项目申请书

项目名称	基于LiteFlow框架开发大模型插件以实现类Dify的后端智能体应用表达式编排能力
项目导师	铂赛东
申请人	陈金阳
日期	2025.6.09
邮箱	2307984361@qq.com

一、项目背景

1. 项目描述

LiteFlow是国内知名的Java规则引擎，目前被应用在数千家企业核心业务侧，社区人数众多，活跃度高。

目前LiteFlow被广泛应用在逻辑编排场景，拥有灵活的DSL规则表示式以及支持度广泛的脚本语言，实用性和特性都非常适合业务复杂的应用侧。

随着AI大模型的兴起，社区很多开发者把LiteFlow当做一个“后端Dify”来构造自己的智能体逻辑流。并且已经有相当一部分的社区同学已经成功落地了。但是LiteFlow并没有从框架层面对大模型有任何支持，LiteFlow目前提供的组件是极其开放的，大模型实现的部分还需要开发者自己去实现的。

所以这次的课题的目标就是为LiteFlow打造一个AI大模型支持的插件，开发者不用自己再去对接大模型，能够直接引用AI插件已定义好的大模型的各种组件加上LiteFlow原有的强大的编排能力形成一个智能体逻辑流程。

这次课题需要涉及当下大模型中的大部分知识块，包括但不仅限于对接主流平台的API，RAG，FunctionCall，MCP等相关概念。

2. 项目需求

    a. 为LiteFlow开发liteflow-ai插件模块

i. ：提供大模型的组件，这些组件需要对接主流平台以及本地Ollama

ii. ：需要在编排层面对这些大模型组件进行输入输出定义的设计

iii. ：需要从设计上支持多模态输入，流式输出，Function Call等的包装支持

b. 为大模型插件提供完善的测试用例工程liteflow-ai-test模块，需要对每一个细分功能点，场景进行支持

c. 为开发好的liteflow-ai插件创建演示工程liteflow-ai-example工程，综合利用liteflow-ai开发的演示工程。

3. 项目地址

https://gitee.com/dromara/liteFlow

二、源码理解

1. 规则解析

LiteFlow的规则解析核心是**基于QLExpress表达式引擎的EL解析系统**。

代码块

```
/**
 * EL解析引擎
 */
public final static ExpressRunner EXPRESS_RUNNER = new ExpressRunner();
```

解析器在静态初始化时注册了所有流程控制操作符，包括THEN、WHEN、IF、FOR、WHILE等，每个操作符对应特定的流程控制结构。

代码块

```
static {
    // 初始化QLExpress的Runner
    EXPRESS_RUNNER.addFunction(ChainConstant.THEN, new ThenOperator());
    EXPRESS_RUNNER.addFunction(ChainConstant.WHEN, new WhenOperator());
    // ...
}
```

解析过程采用两段式构建模式，这样设计是为了解决Chain间的循环依赖问题。

1. 第一阶段：先注册所有 chain 到 FlowBus 中

代码块

```
// 先在元数据里放上chain
for (Element e : chainList) {
    // 校验加载的 chainName 是否有重复的
    // TODO 这里是否有个问题，当混合格式加载的时候，2个同名的Chain在不同的文件里，就不行了
    String chainId = Optional.ofNullable(e.attributeValue(ID)).orElse(e.attributeValue(NAME));
    // 检查 chainName
    checkChainId(chainId, e.getText());
    if (!chainIdSet.add(chainId)) {
        throw new ChainDuplicateException(StrUtil.format("[chain name duplicate] chainName={}", chainId));
    }
    // 如果是禁用，就不解析了
    if (!getEnableByElement(e)) {
        continue;
    }

    FlowBus.addChain(chainId);
    if(ElRegexUtil.isAbstractChain(e.getText())){
        abstratChainMap.put(chainId,e);
        //如果是抽象chain，则向其中添加一个AbstractCondition,用于标记这个chain为抽象chain
        Chain chain = FlowBus.getChain(chainId);
        chain.getConditionList().add(new AbstractCondition());
    }
};
```

2. 第二阶段：解析每个 chain 的具体内容

代码块

```
// 解析每一个chain
for (Document document : documentList) {
    Element rootElement = document.getRootElement();
    List<Element> chainList = rootElement.elements(CHAIN);
    for(Element chain:chainList){
        // 如果是禁用，就不解析了
        if (!getEnableByElement(chain)) {
            continue;
        }

        //首先需要对继承自抽象Chain的chain进行字符串替换
```

```
            parseImplChain(abstratChainMap, implChainSet, chain);
            //如果一个chain不为抽象chain，则进行解析
            String chainName =
    Optional.ofNullable(chain.attributeValue(ID)).orElse(chain.attributeValue(NAME)
    );
            if(!abstratChainMap.containsKey(chainName)){
                parseOneChainConsumer.accept(chain);
            }
        }
    }
```

系统支持延迟解析模式，当配置为PARSE_ONE_ON_FIRST_EXEC时，Chain在首次执行时才进行解析，提高启动性能。

2. 脚本解析

脚本解析采用**SPI机制的工厂模式**。ScriptExecutorFactory作为工厂类，通过ServiceLoader动态加载不同语言的脚本执行器。

对于大部分脚本语言，LiteFlow提供了JSR223ScriptExecutor统一抽象实现，支持Groovy、JavaScript等符合JSR-223规范的脚本引擎。

脚本在加载时会进行预编译并缓存，避免运行时重复编译的性能开销。

3. 脚本执行

脚本执行通过ScriptExecutor的execute方法实现，执行前会绑定上下文参数，包括自定义上下文、元数据、用户定义的Bean等。

JSR223实现中，使用预编译的CompiledScript对象执行，通过Bindings传递参数，确保执行效率。

执行过程支持完整的生命周期管理，包括beforeProcess、afterProcess、onSuccess、onError等钩子方法。

4. 存储插件热更新(以 SQL 为例)

SQL存储插件的热更新是LiteFlow最精巧的设计之一，其核心原理是**基于SHA1值比对的轮询检测机制**。

1. **初始化阶段**：SQL解析器启动时注册轮询任务，分别监听Chain和Script的变更。

2. **变更检测原理**：AbstractSqlReadPollTask维护一个SHA值映射表，存储每个规则/脚本的内容哈希值。轮询时重新计算SHA1值，通过比对检测是否发生变更。

3. **热更新执行：**检测到变更后，系统计算集合差集确定新增、修改、删除的元素，然后分别调用doSave和doDelete方法。

4. **规则更新实现：**对于Chain的变更，ChainReadPollTask通过LiteFlowChainELBuilder重新构建Chain，对于删除操作则调用FlowBus.removeChain移除。

5. **轮询调度：**JDBCHelper通过ScheduledThreadPoolExecutor定时执行轮询任务，轮询间隔可配置。

三、项目需求分析

1. 开箱即用的 AI 能力封装

LF（LogicFlow）需要结合大模型能力，提供开箱即用的 AI 组件，开发者无需编写集成代码即可直接使用。这要求平台提前封装好调用大模型的组件插件，并简化开发者接入门槛。

2. 大模型调用能力封装与 Client 选型

需要选型确定与大模型交互的客户端工具（如 Spring AI、LangChain4j 等），基于选定工具封装好与大模型平台的交互插件。封装时主要关注：

- 如何传参
- 如何管理会话
- 如何处理上下文
- 暂不考虑本地向量化、向量数据库、记忆存储等内容，先聚焦对接大模型平台已有能力。

3. 大模型平台对接与能力边界设定

需要明确对接哪些大模型平台（如 OpenAI、Moonshot、阿里通义、文心一言等），并思考如何兼顾不同平台的 API 适配与能力整合。强调“骨架打结实”，不追求大而全，关注可扩展性。

4. AI 组件使用方式设计（后端 Dify 模式）

AI 组件需支持像 Dify 那样的使用方式，开发者在流程编排中可以直接拖拽使用、填写参数，无需开发者自定义代码。重点是：

- 如何声明组件可配置参数
- 如何在配置界面中展示、验证这些参数
- 如何实现统一的组件调用入口

5. AI 组件出参结构设计与上下文绑定

出参结构要支持通过上下文机制在多个组件之间传递。需明确：

- 出参结构如何统一（如 result、text、choices）
- 如何用表达式引用上一个组件的结果
- 是否需限制引用范围
- 如何在 DSL 中支持“引用AI组件字段作为入参”

