快速搭建一个基于 react 框架的 agent

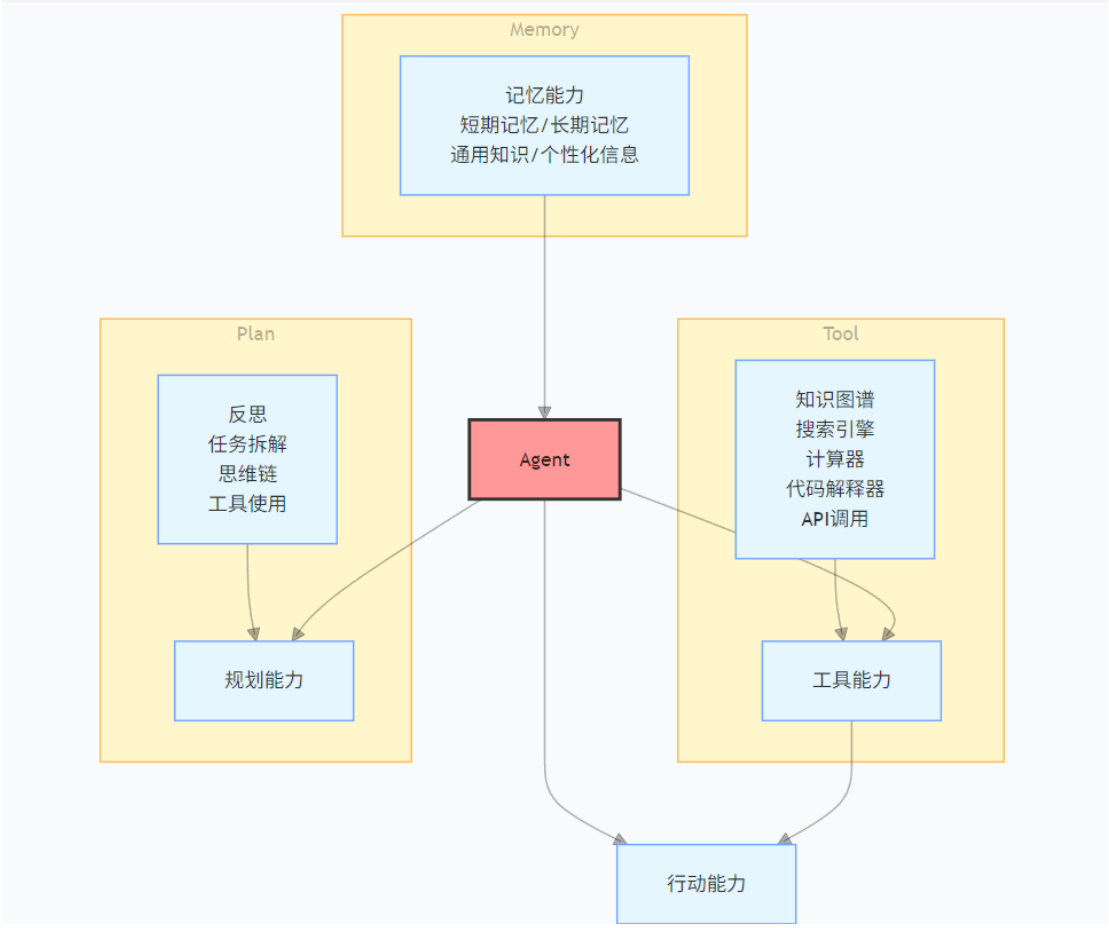
1. 背景

这个文档不仅提供了一些基础知识 agent 和不同框架，同时搭建了一个简易版的 agent 进行代码审查，基于 react 框架进行。

1.1 agent 架构

LLM：扮演了 Agent 的“大脑”， 是 Agents 的核心，承担着存储知识、记忆、信息处理和决策等能力。

- 规划：提供规划能力，帮助 Agents 将复杂问题分解规划为更小、更简单的子任务，并逐个解决。同时，Agents 可以对过去的行为进行自我批判和反思，从错误中吸取经验，使 Agents 能够修正以往的决策、纠正之前的失误，从而不断优化其性能。
- 记忆：对 Agents 系统而言，记忆可以定义为用于获取、存储、保留以及随后检索信息的过程。它可以利用记录下来的记忆来促进未来的行动。记忆模块可以帮助 Agents 积累经验、自我进化，并以更一致、合理和有效的方式行动。
- 工具：Agents 可以学习如何调用外部 API，以获取模型权重中缺少的额外信息，这些信息通常在预训练后很难更改，包括当前信息、代码执行能力、专有信息源的访问等。



