一、AI Agent

1.定义

> AI Agent 不是一个简单的聊天机器人，而是具有目标导向性，自主性和自适应的智能体。AI Agent是一个能够感知环境，自主决策，执行动作，并且持续追寻目标的智能系统。
>
> 核心思想：**AI Agent = LLM（大脑） + Memory（记忆） + Tools（手脚） + Planning（思维） + Loop（迭代）**

2.四大能力

能力	说明	技术实现示例
感知（**Perception**）	获取外部信息，理解上下文	接收用户输入、读取文件、调用API、解析网页
规划（**Planning**）	对目标进行拆分步骤	任务分解（Task Decomposition）、思维链（Chain-of-Thought）
记忆（**memory**）	存储和调用历史信息	短期记忆（上下文）、长期记忆（调用向量数据库）
工具使用（**Tools Use**）	调用外部能力扩展自身功能	搜索、代码执行、数据库查询、发送邮件等

> ☑ **关键区别**：传统AI模型（如GPT）是“被动响应”，而AI Agent是“主动执行”。

3.基本构架

```
用户输入
  ↓
感知模板          （用户输入的文本，图片，视频等）
  ↓
规划模块          （对用户的需求，进行任务拆解和规划）
  ↓
决策和推理模块    （LLM作为大脑进行思考和决策 ）
  ↓
工具调用模块      （调用外部工具，比如调用搜索、代码解释器、API等）
  ↓
执行动作模块      （输出结果、修改状态、发送请求）
  ↓
记忆模块          （存储上下文、经验、知识）
  ↓
环境反馈模块      （观察执行结果，决定是否重试或调整）
```

4.核心能力详解

1. LLM 作为“大脑”（Reasoning Engine）

- Agent 的核心是大语言模型（如 GPT-4、Claude、Llama）。
- 它负责：
  - 理解用户意图
  - 生成计划
  - 做出决策
  - 调用工具的判断
- 关键机制：
  - **Function Calling / Tool Calling**：让LLM“知道”它可以调用哪些工具。

2.记忆（Memory）

类型	说明	示例
短期记忆	保存当前会话上下文	`ConversationBufferMemory`
长期记忆	跨会话、跨任务的知识存储	向量数据库（Chroma, Pinecone）
实体记忆	记住关键实体（如人名、地点）	记录“用户喜欢科技类文章”

> 📌 记忆是Agent“成长”的基础，避免每次从零开始。

3. 工具（Tools）

Agent 通过工具扩展能力，常见工具包括：

- `SearchTool`：联网搜索（如Tavily、SerpAPI）
- `CodeInterpreterTool`：执行Python代码（计算、绘图、数据处理）
- `GmailTool`：发送邮件
- `FileTool`：读写文件
- `Calculator`：数学计算
- `Custom API`：调用企业内部系统

> ✅ Agent 的能力 = LLM + 工具库

4. 规划（Planning）

Agent 如何思考和拆解任务？常见策略：

策略	说明
单步循环（**Zero-shot ReAct**）	LLM 直接思考 → 行动 → 观察 → 再思考（单步循环）
多路最优选择（**Tree of Thoughts (ToT)**）	生成多个可能思路，评估后选择最优路径
大任务拆解小任务（**Task Decomposition**）	将大任务拆分为子任务（如“写报告” → 查资料 → 写大纲 → 写正文）
自我思考，重试（**Reflexion**）	执行失败后自我反思，调整策略重试

5. 状态管理（State Management）

- 在复杂任务中，Agent 需要维护一个

  共享状态（State），记录：

  - 当前任务
  - 已完成步骤
  - 中间结果
  - 错误日志
- **LangGraph** 正是基于“状态机”（State Machine）来管理这种复杂流程。

5.AI Agent 的类型

类型	特点	示例
反应式 Agent	感知 → 立即行动，无长期记忆	简单客服机器人
基于模型的 Agent	维护内部状态模型，能处理部分历史	带记忆的聊天机器人
目标驱动 Agent	为达成目标自主规划和执行	AutoGPT、BabyAGI
多 Agent 系统	多个Agent协作完成任务	Researcher + Writer + Reviewer 团队
自治 Agent	长期运行，自我维护和优化	数字员工、AI助手

6.AI Agent 的挑战与局限

挑战	说明
幻觉（Hallucination）	LLM 可能编造虚假信息或错误调用工具
效率问题	多次调用LLM导致延迟和成本高
安全性	可能执行危险操作（如删除文件、发送错误邮件）
可解释性差	“黑箱”决策，难以追踪错误原因
长期稳定性	复杂任务可能陷入死循环或崩溃

7.AI Agent 的应用场景

领域	应用案例
办公自动化	自动生成周报、会议纪要、PPT
客户服务	智能客服、订单查询、问题解决
研发辅助	代码生成、Bug 修复、文档编写
数据分析	自动爬取数据、清洗、分析、可视化
内容创作	写文章、脚本、营销文案、小说
个人助理	安排行程、订餐、查天气、学习辅导
金融投资	股票分析、风险评估、自动化交易

