AI Agent

1. Agent是什么

- Agent是一种基于大语言模型的智能体，它能够自主理解、规划决策并执行复杂的任务。
- 与传统的大模型助手不同，Agent不仅能提供指导，还会主动参与到任务的执行过程中，帮助解决实际操作中的问题。
- 这种智能体不仅仅是思考者，更是行动者，能够自主进行决策并采取行动。

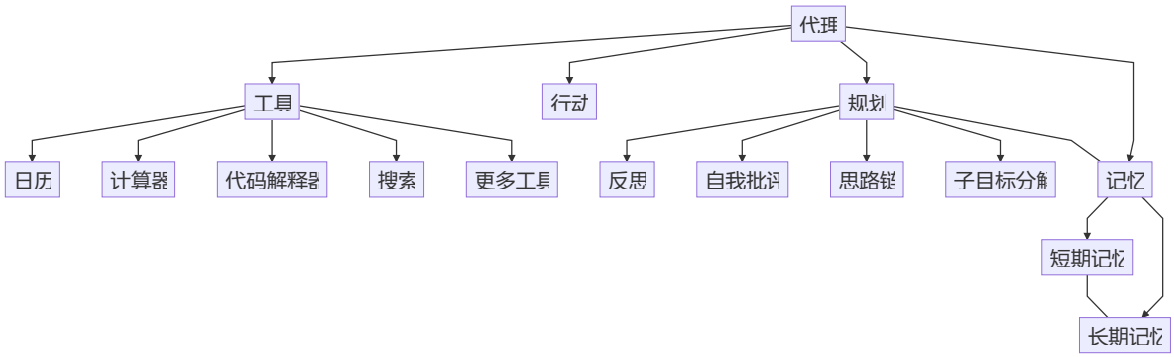
2. Agent的组成

- Agent有四部分组成，四个部分分别由Agent来进行调度
    - **Planning（规划）**：规划是 agent 做出决策、策略和下一步行动的部分。通过链式思考（chain of thought）或自我反思（reflection）等方式，agent 能够进行复杂的决策流程。规划是每次任务执行的起点，决定了行动的方向和执行步骤。
    - **Memory（记忆）**：记忆包括短时记忆和长时记忆，参与 agent 的决策过程。agent 会从过去的经验中汲取信息，就像人类在做决策时回忆以前遇

到过的情况，并将相关信息整合进当前决策中。记忆的参与可以提高决策的精确度和连贯性。

- **Action（行动）**：行动是指 agent 最终执行任务的步骤。规划之后，agent 会根据决策结果选择并调度适当的工具来完成特定任务。行动部分是通过调用工具库中的工具来实现目标。这类似于人在计划后采取实际行动的过程。
- **Tools（工具）**：工具是 agent 执行任务的手段。当规划和记忆得出决策后，agent 会从工具库中选择适当的工具来执行任务。这些工具可以是各种功能模块，如感知、计算或外部接口，帮助 agent 实现其行动目标。

- Agents = LLM + 规划 + 记忆 + 工具使用



> 示例

3. Langchain中的Agents的实现

在使用 LangChain 构建的智能体 (Agent) 中，其执行流程主要由需求处理、决策循环和工具调用组成。以下是智能体在执行任务时的一般步骤：

3.1 需求输入与组合 (User Request & Prompt Generation)

- **用户需求/问题**: 首先，用户提出需求或问题。
- **组合提示词**: LangChain 会将用户的问题与预设的 `prompt 模板`（提示词模板）进行组合。这一步生成了用于处理问题的初始提示信息，类似于给模型设定上下文。

3.2 增强式生成 (ReAct Loop)

- **调用大模型**: 在接收到组合后的提示词后，智能体会通过调用大语言模型 (LLM) 进行进一步的推理和生成。
- **调用相关上下文**: 在此过程中，智能体会参考多种资料来源，包括上下文信息、历史记录以及内存中的相关内容。这形成了一个循环过程，使得智能体可以根据现有数据进行多次思考和生成。

3.3 决策循环 (Decision Loop)

- **查询记忆 (Memory)**: 智能体首先会检查其记忆库，寻找是否有与当前问题相关的上下文信息或帮助解决问题的历史记录。
- **工具查询与使用 (Tool Usage)**: 智能体会调用已配置的工具库，寻找能够解决当前问题的工具。工具可以

