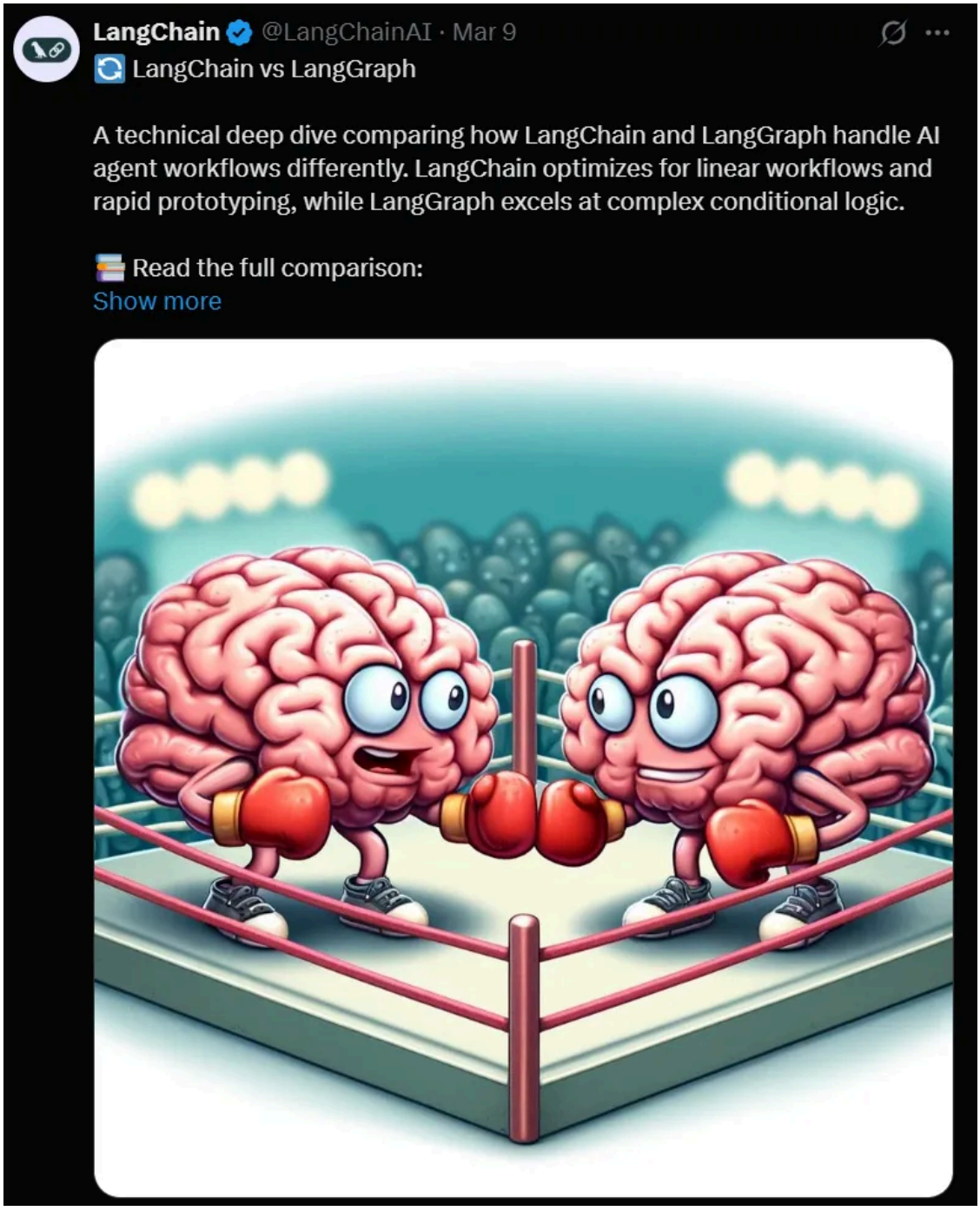
用LangChain还是LangGraph？官方终于站出来表态了

原创 南七无名式 PyTorch研习社 2025年03月11日 08:01 安徽

无论是个人还是企业，当我们想要使用 LLM（大模型）的功能开发出有趣或者有价值的应用时，第一个技术问题通常是“使用什么工具？”

在本文中，我们将深入探讨用于构建 LLM 应用程序的两个最流行的框架：LangChain 和 LangGraph。

现在生成式 AI 的开发正处于井喷时期，几乎每天都会出现各种新框架和新技术，所以各位在阅读本文时请记住，今天是正确的，明天可能就不正确了！



LangChain 和 LangGraph 的基础组件

通过理解每个框架的不同基础元素，你将更深入地理解它们在处理某些核心功能方面的关键区别。本节的描述并未穷尽列出每个框架的所有组件，而是为理解它们的整体方法差异提供了一个坚实的基础。