二、langchian

> **LangChain** 是一个用于开发由大型语言模型（LLMs）驱动的应用程序的框架。

1.架构

> LangChain 是一个由多个包组成的框架

langchain-core

> 此包包含不同组件的基础抽象以及将它们组合在一起的方式。核心组件（如聊天模型、向量存储、工具等）的接口在此处定义。此处未定义任何第三方集成。依赖项非常轻量。

langchain

> 主 `langchain` 包包含构成应用程序认知架构的链和检索策略。这些不是第三方集成。此处的所有链、代理和检索策略并非特定于任何一个集成，而是适用于所有集成的通用策略。

集成软件包

> 流行的集成有它们自己的包（例如 `langchain-openai`、`langchain-anthropic` 等），以便它们可以正确地进行版本控制并保持适当的轻量级。

更多信息请参阅

- 集成包列表
- 您可以在其中找到每个集成包详细信息的 API 参考。

langchain-community

> 此包包含由 LangChain 社区维护的第三方集成。关键集成包已分离出来（参见上文）。这包含各种组件（聊天模型、向量存储、工具等）的集成。此包中的所有依赖项都是可选的，以尽可能保持包的轻量级。

langgraph

> `langgraph` 是 `langchain` 的一个扩展，旨在通过将步骤建模为图中的边和节点，使用 LLM 构建健壮的有状态多代理应用程序。LangGraph 提供了用于创建常见类型代理的高级接口，以及用于组合自定义流程的低级 API。

延伸阅读

- 查看我们的 LangGraph 概述 此处。

- 查看我们的 LangGraph 学院课程 此处。

langserve

> 一个用于将 LangChain 链部署为 REST API 的包。这使得部署生产就绪的 API 变得容易。

重要

LangServe 主要设计用于部署简单的 Runnable 并与 langchain-core 中众所周知的原语协同工作。

如果您需要 LangGraph 的部署选项，您应该查看 LangGraph 平台（测试版），它更适合部署 LangGraph 应用程序。

更多信息，请参阅 LangServe 文档。

LangSmith

> 一个开发者平台，可让您调试、测试、评估和监控 LLM 应用程序。

更多信息，请参阅 LangSmith 文档

2.核心模块

2.1 聊天模型（LLM）

2.1.1 什么是聊天模型?

聊天模型（Chat Model） 是现代大型语言模型（LLM）的主流交互方式。它接收一个**消息列表**作为输入，返回一条**消息**作为输出。

与传统的"字符串输入 → 字符串输出"模型不同，聊天模型更贴近真实对话，支持角色区分（如系统、用户、助手），并能处理多轮对话、工具调用等复杂场景。

> ✅ 提示：在 LangChain 中，"LLM" 和 "聊天模型" 常被互换使用，但技术上 **聊天模型更现代、功能更丰富。**

2.1.2 核心特性

LangChain 的聊天模型支持以下高级功能：

特性	说明
✅ **工具调用（Tool Calling）**	模型可调用外部函数、API、数据库等
✅ **结构化输出（Structured Output）**	强制模型返回 JSON、Pydantic 模型等格式
✅ **多模态（Multimodal）**	支持图像、音频等非文本输入（部分模型）
✅ **流式输出（Streaming）**	实时接收模型生成的文本片段
✅ **异步支持**	支持 `async/await` 高效处理并发请求
✅ **缓存与调试**	支持响应缓存、LangSmith 集成调试

2.1.3 支持的模型提供商

LangChain 支持多种主流模型，分为两类：

类型	说明	示例
官方模型	官方维护，功能完整	`langchain-openai`，`langchain-anthropic`
社区模型	社区贡献	`langchain-community` 包中

常见支持的聊天模型类名：

- `ChatOpenAI`：OpenAI 的 GPT 系列
- `ChatAnthropic`：Anthropic 的 Claude 系列
- `ChatOllama`：本地运行的 Ollama 模型
- `ChatGoogleVertex`：Google Vertex AI
- `ChatBedrock`：Amazon Bedrock

> ⚠ 注意：类名**带** `Chat` **前缀**的才是聊天模型接口。不带 `Chat` 或带 `LLM` 后缀的是旧版模型，**不推荐使用。**

2.1.4 标准参数（Standard Parameters）

参数	类型	说明
`model`	str	模型名称，如 `"gpt-3.5-turbo"`
`temperature`	float	输出随机性，0（确定）~ 1（随机）
`max_tokens`	int	最大输出 token 数
`timeout`	float	请求超时时间（秒）
`stop`	list[str]	停止生成的字符串序列
`max_retries`	int	失败后最大重试次数
`api_key`	str	认证密钥
`base_url`	str	自定义 API 地址（如本地部署）
`rate_limiter`	BaseRateLimiter	速率限制器，防限流