是查询数据库、API 调用等，帮助智能体处理任务中的具体步骤。
- **问题拆解与多步决策**: 在这个过程中，智能体可能不会直接给出一个完整的答案。它会将问题分解成多个步骤，每一步进行独立决策。这与简单的对话式模型不同，智能体可以在执行任务时多次调用 AI 接口，不断优化其答案。

3.4 多轮推理与执行 (Iterative Reasoning & Execution)

- 在某些复杂任务中，智能体可能需要多次进行推理。在每一轮推理中，它会根据新的信息调整决策并调用相应的工具。
- **拆解决策**: 每一步决策都会被拆解为多个子步骤，每个步骤会进一步推进任务的完成。这种方式确保了智能体能够分阶段解决复杂问题，而不是一次性给出答案。

3.5 最终输出

- 在经过多轮推理和工具调用之后，智能体会得出最终的答案或结果并返回给用户。

4. Agent的功能实现

- Agent的实现Demo

```
from langchain_core.prompts import
PromptTemplate
from langchain_openai import ChatOpenAI
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_too
ls import load_tools
llm = ChatOpenAI(model="gpt-4o-mini")
# 工具加载函数:利用工具来增强模型: llm-math计算,
wikipedia
# llm-math => 是一个已经封装好的数据计算工具包
# wikipedia => 维基百科工具包
tools = load_tools(["llm-math",
"wikipedia"], llm=llm)

# 初始化一个智能体（agent)
# 参数一 => 之前加载的工具
# 参数二 => 语言模型
# 参数三 => 智能体的类型,不需要针对特定任务进行训练
# 参数四 => 智能体在执行任务时是否输出详细的日志信
息
# 参数五 => 智能体在处理输入时是否应该处理解析错误
agent = initialize_agent(tools=tools,
                         llm=llm,

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

```
handle_parsing_errors=True)

prompt_template = "明朝建立什么时候,皇帝是谁?"

prompt =
PromptTemplate.from_template(prompt_templat
e)

res = agent.invoke(prompt)
print(res)
```

5. Agents的内置类型

1. **OpenAI Functions Agent**
   - `OPENAI_FUNCTIONS` 类型主要用于与 OpenAI 的函数调用进行交互。
   - 其特点只支持OpenAI。

> 示例

```
from langchain_openai import ChatOpenAI
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_t
ools import load_tools

llm = ChatOpenAI(model="gpt-4o-mini")
```

```
tools = load_tools(["llm-math",
"wikipedia"], llm=llm)

agent = initialize_agent(tools=tools,
                         llm=llm,
                         # 特点只支持
OpenAI

agent=AgentType.OPENAI_FUNCTIONS,
                         verbose=True,

handle_parsing_errors=True)

res = agent.invoke("乌克兰总统现在多大了?")
print(res)
```

2. **Zero-shot ReAct Description Agent**
   - 零样本增强生成型
     - **Zero-shot**：零样本学习。指无需示例或训练，模型就能够执行任务。
     - **ReAct**：增强生成。它结合了零样本的能力来生成和处理任务。
   - 零样本增强式生成，即在没有示例的情况下可以自主的进行对话的类型

> 示例

```
from langchain_openai import OpenAI
```

```
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_t
ools import load_tools

llm = OpenAI(model="gpt-3.5-turbo-
instruct")

tools = load_tools(["llm-math",
"wikipedia"], llm=llm)

agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.ZERO_SHOT_REACT_DESCRIPTI
ON,
                         verbose=True,

handle_parsing_errors=True)

res = agent.invoke("乌克兰总统现在多大了?")
print(res)
```

3. **Zero-shot Chat ReAct Description Agent**
    - 和上述类似，但该类型使用 **chat model**（聊天模型）来生成内容。
    - 适用于对话模型相关的任务，如需要生成对话或互动的场景。

- `CHAT_ZERO_SHOT_REACT_DESCRIPTION` 相比较 `ZERO_SHOT_REACT_DESCRIPTION`=> 使用了 ChatModel

> 示例

```
from langchain_core.prompts import
PromptTemplate
from langchain_openai import ChatOpenAI
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_t
ools import load_tools

llm = ChatOpenAI(model="gpt-4o-mini")

tools = load_tools(["llm-math",
"wikipedia"], llm=llm)

agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.CHAT_ZERO_SHOT_REACT_DESC
RIPTION,
                         verbose=True,

handle_parsing_errors=True)
```

```
prompt_template = "乌克兰总统泽连斯基今年多大了?"

