# 第3章





视频讲解

# 定制化聊天语言模型

第2章我们介绍了LangChain4j框架为开发者提供的聊天语言模型。这是构建企业级LLM应用的基础，但不是全部。在现实开发过程中，开发者需要基于业务场景需求完成对聊天语言模型的定制化开发，这时候就需要引入本章介绍的各个高级技术组件，包括如何使用聊天记忆功能为聊天模型添加状态性，如何使用Tool组件开发自定义的聊天逻辑，以及如何灵活且高效地控制聊天模型的输出过程及结果。

## 3.1 嵌入聊天记忆

在第2章中引入了AI服务组件来构建灵活的聊天功能。在与LLM进行聊天的过程中，有时候我们希望LLM能够记住与用户的对话内容，以便在对话中提供上下文相关的回答。这里的上下文可以包含对话历史、用户偏好、会话状态等信息。显然，在聊天过程中添加记忆功能能够提升用户体验，LangChain4j也为开发者提供了一个专门的组件，即ChatMemory。本节内容就将围绕这一话题展开讨论。

### 3.1.1 模型的状态性和聊天记忆机制

基于ChatLanguageModel，可以使用如例3-1所示的generate方法来触发对LLM的调用。

**【例3-1】** ChatLanguageModel中generate方法的定义。

```
1  Response<AiMessage> generate(List<ChatMessage> messages);
```

为什么这里需要提供一组而不是一个ChatMessage对象作为该方法输入的参数呢？这是因为LLM天生是无状态(Stateless)的，这意味着它们不维护对话的状态。因此，如果想要支持多轮对话，就应该注意管理对话的状态。

假设想要构建一个聊天机器人，那么想象一下用户和聊天机器人之间的一个简单多轮对话，如例3-2所示。

**【例3-2】** 简单多轮会话示例。

```
1  用户:你好,我叫天涯兰。
2  AI:嗨,天涯兰,我能帮你什么吗?
3  用户:我叫什么名字?
4  AI:天涯兰。
```

这时候，如果使用 ChatLanguageModel，交互过程应该如例 3-3 所示。

**【例 3-3】** ChatLanguageModel 多轮会话实现方式。

```
UserMessage firstUserMessage = UserMessage.from("你好,我叫天涯兰。");
AiMessage firstAiMessage = model.generate(firstUserMessage).content(); //嗨,天涯兰,我能
//帮你什么吗?

UserMessage secondUserMessage = UserMessage.from("我叫什么名字?");
AiMessage secondAiMessage = model.generate(firstUserMessage, firstAiMessage, secondUserMessage).
content(); //天涯兰
```

可以看到，在 generate 方法的第二次调用中不仅提供了单个 secondUserMessage，还提供了对话中的先前所有消息。

提供多个 ChatMessage 对象作为输入使得 LLM 能够理解对话的上下文和流程，从而能够在多个轮次上生成更相关和连贯的回复。但是，手动维护和管理这些消息是烦琐的。因此，LangChain4j 引入了 ChatMemory 这个概念。ChatMemory 在整个多轮对话中起到的作用如图 3-1 所示。

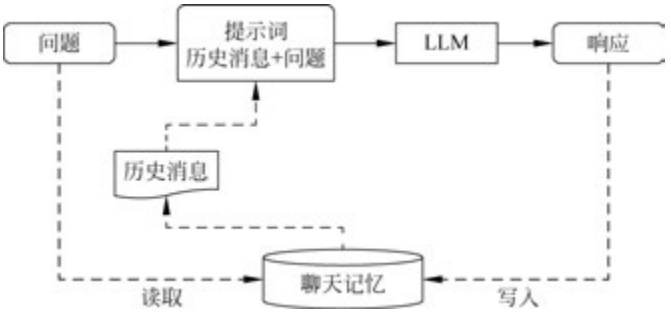



**图 3-1 ChatMemory 在多轮会话中的作用**

从图 3-1 可以看到，一个 ChatMemory 需要支持两个基本操作，即读取和写入。在多轮对话中，其中一些输入直接来自用户，但有些输入可能来自 ChatMemory。在一次典型的对话交互过程中，聊天模型至少要和 ChatMemory 进行两次交互。

(1) 在接收到初始用户输入之后但在执行核心逻辑之前，聊天模型将从 ChatMemory 中读取并扩充用户输入，这个步骤类似于缓存读取操作。

(2) 在执行核心逻辑之后但在返回响应之前，聊天模型将把当前运行的输入和输出写入 ChatMemory，以便在未来的运行中可以引用它们，这个步骤类似于缓存写入操作。

注意，聊天记忆(Memory)和聊天记录(History)是两个类似但又不同的概念。聊天记录保存用户和 LLM 之间的所有消息。在用户界面中，历史记录是用户看到的内容，它代表了实际说过的话。而聊天记忆保存了一些信息，这些信息被提供给 LLM 以使其表现得好像“记住”了对话。

根据使用的内存算法，ChatMemory 可以以多种方式修改聊天记录，常见的包括以下 4 种。

(1) 清除一些消息。

(2) 汇聚多个分开的消息。

(3) 从消息中删除不重要的细节。

(4) 向消息中注入额外信息(如用于 RAG 的信息)或指令(例如用于结构化输出的指令)等。

LangChain4j 目前仅提供聊天记忆功能，而不提供聊天记录。如果想要获取完整的历史

记录,需要开发者手动进行保存和处理。

### 3.1.2 ChatMemory 类型和使用方法

LangChain4j 为开发者提供了 ChatMemory 抽象以及多种开箱即用的实现。ChatMemory 可以配合 ChatLanguageModel 等独立的低阶组件使用,也可以作为 AI 服务等高阶组件的一部分。在本节内容中,我们将从 ChatMemory 的定义出发给出它的实现和使用方法。

#### 1. ChatMemory 的定义和实现

在 LangChain4j 中,ChatMemory 的定义如例 3-4 所示。

【例 3-4】 ChatMemory 的定义。

```
1  public interface ChatMemory {
2      Object id();
3
4      void add(ChatMessage message);
5
6      List<ChatMessage> messages();
7
8      void clear();
9  }
```

对于 ChatMemory 而言,开发者可以执行新增、查询和清除操作,同时每个 ChatMemory 都有一个唯一的 Id 与之关联。

LangChain4j 提供了两种即插即用的 ChatMemory 实现,即 MessageWindowChatMemory 和 TokenWindowChatMemory。读者可以参考这两个实现类中的变量定义来进行对比学习,如例 3-5 所示。

