. Chroma 客户端对象操作

安装完成后就可以在 Python 语言中导入 chromadb 模块，并创建一个 Chroma 客户端对象，命令如下：

```
import chromadb
chroma_client = chromadb.Client()
```

如果用的是服务器上的数据库，则命令如下：

```
import chromadb
chroma_client = chromadb.HttpClient(host="localhost", port=8000)
```

如果要保存在本地，则命令如下：

```
import chromadb
client = chromadb.PersistentClient(path="/path/to/data")
```

2. 创建和管理集合

集合（Collection）是 ChromaDB 中存储嵌入、文档、元数据的地方，类似于关系数据库中的表（Table）。用客户端对象的 create_collection 方法创建一个集合，并且指定一个名称，命令如下：

```
collection = chroma_client.create_collection(name="my_collection")
```

也可以用 list_collections 方法查看已有的集合，或用 delete_collection 方法删除一个集合，命令如下：

```
chroma_client.list_collections()
chroma_client.delete_collection(name="my_collection")
```

还有一些其他常用的方法，如获取一个存在的 collection 对象：

```
collection = client.get_collection("testname")
```

如果不存在就创建 collection 对象，一般用下面的代码更多一点：

```
collection = client.get_or_create_collection("testname")
```

3. 集合对象及相似检索操作

下面通过完整的代码进行演示，代码如下：

```
#第 5 章/ChromaDemo.py
#导入 ChromaDB 库
import chromadb
#从 LangChain 库中导入 TextLoader 类
from langchain.document_loaders import TextLoader
#从 LangChain 库中导入 ModelScopeEmbeddings 类
from langchain.embeddings import ModelScopeEmbeddings
#从 LangChain 库中导入 CharacterTextSplitter 类
from langchain.text_splitter import CharacterTextSplitter
#从 LangChain 库中导入 Chroma 类
from langchain.vectorstores import Chroma
#创建一个 Chroma 客户端
chroma_client = chromadb.Client()
#如果使用的是服务器上的数据库，则可以使用下面的代码
#chroma_client = chromadb.HttpClient(host="localhost", port=8000)
#创建一个持久化的客户端，用于将数据保存在本地
client = chromadb.PersistentClient(path="/data/chroma")
#创建一个名为 my_collection 的集合
collection = chroma_client.create_collection(name="my_collection")
#查看已有的集合
chroma_client.list_collections()
#删除一个名为 my_collection 的集合
chroma_client.delete_collection(name="my_collection")
#获取一个已存在的 collection 对象
collection = client.get_collection("table1")
#如果不存在就创建一个名为"table1"的 collection 对象
collection = client.get_or_create_collection("table1")
#向集合中添加文档
collection.add(
    documents=["文档 1", "文档 2"],
    metadatas=[{"source": "my_source"}, {"source": "my_source"}],
    ids=["id1", "id2"]
)
#统计集合中的文档数量
collection.count()
#批量添加文档
collection.add(
    embeddings=[[1.6, 2.8, 6.6], [9.9, 2.6, 2.8]],
    metadatas=[{"style": "style1"}, {"style": "style2"}],
    ids=["uri9", "uri10"],
)
collection.add(
```

```
    documents=["doc001", "doc002"],
    metadatas=[{"style": "style1"}, {"style": "style2"}],
    ids=["id1", "id2"],
)
#增加或者更新文档,如果原来就有,则覆盖;如果原来没有,则添加
collection.upsert(
    ids=["id1", "id2", "id3", ...],
    embeddings=[[1.6, 2.9, 3.5], [2.9, 6.8, 9.9], [1.9, 2.8, 8.8], ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "5"},
{"chapter": "29", "verse": "11"}, ...],
    documents=["doc001", "doc002", "doc003", ...],
)
#查看集合中的前 5 个文档
collection.peek()
#查询集合中的文档
collection.query(
    query_embeddings=[[2.6, 2.9, 3.5], [6.1, 6.8, 8.8]],
    n_results=2,
    where={"style": "style2"}
)
#读取原始文档
raw_documents = TextLoader("分布式机器学习实践.txt").load()
#分割文档
text_splitter = CharacterTextSplitter(chunk_size=128, chunk_overlap=0)
documents = text_splitter.split_documents(raw_documents)
#生成 Embedding 向量
model_id = "damo/nlp_corom_sentence-embedding_chinese-base"
embeddings = ModelScopeEmbeddings(model_id=model_id)
db = Chroma.from_documents(documents, embedding=embeddings)
#相似检索
query = "第 6 章 Spark 分布式机器学习平台的具体目录是什么?"
docs = db.similarity_search(query, k=5)
#打印结果
for doc in docs:
    print("metadata:", doc.metadata)
    print("page_content:", doc.page_content)
```

在探讨了各种向量数据库,包括 Faiss、Milvus、Pinecone 和 Chroma 之后,接下来转向实际应用这些技术的场景。重点讨论如何利用向量数据库和其他相关技术,特别是 RAG,以此来解决企业在实际应用中遇到的问题。

5.4 RAG 应用实践

本节将探讨如何利用 RAG 构建企业私有数据的知识问答系统,以及面对大模型落地时所遇到的挑战和相应的优化策略。

5.4.1 基于大模型构建企业私有数据的知识问答

随着大模型的浪潮席卷各行各业，其在处理通用性问题上的卓越表现已得到广泛认可。然而，当面对特定的专业场景、行业细分领域或企业私有数据时，通用大模型往往会暴露出专业知识储备不足的短板。在这种情况下，传统的大模型训练微调方法虽然能够在一定程度上提升模型的专业性，但其高昂的成本和复杂的实施过程往往令许多企业望而却步。相比之下，基于 RAG 的技术方案以其高效、灵活的特点，逐渐成为解决这一问题的优选。RAG 架构通过结合检索技术和生成模型的优势，不仅能在保持大模型原有功能的基础上，有效补充专业知识，还能够根据企业私有数据的特点进行定制化调整，从而实现更为精准的知识问答服务。接下来将从 RAG 架构的基本原理出发，深入探讨其在构建企业私有数据知识问答系统中的应用。下面将详细介绍 RAG 的关键技术细节，包括数据准备、检索策略、提示工程等方面，并通过一份具体的实践案例，展示如何在实际操作中运用 RAG 技术，为企业打造一款既专业又高效的私有数据知识问答系统。RAG 的核心思想在于通过检索机制获取相关知识，并将这些知识巧妙地融入提示中，从而使大模型在生成答案时能够充分参考这些知识，以提供更加准确、合理的回答。简而言之，RAG 的核心可以被理解为“检索＋生成”的过程。在这个过程中，首先涉及的是高效的向量数据库，它承担着存储和检索知识的重要任务。通过这种技术，可以迅速地找到与问题相关的信息，这被称为“召回目标知识”。紧接着，借助强大的大模型和精心设计的提示工程，这些召回的知识被有效地利用，进而生成满足用户需求的答案。RAG 的应用流程大致可以分为两个阶段。

（1）数据准备阶段：此阶段包括数据提取、文本分割、向量化处理及最终的数据入库。具体来讲，首先从各种来源提取相关文本数据，然后对这些数据进行分割，以适应后续的处理需求。接下来，使用 Embedding 技术将文本转换为向量形式，这一步骤对于后续的快速检索至关重要。最后，将这些向量化的数据存储到数据库中，以备后续查询和使用。

（2）应用阶段：当用户提出问题时，系统首先进行数据检索，从数据库中召回与问题相关的知识，然后将这些知识注入提示中，作为大模型生成答案的参考。最终大模型根据这些信息生成一个精确且符合要求的答案。

在实施 RAG 架构时，需要注意各个环节的技术细节和潜在问题，例如，在数据提取阶段，需要确保数据来源的多样性和质量；在文本分割和向量化处理过程中，要考虑到不同语言的特性和复杂性；在数据入库时，要考虑如何优化存储结构和检索效率。此外，在应用阶段，如何设计有效的提示词以引导大模型生成高质量的答案也是一个关键问题。

