ASD地板时光干预辅助系统 - 模块化架构与开发顺序设计

文档日期: 2026-01-26（2026-01-27更新） **版本**: v2.1 (Python版) **目标**: 顶层模块划分，明确职责与调用关系 **技术栈**: 后端统一使用 Python

---

目录

---

1. 架构设计原则

1.1 设计目标

- **高内聚低耦合**: 相关功能聚合在一起，模块间通过清晰接口通信
- **面向接口编程**: 先定义接口，再实现Mock，最后替换真实实现
- **并行开发**: 框架组和模块组可以并行开发，互不阻塞
- **渐进式开发**: 先用Mock跑通流程，再逐步替换真实模块
- **易于集成**: 所有模块最终集成到LangGraph框架

1.2 三层架构

```
第三层: LangGraph工作流层
  - 编排所有模块，实现完整业务流程
  - 只依赖接口，不依赖具体实现

第二层: 业务逻辑层
  - 实现核心业务逻辑，调用基础设施层
  - 提供接口定义 + Mock实现 + 真实实现

第一层: 基础设施层
  - 提供数据存储、AI能力等基础服务
  - 提供接口定义 + Mock实现 + 真实实现
```

1.3 面向接口编程策略

开发流程:

```
1. 定义所有模块接口（Python ABC抽象基类）
2. 实现所有Mock版本（返回假数据）
3. LangGraph调用接口（不关心Mock还是真实）
4. 跑通完整流程（全Mock）
5. 逐个替换真实实现
6. 通过配置切换Mock/真实
```

优势:

- ✅ 第1周就能跑通完整流程
- ✅ 快速验证架构设计
- ✅ 框架组和模块组并行开发
- ✅ 降低风险，随时可回退
- ✅ 易于测试和调试

---

2. 核心模块划分

说明: 每个模块包含三部分：接口定义、Mock实现、真实实现

第一层: 基础设施层（6个模块）

模块1: SQLite数据管理模块

接口定义:

```python
from abc import ABC, abstractmethod
from typing import Optional

class ISQLiteService(ABC):
    @abstractmethod
    async def get_child(self, child_id: str) -> Optional[ChildProfile]:
        pass

    @abstractmethod
    async def save_child(self, profile: ChildProfile) -> None:
        pass

    @abstractmethod
    async def create_session(self, child_id: str, game_id: str) -> str:
        pass
```

```
    @abstractmethod
    async def get_session(self, session_id: str) -> Optional[Session]:
        pass

    @abstractmethod
    async def save_weekly_plan(self, plan: WeeklyPlan) -> str:
        pass

    @abstractmethod
    async def get_weekly_plan(self, plan_id: str) -> Optional[WeeklyPlan]:
        pass
```

Mock实现:

- 返回硬编码的假数据
- 数据存在内存中
- 快速响应，便于测试

真实实现:

- 连接SQLite数据库
- 真实的CRUD操作
- 事务管理

职责: 管理所有结构化数据的存储和查询

核心功能:

- 孩子档案的增删改查
- 干预会话记录的存储和查询
- 周计划和日历事件的管理
- 观察记录的存储

被谁调用: 所有需要持久化数据的业务模块

依赖: 无

模块2: Graphiti记忆网络模块

接口定义:

```
from abc import ABC, abstractmethod
from typing import List
```

```python
class IGraphitiService(ABC):
    @abstractmethod
    async def save_memories(self, child_id: str, memories: List[Memory]) ->
None:
        pass

    @abstractmethod
    async def get_recent_memories(self, child_id: str, days: int) ->
List[Memory]:
        pass

    @abstractmethod
    async def analyze_trends(self, child_id: str, dimension: str) ->
TrendAnalysis:
        pass

    @abstractmethod
    async def detect_milestones(self, child_id: str) -> List[Milestone]:
        pass

    @abstractmethod
    async def build_context(self, child_id: str) -> ContextData:
        pass
```

Mock实现:

- 返回假的趋势数据（如"improving"）
- 返回假的里程碑列表
- 不实际存储数据

真实实现:

- 连接Graphiti网络
- 批量写入记忆节点
- 真实的时序分析

职责: 管理长期时序记忆，提供趋势分析和里程碑检测

核心功能:

- 将观察事件写入记忆网络（批量写入）
- 查询孩子的历史记忆
- 分析各维度的时序趋势
- 检测里程碑事件
- 检测平台期
- 构建当前上下文（最近趋势、关注点等）

被谁调用: 记忆更新模块、再评估模块、对话助手模块

依赖: 无

模块3: 知识库与RAG模块

接口定义:

```
from abc import ABC, abstractmethod
from typing import List, Optional, Any

class IKnowledgeRAGService(ABC):
    # 分层检索
    @abstractmethod
    async def search_methodology(self, query: str, top_k: int) ->
List[Knowledge]:
        pass

    @abstractmethod
    async def search_games(self, query: str, filters: GameFilters, top_k:
int) -> List[Game]:
        pass

    @abstractmethod
    async def search_scale(self, query: str, top_k: int) -> List[Knowledge]:
        pass

    @abstractmethod
    async def search_cases(self, query: str, top_k: int) -> List[Case]:
        pass

    # 跨层检索
    @abstractmethod
    async def search_across_layers(self, query: str, layers: List[str],
top_k: int) -> List[Knowledge]:
        pass

    # 按维度检索（游戏专用）
    @abstractmethod
    async def search_games_by_dimension(self, dimension: str, difficulty:
str, top_k: int) -> List[Game]:
        pass

    @abstractmethod
    async def search_games_by_interest(self, interest: str, top_k: int) ->
List[Game]:
        pass

    # 混合检索
    @abstractmethod
    async def hybrid_search(self, query: str, filters: Any, top_k: int) ->
List[Any]:
```

```python
        pass

    # 获取详情
    @abstractmethod
    async def get_game(self, game_id: str) -> Optional[Game]:
        pass

    @abstractmethod
    async def get_knowledge(self, knowledge_id: str) -> Optional[Knowledge]:
        pass
```

职责: 管理分层知识库，提供基于RAG的知识检索和推荐

RAG技术选型与架构设计

1. 知识库分层结构

第一层: 方法论知识库（理论层）

- 内容类型: 长文档、理论文章、专业书籍章节
- 数据量: 100-200篇文档
- 特点: 内容深度高、结构化程度低、需要理解上下文
- 示例:
  - 地板时光六大情绪发展里程碑详解
  - DIR模型理论基础
  - 干预原则和禁忌
  - 观察框架和评估方法
  - 家长指导手册

第二层: 游戏知识库（实践层）

- 内容类型: 结构化游戏数据
- 数据量: MVP阶段50个游戏，后期扩展到200+
- 特点: 高度结构化、元数据丰富、需要精确匹配
- 数据结构:

```
游戏 {
  基本信息: {名称, 描述, 适用年龄, 所需道具}
  目标维度: [eye_contact, two_way_communication, ...]
  难度等级: easy/medium/hard
  玩法步骤: [{step, instruction, tips}, ...]
  游戏变体: [{name, description, difficulty_change}, ...]
```

```
  注意事项: [...]
  成功案例: [...]
}
```

第三层: 量表知识库（评估层）

- 内容类型: 标准化量表题目和解读
- 数据量: 3套量表（CARS-2、ADOS-2、ATEC）
- 特点: 高度标准化、需要精确检索
- 数据结构:

```
量表 {
  量表类型: CARS-2/ADOS-2/ATEC
  题目编号: 1-50
  题目内容: "..."
  评分标准: {1分: "...", 2分: "...", ...}
  解读说明: "..."
  相关维度: [...]
}
```

第四层: 案例知识库（经验层，可选）

- 内容类型: 脱敏后的真实案例
- 数据量: 后期积累，初期可无
- 特点: 情境化、需要相似度匹配
- 数据结构:

```
案例 {
  孩子年龄: 3岁
  问题类型: "眼神接触困难"
  干预方法: "积木传递游戏"
  干预周期: 4周
  干预结果: "眼神接触从0次/天提升到5次/天"
  关键经验: [...]
}
```

---

2. RAG技术选型对比

方案A: 传统RAG（推荐用于MVP）

技术栈:

- 向量数据库: PostgreSQL + pgvector
- Embedding模型: 通义千问Embedding / OpenAI text-embedding-3
- 检索策略: 向量相似度 + 元数据过滤

优势:

- ✅ 简单易实现
- ✅ 成本低
- ✅ 适合结构化数据（游戏库）
- ✅ 查询速度快

劣势:

- ⚠️ 对长文档理解有限
- ⚠️ 缺乏知识图谱关系

适用场景:

- 游戏知识库（高度结构化）
- 量表知识库（标准化）
- MVP快速验证

方案B: GraphRAG（推荐用于方法论知识库）

技术栈:

- 知识图谱: Neo4j / 自建图结构
- 实体提取: LLM提取实体和关系
- 检索策略: 图遍历 + 向量检索

优势:

- ✅ 理解实体间关系
- ✅ 适合复杂理论文档
- ✅ 可以回答"为什么"类问题
- ✅ 支持多跳推理

劣势:

