

LangChain: 多任务应用开发



>> 今天的学习目标

LangChain: 多任务应用开发

- LangChain基本概念
- LangChain中的tools (serpapi, llm-math)
- LangChain中的Memory
- Case: 动手搭建本地知识智能客服 (理解ReAct)
- CASE: 工具链组合设计 (LangChain Agent)
- CASE: 搭建故障诊断Agent (LangChain Agent)
- 使用LCEL构建任务链
- AI Agent对比

LangChain基本概念

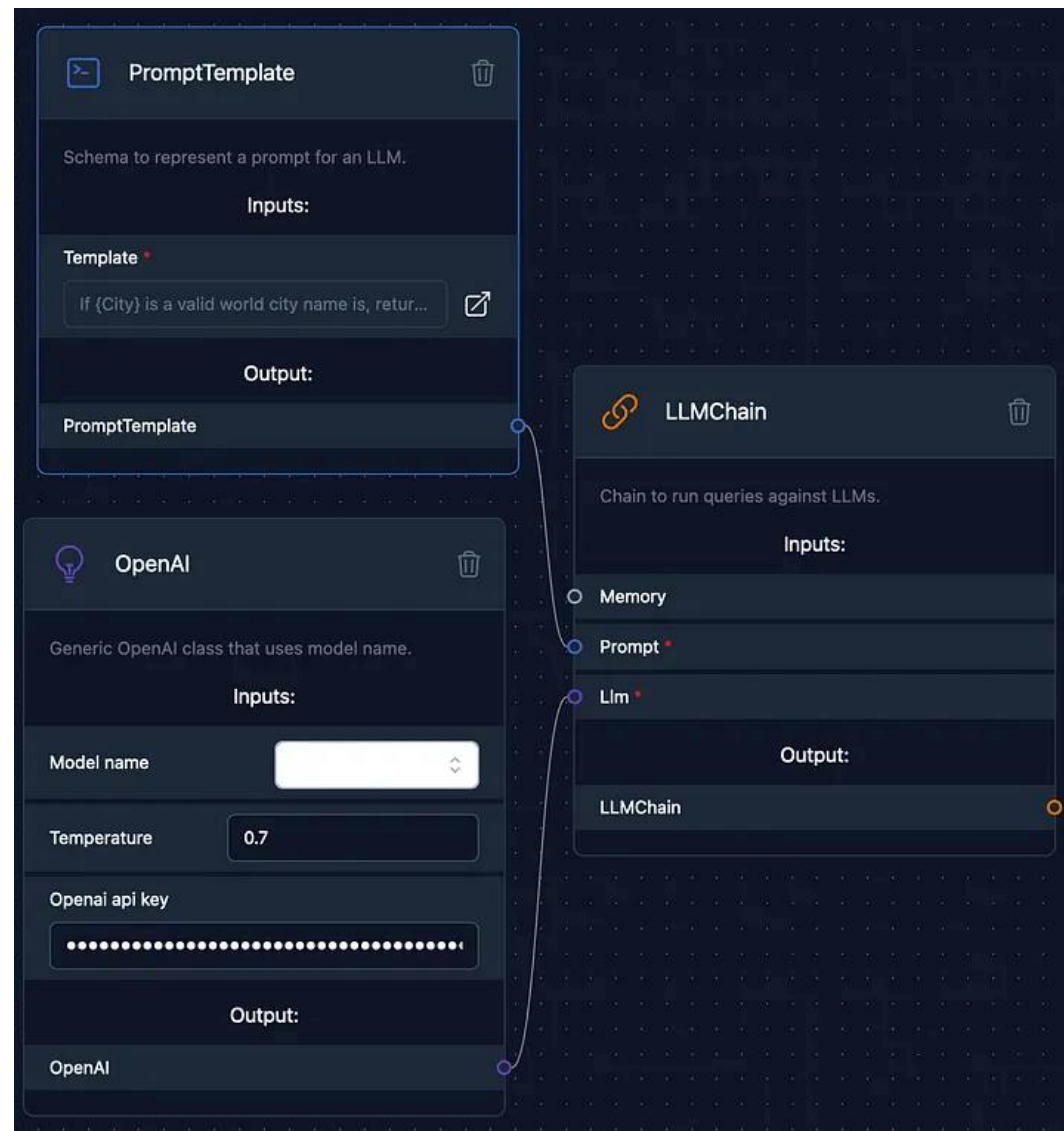
LangChain

LangChain:

- 提供了一套工具、组件和接口
- 简化了创建LLM应用的过程



LangChain



LangChain

LangChain是由多个组件组成的：

- Models：模型，比如GPT-4o
- Prompts：提示，包括提示管理、提示优化和提示序列化
- Memory：记忆，用来保存和模型交互时的上下文

- Indexes：索引，用于结构化文档，方便和模型交互
如果要构建自己的知识库，就需要各种类型文档的加载，转换，长文本切割，文本向量计算，向量索引存储查询等
- Chains：链，一系列对各种组件的调用
- Agents：代理，决定模型采取哪些行动，执行并且观察流程，直到完成为止

LangChain

```
from langchain.prompts import PromptTemplate
from langchain_community.llms import Tongyi
import dashscope
import os

# 从环境变量获取 dashscope 的 API Key
api_key = os.environ.get('DASHSCOPE_API_KEY')
dashscope.api_key = api_key

# 加载 Tongyi 模型
llm = Tongyi(model_name="qwen-turbo",
dashscope_api_key=api_key) # 使用通义千问qwen-turbo模型

# 创建Prompt Template
prompt = PromptTemplate(
    input_variables=["product"],
    template="What is a good name for a company that makes
{product}?",
)
```

新推荐用法：将 prompt 和 llm 组合成一个"可运行序列"

```
chain = prompt | llm
```

使用 invoke 方法传入输入

```
result1 = chain.invoke({"product": "colorful socks"})
```

```
print(result1)
```

```
result2 = chain.invoke({"product": "广告设计"})
```

```
print(result2)
```

