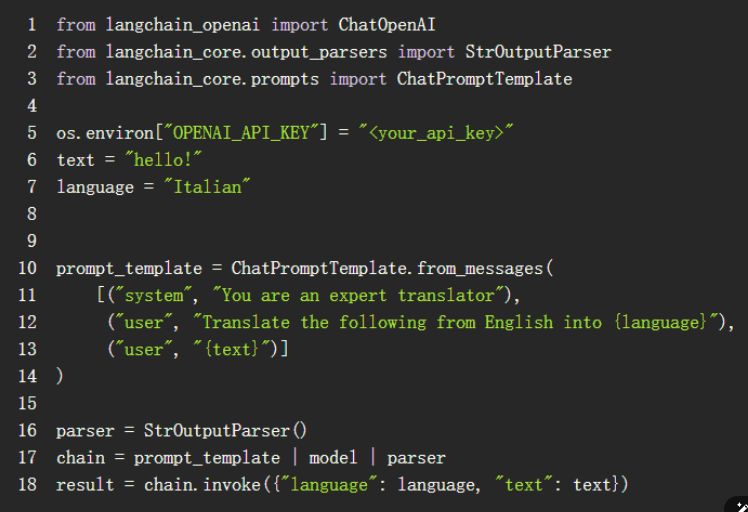
**LangChain分析部分**

**LangChain缺陷**

Langchain最大的缺陷在于：过度抽象。

**LangChain 的过度抽象在实际使用过程中的缺陷体现：**





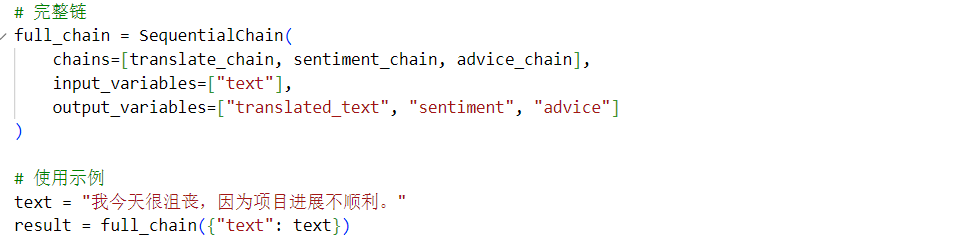
在实现相同的功能（翻译）时，使用LangChain需要引3个抽象（ChatPromptTemplate, StrOutputParser, chain），使得代码的复杂性提升，过度的抽象虽然隐藏了实现的细节，但限制了实际使用过程中的灵活性，必须满足给定的数据结构要求进行设计，故开发者就必须花费大量时间去理解和调试与实际功能代码无关的内部框架代码。

**LangChain 嵌套抽象（在抽象的概念上使用抽象）带来的潜在弊端：**

如图示例，每个 LLMChain 实现了特定功能的抽象，如翻译、情感分析和建议生成，通过抽象化的方式（使用 PromptTemplate 定义输入和输出），使得每个链可以独立处理其特定的任务。

而full\_chain 实现了高层次的抽象，依赖于之前定义的低层次链（translate\_chain、sentiment\_chain 和 advice\_chain），实现完整功能。

