第 1 章

LangGraph 基础技术

LangGraph 作为 LangChain 生态体系下的图结构智能体开发框架，核心定位是为复杂多步骤智能体任务提供高效的流程建模与执行能力。其核心设计思路是通过可视化的节点（智能体功能单元）与边（任务流转逻辑）组合，实现智能体的状态管控、分支决策及循环执行，完美适配智能体开发中推理链优化、复杂任务拆解、多智能体协同等核心场景。与传统的智能体开发框架相比，LangGraph 的核心竞争力体现在动态工作流编排、状态持久化存储及多智能体协作效率上，其关键技术组件包含节点（功能落地载体）、边（逻辑控制核心）、状态（数据缓存中心）与条件分支（智能决策机制）。相较于侧重线性流程的 LangChain、侧重数据索引的 LlamaIndex 等框架，LangGraph 在智能体复杂流程管控、动态适配能力及多智能体协同效能上具备显著优势，为智能体从原型开发到生产部署提供了全流程技术支撑。

1.1 智能体开发视角下的 LangGraph 定义

随着 AI 智能体应用场景的不断深化，简单的问答式交互已无法满足企业级智能体的开发需求，开发者亟需一种能够高效管理智能体状态、协调多组件交互、支撑复杂任务执行的技术框架。从智能体开发的核心诉求出发，无论是基础的任务型智能体，还是复杂的多智能体协同系统，其核心痛点均集中在状态管理、流程管控与组件协同三大方面，而 LangGraph 正是为解决这些痛点而生。

从技术本质来看，LangGraph 是一款轻量化 Python 开发库，其核心设计摒弃了对提示词（Prompt）或固定架构的过度抽象，转而聚焦于智能体开发的底层基础设施搭建，为开发者提供最大化的开发灵活性与流程控制权。其设计理念源于计算机科学中的状态机理论，针对智能体开发场景进行了专项优化，核心目标是让开发者能够快速构建具备持久状态、复杂逻辑流转能力的 LLM 智能体，实现真正意义上的智能化、场景化交互。

在智能体开发场景中，LangGraph 的核心设计逻辑是将智能体的执行流程抽象为一个图（Graph）

结构，各组成部分的技术定位与功能说明如下。

- 节点（Nodes）：作为智能体的核心功能单元，对应智能体的具体执行模块，可理解为智能体中的“功能执行者”。每个节点可独立实现特定功能，例如 LLM 推理、数据解析、外部工具调用、多智能体通信等，是智能体完成具体任务的核心载体。
- 边（Edges）：本质是基于 Python 的逻辑控制函数，核心作用是根据智能体当前的状态数据，决策下一个执行节点，是实现智能体流程分支、循环流转的核心技术组件，直接决定了智能体的执行逻辑与灵活性。
- 状态（State）：相当于智能体的“记忆系统”，负责存储和跟踪智能体执行过程中的所有关键数据，包括输入信息、中间结果、执行记录、外部反馈等，确保智能体能够维持上下文连贯性，做出符合场景需求的决策。

基于这种图结构设计，开发者在智能体开发过程中可实现以下核心需求：

- 构建具备清晰状态转换逻辑的智能体，使复杂的智能体执行行为可预测、可管控，降低开发与调试成本。
- 灵活实现复杂的条件分支与循环逻辑，让智能体能够根据不同的任务场景、输入数据做出差异化响应，提升智能体的场景适配能力。
- 实现智能体状态的持久化与恢复，即便系统出现重启、故障等异常情况，智能体也能从断点继续执行，保障长时间运行任务的稳定性。
- 快速搭建多智能体协同系统，让多个专业化智能体模块分工协作，共同解决单一智能体无法完成的复杂任务，提升智能体的任务处理能力。

1.2　LangGraph 在智能体开发中的核心技术优势

LangGraph 针对智能体开发的核心痛点，提供了一系列差异化技术优势，使其成为复杂智能体开发的首选框架，能够有效降低智能体开发门槛、提升开发效率、保障部署稳定性，具体优势如下。

- 持久化执行（Durable Execution）：专为长时间运行的智能体任务设计，支持智能体在故障、重启等异常场景下，从断点精准恢复执行，避免因临时问题导致任务失败，大幅提升智能体的运行稳定性，适配客服智能体、任务调度智能体等需要持续运行的场景。
- 人机协作（Human-in-the-loop）：支持在智能体执行流程的任意节点插入人类监督与干预逻辑，开发者或用户可实时检查、修改智能体状态，实现人类智慧与 AI 能力的深度融合，解决智能体在复杂场景下决策偏差的问题，提升智能体的可靠性。
- 全维度记忆管理（Comprehensive Memory）：支持智能体同时具备短期工作记忆与跨会话长期持久记忆，短期记忆用于存储当前任务的上下文信息，长期记忆用于积累历史交互数据，使智能体能够提供个性化、上下文连贯的服务，适配个性化助手、客户服务等场景。
- 可视化调试能力（Debugging）：与 LangSmith 工具深度集成，提供可视化的执行路径追踪、状态转换监控功能，开发者可清晰查看智能体的每一步执行过程、节点交互逻辑及状态变化，大幅简化智能体开发中的故障排查与流程优化工作。

- 生产级部署支持（Production-ready Deployment）：具备可扩展的基础设施设计，专门适配有状态、长时间运行的智能体工作流，能够满足企业级智能体的高可靠性、高性能需求，支持从原型开发到生产部署的无缝衔接，降低智能体的落地成本。
- 灵活工具集成：支持无缝对接各类外部工具、API 及数据资源，打破 LLM 的能力边界，使智能体能够执行实际操作（如文件读写、数据库查询、第三方服务调用）、获取实时信息，拓展智能体的应用场景。
- 原生多智能体支持：内置多智能体协同架构，无须额外开发即可实现多个 LLM 实例或智能体模块的分工协作、信息交互，支持复杂任务的拆解与分发，适配多智能体办公、复杂流程自动化等场景。

1.3 LangGraph 智能体开发的关键技术组件

LangGraph 的核心架构围绕智能体开发需求设计，由多个关键技术组件协同工作，覆盖智能体流程建模、状态管理、监控干预、工具集成等全流程。本节介绍这些组件的功能与应用场景。

1.3.1 图架构核心组件

图架构是 LangGraph 实现智能体流程管控的核心，所有智能体的执行逻辑均基于图结构展开，主要包含以下组件。

- 状态图（Stateful Graphs）：智能体流程建模的基础载体，每个节点对应智能体的一个执行步骤，核心特点是具备状态记忆能力，能够保留先前执行步骤的所有信息，实现上下文的连续处理，是构建有状态智能体的核心组件。
- 循环图（Cyclical Graph）：支持在图结构中设置循环路径（从某一节点出发，最终回到该节点），是智能体实现循环执行逻辑的关键，适用于需要重复执行某一任务（如数据校验、多轮交互）的智能体开发场景。
- 节点（Nodes）：智能体功能的具体实现单元，开发者可根据需求自定义节点类型，例如通过 ToolNode 实现外部工具调用，通过 LLMNode 实现文本生成与推理，通过 ConditionalNode 实现决策判断，节点的模块化设计使其具备高度的灵活性与可扩展性。
- 边（Edges）：智能体流程流转的控制核心，本质是逻辑决策函数，能够根据当前状态数据（如节点执行结果、外部输入），动态决定下一个执行节点，支持固定流转、条件分支等多种流转逻辑，是实现智能体灵活决策的关键。

1.3.2 监控与干预组件

人机协同（Human-in-the-loop）作为智能体的监控与干预机制，支持在智能体执行流程的任意节点设置人类介入点，当智能体遇到无法决策、执行异常等情况时，可触发人类干预，接收人类反馈后继续执行，提升智能体的可靠性与场景适配能力。在智能体开发中，该组件可用于解决复杂场景下的决策偏差、异常处理等问题。