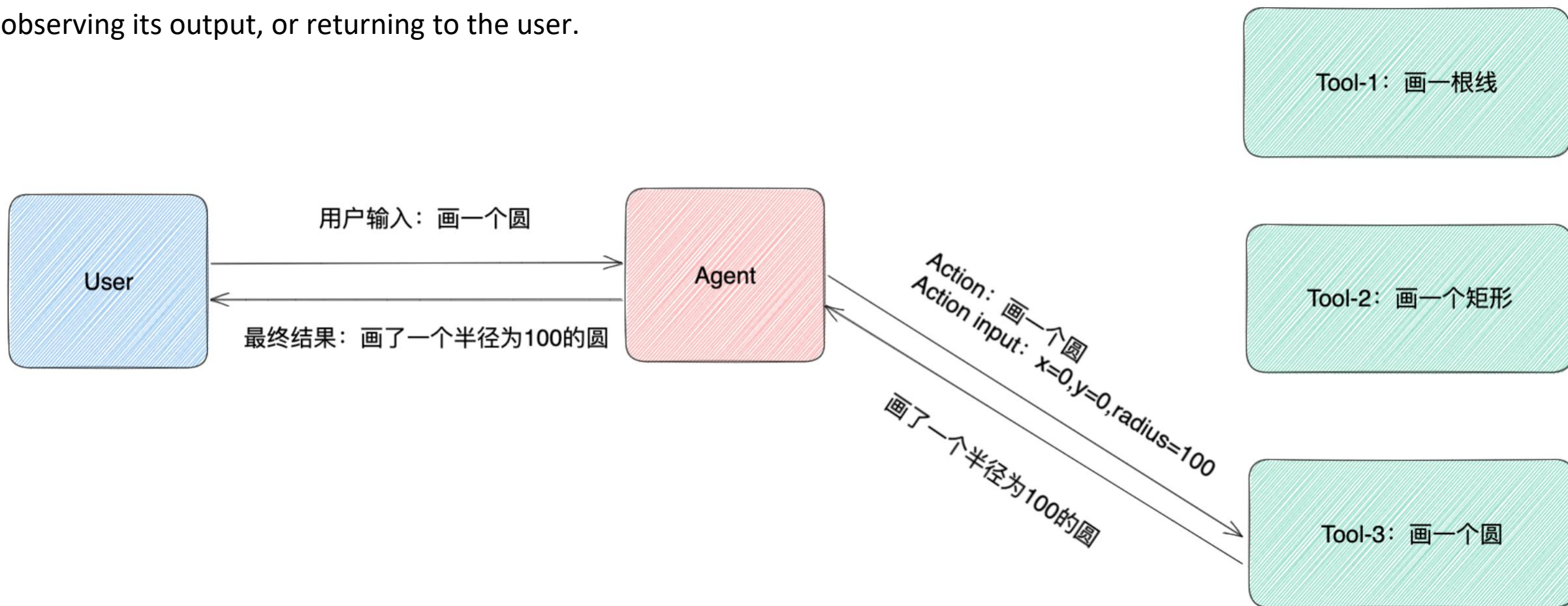
PromptTemplate 是 LangChain 的提示词模板类。

chain = prompt | llm，可以直接将 prompt 的输出作为 llm 的输入，形成一个可运行的链式结构，简化了原先 LLMChain 的写法。

Agent的作用

LangChain官方文档中对Agent的定义：

they use an LLM to determine which actions to take and in what order. An action can either be using a tool and observing its output, or returning to the user.



Tools

Apify

ArXiv API Tool

AWS Lambda API

Shell Tool

Bing Search

ChatGPT Plugins

DuckDuckGo Search

File System Tools

Google Places

Google Search

Google Serper API

Gradio Tools

GraphQL tool

HuggingFace Tools

Human as a tool

IFTTT WebHooks

Metaphor Search

Call the API

Use Metaphor as a tool

OpenWeatherMap API

Python REPL

Requests

SceneXplain

Search Tools

SearxNG Search API

SerpAPI

Twilio

Wikipedia

Wolfram Alpha

YouTubeSearchTool

Zapier Natural Language Actions API

Example with SimpleSequentialChain

在LangChain中集成了一些常用的tools，你也可以自定义自己的工具

Tools

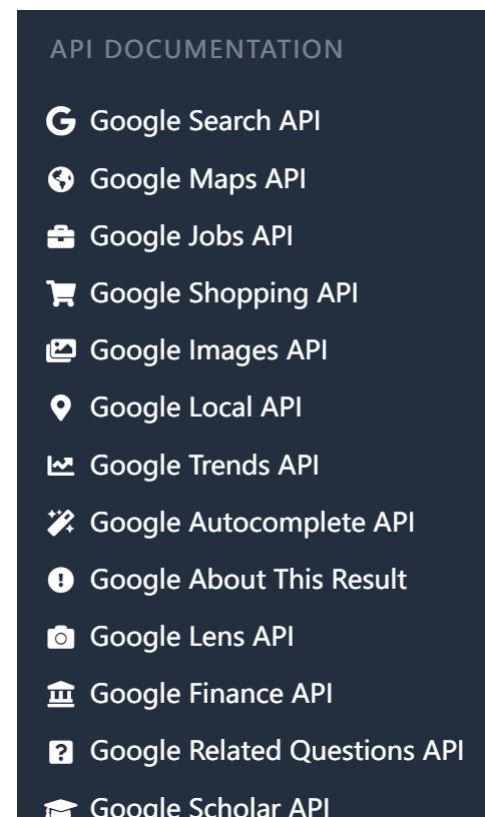
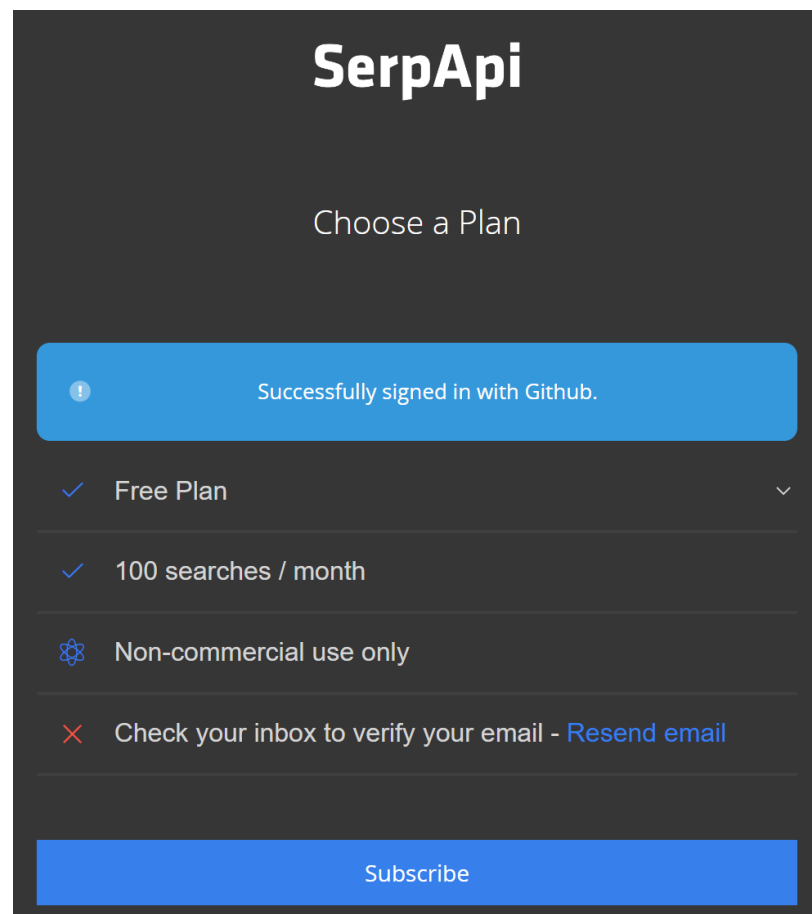
Serpapi工具:

- 支持Google, Baidu, Yahoo, Ebay, YouTube等多种API查询

`pip install google-search-results`

<https://serpapi.com/>

TO DO: 让LangChain可以使用Serpapi工具
(Google搜索API) 进行问题的解答, 比如
`query = "今天是几月几号?历史上的今天有哪些名人出生"`



Tools

```
import os

from langchain.agents import load_tools
from langchain.agents import initialize_agent

from langchain.llms import OpenAI

from langchain.agents import AgentType

# 加载 OpenAI 模型

llm = OpenAI(temperature=0, max_tokens=2048)

# 加载 serpapi 工具

tools = load_tools(["serpapi"])

# 工具加载后需要初始化, verbose=True 代表打印执行详情

agent = initialize_agent(tools, llm,

agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
```

LLMChain2.ipynb

运行 agent

```
agent.run("今天是几月几号?历史上的今天有哪些名人出生")
```

> Entering new AgentExecutor chain...

我需要查找今天的日期和历史上的名人出生日期

Action: Search

Action Input: 今天是几月几号?历史上的今天有哪些名人出生

Observation: 出生编辑 • 1008年: 亨利一世, 法國卡佩王朝國王。(• 1654年: 康熙帝, 清朝皇帝。(• 1677年: 弗朗索瓦絲 • 瑪麗 • 德 • 波旁, 法国奧爾良公爵夫人。(• 1825年: 托马斯 • 亨利 • 赫 ...

Thought: 我现在可以确定今天的日期

Action: Search

Action Input: 今天是几月几号

Observation: 公历(阳历) : 2023年5月27日星期六 农历(阴历) : 2023年四月初九(闰2月) 黄历: 癸卯年丁巳月乙酉日 生肖: 兔 节气: 星座: 双子座 节日: 5节日大全 • 2023年在线日历表.

