# 第九章 知识图谱与语义检索增强机制

## 一、语义向量检索与索引构建原理

语义向量（Semantic Vector）检索是知识检索增强生成（RAG）系统的核心组件之一，它通过将文本转换为高维向量空间中的点，使得计算机能够理解和比较文本的语义相似性，从而实现更加精确和相关的信息检索。

### （一）语义向量概念与优势

语义向量是指将文本内容映射到高维向量空间的数值表示，使得语义相似的文本在向量空间中的距离较近，而语义差异大的文本在向量空间中的距离较远[1]。这种表示方法使得计算机能够以数学方式处理和比较文本的语义内容，而不仅仅是基于关键词匹配。

语义向量的核心优势在于：1）**捕捉语义关系**：能够识别同义词、近义词和相关概念，而不仅仅是精确匹配。2）**处理多语言内容**：可以在不同语言之间建立语义联系。 3）**降低维度灾难**：通过将文本映射到固定维度的向量空间，避免了传统词袋模型的高维稀疏问题。4）**支持高效检索**：向量表示支持快速的相似度计算和近似最近邻搜索。

### （二）文本嵌入模型

文本嵌入模型（Text Embedding Model）是生成语义向量的核心技术，它将文本转换为固定维度的数值向量。主流的文本嵌入模型包括：

1. 基于预训练语言模型的嵌入

当前主流的文本嵌入技术不断演进，形成了多个具有代表性的模型体系。**BERT系列**模型通过提取其中间层的上下文表示，捕捉词语间的深层语义关系，尤其是如 Sentence-BERT [2] 这类变体，通过在句子级别训练，使嵌入更加适用于语义匹配与检索场景。**OpenAI 嵌入模型**则推出了如 *text-embedding-ada-002* 的高效模型，该模型在多种语言任务上表现出优异的通用性与计算效率，广泛应用于搜索引擎、推荐系统等嵌入驱动的实际应用中 [3]。此外，**E5 系列模型**由微软提出，通过对提示优化与语义一致性建模，进一步提升了在问答检索、文档匹配等场景中的准确率和鲁棒性，成为当前嵌入技术的强力竞争者之一 [4]。这些模型从不同角度推动了文本语义理解向更深层次发展。

1. 专用嵌入模型

在句子嵌入技术的持续发展中，SimCSE 模型以其创新性的对比学习方法脱颖而出，通过正负样本对构建，显著提升了句子语义表示的一致性与判别能力 [5]。而面向中文及多语言场景，BGE 模型则进行了深入优化，具备优异的跨语言语义对齐能力，尤其适用于中文搜索与问答等任务 [6]。此外，GTE（General Text Embedding）作为一款通用型文本嵌入模型，在多语言和多任务环境中展现出稳定表现，广泛应用于多语种检索、问答系统等领域 [7]。在实际部署中，嵌入模型的选择需综合考量多种因素：包括是否支持多语言、生成向量的维度（通常为384至1536）、模型大小与推理速度对计算资源的要求、在特定领域中的表现差异，以及其所能处理的最大上下文长度等。这些特性直接关系到模型在实际场景中的性能与适配性。

### （三）向量相似度计算

在向量空间中，文本之间的语义相似度通常通过计算向量间的距离或相似度度量来确定。常用的度量方法包括：

1. **余弦相似度（Cosine Similarity）**：测量两个向量之间的夹角余弦值，范围在[-1, 1]之间，值越大表示越相似。
2. **欧氏距离（Euclidean Distance）**：测量向量空间中两点之间的直线距离，值越小表示越相似。
3. **点积（Dot Product）**：在向量已经归一化的情况下，点积等同于余弦相似度，计算效率更高。
4. L1距离（Manhattan Distance）：测量两个向量各维度差的绝对值之和。

在实际应用中，余弦相似度是最常用的度量方法，因为它对向量长度不敏感，能更好地捕捉语义方向的相似性。

### （四）向量索引构建

向量索引是一种数据结构，用于高效存储和检索高维向量。在大规模语义检索系统中，向量索引是实现快速相似度搜索的关键技术。

#### 1. 向量索引的基本原理

传统的线性搜索（暴力搜索）需要计算查询向量与所有存储向量之间的相似度，时间复杂度为O(n)，在大规模数据集上效率低下。向量索引通过预处理和组织向量数据，将搜索复杂度降低到次线性甚至对数级别。

#### 2.主流向量索引算法

在构建大规模语义检索系统时，高效的向量索引机制是保障查询速度与准确率的关键。目前主流的向量索引技术大体可以归类为五类：基于树的方法、哈希方法、量化方法、图结构方法以及混合方法。

对于低维向量（一般小于20维），树结构如KD树、VP树和Ball树表现稳定。KD树通过递归划分坐标轴空间构建二叉树，能够快速定位目标区域；而VP树和Ball树则采用基于距离的空间划分方式，较适合非均匀分布数据。这类方法在高维场景中易受“维度灾难”影响，效率下降明显，因此在实际中更多用于嵌入维度受控的小型系统。

为应对高维向量的近似最近邻检索（ANN）问题，基于哈希和量化的方法应运而生。局部敏感哈希（Locality Sensitive Hashing, LSH）通过构造一组哈希函数，使得相似向量被映射到相同哈希桶中，从而实现子线性时间的检索效率 [8]。与之不同的是，乘积量化（Product Quantization, PQ）将向量拆分为多个子空间，每个子空间分别进行聚类量化，并通过查表还原近似距离。这种方法在保持准确度的同时显著压缩了存储体积 [9]。在此基础上，优化乘积量化（Optimized PQ, OPQ）通过旋转原始向量空间以获得更优的子空间划分效果，进一步提高了压缩精度。

近年来，以图为基础的索引结构获得了广泛关注。典型如分层可导航小世界图（Hierarchical Navigable Small World, HNSW），它构建了多层稀疏图结构，通过贪婪跳跃搜索实现对数级别的查询复杂度和优异的查全率 [10]。其他图结构如NSG（Navigating Spreading-out Graph）优化了连边策略以进一步提高效率；而DiskANN则专门设计用于磁盘环境，在资源受限场景中展现出色表现。

为了兼顾多方面性能，混合方法被广泛采用。例如FAISS系统中集成的IVF+PQ结构，先通过倒排文件（Inverted File Index）缩小候选集合，再对候选项应用量化向量精检；而IVFADC进一步结合了加性距离计算技术，在大规模在线系统中具有极强的实用价值。

#### 综上，不同向量索引策略各有优势，实际应用中需综合考虑数据分布特性、硬件资源限制、搜索精度要求等因素，选择最适合的结构组合，从而在检索性能与系统开销之间取得最优平衡。3.向量索引的性能指标（**表**）

评估向量索引性能的主要指标包括：

表 1 向量索引的性能指标

| 指标 | 含义 |
| --- | --- |
| 查询时间 | 执行一次近似最近邻搜索的平均时间 |
| 召回率 | 近似搜索返回的正确结果占真实最近邻的比例 |
| 内存占用  构建时间  可扩展性 | 索引结构和向量数据的总内存消耗  创建索引所需的时间  随着数据规模增长，性能下降的程度 |

#### 4.向量数据库

向量数据库是专门设计用于存储、索引和查询向量数据的数据库系统。主流的向量数据库包括

表 2 常见向量数据库

| 向量数据库 | 简要描述 |
| --- | --- |
| Milvus | 开源的向量数据库，支持多种索引类型和相似度度量方法[11] |
| Pinecone | 云原生向量数据库，专注于简化向量搜索的部署和管理 |
| Weaviate  Qdrant  Chroma  FAISS | 开源的向量搜索引擎，支持向量和结构化数据的混合查询  高性能的向量相似度搜索引擎，支持过滤和元数据管理  轻量级的向量数据库，专为RAG应用设计  Facebook AI开发的高效相似性搜索库，支持多种索引类型[12] |

### （五）文本分块策略

在构建语义检索系统时，如何将长文本分割成适当大小的块（Chunks）是一个关键问题，它直接影响检索的精度和效率。

#### 1.分块的重要性

适当的分块策略能够：1）确保每个块包含足够的上下文信息，便于理解和回答查询。2）避免块过大导致的相关信息稀释问题。3）优化向量表示的质量，提高检索精度。4）适应嵌入模型的上下文长度限制。

#### 2.主要分块策略

在基于嵌入检索或RAG系统中，原始文本往往需要切分成更小的语义单元，即“文本块”（chunk），以适配嵌入模型输入限制并提升检索效果。分块策略的选择将直接影响到语义表示的完整性、系统的召回性能以及最终生成质量。因此，合理的分块机制是系统设计中的关键一环。

一种常用方式是**固定大小分块**。这类方法按照字符数、词元数（token）或句子数进行截断。例如，将文档划分为每段512字符或256 token 的块，既方便统一处理，又便于与诸如 BERT 或 Ada 等嵌入模型的输入长度对齐。如果任务对语义结构要求较高，按照固定的句子数量分割也常被采用，以保持语言单位的逻辑完整。

然而，固定分块方法在面对主题转换频繁、语义跨度大的文本时可能产生“断裂”问题，影响上下文理解。为此，引入**语义感知分块策略**可以有效缓解该问题。这类方法依据内容本身的结构与语义信息进行划分，例如按自然段（Paragraph Chunking）切分，可保留文档原始段落逻辑；也可通过主题检测或句间语义相似度分析，识别语义断点，进而完成更精准的语义块生成。

为了进一步提升上下文保留能力，许多系统采用了重叠分块（Overlapping Chunking）技术。最典型的是滑动窗口方法，即每个文本块包含部分前一个块的末尾内容，从而缓解模型“遗忘前文”问题。更为复杂的分层重叠策略还支持构造不同粒度的块，满足快速粗略过滤与细致语义比对的双重需求。

此外，当前也有研究尝试融合多种策略，形成**混合型分块方法**。例如，自适应分块会根据文本密度、长度与复杂度自动调整块大小，而结构感知分块则结合文档的格式特征（如标题、编号、列表、表格）进行划分，更贴近实际文档组织方式。这种策略尤其适用于企业文档、法律条文、技术手册等结构清晰的资料。

分块策略的选择需综合考虑任务类型、嵌入模型的输入上限、系统召回效率与语义完整性要求。固定分块实现简单且适配性强，适合快速构建原型；而语义感知与重叠分块则在注重效果与精度的实际系统中更具优势。未来分块技术的发展趋势将向更智能、结构化与自适应方向演进，助力构建高质量的语义索引系统。