2.1.5 核心方法

方法	说明
`.invoke(messages)`	同步调用，返回单条消息
`.stream(messages)`	流式输出，逐字返回生成内容
`.batch(messages_list)`	批量处理多个请求
`.bind_tools(tools)`	绑定工具供模型调用
`.with_structured_output(schema)`	强制返回结构化数据

2.1.6 示例

直接调用

```
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_community.llms.ollama import Ollama

# 初始化模型
chat  = Ollama(
    model="deepseek-r1:1.5b",  # 使用的模型名称
    base_url="http://localhost:11434",  # Ollama 服务地址
    temperature=0.7  # 可选参数
)

# 构造消息
messages = [
    SystemMessage(content="你是一个乐于助人的助手。"),
    HumanMessage(content="中国的首都是哪里？")
]

# 调用模型
response = chat.invoke(messages)
print(response)
# 输出：中国的首都是北京。
```

调用工具

```
from langchain_core.messages import HumanMessage
from langchain_ollama import ChatOllama  # 注意：是 ChatOllama
from langchain_core.tools import tool

# 定义工具
@tool
def get_weather(location: str) -> str:
    """查询某个城市的天气"""
    return f"{location} 当前天气：晴，25°C"

# 初始化模型（使用 ChatOllama）
chat = ChatOllama(
    model="qwen3:1.7b",  # 推荐使用支持工具调用的模型，如 qwen:14b-chat, dolphin-
llama3 等
    base_url="http://localhost:11434",
    temperature=0.3,
).bind_tools([get_weather])

# 构造消息
messages = [HumanMessage(content="北京天气怎么样？")]

# 调用模型
ai_msg = chat.invoke(messages)

# 输出结果
print(ai_msg)
print("\n工具调用信息：")
print(ai_msg.tool_calls)
```

2.2 消息系统

2.2.1 消息的基本组成

每条消息通常包含以下信息：

1. **角色**（Role）：定义消息的类型和来源。
2. **内容**（Content）：消息的实际内容，可以是文本或结构化数据。
3. **元数据**（Metadata）：可选的附加信息，如 ID、名称、token 使用量等。

2.2.2 消息角色（Roles）

角色	描述
`system`	用于引导模型行为，设定对话上下文或角色（如“你是一个烹饪专家”）。并非所有模型都直接支持。
`user`	用户输入的内容，代表与模型交互的用户。
`assistant`	模型生成的响应，可能包含文本或工具调用请求。
`tool`	工具调用的结果返回给模型的消息。
`function`（旧版）	对应 OpenAI 的旧版函数调用 API，**应使用 `tool` 替代。**

2.2.3 主要消息类型

LangChain 提供了多种消息类，均继承自 `BaseMessage`。

1. `SystemMessage` - 系统消息

用于设定对话的初始上下文或行为准则。

```
from langchain_core.messages import SystemMessage

system_msg = SystemMessage(content="你是一个专业的厨师，擅长解释烹饪技巧。")
```

> **注意**：不同模型对系统消息的支持方式不同：
>
> - 有些通过 `role="system"` 直接支持。
> - 有些通过专用 API 参数传递。
> - 不支持的模型，LangChain 会尝试将其合并到 `HumanMessage` 中。

---

2. `HumanMessage` - 用户消息

代表用户的输入。

```
from langchain_core.messages import HumanMessage

human_msg = HumanMessage(content="如何煮意大利面？")
```

> **提示**：当直接传入字符串时，LangChain 会自动将其转换为 `HumanMessage`：

> Python深色版本
>
> ```
> model.invoke("Hello, how are you?")
> # 等价于 model.invoke([HumanMessage(content="Hello, how are you?")])
> ```

---

3. `AIMessage` - 助手消息

代表模型的响应，可能包含文本或工具调用。

```
from langchain_core.messages import HumanMessage, AIMessage

# 调用模型
ai_message = model.invoke([HumanMessage(content="讲个笑话吧")])
print(ai_message.content)
# 输出示例： "当然！稻草人为什么获奖？因为它在自己的领域表现出色！"
```

`AIMessage` 的主要属性：

属性	类型	描述
`content`	原始	响应内容，可以是字符串或字典列表（多模态）。
`tool_calls`	标准化	工具调用请求列表。
`invalid_tool_calls`	标准化	解析失败的工具调用。
`usage_metadata`	标准化	token 使用统计。
`id`	标准化	消息唯一标识符。
`response_metadata`	原始	模型特定的响应元数据。

---

4. `AIMessageChunk` - 流式响应块

用于流式传输模型响应，用户可以实时看到输出。

```
# 流式输出
for chunk in model.stream([HumanMessage("天空是什么颜色？")]):
    print(chunk.content, end="", flush=True)
```

> `AIMessageChunk` 支持 `+` 操作符合并为完整的 `AIMessage`：
>
> ```
> full_message = chunk1 + chunk2 + chunk3
> ```

---

5. `ToolMessage` - 工具消息

将工具调用结果返回给模型。

```
from langchain_core.messages import ToolMessage

tool_response = ToolMessage(
    content="2025年9月17日的天气：晴，气温25°C",
    tool_call_id="call_abc123"  # 必须匹配之前的 tool_call_id
)
```