1. 数据准备阶段

数据准备阶段是构建 RAG 知识问答系统的关键步骤，它通常是一个离线过程，涉及将私有数据转换为可用于检索和生成答案的形式。这个阶段主要包括以下几个环节。

1）数据提取

首先，需要从企业内部的各种数据源中提取相关信息。这可能包括处理多种格式的数据，如文本、图像或视频，以及从不同的系统和平台中获取数据。为了统一处理这些数据，它

们会被转换成一个共同的格式或范式。此外，还需要对数据进行预处理，例如过滤掉无关信息、压缩数据以减少存储空间的需求，以及格式化数据以便后续处理。同时，提取数据中的元数据，如文件名、标题和时间戳等，这些信息对于后续的数据管理和检索至关重要。

2）文本分割

需要文本分割的原因是大模型的上下文窗口是有限制的，所以需要把大的文本切割成多段，每次提问，只给大模型相关的文本段＋提示词，以供大模型回答。在进行文本分割时，需要考虑两个主要因素：一是所使用的 Embedding 模型对 Token 数量的限制，二是保持文本的语义完整性以保证检索效果。常见的文本分割方法包括按句子分割，保留每个句子的完整语义，或者按照固定的长度分割文本，尽管这可能会导致一些语义信息的丢失，通常会在头尾部分添加一些冗余信息来缓解这个问题。

3）向量化

向量化是将文本数据转换为数值形式向量的过程，这对于后续检索很重要。前面已经讲解了文本向量模型，例如 GTE、acge_text_embedding 模型、BGE、M3E 及 OpenAI 的 text-embedding 模型等。这些模型虽能满足大多数需求，但在特定场景下，如涉及专业术语或专有词汇时，就需要对开源模型进行微调或训练一个专门针对特定场景的 Embedding 模型。

4）数据入库

完成向量化之后，下一步是构建索引并将向量化的数据存储到数据库中。适用于 RAG 场景的数据库包括 Faiss、Chroma、Milvus 和 Pinecone 等，这些数据库在前面章节也详细讲解过。选择合适的数据库通常需要考虑业务需求、硬件条件和性能要求等多种因素。

总体来讲，数据准备阶段是构建 RAG 知识问答系统的基础，需要仔细规划和执行，以确保系统的准确性和效率。

2. 应用阶段

在应用阶段，目标是根据用户的提问，通过高效的检索手段，找到与问题最相关的知识，并将其融入提示中。随后，大模型会结合当前的提问和检索到的相关知识生成准确的答案。这一阶段的关键步骤包括数据检索和注入 Prompt。

1）相似性检索

数据检索是应用阶段的第 1 步，它的目的是从大量的数据中快速找到与用户提问最相关的知识。为了提高检索的准确性和覆盖率，通常会采用多种检索方法的组合。常见的数据检索方法包括相似性检索和全文检索。相似性检索是通过计算查询向量与数据库中所有存储向量之间的相似性得分来确定哪些记录与查询最匹配。常用的相似性计算方法包括余弦相似性、欧氏距离和曼哈顿距离等。全文检索则是一种经典的检索方式，它在数据存储时通过关键词构建倒排索引，在检索时通过关键词进行全文搜索，以找到对应的记录。

2）相似文本段＋Prompt 提示词拼接以供大模型回答

Prompt 是大模型的直接输入，对于模型输出的准确性起着至关重要的作用。在 RAG 场景中，一个有效的 Prompt 通常包括任务描述、背景知识（检索到的相关文本）和任务指令（通常是用户的具体提问）。根据任务的具体场景和大模型的性能，还可以在 Prompt 中加

入其他指令来进一步优化模型的输出。

例如，一个简单的知识问答场景的 Prompt 可能如下。

【任务描述】

作为一个 AI 助手，你的任务是提供准确的信息来回答用户的问题。请仔细阅读下面提供的【背景知识】，并基于这些信息给出你的回答。

【背景知识】

{content} //这里填充的是通过 AI 检索算法从大量数据中检索出来的相关内容，可能是关于某个主题的详细解释、一组统计数据或者某个事件的描述。

【问题】

请问最新的 AI 技术在自然语言处理领域的应用有哪些突破？

值得注意的是，Prompt 的设计并没有固定的规则，而是依赖于设计者的经验和对大模型输出的理解。在实际应用中，往往需要根据大模型的实际输出对 Prompt 进行持续调优，以达到最佳的问答效果。

3. 实际案例代码实践

企业拥有大量的私有数据，这些数据可能存储在关系数据库（如 MySQL）中，或者在大数据平台上，如 Hadoop 的 HDFS、Hive、HBase，也可能存在于实时数据仓库（如 GreenPlum、ClickHouse、Doris）中，甚至是以文档形式存在的，如 TXT 文本、Word 文档或 PDF 文件。无论数据以何种形式存在，目标都是将这些数据读取出来，并将其转换成适合处理的字符串格式。本案例需扫描目录上方二维码，将分布式机器学习实战.txt 下载到本地，然后用这个文本文件来模拟企业私有数据，并基于 LangChain 开发框架，实现一种简单的 RAG 问答应用示例。

1）环境准备

在开始构建基于企业私有数据的 RAG 问答系统之前，需要准备好必要的环境和工具。使用 pip 命令安装即可，命令如下：

```
pip install datasets langchain sentence_transformers tqdm chromadb langchain_wenxin
```

其中，datasets 是一个用于处理和加载数据集的库，它可以方便地加载和处理数据。LangChain 是一个强大的开发框架，前面章节讲解过，它提供了构建复杂对话系统所需的组件和工具。sentence_transformers 是一个用于句子嵌入的库，它可以将文本转换为向量，以便进行后续处理，本案例的文本向量模型使用 M3E，它是一款功能强大的开源 Embedding 模型，它支持微调和本地部署，适用于各种自然语言处理任务，通过 M3E，可以将文本数据转换为向量形式，以便进行后续的相似性检索。tqdm 是一个进度条库，它可以在执行长时间的任务时看到当前的进度，从而更好地监控程序的运行状态。Chroma 是一个高性能的向量数据库，它支持快速地进行相似性检索和大规模数据存储。通过安装 Chroma，可以高效地管理和检索私有数据集。文心一言是百度推出的大模型，通过安装文心一言插件，可以利用其强大的语言生成能力来回答用户的问题。

在安装完上述依赖后，环境就准备好了，接下来就可以开始构建 RAG 问答系统了。

2）加载本地数据并分割文档，然后更新向量数据库

在构建 RAG 问答系统时，经常需要处理大量的文本数据。为了更好地管理和检索这些数据，可以将它们分割成较小的文本段。以下是使用 Python 和 LangChain 库来实现这一过程的详细步骤：首先，需要加载文本数据，本案例使用“分布式机器学习实践.txt”文本文件，其中包含了要处理的数据，使用 LangChain 库的 TextLoader 类来加载这个文件。接下来将加载的文档分割成更小的文本段。在这里选择按照固定字符长度（例如 128 个字符）来分割文本。为此，可以使用 LangChain 的 CharacterTextSplitter 类，这样就得到了一个由多个文本段组成的列表，每个文本段的长度都不超过 128 个字符。这对于后续的文本处理和检索是非常有用的。

需要注意的是，虽然使用了固定字符长度来分割文本，但在实际应用中，可能会根据具体的应用场景和需求来选择不同的分割策略，从而将一个长文档分割成更小的块，以便适合模型的上下文窗口。LangChain 有许多内置的文档转换器，可轻松地拆分、组合、过滤和操作文档。当需要处理很长的文本时，有必要将文本分割成块。在理想情况下，希望将语义相关的文本片段放在一起，“语义相关”的含义可能取决于文本的类型，文本拆分器的工作方式如下：

（1）将文本分成语义上有意义的小块（通常是句子）。

（2）开始将这些小块组合成一个更大的块，直到达到一定的大小（通过某些函数来测量）。

（3）一旦达到该大小，将该文本块作为自己的文本块，然后开始创建一个有一些重叠的新文本块（以保持文本块之间的上下文），例如，如果希望保留完整的句子结构，则可以选择按照句子边界来分割文本。这可以通过 LangChain 的 SentenceTextSplitter 类来实现。

