

> Menu

概述

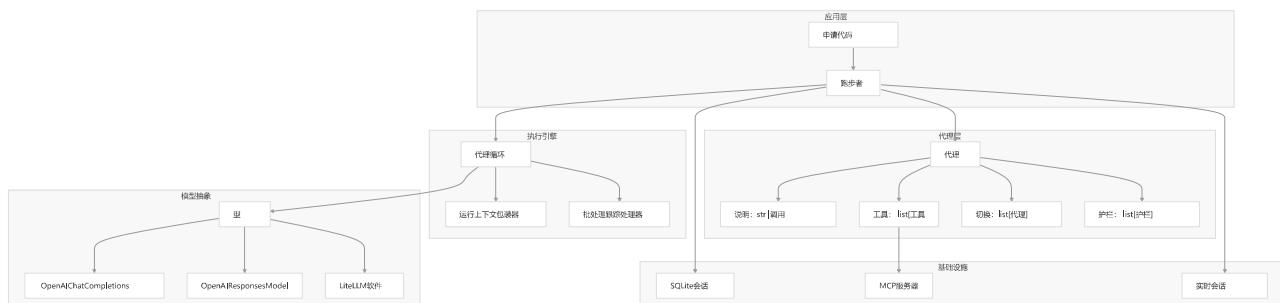
相关源文件

本文档简要介绍了 OpenAI Agents SDK，这是一个用于构建多代理工作流的 Python 框架。它涵盖了 SDK 的架构、核心组件和基本概念。有关安装和快速入门说明，请参阅[入门](#)。有关特定组件的详细信息，请参阅[核心组件和高级功能](#)。

目的和架构

OpenAI Agents SDK 是一个轻量级但功能强大的框架，旨在构建具有多个协作代理的对话式 AI 应用程序。该 SDK 抽象化了管理 LLM 交互、对话状态和代理协调的复杂性，同时通过工具、交接和自定义集成提供可扩展性。

高级系统架构



来源: pyproject.toml | 1-141 | README.md | 1-312 | src/agents/util/ init .py | 1

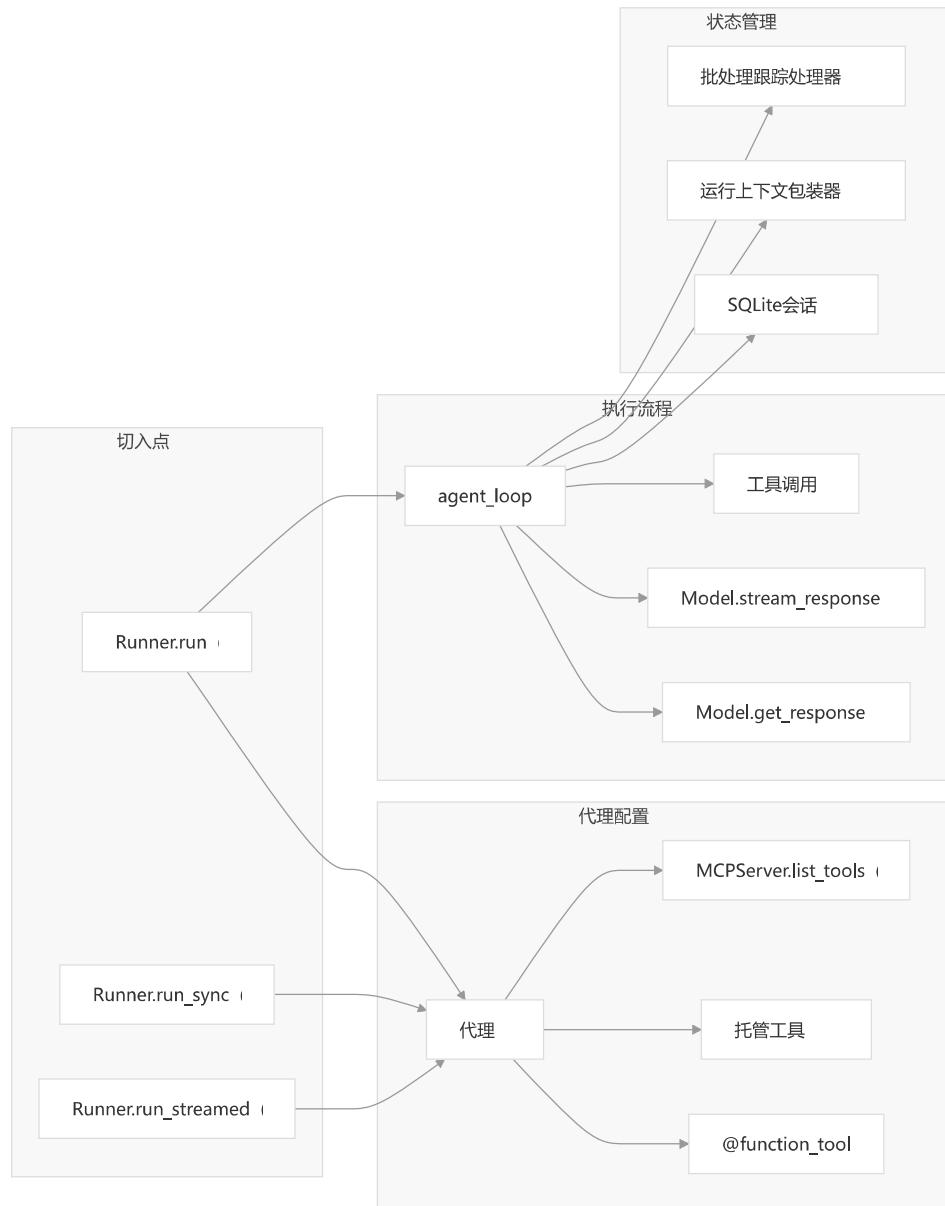
核心组件关系

向 Devin 询问有关 openai/openai-agents-python 的信息

深入研究



→



来源： README.md | 48-126 | 测试/语音/test_workflow.py | 32-98

向 Devin 询问有关 openai/openai-agents-python 的信息

深入研究

该类是封装具有配置、指令、工具和行为约束的 LLM 的中心抽象。每个代理代表工作流程中的专用角色。 Agent

工具和功能

工具通过 Python 函数的装饰器、预构建功能的托管工具以及用于外部服务集成的 MCP（模型上下文协议）服务器扩展代理功能。 @function_tool

切换

切换通过专门的工具调用机制实现代理到代理委派。它们允许复杂的工作流程，其中不同的代理处理任务的不同方面。

代理循环

执行模型以代理循环为中心，该循环：

1. 使用当前上下文和消息历史记录调用 LLM
2. 处理工具调用和函数调用
3. 处理交接给其他代理
4. 持续到达最终输出条件
5. 在整个过程中管理对话状态和跟踪

会话和记忆

该类提供自动对话历史记录管理，消除了代理运行之间的手动状态处理。自定义会话实现可以与外部存储系统集成。 SQLiteSession

跟踪和可观测性

通过捕获代理执行流程、工具调用和性能指标的内置跟踪。该系统支持与外部可观测性平台集成。

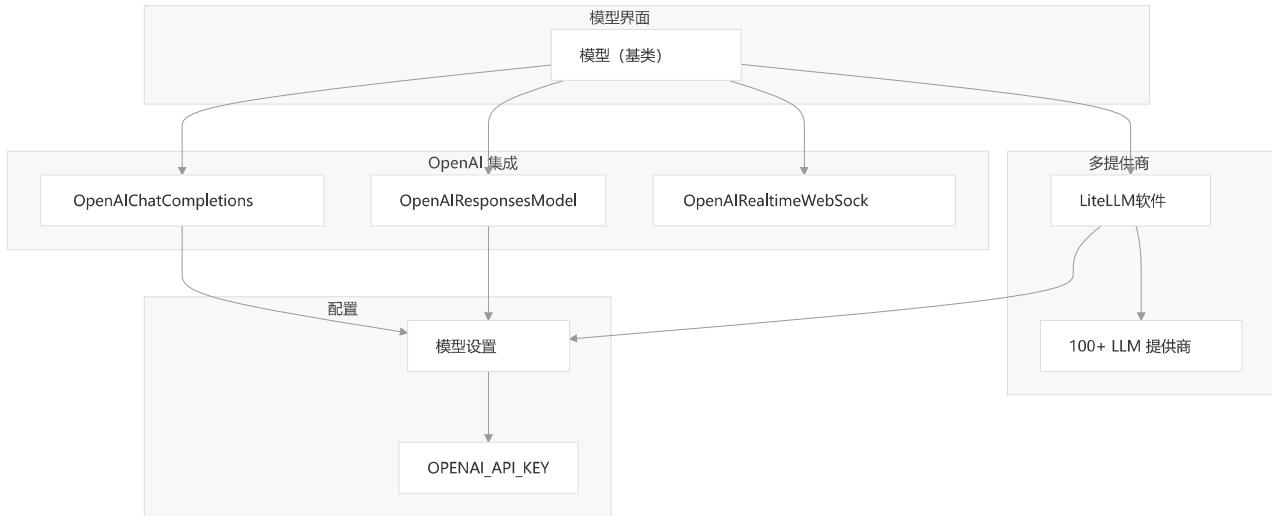
BatchTraceProcessor

来源： README.md | 10-17 README.md | 128-151 README.md | 156-158

模型提供者支持

向 Devin 询问有关 openai/openai-agents-python 的信息

深入研究



SDK 通过通用接口抽象模型交互，同时支持多个后端： `Model`

- 用于标准文本交互的 **OpenAI 聊天完成 API**
- 用于结构化输出和函数调用的 **OpenAI 响应 API**
- 用于语音和流交互的 **OpenAI 实时 API**
- **LiteLLM 集成**，支持 100+ 其他模型提供商

来源： `pyproject.toml` | 39 `README.md` | 3 `README.md` | 305-306

高级功能

实时和语音处理

和语音工作流程组件通过复杂的中断处理和轮次管理实现实时音频交互。 `RealtimeSession`

可视化和调试

可选依赖项通过代理结构和执行流的图形表示形式提供代理工作流可视化。 `viz`

扩展

向 Devin 询问有关 `openai/openai-agents-python` 的信息

深入研究



> 菜单

入门

[相关源文件](#)

本快速入门指南将向您展示如何使用 OpenAI Agents SDK 创建您的首个代理。您将通过实际示例学习基本的代理创建、工具使用和执行模式。

安装和设置

环境设置

设置 Python 环境：

```
# Using uv (recommended)
uv venv
source .venv/bin/activate # On Windows: .venv\Scripts\activate

# Or using venv
python -m venv env
source env/bin/activate # On Windows: env\Scripts\activate
```

安装代理 SDK：

```
pip install openai-agents
```

可选依赖项：

```
pip install 'openai-agents[voice]'      # For voice support (numpy, websockets)
pip install 'openai-agents[viz]'        # For visualization (graphviz)
pip install 'openai-agents[litellm]'    # For LiteLLM provider support
pip install 'openai-agents[realtime]'   # For realtime features (websockets)
```

向 Devin 询问 openai/openai-agents-python

深入研究



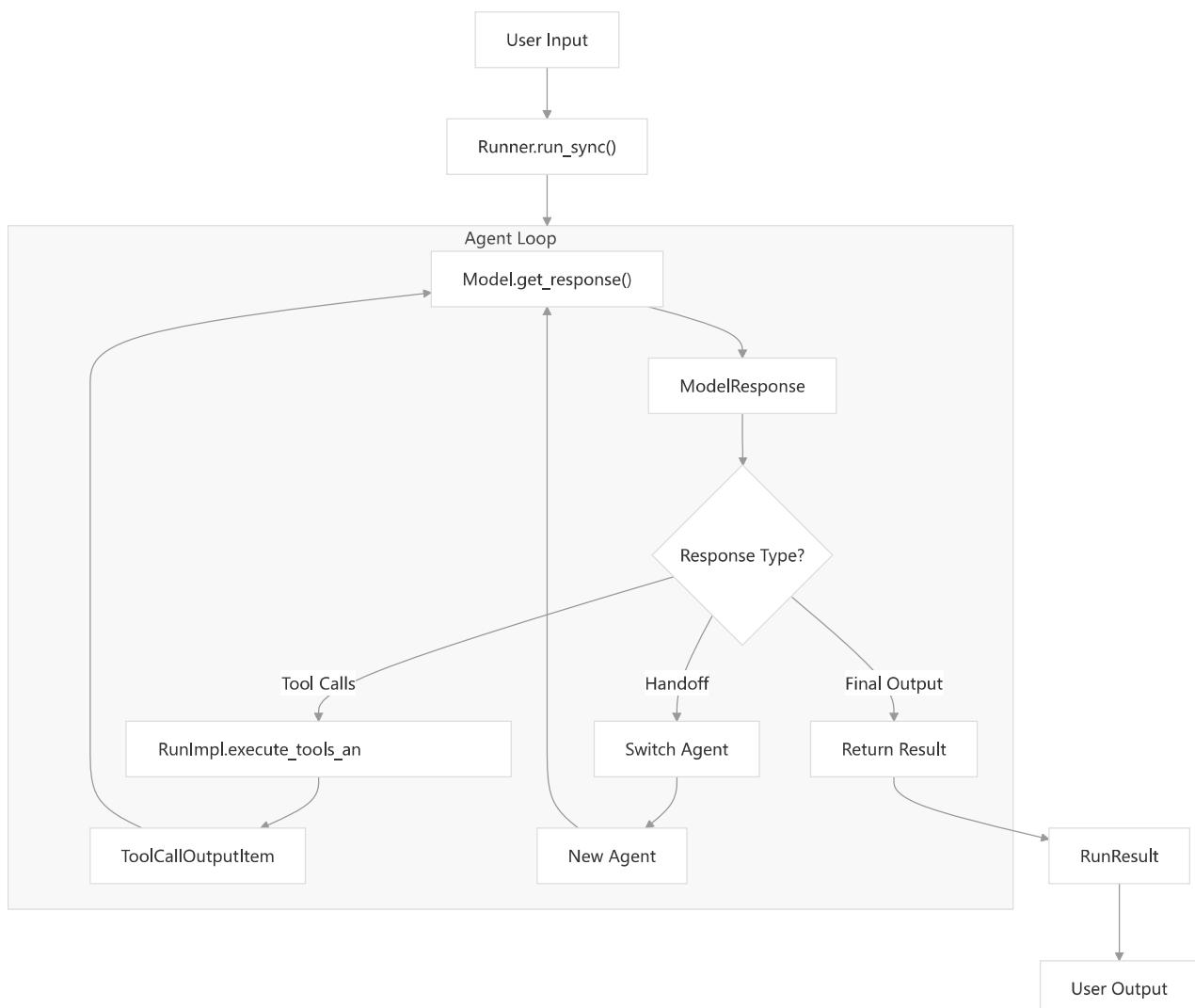
```
export OPENAI_API_KEY="your-api-key-here"
```

资料来源： 自述文件.md | 21-44 pyproject.toml | 36-40

核心组件

OpenAI Agents SDK 围绕三个主要组件： Agent 、 Runner 和工具。

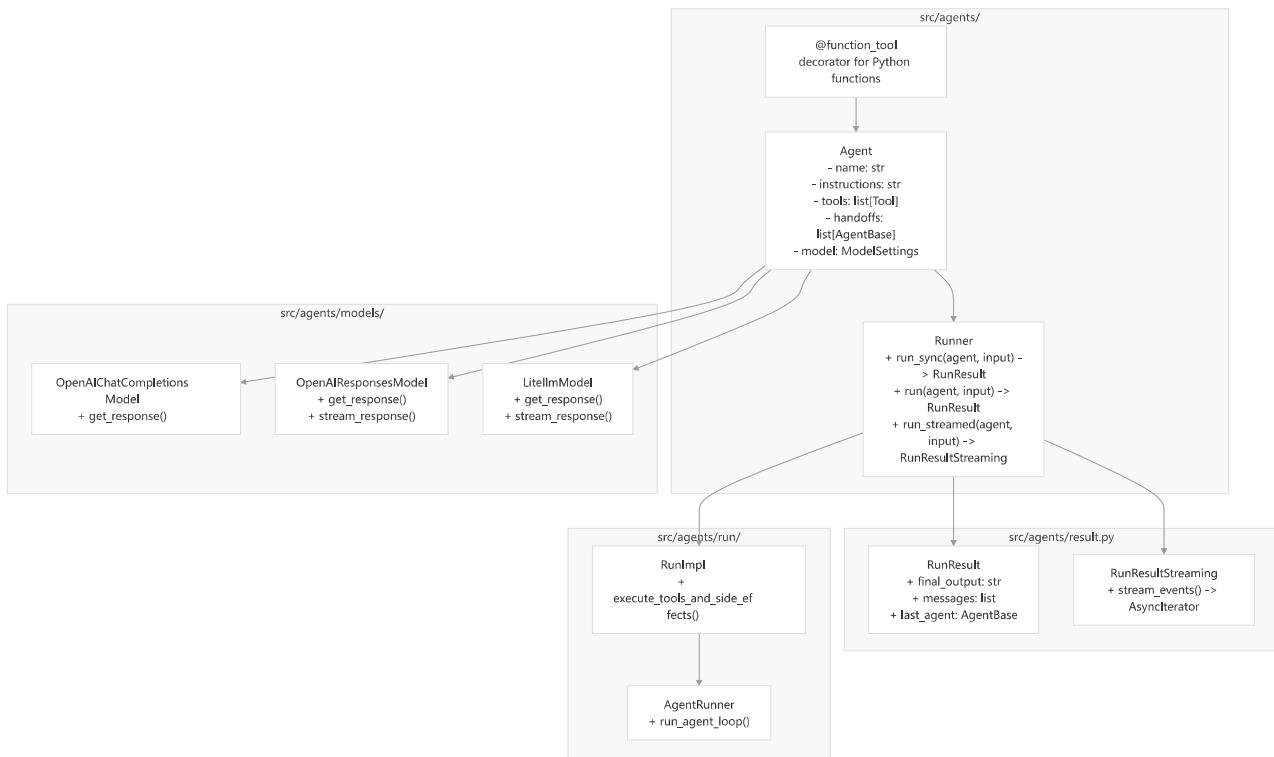
代理执行流程



核心类和方法

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源：自述文件.md | 48-200 pyproject.toml | 9-16

您的第一位经纪人

基本 Hello World

最简单的代理只需要一个名称和指令：

```

from agents import Agent, Runner

agent = Agent(name="Assistant", instructions="You are a helpful assistant")

result = Runner.run_sync(agent, "Write a haiku about recursion in programming.")
print(result.final_output)

# Code within the code,
# Functions calling themselves,
# Infinite Loop's dance.

```

这将创建一个 `Agent` 实例并用于 `Runner.run_sync()` 同步执行。其中 `RunResult.final_output` 包含代理的

向 Devin 询问 openai/openai-agents-python

深入研究

当您调用 `Runner.run_sync()` 或时 `Runner.run()`，SDK 会执行一个循环：

1. **模型调用**: 将对话历史记录发送到配置的语言模型
2. **响应处理**: 解析模型对工具调用、交接或最终输出的响应
3. **工具执行**: 如果存在工具调用，则并行执行它们
4. **切换处理**: 如果请求切换，则切换到新的代理
5. **循环继续**: 返回步骤 1 并更新上下文
6. **最终输出**: 当不需要工具/交接时，返回结果

循环持续进行，直到产生或 `max_turns` 达到最终输出。

资料来源：自述文件.md | 125-135

添加工具

功能工具

工具可以通过将代理连接到外部系统或自定义逻辑来扩展其功能。使用 `@function_tool` 装饰器将 Python 函数转换为代理工具：

```
import asyncio

from agents import Agent, Runner, function_tool


@function_tool
def get_weather(city: str) -> str:
    return f"The weather in {city} is sunny."


agent = Agent(
    name="Hello world",
    instructions="You are a helpful agent.",
    tools=[get_weather],
)

async def main():
    result = await Runner.run(agent, input="What's the weather in Tokyo?")
```

向 Devin 询问 openai/openai-agents-python

深入研究

```
if __name__ == "__main__":
    asyncio.run(main())
```

工具执行流程

向 Devin 询问 openai/openai-agents-python

深入研究

向 Devin 询问 openai/openai-agents-python

深入研究



装饰 `@function_tool` 器根据您的函数签名和文档字符串自动生成工具模式，使该工具可被语言模型调用。

资料来源： 自述文件.md | 100-126

座席交接

多代理工作流

切换使代理能够根据上下文或用户需求将控制权转移给专门的代理：

```

from agents import Agent, Runner
import asyncio

spanish_agent = Agent(
    name="Spanish agent",
    instructions="You only speak Spanish.",
)

english_agent = Agent(
    name="English agent",
    instructions="You only speak English",
)

triage_agent = Agent(
    name="Triage agent",
    instructions="Handoff to the appropriate agent based on the language of the request.",
    handoffs=[spanish_agent, english_agent],
)
  
```



```

async def main():
    result = await Runner.run(triage_agent, input="Hola, ¿cómo estás?")
    print(result.final_output)
  
```

向 Devin 询问 openai/openai-agents-python

深入研究

切换流程

向 Devin 询问 openai/openai-agents-python

深入研究

向 Devin 询问 openai/openai-agents-python

深入研究



`handoffs` 类上的参数定义 `Agent` 了当前代理可以将控制权转移给哪些代理。该 `Runner` 进程负责从模型中切换调用，并将执行上下文切换到目标代理。

资料来源：自述文件.md | 67-96

执行模式

同步与异步

向 Devin 询问 openai/openai-agents-python

深入研究

> 菜单

核心组件

[相关源文件](#)

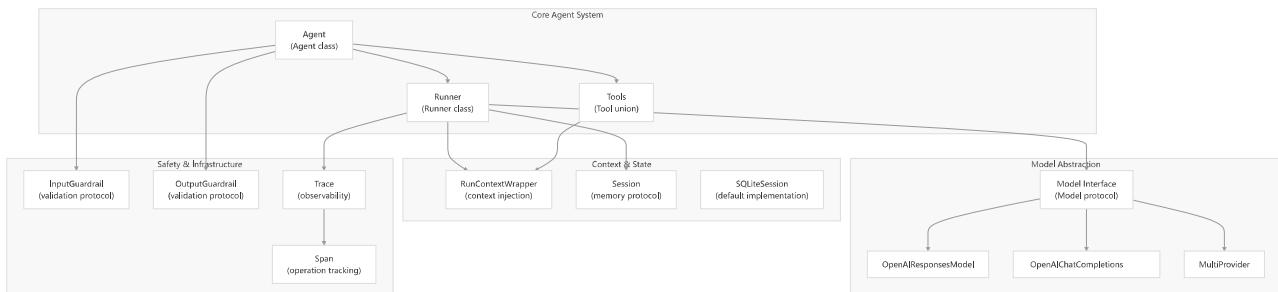
本文档涵盖了 OpenAI Agents SDK 的基本构建块，这些构建块协同工作，用于创建和执行 AI 代理应用程序。这些组件构成了所有代理功能的基础。

有关特定组件的详细信息，请参阅以下部分：[代理](#)、[工具和功能模式](#)、[运行器和执行](#)、[模型和提供程序](#)、[上下文管理](#)、[护栏和安全](#)以及[会话管理](#)。有关跟踪和 MCP 集成等高级功能，请参阅[高级功能](#)。

组件架构概述

SDK 遵循分层架构，其中每个核心组件在代理执行管道中都有特定的职责。

核心组件关系



资料来源： [src/agents/agent.py](#) | 125-301 [src/agents/tool.py](#) | 277-286

[src/agents/__init__.py](#) | 1-293

代理人：中央协调者

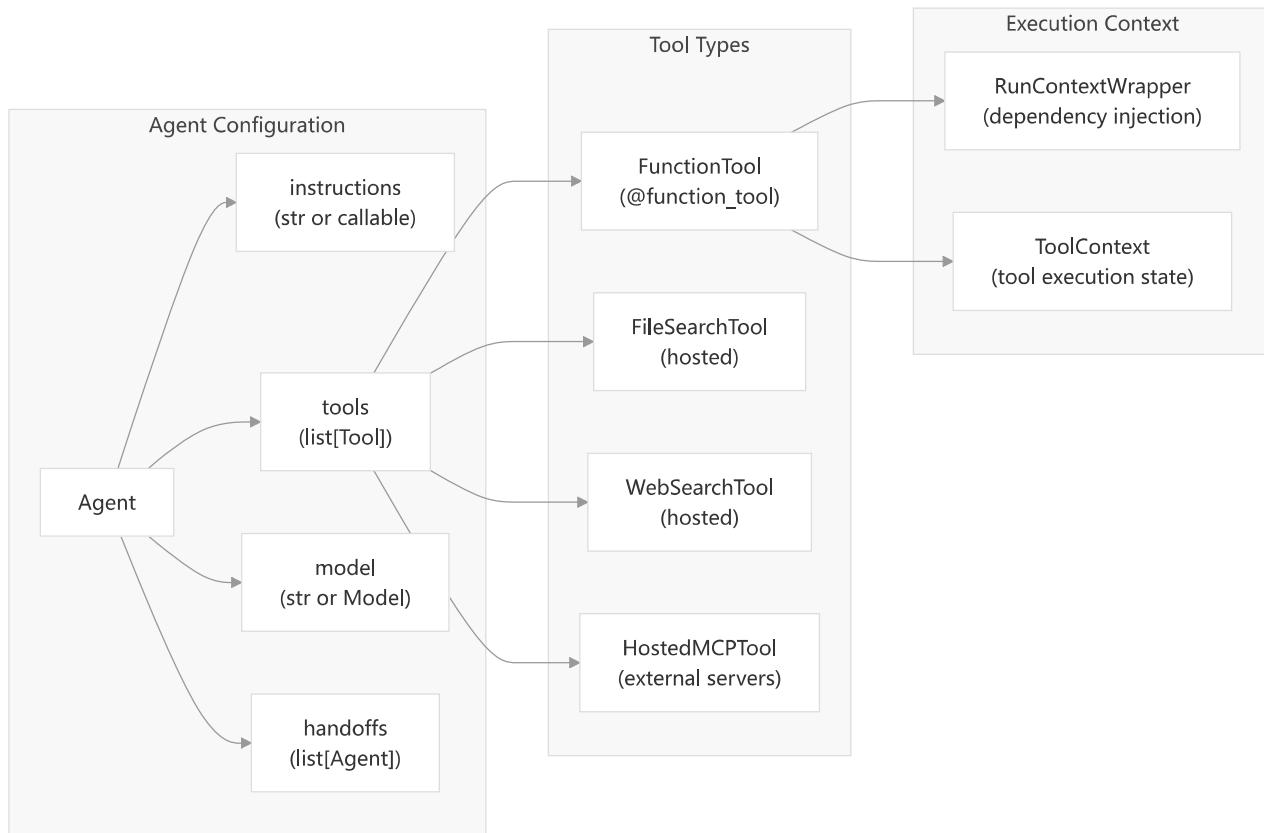
该类 **Agent** 是定义 AI 代理的主要接口。它封装了创建功能代理所需的所有配置，包括指令、工具和行为策略。

向 Devin 询问 [openai/openai-agents-python](#)

深入研究

成分	类型	目的
tools	list[Tool]	可用的功能和能力
handoffs	list[Agent Handoff]	专门任务的授权目标
model	str Model	要使用的 LLM 实现
guardrails	list[InputGuardrail OutputGuardrail]	安全验证机制

代理配置和工具集成



资料来源： src/agents/agent.py | 125-172 src/agents/tool.py | 277-286

工具生态系统

工具是代理与外部系统交互并执行操作的主要机制。SDK 通过统一的 `Tool` 联合类型支持多种工具类型。

项目类别	功能	示例
向 Devin 询问 openai/openai-agents-python		

深入研究

工具类型	班级	用例
网页搜索	WebSearchTool	互联网搜索功能
计算机使用	ComputerTool	GUI自动化和控制
MCP 工具	HostedMCPTool	外部 MCP 服务器集成
代码解释器	CodeInterpreterTool	沙盒代码执行
图像生成	ImageGenerationTool	AI图像创作

装饰 `function_tool` 器提供了从 Python 函数创建自定义工具的主要接口：

```
@function_tool
def my_tool(context: RunContextWrapper, param: str) -> str:
    """Tool description for the LLM."""
    return f"Result: {param}"
```

