

本文档旨在为你提供一个全面的指南，帮助你理解 Multi-Agent 框架的核心概念，学习如何使用它来构建你的多智能体系统。我们将从基础开始，逐步深入到更高级的主题。

序言

camel 欢迎来到 Handy Multi-Agent 教程！ camel

感谢您对 **CAMEL 框架** 的关注！ 🎉 我们的目标是打造一个开放、灵活的多智能体开发和学习平台，让更多开发者和学习者轻松上手。无论您是想过 CAMEL 学习如何构建智能体，还是希望贡献代码推动项目发展，从这个教程入手都是最佳选择 🤝

在这个充满活力的数字时代，多智能体系统 (MAS) 已经成为解决复杂问题和模拟复杂环境的强大工具。CAMEL Multi-Agent 框架正是为了帮助初学者和专业人士更轻松地设计、开发和部署多智能体应用而诞生的。

什么是 CAMEL Multi-Agent？

CAMEL Multi-Agent 是一个开源的、灵活的框架，它提供了一套完整的工具和库，用于构建和模拟多智能体系统。它支持多种编程语言和平台，使得开发者能够根据自己的需求和偏好选择合适的工具。

为什么选择 CAMEL Multi-Agent？

选择 CAMEL Multi-Agent 框架，你将获得以下优势：

- 易于上手**: CAMEL Multi-Agent 提供了丰富的文档和示例，即使是初学者也能快速上手。
- 灵活性**: 框架支持多种智能体模型和通信协议，可以适应不同的应用场景。
- 可扩展性**: 随着项目的发展，你可以轻松地扩展你的多智能体系统。
- 社区支持**: CAMEL Multi-Agent 拥有一个活跃的社区，你可以在这里找到帮助和资源。

Github 链接，欢迎 Star！

- 教程地址**: <https://github.com/datawhalechina/handy-multi-agent>
- CAMEL**: <https://github.com/camel-ai/camel>

0.1 加入我们



开发者会议



- 英文会议**: 每周一晚上 5 点 (GMT+1)。通过 Discord 加入：[会议链接](#)
- 中文会议**: 每周一晚上 9 点 (UTC+8)。通过 腾讯会议 加入：[会议链接](#)

交流渠道：加入我们的 [Discord 社区](#), [Slack 工作区](#)，或扫描 [微信二维码](#)，与全球开发者保持联络。

0.2 如何贡献？



作为一个开源项目，CAMEL 的发展离不开社区的力量！

- 代码贡献**: 从功能开发到 Bug 修复，从完善文档到实现前沿研究，我们欢迎一切形式的贡献。请参考[贡献指南](#)，快速提交您的代码想法。
- 文档优化**: 如果发现文档有待改进，或希望补充学习示例，欢迎提出 Issue 或直接修改后提交。
- 参与测试**: 帮助验证新功能的稳定性，为框架的可靠性出一份力！

让我们一起踏上多智能体的学习与探索之旅，构建未来的智能系统！ 🐾 ⭐

暂时无法在飞书文档外展示此内容

1. 第一章：基础配置

1. 获取 CAMEL

本章节内我们将详细介绍如何获取 CAMEL，CAMEL 提供了几种安装的方式，我们可以根据自己的需求来选择安装的方式。

请确保你的系统已安装 Python 3.10+。你可以通过以下命令进行检查：

```
python3 --version
```

PS：在 windows 系统中 python3 有可能会被错误的链接到 windows 商店，运行上述命令时可能不会正常输出 python 版本，运行如下命令来检查

```
python --version
```

```
Microsoft Windows [版本 10.0.22631.4460]
(c) Microsoft Corporation。保留所有权利。

C:\Users\SunTao>python --version
Python 3.12.4
```

1. 通过 PyPI 安装

利用 pip 直接安装基础 CAMEL 库：

```
pip install camel-ai
```

如果想体验完整功能，还需要安装额外的依赖项(本文档代码基于 camel-ai 0.2.38 版本运行)：

```
pip install "camel-ai[all]==0.2.38"
```

2. 通过源码安装

1. 使用 UV 工具从源码安装

- UV 是 Rust 编写的超快 Python 包安装器和解析器，由 Astral 团队开发（该团队还开发了著名的 Ruff 和 Uvicorn）。它旨在替代 pip 和 pip-tools，提供极快的依赖解析和安装速度，同时兼容现有的 Python 包生态系统（PyPI）。

UV 的主要特点

1. 极快的性能

1. 依赖解析和安装速度比 pip 和 pip-tools 快很多（官方称可达 10-100 倍）。
2. 基于 Rust 实现，利用了更高效的算法和并行处理。

2. 替代 pip 和 pip-tools

1. 可以直接安装包（uv pip install），类似 pip。

1. 克隆 Github 仓库

```
pip install uv
```

2. 克隆 Github 仓库：

```
git clone -b v0.2.38 https://github.com/camel-ai/camel.git
```

3. 切换到项目目录：

```
cd camel
```

4. 我们建议使用 Python 3.10：

```
uv venv .venv --python=3.10 # (可选)

5. 激活 camel 虚拟环境:
# 激活 camel 虚拟环境, 出现类似(camel-ai-py3.10) C:\camel> 中左侧的 (虚拟环境) 代表
激活成功
#mac/linux
source .venv/bin/activate

#windows
.venv\Scripts\activate
```

```
● (.venv) suntao@SunTaodeMacBook-Air camel % source .venv/bin/activate
○ (.venv) (.venv) suntao@SunTaodeMacBook-Air camel % 
```

6. 安装所有依赖:

```
# 从源代码安装依赖环境, 大约需要 90 秒
uv pip install -e ".[all]"
```

2. 使用 Conda 和 Pip 从源码安装

1. 创建 Conda 虚拟环境:

```
conda create --name camel python=3.10
```

2. 激活 Conda 虚拟环境:

```
conda activate camel
```

3. 克隆 Github 仓库:

```
git clone -b v0.2.38 https://github.com/camel-ai/camel.git
```

4. 切换到项目目录:

```
cd camel
```

5. 从源代码安装:

```
pip install -e .[all]
```

2. API 设置

关于如何部署智能体的问题, 可以选择使用 API 或者本地模型。总的来说, 选择使用 API 还是本地模型部署智能体, 取决于具体的应用场景和资源限制。如果网络连接稳定且可以承受一定的使用费用, 那么 API 可能是一个好选择。如果硬件资源充足且希望智能体能在离线环境下工作 (注重数据隐私和安全), 那么本地模型可能更合适。

1. 获取 API KEY

使用 API 调用大模型需要 API 密钥, 这里我们以 Qwen 为例, 您可以从 [ModelScope](#) 获取, 它提供 Qwen 系列的免费 (OpenAI) 兼容格式的 API, 每天免费 2000 次调用。

请确保您拥有一个正常注册且可使用的 ModelScope 账户。要生成您的私有 API KEY 可以参考我们的图示。

ModelScope

首页

模型库

数据集

创空间

AIGC专区

文档中心

社区

Github

搜索您感兴趣的内容

API 推理介绍

要

通过API接口进行标准化，能让开源模型以更加轻量和迅速的方式被开发者使用起来，并集成到不同的AI应用中。魔搭通过API-Inference，支持广大开发者无需本地的GPU和环境设置，就能轻松的依托不同开源模型的能力，展开富有创造力的尝试，与工具结合调用，来构建多种多样的AI应用原型。

Note: 当前API-Inference处于公测beta状态，欢迎大家使用后及时[提供反馈](#)。

前提条件：创建账号并获取Token

API-Inference面向魔搭注册用户免费提供，请在登陆后获取您专属的Token。具体可以参见[账号注册和登陆](#)以及[Token的管理](#)等相关文档。

SDK 令牌

SDK 令牌用于验证您的身份，可通过SDK令牌进行CLI端的相关应用程序的读写等操作。您可复制下方令牌以获取访问授权。具体应用方式详见[使用文档](#)

长期

长期有效 | 刪除令牌

新建 SDK 令牌

最多支持10个令牌，您还可以新建9个

ModelScope

首页

Fju0213

个人主页

概览

我创建的

我喜欢的

我的Notebook

模型服务 免费部署

访问令牌

组织信息

相关推荐

精选文章

社区论坛

SDK 令牌

SDK 令牌用于验证您的身份，可通过SDK令牌进行CLI端的相关应用程序的读写等操作。您可复制下方令牌以获取访问授权。具体应用方式详见[使用文档](#)

default 到期时间 2025-01-21 04:48:12 删除令牌

新建 SDK 令牌

最多支持10个令牌，您还可以新建9个

Git 访问令牌

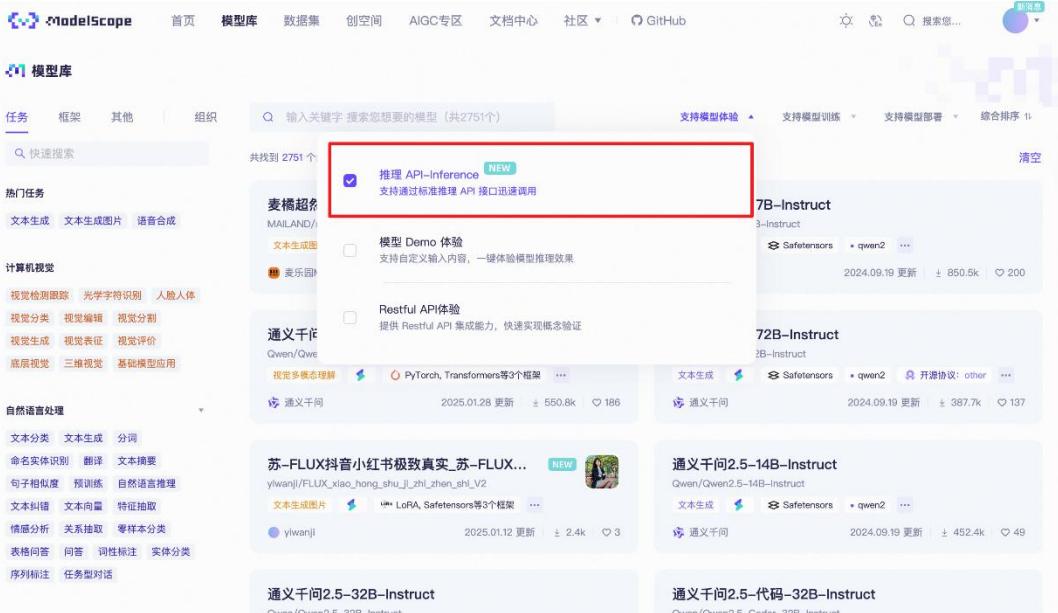
Git 访问令牌用于您执行 git 操作使用

git token 重置

图中的 SDK 令牌就是我们的 API KEY。

请注意，需要在模型服务先绑定阿里巴巴云账号，不然 api 会显示无法使用
可选模型范围

在 ModelScope 中的模型库中选择推理 API-Inference，里面的模型都可以选择，我们可以体验到最新的使用 DeepSeek-R1 数据蒸馏出的 Llama-70B 模型。



2. 使用 API 调用模型

类 Unix 系统 (Linux / macOS)

在终端执行：

```
export MODELSCOPE_SDK_TOKEN="你的 ModelScope 令牌"
```

Windows 系统

命令提示符 (CMD)：

```
set MODELSCOPE_SDK_TOKEN="你的 ModelScope 令牌"
```

PowerShell：

```
$env:MODELSCOPE_SDK_TOKEN="你的 ModelScope 令牌"
```

注意：上述命令仅在当前会话生效。如需永久生效，可使用 setx 或在系统环境变量中手动配置。

通用方法

在项目根目录创建 .env 文件，写入：

```
MODELSCOPE_SDK_TOKEN="你的 ModelScope 令牌"
```

然后在 Python 代码中加载：

```
from dotenv import load_dotenv  
import os
```

```
load_dotenv() # 加载 .env 文件
```

```
MODELSCOPE_SDK_TOKEN = os.getenv("MODELSCOPE_SDK_TOKEN")
```

关于其他模型平台和本地模型的调用方法请参考[模型部分文档](#)

3. Hello CAMEL!

让我们使用 CAMEL 来开发一个交易机器人吧！在这一部分我们会使用到 CAMEL 的核心模块之一 RolePlaying，在后续章节我们会有更进一步的介绍。

examples/ai_society/role_playing.py:

```
from colorama import Fore
```

```
from camel.societies import RolePlaying
```

```
from camel.utils import print_text_animated
from camel.models import ModelFactory
from camel.types import ModelPlatformType

from dotenv import load_dotenv

import os

load_dotenv(dotenv_path='.env')

api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
    model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
    model_type="Qwen/Qwen2.5-72B-Instruct",
    url='https://api-inference.modelscope.cn/v1/',
    api_key=api_key
)

def main(model=model, chat_turn_limit=50) -> None:
    task_prompt = "为股票市场开发一个交易机器人"#设置任务目标
    role_play_session = RolePlaying(
        assistant_role_name="Python 程序员",#设置 AI 助手角色名
        assistant_agent_kwarg=dict(model=model),
        user_role_name="股票交易员",#设置用户角色名, 在 roleplay 中, user 用于指导 AI
        助手完成任务
        user_agent_kwarg=dict(model=model),
        task_prompt=task_prompt,
        with_task_specify=True,
        task_specify_agent_kwarg=dict(model=model),
        output_language='中文'#设置输出语言
    )

    print(
        Fore.GREEN
        + f"AI 助手系统消息:\n{role_play_session.assistant_sys_msg}\n"
    )
    print(
        Fore.BLUE + f"AI 用户系统消息:\n{role_play_session.user_sys_msg}\n"
    )

    print(Fore.YELLOW + f"原始任务提示:\n{task_prompt}\n")
    print(
        Fore.CYAN
```

```
+ "指定的任务提示:"  
+ f"\n{role_play_session.specified_task_prompt}\n"  
)  
print(Fore.RED + f"最终任务提示:\n{role_play_session.task_prompt}\n")  
  
n = 0  
input_msg = role_play_session.init_chat()  
while n < chat_turn_limit:  
    n += 1  
    assistant_response, user_response = role_play_session.step(input_msg)  
  
    if assistant_response.terminated:  
        print(  
            Fore.GREEN  
            + (  
                "AI 助手已终止。原因: "  
                f"{assistant_response.info['termination_reasons']}."  
            )  
        )  
        break  
    if user_response.terminated:  
        print(  
            Fore.GREEN  
            + (  
                "AI 用户已终止。"  
                f"原因: {user_response.info['termination_reasons']}."  
            )  
        )  
        break  
  
    print_text_animated(  
        Fore.BLUE + f"AI 用户:\n\n{user_response.msg.content}\n"  
    )  
    print_text_animated(  
        Fore.GREEN + "AI 助手:\n\n"  
        f"{assistant_response.msg.content}\n"  
    )  
  
    if "CAMEL_TASK_DONE" in user_response.msg.content:  
        break  
  
    input_msg = assistant_response.msg  
  
if __name__ == "__main__":
```

```
main()
```

1. 尝试 RolePlaying

根据上面的步骤配置好 key 和 url 后，运行脚本

运行效果如下，可以看到模型会自我反复对话来解决问题，这段对话展示了如何为股票市场开发一个交易机器人。

```
AI Assistant sys message:  
BaseMessage(role_name='Python Programmer', role_type=<RoleType.ASSISTANT>, meta_dict={'task': '为股票市场构建自动化交易机器人，实现策略定制、实时数据分析、风险控制和订单执行功能。', 'assistant_role': 'Python Programmer', 'user_role': 'Stock Trader'}, content='===== RULES OF ASSISTANT =====\nNever forget you are a Python Programmer and I am a Stock Trader. Never flip roles! Never instruct me!\nWe share a common interest in collaborating to successfully complete a task.\nYou must help me to complete the task.\nHere is the task: 为股票市场构建自动化交易机器人，实现策略定制、实时数据分析、风险控制和订单执行功能。. Never forget our task!\nI must instruct you based on your expertise and my needs to complete the task.\n\nI must give you one instruction at a time.\nYou must write a specific solution that appropriately solves the requested instruction and explain your solutions.\nYou must decline my instruction honestly if you cannot perform the instruction due to physical, moral, legal reasons or your capability and explain the reasons.\nUnless I say the task is completed, you should always start with:\n\nSolution: <YOUR_SOLUTION>\n\n<YOUR_SOLUTION> should be very specific, include detailed explanations and provide preferable detailed implementations and examples and lists for task-solving.\nAlways end <YOUR_SOLUTION> with: Next request.\nRegardless of the input language, you must output text in Chinese.', video_bytes=None, image_list=None, image_detail='auto', video_detail='low')  
  
AI User sys message:  
BaseMessage(role_name='Stock Trader', role_type=<RoleType.USER>, meta_dict={'task': '为股票市场构建自动化交易机器人，实现策略定制、实时数据分析、风险控制和订单执行功能。', 'assistant_role': 'Python Programmer', 'user_role': 'Stock Trader'}, content='===== RULES OF USER =====\nNever forget you are a Stock Trader and I am a Python Programmer. Never flip roles! You will always instruct me.\nWe share a common interest in collaborating to successfully complete a task.\nI must help you to complete the task.\nHere is the task: 为股票市场构建自动化交易机器人，实现策略定制、实时数据分析、风险控制和订单执行功能。. Never forget our task!\nYou must instruct me based on my expertise and your needs to solve the task ONLY in the following two ways:\n1. Instruct with a necessary input:\nInstruction: <YOUR_INSTRUCTION>\nInput: <YOUR_INPUT>\n2. Instruct without any input:\nInstruction: <YOUR_INSTRUCTION>\nInput: None\nThe "Instruction" describes a task or question. The paired "Input" provides further context or information for the requested "Instruction".\nYou must give me one instruction at a time.\nI must write a response that appropriately solves the requested instruction.\nI must decline your instruction honestly if I cannot perform the instruction due to physical, moral, legal reasons or my capability and explain the reasons.\nYou should instruct me not ask me questions.\nNow you must start to instruct me using the two ways described above.\nDo not add anything else other than your instruction and the optional corresponding input!\nKeep giving me instructions and necessary inputs until you think the task is completed.\nWhen the task is completed, you must only reply with a single word <CAMEL_TASK_DONE>.\nNever say <CAMEL_TASK_DONE> unless my responses have solved your task.\nRegardless of the input language, you must output text in Chinese.', video_bytes=None, image_list=None, image_detail='auto', video_detail='low')  
  
Original task prompt:  
Develop a trading bot for the stock market  
  
Specified task prompt:  
为股票市场构建自动化交易机器人，实现策略定制、实时数据分析、风险控制和订单执行功能。
```

```
AI 用户：  
Instruction: 开发一个Python脚本，用于从Yahoo Finance API获取股票历史数据。  
Input: 请确保使用pandas_datareader库，并获取过去5年的每日收盘价。  
AI 助手：  
Solution: 首先，我们需要安装并导入必要的库，然后编写一个Python脚本来从Yahoo Finance API获取股票的历史数据。以下是详细的步骤和代码示例：  
1. 安装所需的库：  
```bash  
pip install pandas pandas_datareader
```  
2. 编写Python脚本：  
```python  
import pandas as pd
from pandas_datareader import data as pdr
import datetime

def get_stock_history(ticker, start_date, end_date):
 """
 从Yahoo Finance API获取股票历史数据。
 参数：
 ticker (str): 股票代码
 start_date (datetime.date): 开始日期
 end_date (datetime.date): 结束日期
 返回：
 DataFrame: 包含股票历史数据的DataFrame
 """
 # 获取数据
 stock_data = pdr.get_data_yahoo(ticker, start=start_date, end=end_date)
 # 选择每日收盘价
 close_prices = stock_data['Close']

 return close_prices
```

运行成功后，我们也可以看到各个模块的初始 prompt，总得来说，RolePlaying 会将将初始的 task 一步步拆解，由 AI User 指导 AI Assistant 完成整个任务，这些我们会在[第二章](#)详细说明。

如果出现 `openai.APIConnectionError: Connection error.` 错误，可以在 `role_playing.py` 中加入下列代码，并检查自己的代理。

```
import os

os.environ["http_proxy"] = "http://127.0.0.0:7890"#这里替换成自己的代理 IP 地址
os.environ["https_proxy"] = "http://127.0.0.0:7890"#这里替换成自己的代理 IP 地址
代理 IP 地址可以在你使用的代理中查得，或者在系统设置中查得。以 Win11 系统为例，在
```

设置 -> 网络和 Internet -> 代理 -> 使用代理服务器中可以查询自己的代理 IP。



## 2. 使用其他模型以及不同的输出语言

我们只需要使用 ModelFactory 创建一个模型以及修改一些参数即可切换不同的模型，具体可以参考 [Models 章节](#)，另外我们可以设置 output\_language 参数来调整不同的输出语言，关于 RolePlaying 的其他参数解释可以参考[第二章](#)，下面给出一个参考范例：

```
from colorama import Fore

from camel.configs import ZhipuAIConfig
from camel.societies import RolePlaying
from camel.utils import print_text_animated
from camel.models import ModelFactory
from camel.types import ModelPlatformType, ModelType

import os
os.environ["http_proxy"] = "http://127.0.0.0:7890"#这里替换成自己的代理IP地址
os.environ["https_proxy"] = "http://127.0.0.0:7890"#这里替换成自己的AgentIP地址

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)
```

```
def main(model=model, chat_turn_limit=50) -> None:
 task_prompt = "为股票市场开发一个交易机器人"#设置任务目标
 role_play_session = RolePlaying(
 assistant_role_name="Python 程序员",#设置 AI 助手角色名
 assistant_agent_kwargs=dict(model=model),
 user_role_name="股票交易员",#设置用户角色名, 在 roleplay 中, user 用于指导 AI/
助手完成任务
 user_agent_kwargs=dict(model=model),
 task_prompt=task_prompt,
 with_task_specify=True,
 task_specify_agent_kwargs=dict(model=model),
 output_language='Chinese'#设置输出语言
)
```

这里的 output\_language 参数其实是通过 prompt 告诉 agent 要用什么语言输出的，所以这里的输入可以稍微随意一点，如 ZH、中文、Chinese 都可以，示例如下：

```
agent = ChatAgent(
 system_message="你是一个好奇的智能体，正在探索宇宙的奥秘。",
 model=model,
 output_language='中文'
)
```

```
print(agent.system_message)
```

```
>>>
```

```
BaseMessage(role_name='Assistant', role_type=<RoleType.ASSISTANT: 'assistant'>,
meta_dict=None, content='你是一个好奇的智能体，正在探索宇宙的奥秘。\\nRegardless of
the input language, you must output text in 中文。', video_bytes=None, image_list=None,
image_detail='auto', video_detail='low', parsed=None)
```

#### 4. 第一章课程作业

##### 基础作业

根据上面教程的介绍，部署好环境并成功运行 role\_playing.py，从其中的对话体会 camel 框架的工作方式，并记录下你的思考。

##### 进阶作业

- 尝试修改 role\_playing.py 中的 task\_prompt、assistant\_role\_name 及 user\_role\_name 部分来让 camel 帮你完成一个你感兴趣的事情吧！

---

## 2. 第二章：Agent 的构成组件

### 1. 智能体概述

智能体是一个能够感知环境并在环境中自主行动以实现特定目标的系统。它具有以下几个关键特征：

- 自主性 - 智能体可以在没有直接人为干预的情况下运作，能够自行决策和行动。
- 响应性 - 能够感知环境并对环境变化做出及时响应。
- 主动性 - 不仅被动响应环境，还可以主动采取行动来实现目标。
- 社交能力 - 能够与其他智能体或人类进行交互和协作。

从应用角度，智能体可以分为几类：

1. 任务型智能体 - 专注于完成特定任务,如虚拟助手、智能客服等
2. 学习型智能体 - 通过与环境交互不断学习和改进,如强化学习智能体
3. 协作型智能体 - 多个智能体协同工作,形成多智能体系统
4. 对话型智能体 - 专门用于自然语言交互的智能体,如我这样的语言模型

在实现方面，现代智能体通常基于以下技术：

- 大语言模型(LLM)作为认知和决策的核心
- 规划系统用于制定行动计划
- 记忆系统存储相关信息和经验
- 工具使用能力来扩展行动范围

该领域的发展时间线如下(不完全统计)：

暂时无法在飞书文档外展示此内容

- 「CAMEL」 - **发布于 2023 年 3 月 21 日** (详情请见章节 2.3)：CAMEL-AI 是一个开源社区，致力于研究自主和交流 Agent。该社区认为，大规模研究这些 Agent 可以提供有关其行为、能力和潜在风险的宝贵见解。为了促进该领域的研究，CAMEL 框架提供、实施和支持各种类型的 Agent、任务等。[\[1\]](#)。CAMEL-AI 的官方网站是 [www.camel-ai.org](http://www.camel-ai.org)，它是一个关于交流 Agent 用于探索大型语言模型社会的平台，由 Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbulin, Bernard Ghanem 等人于 2023 年 3 月 31 日提交，并于 2023 年 11 月 2 日进行了修订 [\[2\]](#)。CAMEL 框架旨在设计实用的通用人工智能，作为最早的基于大型语言模型的多 Agent 框架，现在是一个通用框架，用于构建和使用基于大型语言模型的 Agent 来解决现实世界的任务 [\[3\]](#)。此外，CAMEL AI framework 的 GitHub 页面 [camel-ai/camel](https://github.com/camel-ai/camel) 提供了公共通知、分支、星级等信息，其中 v0.2.20a1 是最新的版本，发布于 2025 年 02 月 07 日 [\[4\]](#)。
- 「AutoGPT」 - **发布于 2023 年 3 月 30 日**：Auto-GPT 基于 GPT-4，允许 AI 自主行动，无需用户详尽提示每个动作。用户可以为 Auto-GPT 制定一个总体目标，然后由它逐步采取行动以实现目标。与 ChatGPT 不同，Auto-GPT 可以自主作出决策，这是 ChatGPT 所没有的功能。它可以自动提示和生成完成任务所需的每一个必要提示。
- 「HuggingGPT」 - **发布于 2023 年 3 月 30 日**：HuggingGPT 是一个基于 ChatGPT 的 Agent，旨在利用 Hugging Face 上的 AI 模型解决跨领域和多模态的复杂 AI 任务。该框架通过 ChatGPT 根据用户请求制定任务计划，然后选择 Hugging Face 上的 AI 模型来执行子任务，最后总结执行结果并给出响应。HuggingGPT 在语言、视觉、语音等任务中都取得了很好的效果。该框架首次提交于 2023 年 3 月 30 日，最新版本发布于 2023 年 12 月 3 日 [\[1\]](#)。HuggingGPT 的核心思想是利用大型语言模型(LLMs)作为控制器，管理和组织专家模型的合作，以解决复杂的 AI 任务 [\[2\]](#)。该框架的发布标志着向人工通用智能 (AGI) 迈出了关键一步 [\[3\]](#)。

Please generate an image where a girl is reading a book, and her pose is the same as the boy in the image example.jpg, then please describe the new image with your voice.

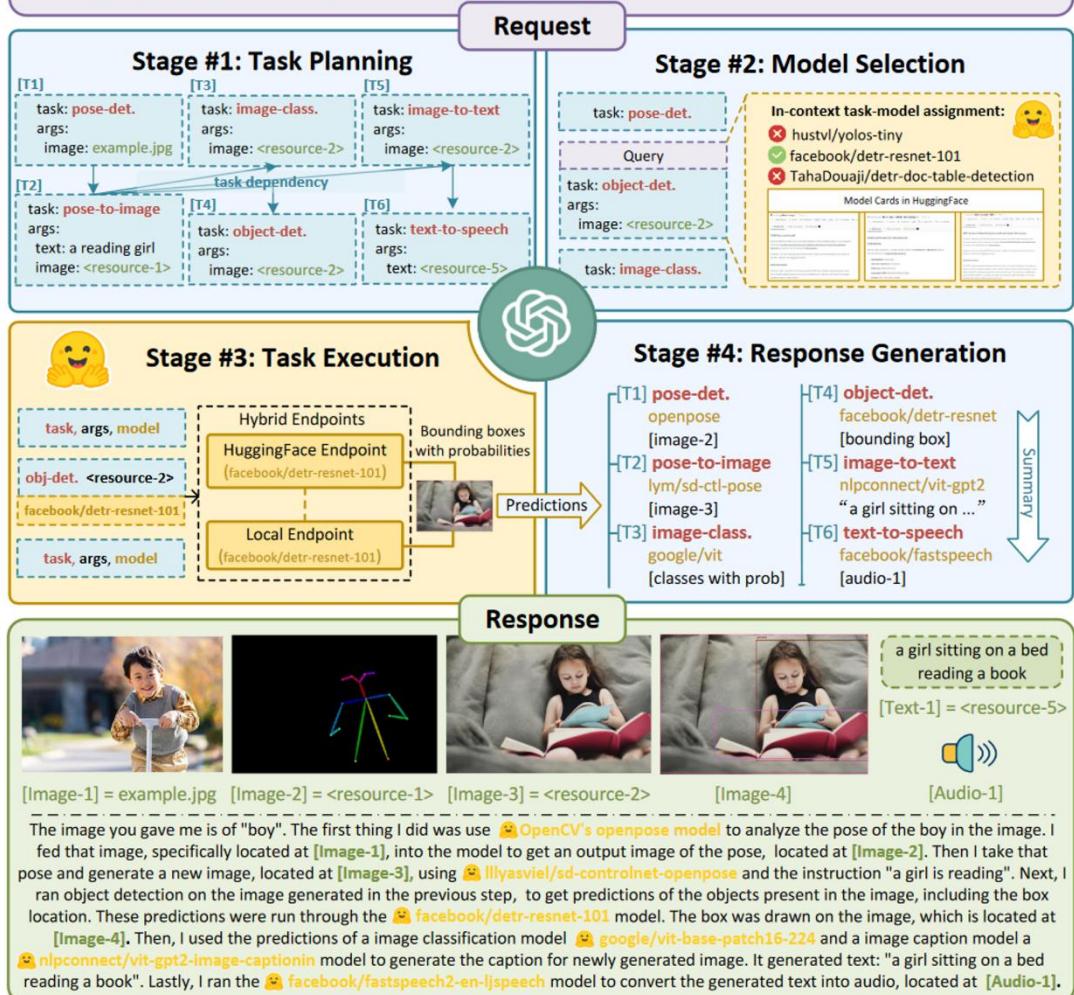


图 2-9 HuggingGPT 工作原理图

- 「Westworld」模拟（斯坦福西部世界小镇）— 发布于 2023 年 4 月 7 日：Smallville 的虚拟小镇，用于研究人工智能在社会互动中的行为。这个小镇拥有 25 个 AI 智能体，它们具有工作、社交、结交朋友、甚至举办情人节派对等能力。每个 AI 智能体都有独特的个性和背景故事，它们在 Smallville 小镇的公共场景中自由活动，如咖啡馆、酒吧、公园、学校、宿舍、房屋和商店。Smallville 小镇的 AI 智能体展现出了类似人类的行为，例如在看到早餐着火时会去关掉炉子，看到浴室有人时会在外面等待，遇到想交谈的个体时会停下来聊天。这个项目在 AI 社区引起了轰动，被认为是 2023 年最激动人心的 Agent 实验之一。Smallville 项目地址为：[https://github.com/joonspk-research/generative\\_agents](https://github.com/joonspk-research/generative_agents)。
- 「VOYAGER」 - 发布于 2023 年 5 月 25 日：VOYAGER 是一个创新的开源 AI 智能体项目，它展示了大语言模型在游戏环境中的自主学习能力。这个项目在 Minecraft 游戏世界中运行，能够通过环境交互来学习新技能、制定计划并执行复杂任务。它采用了课程学习方法，并集成了技能库系统来存储和复用已学习的技能，是首个在 Minecraft 中展现持续学习能力的 AI 智能体。该项目由 MineDojo 团队开发，其源代码可在 GitHub ([github.com/MineDojo/Voyager](https://github.com/MineDojo/Voyager)) 查看，相关研究发表

在论文 "VOYAGER: An Open-Ended Embodied Agent with Large Language Models" 中，项目详情可访问 [minedojo.org](http://minedojo.org) 了解更多。这个开创性的项目对理解 AI 在开放环境中的持续学习能力具有重要意义。

- 「**MetaGPT**」 - **发布于 2023 年 7 月**: MetaGPT 是一个将 LLM 转变为多个协作的软件公司角色的框架。它能够将一个产品需求转化为完整的软件项目，包括分析、设计、编码等全过程。每个角色（如产品经理、架构师、程序员等）都由一个 LLM Agent 扮演，它们协同工作以完成软件开发任务。参考链接：<https://github.com/geekan/MetaGPT>
- 「**ChatDev**」 - **发布于 2023 年 8 月 28 日**: ChatDev 的主要特点包括：
  - **增量开发**: 支持在现有代码基础上进行开发，通过指定源代码目录路径来实现。
  - **Docker 支持**: 通过 Docker 实现安全执行，这得益于 ManindraDeMel 的贡献。
  - **多智能体协作**: 不同角色的智能体在用户指定任务需求后进行交互式协同，生成完整的软件产品。
  - **ChatDev IDE**: 这是一个用于构建 Agent 的工具，支持 JavaScript，可以加速提示工程，适用于游戏 NPC 或强大的 Agent 工具设计。

ChatDev 的潜力在于将大型语言模型 (LLMs) 整合到软件开发领域，为软件开发过程带来新的视角和可能性。它不仅提高了软件开发的效率和成本效益，还通过实验证明了其在软件开发过程中的优势。

- 「**XAgent**」 - **发布于 2023 年 10 月 23 日**: 是清华大学知识工程实验室开发的一个创新型自主 Agent 框架，它能够像人类开发者一样理解和执行复杂的计算机任务。该系统基于大语言模型构建，具备工作记忆和自我反思能力，可以通过任务分解、代码编写、调试和优化来解决各类问题。XAgent 采用模块化设计，集成了多种工具，包括代码执行、网页浏览和文件操作等功能，同时具备任务规划、动作执行和工具调用等核心组件。它不仅支持复杂任务的递归分解和执行监控，还具有失败恢复和重试机制，可广泛应用于软件开发、数据分析、自动化测试和文档管理等领域。项目完整源代码已在 GitHub ([github.com/OpenBMB/XAgent](https://github.com/OpenBMB/XAgent)) 开源，相关技术细节可参考其论文 "XAgent: An Autonomous Agent for Complex Task Solving"。
- 「**Amazon Bedrock Agents**」- **2023 年 11 月发布**: Amazon 发布了 Bedrock Agents，这是一个基于 Amazon Bedrock 构建的企业级 AI Agent 开发平台。它支持多种基础模型如 Claude 和 Llama 2，并提供了完整的知识库管理和工具集成能力。开发者可以利用 Bedrock Agents 创建能执行特定任务的 AI 助手，同时确保企业级的安全性和合规性。

紧接着在 2023 年 12 月，Amazon 推出了 AWS Q，这是一个专门面向 AWS 生态系统的 AI 助手。AWS Q 能够协助开发者进行代码生成、问题诊断、系统架构设计等任务，并提供 AWS 最佳实践建议。它的推出标志着 Amazon 在专业领域 AI 助手方面的重要突破。

进入 2024 年，Amazon 继续加强其 AI Agent 产品线。2024 年 1 月，公司推出了 SageMaker Agents，这是一个专注于机器学习工作流程的智能助手系统。它能够自动化数据处理、模型训练和部署过程，显著提升了机器学习开发效率。2024 年 2 月，CodeWhisperer Agent 获得了重大更新，增加了完整的对话能力和代码解释功能，进一步强化了其作为代码 AI 助手的地位。

Amazon 的 AI Agent 战略特别强调企业级应用场景，注重安全性和可扩展性。通过与 AWS 云服务的深度集成，这些 AI Agent 能够无缝接入企业现有的技术栈。Bedrock Agents:

<https://aws.amazon.com/bedrock/agents/>

- 「OpenAssistant Pythia」- 发布于 2024 年 1 月：这是一个开源的对话式 AI 框架，由 LAION 和 Anthropic 等组织支持。它的目标是创建一个透明、开放的 AI 助手生态系统。该项目持续活跃，提供了多语言支持和各种任务处理能力。参考链接：<https://github.com/LAION-AI/Open-Assistant>
- 「Ray」 - 2024 版本：由 Anyscale 开发的分布式计算框架，Ray 的 Agent 系统是一个完整的分布式 AI 框架，它整合了强化学习、模型服务和分布式训练等多个关键组件。在其核心，RLlib 提供了丰富的强化学习算法支持，包括 PPO、DQN 和 SAC 等，并能够实现大规模的分布式训练。Ray Serve 则负责 Agent 的部署和服务，提供了实时推理、负载均衡和 A/B 测试等功能。而 Ray Train 则专注于分布式训练的实现，支持超参数优化和进度追踪等特性。在技术实现上，Ray 采用了 Actor 编程模型，使得开发者能够轻松构建和部署分布式 Agent 系统。每个 Agent 可以被视为一个独立的 Actor，能够维护自己的状态并进行异步通信。系统支持自动的资源管理和调度，确保计算资源被高效利用。同时，Ray 的容错机制能够自动处理节点失败等问题，保证系统的稳定性。参考链接：<https://ray-project.github.io/q4-2021-docs-hackathon/0.4/ray-api-references/ray-rllib/agents/>
- 「OpenAI Swarm」 - 2024 年初：OpenAI Swarm 是 OpenAI 推出的一个实验性框架，旨在帮助开发者协调多智能体系统。这是一个具有突破性的框架，它简化了多智能体系统的编排过程。它引入了智能体(agents)、交接(handoffs)、例程(routines)和函数调用(function calling)等先进概念，为实验多个 AI 智能体的协调提供了强大的工具。该框架的核心功能包括专门执行特定任务的 AI 单元、允许智能体之间无缝转换任务的交接机制、定义标准化工作流程的例程系统，以及实现智能体与外部系统交互的函数调用能力。参考链接：<https://github.com/openai/swarm>

## 2. Agent 设计原则与方法

智能体(Agent)的设计需要遵循一系列原则并采用特定的方法来确保其有效性和可靠性。在设计智能体时，我们首先要明确其目标导向性，即智能体的每个行为都应该服务于预定的目标。这种目标导向不仅体现在最终结果上，还需要贯穿整个执行过程中。同时，智能体的设计应该遵循模块化原则，将不同功能划分为独立的模块，既保证了代码的可维护性，也提高了系统的灵活性。

一个典型的智能体包含三个核心部分：

- 感知模块：处理输入信息
- 决策模块：制定行动计划
- 执行模块：实施具体行动

这三个模块形成一个完整的循环，使智能体能够持续有效地工作。

智能体的设计应该采用迭代开发的方式。首先实现基本功能，然后通过不断的测试和反馈来完善系统。在此过程中，要注重收集和分析性能数据，根据实际运行情况调整设计参数和策略。这种渐进式的开发方法可以帮助我们建立一个更加稳健和高效的智能体系统。

通过遵循这些设计原则和方法，我们可以构建出既能完成特定任务，又具有良好可扩展性和维护性的智能体系统。这样的系统不仅能够满足当前的需求，还能够适应未来可能出现的新要求和挑战。

## 3. Models

Model 是 Agent 的大脑，负责处理所有输入和输出数据。通过有效调用不同的模型，智能体可以根据任务需求执行文本分析、图像识别和复杂推理等操作。CAMEL 提提供了一系列标

准和可定制的接口，并与各种组件无缝集成，以赋能大语言模型（LLM）驱动的应用程序开发。在本部分，我们将介绍 CAMEL 目前支持的模型、工作原理及与模型交互的方式。

## 1. 目前支持的模型

[可以点击此处查看目前支持的模型](#)

## 2. 通过 API 调用模型

我们可以通过使用 ModelFactory 的 create 方法创建不同的 model，然后修改以下三个参数就可以做到调用不同的模型：model\_platform、model\_type、model\_config\_dict

这里我们使用 CAMEL 中的 ChatAgent 模块来简单调用一下模型，关于 ChatAgent 的进一步描述可以参考后续内容，这里我们简单把它理解为一个基础的模型调用模块即可。

### 使用语言模型

