# 函数调用快速提取结构化数据使用技巧

# 函数调用快速提取结构化数据使用技巧

## 01. 结构化输出策略与选择

让 LLM 返回结构化输出非常重要，这是因为 LLM 应用程序的输出通常用于下游程序，这些应用程序需要特定的参数，目前常见的几种让 LLM 结构化输出的策略有：

1. `Prompt`：通过 prompt 让 LLM 输出特定结构的内容，兼容所有 LLM，但是输出不稳定。
2. `函数/工具调用`：让 LLM 绑定函数，并设置 `选择模式` 为强制，让 LLM 强制调用函数，从而获取结构化输出数据。
3. `JSON模式`：对于支持 JSON 模式输出的 LLM，还可以通过设置输出结构为 `JSON模式`，从而获取结构化数据。

其中后两种输出模式会更稳定一些，在 LangChain 中为后两种方法封装了 `.with_structured_output()` 方法，这也是获取结构化输出的最简单和最可靠的方法，在 `.with_structured_output()` 的底层会使用 LLM 原生的 `函数调用` 或 `JSON模式`。

该方法接受一个 `BaseModel子类` 作为输入，该**子类需要指定输出属性的名称、描述和类型**，该方法返回的是一个 Runnable 可运行对象，但并不是输出字符串或者 AI 消息，而是**输出与给定模式对应的对象，其中模式可以指定为 JSON 模式（返回一个字典）或 Pydantic 类（返回一个 Pydantic 对象）**。

另外检测一个 LLM 是否支持 `.with_structured_output()` 可以通过查看源码或者在 LangChain 的大语言模型 `高级功能` 列表中查看，链接：https://imooc-langchain.shortvar.com/docs/integrations/chat/

例如在前面的课时中，我们学习的 `DocTran` 在底层就是使用 `函数调用` 来实现结构化输出，如果使用这节课所学习的知识来完成一个 `QA问答数据` 的提取，代码示例如下：

```
import dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


class QAExtra(BaseModel):
    """一个问答键值对工具，传递对应的假设性问题+答案"""
    question: str = Field(description="假设性问题")
    answer: str = Field(description="假设性问题对应的答案")


llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
structured_llm = llm.with_structured_output(QAExtra)

prompt = ChatPromptTemplate.from_messages([
    ("system", "请从用户传递的query中提取出假设性的问题+答案。"),
    ("human", "{query}")
])

chain = {"query": RunnablePassthrough()} | prompt | structured_llm
```

```
print(chain.invoke("我叫慕小课，我喜欢打篮球，游泳"))
```

输出内容：

```
question='你喜欢打篮球吗？' answer='是的，我喜欢打篮球。'
```

在 `.with_structured_output()` 的底层会优先使用 `函数调用` 模式，如果 LLM 支持 `JSON模式`，还可以在函数内传递多一个参数 `method="json_mode"`，即使用大语言模型的 `JSON输出模式` 来操作，修改代码如下：

```
import dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


class QAExtra(BaseModel):
    """一个问答键值对工具，传递对应的假设性问题+答案"""
    question: str = Field(description="假设性问题")
    answer: str = Field(description="假设性问题对应的答案")


llm = ChatOpenAI(model="gpt-4o")
structured_llm = llm.with_structured_output(QAExtra, method="json_mode")

prompt = ChatPromptTemplate.from_messages([
    ("system", "请从用户传递的query中提取出假设性的问题+答案。响应格式为JSON，并携带`question`和`answer`两个字段。"),
    ("human", "{query}")
])

chain = {"query": RunnablePassthrough()} | prompt | structured_llm

print(chain.invoke("我叫慕小课，我喜欢打篮球，游泳。"))
```

输出内容：

```
question='你叫什么名字？你喜欢做什么？' answer='我叫慕小课，我喜欢打篮球，游泳。'
```

在上面的代码中修改了几个部分：

1. `gpt-4o模型`：在 OpenAI 提供的模型中，并不是所有模型都支持 JSON 模式的，需要查看文档确认。
2. `prompt提示模板`：在 JSON 模式中，需要在 Prompt 中告知 LLM 输出 JSON 的结构信息，要不然会报错。

不过因为支持 JSON 模式的大语言模型较少，所以使用起来限制也比较多，还会对原始的 Prompt 产生干扰，所以尽可能使用 `函数调用` 的形式来结构化数据，会更稳定。

> **Important**

> 在创建 BaseModel 子类时，类的名称、文档字符串以及参数的名称和提供的描述也非常重要，因为在该函数的底层大部分使用的都是函数调用，只有添加上这些信息后，才可以将对应的 LLM 绑定函数调用，获取正确的结果。

## 02. .with_structured_output() 源码解析

在 `.with_structured_output()` 底层，会通过传递的 `method` 来执行不同的 `运行时参数绑定`，例如使用 `函数调用` 亦或者是 `JSON模式` 输出，核心代码：

```
def with_structured_output(
    self,
    schema: Optional[_DictOrPydanticClass] = None,
    *,
    method: Literal["function_calling", "json_mode"] = "function_calling",
    include_raw: bool = False,
    **kwargs: Any,
) -> Runnable[LanguageModelInput, _DictOrPydantic]:
    if kwargs:
        raise ValueError(f"Received unsupported arguments {kwargs}")
    is_pydantic_schema = _is_pydantic_class(schema)
    if method == "function_calling":
        if schema is None:
            raise ValueError(
                "schema must be specified when method is 'function_calling'. "
                "Received None."
            )
        llm = self.bind_tools([schema], tool_choice="any")
        if is_pydantic_schema:
            output_parser: OutputParserLike = PydanticToolsParser(
                tools=[schema], first_tool_only=True
            )
        else:
            key_name = convert_to_openai_tool(schema)["function"]["name"]
            output_parser = JsonOutputKeyToolsParser(
                key_name=key_name, first_tool_only=True
            )
    elif method == "json_mode":
        llm = self.bind(response_format={"type": "json_object"})
        output_parser = (
            PydanticOutputParser(pydantic_object=schema)
            if is_pydantic_schema
            else JsonOutputParser()
        )
    else:
        raise ValueError(
            f"Unrecognized method argument. Expected one of 'function_calling' or
"
            f"'json_mode'. Received: '{method}'"
        )

    if include_raw:
        parser_assign = RunnablePassthrough.assign(
            parsed=itemgetter("raw") | output_parser, parsing_error=lambda _:
None
        )
```

```
        parser_none = RunnablePassthrough.assign(parsed=lambda _: None)
        parser_with_fallback = parser_assign.with_fallbacks(
            [parser_none], exception_key="parsing_error"
        )
        return RunnableMap(raw=llm) | parser_with_fallback
    else:
        return llm | output_parser
```

所以哪怕不使用 LangChain，该思路也非常值得借鉴，在需要使用 LLM 获取结构化数据时，只需要构建一个 `虚假的函数`，让 LLM 绑定该函数，并且设置 `tool_choice="any"` 即强制调用所有函数，这样就可以在最大程度上确保 LLM 能稳定地输出结构化数据。

# 函数调用出错捕获提升程序健壮性

# 函数调用出错处理提升程序健壮性

在执行函数调用的过程中，错误几乎是不可避免的，无论是大语言模型错误地传递了参数，还是工具内部抛出的错误，如果错误不做任何处理操作，都会让程序变得异常脆弱，所可以我们可以考虑在链中构建错误处理/捕获来减轻这些故障出现的概率。

## 01. try/except 捕获工具错误

首先我们先来构建一个`复杂工具`，并让 LLM 尝试调用这个工具，并故意引发一些错误，示例代码如下：

```
import dotenv
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


@tool
def complex_tool(int_arg: int, float_arg: float, dict_arg: dict) -> int:
    """使用复杂工具进行复杂计算操作"""
    return int_arg * float_arg


# 1.创建大语言模型并绑定工具
llm = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
llm_with_tools = llm.bind_tools([complex_tool])

# 2.创建链并执行工具
chain = llm_with_tools | (lambda msg: msg.tool_calls[0]["args"]) | complex_tool

# 3.调用链
print(chain.invoke("使用复杂工具，对应参数为5和2.1"))
```

