LangChain 初入门 - 简化 LLM 开发难度
章节简介

LangChain 框架认识与初步使用

- 学习了解 LangChain 基础，为什么需要 AI 应用开发框架，而不是直接基于 LLM 开发，以及选择 LangChain 框架的原因。

- 学习使用 LangChain 组件，了解其内部结构，掌握 Model 与 Prompt 组件的使用技巧，学会使用 LCEL 表达式构建链条应用。

项目接入 LangChain 与 Git 版本控制

- 在项目中接入 LangChain ，并将对应的 LLM 对话接口改写为 LangChain 的形式，并解读项目 API 文档与撰写规则。
- 学习如何使用 Git 管理项目并配置远程仓库，实现对代码的版本控制，每周课程生成对应一个版本。
- 分享在不同场景下，不同的 Git 操作流程，涵盖不同迭代模式、 IDE 下的 Git 操作技巧等。

Callback 系统与 LangSmith 平台

- 学习 LangChain 中的 Callback 回调系统，实时监控 LLM 应用程序的各个阶段，或执行特定操作。
- 学习如何注册与设置 LangSmith ，以及使用 LangSmith 监控 LLM 应用生命周期的技巧。
- 在项目中接入 LangSmith ，并调试、测试、评估和监控第一个聊天机器人，找出存在的问题。

为什么选择 LangChain 及 LangChain 简介

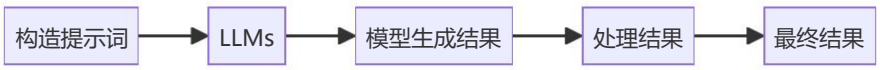
为什么选择LangChain及LangChain简介

01. 为什么选择LangChain

1.1 为什么需要大模型的应用框架

大模型一般有两种形态“呈现”在我们眼前，一种是训练好的那种二进制文件，另外一种是将大模型的二进制文件进行部署之后暴露出一些相应的接口，但是无论是那种形式，LLM 只提供了一个非常基础的调用方式，当我们要构建一个复杂的 Chat Bot 时，就需要考虑如何保存聊天的上下文、如何进行网络检索、如何加载本地数据、如何便捷管理 Prompt 等等工程问题。

甚至是当我们切换到不同的 LLM 时，模型的输入和输出结构差异都非常巨大，微量的需求就要修改大量的代码，或者在业务代码中做大量的判断与识别，让代码可维护性极差，但是其实不同 LLM 的交互流程其实都非常接近，如下可以看成是一个基础聊天机器人的链条，传入提示词，输出对应的结果：



以下是使用 OpenAI 与文心一言大模型 SDK 写的代码对比：

```
from openai import OpenAI

def openai_completion():
    """OpenAI大语言模型聊天接口"""
    # 1.提取从接口中获取的输入，POST
    req = CompletionReq()
    if not req.validate():
        return validate_error_json(req.errors)

    # 2.构建OpenAI客户端，并发起请求
    client = OpenAI(base_url=os.getenv("OPENAI_API_BASE"))

    # 3.得到请求响应，然后将OpenAI的响应传递给前端
    completion = client.chat.completions.create(
        model="gpt-3.5-turbo-16k",
        messages=[
            {"role": "system", "content": "你是OpenAI开发的聊天机器人，请根据用户的输入
回复对应的信息"},
            {"role": "user", "content": req.query.data},
        ]
    )

    content = completion.choices[0].message.content

    return success_json({"content": content})
```

```
import qianfan

def wenxin_completion():
    """文心一言大语言模型聊天接口"""
    # 1.提取从接口中获取的输入，POST
    req = CompletionReq()
```

```
    if not req.validate():
        return validate_error_json(req.errors)

    # 2.构建文心一言客户端
    client = qianfan.ChatCompletion()

    # 3.得到请求响应，然后将文心一言的响应传递给前端
    resp = client.do(messages=[
        {"role": "user", "content": "你好"}
    ])

    content = resp["body"]

    return success_json({"content": content})
```

除了在代码开发这方面有大量的疑难杂项之外，对 LLM 的运行流程、输出、费用统计、错误监控也是一个非常重要的步骤，这些功能 基础的 LLM 均没有提供，需要程序员自行开发与对接，开发这些功能的耗时甚至会超过业务的部分，极大提升了 AI 应用开发的难度。

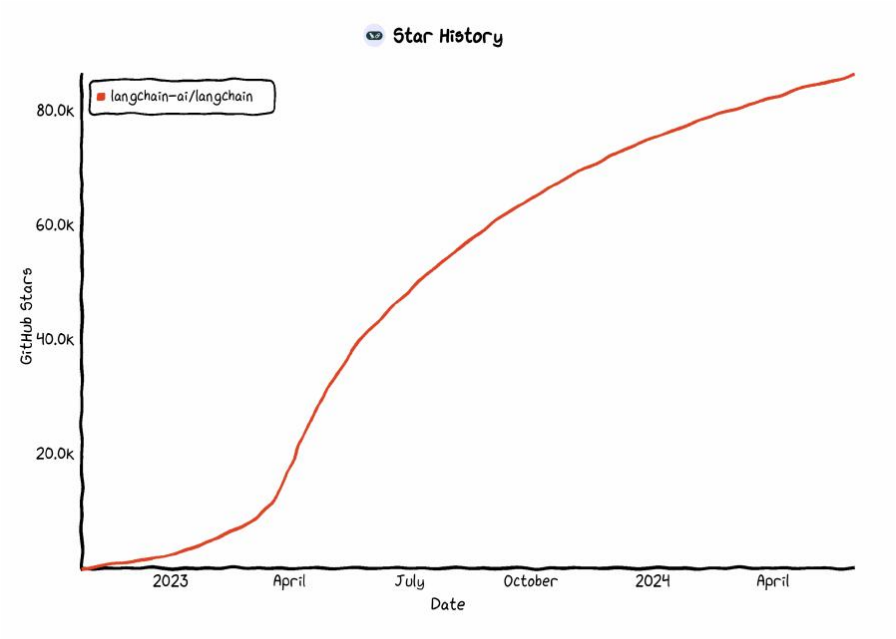
为了解决以上这些问题，AI 应用开发框架应运而生，其中最热门、更新速度最快、最稳定的框架就是 LangChain，而且目前 LangChain 提供了 Python 和 JavaScript 两个版本，适配了当前 AI 环境下最热门的两种语言。

- LangChain-Python：https://github.com/langchain-ai/langchain
- LangChain-JavaScript：https://github.com/langchain-ai/langchainjs

1.2 为什么选择 LangChain

为什么选择 LangChain？主要有几个方面：`认可度`、`持续维护`、`易用性`、`可扩展性`、`稳定版本`、`可观察`。

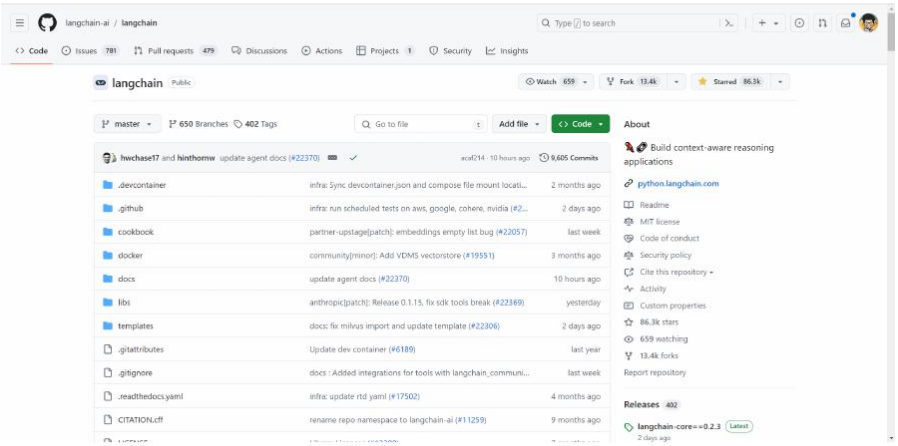
首先是`认可度`，目前在 Github 上短短一年多的时间 LangChain 已获得了 86.3k star，是目前 star 最多的 AI 应用开发框架，并且其上升速度非常恐怖，并且 LangChain 拥有活跃的开发者社区和丰富的文档资源，能够快速解决问题和获取帮助。



其次是 `持续维护` ，LangChain 框架经历数次融资，累计融资数千万美金，拥有足够的资源对框架进行长期及稳定更新的维护，在短短一年多的时间里 LangChain 团队提交了接近 10000 次 commit 进行数百个大小版本的更新与维护，创下 AI 应用开发领域之最。

项目也从早期的一个副业，迅速发展成数万个 LLM 应用的支柱，不乏一些知名项目：早期 Dify、MetaGPT、百度智能云千帆等，这些项目均校验了 LangChain 在生产环境中使用的可能。

作为一个年轻而有活力的框架，LangChain 正在彻底改变工业和技术，改变我们与技术的每一次互动。



除此之外，`易用性` 和 `可扩展性` 也是 LangChain 的特性，LangChain 屏蔽了不同 LLM 调用接口、输入格式、输出格式之间的差异，用户无需深究不同 LLM 的细节，创建模型后，就可以直接使用。

在扩展性方面，除了 LangChain 提供的大量第三方组件的集成，也可以将自有的本地大语言模型、工具、数据库、文本嵌入等任何内容快速接入到 LangChain 中，LangChain 好比一把瑞士军刀，提供了近 700 种集成，涵盖从 LLM 到向量存储，再到 Agent 使用的各种工具。

至此不再需要为每一个任务找一个新工具，它提供了一站式的解决方案，不管你遇到什么问题，打钉子、拧螺丝、剪线，工具箱里总有一个合适的工具等着你。

例如以下是接入 LangChain 后，使用 OpenAI 和文心一言大语言模型的示例，修改 LLM 时，无需修改任何业务性的代码，直接切换不同的 LLM 即可：

```
from langchain_community.chat_models.baidu_qianfan_endpoint import
QianfanChatEndpoint
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

def completion():
    """聊天接口"""
    # 1.提取从接口中获取的输入，POST
    req = CompletionReq()
    if not req.validate():
        return validate_error_json(req.errors)

    # 2.构建LLM客户端
    llm = ChatOpenAI()
    # 百度文心一言大语言模型
    # llm = QianfanChatEndpoint()

    # 3.将llm与输出解析器创建链
    chain = llm | StrOutputParser()

    # 3.得到请求响应，并解析出字符串
    content = chain.invoke(req.query.data)

    return success_json({"content": content})
```

而像这类 LLM，在 LangChain 集成封装了超过 60 个，几乎市面上能找到的大语言模型，无论是开源还是闭源，LangChain 都进行了封装，并且保持了高强度的更新。

在 `可观测性` 上，LangChain 对比同类的 AI 应用框架，提供了独一无二的 LangSmith 平台，LangSmith 的一个核心优势是为你的 LLM 应用提供一流的调试体验，详细记录了正在执行的每个步骤、每个步骤的输入、输出、所需时间等数据。并以用户友好的方式展示这些信息，让你能够识别哪些步骤耗时最长，进入一个调试区域来处理 LLM 意外反馈的问题，追踪 Token 使用情况等。

最重要的转折点在 2024 年 1 月 8 日，LangChain 团队发布了第一个稳定版本 0.1.0 版本，并推出了 `LangGraph` 与 `LangSmith`。

这个版本完全向后兼容，并且有 `Python` 和 `JavaScript` 两个版本，甚至基于 LangGraph 组件，可以便捷使用 LangChain 组件构建工作流任务，至此，Agent 从一开始的黑箱模式变得可观测，完成最后一块拼图，实现了在企业生产环境中可以稳定使用 LangChain 的可能。