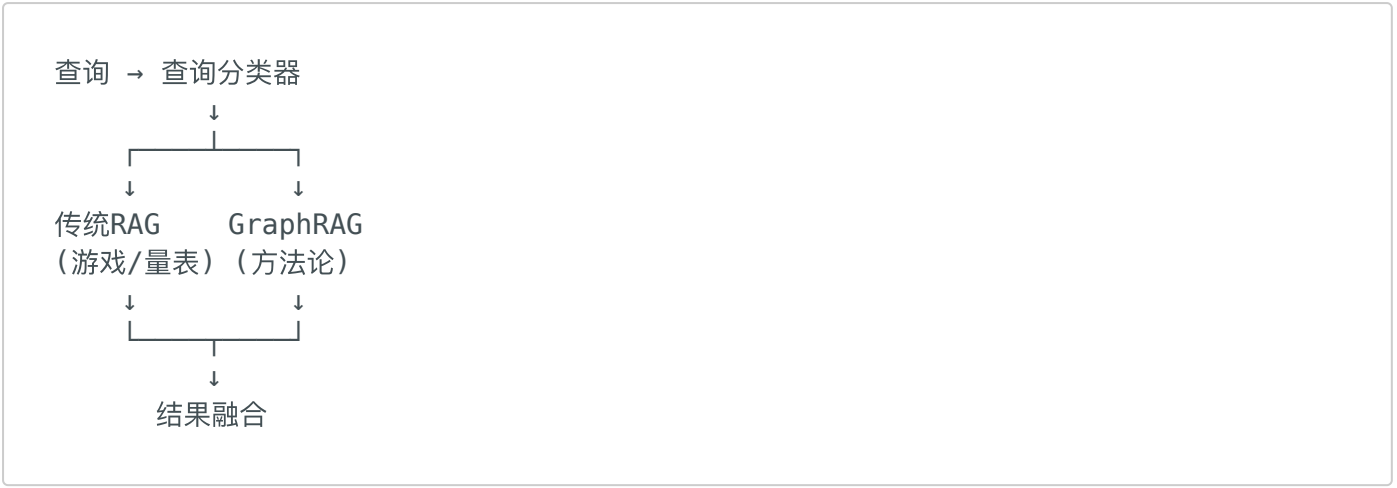
- ⚠️ 实现复杂
- ⚠️ 成本高（需要LLM提取实体）
- ⚠️ 构建知识图谱耗时

适用场景:

- 方法论知识库（理论文档）
- 需要理解概念间关系
- 后期优化阶段

方案C: 混合RAG（推荐的最终方案）

架构设计:



分层策略:

- 第一层（方法论）: GraphRAG（理解概念关系）
- 第二层（游戏）: 传统RAG（精确匹配）
- 第三层（量表）: 传统RAG（标准化检索）
- 第四层（案例）: 传统RAG（相似度匹配）

优势:

- ✅ 各取所长
- ✅ 灵活扩展
- ✅ 性能和效果平衡

---

3. MVP阶段推荐方案

采用传统RAG + 分层管理

理由:

1. 快速实现，2-3周可完成
2. 成本可控

3. 满足MVP需求
4. 后期可升级到混合RAG

技术栈:

- 向量数据库: PostgreSQL + pgvector
- Embedding: 通义千问Embedding（性价比高）
- 检索策略: 向量检索 + 元数据过滤 + 重排序

数据库设计:

```
-- 方法论知识表
CREATE TABLE methodology_knowledge (
  id VARCHAR(50) PRIMARY KEY,
  title VARCHAR(500),
  content TEXT,
  category VARCHAR(50), -- 理论/原则/观察/指导
  tags TEXT[],
  embedding vector(1536), -- pgvector
  metadata JSONB,
  created_at TIMESTAMP
);

-- 游戏知识表
CREATE TABLE game_knowledge (
  id VARCHAR(50) PRIMARY KEY,
  name VARCHAR(200),
  description TEXT,
  age_range VARCHAR(20), -- "2-3岁"
  target_dimensions TEXT[], -- ["eye_contact", ...]
  difficulty VARCHAR(20), -- easy/medium/hard
  required_props TEXT[],
  steps JSONB, -- [{step, instruction, tips}, ...]
  variations JSONB,
  tips TEXT[],
  embedding vector(1536),
  metadata JSONB,
  created_at TIMESTAMP
);

-- 量表知识表
CREATE TABLE scale_knowledge (
  id VARCHAR(50) PRIMARY KEY,
  scale_type VARCHAR(20), -- CARS-2/ADOS-2/ATEC
  item_number INTEGER,
  question TEXT,
  scoring_criteria JSONB,
  interpretation TEXT,
  related_dimensions TEXT[],
  embedding vector(1536),
  created_at TIMESTAMP
);
```

```
-- 向量索引
CREATE INDEX ON methodology_knowledge USING ivfflat (embedding
vector_cosine_ops);
CREATE INDEX ON game_knowledge USING ivfflat (embedding vector_cosine_ops);
CREATE INDEX ON scale_knowledge USING ivfflat (embedding vector_cosine_ops);
```

---

4. 检索策略设计

策略1: 单层精确检索（游戏库）

适用场景: 周计划推荐、按维度查找游戏

流程:

```
1. 构建查询向量（Embedding）
2. 向量相似度检索（pgvector）
3. 元数据过滤（年龄、难度、维度）
4. 重排序（考虑孩子画像）
5. 返回Top K
```

示例:

```
async def search_games_by_dimension(self, dimension: str, difficulty: str,
top_k: int) -> List[Game]:
    # 1. 构建查询
    query = f"适合提升{dimension}能力的{difficulty}难度游戏"
    query_embedding = await self.embed(query)

    # 2. 向量检索 + 元数据过滤
    results = await self.db.fetch_all("""
        SELECT *, 1 - (embedding <=> $1) as similarity
        FROM game_knowledge
        WHERE $2 = ANY(target_dimensions)
          AND difficulty = $3
        ORDER BY embedding <=> $1
        LIMIT $4
    """, query_embedding, dimension, difficulty, top_k)

    return results
```

策略2: 跨层融合检索（对话助手）

适用场景: 家长提问，不确定查哪个库

流程:

1. 查询分类（LLM判断问题类型）
2. 并行检索多个知识库
3. 结果加权融合
4. 去重和重排序
5. 返回Top K

示例:

```
async def search_across_layers(self, query: str, layers: List[str], top_k:
int) -> List[Knowledge]:
    # 1. 查询分类
    query_type = await self.classify_query(query)  # "theory" | "game" |
"scale"

    # 2. 并行检索
    tasks = []
    if 'methodology' in layers:
        tasks.append(self.search_methodology(query, top_k))
    if 'game' in layers:
        tasks.append(self.search_games(query, {}, top_k))
    if 'scale' in layers:
        tasks.append(self.search_scale(query, top_k))

    results = await asyncio.gather(*tasks)
    flat_results = [item for sublist in results for item in sublist]

    # 3. 加权融合（根据查询类型调整权重）
    weights = {
        "theory": {"methodology": 0.7, "game": 0.2, "scale": 0.1},
        "game": {"methodology": 0.1, "game": 0.8, "scale": 0.1},
        "scale": {"methodology": 0.1, "game": 0.1, "scale": 0.8}
    }

    weighted = [
        {**r, "score": r["similarity"] * weights[query_type][r["layer"]]}
        for r in flat_results
    ]

    # 4. 排序和去重
    weighted.sort(key=lambda x: x["score"], reverse=True)
    return weighted[:top_k]
```

策略3: 混合检索（精确+语义）

适用场景: 需要精确匹配某些条件

流程:

```
1. 关键词精确匹配（SQL LIKE）
2. 向量语义检索
3. 结果合并（RRF算法）
4. 返回Top K
```

示例:

```
async def hybrid_search(self, query: str, filters: Any, top_k: int) ->
List[Any]:
    # 1. 关键词检索
    keyword_results = await self.db.fetch_all("""
        SELECT *, 1.0 as keyword_score
        FROM game_knowledge
        WHERE name ILIKE $1 OR description ILIKE $1
    """, f"%{query}%")

    # 2. 向量检索
    embedding = await self.embed(query)
    vector_results = await self.db.fetch_all("""
        SELECT *, 1 - (embedding <=> $1) as vector_score
        FROM game_knowledge
        ORDER BY embedding <=> $1
        LIMIT $2
    """, embedding, top_k * 2)

    # 3. RRF融合
    merged = self.merge_with_rrf(keyword_results, vector_results)

    return merged[:top_k]
```

---

5. Mock实现

Mock实现:

```
class MockRAGService(IKnowledgeRAGService):
    def __init__(self):
        self.mock_games = [
            {
                "id": "game-001",
                "name": "积木传递游戏",
                "description": "通过传递积木建立眼神接触",
                "target_dimensions": ["eye_contact",
"two_way_communication"],
                "difficulty": "easy",
                "age_range": "2-3岁",
```

```
                "steps": [
                    {"step": 1, "instruction": "坐在孩子对面，拿起一块积木"},
                    {"step": 2, "instruction": "等待孩子看向你，然后递给他"},
                    {"step": 3, "instruction": "鼓励孩子递回来"}
                ]
            },
            # ... 更多假游戏
        ]

    async def search_games(self, query: str, filters: GameFilters, top_k:
int) -> List[Game]:
        print(f"[Mock RAG] 检索游戏: {query}")
        return self.mock_games[:top_k]

    async def search_games_by_dimension(self, dimension: str, difficulty:
str, top_k: int) -> List[Game]:
        print(f"[Mock RAG] 按维度检索: {dimension}, {difficulty}")
        result = [
            g for g in self.mock_games
            if dimension in g["target_dimensions"] and g["difficulty"] ==
difficulty
        ]
        return result[:top_k]
```