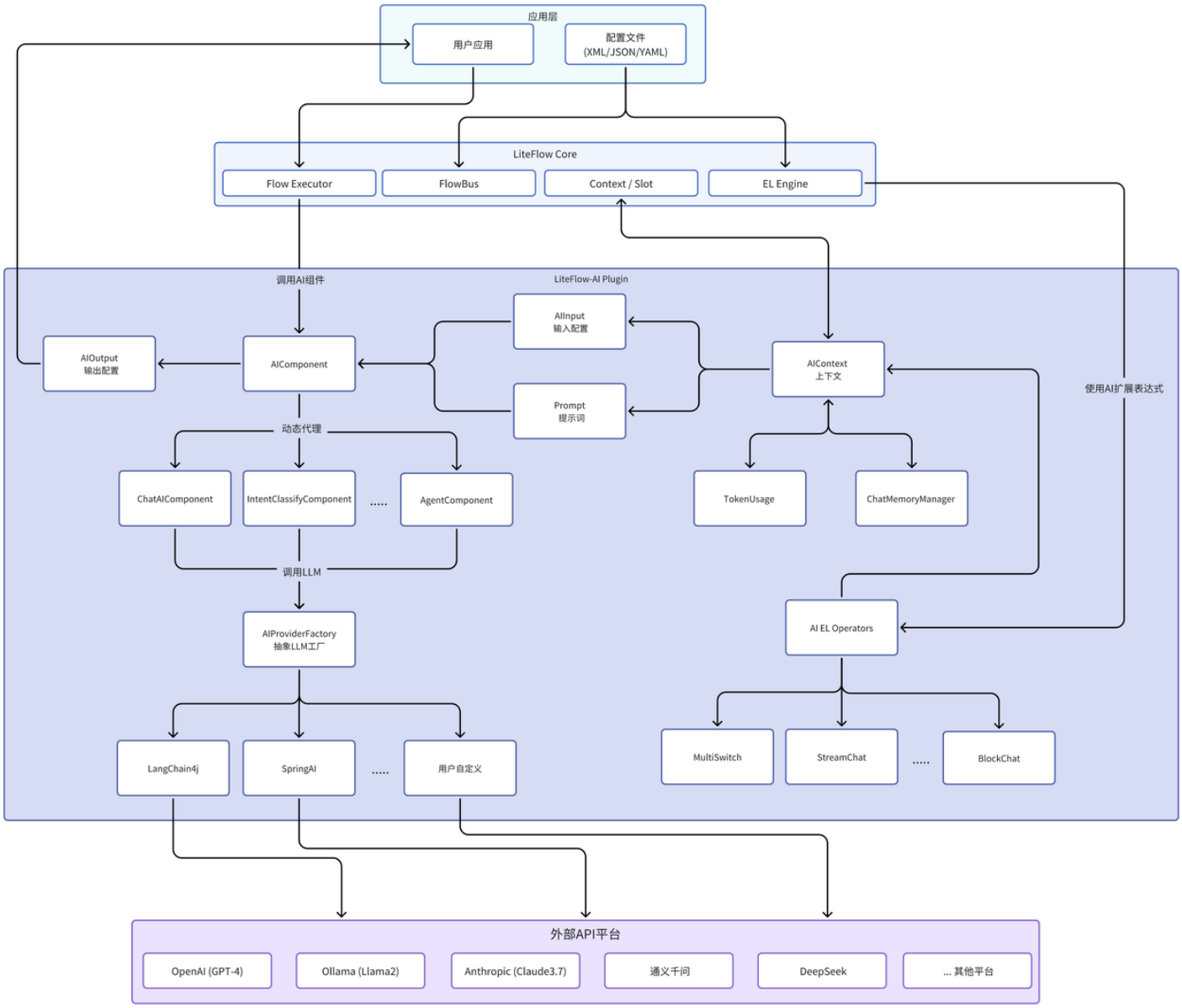
6. 编排语言扩展与表达式增强

评估现有流程表达式是否支持大模型场景，比如：

- 意图分类组件需要表达多选条件分支（目前 SWITCH 只支持单选）
- 多路分支是否支持动态生成
- 是否需要支持新的表达式语法，如多选SWITCH、文本相似度分支等

四、项目实现方案

1. 整体架构



2. AI参数配置

全局配置定义在 `application.yaml / application.properties` 中 (此处参考 langchain4j 大模型属性配置)

代码块

```yaml
liteflow:
  ai:
    global:
      default-provider: openai
      timeout: 30s
```

```yaml
    retry-count: 3
    providers:
      openai:
        api-key: ${OPENAI_API_KEY}
        base-url: https://api.openai.com/v1
        organization: ${OPENAI_ORG}
        models:
          chat: gpt-4o-mini
          embedding: text-embedding-3-small
          vision: gpt-4-vision
        default-params:
          temperature: 0.7
          max-tokens: 2000
          log-requests: false
          log-responses: false
          timeout: 60s
          max-retries: 3
          frequency-penalty: 0
          presence-penalty: 0
          top-p: 1
```

---

3. 大模型交互工厂

在构建 LiteFlow-AI 插件时，一个核心挑战在于如何优雅地集成和管理不同的大模型提供商。每个提供商（如 OpenAI, Ollama, Azure AI 等）都有其独特的 API 和配置方式。为了实现插件的灵活性、可扩展性和易用性，设计了一套基于抽象工厂模式的 LLM 提供商管理机制。

3.1 设计思路与目标

目标是创建一个统一的接入层，使得 LiteFlow AI 组件无需关心底层大模型的具体实现细节，可以透明地切换和使用不同提供商的服务。这需要：

1. **统一的接口**：定义一个标准的 AI 服务契约。
2. **配置驱动**：能够通过外部配置来指定和参数化 AI 提供商。
3. **动态加载与扩展**：方便未来集成新的 AI 提供商，而无需修改核心代码。
4. **生命周期管理**：有效管理提供商实例的创建、初始化和销毁。

