Movie Knowledge Graph (IMDb CSV → GraphML)

本项目的目标是：
从 3 个 IMDb 相关的 CSV 文件中，**抽取公共字段**，构建一张以“电影”为中心的**知识图谱（Knowledge Graph）**，并导出为 `GraphML` 格式，方便在 Neo4j、Gephi、Cytoscape 或 Python 中进行查询和可视化。

---

1. 数据来源（Data Sources）

当前使用的数据文件示例：

- `data/IMDb_Dataset.csv`
- `data/IMDb_Dataset_2.csv`
- `data/IMDb_Dataset_3.csv`

这三个文件的列名不完全一致，但有一组**公共字段**在三个文件中都存在，本项目只基于这些共同字段建图：

- `Title`
- `Year`
- `IMDb Rating`
- `MetaScore`
- `Duration (minutes)`
- `Certificates`
- `Genre`
- `Director`
- `Star Cast`

> **重要约束：**
> 整个图谱只使用以上公共字段，不依赖任何某个文件独有的列（如海报链接、第二/第三类型等）。

---

2. 图谱结构概述（Graph Schema）

本图谱是一个**有向多重图**（`MultiDiGraph`），包含 4 类节点和 4 种主要关系。

2.1 节点类型（Node Types）

2.1.1 Movie 节点

- **类型标记**：`type = "movie"`
- **节点 ID 规则**：
  `movie::<Title> (<Year>)`
  例如：`movie::Inception (2010)`
- **来源字段（按 Title + Year 聚合后的属性）：**

- `title` ← `Title`
- `year` ← `Year`
- `imdb_rating` ← 同一电影在多个 CSV 中的 `IMDb Rating` 的**均值**
- `metascore` ← 同一电影的 `MetaScore` 的均值
- `duration_minutes` ← 同一电影的 `Duration (minutes)` 的均值
- `certificate` ← 同一电影的 `Certificates` 中出现次数最多的值
- `genre_primary` ← 同一电影的 `Genre` 中出现次数最多的值

> **去重策略：**
> 使用 `(Title, Year)` 作为电影实体的逻辑主键。
> 即：同名同年的电影记录（不管来自哪个 CSV）被视为同一部电影，合并为一个 Movie 节点。

2.1.2 Person 节点（导演 / 演员）

- **类型标记**：`type = "person"`
- **节点 ID 规则**：
  `person::<Name>`
  例如：`person::Tom Cruise`
- **来源字段：**
  - `Director`
  - `Star Cast`
- **节点属性：**
  - `name`：人物姓名（从 Director 或 Star Cast 中抽取）

> 一个 Person 节点可以同时作为导演和演员，具体角色通过边的类型体现（DIRECTED / ACTED_IN），而不是通过节点属性区分。

2.1.3 Genre 节点（类型）

- **类型标记**：`type = "genre"`
- **节点 ID 规则**：
  `genre::<Name>`
  例如：`genre::Action`
- **来源字段：**
  - `Genre`（仅使用公共字段里的主类型）
- **节点属性：**
  - `name`：类型名称（例如 "Action", "Drama", "Documentary"）

2.1.4 Certificate 节点（分级）

- **类型标记**：`type = "certificate"`
- **节点 ID 规则**：
  `certificate::<Name>`
  例如：`certificate::PG-13`
- **来源字段**：
  - `Certificates`
- **节点属性**：
  - `name`：分级名称（例如 "PG", "PG-13", "R", "G" 等）

2.2 关系类型（Edge Types）

所有边都带有 `relation` 属性，用于标识关系类型。

2.2.1 DIRECTED（导演了）

- **方向**：`Person → Movie`
- **关系类型标记**：`relation = "DIRECTED"`
- **来源字段**：`Director`

含义：
某个 Person 节点是某个 Movie 节点的导演。

2.2.2 ACTED_IN（参演了）

- **方向**：`Person → Movie`
- **关系类型标记**：`relation = "ACTED_IN"`
- **来源字段**：`Star Cast`

含义：
某个 Person 节点在某个 Movie 节点中出演 / 参与演出。

> `Star Cast` 原本是一个混合姓名的字符串字段，如 `"Tom CruiseHayley AtwellVing Rhames"`，脚本中通过启发式规则进行姓名拆分，然后为每个姓名建立 Person 节点并连边。

2.2.3 HAS_GENRE（属于类型）

- **方向**：`Movie → Genre`
- **关系类型标记**：`relation = "HAS_GENRE"`
- **来源字段**：`Genre`

含义：
某个 Movie 节点属于某个 Genre 节点。

> 当前版本只使用公共字段中的主类型 `Genre`。
> 如果未来需要多类型（第二、第三类型），可以在 schema 中扩展更多 HAS_GENRE 边。

2.2.4 HAS_CERTIFICATE（具有分级）

- **方向**：`Movie → Certificate`
- **关系类型标记**：`relation = "HAS_CERTIFICATE"`
- **来源字段**：`Certificates`