但过度的抽象与嵌套会产生代码复杂理解困难、依赖关系复杂、性能开销较大等问题。



**策略模式**

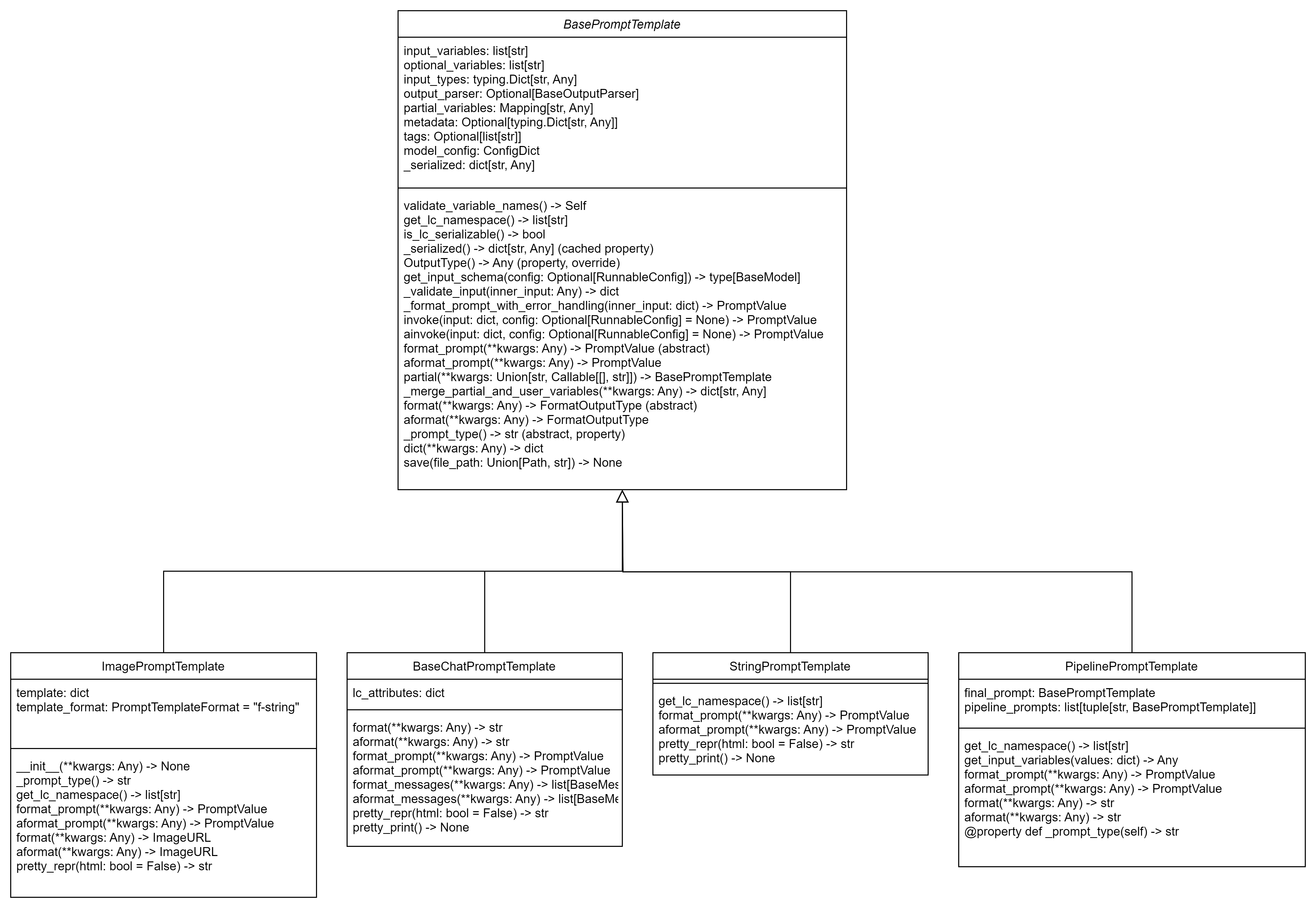
在LangChain的prompt模块中有策略模式的体现。

基本理论：

策略模式的核心思想是将算法的定义与使用分离，即将不同的算法封装成独立的类，并使它们可以相互替换。

这样可以在不修改客户端代码的情况下，通过替换算法类来改变算法的行为，从而提高系统的灵活性和可扩展性。

其遵循了开闭原则，即对扩展开放，对修改关闭，使得系统在不修改原有代码的基础上能够轻松地添加新的算法或修改现有算法。



**类的分析说明：**

BasePromptTemplate

"""Base class for all prompt templates, returning a prompt."""

它是所有提示模板的抽象基类，提供所有提示模板类的基础接口和属性。

该类通过定义和格式化输入变量、处理部分变量、格式化输出、以及与可运行的提示相关的元数据等功能，提供构建复杂的提示模板的框架。

ImagePromptTemplate 继承自 BasePromptTemplate，用于处理图像提示。

BaseChatPromptTemplate 继承自 BasePromptTemplate，用于处理适用于聊天上下文的提示。

StringPromptTemplate 继承自 BasePromptTemplate，用于处理字符串提示的格式化和表示。

PipelinePromptTemplate 继承自 BasePromptTemplate，用于提供灵活且模块化的方式来构建复杂的提示，通过将多个提示模板组合在一起，允许开发者重用现有的模板，从而减少重复代码。

**其应用于LangChain的表现与优势在于：**

1 灵活性和可扩展性

策略模式使得不同的提示模板可以轻松地互换或添加新模板。

如果需要支持新的提示类型，只需创建一个新的模板类并实现 BasePromptTemplate 接口，而不需要对现有的代码进行修改，极大地提高了系统的灵活性和可扩展性。