【例 3-5】 ChatMemory 两个实现类的变量定义。

```
 1  public class MessageWindowChatMemory implements ChatMemory {
 2      private static final Logger log = LoggerFactory.getLogger(MessageWindowChatMemory.class);
 3      private final Object id;
 4      private final Integer maxMessages;
 5      private final ChatMemoryStore store;
 6      ...
 7  }
 8
 9  public class TokenWindowChatMemory implements ChatMemory {
10      private static final Logger log = LoggerFactory.getLogger(TokenWindowChatMemory.
    class);
11      private final Object id;
12      private final Integer maxTokens;
13      private final Tokenizer tokenizer;
14      private final ChatMemoryStore store;
15      ...
16  }
```

可以看到,在 MessageWindowChatMemory 和 TokenWindowChatMemory 中都定义了一个 Integer 的变量代表最新的消息数和 Token 数。同时,TokenWindowChatMemory 中还有一个专门用于对文本内容进行分词的分词器(Tokenizer)组件。为什么要这样设计呢?这

就不得不介绍一个非常重要的概念，即消息的淘汰策略。在 ChatMemory 中引入淘汰策略有以下 3 个必要的原因。

（1）适应 LLM 的上下文窗口。LLM 一次处理的 Token 数量有上限。在某些情况下，对话可能会超出这个上限，这时候就应该淘汰一些消息。一般而言，最旧的消息会被淘汰，但如果需要的话也可以实现更复杂的算法。

（2）控制成本。每个 Token 都有成本，不断增加的聊天消息记录使得对 LLM 的每次调用逐渐变得更加昂贵。淘汰不必要的消息可以降低成本。

（3）控制延迟。发送给 LLM 的 Token 越多，处理它们所需的时间就越长。

MessageWindowChatMemory 和 TokenWindowChatMemory 这两个实现类底层都使用了滑动窗口机制，但具备不同的淘汰策略。MessageWindowChatMemory 相对简单，它作为一个滑动窗口保留最近的 $N$ 条消息，并淘汰不再适合的旧消息。然而，因为每条消息可能包含不同数量的 Token，所以 MessageWindowChatMemory 主要适用于快速原型开发。而更复杂的选择是 TokenWindowChatMemory，它保留最近的 $N$ 个 Token，并根据需要淘汰旧消息。消息是不可分割的，如果一条消息不符合，它将被完全淘汰。显然，TokenWindowChatMemory 需要一个分词器来计算每个 ChatMessage 中的 Token 数量，这就是该类中 Tokenizer 变量的作用。

ChatMemory 可以同时与 LangChain4j 中的低阶和高阶组件进行集成。在接下来的内容中，大家通过具体的示例来学习这个组件。

### 2. ChatMemory 的使用方法

先来看一个 ChatMemory 和 ChatLanguageModel 结合在一起使用的代码示例，如例 3-6 所示。

**【例 3-6】** ChatMemory 和 ChatLanguageModel 结合使用示例。

```
1  ChatLanguageModel model = OpenAiChatModel.withApiKey(ApiKeys.OPENAI_API_KEY);
2
3  ChatMemory chatMemory = TokenWindowChatMemory.withMaxTokens(300, new OpenAiTokenizer("gpt-
   3.5-turbo"));
4
5  chatMemory.add(userMessage("你好,我叫天涯兰。"));
6  AiMessage answer = model.generate(chatMemory.messages()).content();
7  System.out.println(answer.text()); //Hello Klaus! How can I assist you today?
8  chatMemory.add(answer);
9
10 chatMemory.add(userMessage("我叫什么名字?"));
11 AiMessage answerWithName = model.generate(chatMemory.messages()).content();
12 System.out.println(answerWithName.text());
13 chatMemory.add(answerWithName);
```

这里构建了一个 ChatLanguageModel 来实现聊天模型。然后，创建了一个 OpenAiTokenizer 并将它赋值给 TokenWindowChatMemory。在聊天过程中分别往这个 ChatMemory 中传入了 2 条 ChatMessage，执行之后得到的效果：

```
1  你好天涯兰,有什么可以帮助你的吗?
2  你的名字是天涯兰。
```

显然，上述代码过于低阶，开发者需要手工管理 ChatMemory 中的数据和状态。但这种做法确保开发者对 ChatMemory 有完全的控制权。如果不希望存储一些消息以节省 Token，

就可以决定是否要将特定消息添加到 Memory 中。而如果需要,还可以在保存消息之前对其进行修改或处理。

如果不需要对 ChatMemory 有过多的控制,而是希望把这部分工作交给 LangChaing4j,那么,通常的做法是把 ChatMemory 和 AI 服务整合在一起使用,示例代码如例 3-7 所示。

**【例 3-7】** ChatMemory 和 AI 服务结合使用示例。

```
public class ServiceWithMemory {

    interface Assistant {

        String chat(String message);
    }

    public static void main(String[] args) {

        ChatMemory chatMemory = MessageWindowChatMemory.withMaxMessages(10);

        Assistant assistant = AiServices.builder(Assistant.class)
                .chatLanguageModel(OpenAiChatModel.withApiKey(ApiKeys.OPENAI_API_KEY))
                .chatMemory(chatMemory)
                .build();

        String answer = assistant.chat("你好,我叫天涯兰。");
        System.out.println(answer);

        String answerWithName = assistant.chat("我叫什么名字?");
        System.out.println(answerWithName);
    }
}
```

上述代码具有同样的执行结果,但显然更为简单精练。开发者要做的事情就是构建一个 ChatMemory 实例然后赋值给 AiService 即可。

大家再看一个复杂一点的示例,如果希望为每个用户保存单独的聊天记忆,那么可以采用如例 3-8 所示的实现方法。

**【例 3-8】** 用户级别保存单独聊天记忆实现示例。

```
public class ServiceWithMemoryForEachUser {

    interface Assistant {

        String chat(@MemoryId int memoryId, @UserMessage String userMessage);
    }

    public static void main(String[] args) {

         Assistant assistant = AiServices.builder(Assistant.class)
                .chatLanguageModel(OpenAiChatModel.withApiKey(ApiKeys.OPENAI_API_KEY))
                .chatMemoryProvider(memoryId -> MessageWindowChatMemory.withMaxMessages(10))
                .build();

        System.out.println(assistant.chat(1, "你好,我叫天涯兰"));
        System.out.println(assistant.chat(2, "你好,我叫郑天民"));
        System.out.println(assistant.chat(1, "我叫什么名字?"));
```

```
        System.out.println(assistant.chat(2, "我叫什么名字?"));
    }
}
```