总体来讲，通过以上步骤，可以有效地将大型的文本数据分割成更小的文本段，然后对分割后的文本段进行嵌入，并写入向量数据库。这里选用北京智源人工智能研究院的 BGE 向量模型作为 Embedding 模型，向量数据库选用 Chroma。首先加载整个本地数据并分割文档，然后更新向量数据库，代码如下：

```
#第 5 章/RAGTextLoadSplitAndUpdateChroma.py
#导入 LangChain 库的 TextLoader 类，用于加载本地文本文件
from langchain.document_loaders import TextLoader
#导入 LangChain 库的 CharacterTextSplitter 类，用于按字符分割文本
from langchain.text_splitter import CharacterTextSplitter
#导入 SentenceTextSplitter，用于将一个字符串切割成独立的句子
from langchain import SentenceTextSplitter
from langchain.embeddings import HuggingFaceBgeEmbeddings
from langchain.vectorstores import Chroma
#导入 LangChain 库的 vectorstores 模块，用于创建向量数据库
from langchain.vectorstores import Chroma
#创建一个 TextLoader 对象，指定要加载的本地文本文件路径
loader = TextLoader("分布式机器学习实践.txt")
#使用 loader 对象的 load 方法加载文档，返回一个包含文档内容的列表
```

```
documents = loader.load()
#创建一个 CharacterTextSplitter 对象,将分块大小设置为 128 个字符,分块之间无重叠
text_splitter = CharacterTextSplitter(chunk_size=128, chunk_overlap=0)
#使用 text_splitter 对象的 split_documents 方法对加载的文档进行分割,返回分割后的文档
#列表
documents = text_splitter.split_documents(documents)
#创建一个 SentenceTextSplitter 实例
splitter = SentenceTextSplitter()
#待切割的长文本
text = "这是一个长文本,我们将其切割成句子。每个句子将被正确分隔开。实际用你的待分割的
长文本替换"
#使用 SentenceTextSplitter 切割文本,得到句子后可以再拼接更长的文本段
sentences = splitter.split_text(text)
#输出切割得到的句子列表,这里开始可以自行根据实际场景将这些句子中的小块组合成一个更大
#的块,直到达到一定大小(通过某些函数来测量)
for sentence in sentences:
    print(sentence)       #这里根据实际情况实现句子拼接
#使用北京智源人工智能研究院的 BGE 向量模型
model_name = "BAAI/bge-base-zh"
model_kwargs = {'device': 'cpu'}
encode_kwargs = {'normalize_embeddings': True}
embedding = HuggingFaceBgeEmbeddings(
                model_name=model_name,
                model_kwargs=model_kwargs,
                encode_kwargs=encode_kwargs,
                query_instruction="为文本生成向量表示用于文本检索"
            )
#创建一个 Chroma 对象,用于将文档加载到向量数据库中
db = Chroma.from_documents(documents, embedding)
#执行相似性搜索
db.similarity_search("第 6 章 Spark 分布式机器学习平台的具体细节目录是什么?")
```

3) Prompt 提示模板设计

下面给出一个 Prompt 模板设计例子,在实际业务场景中需要针对场景特点有针对性地进行调优。

【任务描述】

请仔细阅读并理解用户提供的上下文信息,然后根据这些信息回答问题。务必遵循我们的回答指南。

【背景知识】

{{context}}

【回答指南】

- 需要确保您的回答完全基于所提供的背景知识,不要依赖外部信息或常识。
- 如果您无法找到相关信息来回答用户的问题,则直接回复“抱歉,我无法找到与这个问题相关的答案。”

{question}

4）基于 RetrievalQAChain 构建知识问答

ConversationalRetrievalChain 是基于 RetrievalQAChain 构建的，它能够跟踪和管理历史聊天记录。具体来讲，它通过定义一个名为 memory 的组件来实现这一点。在操作过程中，ConversationalRetrievalChain 首先会将历史问题和当前输入问题合并成一个新的独立的问题，然后它会使用这个新问题去检索相关的知识。最后，它将检索到的知识和生成的新问题一起输入一个提示中，以便大型模型可以生成相应的回答。这个过程不仅考虑了当前的问题，还考虑了之前的对话历史，使生成的回答更加连贯和一致。这种方法在处理复杂的、多轮的对话任务时尤其有用，因为它可以帮助模型更好地理解和回应用户的需求，示例代码如下：

```
#第 5 章/RAGRetrievalqaChain.py
#导入 LangChain 库的相关模块
from langchain import LLMChain
from langchain_wenxin.llms import Wenxin
from langchain.prompts import PromptTemplate
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationalRetrievalChain
from langchain.prompts.chat import ChatPromptTemplate,
SystemMessagePromptTemplate, HumanMessagePromptTemplate
#选择百度文心一言大模型，这里使用的是 Wenxin 的 ERNIE-Bot 模型
llm = Wenxin(model="ernie-bot", baidu_api_key="baidu_api_key", baidu_secret_key=
"baidu_secret_key")
#创建一个检索器，用于从数据库中检索信息。db 变量接上面 RAGTextLoadSplitAndUpdateChroma.py
#的 db = Chroma.from_documents(documents, embeddings)
retriever = db.as_retriever()
#创建一个会话缓冲内存，用于存储和管理对话历史
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
#创建一个基于大模型的对话检索链，这个链结合了检索器和会话缓冲内存
qa = ConversationalRetrievalChain.from_llm(llm, retriever, memory=memory)
#使用创建的对话检索链来回答一个问题
qa({"question": "第 6 章 Spark 分布式机器学习平台的具体细节目录是?"})
```

当构建了基于大模型的企业私有数据知识问答系统后，接下来的挑战是如何优化这个系统，使其在实际应用中更加高效和准确。

5.4.2 应对大模型落地挑战的优化策略

面对大模型落地在实际应用中会遇到各种挑战，对 RAG 来讲，从文档预处理、嵌入模型的选择与优化、向量数据库的构建与管理、高效的查询检索机制，到最终生成高质量回答的每个环节都蕴含着丰富的优化策略空间。以下是在这 5 个核心环节中穿插的 12 项具体优化措施，旨在全面提升系统的效能、准确度及用户体验。

1. 文档预处理

在构建一个高效的 RAG 系统时，首先需要准备各种数据及知识文档。在实际应用中，这些数据可能存储在关系数据库（如 MySQL）中，或者在大数据平台上，如 Hadoop 的

HDFS、Hive、HBase，也可能存在于实时数据仓库（如 GreenPlum、ClickHouse、Doris）中，还有各种格式的知识源，包括 Word 文档、TXT 文件、CSV 数据表、Excel 表格，甚至 PDF 文件、图片和视频等多模态数据，因此，第 1 步是利用专门的文档加载器（如 PDF 提取器）或多模态模型（如 OCR 技术），将这些多样化的知识源转换为大型语言模型可以理解的纯文本数据，例如，在处理 PDF 文件时，可以使用 PDF 提取器来提取文本内容，而对于图片和视频，OCR 技术则能够识别并转换其中的文字信息。然而，由于文档可能存在过长的现象，所以需要执行一个关键的步骤：文档切片。这意味着将长篇文档分割成多个较小的文本块，以便更有效地处理和检索信息。这样做不仅有助于减轻模型的负担，还能提高信息检索的准确性。为了构建一个高性能的 RAG 系统，确保原始知识数据的准确性和清洁度至关重要，以下是一些关键的数据优化策略。

1）优化文档读取器和多模态模型

为了确保数据的准确性，特别是在处理像 CSV 表格这样的文件时，需要优化文档读取器和多模态模型。单纯的文本转换可能会丢失表格的原有结构，因此需要引入额外的机制来在文本中恢复表格结构，例如使用分号或其他符号来区分数据。

2）基本文本清理

对知识文档进行基本的文本清理，包括规范文本格式、去除特殊字符和不相关信息，以及删除重复文档或冗余信息。

3）实体解析

消除实体和术语的歧义，以实现一致的引用，例如，将“LLM”“大语言模型”和“大模型”标准化为通用术语。

4）文档划分

合理地划分不同主题的文档，确定不同主题应该集中在一处还是分散在多处。如果人类无法轻松地判断需要查阅哪个文档才能回答常见的问题，则检索系统也无法做到。

5）数据增强

通过同义词、释义甚至其他语言的译文来增加语料库的多样性。

6）用户反馈循环

基于现实世界用户的反馈不断地更新数据库，标记它们的真实性。

7）时间敏感数据

对于经常更新的主题，实施一种机制来使过时的文档失效或更新。

通过这些优化，可以确保 RAG 系统的知识库是准确、清洁且易于检索的，从而提高系统的性能和用户体验。

2. 文本分块

在 RAG 系统中，将文档分割成多个文本块并进行向量嵌入的目的是在保持语义连贯性的同时，减少噪声，提高与用户查询的相关性。选择合适的分块策略是关键，需要在包含足够上下文信息和避免过多不相关信息之间找到平衡。以下是几种分块方法和分块大小的选择建议。