1.3.3　工具与集成组件

该类组件主要用于拓展智能体的能力边界，实现与外部资源的无缝对接，支撑复杂智能体的开发需求。

- RAG（Retrieval-Augmented Generation，检索增强生成）：将 LLM 的推理能力与外部文档、数据库等资源结合，通过检索相关上下文信息，为智能体的决策与响应提供支撑，解决 LLM 知识滞后、上下文不足的问题，适用于知识问答、文档分析等类型的智能体开发。
- 工作流（Workflow）：通过节点的有序排列与边的逻辑配置，定义智能体的执行序列，开发者可根据任务需求，灵活设计动态工作流，实现复杂任务的拆解与分步执行，提升智能体的流程管控能力。
- API：LangGraph 提供完善的编程接口，开发者可通过 API 以编程方式创建图结构、配置节点与边、管理状态数据，实现智能体的自动化开发与集成，适配大规模、标准化的智能体开发需求。
- LangSmith：作为 LangGraph 的配套开发工具，提供智能体开发、调试、监控全流程支持，可实现 LLM 初始化、条件边配置、执行路径追踪、性能优化等功能，是提升智能体开发效率、降低调试成本的核心工具。

上述组件的协同工作，构成了 LangGraph 完整的智能体开发技术体系，开发者可基于这些组件，快速构建具备复杂流程管控、多能力集成、高可靠性的智能体系统。

1.4　智能体开发框架对比：LangGraph 与主流方案

在智能体开发领域，LangGraph、CrewAI、OpenAI Swarm 是当前应用最广泛的三款框架，三者在设计理念、技术优势、适用场景上存在显著差异，明确各框架的特点有助于开发者根据智能体项目需求选择合适的技术方案。本节从智能体开发的角度，对三款框架进行全面对比，为技术选型提供参考。

1.4.1　LangGraph 的技术特点

LangGraph 以图结构为核心，聚焦于复杂流程智能体的开发，尤其适用于自然语言处理（Natural Language Processing，NLP）相关的智能体场景，其技术特点可从优势与挑战两个方面展开。

1. 核心优势

- 流程可视化与可管控：图驱动架构使智能体的任务依赖、执行流程可被可视化呈现，开发者可清晰梳理各组件的交互逻辑，便于流程优化与团队协作，尤其适用于聊天机器人、虚拟助手等需要清晰流程管控的智能体开发。
- 架构模块化与可扩展：节点与边的模块化设计，使其能够与 LangChain、RAG 工具、外部 API 等无缝集成，支持从 NLP 任务到基础数据分析的广泛智能体应用场景，具备较强的灵活适配能力。

- 任务流程精细化管理：通过状态管理与条件分支逻辑，实现智能体任务的精细化管控，支持复杂的分支决策与循环执行，能够适配多步骤、高复杂度的智能体任务需求。
- 团队协作效率提升：可视化的流程表示，便于开发人员、数据科学家、领域专家之间的沟通协作，可快速对齐智能体的执行逻辑，降低跨角色协作成本。
- 故障排查便捷：结合 LangSmith 的可视化调试工具，可快速追踪智能体的执行路径与状态变化，直观识别任务执行中的错误与漏洞，大幅提升智能体的调试效率。

2. 开发挑战

- 学习门槛较高：对于不熟悉图结构模型、状态机理论的开发者，需要花费一定时间掌握 LangGraph 的核心设计逻辑与开发方式，上手难度相对较大。
- 大规模协同能力有限：虽然支持多智能体协同，但在需要数百、数千个智能体大规模协作的场景中，其架构的执行效率会受到影响，适配性有限。
- 高级 AI 功能支撑不足：对于涉及机器学习模型训练、强化学习、实时大数据处理等复杂技术场景的智能体，LangGraph 的原生支持能力有限，需要额外集成相关工具。
- 初始配置复杂：对于新手开发者而言，基于 LangGraph 配置复杂的智能体工作流、设置状态管理与条件分支，需要投入较多的时间与精力。

1.4.2 CrewAI 的技术特点

CrewAI 的核心设计理念是简化智能体开发流程，强调易用性与多智能体协同，主要适用于需要人机密切交互、团队任务自动化的智能体场景，其技术特点如下。

1. 核心优势

- 易用性突出：提供拖放式界面与预构建模板，无须复杂的编程操作，非技术人员也能快速上手，大幅降低智能体的开发门槛，适用于非技术团队的智能体开发需求。
- 人机协同优化：专门针对人机协同场景进行优化，支持人类操作员与智能体的无缝交互，可实时介入智能体的执行过程，适用于客户支持、应急响应等需要人类干预的智能体场景。
- 实时监控与控制：提供完善的实时监控工具，可实时查看智能体的执行状态与性能表现，人类操作员可在必要时进行干预，保障智能体的执行可靠性。
- 多智能体可扩展性强：原生支持多智能体协同架构，专为大规模多智能体环境设计，能够轻松扩展至多个智能体协同工作，适用于智能工厂、物流协调等大规模团队任务自动化场景。
- 模板可定制：提供丰富的可定制模板，开发者可根据具体业务需求，快速调整模板配置，适配不同行业、不同场景的智能体开发需求，提升开发效率。

2. 开发挑战

- 复杂技术场景适配不足：过于注重易用性，导致其在复杂技术场景（如 RAG 深度集成、复杂数据处理）中的适配能力有限，无法满足高技术复杂度的智能体开发需求。
- 计算效率有待提升：对人机协同与易用性的侧重，在一定程度上牺牲了计算效率，在处理大规模数据、高并发任务时，性能表现不佳。

- 高级功能学习成本高：虽然基础操作简单，但对于不熟悉多智能体系统的团队，要充分利用其高级协同功能，仍需要进行额外的培训与学习。
- 大规模部署资源消耗大：当智能体规模扩展至数百、数千个时，需要消耗大量的计算资源，且对人工监督的依赖度较高，部署成本也较高。

1.4.3　OpenAI Swarm 的技术特点

OpenAI Swarm 专为大规模、计算密集型智能体任务设计，核心优势在于强大的数据处理与高性能计算能力，适用于需要处理复杂数据集、实时分析的智能体场景，其技术特点如下。

1. 核心优势

- 检索型工作流支持强劲：在知识检索、数据集成、实时分析等场景中表现出色，能够高效处理检索密集型任务，适用于需要大量数据支撑的智能体开发（如金融数据分析、医疗信息检索）。
- 开源工具集成性好：与 Chroma、LangChain 等主流开源工具无缝兼容，可快速集成各类数据处理、LLM 推理工具，拓展智能体的能力边界，降低开发成本。
- 实时处理能力强：支持高性能数据分析与实时处理，能够快速响应任务需求，做出实时决策，适用于实时监控、动态调度等需要快速响应的智能体场景。
- 计算性能卓越：具备强大的高性能计算能力，能够处理极高的计算工作负载，可轻松扩展至大规模、数据密集型智能体应用场景。
- 高级 AI 模型支持完善：原生支持强化学习、监督学习、深度学习网络等高级机器学习模型，能够支撑复杂技术场景的智能体开发，适用于需要高级 AI 能力的项目。

2. 开发挑战

- 手动配置成本高：要充分发挥其性能优势，需要进行大量的手动配置与优化，尤其是针对专业场景，配置过程复杂，耗时较长。
- 计算成本高昂：高级功能与高性能计算能力伴随着较高的计算成本，对于中小型项目而言，部署成本较高，性价比不高。
- 遗留系统集成困难：其高级功能与架构设计，在与老旧、灵活性差的遗留系统集成时，会遇到较多兼容性问题，集成难度较大。
- 小型应用适配性差：对于不需要复杂数据处理、大规模计算的小型智能体应用，其高级功能过于冗余，存在“割鸡用牛刀”的问题，开发与部署成本不划算。
- 维护成本高：由于其功能复杂、计算需求高，需要持续进行系统维护与参数微调，才能保障智能体的稳定运行，维护成本较高。