prompt = PromptTemplate.from_template(prompt_template)

res = agent.invoke(prompt)
print(res)
```

4. **Conversation ReAct Description Agent**
   - 该类型专门针对对话场景设计，能够在与用户交互时进行更智能的增强生成。
   - 需要添加记忆模块

> 示例

```
from langchain_openai import OpenAI
from langchain.agents import initialize_agent, AgentType
from langchain_community.agent_toolkits.load_tools import load_tools


# 初始化 LLM 和工具
llm = OpenAI(model="gpt-3.5-turbo-instruct")
tools = load_tools(["llm-math", "wikipedia"], llm=llm)
```

```
# 记忆组件
from langchain.memory import
ConversationBufferMemory

memory = ConversationBufferMemory(
    memory_key="chat_history"
)

# 初始化 agent
agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.CONVERSATIONAL_REACT_DESC
RIPTION,
                         verbose=True,

handle_parsing_errors=True,
                         memory=memory)

# 执行调用
agent.invoke({"input": "我叫做先知,我喜欢吃
果冻?"})

agent.invoke({"input": "我姓什么?"})

agent.invoke({"input": "我喜欢吃什么?"})
```

5. **Structured Chat Zero-shot ReAct Description Agent**

- 结构化对话生成增强型 `agent`。
- 和普通的 Zero-shot Chat 类似，但它的特点在于返回的结果是结构化数据（如 JSON、XML），适合进一步的数据处理或集成工作流。

> 示例

```
from langchain_openai import ChatOpenAI
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_t
ools import load_tools

# 初始化 LLM 和工具
llm = ChatOpenAI(model="gpt-4o-mini")
tools = load_tools(["llm-math",
"wikipedia"], llm=llm)

# 记忆组件
from langchain.memory import
ConversationBufferMemory
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True  # 确保返回的是消
息对象，而不是纯字符串
)

# 初始化 agent
agent = initialize_agent(tools=tools,
```

```
                                llm=llm,

agent=AgentType.CHAT_CONVERSATIONAL_REACT
_DESCRIPTION,
                                verbose=True,

handle_parsing_errors=True,
                                memory=memory)

# 执行调用
agent.invoke({"input": "我叫做先知,我喜欢吃
果冻?"})

agent.invoke({"input": "我姓什么?"})

agent.invoke({"input": "我喜欢吃什么?"})
```

- Langchain里面所有的Agent类型都封装在**AgentType**当中

6. 如何解决Agent中memory丢失

6.1 memory记忆丢失

- 添加记忆

> 示例

```
from langchain_openai import ChatOpenAI
```

```
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_too
ls import load_tools
from langchain.memory import
ConversationBufferMemory
# 初始化 LLM 和工具
llm = ChatOpenAI(model="gpt-4o-mini")
tools = load_tools(["llm-math",
"wikipedia"], llm=llm)

# 记忆组件
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True  # 确保返回的是消息对
象，而不是纯字符串
)

# 初始化 agent
agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.OPENAI_FUNCTIONS,
                         verbose=True,

handle_parsing_errors=True,
                         memory=memory)

# 执行调用
```

```
agent.invoke({"input": "我叫做先知,我喜欢吃果冻?"})
agent.invoke({"input": "我姓什么?"})
agent.invoke({"input": "我喜欢吃什么?"})

# 输出prompt => 输出结果当中,并没有chat_history记忆模块
print(agent.agent.prompt)
print("-------------")
print(agent.agent.prompt.input_variables)
```

- 我们希望的就是将memory记忆插入到提示词当中去,这样才能够记住东西,但是在输出层prompt当中,并没有看到chat_history,但是,此时memory记忆其实是处于丢失的状态

6.2 使用代理参数Agent解决记忆丢失

- 代理参数`agent_kwargs`是在创建或配置 LangChain Agent 时使用的参数集合。
- 通过传递`agent_kwargs`，您可以控制 Agent 的各种设置，从而更好地适应特定的任务需求。
- 示例