输出内容如下：

```
Traceback (most recent call last):
  File "D:\Project\llmops\llmops-api\study\41-工具调用\1.错误捕获.py", line 29, in
<module>
    print(chain.invoke("使用复杂工具，对应参数为5和2.1"))
  File "D:\Project\llmops\llmops-api\venv\lib\site-
packages\langchain_core\runnables\base.py", line 2875, in invoke
    input = step.invoke(input, config)
  File "D:\Project\llmops\llmops-api\venv\lib\site-
packages\langchain_core\tools.py", line 427, in invoke
    return self.run(tool_input, **kwargs)
  File "D:\Project\llmops\llmops-api\venv\lib\site-
packages\langchain_core\tools.py", line 615, in run
    raise error_to_raise
  File "D:\Project\llmops\llmops-api\venv\lib\site-
packages\langchain_core\tools.py", line 578, in run
    tool_args, tool_kwargs = self._to_args_and_kwargs(tool_input)
  File "D:\Project\llmops\llmops-api\venv\lib\site-
packages\langchain_core\tools.py", line 501, in _to_args_and_kwargs
```

```
    tool_input = self._parse_input(tool_input)
  File "D:\Project\llmops\llmops-api\venv\lib\site-
packages\langchain_core\tools.py", line 454, in _parse_input
    result = input_args.parse_obj(tool_input)
  File "D:\Project\llmops\llmops-api\venv\lib\site-packages\pydantic\v1\main.py",
line 526, in parse_obj
    return cls(**obj)
  File "D:\Project\llmops\llmops-api\venv\lib\site-packages\pydantic\v1\main.py",
line 341, in __init__
    raise validation_error
pydantic.v1.error_wrappers.ValidationError: 1 validation error for
complex_toolSchema
dict_arg
  field required (type=value_error.missing)
```

在上面的示例中，因为工具的描述并不清晰，大语言模型只针对前 2 个参数生成了对应的数值，第 3 个参数并没有生成，所以引发了 `pydantic` 数据校验错误，对于这类错误，通用的处理方法是在工具调用步骤上使用 `try/except` 进行错误的捕获，并在出错时返回游泳的提示信息，修正代码部分如下：

```
def try_except_tool(tool_args: dict, config: RunnableConfig) -> Any:
    try:
        return complex_tool.invoke(tool_args, config)
    except Exception as e:
        return f"调用工具时使用以下参数:\n\n{tool_args}\n\n引发了以下错
误:\n\n{type(e)}: {e}"

chain = llm_with_tools | (lambda msg: msg.tool_calls[0]["args"]) |
try_except_tool
```

输出内容：

```
调用工具时使用以下参数:

{'int_arg': 5, 'float_arg': 2.1}

引发了以下错误:

<class 'pydantic.v1.error_wrappers.ValidationError'>: 1 validation error for
complex_toolSchema
dict_arg
  field required (type=value_error.missing)
```

在这种模式下，如果将 `工具返回的错误消息` 重新返回给 LLM 时，LLM 会重新判断并继续执行 `函数调用`，从而将参数补全，让程序正常执行，也是最推荐的一种错误处理方案，即在 `工具内部` 进行错误的捕获，并将错误信息独立返回。

## 02. 回退与重试处理

在前面的课时中，学习 Runnable 可运行组件时，我们学习了如果 Runnable 可运行组件出错，可以执行 `回退` 和 `重试` 两种策略，在函数调用中也可以使用这两种策略来处理，例如：

1. 在函数调用参数生成错误时，可以考虑回退到一个更好的模型，例如平时使用 `gpt-3.5-turbo-16k` 模型，在出错时，回退到 `gpt-4o` 模型上进行重新试验。

2. 亦或者是触发重试机制，并且在重试的时候，携带上错误信息，让 LLM 强大的自然语言处理功能进行自我纠正。

使用更好的模型进行回退，操作起来非常简单，只需要定义一个更强大的模型，并创建一条新链，然后使用 `Runnable可运行组件` 的 `.with_fallbacks()` 即可绑定对应的回退策略，示例代码如下：

```
import dotenv
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


@tool
def complex_tool(int_arg: int, float_arg: float, dict_arg: dict) -> int:
    """使用复杂工具进行复杂计算操作"""
    return int_arg * float_arg


# 1.创建大语言模型并绑定工具
llm = ChatOpenAI(model="gpt-3.5-turbo-16k").bind_tools([complex_tool])
better_llm = ChatOpenAI(model="gpt-4o").bind_tools(
    [complex_tool], tool_choice="complex_tool",
)

# 2.创建链并执行工具
better_chain = better_llm | (lambda msg: msg.tool_calls[0]["args"]) |
complex_tool
chain = (llm | (lambda msg: msg.tool_calls[0]["args"]) |
complex_tool).with_fallbacks([better_chain])

# 3.调用链
print(chain.invoke("使用复杂工具，对应参数为5和2.1"))
```

`回退策略` 并不一定是万能的，有的时候哪怕使用了参数更大的模型，生成的 `函数调用` 参数依旧不符合规范，这种情况就要考虑下工具的相关描述、Prompt 编写得是否存在问题。

另外一种优化策略是 `携带错误信息的重试策略`，让 LLM 纠正其行为，只需要在抛出错误时，将错误信息一起携带给 LLM，让其重新操作即可，示例代码如下：

```
from typing import Any

import dotenv
from langchain_core.messages import ToolCall, AIMessage, ToolMessage,
HumanMessage
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableConfig
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


class CustomToolException(Exception):
    """自定义的工具错误异常"""
```

```
    def __init__(self, tool_call: ToolCall, exception: Exception) -> None:
        super().__init__()
        self.tool_call = tool_call
        self.exception = exception


@tool
def complex_tool(int_arg: int, float_arg: float, dict_arg: dict) -> int:
    """使用复杂工具进行复杂计算操作"""
    return int_arg * float_arg


def tool_custom_exception(msg: AIMessage, config: RunnableConfig) -> Any:
    try:
        return complex_tool.invoke(msg.tool_calls[0]["args"], config)
    except Exception as e:
        raise CustomToolException(msg.tool_calls[0], e)


def exception_to_messages(inputs: dict) -> dict:
    # 1.从输入中提取错误信息
    exception = inputs.pop("exception")
    # 2.将历史消息添加到原始输入中，以便模型直到它在上一次工具调用中犯了一个错误
    messages = [
        AIMessage(content="", tool_calls=[exception.tool_call]),
        ToolMessage(tool_call_id=exception.tool_call["id"],
content=str(exception.exception)),
        HumanMessage(content="最后一次工具调用引发了异常，请尝试使用更正的参数再次调用该工
具，不要重复犯错。")
    ]
    inputs["last_output"] = messages
    return inputs


# 1.创建prompt，并预留占位符，用于存储错误输出信息
prompt = ChatPromptTemplate.from_messages([
    ("human", "{query}"),
    ("placeholder", "{last_output}"),
])

# 2.创建大语言模型并绑定工具
llm = ChatOpenAI(model="gpt-4o", temperature=0).bind_tools(tools=[complex_tool])

# 3.创建链并执行工具
chain = prompt | llm | tool_custom_exception
self_correcting_chain = chain.with_fallbacks(
    [exception_to_messages | chain], exception_key="exception",
)

# 4.调用自我纠正链完成任务
print(self_correcting_chain.invoke({"query": "使用复杂工具，对应参数为5和2.1"}))
```

输出内容：

```
10.5
```

# 多模态 LLM 执行函数调用的技巧

# 多模态LLM执行函数调用的技巧

目前市面上的 LLM 除了能接收文本数据，还出现了不少多模态 LLM，这些 LLM 不仅能接收数据，还能接收 `图像`、`音频` 等内容，并且这类 `多模态LLM` 也支持 `函数调用` 功能，要想使用这些模型调用工具，只需要按照正常的方式将工具绑定到模型上，在传递 `消息` 给大语言模型时，按照 `多模态LLM` 的特定规则即可。

## 01. GPT-4o 多模态输入

OpenAI 提供的 GPT-4o 模型是一个多模态的模型（输入多模态），在传递 `消息列表` 时，可以在 `人类消息` 中添加 `图片地址`，这样即可将特定的图片上传给 GPT-4o 模型进行识别，从而实现多模态输入。

- GPT-4o 多模态输入文档链接：https://platform.openai.com/docs/api-reference/chat/create

官方提供原生示例如下：

```
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
    model="gpt-4o",
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": "What's in this image?"},
                {
                    "type": "image_url",
                    "image_url": {
                        "url":
"https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisconsin-madison-the-nature-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-nature-boardwalk.jpg"
                    },
                },
            ],
        }
    ],
    max_tokens=300,
)

print(response.choices[0])
```