Thought: 我现在可以确定最终答案

Final Answer: 今天是2023年5月27日, 历史上的今天有亨利一世、康熙帝、弗朗索瓦絲 • 瑪麗 • 德 • 波旁、托马斯 • 亨利 • 赫等名人出生。

> Finished chain.

'今天是2023年5月27日, 历史上的今天有亨利一世、康熙帝、弗朗索瓦絲 • 瑪麗 • 德 • 波旁、托马斯 • 亨利 • 赫等名人出生。'

根据工具的描述, 和请求的string 来决定使用哪个工具

Tools (使用多个Tools)

llm-math工具:

- 给Agent提供数学相关的计算, 比如: 当前温度的0.5次方是多少?

TO DO: 当前北京的温度是多少华氏度, 这个温度的1/4是多少?

Thinking: Agent需要使用哪些tools?

Serpapi, 搜索当前北京的温度

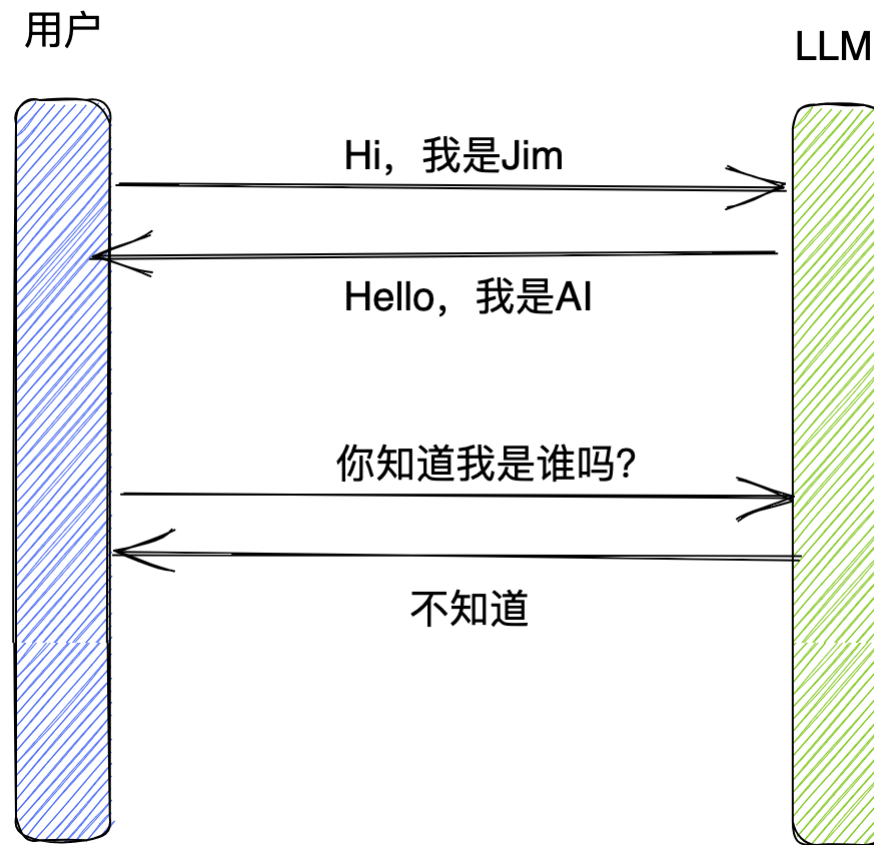
llm-math, 计算这个温度的1/4

Memory

Memory:

- Chains 和 Agent之前是无状态的，如果你想让他能记住之前的交互，就需要引入 内存
- 可以让LLM拥有短期记忆

对话过程中，记住用户的input 和 中间的output



Memory

在LangChain中提供了几种短期记忆的方式：

- **BufferMemory**

将之前的对话完全存储下来，传给LLM

- **BufferWindowMemory**

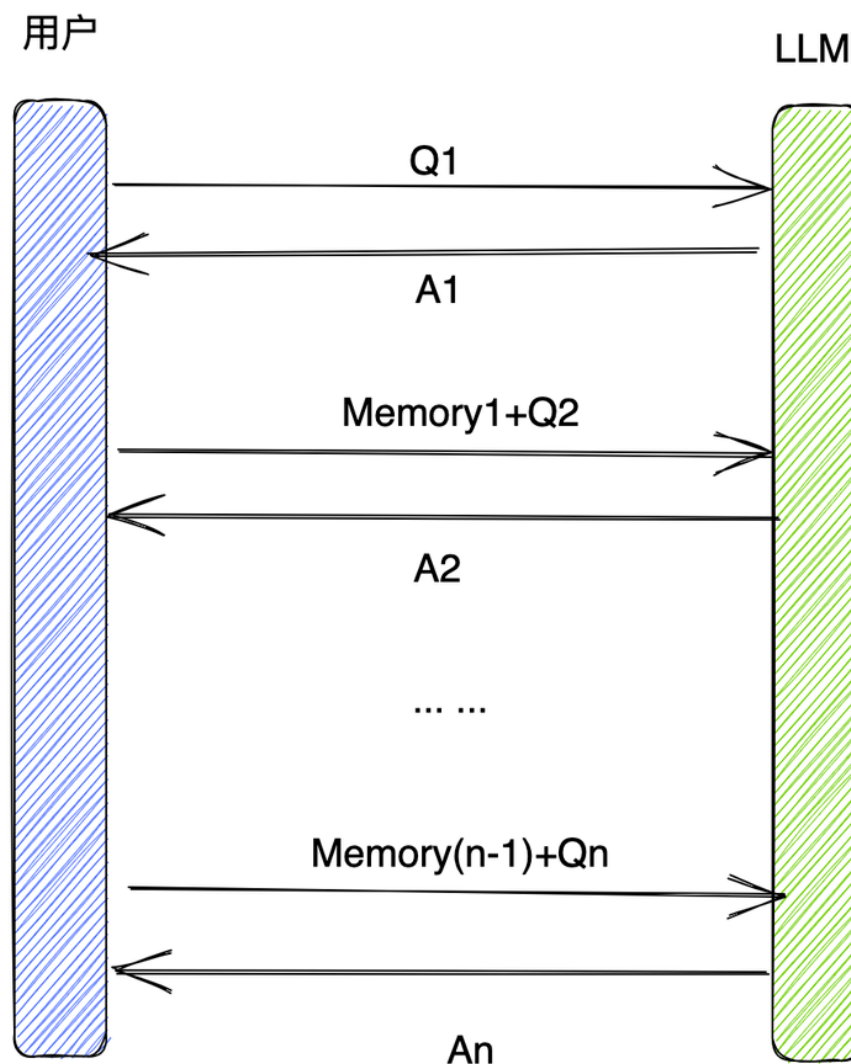
最近的k组对话存储下来，传给LLM

- **ConversationMemory**

对对话进行摘要，将摘要存储在内存中，相当于将压缩过的历史对话传递给LLM

- **VectorStore-backed Memory**

将之前所有对话通过向量存储到VectorDB（向量数据库）中，每次对话，会根据用户的输入信息，匹配向量数据库中最相似的k组对话



Memory

```
# 使用带有memory的ConversationChain
conversation = ConversationChain(llm=llm, verbose=True)
output = conversation.predict(input="Hi there!")
print(output)
```

> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

Human: Hi there!

AI:

> Finished chain.

Hi there! It's nice to meet you. How can I help you today?

```
output = conversation.predict(input="I'm doing well!
Just having a conversation with an AI.")
print(output)
```

> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

Human: Hi there!

AI: Hi there! It's nice to meet you. How can I help you today?

Human: I'm doing well! Just having a conversation with an AI.

AI:

> Finished chain.