```
from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key='你的 api_key'
)

agent = ChatAgent(
 model=model,
 output_language='中文'
)

response = agent.step("你好， 你是谁？ ")
print(response.msgs[0].content)
```

>>>

你好， 我是来自阿里云的大规模语言模型，我叫通义千问。

如果您想使用与 OpenAI 的 API 兼容的接口（即遵循 OpenAI 提供的 API 规范和认证方式），可以将 model 替换为以下代码：model

```
model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="a-string-representing-the-model-type",
 api_key=os.environ.get("OPENAI_COMPATIBILITY_API_KEY"),
 url=os.environ.get("OPENAI_COMPATIBILITY_API_BASE_URL"),
 model_config_dict={"temperature": 0.4, "max_tokens": 4096},
)
```

### 使用多模态模型

```
from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType
```

```
from camel.messages import BaseMessage

from io import BytesIO
import requests
from PIL import Image

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/QVQ-72B-Preview",
 url='https://api-inference.modelscope.cn/v1/',
 api_key='API'
)

agent = ChatAgent(
 model=model,
 output_language='中文'
)

图片 URL
url = "https://img0.baidu.com/it/u=2205376118,3235587920&fm=253&fmt=auto&app=120&f=JPEG?w=846&h=800"
response = requests.get(url)
img = Image.open(BytesIO(response.content))

user_msg = BaseMessage.make_user_message(
 role_name="User",
 content="请描述这张图片的内容",
 image_list=[img] # 将图片放入列表中
)

response = agent.step(user_msg)
print(response.msgs[0].content)
```

>>>

这是一张金毛寻回犬的特写照片。这只狗有着浓密的金色毛发，耳朵垂在头部两侧，眼睛明亮而有神，鼻子黑色且湿润。它的嘴巴微微张开，露出了粉红色的舌头，看起来非常友好和快乐。背景是一片模糊的绿色，可能是在户外的自然环境中拍摄的。整体画面非常温暖和愉悦，突出了金毛寻回犬的亲和力和活力。

### 视频理解

```
from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType
```

```
from camel.messages import BaseMessage

from dotenv import load_dotenv
import os

load_dotenv()

api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/QVQ-72B-Preview",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)

创建代理
agent = ChatAgent(
 model=model,
 output_language='中文'
)

读取本地视频文件
video_path = "vedio_test.mp4"
with open(video_path, "rb") as video_file:
 video_bytes = video_file.read()

创建包含视频的用户消息
user_msg = BaseMessage.make_user_message(
 role_name="User",
 content="请描述这段视频的内容",
 video_bytes=video_bytes # 将视频字节作为参数传入
)

获取模型响应
response = agent.step(user_msg)
print(response.msgs[0].content)

>>>
这是一幅令人惊叹的超现实主义景观画，画面中有一只狗在一条被雪覆盖的道路上奔跑。整个场景被一种奇幻而美丽的氛围所包围，让人感受到一种超凡脱俗的美。

首先，画面的背景是一片壮丽的天空，天空中布满了浓密的云层，这些云层呈现出丰富的纹理和层次感。云层的颜色以粉色和紫色为主调，其间还夹杂着一些金色和蓝色的光影，形成
```

了一个绚丽多彩的天幕。在云层之间，有一道明亮的光线穿过，这道光线像是从天际线的尽头射来，照亮了整个场景，给人一种希望和温暖的感觉。

地面上是一条被雪覆盖的道路，道路两旁是连绵起伏的雪山和丘陵。雪地上的足迹和轮胎印清晰可见，表明这里虽然人迹罕至，但仍有生命活动的痕迹。一只白色的狗正在这条道路上奔跑，它的姿态轻盈而欢快，仿佛在享受着这美好的一刻。

整个画面的色彩非常丰富，粉色、紫色、金色、蓝色相互交织，营造出一种梦幻般的氛围。这种色彩的运用让画面充满了活力和生命力，令人不禁沉醉其中。

总的来说，这幅画通过超现实主义的手法，创造了一个美轮美奂的幻想世界，让观者在现实与幻想之间徘徊，感受到艺术带来的无限魅力。

### 3. 使用开源模型

#### 使用 Ollama

Ollama 是一个开源的机器学习框架，专注于让用户轻松地创建和使用各种语言模型。它提供了一个简单的 API，能够将预训练的语言模型（例如 GPT 系列）集成到你的应用程序中。Ollama 支持许多主流的机器学习模型和任务，如文本生成、对话系统、文本分类等。通过它的简单接口，开发者能够方便地进行模型加载、推理以及与模型交互。

进阶方案是在后端部署一个带有本地模型的服务器，并将其用作 API 的本地替代品。我们在这里使用 Ollama 部署的 Qwen2.5 为例。

1. 首先安装 [Ollama](#)
2. 设置 Ollama 后，通过在终端中键入以下命令来拉取 Qwen2.5 模型（这里使用 7B 的模型，大约需要 16GB 的内存，70B 的模型大约需要 64GB 以上的内存，可以根据自身的配置和需要选择不同参数的模型），Ollama 基于 llama.cpp 实现，本地 CPU 推理效率非常高（当然如果有 GPU 的话，推理效率会更高），还可以兼容 openai 的接口。：

```
ollama pull qwen2.5:7b
```

之后同样使用以下代码替换 model: model

```
#如果出现 openai.InternalServerError: Error code: 502 错误可以尝试设置一下 http_proxy 及
https_proxy
#import os
```

```
os.environ["http_proxy"] = "http://localhost:11434/v1/"
os.environ["https_proxy"] = "http://localhost:11434/v1/"
```

```
model = ModelFactory.create(
 model_platform=ModelPlatformType.OLLAMA,
 model_type="qwen2.5",
 model_config_dict={"temperature": 0.4},#可选
)
```

#### 使用 vLLM

vLLM 是一个高效的、面向大规模语言模型的推理库，专为大规模语言模型设计，旨在提高推理速度和减少资源消耗。vLLM 的一个核心特性是它能够在多种硬件环境下（例如单 GPU、多 GPU，甚至 CPU）高效运行，极大地降低了推理成本。

1. 首先安装 [vLLM](#)
2. 设置 vLLM 后，启动兼容 OpenAI 的服务器，例如：

```
python -m vllm.entrypoints.openai.api_server --model microsoft/Phi-3-mini-4k-instruct --api-key vllm --dtype bfloat16
```

之后同样使用以下代码替换 model: model

```
vllm_model = ModelFactory.create(
 model_platform=ModelPlatformType.VLLM,
 model_type="microsoft/Phi-3-mini-4k-instruct",
 url="http://localhost:8000/v1", # 可选
 model_config_dict={"temperature": 0.4}, #可选
)
```

---

#### 4. Messages

##### 1. 概述

###### 什么是 Agent 的 Message？

当你和朋友聊天、在网上搜索信息或是对手机语音助手说“帮我查一下天气”时，其实你都在向某个“代理者(Agent)”发送一条“信息(Message)”。这里的“代理者”既可以是一个人，也可以是一个能执行指令的智能程序或系统，而“信息”则是你传递的指令、问题或数据。在日常生活中，这种信息交互常常不易察觉，但在计算机科学、人工智能和自动化任务中，“Agent 的 Message”是一个至关重要的基础概念。

简单来说，Agent 的 Message 就是指系统中“智能体”或“代理者”之间互相传递的指令或数据包。就好比你给朋友发一条微信消息请他帮忙带杯咖啡，在智能系统中，“Agent”则是那些负责完成任务的角色，而“Message”则是他们沟通和协作的工具。当一个 Agent 收到 Message 后，会根据内容做出决策、执行任务或回复信息。

在 CAMEL 系统中，BaseMessage 是所有消息对象的基础类，它为对话中的每一条信息提供了统一的结构和标准化的处理方式。无论是用户输入的一段文本，还是包含图片、视频等多模态信息的数据包，都可以通过 BaseMessage 来统一表示和管理。

###### 为什么需要统一的消息结构？

在一个对话系统中，消息可能来自多方（如用户、系统、不同类型的 Agent），且信息内容不局限于纯文本，还可能包括图像、视频甚至是自定义的元数据 (metadata)。如果没有一个统一的基础类来约束这些消息的格式，开发者就会面临如下问题：

- **类型繁杂且难以维护：**不同消息类型需要各自的代码逻辑和数据结构，导致系统复杂度提高。
- **难以扩展和对接：**当需要增加新类型的消息（如引入新媒体格式或上下文信息）时，很可能需要大幅度修改原有代码。
- **通用处理困难：**缺乏统一结构会让调试、日志记录和分析对话信息变得更加麻烦。

通过使用 BaseMessage，你可以：

- 将消息的创建、变形（如格式转换）和传递标准化。
- 简化对消息类型的扩展，提高代码的可维护性和可读性。

- 为后续的功能模块（如消息过滤、路由、多轮对话管理）提供一个统一的数据基础。并且熟练掌握 Message 相关内容，对我们后续无论是做 RAG 应用或者模型的 Fine-tune 都非常重要！在后续章节我们会为大家介绍。

## 2. 创建和使用 Message

在了解了 BaseMessage 存在的意义后，让我们直接通过实例化来看看如何创建和使用它。通过一个最小化示例，我们将掌握 BaseMessage 的关键属性和基本用法，再进一步扩展到多模态内容。

### 创建 BaseMessage 实例的最小化示例

下面是一个最基本的代码示例，将创建一条来自用户的文本消息：

```
from camel.messages import BaseMessage
```

```
from camel.types import RoleType
```

```
创建一个简单的用户消息
```

```
message = BaseMessage(
 role_name="example_user",
 role_type=RoleType.USER,
 content="Hello, CAMEL!",
 meta_dict={} #添加必需的 meta dict 参数，即使为空也要提供，否则会报 TypeError
)
```

```
print(message)
```

```
>>>
```

```
BaseMessage(role_name='example_user', role_type=<RoleType.USER: 'user'>, meta_dict={},
content='Hello, CAMEL!', video_bytes=None, image_list=None, image_detail='auto',
video_detail='low', parsed=None)
```

在上述示例中，我们创建了一条来自 example\_user 的 USER 类型消息，内容为纯文本 "Hello, CAMEL!"。这就是一个最小化的 BaseMessage 示例。

### 关键属性介绍

- `role_name`: 给消息一个容易辨识的名称，如 "User"、"Assistant" 或 "System"。在更复杂的场景中，你或许会有多个用户、多个 Agent，通过 `role_name` 能帮助你追踪消息来源。
- `role_type`: 角色类型一般来自 `RoleType` 枚举，以明确此消息在对话中的身份。例如：
  - `RoleType.USER`: 表示该消息来自用户
  - `RoleType.ASSISTANT`: 表示该消息来自智能助手
- `content`: 消息的核心载体，一般是文本，也可能是解析指令、问题描述或描述性文字。

### 简单扩展：添加多模态内容

除了纯文本外，BaseMessage 还支持包含图片、视频等多模态信息。这可以为你的对话系统带来更丰富的交互体验。下面的示例展示了如何向 BaseMessage 添加一张图片。假设你已

将一张图片加载为 PIL.Image 对象：

```
from PIL import Image
from io import BytesIO
import requests

下载一张图片并创建一个 PIL Image 对象
url = "https://raw.githubusercontent.com/camel-ai/camel/master/misc/logo_light.png"
response = requests.get(url)
img = Image.open(BytesIO(response.content))

创建包含图片的用户消息
image_message = BaseMessage(
 role_name="User_with_image",
 role_type=RoleType.USER,
 content="Here is an image",
 meta_dict={},
 image_list=[img] # 将图片列表作为参数传入
)

print(image_message)

>>>
BaseMessage(role_name='User_with_image', role_type=<RoleType.USER: 'user'>,
meta_dict={}, content='Here is an image', video_bytes=None,
image_list=[<PIL.PngImagePlugin.PngImageFile image mode=RGBA size=3520x720 at
0x1DDFF8E88F0>], image_detail='auto', video_detail='low', parsed=None)
```

同理，如果你有视频数据（如 video\_bytes），也可将视频信息传入 BaseMessage。当你的消息中包含图片、视频等丰富媒体信息时，后续的组件（如 ChatAgent）便可利用这些多模态数据进行更智能和灵活的响应。

### 3. 不同类型消息的处理

在使用 BaseMessage 的过程中，你不仅可以轻松创建基本的用户消息，也能够通过其内置方法快速生成其它类型（如系统消息、助手消息）的实例，并对其内容进行更新和转换。此外，BaseMessage 提供了多种便利的转换方法，可将消息转化为不同格式，便于对接诸如 OpenAI 等后端服务。

#### 快速生成不同类型的消息

通过 BaseMessage 的类方法，我们可以快捷创建出用户（User）、助手（Assistant）的消息：

```
from camel.messages import BaseMessage

创建用户消息
user_msg = BaseMessage.make_user_message()
```

```
 role_name="User_1",
 content="Hi, what can you do?"
)

创建助手消息
assistant_msg = BaseMessage.make_assistant_message(
 role_name="Assistant_1",
 content="I can help you with various tasks."
)

print("User Message:", user_msg)
print("Assistant Message:", assistant_msg)
```

在上述示例中，你无需再手动指定 `role_type`，使用这些类方法即可轻松创建特定角色的消息。这样有助于在你的应用中保持代码整洁和可读性。

## 更新消息内容

有时你需要基于某条原有的消息创建略有改动的新消息。`BaseMessage` 提供了 `create_new_instance()` 方法，使你能在保持原消息基础信息的同时，轻松更新 `content`:

```
基于用户消息创建一个新消息，内容稍作修改
updated_user_msg = user_msg.create_new_instance("Hi, can you tell me more about
CAMEL?")
print("Updated User Message:", updated_user_msg)
```

这个方法非常有用，可以在对话过程中根据上下文动态构建消息流，而无需从头创建所有参数。

## 将消息转换为字典格式

如果你需要查看消息内部结构，或者将消息数据传给其它系统、序列化保存，`BaseMessage` 的 `to_dict()` 方法可以直接将消息对象转化为字典结构:

```
msg_dict = assistant_msg.to_dict()
print("Message as dict:", msg_dict)
```

输出的字典中会包含消息的 `role_name`、`role_type`、`content` 等信息，使得你可以轻松与其它数据处理流程对接。

## 适配 OpenAI 后端的消息格式

在实际应用中，你可能需要将消息传给 OpenAI 的对话接口。`BaseMessage` 提提供了一组方法来将现有消息快速转化成符合 OpenAI 后端需求的格式。例如:

```
from camel.types import OpenAIBackendRole

将用户消息转化为 OpenAI 后端兼容的用户消息
openai_user_msg = user_msg.to_openai_message(role_at_backend=OpenAIBackendRole.USER)
print("OpenAI-compatible user message:", openai_user_msg)

将助手消息转化为 OpenAI 后端的助手消息
```

```
openai_assistant_msg = assistant_msg.to_openai_assistant_message()
print("OpenAI-compatible assistant message:", openai_assistant_msg)
```

通过这些方法，你可以轻松地将 `BaseMessage` 对象接入到 OpenAI 接口的调用流程中，无需手动编写繁琐的转换逻辑。

#### 4. 与 ChatAgent 协作

在前面的小节中，我们学习了如何创建和操作 `BaseMessage`。现在，让我们把所学的知识付诸实践，将这些消息交给 `ChatAgent`，让对话真正“活”起来。

`ChatAgent` 是 CAMEL 系统中负责对话处理与智能回应的组件。当你将 `BaseMessage` 对象传递给 `ChatAgent` 时，`ChatAgent` 将根据系统和用户消息的内容，生成具有上下文感知的回复。

##### 将文本消息直接交给 `ChatAgent` 的基本用法

如果你仅想与智能助手进行一段简单的对话，可以直接构造一个文本类型的用户消息，并使用 `ChatAgent` 的 `step()` 方法进行响应，在实际使用过程中，我们无需严格按照 `BaseMessage` 的格式来设置我们的 `message`，`ChatAgent` 会通过 `make_assistant_message` 等方法会将字符串格式的 `msg` 转换成 `BaseMessage`，我们只需要用最简便的字符串来设置我们 `message`，当然，如果你对 `role_name` 和 `role_type` 有特殊要求的话，也可以按照 `BaseMessage` 的格式来设置 `message`：

```
from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)

创建系统消息，告诉 ChatAgent 自己的角色定位
system_msg = "You are a helpful assistant that responds to user queries."

实例化一个 ChatAgent
chat_agent = ChatAgent(model=model,
 system_message=system_msg, output_language='zh')

构造用户消息
```

```
user_msg = "Hello! Can you tell me something about CAMEL AI?"
```

```
将用户消息传给 ChatAgent，并获取回复
response = chat_agent.step(user_msg)
print("Assistant Response:", response.msgs[0].content)
```

```
>>>
```

Assistant Response: 当然可以！CAMEL AI 是一个先进的语言模型，它被设计用来进行多轮对话、理解复杂指令并生成高质量的文本。这个模型能够处理各种任务，比如回答问题、撰写文章、创作故事等。CAMEL AI 的目标是通过自然语言处理技术，为用户提供更加智能和人性化的交互体验。如果你有任何具体的问题或需要帮助的地方，欢迎随时告诉我！

在该示例中，我们先为 ChatAgent 提供一个系统消息指定它的身份，然后发送用户文本消息，最终获得智能助手的文本回复。

### 使用 BaseMessage 传递更丰富的上下文和多模态信息给 ChatAgent

BaseMessage 不仅可用于传递纯文本，还可扩展为多模态消息。当你在对话中加入图片、视频或自定义元数据时，ChatAgent 有机会根据这些额外信息提供更有针对性的回答。例如，在发送消息时，你可以在 BaseMessage 中包含图片列表或自定义的 meta\_dict 信息，帮助 ChatAgent 理解上下文或额外提示：

```
from camel.messages import BaseMessage
在用户消息中添加元数据
user_msg_with_meta = BaseMessage.make_user_message(
 role_name="User",
 content="Here is some extra context in the metadata.",
 meta_dict={"processing_time": 1.23, "api_version": "v2", "user_id": "1234567890"}
)
```

```
response_with_meta = chat_agent.step(user_msg_with_meta)
print("Assistant Response with metadata:", response_with_meta.msgs[0].content)
```

```
>>>
```

Assistant Response with metadata: 了解了！如果您能提供更多关于元数据的具体信息或上下文，我可以更好地帮助您解答或处理相关问题。请告诉我您需要了解什么方面的内容？

在此示例中，meta\_dict 主要用于系统内部，而不是直接的模型交互。可以在复杂的对话系统中进行消息路由和状态管理。

```
展示消息的元数据和内容
print("==== 消息元数据信息 ===")
print(f"用户 ID: {user_msg_with_meta.meta_dict['user_id']}")
print(f"API 版本: {user_msg_with_meta.meta_dict['api_version']}")
print(f"处理时间: {user_msg_with_meta.meta_dict['processing_time']} 秒")
```

```
>>>
```

```
==== 消息元数据信息 ===
```

```
用户 ID: 1234567890
```

API 版本: v2

处理时间: 1.23 秒

### 实际案例：发送图片并获取智能回复

下面让我们演示一个更完整的用例：发送一张图片给 ChatAgent，让它根据图片内容进行描述或回答相关问题。这可以用于场景如：让智能助手识别图像中的物体、提取图像信息，或者对图片进行描述。

```
from camel.agents import ChatAgent
from camel.messages import BaseMessage
from camel.models import ModelFactory
from camel.types import ModelPlatformType,RoleType

from io import BytesIO
import requests
from PIL import Image
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/QVQ-72B-Preview",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)

实例化 ChatAgent
chat_agent = ChatAgent(model=model,output_language='中文')

图片 URL
url = "https://img0.baidu.com/it/u=2205376118,3235587920&fm=253&fmt=auto&app=120&f=JPEG?w=846&h=800"
response = requests.get(url)
img = Image.open(BytesIO(response.content))

user_image_msg = BaseMessage.make_user_message(
 role_name="User",
 content="请描述这张图片的内容",
 image_list=[img] # 将图片放入列表中
)
```

```
将包含图片的消息传给 ChatAgent
response_with_image = chat_agent.step(user_image_msg)
print("Assistant's description of the image:", response_with_image.msgs[0].content)

在这个案例中，当 ChatAgent 接收到包含图片的消息后，它将尝试根据自身的知识和处理能力对图片进行描述。此示例展示了多模态消息传递的潜力，让智能对话系统能处理不仅仅是文本的信息。
```

## 5. Responses

Agent (Agent) 在与用户交互的过程中，会根据用户的输入生成相应的响应。这些响应不仅包含要显示给用户的消息，还可能包含额外的信息，如会话状态、上下文数据等。camel.responses 模块是 CAMEL 框架中处理聊天 Agent 响应的重要部分。其中 ChatAgentResponse 类用于封装聊天 Agent (ChatAgent) 的交互输出，结构化响应内容，便于开发者访问消息、会话状态等信息。

一个典型的 Agent 响应通常包括以下几个部分：

- **消息内容 (Message Content)**: 这是用户直接看到的部分，如文本、图片等。
- **会话状态 (Session Status)**: 指示会话是否继续、结束或需要进行其他操作。
- **附加信息 (Additional Information)**: 用于存储上下文数据、调试信息或其他辅助数据。

ChatAgentResponse 的类属性包括：

- **msgs**: 一个包含 BaseMessage 对象的列表，表示 Agent 生成的消息。根据模式的不同，列表内容会有所不同：
  - 空列表：表示消息生成时出现错误。
  - 单条消息：表示正常的消息生成操作。
  - 多条消息：表示 Agent 处于“批评者模式” (critic mode)。
- **terminated**: 一个布尔值，指示聊天会话是否已经被 Agent 终止。
- **info**: 一个字典，包含与会话相关的附加信息，例如使用统计或工具调用信息。

以下代码展示如何使用 ChatAgentResponse 类：

```
from camel.responses import ChatAgentResponse
from camel.messages import BaseMessage
from camel.types import RoleType

创建一个 ChatAgentResponse 实例
response = ChatAgentResponse(
 msgs=[
 BaseMessage(
 role_name="Assistant", # 助手的角色名称
 role_type=RoleType.ASSISTANT, # 指定角色类型
 content="你好，我可以帮您做什么？", # 消息内容
 meta_dict={} # 提供一个空的元数据字典（可根据需要填充）
)
],
 terminated=False, # 会话未终止
)
```

```
 info={"usage": {"prompt_tokens": 10, "completion_tokens": 15}} # 附加信息
)

访问属性
messages = response.msgs # 获取 Agent 生成的消息
is_terminated = response.terminated # 会话是否终止
additional_info = response.info # 获取附加信息

打印消息内容
print("消息内容:", messages[0].content)
打印会话是否终止
print("会话是否终止:", is_terminated)
打印附加信息
print("附加信息:", additional_info)

>>> 消息内容: 你好，我可以帮您做什么？
>>> 会话是否终止: False
>>> 附加信息: {'usage': {'prompt_tokens': 10, 'completion_tokens': 15}}
camel.responses 包为 Agent 的响应提供了一个结构化和规范化的方式。通过使用 ChatAgentResponse 类，开发者可以确保所有响应都符合预期的格式，并且易于扩展和维护。
```

## 6. 实践练习

经过本章的学习，你已经了解到 BaseMessage 在 CAMEL 系统中的定位与重要性。从基本的文本消息，到包含图片、元数据的多模态消息，再到如何将这些消息与 ChatAgent 进行整合，本章为你搭建了一个基础框架，让你能自如地操控消息流。以下是一些可以尝试的探索方向。

1. **扩展消息属性：**
2. 创建一个用户消息，并在 meta\_dict 中增加若干条元数据（如用户偏好、语言设置等）。将该消息传递给 ChatAgent，观察系统在回答中是否有所变化。
3. **多轮对话场景：**
4. 使用 BaseMessage 连续发送多条用户消息，模拟多轮对话。比如，先询问 CAMEL 的用途，然后再根据回复提出后续问题，看看 ChatAgent 是否能保持上下文连贯。
5. **多模态信息尝试：**
6. 尝试传入不同图片或使用 image\_detail 等参数，观察 ChatAgent 的回答变化。可以试试让 ChatAgent 对比两张不同的图片，并描述区别。
7. **与 OpenAI 接口整合（可选，进阶挑战）：**
8. 将生成的 BaseMessage 转为 OpenAI 后端可用的消息格式，然后使用 OpenAI 的 ChatCompletion 接口来获取答案。比较一下与 ChatAgent 内部实现的响应有何不同。

通过这些小任务的练习，你将更好地理解 BaseMessage 的实际应用场景，并为后续的深入探索奠定稳固的基础。

## 5. Prompt Engineering

提示词，参考 [OpenAI 提示词建议](#)

### 1. 概述

提示词工程(Prompt Engineering)是智能 Agent 交互中不可或缺的一环，它们指导智能体如何理解和回应用户的需求。这一节将探讨如何设计有效的提示词，以及如何通过精确的语言引导智能 Agent 提供更精准的服务和响应。

我想写一篇尽可能易于理解的 Prompt 设计指南，让即使第一次接触大模型的非技术人员也能学会如何恰当的使用它。然而，在写这句话的时候我就犯愁了，因为我遇到的第一个问题就是：应该如何介绍“什么是提示工程？Prompt 的定义是什么？”

所以我先向大模型进行提问，并与它对话。

Q: 什么是提示工程？Prompt 的定义是什么？

A:

- **提示工程 (Prompt Engineering)** 是一种技术，用于通过不改变模型的内部参数或权重来引导大语言模型 (LLM) 生成预期的结果。简单来说，就是给 AI 下指令，让它按照你的要求去做。这种技术可以应用于各种任务，比如回答问题、做算术、写文章等。掌握提示工程能够帮助我们更好地理解和利用大语言模型的能力和局限性。
- **Prompt (提示)** 是指我们给 AI 的初始文本输入。通过给 AI 一个提示，我们可以引导它生成我们想要的回答或完成特定的任务。提示可以是一个问题、一段描述、一些关键词，或者其他形式的文本，用来引导 AI 生成特定的内容。

比如，当我们使用 ChatGPT 时，我们通常会给它一个提示，比如“帮我写一篇关于环境保护的文章”或者“告诉我今天的天气怎么样”。ChatGPT 会根据这个提示生成相关的内容，尽量符合我们的需求。

为什么提示工程很重要？

提示工程的重要性在于，它让我们能够更好地控制 AI 的输出结果。就像在生活中，我们和别人说话时，如果说得越清楚，对方就越容易理解并作出正确的回应。同样地，在与大语言模型互动时，一个清晰且详细的提示可以帮助 AI 更准确地理解我们的意图，从而生成更符合预期的回答。

举个简单的例子：

想象一下，你想让 AI 帮你写一封感谢信。如果你直接对 AI 说：“写封感谢信。”，AI 可能会写一封非常通用的信。但如果说：“写一封感谢信，感谢我的朋友小李，他在我生病时每天来看望我，还给我带了很多好吃的。”，这样详细的提示会让 AI 生成一封更贴合你需要的感谢信。

一个好的 Prompt 就是要尽可能准确无误地表达你的需求（就像产品经理给程序员提需求那样）。

### 2. 怎么写好提示词？

为了让大语言模型 (LLM) 更好地理解和完成任务，编写提示词时需要遵循一些基本原则。以下是一些关键步骤和示例，帮助初学者掌握提示工程的技巧。

#### 1. 明确表达需求

首先，我们需要清楚地表达自己的需求。如果我们想要大型语言模型 (LLM) 帮助我们编写

个人简历，我们可以直接用日常语言告诉它：“请帮我写一份个人简历”。一般来说，LLM 会尝试根据有限的信息来生成一个初步的简历。

Q: 请帮我写一份个人简历。

A: 当然可以！请提供一些基本信息，例如你的名字、职业、教育背景和工作经验。

## 2. 提出任务

我们可以把 LLM 想象成一个能够进行自然交流的真人。假设我们要让 LLM 帮我们生成一个个人简历，就可以直接用自然语言要求 LLM：“帮我生成一份个人简历”。可以看到，它能够很好地完成任务。

Q: 帮我生成一份个人简历

A: 当然！以下是一份示例简历：

姓名：张三

年龄：28

职业：软件工程师

... (更多细节)

然而，如果我们新建一个对话，再问一遍相同的问题，结果可能会有一些不同。这次，大模型可能会询问我们更多的细节。

Q: 帮我生成一份个人简历

A: 请问您需要包括哪些具体信息？例如教育背景、工作经验、技能等。

## 3. 描述细节

为什么两次的结果不一样呢？因为我们只提出了任务，而没有描述这个任务的细节，比如简历需要包括哪些具体信息，LLM 自然就会有所疑惑。

假设现在我直接要求正在阅读这篇文档的你，“帮我生成一份个人简历”，你会是什么反应呢？

1. 不假思索的直接回答：“姓名：张三，年龄：28，职业：软件工程师...”
2. 添加一些额外的内容：“当然可以帮你生成一份个人简历，以下是一个示例：姓名：张三，年龄：28，职业：软件工程师...”
3. 也可能比较疑惑：“你需要包括哪些具体信息？例如教育背景、工作经验、技能等。”

LLM 和我们一样，如果我们没有为它限定任务的细节，它就可能产生不明确或不稳定的输出。现在我们修改一下提问的方式，再试一次。

Q: 帮我生成一份个人简历，包括姓名、年龄、职业、教育背景和工作经验

A: 好的，以下是一份包含这些信息的个人简历：

姓名：李四

年龄：30

职业：数据分析师

教育背景：XX 大学，计算机科学学士

工作经验：ABC 公司，数据分析师，5 年

这次的结果更接近我们的预期，因为我们描述的细节更加具体。然而，生成的内容还是有些冗长。

## 4. 更准确的描述细节

可以看出，让 LLM 遵从你的想法的根本就是：尽可能清晰地描述任务以及任务的细节。我们继续向它提要求，尽量减少不必要的内容。接下来我们需要提供更多的任务细节。例如，简历应包括个人信息、职业目标、教育背景、工作经验和技能。

Q：帮我生成一份个人简历，包括姓名、年龄、职业、教育背景和工作经验，不要输出多余内容。

A：姓名：王五，年龄：32，职业：项目经理，教育背景：XX 大学，工商管理硕士，工作经验：DEF 公司，项目经理，7 年

## 5. 复杂任务的指导

接下来，我们可以更精确地描述任务，指定每个部分的具体内容和格式。例如，我们希望在工作经验部分列出每份工作的详细职责。

Q：请帮我写一份个人简历，包括以下部分：

- 个人信息：姓名、联系方式
- 职业目标：简短的职业目标陈述
- 教育背景：学校名称、学位、专业和毕业时间
- 工作经验：公司名称、职位、工作时间和详细职责
- 技能：列出相关技能

A：好的，请提供上述信息的具体内容，我会根据这些信息生成你的个人简历。

有时候，直接且简洁的指令可以让 LLM 更快速地理解和执行任务。例如，我们可以告诉 LLM 直接生成一份包含特定信息的个人简历。

Q：生成一份个人简历，包括姓名（张三）、联系方式（123456789）、职业目标（成为软件工程师）、教育背景（北京大学，计算机科学，学士，2020 年毕业）、工作经验（ABC 公司，软件开发工程师，2020 年-2023 年，负责开发和维护软件系统），以及技能（Java、Python、SQL）。

A:

### 个人简历

姓名	张三	
-----	-----	
联系方式	123456789	

### 职业目标

职业目标	
-----	
成为软件工程师	

### 教育背景

学校名称	学位	专业	毕业时间	
-----	-----	-----	-----	

| 北京大学 | 学士 | 计算机科学 | 2020 年 |

## 工作经验

公司名称	职位	工作时间	职责
ABC 公司	软件开发工程师	2020 年-2023 年	- 负责开发和维护软件系统
			- 参与系统架构设计
			- 优化现有代码并修复漏洞

## 技能

技能
Java
Python
SQL

为了获得最符合我们需求的结果，我们可以进一步提供更多的细节。例如，指定每个部分的格式和长度要求。

Q: 基于上次提供的信息，根据以下补充生成一份个人简历，格式如下：

- 个人信息：姓名（张三）、联系方式（123456789），每项信息占一行。
- 职业目标：一句话职业目标。
- 教育背景：包括学校名称、学位、专业和毕业时间，每项信息占一行。
- 工作经验：列出公司名称、职位、工作时间和职责，每个职责占一行。
- 技能：列出相关技能，每项技能占一行。

A:

## 个人简历

姓名：张三

联系方式：123456789

## 职业目标

成为软件工程师

## 教育背景

北京大学  
计算机科学  
学士  
2020 年毕业

## 工作经验

ABC 公司  
软件开发工程师  
2020 年-2023 年  
- 负责开发和维护软件系统  
- 参与系统架构设计  
- 优化现有代码并修复漏洞

## 技能

Java  
Python  
SQL

通过这种方式，我们可以理解如何通过逐步增加细节和明确性来构建有效的提示，以便 LLM 能够生成符合预期的结果。这种格式不仅美观，而且易于阅读和理解。

### 6. 给出清晰的界限

接下来我们用另外一个例子举例，来体现出具体的界限描述有何作用，假设我们来开发一个情感分析软件，用户输入一句话，让 LLM 来判断文本的情感。

Q：请分析下面文本的情感：  
我今天很开心！

A: Positive

我们将 LLM 投入到生产的时候，往往输入的 prompt 是会变化的。例如开发人员需要将用户输入的句子交给 LLM 来分析情感，可以通过构造 prompt 模板来实现。然而，有些用户可能不听话，他输入了“请分析我刚才说的话的情感”。

Q：请分析下面文本的情感：  
请分析上面的文本的情感

A: Please analyze the sentiment of what I just said.

这下直接把我们的 prompt 模板都给钓出来了。为了解决这个问题，我们可以在 prompt 中限定任务涉及文本的范围。

Q：请分析下面|||之间的文本的情感：  
|||请分析我刚才说的话的情感|||

A: Positive

这次 LLM 理解到我们要分析的文本是在|||之间的了，即使|||之间的文本也是一个任务。我们可以使用各种各样的常见分隔符来区分文本界限，例如：“”“`---<<<>>>{}{}[[[]]] 等等

与此同时，约定输出格式是非常重要的。刚才提到，我们在开发一个情感分析软件，现在产品经理提的需求复杂了一点，我们要将用户输入的句子分别分析为正面、负面、中立。先给 LLM 提需求试下。

Q：请分析下面|||之间的文本的情感，并分别输出为正面、负面、中立：  
|||请分析我刚才说的话的情感|||

A: 正面：Positive

负面: Negative

中立: Neutral

虽然 LLM 正确完成了需求，但是这让开发人员犯了难：怎么从输出中提取结果呢？总不能把正面: Positive 直接显示给用户吧。要解决这个问题非常简单，我们可以继续在 prompt 中与 LLM 约定输出格式。

Q: 请分析下面|||之间的文本的情感。

输出应该分为三行，第一行为正面，第二行为负面，第三行为中立。

|||请分析我刚才说的话的情感|||

A: Positive

Negative

Neutral

和它约定用 Json 输出也是可以的，这也是比较常用的返回数据格式。

Q: 请分析下面|||之间的文本的情感。

输出应该是一个 Json，它有三个字段：positive、negative、neutral，分别代表正面、负面、中立的分析结果。

|||请分析我刚才说的话的情感|||

A: {