资料来源： src/agents/tool.py | 277-286 src/agents/tool.py | 329-470

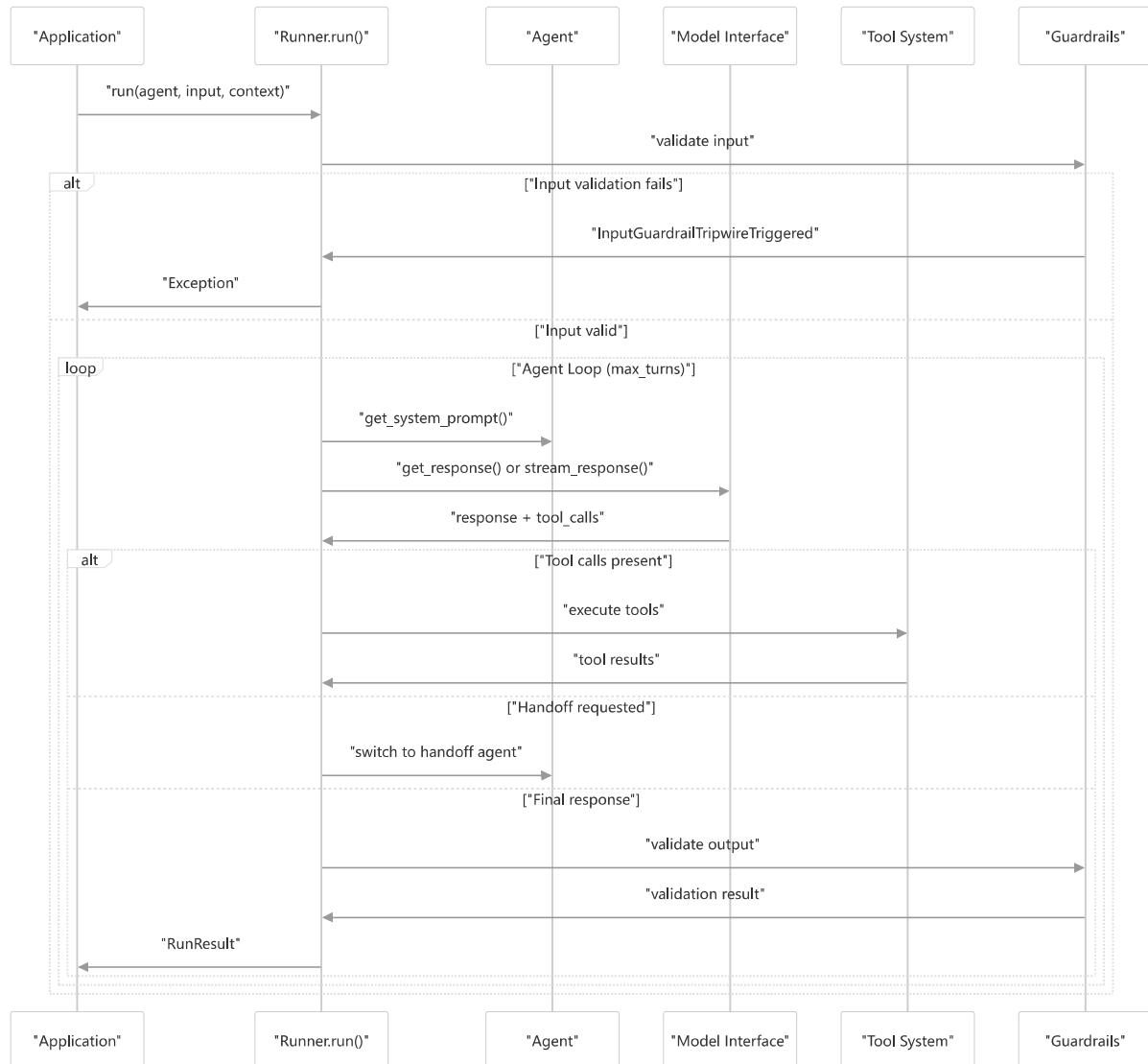
执行引擎

该类 `Runner` 管理完整的代理执行生命周期，包括代理循环、工具调用、交接和结果生成。

代理执行流程

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源： src/agents/run.py src/agents/agent.py | 282-301

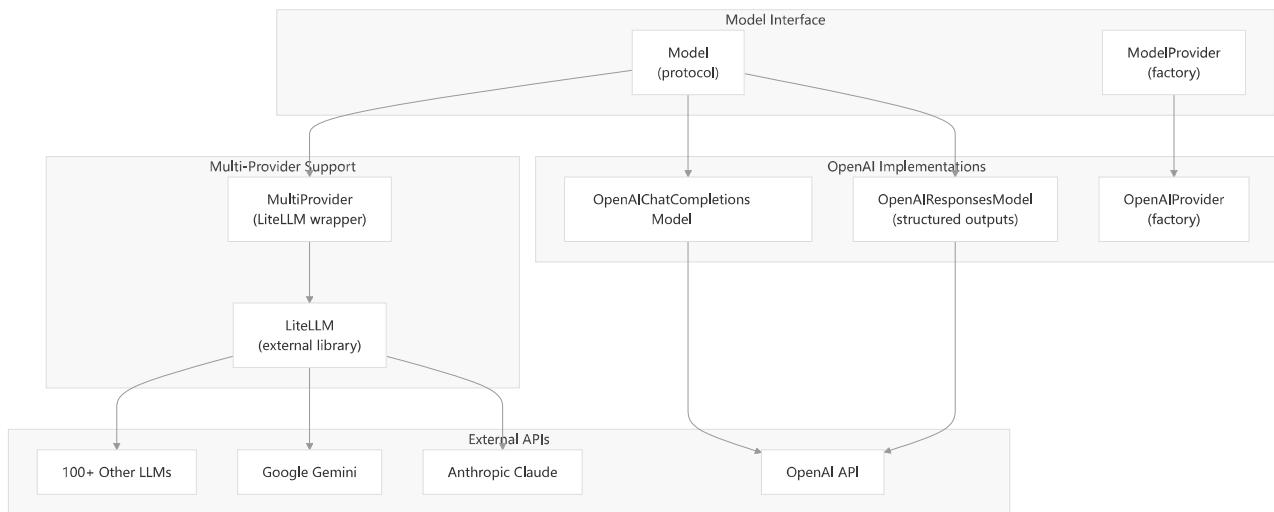
模型抽象层

SDK 提供了灵活的模型抽象，通过统一的接口支持多个 LLM 提供程序。

模型实现	目的	API
<code>OpenAIResponsesModel</code>	OpenAI Responses API 支持	结构化输出，托管工具
<code>OpenAIChatCompletionsModel</code>	OpenAI 聊天完成 API	传统聊天界面
<code>MultiProvider</code>	LiteLLM 集成	100多家法学硕士（LLM）提供商

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源：[src/agents/models/interface.py](#) [src/agents/models/openai_responses.py](#)

[src/agents/models/multi_provider.py](#)

上下文管理系统

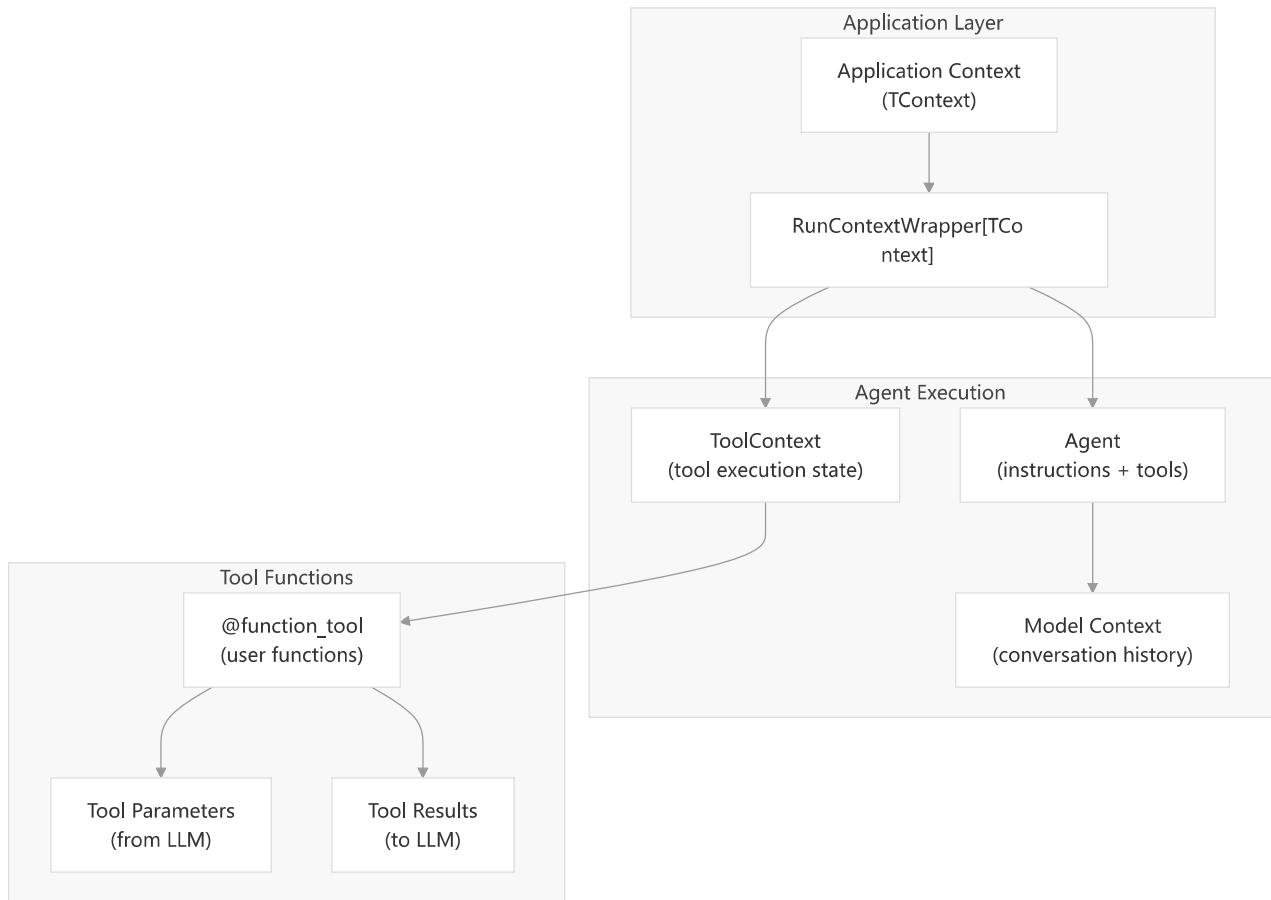
上下文管理将应用程序状态与 LLM 可见上下文分离，从而实现安全灵活的代理交互。

成分	目的	范围
<code>RunContextWrapper[TContext]</code>	依赖注入容器	本地应用程序状态
<code>ToolContext</code>	工具执行环境	特定于工具的上下文
代理上下文	LLM对话状态	模型可见的历史记录

上下文流架构

向 Devin 询问 [openai/openai-agents-python](#)

深入研究



资料来源： [src/agents/run_context.py](#) [src/agents/tool_context.py](#) [src/agents/agent.py](#) | 282-301

基础设施组件

几个基础设施组件为生产代理应用程序提供了必要的功能。

护栏系统

护栏提供输入和输出验证以确保代理行为的安全：

- `InputGuardrail[TContext]` - 在处理之前验证用户输入
- `OutputGuardrail[TContext]` - 返回前验证代理输出
- `@input_guardrail` 以及 `@output_guardrail` 易于创建的装饰器

会话管理

会话管理是通过 `ConversationManager` 实现的。

向 Devin 询问 [openai/openai-agents-python](#)

深入研究



> 菜单

代理商

相关源文件

本文档涵盖了 `Agent` 类，它是 OpenAI Agents SDK 中创建 AI 助手的核心构建块。代理封装了 LLM 配置、指令、工具和行为模式，以创建可重用的 AI 组件。

有关执行代理的信息，请参阅[Runner 和 Execution](#)。有关工具创建和集成的详细信息，请参阅[工具和功能模式](#)。有关代理对话历史记录管理，请参阅[会话管理](#)。

代理类结构

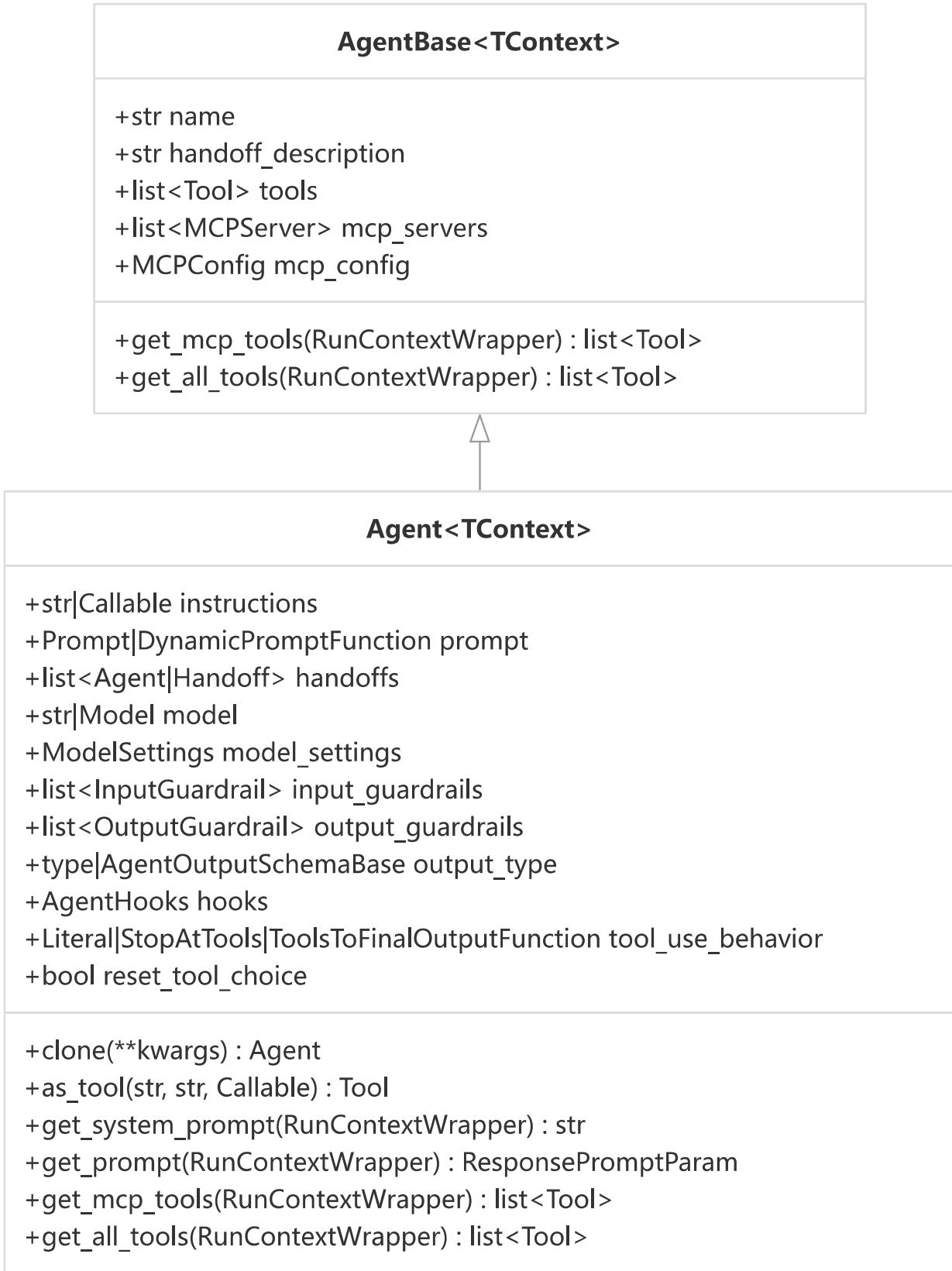
该类 `Agent` 被定义为一个数据类，它继承自 `AgentBase` 上下文类型，并且是其通用类型 `TContext`。此结构支持类型安全的依赖注入和配置重用。

代理类层次结构

向 Devin 询问 openai/openai-agents-python

深入研究





资料来源： src/agents/agent.py 69-123 src/agents/agent.py 125-319

向 Devin 询问 openai/openai-agents-python

深入研究

财产	类型	目的
name	str	代理所需的标识符
instructions	str function	系统提示或动态指令生成器
model	str Model	LLM 实现（默认为“gpt-4o”）
tools	list[Tool]	代理可用的工具

高级配置

财产	类型	目的
prompt	Prompt DynamicPromptFunction	Responses API 提示配置
handoffs	list[Agent[Any] Handoff[TContext, Any]]	委派分代理
output_type	type[Any] AgentOutputSchemaBase	结构化输出模式
model_settings	ModelSettings	温度、top_p、tool_choice配置
input_guardrails	list[InputGuardrail[TContext]]	执行前验证
output_guardrails	list[OutputGuardrail[TContext]]	执行后验证
tool_use_behavior	Literal StopAtTools ToolsToFinalOutputFunction	控制如何处理工具调用
hooks	AgentHooks[TContext]	生命周期事件回调
reset_tool_choice	bool	工具使用后是否重置 tool_choice

资料来源： src/agents/agent.py | 139-223

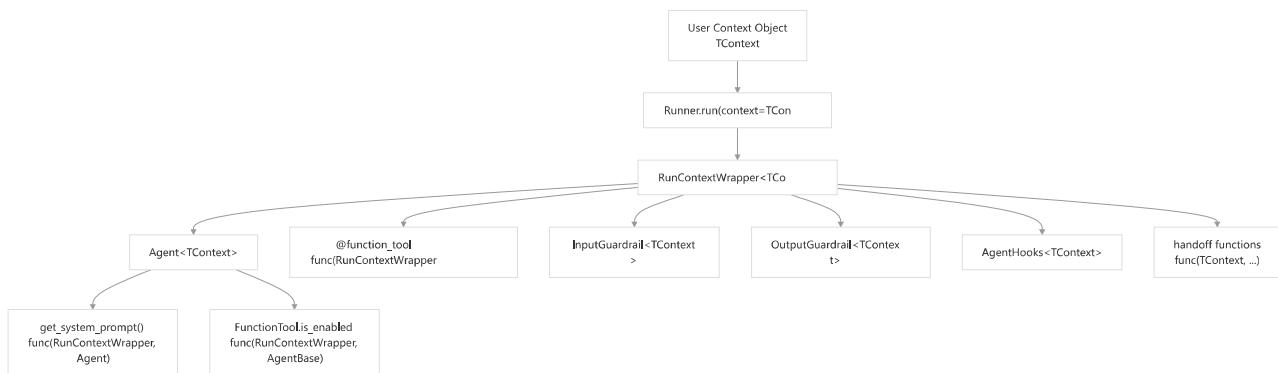
上下文类型系统

代理使用泛型类型来确保整个执行管道的类型安全。上下文类型 `TContext` 在代理运行中的所有组件中必须保持一致。

上下文流架构

向 Devin 询问 openai/openai-agents-python

深入研究

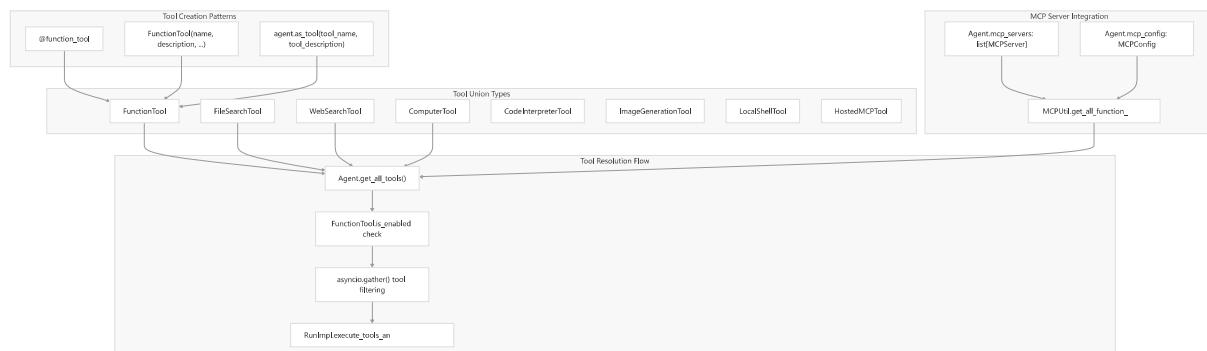


资料来源： src/agents/agent.py | 126 src/agents/run_context.py src/agents/tool.py | 91-94

工具集成架构

代理支持多种工具集成模式，每种模式具有不同的执行特性和功能。

工具类型层次结构



资料来源： src/agents/agent.py | 104-122 src/agents/tool.py | 277-286
src/agents/tool.py | 298-470

代理方法和生命周期

关键代理方法

该类 `Agent` 提供了几个重要的配置和执行方法：

方法	签名	目的
向 Devin 询问 openai/openai-agents-python		

深入研究

方法

	签名	目的
as_tool(tool_name, tool_description, custom_output_extractor)	Tool	将代理转换为 FunctionTool 使用 @function_tool 装饰器
get_system_prompt(run_context)	str None	解析 instructions (静态字符串或可调用)
get_prompt(run_context)	ResponsePromptParam None	prompt 通过以下方式解决 PromptUtil.to_model_input()
get_mcp_tools(run_context)	list[Tool]	mcp_servers 从 via 获取工具 MCPUtil.get_all_function_tools()
get_all_tools(run_context)	list[Tool]	聚合 MCP 工具 + tools 通过启用 asyncio.gather()

动态指令解析

向 Devin 询问 openai/openai-agents-python

深入研究

工具使用行为配置

该 `tool_use_behavior` 属性控制 `FunctionTool` 执行后如何处理调用：

行为类型	价值	处理逻辑
<code>Literal["run_llm_again"]</code>	<code>"run_llm_again"</code>	默认：工具运行，结果发送到 LLM 进行处理
<code>Literal["stop_on_first_tool"]</code>	<code>"stop_on_first_tool"</code>	第一个工具输出 <code>final_output</code> 直接变为
<code>StopAtTools</code>	<code>{"stop_at_tool_names": ["tool1", "tool2"]}</code>	当调用任何指定工具时停止
<code>ToolsToFinalOutputFunction</code>	<code>Callable[[RunContextWrapper, list[FunctionToolResult]], MaybeAwaitable[ToolsToFinalOutputResult]]</code>	自定义函数处理工具结果

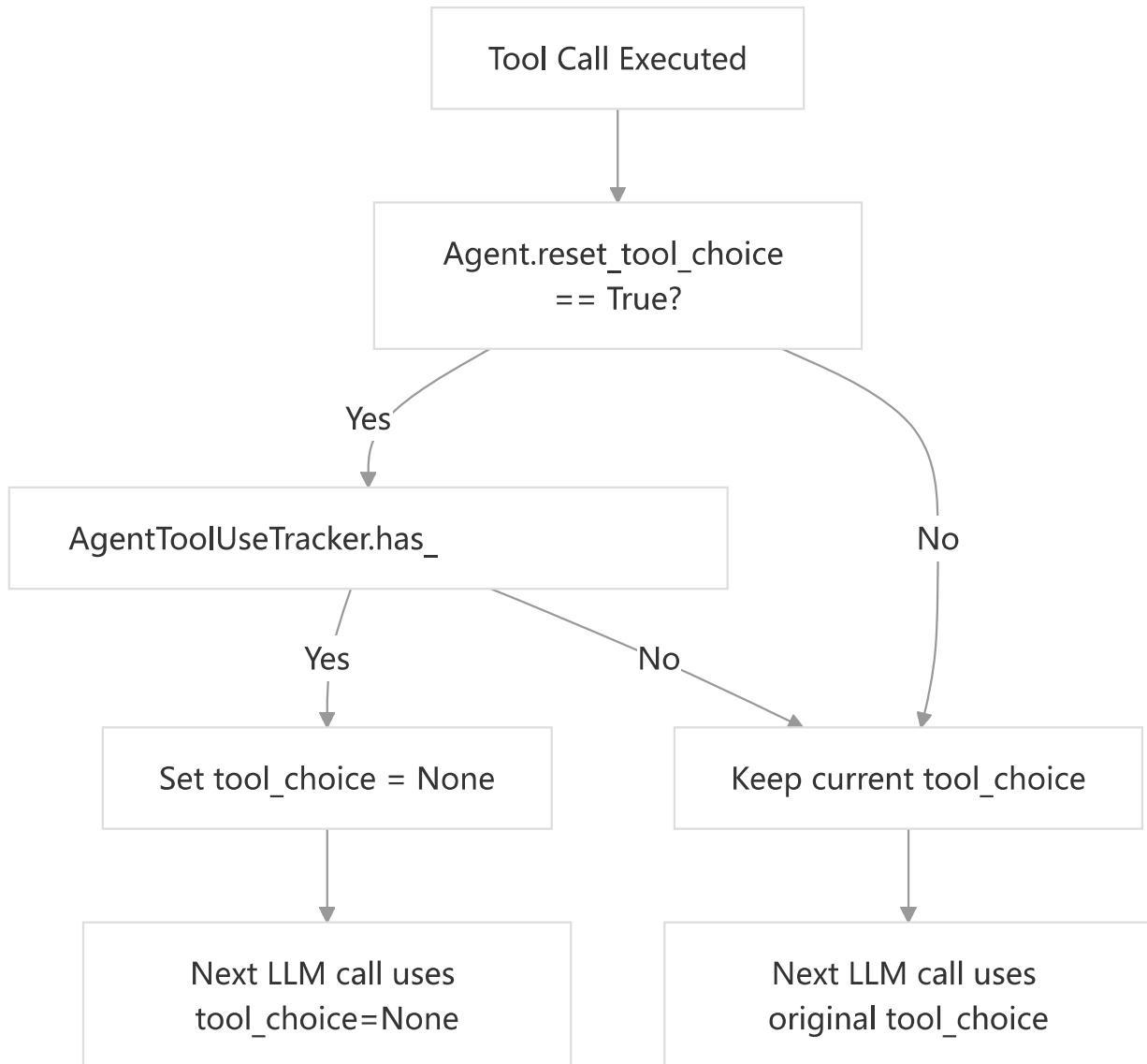
工具选择重置机制

该 `reset_tool_choice` 属性（默认 `True`）通过以下方式防止无限循环

`RunImpl.maybe_reset_tool_choice()` :

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源： src/agents/agent.py | 201-223 src/agents/agent.py | 46-52

测试/test_tool_choice_reset.py | 10-63

代理作为工具模式

使用该方法可以将代理转换为工具 `as_tool()`，从而实现多代理编排模式：



向 Devin 询问 openai/openai-agents-python

深入研究



> 菜单

工具和功能模式

[相关源文件](#)

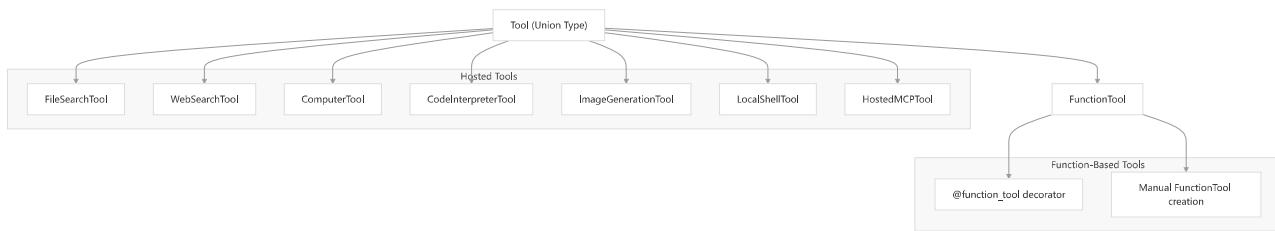
本页介绍 OpenAI Agents SDK 中的工具系统，包括如何将 Python 函数转换为语言模型可以理解和调用的工具模式。本页重点介绍定义、配置以及将工具与代理集成的核心机制。