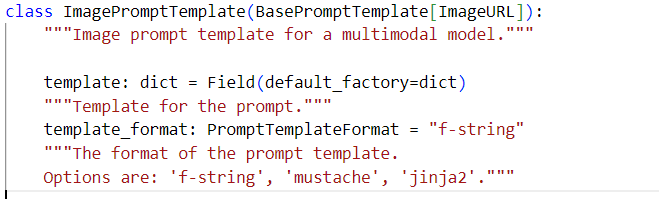
2 清晰的责任分离

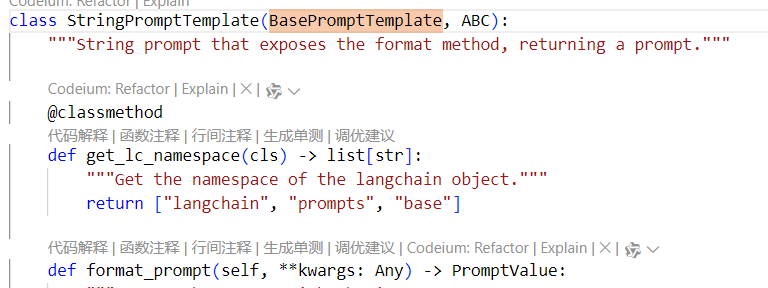
策略模式通过将算法的实现与其使用分离，确保了每个类的责任清晰。

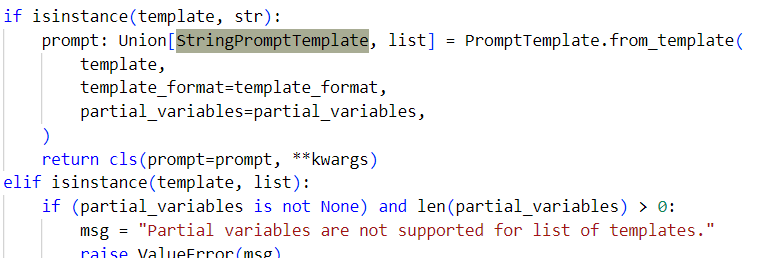
BasePromptTemplate 定义了通用的接口和属性，而各个子类则实现具体的提示逻辑，避免了各个部分之间的耦合。

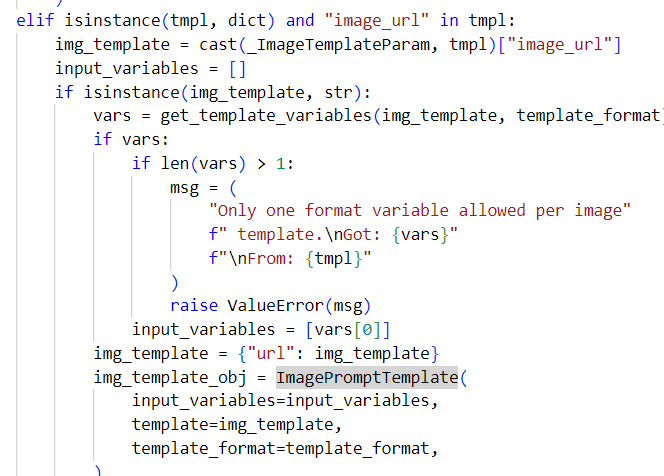
此外，各类的组合与使用可用于支持多模态的交互任务，基于不同的任务可以使用提示模板的框架及其实现得到灵活的组合，故能够增强系统与不同类型数据交互的能力。

