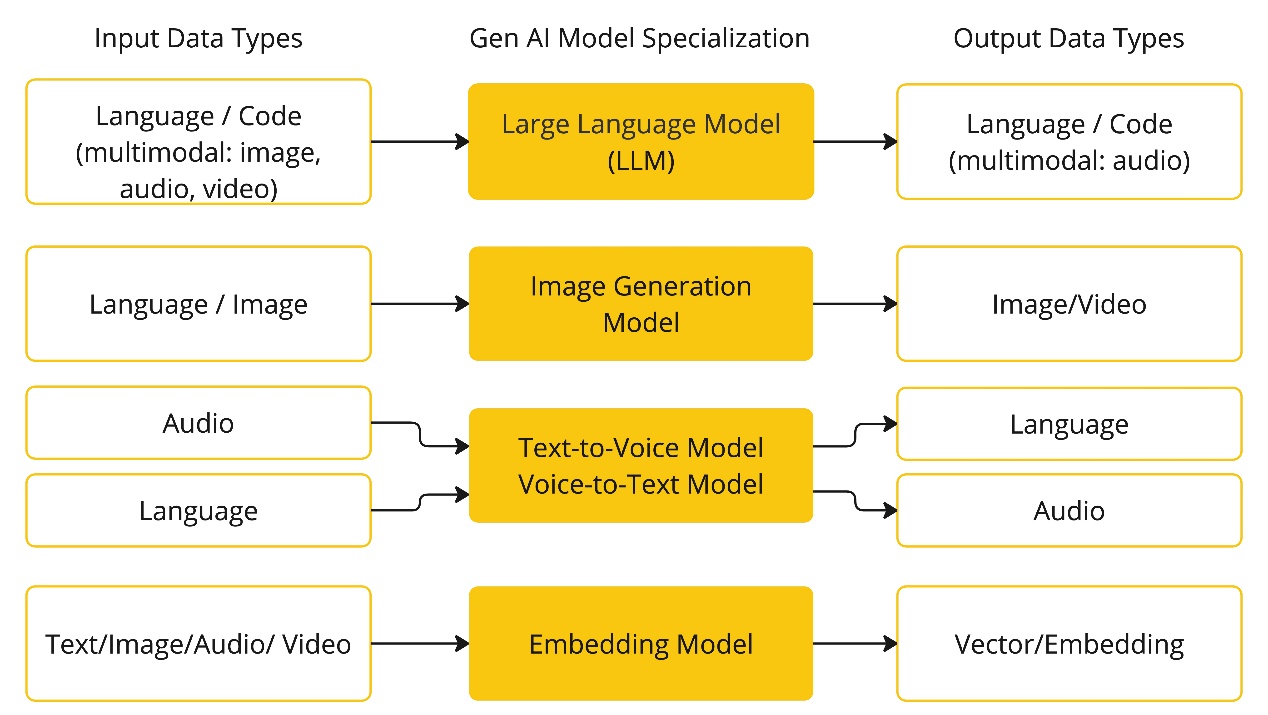
# [**AI 概念**](https://springdoc.cn/spring-ai/concepts.html)

**Model（模型）**

人工智能模型（AI Model）是一种用于处理和生成信息的算法，通常模仿人类的认知功能。通过从大型数据集中学习模式和见解，这些模型可以做出预测、文本、图像或其他输出，从而增强各行业的各种应用。

有许多不同类型的 AI 模型，每种适用于特定的用例。虽然 ChatGPT 及其生成式 AI 能力通过文本输入和输出吸引了用户，但许多模型和公司提供多样化的输入和输出。在 ChatGPT 之前，许多人对文本转图像生成模型，如 Midjourney 和 Stable Diffusion，感到着迷。

下表根据输入和输出类型对几种模型进行了分类：



Spring AI 目前支持处理语言、图像和音频输入和输出的模型。上表中的最后一行接受文本作为输入并输出数字，通常被称为嵌入文本（**Embedding Text**），代表人工智能模型中使用的内部数据结构。Spring AI 支持嵌入，以实现更高级的用例。

GPT 等模型的与众不同之处在于其预训练特性，正如 GPT-Chat Generative Pre-trained Transformer 中的 “P” 所表示的那样。这种预训练功能将人工智能转变为一种通用开发工具，不需要广泛的机器学习或模型训练背景。

**Prompt（提示）**

提示（Prompt）是引导人工智能模型产生特定输出的语言输入的基础。对于熟悉 ChatGPT 的人来说，提示可能只是在对话框中输入并发送到 API 的文本。然而，它包含的内容远不止这些。在许多人工智能模型中，提示文本不仅仅是一个简单的字符串。

ChatGPT 的 API 在一个提示中包含多个文本输入，每个文本输入都被分配了一个角色。例如，system 角色会告诉模型如何操作，并为交互设置上下文。还有 user 角色，通常是来自用户的输入。

编写高效的提示既是一门艺术，也是一门科学。ChatGPT 专为人类对话而设计。这与使用类似 SQL 的东西来进行 “查询/对话” 完全不同。我们必须像与人对话一样与人工智能模型交流。

由于这种交互方式的重要性，“Prompt Engineering（提示工程）” 一词已成为一门独立的学科。提高提示有效性的技术层出不穷。在制作提示语方面投入时间，可以大大提高结果输出。

分享提示语已经成为一种共同的做法，学术界也在积极开展这方面的研究。最近的 [一篇研究论文](https://arxiv.org/abs/2205.11916) 发现，最有效的提示语之一是以 “深呼吸，一步一步来” 开头的。这应该能让你明白为什么语言如此重要。我们至今仍未完全掌握如何充分利用 ChatGPT 3.5 等早期迭代版本的技术，更遑论正在开发中的新一代模型了。

**提示模板**

设计优质提示词的关键在于构建请求的上下文框架，并将部分通用描述替换为用户输入的具体参数值。

该流程采用基于文本的传统模板引擎来实现提示词的创建与管理。为此，Spring AI 使用了开源库 [StringTemplate](https://www.stringtemplate.org/) 作为技术实现方案。

例如，一个简单的提示词模板如下：

Tell me a {adjective} joke about {content}.

Copied!

在 Spring AI 中，提示词模板可类比 Spring MVC 架构中的 “View” 层。系统会提供一个模型对象（通常是 java.util.Map）来填充模板中的占位符，最终 “渲染” 生成的字符串将作为传递给 AI 模型的提示内容。

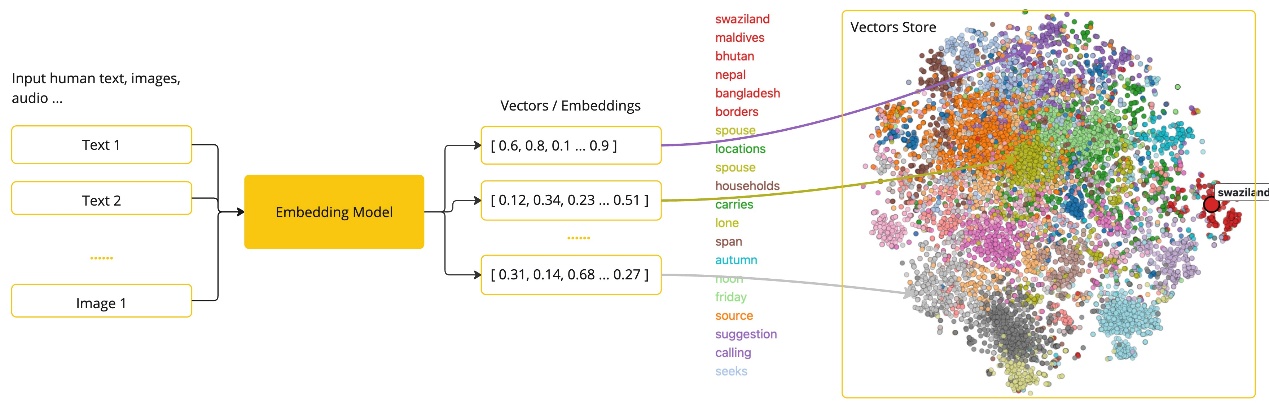
传递给 AI 模型的提示数据格式存在显著差异性。从最初简单的字符串形式，提示词已发展为包含多消息结构的复合形态 — 其中每条消息的字符串都对应着模型需要扮演的特定角色。

**Embedding（嵌入）**

嵌入是将文本、图像或视频转化为数值表示的技术，能够捕捉输入数据之间的关联性。

嵌入通过将文本、图像和视频转换为浮点数数组（称为向量）来工作。这些向量旨在捕捉文本、图像和视频的含义。嵌入数组的长度称为向量的维度。

通过计算两段文本向量表示之间的数值距离，应用程序即可判定原始对象在嵌入向量空间中的相似程度。



作为探索 AI 的 Java 开发者，你无需深究向量表示背后的复杂数学理论或具体实现细节。只需理解它们在 AI 系统中的角色和功能即可，特别是在将 AI 功能集成到应用程序时。

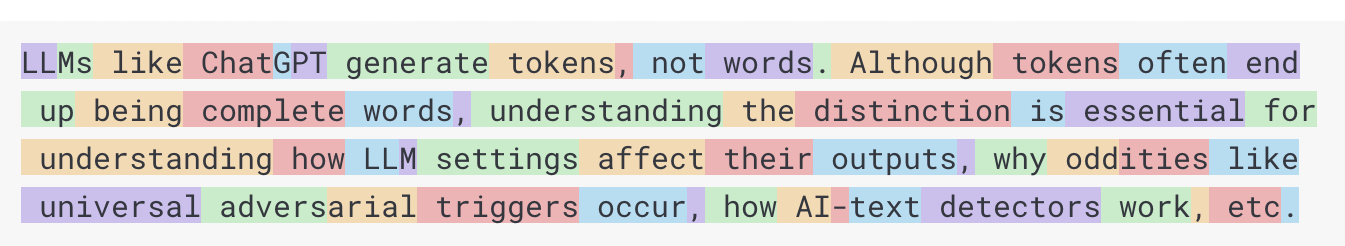
嵌入技术在检索增强生成（RAG）等实际应用中尤为重要。它们将数据表示为语义空间中的点 — 类似于欧氏几何的二维空间，但维度更高。就像欧氏几何中点的坐标决定其远近，语义空间中点的距离反映含义的相似性：话题相近的句子在多维空间中的位置更接近，如同图表上相邻的数据点。这种邻近性助力文本分类、语义搜索及产品推荐等任务，使 AI 能根据概念在这个扩展语义 “地图” 中的 “坐标” 来识别和归类相关内容。

你可以把这个语义空间想象成一个向量。

**Token**

令牌（Token）是 AI 模型运作的基础单元。输入时，模型将词语转换为令牌；输出时，再将令牌转换回词语。

在英语中，1 个 Token 大约对应 0.75 个单词。作为参考，莎士比亚全集约 90 万单词，对应约 120 万 Token。



更关键的是：Token 数量直接决定费用。在使用托管式 AI 模型时，系统会根据消耗的 Token 总量计费 — 无论是输入还是输出的 Token 都会被计入总用量。

此外，模型存在 Token 数量限制（即 “上下文窗口”），这会约束单次 API 调用中可处理的文本量。超出该阈值的文本内容将被直接截断，模型不会进行处理。

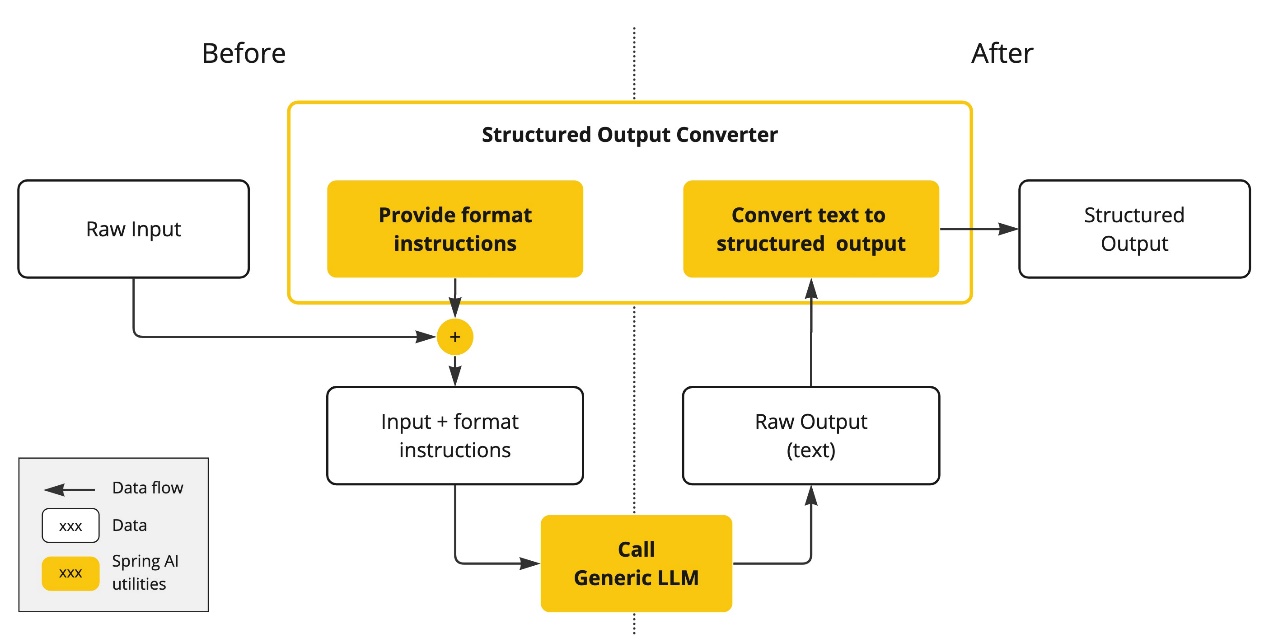
例如，ChatGPT3 的 Token 上限为 4K，而 GPT4 提供 8K、16K 和 32K 等不同规格。Anthropic 的 Claude AI 模型支持 100K Token，Meta 最新研究则实现了 100 万 Token 的突破性容量。

若要用 GPT4 概括莎士比亚全集，你需要设计软件工程策略来分割数据，确保内容不超出模型的上下文窗口限制。Spring AI 项目正为此类任务提供解决方案。

**结构化输出**

AI 模型的输出传统上以 java.lang.String 形式返回 — 即便要求生成 JSON 格式的答复。它可能是格式正确的 JSON，但本质仍是字符串而非 JSON 数据结构。要注意，在提示词中简单要求 “输出 JSON” 并不能百分百保证结果准确性。

这一复杂性催生了一个专门领域：既要设计能生成预期输出的提示词，又需将返回的原始字符串转换为可供应用程序集成的数据结构。



[结构化输出](https://springdoc.cn/spring-ai/api/structured-output-converter.html#_structuredoutputconverter) 转换采用精心制作的提示，通常需要与模型进行多次交互才能实现所需的格式。

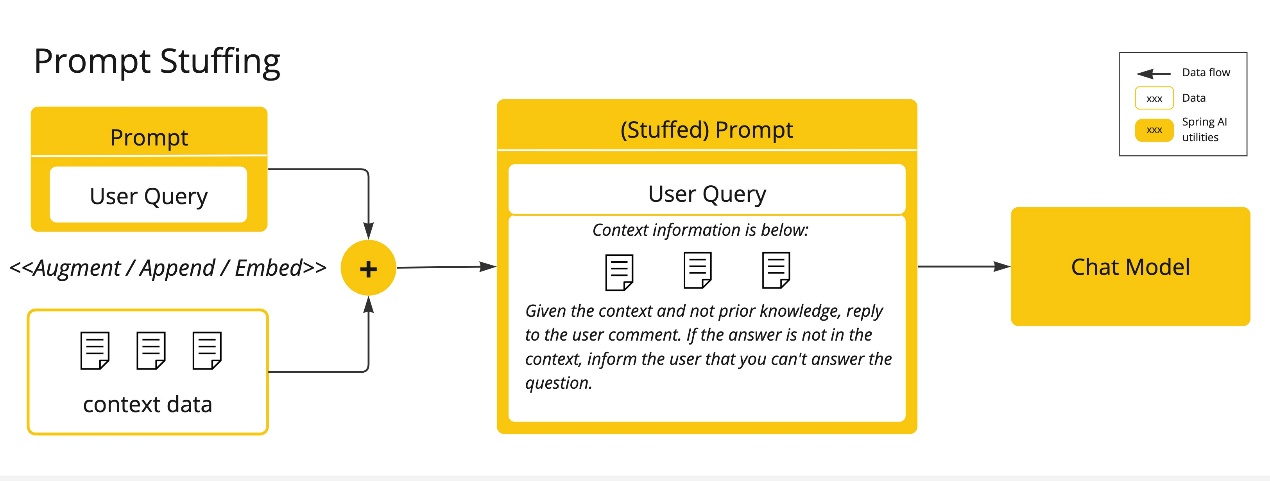
**将你的数据与 API 接入 AI 模型**

如何为 AI 模型提供其未经训练的信息？

需注意，GPT-3.5/4.0 的训练数据仅更新至 2021 年 9 月。因此，对于需要该日期之后知识的提问，模型会回答 “不知道”。有趣的是，该训练数据集大小约为 650GB。

现有三种技术可定制 AI 模型以整合你的数据：

* **微调（Fine-Tuning）**：这项传统机器学习技术通过调整模型内部权重实现定制，但对 GPT 等大模型而言存在双重挑战 — 不仅需要专业机器学习知识，还因模型规模导致计算资源消耗极大。值得注意的是，部分模型可能根本不开放此功能。
* **提示词填充（Prompt Stuffing）**：一种更实用的替代方案是将数据直接嵌入到提供给模型的提示词中。鉴于模型的 Token 限制，需要特定技术确保相关数据能适配上下文窗口。这种方法俗称 “提示词填充”。Spring AI 库能帮助你基于该技术（现多称为检索增强生成/RAG）实现解决方案。



* [**工具调用（Tool Calling）**](https://springdoc.cn/spring-ai/concepts.html#concept-fc)：该技术支持注册工具（用户自定义服务），将大语言模型与外部系统 API 连接。Spring AI 大幅简化了实现 [工具调用](https://springdoc.cn/spring-ai/api/tools.html) 所需的代码量。

**检索增强生成（RAG）**

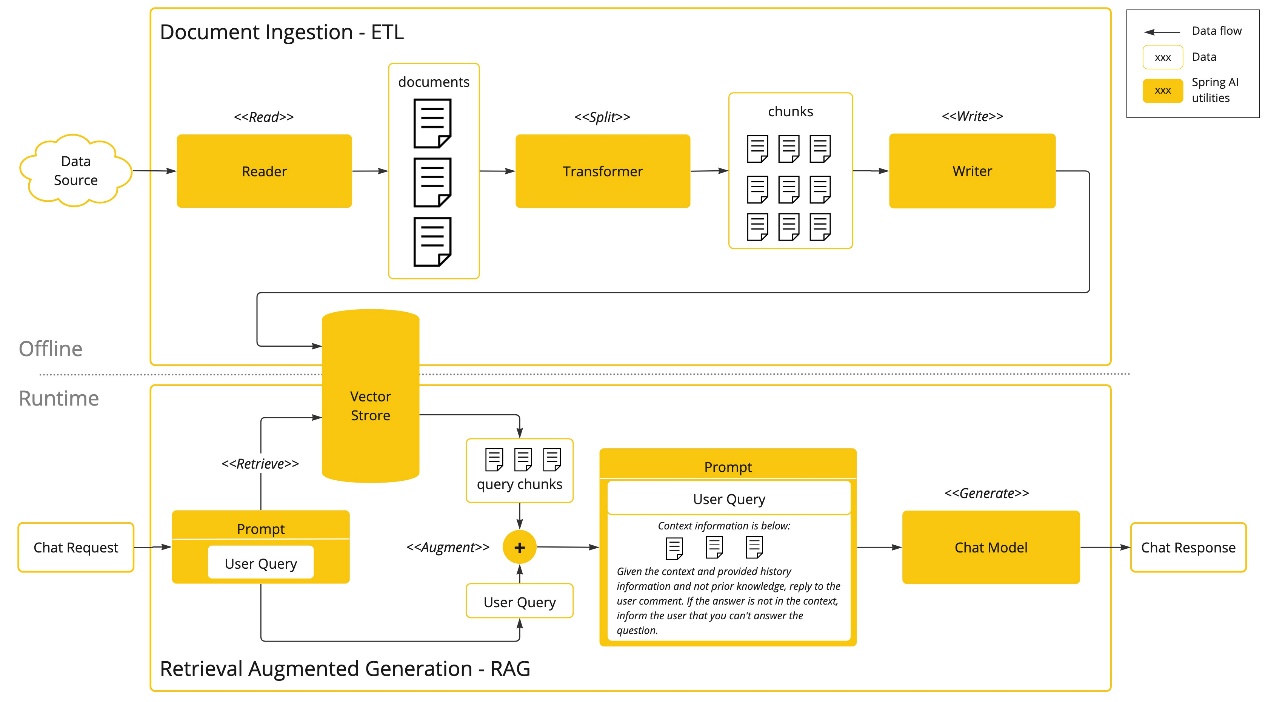
检索增强生成（RAG）技术应运而生，专门解决将相关数据整合到提示词中以获取准确 AI 响应的难题。

该技术采用批处理编程模型：首先从文档中读取非结构化数据，经转换后写入向量数据库。本质上这是一个 ETL（抽取-转换-加载）流程，而向量数据库正是 RAG 技术中检索环节的核心组件。

将非结构化数据加载到向量数据库时，关键转换步骤之一是将原始文档分割成小块。这个分割过程包含两个重要环节：

1. 在保留内容语义边界的前提下，将文档分割成若干部分。例如，对于包含段落和表格的文档，应避免在段落或表格中间分割文档。对于代码，应避免在方法实现的中间部分分割代码。
2. 将初步分割后的文档块进一步细分，确保每个块的体积不超过 AI 模型 Token 限制的较小百分比（如15%-20%）。这种精细化处理既能适配模型上下文窗口，又最大限度保留语义连贯性。

RAG 的下一阶段是处理用户输入。当需要 AI 模型回答用户问题时，系统会将问题与所有 “相似” 的文档片段一起放入提示词中发送给模型。这正是使用向量数据库的原因 — 它极其擅长快速定位语义相似的内容。