有关使用工具运行代理的信息，请参阅[运行器和执行](#)。有关模型上下文协议 (MCP) 集成的详细信息，请参阅[模型上下文协议 \(MCP\)](#)。

工具系统概述

SDK 提供了统一的工具体系，允许 Agent 通过不同类型的工具来执行动作。系统将 Python 函数和外部服务转换为语言模型可以理解和调用的标准化工具接口。

工具类型层次结构



工具系统架构

资料来源： [src/agents/tool.py](#) | 277-287 [src/agents/__init__.py](#) | 67-85

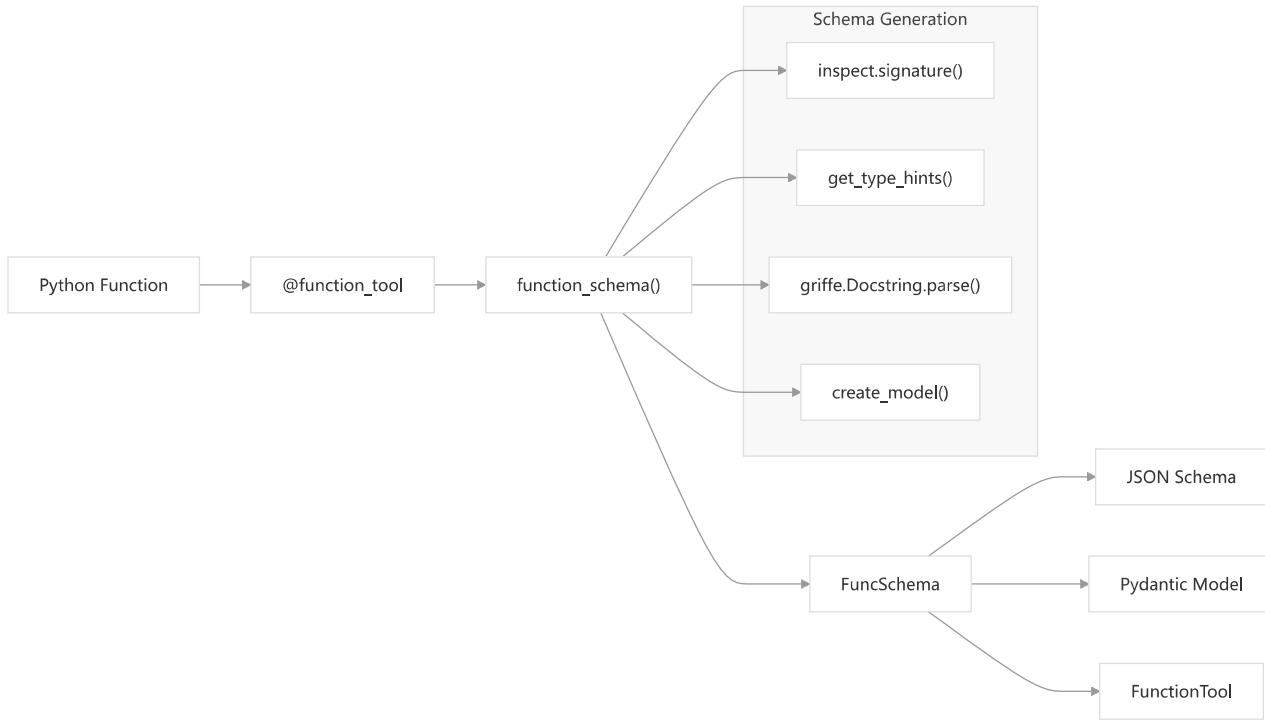
功能工具和模式生成

函数工具是从 Python 函数创建自定义工具的主要机制。系统会自动根据函数签名和文档字符串生成 JSON 模式。

向 Devin 询问 [openai/openai-agents-python](#)

深入研究





功能模式生成管道

资料来源： src/agents/function_schema.py | 188-217 src/agents/tool.py | 329-370

核心架构组件

数据类 `FuncSchema` 捕获将 Python 函数转换为工具所需的所有元数据：

成分	类型	目的
<code>name</code>	<code>str</code>	向 LLM 显示的工具名称
<code>description</code>	<code>str None</code>	来自文档字符串的工具描述
<code>params_pydantic_model</code>	<code>type[BaseModel]</code>	Pydantic 参数模型
<code>params_json_schema</code>	<code>dict[str, Any]</code>	参数的 JSON 架构
<code>signature</code>	<code>inspect.Signature</code>	原始函数签名
<code>takes_context</code>	<code>bool</code>	函数是否接受上下文
<code>strict_json_schema</code>	<code>bool</code>	架构是否为严格模式

向 Devin 询问 openai/openai-agents-python

深入研究

```

@function_tool
def get_weather(city: str) -> str:
    """Get weather for a city.

    Args:
        city: The city name.
    """
    return f"Weather in {city}: sunny"

@function_tool(
    name_override="custom_name",
    description_override="Custom description",
    strict_mode=True,
    is_enabled=True
)
def advanced_tool(ctx: ToolContext, data: str) -> str:
    return f"Processed: {data}"

```

装饰器支持以下配置选项：

范围	类型	默认	目的
name_override	str None	None	覆盖函数名称
description_override	str None	None	覆盖文档字符串描述
docstring_style	DocstringStyle None	自动检测	文档字符串解析风格
use_docstring_info	bool	True	解析文档字符串以获取元数据
failure_error_function	ToolErrorFunction None	默认处理程序	错误处理函数
strict_mode	bool	True	启用严格的 JSON 模式
is_enabled	bool Callable	True	工具可用性控制

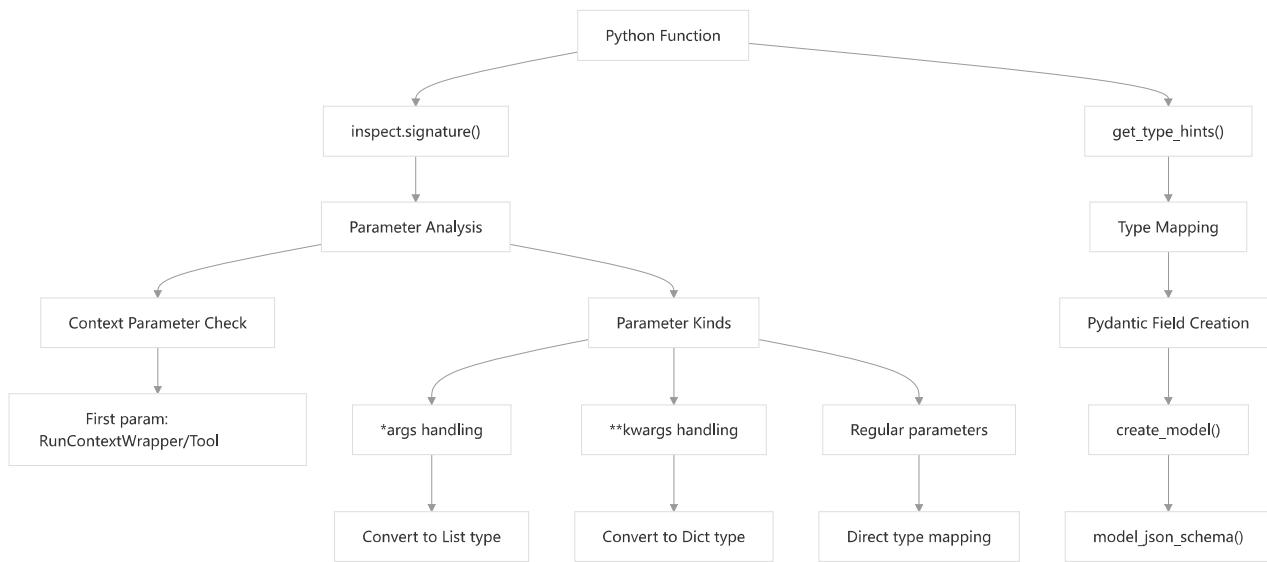
资料来源： src/agents/tool.py | 298-370

模式生成过程

函数签名分析

向 Devin 询问 openai/openai-agents-python

深入研究



函数签名分析过程

资料来源： `src/agents/function_schema.py` | 230-262 `src/agents/function_schema.py` | 265-337

文档字符串解析

系统使用该 `griffe` 库来解析多种格式的文档字符串：

风格	检测模式	例子
谷歌	<code>Args:</code> , <code>Returns:</code>	标准谷歌风格
狮身人面像	<code>:param name:</code> , <code>:return:</code>	Sphinx 自动文档风格
Numpy	<code>Parameters</code> \n-----	NumPy 文档样式

解析摘录：

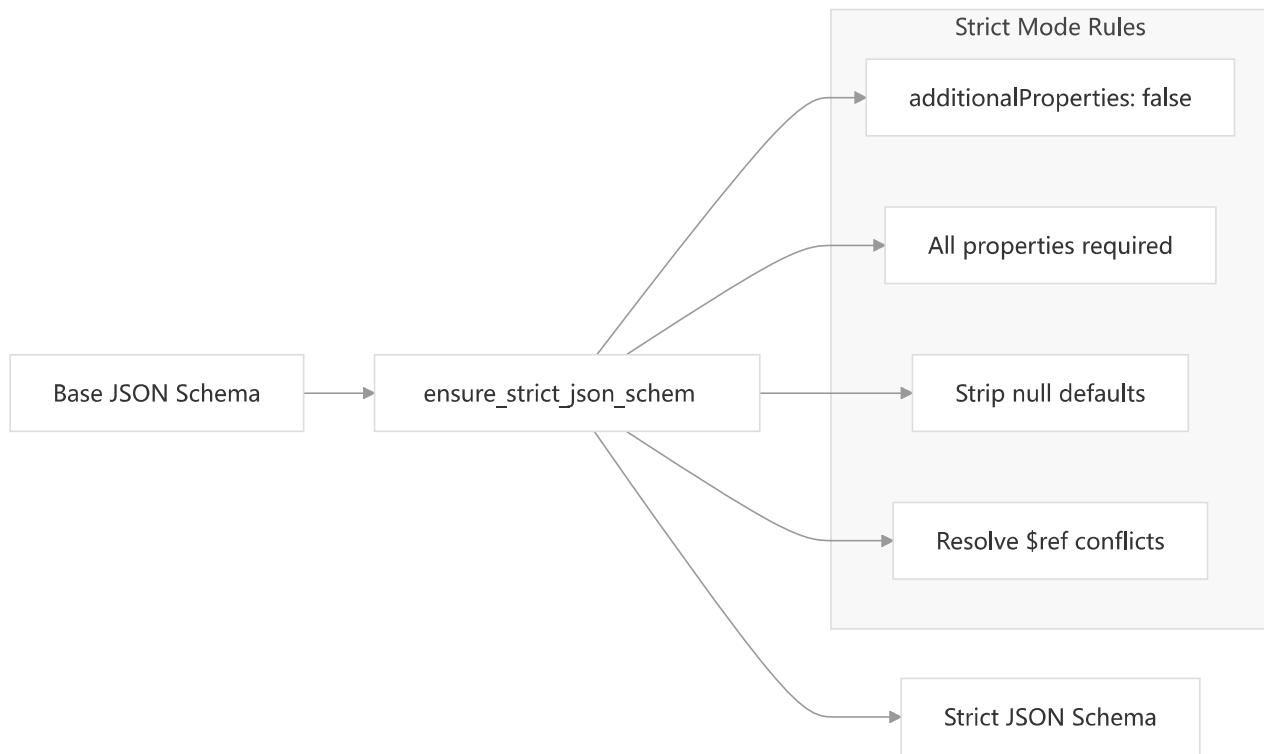
- 来自主文档字符串的函数描述
- 来自 Args/Parameters 部分的参数描述
- 类型信息（尽管类型提示优先）

资料来源： `src/agents/function_schema.py` | 89-132 `src/agents/function_schema.py` | 146-186

严格模式架构执行

向 Devin 询问 `openai/openai-agents-python`

深入研究



严格模式模式处理

资料来源： src/agents/strict_schema.py | 18-27 src/agents/strict_schema.py | 51-75

托管工具

托管工具在 LLM 提供商的基础架构上运行并提供预构建的功能：

托管工具类型

工具	班级	目的	提供者
文件搜索	FileSearchTool	矢量商店搜索	OpenAI
网页搜索	WebSearchTool	网络搜索	OpenAI
计算机使用	ComputerTool	计算机控制	OpenAI
代码解释器	CodeInterpreterTool	代码执行	OpenAI
图像生成	ImageGenerationTool	图像创建	OpenAI
MCP 集成	HostedMCPTool	MCP 服务器访问	OpenAI

向 Devin 询问 openai/openai-agents-python

深入研究

工具配置示例

```
# File search with vector store
file_search = FileSearchTool(
    vector_store_ids=["vs_123"],
    max_num_results=5,
    include_search_results=True
)

# Web search with location
web_search = WebSearchTool(
    user_location={"country": "US", "state": "CA"},
    search_context_size="high"
)

# Computer control
computer = ComputerTool(
    computer=MyComputerImpl(),
    on_safety_check=handle_safety_check
)
```

资料来源： docs/tools.md | 21-38

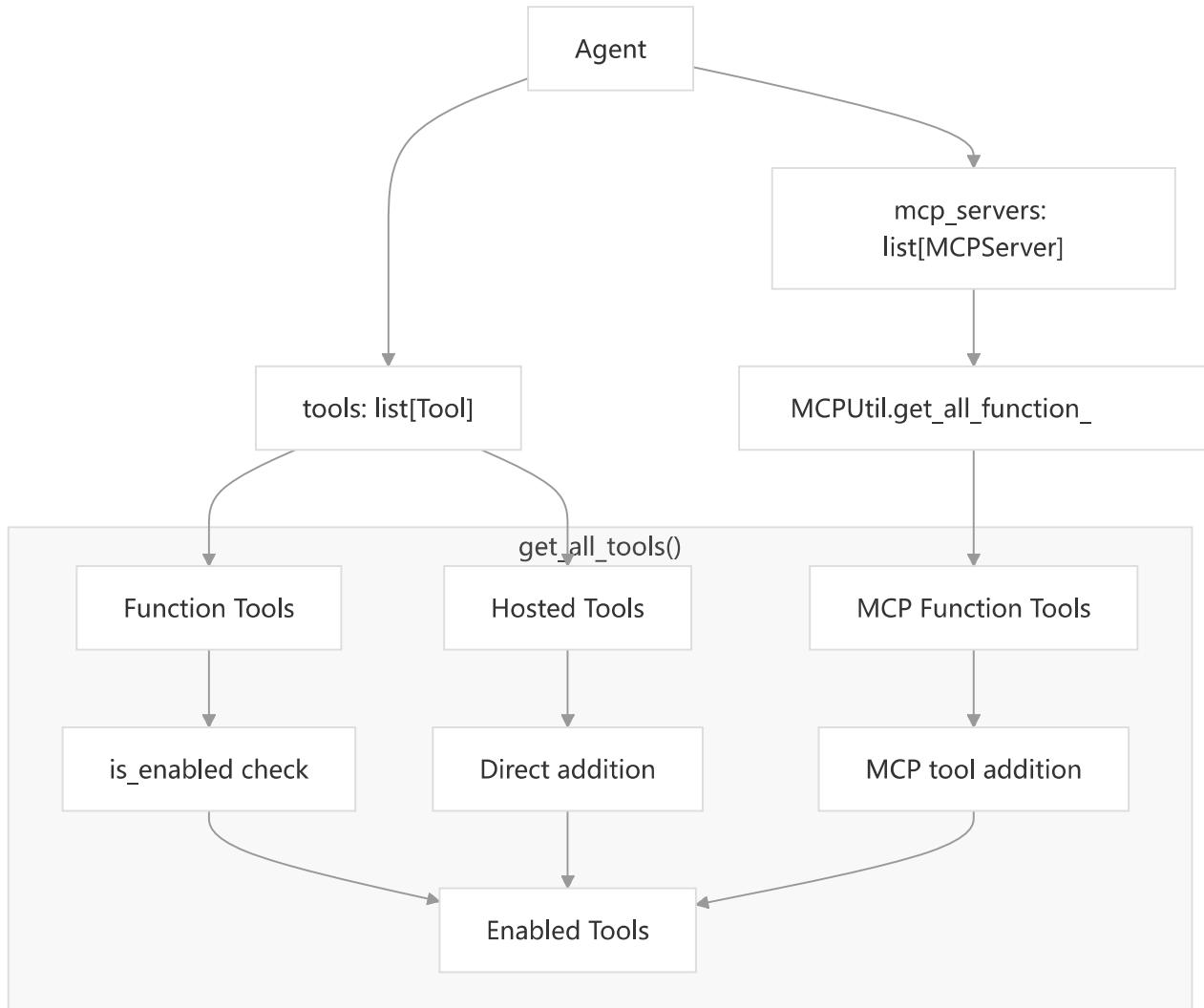
工具与代理集成

代理工具配置

工具通过参数和 MCP 服务器与代理集成 `tools`：

向 Devin 询问 openai/openai-agents-python

深入研究



代理工具集成

资料来源： src/agents/agent.py | 97-123

工具启用控制

可以使用以下 `is_enabled` 参数动态启用或禁用工具：

```

# Static enablement
@function_tool(is_enabled=False)
def disabled_tool():
    pass

# Dynamic enablement
  
```

向 Devin 询问 openai/openai-agents-python

深入研究

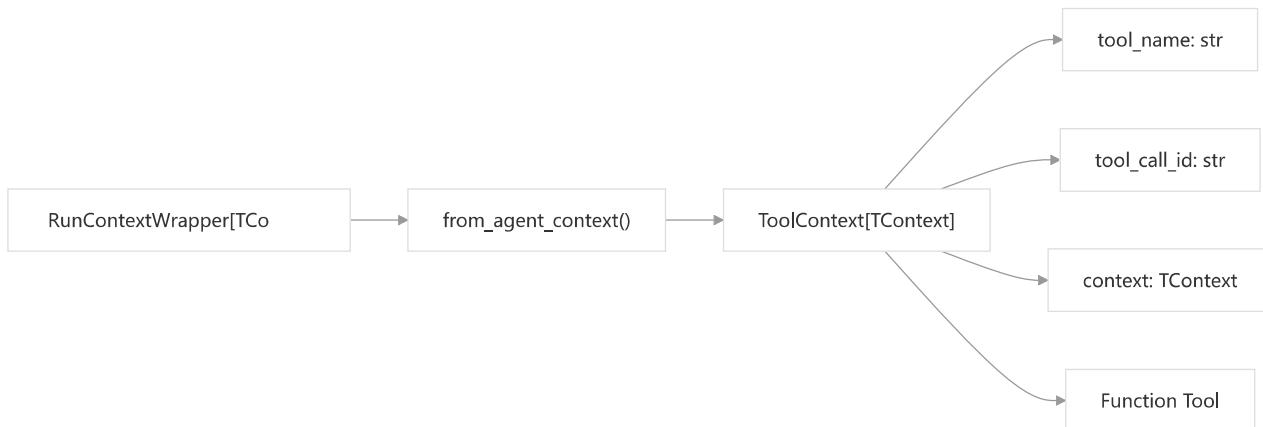
启用检查支持：

- 静态控制的布尔值
- 基于上下文和代理的动态控制可调用函数
- 用于异步启用逻辑的异步函数

资料来源： src/agents/tool.py | 91-95 测试/test_function_tool.py | 291-325

工具执行上下文

函数工具通过以下方式接收执行上下文 `ToolContext`：



工具上下文创建

扩展 `ToolContext` 了 `RunContextWrapper` 特定于工具的信息：

- `tool_name`：被调用的工具的名称
- `tool_call_id`：工具调用的唯一标识符
- `context`：特定于应用程序的上下文对象

资料来源： src/agents/tool_context.py | 17-43

错误处理

工具错误管理

功能工具系统通过以下方式提供可配置的错误处理 `failure_error_function`：

向 Devin 询问 openai/openai-agents-python

深入研究

配置**行为**

自定义函数**调用提供上下文和错误的函数**

None**重新引发应用程序处理的异常**

向 Devin 询问 openai/openai-agents-python

深入研究



> 菜单

跑步者和执行

相关源文件

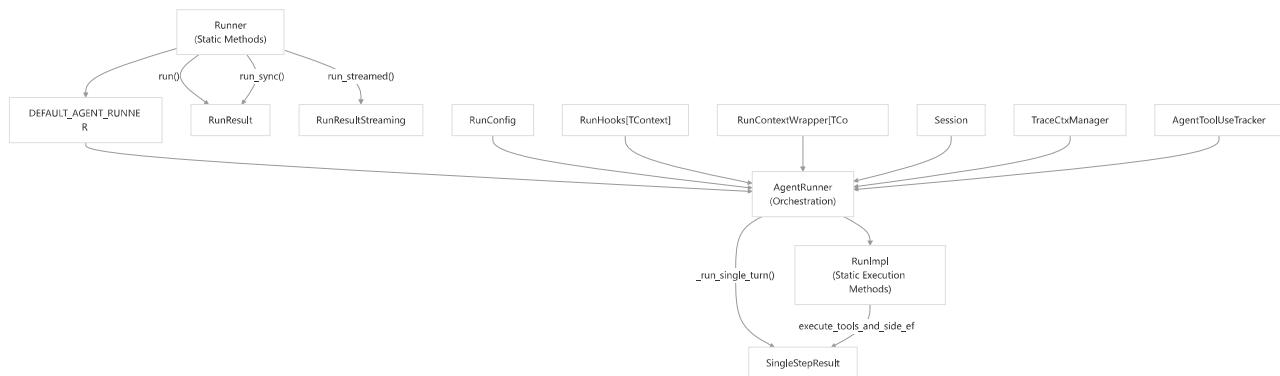
本页介绍了 OpenAI Agents SDK 中支持代理工作流的核心执行系统。它记录了该类如何协调代理执行、处理执行生命周期的 `Runner` 内部类 `AgentRunner` 和类，以及可用于控制代理运行的各种配置选项。

RunImpl

有关代理配置和生命周期挂钩的信息，请参阅[代理](#)。有关工具执行机制的详细信息，请参阅[工具和功能模式](#)。有关流事件处理的信息，请参阅[流和实时事件](#)。有关代理之间的切换机制，请参阅[切换和多代理工作流](#)。

概述

执行系统围绕三个主要类构建，它们协同运行代理工作流：



Runner 执行架构

该类 `Runner` 提供委托给全局 `DEFAULT_AGENT_RUNNER` 实例的静态方法。`AgentRunner` 负责协调执行循环，而 `RunImpl` 包含用于处理模型响应和执行工具的静态方法。

资料来源： src/agents/run.py | 165-320 | src/agents/_run_impl.py | 229-245

向 Devin 询问 openai/openai-agents-python

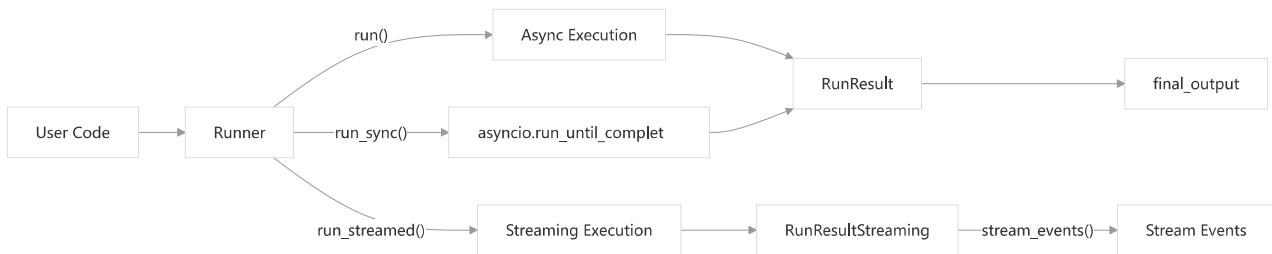
深入研究



→

该类 `Runner` 提供了用于执行代理工作流的主要公共 API。它提供三种执行模式：

方法	描述	返回	用例
<code>run()</code>	异步执行	<code>RunResult</code>	标准异步工作流
<code>run_sync()</code>	同步执行	<code>RunResult</code>	简单的脚本、笔记本
<code>run_streamed()</code>	流式执行	<code>RunResultStreaming</code>	实时用户界面、进度跟踪



Runner 执行模式

该类 `Runner` 将所有实际执行委托给 `AgentRunner` 存储在中的单例实例 `DEFAULT_AGENT_RUNNER`。

资料来源： src/agents/run.py 165-320 src/agents/run.py 62-82

AgentRunner 类

该类 `AgentRunner` 处理代理执行的编排，包括主执行循环、回合管理以及不同执行阶段之间的协调。

执行循环

核心执行逻辑遵循以下模式：

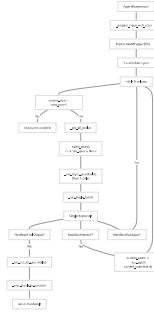
向 Devin 询问 openai/openai-agents-python

深入研究

向 Devin 询问 openai/openai-agents-python

深入研究





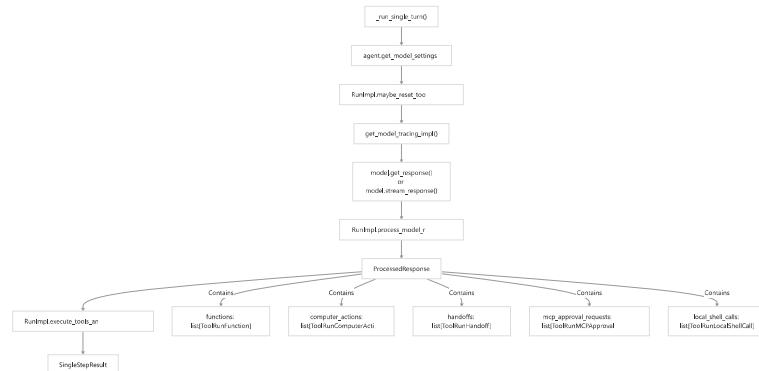
AgentRunner执行流程

执行循环通过局部变量维护状态，并处理返回的不同步骤类型 `_run_single_turn()`。跟踪跨度按代理进行管理，会话处理对话持久性。

资料来源： src/agents/run.py | 372-499 src/agents/_run_impl.py | 178-216

单回合执行

每次轮流调用模型并处理其响应：



单转加工

向 Devin 询问 openai/openai-agents-python

深入研究

资料来源： src/agents/run.py | 856-934 src/agents/_run_impl.py | 384-535

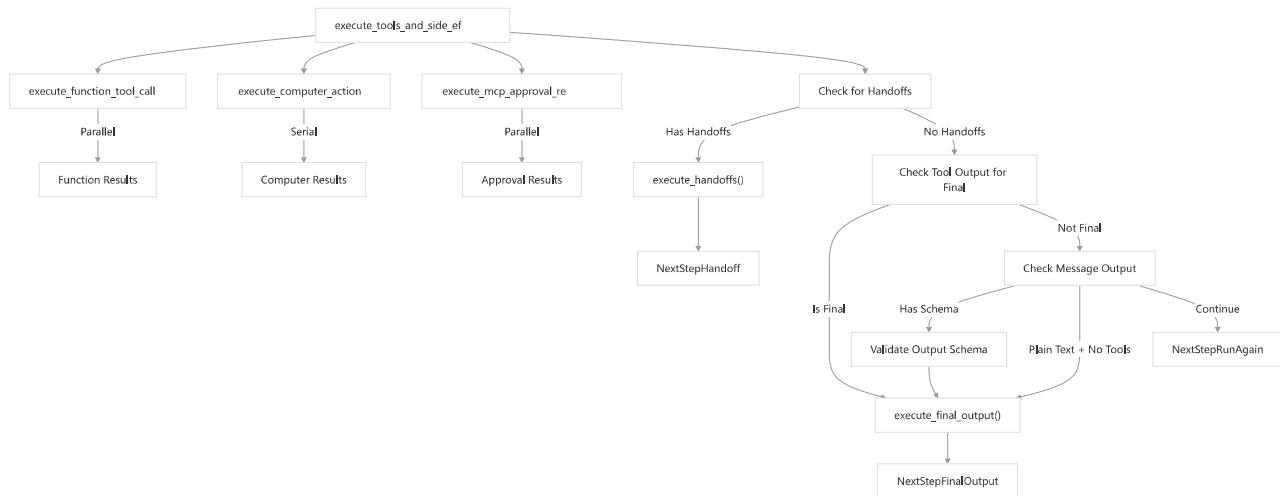
src/agents/_run_impl.py | 372-382

RunImpl 类

该类 RunImpl 包含处理模型响应和执行工具和副作用的核心执行逻辑。

工具和副作用执行

主执行方法协调多种类型的操作：



RunImpl 执行逻辑

资料来源： src/agents/_run_impl.py | 231-369

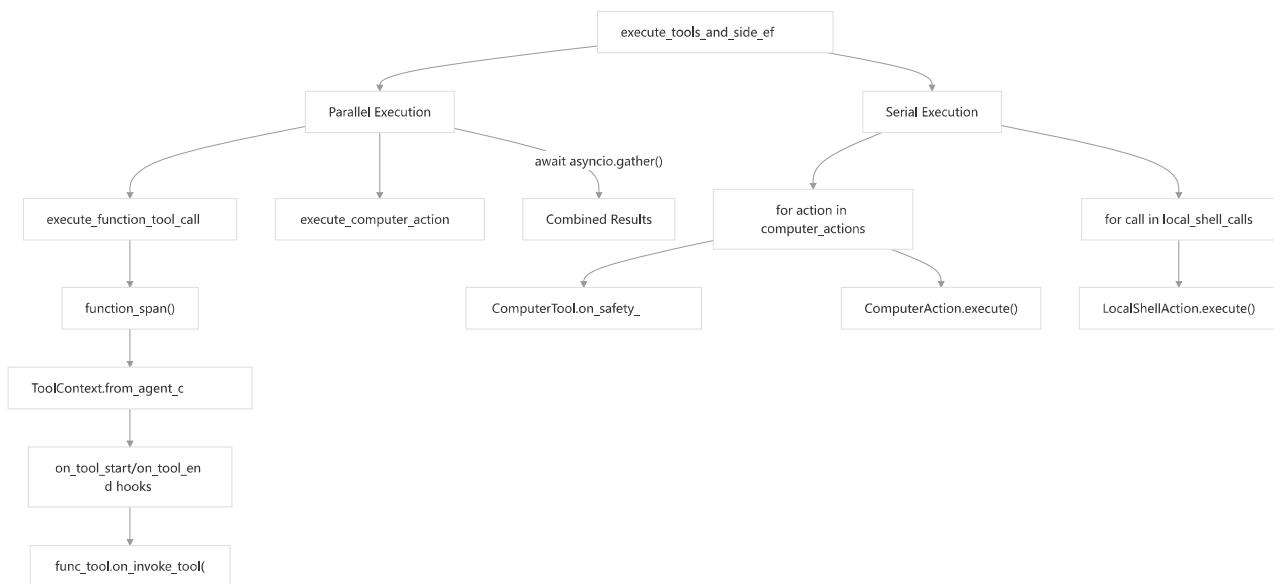
工具执行管道

不同类型的工具采用不同的策略执行：

工具类型	执行方法	并发	原因
功能工具	execute_function_tool_calls()	平行线（asyncio.gather()）	独立运营
计算机操作	execute_computer_actions()	串行（顺序循环）	依赖于状态的操作
本地 Shell	execute_local_shell_calls()	串行（顺序循环）	Shell 状态依赖关系

