0001年1月1日 10 分钟阅读

LangChain 技术架构与实现详解

LangChain 是一个强大的框架，旨在简化和加速基于大型语言模型 (LLM) 的应用程序的开发。它的技术架构和实现围绕着**模块化、可组合性和标准化接口**的核心思想。

以下是对 LangChain 技术架构与实现的详解：

一、核心设计理念与目标

模块化 (Modularity): 将 LLM 应用开发中常见的组件（如 LLM 封装、Prompt 管理、数据连接、记忆、Agent 等）抽象为独立的模块。开发者可以按需选用和组合。

可组合性 (Composability): 通过**LangChain 表达式语言 (LCEL)** 和 `Runnable` 协议，可以像搭乐高积木一样将这些模块轻松连接起来，形成复杂的"链 (Chains)"或"图 (Graphs, via LangGraph)"。

标准化接口 (Standardization): 为不同 LLM 提供商、向量数据库等提供统一的接口，降低切换成本和学习曲线。

端到端应用支持 (End-to-End): 提供从数据加载、处理、LLM 调用、状态管理到 Agent 决策的全流程支持。

生态系统 (Ecosystem): 积极集成大量第三方工具、API 和服务。

二、核心架构组件 (Pillars of LangChain)

LangChain 的架构可以看作由以下几个核心支柱构成：

`Schema` (模式/接口定义):

作用： 定义了 LangChain 中各种数据结构和接口的基本规范。

实现：

`Messages`: (AIMessage, HumanMessage, SystemMessage, ToolMessage, FunctionMessage) - 定义了与聊天模型交互的消息类型。

`Document`: 包含 `page_content` (文本内容) 和 `metadata` (元数据) 的标准文档对象。

`PromptValue`: Prompt 模板格式化后的输出，作为 LLM 的输入。

`LLMResult`, `ChatResult`: LLM 和聊天模型调用的标准输出格式，包含生成内容、token 使用情况等。

技术： 大量使用 Pydantic 进行数据校验和序列化。

目录

文章信息

字数

阅读时间

发布时间

`Models` **(模型接口):**

作用： 封装了与各种 LLM 和嵌入模型的交互逻辑。

实现：

`LLMs`: 针对文本补全型 LLM (如 GPT-3 `text-davinci-003`) 的封装，提供 `invoke` (原 `predict` 或 `__call__`) 方法。

`ChatModels`: 针对聊天型 LLM (如 GPT-3.5-turbo, GPT-4) 的封装，输入输出为 `Messages` 列表。

`Embeddings`: 封装文本嵌入模型 (如 OpenAI Embeddings, HuggingFace Embeddings)，提供 `embed_query` 和 `embed_documents` 方法。

技术：

为不同模型提供商 (OpenAI, Anthropic, HuggingFace, Cohere, Google VertexAI 等) 实现适配器。

支持异步调用 (`ainvoke`, `aembed_documents` 等)。

`Runnable` 协议的实现者。

`Prompts` **(提示工程):**

作用： 帮助构建和管理动态、灵活的提示。

实现：

`PromptTemplate`: 基本的 f-string 风格模板。

`ChatPromptTemplate`: 专门为聊天模型设计，可以包含多种类型的 `MessagePromptTemplate`。

`FewShotPromptTemplate`: 用于构建包含少量示例的提示。

`PipelinePromptTemplate`: 组合多个 Prompt 模板。

Partialing: 预先填充模板中的部分变量。

技术：

模板引擎（如 Jinja2 可选，但核心是 f-string 和 Pydantic 模型的结合）。

输入变量的自动推断和验证。

`Runnable` 协议的实现者。

`Indexes` **(数据索引与检索):**

作用： 连接 LLM 与外部数据源，实现基于检索的生成 (RAG)。

实现：

`Document Loaders`: 从各种来源（文件、网页、数据库、API 等）加载数据为 `Document` 对象。

`Text Splitters`: 将长文档切分为适合 LLM 处理和嵌入的小块。

`VectorStores`: 存储文本块的嵌入向量，并提供高效的相似性搜索。常见的有 FAISS, Chroma, Pinecone, Weaviate, Milvus 等。

