



本章主要介绍大模型应用演化形态、智能体核心定义、四大基础组成、主流分类形态，同时介绍本书开发技术栈与本地运行环境，为后续智能体的开发奠定基础。

1.1 从大模型到 AI Agent

本节我们从大模型到AI Agent的演进逻辑入手，帮助初学者建立对AI Agent的基础认识，主要包括：大模型的应用形态、AI Agent的定义、与普通大模型调用的区别，以及AI Agent可解决各类实际问题等内容。

1.1.1 大模型应用的基本形态

大模型作为新一代人工智能技术核心，正在重塑软件应用的开发方式。围绕语言理解与生成能力，逐渐形成了多种典型应用形态。从简单问答到复杂任务执行，不同形态在能力边界、系统结构以及工程实现上存在明显差异，理解这些基本形态，有助于建立清晰的智能体开发认知基础。

1. 对话式应用形态

对话式应用是最早也是最常见的大模型应用形态，核心特征是以自然语言作为主要交互方式，通过连续对话完成信息获取与问题解答。这类系统通常围绕输入理解、上下文管理和输出生成展开，强调语言流畅性与回答相关性。典型的对话式模型应用如DeepSeek，所谓的对话型就是区别于API调用的一种交互模式，如图1-1所示。

在工程实现上，对话式应用重点在于Prompt设计、对话历史管理以及上下文窗口控制，以保证多轮交互中的一致性与连贯性。此类应用适用于知识问答、学习辅导、文本创作等场景，但其能力主要停留在“生成内容”，缺乏对外部环境的直接操作能力，因此在复杂任务处理上存在一定的局限。



图 1-1 交互式 AI 代表 DeepSeek

2. 工具增强型应用形态

随着应用需求的提升，单纯依赖模型生成已难以满足实际业务需求，引入工具调用机制成为重要发展方向。工具增强型应用在大模型基础上增加了外部能力接口，如计算、数据查询、文件处理或业务系统调用，使模型不仅能“说”，还可以“做”。扣子就是典型的工具增强型应用，如图1-2所示。



图 1-2 工具增强型应用代表扣子 Agent

在这种形态下，模型需要根据任务目标选择合适的工具或技能，并构造参数完成调用，随后结合返回结果生成最终输出。工程实现中涉及工具描述设计、参数结构定义以及调用流程控制等关键环节。这类应用广泛用于数据处理、自动化办公和业务辅助系统，显著扩展了大模型的实际应用能力。

3. 任务驱动型应用形态

任务驱动型应用进一步将大模型嵌入完整的软件执行流程中，以“完成目标”为核心，而不仅是生成文本。这类系统通常具备任务拆解、步骤执行、状态跟踪和结果反馈等能力，通过多轮推理与执行逐步推进任务完成。

相比前两种形态，任务驱动型应用更加关注过程控制与执行逻辑，需要明确任务结构、执行顺序以及终止条件。在工程实现上，常结合 workflow 设计或状态机机制，对任务进行分阶段处理，并在每一步中调用模型或工具完成具体操作。该形态为后续发展为自主智能体奠定基础，是从“语言能力”走向“行动能力”的关键阶段。

1.1.2 AI Agent 的基本定义

AI Agent 通常指一种能够围绕特定目标进行感知、理解、推理、规划和行动的软件系统。它以大模型为核心，但并不等同于大模型本身，而是在模型能力之外加入工具调用、状态管理、记忆机制和任务执行流程，使系统能够从“生成回答”进一步走向“完成任务”。

1. AI Agent 的基本定义

AI Agent 可以理解为“具备一定自主决策能力的智能任务执行单元”。它接收用户输入或外部事件后，会先理解任务目标，再根据当前上下文判断下一步动作，必要时调用外部工具或访问数据资源，最后根据执行结果生成反馈或继续推进任务。

从工程角度看，AI Agent 通常包含四个关键环节：第一是理解目标，明确要解决什么问题；第二是制定步骤，把复杂任务拆成若干可执行动作；第三是调用工具，例如查询数据库、读取文件、访问接口或运行代码；第四是观察结果，根据工具返回的信息继续判断是否完成任务。正因为有这些环节，AI Agent 不只是一个回答问题的模型，而是一个由模型、工具、数据和控制逻辑组成的应用系统。

2. AI Agent 与普通大模型调用的区别

普通大模型调用一般是“输入问题，输出答案”。例如输入“帮我写一段活动通知”，模型直接生成一段文字。这个过程没有复杂任务状态，也不需要调用外部工具，主要依赖模型自身的语言生成能力。

AI Agent 则更强调任务过程。例如输入“帮我整理本周会议纪要，并生成待办事项清单”，Agent 可能需要先读取会议文本，再识别关键议题，提取决策内容，整理负责人和截止时间，最后输出结构化待办事项。如果会议文本来自文件系统，Agent 还需要调用文件读取工具；如果待办事项需要写入任务管理系统，还需要调用对应接口。

因此，AI Agent 的核心区别在于：它不是只给出一句回答，而是能够围绕目标组织一系列动作。简单来说，普通大模型更像“语言生成器”，AI Agent 更像“带工具和流程控制的智能执行器”。

3. 结合例子理解AI Agent

以“个人学习助手Agent”为例，用户输入：

“帮我制订一个三周Python学习计划，每天学习两小时，重点学习数据分析。”

普通聊天机器人可能会直接生成一份学习计划。而AI Agent可以进一步完成更完整的流程：先分析学习目标，判断需要覆盖Python基础、NumPy、Pandas、数据清洗和可视化等内容；再根据三周时间拆分每日任务；然后生成每天的学习主题、练习内容和检查标准；如果系统接入了记忆模块，还可以结合用户之前的学习记录调整难度；如果接入文件工具，还可以把计划保存为本地文档。

再比如“知识库问答Agent”，用户上传一份课程资料后提问：

“这份资料里关于向量数据库的部分主要讲了什么？”

Agent会先解析文档，将内容切分并建立索引；收到问题后，检索相关片段；再把检索结果交给大模型组织回答；最后给出答案和引用来源。这里的大模型并不是凭空回答，而是基于外部资料完成信息检索和内容生成。

由此可以看出，AI Agent的本质是把大模型放入一个可执行的软件系统中，让它具备理解目标、调用能力、处理状态和完成任务的能力。

1.1.3 AI Agent 能解决哪些问题

AI Agent的价值并不体现在单一功能上，而是在于将语言理解、工具调用与任务执行结合起来，从而能够处理更复杂的实际问题。从信息获取到任务执行，再到流程自动化，AI Agent正在逐步改变软件系统解决问题的方式。理解其适用范围，有助于明确应用边界与设计思路。

1. 信息处理与知识获取类问题

AI Agent能够高效处理大规模信息，并从中提取有价值的内容。这类问题通常涉及文本理解、内容总结、资料整理和知识问答等场景。例如在学习、科研或企业内部知识管理中，大量文档难以快速查阅，AI Agent可以通过检索和分析相关内容，生成结构化总结或直接回答问题。

相比传统搜索方式，AI Agent不仅返回信息，还能够对信息进行归纳与解释，从而降低理解成本。在工程实现上，通常结合知识库与检索机制，使系统具备从外部数据中获取信息并生成答案的能力，适用于文档问答、资料分析和知识辅助决策等场景。

2. 多步骤任务执行类问题

AI Agent擅长处理需要分步骤完成的复杂任务，这类问题往往不能通过一次回答解决，而是需要多个环节协同完成。例如生成一份报告，不仅需要理解主题，还需要收集资料、整理结构、逐步生成内容并进行修改。

AI Agent可以将目标拆解为若干子任务，按照一定顺序执行，并在每一步中结合工具或外部数

据完成具体操作。在这个过程中，系统需要记录执行状态，并根据中间结果动态调整后续步骤。相比简单对话系统，这类能力更接近真实工作流程，因此适用于自动化办公、数据处理、内容生成以及项目辅助等场景。

3. 业务流程自动化与系统协同类问题

在实际业务系统中，很多任务涉及多个系统之间的数据流转与操作协同，例如查询数据库、调用接口、生成文件或更新记录。AI Agent可以作为统一的智能入口，通过自然语言理解用户需求，再调用后端服务完成具体操作，从而实现流程自动化。

例如，在企业系统中，可以根据用户指令查询订单状态、生成报表、更新数据或执行审批流程。与传统自动化脚本相比，AI Agent具备更强的灵活性，可以根据不同输入动态选择执行路径。在工程实践中，这类应用通常需要结合权限控制、工具管理和异常处理机制，以保证系统运行的稳定性与安全性。

1.2 AI Agent 的基本组成

AI Agent并非单一模块，而是由多个功能单元协同构成。模型负责理解与推理，工具提供执行能力，记忆承载上下文信息，规划机制组织任务流程。各部分相互配合，形成完整的任务处理与执行体系。一个Agent的基本组成如图1-3所示。



图 1-3 Agent 的基本组成

1.2.1 大模型：理解、生成与推理核心

大模型是AI Agent系统中的核心组件，承担着信息理解、内容生成与决策推理等关键任务。不同于传统程序中固定逻辑的处理方式，大模型通过语言建模能力，将输入信息转化为语义表示，并结合上下文进行动态判断。在智能体运行过程中，大模型既是信息处理中心，也是决策驱动核心，其能力直接决定了智能体的整体表现。

1. 语言理解能力：从文本到语义的转化

大模型首先承担的是对输入信息的理解任务，其核心在于将自然语言表达转化为可处理的语义结构。面对用户输入，大模型不仅识别字面含义，还能够结合上下文推断真实意图。例如，同一句话在不同语境下可能表达不同需求，大模型可以通过上下文信息进行区分，从而避免误解。

在多轮对话中，这种能力尤为重要，需要持续跟踪对话历史并保持语义一致性。此外，大模型还能够处理不完整表达、口语化语句以及带有歧义的输入，通过语义补全与推断提高理解准确性。这种理解能力为后续的任务规划与执行提供了基础。

2. 内容生成能力：从语义理解到结构化输出

在完成理解之后，大模型需要根据任务要求生成输出结果。这种生成能力不仅体现在语言流畅性上，更体现在结构组织与信息表达上。例如，在生成报告、总结内容或回答问题时，大模型能够按照一定逻辑组织信息，使输出具有条理性与可读性。

在工程实践中，生成能力通常结合提示设计进行控制，通过明确任务描述、约束条件与输出格式，使模型输出符合预期。例如，可以要求生成列表、表格描述或结构化数据，从而方便后续处理。高质量的生成能力使大模型不仅可以“说清楚”，还可以“说得规范”，成为智能体输出结果的重要基础。

3. 推理与决策能力：驱动任务执行过程

推理能力使大模型能够在复杂任务中进行步骤分析与决策判断，这是AI Agent区别于普通文本生成系统的重要特征之一。

在任务执行过程中，大模型需要根据当前状态判断下一步动作，例如是否需要调用工具、选择哪种处理方式以及何时结束任务。

这种能力通常表现为对问题的分解、对不同方案的比较以及对执行结果的评估。例如，在处理多步骤任务时，大模型可以逐步分析问题，将复杂目标拆解为多个子任务，并在每一步中进行合理决策。

推理能力并不等同于严格的逻辑推导，但在实际应用中已经能够支持较为复杂的任务流程，是驱动智能体运行的关键动力。

1.2.2 工具：让 Agent 具备行动能力

在AI Agent系统中，仅依赖大模型的语言能力难以完成真实世界中的复杂任务。工具机制的引入，使智能体能够访问外部资源、调用具体功能，从而将“理解与生成”延伸为“执行与操作”。通过工具，智能体可以与文件系统、数据库、网络服务等环境进行交互，真正具备行动能力。

1. 工具的作用与基本形式

工具本质上是对外部能力的封装，可以是函数、接口或服务调用，用于完成模型自身无法直接处理的操作。例如，计算数学表达式、查询数据库、读取本地文件或访问外部接口等。这些操作通常需要精确执行，而不是依赖语言生成，因此必须通过工具实现。