#### 3.分块策略选择考虑的因素

选择合适的分块策略应考虑以下因素：1）**文档类型**：不同类型的文档（如学术论文、网页、代码、对话记录等）可能需要不同的分块策略。2）**查询特性**：预期的查询类型（如事实查询、概念解释、多步骤推理等）会影响最佳分块大小。3）**嵌入模型**：不同的嵌入模型对输入长度有不同的限制和最佳性能点。4）**计算资源**：更细粒度的分块会增加索引大小和检索计算量。5）**领域特性**：特定领域的文本可能有特殊的结构或语义特点需要考虑。

### （六）查询重写与扩展技术

在语义检索系统中，用户的原始查询往往存在表述不清、意图不明、词汇偏差等问题，这会显著降低系统召回相关文档的能力。为了解决这一问题，查询重写（Query Reformulation）与查询扩展（Query Expansion）技术应运而生，旨在对用户输入的查询进行优化，使其更精确地反映信息需求并提升匹配效果。两者虽然目标一致，但侧重点略有不同：重写强调对原始查询的结构与表述进行调整，而扩展则注重在原查询基础上引入更多语义相关的信息。

**查询重写技术**主要包括基于规则与基于模型两大类方法。传统的**基于规则的重写**通过自然语言处理技术对原始查询进行规范化处理。例如，停用词移除可以去除“是”“的”“了”等语义负载较轻的功能词，从而减少噪声；词形还原（Lemmatization）则将不同词形统一为词根，如将“running”还原为“run”；同义词替换也被广泛用于增加表达多样性，缓解词汇不匹配问题。这些操作虽然简洁高效，但难以理解复杂语境中的用户意图。

随着大型语言模型（LLMs）的兴起，**基于模型的查询重写**技术成为主流。这类方法可将自然语言查询自动转换为更适合检索系统处理的结构化或语义丰富表达。例如，GPT 系列模型已被用于生成结构化的检索提示语，或将用户提出的问题改写为更标准的检索问题，从而提升检索召回率[13]。此外，借助模型的上下文理解能力，还可识别查询背后的潜在意图（如“比较价格”“查找位置”等），并据此调整或扩展查询表达，实现更贴合需求的重构。

在与查询重写互补的方向上，**查询扩展技术**通过为原始查询增添语义相近或语义关联的关键词，进一步扩大检索范围。较早期的**基于知识的扩展方法**主要依赖词典、知识图谱或本体库来提供同义词、上下位词（hypernyms/hyponyms）、相关实体等。例如，对查询“苹果”扩展为“水果”“iPhone”“MacBook”等能显著提升系统对不同语义背景的覆盖能力。

另一类重要方法是**基于反馈的扩展**。例如伪相关反馈（Pseudo Relevance Feedback）技术会自动使用初次检索结果中排名靠前的文档中的关键词来扩展查询内容，假设这些文档是相关的；而用户交互反馈则更进一步，通过用户对结果点击、评价等行为进行显式或隐式学习，实现更精准的扩展策略。

近年来，随着生成式模型的发展，**基于生成的扩展方法**也逐渐成熟。例如 HyDE（Hypothetical Document Embedding）方法通过生成一篇与查询语义相关的“假想文档”，并将其向量表示用于检索，从而间接扩展查询语义空间[14]。此外，多角度查询生成（Multi-perspective Query Generation）也在实际应用中展现出强大能力，它可根据不同表达视角生成多个等价但词汇多样的查询，提升系统对表达变化的鲁棒性。查询重写与扩展是构建高性能语义检索系统不可或缺的组成部分。它们不仅改善了用户输入的表达质量，还通过拓展语义空间增强了系统的匹配能力，为构建鲁棒、高召回、强理解能力的现代语义搜索引擎奠定了坚实基础。

### （七）检索策略优化

为了显著提升RAG系统的检索表现，现代语义检索正在从“单通道、粗匹配”的旧范式，迅速演进为融合多策略、多阶段、个性化的智能协同优化体系。**混合检索**、**多阶段排序**与**上下文感知机制**，正成为引领RAG性能跃迁的三大核心方法。

在**混合检索**方面，不同模型之间的协同融合为系统注入了更强的表达能力。例如，将 BERT、E5 或 OpenAI 嵌入模型的结果进行加权组合，不仅能够互补模型间的盲点，也能覆盖更广泛的语义维度。此外，通过将句子级、段落级、甚至文档级的多粒度嵌入结果进行联动，检索系统可在不同语义层次上精准捕捉用户意图。同时，稀疏-密集检索的联动成为标配：如 BM25 优势在于对关键词的精准识别，而向量检索擅长识别语义近似，两者协同可显著提升召回率与相关性[15]。

**多阶段检索方法采用**“粗排+精排”的两步策略最为典型。第一阶段采用高效但粗糙的算法快速筛选候选文档，第二阶段则利用 Cross-Encoder 或 Reranker 进行精细排序，实现速度与质量两个维度同时提升。更高级的**级联检索**逐层缩小搜索范围，让信息获取更聚焦、更高效。若进一步融入用户反馈信号，系统可形成**迭代式优化闭环**，实现“边查边学”的自适应演化。

**上下文感知检索**则打开了“以用户为中心”的新维度。通过引入对话历史，系统可精准理解用户查询的前后语境；通过建模用户偏好与行为轨迹，系统能自动个性化排序；若配合任务感知机制，系统还能识别用户当前处于调试代码、查阅法规或撰写报告等具体任务场景，进而自动切换检索策略和模型架构，真正实现“因人而异、因事制宜”的智能匹配。检索策略优化正引领RAG系统从“强工具”迈向“智能助手”，其系统性演进将成为构建高性能、强适配、人机共融的下一代智能检索体系的核心驱动力。

## 二、知识图谱概增强技术

### （一）知识图谱概念与特征

随着大语言模型能力不断跃升，如何有效引入结构化知识成为提升其可控性、事实一致性和推理深度的关键课题。知识图谱（Knowledge Graph, KG）正是在这一语义增强需求中脱颖而出的重要支撑形式。它不是简单的数据罗列，而是一种以“图”为组织结构，将实体、概念及其关系进行系统建模的知识表达方式。

与传统数据库相比，知识图谱具有鲜明的语义驱动特征。在知识图谱中，每一个节点都代表着一个具备独立语义的实体（如“阿尔伯特·爱因斯坦”），边则表示实体之间的语义关系（如“出生于”），从而构建起一个支持逻辑推理的“语义网络”[17]。这样的结构不仅存储数据，更强调知识之间的联系，使得机器不只是“记住”，而是“理解”。

知识图谱最显著的优势在于其语义明确性。实体和关系均有清晰的语义定义，例如“创立于”与“任职于”尽管在语言表层接近，但在图谱中对应完全不同的关系标签，可显著减少模型理解歧义。同时，知识图谱还具备高度结构灵活性，支持多层级嵌套、多关系组合，能适应从金融到医疗等多种复杂知识体系的建模需求。

在表达能力上，图谱所构建的语义连接极其丰富而精准。它不仅描述事实，还保留关系间的路径结构，支持路径推理、多跳查询等高级任务。例如，通过图结构模型，可以自动发现“爱因斯坦 → 毕业于 → 苏黎世联邦理工学院 → 位于 → 瑞士”这一路径，并据此完成复杂的知识查询与补全。

此外，知识图谱的可扩展性也为其在动态知识更新中的应用提供了便利。新的实体、关系可以随时无缝加入图谱结构中，不影响原有数据一致性。而更为关键的是，它基于连接结构的构造，天然支持逻辑推理与知识发现。结合规则系统或嵌入模型，图谱能够在现有数据上生成新的推断知识，为语言模型提供超越明文语料的“隐含认知”。

### （二）知识图谱基本组成（图和**表**）

知识图谱的基本组成元素包括：

1. **实体（Entity）**：知识图谱中的节点，表示现实世界中的对象或概念，如人物、组织、地点、事件等。每个实体通常有一个唯一标识符和一组属性。
2. **关系（Relation）**：知识图谱中的边，表示实体之间的语义关联，如”出生于”、“工作于”、“创立了”等。关系通常是有向的，具有特定的语义类型。
3. **属性（Attribute）**：实体的特征或特性，如人物的”出生日期”、“职业”等。属性可以看作是实体与文字值之间的特殊关系。
4. **三元组（Triple）**：知识图谱的基本表示单元，由主体（Subject）、谓词（Predicate）和客体（Object）组成，形式为(s, p, o)，表示”主体s通过谓词p与客体o相关联”。例如，(爱因斯坦, 出生于, 德国)。

表 3 知识图谱的基本组成元素

| 元素 | 说明 |
| --- | --- |
| 实体 | 表示现实世界中的对象或概念，带唯一标识符和属性 |
| 关系 | 实体之间的有向语义关联，如“出生于”“工作于”等 |
| 属性 | 实体的特征或特性，连接实体与文字值的特殊关系 |