That's great! It's always nice to have a conversation with someone new. What would you like to talk about?

Case: 动手搭建本地知识 智能客服

Agent设计

AGENT_TMPL = """按照给定的格式回答以下问题。你可以使用下面这些工具：

{tools}

回答时需要遵循以下用---括起来的格式：

Question: 我需要回答的问题

Thought: 回答这个上述我需要做些什么

Action: "{tool_names}" 中的一个工具名

Action Input: 选择这个工具所需要的输入

Observation: 选择这个工具返回的结果

... (这个 思考/行动/行动输入/观察 可以重复N次)

Thought: 我现在知道最终答案

Final Answer: 原始输入问题的最终答案

现在开始回答，记得在给出最终答案前，需要按照指定格式进行一步一步的推理。

Question: {input}

{agent_scratchpad}

"""

这个模版中，最重要的是：

Thought

Action

Action Input

这 3 个部分是 AI 在Chain中进行自我思考的过程

LangChain使用

请输入您的问题: Model 3怎么样?

[1m> Entering new AgentExecutor chain. [0m

[32;1m[1;3mThought: 我需要找到Model 3的产品描述和公司相关信息来回答这个问题。

Action: 查询产品名称

Action Input: Model 3[0m

Observation: [36;1m[1;3m具有简洁、动感的外观设计，流线型车身和现代化前脸。定价23.19-33.19万[0m

[32;1m[1;3m我需要进一步了解公司相关信息，以便回答这个问题。

Action: 公司相关信息

Action Input: 特斯拉[0m

Observation: [33;1m[1;3m特斯拉最知名的产品是什么？公司以什么而闻名？在车辆中引入了哪些驾驶辅助功能？ [0m

Retrying langchain.llms.openai.completion_with_retry.<locals>._completion_with_retry in 4.0 seconds as it raised APIConnectionError: Error communicating with OpenAI: ('Connection aborted.', RemoteDisconnected('Remote end closed connection without response')).

[32;1m[1;3m根据产品描述和公司相关信息，我可以回答这个问题了。

Final Answer: Model 3是一款具有简洁、动感的外观设计，流线型车身和现代化前脸的车型，定价在23.19-33.19万之间。它是特斯拉公司的产品之一，特斯拉公司以电动汽车和可再生能源技术而闻名，Model 3也是一款电动汽车，同时还引入了多种驾驶辅助功能。[0m

[1m> Finished chain. [0m

Model 3是一款具有简洁、动感的外观设计，流线型车身和现代化前脸的车型，定价在23.19-33.19万之间。它是特斯拉公司的产品之一，特斯拉公司以电动汽车和可再生能源技术而闻名，Model 3也是一款电动汽车，同时还引入了多种驾驶辅助功能。

这个prompt中，最重要的是：

Thought

Action

Action Input

可以重复N次，直到

Thought: 我现在知道最终答案

Final Answer: 原始输入问题的最终答案

Case：动手搭建一个本地知识智能客服

```
# 定义LLM
```

```
llm = Tongyi(model_name="qwen-turbo", dashscope_api_key=DASHSCOPE_API_KEY)
```

```
# 自有数据
```

```
tesla_data_source = TeslaDataSource(llm)
```

```
# 定义的Tools
```

```
tools = [
```

```
    Tool(
```

```
        name="查询产品名称",
```

```
        func=tesla_data_source.find_product_description,
```

```
        description="通过产品名称找到产品描述时用的工具，输入应该是产品名称",
```

```
    ),
```

```
    Tool(
```

```
        name="公司相关信息",
```

```
        func=tesla_data_source.find_company_info,
```

```
        description="当用户询问公司相关的问题，可以通过这个工具了解相关信息",
```

```
    ),
```

```
]
```

让LLM能使用tools，按照规定的prompt执行

在这个prompt中，不断的：

Thought => Action, Action Input => Observation

直到

Thought: 我现在知道最终答案

Final Answer: 原始输入问题的最终答案

5-product_llm.py

Case：动手搭建一个本地知识智能客服

```
# 主过程：可以一直提问下去，直到Ctrl+C
```

```
while True:
```

```
    try:
```

```
        user_input = input("请输入您的问题： ")
```

```
        response = agent_executor.run(user_input)
```

```
        output_response(response)
```

```
    except KeyboardInterrupt:
```

```
        break
```

```
# 定义Agent执行器 = Agent + Tools
```

```
agent_executor = AgentExecutor.from_agent_and_tools(
```

```
    agent=agent, tools=tools, verbose=True
```

```
)
```

```
# 定义Agent = llm_chain + output_parser + tools_names
```

```
agent = LLMSingleActionAgent(
```

```
    llm_chain=llm_chain,
```

```
    output_parser=output_parser,
```

```
    stop=["\nObservation:"],
```

```
    allowed_tools=tool_names,
```

```
)
```

Case：动手搭建一个本地知识智能客服

用户定义的模板

```
agent_prompt = CustomPromptTemplate(  
    template=AGENT_TMPL,  
    tools=tools,  
    input_variables=["input", "intermediate_steps"],  
)
```

Agent返回结果解析

```
output_parser = CustomOutputParser()
```

最常用的Chain, 由LLM + PromptTemplate组成

```
llm_chain = LLMChain(llm=llm, prompt=agent_prompt)
```

定义的工具名称

```
tool_names = [tool.name for tool in tools]
```

定义Agent = llm_chain + output_parser + tools_names

```
agent = LLMSingleActionAgent(  
    llm_chain=llm_chain,  
    output_parser=output_parser,  
    stop=["\nObservation:"],  
    allowed_tools=tool_names,  
)
```

具体代码见 5-product_llm.py

ReAct范式

ReAct范式：

ReAct: Synergizing Reasoning and Acting in Language Models, 2022 <https://arxiv.org/abs/2210.03629>

- 将推理和动作相结合，克服LLM胡言乱语的问题，同时提高了结果的可解释性和可信赖度

Thinking：作者对人类的行为的洞察：

人们在从事一项需要多个步骤的任务时，在步骤和步骤之间，或者动作和动作之间，一般都会有推理过程。

ReAct范式

(1) Hotspot QA

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?

(1a) Standard

Answer: iPod



(1b) CoT (Reason Only)

Thought: Let's think step by step. Apple Remote was originally designed to interact with Apple TV. Apple TV can be controlled by iPhone, iPad, and iPod Touch. So the answer is iPhone, iPad, and iPod Touch.

Answer: iPhone, iPad, iPod Touch



(1c) Act-Only

Act 1: Search[Apple Remote]

Obs 1: The Apple Remote is a remote control ...

Act 2: Search[Front Row]

Obs 2: Could not find [Front Row]. Similar: ...

Act 3: Search[Front Row (software)]

Obs 3: Front Row is a discontinued media center software ...

Act 4: Finish[yes]



(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.

Act 1: Search[Apple Remote]

Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple ... originally designed to control the Front Row media center program ...

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.

Act 2: Search[Front Row]

Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports', 'Front Row (software)', ...]

Thought 3: Front Row is not found. I need to search Front Row (software) .

Act 3: Search[Front Row (software)]

Obs 3: Front Row is a discontinued media center software ...

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.

Act 4: Finish[keyboard function keys]



ReAct范式

Question: 除了苹果遥控器, 还有什么设备能解决苹果遥控器最早设计交互的控制问题?

ReAct = Reason + Act

Apple遥控器最早只能控制Front Row软件。

Front Row软件可以被两种设备控制, Apple遥控器和键盘的功能键。

所以, 正确答案是键盘的功能键。

ReAct这个方式, 简单有效, 可解释性强

机器把想法写出来, 然后结合想法和观察, 做出相应的动作

Though: ...

Action: ...

Observation: ...

