知识图谱入门

树懒老K

个人网站　个人微信　公众号

知识图谱入门

树懒老K

作者：树懒老K
出版：树懒老K
年份：2026

目 录

献辞

献给每一个在终端里敲下第一行命令的人。

你们不知道自己在做什么，但你们选择了开始。

这就够了。

序 给每一个想搞懂"数据怎么变成知识"的人

我第一次听说”知识图谱”这个词，是在2012年。那年Google发布了一个叫Knowledge Graph的东西，号称要让搜索引擎”理解”用户的查询意图，而不只是匹配关键词。

当时我觉得这是大公司搞的营销概念。

十二年后，我不这么看了。

2024到2025年，大语言模型（LLM）像一场海啸席卷了整个技术圈。ChatGPT、Claude、Gemini——它们能写诗、能编程、能翻译、能对话。但用着用着你会发现一个致命问题：它们会”一本正经地胡说八道”。

这个问题有个学名叫”幻觉”（Hallucination）。

怎么治幻觉？一个被反复验证的方案是：让大模型去查”靠谱的知识库”，然后再回答。这个方案叫RAG（检索增强生成）。而知识图谱，恰恰是最”靠谱”的知识库之一——因为它的每一条知识都是结构化的、可溯源的、可验证的。

于是知识图谱在AI时代迎来了它的第二春。

但问题来了：知识图谱的学习门槛不低。RDF、OWL、SPARQL、Cypher、向量数据库……光是这些缩写就够劝退一大半人。市面上的书要么太学术（满页公式和证明），要么太浅（只讲概念不动手），要么太旧（还在讲2015年的技术栈）。

这就是这本书出现的时机。

树懒老K做了一件很难的事：他把知识图谱从”听起来很厉害但不知道怎么开始”变成了一个”跟着做就能跑起来”的实操教程。从零装Ubuntu开始，手把手教你搭Neo4j、写Cypher、用Qdrant做向量搜索、最后把LLM和图谱接在一起搭一个GraphRAG系统。

更难得的是，他没有假装自己是专家。他在前言里说”我是边学边写的”——这恰恰是这本书最大的优点。他用一个学习者的视角，把每一个”我当时卡在这里”的地方都讲清楚了。

如果你是一个刚接触知识图谱的开发者，这本书会是你最好的起点。如果你是一个已经在用大模型但想给模型加上”结构化知识”的工程师，这本书后半部分正好是你需要的。

如果你只是想搞明白”知识图谱到底是什么”——读完前三章就够了。

知识图谱不是什么高深的东西。它就是”把世界画成一张图”。

而这本书，就是那支画笔。

—— 一个从2024年开始用知识图谱做产品的工程师

2026年夏

前言 为什么我要写一本关于知识图谱的书

> *知识不是信息。知识是被连接起来的信息。*

一切始于一个问题

2024年，我在做一个项目时需要回答一个问题：”我们公司的所有业务知识，能不能让AI来查？”

听上去很简单。把文档喂给大模型不就行了？

试了之后我才发现，大模型确实能回答很多问题，但它有一个根本性的缺陷：它不知道什么是”对的”。它可以编造一个听起来非常合理的答案，而这个答案和公司实际情况完全不符。

后来有人告诉我一个词：RAG。再后来有人告诉我：RAG做得好，需要知识图谱。

知识图谱？那是什么？

我当时对知识图谱的全部认知来自一篇科普文章，大意是”Google用它来让搜索更智能”。我以为那是大公司才玩得起的东西，跟我没关系。

但当我真正开始学、开始做之后，我发现：知识图谱其实不复杂。它就是把信息组织成”节点”和”边”——谁认识谁、谁属于什么、谁在哪里——然后画成一张图。难的是从零开始的那一步：装环境、选工具、写第一条查询。

而市面上几乎没有一本书能帮我走完这一步。

这本书的读者

这本书是写给和我一样的人的。

你不是计算机科学家。你可能没学过图论，不知道RDF是什么，分不清Neo4j和PostgreSQL有什么本质区别。你听说过”知识图谱”这个词，觉得它很酷，但不知道从哪里开始。

或者你已经开始了，但卡在了某个地方——装环境报错了、Cypher查询写不出来、向量搜索的结果不对——而你找不到一本能帮你debug的书。

这本书就是为你写的。

这本书的设计

全书分四篇，15章。

第一篇（第1–3章）是理论打底。我会用最通俗的语言讲清楚知识图谱是什么、从哪来、能干什么。这三章不写代码，但读完之后你会对知识图谱有一个完整的直觉。

第二篇（第4–7章）是技术底座。RDF、OWL、SPARQL、属性图、Cypher、本体设计、知识抽取——这些是知识图谱的”内功”。每一章都配有实操环节，但重点是理解概念和方法论。

第三篇（第8–12章）是动手实操。从零装Ubuntu开始，手把手教你用Neo4j和Qdrant，用Python操控图谱，最后做一个完整的电影知识图谱项目。这是全书最”硬”的部分——每一步都有代码、有命令、有截图。

第四篇（第13–15章）是进阶与展望。LLM时代的知识图谱（GraphRAG）、生产环境的部署与优化、以及知识图谱在AI Agent和多模态方向的未来。

我的承诺

我不是知识图谱领域的专家。我是一个边学边做的人。

但正因为如此，我知道哪里会卡住、哪里会困惑、哪里会走弯路。我把这些”坑”都写进了这本书里，希望能帮你少踩几个。

这本书不追求学术上的严谨和完备。它追求的是”读完就能动手”。

如果有一句话让你觉得”哦，原来是这么回事”——那我的工作就没白做。

慢慢走，不赶路，但不停步。

—— 树懒老K

2026年夏

凡例 怎么读这本书

关于本书结构

全书分四篇、15章，外加附录。四篇之间是递进关系，但每一章内部是相对独立的。

如果你是零基础，建议从第1章开始顺序阅读。如果你已经了解知识图谱的基本概念，可以直接跳到第二篇的技术底座，或者第三篇的实操部分。

关于代码

本书所有代码都在Ubuntu 22.04环境下测试通过。代码块均标注了语言类型（bash、cypher、python、sparql等），可直接复制运行。

代码中的 `$` 前缀表示终端命令，`neo4j$` 表示Neo4j Browser中的Cypher命令，`>>>` 表示Python交互模式。

关于每章结构

每章的结构大致如下：

- **引子**：一个故事或问题，引出本章主题
- **正文**：概念讲解 + 实操演示

- **动手练习**：你可以立刻动手做的小任务
- **延伸阅读**：推荐的论文、文章、视频
- **本章小结**：一两句话概括本章核心要点

关于难度标记

实操部分用星级标注难度：

- ★☆☆ 入门级：跟着做就行
- ★★☆ 进阶级：需要一些前置知识
- ★★★ 挑战级：需要综合前面多个章节的知识

关于工具版本

本书基于以下工具版本编写：

- Ubuntu 22.04 LTS
- Neo4j 5.x Community Edition
- Qdrant 1.x
- Python 3.10+
- Apache Jena 5.x

工具和库会持续更新，如果命令有变动，请以官方文档为准。

关于反馈

如果书中有任何错误或不清楚的地方，欢迎通过GitHub提交Issue。技术在变，这本书也会跟着更新。

第一章 什么是知识图谱

引子：为什么谷歌比你更懂你?

让我问你一个问题：你有没有在谷歌或者百度搜索过”姚明身高”?

你输入这四个字，按下回车，搜索结果页面的最顶部，大概率会直接弹出一个醒目的卡片——“姚明，身高 2.26 米”。不需要点进任何网页，不需要翻阅任何文章，答案就那样直挺挺地摆在你面前。

你是不是觉得这理所当然?

停一下，仔细想想——这件事其实非常不可思议。

搜索引擎的数据库里存了几万亿个网页，它怎么就知道”姚明”是一个人,”身高”是一个属性，而”2.26米”是这两个东西组合在一起的答案？它是怎么”理解”你的问题的?

再试一个更复杂的：搜索”流浪地球的导演是谁”。搜索引擎会告诉你”郭帆”。可是,”流浪地球”是一部电影,”导演”是一种职业角色,”郭帆”是一个人——搜索引擎是怎么把这三个完全不同类别的东西串联起来，最终给你一个精准答案的?

答案就是：**知识图谱（Knowledge Graph）**。

2012年，谷歌正式提出了”Knowledge Graph”这个概念，并且把它集成到了搜索引擎中。从那以后，搜索结果开始变得”聪明”了——它不再只是给你一堆包含关键词的网页链接，而是开始尝试

理解你的问题，并且直接给你答案。

但这只是冰山一角。知识图谱的能力远不止”让搜索更聪明”这么简单。今天，它已经渗透到了我们生活的方方面面：从智能客服到金融风控，从医疗诊断到药物研发，从个性化推荐到国家安全……知识图谱正在悄悄地改变世界运作的方式。

这本书，就是来帮你搞懂它到底是什么、怎么工作的。

我们从最基础的问题开始——

正文

1.1 知识图谱的”最小单元”：三元组

在开始之前，我想请你先做一个思维实验。

假设你要向一个完全不了解中国电影的外国朋友介绍《流浪地球》，你会怎么说？你可能会说：

- 《流浪地球》的导演是郭帆。
- 《流浪地球》的主演包括吴京。
- 郭帆是中国导演。
- 吴京是中国演员。
- 《流浪地球》是一部科幻电影。

注意到了吗？你自然而然地用了一种非常特别的表达结构：**“什么东西”的”什么属性”是”什么值”**。**“流浪地球”—“导演”—“郭帆”**，这就是一个最基本的知识表达。

在知识图谱的世界里，这种表达方式有一个正式的名字：**三元组（Triple）**。

一个三元组由三个部分组成：

表 5–1

组成部分	名称	例子
第一个元素	**主体（Subject）** / 头实体	流浪地球
第二个元素	**关系（Predicate）** / 谓词 / 边	导演
第三个元素	**客体（Object）** / 尾实体	郭帆

写成更形式化的表达就是：

```
（流浪地球，导演，郭帆）
（流浪地球，主演，吴京）
（郭帆，职业，导演）
（吴京，职业，演员）
（流浪地球，类型，科幻电影）
```

这五个三元组，就构成了一个微型的知识图谱。

你可能会说：这也太简单了吧？不就是把一句话拆成三个部分吗？

对，就是这么简单。但请不要小看这种”简单”——这恰恰是知识图谱最强大的地方。**用最简单、最统一的结构来表达知识**，这是它的设计哲学。

就像乐高积木一样：每一块积木都非常简单，就是一个带凸起的小方块。但当你把成千上万块积木拼接在一起的时候，你可以搭建出城堡、飞船、甚至整个城市。知识图谱的三元组就是这样的”积木块”——每一条都很简单，但当成千上万条、上亿条三元组连接在一起的时候，就构成了一张庞大的”知识之网”。

1.2 实体、关系与属性：图谱的”三驾马车”

在刚才的三元组里，我们其实涉及了三种不同的角色：

实体（Entity）：知识图谱中的”节点”，也就是”东西”。它可以是一个具体的人（姚明）、一部电影（流浪地球）、一个地点（北京）、一个概念（科幻电影），甚至是一个抽象的事物（自由、正义）。实体是知识图谱的主角。

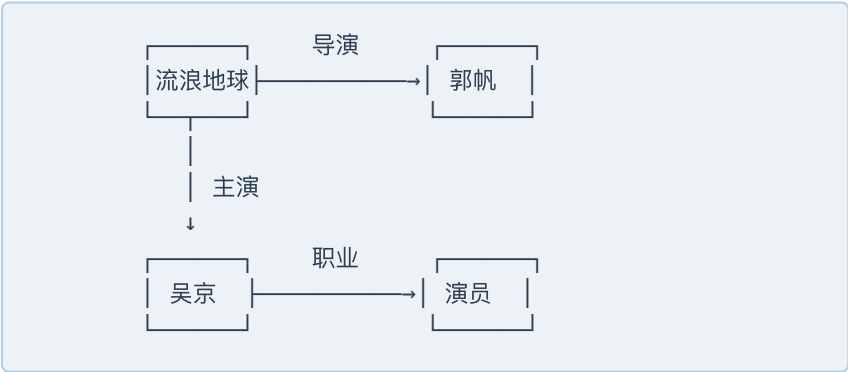
关系（Relationship / Edge）：连接两个实体的”线”。它描述了两个实体之间是怎么关联的。比如”导演”连接了《流浪地球》和郭帆，”出生地”连接了姚明和上海。

属性（Attribute / Property）：描述实体自身特征的”标签”。比如姚明的身高是2.26米，体重是141公斤。属性通常指向一个具体的数值或字符串，而不是另一个实体。

用一个更直观的比喻来理解这三者：

想象一个社交网络。每个人是一个**实体**。你和张三之间有一条”朋友”的**关系**。而你自己的年龄、性别、爱好则是你的**属性**。

把这三样东西画出来，就成了一张”图”：



看到了吗？实体是**点**，关系是**线**，点线交织在一起，就形成了一张”图”——这就是”知识图谱”名字的由来。

1.3 知识图谱 vs. 关系数据库 vs. 电子表格

到这里你可能会问：知识图谱不就是把数据换了种方式存起来吗？我用 Excel 表格不行吗？用 MySQL 数据库不行吗？

这是非常好的问题。我们来做一个对比，看看它们的本质区别在哪里。

电子表格（Excel）的方式

如果你用 Excel 来存储《流浪地球》的信息，你大概会建一个这样的表格：

表 5–2

电影名	导演	主演	类型	上映年份
流浪地球	郭帆	吴京	科幻	2019
战狼2	吴京	吴京	动作	2017
同桌的你	郭帆	周冬雨	爱情	2014

这种方式非常直观，非常适合做统计、筛选、排序。但如果你想回答一个问题：”吴京参演的所有电影中，有哪些导演还执导过其他科幻电影？”——这个问题需要跨越多行、多列进行关联推理。在 Excel 里做这种操作，你大概需要写一堆 VLOOKUP 公式，而且非常不灵活。

关系数据库（MySQL）的方式

用关系数据库，你会建几张表：`电影表`、`导演表`、`演员表`、`参演关系表`……然后通过 SQL 的 JOIN 操作把它们关联起来。

```
SELECT d.name AS director_name
FROM movies m
JOIN movie_actors ma ON m.id = ma.movie_id
JOIN actors a ON ma.actor_id = a.id
JOIN directors d ON m.director_id = d.id
JOIN movies m2 ON d.id = m2.director_id
WHERE a.name = '吴京' AND m2.genre = '科幻';
```

这可以工作，但有几个问题：

1. **模式（Schema）是固定的**。你必须提前把表结构定义好。如果明天你想加入”电影的投资公司”这个信息，你得改表结构，加表或者加列。
2. **关系是隐式的**。表和表之间的关联靠外键（Foreign Key）来维护，你必须写 JOIN 语句才能把关系”挖”出来。如果你想回答”吴京和郭帆之间有什么关系？”——你可能需要写很多复杂的查询。
3. **多跳查询很痛苦**。如果问题变成”吴京的朋友的朋友中，有谁当过导演？”——这种需要多次”跳”的查询，SQL 写起来会非常复杂，而且性能很差。

知识图谱的方式

知识图谱用一种完全不同的思路：

1. **不需要提前定义表结构**。你随时可以添加新的实体、新的关系类型，不需要修改任何”表结构”。
2. **关系是一等公民**。关系直接存在于三元组中，不需要 JOIN，不需要外键。”吴京—参演—流浪地球”本身就是一条数据。
3. **多跳查询是天然擅长的**。”吴京的朋友的朋友中谁当过导演？”——在知识图谱里，这就是沿着”朋友”的边跳两次，再沿着”职业”的边跳到”导演”节点，如此而已。

用一个比喻来总结：

- **电子表格**像一本笔记本——规整、直观，但不善于处理复杂的关联。
- **关系数据库**像一个档案柜——每个抽屉里的档案都分门别类放好了，你要跨抽屉找信息就得一个一个打开来对比。

- **知识图谱**像一张蜘蛛网——所有东西都通过丝线连在一起，你从任何一个点出发，都可以沿着丝线找到任何相关联的东西。

1.4 知识图谱能做什么？不能做什么？

在了解了知识图谱的基本结构之后，我们来看看它的能力边界。很多人对新技术有一种不切实际的期待，觉得它无所不能。知识图谱当然不是万能的，搞清楚它能做什么、不能做什么，对你后续的学习非常重要。

知识图谱擅长的事情：

1. **知识的结构化表达和存储**。把零散的知识变成一张有组织、可查询的网络。比如把一本医学教科书里的所有疾病、症状、药物、治疗方法之间的关系整理成一张图谱。
2. **语义搜索和问答**。理解用户的意图，而不仅仅是关键词匹配。当用户问”治疗高血压的药物有哪些”，知识图谱知道”高血压”是一种疾病，”治疗”是一种关系，”药物”是一种实体类型。
3. **推理和发现**。基于已有的知识进行逻辑推理。如果图谱知道”A是B的父亲”，”B是C的父亲”，就可以推理出”A是C的祖父”。这在药物研发中特别有用——发现”药物X作用于蛋白质Y，蛋白质Y与疾病Z有关”，就可能推理出”药物X可能治疗疾病Z”。
4. **知识融合**。把来自不同来源的知识整合在一起。比如你的公司有客户数据库、CRM系统、社交媒体数据，知识图谱可以把这些信息融合成一张统一的客户知识网络。

5. **可解释的 AI**。在深度学习的”黑箱”时代，知识图谱提供了一种让 AI 决策可解释的途径。因为图谱里的每一条知识都是明确的、可追溯的。

知识图谱不擅长（或需要配合其他技术）的事情：

1. **处理非结构化数据**。知识图谱本身是结构化的。要从一篇文章、一段语音、一张图片中提取知识，你需要先用自然语言处理（NLP）或计算机视觉（CV）技术把它转化为结构化信息，然后才能放入图谱。
2. **常识推理**。人类觉得理所当然的常识——“水往低处流”“火会烫伤人”——在知识图谱中往往需要显式编码，否则系统就不知道。虽然有一些常识知识图谱（如 ConceptNet），但远未覆盖人类的全部常识。
3. **实时性要求极高的场景**。大规模图谱的查询和推理有时会比较慢，对于毫秒级响应的实时系统可能需要配合缓存、索引等优化手段。
4. **主观判断和模糊推理**。”这部电影好不好看？”“这个人是不是好人？”——这类涉及主观评价和模糊边界的问题，知识图谱很难直接回答。
5. **从零开始自动构建**。目前，构建一个高质量的知识图谱仍然需要大量的人工投入——定义本体（Ontology）、标注数据、校验质量——这不是按一个按钮就能完成的事情。

1.5 你的第一个知识图谱：动手实现

好了，理论讲了这么多，让我们动手写代码吧！

下面，我将用最简单的方式——Python 语言加上一个非常轻量的库——来构建你的第一个知识图谱，并且把它画出来。

你需要准备的环境：

- Python 3.8 以上（推荐 3.10+）
- pip 包管理器

首先，安装我们需要的两个库：

```
pip install networkx matplotlib
```

`networkx` 是 Python 中最流行的图（Graph）操作库，`matplotlib` 是画图用的。

完整代码：

```
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib

# 设置中文字体（macOS 用 PingFang，Linux 可能需要改成 SimHei 或
matplotlib.rcParams['font.sans-serif'] = ['PingFang SC', '
matplotlib.rcParams['axes.unicode_minus'] = False

# ==============================
# 第一步：定义你的三元组
# ==============================
triples = [
    ("流浪地球", "导演", "郭帆"),
    ("流浪地球", "主演", "吴京"),
    ("郭帆",     "职业", "导演"),
    ("吴京",     "职业", "演员"),
    ("流浪地球", "类型", "科幻电影"),
]

# ==============================
# 第二步：构建图
# ==============================
G = nx.DiGraph()  # DiGraph = 有向图（Directed Graph）

for subject, predicate, obj in triples:
    G.add_edge(subject, obj, label=predicate)

# ==============================
# 第三步：画图
# ==============================
plt.figure(figsize=(10, 6))

# 使用 spring 布局算法让节点自动排列
pos = nx.spring_layout(G, k=2, seed=42)
```

```
# 画节点
nx.draw_networkx_nodes(G, pos, node_size=2000,
                       node_color='#E8F4FD',
                       edgecolors='#2196F3',
                       linewidths=2)

# 画边（箭头）
nx.draw_networkx_edges(G, pos, arrowstyle='->',
                       arrowsize=20,
                       edge_color='#FF9800',
                       width=2)

# 画节点标签
nx.draw_networkx_labels(G, pos, font_size=12, font_weight=

# 画边的标签（关系名称）
edge_labels = nx.get_edge_attributes(G, 'label')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labe
                             font_size=10, font_color='red

plt.title("我的第一个知识图谱", fontsize=16, fontweight='bold
plt.axis('off')  # 隐藏坐标轴
plt.tight_layout()
plt.savefig("my_first_kg.png", dpi=150, bbox_inches='tight
plt.show()

# ==============================
# 第四步：查询图谱
# ==============================
print("=" * 40)
print("图谱基本信息：")
print(f"  节点数量: {G.number_of_nodes()}")
print(f"  边的数量: {G.number_of_edges()}")
print()
```

```
# 查询：流浪地球的所有关系
print("流浪地球的所有关系：")
for _, target, data in G.edges("流浪地球", data=True):
    print(f"  流浪地球 --[{data['label']}]--> {target}")
print()

# 查询：谁有"职业"这个关系？
print("所有定义了职业的实体：")
for source, target, data in G.edges(data=True):
    if data['label'] == '职业':
        print(f"  {source} 的职业是 {target}")
```

运行这段代码之后，你会看到：

1. 一张可视化的图——五个节点通过带标签的箭头连接在一起。这就是你的第一个知识图谱！
2. 控制台输出的查询结果，展示了你可以如何”询问”这张图谱。

让我解释一下代码中的关键概念：

- `nx.DiGraph()`：创建一个”有向图”。有向意味着边有方向——从主体指向客体。”流浪地球→导演→郭帆”和”郭帆→导演→流浪地球”是完全不同的含义，所以方向很重要。
- `G.add_edge(subject, obj, label=predicate)`：添加一条边。在知识图谱中，一条边就对应一个三元组。主体和客体是边的两端，关系是边上的标签。
- `nx.spring_layout`：一种自动布局算法，它模拟物理世界中弹簧和斥力的效果，让图自动排列得比较美观。

恭喜你！你现在已经拥有了一个可以运行的知识图谱原型。虽然它只有 5 条数据，但它的结构和谷歌的知识图谱、维基数据的知识图谱是一模一样的——只是规模不同而已。

1.6 从5条到50亿条：规模的力量

你可能会想：就这？5条数据有什么用？

确实，5条三元组什么也做不了。但请你想象一下，如果这个数字不是5，而是50亿呢？

谷歌的知识图谱在 2020 年就包含了超过 **700亿条** 事实（facts）。Wikidata——一个开放的、协作的知识图谱——目前拥有超过 **1亿条** 结构化数据，覆盖了人类知识的方方面面。

当三元组的数量达到这个规模的时候，一些有趣的事情就会发生：

涌现性推理能力：当图谱足够大的时候，很多你从未显式编程的推理能力会自然涌现。比如，你的图谱里从来没有”姚明的女儿的父亲是谁”这个三元组，但如果图谱知道”姚明是姚沁蕾的父亲”，它就能回答这个问题——即使从来没有人把这条知识显式地写进去。

跨领域关联发现：一个同时包含生物学知识和医学知识的图谱，可能自动发现”某种蛋白质与某种疾病有关联”——而这种关联，可能是任何单个领域专家都没有注意到的。

语义理解的飞跃：当一个搜索引擎背后有一个包含数十亿实体的图谱时，它就能真正”理解”用户查询的含义，而不是仅仅做关键词匹配。这就是为什么今天的谷歌搜索比2012年之前的搜索”聪明”得多。

这就是规模的力量。知识图谱的价值，随着数据量的增长呈指数级上升。这也是为什么大型科技公司都在疯狂地扩充自己的知识图谱。

1.7 知识图谱的”灵魂”：本体（Ontology）

在结束本章之前，还有一个重要的概念需要介绍——**本体（Ontology）**。

如果三元组是知识图谱的”砖块”，那本体就是它的”蓝图”。

本体定义了： – 有哪些**类型的实体**（人、电影、地点、公司……） – 有哪些**类型的关系**（导演、主演、出生地、隶属于……） – 这些类型之间的**层次和约束**（”导演”是”职业”的一种;”电影”的”导演”必须是”人”，不能是”地点”）

举个简单的例子。如果我们为我们的电影知识图谱定义一个本体，它可能包括：

```
实体类型：人物、电影、职业、类型
关系类型：导演、主演、职业、类型

约束规则：
- "导演"关系的主体必须是"电影"，客体必须是"人物"
- "主演"关系的主体必须是"电影"，客体必须是"人物"
- "职业"关系的主体必须是"人物"，客体必须是"职业"
- "类型"关系的主体必须是"电影"，客体必须是"类型"
```

本体的作用就像城市规划图：它规定了哪里可以建住宅、哪里可以建商业区、道路应该怎么连接。没有本体，知识图谱就是一座没有规划的城市——虽然也能住人，但迟早会陷入混乱。

在实际工程中，本体的设计是知识图谱建设中最重要的决策之一。一个好的本体设计可以让图谱扩展性极强，而一个糟糕的本体设计可能让你在后期付出巨大的重构代价。我们将在后面的章节中详细讨论本体设计的方法论。

1.8 知识图谱与大语言模型：AI时代的”双剑合璧”

既然这本书的副标题提到了”AI时代”，我必须在这里谈一谈知识图谱和大语言模型（LLM，如 GPT、Claude 等）的关系。

很多人问我：现在大语言模型这么厉害，知识图谱是不是过时了？

我的回答恰恰相反：**大语言模型的出现，让知识图谱变得更加重要了。**

为什么？因为大语言模型有几个天然的缺陷，而知识图谱恰好可以弥补这些缺陷：

1. **大语言模型会”幻觉”（Hallucination）**。它会一本正经地告诉你完全错误的信息。而知识图谱中的每一条知识都是经过验证的事实，可以作为”事实锚点”来校正大模型的输出。这就是所谓的 **RAG（Retrieval-Augmented Generation，检索增强生成）** 技术的一个重要来源。
2. **大语言模型的知识是”冻结”的**。它的知识截止到训练数据的截止日期。而知识图谱可以随时更新，为模型提供最新的知识。
3. **大语言模型的推理过程不透明**。你不知道它是从哪里”想”出来的答案。知识图谱可以提供清晰的推理路径——答案经过了哪些实体和关系，一目了然。
4. **大语言模型缺乏结构化知识**。它”知道”很多事，但它的知识是分散在参数中的，很难精确地提取和组织。知识图谱天然就是结构化的。

所以，当今最前沿的 AI 系统，往往是**知识图谱 + 大语言模型**的组合。比如：

- 百度在搜索中使用知识图谱来增强大模型的回答质量
- 微软的 Copilot 使用知识图谱来提供更准确的技术文档回答
- 医疗领域的 AI 系统使用 SNOMED CT 等医学知识图谱来确保诊断建议的准确性

这种”结构化知识 + 生成式 AI”的组合，被很多人认为是通向更强 AI 的关键路径。在本书的后续章节中，我们会详细讨论如何将知识图谱与大语言模型结合。

动手练习

练习 1：设计你的个人知识图谱

选择一个你感兴趣的领域（音乐、体育、游戏、美食……），设计 10–15 个三元组来描述这个领域的知识。要求：

1. 至少包含 3 种不同类型的实体
2. 至少包含 3 种不同类型的关系
3. 画出这些三元组构成的图（可以手绘，也可以用代码）

思考题：你的图谱中是否存在可以”推理”出来的隐含知识？

练习 2：扩展你的代码

在前面的代码基础上，做以下改进：

1. 添加更多三元组（至少 15 条）
2. 实现一个函数 `query_path(graph, start, end)`，找出两个实体之间的路径
3. 实现一个函数 `find_all_related(graph, entity, relation_type)`，找出某个实体通过某种关系连接到的所有其他实体

提示：networkx 提供了 `nx.shortest_path()` 函数来查找两个节点之间的最短路径。

练习 3：对比实验

选择 5 个你熟悉的人物或事物，分别用以下三种方式记录它们的信息：

1. 一张 Excel 表格
2. 一组 SQL 建表和插入语句
3. 一组三元组

然后尝试回答以下问题，比较三种方式的难易程度：

- “找出所有满足条件A和条件B的实体”
- “A 和 B 之间有什么关系？”（假设它们之间有多个间接关系）
- “如果要增加一个新的信息字段，需要改动什么？”

延伸阅读

1. **Google 官方博客：Introducing the Knowledge Graph** (2012)
2. Amit Singhal 撰写的这篇博文是知识图谱概念的起点，阐述了”things, not strings”的核心理念。
3. 链接：https://blog.google/products/search/introducing-knowledge-graph-things-not/
4. **《Knowledge Graphs》by Aidan Hogan 等** (2021)

5. 目前最全面的知识图谱教科书，涵盖了数据模型、查询语言、推理方法等方方面面。免费在线阅读：https://kgbook.org
6. **《知识图谱：方法、实践与应用》** — 王昊奋、漆桂林等
7. 中文社区中最受欢迎的知识图谱入门书籍之一，适合了解中文知识图谱的技术生态。
8. **RDF 和 SPARQL 规范**
9. RDF（Resource Description Framework）是知识图谱最常用的数据模型标准。
10. SPARQL 是查询 RDF 数据的标准语言。
11. W3C 官方文档：https://www.w3.org/RDF/ 和 https://www.w3.org/TR/sparql11-query/

本章小结

知识图谱是一种用”实体—关系—实体”三元组来表达知识的数据结构，它把零散的知识编织成一张互联的”网”，使得机器能够理解、查询和推理人类知识。知识图谱的核心优势在于灵活的模式、显式的关系表达和天然的多跳查询能力，它与关系数据库和电子表格形成了互补而非替代的关系。

第二章 图的故事：从欧拉到AlphaGo

引子：一座桥引出的数学分支

1736年，普鲁士的哥尼斯堡（Königsberg，今天属于俄罗斯，名叫加里宁格勒），有一条普雷格尔河穿城而过。河中间有两座小岛，岛与河岸之间由七座桥连接。

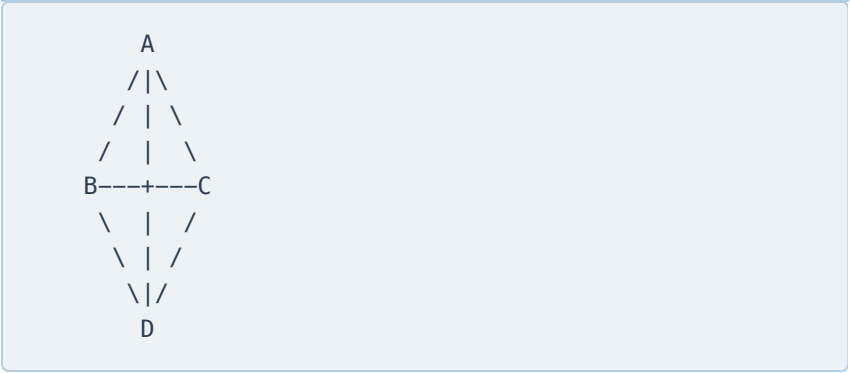
当地的居民茶余饭后喜欢散步，他们发明了一个小游戏：**能不能从某个地方出发，走过所有的七座桥，每座桥只走一次，最后回到起点？**

听起来很简单对不对？很多居民试过无数次，没有人成功。但也没有人能证明它不可能。

这个问题最终被送到了当时最伟大的数学家之一——莱昂哈德·欧拉（Leonhard Euler）的手中。

欧拉做了一件改变数学史的事：他完全忽略了岛的大小、桥的长度、路的宽窄——这些”细节”他统统不管。他只做了一件事：**把陆地抽象成”点”，把桥抽象成”线”。**

就这样，七桥问题变成了一个由 4 个点、7 条线组成的小图：



其中 A、B、C、D 是四块陆地，线条是七座桥。

然后欧拉观察到：如果你想从一个点”路过”而不停下来，你必须从一条线进来、从另一条线出去。这意味着，对于每个”途经”的点，连接到它的线条数必须是**偶数**（进一条、出一条，一进一出就是两条）。

而在哥尼斯堡的这张图里，四个点连接的线条数分别是 3、3、3、5——全部是**奇数**。所以，不可能存在一条”每座桥只走一次”的路线。

七桥问题解决了。但更重要的是，图论（Graph Theory）这个数学分支由此诞生了。

欧拉可能做梦也没想到，他在1736年随手画的这几个点和几条线，会在将近300年后成为整个互联网时代的技术基石之一。

正文

2.1 从欧拉到社交网络：图论的两百年

欧拉开创了图论之后，这个领域在很长一段时间里都是纯数学家的”玩具”。数学家们研究各种图的性质——哪些图可以一笔画完？哪些图可以被着色使得相邻的区域颜色不同？这些问题优美而深邃，但似乎与日常生活没什么关系。

然而，进入20世纪之后，事情开始发生戏剧性的变化。

1950年代–60年代：社交网络的萌芽

社会心理学家开始用”图”来研究人际关系。一个人是一个”节点”，两个人之间的友谊、 acquaintanceship、合作关系是一条”边”。1967年，哈佛大学的斯坦利·米尔格拉姆（Stanley Milgram）做了一个著名的实验：他让美国中西部的志愿者把一封信转发给一个波士顿的目标人物，规则是——你只能把信转给你认识的人。

结果令所有人震惊：信件平均经过了大约 **6次转发** 就到达了目标人物。这就是后来广为人知的”六度分隔”理论——世界上任何两个人之间，最多通过6个中间人就能联系起来。

“六度分隔”本质上是一个图论问题：在”人类社交网络”这张巨大的图中，任意两个节点之间的**最短路径**长度大约为6。

1990年代：万维网与链接图

1989年，蒂姆·伯纳斯–李（Tim Berners–Lee）发明了万维网（World Wide Web）。从图论的角度看，万维网就是一张巨大的有向图：每个网页是一个节点，每个超链接是一条从源网页指向目标

网页的有向边。

到了1998年，全球已有数亿个网页。如何从这张巨大的”网页图”中找到最有价值的页面？这是一个价值千亿美元的问题。

1998年：PageRank 改变世界

两个斯坦福大学的博士生——拉里·佩奇（Larry Page）和谢尔盖·布林（Sergey Brin）——给出了一个优雅的答案：**PageRank 算法**。

PageRank 的核心思想极其简单：一个网页如果有很多其他网页链接到它，那它很可能是一个重要的网页。更进一步，如果链接到它的网页本身也很重要，那它就更加重要。

这实际上是一个在”网页图”上计算节点重要性的算法。佩奇和布林把这个算法变成了一家公司的核心产品——这家公司就是 **Google**。

请注意这里面的联系：欧拉在1736年发明的”图”的概念 → 1998年被两个年轻人用来给整个互联网排名 → 创造了世界上最值钱的公司之一。图论的力量，由此可见一斑。

2010年代：知识图谱登场

2012年，Google 再次把”图”的概念推向了新高度。它不再仅仅把网页当作节点，而是把**人类的知识**——人物、地点、事件、概念——全部变成了节点，然后用各种关系把它们连接起来。这就是我们上一章讲过的 **知识图谱**。

如果说 PageRank 是对”网页图”的分析，那知识图谱就是对”人类知识图”的构建和分析。本质上，它们都是图论思想在不同领域的应用。

2016年–至今：图神经网络与 AlphaGo

2016年，DeepMind 的 AlphaGo 击败了围棋世界冠军李世石。很多人不知道的是，AlphaGo 在评估棋盘局面时，本质上是在处理一种特殊的”图”——棋盘上19×19的交叉点就是节点，相邻交叉点之间的连接就是边。

而近年来最火热的 AI 方向之一——**图神经网络（Graph Neural Network, GNN）**——更是直接把深度学习的能力引入了图结构数据。GNN 在分子结构预测、推荐系统、社交网络分析、药物发现等领域取得了突破性的成果。

从欧拉的铅笔到 AlphaGo 的芯片，”图”这个概念走过了将近三百年的旅程。而这段旅程，远远没有结束。

2.2 图的基本概念：你需要知道的”行话”

好了，历史讲完了。让我们系统地学习一下图论的基本概念。别怕，这些概念都非常直观——你其实已经在日常生活中见过它们无数次了，只是可能不知道它们有正式的名字。

图（Graph）

一个图 G 由两部分组成： – **顶点集（Vertex Set）** V：一组”点”，也叫**节点（Node）** – **边集（Edge Set）** E：一组”线”，每条线连接两个点

记作：G = (V, E)

就这么简单。一个图，就是一堆点和一堆线的组合。

例如：

```
V = {张三, 李四, 王五, 赵六}
E = {(张三, 李四), (李四, 王五), (王五, 赵六), (张三, 王五)}
```

这就描述了一个4个人之间的社交关系图。

顶点 / 节点（Vertex / Node）

图中的”点”。在知识图谱中，节点就是实体——一个人、一部电影、一座城市。

边（Edge）

连接两个节点的”线”。在知识图谱中，边就是关系——“朋友”“导演”“位于”。

边可以携带信息。比如在社交网络中，一条”朋友”边可以附带一个”认识年数”属性；在交通网络中，一条”公路”边可以附带”距离”属性。

度（Degree）

一个节点连接的边的数量，叫做这个节点的**度**。

在社交网络中，一个人的”度”就是他有多少个朋友。在微博/Twitter上，一个用户的”度”就是他关注了多少人 + 多少粉丝。

度这个概念之所以重要，是因为它往往反映了一个节点的重要性。在社交网络中，度最高的人就是”社交达人”或”意见领袖”；在万维网中，度最高的网页就是最权威的网页（比如维基百科的首页）。

路径（Path）

从一个节点出发，沿着边一步一步走到另一个节点，经过的这条路线就叫做**路径**。

比如，在社交网络中，”张三 → 李四 → 王五”就是一条从张三到王五的路径，长度为 2（经过了 2 条边）。

还记得”六度分隔”吗？它说的就是社交网络图中，任意两个节点之间的**最短路径**长度不超过 6。

连通性（Connectivity）

如果图中任意两个节点之间都存在至少一条路径，那么这个图叫做**连通图**。

如果一张社交网络图是连通的，意味着这个网络中的任何人都可以通过中间人联系到任何其他的人。如果不连通，就意味着存在”孤岛”——有些人和主流网络完全脱节。

在知识图谱中，连通性同样非常重要。一个高度连通的知识图谱意味着知识之间的关联很丰富，推理能力更强；而一个碎片化的图谱则意味着知识之间存在”断裂”，很多推理无法进行。

环（Cycle）

如果一条路径的起点和终点是同一个节点，那这条路径就形成了一个**环**。

比如：张三→李四→王五→张三，这就形成了一个环。

有些图有环，有些图没有环。没有环的图叫做**无环图（Acyclic Graph）**。在知识图谱的本体设计中，”is-a”（是一种）关系通常要求无环——因为”猫是一种动物，动物是一种猫”显然是荒谬的。

2.3 图的种类：远比你想象的丰富

“图”这个看似简单的概念，其实有着非常丰富的变体。不同的图有不同的特性，适用于不同的场景。让我带你认识一下最重要的几种。

无向图（Undirected Graph）vs. 有向图（Directed Graph）

这是最基本的区分。

在无向图中，边没有方向。A和B之间的边等价于B和A之间的边。典型的例子就是社交网络中的”朋友”关系——如果张三是李四的朋友，那李四自然也是张三的朋友（我们假设友谊是相互的）。

在有向图中，边有方向。从A指向B的边不等于从B指向A的边。典型的例子就是万维网——网页A可以链接到网页B，但网页B不一定链接回网页A。

知识图谱中的图是**有向图**。因为关系通常是有方向的：”(流浪地球, 导演, 郭帆)”不等于”(郭帆, 导演, 流浪地球)”。

```
无向图：                    有向图：
  A --- B                     A ---→ B
  |     |                     |      ↑
  |     |                     ↓      |
  C --- D                     C ---→ D
```

加权图（Weighted Graph）

在加权图中，每条边都有一个”权重”——一个数字，表示这条边的某种度量。

比如，在地图应用中，两个地点之间的边的权重可以是它们之间的距离或行车时间。在社交网络中，边的权重可以是两个人之间的亲密度。

```
加权图示例（城市间距离，单位：公里）：
        北京
       /    \
   1200/      \1100
     /          \
   上海 ----- 西安
      900
```

知识图谱中的边也可以带权重。比如在医学知识图谱中，”药物A 治疗 疾病B”这条边的权重可以是临床试验的有效率。

二部图（Bipartite Graph）

二部图的节点可以被分成两组，所有的边都连接不同组的节点，同组内的节点之间没有边。

最常见的例子：电商平台的”用户–商品”关系图。一组节点是用户，另一组节点是商品，边表示”用户购买了商品”。用户和用户之间没有”购买”关系，商品和商品之间也没有。

二部图在推荐系统中非常重要。你猜”猜你喜欢”是怎么来的？很大程度上就是在这张二部图上做分析——和你购买了相同商品的人还购买了什么？

多重图（Multigraph）

在简单图中，两个节点之间最多只有一条边。但在多重图中，两个节点之间可以有多条边。

比如，张三和李四之间可以同时存在”朋友”关系、”同事”关系、”邻居”关系——三条不同的边连接同样两个节点。

知识图谱天然是多重图，因为两个实体之间可以有多种不同的关系。

知识图谱：一种特殊的图

知识图谱是一种**有向、标记的（labeled）、多重图**。

- **有向**：关系有方向（主体→客体）

- **标记的**：每条边都有一个标签（关系的类型）
- **多重图**：两个实体之间可以有多条不同类型的边

同时，知识图谱的节点也可以有标签和属性（实体的类型、名字、各种特征值），这让知识图谱比普通的图携带了更丰富的语义信息。

2.4 图在计算机科学中的角色

图论虽然是数学的一个分支，但它在计算机科学中的应用之广泛，可能超出你的想象。可以说，图是计算机科学中最基础、最重要的数据结构之一。

数据结构中的图

在计算机科学的基础课程中，”图”是和”数组”“链表”“树”并列的基本数据结构。很多经典的算法都是图算法：

- **最短路径算法**（Dijkstra、A*）：GPS导航的核心。给定一张路线图，找到从A到B的最短路线。
- **最小生成树算法**（Kruskal、Prim）：用最低的成本连接所有节点。比如设计电缆网络，用最少的电缆把所有城市连接起来。
- **拓扑排序**：给定一组任务和它们之间的依赖关系，找到合理的执行顺序。这在项目管理、编译器优化中非常重要。
- **最大流算法**：在网络中找到从源到汇的最大流量。应用于物流优化、网络带宽分配等。

搜索引擎与图

我们已经提到了 PageRank。但搜索引擎和图的关系远不止于此。搜索引擎的爬虫（Crawler）本质上是在遍历万维网这张巨大的图——从一个网页出发，沿着超链接”走”到下一个网页，如此反复，直到覆盖尽可能多的网页。这个过程在图论中叫做**图的遍历（Graph Traversal）**。

社交网络与图

Facebook 在 2013 年推出了 **Graph Search**（图搜索），允许用户进行像”我朋友的朋友中住在纽约的单身女性”这样的查询。虽然这个产品最终没有成功商业化，但它的技术理念影响深远。

今天，几乎所有社交平台——微信、微博、Twitter、LinkedIn——的底层都在使用图数据库或图计算引擎来处理用户关系、内容推荐、广告投放等业务。

推荐系统与图

当你在淘宝上购物时，”猜你喜欢”背后往往是一张巨大的二部图（用户–商品图）上的算法在运作。当你在抖音上刷视频时，推荐引擎也在分析一张”用户–视频–标签”的三方图。

AI 与图

近年来最引人注目的 AI 突破，很多都与图有关：

- **AlphaFold**：DeepMind 的蛋白质结构预测模型，把蛋白质看作氨基酸残基之间的图，使用图神经网络进行预测。这项工作在 2024 年获得了诺贝尔化学奖。

- **GNN（图神经网络）**：在分子结构分析、社交网络分析、推荐系统等”图结构数据”上，GNN 已经超越了传统的深度学习方法。
- **知识图谱 + LLM**：如上一章所述，将知识图谱与大语言模型结合是当前 AI 领域的重要方向。

2.5 图的存储方式：计算机怎么”记住”一张图?

我们知道了图是什么——一堆点和一堆线。但计算机是怎么存储一张图的呢?

主要有两种方式：

邻接矩阵（Adjacency Matrix）

用一个二维表格（矩阵）来表示图。行和列分别对应节点，如果第 i 行第 j 列的值为 1，表示节点 i 和节点 j 之间有一条边；为 0 则表示没有边。

例如，对于一个4个节点的图：

```
   A  B  C  D
A [0, 1, 0, 1]
B [1, 0, 1, 0]
C [0, 1, 0, 1]
D [1, 0, 1, 0]
```

优点：查询两个节点之间是否有边，非常快（O(1)时间复杂度）。 缺点：如果图很大但很稀疏（大多数节点之间没有边），会浪费大量空间存储0。

想象一下：如果社交网络有10亿用户，邻接矩阵就需要 10亿 × 10亿 = 10^18 个格子。即使每个格子只占1个字节，也需要约100万TB的存储空间——这显然是不现实的。

邻接表（Adjacency List）

用一个列表的列表来表示图。每个节点对应一个列表，列表中存储与它相连的所有节点。

```
A: [B, D]
B: [A, C]
C: [B, D]
D: [A, C]
```

优点：对于稀疏图非常节省空间。现实世界中的图（社交网络、万维网、知识图谱）几乎全都是稀疏图。 缺点：查询两个节点之间是否有边稍慢（需要遍历列表）。

在知识图谱领域，最常用的存储方式既不是邻接矩阵也不是邻接表，而是**三元组表（Triple Store）**——直接把所有的三元组一条一条地存起来。这种方式最灵活，最适合知识图谱的”模式自由”特性。我们将在后续的章节中详细介绍各种知识图谱存储方案。

2.6 图的遍历：如何在一张图中”走路”？

有了图，接下来的问题是：怎么在图中”走”？换句话说，怎么系统地访问图中的节点？

两种最基本的遍历方式：

深度优先搜索（DFS, Depth-First Search）

策略：沿着一条路一直走到黑，走不通了再回头换一条路。

想象你在一座迷宫里。DFS 的策略是：选一个方向一直往前走，碰到死胡同就退回到最近的岔路口，选另一条路继续走。

```
从A出发进行DFS：
A → B → C → D → （回退到C） → （回退到B） → （回退到A） → E → ...
```

广度优先搜索（BFS, Breadth-First Search）

策略：像水波纹一样，一圈一圈地向外扩展。

同样是迷宫，BFS 的策略是：先看看离起点1步远的地方有什么路，再看看离起点2步远的地方，然后3步远的地方……

```
从A出发进行BFS：
A → （1步：B, E） → （2步：C, F） → （3步：D, G） → ...
```

这两种遍历方式看似简单，却是很多复杂算法的基础。在知识图谱中：

- 当你查询”和某个实体有2跳关系的所有实体”时，你实际上在做 BFS。
- 当你沿着一条关系链一路追查下去（比如追查公司的控制链条：A公司 → 母公司B → 母公司C → ……）时，你实际上在做 DFS。

2.7 图算法小课堂：三个你必须知道的算法

在正式进入知识图谱的技术细节之前，我想花一点时间介绍三个在图论和知识图谱中非常重要的算法。理解它们会帮助你更好地理解后续章节的内容。

算法一：PageRank — 谁是”最重要”的节点?

PageRank 的核心思想用一句话概括就是：**一个节点的重要性取决于有多少”重要的”节点指向它。**

这是一个”鸡生蛋、蛋生鸡”的循环定义。PageRank 的天才之处在于它把这个循环变成了一个可以计算的迭代过程：

1. 初始时，每个节点的重要性分数都一样。
2. 每一轮迭代中，每个节点把自己的分数”分配”给它指向的节点。
3. 重复这个过程，直到分数不再变化。

最终收敛的分数就是每个节点的 PageRank 值。

在知识图谱中，PageRank 的思想被广泛用于： – 实体重要性排序（哪些实体在图谱中最”核心”？） – 搜索排序（在多个匹配结果中，哪个更相关？） – 知识推荐（与当前实体最相关的那些实体是哪些？）

算法二：最短路径 — 两个节点之间的”最近”路线

给定一张图，找到从节点 A 到节点 B 的最短路径，可能是图论中最经典的问题了。

在无权图中（所有边的长度都算1），BFS 就可以找到最短路径。在加权图中，**Dijkstra 算法**是最经典的解决方案。

在知识图谱中，最短路径有很多有趣的应用：

- **概念相似度**：两个概念在知识图谱中的”距离”（最短路径长度）可以反映它们的语义相似度。”猫”和”狗”在知识图谱中的路径比”猫”和”汽车”短得多，这和人类的直觉是一致的。
- **关系推理**：找到两个实体之间的连接路径，就是找到了它们之间的”关系链”。
- **知识发现**：如果两个看似无关的实体之间存在一条很短的路径，可能暗示着某种尚未被发现的关联。

算法三：社区发现 — 图中的”小圈子”

在很多图中，节点会自然地聚集成一个个”社区”——社区内部的节点之间连接紧密，社区之间的连接稀疏。

比如社交网络中的”朋友圈”：你和你的大学同学之间联系很多，你的大学同学之间也互相认识，但你们和另一个城市的另一群人之间联系就少得多。

在知识图谱中，社区发现可以用来： – 发现相关概念的聚类（哪些疾病经常一起出现？哪些技术经常一起被使用？） – 检测图谱的质量（一个社区内的知识是否一致、是否完整？） – 辅助本体构建（自动发现实体类型之间的层次关系）

最著名的社区发现算法之一是 **Louvain 算法**，它通过优化一个叫”模块度（Modularity）”的指标来发现社区结构。

2.8 从图论到图数据库：技术栈的演进

图论是数学理论，但要让它在计算机上”跑”起来，我们需要具体的软件工具。这就引出了**图数据库**。

传统的数据库（关系数据库、文档数据库）并不是为图数据设计的。虽然你可以在关系数据库中模拟图结构（用表来存储节点和边），但当图变大、查询变复杂的时候，关系数据库的性能会急剧下降——因为每个”跳”都需要一次 JOIN 操作，多跳查询需要多次 JOIN，代价非常大。

图数据库是专门为存储和查询图数据而设计的数据库系统。它们通常有以下特点：

1. **原生图存储**：节点和边都是数据库中的”一等公民”，不需要用表来模拟。
2. **索引自由邻接（Index-Free Adjacency）**：每个节点直接存储了到相邻节点的”指针”，从一个节点跳到相邻节点是 O(1) 的操作——不需要索引查找。这是图数据库性能的关键。
3. **图查询语言**：提供专门的查询语言来操作图，比如 Cypher（Neo4j）、SPARQL（RDF 图）、Gremlin（Apache TinkerPop）。

当前主流的图数据库包括：

表 6–1

图数据库	类型	查询语言	特点
Neo4j	属性图	Cypher	最流行的图数据库，社区版免费
Apache Jena	RDF 图	SPARQL	语义网标准的实现
Amazon Neptune	属性图 + RDF	Gremlin + SPARQL	AWS 云服务
Nebula Graph	属性图	nGQL	国产开源，性能优秀
HugeGraph	属性图	Gremlin	百度开源，国产

我们将在后续的章节中详细介绍如何使用这些图数据库来存储和查询知识图谱。

2.9 一张图看懂：从欧拉到知识图谱的时间线

最后，让我们用一张”时间线”来回顾一下这段旅程：

```
1736  欧拉解决哥尼斯堡七桥问题，图论诞生
  │
  ├── 1850s  四色猜想提出（地图着色问题）
  │
  ├── 1950s  图论进入社会科学（社交网络分析）
  │
  ├── 1967   米尔格拉姆"六度分隔"实验
  │
  ├── 1989   万维网发明（全球规模的"网页图"）
  │
  ├── 1998   PageRank 算法 → Google 成立
  │
  ├── 2000s  语义网（Semantic Web）运动 → RDF/OWL 标准
  │
  ├── 2007   Neo4j 图数据库发布
  │
  ├── 2012   Google 推出 Knowledge Graph
  │
  ├── 2013   Facebook Graph Search
  │
  ├── 2016   AlphaGo（棋盘上的图 + 深度学习）
  │
  ├── 2018   图神经网络（GNN）爆发
  │
  ├── 2020   Wikidata 超过 1 亿条结构化数据
  │
  └── 2023+  知识图谱 + 大语言模型（RAG、知识增强）
```

从1736年的4个点7条线，到今天数十亿节点、数百亿条边的知识网络——图论走过了一段令人叹为观止的旅程。而你，即将开始自己的图论实践之旅。

动手练习

练习 1：手动构建一个社交网络图

画出你所在班级、公司或朋友圈的社交网络图（至少10个人）。标注以下信息：

1. 每个人的”度”是多少？
2. 谁的度最高？（这个人可能是你圈子里的”社交中心”）
3. 任意两个人之间的最短路径是多少？
4. 你能发现”社区”（小圈子）吗？

提示：用纸笔画即可，也可以使用在线工具 https://csacademy.com/app/graph_editor/

练习 2：用 Python 体验图算法

```
import networkx as nx

# 创建一个简单的社交网络
G = nx.Graph()
G.add_edges_from([
    ("Alice", "Bob"),
    ("Alice", "Charlie"),
    ("Bob", "Charlie"),
    ("Bob", "David"),
    ("Charlie", "Eve"),
    ("David", "Frank"),
    ("Eve", "Frank"),
])

# 计算每个节点的度
print("每个节点的度：")
for node, degree in G.degree():
    print(f"  {node}: {degree}")

# 计算最短路径
path = nx.shortest_path(G, "Alice", "Frank")
print(f"\nAlice 到 Frank 的最短路径: {' → '.join(path)}")

# 计算 PageRank
pagerank = nx.pagerank(G)
print("\nPageRank 排名：")
for node, score in sorted(pagerank.items(), key=lambda x:
    print(f"  {node}: {score:.4f}")

# 发现社区
communities = nx.community.greedy_modularity_communities(G
```

```
print("\n社区划分：")
for i, comm in enumerate(communities):
    print(f"  社区 {i+1}: {set(comm)}")
```

运行这段代码，观察结果。思考：

1. 度最高的节点和 PageRank 最高的节点是同一个人吗？如果不是，为什么？
2. 最短路径的长度和你预期的相符吗？
3. 社区划分的结果合理吗？

练习 3：思考题

1. 为什么万维网是一张**有向图**而不是无向图？如果所有超链接都是双向的（我链接你，你也必须链接我），互联网会变成什么样？
2. 在知识图谱中，”是……的父亲”这个关系构成的子图，是有环的还是无环的？为什么？
3. 如果把你的微信好友列表看作一张图，你觉得它是连通图吗？有没有可能存在”孤岛”？

延伸阅读

1. **《图论导引》（Introduction to Graph Theory）** — Douglas B. West
2. 经典图论教材，适合想深入了解图论数学基础的读者。

3. **Euler 的原始论文：Solutio problematis ad geometriam situs pertinentis** (1736)
4. 欧拉解决七桥问题的原始论文，拉丁文写成，有英文译本。
5. **《Networks, Crowds, and Markets》** — David Easley & Jon Kleinberg
6. 康奈尔大学的网络科学教材，用通俗的语言讲解图论在社会网络、万维网和经济学中的应用。免费在线阅读。
7. **Neo4j Graph Academy** (https://neo4j.com/graphacademy/)
8. Neo4j 官方的图数据库学习平台，提供免费课程，适合想快速上手图数据库的读者。
9. **The Anatomy of a Large-Scale Hypertextual Web Search Engine** — Page & Brin (1998)
10. PageRank 的原始论文，两位 Google 创始人的博士研究成果。

本章小结

图论从1736年欧拉的七桥问题起步，历经社交网络分析、万维网链接分析、PageRank搜索排序，一路演进为今天的知识图谱和图神经网络——它是计算机科学中生命力最持久的数学工具之一。

理解图的基本概念（节点、边、度、路径、连通性）和常见类型（有向/无向、加权、二部图），是掌握知识图谱技术的必要前提。

第三章 图谱全景：类型、应用与生态

引子：人类知识的”亚历山大图书馆”

公元前3世纪，埃及亚历山大城的托勒密王朝修建了一座宏伟的图书馆——亚历山大图书馆。它收藏了当时已知世界的大量手稿，据估计藏书量高达40万卷，是古代世界最大的知识宝库。

然而，这座图书馆最终毁于战火，无数珍贵知识从此失传。后人提起它，无不扼腕叹息。

两千多年后的今天，我们正在用一种全新的方式重建一座”数字亚历山大图书馆”——只不过这一次，它不会被火烧毁，也不会因战争而消亡。更重要的是，它不仅收藏知识，还**理解**知识。

这座数字图书馆的名字，就是**知识图谱**。

但知识图谱并不是一种统一的东西。就像图书馆有大学图书馆、公共图书馆、专业图书馆、儿童图书馆之分一样，知识图谱也有非常多的种类——它们有不同的规模、不同的领域、不同的构建方式、不同的使用场景。

今天，全球范围内已经存在数以千计的知识图谱，从覆盖人类全部知识的”通用图谱”，到只关注某个细分领域的”专业图谱”；从完全由志愿者手工编辑的开放项目，到由大型科技公司投入数亿美元打造的商业系统。

本章，我将带你做一次”知识图谱全景游”——看看这个领域有哪些重要的图谱、它们在哪里被使用、以及围绕它们形成了一个怎样的产业生态。

正文

3.1 通用知识图谱：人类知识的”大百科全书”

通用知识图谱的目标是**覆盖尽可能多的人类知识领域**——人物、地点、事件、概念、科学、文化……什么都包含。它们就像数字世界的百科全书，规模庞大、包罗万象。

以下是目前最重要的几个通用知识图谱：

Wikidata：全球协作的知识宝库

Wikidata 是维基百科基金会于2012年启动的项目，可以看作是维基百科的”结构化版本”。

维基百科用自然语言写成文章，人类读起来很舒服，但机器很难精确地”理解”。Wikidata 做的事情就是：把维基百科中的知识转化为机器可读的结构化数据——也就是我们第一章讲过的三元组。

举个例子。维基百科上关于”姚明”的文章写着：”姚明，1980年9月12日出生于上海市徐汇区，前中国职业篮球运动员……”

而 Wikidata 中，同样的信息被编码为：

```
姚明 (Q30722)
├── 出生日期 (P569): 1980年9月12日
├── 出生地 (P19): 上海
├── 职业 (P106): 篮球运动员
├── 身高 (P2048): 229 cm
├── 效力球队 (P54): 休斯顿火箭、上海东方大鲨鱼
├── 国籍 (P27): 中华人民共和国
└── ... (更多属性)
```

Wikidata 有几个非常突出的特点：

1. **完全开放**：任何人都可以编辑，任何人都可以免费使用。数据采用 CC0 协议（完全放弃版权）。
2. **规模庞大**：截至2024年，Wikidata 拥有超过 **1.1亿条** 结构化数据项（items），数亿条属性声明（statements）。
3. **多语言**：覆盖超过300种语言的标签和描述。
4. **协作编辑**：就像维基百科一样，全球的志愿者共同维护和更新。

Wikidata 是当前知识图谱领域最重要的基础设施之一。很多其他知识图谱都以 Wikidata 为基础或数据来源。

DBpedia：从维基百科中”提炼”出的图谱

DBpedia（Databasepedia）比 Wikidata 更早——它诞生于2007年，是第一个大规模的开放知识图谱。

DBpedia 的思路是：用自动化工具从维基百科的文章中提取结构化信息，然后把这些信息以 RDF 三元组的形式发布。比如，从维基百科关于姚明的文章中，自动提取出”姚明 出生地 上海”“姚明

职业 篮球运动员”等三元组。

DBpedia 的贡献在于它证明了**从非结构化文本中大规模构建知识图谱**是可行的。虽然自动提取的精度不如人工编辑，但规模效应弥补了精度的不足。

DBpedia 和 Wikidata 现在互补共存：DBpedia 侧重从维基百科文本中提取信息，Wikidata 侧重人工编辑的结构化数据。

CN-DBpedia：中文世界的知识图谱

如果你关注中文知识图谱，那 **CN-DBpedia** 是一个必须了解的项目。它由复旦大学知识工场实验室开发，是目前最大的中文通用知识图谱之一。

CN-DBpedia 包含了超过 **3000万** 实体和 **1亿** 以上的三元组，涵盖了百度百科、互动百科等中文百科的知识。

中文知识图谱的构建面临着一些独特的挑战： – 中文没有空格分隔词语，需要额外的分词步骤 – 中文的表达更加灵活多变，信息提取的难度更大 – 中文的命名实体更加多样（同一个外国人可能有多个中文译名）

CN-DBpedia 在中文自然语言处理、中文问答系统等领域有着广泛的应用。

OpenKG：开放知识图谱社区

OpenKG（开放知识图谱）是一个中文开放知识图谱社区和平台，由浙江大学等高校牵头。它的目标是成为知识图谱领域的”GitHub”——让研究者、开发者可以方便地共享、复用知识图谱资

源和工具。

OpenKG 提供了： – 多个开放的中文知识图谱数据集 – 知识图谱构建和管理的工具 – 评测基准和竞赛 – 学术论文和教程资源

对于中文知识图谱的入门者来说，OpenKG 是一个非常值得关注的资源平台。

各大科技公司的知识图谱

除了上述的开放项目，各大科技公司也有自己的内部知识图谱：

- **Google Knowledge Graph**：最大的商业知识图谱之一，支撑谷歌搜索的智能回答。估计包含超过 700 亿条事实。
- **百度知识图谱**：国内最大的中文知识图谱之一，覆盖超过 5500 亿事实，支撑百度搜索、智能问答等产品。
- **微软 Satori / Probase**：微软研究院的知识图谱项目，侧重于常识知识的获取。
- **阿里巴巴 OneGraph**：服务于电商、物流等阿里生态的知识图谱。
- **腾讯知识图谱**：应用于社交、内容推荐、广告等场景。

这些公司的知识图谱虽然不对外开放，但它们的规模和技术水平往往代表了行业的最高水准。

3.2 领域知识图谱：深耕一个行业的”专家系统”

通用知识图谱虽然包罗万象，但在专业领域的深度往往不够。比如，Wikidata 知道”高血压”是一种疾病，但它可能不知道高血压的200种细分分类、每种分类的病理机制、对应的数百种药物的详细用法——这些知识只有在**领域知识图谱**中才能找到。

领域知识图谱专注于特定的行业或学科，通常由领域专家参与构建，数据质量更高，关系更精细。

医学领域：SNOMED CT 与 UMLS

医学是知识图谱应用最成熟的领域之一，原因是：医学知识高度结构化、关系明确，而且对准确性的要求极高——一个错误的推理可能危及生命。

SNOMED CT（Systematized Nomenclature of Medicine — Clinical Terms）是目前世界上最全面的医学知识图谱，包含：

- 超过 **35万** 个医学概念（疾病、症状、手术、药物、解剖结构……）
- 超过 **100万** 条关系
- 覆盖40多个国家的医学术语标准化

SNOMED CT 被广泛应用于电子病历系统、临床决策支持系统、医学研究等领域。当一个医生在电子病历系统中输入”急性心肌梗死”时，系统能够通过 SNOMED CT 自动关联到相关症状（胸痛、心电图异常）、检查项目（心肌酶、冠脉造影）和治疗方案（溶栓、PCI手术）。

UMLS（Unified Medical Language System，统一医学语言系统）由美国国家医学图书馆（NLM）维护，它把多个医学术语标准（包括 SNOMED CT、ICD、MeSH 等）映射到一个统一的框架中，解决了不同医学系统之间术语不一致的问题。

金融领域：反欺诈与风控的核心武器

金融行业是知识图谱商业化最成功的领域之一。在金融场景中，知识图谱主要用于：

反欺诈与反洗钱： 想象一个场景：张三、李四、王五三个人分别在三家不同的银行申请贷款。单独看每个人，申请材料都"很正常"。但如果把他们放在一起看，你会发现：三个人共用同一个紧急联系人、同一家工作单位、同一个家庭住址——这很可能是一个有组织的骗贷团伙。

传统的表格数据库很难发现这种跨实体、跨机构的关联模式。但知识图谱天然擅长这件事——把所有人、所有公司、所有交易放在同一张图上，异常的关系模式一目了然。

供应链金融： 知识图谱可以建模整个供应链网络——核心企业、上游供应商、下游经销商、物流商、金融机构。通过分析图谱中的关系链条，银行可以更准确地评估中小企业的信用风险（即使该企业本身没有足够的信用历史，但如果它和一家信誉良好的核心企业有稳定的供应关系，也可以获得融资）。

投资研究： 知识图谱可以建模上市公司的股权关系、高管关系、行业关系、供应链关系，帮助投资分析师快速发现投资机会和风险。

法律领域：让机器”读懂”法律

法律知识图谱近年来发展迅速。法律领域的特点是：知识体系庞大（数万条法律法规、数百万份判例）、逻辑推理要求高（从法条到判决的逻辑链条）。

典型的法律知识图谱包括：

- **法律法规图谱**：将法律条文之间的关系（上位法/下位法、修订关系、引用关系）编码为图谱。
- **判例图谱**：从海量判例中提取”案由→事实→判决”的模式。
- **法律知识图谱**：整合法律概念、法律主体、法律关系等知识。

应用方面，法律知识图谱可以支撑： – **智能法律咨询**：用户描述案情，系统基于图谱推理可能的法律后果。 – **类案检索**：律师输入案件关键信息，系统找到相似的判例。 – **合同审查**：自动发现合同中的风险条款。

学术领域：论文、学者与机构的网络

学术知识图谱建模的是科学研究的”生态系统”——论文、学者、机构、期刊、会议、研究主题之间的关系。

最知名的学术知识图谱包括：

- **Microsoft Academic Graph（MAG）**：包含 2.5亿+ 篇论文、数千万作者和机构的信息。虽然微软已于2021年停止了该项目的维护，但其数据被 OpenAlex 等项目继承。
- **OpenAlex**：一个完全开放的学术知识图谱，由 OurResearch 组织维护，是 MAG 的”精神继承者”。

- **AMiner**：清华大学开发的学术搜索和挖掘系统，包含数千万学者和论文的关系。
- **Semantic Scholar**：由 Allen AI 研究院开发，侧重于计算机科学和生物医学领域。

学术知识图谱的应用包括：论文推荐、学术合作者发现、研究趋势分析、科研评价等。

3.3 知识图谱的六大应用场景

了解了不同类型的知识图谱之后，让我们看看它们在现实世界中具体被用在了哪里。以下六个应用场景是目前最成熟、最有商业价值的。

应用一：搜索增强（Search Enhancement）

这是知识图谱最原始、也是最成功的应用场景。

传统搜索：你搜索”苹果”，搜索引擎只能基于关键词匹配——返回所有包含”苹果”这两个字的网页。它不知道你是想吃水果还是想看手机。

知识图谱增强的搜索：搜索引擎通过知识图谱知道你之前经常搜索科技产品，推断你更可能对”苹果公司”感兴趣，于是优先展示 iPhone、Mac 等结果。同时，搜索结果页面右侧会展示苹果公司的”知识卡片”——市值、CEO、创始人、主要产品——这些信息直接从知识图谱中获取。

百度、谷歌、Bing 等主流搜索引擎都已深度集成了知识图谱。知识图谱让搜索从”字符串匹配”进化为”语义理解”。

应用二：推荐系统（Recommendation）

你可能每天都在使用推荐系统——淘宝的”猜你喜欢”、抖音的”为你推荐”、网易云音乐的”每日推荐”。

传统的推荐系统主要依赖”协同过滤”——找到和你行为相似的用户，把他们喜欢的东西推荐给你。这种方法有效，但有明显的局限：

1. **冷启动问题**：新用户没有历史行为数据，没法做协同过滤。
2. **可解释性差**：推荐系统告诉你”你可能喜欢X”，但说不清楚为什么。

知识图谱可以显著改善这些问题。通过知识图谱，推荐系统可以：

- 利用实体的属性做推荐：你喜欢《流浪地球》→ 知识图谱知道它是郭帆导演的科幻电影 → 推荐郭帆的其他电影或者其他科幻电影。
- 发现跨领域的关联：你最近搜索了”婴儿床”和”奶粉” → 知识图谱推断你可能刚有了宝宝 → 推荐母婴用品。
- 提供推荐理由：”推荐这部电影，因为它的导演和你之前喜欢的《流浪地球》是同一个导演。”

应用三：风险控制（Risk Control）

金融风控是知识图谱商业化最成功的领域之一，我们在3.2节已经讨论过一些例子。这里再补充一个更具体的场景：

信用卡欺诈检测：

一家银行的信用卡系统中，知识图谱可以实时建模以下关系：

- 持卡人 ↔ 交易记录
- 交易记录 ↔ 商户
- 商户 ↔ POS终端
- 持卡人 ↔ 联系方式（手机号、地址、设备ID）

当一个异常模式出现时——比如5个不同的持卡人在10分钟内使用同一个POS终端进行了大额交易——知识图谱可以立即发出警报，因为这个模式在图上的"形状"明显异常。

传统的关系数据库要做这种"模式检测"非常困难，因为它需要跨多张表进行多跳关联。但在图数据库和知识图谱中，这种"图模式匹配"是天然支持的。

应用四：智能问答（Question Answering）

智能问答系统允许用户用自然语言提问，系统直接返回精确答案（而不是一堆网页链接）。

知识图谱在智能问答中扮演了"知识库"的角色：

用户问："《流浪地球》的导演是谁？"

系统处理流程： 1. **意图识别**：识别出用户在问"导演"这个关系。 2. **实体链接**：将"流浪地球"链接到知识图谱中的对应实体。 3. **图谱查询**：在图谱中查找 (流浪地球, 导演, ?) 的答案。 4. **答案生成**：返回"郭帆"。

对于更复杂的问题——"出演过《流浪地球》的演员中，谁还参演了《战狼》系列？"——系统需要在知识图谱上进行多跳查询和集合运算。

应用五：药物发现（Drug Discovery）

药物研发是知识图谱最有社会价值的应用之一。

一个新药从发现到上市通常需要 **10–15年**，花费 **10–26亿美元**。知识图谱可以帮助大幅缩短这个过程：

靶点发现：知识图谱整合了基因、蛋白质、疾病、药物的多维度关系。通过分析图谱中的路径，可以发现"药物X → 作用于蛋白质Y → 蛋白质Y与疾病Z有关"，从而推断药物X可能治疗疾病Z——这就是所谓的"药物重定位"（Drug Repurposing）。

副作用预测：知识图谱可以建模药物的化学结构、靶点、代谢路径等信息，预测一个候选药物可能产生的副作用，避免在临床试验后期才发现严重问题。

药物–药物相互作用：当一个患者同时服用多种药物时，知识图谱可以帮助检测药物之间是否存在有害的相互作用。

一个成功的案例：2020年新冠疫情期间，研究者使用生物医学知识图谱快速发现了可能有效的候选药物——包括后来被广泛讨论的瑞德西韦（Remdesivir）。

应用六：企业知识管理

在大型企业中，知识往往分散在不同的系统中——CRM系统里有客户信息，ERP系统里有供应链信息，邮件系统里有沟通记录，文件服务器里有项目文档。

企业知识图谱的目标是把这些分散的知识整合成一张统一的知识网络：

- **员工**：姓名、部门、技能、项目经历
- **客户**：公司名、行业、合作历史、联系人
- **项目**：名称、成员、阶段、关联客户
- **文档**：标题、作者、标签、关联项目

有了这样的图谱，员工可以：–“找到所有做过金融行业项目的同事”–“查看客户A相关的所有文档和沟通记录”–“推荐最适合这个项目的专家”

3.4 知识图谱产业链：上游、中游与下游

了解了知识图谱的类型和应用之后，让我们从产业的角度来看看整个生态是怎么运转的。

知识图谱的产业可以分为**上游、中游、下游**三个环节，就像一条河流的上游（水源）、中游（河道）、下游（灌溉区域）一样。

上游：数据源——知识图谱的”原料”

知识图谱需要数据来”喂养”。上游的数据源主要包括：

1. **结构化数据**：已经以表格或数据库形式存在的数据。比如公司内部的CRM数据库、政府公开的数据集。这是最”干净”的数据源，可以直接转化为三元组。
2. **半结构化数据**：有一定结构但不完全规范的数据。比如网页中的表格、百科词条中的信息框（infobox）、JSON格式的API数据。需要通过信息提取技术来转化。

3. **非结构化数据**：纯文本、图片、音视频等。这是最丰富但也最难处理的数据源。需要用 NLP 技术（命名实体识别、关系提取、事件抽取）来从中提取结构化知识。

在 AI 时代，**大语言模型**正在深刻地改变上游数据处理的格局。以前需要专门的 NLP 团队开发的提取管道，现在很多可以直接用大模型来完成——比如用 GPT 来从文本中提取三元组。这大大降低了知识图谱构建的门槛。

中游：平台与工具——知识图谱的"工厂"

中游是知识图谱的"生产车间"——把原始数据加工成可用的知识图谱。这个环节包括：

知识建模（Ontology Engineering）： 设计知识图谱的"蓝图"——定义实体类型、关系类型、属性、约束规则。这是最"烧脑"的环节，需要领域专家和知识工程师的密切合作。

知识获取（Knowledge Acquisition）： 从数据源中提取三元组。主要方法包括： – 基于规则的方法：编写提取规则（正则表达式、模板匹配等） – 基于机器学习的方法：训练模型自动识别实体和关系 – 基于大模型的方法：使用 LLM 进行 zero-shot 或 few-shot 提取 – 众包方法：让大量志愿者参与标注（Wikidata 就采用这种方式）

知识融合（Knowledge Fusion）： 把来自不同数据源的知识合并到同一张图谱中。核心挑战是**实体对齐（Entity Alignment）**——判断两个不同数据源中的实体是否指向同一个东西。比如，"姚

明”（中文维基百科）和”Yao Ming”（英文维基百科）是同一个人，但”苹果”（水果）和”苹果”（公司）是不同的实体。

知识存储（Knowledge Storage）： 选择合适的方式来存储图谱数据。主要选项包括： – **RDF 三元组存储**：如 Apache Jena、Virtuoso、Blazegraph – **属性图数据库**：如 Neo4j、Nebula Graph、HugeGraph – **混合存储**：结合关系数据库和图数据库的优势

知识推理（Knowledge Reasoning）： 基于已有知识推理出新的知识。比如： – 如果知道”姚明出生于上海”和”上海位于中国”，可以推理出”姚明出生于中国” – 如果知道”A是B的子公司”和”B是C的子公司”，可以推理出”A是C的孙公司”

下游：应用与服务——知识图谱的”价值变现”

下游是知识图谱面向最终用户的环节，把”知识”转化为”价值”。主要包括：

知识图谱即服务（KGaaS）： 一些公司提供知识图谱的云服务，用户可以直接调用 API 来查询和推理，不需要自己构建和维护图谱。比如百度的知识图谱 API、Google 的 Knowledge Graph Search API。

垂直行业解决方案： 把知识图谱技术与特定行业的需求结合，形成端到端的解决方案。比如金融行业的反欺诈系统、医疗行业的临床辅助决策系统、法律行业的智能合同审查系统。

知识图谱增强的大模型应用： 这是当前最火热的方向。把知识图谱作为大语言模型的”外挂知识库”，通过 RAG 等技术为大模型提供准确的事实，减少”幻觉”。

3.5 知识图谱的技术标准：RDF 与属性图

在产业生态的介绍之后，有必要提一下知识图谱领域的两个主要技术标准。这两种”流派”在数据模型上有着根本性的差异。

RDF（Resource Description Framework）

RDF 是 W3C（万维网联盟）制定的标准，起源于”语义网”运动。它的核心思想是：

- 所有知识都表示为**三元组**：（主体, 谓词, 客体）
- 主体、谓词、客体都用 **URI**（统一资源标识符）来标识
- 数据交换采用标准格式：Turtle、RDF/XML、JSON-LD 等

一条 RDF 三元组看起来像这样：

```
@prefix ex: <http://example.org/> .
@prefix schema: <http://schema.org/> .

ex:WanderingEarth schema:director ex:GuoFan .
ex:WanderingEarth schema:genre ex:ScienceFiction .
ex:GuoFan schema:jobTitle "导演" .
```

RDF 的优点是标准化程度高、互操作性好（不同的系统可以方便地交换数据），有成熟的推理引擎（基于 OWL 本体逻辑）。缺点是相对复杂、性能在某些场景下不如属性图。

属性图（Property Graph）

属性图是另一种流行的图数据模型，被 Neo4j 等图数据库采用。它的特点是：

- 节点和边都可以有**属性**（key-value 对）
- 节点可以有**标签**（Label）来表示类型
- 边有**类型**和**方向**

用 Cypher 查询语言（Neo4j 的查询语言）表示同样的电影知识：

```
CREATE (:Movie {title: "流浪地球", year: 2019})
       -[:DIRECTED_BY]->
       (:Person {name: "郭帆", occupation: "导演"})

CREATE (:Movie {title: "流浪地球"})
       -[:GENRE]->
       (:Genre {name: "科幻"})
```

属性图的优点是直观、灵活、查询性能好。缺点是标准化程度不如 RDF，不同的属性图数据库之间互操作性较差。

两种流派的融合趋势

近年来，RDF 和属性图两种流派正在逐渐融合。W3C 正在制定 **RDF-star** 标准，允许在 RDF 三元组上直接添加属性（类似属性图的功能）。而 Neo4j 等属性图数据库也在增加对 RDF 和 SPARQL 的支持。

对于初学者来说，了解两种流派的基本区别就够了。在实际项目中，选择哪种主要取决于你的具体需求：如果你需要标准化的数据交换和逻辑推理，选 RDF；如果你需要高性能的图查询和直观的数据模型，选属性图。

3.6 知识图谱生态中的开源项目

最后，让我为你梳理一下知识图谱生态中最重要的开源项目。这些项目是你后续动手实践时会频繁接触到的”工具箱”。

知识图谱构建工具：

表 7–1

项目	功能	特点
spaCy	NLP 管线	命名实体识别、依存分析
HanLP	中文 NLP	支持中文分词、NER、关系抽取
DeepKE	知识抽取	浙大开发，支持实体/关系/属性抽取
OpenIE	开放信息抽取	从文本中提取开放域三元组

图数据库与存储：

表 7–2

项目	类型	特点
Neo4j	属性图数据库	最流行的图数据库，社区版免费
Apache Jena	RDF 存储	支持 SPARQL 查询
Nebula Graph	属性图数据库	国产，分布式，高性能
HugeGraph	属性图数据库	百度开源，支持 Gremlin

图计算与分析：

表 7–3

项目	功能	特点
NetworkX	Python 图分析库	入门必备，功能全面
PyTorch Geometric	图神经网络库	GNN 研究和应用的首选
Apache GraphX	Spark 图计算	适合大规模图计算

知识图谱应用框架：

表 7–4

项目	功能	特点
LangChain	LLM 应用框架	支持知识图谱增强的 RAG
LlamaIndex	知识索引框架	支持从知识图谱中检索
DGL	深度学习图库	亚马逊开源，GNN 训练

动手练习

练习 1：探索 Wikidata

访问 https://www.wikidata.org ，搜索你感兴趣的一个实体（比如你最喜欢的电影、运动员或科学家）。

1. 观察这个实体页面有哪些属性（properties）？
2. 这个实体有多少个三元组（statements）？
3. 点击一个属性值，跳转到另一个实体，观察知识图谱的”链接”特性。
4. 尝试使用 Wikidata 的 SPARQL 端点（https://query.wikidata.org），运行以下查询，找出所有获得过诺贝尔物理学奖的中国出生的人：

```
SELECT ?person ?personLabel WHERE {
  ?person wdt:P166 wd:Q38104 .  # 获得诺贝尔物理学奖
  ?person wdt:P19 ?birthplace .  # 出生地
  ?birthplace wdt:P17 wd:Q148 .  # 出生地所在国家：中国
  SERVICE wikibase:label { bd:serviceParam wikibase:langua
}
```

练习 2：设计一个领域知识图谱

选择一个你熟悉的领域（比如你工作的行业），设计一个小型的领域知识图谱：

1. 定义本体：列出3–5种实体类型和5–8种关系类型

2. 编写15–20个三元组
3. 画出图谱的可视化图
4. 思考：这个图谱可以用来回答什么样的问题？

提示：如果你是一名教师，可以设计一个”课程知识图谱”——包含学生、课程、教师、知识点、教材等实体，以及”选修”“教授”“包含”“使用”等关系。

练习 3：用 SPARQL 查询 RDF 数据

安装 Apache Jena（一个 RDF 框架），然后：

1. 创建一个 Turtle 文件，包含你在第一章练习中编写的三元组
2. 使用 Jena 的 `sparql` 命令行工具执行以下查询：

```
# 查询所有电影及其导演
PREFIX ex: <http://example.org/>

SELECT ?movie ?director
WHERE {
  ?movie ex:导演 ?director .
}
```

```
# 查询郭帆导演过的所有电影
PREFIX ex: <http://example.org/>

SELECT ?movie
WHERE {
  ?movie ex:导演 ex:郭帆 .
}
```

1. 自己写一个查询：找出所有职业为”演员”的人

延伸阅读

1. **Wikidata 官网与文档** (https://www.wikidata.org)
2. 直接体验全球最大的开放知识图谱，学习它的编辑规则和查询方法。
3. **《Knowledge Graphs: Methodology, Applications and Selected Use Cases》** — Fensel 等 (2020)
4. 系统介绍知识图谱方法论和产业应用的书籍。
5. **CN-DBpedia 论文与数据集**
6. 复旦大学知识工场实验室发表的一系列关于中文知识图谱构建的论文。
7. 数据集下载：http://kw.fudan.edu.cn/cndbpedia
8. **OpenKG 平台** (http://openkg.cn)
9. 中文开放知识图谱社区，包含多个中文知识图谱数据集和工具。
10. **Neo4j Graph Data Science 文档**
11. Neo4j 的图数据科学库文档，包含丰富的图算法教程。
12. https://neo4j.com/docs/graph-data-science/
13. **SNOMED CT 浏览器** (https://browser.ihtsdotools.org)

14. 在线浏览全球最大的医学知识图谱，了解医学术语的层次结构和关系。

本章小结

知识图谱的世界远比我们想象的丰富——从 Wikidata 这样覆盖全人类知识的通用图谱，到 SNOMED CT 这样深耕医学领域的专业图谱；从搜索增强、智能推荐到药物发现、金融风控，知识图谱的应用已经渗透到了各行各业。理解”上游数据→中游平台→下游应用”的产业链结构，以及 RDF 与属性图两大技术标准的区别，是进入知识图谱实践领域的必要准备。

第4章 RDF、OWL与SPARQL：语义网三件套

> *在 RDF 中，我们使用的主要是 URL 形式的 URI。但要注意，RDF 中的 URL **不一定要能访问**

—— 它只是一个标识符。

`http://example.org/person/刘备` 这个 URL 实际上在浏览器里什么都打不开，但这完全不影响它在 RDF 中作为标识符的功能。

引子

你有没有想过这样一个问题：互联网上有数十亿个网页，人类可以轻松地浏览和理解它们，但机器却只能看到一堆杂乱无章的 HTML 标签。比如，当你在浏览器里看到”刘备，字玄德，蜀汉开国皇帝”，你立刻就能理解这三者之间的关系——刘备是一个人，玄德是他的字，蜀汉是他建立的政权。但如果让一台机器来理解这段话呢？它看到的只是一串字符，完全不知道”刘备”和”玄德”是同一个人，更不知道”蜀汉”是一个政权。

1999 年，万维网之父 Tim Berners-Lee 提出了一个宏大的愿景：**语义网（Semantic Web）**。他希望互联网上的信息不仅能被人类阅读，还能被机器理解和推理。为了实现这个愿景，需要一套标准的”语言”来描述世界上的事物及其关系。这套语言，就是我们今天要讲的”语义网三件套”——**RDF、OWL 和 SPARQL**。

打个比方：如果说知识图谱是一栋大楼，那么 RDF 就是砖块（数据的基本单元），OWL 就是建筑图纸（定义数据的结构和规则），而 SPARQL 就是电梯（帮你在数据大楼里快速找到想要的信息）。

准备好了吗？让我们一块一块地来搭建这座知识的大楼。

正文

RDF：用三元组描述整个世界

什么是 RDF?

RDF，全称 Resource Description Framework（资源描述框架），是 W3C（万维网联盟）于 1999 年发布的一种数据模型标准。它的核心思想极其简单：**用”主语–谓语–宾语”的三元组来描述一切知识。**

没错，就这么简单。就像我们小时候学造句一样：

表 8–1

主语（Subject）	谓语（Predicate）	宾语（Object）
刘备	字	玄德
诸葛亮	效力于	蜀汉
关羽	使用的武器	青龙偃月刀

每一行就是一个**三元组（Triple）**，它是 RDF 世界中最小的知识单元。

三元组的组成部分

让我们更深入地理解三元组的三个部分：

1. 主语（Subject）

主语是我们要描述的那个”东西”。在 RDF 中，主语必须是一个**资源（Resource）**，而资源通过 **URI（统一资源标识符）** 来唯一标识。

```
<http://example.org/person/刘备>
```

为什么要用 URI？因为”刘备”这个名字可能有重名，但一个 URI 就像身份证号一样，是全球唯一的。这和网页用 URL 来定位是一样的道理——只不过 URI 标识的是”概念”，而 URL 标识的是”网页地址”。

> **URI vs URL vs URN，傻傻分不清？**
>
> 这三个缩写经常让人困惑，简单区分一下： – **URI（统一资源标识符）**：最大的概念，任何能唯一标识一个资源的东西都是 URI – **URL（统一资源定位符）**：URI 的子集，特指那些可以用来”定位”资源的 URI（比如 `https://www.baidu.com` ） – **URN（统一资源名称）**：URI 的另一个子集，特指那些给资源一个”永久名称”的 URI（比如 `urn:isbn:9787111635` ）

除了 URI，RDF 还支持**空白节点（Blank Node）**作为主语。空白节点没有全局唯一的标识符，通常用于描述一些不需要被外部引用的中间结构。你可以把它想象成一个”匿名变量”。

2. 谓语（Predicate）

谓语描述主语和宾语之间的关系，它本身也是一个 URI：

```
<http://example.org/ontology/字>
```

在实际应用中，人们通常使用已有的标准词汇表（vocabulary）中的谓语，比如 Dublin Core（都柏林核心词汇）里的 `dc:title`、`dc:creator` 等，或者 FOAF（Friend of a Friend）里的 `foaf:name`、`foaf:knows` 等。

3. 宾语（Object）

宾语可以是两种类型： – **另一个资源（URI）**：比如 `<http://example.org/person/诸葛亮>` – **字面值（Literal）**：比如 `"玄德"`、`"181"`（数字）、`"181-03-15"^^xsd:date`（带类型的值）

一个简单的 RDF 数据集

让我们用三国演义的人物来构造一小批 RDF 数据：

```
<http://example.org/person/刘备> <http://example.org/ontol
<http://example.org/person/刘备> <http://example.org/ontol
<http://example.org/person/刘备> <http://www.w3.org/1999/0

<http://example.org/person/诸葛亮> <http://example.org/onto
<http://example.org/person/诸葛亮> <http://example.org/onto
<http://example.org/person/诸葛亮> <http://example.org/onto

<http://example.org/person/关羽> <http://example.org/ontol
<http://example.org/person/关羽> <http://example.org/ontol
```

你可能会说：这也太啰嗦了吧！别急，这就引出了下一个话题——RDF 的序列化格式。

RDF 序列化格式：让数据更好读

RDF 是一种**抽象的数据模型**，它需要被”序列化”成具体的文本格式才能存储和传输。就像”一首歌”是抽象的，但你可以把它存成 MP3、FLAC 或 WAV 文件。RDF 有多种序列化格式，我们来逐一认识它们。

1. Turtle（海龟语法）——最推荐的格式

Turtle（Terse RDF Triple Language）是目前最常用、最人性化的 RDF 序列化格式。它通过前缀缩写和语法糖，把冗长的 URI 变得可读性极强。

同样的三国数据，用 Turtle 写出来是这样的：

```
@prefix ex: <http://example.org/ontology/> .
@prefix person: <http://example.org/person/> .
@prefix kingdom: <http://example.org/kingdom/> .
@prefix weapon: <http://example.org/weapon/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

person:刘备  ex:字       "玄德" ;
           ex:建立     kingdom:蜀汉 ;
           rdf:type    ex:人物 .

person:诸葛亮  ex:字     "孔明" ;
             ex:号     "卧龙" ;
             ex:效力于  kingdom:蜀汉 .

person:关羽  ex:字       "云长" ;
           ex:使用武器  weapon:青龙偃月刀 .
```

看到了吗？通过 `@prefix` 定义命名空间前缀后，URI 变得非常简洁。而且当同一个主语有多个谓语-宾语对时，可以用分号 `;` 连接，不必重复主语。

> **小贴士**
>
> 在 Turtle 中，`.` 表示一个三元组的结束，`;` 表示同一个主语的下一个谓语-宾语对，`,` 表示同一个主语和谓语下的多个宾语。

2. N-Triples——最简单粗暴的格式

N-Triples 是 RDF 最朴素的表达形式：每行一个三元组，没有前缀缩写，没有语法糖。

```
<http://example.org/person/刘备> <http://example.org/ontol
<http://example.org/person/刘备> <http://example.org/ontol
<http://example.org/person/诸葛亮> <http://example.org/onto
```

N-Triples 的好处是极其简单、易于机器解析，适合大规模数据的批处理。缺点嘛……你看到了，人类阅读起来眼睛很累。

3. JSON-LD——Web 开发者的最爱

JSON-LD（JSON for Linking Data）把 RDF 数据嵌入到 JSON 格式中，对于熟悉 Web 开发的程序员来说非常友好：

```
{
  "@context": {
    "ex": "http://example.org/ontology/",
    "person": "http://example.org/person/",
    "kingdom": "http://example.org/kingdom/"
  },
  "@id": "person:刘备",
  "@type": "ex:人物",
  "ex:字": "玄德",
  "ex:建立": {
    "@id": "kingdom:蜀汉"
  }
}
```

Google 搜索引擎的结构化数据标记（Schema.org）就大量使用了 JSON-LD 格式。如果你在做 SEO 或者 Web 应用开发，JSON-LD 可能是你最常用的格式。

4. RDF/XML——元老级格式

RDF/XML 是最早的 RDF 序列化格式，使用 XML 语法。虽然现在已经不是首选，但在一些老系统和学术文献中仍然大量存在：

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-synta
         xmlns:ex="http://example.org/ontology/">
  <rdf:Description rdf:about="http://example.org/person/刘
    <ex:字>玄德</ex:字>
    <ex:建立 rdf:resource="http://example.org/kingdom/蜀汉"
  </rdf:Description>
</rdf:RDF>
```

坦白说，RDF/XML 的冗长和难读是很多人对 RDF 产生偏见的原因之一。如果你是从 RDF/XML 开始接触语义网的，我完全理解你的痛苦。请相信，Turtle 格式会给你带来全新的体验。

格式对比速查表

表 8-2

格式	可读性	机器解析	适用场景	文件后缀
Turtle	★★★★★	★★★★	手写、教学、配置文件	`.ttl`
N-Triples	★★	★★★★★	大规模数据处理、流式传输	`.nt`
JSON-LD	★★★★	★★★★★	Web 应用、API、SEO	`.jsonld`
RDF/XML	★	★★★★	遗留系统、学术文献	`.rdf` / `.xml`

三元组构成的图

当我们把大量的 RDF 三元组放在一起时，它们自然形成了一个**有向标记图（Directed Labeled Graph）**。每个 URI 或字面值是一个节点，每个谓语是一条从主语指向宾语的有向边。

想象一下我们的三国数据：

```
[刘备] —字—→ "玄德"
[刘备] —建立—→ [蜀汉]
[刘备] —类型—→ [人物]
[诸葛亮] —字—→ "孔明"
[诸葛亮] —效力于—→ [蜀汉]
[关羽] —字—→ "云长"
[关羽] —使用武器—→ [青龙偃月刀]
```

当你把这些三元组画成图的时候，你会发现：刘备、诸葛亮、关羽都通过”效力于”关系指向了蜀汉，一个隐含的”同僚”关系就浮出水面了。这就是图模型的魅力——**关系本身就是数据**，而且多个实体之间的关系会自动形成一个网络。

在关系数据库中，你需要用外键和 JOIN 来手动拼接这些关系。而在 RDF 图中，关系是”一等公民”，它们和实体一样被显式地存储和查询。

RDFS：RDF 的轻量级扩展

在进入 OWL 之前，我们先认识一下 **RDFS（RDF Schema）**。RDFS 是介于纯 RDF 和 OWL 之间的一个轻量级本体语言，它提供了一些基本的建模能力：

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex: <http://example.org/ontology/> .

# 定义类的层次
ex:武将  rdfs:subClassOf  ex:人物 .
ex:文官  rdfs:subClassOf  ex:人物 .
ex:君主  rdfs:subClassOf  ex:人物 .

# 定义属性的层次
ex:结拜兄弟  rdfs:subPropertyOf  ex:人际关系 .

# 定义属性的定义域和值域
ex:效力于  rdfs:domain  ex:人物 ;
           rdfs:range   ex:势力 .

# 给类和属性加人类可读的标签和注释
ex:人物  rdfs:label    "人物"@zh, "Person"@en ;
         rdfs:comment   "三国时期的人物"@zh .
```

RDFS 的核心功能包括： – **rdfs:subClassOf**：定义类的继承关系。如果”武将”是”人物”的子类，那么所有武将自己都是人物。 – **rdfs:subPropertyOf**：定义属性的继承关系。 – **rdfs:domain / rdfs:range**：约束属性的主语类型和宾语类型。 – **rdfs:label / rdfs:comment**：给资源添加人类可读的标签和注释。

RDFS 很轻量，适合简单的分类体系和轻量级本体。但如果你的需求更复杂——比如需要定义”不相交类”“基数约束”“逆属性”等——那就需要 OWL 上场了。

OWL：给数据立规矩

为什么需要 OWL?

RDF 给了我们描述世界的能力，但它有一个局限：**无法表达复杂的约束和规则**。

比如，用 RDF 你可以说”刘备是一个人”，但你无法表达： – “一个人只能有一个出生日期”（函数型属性） – “如果 A 是 B 的父亲，那 B 就是 A 的孩子”（逆属性） – “人和城市是两个互不相交的类”（不相交类） – “蜀汉的君主必须是蜀汉的人物”（类约束）

OWL（Web Ontology Language） 就是来解决这个问题的。它在 RDF 的基础上增加了丰富的表达能力，让我们可以定义**本体（Ontology）**——一个领域内所有概念、关系和规则的集合。

本体是什么?

“本体”这个词听起来很哲学（确实在哲学里它是个很深的概念），但在知识图谱领域，你可以把它理解为：**一份详细定义了”这个领域里有哪些概念、它们之间是什么关系、有什么约束”的说明书**。

举个例子，如果我们要构建一个”三国演义”的本体，它可能包含：

类（Classes）： – `人物`（Person） – `势力`（Kingdom） – `武器`（Weapon） – `战役`（Battle）

属性（Properties）： – `字`（数据型属性——连接人物和字符串） – `效力于`（对象型属性——连接人物和势力） – `使用武器`（对象型属性——连接人物和武器）

约束（Restrictions）： – 每个人物最多效忠于一个势力（基数约束） – `效力于`的逆属性是`拥有臣属` – `人物`和`战役`是不相交的类

OWL 的核心构件

1. 类（Classes）

OWL 中的类代表领域中的概念。你可以定义类的层次结构（继承关系）：

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex: <http://example.org/ontology/> .

ex:人物  rdf:type  owl:Class .

ex:武将  rdf:type  owl:Class ;
        rdfs:subClassOf  ex:人物 .

ex:文官  rdf:type  owl:Class ;
        rdfs:subClassOf  ex:人物 .

ex:君主  rdf:type  owl:Class ;
        rdfs:subClassOf  ex:人物 .
```

这里我们定义了”武将”“文官”“君主”都是”人物”的子类。如果诸葛亮被声明为一个”文官”，推理引擎就能自动推导出他也是一个”人物”。

2. 属性（Properties）

OWL 把属性分为两大类：

对象型属性（Object Property）——连接两个资源：

```
ex:效力于  rdf:type  owl:ObjectProperty ;
          rdfs:domain  ex:人物 ;      # 主语必须是"人物"
          rdfs:range   ex:势力 .      # 宾语必须是"势力"

ex:拥有臣属  rdf:type  owl:ObjectProperty ;
            owl:inverseOf  ex:效力于 .  # 互为逆属性
```

数据型属性（Datatype Property）——连接资源和一个字面值：

```
ex:字  rdf:type  owl:DatatypeProperty ;
      rdfs:domain  ex:人物 ;
      rdfs:range   xsd:string .

ex:出生年份  rdf:type  owl:DatatypeProperty ;
            rdfs:domain  ex:人物 ;
            rdfs:range   xsd:integer .
```

`rdfs:domain` 和 `rdfs:range` 相当于给属性加了约束：如果有人写了一句 `<http://example.org/kingdom/蜀汉> ex:字 "xxx"`（意思是”蜀汉的字是xxx”），推理引擎就能发现这违反了约束

——因为 `ex:字` 的 domain 是 `ex:人物`，而不是 `ex:势力`。

3. 约束（Restrictions）

OWL 的约束功能非常强大，你可以定义非常精细的规则：

```
# 每个人物最多有一个"字"（函数型属性）
ex:字  rdf:type  owl:FunctionalProperty .

# 一个人物必须属于且仅属于一个势力
ex:忠臣  rdf:type  owl:Class ;
        owl:equivalentClass  [
            rdf:type  owl:Restriction ;
            owl:onProperty  ex:效力于 ;
            owl:cardinality  1           # 恰好一个
        ] .