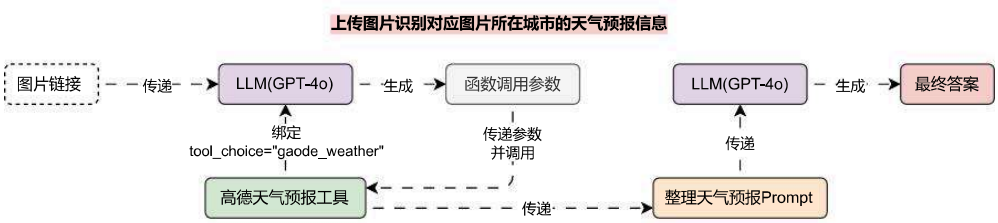
在 LangChain 中，如果消息也是多模态的，只需要按照特定的模型对应的 `消息结构` 历史创建 `LangChain消息` 即可，例如在上面的示例中，`人类消息` 传递了一个列表，包含了 `文本` 和 `图片链接`，转换成 `LangChain消息/提示模板` 如下：

```
prompt = ChatPromptTemplate.from_messages([
    ("human", [
        {"type": "text", "text": "{query}"},
        {"type": "image_url", "image_url": {"url": "{image_url}"}}
    ])
])

print(prompt.invoke({
    "query": "这张图片所在的地址的天气怎么样",
    "image_url":
"https://img1.baidu.com/it/u=644490943,1781886584&fm=253&fmt=auto&app=138&f=JPEG"
}).to_messages())
```

使用技巧和普通的提示模板没有差异，只是将原本的 `字符串` 替换成了 `列表字典` 的格式，输出内容：

```
[HumanMessage(content=[{'type': 'text', 'text': '这张图片所在的地址的天气怎么样'},
{'type': 'image_url', 'image_url': {'url':
'https://img1.baidu.com/it/u=644490943,1781886584&fm=253&fmt=auto&app=138&f=JPEG'
}}])]
```

这里我们以 `GPT-4O` + `天气预报查询工具` + `多模态输入` 三个组件构建一个 LLM 应用，让该 LLM 应用能识别上传图片城市所在的天气信息，运行流程如下：




示例代码如下：

```
import json
import os
from typing import Type, Any

import dotenv
import requests
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import Field, BaseModel
from langchain_core.runnables import RunnablePassthrough
from langchain_core.tools import BaseTool
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


class GaodeWeatherArgsSchema(BaseModel):
    city: str = Field(description="需要查询天气预报的目标城市，例如：广州")


class GaodeWeatherTool(BaseTool):
    """根据传入的城市名查询天气"""
```

```
    name = "gaode_weather"
    description = "当你想询问天气或与天气相关的问题时的工具。"
    args_schema: Type[BaseModel] = GaodeWeatherArgsSchema

    def _run(self, *args: Any, **kwargs: Any) -> str:
        """运行工具获取对应城市的天气预报"""
        try:
            # 1.获取高德API秘钥，如果没有则抛出错误
            gaode_api_key = os.getenv("GAODE_API_KEY")
            if not gaode_api_key:
                return f"高德开放平台API秘钥未配置"

            # 2.提取传递的城市名字并查询行政编码
            city = kwargs.get("city", "")
            session = requests.session()
            api_domain = "https://restapi.amap.com/v3"
            city_response = session.request(
                method="GET",
                url=f"{api_domain}/config/district?keywords=
{city}&subdistrict=0&extensions=all&key={gaode_api_key}",
                headers={"Content-Type": "application/json; charset=utf-8"},
            )
            city_response.raise_for_status()
            city_data = city_response.json()

            # 3.提取行政编码调用天气预报查询接口
            if city_data.get("info") == "OK":
                if len(city_data.get("districts")) > 0:
                    ad_code = city_data["districts"][0]["adcode"]

                    weather_response = session.request(
                        method="GET",
                        url=f"{api_domain}/weather/weatherInfo?city=
{ad_code}&extensions=all&key={gaode_api_key}&output=json",
                        headers={"Content-Type": "application/json; charset=utf-
8"},
                    )
                    weather_response.raise_for_status()
                    weather_data = weather_response.json()
                    if weather_data.get("info") == "OK":
                        return json.dumps(weather_data)

            session.close()
            return f"获取{kwargs.get('city')}天气预报信息失败"
            # 4.整合天气预报信息并返回
        except Exception as e:
            return f"获取{kwargs.get('city')}天气预报信息失败"


# 1.构建prompt
prompt = ChatPromptTemplate.from_messages([
    ("human", [
        {"type": "text", "text": "请获取下上传图片对应城市的天气信息。"},
        {"type": "image_url", "image_url": {"url": "{image_url}"}}
    ]),
])
```

```
weather_prompt = ChatPromptTemplate.from_template("""请整理下传递的城市的天气预报信息，
并以用户友好的方式输出。

<weather>
{weather}
</weather>""")

# 2.构建LLM并绑定工具
llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools(tools=[GaodeWeatherTool()],
tool_choice="gaode_weather")

# 3.创建链应用并执行
chain = (
        {
            "weather": (
                    {"image_url": RunnablePassthrough()}
                    | prompt
                    | llm_with_tools
                    | (lambda msg: msg.tool_calls[0]["args"])
                    | GaodeWeatherTool()
            )
        } | weather_prompt | llm | StrOutputParser()
)

print(chain.invoke("https://imooc-langchain.shortvar.com/guangzhou.jpg"))
```

输出内容：

```
以下是广州未来几天的天气预报信息：

### 广州市天气预报

**发布时间：2024年7月11日 12:00**

#### 2024年7月11日（星期四）
- **白天天气：** 大雨
- **夜间天气：** 大雨
- **白天温度：** 33°C
- **夜间温度：** 25°C
- **白天风向：** 西南
- **夜间风向：** 西南
- **风力等级：** 1-3级

#### 2024年7月12日（星期五）
- **白天天气：** 大雨
- **夜间天气：** 中雨-大雨
- **白天温度：** 33°C
- **夜间温度：** 24°C
- **白天风向：** 北
- **夜间风向：** 北
- **风力等级：** 1-3级

#### 2024年7月13日（星期六）
- **白天天气：** 中雨-大雨
```

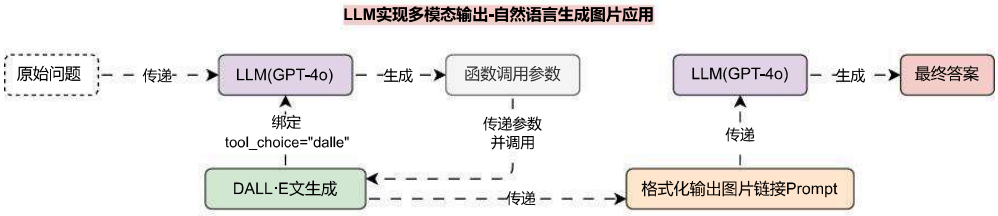
```
- **夜间天气：** 中雨-大雨
- **白天温度：** 32°C
- **夜间温度：** 25°C
- **白天风向：** 西南
- **夜间风向：** 西南
- **风力等级：** 1-3级

#### 2024年7月14日（星期日）
- **白天天气：** 中雨-大雨
- **夜间天气：** 大雨
- **白天温度：** 33°C
- **夜间温度：** 25°C
- **白天风向：** 北
- **夜间风向：** 北
- **风力等级：** 1-3级

请注意天气变化，做好防雨准备。
```

## 02. GPT-4o 接入 DALL·E 文生图

除了输入的多模态，大语言模型的输出其实也可以通过曲线救国的方式实现 `多模态输出`，有使用过 ChatGPT-Plus 的小伙伴肯定了解，当我们和 ChatGPT 对话时，可以让 ChatGPT 帮忙生成对应的图片，该功能底层本质上也是通过 `函数调用` 来实现的，运行原理非常简单：

1. 创建一个根据 `文本` 生成 `图片` 工具，例如可以使用 DALLE，工具的调用参数为生成图片的 Prompt。
2. 将工具绑定到 LLM 上，并预设特定的 Prompt，让 LLM 将用户输入的 query 转换成绘图的 Prompt。
3. 调用工具，将 Prompt 生成图片，并将 `工具消息`、`AI消息` 叠加到历史消息中，再次提问，获得最终答复。

运行流程和刚刚我们创建的 `图片获取天气预报应用` 非常接近，如下：




实现代码如下：

```
import dotenv
from langchain_community.tools.openai_dalle_image_generation import
OpenAIDALLEImageGenerationTool
from langchain_community.utilities.dalle_image_generator import DallEAPIWrapper
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()

dalle = OpenAIDALLEImageGenerationTool(
    name="openai_dalle",
    api_wrapper=DallEAPIWrapper(model="dall-e-3")
)
```

```
llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools(tools=[dalle], tool_choice="openai_dalle")

chain = llm_with_tools | (lambda msg: msg.tool_calls[0]["args"]) | dalle

print(chain.invoke("帮我绘制一幅老爷爷的图片"))
```