注意,在 Assistant 的 chat 方法中定义了一个 memoryId 字段,并在该字段上添加了@MemoryId 注解。添加了@MemoryId 注解的参数值将用于查找属于该用户/对话的 Memory,因此上述代码的执行之后可以得到的效果:

```
你好,天涯兰!有什么可以帮助你的吗?
你好,郑天民先生!有什么可以帮助您的吗?
您告诉我您的名字是天涯兰。您有什么问题或者需要帮助的吗?
您的名字是郑天民先生。您刚刚告诉我了您的名字。有什么其他问题需要我帮助您解决吗?
```

### 3.1.3 ChatMemory 持久化

在介绍 MessageWindowChatMemory 和 TokenWindowChatMemory 这两个 ChatMemory 实现类时,可以看到它们自带一个 ChatMemoryStore 组件,该组件用于实现聊天记录的持久化,如例 3-9 所示。

**【例 3-9】** ChatMemoryStore 的定义。

```
public interface ChatMemoryStore {

    List<ChatMessage> getMessages(Object memoryId);

    void updateMessages(Object memoryId, List<ChatMessage> messages);

    void deleteMessages(Object memoryId);
}
```

注意,这里的 updateMessages 方法在每次向 ChatMemory 添加新的 ChatMessage 时被调用,getMessages 方法在用户向 ChatMemory 请求所有消息时被调用,而 deleteMessages 方法在调用 ChatMemory.clear 方法时被调用。如果不想使用此功能,该方法可以为空。

默认情况下,ChatMemory 将 ChatMessage 存储在内存中,也就是实现了如例 3-10 所示的 InMemoryChatMemoryStore。

**【例 3-10】** InMemoryChatMemoryStore 的实现过程。

```
public class InMemoryChatMemoryStore implements ChatMemoryStore {
    private final Map<Object, List<ChatMessage>> messagesByMemoryId = new ConcurrentHashMap();

    public InMemoryChatMemoryStore() {
    }

    public List<ChatMessage> getMessages(Object memoryId) {
        return (List)this.messagesByMemoryId.computeIfAbsent(memoryId, (ignored) -> {
            return new ArrayList();
        });
    }

    public void updateMessages(Object memoryId, List<ChatMessage> messages) {
        this.messagesByMemoryId.put(memoryId, messages);
    }
```

```
16
17      public void deleteMessages(Object memoryId) {
18          this.messagesByMemoryId.remove(memoryId);
19      }
20  }
```

可以看到,默认的 InMemoryChatMemoryStore 的底层实际上就是一个 ConcurrentHashMap。

如果需要持久化,则可以实现自定义的 ChatMemoryStore 将消息存储在选择的任何持久化存储媒介中。例如,可以实现如例 3-11 所示的一个 PersistentChatMemoryStore。

**【例 3-11】** PersistentChatMemoryStore 的实现方式。

```
 1  static class PersistentChatMemoryStore implements ChatMemoryStore {
 2
 3          private final DB db = DBMaker.fileDB("chat-memory.db").transactionEnable().make();
 4          private final Map<String, String> map = db.hashMap("messages", STRING, STRING).
            createOrOpen();
 5
 6          @Override
 7          public List<ChatMessage> getMessages(Object memoryId) {
 8              String json = map.get((String) memoryId);
 9              return messagesFromJson(json);
10          }
11
12          @Override
13          public void updateMessages(Object memoryId, List<ChatMessage> messages) {
14              String json = messagesToJson(messages);
15              map.put((String) memoryId, json);
16              db.commit();
17          }
18
19          @Override
20          public void deleteMessages(Object memoryId) {
21              map.remove((String) memoryId);
22              db.commit();
23          }
24  }
```

这里基于 mapdb 这个快速、易用的嵌入式 Java 数据库引擎来构建持久化媒介。一旦创建了 PersistentChatMemoryStore,就可以把它赋值给 ChatMemory,如例 3-12 所示。

**【例 3-12】** 将 PersistentChatMemoryStore 赋值给 ChatMemory。

```
1  ChatMemory chatMemory = MessageWindowChatMemory.builder()
2          .maxMessages(10)
3          .chatMemoryStore(new PersistentChatMemoryStore())
4          .build();
```

关于 mapdb 的讨论不是本书的重点,读者只需要知道可以通过这些第三方框架实现任何所需的持久化策略即可。

## 3.2 定制聊天逻辑

在和 LLM 对话的过程中,有时候希望能够引入一些自定义的业务逻辑来干预 LLM 的返

回结果。通过这种做法，开发者可以极大地丰富对话的输出内容，并构建定制化的交互体验。本节内容将探讨这一话题，我们将引入 LangChain4j 中的 Tool 组件来实现这一目标。

### 3.2.1 引入 Tool 组件

本节内容的背景来自这样一个事实：有些 LLM 除了生成文本之外，还可以触发一定的动作(Action)。针对这一事实，在 LLM 中出现了一个被称为“工具(Tool)”或“函数调用(Function Calling)”的概念。因为中文的“工具”一词过于通用，很容易混淆，所以本书统一使用 Tool 这个英文名词来定义这个概念。这一概念允许 LLM 在必要时调用一个或多个可用 Tool 组件，这些 Tool 组件通常由开发者根据业务需求进行定义。

广义上讲，Tool 可以是任何事物，包括网络搜索、调用外部 API，或者执行特定的代码片段等。LLM 实际上不能自己调用 Tool，但它们可以在响应中表达调用特定 Tool 的意图，而不是以纯文本形式响应。作为开发者，应该基于业务参数来执行 Tool，并获取 Tool 执行的结果。

举个例子，LLM 本身并不擅长处理数学计算。如果应用程序涉及偶尔的数学计算，我们可能希望为 LLM 提供一个“数学 Tool”。通过在对 LLM 的请求中声明一个或多个 Tool，LLM 可以决定在必要时调用它们。给定一个数学问题以及一组数学 Tool，为了正确回答问题，LLM 会决定它应该首先调用其中哪个有效的数学 Tool。

Tool 的概念有点抽象，以下通过一些示例来进一步解释 Tool 的效果。假设现在向 LLM 发出如例 3-13 所示的聊天请求。

**【例 3-13】** LLM 聊天请求示例。

```
Request:
 - messages:
     - UserMessage:
         - text: 475695037565 的平方根是多少?
```

那么执行之后可以得到的结果：

```
Response:
 - AiMessage:
     - text: 475695037565 的平方根大约是 689710
```