# "蜀汉五虎将"是"武将"的子类，且必须效力于蜀汉
ex:蜀汉五虎将  rdf:type  owl:Class ;
               rdfs:subClassOf  ex:武将 ,
                   [
                       rdf:type  owl:Restriction ;
                       owl:onProperty  ex:效力于 ;
                       owl:hasValue  kingdom:蜀汉   # 值必须
                   ] .
```

4. 推理（Reasoning）——OWL 最强大的能力

OWL 最迷人的地方在于**推理（Reasoning）**。推理引擎可以根据你定义的规则，自动推导出新的、没有显式写出的知识。

让我用一个更详细的例子来说明推理的威力。假设我们定义了以下本体：

```
# 定义类和层次
ex:武将  rdfs:subClassOf  ex:人物 .
ex:五虎上将  rdfs:subClassOf  ex:武将 .

# 定义属性和约束
ex:效力于  rdf:type  owl:ObjectProperty ;
           rdfs:domain  ex:人物 ;
           rdfs:range   ex:势力 .

ex:拥有臣属  rdf:type  owl:ObjectProperty ;
             owl:inverseOf  ex:效力于 .

ex:统帅  rdf:type  owl:ObjectProperty ;
         rdfs:subPropertyOf  ex:效力于 .
```

然后我们写入以下事实数据：

```
# 事实数据
person:关羽  rdf:type  ex:五虎上将 .
person:关羽  ex:统帅  kingdom:蜀汉 .
```

推理引擎的推导过程（逐步）：

表 8–3

步骤	推导出的知识	使用的规则
1	关羽 是 武将	五虎上将 是 武将 的子类
2	关羽 是 人物	武将 是 人物 的子类
3	关羽 效力于 蜀汉	统帅 是 效力于 的子属性
4	蜀汉 拥有臣属 关羽	拥有臣属 是 效力于 的逆属性

看到了吗？我们只写了 2 条事实数据，推理引擎却帮我们推导出了 4 条隐含知识。在一个包含百万级三元组的大型知识图谱中，这种推理能力可以帮你发现大量隐藏的关联。

推理的分类：

在实际应用中，推理可以分为两种模式：

- **前向推理（Forward Chaining）**：在数据加载时就预先计算好所有隐含知识，存储在数据库中。查询时速度很快，但数据更新时需要重新计算。适合读多写少的场景。
- **后向推理（Backward Chaining）**：在查询时实时计算隐含知识。不需要额外的存储空间，但查询速度相对较慢。适合数据频繁更新的场景。

Apache Jena 内置了多种推理器，你可以选择不同的推理级别：

```
// Jena 推理器选择
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();  //
Reasoner reasoner = ReasonerRegistry.getRDFSReasoner(); //
Reasoner reasoner = ReasonerRegistry.getTransitiveReasoner
```

常见新手陷阱

在学习 RDF 和 OWL 的过程中，有几个常见的坑值得提前了解：

陷阱 1：URI 不等于 URL

很多初学者会尝试在浏览器里打开 RDF 数据中使用的 URI，然后发现页面不存在。记住：RDF 中的 URI 只是标识符，不是可以访问的网页地址。当然，如果你使用了 HTTP URI 并且提供了内容协商（Content Negotiation），那么同一个 URI 可以同时作为标识符和网页地址——这就是 Linked Data 的理念。

陷阱 2：RDF 的开放世界假设

RDF 和 OWL 采用**开放世界假设（Open World Assumption）**，这和关系数据库的**封闭世界假设**完全不同。

举个例子：如果你的 RDF 数据中没有”关羽效力于曹魏”这条三元组，在封闭世界假设下，你可以认为”关羽没有效力于曹魏”。但在开放世界假设下，你只能说”我不知道关羽是否效力于曹魏”——也许有某个你不知道的数据源包含了这条信息。

这个区别在使用推理引擎时尤其重要：推理引擎不会因为某条信息”缺失”就认为它是假的。

陷阱 3：Turtle 中的分号和逗号

初学者经常搞混 `;` 和 `,` 的用法。记住这个口诀：– `;`（分号）：主语不变，换谓语 – `,`（逗号）：主语和谓语都不变，换宾语

```
# 分号：主语"刘备"不变，换谓语
person:刘备  ex:字    "玄德" ;    # 谓语1
           ex:建立   kingdom:蜀汉 .  # 谓语2

# 逗号：主语"刘备"和谓语"结拜兄弟"都不变，换宾语
person:刘备  ex:结拜兄弟  person:关羽 ,  # 宾语1
                        person:张飞 .  # 宾语2
```

陷阱 4：SPARQL 变量的作用域

在 SPARQL 中，WHERE 子句里定义的变量名是局部的，只在当前查询中有效。但如果你在多个查询中使用相同的变量名（比如都叫 `?x`），要注意它们之间没有关联。每个查询是独立的。

另外，FILTER 的作用域是它所在的那个图模式（花括号内的部分）。如果你在 OPTIONAL 块里使用 FILTER，要注意它只会过滤 OPTIONAL 内部的结果，不会影响外层查询。

OWL 的三个子语言

OWL 根据表达能力和计算复杂度分了三个级别：

表 8–4

级别	名称	特点	适用场景
最轻	OWL Lite	简单的分类层次和约束	轻量级本体、分类体系
中等	OWL DL	完整的表达能力，保证可判定性	大多数本体应用
最强	OWL Full	最大灵活性，不保证可计算性	需要最大表达力的场景

> **小贴士**
>
> 对于入门者，从 OWL DL 开始就好。它在表达能力和计算可行性之间取得了很好的平衡。OWL Full 虽然最灵活，但可能导致推理引擎无法给出确定答案（不可判定问题）。

SPARQL：向知识提问的艺术

初识 SPARQL

有了 RDF 来存储数据，有了 OWL 来定义规则，我们还需要一种语言来**查询**这些数据。这就像你有了数据库和表结构之后，还需要 SQL 一样。

SPARQL（发音类似”sparkle”， sparkling 的缩写，全称 **SPARQL Protocol and RDF Query Language**）就是 RDF 世界的 SQL。它是一种标准化的图查询语言，由 W3C 于 2008 年发布（SPARQL 1.0），2013 年升级到 1.1 版本。

SPARQL 和 SQL 有一个根本性的不同：SQL 操作的是二维表格，而 SPARQL 操作的是**图（三元组构成的图）**。SPARQL 查询的核心思想是**图模式匹配**——你描述一个图的形状，SPARQL 引擎帮你在知识图谱中找到所有匹配的子图。

SPARQL 端点（Endpoint）

在关系数据库中，你通常直接连接到数据库服务器执行 SQL。而在 RDF 的世界里，查询通常通过一个叫做 **SPARQL 端点（SPARQL Endpoint）** 的 HTTP 服务来进行。

SPARQL 端点是一个 Web 服务，它接受 HTTP 请求，执行 SPARQL 查询，然后返回结果（通常是 JSON 或 XML 格式）。这意味着你可以用任何编程语言（Python、JavaScript、Java 等）通过 HTTP 请求来查询 RDF 数据，而不需要安装特定的数据库驱动。

很多公开的知识图谱都提供了免费的 SPARQL 端点，你可以直接在浏览器中尝试：

表 8–5

数据集	SPARQL 端点 URL	说明
DBpedia	https://dbpedia.org/sparql	维基百科的结构化版本
Wikidata	https://query.wikidata.org/sparql	维基媒体基金会的知识图谱
UniProt	https://sparql.uniprot.org/sparql	蛋白质知识库
LinkedMDB	http://data.linkedmdb.org/sparql	电影数据库

SPARQL 的四种查询形式

1. SELECT——最常用的查询

SELECT 查询返回一个结果表格，和 SQL 的 SELECT 非常相似：

```
PREFIX ex: <http://example.org/ontology/>
PREFIX person: <http://example.org/person/>
PREFIX kingdom: <http://example.org/kingdom/>

SELECT ?人物名 ?势力名
WHERE {
    ?人物  ex:字     ?人物名 ;
           ex:效力于  ?势力 .
    ?势力  ex:名称   ?势力名 .
}
```

这条查询的意思是:”找出所有人物的字和他们效力的势力名称”。

让我们拆解一下:

- `PREFIX` 定义了命名空间前缀（和 Turtle 中的 `@prefix` 作用相同）
- `SELECT ?人物名 ?势力名` 指定要返回的变量
- `WHERE { ... }` 描述了要匹配的图模式
- 以 `?` 开头的是变量，会在匹配过程中被绑定到具体的值

> **关键概念**
>
> SPARQL 的 WHERE 子句本质上就是一个图模式——你在画一个”模板图”，引擎帮你找到知识图谱中所有和这个模板匹配的部分。

2. CONSTRUCT——构造新的图

CONSTRUCT 查询不返回表格，而是返回一组新的 RDF 三元组。你可以用它来”转换”数据:

```
PREFIX ex: <http://example.org/ontology/>
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>

CONSTRUCT {
    ?人物  vcard:fn     ?名字 ;
           vcard:org    ?势力 .
}
WHERE {
    ?人物  ex:字       ?名字 ;
           ex:效力于    ?势力 .
}
```

这条查询把我们自定义的本体（ex:字、ex:效力于）映射到了标准的 vCard 词汇表（vcard:fn、vcard:org）。这是语义网中非常常见的操作——**数据集成和映射**。

3. ASK——判断是否存在

ASK 查询返回一个简单的 `true` 或 `false`：

```
PREFIX ex: <http://example.org/ontology/>
PREFIX person: <http://example.org/person/>
PREFIX kingdom: <http://example.org/kingdom/>

ASK {
    person:诸葛亮  ex:效力于  kingdom:蜀汉 .
}
```

这条查询问的是：”诸葛亮是否效力于蜀汉？”返回结果是 `true`。

ASK 查询特别适合做**条件检查**，比如在更新数据之前先检查某条数据是否已经存在。

4. DESCRIBE——获取资源描述

DESCRIBE 查询返回关于某个资源的所有相关三元组：

```
PREFIX person: <http://example.org/person/>

DESCRIBE person:诸葛亮
```

这条查询会返回所有以诸葛亮为主语或宾语的三元组，帮你快速了解一个资源的"全貌"。具体返回哪些三元组取决于 SPARQL 引擎的实现，通常会返回所有以该资源为主语的三元组。

SPARQL 进阶：过滤、排序与聚合

FILTER——添加过滤条件

```
PREFIX ex: <http://example.org/ontology/>
PREFIX person: <http://example.org/person/>

SELECT ?人物 ?年份
WHERE {
    ?人物  ex:出生年份  ?年份 .
    FILTER (?年份 < 170)
}
```

FILTER 支持丰富的函数，包括： – 数值比较：`<`、`>`、`=`、`!=` – 字符串函数：`regex(?name, "刘")`、`STRSTARTS(?name, "诸葛")` – 类型检查：`isURI(?x)`、`isLiteral(?x)` – 逻辑运

算：`&&`、`||`、`!`

ORDER BY、LIMIT、OFFSET——排序和分页

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?人物 ?武力值
WHERE {
    ?人物  ex:武力值  ?武力值 .
}
ORDER BY DESC(?武力值)
LIMIT 10
OFFSET 0
```

和 SQL 一样，`ORDER BY` 用于排序（`DESC` 为降序），`LIMIT` 和 `OFFSET` 用于分页。

聚合函数

SPARQL 1.1 支持常用的聚合函数：

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?势力 (COUNT(?人物) AS ?人数) (AVG(?武力值) AS ?平均武力
WHERE {
    ?人物  ex:效力于  ?势力 ;
           ex:武力值  ?武力值 .
}
GROUP BY ?势力
HAVING (COUNT(?人物) > 3)
```

支持的聚合函数包括：`COUNT`、`SUM`、`AVG`、`MIN`、`MAX`、`GROUP_CONCAT`、`SAMPLE`。

SPARQL 高级特性

OPTIONAL——可选匹配

有时候你想查询一些”有就显示，没有也不影响”的信息：

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?人物 ?字 ?号
WHERE {
    ?人物  ex:字  ?字 .
    OPTIONAL { ?人物  ex:号  ?号 . }
}
```

这样即使某个人物没有”号”，他仍然会出现在结果中，只是”号”那一列为空。

UNION——联合查询

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?人物 ?关系
WHERE {
    { ?人物  ex:结拜兄弟  ?兄弟 . BIND("结拜兄弟" AS ?关系) }
    UNION
    { ?人物  ex:师徒  ?师父 . BIND("师徒" AS ?关系) }
}
```

子查询

SPARQL 1.1 支持子查询，可以在查询中嵌套查询：

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?势力 ?人数
WHERE {
    {
        SELECT ?势力 (COUNT(?人物) AS ?人数)
        WHERE {
            ?人物  ex:效力于  ?势力 .
        }
        GROUP BY ?势力
    }
    FILTER (?人数 > 5)
}
```

SPARQL UPDATE——修改数据

SPARQL 1.1 还包括了更新操作（INSERT、DELETE），让我们不仅可以查询，还可以修改知识图谱中的数据：

```
PREFIX ex: <http://example.org/ontology/>
PREFIX person: <http://example.org/person/>
PREFIX kingdom: <http://example.org/kingdom/>

# 插入新数据
INSERT DATA {
    person:赵云  ex:字     "子龙" ;
                ex:效力于  kingdom:蜀汉 ;
                ex:武力值  97 .
}

# 删除旧数据
DELETE DATA {
    person:徐庶  ex:效力于  kingdom:蜀汉 .
}

# 先删后插（修改）
DELETE { ?人物 ex:武力值 ?旧值 }
INSERT { ?人物 ex:武力值 ?新值 }
WHERE {
    ?人物  ex:武力值  ?旧值 .
    BIND(?旧值 + 5 AS ?新值)
}
```

常用 RDF 词汇表

在语义网的生态中，有很多已经定义好的标准词汇表（vocabulary），你可以直接复用而不用自己造轮子。以下是最常用的几个：

FOAF（Friend of a Friend）——描述人际关系：

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

person:诸葛亮  foaf:name    "诸葛亮" ;
               foaf:mbox    <mailto:zhuge@shuhan.cn> ;
               foaf:knows   person:刘备, person:周瑜 ;
               foaf:age     54 .
```

Dublin Core（都柏林核心词汇）——描述数字资源的元数据：

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example.org/book/三国演义>
    dc:title    "三国演义" ;
    dc:creator  "罗贯中" ;
    dc:date     "1522" ;
    dc:subject  "历史小说" .
```

Schema.org——Google、Microsoft、Yahoo 联合推出的结构化数据词汇表，广泛用于搜索引擎优化：

```
{
  "@context": "https://schema.org",
  "@type": "Book",
  "name": "三国演义",
  "author": {
    "@type": "Person",
    "name": "罗贯中"
  },
  "datePublished": "1522"
}
```

SKOS（Simple Knowledge Organization System）——用于描述分类体系、词表、叙词表：

```
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .

ex:武将  skos:prefLabel  "武将"@zh, "Warrior"@en ;
         skos:broader    ex:人物 ;
         skos:scopeNote  "擅长武艺、征战沙场的人物"@zh .
```

RDF 在现实世界中的应用

你可能觉得 RDF 和语义网离我们的日常生活很远，但实际上它们在很多地方默默运行着：

1. Wikidata / DBpedia

维基百科的结构化数据版本。Wikidata 包含超过 1 亿条结构化数据，使用 RDF 格式存储，并提供公开的 SPARQL 端点。你可以在 https://query.wikidata.org 上尝试查询真实世界的知识图谱。

比如，用 SPARQL 查询所有获得诺贝尔物理学奖的中国裔科学家：

```
SELECT ?科学家 ?科学家名 WHERE {
  ?科学家 wdt:P166 wd:Q38104 .      # 获得诺贝尔物理学奖
  ?科学家 rdfs:label ?科学家名 .
  FILTER (lang(?科学家名) = "zh")
}
```

2. Google Knowledge Graph

当你在 Google 搜索"姚明身高"时，右侧出现的那个信息卡片就是 Knowledge Graph 在发挥作用。虽然 Google 内部使用的技术不完全是 RDF，但其结构化数据标记（Rich Snippets）大量使用了 Schema.org + JSON-LD 格式。

3. 生物医药领域

UniProt（蛋白质知识库）、Gene Ontology（基因本体）、DrugBank（药物数据库）等生物信息学数据库大量使用 RDF 和 OWL 来整合和推理跨库数据。这是语义网技术在工业界最成熟的应用领域之一。

链接数据（Linked Data）：让知识互联互通

语义网的终极目标不仅仅是让单个数据集用 RDF 来描述，而是要让不同的数据集之间**互相链接**，形成一个全球性的知识网络。这就是 **Linked Data（链接数据）** 的理念。

Tim Berners-Lee 在 2006 年提出了链接数据的四条原则：

1. **使用 URI 来标识事物**——给每个"东西"一个唯一的名字
2. **使用 HTTP URI**——让人们可以通过 HTTP 协议来查找这些信息
3. **提供有用的信息**——当有人访问一个 URI 时，返回关于这个事物的 RDF 描述
4. **包含指向其他 URI 的链接**——让你的数据和别人的数据关联起来

这四条原则听起来简单，但它们创造了一个奇迹：全球数千个数据集通过互相引用对方的 URI，形成了一个巨大的、互联的知识网络。这就是 **Linked Open Data Cloud**（链接开放数据云）。

举个实际的例子：假设你构建了一个"中国历史人物"的 RDF 数据集，其中刘备的 URI 是 `http://your-dataset.org/person/刘备`。你可以添加一条三元组，把它链接到 DBpedia 上的刘备：

```
<http://your-dataset.org/person/刘备>  owl:sameAs  <http:/
```

`owl:sameAs` 告诉推理引擎："这两个 URI 说的是同一个人"。这样，当有人查询你的数据集时，推理引擎可以自动把 DBpedia 上关于刘备的信息（出生日期、成就、家族关系等）和你的数据合并起来。这就是链接数据的力量——**知识不再是一座座孤岛，而是互联的网络**。

动手练习：用 Apache Jena 搭建本地 SPARQL 端点

什么是 Apache Jena?

Apache Jena 是一个开源的 Java 语义网框架，它提供了 RDF 数据的存储、推理和查询功能。我们要用到的是 Jena 中的 **Fuseki** 服务器——一个开箱即用的 SPARQL 端点。

第一步：安装 Apache Jena Fuseki

方法一：使用 Docker（推荐）

```
# 拉取 Fuseki 镜像
docker pull stain/jena-fuseki

# 启动 Fuseki 服务器
docker run -d \
  --name fuseki \
  -p 3030:3030 \
  -v fuseki-data:/fuseki \
  stain/jena-fuseki \
  --mem /ds
```

启动后，打开浏览器访问 `http://localhost:3030`，你会看到 Fuseki 的 Web 管理界面。

方法二：直接下载安装

```
# 下载 Apache Jena Fuseki
wget https://archive.apache.org/dist/jena/binaries/apache-

# 解压
tar -xzf apache-jena-fuseki-4.9.0.tar.gz
cd apache-jena-fuseki-4.9.0

# 启动服务器
./fuseki-server --mem /ds
```

第二步：准备三国人物数据

创建一个名为 `sanguo.ttl` 的 Turtle 文件：

```
@prefix ex: <http://example.org/ontology/> .
@prefix person: <http://example.org/person/> .
@prefix kingdom: <http://example.org/kingdom/> .
@prefix weapon: <http://example.org/weapon/> .
@prefix battle: <http://example.org/battle/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# === 势力 ===
kingdom:蜀汉  rdf:type  ex:势力 ;
              ex:名称    "蜀汉" ;
              ex:建立者  person:刘备 ;
              ex:都城    "成都" .

kingdom:曹魏  rdf:type  ex:势力 ;
              ex:名称    "曹魏" ;
              ex:建立者  person:曹操 ;
              ex:都城    "洛阳" .

kingdom:东吴  rdf:type  ex:势力 ;
              ex:名称    "东吴" ;
              ex:建立者  person:孙权 ;
              ex:都城    "建业" .

# === 人物 ===
person:刘备  rdf:type   ex:君主 ;
             ex:名称     "刘备" ;
             ex:字       "玄德" ;
             ex:出生年份  "161"^^xsd:integer ;
             ex:武力值   70 ;
             ex:智力值   75 ;
             ex:效力于   kingdom:蜀汉 .
```

```
person:关羽  rdf:type   ex:武将 ;
             ex:名称     "关羽" ;
             ex:字       "云长" ;
             ex:出生年份  "160"^^xsd:integer ;
             ex:武力值   97 ;
             ex:智力值   75 ;
             ex:效力于   kingdom:蜀汉 ;
             ex:使用武器  weapon:青龙偃月刀 ;
             ex:结拜兄弟  person:刘备, person:张飞 .

person:张飞  rdf:type   ex:武将 ;
             ex:名称     "张飞" ;
             ex:字       "翼德" ;
             ex:出生年份  "166"^^xsd:integer ;
             ex:武力值   96 ;
             ex:智力值   45 ;
             ex:效力于   kingdom:蜀汉 ;
             ex:使用武器  weapon:丈八蛇矛 ;
             ex:结拜兄弟  person:刘备, person:关羽 .

person:诸葛亮  rdf:type  ex:文官 ;
               ex:名称    "诸葛亮" ;
               ex:字      "孔明" ;
               ex:号      "卧龙" ;
               ex:出生年份 "181"^^xsd:integer ;
               ex:武力值  38 ;
               ex:智力值  100 ;
               ex:效力于  kingdom:蜀汉 .

person:曹操  rdf:type   ex:君主 ;
             ex:名称     "曹操" ;
             ex:字       "孟德" ;
             ex:出生年份  "155"^^xsd:integer ;
             ex:武力值   72 ;
             ex:智力值   96 ;
```

```
             ex:效力于    kingdom:曹魏 .

person:孙权  rdf:type    ex:君主 ;
             ex:名称      "孙权" ;
             ex:字        "仲谋" ;
             ex:出生年份  "182"^^xsd:integer ;
             ex:武力值    62 ;
             ex:智力值    80 ;
             ex:效力于    kingdom:东吴 .

person:周瑜  rdf:type    ex:武将 ;
             ex:名称      "周瑜" ;
             ex:字        "公瑾" ;
             ex:出生年份  "175"^^xsd:integer ;
             ex:武力值    78 ;
             ex:智力值    96 ;
             ex:效力于    kingdom:东吴 .

# === 武器 ===
weapon:青龙偃月刀  rdf:type  ex:武器 ;
                   ex:名称   "青龙偃月刀" ;
                   ex:重量   "82斤" .

weapon:丈八蛇矛  rdf:type  ex:武器 ;
                 ex:名称   "丈八蛇矛" .
```

第三步：上传数据到 Fuseki

在 Fuseki 的 Web 界面（`http://localhost:3030`）中：

1. 点击“add data”或“dataset”标签
2. 选择“upload files”
3. 上传你的 `sanguo.ttl` 文件

4. 点击“upload”

如果你更喜欢命令行，也可以使用 Jena 提供的工具：

```
# 使用 sparql 命令行工具直接查询本地文件（不需要 Fuseki）
sparql --data=sanguo.ttl --query=query.sparql

# 或者使用 curl 上传到 Fuseki
curl -X POST \
  -H "Content-Type: text/turtle" \
  --data-binary @sanguo.ttl \
  http://localhost:3030/ds/data
```

第四步：执行 SPARQL 查询

在 Fuseki 的 Web 界面中选择“query”标签，试试以下查询：

查询 1：列出所有人物的字和所属势力

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?名称 ?字 ?势力名
WHERE {
    ?人物  ex:名称   ?名称 ;
           ex:字     ?字 ;
           ex:效力于  ?势力 .
    ?势力  ex:名称   ?势力名 .
}
ORDER BY ?名称
```

查询 2：找出蜀汉所有武力值超过 90 的武将

```
PREFIX ex: <http://example.org/ontology/>
PREFIX kingdom: <http://example.org/kingdom/>

SELECT ?名称 ?武力值
WHERE {
    ?人物  ex:名称    ?名称 ;
           ex:武力值   ?武力值 ;
           ex:效力于   kingdom:蜀汉 .
    FILTER (?武力值 > 90)
}
ORDER BY DESC(?武力值)
```

查询 3：统计各势力的人数和平均武力值

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?势力名 (COUNT(?人物) AS ?人数) (AVG(?武力值) AS ?平均武
WHERE {
    ?人物  ex:效力于  ?势力 ;
           ex:武力值  ?武力值 .
    ?势力  ex:名称   ?势力名 .
}
GROUP BY ?势力名
ORDER BY DESC(?人数)
```

查询 4：找出结拜兄弟关系

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?人物名 ?兄弟名
WHERE {
    ?人物  ex:名称      ?人物名 ;
           ex:结拜兄弟   ?兄弟 .
    ?兄弟  ex:名称      ?兄弟名 .
}
```

查询 5：找出谁的智力值最高

```
PREFIX ex: <http://example.org/ontology/>

SELECT ?名称 ?智力值
WHERE {
    ?人物  ex:名称   ?名称 ;
           ex:智力值  ?智力值 .
}
ORDER BY DESC(?智力值)
LIMIT 3
```

第五步：使用 Python 连接 SPARQL 端点

```
from SPARQLWrapper import SPARQLWrapper, JSON

# 连接到 Fuseki SPARQL 端点
sparql = SPARQLWrapper("http://localhost:3030/ds/sparql")

# 编写查询
query = """
PREFIX ex: <http://example.org/ontology/>

SELECT ?名称 ?字 ?势力名
WHERE {
    ?人物  ex:名称   ?名称 ;
           ex:字     ?字 ;
           ex:效力于  ?势力 .
    ?势力  ex:名称   ?势力名 .
}
ORDER BY ?名称
"""

sparql.setQuery(query)
sparql.setReturnFormat(JSON)

# 执行查询
results = sparql.query().convert()

# 打印结果
for result in results["results"]["bindings"]:
    name = result["名称"]["value"]
    zi = result["字"]["value"]
    kingdom = result["势力名"]["value"]
    print(f"{name}（字{zi}）效力于{kingdom}")
```

运行上述代码，你会看到：

```
刘备（字玄德）效力于蜀汉
关羽（字云长）效力于蜀汉
张飞（字翼德）效力于蜀汉
诸葛亮（字孔明）效力于蜀汉
曹操（字孟德）效力于曹魏
孙权（字仲谋）效力于东吴
周瑜（字公瑾）效力于东吴
```

使用 rdflib 本地处理 RDF

如果你不想搭建 Fuseki 服务器，也可以用 Python 的 **rdflib** 库在本地直接加载和查询 RDF 数据：

```
from rdflib import Graph, Namespace, Literal, URIRef
from rdflib.namespace import RDF, RDFS

# 创建一个空的图
g = Graph()

# 加载 Turtle 文件
g.parse("sanguo.ttl", format="turtle")

# 查看图中有多少条三元组
print(f"图中有 {len(g)} 条三元组")

# 定义命名空间
EX = Namespace("http://example.org/ontology/")
PERSON = Namespace("http://example.org/person/")
KINGDOM = Namespace("http://example.org/kingdom/")

# 使用 SPARQL 查询
query = """
PREFIX ex: <http://example.org/ontology/>

SELECT ?名称 ?字 ?武力值
WHERE {
    ?人物  ex:名称   ?名称 ;
           ex:字     ?字 ;
           ex:武力值  ?武力值 .
    FILTER (?武力值 > 90)
}
ORDER BY DESC(?武力值)
"""

results = g.query(query)
print("\n武力值超过90的人物：")
for row in results:
```

```
    print(f"  {row.名称} (字{row.字}) 武力值: {row.武力值}")

# 使用 Python API 遍历图
print("\n所有人物的字: ")
for s, p, o in g.triples((None, EX.字, None)):
    # 找到这个人的名称
    name = list(g.objects(s, EX.名称))[0]
    print(f"  {name}, 字{o}")
```

rdflib 还支持**推理**功能（通过 owlrl 库）：

```
from rdflib import Graph
import owlrl

g = Graph()
g.parse("sanguo.ttl", format="turtle")
g.parse("sanguo_ontology.ttl", format="turtle")

# 执行 RDFS 推理
owlrl.DeductiveClosure(owlrl.RDFS_Semantics).expand(g)

print(f"推理后共有 {len(g)} 条三元组")
# 推理后的三元组数量会比原始数据多，因为推理引擎自动补全了子类、子属性
```

安装所需的 Python 库：

```
pip install SPARQLWrapper rdflib owlrl
```

`SPARQLWrapper` 用于连接远程 SPARQL 端点，`rdflib` 用于本地 RDF 处理，`owlrl` 用于推理。这三个库是 Python 语义网开发的”三件套”。

动手练习

练习 1：扩展三国数据

在上面的数据基础上，添加以下内容：

1. 为以下人物编写 RDF 三元组（至少 10 条）：
2. 赵云（字子龙，蜀汉武将，武力值 97）
3. 吕布（字奉先，先后效力多个势力，武力值 100）
4. 司马懿（字仲达，曹魏文官，智力值 98）
5. 陆逊（字伯言，东吴武将，智力值 95）
6. 为每个人物添加”所属战役”关系（比如：赵云参加了长坂坡之战）
7. 编写 SPARQL 查询来回答以下问题：
8. 谁是三国中武力值最高的人？
9. 参加过”赤壁之战”的人物有哪些？
10. 哪个势力的武将平均智力最高？

练习 2：构建你的个人知识图谱

用 Turtle 格式描述你自己的知识图谱，至少包含：

- 5 个你认识的人（家人、朋友、同事）
- 每个人至少 3 个属性（姓名、关系、所在城市等）
- 人和人之间的关系（认识、同事、亲属等）

- 至少使用 OPTIONAL 和 FILTER 各一次来查询

练习 3：CONSTRUCT 数据转换

使用 CONSTRUCT 查询，把三国人物的数据从自定义本体映射到 FOAF（Friend of a Friend）词汇表：

- `ex:名称` → `foaf:name`
- `ex:效力于` → `foaf:member`
- `ex:结拜兄弟` → `foaf:knows`

提示：FOAF 的命名空间是 `http://xmlns.com/foaf/0.1/`

延伸阅读

1. **W3C RDF 规范**：https://www.w3.org/TR/rdf11-primer/ RDF 1.1 的官方入门指南，是最权威的参考资料。
2. **W3C SPARQL 规范**：https://www.w3.org/TR/sparql11-query/ SPARQL 1.1 查询语言的完整规范。虽然内容很长，但对于想深入理解 SPARQL 的读者来说是必读之作。
3. **OWL 2 Web Ontology Language**：https://www.w3.org/TR/owl2-primer/ OWL 2 的官方入门指南，详细解释了 OWL 的各种构件和推理能力。

4. **Apache Jena 官方文档**：https://jena.apache.org/documentation/ Jena 的完整文档，包括 Fuseki 服务器配置、ARQ 查询引擎、推理引擎等。

5. **Protégé 本体编辑器**：https://protege.stanford.edu/ Stanford 大学开发的本体编辑工具，提供了可视化的方式来创建和编辑 OWL 本体。非常适合想要"看到"本体的初学者。

6. **DBpedia**：https://www.dbpedia.org/ 从维基百科自动提取的大规模知识图谱，是学习 RDF 和 SPARQL 的绝佳实践资源。它提供了公开的 SPARQL 端点供你练习查询。

7. **Linked Open Data Cloud**：https://lod-cloud.net/ 链接开放数据云的全景图，展示了互联网上所有互联的 RDF 数据集。这张图本身就是一个壮丽的知识图谱。

本章小结

本章我们学习了语义网的三大核心技术：

1. RDF（资源描述框架） 是语义网的数据基础。它用"主语-谓语-宾语"三元组来描述知识，简单而强大。我们学习了四种主要的序列化格式：Turtle（最推荐的人类可读格式）、N-Triples（最适合机器处理）、JSON-LD（Web 开发者的首选）和 RDF/XML（元老级格式）。

2. OWL（Web 本体语言） 在 RDF 之上添加了丰富的表达能力，让我们可以定义类、属性和约束，并通过推理引擎自动推导隐含的知识。它是构建严谨本体的核心工具。

3. SPARQL（SPARQL 查询语言） 是 RDF 数据的查询语言，提供了 SELECT、CONSTRUCT、ASK、DESCRIBE 四种查询形式，以及过滤、排序、聚合、子查询等高级特性。

我们还动手实践了：使用 Apache Jena Fuseki 搭建了一个本地 SPARQL 端点，上传了三国人物的 RDF 数据，并执行了多种 SPARQL 查询。

核心要点回顾：

表 8-6

技术	角色	类比	核心概念
RDF	数据模型	砖块	三元组（主语–谓语–宾语）
OWL	本体语言	建筑图纸	类、属性、约束、推理
SPARQL	查询语言	电梯	图模式匹配

在下一章中，我们将切换到另一种主流的知识图谱模型——**属性图（Property Graph）**，并学习它配套的查询语言 **Cypher**。如果说 RDF 是学术界的最爱，那么属性图就是工业界的宠儿。Neo4j、JanusGraph、TigerGraph 等流行的图数据库都采用了属性图模型。让我们继续前进！

第5章 属性图模型与Cypher语言

引子

在上一章中，我们学习了 RDF——一种用”主语–谓语–宾语”三元组来描述世界的标准。RDF 很优雅，但在实际工程中，很多开发者会遇到这样的困惑：

> *“我只是想在每个节点上加一个’创建时间’属性，为什么在 RDF 里这么麻烦？”*
>
> *“我想在’朋友’这条关系上加一个’认识年份’属性，RDF 怎么做？要引入’复合节点’？太绕了吧。”*
>
> *“为什么每条信息都要用一个 URI？我只是想存一个普通的数字而已。”*

如果你也有过类似的困惑，那么今天的主角——**属性图（Property Graph）**——正是为你而来的。

属性图是当今最流行的图数据模型，几乎所有主流图数据库都采用了它：**Neo4j**（全球使用最广泛的图数据库）、**JanusGraph**（Linux 基金会旗下的开源分布式图数据库）、**Amazon Neptune**、**TigerGraph** 等等。GQL（Graph Query Language），作为即将取代 SQL 成为 ISO 标准的图查询语言，也是基于属性图模型的。

如果说 RDF 是学术界的”白月光”，那属性图就是工业界的”实干家”。它不追求理论上的完美，而是追求**工程上的好用**。今天，我们就来认识这位实干家，并学习它的最佳搭档——**Cypher 查询语言**。

正文

从 RDF 到属性图：换一种方式看图

RDF 的”不够用”

让我们先回忆一下上一章的 RDF 模型。在 RDF 中，一切都是三元组：

```
刘备 → 字 → "玄德"
刘备 → 效力于 → 蜀汉
```

这个模型很简洁，但有几个实际问题：

问题 1：节点属性不自然

在 RDF 中，如果你想给”刘备”这个节点加上年龄、身高、创建时间等属性，每个属性都是一条独立的三元组。这在概念上没问题，但在工程实现上，这些”属性三元组”和”关系三元组”混在一起，不好区分。

问题 2：关系上不能直接加属性

假设你想描述”关羽在公元 200 年加入蜀汉”。在 RDF 中，你不能直接在”效力于”这条关系上加”时间”属性。你需要引入一种叫做”具体化（Reification）”的复杂机制，把一个简单的关系变成一个复合节点。

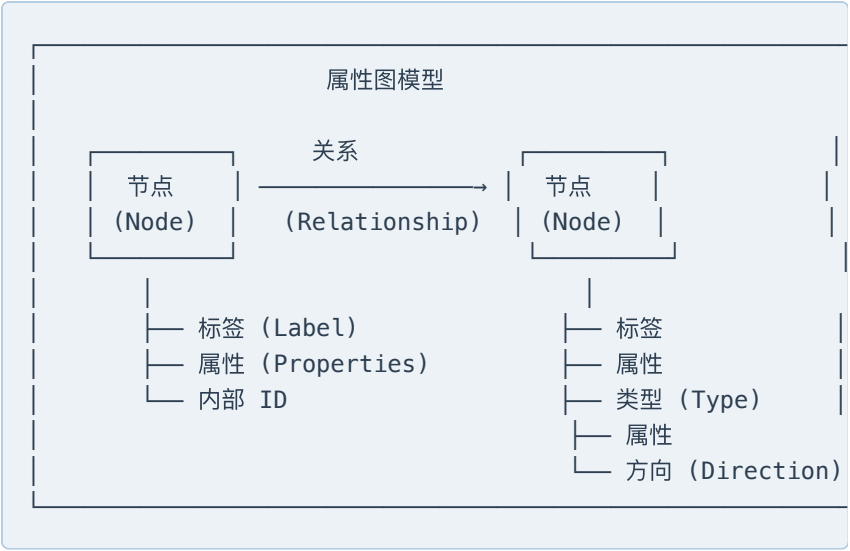
问题 3：没有原生的 ID 机制

RDF 用 URI 来标识一切。URI 很好，但有时候你只想用一个简单的数字 ID（比如自增主键），而不是一个长长的 URL。

属性图就是为了解决这些痛点而生的。

属性图的核心概念

属性图模型由四个核心元素组成：



让我们逐一解释：

1. 节点（Node）

节点是图中的实体。每个节点有：

- **内部 ID**：数据库自动分配的唯一标识（如 `0`、`1`、`2`...）
- **标签（Label）**：一个或多个标签，用来分类（如 `:人物`、`:武将`）
- **属性（Properties）**：键值对，存储节点的详细信息

用 Cypher 语法来表示一个节点：

```
(刘备:人物:君主 {
  姓名: "刘备",
  字: "玄德",
  出生年份: 161,
  武力值: 70,
  智力值: 75
})
```

看到了吗？节点可以直接拥有属性，不需要像 RDF 那样拆成三元组。标签可以有多个——刘备既是一个”人物”，也是一个”君主”。

2. 关系（Relationship）

关系连接两个节点，具有**方向**和**类型**：

```
(刘备)-[:结拜兄弟 {结拜地点: "桃园", 结拜年份: 184}]->(关羽)
```

关系也可以有属性！这就是属性图相比 RDF 的一大优势。”结拜兄弟”这条关系本身就携带了”在哪里结拜”“什么时候结拜”的信息，不需要用复合节点来实现。

3. 标签（Label）

标签用来对节点进行分类。一个节点可以有零个或多个标签：

```
// 单标签
(诸葛亮:人物)

// 多标签
(诸葛亮:人物:文官:蜀汉)

// 通过标签可以快速筛选
MATCH (p:武将) RETURN p    // 只找武将
MATCH (p:人物:蜀汉) RETURN p  // 找蜀汉的人物
```

标签就像是给节点贴的标签贴纸，一个节点可以贴很多张。

4. 属性（Properties）

属性是键值对，可以附加在节点和关系上：

```
支持的属性类型：
- 整数：70, 161
- 浮点数：3.14
- 字符串："刘备"
- 布尔值：true, false
- 列表：["桃园结义", "三顾茅庐"]
- 日期：date("2024-01-01")
- 时间：datetime("2024-01-01T12:00:00")
- 点（空间坐标）：point({x: 1, y: 2})
```

一个完整的属性图示例

让我们用三国人物来构建一个完整的属性图：

```
节点：
├── (刘备:人物:君主) - {字:"玄德", 武力值:70, 智力值:75}
├── (关羽:人物:武将) - {字:"云长", 武力值:97, 智力值:75}
├── (张飞:人物:武将) - {字:"翼德", 武力值:96, 智力值:45}
├── (诸葛亮:人物:文官) - {字:"孔明", 武力值:38, 智力值:100}
├── (曹操:人物:君主) - {字:"孟德", 武力值:72, 智力值:96}
├── (孙权:人物:君主) - {字:"仲谋", 武力值:62, 智力值:80}
├── (周瑜:人物:武将) - {字:"公瑾", 武力值:78, 智力值:96}
├── (蜀汉:势力) - {都城:"成都", 建立年份:221}
├── (曹魏:势力) - {都城:"洛阳", 建立年份:220}
└── (东吴:势力) - {都城:"建业", 建立年份:229}

关系：
├── 刘备 -[:结拜兄弟 {地点:"桃园"}]-> 关羽
├── 刘备 -[:结拜兄弟 {地点:"桃园"}]-> 张飞
├── 关羽 -[:结拜兄弟 {地点:"桃园"}]-> 张飞
├── 刘备 -[:效力]-> 蜀汉
├── 关羽 -[:效力]-> 蜀汉
├── 张飞 -[:效力]-> 蜀汉
├── 诸葛亮 -[:效力]-> 蜀汉
├── 刘备 -[:三顾茅庐]-> 诸葛亮
├── 曹操 -[:效力]-> 曹魏
├── 孙权 -[:效力]-> 东吴
├── 周瑜 -[:效力]-> 东吴
└── 周瑜 -[:对手]-> 诸葛亮
```

这个图的结构非常直观：节点是人或势力，关系描述了人与人、人与势力之间的联系，每个节点和关系上都携带丰富的属性信息。

属性图的来龙去脉

属性图模型并不是某个公司凭空发明的，它来源于图论（Graph Theory）中”属性图”的数学定义——一个由节点和边组成的图，其中节点和边都可以携带键值对属性。

2000 年，瑞典程序员 Emil Eifrem 在从斯德哥尔摩飞往布鲁塞尔的航班上，随手画了一张”谁认识谁”的社交图。他忽然意识到：这种”节点–关系–属性”的模型，恰好可以解决关系数据库中多表 JOIN 性能低下的问题。两年后，他和两位好友一起创立了 Neo4j 项目，这是世界上第一个属性图数据库。

二十多年过去了，Neo4j 已经从一个航班上的灵感变成了全球使用最广泛的图数据库，拥有超过百万的下载量和活跃的开发者社区。属性图模型也被越来越多的数据库厂商采纳，成为了事实上的工业标准。

2019 年，ISO/IEC 正式立项了 **GQL（Graph Query Language）** 国际标准，它将以 openCypher 为基础，融合 PGQL（Oracle）和 SQL/PGQ 的特性，成为继 SQL 之后的第二个 ISO 标准数据库查询语言。这意味着，学会 Cypher，你不仅是在学一种工具，更是在掌握未来图数据库的通用语言。

图的可视化：看见你的知识

属性图有一个 RDF 不太具备的优势：**天然的可视化友好性**。当你把节点画成圆圈、关系画成连线、属性显示在圆圈里面时，一个直观的知识网络就呈现出来了。

Neo4j Browser 内置了强大的图可视化功能。当你执行一条 `MATCH ... RETURN` 查询时，结果不仅可以以表格形式展示，还可以以**图形（Graph）** 模式展示——每个节点变成一个可拖拽的圆点，关系变成连线，你可以通过点击、拖拽、缩放来探索图的拓扑结构。

这种”所见即所得”的交互方式，对于理解复杂关系特别有帮助。比如，当你查询”刘备的 2 度社交圈”时，图形模式会呈现出一个以刘备为中心的放射状网络，你能一眼看出谁是核心人物、哪些是紧密的小圈子、哪些是边缘的弱连接。

在工程实践中，很多团队会用 Neo4j Browser 做快速的数据探索，用 Neo4j Bloom 做面向业务用户的可视化分析，用 Gephi 或 D3.js 做定制化的图可视化应用。可视化的力量在于，它能让那些隐藏在数据深处的模式和异常”浮出水面”，这恰恰是图数据库最擅长的事情。

属性图 vs RDF：该选哪个？

这是很多初学者都会纠结的问题。让我用一张表格来帮你做选择：

表 9–1

维度	RDF	属性图
标准化程度	W3C 国际标准，高度规范	事实标准（Neo4j 推动），正在标准化（GQL）
数据模型	三元组（主语–谓语–宾语）	节点–关系–属性
节点属性	通过三元组间接表达	原生支持键值对
关系属性	需要具体化（复杂）	原生支持
标识符	URI（全局唯一）	内部 ID + 自定义属性
推理能力	强大（OWL + 推理引擎）	较弱（需要手动实现）
查询语言	SPARQL	Cypher / Gremlin / GQL
工具生态	Apache Jena, Protégé, Virtuoso	Neo4j, JanusGraph, Neptune
学习曲线	较陡（URI、命名空间、本体论）	较缓（直观、接近编程思维）
适用场景	学术、开放数据互联、本体推理	企业应用、社交网络、推荐系统、欺诈检测

选择建议：

- 如果你的项目需要**严格的语义推理**、**跨数据集互联**或**遵循 W3C 标准**，选 RDF
- 如果你的项目侧重**工程实现**、**快速开发**、**丰富的节点和关系属性**，选属性图
- 如果你需要两者兼得，一些现代图数据库（如 Stardog、Anzo）同时支持 RDF 和属性图

> **树懒老K的经验**
>
> 在实际项目中，80% 的场景用属性图就够了。除非你在做学术语义网研究、开放数据链接、或需要复杂的本体推理，否则属性图 + Cypher 的组合是最高效的选择。

属性图在工业界的实战应用

属性图之所以在工业界如此受欢迎，是因为它天然契合很多现实世界的业务场景。让我们看几个典型的应用领域：

1. 社交网络分析

这是属性图最经典的应用。微信的好友关系、微博的关注关系、LinkedIn 的职业人脉，都可以用属性图来建模。节点是人，关系是”好友”“关注”“同事”，属性包括”认识时间”“亲密度”“互动频率”等。

典型查询包括： – “你可能认识的人”（2 度人脉推荐） – “你和一个陌生人之间有几度分隔”（六度分隔理论验证） – “找出你朋友圈中的核心人物”（PageRank 中心性分析）

2. 反欺诈与风控

银行和支付机构用属性图来检测欺诈行为。把用户、银行卡、手机号、IP 地址、设备 ID 都建模为节点，把”使用”“持有”“登录”等建模为关系。当多个看似不相关的账户共享同一个设备 ID 或 IP 地址时，图数据库可以迅速发现隐藏的欺诈网络。

这种”团伙欺诈”检测是传统关系数据库的短板——因为你需要做多跳连接查询，而关系数据库在 3 跳以上的 JOIN 性能会急剧下降。

3. 知识图谱与企业搜索

Google 的 Knowledge Graph、百度的知心图谱、微软的 Satori 都在使用图模型来增强搜索体验。当你搜索”姚明的妻子”时，搜索引擎不是简单地做关键词匹配，而是在知识图谱中查找”姚明”节点的”配偶”关系，然后返回对应的节点信息。

企业内部的知识管理也越来越依赖图数据库：员工技能图谱、项目关系网络、文档关联图谱等，都可以帮助员工更高效地找到所需的信息和专家。

4. 推荐系统

电商平台的”猜你喜欢”、视频平台的”为你推荐”、音乐平台的”相似歌曲”，背后都可以用属性图来实现。用户、商品、标签、品类都是节点，”购买过”“浏览过”“收藏了”“属于品类”都是关系。基于图的推荐算法（如 PersonalRank、SimRank）可以利用图的拓扑结构来发现隐藏的偏好关联。

5. 供应链与物流

制造业的供应链管理天然就是一个图结构：原材料供应商→零部件工厂→组装厂→分销商→零售商→消费者。用属性图建模后，你可以快速回答”如果某个供应商断供，会影响哪些最终产品？”这类关键问题。

Cypher：像画画一样查询

什么是 Cypher?

Cypher 是 Neo4j 开发的一种声明式图查询语言。它的设计灵感来源于 ASCII 艺术——用可视化的模式来描述图结构，让你一眼就能看懂查询在做什么。

```
MATCH (person:人物)-[:效力]->(kingdom:势力)
WHERE kingdom.名称 = "蜀汉"
RETURN person.姓名, person.字
```

看到那个 `()-[]->()` 了吗？它就是在用文本”画”一个图的模式。这种直观的语法是 Cypher 最大的魅力所在。

> *2015 年，Neo4j 将 Cypher 开源为 openCypher 项目。2018 年，Cypher 被提交为 ISO GQL（Graph Query Language）标准的基础，GQL 将成为继 SQL 之后的第二个 ISO 标准数据库查询语言。*

Cypher 基础语法

MATCH——匹配图模式

`MATCH` 是 Cypher 最常用的关键词，用来描述你想查找的图模式：

```
// 查找所有"人物"节点
MATCH (p:人物)
RETURN p

// 查找"效力"关系
MATCH (p:人物)-[:效力]->(k:势力)
RETURN p.姓名, k.名称

// 查找特定的人
MATCH (p:人物 {姓名: "刘备"})
RETURN p

// 获取关系的属性
MATCH (p:人物)-[r:结拜兄弟]->(friend:人物)
WHERE p.姓名 = "刘备"
RETURN friend.姓名, r.地点
```

模式语法速查：

```
(n)                          – 匹配任意节点
(n:人物)                     – 匹配标签为"人物"的节点
(n:人物:武将)                – 匹配同时有两个标签的节点
(n {姓名: "刘备"})           – 匹配属性等于指定值的节点
(n:人物 {姓名: "刘备"}) – 标签 + 属性

(n)-[r]->(m)                 – 有向关系
(n)-[r:效力]->(m)            – 指定关系类型
(n)-[r:效力|结拜兄弟]->(m)   – 多种关系类型（OR）
(n)-[r]->(m)<-[s]-(n)   – 复杂模式

// 无方向匹配（匹配两个方向）
(n)-[r:效力]-(m)
```

CREATE——创建节点和关系

```
// 创建一个节点
CREATE (p:人物 {姓名: "赵云", 字: "子龙", 武力值: 97, 智力值: 7

// 创建节点和关系（一步到位）
CREATE (p:人物 {姓名: "马超", 字: "孟起", 武力值: 95, 智力值: 4
       -[:效力]->(:势力 {名称: "蜀汉", 都城: "成都"})

// 在已有节点之间创建关系
MATCH (p:人物 {姓名: "赵云"}), (k:势力 {名称: "蜀汉"})
CREATE (p)-[:效力]->(k)

// 创建双向关系
MATCH (a:人物 {姓名: "刘备"}), (b:人物 {姓名: "诸葛亮"})
CREATE (a)-[:三顾茅庐]->(b),
       (b)-[:辅佐]->(a)
```

MERGE——智能创建（有则匹配，无则创建）

`MERGE` 是 Cypher 中非常优雅的一个操作。它像是 MATCH 和 CREATE 的结合体——如果模式存在就匹配它，不存在就创建它：

```
// 如果不存在这个人物，就创建
MERGE (p:人物 {姓名: "黄忠"})
// 如果是新创建的，设置这些属性
ON CREATE SET p.字 = "汉升", p.武力值 = 93, p.智力值 = 60
// 如果是匹配到已有的，更新最后访问时间
ON MATCH SET p.最后更新 = datetime()

// MERGE 关系
MATCH (p:人物 {姓名: "黄忠"}), (k:势力 {名称: "蜀汉"})
MERGE (p)-[r:效力]->(k)
ON CREATE SET r.加入年份 = 214
```

`MERGE` 特别适合数据导入和增量更新的场景——你不用担心重复创建。

WHERE——过滤条件

`WHERE` 用来添加过滤条件，和 SQL 中的 WHERE 非常相似：

```
// 基本过滤
MATCH (p:人物)
WHERE p.武力值 > 90
RETURN p.姓名, p.武力值

// 多条件组合
MATCH (p:人物)
WHERE p.武力值 > 90 AND p.智力值 > 70
RETURN p.姓名, p.武力值, p.智力值

// 字符串匹配
MATCH (p:人物)
WHERE p.字 STARTS WITH "玄"
   OR p.姓名 CONTAINS "诸"
   OR p.字 ENDS WITH "德"
RETURN p.姓名, p.字

// 正则表达式
MATCH (p:人物)
WHERE p.姓名 =~ ".*[刘关张].*"
RETURN p.姓名

// 判断属性是否存在
MATCH (p:人物)
WHERE p.号 IS NOT NULL
RETURN p.姓名, p.号

// 范围查询
MATCH (p:人物)
WHERE p.出生年份 >= 160 AND p.出生年份 <= 180
RETURN p.姓名, p.出生年份

// IN 操作
```

```
MATCH (p:人物)
WHERE p.姓名 IN ["刘备", "关羽", "张飞"]
RETURN p.姓名, p.字
```

RETURN——返回结果

```
// 返回节点
MATCH (p:人物) RETURN p

// 返回特定属性
MATCH (p:人物) RETURN p.姓名, p.字

// 使用别名
MATCH (p:人物) RETURN p.姓名 AS 名字, p.武力值 AS 武力

// 返回表达式
MATCH (p:人物)
RETURN p.姓名 AS 名字,
       p.武力值 + p.智力值 AS 综合能力

// 去重
MATCH (p:人物)-[:效力]->(k:势力)
RETURN DISTINCT k.名称 AS 势力名

// 排序和分页
MATCH (p:人物)
RETURN p.姓名, p.武力值
ORDER BY p.武力值 DESC
SKIP 2
LIMIT 5
```

聚合函数

Cypher 支持丰富的聚合函数：

```
// COUNT
MATCH (p:人物)
RETURN COUNT(p) AS 总人数

// 分组聚合
MATCH (p:人物)-[:效力]->(k:势力)
RETURN k.名称 AS 势力,
       COUNT(p) AS 人数,
       AVG(p.武力值) AS 平均武力,
       MAX(p.武力值) AS 最高武力,
       MIN(p.智力值) AS 最低智力,
       COLLECT(p.姓名) AS 成员列表

// 去重计数
MATCH (p:人物)-[:效力]->(k:势力)
RETURN k.名称, COUNT(DISTINCT p) AS 人数
```

Cypher vs SQL：同与不同

很多读者是从关系数据库过来的，这里做一个 Cypher 和 SQL 的对比，帮你快速建立认知映射：

查询所有蜀汉的人物：

```
-- SQL 版本
SELECT p.姓名, p.字
FROM 人物 p
JOIN 效力 e ON p.id = e.人物_id
JOIN 势力 k ON e.势力_id = k.id
WHERE k.名称 = '蜀汉'
```

```
-- Cypher 版本
MATCH (p:人物)-[:效力]->(k:势力 {名称: "蜀汉"})
RETURN p.姓名, p.字
```

看到了吗？SQL 需要用两个 JOIN 把三张表连起来，而 Cypher 用一个直观的模式 `()-[]->()` 就描述了同样的关系。这正是图查询语言的优势——**你的查询长得就像你脑海中的那个图**。

查找朋友的朋友（2 度人脉）：

```
-- SQL 版本（需要自连接）
SELECT f2.朋友姓名
FROM 朋友关系 fr1
JOIN 朋友关系 fr2 ON fr1.朋友_id = fr2.人物_id
JOIN 人物 f2 ON fr2.朋友_id = f2.id
WHERE fr1.人物_id = (SELECT id FROM 人物 WHERE 姓名 = '刘备'
  AND f2.id != (SELECT id FROM 人物 WHERE 姓名 = '刘备')
```

```
-- Cypher 版本
MATCH (a:人物 {姓名: "刘备"})-[:朋友*2]-(b:人物)
WHERE a <> b
RETURN DISTINCT b.姓名
```

在 2 度人脉的场景下差异还不算太大，但如果你要查 6 度人脉呢？SQL 需要 6 次 JOIN，性能和可读性都会急剧下降。而 Cypher 只需要把 `*2` 改成 `*6`，简洁而优雅。这就是图数据库在**多跳关系查询**上的碾压级优势。

Cypher 常用函数

Cypher 提供了丰富的内置函数，以下是按类别整理的常用函数：

字符串函数：

```
// 字符串操作
RETURN toUpper("hello")          // "HELLO"
RETURN toLower("HELLO")          // "hello"
RETURN trim("  hello  ")         // "hello"
RETURN substring("三国演义", 0, 2)  // "三国"
RETURN replace("刘备字玄德", "字", "的字是")  // "刘备的字是玄德
RETURN size("诸葛亮")               // 3（字符数）
RETURN split("刘备,关羽,张飞", ",")  // ["刘备", "关羽", "张飞
```

数学函数：

```
RETURN abs(-5)        // 5
RETURN ceil(3.2)      // 4
RETURN floor(3.8)     // 3
RETURN round(3.567, 1)  // 3.6（保留1位小数）
RETURN sqrt(16)       // 4.0
RETURN sign(-5)       // -1
RETURN rand()         // 0 到 1 之间的随机数
```

列表函数：

```
RETURN range(1, 5)           // [1, 2, 3, 4, 5]
RETURN size(["刘备","关羽"])    // 2
RETURN head(["刘备","关羽","张飞"])  // "刘备"
RETURN last(["刘备","关羽","张飞"])   // "张飞"
RETURN tail(["刘备","关羽","张飞"])   // ["关羽", "张飞"]
RETURN reverse([1,2,3])      // [3, 2, 1]
```

日期和时间函数：

```
RETURN date()                      // 当前日期
RETURN date("2024-01-15")          // 指定日期
RETURN datetime()                  // 当前日期时间
RETURN duration("P1Y2M3D")         // 1年2个月3天的时间间隔

// 日期属性
WITH date("2024-03-15") AS d
RETURN d.year, d.month, d.day, d.dayOfWeek
```

类型转换函数：

```
RETURN toInteger("42")         // 42
RETURN toFloat("3.14")         // 3.14
RETURN toString(100)           // "100"
RETURN coalesce(null, "默认值")  // "默认值"（取第一个非null值
```

CASE 表达式

和 SQL 类似，Cypher 也支持 CASE 表达式来做条件判断：

```
MATCH (p:人物)
RETURN p.姓名,
       p.武力值,
       CASE
         WHEN p.武力值 >= 95 THEN "超一流"
         WHEN p.武力值 >= 85 THEN "一流"
         WHEN p.武力值 >= 70 THEN "二流"
         ELSE "三流"
       END AS 武力等级
ORDER BY p.武力值 DESC
```

索引与约束：让查询更快

在生产环境中，数据量往往很大，没有索引的查询会变得很慢。Neo4j 支持多种索引和约束来优化查询性能：

创建索引：

```
// 为"人物"标签的"姓名"属性创建索引
CREATE INDEX 人物姓名索引 FOR (p:人物) ON (p.姓名)

// 为"武将"标签的"武力值"属性创建索引
CREATE INDEX 武将武力索引 FOR (p:武将) ON (p.武力值)

// 复合索引（多属性）
CREATE INDEX 人物综合索引 FOR (p:人物) ON (p.姓名, p.字)

// 全文索引（支持模糊搜索）
CREATE FULLTEXT INDEX 人物全文索引 FOR (p:人物) ON EACH [p.姓

// 查看所有索引
SHOW INDEXES
```

创建约束（Constraint）：

```
// 唯一性约束：每个人物的姓名必须唯一
CREATE CONSTRAINT 人物姓名唯一 FOR (p:人物) REQUIRE p.姓名 IS

// 存在性约束：人物必须有姓名属性（Enterprise Edition）
CREATE CONSTRAINT 人物必须有姓名 FOR (p:人物) REQUIRE p.姓名 I

// 节点键约束：标签 + 属性组合必须唯一（Enterprise Edition）
CREATE CONSTRAINT 人物键 FOR (p:人物) REQUIRE (p.姓名, p.字)

// 查看所有约束
SHOW CONSTRAINTS
```

> **小贴士**
>
> 创建唯一性约束时，Neo4j 会自动为该属性创建一个索引，所以你不需要手动创建。另外，在导入大量数据之前，建议先创建好约束和索引，这样可以确保数据质量和查询性能。

索引对查询性能的影响：

没有索引时，`MATCH (p:人物 {姓名: "刘备"})` 需要扫描所有"人物"标签的节点。有了索引后，Neo4j 可以直接定位到目标节点，查询速度可以提升数个数量级。在数据量超过百万级时，索引的重要性会更加明显。

你可以通过 `EXPLAIN` 和 `PROFILE` 来查看查询执行计划：

```
// 查看执行计划（不实际执行）
EXPLAIN MATCH (p:人物 {姓名: "刘备"}) RETURN p

// 查看执行计划（实际执行并显示每步耗时）
PROFILE MATCH (p:人物 {姓名: "刘备"}) RETURN p
```

Cypher 模式匹配：深入图的奥秘

路径（Path）

路径是 Cypher 中最强大的模式匹配能力之一。通过路径，你可以查找跨越多个节点和关系的复杂连接：

```
// 查找两个节点之间的路径
MATCH path = (a:人物 {姓名: "刘备"})-[*]-(b:人物 {姓名: "曹操"
RETURN path

// 获取路径长度
MATCH path = (a:人物 {姓名: "刘备"})-[*]-(b:人物 {姓名: "曹操"
RETURN length(path) AS 路径长度
```

变长路径——图查询的杀手级特性

变长路径是图数据库区别于关系数据库的核心优势。在关系数据库中，查找”朋友的朋友的朋友”需要多次 JOIN，性能随着深度急剧下降。而在图数据库中，这是最自然不过的操作：

```
// 固定长度：朋友的朋友（2 跳）
MATCH (a:人物 {姓名: "刘备"})-[:结拜兄弟*2]-(b:人物)
RETURN b.姓名

// 变长路径：1 到 3 跳
MATCH (a:人物 {姓名: "刘备"})-[:结拜兄弟*1..3]-(b:人物)
RETURN DISTINCT b.姓名

// 最短路径
MATCH (a:人物 {姓名: "刘备"}), (b:人物 {姓名: "曹操"})
MATCH p = shortestPath((a)-[*]-(b))
RETURN p

// 所有最短路径
MATCH (a:人物 {姓名: "刘备"}), (b:人物 {姓名: "孙权"})
MATCH p = allShortestPaths((a)-[*]-(b))
RETURN p

// 带关系类型过滤的路径
MATCH path = (a:人物 {姓名: "刘备"})-[:效力|结拜兄弟*]-(b:势力
RETURN path
```

模式组合——描述复杂的图结构

Cypher 允许你在一个 MATCH 子句中描述非常复杂的图模式：

```
// 查找"三角关系": A 认识 B, B 认识 C, A 也认识 C
MATCH (a:人物)-[:结拜兄弟]-(b:人物),
      (b)-[:结拜兄弟]-(c:人物),
      (a)-[:结拜兄弟]-(c)
WHERE a.姓名 <> c.姓名  // 排除自己
RETURN a.姓名, b.姓名, c.姓名

// 查找"效力于同一势力"的人
MATCH (a:人物)-[:效力]->(k:势力)<-[:效力]-(b:人物)
WHERE a <> b AND a.姓名 < b.姓名  // 去重
RETURN a.姓名, b.姓名, k.名称

// 查找"文武搭档"
MATCH (warrior:武将)-[:效力]->(k:势力)<-[:效力]-(advisor:文官
RETURN k.名称 AS 势力,
       warrior.姓名 AS 武将,
       advisor.姓名 AS 文官
```

常用子句补充

SET——更新属性

```
// 设置属性
MATCH (p:人物 {姓名: "刘备"})
SET p.武力值 = 72, p.称号 = "昭烈帝"

// 添加标签
MATCH (p:人物 {姓名: "诸葛亮"})
SET p:军师:蜀汉
RETURN p

// 移除属性
MATCH (p:人物 {姓名: "刘备"})
SET p.临时标记 = null  // 设为 null 等于删除

// 移除标签
MATCH (p:人物 {姓名: "诸葛亮"})
REMOVE p:军师
```

DELETE——删除节点和关系

```
// 删除一个节点（该节点不能有关系）
MATCH (p:人物 {姓名: "临时角色"})
DELETE p

// 删除节点和它的所有关系
MATCH (p:人物 {姓名: "临时角色"})
DETACH DELETE p

// 删除特定关系
MATCH (p:人物 {姓名: "徐庶"})-[r:效力]->(k:势力 {名称: "蜀汉"}
DELETE r

// 删除所有数据（慎用！）
MATCH (n)
DETACH DELETE n
```

WITH——中间结果传递

`WITH` 类似于 SQL 中的子查询或 CTE，让你可以将中间结果传递给下一步查询：

```
// 找出每个势力武力值最高的人物
MATCH (p:人物)-[:效力]->(k:势力)
WITH k, p
ORDER BY p.武力值 DESC
WITH k, COLLECT(p)[0] AS 最强武将
RETURN k.名称 AS 势力, 最强武将.姓名 AS 最强武将, 最强武将.武力值

// 链式操作
MATCH (p:人物)
WHERE p.武力值 > 80
WITH p, p.武力值 + p.智力值 AS 综合能力
WHERE 综合能力 > 150
RETURN p.姓名, 综合能力
ORDER BY 综合能力 DESC
```

UNWIND——展开列表

```
// 将列表展开为行
UNWIND ["刘备", "关羽", "张飞"] AS 名字
MATCH (p:人物 {姓名: 名字})
RETURN p.姓名, p.字

// 批量创建
UNWIND [
  {姓名: "魏延", 字: "文长", 武力值: 90},
  {姓名: "姜维", 字: "伯约", 武力值: 89},
  {姓名: "马岱", 字: "伯瞻", 武力值: 80}
] AS data
CREATE (p:人物:武将)
SET p = data
```

动手练习：在 Neo4j Browser 中编写 20 条 Cypher 查询

环境准备

方式一：Neo4j Desktop（推荐本地使用）

1. 访问 https://neo4j.com/download/ 下载 Neo4j Desktop
2. 安装后创建一个新数据库（Community Edition 即可）
3. 启动数据库，打开 Neo4j Browser

方式二：Neo4j Sandbox（免费在线体验）

1. 访问 https://sandbox.neo4j.com/
2. 创建一个空白 Sandbox
3. 直接在浏览器中使用

方式三：Docker

```
docker run -d \
  --name neo4j \
  -p 7474:7474 -p 7687:7687 \
  -e NEO4J_AUTH=neo4j/password123 \
  neo4j:latest
```

启动后访问 `http://localhost:7474` 打开 Neo4j Browser。

初始化数据

在 Neo4j Browser 中执行以下命令，创建三国人物的完整数据：

```
// 清空已有数据
MATCH (n) DETACH DELETE n;

// 创建势力节点
CREATE (shu:势力 {名称: "蜀汉", 都城: "成都", 建立年份: 221})
CREATE (wei:势力 {名称: "曹魏", 都城: "洛阳", 建立年份: 220})
CREATE (wu:势力 {名称: "东吴", 都城: "建业", 建立年份: 229});

// 创建人物节点
CREATE (liubei:人物:君主 {姓名: "刘备", 字: "玄德", 出生年份: 1
CREATE (guanyu:人物:武将 {姓名: "关羽", 字: "云长", 出生年份: 1
CREATE (zhangfei:人物:武将 {姓名: "张飞", 字: "翼德", 出生年份:
CREATE (zhugeliang:人物:文官:军师 {姓名: "诸葛亮", 字: "孔明",
CREATE (zhaoyun:人物:武将 {姓名: "赵云", 字: "子龙", 出生年份:
CREATE (huangzhong:人物:武将 {姓名: "黄忠", 字: "汉升", 出生年
CREATE (caocao:人物:君主 {姓名: "曹操", 字: "孟德", 出生年份: 1
CREATE (simayi:人物:文官 {姓名: "司马懿", 字: "仲达", 出生年份:
CREATE (xuchu:人物:武将 {姓名: "许褚", 字: "仲康", 出生年份: 17
CREATE (sunquan:人物:君主 {姓名: "孙权", 字: "仲谋", 出生年份:
CREATE (zhouyu:人物:武将 {姓名: "周瑜", 字: "公瑾", 出生年份: 1
CREATE (luxun:人物:武将:文官 {姓名: "陆逊", 字: "伯言", 出生年份
CREATE (lvbu:人物:武将 {姓名: "吕布", 字: "奉先", 出生年份: 161

// 创建效力关系
MATCH (liubei:人物 {姓名: "刘备"}), (shu:势力 {名称: "蜀汉"})
CREATE (liubei)-[:效力 {起始年份: 221}]->(shu);

MATCH (guanyu:人物 {姓名: "关羽"}), (shu:势力 {名称: "蜀汉"})
CREATE (guanyu)-[:效力 {起始年份: 200}]->(shu);

MATCH (zhangfei:人物 {姓名: "张飞"}), (shu:势力 {名称: "蜀汉"}
CREATE (zhangfei)-[:效力 {起始年份: 200}]->(shu);

MATCH (zhugeliang:人物 {姓名: "诸葛亮"}), (shu:势力 {名称: "蜀
```

```
CREATE (zhugeliang)-[:效力 {起始年份: 207}]->(shu);

MATCH (zhaoyun:人物 {姓名: "赵云"}), (shu:势力 {名称: "蜀汉"})
CREATE (zhaoyun)-[:效力 {起始年份: 200}]->(shu);

MATCH (huangzhong:人物 {姓名: "黄忠"}), (shu:势力 {名称: "蜀汉
CREATE (huangzhong)-[:效力 {起始年份: 214}]->(shu);

MATCH (caocao:人物 {姓名: "曹操"}), (wei:势力 {名称: "曹魏"})
CREATE (caocao)-[:效力 {起始年份: 220}]->(wei);

MATCH (simayi:人物 {姓名: "司马懿"}), (wei:势力 {名称: "曹魏"}
CREATE (simayi)-[:效力 {起始年份: 208}]->(wei);

MATCH (xuchu:人物 {姓名: "许褚"}), (wei:势力 {名称: "曹魏"})
CREATE (xuchu)-[:效力 {起始年份: 197}]->(wei);

MATCH (sunquan:人物 {姓名: "孙权"}), (wu:势力 {名称: "东吴"})
CREATE (sunquan)-[:效力 {起始年份: 229}]->(wu);

MATCH (zhouyu:人物 {姓名: "周瑜"}), (wu:势力 {名称: "东吴"})
CREATE (zhouyu)-[:效力 {起始年份: 200}]->(wu);

MATCH (luxun:人物 {姓名: "陆逊"}), (wu:势力 {名称: "东吴"})
CREATE (luxun)-[:效力 {起始年份: 203}]->(wu);

// 创建人物间关系
MATCH (a:人物 {姓名: "刘备"}), (b:人物 {姓名: "关羽"})
CREATE (a)-[:结拜兄弟 {地点: "桃园", 年份: 184}]->(b);

MATCH (a:人物 {姓名: "刘备"}), (b:人物 {姓名: "张飞"})
CREATE (a)-[:结拜兄弟 {地点: "桃园", 年份: 184}]->(b);

MATCH (a:人物 {姓名: "关羽"}), (b:人物 {姓名: "张飞"})
CREATE (a)-[:结拜兄弟 {地点: "桃园", 年份: 184}]->(b);
```

```
MATCH (a:人物 {姓名: "刘备"}), (b:人物 {姓名: "诸葛亮"})
CREATE (a)-[:三顾茅庐 {年份: 207}]->(b);

MATCH (a:人物 {姓名: "诸葛亮"}), (b:人物 {姓名: "周瑜"})
CREATE (a)-[:对手]->(b);

MATCH (a:人物 {姓名: "曹操"}), (b:人物 {姓名: "刘备"})
CREATE (a)-[:对手]->(b);

MATCH (a:人物 {姓名: "吕布"}), (b:人物 {姓名: "曹操"})
CREATE (a)-[:对手]->(b);
```

20 条 Cypher 查询练习

查询 1：查看所有人物

```
MATCH (p:人物)
RETURN p.姓名, p.字, labels(p) AS 标签
ORDER BY p.姓名
```

查询 2：查看所有武将

```
MATCH (p:武将)
RETURN p.姓名, p.武力值
ORDER BY p.武力值 DESC
```

查询 3：找出蜀汉的所有人物

```
MATCH (p:人物)-[:效力]->(k:势力 {名称: "蜀汉"})
RETURN p.姓名, p.字
```

查询 4：找出武力值超过 90 的人物

```
MATCH (p:人物)
WHERE p.武力值 > 90
RETURN p.姓名, p.武力值
ORDER BY p.武力值 DESC
```

查询 5：找出智勇双全的人（武力 > 70 且智力 > 70）

```
MATCH (p:人物)
WHERE p.武力值 > 70 AND p.智力值 > 70
RETURN p.姓名, p.武力值, p.智力值,
       p.武力值 + p.智力值 AS 综合能力
ORDER BY 综合能力 DESC
```

查询 6：统计每个势力的人数

```
MATCH (p:人物)-[:效力]->(k:势力)
RETURN k.名称 AS 势力, COUNT(p) AS 人数
ORDER BY 人数 DESC
```

查询 7：每个势力的平均武力值和平均智力值

```
MATCH (p:人物)-[:效力]->(k:势力)
RETURN k.名称 AS 势力,
       ROUND(AVG(p.武力值), 1) AS 平均武力,
       ROUND(AVG(p.智力值), 1) AS 平均智力
ORDER BY 平均武力 DESC
```

查询 8：找出刘备的结拜兄弟

```
MATCH (a:人物 {姓名: "刘备"})-[:结拜兄弟]-(b:人物)
RETURN b.姓名 AS 兄弟, b.字 AS 字
```

查询 9：查找”桃园结义”的所有成员

```
MATCH (a:人物)-[r:结拜兄弟 {地点: "桃园"}]-(b:人物)
RETURN DISTINCT a.姓名 AS 成员
```

查询 10：找出谁和诸葛亮有关系

```
MATCH (a:人物 {姓名: "诸葛亮"})-[r]-(b:人物)
RETURN type(r) AS 关系类型, b.姓名 AS 相关人物
```

查询 11：找出每个势力武力值最高的人

```
MATCH (p:人物)-[:效力]->(k:势力)
WITH k, p
ORDER BY p.武力值 DESC
WITH k, COLLECT(p)[0] AS top
RETURN k.名称 AS 势力, top.姓名 AS 最强武将, top.武力值 AS 武力
```

查询 12：查找所有”对手”关系

```
MATCH (a:人物)-[r:对手]->(b:人物)
RETURN a.姓名 + " 的对手是 " + b.姓名 AS 对手关系
```

查询 13：找出出生最早的人物（最年长）

```
MATCH (p:人物)
RETURN p.姓名, p.出生年份
ORDER BY p.出生年份 ASC
LIMIT 5
```

查询 14：查找和刘备在同一势力的人物

```
MATCH (liubei:人物 {姓名: "刘备"})-[:效力]->(k:势力),
      (other:人物)-[:效力]->(k)
WHERE other <> liubei
RETURN other.姓名 AS 同僚, other.字 AS 字
```

查询 15：统计所有关系类型

```
MATCH ()-[r]->()
RETURN type(r) AS 关系类型, COUNT(r) AS 数量
ORDER BY 数量 DESC
```

查询 16：查找从刘备出发 2 跳可达的所有人物

```
MATCH path = (a:人物 {姓名: "刘备"})-[*1..2]-(b:人物)
WHERE a <> b
RETURN DISTINCT b.姓名 AS 可达人物, length(path) AS 跳数
ORDER BY 跳数
```

查询 17：查找刘备和曹操之间的最短路径

```
MATCH (a:人物 {姓名: "刘备"}), (b:人物 {姓名: "曹操"})
MATCH p = shortestPath((a)-[*]-(b))
RETURN p
```

查询 18：找出有”号”的人物

```
MATCH (p:人物)
WHERE p.号 IS NOT NULL
RETURN p.姓名, p.号
```

查询 19：创建一个新人物并建立关系

```
MATCH (shu:势力 {名称: "蜀汉"})
CREATE (p:人物:武将 {姓名: "关平", 字: "坦之", 武力值: 82, 智力
CREATE (p)-[:效力]->(shu)
CREATE (p)-[:义子 {关系: "义父子"}]->(:人物 {姓名: "关羽"})
RETURN p
```

查询 20：综合分析——谁是三国最全面的武将?

```
MATCH (p:武将)-[:效力]->(k:势力)
WHERE p.武力值 IS NOT NULL AND p.智力值 IS NOT NULL
WITH p, k,
     p.武力值 * 0.6 + p.智力值 * 0.4 AS 综合评分
RETURN p.姓名 AS 姓名,
       k.名称 AS 势力,
       p.武力值 AS 武力,
       p.智力值 AS 智力,
       ROUND(综合评分, 1) AS 综合评分
ORDER BY 综合评分 DESC
LIMIT 5
```

动手练习

练习 1：构建你的朋友圈知识图谱

用 Cypher 在 Neo4j 中建模你的真实社交圈：

1. 创建至少 8 个代表你朋友的节点，包含属性：姓名、职业、城市、兴趣爱好
2. 创建以下关系：
3. `朋友` 关系（你和你的朋友之间）
4. `同事` 关系（有工作关系的朋友之间）
5. `校友` 关系（有同学关系的朋友之间）
6. 在关系上添加属性：认识年份、认识途径

查询挑战： – 找出和你同一个城市的朋友 – 找出你朋友的"朋友"（2 度人脉） – 找出通过"工作"认识的朋友有哪些 – 统计每个城市有多少朋友 – 找出兴趣爱好有交集的朋友（比如都喜欢"读书"的）

练习 2：电影推荐系统

使用 Neo4j 自带的 Movie 示例数据库（在 Browser 中输入 `:play movie-graph`），回答以下问题：

1. Tom Hanks 演过哪些电影？
2. 和 Tom Hanks 合作过的演员有哪些？
3. 哪些导演同时导演过多部获奖电影？
4. 找出两个演员之间共同出演的电影数量（合作次数排行榜）

练习 3：Cypher vs SPARQL 对比

将上一章的三国 RDF 数据用属性图重新建模，然后对比 SPARQL 和 Cypher 在以下查询上的差异：

1. 找出蜀汉所有武将
2. 统计各势力人数
3. 找出结拜兄弟关系

写出两种语言的版本，感受它们在语法和思维方式上的不同。

延伸阅读

1. **Neo4j 官方文档**：https://neo4j.com/docs/ Neo4j 的完整文档，包括 Cypher 手册、性能调优、集群部署等。Cypher Refcard（速查卡）特别推荐打印出来贴在桌边。
2. **Cypher Refcard**：https://neo4j.com/docs/cypher-refcard/current/ Cypher 语法的一页速查卡，涵盖所有关键词和函数。
3. **openCypher 项目**：https://www.opencypher.org/ Cypher 的开源标准化项目，包含语言规范、兼容性测试等。
4. **ISO GQL 标准**：https://www.iso.org/standard/76120.html 即将成为 ISO 国际标准的图查询语言 GQL，以 openCypher 为基础，融合了 SQL/PGQ 和 PGQL 的特性。

5. **Graph Data Modeling**：
   https://neo4j.com/docs/getting-started/data-modeling/ Neo4j 的数据建模指南，教你如何将现实世界的问题转化为属性图模型。
6. **Neo4j Graph Academy**：
   https://graphacademy.neo4j.com/ Neo4j 官方提供的免费在线课程，涵盖 Cypher 入门、图数据建模、GDS（图数据科学）等。非常适合系统性学习。
7. **Gremlin 查询语言**：
   https://tinkerpop.apache.org/docs/current/reference/ Apache TinkerPop 的 Gremlin 是另一种流行的图遍历语言，适用于 JanusGraph、Amazon Neptune 等图数据库。和 Cypher 的声明式风格不同，Gremlin 是命令式的。

本章小结

本章我们学习了属性图模型和 Cypher 查询语言，这是当前工业界最主流的图数据库技术栈。

核心知识回顾：

1. 属性图模型 由节点（Node）、关系（Relationship）、标签（Label）和属性（Properties）四个元素组成。相比 RDF，它的优势在于节点和关系都可以直接携带属性，数据建模更加直观和灵活。

2. 属性图 vs RDF 各有适用场景。RDF 适合需要严格语义推理和开放数据互联的场景；属性图适合快速工程开发和复杂关系建模的场景。在实际项目中，属性图的使用频率更高。

3. Cypher 查询语言 用可视化的 ASCII 艺术模式来描述图查询，核心关键词包括： – `MATCH`：匹配图模式 – `CREATE`：创建节点和关系 – `MERGE`：智能创建（有则匹配，无则创建） – `WHERE`：过滤条件 – `RETURN`：返回结果 – `WITH`：中间结果传递 – `SET` / `DELETE`：更新和删除

4. 模式匹配 是 Cypher 最强大的能力，特别是变长路径（`-[*1..3]-`）和最短路径（`shortestPath`），让图数据库在处理多跳关系查询时远超关系数据库。

关键对比：

表 9–2

特性	RDF + SPARQL	属性图 + Cypher
数据单元	三元组	节点–关系–属性
查询语法	图模式匹配（变量绑定）	ASCII 艺术模式
多跳查询	需要属性路径或递归	原生支持（变长路径）
属性存储	间接（通过三元组）	直接（键值对）
推理能力	强大（OWL）	有限

下一章，我们将学习如何用 Python 来编程操作图数据库，把 Cypher 查询嵌入到你的应用程序中，实现自动化的知识图谱构建和查询。从”手动在浏览器里敲命令”进阶到”用代码驱动知识图谱”。

准备好了吗？让我们继续前进！

第六章 本体设计：给世界画一张地图

> *

—— 路德维希·维特根斯坦*

> *“The limits of my language mean the limits of my world.”*

引子：一个让程序员崩溃的周末

我的朋友小李是个很有生活情调的人。他不仅爱做饭，还想把做饭这件事”数字化”。有一天，他跑来找我，兴奋地说要做一个”智能菜谱推荐系统”——用户说”我想吃点清淡的、有豆腐的”，系统就能推荐一道合适的菜。

听起来不难，对吧？

小李兴冲冲地打开数据库，开始建表。第一张表叫 `recipe`，字段有 name、ingredients、taste……写着写着，他停住了。

“等等，’清蒸鲈鱼’里的’鲈鱼’是一种食材，但’鲈鱼’本身也有产地、季节、替代品的信息。我应该为每种食材单独建一张表吗？那’清蒸’这种烹饪方式呢？’清淡’这种口味呢？川菜和粤菜的分类体系完全不同啊……还有，’麻辣’到底是口味还是菜系特征？”

三个小时后，小李的白板上画满了方框和箭头，像一张巨大的蜘蛛网。他看着这团乱麻，发出了灵魂的呐喊：**“这个世界上的东西，到底该怎么分类才合理啊！”**

后来他又跑来问我：”我花了一整天设计数据库，结果越想越复杂，最后连自己都理不清了。有没有什么科学的方法来做这件事？”

我告诉他：”你遇到的问题，在知识工程领域有一个专门的名字——**本体设计（Ontology Design）**。而且你绝对不是第一个被这个问题折磨的人，从古至今的哲学家和计算机科学家都在思考这件事。”

如果你也有过类似的困惑——面对一堆杂乱的信息，不知道怎么组织才合理——恭喜你，你已经触碰到了知识工程中最核心、也最迷人的问题。

本章我们就来聊聊：如何给纷繁复杂的世界画一张清晰的”地图”，让计算机也能理解事物之间的关系。

什么是本体（Ontology）

从哲学到计算机科学

“Ontology”这个词最早来自哲学，研究的是”存在的本质”——什么东西存在？事物之间有什么关系？世界的本质是什么？古希腊哲学家亚里士多德就写过《范畴篇》，把世间万物分为十个基本范畴，这可以说是最早的”本体设计”了。

到了中世纪，哲学家们为了”共相问题”争得不可开交——“红色”是一种独立的存在，还是只是我们对红色物体的抽象？这种争论持续了上千年。

别被这些高大上的名字吓到。在计算机科学里，本体其实就是一个很朴实的东西：

> 本体是对某个领域中概念及其关系的形式化描述。

用人话说就是：**你先把一个领域里有哪些”东西”想清楚，再把它们之间的”关系”理明白，然后用一种计算机能理解的格式写下来。**

还记得小李的困惑吗？他其实就是在做本体设计——只不过他用的工具是白板和脑子，而不是形式化的语言。如果他用 Protégé 这样的专业工具，先定义清楚”菜品”“食材”“菜系”“烹饪方式”这些概念之间的关系，就不会陷入”越理越乱”的困境了。

为什么需要本体？

你可能会问：直接写代码不行吗？为什么非要先搞一个”本体”？

让我用一个类比来说明。想象你要建造一栋大楼。你可以直接搬砖头、浇水泥，边盖边想”这里该放卧室还是客厅”。但更聪明的做法是先画一张建筑蓝图——哪面墙在哪里，哪个房间多大，水管怎么走，电路怎么布线。

本体就是知识图谱的”建筑蓝图”。它解决三个关键问题：

1. **共识问题**：团队里的所有人对”什么是电影”“什么是导演”有统一的理解
2. **推理问题**：有了形式化的定义，计算机可以自动推理出隐含的关系

3. **复用问题**：一个好的本体可以被多个系统复用，而不是每次都从零开始

本体的三大核心要素

一个完整的本体，通常包含三类核心要素：

1. 类（Class / Concept）

类就是领域中的”概念”或”类别”。它是本体中最基本的构建块。比如在电影领域，我们会定义这样的类：

- `Person`（人物）——所有跟电影相关的人
- `Movie`（电影）——所有的电影作品
- `Genre`（类型）——电影的分类
- `Director`（导演）——拍电影的人
- `Award`（奖项）——各种电影奖项

类是可以有层级的。比如 `Director` 可以是 `Person` 的子类，`Actor` 也可以是 `Person` 的子类。这就形成了一个**类的层级结构（Class Hierarchy）**。层级结构是本体设计中最核心的组织方式，它让我们可以用”继承”的思想来管理概念。子类自动继承父类的所有属性和关系，同时可以添加自己特有的特征。

```
Thing（万物）
├── Person（人物）
│   ├── Actor（演员）
│   ├── Director（导演）
│   └── Screenwriter（编剧）
├── Movie（电影）
│   ├── FeatureFilm（故事片）
│   ├── Documentary（纪录片）
│   └── ShortFilm（短片）
├── Genre（类型）
│   ├── Action（动作）
│   ├── Comedy（喜剧）
│   ├── Drama（剧情）
│   └── SciFi（科幻）
└── Award（奖项）
    ├── Oscar（奥斯卡）
    ├── GoldenGlobe（金球奖）
    └── PalmeDor（金棕榈奖）
```

这种层级结构的好处是：当我们说”诺兰是一位Director”时，系统自动知道”诺兰也是一个Person”，因此他拥有Person的所有属性（如出生日期、国籍等）。

2. 关系（Property / Relationship）

关系描述的是类与类之间、或实例与实例之间的联系。没有关系的类只是一堆孤立的标签，关系才是让知识”活”起来的关键。

关系分为两大类：

对象属性（Object Property）——连接两个实例： - `actedIn`（出演）：Actor → Movie，表示”演员出演了电影” - `directedBy`（导演是）：Movie → Director，表示”电影由某导演执导” - `belongsToGenre`（属于类型）：Movie → Genre，表示”电影属于某类型” - `wonAward`（获得奖项）：Person → Award，表示”某人获得了某奖”

数据属性（Data Property）——连接实例和数据值： - `movieTitle`（电影名称）：Movie → string - `releaseDate`（上映日期）：Movie → date - `rating`（评分）：Movie → float - `personName`（姓名）：Person → string

关系也可以有层级。比如 `directedBy` 和 `actedIn` 都可以是 `involvedIn`（参与）的子关系。当你问”谁参与了这部电影？”时，系统会自动查找所有子关系，包括导演、演员、编剧等。

3. 约束（Constraint / Axiom）

约束是对关系和类的”规矩”——它告诉系统什么是合法的、什么是不合法的。比如：

- 一部电影至少有一个导演（最小基数约束：`directedBy min 1`）
- 一部电影有且只有一个上映日期（精确基数约束：`releaseDate exactly 1`）
- `bornOn` 的值必须是 `xsd:date` 类型（数据类型约束）
- 一个人的出生日期不能晚于当前日期（值域约束）

- `Actor` 和 `Director` 不互斥——同一个人可以同时是演员和导演

约束就像是地图上的”交通规则”——它告诉你哪些路可以走，哪些路不能走，哪些路有限制。好的约束设计可以防止”脏数据”进入知识图谱，从源头保证数据质量。

一个直觉性的类比

如果你觉得上面的概念还是有点抽象，让我用一个类比来帮你建立直觉。假设你在玩一款RPG角色扮演游戏：

表 10–1

本体概念	类比：RPG游戏
类（Class）	角色职业：战士、法师、弓箭手
子类（SubClass）	法师 → 火法师、冰法师、雷法师
实例（Instance）	你创建的那个具体的火法师角色”烈焰之怒”
属性（Property）	角色的属性值：力量80、智力150、敏捷60
关系（Relationship）	师徒关系、队友关系、敌对关系
约束（Constraint）	法师不能装备双手斧；战士不能使用治疗术
推理（Inference）	因为你是火法师，所以你自动免疫火焰伤害

是不是突然觉得没那么难了？本体设计本质上就是为你的知识世界制定”游戏规则”——有哪些角色，角色之间有什么关系，什么行为是被允许的。

本体设计方法论：自上而下 vs 自下而上

明白了本体是什么之后，下一个问题就是：**怎么设计一个好的本体？**

这就像问”怎么画一张好的地图”——没有唯一正确的答案，但有一些经过实践验证的方法论。主流的本体设计方法有两种：**自上而下**和**自下而上**。每种方法都有自己的适用场景，就像不同的工具适合不同的工作。

自上而下（Top-Down）

自上而下的方法是从最抽象、最顶层的概念开始，逐步细化到具体的概念。这是一种”先规划蓝图，再填充细节”的思路。

步骤：

1. **确定领域和范围**：你的本体要覆盖什么领域？要回答哪些问题？
2. **列举核心概念**：先列出最重要的顶层概念（通常 5–10 个）
3. **建立层级结构**：确定概念之间的上下级关系
4. **定义关系和属性**：为概念添加横向的关系和自身的属性
5. **添加约束**：设定基数、值域、定义域等限制
6. **验证实例**：用真实数据检验本体是否合理

适用场景： – 领域知识结构比较清晰、稳定（如生物学分类、图书馆分类法） – 有领域专家可以参与设计 – 需要严格的逻辑推理能力 – 项目周期充裕，有足够的时间做前期规划

例子： 设计一个”动物分类本体”

生物学已经有了非常成熟的分类体系——界、门、纲、目、科、属、种。自上而下的设计几乎是自然发生的：

```
生物界（Living Things）
├── 动物界（Animalia）
│   ├── 脊索动物门（Chordata）
│   │   ├── 哺乳纲（Mammalia）
│   │   │   ├── 灵长目（Primates）
│   │   │   │   ├── 人科（Hominidae）
│   │   │   │   └── 猴科（Cercopithecidae）
│   │   │   ├── 食肉目（Carnivora）
│   │   │   │   ├── 猫科（Felidae）
│   │   │   │   └── 犬科（Canidae）
│   │   │   └── ...
│   │   ├── 鸟纲（Aves）
│   │   └── 爬行纲（Reptilia）
│   └── 节肢动物门（Arthropoda）
│       ├── 昆虫纲（Insecta）
│       └── 甲壳纲（Crustacea）
└── 植物界（Plantae）
    ├── 被子植物门（Angiosperms）
    └── 裸子植物门（Gymnosperms）
```

这个层级结构非常清晰，自上而下的设计很自然，因为生物学分类本身就是一个经过数百年验证的知识体系。

自下而上（Bottom-Up）

自下而上的方法是从具体的实例和数据出发，逐步归纳抽象出概念和关系。这是一种”先观察现象，再总结规律”的思路。

步骤：

1. **收集实例数据**：从真实数据中收集具体的实例
2. **发现模式**：观察实例的共同特征和关系
3. **归纳概念**：将相似的实例归为同一类
4. **抽象层级**：建立类之间的层级关系
5. **定义关系和属性**：补充概念之间的关系
6. **迭代优化**：随着新数据的加入，不断调整本体

适用场景： – 领域知识结构不明确、快速变化（如互联网产品、社交媒体） – 有大量现成数据可以利用 – 需要快速迭代和验证 – 没有领域专家可以咨询

例子： 设计一个”美食推荐本体”

美食领域的分类可没有生物学那么成熟。你可能不知道中国八大菜系的完整理论体系，但你可以从具体的菜品开始观察和归纳：

```
第一步：收集实例
- 宫保鸡丁（鸡肉、花生、辣椒、川菜、爆炒）
- 麻婆豆腐（豆腐、牛肉末、花椒、川菜、烧）
- 红烧肉（五花肉、酱油、糖、鲁菜、红烧）
- 白切鸡（鸡肉、姜、葱、粤菜、水煮）
- 酸菜鱼（鱼片、酸菜、辣椒、川菜、煮）

第二步：发现模式
观察上面这些菜品，我们发现它们都有一些共同的维度：
- 都有具体的食材（鸡肉、豆腐、五花肉……）
- 都属于某个菜系（川菜、鲁菜、粤菜）
- 都使用了某种烹饪方式（爆炒、烧、红烧、水煮）
- 都有某种口味特征（辣、麻、甜、鲜）

第三步：归纳出概念和层级
- 菜品（Dish）：宫保鸡丁、麻婆豆腐……
- 食材（Ingredient）
  ├── 肉类（Meat）：鸡肉、猪肉、牛肉、鱼肉
  ├── 蔬菜（Vegetable）：花生、辣椒、酸菜
  ├── 豆制品（SoyProduct）：豆腐
  └── 调味料（Seasoning）：酱油、糖、花椒
- 菜系（Cuisine）：川菜、鲁菜、粤菜
- 烹饪方式（CookingMethod）：爆炒、红烧、水煮
- 口味（Flavor）：辣、麻、甜、鲜、咸
```

实践中：混合方法（推荐）

在实际项目中，很少有人纯粹使用自上而下或自下而上的方法。更常见的是**混合方法**，也是我最推荐的方式：

1. **先用自上而下建立一个粗略的骨架**——确定 3–5 个最顶层的概念和它们之间的关系

2. **再用自下而上的方法，从真实数据中发现遗漏的概念**——看看实际数据中有哪些东西是你的骨架没有覆盖的
3. **反复迭代，直到本体足够完善**——每轮迭代都会让本体变得更精确

这就像画地图——你先画大洲大洋的轮廓（自上而下），再根据国家、城市的实际分布来填充细节（自下而上），最后反复校对修改。世界地图从来不是一个人一次性画完的，好的本体设计也是如此。

本体设计的七步法

斯坦福大学医学院的 Natalya Noy 和 Deborah McGuinness 在2001年发表了一篇经典论文《Ontology Development 101》，提出了一套系统化的本体开发指南，被称为”七步法”。这篇论文至今仍是本体设计领域被引用最多的参考资料之一。

我把它简化为更适合初学者的版本，并加入了一些实践中的经验教训：

第一步：确定领域和范围

在动手之前，先问自己几个问题，把方向定清楚：

- 我的本体要覆盖什么领域？（电影？美食？医疗？）
- 谁会使用这个本体？（开发者？领域专家？终端用户？）
- 用户会用它来回答什么样的问题？

- 这个本体的边界在哪里？（哪些概念要包含，哪些不包含？）

实用技巧：列出”能力问题”（Competency Questions）

能力问题就是”你的本体应该能回答的问题”。这是本体设计中最重要的一步——如果你的本体设计完成后，这些问题都能被回答，说明你的设计是合格的。

比如对于电影本体，能力问题可能包括：

```
Q1: "周星驰导演了哪些电影？"
Q2: "2020年以后上映的科幻片有哪些？"
Q3: "获得过奥斯卡最佳导演奖的人有谁？"
Q4: "同时担任导演和主演的电影有哪些？"
Q5: "和《肖申克的救赎》同类型的电影推荐？"
Q6: "某部电影的所有演职员表？"
Q7: "某个演员和哪些导演合作过？"
```

这些问题不仅指导了本体的设计方向，将来还可以直接转化为查询语句来验证本体的有效性。

第二步：考察已有本体（不要重复造轮子）

很多领域已经有成熟的本体可以复用。在开始设计之前，先花几个小时搜索一下：

- **Schema.org**：Google、Bing、Yandex等共同维护的通用网页内容本体，覆盖面极广
- **DBpedia Ontology**：基于Wikipedia的通用本体，有数百万个概念
- **FOAF**（Friend of a Friend）：描述人物和社会关系的本体

- **Dublin Core**：描述数字资源元数据的国际标准
- **GoodRelations**：电子商务本体
- **OpenKG**：中文开放知识图谱社区，有大量中文本体资源

复用已有本体的好处不仅是节省时间，更重要的是能保证**互操作性**——你的知识图谱可以和已有的知识图谱对接。

第三步：列举核心概念

把领域中最重要的概念列出来。不用纠结完整性，先列 10–20 个最核心的概念。这一步可以用脑暴的方式，把所有想到的概念都写下来，后面再做筛选和整理。

第四步：建立类的层级

把概念组织成树状的层级结构。常见的层级模式有：

- **按种类分类（is-a）**：Movie → FeatureFilm, Documentary, ShortFilm
- **按部分分类（part-of）**：Car → Engine, Wheel, Body, Interior
- **按角色分类（role）**：Person → Actor, Director, Producer

注意避免两个常见问题：层级太深（超过5层就很难维护）和层级太浅（所有概念都平铺在第一层，失去了分类的意义）。

第五步：定义属性（Properties）

属性分为两类，一定要区分清楚：

- **数据属性（Datatype Property）**：连接到数据值（字符串、数字、日期等）

- 例如：`Movie.releaseDate → "2023-01-15"`
- **对象属性（Object Property）**：连接到其他实例（实体之间的关系）
- 例如：`Movie.director → Person("Christopher Nolan")`

对于每个属性，需要明确定义： – **定义域（Domain）**：这个属性可以用在哪些类上？ – **值域（Range）**：这个属性的值是什么类型？ – **基数（Cardinality）**：每个实例可以有多少个值？

第六步：定义约束（Constraints/Facets）

约束让本体从”描述性的”变成”可推理的”：

- **基数约束**：一部电影可以有1个或多个类型，但至少有1个
- **值域约束**：`releaseDate` 的值必须是日期类型
- **定义域约束**：`directedBy` 只能用在 `Movie` 类上
- **互斥约束**：`Vegetarian` 和 `MeatDish` 不能同时为真
- **传递性约束**：如果A是B的一部分，B是C的一部分，那么A也是C的一部分

第七步：创建实例并验证

用真实数据创建几个实例，然后做三个检查：

1. **一致性检查**：推理机能否检测到逻辑矛盾？
2. **分类检查**：推理出的类层级是否符合你的预期？
3. **能力问题测试**：第一步列出的问题能回答吗？

建模工具实操

Protégé：本体设计的瑞士军刀

Protégé 是斯坦福大学开发的开源本体编辑器，是目前最流行、最成熟的本体建模工具。它支持 OWL（Web Ontology Language）标准，内置多种推理引擎，功能强大且完全免费。无论是学术研究还是工业项目，Protégé 都是首选工具。

安装和基本操作：

1. 从官网下载最新版 Protégé Desktop（推荐 5.6+ 版本）
2. 安装 Java 运行环境（JDK 11 或更高版本，Protégé 基于 Java 构建）
3. 打开 Protégé，选择 File → New Ontology
4. 设置本体的 IRI（Internationalized Resource Identifier），如 `http://example.org/movie-ontology`

核心界面解析：

Protégé 的主界面有几个关键标签页，每个都有明确的职责：

- **Entities 标签页**：管理类（Classes）、对象属性（Object Properties）、数据属性（Data Properties）、个体（Individuals）
- **Class hierarchy 面板**：以树状结构展示类的层级关系，可以拖拽调整
- **Property hierarchy 面板**：展示属性的层级关系
- **Individuals 面板**：创建和管理具体的实例

- **Axioms 面板**：查看和管理所有的公理和约束

  动手试一试：创建一个简单的电影本体

  跟着下面的步骤，你可以在30分钟内完成第一个本体：

```
1. 创建类（在 Entities → Classes 中操作）：
   - owl:Thing（已有的顶层类）
     ├── Person（人物）
     │   ├── Actor（演员）   ← 右键 Person → Add subclass
     │   └── Director（导演）
     ├── Movie（电影）
     ├── Genre（类型）
     └── Award（奖项）

2. 创建对象属性（在 Entities → Object Properties 中操作）：
   - actedIn: Domain=Actor, Range=Movie
     设置方式：选中 actedIn → Description → Add Domain → Act
                              → Add Range → Movie
   - directedBy: Domain=Movie, Range=Director
   - belongsToGenre: Domain=Movie, Range=Genre
   - wonAward: Domain=Person, Range=Award

3. 创建数据属性（在 Entities → Data Properties 中操作）：
   - movieTitle: Domain=Movie, Range=xsd:string
   - releaseDate: Domain=Movie, Range=xsd:date
   - rating: Domain=Movie, Range=xsd:float
   - personName: Domain=Person, Range=xsd:string

4. 创建实例（在 Entities → Individuals 中操作）：
   - 创建 Person 实例 "Christopher_Nolan"
     选中 → Types → 添加类型 Director
   - 创建 Movie 实例 "Inception"
     选中 → Types → 添加类型 Movie
   - 建立关系：Inception → Property assertions → directedBy
   - 设置属性：Inception → Data assertions → releaseDate → '

5. 启动推理机验证：
   菜单 → Reasoner → 选择 HermiT → Start reasoner
   观察：
```

```
  - 有没有红色的错误提示（逻辑矛盾）？
  - Christopher_Nolan 是否被自动识别为 Person 类型？
    （因为 Director 是 Person 的子类，所以实例自动继承父类）
```

Neo4j Data Modeler：图数据库视角

如果你最终要把本体落地到图数据库（这是目前最主流的知识图谱存储方案），Neo4j Data Modeler 是一个更贴近实践的选择。它帮你从”概念设计”过渡到”物理设计”。

与 Protégé 的区别：

表 10-2

特性	Protégé	Neo4j Data Modeler
关注点	逻辑推理、语义网标准	图数据建模、查询优化
表达方式	OWL 形式化语言	节点-关系图模型
输出格式	OWL/RDF/Turtle	Cypher 建图语句
推理能力	强（内置推理引擎）	弱（需要手动编码规则）
适用阶段	概念设计、评审	物理设计、实现
学习曲线	较陡（需要理解OWL）	较平缓（直观的图模型）

Neo4j Data Modeler 基本流程：

1. 打开 Neo4j Data Modeler（在线免费工具）
2. 添加节点标签（Node Labels）：`Person`，`Movie`，`Genre`

3. 添加关系类型（Relationship Types）：`ACTED_IN`，`DIRECTED`，`IN_GENRE`
4. 为节点添加属性（Properties）：如 Movie 的 title、releaseDate
5. 定义约束（Constraints）：如唯一性约束、存在性约束
6. 生成 Cypher 建图语句，直接在 Neo4j 中执行

两种工具如何配合

在实际项目中，我推荐的工作流是：

```
Protégé（概念设计）→ 团队评审确认 → Neo4j Data Modeler（物理建模
```

Protégé 负责”想清楚”——用形式化语言把概念和关系定义好，利用推理引擎验证逻辑正确性。Neo4j Data Modeler 负责”做出来”——把概念模型转化为图数据库的实际模型，考虑索引、约束、查询效率等工程问题。

两者的转换过程中需要注意：OWL 中的一些高级特性（如推理规则、传递属性）在 Neo4j 中需要手动编码为 Cypher 查询，不能自动继承。

案例实战：电影知识图谱的本体设计

让我们从头到尾走一遍电影知识图谱的本体设计流程。这个案例将贯穿本章的所有知识点，帮你把理论变成实践能力。

第一步：能力问题

首先明确我们的知识图谱要回答什么问题：

```
Q1: "《盗梦空间》的导演是谁？"
Q2: "诺兰导演的所有电影有哪些？"
Q3: "2020年后上映的评分8.0以上的科幻片？"
Q4: "同时获得过奥斯卡和金球奖的演员有谁？"
Q5: "和《肖申克的救赎》同类型的电影推荐？"
Q6: "某个演员的合作导演都有哪些？"
Q7: "某部电影的所有演职员表？"
Q8: "某位导演最常用的演员是谁？"
```

第二步：核心概念与层级

根据能力问题，我们提取出以下核心概念：

```
Person（人物）
  ├── Actor（演员）
  ├── Director（导演）
  ├── Screenwriter（编剧）
  └── Producer（制片人）

Movie（电影）
  ├── FeatureFilm（故事片）
  ├── DocumentaryFilm（纪录片）
  ├── AnimatedFilm（动画片）
  └── ShortFilm（短片）

Genre（类型）
  ├── Action（动作）
  ├── Comedy（喜剧）
  ├── Drama（剧情）
  ├── SciFi（科幻）
  ├── Horror（恐怖）
  └── Romance（爱情）

Studio（制片公司）
Award（奖项）
  ├── AcademyAward（奥斯卡奖）
  ├── GoldenGlobe（金球奖）
  └── CannesPalmeOr（戛纳金棕榈奖）

Country（国家）
Language（语言）
```

第三步：关系定义

这是本体设计中最关键的一步——定义概念之间的关系，让知识”连接”起来：

```
对象属性（连接两个实例）：
├── actedIn: Actor → Movie（演员出演电影）
│   └── 逆属性：hasActor: Movie → Actor
├── directedBy: Movie → Director（电影的导演）
│   └── 逆属性：directed: Director → Movie
├── writtenBy: Movie → Screenwriter（电影的编剧）
├── producedBy: Movie → Studio（电影的出品方）
├── belongsToGenre: Movie → Genre（电影的类型）
├── wonAward: Person → Award（获得的奖项）
│   └── 子属性：wonOscar（获得奥斯卡奖）
├── nominatedFor: Person → Award（提名奖项）
├── filmedIn: Movie → Country（拍摄地）
├── spokenLanguage: Movie → Language（对白语言）
└── sequelOf: Movie → Movie（续集关系，自引用）

数据属性（连接实例和数据值）：
├── Movie.title: xsd:string（电影名称）
├── Movie.releaseDate: xsd:date（上映日期）
├── Movie.runtime: xsd:integer（片长，分钟）
├── Movie.rating: xsd:float（评分 0–10）
├── Movie.boxOffice: xsd:decimal（票房，美元）
├── Movie.synopsis: xsd:string（剧情简介）
├── Person.name: xsd:string（姓名）
├── Person.birthDate: xsd:date（出生日期）
├── Person.nationality: xsd:string（国籍）
└── Person.biography: xsd:string（个人简介）
```

注意 `actedIn` 和 `hasActor` 是逆属性——如果”A出演了B”，那么自动可以推出”B有演员A”。这种逆属性声明可以大大减少冗余数据。

第四步：约束设计

约束确保数据的合法性和一致性：

```
Movie 的约束：
  - 必须有至少1个 director（minCardinality 1 on directedBy）
  - 必须有至少1个 genre（minCardinality 1 on belongsToGenre）
  - title 必须有且仅有1个（exactlyCardinality 1）
  - releaseDate 必须有且仅有1个
  - rating 的值必须在 0-10 之间（通过 SHACL 或应用层约束）
  - runtime 必须是正整数

Person 的约束：
  - name 必须有且仅有1个
  - birthDate 最多1个（maxCardinality 1）
  - nationality 可以有多个（有些人有双重国籍）

Award 的约束：
  - awardYear 必须有且仅有1个
  - awardCategory 至少有1个
```

第五步：OWL 代码示例

下面是用 OWL/Turtle 语法写出的部分本体定义。Turtle 是 RDF 的一种简洁可读的序列化格式：

```
@prefix : <http://example.org/movie-ontology#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# 本体声明
<http://example.org/movie-ontology> a owl:Ontology ;
    rdfs:comment "电影知识图谱本体 v1.0" ;
    rdfs:label "Movie Ontology" .

