大语言模型函数调用与 Agent 开发

大语言模型工具上手与定义

- 学习掌握 LangChain 构建自定义工具的 3 种方案，以及不同方案之间的差异性。
- 学习掌握 LangChain 内置工具与工具包的快速上手，实现利用 OpenAI+Dall·E 实现自然语言绘图功能。
- 学习掌握封装基于第三方 API 及内置工具的工具，涵盖了高德天气预报、谷歌搜索工具。

大语言模型函数调用原理及应用

- 学习掌握大语言模型函数调用的作用与应用，了解不支持函数回调的模型兼容方案及对应的注意事项。
- 学习掌握利用大语言模型函数调用实现数据格式化与规范提取技巧，以及 LangChain 底层实现。
- 学习掌握如何使用多模态数据调用工具，为 AI 应用的多模态输入输出构建基础。

Agent 智能体初识与开发

- 深入了解什么是 Agent 智能体以及智能体的组成，以及智能体与大语言模型函数回调的关联。
- 学习掌握使用 LangChain 传统 Agent 组件实现 ReACT 架构的智能体，让大语言模型拥有搜索 + 天气预报查询功能。
- 学习 LangChain 内置封装的 Agent 组件并快速上手，了解 LangChain 传统 Agent 组件的缺陷与不可控性。

LLM 函数调用使用技巧与应用场景

LLM函数调用使用技巧与应用场景

01. 函数调用的本质与组成

在概率架构下，**大语言模型的回复是非结构化且不一致的**，所以将大语言模型集成到其他系统中，是存在一定风险的，例如下游系统中想要 JSON 数据，但是大语言模型返回了一串文本，会让整个流程崩溃。但是如果能更好控制 `响应格式`，就可以轻松地将响应下游集成到其他系统中（虽然通过 prompt 也可以实现，但是不稳定）。

其次，我们都知道大语言模型的训练需要大量的计算资源和时间，因此**它们的知识库通常是在某个知识点之前的数据集上训练**的，例如，GPT-3.5 和 GPT-4 的知识截至 2023 年 10 月，这意味着它们无法提供此后的新消息或事件，为保持时效性，需定期重训模型，单成本高昂且费时。

而 `函数调用(Function Calling)` 的使用就是为了克服大语言模型的上述缺点，LLM 中的函数调用架构其实也非常简单，本质上是对输入 prompt 的自动拆解，利用拆解出来的参数和自动识别出的函数名称，调用外部资源，然后将返回的外部资源添加到 prompt 之中，再次调用 LLMs，从而得到返回结果，运行流程与架构如下：

简单来说，`函数回调` 的作用就是可以更可靠地从大语言模型中获取结构化的数据。

并且目前绝大部分大参数的模型都支持函数调用，而且除了 `系统消息`、`人类消息`、`AI消息` 之外，部分大模型还可以添加 `函数/工具消息`，即将函数的生成结果单独作为消息传递给大模型，让大模型基于工具消息生成对应的内容，例如 GPT 模型。

1. OpenAI 大语言模型聊天接口文档：https://platform.openai.com/docs/api-reference/chat/create
2. 百度文心一言函数调用接口文档：https://cloud.baidu.com/doc/WENXINWORKSHOP/s/5ltxyqupn

对于原生支持 `函数调用` 大语言模型而言，一般来说会提供多一个除了 `prompt` 之外的参数，这个参数用于描述 `函数调用` 的相关参数（并且目前绝大部分 LLM 的函数参数描述格式都保持一致，参考 GPT）。

参数涵盖了**函数名字、函数描述、函数参数、函数调用模型**等，这样大语言模型才能知道可以调用那些函数。

例如 `GPT模型` 执行 `函数调用` 示例：

```
from openai import OpenAI
client = OpenAI()

tools = [
  {
    "type": "function",
    "function": {
      "name": "get_current_weather",
      "description": "Get the current weather in a given location",
      "parameters": {
        "type": "object",
```

```
        "properties": {
          "location": {
            "type": "string",
            "description": "The city and state, e.g. San Francisco, CA",
          },
          "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
        },
        "required": ["location"],
      },
    }
  }
]
messages = [{"role": "user", "content": "What's the weather like in Boston
today?"}]
completion = client.chat.completions.create(
  model="gpt-3.5-turbo-16k",
  messages=messages,
  tools=tools,
  tool_choice="auto"
)

print(completion)
```

例如上方的代码描述了一个函数名为 `get_current_weather`，该函数的作用是查询指定城市的天气信息，函数拥有两个参数 `location` 和 `unit` 分别代表 `城市` 和 `单位`，其中 `城市` 参数是必填的。

当我们提问 `What's the weather like in Boston today?(今天波士顿的天气怎么样？)` 时，大语言模型会根据传递的 `auto` 自动检测需要调用函数，于是从 prompt 中提取出对应的函数参数信息，返回的数据如下（多了一个 `tool_calls` 参数）：

```
{
  "id": "chatcmpl-abc123",
  "object": "chat.completion",
  "created": 1699896916,
  "model": "gpt-4o-mini",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": null,
        "tool_calls": [
          {
            "id": "call_abc123",
            "type": "function",
            "function": {
              "name": "get_current_weather",
              "arguments": "{\n\"location\": \"Boston, MA\"\n}"
            }
          }
        ]
      },
      "logprobs": null,
      "finish_reason": "tool_calls"
```

```
    }
  ],
  "usage": {
    "prompt_tokens": 82,
    "completion_tokens": 17,
    "total_tokens": 99
  }
}
```