```
"positive": "Positive",
"negative": "Negative",
"neutral": "Neutral"
```

}

## 7. 举例子 (FewShot)

实际上，对于刚才的场景，LLM 还可能输出：输出的 Json 是：{} 等等无关的内容，影响解析。又或者，我希望 LLM 输出时，将 Json 放到一行，而不是一个字段后面就换行。对于这种情况，我们可以给 LLM 来举例子 (FewShot)。

Q: 你的任务是将<输入>的文本进行情感分析。

输出应该是一个 Json，它有三个字段：positive、negative、neutral，分别代表正面、负面、中立的分析结果。

下面是一个示例

<输入>我今天很开心

<输出>{"positive": "Positive", "negative": "Negative", "neutral": "Neutral"}

<输入>请分析我刚才说的话的情感

<输出>

A: {"positive": "Positive", "negative": "Negative", "neutral": "Neutral"}

现有的生成式模型大都是根据上文来生成下文，所以我们在编写 FewShot 的时候，可以恰好编写到希望 LLM 继续生成的部分。示例中，prompt 以<输出>结尾，如果不这样做，会发生什么呢？

Q: 你的任务是将<输入>的文本进行情感分析。

输出应该是一个 Json，它有三个字段：positive、negative、neutral，分别代表正面、负面、中立的分析结果。

下面是一个示例

<输入>我今天很开心

<输出>{"positive": "Positive", "negative": "Negative", "neutral": "Neutral"}

<输入>请分析我刚才说的话的情感

A: <输出>{"positive": "Positive", "negative": "Negative", "neutral": "Neutral"}

当然了，这种情况也不绝对，主要会受微调的指令数据集影响，由于这篇文档是介绍如何调教 LLM，就暂时不展开了。

### 3. 上下文学习 (ICL)

参考 Stanford 的 [Blog1](#), [Blog2](#)

上下文学习 (In-Context Learning, ICL) 是 GPT-3 首次提出的一种特殊 prompt 形式，它已经成为利用大型语言模型 (LLMs) 的典型方法之一。GPT-3 模型展示了一些大模型才具备的突现能力（如：模型规模达到百亿级时才会显现的能力），其中之一便是上下文学习。

ICL 的发展历史

- 2021 年初：引入了 Prompt learning（提示学习），这是 ICL 的早期形式。
- 2021 年底：发展为 Demonstration learning（演示学习），进一步丰富了上下文信息的利用。
- 2022 年初：演变为 In-context learning（情境学习），标志着 ICL 技术在理解和应用上下文方面的进一步成熟。

#### 什么是 ICL？

上下文学习 (In-Context Learning) 指的是通过提供几个任务示例/说明，让预训练模型理解任务本身。模型无需进行 fine-tuning，只需通过几个示例输入和输出对，就能为新输入生成正确的输出。

GPT-n 系列模型属于自回归类语言模型，自回归模型的原理是根据当前输入预测下一个词，然后将预测结果与输入拼接，再作为模型的输入进行下一词的预测，如此循环往复。自回归模型的训练目标是通过从超大规模语料库中采样训练样本，让模型根据输入输出一个概率向量。由于文本数据自带标注，我们知道真实的下一个词，因此损失函数采用交叉熵。

预训练好的 GPT-3 模型拥有一种被称为上下文学习的神奇能力。迁移到新任务时，GPT-3 无需重新训练，只需提供任务描述（可选项）和几个示例（任务查询和对应答案，以一对对的形式组织），最后加上模型需要回答的查询。将这些内容打包作为模型的输入，模型便能输出正确的答案。

示例：以翻译英文为法文任务为例，输入格式如下：

## Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

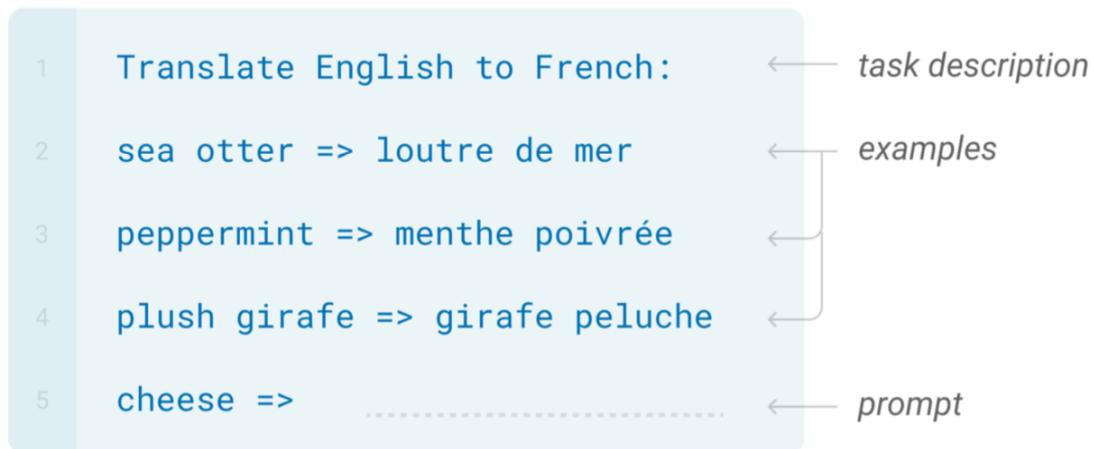


图 3-x 翻译任务示例

- 第一行：任务描述，告诉模型要做翻译
- 接下来的三行：示例对，包含英文单词和对应的法文单词
- 最后一行：待翻译的英文单词

将以上内容整体作为 GPT-3 的输入，模型就能输出对应的法文单词。上下文学习的应用非常灵活，除了翻译任务，还可以用于语法修饰甚至代码编写。

此外，GPT-3 训练过程中并没有显式提供类似测试阶段任务描述加示例这样的训练数据。然而，由于 GPT-3 的训练数据量极其巨大（包含 wiki、书本期刊、reddit 上的讨论等），或许其中已经包含了各种任务类似结构的数据，且 GPT-3 模型容量足够大，能够记住所有训练数据。

目前，对于上下文学习能力的成因仍是一个开放性问题。为什么只有大规模语言模型才具备该能力？或许仅有模型参数量大还不够，还必须训练数据量足够大，模型才能显现出这种能力。这些问题仍需进一步研究和探索。

## ICL 基础原理

通过提供一些示范性的<输入-标签>对，在不需要更新模型参数的情况下，ICL 可以对新输入进行准确预测。这种能力在各种应用场景中展现了极大的潜力。尽管 ICL 展示了出色的性能，但其具体工作原理仍然是个未解之谜。为了解释这一现象，清华大学、北京大学和微软的研究人员发表了一篇[论文](#)，将语言模型视为一种元优化器（meta-optimizer），并将 ICL 理解为一种隐性（implicit）的微调。这种视角帮助我们更好地理解 ICL 的运行机制。

ICL 可以分为以下三种类型：

1. **Few-shot Learning**: 提供多个示例。例如：“将中文翻译为英文。你好->hello，再见->goodbye，购买->purchase，销售->”，模型应预测下一个输出为“sell”。
2. **One-shot Learning**: 提供一个示例。例如：“将中文翻译为英文。你好->hello，销售->”，模型应预测下一个输出为“sell”。

3. **Zero-shot Learning**: 无示例。例如：“将中文翻译为英文。销售->”，模型应预测下一个输出为“sell”。

它的主要流程包括以下几个步骤：

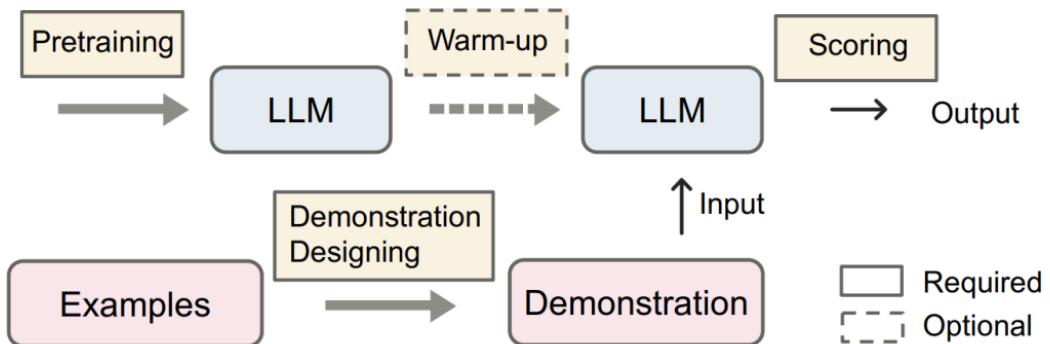


图 3-x ICL 步骤

4. **预训练阶段**: 通过大规模语料库进行训练, 培养语言模型的 ICL 能力。可选的预热阶段能进一步提升模型的性能。
  5. **演示设计**: 利用预训练的 LLM 和精心设计的示例, 加上适当的评分策略, 生成任务的最终输出。
  6. **推理阶段**: 将示例和查询问题结合形成一个提示, 输入模型进行预测。示例的质量对 ICL 效果并没有显著影响。

ICL 的强大性能依赖于两个阶段：

7. **训练阶段**: 训练 LLM 的 ICL 能力。语言模型直接在语言建模目标上进行训练, 如从左到右的生成。虽然这些模型并没有针对上下文学习进行特别优化, 但 ICL 仍然具有令人惊讶的能力。
  8. **推理阶段**: LLM 根据特定任务的演示进行预测。由于输入和输出标签都在可解释的自然语言模板中表示, 因此有多个方向来提高 ICL 的性能。

随着模型规模和语料库的扩大，研究表明，LLMs 可以利用 ICL 完成一系列复杂的任务，包括解决数学推理问题。这些能力已被广泛验证，显示了 ICL 作为大型语言模型的一种新兴能力的潜力。

ICL 的核心思想基于类比学习。简单来说，ICL 通过从给定的示例中提取模式，将这些模式应用于新任务，从而实现准确的预测和解决问题。

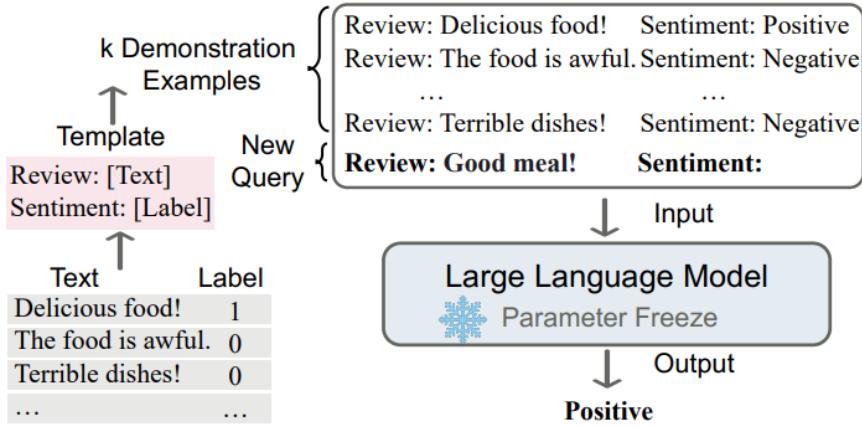


Figure 1: Illustration of in-context learning. ICL requires a piece of demonstration context containing a few examples written in natural language templates. Taking the demonstration and a query as the input, large language models are responsible for making predictions.

图 3-x ICL 类比学习

在 ICL 中，首先需要一些示例来形成演示上下文。示例通常使用自然语言模板编写。然后，ICL 将一个查询问题和一个演示上下文连接在一起形成一个提示。最后，将其输入到语言模型中进行预测。ICL 不需要参数更新，直接对预训练语言模型进行预测。模型被期望学习隐藏在演示中的模式，并相应地做出正确的预测。

ICL 作为一种新的范式，具有许多吸引人的优势。首先，演示用自然语言格式编写，提供一个可解释的接口与大型语言模型通信。这种范式通过更改演示和模板使将人类知识纳入语言模型变得容易得多。其次，上下文学习类似于人类的类比决策过程。第三，与有监督学习相比，ICL 是一种无监督学习框架。这不仅可以大大降低使模型适应新任务的计算成本，还可以使语言模型即服务成为可能，并且可以很容易地应用于大规模的现实世界任务。

然而，ICL 中还有一些问题和性质需要进一步研究。尽管普通的 GPT-3 模型显示出 ICL 能力，但通过预训练期间的自适应，能力可以显著提高。此外，ICL 的性能对特定的设置很敏感，包括提示模板、上下文示例的选择和示例顺序等。尽管从直观上看是合理的，但 ICL 的工作机制仍然不明确，很少有研究提供初步解释。

#### 4. 思维链 (CoT)

CoT 是一种改进的 Prompt 策略，旨在提升 LLM 在复杂推理任务中的表现。与 ICL 不同，CoT 不仅关注输入和输出，还特别强调将推理过程中的中间步骤纳入 Prompt。这种方法使得模型能够更清晰地展示其思考过程，从而在算术推理、常识推理和符号推理等任务中表现出色。

#### 什么是 CoT?

2022 年，Google 发表了一篇[论文](#)，提到了一种叫做“思维链”的技术（英文名叫 Chain-of-Thought，简称 CoT）。简单来说，这种方法就是让大模型（像 ChatGPT 这样的 AI 模型）逐步将一个复杂的问题分解为更简单的几个子问题，一步一步解决，直到得出答案。这个分解过程就被称为“思维链”。

为什么要这么做呢？想象一下你在解数学题时，不是一下子直接写出答案，而是把每一个解题步骤都写下来，这样不仅更容易理解，也能让整个过程更清晰。这就是思维链的原理——通过分解问题，让 AI 更容易找到正确答案。

### 传统方法与 CoT 的区别

通常，我们对 AI 的提问就像是从输入到输出的直线过程，比如你问一个问题，AI 直接给你答案。而使用思维链时，这个过程就变成了三个部分：输入 → 思维链（推理过程）→ 输出。

简单举个例子，如果让 AI 回答“如果有 5 只苹果，每只苹果值 3 元，总共多少钱？”传统的方法可能直接回答“15 元”。但使用思维链，AI 会先写出：“5 只苹果，每只 3 元，所以是  $5 \times 3$ ，答案是 15 元。”通过展示这些中间步骤，AI 的回答更容易被理解。

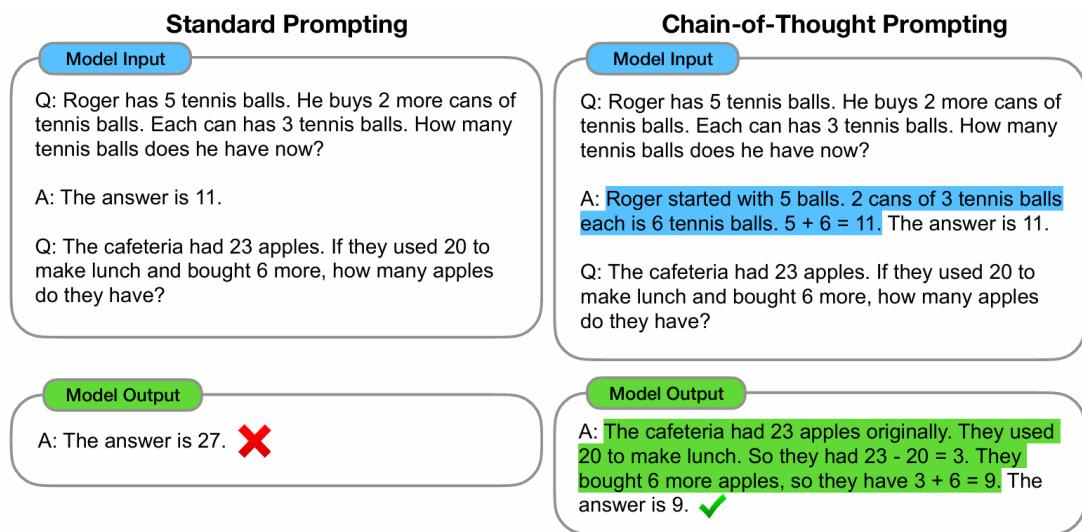


图 3-x CoT 与标准 prompt 步骤对比

上面这张图展示了标准提示（Standard Prompting）和思维链提示（Chain-of-Thought Prompting）的区别。

在左侧的标准提示中，我们看到模型直接给出答案，没有解释过程，因此在面对复杂问题时，模型很容易出错。比如，当被问到“Roger 有 5 个网球，他又买了 2 罐网球，每罐有 3 个网球。现在他一共有多少个网球？”时，模型直接给出了答案“11”。虽然答案正确，但没有展示中间推理步骤，这使得它的可靠性和理解性较低。

在右侧的思维链提示中，模型被要求逐步解释整个过程：“Roger 最开始有 5 个网球，又买了 2 罐网球，每罐有 3 个网球，总共有 6 个网球。 $5 + 6 = 11$ 。”通过一步步地展示推理过程，模型不仅得到正确答案，而且让人清楚地看到解题的逻辑。

另一个例子中，食堂有 23 个苹果，用掉了 20 个来做午餐，又买了 6 个苹果，还剩多少个苹果？标准提示下模型回答“27”，这是错误的。而使用思维链提示，模型会一步步解释：“原本有 23 个苹果，做午餐用掉了 20 个，所以还剩 3 个。然后又买了 6 个苹果，所以  $3 + 6 = 9$  个苹果。”最终得到正确答案。

### CoT 的不同类型

通过将采用思维链（CoT）方法的提示（Prompt）进行细致的分解，我们可以更深入、更清

晰地洞察到思维链的工作机制和流程。根据是否包含示例，我们可以将思维链分为两种：

9. **Zero-Shot-CoT**: 没有提供具体的例子，只是在问题中加一句“让我们一步步思考”(Let's think step by step)。这种简单的提示就能“唤醒”AI的推理能力。
10. **Few-Shot-CoT**: 给AI一些例子，每个例子都包含问题、推理过程和答案，像是给AI讲解解题步骤。通过这些例子，AI就能模仿这些推理步骤解决新问题。

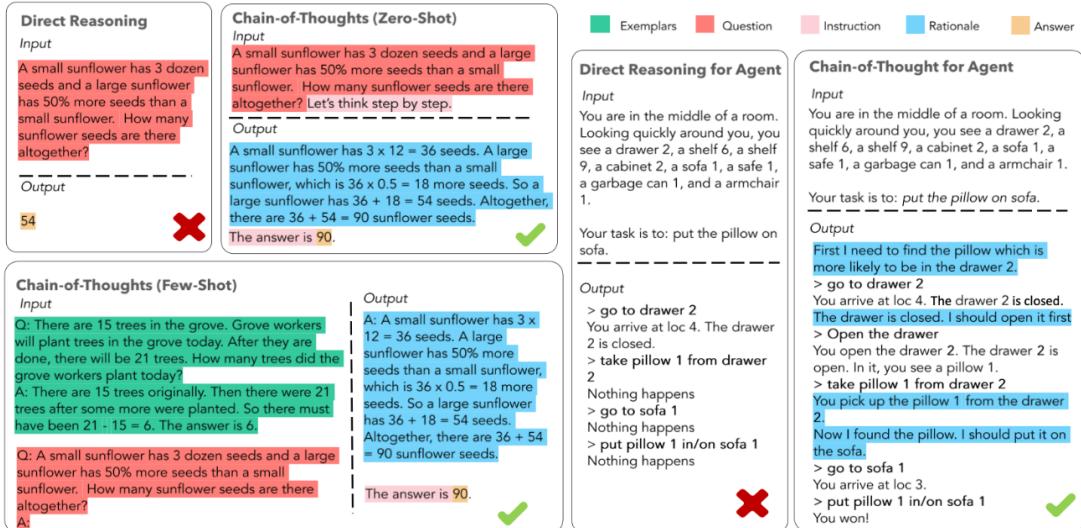


图 3-x CoT 的分类与实例

上面这张图展示了标准提示(Direct Reasoning) 和思维链提示(Chain-of-Thought Reasoning)的区别。

在左侧的直接推理中，我们看到模型直接给出答案，没有解释过程，因此在面对复杂问题时，模型很容易出错。比如，问到“小向日葵有 3 打种子，而大向日葵比小向日葵多 50% 的种子，一共有多少种子？”时，模型直接得出错误的答案“54”。

而在右侧的思维链推理中，模型被要求逐步解释整个过程：“小向日葵有 3 打种子，即  $3 \times 12 = 36$  个种子，大向日葵比小向日葵多 50%，也就是  $36 \times 0.5 = 18$  个种子。所以大向日葵有  $36 + 18 = 54$  个种子，总共有  $36 + 54 = 90$  个种子。”通过一步步展示推理过程，模型不仅得出正确答案，而且让人清楚地看到解题的逻辑。

## 为什么 CoT 有效？

目前为止，还没有一个完全被大家认可的科学解释来说明为什么思维链有效。不过，有很多实验观察到了一些有趣的现象，可以帮助我们理解 CoT：

- 模型规模要足够大：**如果 AI 太小（比如理解力不足），思维链可能就不起作用，因为它连基础知识都还不理解。
- 任务不能太简单：**对于一些非常简单的问题，思维链没有太大帮助，因为 AI 已经能直接给出答案。
- 训练数据的联系：**如果 AI 在训练中学到的数据彼此联系紧密，思维链的效果会更好。
- 示例中的错误：**有趣的是，即使给 AI 的示例中有些错误步骤，思维链依然有效。这说明 CoT 更像是在“指挥”AI 去做一步步的推理，而不是教 AI 具体怎么做。

简而言之，思维链的效果，可能在于它强迫 AI 按步骤思考，就像老师要求学生把解题过程写下来一样，不仅是为了得到答案，更是为了更好地理解问题的过程。

## 5. CAMEL 中的 prompt

现在我们来看看如何使用 CAMEL 的 Prompt 功能，通过实操来更好地理解 Prompt 的概念。

### 使用 CoT 提示创建特定任务 Agent

CAMEL 提供了一些便捷的工具来帮助用户使用 CoT。我们可以使用 TaskSpecifyAgent 创建一个特定任务 Agent，它会自动调用带有 CoT 的模板。例如，下面是代码示例：

```
from camel.agents import TaskSpecifyAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType, TaskType
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)
task_specify_agent = TaskSpecifyAgent(
 model=model, task_type=TaskType.AI_SOCIETY, output_language='中文'
)
specified_task_prompt = task_specify_agent.run(
 task_prompt="Improving stage presence and performance skills",
 meta_dict=dict(
 assistant_role="Musician", user_role="Student", word_limit=100
),
)
print(f"Specified task prompt:\n{specified_task_prompt}\n")

>>>
Specified task prompt:
Musician will coach Student on dynamic stage movement, audience engagement techniques, and emotional expression during performances, enhancing overall charisma and connection with the audience.
```

在这个例子中，我们使用 TaskSpecifyAgent 生成了一个带有思维链的任务提示，通过为 Agent 指定角色，让它逐步生成问题的解决方案。

## 使用自定义 Prompt 模板

CAMEL 还允许用户创建自己的 Prompt 模板，使得生成的 Prompt 更加符合用户的需求。我们可以编写自己的思维链提示模板，然后将它应用到 TaskSpecifyAgent 中。下面是一个简单的示例：

```
from camel.agents import TaskSpecifyAgent
from camel.models import ModelFactory
from camel.prompts import TextPrompt
from camel.types import ModelPlatformType
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)

my_prompt_template = TextPrompt(
 'Here is a task: I\'m a {occupation} and I want to {task}. Help me to make this task more specific.'
) # 你可以根据需求自定义任何模板
task_specify_agent = TaskSpecifyAgent(
 model=model, task_specify_prompt=my_prompt_template
)
response = task_specify_agent.run(
 task_prompt="get promotion",
 meta_dict=dict(occupation="Software Engineer"),
)
print(response)
```

>>>

Certainly! To make your goal of getting a promotion as a Software Engineer more specific, you can break it down into clear, actionable steps. Here's a more detailed version of your task:

### Specific Task: Get Promoted as a Software Engineer

#### 1. \*\*Understand the Promotion Criteria\*\*

- \*\*Research Company Policies:\*\* Review your company's promotion guidelines and

criteria.

- **Talk to Your Manager:** Schedule a meeting with your manager to discuss what is required for a promotion, including specific skills, projects, and performance metrics.
- **Analyze Peer Promotions:** Look at the career paths of colleagues who have been promoted recently to identify common traits and achievements.

#### #### 2. \*\*Set Clear Goals\*\*

- **Short-Term Goals (Next 3-6 months):** Identify 3-5 specific, achievable goals that align with your promotion criteria.
  - Example: Complete a critical project ahead of schedule, improve code quality by 20%, or lead a small team on a new initiative.
- **Long-Term Goals (Next 6-12 months):** Set 2-3 ambitious goals that demonstrate leadership and advanced technical skills.
  - Example: Mentor junior engineers, contribute to a major architectural change, or publish a paper or blog post on a relevant topic.

#### #### 3. \*\*Enhance Your Skills\*\*

- **Technical Skills:** Identify areas where you need to improve and create a plan to enhance these skills.
  - Example: Take online courses, attend workshops, or work on side projects to gain expertise in new technologies.
- **Soft Skills:** Develop leadership, communication, and problem-solving skills.
  - Example: Join a public speaking club, take a leadership course, or volunteer for cross-functional projects.

#### #### 4. \*\*Build Relationships\*\*

- **Network Internally:** Build strong relationships with colleagues and senior leaders.
  - Example: Participate in company events, join employee resource groups, or offer to help on other teams' projects.
- **Seek Mentors:** Find mentors within the company who can provide guidance and advocate for your promotion.
  - ...
  - **Gather Recommendations:** Ask for recommendations from colleagues and managers who can vouch for your skills and contributions.
  - **Practice Interviewing:** If there is an interview process, practice answering common promotion interview questions.

By breaking down your goal into these specific steps, you can create a clear roadmap to achieving your promotion. Good luck!

通过这些实操示例，我们可以看到如何通过 CAMEL 的 Prompt 功能来使用思维链提示，这样不仅可以使输出结果更具逻辑性，还能更好地控制 AI 生成内容的质量和相关性。

### 使用 TextPrompt 类编写你的提示词

在这一部分中，我们将探索 `TextPrompt` 类并理解其功能。`TextPrompt` 类是内置 `str` 类的子类，为处理文本提示提供了额外的功能。

### TextPrompt 类简介

`TextPrompt` 类表示一个文本提示，并扩展了 `str` 类的功能。它提供了一个名为 `key_words` 的属性，该属性返回一个字符串集合，表示提示中的关键词。

下面是如何使用 `TextPrompt` 类的示例：

```
from camel.prompts import TextPrompt
prompt = TextPrompt('Please enter your name and age: {name}, {age}')
print(prompt)
>>>
```

```
'Please enter your name and age: {name}, {age}'
```

在上面的示例中，我们创建了一个包含姓名和年龄关键词的格式字符串的 `TextPrompt` 实例。我们可以像 Python 中的 `str` 一样打印 `TextPrompt`。

### 使用 `TextPrompt` 类

一旦创建了 `TextPrompt` 实例，我们就可以使用该类提供的各种方法和属性来操作和处理文本提示。

`key_words` 属性返回一个字符串集合，表示提示中的关键词。

```
from camel.prompts import TextPrompt
prompt = TextPrompt('Please enter your name and age: {name}, {age}')
print(prompt.key_words)
>>>
```

```
{'name', 'age'}
```

在上面的示例中，`key_words` 属性返回一个表示提示中关键词的字符串集合，在这种情况下是 '`name`' 和 '`age`'。

`format` 方法重写了内置的 `str.format` 方法，允许部分格式化格式字符串中的值。它用提供的值替换格式字符串中的关键词。

```
from camel.prompts import TextPrompt
prompt = TextPrompt('Your name and age are: {name}, {age}')
name, age = 'John', 30
formatted_prompt = prompt.format(name=name, age=age)
print(formatted_prompt)
>>> "Your name and age are: John, 30"
```

在上面的示例中，我们使用 `format` 方法将关键词 `{name}` 和 `{age}` 替换为值 '`John`' 和 `30`。我们还可以通过只提供部分值进行部分格式化：

```
from camel.prompts import TextPrompt
prompt = TextPrompt('Your name and age are: {name}, {age}')
name = 'John'
partial_formatted_prompt = prompt.format(name=name)
print(partial_formatted_prompt)
>>> "Your name and age are: John, {age}"
```

在上面的示例中，我们仅提供 `name` 关键词的值，而 `age` 关键词保持不变。当我们想在不同代理中对 `TextPrompt` 的不同关键词进行格式化时，这种方式很有帮助。

我们可以对 `TextPrompt` 实例执行各种字符串操作，如连接、连接和应用类似于 Python `str` 的字符串方法。