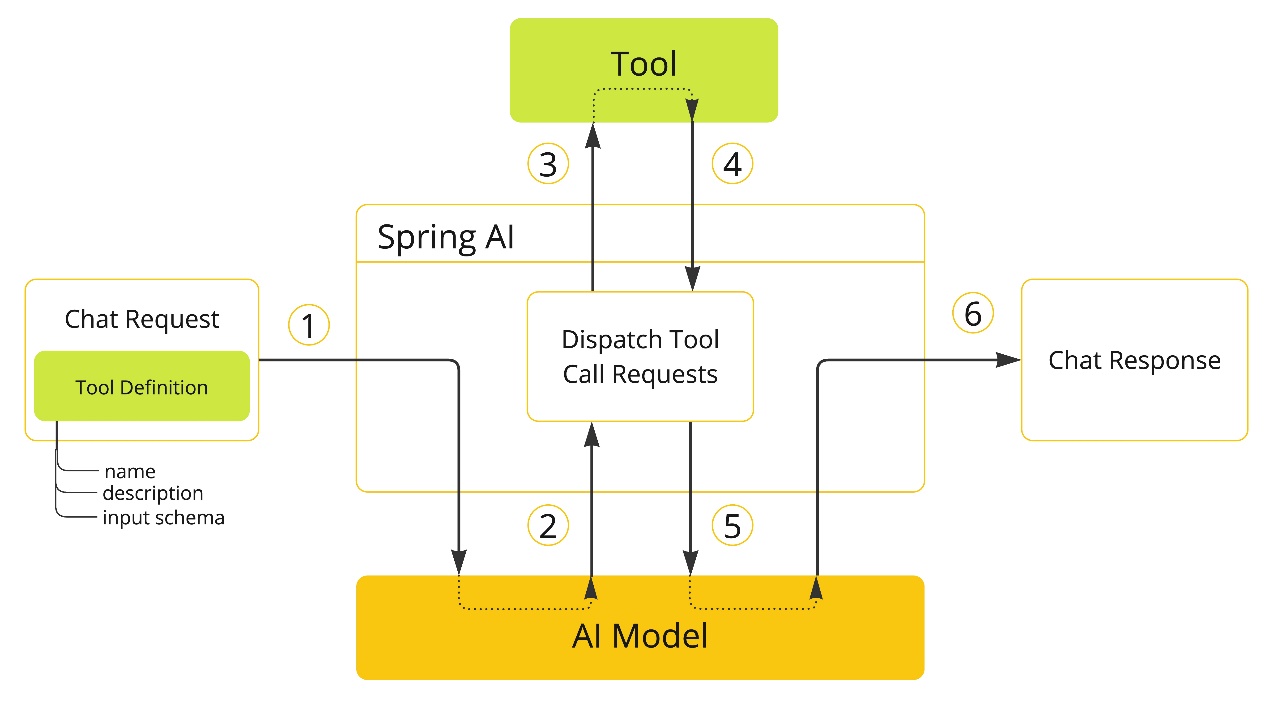
* [ETL Pipeline](https://springdoc.cn/spring-ai/api/etl-pipeline.html)：该管道协调从数据源提取数据、将其存储至结构化向量库的完整流程，确保数据在传递给 AI 模型时处于最佳检索格式。通过系统化的抽取-转换-加载操作，实现非结构化数据到高效可检索形式的精准转化。
* [ChatClient - RAG](https://springdoc.cn/spring-ai/api/chatclient.html#_retrieval_augmented_generation)：该组件说明如何通过 QuestionAnswerAdvisor 在应用中启用检索增强生成功能，实现智能问答与上下文感知的响应生成。

**工具调用**

大语言模型（LLM）在训练完成后即固化，导致知识陈旧，且无法直接访问或修改外部数据。

[工具调用机制（Tool Calling）](https://springdoc.cn/spring-ai/api/tools.html) 有效解决了这些局限。该功能允许你将自定义服务注册为工具，将大语言模型与外部系统 API 连接，使 LLM 能获取实时数据并委托这些系统执行数据处理操作。

Spring AI 极大简化了支持工具调用所需的编码工作，自动处理工具调用的对话交互。你只需将工具定义为带有 @Tool 注解的方法，并通过提示选项提供给模型即可调用。此外，单个提示中可定义和引用多个工具。



1. 要让模型能够调用某个工具，需要在聊天请求中包含该工具的定义。每个工具定义包含名称、描述以及输入参数的 Schema。
2. 当模型决定调用工具时，它会返回一个响应，其中包含工具名称和按照定义 Schema 建模的输入参数。
3. 应用程序负责根据工具名称识别并执行对应工具，同时使用提供的输入参数进行操作。
4. 工具调用的结果由应用程序负责处理。
5. 应用程序将工具调用的结果返回给模型。
6. 模型利用工具调用结果作为额外上下文生成最终响应。

有关如何在不同 AI 模型中使用此功能的详细信息，请参阅 [工具调用](https://springdoc.cn/spring-ai/api/tools.html) 文档。

**评估 AI 响应**

有效评估 AI 系统对用户请求的响应输出，对于确保最终应用的准确性和实用性至关重要。目前已有多种新兴技术可利用预训练模型自身实现这一目标。

该评估流程需要分析生成内容是否契合用户意图及查询上下文，通过相关性、连贯性和事实准确性等指标来衡量 AI 响应的质量。

一种方法是将用户请求和 AI 模型响应同时提交给模型，询问该响应是否符合所提供的数据。

此外，利用向量数据库中存储的信息作为补充数据可以增强评估过程，有助于确定响应的相关性。

Spring AI 项目提供了 Evaluator API，目前支持基础策略来评估模型响应。更多信息请参阅 [评估测试](https://springdoc.cn/spring-ai/api/testing.html) 文档。

## 入门

**Spring Initializr**

前往 [start.springboot.io](https://start.springboot.io/)，选择要在新应用程序中使用的 AI 模型和向量存储。

**组件仓库**

**里程碑 - 使用 Maven Central**

从 1.0.0-M6 开始，Maven Central 中提供了发布版本。无需更改构建文件。

**快照 - 添加 Snapshot 仓库**

要使用快照（和 1.0.0-M6 之前的里程碑）版本，需要在构建文件中添加以下快照仓库。

将以下 Repository 定义添加到 Maven 或 Gradle 构建文件中：

* **Maven**
* **Gradle**

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>

<releases>

<enabled>false</enabled>

</releases>

</repository>

<repository>

<name>Central Portal Snapshots</name>

<id>central-portal-snapshots</id>

<url>https://central.sonatype.com/repository/maven-snapshots/</url>

<releases>

<enabled>false</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

</repositories>

Copied!

**NOTE:** 使用 Maven 构建 Spring AI 快照版本时，请特别注意镜像配置。若你的 settings.xml 文件中配置了如下镜像：

<mirror>

<id>my-mirror</id>

<mirrorOf>\*</mirrorOf>

<url>https://my-company-repository.com/maven</url>

</mirror>

Copied!

通配符 \* 会将所有仓库请求重定向至镜像，导致无法访问 Spring 快照仓库。需修改 mirrorOf 配置排除 Spring 仓库：

<mirror>

<id>my-mirror</id>

<mirrorOf>\*,!spring-snapshots,!central-portal-snapshots</mirrorOf>

<url>https://my-company-repository.com/maven</url>

</mirror>

Copied!

此配置允许 Maven 直接访问 Spring 快照仓库，同时其他依赖仍通过镜像获取。

**依赖管理**

Spring AI 物料清单（BOM）声明了指定版本所有依赖的推荐版本。此为纯 BOM 版本，仅包含依赖管理，不涉及插件声明或 Spring/Spring Boot 的直接引用。你可使用 Spring Boot Parent POM 或 Spring Boot 的 BOM（spring-boot-dependencies）来管理 Spring Boot 版本。

添加 BOM 到项目：

* **Maven**
* **Gradle**

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.ai</groupId>

<artifactId>spring-ai-bom</artifactId>

<version>1.0.0-SNAPSHOT</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

## **聊天客户端 API**

ChatClient 通过 Fluent API 与 AI 模型交互，同时支持同步和流式编程模型。

|  |  |
| --- | --- |
|  | 有关 ChatClient 中命令式与响应式编程模型结合使用的实现，请参阅本文档底部的 [说明](https://springdoc.cn/spring-ai/api/chatclient.html#_implementation_notes)。 |

Fluent API 提供构建 [Prompt](https://springdoc.cn/spring-ai/api/prompt.html#_prompt) 组件的方法，这些组件将作为输入传递给 AI 模型。Prompt 包含指导 AI 模型输出和行为的指令文本。从 API 角度看，提示词由消息集合构成。

AI 模型处理两类主要消息：用户消息（直接来自用户的输入）和系统消息（由系统生成以引导对话）。

这些消息通常包含占位符，运行时根据用户输入进行替换，从而定制 AI 模型对用户输入的响应。

还可以指定一些 Prompt 选项，如要使用的 AI 模型名称，以及控制生成输出随机性/创造性的 temperature 参数。

**创建 ChatClient**

ChatClient 通过 ChatClient.Builder 对象创建。你可获取 Spring Boot 自动配置的 [ChatModel](https://springdoc.cn/spring-ai/api/chatmodel.html) 对应的 ChatClient.Builder 实例，或以编程式自行构建。

**使用自动配置的 ChatClient.Builder**

在最简单的使用场景中，Spring AI 通过 Spring Boot 自动配置生成 ChatClient.Builder Prototype Bean，可直接注入类中使用。以下是获取用户简单请求 String 响应的基础示例：

@RestController

class MyController {

private final ChatClient chatClient;

public MyController(ChatClient.Builder chatClientBuilder) {

this.chatClient = chatClientBuilder.build();

}

@GetMapping("/ai")

String generation(String userInput) {

return this.chatClient.prompt()

.user(userInput)

.call()

.content();

}

}

Copied!

此示例中，userInput 为用户消息内容。call() 方法向 AI 模型发送请求， content() 方法以 String 形式返回模型响应。

**多聊天模型协作**

单一应用中需使用多个聊天模型的典型场景包括：

* 不同任务类型选用不同模型（如复杂推理用高性能模型，简单任务用快速经济型模型）
* 主模型服务不可用时启用备用机制
* 不同模型或配置的 A/B 测试
* 根据用户偏好提供可选的模型
* 组合专用模型（如代码生成与创意内容分别使用不同模型）

Spring AI 默认自动配置单个 ChatClient.Builder Bean，但应用中可能需要使用多个聊天模型。处理方法如下：

所有场景均需通过设置属性 spring.ai.chat.client.enabled=false 来禁用 ChatClient.Builder 自动配置。

该设置允许手动创建多个 ChatClient 实例。

**单一模型类型下的多ChatClient实例**

本节涵盖常见场景：需创建多个使用相同底层模型类型但配置不同的 ChatClient 实例。

*// 以编程式创建 ChatClient实 例*

ChatModel myChatModel = ... *// 已由Spring Boot自动配置完成*

ChatClient chatClient = ChatClient.create(myChatModel);

*// 或使用 Builder 实现更精细控制*

ChatClient.Builder builder = ChatClient.builder(myChatModel);

ChatClient customChatClient = builder

.defaultSystemPrompt("You are a helpful assistant.")

.build();

Copied!

**不同模型类型的 ChatClient 配置**

使用多 AI 模型时，可为每个模型定义独立的 ChatClient Bean：

import org.springframework.ai.chat.ChatClient;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class ChatClientConfig {

@Bean

public ChatClient openAiChatClient(OpenAiChatModel chatModel) {

return ChatClient.create(chatModel);

}

@Bean

public ChatClient anthropicChatClient(AnthropicChatModel chatModel) {

return ChatClient.create(chatModel);

}

}

Copied!

随后可通过 @Qualifier 注解将这些 Bean 注入应用组件：

@Configuration

public class ChatClientExample {

@Bean

CommandLineRunner cli(

@Qualifier("openAiChatClient") ChatClient openAiChatClient,

@Qualifier("anthropicChatClient") ChatClient anthropicChatClient) {

return args -> {

var scanner = new Scanner(System.in);

ChatClient chat;

*// Model selection*

System.out.println("\nSelect your AI model:");

System.out.println("1. OpenAI");

System.out.println("2. Anthropic");

System.out.print("Enter your choice (1 or 2): ");

String choice = scanner.nextLine().trim();

if (choice.equals("1")) {

chat = openAiChatClient;

System.out.println("Using OpenAI model");

} else {

chat = anthropicChatClient;

System.out.println("Using Anthropic model");

}

*// Use the selected chat client*

System.out.print("\nEnter your question: ");

String input = scanner.nextLine();

String response = chat.prompt(input).call().content();

System.out.println("ASSISTANT: " + response);

scanner.close();

};

}

}

Copied!

**多 OpenAI 兼容 API 端点**

OpenAiApi 与 OpenAiChatModel 类提供的 mutate() 方法，支持基于现有实例创建不同属性的变体，特别适用于需对接多个 OpenAI 兼容 API 的场景。

@Service

public class MultiModelService {

private static final Logger logger = LoggerFactory.getLogger(MultiModelService.class);

@Autowired

private OpenAiChatModel baseChatModel;

@Autowired

private OpenAiApi baseOpenAiApi;

public void multiClientFlow() {

try {

*// Derive a new OpenAiApi for Groq (Llama3)*

OpenAiApi groqApi = baseOpenAiApi.mutate()

.baseUrl("https://api.groq.com/openai")

.apiKey(System.getenv("GROQ\_API\_KEY"))

.build();

*// Derive a new OpenAiApi for OpenAI GPT-4*

OpenAiApi gpt4Api = baseOpenAiApi.mutate()

.baseUrl("https://api.openai.com")

.apiKey(System.getenv("OPENAI\_API\_KEY"))

.build();

*// Derive a new OpenAiChatModel for Groq*

OpenAiChatModel groqModel = baseChatModel.mutate()

.openAiApi(groqApi)

.defaultOptions(OpenAiChatOptions.builder().model("llama3-70b-8192").temperature(0.5).build())

.build();

*// Derive a new OpenAiChatModel for GPT-4*

OpenAiChatModel gpt4Model = baseChatModel.mutate()

.openAiApi(gpt4Api)

.defaultOptions(OpenAiChatOptions.builder().model("gpt-4").temperature(0.7).build())

.build();

*// Simple prompt for both models*

String prompt = "What is the capital of France?";

String groqResponse = ChatClient.builder(groqModel).build().prompt(prompt).call().content();

String gpt4Response = ChatClient.builder(gpt4Model).build().prompt(prompt).call().content();

logger.info("Groq (Llama3) response: {}", groqResponse);

logger.info("OpenAI GPT-4 response: {}", gpt4Response);

}

catch (Exception e) {

logger.error("Error in multi-client flow", e);

}

}

}

Copied!

**ChatClient Fluent 风格 API**

ChatClient Fluent 式 API 通过重载 prompt 方法提供三种提示词创建方式：

* `prompt()：无参方法启动 Fluent 式API，支持逐步构建用户消息、系统消息等提示词组件。
* prompt(Prompt prompt)：接收 Prompt 参数，支持通过非 Fluent 式 API 构建的 Prompt 实例。
* prompt(String content)：便捷方法，接收用户文本内容，功能类似前项重载。

**ChatClient 响应**

ChatClient API 通过 Fluent 式接口提供多 种AI 模型响应格式化方式。

**返回 ChatResponse**

AI 模型返回的 [ChatResponse](https://springdoc.cn/spring-ai/api/chatmodel.html#ChatResponse) 是包含丰富信息的结构化响应：

* **元数据**：响应生成详情
* **多响应支持**：[Generations](https://springdoc.cn/spring-ai/api/chatmodel.html#Generation) 数组（每个含独立元数据）
* **Token 统计**：约 3/4 单词计为 1 个 Token（计费依据）

下面是在 call() 方法后调用 chatResponse() 返回包含元数据的 ChatResponse 对象的示例。

ChatResponse chatResponse = chatClient.prompt()

.user("Tell me a joke")

.call()

.chatResponse();

Copied!

**返回实体**

通常需要将返回的 String 映射为实体类，entity() 方法正提供此功能。

例如，给定一个 Java record：

record ActorFilms(String actor, List<String> movies) {}

Copied!

如下所示，通过 entity() 方法可轻松将 AI 模型输出映射至该 record 类：

ActorFilms actorFilms = chatClient.prompt()

.user("Generate the filmography for a random actor.")

.call()

.entity(ActorFilms.class);

Copied!

另提供 entity 重载方法 entity(ParameterizedTypeReference<T> type)，支持泛型集合等复杂类型指定：

List<ActorFilms> actorFilms = chatClient.prompt()

.user("Generate the filmography of 5 movies for Tom Hanks and Bill Murray.")

.call()

.entity(new ParameterizedTypeReference<List<ActorFilms>>() {});

Copied!

**流式响应**

stream() 方法支持异步获取响应，示例如下：

Flux<String> output = chatClient.prompt()

.user("Tell me a joke")

.stream()

.content();

Copied!

也可通过 chatResponse() 方法以 Flux<ChatResponse> 流式获取响应。

未来版本将提供便捷方法支持响应式流直接返回 Java 实体。当前版本需通过 [Structured Output Converter](https://springdoc.cn/spring-ai/api/structured-output-converter.html#StructuredOutputConverter)（结构化输出转换器）显式聚合响应，如下所示。这也演示了 Fluent API 中参数的使用，我们将在后面的文档中详细讨论。

var converter = new BeanOutputConverter<>(new ParameterizedTypeReference<List<ActorsFilms>>() {});

Flux<String> flux = this.chatClient.prompt()

.user(u -> u.text("""

Generate the filmography for a random actor.

{format}

""")

.param("format", this.converter.getFormat()))

.stream()

.content();

String content = this.flux.collectList().block().stream().collect(Collectors.joining());

List<ActorFilms> actorFilms = this.converter.convert(this.content);

Copied!

**提示模版**

ChatClient Fuent 式 API 支持提供含变量的用户/系统消息模板，运行时进行替换。

String answer = ChatClient.create(chatModel).prompt()

.user(u -> u

.text("Tell me the names of 5 movies whose soundtrack was composed by {composer}")

.param("composer", "John Williams"))

.call()

.content();

Copied!

ChatClient 内部使用 PromptTemplate 类处理用户/系统文本，依赖 TemplateRenderer 实现运行时变量替换。Spring AI 默认采用基于 Terence Parr 开发的 [StringTemplate](https://www.stringtemplate.org/) 引擎的 StTemplateRenderer 实现。

Spring AI 还提供 NoOpTemplateRenderer，用于无需模板处理的场景。

Spring AI 还提供 NoOpTemplateRenderer 实现。

|  |  |
| --- | --- |
|  | 通过 .templateRenderer() 在 ChatClient 上直接配置的 TemplateRenderer 仅作用于构建链（Builder Chain）中直接定义的提示内容（如 .user() / .system()）。它不会影响 [Advisor](https://springdoc.cn/spring-ai/api/retrieval-augmented-generation.html#_questionansweradvisor)（如 QuestionAnswerAdvisor）内部使用的模板 — 这些模板有独立的定制机制（参见 [自定义Advisor模板](https://springdoc.cn/spring-ai/api/retrieval-augmented-generation.html#_custom_template) 章节）。 |

如需改用其他模板引擎，可直接向 ChatClient 提供 TemplateRenderer 接口的自定义实现。也可保留默认 StTemplateRenderer 但进行自定义配置。

例如，默认模板变量采用 {} 语法。若提示词中包含 JSON，建议改用 <> 等分隔符避免冲突。示例如下：

String answer = ChatClient.create(chatModel).prompt()

.user(u -> u

.text("Tell me the names of 5 movies whose soundtrack was composed by <composer>")

.param("composer", "John Williams"))

.templateRenderer(StTemplateRenderer.builder().startDelimiterToken('<').endDelimiterToken('>').build())

.call()

.content();

Copied!

**call() 返回值**

ChatClient 的 call() 方法调用后，响应类型有以下几种处理选项：

* String content()：返回响应的字符串内容
* ChatResponse chatResponse()：返回包含多组生成结果及元数据（如消耗的 Token 数）的 ChatResponse 对象。
* ChatClientResponse chatClientResponse()：返回包含 ChatResponse 对象和 ChatClient 执行上下文的 ChatClientResponse 对象，可访问 Advisor 执行期间的附加数据（如 RAG 流程中检索的相关文档）。
* entity() 返回 Java 类型
  + entity(ParameterizedTypeReference<T> type)：用于返回实体类型的集合（Collection）。
  + entity(Class<T> type)：用于返回特定实体类型。
  + entity(StructuredOutputConverter<T> structuredOutputConverter)：通过 StructuredOutputConverter 将 String 转为实体类型。

也可使用 stream() 方法替代 call()。

**stream() 返回值**

ChatClient 的 stream() 方法调用后，响应类型有以下处理选项：

* Flux<String> content()：返回 AI 模型生成字符串的 Flux 流。
* Flux<ChatResponse> chatResponse()：返回包含响应元数据的 ChatResponse 对象 Flux 流。
* Flux<ChatClientResponse> chatClientResponse()：返回包含 ChatResponse 对象和 ChatClient 执行上下文的 ChatClientResponse 对象 Flux 流，可访问 Advisor 执行期间的附加数据（如 RAG 流程检索的相关文档）。

**使用默认值**

在 @Configuration 类中为 ChatClient 配置默认 system（系统）消息可简化运行时代码。预设默认值后，调用时仅需指定 user 消息，无需每次请求重复设置系统消息。

**默认的系统消息**

以下示例将系统消息配置为始终以海盗口吻回复。为避免在运行时代码中重复系统消息，我们将在 @Configuration 类中创建 ChatClient 实例。

@Configuration

class Config {

@Bean

ChatClient chatClient(ChatClient.Builder builder) {

return builder.defaultSystem("You are a friendly chat bot that answers question in the voice of a Pirate")

.build();

}

}

Copied!

并通过 @RestController 调用：

@RestController

class AIController {

private final ChatClient chatClient;

AIController(ChatClient chatClient) {

this.chatClient = chatClient;

}

@GetMapping("/ai/simple")

public Map<String, String> completion(@RequestParam(value = "message", defaultValue = "Tell me a joke") String message) {

return Map.of("completion", this.chatClient.prompt().user(message).call().content());

}

}

Copied!

通过 curl 调用应用端点时，返回结果如下：

❯ curl localhost:8080/ai/simple

{"completion":"Why did the pirate go to the comedy club? To hear some arrr-rated jokes! Arrr, matey!"}

Copied!

**带参数的默认系统消息**

以下示例将在系统消息中使用占位符，以便在运行时（而非设计时）指定回复语气。

@Configuration

class Config {

@Bean

ChatClient chatClient(ChatClient.Builder builder) {

return builder.defaultSystem("You are a friendly chat bot that answers question in the voice of a {voice}")

.build();

}

}

Copied!

@RestController

class AIController {

private final ChatClient chatClient;

AIController(ChatClient chatClient) {

this.chatClient = chatClient;

}

@GetMapping("/ai")

Map<String, String> completion(@RequestParam(value = "message", defaultValue = "Tell me a joke") String message, String voice) {

return Map.of("completion",

this.chatClient.prompt()

.system(sp -> sp.param("voice", voice))

.user(message)

.call()

.content());

}

}

Copied!

通过 httpie 调用应用端点时，返回结果如下：

http localhost:8080/ai voice=='Robert DeNiro'

{

"completion": "You talkin' to me? Okay, here's a joke for ya: Why couldn't the bicycle stand up by itself? Because it was two tired! Classic, right?"

}

Copied!