`Retrievers`: 封装从 VectorStore 或其他来源检索相关文档的逻辑。提供 `get_relevant_documents` 和 `aget_relevant_documents` 方法。

技术：

各种数据格式解析库。

向量相似性搜索算法。

`Runnable` 协议的实现者 (尤其是 Retrievers)。

`Memory` (记忆管理):

作用： 为 Chain 或 Agent 提供短期或长期记忆能力，使对话具有上下文。

实现：

`ChatMessageHistory`: 存储消息历史的基类。

`ConversationBufferMemory`: 简单地将所有消息存储在缓冲区。

`ConversationBufferWindowMemory`: 只保留最近 K 条消息。

`ConversationSummaryMemory`: 使用 LLM 对对话历史进行摘要。

`ConversationKGMemory`: 将对话信息构建为知识图谱。

`VectorStoreRetrieverMemory`: 将对话历史存入向量数据库并进行检索。

技术：

通常与 `Runnable` 结合，在链执行前后加载和保存上下文。

需要定义清晰的 `memory_key`。

`Chains` (链):

作用： 这是 LangChain 的核心概念，通过将多个组件（LLM、Prompt、Retriever 等）按特定顺序或逻辑组合起来，形成一个执行序列。

实现（现在主要通过 LCEL）：

LCEL (LangChain Expression Language): 使用 `|` (管道) 操作符将 `Runnable` 对象链接起来。

`RunnableSequence`: `prompt | llm | parser` 这种形式会隐式创建一个 `RunnableSequence`。

`RunnableParallel`: `{"context": retriever, "question": RunnablePassthrough()}` 这种字典形式会创建一个 `RunnableParallel`，并行执行其分支并将结果合并。

`RunnablePassthrough`: 将输入原样传递，常用于并行分支中。

`RunnableLambda`: 将普通函数包装成 `Runnable`。

`RunnableConfig`: 控制执行过程中的配置，如回调、标签、最大并发数等。

旧式 Chains (依然可用，但推荐 LCEL):

`LLMChain`: 最基础的链，包含 PromptTemplate 和 LLM。

`SequentialChain`: 按顺序执行多个子链。

`RouterChain`: 根据输入路由到不同的子链。

`TransformChain`: 对输入或输出进行自定义转换。

技术:

`Runnable` 协议是 LCEL 的基石。所有参与 `|` 运算的对象都必须实现 `Runnable` 接口（`invoke`, `stream`, `batch` 及其异步版本 `ainvoke`, `astream`, `abatch`）。

`RunnableSequence` 和 `RunnableParallel` 内部实现了调度逻辑，确保数据流正确传递和并发执行。

`Agents` (智能体):

作用: 让 LLM 具备思考、决策和使用工具 (Tools) 的能力，以完成更复杂的任务。

实现:

`Tools`: 封装了外部功能（如搜索、计算器、API 调用、数据库查询等），LLM 可以决定调用哪个工具以及如何调用。

`AgentExecutor`: 驱动 Agent 的核心循环。它接收用户输入，让 LLM 进行思考（输出包含要调用的工具和参数的特殊格式），执行工具，将工具的输出反馈给 LLM，重复此过程直到任务完成或达到停止条件。

Agent 类型 (Prompting 策略):

`Zero-shot ReAct`: 通用的基于 ReAct (Reason+Act) 框架的 Agent。

`Conversational ReAct`: 为对话场景优化的 ReAct Agent。

`OpenAI Functions/Tools Agent`: 利用 OpenAI 模型的函数调用/工具调用能力。

`XML Agent`, `JSON Agent`: 针对特定输出格式的 Agent。

技术:

精巧的 Prompt 工程，指导 LLM 进行思考和工具选择。

输出解析器，从 LLM 的输出中提取工具调用信息。

`AgentExecutor` 是一个特殊的 `Runnable`，内部管理着 LLM、工具集和思考循环。

`Callbacks` (回调系统) & `LangSmith`:

作用： 提供对 LangChain 应用内部事件的监控、日志记录、调试和追踪。

实现：

`CallbackManager` 和 `CallbackHandler` 基类。

`StdOutCallbackHandler`: 将事件打印到标准输出。

`FileCallbackHandler`: 将事件记录到文件。

`LangSmithRunManager` (与 LangSmith 集成): 将运行的详细信息（输入、输出、中间步骤、耗时、token 消耗等）发送到 LangSmith平台，用于可视化、调试和评估。