1.4.4　智能体开发框架选型指南

智能体开发框架选型的核心是匹配智能体项目的业务需求、技术复杂度与部署规模，结合上述三款框架的特点，给出以下选型建议。

- 优先选择 LangGraph：若智能体项目以 NLP 相关任务为核心（如聊天机器人、虚拟助手、

文档分析），则需要复杂的流程管控、上下文连贯的交互能力，且注重流程可视化与可调试性，LangGraph 是最优选择。其图结构设计能够完美适配多步骤推理、复杂任务拆解等需求，同时与 LangChain 生态的集成性，可进一步拓展智能体的能力。

- 优先选择 CrewAI：若项目核心需求是团队任务自动化、人机协同，且开发团队中存在非技术人员，或需要快速搭建智能体原型，则 CrewAI 更为合适。其易用性与模板化设计，能够大幅降低开发门槛，同时具备强大的多智能体协同能力，可适配团队驱动型的智能体场景（如智能办公协作、物流协调）。
- 优先选择 OpenAI Swarm：若智能体项目需要处理大规模数据集、进行实时分析或复杂计算（如金融建模、医疗数据处理、实时监控），且具备充足的计算资源与技术能力，则 OpenAI Swarm 是最佳选择。其高性能计算与高级 AI 模型支持，能够支撑复杂技术场景的智能体开发，满足大规模、数据密集型任务的需求。

每个框架都有其独特的优势，因此请选择与特定业务需求相符的框架。

1.5 本章小结

本章从智能体开发视角，对 LangGraph 框架的基础技术进行了系统阐述，明确了其作为 LangChain 生态下图结构智能体框架的核心定位——以图结构为技术核心，专为解决智能体复杂多步骤任务而设计，通过节点、边、状态、条件分支四大核心组件的协同工作，实现智能体的动态流程管控、状态持久化与多智能体协同。

本章重点分析了 LangGraph 在智能体开发中的核心技术优势，包括持久化执行、人机协同、全维度记忆管理、可视化调试等。这些优势使其能够有效解决智能体开发中的状态管理、流程管控、故障排查等核心痛点，适配多样化的智能体应用场景。同时，详细介绍了 LangGraph 的图架构组件、监控干预组件、工具集成组件的功能与应用，为智能体的技术实现提供了清晰的组件参考。

通过与 CrewAI、OpenAI Swarm 两款主流智能体框架的对比，明确了 LangGraph 的差异化优势与适用场景，为开发者的技术选型提供了实用参考。本章内容为后续基于 LangGraph 进行智能体开发、流程建模、组件集成及生产部署，奠定了坚实的理论基础与技术支撑。

第 2 章

开发环境搭建

如果希望将 AI Agent 落实到具体项目，首先要解决“用什么写”和“怎么写”这两个问题。在本章中，我们将指导读者完成 Agent 开发环境搭建，内容包括 Miniconda 的安装、PyTorch 的安装、PyCharm 的安装、LLM（阿里云百炼 Qwen3）的调用以及创建一个基础聊天机器人。通过这些内容的讲解，读者将掌握开发环境的搭建方法。

2.1 开发环境安装

Python 与 AI Agent 之间有着密不可分的关系。作为当前人工智能领域主流的编程语言，Python 凭借其简洁的语法、丰富的科学计算库（如 NumPy、Pandas）以及强大的 AI 框架支持（如 TensorFlow、PyTorch、Hugging Face），成为开发 AI Agent 的首选工具。本节讲解面向 Python 的相关编程工具 Miniconda、PyTorch 与 PyCharm 的安装。

2.1.1 Miniconda 的下载与安装

Miniconda 的下载与安装步骤如下：

步骤 01 选择 Miniconda 是由于它比 Anaconda 更加小巧。访问 Anaconda 官网（https://www.anaconda.com），登录官网后，找到 Miniconda 介绍页面，如图 2.1 所示。

步骤 02 单击 Download Miniconda Installer 按钮，打开 Miniconda 下载页面。根据读者自己的计算机系统下载相应的 Miniconda 安装包，作者下载的是 Windows 64 位版本的安装包，如图 2.2 所示。

步骤 03 下载下来的安装文件名为 Miniconda3-latest-Windows-x86_64.exe，双击此文件，按安装向导操作，直到完成安装。

步骤 04 验证安装是否成功。打开 Miniconda 提供的 Anaconda Prompt 窗口，执行以下命令验证安装是否成功：

```
conda --version
python --version
```

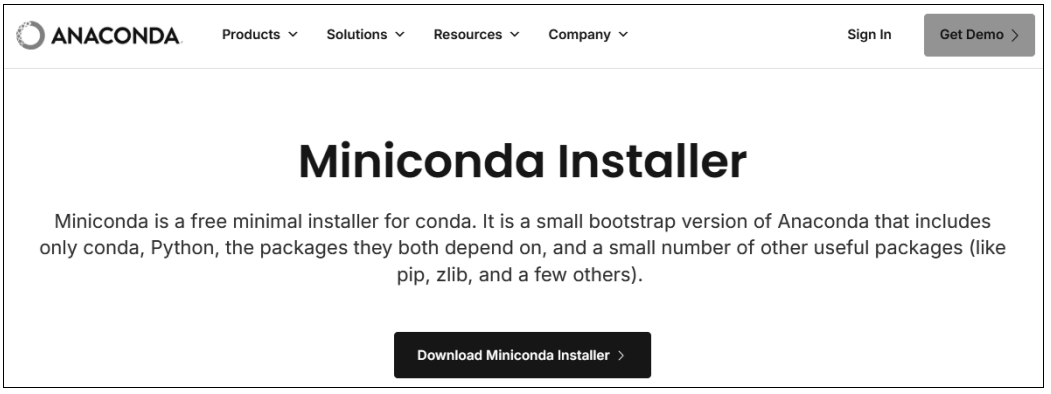

图 2.1　Anaconda 介绍页面

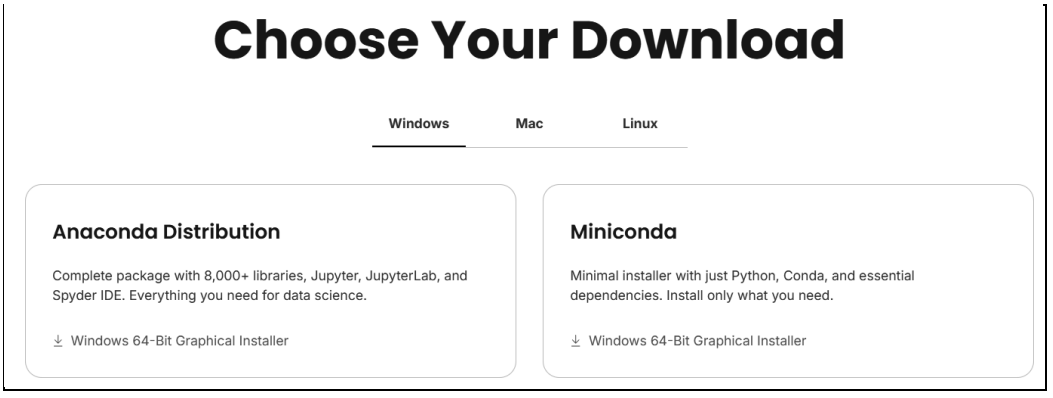

图 2.2　下载 Miniconda Windows 安装包

2.1.2　PyTorch 的下载与安装

在深度学习领域，PyTorch 凭借其卓越的灵活性与强大的功能，已成为业界广泛采用的核心框架之一。其最显著的优势在于支持动态计算图（Dynamic Computation Graph），这一特性使得模型的构建、调试和迭代过程更加直观高效。开发者可以在运行时灵活调整网络结构，实时查看中间结果，极大地提升了研发效率，特别适用于研究探索、模型快速原型设计以及复杂任务的实现。