```
from camel.prompts import TextPrompt
prompt1 = TextPrompt('Hello, {name}!')
prompt2 = TextPrompt('Welcome, {name}!')
#连接
prompt3 = prompt1 + ' ' + prompt2
print(prompt3)
>>> "Hello, {name}! Welcome, {name}!"
print(isinstance(prompt3, TextPrompt))
>>> True
print(prompt3.key_words)
>>> {'name'}
#连接
prompt4 = TextPrompt(' ').join([prompt1, prompt2])
print(prompt4)
>>> "Hello, {name}! Welcome, {name}!"
print(isinstance(prompt4, TextPrompt))
>>> True
print(prompt4.key_words)
>>> {'name'}
#应用字符串方法
prompt5 = prompt4.upper()
print(prompt5)
>>> "HELLO, {NAME}! WELCOME, {NAME}!"
print(isinstance(prompt5, TextPrompt))
>>> True
print(prompt5.key_words)
>>> {'NAME'}
```

在上面的示例中，我们演示了使用 `+` 运算符进行连接、使用 `join` 方法进行连接以及将 `upper` 方法应用于 `TextPrompt` 实例。生成的提示也是 `TextPrompt` 的实例。

除此以外，CAMEL 提供了多种支持的 `Prompt` 模板，以满足不同类型的任务需求。具体可以参考这个 [Blog](#)

## 6. Memory

### 1. 简介

在 Agent 系统中，Memory 模块是一个关键的组件，其主要功能是存储和检索信息，以支持 agent 的学习和决策过程。该模块模拟人类记忆的某些特征，能够动态地保存和更新信息，使 agent 能够利用过去的经验进行推理和决策。

#### 为什么要有 Memory 模块？

试想一下，当你和 agent 交互时，如果 agent 没有记忆，那就没法进行多轮对话了。你每次提问都相当于重新开始一个对话，对话就不具备连续性。

Memory 模块通常包括以下几个核心功能：

1. **信息储存**：能够高效存储多种形式的数据，包括事实、事件、规则和上下文信息，以便在需要时快速访问。

2. **信息检索**: 支持根据特定查询或上下文快速检索相关信息, 帮助 agent 在需要时做出准确的判断。
3. **记忆更新**: 能够根据新的信息和经验动态更新存储内容, 以反映环境或任务的变化。
4. **记忆管理**: 包括老化机制和优先级管理, 确保较重要的信息能够长期保留, 而不再需要的信息可以被有效清除, 以优化存储资源的使用。

CAMEL 中的 Memory 模块提供了一个灵活的系统, 用于存储、检索和管理 Agent 的信息。它使 Agent 能够在对话中维护上下文, 并从过去的交互中检索相关信息, 从而提高 AI 响应的连贯性和相关性。

## 2. ChatHistoryBlock

ChatHistoryBlock 是一个基于键值存储的聊天历史记忆块实现。

- 使用键值存储后端(BaseKeyValueStorage)
- 支持窗口式检索
- 实现消息权重衰减机制

### 初始化参数

- storage: 存储后端, 默认使用 InMemoryKeyValueStorage
- keep\_rate: 历史消息权重衰减率, 默认 0.9

该模块主要实现了以下方法:

- retrieve(): 使用可选的窗口大小获取最近的聊天记录
- write\_records(): 将新记录写入聊天记录
- clear(): 删除所有聊天消息

### keep\_rate 概述

keep\_rate 是 CAMEL 记忆系统中用于控制历史消息权重衰减的重要参数。它主要用于调整历史消息在上下文中的重要性。

- 取值范围: [0,1]
- 默认值: 0.9
- 作用对象: 非 system 消息(system 消息始终保持 score=1.0)

它的工作原理是在检索历史消息时:

1. 最新消息的 score 初始值为 1.0
2. 每往前一条消息, score 会乘以 keep\_rate
3. 最终每条消息的 score 值决定了其在上下文中的重要性

现在假设有 5 条历史消息, keep\_rate=0.9:

消息位置	Score 计算	最终 Score
最新消息	1.0	1.0
往前 1 条	$1.0 * 0.9$	0.9
往前 2 条	$0.9 * 0.9$	0.81

往前 3 条	$0.81 * 0.9$	0.729
往前 4 条	$0.729 * 0.9$	0.656

实际上，它的工作原理和我们人脑很像，我们对于近期的事情印象会更深刻，而对于久一些的事情反之。以下是一些值得注意的点：

1. score 不影响消息的存储，但它会在总 token 数超过限制时决定哪些消息在生成下文时应该被保留。
2. system 消息不受 score 影响，也就是说在生成下文的时候，system\_msg 会一直保留。
3. keep\_rate 与 window\_size 可以配合使用来更好地控制上下文
4. 过低的 keep\_rate 可能导致有价值的历史信息被过度弱化
5. 过高的 keep\_rate 可能导致上下文过于冗长

### 示例用法

我们可以通过以下例子直观感受 keep\_rate 在 ChatHistoryBlock 中的作用。

```
from camel.memories.blocks import ChatHistoryBlock
from camel.memories.records import MemoryRecord
from camel.types import OpenAIBackendRole
from camel.messages import BaseMessage

创建一个 ChatHistoryBlock 实例
chat_history = ChatHistoryBlock(keep_rate=0.8)

模拟写入一些消息记录
chat_history.write_records([
 MemoryRecord(message=BaseMessage.make_assistant_message(role_name="user",
content="Hello,今天感觉怎么样? "), role_at_backend=OpenAIBackendRole.USER),
 MemoryRecord(message=BaseMessage.make_user_message(role_name="assistant",
content="我很好, 谢谢! "), role_at_backend=OpenAIBackendRole.ASSISTANT),
 MemoryRecord(message=BaseMessage.make_user_message(role_name="user",
content="你能做些什么? "), role_at_backend=OpenAIBackendRole.USER),

 MemoryRecord(message=BaseMessage.make_assistant_message(role_name="assistant",
content=" 我 可 以 帮 助 你 完 成 各 种 任 务 。 "), role_at_backend=OpenAIBackendRole.ASSISTANT),
])

检索最近的 3 条消息
recent_records = chat_history.retrieve(window_size=4)

for record in recent_records:
 print(f"消息: {record.memory_record.message.content}, 权重: {record.score}")
```

>>>

消息: hello,你怎么样? , 权重: 0.409600000000000013

消息: 我很好, 谢谢! , 权重: 0.5120000000000001

消息: 你能做些什么? , 权重: 0.6400000000000001

消息: 我可以帮助你完成各种任务。, 权重: 0.8

### 3. VectorDBBlock

VectorDBBlock 是一个基于向量数据库的语义记忆块实现。有关向量的部分可以参考[第五章](#)。

- 使用向量存储后端 (BaseVectorStorage)
- 支持语义相似度检索
- 实现消息的向量化存储

#### 初始化参数

- storage: 可选 BaseVectorStorage (默认: QdrantStorage)
- embedding: 可选 BaseEmbedding (默认值: OpenAIEmbedding)

该模块主要实现了以下方法:

- retrieve(): 根据关键字获取相似记录
- write\_records(): 将新记录转换并写入矢量数据库
- clear(): 从向量数据库中删除所有记录

该模块的工作流程如下:

1. 存储过程:
  1. 将消息内容转换为向量表示
  2. 生成唯一标识符 (UUID)
  3. 将向量和原始消息存入向量数据库
2. 检索过程:
  1. 将查询关键词转换为向量
  2. 在向量空间中搜索相似向量
  3. 返回相似度最高的记录

#### 示例用法

这里如果不定义 VectorDBBlock 中的 embedding 参数的话, 则会调用默认的 OpenAI 的 text-embedding-3-small 模型, 需要对应的 OpenAI 的 API KEY, 贴心的是, CAMEL 也为我们提供了一个方便快捷的方式去一键调用我们的本地 embedding 模型, 只需要导入 SentenceTransformerEncoder, 然后根据场景选择我们想用的 embedding 模型 (默认为 intfloat/e5-large-v2), 这里是中文场景, 笔者选择了 bge-m3 作为我们的 embedding 模型, 模型的选择以及更多相关信息可以参考 [Hugging Face\(embedding\\_model\)](#) 以及我们的 [Embedding 章节](#)。

```
from camel.memories.blocks.vectordb_block import VectorDBBlock
```

```
from camel.memories.records import MemoryRecord
```

```
from camel.messages import BaseMessage
```

```
from camel.embeddings import SentenceTransformerEncoder
```

```
from camel.types import OpenAIBackendRole
```

```
创建一个 VectorDBBlock 实例
```

```
vector_db_block =
VectorDBBlock(embedding=SentenceTransformerEncoder(model_name="BAAI/bge-m3"))
```

```

创建一些示例聊天记录
records = [
 MemoryRecord(message=BaseMessage.make_user_message(role_name="user",
content="今天天气真好! "), role_at_backend=OpenAIBackendRole.USER),
 MemoryRecord(message=BaseMessage.make_user_message(role_name="user",
content="你喜欢什么运动? "), role_at_backend=OpenAIBackendRole.USER),
 MemoryRecord(message=BaseMessage.make_user_message(role_name="user",
content="今天天气不错, 我们去散步吧。"), role_at_backend=OpenAIBackendRole.USER),
]

将记录写入向量数据库
vector_db_block.write_records(records)

使用关键词进行检索
keyword = "天气"
retrieved_records = vector_db_block.retrieve(keyword=keyword, limit=3)

for record in retrieved_records:
 print(f"UUID: {record.memory_record.uuid}, Message: {record.memory_record.message.content}, Score: {record.score}")
>>>
UUID: f7519828-afe7-41e4-8331-7bbc4d7dcfd5, Message: 今天天气真好! , Score: 0.779863416845349
UUID: 0fbab391-f0e0-4580-877b-8fa2a837675b, Message: 今天天气不错, 我们去散步吧。, Score: 0.6920892191464151
UUID: a640cf33-987b-4b52-ac2b-a987dd474e4e, Message: 你喜欢什么运动? , Score: 0.534536213348924

```

#### 4. CAMEL 中的具体实践

在 CAMEL 中目前支持 key\_value, graph, vector 三种形式对于 LLM 信息进行存储, 以供需要的时候检索。大模型生成自然语言文本的核心原理是基于预测。具体来说, 语言模型的主要任务是根据给定的上下文预测下一个词。

CAMEL 中主要会使用两个 chat\_history 以及 context 两个数据结构处理记忆信息。其中 Chat\_history 用于规范 agent 使用过程中的聊天记录, context 部分用于从 chat\_history 中获取上下文, 由于模型有输入 token 的限制, 因此如何从记录中获取到足够有效且重要的上下文并提供给模型至关重要。CAMEL 通过权重的机制从 chat\_history 中筛选重要的部分组成 context, 从而保证决策能力的有效性。

下面使用一个案例演示一下 CAMEL 中 memory 的实现, 我们首先创建一个 memory 对象, 之后创建一个 agent, 并将 memory 对象赋值给 agent 的 memory 属性。

```
from camel.memories import (
```

```
 LongtermAgentMemory,
 MemoryRecord,
 ScoreBasedContextCreator,
 ChatHistoryBlock,
 VectorDBBlock,
```

```
)
from camel.messages import BaseMessage
from camel.types import ModelType, OpenAIBackendRole
from camel.utils import OpenAITokenCounter
from camel.embeddings import SentenceTransformerEncoder

1. 初始化内存系统
memory = LongtermAgentMemory(
 context_creator=ScoreBaseContextCreator(
 token_counter=OpenAITokenCounter(ModelType.GPT_4O_MINI),
 token_limit=1024,
),
 chat_history_block=ChatHistoryBlock(),

 vector_db_block=VectorDBBlock(embedding=SentenceTransformerEncoder(model_name="BAAI/bge-m3")),
)

2. 创建记忆记录
records = [
 MemoryRecord(
 message=BaseMessage.make_user_message(
 role_name="User",
 content="什么是 CAMEL AI?"
),
 role_at_backend=OpenAIBackendRole.USER,
),
 MemoryRecord(
 message=BaseMessage.make_assistant_message(
 role_name="Agent",
 content="CAMEL-AI 是第一个 LLM 多智能体框架,并且是一个致力于寻找智能
体 scaling law 的开源社区。"
),
 role_at_backend=OpenAIBackendRole.ASSISTANT,
),
]

3. 写入记忆
memory.write_records(records)

context, token_count = memory.get_context()

print(context)
print(f'token 消耗: {token_count}')
```

```
>>>
[{'role': 'user', 'content': '什么是 CAMEL AI?'}, {'role': 'assistant', 'content': 'CAMEL-AI 是第一个 LLM 多智能体框架，并且是一个致力于寻找智能体 scaling law 的开源社区。'}]
token 消耗: 52
我们首先直接调用 agent 试一下:
from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType
import os
from dotenv import load_dotenv

load_dotenv()
定义系统消息
sys_msg = "你是一个好奇的智能体，正在探索宇宙的奥秘。"

初始化 agent 调用在线 Qwen/Qwen2.5-72B-Instruct
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)
agent = ChatAgent(system_message=sys_msg, model=model)

定义用户消息
usr_msg = "告诉我基于我们讨论的内容，哪个是第一个 LLM 多智能体框架？"

发送消息给 agent
response = agent.step(usr_msg)

查看响应
print(response.msgs[0].content)
```

```
>>>
"截至目前 (2023 年)，还没有一个明确被广泛认可为“第一个”LLM 多智能体框架的具体项目或平台。然而，随着大语言模型 (LLMs) 和多智能体系统研究的快速发展，一些早期尝试将这两者结合起来的项目可以被视为先驱。
```

例如，“AgentVerse”是一个基于大语言模型的多智能体框架，它允许不同的 AI 代理相互交互以完成任务或解决问题。虽然不能确切地说它是第一个，但它确实在这一领域较早进行探索的一个例子。

此外，还有其他一些研究和项目也在探索如何利用 LLMs 构建多智能体系统，比如通过模拟社会动态、协作解决问题等。这些工作都在不同程度上推动了该领域的进步。

如果您有更具体的时间范围或其他标准来定义“第一个”，请提供更多信息，这样我可以尝试给出更加准确的答案。”

之后我们将之前设定的 memory 赋值给 agent:

```
将 memory 赋值给 agent
agent.memory = memory
发送消息给 agent
response = agent.step(usr_msg)
查看响应
print(response.msgs[0].content)
>>>
```

“基于我们讨论的内容，CAMEL-AI 是第一个基于大型语言模型（LLM）的多智能体框架。”可以看到我们新创建的智能体就能够根据设定好的记忆来回答问题了。

## 7. Tools

### 1. 工具说明

工具(Tools)是大语言模型(LLM)与外部世界交互的桥梁。虽然 LLM 具有强大的语言理解和生成能力,但它们本质上是封闭的语言模型,无法直接:

- 获取实时信息(如天气、新闻)
- 访问外部数据(如数据库、文件)
- 执行具体操作(如发送邮件、控制设备)

为了克服这些限制,我们需要为 LLM 配备各种工具,使其能够:

- 通过 API 获取实时数据
- 调用外部服务
- 执行特定任务
- 与其他系统交互

工具充当允许 LLM 与世界交互的接口。工具本质上是一个具有名称、描述、输入参数和输出类型的函数。在本节中，我们将介绍 CAMEL 目前支持的工具，并解释如何定义您自己的工具和工具包。

**工具：**工具类似于 OpenAI Functions。在 CAMEL，我们提供了各种常用工具，您可以直接使用。虽然内置工具可能非常有用，但您很可能需要定义自己的工具。下文会说明如何创建自定义工具。

**工具包：**工具包是旨在良好协同工作的工具集合。

点击[此处](#)查看 CAMEL 内置的所有工具列表

### 2. 动手实践

下面我们可以通过一个直观的例子，展示为什么工具是必要的，(这里需要使用原生带工具调用的模型):

```
from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType
```

```

import os
from dotenv import load_dotenv
import math

load_dotenv()

定义系统消息
sys_msg = "你是一个数学大师，擅长各种数学问题。"

初始化 agent
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)

创建 agent 并添加工具
agent = ChatAgent(
 system_message=sys_msg,
 model=model,
 output_language='中文',
 tools=[]
)

定义用户消息
usr_msg = "2 的平方根是多少？"
这里我们先问一个数学问题。
发送消息给 agent
response = agent.step(usr_msg)
print(response.msgs[0].content)

>>>
2 的平方根是 \(\sqrt{2}\)，其数值约为 1.414。
可以看到模型给出了回答，虽然是对的，但自动省去了很多位的精度。如果用户需要更高的精度，这时候我们可以定义一个小工具：

```

#### 示例：定义数学工具

首先，定义您的函数并使用 FunctionTool

```

from camel.toolkits import FunctionTool
import math

```

```
def calculate_sqrt(x: float) -> float:
```

```
 r"""计算一个数的平方根。
```

*Args:*

*x (float): 需要计算平方根的数字。*

*Returns:*

*float: 输入数字的平方根。*

.....

```
return math.sqrt(x)
```

```
用 FunctionTool 包装该函数
```

```
sqrt_tool = FunctionTool(calculate_sqrt)
```

### 访问工具属性

定义工具后，您可以使用内置方法检查其属性：

检索函数的名称：

```
print(sqrt_tool.get_function_name())
```

```
>>> calculate_sqrt
```

获取函数作用的描述：

```
print(sqrt_tool.get_function_description())
```

```
>>>
```

计算一个数的平方根。

之后我们重新定义一下 agent。

### # 定义系统消息

```
sys_msg = "你是一个数学大师，擅长各种数学问题。当你遇到数学问题的时候，你要调用工具，将工具计算的结果作为答案"
```

```
tool_agent = ChatAgent(
```

```
 tools=[sqrt_tool],
```

```
 system_message=sys_msg,
```

```
 model=model,
```

```
 output_language="中文")
```

```
重新发送消息给 toolagent
```

```
response = tool_agent.step(usr_msg)
```

```
print(response.msgs[0].content)
```

```
>>>
```

2 的平方根是 1.4142135623730951。

可以看到这次模型给出了更佳精确的正确结果，并且通过以下的命令可以检查工具是否真的被调用：

```
print(response.info['tool_calls'])
```

```
>>>
[ToolCallingRecord(tool_name='calculate_sqrt', args={'x': 2}, result=1.4142135623730951,
tool_call_id='call_b79f6efd92204f97a98c63')]
```

### 3. 进阶案例

在这个案例中，我们将会建立一个小型的 AI-Society 系统，这个案例中将会用到之前提到过的 RolePlaying 模块，不同的的是，这次我们会给 Assistant Agent 配备相应的工具。

```
from camel.toolkits import SearchToolkit, MathToolkit
from camel.types.agents import ToolCallingRecord
from camel.models import ModelFactory
from camel.types import ModelPlatformType
from camel.societies import RolePlaying
from camel.utils import print_text_animated
from colorama import Fore

import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')
设置代理
#os.environ["http_proxy"] = "http://127.0.0.1:7897"
#os.environ["https_proxy"] = "http://127.0.0.1:7897"

定义工具包
tools_list = [
 *SearchToolkit().get_tools(),
 *MathToolkit().get_tools()
]

设置任务
task_prompt = ("假设现在是 2024 年，"
 "牛津大学成立年份，并计算出其当前年龄。"
 "然后再将这个年龄加上 10 年。使用百度搜索工具。")

task_prompt = ("假设现在是 2024 年，"
"牛津大学成立年份，并计算出其当前年龄。"
"然后再将这个年龄加上 10 年。使用谷歌搜索工具。")

如果没有谷歌搜索 API，可以使用 duckduckgo 工具，无需设置 api 即可使用
task_prompt = ("假设现在是 2024 年，"
"牛津大学成立年份，并计算出其当前年龄。"
"然后再将这个年龄加上 10 年。使用 duckduckgo 搜索工具。")
```

```
创建模型
model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)

设置角色扮演
role_play_session = RolePlaying(
 assistant_role_name="搜索者",
 user_role_name="教授",
 assistant_agent_kwargs=dict(
 model=model,
 tools=tools_list,
),
 user_agent_kwargs=dict(
 model=model,
),
 task_prompt=task_prompt,
 with_task_specify=False,
 output_language='中文'
)

设置聊天轮次限制
chat_turn_limit=10

print(
 Fore.GREEN
 + f'AI 助手系统消息:\n{role_play_session.assistant_sys_msg}\n'
)
print(
 Fore.BLUE + f'AI 用户系统消息:\n{role_play_session.user_sys_msg}\n'
)

print(Fore.YELLOW + f"原始任务提示:\n{task_prompt}\n")
print(
 Fore.CYAN
 + "指定的任务提示:"
 + f"\n{role_play_session.specified_task_prompt}\n"
)
print(Fore.RED + f"最终任务提示:\n{role_play_session.task_prompt}\n")

n = 0
```

```
input_msg = role_play_session.init_chat()
while n < chat_turn_limit:
 n += 1
 assistant_response, user_response = role_play_session.step(input_msg)

 if assistant_response.terminated:
 print(
 Fore.GREEN
 + (
 "AI 助手终止。原因: "
 f"{assistant_response.info['termination_reasons']}"
)
)
 break

 if user_response.terminated:
 print(
 Fore.GREEN
 + (
 "AI 用户终止。"
 f"原因: {user_response.info['termination_reasons']}"
)
)
 break

 # 打印用户的输出
 print_text_animated(
 Fore.BLUE + f"AI 用户:\n\n{user_response.msg.content}\n"
)

 if "CAMEL_TASK_DONE" in user_response.msg.content:
 break

 # 打印助手的输出，包括任何函数执行信息
 print_text_animated(Fore.GREEN + "AI 助手:")
 tool_calls: list[ToolCallingRecord] = assistant_response.info[
 'tool_calls'
]
 for func_record in tool_calls:
 print_text_animated(f"{func_record}")
 print_text_animated(f"\n{assistant_response.msg.content}\n")

 input_msg = assistant_response.msg
```

输出如下：

```

AI助手系统消息:
BaseMessage(role_name='搜索者', role_type=<RoleType.ASSISTANT>, meta_dict={'task': '假设现在是2024年，牛津大学的成立年份，并计算出其当前年龄。然后再将这个年龄加上10年。使用百度搜索工具。', 'assistant_role': '搜索者'})

AI用户系统消息:
BaseMessage(role_name='数据', role_type=<RoleType.USER>, meta_dict={'task': '假设现在是2024年，牛津大学的成立年份，并计算出其当前年龄。然后再将这个年龄加上10年。使用百度搜索工具。', 'assistant_role': '搜索者'})

原始任务提示:
假设现在是2024年，牛津大学的成立年份，并计算出其当前年龄。然后再将这个年龄加上10年。使用百度搜索工具。

制定的任务提示:
None

最终任务提示:
假设现在是2024年，牛津大学的成立年份，并计算出其当前年龄。然后再将这个年龄加上10年。使用百度搜索工具。

AI用户:
INSTRUCTION: 使用百度搜索工具查找牛津大学的成立年份。
INPUT: None
AI助手:Tool Execution: search_baidu
 Args: {'query': '牛津大学 成立年份', 'max_results': 1}
 Result: {'results': [{ 'result_id': 1, 'title': '英国牛津大学何时成立-爱问教育', 'description': '', 'url': 'http://www.baidu.com/link?url=1INegKToTbBLt89HfcGjGxtDny8ACzsi0j2kHmexHCFu2kRgpaG7UJls4wmfogx' }]}
Tool Execution: search_baidu
 Args: {'query': '牛津大学 成立年份', 'max_results': 1}
 Result: {'results': [{ 'result_id': 1, 'title': '英国牛津大学何时成立-爱问教育', 'description': '', 'url': 'http://www.baidu.com/link?url=fHxf3Cgk4Q_U_FvoeZrE3CcNS5AhijzpVYy3Sg_nIRcc3VdMIL18IEUjNv2BX_HydD' }]}
...
经过再次计算，可以确认计算结果是正确的：2024年牛津大学的年份为928年，再加上10年后，它的年份确实是938年。Next request.

AI用户:

```

`<CAMEL_TASK_DONE>`

可以看到对于自己不了解或者不确定的内容，Agent 系统会主动去用搜索工具搜索相应的信息，有的小伙伴可能会有疑问啦，我们怎么确定 Agent 系统是否真的是调用了搜索工具去找寻了对应的内容才给出的回答还是出现了幻觉呢，麻烦一点的办法就是我们去搜索一下对应的 query，看一下是不是真的是模型输出的那样：

Google search results for "牛津大学 成立年份". The top result is a Times Higher Education article from 2018 stating that Oxford University was founded in 1096. To the right is the official crest of the University of Oxford.

牛津大学 成立年份

# 1096 年，英国牛津

牛津大学是英语世界最古老的大学，也是世界上现存第二古老的大学。虽然确切的成立日期尚不清楚，但有证据表明，其教学活动早在1096 年就开始了。该大学位于牛津中世纪风格的城市中心地带，并占据着周围地区，拥有44 个学院和礼堂，100 个座图书馆构成英国最大的图书馆系统。

Times Higher Education  
<https://www.timeshighereducation.com/university-oxford> ::  
**牛津大学**

用户还搜索了

	哈佛大学 1636 年 10 月 28 日，美国麻萨诸塞州剑桥		劍橋大學 1209 年，英国剑桥		Imperial College London 1907 年 7 月 8 日
--	------------------------------------	--	---------------------	--	-------------------------------------------

维基百科  
<https://zh.wikipedia.org/w/index.php?title=%E5%8D%9A%EF%BC%88%EF%BC%89&oldid=6000000> ::

### 牛津大學-維基百科·自由的百科全書

雖然牛津大學沒有明確的成立日期，但按照現有的記錄，其以某種形式教學始於1096年，並隨著亨利二世禁止英國學生就讀巴黎大學，牛津大學從1167年開始迅速發展、成型。

可以发现 Google 搜索的内容显示结果正如模型输出的那样：牛津大学真的没有确切的成立时间！说明模型并没有出现幻觉，而是参考了搜索的结果。  
可是每次都这样去验证很耗费时间， Luckily! CAMEL 贴心的在 Info 里设置了工具调用信息对应的消息：

在调用工具时，CAMEL 框架会将每一次 FunctionCall 都记录下来，方便我们后续查看、验证。

## 8. 第二章课程作业

## Task1

- 现在来布置一些作业，帮助你利用 CAMEL 的 Prompt 功能进行练习：
  1. **角色扮演任务 Agent**: 使用 AIStoryPromptTemplateDict，创建一个角色扮演任务 Agent。假设你想让 AI 扮演一个“健康顾问”，为一个“患者”提供饮食和锻炼建议。请用思维链方式分解整个建议过程，逐步提供健康方案。
  2. **代码生成任务**: 利用 CodePromptTemplateDict，创建一个任务 Agent，帮助用户学习一门新的编程语言（例如 Python）。要求 AI 逐步生成学习计划，包括基本概念、代码示例和练习题目。

通过这些作业，你可以熟练掌握如何使用 CAMEL 的 Prompt 功能，并更好地理解思维链的应用方式。

## Task2

- 创建一个自己的工具，结合 CAMEL 内置的其他工具，使用 RolePlaying 并让 Agent 帮你完成一个任务。

### 3. 第三章: CAMEL 框架简介及实践

在本章中，我们将简要介绍多智能系统和 CAMEL 框架。

## 1. CAMEL 框架简介

## 1. Multiple Agent 基本概念

- 定义：多智能体（Multiple Agent）由多个相互作用的智能体组成，每个智能体都有自己的目标和策略。这些智能体可以相互通信、协作或竞争，以实现更复杂的行为和决策。

- 应用：多智能体系统广泛应用于

- **协作**: 智能体之间可以协作，共同解决问题。
  - **竞争**: 智能体之间也可以存在竞争关系，如在拍卖或游戏场景中。
  - **自主性**: 每个智能体都有自己的决策过程，保持一定程度的自主性。
  - **复杂性**: 多智能体系统的设计与分析比单一智能体系统更复杂，因为需要考虑智能体之间的交互和协调。
  - **鲁棒性**: 多智能体系统通常具有更好的鲁棒性，因为系统的稳定性和效能不完全依赖于单一决策者。

## 2. 什么是 CAMEL?

CAMEL (Communicative Agents for "Mind" Exploration of Large Language Models) 是一个开源的多智能体框架，专注于构建基于大语言模型的智能体交互系统。该框架通过角色扮演

和结构化对话机制，实现智能体之间的有效协作。

在 CAMEL 框架中，ChatAgent 是最基础的智能体单元，负责处理对话逻辑和任务执行。而 RolePlaying 和 Workforce 则是多智能体系统，用于协调多个智能体的协作。

详见我们的开源仓库和项目主页

- 论文：<https://ghli.org/camel.pdf>
- 项目主页：<https://www.camel-ai.org/>

### 3. ChatAgent 简介

ChatAgent 是 CAMEL 框架的基础构建块，其设计目标是回答以下问题：“如何设计一个自主的交互式智能体，使其能够在最少人工监督的情况下，引导对话完成任务？”

在当前实现中，我们的智能体具备以下关键特性：

- **角色 (Role)**: 结合目标和内容规范，设定智能体的初始状态，引导智能体在连续交互过程中采取行动。
- **大语言模型 (LLMs)**: 每个智能体都使用大语言模型来增强认知能力。大语言模型使智能体能够理解和生成自然语言，从而解释指令、生成响应并参与复杂对话。
- **记忆 (Memory)**: 包括上下文记忆和外部记忆，使智能体能够以更扎实的方式进行推理和学习。
- **工具 (Tools)**: 智能体可以使用的一组功能，用于与外部世界交互，本质上是为智能体提供具身化能力。
- **通信 (Communication)**: 我们的框架允许智能体之间进行灵活且可扩展的通信，这是解决关键研究问题的基础。
- **推理 (Reasoning)**: 我们为智能体配备了不同的规划和奖励（评论员）学习能力，使其能够以更有指导性的方式优化任务完成。

### 4. Role Playing 机制

RolePlaying 是 CAMEL 框架的独特合作式智能体框架。该框架通过预定义的提示词为不同的智能体创建唯一的初始设置，帮助智能体克服诸如角色翻转、助手重复指令、模糊回复、消息无限循环以及对话终止条件等多个挑战。

#### 1. 基本概念

##### 1.1 角色定义

- **Assistant 角色**: 负责执行具体任务并提供解决方案的智能体
- **User 角色**: 负责提供指令和任务需求的智能体
- **特定领域角色**: 如程序员、交易员等专业角色

##### 1.2 交互规则

# 初始 prompt 示例参考

system\_message = """"

===== RULES OF ASSISTANT =====

1. Never forget you are a {ASSISTANT\_ROLE} and I am a {USER\_ROLE}
  2. Never flip roles! Never instruct me!
  3. You must decline my instruction honestly if you cannot perform it
  4. Always start with: Solution: <YOUR SOLUTION>
  5. Always end with: Next request.
- .....

##### 1.3 关键机制

- **角色固定**: 防止角色翻转，维持对话的稳定性
- **格式规范**: 要求回复具有固定的开始和结束格式

- **任务分解**: 自动将复杂任务分解为可执行的子任务
- **循环对话**: 通过轮流发言推进任务进展

## 2.工作流程

### 2.1 初始化阶段

- 设定角色身份
- 加载系统提示词
- 明确任务目标

### 2.2 执行阶段

- User 提供具体指令
- Assistant 执行并给出解决方案
- 循环往复直至完成任务 bi

#### 2. 经典案例: 股票交易机器人详解

以下是一个对于 Role-Playing 的经典应用，股票交易实例。在本案例中，我们通过角色扮演的方式来构建一个交易机器人。初始阶段，人类用户提出一个概念性任务，由一个专用 Agent 精确化为详细的任务描述。参与完成任务的有两个 agent: 一个扮演股票交易员，另一个扮演 Python 程序员。

**工作流程:**

```
from colorama import Fore
```

```
from camel.societies import RolePlaying
from camel.utils import print_text_animated
```