接下来就可以根据大模型生成的 `tool_calls` 参数调用对应的函数，从而完成本地特定的函数调用，对于部分大模型来说，还可以将函数结果、前面的历史对话消息传递给大模型，让大模型基于这些信息生成对应的内容。

> **Important**
>
> 需要特别注意的是，大语言模型本身的 `函数调用` 并不会调用我们预定义的 `参数`，而是仅仅生成我们需要调用的函数的调用参数而已，具体调用函数的动作，需要我们在应用代码中实现。

从上面的示例中，很容易看出，`函数调用` = `大语言模型(支持函数调用)` + `预定Prompt` + `函数/工具参数列表` + `本地调用代码`。

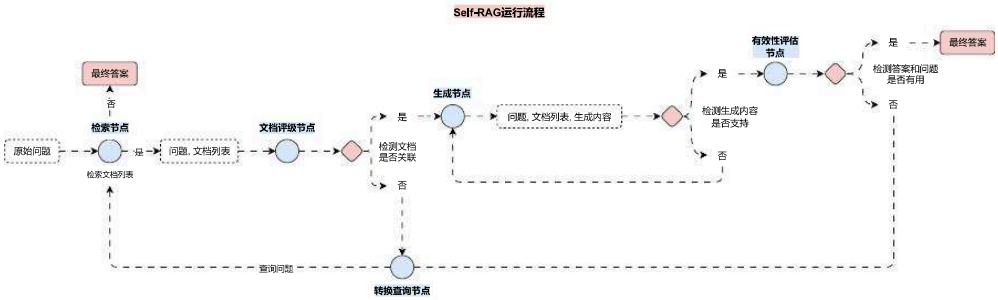
02. 函数调用的想象空间

在函数调用的加持下，让大语言模型作为大脑，接入到上下游系统中，就可以实现以往很多实现不了的功能，**实现万物智能化！**例如：

1. 用户对着微信说：给我每个女性好友发送一条情真切意的拜年消息，还要带一点小幽默。
2. 用户对着富途牛牛：人工智能相关股票，市盈率最低的是哪一个？最近交易量如何？都有那些机构持有？
3. 用户对着京东说：我想买一台 65 寸的电视，不要日货，价格在 5000 元左右。

上述的几个需求，都只需要预设好特定的函数/工具，并将函数/工具的参数描述传递给大语言模型， 大语言模型就会从传递的 prompt 中抽取相关的函数参数，接下来直接执行本地函数的调用即可。

又例如像上一章学习的 `Self-RAG` 优化策略，在该策略的运行流程中，需要让 LLM 自行判断是否需要检索对应的内容，如果需要的话则生成检索内容，否则直接输出内容，同样也可以利用到 `函数调用`，设置 `tool_choice` 为 `auto`，让 LLM 自行做决策即可。



除了单次执行，还可以让 LLM 将需求拆分，拆分成多个任务/函数，让不同的 LLM 执行不同的子任务，并分工合作调用不同的工具/函数，并循环检测输出的内容是否实现了整个任务，最后再将结果输出，恭喜你，发现了**目前基于 LLM 构建通用人工智能的路径**，这也是 2023 年火爆全网的 AutoGPT 的设计架构基础！

LangChain 中的工具与工具包

LangChain中的工具与工具包

01. LangChain 内置工具

前面的课时我们学习过，`函数调用` 是可靠地从大语言模型中获取结构化数据的一个功能，要想使用 `函数调用` 就必须定义好特定的函数或工具，并将工具的相关信息传递给大语言模型，在 LangChain 中封装了大量的 `预设工具`，开箱即用，涵盖了：**搜索、图像生成、百科、视频信息提取、代码执行、数据分析**等。

- LangChain 内置工具列表：https://imooc-langchain.shortvar.com/docs/integrations/tools/

并且在 LangChain 中，所有工具都是 `BaseTool` 的子类，并且工具也是 `Runnable可运行组件`，支持通过 `invoke`、`batch` 和 `stream` 等方法进行调用，使用起来也非常简单，实例化工具后直接调用 invoke 函数即可，除此之外，可以通过 `name`、`description`、`args`、`return_direct` 属性来获取到工具的相关信息。

例如以 `DuckDuckGo搜索` 为例，首先安装特定的依赖库：

```
pip install -U duckduckgo-search
```

示例代码如下：

```
from langchain_community.tools import DuckDuckGoSearchRun

search = DuckDuckGoSearchRun()

print("工具名字: ", search.name)
print("工具描述: ", search.description)
print("工具参数:", search.args)
print("是否直接返回: ", search.return_direct)
print(search.run("LangChain目前最新版本是什么?"))
```