技术：

事件驱动模型。`Runnable` 的 `invoke`, `stream` 等方法在执行的关键节点（开始、结束、出错、子模块调用等）会触发回调。

`RunnableConfig` 中可以指定 `callbacks`。

三、LangChain Expression Language (LCEL) 的核心实现

LCEL 是现代 LangChain 的灵魂。

`Runnable` 协议:

定义了统一的调用接口：`invoke`, `stream`, `batch` 和它们的异步版本 `ainvoke`, `astream`, `abatch`。

所有 LangChain 的核心组件（Prompts, LLMs, Retrievers, Parsers, 甚至是整个 Chains 和 Agents）都实现了这个协议。

`|` (管道操作符):

通过 Python 的 `__or__` 魔术方法实现。

`runnable1 | runnable2` 会创建一个 `RunnableSequence` 对象，该对象依次执行 `runnable1` 和 `runnable2`，并将前者的输出作为后者的输入。

并行执行:

`{"key1": runnable1, "key2": runnable2}` 这种字典语法会创建一个 `RunnableParallel` 对象。

当 `RunnableParallel` 执行时，它会（可能并发地）执行 `runnable1` 和 `runnable2`，并将它们的输出收集到一个字典中：`{"key1": output1, "key2": output2}`。

配置传递 (`RunnableConfig`):

`RunnableConfig` 对象可以在链式调用中传递，用于控制回调、标签、最大并发等。

`runnable.with_config(config)` 可以创建一个绑定了特定配置的新 Runnable。

四、LangGraph (构建有状态的多 Actor 应用)

LangGraph 是 LangChain 的一个扩展库，专门用于构建具有**循环、分支和更复杂状态管理**的 Agent 和应用。

核心概念:

StatefulGraph: 一个图，其节点可以修改共享的状态对象。

Nodes: 可以是任何 `Callable` (通常是 `Runnable`)，接收当前状态，执行操作，并返回对状态的更新。

Edges: 定义了节点之间的转换逻辑。

Conditional Edges: 根据当前状态决定下一个要执行的节点。

技术实现:

底层使用 `networkx` (Python 版) 或类似图库来表示图结构。

状态对象通常使用 Pydantic 模型定义，方便序列化和校验。

调度器负责根据图的结构和当前状态执行节点。

五、实现语言与依赖

主要语言: Python 和 TypeScript/JavaScript (LangChain.js)。Python 版本功能更全，社区更活跃。

核心依赖 (Python):

`pydantic`: 数据校验和模型定义。

`aiohttp` / `httpx`: 异步 HTTP 请求。

`numpy`: 数值计算，尤其在嵌入和向量操作中。

`tenacity`: 重试逻辑。

特定集成库：如 `openai`, `pinecone-client`, `faiss-cpu`, `tiktoken` 等。

六、总结

LangChain 的技术架构是一个精心设计的分层系统：

底层是标准化的接口和数据模式 (`Schema`, `Runnable` 协议)。

中间层是各种可独立使用的模块 (`Models`, `Prompts`, `Indexes`, `Memory`, `Tools`)。

顶层是通过 LCEL 或 LangGraph 将这些模块灵活组合起来的 `Chains` 和 `Agents`。

贯穿始终的是 `Callbacks` 和 `LangSmith` 提供的可观测性。

这种架构使得 LangChain 既能提供开箱即用的高级功能，又能允许开发者深入到底层进行定制和扩展，从而适应从简单脚本到复杂生产级应用的不同需求。其成功很大程度上归功于对 LLM 应用开发模式的深刻理解和抽象，以及 LCEL 带来的优雅组合能力。

LangChain 的 Runnable 协议

LangChain 的 Runnable 协议是其核心抽象之一，它为构建和组合各种组件（如 LLM、Prompt 模板、输出解析器、检索器、工具等）提供了一套标准化的接口和机制。理解它的实现主要涉及以下几个方面：