Summary

- Agent 的核心是把 LLM 当作推理引擎，让它能使用外部工具，以及自己的长期记忆，从而完成灵活的决策步骤，进行复杂任务
- LangChain 里的 Chain 的概念，是由人来定义的一套流程步骤来让 LLM 执行，可以看成是把 LLM 当成了一个强大的多任务工具

典型的 Agent 逻辑（比如 ReAct）：

- 由 LLM 选择工具。
- 执行工具后，将输出结果返回给 LLM
- 不断重复上述过程，直到达到停止条件，通常是 LLM 自己认为找到答案了

CASE：工具链组合设计 (LangChain)

工具链组合设计

工具链组合设计

在Agent系统中，工具链是实现复杂任务的关键组件。通过将多个工具组合，Agent可以逐步处理复杂的问题。

示例：LangChain中的工具加载与组合

```
from langchain.agents import load_tools
from langchain.llms import OpenAI

llm = Tongyi(model_name="qwen-turbo", dashscope_api_key=DASHSCOPE_API_KEY)
tools = load_tools(["serpapi", "llm-math"], llm=llm)
```

这里加载了搜索引擎工具（serpapi）和数学计算工具（llm-math），并将其与语言模型（LLM）组合使用，以处理需要搜索和计算的任务

CASE：工具链组合

TO DO：工具链组合设计

通过组合三个工具来解决复杂任务，包括：

- **文本分析工具 (TextAnalysisTool)**：负责分析文本内容，提取统计信息和情感倾向。
- **数据转换工具 (DataConversionTool)**：实现不同数据格式之间的转换，目前支持JSON和CSV格式互转。
- **文本处理工具 (TextProcessingTool)**：提供文本处理功能，如统计行数、查找文本和替换文本。

测试用例：

- 1、分析以下文本的情感倾向，并统计其中的行数：'这个产品非常好用，我很喜欢它的设计，使用体验非常棒！\n价格也很合理，推荐大家购买。\\n客服态度也很好，解答问题很及时。'
- 2、将以下CSV数据转换为JSON格式：'name,age,comment\\n张三,25,这个产品很好\\n李四,30,服务态度差\\n王五,28,性价比高'
- 3、分析以下文本，找出所有包含'好'的行，然后将结果转换为JSON格式：'产品A：质量好，价格合理\\n产品B：外观好，但价格贵\\n产品C：性价比好，推荐购买'

CASE：工具链组合

Thinking：整体的工作流程是怎样的？

Step1, 用户提交任务描述

Step2, Agent分析任务，决定使用哪些工具

Step3, Agent通过ReAct框架调用相应工具

Step4, 系统整合各工具结果，生成最终回答

CASE：工具链组合

任务1、分析以下文本的情感倾向，并统计其中的行数：'这个产品非常好用，我很喜欢它的设计，使用体验非常棒！\n价格也很合理，推荐大家购买。\n客服态度也很好，解答问题很及时。'

我需要先分析文本的情感倾向，然后统计其中的行数。

行动: 文本分析

行动输入: '这个产品非常好用，我很喜欢它的设计，使用体验非常棒！\n价格也很合理，推荐大家购买。\n客服态度也很好，解答问题很及时。'

观察: {'word_count': 34, 'char_count': 102, 'sentiment': 'positive'}

思考: 我已经得到了情感倾向，现在需要统计行数。

行动: 文本处理

行动输入: operation='count_lines', content='这个产品非常好用，我很喜欢它的设计，使用体验非常棒！\n价格也很合理，推荐大家购买。\n客服态度也很好，解答问题很及时。'

观察: {'line_count': 3}

思考: 我现在已经有了最终答案

回答: 这段文本的情感倾向是积极的（positive），并且它共有3行。

CASE：工具链组合

2、将以下CSV数据转换为JSON格式：'name,age,comment\n张三,25,这个产品很好\n李四,30,服务态度差\n王五,28,性价比高'

我需要将CSV格式的数据转换为JSON格式。

行动: 数据转换

行动输入: {"input_data": "name,age,comment\n张三,25,这个产品很好\n李四,30,服务态度差\n王五,28,性价比高",
"input_format": "csv", "output_format": "json"}

观察: [{"name": "\u5f20\u4e09", "age": "25", "comment": "\u8fd9\u4e2a\u4ea7\u54c1\u5f88\u597d"}, {"name": "\u674e\u56db",
"age": "30", "comment": "\u670d\u52a1\u6001\u5ea6\u5dee"}, {"name": "\u738b\u4e94", "age": "28", "comment":
"\u6027\u4ef7\u6bd4\u9ad8"}]

思考: 我现在已经有了最终答案

回答: CSV数据已成功转换为JSON格式: [{"name": "张三", "age": "25", "comment": "这个产品很好"}, {"name": "李四", "age": "30",
"comment": "服务态度差"}, {"name": "王五", "age": "28", "comment": "性价比高"}]

CASE：工具链组合

Thinking：工具链是如何创建出来的？

`create_tool_chain()` 函数负责初始化各工具并组合成一个完整的工具链：

```
def create_tool_chain():  
    # 创建工具实例  
  
    text_analysis = TextAnalysisTool()  
    data_conversion = DataConversionTool()  
    text_processing = TextProcessingTool()  
  
    # 将工具包装为LangChain工具格式  
    tools = [Tool(...), Tool(...), Tool(...)]  
  
    # 初始化语言模型  
    llm = Tongyi(model_name="qwen-turbo",  
dashscope_api_key=DASHSCOPE_API_KEY)
```

```
# 创建提示模板  
prompt = PromptTemplate.from_template(...)  
  
# 创建代理  
agent = create_react_agent(llm, tools, prompt)  
  
# 创建代理执行器  
agent_executor = AgentExecutor.from_agent_and_tools(...)  
  
return agent_executor
```

1-simple_toolchain.py

CASE：工具链组合

Thinking：提示词模版是怎样的？

你是一个有用的AI助手，可以使用以下工具：

{tools}

可用工具名称: {tool_names}

使用以下格式：

问题: 你需要回答的问题

思考: 你应该始终思考要做什么

行动: 要使用的工具名称

行动输入: 工具的输入

观察: 工具的结果

... (这个思考/行动/行动输入/观察可以重复多次)

回答: 对原始问题的最终回答

开始!

问题: {input}

思考: {agent_scratchpad}

在 LangChain 中，{agent_scratchpad} 是一个重要的占位符变量，用于在智能体（Agent）运行过程中临时存储和传递中间步骤的信息（比如思考过程、工具调用记录等）。它的作用类似于一个“草稿本”

create_react_agent 是 LangChain 中自定义 ReAct 智能体的底层方法。它允许你自定义提示词（Prompt），并将工具链、推理流程等信息以变量的形式插入到提示词中。当你用 PromptTemplate 创建提示词时，LangChain 会自动把 {tools}、{tool_names} 替换为实际内容。

实现细节

ReAct框架： 使用LangChain的ReAct框架，允许代理执行思考-行动-观察的循环：

1. ****思考****： 分析任务，确定下一步行动
2. ****行动****： 调用适当的工具
3. ****观察****： 分析工具返回的结果
4. ****重复****： 根据观察结果继续思考，直到任务完成

工具注册与调用： 每个工具需要以特定格式注册，包括名称、描述和执行函数：

```
Tool(  
    name=tool_instance.name,  
    func=tool_instance.run,  
    description="工具描述..."  
)
```

实现细节

内存与上下文：系统使用 `ConversationBufferMemory` 存储对话历史，使代理能够参考之前的交互

```
memory=ConversationBufferMemory(memory_key="chat_history")
```

打卡：工具链组合设计



在我们的工作中，会有很多工具可以使用，比如：

文本分析工具 (TextAnalysisTool)

数据转换工具 (DataConversionTool)

文本处理工具 (TextProcessingTool)

.....