输出内容：

```
工具名字:  duckduckgo_search
工具描述:  A wrapper around DuckDuckGo Search. Useful for when you need to answer questions about current events. Input should be a search query.
工具参数: {'query': {'title': 'Query', 'description': 'search query to look up', 'type': 'string'}}
是否直接返回:  False

LangChain 基准是一个 python 包和相关数据集，用于促进不同认知架构的实验和基准测试。. 每个基准任务都针对常见 LLM 应用程序中的关键功能，例如基于检索的问答、提取、代理工具使用等。. 对于我们的第一个基准，我们发布了一个关于 LangChain python 文档的问答 ... 它建立在 LCEL 的基础之上，添加了两个重要的组件：轻松定义循环的能力（这对代理很重要，但对链来说不需要）和内置内存。. 在 langchain v0.2 中，我们保留了旧的 AgentExecutor--但 LangGraph 正在成为构建代理的推荐方式。. 我们添加了一个预构建的 LangGraph 对象 ... 奋战一年，LangChain首个稳定版本终于发布，LangGraph把智能体构建为图. 著名的大模型智能体工具，现在有大版本更新了。. 不知不觉，LangChain已经问世一年了。. 作为一个开源框架，LangChain 提供了构建基于大模型的 AI 应用所需的模块和工具，大大降低了 AI 应用 ... LangChain 承诺「让开发人员一个下午就能从一个想法变成可运行的代码」，但随着我们的需求变得越来越复杂，问题也开始浮出水面。. LangChain 变成了阻力的根源，而不是生产力的根源。. 随着 LangChain 的不灵活性开始显现，我们开始深入研究 LangChain 的内部结构 ... LangChain 随时间推移的变化. 我们文档中新增的"LangChain 随时间推移的变化"部分可帮助您及时了解变化。其中提供了有关 LangChain 如何变化、如何升级以及如何将旧版本中的先前概念映射到新版本中的指南。这是社区强烈要求的功能，我们也认为它非常必要!
```

如果需要将工具转换成 `openai` 工具形式参数，可以使用 `convert_to_openai_tool()` 辅助函数，示例如下：

```python
from langchain_community.tools import DuckDuckGoSearchRun
from langchain_core.utils.function_calling import convert_to_openai_tool

search = DuckDuckGoSearchRun()
print(convert_to_openai_tool(search))
```

输出内容：

```
{'type': 'function', 'function': {'name': 'duckduckgo_search', 'description': 'A wrapper around DuckDuckGo Search. Useful for when you need to answer questions about current events. Input should be a search query.', 'parameters': {'type': 'object', 'properties': {'query': {'description': 'search query to look up', 'type': 'string'}}, 'required': ['query']}}}
```

02. LangChain 内置工具包

在 LangChain 中，**工具包是一组设计用于一起执行特定任务的工具集**，它们具有便捷加载的方法——`get_tools()`，并且所有工具包都公开了一个 `get_tools()` 方法，该方法返回一个工具列表，这样就无需一个一个加载特定的工具了。

- LangChain 内置工具包列表：https://imooc-langchain.shortvar.com/docs/integrations/toolkits/

`工具包` 一般是同一个服务提供商提供的一系列工具，例如在 LangChain 中封装了 `Azure AI` 的对应工具包，该工具包提供了 5 个工具：

1. **AzureAiServicesImageAnalysisTool**：用于从图像中提取标题、对象、标签和文本。
2. **AzureAiServicesDocumentIntelligenceTool**：用于从文档中提取文本、表格和键值对。
3. **AzureAiServicesSpeechToTextTool**：用于将语音转录为文本。

4. **AzureAiServicesTextToSpeechTool**：用于将文本合成语音。
5. **AzureAiServicesTextAnalyticsForHealthTool**：用于提取医疗实体。

使用起来也非常简单，首先安装特定的依赖包：

```
pip install -U azure-ai-formrecognizer azure-cognitiveservices-speech azure-ai-
textanalytics azure-ai-vision-imageanalysis
```

然后配置 Azure 的相关环境变量：

```
import os
os.environ["OPENAI_API_KEY"] = "sk-"
os.environ["AZURE_AI_SERVICES_KEY"] = ""
os.environ["AZURE_AI_SERVICES_ENDPOINT"] = ""
os.environ["AZURE_AI_SERVICES_REGION"] = ""
```

接下来就可以创建工具包，示例如下：

```
from langchain_community.agent_toolkits import AzureAiServicesToolkit

toolkit = AzureAiServicesToolkit()
print([tool.name for tool in toolkit.get_tools()])
```

输出内容如下：

```
['azure_ai_services_document_intelligence',
 'azure_ai_services_image_analysis',
 'azure_ai_services_speech_to_text',
 'azure_ai_services_text_to_speech',
 'azure_ai_services_text_analytics_for_health']
```

不过由于 `工具包` 包含了多组工具，一般很少单独拆分出来使用，通常用在 `Agent` 或者 `函数调用` 中，这样无需一次性实例化多个工具，在实际开发中，往往更多的是手动实例化工具进行组装。

创建自定义工具的 3 种技巧与使用场景

创建自定义工具的3种技巧与使用场景

在使用 `函数调用` 或者创建 `智能体` 时，我们需要提供 `工具列表`，以便大语言模型可以使用这些工具，虽然 LangChain 内部集成了大量的工具和工具包，但并不一定适合我们的业务场景，更多场合下我们会使用自定义工具，在 LangChain 中提供了 3 种构建自定义工具的技巧：`@tool 装饰器`、`StructuredTool.from_function()类方法`、`BaseTool子类`，不同的方式有不同的优缺点与应用场景。

01. @tool 装饰器

`@tool` 装饰器是定义自定义工具的最简单方式，可以快速将当前的 `函数` 改造成 `大语言模型工具`，该装饰器默认使用函数名称作为工具名称，但可以通过传递字符串作为第一个参数来覆盖，此外，该装饰器将使用函数的 `文档字符串` 作为工具的描述，所以被装饰的函数必须要提供 `文档字符串`。

使用示例如下：

```
from langchain_core.tools import tool


@tool
def multiply(a: int, b: int) -> int:
    """将传递的两个数字相乘"""
    return a * b


# 打印下该工具的相关信息
print("名称: ", multiply.name)
print("描述: ", multiply.description)
print("参数: ", multiply.args)
print("直接返回: ", multiply.return_direct)

# 调用工具
print(multiply.invoke({"a": 2, "b": 8}))
```