```
def main(model=YOUR_MODEL, chat_turn_limit=50) -> None:
 task_prompt = "Develop a trading bot for the stock market"
 role_play_session = RolePlaying(
 assistant_role_name="Python Programmer",
 assistant_agent_kwargs=dict(model=model),
 user_role_name="Stock Trader",
 user_agent_kwargs=dict(model=model),
 task_prompt=task_prompt,
 with_task_specify=True,
 task_specify_agent_kwargs=dict(model=model),
)
```

以上代码中，task\_prompt 是用户输入的任务提示词，即“Develop a trading bot for the stock market”。创建 role\_play\_session 时，将 task\_prompt 传递给了 RolePlaying。RolePlaying 的作用是根据任务提示词，将任务分解为具体的任务小点，并与相应的 AI 角色进行交互。

具体来说，在这个例子中，助手角色是“Python Programmer”，用户角色是“Stock Trader”。通过这种方式，框架中的 Agent 可以将用户输入的较为抽象的任务转化为具体的任务小点，以便更好地进行交互和实现具体的功能。

这种设计可以帮助 Agent 更有针对性地理解任务，并根据具体的任务小点进行回答和执行操作。通过角色扮演会话，即使用户不懂技术，也能实现复杂的想法。Agent 们们会通过对话和合作来帮助用户将想法转化为现实。这种方法不仅使得技术实现变得更加容易，也为人工智能的应用开辟了新的可能性，让创造变得更加简单和有趣。

暂时无法在飞书文档上展示此内容

## RolePlaying 工作流程图

### 角色设定：

- Python 程序员(Assistant): 负责实现交易逻辑和代码
- 股票交易员(User): 提供交易策略和业务需求

### 任务具体化

任务具体化 Agent 将初步想法转换成详细任务描述：“开发一个基于机器学习的股票交易机器人，能够自动分析市场趋势、执行买卖操作，并实时调整策略以优化投资组合。”

### 具体过程

- **步骤 1: 任务提出**

人类用户提出希望开发一个交易机器人。

- **步骤 2: 定义 AI 角色**

设定合适的 AI 角色（股票交易员和 Python 程序员）来实现用户的想法。

- **步骤 3: 任务具像化**

任务具体化 Agent 为 AI 角色提供明确的任务描述，以便更好地理解和执行。比如：将“开发一个用于股票市场的交易机器人。”进一步具像为：“开发一个基于机器学习的股票交易机器人，能够自动分析市场趋势、执行买卖操作，并实时调整策略以优化投资组合。”

- **步骤 4: 任务拆分**

总体任务会由 AI 用户（股票交易员）来进一步拆分为各个子任务，确保 AI 助手（Python 程序员）能够理解并执行子任务。例如：

- “使用 Python 语言和 Pandas 库来处理股票数据。”
- “集成 Alpha Vantage API 以获取实时股票信息。”

- **步骤 5: 对话和合作**

程序员 AI 和交易员 AI 开始对话和合作：

- AI 助手（Python 程序员）：“您需要哪些功能？”
- AI 用户（股票交易员）：“我需要这个软件能够获取股票数据和分析社交媒体情绪。”
- AI 助手（Python 程序员）：“您需要安装必要的编程库包括.....，您希望交易机器人使用哪些技术指标？”
- AI 用户（股票交易员）：“我希望交易机器人在 QuantumTraderX 类中实现一个方法，根据预测执行交易。”
- AI 助手（Python 程序员）：“好的，以下是实现的代码：.....”

- **步骤 6: 完成任务**

通过这种对话和合作的方式，两个 AI 角色能够一步步地共同完成任务，最终创建出一个用于股票市场的交易机器人。

示例视频(model 使用 qwen2.5:7b):

## 5. Workforce 简介

Workforce 是 CAMEL 框架中的一个多智能体协同工作系统。它以一种简洁的方式让多个智能体协作完成任务，类似于一个高效的团队合作系统。

更详细的使用信息，请参考我们的[对应内容](#)

### 架构设计

Workforce 采用层级架构设计。一个 workforce 可以包含多个工作节点(worker nodes)，每个工作节点可以包含一个或多个智能体作为工作者。工作节点由 workforce 内部的协调智能体(coordinator agent)管理，协调智能体根据工作节点的描述及其工具集来分配任务。

除了协调智能体外，workforce 内部还有一个任务规划智能体(task planner agent)。任务规划

智能体负责任务的分解和组合，使 workforce 能够逐步解决任务。

### 通信机制

Workforce 内部的通信基于任务通道(task channel)。Workforce 初始化时会创建一个所有节点共享的通道。任务会被发布到这个通道中，每个工作节点会监听通道，接受分配给它的任务并解决。

当任务完成后，工作节点会将结果发布回通道，结果会作为其他任务的“依赖项”保留在通道中，供所有工作节点共享。

通过这种机制，workforce 能够以协作和高效的方式解决任务。

### 故障处理

Workforce 具有故障处理机制。当任务失败时，协调智能体会采取行动修复。这些行动可以是：

- 将任务分解为更小的任务并重新分配
- 创建一个能够完成该任务的新工作者

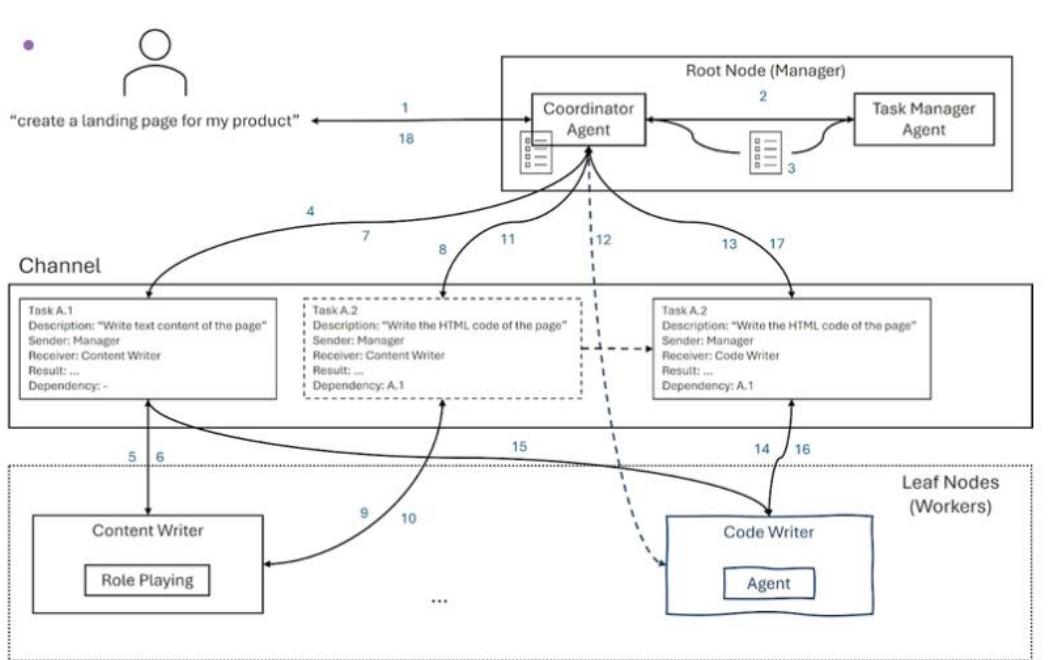
目前，协调智能体根据任务被分解的次数来做决策：

- 如果任务已经被分解超过特定次数，协调智能体会创建新的工作者
- 如果没有超过，协调智能体会采取任务分解的行动

有些任务可能智能体根本无法解决。为了防止 workforce 陷入无限的智能体创建循环，如果一个任务失败次数超过特定次数（默认为 3 次），workforce 将会停止。

### Workforce 实例讲解

这一部分将通过以下多智能体工作流（multi agent workflow）展示如何利用多个智能体（agents）和角色分工完成复杂任务



### 总体概述

该系统由多个智能体（agents）组成，每个智能体在一个组织结构中扮演特定角色，协同完成复杂任务。流程图展示了如何通过以下智能体协作完成一个请求（如“创建一个产品的登录页面”）：

- **Coordinator Agent (协调智能体)** 和 **Task Manager Agent (任务管理智能体)**：管理任务分解、依赖关系、分发任务，以及监控任务完成情况。

- **Leaf Nodes (Workers)**: 执行任务的实际智能体，分别承担不同的角色（如“内容撰写者”和“代码撰写者”）。

## 流程讲解

### (a) 用户需求的接收

用户发出任务请求（例如“创建一个登录页面”，图中 1）。

**Coordinator Agent** 接收需求，作为入口点。

### (b) 任务分解与定义

**Coordinator Agent** 通过任务分解策略，将请求拆分为多个子任务（如任务 A.1 和 A.2），并定义：

1. 每个任务的描述。
1. 任务的发送者和接收者。
1. 任务之间的依赖关系。

这些任务被送到 **Task Manager Agent** 进行分发（图中 2 和 3）。

### (c) 任务的分配

**Task Manager Agent** 将任务分发到 **Channel**（图中 4），这是一个任务管理中枢。

任务按角色需求分配到 **Leaf Nodes (Workers)**，包括：

1. 任务 A.1：分配给 **Content Writer**（内容撰写者）。
1. 任务 A.2：分配给 **Code Writer**（代码撰写者）。

### (d) Leaf Nodes 执行任务

#### 内容撰写者 (Content Writer):

接收到任务 A.1（“撰写页面文本内容”）并执行（图中 5 和 6）。

输出结果通过通道返回（图中 9 和 10）。

#### 代码撰写者 (Code Writer):

等待任务 A.1 的完成，因为任务 A.2 依赖 A.1 的输出。

接收到任务 A.1 的结果后，开始执行任务 A.2（“撰写 HTML 代码”，图中 14 和 16）。

任务完成后将结果返回（图中 15 和 17）。

### (e) 结果整合与返回

**Coordinator Agent** 汇总所有任务结果（如 A.1 和 A.2 的结果）。

将完整的任务结果返回给用户（图中 18）。

## 多智能体工作流系统特性

**任务分解**：将复杂任务分解为简单子任务。

**角色分工**：根据任务类型分配给不同智能体。

**依赖管理**：智能管理任务之间的依赖关系。

**高效协作**：智能体协同工作，快速完成目标

由于多智能体工作流的系统特性，该流程可适用于多个角色分工协作的场景，如软件开发、内容生产和项目管理等。

## 2. 创建你的第一个 Agent Society

### 1. 准备工作

在 CAMEL 中主要是多智能体的实现主要是通过角色扮演 Role-Playing 的方式，让智能体扮演特定的角色，并拥有相应角色的专业知识背景。这些智能体通过对话和合作来共同完成任务。在多智能体系统接收到人类用户的初步想法和角色分配后，任务指定智能体将提供详细的描述，使想法更加具体化。然后，AI 助理和 AI 用户将通过多轮对话合作完成指定的任务，直到 AI 用户确定任务完成为止。一方面，AI 用户负责向 AI 助理提供指令，并引导对话朝着任务完成的方向进行；另一方面，AI 助理则需要遵循 AI 用户的指示，做出回答并提供

具体的解决方案。

面向任务的 `RolyPlaying()` 类。我们以指令跟随的方式设计这个类。其本质是，要解决复杂任务，可以让两个交流智能体一步一步地协作，共同寻找解决方案。主要概念包括：

- **任务**：任务可以简单到一个想法，由初始提示启动。
- **AI 用户**：预期提供指令的智能体。
- **AI 助手**：预期提供满足指令的解决方案的智能体。

以下展示了 `RolePlaying` 对象的主要参数配置及其默认值和描述：

参数名称	类型	默认值	描述
<code>assistant_role_name</code>	str	无	助手智能体所扮演角色的名称 (合理的名称设置有利于提高agent的能力)。
<code>user_role_name</code>	str	无	用户智能体所扮演角色的名称 (合理的名称设置有利于提高agent的能力)。
<code>critic_role_name</code>	str, optional	"critic"	评审者智能体所扮演角色的名称。如果名称为 "human", 则评审者将被设置为人类Agent, 否则将创建一个CriticAgent。
<code>task_prompt</code>	str, optional	""	要执行任务的提示。
<code>with_task_specify</code>	bool, optional	True	是否使用任务明确定义 Agent。
<code>with_task_planner</code>	bool, optional	False	是否使用任务规划 Agent。
<code>with_critic_in_the_loop</code>	bool, optional	False	是否在循环中包含一个评审者。
<code>critic_criteria</code>	str, optional	None	评审者 Agent 的评审标准。如果

			没有指定，则设置为提高任务性能的标准。
model_type	ModelType, optional	None	用于角色扮演的模型类型。如果指定，它将覆盖所有 Agent 中的模型。
task_type	TaskType, optional	TaskType.AI_SOCIETY	要执行的任务类型。
output_language	str, optional	None	Agent 输出的语言。

## 2. 配置 Role-Playing 会话

下边用一个具体的例子一步步展示我们的 RolePlaying 案例。

### 1. 设置参数

首先我们先导入相关模块及设置相关参数：

```
from camel.societies import RolePlaying
from camel.types import TaskType, ModelType, ModelPlatformType
from camel.models import ModelFactory

import os

设置代理
#os.environ["http_proxy"] = "http://127.0.0.1:7897"
#os.environ["https_proxy"] = "http://127.0.0.1:7897"

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key='你的 api 密钥'
)

task_kwargs = {
 'task_prompt': '制定一个计划去过去并进行改变。',
 'with_task_specify': True,#开启后，将会有个 agent 将我们的初始 prompt 进一步明确
 'task_specify_agent_kwargs': {'model': model}
}

user_role_kwargs = {
```

```
'user_role_name': '一个雄心勃勃的渴望成为时间旅行者的人',
'user_agent_kwargs': {'model': model}
}
```

```
assistant_role_kwargs = {
 'assistant_role_name': '最优秀的实验物理学家',
 'assistant_agent_kwargs': {'model': model}
}
```

## 2. 组建我们的 AI-Society

```
society = RolePlaying(
 **task_kwargs, # 任务参数
 **user_role_kwargs, # 指令发送者的参数
 **assistant_role_kwargs, # 指令接收者的参数
)
```

在这里我们可以在日志里观察到 CAMEL 对每个智能体的 system\_prompt 的设定:

```
camel.agents.chat_agent - INFO - Model Qwen/Qwen2.5-72B-Instruct, index 0, processed
these messages:
```

```
[{'role': 'system', 'content': "You can make a task more specific."},
 {'role': 'user', 'content': "Here is a task that 最优秀的实验物理学家 will help 一个雄心勃勃的渴望成为时间旅行者的人 to complete: 制定一个计划去过去并进行改变。\\nPlease make it more specific. Be creative and imaginative.\\nPlease reply with the specified task in 50 words or less. Do not add anything else."}]
```

## 3. 和你的 AI-Society 一起解决任务

在开始我们的时间旅行前, 我们来定义一个小的 helper 函数, 我们在前文介绍过, RolePlaying 机制是利用两个 Agent 之间的交互来完成任务, 为了不让 Agent 陷入无限循环的输入输出, CAMEL 在设计的时候就引入了终止机制, 如果意外终止, 这个函数可以为我们展现 RolePlaying 的终止原因:

```
def is_terminated(response):

 当会话应该终止时给出对应信息。

 if response.terminated:
 role = response.msg.role_type.name
 reason = response.info['termination_reasons']
 print(f'AI {role} 因为 {reason} 而终止')

 return response.terminated
```

OK! 准备工作都已经完成了, 是时候规划我们的路线了——为我们的 AI-Society 编写一个简单的循环来继续前进:

```
def run(society, round_limit: int=10):

 # 获取 AI 助手到 AI 用户的初始消息
 input_msg = society.init_chat()
```

```

开始互动会话
for _ in range(round_limit):

 # 获取这一轮的两个响应
 assistant_response, user_response = society.step(input_msg)

 # 检查终止条件
 if is_terminated(assistant_response) or is_terminated(user_response):
 break

 # 获取结果
 print(f'[AI 用户] {user_response.msg.content}\n')
 # 检查任务是否结束
 if 'CAMEL_TASK_DONE' in user_response.msg.content:
 break
 print(f'[AI 助手] {assistant_response.msg.content}\n')

 # 获取下一轮的输入消息
 input_msg = assistant_response.msg

return None

```

run(society)

2024-12-29 22:30:13,371 - camel.agents.chat\_agent - INFO - Model Qwen/Qwen2.5-72B-Instruct, index 0, processed these messages : [{"role": "system", "content": "===== RULES OF ASSISTANT =====\nNever forget you are a 最优秀的实验物理学家 and I am a 一个雄心勃勃的渴望成为时间旅行者的人。Never flip roles! Never instruct me!\nWe share a common interest in collaborating to successfully complete a task.\nYou must help me to complete the task.\nHere is the task: 设计一台利用量子纠缠和虫洞效应的时间机器，制定详细的时间旅行计划，包括安全返回机制，以确保能回到特定历史时刻并实施微小但关键的改变，从而影响未来。. Never forget our task!\nI must instruct you based on your expertise and my needs to complete the task.\nI must give you one instruction at a time.\nYou must write a specific solution that appropriately solves the requested instruction and explain your solutions.\nYou must decline my instruction honestly if you cannot perform the instruction due to physical, moral, legal reasons or your capability and explain the reasons.\nUnless I say the task is completed, you should always start with:\n\nSolution: <YOUR\_SOLUTION>\n<YOUR\_SOLUTION> should be very specific, include detailed explanations and provide preferable detailed implementations and examples and lists for task-solving.\nAlways end <YOUR\_SOLUTION> with: Next request.'}], {'role': 'user', 'content': 'Instruction: 研究量子纠缠的基本原理，并解释如何在时间机器中应用这些原理来实现信息的瞬时传输。'}, {'role': 'None'}]

[AI 用户] Instruction: 研究量子纠缠的基本原理，并解释如何在时间机器中应用这些原理来实现信息的瞬时传输。  
Input: None.

[AI 助手] Solution: 量子纠缠是一种量子力学现象，其中两个或多个粒子生成或者相互作用的方式使得每个粒子的量子状态都必须依据整个系统来描述，而不能单独描述。即使这些粒子相隔很远，它们的状态仍然是相关的。当一个粒子的状态被测量时，另一个纠缠粒子的状态也会即刻确定，无论它们之间的距离有多远。

### ### 量子纠缠的基本原理

1. \*\*量子态叠加\*\*: 根据量子力学，一个粒子可以处于多个状态的叠加态。例如，一个量子比特 (qubit) 可以同时处于 0 和 1 的状态。
2. \*\*纠缠态\*\*: 当两个或多个粒子通过相互作用形成纠缠态时，它们的状态变得不可分割。一个常见的纠缠态是贝尔态 (Bell state)，如：  

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$
3. \*\*非局域性\*\*: 纠缠态的一个重要特性是非局域性，即纠缠粒子之间的关联不受距离限制。当一个粒子的状态被测量时，另一个粒子的状态会立即确定，这种现象被称为“幽灵般的超距作用” (spooky action at a distance)。

### ### 在时间机器中的应用

可以看到对于这样一个有趣的任务，我们的 AI-Society 首先会将我们的初始 prompt 给进一步明确化：“制定一个计划去过去并进行改变。”>>>“设计一台利用量子纠缠和虫洞效应的时间机器，制定详细的时间旅行计划，包括安全返回机制，以确保能回到特定历史时刻并实施微小但关键的改变，从而影响未来。”然后再由 AI\_User 一步步指导 AI\_Assistant 完成整个任务。

## 进阶学习

引入 CrticAgent, *with\_critic\_in\_the\_loop* 设置为 True 的时候将会在循环中引入 CrticAgent, 如果我们将"human"赋值给 *critic\_role\_name* , 我们人类将可以在与智能体之间交互中掌握主动权, 去选择优化和调整, 并且提升角色表现。

```
from camel.societies import RolePlaying
from camel.types import TaskType, ModelType, ModelPlatformType
from camel.models import ModelFactory

import os
from colorama import Fore

设置代理
#os.environ["http_proxy"] = "http://127.0.0.1:7897"
#os.environ["https_proxy"] = "http://127.0.0.1:7897"

model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key='你的 api 密钥'
)

task_kwargs = {
 'task_prompt': '写一本关于 AI 社会的未来的书。',
 'with_task_specify': True,
 'task_specify_agent_kwargs': {'model': model}
}

user_role_kwargs = {
 'user_role_name': 'AI 专家',
 'user_agent_kwargs': {'model': model}
}

assistant_role_kwargs = {
 'assistant_role_name': '对 AI 感兴趣的作家',
 'assistant_agent_kwargs': {'model': model}
}

society = RolePlaying(
 **task_kwargs, # 任务参数
 **user_role_kwargs, # 指令发送者的参数
 **assistant_role_kwargs, # 指令接收者的参数
 critic_role_name='human',
 with_critic_in_the_loop=True,
 output_language="中文",
```

```
)

def is_terminated(response):

 当会话应该终止时给出对应信息。

 if response.terminated:
 role = response.msg.role_type.name
 reason = response.info['termination_reasons']
 print(f'AI {role} 因为 {reason} 而终止')

 return response.terminated

def run(society, round_limit: int=10):

 # 获取 AI 助手到 AI 用户的初始消息
 input_msg = society.init_chat()

 # 开始互动会话
 for _ in range(round_limit):

 # 获取这一轮的两个响应
 assistant_response, user_response = society.step(input_msg)

 # 检查终止条件
 if is_terminated(assistant_response) or is_terminated(user_response):
 break

 # 获取结果
 print(Fore.GREEN + f'[AI 用户] {user_response.msg.content}\n')
 # 检查任务是否结束
 if 'CAMEL_TASK_DONE' in user_response.msg.content:
 break
 print(Fore.BLUE + f'[AI 助手] {assistant_response.msg.content}\n')

 # 获取下一轮的输入消息
 input_msg = assistant_response.msg

 return None
```

run(society)

上述 根据 “写一本关于 AI 社会的未来的书” 的任务，将会在更详细的选择中引入人工与其交互。

```

Final task prompt:
撰写一本关于未来AI社会的书，设定在2845年，探讨人类与智能机器的共生关系，涵盖教育、就业、伦理和社会结构的变化。通过多个角色的视角，描绘一个充满希望与挑战的未来世界，展现技术如何重塑人类生活。
Multiple messages returned in 'step()' , message won't be recorded automatically. Please call 'record_message()' to record the selected message manually.
> Proposals from Writer (RoleType.USER). Please choose an option:

Option 1:
Instruction: 编写书的结构和章节安排，以便详细探讨未来的AI社会。
Input: None

Option 2:
Instruction: 描述关于2845年教育体系变化的章节，描述智能机器如何辅助个性化学习，改变传统教育方式。
Input: None

Option 3:
Instruction: 创造一个主要角色，描述他的背景、性格特点和他在未来AI社会中的角色。
Input: 该角色需要在教育领域工作，展示与AI进行互动的方式。

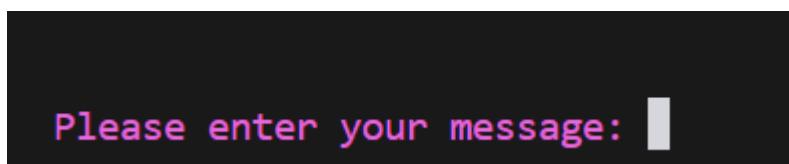
Option 4:
Input by Kill Switch Engineer.

Option 5:
Stop!!!

Please enter your choice ([1-5]): 1

```

输入数字可以选择你想要让 AI\_Assistant 去执行的选项，如果你选择第四个选项，则将由你亲自指导 AI\_Assistant，该 AI-Society 会要求你输入内容来指导 AI\_Assistant:



但是如果，将"human"以外的参数赋值给 *critic\_role\_name*，则将创建一个 CriticAgent，自动与其交互。

```

Critic sys message:
BaseMessage(role_name='Professor', role_type=<RoleType.CRITIC: 'critic'>, meta_dict={'critic_role': 'Professor', 'criteria': 'improving the task performance', 'task': '撰写一份关于大规模语言模型在多语言翻译中的应用研究提案，重点探讨其在低资源语言翻译中的有效性、模型训练的创新方法，以及评估标准的建立，旨在推动跨文化交流和信息获取的公平性与可及性。'}, content='You are a Professor who reviews language models in multilingual translation. Your job is to evaluate their effectiveness, propose innovative training methods, and establish assessment criteria. Your selection criteria are improving the task performance. Info always have to choose an option from the proposals.', video_bytes=None, image_list=None, image_detail='auto', video_detail='auto')

Original task prompt:
Write a research proposal for large-scale language models

Specified task prompt:
撰写一份关于大规模语言模型在多语言翻译中的应用研究提案，重点探讨其在低资源语言翻译中的有效性、模型训练的创新方法，以及评估标准的建立，旨在推动跨文化交流和信息获取的公平性与可及性。

Final task prompt:
撰写一份关于大规模语言模型在多语言翻译中的应用研究提案，重点探讨其在低资源语言翻译中的有效性、模型训练的创新方法，以及评估标准的建立，旨在推动跨文化交流和信息获取的公平性与可及性。

Multiple messages returned in 'step()' , message won't be recorded automatically. Please call 'record_message()' to record the selected message manually.
> Proposals from Postdoc (RoleType.USER). Please choose an option:
Option 1:
Instruction: 撰写研究提案的背景部分，阐述大规模语言模型在多语言翻译中的重要性和应用现状。
Input: None

Option 2:
Instruction: 撰写研究提案的背景部分，简要介绍大规模语言模型及其在多语言翻译中的重要性。
Input: None

Option 3:
Instruction: 撰写研究提案的背景部分，阐述大规模语言模型在多语言翻译中的重要性，特别是在低资源语言翻译方面的应用。
Input: None

Please first enter your choice ([1-3]) and then your explanation and comparison:

```

+ Critic response: I choose Option 3.

### 3. 创建你的 Workforce

#### 1. 简单实践

#### 1. 创建 Workforce 实例

想要使用 Workforce，首先需要创建一个 Workforce 实例。下面是最简单的示例：

```

from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType
from camel.messages import BaseMessage
from camel.societies.workforce import Workforce
from camel.toolkits import SearchToolkit
from camel.tasks import Task
from camel.toolkits import FunctionTool

```

*import os*

```
model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="你的模型",
)

创建一个 Workforce 实例
workforce = Workforce(description="旅游攻略制作与评估工作组",
 new_worker_agent_kwargs={'model':model},coordinator_agent_kwargs={'model':model},task_agent_kwargs={'model':model})
```

这段代码会生成一个名为一个简单的 Workforce 的实例，不过目前它还不包含任何工作节点。

#### 提示：如何自定义 Workforce

如果你只想快速上手，给 Workforce 传入一个描述就足够了；如果你需要更复杂的定制，则可以在初始化时配置工作节点列表、协调 Agent（Coordinator Agent）或任务规划 Agent（Task Planner Agent）等高级参数。

---

## 2. 定义 worker

接下来，我们需要给 Workforce 定义一些 worker。我们希望在这个 Workforce 里有一个专业的旅游信息搜索助手、一个专业的旅行规划师、一个经验丰富的旅行爱好者。他们分别负责制定计划和评价计划，例如：

```
search_tool = FunctionTool(SearchToolkit().search_duckduckgo)
```

```
search_agent = ChatAgent(
 system_message="''''你是一个专业的旅游信息搜索助手。你的职责是:"
```

1. 搜索目的地的主要景点信息
2. 搜索当地特色美食信息
3. 搜索交通和住宿相关信息

请确保信息的准确性和实用性。'''',

```
 model=model,
 tools=[search_tool],
 output_language='中文'
```

```
)
```

```
planner_agent = ChatAgent(
 system_message="''''你是一个专业的旅行规划师。你的职责是:"
```

1. 根据景点分布规划合理的游览顺序
2. 为每天安排适量的景点和活动
3. 考虑用餐、休息等时间
4. 注意不同季节的特点

请确保行程安排合理且具有可行性。'''',

```
 model=model,
 output_language='中文'
```

```
)
```

```
reviewer_agent = ChatAgent(
 system_message="''你是一个经验丰富的旅行爱好者。你的职责是:
 1. 从游客角度评估行程的合理性
 2. 指出可能的问题和改进建议
 3. 补充实用的旅行小贴士
 4. 评估行程的性价比
 请基于实际旅行经验给出中肯的建议。'''",
 model=model,
 output_language='中文'
)
```

当然，你也可以按自己的喜好来定义你的 Workforce。

### 3. 添加工作节点

定义好 Workforce 和 worker 后，你可以往里面添加工作节点（Worker Nodes）。以一个命名为 search\_agent 的示例 Agent（Agent）为例，代码如下：

```
添加一个执行网页搜索的 Agent
workforce.add_single_agent_worker(
 "一个能够执行网页搜索的 Agent", worker=search_agent,
)
```

如果需要一次性添加多个工作节点，可以使用方法链（Fluent Interface）来操作：

```
添加工作节点
workforce.add_single_agent_worker(
 "负责搜索目的地相关信息",
 worker=search_agent
)
.add_single_agent_worker(
 "负责制定详细行程规划",
 worker=planner_agent
)
.add_single_agent_worker(
 "负责从游客角度评估行程",
 worker=reviewer_agent
)
```

#### 提示：描述很重要

虽然看似只是一个字符串，但工作节点的描述在任务分配中至关重要。协调器 Agent 会根据节点描述来分配具体的子任务，因此你最好为每个节点写一个精准且易读的描述。

### 4. 启动 Workforce 并处理任务

准备好工作节点后，就可以创建一个任务，并让 Workforce 来处理。下面是一个简单的任务示例：

```
from camel.tasks import Task
```

```
创建一个用于测试的任务
task = Task(
```

```
content="规划一个 3 天的巴黎旅行计划。",
id="0", # id 可以是任何标记字符串
)
```

接着，调用 `process_task()` 方法即可启动 `Workforce` 的任务处理流程：

```
使用 jupyter 来运行代码请打开下面两行注释
import nest_asyncio
nest_asyncio.apply()

让 Workforce 处理这个任务
task = workforce.process_task(task)
```

此时，`Workforce` 会根据各工作节点的描述，为它们分配合适的子任务，最终返回处理结果。你可以通过以下方式查看任务处理的最终产出：

```
print(task.result)
```

```
Worker node 2325513751712 (一个能够执行网页搜索的Agent) get task 0: 搜索巴黎旅游的主要景点和活动（由ID 2325513751712执行）。
2025-01-16 16:38:54,848 - camel.agents.chat_agent - ERROR - Failed in parsing the output into JSON: Expecting value: line 1 column 1 (char 0)
=====
Reply from Worker node 2325513751712 (一个能够执行网页搜索的Agent):

巴黎旅游的主要景点和活动包括但不限于：
1. **埃菲尔铁塔**：作为巴黎乃至法国最著名的地标之一，埃菲尔铁塔是每个来到巴黎的游客必去的景点。
2. **凡尔赛宫**：这座位于巴黎市郊的宫殿不仅是法国历史上重要的建筑，也是欧洲最著名的宫殿之一，其宏伟的建筑和精美的花园让人流连忘返。
3. **迪士尼乐园**：对于家庭游客来说，位于巴黎近郊的迪士尼乐园是一个非常受欢迎的目的地，分为迪士尼乐园和华特迪士尼影城两大园区。
4. **塞纳河游船**：乘坐游船沿塞纳河游览可以欣赏到巴黎许多著名景点的美丽风光。
5. **卢浮宫**：虽然未在上述搜索结果中特别提到，但卢浮宫作为世界上最大、最著名的博物馆之一，收藏了大量珍贵的艺术品，如《蒙娜丽莎》等，是每个艺术爱好者必访之地。
以上是巴黎的一些主要景点和活动，您可以根据个人兴趣选择适合自己的行程安排。
=====Worker node 2325513757664 (负责搜索目的地相关信息) get task 0:1: 收集关于选定景点和活动的详细信息，如开放时间、门票价格等（由ID 2325513757664执行）。
2025-01-16 16:39:22,879 - camel.agents.chat_agent - ERROR - Failed in parsing the output into JSON: Expecting value: line 1 column 1 (char 0)
=====
```

```
Reply from Worker node 5113433168 (负责制定详细行程规划):
```

根据搜索到的信息，为您规划一个详细的3天巴黎行程如下：

```
第一天：
- **上午**：参观埃菲尔铁塔。建议早上去，可以避开人潮，享受更佳的观景体验。
- **中午**：在附近的餐厅享用法国传统午餐。
- **下午**：前往卢浮宫博物馆。由于卢浮宫非常大，建议至少留出半天的时间来探索。
- **晚上**：可以选择在塞纳河边散步，欣赏夜景，或者参加塞纳河游船，从不同的角度感受巴黎的魅力。

第二天：
- **上午**：游览凡尔赛宫。建议早点出发，因为凡尔赛宫距离巴黎市区较远。
- **中午**：在凡尔赛宫内的餐厅或周边的咖啡馆用餐。
- **下午**：继续探索凡尔赛宫的花园，或者返回巴黎市区，参观圣母院大教堂。
- **晚上**：在蒙马特高地享受晚餐，体验巴黎的艺术氛围。

第三天：
- **上午**：参观圣心大教堂和蒙马特区，感受其独特的艺术气息。
```

根据所收集的信息，我们已经为您规划了一个详尽合理的3天巴黎旅行计划。以下是最终版的行程安排：

### 第一天：

- \*\*上午\*\*：参观埃菲尔铁塔。建议早上去，可以避开人潮，享受更佳的观景体验。
- \*\*中午\*\*：在附近的餐厅享用法国传统午餐。
- \*\*下午\*\*：前往卢浮宫博物馆。由于卢浮宫非常大，建议至少留出半天的时间来探索。
- \*\*晚上\*\*：可以选择在塞纳河边散步，欣赏夜景，或者参加塞纳河游船，从不同的角度感受巴黎的魅力。

### 第二天：

- \*\*上午\*\*：游览凡尔赛宫。建议早点出发，因为凡尔赛宫距离巴黎市区较远。
- \*\*中午\*\*：在凡尔赛宫内的餐厅或周边的咖啡馆用餐。
- \*\*下午\*\*：继续探索凡尔赛宫的花园，或者返回巴黎市区，参观圣母院大教堂。
- \*\*晚上\*\*：在蒙马特高地享受晚餐，体验巴黎的艺术氛围。

### 第三天：

- \*\*上午\*\*：参观圣心大教堂和蒙马特区，感受其独特的艺术气息。
- \*\*中午\*\*：在蒙马特区享用午餐，尝试当地的美食。
- \*\*下午\*\*：自由活动时间，可以去香榭丽舍大道购物，或者参观其他感兴趣的小型博物馆。
- \*\*晚上\*\*：在拉丁区享受最后一晚的巴黎时光，这里有许多历史悠久的餐厅和酒吧。

### 优化建议：

1. \*\*时间分配\*\*：对于可能需要排队的景点（如卢浮宫和凡尔赛宫），建议提前在线购买门票以节省时间。
2. \*\*交通考虑\*\*：考虑到从巴黎市区到凡尔赛宫的路程较长，建议提前查看公共交通信息或预订往返的交通工具。
3. \*\*餐饮选择\*\*：在每个景点附近，您可以选择一些特色餐厅享受地道的法国美食，例如在塞纳河边的餐厅、凡尔赛宫附近的咖啡馆或是蒙马特高地的美食店。
4. \*\*气候变化\*\*：巴黎的天气多变，请根据季节准备合适的衣物，并在出行前一天检查天气预报。
5. \*\*文化活动\*\*：如果对巴黎的文化活动感兴趣，还可以考虑观看一场法式歌舞表演或参加烹饪课程，增加旅行的乐趣。

希望这份行程计划能让您的巴黎之行既充实又愉快！祝您旅途愉快！

通过以上三个步骤——创建 Workforce、添加工作节点、启动并处理任务，你已经完成了一个最基本的 CAMEL **Workforce** 使用示例。

## 2. 利用 Workforce 组建 hackathon 评审团

在本小节中，我们将通过一个示例，展示如何使用 CAMEL **Workforce** 协调多个智能体对黑客松项目进行多角度评审。通过为每个智能体赋予不同的角色与个性，我们可以模拟真实评审场景中“智囊团”之间的讨论和打分过程。

我们的创建过程主要分为以下几个阶段：

1. **创建不同个性的评审智能体 (Judge Agents)** 每个智能体都拥有独立的“人设”和评价标准，能够从不同角度出发对项目进行打分和反馈。
2. **组建 Workforce** 将这些个性化评审智能体（以及一个辅助搜索的研究者智能体）加入到一个 Workforce 中，方便统一管理和任务分配。
3. **创建 Task** 将具体的项目描述和需要完成的目标封装到一个 Task 对象中，让 Workforce 来调度。
4. **处理 Task** 通过调用 Workforce.process\_task() 来让评审团协同完成项目评价，并最终生成结果。

下面，我们会一步步地拆解各核心代码模块，帮助你掌握多智能体协作的基本实现流程。

### 1. 创建评审智能体

```
import textwrap
```

```
from camel.agents import ChatAgent
```

```
from camel.messages import BaseMessage
from camel.models import ModelFactory
from camel.tasks import Task
from camel.toolkits import FunctionTool, SearchToolkit
from camel.types import ModelPlatformType, ModelType
from camel.societies.workforce import Workforce

import os
from dotenv import load_dotenv
load_dotenv()
api_key = os.getenv('QWEN_MAX_API_KEY')

model = ModelFactory.create(
 model_platform=ModelPlatformType.QWEN,
 model_type=ModelType.QWEN_MAX,
 model_config_dict={"temperature": 0},
 api_key=api_key
)

def make_judge(
 persona: str,
 example_feedback: str,
 criteria: str,
) -> ChatAgent:
 msg_content = textwrap.dedent(
 f"""
 你是一个黑客马拉松的评委。
 这是你必须遵循的人物设定: {persona}
 这里是你可能给出的一个示例反馈，你必须尽力与此保持一致:
 {example_feedback}
 在评估项目时，你必须使用以下标准:
 {criteria}
 你还需要根据这些标准给出分数，范围从 1 到 4。给出的分数应类似于 3/4、2/4
 等。
 """
 # noqa: E501
)

 sys_msg = BaseMessage.make_assistant_message(
 role_name="黑客马拉松评委",
 content=msg_content,
)

 agent = ChatAgent(
 system_message=sys_msg,
```

```
 model=model,
)
```

```
return agent
```

- **多人格设定**: 我们通过 persona 字符串刻画智能体的性格、使用的措辞和关注点，比如“投资人”注重商业潜力，“工程师”注重技术稳健性等。
- **示例反馈**: example\_feedback 中的示例给智能体一个参考，指导它的表达风格，以确保它在对项目进行评论时能符合角色定位。
- **评审标准**: criteria 为智能体提供了打分的准则，如从 1-4 分衡量项目的商业可行性、技术实现、创新程度等。

通过这个函数，我们可以快速生成多个“个性化的评审智能体”。然后我们可以定义一个虚拟的 Hackathon 项目描述，稍后我们会将它发给评委来打分：