---

6. 真实实现

真实实现:

- PostgreSQL + pgvector
- 通义千问Embedding API
- 向量检索 + 元数据过滤
- 重排序算法

---

7. 后期优化方向

阶段1（MVP）：传统RAG + 分层管理 **阶段2（3个月后）**：引入GraphRAG处理方法论知识库 **阶段3（6个月后）**：混合RAG + 查询路由优化 **阶段4（1年后）**：个性化RAG + 用户反馈学习

被谁调用: 周计划推荐模块、实时指引模块、对话助手模块、初始评估模块

依赖: 无

模块4: AI视频解析模块

接口定义:

```
from abc import ABC, abstractmethod

class IVideoAnalysisService(ABC):
    @abstractmethod
    async def analyze_video(self, video_path: str, context: AnalysisContext)
-> VideoAnalysisResult:
        pass
```

Mock实现:

- 返回假的分析结果
- 假装检测到3次微笑、5次眼神接触
- 不实际分析视频

真实实现:

- 调用DeepSeek-Vision
- 真实的视频帧提取和分析
- 表情、眼神、动作识别

职责: 分析游戏视频，识别孩子的表情、眼神、动作、声音

被谁调用: 视频分析与验证模块

依赖: 无

模块5: 语音处理模块

接口定义:

```
from abc import ABC, abstractmethod

class ISpeechService(ABC):
    @abstractmethod
    async def speech_to_text(self, audio_path: str) -> str:
        pass

    @abstractmethod
    async def text_to_speech(self, text: str) -> str:
```

```
        """返回音频路径"""
        pass
```

Mock实现:

- STT返回"这是测试文字"
- TTS返回假的音频路径
- 不实际调用语音API

真实实现:

- 调用阿里云语音识别API
- 调用阿里云语音合成API
- 真实的音频处理

职责: 处理语音识别和合成

被谁调用: 观察捕获模块、实时指引模块

依赖: 无

模块6: 文档解析模块

接口定义:

```
from abc import ABC, abstractmethod
from typing import Literal

class IDocumentParserService(ABC):
    @abstractmethod
    async def parse_report(self, file_path: str, file_type: Literal["pdf",
"image"]) -> ParsedReport:
        pass

    @abstractmethod
    async def parse_scale(self, scale_data: ScaleData, scale_type: str) ->
ScaleResult:
        pass
```

Mock实现:

- 返回假的报告解析结果
- 假装提取了诊断、能力评估等

- 不实际做OCR和LLM提取

真实实现:

- OCR识别文字
- 调用LLM结构化提取
- 真实的报告解析

职责: 解析医院报告和量表数据

被谁调用: 初始评估模块

依赖: 无

第二层: 业务逻辑层（10个模块）

说明: 业务模块同样遵循接口定义 + Mock实现 + 真实实现的模式

模块7-16: 业务逻辑模块

为简洁起见，业务逻辑层的10个模块（初始评估、周计划推荐、实时指引、观察捕获、视频分析验证、总结生成、记忆更新、再评估、对话助手、可视化报告）都遵循相同的模式：

接口定义: 定义清晰的方法签名 **Mock实现**: 返回假数据，快速验证流程 **真实实现**: 调用基础设施层，实现真实业务逻辑

详细的接口定义和职责描述保持原文档内容不变。

模块7: 初始评估模块

职责: 处理孩子的初始评估，构建初始画像

核心功能:

- 接收医院报告或量表数据
- 调用文档解析模块解析数据
- 调用大模型融合报告和量表数据
- 生成孩子的初始画像（6大维度评分、优势、短板、兴趣、观察框架）

- 保存到SQLite

输入: 报告文件路径或量表数据

输出: 孩子画像（包含6大维度、自定义维度、优势、短板、兴趣、观察框架、关注点）

调用关系:

- 调用: 文档解析模块、SQLite数据管理模块
- 被调用: LangGraph工作流

依赖: 文档解析模块、SQLite数据管理模块

模块8: 周计划推荐模块

职责: 生成个性化的周游戏计划

核心功能:

- 从SQLite获取孩子画像
- 从Graphiti获取当前趋势和关注点
- 分析需要重点提升的维度（最多2个）
- 设定本周总目标（具体、可衡量、渐进式）
- 调用游戏知识库RAG检索候选游戏
- 生成7天游戏序列（渐进式难度、重复练习、场景切换）
- 为每天设定小目标（数值化、可验证）
- 保存周计划到SQLite

输入: 孩子ID、周开始日期、上周完成情况（可选）

输出: 周计划（包含本周总目标、7个每日计划、推荐理由）

调用关系:

- 调用: SQLite数据管理模块、Graphiti记忆网络模块、游戏知识库与RAG模块
- 被调用: LangGraph工作流

依赖: SQLite、Graphiti、游戏知识库

模块9: 实时指引模块

职责: 在游戏进行中提供分步指引和话术推荐

核心功能:

- 从游戏知识库获取游戏详细玩法
- 根据当前步骤生成指引文本
- 调用语音合成生成语音指引
- 根据孩子画像推荐话术
- 提醒禁忌话术

输入: 游戏ID、当前步骤、孩子画像、当前情况

输出: 步骤指引（文字、语音）、话术推荐、禁忌提醒

调用关系:

- 调用: 游戏知识库与RAG模块、语音处理模块
- 被调用: LangGraph工作流

依赖: 游戏知识库、语音处理模块

模块10: 观察捕获模块

职责: 处理家长的观察记录（快捷按钮、语音记录）

核心功能:

- 接收快捷按钮记录（😊👀🗣️🤝😢）并打上时间戳
- 接收语音记录，调用语音识别转文字
- 调用大模型结构化提取观察内容
- 保存到当前会话

输入:

- 快捷记录: 会话ID、观察类型、时间戳
- 语音记录: 会话ID、音频文件路径

输出: 结构化的观察记录（时间戳、类型、描述、上下文）

调用关系:

- 调用: 语音处理模块、SQLite数据管理模块

- 被调用: LangGraph工作流

依赖: 语音处理模块、SQLite

模块11: 视频分析与验证模块

职责: 分析视频并交叉验证家长的快捷记录

核心功能:

- 调用AI视频解析模块分析视频
- 对比家长的快捷记录和AI分析结果
- 识别一致、冲突、遗漏的观察
- 生成验证报告（不直接修改家长记录，呈现证据让家长判断）

输入: 会话ID、视频文件路径、家长的快捷记录

输出:

- 视频分析结果
- 验证结果（已验证的观察、冲突列表、家长遗漏的观察）

调用关系:

- 调用: AI视频解析模块、SQLite数据管理模块
- 被调用: LangGraph工作流

依赖: AI视频解析模块、SQLite

模块12: 总结生成模块

职责: 生成游戏总结和智能反馈表

核心功能:

- 融合已验证的观察记录、语音记录、视频分析结果
- 生成初步总结（亮点、需要关注的地方）
- 对比上次游戏和历史平均水平
- 基于初步总结动态生成3-5个反馈问题
- 接收家长反馈后，融合所有数据生成最终总结
- 生成可视化总结卡片

输入:

- 初步总结: 会话ID
- 反馈表生成: 初步总结
- 最终总结: 会话ID、家长反馈

输出:

- 初步总结
- 智能反馈表（3-5个问题）
- 最终总结（完整的游戏总结、里程碑标注、下一步建议）

调用关系:

- 调用: SQLite数据管理模块、Graphiti记忆网络模块（读取历史数据）
- 被调用: LangGraph工作流

依赖: SQLite、Graphiti

模块13: 记忆更新模块

职责: 将会话数据写入Graphiti记忆网络

核心功能:

- 融合所有会话数据（观察记录、视频分析、家长反馈、最终总结）
- 批量创建记忆节点（每个观察事件一个节点）
- 建立事件间的关系（因果、时序）
- 更新时序指标数据
- 重新构建当前上下文

输入: 会话ID、会话完整数据

输出:

- 写入成功/失败状态
- 更新后的上下文数据
- 更新后的时序指标

调用关系:

- 调用: Graphiti记忆网络模块、SQLite数据管理模块

- 被调用: LangGraph工作流

依赖: Graphiti、SQLite

关键优化: 使用Graphiti的批量API，一次性写入所有节点和边，减少调用次数

模块14: 再评估模块

职责: 基于新数据重新评估孩子，更新画像

核心功能:

- 从Graphiti检测里程碑事件
- 检测平台期（连续3周无变化）
- 对比基线数据，生成进展报告
- 更新孩子画像（各维度当前水平、兴趣偏好、干预重点）
- 判断是否需要调整周计划

输入: 孩子ID、新会话数据

输出:

- 里程碑列表
- 平台期预警
- 更新后的孩子画像
- 下一步建议
- 是否需要调整计划的标志

调用关系:

- 调用: Graphiti记忆网络模块、SQLite数据管理模块
- 被调用: LangGraph工作流

依赖: Graphiti、SQLite

模块15: 对话助手模块

职责: 处理家长的对话查询，提供个性化回答

核心功能:

- 理解家长的问题
- 根据问题类型路由到不同知识源：
  - 专业问题 → 查询地板时光专业知识库（RAG）
  - 游戏问题 → 查询游戏知识库（RAG）
  - 孩子进步问题 → 查询Graphiti记忆网络
- 融合孩子的实际数据生成个性化回答
- 引用知识来源和相关记忆

输入: 孩子ID、用户消息、对话历史

输出: 回答文本、知识来源、相关记忆

调用关系:

- 调用: 游戏知识库与RAG模块、Graphiti记忆网络模块、SQLite数据管理模块
- 被调用: 独立的对话API（不在主工作流中）

依赖: 游戏知识库、Graphiti、SQLite

说明: 对话助手是独立流程，不在主工作流中，通过独立的API调用

模块16: 可视化与报告模块

职责: 生成可视化图表和报告

核心功能:

- 从SQLite和Graphiti获取数据
- 生成多维度雷达图
- 生成里程碑时间线
- 生成趋势图
- 生成日历热力图
- 生成家长版报告（PDF）
- 生成医生版报告（PDF）

输入: 孩子ID、时间范围、报告类型

输出: 可视化数据、PDF报告文件

调用关系:

- 调用: SQLite数据管理模块、Graphiti记忆网络模块
- 被调用: 前端页面、报告生成API

依赖: SQLite、Graphiti

第三层: LangGraph工作流层（1个模块）

模块17: LangGraph主工作流

职责: 编排所有业务模块，实现完整的7步闭环流程

核心功能:

- 定义State结构（孩子信息、会话数据、上下文、指标、工作流状态）
- 定义节点（每个节点调用一个或多个业务模块）
- 定义边（节点间的流转逻辑）
- 实现HITL暂停点（等待家长填写反馈表）
- 实现Checkpoint机制（支持暂停和恢复）
- 实现条件路由（如是否上传视频、是否需要调整计划）

主要流程:

```
1. 初始评估 → 构建画像
2. 生成周计划 → 加入日历
3. 开始游戏 → 实时指引 → 观察捕获
4. 结束游戏 → 视频分析（可选）→ 观察验证
5. 生成初步总结 → 生成反馈表
6. [HITL暂停] 等待家长填写反馈
7. 生成最终总结 → 记忆更新 → 再评估
8. 判断是否需要调整 → 返回步骤2或结束
```

State管理:

- 孩子画像（从SQLite加载，缓存在State）
- 当前会话数据（实时更新）
- Graphiti上下文（从Graphiti加载，缓存在State，避免重复查询）
- 时序指标（从Graphiti加载，缓存在State）
- 会话历史（从SQLite加载，缓存在State）
- 工作流控制信息（当前节点、是否暂停、Checkpoint ID）

调用关系:

- 调用: 所有业务逻辑层模块
- 被调用: 前端API

依赖: 所有业务逻辑层模块

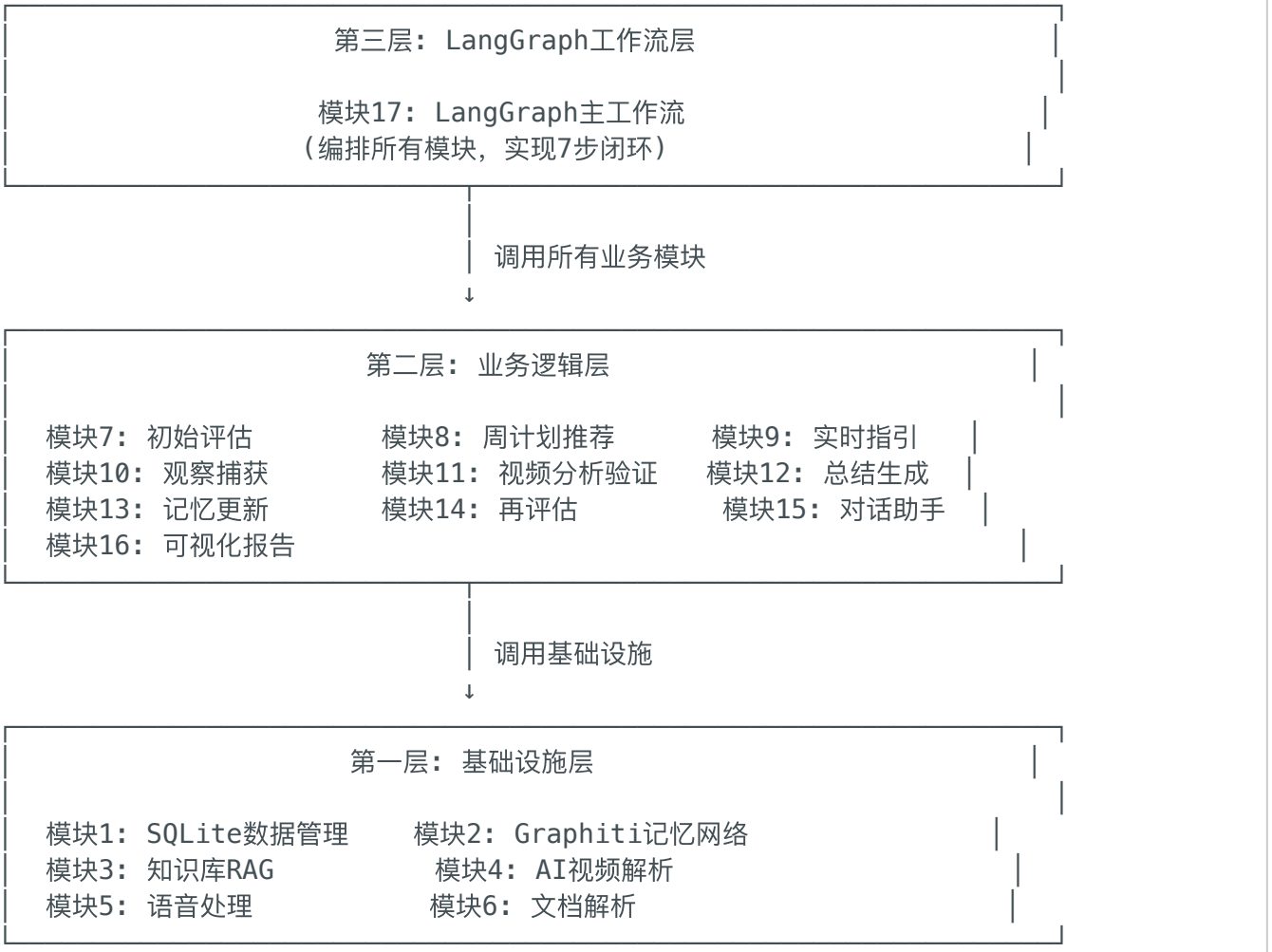
关键设计:

- 尽量少的function call: 优先使用State中已有的数据
- 多用State数据: 避免重复查询Graphiti和SQLite
- Graphiti写入策略: 仅在记忆更新节点统一写入，而非多次调用

---

3. 模块调用关系图

3.1 整体架构图



3.2 详细调用关系

初始评估流程

```
LangGraph工作流
    ↓
模块7: 初始评估
    ├→ 模块6: 文档解析 (解析报告/量表)
    └→ 模块1: SQLite (保存画像)
```

周计划推荐流程

```
LangGraph工作流
    ↓
模块8: 周计划推荐
    ├→ 模块1: SQLite (获取孩子画像)
    ├→ 模块2: Graphiti (获取趋势和关注点)
    ├→ 模块3: 知识库RAG (检索游戏知识库层)
    └→ 模块1: SQLite (保存周计划)
```

游戏实施流程

```
LangGraph工作流
    ↓
模块9: 实时指引
    ├→ 模块3: 知识库RAG (检索游戏知识库层+方法论知识库层)
    └→ 模块5: 语音处理 (生成语音指引)

同时并行:

模块10: 观察捕获
    ├→ 模块5: 语音处理 (语音转文字)
    └→ 模块1: SQLite (保存观察记录)
```

视频分析流程

```
LangGraph工作流
    ↓
模块11: 视频分析与验证
    ├→ 模块4: AI视频解析（分析视频）
    ├→ 模块1: SQLite（获取快捷记录）
    └→ 模块1: SQLite（保存验证结果）
```

总结生成流程

```
LangGraph工作流
    ↓
模块12: 总结生成
    ├→ 模块1: SQLite（获取会话数据）
    ├→ 模块2: Graphiti（获取历史数据，从State读取）
    └→ 模块1: SQLite（保存总结）
```

记忆更新流程

```
LangGraph工作流
    ↓
模块13: 记忆更新
    ├→ 模块1: SQLite（获取会话完整数据）
    ├→ 模块2: Graphiti（批量写入记忆节点和关系）
    └→ 模块2: Graphiti（重新构建上下文）
```

再评估流程

```
LangGraph工作流
    ↓
模块14: 再评估
    ├→ 模块2: Graphiti（检测里程碑、平台期）
    ├→ 模块1: SQLite（获取基线数据）
    └→ 模块1: SQLite（更新画像）
```

对话助手流程（独立）

```
前端对话API
    ↓
模块15: 对话助手
    ├→ 模块3: 知识库RAG（跨层检索: 方法论/游戏/量表知识库）
    ├→ 模块2: Graphiti（查询孩子记忆）
    └→ 模块1: SQLite（获取孩子画像）
```