02. LangChain 简介与基本概念

2.1 LangChain 的乐高世界

小时候我们都玩过乐高积木，通过堆砌各种颜色和形状的积木，我们可以构建出城堡、飞机、甚至整个城市。现在，想象一下，如果有一个数字世界的乐高，我们可以用这样的“积木”来构建智能程序，这些程序能够阅读、理解和撰写文本，与我们对话，甚至自主执行特定的任务。

提示词、工具、大语言模型就是就是一个一个积木块，等待着我们来发掘和搭建，而 LangChain 就是那个让我们将这些语言模型乐高积木组合成有趣应用的工具箱。它并不是一个实物，而是一个开源的 AI 应用开发框架，帮助开发者像搭建乐高一样快速构建和优化基于语言模型的应用。

一个简单的例子：

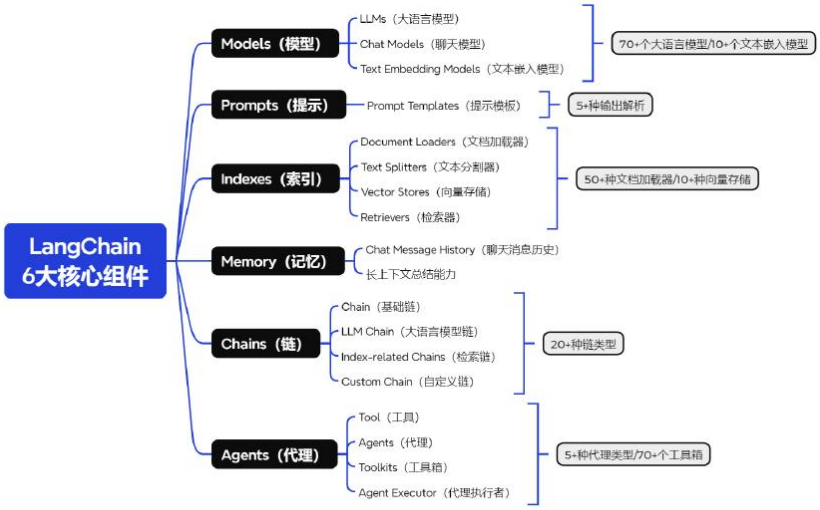
> 假设你想要用 GPT-4 建一个旅行顾问机器人。单独的 GPT-4 就像是一堆杂乱无章的乐高积木。它可能知道很多关于世界各地的信息，但如果不能实时查找最新的航班信息或者酒店价格，它提供的旅行建议可能就不够准确或实用。
>
> 提问：我该带些什么去泰国旅行?
>
> GPT-4 可能会基于以往的数据提供一般性的建议，如防晒霜、泳衣等。

使用 LangChain 的工具+记忆+索引积木块构建一个旅行顾问机器人：

> 配备了LangChain的问答系统，它可以查询实时的天气预报API，了解当前泰国的季节和天气情况，提供更精确的建议，比如“泰国正处于雨季，记得带上雨具和防潮包”。
>
> 提问：泰国哪里的垂钓体验最佳?
>
> LangChain可以帮助连接到最新的旅行博客和垂钓爱好者论坛，甚至直接查阅最近的旅行者评论，给你提供最受推荐的目的地。

2.2 LangChain 的基本概念

LangChain 主要提供了 6 大核心组件帮助我们更好的使用大语言模型，涵盖了 Models（模型）、Prompts（提示）、Indexes（索引）、Memory（记忆）、Chains（链）、Agents（代理），这些组件集成了数十种大语言模型、多样的知识库处理方法以及成熟的应用链、上百种可调用的工具箱，为用户提供了一个快速搭建和部署大语言模型智能应用程序的平台。



LangChain 框架安装及文档介绍

LangChain框架安装及文档介绍

01. LangChain 框架安装及其组成

通过以下命令安装 LangChain 版本的最低依赖要求，并安装基础的 LangChain 社区包。

```
pip install langchain langchain-community
```

如果想和课程版本保持一致，可以使用如下命令安装 `0.2.1` 版本：

```
pip install langchain==0.2.1 langchain-community==0.2.1
```

执行安装 `langchain` 后会自动安装以下扩展包：

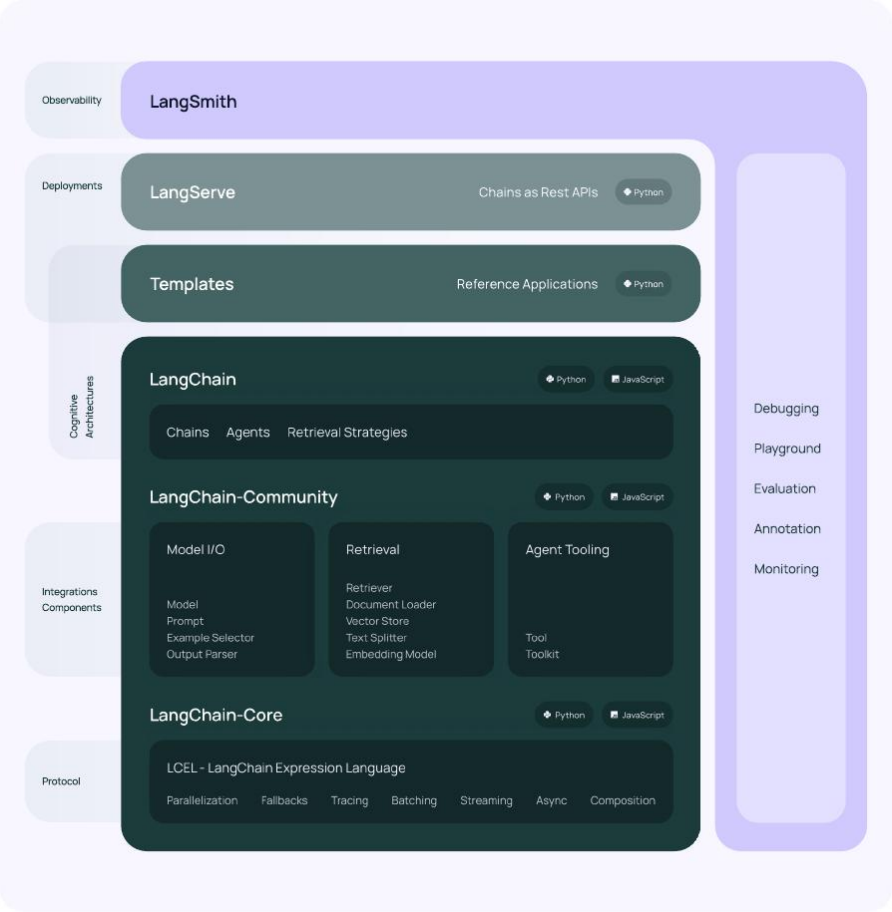
```
Installing collected packages: langsmith, langchain-core, langchain-text-
splitters, langchain, langchain-community
```

电脑上已经安装了 LangChain 的旧版本，也可以进行更新安装：

```
pip install -U langchain==0.2.1 langchain-community==0.2.1
```

LangChain 框架本身由多个开源库组成（v0.2.1版本）：

- langchain-core：基础抽象和 LangChain 表达式语言。
- langchain-community：第三方集成以及合作伙伴包（如 langchain-openai、langchain-anthropic 等），一些集成已经进一步拆分为自己的轻量级包，只依赖于 langchain-core。
- langchain：构建应用程序认知架构的链、代理和检索策略。
- langgraph：通过将步骤构建为图中的边和节点，使用 LLMs 构建健壮且有状态的多参与者应用程序。
- langserve：将 LangChain 链部署为 REST API。
- langsmith：一个开发平台，可以让你调试、测试、评估和监控 LLM 应用程序，并与 LangChain 无缝衔接。



02. LangChain 文档重构与解读

随着 LangChain 功能的不断发展，文档的内容也随之增长，在以往 v0.2.0 之前的版本中，文档的设计尽可能会详细介绍每个功能、每个代码片段、每个概念，这样会导致虽然内容很丰富，但用户往往在寻找特定信息时感到迷茫，这不仅影响了新用户的上手体验，也给长期贡献者带来了不小的挑战。

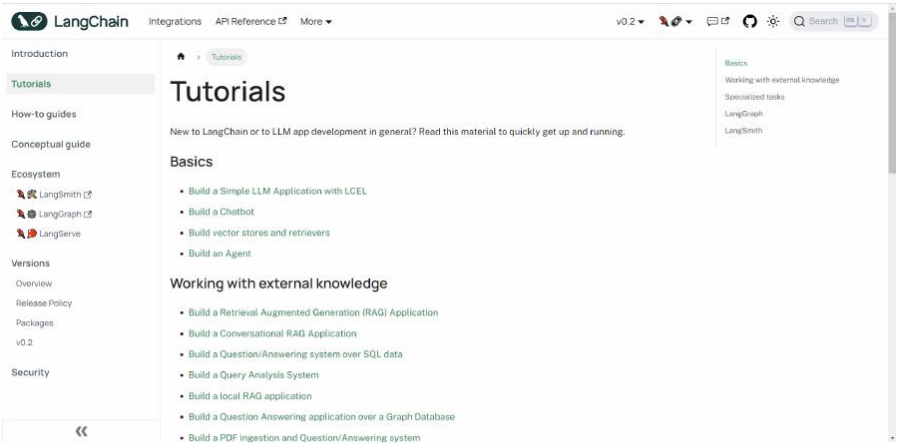
在这样的背景下，LangChain 团队决定对文档进行重构，并引入了一个核心概念——Diátaxis，这是一种全新的文档组织哲学，旨在将内容划分为四个清晰的类别：教程、操作指南、参考与解释。

> Diátaxis 官网：https://diataxis.fr/

而且无论是 LangChain 文档主站，还是 LangGraph、LangSmith 目前都使用了 Diátaxis 哲学对文档进行了重构。

2.1 教程（Tutorials）

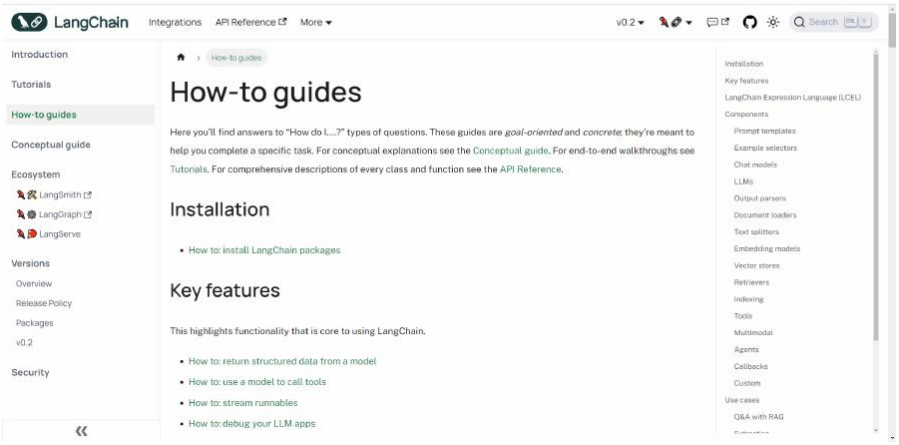
教程着重提供一些可以从零到一上手的一些例子。这些教程不仅仅是简单的步骤说明，更是对概念的深入理解和实践的完美结合。例如，构建本地 RAG 应用程序就是一个典型的教程案例，它通过实际操作让用户理解 RAG 的概念和应用。