含义：
某个 Movie 节点拥有某个 Certificate 分级。

2.3 图的整体形态

整体图结构可以概括为：

- **Movie** 为中心节点
- 向外连接：
  - `Person`（导演/演员）
    - `Person -[:DIRECTED]-> Movie`
    - `Person -[:ACTED_IN]-> Movie`
  - `Genre`（类型）
    - `Movie -[:HAS_GENRE]-> Genre`
  - `Certificate`（分级）
    - `Movie -[:HAS_CERTIFICATE]-> Certificate`

这样构成一个以电影为中心的**异构知识图谱**，支持从电影出发查看导演、演员、类型和分级，也支持从人物或类型出发，反查相关电影。

---

3. 建图流程（Pipeline）

下面说明从 CSV 到 GraphML 的完整构建流程。

3.1 安装依赖

确保已安装 Python 及下面的依赖：

```
pip install pandas networkx
```

3.2 脚本入口

假定构图脚本为：`buildKG.py`
核心步骤如下：

1. 读取三个 CSV 文件
2. 只保留公共字段
3. 合并为一个总表 DataFrame
4. 基于 `(Title, Year)` 合并电影记录，构建 Movie 节点
5. 从 `Director` 和 `Star Cast` 提取 Person 节点与关系
6. 从 `Genre`、`Certificates` 构建 Genre / Certificate 节点与关系
7. 使用 NetworkX 构建有向多重图 `MultiDiGraph`
8. 导出为 `GraphML` 文件（例如 `imdb_kg.graphml`）

3.3 读取和合并数据

在脚本中，首先定义文件列表和公共字段：

```
CSV_FILES = [
    r"data\IMDb_Dataset.csv",
    r"data\IMDb_Dataset_2.csv",
    r"data\IMDb_Dataset_3.csv",
]

COMMON_COLS = [
    "Title",
    "IMDb Rating",
    "Year",
    "Certificates",
    "Genre",
    "Director",
    "Star Cast",
    "MetaScore",
    "Duration (minutes)",
]
```

然后：

1. 对每个 CSV：
    - 读取为 DataFrame
    - 检查是否包含所有 `COMMON_COLS`
    - 只保留这几列
2. 将三个 DataFrame `concat` 合并为一个大表
3. 对数值列进行类型转换（非数字转为 `NaN`）：
    - `IMDb Rating`

- `MetaScore`
- `Duration (minutes)`

4. 对字符串列进行基本清洗（`strip()` 去空格）

3.4 电影去重与 Movie 节点构建

- 使用 `df.groupby(["Title", "Year"])` 对合并后的大表按 `(Title, Year)` 分组。
- 每个分组代表"认为是同一部电影"的多条记录。
- 对每个分组：
  - 聚合数值字段：取均值（`mean()`）
  - 聚合分类字段（`Certificates`, `Genre`）：取出现频率最高的值（众数）
  - 调用 `add_movie_node(...)` 创建一个 Movie 节点：
    - 节点 ID：`movie::<Title> (<Year>)`
    - 节点属性：`title, year, imdb_rating, metascore, duration_minutes, certificate, genre_primary`

> 这样可以把来自不同 CSV 的同一电影整合为一个统一的节点。

3.5 Person 节点与导演/演员关系

3.5.1 导演（Director）

对于每个电影分组：

- 遍历该组的 `Director` 列（去重后）：
  - 为每个 director 名字创建/复用 Person 节点
  - 添加 `DIRECTED` 边：`Person → Movie`

3.5.2 演员（Star Cast）

`Star Cast` 通常是多个名字拼在一起的一段字符串，例如：

- `"Tom CruiseHayley AtwellVing Rhames"`

在脚本中：

- 定义 `split_star_cast(raw)` 函数，对这类字符串进行启发式拆分：
  - 按"小写字母后接大写字母"的位置切开
  - 适当地合并如 `Mc`、`Mac`、`De` 等前缀碎片
- 将每个拆出的姓名作为一个 Person 节点
- 为每个姓名添加 `ACTED_IN` 边：`Person → Movie`

> 拆分规则并不完美，但在没有额外标注的前提下，能较好地自动化构建演员关系。

3.6 Genre / Certificate 节点与关系

在电影分组内：

- 遍历该组的 `Genre` 值（去重后）：
  - 为每个类型创建/复用 Genre 节点
  - 添加 `HAS_GENRE` 边：`Movie → Genre`
- 遍历该组的 `Certificates` 值（去重后）：
  - 为每个分级创建/复用 Certificate 节点
  - 添加 `HAS_CERTIFICATE` 边：`Movie → Certificate`

3.7 使用 NetworkX 构建图并导出 GraphML

整体图结构使用：

```
G = nx.MultiDiGraph()
```

- 每个节点带有：
  - `type` 字段（movie / person / genre / certificate）
  - 以及对应的属性（如 title、year、name 等）
- 每条边带有：
  - `relation` 字段（DIRECTED / ACTED_IN / HAS_GENRE / HAS_CERTIFICATE）