`Runnable` **基类 (Conceptual Interface):** 虽然在 Python 中接口更多是约定俗成的（duck typing），但 `Runnable` 可以被视为一个概念上的基类或协议，它规定了其子类应该实现或继承的一系列方法。一个对象只要实现了这些核心方法，就可以被认为是“Runnable”的。

核心方法 (The “How”): `Runnable` 协议定义了多种执行方法，以适应不同的使用场景：

`invoke(input: Input, config: Optional[RunnableConfig] = None) -> Output`:

实现：这是最基本的同步执行方法。子类通常会重写此方法来实现其核心逻辑。例如，一个 `ChatPromptTemplate` 的 `invoke` 会格式化输入，一个 `ChatModel` 的 `invoke` 会调用 LLM API。

输入 (`Input`) 和输出 (`Output`) 的类型取决于具体的 Runnable。

`config` (`RunnableConfig`) 用于传递运行时配置，如回调函数、标签、递归限制、线程池等。

`ainvoke(input: Input, config: Optional[RunnableConfig] = None) -> Output`:

实现：异步版本的 `invoke`。子类需要实现异步逻辑，通常使用 `async/await`。如果子类没有显式实现 `ainvoke`，基类可能会提供一个默认的包装，将同步的 `invoke` 包装在线程池中执行（但这效率不高，推荐显式实现）。

`stream(input: Input, config: Optional[RunnableConfig] = None) -> Iterator[Output]`:

实现：同步的流式输出方法，返回一个迭代器。例如，LLM 在生成 token 时可以一块一块地输出。

它允许处理部分结果，而无需等待整个响应完成。

`astream(input: Input, config: Optional[RunnableConfig] = None) -> AsyncIterator[Output]`:

实现：异步版本的 `stream`，返回一个异步迭代器。

`batch(inputs: List[Input], config: Optional[Union[RunnableConfig, List[RunnableConfig]]] = None, *, return_exceptions: bool = False, **kwargs: Any) -> List[Output]`:

实现：同步批量处理多个输入。默认实现可能是循环调用 `invoke`，但子类可以重写以实现更高效的批量操作（例如，某些 LLM API 支持批量请求）。

`return_exceptions` 控制当某个输入处理出错时是抛出异常还是返回异常对象。

`abatch(...)`：异步版本的 `batch`。

`transform(input_stream: Iterator[Input], config: Optional[RunnableConfig] = None) -> Iterator[Output]`：

实现：接收一个输入流，并产生一个输出流。用于处理流式输入。

`atransform(...)`：异步版本的 `transform`。

组合与链式调用（The "Magic" - LCEL): LangChain Expression Language (LCEL) 使得 Runnable 可以通过 `|` (管道) 操作符轻松组合。

`__or__(self, other: Union[Runnable[Any, Output], Callable[[Any], Output], Mapping[str, Union[Runnable[Any, Output], Callable[[Any], Output], Any]]]) -> RunnableSequence`：

实现：这是 `|` 操作符背后的魔法。当 `runnable1 | runnable2` 时，实际上会调用 `runnable1.__or__(runnable2)`。

这个方法通常会创建一个 `RunnableSequence` 对象。

`RunnableSequence`：

实现：这是一个特殊的 Runnable，它内部持有一个 Runnable 列表（或更具体地说是 `first` Runnable 和 `last` Runnable，中间可能还有其他 `RunnableSequence`）。

当 `RunnableSequence` 的 `invoke` (或 `ainvoke` 等) 被调用时，它会按顺序执行其内部的 Runnables：
调用第一个 Runnable 的 `invoke` 方法，传入初始输入。

将第一个 Runnable 的输出作为第二个 Runnable 的输入，调用第二个 Runnable 的 `invoke` 方法。

以此类推，直到最后一个 Runnable 执行完毕，其输出作为整个序列的输出。

它负责正确地传递 `RunnableConfig`，并处理流式和批量操作的逻辑。

`RunnableParallel`：

实现：当使用字典形式组合时，例如 `{"context": retriever, "question": RunnablePassthrough()}`，这会创建一个 `RunnableParallel`。