2.2 操作指南（How-to Guides）

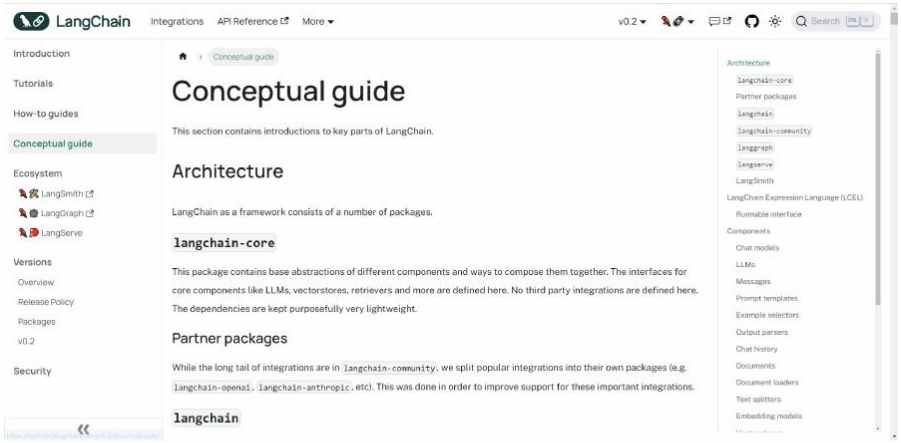
操作指南更侧重于你要去解决一些更复杂的探索或功能时，经常会遇到一些更细节的问题，比如：如何让应用流式输出。

与教程不同，操作指南专注于解决具体的实际的单点问题，帮助用户在现实世界中应用 LangChain 解决问题。



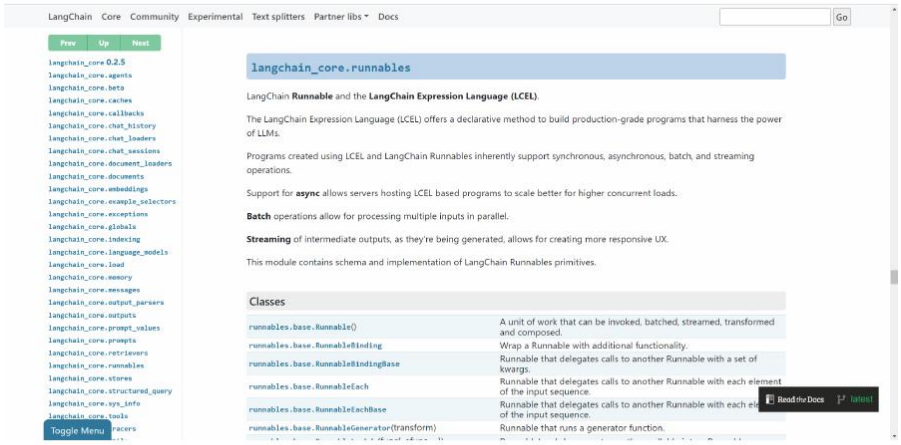
2.3 解释（Explanation）

最后，解释部分旨在阐明和阐释特定的主题或概念。这些内容通常更加深入，旨在帮助用户理解 LangChain 的工作原理和背后的逻辑。新的 LCEL 基础类型页面就是这样的解释性内容。



2.4 参考（Reference）

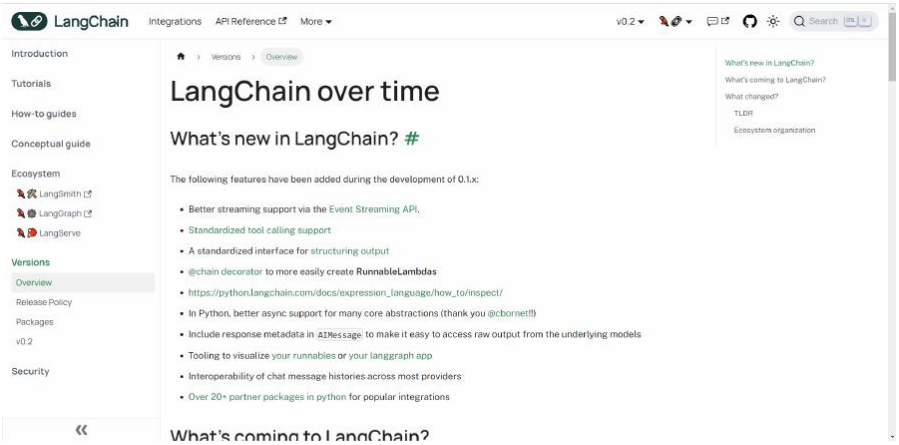
参考部分提供了对 LangChain 中各种机制和技术的详细描述。这些描述通常是技术性的，旨在帮助用户了解如何操作 LangChain 的各个部分。例如，Runnable 接口页面就提供了对 Runnable 接口的详细说明，Runnable 下对应的类、介绍及详细参数说明。



03. 版本变化说明文档

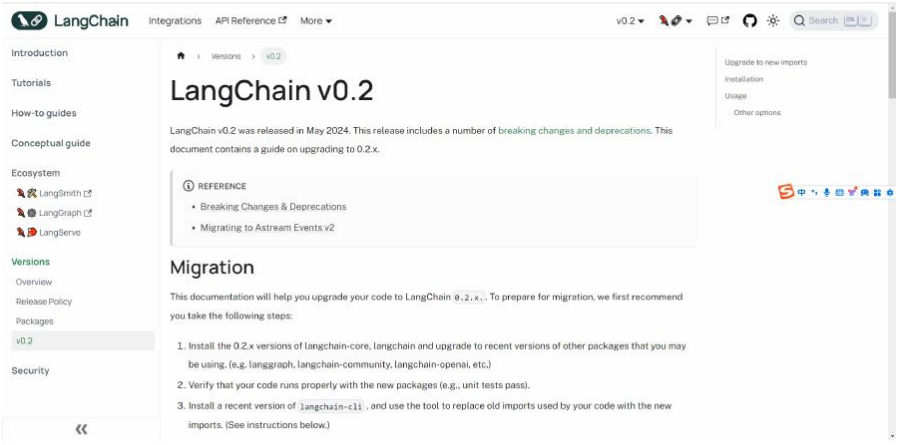
3.1 版本概述（Overview）

在 LangChain 的版本概述中，主要分成 3 个内容：当前版本有什么新功能、未来的版本有什么更新路线、对比上一个版本的变化等。



3.2 版本升级指南（v0.2）

在版本升级指南页面中，提供从旧版本迁移到新版本的详细说明，涵盖了更新哪些包的导入方式、如何配置 langchain-cli、更新的其他配置等信息。

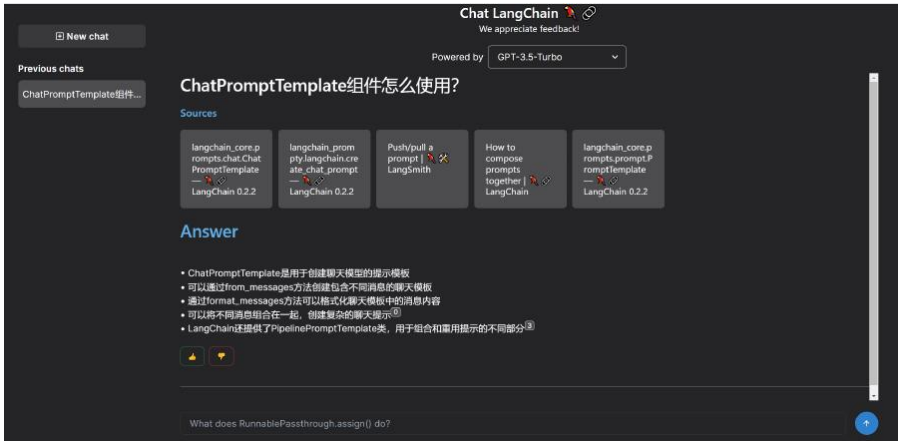


04. Chat LangChain

Chat LangChain 是官方提供的一个文档问答机器人，可以选择不同的模型实现对 LangChain 文档的问答实现快速学习。

例如提问：ChatPromptTemplate组件怎么使用?

对应的问答机器人会快速找到对应知识点关联的文档，并进行简单总结汇总。


New chat
Previous chats
ChatPromptTemplate组件...
Chat LangChain
We appreciate feedback!
Powered by
GPT-3.5-Turbo
ChatPromptTemplate组件怎么使用?
Sources
langchain_core.prompts.chat.ChatPromptTemplate — LangChain 0.2.2
langchain_prompty.langchain.create_chat_prompt — LangChain 0.2.2
Push/pull a prompt | LangSmith
How to compose prompts together | LangChain
langchain_core.prompts.prompt.PromptTemplate — LangChain 0.2.2
Answer
• ChatPromptTemplate是用于创建聊天模型的提示模板
• 可以通过from_messages方法创建包含不同消息的聊天模板
• 通过format_messages方法可以格式化聊天模板中的消息内容
• 可以将不同消息组合在一起，创建复杂的聊天提示[0]
• LangChain还提供了PipelinePromptTemplate类，用于组合和重用提示的不同部分[3]
What does RunnablePassthrough.assign() do?

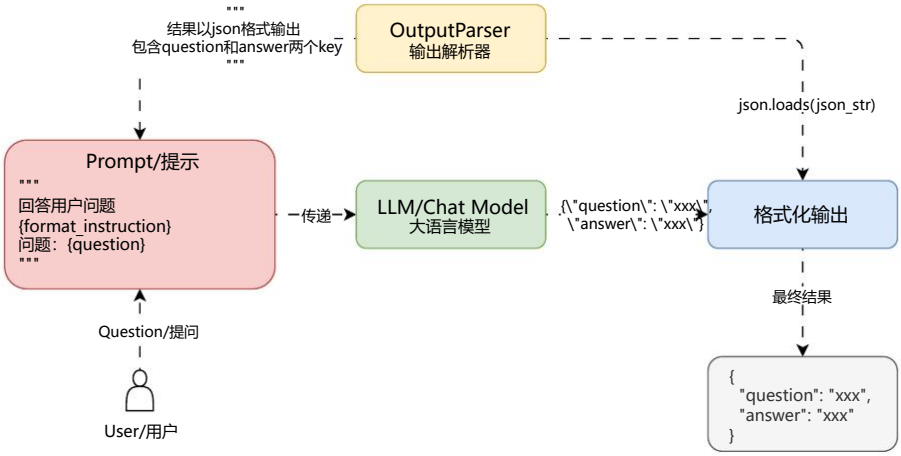
Prompt 组件及使用技巧

Prompt组件及使用技巧

01. Prompt 组件的基本组成

大多数 LLM 应用程序都不会直接将用户输入传递给 LLM。通常，它们会将用户输入添加到一个更大的文本片段中，称为提示模板，该模板提供有关特定任务的附加上下文。

并且 Prompt 是所有 AI 应用交互的起点，例如在 LangChain 中一个最基础的聊天应用机器人的运行流程如下：

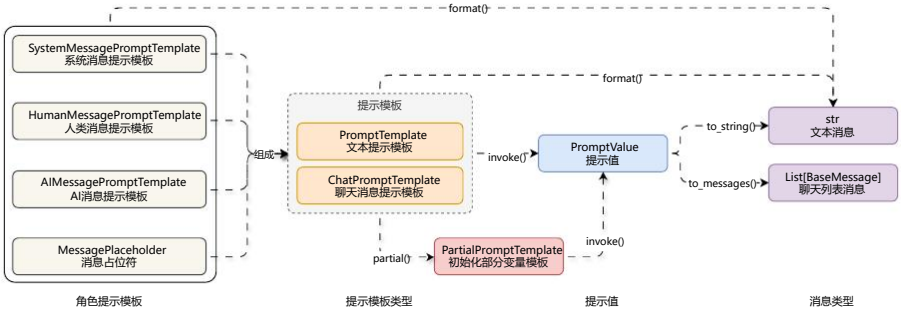


为了适配不同的 LLM，LangChain 封装了 Prompt 组件，并且 Prompt 组件是高可移植性的，同一个 Prompt 可以支持各种 LLM，在切换 LLM 的时候，无需修改 Prompt。在 LangChain 中，Prompt 被分成了两大类：