输出内容：

```
名称:  multiply
描述:  将传递的两个数字相乘
参数:  {'a': {'title': 'A', 'type': 'integer'}, 'b': {'title': 'B', 'type':
'integer'}}
直接返回:  False
16
```

除了使用默认的配置外，`@tool` 装饰器还可以传递多个参数来执行相应的配置，例如传递 `工具名字`、`参数描述`、`是否直接返回` 等。

```
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.tools import tool


class CalculatorInput(BaseModel):
    a: int = Field(description="第一个数字")
```

```
    b: int = Field(description="第二个数字")


@tool("multiply_tool", args_schema=CalculatorInput, return_direct=True)
def multiply(a: int, b: int) -> int:
    """将传递的两个数字相乘"""
    return a * b


# 打印下该工具的相关信息
print("名称: ", multiply.name)
print("描述: ", multiply.description)
print("参数: ", multiply.args)
print("直接返回: ", multiply.return_direct)

# 调用工具
print(multiply.invoke({"a": 2, "b": 8}))
```

输出内容:

```
名称:  multiply_tool
描述:  将传递的两个数字相乘
参数:  {'a': {'title': 'A', 'description': '第一个数字', 'type': 'integer'}, 'b':
{'title': 'B', 'description': '第二个数字', 'type': 'integer'}}
直接返回:  True
16
```

当使用 `Google-style文档字符串` 风格进行函数注释时，还可以设置 `parse_docstring` 为 True，这样 `@tool` 装饰器还可以获取到每个参数的相关解释，例如:

```
@tool(parse_docstring=True)
def foo(bar: str, baz: int) -> str:
    """The foo.

    Args:
        bar: The bar.
        baz: The baz.
    """
    return bar

print(foo.args)
```

输出内容:

```
{'bar': {'title': 'Bar', 'description': 'The bar.', 'type': 'string'}, 'baz':
{'title': 'Baz', 'description': 'The baz.', 'type': 'integer'}}
```

对于一个原有的函数，并且该函数的参数相对来说比较简单，我们可以考虑使用 `@tool` 装饰器来转换该函数，可以极大减少重复性的工作，但是 `@tool` 装饰器装饰的工具并不能同时拥有 `同步` 和 `异步` 方法，只可以单独装饰，例如:

```python
from langchain_core.tools import tool
@tool
async def amultiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b
```

所以对于一些需要同时考虑 `同步` 和 `异步` 的工具来说，`@tool` 装饰器就没法使用了。

02. StructuredTool 类方法

第 2 种快速创建 `工具` 的技巧是使用 `StructuredTool.from_function()` 类方法，该方法提供了比 `@tool` 装饰器更多的配置项，例如同时支持同步和异步等，修改的示例如下：

```python
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.tools import StructuredTool


class CalculatorInput(BaseModel):
    a: int = Field(description="第一个数字")
    b: int = Field(description="第二个数字")


def multiply(a: int, b: int) -> int:
    """将传递的两个数字相乘"""
    return a * b


async def amultiply(a: int, b: int) -> int:
    """将传递的两个数字相乘"""
    return a * b


calculator = StructuredTool.from_function(
    func=multiply,
    coroutine=amultiply,
    name="multiply_tool",
    description="用于将传递的两个整型相乘",
    return_direct=True,
    args_schema=CalculatorInput,
)

# 打印下该工具的相关信息
print("名称: ", calculator.name)
print("描述: ", calculator.description)
print("参数: ", calculator.args)
print("直接返回: ", calculator.return_direct)

# 调用工具
print(calculator.invoke({"a": 2, "b": 8}))
```

输出内容：

```
名称:  multiply_tool
描述:  用于将传递的两个整型相乘
参数:  {'a': {'title': 'A', 'description': '第一个数字', 'type': 'integer'}, 'b':
{'title': 'B', 'description': '第二个数字', 'type': 'integer'}}
直接返回:  True
16
```

对于已有的函数，并且想同时支持 `同步` 和 `异步`，可以考虑使用 `StructuredTool.from_function()` 类方法来实现，该方法的配置项很灵活，而且无需太多额外代码。

03. BaseTool 子类

在 LangChain 中，所有的工具都是 `BaseTool` 子类，并且使用 `@tool` 或者 `StructuredTool.from_function()` 创建的工具，底层都是包装成了 `StructuredTool`，本质上也是 `BaseTool` 的子类。

所以如果对一个自定义工具来说，如果目前并没有任何一段已经实现的代码，则可以考虑继承 `BaseTool` 基类，并实现 `_run()` 方法来实现自定义工具（和使用 `StructuredTool.from_function()` 代码量差异并不是特别大）。

代码示例：

```python
from typing import Any, Type

from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.tools import BaseTool


class CalculatorInput(BaseModel):
    a: int = Field(description="第一个数字")
    b: int = Field(description="第二个数字")


class MultiplyTool(BaseTool):
    """乘法计算工具"""
    name = "multiply_tool"
    description = "将传递的两个数字相乘"
    args_schema: Type[BaseModel] = CalculatorInput

    def _run(self, a: int, b: int) -> Any:
        return a * b


calculator = MultiplyTool()

# 打印下该工具的相关信息
print("名称: ", calculator.name)
print("描述: ", calculator.description)
print("参数: ", calculator.args)
print("直接返回: ", calculator.return_direct)

# 调用工具
print(calculator.invoke({"a": 2, "b": 8}))
```