**相关代码印证（子类继承与实际调用）：**









**迭代器模式**

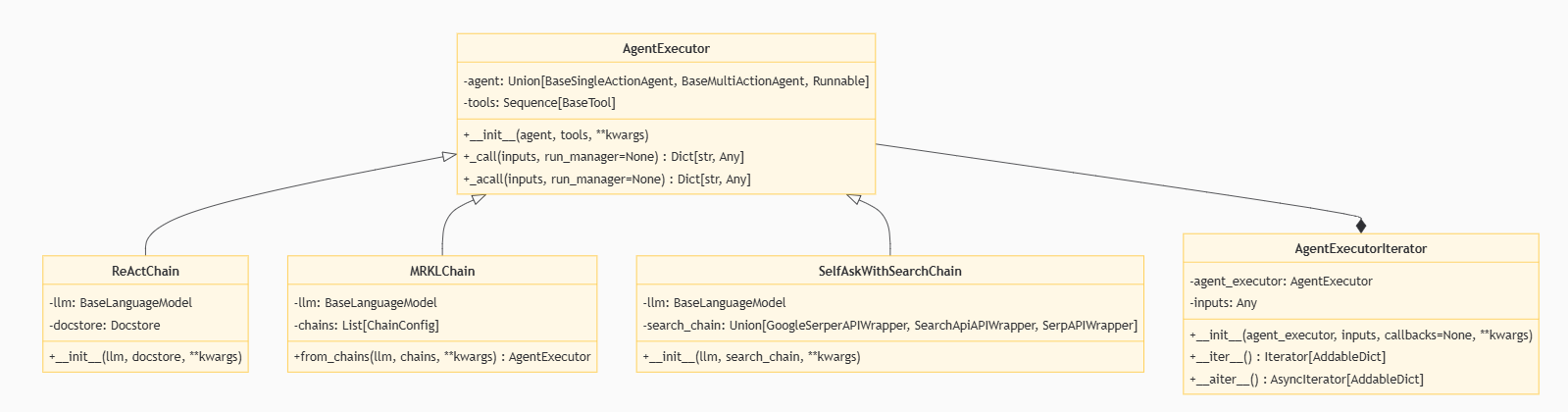
在LangChain的agent模块中有迭代器模式的体现。

**基本理论：**

迭代器模式是一种行为设计模式，它允许客户端遍历一个容器对象中的内容而无需暴露其底层的表现形式。迭代器模式通常包含两个主要的角色：

容器（Container）：它包含一系列元素，提供了创建一个迭代器对象的接口。

迭代器（Iterator）：它是一个对象，用于遍历容器中的元素，通常具有 hasNext()、next()、remove() 等方法。



**类的分析说明：**

AgentExecutor 类:

这是一个执行代理（agent）的类，使用了一系列tools来执行任务。

它负责管理代理的执行流程，包括计划行动、执行行动、观察结果，并根据这些信息决定下一步行动。类中包含同步和异步执行的逻辑，提供了 iter 和 stream 方法来同步迭代执行代理，以及 aiter 和 astream 方法来异步迭代执行代理。

AgentExecutorIterator 类:

这个类是 AgentExecutor 的迭代器，它允许外部代码通过迭代的方式访问 AgentExecutor 的执行过程，负责维护执行过程中的状态，如中间步骤、迭代次数和已过时间。

该类提供了 \_\_iter\_\_ 和 \_\_aiter\_\_ 方法来支持同步和异步迭代，还提供了 \_process\_next\_step\_output 和 \_aprocess\_next\_step\_output 方法来处理每一步的输出，并且决定是否继续执行或停止迭代。

ReActChain类:

继承AgentExecutor 类，实现 ReAct 系统的特定逻辑，使用文档存储（docstore）和语言模型（LLM），初始化时创建与文档存储交互的工具。

MRKLChain类:

继承AgentExecutor 类，实现 MRKL 系统，使用多个链（chains）和语言模型（LLM），

提供了一个类方法 from\_chains 来初始化 MRKL 链。

SelfAskWithSearchChain类:

继承AgentExecutor 类，实现自我询问和搜索的逻辑，使用语言模型（LLM）和搜索链，初始化时创建搜索工具。

**其应用于LangChain的表现在于：**

AgentExecutor 可以被视为容器角色，它包含了执行代理操作的逻辑和状态。

AgentExecutorIterator 则是迭代器角色，它提供了遍历 AgentExecutor 执行过程的方法，如 \_\_iter\_\_ 和 \_\_aiter\_\_ 方法，允许客户端代码以迭代的方式访问 AgentExecutor 的每一步执行结果。

它封装了 AgentExecutor 的执行过程，允许客户端代码通过迭代的方式访问执行的每一步。客户端代码可以通过 AgentExecutorIterator 的 \_\_iter\_\_ 或 \_\_aiter\_\_ 方法来创建迭代器，然后使用迭代器遍历 AgentExecutor 的执行过程。这种设计模式隐藏了 AgentExecutor 的内部实现细节，客户端代码只需关注如何迭代执行过程，而不需要了解具体的执行逻辑。

**其应用于LangChain的优势在于：**

1 解耦：迭代器模式使得 AgentExecutor 的执行逻辑与客户端代码解耦，提高了代码的可维护性。

2 灵活与扩展性：可以轻松添加新的迭代器类来支持不同的遍历方式，而不需要修改现有的容器类或客户端代码，且客户端代码可以根据需要选择不同的迭代方式，如同步或异步迭代，提高了代码的灵活性。