它会并行（或并发）执行其包含的所有 Runnables（以相同的输入或 `RunnablePassthrough()` 传递的输入为基础），并将结果收集

到一个字典中。

类型提示与 `InputType` / `OutputType`：

`Runnable[InputType, OutputType]` 使用泛型来指定输入和输出的类型。这有助于静态分析和提高代码可读性，但 Python 的动态特性意味着这些更多是约定。Pydantic 模型经常被用来定义这些类型，以获得运行时的验证。

`RunnableConfig` 的传递：

实现：`RunnableConfig` 对象在链式调用中会被传递下去。每个 Runnable 都可以访问它，并根据需要使用其中的配置（例如，回调函数会在每个 Runnable 执行前后被触发）。

`Runnable.with_config(config)` 方法可以创建一个绑定了特定配置的新 Runnable 实例，而不改变原始 Runnable。

辅助 Runnables：

`RunnablePassthrough`：它的 `invoke` 方法通常只是原样返回输入。用于在序列中传递数据或作为并行分支的输入。

`RunnableLambda`：允许将任何 Python callable（函数或 lambda）包装成一个 Runnable。

`RunnableBranch`：实现条件逻辑，根据输入选择不同的执行路径。

`RunnableBinding`：用于将 Runnable 与特定的参数、配置或停止序列绑定在一起，创建一个新的、预配置的 Runnable。例如 `llm.bind(stop=["\nObservation:"])`。

总结一下实现机制：

约定接口：定义了一套标准方法 (`invoke`, `stream`, `batch` 及其异步版本)。

魔术方法：`__or__` (for `|`) 用于创建 `RunnableSequence`。

组合类：`RunnableSequence` 和 `RunnableParallel` 是核心的组合器，它们本身也是 Runnable，并负责调度其内部 Runnables 的执行。

配置传递：`RunnableConfig` 贯穿整个调用链。

异步支持：通过 `async/await` 和异步迭代器/生成器提供非阻塞 I/O。

Pydantic (可选但推荐)：用于定义清晰的输入/输出模式和运行时验证。

通过这种方式，LangChain 能够将复杂的 LLM 应用逻辑分解为一系列可重用、可组合、可测试且具有标准接口的组件，极大地简化了开发流程并提高了代码的健壮性和可维护性。

具体实现模拟

LangChain 的 Runnable 协议是整个 LCEL（LangChain Expression Language）的核心基础。让我详细解析它的实现：

核心接口定义

```
from abc import ABC, abstractmethod
from typing import Any, List, Optional,
Iterator, AsyncIterator

class Runnable(ABC):
    """所有 LangChain 组件的基础协议"""

    @abstractmethod
    def invoke(self, input: Any, config:
Optional[RunnableConfig] = None) -> Any:
        """同步执行"""
        pass

    @abstractmethod
    async def ainvoke(self, input: Any,
config: Optional[RunnableConfig] = None) ->
Any:
        """异步执行"""
        pass

    @abstractmethod
    def batch(self, inputs: List[Any], config:
Optional[RunnableConfig] = None) -> List[Any]:
        """批量执行"""
        pass

    @abstractmethod
    def stream(self, input: Any, config:
Optional[RunnableConfig] = None) ->
Iterator[Any]:
        """流式输出"""
        pass

    @abstractmethod
    async def astream(self, input: Any,
config: Optional[RunnableConfig] = None) ->
AsyncIterator[Any]:
        """异步流式输出"""
        pass
```

基础实现类

1. RunnableBase - 提供默认实现

```python
class RunnableBase(Runnable):
    """提供了大部分方法的默认实现"""

    def invoke(self, input: Any, config: Optional[RunnableConfig] = None) -> Any:
        # 默认实现：调用 _call 方法
        return self._call(input, run_manager=self._get_run_manager(config))

    async def ainvoke(self, input: Any, config: Optional[RunnableConfig] = None) -> Any:
        # 默认实现：在线程池中运行同步版本
        return await asyncio.get_event_loop().run_in_executor(
            None, self.invoke, input, config
        )

    def batch(self, inputs: List[Any], config: Optional[RunnableConfig] = None) -> List[Any]:
        # 默认实现：循环调用 invoke
        return [self.invoke(input, config) for input in inputs]

    def stream(self, input: Any, config: Optional[RunnableConfig] = None) -> Iterator[Any]:
        # 默认实现：yield 单个结果
        yield self.invoke(input, config)

    @abstractmethod
    def _call(self, input: Any, run_manager: Optional[CallbackManagerForChainRun] = None) -> Any:
        """子类需要实现的核心逻辑"""
        pass
```