向 Devin 询问 openai/openai-agents-python

深入研究



工具执行策略

工具执行根据操作需求采用不同的模式。功能工具独立并行运行，而计算机和 Shell 操作则按顺序保持状态。

资料来源： src/agents/_run_impl.py | 537-683 src/agents/_run_impl.py | 612-684

配置选项

运行配置

数据 `RunConfig` 类为代理运行提供了全面的配置：



RunConfig 结构

该配置允许对模型、护栏和跟踪进行全局覆盖。`model_provider` 默认设置 `MultiProvider()` 同时支持 OpenAI 和 LiteLLM 模型。

资料来源： src/agents/run.py | 84-141

向 Devin 询问 openai/openai-agents-python

深入研究

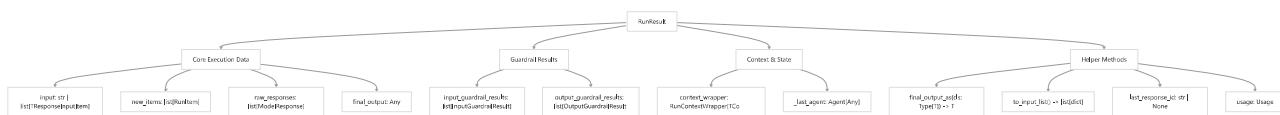
选项	类型	描述
context	TContext	代理的运行时上下文
max_turns	int	最大执行回合数（默认值：10）
hooks	RunHooks[TContext]	生命周期事件回调
run_config	RunConfig	全局运行配置
previous_response_id	str	为了保持 OpenAI Responses API 的连续性
session	Session	对话历史管理

资料来源： src/agents/run.py | 143-163

执行结果

运行结果

标准执行结果包含：



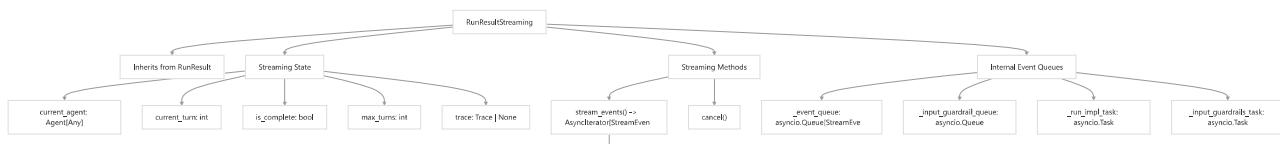
RunResult 结构

提供 RunResult 对执行输出的结构化访问，并包含用于类型转换和对话延续的辅助方法。该 `_last_agent` 字段跟踪哪个代理生成了最终输出。

资料来源： src/agents/result.py | 104-115 src/agents/result.py | 38-103

运行结果流

对于流式执行，结果对象提供了事件流功能：



向 Devin 询问 openai/openai-agents-python

深入研究

流式结果管理多个异步任务和队列，以实时传递事件。该 `_start_streaming()` 方法运行代理循环，同时填充事件队列以供消费者访问。

资料来源： src/agents/result.py | 117-277 src/agents/run.py | 640-708

错误处理

执行系统提供了全面的错误处理和详细的上下文：

向 Devin 询问 openai/openai-agents-python

深入研究



> 菜单

模型和提供者

[相关源文件](#)

本页介绍模型接口及其实现，它们构成了 OpenAI Agents SDK 中语言模型交互的核心抽象层。模型负责处理与各种语言模型提供商的通信、响应格式转换以及流式传输功能。

模型配置参数请参见[模型设置与配置](#)。协调模型调用与工具执行的执行层请参见[Runner 与执行](#)。

模型接口架构

SDK 提供了一个统一的 `Model` 接口，抽象了不同的语言模型提供程序和 API。这使得代理无需更改代理代码即可与任何模型实现配合使用。

资料来源： `src/agents/models/openai_chatcompletions.py` | 37-44

`src/agents/models/openai_responses.py` | 53-64

`src/agents/extensions/models/litellm_model.py` | 56-70

基 `Model` 类定义了所有模型实现必须遵循的接口：

```
async def get_response(  
    self,  
    system_instructions: str | None,  
    input: str | list[TResponseInputItem],  
    model_settings: ModelSettings,  
    tools: list[Tool],  
    output_schema: AgentOutputSchemaBase | None,  
    handoffs: list[Handoff],  
    tracing: ModelTracing,  
    previous_response_id: str | None,  
    prompt: ResponsePromptParam | None = None
```

向 Devin 询问 openai/openai-agents-python

深入研究



```
# Same parameters as get_response
) -> AsyncIterator[TResponseStreamEvent | ResponseStreamEvent]
```

模型追踪

该 `ModelTracing` 参数控制模型调用的可观察性和数据收集。所有模型实现都接受此参数并使用它来确定跟踪行为：

- `tracing.is_disabled()` - 返回跟踪是否完全禁用
- `tracing.include_data()` - 返回是否在跟踪中包含请求/响应数据

常见值包括 `ModelTracing.DISABLED` 完全关闭跟踪、`ModelTracing.ENABLED` 完全跟踪以及 `ModelTracing.ENABLED_WITHOUT_DATA` 不包括实际请求/响应数据的跟踪。

资料来源： src/agents/models/openai_chatcompletions.py | 61-65

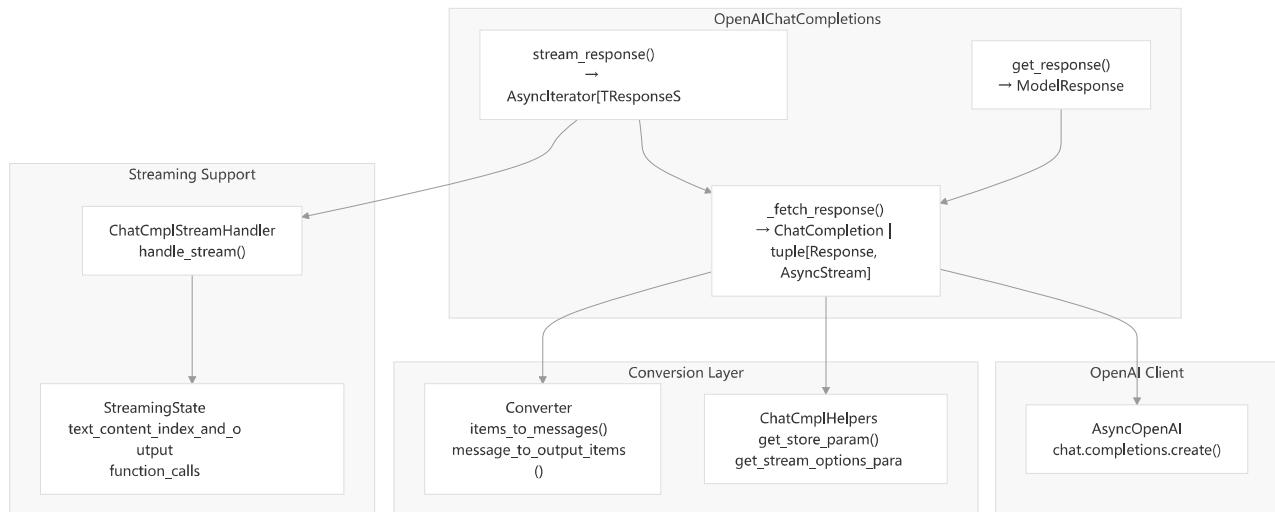
src/agents/models/openai_responses.py | 81-82

src/agents/extensions/models/litellm_model.py | 84-89

OpenAI 模型实现

OpenAIChatCompletions 模型

它 `OpenAIChatCompletionsModel` 使用 OpenAI 的 Chat Completions API 并支持函数调用、结构化输出和流式响应。



向 Devin 询问 openai/openai-agents-python

深入研究

OpenAIChatCompletionsModel 使用该类在 SDK 内部项目格式和 OpenAI 消息格式之间进行转换。

Converter 主要转换方法包括：

- `Converter.items_to_messages()` - 将 SDK 项目转换为 OpenAI ChatCompletionMessageParam 格式
- `Converter.message_to_output_items()` - 将 OpenAI ChatCompletionMessage 转换为 SDK ResponseOutputItem 格式
- `Converter.tool_to_openai()` - 将 SDK 工具转换为 OpenAI 工具格式
- `Converter.convert_tool_choice()` - 处理 tool_choice 参数转换
- `Converter.convert_response_format()` - 将输出模式转换为响应格式

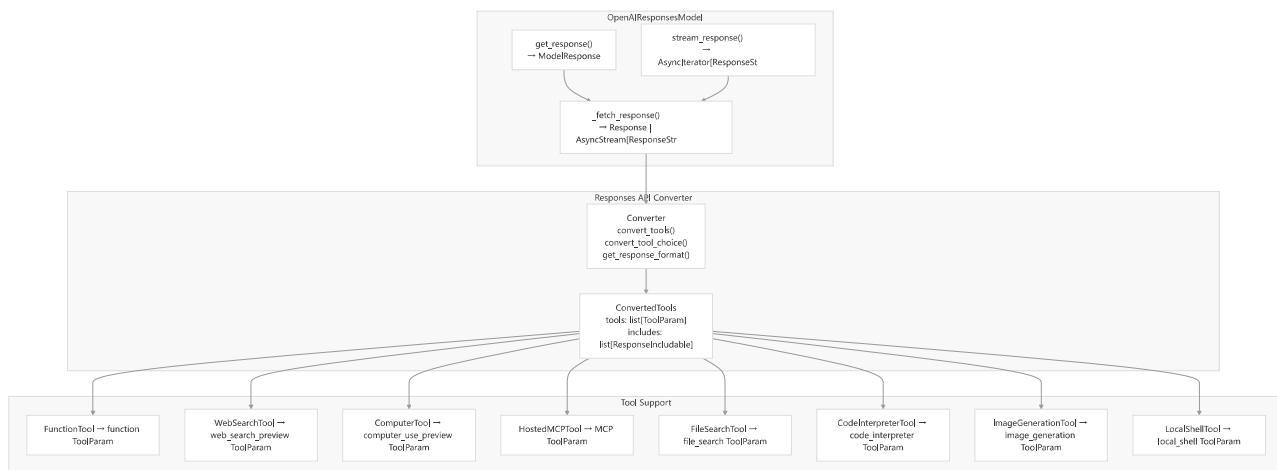
该模型处理工具调用、系统指令、通过的结构化输出 `response_format`，以及来自模型（如）的各种内容类型，包括文本、拒绝、函数调用和推理内容 `deepseek-reasoner`。

资料来源： src/agents/models/openai_chatcompletions.py | 228-295

src/代理/模型/chatcpl_converter.py | 274-464

OpenAIResponses模型

使用 **OpenAIResponsesModel** OpenAI 较新的 Responses API，它为多模式响应和高级工具集成提供了增强的功能。



资料来源： src/agents/models/openai_responses.py | 53-296

src/agents/models/openai_responses.py | 298-463

向 Devin 询问 openai/openai-agents-python

深入研究

- 本机支持 `ComputerTool`、`WebSearchTool`、`FileSearchTool`、`HostedMCPTool`、`CodeInterpreterTool`、`ImageGenerationTool` 和 `LocalShellTool`
- 响应包括其他数据，例如 `file_search_call.results`
- 不同的参数映射（例如，`max_output_tokens` 而不是 `max_tokens`）
- 支持 `MCPToolChoice` MCP 服务器工具选择
- 通过响应格式配置增强结构化输出支持

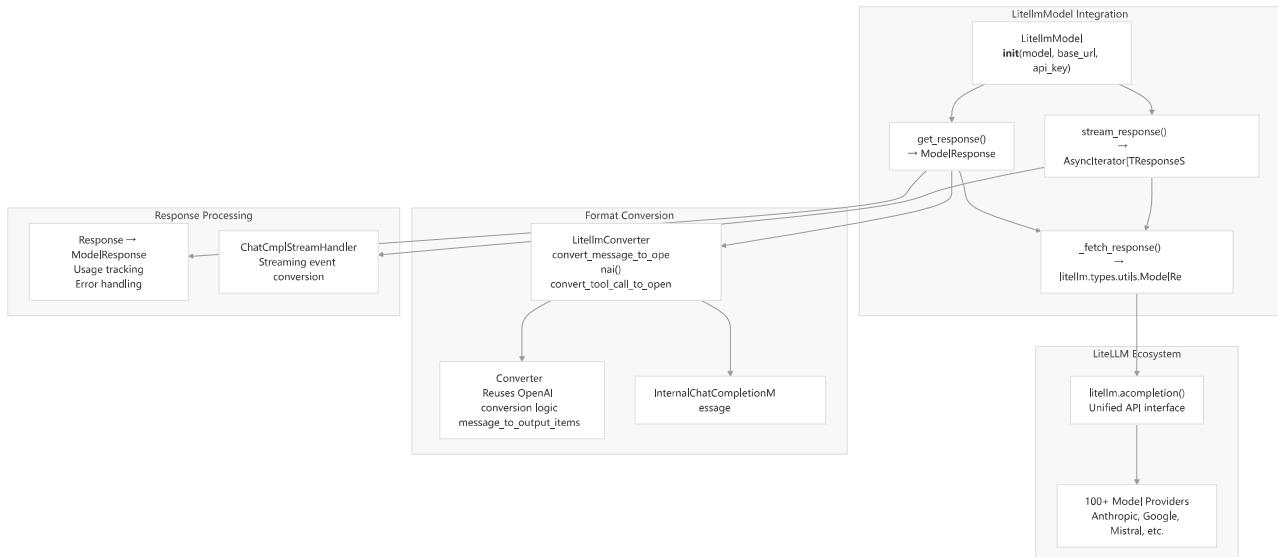
资料来源： `src/agents/models/openai_responses.py` | 267-290

`src/agents/models/openai_responses.py` | 394-452

第三方模型集成

轻型模型

通过 LiteLLM 库可以 `LitellmModel` 访问 100 多个语言模型提供商，从而可以使用来自 Anthropic、Google、Mistral 和其他提供商的模型。



资料来源： `src/agents/extensions/models/litellm_model.py` | 56-354

`src/agents/extensions/models/litellm_model.py` | 356-423

通过几种关键机制来处理提供商 `LitellmModel` 差异：

- **消息转换：** `LitellmConverter.convert_message_to_openai()` 将 LiteLLM 消息格式转换为与 OpenAI 向 Devin 询问 openai/openai-agents-python

深入研究

- **推理内容 InternalChatCompletionMessage**：通过 reasoning_content 字段对推理模型进行特殊处理

src/agents/extensions/models/litellm_model.py | 48-54

- **提供商特定字段**：提取提供商特定字段，如拒绝和注释

src/agents/extensions/models/litellm_model.py | 370-387

该模型支持所有标准 ModelSettings 参数并 extra_args 直接传递给底层 LiteLLM 调用，从而实现特定于提供商的功能，同时保持一致的接口。

资料来源： src/agents/extensions/models/litellm_model.py | 305-327

测试/模型/test_kwarg_functionality.py | 16-60

流和响应处理

流处理架构

SDK 提供了复杂的流支持，可将特定于模型的流格式转换为统一的事件流。



资料来源： src/代理/模型/chatcmpl_stream_handler.py | 68-429

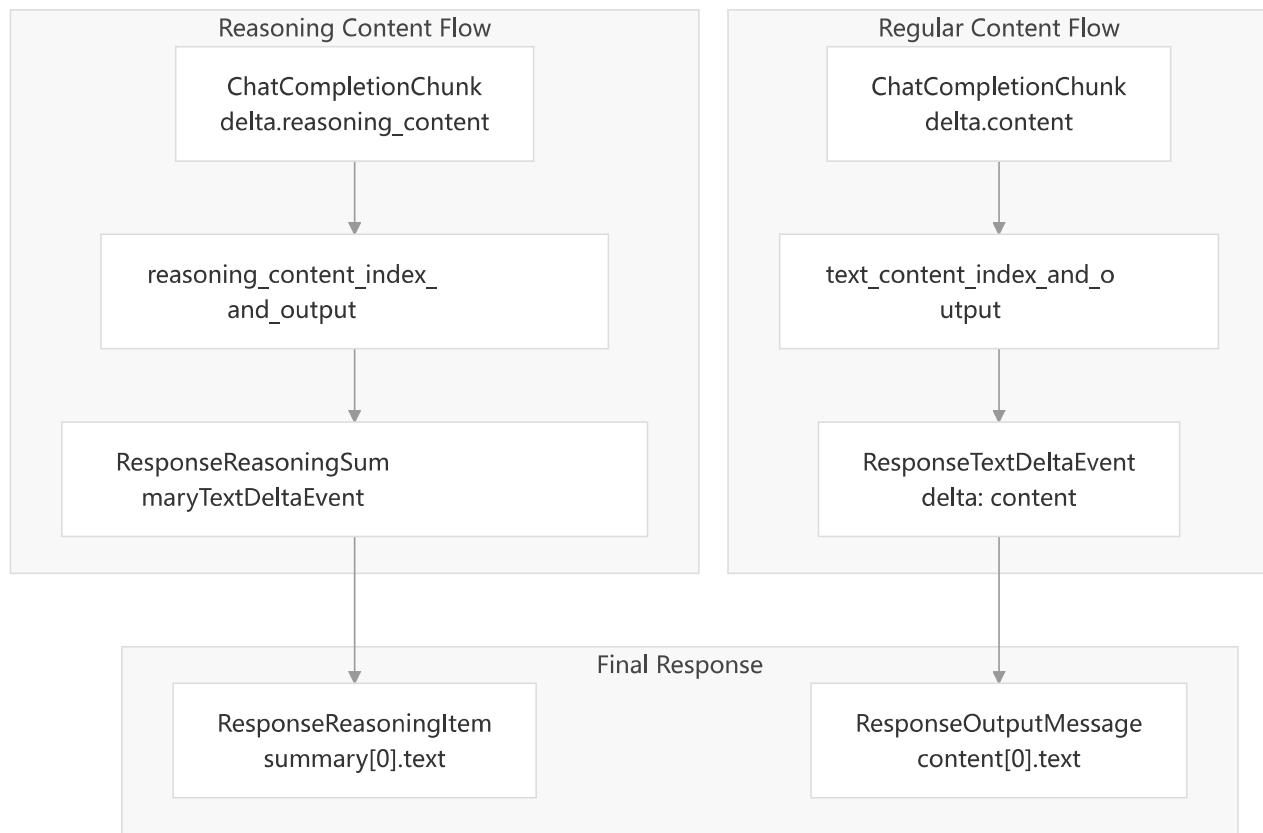
src/代理/模型/chatcmpl_stream_handler.py | 49-66

推理内容支持

最近的模型例如 deepseek-reasoner 在常规响应之外提供推理内容。流处理程序通过专门的事件类型来管理这些内容。

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源： src/代理/模型/chatcmpl_stream_handler.py | 95-142

测试/test_reasoning_content.py | 94-167

模型配置集成

模型接受一个 `ModelSettings` 包含特定于提供程序的配置的对象。每个模型实现会根据底层 API 的功能以不同的方式处理这些设置。

环境	OpenAI 聊天完成	OpenAI 回应	精简法学硕士
<code>temperature</code>	✓	✓	✓
<code>max_tokens</code>	✓	<code>max_output_tokens</code>	✓
<code>tool_choice</code>	✓	✓	✓
<code>parallel_tool_calls</code>	✓	✓	✓
<code>reasoning</code>	✓	✓	✓

向 Devin 询问 openai/openai-agents-python

深入研究



> 菜单

上下文管理

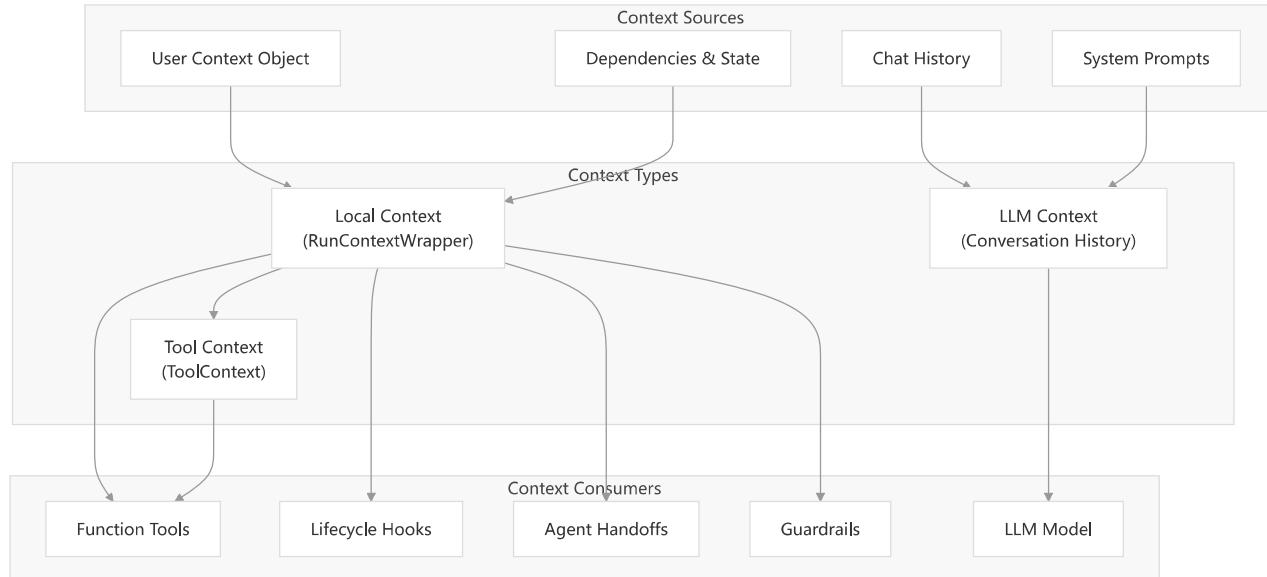
[相关源文件](#)

OpenAI Agents SDK 中的上下文管理提供了一种依赖注入机制，允许您在代理执行过程中传递数据、依赖项和状态。该系统区分本地上下文（代码可用）和 LLM 上下文（响应生成期间语言模型可见的数据）。

有关会话和对话历史管理的详细信息，请参阅[会话管理](#)。有关代理执行流程的详细信息，请参阅[运行和执行](#)。

上下文架构概述

SDK 实现了双上下文系统，将本地应用程序状态与 LLM 可见数据分离：



资料来源：src/agents/run_context.py | 22 src/agents/tool_context.py | 1-43

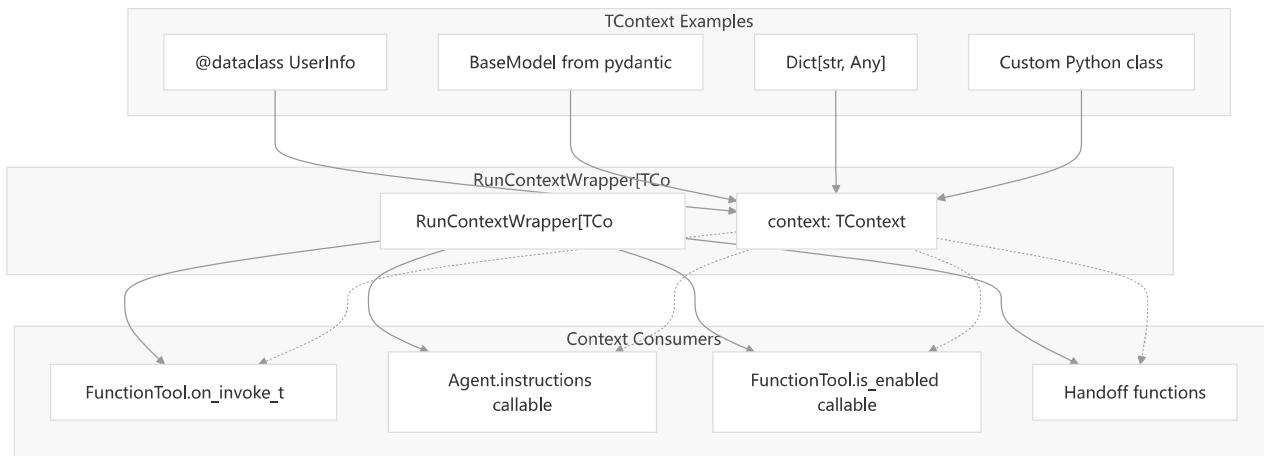
<docs/context.md> | 1-79

向 Devin 询问 [openai/openai-agents-python](#)

深入研究



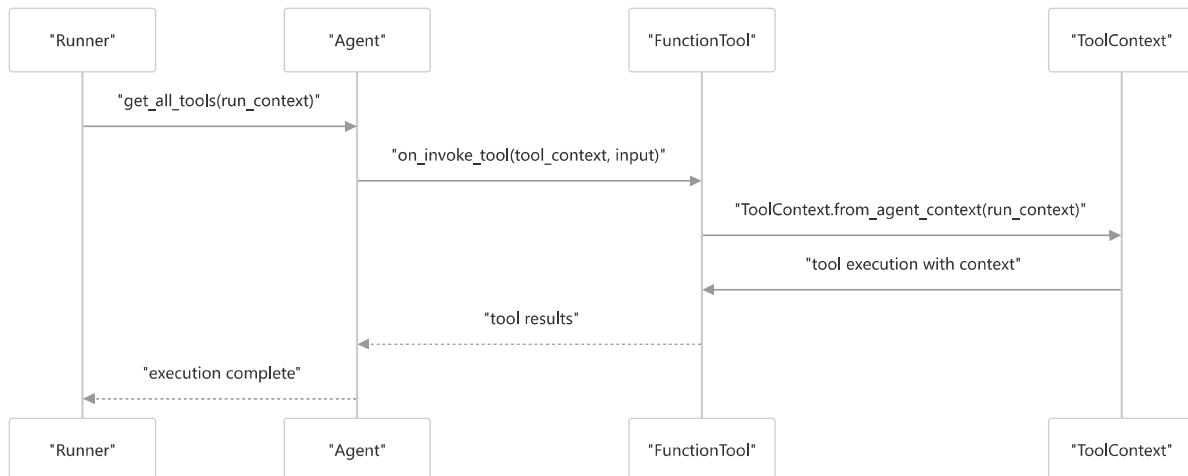
RunContextWrapper 结构



资料来源： src/agents/run_context.py | 22 | docs/context.md | 8-27

上下文流经执行

上下文包装器流经整个代理执行管道，确保对依赖项和状态的一致访问：



资料来源： src/agents/agent.py | 104-122 | src/agents/tool.py | 76-85

src/agents/tool_context.py | 27-42

工具上下文专业化

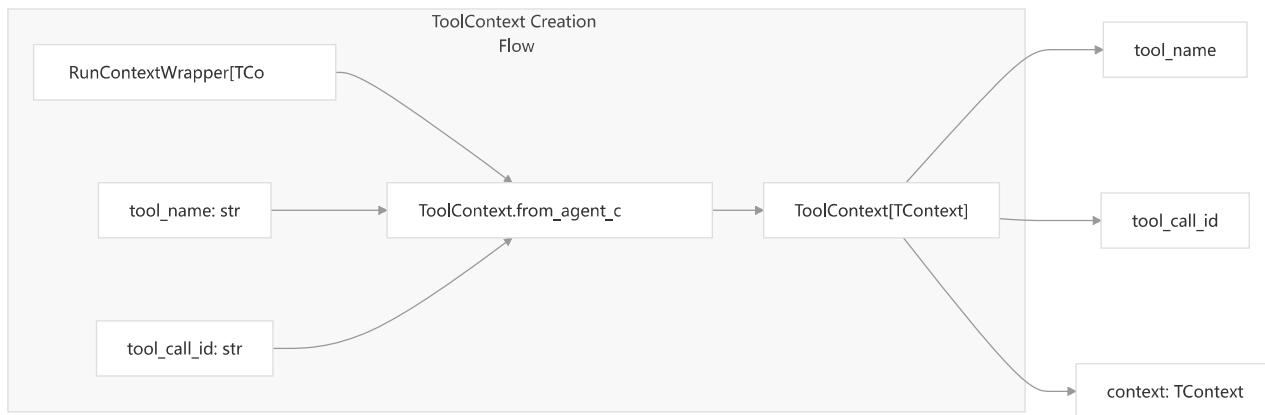
ToolContext 扩展 RunContextWrapper 以提供特定工具目的信息，同时保持对底层上下文的访问。