可视化报告流程（独立）

```
前端页面/报告API
    ↓
模块16: 可视化与报告
    ├→ 模块1: SQLite（获取会话数据、周计划）
    └→ 模块2: Graphiti（获取时序数据、里程碑）
```

3.3 模块依赖矩阵

模块	依赖的模块
模块1: SQLite	无
模块2: Graphiti	无
模块3: 知识库RAG	无
模块4: AI视频解析	无
模块5: 语音处理	无
模块6: 文档解析	无
模块7: 初始评估	模块1, 模块6
模块8: 周计划推荐	模块1, 模块2, 模块3
模块9: 实时指引	模块3, 模块5
模块10: 观察捕获	模块1, 模块5
模块11: 视频分析验证	模块1, 模块4
模块12: 总结生成	模块1, 模块2
模块13: 记忆更新	模块1, 模块2

模块	依赖的模块
模块14: 再评估	模块1, 模块2
模块15: 对话助手	模块1, 模块2, 模块3
模块16: 可视化报告	模块1, 模块2
模块17: LangGraph工作流	模块7-16

---

4. 开发顺序与并行策略（面向接口）

4.1 新的开发策略

第1周：接口定义 + 全Mock + LangGraph框架

目标: 跑通完整流程（全Mock）

任务:

1. 定义所有17个模块的接口（Python ABC抽象基类）
2. 实现所有模块的Mock版本
3. 搭建LangGraph基础框架
4. 实现依赖注入容器
5. 实现核心节点（调用接口）
6. 跑通完整7步闭环（全Mock数据）

成果:

- 一个能跑的完整系统（虽然都是假数据）
- 验证了架构设计
- 团队对流程有清晰认识

并行开发:

- 1人：定义接口 + 搭建框架
- 2-3人：实现Mock模块

第2-4周：替换第一批真实实现

目标: 核心功能可用

优先替换:

- ✅ 模块1: SQLite真实实现（LangGraph Checkpoint需要）
- ✅ 模块6: 文档解析真实实现（核心功能）
- ✅ 模块3: RAG真实实现（至少游戏库）
- ✅ 模块7: 初始评估真实实现

保持Mock:

- 模块2: Graphiti（暂时用SQLite存历史）
- 模块4: 视频分析（先跳过）
- 模块5: 语音处理（先用文字）
- 其他业务模块

成果:

- 能真实上传报告并生成画像
- 能真实检索游戏并生成周计划

并行开发:

- 框架组：继续完善LangGraph
- 模块组：4个模块可并行开发

第5-8周：替换第二批真实实现

目标: 游戏流程可用

优先替换:

- ✅ 模块5: 语音处理真实实现
- ✅ 模块8: 周计划推荐真实实现
- ✅ 模块9: 实时指引真实实现
- ✅ 模块10: 观察捕获真实实现
- ✅ 模块12: 总结生成真实实现

保持Mock:

- 模块2: Graphiti（继续Mock）

- 模块4: 视频分析（继续Mock）
- 模块11: 视频验证（继续Mock）

成果:

- 能完整玩一次游戏
- 能语音记录观察
- 能生成总结

并行开发:

- 5个模块可并行开发

第9-12周：替换第三批真实实现

目标: 完整闭环

优先替换:

- ✅ 模块2: Graphiti真实实现
- ✅ 模块13: 记忆更新真实实现
- ✅ 模块14: 再评估真实实现

保持Mock:

- 模块4: 视频分析（最后实现）
- 模块11: 视频验证（最后实现）

成果:

- 完整的7步闭环
- 长期记忆和趋势分析
- 里程碑检测

并行开发:

- 3个模块可并行开发

第13-16周：替换最后的真实实现

目标: 全功能

优先替换:

- ✅ 模块4: 视频分析真实实现
- ✅ 模块11: 视频验证真实实现
- ✅ 模块15: 对话助手真实实现
- ✅ 模块16: 可视化报告真实实现

成果:

- 所有功能完整
- 视频分析可用
- 对话助手可用

4.2 依赖注入与配置切换

服务容器

```python
# services/container.py
from typing import Dict, Any

class ServiceContainer:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance._services: Dict[str, Any] = {}
        return cls._instance

    def register(self, name: str, implementation: Any) -> None:
        self._services[name] = implementation

    def get(self, name: str) -> Any:
        return self._services.get(name)

container = ServiceContainer()
```

环境配置

```python
# app/config.py
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
```

```
    SQLITE_MODE: str = "mock"  # "mock" | "real"
    GRAPHITI_MODE: str = "mock"
    RAG_MODE: str = "mock"
    VIDEO_MODE: str = "mock"
    SPEECH_MODE: str = "mock"
    DOCUMENT_MODE: str = "mock"
    LLM_MODE: str = "mock"

    class Config:
        env_file = ".env"

settings = Settings()
```

服务工厂

```
# services/factory.py
from app.config import settings
from services.mock.sqlite_mock import SQLiteMock
from services.impl.sqlite_impl import SQLiteImpl

def create_sqlite_service() -> ISQLiteService:
    if settings.SQLITE_MODE == "real":
        return SQLiteImpl()
    return SQLiteMock()

# 初始化
container.register("sqlite_service", create_sqlite_service())
```

使用方式

```
# 全部使用Mock（第1周）
npm run dev

# 使用真实SQLite和RAG（第2-4周）
USE_REAL_SQLITE=true USE_REAL_RAG=true npm run dev

# 逐步增加真实模块（第5-16周）
USE_REAL_SQLITE=true USE_REAL_RAG=true USE_REAL_SPEECH=true npm run dev
```

4.3 LangGraph节点调用接口

关键: 节点只依赖接口，不依赖具体实现

```python
# graph/nodes/assessment.py
from services.container import container

async def assessment_node(state: dict) -> dict:
    # 从容器获取服务（可能是Mock，也可能是真实）
    document_parser = container.get("document_parser")
    assessment_service = container.get("assessment_service")
    sqlite_service = container.get("sqlite_service")

    # 调用接口方法
    parsed_report = await document_parser.parse_report(state["report_path"],
"pdf")
    portrait = await assessment_service.build_portrait(parsed_report)
    await sqlite_service.save_child({**state["child_profile"], "portrait":
portrait})

    return {
        "child_profile": {**state["child_profile"], "portrait": portrait},
        "current_step": "weekly_plan"
    }
```

优势:

- 节点代码不需要修改
- 通过配置切换Mock/真实
- 易于测试和调试

4.4 开发时间线（修订版）

```
月份          Week 1    Week 2    Week 3    Week 4
──────────────────────────────────────────────────
第1个月       [接口定义 + 全Mock + 框架搭建]
                        [替换SQLite + 文档解析 + RAG]

第2个月       [替换评估 + 周计划]
                        [替换语音 + 指引 + 观察 + 总结]

第3个月       [继续替换业务模块]
                        [替换Graphiti + 记忆 + 再评估]

第4个月       [替换视频分析 + 对话助手 + 可视化]
                        [完善和优化]

第5个月       [前端开发 - 4个页面并行]

第6个月       [集成测试与优化]
```

4.5 关键路径分析（修订版）

新的关键路径:

```
Week 1: 接口定义 + 全Mock → 能跑
    ↓
Week 2-4: 替换3个核心模块 → 能用
    ↓
Week 5-8: 替换5个重要模块 → 能玩
    ↓
Week 9-12: 替换3个完善模块 → 完整闭环
    ↓
Week 13-16: 替换4个锦上添花模块 → 全功能
    ↓
Week 17-20: 前端开发
    ↓
Week 21-24: 测试上线
```

并行机会:

- Week 1: 接口定义和Mock实现可并行
- Week 2-4: 3个核心模块可并行开发
- Week 5-8: 5个重要模块可并行开发
- Week 9-12: 3个完善模块可并行开发
- Week 13-16: 4个锦上添花模块可并行开发

风险控制:

- 每周都有可运行的版本
- 随时可以回退到Mock
- 降低技术风险
- 提高团队士气

---

5. LangGraph集成方案

5.1 State结构设计

State是整个工作流的核心数据结构，包含以下部分:

孩子信息

- 孩子ID
- 孩子画像（从SQLite加载，包含6大维度、优势、短板、兴趣等）

当前会话

- 会话ID
- 游戏ID
- 会话状态（未开始、进行中、已结束、分析中）
- 开始时间、结束时间
- 快捷观察记录列表
- 语音观察记录列表
- 是否有视频
- 视频分析结果（如果有）
- 已验证的观察记录（如果有）
- 初步总结（如果有）
- 反馈表（如果有）
- 家长反馈（如果有）
- 最终总结（如果有）
- 归档数据（如果有）

Graphiti上下文（缓存）

- 各维度的趋势分析
- 里程碑列表
- 关注点列表
- 最近记忆
- 最后更新时间

时序指标（缓存）

- 各维度的指标数据点
- 最后更新时间

会话历史（缓存）

- 历史会话列表

周计划

- 当前周计划（如果有）

对话历史

- 对话消息列表

工作流控制

- 当前节点名称
- 下一个节点名称
- 是否HITL暂停
- Checkpoint ID

5.2 主工作流设计

工作流是一个有向图，包含以下节点和边:

节点列表

1. **评估节点**: 调用模块7（初始评估）
2. **周计划生成节点**: 调用模块8（周计划推荐）
3. **游戏开始节点**: 创建会话，初始化会话数据
4. **游戏指引节点**: 调用模块9（实时指引）
5. **游戏结束节点**: 标记会话结束
6. **视频上传节点**: 等待家长上传视频（可选）
7. **视频分析节点**: 调用模块11（视频分析与验证）
8. **观察验证节点**: 验证快捷记录
9. **初步总结节点**: 调用模块12（生成初步总结）
10. **反馈表生成节点**: 调用模块12（生成反馈表）
11. **HITL暂停节点**: 等待家长填写反馈表
12. **最终总结节点**: 调用模块12（生成最终总结）
13. **记忆更新节点**: 调用模块13（记忆更新）
14. **再评估节点**: 调用模块14（再评估）

边的流转逻辑

```
评估节点 → 周计划生成节点 → 游戏开始节点 → 游戏指引节点 → 游戏结束节点

游戏结束节点 → [条件判断]
    - 如果有视频 → 视频分析节点 → 观察验证节点
    - 如果无视频 → 初步总结节点

观察验证节点 → 初步总结节点

初步总结节点 → 反馈表生成节点 → HITL暂停节点

HITL暂停节点 → 最终总结节点 → 记忆更新节点 → 再评估节点

再评估节点 → [条件判断]
    - 如果需要调整计划 → 周计划生成节点
    - 如果不需要调整 → 结束
```

5.3 HITL暂停机制

目的: 在需要家长输入时暂停工作流

实现方式:

1. 工作流执行到HITL暂停节点时，保存当前State到Checkpoint
2. 标记工作流为暂停状态
3. 返回给前端，提示家长填写反馈表
4. 家长提交反馈后，前端调用恢复API
5. 从Checkpoint恢复State，更新家长反馈数据
6. 继续执行工作流

5.4 Checkpoint机制

目的: 支持工作流的暂停和恢复

实现方式:

- 使用LangGraph的SQLite Checkpointer
- 每个会话有唯一的thread_id（如"session-{sessionId}"）
- 在HITL暂停点自动保存Checkpoint
- 恢复时使用相同的thread_id继续执行

5.5 State数据优化策略

核心原则: 尽量少的function call，多用State数据

优化措施:

1. **Graphiti数据缓存**:
    - 在周计划生成节点，从Graphiti获取上下文和指标，缓存到State
    - 后续节点（总结生成、再评估）直接从State读取，不再调用Graphiti
    - 只在记忆更新节点更新Graphiti，并刷新State缓存
2. **SQLite数据缓存**:
    - 在工作流开始时，从SQLite加载孩子画像和会话历史，缓存到State
    - 后续节点直接从State读取
    - 只在需要持久化时写入SQLite
3. **批量操作**:
    - Graphiti写入使用批量API，一次性写入所有节点和边
    - SQLite写入尽量合并，减少IO次数
4. **按需加载**:
    - 只在需要时才加载数据
    - 例如，如果没有视频，就不加载视频分析相关数据

5.6 LangGraph如何调用模块

核心理解: 根据业务场景灵活选择调用方式

两种调用方式

方式1: 强制性调用（确定性流程）

适用场景：流程固定、逻辑明确、不需要智能决策的场景

节点函数直接调用业务模块：

```
节点函数的职责：
1. 从State中读取需要的数据
2. 直接调用一个或多个业务模块
3. 处理模块返回的结果
4. 更新State
5. 返回State的更新部分
```

典型场景：

- 评估节点：必须解析报告 → 必须构建画像
- 记忆更新节点：必须写入Graphiti → 必须更新上下文
- 视频分析节点：必须分析视频 → 必须验证观察

示例: 评估节点（强制性调用）

```
评估节点函数：
  1. 从State读取: 报告文件路径
  2. 直接调用模块6: 文档解析模块.解析报告(文件路径)
  3. 直接调用模块7: 初始评估模块.构建画像(解析结果)
  4. 直接调用模块1: SQLite模块.保存画像(孩子ID, 画像)
  5. 更新State: childProfile = 画像
  6. 返回: {childProfile: 画像, workflow: {nextNode: "周计划生成"}}
```

方式2: 选择性调用（智能决策流程）

适用场景：需要根据上下文智能决策、灵活选择工具的场景

实现机制：使用LLM的Function Call能力

将模块封装为工具，让LLM通过Function Call决定调用哪些工具：

```
实现步骤：
1. 将模块方法封装为工具定义（包含name、description、parameters schema）
2. 在节点函数中，将工具列表和任务描述传给LLM
3. LLM分析任务，返回需要调用的工具和参数（Function Call）
4. 节点函数执行LLM指定的工具调用
5. 将工具执行结果返回给LLM
6. LLM基于工具结果生成最终输出
7. 更新State
```

技术实现：

- 使用LangChain的`bind_tools()`方法绑定工具到LLM

- 或使用OpenAI/DeepSeek的原生Function Call API
- LangGraph会自动处理工具调用循环（LLM → 工具 → LLM）

典型场景：

- 对话助手节点：根据问题类型智能选择查询哪个知识库
- 周计划推荐节点：根据孩子情况智能选择检索策略
- 总结生成节点：根据数据完整性智能选择分析维度

示例: 对话助手节点（选择性调用）

```
对话助手节点函数:
  1. 从State读取: 用户问题, 孩子画像, 对话历史

  2. 定义工具（封装模块方法）:
     工具定义格式:
     {
       name: "search_methodology_knowledge",
       description: "检索地板时光方法论知识库。当用户询问地板时光理论、干预原则、专业术
语时使用。",
       parameters: {
         type: "object",
         properties: {
           query: {type: "string", description: "检索查询文本"},
           topK: {type: "number", description: "返回结果数量", default: 5}
         },
         required: ["query"]
       },
       function: async (query, topK) => {
         // 调用模块3的方法
         return await knowledgeRAG.searchMethodology(query, topK);
       }
     }

     类似地定义其他工具:
     - search_game_knowledge: 检索游戏知识库
     - search_scale_knowledge: 检索量表知识库
     - query_child_memory: 查询孩子Graphiti记忆
     - get_child_trends: 获取孩子趋势数据

  3. 构建提示词:
     system_prompt = """
     你是地板时光专家助手。

     用户信息:
     - 孩子年龄: {年龄}
     - 孩子情况: {画像摘要}

     你的任务是回答用户问题。根据问题类型，使用合适的工具获取信息:
     - 专业理论问题 → 使用 search_methodology_knowledge
     - 游戏相关问题 → 使用 search_game_knowledge
     - 孩子进步问题 → 使用 query_child_memory 和 get_child_trends
     - 量表评估问题 → 使用 search_scale_knowledge
```

```
   可以组合使用多个工具。基于工具返回的信息生成准确、个性化的回答。
   """

   user_message = 用户问题
```

4. 使用LangChain调用LLM with tools:
```
   from langchain_core.messages import HumanMessage, SystemMessage
   from langchain_openai import ChatOpenAI

   # 创建LLM实例
   llm = ChatOpenAI(model="deepseek-chat")

   # 绑定工具
   llm_with_tools = llm.bind_tools(tools)

   # 构建消息
   messages = [
     SystemMessage(content=system_prompt),
     ...对话历史...,
     HumanMessage(content=user_message)
   ]

   # 调用LLM（LLM会返回Function Call请求）
   response = await llm_with_tools.ainvoke(messages)

   # LangGraph自动处理工具调用循环：
   # - 如果response包含tool_calls，执行工具
   # - 将工具结果添加到messages
   # - 再次调用LLM
   # - 重复直到LLM返回最终答案（不再调用工具）
```

5. LLM的决策过程（自动）：
   - LLM分析问题："孩子最近眼神接触有进步吗？"
   - LLM决定调用工具：
     → tool_call_1: query_child_memory(childId, "眼神接触")
     → tool_call_2: get_child_trends(childId, "eye_contact")
   - 执行工具，获取结果
   - LLM基于结果生成回答："根据记录，孩子在过去两周..."