从实践角度来看，PyTorch 提供了简洁清晰的 API 接口，具有良好的可读性和易用性，无论是初学者还是资深研究人员都能迅速上手。同时，它与 Python 生态无缝集成，支持主流数据处理和可视化工具，便于构建完整的深度学习工作流。更重要的是，PyTorch 背后拥有一个活跃且庞大的开源社区，能够持续贡献高质量的模型库、教程资源和技术支持，为实际项目落地提供了强有力的保障。

在本书重点讲解的 Qwen3、DeepSeek 等大语言模型的应用实践中，PyTorch 作为其核心后端框架，承担着模型加载、推理执行、参数微调以及分布式训练等关键任务。LLM 模型的高效部署与精细化微调，高度依赖于 PyTorch 所提供的张量计算能力、自动微分机制以及对多设备（CPU/GPU）的统一支持。因此，正确配置 PyTorch 环境是开展后续所有工作的前提和基础。

在安装层面，PyTorch 提供了 CPU 版本和 GPU 版本两种选择，以适配不同的硬件条件与性能需求。

- CPU 版本：适用于开发测试、轻量级推理或缺乏 GPU 资源的场景，安装简单，兼容性强。
- GPU 版本：能充分利用 NVIDIA CUDA 加速能力，显著提升训练与推理速度，尤其适合大规模模型（如 Qwen3）的实际部署与调优。

1. 安装 CPU 版本的 PyTorch

以 Windows 平台为例，根据读者工作计算机的资源情况，到 PyTorch 官网选择不同的命令进行安装。如果你的计算机没有 NVIDIA 显卡，可以安装 CPU 版本的 PyTorch，只需在 PyTorch 官网选择 CPU 对应的安装命令进行安装即可。例如，想要安装 PyTorch 2.8.0，命令如下：

```
   # CPU only
   pip install torch==2.8.0 torchvision==0.23.0 torchaudio==2.8.0 --index-url
https://download.pytorch.org/whl/cpu
```

2. CUDA、cuDNN、PyTorch 三者之间的关系

相比 CPU 版本的安装，GPU 版本的安装涉及 CUDA 驱动、cuDNN 等底层依赖的匹配，配置更为复杂。考虑到最新大模型对计算资源的高要求，推荐在具备 NVIDIA 显卡的环境中安装 GPU 版本 PyTorch，以充分发挥其并行计算优势，加速模型训练与推理进程。

在配置环境之前，理解 CUDA、cuDNN、PyTorch 三者之间的关系至关重要。

- CUDA：由 NVIDIA 推出的并行计算平台和编程模型。它允许软件开发者和软件工程师使用支持 CUDA 的 GPU 进行通用处理（而不仅仅是图形处理）。简而言之，PyTorch 需要 CUDA 来在 NVIDIA GPU 上执行计算。
- cuDNN：CUDA Deep Neural Network library 的缩写。它是 NVIDIA 提供的针对深度神经网络的 GPU 加速库。它提供了高度优化的常见深度学习操作（如卷积、池化、归一化等）的实现。cuDNN 是运行在 CUDA 上的一个专用加速库，PyTorch 等框架会调用它来极致地提升训练和推理速度。
- PyTorch：一个开源的机器学习框架。它本身不直接与 GPU 通信，而是通过 CUDA 接口来利用 GPU 进行计算。当你安装了 PyTorch 的 CUDA 版本（即 GPU 版本）后，PyTorch 就可以调用 CUDA 和 cuDNN 在 NVIDIA GPU 上高效运行。

它们的调用关系总结为：PyTorch→cuDNN→CUDA→NVIDIA GPU Driver→NVIDIA GPU。因此，安装顺序应该是：先安装 GPU 驱动，再安装 CUDA 和 cuDNN，最后安装对应版本的 PyTorch。

3. GPU 版本 PyTorch 的安装步骤

接下来，我们讲解一下 GPU 版本 PyTorch 的安装步骤。

步骤 01 检查你的 NVIDIA GPU 是否支持 CUDA。确保你的计算机拥有 NVIDIA 显卡，并且它支持 CUDA。几乎所有现代的 NVIDIA GPU（GeForce、RTX、Quadro、Tesla 等）都支持 CUDA。

官方支持列表请查看 https://developer.nvidia.com/cuda-gpus。

步骤02 查看你的显卡驱动版本，确定你需要安装哪个版本的 CUDA。打开 Windows 终端管理员窗口，执行命令：nvidia-smi，在结果中查看右上角显示的驱动版本和最高支持的 CUDA 版本。例如，输出中有一行 CUDA Version:12.8，这表示你的当前驱动最高可以支持 CUDA 12.8。你需要安装版本不高于 12.8 的 CUDA。

步骤03 检查 GPU 是否支持 CUDA。访问 NVIDIA 官网查看你的 GPU 型号是否在支持列表中，或在命令行窗口输入命令：wmic path win32_VideoController get name 来检查。

步骤04 下载 CUDA Toolkit。访问 NVIDIA CUDA 下载页面：https://developer.nvidia.com/cuda-toolkit-archive。选择适合你系统的版本，建议选择 PyTorch 支持的版本，比如 12.6。

步骤05 安装 CUDA。运行下载的.exe 文件，建议使用默认安装路径。安装过程中确保勾选 CUDA Toolkit。

步骤06 验证 CUDA 是否安装成功。在终端管理员窗口执行命令：nvcc -V，若结果显示版本信息，则表示安装成功。

步骤07 接下来下载 cuDNN。cuDNN 是 NVIDIA 的深度神经网络加速库，是 PyTorch GPU 加速的必要组件，下载地址为 https://developer.nvidia.com/rdp/cudnn-download。下载时，记得选择与已安装 CUDA 版本匹配的 cuDNN 版本。

步骤08 安装 cuDNN。解压下载的 ZIP 文件。将 bin、include、lib 文件夹复制到 CUDA 安装目录（例如，默认的目录为 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6）下。

步骤09 检查 cuDNN 版本。查看 CUDA 安装目录下的 cudnn_version.h 文件。

步骤10 打开 Miniconda 提供的 Anaconda Prompt 窗口，安装 PyTorch，注意虚拟环境使用默认的 Base。访问 PyTorch 官网（https://pytorch.org/），根据系统、CUDA 版本选择合适的安装命令，比如 PyTorch 2.8.0 + CUDA 12.6 的安装命令为：

```
    # CUDA 12.6
    pip install torch==2.8.0 torchvision==0.23.0 torchaudio==2.8.0 --index-url
https://download.pytorch.org/whl/cu126
```

步骤11 安装成功后，验证 PyTorch 安装。打开 Python 终端，输入以下代码：

```
    import torch
    print(torch.__version__)
    print(torch.cuda.is_available())            # 输出 True 表示 GPU 可用
    print(torch.backends.cudnn.version())       # 输出 cuDNN 版本
```

如果没有报错且 torch.cuda.is_available()返回 True，则表示安装成功。

2.1.3 PyCharm 的安装与虚拟环境搭建

和其他语言类似，Python 程序的编写可以使用 Windows 自带的编辑器。但是这种方式对于较为复杂的程序工程来说，容易混淆相互之间的层级和交互文件，因此在编写程序时，我们可以使用专用的 Python 编译器 PyCharm。

（1）安装 PyCharm。由于其安装比较简单，就不展开讲解了。读者上网搜索 PyCharm 官网，进入 Download 页面，选择适合自己操作系统的版本安装即可。PyCharm 下载页面如图 2.3 所示。