- Prompt Template：将 Prompt 按照 template 进行一定格式化，针对 Prompt 进行变量处理以及提示词的组合。
- Selectors：根据不同条件去选择不同提示词，或者在不同情况下通过 Selector，选择不同示例去进一步提高 Prompt 支持能力。

本质上 Selectors 只是 Prompt Template 的二次封装，所以在课程中会重点使用 Prompt Template，因为 Selectors 使用范围太窄了，应用场景太小。

对于 Prompt Template，在 LangChain 中，又涵盖了多个子组件，例如：角色提示模板、消息占位符、文本提示模板、聊天消息提示模板、提示、消息等，Prompt Template 的运行流程如下：



不同 Prompt 组件功能的简介：

- PromptTemplate：用于创建文本消息提示模板，用于用于与大语言模型/文本生成模型进行交互。
- ChatPromptTemplate：用于创建聊天消息提示模板，一般用于与聊天模型进行交互。
- MessagePlaceholder：消息占位符，在聊天模型中对不确定是否需要的消息进行占位。
- SystemMessagePromptTemplate：用于创建系统消息提示模板，角色为系统。
- HumanMessagePromptTemplate：用于创建人类消息提示模板，角色为人类。
- AIMessagePromptTemplate：用于创建AI消息提示模板，角色为AI。
- PipelinePromptTemplate：用于创建管道消息，管道消息可以将提示模板作为变量进行快速复用。

Prompt 不同方法的功能简介：

- partial：用于格式化提示模板中的部分变量。
- format：传递变量数据，格式化提示模板为文本消息。
- invoke：传递变量谁，格式化提示模板为提示。
- to_string：将提示/消息提示列表转换成字符串。
- to_messages：用于将消息提示列表转换成字符串。

Prompt 中重载的运算符：

- +运算符：在 Prompt 组件中，对 `+` 运算符使用 `__add__` 方法进行重写，所以几乎所有 Prompt 组件都可以使用 `+` 进行组装拼接。

02. Prompt 组件的格式化

在 Prompt 组件中，默认使用 `f-string` 方法来格式化变量， `f-string` 是 Python 3.6 以后版本中引入的一种特性，用于在字符串中插入表达式的值，语法简洁，直接利用 {} 花括号包裹变量或者表达式，即可执行简单的运算，性能较好，但是只限用在py中。

例如：在传入的提示模板中，使用 `{variable_name}` 来定义提示模板内的变量，在模板中定义了多少个变量，调用时就需要传递多少个变量对应的值，如下。

```
from langchain_core.prompts import PromptTemplate, ChatPromptTemplate

prompt = PromptTemplate.from_template("请将一个关于{subject}的笑话")

print(prompt.format(subject="程序员"))
```

`jinja2` 常被应用于网页开发，与 Flask 和 Django 等框架结合使用。它不仅支持变量替换，还支持其他的控制结构（例如循环和条件语句）以及自定义过滤器和宏等高级功能。此外，它的可用性范围更广，可在多种语境下使用。但与 f-string 不同，使用 jinja2 需要安装相应的库。

```
from langchain_core.prompts import PromptTemplate, ChatPromptTemplate

prompt = PromptTemplate.from_template("请将一个关于{{subject}}的笑话",
template_format="jinja2")

print(prompt.format(subject="程序员"))
```

03. Prompt 使用示例

3.1 基础用法

代码：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
@Time    : 2024/6/8 20:55
@Author  : thezehui@gmail.com
@File    : 1.Prompt基础用法.py
"""
from datetime import datetime

from langchain_core.messages import AIMessage
from langchain_core.prompts import (
    PromptTemplate,
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
    MessagesPlaceholder,
)

prompt = PromptTemplate.from_template("请将一个关于{subject}的冷笑话")
prompt_str = prompt.format(subject="程序员")
print(prompt + "\n")
print(prompt_str + "\n")

print("==========\n")

chat_prompt = ChatPromptTemplate.from_messages([
    ("system", "你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，当前时间是:{now}"),
    MessagesPlaceholder("chat_history"),
    HumanMessagePromptTemplate.from_template("请将一个关于{subject}的冷笑话"),
]).partial(now=datetime.now())
chat_prompt_value = chat_prompt.invoke({
    "subject": "程序员",
    "chat_history": [
        ("human", "我叫慕小课"),
        AIMessage("我是ChatGPT, 有什么能帮到你的么?")
    ],
})
print(chat_prompt)
print(chat_prompt_value)
print(chat_prompt_value.to_messages())
print(chat_prompt_value.to_string())
```

输出：

```
input_variables=['subject'] template='请将一个关于{subject}的冷笑话\n'
请将一个关于程序员的冷笑话

==========
```

```
input_variables=['chat_history', 'subject'] input_types={'chat_history':
typing.List[typing.Union[langchain_core.messages.ai.AIMessage,
langchain_core.messages.human.HumanMessage,
langchain_core.messages.chat.ChatMessage,
langchain_core.messages.system.SystemMessage,
langchain_core.messages.function.FunctionMessage,
langchain_core.messages.tool.ToolMessage]]} partial_variables={'now':
datetime.datetime(2024, 6, 8, 21, 16, 54, 989223)} messages=
[SystemMessagePromptTemplate(prompt=PromptTemplate(input_variables=['now'],
template='你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，当前时间是:{now}')),
MessagesPlaceholder(variable_name='chat_history'),
HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['subject'],
template='请将一个关于{subject}的冷笑话'))]

messages=[SystemMessage(content='你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，当
前时间是:2024-06-08 21:16:54.989223'), HumanMessage(content='我叫慕小课'),
AIMessage(content='我是ChatGPT, 有什么能帮到你的么?'), HumanMessage(content='请将一个关
于程序员的冷笑话')]

[SystemMessage(content='你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，当前时间
是:2024-06-08 21:16:54.989223'), HumanMessage(content='我叫慕小课'),
AIMessage(content='我是ChatGPT, 有什么能帮到你的么?'), HumanMessage(content='请将一个关
于程序员的冷笑话')]

System: 你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，当前时间是:2024-06-08
21:16:54.989223
Human: 我叫慕小课
AI: 我是ChatGPT, 有什么能帮到你的么?
Human: 请将一个关于程序员的冷笑话
```

3.2 字符串提示拼接

代码：

```
from langchain_core.prompts import PromptTemplate

prompt = (
    PromptTemplate.from_template("请将一个关于{subject}的冷笑话")
    + "，让我开心下"
    + "\n使用{language}语言。"
)

print(prompt)

print(prompt.format(subject="程序员", language="中文"))
```

输出：

```
input_variables=['language', 'subject'] template='请将一个关于{subject}的冷笑话，让我
开心下\n使用{language}语言。'

请将一个关于程序员的冷笑话，让我开心下
使用中文语言。
```

3.3 聊天提示拼接

代码：

```
from langchain_core.prompts import ChatPromptTemplate

system_prompt = ChatPromptTemplate.from_messages([
    ("system", "你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，我叫{username}"),
])
human_prompt = ChatPromptTemplate.from_messages([
    ("human", "{query}"),
])

prompt = system_prompt + human_prompt

print(prompt)
print(prompt.format(username="慕小课", query="你好,你是?"))
```

输出：

```
input_variables=['query', 'username'] messages=
[SystemMessagePromptTemplate(prompt=PromptTemplate(input_variables=['username'],
template='你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，我叫{username}')),
HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['query'],
template='{query}'))]

System: 你是OpenAI开发的聊天机器人，请根据用户的提问进行回复，我叫慕小课
Human: 你好,你是?
```

3.4 复用提示模板

代码：

```
from langchain_core.prompts import PipelinePromptTemplate, PromptTemplate

full_template = """{instruction}

{example}

{start}"""
full_prompt = PromptTemplate.from_template(full_template)

# 描述提示模板
instruction_template = "你正在模拟{person}。"
instruction_prompt = PromptTemplate.from_template(instruction_template)

# 示例提示模板
example_template = """下面是一个交互例子:

Q: {example_q}
A: {example_a}"""
example_prompt = PromptTemplate.from_template(example_template)

# 开始提示模板
```

```
start_template = """现在，你是一个真实的人，请回答用户的问题！

Q: {input}
A:"""
start_prompt = PromptTemplate.from_template(start_template)

input_prompts = [
    ("instruction", instruction_prompt),
    ("example", example_prompt),
    ("start", start_prompt),
]

pipeline_prompt = PipelinePromptTemplate(
     final_prompt=full_prompt, pipeline_prompts=input_prompts
)

print(pipeline_prompt.format(
     person="雷军",
     example_q="你最喜欢的汽车是什么?",
     example_a="小米su7",
     input="你最喜欢的手机是什么?"
))
```

输出：

```
你正在模拟雷军。

下面是一个交互例子:

Q: 你最喜欢的汽车是什么?
A: 小米su7

现在，你是一个真实的人，请回答用户的问题！

Q: 你最喜欢的手机是什么?
A:
```

Model 组件及使用技巧

Model组件及使用技巧

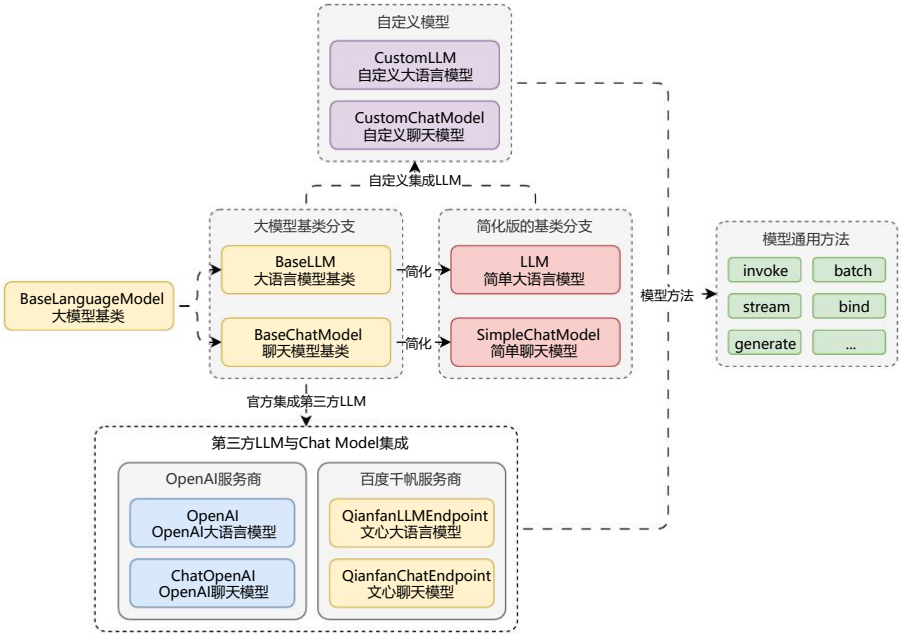
01. Model 组件的基本组成

Model 是 LangChain 的核心组件，但是 LangChain 本身不提供自己的 LLM，而是提供了一个标准接口，用于封装不同类型的 LLM 进行交互，其中 LangChain 为两种类型的模型提供接口和集成：

- LLM：使用纯文本作为输入和输出的大语言模型。
- Chat Model：使用聊天消息列表作为输入并返回聊天消息的聊天模型。

在 LangChain 中，无论是 LLM 亦或者 Chat Model 都可以接受 PromptValue/字符串/消息列表 作为参数，内部会根据模型的类型自动转换成字符串亦或者消息列表，屏蔽了不同模型的差异。

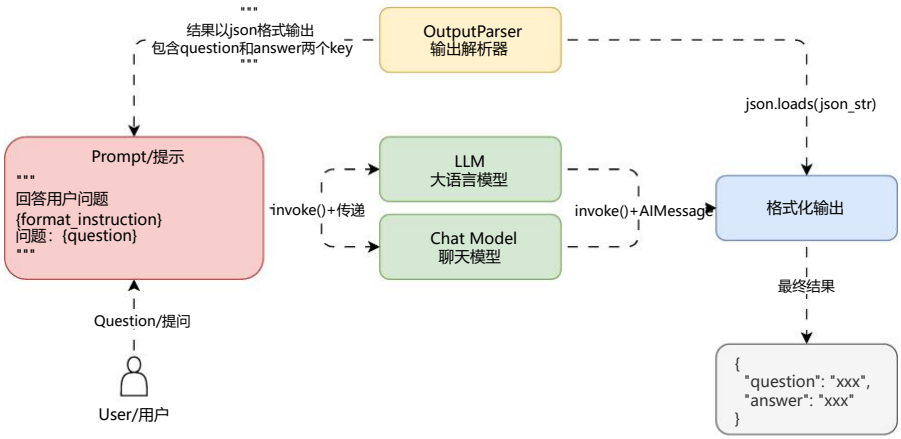
对于 Model 组件，LangChain 有一个模型总基类，并对基类进行了划分：



调用大模型最常用的方法为：

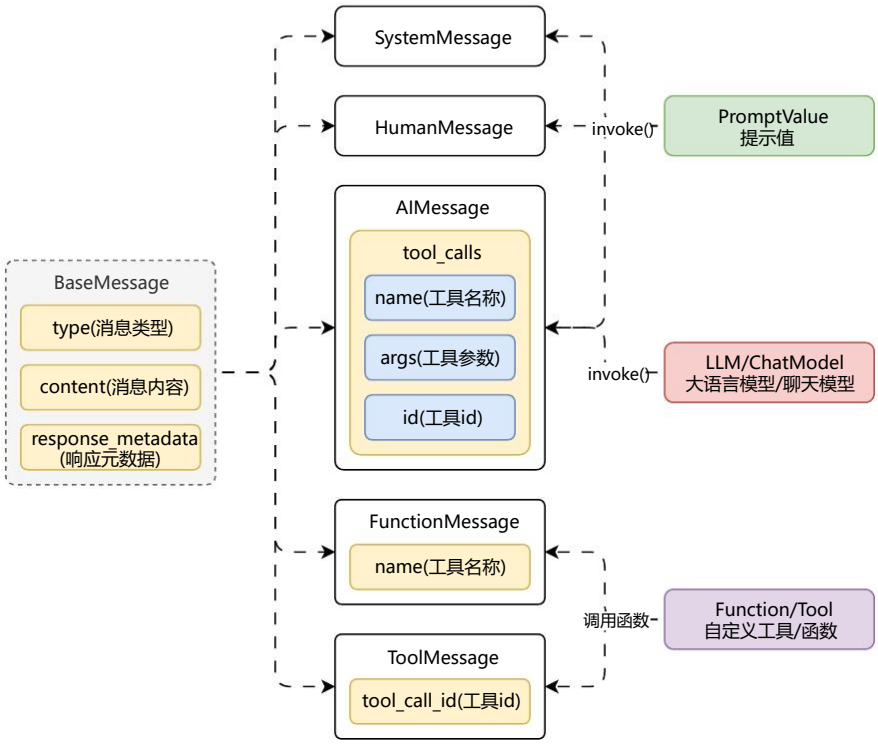
1. invoke：传递对应的文本提示/消息提示，大语言模型生成对应的内容。
2. batch：invoke 的批量版本，可以一次性生成多个内容。
3. stream：invoke 的流式输出版本，大语言模型每生成一个字符就返回一个字符。

基础聊天应用的运行流程更改成如下：



02. Message 组件

在 LangChain 中，Message 是消息组件，并且所有消息都具有 `type(类型)`、`content(内容)`、`response_metadata(响应元数据)`，LangChain 封装的 Message 涵盖了 5 种类型的消息：SystemMessage、HumanMessage、AIMessage、FunctionMessage、ToolMessage，基类及子类如下：



03. Model 组件示例

3.1 LLM与ChatModel使用技巧

代码：

```
from datetime import datetime

import dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

# 1.编排Prompt
prompt = ChatPromptTemplate.from_messages([
    ("system", "你是OpenAI开发的聊天机器人，请回答用户的问题，现在的时间是{now}"),
    ("human", "{query}"),
]).partial(now=datetime.now())

# 2.创建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")

# 3.生成内容
prompt_value = prompt.invoke({"query": "现在是几点，请讲一个关于程序员的冷笑话"})
ai_message = llm.invoke(prompt_value)

# 4.提取内容
print("type:", ai_message.type)
print("content:", ai_message.content)
print("response_metadata:", ai_message.response_metadata)
```

输出：

```
type: ai
content: 现在是18:12。好的，这是一个关于程序员的冷笑话：

为什么程序员总是觉得寂寞？

因为他们总是在和bug独处！
response_metadata: {'token_usage': {'completion_tokens': 53, 'prompt_tokens': 71,
'total_tokens': 124}, 'model_name': 'gpt-3.5-turbo-16k', 'system_fingerprint':
'fp_b28b39ffa8', 'finish_reason': 'stop', 'logprobs': None}
```

3.2 Model批处理

代码：

```
import dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()
```

```
# 1.构建提示模板
prompt = ChatPromptTemplate.from_template("请讲一个关于{subject}的冷笑话")

# 2.构建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")

# 3.批处理获取响应
ai_messages = llm.batch([
    prompt.invoke({"subject": "程序员"}),
    prompt.invoke({"subject": "Python"}),
])

for ai_message in ai_messages:
    print(ai_message.content)
```

输出：

```
当程序员去海滩度假时，他们把沙子编成了二进制。为了放松，他们打算写一个程序，把整个海洋分割成像素并给每个像素着色。
为什么Python喜欢处理字符串?

因为它不喜欢弄得太复杂，总是喜欢把问题拆成小块来处理！
```

3.3 Model流式输出

代码：

```
import dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

# 1.编排Prompt
prompt = ChatPromptTemplate.from_template("你能简单介绍下{subject}么?")

# 2.构建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")

# 3.流式输出
response = llm.stream(prompt.invoke({"subject": "LLM和LLMOps"}))
for chunk in response:
    print(chunk.content, flush=True, end="")
```

输出：

```
LLM 和 LLMOps 都是开发者社区中的术语。LLM 是指"Language Model"，而 LLMOps 则是指与操作相关的语言模型。简单来说，LLM 是一种语言模型，通常用于生成文本、理解语言等任务。而 LLMOps 则是指对语言模型进行操作和管理的实践，包括部署、监控、优化等工作。LLMOps 的目标是确保语言模型在生产环境中的稳定性、性能和安全性。
```

OutputParser 组件及使用技巧

OutputParser 组件及使用技巧

01. 为什么需要输出解析器

我们在使用大语言模型的时候，无论是使用 LangChain，还是直接使用模型的 API，都会遇到大语言模型的输出解析问题，以 OpenAI 的模型为例：

```
from langchain_openai import ChatOpenAI()

llm = ChatOpenAI()

# 示例1
llm.invoke("告诉我1+1等于几?")  # 输出: 1 + 1 等于 2。

# 示例2
llm.invoke("告诉我3个动物的名字。")  # 输出: 好的，这里有三种动物的名字：\n\n1. 狮子\n2. 大熊猫\n3. 斑马

# 示例3
llm.invoke("给我一个json数据, 键为a和b，值为任意的整型。")  # 输出: {\n    "a": 10,\n    "b": 20\n}
```

虽然整体的问题已经回答了，但是返回的内容都是一个字符串，并不是结构化的数据，某些场合下我们需要的仅仅是对应的返回值，而不是多余的内容，就需要对返回的内容进行结构化、格式化或者解析。

例如下方为修改后的提示，尽可能先让大语言模型按照特定的规则输出内容，然后再进行解析

```
from langchain_openai import ChatOpenAI()

llm = ChatOpenAI()

# 示例1, 输出: 2
llm.invoke("告诉我1+1等于几？除了答案其他内容均不要返回。")

# 示例2, 输出: 1.老虎\n2.狮子\n3.斑马
llm.invoke("告诉我3个动物的名字。返回示例：1.狮子\n2.大熊猫")

# 示例3, 输出: {"a": 10, "b": 20}
llm.invoke("给我一个json数据, 键为a和b，值为任意的整型。除了json数据，其他内容均不要返回。")
```

因此，如果需要 LLM 理解你想要的格式，需要向 LLM 告知要输出的结构信息，当 LLM 进行推理并输出之后，我们需要按照和 LLM 约定的格式进行解析，于是就诞生了 `输出解析器` 的概念。

简单拆解：`输出解析器` = `预设提示` + `解析功能`

输出解析器的解析场景：

1. 将非结构化的文本转换为结构化的数据：将文本解析为一个 json 或者是 Python 里的字典等。
2. 截取部分文本：只要结果中的某一部分即可。
3. 自定义处理：在提示中注入其他的指令，告诉语言模型如何去给我返回一个格式的答案，然后解析器可以根据返回提示文本对应的内容做对应的解析。

02. OutputParser 输出解析器

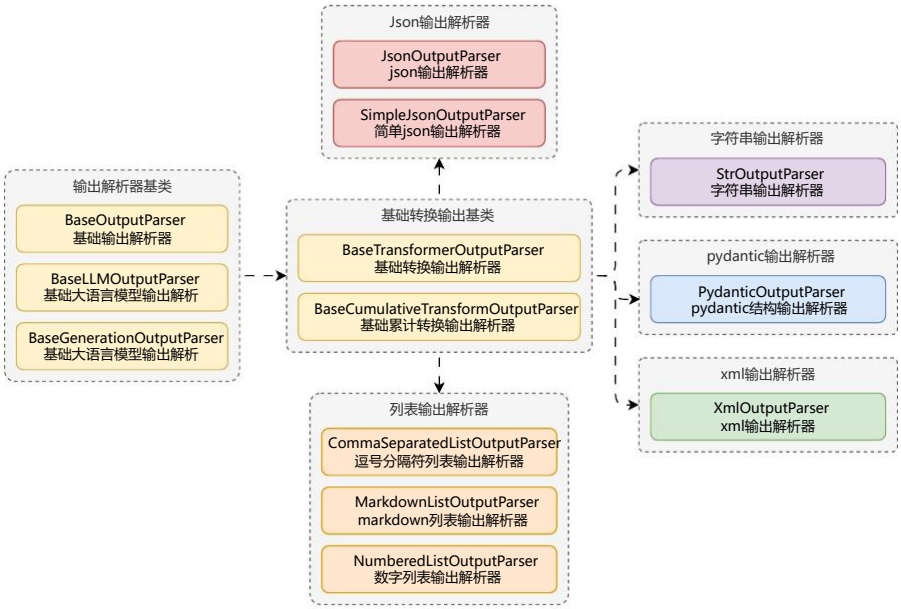
在 LangChain 中预设了大量的输出解析器，可以从StrOutputParser：字符串输出解析器，将原有的内容完整输出。

在 LangChain 中，输出解析器通常包含两个抽象函数的实现，这也是自定义输出解析器需要实现的两个函数：

1. `get_format_instructions`： 用来约定输出的格式，并转换为描述文本。
2. `parse`： 用来解析 LLM 的输出为约定的格式。

但在实际的使用中，输出解析器大部分场景都是解析大语言模型的输出内容，所以可以直接使用 `invoke` 函数进行调用，在 `invoke` 函数的底层，还是调用了 `parse` 函数进行解析。

在 LangChain 中 OutputParser 的基类及子类划分如下：



2.1 StrOutputParser

StrOutputParser 是 LangChain 中使用频率最高的输出解析器，使用也非常简单，将传入的文本原样返回，用于解析得到 AIMessage 中的输出文本，使用示例如下：