向 Devin 询问 openai/openai-agents-python

深入研究

场地	类型	描述
tool_name	str	被调用的工具的名称
tool_call_id	str	工具调用的唯一标识符

工具上下文创建

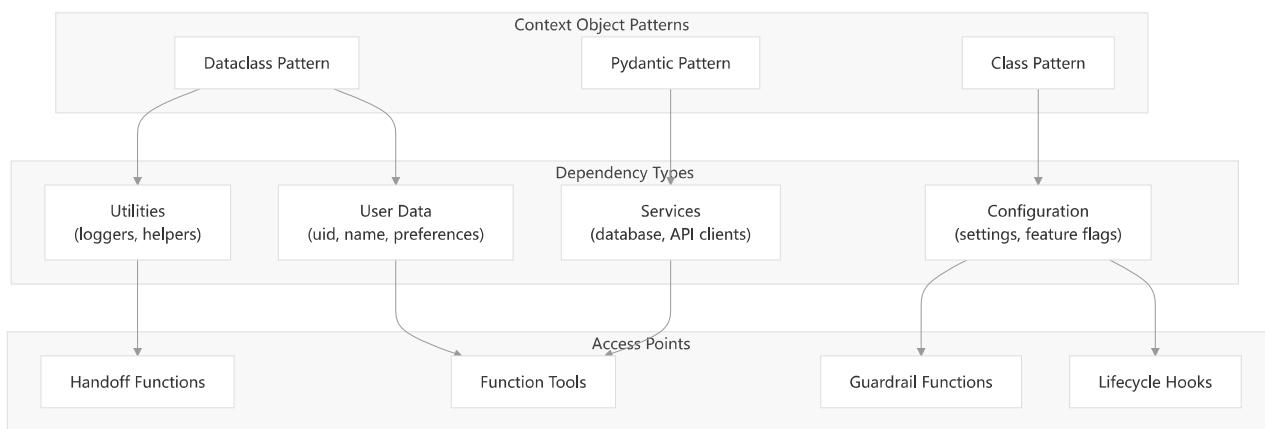


资料来源： src/agents/tool_context.py | 27-42 src/agents/_run_impl.py | 551-555

依赖注入模式

上下文系统通过允许您将依赖项打包到上下文对象中来实现依赖注入：

常见的上下文模式

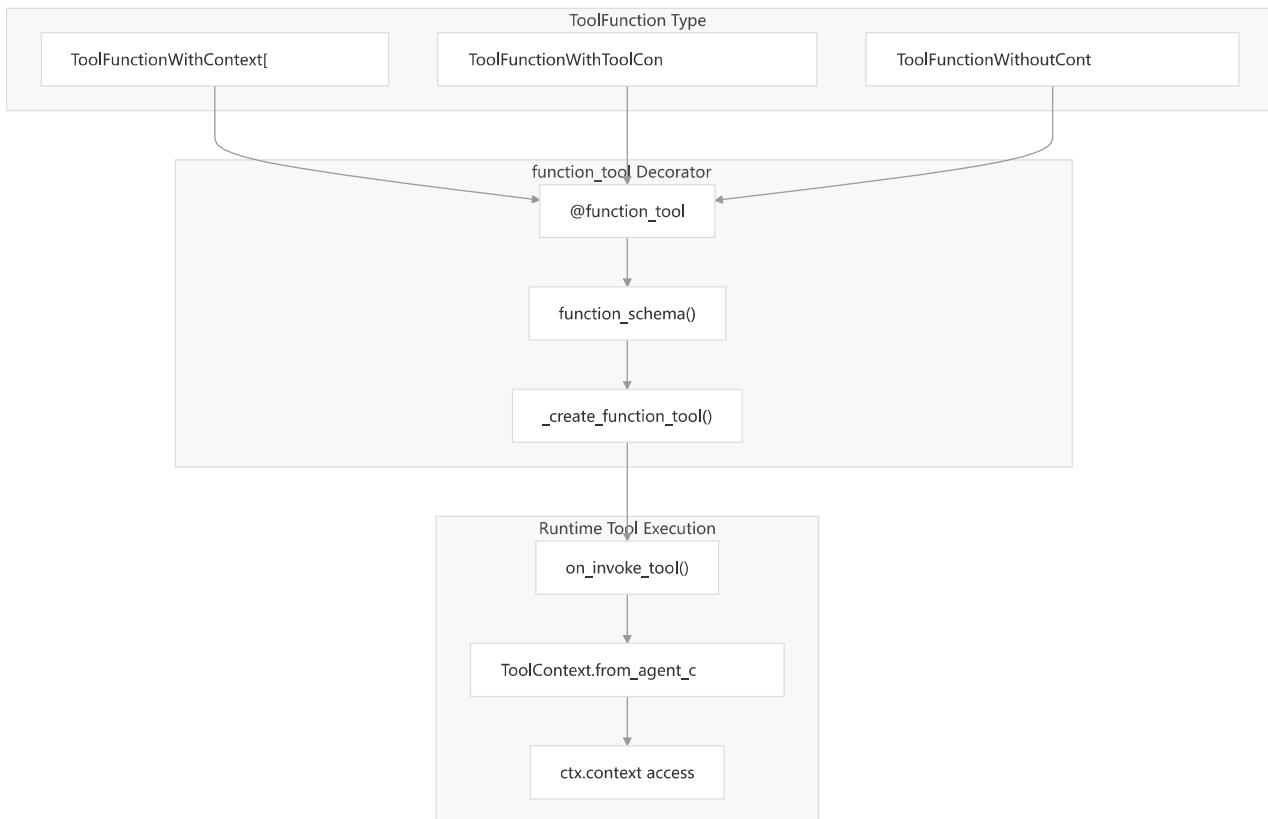


资料来源： docs/context.md | 28-70 测试/test_function_tool_decorator.py | 13-19

向 Devin 询问 openai/openai-agents-python

深入研究

功能工具上下文访问

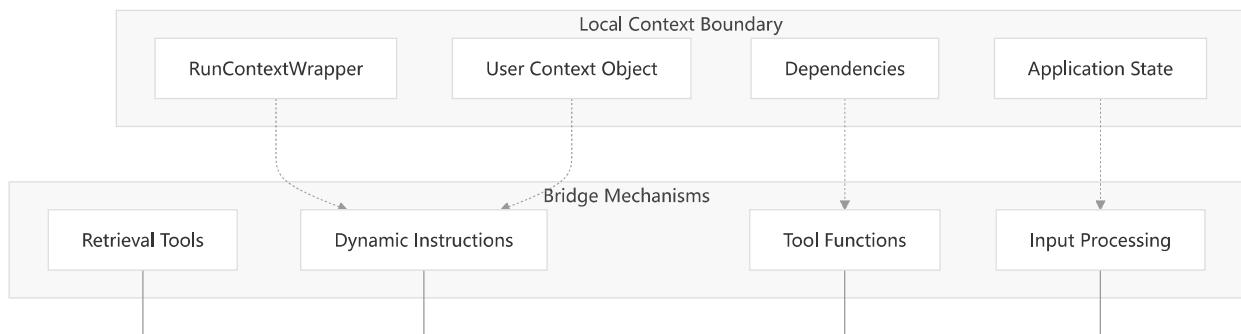


资料来源：
 src/agents/tool.py | 38-46 src/agents/tool.py | 382-430
 src/agents/tool_context.py | 27-42

上下文与 LLM 上下文分离

系统在本地上下文（LLM 不可见）和 LLM 上下文（对话历史）之间保持明确的分离：

上下文可见性模型



向 Devin 询问 openai/openai-agents-python

深入研究

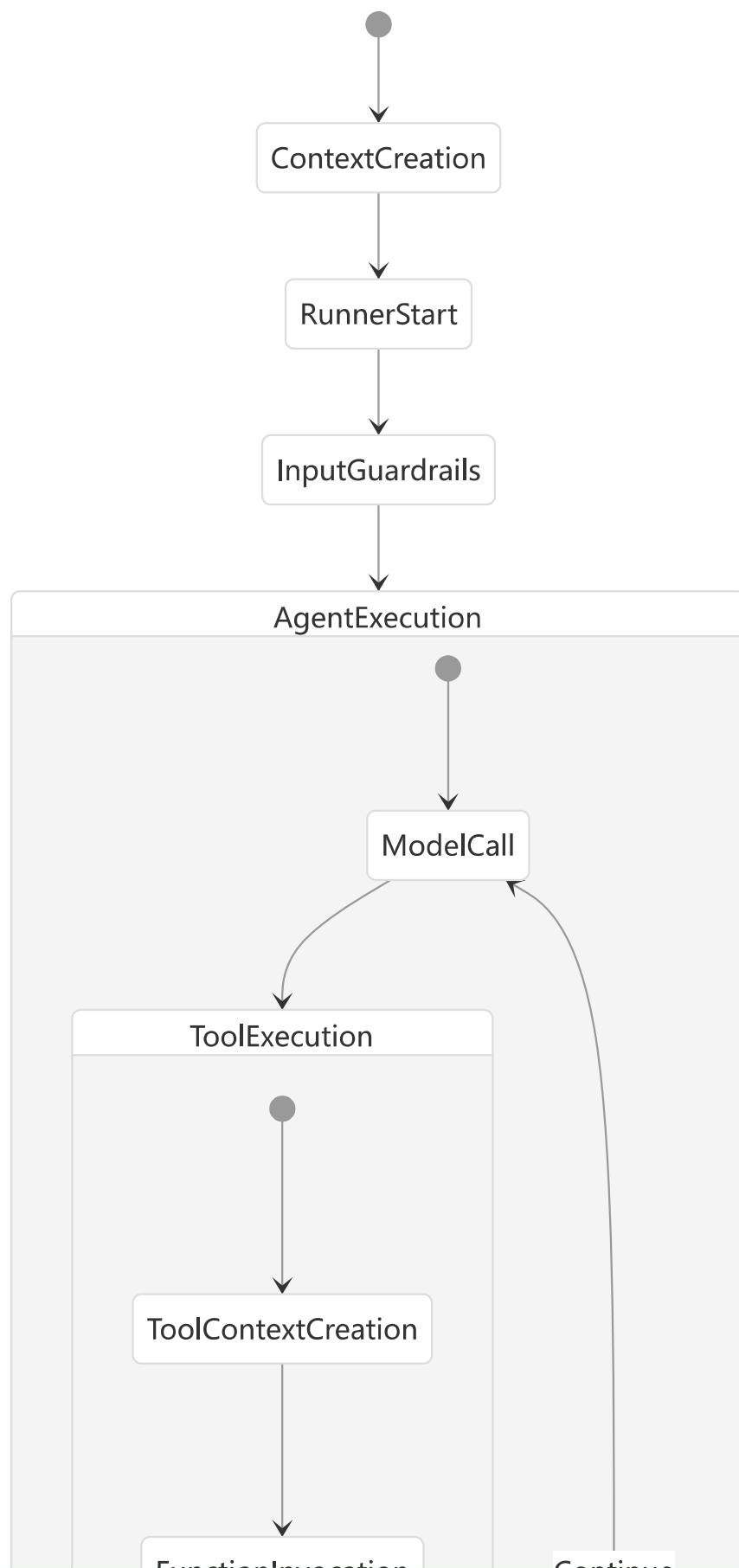
资料来源： docs/context.md | 71-79

代理执行中的上下文生命周期

上下文在整个代理执行过程中遵循特定的生命周期，从而维护所有组件的一致性：

向 Devin 询问 openai/openai-agents-python

深入研究



向 Devin 询问 openai/openai-agents-python

深入研究

> 菜单

护栏和安全

相关源文件

本文档涵盖 OpenAI Agents SDK 中的护栏系统，该系统为代理的输入和输出提供并行安全检查。护栏支持验证、内容过滤和安全机制，当安全条件不满足时，可以暂停代理的执行。

有关一般代理执行流程的信息，请参阅[运行器和执行](#)。有关跟踪护栏执行的信息，请参阅[跟踪和可观察性](#)。

概述

护栏系统提供两种与代理执行并行运行的安全检查：

- **输入护栏**: 在代理处理开始之前验证用户输入
- **输出护栏**: 在将结果返回给用户之前验证代理输出

护栏使用“绊线”机制——当护栏检测到违规行为时，它可以立即停止执行并引发异常，从而防止昂贵的模型调用或不安全的输出到达用户。

护栏类型和生命周期

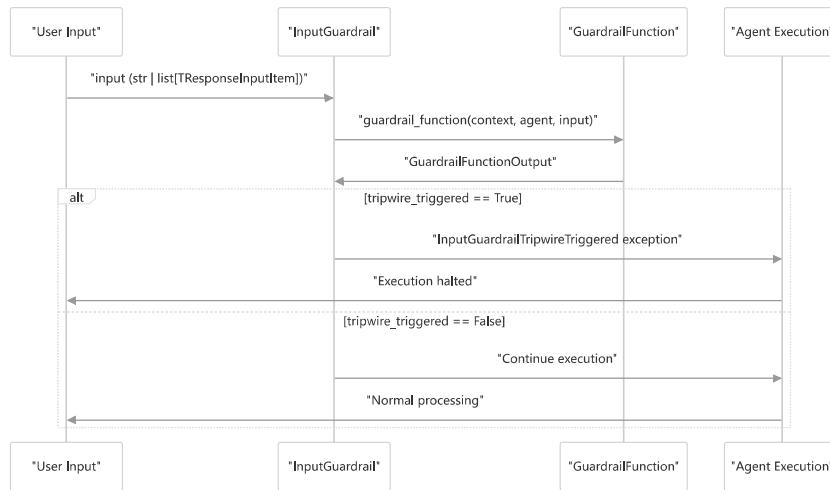
输入护栏

输入护栏分三个阶段执行：

向 Devin 询问 openai/openai-agents-python

深入研究



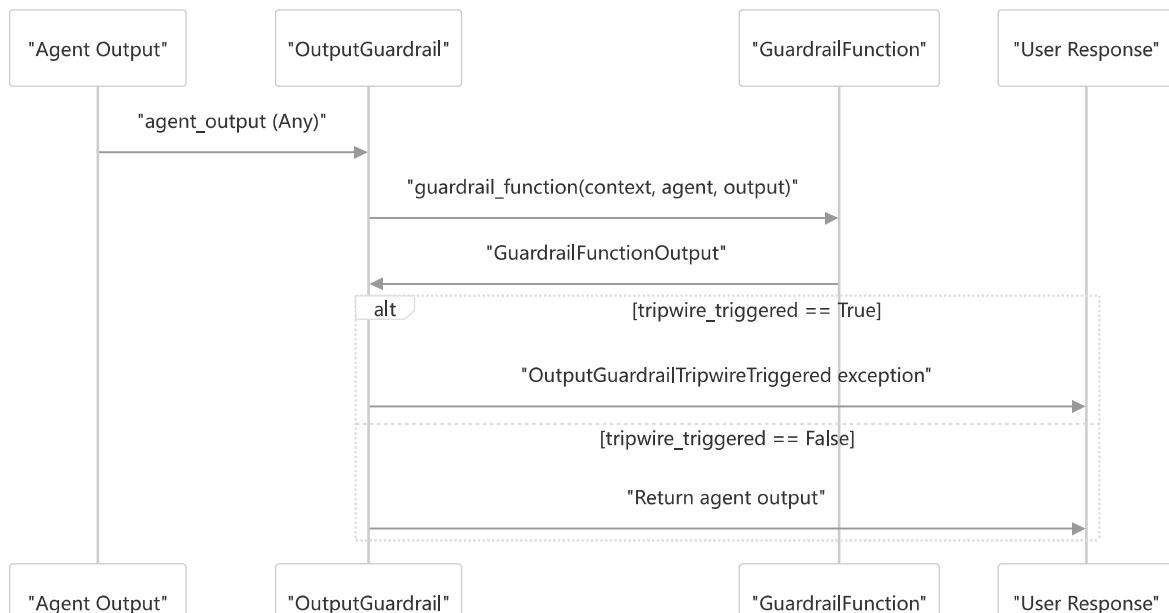


仅当代理是执行链中的第一个代理时，输入护栏才会运行，重点关注用户提供的输入验证。

资料来源： src/agents/guardrail.py | 72-126 | docs/guardrails.md | 10-21

输出护栏

输出护栏遵循类似的模式，但在代理完成后执行：



向 Devin 询问 openai/openai-agents-python

深入研究

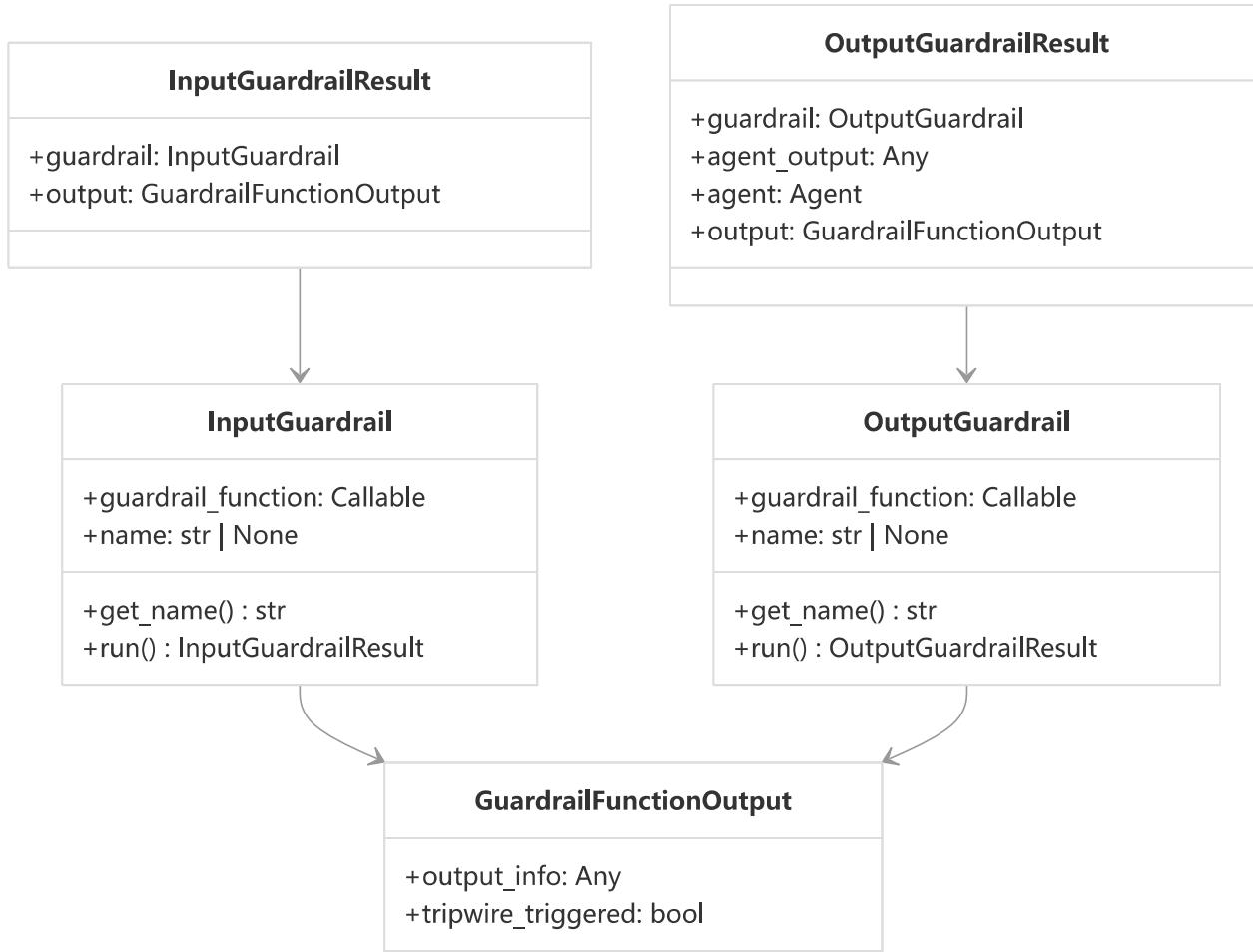
核心组件

护栏功能输出

所有护栏功能的返回类型：

场地	类型	目的
output_info	Any	关于护栏检查的可选详细信息
tripwire_triggered	bool	是否立即停止执行

护栏类



向 Devin 询问 openai/openai-agents-python

深入研究

实现模式

使用装饰器

SDK 提供了用于创建护栏的装饰器：

```
@input_guardrail
async def content_filter(
    ctx: RunContextWrapper[None],
    agent: Agent,
    input: str | list[TResponseInputItem]
) -> GuardrailFunctionOutput:
    # Validation logic here
    return GuardrailFunctionOutput(
        output_info={"check_type": "content_filter"},
        tripwire_triggered=contains_harmful_content
    )
```

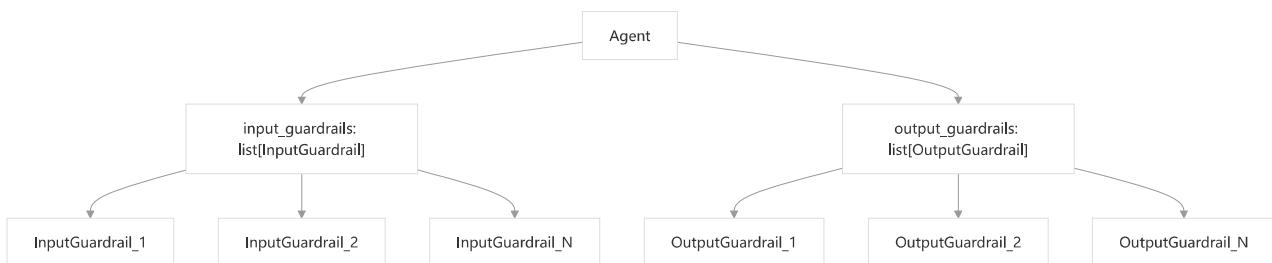
装饰器支持同步和异步函数以及可选命名：

```
@input_guardrail(name="custom_guardrail_name")
def sync_guardrail(...): ...
```

资料来源： src/agents/guardrail.py | 185-256 src/agents/guardrail.py | 258-329

代理集成

护栏配置为 `Agent` 实例上的属性：



这种共置模式确保护栏与特定代理及其安全要求紧密相关。

向 Devin 询问 openai/openai-agents-python

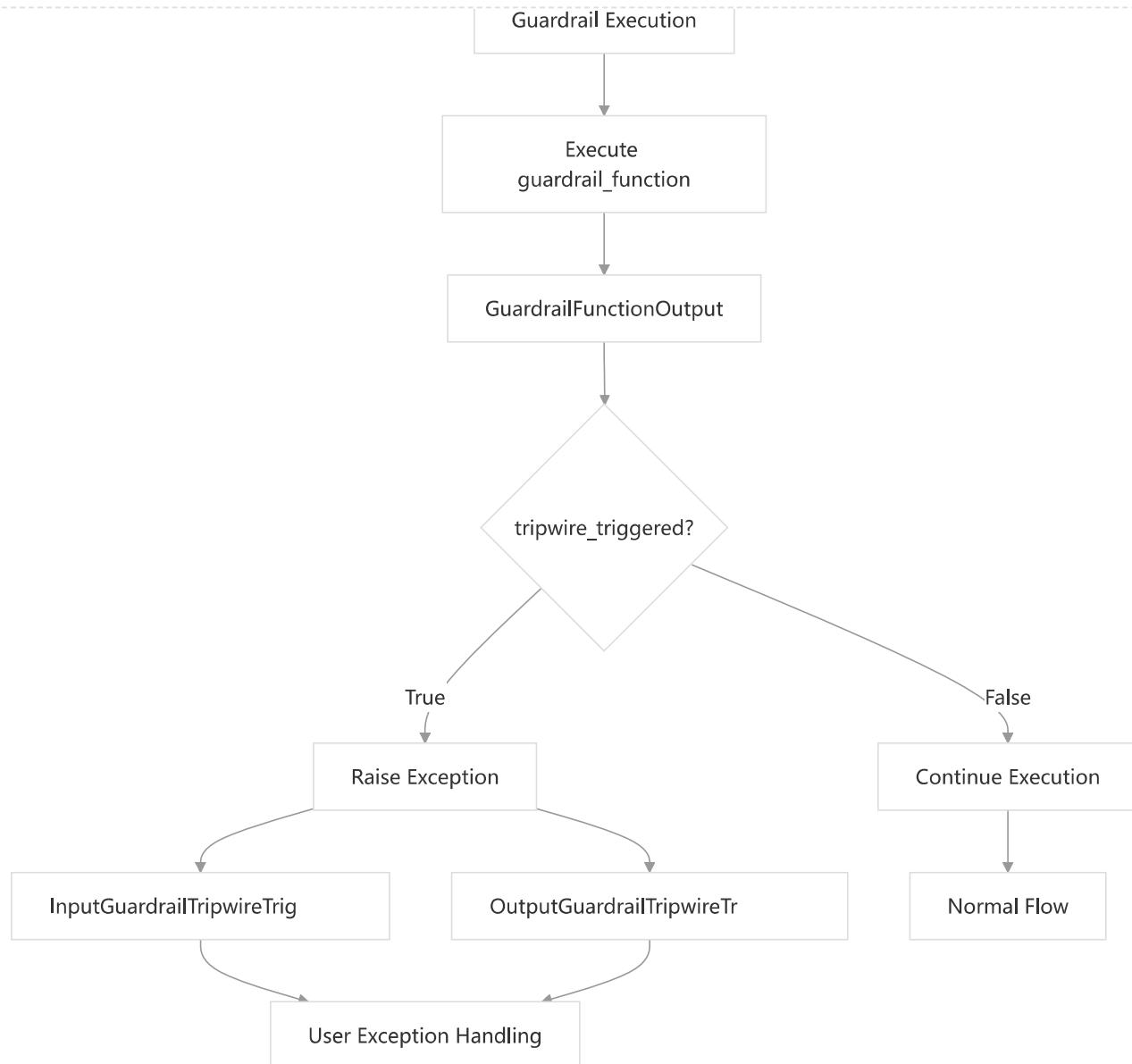
深入研究

绊线系统

继线机制可立即停止执行。

深度维基 openai/openai-agents-python

分享



异常类型

例外

扳机

用法

<code>InputGuardrailTripwireTriggered</code>	输入护栏绊线已激活	处理不安全输入
<code>OutputGuardrailTripwireTriggered</code>	输出护栏绊线已激活	处理不安全的输出

向 Devin 询问 openai/openai-agents-python

深入研究

护栏与代理执行并行运行，以优化性能。这使得我们能够使用轻量级模型进行快速、低成本的安全检查，同时让成本高昂的主模型处理请求。

常见模式：在护栏中使用快速/廉价模型在昂贵的模型执行之前检测恶意使用，从而节省时间和成本。

资料来源：[docs/guardrails.md](#) | 1-3

上下文和状态管理

RunContextWrapper 集成

护栏接收 `RunContextWrapper[TContext]` 提供以下访问权限：

- 本地应用程序上下文
- 会话状态
- 请求元数据
- 依赖注入容器

```
async def context_aware_guardrail(
    ctx: RunContextWrapper[MyContext],
    agent: Agent,
    input: str
) -> GuardrailFunctionOutput:
    user_permissions = ctx.context.user_permissions
    # Context-based validation logic
```

这使得能够根据用户角色、会话状态或特定于应用程序的要求做出上下文敏感的安全决策。

资料来源：[src/agents/guardrail.py](#) | 86-89 [src/agents/guardrail.py](#) | 140-143

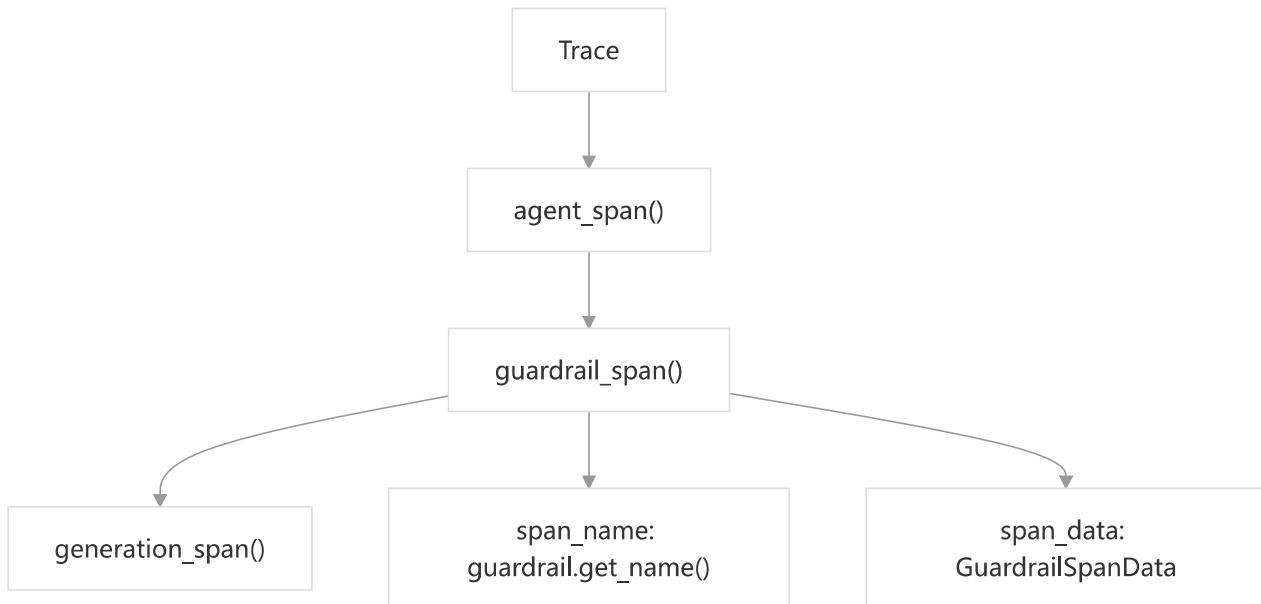
追踪和可观察性

自动创建 Span

护栏会自动包裹 `guardrail_span()` 以便追踪：

向 Devin 询问 [openai/openai-agents-python](#)

