8. 回调处理器

随着技术的发展，无论是为用户实时显示数据，还是为开发者提供即时的系统日志，LangChain都希望系统能够在关键时刻为我们提供即时信息。这正是回调处理器（Callbacks）的用途。回调处理器允许开发者在特定事件发生时执行自定义操作，这在许多场景中都非常有用，例如日志记录、性能监控、流式处理等。

## 8.1 回调处理器的定义

在 LangChain 框架中，回调处理器是一种特殊的包装机制，允许开发者定义一系列的方法来响应不同的事件。每当特定事件被触发，相应的回调处理器方法就会被执行。举个简单的例子，想象一下每当我们的应用开始处理新的数据流时，都能自动记录日志。

回调处理器的设计目的是为了提供一个统一、模块化和可重用的机制，使开发者能够更轻松地为链和 Agent 添加各种回调功能。

下面是使用回调处理器的几个好处：

模块化与可重用性：通过定义回调处理器，你可以创建一组可重用的操作，这些操作可以轻松地在不同的LangChain实例或应用中使用。例如，如果你有多个应用都需要流式输出到WebSocket，那么使用一个统一的 WebSocketStreamingHandler 可以避免重复的代码。

灵活性：回调处理器提供了一个结构化的方式来响应各种事件，而不仅仅是流式输出。这意味着你可以为各种事件（如链开始、链结束、错误发生等）定义特定的逻辑。

与 LangChain 的其他组件紧密集成：回调处理器是为LangChain特别设计的，确保与其内部机制的兼容性和高效性。

代码清晰与维护性：通过使用专门的回调处理器，你的代码结构会更清晰。当其他开发者查看或维护你的代码时，他们可以轻松地找到和理解回调逻辑。

然而，这并不意味着必须使用回调处理器。如果你发现直接在应用逻辑中实现特定功能更方便或更适合你的需求，那么完全可以这样做。LangChain 框架的回调处理器只是提供了一个方便、统一的工具，旨在简化开发者的工作。

### 8.1.1 内置回调处理器

为了简化开发过程，LangChain提供了一系列内置的回调处理器，比如我们运行一个 Agent ，它的底层都使用到了 StdOutCallbackHandler 。例如我们在下面代码中设置 verbose=True ，在 Agent 运行时，也就是事件发生时将 Agent 的相关信息打印到标准输出（通常是控制台或命令行界面）。

import os

os.environ["OPENAI\_API\_KEY"] = "填入你的密钥"

from langchain.agents import load\_tools

from langchain.agents import initialize\_agent

from langchain.agents import AgentType

from langchain.llms import OpenAI

llm = OpenAI()

tools = load\_tools(["llm-math"], llm=llm)

agent = initialize\_agent(

    tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,verbose=True

)

agent.run("9+7")

我们在命令行界面就可以看到以 Entering new AgentExecutor chain... 开始，以 Finished chain 结尾的标准输出。这是因为当运行 Agent 并启用 verbose=True 时，StdOutCallbackHandler 将自动激活，将 Agent 的活动实时打印到标准输出。这为开发者提供了即时的反馈，帮助他们了解Agent的工作情况。

> Entering new AgentExecutor chain...

 I need to add two numbers together

Action: Calculator

Action Input: 9+7

Observation: Answer: 16

Thought: I now know the final answer

Final Answer: 9+7 = 16

> Finished chain.

'9+7 = 16'

但是如果设置为 verbose=False ，我们在命令行界面只会看到大语言模型最终的回答。

'9+7 = 16'

为 LLMChain 添加一个内置回调处理器

我们还可以给链组件和 Agent 添加内置或者自定义的回调处理器。比如我们给一个基础的链组件 LLMChain 添加一个内置回调处理器： StdOutCallbackHandler。我们可以先创建一个链组件。在初始化 LLMChain 的时候，这个链组件没有设置内置回调处理器，也不能设置 verbose=True。

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

from langchain.llms import OpenAI

llm\_chain = LLMChain(llm=OpenAI(),

                     prompt=PromptTemplate.from\_template("{input}"))

llm\_chain.run('上海的旅游景点有哪些？')