(1) 固定大小的分块：简单直接，设定固定的字数，并保持一定的重叠以确保语义上下文不丢失。

(2) 内容分块：根据文档内容进行分块，如标点符号或高级库的句子分割功能。

(3) 递归分块：推荐方法，通过重复应用分块规则递归分解文本，灵活调整块大小以适应不同的信息密度。

(4) 从大到小分块：将文档分割成不同大小的块并存入数据库，保留上下级关系，适用于需要大量存储空间的场景。

(5) 特殊结构分块：针对特定结构化内容的分割器，如 Markdown、LaTeX、代码等。

在选择分块大小时，应考虑以下因素。

(1) 嵌入模型的最佳输入大小：如 OpenAI 的 text-embedding-ada-002 模型在 256 或 512 大小的块上效果较好。

(2) 文档类型和用户查询的长度及复杂性：长篇文章或书籍适合较大分块，而社交媒体帖子适合较小分块。

(3) 用户查询的性质：简短具体查询适合较小分块，复杂查询可能需要较大分块。在实践中，可能需要不断地实验和调整分块大小，128 大小的分块在某些测试中是最佳选择，可以作为起点进行测试。

3. 文本向量化

前面已经讲到了文本向量模型：阿里巴巴的 GTE 向量模型、中文 acge_text_embedding 模型、BGE、M3E、OpenAI 的 text-embedding 模型等，当然随着时间的推移，还可能涌现更多的向量化工具。在企业私有数据 RAG 应用实践中需要根据企业的实际情况来选择不同的向量模型，例如是用开源的，还是商业的；是用中文、英文的，还是中英文混合的；是用通用的，还是基于企业垂直细分数据进行训练微调的，这些因素都需要考虑。

4. 元数据

在向量数据库存储向量信息时，部分数据库支持合并元数据。给向量加上元数据标签，极大地提升了搜索效率，对优化查询结果意义重大。以“日期”这一常见元数据为例，它助力按时间排序及筛选，例如在开发邮件查询应用时，最近的邮件通常与用户查询更契合，而仅凭向量比较难直接评估邮件与查询的相关性。故将邮件日期作为元数据加入，可确保检索时最新邮件优先出现，增强了结果相关度。此外，纳入章节引用、关键文本、标题或关键词等作为元数据，不仅提升了信息检索的精确性，还为用户提供了更翔实、更精准的搜索体验。

5. 多级索引和路由

面对元数据区分上下文类型挑战，采用多重索引技术是解决方案之一。该技术的核心在于数据的分层分类组织，通过创建多个索引有针对性地满足不同查询需求，例如摘要型、具体答案寻求型或时间敏感型问题。配合多级路由机制，确保查询高效导向最适合的索引，依据查询特性智能分配资源，实现精准匹配，例如，针对“最新科幻电影推荐”的查询，系统会先导向热点话题索引，再利用影视专用索引给出推荐。总体而言，多重索引与多级路由策略

能显著地提升大数据处理效能、信息提取精度及用户体验，从而增强系统的整体性能。

6. 索引与查询算法

在处理大规模向量数据的索引与查询中，为了高效地找到与查询向量“足够接近”的数据点，采用了多种策略平衡搜索质量和速度，以及内存开销，以下是关键方法。

(1) 聚类算法：将向量空间划分为多个簇，首先定位查询向量最近的簇，随后在该簇内部进行精细搜索。这种方法虽不完美，但有效地缩小了搜索范围，需权衡精度与效率。

(2) 位置敏感哈希：通过设计哈希函数故意增加输出碰撞，使空间上相近的向量落入同一“桶”，从而快速定位潜在的匹配项。此法牺牲了一定的准确性以换取速度。

(3) 量化乘积：包括聚类后用中心点代表簇内数据的量化过程，以及对高维向量的子空间量化策略，以大幅减少存储需求。尽管引入码本带来额外开销，但通过子空间分割和独立量化，有效地控制了内存增长。

(4) 分层导航小世界算法：模拟社交网络中的“六度分割理论”，从一个起点逐步跳跃至最相似的向量，旨在通过有限步数快速定位最佳匹配，体现了用内存换取速度和质量的策略。

综上所述，这些技术各有侧重，分别通过数据分组、哈希策略、数据压缩及智能导航机制，优化了向量数据的索引与检索过程，力求在不同维度上达到最佳的性能平衡。

7. 查询转换

在 RAG 系统中，优化查询以提升向量匹配效果和搜索质量是关键。以下是几种有效的查询转换策略。

(1) 历史对话整合重述：利用之前的对话上下文，通过大模型重述问题，确保查询不仅准确而且蕴含历史信息，增强向量表达的相关性。

(2) 假设文档嵌入：创造性地生成一个假设性回复，结合用户原始查询一同用于检索，即便假设内容可能不完全准确，它也能指向相关的文档结构和信息，拓宽检索范围。

(3) 退后提示：面对复杂或宽泛查询，构建更高层次、更为概括的辅助问题，例如从具体事件上升到背景领域，以获得更全面的检索结果。

(4) 多查询检索：针对复杂查询，通过大模型生成多个相关子查询，多路并进，确保覆盖所有可能的角度和信息源，提高召回率和精确度。

这些策略共同作用，通过灵活调整和丰富查询表达，增强了 RAG 系统处理复杂请求的能力，提升了搜索的效率与准确性。

8. 检索参数

在向量数据库检索过程中，优化检索参数是提升搜索效率与准确性的关键步骤，以下是核心要点总结。

(1) 稀疏与稠密搜索权重调节：结合稠密向量搜索与稀疏关键字匹配(如 BM25 算法)，根据应用场景调整两者的权重，例如，0.6 的比例意味着更多依赖于向量相似度，同时考虑关键字的重要性，以平衡精确匹配与语义理解。

(2) 结果数量设定：合理设定检索返回的结果数量，确保既全面覆盖查询需求，又避免

信息冗余。更多结果有助于捕捉问题的全貌，但需平衡资源消耗与回答质量。

（3）相似度计算方法选择：选用合适的向量相似度计算方法，如常用的余弦相似度，它侧重于向量方向而非长度，适合评估语义相似性，尤其是在不同长度文本间的比较。选择时需参考嵌入模型的兼容性指导。

这些参数的精细调整，能够使检索过程更加高效且针对性更强，从而在 RAG 系统中实现更高质量的信息检索和问题解答。

9. 高级检索策略

在高级检索策略中，为了优化向量数据库的检索效能和回答质量，一系列创新策略被提出，以下为关键点汇总。

（1）上下文压缩：利用大模型减少文档文块中的无关信息，实现内容精练，提升检索效率和大模型响应质量。

（2）句子窗口搜索：解决上下文不足问题，通过纳入问题匹配文块周围的内容，增强大模型对整体上下文的理解。

（3）父文档搜索：文档分为主文档和子文档两层，匹配子文档后，将其所属父文档一并提交给大模型，确保回答的全面性。

（4）自动合并：基于多层级结构，当某父节点下的多数子节点匹配查询时，直接返回该父节点，提高检索效率。

（5）多向量检索：为文档创建多维度向量，涵盖不同粒度的内容和摘要等，增强检索的全面性和精准度。

（6）多代理检索：结合多种检索策略形成智能代理网络，通过分工合作提升检索效果，适应复杂查询需求。

（7）Self-RAG：引入检索和反思双评分机制，动态决定是否进行检索，并评估检索内容的相关性，以生成更高质的答案。

这些策略旨在通过精细化管理和优化检索过程，增强 RAG 系统的理解深度、响应速度及答案的准确性，是构建高效知识检索与应用系统的重要组成部分。

10. 重排模型

重排模型在语义搜索中的作用主要是解决初次检索结果可能与用户查询意图不完全吻合的问题。尽管初步检索能找出语义上相似的文档，但这些结果未必能直接满足用户的实际需求。重排模型通过以下方式优化检索结果。