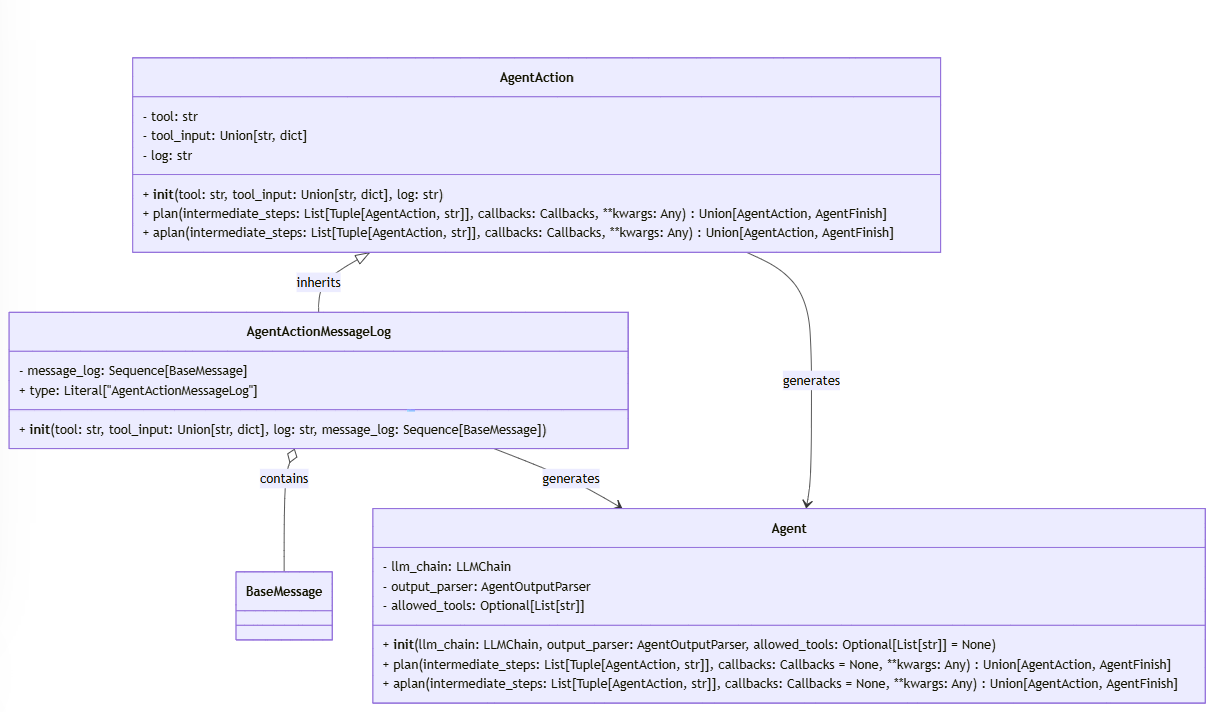
3 封装性：迭代器模式隐藏了容器的内部结构，客户端代码不需要了解容器的具体实现，只需要通过迭代器访问容器的内容。

**命令模式**

在LangChain的agent模块中有命令模式的体现。

**基本理论：**

命令模式将一个请求或操作封装成一个对象，将发起请求的对象（调用者）和执行请求的对象（接收者）解耦，提供了一种灵活的机制来表示、存储、传递和执行请求，调用者通过命令对象来间接地发起请求，而不需要知道具体是哪个接收者会处理这个请求，也不需要知道请求是如何被执行的。



**类的分析说明：**

Agent类:

使用 LLMChain 和 AgentOutputParser 来决定代理的动作，其中plan 和 aplan 方法用于生成 AgentAction 或 AgentFinish 对象。

AgentAction类:

表示代理想要执行的动作，包含工具名称（tool）、工具输入（tool\_input）和日志信息（log）。

AgentActionMessageLog类:

AgentAction 的子类，增加了消息日志（message\_log），用于记录聊天模型生成的消息，type 属性用于序列化和区分不同的 AgentAction 子类。

**其应用于LangChain的表现在于：**

Agent 类作为 Invoker 角色，调用 plan 或 aplan 方法来决定要执行的命令（AgentAction 或 AgentActionMessageLog 对象等），这些方法接收中间步骤和回调，然后使用语言模型（LLM）来决定下一步动作。一旦 Agent 生成了 AgentAction 对象，它就代表了一个具体的执行请求。这个请求可以被直接执行，或者被传递给其他组件（如 AgentExecutor）来执行。