6. `RemoveMessage`（特殊类型）

不对应任何角色，用于在 **LangGraph** 中管理聊天历史，例如删除旧消息以控制上下文长度。

2.2.4 多模态消息支持

消息内容可以是：

- **纯文本**
- **字典列表**（用于图像、音频等）

```
# 示例：包含图像的多模态输入（具体格式依赖模型）
human_msg = HumanMessage(
    content=[
        {"type": "text", "text": "描述这张图片"},
        {"type": "image_url", "image_url": "https://example.com/image.jpg"}
    ]
)
```

> **注意**：多模态支持因模型而异，目前仍处于发展阶段。

2.3.5 对话结构示例

一个典型的对话应遵循合理的结构：

```
from langchain_core.messages import HumanMessage, AIMessage

conversation = [
    HumanMessage(content="你好，你怎么样？"),
    AIMessage(content="我很好，谢谢！"),
    HumanMessage(content="你能给我讲个笑话吗？"),
    AIMessage(content="当然！为什么程序员分不清万圣节和圣诞节？因为 Oct 31 == Dec 25！")
]
```

2.3.6 总结

LangChain 的消息系统提供了一个强大且灵活的抽象层，使开发者能够：

- 使用统一的 API 与多种聊天模型交互。
- 构建复杂的对话流程，包括工具调用和流式响应。
- 支持多模态输入和高级上下文管理。
- 无缝集成 OpenAI 兼容格式。

通过合理使用 `SystemMessage`、`HumanMessage`、`AIMessage` 和 `ToolMessage`，可以构建出功能丰富、响应迅速的对话式 AI 应用。

2.3.7 示例

```
from langchain_community.chat_models import ChatOllama
from langchain_core.messages import (
    SystemMessage,
    HumanMessage,
    AIMessage,
    ToolMessage,
    AIMessageChunk,
)
import json

# 初始化本地 ChatOllama 模型
# 使用 llama3.1 或其他支持对话的模型
model = ChatOllama(
    model="qwen3:1.7b",  # 确保该模型已通过 ollama pull 下载
    temperature=0.7,
    base_url="http://localhost:11434",  # 默认 Ollama 地址
)

# ============================
# 1. SystemMessage: 设定角色
# ============================
system_message = SystemMessage(
    content="你是一位专业的烹饪助手，擅长提供简单易懂的食谱和烹饪技巧。"
)

# ============================
# 2. HumanMessage: 用户提问
# ============================
human_message = HumanMessage(
    content="我想做番茄炒蛋，能告诉我步骤吗？"
)

print("💬 用户: 我想做番茄炒蛋，能告诉我步骤吗？\n")
print("🔍 助手 (流式输出): ", end="", flush=True)

# ============================
# 3. AIMessageChunk: 流式输出响应
# ============================
full_response = ""
for chunk in model.stream([system_message, human_message]):
    if isinstance(chunk, AIMessageChunk):
        content = chunk.content
        print(content, end="", flush=True)
        full_response += content

print("\n")

# 将流式输出合并为完整 AIMessage
ai_message = AIMessage(content=full_response)

# ============================
# 4. 模拟工具调用：查询食材库存
# ============================
# 假设模型决定调用一个“检查库存”工具
# 注意：Ollama 本身不原生支持结构化 tool_call，但我们可以模拟
```

```
# 手动构造一个工具调用请求（实际中可通过提示工程触发）
# 这里我们模拟模型返回了一个需要调用工具的响应

# 重新提问，引导模型“调用工具”
tool_prompt = [
    system_message,
    HumanMessage(content="我有鸡蛋，但不确定有没有番茄。请帮我检查番茄库存。"),
]

print("\n🔍 模拟工具调用流程...")

# 强制让模型“想象”它调用了工具
tool_simulation_prompt = tool_prompt + [
    AIMessage(content="我将调用 'check_inventory' 工具来检查番茄库存。"),
]

# 模拟工具返回结果
tool_result = ToolMessage(
    content=json.dumps({"item": "tomato", "in_stock": True, "quantity": 3}),
    tool_call_id="call_tomato_check_123",  # 匹配假设的调用 ID
    name="check_inventory"  # 可选：工具名称
)

# 将工具结果传回模型，让其生成最终回答
final_response = model.invoke(
    tool_simulation_prompt + [tool_result]
)

print(f"\n📦 工具返回: {{'item': 'tomato', 'in_stock': True, 'quantity': 3}}")
print(f"🤖 助手最终回复: {final_response.content}")
```

2.3 提示模板（Prompt Templates

2.3.1 概述

提示模板（Prompt Templates）是 LangChain 框架中的核心组件之一，用于将用户输入和参数动态地转换为语言模型（LLM）可以理解的指令。它帮助模型更好地理解上下文，生成更相关、连贯的输出。

提示模板接收一个字典作为输入（键对应模板中的变量），输出一个 `PromptValue` 对象，该对象可被传递给 LLM 或聊天模型，也可转换为字符串或消息列表。