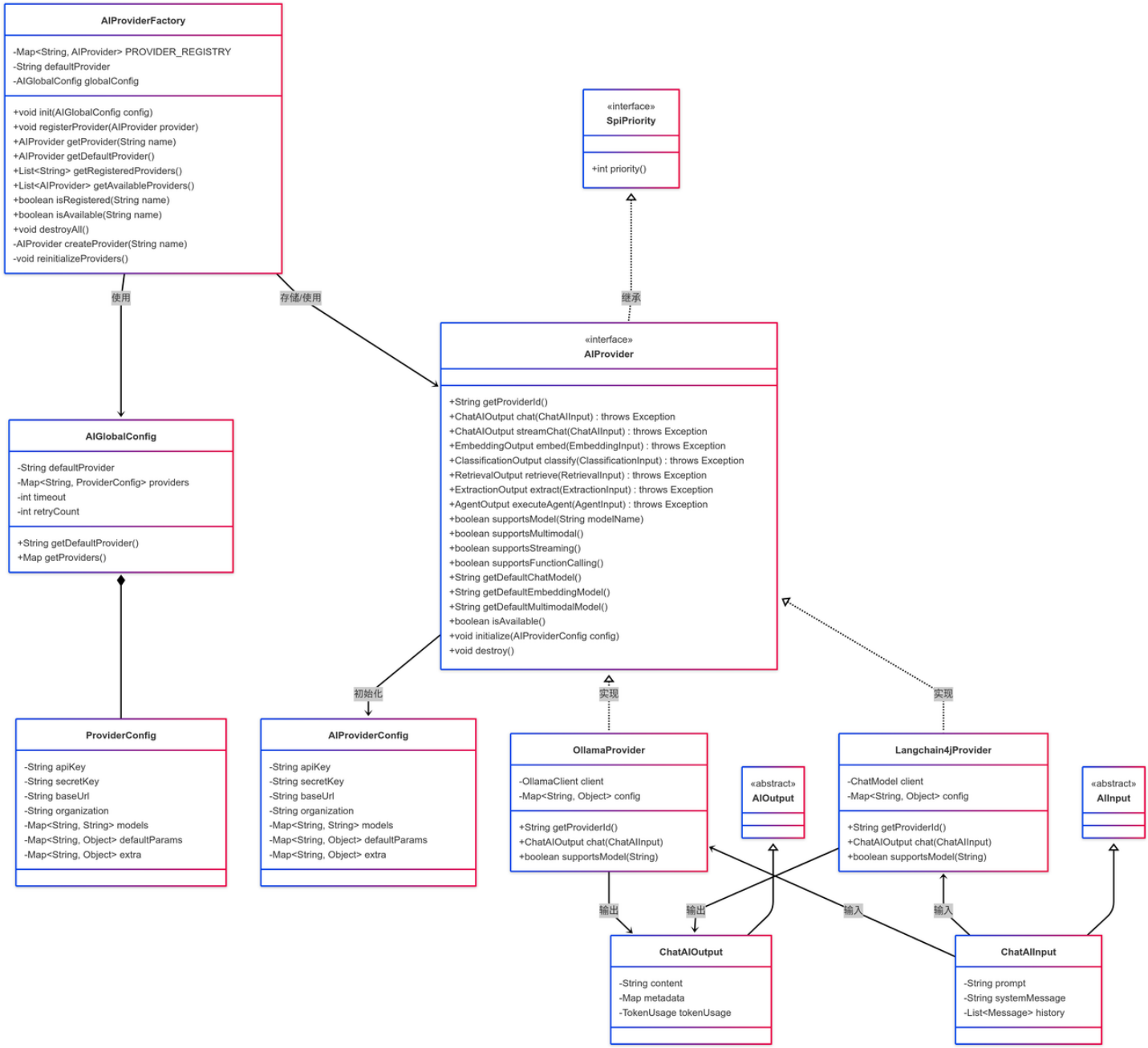
3.2 核心组件设计

- **`AIProvider` 接口：**
  - 定义 `chat()`，`embed()`，`classify()` 等标准 AI 操作方法。
  - 定义能力检测方法（`supportsStreaming()`，`supportsFunctionCalling()`）和生命周期管理方法（`initialize()`，`destroy()`，`isAvailable()`）。
  - 实现 SPI 机制，通过继承 `SpiPriority` 接口（LiteFlow 的 SPI 机制），可以实现提供商的自动发现和优先级排序

- **`AIProviderFactory` 抽象工厂类：**
  - 负责 `AIProvider` 实例的创建、缓存和管理。
  - 通过 `init(AIGlobalConfig)` 方法进行初始化，加载全局配置。
  - `registerProvider()` 方法允许动态或通过 SPI 注册提供商。
  - `getProvider(String name)` 和 `getDefaultProvider()` 方法供应用层获取具体的提供商实例。
  - 通过维护注册表 `PROVIDER_REGISTRY` 实现获取对应的 `AIProvider`。

- **具体提供商实现 (`Langchain4jProvider`，`OllamaProvider`)：**
  - 这些类是 `AIProvider` 接口的具体实现。例如，`Langchain4jProvider` 会封装 Langchain4j 客户端库，适配其 API 调用到 `AIProvider` 接口。`OllamaProvider` 则会直接与 Ollama 服务进行交互。
  - 它们在 `initialize()` 方法中接收 `AIProviderConfig`，并据此设置内部的客户端或参数。
  - 它们负责将 LiteFlow AI 的标准输入（如 `ChatAIInput`）转换为对应平台的请求格式，并将平台的响应转换为标准的输出（如 `ChatAIOutput`）。

3.3 架构图



4. AI组件设计

4.1 节点设计

本次项目 AI 节点定义初步设计为基础的 3 个，优先实现 LLM 强相关的的节点。

节点名称	节点功能	节点输入	节点输出
LLM (大语言模型)	与LLM交互（OpenAI/Anthropic等），	• **provider**: AI 服务商 • **model**: 模型名称	• **text**: 生成的文本内容

	支持聊天/补全/上下文/记忆/视觉/流式/结构化输出	• **systemPrompt**: 系统提示词 • **userPrompt**: 用户提示词(支持{{变量}}占位符) • **temperature**: 温度值(0.0-2.0) • **maxTokens**: 最大 Token 数 • **streaming**: 是否流式输出 • **multiModal**: 多模态支持 • **enableFunctionCalls**: • Function Call 支持	• **usage**: Token 使用统计(prompt/completion/total) • **finishReason**: 结束原因(stop/length/function_call 等) • **functionCalls**: Function 调用结果列表 • **metadata**: 扩展元数据 • **isStreaming**: 是否为流式输出 • **sessionId**: 会话标识
Knowledge Retrieval (知识检索)	通过RAG从数据集检索知识，作为LLM上下文补充	• **embeddingModel**: 嵌入模型名称 • **query**: 查询语句(支持{{变量}}占位符) • **vectorStore**: 向量数据库类型 • **collection**: 检索集合名称 • **topK**: 返回结果数量(默认5) • **scoreThreshold**: 相似度阈值(0.0-1.0) • **metadataFilter**: 元数据过滤器 • **rerank**: 是否重排序	• **documents**: 检索文档列表(id/content/metadata/score/source/summary) • **ragContext**: 格式化的 RAG 上下文 • **sources**: 文档来源列表 • **scores**: 文档相似度得分映射 • **metadata**: 检索元数据
Question Classifier (问题分类)	利用LLM进行文本分类（意图识别）	• **provider**: AI 服务商 • **model**: 模型名称 • **systemPrompt**: 系统提示词 • **userPrompt**: 用户提示词(支持{{变量}}占位符) • **categories**: 预定义分类列表 • **multiLabel**: 多标签支持 • **confidenceThreshold**: 置信度阈值(0.0-1.0) • **temperature**: 温度值	• **results**: 所有分类结果(category/confidence/metadata/reasoning) • **primaryCategory**: 主要分类 • **qualifiedCategories**: 满足阈值的分类集合 • **maxConfidence**: 最高置信度 • **avgConfidence**: 平均置信度

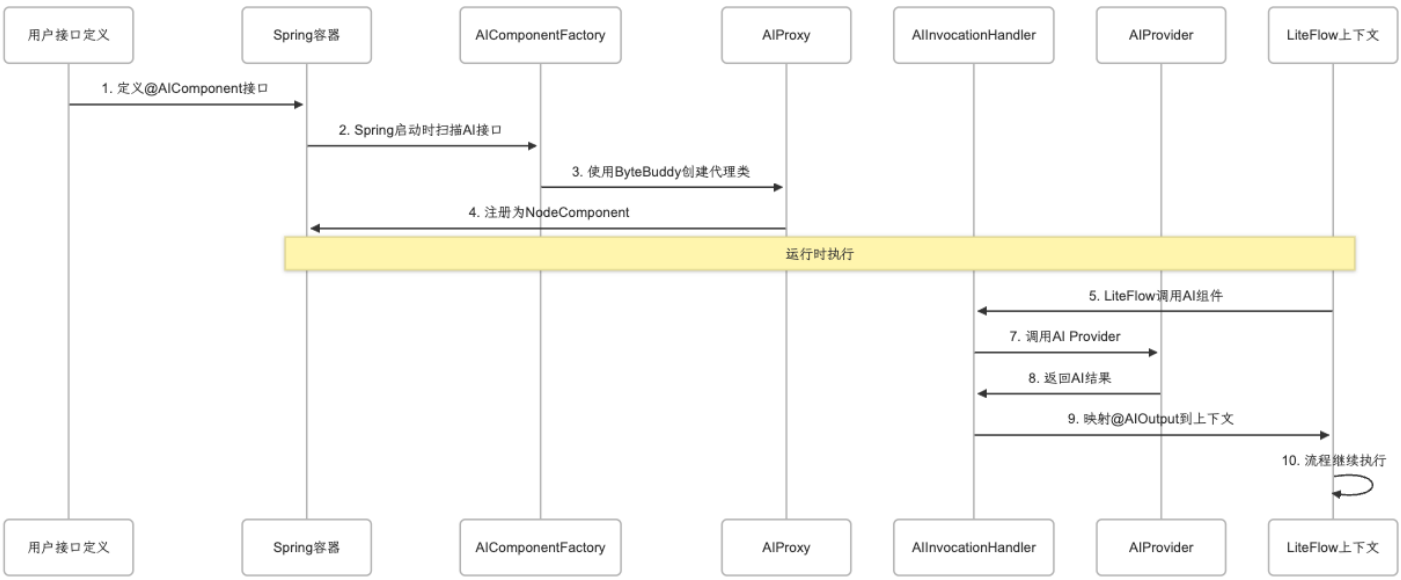
接下来讨论如何将组件整合进入 LiteFlow 框架

4.2 基于 Annotation 与动态代理实现开箱即用的 AI 节点

基本理念是实现开箱即用的大模型集成能力。核心思想是基于Java注解（Annotation）和动态代理技术，让开发者通过简单的接口定义和注解配置即可创建功能完备的AI节点，无需编写具体实现类。

架构设计分为四个主要层次：

1. 接口定义层：开发者通过带注解的接口定义AI组件
2. 代理生成层：运行时通过ByteBuddy动态代理技术生成实现类
3. 调用处理层：解析注解、处理参数映射、调用底层AI服务
4. AI服务层：封装与各大模型平台的交互逻辑