1.2 经典多 agent 架构

1. 静态编排：根据任务场景编写固定拓扑结构的协作模式，一般需要开发者自己编写代码或者借助编排框架实现。

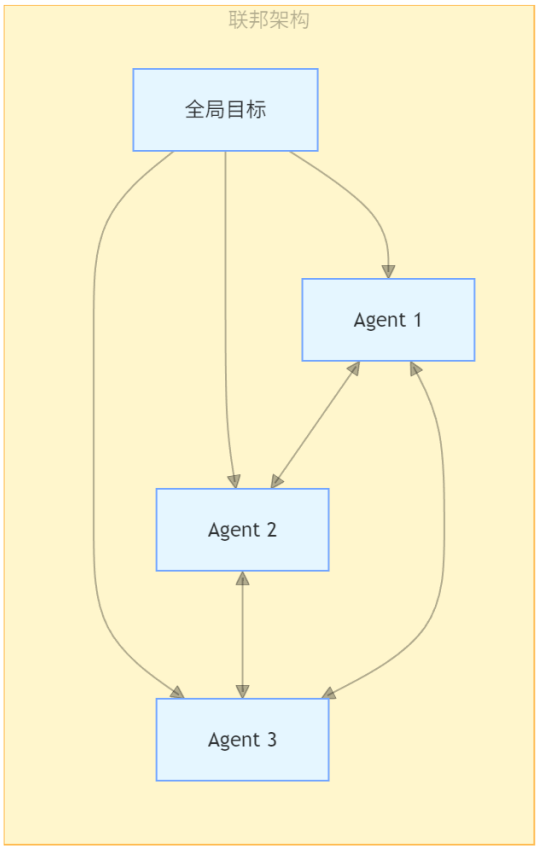
2. 动态编排：根据任务场景由大模型动态组建多 Agent 团队，在处理任务过程中可以根据当前状态动态调整 Agent 协作模式，进行自主协作。

在协作的静态编排中，包含图编排、角色编排和任务编排三种主流方式。如 LangGraph 采用了图编排、AutoGen 采用角色编排、CrewAI 采用任务编排方式。

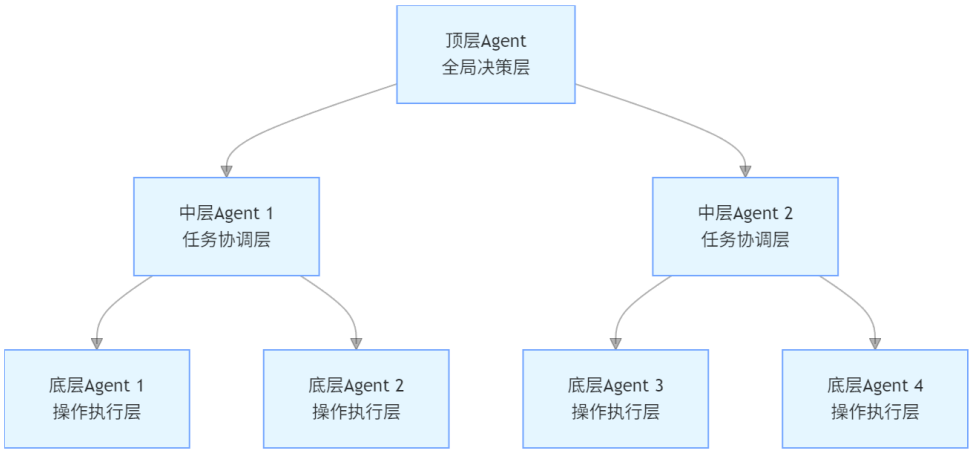
方面	图编排	角色编排	任务编排
结构方式	有向图	角色定义	任务分解
协作模式	流程驱动	对话驱动	任务驱动
复杂度	高	中	中
可视化	很好	一般	好
扩展性	中等	很好	好
适用场景	复杂流程	团队协作	项目管理