# 类定义
:Person a owl:Class ;
    rdfs:label "人物" ;
    rdfs:comment "与电影相关的所有人物" .

:Actor a owl:Class ;
    rdfs:subClassOf :Person ;
    rdfs:label "演员" .

:Director a owl:Class ;
    rdfs:subClassOf :Person ;
    rdfs:label "导演" .

:Movie a owl:Class ;
    rdfs:label "电影" .

:Genre a owl:Class ;
    rdfs:label "类型" .

:Award a owl:Class ;
    rdfs:label "奖项" .

# 对象属性
```

```
:actedIn a owl:ObjectProperty ;
    rdfs:label "出演" ;
    rdfs:domain :Actor ;
    rdfs:range :Movie ;
    owl:inverseOf :hasActor .

:directedBy a owl:ObjectProperty ;
    rdfs:label "导演" ;
    rdfs:domain :Movie ;
    rdfs:range :Director .

:belongsToGenre a owl:ObjectProperty ;
    rdfs:label "属于类型" ;
    rdfs:domain :Movie ;
    rdfs:range :Genre .

:wonAward a owl:ObjectProperty ;
    rdfs:label "获得奖项" ;
    rdfs:domain :Person ;
    rdfs:range :Award .

# 数据属性
:title a owl:DatatypeProperty ;
    rdfs:domain :Movie ;
    rdfs:range xsd:string .

:releaseDate a owl:DatatypeProperty ;
    rdfs:domain :Movie ;
    rdfs:range xsd:date .

:rating a owl:DatatypeProperty ;
    rdfs:domain :Movie ;
    rdfs:range xsd:float .

# 实例
```

```
:Inception a :Movie ;
    :title "Inception" ;
    :releaseDate "2010-07-16"^^xsd:date ;
    :rating 8.8 ;
    :directedBy :ChristopherNolan ;
    :belongsToGenre :SciFi, :Action, :Thriller .

:ChristopherNolan a :Director ;
    :name "Christopher Nolan" ;
    :birthDate "1970-07-30"^^xsd:date .
```

第六步：验证与迭代

创建实例后，打开 Protégé 的推理机（Reasoner），检查三个方面：

1. **一致性检查**：有没有逻辑矛盾？比如某个人被同时声明为”出生日期在2025年”和”导演了2010年的电影”
2. **分类检查**：推理出的类层级是否合理？比如如果一个实例既有 `actedIn` 关系又有 `directed` 关系，推理机会自动将它分类为既是 `Actor` 又是 `Director`
3. **能力问题测试**：用 SPARQL 查询来测试第一步列出的能力问题

```
# 测试 Q2: "诺兰导演的所有电影有哪些？"
PREFIX : <http://example.org/movie-ontology#>
SELECT ?movieTitle WHERE {
    ?movie a :Movie ;
           :directedBy :ChristopherNolan ;
           :title ?movieTitle .
}

# 测试 Q5: "和《肖申克的救赎》同类型的电影推荐？"
PREFIX : <http://example.org/movie-ontology#>
SELECT ?otherMovie ?title WHERE {
    :ShawshankRedemption :belongsToGenre ?genre .
    ?otherMovie :belongsToGenre ?genre ;
                :title ?title .
    FILTER (?otherMovie != :ShawshankRedemption)
}
ORDER BY ?title
```

常见陷阱与最佳实践

经过多年实践，本体设计领域积累了一些经典的教训。我把它们整理成”五大陷阱”和”七条最佳实践”，帮你少走弯路。

陷阱一：过度设计

症状： 想把世界上所有的概念都纳入本体，层级深达十层以上，属性定义了几百个。

问题： 本体太复杂，维护成本极高，团队成员看不懂，实际使用率很低。更糟糕的是，越复杂的本体越容易出现逻辑矛盾，推理机的性能也会急剧下降。

建议： 从核心概念开始，够用就好。记住 YAGNI 原则——You Aren't Gonna Need It（你不会需要它）。如果一个概念目前不会被用到，就不要加进去。你可以在后续迭代中逐步扩展。

陷阱二：类与实例混淆

症状： 把“《盗梦空间》”设计成了一个类，而不是 `Movie` 类的一个实例。

判断标准： 如果这个东西还会有”自己的例子”，那它就是一个类；如果它本身就是一个具体的、唯一的东西，那它就是一个实例。

```
✅ 正确：
  Movie（类） → Inception（实例）
  Genre（类） → SciFi（实例）
  Award（类） → Oscar_BestPicture（实例）

❌ 错误：
  Inception（类） → Scene1（实例） ← 除非你真的要把电影拆成场景
  SciFi（类） → HardSciFi（实例） ← 如果SciFi本身不会有多少子实例
```

一个实用的自检方法：问自己”这个东西可以有多个具体例子吗？”如果答案是”否”，那它应该是一个实例而不是类。

陷阱三：关系与属性混淆

症状： 把”导演姓名”作为一个从 Movie 到 String 的数据属性（`directorName`），而不是通过 `directedBy` 关系连接到 `Director` 实例。

判断标准： 如果这个信息背后还有更多信息，就应该用关系。导演不只是个名字——他还有出生日期、国籍、其他作品、获奖经历等。如果只用一个字符串来存储导演名字，这些信息就全丢了。

```
✅ 推荐：Movie --directedBy--> Director --name--> "Christop
   好处：Director 可以有自己的属性（birthDate、nationality），
         可以建立跨电影的关系（同一个导演的不同作品）

❌ 不推荐：Movie --directorName--> "Christopher Nolan"
   问题：无法查询"诺兰导演的所有电影"，因为导演只是一个字符串
```

陷阱四：忽视推理的代价

症状： 添加了大量复杂的公理和推理规则，结果查询变得极其缓慢，甚至推理机直接卡死。

建议： 在设计时就要考虑推理的复杂度。OWL 有不同的”风味”（Profile）：- **OWL Full**：最强大，但推理不可判定（可能永远算不完）- **OWL DL**：推理可判定，表达能力略有限制，适合大多数场景 - **OWL Lite**：最简单，推理最高效，适合对推理需求不高的场景

对于大多数实际应用，OWL DL 甚至更简单的 RDFS 就足够了。不要为了”炫技”而使用复杂的 OWL 特性。

陷阱五：命名不规范

症状： 类名有 `person`、`Person`、`PERSON`、`people` 多种写法混用，属性名有 `directed_by`、`directedBy`、`DirectedBy` 等多种风格。

最佳实践：统一命名规范并严格执行

表 10–3

元素	命名约定	示例
类	PascalCase（大驼峰）	`Person`，`FeatureFilm`
对象属性	camelCase（小驼峰）	`directedBy`，`actedIn`
数据属性	camelCase（小驼峰）	`releaseDate`，`personName`
实例	PascalCase 或带前缀	`ChristopherNolan` 或 `person:christopher_nolan`
URI	统一的前缀	`http://example.org/movie#`
注释	必须有（用 rdfs:comment）	每个类和属性都要有中文或英文说明

最佳实践清单

把下面这七条贴在工位上，每次设计本体时拿出来看看：

1. **从能力问题出发**：设计始终围绕”能回答什么问题”，不要为了设计而设计
2. **先复用，再创建**：优先使用已有的标准本体（Schema.org、DBpedia等）
3. **迭代开发**：不要追求一步到位，先做最小可行本体（MVO），再逐步扩展
4. **保持简洁**：够用就好，不要过度设计。复杂度是本体最大的敌人
5. **文档化**：为每个类和属性添加清晰的注释（`rdfs:comment`），后来的维护者会感谢你
6. **团队评审**：让领域专家和工程师一起评审，两种视角缺一不可
7. **用真实数据验证**：尽早导入真实数据测试，纸上谈兵永远发现不了真正的问题

动手练习：设计一个”菜谱本体”

练习目标

使用 Protégé 设计一个菜谱领域的本体，能够回答以下能力问题。这个练习将帮助你把本章学到的所有知识付诸实践。

```
Q1: "宫保鸡丁需要哪些食材？"
Q2: "有哪些川菜？"
Q3: "用豆腐能做什么菜？"
Q4: "有哪些适合素食者的低卡菜品？"
Q5: "红烧肉和东坡肉有什么共同食材？"
Q6: "哪些菜可以在30分钟内做完？"
Q7: "如果我没有花生，有什么替代食材？"
```

练习步骤

Step 1：分析能力问题，提取核心概念

从上面的问题中，你能识别出哪些概念？先不看下面的提示，自己想想看。

提示：菜品、食材、菜系、烹饪方式、口味……还有什么？卡路里算不算？烹饪时间算不算？替代食材关系怎么表示？

Step 2：设计类的层级

参考以下框架，但要根据你对能力问题的理解来调整：

```
owl:Thing
├── Dish（菜品）
│   ├── MainDish（主菜）
│   ├── Appetizer（前菜/凉菜）
│   ├── Soup（汤品）
│   ├── Dessert（甜品）
│   └── SideDish（配菜）
├── Ingredient（食材）
│   ├── Meat（肉类）
│   │   ├── Poultry（禽肉）
│   │   └── RedMeat（红肉）
│   ├── Vegetable（蔬菜）
│   ├── Seafood（海鲜）
│   ├── Grain（谷物）
│   ├── Spice（调味料/香料）
│   ├── Legume（豆类）
│   └── Dairy（乳制品）
├── Cuisine（菜系）
│   ├── SichuanCuisine（川菜）
│   ├── CantoneseCuisine（粤菜）
│   ├── ShandongCuisine（鲁菜）
│   └── ...
├── CookingMethod（烹饪方式）
│   ├── StirFry（炒）
│   ├── Steam（蒸）
│   ├── Braise（烧/炖）
│   ├── DeepFry（炸）
│   └── Boil（煮/焯）
└── DietaryTag（饮食标签）
    ├── Vegetarian（素食）
    ├── Vegan（纯素）
    ├── GlutenFree（无麸质）
    └── LowCalorie（低卡）
```

Step 3：定义关系

需要定义的对象属性（至少包括以下这些）：

- `hasIngredient`：Dish → Ingredient（菜品使用的食材）
- `belongsToCuisine`：Dish → Cuisine（菜品属于的菜系）
- `usesCookingMethod`：Dish → CookingMethod（菜品使用的烹饪方式）
- `hasDietaryTag`：Dish → DietaryTag（菜品的饮食标签）
- `canBeSubstitutedBy`：Ingredient → Ingredient（食材替代关系）

需要定义的数据属性：

- `calories`：Dish → xsd:integer（每份卡路里）
- `cookingTime`：Dish → xsd:integer（烹饪时间，分钟）
- `difficulty`：Dish → xsd:integer（难度 1–5）
- `servingSize`：Dish → xsd:integer（份数）
- `ingredientName`：Ingredient → xsd:string（食材名称）

Step 4：添加约束

- 每道菜至少有1种食材（`hasIngredient min 1`）
- 每道菜至少属于1个菜系（`belongsToCuisine min 1`）
- 卡路里值必须为正数
- 难度值必须在 1–5 之间
- 烹饪时间必须为正数

Step 5：创建实例

创建以下实例并关联（这是最有趣的一步，你会看到你的本体”活”起来）：

```
宫保鸡丁:
  - 类型: MainDish
  - 食材: 鸡肉、花生、干辣椒、花椒、葱、姜、蒜、酱油、醋
  - 菜系: 川菜
  - 烹饪方式: 爆炒
  - 卡路里: 350
  - 烹饪时间: 25分钟
  - 难度: 3

麻婆豆腐:
  - 类型: MainDish
  - 食材: 豆腐、牛肉末、豆瓣酱、花椒、葱、姜、蒜
  - 菜系: 川菜
  - 烹饪方式: 烧
  - 卡路里: 280
  - 烹饪时间: 20分钟
  - 难度: 2

白灼菜心:
  - 类型: SideDish
  - 食材: 菜心、蒜、生抽
  - 菜系: 粤菜
  - 烹饪方式: 焯
  - 卡路里: 80
  - 烹饪时间: 10分钟
  - 难度: 1
  - 饮食标签: Vegetarian, Vegan, LowCalorie

红烧肉:
  - 类型: MainDish
  - 食材: 五花肉、酱油、糖、料酒、姜、葱
  - 菜系: 鲁菜
  - 烹饪方式: 红烧
  - 卡路里: 520
```

```
  - 烹饪时间：90分钟
  - 难度：4

食材替代关系：
  - 鸡肉 canBeSubstitutedBy 豆腐（素食替代）
  - 花生 canBeSubstitutedBy 腰果（坚果替代）
  - 牛肉末 canBeSubstitutedBy 素肉末（素食替代）
```

Step 6：用推理机验证

1. 开启 HermiT 推理机（Reasoner → HermiT → Start reasoner）
2. 检查是否有一致性错误（红色的提示框）
3. 尝试用 DL Query 查询：
4. “所有川菜” → 应该返回宫保鸡丁和麻婆豆腐
5. “所有素食菜品” → 应该返回白灼菜心
6. “含豆腐的菜品” → 应该返回麻婆豆腐
7. “低卡（<200卡路里）的菜品” → 应该返回白灼菜心

进阶挑战

完成基本练习后，试试这些进阶扩展来加深理解：

1. **添加”食材替代”推理**：鸡胸肉可以被豆腐替代。设计一个推理规则，当用户要求素食版宫保鸡丁时，自动推荐替代食材方案
2. **添加”营养信息”类**：每道菜关联一个营养信息实例（蛋白质、脂肪、碳水化合物含量），让查询”高蛋白低脂菜品”成为可能
3. **添加”套餐”概念**：设计一个 `MealSet` 类，包含前菜、主菜、汤品、甜品的组合，并定义套餐的营养总量约束

4. **添加多语言支持**：为每个类和属性添加中英文标签（`rdfs:label`），练习本体的国际化

延伸阅读

1. **《Ontology Development 101: A Guide to Creating Your First Ontology》** —— Noy & McGuinness 的经典论文，本体设计七步法的原始出处。虽然写于2001年，但核心思想至今不过时。可从斯坦福大学网站免费下载
2. **《OWL 2 Web Ontology Language Primer》** —— W3C 官方 OWL 2 入门指南，最权威的 OWL 语法参考。https://www.w3.org/TR/owl2-primer/
3. **Protégé 官方教程** —— https://protege.stanford.edu/publications/ ，有大量视频教程和案例，从入门到高级推理
4. **Schema.org** —— https://schema.org/ ，Google 等公司维护的通用本体，非常值得学习其设计模式。特别关注其“Movie”、”Recipe” 等类型的设计
5. **《Semantic Web for the Working Ontologist》** —— Allemang & Hendler 著，语义网和本体工程的最佳实践书籍，语言生动，案例丰富
6. **Neo4j Graph Data Modeling** —— https://neo4j.com/docs/ ，图数据库建模的官方指南，特别是关于”何时用关系 vs 何时用属性”的讨论非常实用

7. 《Ontology Engineering》 —— Kendall & McGuinness 著，2017年出版，更全面地覆盖了本体工程的方方面面

本章小结

本章我们学习了本体设计——知识图谱建设中最关键、也最需要思考的一步。来回顾一下核心要点：

核心概念： – 本体是对领域中概念及其关系的形式化描述，是知识图谱的”建筑蓝图” – 三大要素：类（概念的层级）、关系（属性/关联）、约束（公理/规则） – 类有层级结构（子类继承父类），关系有方向性（定义域→值域），约束有类型（基数、值域、互斥等）

设计方法： – 自上而下：从抽象概念出发，适合结构清晰、有领域专家的领域 – 自下而上：从具体数据出发，适合快速迭代、探索性强的项目 – 混合方法：实践中最常用，先建骨架再填细节

实操要点： – Protégé 用于概念设计和逻辑推理（”想清楚”） – Neo4j Data Modeler 用于物理建模和查询优化（”做出来”） – 先列能力问题，再设计本体，最后用真实数据验证——这三步缺一不可

避坑指南： – 不要过度设计，够用就好（YAGNI原则） – 区分清楚类和实例（用”有没有自己的例子”来判断） – 优先用关系而非扁平属性（信息背后还有信息时，用关系） – 命名规范要统一并严格执行

关键心智模型： 本体设计不是写代码，而是**思考**。它是把一个领域的知识结构”想清楚”并”说清楚”的过程。好的本体设计能让团队达成共识，让推理成为可能，让知识可以复用。

下一章，我们将把这些精心设计的本体落地，完整走一遍知识图谱的构建流程——从数据采集到信息抽取到知识融合到知识存储，把原始数据变成真正的”知识”。

准备好了吗？我们继续出发。

第七章 知识图谱构建全流程

> *
>
> —— 老子《道德经》*

> *“千里之行，始于足下。”*

引子：一场”厨房大冒险”

还记得上一章的小李吗？那个想菜谱推荐系统想到头秃的程序员。

经过上一章的学习，小李终于把菜谱本体设计好了——菜品、食材、菜系、烹饪方式，类层级清晰，关系明确，约束合理。他信心满满地把本体给老板看，老板一拍桌子：”太好了！下周一上线！”

小李愣住了：”等等……我们只有本体（schema），没有数据啊。菜谱数据从哪来？网上有几百万道菜谱，总不能人工一条条录入吧？就算能爬下来，网页上写的都是自然语言，怎么把’取豆腐一块，切成小丁，下锅煸炒’这种话变成结构化的三元组？而且不同网站上同一道菜的做法和配料都不一样，怎么合并？”

老板想了想：”那……给你两周？”

小李叹了口气，开始研究怎么从非结构化文本中自动抽取知识。他发现这个领域叫做”信息抽取”（Information Extraction），是一门有着几十年历史的学科。更让他兴奋的是，近年来大语言模

型（LLM）的出现，让这个领域发生了革命性的变化——以前需要花几个月标注数据、训练模型的工作，现在用几句 Prompt 就能完成个大概。

小李的故事，其实是每一个知识图谱项目都会经历的”灵魂拷问”：**设计好了骨架，怎么往里面填血肉？**

这就是本章要解决的问题。我们将完整走一遍知识图谱构建的全流程，从数据采集到最终入库，手把手教你把”空气”变成”知识”。不管你是像小李一样在做菜谱推荐，还是在构建医疗知识图谱、金融风控图谱，本章的方法论都是通用的。

知识图谱构建的四步流水线

一个完整的知识图谱构建流程，通常包含四个核心环节。这四个环节环环相扣，每一步都建立在前一步的基础之上：

```
数据采集 → 信息抽取 → 知识融合 → 知识存储
(Data        (Information   (Knowledge     (Knowledge
Collection)   Extraction)    Fusion)        Storage)
```

让我用一个生活中的类比来解释这四个步骤：

想象你在编纂一本《中国美食大全》：

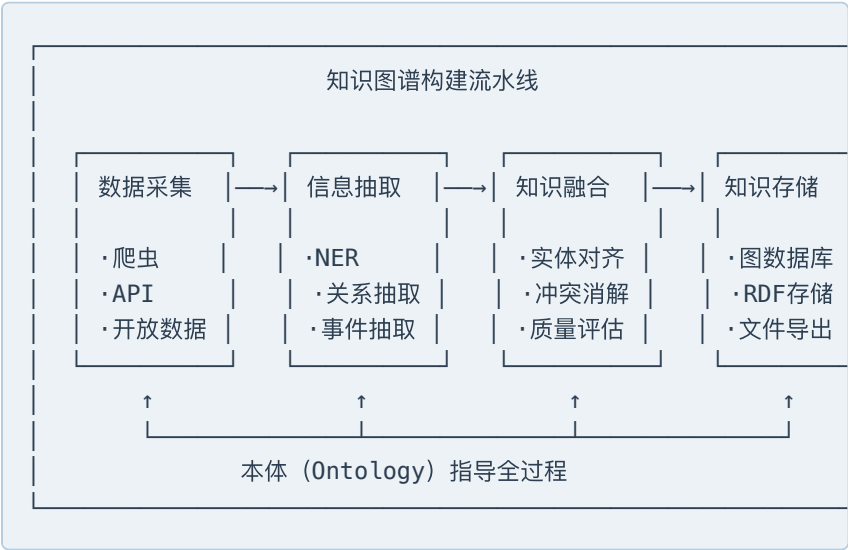
1. **数据采集**：你去菜市场、超市、网上收集各种菜谱、食材信息（收集原材料）

2. **信息抽取**：你阅读每份菜谱，提取出"菜名、食材、调料、烹饪方式"等结构化信息（从原材料中提取精华）
3. **知识融合**：你发现同一道菜在不同地方叫不同名字，不同菜谱的配料用量有冲突，需要统一和消解（整合不同来源的信息）
4. **知识存储**：你把整理好的知识按照章节、分类、索引组织好，存入书中（或数据库中）

每个环节都至关重要。数据采集不好，后面都是无米之炊；信息抽取不准，图谱里全是错误数据；知识融合不做，同一道菜出现三个不同的名字；知识存储不合理，查个数据要等半天。

全景视图

在深入每个环节之前，先看一张全景图，建立整体认知：



注意最下面那行——**本体贯穿整个流程**。它就像建筑蓝图，指导每一步施工。数据采集时，本体告诉你应该收集哪些类型的数据；信息抽取时，本体告诉你应该抽取哪些类型的实体和关系；知识融合时，本体提供实体对齐的判断标准；知识存储时，本体定义了图的结构。

第一步：数据采集

数据来源的三大渠道

知识图谱的数据不会从天而降，你需要主动去收集和获取。根据数据的来源和获取方式，主要有三种渠道，每种都有自己的优缺点。

1. 开放数据集（最省事，推荐优先考虑）

互联网上有大量高质量的开放知识图谱数据集，可以直接使用或作为你知识图谱的”骨架”。这就像盖房子时直接买现成的预制件，而不是自己烧砖。

表 11–1

数据集	领域	格式	规模	适用场景
DBpedia	通用	RDF/Turtle	数十亿三元组	通用知识图谱的基础
Wikidata	通用	JSON/RDF	1亿+条目	多语言、结构化程度高
CN-DBpedia	中文通用	RDF	千万级三元组	中文知识图谱的首选
OpenKG	中文多领域	多种格式	多个子数据集	中文社区贡献的各领域数据
MovieLens	电影	CSV	10万+评分	电影推荐场景
IMDB	电影	TSV	数百万条目	电影知识图谱的首选
Freebase	通用（已停更）	RDF	数十亿三元组	仍有参考价值

动手尝试：下载 IMDB 数据集

IMDB 提供了免费的公开数据集，包含电影、电视剧、演员、导演等完整信息，是构建电影知识图谱的绝佳数据源：

```
import requests
import pandas as pd
import gzip

# IMDB 数据集的下载地址
datasets = {
    "movies": "https://datasets.imdbws.com/title.basics.ts
    "crew": "https://datasets.imdbws.com/title.crew.tsv.gz
    "ratings": "https://datasets.imdbws.com/title.ratings.
    "actors": "https://datasets.imdbws.com/name.basics.tsv
}

def download_and_load(url, filename):
    """下载并加载 IMDB 数据集"""
    print(f"正在下载 {filename}...")
    response = requests.get(url)

    with open(filename, "wb") as f:
        f.write(response.content)

    with gzip.open(filename, "rt", encoding="utf-8") as f:
        df = pd.read_csv(f, sep="\t", na_values=["\\N"])

    print(f"  共 {len(df)} 条记录")
    print(f"  列名: {df.columns.tolist()}")
    return df

# 下载电影基本信息
movies_df = download_and_load(
    datasets["movies"], "title.basics.tsv.gz"
)
print(movies_df.head(3))

# 下载演职员信息
```

```
crew_df = download_and_load(
    datasets["crew"], "title.crew.tsv.gz"
)

# 数据探索：看看有哪些电影类型
print("\n电影类型分布：")
genres = movies_df["genres"].dropna().str.split(",").explo
print(genres.value_counts().head(10))
```

2. API 接口（最规范，适合结构化数据）

很多网站提供结构化的 API 接口，返回的数据已经是 JSON 格式，可以直接使用。这种方式的数据质量通常很高，而且可以增量更新。

```
# 使用 TMDB API 获取电影信息
import requests
import json

API_KEY = "your_tmdb_api_key"  # 在 themoviedb.org 免费注册

def get_movie_details(movie_id):
    """获取电影详情，包括演职员和关键词"""
    url = f"https://api.themoviedb.org/3/movie/{movie_id}"
    params = {
        "api_key": API_KEY,
        "language": "zh-CN",  # 获取中文信息
        "append_to_response": "credits,keywords,release_da
    }
    response = requests.get(url, params=params)
    return response.json()

def search_movie(query):
    """搜索电影"""
    url = "https://api.themoviedb.org/3/search/movie"
    params = {"api_key": API_KEY, "query": query, "languag
    response = requests.get(url, params=params)
    return response.json()["results"]

# 获取《盗梦空间》的详细信息
movie = get_movie_details(27205)
print(f"电影名: {movie['title']}")
print(f"原名: {movie['original_title']}")
print(f"上映日期: {movie['release_date']}")
print(f"评分: {movie['vote_average']}")

# 提取导演
directors = [c["name"] for c in movie["credits"]["crew"]
             if c["job"] == "Director"]
```

```
print(f"导演: {directors}")

# 提取主演（前5位）
top_cast = [c["name"] for c in movie["credits"]["cast"][:5
print(f"主演: {top_cast}")

# 提取类型
genres = [g["name"] for g in movie["genres"]]
print(f"类型: {genres}")

# 将API数据转化为三元组
def movie_to_triples(movie_data):
    """将 API 返回的数据转化为三元组"""
    triples = []
    title = movie_data["title"]

    # 导演关系
    for crew in movie_data["credits"]["crew"]:
        if crew["job"] == "Director":
            triples.append((title, "directedBy", crew["nam

    # 演员关系
    for cast in movie_data["credits"]["cast"]:
        triples.append((title, "actedIn_by", cast["name"])

    # 类型关系
    for genre in movie_data["genres"]:
        triples.append((title, "hasGenre", genre["name"]))

    # 数据属性
    triples.append((title, "releaseDate", movie_data["rele
    triples.append((title, "rating", str(movie_data["vote_

    return triples
```

```
triples = movie_to_triples(movie)
for t in triples[:10]:
    print(f"  {t}")
```

3. 网页爬虫（最灵活，也最辛苦）

当数据没有现成的数据集或 API 时，爬虫是最后的手段。爬虫可以获取任何公开网页上的信息，但需要处理大量的非结构化数据（HTML、自然语言文本），并且要应对网站的反爬机制。

```
# 使用 BeautifulSoup 爬取电影信息（教学示例）
import requests
from bs4 import BeautifulSoup
import time
import json

def scrape_movie_page(url):
    """爬取电影页面信息（示例，实际使用需适配具体网站结构）"""
    headers = {
        "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac O
                      "AppleWebKit/537.36 (KHTML, like Gec
                      "Chrome/120.0.0.0 Safari/537.36",
        "Accept-Language": "zh-CN,zh;q=0.9"
    }

    try:
        response = requests.get(url, headers=headers, time
        response.raise_for_status()
    except requests.RequestException as e:
        print(f"请求失败: {e}")
        return None

    soup = BeautifulSoup(response.text, "html.parser")

    # 这里的选择器需要根据具体网站结构调整
    # 以下是通用模式
    result = {}

    # 提取标题（通常在 h1 或特定的标题元素中）
    title_elem = soup.find("h1")
    if title_elem:
        result["title"] = title_elem.get_text(strip=True)

    # 提取结构化数据（很多网站有 JSON-LD）
```

```
    for script in soup.find_all("script", type="applicatio
        try:
            json_data = json.loads(script.string)
            if isinstance(json_data, dict):
                result.update(extract_from_jsonld(json_dat
        except json.JSONDecodeError:
            continue

    return result

def extract_from_jsonld(data):
    """从 JSON-LD 结构化数据中提取信息"""
    result = {}
    if data.get("@type") == "Movie":
        result["title"] = data.get("name", "")
        result["director"] = data.get("director", {}).get(
        result["genre"] = data.get("genre", [])
        result["datePublished"] = data.get("datePublished"
    return result

# 爬虫注意事项（非常重要！）
RULES = """
爬虫最佳实践：
1. 遵守 robots.txt 协议（用 urllib.robotparser 检查）
2. 控制请求频率（建议每秒不超过1次，礼貌性延时）
3. 使用 Session 复用 TCP 连接，提高效率
4. 做好异常处理和重试机制（网络不可靠）
5. 设置合理的超时时间（避免无限等待）
6. 注意数据版权和隐私合规（不要爬取个人信息）
7. 考虑使用 Scrapy 框架处理大规模爬取任务
"""
print(RULES)
```

数据预处理

不管数据从哪里来，拿到手后通常需要做一些预处理工作：

```
import pandas as pd
import re

def preprocess_movie_data(df):
    """电影数据预处理"""

    # 1. 去除重复数据
    df = df.drop_duplicates(subset=["tconst"])
    print(f"去重后: {len(df)} 条记录")

    # 2. 处理缺失值
    # 对于类型字段，缺失值填充为 "Unknown"
    df["genres"] = df["genres"].fillna("Unknown")

    # 3. 标准化字段
    # 将年份字符串转为整数
    df["startYear"] = pd.to_numeric(df["startYear"], error
    df["runtimeMinutes"] = pd.to_numeric(df["runtimeMinute

    # 4. 展开多值字段（如类型字段 "Action,SciFi,Thriller"）
    df["genre_list"] = df["genres"].str.split(",")

    # 5. 过滤不需要的类型（如成人内容）
    if "isAdult" in df.columns:
        df = df[df["isAdult"] == 0]

    # 6. 过滤过短的内容（如短片、视频）
    if "titleType" in df.columns:
        df = df[df["titleType"].isin(["movie", "tvMovie"])

    print(f"预处理后: {len(df)} 条记录")
    return df

# 数据质量检查
```

```
def check_data_quality(df):
    """检查数据质量"""
    print("=== 数据质量报告 ===")
    print(f"总记录数: {len(df)}")
    print(f"\n缺失值统计:")
    for col in df.columns:
        missing = df[col].isna().sum()
        if missing > 0:
            print(f"  {col}: {missing} ({missing/len(df)*1

    print(f"\n数值字段统计:")
    for col in df.select_dtypes(include="number").columns:
        print(f"  {col}: min={df[col].min()}, max={df[col]
              f"mean={df[col].mean():.1f}")
```

第二步：信息抽取

有了原始数据后，下一步是最关键也最有技术含量的一步——**从非结构化或半结构化数据中抽取结构化的知识**。

所谓"结构化的知识"，主要是**三元组**（Triple），即 `(主语, 谓语, 宾语)` 的形式。比如从句子"克里斯托弗·诺兰执导了电影《盗梦空间》"中，我们要抽取：

```
(盗梦空间, directedBy, 克里斯托弗·诺兰)
```

信息抽取包含三个核心子任务：

- **命名实体识别（NER）**：找出文本中的实体（人名、地名、组织名等）

- **关系抽取（RE）**：找出实体之间的关系（谁导演了谁、谁出演了谁）
- **事件抽取（EE）**：找出事件及其参与者（某年某月在某地举办了某活动）

这三个子任务层层递进，从”有什么”到”什么关系”再到”发生了什么”，逐步构建出完整的知识图谱。

命名实体识别（NER）

NER 的任务是从文本中识别出具有特定类型的实体。这是信息抽取的第一步，也是最基础的一步。

举个例子：

```
输入：克里斯托弗·诺兰于1970年出生在伦敦，是英国著名的电影导演。

输出：
  [克里斯托弗·诺兰] → Person（人物）
  [1970年] → Date（日期）
  [伦敦] → Location（地点）
  [英国] → GPE（地缘政治实体）
  [电影导演] → Title（头衔/职业）
```

NER 的技术经历了四代发展，每一代都有显著的进步：

方法一：基于规则（第一代）

最简单的方法是人写正则表达式和词典来匹配实体。虽然看起来”原始”，但在很多场景下依然有效，而且不需要任何训练数据。

```
import re

def rule_based_ner(text):
    """基于规则的简单NER——适合结构化程度高的场景"""
    entities = []

    # 规则1: 匹配中文日期
    date_patterns = [
        (r"\d{4}年\d{1,2}月\d{1,2}日", "DATE"),
        (r"\d{4}年\d{1,2}月", "DATE"),
        (r"\d{4}年", "YEAR"),
    ]
    for pattern, etype in date_patterns:
        for match in re.finditer(pattern, text):
            entities.append({
                "text": match.group(),
                "type": etype,
                "start": match.start(),
                "end": match.end()
            })

    # 规则2: 匹配电影名（《》括号中的内容）
    movie_pattern = r"《([^》]+)》"
    for match in re.finditer(movie_pattern, text):
        entities.append({
            "text": match.group(1),  # 去掉书名号
            "type": "MOVIE",
            "start": match.start(),
            "end": match.end()
        })

    # 规则3: 基于姓氏词典的人名匹配
    surnames = "赵钱孙李周吴郑王冯陈褚卫蒋沈韩杨朱秦尤许何吕施张孔
    name_pattern = rf"([{surnames}][\u4e00-\u9fa5]{{1,3}})
```

```
    stopwords = {"什么", "怎么", "这个", "那个", "我们", "你们
                 "不是", "可以", "这个", "一个", "如果", "因为
    for match in re.finditer(name_pattern, text):
        if match.group() not in stopwords:
            entities.append({
                "text": match.group(),
                "type": "PERSON",
                "start": match.start(),
                "end": match.end()
            })

    # 规则4: 匹配金额
    money_pattern = r"\d+(?:\.\d+)?(?:亿|万|元|美元|人民币)"
    for match in re.finditer(money_pattern, text):
        entities.append({
            "text": match.group(),
            "type": "MONEY",
            "start": match.start(),
            "end": match.end()
        })

    return entities

# 测试
text = "张艺谋于1950年4月2日出生在陕西省西安市，执导了《英雄》《满城
entities = rule_based_ner(text)
for entity in entities:
    print(f"  [{entity['text']}] → {entity['type']}")

# 输出:
#   [1950年4月2日] → DATE
#   [《英雄》] → MOVIE (注意: 提取的是"英雄")
#   [张艺谋] → PERSON
#   [30亿元] → MONEY
```

优点： 简单直接，不需要训练数据，准确率高（在规则覆盖的范围内），可解释性强 **缺点：** 泛化能力差，无法处理规则未覆盖的情况，需要大量人工维护规则

方法二：基于统计模型（第二代）

经典的统计方法包括 CRF（条件随机场）、HMM（隐马尔可夫模型）等。它们的核心思想是：把 NER 看作一个序列标注问题——对于句子中的每个字（或词），标注它是某个实体的开头（B）、中间（I）还是不是实体（O）。

```
# 使用 sklearn-crfsuite 实现 CRF-based NER
# pip install sklearn-crfsuite

import sklearn_crfsuite
from sklearn_crfsuite import metrics

def char_features(sentence, i):
    """提取第i个字符的特征"""
    char = sentence[i]
    features = {
        "char": char,
        "is_digit": char.isdigit(),
        "is_alpha": char.isalpha(),
        "is_chinese": "\u4e00" <= char <= "\u9fff",
        "is_punct": char in "，。！？、；：""''（）《》【】",
    }

    # 上下文特征（前后各2个字）
    for offset in [-2, -1, 1, 2]:
        idx = i + offset
        if 0 <= idx < len(sentence):
            features[f"char_{offset}"] = sentence[idx]
        else:
            features[f"boundary_{offset}"] = True

    return features

def prepare_features(text):
    """准备特征序列"""
    return [char_features(text, i) for i in range(len(text

# 训练数据：(文本, 标注序列)
# 标注格式：B-PER(人名开头), I-PER(人名中间), B-LOC(地名开头), (
train_data = [
```

```
    ("张艺谋出生于陕西",
     ["B-PER", "I-PER", "I-PER", "O", "O", "O", "B-LOC", '
    ("诺兰导演了盗梦空间",
     ["B-PER", "I-PER", "O", "O", "O", "B-MOV", "I-MOV", '
]

X_train = [prepare_features(text) for text, _ in train_dat
y_train = [labels for _, labels in train_data]

# 训练 CRF 模型
crf = sklearn_crfsuite.CRF(
    algorithm="lbfgs",
    c1=0.1,
    c2=0.1,
    max_iterations=100,
    all_possible_transitions=True
)
crf.fit(X_train, y_train)

# 预测
test_text = "周星驰主演了功夫"
test_features = prepare_features(test_text)
predictions = crf.predict([test_features])[0]

for char, label in zip(test_text, predictions):
    if label != "O":
        print(f"  {char} → {label}")
```

方法三：深度学习（第三代）

目前工业界主流的深度学习 NER 方案是 BERT + CRF 架构。BERT 负责理解上下文语义，CRF 负责保证标注序列的合法性（比如 I-PER 不能出现在 B-PER 之前）。

```
# 使用 HuggingFace Transformers 进行 NER
# pip install transformers torch

from transformers import pipeline

# 加载预训练的中文 NER 模型
# 这里使用 uer 训练的中文 NER 模型
ner_pipeline = pipeline(
    "ner",
    model="uer/roberta-base-finetuned-cluener2020-chinese"
    aggregation_strategy="simple"  # 将子token聚合为完整实体
)

text = "《流浪地球2》由吴京主演，郭帆导演，于2023年1月22日在中国上

results = ner_pipeline(text)
for entity in results:
    print(f"  [{entity['word']}] → {entity['entity_group']}
          f"(置信度: {entity['score']:.2f})")

# 预期输出:
#   [流浪地球2] → Work/Movie (置信度: 0.95)
#   [吴京] → Person (置信度: 0.98)
#   [郭帆] → Person (置信度: 0.97)
#   [2023年1月22日] → Date (置信度: 0.93)
#   [中国] → Location (置信度: 0.96)
```

深度学习的优势： 泛化能力强，可以处理训练数据中未见过的模式；端到端训练，不需要人工设计特征 **深度学习的劣势：** 需要大量标注数据（通常需要数千到数万条标注样本）；模型是”黑盒”，不太可解释

方法四：LLM 辅助（第四代，最前沿）

大语言模型（LLM）的出现，让 NER 进入了一个全新的时代。你可以直接用自然语言指令来完成 NER，不需要任何标注数据：

```
# pip install openai
import openai
import json

def llm_ner(text):
    """使用 LLM 进行命名实体识别"""
    prompt = f"""请从以下文本中提取命名实体，返回 JSON 格式。

实体类型包括：
- Person（人物）
- Movie（电影）
- Organization（组织/公司）
- Location（地点）
- Date（日期）
- Award（奖项）
- Genre（类型/风格）

文本：{text}

返回格式（严格遵循）：
{{
  "entities": [
    {{"text": "实体文本", "type": "实体类型", "start": 起始位
  ]
}}

注意：
1. start 和 end 是实体在原文中的字符位置（从0开始）
2. 不要遗漏任何实体
3. 如果实体类型不在候选列表中，返回 "Other"
"""

    response = openai.chat.completions.create(
        model="gpt-4",
```

```
        messages=[{"role": "user", "content": prompt}],
        temperature=0,  # 低温度保证结果稳定
        response_format={"type": "json_object"}
    )

    result = json.loads(response.choices[0].message.conter
    return result

# 测试
text = "在第95届奥斯卡颁奖典礼上，杨紫琼凭借《瞬息全宇宙》获得最佳女

result = llm_ner(text)
for entity in result["entities"]:
    print(f"  [{entity['text']}] → {entity['type']}")

# 预期输出：
#   [第95届奥斯卡颁奖典礼] → Award
#   [杨紫琼] → Person
#   [瞬息全宇宙] → Movie
#   [最佳女主角奖] → Award
#   [关家永] → Person
```

LLM 做 NER 的优势： – 不需要标注数据，零样本（zero-shot）即可使用 – 可以灵活定义实体类型（只需修改 Prompt） – 对复杂语境的理解能力强（能理解隐喻、指代等） – 支持多语言（同一个 Prompt 可以处理中英文） – 可以同时返回置信度和解释

LLM 做 NER 的劣势： – 速度较慢（秒级响应），不适合大规模实时处理 – 成本较高（API 调用费用） – 输出格式可能不稳定（偶尔返回非法 JSON） – 可能产生幻觉（hallucination）——识别出不存在的实体

最佳实践：LLM 标注 + 小模型部署

这是目前工业界最流行的方案，结合了两者的优势：

```
def llm_assisted_annotation_pipeline(texts, annotation_buc
    """
    LLM 辅助标注流水线

    核心思路：
    1. 用 LLM 标注少量数据（快，但不便宜）
    2. 人工审核和修正标注（确保质量）
    3. 用标注数据训练专用小模型（快且便宜）
    4. 用训练好的模型处理大规模数据

    这样既享受了 LLM 的"零标注"优势，
    又保证了大规模处理时的速度和成本控制。
    """

    labeled_data = []
    failed_count = 0

    # Step 1: 用 LLM 批量标注
    for i, text in enumerate(texts[:annotation_budget]):
        try:
            result = llm_ner(text)
            entities = result["entities"]

            # 转换为 spaCy 训练格式
            spans = []
            for ent in entities:
                spans.append((ent["start"], ent["end"], er

            labeled_data.append((text, {"entities": spans}

            if (i + 1) % 50 == 0:
                print(f"  已标注 {i + 1}/{annotation_budget

        except (json.JSONDecodeError, KeyError) as e:
```

```
            failed_count += 1
            continue

    print(f"\n标注完成：")
    print(f"  成功：{len(labeled_data)} 条")
    print(f"  失败：{failed_count} 条")
    print(f"  成功率：{len(labeled_data)/annotation_budget*
    print(f"\n下一步：人工审核标注质量，然后用 spaCy/Transforme

    return labeled_data
```

关系抽取（Relation Extraction）

识别出实体之后，下一步是找出实体之间的关系。这是知识图谱构建中最具挑战性的任务之一，因为关系的表达在自然语言中非常多样化。

同样的关系”执导”，在不同的文本中可能有完全不同的表达方式：

```
- "《盗梦空间》由克里斯托弗·诺兰执导"
- "诺兰导演的新作《盗梦空间》"
- "克里斯托弗·诺兰是《盗梦空间》的导演"
- "诺兰执导的这部科幻巨作《盗梦空间》"
```

方法一：模板匹配

为每种关系预设一组触发词模板。这种方法简单直接，适合关系表达比较固定的场景：

```
import re

# 关系模板定义（越全面越好）
RELATION_PATTERNS = {
    "directedBy": [
        r"(.+?)(?:由|是)(.+?)(?:执导|导演|导演的)",
        r"(.+?)的导演是(.+?)(?:[,。]|$)",
        r"(.+?)(?:,|，)(.+?)(?:执导|导演|导演的)",
        r"(.+?)(?:是)(.+?)(?:执导的|导演的)",
    ],
    "actedIn": [
        r"(.+?)(?:主演|出演|参演)(?:了)?(.+?)(?:[,。]|$)",
        r"(.+?)(?:在|于)(.+?)(?:中饰演|中扮演)",
        r"(.+?)(?:由)(.+?)(?:主演|出演)",
    ],
    "bornIn": [
        r"(.+?)(?:出生于|生于|出生在)(.+?)(?:[,。]|$)",
    ],
    "belongsToGenre": [
        r"(.+?)(?:是一部|属于)(.+?)(?:片|电影|类型)(?:[,。]|$)"
    ],
    "wonAward": [
        r"(.+?)(?:获得|荣获|摘得|赢得)(?:了)?(.+?)(?:[,。]|$)"
        r"(.+?)(?:凭借|以)(.+?)(?:获得|荣获)",
    ]
}

def template_relation_extraction(text):
    """基于模板的关系抽取"""
    extracted_relations = []

    for relation_name, patterns in RELATION_PATTERNS.items
        for pattern in patterns:
            for match in re.finditer(pattern, text):
```

```
                subject = match.group(1).strip(",。 ")
                object_ = match.group(2).strip(",。 ")

                # 简单过滤：去除过短或过长的匹配
                if 1 < len(subject) < 20 and 1 < len(objec
                    extracted_relations.append({
                        "subject": subject,
                        "relation": relation_name,
                        "object": object_,
                        "method": "template"
                    })

    return extracted_relations

# 测试
texts = [
    "《盗梦空间》由克里斯托弗·诺兰执导，莱昂纳多·迪卡普里奥主演了该片
    "张艺谋出生于陕西省西安市，是中国第五代导演的代表人物。",
    "《流浪地球》是一部科幻片，由郭帆导演。",
]

for text in texts:
    print(f"\n文本：{text[:40]}...")
    relations = template_relation_extraction(text)
    for rel in relations:
        print(f"  ({rel['subject']}, {rel['relation']}, {r
```

方法二：远程监督（Distant Supervision）

远程监督是一种巧妙的思路：如果我们已经知道一些关系（来自已有的知识库），就可以用这些已知关系来自动标注训练数据。

核心假设：**如果两个实体在知识库中存在某种关系，那么同时包含这两个实体的句子很可能也在表达这种关系。**

```
def distant_supervision_demo():
    """
    远程监督示例

    假设我们已有的种子知识：
    - (盗梦空间, directedBy, 克里斯托弗·诺兰)
    - (星际穿越, directedBy, 克里斯托弗·诺兰)
    """

    # 已知关系（种子知识库）
    known_relations = {
        ("盗梦空间", "克里斯托弗·诺兰"): "directedBy",
        ("星际穿越", "克里斯托弗·诺兰"): "directedBy",
        ("肖申克的救赎", "弗兰克·德拉邦特"): "directedBy",
    }

    # 从语料库中收集包含实体对的句子
    sentences = [
        "克里斯托弗·诺兰在执导《盗梦空间》时，使用了大量的实景拍摄和
        "《盗梦空间》获得了多项大奖，票房成绩优异，全球票房超过8亿美
        "诺兰的新作品《星际穿越》再次展现了他独特的叙事风格和对时间机
        "在接受采访时，克里斯托弗·诺兰谈到了他的电影创作理念和灵感来
    ]

    # 自动标注
    labeled_data = []
    for sent in sentences:
        labels = set()
        for (subj, obj), rel in known_relations.items():
            # 检查主体和客体是否同时出现在句子中
            if subj in sent and obj in sent:
                labels.add(rel)
            # 也检查部分匹配（"诺兰" 匹配 "克里斯托弗·诺兰"）
            elif subj in sent:
```

```
                obj_parts = obj.split("·")
                if any(part in sent for part in obj_parts
                    labels.add(rel)

        if labels:
            labeled_data.append({
                "sentence": sent,
                "labels": list(labels),
                "note": "自动标注（可能有噪声）"
            })

    return labeled_data

result = distant_supervision_demo()
for item in result:
    print(f"  句子：{item['sentence'][:50]}...")
    print(f"  标签：{item['labels']} ({item['note']})")
    print()
```

远程监督的核心问题：噪声。 比如”在接受采访时，克里斯托弗·诺兰谈到了他的电影创作理念”——这句话同时包含导演名和电影名，但并没有表达 directedBy 关系。为了解决这个问题，研究者提出了”多实例学习”（Multi-Instance Learning）等方法来降低噪声的影响。

方法三：LLM 直接抽取（推荐方案）

LLM 的出现让关系抽取变得更加简单和准确。你可以直接把文本和候选实体对发给 LLM，让它判断关系：

```
import openai
import json

def llm_relation_extraction(text, entity_pairs=None):
    """使用 LLM 抽取实体对之间的关系"""

    if entity_pairs:
        pairs_text = "\n".join([
            f"  - ({s}, {o})" for s, o in entity_pairs
        ])
        pairs_instruction = f"请从以下实体对中判断关系：\n{pai
    else:
        pairs_instruction = "请自动发现文本中的实体关系对。"

    prompt = f"""你是一个知识图谱关系抽取专家。

给定文本，判断实体对之间的关系类型。如果不存在关系，返回 "no_relati

文本：{text}

{pairs_instruction}

关系类型候选：
- directedBy（A导演了B，主语为电影，客体为导演）
- actedIn（A出演了B，主语为演员，客体为电影）
- bornIn（A出生在B，主语为人物，客体为地点）
- belongsToGenre（A属于B类型）
- wonAward（A获得了B奖项）
- sequelOf（A是B的续集）
- citizenOf（A是B国的公民）
- releaseIn（A在B上映/发布）

请返回 JSON 格式：
{{
```

```
  "relations": [
    {{"subject": "实体1", "relation": "关系类型", "object":
  ]
}}"""

    response = openai.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}],
        temperature=0,
        response_format={"type": "json_object"}
    )

    return json.loads(response.choices[0].message.content)

# 测试
text = """2023年，郭帆导演的《流浪地球2》在中国上映，吴京再次担任主
该片是《流浪地球》的续集，全球票房超过5亿美元。"""

# 方式一：给定实体对
entity_pairs = [
    ("流浪地球2", "郭帆"),
    ("流浪地球2", "吴京"),
    ("流浪地球2", "流浪地球"),
    ("流浪地球2", "中国"),
]

result = llm_relation_extraction(text, entity_pairs)
print("抽取结果：")
for rel in result["relations"]:
    print(f"  ({rel['subject']}, {rel['relation']}, {rel['
          f"[置信度：{rel['confidence']}]")
```

事件抽取（Event Extraction）

事件抽取比关系抽取更复杂，它需要从文本中识别事件及其多个参与者。一个事件通常包含：事件类型、触发词（trigger）、参与者（arguments）、时间、地点等要素。

```
def llm_event_extraction(text):
    """使用 LLM 进行事件抽取"""
    prompt = f"""从以下文本中提取所有事件信息。

文本：{text}

对于每个事件，请提取：
1. event_type: 事件类型（如：movie_release/电影上映, award_ce
   person_birth/人物出生, company_founding/公司成立）
2. trigger: 触发这个事件的关键词或短语
3. participants: 参与事件的角色及其对应的实体列表
4. time: 事件发生的时间（如果文本中提到）
5. location: 事件发生的地点（如果文本中提到）

返回 JSON 格式：
{{
  "events": [
    {{
      "event_type": "事件类型",
      "trigger": "触发词",
      "participants": [{{"role": "角色名", "entity": "实体名
      "time": "时间",
      "location": "地点"
    }}
  ]
}}"""

    response = openai.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}],
        temperature=0,
        response_format={"type": "json_object"}
    )
```

```
    return json.loads(response.choices[0].message.content)

# 测试
text = """第95届奥斯卡颁奖典礼于2023年3月12日在洛杉矶好莱坞杜比剧
杨紫琼凭借《瞬息全宇宙》获得最佳女主角奖，
关继威获得最佳男配角奖。
杰米·李·柯蒂斯同样凭借《瞬息全宇宙》获得最佳女配角奖。"""

result = llm_event_extraction(text)
for event in result["events"]:
    print(f"\n  事件类型: {event['event_type']}")
    print(f"  触发词: {event['trigger']}")
    print(f"  时间: {event.get('time', '未提及')}")
    print(f"  地点: {event.get('location', '未提及')}")
    for p in event.get("participants", []):
        print(f"    {p['role']}: {p['entity']}")
```

第三步：知识融合

从不同数据源抽取出来的知识，往往存在冗余和冲突。同一个实体在不同来源中可能有不同的名字、不同的属性值。知识融合的目标是将这些碎片化的知识整合成一个一致的、高质量的知识图谱。

知识融合包含三个核心子任务：

1. **实体对齐（Entity Alignment）**：识别不同数据源中指向同一实体的不同描述
2. **冲突消解（Conflict Resolution）**：处理不同来源之间的矛盾信息

3. **质量评估（Quality Assessment）**：评估融合后知识图谱的质量

实体对齐（Entity Alignment）

这是知识融合中最关键的步骤。现实世界中，同一个实体在不同数据源中的表示往往千差万别：

```
克里斯托弗·诺兰 = Christopher Nolan = Christopher Edward Nol
《盗梦空间》 = Inception = 全面启动（台湾译名） = 潜行凶间（香港译
```

如果不做实体对齐，你的知识图谱里同一个人会出现三个不同的节点，各带各的属性，彼此不关联——这就是所谓的”知识孤岛”问题。

```
from difflib import SequenceMatcher
import unicodedata

class EntityAligner:
    """实体对齐器：识别和合并指向同一实体的不同描述"""

    def __init__(self, similarity_threshold=0.85):
        self.threshold = similarity_threshold
        self.canonical_entities = {}  # canonical_name ->
        self.alias_map = {}  # alias -> canonical_name

    def normalize(self, name):
        """实体名称标准化处理"""
        if not name:
            return ""
        # 统一为小写
        name = name.strip().lower()
        # 去除 Unicode 标点变体
        name = unicodedata.normalize("NFKC", name)
        # 去除常见分隔符
        name = name.replace("·", "").replace("•", "").repl
        # 去除括号及内容（如 "盗梦空间（电影）" → "盗梦空间"）
        import re
        name = re.sub(r"[([].+?[)]]", "", name).strip()
        return name

    def similarity(self, name1, name2):
        """计算两个名称的相似度（综合多种策略）"""
        n1 = self.normalize(name1)
        n2 = self.normalize(name2)

        # 策略1：精确匹配
        if n1 == n2:
            return 1.0
```

```
        # 策略2：包含关系
        if n1 in n2 or n2 in n1:
            return 0.95

        # 策略3：编辑距离相似度
        edit_sim = SequenceMatcher(None, n1, n2).ratio()

        # 策略4：Jaccard 相似度（基于字符集合）
        set1 = set(n1)
        set2 = set(n2)
        if set1 or set2:
            jaccard = len(set1 & set2) / len(set1 | set2)
        else:
            jaccard = 0

        # 综合得分
        return max(edit_sim, jaccard)

    def find_match(self, entity_name, entity_type):
        """在已知实体中查找最佳匹配"""
        # 先查别名表
        normalized = self.normalize(entity_name)
        if normalized in self.alias_map:
            return self.alias_map[normalized], 1.0

        # 遍历已知实体，计算相似度
        best_match = None
        best_score = 0

        for canonical, info in self.canonical_entities.ite
            # 只比较同类型的实体
            if info.get("type") != entity_type:
                continue
```

```
            score = self.similarity(entity_name, canonical

            # 也检查别名
            for alias in info.get("aliases", []):
                alias_score = self.similarity(entity_name,
                score = max(score, alias_score)

            if score > best_score:
                best_score = score
                best_match = canonical

        if best_score >= self.threshold:
            return best_match, best_score
        return None, best_score

    def add_or_merge(self, name, entity_type, properties):
        """添加新实体或合并到已有实体"""
        existing, score = self.find_match(name, entity_typ

        if existing:
            # 合并：更新属性，添加别名
            info = self.canonical_entities[existing]
            # 属性合并策略：新属性不覆盖已有属性（除非已有为空）
            for k, v in properties.items():
                if k not in info.get("properties", {}) or
                    info.setdefault("properties", {})[k] =
            info.setdefault("aliases", set()).add(name)
            self.alias_map[self.normalize(name)] = existin
            return existing, "merged", score
        else:
            # 新建实体
            self.canonical_entities[name] = {
                "type": entity_type,
                "properties": properties,
                "aliases": {name}
```

```
            }
            self.alias_map[self.normalize(name)] = name
            return name, "created", 1.0

    def report(self):
        """生成对齐报告"""
        print("=== 实体对齐报告 ===")
        print(f"标准实体数: {len(self.canonical_entities)}"
        total_aliases = sum(
            len(info.get("aliases", []))
            for info in self.canonical_entities.values()
        )
        print(f"总别名数: {total_aliases}")
        print(f"平均别名数: {total_aliases / max(len(self.ca

        # 列出有多个别名的实体
        print("\n多别名实体:")
        for name, info in self.canonical_entities.items():
            aliases = info.get("aliases", set())
            if len(aliases) > 1:
                print(f"  {name}: {aliases}")

# 使用示例
aligner = EntityAligner(similarity_threshold=0.80)

# 模拟从不同数据源导入数据
import_sources = [
    # 数据源1：IMDB（英文）
    {"name": "Christopher Nolan", "type": "Person",
     "properties": {"birth_year": 1970, "nationality": "Br
    # 数据源2：豆瓣（中文）
    {"name": "克里斯托弗·诺兰", "type": "Person",
     "properties": {"birth_date": "1970-07-30", "popular_m
    # 数据源3：维基百科（全名）
    {"name": "Christopher Edward Nolan", "type": "Person",
```

```
     "properties": {"birth_place": "London", "spouse": "Em
    # 数据源4：新闻报道（简称）
    {"name": "诺兰", "type": "Person",
     "properties": {"latest_movie": "奥本海默"}},
]

print("实体对齐过程：")
for source in import_sources:
    result_id, action, score = aligner.add_or_merge(
        source["name"], source["type"], source["properties
    )
    print(f"  [{source['name']}] → {action}（标准名：{resul

aligner.report()
```

知识冲突消解

当不同数据源对同一实体给出矛盾的信息时，需要进行冲突消解。比如维基百科说诺兰出生于7月30日，但某个电影网站写的是7月31日——谁是对的？

```
class ConflictResolver:
    """知识冲突消解器"""

    def __init__(self):
        # 数据源可信度权重（根据数据源的质量人工设定）
        self.source_trust = {
            "wikidata": 0.95,      # Wikidata 社区维护，质量
            "imdb": 0.90,          # IMDB 官方数据
            "douban": 0.85,        # 豆瓣
            "wikipedia": 0.88,     # 维基百科
            "user_generated": 0.60, # 用户生成内容
            "web_scraped": 0.70,   # 网页爬取
        }

    def resolve(self, property_name, values_with_sources):
        """
        消解属性冲突

        参数:
        - property_name: 属性名（如 "birth_date"）
        - values_with_sources: 带来源的值列表
          [{"value": "1970-07-30", "source": "wikidata"},
           {"value": "1970-07-31", "source": "douban"}]
        """
        if len(values_with_sources) <= 1:
            return values_with_sources[0]["value"] if valu

        # 策略1：投票法（多数决）
        value_counts = {}
        for item in values_with_sources:
            v = item["value"]
            value_counts[v] = value_counts.get(v, 0) + 1
        voting_result = max(value_counts, key=value_counts
```

```
        # 策略2：加权投票（按数据源可信度加权）
        weighted_scores = {}
        for item in values_with_sources:
            v = item["value"]
            trust = self.source_trust.get(item["source"],
            weighted_scores[v] = weighted_scores.get(v, 0)
        weighted_result = max(weighted_scores, key=weighte

        # 策略3：最新优先（适合经常变化的属性，如票房）
        # 需要有时间戳信息

        return {
            "property": property_name,
            "voting_result": voting_result,
            "weighted_result": weighted_result,
            "recommended": weighted_result,  # 推荐使用加权结
            "all_values": values_with_sources,
            "has_conflict": len(set(v["value"] for v in va
        }

# 测试
resolver = ConflictResolver()

# 场景：诺兰的出生日期有三个不同来源
print("冲突消解示例：")
conflict_data = [
    {"value": "1970-07-30", "source": "wikidata"},
    {"value": "1970-07-31", "source": "douban"},
    {"value": "1970-07-30", "source": "imdb"},
    {"value": "1970-07-30", "source": "wikipedia"},
]

result = resolver.resolve("birth_date", conflict_data)
print(f"  属性：{result['property']}")
print(f"  是否存在冲突：{result['has_conflict']}")
```

```
print(f"  投票结果: {result['voting_result']}")
print(f"  加权结果: {result['weighted_result']}")
print(f"  推荐值: {result['recommended']}")
```

知识质量评估

知识融合后，需要从多个维度评估知识图谱的质量。这就像盖完房子后的验收检查——确保结构稳固、功能完整。

```
class KnowledgeGraphQualityEvaluator:
    """知识图谱质量评估器"""

    def __init__(self, triples, schema=None):
        """
        参数:
        - triples: list of (subject, predicate, object) 元
        - schema: 本体定义（可选），用于检查合规性
        """
        self.triples = triples
        self.schema = schema
        self._build_index()

    def _build_index(self):
        """构建索引以加速查询"""
        self.subject_properties = {}
        self.all_entities = set()

        for s, p, o in self.triples:
            self.all_entities.add(s)
            self.all_entities.add(o)
            if s not in self.subject_properties:
                self.subject_properties[s] = {}
            if p not in self.subject_properties[s]:
                self.subject_properties[s][p] = []
            self.subject_properties[s][p].append(o)

    def completeness(self, required_properties_per_type=No
        """完整性评估：实体的必要属性是否被填充"""
        if not required_properties_per_type:
            required_properties_per_type = {
                "Movie": ["directedBy", "releaseDate", "ti
                "Person": ["name"],
            }
```

```
        scores = []
        for entity, props in self.subject_properties.items
            # 简化：假设所有实体使用相同的必要属性
            all_required = set()
            for req_list in required_properties_per_type.v
                all_required.update(req_list)

            filled = sum(1 for rp in all_required if rp in
            scores.append(filled / len(all_required) if al

        return sum(scores) / len(scores) if scores else 0

    def consistency(self):
        """一致性评估：检查是否存在逻辑矛盾"""
        issues = []

        # 检查1：同一实体的单值属性是否有多个不同值
        for entity, props in self.subject_properties.items
            for prop, values in props.items():
                # 某些属性应该是单值的（如 birthDate）
                single_value_props = {"birthDate", "releas
                if prop in single_value_props and len(set(
                    issues.append({
                        "type": "multi_value_single_proper
                        "entity": entity,
                        "property": prop,
                        "values": values
                    })

        consistency_score = 1 - (len(issues) / max(len(sel
        return max(0, consistency_score), issues

    def coverage(self, expected_entities=None):
        """覆盖率评估：预期实体是否都在图谱中"""
```

```
        if not expected_entities:
            return 1.0  # 没有预期列表，无法计算

        found = sum(1 for e in expected_entities if e in s
        return found / len(expected_entities)

    def generate_report(self, required_props=None):
        """生成综合质量报告"""
        completeness = self.completeness(required_props)
        consistency, issues = self.consistency()

        print("=" * 50)
        print("       知识图谱质量报告")
        print("=" * 50)
        print(f"  三元组总数:    {len(self.triples)}")
        print(f"  实体总数:      {len(self.all_entities)}")
        print(f"  完整性评分:    {completeness:.2%}")
        print(f"  一致性评分:    {consistency:.2%}")
        print(f"  综合评分:      {(completeness + consisten

        if issues:
            print(f"\n  发现 {len(issues)} 个质量问题:")
            for issue in issues[:5]:  # 只显示前5个
                print(f"    - [{issue['type']}] {issue['er
        else:
            print("\n  未发现一致性问题 ✓")

        print("=" * 50)

# 测试
triples = [
    ("Inception", "directedBy", "Christopher Nolan"),
    ("Inception", "releaseDate", "2010-07-16"),
    ("Inception", "rating", "8.8"),
    ("Inception", "genre", "SciFi"),
```

```
    ("Inception", "genre", "Action"),
    ("Interstellar", "directedBy", "Christopher Nolan"),
    ("Interstellar", "releaseDate", "2014-11-07"),
    ("Christopher Nolan", "birthDate", "1970-07-30"),
    ("Christopher Nolan", "nationality", "British"),
    ("Christopher Nolan", "name", "Christopher Nolan"),
]

evaluator = KnowledgeGraphQualityEvaluator(triples)
evaluator.generate_report()
```

第四步：知识存储

知识抽取和融合完成后，最后一步是将知识存入数据库。根据数据模型的不同，主要有两种存储方案：**属性图（Property Graph）**和**RDF 图**。

存入 Neo4j（属性图模型）

Neo4j 是目前最流行的图数据库，使用属性图模型——节点可以有属性和标签，关系也可以有属性。

```
# pip install neo4j
from neo4j import GraphDatabase

class KnowledgeGraphStore:
    """知识图谱存储：将三元组存入 Neo4j"""

    def __init__(self, uri="bolt://localhost:7687",
                 user="neo4j", password="your_password"):
        self.driver = GraphDatabase.driver(uri, auth=(user

    def close(self):
        self.driver.close()

    def create_constraints(self):
        """创建唯一性约束（提高查询效率和数据一致性）"""
        with self.driver.session() as session:
            # 为每种实体类型创建唯一性约束
            labels = ["Person", "Movie", "Genre", "Award",
            for label in labels:
                query = f"CREATE CONSTRAINT IF NOT EXISTS
                try:
                    session.run(query)
                    print(f"  创建约束: {label}.name 唯一")
                except Exception as e:
                    print(f"  约束创建失败 ({label}): {e}")

    def batch_import(self, triples, entity_types, relation
        """
        批量导入三元组

        参数:
        - triples: list of (subject, predicate, object)
        - entity_types: dict, entity_name -> node_label
        - relation_types: dict, predicate -> relationship_
```

```
        """
        with self.driver.session() as session:
            imported = 0
            for subject, predicate, obj in triples:
                subj_type = entity_types.get(subject, "Ent
                obj_type = entity_types.get(obj, "Entity")
                rel_type = relation_types.get(predicate, p

                query = f"""
                MERGE (a:{subj_type} {{name: $subject}})
                MERGE (b:{obj_type} {{name: $obj}})
                MERGE (a)-[r:{rel_type}]->(b)
                RETURN a, b
                """
                session.run(query, subject=subject, obj=ob
                imported += 1

            print(f"  成功导入 {imported} 条三元组")

    def add_properties(self, entity_name, entity_type, pro
        """为实体添加属性"""
        with self.driver.session() as session:
            props_str = ", ".join(f"n.{k} = ${k}" for k in
            query = f"""
            MATCH (n:{entity_type} {{name: $name}})
            SET {props_str}
            RETURN n
            """
            session.run(query, name=entity_name, **propert

    def query(self, cypher):
        """执行 Cypher 查询"""
        with self.driver.session() as session:
            result = session.run(cypher)
            return [record.data() for record in result]
```

```
# 使用示例
store = KnowledgeGraphStore()

# 创建约束
store.create_constraints()

# 导入电影知识
triples = [
    ("盗梦空间", "directedBy", "克里斯托弗·诺兰"),
    ("盗梦空间", "actedIn", "莱昂纳多·迪卡普里奥"),
    ("盗梦空间", "actedIn", "约瑟夫·高登-莱维特"),
    ("盗梦空间", "hasGenre", "科幻"),
    ("盗梦空间", "hasGenre", "动作"),
    ("星际穿越", "directedBy", "克里斯托弗·诺兰"),
    ("星际穿越", "actedIn", "马修·麦康纳"),
    ("星际穿越", "hasGenre", "科幻"),
]

entity_types = {
    "盗梦空间": "Movie", "星际穿越": "Movie",
    "克里斯托弗·诺兰": "Director",
    "莱昂纳多·迪卡普里奥": "Actor",
    "约瑟夫·高登-莱维特": "Actor",
    "马修·麦康纳": "Actor",
    "科幻": "Genre", "动作": "Genre",
}

relation_types = {
    "directedBy": "DIRECTED_BY",
    "actedIn": "ACTED_IN",
    "hasGenre": "HAS_GENRE",
}

store.batch_import(triples, entity_types, relation_types)
```

```
# 添加属性
store.add_properties("盗梦空间", "Movie", {
    "releaseDate": "2010-07-16",
    "rating": 9.3,
    "runtime": 148
})

# 查询验证
print("\n查询：诺兰导演的所有电影")
results = store.query("""
    MATCH (d:Director {name: '克里斯托弗·诺兰'})<-[:DIRECTED
    RETURN m.name AS movie
""")
for r in results:
    print(f"  {r['movie']}")

store.close()
```

导出为 RDF（语义网标准）

如果你的知识图谱需要与语义网生态系统对接，或者需要使用 SPARQL 查询语言，那么 RDF 格式是更好的选择：

```
# pip install rdflib
from rdflib import Graph, Literal, URIRef, Namespace, BNod
from rdflib.namespace import RDF, RDFS, XSD, OWL

def export_to_rdf(triples, namespace_uri="http://example.c
    """将三元组导出为 RDF/Turtle 格式"""
    g = Graph()

    # 定义命名空间
    movie = Namespace(namespace_uri)
    g.bind("movie", movie)
    g.bind("owl", OWL)
    g.bind("rdfs", RDFS)

    for subject, predicate, obj in triples:
        s = movie[subject.replace(" ", "_").replace("·", "
        p = movie[predicate]

        # 智能判断 object 类型
        if obj.replace(".", "").replace("-", "").isdigit()
            o = Literal(obj, datatype=XSD.date)
        elif obj.replace(".", "").isdigit():
            o = Literal(float(obj), datatype=XSD.float)
        else:
            # 默认作为 URI 引用（如果是实体）或字面值
            # 简单判断：如果首字母大写且无空格，视为实体引用
            if obj[0].isupper() and " " not in obj:
                o = movie[obj.replace(" ", "_")]
            else:
                o = movie[obj.replace(" ", "_").replace("·

        g.add((s, p, o))

    # 导出为 Turtle 格式
```

```
    turtle_output = g.serialize(format="turtle")
    return turtle_output

# 测试
triples = [
    ("Inception", "title", "Inception"),
    ("Inception", "directedBy", "Christopher_Nolan"),
    ("Inception", "releaseDate", "2010-07-16"),
    ("Inception", "rating", "8.8"),
]

rdf_output = export_to_rdf(triples)
print(rdf_output)

# 保存到文件
with open("movie_kg.ttl", "w", encoding="utf-8") as f:
    f.write(rdf_output)
print("\nRDF 文件已保存到 movie_kg.ttl")
```

LLM 辅助构建：端到端实战

现在让我们把所有环节串起来，完成一个完整的端到端实战项目——从一篇新闻文章中抽取知识，融合到已有的知识图谱中，并存入 Neo4j。

实战：从新闻到知识图谱

```
"""
完整实战：使用 spaCy + LLM 从新闻文章中端到端构建知识图谱

依赖安装：
  pip install spacy openai neo4j pyvis
  python -m spacy download zh_core_web_sm

运行前提：
  - 配置 OpenAI API Key
  - 启动本地 Neo4j 实例（或使用 Neo4j Aura 云实例）
"""

import spacy
import json
import openai
from collections import defaultdict
from pyvis.network import Network

# ============================================================
# Step 1: 准备新闻文本
# ============================================================

news_article = """
第95届奥斯卡颁奖典礼于2023年3月12日在洛杉矶好莱坞杜比剧院举行。
《瞬息全宇宙》成为最大赢家，斩获七项大奖。
杨紫琼凭借该片获得最佳女主角奖，成为首位获此殊荣的亚裔演员。
关继威获得最佳男配角奖。
该片由关家永和丹尼尔·施因纳特联合执导。

此外，《西线无战事》获得最佳国际影片等四项大奖。
布兰登·费舍凭借《鲸》获得最佳男主角奖。
杰米·李·柯蒂斯凭借《瞬息全宇宙》获得最佳女配角奖。
```

```
值得一提的是，杨紫琼出生于1962年8月6日，来自马来西亚。
她在获奖感言中说："这是希望和可能性的灯塔。"
"""

# ============================================================
# Step 2: 使用 spaCy 进行初步 NER（速度快，成本低）
# ============================================================

print("=== Step 2: spaCy 初步 NER ===")
nlp = spacy.load("zh_core_web_sm")
doc = nlp(news_article)

spacy_entities = defaultdict(list)
for ent in doc.ents:
    spacy_entities[ent.label_].append(ent.text)
    print(f"  [{ent.text}] → {ent.label_}")

# ============================================================
# Step 3: 使用 LLM 增强抽取（精度高，覆盖全）
# ============================================================

print("\n=== Step 3: LLM 增强抽取 ===")

def llm_extract_knowledge(text):
    """使用 LLM 进行端到端的知识抽取"""

    prompt = f"""你是一个知识图谱构建专家。请从以下新闻文本中抽取实

新闻文本：
{text}

请完成以下任务，并以 JSON 格式返回：

1. 实体识别：识别所有实体及其类型
```

```
    类型：Person, Movie, Award, Organization, Location, Date

2. 关系抽取：识别实体对之间的关系
    关系：wonAward, directedBy, actedIn, bornIn, bornOn, cit

3. 事件抽取：识别主要事件（包括事件类型、触发词、参与者、时间、地点）

返回格式：
{{
  "entities": [{{"name": "名称", "type": "类型"}}],
  "relations": [{{"subject": "主体", "predicate": "关系", "
  "events": [{{"type": "事件类型", "trigger": "触发词", "par
}}"""

    response = openai.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "你是精确的知识图谱
            {"role": "user", "content": prompt}
        ],
        temperature=0,
        response_format={"type": "json_object"}
    )

    return json.loads(response.choices[0].message.content)

knowledge = llm_extract_knowledge(news_article)

print(f"  实体数: {len(knowledge.get('entities', []))}")
print(f"  关系数: {len(knowledge.get('relations', []))}")
print(f"  事件数: {len(knowledge.get('events', []))}")

# ============================================================
# Step 4: 可视化知识图谱
# ============================================================
```

```
print("\n=== Step 4: 可视化 ===")

def visualize_knowledge(knowledge, output_file="oscar_kg.h
    """将抽取的知识可视化为交互式图"""
    net = Network(height="600px", width="100%", bgcolor="#
                  font_color="#333333", directed=True)

    # 设置物理引擎
    net.barnes_hut(gravity=-3000, spring_length=150)

    # 颜色映射
    type_colors = {
        "Person": "#4CAF50",   # 绿色
        "Movie": "#2196F3",    # 蓝色
        "Award": "#FF9800",    # 橙色
        "Location": "#9C27B0", # 紫色
        "Date": "#F44336",     # 红色
        "Country": "#00BCD4",  # 青色
    }

    # 添加实体节点
    for entity in knowledge.get("entities", []):
        color = type_colors.get(entity["type"], "#999999")
        net.add_node(
            entity["name"],
            label=entity["name"],
            title=f"类型: {entity['type']}",
            color=color,
            size=20
        )

    # 添加关系边
    for relation in knowledge.get("relations", []):
        net.add_edge(
```

```
            relation["subject"],
            relation["object"],
            title=relation["predicate"],
            label=relation["predicate"],
            arrows="to"
        )

    net.save_graph(output_file)
    print(f"  知识图谱已保存到 {output_file}")

visualize_knowledge(knowledge)

# ============================================================
# Step 5: 打印结构化结果
# ============================================================

print("\n=== 抽取结果汇总 ===")

print("\n实体列表：")
for entity in knowledge.get("entities", []):
    print(f"  [{entity['name']}] → {entity['type']}")

print("\n关系列表：")
for relation in knowledge.get("relations", []):
    conf = relation.get('confidence', 'N/A')
    print(f"  ({relation['subject']}) --{relation['predica

print("\n事件列表：")
for event in knowledge.get("events", []):
    print(f"  事件: {event['type']} (触发词: {event.get('tr
    print(f"    时间: {event.get('time', '未提及')}")
    print(f"    地点: {event.get('location', '未提及')}")
    for p in event.get("participants", []):
        print(f"    {p['role']}: {p['entity']}")
```

效果对比：传统方法 vs LLM 方法

表 11–2

维度	传统方法（规则+统计模型）	深度学习方法	LLM 方法
开发周期	数周至数月	数周	数小时至数天
标注数据需求	大量规则或标注	数千至数万条	零样本或少样本
泛化能力	受限于规则/训练域	受限于训练领域	跨领域泛化强
准确性	在覆盖范围内很高	在训练领域内高	整体高，偶有幻觉
推理速度	毫秒级	毫秒级	秒级
运行成本	低（本地）	低（本地）	较高（API调用）
可解释性	高（规则透明）	低（黑盒）	可通过 Prompt 解释
维护成本	高（维护规则）	中（重训练）	低（改 Prompt）

最佳实践：混合方案（推荐的工程方案）

```
大规模高频数据 → 传统方法（规则 + 预训练小模型），保证速度和成本
长尾罕见场景 → LLM 辅助处理，保证覆盖率
冷启动阶段 → LLM 预标注数据 + 人工审核，快速积累训练集
需要快速验证 → LLM 直接抽取，几小时内出 MVP
```

动手练习：从新闻到三元组

练习目标

综合运用本章学到的所有技术，从一篇科技新闻中完整抽取知识图谱，并可视化展示。

练习材料

使用以下新闻文本：

```
苹果公司于2023年6月5日在加利福尼亚州库比蒂诺的Apple Park举行了WWDC 2023开发者大会。CEO蒂姆·库克发布了Vision Pro头戴式显示器，这是一款混合现实设备，售价3499美元。Vision Pro搭载了M2芯片和全新的R1芯片，运行visionOS操作系统。

此外，苹果还发布了新款MacBook Air 15英寸版本，搭载M2芯片，起售价1299美元。iOS 17也在本次大会上发布，带来了全新的待机模式和联系人海报功能。
```

练习步骤

Step 1：安装依赖

```
pip install spacy openai pyvis
python -m spacy download zh_core_web_sm
```

Step 2：用 spaCy 做初步 NER

```
import spacy

nlp = spacy.load("zh_core_web_sm")
text = """（上面的新闻文本）"""
doc = nlp(text)

print("spaCy NER 结果：")
for ent in doc.ents:
    print(f"  {ent.text} → {ent.label_}")
```

观察结果：spaCy 能识别出哪些实体？有哪些实体被遗漏了？

Step 3：用 LLM 做精确抽取

参考上面实战部分的 `llm_extract_knowledge` 函数，对这篇新闻进行完整的实体、关系、事件抽取。对比 LLM 和 spaCy 的结果，看看 LLM 多识别出了哪些实体和关系。

Step 4：转换为三元组并存储

将 LLM 返回的 JSON 结果转换为三元组列表。可以选择存入 Neo4j（如果已安装）或者导出为 RDF 文件。

Step 5：可视化验证

使用 pyvis 将知识图谱可视化为交互式的 HTML 页面，检查节点和关系是否合理。

```
from pyvis.network import Network

def visualize_kg(triples, output_file="tech_kg.html"):
    """将三元组可视化为交互式图"""
    net = Network(height="600px", width="100%",
                  bgcolor="#ffffff", directed=True)
    net.barnes_hut(gravity=-2000, spring_length=120)

    nodes = set()
    for s, p, o in triples:
        nodes.add(s)
        nodes.add(o)
        net.add_edge(s, o, title=p, label=p, arrows="to")

    for node in nodes:
        net.add_node(node, label=node, title=node, size=15

    net.save_graph(output_file)
    print(f"知识图谱已保存到 {output_file}")
    print(f"  节点数: {len(nodes)}")
    print(f"  边数: {len(triples)}")

# 使用
triples = [
    ("苹果", "举办", "WWDC 2023"),
    ("苹果", "发布", "Vision Pro"),
    ("苹果", "发布", "MacBook Air 15"),
    ("苹果", "发布", "iOS 17"),
    ("Vision Pro", "搭载", "M2芯片"),
    ("Vision Pro", "搭载", "R1芯片"),
    ("Vision Pro", "运行", "visionOS"),
    ("Vision Pro", "售价", "3499美元"),
    ("MacBook Air 15", "搭载", "M2芯片"),
    ("MacBook Air 15", "售价", "1299美元"),
```

```
    ("蒂姆·库克", "担任", "CEO"),
    ("蒂姆·库克", "就职于", "苹果"),
    ("WWDC 2023", "举办地", "Apple Park"),
    ("WWDC 2023", "时间", "2023年6月5日"),
]

visualize_kg(triples)
```

进阶挑战

完成基本练习后，试试这些进阶扩展：

1. **处理多篇新闻**：写一个脚本，接受多个新闻文本，自动爬取并抽取知识图谱，然后融合
2. **增量更新**：当新的新闻到来时，自动与已有知识图谱进行实体对齐和融合
3. **质量评估**：实现一个自动评估脚本，用人工标注的结果计算抽取的 Precision、Recall、F1 分数
4. **查询接口**：为知识图谱添加一个自然语言查询接口，用户可以用自然语言提问
5. **对比实验**：对比只用 spaCy、只用 LLM、spaCy+LLM 三种方案的抽取效果

延伸阅读

1. **《Information Extraction》** —— Jurafsky & Martin 的《Speech and Language Processing》第17章，关系抽取的经典教材，理论扎实

2. **spaCy 官方文档和教程** —— https://spacy.io/ ，工业级 NLP 工具，支持多语言，文档非常友好
3. **HuggingFace NER 教程** —— https://huggingface.co/course ，最前沿的预训练模型实践，有大量可运行代码
4. **OpenAI Cookbook – Structured Outputs** —— https://cookbook.openai.com/ ，结构化输出的最佳实践
5. **Neo4j Graph Data Science** —— https://neo4j.com/docs/graph-data-science/ ，图算法和图分析，进阶必读
6. **《Knowledge Graphs》** —— Hogan et al. 2021，知识图谱的综合性教材，可免费在线获取：https://kgbook.org/
7. **OpenKG（开放知识图谱）** —— http://openkg.cn/ ，中文知识图谱社区，有丰富的中文资源、数据集和工具
8. **《大模型驱动的知识图谱构建》** —— 多篇中文综述论文，搜索 CNKI 或 arxiv 可获取最新进展

本章小结

本章我们完整走了一遍知识图谱构建的全流程。来回顾一下关键知识点：

四步流水线：

```
数据采集 → 信息抽取 → 知识融合 → 知识存储
```

每一步都环环相扣，缺一不可。前一步的质量直接决定了后一步的效果。

数据采集——原材料从哪来： – 三种渠道：开放数据集（最省事）、API 接口（最规范）、网页爬虫（最灵活） – 始终注意数据质量（去重、去噪、标准化）和合规性（robots.txt、版权）

信息抽取——核心技术环节： – NER 四代方法：规则 → 统计模型（CRF） → 深度学习（BERT+CRF） → LLM – 关系抽取：模板匹配 → 远程监督 → 预训练模型 → LLM – 事件抽取：从文本中识别事件及其参与者，LLM 是目前最实用的方案 – 大趋势：LLM 正在重塑信息抽取的范式，零样本抽取成为可能

知识融合——碎片拼图： – 实体对齐：识别不同数据源中的同一实体（名称归一化 + 相似度计算） – 冲突消解：投票法、加权法、时间戳法，各有适用场景 – 质量评估：完整性、一致性、覆盖率，三个维度缺一不可

知识存储——让知识”住”进去： – Neo4j（属性图）：直观的图模型，强大的 Cypher 查询，适合大多数应用 – RDF（语义网）：标准化格式，支持推理，适合需要跨系统对接的场景 – 选择取决于应用场景，不是越”高级”越好

LLM 辅助构建（重点趋势）： – 零样本/少样本抽取，大幅降低标注成本，加速项目启动 – 最佳实践：用 LLM 做标注，训练小模型来部署（兼顾质量和成本） – 混合方案：传统方法处理大规模高频数据，LLM 处理长尾罕见场景

到这里，你已经掌握了从零构建知识图谱的完整技能链。从本体设计（上一章）到数据采集、信息抽取、知识融合、知识存储（本章），你已经有了一套系统化的方法论。

下一章，我们将学习如何在构建好的知识图谱上进行推理和查询——让静态的知识”活”起来，真正为用户创造价值。

继续前进吧，未来的知识工程师！

第八章 Ubuntu 22.04开发环境搭建

> *工欲善其事，必先利其器。*
>
> —— 《论语·卫灵公》

> *难度：★☆☆ 适合入门*
> *预计耗时：2–3小时（含系统安装）*

引子

在正式踏入知识图谱的世界之前，我们需要先做好一件非常重要的事情：**搭建一个称手的工作环境**。

很多初学者——尤其是非计算机专业背景的朋友——在第一次接触开发环境时，往往会被各种名词搞得晕头转向：Linux、虚拟机、WSL、Docker、Python虚拟环境……这些概念一股脑涌来，让人无所适从。

别急。本章就是为你准备的。

我们会像搭积木一样，一块一块地把整个开发环境搭建起来。每一步都会给出完整的命令和终端输出示例，你只需要照着做就行。遇到问题也不用慌——我们会把常见的坑和解决方法都标注出来。

为什么选择Ubuntu？ 这不是个人偏好，而是知识图谱领域的事实标准。几乎所有主流图数据库（Neo4j、JanusGraph、TigerGraph）都优先支持Linux环境；所有AI/ML框架（PyTorch、TensorFlow、LangChain）在Linux上的安装和运行

都最为顺畅；你将来会接触到的Docker容器、Kubernetes编排，也都是Linux原生的。简单来说，选择Ubuntu，就是选择了一条阻力最小的道路。

准备好了吗？让我们开始吧。

8.1 为什么是Ubuntu 22.04?

在开始动手之前，先花一分钟理解我们的选择背后的逻辑。

8.1.1 稳定性的基石

Ubuntu 22.04 LTS（Jammy Jellyfish）是Canonical公司发布的长期支持版本。"LTS"意味着：

- **5年安全更新**：从2022年4月发布到2027年4月，你不需要操心系统过保
- **软件生态成熟**：几乎所有开发工具都已适配，不会遇到兼容性噩梦
- **社区资源丰富**：遇到问题，Google一下就能找到解决方案

8.1.2 开发工具链的完整性

在Ubuntu 22.04上，你可以用一条命令安装几乎所有我们需要用到的工具：

表 12–1

工具	用途	安装方式
Python 3.10+	主力编程语言	apt / deadsnakes PPA
pip	Python包管理	apt
Git	版本控制	apt
Docker	容器化部署	apt (Docker官方源)
Java 17	Neo4j运行时依赖	apt
Neo4j	图数据库	Docker / apt
VS Code	代码编辑器	deb包
curl / jq / httpie	API调试利器	apt / pip

8.1.3 三种安装方式对比

在安装Ubuntu之前，你需要选择一种方式。下表帮你对比：

表 12-2

方式	适合人群	优点	缺点	推荐度
WSL2（Windows子系统）	Windows用户	零成本、与Windows无缝集成	偶尔有网络/文件权限问题	★★★★★
虚拟机（VirtualBox / VMware）	任何平台	完全隔离、可快照回滚	占用资源多、性能损耗	★★★★☆
云服务器（阿里云/腾讯云/AWS）	有一定预算的读者	随时随地访问、性能强	需要付费、依赖网络	★★★★☆

> **树懒老K的建议**
>
> 如果你是Windows用户，强烈推荐WSL2。它让你同时享受Windows的易用性和Linux的强大。如果你是Mac用户，虚拟机或云服务器都可以，但云服务器更适合后续跑大型实验。

8.2 安装Ubuntu 22.04

8.2.1 方式一：WSL2安装（Windows用户首选）

前提条件：Windows 10 版本 2004 及以上，或 Windows 11。

第一步：启用WSL功能

以**管理员身份**打开PowerShell（右键点击”开始”菜单，选择”Windows终端（管理员）”或”PowerShell（管理员）”），执行：

```
wsl --install
```

执行后你会看到类似输出：

```
正在安装: 适用于 Linux 的 Windows 子系统
[==========================100.0%==========================
已成功安装适用于 Linux 的 Windows 子系统。

正在安装: 虚拟机平台
[==========================100.0%==========================
已成功安装虚拟机平台。

正在安装: Ubuntu
[==========================100.0%==========================
已成功安装 Ubuntu。
```

> **注意**
>
> 安装完成后需要**重启电脑**。

第二步：重启并设置用户

重启后，Ubuntu会自动打开一个终端窗口，要求你设置用户名和密码：

```
Installing, this may take a few minutes...
Please create a default UNIX user account. The username do
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: sloth
New password:
Retype new password:
passwd: password updated successfully
Installation successful!
To run a command as administrator (user "root"), use "sudo
See "man sudo_root" for details.

Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.133.1-micros

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

sloth@DESKTOP:~$
```

> **密码不会显示**
>
> 输入密码时屏幕上不会有任何反馈，这是Linux的安全设计，不是Bug。放心输入，然后按回车即可。

第三步：验证安装

在WSL终端中执行：

```
cat /etc/os-release
```

输出：

```
PRETTY_NAME="Ubuntu 22.04.3 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.3 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and
UBUNTU_CODENAME=jammy
```

第四步：更新系统包

这是每次安装Ubuntu后的第一件事——更新软件源并升级已安装的包：

```
sudo apt update && sudo apt upgrade -y
```

输出（截取关键部分）：

```
Hit:1 http://archive.ubuntu.com/ubuntu jammy InRelease
Get:2 http://security.ubuntu.com/ubuntu jammy-security InR
Get:3 http://archive.ubuntu.com/ubuntu jammy-updates InRel
Get:4 http://archive.ubuntu.com/ubuntu jammy-backports InR
...
Fetched 23.4 MB in 8s (2,925 kB/s)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
47 packages can be upgraded. Run 'apt list --upgradable' t
...
Setting up libc6:amd64 (2.35-0ubuntu3.6) ...
...
Processing triggers for libc-bin (2.35-0ubuntu3.6) ...
```

> **小贴士**
>
> `sudo` 表示"以管理员身份执行"。第一次使用sudo时会要求输入你刚设置的密码。 `-y` 表示自动回答"Yes"，省得手动确认。

8.2.2 方式二：虚拟机安装（通用方案）

第一步：下载VirtualBox

访问 https://www.virtualbox.org/wiki/Downloads 下载并安装对应平台的VirtualBox。

第二步：下载Ubuntu ISO镜像

访问 https://releases.ubuntu.com/22.04/ ，下载 `ubuntu-22.04.3-desktop-amd64.iso` （约4.7GB）。

> **树懒老K的建议**
>
> 如果你主要用命令行操作，可以下载Server版（约2GB），更轻量。但Desktop版有图形界面，对新手更友好。

第三步：创建虚拟机

打开VirtualBox，点击”新建”：

表 12-3

设置项	推荐值
名称	Ubuntu22-KG
类型	Linux
版本	Ubuntu (64-bit)
内存	4096 MB（至少）
虚拟硬盘	动态分配，40GB
CPU	2核（至少）

第四步：挂载ISO并安装

1. 在虚拟机设置中，进入”存储” → “没有光盘” → 选择刚下载的ISO文件
2. 启动虚拟机，按照图形向导选择语言、时区、用户名
3. 安装完成后重启，移除ISO挂载

第五步：安装增强工具（可选但推荐）

```
sudo apt install -y virtualbox-guest-utils virtualbox-gues
sudo reboot
```

这将启用剪贴板共享、自适应分辨率等功能。

8.2.3 方式三：云服务器（适合需要算力的读者）

以阿里云为例（腾讯云、AWS操作类似）：

1. 登录 https://ecs.console.aliyun.com/
2. 点击”创建实例”
3. 选择配置：
4. 地域：选择离你最近的节点
5. 实例规格：4核8G（ecs.c7.xlarge，适合学习和小规模实验）
6. 镜像：Ubuntu 22.04 64位
7. 系统盘：ESSD 40GB
8. 带宽：按量付费，5Mbps
9. 设置登录密码或SSH密钥
10. 创建完成后，用SSH连接：

```
ssh root@<你的服务器公网IP>
```

输出：

```
The authenticity of host '47.xxx.xxx.xxx (47.xxx.xxx.xxx)'
ECDSA key fingerprint is SHA256:xxxxxxxxxxxxxxxxxxxxxxxxxx
Are you sure you want to continue connecting (yes/no/[fing
Warning: Permanently added '47.xxx.xxx.xxx' (ECDSA) to the
root@47.xxx.xxx.xxx's password:
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.0-91-generic

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Last login: Mon Jan 15 10:30:00 2024 from 114.xxx.xxx.xxx
root@iZxxxxxxxxx:~#
```

> **安全提醒**
>
> 云服务器暴露在公网，务必设置强密码，并尽快配置SSH密钥登录和防火墙规则。

8.3 Python 3.10+：知识图谱的主力语言

8.3.1 验证当前Python版本

Ubuntu 22.04默认自带Python 3.10。验证一下：

```
python3 --version
```

输出：

```
Python 3.10.12
```

```
pip3 --version
```

输出：

```
pip 22.0.2 from /usr/lib/python3/dist-packages/pip (pythor
```

如果你的版本低于3.10，或者需要更新版本的Python，可以使用deadsnakes PPA：

```
sudo add-apt-repository ppa:deadsnakes/ppa -y
sudo apt update
sudo apt install -y python3.11 python3.11-venv python3.11-
```

8.3.2 pip与虚拟环境

pip是Python的包管理器，就像Ubuntu的apt一样，不过它管理的是Python库。

虚拟环境（venv） 是Python开发的最佳实践。它为每个项目创建独立的”沙盒”，避免不同项目的依赖互相冲突。

创建你的第一个虚拟环境：

```
# 创建项目目录
mkdir -p ~/projects/knowledge-graph
cd ~/projects/knowledge-graph

# 创建虚拟环境
python3 -m venv venv
```

此时项目目录下会多出一个`venv`文件夹：

```
ls -la venv/
```

输出：

```
total 24
drwxrwxr-x 5 sloth sloth 4096 Jan 15 11:00 .
drwxrwxr-x 3 sloth sloth 4096 Jan 15 11:00 ..
drwxrwxr-x 2 sloth sloth 4096 Jan 15 11:00 bin
drwxrwxr-x 2 sloth sloth 4096 Jan 15 11:00 include
drwxrwxr-x 3 sloth sloth 4096 Jan 15 11:00 lib
lrwxrwxrwx 1 sloth sloth    3 Jan 15 11:00 lib64 -> lib
-rw-rw-r-- 1 sloth sloth  152 Jan 15 11:00 pyvenv.cfg
```

激活虚拟环境：

```
source venv/bin/activate
```

激活后，终端提示符会变化：

```
# 激活前
sloth@ubuntu:~/projects/knowledge-graph$

# 激活后（注意前面的 (venv)）
(venv) sloth@ubuntu:~/projects/knowledge-graph$
```

在虚拟环境中安装包：

```
pip install --upgrade pip
pip install neo4j requests
```

输出：

```
Collecting neo4j
  Downloading neo4j-5.16.0.tar.gz (197 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 197.8/197.8 k
Collecting requests
  Downloading requests-2.31.0-py3-none-any.whl (62 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.6/62.6 kB
...
Successfully installed certifi-2023.11.17 charset-normaliz
idna-3.6 neo4j-5.16.0 pytz-2023.3.post1 requests-2.31.0
urllib3-2.1.0
```

退出虚拟环境：

```
deactivate
```

> **黄金法则**
>
> 每个项目一个虚拟环境。永远不要在系统全局环境中 `pip install`，这是新手最常犯的错误之一。

8.3.3 验证Python能正常工作

写一个小测试脚本，确认一切就绪：

```
# 确保虚拟环境已激活
cat > test_python.py << 'EOF'
import sys
import platform

print("=" * 50)
print("Python 环境检查")
print("=" * 50)
print(f"Python 版本: {sys.version}")
print(f"Python 路径: {sys.executable}")
print(f"操作系统: {platform.platform()}")
print(f"架构: {platform.machine()}")
print("=" * 50)
print("✅ 一切正常！Python 已就绪。")
EOF

python test_python.py
```

输出：

```
==================================================
Python 环境检查
==================================================
Python 版本: 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11
Python 路径: /home/sloth/projects/knowledge-graph/venv/bin/
操作系统: Linux-5.15.133.1-microsoft-standard-WSL2-x86_64-w
架构: x86_64
==================================================
✅ 一切正常！Python 已就绪。
```

8.4 Git：版本控制的基石

8.4.1 安装Git

Ubuntu 22.04通常已预装Git。验证：

```
git --version
```

输出：

```
git version 2.34.1
```

如果没有安装：

```
sudo apt install -y git
```

8.4.2 配置Git

第一次使用Git前需要配置身份：

```
git config --global user.name "树懒老K"
git config --global user.email "sloth@example.com"
```

设置默认编辑器（推荐nano，对新手最友好）：

```
git config --global core.editor nano
```

设置默认分支名为main：

```
git config --global init.defaultBranch main
```

验证配置：

```
git config --list
```

输出：

```
user.name=树懒老K
user.email=sloth@example.com
core.editor=nano
init.defaultbranch=main
```

8.4.3 你的第一个Git仓库

```
cd ~/projects/knowledge-graph
git init
```

输出：

```
hint: Using 'main' as the name for the initial branch.
hint: This can be changed with 'git branch -m <name>'.
hint: If this is your first time using Git, configure your
hint:
hint:   git config --global user.email "you@example.com"
hint:   git config --global user.name "Your Name"
Initialized empty Git repository in /home/sloth/projects/k
```

创建`.gitignore`文件，排除不需要跟踪的文件：

```
cat > .gitignore << 'EOF'
# Python 虚拟环境
venv/
__pycache__/
*.pyc
*.pyo

# IDE 配置
.vscode/
.idea/

# 系统文件
.DS_Store
Thumbs.db

# 数据文件（太大不适合版本控制）
data/
*.csv
*.dump
EOF

git add .gitignore
git commit -m "初始化项目并添加 .gitignore"
```

输出：

```
[main (root-commit) a3f2b1c] 初始化项目并添加 .gitignore
 1 file changed, 16 insertions(+)
 create mode 100644 .gitignore
```

8.5 Docker：容器化你的世界

8.5.1 什么是Docker?

如果你从未接触过Docker，可以把它理解为一个**超级轻量级的虚拟机**。它把应用程序及其所有依赖打包成一个”镜像”，在任何安装了Docker的机器上都能以完全相同的方式运行。

对我们来说，Docker的最大好处是：**安装Neo4j、Qdrant等数据库时，一条命令搞定，不用折腾复杂的依赖。**

8.5.2 安装Docker

第一步：卸载旧版本（如果有）

```
sudo apt remove -y docker docker-engine docker.io containe
```

第二步：安装前置依赖

```
sudo apt install -y ca-certificates curl gnupg lsb-release
```

第三步：添加Docker官方GPG密钥

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

第四步：设置Docker软件源

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/a
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

第五步：安装Docker

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io
  docker-buildx-plugin docker-compose-plugin
```

第六步：让当前用户可以运行Docker（免sudo）

```
sudo usermod -aG docker $USER
newgrp docker
```

第七步：验证安装

```
docker --version
```

输出：

```
Docker version 24.0.7, build afdd53b
```

```
docker compose version
```

输出：

```
Docker Compose version v2.23.3
```

运行经典的”hello-world”测试：

```
docker run hello-world
```

输出：

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest: sha256:fc6cf906cbfa013e80938cdf0bb199fbdbb9ce5c7f8

Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be wo

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from
    (amd64)
 3. The Docker daemon created a new container from that im
    executable that produces the output you are currently
 4. The Docker daemon streamed that output to the Docker c
    sent it to your terminal.
```

看到这段话，恭喜你，Docker已经安装成功了！

8.5.3 Docker基本操作速查

在我们后续的实战中，会频繁用到以下Docker命令。现在就熟悉一下：

```
# 查看正在运行的容器
docker ps

# 查看所有容器（包括已停止的）
docker ps -a

# 查看已下载的镜像
docker images

# 拉取镜像
docker pull neo4j:5.15

# 启动容器
docker run -d --name my-neo4j \
  -p 7474:7474 -p 7687:7687 \
  -v $HOME/neo4j/data:/data \
  neo4j:5.15

# 停止容器
docker stop my-neo4j

# 启动已停止的容器
docker start my-neo4j

# 查看容器日志
docker logs my-neo4j

# 进入容器内部
docker exec -it my-neo4j bash

# 删除容器
docker rm my-neo4j
```

> **关键概念**
>
> `-d` 表示后台运行（detached），`-p` 表示端口映射，`-v` 表示数据卷挂载。这些在后面的Neo4j安装中会详细解释。

8.6 Java 17：Neo4j的运行时依赖

Neo4j是用Java编写的，因此需要Java运行环境。Neo4j 5.x要求Java 17。

8.6.1 安装Java 17

```
sudo apt install -y openjdk-17-jdk
```

8.6.2 验证安装

```
java --version
```

输出：

```
openjdk 17.0.9 2023-10-17
OpenJDK Runtime Environment (build 17.0.9+9-Ubuntu-122.04)
OpenJDK 64-Bit Server VM (build 17.0.9+9-Ubuntu-122.04, mi
```

```
javac --version
```

输出：

```
javac 17.0.9
```

8.6.3 配置JAVA_HOME环境变量

很多Java应用（包括Neo4j）依赖`JAVA_HOME`环境变量。

```
# 找到Java安装路径
sudo update-alternatives --config java
```

输出中你会看到类似路径：

```
There is only one alternative in link group java:
  /usr/lib/jvm/java-17-openjdk-amd64/bin/java
```

将其设置为环境变量：

```
# 添加到 ~/.bashrc 末尾
echo '' >> ~/.bashrc
echo '# Java 17 环境变量' >> ~/.bashrc
echo 'export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64'
echo 'export PATH=$JAVA_HOME/bin:$PATH' >> ~/.bashrc

# 立即生效
source ~/.bashrc

# 验证
echo $JAVA_HOME
```

输出：

```
/usr/lib/jvm/java-17-openjdk-amd64
```

8.6.4 安装Maven（可选）

Maven是Java的构建工具。如果你只是使用Neo4j而不打算开发Java插件，可以跳过这步。但安装上也无妨：

```
sudo apt install -y maven
mvn --version
```

输出：

```
Apache Maven 3.6.3
Maven home: /usr/share/maven
Java version: 17.0.9, vendor: Ubuntu
```

8.7 CLI利器：提升效率的命令行工具

Linux的魅力在于命令行。以下这些工具虽然不是必需的，但它们会让你的开发体验提升一个档次。

8.7.1 curl：HTTP请求的瑞士军刀

Ubuntu 22.04已预装curl。验证：

```
curl --version | head -1
```

输出：

```
curl 7.81.0 (x86_64-pc-linux-gnu) libcurl/7.81.0 OpenSSL/3
```

一个典型用法——测试Neo4j的HTTP API：

```
curl -s http://localhost:7474 | python3 -m json.tool
```

8.7.2 jq：JSON处理神器

在后续与图数据库和向量数据库交互时，你会频繁接触JSON数据。jq让你能在命令行中优雅地处理JSON。

```
sudo apt install -y jq
jq --version
```

输出：

```
jq-1.6
```

使用示例：

```
# 格式化JSON输出
echo '{"name":"Neo4j","version":"5.15","type":"graph"}' |

# 提取特定字段
echo '{"name":"Neo4j","version":"5.15"}' | jq '.name'

# 处理数组
echo '[{"name":"Neo4j"},{"name":"Qdrant"}]' | jq '.[].name
```

输出：

```
{
  "name": "Neo4j",
  "version": "5.15",
  "type": "graph"
}
"Neo4j"
"Neo4j"
"Qdrant"
```

8.7.3 httpie：更人性化的curl

httpie让HTTP请求变得像说人话一样简单：

```
pip install httpie
```

对比一下curl和httpie的用法：

```
# curl 写法（冗长）
curl -X POST http://localhost:7474/db/neo4j/tx \
  -H "Content-Type: application/json" \
  -H "Authorization: Basic bmVvNGo6cGFzc3dvcmQ=" \
  -d '{"statements": [{"statement": "RETURN 1"}]}'

# httpie 写法（直观）
http POST http://localhost:7474/db/neo4j/tx \
  Authorization:"Basic bmVvNGo6cGFzc3dvcmQ=" \
  statements:='[{"statement": "RETURN 1"}]'
```

8.7.4 tmux：终端复用器

tmux让你在一个窗口中开多个”面板”，非常适合一边编辑代码一边运行程序：

```
sudo apt install -y tmux
tmux -V
```

输出：

```
tmux 3.2a
```

常用快捷键（先按`Ctrl+b`，松开后再按对应键）：

表 12–4

操作	快捷键	说明
水平分屏	`Ctrl+b` → `"`	上下分屏
垂直分屏	`Ctrl+b` → `%`	左右分屏
切换面板	`Ctrl+b` → 方向键	在不同面板间切换
新建窗口	`Ctrl+b` → `c`	创建新标签页
分离会话	`Ctrl+b` → `d`	退出但保持运行
重新连接	`tmux attach`	恢复之前的会话

8.7.5 htop：系统资源监控

```
sudo apt install -y htop
htop --version
```

输出：

```
htop 3.0.5
```

运行`htop`后你会看到一个彩色界面，显示CPU、内存、进程等信息。这在排查性能问题（比如Neo4j导入大量数据时内存不够用）时非常有用。

> **操作提示**
>
> 在htop界面中按 `q` 退出，按 `F9` 可以发送信号给进程（如kill），按 `F6` 可以按不同字段排序。

8.8 VS Code：最强开发利器

8.8.1 安装VS Code

方法一：通过apt安装（推荐）

```
# 添加微软GPG密钥
curl -fsSL https://packages.microsoft.com/keys/microsoft.a
  sudo gpg --dearmor -o /usr/share/keyrings/microsoft-arch

# 添加VS Code软件源
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/micros
  https://packages.microsoft.com/repos/code stable main" |
  sudo tee /etc/apt/sources.list.d/vscode.list > /dev/null

# 安装
sudo apt update
sudo apt install -y code
```

方法二：WSL用户专属

如果你用的是WSL2，最佳实践是在Windows端安装VS Code，然后通过”WSL扩展”远程连接到Ubuntu。步骤：

1. 在Windows上安装VS Code（https://code.visualstudio.com/）
2. 安装”WSL”扩展（在VS Code扩展市场搜索”WSL”）
3. 在WSL终端中执行：

```
code .
```

VS Code会自动打开并连接到WSL环境。

8.8.2 推荐扩展插件

打开VS Code，进入扩展市场（`Ctrl+Shift+X`），安装以下插件：

表 12–5

插件名	用途	重要性
Python	Python语言支持、调试、IntelliSense	⭐⭐⭐⭐⭐
Jupyter	在VS Code中运行Notebook	⭐⭐⭐⭐⭐
Cypher Query Language	Neo4j Cypher语法高亮	⭐⭐⭐⭐
Docker	Docker容器管理	⭐⭐⭐⭐
GitLens	Git增强：行级历史、blame等	⭐⭐⭐⭐
Markdown All in One	Markdown编辑增强	⭐⭐⭐
Material Icon Theme	文件图标美化	⭐⭐⭐
Prettier	代码格式化	⭐⭐⭐
Remote – SSH	SSH远程开发（云服务器用户）	⭐⭐⭐⭐

8.8.3 VS Code配置优化

按`Ctrl+Shift+P`，输入”Open User Settings (JSON)”，添加以下配置：

```
{
    "python.defaultInterpreterPath": "${workspaceFolder}/v
    "python.terminal.activateEnvironment": true,
    "editor.fontSize": 14,
    "editor.formatOnSave": true,
    "editor.minimap.enabled": false,
    "terminal.integrated.defaultProfile.linux": "bash",
    "files.exclude": {
        "**/__pycache__": true,
        "**/*.pyc": true,
        "**/venv": true
    }
}
```

8.9 环境验证：一键检查脚本

到这里，我们已经安装了所有必需的工具。让我们写一个综合检查脚本，确认一切就绪：

```
cat > ~/projects/knowledge-graph/env_check.sh << 'EOF'
#!/bin/bash

echo "╔═══════════════════════════════════════════════════════╗
echo "║    知识图谱开发环境检查工具 v1.0                      ║"
echo "║    by 树懒老K                                         ║"
echo "╚═══════════════════════════════════════════════════════╝"
echo ""

check_tool() {
    local name=$1
    local cmd=$2
    local version_flag=$3

    printf "%-20s" "$name"
    if command -v $cmd &> /dev/null; then
        version=$($cmd $version_flag 2>&1 | head -1)
        echo "✅ $version"
    else
        echo "❌ 未安装"
    fi
}

echo "--- 系统信息 ---"
echo "操作系统: $(cat /etc/os-release | grep PRETTY_NAME |
echo "内核版本: $(uname -r)"
echo "CPU核心数: $(nproc)"
echo "内存大小: $(free -h | awk '/Mem:/{print $2}')"
echo "磁盘空间: $(df -h / | awk 'NR==2{print $4}') 可用"
echo ""

echo "--- 开发工具 ---"
check_tool "Python" "python3" "--version"
check_tool "pip" "pip3" "--version"
```

```
check_tool "Git" "git" "--version"
check_tool "Docker" "docker" "--version"
check_tool "Docker Compose" "docker" "compose version"
check_tool "Java" "java" "--version"
check_tool "Maven" "mvn" "--version"
check_tool "curl" "curl" "--version"
check_tool "jq" "jq" "--version"
check_tool "httpie" "http" "--version"
check_tool "tmux" "tmux" "-V"
check_tool "htop" "htop" "--version"
echo ""

echo "--- Docker 服务状态 ---"
if docker info &> /dev/null; then
    echo "✅ Docker 守护进程运行中"
    echo "   容器数量: $(docker ps -q | wc -l) 运行中"
    echo "   镜像数量: $(docker images -q | wc -l) 已下载"
else
    echo "❌ Docker 守护进程未运行"
fi
echo ""

echo "--- Python 虚拟环境 ---"
if [ -d "venv" ]; then
    echo "✅ 虚拟环境已创建 (venv/)"
    if [ -n "$VIRTUAL_ENV" ]; then
        echo "   当前状态: ✅ 已激活"
    else
        echo "   当前状态: ⚠️  未激活 (运行 source venv/bin/a
    fi
else
    echo "⚠️  虚拟环境未创建 (运行 python3 -m venv venv)"
fi
echo ""
```

```
echo "═══════════════════════════════════════════════════"
echo "环境检查完成！如果以上全部为 ✅，你已经准备就绪。"
echo "═══════════════════════════════════════════════════"
EOF

chmod +x ~/projects/knowledge-graph/env_check.sh
bash ~/projects/knowledge-graph/env_check.sh
```

预期输出（假设所有工具已安装）：

```
╔═══════════════════════════════════════════════════╗
║    知识图谱开发环境检查工具 v1.0                  ║
║    by 树懒老K                                     ║
╚═══════════════════════════════════════════════════╝

--- 系统信息 ---
操作系统: Ubuntu 22.04.3 LTS
内核版本: 5.15.133.1-microsoft-standard-WSL2
CPU核心数: 8
内存大小: 7.7Gi
磁盘空间: 200G 可用

--- 开发工具 ---
Python              ✅ Python 3.10.12
pip                 ✅ pip 23.3.2 from ... (python 3.10)
Git                 ✅ git version 2.34.1
Docker              ✅ Docker version 24.0.7, build afdd53
Docker Compose      ✅ Docker Compose version v2.23.3
Java                ✅ openjdk 17.0.9 2023-10-17
Maven               ✅ Apache Maven 3.6.3
curl                ✅ curl 7.81.0
jq                  ✅ jq-1.6
httpie              ✅ 3.2.2
tmux                ✅ tmux 3.2a
htop                ✅ htop 3.0.5

--- Docker 服务状态 ---
✅ Docker 守护进程运行中
   容器数量: 0 运行中
   镜像数量: 1 已下载

--- Python 虚拟环境 ---
✅ 虚拟环境已创建 (venv/)
   当前状态: ✅ 已激活
```

```
═══════════════════════════════════════════
环境检查完成！如果以上全部为 ✅，你已经准备就绪。
═══════════════════════════════════════════
```

8.10 常见问题排查（FAQ）

Q1：WSL2中的Docker无法启动？

```
# 检查WSL版本
wsl --list --verbose
```

确保你的发行版使用的是WSL 2。如果是WSL 1，需要转换：

```
wsl --set-version Ubuntu-22.04 2
```

Q2：apt安装速度太慢？

更换为国内镜像源（以阿里云为例）：

```
# 备份原始源
sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak

# 替换为阿里云镜像
sudo sed -i 's/archive.ubuntu.com/mirrors.aliyun.com/g' /e
sudo sed -i 's/security.ubuntu.com/mirrors.aliyun.com/g' /

# 更新
sudo apt update
```

Q3：Docker拉取镜像太慢?

配置Docker镜像加速器（以阿里云为例）：

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
  "registry-mirrors": ["https://<你的专属加速地址>.mirror.al
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

> **提示**
>
> 访问 https://cr.console.aliyun.com/cn-hangzhou/instances/mirrors 获取你的专属加速地址。

Q4：磁盘空间不足怎么办?

```
# 查看磁盘使用情况
df -h

# 清理apt缓存
sudo apt clean
sudo apt autoremove -y

# 清理Docker无用资源
docker system prune -af
```

Q5：WSL2内存占用过大?

在Windows用户目录下创建`.wslconfig`文件：

```
# 在PowerShell中执行
notepad "$env:USERPROFILE\.wslconfig"
```

写入以下内容：

```
[wsl2]
memory=4GB
processors=4
swap=2GB
```

然后重启WSL：

```
wsl --shutdown
```

动手练习

练习 1：从零搭建（基础）

按照本章的步骤，在你的电脑上完成Ubuntu 22.04的安装（选择WSL2、虚拟机或云服务器中任一方式），并安装所有开发工具。运行环境检查脚本，确保所有项目都显示 ✅ 。

练习 2：Python虚拟环境练习（中级）

1. 创建一个新的项目目录 `~/projects/kg-practice`

2. 在其中创建虚拟环境
3. 激活虚拟环境后安装 `neo4j` 和 `requests` 包
4. 编写一个Python脚本，打印出已安装的所有包及版本号

   提示：

```
import pkg_resources
for pkg in pkg_resources.working_set:
    print(f"{pkg.project_name}: {pkg.version}")
```

练习 3：Docker初体验（进阶）

1. 使用Docker运行一个Nginx容器：`docker run -d -p 8080:80 nginx`
2. 在浏览器中访问 `http://localhost:8080`
3. 查看容器日志：`docker logs <容器名或ID>`
4. 停止并删除容器
5. 思考：这个过程和安装一个软件有什么本质区别？

延伸阅读

1. **Ubuntu官方文档**：https://docs.ubuntu.com/ —— 最权威的Ubuntu参考资料
2. **WSL2官方文档**：https://learn.microsoft.com/zh-cn/windows/wsl/ —— 微软出品的WSL完整指南

3. **Docker官方入门**：https://docs.docker.com/get-started/ —— Docker的官方Getting Started教程
4. **VS Code Python教程**：https://code.visualstudio.com/docs/python/python-tutorial —— 在VS Code中配置Python开发的完整指南
5. **Linux命令行圣经**（书籍）：William Shotts著，全面介绍Linux命令行操作的经典著作，免费在线阅读 https://linuxcommand.org/

本章小结

本章我们完成了知识图谱开发环境的搭建，涵盖了以下核心内容：

表 12-6

内容	要点
Ubuntu安装	WSL2（Windows首选）、虚拟机（通用）、云服务器（按需）
Python 3.10+	系统自带，重点掌握pip和venv虚拟环境
Git	版本控制的基础配置，包括身份设置和.gitignore
Docker	容器化引擎，后续Neo4j/Qdrant等数据库的安装基础
Java 17	Neo4j运行时依赖，配置JAVA_HOME环境变量
CLI工具	jq（JSON处理）、httpie（HTTP调试）、tmux（终端复用）、htop（监控）
VS Code	推荐IDE + 必装插件清单

至此，你的开发环境已经万事俱备。从下一章开始，我们将正式进入知识图谱的核心领域——首先是安装和精通Neo4j图数据库。

> **树懒老K的叮嘱**
>
> 环境搭建看似枯燥，但它是后续所有章节的基础。如果你在任何步骤遇到问题，不要跳过，务必解决后再继续。基础不牢，地动山摇。遇到问题时，先把报错信息完整复制到Google搜索——90%的问题别人都遇到过并留下了解决方案。

第九章 Neo4j从安装到精通

> *如果数据是21世纪的石油，那么关系就是炼油厂。*
>
> —— 佚名

> *难度：★☆☆ → ★★☆ 逐步进阶*
> *预计耗时：4–6小时（含动手实践）*

引子

在前面的章节中，我们反复提到一个概念：**知识图谱的核心是”关系”**。传统的表格型数据库（MySQL、PostgreSQL）在处理关系时力不从心——每多一层关系，就要多一次JOIN操作，性能呈指数级下降。

Neo4j的出现彻底改变了这个局面。作为全球最流行的图数据库，Neo4j把”关系”当作一等公民来对待。在Neo4j的世界里，关系不是表格之间的外键，而是实实在在存储在磁盘上的数据。这让”查找某个人的朋友的朋友的朋友”这样的查询，从$O(n^3)$变成了$O(1)$。

本章是全书最核心的实操章节之一。我们将从零开始，一步步把Neo4j从一个空壳数据库，变成一个能够支撑真实业务场景的知识图谱平台。你将学会安装、配置、数据导入、高级查询、性能调优等全套技能。

准备好了吗？让我们开始这段图数据库之旅。

9.1 Neo4j版本选择

在动手之前，我们需要了解Neo4j的不同版本及其适用场景。

9.1.1 版本矩阵

表 13–1

版本	价格	适用场景	核心限制
Community Edition	免费开源	学习、原型验证、小型项目	单节点、无集群、无企业级安全
Enterprise Edition	商业授权	生产环境、大规模部署	需要付费许可
AuraDB Free	云端免费	快速上手、不想自己装环境	50K节点限制、共享资源
AuraDB Professional	云端付费	生产级云服务	按需计费

> **树懒老K的建议**
>
> 学习和原型验证阶段，Community Edition完全够用。如果你想跳过安装步骤直接开始，可以注册一个AuraDB Free云实例（https://neo4j.com/cloud/aura-free/），5分钟就能开始使用。

9.1.2 Neo4j核心概念速览

在我们开始安装之前，先花两分钟理解Neo4j的核心数据模型：

属性图模型（Property Graph Model）

Neo4j使用”属性图”来组织数据，包含四种基本元素：

表 13-2

元素	说明	示例
节点（Node）	图中的实体	一个电影、一个演员
标签（Label）	节点的分类	`:Movie` 、 `:Person`
关系（Relationship）	节点之间的连接	`[:ACTED_IN]` 、 `[:DIRECTED]`
属性（Property）	节点或关系的键值对	`name: "汤姆·汉克斯"`

用一张图来理解：

```
(:Person {name:"汤姆·汉克斯"}) -[:ACTED_IN {role:"阿甘"}]->
```

这个简单的模型看似朴素，却能表达现实世界中极其复杂的关联网络。

9.2 安装Neo4j

我们提供三种安装方式，按推荐度排序。

9.2.1 方式一：Docker安装（最推荐）

这是最干净、最可复现的安装方式。一条命令搞定。

第一步：拉取Neo4j镜像

```
docker pull neo4j:5.15-community
```

输出：

```
5.15-community: Pulling from library/neo4j
a3ed95caeb02: Pull complete
...
2f4aeb20c56e: Pull complete
Digest: sha256:abc123...
Status: Downloaded newer image for neo4j:5.15-community
docker.io/library/neo4j:5.15-community
```

第二步：启动Neo4j容器

```
docker run -d \
  --name neo4j-kg \
  -p 7474:7474 \
  -p 7687:7687 \
  -v $HOME/neo4j/data:/data \
  -v $HOME/neo4j/logs:/logs \
  -v $HOME/neo4j/plugins:/plugins \
  -v $HOME/neo4j/import:/var/lib/neo4j/import \
  -e NEO4J_AUTH=neo4j/password123 \
  -e NEO4J_apoc_export_file_enabled=true \
  -e NEO4J_apoc_import_file_enabled=true \
  -e NEO4J_apoc_import_file_use__neo4j__config=true \
  neo4j:5.15-community
```

让我们逐个解释这些参数的含义：

表 13-3

参数	含义
`-d`	后台运行（detached模式）
`--name neo4j-kg`	给容器起名，方便后续操作
`-p 7474:7474`	映射HTTP端口（Neo4j Browser）
`-p 7687:7687`	映射Bolt端口（应用程序连接用）
`-v .../data:/data`	挂载数据目录，数据持久化
`-v .../logs:/logs`	挂载日志目录
`-v .../plugins:/plugins`	挂载插件目录（APOC、GDS等）
`-v .../import:/var/lib/neo4j/import`	挂载导入目录（CSV导入用）
`-e NEO4J_AUTH=...`	设置初始密码（跳过首次登录修改密码的步骤）
`-e NEO4J_apoc_...`	启用APOC插件的文件读写权限

第三步：确认启动成功

```
# 查看容器状态
docker ps
```

输出：

```
CONTAINER ID   IMAGE                      STATUS        POR
a1b2c3d4e5f6   neo4j:5.15-community       Up 10s        0.0
```

```
# 查看启动日志
docker logs neo4j-kg
```

输出：

```
Fetching versions.json for Plugin 'apoc' from https://neo4
Installing Plugin 'apoc' from https://github.com/neo4j-cor
Applying plugin apoc
Directories in use:
home: /var/lib/neo4j
config: /var/lib/neo4j/conf
logs: /logs
plugins: /plugins
import: /var/lib/neo4j/import
data: /var/lib/neo4j/data
...
2024-01-15 12:00:00.000+0000 INFO  Starting...
2024-01-15 12:00:01.234+0000 INFO  Bolt enabled on 0.0.0.0
2024-01-15 12:00:02.567+0000 INFO  Started.
2024-01-15 12:00:03.890+0000 INFO  Remote interface availa
```

当你看到”Remote interface available”这行日志时，说明Neo4j已经启动成功了！

第四步：在浏览器中打开Neo4j Browser

打开浏览器，访问 `http://localhost:7474`。

你会看到Neo4j Browser的登录界面。用户名是 `neo4j`，密码是 `password123`（就是我们在启动命令中设置的那个）。

> **安全提醒**
>
> `password123` 仅用于学习环境。生产环境请务必使用强密码。

9.2.2 方式二：APT安装（适合长期运行的服务器）

```
# 添加Neo4j GPG密钥
curl -fsSL https://debian.neo4j.com/neotechnology.gpg.key
  sudo gpg --dearmor -o /usr/share/keyrings/neo4j-archive-

# 添加Neo4j软件源
echo "deb [signed-by=/usr/share/keyrings/neo4j-archive-key
  https://debian.neo4j.com stable latest" | \
  sudo tee /etc/apt/sources.list.d/neo4j.list

# 安装
sudo apt update
sudo apt install -y neo4j
```

启动服务：

```
sudo systemctl start neo4j
sudo systemctl status neo4j
```

输出：

```
● neo4j.service - Neo4j Graph Database
     Loaded: loaded (/lib/systemd/system/neo4j.service; en
     Active: active (running) since Mon 2024-01-15 12:00:0
   Main PID: 12345 (java)
      Tasks: 35 (limit: 62987)
     Memory: 256.0M
        CPU: 3.456s
```

首次登录 `http://localhost:7474` 时，默认用户名密码是 `neo4j/neo4j`，系统会强制你修改密码。

9.2.3 方式三：Neo4j Desktop（图形界面）

Neo4j Desktop是一个跨平台的图形化管理工具，适合完全不想碰命令行的读者。

1. 访问 https://neo4j.com/download/ 下载Neo4j Desktop
2. 安装后打开，点击”New Project”
3. 点击”Add Database” → “Create a Local DBMS”
4. 设置名称和密码，点击”Create”
5. 点击”Start”启动数据库
6. 点击”Open”打开Neo4j Browser

> **提示**
>
> Neo4j Desktop自带APOC和GDS插件，无需手动安装，这是它的优势之一。

9.3 Neo4j Browser：你的图形化工作台

9.3.1 界面介绍

打开 `http://localhost:7474` 后，你会看到Neo4j Browser的界面，主要由以下部分组成：

- **左侧导航栏**：数据库信息、节点标签、关系类型、属性键的快速浏览
- **顶部命令栏**：输入Cypher查询的地方（是的，就是在这里写”SQL”）
- **主内容区**：显示查询结果——图可视化、表格数据或文本信息
- **底部状态栏**：显示查询耗时和返回记录数

9.3.2 第一个Cypher查询

在命令栏中输入以下查询，点击右侧的运行按钮（或按 `Ctrl+Enter`）：

```
// 查看数据库统计信息
MATCH (n)
RETURN count(n) AS totalNodes, count(DISTINCT labels(n)) A
```

由于我们的数据库是空的，结果会是：

表 13-4

totalNodes	labelTypes
0	0

让我们创建一些测试数据来感受一下：

```
// 创建第一个节点
CREATE (alice:Person {name: "爱丽丝", age: 28, city: "北京"
RETURN alice;
```

执行后你会看到一个蓝色的圆形节点出现在主内容区，上面标注了”Person”和属性信息。

继续创建更多节点和关系：

```
// 批量创建电影数据
CREATE (bob:Person {name: "鲍勃", age: 32, city: "上海"})
CREATE (charlie:Person {name: "查理", age: 25, city: "广州"

CREATE (matrix:Movie {title: "黑客帝国", year: 1999, rating
CREATE (inception:Movie {title: "盗梦空间", year: 2010, rat
CREATE (interstellar:Movie {title: "星际穿越", year: 2014,

// 创建关系
CREATE (alice)-[:ACTED_IN {role: "崔妮蒂"}]->(matrix)
CREATE (bob)-[:ACTED_IN {role: "柯布"}]->(inception)
CREATE (bob)-[:ACTED_IN {role: "库珀"}]->(interstellar)
CREATE (charlie)-[:DIRECTED]->(inception)
CREATE (alice)-[:KNOWS {since: 2015}]->(bob)
CREATE (bob)-[:KNOWS {since: 2020}]->(charlie)

RETURN *;
```

现在你应该能看到一个小型的社交网络图了。尝试拖拽节点来调整布局——这就是图数据库的魅力所在，数据之间的关系一目了然。

9.3.3 常用Browser命令

除了Cypher查询，Neo4j Browser还支持一些内置命令（以 `:` 开头）：

```
// 查看帮助
:help

// 查看数据库参数
:params

// 查看执行历史
:history

// 查看系统信息
:sysinfo

// 清空所有数据（谨慎使用！）
MATCH (n) DETACH DELETE n;

// 查看样式和主题设置
:style
```

9.4 Cypher查询语言：从入门到进阶

Cypher是Neo4j的查询语言，就像SQL之于关系型数据库。但Cypher的设计灵感来自ASCII art，用可视化的方式描述图模式。

9.4.1 基础CRUD操作

Create（创建）

```
// 创建单个节点
CREATE (p:Person {name: "大卫", age: 30})
RETURN p;

// 创建节点并同时创建关系
CREATE (p:Person {name: "伊芙"})-[:LIKES {score: 5}]->(m:M
RETURN p, m;
```

Read（查询）

```
// 查询所有Person节点
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age DESC;
```

输出：

表 13-5

p.name	p.age
“鲍勃”	32
“大卫”	30
“爱丽丝”	28
“查理”	25

```
// 查询关系
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN p.name, r.role, m.title;
```

输出：

表 13-6

p.name	r.role	m.title
“爱丽丝”	“崔妮蒂”	“黑客帝国”
“鲍勃”	“柯布”	“盗梦空间”
“鲍勃”	“库珀”	“星际穿越”

Update（更新）

```
// 更新属性
MATCH (p:Person {name: "爱丽丝"})
SET p.age = 29, p.email = "alice@example.com"
RETURN p;

// 添加新标签
MATCH (p:Person {name: "爱丽丝"})
SET p:Developer
RETURN labels(p);
```

Delete（删除）

```
// 删除节点（必须没有关系）
MATCH (p:Person {name: "大卫"})
DELETE p;

// 强制删除节点及其所有关系
MATCH (p:Person {name: "大卫"})
DETACH DELETE p;

// 清空整个数据库（危险！）
MATCH (n)
DETACH DELETE n;
```

9.4.2 模式匹配：Cypher的灵魂

Cypher最强大的地方在于模式匹配。你描述的图模式越精确，查询就越高效。

固定深度路径查询

```
// 查找"朋友的朋友"（2跳）
MATCH path = (p1:Person)-[:KNOWS]->(:Person)-[:KNOWS]->(p2
WHERE p1.name = "爱丽丝"
RETURN path;
```

```
// 查找爱丽丝认识的人演过什么电影
MATCH (alice:Person {name: "爱丽丝"})-[:KNOWS]->(friend:Per
RETURN friend.name AS 朋友, movie.title AS 电影;
```

输出：

表 13-7

朋友	电影
“鲍勃”	“盗梦空间”
“鲍勃”	“星际穿越”

可变长度路径查询

```
// 查找1到3跳内的所有可达节点
MATCH path = (p:Person)-[:KNOWS*1..3]->(other:Person)
WHERE p.name = "爱丽丝"
RETURN DISTINCT other.name AS 可达人物, length(path) AS 跳数
```

输出：

表 13-8

可达人物	跳数
“鲍勃”	1
“查理”	2

```
// 查找最短路径
MATCH path = shortestPath(
  (a:Person {name: "爱丽丝"})-[*]-(c:Person {name: "查理"})
)
RETURN path;
```

9.4.3 聚合与统计

```
// 每个演员参演了多少部电影
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name AS 演员, count(m) AS 电影数量
ORDER BY 电影数量 DESC;
```

```
// 电影的演员列表（聚合为数组）
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN m.title AS 电影, collect(p.name) AS 演员列表;
```

输出：

表 13-9

电影	演员列表
“黑客帝国”	[“爱丽丝”]
“盗梦空间”	[“鲍勃”]
“星际穿越”	[“鲍勃”]

```
// 度中心性分析：谁的连接最多？
MATCH (p:Person)
OPTIONAL MATCH (p)-[r]-()
RETURN p.name AS 人物, count(r) AS 连接数
ORDER BY 连接数 DESC;
```

9.4.4 高级查询技巧

WITH子句：管道化数据处理

```
// 找出评分高于平均分的电影
MATCH (m:Movie)
WITH avg(m.rating) AS avgRating
MATCH (m:Movie)
WHERE m.rating > avgRating
RETURN m.title AS 电影, m.rating AS 评分, avgRating AS 平均分
```

UNWIND：展开数组

```
// 批量创建节点
UNWIND ["动作", "科幻", "悬疑", "爱情"] AS genre
CREATE (g:Genre {name: genre})
RETURN g;
```

CASE表达式：条件逻辑

```
MATCH (m:Movie)
RETURN m.title AS 电影, m.rating AS 评分,
  CASE
    WHEN m.rating >= 9.0 THEN "经典"
    WHEN m.rating >= 8.0 THEN "优秀"
    WHEN m.rating >= 7.0 THEN "良好"
    ELSE "一般"
  END AS 评价;
```

子查询（CALL）

```
// 对每个电影统计演员数量
MATCH (m:Movie)
CALL {
    WITH m
    MATCH (p:Person)-[:ACTED_IN]->(m)
    RETURN count(p) AS actorCount
}
RETURN m.title AS 电影, actorCount AS 演员数量;
```

9.5 数据导入：从零构建你的知识图谱

这是本章最实战的部分。我们将学习三种数据导入方式，并最终构建一个包含真实数据的电影知识图谱。

9.5.1 方式一：Cypher CREATE/MERGE

适合小批量数据（几十到几百条）：

```
// MERGE vs CREATE：MERGE不会创建重复节点
MERGE (p:Person {id: "p001"})
ON CREATE SET p.name = "克里斯托弗·诺兰", p.born = 1970
ON MATCH SET p.updatedAt = datetime()
RETURN p;
```

> **MERGE的智慧**
>
> MERGE 会先查找是否存在匹配的节点，如果存在就不创建（可配合 ON MATCH 更新），如果不存在就创建（可配合 ON CREATE 设置初始值）。这避免了重复数据的问题。

9.5.2 方式二：LOAD CSV导入

这是最常用的批量导入方式，适合几千到几十万条数据。

准备CSV文件

创建以下文件并保存到 `$HOME/neo4j/import/movies.csv`：

```
movieId,title,year,rating,genres
m001,肖申克的救赎,1994,9.7,剧情|犯罪
m002,教父,1972,9.3,剧情|犯罪
m003,蝙蝠侠：黑暗骑士,2008,9.2,动作|犯罪|剧情
m004,辛德勒的名单,1993,9.5,剧情|历史|战争
m005,十二怒汉,1957,9.4,剧情
m006,指环王3：王者无敌,2003,9.2,剧情|动作|奇幻
m007,低俗小说,1994,8.9,剧情|犯罪
m008,盗梦空间,2010,9.3,科幻|动作|悬疑
m009,搏击俱乐部,1999,9.0,剧情
m010,黑客帝国,1999,8.7,科幻|动作
```

创建 `$HOME/neo4j/import/persons.csv`：

```
personId,name,born
p001,蒂姆·罗宾斯,1958
p002,摩根·弗里曼,1937
p003,马龙·白兰度,1924
p004,阿尔·帕西诺,1940
p005,克里斯蒂安·贝尔,1974
p006,希斯·莱杰,1979
p007,莱昂纳多·迪卡普里奥,1974
p008,基努·里维斯,1964
p009,布拉德·皮特,1963
p010,爱德华·诺顿,1969
```

创建 `$HOME/neo4j/import/roles.csv`：

```
personId,movieId,role
p001,m001,安迪
p002,m001,瑞德
p003,m002,维托·柯里昂
p004,m002,迈克尔·柯里昂
p005,m003,布鲁斯·韦恩
p006,m003,小丑
p007,m008,柯布
p008,m010,尼奥
p009,m009,泰勒
p010,m009,杰克
```

执行导入

```
// 第一步：导入电影节点
LOAD CSV WITH HEADERS FROM 'file:///movies.csv' AS row
CREATE (m:Movie {
    id: row.movieId,
    title: row.title,
    year: toInteger(row.year),
    rating: toFloat(row.rating),
    genres: split(row.genres, '|')
});

// 第二步：导入人物节点
LOAD CSV WITH HEADERS FROM 'file:///persons.csv' AS row
CREATE (p:Person {
    id: row.personId,
    name: row.name,
    born: toInteger(row.born)
});

// 第三步：创建关系
LOAD CSV WITH HEADERS FROM 'file:///roles.csv' AS row
MATCH (p:Person {id: row.personId})
MATCH (m:Movie {id: row.movieId})
CREATE (p)-[:ACTED_IN {role: row.role}]->(m);
```

执行后的结果：

```
Added 10 labels, created 10 nodes, set 50 properties, star
Added 10 labels, created 10 nodes, set 30 properties, star
Created 10 relationships, started streaming 0 records afte
```

查看导入结果

```
// 查看图谱全貌
MATCH (n)
RETURN labels(n)[0] AS 类型, count(n) AS 数量;
```

表 13–10

类型	数量
“Movie”	10
“Person”	10

```
// 可视化所有关系
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN p, r, m;
```

9.5.3 方式三：APOC导入（适合JSON等复杂格式）

APOC（Awesome Procedures On Cypher）是Neo4j最重要的插件，提供了数百个扩展过程。我们的Docker安装中已自动包含了APOC Core。

导入JSON数据

假设我们有一个JSON格式的电影详情文件 `$HOME/neo4j/import/movie_details.json`：

```
[
  {
    "id": "m001",
    "description": "银行家安迪被冤枉入狱，在肖申克监狱中凭借智慧和
    "director": "弗兰克·德拉邦特",
    "country": "美国",
    "language": "英语"
  },
  {
    "id": "m002",
    "description": "教父维托·柯里昂是黑手党柯里昂家族的掌门人，他
    "director": "弗朗西斯·福特·科波拉",
    "country": "美国",
    "language": "英语"
  }
]
```

使用APOC导入：

```
// 从JSON文件导入
CALL apoc.load.json('file:///movie_details.json') YIELD va
MATCH (m:Movie {id: value.id})
SET m.description = value.description,
    m.country = value.country,
    m.language = value.language
MERGE (d:Person:Director {name: value.director})
MERGE (d)-[:DIRECTED]->(m)
RETURN m.title, m.description;
```

9.6 APOC插件：Neo4j的瑞士军刀

9.6.1 APOC是什么?

APOC全称”Awesome Procedures On Cypher”，它为Cypher补充了大量实用功能，包括：

- 数据导入/导出（JSON、XML、JDBC）
- 图重构（节点合并、关系反转）
- 路径查找（最短路径、所有路径）
- 数据生成（随机图、测试数据）
- 系统管理（数据库信息、性能监控）

9.6.2 常用APOC过程

生成测试数据

```
// 生成随机人物节点（快速填充测试数据）
UNWIND range(1, 100) AS i
CREATE (p:Person {
    id: "test_" + toString(i),
    name: "测试用户" + toString(i),
    age: toInteger(rand() * 50 + 18)
});
```

图重构

```
// 将关系转换为中间节点（"关系节点化"）
// 例如：将 ACTED_IN 关系转换为 Role 节点
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
CALL apoc.refactor.extractNode(r, ['Role'], 'PLAYED_BY', '
YIELD output
RETURN output;
```

导出功能

```
// 导出子图为JSON
CALL apoc.export.json.query(
    "MATCH (p:Person)-[r:ACTED_IN]->(m:Movie) RETURN p, r,
    "subgraph.json"
);

// 导出为Cypher语句（可用于备份和迁移）
CALL apoc.export.cypher.all("backup.cypher", {});
```

数据库元信息查询

```
// 查看数据库统计信息
CALL apoc.meta.stats() YIELD labelCount, relTypeCount,
  propertyKeyCount, nodeCount, relCount;
```

输出：

表 13-11

labelCount	relTypeCount	propertyKeyCount	nodeCount	relCount
4	3	12	20	10

```
// 查看数据库Schema
CALL apoc.meta.schema() YIELD data RETURN data;
```

9.7 GDS：图数据科学库

9.7.1 什么是GDS?

GDS（Graph Data Science）是Neo4j官方的图算法库，提供了60多种图算法，包括：

表 13-12

类别	代表算法	用途
中心性	PageRank、Betweenness	找出图中最重要的节点
社区检测	Louvain、Label Propagation	发现图中的社群结构
相似度	Jaccard、Cosine	衡量节点间的相似程度
路径	Shortest Path、Steiner Tree	寻找最优路径
节点嵌入	Node2Vec、FastRP	将节点映射为向量

9.7.2 安装GDS插件

如果你使用Docker方式安装，需要额外添加GDS插件：

```
# 停止当前容器
docker stop neo4j-kg

# 删除旧容器（数据在挂载卷中，不会丢失）
docker rm neo4j-kg

# 下载GDS插件到plugins目录
wget -P $HOME/neo4j/plugins \
  https://github.com/neo4j/graph-data-science/releases/dow

# 重新启动容器，添加GDS许可
docker run -d \
  --name neo4j-kg \
  -p 7474:7474 \
  -p 7687:7687 \
  -v $HOME/neo4j/data:/data \
  -v $HOME/neo4j/logs:/logs \
  -v $HOME/neo4j/plugins:/plugins \
  -v $HOME/neo4j/import:/var/lib/neo4j/import \
  -e NEO4J_AUTH=neo4j/password123 \
  -e NEO4J_ACCEPT_LICENSE_AGREEMENT=yes \
  -e NEO4J_apoc_export_file_enabled=true \
  -e NEO4J_apoc_import_file_enabled=true \
  -e NEO4J_apoc_import_file_use__neo4j__config=true \
  --memory=4g \
  neo4j:5.15-community
```

验证GDS安装：

```
// 查看GDS版本
RETURN gds.version();
```

输出：

```
"2.6.2"
```

```
// 列出所有可用算法
CALL gds.list() YIELD name
RETURN name
ORDER BY name;
```

9.7.3 PageRank：找出最重要的节点

PageRank是Google创始之初使用的核心算法，用来衡量图中节点的重要性。

```
// 第一步：创建图投影（GDS需要在内存图中操作）
CALL gds.graph.project(
    'movie-graph',          // 图的名称
    ['Person', 'Movie'],     // 节点标签
    {
        ACTED_IN: {orientation: 'UNDIRECTED'},
        KNOWS: {orientation: 'UNDIRECTED'},
        DIRECTED: {orientation: 'UNDIRECTED'}
    }
);
```

输出：

表 13-13

graphName	nodeCount	relationshipCount	projectMillis
“movie-graph”	20	30	15

```
// 第二步：运行PageRank
CALL gds.pageRank.stream('movie-graph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS 名称,
       gds.util.asNode(nodeId).title AS 标题,
       score AS PageRank得分
ORDER BY score DESC
LIMIT 10;
```

输出（示例）：

表 13-14

名称	标题	PageRank得分
“鲍勃”	null	2.847
null	“盗梦空间”	2.351
“爱丽丝”	null	1.923
null	“黑客帝国”	1.654
…	…	…

```
// 第三步：将结果写回节点属性
CALL gds.pageRank.write('movie-graph', {
    writeProperty: 'pagerank'
})
YIELD nodePropertiesWritten, ranIterations;
```

9.7.4 社区检测：Louvain算法

Louvain算法能够自动发现图中的社群结构，非常适合用来分析”哪些人物和电影形成了紧密的群体”。

```
// 运行Louvain社区检测
CALL gds.louvain.stream('movie-graph')
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS 名称,
       gds.util.asNode(nodeId).title AS 标题,
       communityId AS 社区ID
ORDER BY communityId, 名称;
```

输出（示例）：

表 13–15

名称	标题	社区ID
“鲍勃”	null	3
“查理”	null	3
null	“盗梦空间”	3
null	“星际穿越”	3
“爱丽丝”	null	7
null	“黑客帝国”	7
…	…	…

```
// 写回结果并分析社区规模
CALL gds.louvain.write('movie-graph', {writeProperty: 'com
YIELD communityCount, modularity;
```

9.8 性能调优

9.8.1 索引：查询加速器

就像关系型数据库一样，Neo4j的索引能大幅提升查询性能。

```
// 创建索引
CREATE INDEX person_name FOR (p:Person) ON (p.name);
CREATE INDEX movie_title FOR (m:Movie) ON (m.title);
CREATE INDEX movie_year FOR (m:Movie) ON (m.year);

// 创建复合索引
CREATE INDEX person_name_age FOR (p:Person) ON (p.name, p.

// 创建唯一约束（自动创建索引）
CREATE CONSTRAINT movie_id_unique FOR (m:Movie) REQUIRE m.
CREATE CONSTRAINT person_id_unique FOR (p:Person) REQUIRE

// 查看已有索引
SHOW INDEXES;
```

输出：

表 13–16

id	name	state	type	entityType	labelsOrTypes	properties
1	person_name	ONLINE	RANGE	NODE	[“Person”]	[“name”]
2	movie_title	ONLINE	RANGE	NODE	[“Movie”]	[“title”]
3	movie_year	ONLINE	RANGE	NODE	[“Movie”]	[“year”]
4	movie_id_unique	ONLINE	RANGE	NODE	[“Movie”]	[“id”]
…	…	…	…	…	…	…

9.8.2 执行计划分析

使用 `EXPLAIN` 和 `PROFILE` 关键字来分析查询性能：

```
// EXPLAIN 只展示计划，不执行
EXPLAIN
MATCH (p:Person {name: "鲍勃"})-[:ACTED_IN]->(m:Movie)
RETURN p, m;

// PROFILE 展示计划并执行，包含实际数据
PROFILE
MATCH (p:Person {name: "鲍勃"})-[:ACTED_IN]->(m:Movie)
RETURN p, m;
```

PROFILE的输出会显示：

```
Cypher version: CYPHER 5, planner: COST, runtime: PIPELINE
10 total db hits in 2 ms.

+-----------------+----------------+------+---------+-----
| Operator        | Estimated Rows | Rows | DB Hits | Vari
+-----------------+----------------+------+---------+-----
| +ProduceResults |              2 |    2 |       0 | m --
| +Expand(Into)   |              2 |    2 |       5 | m --
| +NodeIndexSeek  |              1 |    1 |       2 | p
+-----------------+----------------+------+---------+-----
```

> ***关键指标：*** *– **DB Hits**：访问存储层的次数，越低越好 – **Rows**：流经每一行的数据量 – 如果看到 `NodeByLabelScan` 而不是 `NodeIndexSeek`，说明没有命中索引，需要检查索引配置*

9.8.3 内存配置

Neo4j的性能很大程度上取决于内存配置。编辑Docker启动参数或 `neo4j.conf`：

```
# 对于Docker安装，通过环境变量配置
docker stop neo4j-kg && docker rm neo4j-kg

docker run -d \
  --name neo4j-kg \
  -p 7474:7474 \
  -p 7687:7687 \
  -v $HOME/neo4j/data:/data \
  -v $HOME/neo4j/logs:/logs \
  -v $HOME/neo4j/plugins:/plugins \
  -v $HOME/neo4j/import:/var/lib/neo4j/import \
  -e NEO4J_AUTH=neo4j/password123 \
  -e NEO4J_ACCEPT_LICENSE_AGREEMENT=yes \
  -e NEO4J_apoc_export_file_enabled=true \
  -e NEO4J_apoc_import_file_enabled=true \
  -e NEO4J_apoc_import_file_use__neo4j__config=true \
  -e NEO4J_server_memory_heap_initial__size=1G \
  -e NEO4J_server_memory_heap_max__size=2G \
  -e NEO4J_server_memory_pagecache_size=1G \
  --memory=4g \
  neo4j:5.15-community
```

内存分配指南：

表 13–17

内存类型	建议值	说明
Heap（堆内存）	总内存的50%	Cypher查询、事务处理
Page Cache（页缓存）	总内存的25%	磁盘数据缓存
系统预留	总内存的25%	操作系统和JVM开销

> **经验法则**
>
> 如果你的图数据库有10GB的数据文件，Page Cache最好设置为10GB，这样所有数据都可以缓存在内存中，避免磁盘IO。

9.8.4 大批量导入优化

当数据量超过百万级时，需要注意导入策略：

```
// ❌ 错误方式：一次性导入所有数据（内存爆炸）
LOAD CSV WITH HEADERS FROM 'file:///huge_file.csv' AS row
CREATE (n:Node {id: row.id, name: row.name});

// ✅ 正确方式：分批导入
:auto LOAD CSV WITH HEADERS FROM 'file:///huge_file.csv' A
CALL {
    WITH row
    CREATE (n:Node {id: row.id, name: row.name})
} IN TRANSACTIONS OF 10000 ROWS;
```

> ***:auto 前缀**：告诉Browser使用自动事务管理。`IN TRANSACTIONS OF 10000 ROWS` 表示每10000行为一个事务提交一次，避免单个大事务占满内存。*

9.9 实战：构建10万节点的电影知识图谱

现在让我们把所有技能综合运用，构建一个大规模的、可用于分析的电影知识图谱。

9.9.1 数据准备

我们将使用Python脚本来生成大规模测试数据：

```
#!/usr/bin/env python3
"""生成10万节点的电影知识图谱测试数据"""

import csv
import random
import os

# 确保导入目录存在
IMPORT_DIR = os.path.expanduser("~/neo4j/import")
os.makedirs(IMPORT_DIR, exist_ok=True)

# 中国电影票房TOP数据（真实数据） + 虚构扩展数据
DIRECTORS = ["张艺谋", "陈凯歌", "冯小刚", "姜文", "吴京", "徐
             "宁浩", "管虎", "林超贤", "郭帆", "贾玲", "文牧野"
             "乌尔善", "路阳", "韩寒", "大鹏", "开心麻花", "曾

ACTORS_POOL = [f"演员_{i}" for i in range(1, 5001)]  # 500
GENRES = ["动作", "喜剧", "科幻", "爱情", "悬疑", "犯罪", "剧情
          "奇幻", "恐怖", "动画", "纪录片", "战争", "历史", "武

# 生成电影CSV (20000部)
with open(os.path.join(IMPORT_DIR, "movies_100k.csv"), 'w'
    writer = csv.writer(f)
    writer.writerow(["movieId", "title", "year", "rating",
    for i in range(1, 20001):
        title = f"电影_{i:05d}"
        year = random.randint(1990, 2024)
        rating = round(random.uniform(3.0, 9.8), 1)
        box_office = round(random.uniform(0.1, 60.0), 2)
        genres = "|".join(random.sample(GENRES, random.ran
        writer.writerow([f"m{i:05d}", title, year, rating,

# 生成导演CSV (200位)
with open(os.path.join(IMPORT_DIR, "directors_100k.csv"),
```

```
    writer = csv.writer(f)
    writer.writerow(["personId", "name", "born", "type"])
    for i, name in enumerate(DIRECTORS, 1):
        writer.writerow([f"d{i:03d}", name, random.randint
    for i in range(len(DIRECTORS)+1, 201):
        writer.writerow([f"d{i:03d}", f"导演_{i}", random.

# 生成演员CSV (5000位)
with open(os.path.join(IMPORT_DIR, "actors_100k.csv"), 'w'
    writer = csv.writer(f)
    writer.writerow(["personId", "name", "born", "type"])
    for i in range(1, 5001):
        writer.writerow([f"a{i:04d}", f"演员_{i}", random.

# 生成参演关系CSV (~80000条关系)
with open(os.path.join(IMPORT_DIR, "cast_100k.csv"), 'w',
    writer = csv.writer(f)
    writer.writerow(["personId", "movieId", "role"])
    for i in range(1, 20001):
        num_actors = random.randint(3, 6)  # 每部电影3-6个演
        actors = random.sample(range(1, 5001), num_actors)
        for j, actor_id in enumerate(actors):
            role = f"角色_{j+1}"
            writer.writerow([f"a{actor_id:04d}", f"m{i:05d

# 生成导演关系CSV (~20000条)
with open(os.path.join(IMPORT_DIR, "directed_100k.csv"), '
    writer = csv.writer(f)
    writer.writerow(["personId", "movieId"])
    for i in range(1, 20001):
        director_id = random.randint(1, 200)
        writer.writerow([f"d{director_id:03d}", f"m{i:05d}

print("数据生成完毕！")
print(f"文件保存在: {IMPORT_DIR}")
```

```
for fname in os.listdir(IMPORT_DIR):
    if fname.endswith('.csv') and '100k' in fname:
        fpath = os.path.join(IMPORT_DIR, fname)
        lines = sum(1 for _ in open(fpath))
        print(f"  {fname}: {lines-1} 条记录")
```

运行脚本：

```
cd ~/projects/knowledge-graph
source venv/bin/activate
python generate_movie_data.py
```

输出：

```
数据生成完毕!
文件保存在: /home/sloth/neo4j/import
  movies_100k.csv: 20000 条记录
  directors_100k.csv: 200 条记录
  actors_100k.csv: 5000 条记录
  cast_100k.csv: 80247 条记录
  directed_100k.csv: 20000 条记录
```

9.9.2 分批导入

```
// 创建约束（导入前创建，提升速度）
CREATE CONSTRAINT movie_id_unique IF NOT EXISTS
  FOR (m:Movie) REQUIRE m.id IS UNIQUE;
CREATE CONSTRAINT person_id_unique IF NOT EXISTS
  FOR (p:Person) REQUIRE p.id IS UNIQUE;

// 导入电影（20000条）
:auto LOAD CSV WITH HEADERS FROM 'file:///movies_100k.csv'
CALL {
    WITH row
    CREATE (m:Movie {
        id: row.movieId,
        title: row.title,
        year: toInteger(row.year),
        rating: toFloat(row.rating),
        boxOffice: toFloat(row.boxOffice),
        genres: split(row.genres, '|')
    })
} IN TRANSACTIONS OF 5000 ROWS;

// 导入导演（200条）
LOAD CSV WITH HEADERS FROM 'file:///directors_100k.csv' AS
CREATE (d:Person:Director {
    id: row.personId,
    name: row.name,
    born: toInteger(row.born)
});

// 导入演员（5000条）
:auto LOAD CSV WITH HEADERS FROM 'file:///actors_100k.csv'
CALL {
    WITH row
```

```
    CREATE (a:Person:Actor {
        id: row.personId,
        name: row.name,
        born: toInteger(row.born)
    })
} IN TRANSACTIONS OF 5000 ROWS;

// 创建参演关系 (~80000条)
:auto LOAD CSV WITH HEADERS FROM 'file:///cast_100k.csv' A
CALL {
    WITH row
    MATCH (a:Person {id: row.personId})
    MATCH (m:Movie {id: row.movieId})
    CREATE (a)-[:ACTED_IN {role: row.role}]->(m)
} IN TRANSACTIONS OF 5000 ROWS;

// 创建导演关系 (~20000条)
:auto LOAD CSV WITH HEADERS FROM 'file:///directed_100k.cs
CALL {
    WITH row
    MATCH (d:Person {id: row.personId})
    MATCH (m:Movie {id: row.movieId})
    CREATE (d)-[:DIRECTED]->(m)
} IN TRANSACTIONS OF 5000 ROWS;
```

9.9.3 全图分析

```
// 全局统计
MATCH (n)
RETURN labels(n) AS 标签, count(n) AS 数量
UNION ALL
MATCH ()-[r]->()
RETURN type(r) AS 关系类型, count(r) AS 数量;
```

表 13–18

标签/类型	数量
[“Movie”]	20,000
[“Person”, “Actor”]	5,000
[“Person”, “Director”]	200
ACTED_IN	~80,000
DIRECTED	20,000

```
// 分析：产量最高的导演TOP10
MATCH (d:Person:Director)-[:DIRECTED]->(m:Movie)
RETURN d.name AS 导演, count(m) AS 导演作品数,
       avg(m.rating) AS 平均评分
ORDER BY 导演作品数 DESC
LIMIT 10;

// 分析：评分最高的电影TOP10
MATCH (m:Movie)
RETURN m.title AS 电影, m.year AS 年份, m.rating AS 评分
ORDER BY m.rating DESC
LIMIT 10;

// 分析：参演最多电影的演员TOP10
MATCH (a:Person:Actor)-[:ACTED_IN]->(m:Movie)
RETURN a.name AS 演员, count(m) AS 参演数量,
       avg(m.rating) AS 平均评分
ORDER BY 参演数量 DESC
LIMIT 10;

// 分析：各年代的电影产量
MATCH (m:Movie)
WITH toInteger(m.year / 10) * 10 AS decade, count(m) AS co
RETURN decade + "-" + (decade + 9) AS 年代, count AS 电影数
ORDER BY decade;

// 分析：类型分布
MATCH (m:Movie)
UNWIND m.genres AS genre
RETURN genre AS 类型, count(*) AS 数量
ORDER BY 数量 DESC;

// 分析：导演的"御用演员"（合作次数最多）
MATCH (d:Person:Director)-[:DIRECTED]->(m:Movie)<-[:ACTED_
```

```
WITH d, a, count(m) AS collaborations
WHERE collaborations >= 3
RETURN d.name AS 导演, a.name AS 演员, collaborations AS 合
ORDER BY collaborations DESC
LIMIT 20;
```

9.9.4 图算法分析

```
// 创建图投影
CALL gds.graph.project(
    'movie-analysis',
    {
        Person: {label: 'Person'},
        Movie: {label: 'Movie'}
    },
    {
        ACTED_IN: {type: 'ACTED_IN', orientation: 'UNDIREC
        DIRECTED: {type: 'DIRECTED', orientation: 'UNDIREC
    }
);

// PageRank：找出最有影响力的节点
CALL gds.pageRank.stream('movie-analysis')
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS node, score
WHERE node:Person
RETURN node.name AS 人物, score AS 影响力得分
ORDER BY score DESC
LIMIT 15;

// 社区检测：发现合作群体
CALL gds.louvain.write('movie-analysis', {
    writeProperty: 'community',
    maxIterations: 20
})
YIELD communityCount, modularity, ranIterations;

// 查看社区分布
MATCH (p:Person)
WITH p.community AS communityId, count(p) AS memberCount
```

```
WHERE memberCount >= 5
RETURN communityId AS 社区ID, memberCount AS 成员数
ORDER BY memberCount DESC
LIMIT 10;

// 清理图投影（释放内存）
CALL gds.graph.drop('movie-analysis');
```

9.10 使用Python连接Neo4j

在后续章节中，我们会频繁使用Python来操作Neo4j。以下是基本的连接和操作方式：

```
"""Neo4j Python 驱动基本操作"""

from neo4j import GraphDatabase

# 连接配置
URI = "bolt://localhost:7687"
AUTH = ("neo4j", "password123")

# 创建驱动实例
driver = GraphDatabase.driver(URI, auth=AUTH)

# 验证连接
with driver.session() as session:
    result = session.run("RETURN gds.version() AS version"
    print(f"GDS版本: {result.single()['version']}")

# 封装常用操作
class MovieGraph:
    def __init__(self, uri, auth):
        self.driver = GraphDatabase.driver(uri, auth=auth)

    def close(self):
        self.driver.close()

    def find_actors_of_movie(self, movie_title):
        """查找某部电影的所有演员"""
        query = """
        MATCH (p:Person:Actor)-[:ACTED_IN]->(m:Movie {titl
        RETURN p.name AS actor, p.born AS born
        ORDER BY p.born
        """
        with self.driver.session() as session:
            result = session.run(query, title=movie_title)
            return [record.data() for record in result]
```

```
    def find_co_actors(self, actor_name):
        """查找某演员的所有合作演员"""
        query = """
        MATCH (a:Person {name: $name})-[:ACTED_IN]->(m:Mov
        WHERE a <> co
        RETURN DISTINCT co.name AS co_actor, count(m) AS c
        ORDER BY collaborations DESC
        """
        with self.driver.session() as session:
            result = session.run(query, name=actor_name)
            return [record.data() for record in result]

    def shortest_path_between(self, name1, name2):
        """查找两人之间的最短路径"""
        query = """
        MATCH path = shortestPath(
            (p1:Person {name: $name1})-[*..6]-(p2:Person {
        )
        RETURN [n IN nodes(path) | n.name] AS names,
               length(path) AS hops
        """
        with self.driver.session() as session:
            result = session.run(query, name1=name1, name2
            record = result.single()
            if record:
                return {"path": record["names"], "hops": r
            return None

    def run_page_rank(self, top_n=10):
        """运行PageRank算法"""
        query = """
        CALL gds.pageRank.stream('movie-analysis')
        YIELD nodeId, score
        WITH gds.util.asNode(nodeId) AS node, score
```

```
        WHERE node:Person
        RETURN node.name AS name, score
        ORDER BY score DESC
        LIMIT $top_n
        """
        with self.driver.session() as session:
            result = session.run(query, top_n=top_n)
            return [record.data() for record in result]

# 使用示例
if __name__ == "__main__":
    graph = MovieGraph(URI, AUTH)

    print("=== 某电影的演员 ===")
    actors = graph.find_actors_of_movie("电影_00001")
    for a in actors[:5]:
        print(f"  {a}")

    print("\n=== 合作演员 ===")
    co_actors = graph.find_co_actors("演员_0001")
    for c in co_actors[:5]:
        print(f"  {c}")

    print("\n=== 最短路径 ===")
    path = graph.shortest_path_between("演员_0001", "演员_1
    print(f"  {path}")

    graph.close()
```

运行结果（示例）：

```
=== 某电影的演员 ===
  {'actor': '演员_1234', 'born': 1965}
  {'actor': '演员_2345', 'born': 1972}
  {'actor': '演员_3456', 'born': 1978}
  {'actor': '演员_4567', 'born': 1985}

=== 合作演员 ===
  {'co_actor': '演员_0234', 'collaborations': 3}
  {'co_actor': '演员_0567', 'collaborations': 2}

=== 最短路径 ===
  {'path': ['演员_0001', '电影_00123', '演员_0456', '电影_01
```

动手练习

练习 1：基础安装与查询（★☆☆）

1. 使用Docker安装并启动Neo4j
2. 在Neo4j Browser中创建5个电影节点和10个演员节点
3. 创建至少15条ACTED_IN关系
4. 编写Cypher查询：找出参演电影最多的演员

练习 2：CSV导入实战（★☆☆）

1. 从 https://www.kaggle.com 下载一个真实的电影数据集（如IMDB数据集）
2. 将数据转换为CSV格式，放入Neo4j导入目录
3. 使用LOAD CSV导入数据

4. 创建索引后，对比有索引和无索引时的查询性能（使用PROFILE）

练习 3：图算法分析（★★☆）

1. 使用第9.9节的Python脚本生成10万节点数据
2. 导入数据后，运行PageRank算法找出最有影响力的演员
3. 运行Louvain社区检测，分析发现的社群结构
4. 用Python驱动编写一个函数：输入演员名字，返回该演员所在社区中的所有其他演员

练习 4：高级Cypher挑战（★★☆）

用Cypher实现以下分析（基于导入的10万节点数据）：

1. 找出"六度分隔"：任意两个演员之间最多通过6层合作关系能否相连？
2. 计算每个导演的"演员多样性"——与其合作的演员中有多少不同的社区？
3. 找出"黄金组合"：导演+演员组合中，合作次数>=3且平均评分最高的TOP5

延伸阅读

1. **Neo4j官方文档**：https://neo4j.com/docs/ —— 最权威的参考资料
2. **Cypher查询语言参考**：https://neo4j.com/docs/cypher-manual/ —— Cypher语法大全

3. **Neo4j Graph Data Science手册**：https://neo4j.com/docs/graph-data-science/ —— GDS算法文档
4. **APOC用户指南**：https://neo4j.com/labs/apoc/ —— APOC插件完整文档
5. **《Graph Databases》（书籍）**：Ian Robinson等著，O'Reilly出版，图数据库领域的经典著作，可在https://neo4j.com/graph-databases-book/ 免费下载
6. **Neo4j Sandbox**：https://sandbox.neo4j.com/ —— 免费的在线Neo4j实验环境，提供预置数据集

本章小结

本章我们深入学习了Neo4j图数据库的方方面面，从安装到精通，覆盖了完整的技能链：

表 13–19

主题	核心内容
版本选择	Community（免费学习）vs Enterprise（生产）vs AuraDB（云端）
安装部署	Docker（推荐）、APT、Neo4j Desktop 三种方式
Browser	图形化查询界面的使用和常用命令
Cypher	CRUD基础、模式匹配、路径查询、聚合统计、高级技巧
数据导入	CREATE/MERGE（小批量）、LOAD CSV（批量）、APOC（复杂格式）
APOC插件	数据导入/导出、图重构、元信息查询
GDS算法	PageRank（中心性）、Louvain（社区检测）等图算法
性能调优	索引、执行计划分析、内存配置、分批导入
Python集成	neo4j驱动的连接、查询封装、算法调用
实战项目	10万节点电影图谱的完整构建和分析流程

树懒老K的总结

Neo4j不仅仅是一个数据库——它是一种全新的数据思维方式。当你开始用”节点”和”关系”来看待世界时，你会发现很多原本复杂的问题变得异常清晰。在接下来的章节中，我们将把Neo4j与向量数据库Qdrant结合，打造AI时代的”黄金组合”。

第十章 Qdrant：向量数据库与图谱的交汇

> ***观察**："阿甘正传"和"温馨感动"的相似度高达0.82，而"阿甘正传"和"终结者"只有0.35。这就是向量搜索的魔力*
>
> —— 它能理解语义，而不只是匹配关键词。

> *难度：★★☆ 中等*
> *预计耗时：3–4小时（含动手实践）*

引子

> *“在AI时代，搜索不再是匹配关键词，而是理解语义。”*

在前面的章节中，我们用Neo4j构建了一个结构化的知识图谱——节点、关系、属性，一切都是有形的、可精确描述的。但现实世界中，大量的信息以非结构化的形式存在：一段电影描述、一篇影评、一段对话记录。这些信息无法简单地用”节点+关系”来表达。

如何让知识图谱”理解”这些非结构化的信息？

答案是：**向量数据库**。

向量数据库的核心思想极其优雅——把文本（图片、音频等）转换成一串数字（向量），然后在高维空间中通过”距离”来衡量语义的相似度。”阿甘正传是一部温暖人心的电影”和”这部电影让人感到温馨和感动”——虽然措辞完全不同，但它们的向量在高维空间中会非常接近。

本章我们将学习Qdrant——一个高性能的开源向量数据库，并探索它如何与Neo4j图数据库协作，构建AI时代的”黄金组合”。

10.1 向量数据库：为什么需要它?

10.1.1 传统搜索的局限

传统的文本搜索（关键词匹配）有一个根本性的问题：**它只认识”字面”，不认识”语义”**。

举个例子。假设你在一个电影数据库中搜索”温暖感人的故事”，传统搜索会去找包含”温暖”、”感人”、”故事”这些关键词的记录。但以下描述可能一个关键词都没命中，却完美匹配你的搜索意图：

- “一个关于希望和救赎的动人故事”
- “看完这部电影，我的眼眶湿润了”
- “在绝望中找到生命的光芒”

关键词搜索对这些完全无能为力。

10.1.2 向量搜索的革命

向量搜索换了一种思路：

1. **编码**：用AI模型（如OpenAI的text-embedding模型）把每段文本转换成一个高维向量，比如1536维的浮点数数组
2. **存储**：把这些向量存入向量数据库
3. **搜索**：把搜索词也转换成向量，然后在数据库中找到”距离最近”的向量

```
文本: "温暖感人的故事"
  ↓ Embedding模型
向量: [0.023, -0.156, 0.892, ..., 0.045]  (1536维)

文本: "一个关于希望和救赎的动人故事"
  ↓ Embedding模型
向量: [0.021, -0.148, 0.885, ..., 0.041]  (1536维)

→ 余弦相似度: 0.96  (非常接近!)
```

10.1.3 向量数据库生态

目前主流的向量数据库有很多，我们选择Qdrant的理由如下：

表 14-1

数据库	特点	Qdrant优势对比
Pinecone	全托管云服务	Qdrant开源、可自托管
Milvus	性能强劲、功能全面	Qdrant更轻量、API更简洁
Weaviate	内置向量化模块	Qdrant的Rust核心性能更优
ChromaDB	极简、Python原生	Qdrant生产级、支持分布式
Qdrant	Rust编写、高性能、REST/gRPC双协议	← 本章主角

10.2 向量嵌入：从文本到高维空间

10.2.1 什么是Embedding?

Embedding（嵌入/向量嵌入） 是把离散的、非结构化的数据（文字、图片、音频）映射到连续的高维向量空间的过程。

想象一个二维空间：

在这个简化的二维空间中，”阿甘正传”和”泰坦尼克号”因为都是温情类电影，所以距离较近；而”终结者”是动作片，距离较远。

真实的Embedding是1536维（OpenAI text-embedding-ada-002）或768维（BERT）甚至更多。虽然人类无法可视化这么高维的空间，但数学原理是一样的：**语义相似的文本，向量距离更近**。

10.2.2 常用的Embedding模型

表 14-2

模型	维度	来源	中文支持	适用场景
text-embedding-ada-002	1536	OpenAI	⭐⭐⭐⭐	通用文本
text-embedding-3-small	1536	OpenAI	⭐⭐⭐⭐	通用文本（新一代）
text-embedding-3-large	3072	OpenAI	⭐⭐⭐⭐⭐	高质量需求
bge-large-zh	1024	BAAI（智源）	⭐⭐⭐⭐⭐	中文最优之一
m3e-base	768	Moka AI	⭐⭐⭐⭐⭐	中文轻量
sentence-transformers	384-768	HuggingFace	⭐⭐⭐	本地部署

> **树懒老K的建议**
>
> 如果你主要处理中文内容，推荐使用 `bge-large-zh` （北京智源研究院出品），它在中文语义理解上远超OpenAI的模型，而且完全免费、可本地部署。

10.2.3 动手体验Embedding

先安装必要的Python库：

```
cd ~/projects/knowledge-graph
source venv/bin/activate
pip install sentence-transformers openai numpy
```

使用开源模型（本地，免费）：

```
"""使用 sentence-transformers 生成文本嵌入"""

from sentence_transformers import SentenceTransformer
import numpy as np

# 加载模型（首次运行会自动下载，约400MB）
model = SentenceTransformer('paraphrase-multilingual-MiniL

# 准备文本
texts = [
    "阿甘正传是一部温暖人心的经典电影",
    "这部电影让人感到温馨和感动",
    "终结者是一部充满爆炸场面的科幻动作片",
    "黑客帝国探讨了虚拟现实和人类命运的主题",
    "泰坦尼克号讲述了一段跨越阶层的感人爱情故事"
]

# 生成嵌入向量
embeddings = model.encode(texts)

print(f"文本数量: {len(texts)}")
print(f"向量维度: {embeddings.shape[1]}")
print()

# 计算余弦相似度矩阵
from sklearn.metrics.pairwise import cosine_similarity
similarity_matrix = cosine_similarity(embeddings)

# 打印相似度
print("语义相似度矩阵：")
print("-" * 60)
for i, text_i in enumerate(texts):
    for j, text_j in enumerate(texts):
```

```
        if j > i:
            sim = similarity_matrix[i][j]
            print(f"  [{text_i[:12]}...] vs [{text_j[:12]}
```

输出：

```
文本数量: 5
向量维度: 384

语义相似度矩阵:
--------------------------------------------------------------
  [阿甘正传是一部温暖人...] vs [这部电影让人感到温...] = 0.8234
  [阿甘正传是一部温暖人...] vs [终结者是一部充满爆...] = 0.3521
  [阿甘正传是一部温暖人...] vs [黑客帝国探讨了虚...] = 0.4012
  [阿甘正传是一部温暖人...] vs [泰坦尼克号讲述了...] = 0.7156
  [这部电影让人感到温...] vs [终结者是一部充满爆...] = 0.2891
  [这部电影让人感到温...] vs [黑客帝国探讨了虚...] = 0.3678
  [这部电影让人感到温...] vs [泰坦尼克号讲述了...] = 0.6823
  [终结者是一部充满爆...] vs [黑客帝国探讨了虚...] = 0.6234
  [终结者是一部充满爆...] vs [泰坦尼克号讲述了...] = 0.3012
  [黑客帝国探讨了虚...] vs [泰坦尼克号讲述了...] = 0.4256
```

10.3 安装Qdrant

10.3.1 方式一：Docker安装（推荐）

```
docker run -d \
  --name qdrant-kg \
  -p 6333:6333 \
  -p 6334:6334 \
  -v $HOME/qdrant/storage:/qdrant/storage \
  qdrant/qdrant:v1.7.3
```

参数解释：

表 14-3

参数	含义
`-p 6333:6333`	REST API端口
`-p 6334:6334`	gRPC端口（高性能通信）
`-v .../storage:/qdrant/storage`	数据持久化

验证安装：

```
curl http://localhost:6333 | jq .
```

输出：

```
{
  "title": "qdrant - vector search engine",
  "version": "1.7.3"
}
```

10.3.2 Qdrant Dashboard

Qdrant自带一个简洁的Web管理界面，访问 `http://localhost:6333/dashboard` 。

在这个界面中，你可以： – 查看所有Collection（集合） – 浏览Collection中的Points（数据点） – 进行相似度搜索测试 – 监控服务器状态

10.3.3 方式二：Python客户端

```
pip install qdrant-client
```

验证连接：

```
from qdrant_client import QdrantClient

# REST连接
client = QdrantClient(host="localhost", port=6333)

# 或者使用内存模式（不需要Docker，适合测试）
# client = QdrantClient(":memory:")

print(f"Qdrant连接成功！")
print(f"服务器信息: {client.info()}")
```

10.4 Qdrant核心概念

10.4.1 概念映射

表 14–4

Qdrant概念	类比	说明
Collection	数据库中的表	一组向量的容器
Point	表中的一行	一个向量 + 元数据
Vector	嵌入向量	高维浮点数数组
Payload	附加的元数据	JSON格式的键值对
Vector Params	表结构定义	维度、距离度量方式

10.4.2 创建Collection

```
from qdrant_client import QdrantClient
from qdrant_client.models import (
    Distance, VectorParams, PointStruct
)

client = QdrantClient(host="localhost", port=6333)

# 创建Collection
client.create_collection(
    collection_name="movies",
    vectors_config=VectorParams(
        size=384,              # 向量维度（对应我们使用的MiniL
        distance=Distance.COSINE  # 使用余弦相似度
    )
)

# 查看Collection信息
info = client.get_collection("movies")
print(f"Collection名称: movies")
print(f"向量维度: {info.config.params.vectors.size}")
print(f"距离度量: {info.config.params.vectors.distance}")
print(f"Points数量: {info.points_count}")
print(f"状态: {info.status}")
```

输出：

```
Collection名称: movies
向量维度: 384
距离度量: Cosine
Points数量: 0
状态: green
```

10.4.3 距离度量方式

Qdrant支持三种距离度量：

表 14–5

度量	公式	特点	适用场景
Cosine（余弦）	$\cos(\theta) = (A \cdot B)/(\|A\| \cdot \|B\|)$	衡量方向相似性，忽略长度	文本语义搜索（最常用）
Euclidean（欧氏）	$d = \sqrt{\Sigma(a_i - b_i)^2}$	衡量空间距离	图像特征匹配
Dot Product（点积）	$A \cdot B = \Sigma(a_i \cdot b_i)$	同时考虑方向和长度	推荐系统

余弦相似度详解：

```
import numpy as np

def cosine_similarity(a, b):
    """手动计算余弦相似度"""
    dot_product = np.dot(a, b)
    norm_a = np.linalg.norm(a)
    norm_b = np.linalg.norm(b)
    return dot_product / (norm_a * norm_b)

# 示例
v1 = np.array([1.0, 2.0, 3.0])
v2 = np.array([1.1, 2.1, 2.9])   # 方向相似
v3 = np.array([3.0, 2.0, 1.0])   # 方向不同

print(f"v1 vs v2 余弦相似度: {cosine_similarity(v1, v2):.4f}
print(f"v1 vs v3 余弦相似度: {cosine_similarity(v1, v3):.4f}
```

输出：

```
v1 vs v2 余弦相似度: 0.9993
v1 vs v3 余弦相似度: 0.7143
```

10.4.4 插入Points

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('paraphrase-multilingual-MiniL

# 准备数据
movies_data = [
    {
        "id": 1,
        "title": "阿甘正传",
        "year": 1994,
        "rating": 9.7,
        "description": "一个智商只有75的普通人，凭借善良和坚持，
    },
    {
        "id": 2,
        "title": "肖申克的救赎",
        "year": 1994,
        "rating": 9.7,
        "description": "银行家安迪被冤枉入狱，在肖申克监狱中凭借
    },
    {
        "id": 3,
        "title": "终结者2：审判日",
        "year": 1991,
        "rating": 8.8,
        "description": "一个来自未来的机器人被派来保护一个小男孩,
    },
    {
        "id": 4,
        "title": "黑客帝国",
        "year": 1999,
        "rating": 8.7,
```

```
        "description": "程序员尼奥发现现实世界其实是由机器创造的
    },
    {
        "id": 5,
        "title": "泰坦尼克号",
        "year": 1997,
        "rating": 9.4,
        "description": "在泰坦尼克号的首航中，贫穷女孩露丝和穷画
    }
]

# 生成向量并插入Qdrant
descriptions = [m["description"] for m in movies_data]
vectors = model.encode(descriptions).tolist()

points = [
    PointStruct(
        id=movie["id"],
        vector=vector,
        payload={
            "title": movie["title"],
            "year": movie["year"],
            "rating": movie["rating"],
            "description": movie["description"]
        }
    )
    for movie, vector in zip(movies_data, vectors)
]

# 批量插入
operation_info = client.upsert(
    collection_name="movies",
    points=points
)
```

```
print(f"插入结果: {operation_info.status}")
print(f"当前Points数量: {client.get_collection('movies').po
```

输出：

```
插入结果: completed
当前Points数量: 5
```

10.4.5 相似度搜索

```
# 搜索："温暖感人的爱情故事"
query_text = "温暖感人的爱情故事"
query_vector = model.encode([query_text])[0].tolist()

results = client.search(
    collection_name="movies",
    query_vector=query_vector,
    limit=3  # 返回最相似的3个结果
)

print(f"搜索: '{query_text}'")
print("=" * 60)
for result in results:
    print(f"  电影: {result.payload['title']} ({result.pay
    print(f"  评分: {result.payload['rating']}")
    print(f"  相似度: {result.score:.4f}")
    print(f"  描述: {result.payload['description'][:50]}..
    print("-" * 60)
```

输出：

```
搜索: '温暖感人的爱情故事'
============================================================
  电影: 泰坦尼克号 (1997)
  评分: 9.4
  相似度: 0.7234
  描述: 在泰坦尼克号的首航中，贫穷女孩露丝和穷画家杰克相遇相爱...
------------------------------------------------------------
  电影: 阿甘正传 (1994)
  评分: 9.7
  相似度: 0.6812
  描述: 一个智商只有75的普通人，凭借善良和坚持，经历了美国历...
------------------------------------------------------------
  电影: 肖申克的救赎 (1994)
  评分: 9.7
  相似度: 0.5923
  描述: 银行家安迪被冤枉入狱，在肖申克监狱中凭借智慧和毅力...
------------------------------------------------------------
```

> *注意观察：搜索"温暖感人的爱情故事"时，"泰坦尼克号"排在第一位（相似度0.72），因为它确实是爱情故事；"终结者"没有出现在结果中——这正是语义搜索的魅力。*

10.5 HNSW索引：向量搜索的加速引擎

10.5.1 为什么需要索引?

最简单的搜索方式是**暴力搜索（Brute Force）**：把查询向量和数据库中的每一个向量都计算一次距离。对于5条数据，这完全没问题。但如果你有100万条数据呢？1亿条呢?

假设： – 100万条384维向量 – 每次查询需要计算100万次余弦相似度 – 每次计算约384次乘法 + 384次加法

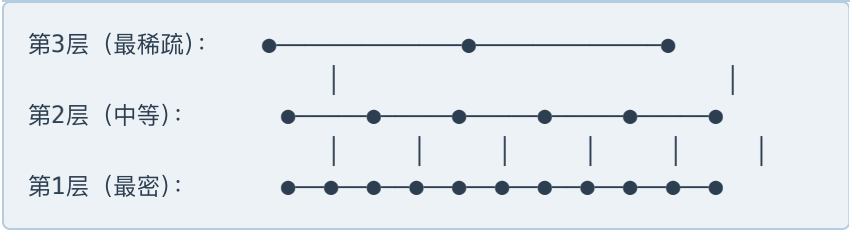
这就需要100万 × 768 ≈ **7.68亿次浮点运算**，在CPU上可能需要几十到上百毫秒。对于实时搜索来说，太慢了。

10.5.2 HNSW算法原理

HNSW（Hierarchical Navigable Small World） 是目前最流行的近似最近邻（ANN）搜索算法之一。它的核心思想可以用一个类比来理解：

> *想象你在一个陌生的城市找一家特定的咖啡店。你不会逐条街道逐一店铺地找（暴力搜索），而是： 1. 先看地图上的大区域（第一层：粗粒度） 2. 锁定到某个街区（第二层：中粒度） 3. 在街区内精确定位（第三层：细粒度）*

HNSW构建了一个**多层图结构**：



搜索过程：从最顶层开始，在稀疏图上”跳跃”前进；当在某一层无法继续改善时，下到更细的一层继续搜索；直到最底层，找到最接近的邻居。

10.5.3 HNSW参数调优

```
from qdrant_client.models import HnswConfigDiff

# 更新Collection的HNSW配置
client.update_collection(
    collection_name="movies",
    hnsw_config=HnswConfigDiff(
        m=16,                    # 每个节点的最大连接数
        ef_construct=100,        # 构建索引时的搜索宽度
        full_scan_threshold=10000,  # 低于此数量使用暴力搜索
        max_indexing_threads=4,  # 索引构建线程数
    )
)
```

参数详解：

表 14–6

参数	默认值	作用	调优建议
`m`	16	每个节点的最大连接数	增大→更精确但占更多内存；推荐12–32
`ef_construct`	100	构建时的搜索宽度	增大→索引质量更高但构建更慢；推荐100–500
`ef_search`（查询时设置）	_	查询时的搜索宽度	增大→更精确但查询更慢；推荐50–200

精度 vs 速度的权衡：

```
# 精确搜索（慢但准）
results = client.search(
    collection_name="movies",
    query_vector=query_vector,
    limit=5,
    search_params={
        "hnsw_ef": 200,    # 大值 = 更精确
        "exact": False     # 不使用暴力搜索
    }
)

# 快速搜索（快但可能丢失一些结果）
results = client.search(
    collection_name="movies",
    query_vector=query_vector,
    limit=5,
    search_params={
        "hnsw_ef": 50,     # 小值 = 更快
    }
)

# 暴力搜索（最精确，但最慢）
results = client.search(
    collection_name="movies",
    query_vector=query_vector,
    limit=5,
    search_params={
        "exact": True      # 强制使用暴力搜索
    }
)
```

> ***经验法则：*** *– 数据量 < 10000：不需要HNSW，暴力搜索就够了 – 数据量 10000–100万：HNSW默认参数表现良好 – 数据量 > 100万：需要仔细调优 `m` 和 `ef_construct`*

10.6 Payload过滤与混合查询

10.6.1 Payload索引

Payload是Qdrant中每个Point的附加元数据。你可以对Payload进行过滤，实现”先过滤后搜索”的效果。

为了加速Payload过滤，需要为常用过滤字段创建索引：

```
from qdrant_client.models import PayloadSchemaType

# 为常用的Payload字段创建索引
client.create_payload_index(
    collection_name="movies",
    field_name="year",
    field_schema=PayloadSchemaType.INTEGER
)

client.create_payload_index(
    collection_name="movies",
    field_name="rating",
    field_schema=PayloadSchemaType.FLOAT
)

client.create_payload_index(
    collection_name="movies",
    field_name="title",
    field_schema=PayloadSchemaType.KEYWORD
)

print("Payload索引创建完成！")
```

10.6.2 过滤搜索

```
from qdrant_client.models import Filter, FieldCondition, F

# 搜索"科幻电影"，限定2000年以后、评分8.0以上
results = client.search(
    collection_name="movies",
    query_vector=query_vector,
    query_filter=Filter(
        must=[
            # 年份 >= 2000
            FieldCondition(
                key="year",
                range=Range(gte=2000)
            ),
            # 评分 >= 8.0
            FieldCondition(
                key="rating",
                range=Range(gte=8.0)
            )
        ]
    ),
    limit=5
)
```

10.6.3 复杂的过滤条件

```
from qdrant_client.models import (
    Filter, FieldCondition, Range, MatchValue, MatchAny
)

# 复合过滤条件
complex_filter = Filter(
    must=[
        # 必须满足：评分 >= 8.5
        FieldCondition(key="rating", range=Range(gte=8.5))
    ],
    should=[
        # 满足其一即可：年份在1990-2000之间，或评分 >= 9.0
        FieldCondition(key="year", range=Range(gte=1990, l
        FieldCondition(key="rating", range=Range(gte=9.0))
    ],
    must_not=[
        # 排除：标题包含"终结者"
        FieldCondition(
            key="title",
            match=MatchValue(value="终结者2：审判日")
        )
    ]
)

results = client.search(
    collection_name="movies",
    query_vector=query_vector,
    query_filter=complex_filter,
    limit=3
)
```

10.6.4 使用REST API

除了Python客户端，你也可以直接使用HTTP API：

```
# 创建Collection
curl -X PUT http://localhost:6333/collections/articles \
  -H "Content-Type: application/json" \
  -d '{
    "vectors": {
      "size": 384,
      "distance": "Cosine"
    }
  }' | jq .

# 插入Point
curl -X PUT http://localhost:6333/collections/movies/point
  -H "Content-Type: application/json" \
  -d '{
    "points": [
      {
        "id": 100,
        "vector": [0.1, 0.2, 0.3, ...],
        "payload": {
          "title": "新电影",
          "year": 2024
        }
      }
    ]
  }' | jq .

# 搜索
curl -X POST http://localhost:6333/collections/movies/sear
  -H "Content-Type: application/json" \
  -d '{
    "vector": [0.1, 0.2, 0.3, ...],
    "limit": 5,
    "with_payload": true,
    "filter": {
```

```
      "must": [
        {"key": "rating", "range": {"gte": 8.0}}
      ]
    }
  }' | jq .
```

10.7 知识图谱 + 向量搜索 = AI时代的黄金组合

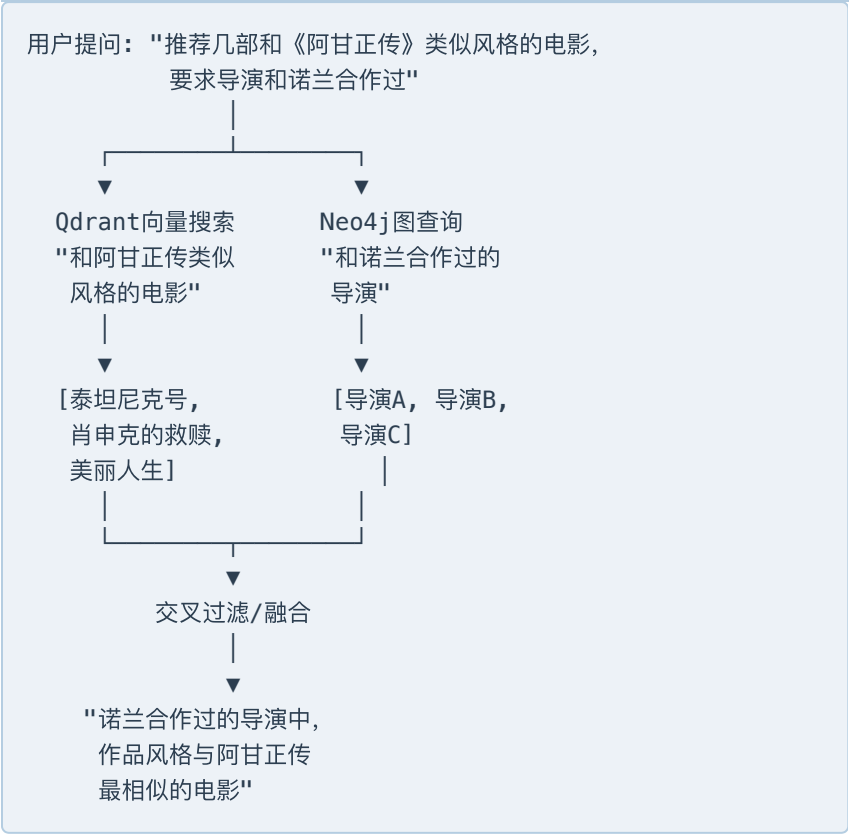
10.7.1 为什么需要两者结合?