（2）完成 PyCharm 的安装后，接下来可以搭建本书示例源码运行的虚拟环境。打开 Miniconda

提供的 Anaconda Prompt 窗口，输入如下命令，创建名为 langgraph 的虚拟环境并激活：

```
conda env list
conda create --name langgraph python=3.12    #这个命令用于创建 langgraph 虚拟环境
conda activate langgraph #这个命令激活 langgraph 虚拟环境
```

图 2.3　PyCharm 下载页面

（3）在 langgraph 虚拟环境中，安装本书配套源码的依赖文件。安装的库比较多，需要一点时间：

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

（4）如果安装 PyTorch 有误或者不工作，可以根据自己 NVIDIA 显卡的具体情况以及 2.1.1 节有关 PyTorch 的介绍，在 langgraph 虚拟环境中重新安装一下 PyTorch。

（5）下载本书配套代码，解压保存到 PyCharm 项目目录 PycharmProjects 中。

（6）在 PyCharm 中打开本书配套代码，设置虚拟环境，界面如图 2.4 所示。至此，本书示例源码运行环境就基本搭建好了。

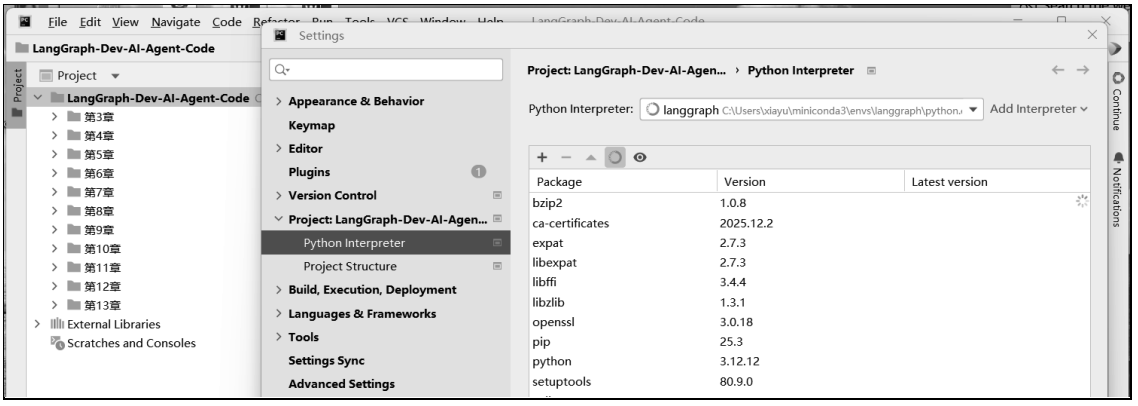

图 2.4　在 PyCharm 中设置代码虚拟运行环境

2.2 LLM 的调用与使用示例

本节将以 ModelScope（魔搭社区）中的 Qwen3 大模型为例，介绍 LLM 的调用与使用。

2.2.1 ModelScope（魔搭社区）

ModelScope（类似于 Hugging Face）是由阿里巴巴集团牵头，联合多家中国顶尖科研机构和高校共同推出的下一代“模型即服务”（Model-as-a-Service，MaaS）共享平台。它的核心目标是降低人工智能模型的应用门槛，促进开源模型的共享、协作与创新。

1. 核心定位与愿景

ModelScope 的愿景是成为 AI 模型领域的 GitHub。就像 GitHub 托管代码一样，ModelScope 致力于托管、分享和运行 AI 模型。它不仅仅是一个模型仓库，更是一个提供从模型探索、体验、微调到部署的一站式服务的生态系统。

其核心用户包括：

- AI 研究人员：发布和验证自己的模型。
- 应用开发者：快速找到并集成现成的模型到自己的应用中，无须从零开始训练。
- 学生和爱好者：学习最前沿的 AI 技术，动手实践。
- 企业用户：寻找商业化的解决方案，降低 AI 研发成本。

2. 主要功能与核心组成部分

ModelScope 社区提供了从模型探索、体验、部署到再开发的全链路服务。

1）庞大的模型库

（1）ModelScope 汇聚了来自阿里巴巴、清华大学、北京大学、浙江大学、商汤科技、澜舟科技等众多顶尖 AI 实验室和高校的开源模型。

（2）覆盖领域广泛：包括自然语言处理（NLP）、计算机视觉（CV）、语音识别与合成、多模态、科学计算等。

（3）模型类型多样：从基础的图像分类、文本 Embedding，到大型语言模型（如 Qwen、Baichuan）、文生图模型（如 Taiyi、SD）、语音合成模型等。

2）在线体验与 Demo

几乎每一个模型都提供了在线试玩（Playground）功能。用户无须安装任何环境，直接在网页上上传图片、输入文本或录音，即可立即看到模型的推理效果，极大地降低了模型体验的门槛。

3）Notebook 开发环境

平台提供了集成好的、云端免费的 Jupyter Notebook 开发环境（基于 PAI-DSW），预装了 ModelScope SDK 和常用依赖。用户可以直接在浏览器中打开 Notebook，编写几行代码即可加载和运行模型，进行原型开发和实验。

4）ModelScope Library（Python SDK）

这是 ModelScope 的核心组件，一个开源 Python 库。通过它，开发者可以用极其简洁的代码（通常只需 2~3 行）在本地或云端环境中下载、加载和推理模型。

示例代码如下：

```
   from modelscope.pipelines import pipeline
   from modelscope.utils.constant import Tasks
   # 创建文本摘要 pipeline
   pipe = pipeline(task=Tasks.text_summarization,
model='damo/nlp_bert_document-segmentation_chinese-base')
   # 输入文本并获取结果
   result = pipe('这里是需要摘要的长篇文章内容...')
   print(result)
```

5）ModelScope Studio

类似于 Hugging Face 的 Spaces，ModelScope Studio（创空间）是一个低代码/无代码的应用构建平台。用户可以利用 Gradio、Streamlit 等框架，快速为自己的模型构建一个功能丰富、界面友好的演示应用，并一键部署和分享给他人。

6）数据集与学习资源

（1）除了模型外，平台还提供了与模型配套使用的训练和评测数据集。

（2）包含丰富的教程、文档、技术博客和视频，帮助用户从入门到精通。

2.2.2　阿里云百炼 Qwen3 的在线调用

首先我们需要登录 Qwen3 官方网站“阿里云百炼”，页面菜单如图 2.5 所示。

图 2.5　“阿里云百炼”菜单

注册并登录后，进入百炼控制台，在页面上单击“大模型服务平台百炼”，打开“阿里云百炼”主页面，再单击“模型服务”，页面左下角有个“密钥管理”，如图 2.6 所示。

图 2.6　获取 API Key

单击“密钥管理”后，进入“密钥管理”页面，在这里既可以创建新的 API Key，也可以查询以前创建的 API Key，如图 2.7 所示。

接下来，单击“模型广场”，在“模型广场”页面选择想要开通的模型，如图 2.8 所示。

对于不同的任务需求，Qwen3 给我们准备了不同的 API 调用示例。例如，在“模型广场”上单击“通义千问 3-Max”，进入模型页面，其中提供了“API 代码示例”，如图 2.9 所示。