**其他默认配置**

可在 ChatClient.Builder 层级指定默认提示词配置。

* defaultOptions(ChatOptions chatOptions)：支持传入 ChatOptions 定义的通用选项或 OpenAiChatOptions 等模型专属选项。详见各模型 ChatOptions 实现的 JavaDoc 文档。
* defaultFunction(String name, String description, java.util.function.Function<I, O> function)：
  + name：函数在用户文本中的引用标识
  + description：描述函数用途，辅助AI模型精准选择函数
  + function：模型按需调用的 Java Function 实例
* defaultFunctions(String…​ functionNames)： 应用上下文中定义的 java.util.Function 的 Bean 名称列表。
* defaultUser(String text)、defaultUser(Resource text)、defaultUser(Consumer<UserSpec> userSpecConsumer)： 这些方法用于定义用户消息。Consumer<UserSpec> 支持通过 lambda 指定用户消息及默认参数。
* defaultAdvisors(Advisor…​ advisor)： Advisor 机制支持修改用于创建 Prompt 的数据。QuestionAnswerAdvisor 实现通过追加与用户消息相关的上下文信息，启用检索增强生成（RAG）模式。
* defaultAdvisors(Consumer<AdvisorSpec> advisorSpecConsumer)： 该方法允许通过 Consumer 使用 AdvisorSpec 配置多个 Advisor。Advisor 可修改最终 Prompt 的生成数据。Consumer<AdvisorSpec> 支持以 lambda 形式添加 Advisor（如 QuestionAnswerAdvisor），该 Advisor 基于用户消息追加相关上下文信息，实现检索增强生成（RAG）模式。

可通过不带 default 前缀的对应方法在运行时覆盖这些默认配置。

* options(ChatOptions chatOptions)
* function(String name, String description, java.util.function.Function<I, O> function)
* functions(String…​ functionNames)
* user(String text)、user(Resource text)、user(Consumer<UserSpec> userSpecConsumer)
* advisors(Advisor…​ advisor)
* advisors(Consumer<AdvisorSpec> advisorSpecConsumer)

**Advisor**

[Advisor API](https://springdoc.cn/spring-ai/api/advisors.html) 为 Spring 应用中的 AI 驱动交互提供灵活强大的拦截、修改和增强能力。

调用 AI 模型时，常见模式是在用户消息基础上追加或增强上下文数据。

此类上下文数据可分为多种类型，常见包括：

* **自有数据**：AI 模型未训练过的数据。即使模型接触过类似数据，追加的上下文数据仍会优先影响响应生成。
* **对话历史**：聊天模型 API 是无状态的。若告知 AI 模型你的姓名，后续交互中它不会记住。必须每次请求都发送对话历史，确保生成响应时考虑先前交互。

**ChatClient 中的 Advisor 配置**

ChatClient Fluent 式 API 提供 AdvisorSpec 接口用于配置 Advisor。该接口支持添加参数、批量设置参数以及向链中添加单个或多个 Advisor。

interface AdvisorSpec {

AdvisorSpec param(String k, Object v);

AdvisorSpec params(Map<String, Object> p);

AdvisorSpec advisors(Advisor... advisors);

AdvisorSpec advisors(List<Advisor> advisors);

}

Copied!

|  |  |
| --- | --- |
|  | Advisor 加入链的顺序至关重要，它决定了它们的执行顺序。每个 Advisor 都会以某种方式修改提示词或上下文，且一个 Advisor 所做的更改会传递给链中的下一个 Advisor。 |

ChatClient.builder(chatModel)

.build()

.prompt()

.advisors(

MessageChatMemoryAdvisor.builder(chatMemory).build(),

QuestionAnswerAdvisor.builder(vectorStore).build()

)

.user(userText)

.call()

.content();

Copied!

在此配置中，MessageChatMemoryAdvisor 将首先执行，将对话历史添加到提示词中。随后，QuestionAnswerAdvisor 将基于用户问题和添加的对话历史执行搜索，可能提供更相关的结果。

[了解 QuestionAnswerAdvisor](https://springdoc.cn/spring-ai/api/retrieval-augmented-generation.html#_questionansweradvisor)

**检索增强生成**

请参阅 [R检索增强生成指南](https://springdoc.cn/spring-ai/api/retrieval-augmented-generation.html)。

**日志**

SimpleLoggerAdvisor 是一个记录 ChatClient 请求和响应数据的 Advisor，可用于调试和监控 AI 交互。

|  |  |
| --- | --- |
|  | Spring AI 支持对 LLM 和向量存储交互的可观测性。详情请参阅 [可观测性指南](https://springdoc.cn/spring-ai/observability/index.html)。 |

启用日志记录需在创建 ChatClient 时向 Advisor 链添加 SimpleLoggerAdvisor，建议将其添加至链的末端：

ChatResponse response = ChatClient.create(chatModel).prompt()

.advisors(new SimpleLoggerAdvisor())

.user("Tell me a joke?")

.call()

.chatResponse();

Copied!

要查看日志，请将 advisor 包的日志级别设为 DEBUG：

logging.level.org.springframework.ai.chat.client.advisor=DEBUG

将此配置添加到 application.properties 或 application.yaml 文件中。

可通过以下构造函数自定义 AdvisedRequest 和 ChatResponse 的日志记录内容：

SimpleLoggerAdvisor(

Function<AdvisedRequest, String> requestToString,

Function<ChatResponse, String> responseToString

)

Copied!

示例用法：

SimpleLoggerAdvisor customLogger = new SimpleLoggerAdvisor(

request -> "Custom request: " + request.userText,

response -> "Custom response: " + response.getResult()

);

Copied!

这允许你根据具体需求定制日志信息。

|  |  |
| --- | --- |
|  | 在生产环境中需谨慎记录敏感信息。 |

**聊天记忆**

ChatMemory 接口定义了聊天对话记忆的存储机制，提供添加消息、检索消息及清空对话历史的方法。

当前内置实现为：MessageWindowChatMemory。

MessageWindowChatMemory 是聊天记忆实现，维护最多指定数量（默认 20 条）的消息窗口。当消息超出限制时，旧消息会被移除（系统消息除外）。若添加新系统消息，则清除所有旧系统消息，确保始终保留最新上下文的同时控制内存占用。

MessageWindowChatMemory 基于 ChatMemoryRepository 抽象层实现，该抽象层提供多种聊天记忆存储方案，包括：

* InMemoryChatMemoryRepository（内存存储）
* JdbcChatMemoryRepository（JDBC 关系型数据库存储）
* CassandraChatMemoryRepository（Cassandra 存储）
* Neo4jChatMemoryRepository（Neo4j 图数据库存储）

详见 [聊天记忆功能](https://springdoc.cn/spring-ai/api/chat-memory.html) 文档获取详细说明和使用示例。

**实现说明**

ChatClient API 的独特之处在于融合了命令式与响应式编程模型。通常应用会择一使用，而非两者兼用。

* 定制 Model 实现的 HTTP 客户端交互时，需同时配置 RestClient 和 WebClient。

|  |  |
| --- | --- |
|  | 由于 Spring Boot 3.4 的缺陷，必须设置 spring.http.client.factory=jdk 属性。默认值 "reactor" 会导致 ImageModel 等 AI 工作流异常。 |

* 流式传输仅通过响应式技术栈支持。因此命令式应用需引入响应式依赖（如 spring-boot-starter-webflux）。
* 非流式传输仅通过 Servlet 技术栈支持。因此响应式应用需引入 Servlet 依赖（如 spring-boot-starter-web），且需知悉部分调用将出现阻塞。
* 工具调用采用命令式设计，会导致工作流阻塞。这将产生不完整的 Micrometer 观测数据（例如 ChatClient 的追踪 span 与工具调用的 span 未关联，且前者因此保持未完成状态）。
* 内置 Advisor 对标准调用执行阻塞操作，对流式调用执行非阻塞操作。用于 Advisor 流式调用的 Reactor Scheduler 可通过各 Advisor 类的 Builder 进行配置。

### **Advisor API**

Spring AI Advisor API 为拦截、修改和增强 Spring 应用中的 AI 交互提供了灵活强大的方式。通过该 API，开发者能构建更复杂、可复用且易维护的 AI 组件。

核心优势包括：封装可复用的生成式 AI 模式、转换与大语言模型（LLM）交互的数据、实现跨模型与用例的可移植性。

如以下示例所示，可通过 [ChatClient](https://springdoc.cn/spring-ai/api/chatclient.html#_advisor_configuration_in_chatclient) API 配置现有 Advisor：

var chatClient = ChatClient.builder(chatModel)

.defaultAdvisors(

MessageChatMemoryAdvisor.builder(chatMemory).build(), *// chat-memory advisor*

QuestionAnswerAdvisor.builder((vectorStore).builder() *// RAG advisor*

)

)

.build();

var conversationId = "678";

String response = this.chatClient.prompt()

*// 运行时设置 advisor 参数*

.advisors(advisor -> advisor.param(ChatMemory.CONVERSATION\_ID, conversationId))

.user(userText)

.call()

.content();

Copied!

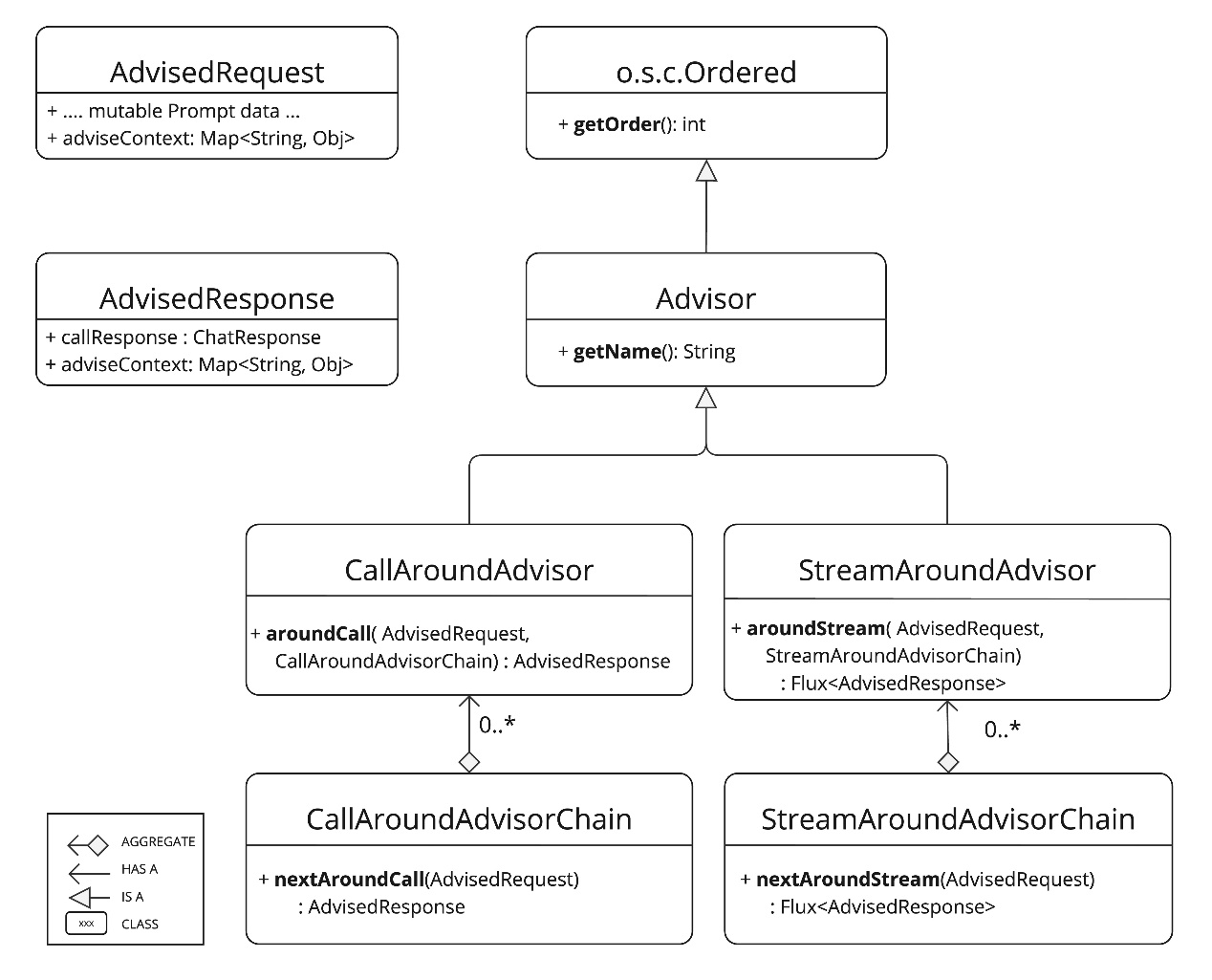
建议在构建时使用构建器（builder）的 defaultAdvisors() 方法注册顾问。

Advisor 组件同时接入可观测性体系，支持查看其执行相关的指标与追踪数据。

* [了解问答 Advisor 功能](https://springdoc.cn/spring-ai/api/retrieval-augmented-generation.html#_questionansweradvisor)
* [了解聊天记忆 Advisor 功能](https://springdoc.cn/spring-ai/api/chat-memory.html#_memory_in_chat_client)

**核心组件**

API 包括用于非流式场景的 CallAroundAdvisor 和 CallAroundAdvisorChain，以及用于流式场景的 StreamAroundAdvisor 和 StreamAroundAdvisorChain。它还包括代表未密封（unsealed）提示请求的 AdvisedRequest 和代表聊天完成响应的 AdvisedResponse。两者都持有一个 advise-context，以便在 Advisor 链中共享状态。

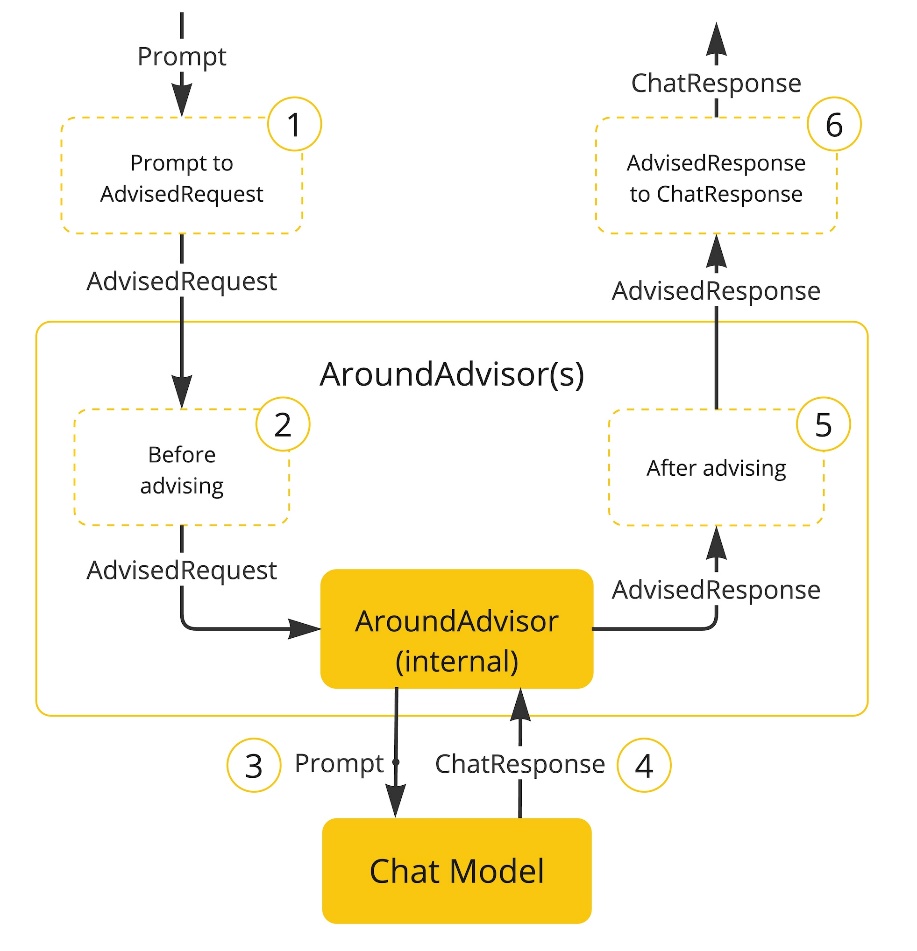


核心 Advisor 方法 nextAroundCall() 与 nextAroundStream() 主要执行以下操作：检查未密封的 Prompt 数据、定制与增强 Prompt 内容、调用 Advisor 链中的下一实体、选择性阻断请求、分析聊天完成响应，并通过抛出异常标识处理错误。

此外，getOrder() 方法决定 Advisor 在链中的执行顺序，而 getName() 则提供唯一的 Advisor 标识名称。

Spring AI 框架构建的 Advisor Chain 支持按 getOrder() 值排序依次调用多个 Advisor — 数值越小优先级越高。链末自动加入的最终顾问会将请求发送至大语言模型（LLM）。

以下流程图展示了 Advisor 链与聊天模型的交互过程：



1. Spring AI 框架将用户的 Prompt 封装为 AdvisedRequest 对象，并创建空的 AdvisorContext 上下文。
2. 链中每个 Advisor 依次处理请求并可进行修改，也可选择阻断请求（不调用下一实体）。若选择阻断，该 Advisor 需负责填充响应内容。
3. 框架提供的最终 Advisor 将请求发送至聊天模型。
4. 聊天模型的响应会逆向传回 Advisor 链，被转换为包含共享 AdvisorContext 实例的 AdvisedResponse 对象。
5. 每个 Advisor 均可处理或修改该响应。
6. 通过提取 ChatCompletion 内容，最终生成的 AdvisedResponse 将返回给客户端。

**Advisor 顺序**

链中 Advisor 的执行顺序由 getOrder() 方法决定。需要了解的要点：

* 数值越小的 Advisor 越优先执行。
* Advisor 链采用栈式结构运作：
  + 链首 Advisor 最先处理请求。
  + 同时也是最后处理响应的环节。
* 控制执行顺序：
  + 将 order 值设为接近 Ordered.HIGHEST\_PRECEDENCE 可确保 Advisor 优先执行（请求处理时最先触发，响应处理时最后触发）。
  + 将 order 值设为接近 Ordered.LOWEST\_PRECEDENCE 可确保 Advisor 最后执行（请求处理时最后触发，响应处理时最先触发）。
* 数值越大表示优先级越低。
* 若多个 Advisor 的 order 值相同，其执行顺序无法保证。

|  |  |
| --- | --- |
|  | 执行顺序与数值看似矛盾的现象源于 Advisor 链的栈式特性：   * 优先级最高（order 值最小）的 Advisor 会被置于栈顶。 * 随着栈展开，它将最先处理请求。 * 而在栈回退时，它将最后处理响应。 |

以下是 Spring Ordered 接口的语义说明：

public interface Ordered {

*/\*\**

*\* 最高优先级的常量值。*

*\* @see java.lang.Integer#MIN\_VALUE*

*\*/*

int HIGHEST\_PRECEDENCE = Integer.MIN\_VALUE;

*/\*\**

*\* 最低优先级的常量值。*

*\* @see java.lang.Integer#MAX\_VALUE*

*\*/*

int LOWEST\_PRECEDENCE = Integer.MAX\_VALUE;

*/\*\**

*\* 获取此对象的顺序值。*

*\* <p>Higher values are interpreted as lower priority. As a consequence,*

*\* the object with the lowest value has the highest priority (somewhat*

*\* analogous to Servlet {@code load-on-startup} values).*

*\* <p>Same order values will result in arbitrary sort positions for the*

*\* affected objects.*

*\* @return the order value*

*\* @see #HIGHEST\_PRECEDENCE*

*\* @see #LOWEST\_PRECEDENCE*

*\*/*

int getOrder();

}

Copied!

|  |  |
| --- | --- |
|  | 对于需要在链中输入输出两端都优先处理的使用场景：   1. 应为每侧使用独立的 Advisor。 2. 通过配置不同的 order 值实现。 3. 利用 advisor context 在它们之间共享状态。 |

**API 概览**

核心 Advisor 接口位于 org.springframework.ai.chat.client.advisor.api 包中。开发自定义 Advisor 时需关注以下关键接口：

public interface Advisor extends Ordered {

String getName();

}

Copied!

同步式与响应式 Advisor 的两个子接口分别为：

public interface CallAroundAdvisor extends Advisor {

*/\*\**

*\* Around advice that wraps the ChatModel#call(Prompt) method.*

*\* @param advisedRequest the advised request*

*\* @param chain the advisor chain*

*\* @return the response*

*\*/*

AdvisedResponse aroundCall(AdvisedRequest advisedRequest, CallAroundAdvisorChain chain);

}

Copied!

以及：

public interface StreamAroundAdvisor extends Advisor {

*/\*\**

*\* Around advice that wraps the invocation of the advised request.*

*\* @param advisedRequest the advised request*

*\* @param chain the chain of advisors to execute*

*\* @return the result of the advised request*

*\*/*

Flux<AdvisedResponse> aroundStream(AdvisedRequest advisedRequest, StreamAroundAdvisorChain chain);

}

Copied!

要在 Advisor 实现中延续 Advisor 链，需使用 CallAroundAdvisorChain 与 StreamAroundAdvisorChain ：

接口是：

public interface CallAroundAdvisorChain {

AdvisedResponse nextAroundCall(AdvisedRequest advisedRequest);

}

Copied!

以及：

public interface StreamAroundAdvisorChain {

Flux<AdvisedResponse> nextAroundStream(AdvisedRequest advisedRequest);

}

Copied!