4.3 核心注解

注解设计上以层次化结构组织，采用 `@AIComponent` 作为顶层组件注解来标识 AI 功能节点，并通过一系列专用子注解来细化不同 AI 能力的实现方式：

- `@AIComponent`：组件级注解，用于定义类作为 AI 节点的基本身份和属性
- 下设多种能力实现注解：
  - `@AIChat`：类级注解，用于定义具备对话生成能力的组件
  - `@AIClassify`：类级注解，用于定义文本或数据分类能力的组件
  - `@AIRetrieval`：类级注解，用于定义知识库检索能力的组件
- 参数处理注解：
  - `@AIInput`：类级注解，用于定义 AI 能力输入的映射规则
  - `@AIOutput`：类级注解，用于定义 AI 能力输出结果的映射规则

代码块

```
@AIComponent           // 组件级注解，定义AI节点身份
  |
  +-- @AIChat          // 类级注解，定义聊天/生成能力
  |
  +-- @AIClassify      // 类级注解，定义分类能力
  |
  +-- @AIRetrieval     // 类级注解，定义知识检索能力
  |
  +-- @AIInput        // 类级注解，定义输入参数映射
  |
  +-- @AIOutput        // 类级注解，定义输出参数映射
```

4.4 标识接口设计模式

所有 AI 组件都采用**标识接口**模式设计，接口本身不定义任何方法，仅作为组件标识和注解载体。动态代理会为这些接口生成对应的 `NodeComponent` 实现类。

以下是三种 AI 节点的标识接口设计示例：

4.4.1 AIChat 节点标识接口

输入定义

代码块

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AIChat {
    String provider() default ""; // AI 服务商
    String model(); // AI 模型名称
    double temperature() default 0.7; // 温度值
    int maxTokens() default 2000; // 最大 Token 数
    String systemPrompt() default "你是一个对话助手"; // 系统提示词
    String userPrompt(); // 用户提示词(支持内联文本和外部资源文件，如 "classpath:prompts/user.md")
    boolean streaming() default false; // 是否启用流式响应
    boolean multiModal() default false; // 是否启用多模态支持
    boolean enableFunctionCalls() default false; // 是否启用 Function Call
    boolean enableMCP() default false; // 是否启用 MCP
}
```

输出定义

代码块

```
public class ChatAIResult {
    private String text;                        // 生成的文本
    private AIUsage usage;                      // Token使用情况
    private String finishReason;                // 结束原因
    private List<FunctionCall> functionCalls; // Function调用结果
    private Map<String, Object> metadata;     // 扩展元数据
    private boolean isStreaming;                // 是否为流式输出
    private String sessionId;                   // 会话ID
}
```

使用示例

代码块

```
// 方式1：简单文本Prompt
@AIChat(
    model = "gpt-4",
    systemPrompt = "你是一个友好的AI助手",
    userPrompt = "用户问题：{{question}}，历史对话：{{history}}",
    temperature = 0.7,
    maxTokens = 2000
)
@AIInput(mapping = {
```

```
    @InputField(name = "question", expression = "context.userQuestion", defaultValue = "你好"),
    @InputField(name = "history", expression = "context.chatHistory", defaultValue = "[]")
})
@AIOutput(
    type = OutputType.TEXT,
    methodExpress = "setData",
    key = "context.response"
)
public interface ChatBotAI {
    // 标识接口，无需定义方法
}

// 方式2：外部资源文件Prompt（适用于长文本场景）
@AIChat(
    model = "gpt-4",
    systemPrompt = "classpath:prompts/chat-system.md",
    userPrompt = "classpath:prompts/chat-user.md",
    temperature = 0.7,
    maxTokens = 2000
)
@AIInput(mapping = {
    @InputField(name = "question", expression = "context.userQuestion"),
    @InputField(name = "history", expression = "context.chatHistory"),
    @InputField(name = "userProfile", expression = "context.userProfile")
})
@AIOutput(
    type = OutputType.TEXT,
    methodExpress = "setData",
    key = "context.response"
)
public interface AdvancedChatBotAI {
    // 标识接口，无需定义方法
}
```

4.4.2 AIClassify 节点标识接口

输入定义

代码块

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AIClassify {
```

```
    String provider() default ""; // AI 服务商
    String model(); // AI 模型名称
    double temperature() default 0.7; // 温度值
    int maxTokens() default 2000; // 最大 Token 数
    String systemPrompt() default "你是一个意图分类助手"; // 系统提示词
    String userPrompt(); // 用户提示词(支持内联文本和外部资源文件，如 "classpath:prompts/user.md")
    String[] categories(); // 预定义分类类别列表(nodeId)
    boolean multiLabel() default true; // 是否支持多标签 (switch or multi-switch)
    // 置信度阈值，高于此阈值认为有效，范围在 0.0-1.0 之间
    double confidenceThreshold() default 0.6;
}
```

输出定义

代码块

```
public class ClassifyAIResult {
    private List<CategoryResult> results;     // 所有分类结果
    private String primaryCategory;           // 主要分类
    private Set<String> qualifiedCategories; // 满足阈值的分类集合
    private double maxConfidence;             // 最高置信度
}

public class CategoryResult {
    private String category;
    private double confidence;
    private Map<String, Object> metadata;
}
```

使用示例

代码块

```
@AIComponent(value = "intentClassifier", name = "意图分类器", type = AITypeEnum.CLASSIFY)
@AIClassify(
    model = "gpt-4",
    systemPrompt = "你是一个意图分类专家",
    userPrompt = "请分析用户意图: {{userText}}",
    categories = {"booking", "inquiry", "complaint", "compliment"},
    multiLabel = true,
    confidenceThreshold = 0.6
)
@AIInput(mapping = {
```

```
    @InputField(name = "userText", expression = "context.inputText", required = true)
})
@AIOutput(
    type = OutputType.JSON,
    methodExpress = "setData",
    key = "context.intentResult",
    entity = ClassifyAIResult.class
)
public interface IntentClassifierAI {
    // 标识接口，无需定义方法
}
```

4.5.3 AIRetrieval 节点标识接口

输入定义

代码块

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AIRetrieval {
    String provider() default ""; // 服务商
    String embeddingModel(); // 嵌入模型
    String query(); // 用户的查询语句，支持占位符动态输入
    String vectorStore(); // 指定向量数据库的类型或名称
    String collection(); // 指定在向量数据库中进行检索的集合(Collection)名称
    int topK() default 5; // 检索结果返回的最高数量(Top K)
    double scoreThreshold() default 0.7; // 相似得分阈值
    String metadataFilter() default ""; // 元数据过滤器
    boolean rerank() default true; // 是否进行重排序
}
```

输出定义

代码块

```
public class RetrievalAIResult {
    private List<Document> documents;         // 检索到的文档
    private String ragContext;                // 格式化的RAG上下文
    private List<String> sources;             // 文档来源列表
    private Map<String, Double> scores;       // 相似度得分
    private RetrievalMetadata metadata;       // 检索元数据
}
```

```
8
9   public class Document {
10      private String id;
11      private String content;
12      private Map<String, Object> metadata;
13      private double score;
14  }
```

使用示例

代码块