经典的架构形式：

- **联邦架构：Agent 平等协作，遵循统一目标，无严格层级**



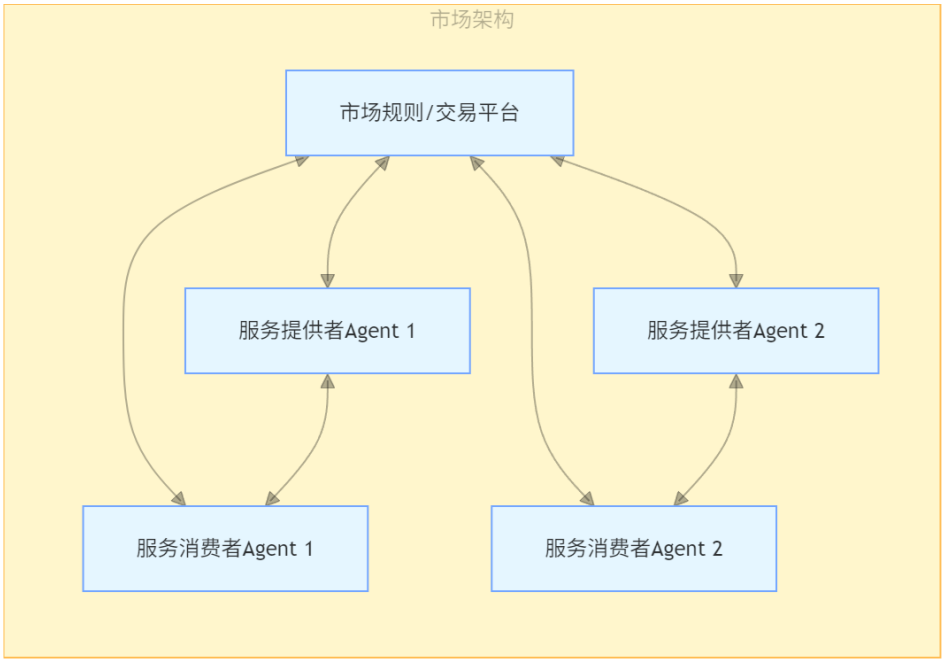
- 结构组成：
  - 包含一个「全局目标（G）」，作为整个系统的统一方向。
  - 三个平等的智能体（Agent 1、Agent 2、Agent 3），构成协作集群。
- 协作逻辑：
  - 全局目标向所有 Agent 传递，指导它们的行动方向（箭头 `G --> A/B/C` 表示）。
  - 各 Agent 之间是平等的双向通信关系（箭头 `<-->` 表示），没有层级从属，通过相互协作共同推进全局目标的实现。
- **分层架构：自上而下的层级控制，上层决策、下层执行**



- 层级划分：
  - 顶层 Agent（全局决策层）：处于架构最上层，负责制定整体目标、全局策略和高层决策，不直接参与具体执行。
  - 中层 Agent（任务协调层）：接收顶层指令，将全局任务拆解为子任务，协调底层 Agent 的工作，是连接决策与执行的中间环节（图中包含 2 个中层 Agent，分别对应不同的任务分支）。
  - 底层 Agent（操作执行层）：处于架构最下层，直接执行具体操作，接收并服从中层 Agent 的指令（图中 4 个底层 Agent 分别隶属于 2 个中层 Agent，分工执行不同子任务）。
- 协作逻辑：

信息和指令通过箭头单向传递（`-->`），呈现 “顶层→中层→底层” 的层级控制关系，下层 Agent 只向上层对应的 Agent 汇报或接收指令，层级间职责明确、分工清晰。