可以看到响应结果虽然很接近，但并不正确，正确的结果应该是 689706.486532(保留小数点后 6 位)。

然后，创建 2 个 Tool，如例 3-14 所示。

**【例 3-14】** 创建 2 个 Tool 示例。

```
@Tool("对给定的两个值求和")
public double sum(double a, double b) {
    return a + b;
}

@Tool("返回给定值的平方根")
public double squareRoot(double x) {
    return Math.sqrt(x);
}
```

这里出现了@Tool 注解，会在本节后续内容中详细介绍这个注解的效果，现在大家只需要

明确可以用它来定义 Tool。有了这两个 Tool 之后,再次执行前面的请求,执行过程如例 3-15 所示。

【例 3-15】 基于 Tool 执行 LLM 聊天请求。

```
Request 1:
- messages:
    - UserMessage:
        - text: 475695037565 的平方根是多少?
- tools:
    - sum(double a, double b): 对给定的两个值求和
    - squareRoot(double x): 返回给定值的平方根

Response 1:
- AiMessage:
    - toolExecutionRequests:
        - squareRoot(475695037565)
```

现在,聊天模型会发起一个新的请求去执行 squareRoot 方法,并根据传入的参数获取对应的结果:

```
Request 2:
- messages:
    - UserMessage:
        - text: 475695037565 的平方根是多少?
    - AiMessage:
        - toolExecutionRequests:
            - squareRoot(475695037565)
    - ToolExecutionResultMessage:
        - text: 689706.486532

Response 2:
- AiMessage:
    - text: 475695037565 的平方根是 689706.486532
```

可以看到,当 LLM 可以访问一组 Tool 时,它可以根据场景选择合适的 Tool 进行调用。这是一个非常强大的功能。在这个示例中只是创建了简单的数学 Tool,但想象一下开发者可以通过这种方式实现任何复杂的定制逻辑,例如根据某个搜索 Tool 找到目标信息并把这些信息进行加工之后通过消息的形式发送给目标用户。

### 3.2.2 Tool 的使用方法

和聊天模型 ChatLanguageModel 一样,对于 LangChain4j 而言,Tool 的使用方法也有两类,一类是使用低阶 API,另一类则是使用高阶 API。

#### 1. 低阶 Tool API

针对低阶 Tool API,可以使用 ChatLanguageModel 中如例 3-16 所示的 generate 方法。

【例 3-16】 ChatLanguageModel 中包括 Tool 的 generate 方法。

```
generate(List<ChatMessage>, List<ToolSpecification>
```

注意到该方法中出现了一个新的 ToolSpecification 对象,该对象包含有关 Tool 的 3 条重要信息。

(1) Tool 的名称。

(2) Tool 的作用和何时应该使用的描述。

(3) 每个 Tool 所拥有参数的描述。

创建 ToolSpecification 有两种方法。一种是手工方法,示例代码如例 3-17 所示。

**【例 3-17】** 手工创建 ToolSpecification 示例。

```
ToolSpecification toolSpecification = ToolSpecification.builder()
    .name("getWeather")
    .description("返回指定城市的天气预报")
    .addParameter("city", type("string"), description("应返回天气预报的城市"))
    .addParameter("temperatureUnit", enums(TemperatureUnit.class))
    .build();
```

显然,这种方法比较烦琐,更为简单的方法是把 Tool 的定义和组合过程分离开来,如例 3-18 所示代码展示了这一实现过程。

**【例 3-18】** Tool 的定义和组合分离示例。

```
class WeatherTools {

    @Tool("返回指定城市的天气预报")
    String getWeather(
            @P("应返回天气预报的城市") String city,
            TemperatureUnit temperatureUnit
    ) {
        ...
    }
}

List < ToolSpecification > toolSpecifications = ToolSpecifications.toolSpecificationsFrom
(WeatherTools.class);
```

这里在 WeatherTools 类中通过@Tool 注解定义了一个 Tool,然后通过 ToolSpecifications 的 toolSpecificationsFrom 方法来构建 ToolSpecification 对象。

一旦成功构建了一组 ToolSpecification,那么就可以把它嵌入聊天模型中,如例 3-19 所示。

**【例 3-19】** ToolSpecification 集成聊天模型示例。

```
UserMessage userMessage = UserMessage.from("明天杭州的天气会怎么样?");
Response < AiMessage > response = model.generate(singletoneList(userMessage), toolSpecifications);
AiMessage aiMessage = response.content();
```

如果 LLM 选择调用某个 Tool,那么返回的 AiMessage 的 toolExecutionRequests 字段中将包含对应的请求信息。每个 ToolExecutionRequest 应该包含 3 个组成部分。

(1) Tool 的全局唯一 ID,如 call_CQQcqBiZ8gr4lrwj8igt6moz。

(2) Tool 的名称,如 getWeather。

(3) 参数,如{ "city": "London", "temperatureUnit": "CELSIUS" }。

如果想将 Tool 执行的结果发送回 LLM,那么就需要创建一个 ToolExecutionResultMessage,并将其与所有先前的消息一起发送,示例代码如例 3-20 所示。

【例 3-20】 将 Tool 执行结果发送回 LLM 示例。

```
String result = "预计明天杭州会下雨。";
ToolExecutionResultMessage toolExecutionResultMessage = ToolExecutionResultMessage.from
(toolExecutionRequest, result);
List<ChatMessage> messages = List.of(userMessage, aiMessage, toolExecutionResultMessage);
Response<AiMessage> response2 = model.generate(messages, toolSpecifications);
```

ToolExecutionResultMessage 也是 LangChain4j 内置的 4 种 ChatMessage 中的一种，关于这种消息类型的第一次讨论出现在本书 2.3 节中，大家可以做一些回顾。

**2. 高阶 Tool API**

如果使用高阶 Tool API，开发者可以用@Tool 注解标记任何 Java 方法，并将其与 AI 服务一起使用。AI 服务会自动将这种方法转换为 ToolSpecifications，并把它们包含在每次与 LLM 的交互请求中。当 LLM 决定调用 Tool 时，AI 服务将自动执行适当的方法，并将方法的返回值送回 LLM。

用@Tool 注解标记的方法可以接收各种类型的、任意数量的参数，而任何使用@Tool 注解标记并在构建 AI 服务时显式指定的 Java 方法都可以被 LLM 执行，例如前面介绍的数学 Tool 就可以采用如例 3-21 所示的代码进行实现。