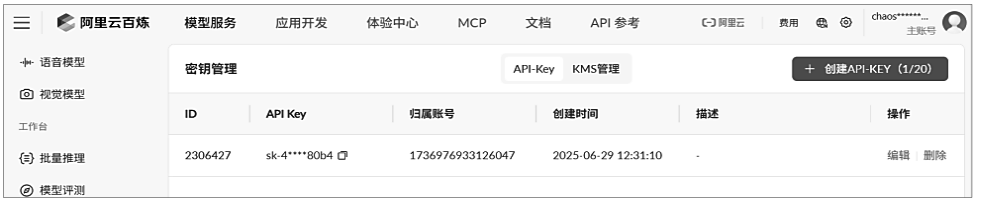

图 2.7　已创建的 API Key

图 2.8　单击通义千问目录

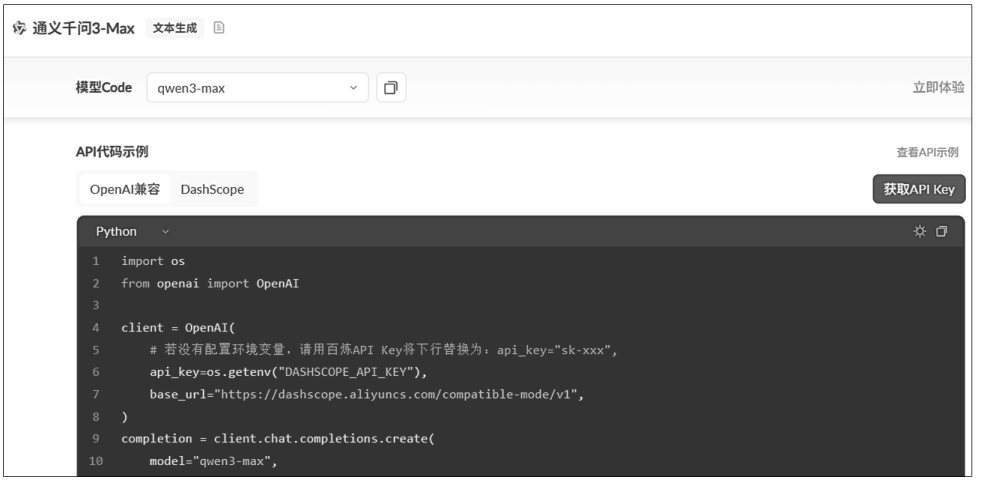

图 2.9　Qwen3 在线 API 调用示例

【示例 2.1】在线 API 调用 Qwen3（qwen3-online-call.py）。

```
import os
```

```
from openai import OpenAI

client = OpenAI(
    # 注意，阿里云百炼 API Key 配置到系统的环境变量 DASHSCOPE API KEY 中
    api key=os.getenv("DASHSCOPE API KEY"),
    base url="https://dashscope.aliyuncs.com/compatible-mode/v1",
)

completion = client.chat.completions.create(
    # 模型列表：https://help.aliyun.com/zh/model-studio/getting-started/models
    model="qwen-plus",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "你是谁？"},
    ],
    # Qwen3 模型通过 enable_thinking 参数控制思考过程（开源版默认为 True，商业版默认为 False）
    # 使用 Qwen3 开源版模型时，若未启用流式输出，请将下一行取消注释，否则会报错
    # extra_body={"enable_thinking": False},
)
print(completion.model_dump_json())
```

运行代码，输出如下：

```
{"id":"chatcmpl-960cac6e-fd83-97f9-9eec-98e68b15a88b","choices":[{"finish_reason":"stop","index":0,"logprobs":null,"message":{"content":"你好！我是通义千问（Qwen），阿里巴巴集团旗下的超大规模语言模型。我能够回答问题、创作文字，比如写故事、写公文、写邮件、写剧本、逻辑推理、编程等等，还能表达观点，玩游戏等。如果你有任何问题或需要帮助，欢迎随时告诉我！😊","refusal":null,"role":"assistant","annotations":null,"audio":null,"function_call":null,"tool_calls":null}}],"created":1774768706,"model":"qwen-plus","object":"chat.completion","service_tier":null,"system_fingerprint":null,"usage":{"completion_tokens":66,"prompt_tokens":22,"total_tokens":88,"completion_tokens_details":null,"prompt_tokens_details":{"audio_tokens":null,"cached_tokens":0}}}
```

读者可以将上面代码中的 API Key 替换成自己的。由于 Qwen3 模型包含思考过程，读者同时还可以自定义思考过程的输出方式。下面是作者重写的参数说明。

（1）model：字符串，必选，指定模型名称。支持通义千问大语言模型（商业版、开源版、Qwen-Long）、通义千问 VL、通义千问 Omni、数学模型、代码模型。需要注意的是，通义千问 Audio 暂不支持 OpenAI 兼容模式，仅支持 DashScope 方式。具体模型名称及计费规则详见模型列表文档。

（2）messages：数组，必选，对话组成的消息列表。其包含以下消息类型。

- SystemMessage：对象，可选，定义模型目标或角色。若设置，则需置于 messages 列表首位。但 QwQ 模型不建议设置，QVQ 模型设置后不生效。
- UserMessage：对象，必选，表示用户发送给模型的消息内容。
- AssistantMessage：对象，可选，记录模型对用户消息的回复。
- ToolMessage：对象，可选，承载工具的输出信息。

（3）stream：布尔值，可选，默认为 false，控制是否流式输出回复。

- false：模型生成完整内容后一次性返回。
- true：逐片段输出（chunk），需实时读取以获取完整结果。仅 Qwen3 商业版（思考模式）、Qwen3 开源版、QwQ、QVQ 支持流式输出。

（4）stream_options：当 stream=true 时，可通过设置{"include_usage": true}在输出末行显示 token 使用量，设为 false 则隐藏该信息。

（5）modalities：仅 Qwen-Omni 模型支持指定输出模态。

- ["text","audio"]：同时输出文本与音频。
- ["text"]：仅输出文本。

（6）temperature：浮点数，可选，控制生成文本的多样性。值越高，结果越随机（范围为[0, 2)）。建议与 top_p 二选一设置，QVQ 模型默认值不建议修改。

（7）top_p：浮点数，可选，采样阈值。值越高，结果越随机（范围为(0, 1.0]）。建议与 temperature 二选一设置，QVQ 模型默认值不建议修改。

（8）top_k：整数，可选，采样候选集大小（≥0）。值越大，随机性越高。设为 None 或大于 100 时仅 top_p 生效。QVQ 模型默认值不建议修改，Python SDK 需通过 extra_body 配置。

（9）presence_penalty：控制内容重复度（范围为[−2.0, 2.0]）。

- 正值减少重复（适合创意场景）。
- 负值增加重复（适合专业文档）。

示例：qwen-vl-plus 系列模型文字提取建议设置为 1.5，QVQ 模型默认值不建议修改。

（10）response_format：指定返回格式。

```
{"type":"text"}：纯文本。
{"type":"json_object"}：结构化 JSON（需在消息中提示模型输出 JSON 格式）。
```

（11）max_tokens：限制返回最大 Token 数，超限内容将被截断。默认值及上限参考模型列表，qwen-vl-ocr 系列默认为 2048，最大为 8192。QwQ/QVQ 模型仅限制回复长度，不限制思考内容。

（12）n：integer（可选）默认值为 1，生成响应的数量，取值范围是 1~4，适用于多结果场景（如创意写作）。仅 qwen-plus 及非思考模式 Qwen3 支持，且传入 tools 时固定为 1。

（13）enable_thinking：控制 Qwen3 模型思考模式，需通过 extra_body 配置。商业版默认为 false，开源版默认为 true。

（14）thinking_budget：设置思考过程最大长度，仅 enable_thinking=true 时生效，适用于 Qwen3 商业版及开源版。

（15）seed：设置随机种子（0~$2^{31}-1$）以获得确定性结果，相同 seed 和其他参数将生成相同内容。

（16）stop：指定终止生成字符串或 token_id，可用于敏感词过滤。数组类型不支持混合 token_id 与字符串。

（17）tools：定义可调用工具列表，当前不支持通义千问 VL/Audio 及数学/代码模型。每个工具对象包含：

- tool_choice：字符串/对象，可选，默认为"auto"。控制工具调用策略，可选"auto" "none"或指定工具名称。
- parallel_tool_calls：boolean（可选），默认值为 false，表示是否开启并行工具调用。相关文档：并行工具调用。可选值：true 表示开启，false 表示不开启。