```
from langchain_core.output_parsers import StrOutputParser

StrOutputParser().parse("程序员的梦工厂")
```

输出内容：

```
程序员的梦工厂
```

源码实现：

```
def parse(self, text: str) -> str:
    """Returns the input text with no changes."""
    return text
```

2.2 JsonOutputParser

JsonOutputParser 用于将对应的 LLM 输出转换为特定格式的 json 或 Python 字典，在使用前必须定义 BaseModel 类，告知解析器需要解析的数据结构，并将解析器提供的描述文本嵌入到提示中，使用示例如下：

```
import dotenv
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


# 1.构建输出json格式类
class Joke(BaseModel):
    joke: str = Field(description="回答用户的冷笑话")
    punchline: str = Field(description="冷笑话的笑点")


# 2.编排提示词
prompt = ChatPromptTemplate.from_template("回答用户的问题。
\n{format_instructions}\n{query}\n")

# 3.构建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")

# 4.创建json输出解析器
parser = JsonOutputParser(pydantic_object=Joke)

# 5.将格式描述嵌入到prompt中
prompt = prompt.partial(format_instructions=parser.get_format_instructions())

# 6.调用模型并解析
prompt_value = prompt.invoke({"query": "请讲一个关于程序员的冷笑话"})
print("prompt_value:", prompt_value.to_string())

ai_message = llm.invoke(prompt_value)
print("ai_message:", ai_message)

joke_json = parser.invoke(ai_message)
print("joke_json:", joke_json)
```

输出内容：

```
{'joke': '为什么程序员总是冷静的？', 'punchline': '因为他们总是有一堆bug在身边。'}
```

源码实现：

```
def get_format_instructions(self) -> str:
    if self.pydantic_object is None:
        return "Return a JSON object."
    else:
        # Copy schema to avoid altering original Pydantic schema.
        schema = {k: v for k, v in
self._get_schema(self.pydantic_object).items()}

        # Remove extraneous fields.
        reduced_schema = schema
        if "title" in reduced_schema:
            del reduced_schema["title"]
        if "type" in reduced_schema:
            del reduced_schema["type"]
        # Ensure json in context is well-formed with double quotes.
        schema_str = json.dumps(reduced_schema)
        return JSON_FORMAT_INSTRUCTIONS.format(schema=schema_str)

def parse_result(self, result: List[Generation], *, partial: bool = False) ->
Any:
    text = result[0].text
    text = text.strip()
    if partial:
        try:
            return parse_json_markdown(text)
        except JSONDecodeError:
            return None
    else:
        try:
            return parse_json_markdown(text)
        except JSONDecodeError as e:
            msg = f"Invalid json output: {text}"
            raise OutputParserException(msg, llm_output=text) from e
```

03. 其他思路参考

如果使用的模型支持 Function Calling 或者 Tool Calling，可以不使用解析器，而是直接定义一个函数，规定它的函数参数为对应要解析的格式，并强制模型调用这个函数，模型就可以精准无误地按照特定的约束输出。

思考&问题：

1. 如果 `OutputParser` 输出解析失败了，该如何解决？有几种解决方案？
2. 针对小参数的模型，并且没有函数/工具回调，哪一类输出解析器会更稳定？

LCEL 表达式与 Runnable 可运行协议

LCEL表达式与Runnable可运行协议

01. 多组件 invoke 嵌套的缺点

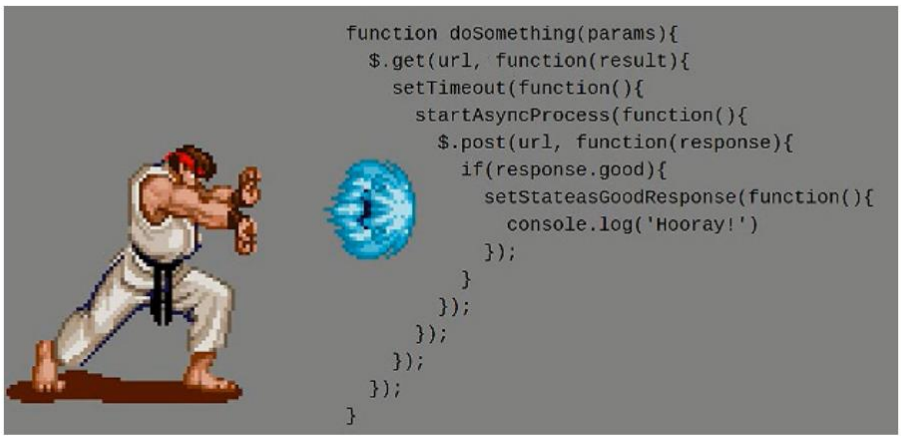
在前面的课时中，我们使用多个组件的 invoke 进行嵌套来创建 LLM 应用，示例代码如下：

```
prompt = ChatPromptTemplate.from_template("{query}")
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
parser = StrOutputParser()

# 获取输出内容
content = parser.invoke(
    llm.invoke(
        prompt.invoke(
            {"query": req.query.data}
        )
    )
)
```

这种写法虽然能实现对应的功能，但是存在很多缺陷：

1. 嵌套式写法让程序的维护性与可阅读性大大降低，当需要修改某个组件时，变得异常困难。
2. 没法得知每一步的具体结果与执行进度，出错时难以排查。
3. 嵌套式写法没法集成大量的组件，组件越来越多时，代码会变成“一次性”代码。

前端代码中的嵌套/回调地狱：



思考：能否将嵌套的写法改成平级的调用，这样就可以屏蔽嵌套带来的大量缺陷。

02. 手写一个"Chain"优化代码

观察发现，虽然 Prompt、Model、OutputParser 分别有自己独立的调用方式，例如：

- `Prompt 组件：` format、invoke、to_string、to_messages。
- `Model 组件：` generate、invoke、batch。
- `OutputParser 组件：` parse、invoke。

但是有一个共同的调用方法：invoke，并且每一个组件的输出都是下一个组件的输入，是否可以将所有组件组装得到一个列表，然后循环依次调用 invoke 执行每一个组件，然后将当前组件的输出作为下一个组件的输入。

源码实现：

```
from typing import Any

import dotenv
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

prompt = ChatPromptTemplate.from_template("{query}")
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
parser = StrOutputParser()


class Chain:
    steps: list = []

    def __init__(self, steps: list):
        self.steps = steps

    def invoke(self, input: Any) -> Any:
        output: Any = input
        for step in self.steps:
            output = step.invoke(output)
            print(step)
            print("执行结果:", output)
            print("================")

        return output


chain = Chain([prompt, llm, parser])

print(chain.invoke({"query": "你好，你是?"}))
```

输出内容：

```
input_variables=['query'] messages=
[HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['query'],
template='{query}'))]
执行结果: messages=[HumanMessage(content='你好，你是?')]
===============
client=<openai.resources.chat.completions.Completions object at
0x000001C6BF694310> async_client=
<openai.resources.chat.completions.AsyncCompletions object at 0x000001C6BF695BD0>
model_name='gpt-3.5-turbo-16k' openai_api_key=SecretStr('**********')
openai_api_base='https://api.xty.app/v1' openai_proxy=''
执行结果: content='你好！我是 ChatGPT，一个由OpenAI开发的人工智能语言模型。我可以回答各种各样
的问题，帮助解决问题，提供信息和创意。有什么我可以帮助你的吗？' response_metadata=
{'token_usage': {'completion_tokens': 72, 'prompt_tokens': 13, 'total_tokens':
85}, 'model_name': 'gpt-3.5-turbo-16k', 'system_fingerprint': 'fp_b28b39ffa8',
'finish_reason': 'stop', 'logprobs': None} id='run-5bf9e183-4b28-4be9-bf65-
ce0ad9590785-0'
===============

执行结果: 你好！我是 ChatGPT，一个由OpenAI开发的人工智能语言模型。我可以回答各种各样的问题，帮
助解决问题，提供信息和创意。有什么我可以帮助你的吗?
===============
你好！我是 ChatGPT，一个由OpenAI开发的人工智能语言模型。我可以回答各种各样的问题，帮助解决问
题，提供信息和创意。有什么我可以帮助你的吗?
```

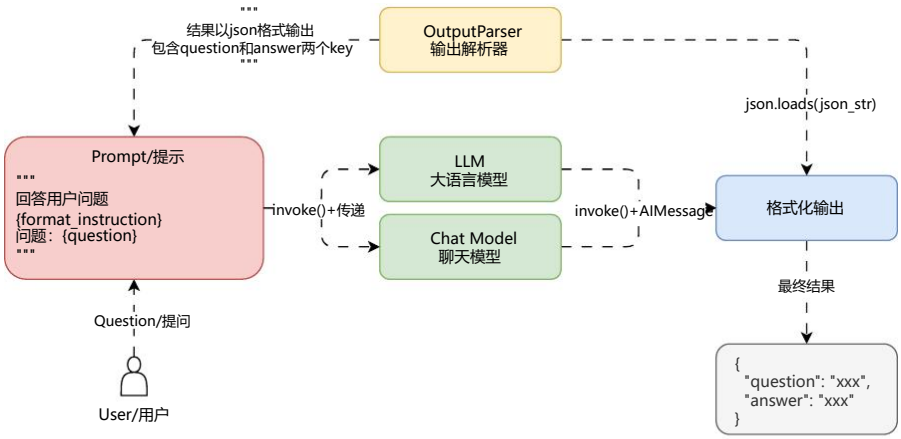
通过自定义"Chain"的方式虽然简化了过程，也支持观察，不过功能过于简陋，在 LangChain 中既然 Prompt、Model、OutputParser 均支持 invoke 方法，底层是否统一了某种规范？并支持以更简单的方式进行调用？

04. Runnable 简介与 LCEL 表达式

为了尽可能简化创建自定义链，LangChain 官方实现了一个 `Runnable` 协议，这个协议适用于 LangChain 中的绝大部分组件，并实现了大量的标准接口，涵盖：

1. `stream`：将组件的响应块流式返回，如果组件不支持流式则会直接输出。
2. `invoke`：调用组件并得到对应的结果。
3. `batch`：批量调用组件并得到对应的结果。
4. `astream`：stream 的异步版本。
5. `ainvoke`：invoke 的异步版本。
6. `abatch`：batch 的异步版本。
7. `astream_log`：除了流式返回最终响应块之外，还会流式返回中间步骤。

除此之外，在 Runnable 中还重写了 `__or__` 和 `__ror__` 方法，这是 Python 中 | 运算符的计算逻辑，所有的 Runnable 组件，均可以通过 | 或者 `pipe()` 的方式将多个组件拼接起来形成一条链。



其中 Runnable 组件的输入类型和输出类型因组件而异：

组件	输入类型	输出类型
Prompt 提示	Dict 字典	PromptValue 提示值
ChatModel 聊天模型	字符串、聊天消息列表、PromptValue 提示值	ChatMessage 聊天消息
LLM 大语言模型	字符串、聊天消息列表、PromptValue 提示值	String 字符串
OutputParser 输出解析器	LLM 或聊天模型的输出	取决于解析器
Retriever 检索器	单个字符串	List of Document 文档列表
Tool 工具	字符串、字典或取决于工具	取决于工具

例如上面自定义"Chain"的写法等同于如下的代码：