```
from langchain_openai import ChatOpenAI
from langchain.agents import
initialize_agent, AgentType
```

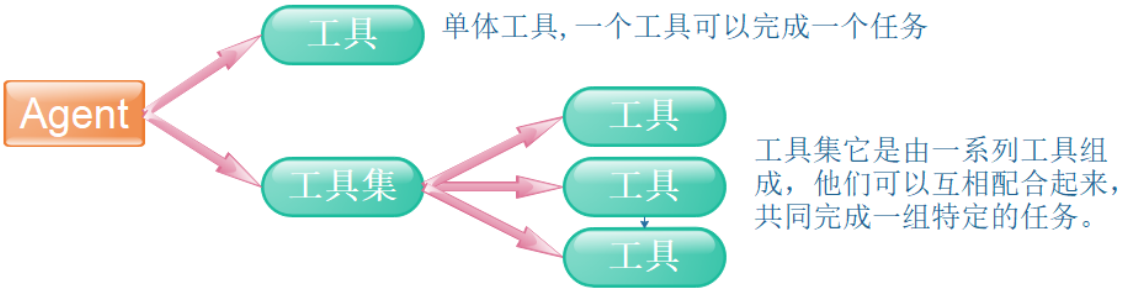
```
from
langchain_community.agent_toolkits.load_too
ls import load_tools
from langchain.memory import
ConversationBufferMemory
from langchain.prompts import
 MessagesPlaceholder
# 初始化 LLM 和工具
llm = ChatOpenAI(model="gpt-4o-mini")
tools = load_tools(["llm-math",
"wikipedia"], llm=llm)

# 记忆组件
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True  # 确保返回的是消息对
象，而不是纯字符串
)

# 我们希望的就是将memory记忆插入到提示词当中去,这
样才能够记住东西,但是在输出层prompt当中,并没有看到
chat_history,此时
# memory记忆其实是并没有起到作用的
# 初始化 agent
agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.OPENAI_FUNCTIONS,
                         verbose=True,
```

```
handle_parsing_errors=True,
                                memory=memory,
                                agent_kwargs={ # 代理参数 => 所以我们agent_kwargs传递参数，把 memory key传入到提示词

 "extra_prompt_messages":[ # 添加额外的提示消息

 MessagesPlaceholder(variable_name="chat_history"), # 将历史消息作为占位符引入提示

 MessagesPlaceholder(variable_name="agent_scratchpad") # 消息列表，可以包含多种类型的消息对象，包括 AI 消息、人类消息、聊天消息、系统消息、函数消息和工具消息
                                    ],
                                })

# 执行调用
agent.invoke({"input": "我叫做先知,我喜欢吃果冻?"})
agent.invoke({"input": "我姓什么?"})
agent.invoke({"input": "我喜欢吃什么?"})

# 输出prompt
print(agent.agent.prompt)
print("--------------")
print(agent.agent.prompt.input_variables)
```

7. Tools的使用

7.1 Agent当中Tools的分类

- 单体工具 => 一个工具就可以完成一个任务
- 工具集 => 工具集它是由一系列工具组成，他们可以互相配合起来，共同完成一组特定的任务。



7.2 工具的使用

- langchain预制了大量的tools，基本这些工具能满足大部分需求
- tools

> 示例

```
from langchain.agents import load_tools
tool_names = [...]
tools = load_tools(tool_names) #使用load方法
```

- Dall-E
  - 是openai出品的文到图AI大模型

> 示例

```
from langchain_core.prompts import
PromptTemplate
from langchain_openai import ChatOpenAI
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_too
ls import load_tools
llm = ChatOpenAI(model="gpt-4o-mini")

# 加载Dall-E => 文生图
tools = load_tools(["dalle-image-
generator"],llm=llm)

agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION
,
                         verbose=True,

handle_parsing_errors=True)

prompt_template = "鸡吃米"

prompt =
PromptTemplate.from_template(prompt_templat
e)

res = agent.invoke(prompt)
```

```
print(res)
```

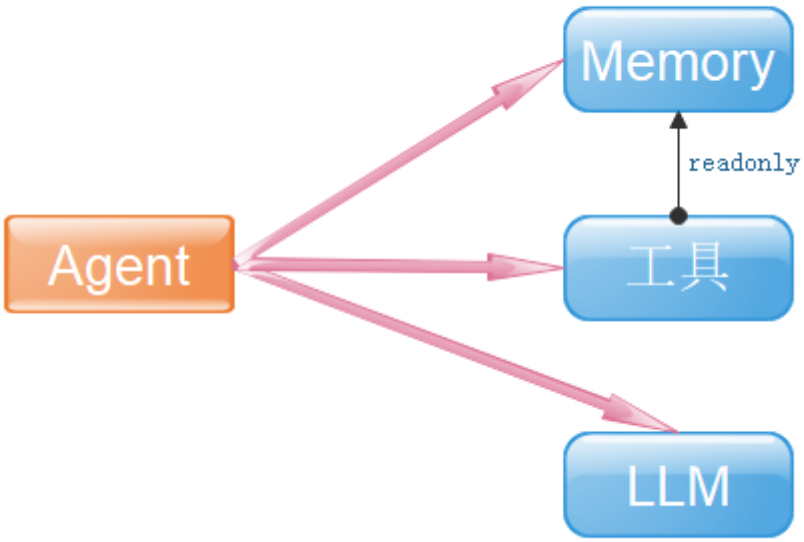
- GraphQL
  - GraphQL 一种api查询语言，类似sql，查找一下和星球大战相关的电影
  - API地址 https://swapi-graphql.netlify.app/.netlify/functions/index

> 示例