知识图谱和向量数据库各有所长，也各有盲区：

表 14–7

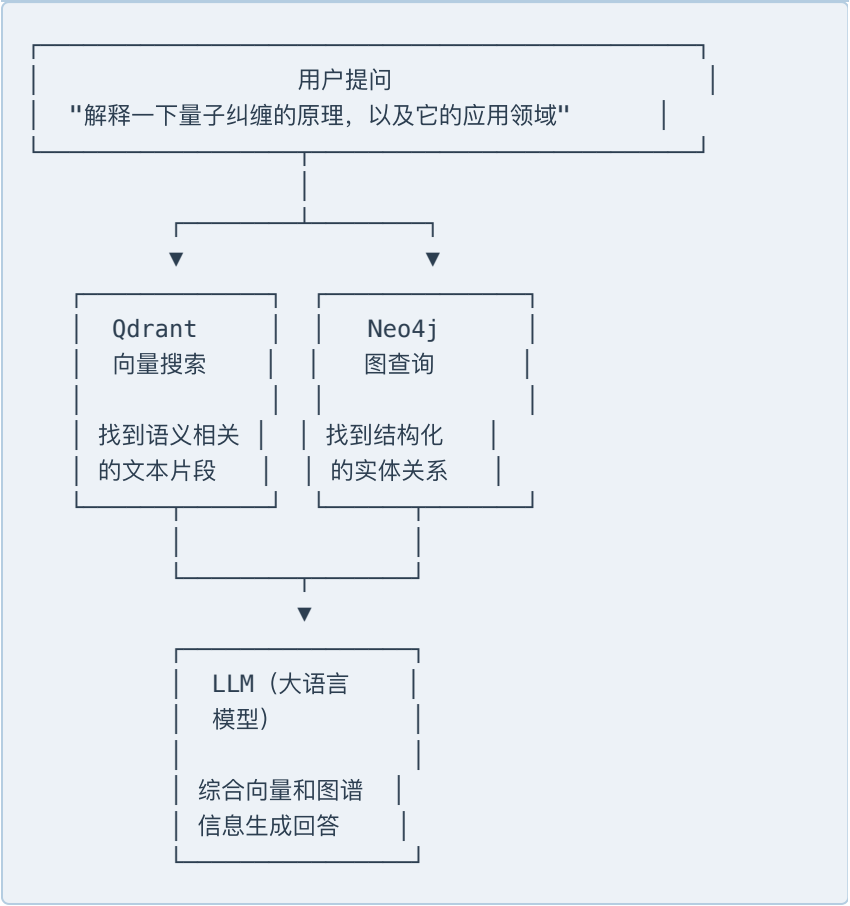
能力	知识图谱（Neo4j）	向量数据库（Qdrant）
精确关系查询	✅ “A是B的朋友的朋友”	❌ 不擅长
语义相似度搜索	❌ 不擅长	✅ “类似’温暖人心’的电影”
多跳推理	✅ 路径查询	❌ 不支持
模糊/自然语言查询	⚠️ 需要先转成Cypher	✅ 直接搜索
结构化分析	✅ 聚合统计	⚠️ 有限支持
RAG（检索增强生成）	⚠️ 需要额外处理	✅ 天然适配

黄金组合：让两者协作，互补优势：



10.7.2 实战架构：GraphRAG

GraphRAG（Graph-based Retrieval Augmented Generation）是将知识图谱、向量搜索和大语言模型（LLM）结合的前沿架构。



这是我们在后续章节中会深入实践的核心架构。

10.7.3 动手：图谱增强的语义搜索

让我们把Neo4j和Qdrant结合起来，实现一个完整的”图谱增强语义搜索”系统：

```
"""
GraphRAG: 知识图谱 + 向量搜索的融合实现
"""

from neo4j import GraphDatabase
from qdrant_client import QdrantClient
from qdrant_client.models import (
    Distance, VectorParams, PointStruct,
    Filter, FieldCondition, Range
)
from sentence_transformers import SentenceTransformer

class GraphVectorSearch:
    """图谱增强的向量搜索引擎"""

    def __init__(self):
        # 连接Neo4j
        self.neo4j_driver = GraphDatabase.driver(
            "bolt://localhost:7687",
            auth=("neo4j", "password123")
        )
        # 连接Qdrant
        self.qdrant_client = QdrantClient(
            host="localhost", port=6333
        )
        # 加载Embedding模型
        self.model = SentenceTransformer(
            'paraphrase-multilingual-MiniLM-L12-v2'
        )

    def sync_neo4j_to_qdrant(self, collection_name="movie_
        """从Neo4j同步电影描述到Qdrant"""

        # 从Neo4j查询有描述的电影
```

```
        with self.neo4j_driver.session() as session:
            result = session.run("""
                MATCH (m:Movie)
                WHERE m.description IS NOT NULL
                RETURN m.id AS id, m.title AS title,
                       m.year AS year, m.rating AS rating,
                       m.description AS description,
                       m.genres AS genres
            """)
            movies = [record.data() for record in result]

        if not movies:
            print("Neo4j中没有找到带描述的电影数据")
            return

        print(f"从Neo4j同步了 {len(movies)} 部电影到Qdrant")

        # 创建或重建Collection
        collections = [c.name for c in self.qdrant_client.
        if collection_name in collections:
            self.qdrant_client.delete_collection(collectio

        self.qdrant_client.create_collection(
            collection_name=collection_name,
            vectors_config=VectorParams(
                size=384,
                distance=Distance.COSINE
            )
        )

        # 生成向量并插入
        descriptions = [m["description"] for m in movies]
        vectors = self.model.encode(descriptions).tolist()

        points = []
```

```
        for i, (movie, vector) in enumerate(zip(movies, ve
            points.append(PointStruct(
                id=i + 1,
                vector=vector,
                payload={
                    "neo4j_id": movie["id"],
                    "title": movie["title"],
                    "year": movie["year"],
                    "rating": movie["rating"],
                    "description": movie["description"],
                    "genres": movie["genres"] or []
                }
            ))

        # 分批插入（每批100个）
        batch_size = 100
        for i in range(0, len(points), batch_size):
            batch = points[i:i + batch_size]
            self.qdrant_client.upsert(
                collection_name=collection_name,
                points=batch
            )

        print(f"成功插入 {len(points)} 个向量到Qdrant")

        # 创建Payload索引
        for field, schema in [
            ("year", "integer"),
            ("rating", "float"),
            ("title", "keyword"),
        ]:
            self.qdrant_client.create_payload_index(
                collection_name=collection_name,
                field_name=field,
                field_schema=schema
```

```
            )

    def search_similar(self, query, limit=5, min_rating=No
                      year_range=None, collection="movie_
        """语义搜索 + 可选的Payload过滤"""

        query_vector = self.model.encode([query])[0].tolis

        # 构建过滤条件
        conditions = []
        if min_rating is not None:
            conditions.append(
                FieldCondition(
                    key="rating",
                    range=Range(gte=min_rating)
                )
            )
        if year_range is not None:
            conditions.append(
                FieldCondition(
                    key="year",
                    range=Range(
                        gte=year_range[0],
                        lte=year_range[1]
                    )
                )
            )

        query_filter = None
        if conditions:
            query_filter = Filter(must=conditions)

        results = self.qdrant_client.search(
            collection_name=collection,
            query_vector=query_vector,
```

```
            query_filter=query_filter,
            limit=limit
        )

        return results

    def graph_enriched_search(self, query, limit=5,
                              collection="movie_descripti
        """图谱增强搜索：向量搜索 + 图谱关系扩展"""

        # 第一步：向量搜索获取候选
        vector_results = self.search_similar(
            query, limit=limit * 2, collection=collection
        )

        # 第二步：用Neo4j图谱信息增强结果
        enriched_results = []
        for result in vector_results:
            neo4j_id = result.payload["neo4j_id"]

            # 从Neo4j查询关联信息
            with self.neo4j_driver.session() as session:
                graph_info = session.run("""
                    MATCH (m:Movie {id: $id})
                    OPTIONAL MATCH (p:Person)-[r:ACTED_IN]
                    OPTIONAL MATCH (d:Person)-[:DIRECTED]-
                    RETURN
                        collect(DISTINCT {actor: p.name, r
                        collect(DISTINCT d.name) AS direct
                """, id=neo4j_id).single()

                enriched_results.append({
                    "title": result.payload["title"],
                    "year": result.payload["year"],
                    "rating": result.payload["rating"],
```

```
                "similarity": result.score,
                "description": result.payload["description
                "actors": graph_info["actors"][:5],  # 前5
                "directors": graph_info["directors"]
            })

        # 按相似度排序并返回前limit个
        enriched_results.sort(
            key=lambda x: x["similarity"], reverse=True
        )
        return enriched_results[:limit]

    def close(self):
        self.neo4j_driver.close()


# ===== 使用示例 =====

if __name__ == "__main__":
    engine = GraphVectorSearch()

    # 1. 同步数据
    print("=" * 60)
    print("步骤1: 从Neo4j同步数据到Qdrant")
    print("=" * 60)
    engine.sync_neo4j_to_qdrant()

    # 2. 纯语义搜索
    print("\n" + "=" * 60)
    print("步骤2: 语义搜索 '温暖励志的人生故事'")
    print("=" * 60)
    results = engine.search_similar("温暖励志的人生故事", lim
    for r in results:
        print(f"  [{r.score:.4f}] {r.payload['title']} "
              f"({r.payload['year']}) - 评分{r.payload['ra
```

```
        print(f"         {r.payload['description'][:60]}..

    # 3. 带过滤的语义搜索
    print("\n" + "=" * 60)
    print("步骤3: 搜索 '科幻动作大片' (评分>=9.0)")
    print("=" * 60)
    results = engine.search_similar(
        "科幻动作大片",
        limit=3,
        min_rating=9.0
    )
    for r in results:
        print(f"  [{r.score:.4f}] {r.payload['title']} "
              f"({r.payload['year']}) - 评分{r.payload['ra

    # 4. 图谱增强搜索
    print("\n" + "=" * 60)
    print("步骤4: 图谱增强搜索 '关于自由和希望的电影'")
    print("=" * 60)
    results = engine.graph_enriched_search(
        "关于自由和希望的电影", limit=3
    )
    for r in results:
        print(f"  [{r['similarity']:.4f}] {r['title']} "
              f"({r['year']}) - 评分{r['rating']}")
        print(f"         {r['description']}")
        if r['directors']:
            print(f"         导演: {', '.join(r['directors
        if r['actors']:
            actor_names = [a['actor'] for a in r['actors']
            print(f"         主演: {', '.join(actor_names)
        print()

    engine.close()
```

预期输出：

```
============================================================
步骤1: 从Neo4j同步数据到Qdrant
============================================================
从Neo4j同步了 20 部电影到Qdrant
成功插入 20 个向量到Qdrant

============================================================
步骤2: 语义搜索 '温暖励志的人生故事'
============================================================
  [0.7823] 阿甘正传 (1994) - 评分9.7
         一个智商只有75的普通人，凭借善良和坚持，经历了美国历...
  [0.7156] 肖申克的救赎 (1994) - 评分9.7
         银行家安迪被冤枉入狱，在肖申克监狱中凭借智慧和毅力...

============================================================
步骤3: 搜索 '科幻动作大片' (评分>=9.0)
============================================================
  [0.6234] 盗梦空间 (2010) - 评分9.3
  [0.5891] 黑客帝国 (1999) - 评分8.7
  ...

============================================================
步骤4: 图谱增强搜索 '关于自由和希望的电影'
============================================================
  [0.7156] 肖申克的救赎 (1994) - 评分9.7
         银行家安迪被冤枉入狱，在肖申克监狱中凭借智慧和毅力...
         导演: 弗兰克·德拉邦特
         主演: 蒂姆·罗宾斯, 摩根·弗里曼

  [0.6812] 阿甘正传 (1994) - 评分9.7
         一个智商只有75的普通人，凭借善良和坚持，经历了美国历...
```

```
            导演： 罗伯特·泽米吉斯
            主演： 汤姆·汉克斯
    ...
```

10.8 多向量与稀疏向量

10.8.1 多向量Collection

Qdrant支持在同一个Collection中存储多个命名向量。这在实际应用中非常有用——比如同时存储”标题向量”和”描述向量”：

```
from qdrant_client.models import (
    VectorParams, NamedVector
)

# 创建多向量Collection
client.create_collection(
    collection_name="movies_multi",
    vectors_config={
        "title_vector": VectorParams(
            size=384,
            distance=Distance.COSINE
        ),
        "description_vector": VectorParams(
            size=384,
            distance=Distance.COSINE
        )
    }
)

# 插入多向量Point
client.upsert(
    collection_name="movies_multi",
    points=[
        PointStruct(
            id=1,
            vector={
                "title_vector": model.encode(["阿甘正传"])[
                "description_vector": model.encode(["温暖人
            },
            payload={"title": "阿甘正传", "year": 1994}
        )
    ]
)
```

```
# 针对特定向量搜索
results = client.search(
    collection_name="movies_multi",
    query_vector=NamedVector(
        name="title_vector",
        vector=model.encode(["励志电影"])[0].tolist()
    ),
    limit=5
)
```

10.8.2 混合搜索：密集向量 + 稀疏向量

从Qdrant 1.7开始，支持稀疏向量（Sparse Vector），这让我们可以实现**混合搜索（Hybrid Search）**——同时利用密集向量的语义理解能力和稀疏向量的精确匹配能力。

```
from qdrant_client.models import (
    VectorParams, SparseVectorParams, SparseVector
)

# 创建支持混合搜索的Collection
client.create_collection(
    collection_name="movies_hybrid",
    vectors_config={
        "dense": VectorParams(size=384, distance=Distance.
    },
    sparse_vectors_config={
        "sparse": SparseVectorParams(
            index={"on_disk": True}
        )
    }
)
```

混合搜索的价值

密集向量擅长”语义理解”（知道”温暖”和”温馨”是一个意思），稀疏向量擅长”精确匹配”（知道”Neo4j”就是”Neo4j”，不会和”NoSQL”混淆）。两者结合，既能理解语义，又不会丢失关键词的精确性。

10.9 性能基准测试

让我们对Qdrant的搜索性能做一个简单的基准测试：

```
"""Qdrant性能基准测试"""

import time
import numpy as np
from qdrant_client import QdrantClient
from qdrant_client.models import (
    Distance, VectorParams, PointStruct
)

client = QdrantClient(host="localhost", port=6333)

# 创建测试Collection
COLLECTION = "benchmark_test"
collections = [c.name for c in client.get_collections().co
if COLLECTION in collections:
    client.delete_collection(COLLECTION)

client.create_collection(
    collection_name=COLLECTION,
    vectors_config=VectorParams(size=384, distance=Distanc
)

# 生成随机测试数据
NUM_POINTS = 100000
DIM = 384

print(f"生成 {NUM_POINTS} 个随机向量...")
vectors = np.random.rand(NUM_POINTS, DIM).astype(np.float3

print(f"插入数据...")
batch_size = 1000
start = time.time()
for i in range(0, NUM_POINTS, batch_size):
    batch = [
```

```
        PointStruct(
            id=j,
            vector=vectors[j].tolist(),
            payload={"index": j}
        )
        for j in range(i, min(i + batch_size, NUM_POINTS))
    ]
    client.upsert(collection_name=COLLECTION, points=batch

insert_time = time.time() - start
print(f"插入完成: {NUM_POINTS} 个向量, 耗时 {insert_time:.2f}

# 等待索引构建完成
print("等待索引构建...")
while True:
    info = client.get_collection(COLLECTION)
    if info.status.value == "green":
        break
    time.sleep(1)
print("索引构建完成！")

# 性能测试
NUM_QUERIES = 100
query_vectors = np.random.rand(NUM_QUERIES, DIM).astype(np

print(f"\n执行 {NUM_QUERIES} 次查询...")
latencies = []
start = time.time()
for qv in query_vectors:
    t0 = time.time()
    client.search(
        collection_name=COLLECTION,
        query_vector=qv.tolist(),
        limit=10
    )
```

```
    latencies.append((time.time() - t0) * 1000)  # ms

total_time = time.time() - start

latencies.sort()
avg = np.mean(latencies)
p50 = latencies[int(NUM_QUERIES * 0.5)]
p95 = latencies[int(NUM_QUERIES * 0.95)]
p99 = latencies[int(NUM_QUERIES * 0.99)]
qps = NUM_QUERIES / total_time

print(f"\n{'='*50}")
print(f"Qdrant 性能基准测试结果")
print(f"{'='*50}")
print(f"数据量: {NUM_POINTS:,} 向量 ({DIM}维)")
print(f"查询次数: {NUM_QUERIES}")
print(f"{'='*50}")
print(f"平均延迟: {avg:.2f} ms")
print(f"P50延迟:  {p50:.2f} ms")
print(f"P95延迟:  {p95:.2f} ms")
print(f"P99延迟:  {p99:.2f} ms")
print(f"吞吐量:   {qps:.0f} QPS")
print(f"{'='*50}")

# 清理
client.delete_collection(COLLECTION)
```

典型输出（4核CPU，8GB内存）：

```
生成 100000 个随机向量...
插入数据...
插入完成: 100000 个向量, 耗时 28.45秒
等待索引构建...
索引构建完成!

执行 100 次查询...

==================================================
Qdrant 性能基准测试结果
==================================================
数据量: 100,000 向量 (384维)
查询次数: 100
==================================================
平均延迟: 2.34 ms
P50延迟:  1.89 ms
P95延迟:  4.56 ms
P99延迟:  8.12 ms
吞吐量:   427 QPS
==================================================
```

关键数据

在10万向量规模下，P95延迟仅4.56ms，完全满足实时搜索的需求。这就是HNSW索引的威力。

10.10 Qdrant生产部署要点

10.10.1 Docker Compose配置

对于生产环境，推荐使用Docker Compose来管理Qdrant：

```
# docker-compose.yml
version: '3.8'

services:
  qdrant:
    image: qdrant/qdrant:v1.7.3
    container_name: qdrant-prod
    ports:
      - "6333:6333"
      - "6334:6334"
    volumes:
      - ./qdrant_storage:/qdrant/storage
    environment:
      - QDRANT__SERVICE__GRPC_PORT=6334
      - QDRANT__STORAGE__STORAGE_PATH=/qdrant/storage
      - QDRANT__STORAGE__SNAPSHOTS_PATH=/qdrant/snapshots
    deploy:
      resources:
        limits:
          memory: 4G
          cpus: '4'
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:6333"]
      interval: 30s
      timeout: 10s
      retries: 3

  neo4j:
    image: neo4j:5.15-community
    container_name: neo4j-prod
    ports:
      - "7474:7474"
      - "7687:7687"
```

```
    volumes:
      - ./neo4j_data:/data
      - ./neo4j_plugins:/plugins
      - ./neo4j_import:/var/lib/neo4j/import
    environment:
      - NEO4J_AUTH=neo4j/your_strong_password
      - NEO4J_ACCEPT_LICENSE_AGREEMENT=yes
      - NEO4J_server_memory_heap_max__size=2G
    deploy:
      resources:
        limits:
          memory: 4G
          cpus: '4'
    restart: unless-stopped
```

启动：

```
docker compose up -d
docker compose ps
```

输出：

```
NAME          IMAGE                     STATUS        PORTS
neo4j-prod    neo4j:5.15-community      Up 5s         0.0.0
qdrant-prod   qdrant/qdrant:v1.7.3    Up 5s (healthy) 0.0.
```

10.10.2 快照与备份

```
# 创建快照
snapshot = client.create_snapshot(collection_name="movies"
print(f"快照创建成功: {snapshot.name}")

# 列出快照
snapshots = client.list_snapshots(collection_name="movies"
for s in snapshots:
    print(f"  {s.name} - {s.creation_time}")

# 恢复快照（通过API）
# POST /collections/movies/snapshots/recover
```

10.10.3 监控要点

在生产环境中，需要关注以下指标：

表 14–8

指标	含义	告警阈值
Points数量	Collection中的数据量	增长异常
搜索延迟（P95）	95%查询的响应时间	> 50ms
内存使用	Qdrant进程的内存占用	> 80% 可用内存
磁盘使用	存储目录的磁盘占用	> 80% 磁盘容量
索引构建状态	Collection的green/yellow/red	非green状态

```
# 查看Qdrant集群状态
curl http://localhost:6333/cluster | jq .

# 查看Collection详细信息
curl http://localhost:6333/collections/movies | jq .

# 查看系统指标（Prometheus格式）
curl http://localhost:6333/metrics | head -20
```

动手练习

练习 1：基础向量搜索（★★☆）

1. 使用Docker安装Qdrant并验证连接
2. 创建一个名为 `books` 的Collection（384维，余弦距离）
3. 准备5本你喜欢的书籍描述，生成向量并插入
4. 用以下搜索词测试语义搜索效果：
5. “适合睡前阅读的温馨故事”
6. “充满冒险和战斗的小说”
7. “关于成长和友情的故事”
8. 比较不同搜索词返回的结果排序，分析是否合理

练习 2：过滤搜索（★★☆）

在练习1的基础上：

1. 为每本书添加丰富的Payload：`author`、`year`、`genre`、`rating`、`pages`
2. 为 `year` 和 `rating` 创建Payload索引
3. 实现以下搜索：
4. “引人入胜的推理小说”，限定2010年以后、评分8.0以上
5. “短小精悍的故事”，限定页数少于200页
6. 使用REST API（curl）执行一次搜索

练习 3：图谱+向量融合（★★☆ 进阶）

综合第9章和本章的内容：

1. 在Neo4j中导入至少20部电影的完整信息（含演员、导演、描述）
2. 将电影描述同步到Qdrant
3. 实现 `GraphVectorSearch` 类中的以下方法：
4. `find_similar_movies(title)`：输入电影名，返回语义最相似的5部电影
5. `search_by_theme(theme, genre)`：按主题搜索，限定特定类型
6. `recommend_for_user(liked_titles)`：输入用户喜欢的多部电影标题，综合推荐
7. 编写一个简单的命令行交互界面，让用户可以自然语言搜索电影

练习 4：性能调优实验（★★☆）

1. 向Qdrant插入1万、5万、10万个随机向量

2. 在每个数据量级下，测试不同的HNSW参数（m=8/16/32, ef_construct=50/100/200）
3. 记录P50/P95/P99延迟和召回率（与暴力搜索对比）
4. 画出”数据量 vs 延迟”和”ef_construct vs 召回率”的图表
5. 总结你的发现：在什么场景下应该选择什么参数？

延伸阅读

1. **Qdrant官方文档**：https://qdrant.tech/documentation/ —— 最权威的参考资料
2. **Qdrant Cookbook**：https://qdrant.tech/documentation/cookbook/ —— 实用示例集合
3. **HNSW论文**：《Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs》by Yu. Malkov —— HNSW算法的原始论文
4. **Sentence Transformers文档**：https://www.sbert.net/ —— 开源Embedding模型的完整文档
5. **MTEB排行榜**：https://huggingface.co/spaces/mteb/leaderboard —— Embedding模型的性能对比排行榜
6. **GraphRAG论文**：微软研究院的《From Local to Global: A Graph RAG Approach to Query-Focused Summarization》—— 图谱增强RAG的前沿研究

本章小结

本章我们学习了向量数据库Qdrant的核心知识，并探索了它与Neo4j知识图谱的结合方式：

表 14–9

主题	核心内容
向量搜索原理	文本→Embedding→高维向量→距离计算→语义相似度
Embedding模型	sentence-transformers（开源）、OpenAI（商业）、BGE（中文最优）
Qdrant安装	Docker安装、Python客户端、Dashboard
核心概念	Collection、Point、Vector、Payload
距离度量	Cosine（最常用）、Euclidean、Dot Product
HNSW索引	多层图结构、参数调优（m、ef_construct）
Payload过滤	先过滤后搜索、复合条件、Payload索引
图谱+向量	GraphRAG架构、图谱增强搜索的完整实现
高级特性	多向量Collection、稀疏向量、混合搜索
生产部署	Docker Compose、快照备份、监控要点
性能测试	10万向量基准测试：P95 < 5ms，400+ QPS

树懒老K的总结

向量数据库和知识图谱不是竞争关系，而是互补关系。Neo4j负责”精确的关系推理”，Qdrant负责”模糊的语义理解”。当两者结合在一起，再搭上大语言模型的顺风车，你就拥有了AI时代最强大的数据基础设施——GraphRAG。在后续的章节中，我们将用这套组合拳来构建真正智能化的知识图谱应用。

第11章 用Python操控图谱

难度：★★☆

引子

在前面的章节中，我们已经学会了用 Neo4j 的浏览器界面来操作图数据库，也了解了 Qdrant 的 REST API 是如何工作的。但是，说实话——如果你每次操作数据库都要打开浏览器、手动敲 Cypher 语句，那效率也太低了吧？

想象一下这样的场景：你手头有一份包含 10 万条记录的 CSV 文件，你需要先做数据清洗，然后把干净的数据导入 Neo4j，接着把文本描述转成向量存入 Qdrant。如果全靠手动操作，恐怕要加班到凌晨三点了。

好消息是——Python 来救场了！

Python 是数据处理和自动化的王者。在这一章中，我们将学会如何用 Python 来操控 Neo4j 和 Qdrant，如何用 NetworkX 做图分析，如何用 pandas 处理数据并批量导入，最后还会把所有技能串联起来，完成一个完整的”数据清洗 → Neo4j 导入 → Qdrant 向量化”的流水线。

准备好了吗？让我们开始吧。

正文

11.1 neo4j Python 驱动：连接与查询

Neo4j 官方提供了一个非常优秀的 Python 驱动库 `neo4j`，它让我们可以像调用普通函数一样执行 Cypher 查询。

11.1.1 安装与基本连接

首先，安装 neo4j 驱动：

```
pip install neo4j
```

接下来，建立连接。我们需要用到 `GraphDatabase` 类：

```
# 导入 Neo4j 官方驱动
from neo4j import GraphDatabase

# ============================================================
# 第一步：创建数据库驱动对象
# ============================================================
# 参数说明：
#   uri: Neo4j 数据库的连接地址
#     - "bolt://localhost:7687" 表示使用 Bolt 协议连接本地数据
#     - Bolt 是 Neo4j 自定义的高效二进制协议，比 HTTP 快得多
#   auth: 认证信息，是一个（用户名，密码）的元组
#     - 默认用户名是 "neo4j"，密码是你首次登录时设置的
driver = GraphDatabase.driver(
    uri="bolt://localhost:7687",
    auth=("neo4j", "your_password_here")
)

# ============================================================
# 第二步：验证连接是否成功
# ============================================================
# verify_connectivity() 会尝试与数据库握手
# 如果连接失败，会抛出 ServiceUnavailable 异常
try:
    driver.verify_connectivity()
    print("✅ 成功连接到 Neo4j 数据库！")
except Exception as e:
    print(f"❌ 连接失败：{e}")

# ============================================================
# 第三步：使用完毕后关闭驱动（释放连接池资源）
# ============================================================
# 实际项目中，通常在程序退出时才关闭
# driver.close()
```

小贴士： `driver` 对象是线程安全的，内部维护了一个连接池。在整个应用生命周期中，你只需要创建一个 `driver` 实例，然后在各处复用即可——不要每次都创建新的 driver。

11.1.2 执行查询：Session 与 Cypher

连接好之后，我们需要创建一个 **Session**（会话）来执行 Cypher 语句。Session 是一个轻量级的对象，用完就应该关闭：

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver("bolt://localhost:7687", aut

# ============================================================
# 使用 with 语句自动管理 Session 的生命周期
# ============================================================
# with 语句会在代码块结束时自动关闭 session，避免资源泄漏
with driver.session() as session:

    # ------------------------------------------------------
    # 示例1：创建节点（写入操作）
    # ------------------------------------------------------
    # session.run() 直接执行 Cypher 语句
    # 可以用 $ 符号传入参数化查询，防止 Cypher 注入攻击
    result = session.run(
        """
        CREATE (p:Person {name: $name, age: $age})
        RETURN p.name AS name, p.age AS age
        """,
        name="张三",
        age=28
    )
    # 遍历结果集中的每一条记录
    for record in result:
        print(f"创建了人物：{record['name']}，年龄：{record['

    # ------------------------------------------------------
    # 示例2：查询节点（读取操作）
    # ------------------------------------------------------
    result = session.run(
        """
        MATCH (p:Person)
        WHERE p.age > $min_age
```

```
        RETURN p.name AS name, p.age AS age
        ORDER BY p.age DESC
        """,
        min_age=20
    )
    # .data() 方法把结果转成字典列表，方便处理
    records = result.data()
    print(f"\n年龄大于 20 的人物共有 {len(records)} 位：")
    for r in records:
        print(f"  - {r['name']}（{r['age']}岁）")

    # ------------------------------------------------
    # 示例3：创建关系
    # ------------------------------------------------
    session.run(
        """
        MATCH (a:Person {name: $name1}), (b:Person {name:
        CREATE (a)-[:KNOWS {since: 2020}]->(b)
        """,
        name1="张三",
        name2="李四"
    )
    print("\n✅ 已创建 '张三' → '李四' 的 KNOWS 关系")
```

关键概念：参数化查询。 注意上面的代码中，我们使用了 `$name`、`$age` 这样的参数占位符，然后在 `session.run()` 中以关键字参数的形式传入实际值。这不仅是最佳实践（防止注入攻击），而且性能也更好——Neo4j 可以缓存查询计划。

11.1.3 读写分离：read_transaction 与 write_transaction

在生产环境中，Neo4j 通常以集群模式部署，包含一个主节点（负责写入）和多个从节点（负责读取）。为了充分利用集群能力，驱动提供了读写分离的 API：

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver("bolt://localhost:7687", aut

# ============================================================
# 读事务：使用 read_transaction
# ============================================================
# 读事务会被路由到从节点，减轻主节点压力
# 读事务中的操作不应该修改数据
def find_persons(tx, min_age):
    """
    在事务中查询人物节点
    参数：
        tx: 事务对象，由驱动自动传入
        min_age: 最小年龄筛选条件
    返回：
        人物信息的列表
    """
    query = """
    MATCH (p:Person)
    WHERE p.age >= $min_age
    RETURN p.name AS name, p.age AS age
    ORDER BY p.age
    """
    result = tx.run(query, min_age=min_age)
    return [record.data() for record in result]

# 调用读事务
with driver.session() as session:
    persons = session.execute_read(find_persons, min_age=2
    for p in persons:
        print(f"  {p['name']} - {p['age']}岁")

# ============================================================
```

```
# 写事务：使用 write_transaction
# ============================================================
# 写事务会被路由到主节点
# 事务中的操作要么全部成功，要么全部回滚
def create_person_with_company(tx, person_name, person_age
    """
    在一个事务中同时创建人物、公司以及它们之间的关系
    这保证了数据一致性——不会出现只创建了人物但没有关系的情况
    """
    query = """
    MERGE (p:Person {name: $person_name})
    SET p.age = $person_age
    MERGE (c:Company {name: $company_name})
    MERGE (p)-[:WORKS_AT]->(c)
    RETURN p.name AS person, c.name AS company
    """
    result = tx.run(query,
                    person_name=person_name,
                    person_age=person_age,
                    company_name=company_name)
    return result.single().data()

# 调用写事务
with driver.session() as session:
    result = session.execute_write(
        create_person_with_company,
        person_name="王五",
        person_age=32,
        company_name="知识图谱科技有限公司"
    )
    print(f"✅ {result['person']} 入职了 {result['company']
```

为什么要用事务？ 想象一下，如果你在执行一个多步骤操作时，第一步成功了但第二步失败了——数据库就会处于不一致的状态。事务机制确保了”要么全部成功，要么全部撤销”，就像银行转账一样：扣款和入账必须同时完成。

11.1.4 批量操作：UNWIND 的高效批量导入

当你需要一次性导入大量数据时，千万不要一条一条地执行 CREATE——那会慢到让你怀疑人生。正确的做法是使用 Cypher 的 `UNWIND` 语句，一次性处理整个列表：

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver("bolt://localhost:7687", aut

# ============================================================
# 准备批量数据（通常来自 CSV、API 或数据库查询结果）
# ============================================================
people_data = [
    {"name": "刘备", "age": 35, "role": "主公"},
    {"name": "关羽", "age": 33, "role": "大将"},
    {"name": "张飞", "age": 30, "role": "大将"},
    {"name": "诸葛亮", "age": 27, "role": "军师"},
    {"name": "赵云", "age": 25, "role": "护卫"},
]

# ============================================================
# 使用 UNWIND 实现批量导入
# ============================================================
def batch_create_people(tx, data_list):
    """
    使用 UNWIND 批量创建人物节点

    UNWIND 的工作原理：
    1. 接收一个列表参数 $people
    2. 将列表中的每个元素依次绑定为 person
    3. 对每个 person 执行 CREATE/MERGE 操作

    这比循环调用 session.run() 快 10-100 倍！
    因为：
    - 只需一次网络往返
    - Neo4j 可以优化批量操作的执行计划
    """
    query = """
    UNWIND $people AS person
```

```
    MERGE (p:Person {name: person.name})
    SET p.age = person.age,
        p.role = person.role,
        p.created_at = datetime()
    RETURN count(p) AS created_count
    """
    result = tx.run(query, people=data_list)
    return result.single()["created_count"]

with driver.session() as session:
    count = session.execute_write(batch_create_people, dat
    print(f"✅ 成功导入 {count} 个人物节点")

# ============================================================
# 批量创建关系（同样的思路）
# ============================================================
relationships = [
    {"from": "刘备", "to": "关羽", "type": "结拜兄弟"},
    {"from": "刘备", "to": "张飞", "type": "结拜兄弟"},
    {"from": "刘备", "to": "诸葛亮", "type": "君臣"},
    {"from": "刘备", "to": "赵云", "type": "君臣"},
]

def batch_create_relationships(tx, rel_list):
    """批量创建关系"""
    query = """
    UNWIND $rels AS rel
    MATCH (a:Person {name: rel.from}), (b:Person {name: re
    MERGE (a)-[r:BONDED {type: rel.type}]->(b)
    RETURN count(r) AS created_count
    """
    result = tx.run(query, rels=rel_list)
    return result.single()["created_count"]
```

```
with driver.session() as session:
    count = session.execute_write(batch_create_relationshi
    print(f"✅ 成功创建 {count} 条关系")
```

性能提示： 当数据量超过 10000 条时，建议将数据分成多个批次，每批 1000–5000 条。这样可以避免单个事务过大导致内存问题。

11.2 qdrant-client：向量数据库的 Python 接口

现在让我们转向向量数据库 Qdrant。`qdrant-client` 是 Qdrant 官方提供的 Python SDK，用起来非常顺手。

11.2.1 安装与连接

```
pip install qdrant-client
```

```
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams

# ============================================================
# 方式1：连接本地 Qdrant 实例（推荐开发阶段使用）
# ============================================================
client = QdrantClient(
    host="localhost",       # Qdrant 服务器地址
    port=6333,              # HTTP 端口（默认 6333）
    # gRPC 端口默认 6334，客户端会自动使用 gRPC 以获得更好性能
)

# ============================================================
# 方式2：连接 Qdrant Cloud（生产环境）
# ============================================================
# cloud_client = QdrantClient(
#     url="https://your-cluster-url.qdrant.io",
#     api_key="your-api-key-here"
# )

# ============================================================
# 方式3：纯内存模式（适合测试和原型验证）
# ============================================================
# memory_client = QdrantClient(":memory:")
# 数据只存在于内存中，程序退出后数据消失

# 测试连接
print("✅ Qdrant 连接成功！")
print(f"服务器信息：{client.get_collections()}")
```

11.2.2 创建 Collection（向量集合）

Collection 就像是关系型数据库中的”表”，但它是专门为向量设计的：

```
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, P

client = QdrantClient(host="localhost", port=6333)

# ============================================================
# 创建 Collection
# ============================================================
collection_name = "movie_embeddings"

# 先检查 Collection 是否已存在，避免重复创建
collections = [c.name for c in client.get_collections().co

if collection_name not in collections:
    client.create_collection(
        collection_name=collection_name,
        vectors_config=VectorParams(
            size=768,                # 向量维度（取决于你使用
                                     # OpenAI text-embedd
                                     # BGE-base-zh → 768
                                     # sentence-transform
            distance=Distance.COSINE  # 相似度度量方式
                                     # 可选：COSINE（余弦）
        )
    )
    print(f"✅ Collection '{collection_name}' 创建成功！")
else:
    print(f"ℹ️ Collection '{collection_name}' 已存在，跳过创
```

11.2.3 上传向量数据

```
from qdrant_client import QdrantClient
from qdrant_client.models import PointStruct
import uuid

client = QdrantClient(host="localhost", port=6333)
collection_name = "movie_embeddings"

# ============================================================
# 准备向量数据
# ============================================================
# 在实际应用中，这些向量应该由嵌入模型生成
# 这里我们用随机向量做演示
import random

movies_to_upload = [
    {
        "id": 1,
        "vector": [random.random() for _ in range(768)],
        "payload": {
            "title": "流浪地球",
            "director": "郭帆",
            "year": 2019,
            "genre": "科幻",
            "description": "太阳即将毁灭，人类启动流浪地球计划"
        }
    },
    {
        "id": 2,
        "vector": [random.random() for _ in range(768)],
        "payload": {
            "title": "哪吒之魔童降世",
            "director": "饺子",
            "year": 2019,
            "genre": "动画",
```

```
            "description": "哪吒虽生而为魔，却逆天改命"
        }
    },
    {
        "id": 3,
        "vector": [random.random() for _ in range(768)],
        "payload": {
            "title": "战狼2",
            "director": "吴京",
            "year": 2017,
            "genre": "动作",
            "description": "冷锋在非洲营救同胞的故事"
        }
    }
]

# ============================================================
# 将数据转换为 PointStruct 格式并上传
# ============================================================
points = [
    PointStruct(
        id=movie["id"],          # 唯一标识符（可以是整数或字符
        vector=movie["vector"],  # 向量数据
        payload=movie["payload"] # 附带的元数据（任意 JSON 结
    )
    for movie in movies_to_upload
]

# upload_points 方法支持批量上传
operation_info = client.upsert(
    collection_name=collection_name,
    wait=True,           # wait=True 表示等待操作完成后再返回
                         # wait=False 表示异步提交，立即返回
    points=points
)
```

```
print(f"✅ 成功上传 {len(points)} 条向量数据")
print(f"操作结果：{operation_info}")
```

11.2.4 向量搜索

这是 Qdrant 最核心的功能——根据一个查询向量，找到最相似的向量：

```
from qdrant_client import QdrantClient
from qdrant_client.models import Filter, FieldCondition, M

client = QdrantClient(host="localhost", port=6333)
collection_name = "movie_embeddings"

# ============================================================
# 基础搜索：找到与查询向量最相似的 Top-K 结果
# ============================================================
# 模拟一个查询向量（实际应用中，这个向量来自嵌入模型）
query_vector = [random.random() for _ in range(768)]

search_results = client.search(
    collection_name=collection_name,
    query_vector=query_vector,
    limit=5,  # 返回最相似的前 5 个结果（Top-K）
)

print("🔍 基础搜索结果：")
for result in search_results:
    print(f"  ID: {result.id}")
    print(f"  标题: {result.payload['title']}")
    print(f"  相似度: {result.score:.4f}")
    print(f"  ---")

# ============================================================
# 带过滤条件的搜索
# ============================================================
# 场景：只在"科幻"类型中搜索
filtered_results = client.search(
    collection_name=collection_name,
    query_vector=query_vector,
    query_filter=Filter(
        must=[
```

```
            # must 表示必须满足的条件（类似 SQL 的 WHERE）
            FieldCondition(
                key="genre",           # payload 中的字段名
                match=MatchValue(
                    value="科幻"       # 精确匹配值
                )
            )
        ]
    ),
    limit=3,
)

print("\n🔍 仅搜索'科幻'类型：")
for result in filtered_results:
    print(f"  {result.payload['title']} - 相似度: {result.s

# ============================================================
# 多条件过滤搜索
# ============================================================
from qdrant_client.models import Range

advanced_results = client.search(
    collection_name=collection_name,
    query_vector=query_vector,
    query_filter=Filter(
        must=[
            # 条件1：年份 >= 2018
            FieldCondition(
                key="year",
                range=Range(gte=2018)
            ),
        ],
        should=[
            # should 表示满足任一条件即可（类似 SQL 的 OR）
            FieldCondition(key="genre", match=MatchValue(v
```

```
            FieldCondition(key="genre", match=MatchValue(v
        ]
    ),
    limit=5,
)

print("\n🔍 2018年后的科幻或动画电影：")
for result in advanced_results:
    print(f"  {result.payload['title']} ({result.payload['
          f"- {result.payload['genre']}")
```

11.3 NetworkX：图分析与可视化

NetworkX 是 Python 中最流行的图分析库。它不像 Neo4j 那样是一个数据库，而是一个纯内存的图计算框架——特别适合做图算法分析和可视化。

11.3.1 安装与基础操作

```
pip install networkx matplotlib
```

```
import networkx as nx
import matplotlib.pyplot as plt

# 设置中文字体支持（macOS 用户）
plt.rcParams['font.sans-serif'] = ['Arial Unicode MS', 'Si
plt.rcParams['axes.unicode_minus'] = False

# ============================================================
# 创建一个有向图
# ============================================================
G = nx.DiGraph()  # DiGraph = Directed Graph（有向图）
                  # 如果是无向图，用 nx.Graph()

# 添加节点（可以附带属性）
G.add_node("刘备", role="主公", kingdom="蜀")
G.add_node("关羽", role="大将", kingdom="蜀")
G.add_node("张飞", role="大将", kingdom="蜀")
G.add_node("诸葛亮", role="军师", kingdom="蜀")
G.add_node("曹操", role="主公", kingdom="魏")
G.add_node("孙权", role="主公", kingdom="吴")

# 添加边（可以附带权重和类型）
G.add_edge("刘备", "关羽", relation="结拜", weight=10)
G.add_edge("刘备", "张飞", relation="结拜", weight=10)
G.add_edge("刘备", "诸葛亮", relation="三顾茅庐", weight=8)
G.add_edge("关羽", "曹操", relation="暂投", weight=3)
G.add_edge("曹操", "孙权", relation="敌对", weight=5)
G.add_edge("刘备", "孙权", relation="联盟", weight=7)

print(f"图的节点数：{G.number_of_nodes()}")
print(f"图的边数：{G.number_of_edges()}")
print(f"所有节点：{list(G.nodes())}")
print(f"所有边：{list(G.edges(data=True))}")  # data=True 显
```

11.3.2 图算法分析

NetworkX 内置了大量图算法，让我们来试试最常用的几个：

```
import networkx as nx

# 创建一个示例图
G = nx.DiGraph()
edges = [
    ("刘备", "关羽"), ("刘备", "张飞"), ("刘备", "诸葛亮"),
    ("诸葛亮", "关羽"), ("诸葛亮", "张飞"),
    ("关羽", "曹操"),
    ("曹操", "孙权"),
    ("孙权", "刘备"),  # 形成环路，增加图的复杂性
]
G.add_edges_from(edges)

# ============================================================
# 算法1：度中心性（Degree Centrality）
# ============================================================
# 衡量一个节点在网络中的"连接度"
# 值越高，说明该节点连接越多其他节点，越"重要"
degree_centrality = nx.degree_centrality(G)
print("📊 度中心性排名：")
for node, score in sorted(degree_centrality.items(),
                          key=lambda x: x[1], reverse=True
    print(f"  {node}: {score:.3f}")

# ============================================================
# 算法2：PageRank（网页排名算法）
# ============================================================
# Google 创始人发明的算法，不仅考虑连接数量，还考虑连接质量
# 被重要节点连接的节点，自身也会变得更重要
pagerank = nx.pagerank(G, alpha=0.85)  # alpha 是阻尼系数，通
print("\n📊 PageRank 排名：")
for node, score in sorted(pagerank.items(),
                          key=lambda x: x[1], reverse=True
    print(f"  {node}: {score:.4f}")
```

```
# ============================================================
# 算法3：最短路径
# ============================================================
# 找到两个节点之间的最短路径
try:
    path = nx.shortest_path(G, source="诸葛亮", target="孙权
    print(f"\n📊 诸葛亮 → 孙权 的最短路径：")
    print(f"  {' → '.join(path)}")
    print(f"  路径长度：{nx.shortest_path_length(G, '诸葛亮'
except nx.NetworkXNoPath:
    print("两个节点之间不存在路径")

# ============================================================
# 算法4：社区检测（Community Detection）
# ============================================================
# 发现图中的紧密子群（社区）
# 注意：社区检测通常在无向图上效果更好
G_undirected = G.to_undirected()
communities = nx.community.girvan_newman(G_undirected)
# girvan_newman 返回一个迭代器，每次返回更细粒度的社区划分
top_communities = next(communities)
print(f"\n📊 社区检测结果（{len(top_communities)} 个社区）：")
for i, community in enumerate(top_communities):
    print(f"  社区 {i+1}：{community}")

# ============================================================
# 算法5：连通性分析
# ============================================================
print(f"\n📊 连通性分析：")
print(f"  是否为强连通图：{nx.is_strongly_connected(G)}")
print(f"  是否为弱连通图：{nx.is_weakly_connected(G)}")

# 找出强连通分量
scc = list(nx.strongly_connected_components(G))
```

```
print(f"  强连通分量数量: {len(scc)}")
for i, component in enumerate(scc):
    print(f"    分量 {i+1}: {component}")
```

11.3.3 图的可视化

```
import networkx as nx
import matplotlib.pyplot as plt

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['Arial Unicode MS', 'Si
plt.rcParams['axes.unicode_minus'] = False

# 创建图
G = nx.DiGraph()
G.add_node("刘备", kingdom="蜀")
G.add_node("关羽", kingdom="蜀")
G.add_node("张飞", kingdom="蜀")
G.add_node("诸葛亮", kingdom="蜀")
G.add_node("曹操", kingdom="魏")
G.add_node("孙权", kingdom="吴")
G.add_edge("刘备", "关羽", relation="结拜")
G.add_edge("刘备", "张飞", relation="结拜")
G.add_edge("刘备", "诸葛亮", relation="君臣")
G.add_edge("关羽", "曹操", relation="暂投")
G.add_edge("曹操", "孙权", relation="敌对")
G.add_edge("刘备", "孙权", relation="联盟")

# ============================================================
# 自定义可视化样式
# ============================================================
fig, ax = plt.subplots(1, 1, figsize=(12, 8))

# 节点颜色映射（按势力着色）
color_map = {"蜀": "#FF6B6B", "魏": "#4ECDC4", "吴": "#45B7
node_colors = [color_map[G.nodes[n].get('kingdom', 'unknow

# 节点大小（根据度数动态调整）
node_sizes = [3000 + G.degree(n) * 500 for n in G.nodes()]
```

```
# 使用 spring 布局算法（力导向布局）
# 它模拟弹簧力，让相连的节点靠近，不相连的节点远离
pos = nx.spring_layout(G, k=2, iterations=50, seed=42)

# 绘制节点
nx.draw_networkx_nodes(
    G, pos,
    node_color=node_colors,
    node_size=node_sizes,
    alpha=0.9,
    edgecolors='black',
    linewidths=2
)

# 绘制节点标签
nx.draw_networkx_labels(
    G, pos,
    font_size=14,
    font_weight='bold',
    font_family='sans-serif'
)

# 绘制边
nx.draw_networkx_edges(
    G, pos,
    edge_color='gray',
    arrows=True,
    arrowsize=20,
    width=2,
    connectionstyle="arc3,rad=0.1"  # 弧线边，避免重叠
)

# 绘制边的标签（关系类型）
edge_labels = {(u, v): d['relation'] for u, v, d in G.edge
nx.draw_networkx_edge_labels(
```

```
    G, pos,
    edge_labels=edge_labels,
    font_size=10,
    font_color='red'
)

plt.title("三国人物关系图", fontsize=18, fontweight='bold')
plt.axis('off')  # 隐藏坐标轴
plt.tight_layout()
plt.savefig("sanguo_graph.png", dpi=150, bbox_inches='tigh
print("✅ 图片已保存为 sanguo_graph.png")
plt.show()
```

11.3.4 从 NetworkX 导出到 Neo4j

这是一个非常实用的技能——你可以先在 NetworkX 中做图分析，然后把结果写回 Neo4j：

```
import networkx as nx
from neo4j import GraphDatabase

# ============================================================
# 第一步：在 NetworkX 中构建和分析图
# ============================================================
G = nx.DiGraph()

# 假设这是从某个数据源加载的图
G.add_edge("A", "B", weight=5)
G.add_edge("B", "C", weight=3)
G.add_edge("A", "C", weight=1)
G.add_edge("C", "D", weight=7)
G.add_edge("D", "A", weight=2)

# 计算 PageRank
pagerank_scores = nx.pagerank(G)
print("PageRank 计算结果：")
for node, score in pagerank_scores.items():
    print(f"  {node}: {score:.4f}")
    # 把 PageRank 分数作为节点属性
    G.nodes[node]['pagerank'] = score

# ============================================================
# 第二步：将分析结果写入 Neo4j
# ============================================================
driver = GraphDatabase.driver("bolt://localhost:7687", aut

def export_networkx_to_neo4j(tx, graph):
    """
    将 NetworkX 图的节点和边批量导入 Neo4j

    策略：
    1. 先导入所有节点（携带属性）
```

```
    2. 再导入所有边（携带权重）
    """
    # 导入节点
    nodes_data = [
        {"id": node, "pagerank": data.get('pagerank', 0.0)
        for node, data in graph.nodes(data=True)
    ]
    tx.run("""
        UNWIND $nodes AS node_data
        MERGE (n:AnalysisNode {id: node_data.id})
        SET n.pagerank = node_data.pagerank
    """, nodes=nodes_data)

    # 导入边
    edges_data = [
        {"source": u, "target": v, "weight": data.get('wei
        for u, v, data in graph.edges(data=True)
    ]
    tx.run("""
        UNWIND $edges AS edge_data
        MATCH (a:AnalysisNode {id: edge_data.source})
        MATCH (b:AnalysisNode {id: edge_data.target})
        MERGE (a)-[r:CONNECTS_TO]->(b)
        SET r.weight = edge_data.weight
    """, edges=edges_data)

    return len(nodes_data), len(edges_data)

with driver.session() as session:
    node_count, edge_count = session.execute_write(
        export_networkx_to_neo4j, G
    )
    print(f"\n✅ 已导出到 Neo4j：{node_count} 个节点，{edge_c
```

11.4 pandas + 图：数据清洗与批量导入

pandas 是 Python 数据处理的核心工具。在这一节中，我们将学习如何用 pandas 清洗数据，然后批量导入 Neo4j。

11.4.1 读取和清洗 CSV 数据

```
import pandas as pd
import numpy as np

# ============================================================
# 创建示例 CSV 数据（模拟真实的脏数据）
# ============================================================
# 在实际项目中，你会用 pd.read_csv("your_file.csv") 读取文件
# 这里我们用 DataFrame 模拟
raw_data = pd.DataFrame({
    "电影名称": ["流浪地球 ", "  哪吒之魔童降世", "战狼2", "", N
                "流浪地球", "你好，李焕英", "唐人街探案3"],
    "导演": ["郭帆", "饺子", "吴京", "未知", "张三",
            "郭帆", "贾玲", " 陈思诚 "],
    "年份": [2019, 2019, 2017, 2020, None, 2019, 2021, 202
    "票房(亿)": ["46.86", "50.35", "56.94", "unknown", "10"
                 "46.86", "54.14", "45.24"],
    "评分": [7.9, 8.4, 7.1, 6.5, None, 7.9, 7.8, 5.3],
    "类型": ["科幻/灾难", "动画/奇幻", "动作/战争", "剧情", "喜
            "科幻,灾难", "喜剧/剧情", "喜剧/悬疑"],
})

print("📋 原始数据：")
print(raw_data)
print(f"\n数据形状：{raw_data.shape[0]} 行 × {raw_data.shape

# ============================================================
# 数据清洗流水线
# ============================================================
def clean_movie_data(df):
    """
    电影数据清洗函数

    清洗步骤：
    1. 去除完全空白的行
```

```
    2. 去除重复行
    3. 清理字符串字段（去空格、统一分隔符）
    4. 转换数据类型
    5. 填充缺失值
    """
    # 步骤1：去除电影名称为空或 NaN 的行
    df = df.dropna(subset=["电影名称"])           # 删除名称为
    df = df[df["电影名称"].str.strip() != ""]    # 删除名称为
    print(f"  去除空行后：{len(df)} 行")

    # 步骤2：清理字符串字段
    # strip() 去除首尾空格
    for col in ["电影名称", "导演", "类型"]:
        df[col] = df[col].str.strip()

    # 步骤3：统一类型分隔符（将逗号、斜杠等统一为 "/"）
    df["类型"] = df["类型"].str.replace(",", "/", regex=Fal
    print(f"  清理字符串后：{len(df)} 行")

    # 步骤4：去除重复行（基于电影名称 + 年份）
    df = df.drop_duplicates(subset=["电影名称", "年份"], kee
    print(f"  去重后：{len(df)} 行")

    # 步骤5：转换票房字段为数值类型
    # errors='coerce' 表示无法转换的值变为 NaN
    df["票房(亿)"] = pd.to_numeric(df["票房(亿)"], errors="c

    # 步骤6：填充缺失值
    df["年份"] = df["年份"].fillna(0).astype(int)   # 缺失年
    df["评分"] = df["评分"].fillna(df["评分"].mean()) # 缺失
    df["票房(亿)"] = df["票房(亿)"].fillna(0)         # 缺失
    print(f"  填充缺失值后：{len(df)} 行")

    # 步骤7：将类型字段拆分为列表
    df["类型列表"] = df["类型"].str.split("/")
```

```
    return df.reset_index(drop=True)  # 重置索引

cleaned = clean_movie_data(raw_data.copy())
print("\n📋 清洗后的数据：")
print(cleaned[["电影名称", "导演", "年份", "票房(亿)", "评分",
```

11.4.2 从 pandas 批量导入 Neo4j

```
import pandas as pd
from neo4j import GraphDatabase

driver = GraphDatabase.driver("bolt://localhost:7687", aut

# 假设 cleaned 是上一步清洗后的 DataFrame

# ============================================================
# 策略：分三步导入——节点、类型节点、关系
# ============================================================

# ------------------------------------------------
# 步骤1：导入电影节点
# ------------------------------------------------
movies_data = cleaned[["电影名称", "导演", "年份", "票房(亿)",

def import_movies(tx, movies):
    """批量创建电影节点"""
    query = """
    UNWIND $movies AS movie
    MERGE (m:Movie {title: movie.电影名称})
    SET m.director = movie.导演,
        m.year = movie.年份,
        m.box_office = movie["票房(亿)"],
        m.rating = movie.评分
    RETURN count(m) AS count
    """
    result = tx.run(query, movies=movies)
    return result.single()["count"]

with driver.session() as session:
    count = session.execute_write(import_movies, movies_da
    print(f"✅ 导入 {count} 部电影")
```

```
# ----------------------------------------------------
# 步骤2：导入类型节点
# ----------------------------------------------------
# 提取所有不重复的类型
all_genres = set()
for genre_list in cleaned["类型列表"]:
    all_genres.update(genre_list)
genres_data = [{"name": g} for g in all_genres]

def import_genres(tx, genres):
    """批量创建类型节点"""
    query = """
    UNWIND $genres AS genre
    MERGE (g:Genre {name: genre.name})
    RETURN count(g) AS count
    """
    result = tx.run(query, genres=genres)
    return result.single()["count"]

with driver.session() as session:
    count = session.execute_write(import_genres, genres_da
    print(f"✅ 导入 {count} 个类型")

# ----------------------------------------------------
# 步骤3：创建电影-类型关系
# ----------------------------------------------------
# 展开每部电影的类型列表，构建关系数据
movie_genre_rels = []
for _, row in cleaned.iterrows():
    for genre in row["类型列表"]:
        movie_genre_rels.append({
            "movie": row["电影名称"],
            "genre": genre
        })
```

```
def import_movie_genre_relations(tx, rels):
    """批量创建电影-类型关系"""
    query = """
    UNWIND $rels AS rel
    MATCH (m:Movie {title: rel.movie})
    MATCH (g:Genre {name: rel.genre})
    MERGE (m)-[:BELONGS_TO]->(g)
    RETURN count(*) AS count
    """
    result = tx.run(query, rels=rels)
    return result.single()["count"]

with driver.session() as session:
    count = session.execute_write(import_movie_genre_relat
    print(f"✅ 创建 {count} 条电影-类型关系")
```

11.5 综合案例：数据清洗 → Neo4j 导入 → Qdrant 向量化

终于到了把所有技能串联起来的时刻了！我们将构建一个完整的 ETL 流水线：从原始数据开始，经过清洗，导入 Neo4j 构建知识图谱，同时把文本描述转为向量存入 Qdrant。

11.5.1 完整流水线脚本

```
"""
============================================================
知识图谱 ETL 流水线
功能：CSV 数据清洗 → Neo4j 图数据库导入 → Qdrant 向量化
作者：树懒老K
============================================================

使用方式：
    pip install neo4j qdrant-client pandas sentence-transf

    # 确保 Neo4j 和 Qdrant 已启动
    python etl_pipeline.py
"""

import pandas as pd
import numpy as np
from neo4j import GraphDatabase
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, P
import logging
import time

# ============================================================
# 配置日志
# ============================================================
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s',
    datefmt='%H:%M:%S'
)
logger = logging.getLogger(__name__)


# ============================================================
```

```
# 第一阶段：数据准备与清洗
# ============================================================
class DataCleaner:
    """
    数据清洗器

    职责：
    - 读取原始数据（CSV/DataFrame）
    - 清洗脏数据
    - 输出干净的、可导入的数据
    """

    def __init__(self):
        self.df = None

    def load_from_csv(self, filepath):
        """从 CSV 文件加载数据"""
        logger.info(f"正在加载数据：{filepath}")
        self.df = pd.read_csv(filepath)
        logger.info(f"加载完成：{len(self.df)} 行数据")
        return self

    def load_from_dataframe(self, df):
        """从 DataFrame 加载数据（用于测试）"""
        self.df = df.copy()
        logger.info(f"加载完成：{len(self.df)} 行数据")
        return self

    def clean(self):
        """
        执行完整的数据清洗流程

        返回：清洗后的 DataFrame
        """
        logger.info("开始数据清洗...")
```

```
        initial_count = len(self.df)
        df = self.df.copy()

        # 1. 去除空行
        df = df.dropna(how="all")  # 所有字段都为空的行才删除
        logger.info(f"  去除全空行: {initial_count} → {len(d

        # 2. 清理字符串字段
        str_cols = df.select_dtypes(include=['object']).co
        for col in str_cols:
            df[col] = df[col].astype(str).str.strip()
            df[col] = df[col].replace({'nan': None, 'None'

        # 3. 去重
        before_dedup = len(df)
        df = df.drop_duplicates()
        logger.info(f"  去重: {before_dedup} → {len(df)}")

        # 4. 数值字段转换
        numeric_candidates = ['年份', 'year', '票房', 'box_
        for col in numeric_candidates:
            if col in df.columns:
                df[col] = pd.to_numeric(df[col], errors='c

        self.df = df
        logger.info(f"清洗完成! {initial_count} → {len(df)}
        return df

    def get_clean_data(self):
        """获取清洗后的数据"""
        return self.df


# ============================================================
# 第二阶段: Neo4j 图数据库导入
```

```
# ============================================================
class Neo4jImporter:
    """
    Neo4j 导入器

    职责：
    - 连接 Neo4j
    - 批量导入节点和关系
    - 创建索引
    """

    def __init__(self, uri="bolt://localhost:7687",
                 user="neo4j", password="your_password"):
        self.driver = GraphDatabase.driver(uri, auth=(user
        logger.info("Neo4j 连接成功")

    def create_constraints(self):
        """
        创建唯一性约束（幂等导入的关键）

        约束的作用：
        - 确保同一属性的节点不会重复创建
        - 自动创建索引，加速 MERGE 查询
        """
        constraints = [
            "CREATE CONSTRAINT movie_title IF NOT EXISTS F
            "CREATE CONSTRAINT person_name IF NOT EXISTS F
            "CREATE CONSTRAINT genre_name IF NOT EXISTS FO
        ]
        with self.driver.session() as session:
            for constraint in constraints:
                try:
                    session.run(constraint)
                except Exception as e:
                    logger.warning(f"约束创建提示：{e}")
```

```
        logger.info("✅ 唯一性约束已创建")

    def import_movies(self, df):
        """导入电影节点"""
        movies = df.to_dict("records")

        def _import(tx, records):
            query = """
            UNWIND $records AS rec
            MERGE (m:Movie {title: rec.title})
            SET m.director = rec.director,
                m.year = toInteger(rec.year),
                m.rating = toFloat(rec.rating),
                m.description = rec.description
            """
            tx.run(query, records=records)

        # 分批导入，每批 500 条
        batch_size = 500
        for i in range(0, len(movies), batch_size):
            batch = movies[i:i + batch_size]
            with self.driver.session() as session:
                session.execute_write(_import, batch)
            logger.info(f"  导入电影：{min(i + batch_size,

    def import_relationships(self, df):
        """导入导演关系"""
        rels = df[["title", "director"]].dropna().to_dict(

        def _import(tx, records):
            # 先创建导演节点
            tx.run("""
                UNWIND $records AS rec
                MERGE (p:Person {name: rec.director})
            """, records=records)
```

```
            # 再创建关系
            tx.run("""
                UNWIND $records AS rec
                MATCH (m:Movie {title: rec.title})
                MATCH (p:Person {name: rec.director})
                MERGE (m)-[:DIRECTED_BY]->(p)
            """, records=records)

        with self.driver.session() as session:
            session.execute_write(_import, rels)
        logger.info(f"✅ 导入 {len(rels)} 条导演关系")

    def close(self):
        """关闭连接"""
        self.driver.close()
        logger.info("Neo4j 连接已关闭")


# ============================================================
# 第三阶段：Qdrant 向量化与导入
# ============================================================
class QdrantVectorizer:
    """
    Qdrant 向量化器

    职责：
    - 将文本转为向量
    - 上传到 Qdrant
    """

    def __init__(self, host="localhost", port=6333,
                 collection_name="movies", model_name="all
        self.client = QdrantClient(host=host, port=port)
        self.collection_name = collection_name
```

```
        # 加载嵌入模型
        try:
            from sentence_transformers import SentenceTrar
            self.model = SentenceTransformer(model_name)
            self.vector_dim = self.model.get_sentence_embe
            logger.info(f"嵌入模型加载成功：{model_name}（{se
        except ImportError:
            logger.warning("sentence-transformers 未安装，将
            self.model = None
            self.vector_dim = 384

    def ensure_collection(self):
        """确保 Collection 存在"""
        collections = [c.name for c in self.client.get_col
        if self.collection_name not in collections:
            self.client.create_collection(
                collection_name=self.collection_name,
                vectors_config=VectorParams(
                    size=self.vector_dim,
                    distance=Distance.COSINE
                )
            )
            logger.info(f"✅ Collection '{self.collection_
        else:
            logger.info(f"ℹ️ Collection '{self.collection_

    def encode_texts(self, texts):
        """
        将文本列表转为向量列表

        如果未安装 sentence-transformers，则返回随机向量（仅用于
        """
        if self.model is not None:
            # 使用模型编码文本
            embeddings = self.model.encode(
```

```
                texts,
                show_progress_bar=True,
                batch_size=64
            )
            return embeddings.tolist()
        else:
            # 测试模式：返回随机向量
            return [np.random.rand(self.vector_dim).tolist

    def upload_vectors(self, df, text_column="description"
        """
        将 DataFrame 中的文本数据向量化并上传到 Qdrant

        参数：
            df: 数据 DataFrame
            text_column: 包含文本描述的列名
            id_column: 用作向量 ID 的列名
        """
        # 准备文本和 payload
        texts = df[text_column].fillna("").tolist()
        payloads = df.to_dict("records")

        # 文本向量化
        logger.info(f"正在编码 {len(texts)} 条文本...")
        vectors = self.encode_texts(texts)

        # 构建 Point 列表
        points = []
        for i, (vector, payload) in enumerate(zip(vectors,
            # 使用哈希值作为唯一 ID（避免冲突）
            point_id = hash(str(payload.get(id_column, i))
            points.append(PointStruct(
                id=abs(point_id),
                vector=vector,
                payload=payload
```

```
            ))

        # 批量上传（每批 100 条）
        batch_size = 100
        for i in range(0, len(points), batch_size):
            batch = points[i:i + batch_size]
            self.client.upsert(
                collection_name=self.collection_name,
                points=batch,
                wait=True
            )
            logger.info(f"  上传向量: {min(i + batch_size,

        logger.info(f"✅ 共上传 {len(points)} 条向量到 '{sel


# ============================================================
# 主流程编排
# ============================================================
def run_pipeline():
    """
    运行完整的 ETL 流水线

    流程:
    1. 数据准备 → 2. 数据清洗 → 3. Neo4j 导入 → 4. Qdrant 向量
    """
    start_time = time.time()

    # ------------------------------------------------------
    # 准备示例数据
    # ------------------------------------------------------
    sample_data = pd.DataFrame({
        "title": [
            "流浪地球", "哪吒之魔童降世", "战狼2",
            "你好，李焕英", "唐人街探案3", "长津湖",
```

```
            "满江红", "独行月球", "人生大事", "消失的她"
        ],
        "director": [
            "郭帆", "饺子", "吴京",
            "贾玲", "陈思诚", "陈凯歌",
            "张艺谋", "张吃鱼", "刘江江", "崔睿"
        ],
        "year": [2019, 2019, 2017, 2021, 2021, 2021, 2023,
        "rating": [7.9, 8.4, 7.1, 7.8, 5.3, 7.2, 7.0, 6.8,
        "description": [
            "太阳即将毁灭，人类在地球表面建造出巨大的推进器，寻找新
            "天地灵气孕育出一颗能量巨大的混元珠，被提炼为灵珠和魔丸
            "冷锋在一次任务中遭遇挫折，后来在非洲国家营救中国同胞。
            "贾晓玲穿越回到1981年，与年轻的母亲李焕英相遇。",
            "唐仁和秦风被卷入一场惊天大案的推理冒险。",
            "抗美援朝战争中，志愿军战士在长津湖战役的英勇事迹。",
            "南宋时期，一个小兵和一位副将被卷入一场巨大的阴谋。",
            "人类为抵御小行星撞击地球，部署了月球核弹，一名维修工被
            "殡葬师莫三妹在出殡中遇到了孤儿武小文，两人建立了特殊的
            "何非的妻子在东南亚失踪，一个陌生女人声称自己就是他的妻
        ]
    })

    # ------------------------------------------------------
    # 阶段1：数据清洗
    # ------------------------------------------------------
    logger.info("=" * 60)
    logger.info("阶段 1/3：数据清洗")
    logger.info("=" * 60)

    cleaner = DataCleaner()
    cleaner.load_from_dataframe(sample_data)
    clean_df = cleaner.clean()

    # ------------------------------------------------------
```

```
    # 阶段2：Neo4j 导入
    # ------------------------------------------------------------
    logger.info("=" * 60)
    logger.info("阶段 2/3：Neo4j 图数据库导入")
    logger.info("=" * 60)

    try:
        importer = Neo4jImporter(
            uri="bolt://localhost:7687",
            user="neo4j",
            password="your_password"
        )
        importer.create_constraints()
        importer.import_movies(clean_df)
        importer.import_relationships(clean_df)
        importer.close()
    except Exception as e:
        logger.error(f"Neo4j 导入失败：{e}")
        logger.info("请确保 Neo4j 已启动，然后重试")

    # ------------------------------------------------------------
    # 阶段3：Qdrant 向量化
    # ------------------------------------------------------------
    logger.info("=" * 60)
    logger.info("阶段 3/3：Qdrant 向量化导入")
    logger.info("=" * 60)

    try:
        vectorizer = QdrantVectorizer(
            host="localhost",
            port=6333,
            collection_name="movie_descriptions"
        )
        vectorizer.ensure_collection()
        vectorizer.upload_vectors(
```

```
            clean_df,
            text_column="description",
            id_column="title"
        )
    except Exception as e:
        logger.error(f"Qdrant 导入失败：{e}")
        logger.info("请确保 Qdrant 已启动，然后重试")

    # ------------------------------------------------------
    # 完成
    # ------------------------------------------------------
    elapsed = time.time() - start_time
    logger.info("=" * 60)
    logger.info(f"🎉 流水线运行完成！耗时：{elapsed:.2f} 秒")
    logger.info("=" * 60)


# 运行流水线
if __name__ == "__main__":
    run_pipeline()
```

这个脚本展示了一个完整的 ETL 流水线架构。在实际项目中，你可以根据需要扩展每个阶段的逻辑——比如增加更多的数据清洗规则、更复杂的图模型设计、更精细的向量搜索策略等。

动手练习

练习：编写一个 Python 脚本，从 CSV 导入 1000 条记录到 Neo4j 和 Qdrant

目标： 综合运用本章所学知识，独立完成一个完整的导入脚本。

要求： 1. 生成或获取一份包含 1000 条记录的 CSV 数据（可以是书籍、商品、用户等任何实体） 2. 使用 pandas 进行数据清洗 3. 将清洗后的数据导入 Neo4j（创建节点和关系） 4. 将文本描述向量化后导入 Qdrant 5. 编写一个查询函数，能同时在 Neo4j 和 Qdrant 中搜索

提示代码框架：

```
"""
练习：1000 条记录的完整导入

运行前请确保：
1. Neo4j 已启动（bolt://localhost:7687）
2. Qdrant 已启动（http://localhost:6333）
3. 已安装依赖：pip install neo4j qdrant-client pandas senter
"""

import pandas as pd
import numpy as np
from neo4j import GraphDatabase
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, P

# ========================
# 第一步：生成 1000 条模拟数据
# ========================
# TODO: 生成 1000 条书籍数据
# 字段：书名、作者、出版社、年份、评分、简介、类型
categories = ["科幻", "文学", "历史", "技术", "哲学", "经济",
authors = [f"作者_{i}" for i in range(100)]
publishers = ["人民文学出版社", "中信出版社", "机械工业出版社",

np.random.seed(42)
n_records = 1000

data = {
    "书名": [f"书籍_{i}" for i in range(n_records)],
    "作者": [np.random.choice(authors) for _ in range(n_re
    "出版社": [np.random.choice(publishers) for _ in range(
    "年份": np.random.randint(1990, 2024, size=n_records),
    "评分": np.round(np.random.uniform(5.0, 10.0, size=n_r
    "简介": [f"这是一本关于{np.random.choice(categories)}的精
```

```
            for _ in range(n_records)],
    "类型": [np.random.choice(categories) for _ in range(n_
}
df = pd.DataFrame(data)

# ========================
# 第二步：数据清洗
# ========================
# TODO: 实现数据清洗
# - 检查并去除重复数据
# - 清理空值
# - 确保数据类型正确

# ========================
# 第三步：导入 Neo4j
# ========================
# TODO: 创建 Book、Author、Publisher、Category 节点
# TODO: 创建 WRITTEN_BY、PUBLISHED_BY、BELONGS_TO 关系
# 提示：使用 UNWIND 批量导入

# ========================
# 第四步：导入 Qdrant
# ========================
# TODO: 将书籍简介向量化
# TODO: 上传到 Qdrant Collection

# ========================
# 第五步：混合查询函数
# ========================
# TODO: 实现一个函数，输入一个关键词：
# 1. 在 Neo4j 中搜索书名或作者包含关键词的书籍
# 2. 在 Qdrant 中搜索简介语义相似的书籍
# 3. 合并去重后返回结果
```

评分标准： - 成功生成和清洗 1000 条数据（20分） - 正确导入 Neo4j 并创建节点和关系（30分） - 正确向量化并导入 Qdrant（30分） - 混合查询函数可用（20分）

延伸阅读

1. **neo4j Python Driver 官方文档** — 深入了解连接池管理、重试机制、集群路由等高级特性。访问 https://neo4j.com/docs/python-manual/current/
2. **Qdrant Python Client 文档** — 学习稀疏向量、多向量、Payload 索引等进阶功能。访问 https://qdrant.github.io/qdrant/redoc/
3. **NetworkX 教程** — 探索更多图算法：社区检测、中心性分析、路径优化等。访问 https://networkx.org/documentation/stable/tutorial.html
4. **pandas 官方用户指南** — 掌握 GroupBy、Merge、Pivot 等高级数据操作。访问 https://pandas.pydata.org/docs/user_guide/
5. **UNWIND 批量导入最佳实践** — Neo4j 官方博客关于高性能批量导入的深度文章，包括分批策略、内存管理等。
6. **sentence-transformers 模型选择指南** — 不同嵌入模型的性能对比和选型建议，帮助你为特定场景选择最优模型。

本章小结

在这一章中，我们掌握了用 Python 操控图谱的四大核心技能：

1. **Neo4j Python 驱动**：学会了如何建立连接、执行查询、使用事务、批量导入。关键要点是：始终使用参数化查询防止注入，使用 UNWIND 实现高效批量操作，使用 read/write transaction 实现读写分离。
2. **Qdrant Python 客户端**：学会了创建 Collection、上传向量、执行搜索（包括带过滤条件的搜索）。Qdrant 的 Python API 设计非常直观，基本上就是”创建集合 → 上传点 → 搜索”三步走。
3. **NetworkX 图分析**：学会了在 Python 中构建图、运行图算法（PageRank、最短路径、社区检测等）、可视化图形、并将分析结果导出到 Neo4j。NetworkX 是一个纯内存库，适合中小规模的图分析任务。
4. **pandas 数据处理**：学会了数据清洗的完整流程（去空、去重、类型转换、缺失值填充），以及如何将清洗后的数据批量导入 Neo4j。

最后，我们将所有技能整合成了一个完整的 ETL 流水线脚本，实现了”数据清洗 → Neo4j 导入 → Qdrant 向量化”的自动化流程。这个模式在实际项目中非常常见，你可以把它作为自己项目

的起点。

在下一章中，我们将用一个完整的实战项目——电影知识图谱——来综合运用前面所有章节的知识，包括构建图谱对话系统。敬请期待！

第12章 实战项目：从零构建一个电影知识图谱

难度：★★★

引子

恭喜你，走到这里你已经掌握了知识图谱的所有基础技能！

在前面的章节中，我们学习了知识图谱的概念、Neo4j 的操作、向量数据库的原理、Python 的图谱操控技巧。但说实话，零散的知识点就像一堆砖头——只有把它们搭建成一座完整的房子，你才能真正体会到知识的力量。

这就是本章的意义所在。我们将一起完成一个从零开始的实战项目：**电影知识图谱与智能对话系统**。

在这个项目中，你会经历一个完整的开发生命周期：

- 设计项目架构
- 采集电影数据
- 用 LLM 抽取结构化信息
- 导入 Neo4j 构建图谱
- 导入 Qdrant 实现语义搜索
- 用 LangChain 搭建对话系统
- 用 Streamlit 搭建可视化前端

每一行代码都是可以直接运行的，每一个步骤都有详细的解释。泡杯咖啡，打开你的 IDE，让我们开始吧。

正文

12.1 项目设计

12.1.1 项目目标

我们的目标很明确：构建一个**能对话的电影知识图谱系统**。具体来说，它能做这些事情：

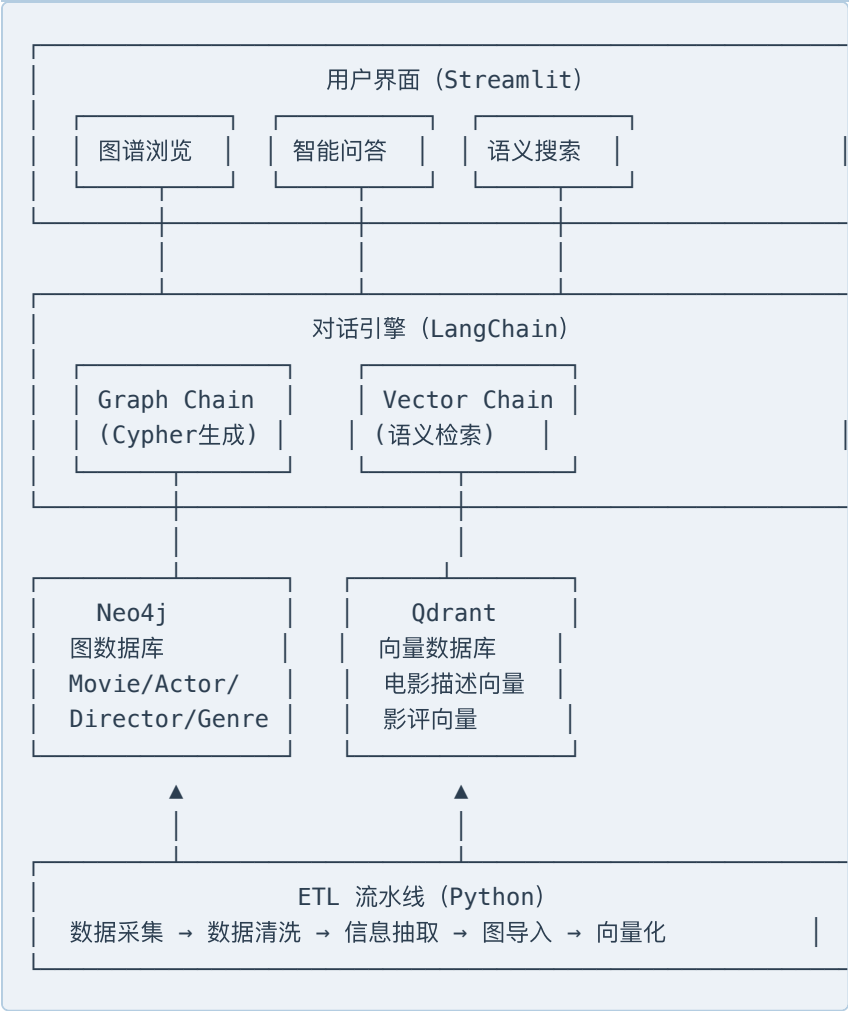
- **图谱浏览**：在可视化界面中浏览电影、演员、导演、类型之间的关系网络
- **图谱查询**：用自然语言提问，比如”周星驰导演过哪些电影？”、”评分最高的科幻片有哪些？”
- **语义搜索**：基于电影描述的语义相似度搜索，比如”有没有类似《流浪地球》那种太空冒险的电影？”
- **智能推荐**：综合图谱关系和语义相似度，为用户推荐电影

12.1.2 技术栈选择

表 16–1

组件	技术选型	理由
图数据库	Neo4j	业界标准，Cypher 查询强大
向量数据库	Qdrant	轻量级，Python SDK 友好
嵌入模型	sentence–transformers	本地运行，无需 API Key
LLM	OpenAI API / 本地 Ollama	信息抽取 + 对话
图操作框架	LangChain	内置 Neo4j 和 Qdrant 集成
前端	Streamlit	快速搭建交互式 Web 应用
数据处理	pandas	数据清洗和处理
数据可视化	pyvis / streamlit–agraph	图谱可视化

12.1.3 系统架构


用户界面（Streamlit）
图谱浏览
智能问答
语义搜索
对话引擎（LangChain）
Graph Chain
（Cypher生成）
Vector Chain
（语义检索）
Neo4j
图数据库
Movie/Actor/
Director/Genre
Qdrant
向量数据库
电影描述向量
影评向量
ETL 流水线（Python）
数据采集 → 数据清洗 → 信息抽取 → 图导入 → 向量化

12.1.4 项目目录结构

```
movie_knowledge_graph/
├── config.py               # 全局配置
├── data/
│   ├── raw/                # 原始采集数据
│   ├── processed/          # 清洗后的数据
│   └── sample_movies.csv   # 示例数据
├── etl/
│   ├── scraper.py          # 数据采集脚本
│   ├── cleaner.py          # 数据清洗
│   ├── extractor.py        # LLM 信息抽取
│   ├── neo4j_loader.py     # Neo4j 导入
│   └── qdrant_loader.py    # Qdrant 导入
├── app/
│   ├── graph_chain.py      # LangChain 图谱链
│   ├── vector_chain.py     # LangChain 向量链
│   ├── chatbot.py          # 对话引擎
│   └── streamlit_app.py    # Streamlit 前端
├── requirements.txt        # Python 依赖
└── README.md
```

12.2 数据采集

12.2.1 TMDB API 采集电影数据

TMDB（The Movie Database）是一个免费的电影数据库，提供丰富的 API。我们将用它来获取电影数据。

首先，你需要到 https://www.themoviedb.org/ 注册一个账号，然后在”设置 → API”中申请一个免费的 API Key。

```
"""
电影数据采集脚本
使用 TMDB API 获取电影数据

使用前请先获取 API Key: https://www.themoviedb.org/settings/a
"""

import requests
import pandas as pd
import time
import json
import os

# ============================================================
# 配置
# ============================================================
TMDB_API_KEY = "your_tmdb_api_key_here"  # 替换为你的 API Ke
TMDB_BASE_URL = "https://api.themoviedb.org/3"
LANGUAGE = "zh-CN"  # 获取中文数据
OUTPUT_DIR = "data/raw"

# 确保输出目录存在
os.makedirs(OUTPUT_DIR, exist_ok=True)


# ============================================================
# TMDB API 封装函数
# ============================================================
def tmdb_request(endpoint, params=None):
    """
    封装 TMDB API 请求

    参数:
        endpoint: API 端点路径（如 "/movie/popular"）
```

```
        params: 额外的查询参数

    返回:
        JSON 响应数据
    """
    url = f"{TMDB_BASE_URL}{endpoint}"

    # 默认参数
    default_params = {
        "api_key": TMDB_API_KEY,
        "language": LANGUAGE,
    }

    # 合并参数
    if params:
        default_params.update(params)

    # 发送请求
    response = requests.get(url, params=default_params, ti
    response.raise_for_status()  # 如果请求失败，抛出异常

    # 礼貌地等待 0.3 秒，避免触发频率限制
    time.sleep(0.3)

    return response.json()


def get_popular_movies(page=1):
    """
    获取热门电影列表

    参数:
        page: 页码 (1-500)，每页 20 部电影
    """
    data = tmdb_request("/movie/popular", {"page": page})
```

```
    return data.get("results", [])


def get_movie_details(movie_id):
    """
    获取电影的详细信息

    包含：完整简介、演职员表、类型、评分等
    """
    # append_to_response 可以一次请求多个子资源
    data = tmdb_request(
        f"/movie/{movie_id}",
        {"append_to_response": "credits,keywords"}
    )
    return data


def get_movie_reviews(movie_id, page=1):
    """
    获取电影的评论
    """
    data = tmdb_request(
        f"/movie/{movie_id}/reviews",
        {"page": page}
    )
    return data.get("results", [])


# ============================================================
# 批量采集流程
# ============================================================
def collect_movies(total_pages=5):
    """
    批量采集电影数据
```

```
    流程：
    1. 遍历热门电影列表（多页）
    2. 获取每部电影的详细信息
    3. 获取每部电影的评论（最多 5 条）
    4. 整合所有数据

    参数：
        total_pages: 采集多少页的热门电影（每页 20 部）

    返回：
        整合后的电影数据列表
    """
    all_movies = []

    # 第一步：获取电影 ID 列表
    print(f"📡 正在获取热门电影列表（共 {total_pages} 页）...")
    movie_ids = []
    for page in range(1, total_pages + 1):
        movies = get_popular_movies(page)
        for movie in movies:
            movie_ids.append(movie["id"])
        print(f"  第 {page} 页：获取 {len(movies)} 部电影")

    print(f"\n共获取 {len(movie_ids)} 部电影 ID")
    print(f"📡 正在获取每部电影的详细信息...\n")

    # 第二步：逐个获取详情
    for i, movie_id in enumerate(movie_ids):
        try:
            # 获取详情
            details = get_movie_details(movie_id)

            # 获取评论（最多 3 条）
            reviews = get_movie_reviews(movie_id)
            review_texts = [r["content"] for r in reviews[
```

```
            # 整合数据
            movie_data = {
                "id": details.get("id"),
                "title": details.get("title", ""),
                "original_title": details.get("original_ti
                "overview": details.get("overview", ""),
                "release_date": details.get("release_date"
                "year": int(details.get("release_date", "0
                "rating": details.get("vote_average", 0),
                "vote_count": details.get("vote_count", 0)
                "runtime": details.get("runtime", 0),
                "revenue": details.get("revenue", 0),
                "budget": details.get("budget", 0),
                "genres": [g["name"] for g in details.get(
                "keywords": [k["name"] for k in details.ge

                # 演职员信息
                "director": "",
                "actors": [],
                "reviews": review_texts,
            }

            # 从 credits 中提取导演和演员
            credits = details.get("credits", {})
            for crew in credits.get("crew", []):
                if crew["job"] == "Director":
                    movie_data["director"] = crew["name"]
                    break

            # 取前 10 位演员
            for cast in credits.get("cast", [])[:10]:
                movie_data["actors"].append({
                    "name": cast["name"],
                    "character": cast.get("character", "")
```

```
                    "order": cast.get("order", 99)
                })

            all_movies.append(movie_data)

            # 进度显示
            if (i + 1) % 10 == 0:
                print(f"  已处理 {i + 1}/{len(movie_ids)} 部

        except Exception as e:
            print(f"  ⚠️ 电影 {movie_id} 获取失败：{e}")
            continue

    print(f"\n✅ 数据采集完成！共获取 {len(all_movies)} 部电影
    return all_movies


# ============================================================
# 保存数据
# ============================================================
def save_movies(movies, filename="movies_raw.json"):
    """将采集的数据保存为 JSON 文件"""
    filepath = os.path.join(OUTPUT_DIR, filename)
    with open(filepath, "w", encoding="utf-8") as f:
        json.dump(movies, f, ensure_ascii=False, indent=2)
    print(f"💾 数据已保存到：{filepath}")


def movies_to_dataframe(movies):
    """将电影数据转换为 DataFrame，方便后续处理"""
    rows = []
    for movie in movies:
        rows.append({
            "id": movie["id"],
            "title": movie["title"],
```

```
            "original_title": movie["original_title"],
            "director": movie["director"],
            "overview": movie["overview"],
            "year": movie["year"],
            "rating": movie["rating"],
            "vote_count": movie["vote_count"],
            "runtime": movie["runtime"],
            "genres": "/".join(movie["genres"]),
            "keywords": "/".join(movie["keywords"]),
            "actors": "|".join([a["name"] for a in movie['
            "reviews": "|||".join(movie["reviews"]),
        })
    return pd.DataFrame(rows)


# ============================================================
# 主程序
# ============================================================
if __name__ == "__main__":
    # 采集 5 页热门电影（约 100 部）
    movies = collect_movies(total_pages=5)

    # 保存原始数据
    save_movies(movies)

    # 转换为 DataFrame 并保存为 CSV
    df = movies_to_dataframe(movies)
    csv_path = os.path.join(OUTPUT_DIR, "movies_raw.csv")
    df.to_csv(csv_path, index=False, encoding="utf-8-sig")
    print(f"💾 CSV 已保存到：{csv_path}")

    # 预览数据
    print(f"\n📋 数据预览：")
    print(df[["title", "director", "year", "rating", "genr
```

注意事项： – TMDB 的 API 有频率限制（约 40 次/10秒），代码中已添加了 `time.sleep(0.3)` 来应对 – 如果你暂时不想注册 TMDB API，我在后面提供了一份可以直接使用的示例数据

12.2.2 使用示例数据（无需 API Key）

如果你不想花时间申请 API Key，可以直接使用下面这份示例数据。我精心整理了 30 部经典电影的数据：

```
"""
生成示例电影数据（无需 API Key）
直接运行即可获得一份包含 30 部经典电影的 CSV 文件
"""

import pandas as pd
import os

os.makedirs("data/processed", exist_ok=True)

movies_data = [
    {
        "id": 1, "title": "流浪地球", "director": "郭帆",
        "year": 2019, "rating": 7.9, "runtime": 125,
        "genres": "科幻/灾难",
        "overview": "太阳即将毁灭，人类在地球表面建造出巨大的推进
        "actors": "吴京/吴孟达/李光洁/赵今麦",
        "keywords": "太空/末日/拯救/科幻"
    },
    {
        "id": 2, "title": "哪吒之魔童降世", "director": "饺子
        "year": 2019, "rating": 8.4, "runtime": 110,
        "genres": "动画/奇幻",
        "overview": "天地灵气孕育出一颗能量巨大的混元珠，元始天尊
        "actors": "吕艳婷/囧森瑟夫/瀚墨/陈浩",
        "keywords": "神话/成长/命运/动画"
    },
    {
        "id": 3, "title": "霸王别姬", "director": "陈凯歌",
        "year": 1993, "rating": 9.6, "runtime": 171,
        "genres": "剧情/爱情",
        "overview": "段小楼和程蝶衣是一对打小一起长大的师兄弟，两
        "actors": "张国荣/张丰毅/巩俐/葛优",
        "keywords": "京剧/历史/人生/经典"
```

```
    },
    {
        "id": 4, "title": "肖申克的救赎", "director": "弗兰克
        "year": 1994, "rating": 9.7, "runtime": 142,
        "genres": "剧情/犯罪",
        "overview": "银行家安迪被冤枉杀害了妻子及其情人，被判处无
        "actors": "蒂姆·罗宾斯/摩根·弗里曼/鲍勃·冈顿",
        "keywords": "监狱/自由/希望/经典"
    },
    {
        "id": 5, "title": "阿甘正传", "director": "罗伯特·泽
        "year": 1994, "rating": 9.5, "runtime": 142,
        "genres": "剧情/爱情",
        "overview": "阿甘是一个智商只有75的低能儿，但他的单纯和善
        "actors": "汤姆·汉克斯/罗宾·怀特/加里·西尼斯",
        "keywords": "励志/人生/美国/经典"
    },
    {
        "id": 6, "title": "千与千寻", "director": "宫崎骏",
        "year": 2001, "rating": 9.4, "runtime": 125,
        "genres": "动画/奇幻",
        "overview": "千寻和爸爸妈妈一同驱车前往新家，在郊外的小路
        "actors": "柊瑠美/入野自由/夏木真理",
        "keywords": "宫崎骏/成长/奇幻/动画"
    },
    {
        "id": 7, "title": "泰坦尼克号", "director": "詹姆斯·
        "year": 1997, "rating": 9.4, "runtime": 194,
        "genres": "剧情/爱情",
        "overview": "1912年4月10日，号称'世界工业史上的奇迹'的豪
        "actors": "莱昂纳多·迪卡普里奥/凯特·温斯莱特",
        "keywords": "爱情/灾难/经典/沉船"
    },
    {
        "id": 8, "title": "盗梦空间", "director": "克里斯托弗
```

```
        "year": 2010, "rating": 9.3, "runtime": 148,
        "genres": "科幻/悬疑/动作",
        "overview": "道姆·柯布是一位经验老道的窃贼，能够潜入人们的
        "actors": "莱昂纳多·迪卡普里奥/约瑟夫·高登/艾伦·佩吉",
        "keywords": "梦境/悬疑/科幻/烧脑"
    },
    {
        "id": 9, "title": "星际穿越", "director": "克里斯托弗
        "year": 2014, "rating": 9.4, "runtime": 169,
        "genres": "科幻/冒险/剧情",
        "overview": "近未来的地球黄沙遍野，粮食作物相继灭绝。前NA
        "actors": "马修·麦康纳/安妮·海瑟薇/杰西卡·查斯坦",
        "keywords": "太空/虫洞/时间/亲情"
    },
    {
        "id": 10, "title": "让子弹飞", "director": "姜文",
        "year": 2010, "rating": 9.0, "runtime": 132,
        "genres": "剧情/喜剧/动作",
        "overview": "民国年间，花钱捐得县长的马邦德携妻及随从走马
        "actors": "姜文/葛优/周润发/刘嘉玲",
        "keywords": "民国/黑色幽默/经典/剧情"
    },
    {
        "id": 11, "title": "疯狂动物城", "director": "拜伦·霍
        "year": 2016, "rating": 9.2, "runtime": 108,
        "genres": "动画/喜剧/冒险",
        "overview": "在动物城中，无论是大象还是蚂蚁都和谐共处。兔
        "actors": "金妮弗·古德温/杰森·贝特曼",
        "keywords": "动画/迪士尼/偏见/梦想"
    },
    {
        "id": 12, "title": "我不是药神", "director": "文牧野"
        "year": 2018, "rating": 9.0, "runtime": 117,
        "genres": "剧情/喜剧",
        "overview": "普通中年男子程勇经营着一家保健品店，失意又失
```

```
        "actors": "徐峥/王传君/周一围/谭卓",
        "keywords": "现实/医疗/人性/社会"
    },
    {
        "id": 13, "title": "黑客帝国", "director": "沃卓斯基
        "year": 1999, "rating": 9.1, "runtime": 136,
        "genres": "科幻/动作",
        "overview": "不久的将来，网络黑客尼奥对这个看似正常的世界
        "actors": "基努·里维斯/劳伦斯·菲什伯恩/凯瑞-安·莫斯",
        "keywords": "虚拟现实/科幻/哲学/经典"
    },
    {
        "id": 14, "title": "大话西游之大圣娶亲", "director":
        "year": 1995, "rating": 9.2, "runtime": 99,
        "genres": "喜剧/奇幻/爱情",
        "overview": "至尊宝被月光宝盒带回到五百年前，遇见紫霞仙子
        "actors": "周星驰/朱茵/莫文蔚/吴孟达",
        "keywords": "经典/喜剧/爱情/西游"
    },
    {
        "id": 15, "title": "活着", "director": "张艺谋",
        "year": 1994, "rating": 9.3, "runtime": 133,
        "genres": "剧情/历史",
        "overview": "福贵是一个富家少爷，好赌成性，终于在一夜之间
        "actors": "葛优/巩俐/姜武",
        "keywords": "历史/人生/苦难/经典"
    },
    {
        "id": 16, "title": "楚门的世界", "director": "彼得·威
        "year": 1998, "rating": 9.3, "runtime": 103,
        "genres": "剧情/科幻",
        "overview": "楚门从出生起就生活在一个巨大的摄影棚中，他身
        "actors": "金·凯瑞/劳拉·琳妮/艾德·哈里斯",
        "keywords": "真人秀/自由/媒体/哲学"
    },
```

```
    {
        "id": 17, "title": "功夫", "director": "周星驰",
        "year": 2004, "rating": 8.6, "runtime": 99,
        "genres": "喜剧/动作",
        "overview": "二十年代的中国，街头混混阿星一心想加入黑帮出
        "actors": "周星驰/元秋/元华/黄圣依",
        "keywords": "功夫/喜剧/周星驰/经典"
    },
    {
        "id": 18, "title": "教父", "director": "弗朗西斯·科波
        "year": 1972, "rating": 9.3, "runtime": 175,
        "genres": "剧情/犯罪",
        "overview": "柯里昂家族是纽约五大黑帮之一。教父维托·柯里昂
        "actors": "马龙·白兰度/阿尔·帕西诺/詹姆斯·凯恩",
        "keywords": "黑帮/家族/权力/经典"
    },
    {
        "id": 19, "title": "龙猫", "director": "宫崎骏",
        "year": 1988, "rating": 9.2, "runtime": 86,
        "genres": "动画/奇幻",
        "overview": "小月和小梅两姐妹跟随父亲搬到了乡下居住。在这
        "actors": "日高法子/的场洋子/糸井重里",
        "keywords": "宫崎骏/童年/奇幻/温馨"
    },
    {
        "id": 20, "title": "流浪地球2", "director": "郭帆",
        "year": 2023, "rating": 8.3, "runtime": 173,
        "genres": "科幻/灾难/冒险",
        "overview": "太阳急速老化，人类启动'移山计划'，在地球表面
        "actors": "吴京/刘德华/李雪健/沙溢",
        "keywords": "太空/末日/AI/科幻"
    },
    {
        "id": 21, "title": "满江红", "director": "张艺谋",
        "year": 2023, "rating": 7.0, "runtime": 157,
```

```
        "genres": "剧情/悬疑/喜剧",
        "overview": "南宋时期，秦桧率兵与金国会谈。会谈前夜，金国
        "actors": "沈腾/易烊千玺/张译/雷佳音",
        "keywords": "悬疑/历史/反转/喜剧"
    },
    {
        "id": 22, "title": "你好，李焕英", "director": "贾玲"
        "year": 2021, "rating": 7.8, "runtime": 128,
        "genres": "喜剧/剧情",
        "overview": "贾晓玲在经历了人生大起大落后，情绪崩溃下意外
        "actors": "贾玲/张小斐/沈腾/陈赫",
        "keywords": "穿越/亲情/喜剧/感动"
    },
    {
        "id": 23, "title": "无间道", "director": "刘伟强",
        "year": 2002, "rating": 9.3, "runtime": 101,
        "genres": "剧情/犯罪/悬疑",
        "overview": "警方和黑帮都在对方内部安插了卧底。警察卧底刘
        "actors": "刘德华/梁朝伟/黄秋生/曾志伟",
        "keywords": "卧底/悬疑/警匪/经典"
    },
    {
        "id": 24, "title": "长津湖", "director": "陈凯歌",
        "year": 2021, "rating": 7.2, "runtime": 176,
        "genres": "战争/历史/剧情",
        "overview": "抗美援朝战争中，中国人民志愿军第九兵团在极寒
        "actors": "吴京/易烊千玺/段奕宏/朱亚文",
        "keywords": "战争/历史/爱国/抗美援朝"
    },
    {
        "id": 25, "title": "寄生虫", "director": "奉俊昊",
        "year": 2019, "rating": 8.8, "runtime": 132,
        "genres": "剧情/悬疑/惊悚",
        "overview": "住在半地下室的金基泽一家四口全是无业游民。长
        "actors": "宋康昊/李善均/赵汝贞/崔宇植",
```

```
        "keywords": "阶级/社会/悬疑/奥斯卡"
    },
    {
        "id": 26, "title": "少年派的奇幻漂流", "director": "
        "year": 2012, "rating": 9.1, "runtime": 127,
        "genres": "剧情/冒险/奇幻",
        "overview": "少年派的父亲经营着一家动物园。在一次举家迁徙
        "actors": "苏拉·沙玛/伊尔凡·可汗/塔布",
        "keywords": "信仰/奇幻/生存/视觉"
    },
    {
        "id": 27, "title": "蜘蛛侠：平行宇宙", "director": "
        "year": 2018, "rating": 8.6, "runtime": 117,
        "genres": "动画/动作/科幻",
        "overview": "迈尔斯·莫拉莱斯是一个普通的中学生，一次意外被
        "actors": "沙梅克·摩尔/杰克·约翰逊/海莉·斯坦菲尔德",
        "keywords": "超级英雄/平行宇宙/动画/漫威"
    },
    {
        "id": 28, "title": "战狼2", "director": "吴京",
        "year": 2017, "rating": 7.1, "runtime": 123,
        "genres": "动作/战争",
        "overview": "脱下军装的冷锋被卷入了一场非洲国家的叛乱。本
        "actors": "吴京/弗兰克·格里罗/吴刚/张翰",
        "keywords": "动作/军事/爱国/英雄"
    },
    {
        "id": 29, "title": "消失的她", "director": "崔睿",
        "year": 2023, "rating": 6.2, "runtime": 122,
        "genres": "悬疑/犯罪",
        "overview": "何非的妻子李木子在东南亚失踪，何非报警后苦寻
        "actors": "朱一龙/倪妮/文咏珊/杜江",
        "keywords": "悬疑/犯罪/反转/东南亚"
    },
    {
```

```
        "id": 30, "title": "人生大事", "director": "刘江江",
        "year": 2022, "rating": 7.3, "runtime": 112,
        "genres": "剧情/喜剧",
        "overview": "殡葬师莫三妹在一次出殡中遇到了孤儿武小文。小
        "actors": "朱一龙/杨恩又/王戈/刘陆",
        "keywords": "殡葬/温情/成长/生活"
    }
]

df = pd.DataFrame(movies_data)
df.to_csv("data/processed/movies_clean.csv", index=False,
print(f"✅ 已保存 {len(df)} 部电影数据到 data/processed/movie
print(df[["title", "director", "year", "rating", "genres"]
```

12.3 信息抽取：用 LLM 提取结构化信息

原始的电影描述是自由文本，我们需要用大语言模型（LLM）从中抽取结构化的实体和关系。

```
"""
LLM 信息抽取模块

从电影描述中抽取:
- 人物（导演、演员）
- 类型标签
- 主题关键词
- 情感分析
"""

import json
import os
from typing import List, Dict, Optional

# ============================================================
# 方式一：使用 OpenAI API（推荐，效果最好）
# ============================================================
class OpenAIExtractor:
    """
    使用 OpenAI GPT 模型进行信息抽取

    优点：准确率高，支持复杂抽取任务
    缺点：需要 API Key，有调用费用
    """

    def __init__(self, api_key=None, model="gpt-3.5-turbo"
        """
        初始化 OpenAI 客户端

        参数:
            api_key: OpenAI API 密钥（也可以从环境变量读取）
            model: 使用的模型名称
        """
        from openai import OpenAI
```

```
        self.client = OpenAI(
            api_key=api_key or os.environ.get("OPENAI_API_
        )
        self.model = model

    def extract_movie_info(self, title, overview, director
        """
        从电影描述中抽取结构化信息