6. 另一个例子：
   - 用户问题："地板时光的核心原则是什么？"
   - LLM决定调用：
     → tool_call: search_methodology_knowledge("地板时光核心原则")
   - 执行工具，获取知识库内容
   - LLM基于知识库生成回答

7. 更新State:
   conversationHistory.append(user_message, assistant_response)

8. 返回：{answer: 回答, sources: 工具调用来源}

示例: 周计划推荐节点（选择性调用）

周计划推荐节点函数：
  1. 从State读取：孩子画像，当前趋势，上周完成情况

  2. 定义工具：
     {
       name: "search_games_by_dimension",
       description: "按目标维度检索游戏。用于查找能提升特定能力维度的游戏。",
       parameters: {
         dimension: {type: "string", description: "目标维度，如eye_contact"},
         difficulty: {type: "string", enum: ["easy", "medium", "hard"]},
         topK: {type: "number", default: 10}
       },
       function: async (dimension, difficulty, topK) => {
         return await knowledgeRAG.searchGamesByDimension(dimension,
difficulty, topK);
       }
     }

     类似地定义：
     - search_games_by_interest: 按兴趣检索游戏
     - get_similar_games: 获取相似游戏
     - check_game_history: 检查游戏历史（避免重复）

  3. 构建提示词：
     system_prompt = """
     你是游戏推荐专家。

     孩子情况：
     - 年龄：{年龄}
     - 需要提升的维度：{短板维度列表}
     - 兴趣：{兴趣列表}
     - 上周完成情况：{完成率、进步情况}

     任务：为本周生成7天游戏计划

     要求：
     1. 重点提升1-2个短板维度
     2. 结合孩子兴趣提高参与度
     3. 渐进式难度（周一简单 → 周日适度挑战）
     4. 同一游戏可重复2-3次巩固
     5. 避免与上周重复太多

     步骤：
     1. 使用工具检索候选游戏（按维度、兴趣、难度）
     2. 使用check_game_history排除最近玩过的
     3. 生成7天游戏序列（JSON格式）

     输出格式：
     {
       weeklyGoal: "本周总目标",
       dailyPlans: [
         {day: "周一", game: "游戏名", goal: "今日目标", reasoning: "为什么选这
个"},
         ...
       ]
     }

```
   """

4. 使用LangChain调用LLM with tools:
   llm_with_tools = llm.bind_tools(tools)
   response = await llm_with_tools.ainvoke(messages)

5. LLM的决策过程（自动）:
   - LLM分析: 孩子需要提升"eye_contact"和"two_way_communication"
   - LLM决定调用:
     → search_games_by_dimension("eye_contact", "easy", 10)
     → search_games_by_dimension("two_way_communication", "easy", 10)
     → search_games_by_interest("积木", 5)  // 如果孩子喜欢积木
   - 获取候选游戏列表
   - LLM再调用:
     → check_game_history(childId, game1.id)
     → check_game_history(childId, game2.id)
     ...
   - 排除最近玩过的
   - LLM基于候选游戏生成7天序列（考虑难度递进、重复练习）

6. 解析LLM输出的JSON

7. 调用模块1: 保存周计划到SQLite（强制性调用）

8. 更新State: currentWeeklyPlan

9. 返回: {weeklyPlan: 计划}
```

Function Call的优势:

- ✅ LLM智能决策：根据上下文选择最合适的工具组合
- ✅ 灵活性高：可以动态调整检索策略
- ✅ 可解释性：可以看到LLM调用了哪些工具
- ✅ 减少硬编码：不需要写复杂的if-else逻辑

Function Call的成本:

- ⚠️ 增加LLM调用次数（每次工具调用都需要LLM决策）
- ⚠️ 增加延迟（多轮LLM调用）
- ⚠️ 增加成本（更多token消耗）

因此选择性调用只用于真正需要智能决策的场景

如何选择调用方式

使用强制性调用的场景:

- ✅ 流程固定，每次都要执行相同的步骤

- ✅ 不需要智能决策，逻辑明确
- ✅ 追求速度和成本效率
- ✅ 需要严格控制流程

示例:

- 评估节点：必须解析报告 → 构建画像
- 记忆更新节点：必须写入Graphiti
- 视频分析节点：必须分析视频
- 游戏开始/结束节点：必须创建/结束会话

使用选择性调用的场景:

- ✅ 需要根据上下文智能决策
- ✅ 工具选择不确定，需要灵活组合
- ✅ 需要LLM理解和推理
- ✅ 追求智能化和灵活性

示例:

- 对话助手节点：根据问题类型选择知识库
- 周计划推荐节点：根据孩子情况选择检索策略
- 总结生成节点：根据数据完整性选择分析维度
- 再评估节点：根据进展情况选择分析重点

混合使用策略

同一个节点可以混合使用两种方式:

示例: 总结生成节点（混合）

```
总结生成节点函数:
  1. 强制性调用: 从SQLite获取会话数据（必须）
  2. 强制性调用: 从State读取历史数据（必须）

  3. 选择性调用: 让LLM决定分析哪些维度
     - 定义工具:
       - analyze_expression_changes() - 分析表情变化
       - analyze_eye_contact_trends() - 分析眼神趋势
       - analyze_interaction_quality() - 分析互动质量
       - compare_with_history() - 与历史对比

     - LLM根据数据完整性选择调用哪些分析工具
     - 例如: 如果有视频 → 调用所有工具
```

```
      - 例如：如果只有快捷记录 → 只调用部分工具

  4. 强制性调用：保存总结到SQLite（必须）

  5. 返回总结
```

工具封装方式

将模块封装为LangGraph工具：

每个业务模块可以提供多个工具函数：

```
模块3：知识库RAG
  提供的工具：
    - search_methodology_knowledge(query, topK)
    - search_game_knowledge(query, filters, topK)
    - search_scale_knowledge(query, topK)
    - search_games_by_dimension(dimension, difficulty, topK)
    - search_games_by_interest(interest, topK)

模块2：Graphiti记忆网络
  提供的工具：
    - query_child_memory(childId, query, timeRange)
    - get_child_trends(childId, dimension, timeRange)
    - get_milestones(childId, timeRange)
    - check_plateau(childId, dimension)

模块1：SQLite数据管理
  提供的工具：
    - get_child_profile(childId)
    - get_session_history(childId, filters)
    - check_game_history(childId, gameId)
```

工具定义包含：

- 工具名称
- 工具描述（让LLM理解何时使用）
- 参数schema
- 执行函数（调用实际的模块方法）

模块调用的具体形式

1. 强制性调用：模块以类或函数的形式提供接口

```
模块1: SQLite数据管理
  - 提供类: SQLiteManager
  - 方法: getChild(childId), saveSession(session), ...
  - 节点中直接调用: const child = await sqliteManager.getChild(childId)

模块2: Graphiti记忆网络
  - 提供类: GraphitiManager
  - 方法: addMemories(memories), analyzeTrends(childId), ...
  - 节点中直接调用: const trends = await
graphitiManager.analyzeTrends(childId)
```

2. 选择性调用：模块提供工具定义

```
模块3: 知识库RAG
  - 提供工具列表:
    [
      {
        name: "search_methodology_knowledge",
        description: "检索地板时光方法论知识库，用于回答专业理论问题",
        parameters: {query: string, topK: number},
        function: (query, topK) => knowledgeRAG.searchMethodology(query,
topK)
      },
      {
        name: "search_game_knowledge",
        description: "检索游戏知识库，用于查找游戏信息和玩法",
        parameters: {query: string, filters: object, topK: number},
        function: (query, filters, topK) => knowledgeRAG.searchGames(query,
filters, topK)
      },
      ...
    ]

  - 节点中使用:
    1. 将工具列表传给LLM
    2. LLM决定调用哪些工具
    3. 执行工具调用
```