```
from langchain_openai import ChatOpenAI
from langchain.agents import
initialize_agent, AgentType
from
langchain_community.agent_toolkits.load_too
ls import load_tools

llm = ChatOpenAI(model="gpt-4o-mini")

tools =
load_tools(["graphql"],graphql_endpoint="ht
tps://swapi-
graphql.netlify.app/.netlify/functions/inde
x")

agent = initialize_agent(tools=tools,
                         llm=llm,

agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION
,
```

```
                                verbose=True,

handle_parsing_errors=True)

graphql_fields = """allFilms {
    films {
      title
      director
      releaseDate
      speciesConnection {
        species {
          name
          classification
          homeworld {
            name
          }
        }
      }
    }
  }

"""

suffix = "Search for the titles of all the
stawars films stored in the graphql
database that has this schema,and answer in
chinese:"

agent.invoke(suffix + graphql_fields)
```

8. 如何让Agent与tool共享记忆



- 当工具需要使用Memory的时候，LangChain 提供了一种称为只读（read-only）的Memory的类型，允许工具直接读取这Memory。

> 示例

```
from langchain.agents import
initialize_agent, Tool, AgentType
from langchain.memory import
ConversationBufferMemory,
ReadOnlySharedMemory
from langchain.chains import LLMChain
from langchain.prompts import
PromptTemplate,MessagesPlaceholder
```

```
from langchain_community.utilities import
SerpAPIWrapper
from langchain_openai import OpenAI

llm = OpenAI(
    temperature = 0,
    model="gpt-3.5-turbo-instruct",
)

template = """以下是一段人工智能与人类的对话:
{chat_history}
根据输入和上面的对话记录写一份对话总结.
输入: {input}"""

prompt = PromptTemplate(
    input_variables=
["input","chat_history"],
    template=template,
)

memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True,
)

readonlymemory =
ReadOnlySharedMemory(memory=memory)

summary_chain = LLMChain(
    llm=llm,
```

```
    prompt=prompt,
    verbose=True,
    memory=readonlymemory
)


import os
os.environ["SERPAPI_API_KEY"] =
"cfe4242ca57fac7a016555c8ea000e9a58def10692
e08f7f8c5cddf6b79d5ae6"
#搜索工具
search = SerpAPIWrapper()
#总结工具
def SummaryChainFun(history):
    print("\n==============总结链开始运行
==============")
    print("输入历史: ",history)
    summary_chain.invoke(history)

tools = [
    Tool(
        name="Search",
        func=search.run,
        description="当需要了解实时的信息或者你
不知道的事时候可以使用搜索工具",
    ),
    Tool(
        name="Summary",
        func=SummaryChainFun,
```

```
        description="当你被要求总结一段对话的时候可以使用这个工具，工具输入必须为字符串，只在必要时使用",
    ),
]

memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True,
)

# agent_chain = initialize_agent(
#     tools,
#     llm,
#
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION
,
#     verbose=True,
#     handle_parsing_errors=True,
#     memory=memory,
# )

prefix = """Have a conversation with a
human, answering the following questions as
best you can. You have access to the
following tools:"""
suffix = """Begin!"
{chat_history}
Question: {input}
{agent_scratchpad}"""
```

```
agent_chain = initialize_agent(
    tools,
    llm,

 agent=AgentType.ZERO_SHOT_REACT_DESCRIPTIO
N,
    verbose=True,
    handle_parsing_errors=True,
    agent_kwargs={
        "prefix":prefix,
        "suffix":suffix,

 "agent_scratchpad":MessagesPlaceholder("ag
ent_scratchpad"),

 "chat_history":MessagesPlaceholder("chat_h
istory"),

 "input":MessagesPlaceholder("input"),
    },
    memory=memory,
)

print(agent_chain.agent.llm_chain.prompt.te
mplate)
agent_chain.run(input="迈克尔乔丹谁?中文回复")
print("----------")
agent_chain.run(input="他的儿子叫什么名字?")
```