AgentAction 和 AgentActionMessageLog 类封装了执行动作所需的所有信息，包括工具名称、工具输入和日志信息，这符合命令模式中命令对象的职责。

**其应用于LangChain的优势在于：**

1灵活性：Agent 类可以生成不同类型的 AgentAction 对象和序列，使得系统可以灵活地处理各种请求。

2扩展性：通过添加新的 AgentAction 子类，可以轻松扩展系统以支持新的命令类型。

3 解耦：Agent 类和执行逻辑之间的解耦使得系统更容易维护和测试。

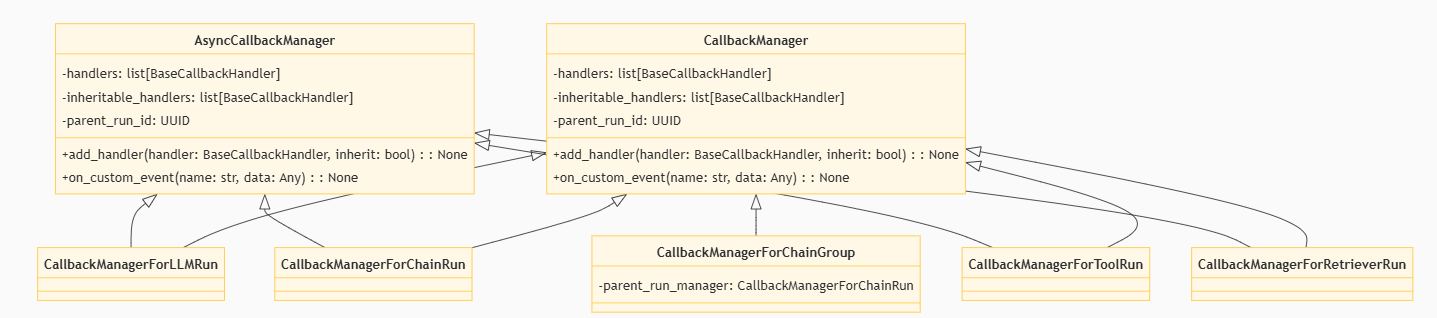
通过将命令对象的生成和执行分离，Agent 类可以独立于具体的执行逻辑。这意味着 Agent 可以专注于决策逻辑，而具体的执行可以由其他组件处理。上述体现了命令模式的原则，提供了一个灵活、可扩展且解耦的架构来处理代理的决策和执行逻辑。

**责任链模式**

在LangChain的核心模块core中的回调管理callbacks manager部分有责任链模式的体现。

基本理论：

责任链模式允许一个请求沿着链传递，直到有一个对象处理它为止。在 langchain 的回调管理系统中，这个请求就是事件，而链就是回调处理器的列表。



**类的分析说明：**

BaseCallbackManager类：

这是一个抽象基类，定义了回调处理器的基本接口。它规定了所有具体回调处理器必须实现的方法，例如 on\_llm\_start、on\_chain\_start 等。

CallbackManager类：

继承BaseCallbackManager类，提供了回调管理的基本功能，包含方法来处理各种事件，如 on\_llm\_start、on\_chain\_start、on\_tool\_start 等。

AsyncCallbackManager类：

继承BaseCallbackManager类，是异步版本的 CallbackManager，用于异步回调处理。

CallbackManagerForLLMRun、CallbackManagerForChainRun、CallbackManagerForToolRun、CallbackManagerForRetrieverRun类：