构图完成后：

```
nx.write_graphml(G, "imdb_kg.graphml")
```

即可生成 GraphML 文件，用于后续查询和可视化。

---

4. 使用方式（简单示例）

4.1 在 Python / NetworkX 中加载 GraphML

```
import networkx as nx

G = nx.read_graphml("imdb_kg.graphml")

print("nodes:", G.number_of_nodes())
print("edges:", G.number_of_edges())
```

4.2 示例：查询某部电影的相关信息

```python
def find_movie_node(G, title, year=None):
    for n, data in G.nodes(data=True):
        if data.get("type") == "movie" and data.get("title") == title:
            if year is None or int(data.get("year", -1)) == year:
                return n
    return None

def show_movie_context(G, title, year=None):
    movie_id = find_movie_node(G, title, year)
    if movie_id is None:
        print("找不到电影:", title, year)
        return

    data = G.nodes[movie_id]
    print(f"电影: {data.get('title')} ({data.get('year')})")
    print(f"  IMDb Rating: {data.get('imdb_rating')}")
    print(f"  MetaScore:   {data.get('metascore')}")
    print(f"  Duration:    {data.get('duration_minutes')} min")

    directors, actors, genres, certs = set(), set(), set(), set()

    # in_edges: 谁指向这部电影（导演/演员）
    for u, v, edge in G.in_edges(movie_id, data=True):
        rel = edge.get("relation")
        ndata = G.nodes[u]
        if rel == "DIRECTED":
            directors.add(ndata.get("name", u))
        elif rel == "ACTED_IN":
            actors.add(ndata.get("name", u))

    # out_edges: 这部电影指向谁（类型/分级）
    for u, v, edge in G.out_edges(movie_id, data=True):
        rel = edge.get("relation")
        ndata = G.nodes[v]
        if rel == "HAS_GENRE":
            genres.add(ndata.get("name", v))
        elif rel == "HAS_CERTIFICATE":
            certs.add(ndata.get("name", v))

    print("  导演:", ", ".join(sorted(directors)) or "无")
    print("  演员:", ", ".join(sorted(actors)) or "无")
    print("  类型:", ", ".join(sorted(genres)) or "无")
    print("  分级:", ", ".join(sorted(certs)) or "无")

# 示例调用
show_movie_context(G, "Inception", 2010)
```

5. 后续扩展方向

本项目当前版本是基于**公共字段**的"最小可用图谱"。未来可以考虑：

- 利用各文件独有的字段（海报链接、多类型字段等）丰富 Movie 属性和关系；
- 引入 "DataSource" 节点表示不同来源；
- 加入国家、语言、制作公司、奖项等实体；
- 把演员间的合作次数、类型偏好等统计信息也编码到图中。

6. 图增强电影问答后端处理逻辑

这一节专门说明：当用户在界面里输入一句话提问时，后端是如何一步步把它变成「对图谱的结构化查询」，再把图上的结果交给大模型，生成最后的自然语言回答的。

整体分成两大阶段：

1. **查询计划生成（Planning）**：从自然语言 → 结构化 JSON 计划；
2. **结果解释 / 回答（Answering）**：从图查询结果 JSON → 中文回答（支持 Markdown）。

入口在 `api_server_stream.py` 的 `/api/qa_stream`：

```python
@app.post("/api/qa_stream")
def qa_stream(req: QuestionRequest):
    question = req.question.strip()

    def event_generator():
        # Step 1: 生成 plan
        plan_resp = client.chat.completions.create(
            model=PLAN_MODEL,
            messages=build_plan_messages(question),
            temperature=0.0,
        )
        plan_raw = plan_resp.choices[0].message.content
        plan = json.loads(plan_raw)

        # Step 2: 在知识图谱上执行计划
        graph_result = execute_plan(plan)

        # 把 plan + graph_result 先发给前端
        yield json.dumps({
            "type": "meta",
            "plan": plan,
            "graph_result": graph_result,
        }) + "\n"

        # Step 3: 调用深度思考模型生成回答（流式）
        completion = client.chat.completions.create(
            model=ANSWER_MODEL,
            messages=build_answer_messages(question, graph_result),
            extra_body={"enable_thinking": True},
            stream=True,
        )
```

```
        for chunk in completion:
            delta = chunk.choices[0].delta
            # reasoning_content → 思考过程
            if getattr(delta, "reasoning_content", None):
                yield json.dumps({
                    "type": "reasoning",
                    "text": delta.reasoning_content,
                }) + "\n"
            # content → 最终回答
            if getattr(delta, "content", None):
                yield json.dumps({
                    "type": "answer",
                    "text": delta.content,
                }) + "\n"

        yield json.dumps({"type": "done"}) + "\n"

    return StreamingResponse(event_generator(), media_type="text/plain; charset=utf-8")
```

下面分别展开「查询计划生成」和「结果解释/回答」两个阶段。

6.1 查询计划生成：从自然语言到结构化 Plan

这一步的目标：**不直接查询图，更不直接回答问题，而是先让大模型充当“查询规划器”**，
把用户的自然语言问题，翻译成一个非常严格的 JSON 结构：

```
{
  "task": "<某一个任务名>",
  "params": { ... 参数字典 ... }
}
```