（1）深入相关性评估：利用深度学习技术，如 Cohere 模型，对初始检索结果进行二次分析，考虑更多维度的特征。

（2）多维度考量：在重排过程中，模型会考量查询意图、词汇的多重含义、用户历史行为及上下文信息，确保结果与用户需求高度匹配。

（3）结果优化排序：依据上述分析，将最符合用户查询意图且最新的、相关性强的内容（如最新上映的科幻电影列表）排在检索结果前列，而将历史概述或边缘相关材料后置。

因此，重排模型是提升检索系统精准度和用户体验的关键环节，通过智能化地调整结果

顺序，确保用户能够快速地获取最相关的信息，是构建高效搜索引擎不可或缺的一部分。在应用 RAG 架构设计系统时，整合重排模型是一个值得推荐的做法，以进一步提升系统性能和用户满意度。

11. 提示词

优化大模型的响应质量，特别是减少主观性和幻觉，增强针对性和实用性，可通过精心设计的提示词实现，关键策略如下。

(1) 明确指示回答依据：在提示词中清晰地说明回答应严格基于搜索结果或其他给定的上下文信息，避免无依据的自由发挥，从而控制主观性，如“回答基于搜索结果，不添加额外信息”。

(2) 角色界定：设定模型角色身份，如“智能客服”，强调目标是提供准确、有助益的问题解决方案，保持回答的专业性和友好性，同时避免冗余信息，确保回答聚焦且实用。

(3) 融合客观性与主观性：根据应用场景，灵活调整提示词，允许模型在必要时融入适度的主观理解或知识解释，平衡客观事实与个性化解读。

(4) 使用 Few-Shot 示例：在提示中包含具体的问答示例，作为模型学习的模板，展示如何有效地结合检索到的知识进行回答。这不仅提升了答案的精确度，还促进了模型在特定场景下的应用能力。

综上所述，通过精细化设计提示词，明确回答规则、界定模型角色、灵活调整回答风格，并运用 Few-Shot 学习策略，可以显著地提升大模型生成内容的客观性、准确性及实用性，同时降低非事实性内容的产生。

12. 大模型生成回答

大模型生成回答环节的核心要点如下。

(1) 大模型选型考量：根据项目需求挑选合适的大模型，考虑因素涉及模型的开放性与专属性、推理成本效率及处理上下文的能力。多样化选项适应不同场景和预算约束。

(2) 开发框架利用：利用成熟的开发框架(如 LLaMAIndex 或 LangChain)加速 RAG 系统的构建。这些框架不仅整合了大模型应用的复杂流程，还内置了便利的调试工具。

(3) 调试与优化工具：框架提供的 Debugging 工具至关重要，允许开发者自定义回调函数来监控模型运行详情，如检查实际调用的上下文片段、追溯检索信息至源头文档，这对于性能调优和问题排查极为有用。

总体而言，大模型的选择与集成是实现高质量内容生成的关键步骤。借助高效开发框架及其配套的调试功能，能够确保模型运行的透明度与可控性，优化整体系统的响应质量和运行效率。

5.5 主流开源 RAG 项目

在大模型与实际应用场景深度融合的过程中，RAG 技术成为解决非结构化数据处理、知识高效利用的关键方案。开源社区的蓬勃发展催生了众多功能各异、架构创新的 RAG

项目，它们通过模块化设计、生态整合能力和场景化优化，为开发者提供了从数据接入、索引构建到智能问答的全流程解决方案。本节将聚焦主流开源 RAG 项目，深入解析其技术特性、系统架构及落地实践，涵盖兼具工程化能力的 RAGFlow、灵活高效的 LLaMAIndex 生态方案，以及代表结构化检索前沿的 GraphRAG 等，为读者呈现 RAG 技术的多元化发展路径与实践价值。

5.5.1 RAGFlow

RAGFlow 是一款依托深度文档理解技术构建的开源 RAG 引擎。RAGFlow 可以为各种规模的企业及个人提供一套精简的 RAG 工作流程，结合大模型针对用户各类不同的复杂格式数据提供可靠的问答及有理有据的引用。