LangChain

使用 LangChain 主要有两种方式：**预定义命令的顺序链（Chain）和 LangChain Agent**。这两种方式在工具和编排方式上有所不同。链采用预定义的线性工作流，而 Agent 则充当一个协调者，可以进行更具动态性（非线性）的决策。

- Chain（链）：一系列步骤的组合，这些步骤可以包括调用 LLM、Agent、工具、外部数据源、过程式代码等。链可以分支，即根据逻辑条件将单一流程拆分为多个路径。
- Agent 或 LLM：LLM 本身能够生成自然语言响应，而 Agent 则结合了 LLM 与额外能力，使其能够进行推理、调用工具，并在调用工具失败时重复尝试。
- Tool（工具）：是可以在链中被调用的代码函数，或由 Agent 触发，以与外部系统交互。
- Prompt（提示词）包括系统提示词（用于指示模型如何完成任务以及可用工具）、来自外部数据源的注入信息（为模型提供更多上下文）、以及用户的输入任务。

LangGraph

LangGraph 采用了一种不同的方法来构建 AI 工作流。正如其名称所示，它以**图（Graph）的方式编排工作流**。由于其在 AI Agent、过程式代码和其他工具之间的灵活处理能力，它更适用于线性链、分支链或简单 Agent 系统难以满足需求的复杂应用场景。LangGraph 设计用于处理更复杂的条件逻辑和反馈循环，比 LangChain 更加强大。

- Graph（图）是一种灵活的工作流组织方式，支持调用 LLM、工具、外部数据源、过程式代码等。LangGraph 还支持循环图（Cyclical Graph），即可以创建循环和反馈机制，使得某些节点能够被多次访问。
- Node（节点）表示工作流中的步骤，例如 LLM 查询、API 调用或工具执行。
- Edge（边）和 Conditional Edge（条件边）：边用于连接节点，定义信息流向，使一个节点的输出作为下一个节点的输入。条件边允许在满足特定条件时，将信息从一个节点流向另一个节点。开发者可以自定义这些条件。
- State（状态）表示应用程序的当前状态，随着信息在图中流动而更新。状态是一个开发者定义的可变 TypedDict 对象，包含当前执行图所需的所有相关信息。LangGraph 会在每个节点自动更新状态。
- Agent 或 LLM：图中的 LLM 仅负责对输入生成文本响应。而 Agent 能力则允许图中包含多个节点，分别代表 Agent 的不同组件（如推理、工具选择和工具执行）。Agent 可以决定在图中采取哪条路径、更新图的状态，并执行比单纯文本生成更多的任务。

相比之下，**LangChain 更适合线性和基于工具的调用，而 LangGraph 更适合复杂的、多路径和具有反馈机制的 AI 工作流**。

各框架在核心功能处理方式上的区别

LangGraph 和 LangChain 在某些能力上有所重叠，但它们处理问题的方式有所不同。LangChain 主要关注线性工作流（通过链）或不同的 AI Agent 模式，而 LangGraph 则专注于创建更灵活、细

粒度的、基于流程的工作流，其中可以包含 AI Agent、工具调用、过程式代码等。

总体而言，LangChain 的学习曲线相对较低，因为它提供了更多的抽象封装和预定义配置，这使得 LangChain 更容易应用于简单的使用场景。而 LangGraph 则允许对工作流设计进行更细粒度的定制，这意味着它的抽象程度较低，开发者需要学习更多内容才能有效使用。

工具调用（Tool Calling）

LangChain

在 LangChain 中，工具的调用方式取决于是在链中按顺序执行一系列步骤，还是仅使用 Agent 能力（不在链中明确定义）。

- 在链中，工具是作为预定义步骤包含的，这意味着它们并不一定是由 Agent 动态调用的，而是在链设计时就已决定了调用哪些工具。
- 当 Agent 不在链中定义时，Agent 具有更大的自主性，它可以根据自己可访问的工具列表，决定调用哪个工具以及何时调用。