输出示例:

```
名称:  multiply_tool
描述:  将传递的两个数字相乘
参数:  {'a': {'title': 'A', 'description': '第一个数字', 'type': 'integer'}, 'b':
{'title': 'B', 'description': '第二个数字', 'type': 'integer'}}
直接返回:  False
16
```

高德天气预报查询插件的集成与编写

高德天气预报查询插件的集成与编写

01. 高德城市与天气预报接口

高德 Web 服务 API 向开发者提供了大量的 HTTP 接口，涵盖了：地理/逆地理编码、路径规划、行政区域查询、IP定位、天气查询、坐标转换、轨迹纠偏等接口，并且向开发者提供了足量的调用额度，用于开发/调试各类地图类型的应用（商用需付费）。

- 高德 Web 服务 API 链接：https://lbs.amap.com/api/webservice/summary
- 高德开放平台控制台：https://console.amap.com/dev/index

在 LLMOps 项目中，我们会集成一个 `根据城市查询天气预报的自定义工具`，所以可以考虑使用高德提供的服务，在使用之前，必须实名认证高德开放平台，并且创建 `应用`，获取好对应的 `秘钥`，并将其配置到环境变量中，例如：

```
# 高德工具
GAODE_API_KEY=00b0****e6e7
```

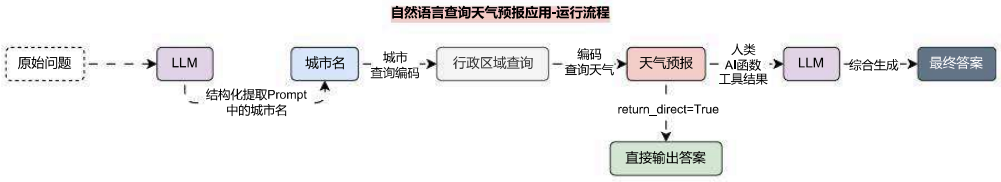
在高德提供的服务中，`天气预报` 查询接口必须传递 `城市编码` 才可以获取对应城市的信息（天气预报），接口如下：

```
https://restapi.amap.com/v3/weather/weatherInfo?city=<城市编码>&key=<用户key>
```

例如 `110101` 代表北京，所以需要一个额外的接口来根据传入的 `城市名` 获取到相应的 `城市编码`（行政区域查询），接口如下：

```
https://restapi.amap.com/v3/config/district?keywords=<城市名>&subdistrict=0&key=<
用户的key>
```

所以要完成一个输入 `城市名` 查询天气预报的函数/工具，必须先调用 `行政区域查询` 将城市转换为城市编码，然后在利用城市编码调用 `天气预报` 接口，从而完成整个流程，如下：



02. 天气预报查询工具实现

由于该接口并没有已经实现好的任何代码，所以可以考虑使用 `BaseTool子类` 的形式来实现，即工具接收一个参数 `city` 表示需要查询天气预报的城市名，然后调用 `行政区域查询` 接口获取该城市对应的 `行政编码`，接下来将 `行政编码` 传递给 `天气预报` 接口，获取最终天气信息。

示例代码如下：

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

import dotenv
```

```python
import requests
from langchain_core.pydantic_v1 import Field, BaseModel
from langchain_core.tools import BaseTool

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"


gaode_weather = GaodeWeatherTool()

print(gaode_weather.invoke({"city": "广州"}))
```

输出内容:


```
{"status": "1", "count": "1", "info": "OK", "infocode": "10000", "forecasts":
[{"city": "广州市", "adcode": "440100", "province": "广东", "reporttime": "2024-08-
12 15:30:27", "casts": [{"date": "2024-08-12", "week": "1", "dayweather": "中雨",
"nightweather": "中雨", "daytemp": "34", "nighttemp": "25", "daywind": "北",
"nightwind": "北", "daypower": "1-3", "nightpower": "1-3", "daytemp_float":
"34.0", "nighttemp_float": "25.0"}, {"date": "2024-08-13", "week": "2",
"dayweather": "中雨", "nightweather": "中雨", "daytemp": "33", "nighttemp": "25",
"daywind": "北", "nightwind": "北", "daypower": "1-3", "nightpower": "1-3",
"daytemp_float": "33.0", "nighttemp_float": "25.0"}, {"date": "2024-08-14",
"week": "3", "dayweather": "中雨", "nightweather": "中雨-大雨", "daytemp": "33",
"nighttemp": "25", "daywind": "北", "nightwind": "北", "daypower": "1-3",
"nightpower": "1-3", "daytemp_float": "33.0", "nighttemp_float": "25.0"},
{"date": "2024-08-15", "week": "4", "dayweather": "中雨-大雨", "nightweather": "中
雨-大雨", "daytemp": "33", "nighttemp": "25", "daywind": "北", "nightwind": "北",
"daypower": "1-3", "nightpower": "1-3", "daytemp_float": "33.0",
"nighttemp_float": "25.0"}]}]}
```

谷歌实时信息搜索插件的集成与编写

谷歌实时信息搜索插件的集成与编写

01. Serper谷歌搜索

谷歌官方虽然提供了 API 搜索功能，但是体验感并不好，而且访问速度非常慢，所以在这里我们考虑使用第三方的谷歌搜索提供商——Serper，Serper 提供的谷歌搜索会在后台浏览器模拟真实的页面运行，完全模仿人类的操作，可以确保获得用户真正看到的内容（目前体验测试来看，Serper 比谷歌官方提供的检索服务内容更全面，响应速度更快）。

参考资料：

- LangChain Serper 谷歌搜索集成：https://imooc-langchain.shortvar.com/docs/integrations/tools/google_serper/
- Serper 官网：https://serper.dev/

可以使用任意的邮箱/Github/Google账号注册登录 SerperAPI 官网， 并管理和创建 API 秘钥，将其配置到环境变量中，如下：