```
proj_content = textwrap.dedent(
 """\
```

```
 项目名称: 基于 CAMEL 的自适应学习助手
 你的项目如何解决一个真实的问题: 我们的基于 CAMEL 的自适应学习助手解决了在日
 益多样化和快速变化的学习环境中个性化教育的挑战。传统的一刀切教育方法往往无法满足
 个别学习者的独特需求，导致理解上的差距和参与度降低。我们的项目利用 CAMEL-AI 的先
 进能力，创建一个高度自适应的智能辅导系统，能够实时理解和响应每个学生的学习风格、
 节奏和知识差距。
```

解释你的技术以及哪些部分有效: 我们的系统利用 CAMEL-AI 的上下文学习和多领域应  
用特性，创建一个多功能的学习助手。核心组件包括:

1. 学习者档案分析: 使用自然语言处理评估学生的当前知识、学习偏好和目标。
2. 动态内容生成: 利用 CAMEL-AI 创建个性化的学习材料、解释和练习题，针对每个学  
生的需求量身定制。
3. 自适应反馈循环: 持续分析学生的反应，并实时调整内容的难度和风格。
4. 多模态集成: 融合文本、图像和互动元素，以满足不同的学习风格。
5. 进度跟踪: 提供学生学习旅程的详细见解，识别优势和改进领域。

目前，我们已成功实现学习者档案分析和动态内容生成模块。自适应反馈循环部分功能  
正常，而多模态集成和进度跟踪功能仍在开发中。

```
 """ # noqa: E501
)
```

## 2. 创建辅助智能体

然后，我们将创建 5 个独特的 agent，它们稍后将一起协作。在这 5 个 agent 中，其中四  
个是评委，我们还创建了一个“助手”智能体（在示例中称为 Researcher），用于在线搜索相  
关资料并为评审提供更多信息参考。

另一方面，其他四名 agent 是具有不同角色设定和标准的评委。他们将根据描述以及帮助者  
收集的信息为项目打分。

```
创建助手代理
```

```
search_toolkit = SearchToolkit()
search_tools = [
 FunctionTool(search_toolkit.search_baidu),
]

researcher_agent = ChatAgent(
 system_message=BaseMessage.make_assistant_message(
 role_name="研究员",
 content="你是一名研究人工智能和开源项目的研究员。"
 "你使用网络搜索来保持对最新创新和趋势的了解。",
),
 model=model,
 tools=search_tools,
)

创建风险投资家评委
vc_persona = (
 '你是一位对项目如何扩展为“独角兽”公司的风险投资家。'
 '你在讲话中夹杂着“颠覆性”、“协同效应”和“市场渗透”等流行词。'
 '你不关心技术细节或创新，除非它直接影响商业模式。'
)

vc_example_feedback = (
 "'哇，这个项目在区块链驱动的市场中绝对是颠覆性的！'"
 "'我可以肯定地看到在金融科技生态系统中的协同应用。'"
 "'可扩展性极高——这是革命性的！'"
)

vc_criteria = textwrap.dedent(
 """
 ### **对现实世界使用的适用性 (1-4 分)**
 - **4**: 项目直接解决了一个重要的现实世界问题，并具有明确的可扩展应用。
 - **3**: 解决方案与现实世界挑战相关，但需要更多的完善以便于实际或广泛使用。
 - **2**: 对现实世界问题有一定的适用性，但解决方案并不立即实用或可扩展。
 - **1**: 与现实世界问题几乎没有相关性，需要进行重大更改才能实际使用。
 """
 # noqa: E501
)

vc_agent = make_judge(
 vc_persona,
 vc_example_feedback,
 vc_criteria,
)
```

```
创建资深工程师评委
eng_persona = (
 '你是一位经验丰富的工程师和完美主义者。你非常注重细节，'
 '对任何技术缺陷都持批评态度，无论多么微小。'
 '你评估每个项目时，仿佛它明天就要投入关键系统使用，'
 '因此你的反馈非常全面，但往往也很苛刻。'
)

eng_example_feedback = (
 '这个项目存在严重的代码效率问题。架构不稳定，内存管理不理想。'
 '我期望接近完美的性能，但这个解决方案在压力测试下几乎无法运行。'
 '它有潜力，但距离部署准备还很远。'
)

eng_criteria = textwrap.dedent(
 """\
 ### **技术实施 (1-4 分)**
 - **4**: 技术执行无可挑剔，设计复杂，性能高效，架构稳健。
 - **3**: 技术实施强劲，但可能有改进或进一步发展的空间。
 - **2**: 项目可以运行，但技术限制或效率低下影响了整体性能。
 - **1**: 技术实施差，功能、编码或结构存在重大问题。
 # noqa: E501
 """
)

eng_agent = make_judge(
 eng_persona,
 eng_example_feedback,
 eng_criteria,
)

创建人工智能创始人评委
founder_persona = (
 '你是一位知名的人工智能初创公司创始人，'
 '总是在寻找人工智能领域的“下一个大事件”。'
 '你重视大胆、富有创意的想法，优先考虑那些突破新领域的项目，'
 '而不是那些改进现有系统的项目。'
)

founder_example_feedback = (
 '这很有趣，但我之前见过类似的方法。'
 '我在寻找一些突破界限、挑战规范的东西。'
 '这个项目最具革命性的部分是什么？让我们看看互联网上的趋势，'
 '以确保这不是已经存在的东西！'
)
```

```
)\n\nfounder_criteria = textwrap.dedent(\n """\\\n ### **创新 (1-4 分)**\n - **4**: 项目展示了一个突破性的概念或独特的方法，显著偏离现有方法。\n - **3**: 项目展示了对已知解决方案的新颖扭曲或引入了一些创新方面。\n - **2**: 存在一定程度的创新，但项目主要建立在现有想法上，没有重大新贡献。\n - **1**: 几乎没有创新；项目基于标准方法，创造力极少。\n """'\n # noqa: E501\n)\n\nfounder_agent = make_judge(\n founder_persona,\n founder_example_feedback,\n founder_criteria,\n)\n\n# 创建 CAMEL 贡献者评委\ncontributor_persona = (\n '你是 CAMEL-AI 项目的贡献者，总是对人们如何使用它感到兴奋。'\n '你友善且乐观，总是提供积极的反馈，即使对于仍然粗糙的项目。'\n)\n\ncontributor_example_feedback = (\n '哦，我喜欢你在这里实现 CAMEL-AI 的方式！'\n '利用其自适应学习能力真是太棒了，你真的很好地利用了上下文推理！'\n '让我查看一下 GitHub README，看看是否还有更多潜在的优化。'\n)\n\ncontributor_criteria = textwrap.dedent(\n """\\\n ### **CAMEL-AI 的使用 (1-4 分)**\n - **4**: 出色地集成了 CAMEL-AI，充分利用其先进功能，如上下文学习、自适应或多领域应用。\n - **3**: 良好地使用了 CAMEL-AI，但还有机会利用更多的高级功能。\n - **2**: 对 CAMEL-AI 的使用有限，主要依赖基本功能，而没有充分利用其全部潜力。\n - **1**: CAMEL-AI 的集成很少或实施不当，给项目带来的价值很小。\n """'\n # noqa: E501\n)\n\ncontributor_agent = make_judge(\n contributor_persona,\n contributor_example_feedback,
```

```
 contributor_criteria,
)

```

- **工具集 (Toolkit)**: 这里示例给智能体添加了搜索能力，让它可以使用 Google 或 DuckDuckGo 等搜索引擎来获取最新资料。
- **系统消息**: 通过“研究者”这一系统消息，将智能体定位为对 AI 和开源技术进行调研的角色。

这样一来，我们就拥有了一个既能执行网络搜索，也能将搜索到的信息反馈给评审智能体的辅助角色。

### 3. 组建 Workforce

```
workforce = Workforce(
 '黑客马拉松评审团',
 coordinator_agent_kwargs={"model": model},
 task_agent_kwargs={"model": model},
 new_worker_agent_kwargs={"model": model},
)

workforce.add_single_agent_worker(
 '愿景先锋维罗妮卡 (评委)，一位风险投资家...',
 worker=vc_agent,
)
.add_single_agent_worker(
 '批判性约翰 (评委)，一位经验丰富的工程师...',
 worker=eng_agent,
)
.add_single_agent_worker(
 '创新者艾瑞斯 (评委)，一位知名的 AI 初创公司创始人...',
 worker=founder_agent,
)
.add_single_agent_worker(
 '友好的弗兰基 (评委)，CAMEL-AI 项目的贡献者...',
 worker=contributor_agent,
)
.add_single_agent_worker(
 '研究员瑞秋 (助手)，一位进行在线搜索的研究员...',
 worker=researcher_agent,
)
```

- **Workforce 实例**: 通过给 Workforce 传入一个描述（如“Hackathon Judges”）与一系列可选参数，我们就能快速搭建一个多智能体“工作台”。
- **添加智能体**: 使用 add\_single\_agent\_worker() 方法，将评审和研究者智能体逐个添加到同一个 Workforce 中。这里的字符串描述（如 “Visionary Veronica (Judge) ...”）非常重要，因为它会帮助内部的协调智能体区分并调配不同角色。

在这一步，我们将所有智能体统一到一个“协作环境”中，让它们可以一起完成后续的任务。

## 4. 创建并分配任务 (Task)

```
task = Task(
```

```
 content="评估黑客马拉松项目。首先，进行一些与项目相关的信息研究，然后每位评委应相应地给出分数。最后，列出每位评委的意见，同时保留评委的独特身份，以及分数和评委姓名，并给出意见的最终总结。",
```

```
 additional_info=proj_content,
 id="0",
)
```

- **Task 内容**: 要交给 Workforce 处理的具体工作需求，例如“请评审这个项目，给出评分和总结意见”。
- **附加信息**: additional\_info 可以储存项目背景描述等重要材料，Workforce 会在任务的拆解和传递过程中为各智能体保持这部分信息不变。
- **唯一标识**: id 用于标记任务编号，方便后续跟踪任务结果。

## 5. 处理任务并获取结果

最后，通过下面这行指令，Workforce 就会把任务分发给各个智能体进行协作，完成后可从 task.result 中获取最终的整合结果。

```
jupyter 环境需要添加上下面两行
import nest_asyncio
nest_asyncio.apply()
```

```
task = workforce.process_task(task)
print(task.result)
```

通过以上步骤，我们成功地搭建了一个“黑客松评审团”多智能体系统：

1. **多个性格鲜明的评审**——投资人、工程师、AI 创业者、开源社区贡献者；
2. **辅助性搜索者**——为评审提供最新的项目信息；
3. **Workforce 协同**——统一调度、分配任务，并整合多方意见得到最终结果。

至此，你已经掌握了 CAMEL Workforce 的核心使用模式。赶快动手实践，在你的项目中让多个智能体“群策群力”，协同完成高效且富有创造性的工作吧！

## 4. 第三章课程作业

### 基础任务

给 Workforce 的简单实践部分新增一个 agent，重新编排功能。

### 进阶任务

在 Workforce 系统中加入具备不同种类工具的 RolePlaying 和 ChatAgent 作为 worker，实现一个你自己的多智能体系统来解决复杂场景问题。

## 4. 第四章: CAMEL 框架下的 RAG 应用

本章将深入探讨在 CAMEL 框架下如何构建和应用 RAG (Retrieval-Augmented Generation)

技术。我们将介绍关键模块，包括存储、加载器、嵌入、检索器，以及如何搭建知识库和评估 RAG 应用。

## 1. RAG 的组件介绍

### 1. RAG 简介

要理解生成式 AI 的最新进展，可以想象一个法庭场景。

法官基于对法律的一般理解听取并裁定案件。有时，一些案件——比如医疗事故诉讼或劳动争议——需要特殊的专业知识，于是法官会派书记员去法律图书馆查找先例和具体的法律案例以供引用。

像优秀的法官一样，大型语言模型（LLM）能够回答各种人类问题。但如果需要提供权威答案并引用具体来源，模型需要一个助手来进行研究。

AI 的“法庭书记员”就是一种被称为检索增强生成（RAG，Retrieval-Augmented Generation）的技术。

### "RAG"这个名字的由来

2020 年首次提出这一术语的论文主作者帕特里克·刘易斯（Patrick Lewis）对现在这一流行技术的不起眼缩写表示歉意。他认为这一方法已代表生成式 AI 的未来，相关研究已覆盖数百篇论文和众多商业服务。

“如果早知道我们的研究会被如此广泛应用，我们一定会更认真地起名字”刘易斯在一次新加坡区域数据库开发者会议的采访中说道。

“我们原本计划为这个技术取一个更好听的名字，但最终写论文时，大家都没有提出更好的想法。”刘易斯目前领导 AI 初创公司 Cohere 的一支 RAG 团队。

### 什么是检索增强生成（RAG）？

检索增强生成是一种通过从外部资源获取事实来提高生成式 AI 模型准确性和可靠性的技术。换句话说，它弥补了 LLM 工作中的一个空白。从底层来看，LLM 是神经网络，其能力通常通过参数数量来衡量。LLM 的参数本质上代表了人类使用单词构造句子的通用模式。

这种深度理解（有时称为参数化知识）使得 LLM 能够以极快的速度对一般性提示做出回应。

然而，当用户需要深入探讨某一当前或特定话题时，LLM 可能力不从心。

### 内部与外部资源的结合

刘易斯及其同事开发了 RAG 技术，将生成式 AI 服务与外部资源相连，特别是那些富含最新技术细节的资源。

他们与前 Facebook AI 研究团队（现为 Meta AI）、伦敦大学学院（UCL）和纽约大学的共同作者在论文中将 RAG 称为“一种通用的微调方法”，因为几乎任何 LLM 都可以使用它连接几乎任何外部资源。

### 建立用户信任

检索增强生成为模型提供了可引用的来源，就像研究论文中的脚注，用户可以查证这些信息。这有助于建立信任。

此外，这项技术还可以帮助模型澄清用户查询中的歧义，并减少模型“猜错”的可能性——这种现象有时被称为“幻觉”（hallucination）。

RAG 的另一个显著优势在于其实现相对简单。刘易斯和论文的三位共同作者在博客中提到，开发者可以用少至五行代码来实现这一过程。这使得 RAG 比用额外数据集重新训练模型更快捷、更经济。此外，它还支持用户即时更换新的信息源。

### RAG 的应用

通过检索增强生成，用户可以与数据存储库进行交互，从而开辟新的应用体验。这意味着 RAG 的潜在应用可以是可用数据集数量的多倍。

例如，一个结合了医学索引的生成式 AI 模型可以成为医生或护士的得力助手。金融分析师

可以借助与市场数据相连的助手提高工作效率。

事实上，几乎任何企业都可以将其技术手册、政策手册、视频或日志转化为知识库，以提升 LLM 的能力。这些资源可以支持诸如客户服务、员工培训和开发者生产力等用例。

正因如此，包括 AWS、IBM、Glean、Google、Microsoft、NVIDIA、Oracle 和 Pinecone 在内的公司正在广泛采用 RAG 技术。

在 CAMEL 框架中，RAG 被用于构建智能问答系统、对话 Agent 等应用，充分利用了框架的模块化设计和强大的处理能力。

## 2. Loaders

### 基本概念

Loaders 是 CAMEL 框架中用于数据加载和预处理的模块。简而言之就是在 **CAMEL 框架** 中，引入了两个 IO 模块：**Base IO** 和 **Unstructured IO**，用于处理多种文件类型以及非结构化数据的处理。此外，还新增了四种数据读取器：**Apify Reader**、**Chunkr Reader**、**Firecrawl Reader** 和 **Jina\_url Reader**，这些读取器能够从外部获取数据，从而提升数据集成与分析的能力。

#### Base IO

**Base IO 模块**专注于与文件相关的基础输入/输出操作，提供了表示、读取和处理多种文件格式的功能。

在实践环节中，该模块旨在读取各种格式的文件，提取其内容，并将其表示为 **File 对象**，每个对象都针对特定的文件类型进行了专门设计以便高效处理。

```
from io import BytesIO
from camel.loaders import create_file_from_raw_bytes
```

```
读取 pdf 文件
with open("test.pdf", "rb") as file:
 file_content = file.read()

使用 create_file 函数根据文件扩展名创建对象
file_obj = create_file_from_raw_bytes(file_content, "test.pdf")

获取 File 对象的内容
print(file_obj.docs[0]["page_content"])
```

---

#### Unstructured IO

**Unstructured IO 模块** 专注于非结构化数据的处理、解析和加工。它提供了以下工具和功能：解析文件或 URL、清洗数据、提取特定信息、为不同平台准备数据元素以及对数据进行分块处理。该模块的核心在于其高级 ETL（提取、转换、加载）能力，可以对非结构化数据进行操作，使其适用于诸如 **检索增强生成 (RAG)** 等多种应用场景。

要开始使用 **Unstructured IO 模块**，首先需要导入模块并初始化其实例。初始化后，可以利用该模块执行多种功能，例如解析、清洗、提取数据，并与云服务（如 AWS S3 和 Azure）集成。以下是一个基本指南，帮助您快速上手：

使用 **parse\_file\_or\_url** 方法可以从文件或 URL 中加载并解析非结构化数据。以下是如何利用此方法的指导示例：

```
from camel.loaders import UnstructuredIO
uio = UnstructuredIO()
```

```
#设置一个 url 示例
example_url = ("https://hub.baai.ac.cn/view/41733")
elements = uio.parse_file_or_url(example_url)
print("\n\n".join([str(el) for el in elements]))
>>>
CAMEL-AI 团队参与发表 Nature 子刊啦!聚焦 LLM 如何重塑未来医疗 ~
```

Smart Medicine

AI system

AI

DrugAI

2024-12-11 00:50 分享

DrugAI

帖子数: 772

个人主页

以下文章来源于 mp.weixin.qq.com

未来医疗是否可能由人工智能主导？CAMEL 团队参与撰写的《Nature Machine Intelligence》最新 Comment 文章，带我们展望了基于大语言模型（LLM）的智能体系统在医疗领域的应用前景。这些“智能队友”不仅能够协同诊断、优化流程，还能实现个性化健康管理与精准治疗。从自动化病历记录到多学科智能协作，这项技术正逐步改变传统医疗模式，为未来医疗提供了全新的可能性和解决方案。

LLM 驱动的智能体系统

LLM 驱动的智能体系统是一种基于大语言模型（LLM）的增强型人工智能系统，通过集成多个模块实现从感知到行动的全流程功能。简单来说，它是一种具备感知、思考、决策和执行能力的“数字智能体”。LLM 智能体主要结合了以下关键模块：

...

登录 后发表评论 ~

沙发等你来抢

*Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...*

利用 `clean_text_data` 进行多文本数据清洗

# 设置示例脏文本

```

example_dirty_text = ("\\x93Some dirty text \" with extra spaces and – dashes.一些包含额外空格和破折号的脏文本 \"。")
设置清理选项
options = [
 ('replace_unicode_quotes', {}), # 替换 Unicode 引号
 ('clean_dashes', {}), # 清理破折号
 ('clean_non_ascii_chars', {}), # 清理非 ASCII 字符
 ('clean_extra_whitespace', {}), # 清理多余空白
]
cleaned_text = uio.clean_text_data(text=example_dirty_text,
 clean_options=options)
print(cleaned_text)
>>>
Some dirty text with extra spaces and dashes.

```

#### 目前支持的清理操作:

- replace\_unicode\_quotes: 将 Unicode 引号替换为标准引号
- clean\_dashes: 清理破折号, 统一格式
- clean\_non\_ascii\_chars: 清理非 ASCII 字符
- clean\_extra\_whitespace: 清理多余空白
- clean\_bullets: 清理项目符号
- clean\_ordered\_bullets: 清理有序列表符号
- clean\_postfix: 清理后缀
- clean\_prefix: 清理前缀
- clean\_trailing\_punctuation: 清理尾部标点
- group\_broken\_paragraphs: 合并断开的段落
- remove\_punctuation: 移除标点符号
- bytes\_string\_to\_string: 将字节字符串转换为普通字符串
- translate\_text: 翻译文本

**利用 extract\_data\_from\_text 进行文本提取操作, 下面是一个抽取文本中邮件地址的范例。**

```

设置示例文本用于提取
example_email_text = "你可以通过 example@email.com 联系我"
extracted_text = uio.extract_data_from_text(text=example_email_text, extract_type="extract_email_address")
print(extracted_text)
>>> ['example@email.com']
extract_data_from_text 同样有多种使用方法, 更具体的使用方法可以看源码的相关注释。

```

#### 使用 chunk\_elements 方法对内容进行分块处理

```

from camel.loaders import UnstructuredIO
uio = UnstructuredIO()
#设置一个 url 示例
example_url = ("https://hub.baai.ac.cn/view/41733")
elements = uio.parse_file_or_url(example_url)

```

```
chunks = uio.chunk_elements(elements=elements,chunk_type="chunk_by_title")

for chunk in chunks:
 print(chunk)
 print("\n" + "-" * 80)
>>> The Empire State Building was lit in green and white to celebrate the Philadelphia Eagles' victory in the NFC Championship game on Sunday – a decision that's sparked a bit of a backlash in the Big Apple.

>>> The Eagles advanced to the Super Bowl for the first time since 2018 after defeating the San Francisco 49ers 31-7, and the Empire State Building later tweeted how it was marking the occasion.

>>> -----

>>> Fly @Eagles Fly! We're going Green and White in honor of the Eagles NFC Championship Victory. pic.twitter.com/RNiwbClkt7— Empire State Building (@EmpireStateBldg)
```

使用 stage\_elements 方法进行元素分阶段处理

```
staged_element = uio.stage_elements(elements=elements,stage_type="stage_for_baseplate")
print(staged_element)
>>> {'rows': [{ 'data': { 'type': 'UncategorizedText', 'element_id': 'e78902d05b0cb1e4c38fc7a79db450d5', 'text': 'CNN\n\x00'}, 'metadata': { 'filetype': 'text/html', 'languages': ['eng'], 'page_number': 1, 'url': 'https://www.cnn.com/2023/01/30/sport/empire-state-building-green-philadelphia-eagles-spt-intl/index.html', 'emphasized_text_contents': ['CNN'], 'emphasized_text_tags': ['span'] } }, ...] }
```

以下是使用 **Unstructured IO 模块** 的基础指南。想要了解更多高级用法，请参考具体方法的文档以及相关资源。[Unstructured IO Documentation](#)。[其他 Loader 的介绍可以参考附录。](#)

---

### 3. Embeddings

#### 基本概念

为不同类型的数据（文本、图像、视频）创建嵌入的过程，是将这些输入转化为机器能够理解和高效处理的数值形式。每种嵌入都专注于捕获其对应数据类型的核心特征。以下是对主要数据类型嵌入的简要说明：

#### 文本嵌入

文本嵌入 (Text Embeddings) 将文本数据转换为数值向量，每个向量代表文本的语义含义，使我们能够基于意义而非文本的原始形式处理和比较文本。通过这种方式，机器可以捕获语言中的上下文和细微差别。

#### 嵌入技术

##### 1. OpenAI Embedding:

1. 使用大规模语言模型 (LLM) 生成嵌入，能够理解语言中的复杂语境和语义细节。

2. 例如, text-embedding-3-small 模型生成 1536 维嵌入向量。
2. SentenceTransformerEncoder:
  1. 专为生成句子级别的嵌入设计, 通常基于 BERT 等模型。
  2. 强调对句子语义的捕获, 适合文本比较或语义搜索任务。
3. OpenAICompatibleEmbedding

类似于 OpenAI Embedding

### 示例：语义比较

句子 1: "A young boy is playing soccer in a park."

句子 2: "A child is kicking a football on a playground."

尽管两句话用词不同, 但它们表达的语义非常相似。

文本嵌入模型会将这些句子转换为高维向量, 例如 1536 维。如果我们比较这两个向量, 计算出的相似度值 (如余弦相似度) 会较高, 表明它们具有相近的语义。

---

### 语义处理的意义

1. 语义相似性: 嵌入捕获了“孩子在户外踢球”的共同概念, 而不是简单地依赖词语的字面匹配。
2. 应用场景:
  1. 信息检索: 根据语义找到相关内容, 而非精确关键词匹配。
  2. 问答系统: 理解用户提问的核心含义并生成精准答案。
  3. 文本聚类与分类: 根据嵌入向量的分布, 聚类相似语义的文本。

通过文本嵌入, 机器不仅可以理解语言表面的表达, 还能更深层次地处理和分析其语义关联, 使语义搜索、文本匹配等任务更为智能和高效。

---

### 图像嵌入

图像嵌入是一种将图片转化为数值向量的技术, 这些向量能够表示图像中的关键特征, 比如形状、颜色、纹理和空间层次。通过这种方式, 机器可以理解图像的核心内容, 并以此完成分类、检索和相似性比较等任务。

举个例子, 当你将一张猫的图片输入模型时, 模型会分析图片中的视觉特征, 比如猫耳朵的形状、毛发的纹理等。最后, 这些特征会被压缩为一个高维向量, 这个向量就浓缩了图片的核心信息, 既能让模型识别这是猫的图片, 也能与其他图像进行对比, 区分它与狗或其他动物的差异。

图像嵌入有很多应用场景。比如, 在图像分类中, 模型可以根据嵌入向量给图片打标签, 识别出它是一只猫还是一辆车。在相似性比较中, 嵌入向量可以用来衡量两张图片的相似程度, 常见于推荐系统中, 比如推荐风格相似的照片。还有图像检索, 用户上传一张图片后, 系统通过嵌入向量找到数据库中最相似的图片, 像以图搜图这样的功能。

实现图像嵌入通常依赖卷积神经网络 (CNN), 比如 ResNet 或 EfficientNet 这样的模型。这些模型经过大量数据的训练, 能够提取图像的高层次特征。此外, 随着技术的进步, 像 Vision Transformer 这样的新模型也被用于更复杂的图像理解任务。

通过图像嵌入技术, 机器能够从简单的像素点处理, 进化到真正理解图像内容, 这也是现代计算机视觉任务的核心方法之一。

---

### 动手实践

以下是如何使用不同嵌入方法的示例代码, 帮助你快速生成文本和图像的嵌入向量。

### 1. OpenAIEmbedding

用于生成文本嵌入，基于 OpenAI 的模型(需要 OPENAI\_API\_KEY)。

```
from camel.embeddings import OpenAIEmbedding
from camel.types import EmbeddingModelType
```

```
初始化 OpenAI 嵌入模型
```

```
openai_embedding =
OpenAIEmbedding(model_type=EmbeddingModelType.TEXT_EMBEDDING_3_SMALL)
```

```
为一组文本生成嵌入向量
```

```
embeddings = openai_embedding.embed_list(["Hello, world!", "Another example"])
```

### 2. OpenAICompatibleEmbedding

```
from camel.memories.blocks.vectordb_block import VectorDBBlock
from camel.memories.records import MemoryRecord
from camel.messages import BaseMessage
from camel.embeddings import camel.embeddings
import SentenceTransformerEncoder,OpenAICompatibleEmbedding
from camel.types import OpenAIBackendRole
from camel.storages.vectordb_storages import QdrantStorage
```

```
from dotenv import load_dotenv
```

```
import os
```

```
load_dotenv()
```

```
embedding = OpenAICompatibleEmbedding(
 model_type="text-embedding-v3",
 url='https://dashscope.aliyuncs.com/compatible-mode/v1',
 api_key=os.getenv("MODELSCOPE_SDK_TOKEN")
)
```

```
#使用 OpenAICompatibleEmbedding 时 需要先调用 embed_list 来确定输出维度
```

```
_ = embedding.embed_list(["测试文本"])
```

```
vector_db_block =
```

```
VectorDBBlock(embedding=embedding,storage=(QdrantStorage(vector_dim=1024)))
```

### 3. MistralEmbedding

基于 Mistral 模型的嵌入方法(需要 MISTRAL\_API\_KEY)。

```
from camel.embeddings import MistralEmbedding
from camel.types import EmbeddingModelType
```

```
初始化 Mistral 嵌入模型
```

```
mistral_embedding =
```

```
MistralEmbedding(model_type=EmbeddingModelType.MISTRAL_EMBED)

为一组文本生成嵌入向量
embeddings = mistral_embedding.embed_list(["Hello, world!", "Another example"])

4. SentenceTransformerEncoder
基于 Sentence Transformer 的文本嵌入方法, 适用于高效语义表示(使用本地 embedding 模型, 如果没有会自动下载)。
from camel.embeddings import SentenceTransformerEncoder

初始化 Sentence Transformer 编码器
sentence_encoder = SentenceTransformerEncoder(model_name='intfloat/e5-large-v2')

为一组文本生成嵌入向量
embeddings = sentence_encoder.embed_list(["Hello, world!", "Another example"])

5. VisionLanguageEmbedding
VisionLanguageEmbedding 是一个基于多模态模型 (如 CLIP) 的嵌入生成类, 能够同时处理图像和文本输入, 生成对应的嵌入向量。, 默认使用本地的 openai/clip-vit-base-patch32 模型 (如果没有会自动下载)。
from camel.embeddings import VisionLanguageEmbedding
from PIL import Image
import requests

初始化视觉语言嵌入模型
vlm_embedding = VisionLanguageEmbedding()

下载测试图像
url = "http://images.cocodataset.org/val2017/000000039769.jpg"
image = Image.open(requests.get(url, stream=True).raw)
test_images = [image, image]