1. 功能特色

RAGFlow 功能特色包含以下几个方面。

1）优质输入，优质输出

面对非结构化数据解析与海量知识检索难题，RAGFlow 凭借深度文档理解技术，践行“优质输入，优质输出”理念。

（1）RAGFlow 依托深度文档理解技术，能够从 PDF、扫描件、表格等 20 多种复杂格式的非结构化数据中精准提取有效信息。

（2）RAGFlow 突破性实现无限上下文场景下的高效检索，在海量文档中快速定位所需知识。

2）模板化文本切片

为破解知识抽取过程不可控、难解释的难题，RAGFlow 打造了模板化文本切片体系。

（1）独创可视化切片管理系统，支持手动调整切片边界。

（2）提供多样化的文本切片模板（学术论文/法律合同/技术手册等），实现可控可解释的知识抽取。

3）可追溯低幻觉输出

为攻克 AI 答案幻觉难题、保障信息可靠，RAGFlow 构建可追溯低幻觉输出体系。

（1）答案生成时自动标注引用来源，支持原始文档快照回溯。

（2）内置幻觉检测机制，通过多维度置信度评分过滤不可靠信息。

4）兼容各类异构数据源

RAGFlow 支持丰富的文件类型，包括 Word 文档、PPT、Excel 表格、TXT 文件、图片、PDF、影印件、复印件、结构化数据、网页等。

5）全自动 RAG 工作流

为满足不同场景下知识问答与系统集成需求，RAGFlow 打造全自动 RAG 工作流。

（1）全面优化的 RAG 工作流可以支持从个人应用乃至超大型企业的各类生态系统。

（2）大模型及向量模型均支持配置。

（3）基于多路召回、融合重排序。

(4) 提供易用的 API,可以被轻松地集成到各类企业系统。

2. 系统架构

RAGFlow 系统是一个高效智能的信息处理平台,其架构由多个关键组件协同运作,实现对复杂查询的快速响应与精准处理,如图 5-9 所示。

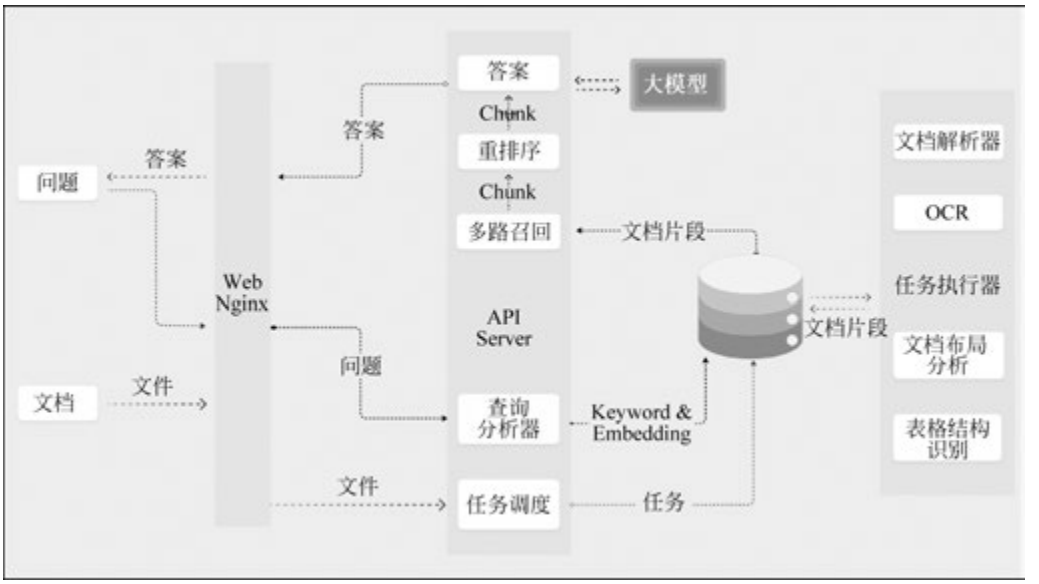

图 5-9 RAGFlow 系统架构

该架构通过模块化设计实现了从数据输入、处理到答案生成的全流程协同,各组件分工明确且高效联动,支撑复杂场景下的智能问答与知识检索需求。

1) 输入部分

RAGFlow 系统的高效运行始于精准的输入,其输入部分涵盖两大关键要素。

(1) 问题:用户输入的查询内容,是系统处理的起点。

(2) 文档:各种格式的原始文档,是信息的来源,如 PDF、Word、Excel 等。通过 Web Nginx 接收文件输入。

2) 核心处理组件

RAGFlow 系统的核心处理组件如同精密的齿轮,协同驱动系统高效运转,各有其关键作用。

(1) 文档解析器:系统的“大脑”,负责解析多种格式文档,提取文本、图像、表格等关键内容,为后续处理奠定基础。

(2) OCR:对文档中的图像文字进行识别转换,将图像中的文字转换为可处理的文本信息。

(3) 任务执行器:执行各类相关任务,保障系统各环节顺利运转。

(4) 文档布局分析:分析文档的布局结构,辅助准确提取信息。

(5) 表格结构识别：专门识别文档中表格的结构，便于提取表格数据。

(6) 任务调度：接收输入的文档，分配相关任务，将文档处理任务分发给对应的组件。

(7) 查询分析器：系统的“神经系统”，深入分析用户查询，提取关键信息，为检索提供精准指导。

(8) 多路召回：作为系统的“搜索引擎”，依据查询分析器提取的关键信息，从海量文档中快速检索相关信息。

(9) 重排序：系统的“过滤器”，对检索到的信息进行排序和过滤，去除冗余和不相关数据，确保输出信息的相关性和价值。

(10) 大模型：系统的“语言生成器”，整合重排后的信息，生成最终答案或输出，保证答案准确且表达自然流畅。

3) 交互与输出部分

RAGFlow 系统的交互与输出部分是连接系统与用户的桥梁，主要包含以下关键要素。

(1) API Server：提供应用程序接口服务，实现各组件间及系统与外部的交互通信。通过 API，系统可以被轻松地集成到各类企业系统。

(2) 答案：最终输出给用户的结果，是经过一系列处理后生成的准确、有用信息，并可通过 Web Nginx 返给用户。

在工作流程上，用户输入查询后，查询分析器先提取关键信息；多路召回模块依据关键信息在文档资源中检索相关数据；重排序模块对检索结果排序过滤；最后大模型根据重排后的信息生成最终答案，高效满足用户信息需求，提升用户体验。

3. 部署使用

RAGFlow 提供了基于 Docker 的轻量化部署方案，支持快速在本地开发环境及在企业生产环境中落地，以下是详细的部署与配置步骤。

1) 启动服务器

为了让 RAGFlow 服务器顺利运行，需要按以下步骤完成服务器启动与基础配置。

(1) 确保 vm. max_map_count 不小于 262144，命令如下：

```
sudo sysctl -w vm.max_map_count=262144
```

这个改动会在下次系统重启时被重置。如果希望做永久改动，则需要在/etc/sysctl. conf 文件里把 vm. max_map_count 的值再相应地更新一遍，在文件中增加 vm. max_map_count＝262144 配置项。

(2) 将仓库克隆到服务器上，命令如下：

```
git clone https://github.com/infiniflow/ragflow.git
```

(3) 进入 docker 文件夹，利用提前编译好的 Docker 镜像启动服务器，使用 CPU 执行嵌入和深度文档相关任务的安装命令如下：

```
docker compose -f docker-compose.yml up -d
```

使用 GPU 加速嵌入和深度文档相关任务的命令如下：

```
docker compose -f docker-compose-gpu.yml up -d
```

如果遇到 Docker 镜像拉不下来的问题，则可以在 docker/.env 文件内根据变量 RAGFLOW_IMAGE 的提示选择华为云或者阿里云的相应镜像。华为云镜像名为 swr.cn-north-4.myhuaweicloud.com/infiniflow/ragflow，阿里云镜像名为 registry.cn-hangzhou.aliyuncs.com/infiniflow/ragflow。

(4) 服务器启动成功后再次确认服务器状态，命令如下：

```
docker logs -f ragflow-server
```

服务器启动后如果出现以下界面提示，则说明服务器启动成功：

```
    ____   ___   ______ ______ __
   / __ \ /   | / ____// ____// ____ _    __
  / /_/ //| | / / __ / /_   / //__ \| | /| / /
 / _, _//___ |/ /_/ //__/  / //_/ /| |/ |/ /
/_/ |_|/_/  |_|\____//_/    /_/ \____/ |__/|__/
Running on all addresses (0.0.0.0)
```

如果在上面的提示信息出来之前，就尝试登录 RAGFlow，则浏览器有可能会提示网络异常。

(5) 在浏览器中输入服务器对应的 IP 地址并登录 RAGFlow，默认的 HTTP 服务器端口为 80。

(6) 在 service_conf.yaml.template 文件的 user_default_llm 栏配置 LLM factory，并在 API_KEY 栏填写和用户选择大模型相对应的 API key。接下来详细介绍具体的系统配置。

2) 系统配置

系统配置涉及以下 3 份文件。

(1) /docker/.env：存放一些基本的系统环境变量，例如 SVR_HTTP_PORT、MYSQL_PASSWORD、MINIO_PASSWORD 等。

(2) /docker/service_conf.yaml.template：配置各类后台服务。

(3) ./docker/docker-compose.yml：系统依赖该文件完成启动。

请务必确保./docker/.env 文件中的变量设置与./docker/service_conf.yaml.template 文件中的配置保持一致。如果需更改默认的 HTTP 服务器端口 80，则可以在./docker/docker-compose.yml 文件中将配置 80:80 改为 YOUR_SERVING_PORT:80。所有系统配置都需要通过系统重启才能生效，命令如下：

```
docker compose -f docker-compose.yml up -d
```

RAGFlow 默认使用 Elasticsearch 存储文本和向量数据，这是由于 Elasticsearch 同时支持文本相似度检索和向量检索，方便做多路召回。

5.5.2 LLaMAIndex 结合 DeepSeek 实践

LLaMAIndex(原 GPT Index)是一款专为大模型设计的开源数据管理框架,核心目标是简化和优化大模型对外部数据的高效管理与查询。传统大模型在处理大规模非结构化数据时存在效率低、数据利用不足等问题,而 LLaMAIndex 通过构建智能索引和增强检索能力,成为连接大模型与外部数据源的桥梁,其核心优势在于支持多样化的数据预处理、灵活的索引构建与大模型的无缝集成,适用于企业知识库、智能客服、文档分析等场景,帮助开发者快速地构建数据驱动的大模型应用。

1. 功能特点

LLaMAIndex 作为先进的 RAG 框架,具备以下核心能力与优势。

1) 多样化索引结构

LLaMAIndex 支持多种索引类型,如向量索引(GPTSimpleVectorIndex)、树状索引(TreeIndex)、图索引等,满足不同数据结构和查询场景的需求,例如,向量索引适用于语义相似性检索,树形索引适合层次化数据组织,图索引则用于处理复杂关系数据。

2) 灵活的数据集成能力

兼容结构化(如数据库)和非结构化数据(如 PDF、文档、API 数据等),通过数据连接器轻松接入多种数据源,支持从本地文件、云端存储到实时 API 的数据读取与解析,实现一站式数据整合。

3) 增强的检索与问答能力

提供基于上下文的智能检索策略,结合语义匹配和上下文增强技术,确保大模型在回答时能精准提取相关信息,提升答案的相关性和准确性。支持多模态检索,适配文本、图像等混合数据场景。

4) 模块化与可扩展性

采用模块化设计,核心组件(如文档读取器、索引引擎、查询模块)可独立替换或扩展。开发者可自定义数据处理流程、索引构建逻辑和检索策略,灵活适配特定业务需求,例如自定义文档解析器或优化检索算法。

5) 快速部署与易用性

作为 Python 库无须额外部署,通过 pip install llama-index 即可快速安装。提供极简 API,入门用户可通过几行代码完成数据导入、索引构建和查询,同时为高级用户开放底层接口,以便进行深度定制。