深入研究



Span 数据结构

护栏跨度捕获：

- 护栏名称和类型
- 输入/输出数据（受敏感数据设置约束）
- 线状态
- 执行时间
- 错误信息（如果适用）

资料来源： docs/tracing.md | 三十六

敏感数据配置

护栏追踪尊重敏感数据设置，通过 `RunConfig.trace_include_sensitive_data` 控制是否在追踪中捕获护栏输入/输出。

资料来源： docs/tracing.md | 82-85

使用模式

基于代理的护栏

向 Devin 询问 openai/openai-agents-python

深入研究

向 Devin 询问 openai/openai-agents-python

深入研究

> 菜单

会话管理

相关源文件

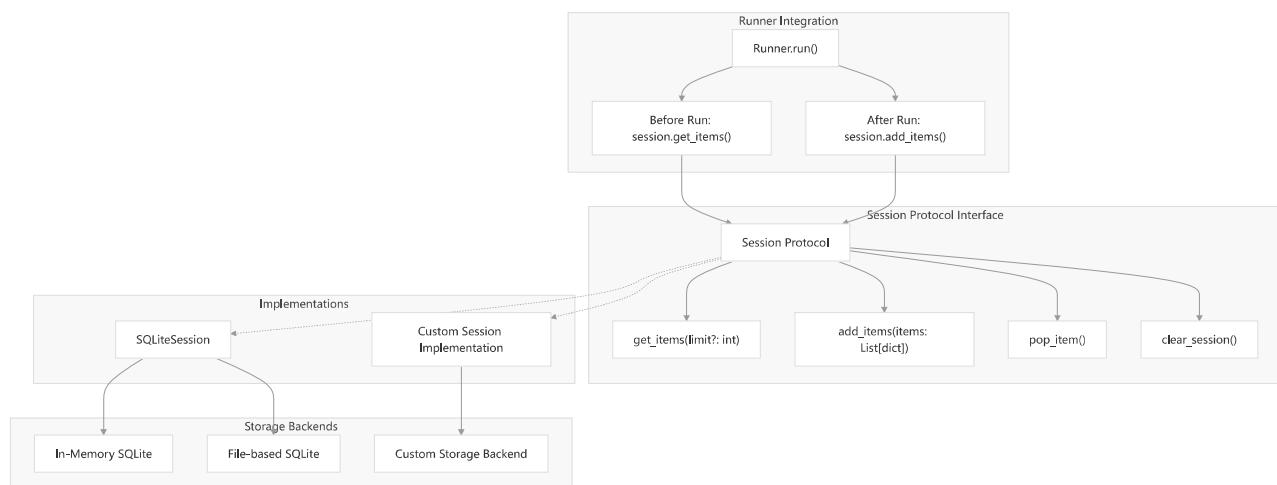
会话管理可在多个代理运行之间自动持久化对话历史记录，从而无需在构建多轮对话应用程序时进行手动状态管理。系统会为不同的会话标识符维护单独的对话上下文，并与 Runner 执行流程无缝集成。

有关使用 手动对话管理的信息 `.to_input_list()`，请参阅[核心组件 - 运行器和执行](#)。有关跟踪对话流的详细信息，请参阅[高级功能 - 跟踪和可观察性](#)。

会话协议和架构

会话系统基于该协议构建 `Session`，该协议定义了用于存储和检索对话历史记录的标准接口。该协议支持不同的存储后端，同时保持一致的行为。

会话协议架构

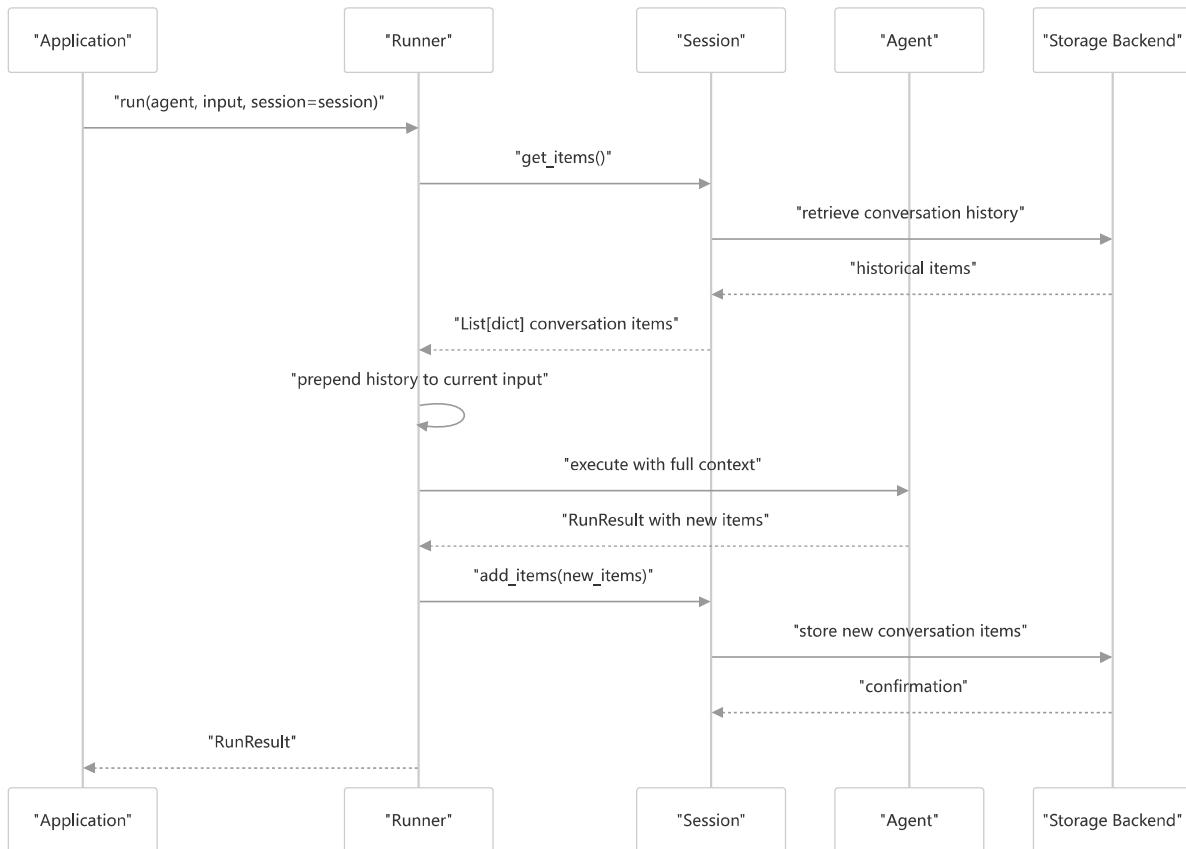


资料来源: [docs/sessions.md](#) | 169-209 [docs/running_agents.md](#) | 90-121

会话生命周期流程

向 Devin 询问 [openai/openai-agents-python](#)

深入研究



资料来源： docs/sessions.md | 48-54 | docs/running_agents.md | 115-120

SQLiteSession 实现

`SQLiteSession` 是使用 SQLite 作为存储后端的主要会话实现。它支持内存和基于文件的持久化。

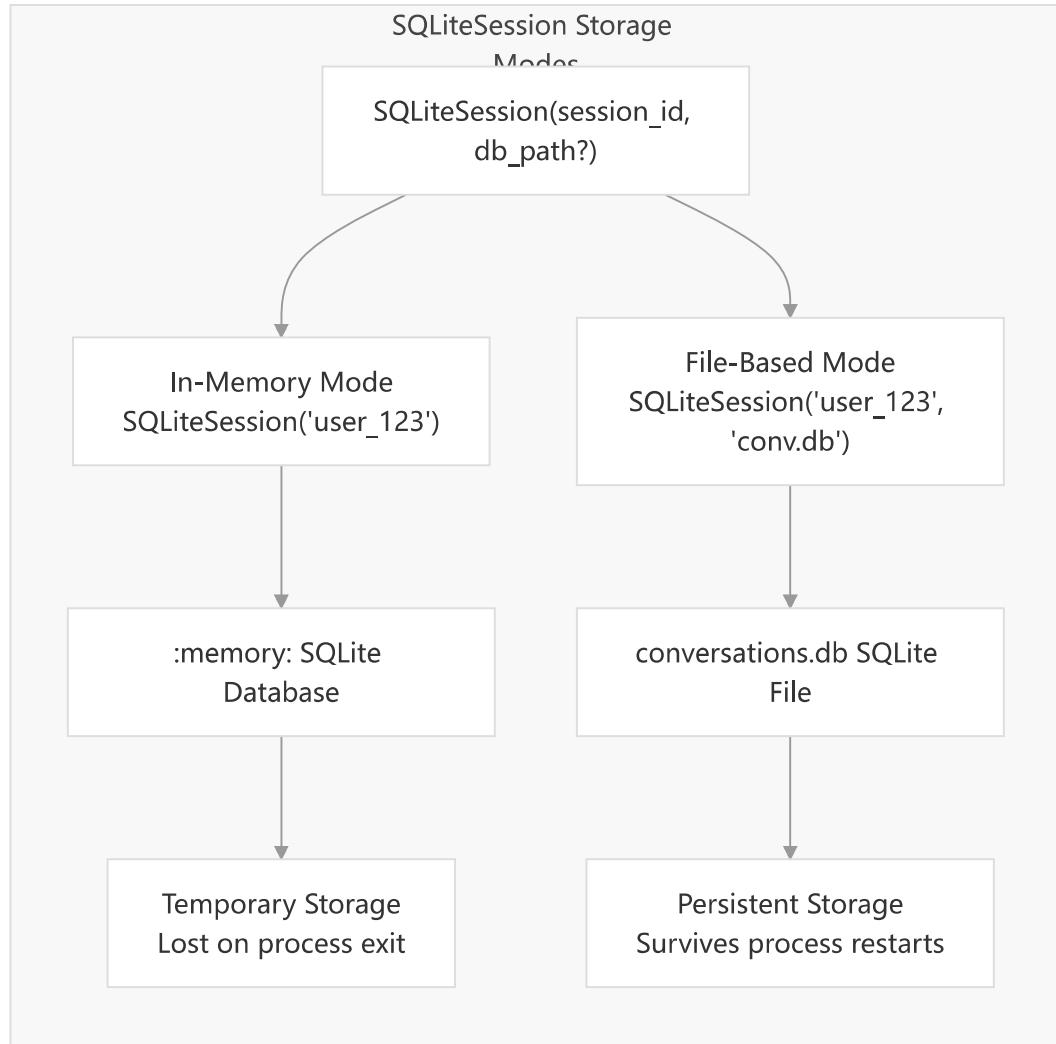
SQLiteSession 配置

范围	类型	描述	例子
<code>session_id</code>	<code>str</code>	对话的唯一标识符	<code>"user_123"</code>
<code>db_path</code>	<code>str</code> (选修的)	SQLite 文件的路径	<code>"conversations.db"</code>

存储选项

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源： docs/sessions.md | 126-143 | 示例/基本/session_example.py | 20-22

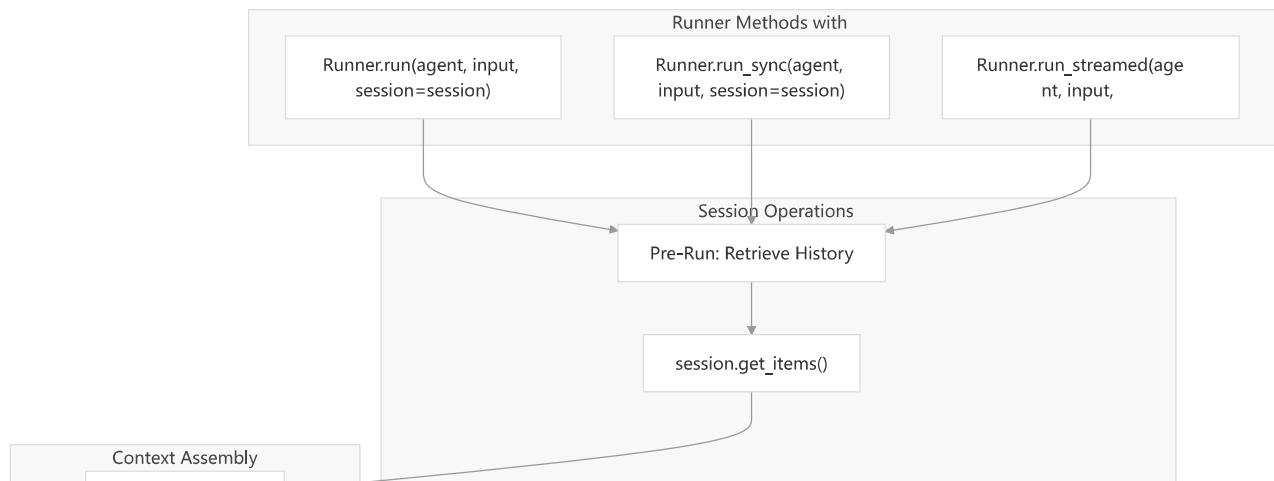
与 Runner 集成

会话 Runner 通过 `session run` 方法中的参数自动与执行系统集成。该集成透明地处理对话历史记录的检索和存储。

运行器会话集成

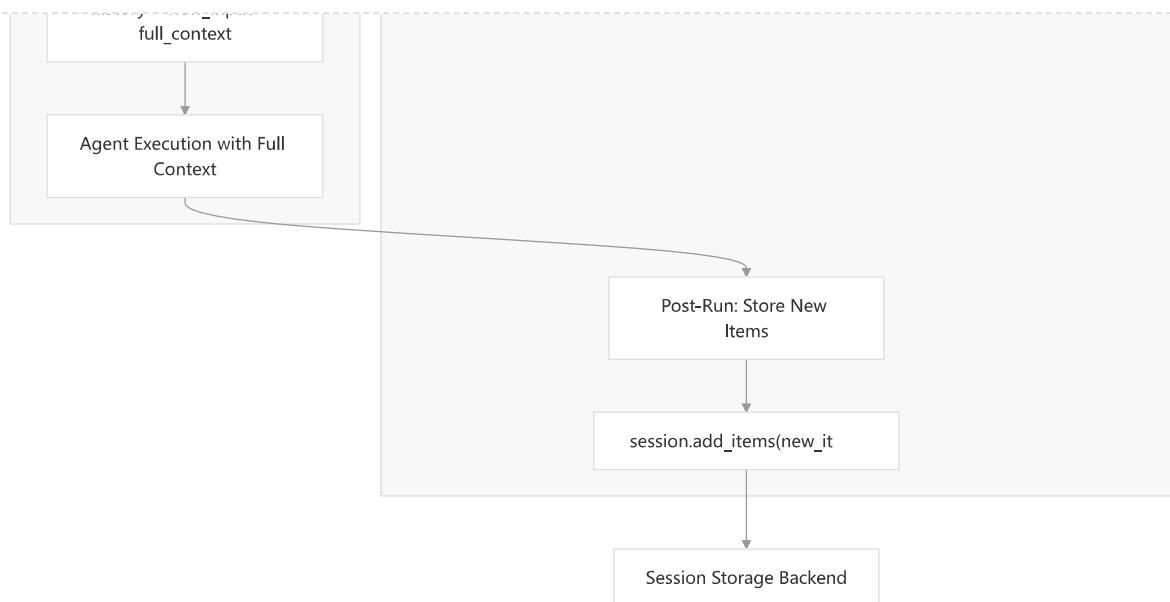
向 Devin 询问 openai/openai-agents-python

深入研究



深度维基 openai/openai-agents-python

分享



资料来源： docs/running_agents.md | 94-113 | docs/sessions.md | 21-44

内存操作

会话系统提供了几种管理对话历史的操作，包括检索、添加、删除和清除对话项目。

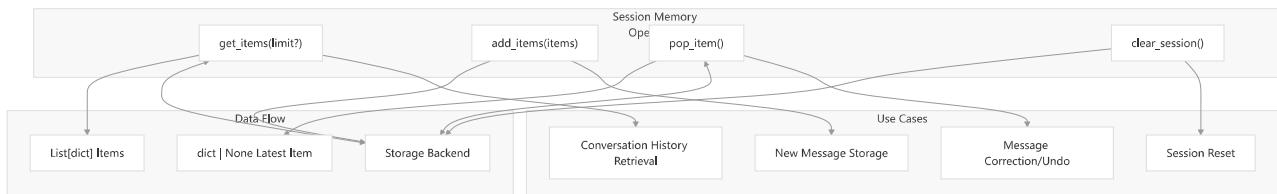
核心内存操作

方法	参数	返回类型	目的
get_items()	limit?: int	List[dict]	检索对话历史记录

向 Devin 询问 openai/openai-agents-python

深入研究

内存操作流程

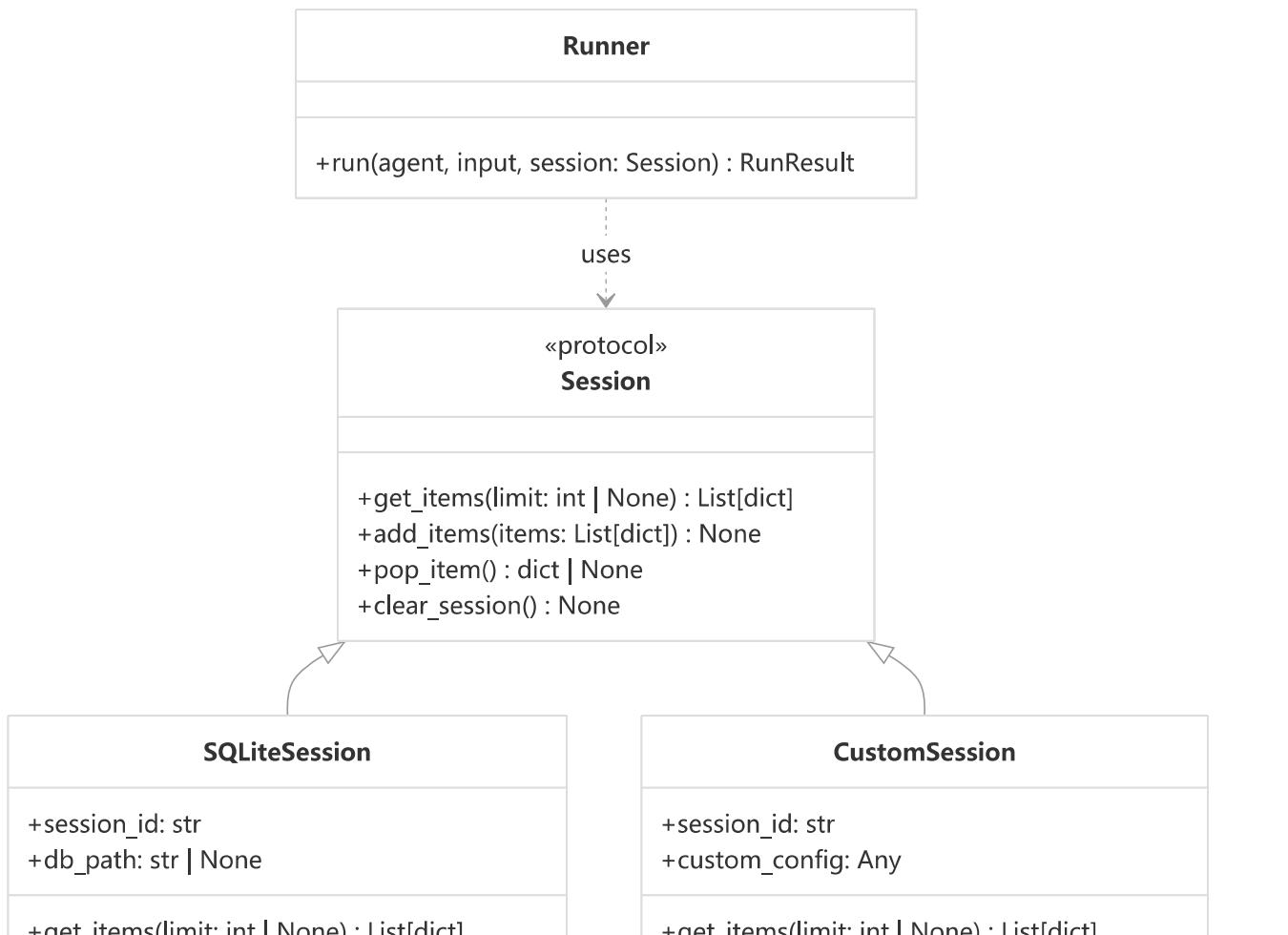


资料来源： docs/sessions.md | 60-84 docs/sessions.md | 85-114

自定义会话实现

该 `Session` 协议通过实现所需的接口方法支持自定义存储后端。自定义实现可以与外部数据库、云存储或专门的对话管理系统集成。

自定义会话协议实现



向 Devin 询问 openai/openai-agents-python

深入研究

资料来源： docs/sessions.md | 168-209

定制实施要求

自定义会话实现必须处理：

- **异步操作：**所有方法都应该异步兼容
- **会话隔离：**不同的 `session_id` 值必须保持单独的对话
- **物品排序：**物品应按时间顺序存储和检索
- **错误处理：**优雅地处理存储后端故障
- **线程安全：**如果存储后端需要，可以进行安全的并发访问

持久性和存储选项

会话持久性根据所选的存储后端而有所不同。系统支持临时和永久存储解决方案。

持久性比较

存储类型	持久性	表现	用例
内存 SQLite	进程寿命	高的	开发、测试
基于文件的 SQLite	永恒的	中等的	单机部署
自定义后端	后端依赖	多变的	生产、分布式系统

向 Devin 询问 openai/openai-agents-python

深入研究

> 菜单

高级功能

[相关源文件](#)

本文档涵盖了 OpenAI Agents SDK 的高级功能，这些功能超越了基本的代理交互。这些功能支持实时对话式 AI、全面的可观察性、多代理编排以及支持语音的应用程序。

有关代理的基本概念和核心功能，请参阅[核心组件](#)。有关各个高级功能的具体实现细节，请参阅专门的子章节：[跟踪和可观察性](#)、[模型上下文协议 \(MCP\)](#)、[实时代理](#)、[代理可视化](#)、[流和实时事件](#)、[切换和多代理工作流](#)以及[语音和音频处理](#)。

高级功能概述

SDK 提供了几个复杂的系统，它们协同工作以支持复杂的代理应用程序：

特征类别	关键组件	主要用例
实时互动	<code>RealtimeSession</code> , <code>OpenAIRuntimeWebSocketMode1</code>	语音对话、实时音频流
可观察性	分布式跟踪、跨度管理	调试、性能监控
多智能体系统	交接、座席协调	复杂的工作流程，专业化
外部集成	MCP 服务器、工具发现	企业系统、外部 API
安全与控制	护栏、中断处理	内容审核、安全限制

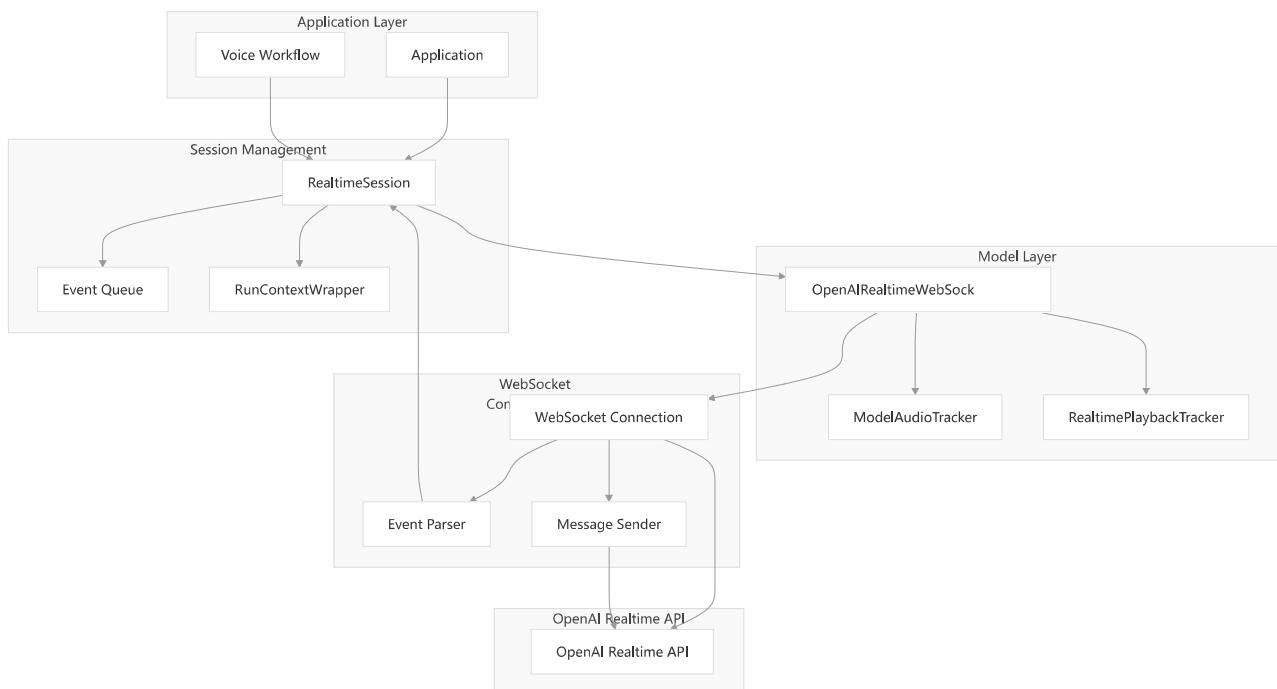
实时代理架构

深度维基 [openai/openai-agents-python](#)

[分享](#)

向 Devin 询问 [openai/openai-agents-python](#)

[深入研究](#)



实时系统架构

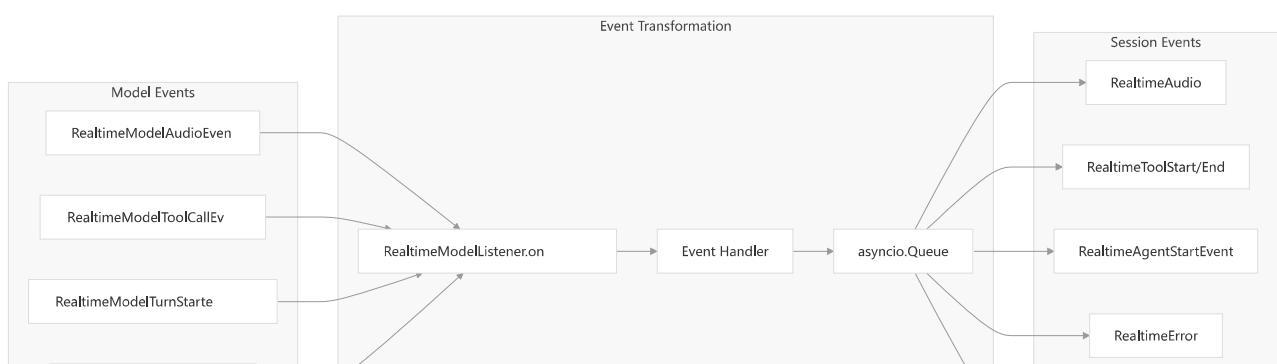
该类 `RealtimeSession` 充当主要协调点，实现 `RealtimeModelListener` 处理来自底层模型的事件的接口。它 `OpenAIRealtimeWebSocketModel` 管理 WebSocket 连接并处理低级协议细节。