```
1   @AIComponent(value = "knowledgeRetriever", name = "知识检索器", type =
    AITypeEnum.RETRIEVAL)
2   @AIRetrieval(
3       embeddingModel = "text-embedding-3-small",
4       query = "{{searchQuery}}",
5       vectorStore = "pinecone",
6       collection = "knowledge_base",
7       topK = 5,
8       scoreThreshold = 0.7,
9       rerank = true
10  )
11  @AIInput(mapping = {
12      @InputField(name = "searchQuery", expression = "context.query", required =
    true)
13  })
14  @AIOutput(
15      type = OutputType.JSON,
16      methodExpress = "setData",
17      key = "context.retrievalResult",
18      entity = RetrievalAIResult.class
19  )
20  public interface KnowledgeRetrieverAI {
21      // 标识接口，无需定义方法
22  }
```

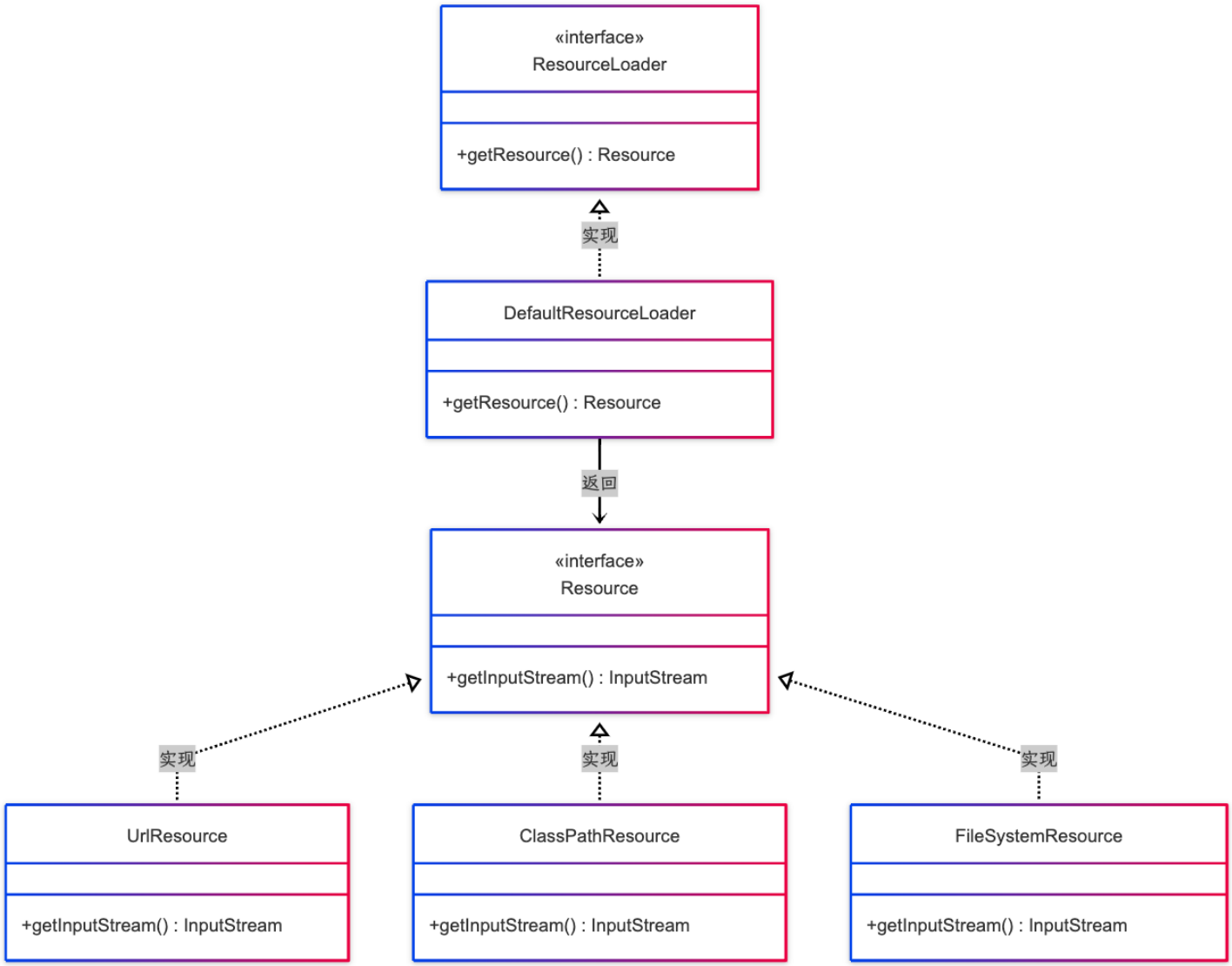
4.5 长文本 Prompt

为了解决节点 `prompt` 编写的格式信息保持问题以及可读性和可维护性较差的问题，我其实是受到了 Spring 资源管理极致的启发，设计了一套长文本提示管理方案。

该方案的核心思想是将提示视为一种可外部化的资源，通过统一的加载和解析机制进行管理和使用，具体设计如下：

1. **内联文本与外部资源统一：**通过前缀解析的方式将内联文本与外部资源文件统一定义。
   a. **内联文本：**前缀以 `text:` 开头或者不加前缀
   b. **外部资源文件：**必须添加前缀，分别为 `classpath:` ，`file:` ，`url:` 。
2. **统一的提示资源抽象：** 借鉴Spring框架对不同来源资源（如XML配置文件）的统一抽象思路，我们将提示定义为一种资源。支持从以下三种标准位置加载提示资源：
   - **ClassPath资源 (`ClassPathResource`)：** 允许将提示文件打包在应用程序的类路径下。用户通过添加 `classpath:` 前缀来指定，例如 `classpath:prompts/my_prompt.md` 。
   - **文件系统资源 (`FileSystemResource`)：** 允许从服务器的本地文件系统加载提示文件，例如 `file:/opt/app/prompts/my_prompt.txt` 。
   - **URL资源 (`UrlResource`)：** 允许通过URL从网络位置（如HTTP/HTTPS服务器、FTP服务器等）加载提示文件，例如 `url:https://example.com/prompts/shared_prompt.txt` 。
3. **内容格式的完整保留与解析：**
   - **格式保真：** 从外部资源文件加载提示时，系统将完整保留文件原始内容的所有格式信息，包括但不限于Markdown标记、代码块、列表、多级缩进以及换行符。这确保了提示的结构和意图能够准确无误地传递给大型语言模型。
   - **统一的变量替换机制：** 无论是内联定义的提示（直接在配置或代码中以字符串形式存在）还是从外部资源文件加载的提示，均支持统一的变量占位符机制。占位符采用 `{{变量名}}` 的形式（例如 `{{userName}}` ， `{{productDetails}}` ）。在实际使用提示前，系统会根据传入的上下文参数动态替换这些占位符。

以下是相关类图



4.6 基于注解的出入参绑定设计

为实现灵活、直观的AI组件参数绑定，我们采用基于注解的声明式方案，主要通过 `@AIInput` 和 `@AIOutput` 实现参数与上下文的映射关系。

4.6.1 入参绑定方案

1. 注解式节点参数绑定：

代码块

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

```java
public @interface AIInput {
    InputField[] mapping() default {}; // 定义所有输入字段的映射
}

@Retention(RetentionPolicy.RUNTIME)
public @interface InputField {
    String name();                          // 字段名（必需）
    String expression();                    // 上下文路径（必需）
    String defaultValue() default "";       // 默认值
    boolean required() default false;       // 是否必需
}
```

2. 模板占位符：

在提示词模板中使用双花括号语法 `{{变量名}}` 引用上下文变量：

```
代码块
"你是一个代码阅读助手，请分析以下代码：{{code.method}}"
```

3. 绑定优先级：

a. 第一优先级：使用 `@InputField` 中的 `expression` 映射在上下文中进行查找

b. 第二优先级：使用 `{{变量名}}` 中的变量名直接在上下文中进行查找

c. 第三优先级：如果上述都未查找到有效值，使用 `@InputField` 中的 `defaultValue`

4. 集成LiteFlow上下文机制：

复用LiteFlow已有的 `LiteflowContextRegexMatcher#searchContext()` 机制进行上下文变量解析，实现与现有框架的无缝集成。

4.6.2 出参绑定方案

1. 返回值映射：

通过 `@AIOutput` 注解将方法返回值映射到上下文中：

```
代码块
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
```

```java
public @interface AIOutput {
    OutputType type() default OutputType.TEXT;  // TEXT 或 JSON
    String methodExpress() default "setData"; // 输出映射
    String key();  // 输出对象映射
    Class<?> entity() default Object.class;  // JSON输出时的实体类定义
    OutputField[] fields() default {};  // 多字段映射
}

public enum OutputType {
    TEXT,    // 纯文本输出
    JSON     // 结构化JSON输出
}

@interface OutputField {
    String name();                    // 字段名
    String methodExpress() default "setData";          // 输出映射
    String key();                     // 输出对象映射
}
```

2. 多值输出：

支持返回复杂对象，并通过注解将其各字段映射至不同上下文变量：

代码块

```java
// 定义输出实体类
public class AnalysisResult {
    private String summary;
    private Double confidence;
    private List<String> categories;
    private Map<String, Object> metadata;
}

@AIOutput(
    type = OutputType.JSON,
    entity = AnalysisResult.class,
    methodExpress = "setResult",
    key = "context.result",
    fields = {
        @OutputField(name = "summary", methodExpress = "setSummary", key = "context.summary"),
        @OutputField(name = "confidence", methodExpress = "setConfidence", key = "context.confidence"),
        @OutputField(name = "categories", methodExpress = "setCategories", key = "context.categories")
```