输出内容：

```
https://oaidalleapiprodscus.blob.core.windows.net/private/org-8mmokaDgFXOdOsE9HC1PNBZM/user-XbaFwYqigWuDrIeZS6l61HgI/img-NxKXd4xBMJZdIixcxYQe5dmd.png?st=2024-08-15T05%3A02%3A53Z&se=2024-08-15T07%3A02%3A53Z&sp=r&sv=2024-08-04&sr=b&rscd=inline&rsct=image/png&skoid=d505667d-d6c1-4a0a-bac7-5c84a87759f8&sktid=a48cca56-e6da-484e-a814-9c849652bcb3&skt=2024-08-14T17%3A52%3A40Z&ske=2024-08-15T17%3A52%3A40Z&sks=b&skv=2024-08-04&sig=WHJO%2BKw9OKRSKOaYLXL%2Btataexl5Vf9lJAGVxvmO9kM%3D
```

生成的图片示例：

# 基于 ReACT 架构的 Agent 智能体设计与实现

# 基于ReACT架构的Agent智能体设计与实现

## 01. Agent 概念与介绍

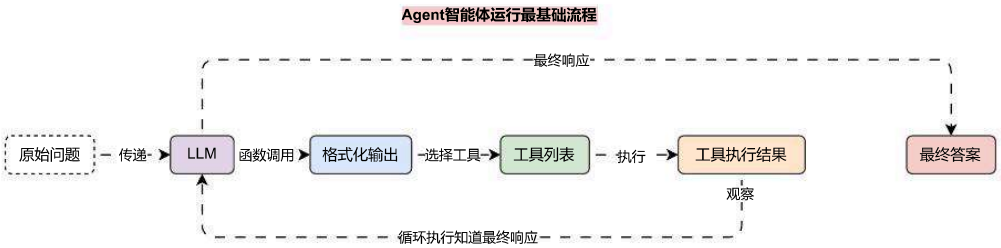
在 LLM 应用中，如果我们知道用户输入所需的工具使用特定顺序时，使用 LCEL 表达式构建链应用非常有用，但是对于某一些特例，我们使用工具的次数与顺序取决与输入，在这种情况下，我们希望让 LLM 本身决定使用工具的次数和顺序，而 `Agent智能体` 能做到这一点。

在 LangChain 中，Agent 是一个核心概念，它代表了一种能够利用语言模型（LLM）和其他工具来执行复杂任务的系统，Agent 设计的目的是为了处理那些语言模型可能无法直接解决的问题，尤其是当这些任务涉及到多个步骤或者需要外部数据源的情况。

无论一个 Agent 设计得多么复杂，使用什么架构，最基础的工作流程其实都非常简单，只有 5 个步骤：

1. 输入理解：Agent 首先解析用户输入，理解其意图和需求。
2. 计划定制：基于对输入的理解，Agent 会定制一个执行计划，决定使用哪些工具和执行的顺序。
3. 工具调用：Agent 按照计划调用相应的工具，执行必要的操作。
4. 结果整合：收集所有工具返回的结果，进行整合和解析，形成最终的输出。
5. 反馈循环：如果任务没有完成或者需要进一步的消息，Agent 可以迭代上述过程直到满足条件为止。

运行流程图如下：




对比前面我们学习的函数调用，其实 `Agent` 的运行流程非常接近，多了一步 `执行工具` 和 `观察结果` 的步骤，甚至在前面的课时中，我们其实已经实现了该流程，只是没有将代码封装到一起而已，所以，对于一个 Agent 来说，其组成模块包括 3 个部分：

1. `Tools`：Agent可以访问的工具集，每个工具通常执行一个特定的功能。
2. `Executor`：执行 Agent计划的逻辑。
3. `Prompt Templates`：指导 Agent 如何理解和处理输入的模板，可以定制化以适应不同任务。

在 LangChain v0.2.0 版本中，有两种实现 Agent 的技巧，一种使用的是 `传统Agent组件`，一种使用 `LangGraph`，`传统Agent组件` 特别适合入门的开发者，所以在这一章中我们会使用该方式，下一章在考虑使用 LangGraph 创建更加复杂、灵活性和控制性更强的 Agent 应用。

针对 `传统Agent组件`，LangChain 团队封装了共计 8 种 Agent，不同的 Agent 适用于不同的聊天模型，表格如下：

| Agent类型 | 模型类型 | 支持聊天历史 | 支持多工具输入 | 支持并行工具调用 | 必要模型参数 | 使用场景 |
| --- | --- | --- | --- | --- | --- | --- |
| 工具调用 | Chat | ✅ | ✅ | ✅ | `tools` | 使用支持工具调用的模型 |
| OpenAI工具 | Chat | ✅ | ✅ | ✅ | `tools` | [遗留]使用最新的OpenAI模型 |
| OpenAI函数 | Chat | ✅ | ✅ | | `functions` | [旧版]使用OpenAI模型，或者已针对函数调用进行微调并公开与OpenAI相同参数的模型 |
| XML | LLM | ✅ | | | | 使用Anthropic或者其他擅长XML的模型 |
| 结构化聊天 | Chat | ✅ | ✅ | | | 需要传递多个工具时 |
| JSON聊天 | Chat | ✅ | | | | 使用擅长JSON的模型 |
| ReACT | LLM | ✅ | | | | 使用的是简单模型 |
| 带搜索的自我问答 | LLM | | | | | 使用的是简单模型并且只有一个搜索工具 |

- Agent 类型文档链接：https://python.langchain.com/v0.1/docs/modules/agents/agent_types/

## 02. ReACT 智能体运行流程与实现

ReACT 是 LangChain 最早支持的 Agent 架构，ReACT = `Reason` + `Action`，即 `推理与行动`，目前绝大部分 Agent 架构都是在 ReACT 架构上进行衍生的，该架构最早是 Yao 等人在 2022 年发布的论文中首次被提出，论文参考地址：https://arxiv.org/pdf/2210.03629。

在 LangChain 中，要想创建基于 ReACT 架构的智能体，其实也非常简单，导入 `AgentExecutor`、`create_react_agent`，在实例化的时候，传递对应的 `工具` + `prompt` 即可，其中 ReACT 架构的智能体 prompt 是有要求的。

- ReACT prompt 文档：https://smith.langchain.com/hub/hwchase17/react

例如实现一个带有 `谷歌实时搜索` 的 ReACT 架构的 `Agent`，其完整示例代码如下：

```
import dotenv
from langchain import hub
from langchain.agents import create_react_agent, AgentExecutor
from langchain_community.tools import GoogleSerperRun
from langchain_community.utilities import GoogleSerperAPIWrapper
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.tools import render_text_description_and_args
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()
```

```
class GoogleSerperArgsSchema(BaseModel):
    query: str = Field(description="执行谷歌搜索的查询语句")


# 1.定义工具与工具列表
google_serper = GoogleSerperRun(
    name="google_serper",
    description=(
        "一个低成本的谷歌搜索API。"
        "当你需要回答有关时事的问题时，可以调用该工具。"
        "该工具的输入是搜索查询语句。"
    ),
    args_schema=GoogleSerperArgsSchema,
    api_wrapper=GoogleSerperAPIWrapper(),
)
tools = [google_serper]

# 2.定义智能体提示模板
prompt = hub.pull("hwchase17/react")

# 3.创建大语言模型
llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
agent = create_react_agent(
    llm=llm,
    tools=tools,
    prompt=prompt,
    tools_renderer=render_text_description_and_args,
)

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

print(agent_executor.invoke({"input": "马拉松的最新世界记录是多少?"}))
```

当提问 `马拉松的最新世界记录是多少？` 时，该智能体的输出内容如下：

```
> Entering new AgentExecutor chain...
I should use the google_serper tool to search for the latest world record in
marathon.
Action: google_serper
Action Input: {'query': 'latest world record in marathon'}Kenyan athlete Kelvin
Kiptum set a men's world record time of 2:00:35 on October 8, 2023, at the 2023
Chicago Marathon. Ethiopian athlete Tigst Assefa broke the ... Missing: query |
Show results with:query. Here's a complete list of the marathon world-record
holders and all time top 26.2-mile runners. Missing: query | Show results
with:query. In the 2023 Chicago marathon Kelvin Kiptum from Kenya, set a new
marathon world record with a time of two hours and 35 seconds, taking 34 seconds
off the ... Missing: query | Show results with:query. Exactly four years ago,
Kipchoge made history with his 1:59:40 unofficial time in Vienna, the fastest any
runner had ever covered the marathon. Missing: query | Show results with:query.
Kenyan Kelvin Kiptum broke the marathon world record to win the Chicago Marathon
in 2 hours and 35 seconds, nearly breaking the two-hour ... Kelvin Kiptum set a
new world record as he ran the Chicago marathon in 2hr 0min 35sec. Subscribe to
Guardian Sport ▻ http://bit.ly/GDNSport ... Missing: query | Show results
with:query. The current marathon world records are held by Kelvin Kiptum
(2:00:35) and Tigist Assefa (2:11:53). Here are the top 20 fastest marathons of
all ... Missing: query | Show results with:query. CHICAGO (AP) — Kelvin Kiptum
set a world record in the Chicago Marathon on Sunday, finishing in 2 hours, 35
seconds to shatter fellow Kenyan ...The latest world record in marathon is
2:00:35, set by Kenyan athlete Kelvin Kiptum at the 2023 Chicago Marathon.
Final Answer: The latest world record in marathon is 2:00:35.

> Finished chain.
{'input': '马拉松的最新世界记录是多少?', 'output': 'The latest world record in
marathon is 2:00:35.'}
```