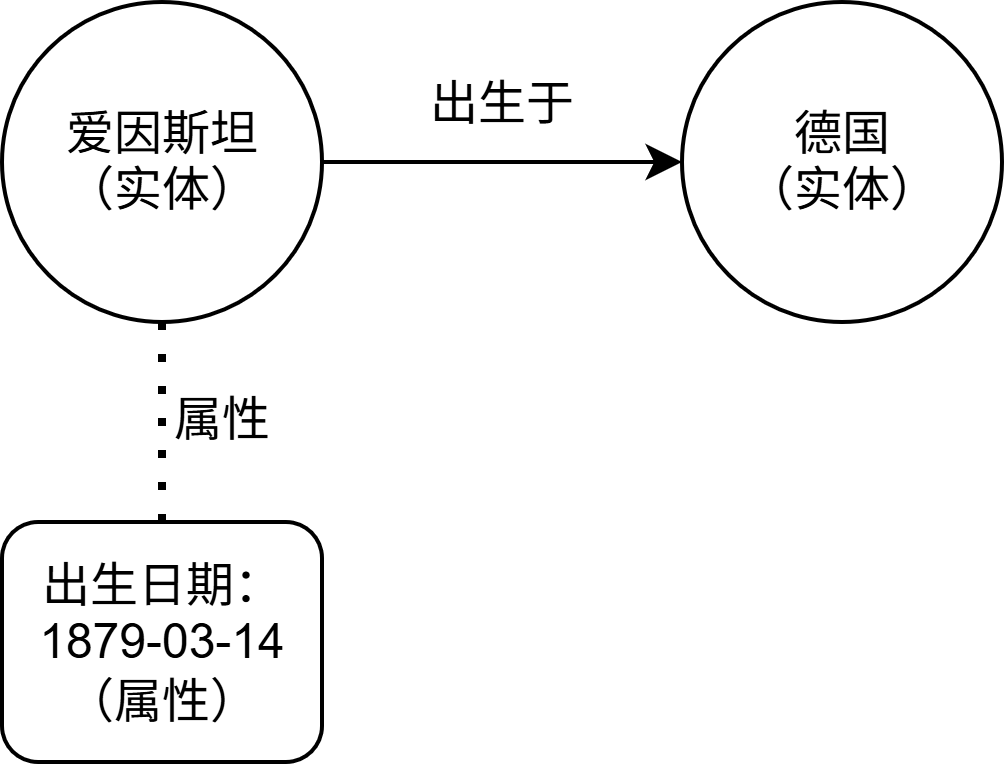


图 1 知识图谱实例

### （三）知识图谱的分类（**表**）

根据覆盖范围和应用领域，知识图谱可以分为以下几类：

1. **通用知识图谱**：覆盖广泛领域的百科全书式知识图谱，如Google Knowledge Graph、Wikidata、DBpedia等。
2. **领域知识图谱**：专注于特定领域的知识图谱，如医学知识图谱（UMLS）、金融知识图谱、法律知识图谱等。
3. **企业知识图谱**：组织内部的知识资产图谱，整合企业内部的数据、文档、专业知识等。
4. **个人知识图谱**：个人知识管理和学习的工具，记录个人的知识、经验和关联。

表 4 知识图谱的分类

| 知识图谱类别 | 覆盖范围 | 应用领域 |
| --- | --- | --- |
| 通用知识图谱 | 广泛领域 | 百科全书式，如Google Knowledge Graph、Wikidata、DBpedia |
| 领域知识图谱 | 特定领域 | 医学（UMLS）、金融、法律等专业领域 |
| 企业知识图谱  个人知识图谱 | 组织内部  个人知识管理 | 企业内部数据、文档、专业知识整合  个人知识、经验及关联记录 |

### （四）知识图谱的表示方法（**图**）

在构建和应用知识图谱的过程中，如何表示图谱中的数据，直接影响其可存储性、可查询性和可推理能力。目前主流的表示方法涵盖语义网标准、图数据库模型、形式逻辑语言以及面向机器学习的向量化方式，每种方法在不同场景下各有优势。

1. RDF（Resource Description Framework）表示法

RDF 是 W3C 推荐的知识表达框架，以三元组 (主语, 谓语, 宾语) 的形式捕捉事实[18]。每个元素均为 URI，可跨平台共享。比如：

* <http://example.org/Albert\_Einstein> <http://example.org/bornIn> <http://example.org/Germany> .

这意味着“爱因斯坦出生于德国”，且每个元素可进一步链接其他语义资源，构建成全球可查询的知识网络。

1. 属性图模型（Property Graph Model）

属性图强调节点和边的“可带属性性”，是图数据库（如Neo4j、TigerGraph）最常见的数据模型。每个节点可拥有键值对属性，每条边不仅有方向和类型，也可携带时间戳、权重等上下文信息。例如：

(Albert\_Einstein {name: "Albert Einstein", birth\_date: "1879-03-14"})

-[BORN\_IN {year: 1879}]->

(Germany {name: "Germany", population: 83000000})

该结构适合企业级图数据库中的复杂查询与分析。

1. OWL（Web Ontology Language）本体表示

OWL 是面向语义推理的本体描述语言，基于描述逻辑，可定义概念的层级、关系的约束以及规则推理。如下所示：

<owl:Class rdf:ID="Person"/>

<owl:Class rdf:ID="Country"/>

<owl:ObjectProperty rdf:ID="bornIn">

<rdfs:domain rdf:resource="#Person"/>

<rdfs:range rdf:resource="#Country"/>

</owl:ObjectProperty>

<owl:NamedIndividual rdf:ID="Albert\_Einstein">

<rdf:type rdf:resource="#Person"/>

<bornIn rdf:resource="#Germany"/>

* </owl:NamedIndividual>

这种方式为后续语义验证和一致性检查提供了形式化基础。

（4）向量化表示（Knowledge Embedding）

为适配深度学习模型，近年来兴起了将知识图谱中的实体与关系嵌入低维向量空间的技术，如 TransE、RotatE、ComplEx 等。以 TransE 为例，它试图满足如下向量关系：

* E(爱因斯坦) + E(出生于) ≈ E(德国)

这种方式不仅便于知识补全和链接预测，还能与语言模型无缝集成，支持知识驱动的神经推理。

综上，不同的表示方法服务于不同的下游目标：RDF 和 OWL 强调可解释性和标准互通，属性图注重工程效率和灵活性，而向量表示则成为大模型语义融合与知识注入的重要桥梁[18]。

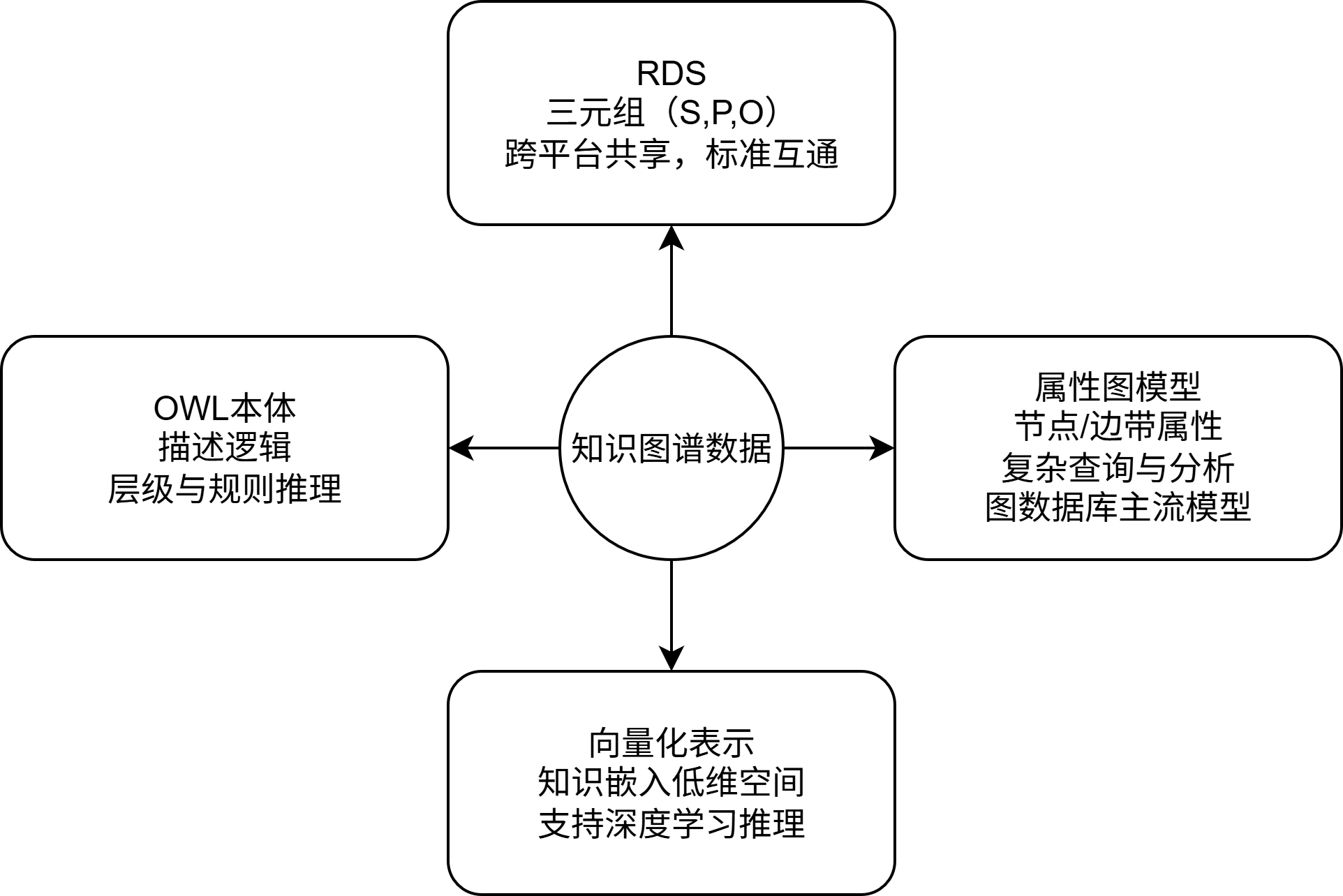


图 2 知识图谱的表示方法

### （五）知识图谱与大型语言模型的结合

尽管大语言模型（LLM）具备强大的语言理解和生成能力，但它们仍面临几个关键挑战，例如知识滞后、推理能力弱、输出解释性不足等。知识图谱的引入，正是为了解决这些限制，为语言模型注入结构化、可控、可验证的外部知识支持，构建更智能、更可信赖的智能体系统。

（1）**增强预训练语义基础。**此阶段引入知识图谱，可以显著增强模型对概念、实体及其关系的掌握程度。一些研究采用三元组文本重构、自监督图游走等方法，将知识图谱内容编码进语言模型的语义空间，使其具备一定的结构化认知能力。这种融合不仅提升了命名实体识别、关系抽取等任务的准确性，还为多跳推理提供了语义基础[18]。

（2）**增强推理阶段准确性**。这个过程中的知识图谱可作为检索增强模块介入。当用户发起一个包含实体的查询时，系统可从图谱中快速提取与之相关的子图结构，作为额外上下文送入LLM生成模块。这种“查询驱动的知识注入”显著提升了对事实性问题的回答准确率，尤其在医疗、金融、法律等对准确性要求极高的领域表现尤为突出。

（3）**增强显式推理能力。**知识图谱的逻辑结构还可实现显式推理能力的增强。通过图谱路径分析、关系规则挖掘或使用本体约束，系统可对模型生成内容进行“结构上的补全与验证”，弥补语言模型仅凭概率机制生成所带来的推理漏洞。特别是在多跳问题求解和因果链路构建等任务中，图谱的路径推理机制能有效提升响应的深度与可信度。