【例 3-21】 @Tool 注解使用示例。

```
class MathCalculator{

    @Tool("对给定的两个值求和")
    public double add(int a, int b) {
        return a + b;
    }

    @Tool("返回给定值的平方根")
    public double squareRoot(double x) {
        return Math.sqrt(x);
    }
}
```

然后，创建一个 AI 服务来实现对话流程，如例 3-22 所示。

【例 3-22】 基于 AI 服务构建对话流程。

```
interface MathGenius {

    String ask(String question);
}

ChatLanguageModel model = OpenAiChatModel.withApiKey(ApiKeys.OPENAI_API_KEY);
MathGenius mathGenius = AiServices.builder(MathGenius.class)
    .chatLanguageModel(model)
    .tools(new Calculator())
    .build();

String answer = mathGenius.ask("475695037565 的平方根是多少?");

System.out.println(answer); //475695037565 的平方根是 689706.486532
```

在上述代码中，当调用 MathGenius 接口的 ask 方法时会发生两次与 LLM 的交互。在这两次交互之间，squareRoot 方法会被自动调用。这点和前面描述的一致。

@Tool 注解有两个可选字段，即代表 Tool 名称的 name 字段和代表 Tool 描述的 value 字段。根据 Tool 的具体功能，即使没有任何描述，LLM 也可能会很好地理解它，但通常建议提供清晰和有意义的名称和描述。

另外，方法参数可以选择使用@P 注解进行标注。@P 注解具有一个必填字段 value，用于提供参数的描述，例如例 3-23 所示的两个方法都包含了@P 注解。

**【例 3-23】** @P 注解使用示例。

```
@Tool("搜索有关给定查询的相关网址,请访问谷歌搜索。")
public List<String> searchGoogle(@P("search query") String query) {
    return googleSearchService.search(query);
}

@Tool("返回给定 URL 的网页内容。")
public String getWebPageContent(@P("URL of the page") String url) {
    Document jsoupDocument = Jsoup.connect(url).get();
    return jsoupDocument.body().text();
}
```

### 3.2.3 整合 Tool 和 ChatMemory

再来看一个示例，这个示例在构建 AI 服务的过程中综合使用了 Tool 和 ChatMemory，如例 3-24 所示。

**【例 3-24】** 综合使用 Tool 和 ChatMemory 示例。

```
public class StringCalculator{

    @Tool("计算字符串的长度。")
    int stringLength(String s) {
        System.out.println("调用 stringLength 方法:s = '" + s + "'");
        return s.length();
    }

    @Tool("计算两个数的和。")
    int add(int a, int b) {
        System.out.println("调用 add 方法:a = " + a + ", b = " + b);
        return a + b;
    }

    @Tool("计算一个数的平方根。")
    double sqrt(int x) {
        System.out.println("调用 sqrt 方法:x = " + x);
        return Math.sqrt(x);
    }
}

interface Assistant {

    String chat(String userMessage);
}

```

```
27  public static void main(String[] args) {
28
29      Assistant assistant = AiServices.builder(Assistant.class)
30          .chatLanguageModel(OpenAiChatModel.withApiKey(ApiKeys.OPENAI_API_KEY))
31          .tools(new Calculator())
32          .chatMemory(MessageWindowChatMemory.withMaxMessages(10))
33          .build();
34
35      String question = "单词 'hello' 和 'world' 的字母数之和的平方根是多少?";
36      String answer = assistant.chat(question);
37      System.out.println(answer);
38  }
```

执行上述代码得到的详细日志结果：

```
1  [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
2  DEBUG: About to execute ToolExecutionRequest { id = "call_CQQcqBiZ8gr4lrwj8igt6moz", name =
   "stringLength", arguments = "{"arg0":"hello"}" } for memoryId default
3  调用 stringLength 方法:s = 'hello'
4  [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
5  DEBUG: Tool execution result: 5
6  [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
7  DEBUG: About to execute ToolExecutionRequest { id = "call_8GftRafIOH4Dp9gJCBVJNq04", name =
   "stringLength", arguments = "{"arg0":"world"}" } for memoryId default
8  调用 stringLength 方法:s = 'world'
9  [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
10 DEBUG: Tool execution result: 5
11 [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
12 DEBUG: About to execute ToolExecutionRequest { id = "call_5vtNyVX3bpIvw9Mrfn6It7GB", name =
   "add", arguments = "{"arg0":5,"arg1":5}" } for memoryId default
13 调用 add 方法:a = 5, b = 5
14 [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
15 DEBUG: Tool execution result: 10
16 [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
17 DEBUG: About to execute ToolExecutionRequest { id = "call_i1qdtLAbnNuffL1s2qFDisdG", name =
   "sqrt", arguments = "{"arg0":10}" } for memoryId default
18 调用 sqrt 方法:x = 10
19 [main] dev.langchain4j.agent.tool.DefaultToolExecutor.execute()
20 DEBUG: Tool execution result: 3.1622776601683795
21 单词 'hello' 和 'world' 的字母数之和的平方根约是 3.162
```

可以看到，这里记录了每次调用 Tool 的日志，同时也展示了 ChatMemory 起到的记忆作用。如果 AI 服务所定义的方法包含用@MemoryId 注解标记的参数，那么开发者还可以使用@ToolMemoryId 注解标记@Tool 注解的参数。通过该注解，提供给 AI 服务方法的值将自动传递给@Tool 注解。这个功能有时候也非常有用。

作为总结，为了增加 LLM 正确识别和使用合适的 Tool，一个好的经验法则是：如果一个人能够理解 Tool 的目的以及如何使用它，那么 LLM 也很可能能够理解并使用它。

## 3.3 控制模型输出

至此，大家已经实现了聊天功能，并且能够实现定制化的聊天记忆和聊天逻辑。那么问题就来了，有没有办法在这些组件的基础上对聊天过程和结果进行更好的控制呢？这就是本节

要讨论的内容。借助 LangChain4j 强大的聊天模型，开发者可以很方便地获取各种模型数据，并实现对聊天过程的灵活控制。

### 3.3.1 控制聊天过程

本节将介绍 LangChain4j 针对聊天过程控制的 3 个非常实用的功能，包括流式控制、可观测性和日志管理。

#### 1. 流式控制

类似 ChatGPT 这样的 LLM 为用户提供了一种流式(Streaming)生成技术。LLM 会以一次生成一个 Token(token-by-token)的方式生成文本，因此许多 LLM 提供商提供了一种逐 Token 流式传输响应的方式，而不是等待整个文本生成完毕之后再将结果一次性传递给用户。流式机制显著提高了用户体验，因为用户无须等待未知长度的时间，几乎可以立即获取响应结果。