        参数:
            title: 电影名称
            overview: 电影简介
            director: 导演（如果已知）

        返回:
            包含抽取结果的字典
        """
        prompt = f"""你是一个专业的电影知识抽取助手。请从以下电影

电影名称：{title}
导演：{director}
简介：{overview}

请以 JSON 格式返回以下字段：
{{
    "themes": ["主题词1", "主题词2", ...],        // 电影的主
    "emotions": ["情感标签1", ...],               // 电影传达
    "setting": "故事发生的时间/地点背景",
    "plot_summary": "一句话概括剧情（20字以内）",
    "target_audience": "目标受众描述",
    "similar_movies_hint": "可能类似的电影类型描述"
}}

请直接返回 JSON，不要添加其他文字。"""
```

```
        response = self.client.chat.completions.create(
            model=self.model,
            messages=[
                {"role": "system", "content": "你是一个精准的
                {"role": "user", "content": prompt}
            ],
            temperature=0.1,  # 低温度 → 更确定性的输出
            max_tokens=500
        )

        # 解析 LLM 返回的 JSON
        content = response.choices[0].message.content.stri
        # 处理可能的 markdown 代码块包裹
        if content.startswith("```"):
            content = content.split("```")[1]
            if content.startswith("json"):
                content = content[4:]

        try:
            result = json.loads(content)
            result["source_title"] = title
            return result
        except json.JSONDecodeError:
            print(f"  ⚠️ JSON 解析失败，电影：{title}")
            return {"source_title": title, "error": "JSON

    def extract_relationships(self, movie_title, overview)
        """
        从描述中抽取人物关系

        返回人物之间的关系列表
        """
        prompt = f"""请从以下电影描述中抽取人物之间的关系。
```

```
电影：{movie_title}
描述：{overview}

以 JSON 数组返回：
[
    {{
        "person1": "人物1名称",
        "person2": "人物2名称",
        "relationship": "关系类型（如：父子、恋人、敌人、师徒等）
    }}
]

如果描述中没有明确的人物关系，返回空数组 []。"""

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[{"role": "user", "content": prompt}]
            temperature=0.1,
            max_tokens=300
        )

        content = response.choices[0].message.content.stri
        if content.startswith("```"):
            content = content.split("```")[1]
            if content.startswith("json"):
                content = content[4:]

        try:
            return json.loads(content)
        except json.JSONDecodeError:
            return []


# ============================================================
# 方式二：使用规则抽取（无需 API，适合简单场景）
```

```
# ============================================================
class RuleBasedExtractor:
    """
    基于规则的信息抽取器

    优点：无需 API，速度快，完全离线
    缺点：抽取能力有限，依赖预定义规则
    """

    def __init__(self):
        # 预定义的情感关键词映射
        self.emotion_keywords = {
            "感动": ["泪目", "感动", "温情", "温暖", "亲情", "
            "紧张": ["悬疑", "紧张", "危险", "追杀", "逃亡", "
            "欢乐": ["喜剧", "搞笑", "幽默", "欢乐", "开心", "
            "震撼": ["震撼", "壮观", "史诗", "宏大", "视觉", "
            "悲伤": ["悲剧", "离别", "死亡", "失去", "孤独", "
            "热血": ["热血", "奋斗", "拼搏", "战斗", "英雄", "
        }

        # 预定义的主题关键词映射
        self.theme_keywords = {
            "科幻": ["太空", "未来", "机器人", "外星", "科技",
            "成长": ["成长", "蜕变", "青春", "梦想", "追寻", "
            "爱情": ["爱情", "恋人", "相遇", "相爱", "感情", "
            "家庭": ["家庭", "父母", "孩子", "亲情", "团聚", "
            "社会": ["社会", "阶级", "贫富", "现实", "制度", "
            "自由": ["自由", "解放", "逃离", "束缚", "打破", "
        }

    def extract_movie_info(self, title, overview, director
        """基于规则抽取电影信息"""
        result = {
            "source_title": title,
            "themes": [],
```

```
            "emotions": [],
            "setting": "未识别",
            "plot_summary": overview[:30] + "..." if len(o
            "target_audience": "通用观众",
        }

        # 抽取主题
        for theme, keywords in self.theme_keywords.items()
            for kw in keywords:
                if kw in overview:
                    result["themes"].append(theme)
                    break  # 每个主题只匹配一次

        # 抽取情感
        for emotion, keywords in self.emotion_keywords.ite
            for kw in keywords:
                if kw in overview:
                    result["emotions"].append(emotion)
                    break

        # 去重
        result["themes"] = list(set(result["themes"]))
        result["emotions"] = list(set(result["emotions"]))

        # 如果没抽取到，给默认值
        if not result["themes"]:
            result["themes"] = ["综合"]
        if not result["emotions"]:
            result["emotions"] = ["未知"]

        return result


# ============================================================
# 批量抽取
```

```
# ============================================================
def batch_extract(movies_df, extractor, use_llm=True):
    """
    对 DataFrame 中的所有电影进行批量信息抽取

    参数:
        movies_df: 电影数据 DataFrame
        extractor: 抽取器实例 (OpenAI 或规则)
        use_llm: 是否使用 LLM

    返回:
        包含抽取结果的列表
    """
    results = []

    for idx, row in movies_df.iterrows():
        print(f"  正在抽取 [{idx+1}/{len(movies_df)}]: {row

        info = extractor.extract_movie_info(
            title=row["title"],
            overview=row.get("overview", ""),
            director=row.get("director", "")
        )
        results.append(info)

    print(f"\n✅ 信息抽取完成! 共处理 {len(results)} 部电影")
    return results


# ============================================================
# 主程序
# ============================================================
if __name__ == "__main__":
    import pandas as pd
```

```
    # 加载电影数据
    df = pd.read_csv("data/processed/movies_clean.csv")

    # 选择抽取方式
    # 如果有 OpenAI API Key，使用 LLM；否则使用规则抽取
    if os.environ.get("OPENAI_API_KEY"):
        print("🤖 使用 OpenAI 进行信息抽取...")
        extractor = OpenAIExtractor()
    else:
        print("🔪 未检测到 OPENAI_API_KEY，使用规则抽取...")
        extractor = RuleBasedExtractor()

    # 执行批量抽取
    extracted = batch_extract(df, extractor)

    # 保存抽取结果
    with open("data/processed/extracted_info.json", "w", e
        json.dump(extracted, f, ensure_ascii=False, indent
    print("💾 抽取结果已保存到 data/processed/extracted_info

    # 预览
    print("\n📋 抽取结果预览：")
    for item in extracted[:3]:
        print(f"  {item['source_title']}")
        print(f"    主题：{item.get('themes', [])}")
        print(f"    情感：{item.get('emotions', [])}")
        print()
```

12.4 Neo4j 导入：构建电影知识图谱

数据准备好了，现在让我们把它导入 Neo4j，构建一个真正的知识图谱。

12.4.1 图模型设计

我们的图模型包含以下节点和关系：

```
节点类型：
- Movie（电影）：title, year, rating, overview, runtime
- Person（人物）：name, role（actor/director）
- Genre（类型）：name
- Theme（主题）：name

关系类型：
- (:Movie)-[:DIRECTED_BY]->(:Person)      # 电影由某人执导
- (:Movie)-[:STARRING]->(:Person)          # 电影由某人主演
- (:Movie)-[:BELONGS_TO]->(:Genre)         # 电影属于某类型
- (:Movie)-[:HAS_THEME]->(:Theme)          # 电影包含某主题
- (:Person)-[:COOPERATED_WITH]->(:Person)  # 人物之间的合作关
```

12.4.2 完整导入脚本

```
"""
Neo4j 图数据库导入脚本

构建完整的电影知识图谱:
Movie - DIRECTED_BY -> Person
Movie - STARRING -> Person
Movie - BELONGS_TO -> Genre
Movie - HAS_THEME -> Theme
"""

import pandas as pd
import json
from neo4j import GraphDatabase
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)
logger = logging.getLogger(__name__)


class MovieGraphBuilder:
    """电影知识图谱构建器"""

    def __init__(self, uri="bolt://localhost:7687",
                 user="neo4j", password="your_password"):
        """初始化 Neo4j 连接"""
        self.driver = GraphDatabase.driver(uri, auth=(user
        logger.info("✅ Neo4j 连接成功")

    def setup_constraints(self):
        """
        创建唯一性约束和索引

        这些约束确保了:
        1. 同一 title 的电影不会重复创建
```

```
        2. 同一 name 的人物不会重复创建
        3. MERGE 操作会自动利用索引，大幅提升导入速度
        """
        constraints = [
            # 节点唯一性约束
            "CREATE CONSTRAINT movie_unique IF NOT EXISTS
            "CREATE CONSTRAINT person_unique IF NOT EXISTS
            "CREATE CONSTRAINT genre_unique IF NOT EXISTS
            "CREATE CONSTRAINT theme_unique IF NOT EXISTS

            # 额外索引（用于加速查询）
            "CREATE INDEX movie_year IF NOT EXISTS FOR (m:
            "CREATE INDEX movie_rating IF NOT EXISTS FOR (
        ]

        with self.driver.session() as session:
            for c in constraints:
                try:
                    session.run(c)
                except Exception as e:
                    logger.warning(f"约束/索引提示：{e}")

        logger.info("✅ 约束和索引已创建")

    def clear_database(self):
        """清空数据库（谨慎使用！）"""
        with self.driver.session() as session:
            session.run("MATCH (n) DETACH DELETE n")
        logger.info("⚠️ 数据库已清空")

    def import_movies(self, df):
        """
        导入电影节点

        使用 MERGE 而不是 CREATE，确保幂等性（重复运行不会产生重复
```

```
        """
        movies = df.to_dict("records")

        def _import(tx, batch):
            tx.run("""
                UNWIND $batch AS movie
                MERGE (m:Movie {title: movie.title})
                SET m.year = toInteger(movie.year),
                    m.rating = toFloat(movie.rating),
                    m.overview = movie.overview,
                    m.runtime = toInteger(coalesce(movie.r
            """, batch=batch)

        batch_size = 100
        for i in range(0, len(movies), batch_size):
            batch = movies[i:i + batch_size]
            with self.driver.session() as session:
                session.execute_write(_import, batch)
            logger.info(f"  电影节点: {min(i + batch_size, 

        logger.info(f"✅ 导入 {len(movies)} 部电影节点")

    def import_directors(self, df):
        """
        导入导演节点并创建 DIRECTED_BY 关系
        """
        directors = df[df["director"].notna()][["title", "

        def _import(tx, batch):
            # 创建导演节点
            tx.run("""
                UNWIND $batch AS item
                MERGE (p:Person {name: item.director})
                SET p.role = 'director'
            """, batch=batch)
```

```
            # 创建关系
            tx.run("""
                UNWIND $batch AS item
                MATCH (m:Movie {title: item.title})
                MATCH (p:Person {name: item.director})
                MERGE (m)-[:DIRECTED_BY]->(p)
            """, batch=batch)

        with self.driver.session() as session:
            session.execute_write(_import, directors)

        logger.info(f"✅ 导入 {len(directors)} 条导演关系")

    def import_actors(self, df):
        """
        导入演员节点并创建 STARRING 关系

        actors 字段格式："演员1/演员2/演员3"
        """
        all_actor_rels = []
        for _, row in df.iterrows():
            actors_str = row.get("actors", "")
            if pd.isna(actors_str):
                continue
            for actor_name in str(actors_str).split("/"):
                actor_name = actor_name.strip()
                if actor_name:
                    all_actor_rels.append({
                        "title": row["title"],
                        "actor": actor_name
                    })

        def _import(tx, batch):
            # 创建演员节点
            tx.run("""
```

```
                UNWIND $batch AS item
                MERGE (p:Person {name: item.actor})
                SET p.role = coalesce(p.role, 'actor')
            """, batch=batch)
            # 创建 STARRING 关系
            tx.run("""
                UNWIND $batch AS item
                MATCH (m:Movie {title: item.title})
                MATCH (p:Person {name: item.actor})
                MERGE (m)-[:STARRING]->(p)
            """, batch=batch)

        batch_size = 200
        for i in range(0, len(all_actor_rels), batch_size)
            batch = all_actor_rels[i:i + batch_size]
            with self.driver.session() as session:
                session.execute_write(_import, batch)

        logger.info(f"✅ 导入 {len(all_actor_rels)} 条演员关

    def import_genres(self, df):
        """导入类型节点和 BELONGS_TO 关系"""
        genre_rels = []
        for _, row in df.iterrows():
            genres_str = row.get("genres", "")
            if pd.isna(genres_str):
                continue
            for genre in str(genres_str).split("/"):
                genre = genre.strip()
                if genre:
                    genre_rels.append({
                        "title": row["title"],
                        "genre": genre
                    })
```

```
        def _import(tx, batch):
            tx.run("""
                UNWIND $batch AS item
                MERGE (g:Genre {name: item.genre})
            """, batch=batch)
            tx.run("""
                UNWIND $batch AS item
                MATCH (m:Movie {title: item.title})
                MATCH (g:Genre {name: item.genre})
                MERGE (m)-[:BELONGS_TO]->(g)
            """, batch=batch)

        with self.driver.session() as session:
            session.execute_write(_import, genre_rels)

        logger.info(f"✅ 导入 {len(genre_rels)} 条类型关系")

    def import_themes(self, extracted_info):
        """
        导入主题节点和 HAS_THEME 关系

        参数:
            extracted_info: LLM 抽取的主题信息列表
        """
        theme_rels = []
        for info in extracted_info:
            title = info.get("source_title", "")
            themes = info.get("themes", [])
            for theme in themes:
                if theme and theme != "综合" and theme != '
                    theme_rels.append({
                        "title": title,
                        "theme": theme
                    })
```

```
        def _import(tx, batch):
            tx.run("""
                UNWIND $batch AS item
                MERGE (t:Theme {name: item.theme})
            """, batch=batch)
            tx.run("""
                UNWIND $batch AS item
                MATCH (m:Movie {title: item.title})
                MATCH (t:Theme {name: item.theme})
                MERGE (m)-[:HAS_THEME]->(t)
            """, batch=batch)

        if theme_rels:
            with self.driver.session() as session:
                session.execute_write(_import, theme_rels)
            logger.info(f"✅ 导入 {len(theme_rels)} 条主题关
        else:
            logger.info("ℹ️ 没有主题数据需要导入")

    def build_cooperation_network(self):
        """
        构建演员之间的合作网络

        逻辑：如果两个演员出现在同一部电影中，则创建 COOPERATED_W
        这是一个纯 Cypher 操作，不需要 Python 数据
        """
        with self.driver.session() as session:
            session.run("""
                MATCH (m:Movie)-[:STARRING]->(a1:Person),
                      (m)-[:STARRING]->(a2:Person)
                WHERE id(a1) < id(a2)
                MERGE (a1)-[c:COOPERATED_WITH]->(a2)
                ON CREATE SET c.movie_count = 1
                ON MATCH SET c.movie_count = c.movie_count
            """)
```

```
        logger.info("✅ 演员合作网络已构建")

    def get_stats(self):
        """获取图谱统计信息"""
        with self.driver.session() as session:
            stats = {}
            for label in ["Movie", "Person", "Genre", "The
                result = session.run(f"MATCH (n:{label}) R
                stats[label] = result.single()["cnt"]

            for rel_type in ["DIRECTED_BY", "STARRING", "B
                result = session.run(f"MATCH ()-[r:{rel_ty
                stats[rel_type] = result.single()["cnt"]

        return stats

    def close(self):
        self.driver.close()
        logger.info("Neo4j 连接已关闭")


# ============================================================
# 主程序
# ============================================================
if __name__ == "__main__":
    # 加载数据
    df = pd.read_csv("data/processed/movies_clean.csv")

    # 加载抽取结果（如果存在）
    extracted_info = []
    try:
        with open("data/processed/extracted_info.json", "r
            extracted_info = json.load(f)
    except FileNotFoundError:
        logger.warning("未找到抽取结果文件，跳过主题导入")
```

```
# 构建图谱
builder = MovieGraphBuilder(
    uri="bolt://localhost:7687",
    user="neo4j",
    password="your_password"
)

# 完整的构建流程
builder.clear_database()        # 清空旧数据（可选）
builder.setup_constraints()     # 创建约束
builder.import_movies(df)       # 导入电影
builder.import_directors(df)    # 导入导演
builder.import_actors(df)       # 导入演员
builder.import_genres(df)       # 导入类型
builder.import_themes(extracted_info)  # 导入主题
builder.build_cooperation_network()     # 构建合作网络

# 输出统计
stats = builder.get_stats()
print("\n📊 知识图谱统计：")
print(f"  节点：")
print(f"    电影：{stats['Movie']}")
print(f"    人物：{stats['Person']}")
print(f"    类型：{stats['Genre']}")
print(f"    主题：{stats['Theme']}")
print(f"  关系：")
print(f"    DIRECTED_BY：{stats['DIRECTED_BY']}")
print(f"    STARRING：{stats['STARRING']}")
print(f"    BELONGS_TO：{stats['BELONGS_TO']}")
print(f"    HAS_THEME：{stats['HAS_THEME']}")
print(f"    COOPERATED_WITH：{stats['COOPERATED_WITH']
```

```
builder.close()
```

12.5 Qdrant 向量化：电影语义嵌入

现在让我们把电影的文本描述转为向量，存入 Qdrant，为后续的语义搜索做准备。

```
"""
Qdrant 向量化导入脚本

将电影描述和影评转为向量，存入 Qdrant
"""

import pandas as pd
import numpy as np
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, P
import logging
import hashlib

logging.basicConfig(level=logging.INFO, format='%(asctime)
logger = logging.getLogger(__name__)


class MovieVectorizer:
    """电影向量化处理器"""

    def __init__(self, host="localhost", port=6333,
                 collection_name="movie_semantic"):
        self.client = QdrantClient(host=host, port=port)
        self.collection_name = collection_name
        self.model = None

    def load_model(self, model_name="paraphrase-multilingu
        """
        加载嵌入模型

        paraphrase-multilingual-MiniLM-L12-v2 的特点:
        - 支持中文和英文
        - 向量维度 384（轻量级）
        - 适合短文本的语义匹配
```

```
        """
        try:
            from sentence_transformers import SentenceTrar
            self.model = SentenceTransformer(model_name)
            self.vector_dim = self.model.get_sentence_embe
            logger.info(f"✅ 模型加载成功：{model_name}（{se
        except ImportError:
            logger.warning("sentence-transformers 未安装，使
            self.vector_dim = 384

    def ensure_collection(self):
        """创建 Collection（如不存在）"""
        collections = [c.name for c in self.client.get_col
        if self.collection_name not in collections:
            self.client.create_collection(
                collection_name=self.collection_name,
                vectors_config=VectorParams(
                    size=self.vector_dim,
                    distance=Distance.COSINE
                )
            )
            logger.info(f"✅ Collection '{self.collection_

            # 创建 Payload 索引（加速过滤搜索）
            self.client.create_payload_index(
                collection_name=self.collection_name,
                field_name="year",
                field_schema="integer"
            )
            self.client.create_payload_index(
                collection_name=self.collection_name,
                field_name="rating",
                field_schema="float"
            )
            self.client.create_payload_index(
```

```
                collection_name=self.collection_name,
                field_name="genre",
                field_schema="keyword"
            )
        else:
            logger.info(f"ℹ️ Collection '{self.collection_

    def encode_texts(self, texts):
        """将文本列表编码为向量列表"""
        if self.model:
            embeddings = self.model.encode(
                texts,
                show_progress_bar=True,
                batch_size=32,
                normalize_embeddings=True  # L2 归一化，提升
            )
            return embeddings.tolist()
        else:
            # 模拟向量（仅用于测试）
            return [np.random.rand(self.vector_dim).tolist

    def generate_point_id(self, text):
        """基于文本内容生成确定性的 Point ID"""
        hash_val = hashlib.md5(text.encode()).hexdigest()
        return int(hash_val[:12], 16)

    def vectorize_and_upload(self, df):
        """
        将电影数据向量化并上传到 Qdrant

        向量化策略：
        - 将标题、简介、类型组合成一段完整的描述文本
        - 对这段文本进行编码，得到一个综合向量
        """
        # 组合文本（将多个字段拼成一段描述）
```

```
        texts = []
        payloads = []

        for _, row in df.iterrows():
            # 构建丰富的描述文本
            description = f"{row['title']}。"
            if pd.notna(row.get("overview")):
                description += str(row["overview"])
            if pd.notna(row.get("genres")):
                description += f" 类型: {row['genres']}。"
            if pd.notna(row.get("director")):
                description += f" 导演: {row['director']}。

            texts.append(description)
            payloads.append({
                "title": str(row["title"]),
                "director": str(row.get("director", "")),
                "year": int(row.get("year", 0)),
                "rating": float(row.get("rating", 0)),
                "genre": str(row.get("genres", "")),
                "overview": str(row.get("overview", "")),
            })

        # 向量化
        logger.info(f"正在编码 {len(texts)} 条文本...")
        vectors = self.encode_texts(texts)

        # 构建 Points
        points = [
            PointStruct(
                id=self.generate_point_id(text),
                vector=vector,
                payload=payload
            )
            for text, vector, payload in zip(texts, vector
```

```
        ]

        # 批量上传
        batch_size = 50
        for i in range(0, len(points), batch_size):
            batch = points[i:i + batch_size]
            self.client.upsert(
                collection_name=self.collection_name,
                points=batch,
                wait=True
            )

        logger.info(f"✅ 已上传 {len(points)} 条向量到 '{sel

    def search_similar(self, query_text, limit=5, genre_fi
                       min_year=None, min_rating=None):
        """
        语义搜索：根据描述文本搜索相似电影

        参数:
            query_text: 查询文本（如"太空冒险类的电影"）
            limit: 返回结果数量
            genre_filter: 类型过滤
            min_year: 最小年份
            min_rating: 最低评分
        """
        # 编码查询文本
        query_vector = self.encode_texts([query_text])[0]

        # 构建过滤条件
        from qdrant_client.models import Filter, FieldCond

        filter_conditions = []
        if genre_filter:
            filter_conditions.append(
```

```
                FieldCondition(key="genre", match=MatchVal
            )
        if min_year:
            filter_conditions.append(
                FieldCondition(key="year", range=Range(gte
            )
        if min_rating:
            filter_conditions.append(
                FieldCondition(key="rating", range=Range(g
            )

        search_filter = Filter(must=filter_conditions) if

        # 执行搜索
        results = self.client.search(
            collection_name=self.collection_name,
            query_vector=query_vector,
            query_filter=search_filter,
            limit=limit
        )

        return results


# ============================================================
# 主程序
# ============================================================
if __name__ == "__main__":
    df = pd.read_csv("data/processed/movies_clean.csv")

    vectorizer = MovieVectorizer(
        host="localhost",
        port=6333,
        collection_name="movie_semantic"
    )
```

```
    vectorizer.load_model()
    vectorizer.ensure_collection()
    vectorizer.vectorize_and_upload(df)

    # 测试搜索
    print("\n🔍 语义搜索测试：")
    results = vectorizer.search_similar("太空冒险类的科幻电影
    for r in results:
        print(f"  [{r.score:.3f}] {r.payload['title']} - {
```

12.6 构建对话系统：LangChain 集成

这是最激动人心的部分——我们要让知识图谱”会说话”了！

```
"""
电影知识图谱对话系统

使用 LangChain 将用户的自然语言问题转换为：
1. Cypher 查询（从 Neo4j 获取结构化信息）
2. 语义搜索（从 Qdrant 获取相关内容）
3. 综合回答（LLM 整合信息生成回答）
"""

import os
from neo4j import GraphDatabase
from qdrant_client import QdrantClient


# ============================================================
# 图谱查询链（Graph Chain）
# ============================================================
class MovieGraphChain:
    """
    图谱查询链

    将自然语言转换为 Cypher 查询，从 Neo4j 获取结构化信息
    """

    def __init__(self, uri="bolt://localhost:7687",
                 user="neo4j", password="your_password",
                 openai_api_key=None):
        self.driver = GraphDatabase.driver(uri, auth=(user
        self.api_key = openai_api_key or os.environ.get("O

    def query_by_director(self, director_name):
        """查询某导演的所有电影"""
        with self.driver.session() as session:
            result = session.run("""
```

```
                MATCH (m:Movie)-[:DIRECTED_BY]->(p:Person
                RETURN m.title AS title, m.year AS year, m
                ORDER BY m.rating DESC
            """, name=director_name)
            return [record.data() for record in result]

    def query_by_genre(self, genre_name, limit=10):
        """按类型查询电影"""
        with self.driver.session() as session:
            result = session.run("""
                MATCH (m:Movie)-[:BELONGS_TO]->(g:Genre {r
                RETURN m.title AS title, m.year AS year, m
                ORDER BY m.rating DESC
                LIMIT $limit
            """, genre=genre_name, limit=limit)
            return [record.data() for record in result]

    def query_actor_movies(self, actor_name):
        """查询某演员参演的电影"""
        with self.driver.session() as session:
            result = session.run("""
                MATCH (m:Movie)-[:STARRING]->(p:Person {na
                RETURN m.title AS title, m.year AS year, m
                ORDER BY m.rating DESC
            """, name=actor_name)
            return [record.data() for record in result]

    def query_movie_detail(self, movie_title):
        """查询电影的完整信息"""
        with self.driver.session() as session:
            result = session.run("""
                MATCH (m:Movie {title: $title})
                OPTIONAL MATCH (m)-[:DIRECTED_BY]->(d:Pers
                OPTIONAL MATCH (m)-[:STARRING]->(a:Person)
                OPTIONAL MATCH (m)-[:BELONGS_TO]->(g:Genre
```

```
                OPTIONAL MATCH (m)-[:HAS_THEME]->(t:Theme)
                RETURN m.title AS title,
                       m.year AS year,
                       m.rating AS rating,
                       m.overview AS overview,
                       m.runtime AS runtime,
                       d.name AS director,
                       collect(DISTINCT a.name) AS actors,
                       collect(DISTINCT g.name) AS genres,
                       collect(DISTINCT t.name) AS themes
            """, title=movie_title)
            record = result.single()
            return record.data() if record else None

    def query_cooperation(self, person_name):
        """查询某人的合作网络"""
        with self.driver.session() as session:
            result = session.run("""
                MATCH (p:Person {name: $name})-[c:COOPERAT
                RETURN other.name AS collaborator,
                       c.movie_count AS movie_count
                ORDER BY movie_count DESC
            """, name=person_name)
            return [record.data() for record in result]

    def natural_language_to_cypher(self, question):
        """
        使用 LLM 将自然语言转换为 Cypher 查询

        这是 Graph Chain 的核心——让 LLM 理解用户的意图,
        生成正确的 Cypher 语句
        """
        if not self.api_key:
            return None
```

```
        from openai import OpenAI
        client = OpenAI(api_key=self.api_key)

        schema_description = """
        图数据库 Schema:

        节点标签和属性:
        - Movie: title(电影名), year(年份), rating(评分), ov
        - Person: name(人名), role(角色:actor/director)
        - Genre: name(类型名)
        - Theme: name(主题名)

        关系类型:
        - (Movie)-[:DIRECTED_BY]->(Person)  导演
        - (Movie)-[:STARRING]->(Person)     主演
        - (Movie)-[:BELONGS_TO]->(Genre)    类型
        - (Movie)-[:HAS_THEME]->(Theme)     主题
        - (Person)-[:COOPERATED_WITH]->(Person) 合作
        """

        prompt = f"""你是一个 Cypher 查询专家。根据以下图数据库

{schema_description}

规则:
1. 只返回 Cypher 查询语句，不要其他解释
2. 使用 MATCH 查询，不要修改数据
3. 返回有意义的字段别名
4. 如果问题无法用该 Schema 回答，返回空字符串

用户问题: {question}

Cypher 查询: """

        response = client.chat.completions.create(
```

```
            model="gpt-3.5-turbo",
            messages=[{"role": "user", "content": prompt}]
            temperature=0,
            max_tokens=300
        )

        cypher = response.choices[0].message.content.strip
        # 清理可能的 markdown 格式
        if cypher.startswith("```"):
            cypher = cypher.split("```")[1]
            if cypher.startswith("cypher"):
                cypher = cypher[6:]

        return cypher

    def execute_natural_language(self, question):
        """执行自然语言查询的完整流程"""
        # 步骤1：转换为 Cypher
        cypher = self.natural_language_to_cypher(question)
        if not cypher:
            return "抱歉，我无法理解你的问题。"

        print(f"  生成的 Cypher：{cypher}")

        # 步骤2：执行 Cypher
        try:
            with self.driver.session() as session:
                result = session.run(cypher)
                records = [record.data() for record in res

            if not records:
                return "没有找到相关结果。"

            return records
        except Exception as e:
```

```
            return f"查询执行失败：{e}"


# ============================================================
# 向量搜索链（Vector Chain）
# ============================================================
class MovieVectorChain:
    """
    向量搜索链

    将用户的查询转为语义搜索，从 Qdrant 获取相关电影
    """

    def __init__(self, host="localhost", port=6333,
                 collection_name="movie_semantic",
                 model_name="paraphrase-multilingual-MiniL
        self.client = QdrantClient(host=host, port=port)
        self.collection_name = collection_name

        try:
            from sentence_transformers import SentenceTrar
            self.model = SentenceTransformer(model_name)
        except ImportError:
            self.model = None

    def search(self, query, limit=5, genre=None, min_ratir
        """语义搜索电影"""
        if self.model is None:
            return []

        # 编码查询
        query_vector = self.model.encode([query])[0].tolis

        # 构建过滤条件
        from qdrant_client.models import Filter, FieldCond
```

```
        filters = []
        if genre:
            filters.append(FieldCondition(key="genre", mat
        if min_rating:
            filters.append(FieldCondition(key="rating", ra

        search_filter = Filter(must=filters) if filters el

        # 搜索
        results = self.client.search(
            collection_name=self.collection_name,
            query_vector=query_vector,
            query_filter=search_filter,
            limit=limit
        )

        return [
            {
                "title": r.payload["title"],
                "score": r.score,
                "overview": r.payload.get("overview", ""),
                "rating": r.payload.get("rating", 0),
                "year": r.payload.get("year", 0),
            }
            for r in results
        ]


# ============================================================
# 对话引擎 (ChatBot)
# ============================================================
class MovieChatbot:
    """
    电影知识图谱对话机器人
```

```
    整合图谱查询和语义搜索，通过 LLM 生成自然语言回答
    """

    def __init__(self, openai_api_key=None):
        self.api_key = openai_api_key or os.environ.get("O
        self.graph_chain = MovieGraphChain(openai_api_key=
        self.vector_chain = MovieVectorChain()
        self.conversation_history = []

    def _generate_response(self, question, context):
        """使用 LLM 根据上下文生成回答"""
        if not self.api_key:
            # 没有 API Key 时，直接返回原始数据
            return f"基于查询结果：\n{context}"

        from openai import OpenAI
        client = OpenAI(api_key=self.api_key)

        # 构建历史对话上下文
        history_text = ""
        for msg in self.conversation_history[-6:]:  # 最近
            history_text += f"用户：{msg['user']}\n助手：{ms

        prompt = f"""你是一个专业的电影推荐顾问，基于以下信息回答
回答要自然、友好，像和朋友聊天一样。
如果推荐电影，请给出简短的推荐理由。

对话历史：
{history_text}

相关数据：
{context}

用户问题：{question}
```

```
请用中文回答："""

        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[{"role": "user", "content": prompt}]
            temperature=0.7,
            max_tokens=500
        )

        return response.choices[0].message.content

    def chat(self, user_message):
        """
        处理用户消息的完整流程

        流程：
        1. 意图识别：判断用户想做什么
        2. 并行查询：同时查图谱和向量
        3. 整合回答：LLM 生成自然语言回答
        """
        print(f"\n🤖 处理中：{user_message}")

        context_parts = []

        # ----------------------------------------------------
        # 图谱查询（结构化信息）
        # ----------------------------------------------------
        try:
            graph_result = self.graph_chain.execute_natura
            if graph_result and graph_result != "没有找到相
                context_parts.append(f"图谱查询结果：{graph_
                print("  ✅ 图谱查询成功")
        except Exception as e:
            print(f"  ⚠️ 图谱查询失败：{e}")
```

```
        # ------------------------------------------------
        # 语义搜索（内容匹配）
        # ------------------------------------------------
        try:
            vector_results = self.vector_chain.search(user
            if vector_results:
                vector_context = "\n".join([
                    f"- {r['title']}（{r['year']}，评分{r['
                    for r in vector_results
                ])
                context_parts.append(f"语义搜索结果：\n{vect
                print("  ✅ 语义搜索成功")
        except Exception as e:
            print(f"  ⚠️ 语义搜索失败：{e}")

        # ------------------------------------------------
        # 整合回答
        # ------------------------------------------------
        full_context = "\n\n".join(context_parts) if conte

        response = self._generate_response(user_message, f

        # 记录对话历史
        self.conversation_history.append({
            "user": user_message,
            "assistant": response
        })

        return response


# ============================================================
# 测试对话系统
# ============================================================
```

```
if __name__ == "__main__":
    bot = MovieChatbot()

    # 测试用例
    test_questions = [
        "周星驰导演过哪些电影？",
        "推荐几部好看的科幻电影",
        "有没有类似《流浪地球》那种太空冒险的电影？",
        "评分最高的电影是哪部？",
        "吴京演过什么电影？",
    ]

    print("=" * 60)
    print("🎬 电影知识图谱对话系统测试")
    print("=" * 60)

    for q in test_questions:
        print(f"\n👤 用户：{q}")
        answer = bot.chat(q)
        print(f"🤖 助手：{answer}")
        print("-" * 40)
```

12.7 前端：Streamlit 可视化界面

最后一步——给我们的系统穿上一件漂亮的外衣！

```
"""
Streamlit 前端应用

功能:
1. 图谱浏览（交互式可视化）
2. 智能对话
3. 语义搜索
4. 图谱统计

运行方式: streamlit run app/streamlit_app.py
"""

import streamlit as st
import pandas as pd
from neo4j import GraphDatabase
from qdrant_client import QdrantClient
import json

# ============================================================
# 页面配置
# ============================================================
st.set_page_config(
    page_title="🎬 电影知识图谱",
    page_icon="🎬",
    layout="wide",
    initial_sidebar_state="expanded"
)

# ============================================================
# 自定义样式
# ============================================================
st.markdown("""
<style>
    .main-header {
```

```
        font-size: 2.5rem;
        font-weight: bold;
        text-align: center;
        margin-bottom: 1rem;
        color: #FF6B6B;
    }
    .sub-header {
        font-size: 1.2rem;
        text-align: center;
        color: #666;
        margin-bottom: 2rem;
    }
    .metric-card {
        background: linear-gradient(135deg, #667eea 0%, #7
        padding: 1.5rem;
        border-radius: 10px;
        color: white;
        text-align: center;
    }
</style>
""", unsafe_allow_html=True)


# ============================================================
# 初始化连接（使用缓存避免重复创建）
# ============================================================
@st.cache_resource
def get_neo4j_driver():
    return GraphDatabase.driver(
        "bolt://localhost:7687",
        auth=("neo4j", "your_password")
    )

@st.cache_resource
def get_qdrant_client():
```

```
    return QdrantClient(host="localhost", port=6333)


# ============================================================
# 侧边栏
# ============================================================
st.sidebar.markdown("# 🎬 电影知识图谱")
st.sidebar.markdown("---")

page = st.sidebar.radio(
    "功能导航",
    ["🏠 首页概览", "💬 智能对话", "🔍 语义搜索", "📊 图谱浏览"
)


# ============================================================
# 页面1：首页概览
# ============================================================
if page == "🏠 首页概览":
    st.markdown('<div class="main-header">🎬 电影知识图谱</d
                unsafe_allow_html=True)
    st.markdown('<div class="sub-header">基于 Neo4j + Qdra
                unsafe_allow_html=True)

    # 获取统计数据
    try:
        driver = get_neo4j_driver()
        with driver.session() as session:
            movie_count = session.run("MATCH (m:Movie) RET
            person_count = session.run("MATCH (p:Person) R
            genre_count = session.run("MATCH (g:Genre) RET
            rel_count = session.run("MATCH ()-[r]->() RETU

        # 统计卡片
        col1, col2, col3, col4 = st.columns(4)
```

```
        col1.metric("🎬 电影数量", movie_count)
        col2.metric("👤 人物数量", person_count)
        col3.metric("🏷️ 类型数量", genre_count)
        col4.metric("🔗 关系数量", rel_count)

    except Exception as e:
        st.error(f"无法连接到 Neo4j：{e}")

    st.markdown("---")
    st.markdown("""
    ### 🛠️ 技术栈

    | 组件 | 技术 |
    |------|------|
    | 图数据库 | Neo4j |
    | 向量数据库 | Qdrant |
    | 嵌入模型 | sentence-transformers |
    | 对话引擎 | LangChain + OpenAI |
    | 前端 | Streamlit |

    ### 📖 使用方法

    1. **智能对话**：在对话框中输入自然语言问题，系统会综合图谱和语
    2. **语义搜索**：输入描述性文本，找到语义最相似的电影
    3. **图谱浏览**：可视化浏览电影、人物、类型之间的关系网络
    4. **统计分析**：查看图谱的各类统计数据
    """)


# ============================================================
# 页面2：智能对话
# ============================================================
elif page == "💬 智能对话":
    st.header("💬 智能对话")
    st.markdown("输入你的问题，AI 将综合图谱知识和语义搜索给出回答
```

```
    # 快捷问题按钮
    st.markdown("**快捷提问：**")
    col1, col2, col3 = st.columns(3)
    with col1:
        q1 = st.button("推荐好看的科幻电影")
    with col2:
        q2 = st.button("周星驰的电影有哪些")
    with col3:
        q3 = st.button("评分最高的电影")

    # 用户输入
    user_input = st.text_input(
        "你的问题：",
        value="推荐好看的科幻电影" if q1 else "周星驰的电影有哪些
        placeholder="例如：推荐几部和《盗梦空间》类似的烧脑电影"
    )

    if st.button("发送", type="primary"):
        if user_input:
            with st.spinner("正在思考中..."):
                # 这里调用对话系统
                # 简化版：直接从 Neo4j 和 Qdrant 获取数据
                try:
                    driver = get_neo4j_driver()

                    # 图谱查询示例
                    with driver.session() as session:
                        result = session.run("""
                            MATCH (m:Movie)
                            WHERE m.rating >= 8.0
                            RETURN m.title AS title, m.rat
                            ORDER BY m.rating DESC
                            LIMIT 5
                        """)
```

```
                        records = [r.data() for r in resul

                    if records:
                        st.markdown("### 🎥 推荐电影")
                        for r in records:
                            st.markdown(f"- **{r['title']}
                    else:
                        st.info("暂无推荐结果")

                except Exception as e:
                    st.error(f"查询失败：{e}")


# ============================================================
# 页面3：语义搜索
# ============================================================
elif page == "🔍 语义搜索":
    st.header("🔍 语义搜索")
    st.markdown("输入描述性文本，找到语义最相似的电影")

    # 搜索输入
    col1, col2 = st.columns([3, 1])
    with col1:
        search_query = st.text_input(
            "搜索内容：",
            placeholder="例如：一部关于太空探索的震撼科幻片"
        )
    with col2:
        num_results = st.slider("结果数量", 3, 20, 5)

    # 过滤条件
    st.markdown("**筛选条件（可选）：**")
    fcol1, fcol2, fcol3 = st.columns(3)
    with fcol1:
        min_rating = st.number_input("最低评分", 0.0, 10.0,
```

```
    with fcol2:
        min_year = st.number_input("最早年份", 1900, 2024,

    if st.button("搜索", type="primary"):
        if search_query:
            with st.spinner("正在搜索..."):
                try:
                    client = get_qdrant_client()

                    # 简化版搜索（实际需要嵌入模型）
                    st.markdown(f"### 🔍 搜索 \"{search_qu

                    # 从 Neo4j 获取电影数据作为展示
                    driver = get_neo4j_driver()
                    with driver.session() as session:
                        result = session.run("""
                            MATCH (m:Movie)
                            WHERE m.rating >= $min_rating
                              AND m.year >= $min_year
                            RETURN m.title AS title, m.ove
                                   m.rating AS rating, m.y
                            ORDER BY m.rating DESC
                            LIMIT $limit
                        """, min_rating=min_rating, min_ye
                             limit=num_results)

                        for record in result:
                            data = record.data()
                            with st.container():
                                st.markdown(f"#### 🎬 {dat
                                st.markdown(f"⭐ 评分: **{d
                                st.markdown(f"{data.get('d
                                st.markdown("---")

                except Exception as e:
```

```
                    st.error(f"搜索失败：{e}")


# ============================================================
# 页面4：图谱浏览
# ============================================================
elif page == "📊 图谱浏览":
    st.header("📊 图谱浏览")
    st.markdown("可视化浏览电影知识图谱的关系网络")

    # 查询选项
    query_type = st.selectbox(
        "选择浏览模式：",
        ["按导演查看", "按类型查看", "按演员查看", "全局概览"]
    )

    driver = get_neo4j_driver()

    if query_type == "按导演查看":
        # 获取所有导演
        with driver.session() as session:
            result = session.run("""
                MATCH (p:Person)<-[:DIRECTED_BY]-(:Movie)
                RETURN DISTINCT p.name AS name
                ORDER BY name
            """)
            directors = [r["name"] for r in result]

        selected = st.selectbox("选择导演：", directors)

        if selected:
            with driver.session() as session:
                result = session.run("""
                    MATCH (m:Movie)-[:DIRECTED_BY]->(p:Per
                    OPTIONAL MATCH (m)-[:STARRING]->(a:Per
```

```
                    OPTIONAL MATCH (m)-[:BELONGS_TO]->(g:G
                    RETURN m.title AS title, m.year AS yea
                           collect(DISTINCT a.name) AS act
                           collect(DISTINCT g.name) AS ger
                """, name=selected)

                st.markdown(f"### 🎬 {selected} 的电影作品"
                for record in result:
                    data = record.data()
                    st.markdown(f"**{data['title']}** ({da
                    st.markdown(f"  演员: {', '.join(data['
                    st.markdown(f"  类型: {', '.join(data['
                    st.markdown("---")

    elif query_type == "按类型查看":
        with driver.session() as session:
            result = session.run("""
                MATCH (g:Genre)
                OPTIONAL MATCH (m:Movie)-[:BELONGS_TO]->(g
                RETURN g.name AS genre, count(m) AS movie_
                ORDER BY movie_count DESC
            """)
            genres = {r["genre"]: r["movie_count"] for r i

        # 类型分布柱状图
        genre_df = pd.DataFrame(list(genres.items()), colu
        st.bar_chart(genre_df.set_index("类型"))

        selected_genre = st.selectbox("选择类型: ", list(ge
        if selected_genre:
            with driver.session() as session:
                result = session.run("""
                    MATCH (m:Movie)-[:BELONGS_TO]->(g:Genr
                    RETURN m.title AS title, m.year AS yea
                    ORDER BY m.rating DESC
```

```
                    LIMIT 10
                """, genre=selected_genre)

                st.markdown(f"### 🎬 {selected_genre} 类型
                for record in result:
                    data = record.data()
                    st.markdown(f"- **{data['title']}** ({

    elif query_type == "全局概览":
        with driver.session() as session:
            result = session.run("""
                MATCH (n)
                RETURN labels(n)[0] AS label, count(n) AS
            """)
            node_stats = {r["label"]: r["count"] for r in

            result = session.run("""
                MATCH ()-[r]->()
                RETURN type(r) AS type, count(r) AS count
            """)
            rel_stats = {r["type"]: r["count"] for r in re

        col1, col2 = st.columns(2)
        with col1:
            st.markdown("### 节点统计")
            for label, count in node_stats.items():
                st.markdown(f"- **{label}**: {count} 个")
        with col2:
            st.markdown("### 关系统计")
            for rtype, count in rel_stats.items():
                st.markdown(f"- **{rtype}**: {count} 条")


# ============================================================
# 页面5：统计分析
```

```
# ============================================================
elif page == "📈 统计分析":
    st.header("📈 统计分析")

    driver = get_neo4j_driver()

    # 电影评分分布
    with driver.session() as session:
        result = session.run("""
            MATCH (m:Movie)
            WHERE m.rating > 0
            RETURN m.rating AS rating, count(m) AS count
            ORDER BY rating
        """)
        rating_dist = {float(r["rating"]): r["count"] for

    st.markdown("### 📊 评分分布")
    rating_df = pd.DataFrame(list(rating_dist.items()), co
    st.bar_chart(rating_df.set_index("评分"))

    # 各年代电影数量
    with driver.session() as session:
        result = session.run("""
            MATCH (m:Movie)
            WHERE m.year > 0
            WITH floor(m.year / 10) * 10 AS decade, count(
            RETURN decade, count
            ORDER BY decade
        """)
        decade_data = {f"{int(r['decade'])}s": r["count"]

    st.markdown("### 📊 各年代电影数量")
    decade_df = pd.DataFrame(list(decade_data.items()), co
    st.bar_chart(decade_df.set_index("年代"))
```

```
    # 最多产的导演
    with driver.session() as session:
        result = session.run("""
            MATCH (m:Movie)-[:DIRECTED_BY]->(p:Person)
            WITH p.name AS director, count(m) AS movie_cou
            WHERE movie_count >= 2
            RETURN director, movie_count, round(avg_rating
            ORDER BY movie_count DESC
            LIMIT 10
        """)

    st.markdown("### 🎬 最多产的导演（至少 2 部作品）")
    director_data = []
    for r in result:
        director_data.append({
            "导演": r["director"],
            "作品数": r["movie_count"],
            "平均评分": r["avg_rating"]
        })
    if director_data:
        st.table(pd.DataFrame(director_data))

    # 最活跃的演员
    with driver.session() as session:
        result = session.run("""
            MATCH (m:Movie)-[:STARRING]->(p:Person)
            WITH p.name AS actor, count(m) AS movie_count
            WHERE movie_count >= 2
            RETURN actor, movie_count
            ORDER BY movie_count DESC
            LIMIT 10
        """)

    st.markdown("### 🌟 最活跃的演员（至少参演 2 部）")
    actor_data = []
```

```
    for r in result:
        actor_data.append({
            "演员": r["actor"],
            "参演数": r["movie_count"]
        })
    if actor_data:
        st.table(pd.DataFrame(actor_data))
```

启动命令：

```
cd movie_knowledge_graph
streamlit run app/streamlit_app.py
```

浏览器会自动打开 `http://localhost:8501`，你就能看到完整的交互式界面了。

12.8 项目回顾与优化建议

12.8.1 我们完成了什么

回顾一下这个项目的完整旅程：

1. **项目设计**：确定了技术栈和系统架构
2. **数据采集**：通过 TMDB API 获取电影数据（或使用示例数据）
3. **信息抽取**：用 LLM/规则从自由文本中抽取结构化信息
4. **Neo4j 导入**：构建了 Movie–Person–Genre–Theme 的完整图谱
5. **Qdrant 向量化**：将电影描述转为语义向量
6. **对话系统**：LangChain 整合图谱查询和语义搜索
7. **Streamlit 前端**：交互式可视化界面

12.8.2 优化方向

这个基础版本已经可以工作了，但还有很多优化空间：

数据层面： – 增加更多电影数据（扩展到 1000+ 部） – 添加影评数据，丰富语义搜索的内容 – 引入用户评分数据，支持个性化推荐

图谱层面： – 添加更多关系类型（如”改编自”、”续集”、”翻拍”） – 引入时间维度（如”同年代电影”关系） – 构建用户画像节点，实现协同过滤推荐

搜索层面： – 实现混合搜索（结合关键词匹配和语义搜索） – 添加搜索结果的重排序（Reranking）机制 – 支持多轮对话的上下文理解

系统层面： – 添加缓存层（Redis）提升查询速度 – 实现增量更新（新电影自动入库） – 添加用户反馈机制（点赞/踩，改进推荐质量） – 部署到云服务器，提供在线访问

12.8.3 生产环境部署建议

```
# docker-compose.yml（生产环境部署参考）
version: '3.8'

services:
  neo4j:
    image: neo4j:5-community
    ports:
      - "7474:7474"
      - "7687:7687"
    environment:
      - NEO4J_AUTH=neo4j/your_secure_password
      - NEO4J_server_memory_heap_max__size=1G
    volumes:
      - neo4j_data:/data

  qdrant:
    image: qdrant/qdrant:latest
    ports:
      - "6333:6333"
    volumes:
      - qdrant_data:/qdrant/storage

  streamlit:
    build: .
    ports:
      - "8501:8501"
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - NEO4J_URI=bolt://neo4j:7687
      - QDRANT_HOST=qdrant
    depends_on:
      - neo4j
      - qdrant
```

```
volumes:
  neo4j_data:
  qdrant_data:
```

动手练习

综合练习：扩展你的电影知识图谱

目标： 在现有项目基础上，至少添加一个新功能。

可选方向（任选其一）：

1. **添加”用户评分”功能**
2. 在 Streamlit 中增加一个”我的评分”页面
3. 用户可以给电影打分
4. 将评分存入 Neo4j（创建 User 节点和 RATED 关系）
5. 基于用户历史评分推荐新电影
6. **添加”电影对比”功能**
7. 用户选择两部电影
8. 系统展示它们的共同演员、共同类型、评分对比等
9. **添加”观影路径”功能**
10. 通过图谱关系找到两部电影之间的”路径”
11. 例如：《流浪地球》→ 吴京 → 战狼2 → 吴刚 → 人民的名义

提示代码（方向1的骨架）：

```
# 在 Neo4j 中添加用户评分
def add_user_rating(driver, username, movie_title, rating)
    """记录用户对电影的评分"""
    with driver.session() as session:
        session.execute_write(lambda tx: tx.run("""
            MERGE (u:User {name: $username})
            MATCH (m:Movie {title: $movie_title})
            MERGE (u)-[r:RATED]->(m)
            SET r.score = toFloat($rating),
                r.timestamp = datetime()
        """, username=username, movie_title=movie_title, r

# 基于相似用户的推荐
def recommend_by_collaborative_filtering(driver, username,
    """协同过滤推荐"""
    with driver.session() as session:
        result = session.run("""
            // 找到与目标用户评分相似的其他用户
            MATCH (u:User {name: $username})-[:RATED]->(m:
            MATCH (other:User)-[:RATED]->(m)
            WHERE other <> u
            WITH other, count(m) AS common_movies
            WHERE common_movies >= 2
            // 找到这些相似用户喜欢但目标用户没看过的电影
            MATCH (other)-[:RATED]->(rec:Movie)
            WHERE NOT (u)-[:RATED]->(rec)
              AND rec.rating >= 7.0
            RETURN rec.title AS title, rec.rating AS ratir
                   count(DISTINCT other) AS recommenders
            ORDER BY recommenders DESC, rating DESC
            LIMIT $limit
        """, username=username, limit=limit)
        return [r.data() for r in result]
```

延伸阅读

1. **LangChain Graph QA Chain 官方文档** — 了解 LangChain 内置的图谱问答链如何自动将自然语言转为 Cypher。访问 https://python.langchain.com/docs/use_cases/graph/
2. **Streamlit 官方教程** — 学习更多 Streamlit 组件：图表、地图、相机输入等。访问 https://docs.streamlit.io/
3. **Neo4j Graph Data Science (GDS) 插件** — 在 Neo4j 中运行 PageRank、社区检测等图算法。访问 https://neo4j.com/docs/graph-data-science/current/
4. **RAG（检索增强生成）架构最佳实践** — 深入了解如何构建生产级的 RAG 系统，包括分块策略、重排序、评估方法等。
5. **TMDB API 完整文档** — 探索更多 API 端点：电视剧、综艺、人物详情等。访问 https://developer.themoviedb.org/reference
6. **《知识图谱：方法、实践与应用》** — 系统学习知识图谱的理论方法和工业应用案例。

本章小结

恭喜你完成了这个实战项目！让我们回顾一下这一路的收获：

我们做了什么： – 从零开始设计了一个完整的知识图谱应用系统 – 通过 TMDB API 采集了真实的电影数据 – 用 LLM 从自由文本中抽取了结构化信息（主题、情感等） – 在 Neo4j 中构建了 Movie–Person–Genre–Theme 四层图谱 – 在 Qdrant 中实现了电影描述的语义向量化 – 用 LangChain 搭建了能同时利用图谱和向量的对话系统 – 用 Streamlit 构建了交互式的可视化前端

你学到了什么： – 一个完整的知识图谱项目的开发流程和架构设计 – 如何把多个技术组件（Neo4j、Qdrant、LangChain、Streamlit）有机地组合在一起 – 信息抽取的两种路径：LLM 抽取（灵活准确）和规则抽取（快速免费） – 混合搜索的思路：结构化查询 + 语义搜索 = 更全面的回答

最重要的是： 你现在拥有了一个可以展示的完整项目。把它部署起来，分享给你的朋友或放在 GitHub 上——这比任何证书都更有说服力。

知识图谱的世界很大，我们这本书只是带你入了门。但正如老子说的:”千里之行，始于足下。”你已经迈出了最关键的一步。

接下来，你可以： – 把这个项目的数据换成你感兴趣的领域（音乐、书籍、历史人物……） – 尝试接入更大的数据集 – 添加更多的图算法分析 – 探索多模态知识图谱（加入图片、视频等）

感谢你的阅读，祝你在知识图谱的旅程中越走越远！

—— 树懒老K

第十三章 LLM+知识图谱：RAG与GraphRAG

> *难度：★★★☆☆ 前置要求：完成第4-12章动手实践，熟悉Neo4j与向量数据库基本操作 本章代码：* `https://github.com/sloth-laok/knowledge-graph-intro/tree/main/ch13`

引子

2023年春天，我参加一场技术沙龙。台上嘉宾正激情澎湃地演示ChatGPT："它能写诗、写代码、做PPT……"台下掌声雷动。

到了提问环节，一位做企业知识库的姑娘站起来："我们公司有十万份技术文档，ChatGPT能准确回答我们内部的技术问题吗？"

台上的沉默，比任何回答都诚实。

大语言模型（LLM）像一个读过全世界图书馆的天才——博闻强记，却也会一本正经地胡说八道。它知道你公司昨天上线了什么新功能吗？它清楚你们项目组的代码规范吗？它了解你们客户合同里那些微妙的条款吗？

不知道。因为它的知识有截止日期，因为它对私有数据一无所知，因为它本质上是在"续写"而非"检索"。

这就是知识图谱在大模型时代迎来第二春的契机。当LLM的"通用智能"遇上知识图谱的"精准知识"，一种全新的范式诞生了——**RAG（Retrieval-Augmented Generation，检索增强生**

成）。而当我们把RAG的检索源从向量数据库换成知识图谱，更激动人心的GraphRAG便登场了。

本章将带你从RAG的基本原理出发，一路走到GraphRAG的前沿，最终动手构建一个完整的GraphRAG问答系统。

知识图谱在LLM时代的新角色

LLM的三个致命短板

在讨论解决方案之前，我们需要精确理解LLM到底”不行”在哪里。

第一，幻觉（Hallucination）。 LLM的生成机制本质是概率预测——给定上文，预测下一个token。这意味着它在缺乏知识时，会”编造”看似合理的答案。一个经典的例子：问GPT-4”鲁迅和周树人是什么关系”，它可能回答”他们是同时代的作家”——实际上，鲁迅就是周树人的笔名。

第二，知识截止（Knowledge Cutoff）。 LLM的训练数据有时间边界。GPT-4的训练数据截止到某个时间点，此后发生的一切——新政策、新论文、新的人事变动——它一无所知。

第三，领域盲区（Domain Blindness）。 LLM在公开数据上训练，对企业的内部文档、私有数据、行业专有术语缺乏理解。你问它”我们公司的A产品和B产品在定价策略上有什么区别”，它只能猜测。

知识图谱：LLM的”外挂大脑”

知识图谱恰好能弥补这三个短板：

表 17–1

LLM的短板	知识图谱的补位
幻觉	提供事实依据，约束生成内容
知识截止	实时更新，保持知识鲜活
领域盲区	结构化存储私有领域知识

但知识图谱在LLM时代也面临角色转变。过去，知识图谱往往是”终点站”——用户直接在图谱上查询、推理。现在，知识图谱更多扮演**中间层（Middleware）**的角色：它在幕后为LLM提供知识弹药，LLM负责把结构化知识翻译成自然语言回答。

用一句话说：**知识图谱从”前台展示”走向”后台赋能”。**

RAG原理：检索增强生成

什么是RAG

RAG（Retrieval–Augmented Generation）由Meta在2020年的论文《Retrieval–Augmented Generation for Knowledge–Intensive NLP Tasks》中提出。其核心思想极其简洁：

在生成回答之前，先从外部知识库中检索相关信息，将检索结果作为上下文注入LLM的Prompt中。

用伪代码表示：

```
def rag_query(user_question):
    # 第一步：检索
    relevant_docs = knowledge_base.search(user_question, t

    # 第二步：构建Prompt
    context = "\n".join([doc.content for doc in relevant_d
    prompt = f"""
    根据以下参考资料回答用户问题。如果参考资料不足以回答，请说明。

    参考资料：
    {context}

    用户问题：{user_question}

    回答：
    """

    # 第三步：生成
    answer = llm.generate(prompt)
    return answer
```

这个三步流程——**检索 → 增强 → 生成**——就是RAG的全部。简单得像”开卷考试”：让LLM带着参考资料回答问题，而不是凭记忆硬猜。

RAG的架构变体

虽然核心思想简单，但RAG在工程实现上有多种变体：

Naive RAG：最基础的实现，就是上面的三步流程。问题在于检索质量直接决定生成质量——如果检索回来的文档不相关，LLM也会被”带偏”。

Advanced RAG：在检索前后增加处理步骤。 – 检索前（Pre-retrieval）：查询重写（Query Rewriting）、查询扩展（Query Expansion）、HyDE（Hypothetical Document Embeddings） – 检索后（Post-retrieval）：重排序（Reranking）、上下文压缩（Context Compression）、文档过滤

Modular RAG：将RAG拆解为可插拔的模块，根据不同场景动态组合。比如简单问题跳过检索直接回答，复杂问题触发多轮检索。

RAG的评估指标

评估RAG系统需要同时关注检索和生成两个环节：

检索质量指标： – **Recall@K**：Top-K检索结果中包含正确答案的比例 – **MRR（Mean Reciprocal Rank）**：正确答案排在第几位的倒数均值 – **NDCG**：考虑排序位置的加权相关性

生成质量指标： – **Faithfulness（忠实度）**：生成的回答是否忠于检索到的上下文 – **Answer Relevance（答案相关性）**：回答是否切题 – **Context Precision（上下文精确率）**：检索到的内容中有多少是真正有用的

RAGAS（Retrieval Augmented Generation Assessment）是目前最流行的RAG评估框架，它利用LLM自身来评估上述指标，无需人工标注。

Vector RAG vs Graph RAG：两条路线的交锋

Vector RAG：向量检索路线

Vector RAG是目前最主流的RAG实现方式。它将文档切分成Chunk，用Embedding模型将每个Chunk转化为向量，存入向量数据库（如Qdrant、Milvus、Pinecone等），查询时通过语义相似度检索最相关的Chunk。

```
# Vector RAG 典型流程
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Qdrant
from langchain.text_splitter import RecursiveCharacterText

# 1. 文档切分
splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunks = splitter.split_documents(documents)

# 2. 向量化并存储
embeddings = OpenAIEmbeddings()
vectorstore = Qdrant.from_documents(chunks, embeddings, ur

# 3. 检索
retriever = vectorstore.as_retriever(search_kwargs={"k": 5
relevant_chunks = retriever.get_relevant_documents("知识图谱
```

Vector RAG的优势： – 实现简单，生态成熟 – 语义匹配能力强，能处理同义词、近义表达 – 适合非结构化文本的大规模检索 – 水平扩展性好，向量数据库天然支持分布式

Vector RAG的劣势： – **丢失结构信息**：文档被切成Chunk后，实体之间的关系、文档之间的引用链路全部断裂 – **全局理解困难**：对于”这个项目整体进展如何”这类需要综合多处信息的问题，Vector RAG往往只能检索到局部信息 – **多跳推理无力**：对于”A的导师的研究方向与B的公司业务有什么交集”这类需要沿着关系链路推理的问题，向量检索几乎无能为力

Graph RAG：图谱检索路线

Graph RAG正是为解决Vector RAG上述痛点而生。它以知识图谱为检索源，利用实体和关系的结构化信息进行检索和推理。

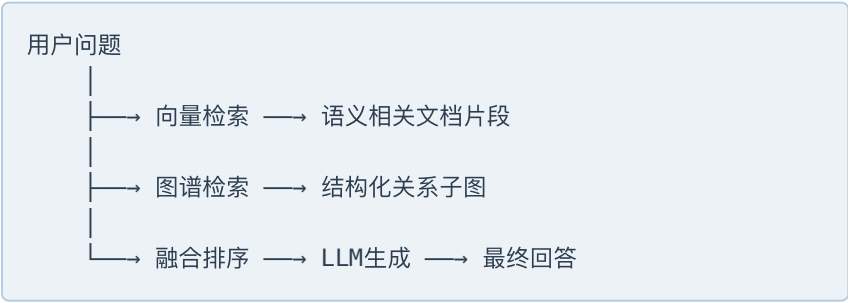
```
# Graph RAG 概念性伪代码
def graph_rag_query(question):
    # 1. 从问题中提取实体和意图
    entities = extract_entities(question)
    intent = classify_intent(question)

    # 2. 在知识图谱中定位相关子图
    subgraph = knowledge_graph.traverse(
        start_entities=entities,
        max_hops=3,
        relevance_filter=intent
    )

    # 3. 将子图转化为自然语言上下文
    context = graph_to_text(subgraph)

    # 4. LLM基于结构化上下文生成回答
    answer = llm.generate(prompt_with_context(question, cc
    return answer
```

Graph RAG的优势： – **保留关系结构**：实体之间的连接关系完整保留，支持多跳推理 – **全局视角**：通过图的连通性，可以发现看似无关信息之间的隐含关联 – **可解释性强**：检索路径清晰可见——从哪个实体出发，沿着什么关系，到达了哪些节点 – **精确查询**：对于”张三在哪个部门”“这个项目的负责人是谁”这类事实性问题，图谱检索远比向量检索精确

Graph RAG的劣势： – 图谱构建成本高，需要实体识别和关系抽取 – 对非结构化文本的覆盖不如向量检索 – Cypher等图查询语言的学习曲线较陡 – 图谱的完整性直接影响检索质量

最佳实践：混合RAG

在实际工程中，最务实的选择是**混合RAG（Hybrid RAG）**——同时使用向量检索和图谱检索，将两路结果融合后送入LLM。



这种混合架构兼顾了语义灵活性和结构精确性，是生产环境中最常见的RAG架构。

Microsoft GraphRAG：社区检测与层级摘要

2024年，微软研究院发布了GraphRAG项目，这是Graph RAG领域最具影响力的开源实现之一。它的核心创新在于引入了**社区检测（Community Detection）**和**层级摘要（Hierarchical Summarization）**两个机制。

GraphRAG的核心思想

传统的Graph RAG在面对”这个数据集的主要主题是什么”这类全局性问题时表现不佳——因为它只能检索到局部子图。微软GraphRAG的解法是：**预先对知识图谱进行社区划分，并为每个社**

区生成摘要，查询时根据问题的粒度选择不同层级的摘要。

整个流程分为两个阶段：

索引阶段（Indexing）： 1. 从源文档中提取实体和关系，构建知识图谱 2. 使用Leiden算法对图进行社区检测，形成层级化的社区结构 3. 为每个社区（以及社区的父社区、祖父社区……）生成摘要

查询阶段（Querying）： – **Local Search**：针对具体实体和局部关系的问题，沿着图谱遍历相关子图 – **Global Search**：针对全局性、主题性的问题，利用社区摘要的map–reduce策略生成回答

社区检测：Leiden算法

Leiden算法是Louvain算法的改进版，用于发现图中的社区结构。它的核心思想是**最大化模块度（Modularity）**——让社区内部的连接尽可能紧密，社区之间的连接尽可能稀疏。

```
# 使用GraphRAG的社区检测（概念性代码）
import networkx as nx
from graphrag.index.community_detection import leiden_comm

# 构建图
G = nx.Graph()
G.add_edges_from(extracted_relationships)

# 层级社区检测
community_hierarchy = leiden_community_detection(
    graph=G,
    max_levels=5,        # 最多5层社区层级
    resolution=1.0,      # 分辨率参数，控制社区粒度
)

# 输出示例：
# Level 0: 1个大社区（全局）
# Level 1: 5个中等社区（主题域）
# Level 2: 23个小社区（子主题）
# Level 3: 87个微型社区（具体话题）
```

层级摘要：从细节到全局

社区检测后，GraphRAG为每个社区生成一份摘要，内容包括： – 社区名称和描述 – 社区包含的关键实体 – 社区内的主要关系和主题 – 社区的重要度评分

这些摘要按层级组织，形成一棵**摘要树（Summary Tree）**：

```
[全局摘要]
├── [主题域A摘要]
│   ├── [子主题A1摘要]
│   │   ├── [话题A1a摘要]
│   │   └── [话题A1b摘要]
│   └── [子主题A2摘要]
├── [主题域B摘要]
│   ├── [子主题B1摘要]
│   └── [子主题B2摘要]
└── [主题域C摘要]
```

Global Search的Map-Reduce策略

面对全局性问题（如”这批文档的核心主题是什么”），GraphRAG采用Map-Reduce策略：

1. **Map阶段**：将每个社区摘要分别送入LLM，让它针对每个社区摘要回答该社区与问题的相关性
2. **Reduce阶段**：将所有Map结果汇总，让LLM生成最终的全局性回答

这种策略的优势在于，它能覆盖整个知识图谱的信息，而不是局限于局部子图。代价是Token消耗较大——Global Search的Token成本通常是Local Search的10-50倍。

LangChain与LlamaIndex集成实战

技术栈选择

本章的实战项目采用以下技术栈：

表 17-2

组件	选型	理由
LLM	OpenAI GPT-4o	效果好，API稳定
图数据库	Neo4j 5.x	业界标准，Cypher支持好
向量数据库	Qdrant	轻量，性能好
框架	LangChain + LangGraph	生态完善，GraphRAG支持好
Embedding	text-embedding-3-small	性价比高

环境准备

```
# 创建项目目录
mkdir graphrag-demo && cd graphrag-demo

# 创建虚拟环境
python -m venv venv
source venv/bin/activate

# 安装依赖
pip install langchain langchain-openai langchain-community
    langgraph neo4j qdrant-client \
    graphrag ragas tiktoken

# 启动Neo4j（Docker）
docker run -d --name neo4j \
    -p 7474:7474 -p 7687:7687 \
    -e NEO4J_AUTH=neo4j/password123 \
    -e NEO4J_PLUGINS='["apoc"]' \
    neo4j:5.20

# 启动Qdrant（Docker）
docker run -d --name qdrant \
    -p 6333:6333 -p 6334:6334 \
    qdrant/qdrant:v1.9.0

# 设置环境变量
export OPENAI_API_KEY="sk-your-key-here"
```

用LangChain构建基础RAG

先从Vector RAG开始，建立基线：

```
# baseline_rag.py
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.vectorstores import Qdrant
from langchain.text_splitter import RecursiveCharacterText
from langchain_community.document_loaders import Directory
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

# 1. 加载文档
loader = DirectoryLoader("./docs", glob="**/*.md")
documents = loader.load()

# 2. 切分文档
splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    separators=["\n## ", "\n### ", "\n\n", "\n", " "]
)
chunks = splitter.split_documents(documents)

# 3. 向量化存储
embeddings = OpenAIEmbeddings(model="text-embedding-3-smal
vectorstore = Qdrant.from_documents(
    documents=chunks,
    embedding=embeddings,
    url="http://localhost:6333",
    collection_name="knowledge_base"
)

# 4. 构建检索链
retriever = vectorstore.as_retriever(search_kwargs={"k": 5

prompt = ChatPromptTemplate.from_template("""
```

```
你是一个知识渊博的助手。请根据以下参考资料回答用户问题。
如果参考资料不足以回答，请明确说明。

参考资料：
{context}

用户问题：{question}

请用中文回答，并提供信息来源。
""")

llm = ChatOpenAI(model="gpt-4o", temperature=0)

chain = (
    {"context": retriever, "question": RunnablePassthrough
    | prompt
    | llm
    | StrOutputParser()
)

# 5. 测试
result = chain.invoke("知识图谱在金融领域有哪些应用？")
print(result)
```

用LangChain接入Neo4j图谱

接下来，我们在Neo4j中构建知识图谱，并通过LangChain的图检索能力接入RAG：

```
# graph_setup.py
from neo4j import GraphDatabase

driver = GraphDatabase.driver("bolt://localhost:7687", aut

def build_knowledge_graph():
    """将文档中的实体和关系写入Neo4j"""
    with driver.session() as session:
        # 创建实体节点
        session.run("""
        MERGE (kg:Topic {name: '知识图谱'})
        MERGE (nlp:Topic {name: '自然语言处理'})
        MERGE (ml:Topic {name: '机器学习'})
        MERGE (finance:Domain {name: '金融'})
        MERGE (healthcare:Domain {name: '医疗'})
        MERGE (ecommerce:Domain {name: '电商'})

        // 创建关系
        MERGE (kg)-[:RELATED_TO {weight: 0.9}]->(nlp)
        MERGE (kg)-[:RELATED_TO {weight: 0.8}]->(ml)
        MERGE (kg)-[:APPLIED_IN {examples: '风控、反欺诈、智
        MERGE (kg)-[:APPLIED_IN {examples: '药物发现、辅助诊
        MERGE (kg)-[:APPLIED_IN {examples: '推荐系统、商品搜

        // 添加具体案例
        MERGE (ant:Company {name: '蚂蚁集团'})
        MERGE (ant)-[:DEVELOPED]->(kg_risk:System {name: '
        MERGE (kg_risk)-[:USES]->(kg)
        MERGE (kg_risk)-[:APPLIED_IN]->(finance)
        """)

build_knowledge_graph()
print("知识图谱构建完成")
```

```
# graph_rag.py
from langchain_community.graphs import Neo4jGraph
from langchain.chains import GraphCypherQAChain
from langchain_openai import ChatOpenAI

# 连接Neo4j图
graph = Neo4jGraph(
    url="bolt://localhost:7687",
    username="neo4j",
    password="password123"
)

# 刷新图schema
graph.refresh_schema()
print("图Schema:", graph.schema)

# 构建Graph Cypher QA链
llm = ChatOpenAI(model="gpt-4o", temperature=0)

chain = GraphCypherQAChain.from_llm(
    llm=llm,
    graph=graph,
    verbose=True,
    return_intermediate_steps=True,
    validate_cypher=True,  # 验证生成的Cypher语法
    cypher_prompt=None     # 使用默认prompt，也可自定义
)

# 测试图检索问答
questions = [
    "知识图谱在哪些行业有应用？",
    "蚂蚁集团开发了什么系统？",
    "知识图谱和自然语言处理有什么关系？"
]
```

```
for q in questions:
    result = chain.invoke({"query": q})
    print(f"\n问题：{q}")
    print(f"回答：{result['result']}")
    print(f"中间步骤：{result['intermediate_steps']}")
```

混合RAG：融合向量与图谱检索

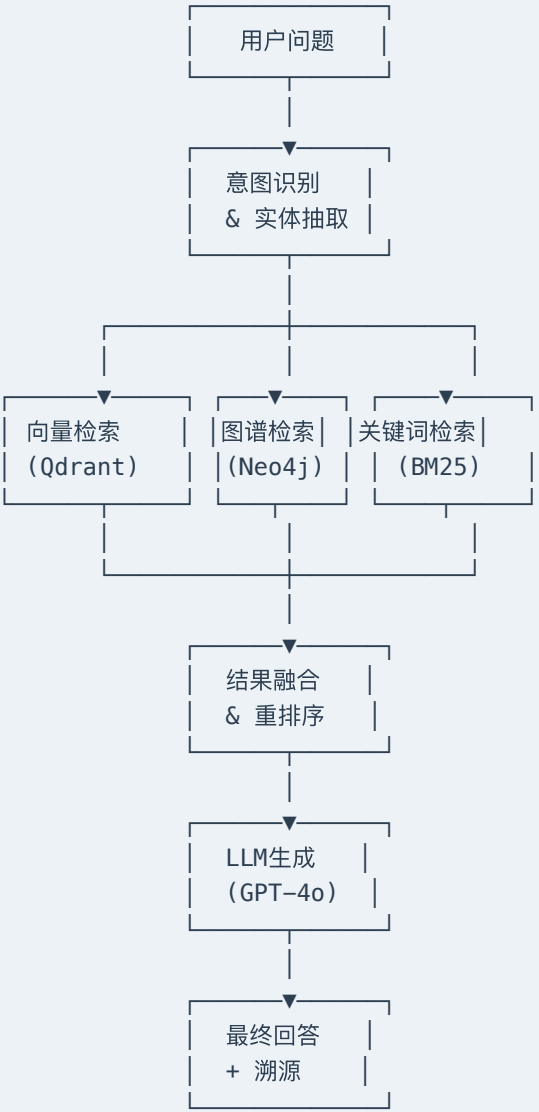
```
# hybrid_rag.py
from langchain_core.runnables import RunnableParallel, Rur
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

def vector_search(query: str) -> str:
    """向量检索"""
    docs = retriever.get_relevant_documents(query)
    return "\n".join([f"[向量检索] {d.page_content}" for d

def graph_search(query: str) -> str:
    """图谱检索"""
    try:
        result = chain.invoke({"query": query})
        return f"[图谱检索] {result['result']}"
    except Exception as e:
        return f"[图谱检索失败] {str(e)}"

# 混合检索Prompt
hybrid_prompt = ChatPromptTemplate.from_template("""
你是一个知识渊博的助手。请综合以下两个检索来源的信息回答用户问题。

来源1 - 文档检索:
{vector_context}

来源2 - 知识图谱检索:
{graph_context}

用户问题: {question}

要求:
1. 优先使用知识图谱提供的结构化事实
```

```
2. 用文档检索补充细节和上下文
3. 如果两个来源有冲突，指出差异
4. 标注信息来源
""")

# 构建混合链
hybrid_chain = (
    {
        "vector_context": lambda x: vector_search(x),
        "graph_context": lambda x: graph_search(x),
        "question": RunnablePassthrough()
    }
    | hybrid_prompt
    | llm
    | StrOutputParser()
)

# 测试混合RAG
result = hybrid_chain.invoke("知识图谱在金融风控中怎么应用？有哪
print(result)
```

动手实战：构建完整的GraphRAG问答系统

项目架构

我们要构建一个能够回答关于”AI技术发展”的问答系统。系统架构如下：


用户问题
意图识别
& 实体抽取
向量检索
(Qdrant)
图谱检索
(Neo4j)
关键词检索
(BM25)
结果融合
& 重排序
LLM生成
(GPT-4o)
最终回答
+ 溯源

第一步：构建知识图谱

```
# step1_build_graph.py
"""
从AI领域的技术文档中构建知识图谱。
包含：技术概念、公司、产品、论文、人物等实体。
"""
from neo4j import GraphDatabase
import json

driver = GraphDatabase.driver("bolt://localhost:7687", aut

def create_constraints():
    """创建唯一性约束，提升查询性能"""
    with driver.session() as session:
        session.run("CREATE CONSTRAINT IF NOT EXISTS FOR (
        session.run("CREATE CONSTRAINT IF NOT EXISTS FOR (
        session.run("CREATE CONSTRAINT IF NOT EXISTS FOR (
        session.run("CREATE CONSTRAINT IF NOT EXISTS FOR (
        session.run("CREATE CONSTRAINT IF NOT EXISTS FOR (
        session.run("CREATE CONSTRAINT IF NOT EXISTS FOR (
        print("约束创建完成")

def populate_graph():
    """填充知识图谱数据"""
    # 从JSON文件加载数据（也可以用LLM从文档中提取）
    with open("ai_knowledge.json", "r") as f:
        data = json.load(f)

    with driver.session() as session:
        # 写入概念和层级关系
        for concept in data["concepts"]:
            session.run("""
                MERGE (c:Concept {name: $name})
```

```
                SET c.description = $description,
                    c.category = $category
            """, name=concept["name"],
                description=concept["description"],
                category=concept["category"])

        # 写入公司实体
        for company in data["companies"]:
            session.run("""
                MERGE (co:Company {name: $name})
                SET f.country = $country,
                    co.founded = $founded
            """, name=company["name"],
                country=company["country"],
                founded=company["founded"])

        # 写入关系
        for rel in data["relationships"]:
            session.run(f"""
                MATCH (a:{rel['source_type']} {{name: $sou
                MATCH (b:{rel['target_type']} {{name: $tar
                MERGE (a)-[r:{rel['relation']}]->(b)
                SET r.since = $since,
                    r.confidence = $confidence
            """, source=rel["source"],
                target=rel["target"],
                since=rel.get("since"),
                confidence=rel.get("confidence", 1.0))

    print(f"知识图谱填充完成：{len(data['concepts'])}个概念，"
          f"{len(data['companies'])}个公司，"
          f"{len(data['relationships'])}条关系")

def verify_graph():
    """验证图谱数据"""
```

```
    with driver.session() as session:
        result = session.run("""
            MATCH (n)
            RETURN labels(n)[0] AS label, count(n) AS cour
            ORDER BY count DESC
        """)
        print("\n图谱统计：")
        for record in result:
            print(f"  {record['label']}: {record['count']}

        result = session.run("""
            MATCH ()-[r]->()
            RETURN type(r) AS type, count(r) AS count
            ORDER BY count DESC
        """)
        print("\n关系统计：")
        for record in result:
            print(f"  {record['type']}: {record['count']}条

create_constraints()
populate_graph()
verify_graph()
```

第二步：实现多路检索与融合

```
# step2_retrievers.py
"""
实现三路检索：向量检索、图谱检索、关键词检索
"""
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_community.vectorstores import Qdrant
from langchain_community.graphs import Neo4jGraph
from langchain_core.documents import Document
from typing import List, Dict, Any
import re

class HybridRetriever:
    def __init__(self, neo4j_url, neo4j_auth, qdrant_url,
        self.embeddings = OpenAIEmbeddings(model="text-emb
        self.llm = ChatOpenAI(model="gpt-4o", temperature=

        # 向量数据库
        self.vectorstore = Qdrant.from_existing_collectior
            embedding=self.embeddings,
            url=qdrant_url,
            collection_name=collection_name
        )

        # 图数据库
        self.graph = Neo4jGraph(
            url=neo4j_url,
            username=neo4j_auth[0],
            password=neo4j_auth[1]
        )
        self.graph.refresh_schema()

    def extract_entities(self, query: str) -> List[str]:
```

```
        """从用户问题中提取关键实体"""
        prompt = f"""从以下问题中提取关键实体（技术名词、公司名、
        以JSON数组格式返回。只返回实体列表，不要其他内容。

        问题：{query}

        返回格式：["entity1", "entity2", ...]
        """
        response = self.llm.invoke(prompt)
        try:
            entities = eval(response.content.strip())
            return entities if isinstance(entities, list)
        except:
            return []

    def vector_retrieve(self, query: str, k: int = 5) -> L
        """向量语义检索"""
        docs = self.vectorstore.similarity_search_with_sc
        results = []
        for doc, score in docs:
            results.append({
                "content": doc.page_content,
                "source": "vector",
                "score": float(score),
                "metadata": doc.metadata
            })
        return results

    def graph_retrieve(self, query: str, entities: List[st
        """知识图谱检索"""
        results = []

        for entity in entities:
            try:
                # 多跳查询：找到实体及其相关节点
```

```
                cypher = """
                MATCH (n)
                WHERE n.name CONTAINS $entity
                OPTIONAL MATCH (n)-[r]->(m)
                WITH n, collect({
                    relation: type(r),
                    target: m.name,
                    target_labels: labels(m)
                })[..5] AS outgoing
                OPTIONAL MATCH (m2)-[r2]->(n)
                WITH n, outgoing, collect({
                    relation: type(r2),
                    source: m2.name,
                    source_labels: labels(m2)
                })[..5] AS incoming
                RETURN n.name AS entity,
                       labels(n) AS labels,
                       n.description AS description,
                       outgoing,
                       incoming
                LIMIT 5
                """
                response = self.graph.query(cypher, {"enti

                for record in response:
                    content = self._format_graph_result(re
                    results.append({
                        "content": content,
                        "source": "graph",
                        "score": 0.9,  # 图谱结果给较高基础分
                        "entity": record.get("entity"),
                        "metadata": {"labels": record.get(
                    })
            except Exception as e:
                print(f"图谱查询失败 [{entity}]: {e}")
```

```
        return results

    def keyword_retrieve(self, query: str, k: int = 3) ->
        """关键词检索（BM25模拟，实际可用Elasticsearch）"""
        # 简化实现：在Neo4j中做全文搜索
        keywords = " ".join(query.split()[:5])
        try:
            cypher = """
            CALL db.index.fulltext.queryNodes('nameIndex',
            YIELD node, score
            RETURN node.name AS name,
                   labels(node) AS labels,
                   node.description AS description,
                   score
            LIMIT $k
            """
            response = self.graph.query(cypher, {"keywords
            results = []
            for record in response:
                results.append({
                    "content": f"{record['name']}: {record
                    "source": "keyword",
                    "score": float(record.get("score", 0.5
                    "metadata": {"labels": record.get("lab
                })
            return results
        except:
            return []

    def _format_graph_result(self, record: Dict) -> str:
        """将图谱查询结果格式化为自然语言"""
        parts = [f"实体: {record['entity']}"]
        if record.get("description"):
            parts.append(f"描述: {record['description']}")
```

```
        if record.get("outgoing"):
            for rel in record["outgoing"]:
                if rel.get("target"):
                    parts.append(f"  → {rel['relation']}:

        if record.get("incoming"):
            for rel in record["incoming"]:
                if rel.get("source"):
                    parts.append(f"  ← {rel['relation']}:

        return "\n".join(parts)

    def retrieve(self, query: str) -> List[Dict]:
        """执行混合检索并融合结果"""
        # 1. 提取实体
        entities = self.extract_entities(query)
        print(f"提取的实体: {entities}")

        # 2. 三路并行检索
        vector_results = self.vector_retrieve(query)
        graph_results = self.graph_retrieve(query, entitie
        keyword_results = self.keyword_retrieve(query)

        # 3. 融合与去重
        all_results = vector_results + graph_results + key
        print(f"检索结果: 向量{len(vector_results)}条, "
              f"图谱{len(graph_results)}条, "
              f"关键词{len(keyword_results)}条")

        # 4. 简单融合排序（按分数降序）
        all_results.sort(key=lambda x: x["score"], reverse

        return all_results[:10]  # 返回Top-10
```

第三步：构建完整的问答链

```
# step3_qa_chain.py
"""
完整的GraphRAG问答系统
"""
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from step2_retrievers import HybridRetriever

class GraphRAGSystem:
    def __init__(self):
        self.retriever = HybridRetriever(
            neo4j_url="bolt://localhost:7687",
            neo4j_auth=("neo4j", "password123"),
            qdrant_url="http://localhost:6333",
            collection_name="ai_knowledge"
        )
        self.llm = ChatOpenAI(model="gpt-4o", temperature=

    def answer(self, question: str) -> dict:
        """回答用户问题"""
        # 1. 混合检索
        results = self.retriever.retrieve(question)

        # 2. 构建上下文
        context_parts = []
        sources = []
        for i, r in enumerate(results, 1):
            context_parts.append(f"[来源{i}|{r['source']}]
            sources.append({
                "source": r["source"],
                "content_preview": r["content"][:100],
```

```
                "score": r["score"]
            })

        context = "\n\n".join(context_parts)

        # 3. 生成回答
        prompt = ChatPromptTemplate.from_template("""
你是一个专业的AI技术顾问。请根据以下检索到的信息回答用户问题。

<span style="font-size:0.5pt;">§TOC16-8§</span>
## 检索到的信息

{context}

<span style="font-size:0.5pt;">§TOC16-9§</span>
## 用户问题

{question}

<span style="font-size:0.5pt;">§TOC16-10§</span>
## 回答要求

1. 基于提供的信息回答，不要编造
2. 如果信息不足以完整回答，明确指出哪些部分是基于检索信息、哪些部分是
3. 在回答末尾列出关键信息来源
4. 如果不同来源的信息有冲突，请指出并给出你的判断

请用中文回答。
""")

        chain = prompt | self.llm | StrOutputParser()
        answer = chain.invoke({
            "context": context,
            "question": question
        })
```

```
        return {
            "answer": answer,
            "sources": sources,
            "retrieval_count": len(results)
        }

    def interactive_mode(self):
        """交互模式"""
        print("=" * 60)
        print("GraphRAG 问答系统已启动")
        print("输入 'quit' 或 'exit' 退出")
        print("=" * 60)

        while True:
            question = input("\n你的问题: ").strip()
            if question.lower() in ("quit", "exit", "q"):
                print("再见!")
                break
            if not question:
                continue

            print("\n检索中...")
            result = self.answer(question)

            print(f"\n{'-' * 60}")
            print(f"回答: \n{result['answer']}")
            print(f"\n{'-' * 60}")
            print(f"检索到 {result['retrieval_count']} 条相
            for i, src in enumerate(result["sources"][:5],
                print(f"  [{i}] 来源: {src['source']} | 相
                print(f"      预览: {src['content_preview']

# 启动系统
```

```
if __name__ == "__main__":
    system = GraphRAGSystem()
    system.interactive_mode()
```

第四步：评估与优化

```
# step4_evaluate.py
"""
使用RAGAS框架评估GraphRAG系统
"""
from ragas import evaluate
from ragas.metrics import (
    faithfulness,
    answer_relevancy,
    context_precision,
    context_recall
)
from datasets import Dataset

def build_eval_dataset():
    """构建评估数据集"""
    eval_data = {
        "question": [
            "知识图谱在金融领域有哪些应用？",
            "GPT-4和Claude 3有什么区别？",
            "什么是RAG技术？它解决了什么问题？",
            "微软GraphRAG的核心创新是什么？",
            "知识图谱和向量数据库哪个更适合多跳推理？"
        ],
        "ground_truth": [
            "知识图谱在金融领域的应用包括风险控制、反欺诈、智能投顾
            "GPT-4和Claude 3都是大语言模型，但在安全性、推理能力
            "RAG是检索增强生成技术，通过在生成前检索外部知识来减少
            "微软GraphRAG的核心创新是引入社区检测和层级摘要，使RA
            "知识图谱更适合多跳推理，因为它保留了实体之间的结构化关
        ],
        "answer": [],       # 系统生成的回答
        "contexts": []      # 检索到的上下文
```

```
    }

    system = GraphRAGSystem()

    for q in eval_data["question"]:
        result = system.answer(q)
        eval_data["answer"].append(result["answer"])
        eval_data["contexts"].append([
            s["content_preview"] for s in result["sources"
        ])

    return Dataset.from_dict(eval_data)

# 运行评估
dataset = build_eval_dataset()

results = evaluate(
    dataset,
    metrics=[faithfulness, answer_relevancy, context_preci
)

print("\n===== RAG 评估结果 =====")
for metric_name, score in results.items():
    print(f"  {metric_name}: {score:.4f}")
```

动手练习

练习1：扩展知识图谱（★★☆☆☆）

在已有的AI知识图谱基础上，添加以下信息： 1. 至少10篇重要的AI论文（包含标题、作者、年份、会议/期刊） 2. 论文与技术的”INSPIRED”关系（某论文启发了某项技术） 3. 人物与公司的”WORKED_AT”关系 4. 为新增节点添加全文索引，测试关键词检索的召回率

练习2：实现查询重写（★★★☆☆）

实现Advanced RAG中的查询重写策略： 1. **HyDE**：让LLM先生成一个假设性回答，用这个回答（而非原始问题）去做向量检索 2. **多查询扩展**：将一个问题改写为3–5个不同角度的子问题，分别检索后合并结果 3. 对比原始查询和重写后查询的Recall@5，量化提升效果

练习3：GraphRAG全局搜索（★★★★☆）

参考微软GraphRAG的思路，在你的知识图谱上实现： 1. 使用NetworkX的`community`模块对图做Louvain社区检测 2. 为每个社区用LLM生成100字以内的摘要 3. 实现Global Search的Map–Reduce流程 4. 用以下全局性问题测试： –“AI领域最重要的三个技术方向是什么？”–“哪些公司在知识图谱领域投入最多？”

练习4：构建对比实验（★★★☆☆）

设计一个对比实验，比较Vector RAG、Graph RAG和Hybrid RAG在以下维度的表现： 1. 事实性问题的准确率（如"XX公司的CEO是谁"） 2. 关系推理的准确率（如"A和B有什么关系"） 3. 全局性问题的覆盖率（如"这个领域的主要趋势"） 4. 响应延迟（从提问到回答的时间） 5. Token消耗（API调用成本）

每个类型至少准备10个测试问题，用表格展示对比结果。

延伸阅读

核心论文

1. **《Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks》**（Lewis et al., 2020）——RAG的奠基之作
2. **《From Local to Global: A Graph RAG Approach to Query-Focused Summarization》**（Edge et al., 2024）——微软GraphRAG论文
3. **《Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection》**（Asai et al., 2024）——自适应RAG
4. **《RAGAS: Automated Evaluation of Retrieval Augmented Generation》**（Es et al., 2024）——RAG评估框架

开源项目

- **microsoft/graphrag**：微软GraphRAG官方实现
- **langchain-ai/langchain**：LangChain框架，包含丰富的RAG组件
- **run-llama/llama_index**：LlamaIndex，另一个优秀的RAG框架
- **explodinggradients/ragas**：RAGAS评估框架
- **neo4j-labs/neodash**：Neo4j可视化仪表盘

在线资源

- LangChain GraphRAG教程
- 微软GraphRAG文档
- Neo4j LLM集成指南
- RAG技术全景图（2024）

本章小结

本章我们从RAG的基本原理出发，一路深入到GraphRAG的前沿实践：

1. **知识图谱在LLM时代找到了新定位**——从"前台展示"走向"后台赋能"，成为LLM的"外挂大脑"
2. **RAG的核心是"开卷考试"**——检索相关文档，注入Prompt，让LLM基于事实生成回答

3. **Vector RAG和Graph RAG各有优劣**——前者擅长语义匹配，后者擅长结构推理，混合使用效果最佳
4. **微软GraphRAG的社区检测+层级摘要**解决了全局性问题的回答难题
5. **工程落地需要完整的评估体系**——RAGAS等框架帮助我们量化RAG系统的检索质量和生成质量

下一章，我们将把目光从算法转向工程——讨论如何将知识图谱系统从开发环境推向生产环境，实现高可用、高性能、可监控的企业级部署。

补充深入：GraphRAG的工程挑战与解决方案

图谱质量的”鸡生蛋”困境

在实际落地GraphRAG时，最让人头疼的不是算法，而是**图谱质量**。

你可能会遇到这样的场景：LLM从问题中提取了实体”知识图谱”，去Neo4j里一查——节点不存在。为什么？因为当初构建图谱的时候，有人用的是”KG”，有人用的是”知识图谱（Knowledge Graph）”，还有人用的是”知识网络”。

实体规范化是GraphRAG工程中绕不开的第一步。我们的经验是：

```
# entity_normalization.py
"""
实体规范化：解决同一实体的多种表述
"""
from langchain_openai import ChatOpenAI

class EntityNormalizer:
    """实体规范化工具"""

    def __init__(self, graph_client, llm=None):
        self.graph = graph_client
        self.llm = llm or ChatOpenAI(model="gpt-4o-mini",
        self.alias_cache = self._build_alias_cache()

    def _build_alias_cache(self):
        """从图谱中预加载所有别名映射"""
        results = self.graph.read("""
            MATCH (n)-[:ALIAS_OF]->(canonical)
            RETURN n.name AS alias, canonical.name AS canc
        """)
        cache = {}
        for r in results:
            cache[r["alias"].lower()] = r["canonical_name"
            # 常见变体
            cache[r["alias"].replace(" ", "").lower()] = r
        return cache

    def normalize(self, raw_entity: str) -> str:
        """将原始实体名规范化"""
        # 第一步：查缓存
        key = raw_entity.strip().lower()
        if key in self.alias_cache:
            return self.alias_cache[key]
```

```
        # 第二步：模糊匹配（编辑距离 <= 2）
        for alias, canonical in self.alias_cache.items():
            if self._edit_distance(key, alias) <= 2:
                return canonical

        # 第三步：LLM判断（最慢但最准）
        return self._llm_resolve(raw_entity)

    def _llm_resolve(self, entity: str) -> str:
        """让LLM判断实体是否已在图谱中存在"""
        # 获取图谱中相似的实体名
        similar = self.graph.read("""
            MATCH (n)
            WHERE toLower(n.name) CONTAINS toLower($partia
            RETURN n.name AS name
            LIMIT 10
        """, partial=entity[:4])

        if not similar:
            return entity  # 图谱中无匹配，返回原始值

        names = [r["name"] for r in similar]
        response = self.llm.invoke(f"""
        用户提到了实体"{entity}"。以下是知识图谱中已有的相似实体：
        {names}

        请判断：
        1. 用户提到的实体是否与列表中某个实体相同？
        2. 如果相同，返回那个实体的名称
        3. 如果不同，返回用户原始的实体名

        只返回实体名，不要其他解释。
        """)
        return response.content.strip()
```

```
    def _edit_distance(self, s1, s2):
        """计算编辑距离"""
        if len(s1) < len(s2):
            return self._edit_distance(s2, s1)
        if len(s2) == 0:
            return len(s1)
        prev_row = range(len(s2) + 1)
        for i, c1 in enumerate(s1):
            curr_row = [i + 1]
            for j, c2 in enumerate(s2):
                insertions = prev_row[j + 1] + 1
                deletions = curr_row[j] + 1
                substitutions = prev_row[j] + (c1 != c2)
                curr_row.append(min(insertions, deletions,
            prev_row = curr_row
        return prev_row[-1]
```

查询路由：决定什么时候用图谱

不是所有问题都需要查询知识图谱。一个好的GraphRAG系统应该有**查询路由器（Query Router）**，根据问题类型选择最优的检索策略：

```
# query_router.py
"""
查询路由器：智能选择检索策略
"""

class QueryRouter:
    """根据问题类型路由到不同的检索管道"""

    ROUTES = {
        "factual": "graph",           # 事实性问题 → 图谱检索
        "relational": "graph",        # 关系推理 → 图谱检索
        "semantic": "vector",         # 语义搜索 → 向量检索
        "global": "graphrag_global", # 全局性问题 → GraphRA
        "temporal": "graph",          # 时间相关 → 图谱（带时
        "creative": "direct_llm",     # 创作性问题 → 直接LLM生
    }

    def __init__(self, llm):
        self.llm = llm

    def classify(self, question: str) -> str:
        """对问题进行意图分类"""
        prompt = f"""将以下问题分类为以下类别之一：
        - factual: 询问具体事实（如"张三的职位是什么"）
        - relational: 询问实体间关系（如"A和B有什么关系"）
        - semantic: 语义搜索类（如"有哪些关于机器学习的应用案例"）
        - global: 全局性问题（如"这个领域的整体趋势是什么"）
        - temporal: 时间相关（如"去年发生了哪些重要事件"）
        - creative: 创作/建议类（如"帮我写一份技术方案"）

        问题：{question}

        只返回类别名称，不要其他内容。
        """
```

```
        category = self.llm.invoke(prompt).content.strip()
        return category if category in self.ROUTES else "s

    def route(self, question: str) -> str:
        """路由到对应的检索管道"""
        category = self.classify(question)
        pipeline = self.ROUTES.get(category, "vector")
        print(f"[路由] 问题分类: {category} → 管道: {pipeline
        return pipeline
```

Token成本控制

GraphRAG的一个现实挑战是**Token成本**。Global Search的map-reduce策略尤其烧钱——对100个社区摘要分别调用LLM，然后再做一次汇总，Token消耗可能是普通RAG的50倍。

几个实用的成本控制策略：

策略一：分层模型。Map阶段用小模型（如GPT-4o-mini）做初步筛选，Reduce阶段用大模型（如GPT-4o）做最终生成。

策略二：摘要缓存。社区摘要变化频率低，可以缓存到Redis中，避免重复生成。

策略三：预算控制。为每个查询设置Token预算上限，超出预算时自动降级为Local Search。

```
# token_budget.py
"""
Token预算管理器
"""
import tiktoken

class TokenBudgetManager:
    """控制RAG查询的Token消耗"""

    def __init__(self, max_tokens_per_query=50000,
                 max_cost_per_day_usd=10.0):
        self.max_per_query = max_tokens_per_query
        self.max_per_day = max_cost_per_day_usd
        self.daily_spent = 0.0
        self.encoder = tiktoken.encoding_for_model("gpt-4o

    def count_tokens(self, text: str) -> int:
        """计算文本的Token数"""
        return len(self.encoder.encode(text))

    def can_afford(self, estimated_tokens: int) -> bool:
        """检查预算是否允许"""
        estimated_cost = estimated_tokens / 1000 * 0.005
        return (self.daily_spent + estimated_cost) < self.

    def choose_strategy(self, question: str, community_cou
        """根据预算选择搜索策略"""
        # 估算Global Search的Token消耗
        global_estimate = community_count * 500 + 2000  #

        if self.can_afford(global_estimate):
            return "global"
        elif self.can_afford(5000):
```

```
            return "local"
        else:
            return "vector_only"  # 降级为纯向量检索
```

生产级GraphRAG的缓存层

在生产环境中，为GraphRAG添加缓存层是提升性能、降低成本的关键手段：

```
# caching_layer.py
"""
GraphRAG多级缓存架构
"""
import hashlib
import json
import redis

class GraphRAGCache:
    """多级缓存管理器"""

    def __init__(self):
        self.redis_client = redis.Redis(host="localhost",
        self.local_cache = {}  # L1：进程内缓存

    def _cache_key(self, question: str, pipeline: str) ->
        """生成缓存键"""
        normalized = question.strip().lower()
        return f"graphrag:{pipeline}:{hashlib.md5(normaliz

    def get(self, question: str, pipeline: str):
        """多级缓存查询"""
        key = self._cache_key(question, pipeline)