（4）**提升事实验证功能。**事实验证功能成为图谱协同的一项核心价值。由于语言模型存在“幻觉”风险，即生成内容貌似合理却与事实相悖，通过对生成结果与图谱中已知事实的比对，可以实现对输出内容的逻辑校验和纠错。这一过程不仅提升模型输出的质量，也为其在关键场景中的应用提供了安全保障。

（5）**促进了可解释性能力**。知识图谱为语言模型提供了重要的解释性支持。在很多任务中，用户并不满足于“是什么”，更想知道“为什么”。图谱可以作为“答案的来源轨迹”，以可视化路径或实体关系链的形式解释模型的判断依据，从而增强用户对智能系统输出的理解与信任。

综上，知识图谱不只是知识的存储容器，更是大型语言模型能力延展的“外脑”。从语义注入、结构对齐到解释支撑，它正在成为构建高性能、可控型语言智能系统的关键技术路径[18]。

## 三、本体和实体构建原理

在构建知识图谱时，如何确保图中的知识不仅结构清晰，还能被一致、准确地解释与推理，是一项核心挑战。这正是“本体”（Ontology）发挥关键作用的场景。本体是对某一领域中核心概念、关系及其语义约束的形式化定义，它为知识图谱提供了统一的语义框架和逻辑基础[19]。

### （一）本体定义与作用

可以将本体理解为知识图谱的“语法规则与字典”，它明确规定了哪些实体可以被纳入图谱、它们之间可以建立哪些类型的语义关系，以及这些关系需要遵守哪些逻辑约束。例如，在医学领域中，“医生”可以“诊断”某种“疾病”，但“医院”不能“发病”——本体通过类型与关系的约束机制确保了这类语义的正确性。

具体来看，本体为知识图谱带来了四项重要能力：

（1）构建了**共享的词汇体系**。在任何知识建模任务中，最棘手的问题之一是不同系统、不同人员使用术语不一致。本体通过标准化术语及其定义，使得不同模块、团队甚至平台可以围绕相同的语义基础进行协同建模与信息交互。

（2）提供了**强约束的语义规范**。实体与关系在知识图谱中并非随意连接，必须符合本体中定义的结构规则，例如某类关系是否必须有方向、是否可以多重赋值、是否具备对称或传递属性等。这样的规范性确保了知识图谱在扩展过程中的一致性与鲁棒性。

（3）**知识推理的基础设施**。通过本体中定义的继承体系（如“心脏病”是“疾病”的子类），以及逻辑规则（如“每个医生只能执业于一个注册地区”），系统可以在已有知识的基础上进行自动补全、错误检测、因果分析等推理任务，大幅提升了知识图谱的智能性。

最后，得益于其形式化与标准化特点，本体极大地**促进了跨系统间的知识共享与融合**。无论是联邦图谱建设，还是企业间的数据对接，只要遵循共同本体规范，原本异构的知识就能在语义层面达成协同，形成更大范围的知识互操作网络。

因此，本体不仅是知识图谱的结构骨架，更是确保其语义正确性、智能行为与系统互联能力的核心引擎[19]。

### （二）本体组成要素（**图**）

一个完整的本体通常包含以下要素：

1. **类（Class）**：概念的集合，如”人物”、“组织”、“地点”等。类可以组织成层次结构，形成分类体系。
2. 属性（Property）：
   * **对象属性（Object Property）**：连接两个实体的关系，如”工作于”、“位于”等。
   * **数据属性（Data Property）**：连接实体和数据值的关系，如”年龄”、“成立时间”等。
3. **实例（Instance）**：类的具体成员，如”爱因斯坦”是”科学家”类的一个实例。
4. **公理（Axiom）**：对类和属性的约束规则，如属性的定义域和值域、类的等价关系、属性的特性（如传递性、对称性）等。
5. **注释（Annotation）**：对本体元素的说明信息，如标签、注释、定义等。

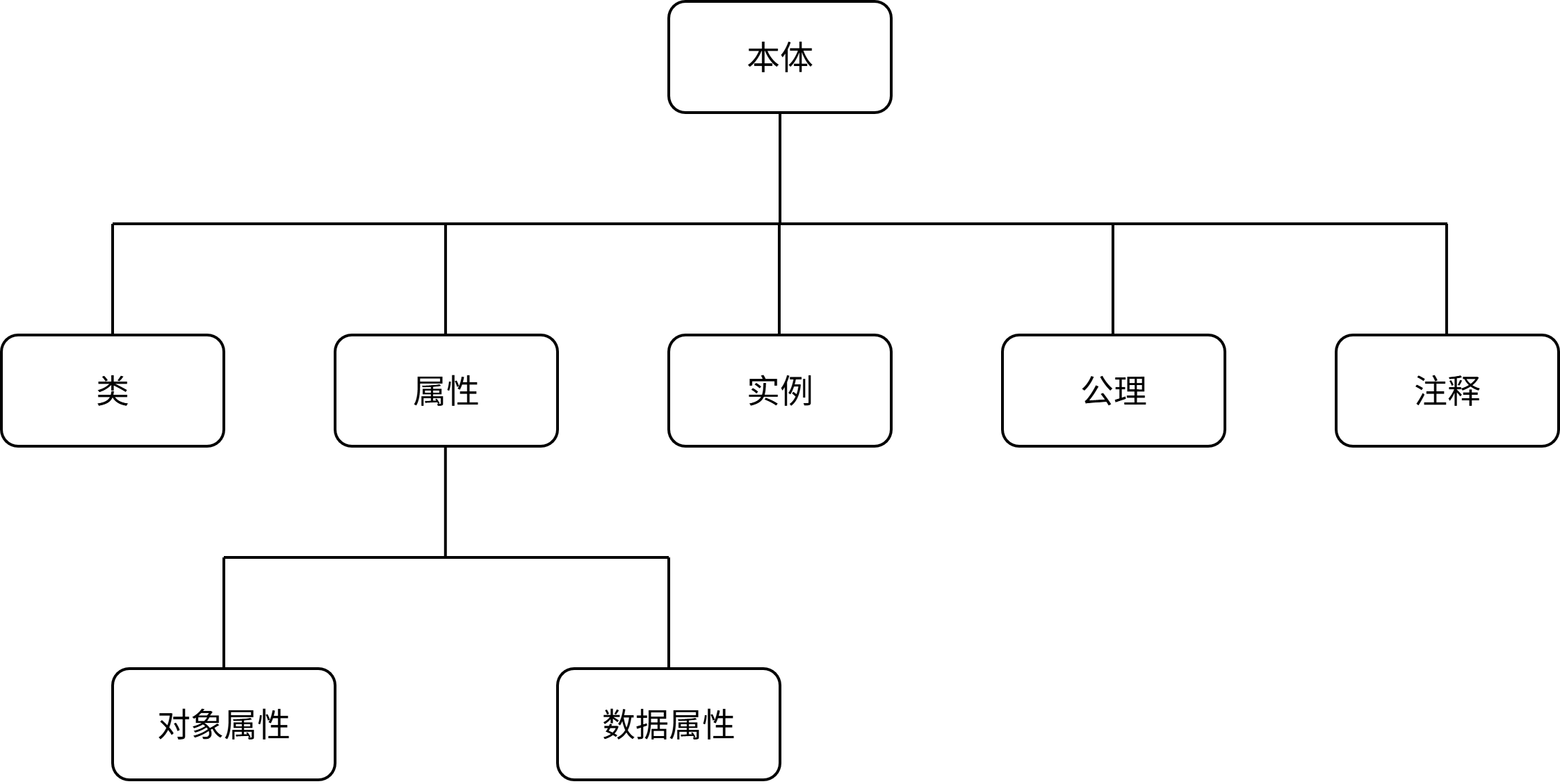


图 3 本体组成要素

### （三）本体构建方法（**图**）

本体构建是一个复杂的工程过程，通常包括以下步骤和方法：

1. **需求分析**：明确本体的目的、范围和用户需求。
2. **知识获取**：从各种来源收集领域知识，包括：
   * 专家访谈和头脑风暴
   * 文献研究和教材分析
   * 现有本体和词汇表的重用
   * 数据挖掘和文本分析
3. **概念化**：识别和组织领域的核心概念和关系。
   * 概念聚类和分类
   * 关系识别和定义
   * 层次结构构建
4. **形式化**：使用形式化语言（如OWL）表示本体。
   * 类和属性的定义
   * 约束和规则的表达
   * 实例的添加
5. **评估与优化**：评估本体的质量并进行优化。
   * 一致性检查
   * 完整性评估
   * 可用性测试
6. **维护与更新**：随着领域知识的发展，持续更新和完善本体。

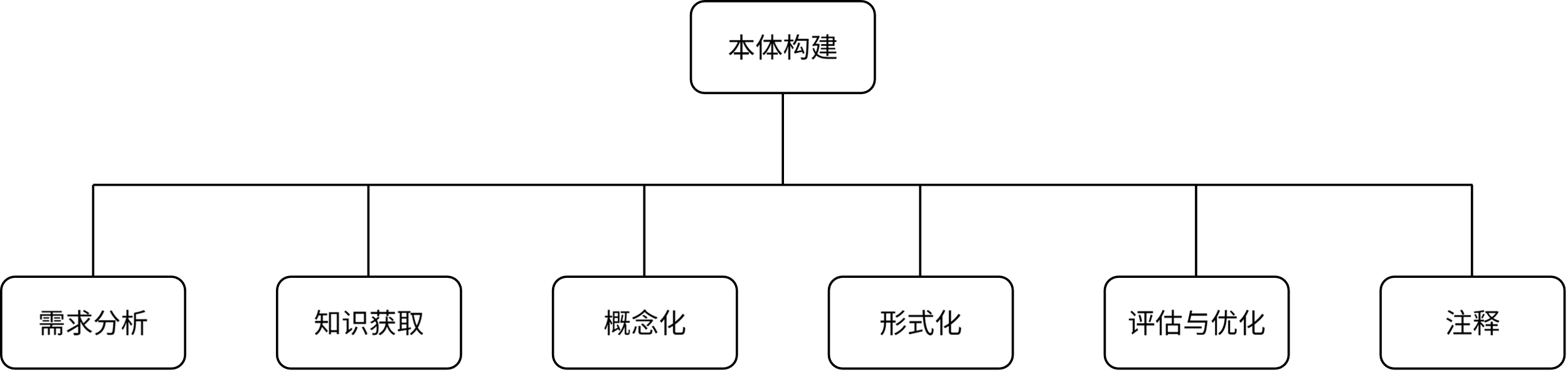


图 4 本体构建方法

### （四）本体构建工具（**表**）

常用的本体构建工具包括：

表 5 常用的本体构建工具

| 工具名称 | 开发者/类型 | 主要功能 |
| --- | --- | --- |
| Protégé | 斯坦福大学，开源 | 支持OWL和RDF，可视化编辑和推理功能 |
| WebProtégé | Protégé的Web版本 | 支持协作编辑和版本控制 |
| TopBraid Composer  NeOn Toolkit  OWLGrEd | 商业软件  开源平台  图形化编辑器 | 支持RDFS、OWL和SHACL，高级建模与验证  支持分布式本体开发  UML风格可视化OWL本体编辑 |