6) 活跃的开源生态

开源且社区活跃,提供丰富的文档、示例代码和第三方集成(如 LangChain、Hugging Face),支持快速获取技术支持和扩展功能,降低开发门槛。

2. 系统架构

LLaMAIndex 的核心架构由多个模块化组件协同工作,确保数据从摄入到生成响应的高效流转。

1）数据摄入层

文档读取器负责解析原始数据，支持 PDF、Markdown、Excel、HTML 等多种格式，提取文本内容并生成可处理的文档对象，例如，SimpleDirectoryReader 可批量读取文件夹内的文档，WebPageReader 用于爬取网页数据。

2）索引构建层

索引构建层是信息检索系统的核心模块，负责将预处理后的文档数据转换为高效可检索的结构化形式。

（1）索引：核心组件，将解析后的文档转换为高效检索的结构，支持多种索引类型。

（2）向量索引：通过嵌入模型（如 BERT、OpenAI Embeddings）将文本转换为向量，基于余弦相似度检索相关内容。

（3）树状索引：将文档分层组织，适合问答场景中的上下文聚合。

（4）图索引：建模实体关系，处理复杂知识图谱数据。

（5）存储管理器：管理索引的持久化存储，支持内存、磁盘或云端存储，便于数据复用和增量更新。

3）查询处理层

查询处理层是信息检索系统的核心功能模块，负责解析用户请求并高效检索相关文档。该层主要包括以下几个关键组件。

（1）查询引擎：接收用户查询，结合索引逻辑生成检索请求，支持多种查询模式（如关键词搜索、语义检索、上下文增强搜索）。

（2）检索器：从索引中提取最相关的文档片段，支持多路召回（如向量召回 ＋ 关键词召回）和重排序，优化检索结果。

4）大模型集成层

大模型集成层作为系统与大型语言模型交互的核心模块，负责将检索结果与用户查询智能融合，生成高质量回答，该层主要包含以下功能。

（1）大模型接口：无缝对接主流大模型（如 OpenAI、LLaMA、DeepSeek 等），将检索到的上下文与用户问题结合，生成最终回答。支持自定义提示模板和模型参数（如温度、最大 Token 数）。

（2）工具集成：结合外部工具（如计算器、API 调用）增强大模型能力，例如通过 ToolSpec 集成自定义工具处理特定任务。

5）生态扩展层

生态扩展层为系统提供丰富的可扩展能力，支持与各类工具和服务深度集成，该层包含以下几个关键组件：

（1）LLaMAHub：提供 300 多种数据加载器和插件，支持快速接入数据库（如 Pinecone、Chroma）、检索工具（如 Faiss）和第三方服务（如 Web 爬虫、表格解析）。

（2）LLaMALab：实验性项目集合，探索前沿 AGI 场景，如自主任务规划（AutoGPT 风格）和多智能体协作。

通过以上架构，LLaMAIndex 实现了从数据摄入、索引构建到智能查询的全流程闭环，成为大模型应用开发中数据管理与检索的核心工具。

LLaMAIndex 凭借高效索引优化提升长文档处理效率，以极简 API 支持快速低代码接入与深度功能定制，并无缝兼容 LangChain 等生态工具，助力构建端到端 RAG 系统。接下来介绍 LLaMAIndex 如何安装部署及使用。

3. 部署使用

LLaMAIndex 是一个 Python 库，不需要单独部署及启动服务。使用 LLaMAIndex 非常简单，使用 pip install llama-index 命令安装即可，接下来通过代码演示其功能的使用，代码如下：

```
#第 5 章/LlamaIndexDeepSeek.py
import os
from llama_index.core import Settings, VectorStoreIndex, SimpleDirectoryReader,
StorageContext, load_index_from_storage
from llama_index.llms.ollama import Ollama
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
#配置嵌入模型和 DeepSeek 大模型
def configure_models():
    Settings.embed_model = HuggingFaceEmbedding(
        model_name="BAAI/bge-large-zh-v1.5"
    )
    Settings.llm = Ollama(
        base_url="http://localhost:11434",
        model="deepseek-r1:671b",
        temperature=0.7,
        request_timeout=300
    )
#处理索引的创建、加载和更新
def handle_index():
    index_persist_dir = "/home/user/ds_emb"
    data_directory = "/home/user/ds"
    new_data_directory = "/home/user/ds_new"
    if not os.path.exists(index_persist_dir):
        documents = SimpleDirectoryReader(data_directory).load_data()
        index = VectorStoreIndex.from_documents(documents)
        index.storage_context.persist(persist_dir=index_persist_dir)
        print("已创建新索引并持久化存储")
    else:
        storage_context = StorageContext.from_defaults(persist_dir=index_
persist_dir)
        index = load_index_from_storage(storage_context)
        print("已加载现有索引")
    if os.path.exists(new_data_directory):
        new_documents = SimpleDirectoryReader(new_data_directory).load_data()
        if len(new_documents) > 0:
            index.insert_nodes(new_documents)
```

```
            index.storage_context.persist(persist_dir=index_persist_dir)
            print("已更新索引并持久化存储")
    return index

if __name__ == "__main__":
    configure_models()
    index = handle_index()
    query_engine = index.as_query_engine()
    try:
        query_question = "请详细阐述文档中提到的核心方法及其应用场景。"
        answer = query_engine.query(query_question)
        print(answer)
    except Exception as e:
        print(f"查询失败: {str(e)}")
```

这段代码展示了如何使用 LLaMAIndex 库结合 DeepSeek 大模型构建并查询知识库，主要步骤如下。

(1) 模型配置：定义 configure_models 函数，将嵌入模型设置为 BAAI/bge-large-zh-v1.5，大模型使用 OLLaMA 服务的 deepseek-r1:671b，如果服务器资源有限，则可以使用小参数量的模型，例如 deepseek-r1:1.5b。

(2) 索引处理：handle_index 函数负责创建、加载和更新索引，将索引持久化存储在/home/user/ds_emb 目录。

(3) 查询操作：在主程序中，调用上述函数配置模型和处理索引，创建查询引擎并对文档进行查询，输出结果，同时处理可能出现的异常。

5.5.3 GraphRAG

GraphRAG 是微软研究院于 2024 年提出的结构化分层检索增强生成技术，通过将知识图谱与大模型深度融合，解决传统 RAG 在处理复杂关系推理和全局语义理解时的瓶颈，其核心创新在于将非结构化文本转换为知识图谱，利用图结构的层次化社区划分和多跳推理能力，显著提升大模型回答的准确性、完整性和可解释性。

1. GraphRAG 核心原理与技术架构

GraphRAG 的核心原理与技术架构围绕知识图谱驱动的分层检索增强生成展开，通过构建结构化知识网络、分层处理语义信息、融合多维度检索策略，形成从数据解析到智能响应的完整技术闭环，具体包括以下关键模块。

1) 知识图谱构建

知识图谱构建是 GraphRAG 实现结构化知识管理的基础，其核心是通过大模型解析文本语义并构建实体关系网络，具体包含以下两个关键步骤。

(1) 实体与关系提取：使用大模型对文本进行多轮解析，提取实体(如人物、事件、概念)和关系(如“属于”“影响”“包含”)，例如，从“爱因斯坦提出相对论”中提取实体“爱因斯坦”“相对论”，关系“提出”。

(2) 图模型创建：将实体和关系组织成无向加权图，节点代表实体，边权重表示关系强度(如出现频率)，例如，“相对论”与“物理学”通过“属于”关系连接，权重根据文本中出现的次数计算。

2) 社区检测与层次聚类

社区检测与层次聚类是 GraphRAG 将知识图谱转换为层次化语义网络的关键环节，通过图算法实现实体的智能分组与结构分层，具体包括以下内容。

(1) Leiden 算法应用：通过 Leiden 算法(Louvain 算法的优化版本)将图划分为多个社区(子图)，社区内节点连接紧密，社区间连接稀疏，例如，将“物理学”相关实体(如“量子力学”“电磁学”)聚为一个社区，将“生物学”相关实体聚为另一个社区。

(2) 层次化结构生成：自底向上构建多层社区结构，每层代表不同粒度的语义聚合，例如，底层社区包含“牛顿力学”“爱因斯坦相对论”，上层社区聚合为“经典物理”“现代物理”。