```
from typing import Any

import dotenv
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel, RunnablePassthrough
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

prompt = ChatPromptTemplate.from_template("{query}")
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
parser = StrOutputParser()

chain = prompt | llm | parser
# 等价于以下写法
composed_chain_with_pipe = (
```

```
    RunnableParallel({"query": RunnablePassthrough()})
    .pipe(prompt)
    .pipe(llm)
    .pipe(parser)
)

print(chain.invoke({"query": "你好，你是？"}))
```

Runnable 底层的运行逻辑本质上也是将每一个组件添加到列表中，然后按照顺序执行并返回最终结果，核心源码：

```
def invoke(self, input: Input, config: Optional[RunnableConfig] = None) ->
Output:
    from langchain_core.beta.runnables.context import config_with_context

    # setup callbacks and context
    config = config_with_context(ensure_config(config), self.steps)
    callback_manager = get_callback_manager_for_config(config)
    # start the root run
    run_manager = callback_manager.on_chain_start(
        dumpd(self),
        input,
        name=config.get("run_name") or self.get_name(),
        run_id=config.pop("run_id", None),
    )

    # 调用所有步骤并逐个执行得到对应的输出，然后作为下一个的输入
    try:
        for i, step in enumerate(self.steps):
            input = step.invoke(
                input,
                # mark each step as a child run
                patch_config(
                    config, callbacks=run_manager.get_child(f"seq:step:{i+1}")
                ),
            )
    # finish the root run
    except BaseException as e:
        run_manager.on_chain_error(e)
        raise
    else:
        run_manager.on_chain_end(input)
        return cast(Output, input)
```

两个 Runnable 核心类的讲解与使用

两个Runnable核心类的讲解与使用

01. RunnableParallel 并行运行

RunnableParallel 是 LangChain 中封装的支持运行多个 Runnable 的类，一般用于操作 Runnable 的输出，以匹配序列中下一个 Runnable 的输入，起到并行运行 Runnable 并格式化输出结构的作用。

例如 RunnableParallel 可以让我们同时执行多条 Chain，然后以字典的形式返回各个 Chain 的结果，对比每一条链单独执行，效率会高很多。

```
import dotenv
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

# 1.编排2个提示模板
joke_prompt = ChatPromptTemplate.from_template("请讲一个关于{subject}的冷笑话，尽可能
短")
poem_prompt = ChatPromptTemplate.from_template("请写一篇关于{subject}的诗，尽可能短")

# 2.创建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")

# 3.创建输出解析器
parser = StrOutputParser()

# 4.构建两条链
joke_chain = joke_prompt | llm | parser
poem_chain = poem_prompt | llm | parser

# 5.使用RunnableParallel创建并行可运行
map_chain = RunnableParallel(joke=joke_chain, poem=poem_chain)

# 6.运行并行可运行组件得到响应结果
resp = map_chain.invoke({"subject": "程序员"})

print(resp)
```

输出内容：

```
{'joke': '为什么程序员总是用尺子测电脑屏幕？因为他们听说了“像素”是屏幕上的一种“尺寸”。',
'poem': '在代码的海洋里徜徉，\n程序员心怀梦想与创意。\n键盘敲击是旋律，\nbug 是诗歌的瑕疵。
\n\n算法如诗的韵律，\n逻辑是句子的构思。\n编程者如诗人般，\n创造出数字的奇迹。'}
```

除了并行执行，RunnableParallel 还可以用于操作 Runnable 的输出，用于产生符合下一个 Runnable 组件的数据。

例如：用户传递数据，并行执行检索策略得到上下文随后传递给 Prompt 组件，如下。

```
import dotenv
```

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel, RunnablePassthrough
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


def retrieval(query: str) -> str:
    """模拟一个检索器，传入查询query，输出文本"""
    print("执行检索:", query)
    return "我叫慕小课，是一名AI应用开发工程师。"


# 1.编排Prompt
prompt = ChatPromptTemplate.from_template("""请根据用户的提问回答问题，可以参考对应的上下文进行回复。

<context>
{context}
<context>

用户的问题是: {query}""")

# 2.构建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo")

# 3.创建输出解析器
parser = StrOutputParser()

# 4.编排链
chain = RunnableParallel(
    context=retrieval,
    query=RunnablePassthrough(),
) | prompt | llm | parser

# 5.调用链生成结果
content = chain.invoke("你好，我叫什么?")

print(content)
```

输出内容：

```
执行检索: 你好，我叫什么?
你好，你叫慕小课。
```

在创建 RunnableParallel 的时候，支持传递字典、函数、映射、键值对数据等多种方式，RunnableParallel 底层会执行检测并将数据统一转换为 Runnable，核心源码如下：

```
# langchain-core/runnables/base.py

def __init__(
    self,
    steps__: Optional[
```

```
            Mapping[
                str,
                Union[
                    Runnable[Input, Any],
                    Callable[[Input], Any],
                    Mapping[str, Union[Runnable[Input, Any], Callable[[Input],
Any]]],
                ],
            ]
    ] = None,
    **kwargs: Union[
        Runnable[Input, Any],
        Callable[[Input], Any],
        Mapping[str, Union[Runnable[Input, Any], Callable[[Input], Any]]],
    ],
) -> None:
    # 1.检测是否传递字典，如果传递，则提取字段内的所有键值对
    merged = {**steps__} if steps__ is not None else {}
    # 2.传递了键值对，则将键值对更新到merged进行合并
    merged.update(kwargs)
    super().__init__(  # type: ignore[call-arg]
        # 3.循环遍历merged的所有键值对，并将每一个元素转换成Runnable
        steps__={key: coerce_to_runnable(r) for key, r in merged.items()}
    )
```

除此之外，在 Chains 中使用时，可以简写成字典的方式，`__or__` 和 `__ror__` 会自动将字典转换成 RunnableParallel，核心源码：

```
# langchain_core/runnables/base.py

def coerce_to_runnable(thing: RunnableLike) -> Runnable[Input, Output]:
    """Coerce a runnable-like object into a Runnable.

    Args:
        thing: A runnable-like object.

    Returns:
        A Runnable.
    """
    if isinstance(thing, Runnable):
        return thing
    elif is_async_generator(thing) or inspect.isgeneratorfunction(thing):
        return RunnableGenerator(thing)
    elif callable(thing):
        return RunnableLambda(cast(Callable[[Input], Output], thing))
    elif isinstance(thing, dict):
        # 如果类型为字典，使用字典创建RunnableParallel并转换成Runnable格式
        return cast(Runnable[Input, Output], RunnableParallel(thing))
    else:
        raise TypeError(
            f"Expected a Runnable, callable or dict."
            f"Instead got an unsupported type: {type(thing)}"
        )
```

02. RunnablePassthrough 传递数据

除了 RunnablePassthrough，在 LangChain 中，另外一个高频使用的 Runnable 类是 RunnablePassthrough，这个类透传上游参数输入，简单来说，就是可以获取上游的数据，并保持不变或者新增额外的键。

通常与 RunnableParallel 一起使用，将数据分配给映射中的新键。

例如：使用 RunnablePassthrough 来简化 invoke 的调用流程。

```
import dotenv
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

# 1.编排prompt
prompt = ChatPromptTemplate.from_template("{query}")

# 2.构建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")

# 3.创建链
chain = {"query": RunnablePassthrough()} | prompt | llm | StrOutputParser()

# 4.调用链并获取结果
content = chain.invoke("你好，你是")

print(content)
```

输出内容：

```
你好！是的，我是ChatGPT，你需要什么帮助吗?
```

RunnablePassthrough() 获取的是整个输入的内容（字符串或者字典），如果想获取字典内的某个部分，可以使用 `itemgetter` 函数，并传入对应的字段名即可，如下：

```
from operator import itemgetter

chain = {"query": itemgetter("query")} | prompt | llm | StrOutputParser()

content = chain.invoke({"query": "你好，你是？"})
```

除此之外，如果想在传递的数据中添加数据，还可以使用 RunnablePassthrough.assign() 方法来实现快速添加。

例如：为

```
import dotenv
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
```

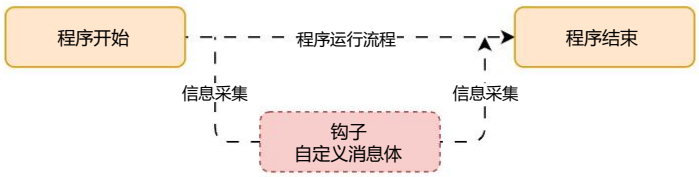
```python
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


def retrieval(query: str) -> str:
    """模拟一个检索器，传入查询query，输出文本"""
    print("执行检索:", query)
    return "我叫慕小课，是一名AI应用开发工程师。"


# 1.编排Prompt
prompt = ChatPromptTemplate.from_template("""请根据用户的提问回答问题，可以参考对应的上下文进行回复。

<context>
{context}
<context>

用户的问题是: {query}""")

# 2.构建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo")

# 3.创建输出解析器
parser = StrOutputParser()

# 4.编排链，RunnablePassthrough.assign写法
chain = (
        RunnablePassthrough.assign(context=lambda query: retrieval(query)) |
        prompt |
        llm |
        parser
)

# 5.调用链生成结果
content = chain.invoke({"query": "你好，我叫什么?"})

print(content)
```

利用回调功能调试链应用 - 让过程更透明

利用回调功能调试链应用-让过程更透明

01. Callback 功能介绍

`Callback` 是 LangChain 提供的回调机制，允许我们在 `LLM` 应用程序的各个阶段使用 `hook`（钩子）。钩子的含义也非常简单，我们把应用程序看成一个一个的处理逻辑，从开始到结束，钩子就是在事件传送到终点前截获并监控事件的传输。



`Callback` 对于记录日志、监控、流式传输等任务非常有用，简单理解， `Callback` 就是记录整个流程的运行情况的一个组件，在每个关键的节点记录响应的信息以便跟踪整个应用的运行情况。

例如：

1. 在 Agent 模块中调用了几次 tool，每次的返回值是什么？
2. 在 LLM 模块的执行输出是什么样的，是否有报错？
3. 在 OutputParser 模块的输出解析是什么样的，重试了几次？

Callback 收集到的信息可以直接输出到控制台，也可以输出到文件，更可以输入到第三方应用，相当于独立的日志管理系统，通过这些日志就可以分析应用的运行情况，统计异常率，运行的瓶颈模块以便优化。

在 LangChain 中，callback 模块中具体实现包括两大功能，对应 `CallbackHandler` 和 `CallbackManager`。

1. CallbackHandler：对每个应用场景比如 Agent 或 Chain 或 Tool 的纪录。
2. CallbackManager：对所有 CallbackHandler 的封装和管理，包括了单个场景的 Handle，也包括运行时整条链路的 Handle。

不过在 LangChain 的底层，这些任务的执行逻辑由回调处理器（`CallbackHandler`）定义。

CallbackHandler 里的各个钩子函数的触发时间如下：

事件	事件触发	相关方法
Chat Model start	当聊天模型启动时	`on_chat_model_start`
LLM start	当大语言模型启动时	`on_llm_start`
LLM new token	当聊天模型或大语言模型生成新 token 时	`on_llm_new_token`
LLM end	当聊天模型或大语言模型结束时	`on_llm_end`
LLM error	当聊天模型或大语言模型发生错误时	`on_llm_error`
Chain start	当链开始运行时	`on_chain_start`

事件	事件触发	相关方法
Chain end	当链结束时	`on_chain_end`
Chain error	当链发生错误时	`on_chain_error`
Tool start	当工具开始运行时	`on_tool_start`
Tool end	当工具结束时	`on_tool_end`
Tool error	当工具发生错误时	`on_tool_error`
Agent action	当智能体执行动作时	`on_agent_action`
Agent finish	当智能结束时	`on_agent_finish`
Retriever start	当检索工具开始运行时	`on_retriever_start`
Retriever end	当检索工具结束时	`on_retriever_end`
Retriever error	当检索工具发生错误时	`on_retriever_error`
Text	运行任意文本时，该接口可以在自定的 Chain、Agent、Tool 上调用该接口。 使用该接口增加了灵活性和可扩展性	`on_text`
Retry	当重试事件时	`on_retry`