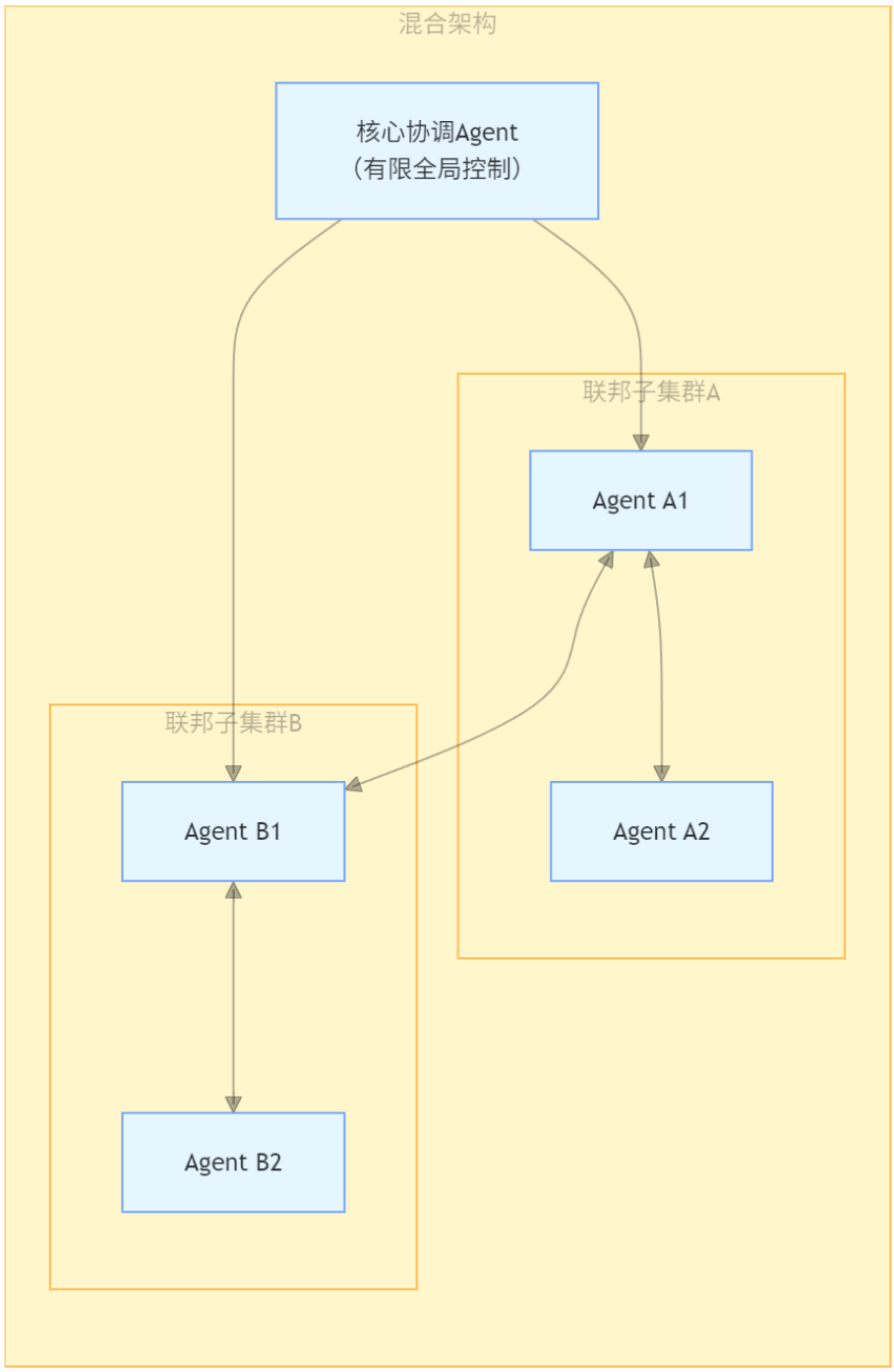
- **市场架构：基于交易 / 规则的分布式协作，类似供需市场**



- 核心组成：
  - 市场规则 / 交易平台（M）：处于架构中心，是整个市场的 “规则制定者” 和 “交易枢纽”，负责定义交易规则、协调供需匹配、保障交易公平性（类似现实中的电商平台、股票交易所等）。
  - 服务提供者 Agent（P1、P2）：代表市场中的 “供给方”，提供商品、服务或资源，需遵守市场规则并通过平台与消费者对接。
  - 服务消费者 Agent（C1、C2）：代表市场中的 “需求方”，提出需求、获取服务，同样需遵守市场规则并通过平台选择提供者。
- 协作逻辑：
  - 所有参与者（提供者和消费者）都与「市场规则 / 交易平台」存在双向交互（`<-->`）：平台向参与者传递规则，参与者向平台反馈信息（如供给、需求、交易结果等）。
  - 提供者与消费者之间也存在直接的双向交互（`<-->`）：基于平台规则完成具体交易（如协商价格、交付服务、反馈评价等）。

这种架构强调去中心化的自主决策和基于规则的自由协作，适合需要多主体自主交易、资源优化配置的场景（如共享经济、供应链协作、分布式服务交易等），通过 “平台 + 供需方” 的模式实现动态平衡。

- **混合架构：核心 Agent 协调 + 子集群联邦协作的组合模式**



1.3 langchain 核心模块

代理 (Agents)：

工具（Tools）：语言模型与其他资源的交互方式。

代理（Agents）：驱动决策的语言模型或者多模态模型。

工具包（Toolkits）：一组工具，当它们一起使用时可以完成特定的任务。

代理执行器（Agent Executor）：负责运行带有工具的代理的逻辑。

- 作用：智能决策和执行的核心组件
- 功能：理解用户意图、选择工具、协调执行流程
- 特点：可以自主决定使用哪些工具来完成任务

链 (Chains)