这些类是 CallbackManager 的特定场景实现，分别用于 LLM 运行、链运行、工具运行和检索器运行。

CallbackManagerForChainGroup类：

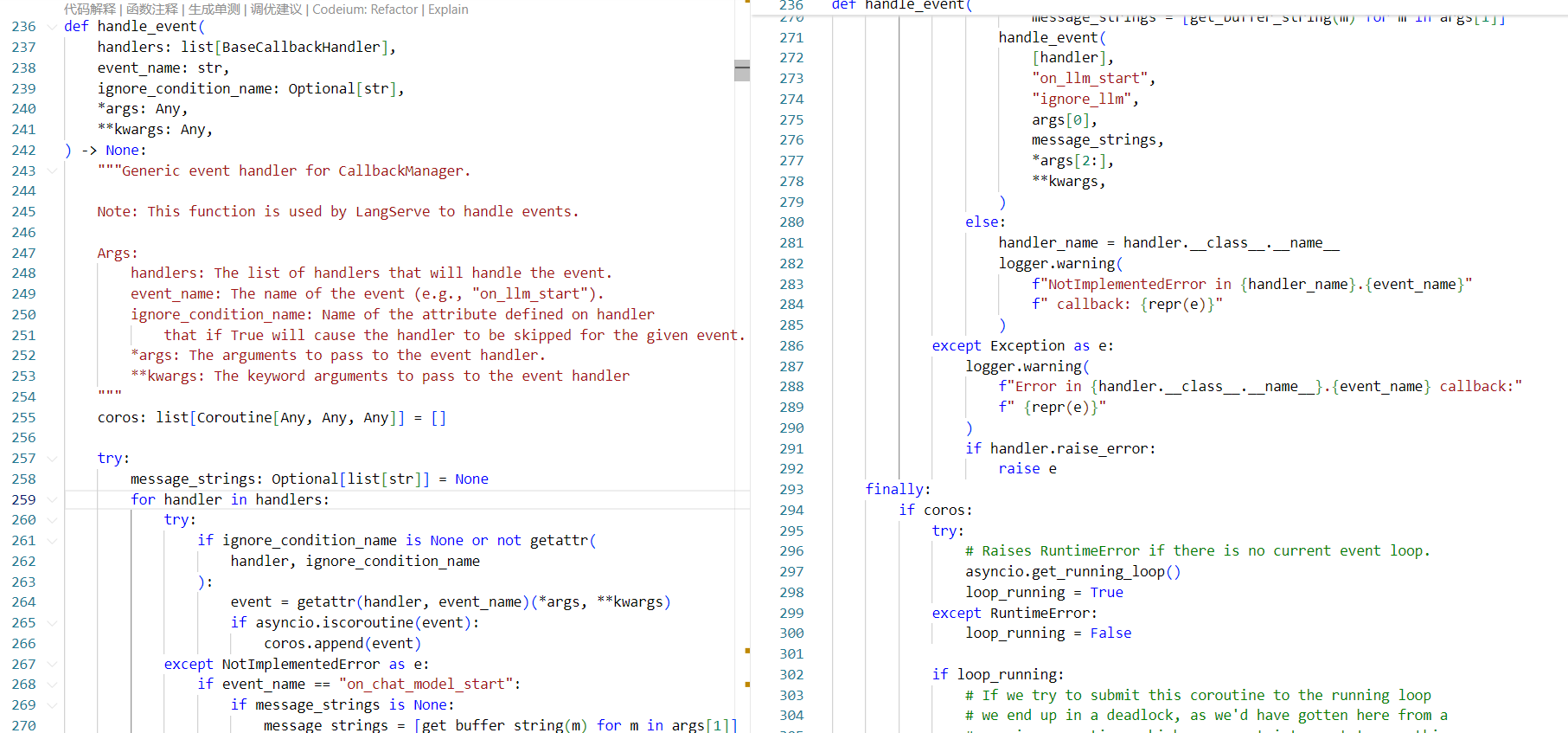
用于管理链组的回调，可以合并多个回调管理器。

**其应用于LangChain的表现与优势在于：**

在 langchain 框架中，责任链模式的体现主要是通过 CallbackManager 类及其子类中的 handle\_event 函数和 ahandle\_event 异步函数来实现的。上述函数负责遍历注册的回调处理器（handlers），并按顺序调用它们，直到某个处理器处理了事件或者所有处理器都忽略该事件。

在 CallbackManager 类中，可以通过 add\_handler 方法动态地添加回调处理器到责任链中。而handle\_event 函数（同步）和 ahandle\_event 函数（异步）是触发事件的关键。它们接收一系列回调处理器和一个事件名称，然后按顺序调用这些处理器直到事件被处理。每个回调处理器都实现了特定的事件处理方法，如 on\_llm\_start、on\_chain\_end 等。如果一个处理器不处理特定的事件，它可以选择不做任何操作，或者将事件传递给链中的下一个处理器。

以handle\_event 函数代码为例：



for handler in handlers 循环遍历所有注册的回调处理器；getattr(handler, event\_name)(\*args, \*\*kwargs) 调用当前处理器的特定事件处理方法。