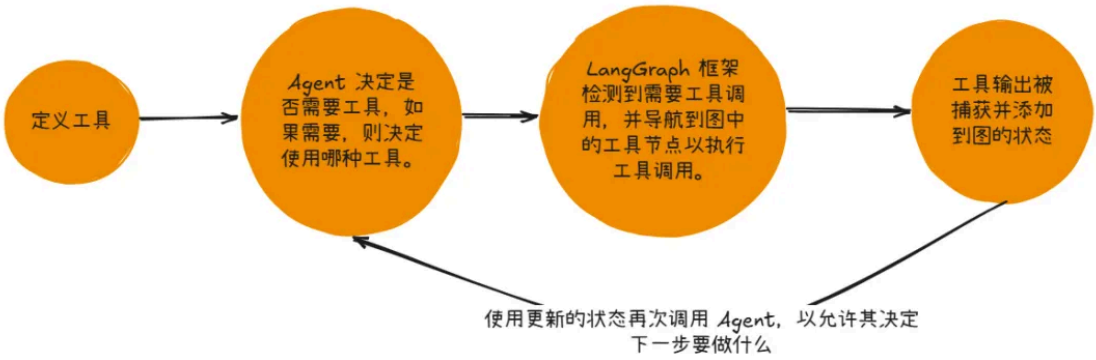
链方式的流程示例：

Agent 方式的流程示例：

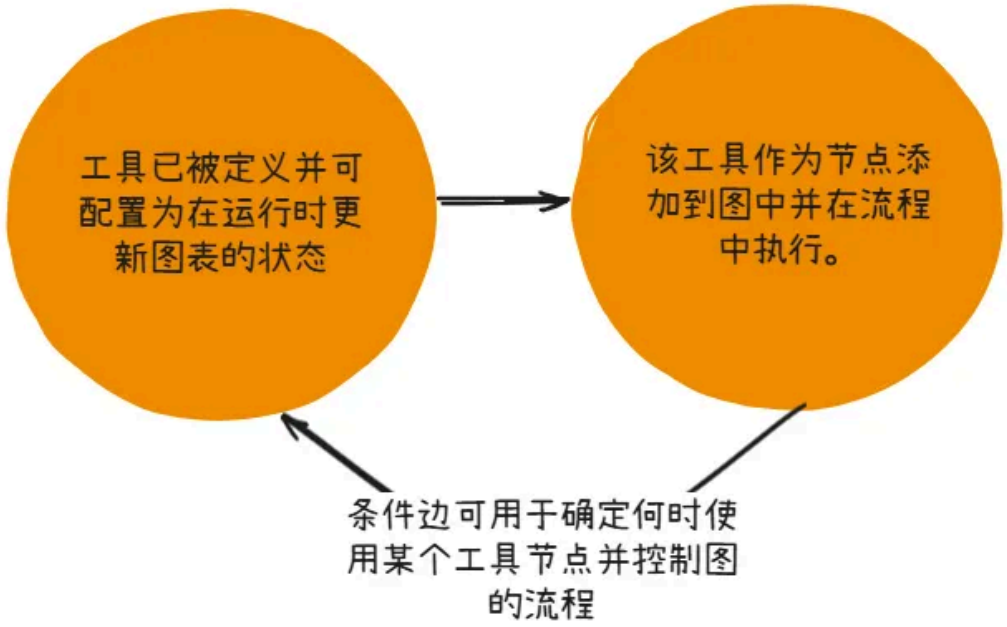
LangGraph

在 LangGraph 中，工具通常被表示为图上的一个节点。如果图中包含一个 Agent，那么 Agent 负责决定调用哪个工具，这一决策基于其推理能力。当 Agent 选择了某个工具后，工作流会跳转到对应的“工具节点”（Tool Node），以执行该工具的操作。在 Agent 和工具节点之间的边可以包含条件逻辑（Conditional Logic），从而增加额外的判断逻辑，以决定是否执行某个工具。这样，开发者可以拥有更精细的控制能力。如果图中没有 Agent，那么工具的调用方式类似于 LangChain 的链，即根据预定义的条件逻辑在工作流中执行工具。

包含 Agent 的图流程示例：



没有 Agent 的图的流程示例：



对话历史和记忆

LangChain

LangChain 提供内置的抽象层来处理对话历史和记忆。它支持不同粒度级别（granularity）的记忆管理，从而控制传递给 LLM 的 token 数量，主要包括以下几种方式：

- 完整的会话历史（Full session conversation history）
- 摘要版本的对话历史（Summarized conversation history）
- 自定义定义的记忆（Custom defined memory）