**实现 Advisor**

创建 Advisor 需实现 CallAroundAdvisor 或 StreamAroundAdvisor（或两者）。核心实现方法是：非流式用 nextAroundCall() ，流式用 nextAroundStream()。

**示例**

通过具体示例演示如何实现观察型与增强型 Advisor。

**日志 Advisor**

我们可以实现一个简单的日志 Advisor，在调用链中下一 Advisor 前后，分别记录 AdvisedRequest 和 AdvisedResponse。该实现仅观察请求与响应而不修改，同时支持非流式与流式场景。

public class SimpleLoggerAdvisor implements CallAroundAdvisor, StreamAroundAdvisor {

private static final Logger logger = LoggerFactory.getLogger(SimpleLoggerAdvisor.class);

@Override

public String getName() {

return this.getClass().getSimpleName();

}

@Override

public int getOrder() {

return 0;

}

@Override

public AdvisedResponse aroundCall(AdvisedRequest advisedRequest, CallAroundAdvisorChain chain) {

logger.debug("BEFORE: {}", advisedRequest);

AdvisedResponse advisedResponse = chain.nextAroundCall(advisedRequest);

logger.debug("AFTER: {}", advisedResponse);

return advisedResponse;

}

@Override

public Flux<AdvisedResponse> aroundStream(AdvisedRequest advisedRequest, StreamAroundAdvisorChain chain) {

logger.debug("BEFORE: {}", advisedRequest);

Flux<AdvisedResponse> advisedResponses = chain.nextAroundStream(advisedRequest);

return new MessageAggregator().aggregateAdvisedResponse(advisedResponses,

advisedResponse -> logger.debug("AFTER: {}", advisedResponse));

}

}

Copied!

|  |  |
| --- | --- |
|  | 为 Advisor 提供一个唯一的名称。 |
|  | 你可以通过设置 order 值来控制执行顺序。较低的值优先执行。 |
|  | MessageAggregator 是工具类，用于将 Flux 响应聚合为单个 AdvisedResponse。适用于需观察完整响应（而非流中单项）的日志记录等处理场景。注意：此为只读操作，不可修改响应内容。 |

**重读（Re2）Advisor**

论文 [重读可提高大型语言模型的推理能力](https://arxiv.org/pdf/2309.06275) 提出的 Re-Reading（Re2）技术，通过以下方式增强输入提示词来优化模型推理表现：

{Input\_Query}

Read the question again: {Input\_Query}

实现应用 Re2 技术的 Advisor 可按如下方式处理用户输入查询：

public class ReReadingAdvisor implements CallAroundAdvisor, StreamAroundAdvisor {

private AdvisedRequest before(AdvisedRequest advisedRequest) {

Map<String, Object> advisedUserParams = new HashMap<>(advisedRequest.userParams());

advisedUserParams.put("re2\_input\_query", advisedRequest.userText());

return AdvisedRequest.from(advisedRequest)

.userText("""

{re2\_input\_query}

Read the question again: {re2\_input\_query}

""")

.userParams(advisedUserParams)

.build();

}

@Override

public AdvisedResponse aroundCall(AdvisedRequest advisedRequest, CallAroundAdvisorChain chain) {

return chain.nextAroundCall(this.before(advisedRequest));

}

@Override

public Flux<AdvisedResponse> aroundStream(AdvisedRequest advisedRequest, StreamAroundAdvisorChain chain) {

return chain.nextAroundStream(this.before(advisedRequest));

}

@Override

public int getOrder() {

return 0;

}

@Override

public String getName() {

return this.getClass().getSimpleName();

}

}

Copied!

|  |  |
| --- | --- |
|  | before 方法通过应用重读技术增强用户输入查询。 |
|  | aroundCall 方法拦截非流式请求并应用重读技术。 |
|  | aroundStream 方法拦截流式请求并应用重读技术。 |
|  | 通过设置 order 值控制执行顺序 — 数值越小优先级越高。 |
|  | 为 Advisor 提供唯一标识名称。 |

**Spring AI 内置 Advisor**

Spring AI 框架提供了多个内置 Advisor，以增强你的 AI 交互。以下是可用 Advisor 的概述：

**Chat Memory Advisor**

这些 Advisor 在聊天记忆库中管理对话历史：

* MessageChatMemoryAdvisor

检索记忆并将其作为信息集合添加到提示中。这种方法可以保持对话历史的结构。注意，并非所有人工智能模型都支持这种方法。

* PromptChatMemoryAdvisor

检索记忆内容并将其整合到提示词的系统文本中。

* VectorStoreChatMemoryAdvisor

从向量存储库检索记忆内容并注入提示词的系统文本。该 Advisor 能高效搜索海量数据集中的相关信息。

**问答（QA） Advisor**

* QuestionAnswerAdvisor

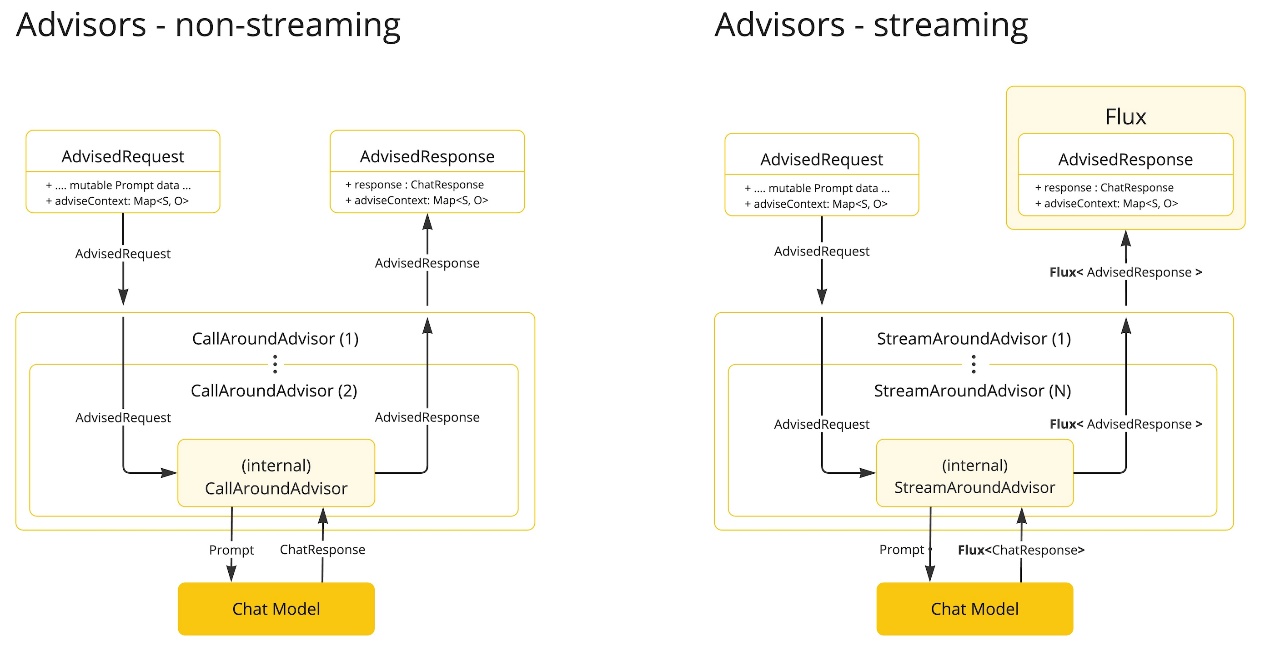
该 Advisor 通过向量存储实现问答功能，采用检索增强生成（RAG）模式。

**内容安全 Advisor**

* SafeGuardAdvisor

基础防护型 Advisor，用于阻止模型生成有害或不恰当内容。

**流式与非流式对比**



* 非流式 Advisor 处理完整请求与响应。
* 流式 Advisor 采用响应式编程理念（如 Flux 处理响应），以连续流形式处理请求与响应。

@Override

public Flux<AdvisedResponse> aroundStream(AdvisedRequest advisedRequest, StreamAroundAdvisorChain chain) {

return Mono.just(advisedRequest)

.publishOn(Schedulers.boundedElastic())

.map(request -> {

*// This can be executed by blocking and non-blocking Threads.*

*// Advisor before next section*

})

.flatMapMany(request -> chain.nextAroundStream(request))

.map(response -> {

*// Advisor after next section*

});

}

Copied!

**最佳实践**

1. 保持 Advisor 功能单一化以提升模块性。
2. 必要时通过 adviseContext 在 Advisor 间共享状态。
3. 同时实现流式与非流式版本以获得最佳灵活性。
4. 谨慎规划 Advisor 链顺序以确保数据流正确。

**向后兼容性**

|  |  |
| --- | --- |
|  | AdvisedRequest 类已迁移至新包。 |

**重大 API 变更**

Spring AI Advisor 链从 **1.0 M2** 到 **1.0 M3** 版本发生重大变更，主要调整如下：

**Advisor 接口**

* 1.0 M2 版本中曾存在独立的 RequestAdvisor 与 ResponseAdvisor 接口。
  + RequestAdvisor 在 ChatModel.call 与 ChatModel.stream 方法前触发。
  + ResponseAdvisor 则在上述方法执行后调用。
* 1.0 M3 版本中这些接口已被替换为：
  + CallAroundAdvisor
  + StreamAroundAdvisor
* 原属 ResponseAdvisor 的 StreamResponseMode 已被移除。

**上下文 Map 处理**

* 在 1.0 M2 中：
  + 上下文 Map 曾作为独立的方法参数存在。
  + 该 Map 为可变对象并在链中传递。
* 在 1.0 M3 中：
  + 上下文 Map 现已整合至 AdvisedRequest 与 AdvisedResponse 记录中。
  + Map 为不可变对象。
  + 需通过 updateContext 方法更新上下文 - 该方法会生成包含更新内容的新不可变 Map。

1.0 M3 中更新上下文的示例：

@Override

public AdvisedResponse aroundCall(AdvisedRequest advisedRequest, CallAroundAdvisorChain chain) {

this.advisedRequest = advisedRequest.updateContext(context -> {

context.put("aroundCallBefore" + getName(), "AROUND\_CALL\_BEFORE " + getName()); *// Add multiple key-value pairs*

context.put("lastBefore", getName()); *// Add a single key-value pair*

return context;

});

*// Method implementation continues...*

}

## **提示（Prompt）**

提示词是引导 AI 模型生成特定输出的输入，其设计和措辞显著影响模型响应。

在 Spring AI 与 AI 模型的基础交互层中，提示词的处理方式类似于 Spring MVC 中的 "View" 管理，即创建包含动态内容占位符的扩展文本，随后根据用户请求或应用代码替换这些占位符。另一种类比是包含表达式占位符的 SQL 语句。

随着 Spring AI 的发展，它将引入更高级别的 AI 模型交互抽象。本节描述的基础类在角色和功能上可类比 JDBC — 例如 ChatModel 类类似于 JDK 中的核心 JDBC 库，而构建于 ChatModel 之上的 ChatClient 类则可比作 JdbcClient，通过 Advisor 实现更高级的构造：考虑与模型的历史交互、用附加上下文文档增强提示词，并引入代理行为。

提示词结构在 AI 领域持续演进：最初是简单字符串，逐渐发展为包含 "USER:" 等模型可识别的特定输入占位符。OpenAI 进一步引入结构化设计，在 AI 模型处理前将多条消息按不同角色分类。

**API 概览**

**Prompt**

通常使用 ChatModel 的 call() 方法，该方法接收 Prompt 实例并返回 ChatResponse。

Prompt 类作为有序 Message 对象和请求 ChatOptions 的容器。每个 Message 在提示中扮演独特角色，其内容和意图各异 — 从用户询问到 AI 生成响应，再到相关背景信息。这种结构支持与 AI 模型的复杂交互，因为提示由多条消息构建而成，每条消息在对话中承担特定角色。

以下是 Prompt 类的简化版本（省略构造函数和工具方法）：

public class Prompt implements ModelRequest<List<Message>> {

private final List<Message> messages;

private ChatOptions chatOptions;

}

Copied!

**Message**

Message 接口封装了 Prompt 文本内容、元数据属性集合以及称为 MessageType 的分类标识。

接口定义如下：

public interface Content {

String getContent();

Map<String, Object> getMetadata();

}

public interface Message extends Content {

MessageType getMessageType();

}

Copied!

多模态消息类型还实现了 MediaContent 接口，提供 Media 内容对象列表。

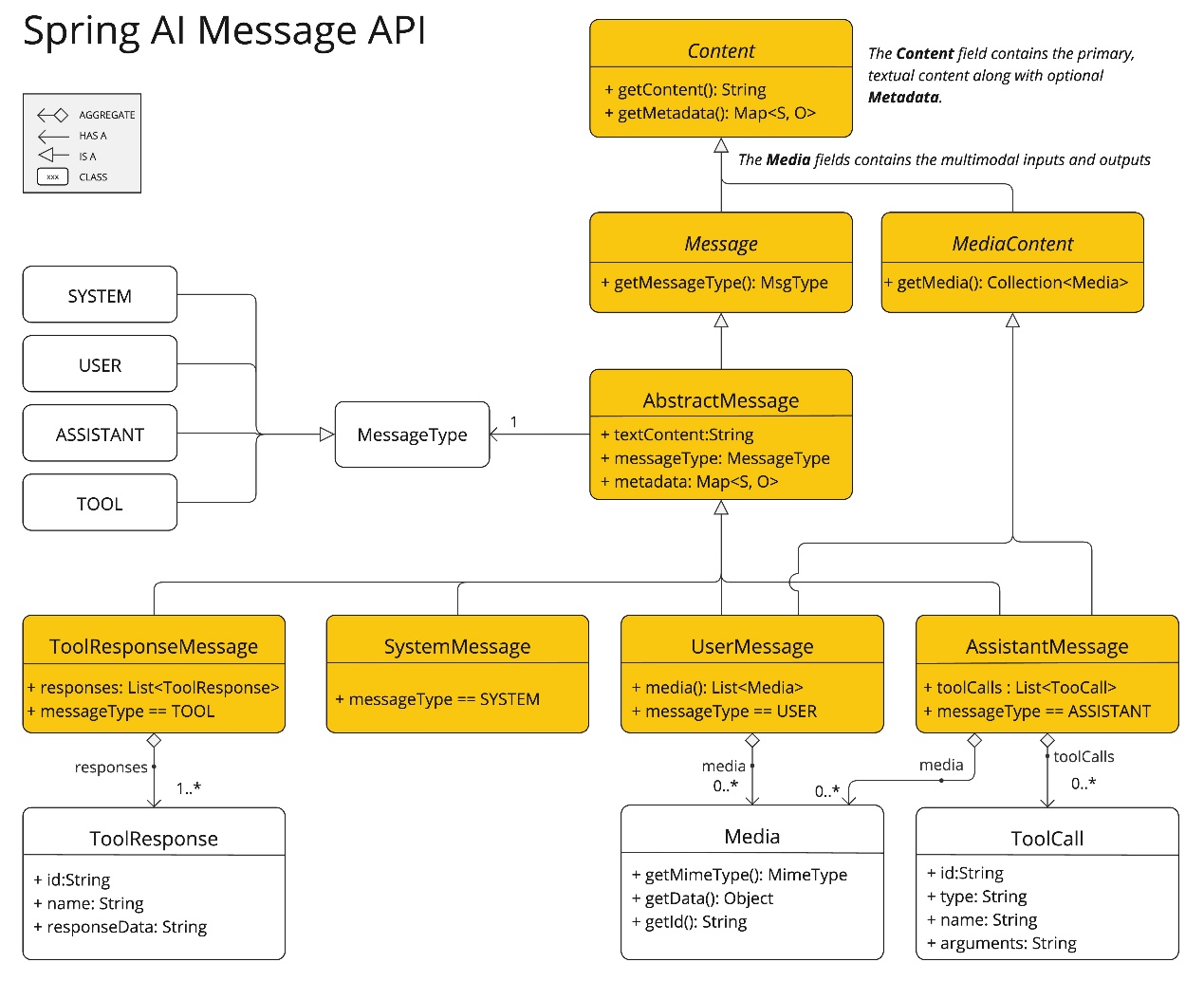
public interface MediaContent extends Content {

Collection<Media> getMedia();

}

Copied!

Message 接口的多种实现对应 AI 模型可处理的不同消息类别，模型根据对话角色区分消息类型。



这些角色由 MessageType 有效映射，如下所述。

**角色（Role）**

每条消息被分配特定角色，这些角色对消息进行分类，向 AI 模型阐明提示每个片段的上下文和目的。这种结构化方法通过让提示的每个部分在交互中扮演明确角色，增强了与 AI 沟通的精细度和有效性。

主要角色包括：

* System 角色：指导 AI 的行为和响应风格，设定 AI 解释和回复输入的参数或规则，类似于在开始对话前向 AI 提供指令。
* User 角色：代表用户的输入 — 包括问题、命令或对 AI 的陈述。该角色构成 AI 响应的基础，具有根本重要性。
* Assistant 角色：AI 对用户输入的响应，不仅是答案或反应，更对维持对话流至关重要。通过追踪 AI 之前的响应（其 "Assistant Role" 消息），系统确保连贯且上下文相关的交互。助手消息也可能包含函数工具调用请求信息 — 这是 AI 的特殊功能，在需要时执行计算、获取数据等超越对话的特定任务。
* Tool/Function 角色：专注于响应工具调用类助手消息，返回附加信息。

角色在 Spring AI 中以枚举形式表示，如下所示：

public enum MessageType {

USER("user"),

ASSISTANT("assistant"),

SYSTEM("system"),

TOOL("tool");

...

}

Copied!

**PromptTemplate**

Spring AI 中提示词模板化的核心组件是 PromptTemplate 类，专为简化结构化提示词的创建而设计，这些提示词随后会发送给 AI 模型处理。

public class PromptTemplate implements PromptTemplateActions, PromptTemplateMessageActions {

*// Other methods to be discussed later*

}

Copied!

该类使用 TemplateRenderer API 渲染模板。默认情况下，Spring AI 采用基于 Terence Parr 开发的开源 [StringTemplate](https://www.stringtemplate.org/) 引擎的 StTemplateRenderer 实现。模板变量通过 {} 语法标识，但也可配置为其他分隔符语法。

public interface TemplateRenderer extends BiFunction<String, Map<String, Object>, String> {

@Override

String apply(String template, Map<String, Object> variables);

}

Copied!

Spring AI 通过 TemplateRenderer 接口处理模板字符串中的变量替换，默认实现使用 [[StringTemplate]](https://springdoc.cn/spring-ai/api/prompt.html#StringTemplate)。若需自定义逻辑，可提供自己的 TemplateRenderer 实现。对于无需模板渲染的场景（如模板字符串已完整），可使用提供的 NoOpTemplateRenderer。

*使用自定义 StringTemplate 渲染器的示例（采用 '<' 和 '>' 分隔符）：*

PromptTemplate promptTemplate = PromptTemplate.builder()

.renderer(StTemplateRenderer.builder().startDelimiterToken('<').endDelimiterToken('>').build())

.template("""

Tell me the names of 5 movies whose soundtrack was composed by <composer>.

""")

.build();

String prompt = promptTemplate.render(Map.of("composer", "John Williams"));

Copied!

该类实现的接口支持提示词创建的不同方面：

PromptTemplateStringActions 专注于创建和渲染提示词字符串，代表最基础的提示生成形式。

PromptTemplateMessageActions 专为通过生成和操作 Message 对象来创建提示词而设计。

PromptTemplateActions 设计用于返回 Prompt 对象，该对象可传递给 ChatModel 生成响应。

虽然这些接口在许多项目中可能不会被广泛使用，但它们展示了提示词创建的不同方法。

实现的接口包括：

public interface PromptTemplateStringActions {

String render();

String render(Map<String, Object> model);

}

Copied!

String render()：将提示词模板渲染为最终字符串格式（无需外部输入），适用于不含占位符或动态内容的模板。

String render(Map<String, Object> model)：增强渲染功能以包含动态内容，使用 Map<String, Object> 参数（Key 为提示模板中的占位符名称， Value 为待插入的动态内容）。

public interface PromptTemplateMessageActions {

Message createMessage();

Message createMessage(List<Media> mediaList);

Message createMessage(Map<String, Object> model);

}

Copied!

* Message createMessage()：创建不带附加数据的 Message 对象，适用于静态或预定义的消息内容。
* Message createMessage(List<Media> mediaList)：创建包含静态文本和媒体内容的 Message 对象。
* Message createMessage(Map<String, Object> model)：扩展消息创建以整合动态内容，接收 Map<String, Object> 参数（每条记录代表消息模板中的占位符及其对应的动态值）。

public interface PromptTemplateActions extends PromptTemplateStringActions {

Prompt create();

Prompt create(ChatOptions modelOptions);

Prompt create(Map<String, Object> model);

Prompt create(Map<String, Object> model, ChatOptions modelOptions);

}

Copied!

* Prompt create()：生成不带外部数据输入的 Prompt 对象，适用于静态或预定义的提示。
* Prompt create(ChatOptions modelOptions)：生成不带外部数据输入但含聊天请求特定选项的 Prompt 对象。
* Prompt create(Map<String, Object> model)：扩展提示创建能力以包含动态内容，接收 Map<String, Object> 参数（每个 Map 条目代表提示模板中的占位符及其关联的动态值）。
* Prompt create(Map<String, Object> model, ChatOptions modelOptions)：扩展提示创建能力以包含动态内容和聊天请求特定选项，接收 Map<String, Object> 参数（每个 Map 条目代表提示模板中的占位符及其关联的动态值）及 ChatOptions 参数。

**示例用法**

以下是 AI Workshop 中关于 [PromptTemplates](https://github.com/Azure-Samples/spring-ai-azure-workshop/blob/main/2-README-prompt-templating.md) 的简单示例：

PromptTemplate promptTemplate = new PromptTemplate("Tell me a {adjective} joke about {topic}");

Prompt prompt = promptTemplate.create(Map.of("adjective", adjective, "topic", topic));

return chatModel.call(prompt).getResult();

Copied!

以下是 AI Workshop 中关于 [Role](https://github.com/Azure-Samples/spring-ai-azure-workshop/blob/main/3-README-prompt-roles.md) 的另一个示例：

String userText = """

Tell me about three famous pirates from the Golden Age of Piracy and why they did.

Write at least a sentence for each pirate.

""";

Message userMessage = new UserMessage(userText);

String systemText = """

You are a helpful AI assistant that helps people find information.

Your name is {name}

You should reply to the user's request with your name and also in the style of a {voice}.

""";

SystemPromptTemplate systemPromptTemplate = new SystemPromptTemplate(systemText);

Message systemMessage = systemPromptTemplate.createMessage(Map.of("name", name, "voice", voice));

Prompt prompt = new Prompt(List.of(userMessage, systemMessage));

List<Generation> response = chatModel.call(prompt).getResults();

Copied!

该示例展示如何通过 SystemPromptTemplate 构建 Prompt 实例：使用 system 角色创建含占位值的 Message，再与 user 角色的 Message 组合成提示，最终传递给 ChatModel 获取生成式响应。

**使用自定义模板渲染器**

你可通过实现 TemplateRenderer 接口并将其传入 PromptTemplate 构造函数来使用自定义模板渲染器，也可继续使用默认的 StTemplateRenderer 但采用自定义配置。

默认情况下，模板变量通过 {} 语法标识。若提示中包含 JSON，建议改用 < 和 > 等分隔符以避免与JSON语法冲突。

PromptTemplate promptTemplate = PromptTemplate.builder()

.renderer(StTemplateRenderer.builder().startDelimiterToken('<').endDelimiterToken('>').build())

.template("""

Tell me the names of 5 movies whose soundtrack was composed by <composer>.

""")

.build();

String prompt = promptTemplate.render(Map.of("composer", "John Williams"));

Copied!

**使用 Resource 替代原始字符串**

Spring AI 支持 org.springframework.core.io.Resource 抽象，因此可将提示数据存入文件并直接用于 PromptTemplate。例如：在 Spring 托管组件中定义字段来获取 Resource。

@Value("classpath:/prompts/system-message.st")

private Resource systemResource;

Copied!

然后将该资源直接传递给 SystemPromptTemplate。

SystemPromptTemplate systemPromptTemplate = new SystemPromptTemplate(systemResource);

Copied!