## 03. ReACT 智能体的缺陷

在 ReACT 架构中，通过对 LLM 的输出内容进行信息提取用于执行不同的逻辑，如果向 `ReACT智能体` 提问 `马拉松的世界记录是多少？`，它底层 `推理-行动` 的完整 Prompt 如下：

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

google_serper - 一个低成本的谷歌搜索API。当你需要回答有关时事的问题时，可以调用该工具。该工具
的输入是搜索查询语句。, args: {'query': {'title': 'Query', 'description': '执行谷歌搜索
的查询语句', 'type': 'string'}}

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 [google_serper]
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

Begin!
```

```
Question: 马拉松的最新世界记录是多少?
Thought:I should use the google_serper tool to search for the latest world record
in marathon.
Action: google_serper
Action Input: {'query': 'latest world record in marathon'}
Observation: Kenyan athlete Kelvin Kiptum set a men's world record time of
2:00:35 on October 8, 2023, at the 2023 Chicago Marathon. Ethiopian athlete Tigst
Assefa broke the ... Missing: query | Show results with:query. Here's a complete
list of the marathon world-record holders and all time top 26.2-mile runners.
Missing: query | Show results with:query. In the 2023 Chicago marathon Kelvin
Kiptum from Kenya, set a new marathon world record with a time of two hours and
35 seconds, taking 34 seconds off the ... Missing: query | Show results
with:query. Exactly four years ago, Kipchoge made history with his 1:59:40
unofficial time in Vienna, the fastest any runner had ever covered the marathon.
Missing: query | Show results with:query. Kenyan Kelvin Kiptum broke the marathon
world record to win the Chicago Marathon in 2 hours and 35 seconds, nearly
breaking the two-hour ... Kelvin Kiptum set a new world record as he ran the
Chicago marathon in 2hr 0min 35sec. Subscribe to Guardian Sport ▻
http://bit.ly/GDNSport ... Missing: query | Show results with:query. The current
marathon world records are held by Kelvin Kiptum (2:00:35) and Tigist Assefa
(2:11:53). Here are the top 20 fastest marathons of all ... Missing: query | Show
results with:query. CHICAGO (AP) — Kelvin Kiptum set a world record in the
Chicago Marathon on Sunday, finishing in 2 hours, 35 seconds to shatter fellow
Kenyan ...
Thought: The latest world record in marathon is 2:00:35, set by Kenyan athlete
Kelvin Kiptum at the 2023 Chicago Marathon.
Final Answer: The latest world record in marathon is 2:00:35.
```

可以完整的观察到 `Question(原始问题)`、`Thought(观察)`、`Action(执行动作)`、`Action input(动作输入)`、`Observation(观察工具输出)`、`Thought(观察)`、`Final Answer(最终答案)` 一系列完整的运行流程，输出格式非常规范，所以程序不会抛出错误。

但是我们都知道 LLM 的输出是不稳定的，假设我们提问 `你好，你是？` 时，按推理来说，这段原始提问是不需要调用工具，所以可以直接输出答案，对于 ReACT 来说，需要大语言模型输出如下格式的数据：

```
I now know the final answer
Final Answer: 我是一个人工智能助手，旨在帮助回答问题和提供信息。请问有什么我可以帮忙的吗？
```

和上一次生成内容组装成完整的对话语句如下：

```
Question: 你好，你是?
Thought: I now know the final answer
Final Answer: 我是一个人工智能助手，旨在帮助回答问题和提供信息。请问有什么我可以帮忙的吗？
```

这样 `ReACT智能体` 底层检测到 `Final Answer` 这个关键词，就可以提取出最终答案，但是 LLM 的输出是及其不稳定的，在实际测试中，哪怕是 GPT-4o 模型，它会输出如下的数据：

```
我是一个人工智能助手，旨在帮助回答问题和提供信息。请问有什么我可以帮忙的吗？
```

乍得一看好像很合理，但是 `ReACT智能体` 是根据不同的输出结构来提取出后续要执行的步骤是什么，如果这个时候组装上之前的提问，就变成了：

```
Question: 你好，你是?
Thought: 我是一个人工智能助手，旨在帮助回答问题和提供信息。请问有什么我可以帮忙的吗？
```

`ReACT智能体` 检测到只有 `Thought` 即观察，并没有后续的步骤了，肯定抛出错误，因为在 `Thought` 后一般会携带 `Action` 或者 `Final Answer`，程序识别不了后续的步骤是什么，结果肯定出错。

另外 `ReACT智能体` 在使用时，如果需要修改 Prompt 为中文，一般还需要同步修改 `输出解析器`，使用代价非常大。

这是因为 `ReACT智能体` 底层是通过解析响应内容中 `特定关键词` 来提醒后续步骤的，如果 `Prompt` 改成中文，不调整 `输出解析器`，在代码底层还是会提取 `Thought` 的数据，程序 100% 会抛出错误。

# 基于工具调用的智能体设计与实现

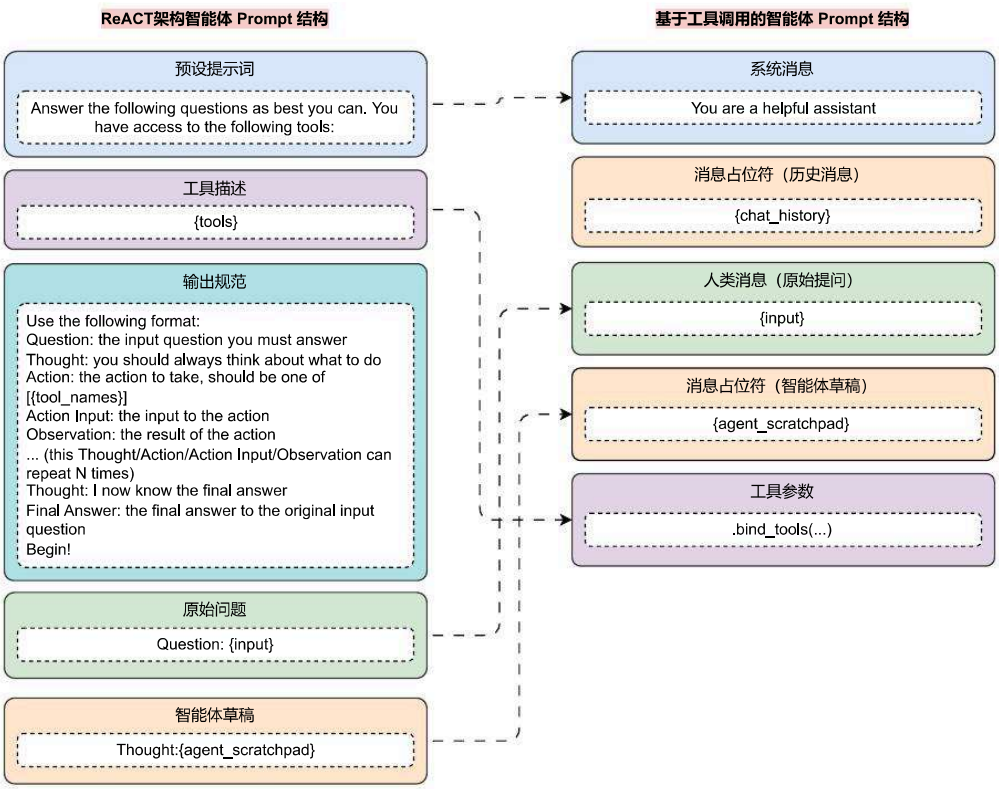
# 基于工具调用的智能体设计与实现

## 01. 工具调用智能体

基于 ReACT 架构的智能体会将 `tools(工具描述)`、`agent_scratchpad(智能体草稿)`、`工具结果`、`推理` 等内容全部放到同一个 `prompt` 中，并通过提取 `LLM的规范输出` 来决定下一步的操作，这种模式会随着 LLM 输出的随机性，不同 LLM 性能的差异让程序变得异常脆弱。

而且 ReACT 架构早期设计之初是针对 `LLM(文本补全模型)` 进行设计的，即传入一段话，让 LLM 补全其后续的文本，随着 LLM 的发展，消息设计更友好、结构化输出更稳定的函数调用、性能更强大的 ChatModel 发布了，可以考虑将 ReACT 迁移到基于 `聊天消息` + `工具调用` 的架构上，思想不变，但是使用更稳定的 `消息列表` + `工具调用`。

迁移流程图如下：




在上述的 `工具调用智能体Prompt` 中，`输出规范` 会通过检测 `LLM` 是输出 `文本内容` 还是 `工具调用参数` 来判断下一步是什么，这样性能更加稳定，而且对于绝大部分 `LLM` 来说，`工具调用` 支持一次性调用生成多个工具的参数，性能会更强。

在 LangChain 中，其实也为 `基于工具调用的Agent` 封装了一个快速创建的方法 `create_tool_calling_agent()` 和 `预设Prompt`。

- LangChain hub 工具调用 Prompt 链接：https://smith.langchain.com/hub/hwchase17/openai-tools-agent
- 基于工具调用的智能体文档：https://python.langchain.com/v0.1/docs/modules/agents/agent_types/tool_calling/

## 02. 实现示例

在 LangChain 中，要实现 `工具调用Agent` 其实也非常简单，步骤其实和 `ReACT-Agent` 一模一样，创建好工具列表、Prompt、LLM(支持工具调用)，然后使用 `create_tool_calling_agent()` 创建智能体，接下来创建智能体执行者完成包装即可。

例如实现一个可以根据用户输入实现自主选择 `联网搜索` + `文生图` 的智能体，示例代码如下：