2. RunnableLambda - 将函数包装成 Runnable

```
class RunnableLambda(RunnableBase):
    """将普通函数转换为 Runnable"""

    def __init__(self, func: Callable, afunc: Optional[Callable] = None):
        self.func = func
        self.afunc = afunc

    def _call(self, input: Any, run_manager: Optional[CallbackManagerForChainRun] = None) -> Any:
        return self.func(input)

    async def _acall(self, input: Any, run_manager: Optional[AsyncCallbackManagerForChainRun] = None) -> Any:
        if self.afunc:
            return await self.afunc(input)
        # 降级到同步版本
        return await asyncio.get_event_loop().run_in_executor(None, self.func, input)
```

组合操作符实现

1. 管道操作符 (|)

```
class RunnableSequence(RunnableBase):
    """实现管道组合 A | B | C"""

    def __init__(self, steps: List[Runnable]):
        self.steps = steps

    def _call(self, input: Any, run_manager:
Optional[CallbackManagerForChainRun] = None) -
> Any:
        # 依次执行每个步骤
        for step in self.steps:
            input = step.invoke(input, config)
        return input

    async def _acall(self, input: Any,
run_manager:
Optional[AsyncCallbackManagerForChainRun] =
None) -> Any:
        # 异步版本
        for step in self.steps:
            input = await step.ainvoke(input,
config)
        return input

    def stream(self, input: Any, config:
Optional[RunnableConfig] = None) ->
Iterator[Any]:
        # 流式处理：只有最后一个步骤产生流
        for step in self.steps[:-1]:
            input = step.invoke(input, config)

        # 最后一个步骤的流式输出
        yield from
self.steps[-1].stream(input, config)

# 在 RunnableBase 中定义 __or__ 方法
class RunnableBase:
    def __or__(self, other: Runnable) ->
RunnableSequence:
        return RunnableSequence([self, other])
```

2. 并行操作符

```
class RunnableParallel(RunnableBase):
    """并行执行多个 Runnable"""

    def __init__(self, steps: Dict[str, Runnable]):
        self.steps = steps

    def _call(self, input: Any, run_manager: Optional[CallbackManagerForChainRun] = None) -> Dict[str, Any]:
        # 并行执行所有步骤
        return {
            key: runnable.invoke(input, config)
            for key, runnable in self.steps.items()
        }

    async def _acall(self, input: Any, run_manager: Optional[AsyncCallbackManagerForChainRun] = None) -> Dict[str, Any]:
        # 真正的异步并行
        results = await asyncio.gather(*[
            runnable.ainvoke(input, config)
            for runnable in self.steps.values()
        ])
        return dict(zip(self.steps.keys(), results))
```

配置和回调系统

```
@dataclass
class RunnableConfig:
    """运行时配置"""
    tags: List[str] =
field(default_factory=list)
    metadata: Dict[str, Any] =
field(default_factory=dict)
    callbacks: Optional[Callbacks] = None
    run_name: Optional[str] = None
    max_concurrency: Optional[int] = None
    recursion_limit: int = 25
    configurable: Dict[str, Any] =
field(default_factory=dict)

class CallbackManagerForChainRun:
    """管理运行时的回调"""

    def on_chain_start(self, serialized:
Dict[str, Any], inputs: Any):
        for handler in self.handlers:
            handler.on_chain_start(serialized,
inputs)

    def on_chain_end(self, outputs: Any):
        for handler in self.handlers:
            handler.on_chain_end(outputs)

    def on_chain_error(self, error:
Exception):
        for handler in self.handlers:
            handler.on_chain_error(error)
```

高级特性实现

1. 重试机制

```
class RunnableRetry(RunnableBase):
    """带重试的 Runnable"""

    def __init__(self, runnable: Runnable,
max_attempts: int = 3):
        self.runnable = runnable
        self.max_attempts = max_attempts

    def _call(self, input: Any, run_manager:
Optional[CallbackManagerForChainRun] = None) -
> Any:
        last_error = None
        for attempt in
range(self.max_attempts):
            try:
                return
self.runnable.invoke(input)
            except Exception as e:
                last_error = e
                if attempt < self.max_attempts
- 1:
                    time.sleep(2 ** attempt)
# 指数退避
        raise last_error
```