在系统设计中，每个工具通常包含名称、功能描述和参数定义，使大模型能够理解其用途并正确调用。通过这种方式，模型可以在推理过程中选择合适的工具，将任务从“生成答案”转变为“执行操作并返回结果”。

2. 工具调用的工作机制

工具调用通常由大模型根据当前任务状态进行决策。当模型判断某一步需要外部能力时，会输出调用指令，包括工具名称及对应参数。系统接收到该指令后执行具体函数或接口，并将返回结果重新交给模型进行处理。

整个过程形成“判断—调用—返回—再判断”的循环。例如，在处理数据查询任务时，模型先分析问题，再调用数据库工具获取数据，随后根据查询结果生成最终回答。在这一过程中，工具调用并非一次性完成，而是可能多次迭代，从而逐步推进任务执行。该机制使智能体具备动态决策与执行能力。

3. 工具对智能体能力边界的扩展

工具机制极大地扩展了AI Agent的能力范围，使其能够参与更复杂的应用场景。通过接入不同类型的工具，智能体可以完成跨系统操作，例如读取文件后进行分析，再将结果写入数据库或生成报告。

工具的组合使用还可以构建更复杂的任务流程，例如先检索信息，再进行计算，最后输出结构化结果。在工程实践中，工具不仅能够提升功能能力，也引入了新的设计要求，例如参数校验、异常处理和权限控制等。可以将工具视为智能体的“手脚”，负责具体执行操作，而大模型负责决策与调度，两者结合形成完整的任务执行体系。

1.2.3 记忆：保存上下文和历史信息

在AI Agent系统中，单次输入与输出往往难以支撑连续任务处理，记忆机制用于保存上下文与历史信息，使智能体具备连续理解与长期交互能力。通过对历史数据的记录与利用，智能体能够在多轮对话与复杂任务中保持一致性，从而避免重复计算与信息丢失。

1. 上下文记忆：支撑多轮对话理解

上下文记忆主要用于保存当前对话中的历史信息，使智能体能够在多轮交互中保持语义连贯。例如，在连续提问过程中，后续问题往往依赖前文内容，如果没有上下文支持，系统只能按单轮输入处理，容易导致理解偏差。

通过维护消息列表或对话历史，智能体可以在每次处理时参考已有信息，从而准确识别用户意图。在工程实现中，上下文记忆通常以结构化形式存储，并通过裁剪或筛选控制长度，以避免上下文过长影响性能。这类记忆强调短期信息管理，是实现自然对话的基础。

2. 长期记忆：积累用户与任务信息

长期记忆用于保存跨会话的信息，使智能体能够在不同时间点持续利用已有数据。例如，用户偏好、历史行为记录或已完成任务结果，都可以作为长期记忆的一部分。在实际应用中，这类记忆通常存储在文件或数据库中，通过查询与更新机制进行管理。

长期记忆不仅可以提升个性化体验，还可以减少重复输入，提高任务执行效率。例如，学习助手可以记录学习进度，从而在后续交互中自动调整内容难度。相比上下文记忆，长期记忆更强调信息积累与持续使用。

3. 记忆在任务执行中的作用

记忆机制不仅用于对话，还在任务执行过程中发挥关键作用。在多步骤任务中，智能体需要记录当前进度、已完成步骤以及中间结果，以便在后续操作中进行参考。例如，在数据处理或报告生成过程中，前一步的结果往往直接影响下一步决策。

如果缺乏记忆支持，系统需要重复计算或重新分析，效率较低。通过合理设计记忆结构，智能体可以实现状态跟踪、结果复用和错误恢复，从而提升整体执行能力。在工程实践中，记忆通常与任务状态管理结合，成为智能体运行的重要组成部分。

1.2.4 规划：把目标拆解成步骤

规划能力是AI Agent从“被动响应”走向“主动执行”的关键。面对复杂目标时，系统不能只生成一次性回答，而需要判断任务应如何展开、先做什么、后做什么以及何时结束。通过规划机制，智能体可以将抽象目标拆解为清晰步骤，使后续的工具调用和任务执行更有条理。

1. 目标理解：明确任务要达成的结果

规划的起点是目标理解，也就是判断任务最终需要得到什么结果。用户输入往往具有一定概括性，例如“整理一份学习计划”“分析一份文档”“生成一份调研报告”等，这些表达并没有直接给出完整的执行步骤。

智能体需要先识别任务类型、输出形式、限制条件和关键需求，再确定后续处理方向。目标理解不只是提取关键词，还要明确任务边界，例如是否需要读取文件、是否需要调用工具、是否需要生成结构化结果。只有目标清晰，后续规划才不会偏离方向。

2. 步骤拆解：将复杂任务转化为可执行流程

复杂任务通常无法一次性完成，需要被拆分成多个相对独立的子任务。以生成调研报告为例，智能体可以先确定主题，再收集资料，随后整理大纲、撰写内容、检查格式并输出结果。每个步骤都应有明确的输入和输出，便于系统逐步执行。

在工程实现中，步骤拆解可以由Prompt引导模型完成，也可以通过 workflows 或状态机提前定义。合理的步骤拆解能够降低任务复杂度，使智能体在每个阶段只处理一个较小的问题，从而提高执行稳定性和结果可控性。

3. 执行控制：根据反馈调整后续动作

规划并不是一次生成后固定不变的列表，在真实任务中，执行结果可能与预期不同，因此需

要根据反馈调整后续动作。例如，检索资料不足时，需要重新改写查询；工具调用失败时，需要重试或切换方案；生成结果不符合要求时，需要进入修改流程。

执行控制要求智能体记录当前步骤、判断完成状态，并决定是否继续、回退或终止。通过这种动态调整机制，规划能力不再只是“列步骤”，而是成为连接目标、工具、状态和反馈的控制核心，使智能体能够更稳定地完成多步骤任务。

1.3 AI Agent 的常见类型

随着应用场景的拓展，AI Agent呈现出多种形态。单智能体适用于基础任务处理，工具增强型智能体具备外部能力调用，知识库增强型智能体面向信息检索，多智能体与自主智能体则承担更复杂的协作与决策任务，如图1-4所示。

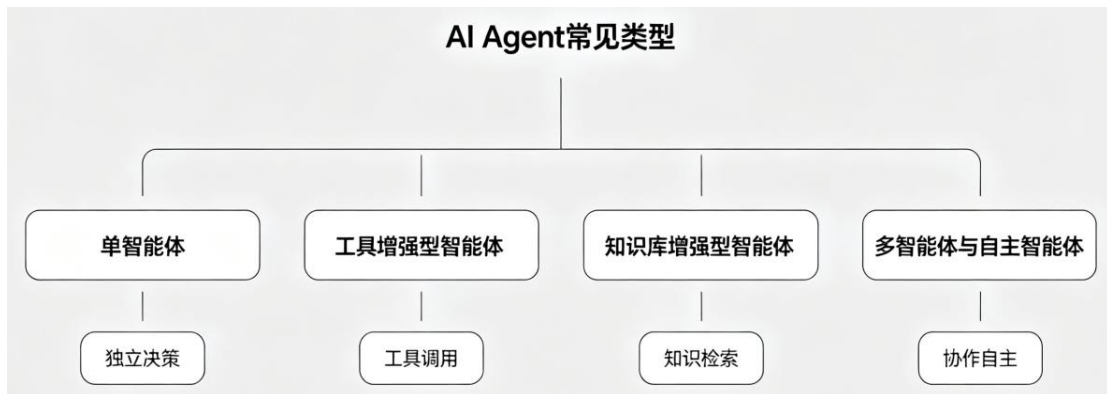


图 1-4 智能体的常见类型

1.3.1 单智能体

单智能体是AI Agent系统中最基础的形态，指由一个智能体独立接收任务、理解需求、调用模型或工具，并生成最终结果的应用结构。在这种模式下，系统中通常只有一个主要决策单元，所有任务处理逻辑都围绕该智能体展开。它可以完成问答、文本生成、文件处理、简单数据查询、学习计划生成等任务，是入门AI Agent开发时最适合首先掌握的类型。

单智能体的优势在于结构清晰、实现简单、调试方便。开发过程中只需要关注一个智能体的输入、提示词、工具列表、记忆方式和输出格式，不需要处理多个智能体之间的协作、通信和冲突问题。因此，它非常适合用于构建个人助理、文档问答助手、写作助手、学习助手等轻量级应用。

从工程角度看，单智能体通常包括用户输入、模型调用、工具调用、上下文管理和结果输出几个部分。如果任务较简单，智能体可以直接调用大模型生成回答；如果任务需要外部能力，则可以调用预先注册的工具，例如读取文件、查询数据或进行计算。对于初学者来说，单智能体能够帮

助理解AI Agent的基本运行流程，为后续学习工具增强型智能体、知识库增强型智能体和多智能体系统打下基础。

1.3.2 工具增强型智能体

工具增强型智能体是在单智能体基础上加入工具调用能力的一类Agent。普通单智能体主要依赖大模型完成理解和生成，而工具增强型智能体可以在需要时调用外部函数、接口或服务，从而完成模型本身无法可靠完成的操作。例如计算、查询时间、读取文件、检索数据库、调用业务接口等，都可以通过工具机制交给程序执行，再由大模型根据工具结果组织最终回答。

这类智能体的核心特点是“模型负责判断，工具负责执行”。大模型根据用户需求判断是否需要使用工具，并生成工具调用参数；系统接收到调用请求后执行对应工具，再把执行结果返回给模型。通过这种方式，智能体不再只是生成文本，而是能够连接真实数据和外部系统，具备一定行动能力。

在工程开发中，工具增强型智能体需要重点设计工具名称、功能描述、参数结构和返回格式。工具描述越清晰，模型越容易正确选择工具；参数校验越严格，系统越不容易出现错误调用。同时，还需要处理工具调用失败、超时、权限限制和调用次数控制等问题。个人助理Agent、数据查询Agent、自动化办公Agent都属于典型应用。工具增强型智能体是从“会回答”走向“能操作”的关键阶段，也是后续RAG、多智能体和自主智能体开发的重要基础。

1.3.3 知识库增强型智能体

知识库增强型智能体是在大模型基础上接入外部知识库的一类Agent，主要用于解决大模型无法准确掌握私有资料、实时资料或专业资料的问题。普通大模型虽然具备较强的语言理解和生成能力，但其回答主要依赖训练阶段形成的知识，面对企业文档、课程资料、产品手册、论文材料等具体内容时，容易出现信息不准确或凭空生成的问题。知识库增强型智能体通过检索外部资料，再结合大模型生成回答，可以显著提升答案的可靠性。

这类智能体的核心流程通常包括文档解析、文本切分、向量化存储、相似度检索和答案生成。用户提出问题后，系统会先从知识库中查找相关片段，再把检索结果作为上下文提供给大模型，由模型基于这些资料组织回答。这样生成的内容不再完全依赖模型记忆，而是有明确的信息来源。

在工程应用中，知识库增强型智能体常用于PDF文档问答、企业制度查询、课程资料学习、技术文档助手和产品知识库客服等场景。开发时需要重点关注文档切分粒度、检索准确率、引用来源显示和回答不确定性的处理。它的价值不只是让模型“知道更多”，而是让模型能够围绕指定资料进行回答，是构建可信AI应用的重要基础。

1.3.4 多智能体与自主智能体

多智能体与自主智能体属于更高阶的Agent形态，主要用于处理单个智能体难以独立完成的复

杂任务。多智能体强调角色分工与协作，通过设置不同职责的智能体，让系统分别完成资料收集、内容分析、结果审查、方案优化等工作。例如，在调研报告生成任务中，可以设置资料收集Agent、分析写作Agent和审稿优化Agent，使任务流程更加清晰，输出结果也更容易控制。

自主智能体则更强调目标驱动和连续执行能力。与普通Agent相比，自主智能体不是只完成一次回答或一次工具调用，而是能够根据目标自行拆解步骤，执行任务，观察结果，并在失败或信息不足时调整后续行动。其典型运行过程包括目标理解、计划生成、工具选择、状态记录、结果观察和任务终止判断。