6.1.1 约束模型输出的 System Prompt：`PLAN_SYSTEM_PROMPT`

在 `prompts.py` 中，`PLAN_SYSTEM_PROMPT` 详细列出了后端支持的「任务类型」和每种任务对应的参数：

```
PLAN_SYSTEM_PROMPT = """
你是一个“电影知识图谱查询规划器”。

现在有一个电影知识图谱，图里有这些查询接口（Python 函数）可以调用：

1. movie_basic_info
   - 描述：查询一部电影的基本信息（导演、演员、类型、分级、评分等）。
   - 对应函数：get_movie_basic_info(title)
   - params:
       - title: 电影名（字符串，必填）

2. movies_by_director
   - 描述：按导演查询他导演过的电影，可选年份过滤。
   - 对应函数：get_movies_by_director(name, year_min=None, year_max=None, limit=None)
   - params:
```

```
        - name: 导演名字（字符串，必填）
        - year_min: 最小年份（整数，可选）
        - year_max: 最大年份（整数，可选）
        - limit: 返回电影数量上限（整数，可选）

3. movies_by_actor
   - 描述：按演员查询他参演过的电影，可选年份过滤。
   - 对应函数：get_movies_by_actor(name, year_min=None, year_max=None, limit=None)
   - params 同上，只是 name 是演员名字。

4. movies_by_genre
   - 描述：按类型（Genre）查询电影，可选评分过滤。
   - 对应函数：get_movies_by_genre(genre_name, rating_min=None, limit=None)
   - params:
        - genre: 类型名称，如 "Action"、"Drama"（必填）
        - rating_min: IMDb 最小评分（浮点，可选）
        - limit: 返回电影数量上限（整数，可选）

5. similar_movies
   - 描述：找与某部电影相似的电影（基于共享导演/演员/类型的图结构）。
   - 对应函数：get_similar_movies_by_neighbors(title, top_k=None)
   - params:
        - title: 电影名（必填）
        - limit: 返回电影数目上限（整数，可选）

6. other_movies_by_director_of_movie
   - 描述：给定一部电影，先找到它的导演，再列出这些导演的其它作品。
   - 对应函数：get_other_movies_by_director_of_movie(title)
   - params:
        - title: 电影名（必填）

7. co_actors
   - 描述：查询某个演员的“合作演员”，按合作次数排序。
   - 对应函数：get_co_actors(name, top_k=None)
   - params:
        - name: 演员名字（必填）
        - limit: 返回的合作演员数量上限（整数，可选）

你的任务：
- 只负责把“用户的问题”转换成一个 JSON 查询计划。
- 不要直接回答问题内容，也不要解释。
- 只输出一个 JSON 对象，键为 "task" 和 "params"。
- 不要在 JSON 外多输出任何文字（没有注释、没有解释、没有 Markdown 代码块）。

严格的返回格式要求：

1. 你的整个回复必须是 **一段合法的 JSON**，不能有任何多余字符：
   - 不能出现 Markdown 代码块标记，例如 ```json 或 ```。
   - 不能出现中文说明文字、前后缀、注释等。
   - 不能在 JSON 前后追加其他内容。

2. JSON 的顶层结构必须是一个对象，且只包含两个字段：
   {
```

```
    "task": "<字符串>",
    "params": { ... }
  }

3. "task" 字段：
  - 类型：字符串
  - 取值只能是下面这几个之一：
    "movie_basic_info",
    "movies_by_director",
    "movies_by_actor",
    "movies_by_genre",
    "similar_movies",
    "other_movies_by_director_of_movie",
    "co_actors"

4. "params" 字段：
  - 类型：对象（可以为空对象 {}）
  - 里面只允许放需要的参数，不要放多余或未知字段。
  - 字段名全部用英文，例如：
      - 对于电影名用 "title"
      - 对于导演名用 "name"
      - 对于演员名用 "name"
      - 对于类型名用 "genre"
      - 对于年份下限用 "year_min"
      - 对于年份上限用 "year_max"
      - 对于评分下限用 "rating_min"
      - 对于数量上限用 "limit"
  - 各字段的类型：
      - "title" / "name" / "genre": 字符串
      - "year_min" / "year_max" / "limit": 整数
      - "rating_min": 浮点数（例如 8.0）

5. 如果用户问题中没有明确提到某个参数，就不要乱填，干脆不放进 "params" 里。

请务必记住：
- 你的回复中不能包含任何中文提示、解释或额外文本。
- 你的回复中不能包含 Markdown 代码块标记。
- 你的回复中不能包含注释。
- 你的回复必须是 ChatCompletion 的 content 字段可以直接被 json.loads() 正确解析的 JSON 字符串。
"""
```

这段 System Prompt 的作用可以概括为：

- 把模型的「角色」限定为**查询规划器**而不是回答者；
- 明确支持哪些任务、每个任务有哪些参数；
- 严格约束输出格式，使得后端可以安全地 `json.loads()`；
- 禁止出现 Markdown/注释/额外文字，防止打断 JSON 解析。