### （五）实体构建与知识抽取

实体构建是将领域中的具体对象识别并添加到知识图谱中的过程，通常涉及知识抽取技术。

#### 1.知识抽取的主要方法

1. 规则基础方法：
   * **模式匹配**：使用预定义的语言模式识别实体和关系。
   * **正则表达式**：使用正则表达式匹配特定格式的实体（如日期、邮箱等）。
   * **词典匹配**：使用预定义的实体词典进行匹配。
2. 机器学习方法：
   * **命名实体识别（NER）**：识别文本中的实体及其类型，如人名、地名、组织名等[21]。
   * **关系抽取（RE）**：识别实体之间的语义关系。
   * **事件抽取（EE）**：识别文本中描述的事件及其参与者。
3. 深度学习方法：
   * **BERT/RoBERTa等预训练模型**：利用预训练语言模型进行实体和关系抽取。
   * **图神经网络（GNN）**：利用图结构信息进行知识抽取和补全。
   * **远程监督学习**：利用已有知识库自动标注训练数据。
4. 大型语言模型方法：
   * **提示工程**：设计特定提示，引导大型语言模型抽取结构化知识。
   * **少样本学习**：通过少量示例，使大型语言模型学习抽取模式。
   * **自我验证**：利用模型的推理能力验证抽取结果的一致性。

#### 2.实体链接与消歧

实体链接（Entity Linking）是将文本中提及的实体与知识库中的实体条目关联起来的过程，它通常包括以下步骤：

1. **实体提及检测**：识别文本中可能指代实体的片段。
2. **候选实体生成**：为每个实体提及生成可能对应的知识库实体列表。
3. **实体消歧**：从候选实体中选择最匹配的一个，解决同名异义（一个名称对应多个实体）的问题。
4. **新实体发现**：识别知识库中不存在的新实体，并决定是否添加到知识库中。

实体消歧的主要方法包括： - **上下文相似度**：比较实体提及的上下文与候选实体的描述信息。 - **图结构信息**：利用知识图谱的结构信息进行消歧。 - **集体消歧**：同时考虑文档中所有实体提及，寻求全局最优解。 - **深度学习方法**：使用神经网络模型学习实体表示和匹配函数。

### （六）知识融合与质量控制

在构建大规模知识图谱时，通常需要整合多个来源的知识，这就涉及知识融合和质量控制问题。

#### 1.知识融合方法

1. **实体对齐**：识别不同来源中指代同一实体的记录，并将它们合并。
   * 基于属性相似度的方法
   * 基于结构相似度的方法
   * 基于嵌入的方法
2. **冲突解决**：处理不同来源提供的冲突信息。
   * 基于可信度的方法（选择更可靠来源的信息）
   * 基于时间的方法（选择最新的信息）
   * 基于多数投票的方法
   * 保留所有信息并标注来源
3. **知识补全**：填补知识图谱中的缺失信息。
   * 基于规则的推理
   * 基于统计的推理
   * 知识图谱嵌入方法（如TransE、RotatE等）

#### 2.质量控制方法

1. **一致性检查**：验证知识是否符合预定义的约束和规则。
   * 类型约束检查
   * 基数约束检查
   * 逻辑一致性检查
2. **完整性评估**：评估知识图谱的覆盖范围和完整程度。
   * 实体覆盖率
   * 关系覆盖率
   * 属性完整度
3. **准确性验证**：验证知识的事实准确性。
   * 基于外部权威来源的验证
   * 基于众包的验证
   * 基于统计模式的异常检测
4. **可用性评估**：评估知识图谱在实际应用中的表现。
   * 查询性能
   * 推理能力
   * 应用场景适配度

## 四、LlamaIndex框架与语义检索流程集成

大语言模型（LLMs）应用开发中如何高效利用私有知识成为关键挑战。LlamaIndex是一个专为LLM应用设计的数据框架，其核心使命在于赋能开发者，将多样化的自定义数据源与强大的LLM能力无缝集成[22]。这解决了如何让模型精准“理解”和“调用”非公开信息的核心问题。LlamaIndex 提供了一套强大的工具集与模块化组件。开发者利用这些工具，能够系统性地设计和构建复杂的检索增强生成（RAG）系统[22]。这类系统通过智能检索相关上下文信息并输入给LLM，显著提升了生成内容的准确性、相关性和知识深度，是提升LLMs在专业领域应用潜力的关键技术架构。

### （一） LlamaIndex的核心概念与架构（**图**）

在大语言模型（LLM）构建信息检索增强系统（Retrieval-Augmented Generation, RAG）的实践中，**LlamaIndex** 提供了一种结构清晰、模块化且可扩展的框架。它不仅解决了大模型如何与外部知识结合的问题，还通过“索引-检索-生成”的流程，提升了模型在复杂查询任务中的响应能力。LlamaIndex 的系统架构可分为若干核心组件，每个组件在信息流中承担特定职责，共同构建出完整的查询增强机制。下图展示了 LlamaIndex 系统的逻辑架构：

✅【画图建议】：绘制结构图，突出“数据输入 → 节点分割 → 索引构建 → 信息检索 → 查询生成”的五级数据流，并展示各模块之间的调用关系。

（1）文档（Documents）：知识的原始载体

系统接收的输入可以是多种格式的文档，例如 .txt 文件、网页内容、数据库记录、甚至是 PDF、Word 或 Markdown 文件。这些原始数据通过 **数据连接器** 接入系统，成为后续处理的起点。

（2）节点（Nodes）：被分割的知识单元

文档在被加载后，首先会被**解析并分割为多个节点（Node）**。每个节点可以视作一个语义块，通常是一个段落、一句话，或一个逻辑完整的小节。节点是系统内部构建索引、进行向量化与检索操作的基本单位。

（3）索引（Indices）：构建高效检索的数据结构

LlamaIndex 支持多种索引类型（如向量索引、树形索引、关键词倒排索引等），这些索引结构根据节点信息建立，以支持后续的高效查询与过滤。例如，向量索引利用嵌入模型将节点转为高维向量，实现语义层面的相似性搜索。

（4）检索器（Retrievers）：桥接索引与查询的调度器

在用户提出查询后，检索器负责从构建好的索引中选择与查询最相关的节点。检索器可以按关键词、向量距离、主题聚类等方式对索引内容进行召回，结果被用于支持后续的大模型响应。

（5）查询引擎（Query Engine）：集成推理与回答生成的中枢

查询引擎将检索器返回的相关节点输入到大语言模型中，以生成回答、摘要或报告。该引擎不仅仅是执行“检索+拼接”的逻辑，往往还集成了提示工程、上下文压缩、多跳检索等机制，使生成结果更加准确与有逻辑性。

（6）数据连接器（Data Connectors）：外部世界的接口桥梁

数据连接器是 LlamaIndex 架构的输入层。它们支持从各种数据源中抓取信息，包括本地文件系统、Web API、SQL 数据库、Notion 页面、Google Drive 文件等。这一机制保证了系统的可扩展性与场景适应能力。