2.3.2 主要类型

LangChain 提供了两种主要类型的提示模板：

1. `PromptTemplate` - 字符串提示模板

适用于简单的文本输入场景，用于格式化单个字符串。

```
from langchain_core.prompts import PromptTemplate

# 定义模板
prompt_template = PromptTemplate.from_template(
    "讲一个关于 {topic} 的笑话。"
)

# 调用模板
result = prompt_template.invoke({"topic": "猫"})
print(result.text)
# 输出示例： "为什么猫喜欢坐在键盘上？因为它们想控制你的生活！"
```

> **API 参考**：PromptTemplate

2. `ChatPromptTemplate` - 聊天提示模板

用于构建**消息列表**，适用于与聊天模型交互的复杂对话场景。模板由多个消息对组成，支持系统消息、用户消息等。

```
from langchain_core.prompts import ChatPromptTemplate

# 定义聊天模板
prompt_template = ChatPromptTemplate([
    ("system", "你是一个乐于助人的助手。"),
    ("user", "讲一个关于 {topic} 的笑话。")
])

# 调用模板
prompt_value = prompt_template.invoke({"topic": "程序员"})
print(prompt_value.to_messages())
```

> **输出**:
>
> ```
> [SystemMessage(content='你是一个乐于助人的助手。'),
>  HumanMessage(content='讲一个关于 程序员 的笑话。')]
> ```

> **API 参考**：ChatPromptTemplate

3. `MessagesPlaceholder` - 消息占位符

用于在模板中插入**动态的消息列表**，例如聊天历史。这是实现记忆功能的关键。

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage

# 模板包含系统消息 + 动态消息占位符
prompt_template = ChatPromptTemplate([
    ("system", "你是一个乐于助人的助手。"),
    MessagesPlaceholder("history"),  # 历史消息插入此处
    ("user", "{input}")
])

# 模拟聊天历史
history = [
    HumanMessage(content="你好！"),
    AIMessage(content="你好！有什么我可以帮忙的吗？")
```

```
]

# 生成最终提示
prompt_value = prompt_template.invoke({
    "history": history,
    "input": "你能给我讲个笑话吗？"
})

print(prompt_value.to_messages())
```

> 这将生成一个包含系统消息、历史消息和新用户输入的完整消息序列。

替代写法：使用 `placeholder`

你也可以不显式使用 `MessagesPlaceholder`，而是通过字符串占位符方式：

Python深色版本

```
prompt_template = ChatPromptTemplate([
    ("system", "你是一个助手。"),
    ("placeholder", "{history}"),  # 使用字符串占位符
    ("user", "{input}")
])
```

2.3.3 核心功能总结

功能	说明
变量填充	使用 `{variable}` 语法动态插入内容
消息结构化	支持系统、用户、助手等角色的消息组织
历史集成	通过 `MessagesPlaceholder` 插入聊天历史
统一输出	输出 `PromptValue`，兼容 LLM 和 ChatModel
灵活扩展	可组合多个模板、支持部分格式化等

2.4 表达式语言（LCEL）

2.4.1 概述

LangChain 表达式语言（LangChain Expression Language，简称 **LCEL**）是 LangChain 的核心编程范式，用于以声明式、可组合的方式构建可运行（Runnable）的链（Chain）。它允许开发者将小型组件（如提示模板、模型、解析器等）像积木一样连接起来，形成复杂的应用流程。

LCEL 不仅提升了代码的可读性和可维护性，还为生产环境提供了关键优势：**自动优化执行、异步支持、流式传输、LangSmith 追踪和无缝部署能力。**

2.4.2 核心理念

- **声明式编程**：你描述“要做什么”，而不是“如何做”。LangChain 负责优化运行时执行。
- **所有组件都是 `Runnable`**：无论是提示、模型还是自定义函数，都实现了统一的 `Runnable` 接口。
- **链即 Runnable**：使用 LCEL 构建的“链”本身也是一个 `Runnable`，可以嵌套、复用和部署。

> ✅ 简单说：LCEL 是 LangChain 的“胶水语言”，让你轻松拼接 AI 应用的各个部分。

2.4.3 核心组合原语

LCEL 提供了两个基本的组合方式来构建链：

1. `RunnableSequence` - 顺序执行

将多个组件按顺序链接，前一个的输出作为后一个的输入。

```
from langchain_core.runnables import RunnableSequence
from langchain_core.prompts import PromptTemplate
from langchain_community.chat_models import ChatOllama
from langchain_core.output_parsers import StrOutputParser

# 定义组件
prompt = PromptTemplate.from_template("讲一个关于 {topic} 的笑话。")
model = ChatOllama(model="llama3.1", base_url="http://localhost:11434")
parser = StrOutputParser()