```
import dotenv
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain_community.tools import GoogleSerperRun
from langchain_community.tools.openai_dalle_image_generation import
OpenAIDALLEImageGenerationTool
from langchain_community.utilities import GoogleSerperAPIWrapper
from langchain_community.utilities.dalle_image_generator import DallEAPIWrapper
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()


class GoogleSerperArgsSchema(BaseModel):
    query: str = Field(description="执行谷歌搜索的查询语句")


class DallEArgsSchema(BaseModel):
    query: str = Field(description="输入应该是生成图像的文本提示(prompt)")


# 1.定义工具与工具列表
google_serper = GoogleSerperRun(
    name="google_serper",
    description=(
        "一个低成本的谷歌搜索API。"
        "当你需要回答有关时事的问题时，可以调用该工具。"
        "该工具的输入是搜索查询语句。"
    ),
    args_schema=GoogleSerperArgsSchema,
    api_wrapper=GoogleSerperAPIWrapper(),
)
dalle = OpenAIDALLEImageGenerationTool(
    name="openai_dalle",
    api_wrapper=DallEAPIWrapper(model="dall-e-3"),
    args_schema=DallEArgsSchema,
)
tools = [google_serper, dalle]

# 2.定义工具调用agent提示词模板
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant"),
    ("placeholder", "{chat_history}"),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}"),
])
```

```python
# 2.创建大语言模型
llm = ChatOpenAI(model="gpt-4o-mini")

# 3.创建agent与agent执行者
agent = create_tool_calling_agent(
    prompt=prompt,
    llm=llm,
    tools=tools,
)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

print(agent_executor.invoke({"input": "请帮我绘制一张老爷爷爬山的图片"}))
```

输出内容:

```
> Entering new AgentExecutor chain...

Invoking: `openai_dalle` with `{'query': 'An elderly man climbing a mountain,
with a determined expression, wearing typical hiking gear. The background shows a
scenic mountain landscape with trees and a clear sky. The scene conveys a sense
of adventure and resilience.'}`


https://dalleproduse.blob.core.windows.net/private/images/6fa42821-1331-4b18-
90dc-12364b6fb390/generated_00.png?se=2024-08-
19T13%3A32%3A20Z&sig=vro%2B%2FrpJbsEbrMxeQnV8rbkE5GKU5tTqlCEvc908V2E%3D&ske=2024-
08-23T22%3A55%3A33Z&skoid=09ba021e-c417-441c-b203-c81e5dcd7b7f&sks=b&skt=2024-08-
16T22%3A55%3A33Z&sktid=33e01921-4d64-4f8c-a055-5bdaffd5e33d&skv=2020-10-
02&sp=r&spr=https&sr=b&sv=2020-10-02这是我为您绘制的老爷爷爬山的图片:

![老爷爷爬山](https://dalleproduse.blob.core.windows.net/private/images/6fa42821-
1331-4b18-90dc-12364b6fb390/generated_00.png?se=2024-08-
19T13%3A32%3A20Z&sig=vro%2B%2FrpJbsEbrMxeQnV8rbkE5GKU5tTqlCEvc908V2E%3D&ske=2024-
08-23T22%3A55%3A33Z&skoid=09ba021e-c417-441c-b203-c81e5dcd7b7f&sks=b&skt=2024-08-
16T22%3A55%3A33Z&sktid=33e01921-4d64-4f8c-a055-5bdaffd5e33d&skv=2020-10-
02&sp=r&spr=https&sr=b&sv=2020-10-02)

希望您喜欢这幅作品!

> Finished chain.
{'input': '请帮我绘制一张老爷爷爬山的图片', 'output': '这是我为您绘制的老爷爷爬山的图片:
\n\n![老爷爷爬山]
(https://dalleproduse.blob.core.windows.net/private/images/6fa42821-1331-4b18-
90dc-12364b6fb390/generated_00.png?se=2024-08-
19T13%3A32%3A20Z&sig=vro%2B%2FrpJbsEbrMxeQnV8rbkE5GKU5tTqlCEvc908V2E%3D&ske=2024-
08-23T22%3A55%3A33Z&skoid=09ba021e-c417-441c-b203-c81e5dcd7b7f&sks=b&skt=2024-08-
16T22%3A55%3A33Z&sktid=33e01921-4d64-4f8c-a055-5bdaffd5e33d&skv=2020-10-
02&sp=r&spr=https&sr=b&sv=2020-10-02)\n\n希望您喜欢这幅作品! '}
```

修改成提问`马拉松的世界记录是多少?`，该智能体的回复如下:

```
> Entering new AgentExecutor chain...

Invoking: `google_serper` with `{'query': '马拉松世界纪录 2023'}`


2023年4月的伦敦马拉松赛，他以2小时1分25秒夺冠，这个成绩仅比当时的世界纪录慢了16秒。 2023年10月的芝加哥马拉松赛，以2小时00分35秒打破世界纪录。 仅有的三次马拉松经历，各个都是世界前六的好成绩。截至2023年10月，马拉松的世界纪录是由选手在芝加哥马拉松上创造的，时间为2小时00分35秒。

> Finished chain.
{'input': '马拉松的世界记录是多少？', 'output': '截至2023年10月，马拉松的世界纪录是由选手在芝加哥马拉松上创造的，时间为2小时00分35秒。'}
```

# 内置的其他 Agent 类型介绍与上手

# 内置的其他 Agent 类型介绍与上手

## 01. 内置的其他 Agent 介绍

在 LangChain v 0.2.0 版本之前封装了大量基于 `传统Agent组件` 的 Agent 智能体创建方法，这些组件的设计思路其实都是以 `推理-行动-观察` 为思想，不同类型的 Agent 会进行一些额外的扩展，例如 `记忆`、`外挂知识库`、`多角色`、`反思` 等，甚至是 `多Agent协作` 等。

而 `LangChain` 中封装的 `Agent` 也是基于 `推理-行动-观察` 思想，并为不同类型的 LLM/ChatModel 设计了不同的 `运行流程` 与 `prompt`，例如有些大语言模型擅长解读和回复 XML 类型的数据（Authropic），而有些模型擅长解读和回复 JSON 数据，而有些模型又擅长结构化输出，所以对于不同类型的模型，可以使用的 `Agent创建方法` 来创建不同类型的 Agent。

- LangChain 不同类型 Agent 文档：https://python.langchain.com/v0.1/docs/modules/agents/agent_types/xml_agent/

以 XMLAgent 为例，创建和使用的技巧也非常简单，只需修改 `prompt` 与创建 `Agent` 的方法即可，其他的无需任何调整：

```
# 创建XMLAgent提示模板
prompt = ChatPromptTemplate.from_messages([
    ("human", """You are a helpful assistant. Help the user answer any questions.