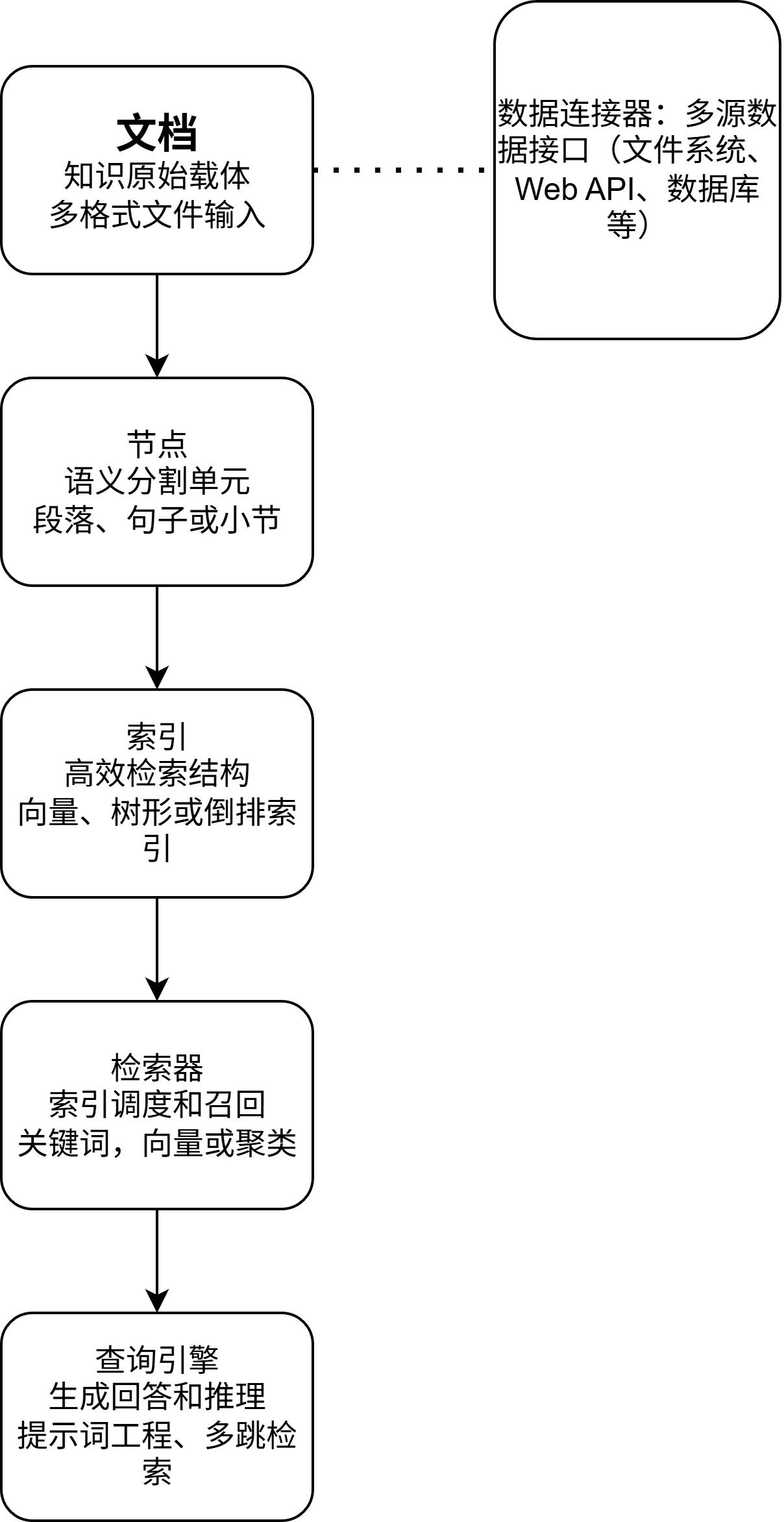


图 5 LlamaIndex架构

### （二）LlamaIndex的主要索引类型

LlamaIndex 提供了多种索引类型，以适配不同规模、结构和需求的数据检索场景。这些索引在内部处理机制和适用任务上各有侧重，构成了该框架灵活且高效的信息检索能力基础。

最为常用的是 **向量存储索引（Vector Store Index）**。该类型基于向量相似度构建，将文档分割后的每个节点转化为语义向量，并存储在向量数据库中（如 FAISS、Milvus 等）。在查询时，用户的问题也被转化为向量，通过向量空间中相似度计算，实现语义上的模糊匹配与智能召回，特别适用于开放性问答与语义搜索任务。

对于小型数据集或对顺序处理有要求的应用，**列表索引（List Index）** 是比较简单而直接的方案。它将所有节点按顺序存储在列表中，在查询阶段逐个比较节点内容，可顺序处理也可并行执行，适用于教学演示、文档校对等场景。

当面对结构层级分明或逻辑嵌套较强的文档，如多级目录资料、法规文件或技术手册时，**树索引（Tree Index）** 显得尤为合适。该方法将节点组织为树形结构，每一层的父节点都是其子节点内容的摘要。查询时从根节点出发，逐层筛选最相关的子树，可显著压缩上下文长度，是构建多轮摘要式问答系统的理想选择。

在需要高精度文本定位的场合，**关键词表索引（Keyword Table Index）** 提供了传统的倒排索引机制。它将关键词映射至包含该关键词的所有节点，支持用户输入的关键词直接匹配文档片段，适用于法务文档、政策法规、产品说明等对词汇匹配要求较高的检索任务。

**知识图谱索引（Knowledge Graph Index）** 非常适合企业知识管理、科研文献关联分析等高阶应用场景。将文档中的实体与关系提取出来，构建结构化的图数据库。查询时不仅能通过图结构遍历找到相关实体，还支持基于图结构的逻辑推理与上下位概念扩展。

最后，**文档摘要索引（Document Summary Index）** 则强调信息概览能力。该类型为每个文档生成摘要，并以摘要作为检索的核心内容，用于快速了解文档整体信息，是文献预筛选、内容推荐等任务的有效手段。

下述代码片段展示了使用 LlamaIndex 创建向量索引并执行语义查询的典型过程：

from llama\_index.core import VectorStoreIndex, SimpleDirectoryReader

# 加载本地文档数据

documents = SimpleDirectoryReader("data").load\_data()

# 创建向量索引

index = VectorStoreIndex.from\_documents(documents)

# 构建查询引擎并执行问题检索

query\_engine = index.as\_query\_engine()

response = query\_engine.query("什么是知识图谱？")

通过不同索引类型的组合使用，LlamaIndex 能够构建出面向多样场景的知识增强系统，实现从精确匹配到语义推理、从摘要浏览到图结构探索的多层次智能检索能力。

表 6 LlamaIndex支持的6种索引特点对比

| 索引类型 | 数据结构 | 检索方式 | 适用场景 | 优势 | 劣势 |
| --- | --- | --- | --- | --- | --- |
| 向量存储索引（Vector Store Index） | 向量数据库 | 语义相似度检索（向量匹配） | 语义搜索、问答系统、自然语言接口 | 语义理解强，模糊匹配能力强，支持大规模检索 | 依赖嵌入模型质量，构建与存储成本相对较高 |
| 列表索引（List Index） | 顺序列表 | 顺序或并行遍历 | 小规模数据集、语义匹配明确、教学演示 | 实现简单、适合快速测试与展示 | 检索效率低，不适合大规模数据 |
| 树索引（Tree Index） | 层次树结构 | 逐层递归选择子节点（摘要驱动） | 分层结构文档、章节摘要、长文档多轮问答 | 支持摘要、多轮决策，能压缩上下文 | 构建复杂，依赖有效的摘要方法 |
| 关键词表索引（Keyword Table Index） | 关键词映射表 | 倒排索引、关键词匹配 | 法规、产品手册、具备关键词可控性的查询 | 检索精度高、响应速度快 | 无法处理同义词与语义变体，缺乏语义泛化能力 |
| 知识图谱索引（Knowledge Graph Index） | 图结构（实体+关系） | 图遍历与关系推理 | 知识管理、科研文献网络、实体间复杂关联查询 | 可结构化存储关系、支持推理与图分析 | 构建成本高、实体提取与图生成依赖外部模型与算法 |
| 文档摘要索引（Document Summary Index） | 文档摘要集 | 摘要匹配与粗粒度语义查找 | 文档导航、内容推荐、快速预览文献 | 可压缩检索开销、概览能力强 | 信息粒度粗，难以精确定位具体内容 |

### （三） LlamaIndex的高级检索策略

LlamaIndex提供了多种高级检索策略，以提高检索的准确性和相关性：

* **混合检索（Hybrid Retrieval）**：LlamaIndex 的混合检索策略通过融合语义向量检索与关键词匹配，实现语义泛化与精确定位的双重优势。相比单一检索方式，混合策略能够在理解用户意图的同时，保证对关键术语的精准命中，从而显著提升复杂查询场景下的检索准确性与结果相关性，广泛适用于技术文档问答、法规查询等对“理解 + 精准”都有高要求的应用环境。示例如下：from llama\_index.core.retrievers import VectorIndexRetriever, KeywordTableRetriever  
  from llama\_index.core.query\_engine import RetrieverQueryEngine  
  from llama\_index.core.retrievers import BM25Retriever  
    
  # 创建向量检索器  
  vector\_retriever = VectorIndexRetriever(index=vector\_index)  
    
  # 创建BM25检索器  
  bm25\_retriever = BM25Retriever.from\_defaults(index=vector\_index)  
    
  # 创建混合检索器  
  from llama\_index.core.retrievers import HybridRetriever  
  hybrid\_retriever = HybridRetriever(  
   vector\_retriever=vector\_retriever,  
   keyword\_retriever=bm25\_retriever,  
   similarity\_top\_k=2,  
  )  
    
  # 创建查询引擎  
  query\_engine = RetrieverQueryEngine.from\_args(  
   retriever=hybrid\_retriever  
  )

1. **重排序检索（Reranking Retrieval）**：是一种两阶段检索策略，先通过基础索引召回一批候选节点，再利用更强大的语言模型对这些结果进行语义重排。相比直接返回初始匹配结果，这种方法能显著提升答案的相关性和精度，特别适用于对响应质量要求较高的场景，如法律检索、学术问答和精细化推荐系统中。

**from** llama\_index.core.postprocessor **import** SentenceTransformerRerank  
*# 创建重排序处理器*  
rerank = SentenceTransformerRerank(  
 model\_name="cross-encoder/ms-marco-MiniLM-L-12-v2",  
 top\_n=2  
)  
*# 创建查询引擎*  
query\_engine = RetrieverQueryEngine.from\_args(  
 retriever=retriever,  
 node\_postprocessors=[rerank]  
)

1. **自动查询分解（Auto Query Decomposition）**：是一种面向复杂问题的智能检索策略，它将冗长或多层含义的查询自动拆解为若干子问题，分别执行检索后再聚合结果。这种方式有效提升了系统对多步骤推理、因果链条分析等复杂任务的理解与响应能力，广泛应用于科研问答、多轮对话和任务型智能助手等场景。

**from** llama\_index.core.query\_engine **import** SubQuestionQueryEngine  
  
*# 创建子问题查询引擎*  
query\_engine = SubQuestionQueryEngine.from\_defaults(  
 query\_engine=base\_query\_engine,  
 verbose=True  
)

1. **上下文感知检索（Context-Aware Retrieval）**：能够在处理查询时融合对话历史或用户上下文信息，从而实现更贴合语境的内容召回。这一策略在多轮对话中尤为关键，可有效避免“答非所问”或语义漂移，显著提升交互的连贯性与智能性，广泛应用于智能客服、对话式搜索与个性化助手等系统。

**from** llama\_index.core.query\_engine **import** SubQuestionQueryEngine

from llama\_index.core import VectorStoreIndex, SimpleDirectoryReader# Step 1：加载文档

documents = SimpleDirectoryReader("data").load\_data()

# Step 2：构建基础向量索引和查询引擎

index = VectorStoreIndex.from\_documents(documents)

base\_query\_engine = index.as\_query\_engine()

# Step 3：构建支持上下文感知的子问题查询引擎

contextual\_query\_engine = SubQuestionQueryEngine.from\_defaults(

query\_engine=base\_query\_engine,

verbose=True # 可选，打印中间子问题分解与回答过程

)

# Step 4：进行多步骤或上下文依赖的复杂查询

response = contextual\_query\_engine.query(

"请总结一下这份政策的主要内容，并说明与上一版政策的关键变化有哪些？"

)