**提示词工程**

在生成式 AI 中，提示词创建是开发者的关键任务。这些提示词的质量和结构显著影响 AI 输出的有效性。投入时间精心设计提示词可大幅提升 AI 生成结果。

在 AI 社区中，分享和讨论提示词是常见实践。这种协作方式不仅创造了共享学习环境，还能促成高效提示词的识别与应用。

该领域研究常涉及分析和比较不同提示词在各种情境下的有效性。例如，一项重要研究表明，以 “深呼吸后逐步解决这个问题” 开头的提示词能显著提升解题效率，这凸显了精心设计的语言对生成式AI系统性能的影响。

掌握提示词的最有效使用方式（尤其是随着 AI 技术的快速发展）是持续挑战。应认识到提示词工程的重要性，并考虑采用社区和研究中的洞见来改进提示创建策略。

**创建高效提示词**

开发提示词时，需整合以下关键组件以确保清晰度和有效性：

* **指令**：向 AI 提供清晰直接的指示，类似于与人沟通的方式。这种明确性对帮助AI “理解” 预期目标至关重要。
* **外部上下文**：必要时包含相关背景信息或对 AI 响应的具体指导。这种 “外部上下文” 构建了提示框架，帮助 AI 理解整体场景。
* **用户输入**：这是直接部分 — 用户构成提示核心的明确请求或问题。
* **输出指示器**：这部分可能很棘手。它需要指定 AI 响应的期望格式（如 JSON），但需注意 AI 可能不会严格遵循该格式。例如，它可能在实际 JSON 数据前添加 “这是您的JSON” 等短语，或有时生成不准确的类 JSON 结构。

为 AI 提供预期问答格式的示例在构建提示词时极为有益。这种做法帮助AI “理解” 查询的结构与意图，从而产生更精准相关的响应。虽然本文档未深入这些技术，但它们为提示词工程的进一步探索提供了起点。

以下是供进一步研究的资源列表：

**基础技巧**

* [**文本摘要**](https://www.promptingguide.ai/introduction/examples.en#text-summarization)：  
  将冗长文本压缩为简洁摘要，捕捉关键要点和核心思想，同时省略次要细节。
* [**问答**](https://www.promptingguide.ai/introduction/examples.en#question-answering)：  
  专注于从给定文本中根据用户提问提取特定答案，实质是精确定位并提取与查询相关的信息。
* [**文本分类**](https://www.promptingguide.ai/introduction/examples.en#text-classification)：  
  系统地将文本归入预定义类别，通过分析内容将其分配至最匹配的类别。
* [**对话**](https://www.promptingguide.ai/introduction/examples.en#conversation)：  
  创建交互式对话，使AI能与用户进行往复交流，模拟自然对话流。
* [**代码生成**](https://www.promptingguide.ai/introduction/examples.en#code-generation)：  
  根据特定用户需求或描述生成功能性代码片段，将自然语言指令转化为可执行代码。

**高级技巧**

* [**零样本**](https://www.promptingguide.ai/techniques/zeroshot)**、**[**小样本学习**](https://www.promptingguide.ai/techniques/fewshot)：  
  使模型能在极少或没有特定问题类型示例的情况下做出准确预测或响应，利用已学习的泛化能力理解并执行新任务。
* [**思维链**](https://www.promptingguide.ai/techniques/cot)：  
  串联多个AI响应以创建连贯且上下文感知的对话，帮助AI保持讨论主线，确保相关性与连续性。
* [**ReAct（推理+行动）**](https://www.promptingguide.ai/techniques/react)：  
  该方法中，AI 首先分析（推理）输入内容，随后确定最合适的行动方案或响应，将理解过程与决策机制相结合。

**微软 Guidance 框架**

* [**提示创建与优化框架**](https://github.com/microsoft/guidance)：  
  微软提供结构化方法来开发和改进提示词，该框架指导用户创建能有效引导 AI 模型产生预期响应的提示，优化交互的清晰度和效率。

**Token**

Token 是 AI 模型处理文本的关键要素，作为桥梁将人类理解的单词转换为模型可处理的格式。该转换分两阶段进行：输入时单词转为 Token，输出时 Token 再转回单词。

Token 化（将文本分解为 Token 的过程）是 AI 模型理解和处理语言的基础。模型通过这种 Token 化格式来理解并响应提示词。

要理解 Token，可将其视为单词的组成部分。通常一个 Token 相当于单词的约四分之三，例如莎士比亚全集约 90 万单词，会转换为约 120 万Token。

通过 [OpenAI Tokenizer UI](https://platform.openai.com/tokenizer) 实验观察单词如何转换为 Token。

Token 除在 AI 处理中的技术角色外，还对计费和模型能力具有实际影响：

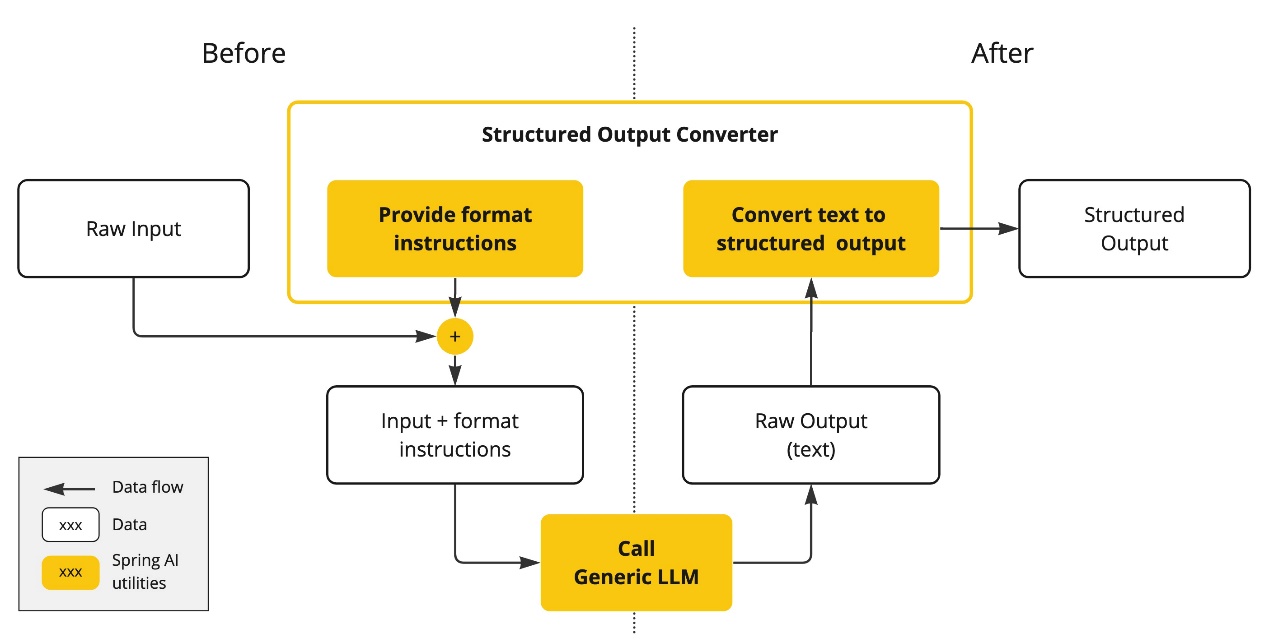
* 计费：AI 模型服务通常按 Token 使用量计费。输入（提示词）和输出（响应）均计入总 Token 数，因此更短的提示词更具成本效益。
* 模型限制：不同 AI 模型拥有不同的 Token 限制，即其 “上下文窗口” — 单次可处理的最大信息量。例 如GPT-3 限制为 4K Token，而 Claude 2 和 Meta Llama 2 等模型可达 100K Token，某些研究模型甚至能处理 100 万 Token。
* 上下文窗口：模型的 Token 限制决定了其上下文窗口大小。超过该限制的输入不会被处理。关键在于仅发送最小必要有效信息集，例如查询《哈姆雷特》时，无需包含莎士比亚其他作品的 Token。
* 响应元数据：AI 模型响应的元数据包含使用的 Token 数，这是管理使用量和成本的关键信息。

## **结构化输出转换器**

|  |
| --- |
| 截至 2024 年 5 月 2 日，旧版 OutputParser、BeanOutputParser、 ListOutputParser 和 MapOutputParser 类已弃用，推荐改用新版 StructuredOutputConverter、BeanOutputConverter、ListOutputConverter 和 MapOutputConverter 实现。后者可直接替换前者并提供相同功能，此次变更主要出于命名规范化考虑（实际并无解析操作），同时也为对齐 Spring 的 org.springframework.core.convert.converter 包并引入增强功能。 |

大语言模型生成结构化输出的能力对依赖可靠解析结果的下游应用至关重要。开发者希望快速将 AI 模型结果转换为 JSON、XML 或 Java Class 等数据类型，以便传递给其他应用函数和方法。

Spring AI 结构化输出转换器（**Structured Output Converter**）帮助将 LLM 输出转为结构化格式。如下图所示，该方案围绕 LLM 文本补全端点运作：



使用通用补全 API 从大语言模型（LLM）生成结构化输出需谨慎处理输入输出。结构化输出转换器在 LLM 调用前后起关键作用，确保获得预期输出结构。

在 LLM 调用前，转换器向提示词追加格式指令，为模型生成预期输出结构提供明确指导。这些指令作为蓝图，引导模型响应符合指定格式。

LLM 调用后，转换器（**Converter**）将模型的原始文本输出转换为结构化类型实例。该转换过程包括解析原始文本输出，并将其映射 为JSON、XML 或领域特定数据结构等对应的结构化数据表示。

|  |  |
| --- | --- |
|  | StructuredOutputConverter 会尽力将模型输出转换为结构化格式，但 AI 模型并不保证按请求返回结构化输出（可能无法理解提示或生成所需结构）。建议实现验证机制以确保模型输出符合预期。 |
|  | StructuredOutputConverter 不用于 LLM [工具调用](https://springdoc.cn/spring-ai/api/tools.html) 功能，因此特性默认已提供结构化输出。 |

**结构化输出 API**

StructuredOutputConverter 接口允许从基于文本的 AI 模型输出中获取结构化结果，例如映射到 Java Class 或值数组。其接口定义为：

public interface StructuredOutputConverter<T> extends Converter<String, T>, FormatProvider {

}

Copied!

该接口结合了 Spring 的 [Converter<String, T>](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/convert/converter/Converter.html) 和 FormatProvider 接口。

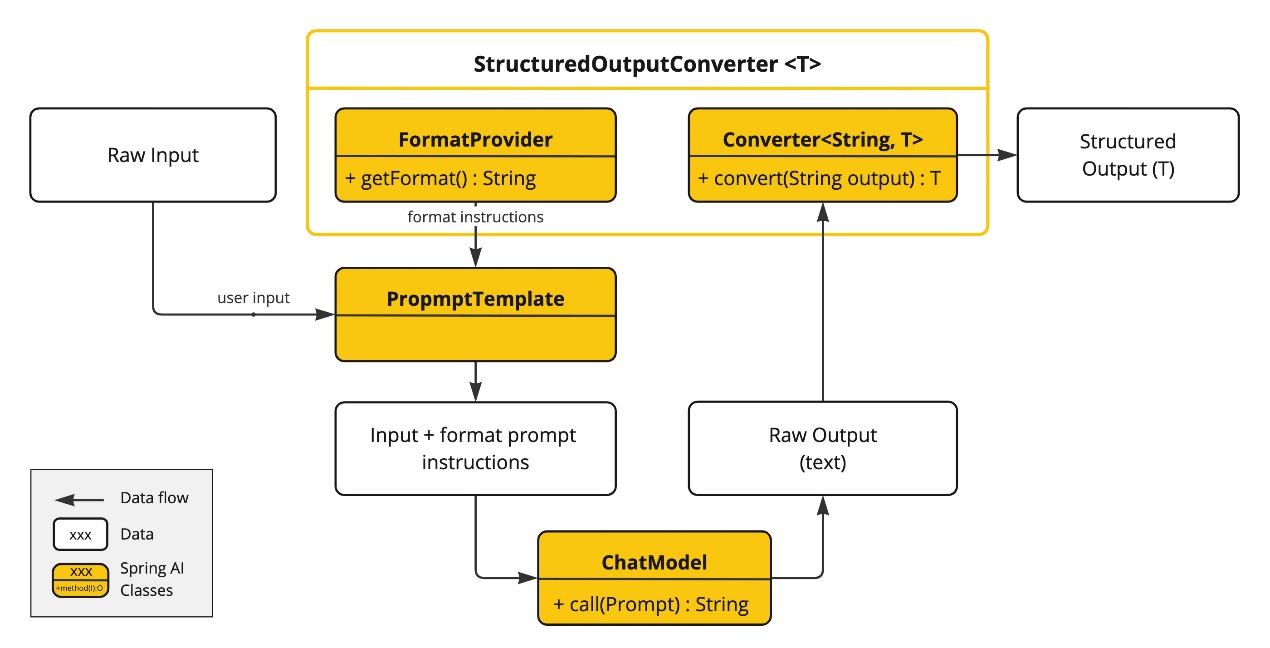
public interface FormatProvider {

String getFormat();

}

Copied!

下图展示了使用结构化输出 API 时的数据流：



FormatProvider 向 AI 模型提供特定格式指南，使其生成可被 Converter 转换为目标类型 T 的文本输出。以下是此类格式指令的示例：

Your response should be in JSON format.

The data structure for the JSON should match this Java class: java.util.HashMap

Do not include any explanations, only provide a RFC8259 compliant JSON response following this format without deviation.

格式指令通常通过 [PromptTemplate](https://springdoc.cn/spring-ai/api/prompt.html#_prompttemplate) 追加到用户输入末尾，如下所示：

StructuredOutputConverter outputConverter = ...

String userInputTemplate = """

... user text input ....

{format}

"""; *// 包含 "format" 占位符的用户输入。*

Prompt prompt = new Prompt(

new PromptTemplate(

this.userInputTemplate,

Map.of(..., "format", outputConverter.getFormat()) *// 将 "format" 占位符替换为转换器的格式指令。*

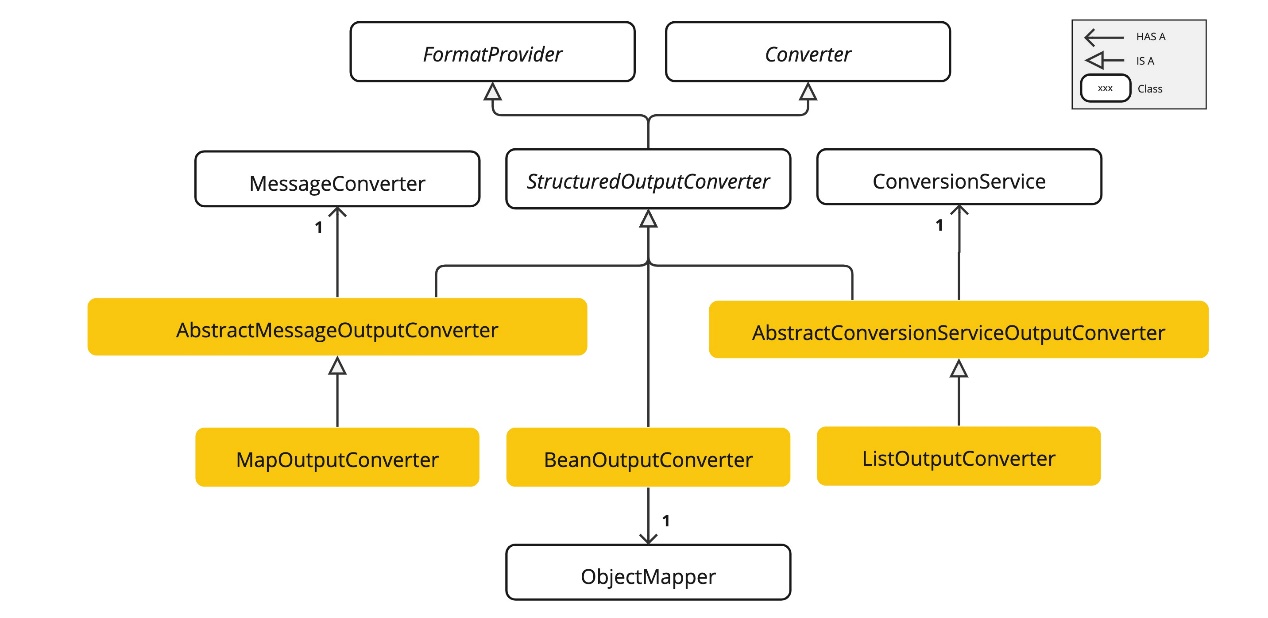
).createMessage());

Copied!

Converter<String, T> 负责将模型的文本输出转换为指定类型 T 的实例。

**可用的转换器**

目前 Spring AI 提供以下实现：AbstractConversionServiceOutputConverter、AbstractMessageOutputConverter、BeanOutputConverter、MapOutputConverter和ListOutputConverter。



* AbstractConversionServiceOutputConverter<T> - 提供预配置的 [GenericConversionService](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/convert/support/GenericConversionService.html) 用于将 LLM 输出转换为目标格式，未提供默认 FormatProvider 实现。
* AbstractMessageOutputConverter<T> - 提供预配置的 [MessageConverter](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jms/support/converter/MessageConverter.html) 用于将 LLM 输出转换为目标格式，未提供默认 FormatProvider 实现。
* BeanOutputConverter<T> - BeanOutputConverter<T>：通过配置 Java 类或 [ParameterizedTypeReference](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/ParameterizedTypeReference.html)，该转换器（Converter）使用 FormatProvider 实现指导 AI 模型生成符合 DRAFT\_2020\_12、JSON Schema 的响应（基于指定 Java 类生成），随后用 ObjectMapper 将 JSON 输出反序列化为目标类的 Java 对象实例。
* MapOutputConverter - 继承 AbstractMessageOutputConverter 的功能，通过 FormatProvider 实现引导 AI 模型生成符合 RFC8259 标准的 JSON 响应，并利用提供的 MessageConverter 将 JSON Payload 转换为 java.util.Map<String, Object> 实例。
* ListOutputConverter - 继承 AbstractConversionServiceOutputConverter，包含专为逗号分隔列表输出定制的 FormatProvider 实现。该转换器利用提供的 ConversionService 将模型文本输出转换为 java.util.List。

**使用转换器**

以下章节提供使用现有转换器生成结构化输出的指南。

**BeanOutputConverter**

以下示例展示如何使用 BeanOutputConverter 生成演员作品表：

表示演员作品表的目标 record 类型：

record ActorsFilms(String actor, List<String> movies) {

}

Copied!

以下是使用高阶 Fluent 式 ChatClient API 应用 BeanOutputConverter 的方式：

ActorsFilms actorsFilms = ChatClient.create(chatModel).prompt()

.user(u -> u.text("Generate the filmography of 5 movies for {actor}.")

.param("actor", "Tom Hanks"))

.call()

.entity(ActorsFilms.class);

Copied!

或直接使用底层 ChatModel API：

BeanOutputConverter<ActorsFilms> beanOutputConverter =

new BeanOutputConverter<>(ActorsFilms.class);

String format = this.beanOutputConverter.getFormat();

String actor = "Tom Hanks";

String template = """

Generate the filmography of 5 movies for {actor}.

{format}

""";

Generation generation = chatModel.call(

new PromptTemplate(this.template, Map.of("actor", this.actor, "format", this.format)).create()).getResult();

ActorsFilms actorsFilms = this.beanOutputConverter.convert(this.generation.getOutput().getText());

Copied!

**生成模式中的属性顺序**

BeanOutputConverter 通过 @JsonPropertyOrder 注解支持自定义 JSON Schema 中的属性顺序，该注解允许指定属性在模式中的精确出现顺序（无视 class 或 record 中的声明顺序）。

例如，确保 ActorsFilms record 中的属性特定顺序：

@JsonPropertyOrder({"actor", "movies"})

record ActorsFilms(String actor, List<String> movies) {}

Copied!

该注解同时适用于 record 和常规 Java class。

**泛型 Bean 类型**

使用 ParameterizedTypeReference 构造函数指定更复杂的目标类结构。例如，表示演员及其作品表的列表：

List<ActorsFilms> actorsFilms = ChatClient.create(chatModel).prompt()

.user("Generate the filmography of 5 movies for Tom Hanks and Bill Murray.")

.call()

.entity(new ParameterizedTypeReference<List<ActorsFilms>>() {});

Copied!

或直接使用底层 ChatModel API：

BeanOutputConverter<List<ActorsFilms>> outputConverter = new BeanOutputConverter<>(

new ParameterizedTypeReference<List<ActorsFilms>>() { });

String format = this.outputConverter.getFormat();

String template = """

Generate the filmography of 5 movies for Tom Hanks and Bill Murray.

{format}

""";

Prompt prompt = new PromptTemplate(this.template, Map.of("format", this.format)).create();

Generation generation = chatModel.call(this.prompt).getResult();

List<ActorsFilms> actorsFilms = this.outputConverter.convert(this.generation.getOutput().getText());

Copied!

**MapOutputConverter**

以下片段展示如何使用 MapOutputConverter 将模型输出转换为 Map 中的数值列表：

Map<String, Object> result = ChatClient.create(chatModel).prompt()

.user(u -> u.text("Provide me a List of {subject}")

.param("subject", "an array of numbers from 1 to 9 under they key name 'numbers'"))

.call()

.entity(new ParameterizedTypeReference<Map<String, Object>>() {});

Copied!

或直接使用底层 ChatModel API：

MapOutputConverter mapOutputConverter = new MapOutputConverter();

String format = this.mapOutputConverter.getFormat();

String template = """

Provide me a List of {subject}

{format}

""";

Prompt prompt = new PromptTemplate(this.template,

Map.of("subject", "an array of numbers from 1 to 9 under they key name 'numbers'", "format", this.format)).create();

Generation generation = chatModel.call(this.prompt).getResult();

Map<String, Object> result = this.mapOutputConverter.convert(this.generation.getOutput().getText());

Copied!

**ListOutputConverter**

以下片段展示如何使用 ListOutputConverter 将模型输出转换为冰淇淋口味列表：

List<String> flavors = ChatClient.create(chatModel).prompt()

.user(u -> u.text("List five {subject}")

.param("subject", "ice cream flavors"))

.call()

.entity(new ListOutputConverter(new DefaultConversionService()));

Copied!

或直接使用底层 ChatModel API：

ListOutputConverter listOutputConverter = new ListOutputConverter(new DefaultConversionService());

String format = this.listOutputConverter.getFormat();

String template = """

List five {subject}

{format}

""";

Prompt prompt = new PromptTemplate(this.template,

Map.of("subject", "ice cream flavors", "format", this.format)).create();

Generation generation = this.chatModel.call(this.prompt).getResult();

List<String> list = this.listOutputConverter.convert(this.generation.getOutput().getText());

Copied!