（18）translation_options：翻译模型专用配置参数，需通过 extra_body 传递。

（19）enable_search：控制是否启用互联网搜索。

- true：允许模型参考搜索结果（可能增加 Token 消耗）。
- false：禁用搜索，支持强制搜索配置。

（20）search_options：对象，可选，联网搜索策略配置，仅 enable_search=true 时生效。

不同参数定义了用户在使用 Qwen3 在线 API 时能够进行的操作。除了基本的文本输入输出外，对于更多的使用示例和样式，读者可以自行验证。

2.3　实战案例：创建一个基础聊天机器人

基于 LangGraph 搭建状态机框架，定义对话状态与节点，对接阿里云百炼提供的 DeepSeek 模型构建基础对话流程。

2.3.1　创建调用 deepseek-v3 的聊天机器人

现在，让我们开始创建一个基础的聊天机器人。首先创建一个名为 basic-chatbot.py 的文件，并添加以下代码。

【示例 2.2】基础聊天机器人实现代码（basic-chatbot.py）。

```
from typing import Annotated
from langchain_openai import ChatOpenAI
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START
from langgraph.graph.message import add_messages

# 启用 dotenv 读取环境变量
import os
from dotenv import load_dotenv

# 加载 .env 文件中的环境变量
load_dotenv()

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)
```

```
    # 从环境变量读取 API Key
    llm = ChatOpenAI(
        model="deepseek-v3",
        api key=os.getenv("DASHSCOPE API KEY"),  # 从 .env 读取
        base url="https://dashscope.aliyuncs.com/compatible-mode/v1",
        temperature=0
    )

    def chatbot(state: State):
        return {"messages": [llm.invoke(state["messages"])]}

    # 添加节点和边
    graph_builder.add_node("chatbot", chatbot)
    graph_builder.add_edge(START, "chatbot")
    graph = graph_builder.compile()

    def stream_graph_updates(user_input: str):
        for event in graph.stream({"messages": [{"role": "user", "content":
user_input}]}):
            for value in event.values():
                print("Assistant:", value["messages"][-1].content)

    while True:
        try:
            user_input = input("User: ")
            if user_input.lower() in ["quit", "exit", "q"]:
                print("Goodbye!")
                break
            stream_graph_updates(user_input)
        except KeyboardInterrupt:
            print("\nGoodbye!")
            break
```

运行输出（可能与此处给出的结果存在区别）：

```
    User: 请问南果梨的产地与水果特色?
    Assistant: 南果梨是一种具有独特风味和地域特色的水果，以下是关于其产地和特色的详细介绍：

    ### 产地
    1. 核心产区
       - 辽宁省鞍山市：尤其是海城市、千山区（原旧堡区）及周边地区，是南果梨最著名的产地，这里的
气候和土壤条件（偏酸性棕壤土）非常适宜其生长。
       - 其他地区：辽宁省的辽阳、营口等地也有种植，但品质和风味以鞍山产的最为突出。

    2. 生长环境
       - 南果梨喜昼夜温差大的温带气候，鞍山地区秋季光照充足、温差显著，利于糖分积累，形成独特香
气。

    ### 水果特色
    1. 外观与口感
       - 外形：果实较小（单果约 50~100 克），成熟时果皮黄绿色带红晕，表面有细小果点。
```

```
        - 质地：刚采摘时脆硬，后熟后果肉变软糯，细腻多汁，入口即化。
        - 风味：酸甜适中，具有浓郁的复合果香（类似香蕉、玫瑰的香气），甜度可达 14%~16%。

    2. 独特后熟特性
        - 需常温放置 3~7 天后熟，果肉由硬变软，香气充分释放，此时为最佳食用期。

    3. 营养价值
        - 富含维生素 C、花青素、矿物质（如钙、铁），有抗氧化作用，有助于促进消化。

    4. 文化地位
        - 被列为中国四大名梨之一（与库尔勒香梨、莱阳梨等并列），2005 年成为国家地理标志产品，是鞍山的农业名片。

    ### 其他信息
    - 采收与保鲜：9 月上旬成熟，常温保存约 1~2 周，冷藏可延长至 1 个月，但后熟后需尽快食用。
    - 深加工：常用于制作果汁、果酒、果脯等产品，进一步拓展其经济价值。

    南果梨以其“香气袭人、口感独特”著称，是兼具地域性和品质优势的特色水果，适合喜欢酸甜风味和馥郁果香的人群品尝。
    User:
```

2.3.2　案例代码解析

1. 导入必要的库

这部分代码的作用是引入程序运行所依赖的所有工具和框架。

```
from typing import Annotated

from langchain_openai import ChatOpenAI
from typing_extensions import TypedDict

from langgraph.graph import StateGraph, START
from langgraph.graph.message import add_messages

import os
from dotenv import load_dotenv
```

（1）from typing import Annotated：这是 Python 3.9+提供的类型提示增强功能。它允许为变量或函数参数添加元数据（注解）。在 LangGraph 中，它被用来为状态中的某些字段（如 messages）附加特殊的行为。

（2）from langchain_openai import ChatOpenAI：这是 LangChain 库中用于初始化大模型的统一入口函数。它可以根据提供的模型名称（如"deepseek-v3"）自动选择并配置对应的模型封装。

（3）from typing_extensions import TypedDict：用于创建强类型的字典。在 LangGraph 中，它是定义工作流状态（State）结构的标准方式，能让状态的每个字段都有明确的类型定义，提高代码可读性和健壮性。

（4）from langgraph.graph import StateGraph,START：StateGraph 是 LangGraph 的核心类，用于构建一个由节点和边组成的有状态工作流（图）。START 是一个特殊的常量，代表图中的起始节点。

（5）from langgraph.graph.message import add_messages：这是一个 LangGraph 提供的状态更新策略。它定义了当新的消息产生时，应该如何更新状态中的 messages 列表。具体来说，它会将新消息追加到列表末尾，并自动处理 ai 和 human 角色的消息封装。

（6）import os：Python 内置的用于与操作系统交互的库。这里主要用来获取环境变量。

（7）from dotenv import load_dotenv：一个第三方库，用于从.env 文件中加载环境变量到程序的运行环境中。这是管理 API 密钥等敏感信息的最佳实践。

2. 加载环境变量

```
# 加载.env 文件中的环境变量
load_dotenv()
```

这行代码会查找当前目录下名为.env 的文件，并将文件中定义的键值对（如 DASHSCOPE_API_KEY=sk-xxx）加载到 Python 的 os.environ 字典中，目的是安全地管理敏感信息（如 API 密钥），避免将它们硬编码在代码中。

3. 定义状态（State）

```
class State(TypedDict):
    messages: Annotated[list, add_messages]
```

这是整个工作流的核心数据结构，它定义了在图的各个节点之间传递和修改的数据。

（1）class State(TypedDict)语句定义了一个名为 State 的类，它继承自 TypedDict。这意味着 State 的实例是一个字典，但它的键（messages）和对应的值类型（list）是固定的。

（2）messages:Annotated[list, add_messages]语句中，messages 是状态中唯一的字段，它将存储一个消息列表，每个消息都是一个字典（如{"role":"user","content":"你好"}）。Annotated[list, add_messages]在这里发挥了关键作用，它告诉 LangGraph，messages 字段是一个列表，当有新的消息需要更新到这个字段时，应该使用 add_messages 策略。add_messages 会智能地将新消息追加到现有列表的末尾，确保对话历史的连续性。

4. 创建图构建器

```
graph_builder = StateGraph(State)
```

这行代码创建了一个 StateGraph 的实例，命名为 graph_builder。在创建时，我们将刚刚定义的 State 类作为参数传入。这告诉图构建器，整个工作流将围绕 State 这种数据结构来展开。graph_builder 将负责管理节点、边以及状态在它们之间的流转。