# 方法1：使用 RunnableSequence
chain = RunnableSequence([prompt, model, parser])

# 调用
result = chain.invoke({"topic": "猫"})
print(result)
```

---

2. `RunnableParallel` - 并行执行

并发运行多个组件，输入相同，输出合并为字典。

```
from langchain_core.runnables import RunnableParallel

# 并行生成两个不同主题的笑话
parallel_chain = RunnableParallel({
    "joke_about_cats": prompt | model | parser,
    "joke_about_dogs": prompt | model | parser
})

result = parallel_chain.invoke({"topic": "猫"})  # 注意：这里 topic 只影响 cats
print(result)
# 输出示例： {'joke_about_cats': '...', 'joke_about_dogs': '...'}
```

简洁语法：`|` 运算符

LCEL 重载了 `|` 运算符，使链的写法更简洁直观。

```
# 推荐写法：使用 | 运算符
chain = prompt | model | parser

# 调用
result = chain.invoke({"topic": "程序员"})
print(result)
```

这等价于：

```
chain = RunnableSequence([prompt, model, parser])
```

> 💡 你也可以使用 `.pipe()` 方法，效果相同：`prompt.pipe(model).pipe(parser)`

2.4.4 自动类型转换（Type Coercion）

LCEL 会自动将某些 Python 类型转换为 `Runnable`：

原始类型	自动转换为
字典 `{}`	`RunnableParallel`
函数 `def` 或 `lambda`	`RunnableLambda`

字典 → RunnableParallel

```
# 字典会自动转为并行执行
chain = {
    "original": lambda x: x,
    "joke": prompt | model | parser
} | StrOutputParser()  # 后续处理
```

函数 → RunnableLambda

```
# 函数自动转为 Runnable
chain = (lambda x: x.upper()) | model
```

> ⚠ 注意：原始字典或函数不能直接调用 `.invoke()`，只有在 LCEL 表达式中才会被转换。

2.4.5 LCEL 的核心优势

优势	说明
✅ **并行执行优化**	`RunnableParallel` 自动并发执行，减少延迟。
✅ **异步支持**	所有链天然支持 `ainvoke`，`astream` 等异步方法。
✅ **流式传输**	支持 `stream()` 实时输出，优化首 token 时间。
✅ **LangSmith 自动追踪**	所有步骤自动记录，便于调试和监控。
✅ **标准 API**	所有链都实现 `Runnable` 接口，易于复用。
✅ **可部署**	可通过 LangServe 直接部署为 REST API。

2.3.5 总结

LCEL 是构建 LangChain 应用的现代推荐方式，它：

- 提供了简洁、声明式的语法（如 `prompt \| model \| parser`）。
- 自动优化执行性能（并行、流式、异步）。
- 深度集成可观测性（LangSmith）和部署能力（LangServe）。
- 是从旧版 `LLMChain` 等类迁移的首选方案。

对于大多数简单到中等复杂度的链，**应优先使用 LCEL**。当应用需要复杂状态或控制流时，再考虑升级到 **LangGraph**。

2.5 工具调用（Tool Calling）

2.5.1 概述

工具调用（Tool Calling）是 LangChain 的核心能力之一，它允许语言模型（LLM）不仅仅是生成文本，还能**直接与外部系统交互**，例如调用 API、查询数据库、执行计算等。

通过工具调用，AI 应用可以具备"行动"能力，从被动回答问题转变为**主动执行任务**，是构建智能 Agent 的关键技术。

> 💡 **提示**：工具调用有时也被称为"函数调用"（Function Calling），在 LangChain 中这两个术语可互换使用。

2.5.2 核心概念：工具调用的四个步骤

工具调用遵循一个清晰的工作流程：

1. 工具创建（Tool Creation）

使用 `@tool` 装饰器将普通 Python 函数定义为一个"工具"。工具包含函数逻辑和其输入/输出的结构化描述。

创建一个乘法工具

```
from langchain_core.tools import tool

@tool
def multiply(a: int, b: int) -> int:
    """将 a 和 b 相乘。"""
    return a * b
```

2. 工具绑定（Tool Binding）

将创建好的工具绑定到支持工具调用的模型上。使用 `.bind_tools()` 方法告诉模型"你可以调用这些工具"。

```
from langchain_community.chat_models import ChatOllama

# 初始化支持工具调用的模型（如 llama3.1）
model = ChatOllama(model="llama3.1", base_url="http://localhost:11434")

# 绑定工具
model_with_tools = model.bind_tools([multiply])
```

> ⚠ 注意：并非所有模型都支持工具调用。Ollama 的 `llama3.1` 支持基本的工具调用功能。

3. 工具调用（Tool Calling）

向模型发送请求。模型会根据输入内容**自主决定**是否调用工具，以及调用哪个工具。

```
# 不相关的输入 → 模型不会调用工具
result1 = model_with_tools.invoke("你好！")
print(result1.content)  # 输出： "你好！有什么我可以帮忙的吗？"

# 相关的输入 → 模型决定调用 multiply 工具
result2 = model_with_tools.invoke("2 乘以 3 是多少？")
```

---

4. 工具执行（Tool Execution）

如果模型决定调用工具，其响应中会包含 `tool_calls` 属性。你可以提取这些调用并执行对应的工具函数。