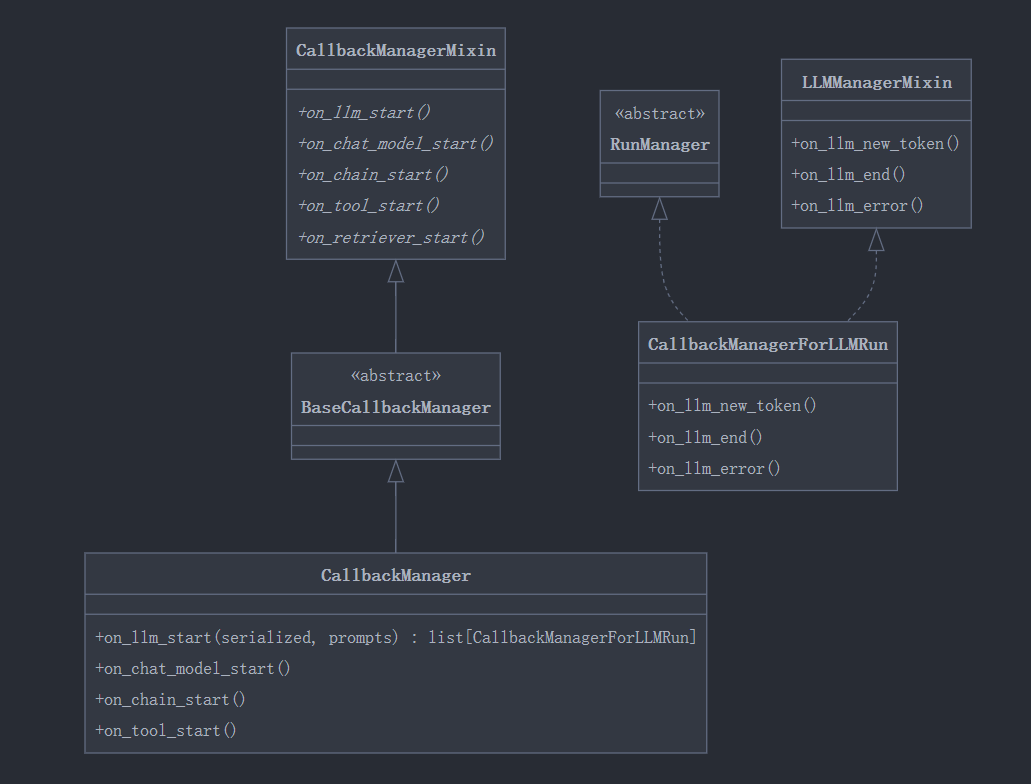
如果事件处理方法返回一个协程（asyncio.iscoroutine），则将其添加到 coros 列表中，稍后将一起执行这些协程；如果处理器方法抛出 NotImplementedError 或其他异常，将记录警告或重新抛出异常，便允许链中的下一个处理器有机会处理该事件。

**观察者模式**

在LangChai核心模块的callback部分中有观察者模式的体现

**基本理论：**

观察者模式的核心算法在于当被观察者的状态发生改变时，所有依赖于它的对象即观察者都会得到通知并自动更新。其提供了一种灵活的、松耦合的方式来处理对象间的交互，使得系统更加易于维护和扩展。



补：类图中补充一个从CallbackManager类指向CallbackManagerForLLMRun的create箭头

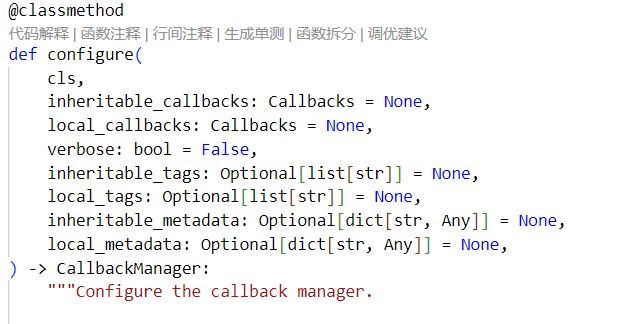
**其应用于LangChain的表现与有事在于：**

langchain\_core.callbacks.manager.py文件中定义了CallbackManager类，用于管理回调函数，例如当LLM有事件发生时（Subject被观察者主要有LLM、Chain、Tool等核心组件），CallbackManager会遍历观察者列表，调用每个观察者的相应方法，通知它们事件的发生。

通过使用观察者模式，LangChain实现了灵活、可扩展的回调系统，能够方便地添加各种监控、日志、调试等功能。

**相关代码印证：**

观察者注册机制



状态变化通知机制