6.1.2 Few-shot 示例：`PLAN_FEWSHOT`

在此基础上，还通过 `PLAN_FEWSHOT` 给了若干典型例子，示范“问题 → 计划”的映射：

```
PLAN_FEWSHOT = [
    {
        "user": "《Inception》的导演和主要演员是谁？",
        "assistant": {
            "task": "movie_basic_info",
            "params": {
                "title": "Inception"
            }
        },
    },
    {
        "user": "Christopher Nolan 2000 年之后导演过哪些电影？最多给我 10 部。",
        "assistant": {
            "task": "movies_by_director",
            "params": {
                "name": "Christopher Nolan",
                "year_min": 2000,
                "limit": 10
            }
        },
    },
    {
        "user": "给我推荐几部 IMDb 评分大于 8 的动作片。",
        "assistant": {
            "task": "movies_by_genre",
            "params": {
                "genre": "Action",
                "rating_min": 8.0,
                "limit": 10
            }
        },
    },
    {
        "user": "我很喜欢《Inception》，还有哪些类似的电影可以看？",
        "assistant": {
            "task": "similar_movies",
            "params": {
                "title": "Inception",
                "limit": 10
            }
        },
    },
    {
        "user": "和《Inception》同一个导演的其他电影有哪些？",
        "assistant": {
            "task": "other_movies_by_director_of_movie",
            "params": {
                "title": "Inception"
            }
```

```
        },
    },
    {
        "user": "Tom Cruise 演过哪些电影？",
        "assistant": {
            "task": "movies_by_actor",
            "params": {
                "name": "Tom Cruise"
            }
        },
    },
    {
        "user": "经常和 Tom Cruise 合作的演员有哪些？",
        "assistant": {
            "task": "co_actors",
            "params": {
                "name": "Tom Cruise",
                "limit": 10
            }
        },
    },
]
```

这些 few-shot 告诉模型：

- 同样是“关于 Inception 的问题”，有时是查基本信息，有时是找相似电影，有时是找同一导演的其他作品；
- 年份、评分、数量等条件如何映射到 `year_min`、`rating_min`、`limit`；
- 导演/演员/类型分别用哪个字段表达。

6.1.3 实际调用过程：`generate_plan`

在代码侧，以上 Prompt 被封装进 `generate_plan`：

```python
def build_plan_messages(question: str):
    messages = [
        {"role": "system", "content": PLAN_SYSTEM_PROMPT},
    ]
    for ex in PLAN_FEWSHOT:
        messages.append({"role": "user", "content": ex["user"]})
        messages.append({
            "role": "assistant",
            "content": json.dumps(ex["assistant"], ensure_ascii=False),
        })
    messages.append({"role": "user", "content": question})
    return messages


def generate_plan(question: str) -> Dict[str, Any]:
    messages = build_plan_messages(question)
    resp = client.chat.completions.create(
        model=PLAN_MODEL,                # qwen3-max
        messages=messages,
```

```
        temperature=0.0,
    )
    raw = resp.choices[0].message.content
    try:
        plan = json.loads(raw)
    except json.JSONDecodeError:
        # 兜底方案：把问题当成电影名
        plan = {
            "task": "movie_basic_info",
            "params": {"title": question.strip()},
        }
    return plan
```

到这里为止，我们已经从自然语言问题得到了一个结构化的 plan。

6.2 图查询执行：`execute_plan`

有了 plan，下一步就是在知识图谱上执行，调用 `kg_api.py` 中已经封装好的查询函数。

核心函数 `execute_plan` 的思路是：

1. 解析 `plan["task"]`、`plan["params"]`；
2. 根据 `task` 路由到不同的 API；
3. 把图查询结果包装成统一结构返回。

示意代码（省略部分分支）：

```python
def execute_plan(plan: Dict[str, Any]) -> Dict[str, Any]:
    task = plan.get("task")
    params = plan.get("params") or {}

    if task == "movie_basic_info":
        title = params.get("title")
        data = kg_api.get_movie_basic_info(title)
        return {"task": task, "params": params, "result": data}

    elif task == "movies_by_director":
        name = params.get("name")
        year_min = params.get("year_min")
        year_max = params.get("year_max")
        limit = params.get("limit")
        data = kg_api.get_movies_by_director(
            name,
            year_min=year_min,
            year_max=year_max,
            limit=limit,
        )
        return {"task": task, "params": params, "result": data}

    elif task == "co_actors":
        name = params.get("name")
```

```
        limit = params.get("limit")
        data = kg_api.get_co_actors(name, top_k=limit)
        return {"task": task, "params": params, "result": data}

    # ... 其它 task 同理
```

`kg_api.py` 内部已经负责了：

- 如何从 `imdb_kg.graphml` 中根据 title / name 找到对应节点；
- 如何沿着导演/演员/类型等边遍历，过滤年份、评分等条件；
- 返回结构化的 Python 对象（字典/列表）。

`execute_plan` 不关心图的细节，只充当“任务分发器”。

6.3 结果解释 / 回答：从图结果到自然语言

当 `execute_plan` 返回了：

```
{
  "task": "...",
  "params": { ... },
  "result": <图查询出来的内容>
}
```