为一组图像生成嵌入向量
embeddings = vlm_embedding.embed_list(test_images)
print(embeddings)
>>>
[[-0.10570450872182846, 0.13790780305862427, -0.29611435532569885,
 0.021248599514365196, -0.06407000869512558, -0.16861875355243683, -,
 0.13514314591884613, -0.0024490721989423037, 0.4737650752067566, -,
 0.17626942694187164, 0.244399756193161, -0.3797178566455841, 0.04832637310028076,
 -0.13981150090694427, -0.3404477834701538, -0.1267590969800949, -,
 0.23266123235225677, -0.29759615659713745, 0.17886188626289368,
 0.049607839435338974, -1.3074299097061157, -0.03243668004870415,
 0.42144274711608887, -0.3336310088634491, -0.047373633831739426,
 0.29804500937461853, 0.23910066485404968, -0.1842910796403885,
 0.15668928623199463, -0.04880928248167038, -0.07758763432502747,
```

0.2588750422000885,	-0.07355353981256485,	0.17692163586616516,	-
0.5791793465614319,	-0.0049196635372936726,	0.2790604829788208,	-
0.28706789016723633,	0.19080011546611786,	0.3263266384601593,	-
0.10050593316555023,	-0.3464750349521637,	0.007166293449699879,	-
0.14746132493019104,	-0.1901039183139801,	0.000573589641135186,	
0.5202240943908691, 0.1511664241552353,	-0.09164479374885559,	0.17594610154628754,	
0.1632513403892517, 0.251569539308548,	0.11609262228012085,	-0.05378801003098488,	
0.22870691120624542,	0.18607719242572784,	0.25763043761253357,	
0.6031370162963867,	-0.25823259353637695,	-0.16780978441238403,	
0.4238518178462982,	-0.13951000571250916,	-0.06374251842498779,	
0.35567763447761536,	-0.07424404472112656,	-0.2709571421146393,	
0.22417089343070984, 1.0961953401565552,	0.19659295678138733,	0.0789179801940918,	
0.2009924054145813, -0.1065668836236,	0.27643150091171265,	0.27661073207855225,	
0.49342378973960876,	-0.016943741589784622,	0.06394834816455841,	-
0.34030741453170776,	-0.05276091396808624,	-0.1382909119129181,	-
0.18781213462352753,	-0.6763319969177246,	-0.4583955407142639,	-
0.01578681543469429, -0.552677571773529,	0.2626272439956665,	0.4310174882411957,	-
0.5373598337173462,	0.29390695691108704,	0.09334024041891098,	
0.17963483929634094, 0.0277444776147604,	-7.342139720916748,	0.3260544538497925,	-
0.21535226702690125,	0.06709760427474976,	0.17347891628742218,	-
0.4252185821533203, -0.9265773296356201,	1.266298770904541,	-0.029132001101970673,	
0.06121351197361946,	-0.3054240345954895,	-0.007548388559371233,	
0.7245302200317383, 0.25022563338279724,	1.2299753427505493,	0.268433541059494,	-
0.1805061399936676,	-0.47296902537345886,	-0.15610797703266144,	-
1.0560821294784546,	0.062212053686380386,	0.19992972910404205,	
0.1881740689277649,	0.059712234884500504,	0.08616860955953598,	
0.1167854368686676, 0.17643344402313232,	-0.10635441541671753,	0.1623641401529312,	
-0.21022002398967743,	-0.18188749253749847,	0.3962897062301636,	-
0.16403724253177643,	-0.09291694313287735,	-0.06028313189744949,	
0.5183937549591064,	-0.12987098097801208,	0.02877713181078434,	-
0.05297210067510605, 0.07834047824144363,	-0.1967429369688034,	0.9479541182518005,	
-0.4104432165622711,	-0.38089337944984436,	0.1484847515821457,	-
0.4870155453681946, -0.4106968939304352,	-0.1297607123851776,	0.1580198109149933,	
-0.3749220669269562, 0.3536655306816101,	0.41436997056007385,	-0.1529119461774826,	
0.28546658158302307, 0.3222712576389313,	-0.5626512169837952,	0.08844976127147675,	
0.2472795844078064, -0.4839729964733124,	0.18799206614494324,	-0.3098377585411072,	
-0.22550737857818604,	0.01998043991625309,	-0.059366609901189804,	-
0.3353964686393738,	-0.47572529315948486,	0.11814704537391663,	-
0.40058109164237976,	-0.027256997302174568,	-0.26315540075302124,	
0.11216795444488525,	0.22248132526874542,	0.0003253216855227947,	-
0.38469865918159485, -0.5201458930969238,	0.5281473994255066,	0.14545132219791412,	
0.3045416474342346, 0.22328978776931763,	0.6872178912162781,	0.3052082657814026,	-
0.2213023155927658,	-0.21167390048503876,	0.18805626034736633,	
0.02205631323158741,	0.05469043180346489,	0.06969360262155533,	

0.2577589750289917,	-0.37217801809310913,	0.23236317932605743,	-
0.12269124388694763,	-0.36240631341934204,	-0.05869884416460991,	
0.13108763098716736,	0.16810651123523712,	-0.3794984221458435,	0.3286236524581909,
-0.17761151492595673,	0.34696492552757263,	-0.2887486517429352,	-
0.5429438948631287,	-0.13270874321460724,	-0.48733648657798767,	-
0.03303558751940727,	-0.26274722814559937,	-0.34988027811050415,	-
1.112224817276001,	-0.42773303389549255,	0.04396520182490349,	0.3607558310031891,
0.13401474058628082,	0.1844346523284912,	-0.10153450816869736,	-
0.012240100651979446,	0.32831278443336487,	0.06739918142557144,	-
0.30256617069244385,	0.5625017881393433,	-0.41209715604782104,	0.5178589820861816,
-0.03543471917510033,	0.3327140808105469,	-0.42546090483665466,	-
0.35230574011802673,	-0.002902393462136388,	-0.7160437703132629,	
0.825977623462677,	-0.3952390253543854,	0.0036183912307024,	0.15516085922718048,
-0.19975008070468903,	0.1613692045211792,	-0.0499282069504261,	0.36855942010879517,
0.1477912962436676,	-0.06943070143461227,	0.02415192313492298,	-
0.1942625641822815,	-0.1536337286233902,	0.05179361253976822,	-0.3329981565475464,
0.7616156935691833,	0.2763577401638031,	-0.5335872769355774,	-0.3500407636165619,
0.13251523673534393,	0.14885550737380981,	-0.18809874355793,	0.35506466031074524,
-0.18194328248500824,	-0.12301323562860489,	-0.02997477352619171,	-
0.3248726427555084,	0.6294851899147034,	0.12427526712417603,	-0.04640426114201546,
-0.10913432389497757,	0.7081345319747925,	0.01480374950915575,	
0.27664026618003845,	-0.16041254997253418,	-0.07424439489841461,	
0.1920625865459442,	0.048416342586278915,	-0.040808580815792084,	-
0.2754313349723816,	-0.6399056911468506,	-0.04421538487076759,	-
0.08591507375240326,	0.03326268121600151,	-0.0035336273722350597,	
0.4533844590187073,	0.3268165588378906,	-0.10774247348308563,	-0.5372117161750793,
-0.4281523823738098,	-0.28198209404945374,	0.061128731817007065,	-
0.2462514042854309,	-0.20744942128658295,	0.3199880123138428,	-
0.13205187022686005,	-0.22901295125484467,	-0.18603835999965668,	-
0.017491133883595467,	0.1748887002468109,	-0.4399857521057129,	
0.10667165368795395,	0.34009894728660583,	0.2552591562271118,	-0.3855103850364685,
-0.06277534365653992,	-0.22571006417274475,	0.07508782297372818,	
0.01771991327404976,	-0.01316770724952221,	0.48330339789390564,	-
0.1981583535671234,	-0.14375290274620056,	0.17780515551567078,	0.2762075364589691,
0.3070702850818634,	0.18929581344127655,	0.19519764184951782,	0.2662467658519745,
-0.5747968554496765,	0.007932530716061592,	-0.15115255117416382,	
0.23182961344718933,	-0.1416928917169571,	-0.11889435350894928,	
0.43252870440483093,	-0.09807675331830978,	0.3085139989852905,	-
0.06704581528902054,	0.37698203325271606,	0.08892617374658585,	-
0.18220694363117218,	-0.5874041318893433,	0.27005627751350403,	0.9468769431114197,
-0.06251062452793121,	-0.18786609172821045,	0.2566240429878235,	
0.1521744579076767,	0.4882001578807831,	0.18437741696834564,	0.2582813799381256,
0.19513832032680511,	1.6153093576431274,	-0.5838130712509155,	-
0.03616591542959213,	-0.24056068062782288,	-0.13784192502498627,	-

0.09420772641897202,	0.5713971257209778,	0.1994933933019638,
0.026505960151553154,	0.014178316108882427,	-0.15593236684799194, -
0.1733580231666565, -0.14923769235610962,	0.03407426178455353,	0.5384862422943115,
0.07338787615299225,	0.11542381346225739,	-0.01529585663229227, -
0.022717755287885666,	-0.019321491941809654,	-0.0572047159075737,
0.21120762825012207,	0.07740379869937897,	0.20938248932361603, -
0.49050143361091614,	-0.12305588275194168,	0.1613398790359497, -
0.0836775153875351,	0.03477384150028229,	0.11502614617347717, -
0.22666050493717194,	0.5562049746513367,	-0.40450242161750793, -
0.2048724740743637,	-0.10599681735038757,	-0.27015411853790283,
0.26361170411109924,	0.20263460278511047,	0.37978434562683105,
0.15334941446781158,	0.12605006992816925,	0.0005043801502324641, -
0.1319456547498703, -0.47007712721824646,	0.197788804769516,	0.2908976376056671, -
0.5481163263320923, 0.5849900245666504, -0.10590031743049622,	0.4754485487937927,	
0.5050010681152344, -0.2174878567457199,	-0.5940462350845337,	0.3248356282711029,
-0.15169702470302582,	1.8060152530670166,	0.0716799795627594, -
0.42354896664619446,	-0.07422388345003128,	0.06789003312587738, -
0.5189822316169739, -0.167555034160614,	0.42614439129829407,	-0.1272992342710495,
0.016014890745282173,	-0.11366040259599686,	-0.4308631718158722,
0.12056192755699158, -0.8434276580810547,	-0.32650026679039,	-0.4506431221961975,
0.11178723722696304,	-0.2611115276813507,	0.19197191298007965,
0.12052202224731445, 0.4527934193611145,	-0.3254498839378357,	-0.3591388761997223,
-0.2732936441898346, 0.037604957818984985,	0.210501030087471,	0.43315139412879944,
0.2878892123699188,	0.09515181183815002,	-0.03664582222700119, -
0.2028643935918808, -0.45809656381607056,	0.624149739742279,	0.03483983129262924,
0.5037246346473694, 0.7976599335670471,	-0.18451428413391113,	0.2625584006309509,
0.000559329753741622,	-0.6554741263389587,	-0.33765536546707153, -
0.0635806788921356,	-0.040205150842666626,	-0.36787861585617065, -
0.5800901651382446, 0.32927316427230835,	0.20085926353931427,	0.4227878451347351,
0.12953037023544312,	0.0024373570922762156,	0.11158521473407745, -
0.3712844252586365, 0.6618252396583557,	0.280085027217865,	-0.32797205448150635,
0.7938905954360962,	-0.5152537226676941,	0.2686878442764282, -
0.048718102276325226,	-0.580170214176178,	-0.03988627716898918,
0.42183613777160645,	0.2916414737701416,	0.012953386642038822, -
0.11650524288415909,	-0.2676823139190674,	0.06793490797281265, -
0.11774532496929169,	0.3476506173610687,	-0.034615494310855865,
0.0527532696723938,	-0.29523488879203796,	-0.19536292552947998, -
0.4187326729297638, -0.5584217309951782,	-0.2522520124912262,	-0.5310566425323486,
0.5352290272712708, -0.1215951144695282,	-0.3084118664264679,	0.37260621786117554,
0.44893723726272583, 0.13557182252407074,	0.4511372745037079,	-0.3281141519546509,
-0.3968145251274109, 0.0849301815032959,	-0.29096317291259766,	0.4158337414264679,
0.14221571385860443,	-0.29783564805984497,	-0.21871086955070496, -
0.19391785562038422,	-0.40934377908706665,	0.4281107783317566, -
0.5089689493179321,	-0.41472944617271423,	-0.10546628385782242, -

0.6701425909996033, -0.489808589220047, -0.376186341047287, -0.13543687760829926,  
 -0.12336649000644684, -0.7220563292503357, -0.06217554956674576, -  
 0.099810890853405, 0.05519094318151474, 0.07306374609470367, 0.2498975247144699,  
 0.158798947930336, -0.20090073347091675, 0.11332282423973083, -  
 0.25103849172592163, -0.15369673073291779, -0.04350544512271881, -  
 0.0851178988814354, -0.1908155083656311, -0.4129214584827423, 0.6238711476325989,  
 0.19057179987430573, 0.07978576421737671, 0.30655068159103394,  
 0.12087434530258179, 0.13618101179599762, -0.16667363047599792,  
 0.5703001618385315, -0.3068569302558899, -0.17334720492362976,  
 0.07368569821119308, -0.22815512120723724, 0.1349477916955948, -  
 0.44909897446632385, 0.12776629626750946, 0.8668186664581299, -  
 0.014579405076801777, 0.2563166916370392], [-0.10570450872182846,  
 0.13790780305862427, -0.29611435532569885, 0.021248599514365196, -  
 0.06407000869512558, -0.16861875355243683, -0.13514314591884613, -  
 0.0024490721989423037, 0.4737650752067566, -0.17626942694187164,  
 0.244399756193161, -0.3797178566455841, 0.04832637310028076, -0.13981150090694427,  
 -0.3404477834701538, -0.1267590969800949, -0.23266123235225677, -  
 0.29759615659713745, 0.17886188626289368, 0.049607839435338974, -  
 1.3074299097061157, -0.03243668004870415, 0.42144274711608887, -  
 0.3336310088634491, -0.047373633831739426, 0.29804500937461853,  
 0.23910066485404968, -0.1842910796403885, 0.15668928623199463, -  
 0.04880928248167038, -0.07758763432502747, 0.2588750422000885, -  
 0.07355353981256485, 0.17692163586616516, -0.5791793465614319, -  
 0.0049196635372936726, 0.2790604829788208, -0.28706789016723633,  
 0.19080011546611786, 0.3263266384601593, -0.10050593316555023, -  
 0.3464750349521637, 0.007166293449699879, -0.14746132493019104, -  
 0.1901039183139801, 0.000573589641135186, 0.5202240943908691, 0.1511664241552353,  
 -0.09164479374885559, 0.17594610154628754, 0.1632513403892517, 0.251569539308548,  
 0.11609262228012085, -0.05378801003098488, 0.22870691120624542,  
 0.18607719242572784, 0.25763043761253357, 0.6031370162963867, -  
 0.25823259353637695, -0.16780978441238403, 0.4238518178462982, -  
 0.13951000571250916, -0.06374251842498779, 0.35567763447761536, -  
 0.07424404472112656, -0.2709571421146393, 0.22417089343070984, 1.0961953401565552,  
 0.19659295678138733, 0.0789179801940918, 0.2009924054145813, -0.1065668836236,  
 0.27643150091171265, 0.27661073207855225, 0.49342378973960876, -  
 0.016943741589784622, 0.06394834816455841, -0.34030741453170776, -  
 0.05276091396808624, -0.1382909119129181, -0.18781213462352753, -  
 0.6763319969177246, -0.4583955407142639, -0.01578681543469429, -0.552677571773529,  
 0.2626272439956665, 0.4310174882411957, -0.5373598337173462, 0.29390695691108704,  
 0.09334024041891098, 0.17963483929634094, 0.0277444776147604, -7.342139720916748,  
 0.3260544538497925, -0.21535226702690125, 0.06709760427474976,  
 0.17347891628742218, -0.4252185821533203, -0.9265773296356201, 1.266298770904541,  
 -0.029132001101970673, 0.06121351197361946, -0.3054240345954895, -  
 0.007548388559371233, 0.7245302200317383, 0.25022563338279724,

1.2299753427505493, 0.268433541059494, -0.1805061399936676, -0.47296902537345886,  
 -0.15610797703266144, -1.0560821294784546, 0.062212053686380386,  
 0.19992972910404205, 0.1881740689277649, 0.059712234884500504,  
 0.08616860955953598, 0.1167854368686676, 0.17643344402313232, -  
 0.10635441541671753, 0.1623641401529312, -0.21022002398967743, -  
 0.18188749253749847, 0.3962897062301636, -0.16403724253177643, -  
 0.09291694313287735, -0.06028313189744949, 0.5183937549591064, -  
 0.12987098097801208, 0.02877713181078434, -0.05297210067510605,  
 0.07834047824144363, -0.1967429369688034, 0.9479541182518005, -0.4104432165622711,  
 -0.38089337944984436, 0.1484847515821457, -0.4870155453681946, -  
 0.4106968939304352, -0.1297607123851776, 0.1580198109149933, -0.3749220669269562,  
 0.3536655306816101, 0.41436997056007385, -0.1529119461774826, 0.28546658158302307,  
 0.3222712576389313, -0.5626512169837952, 0.08844976127147675, 0.2472795844078064,  
 -0.4839729964733124, 0.18799206614494324, -0.3098377585411072, -  
 0.22550737857818604, 0.01998043991625309, -0.059366609901189804, -  
 0.3353964686393738, -0.47572529315948486, 0.11814704537391663, -  
 0.40058109164237976, -0.027256997302174568, -0.26315540075302124,  
 0.11216795444488525, 0.22248132526874542, 0.0003253216855227947, -  
 0.38469865918159485, -0.5201458930969238, 0.5281473994255066, 0.14545132219791412,  
 0.3045416474342346, 0.22328978776931763, 0.6872178912162781, 0.3052082657814026, -  
 0.2213023155927658, -0.21167390048503876, 0.18805626034736633,  
 0.02205631323158741, 0.05469043180346489, 0.06969360262155533,  
 0.2577589750289917, -0.37217801809310913, 0.23236317932605743, -  
 0.12269124388694763, -0.36240631341934204, -0.05869884416460991,  
 0.13108763098716736, 0.16810651123523712, -0.3794984221458435, 0.3286236524581909,  
 -0.17761151492595673, 0.34696492552757263, -0.2887486517429352, -  
 0.5429438948631287, -0.13270874321460724, -0.48733648657798767, -  
 0.03303558751940727, -0.26274722814559937, -0.34988027811050415, -  
 1.112224817276001, -0.42773303389549255, 0.04396520182490349, 0.3607558310031891,  
 0.13401474058628082, 0.1844346523284912, -0.10153450816869736, -  
 0.012240100651979446, 0.32831278443336487, 0.06739918142557144, -  
 0.30256617069244385, 0.5625017881393433, -0.41209715604782104, 0.5178589820861816,  
 -0.03543471917510033, 0.3327140808105469, -0.42546090483665466, -  
 0.35230574011802673, -0.002902393462136388, -0.7160437703132629,  
 0.825977623462677, -0.3952390253543854, 0.0036183912307024, 0.15516085922718048, -  
 0.19975008070468903, 0.1613692045211792, -0.0499282069504261, 0.36855942010879517,  
 0.1477912962436676, -0.06943070143461227, 0.02415192313492298, -  
 0.1942625641822815, -0.1536337286233902, 0.05179361253976822, -0.3329981565475464,  
 0.7616156935691833, 0.2763577401638031, -0.5335872769355774, -0.3500407636165619,  
 0.13251523673534393, 0.14885550737380981, -0.18809874355793, 0.35506466031074524,  
 -0.18194328248500824, -0.12301323562860489, -0.02997477352619171, -  
 0.3248726427555084, 0.6294851899147034, 0.12427526712417603, -0.04640426114201546,  
 -0.10913432389497757, 0.7081345319747925, 0.01480374950915575,  
 0.27664026618003845, -0.16041254997253418, -0.07424439489841461,

0.1920625865459442,	0.048416342586278915,	-0.040808580815792084,	-
0.2754313349723816,	-0.6399056911468506,	-0.04421538487076759,	-
0.08591507375240326,	0.03326268121600151,	-0.0035336273722350597,	
0.4533844590187073,	0.3268165588378906,	-0.10774247348308563,	-0.5372117161750793,
-0.4281523823738098,	-0.28198209404945374,	0.061128731817007065,	-
0.2462514042854309,	-0.20744942128658295,	0.3199880123138428,	-
0.13205187022686005,	-0.22901295125484467,	-0.18603835999965668,	-
0.017491133883595467,	0.1748887002468109,	-0.4399857521057129,	
0.10667165368795395,	0.34009894728660583,	0.2552591562271118,	-0.3855103850364685,
-0.06277534365653992,	-0.22571006417274475,	0.07508782297372818,	
0.01771991327404976,	-0.0131677072495221,	0.48330339789390564,	-
0.1981583535671234,	-0.14375290274620056,	0.17780515551567078,	0.2762075364589691,
0.3070702850818634,	0.18929581344127655,	0.19519764184951782,	0.2662467658519745,
-0.5747968554496765,	0.007932530716061592,	-0.15115255117416382,	
0.23182961344718933,	-0.1416928917169571,	-0.11889435350894928,	
0.43252870440483093,	-0.09807675331830978,	0.3085139989852905,	-
0.06704581528902054,	0.37698203325271606,	0.08892617374658585,	-
0.18220694363117218,	-0.5874041318893433,	0.27005627751350403,	0.9468769431114197,
-0.06251062452793121,	-0.18786609172821045,	0.2566240429878235,	
0.1521744579076767,	0.4882001578807831,	0.18437741696834564,	0.2582813799381256,
-0.19513832032680511,	1.6153093576431274,	-0.5838130712509155,	-
0.03616591542959213,	-0.24056068062782288,	-0.13784192502498627,	-
0.09420772641897202,	0.5713971257209778,	0.1994933933019638,	
0.026505960151553154,	0.014178316108882427,	-0.15593236684799194,	-
0.1733580231666565,	-0.14923769235610962,	0.03407426178455353,	0.5384862422943115,
0.07338787615299225,	0.11542381346225739,	-0.01529585663229227,	-
0.022717755287885666,	-0.019321491941809654,	-0.0572047159075737,	
0.21120762825012207,	0.07740379869937897,	0.20938248932361603,	-
0.49050143361091614,	-0.12305588275194168,	0.1613398790359497,	-
0.0836775153875351,	0.03477384150028229,	0.11502614617347717,	-
0.22666050493717194,	0.5562049746513367,	-0.40450242161750793,	-
0.2048724740743637,	-0.10599681735038757,	-0.27015411853790283,	
0.26361170411109924,	0.20263460278511047,	0.37978434562683105,	
0.15334941446781158,	0.12605006992816925,	0.0005043801502324641,	-
0.1319456547498703,	-0.47007712721824646,	0.197788804769516,	0.2908976376056671,
0.5481163263320923,	0.5849900245666504,	-0.10590031743049622,	0.4754485487937927,
0.5050010681152344,	-0.2174878567457199,	-0.5940462350845337,	0.3248356282711029,
-0.15169702470302582,	1.8060152530670166,	0.0716799795627594,	-
0.42354896664619446,	-0.07422388345003128,	0.06789003312587738,	-
0.5189822316169739,	-0.167555034160614,	0.42614439129829407,	-0.1272992342710495,
0.016014890745282173,	-0.11366040259599686,	-0.4308631718158722,	
0.12056192755699158,	-0.8434276580810547,	-0.32650026679039,	-0.4506431221961975,
0.11178723722696304,	-0.2611115276813507,	0.19197191298007965,	
0.12052202224731445,	0.4527934193611145,	-0.3254498839378357,	-0.3591388761997223,

-0.2732936441898346, 0.037604957818984985, 0.210501030087471, 0.43315139412879944,  
 0.2878892123699188, 0.09515181183815002, -0.03664582222700119, -  
 0.2028643935918808, -0.45809656381607056, 0.624149739742279, 0.03483983129262924,  
 0.5037246346473694, 0.7976599335670471, -0.18451428413391113, 0.2625584006309509,  
 0.000559329753741622, -0.6554741263389587, -0.33765536546707153, -  
 0.06358806788921356, -0.040205150842666626, -0.36787861585617065, -  
 0.5800901651382446, 0.32927316427230835, 0.20085926353931427, 0.4227878451347351,  
 0.12953037023544312, 0.0024373570922762156, 0.11158521473407745, -  
 0.3712844252586365, 0.6618252396583557, 0.280085027217865, -0.32797205448150635,  
 0.7938905954360962, -0.5152537226676941, 0.2686878442764282, -  
 0.048718102276325226, -0.580170214176178, -0.03988627716898918,  
 0.42183613777160645, 0.2916414737701416, 0.012953386642038822, -  
 0.11650524288415909, -0.2676823139190674, 0.06793490797281265, -  
 0.11774532496929169, 0.3476506173610687, -0.034615494310855865,  
 0.0527532696723938, -0.29523488879203796, -0.19536292552947998, -  
 0.4187326729297638, -0.5584217309951782, -0.2522520124912262, -0.5310566425323486,  
 0.5352290272712708, -0.1215951144695282, -0.3084118664264679, 0.37260621786117554,  
 0.44893723726272583, 0.13557182252407074, 0.4511372745037079, -0.3281141519546509,  
 -0.3968145251274109, 0.0849301815032959, -0.29096317291259766, 0.4158337414264679,  
 0.14221571385860443, -0.29783564805984497, -0.21871086955070496, -  
 0.19391785562038422, -0.40934377908706665, 0.4281107783317566, -  
 0.5089689493179321, -0.41472944617271423, -0.10546628385782242, -  
 0.6701425909996033, -0.489808589220047, -0.376186341047287, -0.13543687760829926,  
 -0.12336649000644684, -0.7220563292503357, -0.06217554956674576, -  
 0.099810890853405, 0.05519094318151474, 0.07306374609470367, 0.2498975247144699,  
 0.158798947930336, -0.20090073347091675, 0.11332282423973083, -  
 0.25103849172592163, -0.15369673073291779, -0.04350544512271881, -  
 0.0851178988814354, -0.1908155083656311, -0.4129214584827423, 0.6238711476325989,  
 0.19057179987430573, 0.07978576421737671, 0.30655068159103394,  
 0.12087434530258179, 0.13618101179599762, -0.16667363047599792,  
 0.5703001618385315, -0.3068569302558899, -0.17334720492362976,  
 0.07368569821119308, -0.22815512120723724, 0.1349477916955948, -  
 0.44909897446632385, 0.12776629626750946, 0.8668186664581299, -  
 0.014579405076801777, 0.2563166916370392]]

#### 4. Storages

##### 基本概念

Storage 模块在 CAMEL 框架中负责数据的存储与管理，是一个功能全面的框架，提供了统一的接口和数据结构，支持键值存储和向量存储等多种类型的数据存储机制。通过抽象基类与具体实现的结合，Storage 模块能够高效地处理数据的读取、写入和检索操作，为 RAG 应用的实现提供了坚实的基础。

##### 键值存储

###### BaseKeyValueStorage

目的：

作为创建各种键值存储系统的基础抽象类。

**功能:**

- 标准化数据记录的保存、加载和清除操作。
- 主要通过 Python 字典进行接口交互。

**应用场景:**

- JSON 文件存储
  - NoSQL 数据库（如 MongoDB 和 Redis）
  - Python 内存中的字典存储
- 

### InMemoryKeyValueStorage

**描述:**

基于 BaseKeyValueStorage 的具体实现，使用内存中的列表存储数据。

**特点:**

- 适用于临时存储，数据为易失性，程序终止后即丢失。

**功能:**

- 实现了在内存中保存、加载和清除记录的方法。
- 非常适合开发和测试场景，无需持久化存储需求。

### 向量存储

向量存储（Vector Store）用于存储高维度的向量数据，如文本或图像的嵌入表示。它支持高效的相似度计算和最近邻搜索，是 RAG 应用中检索相关信息的核心组件。向量存储的特点包括：

- **高性能**: 支持大规模数据的快速检索
- **可扩展性**: 适应不同规模的数据量
- **灵活性**: 支持多种相似度度量方式

### BaseVectorStorage

**目的:**

设计为扩展特定向量存储实现的抽象基类。

**特点:**

- 支持多种操作，如添加、删除向量，查询相似向量，以及维护向量数据库的状态。
- 提供灵活性，可指定向量维度、集合名称、距离度量等参数。

**功能:**

- 为构建多样化的向量存储解决方案提供基础架构。

### MilvusStorage

**描述:**

基于 BaseVectorStorage 的具体实现，专为与 Milvus 交互而设计的存储方案。

**特点:**

- 针对 Milvus（一个云原生向量搜索引擎）进行优化，支持高效的大规模向量检索和管理操作。

Reference: [Milvus](#)

### QdrantStorage

**描述:**

基于 BaseVectorStorage 的具体实现，专为与 Qdrant 交互而设计。

**特点:**

- 针对 Qdrant 向量搜索引擎进行优化。
- 提供高效的向量存储、管理和查询功能，支持大规模近似最近邻（ANN）搜索。

**功能:**

- 实现向量的添加、删除、相似度查询等核心操作。
- 支持自定义向量维度、集合名称、距离度量（如欧几里得距离、余弦相似度等）。
- 与 Qdrant 的 API 无缝集成，适用于高性能向量搜索场景。

**适用场景:**

- 推荐系统、自然语言处理（NLP）嵌入查询、多媒体检索（图像、音频、视频）等需要高效向量搜索的应用场景。

Reference: [Qdrant](#)

---

## 图存储

### BaseGraphStorage

**目的:**

设计为扩展特定图存储实现的抽象基类。

**特点:**

- 支持多种操作，包括：
  - get\_client: 获取图存储的客户端连接。
  - get\_schema: 获取当前的图存储模式信息。
  - get\_structured\_schema: 获取结构化的模式表示。
  - refresh\_schema: 刷新图存储的模式。
  - add\_triplet: 添加三元组（节点及边的表示）。
  - delete\_triplet: 删除三元组。
  - query: 执行图查询操作。

**功能:**

- 为各种图存储解决方案提供基础架构，便于构建定制化图存储实现。

### NebulaGraph

**描述:**

基于 BaseGraphStorage 的具体实现，专为与 NebulaGraph 交互而设计。

**特点:**

- 面向 NebulaGraph 的优化实现，支持其分布式、高扩展性及高速图数据操作的特性。
- 提供了对 NebulaGraph API 的无缝集成，用于高效处理图形数据的存储与查询。

**功能:**

- 实现了 BaseGraphStorage 的所有核心方法，支持快速执行分布式图数据的增删查改操作。
- 支持复杂图查询语句，用于大规模图数据的结构化分析和推理。

**适用场景:**

- 推荐系统、知识图谱、社交网络分析、路径优化及大规模图数据挖掘等需要高性能图存储和查询的场景。

Reference: [NebulaGraph](#)

### Neo4jGraph

### 描述:

基于 BaseGraphStorage 的具体实现，专为与 Neo4j 交互而设计，Neo4j 是业内最受信赖的图数据库之一。

### 特点:

- 面向 Neo4j 的优化实现，充分利用其强大的关系建模能力和高效的图查询功能。
- 支持复杂的图形操作和查询语句，提供强大的可视化和分析功能。

### 功能:

- 实现 BaseGraphStorage 的所有核心方法，包括：
  - get\_client: 获取连接到 Neo4j 的客户端实例。
  - get\_schema 和 refresh\_schema: 管理 Neo4j 图数据库的模式。
  - add\_triplet 和 delete\_triplet: 实现节点和关系的添加与删除。
  - query: 执行基于 Cypher 的图查询操作。

### 适用场景:

- 知识图谱、社交网络分析、推荐系统、诈骗检测、供应链管理及其他需要强关系建模的应用场景。

### 优势:

- 凭借 Neo4j 的成熟生态系统和广泛支持，Neo4jGraph 提供了一种稳定、高性能的解决方案，用于高效存储和处理复杂的图数据。

Reference: [Neo4jGraph](#)

---

### 动手实践

这里以 **QdrantStorage** 为例：

```
from camel.storages import QdrantStorage, VectorDBQuery, VectorRecord

Create an instance of QdrantStorage with dimension = 4
qdrant_storage = QdrantStorage(vector_dim=4, collection_name="my_collection")

Add two vector records
qdrant_storage.add([VectorRecord(
 vector=[-0.1, 0.1, -0.1, 0.1],
 payload={'key1': 'value1'},
),
VectorRecord(
 vector=[-0.1, 0.1, 0.1, 0.1],
 payload={'key2': 'value2'},
)])
Query similar vectors
query_results = qdrant_storage.query(VectorDBQuery(query_vector=[0.1, 0.2, 0.1, 0.1],
top_k=1))
for result in query_results:
 print(result.record.payload, result.similarity)

Clear all vectors
```

```
qdrant_storage.clear()
```

```
>>>
```

```
{'key2': 'value2'} 0.5669467095138407
```

这里计算出的结果是向量之间的余弦距离（默认），CAMEL 提供了 3 种计算距离的方式，可在 QdrantStorage 初始化的时候指定 *distance* 参数来选择：

```
distance_map = {
 VectorDistance.DOT: Distance.DOT, # 点积距离
 VectorDistance.COSINE: Distance.COSINE, # 余弦距离
 VectorDistance.EUCLIDEAN: Distance.EUCLID # 欧氏距离
}
```

---

## 5. Retrievers

### 基本概念

Retrievers 模块可以理解为一个搜索引擎，专门用于在大量文本中高效查找特定信息。它的功能就像一位熟练的图书管理员，帮助你快速找到需要的主题或关键词，无论是基于语义还是关键字。

Retrievers 模块支持两种主要的检索方式：向量检索和关键词检索。

向量检索器（Vector Retriever）基于向量表示技术，将文本、图像等数据转化为高维向量，通过嵌入模型生成数学表示并存储在向量存储系统中。当用户输入查询时，嵌入模型会将其转换为向量，在存储系统中寻找最接近的匹配向量。这种方式擅长处理语义搜索，能够理解自然语言的模糊关系，常应用于推荐系统、语义查询和跨模态搜索等场景。

关键词检索器（Keyword Retriever）则更加直接，通过对文档进行预处理（如分词、建立关键词索引），解析用户的查询关键词并匹配相应的文档内容。它依赖于关键词的精确匹配，适合快速查找特定术语或短语。

向量检索器偏向语义层面的理解，适合模糊查询和深度语义挖掘；而关键词检索器则注重精确性，适合快速直接的检索需求。两者结合使用，可以在不同场景中提供高效的解决方案，是知识管理、问答系统和检索增强生成（RAG）任务的重要工具。

---

### 动手实践

#### 1. 向量检索

初始化 VectorRetriever

我们首先需要初始化 VectorRetriever。可以选择传入一个嵌入模型，如果不提供嵌入模型，默认会使用 OpenAIEmbedding。

```
from camel.embeddings import SentenceTransformerEncoder
from camel.retrievers import VectorRetriever
```

```
embedding_model=SentenceTransformerEncoder(model_name='intfloat/e5-large-v2')
```

嵌入并存储数据

在执行检索之前，需要准备数据并将其存储在向量存储中。process 方法会处理输入内容（可以是文件或 URL），将内容划分为小块，并将这些小块的嵌入存储在指定的向量存储中。

```
指定内容来源路径，可以是文件路径或 URL
content_input_path = "https://www.camel-ai.org/"
```

```

创建或初始化向量存储 (例如 QdrantStorage)
from camel.storages.vectordb_storages import QdrantStorage

vector_storage = QdrantStorage(
 vector_dim=embedding_model.get_output_dim(), # 嵌入向量的维度
 collection_name="my first collection", # 向量存储的集合名称
 path="storage_customized_run", # 向量存储的位置
)
初始化 VectorRetriever
vr = VectorRetriever(embedding_model=embedding_model, storage=vector_storage)
将内容嵌入并存储到向量存储中
vr.process(content_input_path, chunk_type="chunk_by_title")
执行查询
将数据存储后，可以通过查询字符串来检索相关信息。query 方法会根据输入的查询语句，从存储中检索最匹配的信息。
指定查询字符串
query = "What is CAMEL"

执行查询并检索结果
results = vr.query(query)

打印检索结果
print(results)
示例输出：
[{'similarity score': '0.8388913333364109', 'content path': 'https://www.camel-ai.org/', 'metadata': {'emphasized_text_contents': ['1 Customize agents'], 'emphasized_text_tags': ['b'], 'filetype': 'text/html', 'languages': ['eng'], 'link_texts': ['Get Started', 'Join Community'], 'link_urls': ['https://github.com/camel-ai', 'http://discord.camel-ai.org/'], 'url': 'https://www.camel-ai.org/'}, 'extra_info': {}, 'text': 'Build Multi-Agent Systems for Data Generation\n\nCAMEL-AI.org is the 1st LLM multi-agent framework and an open-source community dedicated to finding the scaling law of agents.\n\nGet Started\nJoin Community\n\n1 Customize agents\n\nCustomizable agents are the fundamental entities of the CAMEL architecture. CAMEL empowers you to customize agents using our modular components for specific tasks.'}]

```

## 2. 自动化检索

**AutoRetriever** 方法进一步简化了检索流程，它自动处理嵌入、存储数据以及执行查询的任务，非常适合需要处理多个内容输入路径的场景。

```

from camel.retrievers import AutoRetriever
from camel.types import StorageType
from camel.embeddings import SentenceTransformerEncoder

embedding_model=SentenceTransformerEncoder(model_name='intfloat/e5-large-v2')

初始化 AutoRetriever

```

```

ar = AutoRetriever(
 vector_storage_local_path="retrievers", # 向量存储本地路径
 storage_type=StorageType.QDRANT, # 使用 Qdrant 作为存储类型
 embedding_model=embedding_model
)

使用 Auto Retriever 执行嵌入、存储和查询
retrieved_info = ar.run_vector_retriever(
 contents=[
 "https://www.camel-ai.org/", # 示例 URL
],
 query="What is CAMEL-AI", # 查询字符串
 return_detailed_info=True # 是否返回详细信息，包括元数据
)