运行 LLMChain 后，大语言模型回答的是：

上海的旅游景点有：\n\n1. 上海迪士尼乐园\n2. 东方明珠塔\n3. 南京路步行街\n4. 上海外滩\n5. 上海野生动物园\n6. 外白渡桥\n7. 南京路商业街\n8. 上海科技馆\n9. 上海老城隍庙\n10. 上海博物馆\n11. 上海浦江夜游\n12. 上海水上乐园\n13. 上海徐汇森林公园\n14. 上海金茂大厦\n...

如果我们想要监控这个链组件，添加一些回调逻辑，比如我们想要命令行输出这个链运行的相关信息。我们可以给这个链组件增加一个回调处理器。这里我们导入内置的 StdOutCallbackHandler，并且创建创建它的实例 handler\_1 。

from langchain.callbacks import StdOutCallbackHandler

handler\_1 = StdOutCallbackHandler()

回调处理器的使用很简单，我们给 LLMChain 链组件添加 callbacks=[handler\_1], 列表的数据格式这意味着我们是可以添加多个回调处理器，完成不同的任务逻辑。

llm\_chain = LLMChain(llm=OpenAI(),callbacks=[handler\_1],

                     prompt=PromptTemplate.from\_template("{input}"))

llm\_chain.run('上海的旅游景点有哪些？')

这样我们就给一个 LLMChain 链组件添加了一个回调处理器，内置的 StdOutCallbackHandler 完成的是做标准的链组件的打印输出。当我们继续运行这个链组件之后，我们在命令行界面就可以看到以 Entering new AgentExecutor chain... 开始，以 Finished chain 结尾的标准输出。

> Entering new LLMChain chain...

Prompt after formatting:

上海的旅游景点有哪些？

> Finished chain.

'\n\n上海的旅游景点有：\n\n1. 东方明珠广播电视塔\n2. 豫园\n3. 外滩\n4. 南京路步行街\n5. 上海野...'\n\n

### 8.1.2 自定义回调处理器

在开发过程中，我们常常遇到一些特定的需求，例如为每个用户请求单独创建日志文件，或在发生某一关键事件时及时发送通知。这些需求超出了通常的范围，而自定义回调处理器在此时发挥了其重要的作用。尽管LangChain提供的内置回调处理器功能强大，但有时仍然不能满足某些特定场景的需求。为了真正充分利用回调处理器，我们需要设计并实现自己的处理器。

每一个链，从它被创建到最终被销毁，都会经历多个关键阶段。在这个生命周期中，链的每一个阶段都可能会触发某些事件。为了确保在恰当的时机介入并执行相应的操作，我们需要深入了解每一个阶段并对其进行精确的控制。这样，当链经历某个特定的阶段时，我们就可以在相应的回调处理器方法中执行预定的代码。

在某些情况下，我们可能只希望在特定的请求中执行特定的代码，而不是在链的整个生命周期中。这种需求可以通过请求回调来实现。请求回调为我们提供了这种特定于请求的灵活性。例如，想象一个场景，你只希望在某个特定的请求中记录日志，而不是在所有的请求中都这样做。通过使用请求回调，你可以轻松地达到这个目的。

不过，无论我们如何设计和使用回调处理器，最关键的部分始终是理解自己的业务逻辑需求。只有当我们清晰地知道自己想要达到的目的时，我们才能在正确的回调处理器方法中插入合适的代码，确保在合适的时机执行正确的操作。自定义回调处理器为我们提供了一个强大的框架，但如何充分利用这个框架，最终取决于我们对业务需求的理解和技术的运用。

BaseCallbackHandler详解

在编写自定义回调处理器之前，了解其背后的基础类是非常重要的。 BaseCallbackHandler 正是这样一个核心类，它为我们提供了一个强大而灵活的框架，使我们可以轻松地响应和处理各种事件。这个类定义了一系列的方法，每一个都与LangChain中的一个特定事件相对应。只有深入理解了这些事件和方法，我们才能有效地为我们的应用编写自定义的回调处理器。