# Step 5：输出结果

1. print(response)**路由检索（Router Retrieval）**：是一种面向多领域知识系统的智能调度策略。它能够根据查询的内容或意图，将请求动态路由至最合适的检索器或索引类型，从而实现针对性更强、效率更高的问答过程。该策略非常适合同时管理法律、医疗、教育等多个知识域的系统，在复杂场景中实现模块化、领域感知的精准信息获取。

**from** llama\_index.core.selectors **import** LLMSingleSelector  
**from** llama\_index.core.query\_engine **import** RouterQueryEngine  
  
*# 创建路由查询引擎*  
query\_engine = RouterQueryEngine.from\_defaults(  
 selector=LLMSingleSelector.from\_defaults(),  
 query\_engines={  
 "科学": science\_engine,  
 "历史": history\_engine,  
 "艺术": art\_engine  
 },  
 verbose=True  
)

### （四） LlamaIndex的查询引擎与响应合成

在 LlamaIndex 的整体架构中，“查询引擎”负责管理检索后的查询处理与响应生成，是连接用户问题与模型输出之间的核心桥梁。与前文介绍的**索引结构与检索策略**不同，查询引擎不仅调用索引结果，还决定了这些结果**如何被解释、整合、组织并最终呈现给用户**。针对不同查询需求与场景，LlamaIndex 提供了多种类型的查询引擎与响应合成策略，构成灵活而强大的问答系统基础。

#### 1.查询引擎类型概述

1）**基本查询引擎（Basic Query Engine）**：最为直观的查询引擎，负责将检索器召回的节点直接送入语言模型，由模型生成回答。适用于结构简单、信息集中、无需复杂加工的问答场景，如 FAQ 检索、定义查询等。它体现了“检索 → 生成”的最小闭环机制。

from llama\_index.core import SimpleDirectoryReader, VectorStoreIndex

# 加载文档并构建向量索引

documents = SimpleDirectoryReader("data").load\_data()

index = VectorStoreIndex.from\_documents(documents)

# 构建基础查询引擎

basic\_query\_engine = index.as\_query\_engine()

# 发起查询

response = basic\_query\_engine.query("介绍一下碳中和的基本概念。")

print(response)2）**总结查询引擎（Summary Query Engine）**：该引擎在回答过程中强调信息的整合与压缩能力，对多个相关节点进行语义摘要，提炼核心要点，形成结构化输出。非常适合政策法规、报告类文档中的“请总结关键内容”“写一个摘要”类任务，提升响应的浓缩度与概括力。

from llama\_index.core.query\_engine import SummaryQueryEngine

from llama\_index.core import SimpleDirectoryReader, VectorStoreIndex

# 加载文档和索引

documents = SimpleDirectoryReader("data").load\_data()

index = VectorStoreIndex.from\_documents(documents)

# 构建摘要型查询引擎

summary\_engine = SummaryQueryEngine.from\_args(index)

# 提出总结类问题

response = summary\_engine.query("请总结这份技术报告的主要观点。")

print(response)3）**SQL 查询引擎（SQL Query Engine）**：面向结构化数据库场景，此引擎能够将自然语言查询自动解析为 SQL 查询语句，并连接底层数据库执行查询，返回结构化结果。适用于企业报表、库存查询、客户信息系统等需要数据接口连接的问答任务。

from llama\_index.core.query\_engine import SQLQueryEngine

from sqlalchemy import create\_engine

from llama\_index.core.indices.struct\_store import SQLStructStoreIndex

# 创建数据库连接（使用 SQLite 示例）

engine = create\_engine("sqlite:///example.db")

# 构建 SQL 索引

sql\_index = SQLStructStoreIndex.from\_sql\_database(sql\_database=engine)

# 构建 SQL 查询引擎

sql\_query\_engine = SQLQueryEngine(sql\_index)

# 发起结构化自然语言查询

response = sql\_query\_engine.query("有哪些员工的工资高于 10000 元？")

print(response)

4）**知识图谱查询引擎（Knowledge Graph Query Engine）**：该引擎能够在知识图谱结构上进行节点查询与图逻辑推理，将实体关系图作为语义支撑，实现精准的结构化问答、实体上下位推理、路径搜索等复杂查询。特别适用于科研文献网络、企业知识图谱、医疗知识库等高复杂度场景。

from llama\_index.core.query\_engine import KnowledgeGraphQueryEngine

from llama\_index.core.indices.knowledge\_graph.base import KnowledgeGraphIndex

from llama\_index.core import SimpleDirectoryReader

# 加载文档

documents = SimpleDirectoryReader("data").load\_data()

# 构建知识图谱索引

kg\_index = KnowledgeGraphIndex.from\_documents(documents)

# 创建知识图谱查询引擎

kg\_query\_engine = KnowledgeGraphQueryEngine(kg\_index)

# 进行实体关系推理查询

response = kg\_query\_engine.query("哪位科学家提出了相对论？")

print(response)

5）**多步查询引擎（Multi-Step Query Engine）**：面向多跳推理任务，能够将一个复杂问题分解为若干子查询，并按逻辑依赖关系逐步执行，每步结果作为下一步的输入。这种查询引擎融合了自动任务分解、依赖建模与上下文跟踪能力，是 LlamaIndex 在多轮推理方向的重要能力体现。

from llama\_index.core.query\_engine import MultiStepQueryEngine

from llama\_index.core import SimpleDirectoryReader, VectorStoreIndex

# 加载文档和索引

documents = SimpleDirectoryReader("data").load\_data()

index = VectorStoreIndex.from\_documents(documents)

# 创建多步骤推理引擎

multi\_step\_engine = MultiStepQueryEngine.from\_defaults(query\_engine=index.as\_query\_engine())

# 提出复杂问题（需要多步思考）

response = multi\_step\_engine.query("请先解释碳排放的定义，再说明其对气候变化的影响。")

#### print(response)2.响应合成策略概述

**响应合成（Response Synthesis）**是将检索到的多个内容片段组织成**连贯、信息丰富、逻辑清晰的自然语言回答**的过程。LlamaIndex 提供三种核心合成机制，用以适配不同的上下文规模、信息复杂度与表现形式要求。

**1）重构式合成（Refine Synthesis）：**系统会逐一处理检索结果，每次在已有回答基础上继续完善，通过“多轮精炼”的方式不断融合新信息。这种策略特别适合综合多个来源、递进分析、或在动态构建答案的场景下使用，例如综合多篇报道回答一个事件的问题。

# 创建支持 refine 模式的查询引擎

refine\_engine = index.as\_query\_engine(response\_mode="refine")

# 查询：逐步精炼多个节点形成最终回答

response = refine\_engine.query("请总结这篇文章中关于数字化转型的观点。")

print(response)

**2）压缩式合成（Compact Synthesis）：**将多个检索节点压缩成紧凑的上下文表示，再统一输入语言模型生成回答。该方法优于传统拼接式处理，能在上下文窗口有限的情况下整合更多信息，是处理大量召回数据（如语义搜索 Top-K=20）时的推荐方式。

# 创建支持 compact 模式的查询引擎

compact\_engine = index.as\_query\_engine(response\_mode="compact")

# 查询：适合处理大量内容的语义压缩总结

response = compact\_engine.query("归纳这本书中关于人工智能发展的核心论点。")

print(response)

**3）树式合成（Tree Synthesis）：**利用树状结构将检索节点分层组织，从叶节点逐步向上合并内容并生成局部摘要，最终形成顶层答案。这种自底向上的分层合成方式适用于需要结构理解的复杂查询，或文档层级清晰的问题，如“请总结本章每节的核心观点”。

from llama\_index.core.indices.tree import TreeIndex

# 使用树结构索引

tree\_index = TreeIndex.from\_documents(documents)

# 创建树式合成查询引擎

tree\_engine = tree\_index.as\_query\_engine(response\_mode="tree\_summarize")

# 查询：根据结构分层总结

response = tree\_engine.query("请按照章节结构总结这篇技术白皮书。")

print(response)

因此，查询引擎与检索策略相辅相成，前者决定如何提问与处理，后者决定如何找信息；而响应合成机制则进一步决定“找到的信息如何组织起来”呈现给用户。比如，向量检索配合基本查询引擎，可以快速获得相似内容；而关键词倒排配合 SQL 或图谱引擎，可以实现结构精确定位；进一步，配合压缩式合成，可有效处理大规模召回，配合树式合成则能实现结构化表达。LlamaIndex 将“索引 + 检索 + 查询 + 合成”一体化设计，为构建灵活、可定制、具解释性的 RAG 系统提供了完整技术栈。（五）LlamaIndex与知识图谱的集成

LlamaIndex 支持与知识图谱深度集成，融合结构化知识与语言模型的生成能力，为复杂语义理解和推理任务提供强大支持。开发者可以从文本中自动构建知识图谱，借助 Neo4j、NetworkX 等后端存储实体与关系，并通过自然语言实现实体、关系及路径级别的智能查询。同时，LlamaIndex 还支持多跳推理与图结构分析，以及向量检索与图谱查询的混合策略，使得系统能同时利用非结构化文本与结构化知识，具有广泛应用于复杂问答、业务知识库、应用智能助理等高阶场景。

1. **知识图谱构建**：LlamaIndex 支持从文档中自动提取实体与语义关系，构建结构化的知识图谱，为大模型提供明确的知识边界与上下位概念支持。构建后的图谱可存储于多种后端系统，如轻量级的 NetworkX 或图数据库 Neo4j，便于可视化展示与高效查询。示例如下：

**from** llama\_index.core **import** KnowledgeGraphIndex  
**from** llama\_index.core.storage.storage\_context **import** StorageContext  
**from** llama\_index.core.graph\_stores **import** SimpleGraphStore  
  
*# 创建图存储*  
graph\_store = SimpleGraphStore()  
storage\_context = StorageContext.from\_defaults(graph\_store=graph\_store)  
  
*# 创建知识图谱索引*  
kg\_index = KnowledgeGraphIndex.from\_documents(  
 documents,  
 storage\_context=storage\_context,  
 max\_triplets\_per\_chunk=10  
)

1. **知识图谱查询**：LlamaIndex 支持以自然语言直接查询知识图谱，实现对实体、关系及多跳路径的智能检索。用户无需掌握复杂的图查询语言，即可通过语义表达触发实体定位、关系分析与路径探索，使图谱查询更直观、更智能。示例如下：

*# 创建知识图谱查询引擎*  
query\_engine = kg\_index.as\_query\_engine()  
*# 执行查询*  
response = query\_engine.query("爱因斯坦的主要贡献是什么？")