在实际开发中，多智能体和自主智能体常常结合使用。多智能体提供专业分工，自主机制提供持续推进能力。但这类系统的复杂度也更高，需要处理任务分配、状态同步、结果汇总、冲突解决和执行边界控制等问题。因此，开发时通常应先从单智能体、工具增强型智能体和知识库增强型智能体入手，再逐步引入多角色协作和自主规划机制。

1.4 本书技术栈与运行环境说明

AI Agent开发依赖稳定的开发环境与清晰的学习路径。合理选择技术栈、规范项目结构，并在单机环境中完成案例实践，是理解智能体运行机制与掌握开发方法的重要前提。

1.4.1 AI Agent 开发技术栈速览

AI Agent开发并不是单纯调用一个大模型接口，而是由语言模型、工具函数、数据存储、接口服务、前端交互和部署环境共同组成的工程体系。为保证后续代码能够在普通个人计算机上复现，技术栈以轻量、通用、易安装为原则，优先选择Python生态中成熟稳定的工具，并通过命令行、接口请求和本地服务完成完整开发流程。本部分仅是本书的技术栈速览，更详细的过程可以参考第2章进行学习。

本书主要使用Python作为核心开发语言。Python语法简洁，第三方库丰富，适合完成大模型调用、工具封装、RAG知识库、工作流编排和Web服务开发等任务。推荐使用Python 3.11或Python 3.12，二者兼顾兼容性与运行性能。注意，书中命令均在cmd或PowerShell中运行，如图1-5所示。

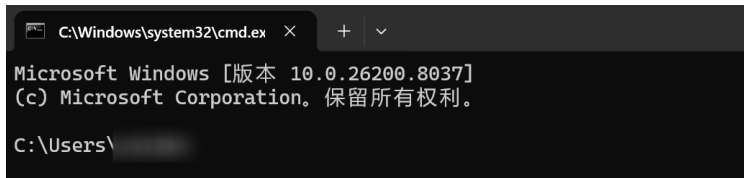


图 1-5 cmd 命令行

Windows PowerShell如图1-6所示。

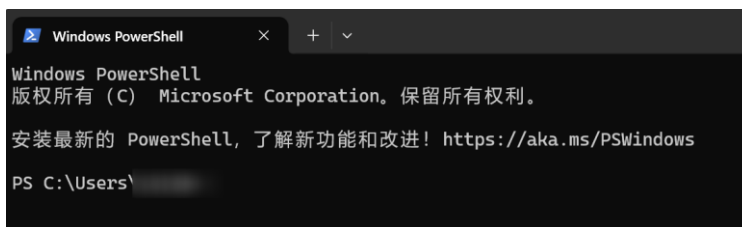


图 1-6 Windows PowerShell 命令行

在终端中检查Python版本：

```
python --version
```

正常情况下，可以看到如下输出：

```
Python 3.11.9
```

如果系统中同时存在多个Python版本，可以使用：

```
python3 --version
```

项目依赖管理使用uv或pip均可。uv安装速度较快，适合后续章节频繁创建虚拟环境和安装依赖。安装uv后，可以创建项目目录并初始化环境：

```
mkdir ai-agent-book
cd ai-agent-book
uv init
uv venv
```

激活虚拟环境：

```
# Windows PowerShell
.venv\Scripts\activate

# macOS / Linux
source .venv/bin/activate
```

激活成功后，命令行前面通常会出现：

```
(.venv)
```

venv目录如图1-7所示。

安装本书常用依赖：

```
uv add openai python-dotenv pydantic fastapi
uvicorn requests rich
```

安装完成后，可以查看当前依赖：

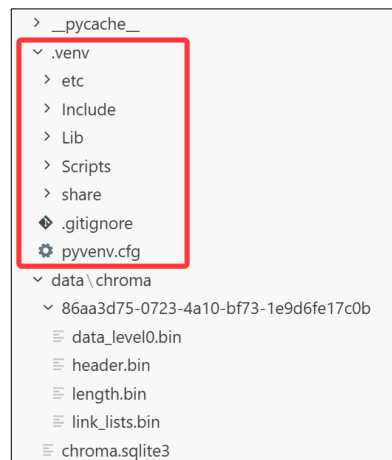


图 1-7 venv 目录

```
uv pip list
```

输出结果中会包含如下内容：

```
openai
python-dotenv
pydantic
fastapi
uvicorn
requests
rich
```

大模型接口调用使用OpenAI风格SDK。该类SDK既可以连接官方模型服务，也可以在兼容接口中使用其他模型服务。在项目根目录下创建.env文件，用于保存接口密钥和模型配置：

```
OPENAI_API_KEY=替换为实际密钥
OPENAI_BASE_URL=https://api.openai.com/v1
OPENAI_MODEL=gpt-4.1-mini
```

随后创建main.py文件，完成第一次SDK调用：

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)
response = client.chat.completions.create(
    model=os.getenv("OPENAI_MODEL"),
    messages=[
        {"role": "system", "content": "你是一个严谨的AI Agent开发助手。"},
        {"role": "user", "content": "用一句话解释什么是AI Agent。"}
    ],
    temperature=0.3
)

print(response.choices[0].message.content)
```

运行程序：

```
python main.py
```

可以得到如下输出：

AI Agent是一种能够理解目标、调用工具并按照步骤完成任务的智能软件系统。

除SDK调用外，后续还会涉及HTTP接口方式。使用curl可以直接向模型服务提交JSON请求，这有助于理解SDK底层实际发送的数据结构：

```
curl https://api.openai.com/v1/chat/completions ^
-H "Authorization: Bearer %OPENAI_API_KEY%" ^
-H "Content-Type: application/json" ^
-d "{\"model\": \"gpt-4.1-mini\", \"messages\": [{\"role\": \"user\",
\"content\": \"解释AI Agent的作用\"}]}"
```

在macOS或Linux中可以写成：

```
curl https://api.openai.com/v1/chat/completions \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "model": "gpt-4.1-mini",
  "messages": [
    {
      "role": "user",
      "content": "解释AI Agent的作用"
    }
  ]
}'
```

返回结果通常是JSON格式，其中核心内容位于choices字段中：

```
{
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "AI Agent可以根据目标理解任务、调用工具、处理信息并生成结果。"
      }
    }
  ]
}
```

后端服务使用FastAPI。FastAPI适合构建轻量级AI应用接口，可以把Agent封装成HTTP服务，供Web页面、命令行工具或其他系统调用。创建api_server.py文件：

```
import os
from dotenv import load_dotenv
from fastapi import FastAPI
from pydantic import BaseModel
from openai import OpenAI

load_dotenv()
app = FastAPI(title="AI Agent Service")
```

```

client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)

class ChatRequest(BaseModel):
    message: str
class ChatResponse(BaseModel):
    answer: str

@app.post("/chat", response_model=ChatResponse)
def chat(req: ChatRequest):
    response = client.chat.completions.create(
        model=os.getenv("OPENAI_MODEL"),
        messages=[
            {"role": "system", "content": "你是一个AI Agent开发助手。"},
            {"role": "user", "content": req.message}
        ],
        temperature=0.3
    )
    return ChatResponse(answer=response.choices[0].message.content)

```

启动服务:

```
uvicorn api_server:app --reload --host 127.0.0.1 --port 8000
```

终端中会出现如下信息:

```

Uvicorn running on http://127.0.0.1:8000
Application startup complete.

```

通过JSON提交请求:

```

curl -X POST http://127.0.0.1:8000/chat \
-H "Content-Type: application/json" \
-d '{"message": "AI Agent和普通聊天机器人有什么区别? "}'

```

返回结果如下:

```

{
  "answer": "普通聊天机器人主要回答问题，而AI Agent可以围绕目标调用工具、管理状态并执行多步骤任务。"
}

```

数据校验使用Pydantic。后续章节中，工具参数、Agent输入、结构化输出、任务状态都会通过Pydantic定义数据结构。下面的代码展示了一个任务请求结构:

```

from pydantic import BaseModel, Field

class AgentTask(BaseModel):
    task_name: str = Field(description="任务名称")

```

```

    user_goal: str = Field(description="用户目标")
    max_steps: int = Field(default=5, description="最大执行步数")

    task = AgentTask(
        task_name="学习计划生成",
        user_goal="生成一份Python入门学习计划",
        max_steps=3
    )

    print(task.model_dump())

```

运行结果：

```

{'task_name': '学习计划生成', 'user_goal': '生成一份Python入门学习计划',
'max_steps': 3}

```

知识库部分使用本地向量库。入门阶段优先使用Chroma或FAISS，避免引入复杂数据库服务。

安装Chroma：

```
uv add chromadb
```

创建一个简单的本地知识库：

```

import chromadb

client = chromadb.PersistentClient(path="./data/chroma")
collection = client.get_or_create_collection(name="book_notes")
collection.add(
    documents=[
        "AI Agent由大模型、工具、记忆和规划组成。",
        "RAG可以让大模型基于外部文档回答问题。"
    ],
    ids=["doc1", "doc2"]
)
result = collection.query(
    query_texts=["AI Agent由哪些部分组成? "],
    n_results=2
)

print(result["documents"])

```

运行后可以看到如下结果：

```
[[ 'AI Agent由大模型、工具、记忆和规划组成。', 'RAG可以让大模型基于外部文档回答问题。' ]]
```

本地界面使用Streamlit。它能够快速搭建聊天页面、文件上传页面和结果展示页面，适合教学与上机练习。安装依赖：

```
uv add streamlit
```

创建web_app.py文件：

```
import streamlit as st

st.set_page_config(page_title="AI Agent应用", layout="centered")
st.title("AI Agent应用界面")
user_input = st.text_input("请输入任务目标")
if st.button("提交任务"):
    if user_input.strip():
        st.success("任务已接收")
        st.write(f"任务目标: {user_input}")
    else:
        st.warning("请输入有效内容")
```

启动页面:

```
streamlit run web_app.py
```

浏览器会打开本地页面:

```
http://localhost:8501
```

AI Agent服务页面如图1-8所示,页面上会列出当前服务提供的路径,例如POST /chat,写清请求体字段、返回结构。用户可以直接在页面中填写参数、点击Execute发起请求,下方会显示响应状态和返回的JSON,相当于自带的小型接口调试工具。该页面无须再单独安装Postman,就能验证Agent服务是否正常、密钥与模型配置是否生效。

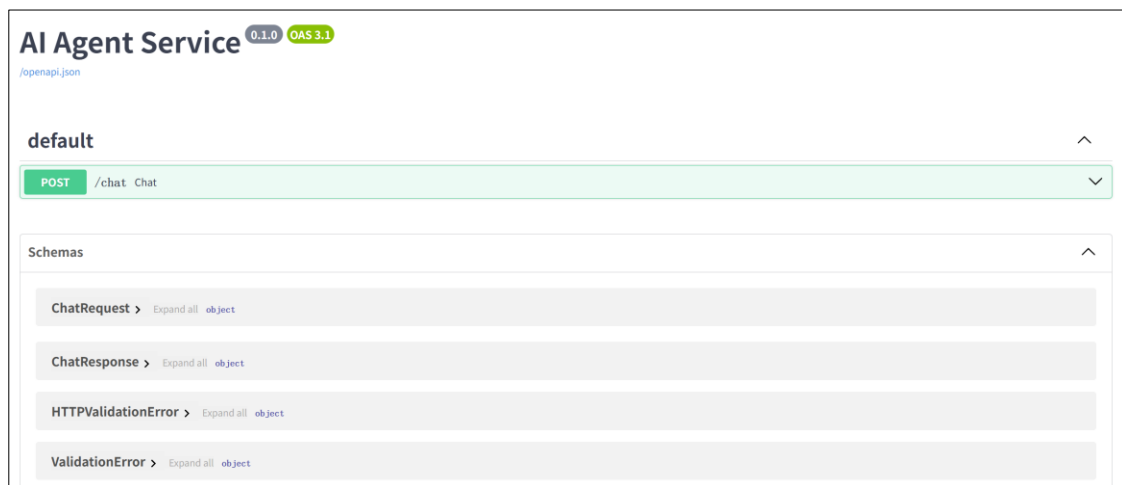


图 1-8 AI Agent 服务页面

点击Execute即可发起请求,如图1-9所示。

Curl响应如图1-10所示。

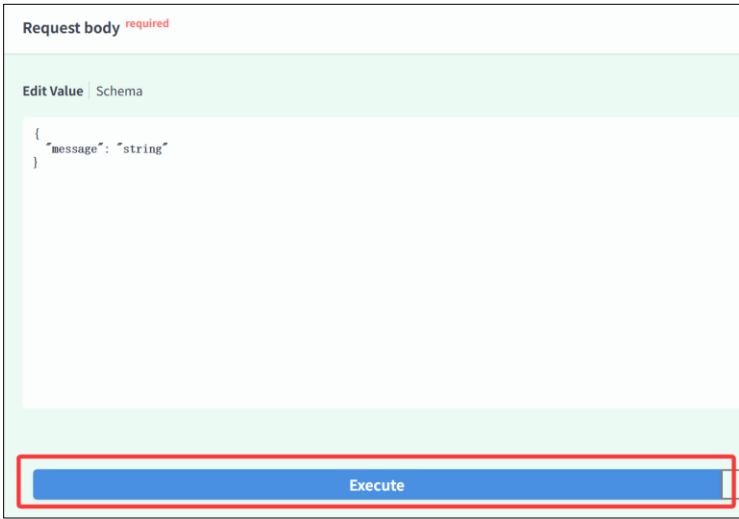


图 1-9 点击 Execute 发起请求

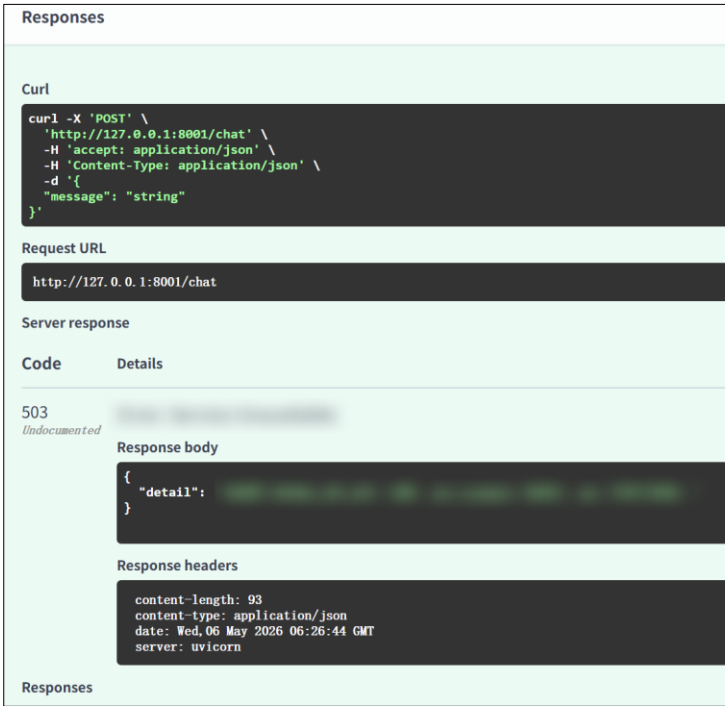


图 1-10 Curl 响应

部署部分使用Docker。Docker用于统一运行环境，避免不同计算机因Python版本、系统路径或依赖差异导致代码无法运行。响应返回和验证结果如图1-11所示。

Code	Description
200	Successful Response
	Media type application/json
	Controls Accept header.
	Example Value Schema
	<pre>{ "answer": "string" }</pre>
422	Validation Error
	Media type application/json
	Example Value Schema
	<pre>{ "detail": [{ "loc": ["string", 0], "msg": "string", "type": "string", "input": "string", "ctx": {} }] }</pre>

图 1-11 响应返回和验证结果

创建Dockerfile:

```
FROM python:3.11-slim  
  
WORKDIR /app  
COPY pyproject.toml .  
COPY . .  
  
RUN pip install uv && uv pip install --system -r requirements.txt  
CMD ["uvicorn", "api_server:app", "--host", "0.0.0.0", "--port", "8000"]
```

如果使用requirements.txt管理依赖，可以生成依赖文件:

```
uv pip freeze > requirements.txt
```

构建镜像:

```
docker build -t ai-agent-service .
```

运行容器:

```
docker run --env-file .env -p 8000:8000 ai-agent-service
```

浏览器访问：

```
http://127.0.0.1:8000/docs
```

可以看到FastAPI自动生成的接口文档页面。日志输出使用rich提升可读性，如图1-12所示。

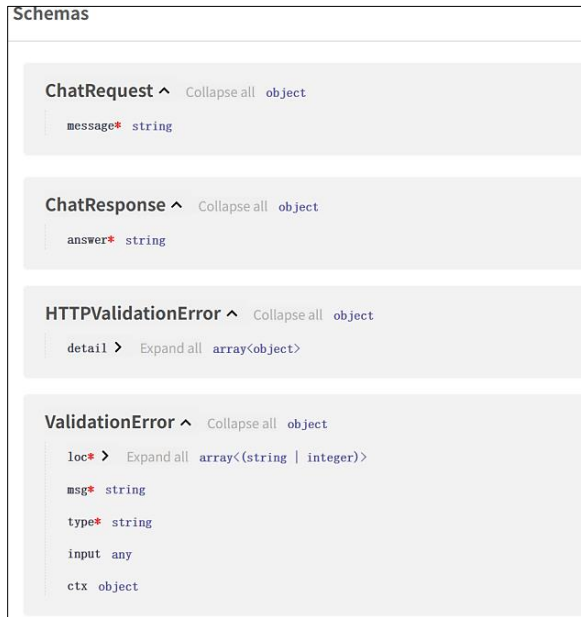


图 1-12 FastAPI 自动生成的接口文档

后续章节中，Agent的模型调用、工具调用、状态变化都会通过日志展示，方便排查问题：

```
from rich.console import Console

console = Console()

console.print("[bold green]Agent启动成功[/bold green]")
console.print("[yellow]正在调用工具: calculator[/yellow]")
console.print("[blue]任务执行完成[/blue]")
```

运行结果会以彩色形式显示在终端中：

```
Agent启动成功
正在调用工具: calculator
任务执行完成
```

由此形成一套完整的入门级技术栈：使用Python编写核心逻辑，使用OpenAI风格SDK调用大模型，使用Pydantic约束输入输出，使用FastAPI封装服务接口，使用Chroma构建本地知识库，使用Streamlit实现交互界面，使用Docker完成单机部署。后续案例都将在这套基础环境上逐步扩展，使每个Agent能力都能通过命令、代码和接口请求得到验证。

1.4.2 单机运行环境说明

AI Agent开发强调可复现与可运行，稳定的单机环境是后续学习与实践的基础。通过在本地搭建统一的软件环境，可以保证代码在不同章节中持续运行，避免因系统差异导致的问题，同时也便于调试、测试与扩展，实现从编写代码到运行服务的完整流程。

本地开发建议使用Python 3.11版本，兼顾稳定性与兼容性。在终端中检查版本：

```
python --version
```

若版本不符合要求，可通过官网下载并安装对应版本。为避免不同项目之间依赖冲突，需要为每个项目创建独立的虚拟环境：

```
python -m venv .venv
```

激活环境：

```
# Windows
.venv\Scripts\activate

# macOS / Linux
source .venv/bin/activate
```

激活成功后，终端会显示.venv标识。随后安装基础依赖：

```
pip install openai python-dotenv fastapi uvicorn pydantic
```

项目中统一使用.env文件管理密钥与配置。在根目录创建文件：

```
OPENAI_API_KEY=替换为实际密钥
OPENAI_BASE_URL=https://api.openai.com/v1
OPENAI_MODEL=gpt-4.1-mini
```

代码中通过如下方式加载配置：

```
from dotenv import load_dotenv
import os

load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
```

本地运行程序：

```
python main.py
```

若使用接口服务，则启动FastAPI：

```
uvicorn api_server:app --reload
```

浏览器访问：

```
http://127.0.0.1:8000/docs
```

即可查看接口并进行测试。所有案例均基于单机运行设计，无须额外服务器或分布式环境，通过本地终端、浏览器和文件系统即可完成开发、调试与验证。这种环境配置方式简单可靠，适合在学习阶段反复实验与功能扩展。

1.5 本章小结

本章围绕AI Agent的基本概念展开，从大模型应用形态入手，说明对话式、工具增强与任务驱动的发展路径；给出了AI Agent的基本定义及典型示例；进一步梳理模型、工具、记忆与规划4个核心组成部分，介绍单智能体、工具增强型智能、知识库增强型智能体以及多智能体与自主智能体等常见类型，并说明学习所需的技术栈与单机运行环境基础。

第 2 章

开发环境搭建与第一个Agent程序



本章将从开发环境入手，详细讲解操作系统、终端、编辑器的选择，Python版本与虚拟环境的配置，以及依赖管理、API Key与环境变量的设置；同时介绍Python项目的基础结构、数据校验、日志输出与异常处理方法，逐步引导读者完成大模型SDK安装、模型调用、命令行聊天程序与流式输出的实现，最后，我们编写并测试一个具备系统提示词和对话历史管理能力的最小Agent程序。

2.1 开发环境准备

一个稳定的开发环境是AI Agent运行的基础，大模型调用、依赖安装、接口服务和本地调试，都依赖统一的软件环境与规范的工具链。本节主要介绍编辑器的选择、Python版本和依赖安装、环境配置等内容。

2.1.1 操作系统、终端与编辑器选择

1. 操作系统选择与开发环境差异

目前AI Agent开发主要以Windows、macOS和Linux三类操作系统为主。其中，Windows适合初学者快速入门，软件安装简单，图形界面友好，大部分开发工具都可以直接使用。macOS由于自带UNIX环境，在命令行操作、Python环境管理以及Docker运行方面体验较好，也是很多AI开发者常用的平台。Linux则更偏向服务器与生产环境，很多云端部署最终都会运行在Linux系统中，因此后续如果涉及部署或服务化开发，也需要逐渐熟悉Linux基本命令。

从学习角度看，只要能够正常安装Python、运行终端命令并启动本地服务，三类系统都可以完成本书中的案例实践。对于Windows系统，建议优先使用Windows Terminal或PowerShell，并配合VS Code进行开发；如果使用macOS或Linux，则可以直接使用系统自带终端完成环境配置与程序运行。

2. 终端工具在AI Agent开发中的作用

很多初学者会把终端理解为“黑色命令窗口”，但在实际开发中，终端是整个AI工程的重要

控制中心。依赖安装、虚拟环境创建、服务启动、日志查看、Docker运行以及接口测试，都需要通过终端完成。可以说，终端是连接代码、运行环境与系统服务的核心入口。

例如，安装Python依赖时，需要输入：