在 LangChain4j 中，针对 ChatLanguageModel，框架提供了对应流式版本的 StreamingChatLanguageModel 接口。该接口具有和 ChatLanguageModel 类似的 API，但可以流式传输响应。StreamingChatLanguageModel 接口的定义如例 3-25 所示。

**【例 3-25】** StreamingChatLanguageModel 的定义。

```
1  public interface StreamingChatLanguageModel {
2
3      default void generate(String userMessage, StreamingResponseHandler<AiMessage> handler) {
4          generate(singletonList(UserMessage.from(userMessage)), handler);
5      }
6
7      default void generate(UserMessage userMessage, StreamingResponseHandler<AiMessage> handler) {
8          generate(singletonList(userMessage), handler);
9      }
10
11     void generate(List<ChatMessage> messages, StreamingResponseHandler<AiMessage> handler);
12
13     default void generate(List<ChatMessage> messages, List<ToolSpecification>
   toolSpecifications, StreamingResponseHandler<AiMessage> handler) {
14         throw new IllegalArgumentException("Tools are currently not supported by this model");
15     }
16
17     default void generate(List<ChatMessage> messages, ToolSpecification toolSpecification,
   StreamingResponseHandler<AiMessage> handler) {
18         throw new IllegalArgumentException("Tools are currently not supported by this model");
19     }
20 }
```

与 ChatLanguageModel 类似，StreamingChatLanguageModel 同样提供了一组 generate 重载方法，而这些方法接收 StreamingResponseHandler 接口作为参数。StreamingResponseHandler 类似于一种回调机制，如例 3-26 所示。

**【例 3-26】** StreamingResponseHandler 的定义。

```
1  public interface StreamingResponseHandler<T> {
2      void onNext(String token);
3
```

```
    default void onComplete(Response<T> response) {
    }

    void onError(Throwable error);
}
```

可以看到，StreamingResponseHandler 定义了 3 个回调方法，分别对应聊天模型 3 个不同的执行阶段。通过实现 StreamingResponseHandler 接口，开发者可以定义针对 3 类事件的操作。

(1) 当生成下一个 Token 时，聊天模型将调用 onNext(String token)方法。例如，可以在 Token 可用时直接将其发送到用户界面。

(2) 当 LLM 完成生成时，将调用 onComplete(Response<T> response)方法。在这里，T 在使用 StreamingChatLanguageModel 时代表 AiMessage，而在使用 StreamingLanguageModel 时则代表 String。而 Response 对象包含完整的响应，本节后续内容会对这个 Response 对象进行详细的展开。

(3) 当发生错误时，聊天模型将调用 onError(Throwable error)方法，可以在这个方法中添加对异常的自定义处理。

对应地，StreamingChatLanguageModel 接口也有一组支持流式控制的实现类，例如 OpenAiStreamingChatModel。如例 3-27 所示的是如何使用 OpenAiStreamingChatModel 实现流式处理的示例。

**【例 3-27】** OpenAiStreamingChatModel 流式处理示例。

```
StreamingChatLanguageModel model = OpenAiStreamingChatModel.withApiKey(System.getenv
("OPENAI_API_KEY"));

String userMessage = "讲一个关于 Java 的笑话";

model.generate(userMessage, new StreamingResponseHandler<AiMessage>() {

    @Override
    public void onNext(String token) {
        System.out.println("onNext: " + token);
    }

    @Override
    public void onComplete(Response<AiMessage> response) {
        System.out.println("onComplete: " + response);
    }

    @Override
    public void onError(Throwable error) {
        error.printStackTrace();
    }
});
```

执行上述代码得到的结果：

```
onNext():
onNext(): 为
onNext(): 什
```

```
 4  onNext(): 么
 5  onNext(): Java
 6  onNext(): 程序
 7  onNext(): 员
 8  onNext(): 总
 9  onNext(): 是
10  onNext(): 喜
11  onNext(): 欢
12  onNext(): 喝
13  onNext(): 茶
14  onNext(): ?
15
16
17  onNext(): 因
18  onNext(): 为
19  onNext(): 他
20  onNext(): 们
21  onNext(): 喜
22  onNext(): 欢
23  onNext(): 在
24  onNext(): 代码
25  onNext(): 中
26  onNext(): 加
27  onNext(): 入
28  onNext(): 一
29  onNext(): 些
30  onNext(): 泡
31  onNext(): 茶
32  onNext(): 的
33  onNext(): 时间
34  onNext(): 。
35  onComplete(): Response {
36    "content": {
37       "AiMessage": {
38         "text": "为什么 Java 程序员总是喜欢喝茶?因为他们喜欢在代码中加入一些泡茶的时间。",
39         "toolExecutionRequests": null
40       }
41    },
42    "tokenUsage": {
43      "TokenUsage": {
44        "inputTokenCount": 16,
45        "outputTokenCount": 40,
46        "totalTokenCount": 56
47      }
48    },
49    "finishReason": "STOP"
50  }
```

基于上述结果，大家实际上也可以看出 OpenAI 拆分 Token 的过程。

当然，开发者可以构建一组 ChatMessage，然后把它们全部传给 OpenAiStreamingChatModel，如例 3-28 所示。

**【例 3-28】** OpenAiStreamingChatModel 传入一组 ChatMessage 示例。

```
1  List<ChatMessage> messages = asList(
2          systemMessage("你是个非常幽默的助手。"),
```

```
3          userMessage("讲一个关于 Java 的笑话。")
4  );
5
6  model.generate(messages , new StreamingResponseHandler < AiMessage >() {
7          ...
8  }
```

注意，如果想要体验 StreamingChatLanguageModel，那么必须使用真实的 ApiKey，LangChain4j 内置的演示 ApiKey 不支持流式控制。

**2. 可观测性**

如果想要理解 ChatLanguageModel 和 StreamingChatLanguageModel 在聊天过程中具体发生了什么，可以使用 ModelListener 组件对聊天过程进行观测。ModelListener 同样是一个接口，扮演了模型监听器的角色，能够实时捕获 ChatLanguageModel 和 StreamingChatLanguageModel 的运行状态。ModelListener 接口的定义如例 3-29 所示。

**【例 3-29】** ModelListener 的定义。

```
1  public interface ModelListener < Request, Response > {
2      @Experimental
3      default void onRequest(Request request) {
4      }

5      @Experimental
6      default void onResponse(Response response, Request request) {
7      }
8
9      @Experimental
10     default void onError(Throwable error, Response response, Request request) {
11     }
12 }
```