**1. LLM事件**

- on\_llm\_start : 当LLM启动并开始处理请求时，这个方法会被调用。这为我们提供了一个机会，例如，初始化某些资源或记录开始时间。

- on\_llm\_new\_token : 每当LLM生成一个新的令牌时，这个方法就会被执行。这在流式处理中特别有用，允许我们实时捕获和处理每一个生成的令牌。

- on\_llm\_end : 当LLM完成其任务并生成了完整的输出时，这个方法会被调用。这为我们提供了一个机会进行清理操作或记录任务的完成时间。

- on\_llm\_error : 如果在LLM处理过程中发生任何错误，这个方法将被执行。我们可以在这里添加错误日志或执行其他的错误处理操作。

**2. 聊天模型事件**

- on\_chat\_model\_start : 这个方法在Chat Model 类包装器开始其工作时被调用，为我们提供了一个机会进行初始化操作或其他准备工作。

**3. 链事件**

- on\_chain\_start : 当链开始执行时，这个方法会被调用。我们可以在这里进行一些初始化操作。

- on\_chain\_end : 在链完成所有任务后，这个方法会被执行。这为我们提供了一个机会进行清理操作或收集结果。

- on\_chain\_error : 如果链在执行过程中遇到错误，这个方法将被调用。这允许我们进行错误处理或日志记录。

**4. 工具事件**

- on\_tool\_start : 当工具开始其任务时，这个方法会被调用。

- on\_tool\_end : 在工具成功完成其任务后，这个方法会被执行。

- on\_tool\_error : 如果工具在执行过程中发生错误，这个方法将被执行。

**5. 其他事件**

- on\_text : 当需要处理任意文本时，这个方法会被调用。这为我们提供了一个机会对文本进行处理或分析。

- on\_agent\_action : 当代理执行某个特定的操作时，这个方法会被执行。

- on\_agent\_finish : 在代理完成其所有操作后，这个方法会被调用。

BaseCallbackHandler 为我们提供了一个强大的框架，使我们可以轻松地定义和处理各种事件。深入理解这个基础类是编写自定义回调处理器的关键。

自定义回调处理器

要 自定义自己的回调处理器，你可以继承 BaseCallbackHandler 并重写你需要的方法。例如，如果你想在大语言模型输出开始时打印一条消息，可以这样做：

class MyCallbackHandler(BaseCallbackHandler):

    def on\_llm\_start(self, serialized, prompts, kwargs):

        print("LLM has started!")

使用回调处理器

一旦你 自定义了自己的回调处理器，就可以将其传递给LangChain，以便在相应的事件发生时执行你的方法。

例如：

handler = MyCallbackHandler()

chain = LLMChain(llm=llm, prompt=prompt, callbacks=[handler])

## 8.2本章小结

在本章中，我们深入探讨了LangChain的回调处理器机制及其在实际应用中的价值。

回调处理器在LangChain中扮演着至关重要的角色。它是一种特殊的机制，允许开发者为不同的事件定义相应的方法。当LangChain中的某个特定事件被触发时，与之对应的回调处理器方法就会自动执行。这种自动响应特定事件的能力，使得开发者可以在合适的时机介入，执行各种操作，如日志记录、数据处理等。

我们通过代码示例详细介绍了如何使用LangChain内置的回调处理器，如StdOutCallbackHandler，AimCallbackHandler，WandbCallbackHandler等。这些内置处理器涵盖了各种常见的开发需求，从简单的日志记录到与第三方服务的集成，都可以轻松实现。

当然，LangChain的真正魔法在于它允许开发者创建自己的自定义回调处理器。我们详细探讨了如何设计并实现自定义的回调处理器，特别强调了理解链的生命周期和请求回调的重要性。深入BaseCallbackHandler这个基础类，我们了解到它定义了多个与LangChain事件相对应的方法，为开发者提供了一个强大的框架来响应和处理这些事件。

。