**支持的AI模型**

以下 AI 模型已测试支持 List、Map 和 Bean 结构化输出：

|  |  |
| --- | --- |
| 模型 | 集成测试 / 示例 |
| [OpenAI](https://springdoc.cn/spring-ai/api/chat/openai-chat.html) | [OpenAiChatModelIT](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/test/java/org/springframework/ai/openai/chat/OpenAiChatModelIT.java) |
| [Anthropic Claude 3](https://springdoc.cn/spring-ai/api/chat/anthropic-chat.html) | [AnthropicChatModelIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-anthropic/src/test/java/org/springframework/ai/anthropic/AnthropicChatModelIT.java) |
| [Azure OpenAI](https://springdoc.cn/spring-ai/api/chat/azure-openai-chat.html) | [AzureOpenAiChatModelIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-azure-openai/src/test/java/org/springframework/ai/azure/openai/AzureOpenAiChatModelIT.java) |
| [Mistral AI](https://springdoc.cn/spring-ai/api/chat/mistralai-chat.html) | [MistralAiChatModelIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-mistral-ai/src/test/java/org/springframework/ai/mistralai/MistralAiChatModelIT.java) |
| [Ollama](https://springdoc.cn/spring-ai/api/chat/ollama-chat.html) | [OllamaChatModelIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-ollama/src/test/java/org/springframework/ai/ollama/OllamaChatModelIT.java) |
| [Vertex AI Gemini](https://springdoc.cn/spring-ai/api/chat/vertexai-gemini-chat.html) | [VertexAiGeminiChatModelIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-vertex-ai-gemini/src/test/java/org/springframework/ai/vertexai/gemini/VertexAiGeminiChatModelIT.java) |

**内置 JSON Schema**

部分 AI 模型提供专用配置选项以生成结构化（通常为 JSON）输出。

* [OpenAI 结构化输出](https://springdoc.cn/spring-ai/api/chat/openai-chat.html#_structured_outputs) 确保模型生成严格符合所提供 JSON Schema 的响应。可选模式包括：保证生成有效 JSON 的 JSON\_OBJECT，或通过 spring.ai.openai.chat.options.responseFormat 配置、确保响应匹配指定 schema 的 JSON\_SCHEMA。
* [Azure OpenAI](https://springdoc.cn/spring-ai/api/chat/azure-openai-chat.html) 提 供 spring.ai.azure.openai.chat.options.responseFormat 选项指定模型输出格式。设置为 { "type": "json\_object" } 可启用 JSON 模式，确保模型生成的消息为有效 JSON。
* [Ollama](https://springdoc.cn/spring-ai/api/chat/ollama-chat.html) 提供 spring.ai.ollama.chat.options.format 选项指定响应格式，当前仅接受 JSON 值。
* [Mistral AI](https://springdoc.cn/spring-ai/api/chat/mistralai-chat.html) 提供 spring.ai.mistralai.chat.options.responseFormat 选项指定响应格式，设置为 { "type": "json\_object" } 可启用 JSON 模式，确保模型生成的消息为有效 JSON。

## **多模态 API**

*“所有自然联系的事物都应结合起来传授” — 约翰-阿莫斯-夸美纽斯，《世界图绘》，1658 年*

人类通过多模态数据输入并行处理知识。我们的学习方式和体验都是多模态的 — 不只有视觉、听觉或文本的单一感知。

然而，机器学习往往专注于处理单一模态的专用模型。例如，我们开发音频模型用于文本转语音或语音转文本任务，开发计算机视觉模型用于目标检测和分类等任务。

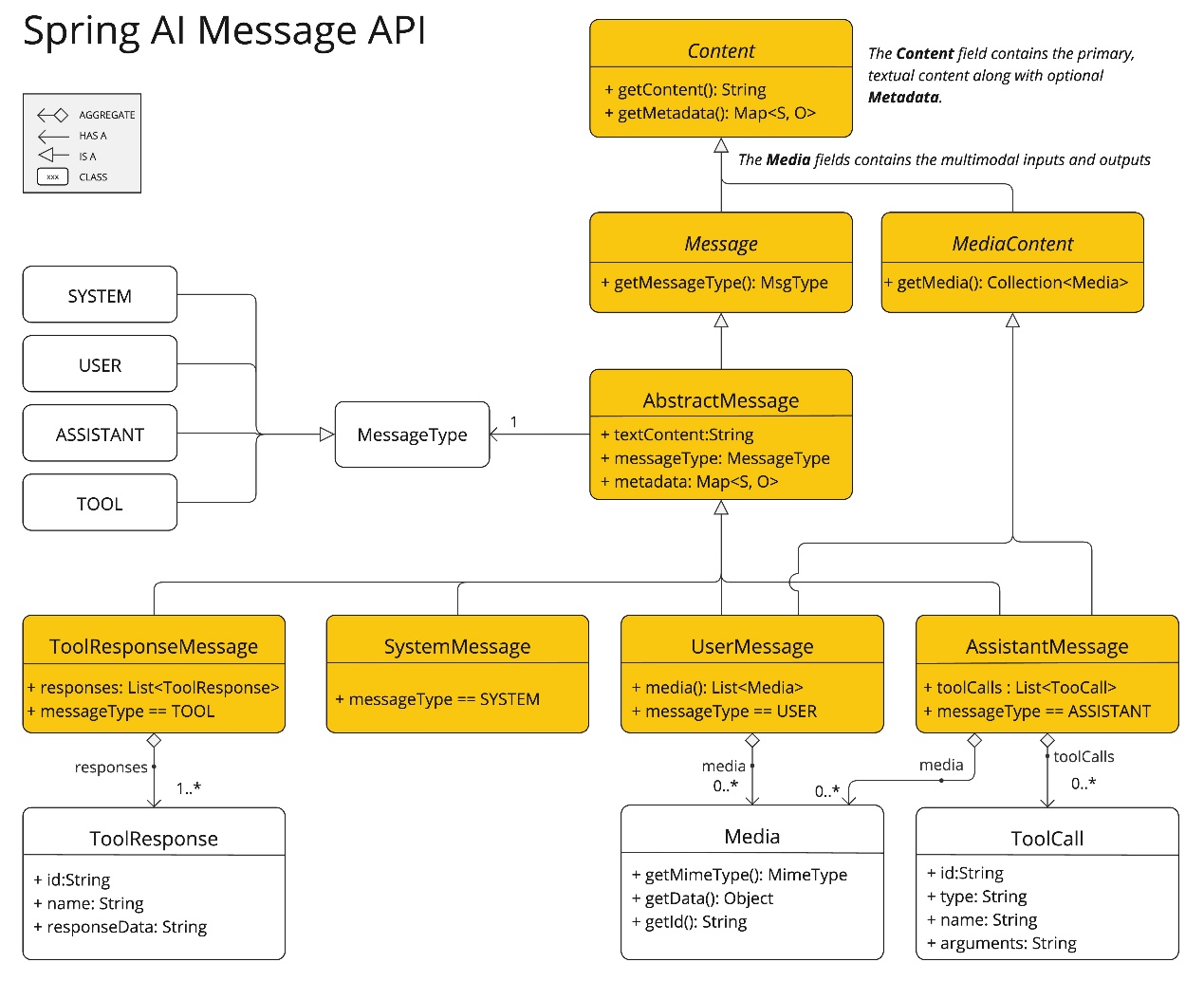
然而，新一代多模态大语言模型正在兴起。例如 OpenAI 的GPT-4o、Google 的Vertex AI Gemini 1.5、Anthropic 的 Claude3，以及开源模型 Llama3.2、LLaVA 和 BakLLaVA，都能接受文本、图像、音频和视频等多种输入，并通过整合这些输入生成文本响应。

|  |  |
| --- | --- |
|  | 多模态大语言模型（LLM）的特性使其能够结合图像、音频或视频等其他模态处理并生成文本。 |

**Spring AI 多模态**

多模态性指模型同时理解和处理文本、图像、音频及其他数据格式等多源信息的能力。

Spring AI Message API 提供了支持多模态 LLM 所需的所有抽象。



UserMessage 的 content 字段主要用于文本输入，而可选的 media 字段允许添加图像、音频和视频等多模态内容。MimeType 指定模态类型，根据所用 LLM 的不同，Media 数据字段可以是原始媒体内容（作为 Resource 对象）或内容 URI。

|  |  |
| --- | --- |
|  | 目前 media 字段仅适用于用户输入消息（如 UserMessage），对系统消息无意义。包含 LLM 响应的 AssistantMessage 仅提供文本内容。要生成非文本媒体输出，应使用专用的单模态模型。 |

例如，我们可以将下图（multimodal.test.png）作为输入，要求 LLM 解释它所识别的内容。



对于大多数多模态 LLM，Spring AI 代码通常如下所示：

var imageResource = new ClassPathResource("/multimodal.test.png");

var userMessage = new UserMessage(

"Explain what do you see in this picture?", *// 内容*

new Media(MimeTypeUtils.IMAGE\_PNG, this.imageResource)); *// 媒体*

ChatResponse response = chatModel.call(new Prompt(this.userMessage));

Copied!

或使用 Fluent 风格的 [ChatClient](https://springdoc.cn/spring-ai/api/chatclient.html) API：

String response = ChatClient.create(chatModel).prompt()

.user(u -> u.text("Explain what do you see on this picture?")

.media(MimeTypeUtils.IMAGE\_PNG, new ClassPathResource("/multimodal.test.png")))

.call()

.content();

Copied!

并生成类似响应：

*这是一幅设计简洁的水果碗图像。碗体由金属制成，带有弯曲的金属丝边缘，形成开放式结构，可从各个角度看到水果。碗内两根黄色香蕉置于看似红苹果的果实上方，香蕉皮上的棕色斑点表明其略微过熟。碗口配有金属环，可能用作提手。碗置于中性色调背景的平面上，清晰呈现碗内水果。*

Spring AI 为以下聊天模型提供多模态支持：

* [Anthropic Claude 3](https://springdoc.cn/spring-ai/api/chat/anthropic-chat.html#_multimodal)
* [AWS Bedrock Converse](https://springdoc.cn/spring-ai/api/chat/bedrock-converse.html#_multimodal)
* [Azure Open AI (e.g. GPT-4o models)](https://springdoc.cn/spring-ai/api/chat/azure-openai-chat.html#_multimodal)
* [Mistral AI (e.g. Mistral Pixtral models)](https://springdoc.cn/spring-ai/api/chat/mistralai-chat.html#_multimodal)
* [Ollama (e.g. LLaVA, BakLLaVA, Llama3.2 models)](https://springdoc.cn/spring-ai/api/chat/ollama-chat.html#_multimodal)
* [OpenAI (e.g. GPT-4 and GPT-4o models)](https://springdoc.cn/spring-ai/api/chat/openai-chat.html#_multimodal)
* [Vertex AI Gemini (e.g. gemini-1.5-pro-001, gemini-1.5-flash-001 models)](https://springdoc.cn/spring-ai/api/chat/vertexai-gemini-chat.html#_multimodal)

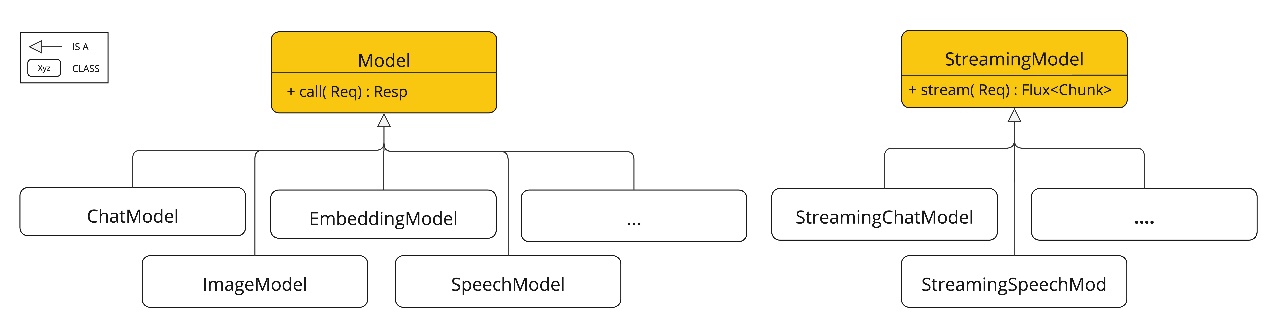
## **Spring AI API**

**简介**

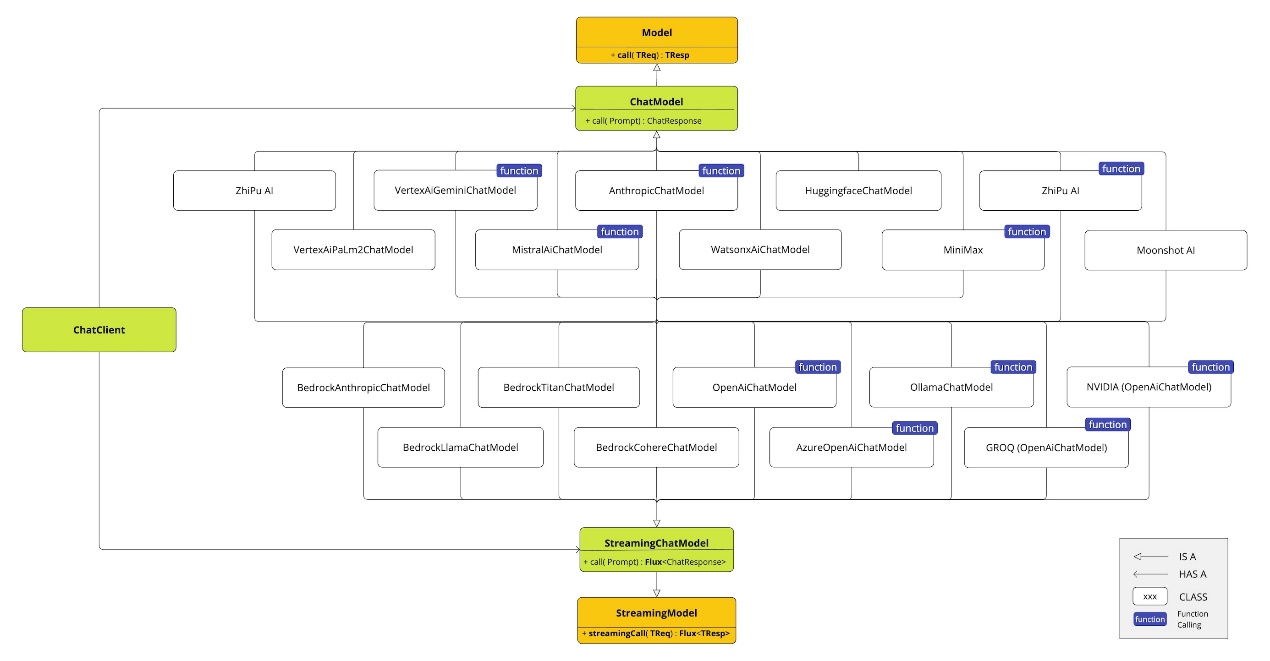
Spring API 涵盖广泛功能，各核心特性均有独立章节详解。主要功能概览如下：

**AI 模型 API**

跨 AI 供应商的可移植模 API，支持聊天、文生图、音频转录、文生语音及嵌入模型。同时提供同步和流式 API 选项，并支持访问模型特定功能。



支持 OpenAI、微软、亚马逊、谷歌、Amazon Bedrock、Hugging Face 等厂商的 AI 模型。

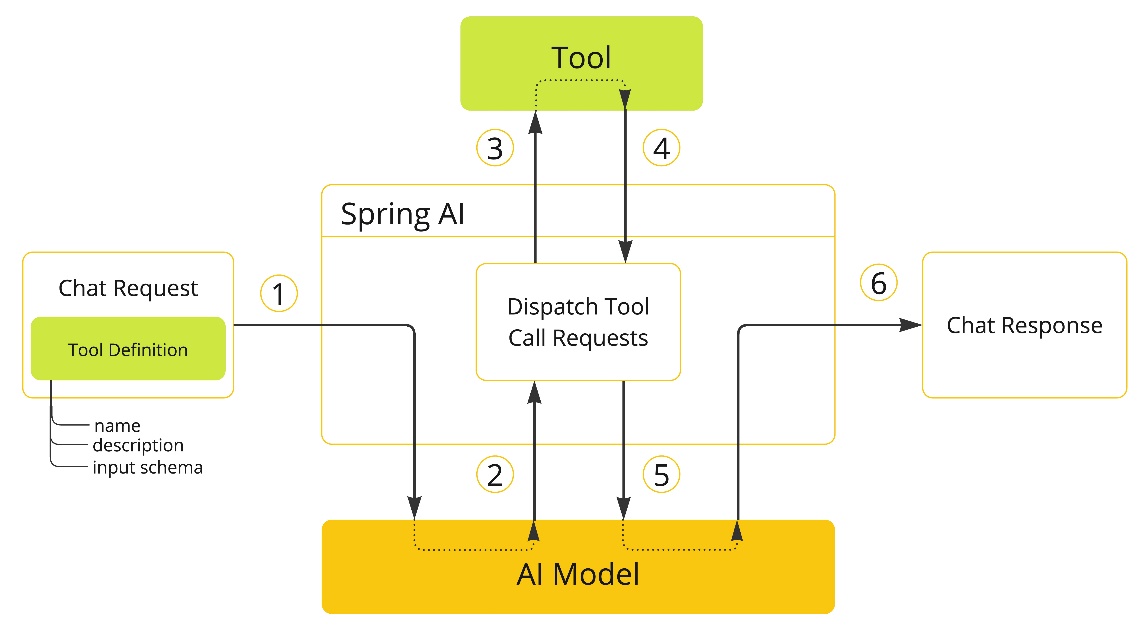


**向量存储 API**

跨多供应商的可移植向量存储 API，包含创新的类 SQL 元数据过滤接口（同样具备可移植性），目前已支持 14 种向量数据库。

**工具调用 API**

Spring AI 简化了 AI 模型调用服务的流程，支持通过 @Tool 注解方法或 POJO 式 java.util.Function 对象实现。



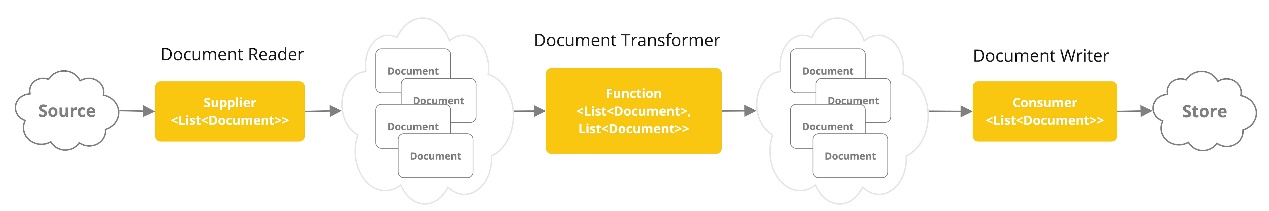
详见 [Spring AI 工具调用文档](https://springdoc.cn/spring-ai/api/tools.html)。

**自动配置**

提供 AI 模型与向量存储的 Spring Boot 自动配置及 Starter 支持。

**ETL 数据工程**

ETL 数据工程框架：为向量数据库提供数据加载基础架构，助力实现检索增强生成模式，使你能将自有数据整合至 AI 模型响应中。



## **OpenAI**

Spring AI 支持来自 ChatGPT 母公司 OpenAI 的多种 AI 语言模型。OpenAI 通过创建行业领先的文本生成模型和嵌入技术，在推动 AI 驱动文本生成领域的发展中发挥了关键作用。

**先决条件**

你需要创建 OpenAI API Key 以访问 ChatGPT 模型。

在 [OpenAI 注册页面](https://platform.openai.com/signup) 创建账户，并于 [API Key 页面](https://platform.openai.com/account/api-keys) 生成访问令牌。

Spring AI 项目定义了名为 spring.ai.openai.api-key 的配置属性，你应将其设置为从 openai.com 获取的 API Key。

你可以在 application.properties 文件中设置此配置属性：

spring.ai.openai.api-key=<your-openai-api-key>

Copied!

为提升处理 API Key 等敏感信息的安全性，可使用 Spring 表达式语言（SpEL）引用自定义环境变量：

# In application.yml

spring:

ai:

openai:

api-key: ${OPENAI\_API\_KEY}

Copied!

# In your environment or .env file

export OPENAI\_API\_KEY=<your-openai-api-key>

Copied!

你还可以在应用程序代码中以编程方式设置此配置：

*// Retrieve API key from a secure source or environment variable*

String apiKey = System.getenv("OPENAI\_API\_KEY");

Copied!

**添加仓库和 BOM**

Spring AI 构件已发布至 Maven Central 和 Spring Snapshot 仓库。请参阅 [构件仓库](https://springdoc.cn/spring-ai/getting-started.html#artifact-repositories) 章节将这些仓库添加到构建系统中。

为统一依赖版本管理，Spring AI 提供 BOM（物料清单）确保项目中使用一致的 Spring AI 版本。请参考 [依赖管理](https://springdoc.cn/spring-ai/getting-started.html#dependency-management) 章节将 Spring AI BOM 添加到构建系统中。

**自动配置**

|  |  |
| --- | --- |
|  | Spring AI 自动配置及 Starter 模块的 artifact 命名已发生重大变更。具体升级说明请参阅 [更新文档](https://springdoc.cn/spring-ai/upgrade-notes.html)。 |

Spring AI 为 OpenAI 聊天客户端提供 Spring Boot 自动配置。要启用该功能，请将以下依赖项添加到项目的 Maven pom.xml 或 Gradle build.gradle 构建文件中：

* **Maven**
* **Gradle**

<dependency>

<groupId>org.springframework.ai</groupId>

<artifactId>spring-ai-starter-model-openai</artifactId>

</dependency>

Copied!

|  |  |
| --- | --- |
|  | 请参考 [依赖管理](https://springdoc.cn/spring-ai/getting-started.html#dependency-management) 章节，将 Spring AI BOM 添加到构建文件中。 |

**聊天配置**

**重试配置**

前缀 spring.ai.retry 用作配置属性前缀，用于配置 OpenAI 聊天模型的重试机制。

| **属性** | **说明** | **默认值** |
| --- | --- | --- |
| spring.ai.retry.max-attempts | 最大重试次数。 | 10 |
| spring.ai.retry.backoff.initial-interval | 指数退避策略的初始休眠时长。 | 2 sec. |
| spring.ai.retry.backoff.multiplier | 退避间隔乘数。 | 5 |
| spring.ai.retry.backoff.max-interval | 最大退避时长。 | 3 min. |
| spring.ai.retry.on-client-errors | 若为 false，则抛出 NonTransientAiException 异常，且不对 4xx 客户端错误代码进行重试 | false |
| spring.ai.retry.exclude-on-http-codes | 不应触发重试的 HTTP 状态码列表（例如用于抛出 NonTransientAiException 异常）。 | empty |
| spring.ai.retry.on-http-codes | 应触发重试的 HTTP 状态码列表（例如用于抛出 TransientAiException 异常）。 | empty |

**连接配置**

前缀 spring.ai.openai 用作属性前缀，用于配置 OpenAI 连接参数。

| **属性** | **说明** | **默认值** |
| --- | --- | --- |
| spring.ai.openai.base-url | 要连接到的 URL | [api.openai.com](https://api.openai.com/) |
| spring.ai.openai.api-key | API Key | - |
| spring.ai.openai.organization-id | 可选参数，用于指定 API 请求使用的组织。 | - |
| spring.ai.openai.project-id | 可选参数，用于指定 API 请求使用的项目。 | - |

|  |  |
| --- | --- |
|  | 对于属于多个组织的用户（或通过旧版用户 API Key访问项目），可选择性指定 API 请求使用的组织及项目。此类 API 请求的使用量将计入指定组织和项目。 |

**配置属性**

|  |  |
| --- | --- |
|  | 聊天自动配置的启用/禁用现在通过顶级属性配置，前缀为 spring.ai.model.chat：   * 启用方式：spring.ai.model.chat=openai（默认已启用） * 禁用方式：spring.ai.model.chat=none（或任何非 openai 的值）   此项变更是为了支持多模型配置。 |