可以看到，ModelListener 的行为和 StreamingResponseHandler 有些类似，但它并不关注具体的 Token，而是对聊天模型的输入输出进行了监控。ModelListener 的使用方式如例 3-30 所示。

**【例 3-30】** ModelListener 的使用方式。

```
13 ModelListener < ChatLanguageModelRequest, ChatLanguageModelResponse > modelListener =
14         new ModelListener < ChatLanguageModelRequest, ChatLanguageModelResponse >() {
15
16                 @Override
17                 public void onRequest(ChatLanguageModelRequest request) {
18                         System.out.println("Request: " + request.messages());
19                 }
20
21                 @Override
22                 public void onResponse(ChatLanguageModelResponse response, ChatLanguageModelRequest
   request) {
23                       System.out.println("Response: " + response.aiMessage());
24                 }
25
26                 @Override
```

```
27                public void onError(Throwable error, ChatLanguageModelResponse response,
    ChatLanguageModelRequest request) {
28                        error.printStackTrace();
29                }
30            };
31
32    ChatLanguageModel model = OpenAiChatModel.builder()
33            .apiKey("demo")
34            .listeners(singletonList(modelListener))
35            .build();
36
37      model.generate("讲一个关于 Java 的笑话。");
```

执行上述代码得到的结果：

```
1   Request: [UserMessage { name = null contents = [TextContent { text = "讲一个关于 Java 的笑
    话。" }] }]
2   Response: AiMessage { text = "为什么程序员喜欢用 Java 写代码?因为 Java 能够让他们"咖啡"饮
    一整天!哈哈哈!" toolExecutionRequests = null }
```

这里出现的 Request 和 Response 分别是 ChatLanguageModelRequest 和 ChatLanguage ModelResponse 这组请求/响应对象，它们的定义如例 3-31 所示。

**【例 3-31】** ChatLanguageModelRequest 和 ChatLanguageModelResponse 的定义。

```
 1  public class ChatLanguageModelRequest {
 2      private final String model;
 3      private final Double temperature;
 4      private final Double topP;
 5      private final Integer maxTokens;
 6      private final List<ChatMessage> messages;
 7      private final List<ToolSpecification> toolSpecifications;
 8      ...
 9  }
10
11  public class ChatLanguageModelResponse {
12      private final String id;
13      private final String model;
14      private final TokenUsage tokenUsage;
15      private final FinishReason finishReason;
16      private final AiMessage aiMessage;
17      ...
18  }
```

不难看出，开发者通过上述对象可以获取 ChatLanguageModel 在运行过程中的大量元数据，这些元数据非常有用。通过这些元数据，开发者可以实现一套自定义的监控机制，从而更好地掌握模型的运行状态，并能够实现对核心数据的统计分析。

### 3. 日志管理

当然，如果想要掌握 ChatLanguageModel 在执行过程中的各种信息，也可以使用 LangChain4j 框架提供的日志功能。日志功能非常简单，直接使用 ChatLanguageModel 的 logRequests 和 logResponses 这两个工具方法即可，如例 3-32 所示。

**【例 3-32】** ChatLanguageModel 的日志方法。

```
OpenAiChatModel.builder()
    ...
    .logRequests(true)
    .logResponses(true)
    .build();
```

这里给出一个具体的示例，如例 3-33 所示。

【例 3-33】 ChatLanguageModel 的日志管理示例。

```
ChatLanguageModel model = OpenAiChatModel.builder()
    .apiKey(ApiKeys.OPENAI_API_KEY)
    .modelName("gpt-3.5-turbo")
    .temperature(0.3)
    .timeout(ofSeconds(60))
    .logRequests(true)
    .logResponses(true)
    .build();

String prompt = "解释如何在三行内制作一幅美丽的画作。";
String response = model.generate(prompt);
System.out.println(response);
```

执行上述代码得到的结果：

```
[main] dev.ai4j.openai4j.RequestLoggingInterceptor.logDebug()
DEBUG: Request:
- method: POST
- url: https://api.openai.com/v1/chat/completions
- headers: [Authorization: Bearer sk-sU...Tq], [User-Agent: langchain4j-openai]
- body: {
  "model": "gpt-3.5-turbo",
  "messages": [
    {
      "role": "user",
      "content": "解释如何在三行内制作一幅美丽的画作。"
    }
  ],
  "temperature": 0.3
}
[main] dev.ai4j.openai4j.ResponseLoggingInterceptor.logDebug()

DEBUG: Response:
- status code: 200
- headers: [content-type: application/json], [openai-organization: user-lektnzy5j70hyyomgmdez48t],
[openai-processing-ms: 1911], [openai-version: 2020-10-01], [strict-transport-security:
max-age=15552000; includeSubDomains; preload], [x-ratelimit-limit-requests: 10000], [x-
ratelimit-limit-tokens: 60000], [x-ratelimit-remaining-requests: 9999], [x-ratelimit-
remaining-tokens: 59969], [x-ratelimit-reset-requests: 8.64s], [x-ratelimit-reset-
tokens: 31ms], [x-request-id: req_1bb4d21f7bfdcf3058e3d00366c87a4c], [cf-cache-status:
DYNAMIC], [set-cookie: __cf_bm=kJVIcFQM3heeyyjCyKpvS.NV2h499SwRFvuk.pqzAw0-1721094754-
1.0.1.1-75BZG1Ut45j_0xHPjiu_KjPBnfvEjbes0Pj.6xULwVMtupanyA4PCPWoKHOSQIerfMPBu3j_
KVH9TjwTFBWSEQ; path=/; expires=Tue, 16-Jul-24 02:22:34 GMT; domain=.api.openai.com;
HttpOnly; Secure; SameSite=None], [x-content-type-options: nosniff], [set-cookie:
_cfuvid=nM7PNk0Do7DCGRXNUrds_HOv0vNwUgQWtTCc1f32vek-1721094754963-0.0.1.1-604800000;
path=/; domain=.api.openai.com; HttpOnly; Secure; SameSite=None], [server: cloudflare], [cf-
ray: 8a3e5a7d491b5275-LAX], [alt-svc: h3=":443"; ma=86400]
```

```
- body: {
  "id": "chatcmpl-9lRejAexopRDL5p93XEY7ZBAtfsf2",
  "object": "chat.completion",
  "created": 1721094753,
  "model": "gpt-3.5-turbo-0125",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "用丰富的色彩勾勒出一片绚丽的晚霞,映衬着远处的山峦和湖水。\n在画面中加入一只自由飞翔的彩虹鹦鹉,为整幅作品增添生机和活力。\n最后用细腻的笔触描绘出一株盛开的鲜花,点缀在画面中央,使整幅画作更加生动和美丽。"
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
   "usage": {
    "prompt_tokens": 27,
    "completion_tokens": 143,
    "total_tokens": 170
  },
  "system_fingerprint": null
}

用丰富的色彩勾勒出一片绚丽的晚霞,映衬着远处的山峦和湖水。
在画面中加入一只自由飞翔的彩虹鹦鹉,为整幅作品增添生机和活力。
最后用细腻的笔触描绘出一株盛开的鲜花,点缀在画面中央,使整幅画作更加生动和美丽。
```