3) 社区摘要生成

社区摘要生成是 GraphRAG 实现知识分层提炼与高效检索的关键步骤，通过对不同层级社区的语义浓缩和信息聚合，构建多级知识表示，具体包括以下内容。

(1) 叶级社区处理：对最底层社区(叶级)的实体、关系和声明进行摘要，优先保留高重要性节点(如高连接度实体)，例如，叶级社区“相对论”的摘要包含“质能方程”“时空弯曲”等核心概念。

(2) 层次摘要聚合：从底层到顶层逐层生成社区摘要，高层摘要整合子社区信息，例如，“现代物理”社区摘要包含“相对论”“量子力学”的核心贡献。

4) 检索与生成增强

检索与生成增强是 GraphRAG 依托知识图谱结构优势实现智能问答的核心能力，通过融合层次化社区结构与图关系推理，支持从宏观到微观的多粒度信息检索与复杂逻辑生成，具体包括以下内容。

(1) 全局搜索：针对宏观问题(如“物理学的发展历程”)，通过社区摘要快速定位相关领域，结合层次结构生成全局回答，例如，检索“现代物理”社区摘要，整合“相对论”和“量子力学”的发展脉络。

(2) 本地搜索：针对具体实体问题(如“相对论的核心观点”)，在图中遍历邻居节点，提取关联信息，例如，从“相对论”节点扩展到“质能方程”“时空弯曲”等子节点。

(3) 多跳推理：利用图结构的关系链进行复杂推理，例如，在回答“相对论如何影响核能发展”时，通过“相对论→质能方程→核反应→核能”的路径提取关键信息。

2. GraphRAG 技术优势与应用场景

GraphRAG 通过知识图谱与大模型的深度融合，在技术层面构建了结构化知识处理的核心优势，同时在企业、学术、智能服务等领域形成了具有标杆意义的应用范式。

1) 技术优势

GraphRAG 的技术优势源于其对知识图谱结构化能力的深度利用，通过语义关系建模、层次化知识组织和智能检索策略，在回答质量、计算效率、可解释性及扩展性上实现了对

传统 RAG 的显著超越，具体体现在以下几个方面。

(1) 更高的回答质量：通过知识图谱的结构化信息，GraphRAG 能捕捉实体间的深层关系，避免传统 RAG 的碎片化检索问题，例如，在回答“爱因斯坦与牛顿理论的区别”时，GraphRAG 可通过图结构直接对比两者的核心假设和应用场景。

(2) 更低的计算成本：社区摘要和层次结构减少了大模型处理的 Token 量。微软研究表明，GraphRAG 的 Token 使用量比传统 RAG 减少 26%～97%，显著地降低了成本和延迟。

(3) 增强的可解释性：知识图谱的显式关系链使推理过程透明，例如，回答“核能的原理”时，GraphRAG 可展示“相对论→质能方程→核裂变→能量释放”的推理路径，便于用户理解。

(4) 灵活的扩展性：支持增量更新知识图谱，适配动态数据，例如，添加“量子计算”相关文档后，GraphRAG 可自动将其整合到“现代物理”社区中。

2) 典型应用场景

GraphRAG 凭借其知识图谱驱动的结构化知识处理能力，在企业级知识管理、学术研究辅助、智能问答系统等领域展现出显著的应用价值，典型场景包括以下几种。

(1) 企业知识库：构建内部知识图谱，支持员工快速查询跨部门信息。例如，LinkedIn 通过 GraphRAG 优化客户服务，平均问题解决时间减少 28.6%。

(2) 学术研究：整合文献数据，辅助科研人员发现跨领域关联。例如，在医学研究中，GraphRAG 可关联“疾病→基因→药物”的关系链，加速新药研发。

(3) 智能问答系统：结合多模态数据(如图像、视频)，提供结构化回答。例如，电商场景中，GraphRAG 可关联“商品→品牌→用户评价→促销活动”，生成全面的推荐理由。

3. GraphRAG 开源工具与生态

GraphRAG 依托活跃的开源生态提供了丰富的工具支持，其中 LLaMAIndex 通过 KnowledgeGraphIndex 组件支持从文本自动提取实体、关系并构建知识图谱，实现与大模型的无缝对接；同时兼容 Neo4j、NebulaGraph 等主流图数据库，能够高效地存储和查询大规模知识图谱，为复杂语义推理和多模态数据处理提供底层支撑。

4. 部署使用

GraphRAG 的 Python 版本要求是 Python 3.10～3.12，使用 pip install graphrag 安装，GraphRAG 库包含一个用于无代码入门的 CLI，具体的使用步骤如下。

1) 设置数据项目

创建一个文件夹目录，用来存放要索引的原始文档数据，命令如下：

```
mkdir -p ./ragtest/input
```

测试数据可以从这里下载并存放到./ragtest/input 目录下，命令如下：

```
curl https://www.gutenberg.org/cache/epub/24022/pg24022.txt
-o ./ragtest/input/book.txt
```

2）设置工作区变量

然后需要初始化工作区，运行的命令如下：

```
graphrag init --root ./ragtest
```

这将在./ragtest目录中创建两个文件：.env和settings.yaml。.env包含运行GraphRAG管道所需的环境变量。如果查看该文件，则会看到定义了一个环境变量GRAPHRAG_API_KEY=<API_KEY>，这是OpenAI API或Azure OpenAI端点的API密钥。将其替换为自己的API密钥。在settings.yaml文件中包含管道的设置，可以修改此文件来更改管道设置。

3）创建索引

创建索引使用的命令如下：

```
graphrag index --root ./ragtest
```

此过程需要一些时间，具体取决于输入数据的大小、使用的模型及文本块大小。索引完成后会看到一个名为./ragtest/output的新文件夹，其中包含一系列Parquet文件。

4）运行查询引擎

通过全局搜索提取文档级核心语义，适用于概括性、总结性问题，命令如下：

```
graphrag query \
--root ./ragtest \          #指定工作区根目录
--method global \           #启用全局搜索(基于社区分层结构)
--query "这个故事的核心主题是什么?"
```

通过本地搜索定位特定实体及其关联关系，适用于细节性、关联性问题，命令如下：

```
graphrag query \
--root ./ragtest \          #指定工作区根目录
--method local \            #启用本地搜索(基于实体邻域遍历)
--query "斯克鲁奇是怎样一个人？他与其他主要角色的关系如何?"
```

全局搜索与本地搜索形成互补的检索策略：前者适用于“核心主题”“整体脉络”等宏观问题，通过社区分层结构快速定位跨章节的语义关联；后者则针对“角色关系”“细节解析”等具体问题，通过实体邻域遍历提取精准上下文，两者结合实现从宏观语义概括到微观实体推理的全场景覆盖。

GraphRAG通过知识图谱的结构化优势，为大模型提供了更强大的上下文理解和推理能力，是RAG技术的重要演进方向，其在准确性、效率和可解释性上的突破，将推动大模型在企业级应用、科研和智能服务等领域的落地。

本章深入剖析了检索增强生成技术的核心原理与实践应用。从RAG的技术架构、文本向量模型选择、向量数据库部署到企业级知识问答系统的构建，全面展示了如何通过检索与生成的有效结合，突破大模型的幻觉与知识局限。重点解析了分块向量化、混合搜索索

引、查询转换路由、响应合成等关键技术模块，并对比了微调与 RAG 的优劣势，为不同场景下的技术选型提供指导。通过阿里巴巴 GTE、智源 BGE 等前沿向量模型，结合 Milvus、Pinecone 等向量数据库的实践案例，验证了 RAG 在提升答案准确性、知识实时性与可追溯性上的显著优势。最后，探讨了主流开源 RAG 项目（如 RAGFlow、LLaMAIndex、GraphRAG）的设计理念与部署方案，为开发者提供了高效落地的技术路径。

RAG 技术的成熟为大模型注入了动态知识检索能力，而 AI Agent 的崛起则进一步地推动了大模型向自主化、多任务协作的方向演进。AI Agent 通过整合规划、记忆、工具调用与多模态感知能力，使大模型从被动应答升级为主动执行复杂任务的智能体。第 6 章将系统解析 AI Agent 的核心原理，深入探讨 AutoGPT、MetaGPT、CrewAI 及 OpenManus 通用 AI 智能体等主流框架的技术实现，并揭示其如何与 RAG、工具链、环境交互深度融合，开启下一代智能系统的无限可能。