使用LangChain将这些工具组合起来进行使用，完成复杂的任务

CASE: 搭建故障诊断Agent (LangChain)

CASE：搭建故障诊断Agent

@1-simple_toolchain.py 理解代码，帮撰写文档.md

好的，我将为你分析 `1-simple_toolchain.py` 文件并创建一个 Markdown 文档来解释它。

这个脚本的核心功能是演示如何使用 LangChain 框架创建一个包含自定义工具的 Agent。它定义了三个自定义工具：`TextAnalysisTool` (文本分析), `DataConversionTool` (数据转换), 和 `TextProcessingTool` (文本处理), 然后将这些工具集成到一个 ReAct Agent 中, 该 Agent 使用通义千问 (Tongyi) LLM 来决定调用哪个工具来完成用户指定的任务。

我现在将为你创建 `README.md` 文件, 其中包含对该脚本的详细说明。

```
README.md (new) +81 -1 •
# 1-simple_toolchain.py 文档

本文档介绍了 `1-simple_toolchain.py` 脚本的功能和用法。该脚本演示了如

## 功能概述

脚本的主要功能包括：

1. **定义自定义工具**：
    * `TextAnalysisTool`：分析文本内容，提取字数、字符数和基本的情感倾向。
    * `DataConversionTool`：在 JSON 和 CSV 数据格式之间进行转换。
    * `TextProcessingTool`：对文本进行处理，如统计行数、查找文本、替换文本。

2. **创建 LangChain Agent**：
    * 使用通义千问 (Tongyi) `qwen-turbo` 模型作为 LLM。
    * 将上述自定义工具集成到 Agent 中。
    * 使用 ReAct (Reasoning and Acting) 模式，让 Agent 能够思考并选择合适的工具来执行任务。
    * 包含对话记忆 (`ConversationBufferMemory`) 以支持多轮交互。

3. **执行任务**：
    * 提供 `process_task` 函数，接收用户输入的任务描述。
    * 使用创建的 Agent Executor 来处理任务，并返回最终结果。
```



```
README.md > # 1-simple_toolchain.py 文档
1 # 1-simple_toolchain.py 文档
2
3 本文档介绍了 `1-simple_toolchain.py` 脚本的功能和用法。该脚本
  演示了如何使用 LangChain 框架创建一个集成自定义工具的智能
  Agent。
4
5 ## 功能概述
6
7 脚本的主要功能包括：
8
9 1. **定义自定义工具**：
10    * `TextAnalysisTool`：分析文本内容，提取字数、字符数和基
      本的情感倾向。
11    * `DataConversionTool`：在 JSON 和 CSV 数据格式之间进
      行转换。
12    * `TextProcessingTool`：对文本进行处理，如统计行数、查
      找文本、替换文本。
13 2. **创建 LangChain Agent**：
14    * 使用通义千问 (Tongyi) `qwen-turbo` 模型作为 LLM。
15    * 将上述自定义工具集成到 Agent 中。
16    * 使用 ReAct (Reasoning and Acting) 模式，让 Agent 能
      够思考并选择合适的工具来执行任务。
17    * 包含对话记忆 (`ConversationBufferMemory`) 以支持多轮
      交互。
18 3. **执行任务**：
19    * 提供 `process_task` 函数，接收用户输入的任务描述。
20    * 使用创建的 Agent Executor 来处理任务，并返回最终结果。
21
```

CASE：搭建故障诊断Agent

帮我搭建一个网络故障诊断类的Agent，工具之间存在串联，第一个工具的输出是第二个工具的输入；如果有10个工具的话，会有很多输入输出的关系。

希望Agent，在适当的时候调用适合的工具；

===

可以参考@1-simple_toolchain.py 编写新的Python

Let's create the new file 2-network_diagnosis_agent.py.

```
2-network_diagnosis_agent.py +281 -1 •10 ✓  
import os  
import json  
import re  
from typing import List, Union, Dict, Any, Optional  
  
from langchain.agents import Tool, AgentExecutor, create_react_ag  
from langchain.prompts import PromptTemplate  
from langchain.schema import AgentAction, AgentFinish  
from langchain_community.llms import Tongyi  
from langchain.memory import ConversationBufferMemory
```

--- 环境设置 ---

确保设置了您的通义千问 API 密钥

您可以通过环境变量 DASHSCOPE_API_KEY 设置，或者直接在这里修改

> 10 linter errors

Auto-fix



2-network_diagnosis_agent.py > ...

```
1 import os  
2 import json  
3 import re  
4 from typing import List, Union, Dict, Any, Optional  
5  
6 from langchain.agents import Tool, AgentExecutor, create_reac  
7 from langchain.prompts import PromptTemplate  
8 from langchain.schema import AgentAction, AgentFinish  
9 from langchain_community.llms import Tongyi  
10 from langchain.memory import ConversationBufferMemory  
11  
12 # --- 环境设置 ---  
13 # 确保设置了您的通义千问 API 密钥  
14 # 您可以通过环境变量 DASHSCOPE_API_KEY 设置，或者直接在这里修改  
15 # DASHSCOPE_API_KEY = os.getenv('DASHSCOPE_API_KEY', '您的API'  
16 # 为方便演示，我们直接在此处硬编码（请注意在生产环境中保护好您的密  
17 DASHSCOPE_API_KEY = 'sk-882e296067b744289acf27e6e20f3ec0' # 请  
18  
19 # --- 自定义网络诊断工具 ---  
20
```

Qodo Gen: Options | Test this class

```
21 class PingTool:
```

```
22     """执行 Ping 操作以检查与目标的网络连通性。"""
```

Qodo Gen: Options | Test this method |

```
23     def __init__(self):
```

```
24         self.name = "网络连通性检查 (Ping)"
```

```
25         self.description = "检查本机到指定主机名或 IP 地址的网络
```

```
26
```

Qodo Gen: Options | Test this method |

```
27     def run(self, target: str) -> str:
```

```
28         """模拟执行 ping 命令。
```

CASE：搭建故障诊断Agent

帮我理解 @2-network_diagnosis_agent.py 并撰写项目逻辑 .md

Thought for 11 seconds

network_diagnosis_agent_logic.md +137 -1 ✓

```
# 网络故障诊断 Agent 项目逻辑

本文档详细介绍了 `2-network_diagnosis_agent.py` 的项目逻辑，它是一个

## 1. 项目概述

该项目创建了一个智能 Agent，能够根据用户描述的网络问题，通过逻辑分析和工具调用，一步步诊断网络故障，最终给出可能的问题原因和解决方案。

关键特点：
- 使用 通义千问 (Tongyi) 作为底层大模型
- 采用 ReAct (Reasoning and Acting) 推理模式
- 提供多种模拟的网络诊断工具
- 实现了工具之间的串联调用，其中一个工具的输出可以成为另一个工具的输入
- 具备会话记忆功能
```

网络故障诊断 Agent 项目逻辑

本文档详细介绍了 `2-network_diagnosis_agent.py` 的项目逻辑，它是一个使用 LangChain 框架和大型语言模型（LLM）构建的网络故障诊断智能系统。

1. 项目概述

该项目创建了一个智能 Agent，能够根据用户描述的网络问题，通过逻辑分析和工具调用，一步步诊断网络故障，最终给出可能的问题原因和解决方案。

关键特点：

- 使用 **通义千问 (Tongyi)** 作为底层大模型
- 采用 **ReAct (Reasoning and Acting)** 推理模式
- 提供多种模拟的**网络诊断工具**
- 实现了工具之间的**串联调用**，其中一个工具的输出可以成为另一个工具的输入
- 具备**会话记忆**功能

2. 系统架构

系统架构由以下核心组件构成：

...

用户问题 → Agent (LLM + 推理框架) → 工具集 → 诊断结果

...