```
# 谷歌Serper搜索
SERPER_API_KEY=9b5bd82f638b1e51a9bf14802a0a2799286b194e
```

在 LangChain 中，已经为 SerpAPI 搜索接口进行了工具的封装，目前有两个内置工具：**GoogleSerperRun、GoogleSerperResults**，其中 `GoogleSerperRun` 会单纯返回搜索文本的内容，而 `GoogleSerperResults` 还会包含检索网页链接等元数据。

不过 LangChain 封装的 `GoogleSerperRun` 工具内部并没有添加 `工具参数说明`，并且 `工具描述` 也是英文的，对于一些参数量比较小，需要将其相应的描述等改成中文情景。

另外 `Serper谷歌搜索` 还有一个很接近的工具——`SerpAPI`，这两个产品在名字和功能上非常接近，而且也特别容易弄混，该工具在 LangChain 中也进行封装，参考资料：

- LangChain SerpAPI 谷歌搜索集成：https://imooc-langchain.shortvar.com/docs/integrations/tools/serpapi/
- SerpAPI 官网：https://serpapi.com/

02. 谷歌实时信息搜索工具实现

在 LangChain 中，实现对 `GoogleSerperRun` 工具的二次封装其实非常简单，甚至不需要重新定义一个工具类，只需要在实例化的时候，传递对应的参数即可，示例如下：

```
import dotenv
from langchain_community.tools import GoogleSerperRun
from langchain_community.utilities import GoogleSerperAPIWrapper
from langchain_core.pydantic_v1 import BaseModel, Field


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


dotenv.load_dotenv()

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(),
)

print(google_serper.invoke("马拉松的世界记录是多少?"))
```

输出示例:

```
2004年1月1日，国际田联宣布，马拉松开始拥有世界纪录。 2023年10月8日晚（北京时间），在芝加哥马拉松比赛中，肯尼亚人基普图姆在芝加哥马拉松比赛中以2小时00分35秒的成绩打破基普乔格保持的世界纪录。
```

ChatModel 使用函数调用的技巧与流程

ChatModel使用函数调用的技巧与流程

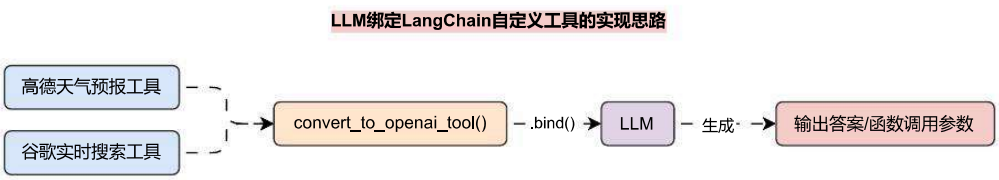
01. bind()与bind_tools()

在前面的课时中，我们学习过在 OpenAI 的 GPT 模型如何添加工具的描述参数，即在实例化的时候传递多一个 `tools` 参数，如下：

```
completion = client.chat.completions.create(
  model="gpt-3.5-turbo-16k",
  messages=messages,
  tools=tools,
  tool_choice="auto"
)
```

该流程有没有很熟悉，其实除了上面的这些参数，我们还可以传递 `temperature` 温度参数，这些参数本质上都是模型生成内容时传递的 `运行时参数`，所以在 LangChain 中，我们可以使用 `convert_to_openai_tool` 将自定义工具转换成符合 GPT 模型的参数格式，使用 `.bind()` 函数来传递对应的 `tools` 和 `tool_choice`，从而完成对大语言模型函数的绑定。

运行流程如下：



不过由于上述的步骤太过于常见，所以 LangChain 团队单独对支持 `函数调用` 的大语言模型添加了 `.bind_tools()` 函数，可以快捷地完成此操作，例如 GPT 模型的 `.bind_tools()` 函数的核心代码如下：

```
def bind_tools(
    self,
    tools: Sequence[Union[Dict[str, Any], Type[BaseModel], Callable, BaseTool]],
    *,
    tool_choice: Optional[
        Union[dict, str, Literal["auto", "none", "required", "any"], bool]
    ] = None,
    **kwargs: Any,
) -> Runnable[LanguageModelInput, BaseMessage]:

    formatted_tools = [convert_to_openai_tool(tool) for tool in tools]
    if tool_choice:
        if isinstance(tool_choice, str):
            # tool_choice is a tool/function name
            if tool_choice not in ("auto", "none", "any", "required"):
                tool_choice = {
                    "type": "function",
                    "function": {"name": tool_choice},
                }
            # 'any' is not natively supported by OpenAI API.
            # We support 'any' since other models use this instead of 'required'.
```