后端会把这个结构交给第二个大模型（`qwen3-8b`），让它在**只基于图结果**的前提下生成最终回答。

6.3.1 回答阶段的 System Prompt：`ANSWER_SYSTEM_PROMPT`

同样在 `prompts.py` 中：

```
ANSWER_SYSTEM_PROMPT = """
你是一个电影问答助手。

注意：
- 我已经帮你在一个电影知识图谱上执行好了查询，图谱中信息是真实可靠的。
- 我会把“用户问题”和“图查询结果(JSON)”都给你。
- 你的任务是：根据这些查询结果，用中文回答用户的问题。
- 如果查询结果为 null 或空列表，就如实告诉用户：
    “在当前图谱中没有在图谱里查到相关信息”，可以适当安慰一下。
- 不要编造图谱中没有的信息，也不要瞎编电影。
- 回答时可以适当组织结构，比如列表、项目符号等，但不要再输出原始 JSON。
"""
```

作用：

- 把模型的「角色」设定为**基于图结果的解释器**，而不是自由聊天模型；
- 强调：
    - 图结果是真实的；
    - 不能瞎编图中不存在的电影或事实；

- 没查到就老实说；
- 可以使用列表/标题等 Markdown 结构（方便前端渲染）。

6.3.2 组装消息并开启深度思考：`build_answer_messages` & `generate_answer`

消息构造函数：

```
def build_answer_messages(question: str, exec_result: Dict[str, Any]):
    system_prompt = ANSWER_SYSTEM_PROMPT
    user_content = f"""用户问题：
{question}

图查询结果（JSON）：
{json.dumps(exec_result, ensure_ascii=False, indent=2)}
"""
    return [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_content},
    ]
```

可以看到：

- `system` 里是刚才的 `ANSWER_SYSTEM_PROMPT`；
- `user` 消息里同时打包：
  - 原始自然语言问题；
  - 完整的图查询结果 JSON（带缩进，便于模型阅读）。

生成回答函数：

```
def generate_answer(question: str, exec_result: Dict[str, Any]) -> str:
    messages = build_answer_messages(question, exec_result)
    answer = stream_chat(
        model=ANSWER_MODEL,          # qwen3-8b
        messages=messages,
        enable_thinking=True,        # 开启思考模式
        debug_name="回答问题",
    )
    return answer
```

这里的 `stream_chat` 封装了：

- `extra_body={"enable_thinking": True}`——开启思考模式；
- 对返回的 `delta.reasoning_content` 和 `delta.content` 做流式处理。

在 HTTP 接口版本中，我们没有直接返回完整字符串，而是把每一小块都包装为 JSON 行发给前端：

- `type="reasoning"`：思考过程；
- `type="answer"`：最终回答文本（Markdown）；

这样前端就可以左侧滚动展示 reasoning，右侧用 Markdown 渲染 answer。

6.4 整个 QA 任务的端到端流程

把上面的步骤串起来，一个完整的 QA 流程大致是：

1. **用户提问**
   - 例如：

     > “和《Inception》同一个导演的其他电影有哪些？”

2. **规划阶段（qwen3-max）**
   - 调用 `generate_plan(question)`；
   - 利用 `PLAN_SYSTEM_PROMPT` + `PLAN_FEWSHOT` 约束输出；
   - 得到一个 JSON 计划，例如：

```
{
  "task": "other_movies_by_director_of_movie",
  "params": {
    "title": "Inception"
  }
}
```

   - 后端通过 `/api/qa_stream` 的第一条 `meta` 消息把这个 plan 回传给前端展示。

3. **图查询阶段（kg_api）**
   - 调用 `execute_plan(plan)`；
   - 根据 `task` 路由到 `kg_api.get_other_movies_by_director_of_movie("Inception")`；
   - 得到：
     - 原电影的基本信息；
     - 该电影导演的姓名；
     - 该导演的其他作品列表（包含标题、年份、评分等）。
   - 统一包装为：

```
{
  "task": "other_movies_by_director_of_movie",
  "params": {"title": "Inception"},
  "result": { ... }
}
```

   - 同样通过 `meta` 消息回传给前端作为 Step 2。

4. **回答阶段（qwen3-8b 思考模式）**
   - 调用 `generate_answer(question, exec_result)`：
     - 用 `ANSWER_SYSTEM_PROMPT` 限制模型只基于图结果回答；
     - 把完整 `exec_result` JSON 作为 user 内容的一部分；
     - 开启 `enable_thinking=True` + `stream=True`。
   - 后端在循环中不断读取模型输出：

        - `reasoning_content` → `{"type": "reasoning", "text": "..."}`
        - `content` → `{"type": "answer", "text": "..."}`
    - 直到模型结束，发送一条 `{"type": "done"}`。
5. **前端展示**
    - 不在本节展开细节，只需要知道：
        - 前端逐行读流；
        - `meta` 更新 Step1/Step2 面板；
        - `reasoning` 追加到“思考过程”窗口；
        - `answer` 追加到 Markdown 缓冲区并重新渲染为 HTML。