5.7 完整工作流执行示例

场景: 完成一次游戏干预的完整流程

前置条件: 孩子已完成初始评估，本周计划已生成

执行流程:

```
步骤1: 前端触发"开始游戏"
  ↓
步骤2: 调用LangGraph工作流API
  - 传入: childId, gameId
  - 工作流从"游戏开始节点"启动
  ↓
步骤3: 游戏开始节点执行
  节点函数:
    1. 从State读取: childId, currentWeeklyPlan
    2. 调用模块1: sessionService.createSession(childId, gameId)
    3. 调用模块1: sessionService.startSession(sessionId)
    4. 更新State: currentSession = {sessionId, gameId, status:
"in_progress", ...}
    5. 返回: {currentSession, workflow: {nextNode: "游戏指引"}}
  ↓
步骤4: 游戏指引节点执行
  节点函数:
    1. 从State读取: gameId, childProfile
    2. 调用模块9: guidanceService.getStepGuidance(gameId, step=1, context)
    3. 调用模块5: speechService.textToSpeech(指引文字)
    4. 返回指引给前端（文字+语音）
    5. 等待前端通知"进入下一步"或"结束游戏"
  ↓
步骤5: 游戏进行中（前端循环）
  - 家长点击快捷按钮 → 调用API → 模块10.captureQuickObservation()
  - 家长按住说话 → 上传音频 → 调用API → 模块10.captureVoiceObservation()
  - 家长点击"下一步" → 返回步骤4获取下一步指引
  - 家长点击"结束游戏" → 进入步骤6
  ↓
步骤6: 游戏结束节点执行
  节点函数:
    1. 从State读取: sessionId
    2. 调用模块1: sessionService.endSession(sessionId)
    3. 更新State: currentSession.status = "ended", currentSession.endTime =
now
    4. 返回: {currentSession, workflow: {nextNode: "视频上传判断"}}
  ↓
步骤7: 条件边判断（是否上传视频）
  条件函数:
    1. 前端询问家长是否上传视频
    2. 如果上传 → 更新State.currentSession.hasVideo = true → 返回"视频分析节点"
    3. 如果跳过 → 更新State.currentSession.hasVideo = false → 返回"初步总结节点"

  假设家长选择上传视频 ↓

步骤8: 视频分析节点执行（后台异步）
  节点函数:
    1. 从State读取: sessionId, videoPath, quickObservations
    2. 调用模块11: videoAnalysisService.analyzeVideo(sessionId, videoPath)
       - 模块11内部调用模块4: AI视频解析
    3. 调用模块11: videoAnalysisService.crossValidate(quickObservations, 视频
结果)
    4. 更新State: currentSession.videoAnalysisResult = 结果
    5. 更新State: currentSession.validatedObservations = 已验证的观察
    6. 返回: {currentSession, workflow: {nextNode: "初步总结"}}
  ↓
```

```
步骤9: 初步总结节点执行
  节点函数:
    1. 从State读取: sessionId, currentSession, metrics, currentContext
    2. 调用模块12: summaryService.generatePreliminarySummary(sessionId)
       - 模块12内部从State读取历史数据（不再调用Graphiti）
    3. 更新State: currentSession.preliminarySummary = 初步总结
    4. 返回: {currentSession, workflow: {nextNode: "反馈表生成"}}
  ↓
步骤10: 反馈表生成节点执行
  节点函数:
    1. 从State读取: preliminarySummary
    2. 调用模块12: summaryService.generateFeedbackForm(preliminarySummary)
    3. 更新State: currentSession.feedbackForm = 反馈表
    4. 返回: {currentSession, workflow: {nextNode: "HITL暂停", isHITLPaused:
true}}
  ↓
步骤11: HITL暂停节点执行
  节点函数:
    1. 保存Checkpoint到SQLite
    2. 返回反馈表给前端
    3. 工作流暂停，等待家长填写
  ↓
步骤12: 家长填写反馈表（前端）
  - 家长填写3-5个问题
  - 点击"提交"
  - 前端调用恢复API，传入parentFeedback
  ↓
步骤13: 工作流恢复，最终总结节点执行
  节点函数:
    1. 从Checkpoint恢复State
    2. 更新State: currentSession.parentFeedback = 家长反馈
    3. 调用模块12: summaryService.generateFinalSummary(sessionId,
parentFeedback)
       - 模块12融合所有数据（观察、视频、反馈、State中的历史数据）
    4. 更新State: currentSession.finalSummary = 最终总结
    5. 返回: {currentSession, workflow: {nextNode: "记忆更新"}}
  ↓
步骤14: 记忆更新节点执行
  节点函数:
    1. 从State读取: childId, currentSession（包含所有数据）
    2. 调用模块13: memoryService.saveSessionMemory(childId, sessionData)
       - 模块13内部调用模块2: Graphiti批量写入
    3. 调用模块13: memoryService.buildContext(childId)
       - 重新构建上下文
    4. 更新State: currentContext = 新上下文, metrics = 新指标
    5. 返回: {currentContext, metrics, workflow: {nextNode: "再评估"}}
  ↓
步骤15: 再评估节点执行
  节点函数:
    1. 从State读取: childId, currentSession, currentContext, metrics
    2. 调用模块14: reassessmentService.reassessChild(childId, sessionData)
       - 模块14主要从State读取数据，只在必要时调用Graphiti
    3. 调用模块14: reassessmentService.detectMilestones(childId)
    4. 调用模块14: reassessmentService.updatePortrait(childId)
    5. 更新State: childProfile = 更新后的画像
    6. 判断是否需要调整计划
    7. 返回: {childProfile, needsAdjustment: false, workflow: {nextNode:
```

```
"END"}}
  ↓
步骤16: 条件边判断（是否需要调整计划）
  条件函数:
    1. 从State读取: needsAdjustment
    2. 如果 needsAdjustment == true → 返回"周计划生成节点"
    3. 如果 needsAdjustment == false → 返回"END"

  假设不需要调整 ↓

步骤17: 工作流结束
  - 返回最终State给前端
  - 前端展示最终总结
  - 前端更新日历（标记今日游戏完成✅）
```

关键观察点

1. **灵活选择调用方式**: 根据业务场景选择强制性或选择性调用
2. **节点函数职责清晰**: 编排逻辑在节点，业务逻辑在模块
3. **State是中心**: 所有数据通过State流转
4. **减少外部调用**: 优先从State读取，只在必要时调用外部系统
5. **批量操作**: Graphiti写入在记忆更新节点一次性完成
6. **HITL暂停**: 使用Checkpoint机制实现暂停和恢复
7. **智能与确定性结合**: 确定性流程用强制调用，需要决策的用选择性调用

实际应用建议

主工作流中的节点分类:

确定性节点（强制性调用）:

- 评估节点
- 游戏开始/结束节点
- 视频分析节点
- 记忆更新节点
- HITL暂停节点

智能决策节点（选择性调用）:

- 周计划推荐节点（智能选择游戏检索策略）
- 总结生成节点（智能选择分析维度）
- 再评估节点（智能选择评估重点）

独立智能节点（选择性调用）:

- 对话助手节点（智能选择知识库和查询策略）

5.8 独立流程

对话助手和可视化报告不在主工作流中，而是独立的API:

对话助手API

- 接收用户消息
- 调用模块15（对话助手）
- 返回回答

可视化报告API

- 接收查询参数（孩子ID、时间范围等）
- 调用模块16（可视化与报告）
- 返回可视化数据或PDF文件

---

6. 总结

6.0 知识库数据组织详细设计

数据库表结构

方法论知识表 (methodology_knowledge)

- id: 知识条目ID
- title: 标题
- content: 内容
- category: 分类（理论/原则/观察/指导/术语）
- tags: 标签（数组）
- embedding: 向量（pgvector）
- metadata: 元数据（JSON）

游戏知识表 (game_knowledge)

- id: 游戏ID
- name: 游戏名称
- description: 游戏描述
- age_range: 适用年龄范围
- target_dimensions: 目标维度（数组）
- difficulty_level: 难度等级（1-5）
- required_props: 所需道具（数组）
- steps: 玩法步骤（JSON数组）
- variations: 游戏变体（JSON数组）
- tips: 注意事项（数组）
- embedding: 向量（pgvector）
- metadata: 元数据（JSON）

量表知识表 (scale_knowledge)

- id: 量表条目ID
- scale_type: 量表类型（CARS-2/ADOS-2/ATEC）
- item_number: 题目编号
- question: 题目内容
- scoring_criteria: 评分标准（JSON）
- interpretation: 解读说明
- embedding: 向量（pgvector）
- metadata: 元数据（JSON）

案例知识表 (case_knowledge) - 可选

- id: 案例ID
- title: 案例标题
- description: 案例描述
- child_age: 孩子年龄
- problem_type: 问题类型
- intervention: 干预方法
- outcome: 干预结果
- lessons: 经验教训
- embedding: 向量（pgvector）
- metadata: 元数据（JSON）

检索策略详解

1. 单层检索

- 指定知识层级（如只检索游戏知识库）
- 使用向量相似度排序
- 可添加过滤条件（如年龄、难度、维度）

2. 跨层检索

- 同时检索多个知识层级
- 对不同层级的结果进行加权融合
- 根据问题类型动态调整权重

3. 混合检索

- 向量检索（语义相似度）
- 关键词过滤（精确匹配）
- 元数据过滤（年龄、难度等）
- 多种策略组合

4. 上下文增强检索

- 结合孩子画像（优势、短板、兴趣）
- 结合当前目标（本周要提升的维度）
- 结合历史数据（之前玩过的游戏）
- 动态调整检索结果

知识库初始化

方法论知识库（约100-200条）

- 地板时光核心文档
- 专业书籍章节
- 专家指导手册
- 常见问题解答

游戏知识库（MVP阶段50个游戏）

- 按年龄分组（1-2岁、2-3岁、3-4岁、4-5岁）
- 按维度分组（6大维度各8-10个游戏）
- 按难度分级（简单、中等、困难）

量表知识库（3套量表）

- CARS-2完整题目和评分标准
- ADOS-2完整题目和评分标准
- ATEC完整题目和评分标准

知识库更新策略

静态知识（方法论、量表）

- 初始化时一次性导入
- 定期人工审核和更新
- 版本控制

动态知识（游戏、案例）

- 初始化时导入基础游戏
- 后期可持续添加新游戏
- 可根据使用反馈优化游戏描述

6.1 模块化优势

1. **清晰的职责划分**: 每个模块职责单一，边界清晰
2. **解耦合**: 模块间通过接口通信，依赖关系明确
3. **可并行开发**: 同层模块可以并行开发，加快进度
4. **易于测试**: 每个模块可以独立测试
5. **易于维护**: 模块边界清晰，便于后续迭代
6. **易于集成**: 所有模块最终集成到LangGraph框架

6.2 开发建议（面向接口）

1. **接口先行**: 第1周就定义好所有接口，不要边开发边定义
2. **Mock快速验证**: 用Mock快速跑通流程，验证架构设计
3. **渐进式替换**: 一个个替换真实实现，不要一次性全换
4. **持续集成**: 每替换一个模块就测试，确保系统可运行
5. **灵活配置**: 通过环境变量控制使用Mock还是真实
6. **并行开发**: 框架组和模块组可以并行工作

6.3 风险控制（面向接口）

1. **技术验证**: 对Graphiti、视频分析等新技术提前验证（可以先Mock）
2. **随时回退**: 如果真实实现有问题，可以立即回退到Mock
3. **定期Review**: 每周Review进度，检查Mock替换情况
4. **及时沟通**: 接口定义需要团队共识，及时沟通调整

6.4 关键设计原则（面向接口）

1. **依赖接口不依赖实现**: LangGraph节点只调用接口
2. **依赖注入**: 通过容器管理服务实例
3. **配置驱动**: 通过配置切换Mock/真实
4. **Mock数据真实**: Mock返回的数据结构要和真实一致
5. **接口稳定**: 接口一旦定义，尽量不要修改

文档结束