```
            if tool_choice == "any":
                tool_choice = "required"
        elif isinstance(tool_choice, bool):
            tool_choice = "required"
        elif isinstance(tool_choice, dict):
            tool_names = [
                formatted_tool["function"]["name"]
                for formatted_tool in formatted_tools
            ]
            if not any(
                tool_name == tool_choice["function"]["name"]
                for tool_name in tool_names
            ):
                raise ValueError(
                    f"Tool choice {tool_choice} was specified, but the only "
                    f"provided tools were {tool_names}."
                )
        else:
            raise ValueError(
                f"Unrecognized tool_choice type. Expected str, bool or dict. "
                f"Received: {tool_choice}"
            )
        kwargs["tool_choice"] = tool_choice
    return super().bind(tools=formatted_tools, **kwargs)
```

02. LLM绑定天气预报与谷歌实时搜索

例如要将前两节课实现的 `天气预报` 与 `谷歌实时搜索` 接入到 GPT 模型中，其实只需要创建好自定义工具后，将自定义工具组装成列表，然后调用 `.bind_tools()` 函数即可完成 LLM 对函数的绑定，当大语言模型返回的内容携带 `函数调用参数` 时，可以通过 `.tool_calls` 属性来获取对应的信息。

例如：将 GPT 模型绑定 `天气预报` 与 `谷歌实时`，并且当执行工具调用时，将 `工具结果` 附加到历史消息列表中，再次传递给大语言模型，让其生成对应的内容，完整代码如下：

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

import dotenv
import requests
from langchain_community.tools import GoogleSerperRun
from langchain_community.utilities import GoogleSerperAPIWrapper
from langchain_core.messages import ToolMessage
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")
```

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


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"
```

```python
# 1.定义工具列表
gaode_weather = GaodeWeatherTool()
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(),
)
tool_dict = {
    gaode_weather.name: gaode_weather,
    google_serper.name: google_serper,
}
tools = [tool for tool in tool_dict.values()]

# 2.创建Prompt
prompt = ChatPromptTemplate.from_messages([
    ("system", "你是由OpenAI开发的聊天机器人，可以帮助用户回答问题，必要时刻请调用工具帮助用户解答"),
    ("human", "{query}"),
])

# 3.创建大语言模型并绑定工具
llm = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)
llm_with_tool = llm.bind_tools(tools=tools)

# 4.创建链应用
chain = {"query": RunnablePassthrough()} | prompt | llm_with_tool

# 5.解析输出
query = "广州现在天气怎样，有什么适合穿的衣服呢"
resp = chain.invoke(query)
tool_calls = resp.tool_calls

# 6.判断是工具调用还是正常输出结果
if len(tool_calls) <= 0:
    print("生成内容:", resp.content)
else:
    # 7.将历史的系统消息、人类消息、AI消息组合
    messages = prompt.invoke(query).to_messages()
    messages.append(resp)

    # 8.循环遍历所有工具调用信息
    for tool_call in tool_calls:
        tool = tool_dict.get(tool_call.get("name"))
        print("正在执行工具: ", tool.name)
        id = tool_call.get("id")
        content = tool.invoke(tool_call.get("args"))
        print("工具输出: ", content)
        messages.append(ToolMessage(
            content=content,
            tool_call_id=id,
        ))
```

```
    print("输出内容: ", llm.invoke(messages))
```

输出内容:

```
正在执行工具:  gaode_weather
工具输出:  {"status": "1", "count": "1", "info": "OK", "infocode": "10000",
"forecasts": [{"city": "\u5e7f\u5dde\u5e02", "adcode": "440100", "province":
"\u5e7f\u4e1c", "reporttime": "2024-08-12 18:30:27", "casts": [{"date": "2024-08-
12", "week": "1", "dayweather": "\u4e2d\u96e8", "nightweather": "\u4e2d\u96e8",
"daytemp": "34", "nighttemp": "25", "daywind": "\u5317", "nightwind": "\u5317",
"daypower": "1-3", "nightpower": "1-3", "daytemp_float": "34.0",
"nighttemp_float": "25.0"}, {"date": "2024-08-13", "week": "2", "dayweather":
"\u4e2d\u96e8", "nightweather": "\u4e2d\u96e8", "daytemp": "33", "nighttemp":
"26", "daywind": "\u5317", "nightwind": "\u5317", "daypower": "1-3",
"nightpower": "1-3", "daytemp_float": "33.0", "nighttemp_float": "26.0"},
{"date": "2024-08-14", "week": "3", "dayweather": "\u4e2d\u96e8", "nightweather":
"\u5927\u96e8", "daytemp": "33", "nighttemp": "25", "daywind": "\u5317",
"nightwind": "\u5317", "daypower": "1-3", "nightpower": "1-3", "daytemp_float":
"33.0", "nighttemp_float": "25.0"}, {"date": "2024-08-15", "week": "4",
"dayweather": "\u5927\u96e8", "nightweather": "\u5927\u96e8", "daytemp": "32",
"nighttemp": "25", "daywind": "\u5317", "nightwind": "\u5317", "daypower": "1-3",
"nightpower": "1-3", "daytemp_float": "32.0", "nighttemp_float": "25.0"}]}]}
输出内容:  content='广州目前的天气情况是中雨，白天气温为34摄氏度，夜间气温为25摄氏度。根据天气
情况，建议您穿着轻薄透气的衣物，同时带上一把雨伞以备不时之需。请注意保持身体健康，避免长时间暴露在
雨中。' response_metadata={'token_usage': {'completion_tokens': 121,
'prompt_tokens': 665, 'total_tokens': 786}, 'model_name': 'gpt-3.5-turbo-16k-
0613', 'system_fingerprint': None, 'finish_reason': 'stop', 'logprobs': None}
id='run-0403f348-e59d-4ca7-9ad1-951a585d2010-0' usage_metadata={'input_tokens':
665, 'output_tokens': 121, 'total_tokens': 786}
```

学习到这里，其实各位同学已经在有意无意之间构建出了第一个用程序实现的 `Agent`，这个 `Agent` 拥有 `实时信息搜索` 和 `天气预报查询` 功能，但是有没有发现一个问题，原本我们使用 LECL 表达式构建的链应用是非常优雅的，但是加入了 `判断`、`循环`、`工具调用` 等模块后，维护起来也相对吃力，不过这个问题很快就会解决，在 LangChain 中针对 Agent 应用的创建，目前有两种封装好的策略，一种使用传统 Agent，另外一种使用 LangGraph。

不支持函数调用的大语言模型解决技巧

不支持函数调用的大语言模型解决技巧

01. Prompt实现函数调用功能思路

由于市面上的大语言模型众多，并不是所有的大语言模型都支持 `函数调用` 这个功能的，还存在相当多的大模型并不支持（特别是开源模型），对于这类模型，其实也可以通过编写特定的 `Prompt`，即可让模型调用适当的工具，原理其实就是让大语言模型尽可能按照特定的格式进行输出（例如输出函数的调用参数 JSON 数据）。

所以在这种解决方案中，就需要将 `函数调用` 的相关参数嵌入到 `prompt` 中一起传递给大语言模型，并且告知大语言模型生成参数的响应格式，接下来针对大语言模型输出的内容进行解析，如果解析到参数则调用函数，否则正常输出。

该解决方案虽然可以临时为大语言模型添加 `函数调用` 功能，但是缺陷也非常大：

1. 大模型输出的不确定性，并不是每一次都可以正确调用函数的，应用很容易出错。
2. 如果函数参数很复杂，产生的 prompt 可能会非常长，占用大模型的上下文。
3. 没法一次性调用多个工具，或者一次性调用多个工具，会让大语言模型的输出变得异常混乱，对模型考验能力大。

在 LangChain 中，除了封装了 `convert_to_openai_tool()` 工具快速将工具转换成 GPT 模型的函数参数，还可以使用 `render_text_description_and_args()` 或者 `render_text_description()` 快速将工具转换成 `描述文本`，使用技巧一模一样，其中前者转换的描述带参数结构信息，后者仅为函数基础介绍。

02. Prompt函数调用示例

例如在上节课的 `GPT绑定函数调用参数` 示例中，将示例修改成通过 `prompt` 填充函数参数，并让大语言模型按照特定的规则生成相应的数据（可以使用任意的大语言模型来完成这个示例），完整代码如下：