资料来源： `src/agents/realtime/session.py` | 52-99

`src/agents/realtime/openai_realtime.py` | 131-145

事件驱动编程模型

实时系统使用复杂的事件转换管道，将低级模型事件转换为高级会话事件。这种分离机制允许应用程序处理语义事件，而 SDK 则负责处理协议细节。



向 Devin 询问 `openai/openai-agents-python`

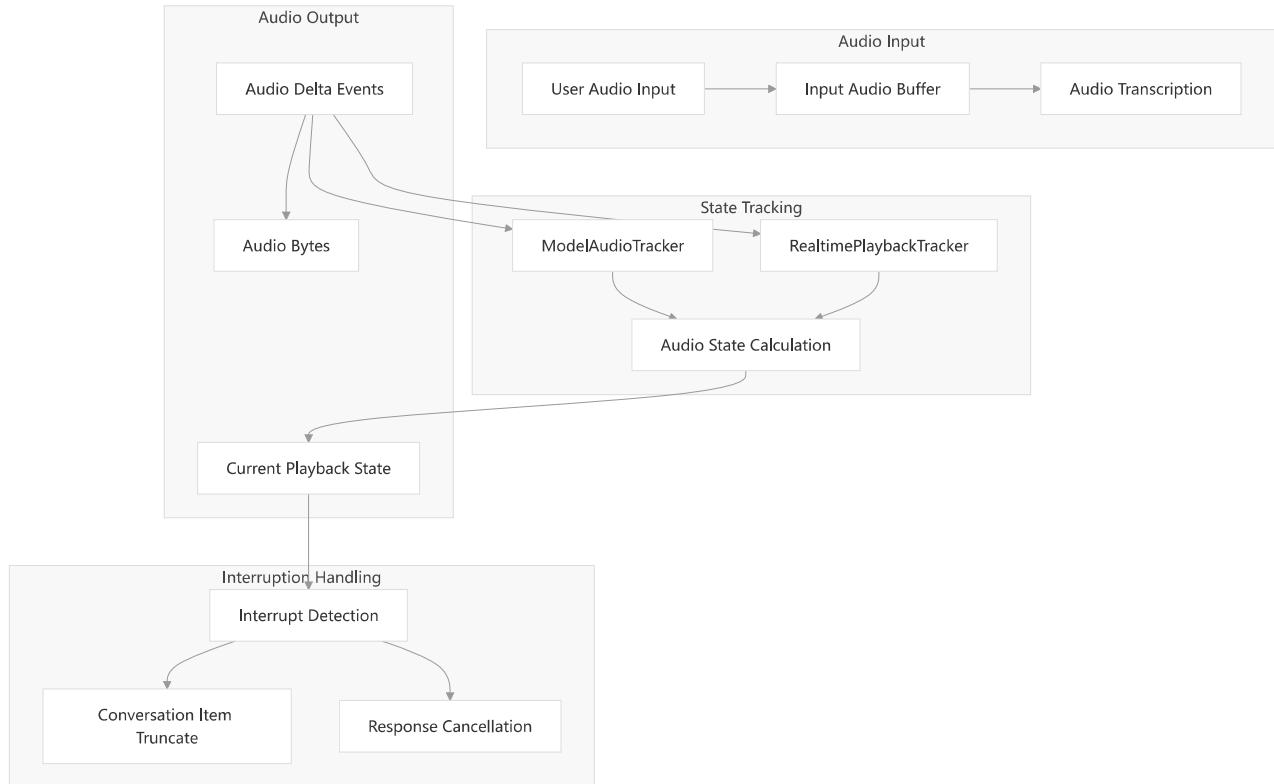
深入研究

事件系统提供两个抽象级别：反映 WebSocket 协议的原始模型事件，以及代表应用程序级概念的已处理会话事件。该 `RealtimeSession.on_event()` 方法实现了转换逻辑。

资料来源：`src/agents/realtime/session.py` | 183-281 `src/agents/realtime/model_events.py` | 1-188
`src/agents/realtime/events.py` | 1-235

音频状态管理

实时音频处理需要精确的状态跟踪来处理中断、播放同步和轮换。SDK 针对不同的用例提供了多种跟踪机制。



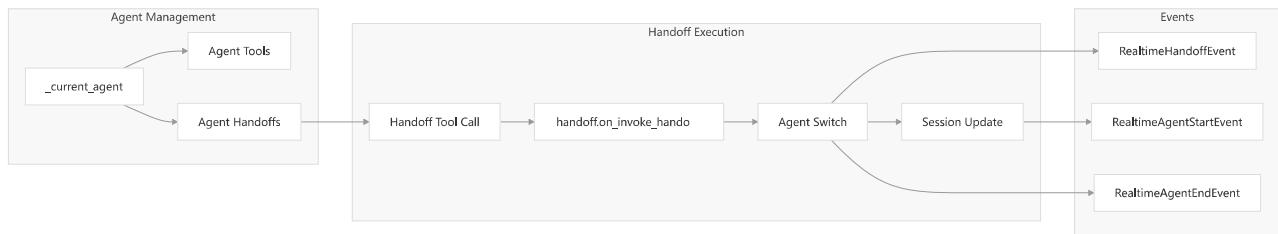
音频状态管理系统

该系统 `ModelAudioTracker` 能够维护音频内容的精确时间信息，从而实现精准的中断处理。当用户中断时，系统会计算精确的毫秒偏移量，并相应地截断对话内容。

资料来源：`src/agents/realtime/openai_realtime.py` | 303-325
`src/agents/realtime/openai_realtime.py` | 326-366 `src/agents/realtime/openai_realtime.py` | 371-387

向 Devin 询问 `openai/openai-agents-python`

深入研究



多代理切换流程

在交接过程中，会话会使用新代理的工具和指令更新模型配置，确保无缝过渡。该 `_handle_tool_call()` 方法负责协调整个交接过程。

资料来源： src/agents/realtime/session.py | 327-378 | src/agents/realtime/session.py | 556-578

配置和安全

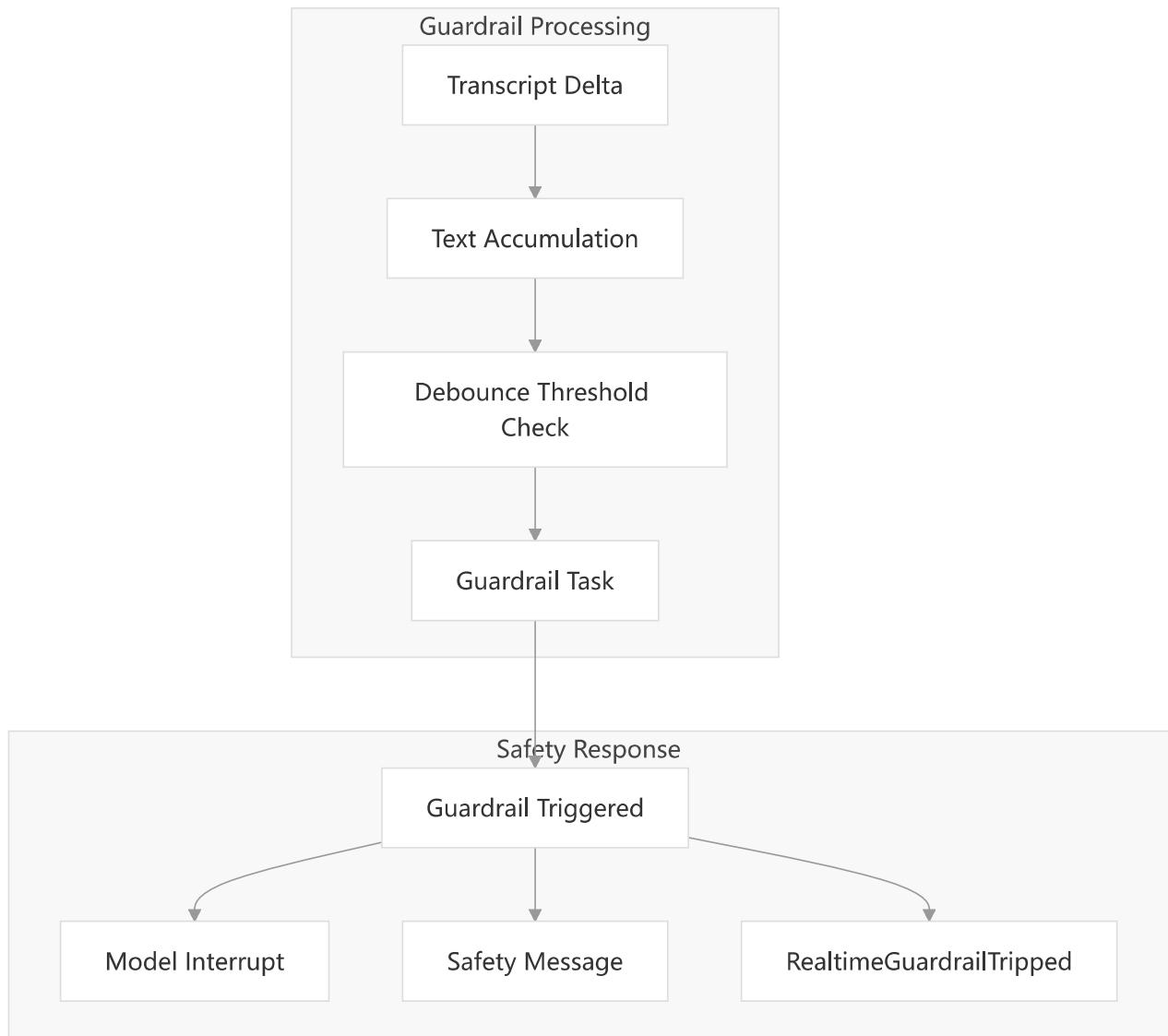
实时系统提供了丰富的配置选项，涵盖模型行为、音频格式、转弯检测和安全机制。护栏可以实时应用，以阻止不适当的内容生成。

配置区域	关键设置	目的
音频格式	<code>input_audio_format</code> , <code>output_audio_format</code>	PCM16、G711编解码器支持
转弯检测	<code>type</code> ,, <code>eagerness</code> <code>interrupt_response</code>	语音活动检测
安全	<code>output_guardrails</code> , <code>debounce_text_length</code>	内容审核
追踪	<code>workflow_name</code> ,, <code>group_id</code> <code>metadata</code>	可观察性集成

护栏系统采用去抖文本累积技术，避免过度处理，同时保持安全覆盖。当护栏触发时，系统会自动中断模型并提供反馈。

向 Devin 询问 openai/openai-agents-python

深入研究



实时护栏系统

向 Devin 询问 openai/openai-agents-python

深入研究

> 菜单

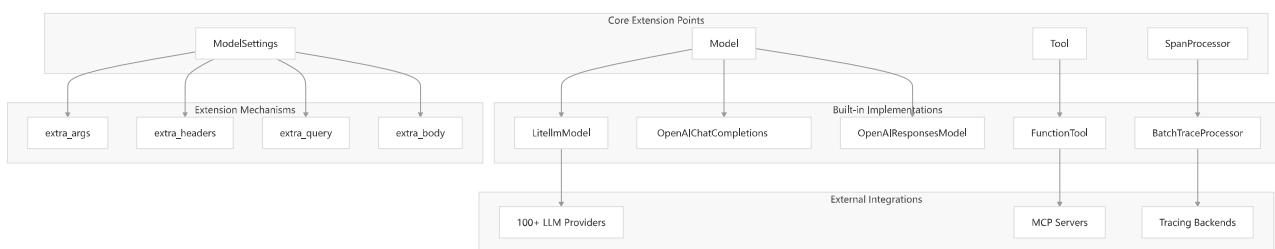
扩展和集成

相关源文件

本文档涵盖 OpenAI Agents SDK 中的第三方集成和扩展模式。该 SDK 提供了多个扩展点，用于集成外部模型提供程序、自定义配置以及构建自定义工具和处理器。

SDK 的扩展架构围绕三个主要领域：用于多提供商 LLM 访问的 LiteLLM 集成、通过 进行的高级模型配置 ModelSettings 以及用于构建专用组件的自定义扩展模式。

扩展架构概述



资料来源：[src/agents/extensions/models/litellm_model.py](#) | 56-71

[src/agents/models/openai_chatcompletions.py](#) | 37-44 [src/agents/models/openai_responses.py](#) | 53-64

[src/agents/model_settings.py](#) | 58-135

LiteLLM 集成

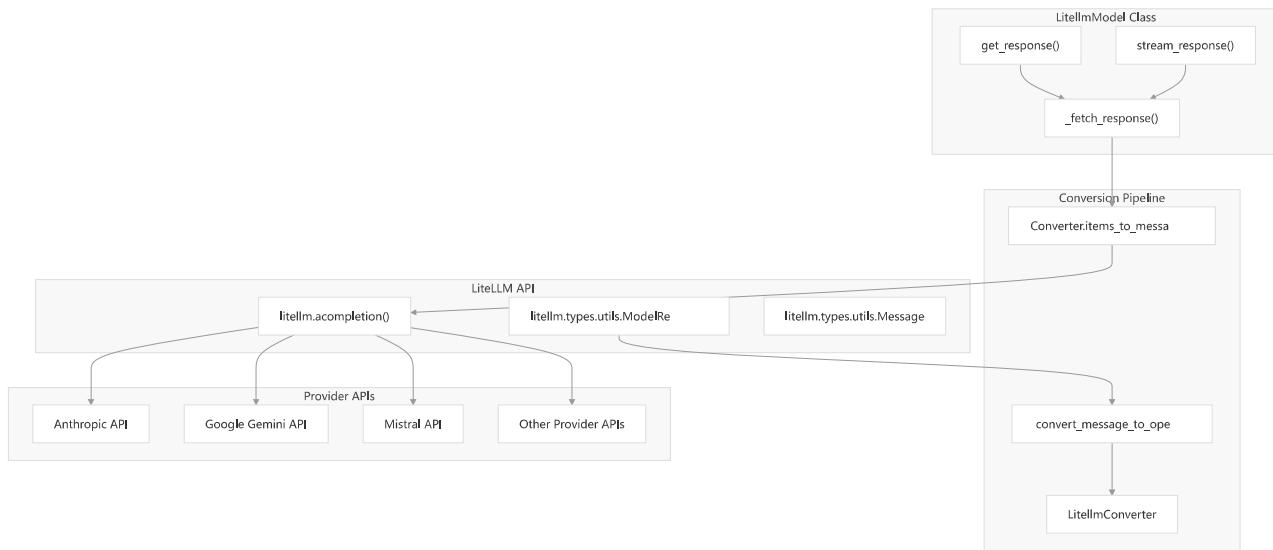
该类 `LitellmModel` 支持通过 LiteLLM 库访问 100 多个模型提供商，支持 Anthropic Claude、Google Gemini、Mistral AI 以及 OpenAI 之外的许多其他提供商。该集成保持与 SDK 代理系统的完全兼容性，同时提供对各种 LLM 提供商的透明访问。

LitellmModel 实现

向 Devin 询问 [openai/openai-agents-python](#)

深入研究





资料来源： `src/agents/extensions/models/litellm_model.py` | 56-159

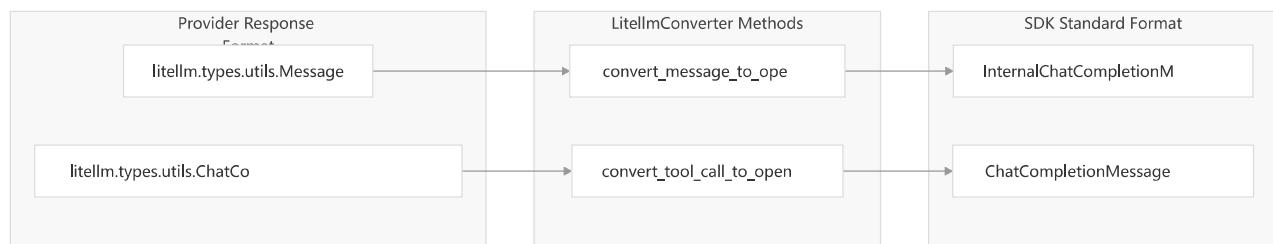
`src/agents/extensions/models/litellm_model.py` | 238-328

提供者消息转换

处理 `LitellmConverter` 提供商特定响应格式和 SDK 标准化格式之间的转换：

转换方法	输入类型	输出类型	目的
<code>convert_message_to_openai</code>	<code>litellm.types.utils.Message</code>	<code>ChatCompletionMessage</code>	主要信息转换
<code>convert_tool_call_to_openai</code>	<code>litellm.types.utils.ChatCompletionMessageToolCall</code>	<code>ChatCompletionMessageToToolCall</code>	工具调用转换
<code>convert_annotations_to_openai</code>	<code>ChatCompletionAnnotation[]</code>	<code>Annotation[]</code>	引用/注释转换

消息转换过程



向 Devin 询问 `openai/openai-agents-python`

深入研究

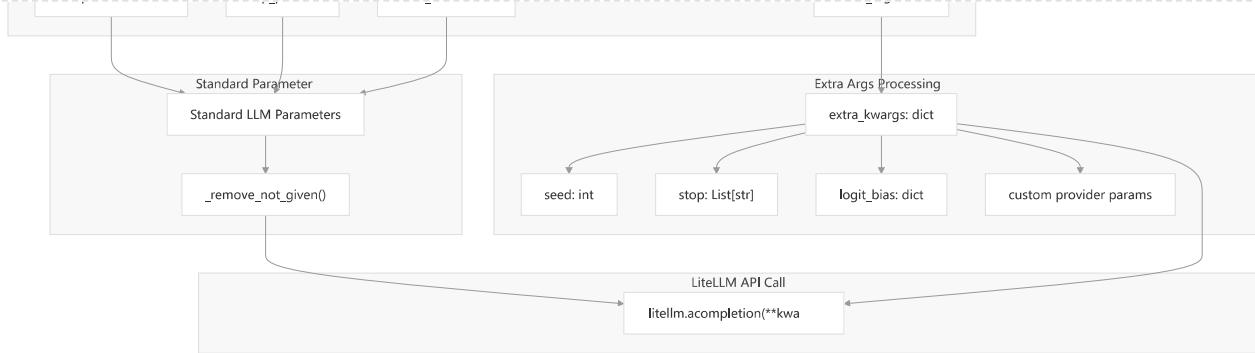
参数转发和配置

LiteLLM 集成通过以下方式支持全面的参数转发 `ModelSettings.extra_args` :

通过 `LitellmModel` 的参数流

深度维基 openai/openai-agents-python

分享



资料来源： src/agents/extensions/models/litellm_model.py | 297-328

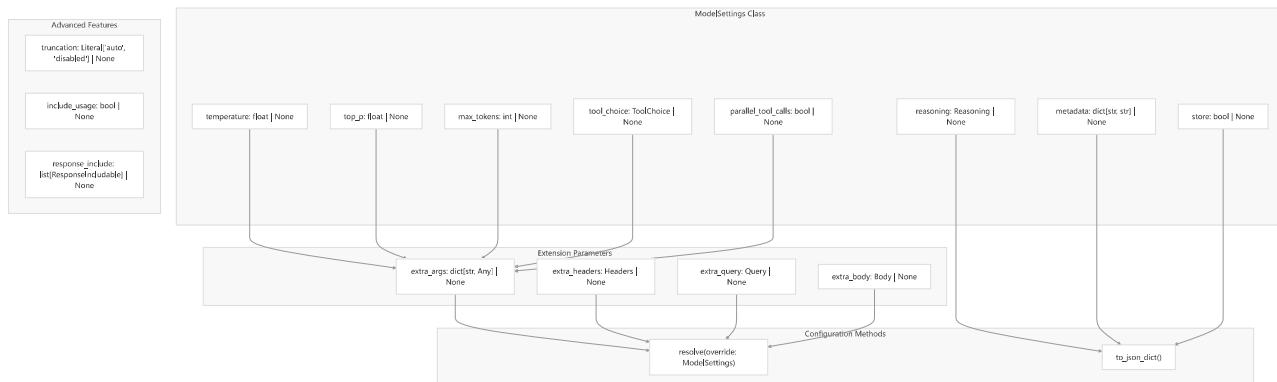
src/agents/extensions/models/litellm_model.py | 350-353

模型设置和配置

该类 `ModelSettings` 为 SDK 中的所有模型提供程序提供主要的配置接口。它支持标准 LLM 参数、高级功能以及通过多种扩展机制实现的可扩展配置。

ModelSettings 架构

ModelSettings 配置类别



向 Devin 询问 openai/openai-agents-python

深入研究

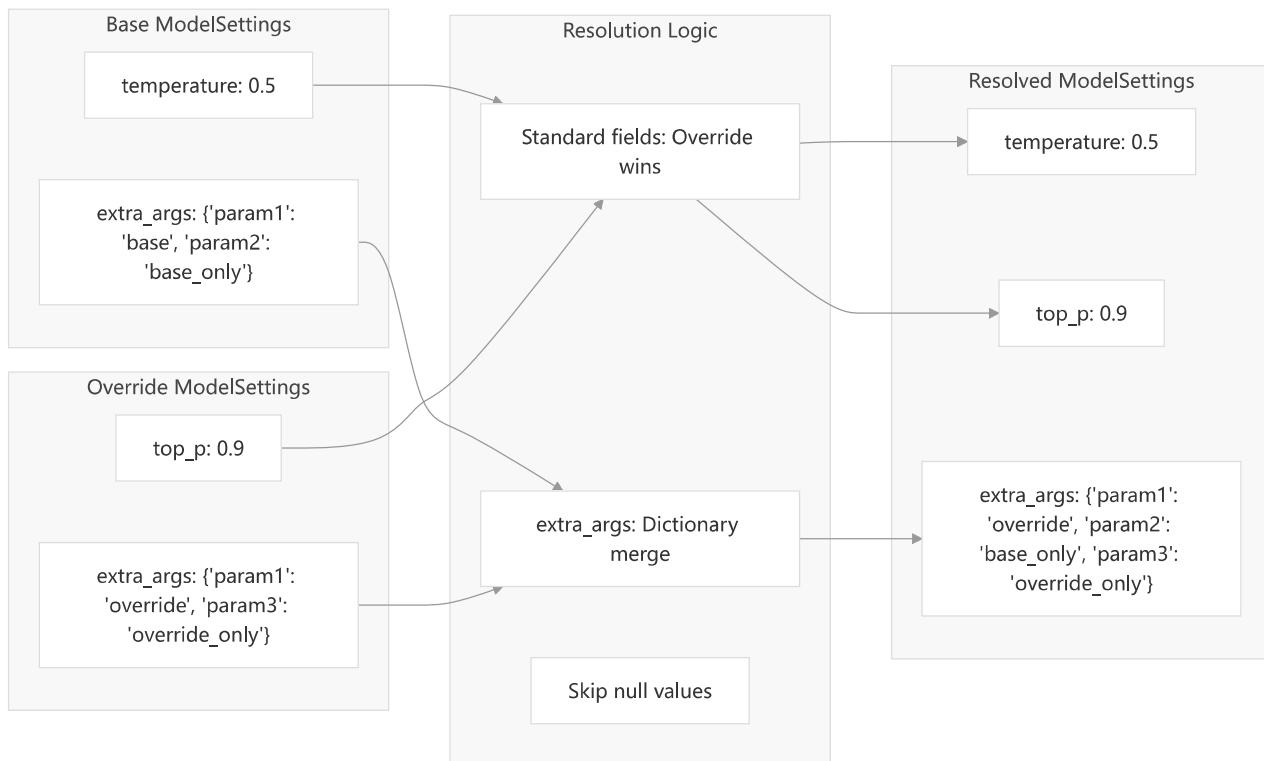
SDK 提供了四种主要的扩展机制来定制模型行为：

扩展类型	类型定义	用例	例子
extra_args	dict[str, Any]	特定于提供商的 API 参数	{"seed": 42, "logit_bias": {123: -100}}
extra_headers	Headers	自定义 HTTP 标头	{"Authorization": "Bearer token"}
extra_query	Query	附加查询参数	{"api_version": "2024-01"}
extra_body	Body	自定义请求正文字段	{"custom_config": {"mode": "enhanced"}}

配置解析

该 `ModelSettings.resolve()` 方法可实现具有智能合并的分层配置：

配置解析过程

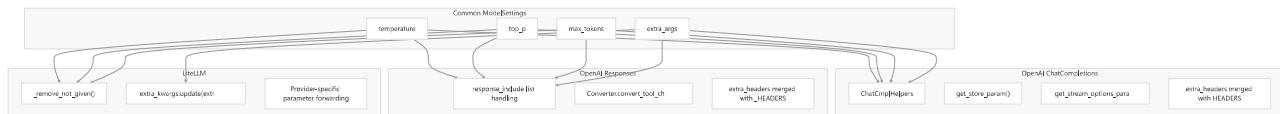


资料来源： src/agents/model_settings.py | 136-157

特定于提供商的配置

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源： `src/agents/models/openai_chatcompletions.py` | 267-295

`src/agents/models/openai_responses.py` | 267-290

`src/agents/extensions/models/litellm_model.py` | 297-327

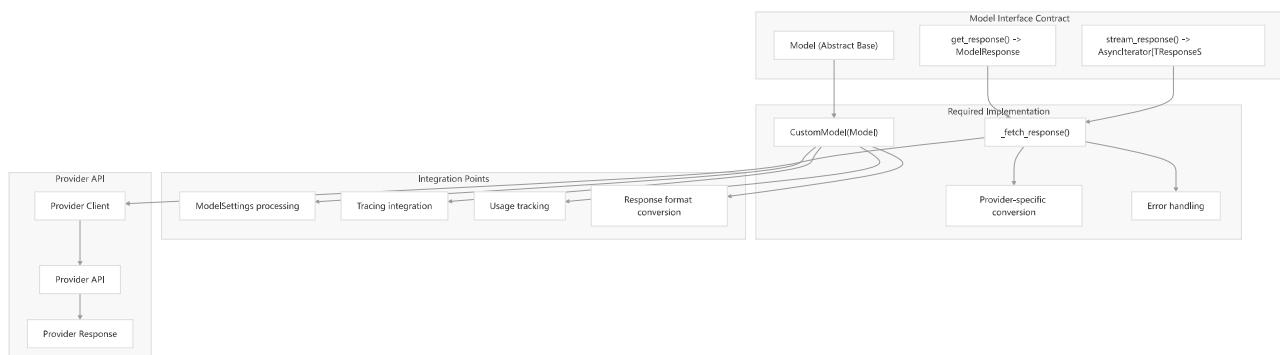
自定义扩展

SDK 提供了多个扩展点，用于构建与代理系统无缝集成的自定义组件。自定义扩展可以包括自定义模型提供程序、专用工具、自定义处理器以及特定领域的集成。

自定义模型提供程序实现

自定义模型提供程序实现 `Model` 接口以与任何 LLM 提供程序或推理系统集成：

模型接口实现模式



资料来源： `src/agents/extensions/models/litellm_model.py` | 56-71

`src/agents/models/openai_chatcompletions.py` | 37-44

自定义工具扩展

自定义工具通过实现专门功能或与外部服务集成来扩展代理的功能：

扩展类型	基类	用例	实现模式
------	----	----	------

向 Devin 询问 `openai/openai-agents-python`

深入研究

扩展类型**基类****用例****实现模式**

异步工具

Tool

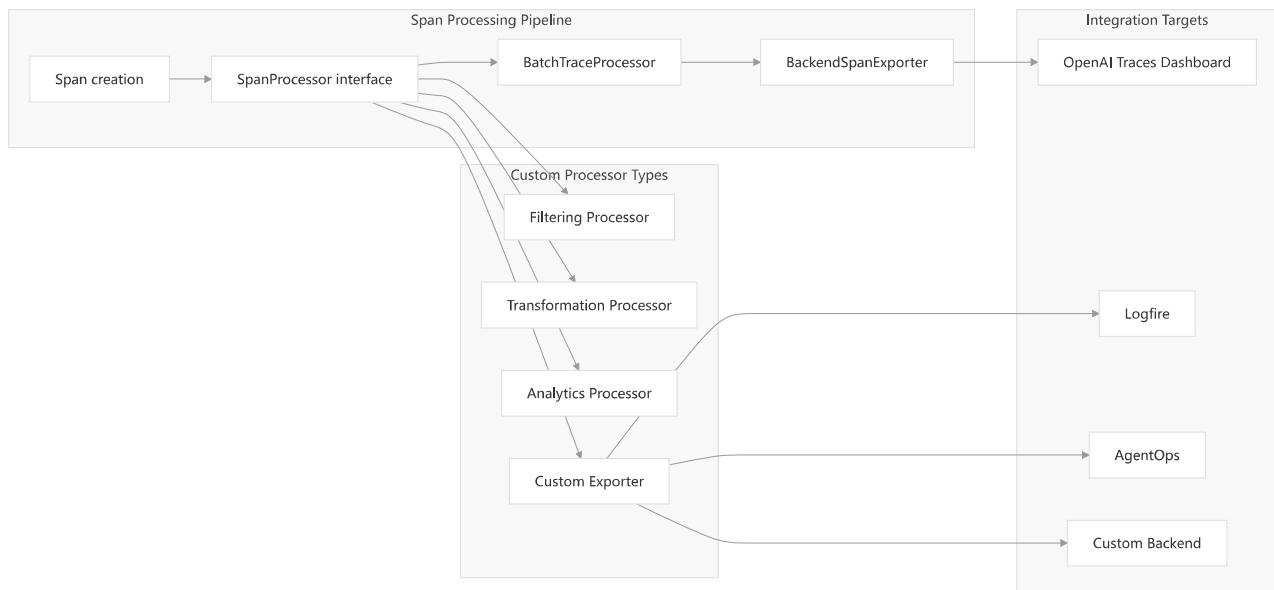
长时间运行的操作

异步函数实现

自定义处理器扩展

自定义处理器可以专门处理代理执行跟踪和事件：

定制处理器架构



资料来源： src/agents/model_settings.py | 58-135

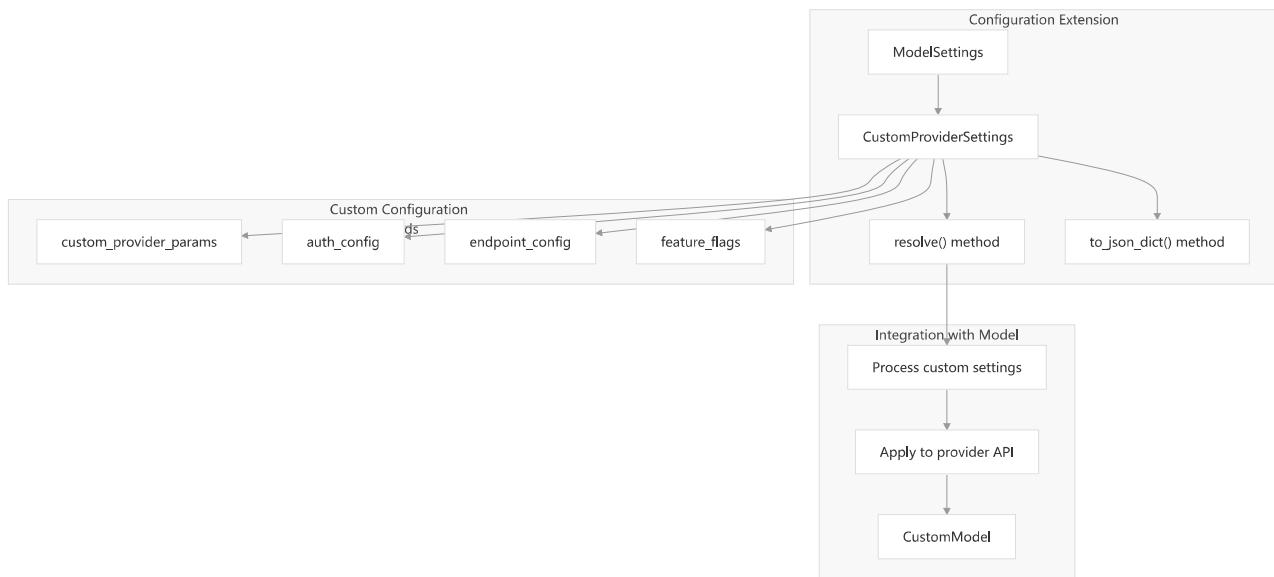
扩展开发模式

开发自定义扩展的常见模式：

配置扩展模式

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源：src/agents/model_settings.py | 136-171