最终用户看到的是：

- 这次问题被解析成了什么查询任务（task + params）；
- 图谱上实际查到了哪些结构化数据；
- 模型是如何在这些数据的基础上“思考”的（reasoning）；
- 以及整理好的自然语言答案（answer）。

这就是整个“图增强电影问答”的后端处理逻辑与提示词设计的完整闭环。

7. 整体架构概览

从工程视角看，这个项目可以分成五层，由下到上依次为：

1. **数据层（CSV & GraphML）**
2. **图谱访问层（KG API）**
3. **LLM 规划 & 回答层（Prompts + Qwen 模型）**
4. **服务层（HTTP Streaming API）**
5. **交互层（前端页面）**

可以抽象成下面这样一张“自下而上”的链路：

```
IMDb CSV（三个文件）
      |
      ▼
buildKG.py  ──▶  imdb_kg.graphml
      |
      ▼
kg_api.py（对 GraphML 的封装）
      |
      ▼
movie_qa.py  +  prompts.py（规划 + 执行 + 回答）
      |
      ▼
api_server_stream.py（/api/qa_stream 流式接口）
      |
      ▼
浏览器前端（展示 Plan / Graph Result / Reasoning / Answer）
```

各层职责简要说明如下：

7.1 数据层：CSV → GraphML

- **输入**：三个 IMDb CSV 文件，只使用公共字段（Title / Year / IMDb Rating / MetaScore / Duration / Certificates / Genre / Director / Star Cast）。
- **核心脚本**：`buildKG.py`
  - 合并三份 CSV；
  - 按 `(Title, Year)` 去重，聚合数值/分类字段；
  - 利用 NetworkX 构建 `MultiDiGraph`；
  - 导出为 `imdb_kg.graphml`。

这一层的输出是"静态图谱文件"，后续所有查询都围绕它展开。

7.2 图谱访问层：`kg_api.py`

- **输入**：`imdb_kg.graphml`
- **核心职责**：
  - 加载 GraphML 到 NetworkX；
  - 封装一组**稳定的查询接口**，例如：
    - `get_movie_basic_info(title)`
    - `get_movies_by_director(name, ...)`
    - `get_movies_by_actor(name, ...)`
    - `get_movies_by_genre(genre, ...)`
    - `get_similar_movies_by_neighbors(title, ...)`
    - `get_other_movies_by_director_of_movie(title)`
    - `get_co_actors(name, ...)` 等。□filecite□turn3file0□
  - 屏蔽图结构细节（节点/边 ID、属性字段名），对上层只暴露"函数 + 参数"的调用方式。

可以把这一层理解为"图数据库的 DAO（数据访问层）"。

7.3 LLM 规划 & 回答层：`prompts.py` + `movie_qa.py`

这一层是整个系统的"脑子"，负责把自然语言和图查询函数连接起来。

- **prompts.py**：
  - 定义查询规划 Prompt：`PLAN_SYSTEM_PROMPT` + `PLAN_FEWSHOT`；
    - 限定模型只输出 `{ "task": ..., "params": {...} }` 形式的 JSON；
    - 明确列出每个 task 对应的 `kg_api` 函数和参数。□filecite□turn3file0□
  - 定义回答 Prompt：`ANSWER_SYSTEM_PROMPT`；
    - 告诉模型只能基于图结果回答，不能胡编；
    - 引导模型用结构化的中文回答（支持 Markdown）。

- **movie_qa.py**：
  - `generate_plan(question)`：调用 `qwen3-max` + 规划 Prompt → 生成查询计划；
  - `execute_plan(plan)`：根据 `plan["task"]` 路由到 `kg_api` 对应函数 → 拿到图查询结果；
  - `generate_answer(question, exec_result)`：调用 `qwen3-8b`（思考模式）+ 回答 Prompt → 生成最终回答；
  - `answer_question(question)`：将上述三步串成一个完整 pipeline。

这一层完成了"自然语言 ↔ 函数调用 ↔ 自然语言"的闭环。

7.4 服务层：`api_server_stream.py`

- 提供 HTTP 接口 `/api/qa_stream`，对外暴露"单轮问答"能力；
- 内部逻辑：
  1. 从请求中取出 `question`；
  2. 调用 LLM 规划模型生成 `plan`；
  3. 调用 `execute_plan(plan)` 在图上执行；
  4. 先发送一条 `type=meta` 的消息给前端（包含 plan + graph_result）；
  5. 再调用回答模型，以流式方式把 `reasoning_content` 和 `content` 逐条发给前端（`type=reasoning` / `type=answer`）；
  6. 最后发送一条 `type=done`。

这一层的输出是一条基于行分隔 JSON 的文本流，前端只要按行解析即可。

7.5 交互层：前端页面（简要）

- 通过 `fetch("/api/qa_stream")` 发送用户问题；
- 使用 ReadableStream 按行读取后端返回的 JSON；
- 根据 `type` 字段更新不同区域：
  - `meta` → Step1/Step2（Plan & Graph Result）卡片；
  - `reasoning` → 左侧"思考过程"窗口；
  - `answer` → 右侧 Markdown 回答面板；
  - `error` / `done` → 提示 & 状态更新。