5. 初始化模型

```
llm = ChatOpenAI(
    model="deepseek-v3",
    api_key=os.getenv("DASHSCOPE_API_KEY"),  # 从 .env 读取
    base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
    temperature=0
)
```

这行代码初始化了一个 DeepSeek 模型（deepseek-v3 模型）。

（1）ChatOpenAI("deepseek-v3",...)：ChatOpenAI 函数根据"deepseek-v3"这个字符串，自动识别并调用 LangChain 中与阿里云百炼提供的 DeepSeek 模型对应的封装。

（2）api_key=os.getenv("DASHSCOPE_API_KEY")：从环境变量中获取 DASHSCOPE 的 API 密钥，并传递给模型。如果环境变量未设置，os.getenv()会返回 None，可能导致模型初始化失败。

6. 定义节点函数（Node Function）

```
def chatbot(state: State):
    return {"messages": [llm.invoke(state["messages"])]}
```

这是一个节点函数，它定义了图中一个具体的处理步骤。当工作流执行到这个节点时，这个函数就会被调用。

（1）chatbot(state: State)：函数接收一个 state 参数，它的类型是我们之前定义的 State。这个 state 包含了当前对话的所有信息（主要是 messages 列表）。

```
llm.invoke(state["messages"]):#这是函数的核心逻辑
```

它将 state 中的 messages 列表（即完整的对话历史）传递给大模型 llm。模型会根据这些对话历史生成一个新的回复。

（2）return {"messages":[...]}：函数的返回值是一个字典。这个字典的结构必须与 State 类型相匹配。它告诉图构建器 graph_builder：“请用我返回的这个新字典来更新全局状态”。

- llm.invoke(...)返回的是一个 ChatMessage 对象。
- [llm.invoke(...)]将这个对象放入一个列表中。
- {"messages":[...]}表示：“我要更新 State 中的 messages 字段，新的值是一个包含 AI 回复的列表”。

由于我们在 State 定义中使用了 add_messages 注解，LangGraph 会自动将这个新列表中的消息追加到原有的 messages 列表后面，而不是替换它。

7. 构建图（Graph）

```
# 添加节点和边
graph_builder.add_node("chatbot", chatbot)
graph_builder.add_edge(START, "chatbot")
graph = graph_builder.compile()
```

这部分代码负责将节点和它们之间的连接关系组装成一个可执行的图。

- graph_builder.add_node("chatbot",chatbot): 向图中添加一个名为"chatbot"的节点，这个节点的执行逻辑由我们定义的 chatbot 函数来实现。
- graph_builder.add_edge(START,"chatbot"): 定义了一条从 START 节点到"chatbot"节点的边。这告诉图执行引擎：“工作流开始时，请首先执行'chatbot'节点”。
- graph = graph_builder.compile(): 这是最后一步，也是关键的一步。它将 graph_builder 对象编译成一个最终可执行的 graph 对象。在这个阶段，LangGraph 会进行一些内部优化和验证，确保图的结构是正确的。

8. 实现流式响应

```
    def stream_graph_updates(user_input: str):
        for event in graph.stream({"messages": [{"role": "user", "content":
user_input}]}):
            for value in event.values():
                print("Assistant:", value["messages"][-1].content)
```

这个函数封装了如何与已编译好的 graph 进行交互，以实现流式输出（即模型边生成、程序边打印）。

- graph.stream(...)：调用 graph 对象的 stream 方法来执行图。与 invoke 方法一次性获取所有结果不同，stream 会以流式生成的方式逐步返回执行过程中的更新。
- {"messages":[{"role":"user","content":user_input}]}：这是启动图时传入的初始状态。它是一个 State 类型的字典，包含用户的最新输入。
- for event in graph.stream(...)：遍历流式返回的事件（event）。每个 event 代表图执行过程中的一个状态更新。
- for value in event.values(): 每个 event 可能包含多个节点的更新，但在我们这个简单的图中，每次更新只来自"chatbot"节点。
- print("Assistant:",value["messages"][-1].content):
  - value：是更新后的部分状态（一个 State 字典）。
  - value：["messages"]是更新后的消息列表。
  - [-1]：取列表中的最后一个元素，也就是大模型刚刚生成的最新一条消息。
  - .content：获取该消息的内容并打印。

9. 主循环（Main Loop）

```
    while True:
        try:
            user_input = input("User: ")
            if user_input.lower() in ["quit", "exit", "q"]:
                print("Goodbye!")
                break
            stream_graph_updates(user_input)
        except KeyboardInterrupt:
            print("\nGoodbye!")
            break
```

这是程序的入口点，一个标准的无限循环，用于持续与用户交互。

- while True：启动一个无限循环。
- user_input = input("User: ")：在控制台打印"User:"提示符，并等待用户输入。
- if user_input.lower() in ["quit","exit","q"]:break：如果用户输入了退出（exit）指令，打印告别信息并跳出循环，程序结束。
- stream_graph_updates(user_input)：如果用户输入了正常内容，则调用我们定义的流式响应函数，将用户输入传递给 graph 并处理和打印 AI 的回复。

- except KeyboardInterrupt：捕获用户按下 Ctrl+C 键的中断信号，同样打印告别信息并优雅地退出程序。

10. 总结

这段代码完整地演示了如何使用 LangGraph 构建一个最简单的、具有状态记忆（对话历史）的聊天机器人。

- 核心思想：将对话流程建模为一个状态（State）在图（Graph）中流转的过程。
- 状态（State）：State 类定义了对话的全部数据（messages）。
- 节点（Node）：chatbot 函数定义了具体的处理逻辑（调用 LLM）。
- 图（Graph）：StateGraph 将状态和节点组织起来，形成一个可执行的工作流。
- 执行：通过 graph.stream()触发整个流程，并以流式方式获取结果。

2.3.3　运行聊天机器人

现在可以运行这个简单的聊天机器人了。确保你已经在.env 文件中设置了有效的 DeepSeek API 密钥，然后在 Miniconda Prompt 命令行窗口中运行：

```
python basic-chatbot.py
```

或者直接在 PyCharm 中运行此代码，你将看到一个交互式的命令行界面，通过此界面可以与聊天机器人对话：

```
User: 请问南果梨的产地与水果特色?
Assistant: 当然！南果梨是中国非常有特色的一种水果，以其独特的香气和口感而闻名。以下是关于它的产地和水果特色的详细介绍：

一、核心产地

南果梨最著名、最核心的产地是辽宁省鞍山市的千山区（特别是大屯镇和接文镇）以及海城市。
...
User: exit
Goodbye!
```

2.4　本章小结

本章系统地介绍了大模型 Agent 开发的环境配置，为解决“用什么写”和“怎么写”奠定了基础。本书选用 Python3 作为核心语言，因其具备成熟的科学计算生态、与主流深度学习框架的深度集成以及低教学成本三大优势。为克服原生 Python 的环境管理难题，本章推荐使用 Miniconda 发行版，其开箱即用的科学计算库、强大的环境隔离能力和跨平台一致性为开发提供了极大便利。随后，本章还阐述了 PyTorch 框架因其动态计算图带来的灵活性与高效性。

通过指导读者完成从 Miniconda 安装、PyTorch 环境安装、PyCharm 安装与使用，到 Qwen3 模型在线 API 调用等全流程，本章旨在让读者初步掌握搭建一个功能完备且高效的 Agent 开发环境的

具体方法，为后续的开发铺平道路。

最后实现一个基础聊天机器人：基于 LangGraph 搭建状态机框架，定义对话状态与节点，对接 DeepSeek 模型构建基础对话流程。这个简洁示例清晰呈现了从环境搭建到功能落地的完整流程，帮助读者快速掌握 LangGraph 集成大模型开发聊天机器人的核心方法。