在 LangChain 中使用回调，使用 CallbackHandler 几种方式：

1. 在运行 invoke 时传递对应的 config 信息配置 callbacks（推荐）。
2. 在 Chain 上调用 with_config 函数，传递对应的 config 并配置 callbacks（推荐）。
3. 在构建大语言模型时，传递 callbacks 参数（不推荐）。

在 LangChain 中提供了两个最基础的 CallbackHandler，分别是：`StdOutCallbackHandler` 和 `FileCallbackHandler`。

使用示例如下：

```
import dotenv
from langchain_core.callbacks import StdOutCallbackHandler
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

# 1.编排prompt
prompt = ChatPromptTemplate.from_template("{query}")

# 2.创建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
```

```
# 3.构建链
chain = {"query": RunnablePassthrough()} | prompt | llm | StrOutputParser()

# 4.调用链并执行
content = chain.stream(
    "你好，你是？",
    config={"callbacks": [StdOutCallbackHandler()]}
)

for chunk in content:
    pass
```

02. 自定义回调

在 LangChain 中，想创建自定义回调处理器，只需继承 `BaseCallbackHandler` 并实现内部的部分接口即可，例如：

```
from langchain_core.callbacks import StdOutCallbackHandler, BaseCallbackHandler

class LLMOpsCallbackHandler(BaseCallbackHandler):
    """自定义LLMOps回调处理器"""

    def on_llm_start(
            self,
            serialized: Dict[str, Any],
            prompts: List[str],
            *,
            run_id: UUID,
            parent_run_id: Optional[UUID] = None,
            tags: Optional[List[str]] = None,
            metadata: Optional[Dict[str, Any]] = None,
            **kwargs: Any,
    ) -> Any:
        print("on_llm_start serialized:", serialized)
        print("on_llm_start prompts:", prompts)

    def on_llm_new_token(
            self,
            token: str,
            *,
            chunk: Optional[Union[GenerationChunk, ChatGenerationChunk]] = None,
            run_id: UUID,
            parent_run_id: Optional[UUID] = None,
            **kwargs: Any,
    ) -> Any:
        print("on_llm_new_token:", token)
```

LangSmith 平台介绍与使用
从原型到生产

API 接口文档介绍与接口统一

API接口文档介绍与接口统一

01. API 接口文档要素

在项目开发汇总中，前后端项目一般是分离开发的（前后端分离）。应用程序的开发，需要由前后端工程师共同定义接口，编写接口文档，之后大家都根据这个接口文档进行开发，直到项目结束。

API 接口文档带来的便利性：

1. 项目开发过程中前后端工程师有一个统一的文件进行沟通交流开发。
2. 项目维护中或项目人员更迭的时候，方便后期人员查看、维护。

目前市面上大部分公司开发人员使用的接口实现规范有两种：Restful API、RPC。这两种规范有不同的表现形式和适用场景。

- RPC（Remote Procedure Call）：远程过程调用，一般用在微服务架构中，通过接口定义语言让接口像本地函数一样调用，并且接口往往都是统一的，调用者只需要知道函数名和参数即可，以动作为主。
- RESTful API：把服务端提供的所有数据/文件都看成资源，那么通过 API 接口请求数据的操作，本质上就是对资源的操作，以资源名称结合 HTTP 请求动作，就可以说明当前 api 接口的功能是什么，一般用于 Web 应用和对外开放的 API 服务、单体服务。

一份 API 接口文档一般分为接口描述、接口地址、请求方法、请求参数、响应内容、错误代码、示例几个部分。

1. 接口描述：简单描述接口的逻辑和作用。
2. 接口地址：接口的正式 URL 和测试 URL，需求方通过调用接口 URL，获取响应内容。
3. 请求方法：一般来说，接口最常见的请求方法为 GET 和 POST 两种方式，即读接口和写接口。通过这两种方法实现对数据增删改查。
4. 请求参数：即需要请求的字段名的名称和规则，都是哪些字段，字段的类型是什么，是否必填字段等。
5. 响应内容：接口返回的字段名称和规则。
6. 错误代码：对接口的错误用代码进行归类。
7. 示例：请求或者响应的示例，用于辅助接口使用。

02. 项目 API 文档解读

LLMOps 项目的 API 文档以功能模块进行划分，例如：授权认证模块、AI 应用模块、工作流模块、知识库模块、开放 API 模块等，如果资源有多种操作方式，则资源路径为 `复数`，否则为 `单数`。

对于需要授权校验的接口，需要在 `header` 中添加 `Authorization: Bearer access_token`，接口业务共有 6 种响应状态：`success(成功)`、`fail(通用失败)`、`not_found(未找到)`、`unauthorized(未授权)`、`forbidden(无权限)`、`validate_error(数据校验失败)`。

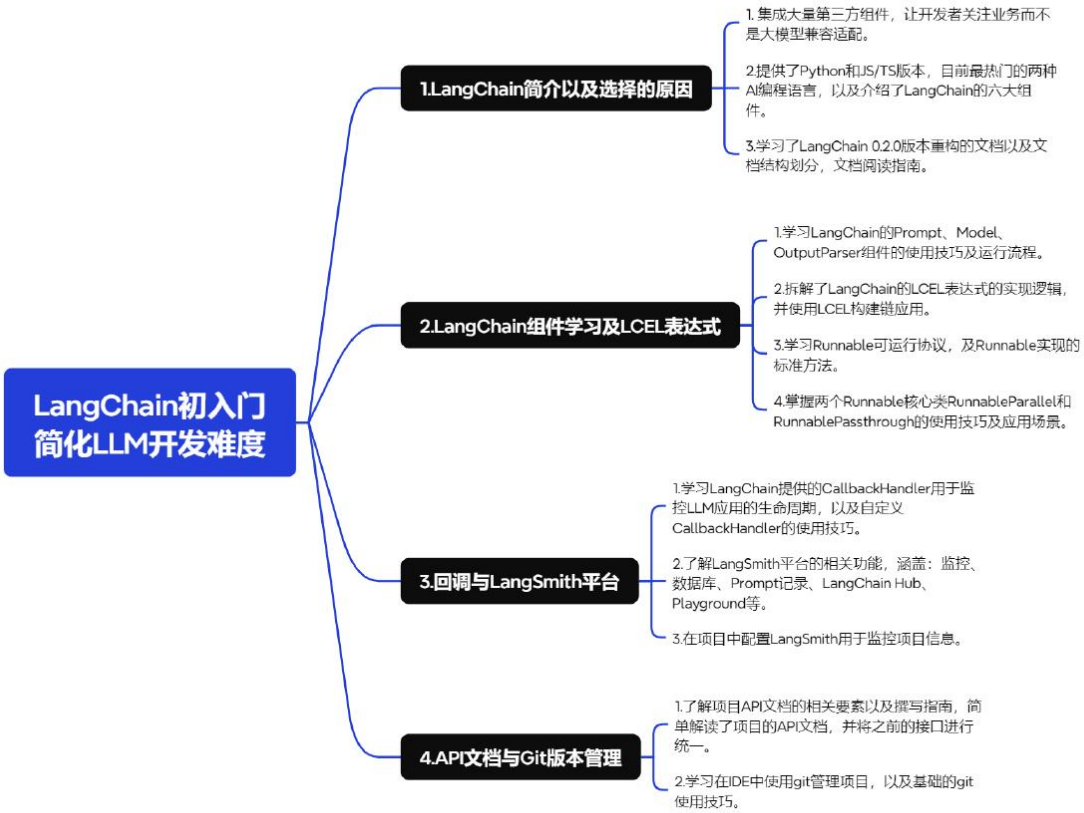
接口固定包含 3 个响应字段：`code`、`data` 和 `message`，分别代表 `业务状态码`、`业务数据` 和 `接口附加信息`。

项目接口文档涵盖：接口说明、接口信息、接口参数、请求示例、响应示例。


产品需求拆解
UI界面设计思路
UI界面接口功能拆分
API文档编写
数据库迁移设计
后端功能逻辑实现
前端页面编写
前端页面API对接

接口文档：


T 项目API文档.md - Typora
文件(F) 编辑(E) 段落(P) 格式(O) 视图(V) 主题(T) 帮助(H)
文件
大纲
04. 上传文件模块
4.1 将文件上传到对象存储模块
4.2 将图片上传到对象存储模块
05. 处理规则模块
5.1 获取默认的文档处理规则
05. 知识库模块
06. 文档模块
07. 应用模块
7.1 创建新应用
7.2 获取AI应用列表分页数据
7.3 查看指定AI应用的详情
7.4 修改AI应用的基础信息
7.5 删除指定的 AI 应用
7.6 更新指定 AI 应用的站点状态
7.7 更新指定 AI 应用的开放 API 状态
7.8 更新/发布指定的 AI 应用配置
7.9 AI应用调试对话
08. 工具提供者模块
8.1 获取所有工具提供者列表信息
8.2 获取指定工具提供者的图标
8.3 获取指定提供者下的工具实体列表
09. 大语言模型提供者模块
}
7.9 AI应用调试对话
接口说明：用于在编排 AI 应用时进行 debug 调试，支持传递对应的 AI 应用配置信息进行实时测试。
接口信息：授权 + POST:/apps/:app_id/debug
接口参数：
请求参数：
app_id -> str：需要调试的 AI 应用 id，格式为 uuid。
query -> str：用户发起的提问信息。
files_ids -> list[str]：附加的文件 id 列表，类型为数组，子元素为 uuid 格式的 id 字符串。
app_config -> dict：AI 应用的编排配置
provider -> str：模型服务提供商。
model -> str：使用的大语言模型。
parameter -> dict：关联大语言模型的参数。
prompt -> str：预编译的提示词。
opening_statement -> str：应用开场对白。
suggested_question -> list：建议问题列表，最多不能超过 5 个。
suggested_question_after_answer -> dict：是否在回答后生成建议问题，类型为字典，内含 enabled 字段（类型为布尔值），用于检测是否开启。
speech_to_text -> dict：是否开启语音转文本输入，类型为字典，内含 enabled 字段（类型为布尔值），用于检测是否开启。
text_to_speech -> dict：是否开启文本转语音，类型为字典，内含 enabled 字段（类型为布尔值），用于检测是
13425 词


LangChain初入门
简化LLM开发难度
1.LangChain简介以及选择的原因
1. 集成大量第三方组件，让开发者关注业务而不是大模型兼容适配。
2.提供了Python和JS/TS版本，目前最热门的两种AI编程语言，以及介绍了LangChain的六大组件。
3.学习了LangChain 0.2.0版本重构的文档以及文档结构划分，文档阅读指南。
2.LangChain组件学习及LCEL表达式
1.学习LangChain的Prompt、Model、OutputParser组件的使用技巧及运行流程。
2.拆解了LangChain的LCEL表达式的实现逻辑，并使用LCEL构建链应用。
3.学习Runnable可运行协议，及Runnable实现的标准方法。
4.掌握两个Runnable核心类RunnableParallel和RunnablePassthrough的使用技巧及应用场景。
3.回调与LangSmith平台
1.学习LangChain提供的CallbackHandler用于监控LLM应用的生命周期，以及自定义CallbackHandler的使用技巧。
2.了解LangSmith平台的相关功能，涵盖：监控、数据库、Prompt记录、LangChain Hub、Playground等。
3.在项目中配置LangSmith用于监控项目信息。
4.API文档与Git版本管理
1.了解项目API文档的相关要素以及撰写指南，简单解读了项目的API文档，并将之前的接口进行统一。
2.学习在IDE中使用git管理项目，以及基础的git使用技巧。