```
pip install openai fastapi uvicorn
```

启动FastAPI服务时，需要执行：

```
uvicorn api_server:app --reload
```

运行Agent程序时，则需要：

```
python main.py
```

终端不仅用于执行命令，还承担调试作用。当程序报错时，错误信息通常会直接输出在终端中，因此阅读日志、定位问题和分析异常，也是开发过程中非常重要的能力。

3. 代码编辑器与开发效率提升

代码编辑器是AI Agent开发中的核心工作平台。当前最推荐的编辑器是Visual Studio Code，也就是常说的VS Code。它轻量、免费、插件丰富，并且对C++、Python、大模型开发和Web服务支持较好，非常适合AI Agent开发学习。VS Code操作界面如图2-1所示，左侧为文件区，下侧为命令行区，右侧为Copilot交互区。本书主要以命令行操作为主，尽可能避免过多的UI交互操作，防止频繁更新UI导致本书可操作性下降。

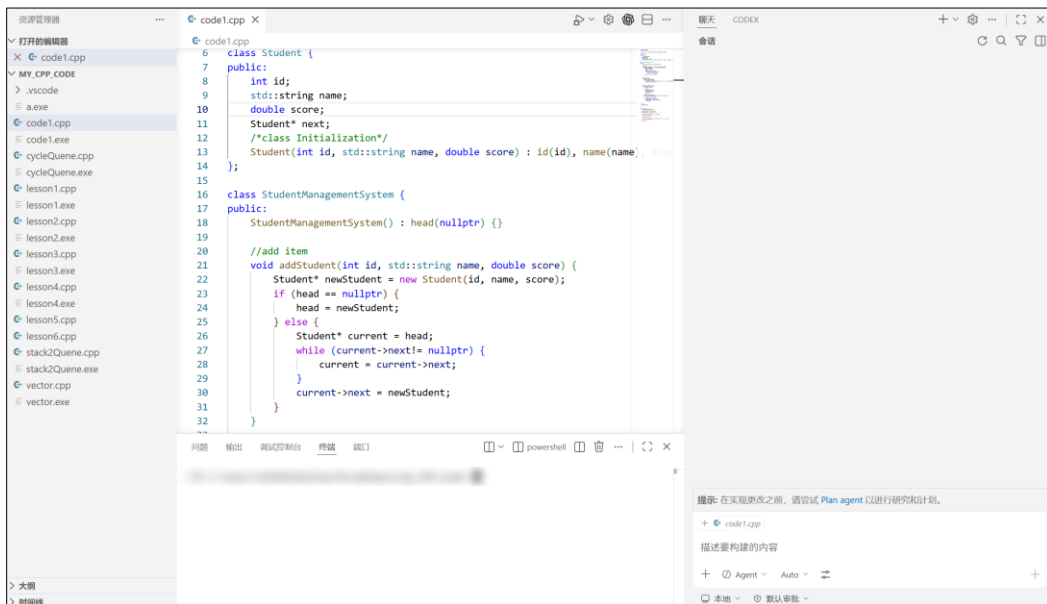


图 2-1 VS Code 操作界面

安装VS Code后，建议同时安装Python插件和Pylance插件，用于语法高亮、自动补全和错误检查。打开项目目录后，编辑器会自动识别Python环境，并在编写代码时给出提示。例如，输入模型调用代码时，编辑器会自动提示函数名称和参数结构，大幅降低代码书写难度。

除此之外，VS Code还内置终端功能，可以直接在编辑器中执行命令、运行程序和查看日志，不需要频繁切换窗口。在后续开发中，还会使用到JSON文件编辑、Markdown文档查看以及Git版本管理等功能，而这些都可以在VS Code中完成。因此，一个稳定、易用的编辑器环境，会显著提升AI Agent开发效率。

2.1.2 Python 版本与虚拟环境配置

Python环境是AI Agent项目运行的基础。模型调用、接口服务、数据处理和工具封装都依赖稳定的Python版本与独立的依赖环境。合理配置Python版本和虚拟环境，可以避免依赖冲突、版本不兼容和系统环境污染，使后续案例能够按照统一方式运行。

1. Python版本选择

建议选择Python 3.11或Python 3.12版本，这两个版本在兼容性、性能和生态支持方面较为稳定，适合大模型SDK、FastAPI、Pydantic、向量数据库等常用工具。版本过低可能导致部分依赖无法安装，版本过新则可能出现个别库尚未完全适配的问题。

安装完成后，可以在终端中检查版本：

```
python --version
```

如果系统提示找不到命令，可以尝试：

```
python3 --version
```

正常输出如下：

```
Python 3.11.9
```

在Windows系统中，安装Python时需要勾选“添加到系统路径”选项，否则终端可能无法识别python命令。macOS和Linux系统通常自带Python，但系统自带版本可能较旧，建议单独安装开发用Python环境，避免影响系统组件。

2. 虚拟环境的作用

虚拟环境用于为每个项目创建独立的Python运行空间。不同项目需要的依赖版本可能不同，如果所有依赖都安装到系统环境中，很容易造成版本冲突。例如，某个项目需要较新的Pydantic，而另一个旧项目依赖旧版本Pydantic，如果共用同一个环境，就可能导致程序运行异常。

创建虚拟环境后，项目依赖会被安装到当前项目目录下，不会影响其他项目。这样做的好处是环境清晰、便于迁移、易于复现，也方便后续生成依赖文件。对于AI Agent项目而言，虚拟环境

尤其重要，因为后续会安装大模型SDK、Web框架、文档解析库、向量数据库客户端等多个依赖包，如果没有隔离机制，环境很快会变得混乱。

在项目目录下创建虚拟环境：

```
python -m venv .venv
```

执行完成后，项目中会出现.venv目录，用于保存该项目独立的Python解释器和依赖包。

3. 虚拟环境激活与验证

虚拟环境创建完成后，需要先激活再安装依赖。在Windows PowerShell中执行：

```
.venv\Scripts\activate
```

在macOS或Linux终端中执行：

```
source .venv/bin/activate
```

激活成功后，终端命令行前面通常会出现如下标识：

```
(.venv)
```

此时再安装依赖，包会进入当前项目的虚拟环境，而不是系统环境。例如，安装基础依赖：

```
pip install openai python-dotenv fastapi uvicorn pydantic
```

安装后可以检查依赖列表：

```
pip list
```

也可以确认当前Python解释器路径：

```
python -c "import sys; print(sys.executable)"
```

输出路径中如果包含.venv，说明当前正在使用项目虚拟环境。完成开发后，可以通过以下命令退出虚拟环境：

```
deactivate
```

通过Python版本检查、虚拟环境创建、激活和验证，项目便具备了稳定的运行基础。后续所有代码都应在对应虚拟环境中运行，以保证依赖版本一致和实验结果可复现。

2.1.3 pip、uv 与依赖安装

在AI Agent开发中，第三方依赖几乎贯穿整个工程过程。大模型SDK、Web框架、向量数据库、文档解析工具以及工作流框架，都需要通过依赖管理工具进行安装与维护。

1. pip的作用与基础使用

pip是Python官方默认提供的包管理工具，主要用于安装、升级和卸载第三方依赖库。AI Agent开发中，大部分常用组件都可以通过pip直接安装。例如安装OpenAI SDK：

```
pip install openai
```

安装FastAPI和Uvicorn：

```
pip install fastapi uvicorn
```

如果需要一次性安装多个依赖，可以连续写在同一条命令中：

```
pip install openai fastapi uvicorn python-dotenv pydantic
```

安装完成后，可以通过以下命令查看当前环境中的依赖列表：

```
pip list
```

输出结果如下：

Package	Version
openai	1.35.10
fastapi	0.115.0
uvicorn	0.30.6
pydantic	2.8.2
python-dotenv	1.0.1

其中，第一列表示依赖名称，第二列表示当前安装版本。开发过程中如果出现版本不兼容问题，通常需要优先检查这里的版本信息。

2. uv的特点与安装方式

随着Python工程规模不断增大，传统pip在安装速度和依赖解析方面逐渐暴露出效率问题，因此越来越多AI项目开始使用uv。uv是近年来较流行的Python依赖管理工具，其底层使用高性能实现，在依赖下载和环境解析方面速度明显快于传统pip。

安装uv：

```
pip install uv
```

安装完成后，可以检查版本：

```
uv --version
```

正常输出如下：

```
uv 0.4.20
```

使用uv安装依赖：

```
uv pip install openai fastapi uvicorn
```

与传统pip相比，uv会更快完成依赖解析与下载，尤其在依赖较多时优势更明显。此外，uv还支持快速创建虚拟环境和管理项目依赖，在后续大型Agent项目开发中会更加方便。

3. 依赖文件与环境统一

随着项目功能增加，依赖数量会逐渐增多。如果只依赖手动安装，很容易遗漏某些库，因此通常需要使用依赖文件统一管理项目环境。最常见的方式是生成requirements.txt文件。

导出当前环境依赖：

```
pip freeze > requirements.txt
```

文件内容如下：

```
openai==1.35.10
fastapi==0.115.0
uvicorn==0.30.6
pydantic==2.8.2
python-dotenv==1.0.1
```

当项目迁移到其他计算机时，只需要执行：

```
pip install -r requirements.txt
```

即可自动安装所有依赖，从而保证不同环境中的版本一致性。这种方式在团队协作、课程实验和项目部署中非常常见，也是AI工程开发的重要基础。

除了依赖导出外，还需要注意虚拟环境中的安装操作。所有依赖都应优先安装在项目虚拟环境内，而不是系统全局环境中，否则容易导致不同项目之间出现版本冲突。随着后续章节引入RAG、工作流和多智能体框架，依赖管理的重要性会进一步体现，因此建立规范的安装与管理习惯十分必要。

2.1.4 API Key 与环境变量配置

大模型服务通常需要通过API接口进行访问，而接口调用必须依赖身份认证信息。API Key用于标识调用者身份，并控制模型服务的访问权限和调用额度。为了避免在代码中直接暴露敏感信息，实际开发中通常会使用环境变量保存配置，从而提高项目的安全性与可维护性。

1. API Key的作用与获取方式

API Key可以理解为访问大模型服务的“身份凭证”。当程序向模型服务器发送请求时，需要携带正确的API Key，否则接口会拒绝访问。在AI Agent开发中，无论是对话生成、工具调用还是Embedding向量生成，都需要依赖API Key完成认证。

不同模型平台获取API Key的方式略有区别，但整体流程基本一致。通常需要先注册平台账号，然后进入开发者控制台创建密钥。创建成功后，会得到如下字符串：

```
sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

这类密钥通常具有调用权限和计费权限，因此不能直接写入公开代码，也不建议上传到Git仓库。很多平台一旦检测到密钥泄露，会自动禁用对应Key，从而导致程序无法继续运行。

2. 环境变量的配置方式

为了安全管理API Key，工程开发中通常使用环境变量保存敏感配置。所谓环境变量，本质上是系统中的键值配置，程序运行时可以动态读取，而不需要把敏感信息直接写死在代码中。

AI Agent项目中最常见的方式是使用.env文件。在项目根目录创建：

```
.env
```

文件内容如下：

```
OPENAI_API_KEY=替换为实际Key
OPENAI_BASE_URL=https://api.openai.com/v1
OPENAI_MODEL=gpt-4.1-mini
```

其中：

- (1) OPENAI_API_KEY用于保存接口密钥。
- (2) OPENAI_BASE_URL用于指定模型接口地址。
- (3) OPENAI_MODEL用于指定默认模型名称。

随后安装环境变量读取工具：

```
pip install python-dotenv
```

在Python程序中读取配置：

```
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
base_url = os.getenv("OPENAI_BASE_URL")
model_name = os.getenv("OPENAI_MODEL")

print(api_key)
print(base_url)
print(model_name)
```

运行程序后，会输出对应的配置内容。

3. 模型接口调用中的环境变量使用

完成环境变量配置后，大模型SDK即可直接读取配置完成接口调用。这样做的好处是，当模型地址、API Key或模型名称发生变化时，只需要修改.env文件，而不需要改动业务代码。

例如，创建模型客户端：

```
from openai import OpenAI
import os
from dotenv import load_dotenv

load_dotenv()
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)
```

发送请求：

```
response = client.chat.completions.create(
    model=os.getenv("OPENAI_MODEL"),
    messages=[
        {
            "role": "user",
            "content": "解释什么是AI Agent"
        }
    ]
)
print(response.choices[0].message.content)
```

输出结果如下：

```
AI Agent是一种能够理解目标、调用工具并完成任务的软件系统。
```

在实际开发中，除了API Key之外，数据库地址、Redis配置、向量库路径和服务端口等内容也常通过环境变量统一管理。随着项目规模增加，环境变量会逐渐成为整个系统配置管理的重要组成部分。

2.2 Python 项目基础

本节将重点讲解AI Agent项目的目录结构设计、模块导入方法与配置文件管理，同时介绍Pydantic数据模型、日志输出与异常处理的实用技巧。

2.2.1 项目目录结构

随着AI Agent功能不断增加，项目中的代码、配置、数据和工具数量也会逐渐增多。如果缺少

清晰的目录结构，代码很容易出现混乱，后续维护与扩展也会变得困难。合理的项目目录不仅能够提高开发效率，还能帮助快速定位模块功能，是工程化开发的重要基础。

1. 为什么需要规范的目录结构

很多初学者在编写代码时，习惯把所有逻辑都写在一个文件中。项目规模较小时，这种方式问题并不明显；但当系统逐渐加入工具调用、知识库、工作流、Web接口和日志系统后，单文件结构会迅速失控。例如，模型调用代码、工具函数、配置加载和接口逻辑全部混在一起，会导致代码难以阅读，也不利于后续功能扩展。