可以看到,这里分别打印了 Request 和 Response 这两部分日志。从这些日志中可以看到 OpenAI 返回了很多有用的元数据,这些元数据对于开发者理解 LLM 的运行机制有一定的帮助。

### 3.3.2 响应结果和 OutputParser

针对 ChatLanguageModel 的返回结果,开发者可以控制它的响应过程。这个过程需要开发者对 LangChain4j 中的两个核心对象有一定的了解,即 Response 对象和 OutputParser 对象。

#### 1. Response

Response 是聊天模型响应结果的包装器,经常会看到它作为各种 ChatLanguageModel 调用方法的返回类型,该类的定义如例 3-34 所示。

**【例 3-34】** Response 的定义。

```
public class Response<T> {
    private final T content;
    private final TokenUsage tokenUsage;
    private final FinishReason finishReason;
```

```
    ...
}
```

可以看到，除了代表响应内容的 content 字段之外，Response 还包含有关对话过程中所生成的一些元数据。首先，它包括 TokenUsage，该类定义了本次交互的输入（提供给 generate 方法的所有 ChatMessages）包含多少个 Token、输出生成了多少个 Token（包含在 AiMessage 中）以及输入＋输出的总 Token 数统计信息，如例 3-35 所示。

**【例 3-35】** TokenUsage 中的字段定义。

```
public class TokenUsage {
    private final Integer inputTokenCount;
    private final Integer outputTokenCount;
    private final Integer totalTokenCount;
    ...
}
```

在商用环境下，LLM 的调用成本是必须要考虑的一个因素，开发者可以利用该信息来计算调用 LLM 的成本。

另外，Response 还包含一个 FinishReason 对象，这是一个定义各种生成停止原因的枚举。FinishReason 枚举定义如例 3-36 所示。

**【例 3-36】** FinishReason 枚举定义。

```
public enum FinishReason {
    STOP,                //模型调用结束，因为模型认定请求已完成
    LENGTH,              //模型调用结束，因为达到了 Token 可用长度
    TOOL_EXECUTION,      //模型调用结束，表明需要执行工具
    CONTENT_FILTER,      //模型调用结束，表明需要内容过滤
    OTHER;               //模型调用结束，出于其他原因
}
```

通常情况下，如果 LLM 决定停止生成内容，这个枚举值将是 FinishReason. STOP。

### 2. OutputParser

LLM 输出文本，但很多时候开发者希望获得比文本更结构化的信息。这就是输出解析器（OutputParser）的作用。和 LangChain 一样，LangChain4j 也设计了一个 OutputParser 接口用来对模型的输出结果进行解析，该接口如例 3-37 所示。

**【例 3-37】** OutputParser 接口定义。

```
public interface OutputParser<T> {

    T parse(String text);

    String formatInstructions();
}
```

可以看到，OutputParser 接口非常简单，核心就是一个支持泛型的 parse 方法，该方法用来接收聊天模型的响应字符串并将其解析为某种数据结构。

在 LangChain4j 中内置了一组 OutputParser 接口的实现类，分别面向不同的数据类型。这些实现类都比较简单，例如 DoubleOutputParser 和 LocalDateTimeOutputParser 类的实现

过程如例 3-38 所示。

【例 3-38】 DoubleOutputParser 和 LocalDateTimeOutputParser 的实现过程。

```
public class DoubleOutputParser implements OutputParser<Double> {
    public DoubleOutputParser() {
    }

    public Double parse(String string) {
        return Double.parseDouble(string);
    }

    public String formatInstructions() {
        return "floating point number";
    }
}

public class LocalDateTimeOutputParser implements OutputParser<LocalDateTime> {
    public LocalDateTimeOutputParser() {
    }

    public LocalDateTime parse(String string) {
        return LocalDateTime.parse(string, DateTimeFormatter.ISO_LOCAL_DATE_TIME);
    }

    public String formatInstructions() {
        return "yyyy-MM-ddTHH:mm:ss";
    }
}
```

OutputParser 的使用方法也很明确，就是将 parse 方法作用于 Response 所返回的字符串内容，示例代码如例 3-39 所示。

【例 3-39】 OutputParser 使用示例。

```
ChatLanguageModel model = ...;
String response = model.generate(prompt);

OutputParser<Double> parser = new DoubleOutputParser();
Double result = parser.parse(response);
```

原则上，开发者可以根据需要对 OutputParser 进行扩展，从而实现各种自定义的 OutputParser 组件。开发者要做的事情就是把来自模型的响应字符串进行解析，从而构建符合业务需求的自定义对象即可。自定义 OutputParser 往往与 AI 服务组件整合在一起使用，这样就可以将 AI 服务调用的返回类型从 String 更改为其他类型。当前，AI 服务支持以下 9 种返回类型。

（1）String。

（2）AiMessage。

（3）boolean/Boolean，如果需要获取“是”或“否”的答案。

（4）byte/Byte/short/Short/int/Integer/BigInteger/long/Long/float/Float/double/Double/BigDecimal。

（5）Date/LocalDate/LocalTime/LocalDateTime。

（6）List < String >/Set < String >，如果希望以项目符号列表的形式获取答案。

（7）任何枚举，如果希望对文本进行分类，例如情感、用户意图等。

（8）任何自定义 POJO（Plain Old Java Object）。

（9）Result < T >，如果需要访问 TokenUsage 等内容。

除了返回类型为 String、AiMessage 或 Response < AiMessage >外，AI 服务将自动在 UserMessage 的末尾附加指示 LLM 应该以哪种格式回复的说明。在方法返回之前，AI 服务将 LLM 的输出解析为所需的类型。通过这种方式，OutputParser 就能和 AI 服务进行无缝集成。