```
# 检查是否包含工具调用
if result2.tool_calls:
    for tool_call in result2.tool_calls:
        print("模型请求调用工具:", tool_call)
        # 输出示例：
        # {'name': 'multiply', 'args': {'a': 2, 'b': 3}, 'id': 'call_abc123',
'type': 'tool_call'}

        # 执行工具
        tool_output = multiply.invoke(tool_call["args"])
        print("工具执行结果:", tool_output)  # 输出：6
```

2.5.3 关键特性与高级用法

✅ `tool_calls` 属性

模型响应（`AIMessage`）中的 `tool_calls` 是一个列表，每个元素包含：

- `name`：工具名称
- `args`：调用参数（字典）
- `id`：调用唯一 ID
- `type`：类型（通常是 `tool_call`）

✅ 强制模型调用工具（`tool_choice`）

你可以强制模型必须调用某个特定工具，或从给定列表中选择一个工具。

```
# 强制必须调用 multiply 工具
model_force = model.bind_tools([multiply], tool_choice="multiply")
```

这在构建确定性行为的 Agent 时非常有用。

✅ 工具的最佳实践

建议	说明
工具职责单一	简单、功能明确的工具更容易被模型正确使用。
提供清晰的名称和描述	工具的函数名和 docstring 是模型理解其用途的关键。
避免工具过多	从大量工具中选择会增加模型的决策难度。
使用支持工具调用的模型	经过微调的模型（如 llama3.1）在工具调用方面表现更好。

与 LangGraph 集成

在实际应用中，通常不会手动检查 `tool_calls`。**LangGraph** 提供了 `ToolNode` 等预构建组件，可以自动代表用户执行工具调用，大大简化开发。

```
# 伪代码示例（LangGraph 中的 ToolNode）
from langgraph.prebuilt import ToolNode

tool_node = ToolNode([multiply])  # 自动执行 multiply 工具
```

2.5.4 总结

工具调用 是让 AI 模型"动起来"的关键机制：

- 使用 `@tool` 创建工具。
- 使用 `.bind_tools()` 将工具绑定到模型。
- 模型根据输入决定是否调用工具。
- 响应中的 `tool_calls` 包含了执行所需的所有信息。
- 支持强制调用、流式传输、错误处理等高级功能。

它是构建**智能代理**（Agent）、**自动化工作流**和**增强型问答系统**的基础。结合 `LCEL` 和 `LangGraph`，你可以构建出功能强大、可维护的生产级 AI 应用。

2.6 检索器（Retrievers）

2.6.1 什么是检索器（Retriever）？

检索器（Retriever） 是 LangChain 中用于从数据源中**获取相关文档**的组件。它是一个接口，定义了一个简单的方法：

```
def invoke(query: str) -> List[Document]
```

即：输入一个查询字符串，返回一组相关的 `Document` 对象。

> ✅ 与 `PromptTemplate`、`LLM` 并列，是构建 RAG（检索增强生成）系统的核心组件之一。

2.6.2 Retriever vs LLM：关键区别

特性	Retriever	LLM
输入	`str` 或 `dict`	`str` / `PromptValue` / `List[Message]`
输出	`List[Document]`	`str` / `ChatResult`
接口方法	`.invoke()` / `.batch()` / `.stream()`	同左
是否可缓存	✅ 是	✅ 是
是否支持流式	❌ 否（返回完整列表）	✅ 是

> 📌 提示：Retriever 的输出通常作为上下文输入给 LLM，用于生成最终回答。

2.6.3 核心方法（Standard Interface）

所有 Retriever 都实现以下方法：

方法	说明
`.invoke(query)`	单次检索，返回 `List[Document]`
`.batch(queries)`	批量检索多个查询
`.stream(query)`	❌ 不支持流式（整体返回）

```
docs = retriever.invoke("量子计算是什么？")
print(len(docs))  # 输出：4
```

2.6.4 常见检索器类型

LangChain 提供了多种内置 Retriever，适用于不同场景：

1. 向量存储检索器（VectorStoreRetriever）

最常用的类型，基于向量相似度检索。

```
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

vectorstore = Chroma(embedding_function=OpenAIEmbeddings())
retriever = vectorstore.as_retriever()  # 返回 Retriever 接口
```

✅ 支持参数：

- `k`：返回文档数
- `search_type`：`similarity`（默认）、`mmr`（最大边际相关性）、`similarity_score_threshold`

```
retriever = vectorstore.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"score_threshold": 0.75}
)
```

---

2. BM25Retriever（关键词匹配）

基于 BM25 算法进行**关键词检索**，适合精确匹配术语。

```
from langchain.retrievers import BM25Retriever

docs = ["猫喜欢爬树", "狗喜欢追球", "鸟会飞"]
retriever = BM25Retriever.from_texts(docs, k=1)

result = retriever.invoke("猫喜欢什么？")
# 返回最匹配的文档
```

> ✅ 优点：无需嵌入模型，适合专业术语检索
> ❌ 缺点：无法理解语义

---

3. EnsembleRetriever（混合检索）

组合多个检索器的结果，提升召回率。