规范的目录结构本质上是对功能进行分类管理。不同功能放入不同目录后，可以降低模块之间的耦合度，使代码更容易维护。例如，修改工具逻辑时，只需要进入工具目录；修改接口服务时，只需要查看API目录。对于AI Agent项目而言，这种模块化结构尤为重要，因为后续系统往往涉及模型、工具、记忆、RAG和多Agent协同等多个部分。

2. 常见AI Agent项目目录设计

一个基础AI Agent项目通常包含配置、核心逻辑、工具、数据和接口等目录。下面是较常见的结构：

```
ai-agent-project/  
├── app/  
│   ├── agent/  
│   ├── tools/  
│   ├── rag/  
│   ├── api/  
│   └── memory/  
├── data/  
├── logs/  
├── tests/  
├── .env  
├── requirements.txt  
└── main.py
```

其中：

- (1) `app`用于存放核心业务代码。
- (2) `agent`用于保存Agent运行逻辑。
- (3) `tools`用于保存工具函数。
- (4) `rag`用于存放知识库相关代码。
- (5) `api`用于提供接口服务。

- (6) `memory`用于存放记忆模块代码。
- (7) `data`用于保存文档和向量数据。
- (8) `logs`用于日志输出。
- (9) `tests`用于存放测试代码。

这种结构的优点是职责清晰，不同功能彼此独立。即使后续功能扩展，也可以继续新增子目录，而不会破坏原有结构。

3. 主入口文件与运行流程

AI Agent项目通常会有一个主入口文件，用于启动程序或初始化系统。最常见的是：

```
main.py
```

该文件负责加载配置、初始化模型客户端、创建Agent对象并启动运行流程。例如：

```
from app.agent.chat_agent import ChatAgent

agent = ChatAgent()
result = agent.run("解释什么是AI Agent")
print(result)
```

运行方式：

```
python main.py
```

如果项目后续增加Web服务，则可能新增：

```
api_server.py
```

用于启动FastAPI接口服务：

```
uvicorn api_server:app --reload
```

随着项目规模扩大，入口文件可能会进一步拆分，例如命令行启动入口、Web服务入口和任务调度入口分别独立。但无论结构如何变化，核心原则始终是“功能解耦、职责清晰、方便维护”。

2.2.2 模块导入与配置文件

随着AI Agent项目逐渐复杂，代码文件数量会不断增加。为了让不同功能之间能够协同工作，需要通过模块导入组织代码结构；而模型地址、API Key、数据库路径等配置内容，则需要通过配置文件统一管理。合理的模块组织与配置管理，可以显著提升项目的可维护性与扩展能力。

1. Python模块导入机制

在Python项目中，每个.py文件本质上都可以视为一个模块。通过模块导入，不同文件之间可以共享函数、类和变量，从而避免重复编写代码。例如，将模型调用逻辑单独放入：

```
app/model/client.py
```

文件内容：

```
from openai import OpenAI

def create_client(api_key, base_url):
    return OpenAI(
        api_key=api_key,
        base_url=base_url
    )
```

在其他文件中即可直接导入：

```
from app.model.client import create_client
```

随后调用：

```
client = create_client(api_key, base_url)
```

这种方式能够将模型初始化逻辑集中管理。如果后续需要切换模型服务，只需要修改一个文件即可，而不需要在多个位置重复修改代码。

2. 包结构与初始化文件

当项目包含多个目录时，需要通过包结构来管理模块关系。在Python中，只要目录下存在：

```
__init__.py
```

该目录就可以作为Python包进行导入。例如：

```
app/
├── agent/
│   ├── __init__.py
│   └── chat_agent.py
```

此时可以直接导入：

```
from app.agent.chat_agent import ChatAgent
```

如果缺少__init__.py文件，某些环境下可能会出现模块无法识别的问题。虽然新版本Python已经对部分目录结构进行了优化，但在工程开发中，仍建议保留初始化文件，以保证项目结构更加稳定。

此外，还需要注意模块命名规范。文件名应尽量使用小写字母与下画线，例如：

```
chat_agent.py
tool_manager.py
memory_store.py
```

不建议使用空格、中文或特殊符号，否则可能导致导入异常。

3. 配置文件的组织方式

AI Agent项目通常包含大量配置项，例如API Key、模型名称、数据库地址、日志等级和服务端口等。如果把这些配置直接写入代码，不仅不利于维护，也存在安全风险。因此，工程开发中通常会使用配置文件统一管理。

最常见的是.env文件：

```
OPENAI_API_KEY=替换为实际Key
OPENAI_MODEL=gpt-4.1-mini
SERVER_PORT=8000
LOG_LEVEL=INFO
```

随后通过：

```
from dotenv import load_dotenv
import os

load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
model_name = os.getenv("OPENAI_MODEL")
```

读取配置内容。除了.env之外，也可以使用JSON配置文件。例如：

```
config.json
```

内容如下：

```
{
  "model": "gpt-4.1-mini",
  "temperature": 0.3,
  "max_tokens": 2048
}
```

读取方式：

```
import json

with open("config.json", "r", encoding="utf-8") as f:
    config = json.load(f)
    print(config["model"])
```

运行结果：

```
gpt-4.1-mini
```

在实际开发中，环境变量更适合保存敏感信息，而JSON文件更适合保存普通业务配置。通过模块导入与配置文件管理，可以让AI Agent项目逐渐形成清晰、稳定的工程结构，为后续功能扩展和服务部署打下基础。

2.2.3 Pydantic 数据模型、日志输出与异常处理

AI Agent程序不仅要能够调用模型，还要能够稳定处理输入、记录运行过程并应对错误情况。Pydantic数据模型、日志输出与异常处理分别对应数据规范、过程可见和故障控制三个方面。三者结合后，可以让程序从简单脚本逐步具备工程化应用的基本特征。

02

1. Pydantic数据模型：规范输入与输出结构

Pydantic是Python中常用的数据校验工具，适合用来定义Agent的输入、输出、工具参数和任务状态。普通字典虽然使用方便，但字段是否存在、类型是否正确、数据是否符合要求都需要手动检查。Pydantic可以通过类的方式提前声明字段结构，并在数据创建时自动完成校验。

例如，定义一个Agent请求模型：

```
from pydantic import BaseModel, Field

class AgentRequest(BaseModel):
    user_input: str = Field(description="用户输入内容")
    max_steps: int = Field(default=5, description="最大执行步数")
```

创建数据对象：

```
request = AgentRequest(
    user_input="生成一份Python学习计划",
    max_steps=3
)
print(request.model_dump())
```

运行结果如下：

```
{'user_input': '生成一份Python学习计划', 'max_steps': 3}
```

如果传入错误类型，Pydantic会自动抛出校验异常，从而避免错误数据继续进入模型调用、工具调用或任务执行流程。对于AI Agent来说，结构清晰的数据模型能够提高系统稳定性，也方便后续接口服务和工作流模块的复用。

2. 日志输出：记录程序运行过程

日志用于记录程序运行时发生了什么，是调试AI Agent的重要工具。模型调用是否成功、工具是否被触发、任务执行到哪一步、异常出现在哪个环节，都可以通过日志进行追踪。与直接使用print相比，日志系统可以区分信息级别，并且更适合长期维护。

Python内置logging模块，可以直接用于项目开发：

```
import logging

logging.basicConfig()
```

```
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s"
)
logging.info("Agent程序启动")
logging.info("开始处理用户输入")
logging.warning("工具调用结果为空")
```

运行结果如下：

```
2026-05-11 10:30:12 - INFO - Agent程序启动
2026-05-11 10:30:12 - INFO - 开始处理用户输入
2026-05-11 10:30:12 - WARNING - 工具调用结果为空
```

在Agent开发中，建议至少记录模型请求开始、模型返回结果、工具调用名称、工具返回内容和异常信息。这样在程序结果不符合预期时，可以根据日志快速定位问题，而不是盲目修改代码。

3. 异常处理：保证程序稳定运行

异常处理用于应对程序运行中的错误情况，例如API Key缺失、模型接口超时、工具参数错误、文件不存在或JSON解析失败等。如果不进行异常处理，程序一旦出错就会直接终止，影响用户体验和后续任务执行。

可以使用try...except捕获错误：

```
import os
from dotenv import load_dotenv

load_dotenv()
try:
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key:
        raise ValueError("未检测到OPENAI_API_KEY配置")

    print("API Key读取成功")
except ValueError as e:
    print(f"配置错误: {e}")
except Exception as e:
    print(f"未知错误: {e}")
```

运行结果可能如下：

```
配置错误: 未检测到OPENAI_API_KEY配置
```

在工程开发中，不同错误应尽量给出明确提示。例如，配置错误应提示检查.env文件，网络错误应提示检查接口地址或代理设置，工具参数错误应提示输入字段是否正确。对于AI Agent系统来说，数据模型保证输入规范，日志输出保证过程可见，异常处理保证程序稳定，三者共同构成可靠运行的基础。

2.3 编写大模型调用程序

完成环境配置后，本节我们将介绍大模型接口调用的流程与方法，这也是AI Agent运行的核心环节。我们将从大模型SDK安装、API Key与模型参数配置入手，讲解如何完成第一次模型调用，再逐步实现命令行聊天程序、多轮对话功能与流式输出，整个过程构成了AI应用与模型服务之间的基础通信链路，也是后续工具调用与 workflows 执行的重要前提。

2.3.1 安装 SDK 并完成模型调用

大模型调用是AI Agent开发的起点。完成SDK安装后，程序便可以通过统一接口向模型服务发送请求，并接收模型返回的文本结果。这个过程虽然代码量不多，但涉及依赖安装、密钥读取、客户端创建、消息组织和结果解析，是后续开发工具调用型Agent、知识库Agent和多智能体系统的基础。

1. 安装大模型SDK

SDK是模型服务提供的开发工具包，用于简化接口调用过程。如果直接使用HTTP请求，需要手动组织请求头、请求体和返回解析；使用SDK后，只需要调用封装好的方法即可完成请求。以OpenAI风格的SDK为例，先确认已经进入项目虚拟环境，然后安装依赖：

```
pip install openai python-dotenv
```

安装完成后，可以查看依赖是否存在：

```
pip list
```

输出结果中应包含如下内容：

```
openai          1.35.10
python-dotenv   1.0.1
```

其中，openai用于调用模型接口，python-dotenv用于读取.env配置文件。

2. 配置API Key与模型参数

模型调用需要密钥和模型名称。为避免把密钥直接写入代码，通常在项目根目录创建.env文件：

```
OPENAI_API_KEY=替换为实际API Key
OPENAI_BASE_URL=https://api.openai.com/v1
OPENAI_MODEL=gpt-4.1-mini
```

随后在Python代码中加载配置：

```
import os
from dotenv import load_dotenv
```

```
load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
base_url = os.getenv("OPENAI_BASE_URL")
model_name = os.getenv("OPENAI_MODEL")

print(model_name)
```

运行命令：

```
python main.py
```

输出结果如下：

```
gpt-4.1-mini
```

如果输出为None，通常说明.env文件位置不正确、字段名称拼写错误，或程序并未在项目根目录下运行。模型配置建议集中放在环境变量中，便于后续切换模型、修改接口地址和部署到不同环境。