2. 绑定参数

```
class RunnableBinding(RunnableBase):
    """预绑定参数的 Runnable"""

    def __init__(self, runnable: Runnable,
kwargs: Dict[str, Any]):
        self.runnable = runnable
        self.kwargs = kwargs

    def _call(self, input: Any, run_manager:
Optional[CallbackManagerForChainRun] = None) -
> Any:
        # 合并输入和预绑定的参数
        if isinstance(input, dict):
            merged_input = {**self.kwargs,
**input}
        else:
            merged_input = {"input": input,
**self.kwargs}
        return
self.runnable.invoke(merged_input)

# 在 RunnableBase 中
class RunnableBase:
    def bind(self, **kwargs) ->
RunnableBinding:
        return RunnableBinding(self, kwargs)
```

3. 流式处理优化

```
class RunnableGenerator(RunnableBase):
    """支持生成器的 Runnable"""

    def __init__(self, func: Callable[..., Iterator]):
        self.func = func

    def stream(self, input: Any, config: Optional[RunnableConfig] = None) -> Iterator[Any]:
        # 直接返回生成器
        yield from self.func(input)

    def invoke(self, input: Any, config: Optional[RunnableConfig] = None) -> Any:
        # 收集所有输出
        chunks = list(self.stream(input, config))
        # 尝试合并输出
        if chunks and hasattr(chunks[0], "__add__"):
            return sum(chunks[1:], chunks[0])
        return chunks
```

类型安全支持

```
from typing import TypeVar, Generic

InputType = TypeVar("InputType")
OutputType = TypeVar("OutputType")

class Runnable(Generic[InputType, OutputType], ABC):
    """带类型参数的 Runnable"""

    @abstractmethod
    def invoke(self, input: InputType, config: Optional[RunnableConfig] = None) -> OutputType:
        pass

    def __or__(self, other: "Runnable[OutputType, T]") -> "Runnable[InputType, T]":
        """类型安全的管道组合"""
        return RunnableSequence[InputType, T]([self, other])
```

实际使用示例

```
# 1. 简单函数转 Runnable
add_one = RunnableLambda(lambda x: x + 1)
multiply_two = RunnableLambda(lambda x: x * 2)

# 2. 组合
chain = add_one | multiply_two  # (x + 1) * 2

# 3. 并行
parallel = RunnableParallel({
    "add": add_one,
    "multiply": multiply_two
})

# 4. 复杂组合
complex_chain = (
    RunnableLambda(lambda x: {"number": x})
    | parallel
    | RunnableLambda(lambda x: x["add"] +
x["multiply"])
)

# 5. 执行
result = complex_chain.invoke(5)  # ((5 + 1) +
(5 * 2)) = 16
```

关键设计决策

统一接口：所有组件都实现相同的接口，便于组合

同步/异步双支持：适应不同的使用场景

流式优先：原生支持流式处理，适合 LLM 输出

配置传递：通过 config 参数传递运行时配置

类型安全：支持泛型，提供类型提示

可观测性：内置回调系统，支持追踪和监控

这个设计让 LangChain 的各种组件可以像乐高积木一样自由组合，同时保持了良好的性能和可维护性。

实际使用示例

```
# 1. 简单函数转 Runnable
add_one = RunnableLambda(lambda x: x + 1)
multiply_two = RunnableLambda(lambda x: x * 2)

# 2. 组合
chain = add_one | multiply_two  # (x + 1) * 2

# 3. 并行
parallel = RunnableParallel({
    "add": add_one,
    "multiply": multiply_two
})

# 4. 复杂组合
complex_chain = (
    RunnableLambda(lambda x: {"number": x})
    | parallel
    | RunnableLambda(lambda x: x["add"] + x["multiply"])
)

# 5. 执行
result = complex_chain.invoke(5)  # ((5 + 1) + (5 * 2)) = 16
```

分享这篇文章