You have access to the following tools:

{tools}

In order to use a tool, you can use <tool></tool> and <tool_input></tool_input>
tags. You will then get back a response in the form <observation></observation>
For example, if you have a tool called 'search' that could run a google search,
in order to search for the weather in SF you would respond:

<tool>search</tool><tool_input>weather in SF</tool_input>
<observation>64 degrees</observation>

When you are done, respond with a final answer between <final_answer>
</final_answer>. For example:

<final_answer>The weather in SF is 64 degrees</final_answer>

Begin!

Previous Conversation:
{chat_history}

Question: {input}
{agent_scratchpad}"""),
])

# 创建大语言模型
llm = ChatOpenAI(model="gpt-4o-mini")

# 创建agent与agent执行者
```

```
agent = create_xml_agent(
    prompt=prompt,
    llm=llm,
    tools=tools,
)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

print(agent_executor.invoke({"input": "马拉松的世界记录是多少？", "chat_history":
""}))
```

输出内容：

```
> Entering new AgentExecutor chain...
<tool>google_serper</tool><tool_input>马拉松 世界记录 2023今年4月23日的伦敦马拉松，基普图姆以2小时01分25秒夺冠，大幅打破基普乔格保持的赛道纪录，与世界纪录只差16秒。 这一表现，让他被评为2023年男子场外赛事世界年度最佳运动员。<final_answer>截至2023年，马拉松的世界纪录是2小时01分09秒，由埃利乌德·基普乔格于2018年创造。</final_answer>

> Finished chain.
{'input': '马拉松的世界记录是多少？', 'chat_history': '', 'output': '截至2023年，马拉松的世界纪录是2小时01分09秒，由埃利乌德·基普乔格于2018年创造。'}
```

切换到 `JSONAgent` 同样只需要更改 `prompt` 与 `create_xml_agent()` 方法即可，修正的 prompt 如下：

```
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
  ("system", """Assistant is a large language model trained by OpenAI.

Assistant is designed to be able to assist with a wide range of tasks, from answering simple questions to providing in-depth explanations and discussions on a wide range of topics. As a language model, Assistant is able to generate human-like text based on the input it receives, allowing it to engage in natural-sounding conversations and provide responses that are coherent and relevant to the topic at hand.

Assistant is constantly learning and improving, and its capabilities are constantly evolving. It is able to process and understand large amounts of text, and can use this knowledge to provide accurate and informative responses to a wide range of questions. Additionally, Assistant is able to generate its own text based on the input it receives, allowing it to engage in discussions and provide explanations and descriptions on a wide range of topics.

Overall, Assistant is a powerful system that can help with a wide range of tasks and provide valuable insights and information on a wide range of topics. Whether you need help with a specific question or just want to have a conversation about a particular topic, Assistant is here to assist."""),
  ("placeholder", "{chat_history}"),
  ("human", """TOOLS
------
Assistant can ask the user to use tools to look up information that may be helpful in answering the users original question. The tools the human can use are:
```

```
{tools}

RESPONSE FORMAT INSTRUCTIONS
----------------------------

When responding to me, please output a response in one of two formats:

**Option 1:**
Use this if you want the human to use a tool.
Markdown code snippet formatted in the following schema:

```json
{{
    "action": string, \ The action to take. Must be one of {tool_names}
    "action_input": string \ The input to the action
}}
```

**Option #2:**
Use this if you want to respond directly to the human. Markdown code snippet
formatted in the following schema:

```json
{{
    "action": "Final Answer",
    "action_input": string \ You should put what you want to return to use here
}}
```

USER'S INPUT
--------------------
Here is the user's input (remember to respond with a markdown code snippet of a
json blob with a single action, and NOTHING else):

{input}"""),
  ("placeholder", "{agent_scratchpad}"),
])
```

其他类型的 Agent 也是一模一样的操作，修改 `prompt` 并切换到不同的 `Agent创建方法` 即可。

## 02. 内置 Agent 的异同点

通过前几节课学习的 `ReACTAgent`、`工具调用Agent`、`XMLAgent` 示例演示，其实可以很容易发现这些 Agent 的异同点，首先是相同点：

1. 所有 `Agent` 都拥有 `input`、`agent_scratchpad` 两个输入变量，表示原始问题和智能体草稿。
2. 所有 `Agent` 都是单 `Agent` 自我执行，无法与其他 Agent 进行相互协作。
3. 所有 `Agent` 都可以通过切换 `prompt` 与 `create_xxx_agent()` 方法快速切换 Agent 而无需修改大量代码。
4. 所有 `Agent` 设计思想都是基于 `推理-行动-观察`，只是不同的 Prompt 有所差异。
5. 所有 `Agent` 都是使用同一个 `LLM` 进行 `推理` 与 `答案生成`，并不支持 `多LLM` 分工。

6. 除了 `基于工具调用的Agent`，其他类型的智能体要修改 Prompt 适配特定语言一般都需要同步修改 `输出解析器`。

有差异的地方也非常明显：

1. 不同 `Agent` 的提示词风格有所差异，有的 `Agent` 会将 `tools` 也填写到 `prompt` 中，有的使用文本提示，有的使用消息提示。
2. 不同 `Agent` 的 `输出解析器` 不一致，绝大部分取决于 `prompt` 的差异，有的支持 `多工具`，有的不支持。
3. 不同 `Agent` 的 `输入编码方式` 不一致，绝大部分取决于 `prompt` 和 `LLM` 的差异，有的支持 `历史记忆输入`，有的不支持。

课后练习：

> **Important**
>
> 请根据 LangChain 官方文档提供的内置工具，实现一个集成 `加减乘除(多个自定义工具)`、`天气预报`、`谷歌实时搜索` 的 Agent 智能体，并提问 `2024年巴黎奥运会金牌榜前3名的国家总金牌数、平均金牌数分别是多少？`，并检测下 `ReACT智能体`、`工具调用智能体`、`XML智能体` 在该问题下的表现，思考其存在的缺陷。

# AgentExecutor 源码解析与 Agent 组件缺陷

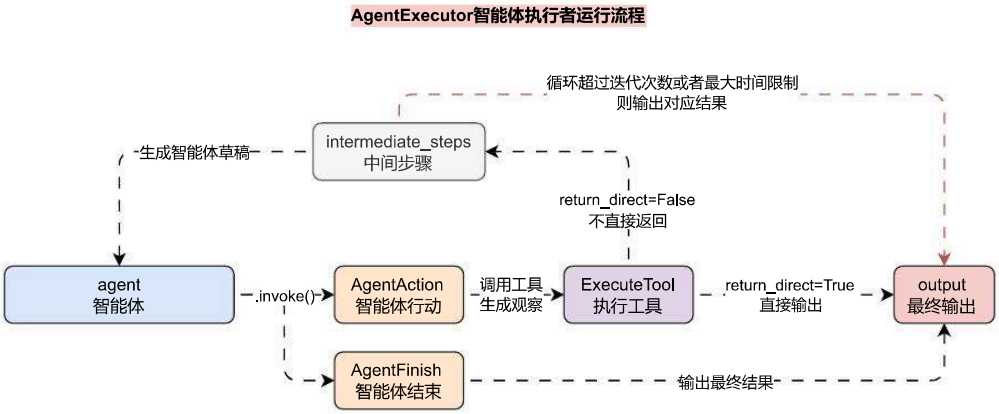
# AgentExecutor源码解析与Agent组件缺陷

## 01. AgentExecutor 源码解析

在 LangChain 中，无论是什么类型的 Agent（内置封装），都必须通过 `AgentExecutor` 来创建执行者才可以运行具有 `循环` + `工具执行` 的智能体，在 `智能体执行者` 的底层，实际操作是调用 Agent 智能体，执行它选择的操作/工具，将操作输出传递回 Agent，然后重复，伪代码如下：

```
next_action = agent.get_action(...)
while next_action != AgentFinish:
    observation = run(next_action)
    next_action = agent.get_action(..., next_action, observation)
return next_action
```

简易后的运行流程如下：




其核心代码是 `AgentExecutor` 中的 `_call()` 函数，核心代码如下：

```
def _call(
    self,
    inputs: Dict[str, str],
    run_manager: Optional[CallbackManagerForChainRun] = None,
) -> Dict[str, Any]:
    # 将工具列表组装成字典便于查询
    name_to_tool_map = {tool.name: tool for tool in self.tools}
    color_mapping = get_color_mapping(
        [tool.name for tool in self.tools], excluded_colors=["green", "red"]
    )
    # 定义变量存储中间步骤，类型为元组列表，第1个值为智能体行动，第2个值为行动/工具结果(观察)
    intermediate_steps: List[Tuple[AgentAction, str]] = []
    # 定义变量记录迭代次数+开始时间
    iterations = 0
    time_elapsed = 0.0
    start_time = time.time()
    # 执行循环知道不能遍历
    while self._should_continue(iterations, time_elapsed):
        # 获取下一步的的输出，结构为元组列表，和中间步骤接近
        next_step_output = self._take_next_step(
            name_to_tool_map,
```