        # L1：进程内缓存（最快，容量小）
        if key in self.local_cache:
            return self.local_cache[key]

        # L2：Redis缓存（快，容量大）
        cached = self.redis_client.get(key)
        if cached:
            result = json.loads(cached)
            self.local_cache[key] = result  # 回填L1
            return result
```

```
        return None

    def set(self, question: str, pipeline: str, result: di
            ttl: int = 3600):
        """写入缓存"""
        key = self._cache_key(question, pipeline)

        # 写入L1
        self.local_cache[key] = result

        # 写入L2
        self.redis_client.setex(key, ttl, json.dumps(resul

    def invalidate(self, entity_name: str):
        """
        当知识图谱更新时，使相关缓存失效
        策略：简单粗暴地清除所有缓存（生产环境可以用更精细的策略）
        """
        # 清除Redis中所有graphrag相关缓存
        keys = self.redis_client.keys("graphrag:*")
        if keys:
            self.redis_client.delete(*keys)

        # 清除L1
        self.local_cache.clear()

        print(f"缓存已清除（实体 {entity_name} 更新触发）")
```

这些工程实践看似琐碎，却是GraphRAG从Demo走向生产的关键分水岭。一个好的Demo可以在三小时内搭起来，但一个能扛住生产流量的系统，往往需要数月的打磨。不要低估这些”脏活累活”的价值——它们才是工程能力的真正体现。

补充深入：从检索到生成的全链路优化

检索结果的”翻译”问题

GraphRAG有一个常被忽视的难题：**如何把图检索结果”翻译”成LLM能理解的上下文?**

向量检索的结果是文档片段，天然就是文本，LLM直接就能读懂。但图谱检索的结果是结构化的——节点、关系、属性、路径——如果不做适当的”翻译”，LLM可能会误解甚至忽略这些信息。

我们试过三种翻译策略：

策略一：JSON直传。 把图查询结果直接序列化为JSON塞进Prompt。优点是信息完整，缺点是LLM对JSON的理解有时不稳定，尤其是深层嵌套结构。

策略二：三元组文本化。 把每条关系翻译成”主语–谓语–宾语”的自然语言句子。比如 `(:Person{name:"张三"})-[:WORKS_AT]->(:Company{name:"腾讯"})` 翻译为”张三在腾讯工作”。这种方式LLM理解最准确，但会丢失部分属性信息。

策略三：结构化描述。 把子图转化为结构化的描述文本，保留层级关系但使用自然语言表达。这是我们目前效果最好的方式：

```
def graph_to_context(subgraph_results):
    """将图检索结果转化为LLM友好的上下文文本"""
    lines = []
    for record in subgraph_results:
        entity = record.get("entity", "未知")
        description = record.get("description", "")

        line = f"【{entity}】"
        if description:
            line += f" {description}"

        # 出向关系
        for rel in record.get("outgoing", []):
            if rel.get("target"):
                line += f"\n  • {entity} 的{rel['relation'

        # 入向关系
        for rel in record.get("incoming", []):
            if rel.get("source"):
                line += f"\n  • {rel['source']} 的{rel['re

        lines.append(line)

    return "\n\n".join(lines)
```

长上下文下的RAG困境

随着LLM上下文窗口越来越大（GPT-4o支持128K token，Claude 3.5支持200K token），一个自然的疑问是：**还需要RAG吗？把所有文档都塞进Prompt不就行了？**

答案是：**仍然需要，但原因变了。**

长上下文解决了”能塞多少”的问题，但没有解决以下问题：

注意力稀释。 研究表明，LLM对Prompt中间部分的关注度显著低于开头和结尾（这就是所谓的”中间丢失”现象，Lost in the Middle）。塞入太多不相关的文档反而会干扰模型对关键信息的注意力。

成本问题。 128K token的输入，按GPT-4o的定价约0.64美元一次查询。如果每个用户提问都全量灌入，成本很快会失控。RAG通过精准检索，把实际输入控制在5K-15K token，成本降低一到两个数量级。

延迟问题。 128K token的推理延迟可能是5K token的10-20倍。对于交互式应用，这种延迟是用户无法接受的。

所以，RAG在长上下文时代的角色从”必须”变成了”优化”——但仍然是极其重要的优化。

评估GraphRAG的独特挑战

评估Vector RAG相对直接——检索的文档和标准答案对比就行。但评估GraphRAG更复杂，因为你需要同时评估：

图谱质量：实体是否完整？关系是否准确？属性是否过时？

查询质量：LLM生成的Cypher是否正确？是否遗漏了关键路径？

推理质量：多跳推理链是否合理？是否存在逻辑断裂？

答案质量：最终回答是否准确、完整、有依据？

一个实用的评估框架是”四层评估法”：

```
第一层：图谱覆盖率（Graph Coverage）
  - 测试集中涉及的所有实体是否在图谱中存在？
  - 测试集中涉及的所有关系是否在图谱中体现？

第二层：检索精确率（Retrieval Precision）
  - 图谱检索是否找到了正确的子图？
  - 是否遗漏了关键的关系路径？

第三层：推理正确性（Reasoning Correctness）
  - LLM是否正确理解了检索到的结构化信息？
  - 多跳推理链是否存在逻辑跳跃或断裂？

第四层：回答质量（Answer Quality）
  - 最终回答是否准确回答了用户问题？
  - 是否正确引用了图谱中的事实依据？
  - 是否标注了不确定性或信息不足的部分？
```

每一层都需要独立的评估指标和测试用例。建议为每层至少准备20个标准测试案例，覆盖简单查询到复杂多跳推理的各个难度级别。

真实项目中的踩坑记录

最后分享几个我在GraphRAG项目中踩过的坑，希望对你的实践有帮助：

坑一：过度依赖LLM生成Cypher。 LangChain的GraphCypherQAChain让LLM直接生成Cypher查询，看起来很酷，但在生产中LLM生成的Cypher出错率高达30–40%。尤其是涉

及复杂JOIN、聚合和路径查询时，LLM经常生成语法正确但语义错误的Cypher。我们的解决方案是：预定义一组Cypher模板，LLM只负责填充参数，而不是从头生成整个查询。

坑二：图谱太"干净"反而不好。 早期我们追求图谱的"纯净"——每个实体只有一个规范名称，每个关系只保留最核心的。结果发现，这种"过度清洗"反而让图谱检索效果变差。因为用户提问时使用的表述往往是不规范的，保留一些冗余的别名关系、描述性文本节点，反而能提升检索召回率。

坑三：忽视了检索和生成的衔接。 我们曾经花大量时间优化检索质量（把Recall@5从70%提升到90%），却发现最终的问答质量提升不明显。排查后发现，问题出在检索结果的组织方式上——20条检索结果被粗暴地拼接在一起，信息重复、矛盾、层次混乱，LLM读完反而"蒙圈"了。后来我们加了一个"上下文整理"环节（去重、排序、标注来源），效果立竿见影。

坑四：没有做查询缓存。 在生产环境中，我们发现约40%的查询是重复或高度相似的（尤其是高频问题）。不做缓存意味着每次都完整地跑一遍检索+生成，白白烧Token。加上Redis缓存后，平均响应延迟从3.2秒降到了0.8秒，日均Token消耗减少了35%。

这些经验教训有一个共同的主题：**GraphRAG不是一个算法问题，而是一个系统工程问题。** 算法只占30%的功夫，剩下70%是数据质量、工程架构和持续运维。

选型决策树：什么时候用Vector RAG、Graph RAG还是Hybrid?

面对一个新项目，最常见的疑问是:”我该选哪种RAG? ”以下是我在实践中总结的决策树:

纯Vector RAG就够了，当： – 你的知识源主要是长文本（文档、报告、文章） – 用户问题以语义搜索为主（”有没有关于XX的案例”） – 项目时间紧，需要快速上线 – 团队缺乏图数据库经验 – 知识更新频率低（周级别以上）

应该考虑Graph RAG，当： – 你的知识源包含大量结构化关系（组织架构、供应链、知识体系） – 用户经常问关系推理类问题（”A和B有什么关系”“谁负责XX项目”） – 需要可解释的推理过程（金融合规、医疗决策） – 数据源之间需要关联分析（跨系统查询）

必须上Hybrid RAG，当： – 同时有大量非结构化文本和结构化关系数据 – 用户问题类型多样（既有事实查询，也有语义搜索） – 对回答质量有高要求（准确率>90%） – 预算充足（Hybrid的Token消耗是单一RAG的2–3倍）

还有一个经验法则：**先用Vector RAG做一个MVP，建立基线指标。** 然后逐步引入Graph RAG，对比每次改进的效果。不要一上来就追求”最复杂最完善的架构”——先跑起来，再迭代优化。

在知识图谱和RAG的结合这条路上，务实比炫技更重要。记住，你的用户不关心你用的是哪种RAG，他们只关心能不能快速得到准确的回答。

第十四章 性能优化与生产部署

> *难度：★★★☆☆ 前置要求：完成第4-12章动手实践，具备Docker和Linux基础运维经验 本章代码：* `https://github.com/sloth-laok/knowledge-graph-intro/tree/main/ch14`

引子

2024年初，我帮一家做智能客服的创业公司做技术评审。他们的知识图谱系统跑在MacBook Pro上，Demo演示行云流水，客户连连称赞。

三个月后，系统上线。第一天，500个并发用户涌入。Neo4j查询延迟飙升到8秒，Qdrant的内存占用触发OOM，整个系统像一只被踩了尾巴的猫——尖叫着瘫了。

CTO给我打电话："老王，我们的图谱在笔记本上跑得好好的，怎么一上生产就拉胯了？"

这个问题，几乎每个知识图谱工程师都会遇到。**从Demo到Production，中间隔着一片海。** 这片海里藏着数据同步的暗流、高并发的浪涛、监控告警的灯塔，以及安全合规的礁石。

本章就是一份"渡海指南"。我们将系统地讨论Neo4j集群部署、Qdrant分布式架构、数据同步策略、监控告警体系，以及一份从开发到生产的完整Checklist。

Neo4j集群与高可用

为什么单机不够用

单机Neo4j在开发和测试阶段绰绰有余，但生产环境面临三个核心挑战：

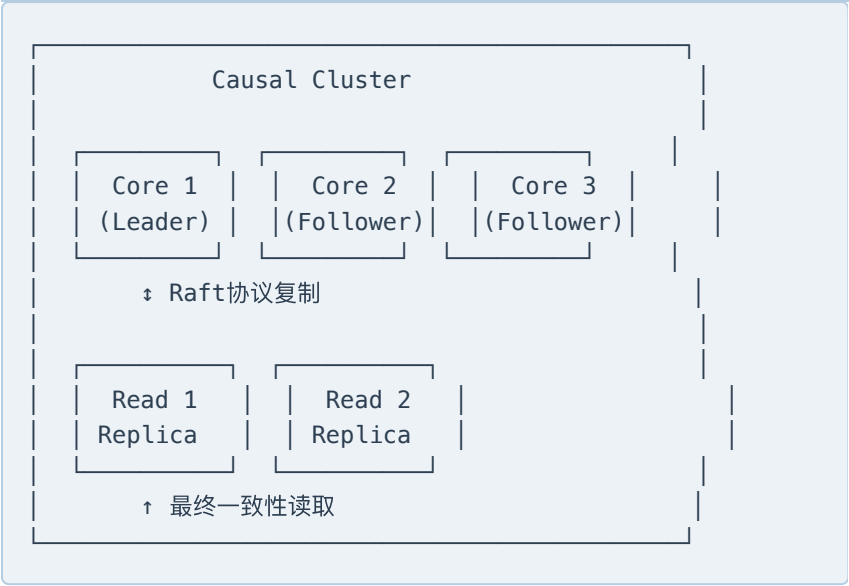
容量瓶颈：单机内存和磁盘限制了图谱规模。当节点数超过千万、关系数超过亿级时，单机查询性能急剧下降。

可用性风险：单点故障意味着整个系统的停摆。对于7x24在线的业务，哪怕是计划内的维护升级，也需要不停机方案。

读写冲突：高并发场景下，写操作（创建节点、更新关系）需要获取锁，与读操作竞争资源，导致整体吞吐量下降。

Neo4j Causal Clustering

Neo4j企业版提供了**Causal Clustering（因果集群）**方案，由三种角色的节点组成：



Core节点：负责事务处理和数据一致性。至少需要3个Core节点（Raft协议要求多数派存活）。所有写操作由Leader处理，通过Raft协议复制到Follower。

Read Replica节点：只读副本，用于分担读负载。它们从Core节点异步拉取数据，提供最终一致性的读取能力。可以横向扩展，数量不受限制。

```
# docker-compose-neo4j-cluster.yml
# Neo4j因果集群部署（3 Core + 2 Read Replica）

version: '3.8'

services:
  neo4j-core1:
    image: neo4j:5.20-enterprise
    container_name: neo4j-core1
    hostname: neo4j-core1
    environment:
      - NEO4J_AUTH=neo4j/your_strong_password
      - NEO4J_dbms_default__database=neo4j
      # 集群配置
      - NEO4J_dbms_cluster_discovery_endpoints=neo4j-core1
      - NEO4J_dbms_cluster_default__databases__mode=PRIMAR
      - NEO4J_dbms_cluster_raft_advertised__address=neo4j-
      - NEO4J_dbms_cluster_raft_listen__address=0.0.0.0:70
      - NEO4J_dbms_cluster_discovery_advertised__address=n
      - NEO4J_dbms_cluster_discovery_listen__address=0.0.0
      - NEO4J_dbms_cluster_transaction_advertised__address
      - NEO4J_dbms_cluster_transaction_listen__address=0.0
      # 性能调优
      - NEO4J_server_memory_heap_initial__size=2G
      - NEO4J_server_memory_heap_max__size=4G
      - NEO4J_server_memory_pagecache_size=2G
    ports:
      - "7474:7474"
      - "7687:7687"
    volumes:
      - neo4j-core1-data:/data
      - neo4j-core1-logs:/logs
    networks:
      - neo4j-cluster
```

```
  neo4j-core2:
    image: neo4j:5.20-enterprise
    container_name: neo4j-core2
    hostname: neo4j-core2
    environment:
      - NEO4J_AUTH=neo4j/your_strong_password
      - NEO4J_dbms_cluster_discovery_endpoints=neo4j-core1
      - NEO4J_dbms_cluster_default__databases__mode=PRIMAR
      - NEO4J_dbms_cluster_raft_advertised__address=neo4j-
      - NEO4J_dbms_cluster_raft_listen__address=0.0.0.0:70
      - NEO4J_dbms_cluster_discovery_advertised__address=n
      - NEO4J_dbms_cluster_discovery_listen__address=0.0.0
      - NEO4J_dbms_cluster_transaction_advertised__address
      - NEO4J_dbms_cluster_transaction_listen__address=0.0
      - NEO4J_server_memory_heap_initial__size=2G
      - NEO4J_server_memory_heap_max__size=4G
      - NEO4J_server_memory_pagecache_size=2G
    volumes:
      - neo4j-core2-data:/data
      - neo4j-core2-logs:/logs
    networks:
      - neo4j-cluster

  neo4j-core3:
    image: neo4j:5.20-enterprise
    container_name: neo4j-core3
    hostname: neo4j-core3
    environment:
      - NEO4J_AUTH=neo4j/your_strong_password
      - NEO4J_dbms_cluster_discovery_endpoints=neo4j-core1
      - NEO4J_dbms_cluster_default__databases__mode=PRIMAR
      - NEO4J_dbms_cluster_raft_advertised__address=neo4j-
      - NEO4J_dbms_cluster_raft_listen__address=0.0.0.0:70
      - NEO4J_dbms_cluster_discovery_advertised__address=n
```

```
      - NEO4J_dbms_cluster_discovery_listen__address=0.0.0
      - NEO4J_dbms_cluster_transaction_advertised__address
      - NEO4J_dbms_cluster_transaction_listen__address=0.0
      - NEO4J_server_memory_heap_initial__size=2G
      - NEO4J_server_memory_heap_max__size=4G
      - NEO4J_server_memory_pagecache_size=2G
    volumes:
      - neo4j-core3-data:/data
      - neo4j-core3-logs:/logs
    networks:
      - neo4j-cluster

  neo4j-read1:
    image: neo4j:5.20-enterprise
    container_name: neo4j-read1
    hostname: neo4j-read1
    environment:
      - NEO4J_AUTH=neo4j/your_strong_password
      - NEO4J_dbms_cluster_discovery_endpoints=neo4j-core1
      - NEO4J_dbms_cluster_default__databases__mode=SECOND
      - NEO4J_server_memory_heap_initial__size=2G
      - NEO4J_server_memory_heap_max__size=4G
      - NEO4J_server_memory_pagecache_size=2G
    ports:
      - "7475:7474"
      - "7688:7687"
    volumes:
      - neo4j-read1-data:/data
    networks:
      - neo4j-cluster

  neo4j-read2:
    image: neo4j:5.20-enterprise
    container_name: neo4j-read2
    hostname: neo4j-read2
```

```
    environment:
      - NEO4J_AUTH=neo4j/your_strong_password
      - NEO4J_dbms_cluster_discovery_endpoints=neo4j-core1
      - NEO4J_dbms_cluster_default__databases__mode=SECOND
      - NEO4J_server_memory_heap_initial__size=2G
      - NEO4J_server_memory_heap_max__size=4G
      - NEO4J_server_memory_pagecache_size=2G
    ports:
      - "7476:7474"
      - "7689:7687"
    volumes:
      - neo4j-read2-data:/data
    networks:
      - neo4j-cluster

volumes:
  neo4j-core1-data:
  neo4j-core1-logs:
  neo4j-core2-data:
  neo4j-core2-logs:
  neo4j-core3-data:
  neo4j-core3-logs:
  neo4j-read1-data:
  neo4j-read2-data:

networks:
  neo4j-cluster:
    driver: bridge
```

读写分离策略

在应用层实现读写分离是提升集群性能的关键：

```
# neo4j_cluster_client.py
"""
Neo4j集群读写分离客户端
"""
from neo4j import GraphDatabase, RoutingControl
from typing import Optional, List, Dict, Any

class Neo4jClusterClient:
    """支持读写分离的Neo4j集群客户端"""

    def __init__(self, cluster_uris: List[str], auth: tupl
        # 使用bolt+routing协议，驱动自动发现集群拓扑
        self.driver = GraphDatabase.driver(
            cluster_uris[0],  # 连接任一Core节点即可
            auth=auth,
            max_connection_pool_size=100,
            connection_acquisition_timeout=30,
            max_transaction_retry_time=15,
            # 连接健康检查
            connection_timeout=10,
            keepalive=True
        )

    def read(self, query: str, params: Optional[Dict] = No
        """
        读操作：路由到Read Replica
        利用RoutingControl.READ让驱动自动选择读节点
        """
        records, _, _ = self.driver.execute_query(
            query,
            parameters_=params or {},
            routing_=RoutingControl.READ,
            database_="neo4j"
        )
```

```
        return [dict(record) for record in records]

    def write(self, query: str, params: Optional[Dict] = N
        """
        写操作：路由到Leader
        """
        records, _, _ = self.driver.execute_query(
            query,
            parameters_=params or {},
            routing_=RoutingControl.WRITE,
            database_="neo4j"
        )
        return [dict(record) for record in records]

    def read_complex(self, query: str, params: Optional[Di
                     timeout: int = 30) -> List[Dict]:
        """
        复杂读操作：使用显式事务，支持超时控制
        """
        with self.driver.session(
            default_access_mode=RoutingControl.READ,
            database="neo4j"
        ) as session:
            result = session.run(query, params or {}, time
            return [dict(record) for record in result]

    def write_in_transaction(self, query: str, params: Opt
        """
        事务写操作：确保原子性
        """
        with self.driver.session(
            default_access_mode=RoutingControl.WRITE,
            database="neo4j"
        ) as session:
            return session.execute_write(
```

```
                lambda tx: list(tx.run(query, params or {}
            )

    def batch_write(self, query: str, params_list: List[Di
        """
        批量写入：使用UNWIND减少事务开销
        """
        batch_query = f"""
        UNWIND $batch AS row
        {query}
        """
        with self.driver.session(
            default_access_mode=RoutingControl.WRITE
        ) as session:
            for i in range(0, len(params_list), batch_size
                batch = params_list[i:i+batch_size]
                session.execute_write(
                    lambda tx: list(tx.run(batch_query, ba
                )
        print(f"批量写入完成：{len(params_list)}条记录")

    def health_check(self) -> Dict:
        """集群健康检查"""
        try:
            result = self.driver.execute_query(
                "CALL dbms.cluster.overview() YIELD id, na
                routing_=RoutingControl.READ
            )
            return {"status": "healthy", "members": len(re
        except Exception as e:
            return {"status": "unhealthy", "error": str(e)

    def close(self):
        self.driver.close()
```

```
# 使用示例
client = Neo4jClusterClient(
    cluster_uris=["bolt://neo4j-core1:7687"],
    auth=("neo4j", "your_strong_password")
)

# 读操作 → 自动路由到Read Replica
entities = client.read("""
    MATCH (n:Person)
    RETURN n.name AS name, n.title AS title
    LIMIT 100
""")

# 写操作 → 自动路由到Leader
client.write("""
    MERGE (n:Person {name: $name})
    SET n.updated_at = datetime()
""", {"name": "张三"})

# 批量写入
batch_data = [{"name": f"User_{i}", "age": 20 + i % 50} fo
client.batch_write(
    "MERGE (n:Person {name: row.name}) SET n.age = row.age
    batch_data
)
```

Neo4j Fabric：跨库查询

当数据量大到需要分库存储时，Neo4j Fabric提供了跨库查询能力。它像一个”查询路由器”，将一条Cypher查询拆解为多个子查询，分发到不同的数据库执行，最后合并结果。

```
# fabric_example.py
"""
Neo4j Fabric 跨库查询示例

场景：电商知识图谱按品类分库
- kg_electronics: 电子产品子图
- kg_clothing: 服装子图
- kg_food: 食品子图
"""

# Fabric配置（neo4j.conf）
FABRIC_CONFIG = """
# 启用Fabric
dbms.fabric.routing.enabled=true

# 定义Fabric数据库
dbms.fabric.database.name=fabric_all

# 数据源配置
dbms.fabric.routing.group.1.uri=bolt://neo4j-core1:7687
dbms.fabric.routing.group.1.database=kg_electronics
dbms.fabric.routing.group.2.uri=bolt://neo4j-core1:7687
dbms.fabric.routing.group.2.database=kg_clothing
dbms.fabric.routing.group.3.uri=bolt://neo4j-core1:7687
dbms.fabric.routing.group.3.database=kg_food
"""

# 跨库查询Cypher示例
FABRIC_QUERY = """
// 查询所有品类中与"华为"相关的产品
USE fabric_all

CALL {
    USE kg_electronics
```

```
    MATCH (p:Product)-[:BRAND]->(b:Brand {name: '华为'})
    RETURN p.name AS product, '电子产品' AS category
}

UNION

CALL {
    USE kg_clothing
    MATCH (p:Product)-[:BRAND]->(b:Brand {name: '华为'})
    RETURN p.name AS product, '服装' AS category
}

UNION

CALL {
    USE kg_food
    MATCH (p:Product)-[:BRAND]->(b:Brand {name: '华为'})
    RETURN p.name AS product, '食品' AS category
}

RETURN product, category
ORDER BY category, product
"""
```

Neo4j查询性能优化要点

无论是否使用集群，以下优化策略都至关重要：

索引优化：

```
// 1. 创建复合索引（最常用）
CREATE INDEX idx_person_name_dept FOR (p:Person) ON (p.nam

// 2. 全文索引（支持模糊搜索）
CREATE FULLTEXT INDEX nameSearch FOR (n:Person|Company) ON

// 3. 关系索引（加速关系遍历）
CREATE INDEX idx_works_at FOR ()-[r:WORKS_AT]-() ON (r.sin

// 4. 查看索引使用情况
CALL db.stats.profile("MATCH (p:Person {name: '张三'}) RETU
```

Cypher优化：

```
// ❌ 反面教材：笛卡尔积
MATCH (a:Person), (b:Company)
WHERE a.name = '张三' AND b.name = '腾讯'
MERGE (a)-[:WORKS_AT]->(b)

// ✅ 正确做法：精确匹配
MATCH (a:Person {name: '张三'})
MATCH (b:Company {name: '腾讯'})
MERGE (a)-[:WORKS_AT]->(b)

// ❌ 反面教材：遍历整个图
MATCH (a)-[*]->(b)
RETURN a, b

// ✅ 正确做法：限制遍历深度
MATCH (a:Person {name: '张三'})-[*1..3]->(b)
RETURN a, b
```

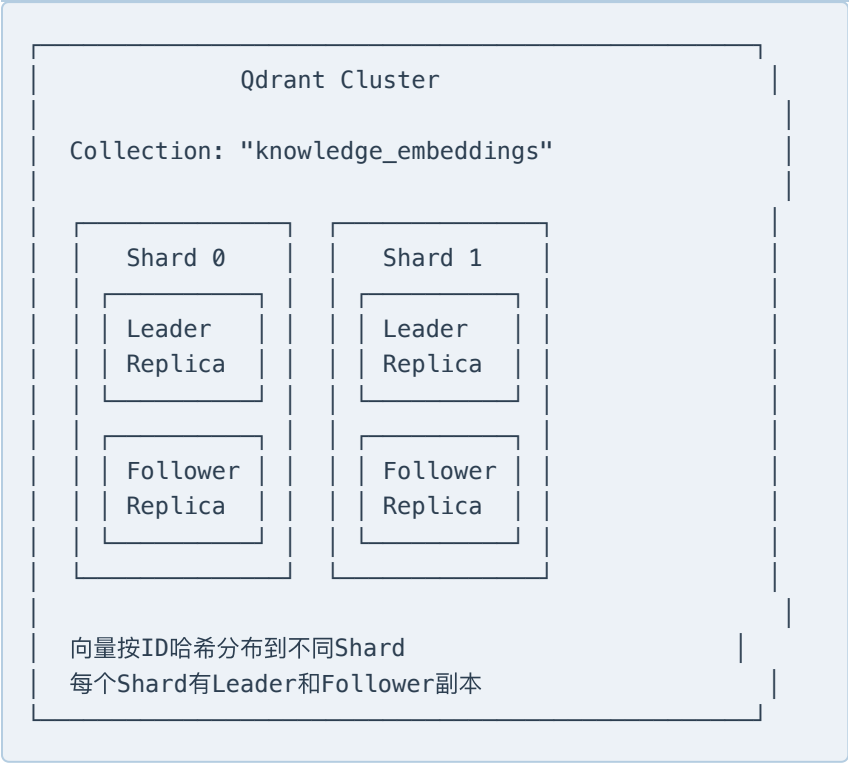
Qdrant分布式部署

为什么需要分布式向量数据库

当你的Embedding向量规模超过千万级别时，单机Qdrant会面临： – **内存瓶颈**：1亿个768维float32向量约需300GB内存 – **查询延迟**：大规模近似搜索的延迟线性增长 – **可用性**：单点故障导致向量检索服务不可用

Qdrant分布式架构

Qdrant的分布式架构基于**Raft一致性协议**，支持分片（Sharding）和副本（Replication）：


Qdrant Cluster
Collection: "knowledge_embeddings"
Shard 0
Leader Replica
Follower Replica
Shard 1
Leader Replica
Follower Replica
向量按ID哈希分布到不同Shard
每个Shard有Leader和Follower副本

```
# docker-compose-qdrant-cluster.yml
version: '3.8'

services:
  qdrant-node1:
    image: qdrant/qdrant:v1.9.0
    container_name: qdrant-node1
    hostname: qdrant-node1
    environment:
      - QDRANT__CLUSTER__ENABLED=true
      - QDRANT__SERVICE__GRPC_PORT=6334
    ports:
      - "6333:6333"
      - "6334:6334"
    volumes:
      - qdrant-node1-data:/qdrant/storage
    command: ["./qdrant", "--uri", "http://qdrant-node1:63
    networks:
      - qdrant-cluster

  qdrant-node2:
    image: qdrant/qdrant:v1.9.0
    container_name: qdrant-node2
    hostname: qdrant-node2
    environment:
      - QDRANT__CLUSTER__ENABLED=true
    volumes:
      - qdrant-node2-data:/qdrant/storage
    command: ["./qdrant", "--uri", "http://qdrant-node2:63
    networks:
      - qdrant-cluster

  qdrant-node3:
    image: qdrant/qdrant:v1.9.0
```

```
    container_name: qdrant-node3
    hostname: qdrant-node3
    environment:
      - QDRANT__CLUSTER__ENABLED=true
    volumes:
      - qdrant-node3-data:/qdrant/storage
    command: ["./qdrant", "--uri", "http://qdrant-node3:63
    networks:
      - qdrant-cluster

volumes:
  qdrant-node1-data:
  qdrant-node2-data:
  qdrant-node3-data:

networks:
  qdrant-cluster:
    driver: bridge
```

```
# qdrant_cluster_setup.py
"""
Qdrant集群初始化：创建分布式Collection
"""
from qdrant_client import QdrantClient
from qdrant_client.http.models import (
    VectorParams, Distance, ReplicasConfig,
    ShardingMethod, CollectionStatus
)

# 连接集群（连接任一节点即可）
client = QdrantClient(host="localhost", port=6333)

# 初始化集群（只需在第一个节点执行一次）
# POST /cluster/init
# 或通过API：
client.init_shards_group()

# 创建分布式Collection
client.create_collection(
    collection_name="knowledge_embeddings",
    vectors_config=VectorParams(
        size=1536,           # text-embedding-3-small的维度
        distance=Distance.COSINE,
    ),
    # 分片配置
    shard_number=6,          # 6个分片，均匀分布到3个节点
    sharding_method=ShardingMethod.AUTO,
    # 副本配置
    replication_factor=2,    # 每个分片2个副本（1 Leader + 1
    write_consistency_factor=1,  # 写入至少1个副本成功即可
)

# 验证Collection状态
```

```
info = client.get_collection("knowledge_embeddings")
print(f"Collection状态: {info.status}")
print(f"分片数: {info.config.params.shard_number}")
print(f"副本因子: {info.config.params.replication_factor}")
print(f"向量数: {info.points_count}")

# 创建索引优化配置
from qdrant_client.http.models import HnswConfigDiff

client.update_collection(
    collection_name="knowledge_embeddings",
    hnsw_config=HnswConfigDiff(
        m=16,                # 每个节点的最大连接数
        ef_construct=200,    # 构建时的搜索范围
        full_scan_threshold=10000  # 低于此阈值使用暴力搜索
    )
)
print("HNSW索引配置已更新")
```

Qdrant性能调优

```
# qdrant_optimization.py
"""
Qdrant查询性能优化策略
"""
from qdrant_client import QdrantClient
from qdrant_client.http.models import (
    SearchRequest, SearchParams, Filter,
    FieldCondition, MatchValue, QuantizationConfig,
    ScalarQuantization, ScalarType, QuantizationType
)

client = QdrantClient(host="localhost", port=6333)

# 策略1：量化压缩（减少内存占用，提升搜索速度）
client.update_collection(
    collection_name="knowledge_embeddings",
    quantization_config=QuantizationConfig(
        scalar=ScalarQuantization(
            type=QuantizationType.SCALAR,
            scalar=ScalarType(
                type="int8",        # float32 → int8，内存减
                always_ram=True     # 量化向量常驻内存
            )
        )
    )
)

# 策略2：Payload索引（加速过滤查询）
from qdrant_client.http.models import PayloadSchemaType

client.create_payload_index(
    collection_name="knowledge_embeddings",
```

```
    field_name="category",
    field_schema=PayloadSchemaType.KEYWORD
)
client.create_payload_index(
    collection_name="knowledge_embeddings",
    field_name="created_at",
    field_schema=PayloadSchemaType.DATETIME
)

# 策略3：批量搜索（减少网络往返）
search_requests = [
    SearchRequest(
        vector=query_vector_1,
        limit=5,
        with_payload=True,
        params=SearchParams(hnsw_ef=128, exact=False)
    ),
    SearchRequest(
        vector=query_vector_2,
        limit=5,
        filter=Filter(must=[
            FieldCondition(key="category", match=MatchValu
        ]),
        with_payload=True
    ),
]

results = client.search_batch(
    collection_name="knowledge_embeddings",
    requests=search_requests
)

for i, result in enumerate(results):
    print(f"\n查询{i+1}结果：")
    for hit in result:
```

```
        print(f"  {hit.payload.get('title', 'N/A')}: {hit.

# 策略4：推荐搜索（利用已有偏好做个性化检索）
from qdrant_client.http.models import RecommendRequest

recommend_results = client.recommend(
    collection_name="knowledge_embeddings",
    positive=[point_id_1, point_id_2],  # 用户喜欢的文档
    negative=[point_id_3],              # 用户不喜欢的文档
    limit=10
)
```

数据同步策略

生产环境中，知识图谱的数据通常来自多个上游系统。如何保持图谱与源数据的一致性，是一个核心工程挑战。

三种同步模式

模式一：定时全量同步（Scheduled Full Sync）

最简单粗暴的方式——定时把所有源数据重新灌入图谱。适合数据量小、更新频率低的场景。

```
# sync_scheduled.py
"""
定时全量同步：每天凌晨2点执行
适合：数据量 < 100万条，更新频率低（日级别）
"""
import schedule
import time
from datetime import datetime

def full_sync():
    """全量同步流程"""
    start_time = datetime.now()
    print(f"[{start_time}] 开始全量同步...")

    # 1. 从上游系统拉取最新数据
    source_data = fetch_from_source_systems()
    print(f"  拉取到 {len(source_data)} 条记录")

    # 2. 数据清洗和转换
    cleaned_data = clean_and_transform(source_data)

    # 3. 标记旧数据为"待删除"
    mark_existing_as_stale()

    # 4. 写入新数据（UPSERT）
    batch_upsert(cleaned_data)

    # 5. 删除仍标记为"待删除"的旧数据
    remove_stale_records()

    # 6. 重建索引
    rebuild_indexes()

    elapsed = (datetime.now() - start_time).total_seconds(
```

```
    print(f"[{datetime.now()}] 全量同步完成，耗时 {elapsed:.1

# 定时任务
schedule.every().day.at("02:00").do(full_sync)

while True:
    schedule.run_pending()
    time.sleep(60)
```

模式二：CDC（Change Data Capture，变更数据捕获）

监听数据库的变更日志（如MySQL的binlog），实时捕获增删改操作，增量同步到图谱。

```
# sync_cdc.py
"""
CDC增量同步：监听MySQL binlog，实时同步到Neo4j
使用python-mysql-replication库
"""
import pymysql
from pymysqlreplication import BinLogStreamReader
from pymysqlreplication.row_event import (
    WriteRowsEvent, UpdateRowsEvent, DeleteRowsEvent
)
from neo4j import GraphDatabase
import json

# MySQL连接配置
mysql_settings = {
    "host": "mysql-master",
    "port": 3306,
    "user": "repl_user",
    "passwd": "repl_password"
}

# Neo4j连接
neo4j_driver = GraphDatabase.driver(
    "bolt://neo4j-core1:7687",
    auth=("neo4j", "your_password")
)

def sync_cdc():
    """CDC同步主循环"""
    stream = BinLogStreamReader(
        connection_settings=mysql_settings,
        server_id=100,           # 唯一server_id
        blocking=True,           # 持续监听
        only_events=[WriteRowsEvent, UpdateRowsEvent, Dele
```

```
        only_tables=["users", "products", "orders"],
        resume_stream=True       # 从上次位置继续
    )

    batch = []
    batch_size = 100

    for binlog_event in stream:
        for row in binlog_event.rows:
            if isinstance(binlog_event, WriteRowsEvent):
                batch.append(("CREATE", binlog_event.table
            elif isinstance(binlog_event, UpdateRowsEvent)
                batch.append(("UPDATE", binlog_event.table
            elif isinstance(binlog_event, DeleteRowsEvent)
                batch.append(("DELETE", binlog_event.table

        # 批量处理
        if len(batch) >= batch_size:
            process_batch(batch)
            batch = []

    # 处理剩余
    if batch:
        process_batch(batch)

def process_batch(batch):
    """批量处理变更事件"""
    with neo4j_driver.session() as session:
        for event_type, table, data in batch:
            try:
                if event_type == "CREATE":
                    handle_create(session, table, data)
                elif event_type == "UPDATE":
                    handle_update(session, table, data)
                elif event_type == "DELETE":
```

```
                    handle_delete(session, table, data)
            except Exception as e:
                print(f"处理失败 [{event_type}] {table}: {e
                # 写入死信队列，后续人工处理
                write_to_dead_letter_queue(event_type, tab

def handle_create(session, table, data):
    """处理新增事件"""
    if table == "users":
        session.run("""
            MERGE (u:User {id: $id})
            SET u.name = $name,
                u.email = $email,
                u.department = $department,
                u.synced_at = datetime()
        """, id=data["id"], name=data["name"],
            email=data["email"], department=data.get("depa
    elif table == "products":
        session.run("""
            MERGE (p:Product {id: $id})
            SET p.name = $name,
                p.category = $category,
                p.price = $price,
                p.synced_at = datetime()
        """, id=data["id"], name=data["name"],
            category=data["category"], price=data["price"]

def handle_update(session, table, data):
    """处理更新事件"""
    # 与CREATE类似，使用MERGE + SET
    handle_create(session, table, data)

def handle_delete(session, table, data):
    """处理删除事件"""
    if table == "users":
```

```
        session.run("""
            MATCH (u:User {id: $id})
            DETACH DELETE u
        """, id=data["id"])

# 启动CDC同步
sync_cdc()
```

模式三：事件驱动同步（Event-Driven）

上游系统通过消息队列（Kafka/RabbitMQ）发布变更事件，图谱服务消费事件并更新。

```
# sync_event_driven.py
"""
事件驱动同步：消费Kafka消息，实时更新图谱
"""
from kafka import KafkaConsumer
from neo4j import GraphDatabase
import json

consumer = KafkaConsumer(
    "knowledge_graph_updates",
    bootstrap_servers=["kafka-broker1:9092", "kafka-broker
    group_id="neo4j-sync-group",
    auto_offset_reset="earliest",
    enable_auto_commit=False,     # 手动提交offset
    value_deserializer=lambda m: json.loads(m.decode("utf-
    max_poll_records=500          # 每次最多拉取500条
)

neo4j_driver = GraphDatabase.driver(
    "bolt://neo4j-core1:7687",
    auth=("neo4j", "your_password")
)

# 事件处理器注册表
handlers = {}

def register_handler(event_type):
    def decorator(func):
        handlers[event_type] = func
        return func
    return decorator

@register_handler("entity.created")
def handle_entity_created(event):
```

```
    """实体创建"""
    entity = event["payload"]
    with neo4j_driver.session() as session:
        labels = ":".join(entity["labels"])
        props = entity["properties"]
        session.run(f"""
            MERGE (n:{labels} {{id: $id}})
            SET n += $props, n.synced_at = datetime()
        """, id=entity["id"], props=props)

@register_handler("entity.updated")
def handle_entity_updated(event):
    """实体更新"""
    entity = event["payload"]
    with neo4j_driver.session() as session:
        session.run("""
            MATCH (n {id: $id})
            SET n += $changes, n.synced_at = datetime()
        """, id=entity["id"], changes=entity["changes"])

@register_handler("entity.deleted")
def handle_entity_deleted(event):
    """实体删除"""
    entity_id = event["payload"]["id"]
    with neo4j_driver.session() as session:
        session.run("MATCH (n {id: $id}) DETACH DELETE n",

@register_handler("relationship.created")
def handle_relationship_created(event):
    """关系创建"""
    rel = event["payload"]
    with neo4j_driver.session() as session:
        session.run(f"""
            MATCH (a {{id: $source_id}})
            MATCH (b {{id: $target_id}})
```

```
            MERGE (a)-[r:{rel['type']}]->(b)
            SET r += $props, r.synced_at = datetime()
        """, source_id=rel["source_id"],
            target_id=rel["target_id"],
            props=rel.get("properties", {}))

def consume_loop():
    """消息消费主循环"""
    print("事件驱动同步服务已启动，等待消息...")

    for message in consumer:
        try:
            event = message.value
            event_type = event.get("type")

            if event_type in handlers:
                handlers[event_type](event)
            else:
                print(f"未知事件类型: {event_type}")

            # 每处理一条消息就提交一次offset
            consumer.commit()

        except Exception as e:
            print(f"处理消息失败: {e}, message: {message.val
            # 不提交offset，下次重试
            # 连续失败3次则发送到死信topic
            continue

# 启动
consume_loop()
```

同步策略对比

表 18–1

维度	定时全量	CDC	事件驱动
实时性	低（小时级）	高（秒级）	高（毫秒级）
实现复杂度	低	中	高
对源系统侵入	高（全量读取）	低（读日志）	无（被动消费）
数据一致性	最终一致	最终一致	最终一致
适用场景	小数据集，低频更新	已有数据库，增量同步	微服务架构，实时性要求高
基础设施依赖	无	数据库binlog权限	消息队列

监控与告警：Prometheus + Grafana

监控体系设计

一个完善的知识图谱监控系统需要覆盖四个层面：

```
┌─────────────────────────────────────────┐
│              业务指标层                  │
│  查询成功率、平均响应时间、QPS            │
├─────────────────────────────────────────┤
│              应用指标层                  │
│  Neo4j查询耗时、Qdrant搜索延迟           │
│  连接池使用率、事务排队数                │
├─────────────────────────────────────────┤
│              基础设施层                  │
│  CPU、内存、磁盘IO、网络带宽             │
├─────────────────────────────────────────┤
│              数据质量层                  │
│  节点增长率、孤立节点数、关系完整性      │
└─────────────────────────────────────────┘
```

Neo4j指标暴露与采集

```
# prometheus-neo4j.yml
# Prometheus采集Neo4j指标

# Neo4j配置：启用指标暴露
# 在neo4j.conf中添加：
NEO4J_METRICS_CONFIG = """
dbms.metrics.enabled=true
dbms.metrics.prometheus.enabled=true
dbms.metrics.prometheus.endpoint=0.0.0.0:2004
"""
```

```
# prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  # Neo4j集群指标
  - job_name: 'neo4j-cluster'
    metrics_path: '/metrics'
    static_configs:
      - targets:
        - 'neo4j-core1:2004'
        - 'neo4j-core2:2004'
        - 'neo4j-core3:2004'
        - 'neo4j-read1:2004'
        - 'neo4j-read2:2004'
        labels:
          service: 'neo4j'

  # Qdrant集群指标
  - job_name: 'qdrant-cluster'
    metrics_path: '/metrics'
    static_configs:
      - targets:
        - 'qdrant-node1:6333'
        - 'qdrant-node2:6333'
        - 'qdrant-node3:6333'
        labels:
          service: 'qdrant'

  # 应用层自定义指标
  - job_name: 'kg-app'
    static_configs:
      - targets: ['kg-app:8080']
```

```
        labels:
          service: 'knowledge-graph-app'

# 告警规则
rule_files:
  - 'alert_rules.yml'

# Alertmanager配置
alerting:
  alertmanagers:
    - static_configs:
        - targets: ['alertmanager:9093']
```

自定义应用指标

```
# metrics.py
"""
应用层指标暴露：使用prometheus_client库
"""
from prometheus_client import (
    Counter, Histogram, Gauge, Summary,
    generate_latest, CONTENT_TYPE_LATEST
)
from flask import Flask, Response
import time
from functools import wraps

app = Flask(__name__)

# 定义指标
neo4j_query_total = Counter(
    'neo4j_query_total',
    'Total Neo4j queries',
    ['query_type', 'status']  # 读/写，成功/失败
)

neo4j_query_duration = Histogram(
    'neo4j_query_duration_seconds',
    'Neo4j query duration',
    ['query_type'],
    buckets=[0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0, 10.0]
)

qdrant_search_duration = Histogram(
    'qdrant_search_duration_seconds',
    'Qdrant search duration',
    buckets=[0.01, 0.05, 0.1, 0.2, 0.5, 1.0]
```

```
)

graph_node_count = Gauge(
    'graph_node_count',
    'Total nodes in knowledge graph',
    ['label']
)

graph_relationship_count = Gauge(
    'graph_relationship_count',
    'Total relationships in knowledge graph',
    ['type']
)

sync_lag_seconds = Gauge(
    'sync_lag_seconds',
    'Data sync lag from source system'
)

rag_query_total = Counter(
    'rag_query_total',
    'Total RAG queries',
    ['source', 'status']
)

# 装饰器：自动记录查询耗时
def track_neo4j_query(query_type="read"):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            start = time.time()
            try:
                result = func(*args, **kwargs)
                neo4j_query_total.labels(query_type=query_
                return result
```

```
            except Exception as e:
                neo4j_query_total.labels(query_type=query_
                raise
            finally:
                duration = time.time() - start
                neo4j_query_duration.labels(query_type=que
        return wrapper
    return decorator

# 暴露指标端点
@app.route('/metrics')
def metrics():
    return Response(
        generate_latest(),
        mimetype=CONTENT_TYPE_LATEST
    )

# 定期更新图谱统计指标
def update_graph_stats():
    """每分钟更新一次图谱统计"""
    from neo4j import GraphDatabase
    driver = GraphDatabase.driver("bolt://neo4j-core1:7687
                                  auth=("neo4j", "your_pas
    with driver.session() as session:
        # 节点统计
        result = session.run("""
            MATCH (n)
            RETURN labels(n)[0] AS label, count(n) AS cnt
        """)
        for record in result:
            label = record["label"] or "Unknown"
            graph_node_count.labels(label=label).set(recor

        # 关系统计
        result = session.run("""
```

```
            MATCH ()-[r]->()
            RETURN type(r) AS type, count(r) AS cnt
        """)
        for record in result:
            graph_relationship_count.labels(type=record["t

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

告警规则配置

```
# alert_rules.yml
groups:
  - name: neo4j_alerts
    rules:
      # Neo4j查询延迟过高
      - alert: Neo4jHighQueryLatency
        expr: histogram_quantile(0.95, neo4j_query_duratic
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Neo4j P95查询延迟超过2秒"
          description: "过去5分钟，P95查询延迟持续超过2秒，当前

      # Neo4j连接池耗尽
      - alert: Neo4jConnectionPoolExhausted
        expr: neo4j_connection_pool_active / neo4j_connect
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "Neo4j连接池使用率超过90%"

      # Neo4j集群成员离线
      - alert: Neo4jClusterMemberDown
        expr: neo4j_cluster_member_available == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Neo4j集群节点离线: {{ $labels.instance }
```

```
  - name: qdrant_alerts
    rules:
      # Qdrant搜索延迟过高
      - alert: QdrantHighSearchLatency
        expr: histogram_quantile(0.95, qdrant_search_durat
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Qdrant P95搜索延迟超过500ms"

      # Qdrant内存使用过高
      - alert: QdrantHighMemoryUsage
        expr: qdrant_memory_usage_bytes / qdrant_memory_li
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Qdrant内存使用率超过85%"

  - name: sync_alerts
    rules:
      # 数据同步延迟过大
      - alert: SyncLagHigh
        expr: sync_lag_seconds > 300
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "数据同步延迟超过5分钟"

      # 同步服务停止
      - alert: SyncServiceDown
        expr: up{job="kg-sync"} == 0
        for: 2m
```

```
        labels:
          severity: critical
        annotations:
          summary: "数据同步服务不可用"
```

Grafana仪表盘

```
{
  "dashboard": {
    "title": "知识图谱生产监控",
    "panels": [
      {
        "title": "Neo4j 查询QPS",
        "type": "graph",
        "targets": [
          {
            "expr": "rate(neo4j_query_total[5m])",
            "legendFormat": "{{query_type}} - {{status}}"
          }
        ]
      },
      {
        "title": "Neo4j P95 查询延迟",
        "type": "graph",
        "targets": [
          {
            "expr": "histogram_quantile(0.95, rate(neo4j_c
            "legendFormat": "P95 {{query_type}}"
          }
        ]
      },
      {
        "title": "Qdrant 搜索延迟",
        "type": "heatmap",
        "targets": [
          {
            "expr": "rate(qdrant_search_duration_seconds_b
          }
        ]
```

```
      },
      {
        "title": "图谱规模",
        "type": "stat",
        "targets": [
          {
            "expr": "sum(graph_node_count)",
            "legendFormat": "总节点数"
          },
          {
            "expr": "sum(graph_relationship_count)",
            "legendFormat": "总关系数"
          }
        ]
      },
      {
        "title": "数据同步延迟",
        "type": "gauge",
        "targets": [
          {
            "expr": "sync_lag_seconds",
            "legendFormat": "同步延迟(秒)"
          }
        ],
        "thresholds": [
          {"value": 0, "color": "green"},
          {"value": 60, "color": "yellow"},
          {"value": 300, "color": "red"}
        ]
      }
    ]
  }
}
```

从开发到生产：完整Checklist

阶段一：开发环境验收

- [ ] 所有Cypher查询经过EXPLAIN分析，无笛卡尔积
- [ ] 关键查询路径有索引覆盖
- [ ] 单元测试覆盖率 > 80%
- [ ] 图数据库Schema文档化
- [ ] 数据质量规则明确定义
- [ ] RAG系统基线指标已建立（Recall@K、Faithfulness等）

阶段二：性能测试

```
# load_test.py
"""
知识图谱系统压力测试
使用locust进行负载测试
"""
from locust import HttpUser, task, between
import random

class KnowledgeGraphUser(HttpUser):
    wait_time = between(0.5, 2)

    @task(5)
    def search_entities(self):
        """实体搜索（高频操作）"""
        entities = ["知识图谱", "机器学习", "深度学习", "自然语
        self.client.post("/api/search", json={
            "query": random.choice(entities),
            "search_type": "entity"
        })

    @task(3)
    def graph_traverse(self):
        """图谱遍历（中频操作）"""
        self.client.post("/api/traverse", json={
            "start_entity": "知识图谱",
            "max_hops": 3
        })

    @task(2)
    def rag_query(self):
        """RAG问答（低频但高成本操作）"""
        questions = [
```

```
            "知识图谱在金融领域有哪些应用？",
            "什么是GraphRAG？",
            "Neo4j和JanusGraph有什么区别？"
        ]
        self.client.post("/api/rag/ask", json={
            "question": random.choice(questions)
        })

    @task(1)
    def write_entity(self):
        """写入操作（最低频）"""
        self.client.post("/api/entities", json={
            "name": f"TestEntity_{random.randint(1, 100000
            "type": "Concept",
            "description": "压力测试生成的实体"
        })
```

阶段三：安全加固

```
# security_checklist.yml
# 安全检查清单

authentication:
  - name: "Neo4j密码策略"
    check: |
      - 密码长度 >= 16位
      - 包含大小写字母、数字、特殊字符
      - 禁止使用默认密码
      - 90天强制更换

  - name: "API密钥管理"
    check: |
      - 所有API密钥存储在密钥管理服务（Vault/AWS Secrets Manage
      - 密钥定期轮换
      - 不在代码或配置文件中硬编码密钥

authorization:
  - name: "Neo4j RBAC"
    config: |
      // 创建角色
      CREATE ROLE reader;
      GRANT TRAVERSE ON GRAPH neo4j TO reader;
      GRANT MATCH {*} ON GRAPH neo4j TO reader;

      CREATE ROLE writer;
      GRANT ALL ON GRAPH neo4j TO writer;

      CREATE ROLE admin;
      GRANT ROLE admin TO user_admin;

      // 最小权限原则
```

```
      CREATE ROLE rag_service;
      GRANT MATCH {*} ON GRAPH neo4j TO rag_service;
      GRANT CREATE ON GRAPH neo4j NODES RAGCache TO rag_se

network:
  - name: "网络隔离"
    check: |
      - Neo4j/Qdrant不暴露公网端口
      - 使用VPC/安全组限制访问来源
      - API Gateway作为唯一入口
      - TLS 1.3加密所有通信
      - 内部服务间使用mTLS

backup:
  - name: "备份策略"
    config: |
      # Neo4j在线备份（企业版）
      neo4j-admin database backup neo4j \
        --to-path=/backup/neo4j \
        --type=FULL \
        --compress=true

      # 定时备份（crontab）
      0 3 * * * /opt/neo4j/bin/neo4j-admin database backup

      # Qdrant快照
      curl -X POST "http://qdrant:6333/collections/knowled

      # 备份保留策略
      - 每日全量备份，保留7天
      - 每周全量备份，保留4周
      - 每月全量备份，保留12个月
      - 备份异地存储（跨区域/跨云）

  - name: "灾难恢复"
```

```
    check: |
      - RTO（恢复时间目标）< 1小时
      - RPO（恢复点目标）< 5分钟
      - 每季度进行灾难恢复演练
      - 恢复流程文档化
```

阶段四：上线部署

```
# docker-compose-production.yml
# 生产环境完整部署

version: '3.8'

services:
  # API Gateway
  nginx:
    image: nginx:1.25-alpine
    ports:
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl:/etc/nginx/ssl
    depends_on:
      - kg-app
    networks:
      - frontend
      - backend

  # 知识图谱应用服务
  kg-app:
    image: your-registry/kg-app:latest
    environment:
      - NEO4J_URI=bolt://neo4j-core1:7687
      - QDRANT_HOST=qdrant-node1
      - REDIS_URL=redis://redis:6379
      - LOG_LEVEL=INFO
    deploy:
      replicas: 3
      resources:
        limits:
```

```
          cpus: '4'
          memory: 8G
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/h
      interval: 30s
      timeout: 10s
      retries: 3
    networks:
      - backend

  # Redis缓存层
  redis:
    image: redis:7-alpine
    command: redis-server --maxmemory 2gb --maxmemory-poli
    volumes:
      - redis-data:/data
    networks:
      - backend

  # Prometheus监控
  prometheus:
    image: prom/prometheus:v2.50.0
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/promet
      - ./monitoring/alert_rules.yml:/etc/prometheus/alert
      - prometheus-data:/prometheus
    networks:
      - monitoring

  # Grafana仪表盘
  grafana:
    image: grafana/grafana:10.4.0
    ports:
      - "3000:3000"
    environment:
```

```
      - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_ADMIN_PASSWOF
    volumes:
      - ./monitoring/grafana/dashboards:/var/lib/grafana/d
      - ./monitoring/grafana/provisioning:/etc/grafana/pro
    networks:
      - monitoring

volumes:
  redis-data:
  prometheus-data:

networks:
  frontend:
  backend:
    internal: true    # backend网络不允许外部访问
  monitoring:
    internal: true
```

动手练习

练习1：搭建Neo4j集群（★★★☆☆）

使用Docker Compose搭建一个3 Core + 1 Read Replica的Neo4j因果集群： 1. 验证集群状态：所有节点在线，Leader已选举 2. 在Leader上写入1000条测试数据 3. 分别在Core节点和Read Replica节点上执行查询，验证读写分离 4. 模拟一个Core节点宕机，验证集群仍然可用 5. 恢复宕机节点，验证数据自动同步

练习2：Qdrant性能基准测试（★★☆☆☆）

在你的Qdrant实例上进行性能基准测试： 1. 生成100万个随机768维向量 2. 分别测试以下配置的搜索延迟（P50、P95、P99）： - 无量化 vs int8量化 - hnsw_ef=64 vs hnsw_ef=128 vs hnsw_ef=256 - 单机 vs 2分片分布式 3. 绘制延迟-召回率曲线图 4. 记录每种配置的内存占用

练习3：实现CDC同步管道（★★★★☆）

搭建一个MySQL → Neo4j的CDC同步管道： 1. 在MySQL中创建`users`和`departments`表，插入测试数据 2. 使用python-mysql-replication监听binlog 3. 实现增删改三种事件的同步处理 4. 插入1000条记录，验证全部同步到Neo4j 5. 测试删除和更新操作的同步正确性 6. 测量同步延迟（从MySQL写入到Neo4j可见的时间差）

练习4：构建Grafana监控仪表盘（★★★☆☆）

为知识图谱系统搭建完整的监控体系： 1. 启动Prometheus + Grafana（Docker Compose） 2. 配置Prometheus采集Neo4j和自定义应用指标 3. 在Grafana中创建仪表盘，包含以下面板： - Neo4j QPS（按读/写分类） - 查询延迟分布（P50/P95/P99） - 图谱规模（节点数/关系数） - 连接池使用率 4. 配置至少3条告警规则，使用邮件或Webhook通知 5. 模拟一次查询延迟飙升，验证告警触发

延伸阅读

官方文档

1. Neo4j Operations Manual——集群部署、备份、监控的权威指南
2. Qdrant Distributed Deployment——Qdrant分布式架构官方文档
3. Prometheus + Grafana Getting Started——监控系统入门

进阶书籍

1. 《Designing Data-Intensive Applications》（Martin Kleppmann）——分布式系统设计的经典
2. 《Database Internals》（Alex Petrov）——深入理解存储引擎和分布式数据库
3. 《Site Reliability Engineering》（Google SRE团队）——生产运维的方法论

开源工具

- Debezium：CDC框架，支持MySQL、PostgreSQL、MongoDB等
- Maxwell：轻量级MySQL binlog读取器
- Neo4j Operations Manager：Neo4j集群管理工具
- Grafana Neo4j Plugin：Neo4j数据源插件

本章小结

本章我们完成了从开发到生产的跨越：

1. **Neo4j集群**：通过Causal Clustering实现高可用和读写分离，Core节点保证数据一致性，Read Replica分担读负载
2. **Qdrant分布式**：通过分片和副本实现水平扩展，量化压缩和HNSW调优是性能关键
3. **数据同步**：三种模式各有适用场景——定时全量适合小数据集，CDC适合数据库增量同步，事件驱动适合微服务架构
4. **监控告警**：Prometheus + Grafana构建四层监控体系（业务、应用、基础设施、数据质量），告警规则覆盖关键指标
5. **安全检查清单**：认证、授权、网络隔离、备份恢复，每一环都不能少
6. **上线Checklist**：从开发验收到性能测试到安全加固到部署上线，四步走确保万无一失

知识图谱系统的生产部署不是终点，而是运维的起点。接下来，让我们把视野投向更远的未来——探讨知识图谱在AI Agent、多模态和自主知识系统中的发展方向。

补充深入：生产环境的性能调优实战

Neo4j查询性能调优的”三板斧”

生产环境中，Neo4j的性能问题往往不是”机器不够好”，而是”查询写得烂”。以下是我在生产调优中最常用的三板斧。

第一板斧：EXPLAIN一切。 在Cypher查询前面加上`EXPLAIN`关键字，就能看到查询的执行计划——它会告诉你这个查询是否走了索引、是否产生了笛卡尔积、预估要扫描多少行数据。

```
// 查看执行计划
EXPLAIN MATCH (p:Person {name: '张三'})-[:WORKS_AT]->(c:Co
RETURN p, c.name;

// 关键指标：
// - Estimated Rows: 预估返回行数（越小越好）
// - Db Hits: 数据库访问次数（越少越好）
// - Page Cache Hits/Misses: 缓存命中率（命中越多越好）
```

如果你看到`Estimated Rows`是天文数字（比如10亿），几乎可以肯定查询有问题。最常见的原因是：缺失索引、笛卡尔积、或未限制遍历深度。

第二板斧：参数化查询。 永远不要在Cypher中硬编码值。参数化查询不仅更安全（防注入），更重要的是Neo4j可以缓存查询计划，大幅提升重复查询的性能。

```
# 错误示范：每次查询都是"新"查询，无法复用执行计划
session.run("MATCH (p:Person {name: '张三'}) RETURN p")
session.run("MATCH (p:Person {name: '李四'}) RETURN p")

# 正确示范：参数化查询，复用执行计划
session.run("MATCH (p:Person {name: $name}) RETURN p", nam
session.run("MATCH (p:Person {name: $name}) RETURN p", nam
```

第三板斧：批量操作代替循环。 当你需要插入或更新大量数据时，千万不要在循环中逐条执行Cypher。使用 `UNWIND` 把列表展开为批量操作，性能提升通常在10–100倍。

```
// ❌ 慢：逐条插入
// for person in persons:
//     session.run("CREATE (p:Person {name: $name})", name

// ✅ 快：UNWIND批量插入
UNWIND $persons AS person
MERGE (p:Person {id: person.id})
SET p.name = person.name,
    p.department = person.department,
    p.updated_at = datetime()
```

连接池管理：被忽视的性能杀手

很多团队在生产环境中遇到的性能问题，根源不在数据库，而在**连接池配置不当**。

Neo4j的每个查询都需要一个连接。如果连接池太小，高并发时大量请求排队等待连接；如果连接池太大，维护连接的开销反而拖累性能。

经验公式：**连接池大小 = 并发查询数 × 1.2（留20%余量）**。对于典型的Web应用，并发查询数通常在50–200之间，所以连接池设置在60–240比较合理。

```
# 生产级连接池配置
from neo4j import GraphDatabase

driver = GraphDatabase.driver(
    "bolt://neo4j-core1:7687",
    auth=("neo4j", "your_password"),
    # 连接池配置
    max_connection_pool_size=100,         # 最大连接数
    connection_acquisition_timeout=30,    # 获取连接超时（秒）
    max_connection_lifetime=3600,         # 连接最大存活时间（
    connection_timeout=10,                # 连接建立超时（秒）
    # 重试配置
    max_transaction_retry_time=15,        # 事务重试超时（秒）
    # 连接健康检查
    keepalive=True                        # 保持连接活跃
)
```

另一个容易踩的坑：**不要在每次请求时创建新的Driver实例。**Driver的创建涉及连接建立和集群拓扑发现，开销不小。正确的做法是在应用启动时创建一个全局Driver，所有请求共享。

Qdrant的量化策略详解

Qdrant支持两种量化方式：**标量量化（Scalar Quantization）**和**乘积量化（Product Quantization）**。它们的选择直接影响内存占用和搜索精度。

标量量化（int8）：将float32压缩为int8，内存减少4倍。精度损失通常在1–3%，对于大多数应用完全可以接受。这是我们推荐的默认选择。

乘积量化（PQ）：将高维向量分割为多个子空间，在每个子空间内独立量化。内存可以减少8–16倍，但精度损失较大（5–10%），且搜索速度可能变慢（需要额外的重排序步骤）。适合超大规模（10亿+向量）的场景。

一个实用的对比实验：

```
# 量化策略对比实验
import time
import numpy as np

def benchmark_quantization(client, collection_name, query_
    """对比不同量化策略的搜索性能"""

    # 基准测试（无量化）
    start = time.time()
    results_fp32 = []
    for vec in query_vectors:
        hits = client.search(collection_name, vec, limit=k
                             search_params=SearchParams(exa
        results_fp32.append([h.id for h in hits])
    fp32_time = time.time() - start

    # int8量化测试
    start = time.time()
    results_int8 = []
    for vec in query_vectors:
        hits = client.search(collection_name, vec, limit=k
        results_int8.append([h.id for h in hits])
    int8_time = time.time() - start

    # 计算精度（Recall@K）
    recalls = []
    for fp32_ids, int8_ids in zip(results_fp32, results_in
        overlap = len(set(fp32_ids) & set(int8_ids))
        recalls.append(overlap / k)
    avg_recall = np.mean(recalls)

    print(f"无量化: {fp32_time:.2f}秒")
    print(f"int8量化: {int8_time:.2f}秒 (速度提升: {fp32_tim
    print(f"平均Recall@{k}: {avg_recall:.4f}")
```

```
    return {
        "fp32_time": fp32_time,
        "int8_time": int8_time,
        "speedup": fp32_time / int8_time,
        "recall": avg_recall
    }
```

故障演练：当Neo4j Leader宕机时

生产环境中最怕的不是性能问题，而是**故障时的手足无措**。所以，定期做故障演练是必须的。

以下是一次Neo4j Leader宕机演练的标准流程：

```
# 1. 确认当前集群状态
cypher-shell -u neo4j -p your_password \
  "CALL dbms.cluster.overview() YIELD id, name, databases"

# 2. 找到当前Leader
# 在结果中找到 databases 字段中 neo4j: LEADER 的节点

# 3. 模拟Leader宕机（停止容器）
docker stop neo4j-core1

# 4. 观察集群行为（等待10-30秒）
# - 剩余Core节点会触发Leader重选
# - 应用层会短暂收到连接错误
# - Neo4j驱动会自动发现新拓扑并重连

# 5. 验证新Leader已选举
cypher-shell -a bolt://neo4j-core2:7687 -u neo4j -p your_p
  "CALL dbms.cluster.overview()"

# 6. 验证读写功能恢复
cypher-shell -a bolt://neo4j-core2:7687 -u neo4j -p your_p
  "CREATE (:TestNode {created_at: datetime()}) RETURN 'wri

# 7. 恢复宕机节点
docker start neo4j-core1

# 8. 验证节点重新加入集群
# 等待30秒后检查
cypher-shell -u neo4j -p your_password \
  "CALL dbms.cluster.overview()"
```

演练要点： - **记录每一步的时间戳**：从Leader宕机到新Leader选举完成需要多少秒？这个时间就是你的RTO（恢复时间目标）。 - **观察应用层行为**：应用是否正确处理了短暂的连接中断？是否有重试机制？用户是否感知到了错误？ - **验证数据一致性**：恢复后的节点是否自动同步了宕机期间的数据？

建议每季度做一次这样的演练。第一次你会手忙脚乱，第三次你就能一边喝咖啡一边看着系统自动恢复了。

安全加固的纵深防御

生产环境的安全不是加一把锁就够了，而是需要**纵深防御（Defense in Depth）**——多层防护，任何一层被突破都不会导致全面失守。

对于知识图谱系统，纵深防御包括五层：

第一层：网络隔离。 Neo4j和Qdrant部署在内网（VPC），不暴露任何公网端口。所有外部访问必须通过API Gateway。这是最基础也是最重要的一层——大部分安全事件都是因为数据库直接暴露在公网上。

第二层：传输加密。 所有外部通信使用TLS 1.3。内部服务间通信使用mTLS（双向TLS），确保不仅数据加密，而且通信双方互相验证身份。

第三层：认证与授权。 Neo4j的RBAC（基于角色的访问控制）为不同服务分配最小权限。RAG服务只需要读权限，数据同步服务需要写权限，管理后台需要管理权限——绝不能用同一个高权限账

户做所有事。

第四层：数据加密。 静态数据加密（Encryption at Rest）保护磁盘上的数据。即使有人物理接触了服务器硬盘，也无法读取数据。Neo4j企业版和Qdrant都支持透明数据加密（TDE）。

第五层：审计日志。 所有数据访问行为都记录审计日志——谁在什么时间查询了什么数据、修改了什么关系。这不仅是安全需要，也是合规要求（尤其是金融和医疗行业）。

```
# audit_logger.py
"""
知识图谱访问审计日志
"""
import logging
from datetime import datetime
from functools import wraps

# 配置审计日志
audit_logger = logging.getLogger("kg_audit")
handler = logging.FileHandler("/var/log/kg/audit.log")
handler.setFormatter(logging.Formatter(
    "%(asctime)s | %(message)s"
))
audit_logger.addHandler(handler)
audit_logger.setLevel(logging.INFO)

def audit_log(operation_type):
    """审计日志装饰器"""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # 记录操作开始
            user = kwargs.get("user", "system")
            query = kwargs.get("query", str(args))

            audit_logger.info(
                f"START | user={user} | op={operation_type
                f"query={query[:200]}"
            )

            start_time = datetime.now()
            try:
                result = func(*args, **kwargs)
```

```
                elapsed = (datetime.now() - start_time).to

                audit_logger.info(
                    f"SUCCESS | user={user} | op={operatio
                    f"elapsed={elapsed:.3f}s | "
                    f"result_count={len(result) if isinsta
                )
                return result
            except Exception as e:
                elapsed = (datetime.now() - start_time).to
                audit_logger.error(
                    f"ERROR | user={user} | op={operation_
                    f"elapsed={elapsed:.3f}s | error={str(
                )
                raise
        return wrapper
    return decorator
```

这五层防御就像洋葱——每一层都不复杂，但叠加起来就让攻击者"泪流满面"。很多团队在第一层（网络隔离）和第二层（传输加密）就停了，这是非常危险的。一个被攻破的API密钥就能穿透前两层，直达你的数据库。

补充深入：生产环境的运维自动化

自动化备份与恢复验证

备份不难，难的是**验证备份可以恢复**。很多团队做了半年备份，直到需要恢复时才发现备份文件损坏了、或者恢复流程跑不通。这种"薛定谔的备份"在生产环境中是定时炸弹。

以下是一个自动化的备份验证脚本，建议每周至少执行一次：

```
# backup_verification.py
"""
自动化备份验证：确保备份文件真的可以恢复
"""
import subprocess
import os
from datetime import datetime, timedelta

class BackupVerifier:
    """备份验证器"""

    def __init__(self, backup_dir, verify_dir):
        self.backup_dir = backup_dir
        self.verify_dir = verify_dir  # 独立的验证环境

    def verify_neo4j_backup(self, backup_file: str) -> dic
        """验证Neo4j备份的完整性"""
        result = {
            "file": backup_file,
            "timestamp": datetime.now().isoformat(),
            "steps": []
        }

        # 步骤1：检查备份文件大小
        file_size = os.path.getsize(backup_file)
        result["steps"].append({
            "step": "检查文件大小",
            "status": "pass" if file_size > 1024 * 1024 el
            "detail": f"{file_size / 1024 / 1024:.1f} MB"
        })

        # 步骤2：恢复到验证环境
        try:
            subprocess.run([
```

```
                "neo4j-admin", "database", "restore",
                f"--from-path={backup_file}",
                f"--to-path={self.verify_dir}",
                "--overwrite-destination"
            ], check=True, capture_output=True, timeout=60
            result["steps"].append({
                "step": "恢复到验证环境",
                "status": "pass",
                "detail": "恢复成功"
            })
        except subprocess.CalledProcessError as e:
            result["steps"].append({
                "step": "恢复到验证环境",
                "status": "fail",
                "detail": e.stderr.decode()
            })
            return result

        # 步骤3：启动验证实例并执行一致性检查
        try:
            # 在验证环境启动Neo4j并运行一致性检查
            check_result = subprocess.run([
                "neo4j-admin", "database", "check-consiste
                f"--database=neo4j",
                f"--from-path={self.verify_dir}"
            ], capture_output=True, timeout=300)

            is_consistent = check_result.returncode == 0
            result["steps"].append({
                "step": "一致性检查",
                "status": "pass" if is_consistent else "fa
                "detail": check_result.stdout.decode()[:50
            })
        except Exception as e:
            result["steps"].append({
```

```
            "step": "一致性检查",
            "status": "fail",
            "detail": str(e)
        })

    # 步骤4：验证关键数据存在
    try:
        from neo4j import GraphDatabase
        verify_driver = GraphDatabase.driver(
            f"bolt://localhost:7688",  # 验证环境端口
            auth=("neo4j", "verify_password")
        )
        with verify_driver.session() as session:
            count = session.run(
                "MATCH (n) RETURN count(n) AS cnt"
            ).single()["cnt"]
            result["steps"].append({
                "step": "数据完整性",
                "status": "pass" if count > 1000 else
                "detail": f"节点数: {count}"
            })
        verify_driver.close()
    except Exception as e:
        result["steps"].append({
            "step": "数据完整性",
            "status": "fail",
            "detail": str(e)
        })

    # 总结
    statuses = [s["status"] for s in result["steps"]]
    result["overall"] = "pass" if all(s == "pass" for
        else "fail"

    return result
```

```
    def scheduled_verification(self):
        """定时备份验证（建议每周执行一次）"""
        # 找到最新的备份文件
        backups = sorted(
            [f for f in os.listdir(self.backup_dir) if f.e
            reverse=True
        )

        if not backups:
            print("警告：未找到任何备份文件！")
            return

        latest_backup = os.path.join(self.backup_dir, back
        print(f"验证最新备份：{latest_backup}")

        result = self.verify_neo4j_backup(latest_backup)

        print(f"\n验证结果：{result['overall'].upper()}")
        for step in result["steps"]:
            icon = "✓" if step["status"] == "pass" else "✗
            print(f"  {icon} {step['step']}: {step['detail

        if result["overall"] == "fail":
            # 发送告警
            self.send_alert(result)
```

容量规划：什么时候该扩容

生产运维中另一个重要工作是**容量规划**——提前预判资源瓶颈，而不是等到系统崩了才手忙脚乱。

对于Neo4j，关键的容量指标包括：

堆内存使用率：当JVM堆内存使用率持续超过80%，说明你的图谱规模正在逼近当前内存的承载极限。经验法则是，堆内存应该至少是图谱store文件大小的1.5倍。

页缓存命中率：Neo4j使用页缓存（Page Cache）加速磁盘数据访问。当页缓存命中率低于90%，意味着频繁的磁盘IO，查询性能会显著下降。此时应该增加页缓存大小或添加Read Replica分担负载。

查询队列深度：如果Neo4j的并发查询数持续接近连接池上限，查询请求开始排队等待，用户会感受到明显的延迟增加。

对于Qdrant，关键指标是：

内存使用率：当Qdrant的内存使用超过物理内存的80%，系统可能开始使用swap，搜索延迟会急剧上升。此时应该考虑：增加节点数量、启用量化压缩、或将旧的Collection归档到冷存储。

磁盘使用率：Qdrant的快照和WAL日志会占用磁盘空间。当磁盘使用超过85%，应该清理旧快照或扩容磁盘。

一个简单的容量预警脚本：

```
# capacity_planning.py
"""
容量规划预警系统
"""

class CapacityMonitor:
    def __init__(self, neo4j_client, qdrant_client):
        self.neo4j = neo4j_client
        self.qdrant = qdrant_client
        self.alerts = []

    def check_neo4j_capacity(self):
        """检查Neo4j容量"""
        # 获取存储大小
        store_info = self.neo4j.read("""
            CALL dbms.database.info()
            YIELD key, value
            WHERE key IN ['storeSize', 'idAllocation']
            RETURN key, value
        """)

        # 获取JVM堆内存使用
        jvm_info = self.neo4j.read("""
            CALL dbms.queryJmx(
                'org.neo4j:instance=*,name=Page cache'
            ) YIELD attributes
            RETURN attributes
        """)

        # 检查页缓存命中率
        cache_stats = self.neo4j.read("""
            CALL dbms.queryJmx(
                'org.neo4j:instance=*,name=Page cache'
            ) YIELD attributes
```

```
            RETURN
                attributes['HitRatio']['value'] AS hit_rat
                attributes['UsedBytes']['value'] AS used_b
                attributes['TotalBytes']['value'] AS total
        """)

        if cache_stats:
            hit_ratio = cache_stats[0]["hit_ratio"]
            if hit_ratio < 0.90:
                self.alerts.append({
                    "severity": "warning",
                    "component": "neo4j",
                    "metric": "page_cache_hit_ratio",
                    "value": hit_ratio,
                    "threshold": 0.90,
                    "recommendation": "增加页缓存大小或添加Re
                })

    def check_qdrant_capacity(self):
        """检查Qdrant容量"""
        collections = self.qdrant.list_collections()

        for col_info in collections:
            info = self.qdrant.get_collection(col_info.nam

            # 向量数增长趋势
            point_count = info.points_count
            vector_size = info.config.params.vectors.size

            # 估算内存占用（简化公式）
            estimated_memory_gb = (
                point_count * vector_size * 4  # float32 =
                / (1024 ** 3)
            )
```

```
            if estimated_memory_gb > 50:  # 单个Collection
                self.alerts.append({
                    "severity": "warning",
                    "component": "qdrant",
                    "collection": col_info.name,
                    "metric": "estimated_memory_gb",
                    "value": estimated_memory_gb,
                    "recommendation": "考虑启用量化压缩或增加
                })

    def generate_report(self) -> str:
        """生成容量报告"""
        self.check_neo4j_capacity()
        self.check_qdrant_capacity()

        if not self.alerts:
            return "所有组件容量正常，无需扩容。"

        report = "容量预警报告\n" + "=" * 40 + "\n"
        for alert in self.alerts:
            report += (
                f"[{alert['severity'].upper()}] {alert['co
                f"  指标: {alert['metric']} = {alert['valu
                f"  建议: {alert['recommendation']}\n\n"
            )

        return report
```

灾难恢复演练手册

生产环境中，灾难恢复（Disaster Recovery, DR）不是”有了备份就行”，而是一套完整的流程。以下是一份精简版的灾难恢复演练手册：

场景一：单节点故障（Neo4j Core节点宕机） – 预期：集群自动选举新Leader，读写服务在30秒内恢复 – 操作：停止一个Core节点容器 – 验证：写入测试数据，确认成功 – 恢复：重启容器，确认数据同步完成 – 验收标准：RTO < 1分钟，零数据丢失

场景二：数据库损坏（误删数据） – 预期：从最近一次备份恢复 – 操作：在Neo4j中执行错误的DETACH DELETE – 验证：确认数据丢失 – 恢复：停止服务→从备份恢复→重启服务 – 验收标准：RTO < 30分钟，RPO < 上次备份时间点

场景三：整个数据中心不可用 – 预期：切换到备用数据中心 – 操作：模拟主数据中心网络中断 – 验证：备用数据中心的Neo4j和Qdrant可访问 – 恢复：DNS切换到备用数据中心 – 验收标准：RTO < 1小时，RPO < 5分钟（取决于同步延迟）

每季度至少演练场景一和场景二。场景三每年演练一次即可。演练的关键不是”能不能恢复”，而是”团队是否知道怎么做”——确保每个值班工程师都能独立完成恢复操作，而不是只有架构师才会。

运维是一门”无聊”的学问——最好的运维就是什么都不会发生。但要做到”什么都不会发生”，背后需要大量的准备、演练和自动化。正如一句运维界的名言所说：**“希望是最好的计划，准备是最好的希望。”**

补充深入：可观测性与故障排查实战

分布式追踪：一个请求的完整旅程

在知识图谱的微服务架构中，一个用户请求可能经过五六个服务——API Gateway、RAG服务、Neo4j、Qdrant、LLM API、缓存层。当用户反馈"系统很慢"时，你怎么知道瓶颈在哪里？

答案是**分布式追踪（Distributed Tracing）**。它为每个请求分配一个唯一的TraceID，记录请求在每个服务中的停留时间。

```
# tracing.py
"""
使用OpenTelemetry为知识图谱系统添加分布式追踪
"""
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProces
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter
from opentelemetry.instrumentation.neo4j import Neo4jInstr
from opentelemetry.instrumentation.requests import Request

# 初始化追踪
provider = TracerProvider()
exporter = OTLPSpanExporter(endpoint="http://jaeger:4317")
provider.add_span_processor(BatchSpanProcessor(exporter))
trace.set_tracer_provider(provider)

tracer = trace.get_tracer("knowledge-graph-service")

# 自动注入Neo4j和HTTP追踪
Neo4jInstrumentor().instrument()
RequestsInstrumentor().instrument()

class TracedRAGService:
    """带分布式追踪的RAG服务"""

    def __init__(self, graph_client, vector_client, llm):
        self.graph = graph_client
        self.vector = vector_client
        self.llm = llm

    def answer(self, question: str) -> str:
        with tracer.start_as_current_span("rag_query") as
            root_span.set_attribute("question.length", len
```

```
            # 1. 实体抽取
            with tracer.start_as_current_span("entity_extr
                entities = self.extract_entities(question)
                root_span.set_attribute("entities.count",

            # 2. 并行检索
            with tracer.start_as_current_span("hybrid_retr
                # 向量检索
                with tracer.start_as_current_span("vector_
                    vector_results = self.vector.search(qu
                    ret_span.set_attribute("vector.results

                # 图谱检索
                with tracer.start_as_current_span("graph_s
                    graph_results = self.graph.search(enti
                    ret_span.set_attribute("graph.results"

            # 3. LLM生成
            with tracer.start_as_current_span("llm_generat
                answer = self.llm.generate(question, conte
                gen_span.set_attribute("llm.tokens_used",

            root_span.set_attribute("answer.length", len(a
            return answer
```

有了分布式追踪，当用户说”查询很慢”时，你打开Jaeger界面，输入TraceID，就能一目了然地看到：

```
[rag_query] ──────────────────────────────────── 3.2s
  ├─ [entity_extraction] ─── 0.4s
  ├─ [hybrid_retrieval] ──────────── 1.8s
  │    ├─ [vector_search] ── 0.3s
  │    └─ [graph_search] ──────── 1.5s  ← 瓶颈在这里!
  └─ [llm_generation] ──── 1.0s
```

在这个例子中，图谱查询占了1.5秒，是主要瓶颈。进一步排查可能发现是某个Cypher查询缺少索引，或者遍历深度设置过大。没有分布式追踪，你可能要花几个小时在日志中大海捞针；有了它，几分钟就能定位问题。

日志管理的"三不要"原则

日志是运维的眼睛，但很多团队把日志搞成了一团浆糊。我总结了三条"不要"原则：

不要记录太多。 很多开发者觉得"多记点总没错"，结果日志量比数据量还大，磁盘空间告急，查找有效信息如同大海捞针。生产环境的日志应该只记录：错误和异常、关键业务事件（用户查询、数据同步、配置变更）、性能指标（查询延迟、缓存命中率）。调试级别的日志只在排查问题时临时开启。

不要非结构化。 "ERROR: something went wrong"这样的日志毫无用处。好的日志应该是结构化的，可以被日志系统解析和聚合：

```
# 结构化日志示例
import structlog

logger = structlog.get_logger()

# 好的日志
logger.info(
    "graph_query_completed",
    query_type="read",
    cypher="MATCH (p:Person {name: $name}) RETURN p",
    duration_ms=45,
    result_count=1,
    user_id="user_123",
    cache_hit=True
)

# 这样你可以轻松在日志系统中做聚合分析：
# - 过去1小时所有graph_query_completed事件的P95延迟是多少？
# - 哪些query_type的error_rate最高？
# - cache_hit的比例是多少？
```

不要忽视日志轮转。 生产环境的日志必须有轮转（rotation）策略——按大小或时间切割日志文件，保留最近N天，自动清理旧日志。否则，一个运行半年的服务可能积累上百GB的日志文件，不仅浪费存储，还可能在磁盘满时导致服务崩溃。

一个真实的故障排查案例

最后分享一个让我印象深刻的生产故障排查案例。

某天凌晨2点，值班工程师收到告警：Neo4j查询延迟从正常的50ms飙升到了5秒。

第一反应：看监控。 Grafana仪表盘显示，延迟飙升是从凌晨1:30开始的。同时，CPU使用率从30%涨到了95%，页缓存命中率从95%掉到了40%。

第二反应：看日志。 Neo4j日志显示大量`Long running query`警告。具体是哪条查询？通过分布式追踪发现，是一条定时执行的Cypher查询：

```
MATCH (a:User)-[:PURCHASED]->(p:Product)<-[:PURCHASED]-(b:
WHERE a.id <> b.id
WITH a, b, count(p) AS common_products
WHERE common_products >= 3
MERGE (a)-[:SIMILAR_INTEREST {score: common_products}]->(b
```

这条查询做了用户购买行为的协同过滤分析——找出购买了3个以上相同产品的用户，并创建"相似兴趣"关系。

第三反应：分析根因。 问题是，这条查询在生产环境中产生了**笛卡尔积**。用户表有50万条记录，查询尝试匹配所有用户对——50万 × 50万 = 2500亿次比较。在测试环境中只有1000个用户，所以从未暴露这个问题。

解决方案： 重写查询，使用更高效的方式：

```
// 优化后：以产品为中间节点，避免笛卡尔积
MATCH (p:Product)<-[:PURCHASED]-(a:User)
WITH p, collect(a) AS buyers
WHERE size(buyers) >= 2
UNWIND buyers AS user_a
UNWIND buyers AS user_b
WHERE id(user_a) < id(user_b)  // 去重：只处理每对用户一次
WITH user_a, user_b, count(p) AS common
WHERE common >= 3
MERGE (user_a)-[:SIMILAR_INTEREST {score: common}]->(user_
```

优化后，这条查询从5秒降到了200ms，CPU使用率恢复正常。

教训： 生产环境的数据规模和测试环境完全不同。任何涉及多表匹配或图遍历的查询，都必须在**接近生产规模的数据**上做性能测试。在1000条数据上跑得很好的查询，在50万条数据上可能就是个定时炸弹。

这个案例完美地诠释了运维界的一条铁律：**测试环境里没问题，不代表生产环境没问题。** 差距往往不在代码逻辑，而在数据规模。

生产运维的心法

写完这一章，我想分享一些关于生产运维的个人体会。

很多人觉得运维是”脏活累活”——部署、监控、备份、扩容，没有算法工程师的光环，也没有架构师的荣耀。但我的经验是：**运维能力是一个工程师最被低估的核心竞争力。**

好的算法可以让系统”跑起来”，好的运维才能让系统”一直跑着”。在生产环境中，”一直跑着”比”跑得快”重要一百倍。用户不会因为你的RAG系统Recall@5从90%提升到了92%而兴奋，但系统宕机5分钟就足以让你的客户投诉电话打爆客服。

运维的心法可以总结为三个词：**预防、自动化、可恢复。**

预防胜于治疗。在问题发生之前发现它——通过监控发现趋势性变化（内存缓慢增长、磁盘逐渐填满），而不是等到系统崩溃才反应。

自动化减少人为失误。凡是重复执行超过三次的操作，都应该写成脚本或流水线。人是最不可靠的环节——凌晨三点被叫醒时，你很难保证手动执行的12步恢复流程不会遗漏某一步。

可恢复是最后的安全网。无论预防措施多完善，故障总会发生。关键是在故障发生时能快速恢复——有备份、有预案、有演练。

做好这三点，你的知识图谱系统就能在生产环境中稳稳地跑下去。

第十五章 图谱的未来：Agent、多模态与自主知识系统

> *难度：★★☆☆☆（概念为主，少量代码演示） 前置要求：完成前14章阅读，对知识图谱全貌有基本认知 本章定位：前瞻性与思考性章节，帮助读者把握行业方向*

引子

1956年，达特茅斯会议上，一群科学家首次提出了”人工智能”这个概念。他们乐观地预测：”一个夏天的努力，就能让机器像人一样思考。”

近七十年过去，AI走过了符号主义、连接主义、统计学习、深度学习……每一次浪潮都带来新的希望，也留下未竟的遗憾。

知识图谱的命运也是如此。2012年Google知识图谱上线时，业界欢呼”语义搜索的时代来了”。2018年，随着BERT等预训练语言模型的崛起，知识图谱被不少人判了”死刑”——“谁还需要手动标注实体和关系？LLM什么都学会了。”

然后2023年，大语言模型爆发了。讽刺的是，正是LLM的”幻觉”问题，把知识图谱从边缘拉回了舞台中央。人们发现：LLM越强大，越需要一个可靠的”事实锚点”来约束它的胡说八道。

但这一次，知识图谱的角色变了。它不再是那个需要人工精心标注的”静态百科全书”，而是正在演变成一个**活的、会呼吸的知识有机体**——能自我更新、自我修正、甚至参与推理和决策。

本章是这本书的最后一章，也是最”虚”的一章。我们不写太多代码，而是站在2025年的时间节点上，眺望知识图谱与AI交汇的未来图景。

知识图谱在AI Agent中的角色

什么是AI Agent

AI Agent（智能体）是2024–2025年AI领域最热的方向之一。与普通LLM”问一句答一句”不同，Agent具有**自主性**——它能感知环境、制定计划、执行行动、反思结果，并不断迭代。

用一个比喻：LLM是一个”参谋”——你问它问题，它给建议；Agent是一个”将军”——你给它目标，它自己想办法完成。

Agent的核心循环可以概括为：

```
感知（Perceive）→ 思考（Think）→ 行动（Act）→ 反思（Reflect）
        ↑                                                    
        └────────────────────────────────────────────────────
```

在这个循环中，知识图谱扮演着三个关键角色：**记忆（Memory）**、**推理（Reasoning）**和**规划（Planning）**的基础设施。

记忆：Agent的长期知识库

Agent的记忆系统通常分为三类：

工作记忆（Working Memory）：当前对话的上下文，受LLM上下文窗口限制。

情景记忆（Episodic Memory）：过去交互的具体经历，用于学习用户偏好和历史模式。

语义记忆（Semantic Memory）：关于世界的通用知识和领域专业知识——这正是知识图谱的用武之地。

```
# agent_memory.py
"""
AI Agent的记忆系统架构（概念性实现）
展示知识图谱如何作为语义记忆支撑Agent
"""
from dataclasses import dataclass
from typing import List, Dict, Optional
from datetime import datetime

@dataclass
class Memory:
    content: str
    memory_type: str      # working / episodic / semantic
    timestamp: datetime
    importance: float     # 0-1, 记忆重要度
    access_count: int = 0
    source: str = ""

class AgentMemorySystem:
    """Agent记忆系统"""

    def __init__(self, graph_client, vector_client):
        self.graph = graph_client        # 知识图谱：语义记忆
        self.vector = vector_client      # 向量数据库：情景记
        self.working_memory = []          # 列表：工作记忆

    def store_episodic(self, event: str, context: str):
        """存储情景记忆（具体事件）"""
        # 向量化后存入向量数据库
        embedding = self.get_embedding(event + context)
        self.vector.upsert(
            collection="episodic_memory",
            point_id=generate_id(),
            vector=embedding,
```

```
            payload={
                "content": event,
                "context": context,
                "timestamp": datetime.now().isoformat(),
                "importance": self.estimate_importance(eve
            }
        )

    def query_semantic(self, question: str) -> List[Dict]:
        """查询语义记忆（知识图谱）"""
        # 从问题中提取实体
        entities = self.extract_entities(question)

        # 在知识图谱中检索相关知识
        results = self.graph.read("""
            MATCH (n)-[r]->(m)
            WHERE n.name IN $entities OR m.name IN $entiti
            RETURN n.name AS subject,
                   type(r) AS relation,
                   m.name AS object,
                   n.description AS context
            LIMIT 20
        """, {"entities": entities})

        return results

    def recall(self, query: str, top_k: int = 10) -> List[
        """
        综合回忆：同时从三种记忆中检索
        """
        recalled = []

        # 1. 工作记忆：直接返回最近的对话
        recalled.extend(self.working_memory[-5:])
```

```
        # 2. 情景记忆：向量相似度检索
        embedding = self.get_embedding(query)
        episodic_results = self.vector.search(
            collection="episodic_memory",
            query_vector=embedding,
            limit=top_k
        )
        for hit in episodic_results:
            recalled.append(Memory(
                content=hit.payload["content"],
                memory_type="episodic",
                timestamp=datetime.fromisoformat(hit.paylo
                importance=hit.payload["importance"]
            ))

        # 3. 语义记忆：知识图谱检索
        semantic_results = self.query_semantic(query)
        for record in semantic_results:
            recalled.append(Memory(
                content=f"{record['subject']} -[{record['r
                memory_type="semantic",
                timestamp=datetime.now(),
                importance=0.8,
                source="knowledge_graph"
            ))

        # 按重要度和相关性排序
        recalled.sort(key=lambda m: m.importance, reverse=
        return recalled[:top_k]

    def forget(self, threshold: float = 0.1):
        """
        记忆遗忘：删除低重要度的旧记忆
        模拟人类大脑的遗忘机制
        """
```

```
        # 情景记忆衰减
        self.vector.delete_by_filter(
            collection="episodic_memory",
            filter={
                "must": [
                    {"key": "importance", "range": {"lt":
                ]
            }
        )
        print(f"遗忘完成：清除了重要度 < {threshold} 的旧记忆")
```

推理：图谱驱动的逻辑链

LLM的推理是”黑箱”的——你不知道它为什么得出某个结论。知识图谱可以为Agent提供**可解释的推理链**。

想象这样一个场景：Agent需要判断”张三是否适合担任项目经理”。

纯LLM推理（不可解释）：

> *“张三适合担任项目经理，因为他有丰富的项目管理经验。”*

图谱增强推理（可解释）：

> *1. 张三 → 担任过 → 3个百万级项目（来源：HR系统）*
> *2. 张三 → 持有 → PMP证书（来源：资质库）*
> *3. 张三 → 管理过 → 15人团队（来源：项目档案）*
> *4. 张三 → 获得 → 2024年最佳项目经理奖（来源：荣誉记录）*

> *5. 结论：张三适合担任项目经理（置信度：0.92）*

```
# agent_reasoning.py
"""
图谱驱动的可解释推理
"""

class GraphReasoningEngine:
    """基于知识图谱的推理引擎"""

    def __init__(self, graph_client, llm):
        self.graph = graph_client
        self.llm = llm

    def reason_with_evidence(self, hypothesis: str) -> Dic
        """
        对一个假设进行证据推理

        示例：hypothesis = "张三适合担任项目经理"
        """
        # 1. 分解假设为可验证的子命题
        sub_claims = self.decompose_hypothesis(hypothesis)
        # → ["张三有项目管理经验", "张三有相关资质", "张三有团队管

        # 2. 对每个子命题在知识图谱中搜索证据
        evidence_chain = []
        for claim in sub_claims:
            evidence = self.search_evidence(claim)
            evidence_chain.append({
                "claim": claim,
                "evidence": evidence,
                "verified": len(evidence) > 0,
                "confidence": self.calculate_confidence(ev
            })

        # 3. 综合评估
```

```
        verified_count = sum(1 for e in evidence_chain if
        total_count = len(evidence_chain)
        overall_confidence = sum(e["confidence"] for e in

        # 4. 生成可解释的推理报告
        reasoning_report = self.generate_report(
            hypothesis=hypothesis,
            evidence_chain=evidence_chain,
            overall_confidence=overall_confidence
        )

        return {
            "hypothesis": hypothesis,
            "conclusion": "支持" if overall_confidence > 0
            "confidence": overall_confidence,
            "evidence_chain": evidence_chain,
            "report": reasoning_report
        }

    def search_evidence(self, claim: str) -> List[Dict]:
        """在知识图谱中搜索支持或反驳某个命题的证据"""
        entities = self.extract_entities(claim)

        # 多跳搜索：从实体出发，沿关系链路寻找相关证据
        results = self.graph.read("""
            MATCH path = (n)-[*1..3]-(m)
            WHERE n.name IN $entities
            WITH path, nodes(path) AS nodes, relationships
            UNWIND nodes AS node
            WITH path, node, rels
            WHERE node.description IS NOT NULL
            RETURN
                [x IN nodes(path) | x.name] AS entity_chai
                [r IN relationships(path) | type(r)] AS re
                node.description AS evidence_text
```

```
            LIMIT 10
        """, {"entities": entities})

        return results

    def multi_hop_inference(self, start_entity: str, quest
        """
        多跳推理：沿着图谱的关系链进行推理

        示例：从"张三"出发，推理"张三可能感兴趣的培训课程"
        路径：张三 → 担任(后端工程师) → 需要技能(Java, 微服务) →
              相关课程(高级Java并发, Spring Cloud实战)
        """
        # 获取多跳路径
        paths = self.graph.read("""
            MATCH path = (start {name: $start})-[*2..4]->(
            WHERE end:Course OR end:Training
            RETURN
                [n IN nodes(path) | n.name] AS entities,
                [r IN relationships(path) | type(r)] AS re
                length(path) AS hops
            ORDER BY hops
            LIMIT 10
        """, {"start": start_entity})

        # 让LLM基于推理路径生成自然语言解释
        prompt = f"""基于以下推理路径回答问题。

起点：{start_entity}
问题：{question}

推理路径：
"""
        for p in paths:
            chain = " → ".join(
```

```
                f"{p['entities'][i]} -[{p['relations'][i]}
                for i in range(len(p['relations']))
            ) + p['entities'][-1]
            prompt += f"- {chain}\n"

        prompt += "\n请用自然语言解释推理过程和结论。"

        return self.llm.invoke(prompt).content
```

规划：图谱辅助的任务分解

Agent执行复杂任务时需要**规划（Planning）**——将大目标分解为可执行的子任务。知识图谱可以提供任务分解的”路线图”。

```
# agent_planning.py
"""
知识图谱辅助的任务规划
"""

class GraphPlanner:
    """利用知识图谱进行任务规划"""

    def __init__(self, graph_client, llm):
        self.graph = graph_client
        self.llm = llm

    def plan(self, goal: str) -> List[Dict]:
        """
        基于知识图谱制定执行计划

        示例：goal = "为新产品搭建知识图谱"
        """
        # 1. 在知识图谱中查找类似任务的执行路径
        similar_plans = self.graph.read("""
            MATCH (g:Goal)-[:REQUIRES]->(step:TaskStep)
            WHERE g.name CONTAINS $goal_keyword
            WITH g, collect(step) AS steps
            MATCH (g)-[:REQUIRES]->(step)-[:DEPENDS_ON*0..
            RETURN g.name AS similar_goal,
                   collect({
                       step: step.name,
                       description: step.description,
                       estimated_hours: step.estimated_hou
                       dependencies: step.dependencies
                   }) AS plan_steps
            LIMIT 5
        """, {"goal_keyword": self.extract_keyword(goal)})
```

```
        # 2. 如果没有找到匹配的计划模板，让LLM生成
        if not similar_plans:
            plan_steps = self.generate_plan(goal)
        else:
            # 基于历史计划模板 + LLM适配当前场景
            plan_steps = self.adapt_plan(goal, similar_pla

        # 3. 构建执行DAG（有向无环图）
        execution_dag = self.build_dag(plan_steps)

        return execution_dag

    def build_dag(self, steps: List[Dict]) -> List[Dict]:
        """构建任务依赖DAG"""
        dag = []
        for i, step in enumerate(steps):
            dag.append({
                "id": i,
                "task": step["name"],
                "description": step.get("description", "")
                "dependencies": step.get("dependencies", [
                "estimated_hours": step.get("estimated_hou
                "status": "pending"
            })

        # 拓扑排序，确定执行顺序
        sorted_dag = self.topological_sort(dag)
        return sorted_dag
```

Agent + KG的实际案例

案例一：智能客服Agent

某银行的智能客服系统，Agent需要处理”我想办理信用卡”这样的请求。

- **感知**：识别用户意图是”申请信用卡”
- **思考**（图谱推理）：
- 查询客户画像 → 年龄28岁，月收入15K，信用评分720
- 沿关系链推理 → 符合金卡申请条件
- 关联产品推荐 → 用户有境外消费记录 → 推荐全币种信用卡
- **行动**：引导用户填写申请表，同时推荐匹配的产品
- **反思**：如果用户拒绝推荐，记录原因，更新知识图谱中的偏好关系