CASE：搭建故障诊断Agent

3. 诊断工具详解

3.1 网络连通性检查工具 (PingTool)

- **功能**：检查从本机到指定主机名或 IP 的网络连通性
- **输入**：目标主机名或 IP 地址
- **输出**：连通性状态（成功/失败）和网络延迟
- **使用场景**：验证网络连接是否通畅，检测网络延迟

3.2 DNS 解析查询工具 (DNSTool)

- **功能**：将主机名解析为 IP 地址
- **输入**：需要解析的主机名
- **输出**：解析后的 IP 地址或解析失败信息
- **使用场景**：诊断 DNS 解析问题，为后续的连通性测试提供 IP 地址

3.3 本地网络接口检查工具 (InterfaceCheckTool)

- **功能**：检查本地网络接口的状态
- **输入**：（可选）接口名称
- **输出**：接口状态信息（是否启用、IP 地址、子网掩码等）
- **使用场景**：检查本地网络配置是否正确

3.4 网络日志分析工具 (LogAnalysisTool)

- **功能**：在系统或应用日志中搜索网络相关问题
- **输入**：关键词和可选的时间范围
- **输出**：匹配的日志条目
- **使用场景**：查找历史网络错误记录，发现非实时故障的线索

CASE：搭建故障诊断Agent

4.1 工具链接示例

系统能够实现工具间的串联调用，例如：

...

问题：无法访问网站 www.example.com

↓

DNS解析查询(www.example.com) → 返回 IP: 93.184.216.34

↓

网络连通性检查(93.184.216.34) → 返回连接超时

↓

本地网络接口检查() → 返回接口正常

↓

网络日志分析("timeout") → 发现相关错误日志

↓

诊断结论：可能是网络路由问题或目标服务器不可用

...

CASE：搭建故障诊断Agent

这个例子中，可以使用 `AgentType.ZERO_SHOT_REACT_DESCRIPTION`

这是 LangChain 框架内置的一种 Agent 类型，属于“Zero-Shot ReAct”范式。

Zero-Shot：指的是大模型在没有额外训练或示例的情况下，直接根据提示词（Prompt）和工具描述来推理如何调用工具。

ReAct：是一种“思考-行动-观察”循环范式（Reasoning + Acting），即

- 模型会先思考（Thought），
- 再决定调用哪个工具（Action），
- 然后观察工具的输出（Observation），
- 再继续推理，直到得出结论。

2-network_diagnosis_agent.py

CASE：搭建故障诊断Agent

AgentType.ZERO_SHOT_REACT_DESCRIPTION 的特点：

自动读取你传入的工具（Tool）列表，每个工具的 name 和 description 会被自动拼接到系统提示词中。

Agent 会根据用户输入和工具描述，自动决定调用哪个工具、如何调用。

你不需要自己写复杂的 Prompt，LangChain 内部会自动生成一份标准的英文提示词，格式大致如右侧：

Answer the following questions as best you can. You have access to the following tools:

<工具1名称>: <工具1描述>

<工具2名称>: <工具2描述>

...

Use the following format:

Question: the input question you must answer

Thought: you should always think about what to do

Action: the action to take, should be one of [工具1, 工具2, ...]

Action Input: the input to the action

Observation: the result of the action

... (this Thought/Action/Action Input/Observation can repeat N times)

Thought: I now know the final answer

Final Answer: the final answer to the original input question

2-network_diagnosis_agent.py

CASE：搭建故障诊断Agent

需要检查本机到 www.example.com 的网络连通性。

Action: 网络连通性检查 (Ping)

Action Input: www.example.com--- 模拟执行 Ping: www.example.com -

--

Observation: Ping www.example.com 成功：延迟 59ms。

Thought:网络连通性检查显示可以成功访问 www.example.com，因此问题可能不在于网络连接。接下来我需要检查 DNS 解析是否正常。

Action: DNS解析查询

Action Input: www.example.com--- 模拟 DNS 查询: www.example.com

Observation: DNS 解析 www.example.com 成功：IP 地址是 93.184.216.34

Thought:DNS 解析查询显示 www.example.com 的 IP 地址解析正常。既然网络连通性和 DNS 解析都没有问题，可能是其他因素导致浏览器连接超时。

Action: 网络日志分析

Action Input: keywords: 'timeout', time_range: '过去1小时'--- 模拟分析日志: 关键词='keywords: 'timeout', time_range: '过去1小时', 时间范围='过去1小时' ---

Observation: 在过去1小时的日志中找到 3 条与 'keywords: 'timeout', time_range: '过去1小时' 相关的条目：

- [Error] 连接到 10.0.0.88:8080 超时
- [Warning] 对 api.external.com 的请求超时
- [Error] 内部服务通信超时

Thought:网络日志分析显示有几条连接超时的记录，但它们与外部服务或内部网络相关，可能并不直接涉及 www.example.com。由于网络连通性和 DNS 解析都没有问题，浏览器无法访问 www.example.com 的原因可能与其他因素有关，例如防火墙规则、代理设置或浏览器配置。

Final Answer: 您的网络连通性和 DNS 解析均正常，但仍然无法通过浏览器访问 www.example.com。建议您.....

LCEL构建任务链

LCEL构建任务链

LCEL 是 LangChain 推出的链式表达式语言，支持用"`|`"操作符将各类单元（如Prompt、LLM、Parser等）组合。

每个"`|`"左侧的输出会自动作为右侧的输入，实现数据流式传递。

优势：

- 代码简洁，逻辑清晰，易于多步任务编排。
- 支持多分支、条件、并行等复杂链路。
- 易于插拔、复用和调试每个子任务。

典型用法：

- 串联：`A | B | C`，A的输出传给B，B的输出传给C。
- 分支：`{"x": A, "y": B}`，并行执行A和B。
- 支持流式：如`.stream()`方法可边生成边消费。

LCEL构建任务链

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.llms import Tongyi # 导入通义千问Tongyi模型
from langchain_core.output_parsers import StrOutputParser

llm = Tongyi(model_name="qwen-turbo", dashscope_api_key=DASHSCOPE_API_KEY, stream=True)
# 定义三个子任务： 翻译->处理->回译
translate_to_en = ChatPromptTemplate.from_template("Translate this to English: {input}") | llm | StrOutputParser()
process_text = ChatPromptTemplate.from_template("Analyze this text: {text}") | llm | StrOutputParser()
translate_to_cn = ChatPromptTemplate.from_template("Translate this to Chinese: {output}") | llm | StrOutputParser()
# 组合成多任务链
workflow = {"text": translate_to_en} | process_text | translate_to_cn
# 使用stream方法，边生成边打印
for chunk in workflow.stream({"input": "北京有哪些好吃的地方，简略回答不超过200字"}):
    print(chunk, end="", flush=True)
print() # 换行
```

3-lcel-demo.py

LCEL构建任务链

TO DO: 将工具链组合 1-simple_toolchain.py, 改成 LCEL任务链的形式

```
# 工具链 (LCEL风格)
```

```
tools = {
```

```
    "文本分析": RunnableLambda(lambda x: text_analysis.run(x["text"])),
```

```
    "数据转换": RunnableLambda(lambda x: data_conversion.run(x["input_data"], x["input_format"], x["output_format"])),
```

```
    "文本处理": RunnableLambda(lambda x: text_processing.run(x["operation"], x["content"], **x.get("kwargs", {}))),
```

```
}
```

```
# LCEL任务链
```

```
def lcel_task_chain(task_type, params):
```

```
    if task_type not in tools:
```

```
        return "不支持的工具类型"
```

```
    return tools[task_type].invoke(params)
```

2-simple_toolchain.py

LCEL构建任务链

示例1：文本分析

```
result1 = lcel_task_chain("文本分析", {"text": "这个产品非常好用，我很喜欢它的设计，使用体验非常棒！"})  
print("示例1结果：", result1)
```

示例2：数据格式转换