```
18        }
19    )
```

3. 集成LiteFlow上下文机制：

复用LiteFlow已有的 `LiteflowContextRegexMatcher#searchAndSetContext()` 机制进行上下文变量赋值，实现与现有框架的无缝集成。

4.7 流式输出

4.7.1 问题分析

当前 LiteFlow 的 FlowExecutor 采用同步执行模式，在实现流式输出场景时面临以下关键技术挑战：

1. **同步阻塞限制**：现有的 execute2Future 方法虽然提供异步执行能力，但 Future 接口的结果获取机制仍然是同步阻塞的，这使得主线程无法及时响应流式输出需求。
2. **回调处理缺失**：在需要实现"流式输出 + 异步回调"的复合场景时（如向前端推送实时数据的同时，在流程完成后执行清理工作、发送完成通知等业务逻辑），传统的 Future 模式无法提供优雅的异步回调支持。

假设用户需要直接返回一个 SseEmitter 到前端，同时需要在流程结束的时候处理相关的业务逻辑，那么因为 Future 接口会阻塞主线程，就会导致流式输出已经执行完毕，这个请求才结束，因此这个方案并不合理。

4.7.2 设计方案

1. 为 `FlowExecutor` 扩展基于 `CompletableFuture` 的执行方法，提供真正的非阻塞异步执行能力：

代码块

```
1   // 支持异步回调的执行方法
2   public CompletableFuture<LiteflowResponse> execute2CompletableFuture(
3       String chainId, Object param, Class<?>... contextBeanClazzArray) {
4       return CompletableFuture.supplyAsync(
```

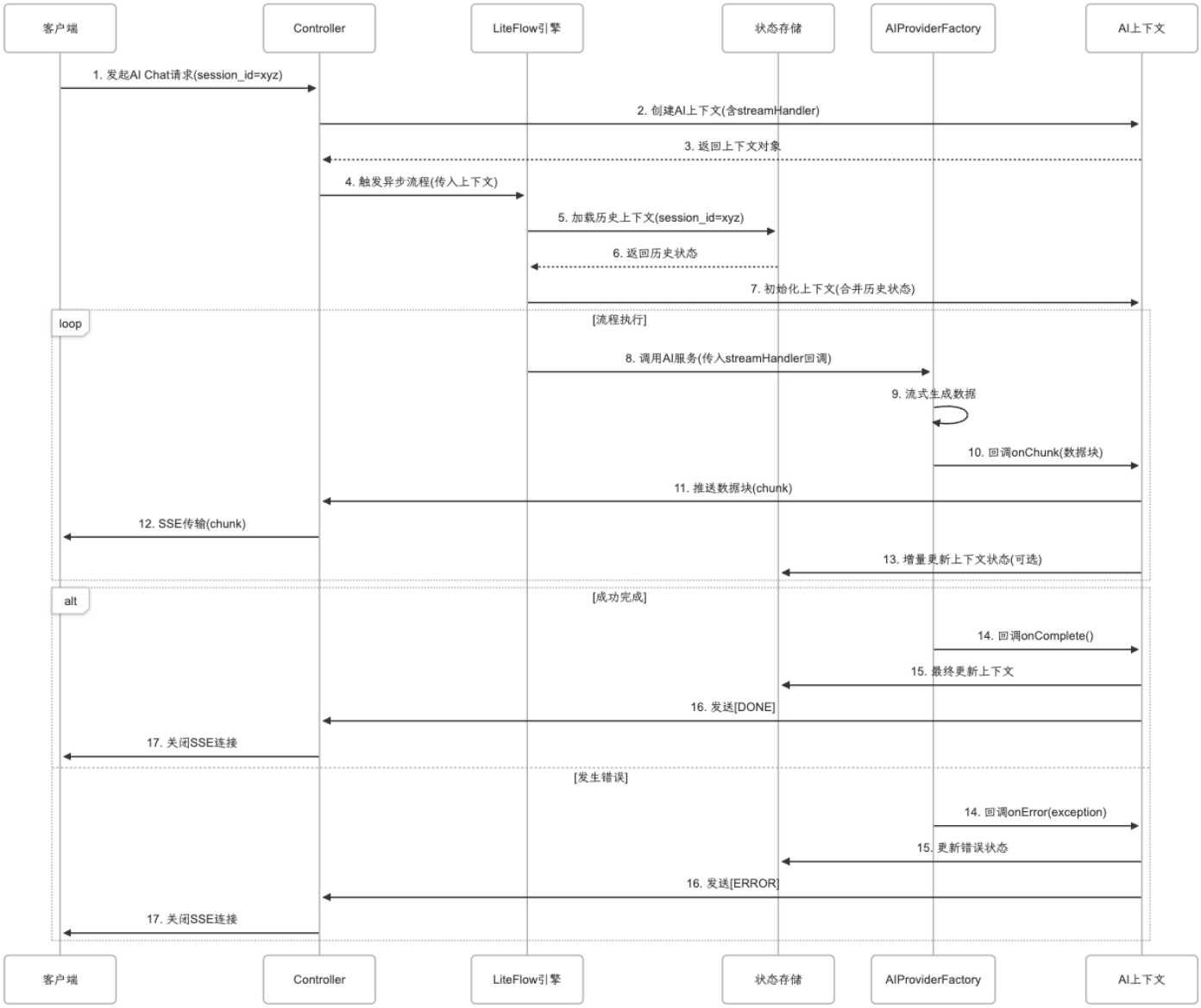
```
5           () -> FlowExecutorHolder.loadInstance().execute2Resp(chainId, param,
    contextBeanClazzArray),
6
    ExecutorHelper.loadInstance().buildMainExecutor(liteflowConfig.getMainExecutorC
    lass())
7       );
8   }
```

该方案的核心优势在于 CompletableFuture 提供的丰富回调API（如 `thenAccept`、`thenCompose`、`exceptionally` 等），使开发者能够以声明式方式定义流程完成后的处理逻辑。

2. 在 AI 上下文中添加流式输出处理器字段，保证每次流程执行前可以根据业务流程自定义处理逻辑。

4.7.3 实现思路

1. 为 `FlowExecutor` 扩展基于 `CompletableFuture` 的执行方法
2. 在AI 上下文中新增 `streamHandler` 字段，在执行流程前交给用户传入
3. 采用回调接口模式，将流式数据片段实时传递给用户定义的处理器
4. 提供完整的流式处理生命周期回调，包括数据块处理、完成通知、错误处理



4.7.4 接口层设计

定义标准的流式处理器接口 `StreamHandler` 交给用户实现，包含三个核心回调方法：

- `onChunk(String chunk)`：处理每个数据片段
- `onComplete(ChatAIOutput result)`：流式输出完成时调用
- `onError(Exception error)`：发生错误时调用

代码块

```
public interface StreamHandler {

    void onChunk(String chunk, AIContext aiContext);

    void onComplete(ChatAIOutput result, AIContext aiContext);

    void onError(Exception error, AIContext aiContext);
```

```
8   }
```

4.8 ByteBuddy实现动态代理和方法拦截

动态代理类将调用 `determineNodeComponentClass(aiComponent.type())` 方法，根据AI组件的类型（`AITypeEnum`）动态确定代理类需要继承的父类。例如，`CHAT` 类型对应 `NodeComponent`，`CLASSIFY` 类型对应 `NodeMultiSwitchComponent`。

针对 `NodeComponent` 的核心方法 `process`：使用 `.method(ElementMatchers.named("process"))` 精确匹配，并指定 `.intercept(InvocationHandlerAdapter.of(new AIInvocationHandler(...)))` 进行拦截

代码块

```
1   private NodeComponent createByteBuddyProxy(AIProxyWrapBean aiProxyWrapBean)
    throws Exception {
2       AIComponent aiComponent = aiProxyWrapBean.getAiComponent();
3
4       // 确定继承的NodeComponent类型
5       Class<? extends NodeComponent> nodeComponentClass =
    determineNodeComponentClass(aiComponent.type());
6
7       // 创建ByteBuddy代理
8       Object instance = new ByteBuddy()
9           .subclass(nodeComponentClass)
10          .name(generateProxyClassName(aiProxyWrapBean))
11          .implement(aiProxyWrapBean.getInterfaceClass())
12          // 拦截process方法，这是NodeComponent的核心方法
13          .method(ElementMatchers.named("process"))
14          .intercept(InvocationHandlerAdapter.of(new
    AIInvocationHandler(aiProxyWrapBean, aiProviderFactory)))
15          // 拦截接口方法
16
    .method(ElementMatchers.isDeclaredBy(aiProxyWrapBean.getInterfaceClass()))
17          .intercept(InvocationHandlerAdapter.of(new
    AIInvocationHandler(aiProxyWrapBean, aiProviderFactory)))
18          .make()
```