1. **知识图谱推理**：借助知识图谱结构支持智能推理和深层知识发现，能够在实体与关系构成的图中执行多跳路径分析，揭示隐含的语义关联与逻辑链条。这种能力不仅提升了问答系统的推理深度，还赋予模型跨节点联想与因果分析的能力。示例如下：

# 假设知识图谱已构建完成，命名为 kg\_index

# 创建知识图谱查询引擎（具备推理能力）

query\_engine = kg\_index.as\_query\_engine()

# 执行具有推理性质的自然语言查询

response = query\_engine.query("谁是量子力学发展过程中与爱因斯坦有争论的人？")

# 输出推理结果

print(response)

* 示例中，系统可能通过图谱中的路径分析，从“爱因斯坦”节点出发，检索到其曾争论过的关键人物（如玻尔、海森堡等），并结合关系类型（如“科学争议”或“学术辩论”）构建响应。这类查询不依赖关键词命中，而依赖于知识结构中的关系链与跳数限制，体现了 LlamaIndex 在知识推理与关联发现方面的能力。

1. **混合查询**：支持将向量检索与知识图谱查询有机融合，使系统能够同时利用非结构化文本的语义丰富性与结构化知识的逻辑严谨性。在实际应用中，这种策略可实现更全面的知识覆盖与更强的语义推理能力，尤其适用于复杂问答、跨域信息整合与高可靠性智能助手等场景。示例如下：

**from** llama\_index.core.query\_engine **import** ComposableGraphQueryEngine  
  
*# 创建混合查询引擎*  
query\_engine = ComposableGraphQueryEngine(  
 kg\_index=kg\_index,  
 vector\_index=vector\_index,  
 verbose=True  
)

### （六）LlamaIndex的应用场景与最佳实践

表 7 LlamaIndex典型应用场景与最佳实践对照表

| 应用场景 | 描述 | 推荐实践策略 |
| --- | --- | --- |
| 文档问答系统 | 基于特定文档集合进行语义问答 | 使用向量存储索引 + 混合检索 + 重排序策略 |
| 知识库管理 | 企业或个人知识内容的组织与查询 | 结合多种索引类型（关键词、向量、树索引） |
| 对话式搜索 | 保持上下文的多轮自然语言检索 | 使用上下文感知检索 + 子问题引擎（SubQuestion Engine） |
| 内容生成与摘要 | 基于查询结果生成摘要或文章片段 | 采用总结查询引擎 + 压缩/重构合成策略 |
| 多模态检索 | 同时处理文本、图像等多种模态信息 | 使用多模态索引 + 图文联合嵌入模型（如CLIP） |

LlamaIndex 提供了灵活的索引机制与可组合的检索引擎，适用于构建多种基于语言模型的信息系统。在实际应用中，不同任务场景对检索方式、上下文处理、响应组织能力提出了不同要求。表1展示了五类最具代表性的应用场景，并为每类场景推荐了相应的最佳实践策略。

在**文档问答系统**中，系统需从大量非结构化文档中找出与提问相关的片段，并基于语义内容生成回答。此类任务推荐使用**向量索引**作为主索引方式，并结合**混合检索**与**重排序机制**来平衡覆盖率与回答精度。

对于**企业知识库管理**这类包含结构化与半结构化信息的系统，建议根据知识内容的特点灵活选用多种索引方式（如关键词索引、树索引、图谱索引），以提升查询适配度和系统维护灵活性。

在面向用户的**对话式搜索系统**中，系统需要理解查询上下文、保留对话历史，并对上下轮的问题进行一致性响应，因此推荐使用\*\*上下文感知检索器（Context-Aware Retriever）\*\*与子问题分解模块，实现更自然的多轮交互。

当系统的任务是生成摘要或内容重组时，如自动写作助手、报告生成器，使用总结查询引擎（Summary Query Engine）配合压缩式或重构式响应合成策略，可以有效生成结构化、有条理的内容摘要。

而在**多模态检索系统**中，文本、图像、表格等不同模态的信息需被统一编码和检索，此时需引入支持图文联合嵌入或 CLIP 模型的多模态索引器，并进行特定向量融合设计。以上策略的组合体现了 LlamaIndex 架构的高度可插拔性与应用弹性。

表 8 LlamaIndex构建项目的五阶段实践流程

|  |  |  |  |
| --- | --- | --- | --- |
| 实践环节 | 主要任务目标 | 典型难点 | 实践要点建议 |
| I阶段：数据预处理优化 | 提取高质量文本片段，提升后续检索精度与上下文一致性 | 分块尺度难以统一，内容可能断裂 | - 合理设置分块大小（如 512/1024 tokens） |
| - 添加作者、时间等元数据作为检索条件 |
| II阶段：索引策略选择 | 根据数据规模与结构选择最合适的索引结构 | 不同文档混合使用时难统一结构 | - 小文档可用向量索引 |
| - 长结构内容推荐树状索引 |
| - 多模态可引入混合索引 |
| III阶段：检索参数调优 | 提升召回结果的相关性、控制性能开销 | top\_k 参数设定不当导致噪声或遗漏 | - top\_k 建议从 3~10 实验起步 |
| - 尝试不同相似度度量（cosine/dot/L2） |
| IV阶段响应生成优化 | 让大模型输出高质量、可控的语义回答 | 回答冗长、偏离重点、幻觉问题 | - 编写 Prompt 模板引导结构化生成 |
| - 可引入自动评估指标（如 GPTScore、BLEU） |
| V阶段：系统集成与部署 | 提升系统运行稳定性和响应效率，保证用户体验 | 出错无兜底，缓存策略不当 | - 实现结果缓存（如相同查询缓存 Embedding + Answer） |
| - 设置异常回退逻辑 |

为了构建一个稳定、准确且可扩展的 LlamaIndex 应用系统，开发者需要遵循一套科学的开发流程。表2从工程实施的角度，将整个流程分为五个关键阶段，分别对应数据准备、索引设计、检索调优、响应生成与系统部署五个连续环节，每个阶段均伴随典型挑战与核心优化点。

第一阶段是**数据预处理优化**，这一步决定了后续检索系统的表现上限。开发者需要合理划分文档片段（如设置滑动窗口、按段落分块），同时补充元数据（如来源、时间、类别）以辅助更精准的查询过滤。这一阶段的挑战在于平衡上下文完整性与检索粒度。

第二阶段是**索引策略选择**，需根据数据的体量、类型与访问模式决定使用何种索引。小规模数据集可以直接使用向量索引；结构化分层内容推荐使用树索引；而多模态、跨领域系统往往需要混合索引结构的支持。

第三阶段聚焦于**检索参数调优**，开发者需要对如 top\_k（返回节点数量）、相似度度量方式（如 cosine、dot）等参数进行实验，找到最适配数据与任务的配置。这一阶段往往决定了召回质量与效率之间的平衡。

第四阶段是**响应生成优化**，旨在提升大模型输出的语言质量、信息覆盖与逻辑组织能力。设计合适的提示（Prompt）模板和结构化输出格式，可以显著提升系统的专业表现。此阶段可引入 BLEU、ROUGE 或 GPTScore 等自动评估工具，构建闭环优化体系。

最后，**系统集成与部署**阶段是将模型能力转化为可服务化产品的关键。开发者需合理引入缓存策略（如向量缓存、回答缓存），并实现异常回退机制（如查询失败的兜底回答），以保障系统的响应速度与用户体验。

## 参考文献

[1] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. arXiv preprint arXiv:1908.10084.

[2] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[3] OpenAI. (2022). OpenAI API. https://beta.openai.com/docs/guides/embeddings

[4] Wang, L., Lyu, Q., & Gao, J. (2023). E5: A new text embedding by weakly-supervised contrastive pre-training. arXiv preprint arXiv:2212.03533.

[5] Gao, T., Yao, X., & Chen, D. (2021). SimCSE: Simple contrastive learning of sentence embeddings. arXiv preprint arXiv:2104.08821.

[6] BAAI. (2023). BGE: BAAI general embeddings. https://github.com/FlagOpen/FlagEmbedding

[7] Li, J., Sun, C., Patwary, M., Shoeybi, M., & Dey, P. (2023). GTE: General text embeddings for retrieval and generation. arXiv preprint arXiv:2308.03281.

[8] Andoni, A., & Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of the ACM, 51(1), 117-122.

[9] Jégou, H., Douze, M., & Schmid, C. (2011). Product quantization for nearest neighbor search. IEEE transactions on pattern analysis and machine intelligence, 33(1), 117-128.

[10] Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE transactions on pattern analysis and machine intelligence, 42(4), 824-836.

[11] Milvus Team. (2023). Milvus: An open-source vector database. https://milvus.io/

[12] Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3), 535-547.

[13] Mao, Y., Xiong, C., Sabharwal, A., & Radev, D. (2023). Query rewriting for retrieval-augmented large language models. arXiv preprint arXiv:2305.14283.

[14] Gao, L., Ma, X., Lin, J., & Callan, J. (2022). Precise zero-shot dense retrieval without relevance labels. arXiv preprint arXiv:2212.10496.

[15] Luan, Y., Eisenstein, J., Toutanova, K., & Collins, M. (2021). Sparse, dense, and attentional representations for text retrieval. Transactions of the Association for Computational Linguistics, 9, 329-345.

[16] Ragas Team. (2023). Ragas: Evaluation framework for your RAG pipelines. https://github.com/explodinggradients/ragas

[17] Hogan, A., Blomqvist, E., Cochez, M., d’Amato, C., Melo, G. D., Gutierrez, C., … & Zimmermann, A. (2021). Knowledge graphs. ACM Computing Surveys, 54(4), 1-37.

[18] W3C. (2014). RDF 1.1 concepts and abstract syntax. https://www.w3.org/TR/rdf11-concepts/

[19] Gruber, T. R. (1993). A translation approach to portable ontology specifications. Knowledge acquisition, 5(2), 199-220.

[20] Musen, M. A. (2015). The Protégé project: A look back and a look forward. AI matters, 1(4), 4-12.

[21] Li, J., Sun, A., Han, J., & Li, C. (2022). A survey on deep learning for named entity recognition. IEEE Transactions on Knowledge and Data Engineering, 34(1), 50-70.

[22] Liu, J. (2023). LlamaIndex: A data framework for LLM applications. https://github.com/jerryjliu/llama\_index