前缀 spring.ai.openai.chat 用于配置 OpenAI 聊天模型实现的属性前缀。

| **属性** | **说明** | **默认值** |
| --- | --- | --- |
| spring.ai.openai.chat.enabled （已移除且不再生效） | 启用 OpenAI 聊天模型 | true |
| spring.ai.model.chat | 启用 OpenAI 聊天模型 | openai |
| spring.ai.openai.chat.base-url | 可选参数，用于覆盖 spring.ai.openai.base-url 属性以提供聊天专用 URL。 | - |
| spring.ai.openai.chat.completions-path | 附加到 Base URL 的路径。 | /v1/chat/completions |
| spring.ai.openai.chat.api-key | 可选参数，用于覆盖 spring.ai.openai.api-key 以提供聊天专用 API Key。 | - |
| spring.ai.openai.chat.organization-id | 可选参数，用于指定 API 请求使用的组织。 | - |
| spring.ai.openai.chat.project-id | 可选参数，用于指定 API 请求使用的项目。 | - |
| spring.ai.openai.chat.options.model | 使用的 OpenAI 聊天模型名称。可选模型包括：gpt-4o、gpt-4o-mini、gpt-4-turbo、gpt-3.5-turbo 等。详情请参阅 [模型](https://platform.openai.com/docs/models) 页面。 | gpt-4o-mini |
| spring.ai.openai.chat.options.temperature | 采样温度值参数，用于控制生成内容的表观创造性。较高值会使输出更随机，较低值则使结果更集中和确定。不建议在同一补全请求中同时修改 temperature 和 top\_p 参数，因这两个设置的交互影响难以预测。 | 0.8 |
| spring.ai.openai.chat.options.frequencyPenalty | 介于 -2.0 至 2.0 之间的数值。正值会根据新 Token 在当前文本中的现有频率进行惩罚，从而降低模型逐字重复相同内容的可能性。 | 0.0f |
| spring.ai.openai.chat.options.logitBias | 调整指定 Token 在补全结果中出现的概率。 | - |
| spring.ai.openai.chat.options.maxTokens | （已弃用，建议改用 maxCompletionTokens）聊天补全中生成的最大 Token 数。输入 Token 和生成 Token 的总长度受模型上下文长度限制。 | - |
| spring.ai.openai.chat.options.maxCompletionTokens | 生成补全结果时可产生的 Token 数量上限（包括可见输出 Token 和推理 Token）。 | - |
| spring.ai.openai.chat.options.n | 为每条输入消息生成的聊天补全选项数量。请注意，系统将根据所有选项生成的 Token 总数计费。保持 n=1 可最大限度降低成本。 | 1 |
| spring.ai.openai.chat.options.store | 是否存储此聊天补全请求的输出以用于模型改进。 | false |
| spring.ai.openai.chat.options.metadata | 开发者定义的标签和值，用于在聊天补全仪表板中筛选补全结果。 | empty map |
| spring.ai.openai.chat.options.output-modalities | 模型为此请求生成的输出类型。大多数模型默认生成文本。 gpt-4o-audio-preview 模型还可生成音频。如需同时获取文本和音频响应，可使用：text, audio。流式传输不支持此功能。 | - |
| spring.ai.openai.chat.options.output-audio | 音频生成的参数配置。当通过 output-modalities: audio 请求音频输出时必须设置。 需使用 gpt-4o-audio-preview 模型，且不支持流式补全。 | - |
| spring.ai.openai.chat.options.presencePenalty | 介于 -2.0 和 2.0 之间的数值。正值会根据新 Token 是否已出现在当前文本中进行惩罚，从而增加模型讨论新话题的可能性。 | - |
| spring.ai.openai.chat.options.responseFormat.type | 兼容 GPT-4o、GPT-4o mini、GPT-4 Turb`o 及所有新于 `gpt-3.5-turbo-1106 的 GPT-3.5 Turbo 模型。 JSON\_OBJECT 类型启用 JSON 模式，确保模型生成的消息是有效 JSON JSON\_SCHEMA 类型启用 [结构化输出](https://platform.openai.com/docs/guides/structured-outputs)，确保模型匹配你提供的 JSON Schema（需同时设置 responseFormat.schema 属性） | - |
| spring.ai.openai.chat.options.responseFormat.name | 响应格式模式名称。仅当 responseFormat.type=JSON\_SCHEMA 时适用。 | custom\_schema |
| spring.ai.openai.chat.options.responseFormat.schema | 响应格式 JSON模 式。仅适用于 responseFormat.type=JSON\_SCHEMA。 | - |
| spring.ai.openai.chat.options.responseFormat.strict | 响应格式 JSON Schema 严格遵循级别。仅适用于 responseFormat.type=JSON\_SCHEMA。 | - |
| spring.ai.openai.chat.options.seed | 该功能处于测试阶段。若指定此参数，系统将尽力进行确定性采样，使得使用相同种子和参数的重复请求应返回相同结果。 | - |
| spring.ai.openai.chat.options.stop | 最多可设置 4 个终止序列，当 API 生成到这些序列时将停止产生后续 Token。 | - |
| spring.ai.openai.chat.options.topP | 温度采样的替代方案——核采样（nucleus sampling），模型仅考虑概率质量累加达 top\_p 值的 Token 结果。例如 0.1 表示仅考虑概率质量排名前 10% 的 Token。通常建议仅修改此参数或 temperature 参数，而非同时调整两者。 | - |
| spring.ai.openai.chat.options.tools | 模型可调用的工具列表。当前仅支持将函数作为工具使用。通过此参数提供函数列表，模型可能为其生成 JSON 格式的输入。 | - |
| spring.ai.openai.chat.options.toolChoice | 控制模型是否调用函数及调用方式： none：模型不调用函数，仅生成消息（无函数时的默认值） auto：模型自主选择生成消息或调用函数（存在函数时的默认值） 通过 {"type": "function", "function": {"name": "my\_function"}} 指定特定函数时，强制模型调用该函数 | - |
| spring.ai.openai.chat.options.user | 表示终端用户的唯一标识符，可帮助 OpenAI 监控和检测滥用行为。 | - |
| spring.ai.openai.chat.options.functions | 函数名称列表，用于在单次提示请求中启用函数调用功能。这些名称对应的函数必须存在于 functionCallbacks 注册表中。 | - |
| spring.ai.openai.chat.options.stream-usage | （仅限流式传输）设置后将为整个请求添加包含 Token 使用统计信息的附加数据块。该数据块的 choices 字段为空数组，其他所有数据块也将包含 usage 字段，但其值为 null。 | false |
| spring.ai.openai.chat.options.parallel-tool-calls | 是否在工具使用时启用 [并行函数调用](https://platform.openai.com/docs/guides/function-calling/parallel-function-calling)。 | true |
| spring.ai.openai.chat.options.http-headers | 可选参数，为聊天补全请求添加 HTTP Header。要覆盖 api-key 需使用 Authorization Header Key，并在值前添加 Bearer 前缀。 | - |
| spring.ai.openai.chat.options.proxy-tool-calls | 若为 true，Spring AI 将不会内部处理函数调用，而是将其代理给客户端。此时客户端需负责处理函数调用、调度至对应函数并返回结果。若为 false（默认值）， Spring AI 将在内部处理函数调用。仅适用于支持函数调用的聊天模型。 | false |

|  |  |
| --- | --- |
|  | 你可覆盖 ChatModel 和 EmbeddingModel 实现的通用配置 spring.ai.openai.base-url 与 spring.ai.openai.api-key。若设置了 spring.ai.openai.chat.base-url 和 spring.ai.openai.chat.api-key 属性，其优先级高于通用属性。此功能适用于需为不同模型和不同终端节点使用不同 OpenAI 账户的场景。 |
|  | 所有以 spring.ai.openai.chat.options 为前缀的属性，均可通过在 Prompt 调用中添加请求特定的 [运行时属性](https://springdoc.cn/spring-ai/api/chat/openai-chat.html#chat-options) 进行运行时覆盖。 |

**运行时属性**

[OpenAiChatOptions.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/main/java/org/springframework/ai/openai/OpenAiChatOptions.java) 类提供模型配置参数，包括使用的模型、温度值、频率惩罚系数等。

启动时，默认选项可通过 OpenAiChatModel(api, options) 构造函数或 spring.ai.openai.chat.options.\* 属性进行配置。

运行时，你可通过向 Prompt 调用添加新的请求特定选项来覆盖默认配置。例如，针对特定请求覆盖默认模型和温度值：

ChatResponse response = chatModel.call(

new Prompt(

"Generate the names of 5 famous pirates.",

OpenAiChatOptions.builder()

.model("gpt-4o")

.temperature(0.4)

.build()

));

Copied!

|  |  |
| --- | --- |
|  | 除模型特定的 [OpenAiChatOptions](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/main/java/org/springframework/ai/openai/OpenAiChatOptions.java) 外，你还可使用通过 [ChatOptionsBuilder#builder()](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-client-chat/src/main/java/org/springframework/ai/chat/prompt/ChatOptionsBuilder.java) 创建的可移植 [ChatOptions](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-client-chat/src/main/java/org/springframework/ai/chat/prompt/ChatOptions.java) 实例。 |

**函数调用**

你可为 OpenAiChatModel 注册自定义 Java 函数，使 OpenAI 模型能智能输出包含参数的 JSON 对象来调用一个或多个注册函数。这是将 LLM 能力与外部工具及 API 连接的强大技术。了解更多关于 [工具调用](https://springdoc.cn/spring-ai/api/tools.html) 的信息。

**多模态**

多模态指模型同时理解和处理多种来源信息的能力，包括文本、图像、音频及其他数据格式。OpenAI 支持文本、视觉和音频输入模态。

**视觉**

支持视觉多模态的 OpenAI 模型包括 gpt-4、gpt-4o 和 gpt-4o-mini。详情请参阅 [视觉指南](https://platform.openai.com/docs/guides/vision)。

OpenAI [用户消息 API](https://platform.openai.com/docs/api-reference/chat/create#chat-create-messages) 支持将 base64 编码图像或图像 URL 列表与消息结合。Spring AI 的 [Message](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-model/src/main/java/org/springframework/ai/chat/messages/Message.java) 接口通过引入 [Media](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-model/src/main/java/org/springframework/ai/model/Media.java) 类型支持多模态 AI 模型，该类型利用 Spring 的 org.springframework.util.MimeType 和 org.springframework.core.io.Resource 处理消息中的媒体附件数据及元信息。

以下代码示例节选自 [OpenAiChatModelIT.java](https://github.com/spring-projects/spring-ai/blob/c9a3e66f90187ce7eae7eb78c462ec622685de6c/models/spring-ai-openai/src/test/java/org/springframework/ai/openai/chat/OpenAiChatModelIT.java#L293)，展示如何通过 gpt-4o 模型将用户文本与图像结合处理：

var imageResource = new ClassPathResource("/multimodal.test.png");

var userMessage = new UserMessage("Explain what do you see on this picture?",

new Media(MimeTypeUtils.IMAGE\_PNG, this.imageResource));

ChatResponse response = chatModel.call(new Prompt(this.userMessage,

OpenAiChatOptions.builder().model(OpenAiApi.ChatModel.GPT\_4\_O.getValue()).build()));

Copied!

|  |  |
| --- | --- |
|  | 自 2024 年 6 月 17 日起，GPT\_4\_VISION\_PREVIEW 模型将仅对现有用户继续可用。若非现有用户，请使用 GPT\_4\_O 或 GPT\_4\_TURBO 模型。详情参见 [此处](https://platform.openai.com/docs/deprecations/2024-06-06-gpt-4-32k-and-vision-preview-models)。 |

或使用 gpt-4o 模型的等效图像 URL：

var userMessage = new UserMessage("Explain what do you see on this picture?",

new Media(MimeTypeUtils.IMAGE\_PNG,

URI.create("https://docs.spring.io/spring-ai/reference/\_images/multimodal.test.png")));

ChatResponse response = chatModel.call(new Prompt(this.userMessage,

OpenAiChatOptions.builder().model(OpenAiApi.ChatModel.GPT\_4\_O.getValue()).build()));

Copied!

|  |  |
| --- | --- |
|  | 你还可以传递多张图像进行处理。 |

该示例展示模型接收 multimodal.test.png 图像作为输入：



并结合文本消息 “Explain what do you see on this picture?”，生成的响应示例如下：

This is an image of a fruit bowl with a simple design. The bowl is made of metal with curved wire edges that

create an open structure, allowing the fruit to be visible from all angles. Inside the bowl, there are two

yellow bananas resting on top of what appears to be a red apple. The bananas are slightly overripe, as

indicated by the brown spots on their peels. The bowl has a metal ring at the top, likely to serve as a handle

for carrying. The bowl is placed on a flat surface with a neutral-colored background that provides a clear

view of the fruit inside.

**音频**

支持音频输入多模态的 OpenAI 模型包括 gpt-4o-audio-preview。详情请参阅 [音频](https://platform.openai.com/docs/guides/audio) 指南。

OpenAI [用户消息 API](https://platform.openai.com/docs/api-reference/chat/create#chat-create-messages) 支持将 base64 编码音频文件与消息结合。Spring AI 的 [Message](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-model/src/main/java/org/springframework/ai/chat/messages/Message.java) 接口通过引入 [Media](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-client-chat/src/main/java/org/springframework/ai/chat/messages/Media.java) 类型支持多模态 AI 模型，该类型利用 Spring 的 org.springframework.util.MimeType 和 org.springframework.core.io.Resource 处理消息中的音频附件数据及元信息。当前 OpenAI 仅支持以下媒体类型：audio/mp3 和 audio/wav。

以下代码示例节选自 [OpenAiChatModelIT.java](https://github.com/spring-projects/spring-ai/blob/c9a3e66f90187ce7eae7eb78c462ec622685de6c/models/spring-ai-openai/src/test/java/org/springframework/ai/openai/chat/OpenAiChatModelIT.java#L442)，展示如何通过 gpt-4o-audio-preview 模型将用户文本与音频文件结合处理：

var audioResource = new ClassPathResource("speech1.mp3");

var userMessage = new UserMessage("What is this recording about?",

List.of(new Media(MimeTypeUtils.parseMimeType("audio/mp3"), audioResource)));

ChatResponse response = chatModel.call(new Prompt(List.of(userMessage),

OpenAiChatOptions.builder().model(OpenAiApi.ChatModel.GPT\_4\_O\_AUDIO\_PREVIEW).build()));

Copied!

|  |  |
| --- | --- |
|  | 你还可以传递多个音频文件进行处理。 |

**输出音频**

支持音频输入多模态的 OpenAI 模型包括 gpt-4o-audio-preview。详情请参阅 [音频](https://platform.openai.com/docs/guides/audio) 指南。

OpenAI [Assystant Message API](https://platform.openai.com/docs/api-reference/chat/create#chat-create-messages) 可以在消息中包含一组 Base64 编码的音频文件。Spring AI 的 [Message](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-model/src/main/java/org/springframework/ai/chat/messages/Message.java) 接口通过引入 [Media](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-model/src/main/java/org/springframework/ai/chat/messages/Media.java) 类型来支持多模态 AI 模型。该类型包含有关消息中媒体附件的数据和详细信息，使用了 Spring 的 org.springframework.util.MimeType 和 org.springframework.core.io.Resource 来表示原始媒体数据。目前，OpenAI 仅支持以下音频类型：audio/mp3 和 audio/wav。

以下是一个代码示例，演示了使用 gpt-4o-audio-preview 模型响应用户文本和音频字节数组的情况：

var userMessage = new UserMessage("Tell me joke about Spring Framework");

ChatResponse response = chatModel.call(new Prompt(List.of(userMessage),

OpenAiChatOptions.builder()

.model(OpenAiApi.ChatModel.GPT\_4\_O\_AUDIO\_PREVIEW)

.outputModalities(List.of("text", "audio"))

.outputAudio(new AudioParameters(Voice.ALLOY, AudioResponseFormat.WAV))

.build()));

String text = response.getResult().getOutput().getContent(); *// audio transcript*

byte[] waveAudio = response.getResult().getOutput().getMedia().get(0).getDataAsByteArray(); *// audio data*

Copied!

你必须在 OpenAiChatOptions 中指定 audio 模态以生成音频输出。AudioParameters 类为音频输出提供语音和音频格式。

**结构化输出**

OpenAI 提供定制化 [结构化输出](https://platform.openai.com/docs/guides/structured-outputs) API，确保模型生成严格符合你提供的 JSON Schema 的响应。除现有的 Spring AI 模型无关 [结构化输出转换器](https://springdoc.cn/spring-ai/api/structured-output-converter.html) 外，这些 API 提供更精细的控制与精确度。

|  |  |
| --- | --- |
|  | 当前，OpenAI 支持 [JSON Schema 语言格式的子集](https://platform.openai.com/docs/guides/structured-outputs/supported-schemas)。 |

**配置**

Spring AI 支持通过 OpenAiChatOptions Builder 以编程方式配置响应格式，或通过 application properties 进行配置。

**使用聊天选项构建器**

你可以通过 OpenAiChatOptions Builder 以编程方式设置响应格式，如下所示：

String jsonSchema = """

{

"type": "object",

"properties": {

"steps": {

"type": "array",

"items": {

"type": "object",

"properties": {

"explanation": { "type": "string" },

"output": { "type": "string" }

},

"required": ["explanation", "output"],

"additionalProperties": false

}

},

"final\_answer": { "type": "string" }

},

"required": ["steps", "final\_answer"],

"additionalProperties": false

}

""";

Prompt prompt = new Prompt("how can I solve 8x + 7 = -23",

OpenAiChatOptions.builder()

.model(ChatModel.GPT\_4\_O\_MINI)

.responseFormat(new ResponseFormat(ResponseFormat.Type.JSON\_SCHEMA, this.jsonSchema))

.build());

ChatResponse response = this.openAiChatModel.call(this.prompt);

Copied!

|  |  |
| --- | --- |
|  | 遵循 OpenAI 的 [JSON Schema 语言格式子集规范](https://platform.openai.com/docs/guides/structured-outputs/supported-schemas)。 |

**集成 BeanOutputConverter 工具类**

你可利用现有 [BeanOutputConverter](https://springdoc.cn/spring-ai/api/structured-output-converter.html#_bean_output_converter) 工具类自动从 Domain 对象生成 JSON Schema，并将结构化响应转换为 Domain 实例：

* **Java**
* **Kotlin**

record MathReasoning(

@JsonProperty(required = true, value = "steps") Steps steps,

@JsonProperty(required = true, value = "final\_answer") String finalAnswer) {

record Steps(

@JsonProperty(required = true, value = "items") Items[] items) {

record Items(

@JsonProperty(required = true, value = "explanation") String explanation,

@JsonProperty(required = true, value = "output") String output) {

}

}

}

var outputConverter = new BeanOutputConverter<>(MathReasoning.class);

var jsonSchema = this.outputConverter.getJsonSchema();

Prompt prompt = new Prompt("how can I solve 8x + 7 = -23",

OpenAiChatOptions.builder()

.model(ChatModel.GPT\_4\_O\_MINI)

.responseFormat(new ResponseFormat(ResponseFormat.Type.JSON\_SCHEMA, this.jsonSchema))

.build());

ChatResponse response = this.openAiChatModel.call(this.prompt);

String content = this.response.getResult().getOutput().getContent();

MathReasoning mathReasoning = this.outputConverter.convert(this.content);

Copied!

|  |  |
| --- | --- |
|  | 虽然对 JSON Schema 是可选的，但 OpenAI [要求](https://platform.openai.com/docs/guides/structured-outputs/all-fields-must-be-required#all-fields-must-be-required) 必填字段以确保结构化响应的正确性。Kotlin 反射会根据类型可空性和参数默认值推断属性是否必填，因此在大多数情况下不需要使用 @get:JsonProperty(required = true)。@get:JsonProperty(value = "custom\_name") 可用于自定义属性名称。请确保使用 @get: 语法在相关 getter 上生成注解，详见 [相关文档](https://kotlinlang.org/docs/annotations.html#annotation-use-site-targets)。 |

**通过 Application Properties 进行配置**

或者，使用 OpenAI 自动配置时，可通过以下应用属性配置响应格式：

spring.ai.openai.api-key=YOUR\_API\_KEY

spring.ai.openai.chat.options.model=gpt-4o-mini

spring.ai.openai.chat.options.response-format.type=JSON\_SCHEMA

spring.ai.openai.chat.options.response-format.name=MySchemaName

spring.ai.openai.chat.options.response-format.schema={"type":"object","properties":{"steps":{"type":"array","items":{"type":"object","properties":{"explanation":{"type":"string"},"output":{"type":"string"}},"required":["explanation","output"],"additionalProperties":false}},"final\_answer":{"type":"string"}},"required":["steps","final\_answer"],"additionalProperties":false}

spring.ai.openai.chat.options.response-format.strict=true

Copied!

**Sample Controller**

[Create](https://start.spring.io/) a new Spring Boot project and add the spring-ai-starter-model-openai to your pom (or gradle) dependencies.

Add an application.properties file under the src/main/resources directory to enable and configure the OpenAi chat model:

spring.ai.openai.api-key=YOUR\_API\_KEY

spring.ai.openai.chat.options.model=gpt-4o

spring.ai.openai.chat.options.temperature=0.7

Copied!

|  |  |
| --- | --- |
|  | 将 api-key 替换为你的 OpenAI 凭证。 |

这将创建一个可注入到类中的 OpenAiChatModel 实现。以下是一个使用该聊天模型进行文本生成的简单 @RestController 类示例：

@RestController

public class ChatController {

private final OpenAiChatModel chatModel;

@Autowired

public ChatController(OpenAiChatModel chatModel) {

this.chatModel = chatModel;

}

@GetMapping("/ai/generate")

public Map<String,String> generate(@RequestParam(value = "message", defaultValue = "Tell me a joke") String message) {

return Map.of("generation", this.chatModel.call(message));

}

@GetMapping("/ai/generateStream")

public Flux<ChatResponse> generateStream(@RequestParam(value = "message", defaultValue = "Tell me a joke") String message) {

Prompt prompt = new Prompt(new UserMessage(message));

return this.chatModel.stream(prompt);

}

}

Copied!

**手动配置**

[OpenAiChatModel](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/main/java/org/springframework/ai/openai/OpenAiChatModel.java) 实现了 ChatModel 和 StreamingChatModel 接口，并通过 [底层 OpenAiApi 客户端](https://springdoc.cn/spring-ai/api/chat/openai-chat.html#low-level-api) 连接 OpenAI 服务。

将 spring-ai-openai 依赖添加到项目的 Maven pom.xml 文件中：

<dependency>

<groupId>org.springframework.ai</groupId>

<artifactId>spring-ai-openai</artifactId>

</dependency>

Copied!

或添加到 Gradle build.gradle 构建文件中。

dependencies {

implementation 'org.springframework.ai:spring-ai-openai'

}

Copied!

|  |  |
| --- | --- |
|  | 请参考 [依赖管理](https://springdoc.cn/spring-ai/getting-started.html#dependency-management) 章节，将 Spring AI BOM 添加到构建文件中。 |

接下来，创建 OpenAiChatModel 实例并用于文本生成：

var openAiApi = OpenAiApi.builder()

.apiKey(System.getenv("OPENAI\_API\_KEY"))

.build();

var openAiChatOptions = OpenAiChatOptions.builder()

.model("gpt-3.5-turbo")

.temperature(0.4)

.maxTokens(200)

.build();

var chatModel = new OpenAiChatModel(this.openAiApi, this.openAiChatOptions);

ChatResponse response = this.chatModel.call(

new Prompt("Generate the names of 5 famous pirates."));

*// Or with streaming responses*

Flux<ChatResponse> response = this.chatModel.stream(

new Prompt("Generate the names of 5 famous pirates."));

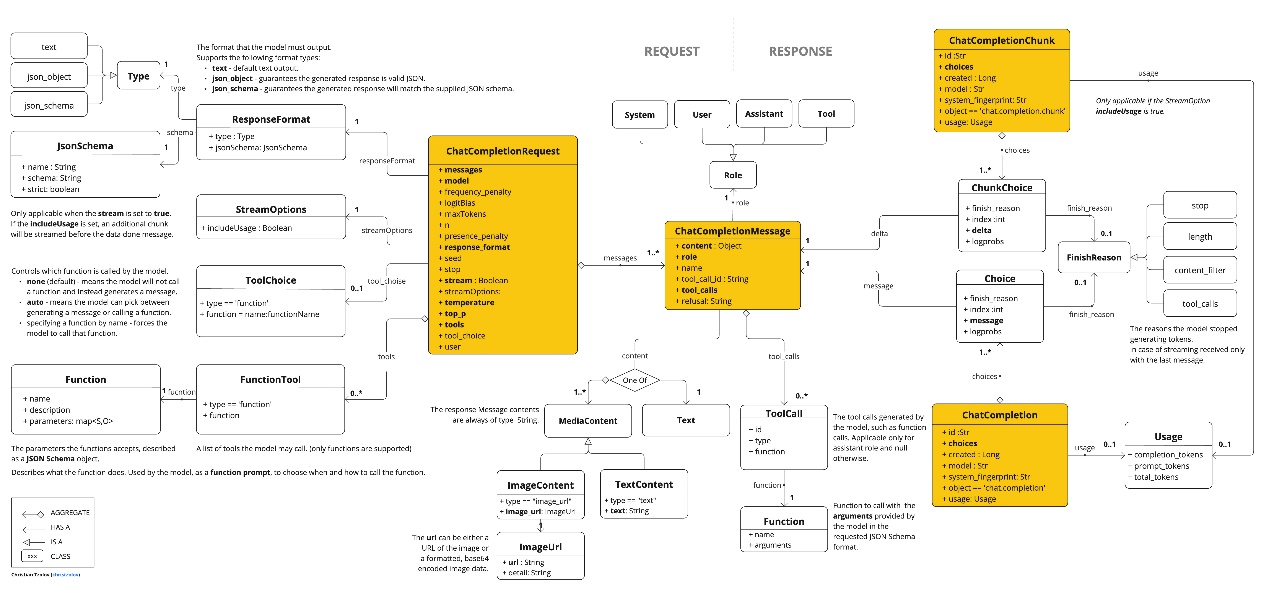
Copied!