```
        .load(AIComponentFactory.class.getClassLoader(), ClassLoadingStrategy.Default.INJECTION)
        .getLoaded()
        .newInstance();

    return (NodeComponent) instance;
}

private Class<? extends NodeComponent> determineNodeComponentClass(AITypeEnum aiType) {
    switch (aiType) {
        case CHAT:
            return NodeComponent.class;
        case CLASSIFY:
            return NodeMultiSwitchComponent.class;
        case KNOWLEDGE_RETRIEVAL:
            return NodeComponent.class;
        default:
            return NodeComponent.class;
    }
}
```

在 `AIInvocationHandler` 中如果方法是 `process`，那么将使用 AI 节点的方法配置进行方法拦截

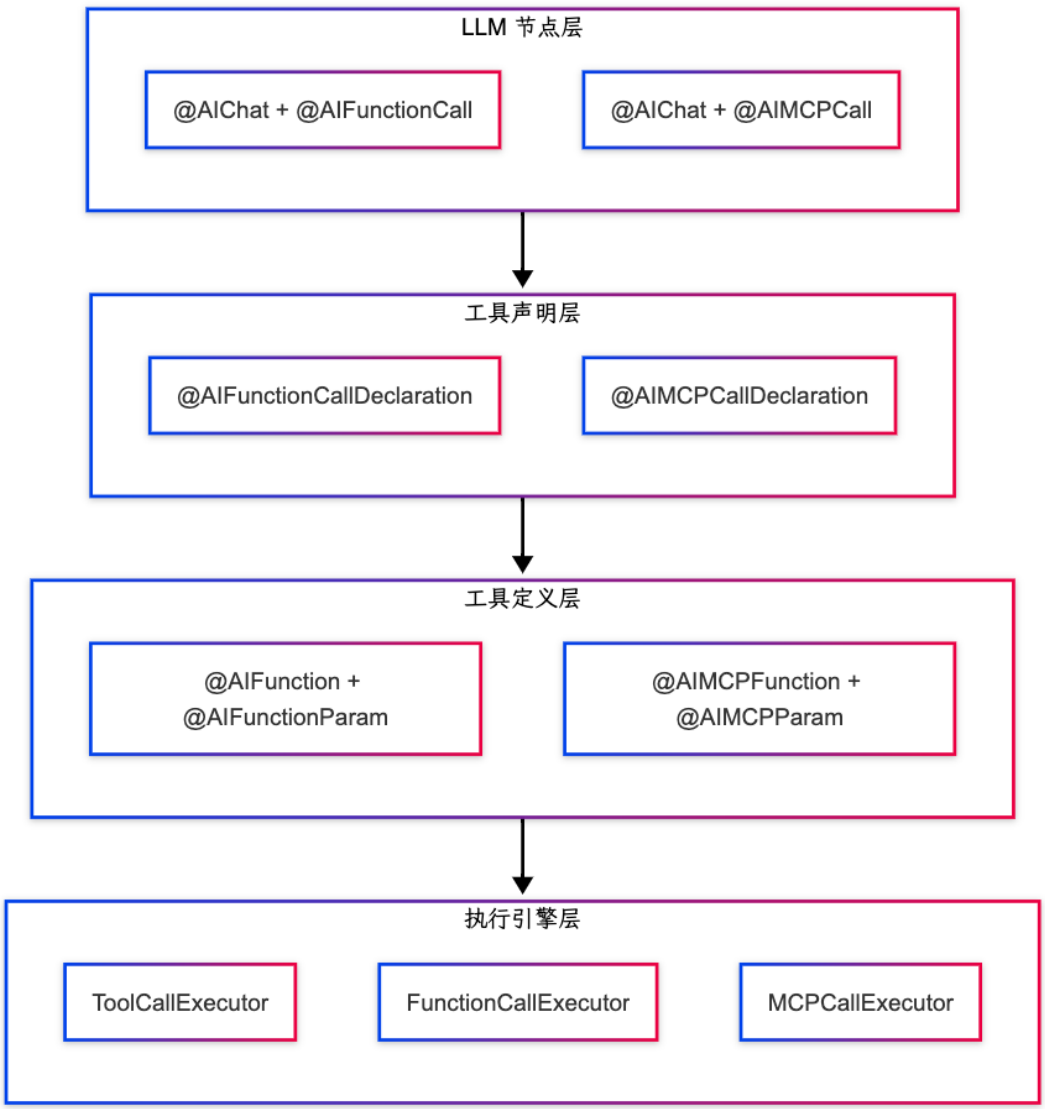
代码块

```
public class AIInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        if ("process".equals(method.getName()) && method.getParameterCount() == 0) {
            return executeAIProcess((NodeComponent) proxy);
        }

        return method.invoke(proxy, args);
    }

    private Void executeAIProcess(NodeComponent nodeComponent) throws Exception {
        AIMethodWrapBean primaryMethod = aiProxyWrapBean.getMethodWrapBeans().get(0);

        executeAIMethod(nodeComponent, primaryMethod, null);

```

```
16              return null;
17          }
18      }
```

5. FunctionCall 与 MCP

采取“定义与声明分离”的设计



5.1 Function Call

@AIFunction - 定义可调用的函数

代码块

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AIFunction {
    String id();                    // 函数唯一标识
    String name() default "";       // 函数显示名称
    String description() default ""; // 函数描述
    String category() default "";   // 函数分类
    boolean async() default false;  // 是否异步执行
    int timeout() default 30;       // 超时时间(秒)
}
```

@AIFunctionParam - 定义函数参数

代码块

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface AIFunctionParam {
    String name();                  // 参数名称
    String description() default ""; // 参数描述
    String type() default "string"; // 参数类型
    boolean required() default true; // 是否必需
    String defaultValue() default ""; // 默认值
    String[] enumValues() default {}; // 枚举值
}
```

@AIFunctionCallDeclaration - 在 LLM 节点中声明函数调用

代码块

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AIFunctionCalling {
    String[] functions(); // 可用函数列表
    boolean logFunctionCalls() default false; // 是否记录函数调用历史
}
```

函数定义示例

代码块

```
@AIFunction(
    id = "get_weather",
    name = "获取天气信息",
    description = "根据城市名称获取当前天气信息"
)
public WeatherInfo getWeather(
    @AIFunctionParam(name = "city", description = "城市名称") String city,
    @AIFunctionParam(name = "unit", description = "温度单位",
                     enumValues = {"celsius", "fahrenheit"},
                     defaultValue = "celsius") String unit
) {
    // 实际的天气查询逻辑
    return weatherAPI.getCurrentWeather(city, unit);
}
```

函数声明示例

代码块

```
@AIComponent(value = "weatherBot", name = "天气助手")
public interface WeatherBotAI {

    @AIChat(
        model = "gpt-4",
        userPrompt = "用户询问: {{userQuestion}}"
    )
    @AIFunctionCallDeclaration(
        // 仅声明使用哪些函数，不关心实现
        functionIds = {
            "getWeather"
        }
    )
    @AIOutput("context.response")
    ChatAIOutput handleWeatherQuery();
}
```

5.2 MCP

@AIMCPFunction - 定义 MCP 服务功能

代码块

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
```

```java
public @interface AIMCPFunction {
    String id();                    // MCP 功能标识
    String mcpServer();             // MCP 服务器标识
    String endpoint();              // MCP 端点路径
    String method() default "POST"; // HTTP 方法
    String description() default ""; // 功能描述
    int timeout() default 60;       // 超时时间(秒)
    boolean cacheable() default false; // 是否可缓存
}
```

@AIMCPParam - 定义 MCP 参数

代码块

```java
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface AIMCPParam {
    String name();                  // 参数名称
    String description() default ""; // 参数描述
    String jsonPath() default "";   // JSON 路径映射
    boolean required() default true; // 是否必需
    String defaultValue() default ""; // 默认值
}
```

@AIMCPCallDeclaration - 在 LLM 节点中声明 MCP 调用

代码块

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(AIMCPCallDeclarations.class)
public @interface AIMCPCallDeclaration {
    String mcpFunctionId();         // 引用的MCP功能ID
    String alias() default "";      // 功能别名
    CallMode mode() default CallMode.AUTO; // 调用模式
    String condition() default "";  // 调用条件
    String[] headerMapping() default {}; // 请求头映射
    String[] paramMapping() default {}; // 参数映射
    int retryCount() default 3;     // 重试次数
    boolean optional() default false; // 是否可选
}
```

6. 表达式扩展

LiteFlow 现有的 EL 表达式非常强大，能够满足绝大多数通用流程的编排需求。然而，在引入 AI 能力后，一些特定的 AI 交互模式和业务场景可能会对现有的表达式提出新的挑战或需求。因此，对表达式进行适当的扩展，是确保 LiteFlow-AI 插件能够高效、自然地编排复杂智能流程的关键。