链（Chain）是对多个独立组件进行端到端封装的一种方式。

- 作用：将多个组件连接成执行序列
- 功能：定义固定的工作流程，按顺序执行各个步骤
- 特点：适合有明确步骤的任务，如代码分析流程

索引链：实现对自身文档的问答：1.Stuffing（填充）• 使用 StuffDocumentsChain，将所有数据直接作为上下文添加到提示中 • 只需一次 LLM 调用，可一次性访问所有数据 • 受上下文长度限制，只适用于小数据片段 2. Map Reduce（映射-归约）• 使用 MapReduceDocumentsChain，对每个数据块运行提示后组合输出 • 可扩展到大文档，支持并行化处理 • 需要多次 LLM 调用，组合时可能丢失信息 3. Refine（优化）• 逐文档优化输出，保持更多相关上下文 • 比 MapReduce 损失信息更少 • 无法并行化，对文档顺序敏感

选择建议：小文档用 Stuffing，大文档需并行用 Map Reduce，需保持上下文用 Refine。

LLM 链：LLM 链是 LangChain 中最基础的组件，它将语言模型(LLM)与提示词(Prompt)连接起来，形成可重复使用的执行单元。

提示选择器：为不同的模型选择一共 prompt

索引 (Indexes)

- 作用：管理和检索文档、知识库
- 功能：文档存储、向量化、相似性搜索
- 特点：支持 RAG（检索增强生成）应用

原始文档 → 文档加载 → 文本拆分 → 向量化 → 存储 → 检索查询

↓ ↓ ↓ ↓ ↓ ↓

各种格式 统一格式 小片段 向量表示 向量库 相关结果

内存 (Memory)

内存（Memory）是在对话过程中存储和检索数据的概念。主要有两种方法：

1. 根据输入，获取任何相关的数据。
2. 根据输入和输出，相应地更新状态。

内存主要分为两种类型：短期内存和长期内存。

短期内存通常指的是如何在单个对话的上下文中传递数据（通常是先前的聊天消息或其摘要）。

长期内存处理的是如何在对话之间获取和更新信息的问题。

纪录聊天历史：

目前，与语言模型的主要接口是通过聊天界面进行的。ChatMessageHistory 类负责记录所有先前的聊天互动。然后，可以直接将它们传递回模型，以某种方式进行总结，或者进行某种组合。

ChatMessageHistory 提供了两个方法和一个属性。它提供的两个方法是 add_user_message 和 add_ai_message，用于分别存储来自用户的消息和 AI 的响应。它提供的属性是 messages 属性，用于访问所有先前的消息。

模型 (Models)

- 作用：提供 AI 能力的核心组件
- 功能：语言模型接口、多模态模型支持
- 特点：抽象化设计，支持多种 AI 服务商

三种模型：1. LLMs：文本到文本的转换 2. 聊天模型：消息到消息的对话 3. 嵌入模型：文本到向量的转换

提示 (Prompt)

- 作用：管理和优化 AI 的输入提示
- 功能：提示模板、动态提示生成、提示优化
- 特点：提高 AI 响应的质量和一致性

现在编写模型的新方法是通过提示。 "提示" 指的是模型的输入。这个输入通常不是硬编码的，而是通常由多个组件构成的。PromptTemplate 负责构建这个输入。LangChain 提供了多个类和函数，使构建和使用提示更加容易。

本文档部分分为四个部分：

PromptValue：表示模型输入的类。

Prompt Templates:负责构建 PromptValue 的类。

示例选择器 Example Selectors:在提示中包含示例通常是有用的。这些示例可以硬编码，但如果它们是动态选择的，则通常更有用。

输出解析器 Output Parsers:语言模型（和聊天模型）输出文本。但是许多时候，您可能想获得比仅文本更有结构化的信息。这就是输出解析器发挥作用的地方。输出解析器负责（1）指示模型如何格式化输出，（2）将输出解析为所需格式（包括必要时进行重试）。

模式 (Schema)

- 作用：定义数据结构和接口规范
- 功能：消息格式、工具输入输出、数据类型定义
- 特点：确保组件间数据交换的一致性

1.4 langgragh 核心模块

1. 定义状态模型

状态是图的核心，定义了应用运行时的数据结构。

```
Python
from typing import List, Dict
from pydantic import BaseModel

class ChatState(BaseModel):
    messages: List[Dict[str, str]] = []
    current_input: str = ""
    tools_output: Dict[str, str] = {}
    final_response: str = ""
```