这一层不改变语义，只负责可视化和交互体验。

---

8. 改进方向与下一步工作

目前的系统已经具备"从 CSV 构图 → 基于图的电影问答"的端到端能力，但从图谱本身、查询规划、回答质量以及系统工程等几个维度，还有不少扩展空间。

8.1 图谱层面的改进

1. **引入更多实体和关系**
   - 目前只建了 Movie / Person / Genre / Certificate 四类节点，可以考虑增加：
     - 国家（Country）、语言（Language）、公司（Production Company）、奖项（Awards）等；
     - “系列关系”（续集/前传）等电影间的边；
   - 有了这些扩展，可以回答更多类型的问题，例如：
     - “某个国家出品的高分动作片”；
     - “同一系列的所有电影及顺序”等。
2. **更精细的演员解析与补充**
   - 当前 `Star Cast` 拆分使用启发式规则，对部分姓名可能拆分不完美；
   - 可以：
     - 引入更鲁棒的姓名解析库；
     - 结合外部数据源（例如官方 IMDb 列表）修正演员列表；
     - 为演员增加更多属性（性别、国籍、出道年份等）。
3. **节点消歧与别名处理**
   - 对于同名不同人 / 不同电影（重名不同年份）已经通过 `(Title, Year)` 做了初步区分；
   - 进一步可以建立别名表，例如：
     - 不同地区译名；
     - 带副标题/子标题的变体；
   - 便于用户用中文或不完整英文标题进行模糊查询。
4. **图结构特征的利用**
   - 目前的相似电影只基于一跳邻居（共享导演/演员/类型的次数）；
   - 可以考虑：
     - 利用共现次数、路径长度、PageRank 等图算法；
     - 为每部电影计算一些“图特征向量”，便于后续做更精细的推荐或聚类。

8.2 查询规划与提示词优化

1. **支持多步查询计划**
   - 目前 `task` 只允许是一种类型，后续可以将 plan 扩展为“步骤列表”，例如：

```
{
  "steps": [
    {"task": "movies_by_director", ...},
    {"task": "movies_by_genre", ...},
    {"task": "intersect", ...}
  ]
}
```

   - 这样可以支持更复杂的问题：

        - “诺兰执导的电影中，哪几部同时属于科幻又评分大于 8？”
2. **对规划模型增加鲁棒性约束**
    - 增加更多 few-shot 示例，覆盖：
        - 没有年份/评分时如何处理；
        - 中文问题、混合中英文名字的情况；
    - 增加“错误示例 + 纠正”型 few-shot，引导模型避免常见错误：
        - 多输出字段；
        - 误用字段名（如 `director_name` vs `name`）。
3. **加入“无法识别意图”的安全兜底**
    - 当问题完全与电影无关时，可以让规划模型输出一个特殊 `task`：
        - 例如 `task = "unsupported"`；
    - 后端检测到后，直接给用户友好提示，而不是硬套某个查询函数。

8.3 回答阶段的增强

1. **引用支持（Citations）**
    - 在回答中标注关键信息来源自哪些电影/节点，例如：
        - “《Inception》(2010)，IMDb 评分约为 8.8（来自图谱数据）。”
    - 前端可以给节点增加可点击的“详情弹窗”，展示原始图数据。
2. **多粒度回答模式**
    - 支持“简要答复”和“详细分析”两种模式：
        - 简要模式：只列出结果和关键信息；
        - 详细模式：解释为什么推荐这些电影（相同导演/演员/类型等原因）。
3. **多语言支持**
    - 目前 Prompt 默认输出中文；
    - 可以在回答 Prompt 中增加“跟随用户语言”的约束：
        - 检测问题语言，自动用相同语言回答；
        - 或增加 `language` 参数明确指定输出语言。

8.4 系统工程与体验优化

1. **缓存与加速**
    - 对热门问题或中间结果（例如某个导演的作品列表）进行缓存；
    - 复用相同 plan 的执行结果，减少重复图查询开销。
2. **日志与可观测性**
    - 对每一次请求记录：
        - question / plan / graph_result 摘要 / answer；
    - 便于后续回放和调参与 Prompt 迭代。
3. **评测与对比实验**

    - 构造一小批标注好的 QA 数据集；
    - 对比不同 Prompt、不同规划模型、不同图构建方案对最终回答质量的影响。
4. **交互与可视化**
    - 在现在的 Plan / Graph Result / Reasoning / Answer 四块基础上：
        - 增加“局部子图可视化”（例如点击某部电影时弹出一跳邻居图）；
        - 支持将结果导出为 Markdown 报告。

整体而言，这套架构已经把“结构化图谱”与“大模型”比较干净地分层解耦：

- 底层图怎么构建、怎么演化，集中在数据层和 `kg_api` 中解决；
- 上层自然语言理解与回答，集中在 Prompt 和 Qwen 模型上演进；
- 中间的计划执行逻辑则起到“编排器”的作用，确保模型和图谱彼此各司其职。