此外，开发者还可以自定义长期记忆系统，将对话历史存储在外部数据库中，并在需要时检索相关记忆。

LangGraph

在 LangGraph 中，State（状态）负责管理记忆，它通过记录每个时刻定义的变量来跟踪状态信息。State 可以包括：

- 对话历史
- 任务执行的各个步骤

- 语言模型上一次的输出结果
- 其他重要信息

State 可以在节点之间传递，这样每个节点都能获取当前系统的状态。然而，LangGraph 本身不提供跨会话的长期记忆功能，如果开发者需要持久化存储记忆，可以引入特定的节点，用于将记忆和变量存入外部数据库，以便后续检索。

开箱即用的 RAG 能力

LangChain

LangChain 原生支持复杂的 RAG 工作流，并提供了一套成熟的工具，方便开发者将 RAG 集成到应用程序中。例如，它提供了：

- 文档加载（Document Loading）
- 文本解析（Text Parsing）
- Embedding 生成（Embedding Creation）
- 向量存储（Vector Storage）
- 检索能力（Retrieval Capabilities）

开发者可以直接使用 LangChain 提供的 API（如 langchain.document_loaders、langchain.embeddings 和 langchain.vectorstores）来实现 RAG 工作流。

LangGraph

在 LangGraph 中，RAG 需要开发者自行设计，并作为图结构的一部分实现。例如，开发者可以创建单独的节点，分别用于：

- 文档解析（Document Parsing）
- Embedding 计算（Embedding Computation）
- 语义检索（Retrieval）

这些节点之间可以通过普通边（Normal Edges）或条件边（Conditional Edges）进行连接，而各个节点的状态可以用于传递信息，以便在 RAG 流水线的不同步骤之间共享数据。

并行处理（Parallelism）

LangChain

LangChain 允许并行执行多个链或 Agent，可以使用 RunnableParallel 类来实现基本的并行处理。

但如果需要更高级的并行计算或异步工具调用，开发者需要使用 Python 库（如 asyncio）自行实现。

LangGraph

LangGraph 天然支持并行执行节点，只要这些节点之间没有依赖关系（例如，一个 LLM 的输出不能作为下一个节点的输入）。这意味着多个 Agent 可以同时运行，前提是它们不是相互依赖的节

点。

此外，LangGraph 也支持：
- 使用 RunnableParallel 运行多个 Graph
- 通过 Python 的 asyncio 库并行调用工具

重试逻辑和错误处理

LangChain

LangChain 的错误处理需要由开发者显式定义，可以通过：
- 在链中引入重试逻辑（Retry Logic）
- 在 Agent 中处理工具调用失败的情况

LangGraph

LangGraph 可以直接在工作流中嵌入错误处理逻辑，方法是将错误处理作为一个独立的节点。
- 当某个任务失败时，可以跳转到另一个错误处理节点，或者在当前节点进行重试。
- 失败的节点会被单独重试，而不是整个工作流重新执行。
- 这样，图可以从失败的地方继续执行，而不需要从头开始。

如果你的任务涉及多个步骤和工具调用，这种错误处理机制可能会非常重要。

总之

你可以：
- ✅ 仅使用 LangChain
- ✅ 仅使用 LangGraph
- ✅ 同时使用 LangChain 和 LangGraph

此外，你也可以将 LangGraph 的图结构编排能力与其他 Agent 框架（如微软的 AutoGen）结合，例如：将 AutoGen 的 Agent 作为 LangGraph 的节点

LangChain 和 LangGraph 各有优势，选择合适的工具可能会让人感到困惑。

那么，应该在什么情况下使用？

仅使用 LangChain：
- ✅ 你需要快速构建 AI 工作流，例如：
- 线性任务（Linear Tasks）：文档检索、文本生成、摘要等预定义的工作流
- AI Agent 需要动态决策，但你不需要对复杂流程进行精细控制

仅使用 LangGraph：
- ✅ 你的应用场景需要非线性（Non-linear）工作流，例如：
- 任务涉及多个组件的动态交互
- 需要条件判断、复杂的分支逻辑、错误处理或并行执行

- 你愿意自行实现 LangChain 未提供的部分功能

同时使用 LangChain 和 LangGraph：

✅ 你希望：

- 利用 LangChain 现成的抽象能力（如 RAG 组件、对话记忆等）
- 同时使用 LangGraph 的非线性编排能力

两者结合，可以充分发挥各自的优势，打造更加灵活和强大的 AI 工作流。

https://towardsdatascience.com/ai-agent-workflows-a-complete-guide-on-whether-to-build-with-langgraph-or-langchain-117025509fa0/

PyTorch研习社
打破知识壁垒，做一名知识的传播者
715篇原创内容

公众号