测试自定义扩展

SDK 提供了用于验证自定义扩展的测试模式和实用程序：

测试区域	图案	关键验证点
参数转发	模拟提供者，捕获 kwargs	自定义参数正确传递
响应转换	模拟响应，验证格式	SDK 格式兼容性
错误处理	异常注入	适当的错误传播
配置解析	多级设置	合并逻辑正确性

资料来源：测试/模型/test_kwargs_functionality.py | 16-178

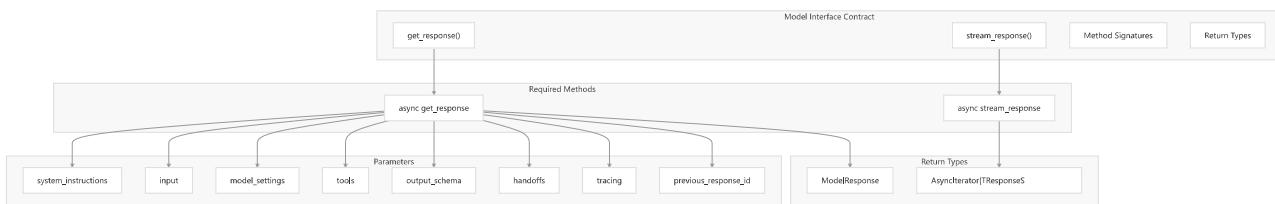
测试/test_openai_chatcompletions.py | 232-384

自定义提供商集成

SDK 通过 `Model` 接口支持自定义模型提供程序，从而实现与任何 LLM 提供程序或自定义推理系统的集成。

向 Devin 询问 openai/openai-agents-python

深入研究



资料来源： src/agents/models/interface.py

实现模式

自定义提供程序应遵循以下实现模式：

1. 继承自 Model 接口
2. 实现所需的异步方法
3. 处理 ModelSettings 配置
4. 在内部格式和提供商格式之间转换
5. 支持追踪集成
6. 提供适当的错误处理

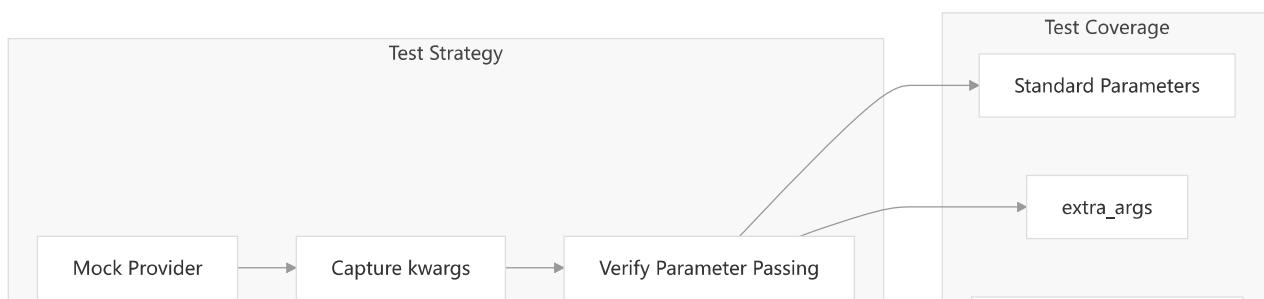
资料来源： src/agents/extensions/models/litellm_model.py | 48-346

src/agents/models/openai_chatcompletions.py | 36-320

测试集成扩展

SDK 为集成扩展提供了全面的测试模式：

参数转发测试



向 Devin 询问 openai/openai-agents-python

深入研究

> 菜单

开发和测试

相关源文件

本文档涵盖 OpenAI Agents SDK 的开发工作流程、测试模式和项目维护，并为贡献者提供配置管理、测试策略和文档系统方面的技术指导。

有关核心 SDK 功能，请参阅[核心组件](#)。有关部署模式，请参阅[扩展和集成](#)。

配置和设置

OpenAI Agents SDK 使用 `uv` 工作区管理与 `hatchling` 构建后端。它 `pyproject.toml` 定义了项目元数据、依赖项组和工具配置。

依赖架构

该项目将核心依赖项与可选功能组和开发工具分开：

依赖组	关键软件包	目的
核	<code>openai>=1.97.1</code> , , <code>pydantic>=2.10</code> <code>griffe>=1.5.6</code>	基本 SDK 功能
voice	<code>numpy>=2.2.0</code> , <code>websockets>=15.0</code>	音频处理能力
viz	<code>graphviz>=0.17</code>	代理可视化
litellm	<code>litellm>=1.67.4.post1</code>	多供应商法学硕士 (LLM) 支持
realtime	<code>websockets>=15.0</code>	实时通信
dev	<code>pytest</code> , , <code>mypy</code> <code>ruff</code> <code>mkdocs</code>	开发工具链

开发工具链配置

向 Devin 询问 `openai/openai-agents-python`



皱褶配置 ([tool.ruff])

- `line-length = 100` 和 `target-version = "py39"`
- `[tool.ruff.lint]` 包含的 Linting 规则 `select = ["E", "W", "F", "I", "B", "C4", "UP"]`
- 通过导入排序 `isort = {combine-as-imports = true, known-first-party = ["agents"]}`

MyPy 配置 ([tool.mypy])

- `strict = true` 通过选择性放松 `disallow_incomplete_defs = false`
- 模块覆盖 `[[tool.mypy.overrides]]` 外部依赖项，例如 `sounddevice.*`

覆盖范围配置 ([tool.coverage])

- 通过以下方式进行源跟踪 `source = ["tests", "src/agents"]`
- `TYPE_CHECKING` 块和 `@abc.abstractmethod` 定义的排除

资料来源： pyproject.toml | 1-80 pyproject.toml | 82-126

测试模式

SDK 使用 `pytest` 来 `pytest-asyncio` 测试异步工作流、流式响应和模型交互。测试配置在 中定义 `[tool.pytest.ini_options]`。

Pytest 配置和标记

该 `[tool.pytest.ini_options]` 部分配置：

- `asyncio_mode = "auto"` 用于自动异步测试检测
- `asyncio_default_fixture_scope = "session"` 用于共享事件循环
- 自定义标记，如 `allow_call_model_methods` 用于真实模型集成测试
- 预期异步协程警告的警告过滤器

测试基础设施映射

向 Devin 询问 openai/openai-agents-python

深入研究

- 实现 `async def stream_response()` 返回 `AsyncIterator[TResponseStreamEvent]`
- 用于 `ResponseTextDeltaEvent` 流 `ResponseCompletedEvent` 模拟
- 方法类似 `set_next_output()` 和 `add_multiple_turn_outputs()` 配置测试场景

使用 inline-snapshot 进行快照测试

该 `inline-snapshot` 包直接在测试代码中捕获复杂的数据结构。该 `snapshot()` 函数验证对话历史记录和工作流状态：

```
assert workflow._input_history == snapshot([
    {"content": "transcription_1", "role": "user"},
    {
        "arguments": '{"a": "b"}',
        "call_id": "2",
        "name": "some_function",
        "type": "function_call",
        "id": "1",
    },
    # Additional conversation turns...
])

```

该 `[tool.inline-snapshot]` 配置用于 `ruff format` 通过进行代码格式化 `format-command = "ruff format --stdin-filename {filename}"`。

异步工作流测试

测试函数使用 `@pytest.mark.asyncio` 修饰符来实现异步执行。该 `test_single_agent_workflow()` 函数演示了以下测试模式：

- `SingleAgentVoiceWorkflow` 使用 `Agent` 实例进行初始化
- `workflow.run()` 对结果进行异步迭代
- `workflow._input_history` 通过和验证内部状态 `workflow._current_agent`
- `FakeStreamingModel` 使用预配置输出进行多轮对话测试

资料来源： `pyproject.toml` | 128-141 | 测试/语音/test_workflow.py | 32-200

开发工具和工作流程

向 Devin 询问 `openai/openai-agents-python`

深入研究

- Pyflakes 用于未使用的导入和变量 (F)
- 使用 isort 进行导入排序 (I)
- 常见 bug 的 Bugbear (B)
- 理解力提升 (C4)
- Python 升级建议 (UP)

类型检查在严格模式下强制执行 `mypy`，确保整个代码库的类型安全，同时为逐步采用类型提供一定的灵活性。

资料来源： `pyproject.toml` | 82-108

覆盖率报告

代码覆盖率跟踪涵盖测试文件和源代码，并可智能排除类型检查块、抽象方法和调试日志记录。该配置会按覆盖率百分比对覆盖率报告进行排序，以突出显示需要关注的区域。

资料来源： `pyproject.toml` | 114-126

发布流程和版本控制

SDK 遵循经过修改的语义版本控制方案，其格式为 `0.Y.Z`，其中前导零表示快速演进。版本增量遵循特定规则：

- 次要版本 (Y)：对公共接口的重大更改
- 补丁版本 (Z)：非重大更改，包括错误修复、新功能和 Beta 功能更新

发布流程包含维护重大变更日志，以帮助用户了解升级的影响。最近的重大变更主要集中在类型系统改进上，例如更改 `Agent` 参数以 `AgentBase` 提高类型安全性。

资料来源： 文档/发布.md | 1-29

版本管理

该项目在补丁版本内保持向后兼容性，同时清晰地记录次要版本中的重大变更。希望避免重大变更的用户可以锁定到特定的 `0.Y.x` 版本范围。

资料来源： 文档/发布.md | 3-19

向 Devin 询问 `openai/openai-agents-python`

深入研究

向 Devin 询问 openai/openai-agents-python

深入研究



> 菜单

示例和模式

相关源文件

本页面提供使用 OpenAI Agents SDK 构建应用程序的常见使用模式、实际示例和最佳实践。它演示了代理配置、工具集成、会话管理、多代理工作流、防护措施和跟踪等核心概念的实际实现。

有关组件的详细文档，请参阅[核心组件](#)。有关跟踪和 MCP 集成等高级功能，请参阅[高级功能](#)。

基本代理创建模式

SDK 提供了几种用于创建和配置具有不同功能的代理的基本模式。

简单代理模式

最基本的代理模式涉及创建一个 `Agent` 带有指令的代理并使用以下命令运行它 `Runner`：

```
from agents import Agent, Runner

agent = Agent(name="Assistant", instructions="You are a helpful assistant")
result = Runner.run_sync(agent, "Write a haiku about recursion in programming.")
```

模式分解：

- `Agent` 带有 `name` 和 `instructions` 参数的类
- `Runner.run_sync()` 用于同步执行
- `Runner.run()` 用于异步执行（需要 `await`）

资料来源：自述文件.md | 154-165

工具启用代理模式

向 Devin 询问 openai/openai-agents-python

深入研究



```
@function_tool
def get_weather(city: str) -> str:
    return f"The weather in {city} is sunny."

agent = Agent(
    name="Weather Assistant",
    instructions="You are a helpful agent.",
    tools=[get_weather],
)

result = await Runner.run(agent, "What's the weather in Tokyo?")
```

模式分解：

- `@function_tool` 用于函数到工具转换的装饰器
- 工具已添加到 `Agent.tools` 列表
- 自动函数模式生成和工具执行

资料来源：自述文件.md | 204-232

代理循环执行流程

代理执行循环的内部工作方式如下：

向 Devin 询问 openai/openai-agents-python

深度维基 openai/openai-agents-python

分享



向 Devin 询问 openai/openai-agents-python

深入研究

关键组件：

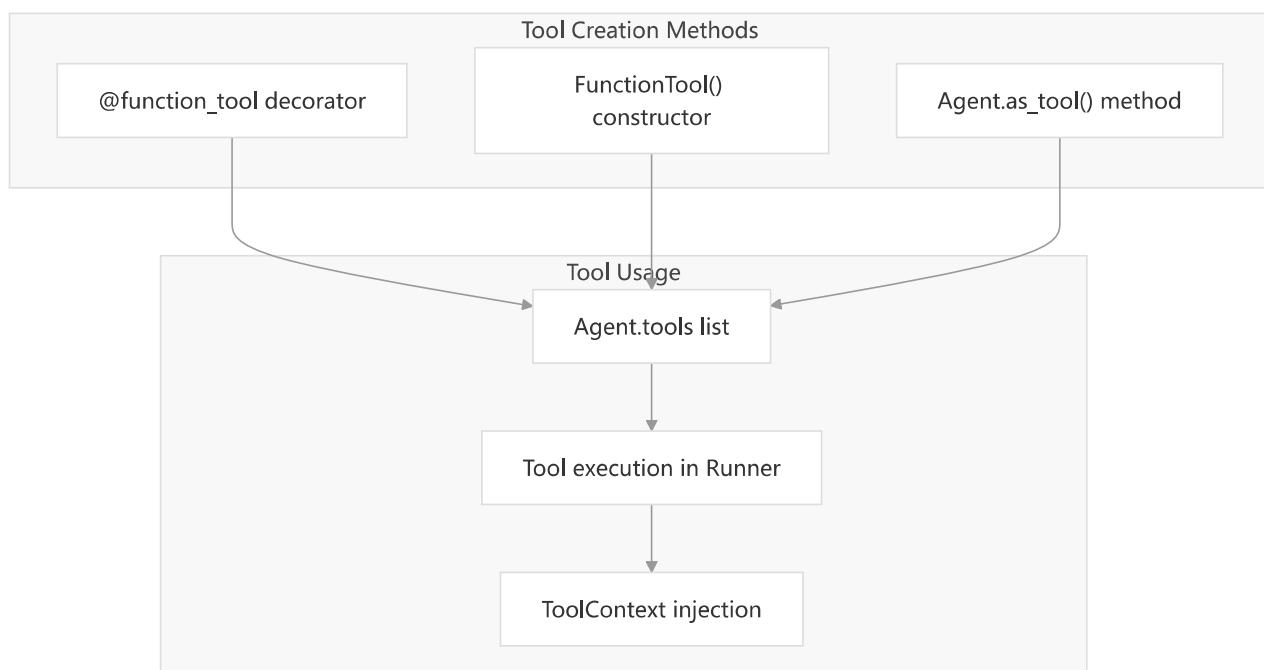
- **Runner** 管理执行循环
- **Agent** 提供配置和行为
- **ModelResponse** 包含 LLM 输出
- 工具调用触发函数执行
- 代理之间的切换控制

资料来源：自述文件.md | 234-257

工具集成模式

SDK 支持多种将工具与代理集成的方法。

功能工具创建模式



工具创建方法：

1. 装饰器模式：@function_tool Python 函数

向 Devin 询问 openai/openai-agents-python

深入研究

工具执行上下文模式

工具通过依赖注入接收执行上下文：

```
from agents import function_tool, ToolContext

@function_tool
def context_aware_tool(message: str, context: ToolContext) -> str:
    # Access agent information, session, tracing, etc.
    return f"Agent {context.agent.name} processed: {message}"
```

上下文组件：

- `ToolContext` 提供对当前代理、会话和执行状态的访问
- 工具声明上下文参数时自动注入
- 使工具能够访问更广泛的执行环境

资料来源：基于上下文管理文档

会话管理模式

会话管理支持跨多个代理运行的对话历史记录持久化。

基本会话模式

```
from agents import Agent, Runner, SQLiteSession

agent = Agent(name="Assistant", instructions="Reply very concisely.")
session = SQLiteSession("conversation_123")

# First turn
result = await Runner.run(agent, "What city is the Golden Gate Bridge in?", session=session)

# Second turn - automatic context retention
result = await Runner.run(agent, "What state is it in?", session=session)
```

会话生命周期：

向 Devin 询问 openai/openai-agents-python

深入研究

资料来源：自述文件.md | 26-61

自定义会话实现模式

```
from agents.memory import Session
from typing import List

class MyCustomSession:
    def __init__(self, session_id: str):
        self.session_id = session_id

    @async def get_items(self, limit: int | None = None) -> List[dict]:
        # Retrieve conversation history
        pass

    @async def add_items(self, items: List[dict]) -> None:
        # Store new items
        pass

    @async def pop_item(self) -> dict | None:
        # Remove most recent item
        pass

    @async def clear_session(self) -> None:
        # Clear all items
        pass
```

会话协议要求：

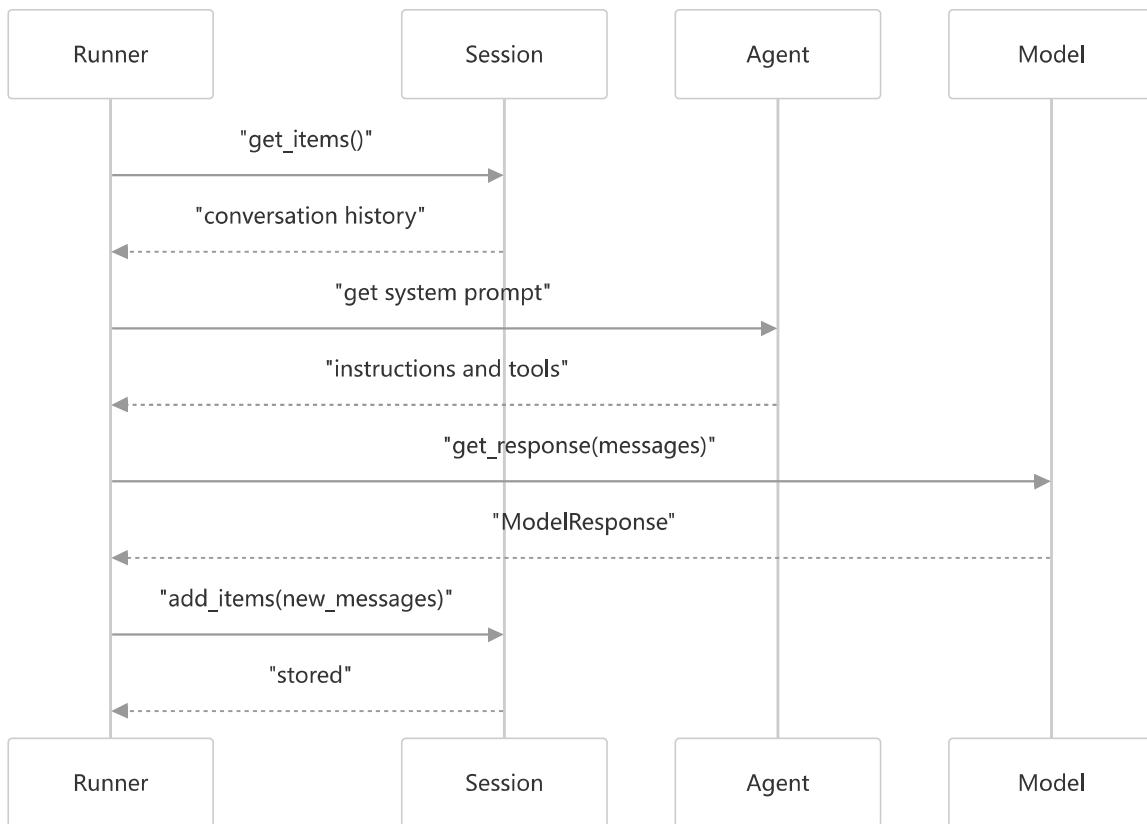
- `get_items()`：检索对话历史记录
- `add_items()`：存储新的对话项目
- `pop_item()`：移除并返回最新项目
- `clear_session()`：清除会话数据

资料来源：自述文件.md | 88-126

会话数据流

向 Devin 询问 openai/openai-agents-python

深入研究



数据流组件：

- **Runner** 协调会话交互
- **Session** 提供历史持久性
- **Agent** 贡献系统配置
- **Model** 生成响应

来源：基于 README.md 中的会话管理示例

多代理工作流模式

SDK 通过切换和代理编排支持复杂的多代理交互。

座席交接模式

```

from agents import Agent, Runner

spanish_agent = Agent(
  
```

向 Devin 询问 openai/openai-agents-python

深入研究

```
name="English agent",
instructions="You only speak English",
)

triage_agent = Agent(
    name="Triage agent",
    instructions="Handoff to the appropriate agent based on the language of the request.",
    handoffs=[spanish_agent, english_agent],
)

result = await Runner.run(triage_agent, "Hola, ¿cómo estás?")
```

交接组件：

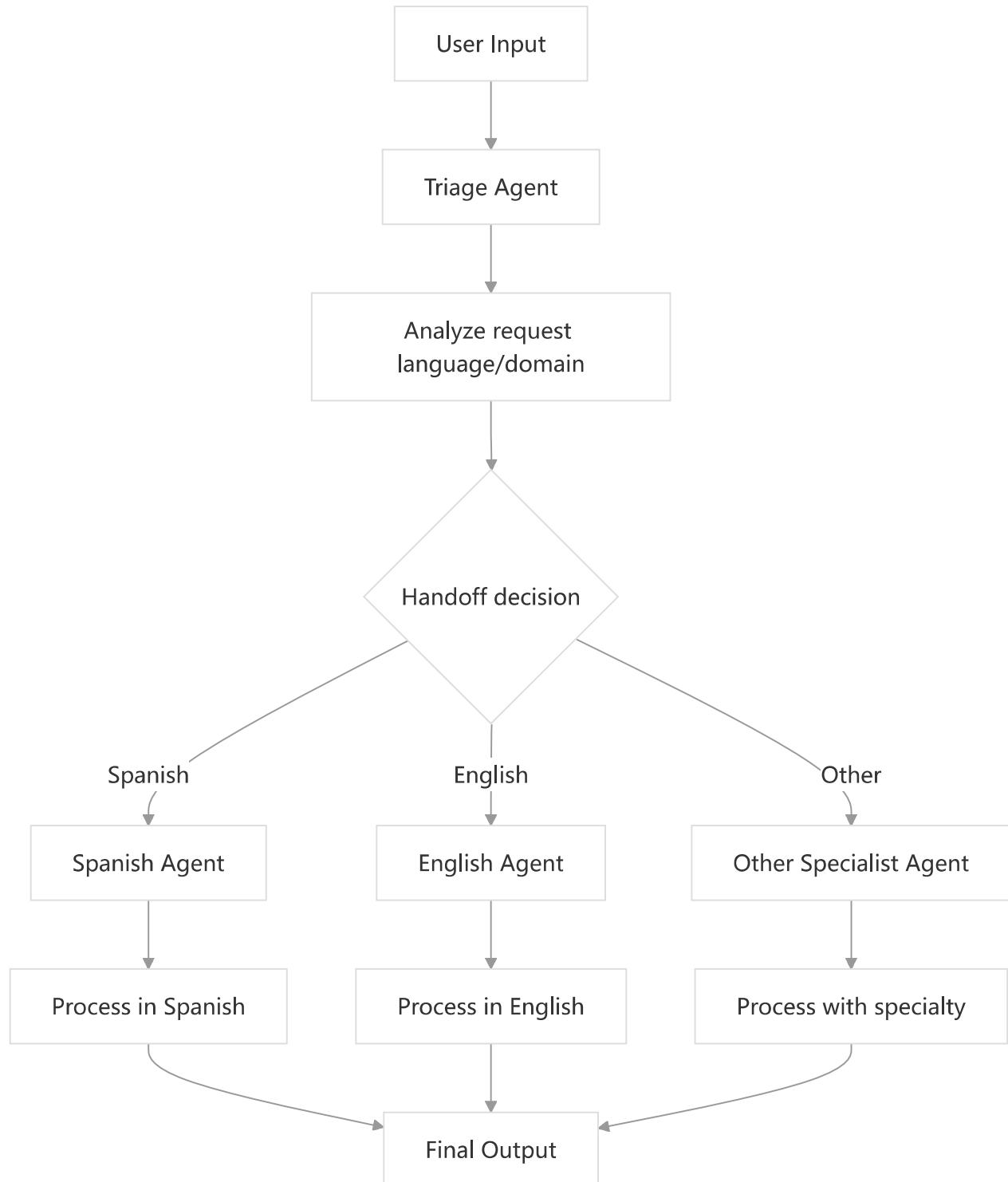
- `Agent.handoffs` 列表定义可用的切换目标
- 分诊人员确定合适的专家
- `Runner` 自动管理代理转换

资料来源：自述文件.md | 171-202

多代理编排流程

向 Devin 询问 openai/openai-agents-python

深入研究



编排模式：

- **分类模式：**中央代理路由至专家
- **顺序模式：**线性切换链

向 Devin 询问 openai/openai-agents-python

深入研究

Jupyter Notebook 模式

对于具有现有事件循环的环境（如 Jupyter 笔记本）：

```
# Use await directly without asyncio.run()
result = await Runner.run(agent, "Write a haiku about recursion in programming.")
```

Jupyter 注意事项：

- 现有的事件循环允许直接 `await` 使用
- 无需 `asyncio.run()` 包装
- `Runner.run_sync()` 也可用于同步执行

资料来源：示例/基本/hello_world_jupyter.ipynb | 27-34

环境配置模式

```
import os

# Set OpenAI API key
os.environ["OPENAI_API_KEY"] = "your-api-key"

# Optional: Configure other environment variables
os.environ["OPENAI_MODEL"] = "gpt-4o"
```

配置要求：

- `OPENAI_API_KEY` 需要环境变量
- 可选模型和提供商配置
- SDK 自动处理身份验证

资料来源：自述文件.md | 167

异步/同步执行模式

```
# Asynchronous pattern
async def example():
    pass
```

向 Devin 询问 openai/openai-agents-python

深入研究

```
result = Runner.run_sync(agent, "Hello")
return result

# Mixed pattern with asyncio
import asyncio

def main():
    result = asyncio.run(Runner.run(agent, "Hello"))
    return result
```

执行模式选择：

- `Runner.run()`：异步，需要 `await`
- `Runner.run_sync()`：同步，阻塞直至完成
- 根据应用程序架构和性能需求进行选择

资料来源：自述文件.md | 38-60

向 Devin 询问 openai/openai-agents-python

深入研究