```
from langchain.retrievers import EnsembleRetriever

ensemble = EnsembleRetriever(
    retrievers=[vectorstore_retriever, bm25_retriever],
    weights=[0.5, 0.5]
)

result = ensemble.invoke("动物的习性")
```

> 使用 **RRF（倒数排序融合）** 算法对结果去重并排序。

4. ContextualCompressionRetriever（上下文压缩）

先检索，再用 LLM **压缩或过滤**不相关的内容。

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

compressor = LLMChainExtractor.from_llm(llm)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=vectorstore_retriever
)
```

> ✅ 优点：减少噪声，提升生成质量
> ⚠️ 缺点：增加延迟和成本

5. ParentDocumentRetriever（父子文档检索）

适用于长文档切分场景：**小块检索 + 大块生成。**

流程：

1. 将文档切分为小 chunk（用于检索）
2. 检索到小 chunk 后，返回其所属的“父文档”（大块）

```
from langchain.retrievers import ParentDocumentRetriever
from langchain_community.vectorstores import Chroma
from langchain_community.storage import InMemoryStore

retriever = ParentDocumentRetriever(
    vectorstore=Chroma(...),
    docstore=InMemoryStore(),
    child_splitter=RecursiveCharacterTextSplitter(chunk_size=200),
    parent_splitter=RecursiveCharacterTextSplitter(chunk_size=1000),
)
```

> ✅ 适用：书籍、长报告、法律条文等长文本 RAG

6. TimeWeightedVectorStoreRetriever（时间加权）

为文档添加时间衰减因子，**越新的文档权重越高。**

```
from langchain.retrievers import TimeWeightedVectorStoreRetriever

retriever = TimeWeightedVectorStoreRetriever(
    vectorstore=vectorstore,
    decay_rate=0.01,
    k=2
)
```

> ✅ 适用：记忆系统、用户行为追踪、新闻推荐

7. KNNRetriever（K近邻）

直接在 DataFrame 或 NumPy 数组上做 KNN 检索。

```
from langchain.retrievers import KNNRetriever
import pandas as pd

df = pd.DataFrame({"text": ["机器学习", "深度学习"], "embedding": [[1,2], [3,4]]})
retriever = KNNRetriever(df=df, text_column="text",
embedding_column="embedding")
```

> ✅ 适合结构化数据 + 嵌入混合检索

2.6.5 高级功能

1. 多查询生成（Multi-Query Retriever）

用 LLM 为原始查询生成多个变体，提升召回率。

```
from langchain.retrievers import MultiQueryRetriever

retriever = MultiQueryRetriever.from_llm(
    retriever=vectorstore.as_retriever(),
    llm=ChatOpenAI()
)

# 用户问："气候变化的影响"
# 可能生成："全球变暖的后果"、"气候变暖对生态的影响"等
```

2. 自定义 Retriever

继承 `BaseRetriever` 实现自定义逻辑：

```
from langchain_core.retrievers import BaseRetriever
from langchain_core.documents import Document

class MyRetriever(BaseRetriever):
    def _get_relevant_documents(self, query):
        return [Document(page_content="自定义结果")]

retriever = MyRetriever()
```

2.6.6 使用场景建议

场景	推荐 Retriever
通用语义检索	`VectorStoreRetriever`
专业术语匹配	`BM25Retriever`
提高召回率	`EnsembleRetriever`、`MultiQueryRetriever`
长文档处理	`ParentDocumentRetriever`
实时性要求高	`TimeWeightedVectorStoreRetriever`
需要过滤噪声	`ContextualCompressionRetriever`

2.6.7 示例

```
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_ollama import OllamaEmbeddings  # ✅ 新导入
from langchain_community.vectorstores import Chroma

# 1. 加载文档
loader = TextLoader("data.txt", encoding="utf-8")
docs = loader.load()

# 2. 分割文本
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=300,
    chunk_overlap=50
)
splits = text_splitter.split_documents(docs)

# 3. 使用 langchain-ollama 的嵌入模型
embeddings = OllamaEmbeddings(
    model="nomic-embed-text",
    base_url="http://localhost:11434",
    keep_alive=-1  # -1 表示永久保留在内存中
)

# 4. 存入向量数据库
vectorstore = Chroma.from_documents(
    documents=splits,
    embedding=embeddings,
    persist_directory="./chroma_db_nomic"
)

print("✅ 数据已使用 nomic-embed-text 嵌入并存入向量数据库！")

# 5. 创建检索器
retriever = vectorstore.as_retriever(search_kwargs={"k": 2})

# 6. 查询测试
query = "LangChain 是做什么的？"
result = retriever.invoke(query)
```

```
print(f"\n🔍 查询: {query}")
for i, doc in enumerate(result):
    print(f"\n--- 结果 {i+1} ---")
    print(doc.page_content)
```