6.1 现有表达式的局限性分析

- `SWITCH` 表达式的单选限制：
  - LiteFlow 的 `SWITCH(a).to(b, c, d)` 表达式非常适合基于单个条件值进行单路径选择的场景。
  - 但在 AI 领域，例如 **意图识别 (Intent Classification)**，大模型往往能识别出用户的多个意图，并给出每个意图的置信度。例如，用户说“我想订一张去北京的机票，顺便问下天气”，这可能同时包含“订票”和“天气查询”两个意图。如果只用 `SWITCH`，则只能选择一个主要意图进行处理，可能会丢失信息或需要复杂的预处理逻辑将多意图合并为单一决策点。

6.2 新增与增强表达式的思考

6.2.1 `MULTI_SWITCH`

`MULTI_SWITCH` 是本次表达式扩展的重点，旨在解决 `SWITCH` 单选的局限性，使其能更好地适应 AI 场景中常见的“多判定-多分支”需求。

- **输入源**：`MULTI_SWITCH` 的输入不再是单一的组件 ID，而应该是一个能够产生 **多个决策值** 的表达式或组件。我们可以定义一个 `NodeMultiSwitchComponent`，定义一个返回参数为 `Set<String>` 的 `processMultiSwitch` 方法获取目标 `NodeId` 集合。例如，意图分类组件输出的意图ID列表 `["booking", "inquiry"]`。
- **执行模式 (Mode)**：
  - `PARALLEL` (默认): 如果输入源产生了多个匹配的决策值，所有匹配的分支将并行执行。

- `SEQUENTIAL`：如果输入源产生了多个匹配的决策值，匹配的分支将按照其在 `.to()` 中声明的顺序串行执行。
- `FIRST_MATCH`：只执行第一个匹配到的分支（类似传统 `SWITCH`，但源可以是多值的）

- **`MultiSwitchOperator`**：
  - `build(Object[] objects)` 方法会解析 `MULTI_SWITCH(expression)` 中的 `expression`。这个 `expression` 会在运行时被求值，预期结果是一个集合(Set)。
  - 它会构造一个 `MultiSwitchCondition` 对象。
- **`MultiSwitchCondition`**：
  - 存储了源表达式、执行模式、目标分支映射（决策值 -> 流程/组件）、分支条件、默认分支等。
  - 在流程执行到此条件时，它会：
    i. 求值源表达式，获取决策值集合。
    ii. 遍历决策值集合。
    iii. 对于每个决策值，查找 `.to()` 中是否有匹配的分支。
    iv. 如果找到匹配分支，再检查 `.condition()` 中是否有为该分支定义的附加条件，并求值。
    v. 根据执行模式（`PARALLEL`，`SEQUENTIAL`，`FIRST_MATCH`）和匹配/条件满足情况，调度执行相应的子流程。
    vi. 如果没有分支被触发且定义了 `default`，则执行默认分支。

以改造 `SwitchCondition#executeCondition` 为例（暂未编写执行模式的代码），蓝色代码为修改部分，主要是将结果获取从单个 `targetId` 改为获取 `Set` 集合，实现多路选择的效果。

代码块

```
@Override
public void executeCondition(Integer slotIndex) throws Exception {
    // 获取switch node
    Node switchNode = this.getSwitchNode();
    // 获取target List
    List<Executable> targetList = this.getTargetList();

    // 提前设置 chainId，避免无法在 isAccess 方法中获取到
    switchNode.setCurrChainId(this.getCurrChainId());

    // 先去判断isAccess方法，如果isAccess方法都返回false，整个SWITCH表达式不执行
    if (!switchNode.isAccess(slotIndex)) {
        return;
    }
```

```
15
16      // 先执行switch节点
17      switchNode.execute(slotIndex);
18
19      // 拿到multi-switch节点的结果
20      Set<String> targetIds = switchNode.getItemResultMetaValue(slotIndex);
21
22      Slot slot = DataBus.getSlot(slotIndex);
23
24      List<Executable> targetExecutors = new ArrayList<>();
25      if (CollectionUtil.isNotEmpty(targetIds)) {
26          for (String targetId : targetIds) {
27              if (StrUtil.isNotBlank(targetId)) {
28                  // 这里要判断是否使用tag模式跳转
29                  if (targetId.contains(TAG_FLAG)) {
30                      String[] target = targetId.split(TAG_FLAG, 2);
31                      String _targetId = target[0];
32                      String _targetTag = target[1];
33                      Executable executor =
    targetList.stream().filter(executable -> {
34                          return (StrUtil.startWith(_targetId, TAG_PREFIX) &&
    ObjectUtil.equal(_targetTag,executable.getTag()))
35                                  || ((StrUtil.isEmpty(_targetId) ||
    _targetId.equals(executable.getId()))
36                                  && (StrUtil.isEmpty(_targetTag) ||
    _targetTag.equals(executable.getTag())));
37                      }).findFirst().orElse(null);
38                      if (ObjectUtil.isNotNull(executor)) {
39                          targetExecutors.add(executor);
40                      }
41                  }
42                  else {
43                      Executable executor = targetList.stream()
44                              .filter(executable ->
    ObjectUtil.equal(executable.getId(),targetId) )
45                              .findFirst()
46                              .orElse(null);
47                      if (ObjectUtil.isNotNull(executor)) {
48                          targetExecutors.add(executor);
49                      }
50                  }
51              }
52          }
53      }
54
55      if (CollectionUtil.isEmpty(targetExecutors)) {
56          // 没有匹配到执行节点，则走默认的执行节点
```

```
57          targetExecutors.add(this.getDefaultExecutor());
58      }
59
60      for (Executable targetExecutor : targetExecutors) {
61          if (ObjectUtil.isNotNull(targetExecutor)) {
62              // switch的目标不能是Pre节点或者Finally节点
63              if (targetExecutor instanceof PreCondition || targetExecutor
    instanceof FinallyCondition) {
64                  String errorInfo = StrUtil.format(
65                          "[{}]:switch component[{}] error, switch target node
    cannot be pre or finally",
66                          slot.getRequestId(),
    this.getSwitchNode().getInstance().getDisplayName());
67                  throw new SwitchTargetCannotBePreOrFinallyException(errorInfo);
68              }
69              targetExecutor.setCurrChainId(this.getCurrChainId());
70              targetExecutor.execute(slotIndex);
71          }
72          else {
73              String errorInfo = StrUtil.format("[{}]:no target node find for
    the component[{}],target str is [{}]",
74                      slot.getRequestId(),
    this.getSwitchNode().getInstance().getDisplayName(), targetExecutor.getId());
75              throw new NoSwitchTargetNodeException(errorInfo);
76          }
77      }
78  }
```

五、项目开发规划

时间范围	主要任务
第一阶段：核心架构搭建与基础 AI 节点开发 (7月1日 - 7月31日)	
7月1日 - 7月15日	核心架构设计：完成 LiteFlow-AI 插件的整体架构设计，包括 AIProviderFactory、核心注解（@AIComponent, @AIChat 等）、动态代理机制及全局配置方案。
7月16日 - 7月31日	核心功能实现：实现 AIProviderFactory；支持至少两家主流大模型（如 OpenAI、Ollama）；完成核心注解解析与 ByteBuddy 动态代理逻辑；开发 AIChat、AIClassify、AIRetrieval 三类基础 AI 节点的核心逻辑。
第二阶段：AI 节点功能增强与测试体系构建 (8月1日 - 8月31日)	

8月1日 - 8月15日	AI 节点功能增强：为 AIChat 增加流式输出；支持 Function Call（包括注解及调用逻辑）；按需支持多模态输入；增强 AIClassify 的多标签/置信度功能；初步集成 AIRetrieval 的向量检索。
8月16日 - 8月31日	搭建测试模块与示例工程：建立 liteflow-ai-test 测试模块，覆盖单元与集成测试；搭建 liteflow-ai-example 演示工程，展示基础 AI 节点使用场景。
第三阶段：功能完善、综合测试与文档输出 (9月1日 - 9月30日)	
9月1日 - 9月15日	表达式扩展与综合示例：实现如 MULTI_SWITCH 等表达式扩展，开发复杂流程场景以覆盖所有 AI 节点特性；增强演示工程。
9月16日 - 9月30日	文档与交付准备：编写完整的插件用户手册（安装、配置、使用指南等）、API 文档；整理代码注释；准备项目总结和演示材料。