3. 完成第一次模型调用

完成依赖安装和配置读取后，即可创建模型客户端并发送请求。创建main.py文件：

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)

response = client.chat.completions.create(
    model=os.getenv("OPENAI_MODEL"),
    messages=[
        { "role": "system",
          "content": "你是一个专业的AI Agent开发助手。" },
        { "role": "user",
          "content": "用一句话说明AI Agent的作用。" }
    ],
    temperature=0.3
)

answer = response.choices[0].message.content
print(answer)
```

运行程序：

```
python main.py
```

可以得到如下输出：

AI Agent能够理解用户目标、调用工具并按照步骤完成任务。

这段程序完成了最基本的模型调用流程：读取配置、创建客户端、组织消息、发送请求、解析返回结果。`system`消息用于设定模型角色与行为风格，`user`消息用于提交实际问题，`temperature`用于控制输出随机性。对于工程开发而言，第一次模型调用的意义不只在于获得回答，更在于验证本地环境、SDK、API Key和网络连接是否全部可用。

02

2.3.2 编写命令行聊天程序

命令行聊天程序是大模型应用最基础的交互形态。它不依赖复杂界面，只通过终端接收用户输入，并将模型回复直接输出到屏幕。通过这种方式，可以清楚观察输入、请求、返回和多轮对话之间的关系，也便于后续逐步加入记忆、工具调用和Agent控制逻辑。

1. 命令行聊天程序的基本结构

命令行聊天程序通常由三部分组成：读取用户输入、调用大模型接口、输出模型回复。程序运行后会进入循环状态，每次从终端读取一段文字，把它作为用户消息发送给模型，再将模型返回内容打印出来。如果输入退出命令，程序结束运行。

创建`chat_cli.py`文件：

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)

model_name = os.getenv("OPENAI_MODEL")
while True:
    user_input = input("用户: ")
    if user_input.strip() in ["退出", "exit", "quit"]:
        print("程序已结束")
        break
    response = client.chat.completions.create(
        model=model_name,
        messages=[
            {
                "role": "system",
                "content": "你是一个专业、清晰、耐心的AI助手。"
            },
```

```
        {
            "role": "user",
            "content": user_input
        }
    ],
    temperature=0.3
)
answer = response.choices[0].message.content
print(f"助手: {answer}")
```

运行程序：

```
python chat_cli.py
```

终端交互结果如下：

```
用户：什么是AI Agent？
助手：AI Agent是一种能够理解任务目标、调用工具并按照步骤完成任务的软件系统。
```

这个程序已经具备最基础的聊天能力，但每次请求只包含当前输入，模型无法记住前面的对话内容。

2. 加入多轮对话历史

要让命令行程序具备连续对话能力，需要保存历史消息。大模型接口中的`messages`本质上是一个消息列表，列表中包含系统提示词、用户消息和助手回复。每次对话时，将新的用户输入追加到列表中，再把完整消息列表发送给模型，模型就可以基于前文进行回答。

修改后的代码如下：

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)
model_name = os.getenv("OPENAI_MODEL")
messages = [
    {
        "role": "system",
        "content": "你是一个专业、清晰、耐心的AI助手。"
    }
]
while True:
    user_input = input("用户: ")
    if user_input.strip() in ["退出", "exit", "quit"]:
```

```

        print("程序已结束")
        break
    messages.append({
        "role": "user",
        "content": user_input
    })
    response = client.chat.completions.create(
        model=model_name,
        messages=messages,
        temperature=0.3
    )
    answer = response.choices[0].message.content
    messages.append({
        "role": "assistant",
        "content": answer
    })

    print(f"助手: {answer}")

```

运行效果如下:

```

用户: 什么是AI Agent?
助手: AI Agent是一种能够理解目标、调用工具并完成任务的软件系统。
用户: 它和普通聊天机器人有什么区别?
助手: 普通聊天机器人主要负责回答问题, 而AI Agent更强调围绕目标执行任务, 例如调用工具、记录状态和分步骤完成操作。

```

第二次提问中的“它”能够被正确理解, 是因为程序把第一轮对话也传入了模型。由此可以看出, 对话历史是实现连续交互的重要基础。

3. 增加基础异常处理与输入控制

真实程序不能只考虑正常输入, 还需要处理空输入、接口异常和配置缺失等情况。命令行聊天虽然简单, 但也应具备基本稳定性。可以在循环中加入输入检查, 并对模型调用过程进行异常捕获。

完善后的代码如下:

```

import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
base_url = os.getenv("OPENAI_BASE_URL")
model_name = os.getenv("OPENAI_MODEL")

if not api_key:
    raise ValueError("未检测到OPENAI_API_KEY, 请检查.env文件配置")
client = OpenAI(

```

```

    api_key=api_key,
    base_url=base_url
)
messages = [
    {
        "role": "system",
        "content": "你是一个专业、清晰、耐心的AI助手。"
    }
]
print("命令行聊天程序已启动，输入"退出"结束程序。")

while True:
    user_input = input("用户: ").strip()
    if user_input in ["退出", "exit", "quit"]:
        print("程序已结束")
        break
    if not user_input:
        print("助手: 请输入有效内容。")
        continue
    messages.append({
        "role": "user",
        "content": user_input
    })
    try:
        response = client.chat.completions.create(
            model=model_name,
            messages=messages,
            temperature=0.3
        )
        answer = response.choices[0].message.content
        messages.append({
            "role": "assistant",
            "content": answer
        })
        print(f"助手: {answer}")
    except Exception as e:
        print(f"助手: 模型调用失败，错误信息: {e}")

```

运行后，如果输入空内容，会得到：

```

用户：
助手：请输入有效内容。

```

如果.env中未配置OPENAI_API_KEY，程序会给出明确错误提示：

```

ValueError: 未检测到OPENAI_API_KEY，请检查.env文件配置

```

通过这个命令行聊天程序，可以完成从单轮模型调用到多轮对话管理的过渡。它虽然没有图

形界面，但已经包含AI应用的核心运行链路：读取输入、维护上下文、调用模型、解析输出和处理异常。后续只需要继续加入工具注册、记忆存储和任务规划逻辑，就可以逐步演进为完整的Agent程序。

2.3.3 实现流式输出

02

流式输出可以让模型回复像打字一样逐步显示，而不是等待完整结果一次性返回。对于聊天程序和Agent应用来说，流式输出能够明显改善交互体验，尤其在回答较长、任务处理较复杂时，可以让用户更快看到系统正在响应，也便于观察模型生成过程。

1. 流式输出的基本原理

普通模型调用会在服务端生成完整内容后一次性返回，程序只有等到全部生成结束，才能拿到最终结果。流式输出则不同，模型会把生成内容拆成多个小片段逐步返回，程序一边接收，一边打印到终端。因此，用户不需要等待完整回答生成完毕，就能看到内容逐字或逐段显示。

普通调用方式如下：

```
response = client.chat.completions.create(
    model=model_name,
    messages=messages
)

answer = response.choices[0].message.content
print(answer)
```

这种方式只有在response完整返回后才会打印结果。流式调用需要增加stream=True参数：

```
stream = client.chat.completions.create(
    model=model_name,
    messages=messages,
    stream=True
)
```

此时返回结果不再是一个完整对象，而是一个可以逐步遍历的数据流。程序需要通过循环读取每个片段，并将片段内容输出到屏幕。流式输出并不会改变模型的理解能力，只是改变了结果返回方式，适合聊天程序、Web前端和长文本生成场景。

2. 在命令行中实现流式打印

基于前面的命令行聊天程序，可以将模型调用部分改为流式输出。创建chat_stream.py文件：

```
import os
from dotenv import load_dotenv
from openai import OpenAI
```

```
load_dotenv()

api_key = os.getenv("OPENAI_API_KEY")
base_url = os.getenv("OPENAI_BASE_URL")
model_name = os.getenv("OPENAI_MODEL")

if not api_key:
    raise ValueError("未检测到OPENAI_API_KEY, 请检查.env文件配置")
client = OpenAI(
    api_key=api_key,
    base_url=base_url
)
messages = [
    {
        "role": "system",
        "content": "你是一个专业、清晰、耐心的AI助手。"
    }
]
print("命令行流式聊天程序已启动, 输入“退出”结束程序。")

while True:
    user_input = input("\n用户: ").strip()
    if user_input in ["退出", "exit", "quit"]:
        print("程序已结束")
        break
    if not user_input:
        print("助手: 请输入有效内容。")
        continue
    messages.append({
        "role": "user",
        "content": user_input
    })
    try:
        stream = client.chat.completions.create(
            model=model_name,
            messages=messages,
            temperature=0.3,
            stream=True
        )
        print("助手: ", end="", flush=True)

        full_answer = ""
        for chunk in stream:
            delta = chunk.choices[0].delta.content
            if delta:
                print(delta, end="", flush=True)
                full_answer += delta
        print()
        messages.append({
```

```

        "role": "assistant",
        "content": full_answer
    })
except Exception as e:
    print(f"\n助手: 模型调用失败, 错误信息: {e}")

```

运行程序:

```
python chat_stream.py
```

终端显示效果如下:

命令行流式聊天程序已启动, 输入“退出”结束程序。

用户: 介绍一下AI Agent

助手: AI Agent是一种能够理解任务目标、调用工具并按照步骤完成任务的软件系统。它通常由大模型、工具、记忆和规划模块组成。

在真实运行时, 助手回复会逐步显示, 而不是一次性出现。代码中的`print(delta,end="", flush=True)`非常关键, 其中`end=""`表示不自动换行, `flush=True`表示立即刷新终端输出, 否则内容可能会被缓存, 导致显示不够及时。

3. 保存完整回复与处理流式异常

流式输出时, 模型回复被拆成多个片段返回。如果后续还需要继续多轮对话, 就必须把这些片段拼接成完整回复, 并保存到`messages`中; 否则, 模型虽然已经输出了内容, 但下一轮对话无法获得上一轮助手回复, 连续对话能力会受到影响。

下面这段逻辑用于保存完整回答:

```

full_answer = ""

for chunk in stream:
    delta = chunk.choices[0].delta.content
    if delta:
        print(delta, end="", flush=True)
        full_answer += delta
messages.append({
    "role": "assistant",
    "content": full_answer
})

```

流式输出还需要注意异常处理。由于模型返回过程会持续一段时间, 网络中断、接口超时或服务异常都可能发生在输出过程中。因此, 流式调用最好放在`try...except`结构内, 并在异常发生时给出明确提示:

```

try:
    stream = client.chat.completions.create(

```

```
        model=model_name,
        messages=messages,
        stream=True
    )
    for chunk in stream:
        delta = chunk.choices[0].delta.content
        if delta:
            print(delta, end="", flush=True)
    except Exception as e:
        print(f"流式输出中断，错误信息：{e}")
```

如果接口异常，终端会显示如下结果：

```
流式输出中断，错误信息：Connection error.
```

流式输出不仅提升了命令程序的交互体验，也为后续Web聊天界面、SSE接口和Agent执行过程展示奠定基础。对于AI Agent应用而言，流式输出可以让任务执行过程更加可见，使用户在等待长结果时能够持续获得反馈。

2.4 编写简单的 Agent

当大模型具备稳定调用能力后，系统便可以进一步向AI Agent形态演进。通过加入任务输入、上下文管理和输出控制机制，普通模型调用程序开始具备基本的智能体特征，也为后续工具增强、知识库接入与自主任务执行奠定基础。

本书我们编写一个简单的Agent，其原理框图如图2-2所示。



图 2-2 一个简单 Agent 的原理图

2.4.1 Agent 输入、处理与输出流程

简单Agent并不是复杂系统的堆叠，而是围绕“输入、处理、输出”形成的基础运行流程。用户输入提供任务目标，处理环节完成理解、组织上下文和调用模型，输出环节将模型结果返回给用户。掌握这一流程后，后续加入工具、记忆和规划机制都会更加自然。

1. 输入环节：接收任务并整理为标准消息

Agent的输入可以来自命令行、网页表单、接口请求或文件内容。无论来源如何，进入模型之前都需要整理成统一格式。对于大模型接口来说，最常见的输入结构是消息列表，其中包含系统消息和用户消息。系统消息用于规定Agent的身份、能力边界和回答风格，用户消息则用于保存当前任务内容。

例如，一个最简单的输入结构如下：