```
csv_data = "name,age,comment\n张三,25,这个产品很好\n李四,30,服务态度差\n王五,28,性价比高"  
result2 = lcel_task_chain("数据转换", {"input_data": csv_data, "input_format": "csv", "output_format": "json"})  
print("示例2结果：", result2)
```

示例3：文本处理

```
text = "第一行内容\n第二行内容\n第三行内容"  
result3 = lcel_task_chain("文本处理", {"operation": "count_lines", "content": text})  
print("示例3结果：", result3)
```

LCEL构建任务链

示例4：多步串联（先文本分析，再统计行数）

先分析文本情感，再统计文本行数，展示LCEL链式组合

text4 = "这个产品非常好用，我很喜欢它的设计，使用体验非常棒！\n价格也很合理，推荐大家购买。\n客服态度也很好，解答问题很及时。"

第一步：文本分析

```
analysis_result = lcel_task_chain("文本分析", {"text": text4})
```

第二步：统计行数

```
line_count_result = lcel_task_chain("文本处理", {"operation": "count_lines", "content": text4})
```

```
print("示例4结果（多步串联）：\n文本分析->", analysis_result, "\n行数统计->", line_count_result)
```

LCEL构建任务链

```
# 示例5：条件分支（根据文本长度选择不同处理方式）
# 如果文本长度大于20，做文本分析，否则只统计行数
text5 = "短文本示例"
if len(text5) > 20:
    result5 = lcel_task_chain("文本分析", {"text": text5})
    print("示例5结果（条件分支）：文本较长，执行文本分析->", result5)
else:
    result5 = lcel_task_chain("文本处理", {"operation": "count_lines", "content": text5})
    print("示例5结果（条件分支）：文本较短，只统计行数->", result5)
```

Summary

Thinking: 1-simple_toolchain.py 和 2-simple_toolchain.py 的区别是什么?

1-simple_toolchain.py 是基于 LangChain 的 Agent 框架（如 create_react_agent、AgentExecutor 等），通过 Agent 结合工具链，自动解析用户输入，智能选择和调用合适的工具，支持多轮推理和工具调用。

适合需要"智能决策+多工具自动调度"的复杂任务。

2-simple_toolchain.py 是基于 LCEL（LangChain Expression Language）表达式链式组合。

核心思想是用 LCEL 的 RunnableLambda 等方式，将各个工具函数以"数据流"方式串联或分支，手动调度任务链。

适合需要"自定义流程、明确步骤、可控组合"的多工具任务。

任务链的每一步由开发者显式指定（如先文本分析再统计行数，或条件分支）。

不具备 Agent 的"自主决策"能力，所有流程和分支都需手动编排。

AI Agent对比

工具	核心定位	架构特点	适用场景
LangChain	开源LLM应用开发框架	基于链（Chain）的线性或分支工作流，支持Agent模式	快速构建RAG、对话系统、工具调用等线性任务
LangGraph	LangChain的扩展，专注于复杂工作流	基于图（Graph）的循环和条件逻辑，支持多Agent协作	需要循环、动态分支或状态管理的复杂任务（如自适应RAG、多Agent系统）
Qwen-Agent	通义千问的AI Agent框架	基于阿里云大模型，支持多模态交互与工具调用	开源，集成多种工具，MCP调用
Coze	字节跳动的无代码AI Bot平台	可视化拖拽界面，内置知识库、多模态插件	快速部署社交平台机器人、轻量级工作流
Dify	开源LLM应用开发平台	API优先，支持Prompt工程与灵活编排	开发者定制化LLM应用，需深度集成或私有化部署

AI Agent工具定位与对比

AI Agent对比

(1) workflow编排

LangChain: 线性链, 适合固定流程任务 (如文档问答)。

LangGraph: 支持循环、条件边和状态传递, 适合动态调整的复杂逻辑 (如多轮决策)。

Coze: 可视化工作流, 支持嵌套和批处理, 但灵活性较低。

Dify: 基于自然语言定义工作流, 适合API集成和Prompt调优。

(2) 工具调用与扩展性

LangChain/LangGraph: 工具作为链或图的节点, 支持自定义工具和重试逻辑。

Coze: 依赖预置插件生态, 扩展需通过开放平台。

Dify: 支持OpenAPI集成, 适合技术栈复杂的场景。

AI Agent对比

(3) RAG (检索增强生成)

LangChain: 开箱即用的文档加载、向量检索功能。

LangGraph: 需手动设计RAG节点, 但支持反馈循环优化检索质量。

Dify/Coze: Dify提供基础RAG支持, Coze依赖知识库管理。

(4) 多模态与部署

Coze: 支持图像、视频生成, 可直接发布至社交平台。

Qwen-Agent: 开源架构, 支持三级索引, 以及工具调用, MCP协议调用, 使用方便。

Dify: 专注私有化部署, 适合企业内网。

AI Agent对比

AI Agent选择建议：

- 无代码开发： Coze
- 快速原型开发： LangChain（线性任务）或 Qwen-Agent。
- 复杂Agent系统： LangGraph（多Agent协作）或 Dify（API深度集成）。
- 企业私有化： Dify（开源部署）,Qwen-Agent 或 LangChain+LangGraph（灵活组合）

CASE：搭建故障诊断Agent（Coze）

Coze workflow 适配性分析：

✅ 适合搭建的典型场景

标准化诊断流程（如OSI分层排查）

多工具串联调用（Ping→Traceroute→路由检查）

条件分支决策（根据前序结果动态调整路径）

团队知识沉淀（解决方案自动入库）

⚠️ 需注意的局限性

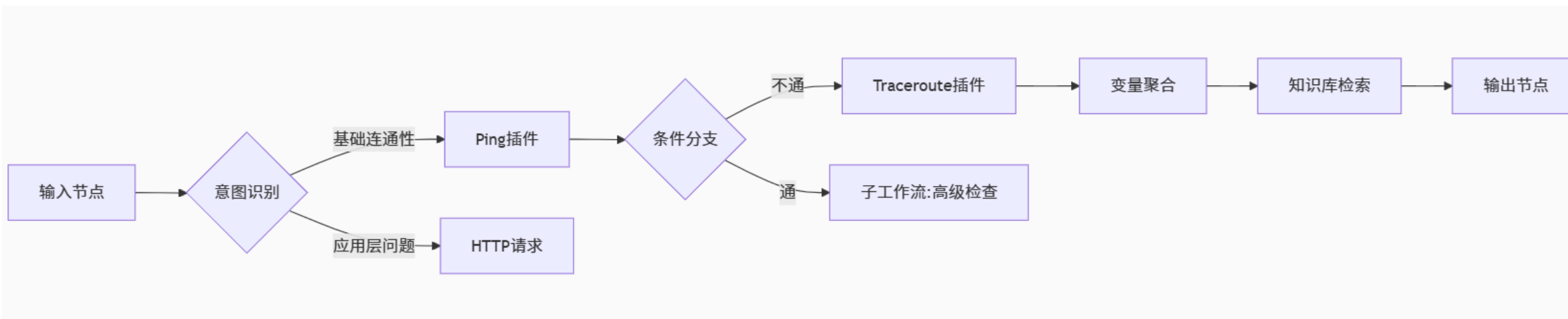
实时设备交互：无法直接SSH登录思科设备（需通过API中转）

复杂协议分析：如BGP状态机诊断需依赖外部系统

大规模拓扑处理：超过50节点的网络建议拆分子工作流

CASE：搭建故障诊断Agent（Coze）

Coze workflows 中的节点设计：









关键关系说明

数据流：上游节点的输出变量（如ping_result）作为下游节点的输入参数

控制流：条件分支节点决定后续执行路径（类似if-else）

异步调用：HTTP请求节点与其他节点可并行执行

状态共享：通过全局变量（如global.failed_devices）跨节点传递数据



Thank You
Using data to solve problems