```
import json
import os
from typing import Type, Any, TypedDict, Dict, Optional

import dotenv
import requests
from langchain_community.tools import GoogleSerperRun
from langchain_community.utilities import GoogleSerperAPIWrapper
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import Field, BaseModel
from langchain_core.runnables import RunnablePassthrough, RunnableConfig
from langchain_core.tools import BaseTool, render_text_description_and_args
from langchain_openai import ChatOpenAI

dotenv.load_dotenv()
```

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


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"


class ToolCallRequest(TypedDict):
    """工具调用请求字典"""
    name: str
    arguments: Dict[str, Any]


def invoke_tool(
        tool_call_request: ToolCallRequest, config: Optional[RunnableConfig] =
None,
):
    """
    我们可以使用的执行工具调用的函数。

    :param tool_call_request: 一个包含键名和参数的字典，名称必须与现有工具的名称匹配，参数是
该工具的参数。
    :param config: 这是LangChain使用的包含回调、元数据等信息的配置信息。
    :return: 请求工具的输出内容。
    """
    tool_name_to_tool = {tool.name: tool for tool in tools}
    name = tool_call_request["name"]
    requested_tool = tool_name_to_tool[name]
    return requested_tool.invoke(tool_call_request["arguments"], config=config)


# 1.定义工具列表
gaode_weather = GaodeWeatherTool()
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(),
)
tool_dict = {
    gaode_weather.name: gaode_weather,
    google_serper.name: google_serper,
}
tools = [tool for tool in tool_dict.values()]

system_prompt = """您是一个由OpenAI开发的聊天机器人，可以访问以下一组工具。
以下是每个工具的名称和描述：

{rendered_tools}

根据用户输入，返回要使用的工具的名称和输入。
将您的响应作为具有`name`和`arguments`键的JSON块返回。
`arguments`应该是一个字典，其中键对应于参数名称，值对应与请求的值。"""
prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
```

```
    ("human", "{query}"),
]).partial(rendered_tools=render_text_description_and_args(tools))

llm = ChatOpenAI(model="gpt-3.5-turbo-16k", temperature=0)

chain = (
        {"query": RunnablePassthrough()}
        | prompt
        | llm
        | JsonOutputParser()
        | RunnablePassthrough.assign(output=invoke_tool)
)

print(chain.invoke("马拉松世界记录是多少?"))
```

输出内容:

```
{'name': 'google_serper', 'arguments': {'query': '马拉松世界记录是多少'}, 'output': '2004年1月1日，国际田联宣布，马拉松开始拥有世界纪录。 2023年10月8日晚（北京时间），在芝加哥马拉松比赛中，肯尼亚人基普图姆在芝加哥马拉松比赛中以2小时00分35秒的成绩打破基普乔格保持的世界纪录。'}
```

如果该链应用没法正常调用函数也很正常，因为在链中使用了 `JsonOutputParser()` 输出解析器，但大语言模型并不会所有回答都输出 JSON 信息，如果某一次回复大语言模型输出了 `正常字符串`，例如提问“你好，你是？”，这个时候链应用就崩溃了，所以在生产环境中，涉及到 `函数调用`、`路由逻辑` 等需要规范化数据的内容，尽可能使用支持 `函数调用` 的大语言模型，避免程序变得很脆弱。