案例二：科研助手Agent

一个帮助科研人员做文献综述的Agent：

- **感知**：用户输入”大模型在药物发现中的应用现状”
- **思考**（图谱规划）：
- 在学术知识图谱中定位”大模型”和”药物发现”两个概念
- 沿引用关系找到桥接两者的关键论文
- 按时间线追踪研究进展
- 识别核心研究者及其合作网络
- **行动**：生成结构化的文献综述，包含时间线、关键论文、研究热点
- **反思**：用户追问某篇论文的细节 → Agent深入检索该论文的引用上下文

多模态知识图谱

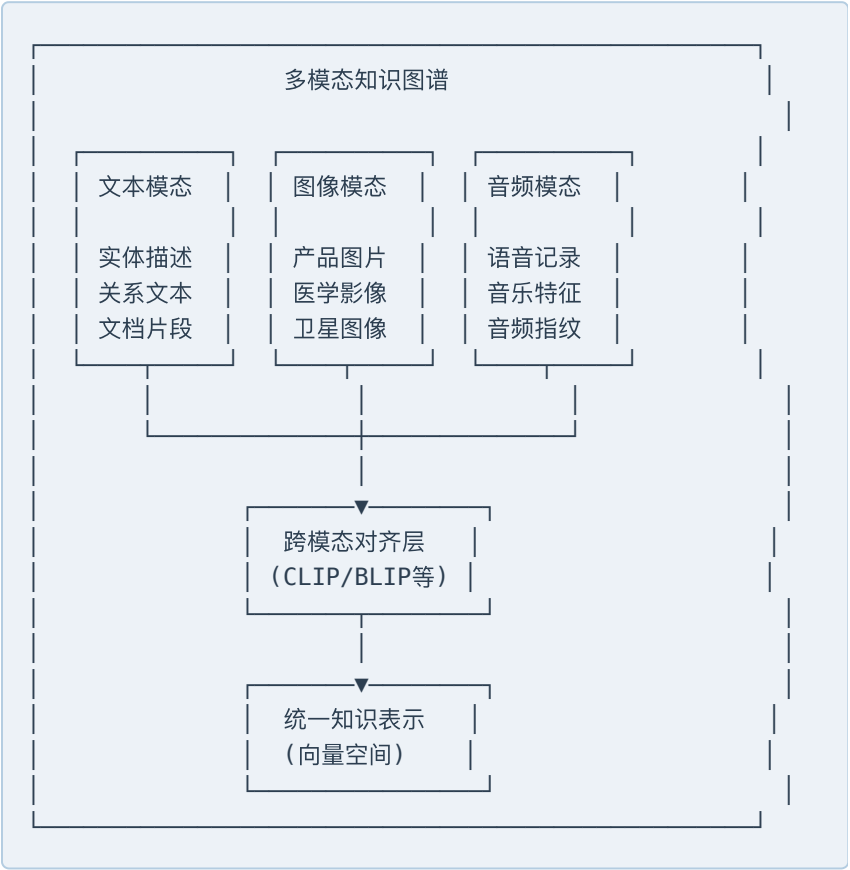
从文本到全模态

传统的知识图谱以文本为主——实体名称、属性值、关系描述都是文字。但真实世界的知识远不止文字：

- 一张产品图片胜过千言描述
- 一段手术视频包含丰富的医学知识
- 一首音乐承载情感和文化信息
- 一个化学分子结构图表达精确的科学知识

多模态知识图谱（Multimodal Knowledge Graph, MMKG）正是为了捕获和关联这些跨模态的知识而生。

MMKG的架构



图像实体对齐

多模态知识图谱的核心挑战之一是**跨模态实体对齐**——识别不同模态中的同一实体。

```
# multimodal_kg.py
"""
多模态知识图谱概念性实现
"""
import clip
import torch
from PIL import Image

class MultimodalKnowledgeGraph:
    """多模态知识图谱"""

    def __init__(self, graph_client, clip_model):
        self.graph = graph_client
        self.clip_model = clip_model  # CLIP模型用于跨模态对

    def add_image_entity(self, image_path: str, entity_nam
                         properties: Dict):
        """
        添加图像实体
        1. 用CLIP提取图像特征向量
        2. 在图谱中创建节点，关联图像向量
        """
        # 提取图像embedding
        image = Image.open(image_path)
        image_embedding = self.clip_model.encode_image(ima

        # 创建图谱节点
        self.graph.write("""
            MERGE (n:Entity {name: $name})
            SET n += $props,
                n.modality = 'image',
                n.image_embedding = $embedding,
                n.image_path = $path
        """, name=entity_name, props=properties,
```

```
            embedding=image_embedding.tolist(),
            path=image_path)

    def cross_modal_search(self, query_text: str, top_k: i
        """
        跨模态检索：用文本搜索图像实体

        示例：query_text = "红色的跑车"
        → 返回图谱中与之最匹配的图像实体
        """
        # 文本编码
        text_embedding = self.clip_model.encode_text(query

        # 在图像向量空间中检索
        image_entities = self.graph.read("""
            MATCH (n:Entity)
            WHERE n.modality = 'image'
              AND n.image_embedding IS NOT NULL
            WITH n,
                 gds.similarity.cosine(n.image_embedding,
            WHERE similarity > 0.5
            RETURN n.name AS name,
                   n.image_path AS image,
                   n.description AS description,
                   similarity
            ORDER BY similarity DESC
            LIMIT $k
        """, query_vec=text_embedding.tolist(), k=top_k)

        return image_entities

    def build_cross_modal_links(self):
        """
        自动构建跨模态关联
        找到文本实体和图像实体之间的语义连接
```

```
        """
        # 找到所有未关联的文本-图像实体对
        candidates = self.graph.read("""
            MATCH (text:Entity {modality: 'text'})
            MATCH (image:Entity {modality: 'image'})
            WHERE NOT (text)-[:SAME_AS]-(image)
              AND text.name = image.name
            RETURN text.name AS entity_name
        """)

        # 对匹配名称的实体创建SAME_AS关系
        for record in candidates:
            self.graph.write("""
                MATCH (text:Entity {name: $name, modality:
                MATCH (image:Entity {name: $name, modality
                MERGE (text)-[:SAME_AS {confidence: 0.95}]
            """, name=record["entity_name"])

        print(f"创建了 {len(candidates)} 条跨模态关联")
```

视频知识抽取

视频是最复杂的多模态数据——它同时包含画面、语音、文字和时间序列信息。从视频中抽取知识是一个前沿研究方向。

```
视频 → 关键帧提取 → 目标检测(人物/物体/场景)
     → 语音识别 → 文本 → 实体/关系抽取
     → OCR → 屏幕文字 → 结构化信息
     → 动作识别 → 事件抽取
     → 时间线构建 → 时序知识图谱
```

一个实际应用场景：从手术视频中自动抽取医学知识。 – 识别手术步骤（切皮→分离→缝合…） – 标注使用的器械 – 记录关键决策点 – 构建手术知识库，用于培训新医生

多模态RAG

将多模态知识图谱与RAG结合，可以实现**多模态问答**——用户用文字提问，系统返回包含图片、图表的富媒体回答。

```
# multimodal_rag.py
"""
多模态RAG：文字提问，图文混合回答
"""

class MultimodalRAG:
    def __init__(self, mmkg, llm, image_generator=None):
        self.mmkg = mmkg        # 多模态知识图谱
        self.llm = llm
        self.image_generator = image_generator  # 可选：图片

    def answer(self, question: str) -> Dict:
        # 1. 文本检索
        text_results = self.text_search(question)

        # 2. 跨模态检索（找到相关图片）
        image_results = self.mmkg.cross_modal_search(quest

        # 3. 构建多模态上下文
        context = self.build_multimodal_context(text_resul

        # 4. LLM生成文本回答（引用图片位置）
        text_answer = self.llm.invoke(f"""
        基于以下信息回答问题。回答中可以引用相关图片，用[图片N]标记

        {context}

        问题：{question}
        """).content

        # 5. 组装图文混合回答
        return {
            "text": text_answer,
            "images": [
```

```
                {"url": r["image"], "caption": r["descript
                for r in image_results
            ],
            "sources": text_results
        }
```

自主知识演化

知识图谱的”生命化”

传统的知识图谱像一个**图书馆**——知识被编目后静静等待查询。未来的知识图谱更像一个**大脑**——它能自我学习、自我更新、自我修正。

我们称之为**自主知识演化（Autonomous Knowledge Evolution）**，包含四个核心能力：

能力一：自动知识更新

```
# auto_update.py
"""
自动知识更新：Agent定期巡检并更新知识图谱
"""

class KnowledgeAutoUpdater:
    """知识自动更新Agent"""

    def __init__(self, graph_client, llm, source_systems):
        self.graph = graph_client
        self.llm = llm
        self.sources = source_systems  # 外部数据源列表

    def scheduled_update(self):
        """定时更新流程"""
        # 1. 扫描各数据源的变更
        changes = self.scan_changes()
        print(f"发现 {len(changes)} 处变更")

        # 2. 分类变更类型
        for change in changes:
            change_type = self.classify_change(change)

            if change_type == "factual_update":
                # 事实性更新：直接修改
                self.apply_factual_update(change)

            elif change_type == "schema_evolution":
                # Schema演化：需要审核
                self.propose_schema_change(change)

            elif change_type == "conflict":
```

```
                # 冲突：需要仲裁
                self.resolve_conflict(change)

    def detect_stale_knowledge(self, threshold_days: int =
        """
        检测过时知识
        找出长时间未更新且可能被验证为错误的知识
        """
        stale_knowledge = self.graph.read("""
            MATCH (n)
            WHERE n.updated_at < datetime() - duration({da
              AND n.verification_count < 2
            WITH n,
                 datetime() - n.updated_at AS age
            RETURN n.name AS entity,
                   n.description AS current_knowledge,
                   age.days AS days_since_update,
                   n.source AS original_source
            ORDER BY age.days DESC
            LIMIT 100
        """, days=threshold_days)

        # 让LLM评估每条知识的时效性
        for item in stale_knowledge:
            freshness = self.evaluate_freshness(item)
            if freshness == "outdated":
                self.flag_for_review(item)

        return stale_knowledge

    def evaluate_freshness(self, knowledge_item: Dict) ->
        """评估知识的时效性"""
        prompt = f"""
        评估以下知识的时效性：
        - 知识内容：{knowledge_item['current_knowledge']}
```

```
        - 距上次更新：{knowledge_item['days_since_update']}天
        - 原始来源：{knowledge_item['original_source']}

        请判断：
        1. 这条知识是否可能已经过时？（stable/possibly_outdated
        2. 建议的操作？（keep/review/update/remove）
        3. 需要验证的要点？
        """
        response = self.llm.invoke(prompt)
        return response.content
```

能力二：冲突解决

当多个来源提供矛盾信息时，知识图谱需要自主检测和解决冲突。

```
# conflict_resolution.py
"""
知识冲突检测与解决
"""

class ConflictResolver:
    """知识冲突解决器"""

    def detect_conflicts(self) -> List[Dict]:
        """检测图谱中的矛盾信息"""
        # 策略1：属性值冲突（同一实体的同一属性有多个不同值）
        conflicts = self.graph.read("""
            MATCH (n)-[r:HAS_ATTRIBUTE]->(a:Attribute)
            WITH n, a.name AS attr_name,
                 collect(DISTINCT a.value) AS values,
                 collect(r.source) AS sources
            WHERE size(values) > 1
            RETURN n.name AS entity,
                   attr_name AS attribute,
                   values AS conflicting_values,
                   sources AS data_sources
        """)

        # 策略2：时序矛盾（事件的先后顺序不一致）
        temporal_conflicts = self.graph.read("""
            MATCH (e1:Event)-[:BEFORE]->(e2:Event)
            MATCH (e2)-[:BEFORE]->(e3:Event)
            MATCH (e3)-[:BEFORE]->(e1)  // 循环依赖 = 矛盾
            RETURN e1.name AS event1, e2.name AS event2, e
        """)

        return conflicts + temporal_conflicts

    def resolve(self, conflict: Dict) -> Dict:
```

```
        """
        解决冲突的策略
        """
        # 策略1：来源可信度加权
        source_credibility = self.get_source_credibility(
            conflict["data_sources"]
        )
        most_credible_source = max(
            source_credibility, key=source_credibility.get
        )

        # 策略2：时间新鲜度优先（最新的数据更可信）
        # 策略3：多数投票（多数来源一致的值更可信）
        # 策略4：LLM仲裁（让大模型综合判断）

        resolution = {
            "conflict": conflict,
            "strategy": "source_credibility",
            "chosen_value": self.get_value_from_source(
                conflict, most_credible_source
            ),
            "chosen_source": most_credible_source,
            "confidence": source_credibility[most_credible
            "rejected_values": [
                v for v in conflict["conflicting_values"]
                if v != self.get_value_from_source(conflic
            ]
        }

        # 应用解决结果
        self.apply_resolution(resolution)

        # 记录解决日志（可追溯）
```

```
        self.log_resolution(resolution)

        return resolution
```

能力三：知识遗忘

人类大脑会自动遗忘不再有用的信息，这反而帮助我们保持思维的敏锐。知识图谱也需要类似的机制——**知识遗忘（Knowledge Forgetting）**。

```
# knowledge_forgetting.py
"""
知识遗忘机制：让图谱保持"精简"
"""

class KnowledgeForgetting:
    """知识遗忘管理器"""

    def calculate_decay(self, node_id: str) -> float:
        """
        计算知识节点的衰减分数

        衰减因子：
        - 时间：距上次访问/更新的时间越长，衰减越大
        - 频率：被查询的次数越少，衰减越大
        - 关联度：与其他节点的连接越少，衰减越大
        - 验证状态：未被验证的知识衰减更快
        """
        node_info = self.graph.read("""
            MATCH (n {id: $id})
            OPTIONAL MATCH (n)-[r]-()
            RETURN
                n.access_count AS access_count,
                n.last_accessed AS last_accessed,
                n.updated_at AS updated_at,
                n.verified AS verified,
                count(r) AS degree
        """, id=node_id)

        info = node_info[0]

        # 时间衰减（指数衰减模型）
        days_since_access = self.days_since(info["last_acc
        time_decay = math.exp(-0.01 * days_since_access)
```

```
        # 频率因子
        frequency_factor = min(info["access_count"] / 100,

        # 关联度因子
        connectivity_factor = min(info["degree"] / 10, 1.0

        # 验证因子
        verification_factor = 1.0 if info["verified"] else

        # 综合衰减分数（0=完全衰减，1=完全保留）
        retention_score = (
            0.3 * time_decay +
            0.3 * frequency_factor +
            0.2 * connectivity_factor +
            0.2 * verification_factor
        )

        return retention_score

    def pruning_cycle(self, threshold: float = 0.2):
        """
        定期剪枝：移除衰减分数低于阈值的孤立知识
        """
        # 获取所有节点
        all_nodes = self.graph.read("MATCH (n) RETURN n.id

        to_archive = []
        to_delete = []

        for node in all_nodes:
            score = self.calculate_decay(node["id"])

            if score < threshold * 0.5:
                # 严重衰减：标记删除
```

```
                to_delete.append(node["id"])
            elif score < threshold:
                # 轻度衰减：归档（移至冷存储）
                to_archive.append(node["id"])

        # 归档：移至历史图谱
        self.archive_to_cold_storage(to_archive)

        # 删除：从活跃图谱中移除
        self.soft_delete(to_delete)  # 软删除，可恢复

        print(f"剪枝完成：归档 {len(to_archive)} 个节点，"
              f"删除 {len(to_delete)} 个节点")
```

能力四：知识进化与概念漂移

知识不是静态的——概念的含义会随时间演变。"手机"在2000年指的是通讯工具，在2025年更多是一个智能终端。知识图谱需要追踪这种**概念漂移（Concept Drift）**。

```
# concept_drift.py
"""
概念漂移检测：追踪知识含义的演变
"""

class ConceptDriftDetector:
    def detect_drift(self, concept: str, time_window: str
        """
        检测某个概念的含义是否随时间发生了变化
        """
        # 获取概念在不同时间点的描述
        temporal_descriptions = self.graph.read("""
            MATCH (c:Concept {name: $concept})-[r:DESCRIBE
            WHERE snapshot.timestamp > datetime() - durati
            RETURN snapshot.description AS description,
                   snapshot.timestamp AS timestamp,
                   snapshot.source AS source
            ORDER BY snapshot.timestamp
        """, concept=concept, window=time_window)

        if len(temporal_descriptions) < 2:
            return {"drift_detected": False, "reason": "数

        # 用LLM比较不同时间点的描述
        comparison = self.llm.invoke(f"""
        分析以下概念在不同时间点的描述，判断其含义是否发生了显著变化

        概念：{concept}

        描述时间线：
        {self.format_temporal_descriptions(temporal_descri

        请回答：
        1. 概念含义是否发生了变化？（是/否/部分变化）
```

```
        2. 变化的性质是什么？（扩展/缩小/转移/分裂）
        3. 变化的关键节点（哪个时间点发生了关键转变）
        4. 建议的更新操作
        """)

        return {
            "concept": concept,
            "drift_detected": True,
            "analysis": comparison.content,
            "temporal_descriptions": temporal_descriptions
        }
```

行业趋势：2025–2030发展预测

近期（2025–2026）：GraphRAG成为标配

预测1：GraphRAG将从前沿研究变为生产标配。

随着微软、LangChain、LlamaIndex等框架的成熟，GraphRAG的实现门槛将大幅降低。到2026年，任何涉及企业知识的AI应用，如果不接入知识图谱做RAG，将被视为”不专业”——就像2020年的应用不做HTTPS一样。

预测2：知识图谱构建成本下降80%。

LLM驱动的自动实体识别和关系抽取将大幅降低图谱构建成本。目前构建一个十万节点的知识图谱可能需要数周人工标注，两年内这个过程将缩短到数天，且人工只需做审核而非标注。

预测3：向量数据库与图数据库开始融合。

Qdrant、Milvus等向量数据库将内置图索引能力；Neo4j等图数据库将原生支持向量搜索。两者的边界逐渐模糊，”Graph + Vector”混合数据库将成为新品类。

中期（2027–2028）：自主知识系统兴起

预测4：企业级”知识操作系统”出现。

类似操作系统管理硬件资源，”知识操作系统”将统一管理企业的文档、数据库、图谱、模型。它将具备： – 自动发现和抽取知识 – 跨部门知识共享和权限管理 – 知识质量自动监控 – 知识驱动的业务决策支持

预测5：个人知识图谱成为新基础设施。

每个人将拥有自己的知识图谱——整合你的阅读记录、对话历史、工作产出、社交关系。它是你数字分身的”大脑”，让你的AI助手真正”懂你”。

预测6：知识图谱+Agent成为企业AI的主流架构。

“LLM + KG + Agent”将取代”LLM + Prompt Engineering”，成为企业AI应用的标准架构。知识图谱不再是可选组件，而是Agent的”操作系统级”基础设施。

远期（2029–2030）：知识的自我进化

预测7：知识图谱实现自我构建和自我修复。

AI系统将能够： – 从任意非结构化数据源自动构建和更新知识图谱 – 自动检测和修复知识冲突 – 预测性地补充可能需要的知识（在用户提问之前就已准备好答案） – 跨语言、跨模态的知识自动对齐

预测8:"知识即服务"成为云计算的新层级。

类似IaaS、PaaS、SaaS,"KaaS（Knowledge as a Service)"将成为云计算的新服务层级。企业不再需要自己构建和维护知识图谱，而是按需订阅行业知识图谱服务。

不确定性：需要警惕的风险

风险1：知识图谱的"大模型替代论"可能卷土重来。

如果LLM的幻觉问题被根本性解决（比如通过更好的训练范式)，知识图谱作为"事实锚点"的价值可能被削弱。

风险2：数据隐私和合规性挑战。

个人知识图谱和企业知识图谱涉及大量敏感数据。如何在利用知识的同时保护隐私，是一个尚未解决的难题。

风险3：知识偏见被系统化放大。

如果知识图谱的自动构建依赖LLM，而LLM本身存在偏见，这些偏见可能被系统性地写入图谱并放大。

职业发展路径

知识图谱工程师（Knowledge Graph Engineer）

核心职责：设计、构建和维护知识图谱系统。

技能树：

```
必备技能
├── 图数据库（Neo4j, JanusGraph, TigerGraph）
├── 知识建模（本体设计、Schema设计）
├── NLP基础（NER、RE、实体对齐）
├── Cypher / SPARQL 查询语言
└── Python编程

进阶技能
├── 分布式图计算（GraphX, DGL）
├── 图神经网络（GNN, GraphSAGE）
├── RAG系统设计与优化
├── 数据工程（ETL, CDC, 流处理）
└── 系统运维（监控、集群管理）
```

薪资参考（2025年国内市场）： - 初级（1-3年）：20-35万/年 - 中级（3-5年）：35-55万/年 - 高级（5年+）：55-80万/年

图谱架构师（Graph Architect）

核心职责：设计知识图谱系统的整体架构，包括数据模型、存储方案、查询优化和系统集成。

与工程师的区别：工程师关注"怎么实现"，架构师关注"怎么设计"。架构师需要理解业务需求，将其翻译为技术方案。

典型成长路径：

```
知识图谱工程师（2–3年）
      ↓
高级知识图谱工程师（2–3年）
      ↓
图谱架构师
      ↓
首席架构师 / 技术VP
```

AI产品经理（知识方向）

核心职责：定义知识驱动的AI产品，协调技术团队和业务需求。

需要的知识图谱理解： – 知识图谱能做什么、不能做什么 – 不同技术方案的优缺点和适用场景 – 数据质量和知识质量对产品体验的影响 – RAG系统的评估指标和优化方向

适合人群：有技术背景但对产品设计和商业更感兴趣的人。

新兴角色：知识运营工程师（Knowledge Ops Engineer）

这是一个正在诞生的新角色——专注于知识图谱的日常运营和维护。

核心职责： – 知识质量监控和治理 – 知识更新和冲突解决 – 用户反馈驱动的知识修正 – 知识使用分析和优化

为什么需要这个角色：随着知识图谱成为企业核心基础设施，”建完图谱”只是开始，”运营图谱”才是持久战。就像DevOps之于软件开发，KnowledgeOps之于知识图谱。

给不同背景读者的建议

如果你是后端工程师：从Neo4j和RAG系统入手，这是你最容易切入的方向。你的系统设计能力和运维经验是巨大优势。

如果你是数据工程师：从知识抽取和数据管道入手。你已经熟悉ETL，而知识图谱的构建本质上就是一个”智能ETL”过程。

如果你是算法工程师：从图神经网络和RAG优化入手。你的机器学习基础让你能深入理解Embedding、GNN等核心技术。

如果你是产品经理：先理解知识图谱的能力边界（能做什么、不能做什么），然后找一个具体的业务场景做MVP。不要试图一上来就做一个”全知全能的系统”。

如果你是学生：打好基础——图论、数据库、NLP、机器学习。同时多做项目，从一个小规模的知识图谱系统开始，积累端到端的经验。

动手练习

练习1：设计一个Agent记忆系统（★★☆☆☆）

为你日常使用的某个场景（如个人笔记助手、工作助手等）设计一个Agent记忆系统： 1. 定义工作记忆、情景记忆、语义记忆分别存储什么内容 2. 设计知识图谱的Schema（节点类型、关系类型） 3. 实现一个简单的回忆（Recall）函数，能同时从三种记忆中检索 4. 思考：什么信息应该被”遗忘”？设计遗忘规则

练习2：构建一个多模态知识节点（★★★☆☆）

选一个你感兴趣的实体（如一部电影、一个景点、一款产品），构建包含多种模态的知识： 1. 文本描述（基本信息、详细评价） 2. 图片（海报、照片、截图） 3. 用CLIP模型提取图片的向量表示 4. 在Neo4j中创建节点，关联文本和图片向量 5. 实现”用文字搜图片”的跨模态检索

练习3：实现知识冲突检测（★★★☆☆）

在一个已有的知识图谱中，模拟并检测知识冲突： 1. 为同一实体的同一属性手动写入3个不同的值（来自不同”来源”） 2. 编写Cypher查询检测属性冲突 3. 实现一个简单的冲突解决策略（如来源可信度加权） 4. 记录解决结果，保留被拒绝的值作为”历史版本”

练习4：绘制你的职业路线图（★☆☆☆☆）

基于本章讨论的职业发展路径，为自己设计一份3年职业发展规划： 1. 你当前的技能储备是什么？ 2. 你想往哪个方向发展？（工程师/架构师/产品/研究） 3. 未来12个月需要补齐的3项核心技能是什么？ 4. 找到一个可以练手的项目（开源贡献/个人项目/公司内部项目）

延伸阅读

前沿论文

1. **《A Survey on Knowledge Graphs: Representation, Acquisition, and Applications》**（Ji et al., 2022）——知识图谱综述，持续更新
2. **《Large Language Models meet Knowledge Graphs: A Survey》**（Zhu et al., 2024）——LLM+KG交叉领域综述
3. **《Multi-Modal Knowledge Graph Construction and Application: A Survey》**（Li et al., 2024）——多模态知识图谱综述
4. **《A Survey on LLM-based Multi-Agent Systems》**（Guo et al., 2024）——多Agent系统综述

行业报告

1. **Gartner：《Hype Cycle for Artificial Intelligence》**——每年更新，追踪AI技术成熟度
2. **McKinsey：《The State of AI》**——AI行业年度报告
3. **Stanford HAI：《AI Index Report》**——学术视角的AI发展报告

思想性读物

1. **《Life 3.0》**（Max Tegmark）——关于AI与人类未来的思考
2. **《The Master Algorithm》**（Pedro Domingos）——关于通用学习算法的畅想

3. 《Knowledge Graphs》（Aidan Hogan等）——知识图谱的系统性教材（免费在线版）

社区与会议

- **ISWC（International Semantic Web Conference）**：语义网和知识图谱顶级学术会议
- **KDD（Knowledge Discovery and Data Mining）**：数据挖掘顶会，KG相关论文多
- **Neo4j Community**：最活跃的图数据库社区
- **GraphRAG Discord**：微软GraphRAG项目讨论社区

本章小结

作为全书的最后一章，我们把视野从技术细节拉升到行业全景：

1. **知识图谱在AI Agent中的角色**：作为记忆系统（语义记忆）、推理引擎（可解释推理链）和规划工具（任务DAG），知识图谱是Agent不可或缺的”认知基础设施”
2. **多模态知识图谱**：从纯文本扩展到图片、音频、视频等多模态数据。CLIP等跨模态模型是关键技术——它让不同模态的数据能在同一个向量空间中对齐和检索

3. **自主知识演化**：未来的知识图谱将具备自我更新、冲突解决、知识遗忘和概念漂移检测的能力，从”静态图书馆”进化为”活的有机体”
4. **行业趋势**：2025–2026年GraphRAG成为标配；2027–2028年自主知识系统兴起；2029–2030年知识实现自我进化。向量数据库和图数据库的融合将催生新品类
5. **职业发展**：知识图谱工程师、图谱架构师、AI产品经理、知识运营工程师——多条路径可选，关键是找到你的切入点和差异化优势

写到这里，这本书就结束了。

回顾全书，我们从知识图谱的基本概念出发（第一章），一步步学会了建模（第二、三章）、构建（第四至六章）、查询（第七、八章）、与AI集成（第九至十三章），最终走向生产部署（第十四章）和未来展望（本章）。

知识图谱不是银弹。它不会自动解决所有问题，也不会让AI突然变得完美。但它是当前AI生态中**最被低估的基础设施之一**。当大多数人在追逐最新的模型时，那些愿意沉下来构建”知识地基”的团队，往往能在关键时刻展现出惊人的优势。

最后送一句话：**模型会变，但知识的价值永恒。**

祝你在知识图谱的旅程中，找到属于自己的那片星空。

—— 树懒老K，2025年

补充深入：知识图谱与LLM的共生关系

“LLM取代知识图谱”是个伪命题

每隔一段时间，就会有人问我：”大模型这么强，知识图谱是不是要凉了？”

我的回答是：**不但不会凉，反而会变得更重要。** 原因有三：

第一，LLM越强，幻觉越危险。 弱模型的幻觉容易被识别——它胡说八道的时候你会觉得”这AI不太行”。但强模型的幻觉极其逼真——它用流畅的语言、合理的逻辑、甚至编造的引用，让你误以为它说的是对的。这种”高质量的谎言”在企业应用中是致命的。知识图谱作为”事实锚点”的价值，随着LLM的增强反而在增大。

第二，知识的时效性不可替代。 GPT-5、GPT-6乃至未来的GPT-N，无论模型多强大，它的知识永远有一个截止日期。而企业需要的知识是实时的——今天签的合同、昨天上线的功能、上周的人事变动。这些实时知识的最佳载体是知识图谱，而不是需要重新训练的模型。

第三，可解释性是硬需求。 在金融风控、医疗诊断、法律合规等领域，”为什么得出这个结论”比”结论本身”更重要。LLM是黑箱，知识图谱是白箱。当监管机构要求你解释AI的决策过程时，你不可能说”这是大模型告诉我的”——你需要一条清晰的证据链，而知识图谱恰好能提供这个。

知识图谱构建范式的变革

LLM不仅没有取代知识图谱，反而正在**革命性地降低图谱的构建成本**。

传统知识图谱的构建是”人力密集型”的：领域专家定义本体，标注团队标注数据，NLP工程师训练抽取模型，质量团队做校验……一个中等规模的图谱可能需要半年时间和数十人的团队。

LLM时代的构建范式变成了”AI辅助+人工审核”：

```
原始文档（合同、报告、邮件、网页...）
    |
    ▼
LLM实体识别（自动，准确率85-95%）
    |
    ▼
LLM关系抽取（自动，准确率75-90%）
    |
    ▼
LLM实体对齐（自动，准确率80-90%）
    |
    ▼
人工审核（只需处理AI不确定的部分）
    |
    ▼
写入知识图谱
```

这种方式将人力成本降低了70-80%，构建周期从半年缩短到数周。更重要的是，它降低了知识图谱的技术门槛——不再需要专门的NLP工程师，一个懂业务的Python开发者就能完成图谱构建。

一个真实的场景：企业知识大脑

让我用一个真实案例来串联本章的概念。

某中型制造企业（约2000人）面临一个典型问题：**知识流失**。每年约15%的员工离职，带走了大量的隐性知识——客户偏好、工艺参数、故障处理经验。新员工平均需要6个月才能上手，期间生产效率下降约30%。

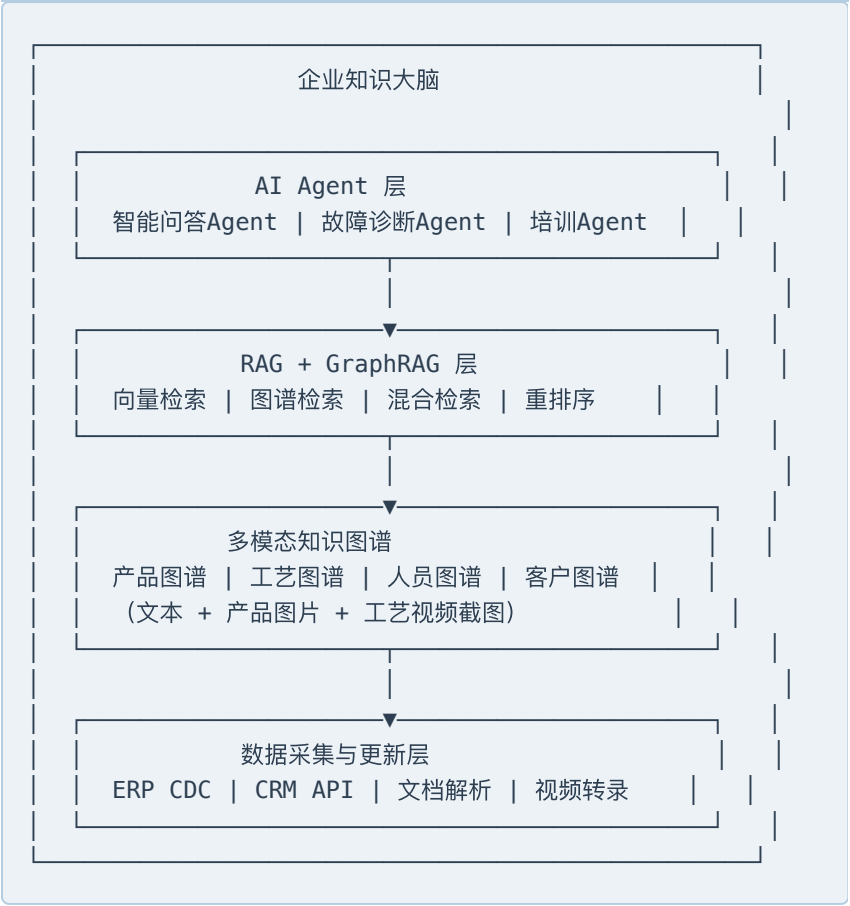
他们决定构建一个”企业知识大脑”，整合以下数据源：

结构化数据：ERP系统的产品BOM、工艺路线；CRM系统的客户信息；HR系统的组织架构。

半结构化数据：设备维护工单、质量检测报告、项目周报。

非结构化数据：技术文档、培训视频（转文字）、老员工的经验总结（口述录音转文字）。

整个系统的架构如下：



上线一年后的关键指标： – 新员工上手时间从6个月缩短到3个月 – 设备故障诊断时间从平均4小时缩短到45分钟 – 客户问题首次解决率从65%提升到88% – 每年节省的知识流失成本估算约800万元

这个案例展示了知识图谱在AI Agent时代的核心价值：**它不是炫技的工具，而是解决实际业务问题的基础设施。**

给初学者的务实建议

如果你读到这里，对知识图谱的未来感到兴奋，我想给你几条务实的建议：

第一，不要追求”完美图谱”。 很多初学者花大量时间设计完美的本体（Ontology），试图一次性建模所有知识。这是错误的。知识图谱应该从一个小而精的核心开始，随着业务发展逐步扩展。先解决一个具体的业务问题，再考虑全局优化。

第二，技术选型要”够用就好”。 Neo4j社区版 + Python + LangChain就能覆盖80%的场景。不要一上来就搞Causal Clustering、分布式部署——那是生产规模的问题，不是学习阶段的问题。

第三，重视数据质量甚于技术架构。 一个数据质量好的小图谱，比一个数据垃圾的大图谱有用得多。花时间做数据清洗、实体规范化、关系验证，这些”脏活”的回报远大于追逐最新的技术框架。

第四，保持对新技术的敏感度，但不要追风。 GraphRAG很火，但不一定适合你的场景。多模态知识图谱很酷，但你可能连单模态都没做好。评估新技术的标准只有一个：**它能否解决你当前遇到的具体问题？**

第五，加入社区。 知识图谱领域发展很快，一个人很难跟上所有进展。加入Neo4j社区、关注GraphRAG的GitHub讨论、参加ISWC等学术会议，和同行交流是最高效的学习方式。

知识图谱这条路，走的人不多，但风景很好。祝你在旅途中找到属于自己的那片星空。

补充深入：知识图谱在垂直行业的深度应用展望

医疗健康：从辅助诊断到个性化治疗

知识图谱在医疗领域的前景可能是所有垂直行业中最令人兴奋的。

当前的医疗知识图谱主要用于**辅助诊断**——将症状、检查结果映射到可能的疾病。但未来的方向是**个性化治疗方案推荐**：基于患者的基因组数据、病史、生活习惯，在知识图谱中构建”患者数字孪生”，然后沿着图谱的关系链推理最适合的治疗方案。

想象这样的场景：一位新确诊的糖尿病患者在知识图谱中不是一个简单的”糖尿病”标签，而是一个丰富的子图——包括他的基因变异位点（影响药物代谢）、并发症历史（肾脏功能轻度受损）、职业特征（久坐办公）、饮食偏好（高碳水）。图谱沿着这些关系推理，不是推荐”标准糖尿病治疗方案”，而是推荐”对这个具体患者最可能有效的个性化方案”。

这种应用的核心挑战不在于技术，而在于**数据隐私**和**知识验证**。医疗知识的错误代价极高，任何图谱中的错误关系都可能导致错误的治疗建议。因此，医疗知识图谱需要比普通图谱高得多的验证标准和审计机制。

金融科技：从反欺诈到智能投研

金融行业是知识图谱落地最快的领域之一，目前已经广泛应用于反欺诈（识别关联交易、异常资金流转）和信用评估（企业关联风险传导）。

下一个爆发点是**智能投研**。传统的投资研究依赖分析师手动阅读大量报告、新闻和财报。一个整合了以下知识的图谱将彻底改变投研流程：

- **公司知识**：组织架构、管理层关系、股权结构、子公司关联
- **行业知识**：产业链上下游关系、竞争格局、政策法规
- **市场知识**：股价走势、交易量、机构持仓变化
- **舆情知识**：新闻报道情感倾向、社交媒体讨论热度
- **另类数据**：卫星图像（停车场车辆数→零售业绩预测）、招聘数据（大规模招聘→业务扩张信号）

投资分析师可以通过图谱进行多维度推理：”某公司CEO最近密集拜访了哪些上下游企业？这些企业的近期财报表现如何？社交媒体上的情绪变化趋势是什么？”——这些跨维度的关联分析，正是知识图谱的独特优势。

智能制造：从故障诊断到供应链韧性

制造业的知识图谱正在从”故障诊断”（设备出问题了怎么修）向”供应链韧性管理”演进。

2020年以来的全球供应链危机让制造企业深刻意识到：**了解自己供应链的全貌有多重要。**一个典型的制造企业可能有5000个供应商，分布在30个国家。当某个地区发生地震、疫情或政治动荡时，哪些供应商受影响？替代供应商在哪里？库存能撑多久？

这些问题的答案散落在ERP系统、采购邮件、合同文档、新闻报道中。知识图谱的价值在于将这些碎片化的信息连接成一张完整的供应链知识网络，让企业在危机发生时能在分钟级别（而非周级别）做出响应。

教育：知识图谱驱动的个性化学习

教育领域可能是知识图谱最能”造福普通人”的应用场景。

传统的教育模式是”一对多”——一个老师面对几十个学生，用同样的教材、同样的进度、同样的考核。知识图谱可以实现真正的”因材施教”：

- **学科知识图谱**：将每个知识点及其前置依赖关系建模为图。学习”微积分”之前需要掌握”函数极限”，而”函数极限”又依赖”数列基础”。
- **学生认知图谱**：追踪每个学生的知识掌握状态。他在”三角函数”上花了3小时，做了20题，正确率60%——说明这个知识点还没掌握，需要加强。
- **自适应推荐**：基于学科图谱和学生认知图谱的交叉推理，为每个学生生成个性化的学习路径。”你不需要从头学微积分，你只需要先补齐函数极限的薄弱环节。”

这种个性化学习系统不是未来的梦想——它正在发生。科大讯飞、好未来等公司已经在部分学校试点类似系统，初步效果显示学生的学习效率提升了30–50%。

写在最后：知识图谱的哲学思考

回到最根本的问题：**知识图谱到底是什么？**

从技术角度看，它是一种数据结构——节点、关系、属性的组合。

从工程角度看，它是一种基础设施——为AI应用提供结构化知识的中间层。

但从哲学角度看，知识图谱是人类认知世界方式的一种数字化映射。

人类理解世界，靠的不是记忆所有事实，而是建立事物之间的联系。你不需要记住”水在100度沸腾”这个孤立的事实——你需要理解的是”气压与沸点的关系”“分子运动与温度的关系”这些更深层的联系。有了这些联系，你不仅能解释水为什么在100度沸腾，还能推理出为什么高压锅能更快地煮熟食物。

知识图谱的本质，就是把这些”联系”数字化、结构化、可计算化。它不是要替代人类的思考，而是要帮助人类（现在也包括AI）更高效地组织和利用知识。

在AI越来越强大的时代，知识图谱的价值不是变小了，而是变大了。因为AI越强大，越需要可靠的知识基础来约束和引导它的行为。正如一辆越快的赛车越需要好的赛道——知识图谱就是AI的”

赛道”。

这本书到这里就结束了。但你的知识图谱之旅，才刚刚开始。

尾声 图谱之外

当你翻到这一页，意味着你已经走完了一段从概念到实战的旅程。从最朴素的”节点—边—节点”三元组，到用 Neo4j 搭建属性图、用 SPARQL 查询 RDF、用 Qdrant 承载向量检索、最终让知识图谱与大语言模型握手——你已经不再是知识图谱的旁观者，而是它的建造者。

但我想说的是：知识图谱不是终点。

过去几年，我们目睹了 AI 领域的剧变。大语言模型以摧枯拉朽之势重塑了人机交互的方式，一度有人宣称”知识图谱已死”。然而事实恰恰相反。当 LLM 的幻觉问题暴露、当企业级应用对准确性和可解释性提出严苛要求时，人们重新把目光投向了知识图谱——不是因为 LLM 不够好，而是因为两者天然互补。LLM 擅长理解语言的模糊与弹性，而知识图谱擅长锚定事实的精确与结构。RAG 架构的流行，只是这场融合的序幕。

更深一层来看，知识图谱所代表的”结构化知识”传统，与深度学习所代表的”统计模式识别”传统，正在走向真正的汇流。前者强调因果、逻辑与可解释性，后者擅长直觉、泛化与创造性。两者的结合不是简单的技术堆叠，而是一种认知范式的升级。当 AI Agent 开始在复杂任务中自主调用工具、查询知识、规划路径时，它背后依赖的，正是你所构建的那张关系网络。

图谱之外，是更广阔的知识工程版图。

回望人类的知识史，从结绳记事到活字印刷，从百科全书到搜索引擎，每一次知识组织方式的革新都深刻改变了文明的进程。知识图谱并非凭空冒出来的新概念——它是人类数千年知识管理传统

的最新一章。当你构建一个知识图谱时，你做的事情本质上与编纂《永乐大典》的解缙、编写《大英百科全书》的编辑、设计 Google 搜索排序算法的工程师并无二致：你都在试图让混乱的知识变得有序，让孤立的洞见彼此连接。这是一项古老而尊贵的工作。

未来的知识系统不会只有一种形态。它可能是动态更新的实时知识图谱，可能是嵌入在向量空间中的隐性知识网络，可能是 Agent 自主维护的内部记忆图，也可能是多模态知识图谱——同时承载文本、图像、视频和传感器数据。随着具身智能和数字孪生的发展，知识图谱甚至可能从数字世界延伸到物理世界，成为机器人理解环境、做出决策的认知基底。图结构本身没有边界，它只是一种思维方式：用关系去理解世界。

而你所掌握的技能——数据建模、知识抽取、图存储与查询、图算法分析、与 AI 模型协作——这些能力不会因为某项技术的更迭而失效。它们沉淀在你的工程直觉里，成为你面对下一个挑战时的底层操作系统。十年前的搜索引擎工程师不会想到今天会有向量数据库，五年前的 NLP 研究者也不会预料到大语言模型的涌现能力。但他们对信息组织方式的深刻理解，始终是穿越技术周期的锚点。

在这个领域，最值得珍惜的不是某一种工具或框架，而是”结构化思考”的习惯。当你面对混乱的信息，本能地想要找到实体、厘清关系、构建层次——你就已经具备了知识工程师的核心素养。工具会换代，语言会更替，但这种思维方式会一直有价值。

或许你会问：既然技术在不断演进，我今天学的东西明天会不会过时？我的回答是：技术会过时，但认知不会。你用 Cypher 写过的每一条查询、用 PageRank 分析过的每一个网络、为 RAG 系统调试过的每一次检索——这些经历已经内化为你的判断力。当新的工具出现时，你会比没有这些经历的人更快地理解它的本质，更准确地评估它的优劣。这就是”学过”与”没学过”的区别——不在于你记住了多少语法，而在于你建立了怎样的认知框架。

我还想提醒一点：不要忽视知识图谱中”知识”这个词的分量。技术的核心从来不是数据库或算法，而是你对业务的理解、对领域的洞察、对问题本质的把握。一个对供应链了如指掌的行业专家，即使只会写最简单的 Cypher，也能构建出比技术高手更有价值的知识图谱。技术是手段，知识才是灵魂。

最后，我想用一个隐喻来结束这本书。知识图谱就像一棵树：根是数据，茎是结构，枝是关系，叶是应用。而让这棵树持续生长的，是不断流入的新知识、新需求、新洞察。没有一棵树是种下去就不管了——它需要你浇水、修剪、施肥，也需要你耐心等待它从幼苗长成大树。

你已经学会了种树。

现在，去种你自己的那片森林吧。

参考文献

以下文献按章节顺序编排，涵盖本书引用的核心论文、书籍、技术文档和在线资源。

第 1 章 什么是知识图谱

1. Singhal, A. (2012). “Introducing the Knowledge Graph: things, not strings.” *Google Blog*.
2. Berners-Lee, T., Hendler, J., & Lassila, O. (2001). “The Semantic Web.” *Scientific American*, 284(5), 34–43.
3. Hogan, A., Blomqvist, E., Cochez, M., et al. (2021). “Knowledge Graphs.” *ACM Computing Surveys*, 54(4), 1–37.
4. Ji, S., Pan, S., Cambria, E., et al. (2022). “A Survey on Knowledge Graphs: Representation, Acquisition, and Applications.” *IEEE Transactions on Neural Networks and Learning Systems*, 33(2), 494–514.
5. 肖仰华 (2020). 《知识图谱：概念与技术》. 电子工业出版社.

第 2 章 知识图谱的核心概念

1. Gruber, T. R. (1993). “A Translation Approach to Portable Ontology Specifications.” *Knowledge Acquisition*, 5(2), 199–220.

2. W3C (2004). “RDF Primer.” W3C Recommendation. https://www.w3.org/TR/rdf-primer/
3. W3C (2014). “RDF 1.1 Concepts and Abstract Syntax.” W3C Recommendation.
4. OWL Working Group (2009). “OWL 2 Web Ontology Language Document Overview.” W3C Recommendation.
5. Angles, R., & Gutierrez, C. (2008). “Survey of Graph Database Models.” *ACM Computing Surveys*, 40(1), 1–39.

第 3 章 知识图谱的生命周期

1. Poggi, A., Lembo, D., Calvanese, D., et al. (2008). “Linking Data to Ontologies.” *Journal on Data Semantics*, X, 133–173.
2. Dong, X. L., & Srivastava, D. (2015). *Big Data Integration*. Morgan & Claypool.
3. 王昊奋, 漆桂林, 潘文月 (2019). 《知识图谱：方法、实践与应用》. 清华大学出版社.

第 4 章 知识抽取

1. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” *NAACL-HLT 2019*.

2. Wei, J., Bosma, M., Zhao, V., et al. (2022). “Finetuned Language Models Are Zero-Shot Learners.” *ICLR 2022*.
3. Zhong, Z., & Ji, K. (2012). “Relation Extraction: Perspective from Convolutional Neural Networks.” *NAACL Workshop*.
4. Han, X., Zhu, H., Yu, P., et al. (2018). “FewRel: A Large-Scale Supervised Few-Shot Relation Classification Dataset.” *EMNLP 2018*.
5. Luan, Y., Wadden, D., He, L., et al. (2019). “A General Framework for Information Extraction.” *AAAI 2019*.

第 5 章 Neo4j 从零开始

1. Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph Databases* (2nd ed.). O’Reilly Media.
2. Neo4j, Inc. “Neo4j Cypher Manual.” https://neo4j.com/docs/cypher-manual/
3. Neo4j, Inc. “Neo4j Operations Manual.” https://neo4j.com/docs/operations-manual/
4. Needham, M., & Hodler, A. E. (2019). *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O’Reilly Media.

第 6 章 Cypher 查询语言

1. Francis, N., Green, A., Guagliardo, P., et al. (2018). “Cypher: An Evolving Query Language for Property Graphs.” *SIGMOD 2018*.
2. Neo4j, Inc. “APOC User Guide.” https://neo4j.com/docs/apoc/
3. Neo4j, Inc. “Graph Data Science Library.” https://neo4j.com/docs/graph-data-science/

第 7 章 RDF 与 SPARQL

1. Harris, S., Seaborne, A., & Prud’hommeaux, E. (2013). “SPARQL 1.1 Query Language.” W3C Recommendation.
2. Apache Software Foundation. “Apache Jena Documentation.” https://jena.apache.org/
3. Arenas, M., Bertails, A., & Prud’hommeaux, E. (2009). “A Direct Mapping of Relational Data to RDF.” W3C Working Draft.
4. Berners-Lee, T. (2006). “Linked Data.” https://www.w3.org/DesignIssues/LinkedData.html

第 8 章 知识图谱推理

1. Nickel, M., Murphy, K., Tresp, V., & Gabrilovich, E. (2016). “A Review of Relational Machine Learning for Knowledge Graphs.” *Proceedings of the IEEE*, 104(1), 11–33.
2. Bordes, A., Usunier, N., Garcia-Duran, A., et al. (2013). “Translating Embeddings for Modeling Multi-relational Data.” *NeurIPS 2013*.
3. Sun, Z., Deng, Z. H., Nie, J. Y., & Tang, J. (2019). “RotateE: Knowledge Graph Embedding by Relational Rotation in Complex Space.” *ICLR 2019*.
4. Kazemi, S. M., & Poole, D. (2018). “Simple Embedding for Link Prediction in Knowledge Graphs.” *NeurIPS 2018*.
5. Toutanova, K., & Chen, D. (2015). “Observed versus Latent Features for Knowledge Base and Text Inference.” *ACL Workshop*.

第 9 章 图算法与图分析

1. Barabási, A. L. (2016). *Network Science*. Cambridge University Press.
2. Blondel, V. D., Guillaume, J. L., Lambiotte, R., & Lefebvre, E. (2008). “Fast Unfolding of Communities in Large Networks.” *Journal of Statistical Mechanics*, 2008(10), P10008.

3. Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). “The PageRank Citation Ranking.” Stanford InfoLab Technical Report.
4. Kipf, T. N., & Welling, M. (2017). “Semi-Supervised Classification with Graph Convolutional Networks.” *ICLR 2017*.
5. Veličković, P., Cucurull, G., Casanova, A., et al. (2018). “Graph Attention Networks.” *ICLR 2018*.

第 10 章 知识图谱与大语言模型

1. Lewis, P., Perez, E., Piktus, A., et al. (2020). “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” *NeurIPS 2020*.
2. Brown, T. B., Mann, B., Ryder, N., et al. (2020). “Language Models are Few-Shot Learners.” *NeurIPS 2020*.
3. Wei, J., Wang, X., Schuurmans, D., et al. (2022). “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.” *NeurIPS 2022*.
4. Gao, Y., Xiong, Y., Gao, X., et al. (2023). “Retrieval-Augmented Generation for Large Language Models: A Survey.” *arXiv preprint arXiv:2312.10997*.
5. Schick, T., Dwivedi-Yu, J., Dessì, R., et al. (2023). “Toolformer: Language Models Can Teach Themselves to Use Tools.” *NeurIPS 2023*.

第 11 章 向量数据库与语义搜索

1. Johnson, J., Douze, M., & Jégou, H. (2019). “Billion-Scale Similarity Search with GPUs.” *IEEE Transactions on Big Data*, 7(3), 535–547.
2. Malkov, Y. A., & Yashunin, D. A. (2020). “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs.” *IEEE TPAMI*, 42(4), 824–836.
3. Qdrant Technologies. “Qdrant Documentation.” https://qdrant.tech/documentation/
4. OpenAI. “Embeddings API Documentation.” https://platform.openai.com/docs/guides/embeddings

第 12 章 RAG 架构与实践

1. Lewis, P., Perez, E., Piktus, A., et al. (2020). [同第 10 章文献 40]
2. LangChain, Inc. “LangChain Documentation.” https://python.langchain.com/docs/
3. LlamaIndex. “LlamaIndex Documentation.” https://docs.llamaindex.ai/
4. Gao, Y., et al. (2023). [同第 10 章文献 43]

5. Sarthi, P., Abdullah, S., Tuli, A., et al. (2024). “Advanced RAG Techniques.” *arXiv preprint*.

第 13 章 行业实战案例

1. Noy, N., Gao, Y., Jain, A., et al. (2019). “Industry-scale Knowledge Graphs: Lessons and Challenges.” *Communications of the ACM*, 62(8), 36–43.
2. Dong, X. L., & Srivastava, D. (2015). [同第 3 章文献 12]
3. Amazon Science (2019). “Building the Amazon Product Knowledge Graph.”
4. Wang, Q., Mao, Z., Wang, B., & Guo, L. (2017). “Knowledge Graph Embedding: A Survey of Approaches and Applications.” *IEEE TKDE*, 29(12), 2724–2743.

第 14 章 部署、运维与性能优化

1. Neo4j, Inc. “Neo4j Performance Tuning.” https://neo4j.com/docs/operations-manual/current/performance/
2. Docker, Inc. “Docker Documentation.” https://docs.docker.com/
3. Kubernetes Authors. “Kubernetes Documentation.” https://kubernetes.io/docs/

4. Prometheus Authors. “Prometheus Documentation.” https://prometheus.io/docs/
5. Grafana Labs. “Grafana Documentation.” https://grafana.com/docs/

第 15 章 前沿趋势与未来展望

1. Zhu, Y., Wang, X., Chen, J., et al. (2023). “LLMs for Knowledge Graph Construction.” *arXiv preprint*.
2. Chen, X., Li, S., Wang, L., et al. (2024). “Multi-Modal Knowledge Graphs: A Survey.” *arXiv preprint*.
3. Wang, L., Ma, C., Feng, X., et al. (2024). “A Survey on Large Language Model based Autonomous Agents.” *Frontiers of Computer Science*.
4. Pan, S., Luo, L., Wang, Y., et al. (2024). “Unifying Large Language Models and Knowledge Graphs: A Roadmap.” *IEEE TKDE*.
5. Yao, Y., Ye, Z., Li, P., et al. (2023). “A Benchmark for Large Language Models on Knowledge Graphs.” *arXiv preprint*.

在线资源

- W3C RDF 规范: https://www.w3.org/RDF/

- W3C SPARQL 规范: https://www.w3.org/TR/sparql11-query/
- Neo4j 官方文档: https://neo4j.com/docs/
- Qdrant 官方文档: https://qdrant.tech/documentation/
- Apache Jena: https://jena.apache.org/
- RDFLib: https://rdflib.readthedocs.io/
- LangChain: https://python.langchain.com/
- LlamaIndex: https://docs.llamaindex.ai/
- OpenKG 开放知识图谱: http://openkg.cn/
- DBpedia: https://www.dbpedia.org/
- Wikidata: https://www.wikidata.org/
- Stanford CS224W 课程: https://web.stanford.edu/class/cs224w/
- Neo4j Graph Academy: https://graphacademy.neo4j.com/

说明：

部分文献在多个章节中被引用，为避免重复，后续引用以”同第 X 章文献 N”标注。所有 URL 截至 2026 年 6 月可访问，若链接失效请通过搜索引擎查找对应资源。

附录

本附录汇集了书中涉及的核心技术速查表，方便读者在日常开发和运维中随时翻阅。每一节都以表格形式呈现，力求”一眼看懂、即用即查”。

A. Ubuntu 常用命令速查表

A.1 文件与目录操作

表 22–1

命令	说明	常用示例
`ls`	列出目录内容	`ls -la`（详细列表含隐藏文件）、`ls -lh`（人类可读的文件大小）
`cd`	切换工作目录	`cd ~`（回到主目录）、`cd -`（回到上一个目录）、`cd /path/to/dir`
`mkdir`	创建目录	`mkdir mydir`、`mkdir -p a/b/c`（递归创建）
`cp`	复制文件或目录	`cp src.txt dst.txt`、`cp -r src_dir/ dst_dir/`（递归复制）
`mv`	移动或重命名	`mv old.txt new.txt`（重命名）、`mv file.txt /target/dir/`
`rm`	删除文件或目录	`rm file.txt`、`rm -rf dir/`（强制递归删除，**慎用**）
`chmod`	修改文件权限	`chmod 755 script.sh`、`chmod +x run.sh`、`chmod 644 config.yml`
`chown`	修改文件所有者	`chown user:group file.txt`、`chown -R user dir/`
`find`	搜索文件	`find / -name "*.log"`、`find . -size +100M`、`find . -mtime -7`

命令	说明	常用示例
`grep`	搜索文件内容	`grep -r "error" /var/log/`、`grep -i "warning" app.log`、`grep -n "TODO" *.py`
`ln`	创建链接	`ln -s /target/path linkname`（软链接）、`ln target linkname`（硬链接）
`tar`	打包与压缩	`tar -czvf archive.tar.gz dir/`（压缩）、`tar -xzvf archive.tar.gz`（解压）
`wc`	统计行/词/字节	`wc -l file.txt`（行数）、`wc -w file.txt`（词数）

A.2 软件包管理

表 22-2

命令	说明	常用示例
`apt update`	更新软件源索引	`sudo apt update`
`apt upgrade`	升级所有已安装的软件包	`sudo apt upgrade -y`
`apt install`	安装软件包	`sudo apt install nginx python3-pip -y`
`apt remove`	卸载软件包（保留配置）	`sudo apt remove nginx`
`apt purge`	卸载软件包（含配置文件）	`sudo apt purge nginx`
`apt autoremove`	清理不再需要的依赖	`sudo apt autoremove -y`
`apt search`	搜索软件包	`apt search neo4j`
`apt show`	查看软件包信息	`apt show python3`
`dpkg -i`	安装本地 .deb 包	`sudo dpkg -i package.deb`
`snap install`	通过 Snap 安装	`sudo snap install code --classic`

命令	说明	常用示例
`snap list`	列出已安装的 Snap 包	`snap list`

A.3 系统管理

表 22-3

命令	说明	常用示例
`systemctl start`	启动服务	`sudo systemctl start neo4j`
`systemctl stop`	停止服务	`sudo systemctl stop neo4j`
`systemctl restart`	重启服务	`sudo systemctl restart nginx`
`systemctl status`	查看服务状态	`systemctl status neo4j`
`systemctl enable`	设置开机自启	`sudo systemctl enable docker`
`systemctl disable`	取消开机自启	`sudo systemctl disable docker`
`journalctl`	查看系统日志	`journalctl -u neo4j --since today`、`journalctl -f`（实时跟踪）
`service`	传统服务管理（兼容）	`sudo service ssh restart`
`top`	实时进程监控	`top`（按 `q` 退出）

命令	说明	常用示例
`htop`	增强版进程监控	`htop`（支持鼠标交互、颜色高亮）
`ps`	查看进程列表	`ps aux`、`ps aux | grep python`
`kill`	终止进程	`kill PID`、`kill -9 PID`（强制终止）
`df`	查看磁盘使用情况	`df -h`
`du`	查看目录大小	`du -sh /var/log/`、`du -h --max-depth=1 /`
`free`	查看内存使用情况	`free -h`
`uname`	查看系统信息	`uname -a`
`uptime`	查看系统运行时间与负载	`uptime`

A.4 网络工具

表 22–4

命令	说明	常用示例
`curl`	HTTP 请求工具	`curl -X GET http://localhost:7474`、`curl -d '{"key":"val"}' -H "Content-Type: application/json" http://api/endpoint`
`wget`	下载文件	`wget https://example.com/file.tar.gz`、`wget -O output.html URL`
`ping`	测试网络连通性	`ping -c 5 google.com`
`ssh`	远程登录	`ssh user@host`、`ssh -p 2222 user@host`、`ssh -i key.pem user@host`
`scp`	远程文件复制	`scp file.txt user@host:/remote/path/`、`scp -r dir/ user@host:/remote/`
`ss`	查看网络端口与连接	`ss -tulpn`（监听端口）、`ss -tnp`（已建立连接）
`ip`	网络接口与路由管理	`ip addr show`、`ip route show`、`ip link set eth0 up`
`ufw`	防火墙管理	`sudo ufw enable`、`sudo ufw allow 22`、`sudo ufw allow 7474/tcp`、`sudo ufw status`

命令	说明	常用示例
`iptables`	底层防火墙规则	`sudo iptables -L -n`、`sudo iptables -A INPUT -p tcp --dport 80 -j ACCEPT`
`netstat`	网络统计（旧版工具）	`netstat -tulpn`（推荐用 `ss` 替代）
`dig`	DNS 查询	`dig example.com`、`dig MX example.com`

A.5 Docker 常用命令

表 22-5

命令	说明	常用示例
`docker run`	创建并启动容器	`docker run -d --name myneo -p 7474:7474 -p 7687:7687 neo4j:latest`
`docker ps`	查看运行中的容器	`docker ps`（运行中）、`docker ps -a`（全部）
`docker stop`	停止容器	`docker stop myneo`
`docker start`	启动已停止的容器	`docker start myneo`
`docker rm`	删除容器	`docker rm myneo`、`docker rm -f myneo`（强制）
`docker images`	列出本地镜像	`docker images`
`docker rmi`	删除镜像	`docker rmi neo4j:latest`
`docker pull`	拉取镜像	`docker pull neo4j:5-community`
`docker logs`	查看容器日志	`docker logs myneo`、`docker logs -f myneo`（实时跟踪）

命令	说明	常用示例
`docker exec`	在容器内执行命令	`docker exec -it myneo bash`、`docker exec myneo cypher-shell`
`docker cp`	容器与主机间复制文件	`docker cp myneo:/data/backup ./backup`
`docker inspect`	查看容器详细信息	`docker inspect myneo`
`docker volume`	数据卷管理	`docker volume create neo4jdata`、`docker volume ls`
`docker network`	网络管理	`docker network create kg-net`、`docker network ls`
`docker compose up`	启动 Compose 服务	`docker compose up -d`
`docker compose down`	停止并移除 Compose 服务	`docker compose down`、`docker compose down -v`（含数据卷）
`docker compose logs`	查看 Compose 日志	`docker compose logs -f neo4j`
`docker system prune`	清理未使用的资源	`docker system prune -a --volumes`

B. Neo4j Cypher 命令速查表

B.1 基础 CRUD 操作

表 22–6

操作	Cypher 语句	说明
创建节点	`CREATE (n:Person {name: '张三', age: 30})`	创建带标签和属性的节点
创建关系	`CREATE (a)-[:KNOWS {since: 2020}]->(b)`	创建带属性的有向关系
匹配节点	`MATCH (n:Person) RETURN n`	查询所有 Person 节点
条件匹配	`MATCH (n:Person) WHERE n.age > 25 RETURN n`	带过滤条件的查询
匹配关系	`MATCH (a:Person)-[r:KNOWS]->(b:Person) RETURN a, r, b`	查询节点及关系
更新属性	`MATCH (n:Person {name: '张三'}) SET n.age = 31 RETURN n`	修改节点属性
添加标签	`MATCH (n:Person {name: '张三'}) SET n:Employee RETURN n`	给节点添加新标签

操作	Cypher 语句	说明
删除属性	`MATCH (n:Person {name: '张三'}) REMOVE n.age RETURN n`	删除节点属性
删除标签	`MATCH (n {name: '张三'}) REMOVE n:Employee RETURN n`	移除节点标签
删除节点	`MATCH (n:Temp) DELETE n`	删除无关系的节点
级联删除	`MATCH (n:Temp) DETACH DELETE n`	删除节点及其所有关系
合并节点	`MERGE (n:Person {id: '001'}) ON CREATE SET n.created = timestamp() ON MATCH SET n.updated = timestamp()`	不存在则创建，存在则更新
合并关系	`MERGE (a:Person {name: '张三'})-[:KNOWS]->(b:Person {name: '李四'})`	合并关系（幂等操作）

B.2 模式匹配

表 22-7

模式	Cypher 语句	说明
可变长度路径	`MATCH p = (a)-[:KNOWS*1..5]->(b) RETURN p`	1 到 5 跳的关系路径
任意长度路径	`MATCH p = (a)-[:KNOWS*]->(b) RETURN p`	不限跳数（注意性能）
最短路径	`MATCH p = shortestPath((a:Person {name:'张三'})-[*]-(b:Person {name:'王五'})) RETURN p`	两个节点间的最短路径
所有最短路径	`MATCH p = allShortestPaths((a)-[*]-(b)) RETURN p`	返回所有最短路径
可选匹配	`OPTIONAL MATCH (n:Person)-[:WORKS_AT]->(c:Company) RETURN n.name, c.name`	左外连接，不匹配返回 null
多模式匹配	`MATCH (n:Person) MATCH (n)-[:KNOWS]->(friend) RETURN n, collect(friend.name)`	多段 MATCH 组合

模式	Cypher 语句	说明
路径函数	`MATCH p = (a)-[*]->(b) RETURN nodes(p), relationships(p), length(p)`	提取路径中的节点、关系和长度
关系方向	`MATCH (a)-[:KNOWS]->(b)` / `MATCH (a)-[:KNOWS]-(b)` / `MATCH (a)<-[:KNOWS]-(b)`	出向 / 无向 / 入向
多关系类型	`MATCH (a)-[:KNOWS|WORKS_WITH]->(b) RETURN a, b`	匹配多种关系类型

B.3 聚合与排序

表 22-8

功能	Cypher 语句	说明
计数	`MATCH (n:Person) RETURN count(n)`	统计节点数量
去重计数	`MATCH (n:Person) RETURN count(DISTINCT n.city)`	统计不同城市的数量
收集为列表	`MATCH (n:Person) RETURN collect(n.name)`	将结果收集为列表
分组聚合	`MATCH (n:Person) RETURN n.city, count(*), avg(n.age)`	按城市分组统计人数和平均年龄
排序	`MATCH (n:Person) RETURN n ORDER BY n.age DESC`	按年龄降序排列
分页 — LIMIT	`MATCH (n:Person) RETURN n ORDER BY n.name LIMIT 10`	取前 10 条
分页 — SKIP	`MATCH (n:Person) RETURN n ORDER BY n.name SKIP 20 LIMIT 10`	跳过前 20 条，取 10 条

功能	Cypher 语句	说明
WITH 管道	`MATCH (n:Person) WITH n, size((n)-[:KNOWS]->()) AS friends WHERE friends > 5 RETURN n, friends`	中间结果传递与过滤
统计函数	`sum()`, `avg()`, `min()`, `max()`, `stDev()`, `percentileDisc()`	常用聚合函数
列表函数	`head()`, `last()`, `tail()`, `range()`, `reduce()`	列表操作函数

B.4 索引与约束

表 22-9

功能	Cypher 语句	说明
创建索引	`CREATE INDEX person_name FOR (n:Person) ON (n.name)`	为 Person.name 创建索引
复合索引	`CREATE INDEX person_name_age FOR (n:Person) ON (n.name, n.age)`	多属性复合索引
全文索引	`CREATE FULLTEXT INDEX personNameDesc FOR (n:Person) ON EACH [n.name, n.description]`	全文索引
查询全文索引	`CALL db.index.fulltext.queryNodes('personNameDesc', '张*') YIELD node, score`	全文检索
删除索引	`DROP INDEX person_name`	删除指定索引
查看所有索引	`SHOW INDEXES`	列出全部索引
唯一约束	`CREATE CONSTRAINT person_id_unique FOR (n:Person) REQUIRE n.id IS UNIQUE`	属性唯一性约束

功能	Cypher 语句	说明
存在约束（企业版）	`CREATE CONSTRAINT person_name_exists FOR (n:Person) REQUIRE n.name IS NOT NULL`	属性必须存在
节点键约束（企业版）	`CREATE CONSTRAINT person_key FOR (n:Person) REQUIRE n.id IS NODE KEY`	节点键（唯一 + 非空）
关系约束	`CREATE CONSTRAINT knows_since FOR ()-[r:KNOWS]-() REQUIRE r.since IS NOT NULL`	关系属性约束
查看约束	`SHOW CONSTRAINTS`	列出全部约束

B.5 APOC 常用函数

表 22-10

函数	说明	示例
`apoc.load.json(url)`	从 URL 加载 JSON 数据	`CALL apoc.load.json('http://api.example.com/data') YIELD value`
`apoc.load.csv(url)`	从 URL/文件加载 CSV	`CALL apoc.load.csv('file:///data.csv') YIELD map`
`apoc.load.jdbc(url, sql)`	从 JDBC 数据源加载数据	`CALL apoc.load.jdbc('jdbc:mysql://host/db', 'SELECT * FROM users')`
`apoc.merge.node(labels, props)`	合并节点（事务安全）	`CALL apoc.merge.node(['Person'], {id: '001'}, {name: '张三'})`
`apoc.merge.relationship(r, type, props)`	合并关系	`CALL apoc.merge.relationship(a, 'KNOWS', {}, {}, b)`
`apoc.create.node(labels, props)`	动态创建节点	`CALL apoc.create.node(['Person', 'Employee'], {name: '李四'})`

函数	说明	示例
`apoc.path.expand(start, rels, labels, min, max)`	路径扩展	`CALL apoc.path.expand(startNode, 'KNOWS>', '+Person', 1, 5)`
`apoc.path.subgraphNodes(start, config)`	子图节点发现	`CALL apoc.path.subgraphNodes(startNode, {maxLevel: 3})`
`apoc.periodic.iterate(cypher, cypher, config)`	批量执行 Cypher	`CALL apoc.periodic.iterate('MATCH (n) RETURN n', 'SET n.processed = true', {batchSize: 1000})`
`apoc.export.csv.all(file)`	导出全图为 CSV	`CALL apoc.export.csv.all('export.csv')`
`apoc.import.csv(nodes, rels, config)`	从 CSV 导入图	`CALL apoc.import.csv([{fileName: 'nodes.csv'}], [{fileName: 'rels.csv'}])`
`apoc.meta.graph()`	可视化图模式（元图）	`CALL apoc.meta.graph()`
`apoc.refactor.rename.label(old, new)`	重命名标签	`CALL apoc.refactor.rename.label('OldLabel', 'NewLabel')`

B.6 GDS 图算法库

表 22-11

算法类别	函数	说明
中心性	`gds.pageRank.stream(graphName)`	PageRank 中心性 — 衡量节点影响力
	`gds.betweennessCentrality.stream(graphName)`	介数中心性 — 衡量节点"桥梁"角色
	`gds.closenessCentrality.stream(graphName)`	接近中心性 — 衡量节点到其他节点的平均距离
	`gds.degreeCentrality.stream(graphName)`	度中心性 — 衡量节点的连接数量
	`gds.eigenvectorCentrality.stream(graphName)`	特征向量中心性
社区发现	`gds.louvain.stream(graphName)`	Louvain 社区检测 — 发现社区结构
	`gds.louvain.write(graphName, {writeProperty: 'community'})`	Louvain 结果写回
	`gds.labelPropagation.stream(graphName)`	标签传播算法

算法类别	函数	说明
	`gds.weaklyConnectedComponents.stream(graphName)`	弱连通分量
	`gds.connectedComponents.stream(graphName)`	强连通分量（有向图）
相似度	`gds.nodeSimilarity.stream(graphName)`	节点相似度（Jaccard / Overlap）
	`gds.knn.stream(graphName)`	K 近邻算法
路径查找	`gds.shortestPath.dijkstra.stream(graphName, config)`	Dijkstra 最短路径
	`gds.allShortestPaths.stream(graphName)`	全源最短路径
	`gds.steinerTree.stream(graphName, config)`	Steiner 树
图投影	`gds.graph.project(graphName, nodeSpec, relSpec)`	创建内存图投影
	`gds.graph.project.cypher(graphName, nodeQuery, relQuery)`	用 Cypher 创建投影
	`gds.graph.drop(graphName)`	删除图投影
	`gds.graph.list()`	列出所有图投影

算法类别	函数	说明
节点嵌入	`gds.fastRP.stream(graphName, {embeddingDimension: 64})`	FastRP 节点嵌入
	`gds.hashGNN.stream(graphName, config)`	HashGNN 嵌入
管道/训练	`gds.pipeline.create(name, config)`	创建机器学习管道
	`gds.beta.graphSage.train(graphName, config)`	GraphSAGE 训练

C. Qdrant API 速查表

C.1 Collection 管理

表 22–12

操作	HTTP 方法 & 路径	请求体示例
创建 Collection	`PUT /collections/{name}`	`{"vectors": {"size": 1536, "distance": "Cosine"}}`
删除 Collection	`DELETE /collections/{name}`	—
查看 Collection 信息	`GET /collections/{name}`	—
列出所有 Collection	`GET /collections`	—
更新 Collection 参数	`PATCH /collections/{name}`	`{"hnsw_config": {"m": 32}}`
创建快照	`POST /collections/{name}/snapshots`	—

操作	HTTP 方法 & 路径	请求体示例
从快照恢复	`PUT /collections/{name}/snapshots/recover`	`{"location": "file:///path/to/snapshot.snapshot"}`

C.2 Point 操作

表 22–13

操作	HTTP 方法 & 路径	请求体示例
插入/更新（Upsert）	`PUT /collections/{name}/points`	`{"points": [{"id": 1, "vector": [0.1, 0.2, ...], "payload": {"city": "北京"}}]}`
批量 Upsert	`PUT /collections/{name}/points`	同上，points 数组可包含大量条目
获取指定 Point	`GET /collections/{name}/points/{id}`	—
批量获取	`POST /collections/{name}/points`	`{"ids": [1, 2, 3]}`
删除 Point	`POST /collections/{name}/points/delete`	`{"points": [1, 2, 3]}` 或 `{"filter": {...}}`
更新 Payload	`POST /collections/{name}/points/payload`	`{"payload": {"city": "上海"}, "points": [1, 2]}`
删除 Payload 字段	`POST /collections/{name}/points/payload/delete`	`{"keys": ["city"], "points": [1]}`

操作	HTTP 方法 & 路径	请求体示例
覆盖 Payload	`PUT /collections/{name}/points/payload`	`{"payload": {"city": "深圳"}, "points": [1]}`

C.3 搜索操作

表 22–14

操作	HTTP 方法 & 路径	请求体示例
向量搜索	`POST /collections/{name}/points/search`	`{"vector": [0.1, 0.2, ...], "limit": 10, "with_payload": true}`
带过滤的搜索	`POST /collections/{name}/points/search`	`{"vector": [...], "limit": 10, "filter": {"must": [{"key": "city", "match": {"value": "北京"}}]}}`
滚动查询	`POST /collections/{name}/points/scroll`	`{"limit": 100, "with_payload": true, "with_vector": false}`
推荐查询	`POST /collections/{name}/points/recommend`	`{"positive": [1, 2], "negative": [3], "limit": 10}`
分组搜索	`POST /collections/{name}/points/search/groups`	`{"vector": [...], "limit": 10, "group_by": "city", "group_size": 3}`
批量搜索	`POST /collections/{name}/points/search/batch`	`{"searches": [{"vector": [...], "limit": 5}, {"vector": [...], "limit": 5}]}`

操作	HTTP 方法 & 路径	请求体示例
计数	`POST /collections/{name}/points/count`	`{"filter": {...}, "exact": true}`

C.4 Filter 语法

表 22-15

条件类型	JSON 结构	说明
精确匹配	`{"key": "city", "match": {"value": "北京"}}`	字段值完全匹配
文本匹配	`{"key": "description", "match": {"text": "知识图谱"}}`	全文匹配
多值匹配	`{"key": "city", "match": {"any": ["北京", "上海"]}}`	匹配多个值之一
排除匹配	`{"key": "status", "match": {"except": ["deleted"]}}`	排除特定值
数值范围	`{"key": "age", "range": {"gte": 18, "lte": 65}}`	数值区间（gte/lte/gt/lt）
地理范围	`{"key": "location", "geo_bounding_box": {"top_left": {...}, "bottom_right": {...}}}`	地理边界框
地理半径	`{"key": "location", "geo_radius": {"center": {"lat": 39.9, "lon": 116.4}, "radius": 10000}}`	指定半径范围（米）
空值检查	`{"is_empty": {"key": "email"}}`	字段是否为空

条件类型	JSON 结构	说明
空值检查（null）	`{"is_null": {"key": "email"}}`	字段是否为 null
must（全部满足）	`{"must": [条件1, 条件2, ...]}`	逻辑 AND
should（满足其一）	`{"should": [条件1, 条件2, ...]}`	逻辑 OR
must_not（全部排除）	`{"must_not": [条件1, 条件2, ...]}`	逻辑 NOT
嵌套过滤	`{"nested": {"key": "comments", "filter": {"must": [...]}}}`	嵌套对象过滤

C.5 批量操作与集群管理

表 22–16

操作	说明
批量 Upsert	在单次 `PUT /collections/{name}/points` 请求中包含多个 points 条目，建议每批 100—1000 条
批量搜索	`POST /collections/{name}/points/search/batch`，一次请求发送多个搜索任务
批量删除	`POST /collections/{name}/points/delete`，使用 filter 或 ID 列表批量删除
集群状态	`GET /cluster` — 查看集群整体状态
节点信息	`GET /cluster/peer` — 查看当前节点信息
集合分片	`GET /collections/{name}/cluster` — 查看集合分片分布
迁移分片	`POST /collections/{name}/cluster` — 在节点间迁移分片
副本管理	`PUT /collections/{name}` 中设置 `replication_factor` 和 `write_consistency_factor`

D. SPARQL 命令速查表

D.1 基础查询模式

表 22–17

模式	SPARQL 语句	说明
SELECT 查询	`SELECT ?name ?age WHERE { ?person rdf:type :Person . ?person :name ?name . ?person :age ?age . }`	基本变量查询
带前缀	`PREFIX foaf: <http://xmlns.com/foaf/0.1/> SELECT ?name WHERE { ?person foaf:name ?name . }`	声明命名空间前缀
DISTINCT 去重	`SELECT DISTINCT ?type WHERE { ?s rdf:type ?type . }`	去重查询
限制结果数	`SELECT ?s WHERE { ?s rdf:type :Person } LIMIT 10 OFFSET 20`	分页查询
排序	`SELECT ?name ?age WHERE { ... } ORDER BY DESC(?age)`	降序排列
ASK 查询	`ASK { :张三 :knows :李四 }`	判断是否存在匹配模式，返回 true/false
CONSTRUCT 查询	`CONSTRUCT { ?person :fullName ?name } WHERE { ?person foaf:name ?name }`	构造新三元组

模式	SPARQL 语句	说明
DESCRIBE 查询	`DESCRIBE :张三`	获取关于资源的所有已知三元组

D.2 FILTER 与 OPTIONAL

表 22-18

功能	SPARQL 语句	说明
字符串过滤	`FILTER(STRSTARTS(?name, "张"))`	名称以”张”开头
正则表达式	`FILTER(REGEX(?name, "^张.*", "i"))`	正则匹配
数值比较	`FILTER(?age >= 18 && ?age <= 65)`	年龄范围
语言标签	`FILTER(LANG(?label) = "zh")`	筛选中文标签
数据类型	`FILTER(DATATYPE(?value) = xsd:integer)`	按数据类型过滤
存在判断	`FILTER EXISTS { ?person :worksAt ?company }`	子模式存在
不存在判断	`FILTER NOT EXISTS { ?person :retired true }`	子模式不存在
IN 操作	`FILTER(?city IN ("北京", "上海", "广州"))`	值在列表中
逻辑运算	`FILTER(?age > 18 && (?city = "北京" || ?city = "上海"))`	复合逻辑

功能	SPARQL 语句	说明
OPTIONAL	`OPTIONAL { ?person :email ?email }`	可选模式，不匹配时变量为 unbound
BIND	`BIND(CONCAT(?firstName, " ", ?lastName) AS ?fullName)`	绑定计算结果
COALESCE	`BIND(COALESCE(?email, "无邮箱") AS ?contact)`	提供默认值
IF 表达式	`BIND(IF(?age >= 18, "成人", "未成年") AS ?category)`	条件表达式

D.3 聚合与分组

表 22–19

功能	SPARQL 语句	说明
COUNT	`SELECT (COUNT(?person) AS ?total) WHERE { ?person rdf:type :Person }`	计数
SUM	`SELECT (SUM(?salary) AS ?totalSalary) WHERE { ... }`	求和
AVG	`SELECT (AVG(?age) AS ?avgAge) WHERE { ... }`	平均值
MIN / MAX	`SELECT (MIN(?age) AS ?youngest) (MAX(?age) AS ?oldest) WHERE { ... }`	最小/最大值
GROUP_CONCAT	`SELECT ?dept (GROUP_CONCAT(?name; SEPARATOR=", ") AS ?members) WHERE { ... } GROUP BY ?dept`	分组连接
GROUP BY	`SELECT ?city (COUNT(*) AS ?pop) WHERE { ... } GROUP BY ?city`	按城市分组
HAVING	`SELECT ?city (COUNT(*) AS ?pop) WHERE { ... } GROUP BY ?city HAVING (COUNT(*) > 100)`	分组后过滤
SAMPLE	`SELECT ?dept (SAMPLE(?name) AS ?example) WHERE { ... } GROUP BY ?dept`	随机取样

功能	SPARQL 语句	说明
GROUP BY 多变量	`SELECT ?dept ?role (COUNT(*) AS ?cnt) WHERE { ... } GROUP BY ?dept ?role`	多维度分组

D.4 CONSTRUCT 与 DESCRIBE

表 22–20

功能	SPARQL 语句	说明
CONSTRUCT 生成新图	`CONSTRUCT { ?person :hasSkill ?skill } WHERE { ?person :skill ?skill . FILTER(?skill != "obsolete") }`	根据查询结果构造新三元组
CONSTRUCT + 模板	`CONSTRUCT { ?person rdfs:label ?label } WHERE { ?person :name ?name . BIND(LCASE(?name) AS ?label) }`	使用 BIND 转换
DESCRIBE 单资源	`DESCRIBE <http://example.org/person/张三>`	获取资源描述
DESCRIBE 查询结果	`DESCRIBE ?person WHERE { ?person :worksAt :CompanyA }`	描述匹配的所有资源
CONSTRUCT 合并数据	`CONSTRUCT { ?s ?p ?o } WHERE { { GRAPH :g1 { ?s ?p ?o } } UNION { GRAPH :g2 { ?s ?p ?o } } }`	跨图合并

D.5 联邦查询（SERVICE）

表 22–21

功能	SPARQL 语句	说明
远程端点查询	`SELECT ?name WHERE { SERVICE <http://dbpedia.org/sparql> { ?s rdfs:label ?name . FILTER(LANG(?name)="zh") } }`	查询远程 SPARQL 端点
联邦 + 本地组合	`SELECT ?localName ?dbpediaLabel WHERE { ?person :name ?localName . SERVICE <http://dbpedia.org/sparql> { ?dbp rdfs:label ?dbpediaLabel . FILTER(?dbpediaLabel = ?localName) } }`	本地与远程数据联合查询
可选联邦查询	`SELECT ?name ?extInfo WHERE { ?person :name ?name . OPTIONAL { SERVICE <http://remote.endpoint/sparql> { ?ext :relatedTo ?name . ?ext :info ?extInfo } } }`	远程查询失败时不影响主查询
命名图	`SELECT ?s ?p ?o WHERE { GRAPH <http://example.org/graph1> { ?s ?p ?o } }`	查询指定命名图
所有命名图	`SELECT ?g ?s ?p ?o WHERE { GRAPH ?g { ?s ?p ?o } }`	遍历所有命名图

E. 推荐学习资源与社区

E.1 书籍

表 22–22

书名	作者	说明
《知识图谱：方法、实践与应用》	王昊奋等	中文知识图谱领域经典，系统介绍方法论
《知识图谱：概念与技术》	肖仰华	学术视角，覆盖知识图谱核心理论
《Knowledge Graphs》	Aidan Hogan 等	全面的学术专著，涵盖 RDF、SPARQL、推理等
《Graph Databases》	Ian Robinson 等	Neo4j 官方推荐，图数据库入门必读
《Network Science》	Albert-László Barabási	网络科学经典，理解图算法的理论基础
《Graph Algorithms》	Mark Needham & Amy E. Hodler	Neo4j GDS 实战指南
《Representation Learning for Dynamic (Knowledge) Graphs》	多篇综述合集	知识图谱嵌入与动态表示学习
《Building Knowledge Graphs》	Jesús Barrasa 等	实战导向，从数据到知识图谱的构建流程

书名	作者	说明
《The Practitioner's Guide to Graph Data》	Denise Gosnell & Matthias Broecheler	图数据实战，涵盖 TigerGraph 等工具

E.2 学术论文与综述

表 22–23

论文/综述	作者/来源	说明
“A Survey on Knowledge Graph Embedding”	Wang et al., 2017	知识图谱嵌入方法综述
“Knowledge Graph Embedding: A Survey of Approaches and Applications”	Cai et al., IEEE TKDE 2020	嵌入方法全面综述
“A Survey on Knowledge Graphs: Representation, Acquisition, and Applications”	Ji et al., IEEE TNNLS 2022	知识图谱全景综述
“Retrieval–Augmented Generation for Knowledge–Intensive NLP Tasks”	Lewis et al., NeurIPS 2020	RAG 原始论文
“Graph Neural Networks: A Review of Methods and Applications”	Zhou et al., AI Open 2020	GNN 方法综述
“Translating Embeddings for Modeling Multi–relational Data”	Bordes et al., NeurIPS 2013	TransE 原始论文
“BERT: Pre–training of Deep Bidirectional Transformers”	Devlin et al., NAACL 2019	BERT 原始论文
“Language Models are Few–Shot Learners”	Brown et al., NeurIPS 2020	GPT–3 论文

E.3 在线课程与教程

表 22–24

资源	平台	说明
Neo4j Graph Academy	neo4j.com/graph-academy	官方免费课程，含认证考试
Stanford CS224W: Machine Learning with Graphs	YouTube / Stanford Online	Jure Leskovec 主讲，图机器学习
Stanford CS520: Knowledge Graphs	Stanford Online	知识图谱专题课程
DeepLearning.ai — Knowledge Graphs	Coursera	Andrew Ng 平台上的 KG 课程
RDF 与 SPARQL 教程	W3C 官方文档	RDF/SPARQL 规范与教程
知识图谱开放课程	浙江大学等	国内高校 KG 公开课
Qdrant 官方文档与教程	qdrant.tech/documentation	向量数据库入门与进阶

E.4 开源项目与数据集

表 22–25

项目/数据集	链接	说明
Wikidata	wikidata.org	全球最大开放知识图谱
DBpedia	dbpedia.org	从 Wikipedia 抽取的结构化知识图谱
CN-DBpedia	cndbpedia.com	中文百科知识图谱
OpenKG	openkg.cn	中文开放知识图谱社区
Freebase（存档）	freebase.com	Google 早期知识图谱（已停止更新）
YAGO	yago-knowledge.org	基于 Wikipedia + WordNet 的知识图谱
FB15k-237 / WN18RR	学术基准数据集	知识图谱嵌入评测标准数据集
Neo4j	neo4j.com	最流行的图数据库
Qdrant	qdrant.tech	高性能开源向量数据库
Apache Jena	jena.apache.org	RDF/SPARQL Java 框架
RDFLib	rdflib.readthedocs.io	Python RDF 处理库
spaCy	spacy.io	NLP 工具，支持实体识别与关系抽取

项目/数据集	链接	说明
DeepKE	github.com/zjunlp/DeepKE	知识抽取工具包（NER / RE / KE）
LlamaIndex	llamaindex.ai	LLM + 外部知识索引框架
LangChain	python.langchain.com	LLM 应用开发框架

E.5 技术社区与会议

表 22–26

社区/会议	说明
ISWC（International Semantic Web Conference）	语义网与知识图谱顶级学术会议
ESWC（Extended Semantic Web Conference）	欧洲语义网会议
K-CAP（Knowledge Capture）	知识获取与建模国际会议
ACL / EMNLP / NAACL	NLP 顶会，大量 KG 相关论文
NeurIPS / ICML / ICLR	机器学习顶会，图学习与表示学习
Neo4j Community Forum	Neo4j 官方社区论坛
中文知识图谱社区（OpenKG）	国内 KG 学术交流与开源协作
Knowledge Graph Conference (KGC)	面向工业界的知识图谱会议
Graph + AI Summit	图技术与 AI 融合的行业峰会
GitHub Topics: knowledge-graph	GitHub 上 KG 相关开源项目聚合
知乎 — 知识图谱话题	中文技术讨论与经验分享

社区/会议	说明
CSDN / 掘金 — 知识图谱专栏	中文技术博客与教程

F. 术语表（中英对照）

表 22–27

中文术语	英文术语	简要定义
知识图谱	Knowledge Graph	以图结构组织和表示知识的系统，由实体、关系和属性构成
实体	Entity	知识图谱中的节点，代表现实世界的对象或概念
关系	Relation / Relationship	连接两个实体的边，表示实体间的语义关联
属性	Property / Attribute	描述实体或关系特征的数据字段
三元组	Triple	知识图谱的基本单元，格式为（主体, 谓词, 客体）
本体	Ontology	对领域概念及其关系的形式化定义，是知识图谱的模式层
本体工程	Ontology Engineering	设计、构建和维护本体的系统化方法
模式层	Schema Layer (TBox)	知识图谱的概念模型，定义实体类型和关系类型
数据层	Data Layer (ABox)	知识图谱的实例层，存储具体的实体和关系数据

中文术语	英文术语	简要定义
知识抽取	Knowledge Extraction	从非结构化或半结构化数据中提取实体、关系和属性的过程
命名实体识别	Named Entity Recognition (NER)	从文本中识别出人名、地名、机构名等命名实体
关系抽取	Relation Extraction (RE)	从文本中自动识别实体间的语义关系
事件抽取	Event Extraction	从文本中识别事件类型及其参与者
知识融合	Knowledge Fusion	将多源异构数据整合为统一知识图谱的过程
实体对齐	Entity Alignment	识别不同知识图谱中指向同一现实实体的节点
实体链接	Entity Linking	将文本中的实体提及链接到知识库中的对应实体
实体消歧	Entity Disambiguation	区分同名实体指向不同现实对象的过程
知识推理	Knowledge Reasoning	基于已有知识推导出新知识的过程
知识补全	Knowledge Graph Completion	预测知识图谱中缺失的三元组

中文术语	英文术语	简要定义
知识图谱嵌入	Knowledge Graph Embedding (KGE)	将实体和关系映射到低维连续向量空间
TransE	Translating Embeddings	经典 KGE 方法，将关系建模为实体间的平移向量
图神经网络	Graph Neural Network (GNN)	在图结构数据上进行学习的深度学习模型
图卷积网络	Graph Convolutional Network (GCN)	基于谱图理论的图神经网络变体
图注意力网络	Graph Attention Network (GAT)	引入注意力机制的图神经网络
消息传递	Message Passing	GNN 中节点间传递和聚合信息的核心机制
图数据库	Graph Database	以图模型存储和查询数据的数据库系统
属性图	Property Graph	节点和关系均可携带属性的图数据模型
资源描述框架	Resource Description Framework (RDF)	W3C 标准的图数据模型，用三元组描述资源
SPARQL	SPARQL Protocol and RDF Query Language	RDF 图数据的标准查询语言

中文术语	英文术语	简要定义
Cypher	Cypher Query Language	Neo4j 开发的图查询语言
Gremlin	Gremlin	Apache TinkerPop 的图遍历语言
向量数据库	Vector Database	专门存储和检索高维向量的数据库
嵌入向量	Embedding Vector	将离散数据映射到连续向量空间的数值表示
余弦相似度	Cosine Similarity	衡量两个向量方向相似程度的指标
近似最近邻	Approximate Nearest Neighbor (ANN)	牺牲少量精度换取速度的最近邻搜索方法
HNSW	Hierarchical Navigable Small World	高效 ANN 索引算法
检索增强生成	Retrieval-Augmented Generation (RAG)	将信息检索与 LLM 生成结合的架构
大语言模型	Large Language Model (LLM)	基于海量文本训练的超大规模语言模型
提示工程	Prompt Engineering	设计有效提示词以引导 LLM 输出的技术
思维链	Chain of Thought (CoT)	让 LLM 逐步推理的提示技术

中文术语	英文术语	简要定义
图算法	Graph Algorithm	在图结构上执行的计算方法
PageRank	PageRank	衡量节点重要性的链接分析算法
社区发现	Community Detection	发现图中紧密连接子群的方法
Louvain算法	Louvain Method	高效的社区发现算法，优化模块度
最短路径	Shortest Path	图中两节点间权值最小的路径
中心性	Centrality	衡量节点在网络中重要程度的一组指标
介数中心性	Betweenness Centrality	衡量节点作为”桥梁”角色的中心性指标
子图	Subgraph	图中节点和关系的子集
图投影	Graph Projection	将数据库中的图数据加载到内存以执行算法
知识图谱构建	Knowledge Graph Construction	从零或在已有基础上构建知识图谱的全过程
ETL	Extract, Transform, Load	数据抽取、转换和加载的标准化流程

中文术语	英文术语	简要定义
数据治理	Data Governance	对数据资产进行管理和规范化的体系
语义网	Semantic Web	Tim Berners-Lee 提出的下一代 Web 愿景
链接数据	Linked Data	在 Web 上发布和互联结构化数据的原则
OWL	Web Ontology Language	用于定义和推理本体的标准语言
RDFS	RDF Schema	RDF 的模式扩展语言
图灵完备	Turing Complete	计算系统能模拟任何图灵机的能力
幻觉	Hallucination	LLM 生成看似合理但实际错误的内容
Grounding	Grounding	将 LLM 输出与外部知识库对齐验证
图谱问答	Knowledge Graph Question Answering (KGQA)	基于知识图谱回答自然语言问题
图嵌入	Graph Embedding	将整个图或图中节点/边/子图映射为向量

中文术语	英文术语	简要定义
联邦查询	Federated Query	同时查询多个 SPARQL 端点的查询方式
分片	Sharding	将数据分散存储在多个节点上的技术
模块度	Modularity	衡量社区划分质量的指标

> **编者注：**
>
> 本附录中的命令和 API 基于写作时的软件版本整理。软件迭代较快，实际使用时请以官方最新文档为准。Neo4j 命令基于 5.x 版本，Qdrant 基于 1.x 版本，Ubuntu 基于 22.04 LTS。

后记 写给未来的知识工程师

写这本书的念头，起源于一个很具体的时刻。

2024 年初，我在一个企业知识管理项目中遇到了一个典型困境：客户有几百万份文档、几十个业务系统、数千条审批流程，但当新员工问”这个零件的供应商是谁”时，没有人能快速给出准确答案。信息散落在各处，彼此孤立，像一座座没有桥梁连接的岛屿。

我们用了知识图谱来解决这个问题。过程并不顺利——数据质量差、本体设计反复推翻、ETL 管道频繁崩溃、团队成员对”图数据库”充满怀疑。但最终上线的那个系统，让一个入职三个月的新人能在五秒内找到她需要的答案。那一刻，我觉得所有的折腾都值了。

后来，越来越多的朋友问我：”知识图谱到底怎么入门？”我发现市面上的资料要么过于学术，满篇公式让人望而却步；要么过于碎片，东一篇西一篇拼不成体系。于是我决定把这些经验写下来。不是因为我掌握了全部答案，而是因为我踩过的坑实在太多了。多到我觉得，如果能有一本书帮后来者少踩哪怕一个坑，这本书就有了存在的意义。

写作的过程比想象中艰难得多。知识图谱是一个横跨多个学科领域的交叉地带——数据库、自然语言处理、图论、语义网、机器学习——任何一个方向都可以写一整本书。我不得不在”全面”和”可读”之间反复权衡，在”理论深度”和”上手速度”之间做出取舍。最终我选择了偏实战的路线：先让你能把东西做出来，再引导你去理解背后的原理。因为我相信，对于大多数读者来说，”做出来”带来的正反馈，才是持续学习最可靠的燃料。

写作期间，有不少朋友帮我审校初稿。一位做后端开发的朋友反馈说：”第七章 SPARQL 那节，我看了三遍才看懂，但看懂之后发现其实也没那么难。”这让我意识到，很多时候”难”不是知识本身的问题，而是讲述方式的问题。于是我反复打磨那些”卡壳”的章节，加入更多的类比和示意图，直到读起来像跟朋友聊天一样自然。当然，肯定还有一些地方不够好——这是技术写作的宿命，也是下一版的起点。

如果你读完这本书，想要继续深入，我有几个建议。

第一，不要只做练习题。找一个真实的问题——工作中的、生活中的、你好奇的——用知识图谱去解决它。真实问题的混乱和不确定性，是最好的老师。

第二，保持对底层原理的好奇。Cypher 语法会用就行，但理解属性图的存储原理会让你写出更高效的查询。向量搜索能跑通就行，但理解 ANN 索引的原理会让你在性能调优时少走弯路。

第三，不要把自己限定在”知识图谱工程师”这个标签里。未来的技术人才不会被某种单一技术定义。你是解决信息问题的人，图谱只是你工具箱中的一把利器。

第四，关注社区。知识图谱领域的开源生态非常活跃，Neo4j 社区、OpenKG、Qdrant 社区都有大量高质量的讨论和实践分享。一个人的摸索是孤独的，社区会让你知道你不是一个人在走这条路。

第五，允许自己慢下来。在这个信息爆炸的时代，我们太容易被焦虑裹挟——新框架发布了要学、新模型出来了要追、新论文发了要读。但真正扎实的成长从来不是靠速度堆出来的。有时候，花一个下午把一个简单的知识图谱项目做到极致，比蜻蜓点水地看十篇论文收获更大。慢，不是偷懒，是一种对质量的坚持。

这本书从第一行字到最后一个句号，大约花了一年半的时间。期间 Neo4j 发布了新版本，Qdrant 的功能迭代了好几轮，大语言模型的能力又跃升了一个台阶。有好几次，我觉得写完的内容已经过时了，推倒重来。但最终我还是选择把它们留了下来——因为比起追逐最新的技术名词，更重要的是把”为什么这样做”讲清楚。技术永远在跑，这本书也只能是某个时刻的快照。真正持久的，是你从中学到的思考方式。

写这本书的日子里，我常常在深夜对着屏幕发呆，思考如何把一个复杂的概念讲得更简单一些。有时候写了一整天，第二天重读觉得不满意，又全部删掉重来。这个过程让我深刻体会到了费曼的那句话：”如果你不能用简单的语言解释它，说明你还没有真正理解它。”写作本身就是最深刻的学习。

谢谢你读到这里。无论你是刚入行的数据工程师、正在转型的后端开发者、还是对 AI 充满好奇的产品经理——我希望这本书曾经给过你哪怕一次”原来如此”的瞬间。

那就够了。

慢慢走，不赶路，但不停步。

—— 树懒老K

2026年夏，于断层之上

树懒老K（拙一）

30年企业服务经验 · 专注AI智能体与组织变革

个人网站　　个人微信　　公众号

慢一点，深一度