2. 创建处理节点

节点是图的基本单元，每个节点执行特定的任务。

节点通常是 Python 函数,用于处理状态并返回更新后的状态:

```
Python
async def process_input(state: ChatState) -> ChatState:
    # 处理用户输入
    messages = state.messages + [{"role": "user", "content":
state.current_input}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
```

```
        tools_output=state.tools_output
    )

async def generate_response(state: ChatState) -> ChatState:
    # 使用 LLM 生成回复
    response = await llm.ainvoke(state.messages)
    messages = state.messages + [{"role": "assistant", "content":
response}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
        tools_output=state.tools_output,
        final_response=response
    )
```

3. 边

边定义了节点之间的连接关系和路由逻辑:

```
Python
from typing import List, Dict
from pydantic import BaseModel

class ChatState(BaseModel):
    messages: List[Dict[str, str]] = []
    current_input: str = ""
    tools_output: Dict[str, str] = {}
    final_response: str = ""
```

4. 搭建一个简单的聊天机器人

```
Python
from typing import List, Dict, Tuple
from pydantic import BaseModel
from langgraph.graph import StateGraph, END
from langchain_core.language_models import ChatOpenAI

# 1. 定义状态 class ChatState(BaseModel):
    messages: List[Dict[str, str]] = []
    current_input: str = ""
    should_continue: bool = True

# 2. 定义节点函数
async def process_user_input(state: ChatState) -> ChatState:
```

```
    """处理用户输入"""
    messages = state.messages + [{"role": "user", "content":
state.current_input}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
        should_continue=True
    )

async def generate_ai_response(state: ChatState) -> ChatState:
    """生成 AI 回复"""
    llm = ChatOpenAI(temperature=0.7)
    response = await llm.ainvoke(state.messages)
    messages = state.messages + [{"role": "assistant", "content":
response}]
    return ChatState(
        messages=messages,
        current_input=state.current_input,
        should_continue=True
    )

def should_continue(state: ChatState) -> str:
    """决定是否继续对话"""if "goodbye" in
state.current_input.lower():
        return "end"return "continue"# 3. 构建图
workflow = StateGraph(ChatState)

# 添加节点
workflow.add_node("process_input", process_user_input)
workflow.add_node("generate_response", generate_ai_response)

# 添加边
workflow.add_edge("process_input", "generate_response")
workflow.add_conditional_edges(
    "generate_response",
    should_continue,
    {
        "continue": "process_input",
        "end": END
    }
)

# 4. 编译图
app = workflow.compile()
```

```
# 5. 运行对话
async def chat():
    state = ChatState()
    while True:
        user_input = input("You: ")
        state.current_input = user_input
        state = await app.ainvoke(state)
        print("Bot:", state.messages[-1]["content"])
        if not state.should_continue:
            break# 运行聊天 import asyncio
asyncio.run(chat())
```

1.5 autogen 框架

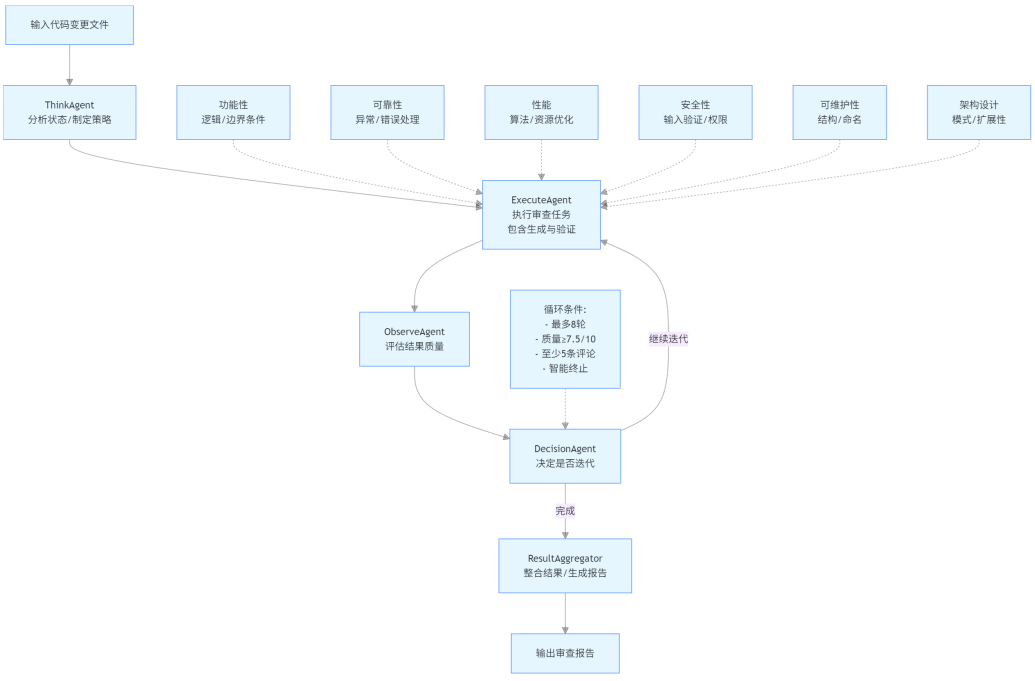
<后续更新>

2. 代码审查 agent <基于 react 框架的 agent>

2.1 基于 langchain 框架

代码后续会进一步完善上传到 github 上，目前是一个不带工具的 react 框架的简易版实现。

系统采用 5 个专门 Agent 协同工作:



架构解析：

1. LLM 包装器 (24-88 行)

class AlibabaLLM:

"""适配器模式 - 将阿里云 API 适配为 LangChain 兼容接口"""

1. 配置管理: 从 config 模块获取参数，支持动态覆盖
2. API 调用: 封装阿里云通义千问 API 调用逻辑
3. 消息转换: LangChain 消息格式 → 阿里云 API 格式
4. 错误处理: 完整的异常捕获和降级处理
5. 响应封装: 动态创建 Response 对象模拟 LangChain 接口

2. 状态管理和思考 Agent (91-137 行)

class CodeReviewState(TypedDict):

"""状态驱动架构 - 将整个审查流程建模为状态机"""

核心数据流:

- diff_content → generated_reviews → validated_reviews → final_report

循环控制字段:

- iteration_count: 当前迭代轮次
- max_iterations: 最大迭代限制
- quality_threshold: 质量阈值 (停止条件)
- improvement_needed: 是否需要继续改进

元数据字段:

- current_step: 当前执行步骤 (用于流程跟踪)
- strategy_notes: 策略笔记 (记录决策过程)
- previous_results: 历史结果 (迭代改进参考)

ThinkAgent 设计模式

策略模式: 根据当前状态动态制定审查策略

职责分离: 专门负责"思考"阶段，不执行具体任务

graph TD

A[构建状态摘要] --> B[生成分析提示词]

B --> C[调用 LLM 分析]

C --> D[解析策略响应]

D --> E[更新状态信息]

3. ExecuteAgent - 执行引擎 (208-274 行)

```
Python
  class ExecuteAgent:
      """组合模式 - 整合 ReviewGenerationAgent +
ReviewValidationAgent"""

      def __init__(self, llm):
          self.review_generator = ReviewGenerationAgent(llm)  # 生
成器
          self.review_validator = ReviewValidationAgent(llm)  # 验
证器
```

条件分支逻辑:

```
Plain Text
  if state['iteration_count'] == 1 or not
state.get('generated_reviews'):
      # 首轮生成
      state = self.review_generator.generate_reviews(state)
  else:
      # 迭代改进
      state = self._improve_reviews(state)
```

统一验证

```
Plain Text
  state = self.review_validator.validate_reviews(state)
```

4. 评论改进机制 (243-274 行)

改进策略分析:

1. 类型分布统计: 分析前轮评论的问题类型分布
2. 平衡性优化: 识别过度集中的问题类型
3. 覆盖度提升: 补充缺失的问题维度
4. 质量提升: 提高评论深度和具体性

动态提示词生成:

- 基于历史数据生成个性化改进提示
- 针对性地指导 LLM 进行优化

5. 动态提示词策略和 ObserveAgent (275-334 行)

策略模式的高级应用:

临时替换系统提示词

```
original_prompt = self.review_generator.system_prompt
self.review_generator.system_prompt = improvement_prompt
```

执行改进生成

```
state = self.review_generator.generate_reviews(state)
```

恢复原始提示词

```
self.review_generator.system_prompt = original_prompt
```

设计亮点:

- 非侵入式: 不修改原始 Agent，只临时注入定制化提示词
- 上下文感知: 基于历史数据生成针对性改进指令
- 状态恢复: 确保 Agent 状态的一致性

6. ObserveAgent - 质量评估专家 (295-334 行)

职责分工:

- 质量评估: 多维度评估当前轮次结果
- 迭代建议: 基于评估结果决定是否继续
- 改进指导: 为下一轮提供具体改进方向

评估维度:

1. 评论质量: 准确性、实用性、覆盖度
2. 问题识别: 新问题发现能力
3. 改进程度: 相比上轮的提升幅度
4. 迭代价值: 继续投入的性价比

智能决策规则:

- 评论数量 < 5 条 → 继续迭代

- 质量评分 < 7 分 → 继续迭代
- 问题类型单一 → 继续迭代

7. DecisionAgent - 智能决策引擎 (560-607 行)

决策树结构:

迭代决策

├── 硬限制检查

│ ├── 最大迭代次数 (max_iterations)

│ └── 质量阈值达成 (quality_threshold)

├── 强制继续条件

│ ├── 评论数量不足 (<5 条)

│ └── 质量评分过低 (<7.0)

├── 多样性检查

│ └── 问题类型单一 (<3 种)

└── 防护机制

└── 无限循环保护 (>=5 轮 && >=6.0 分)

量化决策标准:

- 覆盖度指标: 评论数量 ≥ 5 条
- 质量指标: 评分 ≥ 7.0/10
- 多样性指标: 问题类型 ≥ 3 种
- 效率指标: 迭代轮次限制

容错设计:

- 降级策略: 5 轮后强制停止，避免无限循环
- 质量兜底: 6.0 分作为可接受下限
- 透明化: 每个决策都有详细日志输出

8. ReviewGenerationAgent - 核心评论生成器 (609-634 行)

分析维度设计:

1. 功能性: 逻辑正确性、边界条件
2. 可靠性: 异常处理、资源管理

3. 性能: 算法效率、瓶颈识别
4. 安全性: 输入验证、权限检查
5. 可维护性: 代码结构、命名规范
6. 架构设计: 设计模式、扩展性

深度分析要求:

- 表面问题 → 设计缺陷分析
- 局部代码 → 系统级影响评估
- 当前功能 → 可扩展性考虑
- 开发阶段 → 生产风险识别

9. ReviewValidationAgent 和 ResultAggregator (975-1051 行)

1. 分类聚合: 按问题类型分组 (_group_reviews_by_type)
2. 优先级排序: 按严重程度排序 (_sort_reviews_by_severity)
3. 统计分析: 生成数值化指标
4. 报告生成: 构建结构化最终报告

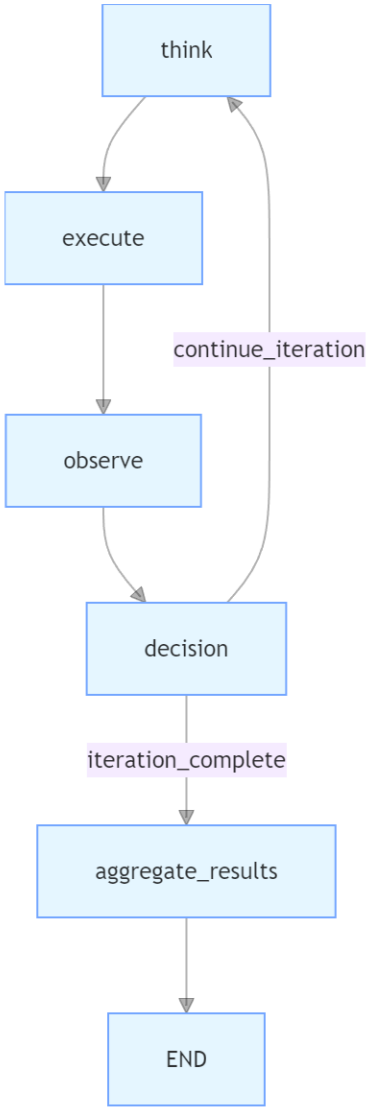
10. 工作流编排核心 (1132-1229 行)

LangGraph 工作流架构:

```
Plain Text
  def create_code_review_workflow():
      # 1. Agent 实例化
      think_agent = ThinkAgent(llm)
      execute_agent = ExecuteAgent(llm)
      observe_agent = ObserveAgent(llm)
      decision_agent = DecisionAgent()
      result_aggregator = ResultAggregator()
      # 2. 状态图构建
      workflow = StateGraph(CodeReviewState)
      # 3. 节点映射
      workflow.add_node("think", think_agent.think_and_plan)
      workflow.add_node("execute",
execute_agent.execute_review_tasks)
      workflow.add_node("observe",
```

```
observe_agent.observe_and_evaluate)
      workflow.add_node("decision", decision_agent.make_decision)
      workflow.add_node("aggregate_results",
result_aggregator.aggregate_results)
```

流程图结构:



条件边设计:

```
Python
  workflow.add_conditional_edges(
      "decision",
      lambda state: "think" if state['current_step'] ==
'continue_iteration' else
  "aggregate_results",
```

```
    {
        "think": "think",                    # 继续循环
        "aggregate_results": "aggregate_results"  # 结束循环
    }
)
```

2.2 核心代码

[code_review_system.py]