```
messages = [
    {
        "role": "system",
        "content": "你是一个AI学习助手，回答要清晰、准确。"
    },
    {
        "role": "user",
        "content": "帮我解释什么是AI Agent。"
    }
]
```

其中，role表示消息角色，content表示消息内容。输入环节的重点不是简单接收文字，而是把原始输入转换为模型能够理解的结构化上下文。如果后续需要加入多轮对话，还可以继续向messages中追加历史消息，使Agent能够结合前文进行回答。

2. 处理环节：调用模型并生成中间结果

处理环节是Agent运行的核心。程序会将整理好的消息发送给大模型，由模型完成语义理解、内容生成和初步推理。此时，Agent虽然还没有工具调用和任务规划能力，但已经具备基本的信息处理能力，能够根据输入生成相应回答。

模型调用代码如下：

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)
response = client.chat.completions.create(
    model=os.getenv("OPENAI_MODEL"),
    messages=messages,
    temperature=0.3
)
```

这段代码完成了从本地程序到模型服务的请求发送。`model`指定调用的模型名称，`messages`保存输入上下文，`temperature`用于控制生成结果的随机性。对于入门阶段的Agent，处理环节可以先保持简单，重点关注请求是否成功、返回是否正常、内容是否符合任务要求。

3. 输出环节：解析结果并返回给用户

模型返回的结果通常是一个结构化对象，不能直接作为最终文本使用，需要从返回对象中取出具体回答内容。对于聊天模型接口，常见路径是读取`choices`中的第一条结果，再取出`message.content`字段。

结果解析代码如下：

```
answer = response.choices[0].message.content
print(answer)
```

运行结果如下：

```
AI Agent是一种能够理解任务目标、调用工具并完成任务的软件系统。
```

在完整程序中，输出不仅可以打印到终端，也可以返回给网页、接口或文件系统。例如，命令行程序直接使用`print`，Web服务则可以将结果封装成JSON返回。输出环节还可以进一步加入格式控制，例如要求返回列表、步骤说明或结构化数据。由此可见，Agent的基础流程虽然简单，但已经包含输入整理、模型处理和结果输出三个关键环节。后续工具调用、记忆管理和任务规划都是在这个基本流程上逐步扩展出来的。

2.4.2 为 Agent 添加系统提示词

系统提示词是Agent行为控制的基础配置，用于规定智能体的角色定位、任务范围、回答风格和执行边界。

相比普通用户输入，系统提示词具有更高优先级，会在每次模型调用时作为背景规则传入模型。合理设计系统提示词可以让Agent输出更加稳定，也更符合具体应用场景。

1. 系统提示词的作用

系统提示词可以理解为Agent的“运行规则”。没有系统提示词时，大模型只会根据用户输入直接生成回答，输出风格和内容边界不够稳定。加入系统提示词后，可以提前规定Agent应该扮演什么角色、采用什么表达方式、遵守哪些限制条件，以及遇到不确定问题时如何处理。

例如，一个学习助手Agent的系统提示词可以写成：

```
system_prompt = "你是一个Python学习助手，回答要清晰、准确，适合初学者理解。"
```

将其放入消息列表：

```
messages = [
```

```
{
    "role": "system",
    "content": system_prompt
},
{
    "role": "user",
    "content": "解释一下Python中的列表。"
}
]
```

这样模型在回答时会更倾向于使用教学语气，并避免输出过于复杂的解释。系统提示词并不是简单装饰，而是影响Agent行为的关键配置。

2. 系统提示词的基本组成

一个实用的系统提示词通常包含角色、任务、约束和输出要求4类信息，如图2-3所示。

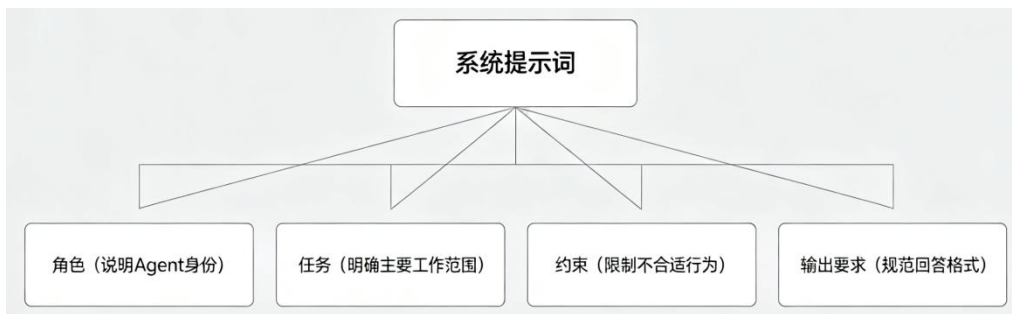


图 2-3 系统提示词的组成

角色用于说明Agent的身份，任务用于说明主要工作范围，约束用于限制不合适行为，输出要求用于规范回答格式。

例如：

```
system_prompt = """
你是一个AI Agent开发助手，主要帮助用户理解AI Agent的基本概念和代码实现。
回答时要求：
1. 语言清晰，适合有Python基础的初学者；
2. 遇到代码问题时，优先给出可运行代码；
3. 不确定的信息需要明确说明；
4. 不输出与问题无关的内容。
"""
```

这段提示词比单句角色描述更完整。它不仅告诉模型“是什么角色”，还限定了回答方式和内容边界。在Agent开发中，系统提示词越清晰，输出越容易控制。对于后续工具调用型Agent，还可以在系统提示词中说明可用工具、调用原则和失败处理方式。

3. 在Agent程序中使用系统提示词

为了便于维护，系统提示词通常不直接散落在代码各处，而是集中定义或放入配置文件。下面给出一个简单Agent程序，将系统提示词作为固定规则传入模型。

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
client = OpenAI(
    api_key=os.getenv("OPENAI_API_KEY"),
    base_url=os.getenv("OPENAI_BASE_URL")
)

system_prompt = """
你是一个AI Agent入门学习助手。
回答要求：
1. 表达清楚，避免复杂术语堆砌；
2. 优先结合例子解释概念；
3. 回答内容要围绕用户问题展开；
4. 不编造不存在的功能和事实。
"""

messages = [
    {
        "role": "system",
        "content": system_prompt
    },
    {
        "role": "user",
        "content": "AI Agent和普通聊天机器人有什么区别？"
    }
]

response = client.chat.completions.create(
    model=os.getenv("OPENAI_MODEL"),
    messages=messages,
    temperature=0.3
)

print(response.choices[0].message.content)
```

运行程序：

```
python main.py
```

输出结果如下：

普通聊天机器人主要负责根据用户问题生成回答，而AI Agent不仅能回答问题，还可以围绕目标调用工具、记录状态并分步骤完成任务。例如，聊天机器人可以解释什么是学习计划，AI Agent则可以根据学习目标生成计划，并把结果保存为文件。

可以看到，系统提示词使回答更贴近“入门学习助手”的定位。后续如果需要将Agent改造成代码助手、知识库助手或个人助理，只需要调整系统提示词和相关工具能力，就可以改变Agent的行为方向。

2.4.3 保存对话历史、运行并测试最小 Agent 程序

02

对话历史是最小Agent程序具备连续交互能力的关键。没有历史记录时，每次输入都是孤立请求，模型无法理解前文中的指代关系。通过保存用户消息与助手回复，Agent可以在多轮对话中保持上下文一致，并逐步形成可运行、可测试的基础程序。

1. 保存对话历史的基本方法

在大模型调用中，对话历史通常以消息列表的形式保存。列表中的每一项都包含角色和内容，系统提示词用于定义Agent行为，用户消息用于保存输入，助手消息用于保存模型回复。每完成一轮对话，就把用户输入和助手回答追加到列表中。

基本结构如下：

```
messages = [
    {
        "role": "system",
        "content": "你是一个AI Agent入门学习助手，回答要清晰、准确。"
    }
]
```

每次接收用户输入后，追加用户消息：

```
messages.append({
    "role": "user",
    "content": user_input
})
```

模型生成回答后，再追加助手消息：

```
messages.append({
    "role": "assistant",
    "content": answer
})
```

这样，下一轮调用模型时，messages中已经包含前面的对话内容。模型可以根据历史信息理解“它”“这个概念”“刚才说的内容”等指代，从而实现连续对话。需要注意的是，历史消息不能无限增长，后续项目中通常会加入裁剪或摘要机制。

2. 编写最小Agent程序

最小Agent程序可以理解为一个具备系统提示词、对话历史和模型调用能力的命令行智能体。

它虽然还没有工具调用、知识库和任务规划，但已经具备AI Agent的基础运行框架。

创建mini_agent.py文件：

```
import os
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
base_url = os.getenv("OPENAI_BASE_URL")
model_name = os.getenv("OPENAI_MODEL")

if not api_key:
    raise ValueError("未检测到OPENAI_API_KEY，请检查.env文件配置")
client = OpenAI(
    api_key=api_key,
    base_url=base_url
)

messages = [
    {
        "role": "system",
        "content": "你是一个AI Agent入门学习助手，回答要清晰、准确，并尽量结合例子说明。"
    }
]

print("最小Agent程序已启动，输入\"退出\"结束程序。")

while True:
    user_input = input("\n用户: ").strip()
    if user_input in ["退出", "exit", "quit"]:
        print("程序已结束")
        break
    if not user_input:
        print("Agent: 请输入有效内容。")
        continue
    messages.append({
        "role": "user",
        "content": user_input
    })
    try:
        response = client.chat.completions.create(
            model=model_name,
            messages=messages,
            temperature=0.3
        )
        answer = response.choices[0].message.content
        messages.append({
            "role": "assistant",
            "content": answer
        })
```

```
    })  
    print(f"Agent: {answer}")  
except Exception as e:  
    print(f"Agent: 模型调用失败, 错误信息: {e}")
```

运行程序:

```
python mini_agent.py
```

运行结果如下:

最小Agent程序已启动, 输入"退出"结束程序。

用户: 什么是AI Agent?

Agent: AI Agent是一种能够理解任务目标、调用工具并按照步骤完成任务的软件系统。

用户: 它和普通聊天机器人有什么区别?

Agent: 普通聊天机器人主要负责回答问题, 而AI Agent更强调任务执行能力。这里的“它”指的是AI Agent, 它不仅能生成回答, 还可以结合工具、记忆和规划完成多步骤任务。

第二轮对话中, 模型能够理解“它”指代AI Agent, 说明对话历史已经生效。

3. 运行测试与问题检查

最小Agent程序完成后, 需要从输入处理、上下文保持、异常提示和退出逻辑几个方面进行测试。首先测试正常问题输入, 确认模型能够返回有效回答; 然后测试连续提问, 检查上下文是否被保留; 再测试空输入, 确认程序不会向模型发送无效请求; 最后测试退出命令, 确认程序能够正常结束。

可以按照如下方式测试:

用户: 解释一下Agent中的工具

Agent: 工具是Agent调用外部能力的接口, 例如计算器、文件读取器或数据库查询器。

用户: 它为什么重要?

Agent: 这里的“它”指工具。工具很重要, 因为大模型本身主要负责理解和生成, 而工具可以让Agent执行真实操作。

如果程序报错, 需要优先检查三类问题。第一, .env文件是否存在, 字段名称是否正确; 第二, 虚拟环境是否已激活, openai和python-dotenv是否已经安装; 第三, 网络连接和模型接口地址是否可用。也可以在程序启动时打印模型名称, 确认配置读取成功:

```
print(f"当前模型: {model_name}")
```

如果输出为:

```
当前模型: None
```

说明环境变量未正确加载。经过这些测试后, 一个最小Agent程序就具备了基本可运行能力。

后续只需要在此基础上继续加入工具列表、记忆存储、结构化输出和任务规划机制，就可以逐步扩展成更完整的智能体系统。

2.5 本章小结

本章围绕AI Agent开发的基础环境与最小运行流程展开，介绍了操作系统、终端与编辑器选择，说明了Python版本、虚拟环境以及依赖安装方式；进一步讲解了项目目录结构、模块导入、配置文件、Pydantic数据模型、日志输出与异常处理；随后完成了大模型SDK安装、模型调用、命令行聊天程序与流式输出实现，并最终构建了具备系统提示词与对话历史管理能力的最小Agent程序。