OpenAiChatOptions 为聊天请求提供配置信息。OpenAiApi.Builder 和 OpenAiChatOptions.Builder 分别是 API 客户端和聊天配置的 Fluent 式选项构建器。

**底层 OpenAiApi 客户端**

[OpenAiApi](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/main/java/org/springframework/ai/openai/api/OpenAiApi.java) 是 [OpenAI Chat API](https://platform.openai.com/docs/api-reference/chat) 的轻量级 Java 客户端。

以下类图展示 OpenAiApi 聊天接口及核心组件：



以下是编程方式使用该 API 的简单代码片段：

OpenAiApi openAiApi = OpenAiApi.builder()

.apiKey(System.getenv("OPENAI\_API\_KEY"))

.build();

ChatCompletionMessage chatCompletionMessage =

new ChatCompletionMessage("Hello world", Role.USER);

*// Sync request*

ResponseEntity<ChatCompletion> response = this.openAiApi.chatCompletionEntity(

new ChatCompletionRequest(List.of(this.chatCompletionMessage), "gpt-3.5-turbo", 0.8, false));

*// Streaming request*

Flux<ChatCompletionChunk> streamResponse = this.openAiApi.chatCompletionStream(

new ChatCompletionRequest(List.of(this.chatCompletionMessage), "gpt-3.5-turbo", 0.8, true));

Copied!

更多信息请参考 [OpenAiApi.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/main/java/org/springframework/ai/openai/api/OpenAiApi.java) 的 JavaDoc 文档。

**底层 API 示例**

* [OpenAiApiIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/test/java/org/springframework/ai/openai/api/OpenAiApiIT.java) 测试提供了使用该轻量级库的通用示例。
* [OpenAiApiToolFunctionCallIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-openai/src/test/java/org/springframework/ai/openai/api/tool/OpenAiApiToolFunctionCallIT.java) 测试展示了如何使用底层 API 调用工具函数。基于 [OpenAI 函数调用](https://platform.openai.com/docs/guides/function-calling/parallel-function-calling) 教程实现。

**API 密钥管理**

Spring AI 通过 ApiKey 接口及其实现提供灵活的 API 密钥管理。默认实现 SimpleApiKey 适用于大多数场景，你也可创建自定义实现以满足复杂需求。

**默认配置**

默认情况下，Spring Boot 自动配置会使用 spring.ai.openai.api-key 属性创建 API Key Bean：

spring.ai.openai.api-key=your-api-key-here

Copied!

**自定义 API Key 配置**

你可通过 Builder 模式使用自定义 ApiKey 实现创建 OpenAiApi 实例：

ApiKey customApiKey = new ApiKey() {

@Override

public String getValue() {

*// Custom logic to retrieve API key*

return "your-api-key-here";

}

};

OpenAiApi openAiApi = OpenAiApi.builder()

.apiKey(customApiKey)

.build();

*// Create a chat client with the custom OpenAiApi instance*

OpenAiChatClient chatClient = new OpenAiChatClient(openAiApi);

Copied!

此方式适用于以下场景：

* 从安全密钥库获取 API Key
* 动态轮换 API Key
* 实现自定义 API Key 选择逻辑

## DeepSeek

Spring AI 支持 DeepSeek 的多种 AI 语言模型。你可以通过 DeepSeek 语言模型构建多语言对话助手。

**前提条件**

你需要创建 DeepSeek API Key 才能访问 DeepSeek 语言模型。

在 [DeepSeek 注册页面](https://platform.deepseek.com/) 创建账户，并在 [API Key 页面](https://platform.deepseek.com/api_keys) 生成访问令牌（Token）。

Spring AI 项目定义了名为 spring.ai.deepseek.api-key 的配置属性，你需将其设置为从 API Key 页面获取的密钥值。

你可以在 application.properties 文件中设置该配置属性：

spring.ai.deepseek.api-key=<your-deepseek-api-key>

Copied!

为增强处理敏感信息（如 API Key）时的安全性，可以使用 Spring 表达式语言（SpEL）引用自定义环境变量：

# In application.yml

spring:

ai:

deepseek:

api-key: ${DEEPSEEK\_API\_KEY}

Copied!

# In your environment or .env file

export DEEPSEEK\_API\_KEY=<your-deepseek-api-key>

Copied!

你还可以在应用程序代码中以编程方式设置此配置：

*// Retrieve API key from a secure source or environment variable*

String apiKey = System.getenv("DEEPSEEK\_API\_KEY");

Copied!

**添加仓库和 BOM**

Spring AI 组件已发布在 Spring Milestone 和 Snapshot 仓库。请参阅 [Artifact 仓库](https://springdoc.cn/spring-ai/getting-started.html#artifact-repositories) 章节将这些仓库添加到你的构建系统。

为简化依赖管理，Spring AI 提供 BOM 以确保整个项目使用统一版本的 Spring AI。请参阅 [依赖管理](https://springdoc.cn/spring-ai/getting-started.html#dependency-management) 章节将 Spring AI BOM 添加到你的构建系统。

**自动配置**

Spring AI 为 DeepSeek 聊天模型提供 Spring Boot 自动配置。要启用该功能，请将以下依赖添加到项目的 Maven pom.xml 文件中：

<dependency>

<groupId>org.springframework.ai</groupId>

<artifactId>spring-ai-starter-model-deepseek</artifactId>

</dependency>

Copied!

或添加到 Gradle build.gradle 文件中：

dependencies {

implementation 'org.springframework.ai:spring-ai-starter-model-deepseek'

}

Copied!

|  |  |
| --- | --- |
|  | 请参阅 [依赖管理](https://springdoc.cn/spring-ai/getting-started.html#dependency-management) 章节将 Spring AI BOM 添加到你的构建文件。 |

**聊天属性**

**重试属性**

前缀 spring.ai.retry 用于配置 DeepSeek 聊天模型的重试机制属性。

| **属性** | **说明** | **默认值** |
| --- | --- | --- |
| spring.ai.retry.max-attempts | 最大重试次数 | 10 |
| spring.ai.retry.backoff.initial-interval | 指数退避策略的初始休眠时间 | 2 sec. |
| spring.ai.retry.backoff.multiplier | 退避间隔乘数 | 5 |
| spring.ai.retry.backoff.max-interval | 最大退避时长 | 3 min. |
| spring.ai.retry.on-client-errors | 若为 false，则抛出 NonTransientAiException，且不对 4xx 客户端错误代码尝试重试 | false |
| spring.ai.retry.exclude-on-http-codes | 不应触发重试的 HTTP 状态码列表（例如用于抛出 NonTransientAiException）。 | empty |
| spring.ai.retry.on-http-codes | 应触发重试的 HTTP 状态码列表（例如用于抛出 TransientAiException）。 | empty |

**连接属性**

前缀 spring.ai.deepseek 用于配置连接 DeepSeek 服务的属性参数。

| **属性** | **说明** | **默认值** |
| --- | --- | --- |
| spring.ai.deepseek.base-url | 要连接到的 URL | [api.deepseek.com](https://api.deepseek.com/) |
| spring.ai.deepseek.api-key | API Key | - |

**连接属性**

前缀 spring.ai.deepseek.chat 用于配置 DeepSeek 聊天模型实现的属性参数。

| **属性** | **说明** | **默认值** |
| --- | --- | --- |
| spring.ai.deepseek.chat.enabled | 启用 DeepSeek 聊天模型 | true |
| spring.ai.deepseek.chat.base-url | 可选地覆盖 spring.ai.deepseek.base-url 以提供聊天专用的 URL | [api.deepseek.com/](https://api.deepseek.com/) |
| spring.ai.deepseek.chat.api-key | 可选地覆盖 spring.ai.deepseek.api-key 以提供聊天专用的 API Key | - |
| spring.ai.deepseek.chat.completions-path | 聊天补全端点的路径。 | /chat/completions |
| spring.ai.deepseek.chat.beta-prefix-path | 测试功能端点的前缀路径 | /beta/chat/completions |
| spring.ai.deepseek.chat.options.model | 要使用的模型的ID。你可以使用 deepseek-coder 或 deepseek-chat。 | deepseek-chat |
| spring.ai.deepseek.chat.options.frequencyPenalty | 介于 -2.0 到2 .0 之间的数值。正值会根据 Token 在已生成文本中的现有频率进行惩罚，从而降低模型逐字重复相同内容的可能性。 | 0.0f |
| spring.ai.deepseek.chat.options.maxTokens | 聊天补全中生成的最大 Token 数量。输入 Token 和生成 Token 的总长度受模型上下文长度限制。 | - |
| spring.ai.deepseek.chat.options.presencePenalty | 介于 -2.0 到 2.0 之间的数值。正值会根据 Token 是否已在已生成文本中出现进行惩罚，从而提高模型讨论新话题的可能性。 | 0.0f |
| spring.ai.deepseek.chat.options.stop | 最多可设置 4 个停止序列，当 API 生成这些序列时将终止输出。 | - |
| spring.ai.deepseek.chat.options.temperature | 采样温度值，范围0到2。较高值（如0.8）使输出更随机，较低值（如0.2）使输出更集中和确定。通常建议仅调整此参数或 top\_p 参数，而非同时修改两者。 | 1.0F |
| spring.ai.deepseek.chat.options.topP | 温度采样的替代方案——核采样（nucleus sampling），模型仅考虑概率质量累计达到 top\_p 阈值的token。例如0.1表示仅考虑概率质量排名前 10% 的 Token。通常建议仅调整此参数或 temperature 参数，而非同时修改两者。 | 1.0F |
| spring.ai.deepseek.chat.options.logprobs | 是否返回输出 Token 的对数概率。若为 true，则返回消息内容中每个输出 Token 的对数概率。 | - |
| spring.ai.deepseek.chat.options.topLogprobs | 0 到 20 之间的整数，指定每个 Token 位置返回的最可能 Token 数量（每个附带关联的对数概率）。若使用此参数，必须将 logprobs 设为 true。 | - |

|  |  |
| --- | --- |
|  | 你可以为 ChatModel 实现覆盖通用的 spring.ai.deepseek.base-url 和 spring.ai.deepseek.api-key 配置。若设置了 spring.ai.deepseek.chat.base-url 和 spring.ai.deepseek.chat.api-key 属性，它们将优先于通用属性生效。此特性适用于需要为不同模型使用不同 DeepSeek 账户及终端点的场景。 |
|  | 所有以 spring.ai.deepseek.chat.options 为前缀的属性，都可通过在 Prompt 调用中添加请求特定的 [运行时选项](https://springdoc.cn/spring-ai/api/chat/deepseek-chat.html#chat-options) 进行覆盖。 |

**运行时选项**

[DeepSeekChatOptions.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-deepseek/src/main/java/org/springframework/ai/deepseek/DeepSeekChatOptions.java) 提供模型配置参数，包括使用的模型名称、温度值（temperature）、频率惩罚系数（frequency penalty）等。

在启动时，默认选项可通过 DeepSeekChatModel(api, options) 构造函数或 spring.ai.deepseek.chat.options.\* 属性进行配置。

在运行时，你可以通过向 Prompt 调用添加新的、特定于请求的选项来覆盖默认配置。例如，针对特定请求覆盖默认模型和温度参数：

ChatResponse response = chatModel.call(

new Prompt(

"Generate the names of 5 famous pirates. Please provide the JSON response without any code block markers such as ```json```.",

DeepSeekChatOptions.builder()

.withModel(DeepSeekApi.ChatModel.DEEPSEEK\_CHAT.getValue())

.withTemperature(0.8f)

.build()

));

Copied!

|  |  |
| --- | --- |
|  | 除了模型专用的 [DeepSeekChatOptions](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-deepseek/src/main/java/org/springframework/ai/deepseek/DeepSeekChatOptions.java) 外，你还可以使用通过 [ChatOptionsBuilder#builder()](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-core/src/main/java/org/springframework/ai/chat/ChatOptionsBuilder.java) 创建的通用 [ChatOptions](https://github.com/spring-projects/spring-ai/blob/main/spring-ai-core/src/main/java/org/springframework/ai/chat/ChatOptions.java) 实例。 |

**示例 Controller（自动配置）**

[创建](https://start.springboot.io/) 一个新的 Spring Boot 项目，并在 pom（或gradle）依赖项中添加 spring-ai-starter-model-deepseek。

在 src/main/resources 目录下添加 application.properties 文件以启用并配置 DeepSeek 聊天模型：

spring.ai.deepseek.api-key=YOUR\_API\_KEY

spring.ai.deepseek.chat.options.model=deepseek-chat

spring.ai.deepseek.chat.options.temperature=0.8

Copied!

|  |  |
| --- | --- |
|  | 将 api-key 替换为你的 DeepSeek 凭证密钥。 |

这将创建一个可注入到类中的 DeepSeekChatModel 实现。以下是一个简单的 @Controller 类示例，使用该聊天模型进行文本生成：

@RestController

public class ChatController {

private final DeepSeekChatModel chatModel;

@Autowired

public ChatController(DeepSeekChatModel chatModel) {

this.chatModel = chatModel;

}

@GetMapping("/ai/generate")

public Map generate(@RequestParam(value = "message", defaultValue = "Tell me a joke") String message) {

return Map.of("generation", chatModel.call(message));

}

@GetMapping("/ai/generateStream")

public Flux<ChatResponse> generateStream(@RequestParam(value = "message", defaultValue = "Tell me a joke") String message) {

var prompt = new Prompt(new UserMessage(message));

return chatModel.stream(prompt);

}

}

Copied!

**聊天前缀补全**

聊天前缀补全遵循 Chat Completion API 规范，用户提供助手（assistant）的消息前缀，由模型完成后续消息内容。

使用前缀补全时，用户必须确保消息列表中的最后一条是 DeepSeekAssistantMessage 类型。

以下是一个完整的聊天前缀补全 Java 代码示例。本示例将助手的消息前缀设置为 "```python\n" 以强制模型输出 Python 代码，并通过设置stop参数为 [‘`’] 阻止模型生成额外解释：

@RestController

public class CodeGenerateController {

private final DeepSeekChatModel chatModel;

@Autowired

public ChatController(DeepSeekChatModel chatModel) {

this.chatModel = chatModel;

}

@GetMapping("/ai/generatePythonCode")

public String generate(@RequestParam(value = "message", defaultValue = "Please write quick sort code") String message) {

UserMessage userMessage = new UserMessage(message);

Message assistantMessage = DeepSeekAssistantMessage.prefixAssistantMessage("```python\\n");

Prompt prompt = new Prompt(List.of(userMessage, assistantMessage), ChatOptions.builder().stopSequences(List.of("```")).build());

ChatResponse response = chatModel.call(prompt);

return response.getResult().getOutput().getText();

}

}

Copied!

**推理模型（deepseek-reasoner）**

deepseek-reasoner 是 DeepSeek 开发的推理模型。在生成最终答案前，该模型会先生成思维链（Chain of Thought, CoT）以提高响应准确性。我们的 API 支持用户访问 deepseek-reasoner 生成的 CoT 内容，便于查看、展示和提炼。

你可以通过 DeepSeekAssistantMessage 获取 deepseek-reasoner 生成的思维链（CoT）内容。

public void deepSeekReasonerExample() {

DeepSeekChatOptions promptOptions = DeepSeekChatOptions.builder()

.model(DeepSeekApi.ChatModel.DEEPSEEK\_REASONER.getValue())

.build();

Prompt prompt = new Prompt("9.11 and 9.8, which is greater?", promptOptions);

ChatResponse response = chatModel.call(prompt);

*// Get the CoT content generated by deepseek-reasoner, only available when using deepseek-reasoner model*

DeepSeekAssistantMessage deepSeekAssistantMessage = (DeepSeekAssistantMessage) response.getResult().getOutput();

String reasoningContent = deepSeekAssistantMessage.getReasoningContent();

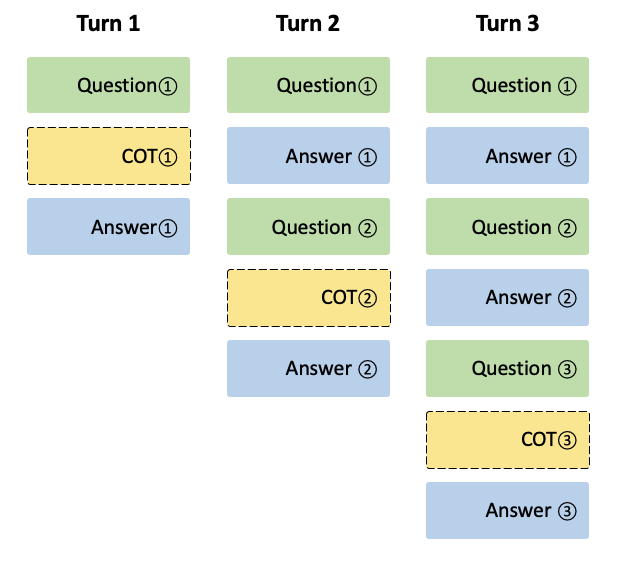
String text = deepSeekAssistantMessage.getText();

}

Copied!

**推理模型多轮对话**

在每轮对话中，模型会输出思维链（reasoning\_content）和最终答案（content）。在下一轮对话时，前几轮的思维链不会拼接到上下文中，如下所示：



请注意：若输入消息序列中包含 reasoning\_content 字段，API 将返回 400 错误。因此，在发起 API 请求前应移除 API 响应中的 reasoning\_content 字段，如 API 示例所示。

public String deepSeekReasonerMultiRoundExample() {

List<Message> messages = new ArrayList<>();

messages.add(new UserMessage("9.11 and 9.8, which is greater?"));

DeepSeekChatOptions promptOptions = DeepSeekChatOptions.builder()

.model(DeepSeekApi.ChatModel.DEEPSEEK\_REASONER.getValue())

.build();

Prompt prompt = new Prompt(messages, promptOptions);

ChatResponse response = chatModel.call(prompt);

DeepSeekAssistantMessage deepSeekAssistantMessage = (DeepSeekAssistantMessage) response.getResult().getOutput();

String reasoningContent = deepSeekAssistantMessage.getReasoningContent();

String text = deepSeekAssistantMessage.getText();

messages.add(new AssistantMessage(Objects.requireNonNull(text)));

messages.add(new UserMessage("How many Rs are there in the word 'strawberry'?"));

Prompt prompt2 = new Prompt(messages, promptOptions);

ChatResponse response2 = chatModel.call(prompt2);

DeepSeekAssistantMessage deepSeekAssistantMessage2 = (DeepSeekAssistantMessage) response2.getResult().getOutput();

String reasoningContent2 = deepSeekAssistantMessage2.getReasoningContent();

return deepSeekAssistantMessage2.getText();

}

Copied!

**手动配置**

[DeepSeekChatModel](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-deepseek/src/main/java/org/springframework/ai/deepseek/DeepSeekChatModel.java) 实现了 ChatModel 和 StreamingChatModel 接口，并通过 [底层 DeepSeekApi 客户端](https://springdoc.cn/spring-ai/api/chat/deepseek-chat.html#low-level-api) 连接 DeepSeek 服务。

将以下依赖添加到项目的 Maven pom.xml 文件中：

<dependency>

<groupId>org.springframework.ai</groupId>

<artifactId>spring-ai-deepseek</artifactId>

</dependency>

Copied!

或添加到 Gradle build.gradle 文件中：

dependencies {

implementation 'org.springframework.ai:spring-ai-deepseek'

}

Copied!

|  |  |
| --- | --- |
|  | 请参阅 [依赖管理](https://springdoc.cn/spring-ai/getting-started.html#dependency-management) 章节将 Spring AI BOM 添加到你的构建文件。 |

接下来，创建 DeepSeekChatModel 并用于文本生成：

var deepSeekApi = new DeepSeekApi(System.getenv("DEEPSEEK\_API\_KEY"));

var chatModel = new DeepSeekChatModel(deepSeekApi, DeepSeekChatOptions.builder()

.withModel(DeepSeekApi.ChatModel.DEEPSEEK\_CHAT.getValue())

.withTemperature(0.4f)

.withMaxTokens(200)

.build());

ChatResponse response = chatModel.call(

new Prompt("Generate the names of 5 famous pirates."));

*// Or with streaming responses*

Flux<ChatResponse> streamResponse = chatModel.stream(

new Prompt("Generate the names of 5 famous pirates."));

Copied!

DeepSeekChatOptions 提供聊天请求的配置信息，其内置的 DeepSeekChatOptions.Builder 支持 Fluent 式选项构建。

**底层 DeepSeekApi 客户端**

[DeepSeekApi](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-deepseek/src/main/java/org/springframework/ai/deepseek/api/DeepSeekApi.java) 是用于 [DeepSeek API](https://platform.deepseek.com/api-docs/) 的轻量级 Java 客户端。

以下是编程方式调用 API 的简单示例：

DeepSeekApi deepSeekApi =

new DeepSeekApi(System.getenv("DEEPSEEK\_API\_KEY"));

ChatCompletionMessage chatCompletionMessage =

new ChatCompletionMessage("Hello world", Role.USER);

*// Sync request*

ResponseEntity<ChatCompletion> response = deepSeekApi.chatCompletionEntity(

new ChatCompletionRequest(List.of(chatCompletionMessage), DeepSeekApi.ChatModel.DEEPSEEK\_CHAT.getValue(), 0.7f, false));

*// Streaming request*

Flux<ChatCompletionChunk> streamResponse = deepSeekApi.chatCompletionStream(

new ChatCompletionRequest(List.of(chatCompletionMessage), DeepSeekApi.ChatModel.DEEPSEEK\_CHAT.getValue(), 0.7f, true));

Copied!

更多详细信息请参阅 [DeepSeekApi.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-deepseek/src/main/java/org/springframework/ai/deepseek/api/DeepSeekApi.java) 的 JavaDoc 文档。

**DeepSeekApi 示例**

[DeepSeekApiIT.java](https://github.com/spring-projects/spring-ai/blob/main/models/spring-ai-deepseek/src/test/java/org/springframework/ai/deepseek/api/DeepSeekApiIT.java) 测试类提供了一些通用示例，展示了如何使用该轻量级库与 DeepSeek 提供的 API 进行交互。