```
                color_mapping,
                inputs,
                intermediate_steps,
                run_manager=run_manager,
            )
            # 检测下一步输出是否为结束标记，如果是则直接返回
            if isinstance(next_step_output, AgentFinish):
                return self._return(
                    next_step_output, intermediate_steps, run_manager=run_manager
                )

            # 将数据添加到中间步骤
            intermediate_steps.extend(next_step_output)
            # 检测是否只有一个步骤，并且该工具需要直接返回(return_direct=True)
            if len(next_step_output) == 1:
                next_step_action = next_step_output[0]
                # 检测是否直接返回
                tool_return = self._get_tool_return(next_step_action)
                if tool_return is not None:
                    return self._return(
                        tool_return, intermediate_steps, run_manager=run_manager
                    )
            # 更新迭代次数+时间间隔
            iterations += 1
            time_elapsed = time.time() - start_time
        # 超过时间限制或者迭代次数则获取停止响应输出并返回最后内容
        output = self.agent.return_stopped_response(
            self.early_stopping_method, intermediate_steps, **inputs
        )
        return self._return(output, intermediate_steps, run_manager=run_manager)
```

## 02. 传统 Agent 组件的缺陷

LangChain 封装的 `Agent` 和 `Agent执行者` 虽然解决了 LCEL 表达式创建的单链应用没法执行 `循环步骤` 的问题，对于一些简单类型的 Agent 智能体创建已经足够使用，但是仍然存在不少缺陷。

假设我们要创建一个 Agent 应用，该 Agent 可以处理数学和物理问题，并由两个不同的 LLM 负责不同的模块，根据用户的提问使用不同的 LLM + 工具列表 + 线路来回答用户的问题，传统的 Agent 就无能为力了。

其实除了上述的问题，传统 Agent 还存在不少缺陷，如下：

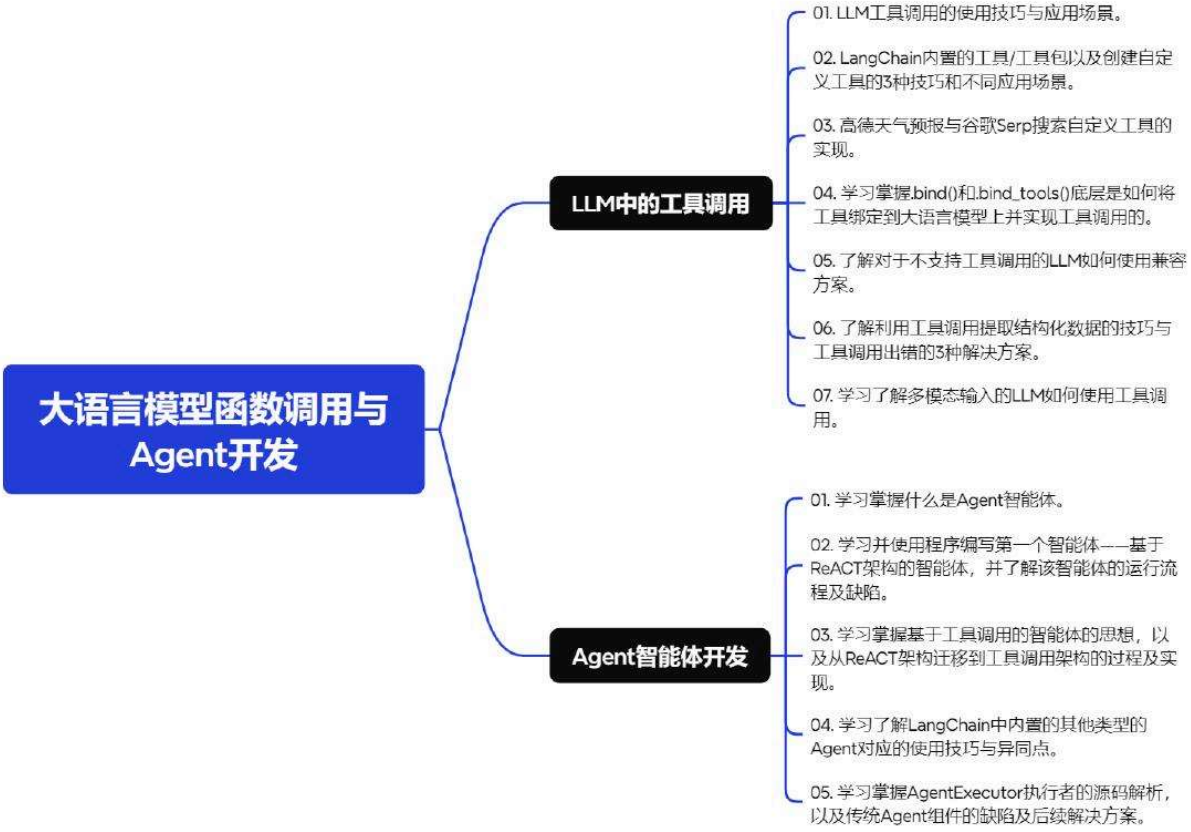
1. 只有 `循环步骤` 并没有 `条件步骤`，一个 Agent 应用只能一条路走到黑，不能执行不同的路由；
2. 没法亦或者很难将多个 `Agent` 融合起来相互协作；
3. 因对 Prompt 与输出解析器的过度封装，导致要修改 Agent 内部的方案变得异常困难；
4. 无论是 Agent 还是 AgentExecutor，因其 `黑盒机制`，无法在执行的过程中进行额外的干预；
5. 想对 `Agent` 进行扩展或者动态切换 LLM 难度非常大，例如 `添加记忆`、`切换LLM` 等；

这也是 LangChain 旧版本被人诟病的原因，包括 2023 年底在 LLM 领域很火的一篇文章《**为什么我放弃了 LangChain?** 》，链接：https://www.jiqizhixin.com/articles/2024-06-24-10。

这是因为在 LangChain 设计的早期（v0.1版本之前），早期的传统链（非LCEL表达式）、传统 Agent 智能体的过度封装，并且层层抽象，让代码变得非常复杂（就像一个组件同时拥有 N 种调用方法一样）。

不过这个弊端随着 LCEL 表达式和今年年初的 LangGraph 发布，普遍都被解决了，更易用与统一的接口（LCEL 表达式统一 invoke、stream、batch 接口），控制性更高、支持循环/逻辑的 LangGraph 图结构程序、实时监控 LLM 的 LangSmith 平台，让 LLM/Agent 应用的开发变得超级简单，甚至串联数百个节点、上百种工具与逻辑路线的 Agent 工作流也成为了现实。

但是对于 `传统Agent组件` 来讲，我们还是需要去掌握，因为在复杂度较小的情况下，该思路还是非常值得借鉴的，但对于一些复杂应用就无能为力了，下一章我们会来学习 LangChain 的最后一块拼图——**LangGraph**，掌握利用 LangGraph 构建复杂应用的技巧及底层运行原理，并且**在 LLMOps 项目中，将 LangGraph 与知识库、工具、审核、多用户/多应用、多模态输入输出等功能结合起来**，在实战中感受 LangGraph + LECL 表达式带来的酣畅淋漓的体验。


大语言模型函数调用与Agent开发
LLM中的工具调用
01. LLM工具调用的使用技巧与应用场景。
02. LangChain内置的工具/工具包以及创建自定义工具的3种技巧和不同应用场景。
03. 高德天气预报与谷歌Serp搜索自定义工具的实现。
04. 学习掌握.bind()和.bind_tools()底层是如何将工具绑定到大语言模型上并实现工具调用的。
05. 了解对于不支持工具调用的LLM如何使用兼容方案。
06. 了解利用工具调用提取结构化数据的技巧与工具调用出错的3种解决方案。
07. 学习了解多模态输入的LLM如何使用工具调用。
Agent智能体开发
01. 学习掌握什么是Agent智能体。
02. 学习并使用程序编写第一个智能体——基于ReACT架构的智能体，并了解该智能体的运行流程及缺陷。
03. 学习掌握基于工具调用的智能体的思想，以及从ReACT架构迁移到工具调用架构的过程及实现。
04. 学习了解LangChain中内置的其他类型的Agent对应的使用技巧与异同点。
05. 学习掌握AgentExecutor执行者的源码解析，以及传统Agent组件的缺陷及后续解决方案。