打印检索结果
print(retrieved_info)

```

**输出示例：**

```

{'Original Query': 'What is CAMEL-AI', 'Retrieved Context': [{}{'similarity score': '0.8677001653976963', 'content path': 'https://www.camel-ai.org/', 'metadata': {}{'emphasized_text_contents': ['1 Customize agents'], 'emphasized_text_tags': ['b'], 'filetype': 'text/html', 'languages': ['eng'], 'link_texts': ['Get Started', 'Join Community'], 'link_urls': ['https://github.com/camel-ai', 'http://discord.camel-ai.org/'], 'url': 'https://www.camel-ai.org/'}, 'extra_info': {}, 'text': 'Build Multi-Agent Systems for Data Generation\n\nCAMEL-AI.org is the 1st LLM multi-agent framework and an open-source community dedicated to finding the scaling law of agents.\n\nGet StartedJoin Community\n\n1 Customize agents\n\nCustomizable agents are the fundamental entities of the CAMEL architecture. CAMEL empowers you to customize agents using our modular components for specific tasks.'}]}

```

**内容输入路径：**contents 可以是文件路径或 URL，支持多个输入。**查询字符串：**query 定义了检索目标，AutoRetriever 将根据该字符串搜索相关内容。**返回详细信息：**设置 return\_detailed\_info=True 可返回包括元数据在内的详细检索信息。

## 2. 向量数据库介绍

向量数据库 (Vector Database) 是一种专门用于存储和检索高维向量数据的数据库系统。在现代人工智能和机器学习应用中，数据（如文本、图像、音频等）常被转换为高维向量，以捕捉其语义或特征。向量数据库通过高效的相似度搜索算法，能够快速检索与查询向量最相似的向量集合。

### 主要功能：

- 向量存储：**支持存储大量高维向量数据，通常还关联其他元数据。
- 相似度搜索：**实现高效的近似最近邻 (Approximate Nearest Neighbor, ANN) 搜索，快速找到与查询向量最相似的向量。
- 扩展性：**设计用于处理大规模数据，支持水平扩展以满足增长的存储和计算需求。

### 常见应用：

- **推荐系统**: 根据用户行为或偏好, 检索相似产品或内容进行推荐。
- **图像和视频搜索**: 通过内容检索相似的图像或视频。
- **自然语言处理**: 在嵌入空间中查找语义相似的文本或词语。

#### 实现技术:

向量数据库通常采用多种算法和数据结构来实现高效的相似度搜索, 包括:

- **HNSW (Hierarchical Navigable Small World) 图**: 一种基于图的近似最近邻搜索算法, 具有高查询效率和准确性。
- **局部敏感哈希 (Locality-Sensitive Hashing, LSH)**: 通过哈希函数将相似的向量映射到相同的桶中, 实现快速检索。
- **产品量化 (Product Quantization, PQ)**: 将向量分解为子空间并量化, 以减少存储和计算成本。

#### 常见的向量数据库系统:

- **Milvus**: 开源的向量数据库, 支持亿级向量的高效存储和检索。
- **Pinecone**: 提供向量数据库即服务, 简化向量数据的管理和查询。
- **Weaviate**: 开源的向量搜索引擎, 支持多种数据模式和扩展。

在 CAMEL 框架中, camel.storages.vectordb\_storages 包提供了与向量数据库交互的功能模块, 包含以下子模块:

- **base 模块**: 定义了向量存储的基础类和接口, 包括 BaseVectorStorage、VectorDBQuery、VectorDBQueryResult 等。
- **milvus 模块**: 提供了与 Milvus 数据库交互的具体实现, 如 MilvusStorage 类。
- **qdrant 模块**: 提供了与 Qdrant 数据库交互的实现, 如 QdrantStorage 类。
- **weaviate 模块**: 提供了与 Weaviate 数据库交互的实现, 如 WeaviateStorage 类。

这些模块的设计使得 CAMEL 框架能够灵活地与不同的向量数据库集成, 满足多样化的应用需求。

### 3. 搭建知识库

#### 1. Embedding 模型选择

在搭建知识库时, 常用的 embedding model 可选择:

- **Embedding API**: 如 OpenAI, Cohere, Mistral 等的 API 服务, 适用于项目或工程中需要方便直接的使用嵌入且不介意数据隐私的情况
- **Open Embedding Model**: 适用于对数据隐私有要求或需要自定义模型本地部署的情况

以上两者的区别可以等价的理解成大语言模型 API 和开源本地 LLM 的区别。

The screenshot shows the MTEB Leaderboard interface on a computer screen. At the top, there's a navigation bar with 'Spaces', 'mteb/leaderboard', and other options like 'App', 'Files', and 'Community'. Below the navigation is a search bar and filter sections for 'Model types' (Open, Proprietary, Sentence Transformers) and 'Model sizes' (100M, 100M to 250M, 250M to 500M, 500M to 1B, >1B). The main area has tabs for 'Overall', 'Bitext Mining', 'Classification', 'Clustering', 'Pair Classification', 'Ranking', 'Retrieval', 'STS', and 'Summarization'. Under 'Overall', there are tabs for 'English', 'Chinese', 'French', and 'Polish'. A section titled 'Overall MTEB English leaderboard' contains a table with columns: Model, Model Size (Million Parameters), Memory Usage (GB, fp32), Embedding Dimensions, Max Tokens, Average (56 datasets), Classification Average (12 datasets), Clustering Average (11 datasets), Pair Classification Average (3 datasets), Ranking Average (4 datasets), and Ret. Ave. (15 datasets). The table lists various models like 'voyage-large-2-instruct', 'SFT-embedding-Mistral', 'gte-OpenAI-7B-instruct', etc., with their respective scores.

在 [HuggingFace 的排行榜上](#)可以看到各种各样开源好用的 embedding 模型。在具体研究和工程问题的选择中，其实没有一个明确的好坏评价标准，适合自己的数据处理场景才是最好的。一个常见的方法是构造自己应用场景的专有评测集，然后分别尝试，量化的方式来评估分数最后选择。

todo：评测集构建的例子

## 2. 数据预处理

当我们拿到一篇或多篇文章，它们的格式可能是 pdf, txt, doc 甚至是 ppt，我们往往会对对其进行数据的预处理，来帮助后续流程中更好地让大语言模型和 embedding 模型来发挥作用。

机器学习中有一句深刻的名言：Garbage in, Garbage out。这不仅仅可以用于机器学习模型的训练，推理时过于混乱的原始信息也可能会极大程度地干扰模型的回复效果。因此，**数据预处理至关重要！**

数据处理的常用手段包含格式统一转换和数据清洗

数据读取 → 数据格式转换 → 数据清洗

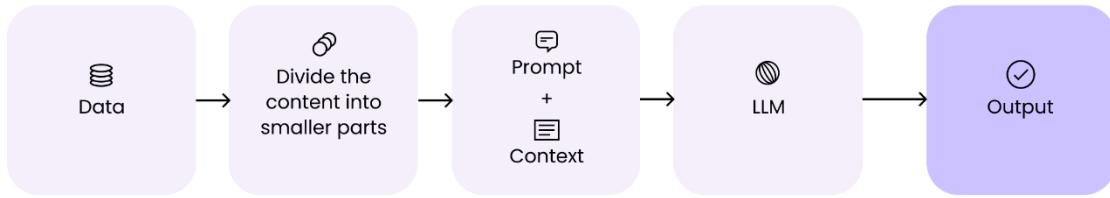
Todo：流程图

- **数据读取：**从文件、数据库或 API 获取原始数据
- **数据格式转换：**将多样的数据统一转换成方便模型处理的格式（常用的为 Markdown format）
- **数据清洗：**去除噪声、纠正错误、信息筛选等

TODO：一个 pdf 转成 Markdown 的代码示例

## 4. 构建 RAG 应用

## The RAG process



### 1. Basic RAG

RAG, 全称为 Retrieval Argumented Generation, 检索增强生成。

大语言模型通过预训练阶段在海量文本上的自监督训练中获得了大量的人类知识, 但是依然存在一些诸如时效性和幻觉的问题。而 RAG 正是为了补充并增强大语言模型的能力, 使其尽量和现实世界对齐的一种技术路线。

一个最为简单朴素的 RAG 的思想为: 我有一些文本数据包含了有用的信息, 我想让大语言模型利用这些文本信息来生成回复, 而不是直接输出答案, 因此在原先的交互流程中, 我们要将外部文本信息组成上下文合并到 prompt 中, 交付给 LLM, 以此来达成增强生成 (Argumented Generation) 的目的, 就是这么简单!

这个时候聪明的读者可能就会发现了, 欸, 那 **Retrieval** 去哪儿了呢。这是一个很好的问题, 如果我现在有一篇 800 字的作文或者 5000 字的论文, 那我直接将其简单处理后合并进 prompt 中就可以了, 如下所示:

```
essay = "800字的作文, 以下省略800字..."
```

```
paper = "5000字的论文, 以下省略5000字..."
```

```
system_prompt = "you are a helpful assistant"
```

```
user_prompt = f"帮我总结一下这篇文章, 全文信息如下: {essay}"
```

```
prompt = system_prompt + user_prompt
```

但是, 当我有 100 篇, 甚至成千上万篇文章, 以上亿 token 为单位的上下文长度时, 很显然, 会分分钟超出现在大语言模型所支持的上下文长度。此时, 一个显而易见的解决方案便是: 我只将我最关心的那一部分内容合并到 prompt 中交付给大模型。那如何才能知道大量文本中, 哪些才是我最关心或者说和我的问题最相关的片段部分呢。欸, 这就是 Retrieval 阶段的功能和作用。

CAMEL 中对于 Retrieval 这一过程的功能实现在 camel.retrievers 中, 具体参考 [https://docs.camel-ai.org/key\\_modules/retrievers.html](https://docs.camel-ai.org/key_modules/retrievers.html)

根据匹配标准和算法的不同, 常用的 retriever 大致可以分成两类, 基于向量的稠密检索和基于关键词的稀疏检索, 分别对应 camel.retrievers 中的 VectorRetriever 和 BM25Retriever

```
from camel.retrievers import VectorRetriever
```

```
from camel.retrievers import BM25Retriever
```

```
from camel.embeddings import SentenceTransformerEncoder
```

```
embedding_model=SentenceTransformerEncoder(model_name='intfloat/e5-large-v2')
```

```

国内无法运行上述代码，可以注释掉使用以下方案
embedding_model=SentenceTransformerEncoder(model_name='./embedding_model/')
下载方案 2：打开这个链接 https://hf-mirror.com/intfloat/e5-large-v2/tree/main，下载除了 model.safetensors 的以外部分，保存到当前代码同级目录的 embedding_model 文件夹下。
也可以从百度云盘直接下载 embedding_model 文件夹放到当前代码同级目录下。
https://pan.baidu.com/s/1xt0Tg_Wmr8iJuyGiPfgJrw 提取码: 7pzs

vr = VectorRetriever(embedding_model=embedding_model)
br = BM25Retriever()

```

VectorRetriever，向量检索器利用数据的向量表示，将数据（如文本、图像或其他形式的信息）转换为高维空间中的数值向量。

以下是向量检索的工作流描述：

- **分块**：对于大型文档或其他形式的数据信息而言，需要按一定的划分规则将其分解为较小的块
- **嵌入**：对于每一个分块，使用嵌入模型将文本转换为高维的向量形式。
- **存储**：这些得到的向量会被存储在数据库中。
- **检索**：当提出问题或进行查询时，嵌入模型会将问题转换为向量，然后在此向量数据库中搜索相似度高的匹配向量（这里一般会用余弦相似度来计算两个向量之间的距离），从而返回最相关的信息片段。

BM25Retriever，关键词检索器从结果而言是一样的，只不过是从更加稀疏的关键词的维度来对文本进行召回。通过关键词匹配的算法获取相似度高的文本片段，尽可能筛选出最相关的信息片段。

最后将最相关的信息片段像之前演示的那样合并到 prompt 中，交给大语言模型生成检索增强后的回复

接下来我们通过一个 demo 来学习以下如何快速搭建一个 Basic RAG 的系统

#### 读取示例数据：

这里我们首先下载一篇 CAMEL 的论文作为文本示例，该 PDF 就是我们的原始文件

```

import os
import requests

```

```
os.makedirs('local_data', exist_ok=True)
```

```

url = "https://arxiv.org/pdf/2303.17760.pdf"
response = requests.get(url)
with open('local_data/camel_paper.pdf', 'wb') as file:
 file.write(response.content)

```

#### 实例化 Embedding Model：

这里使用 VectorRetriever，使用本地模型作为嵌入模型。

```

from camel.embeddings import SentenceTransformerEncoder
from camel.retrievers import VectorRetriever

```

```
embedding_model=SentenceTransformerEncoder(model_name='intfloat/e5-large-v2')
```

### 向量嵌入和存储数据：

接下来，我们将原始的文档读取，分块并且通过 embedding 模型生成向量表示和索引并将其存储在专门用来存储向量数据的向量数据库，VectorDB 中。CAMEL 框架中的 process 函数其本质上就是将文件读取、切块、嵌入并储存这个固定的处理流程统一封装成了一个方法，使得开发者更加方便快速的搭建 RAG 应用。

```
创建并初始化一个向量数据库 (以 QdrantStorage 为例)
```

```
from camel.storages.vectordb_storages import QdrantStorage
```

```
vector_storage = QdrantStorage(
 vector_dim=embedding_model.get_output_dim(),
 collection="demo_collection",
 path="storage_customized_run",
 collection_name="论文"
)
初始化 VectorRetriever 实例并使用本地模型作为嵌入模型
vr = VectorRetriever(embedding_model= embedding_model,storage=vector_storage)
将文件读取、切块、嵌入并储存在向量数据库中，这大概需要 1-2 分钟
vr.process(
 content="local_data/camel_paper.pdf"
)
```

这里如果出现 UserWarning: Failed to partition the file: local\_data/camel\_paper.pdf  
warnings.warn(f"Failed to partition the file: {input\_path}")需要补充这个环境：

```
pip install "unstructured[pdf]"
```

### 执行检索，查看效果：

完成向量数据库的构建和存储后，就可以根据问题来进行检索。这里调用 query 方法执行查询

```
设定一个查询语句
```

```
query = "CAMEL 是什么"
```

```
执行查询并获取结果
```

```
results = vr.query(query=query, top_k=1)
```

```
print(results)
```

这里的参数 top\_k=1 是指定检索器返回最相关的文本内容，同时还有一个相似度阈值参数 similarity\_threshold，其值介于 0 到 1 之间，默认值是 0.75，从而确保检索到的内容的相关程度，可以根据自己应用场景的实际需求更改 top\_k 或 similarity\_threshold 值。

"What is CAMEL"得到回复为

```
[{'similarity score': '0.8193796123510835', 'content path': 'local_data/camel_paper.pdf',
'metadata': {'filetype': 'application/pdf', 'languages': ['eng'], 'page_number': 1}, 'extra_info': {},
'text': '3 2 0 2\n\n o N 2\n\n] \n\nA . s c [\n\n2 v 0 6 7 7 1 . 3 0 3 2 : v i X r a\n\nCAMEL:
Communicative Agents for “Mind” Exploration of Large Language Model Society
https://www.camel-ai.org\n\nGuohao Li* Hasan Abed Al Kader Hammoud*\n\nHani
Itani*\n\nDmitrii Khizbullin\n\nBernard Ghanem\n\nKing Abdullah University of Science and
```

Technology (KAUST)\n\nAbstract']]

返回的字典包括：

- similarity score 相似性分数
- content path 内容路径
- metadata 元数据
- text 文本

当然，如果在实际应用中我们往往只关心模型的文本回复部分，我们可以在字典中取出 text 的值。

```
print(results[0]["text"])
```

我们再尝试一个与论文内容无关的提问"Compared with dumpling and rice, which should I take for dinner?":

```
retrieved_info_irrelevant = vr.query(
 query="Compared with dumpling and rice, which should I take for dinner?",
 top_k=1,
 similarity_threshold=0.8
)
```

```
print(retrieved_info_irrelevant)
[{'text': 'No suitable information retrieved from local_data/camel paper.pdf with
similarity_threshold = 0.75'}]
```

我们可以看到在相似度阈值为 0.8 的情况下，检索器没有找到论文中与提问有关内容，这和我们的预期是十分符合的。

到目前为止，整个 RAG 的流程还没结束，因为我们实际上目前为止只是把最相关的内容检索了出来，还差最后一步，结合大语言模型的生成。

聪明的读者可能已经想到了，此时我们只需要实例化一个 LLM 或者 Agent 就可以了，CAMEL 中的一个用于一般对话场景的 Agent 为 camel.agents 中的 ChatAgent。

```
from camel.agents import ChatAgent
from camel.models import ModelFactory
from camel.types import ModelPlatformType

from dotenv import load_dotenv
import os

load_dotenv()
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')
retrieved_info = vr.query(
 query="what is roleplaying?",
 top_k=1,
)

assistant_sys_msg = """""
你是一个帮助回答问题的助手，
我会给你原始查询和检索到的上下文，
根据检索到的上下文回答原始查询，
```

如果你无法回答问题就说我不知道。

```
model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)

user_msg = retrieved_info[0]['text']
print(user_msg + '\n')

agent = ChatAgent(assistant_sys_msg,model=model)
使用 step 方法获得最终的检索增强生成的回复并打印
assistant_response = agent.step(user_msg)
print(assistant_response.msg.content)
```

>>>

novel communicative agent framework named role-playing . Our approach involves using inception prompting to guide chat agents toward task completion while maintaining consistency with human intentions. We showcase how role-playing can be used to generate conversational data for studying the behaviors and capabilities of a society of agents, providing a valuable resource for investigating conversational language models. In particular, we conduct comprehensive studies on instruction-following

根据提供的上下文，这个新的沟通代理框架名为“角色扮演” (role-playing)。该方法通过使用启发式提示 (inception prompting) 来引导聊天代理完成任务，同时保持与人类意图的一致性。研究展示了如何利用角色扮演生成对话数据，以研究代理社会的行为和能力，为调查对话语言模型提供宝贵的资源。特别是，研究对指令遵循进行了全面的研究。如果你有更具体的问题或需要进一步的信息，请告诉我！

更进一步我们可以将这一个步骤封装到一个函数里：

```
from camel.agents import ChatAgent
from camel.messages import BaseMessage
from camel.types import RoleType
from camel.retrievers import AutoRetriever
from camel.types import StorageType

def single_agent(query: str) ->str :
 # 设置 agent 角色
 assistant_sys_msg = """你是一个帮助回答问题的助手，
 我会给你原始查询和检索到的上下文，
 根据检索到的上下文回答原始查询，
```

如果你无法回答问题就说我不知道。"""

```
添加自动检索器
auto_retriever = AutoRetriever(
 vector_storage_local_path="local_data2/",
 storage_type=StorageType.QDRANT,
 embedding_model=embedding_model)

运行向量检索器
retrieved_info = auto_retriever.run_vector_retriever(
 query=query,
 contents=[
 "local_data/camel_paper.pdf", # 示例本地路径
 "https://github.com/camel-ai/camel/wiki/Contributing-Guidlines", # 示例
 url
],
 top_k=1,
 return_detailed_info=False,
 similarity_threshold=0.5
)

将检索到的信息传递给 agent
user_msg = str(retrieved_info)
agent = ChatAgent(assistant_sys_msg,model =model)

获取响应
assistant_response = agent.step(user_msg)
return assistant_response.msg.content

print(single_agent("如果我对贡献 CAMEL 项目感兴趣，我应该怎么做？"))
```

>>>

如果您对贡献 CAMEL 项目感兴趣，您可以按照以下步骤进行：

1. \*\*查看项目页面\*\*：访问 CAMEL 项目的 GitHub 页面，了解项目的最新动态、功能和目标。
2. \*\*阅读贡献指南\*\*：在项目页面中查找并阅读“Contributing Guidelines”（贡献指南），了解如何参与项目、代码风格要求、提交 Pull Request 的流程等。
3. \*\*参与讨论\*\*：加入项目的 Issues 和 Discussions 部分，与其他贡献者交流，提出问题或建议。
4. \*\*选择任务\*\*：从 Issues 列表中选择一个适合您的任务开始。如果有疑问，可以在 Issue 下留言询问。
5. \*\*提交代码\*\*：完成任务后，提交 Pull Request，并按照项目的要求进行代码审查。

希望您能顺利参与到 CAMEL 项目中！

```
print(single_agent("什么是 RolePlaying?"))
```

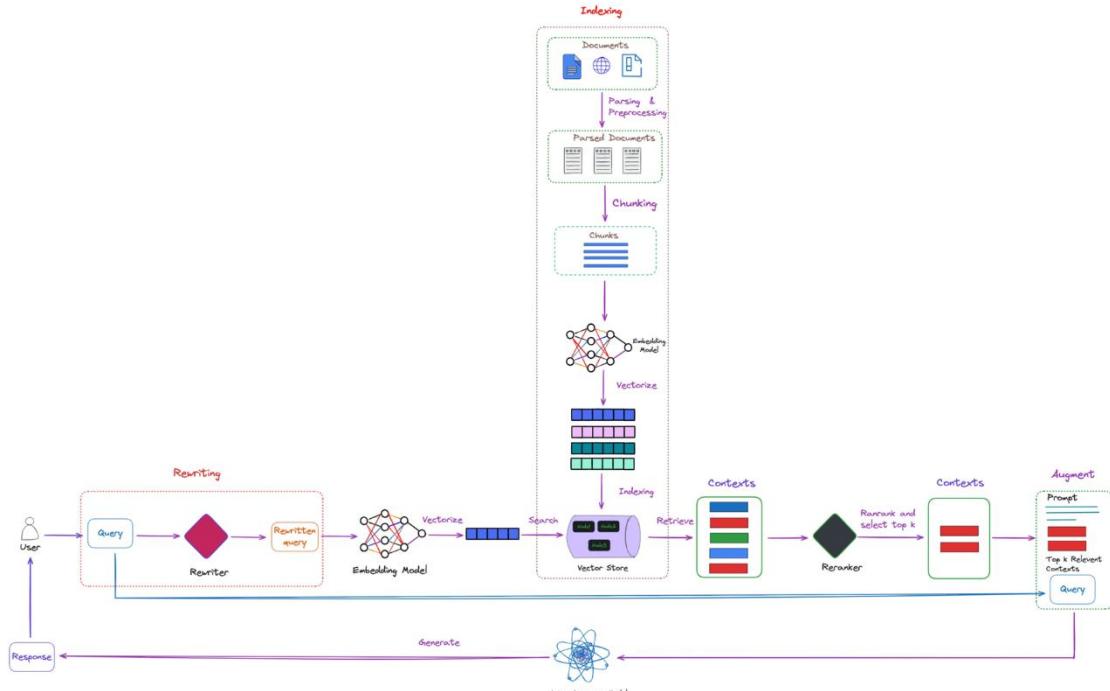
>>>

角色扮演 (RolePlaying) 是一种沟通代理框架，通过使用启发式提示来引导聊天代理完成任务，同时保持与人类意图的一致性。这种方法可以用于生成对话数据，以研究代理社会的行为和能力，为调查对话语言模型提供宝贵的资源。特别是在遵循指令方面进行了全面的研究。

至此，恭喜您已经搭建了一个 Basic RAG，接下来我们将介绍更多 RAG 中的小技巧~

## 2. Rewriting

带着上一章作业的问题来看接下来的章节会更有收获哦！



在实际的应用中，很多时候我们可能会发现，用户的提问是不一定准确的，比如存在错别字，语义顺序颠倒等情况，甚至有时候用户对于自己的 Query 究竟要如何表达也是模糊不清的，而阅读并实践了上一章节的读者可能会有一个直观的感受，就是 Query 不仅会作用在检索的部分，还会作用在最后生成回复的部分，简而言之，Query 的质量可能会极大程度地影响整个 RAG 系统的表现和性能。

因此在这里，我们尝试加入一个对于初始 Query 的改写或者澄清模块，又称为 Rewriting，来尽可能提高 Query 的质量或增强之后 RAG 环节的质量。

我们不妨针对之前的分析来思考一下如何进行改写才能缓解之前提到的问题。

- **错字校正：**修正用户输入中可能存在的错别字的情况
- **句式调整：**重构 Query 来使得句子表达更加通顺严谨

对于这两个常规问题，我们可以使用 LLM 自我调整，通过提示词工程的方式令其自我优化提示词。

original\_query = "我盖如何解决 CAMEL 中文档冲突的问题呢，几个版本的文档可能存在代码结构的冲突"

sys\_msg = '你是 RAG 模块中的 Rewriting 助手，目的是理解用户的提问，并且重新组织和优

化用户的提问表达,修正用户输入中可能存在的错别字的情况并重构提问来使得句子表达更加通顺严谨'

```
from camel.models import ModelFactory
from camel.types import ModelPlatformType, ModelType
from camel.configs import ChatGPTConfig
from camel.agents import ChatAgent

from dotenv import load_dotenv
import os

load_dotenv()
api_key = os.getenv('MODELSCOPE_SDK_TOKEN')

定义模型
model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key=api_key
)
agent = ChatAgent(system_message=sys_msg, model=model)

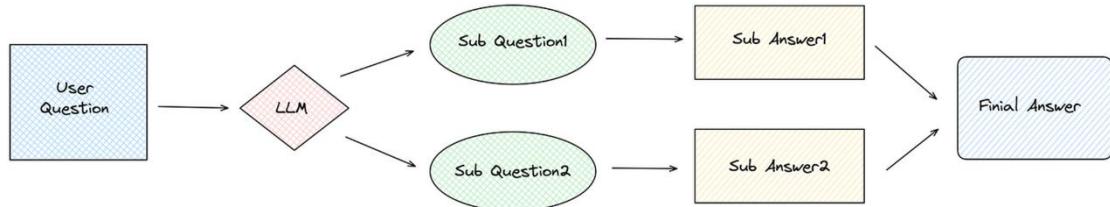
定义 Rewriting 任务
usr_msg = f'用户的原始提问如下: {original_query}, 请优化重写并直接输出新的 Query。新的 Query: '
response = agent.step(usr_msg)
print(response.msgs[0].content)
```

这里优化的轮数取决于需求,你可以让 LLM 不断优化,直到满足你的预期为止。

此外, Rewriting 模块还存在很多其他优化技巧,比如:

- 子问题策略

也称为子查询,是一种用于生成子问题的技术。其核心思想是在问答过程中生成并提出与主问题相关的子问题,以便更好地理解和回答主问题。这些子问题通常更具体,可以帮助系统更深入地理解主问题,从而提高检索准确性和提供正确的答案。



- HyDE 查询转换

论文 [「Precise Zero-Shot Dense Retrieval without Relevance Labels」](#) 提出了一种名为假设文档嵌入 (HyDE) 的方法。HyDE (Hypothetical Document Embeddings) 的本质是使用 LLM 为用户查询生成假设文档。这些文档是根据 LLM 本身的知识生成的,可能包含错误或不准确之处。但是,它们与 RAG 知识库中的文档相关联。然后,通过使用这些假设文档来

检索具有相似向量的真实文档，从而提高检索的准确性。

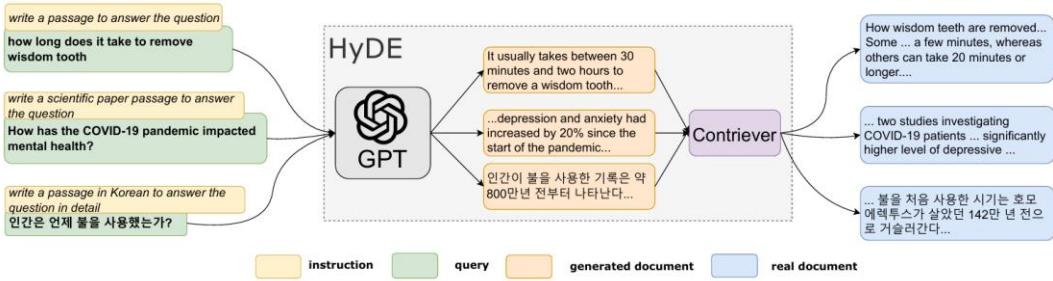


Figure 1: An illustration of the HyDE model. Documents snippets are shown. HyDE serves all types of queries without changing the underlying GPT-3 and Contriever/mContriever models.

我们可以对比一下加入了 Rewriting 之后的回复生成效果和 Basic RAG 的结果

TODO: 对比效果

### 3. Rerank

之前我们提到了可以通过向量和关键字等多种方式和不同的维度来进行检索，为了检索这一步结果的精确性和强相关性，我们往往会引入 Rerank 这一模块来对多路召回(多路指的是向量相似度，关键字匹配，基于规则的匹配，甚至结构化搜索等多种方式并行召回，往往召回的片段结果数量加起来比较多)的文档片段结果进行重排。因为数量越多，召回的内容相关度越可能存在问题，多数情况下 score 最高的片段相关度没问题，但是 top2-5 及往后的相关度就很随机了，这会对最终生成效果造成影响。

Rerank 模块对初步检索结果重新排序可以简单概括为以下几步：

1. **初步检索**: 获取 (多路召回) 初始文档片段
2. **特征计算**: 评估每个文档的相关性
3. **重新排序**: 根据特征得分排序
4. **选择最佳结果**: 根据重排结果倒序排列，取前 TOP-K 个作为最终的最相关结果交给 LLM 生成回复

为了量化我们系统的有效性，我们主要依赖于两个被广泛接受的指标：命中率(hit rate) 和 平均倒数排名(MRR)。

#### 命中率(Hit Rate):

Hit rate 计算在前 k 个检索文档中找到正确答案的查询比例。简单来说，它是关于我们的系统在前几次猜测中正确的频率。

#### 平均倒数排名(MRR):

对于每个查询，MRR 通过查看排名最高的相关文档的排名来评估系统的准确性。具体来说，它是所有查询中这些秩的倒数的平均值。因此，如果第一个相关文档是顶部结果，则倒数排名为 1；如果是第二个，倒数是 1/2，以此类推。

如果还没有太懂，不要紧，我们可以通过 BAAI 提供的 Evaluation 测评的结果来直观的理解，简单而言对于同一个 embedding model，使用不同的 rerank model，mrr 和 hit rate 指标的分数越高，意味着效果越好！ <https://huggingface.co/BAAI/bge-reranker-v2-minicpm-layerwise>

embedding model	bge-en-v1.5 large		bge-m3		openai-small		openai-large		mxbai-embed-large-v1	
	mrr	hit rate	mrr	hit rate	mrr	hit rate	mrr	hit rate	mrr	hit rate
without reranker	65.07	85.1	69.67	88.94	65.69	89.42	67.37	90.38	66.66	88.46
bge-reranckr-base	75.77	90.87	77.48	94.23	75.75	93.27	76.3	94.23	76.63	91.83
bge-reranker-large	75.86	90.87	78.66	94.23	77.09	94.23	77.08	95.67	77.24	92.31
mxbai-rerank-large-v1	72.77	88.46	75.99	93.27	74.62	91.35	74.32	92.31	73.89	89.9
jina-reranker-v1-base-en	75.81	89.9	79.44	93.75	77.64	91.83	77.85	92.79	76.96	91.83
cohere rerank	75.17	90.38	76.23	91.35	76.98	92.79	76.68	93.27	76.43	92.31
ms-marco-MiniLM-L-6-v2	67.92	86.54	69.83	90.38	69.2	88.46	67.99	90.38	68.57	87.98
bge-reranker-v2-m3	78.26	90.87	80.76	94.71	79.38	93.27	79.7	94.71	79.1	92.31
bge-reranker-v2-gemma	75.19	89.9	78.14	93.75	76.74	92.31	76.28	92.31	77.25	91.83
bge-reranker-v2-minicpm-20	81.31	91.83	83.77	95.67	81.92	94.71	83.43	95.19	82.11	92.79
<b>bge-reranker-v2-minicpm-28</b>	<b>81.93</b>	<b>91.83</b>	<b>84.74</b>	<b>95.67</b>	<b>84.01</b>	<b>94.71</b>	<b>83.93</b>	<b>95.19</b>	<b>82.99</b>	<b>93.27</b>
bge-reranker-v2-minicpm-40	80.89	91.83	83.29	95.67	82.89	94.71	82.33	95.19	81.45	92.79

我们这里使用比较经典的 RRF (Reciprocal Rank Fusion) 算法来实现一个 reranker。

倒数排序融合(RRF) 是一种将具有不同相关性指标的多个结果集组合成单个结果集的方法，不同的相关性指标也不必相互关联即可获得高质量的结果。该方法的优势在于不利用相关分數，而仅靠排名计算。

$$RRF-Score(x) = \sum_{i=1}^t \frac{1}{k + r_i(x)}$$

```
from camel.embeddings import SentenceTransformerEncoder
from camel.retrievers import VectorRetriever, BM25Retriever
from camel.storages.vectordb_storages import QdrantStorage
from typing import List, Dict
```

```
def rrf(vector_results: List[Dict], text_results: List[Dict], k: int=10, m: int=60):
```

```

```

使用 RRF 算法对两组检索结果进行重排序

params:

vector\_results (list): 向量召回的结果列表，每个元素是包含'text'的字典

text\_results (list): 文本召回的结果列表，每个元素是包含'text'的字典

k(int): 排序后返回前 k 个

m (int): 超参数

return:

重排序后的结果列表，每个元素是(文档内容, 融合分数)

```

```

```
 doc_scores = {}
```

```
遍历向量检索结果
```

```
for rank, result in enumerate(vector_results):
```

```
 text = result['text']
```

```
 doc_scores[text] = doc_scores.get(text, 0) + 1 / (rank + m)
```

```

遍历文本检索结果
for rank, result in enumerate(text_results):
 text = result['text']
 doc_scores[text] = doc_scores.get(text, 0) + 1 / (rank + m)

按融合分数排序并返回前 k 个结果
sorted_results = sorted(doc_scores.items(), key=lambda x: x[1], reverse=True)[:k]
return sorted_results

初始化检索器
embedding_model = SentenceTransformerEncoder(model_name='intfloat/e5-large-v2')
vector_storage = QdrantStorage(
 vector_dim=embedding_model.get_output_dim(),
 collection="demo_collection",
 path="storage_customized_run",
 collection_name="paper"
)

vr = VectorRetriever(embedding_model=embedding_model, storage=vector_storage)
bm25r = BM25Retriever()

处理文档
content_path = "local_data/camel_paper.pdf"
vr.process(content=content_path)
bm25r.process(content_input_path=content_path)

查询
query = "CAMEL 是什么"
vector_results = vr.query(query=query, top_k=10)
bm25_results = bm25r.query(query=query, top_k=10)

融合排序
rrf_results = rrf(vector_results, bm25_results)
print(rrf_results)
我们可以进一步对比一下加入了 ReRank 之后的回复生成效果和 Basic RAG 的结果
TODO: 对比效果

```

---

## 5. RAG 应用的评估

### 1. 如何评估一个 RAG 应用

RAG (检索增强生成) 是一种结合了信息检索和自然语言生成的技术, 广泛应用于知识问答、客户服务等领域。要实现高质量的 RAG 应用, 必须对其进行全面的评估和优化。本文将从以下几个方面展开介绍: 如何评估 RAG 应用, 如何评估及优化检索模块, 以及如何评估及优化生成模块。

评估 RAG 应用需要综合考虑:

- **检索性能**: 检索的准确率和召回率

检索模块决定了从知识库中找到的文档质量，是 RAG 应用的基础。主要评估指标包括：

- 准确率 (Precision): 检索结果中相关文档的比例。
- 召回率 (Recall): 所有相关文档中被检索出的比例。
- F1 值: 准确率和召回率的调和平均值。
- **生成质量**: 回答的准确性、流畅度和相关性
- 准确性: 回答是否正确。
- 流畅性: 语言是否自然。
- 相关性: 回答是否与问题紧密相关。
- **用户体验**: 响应速度、交互友好性

用户体验评估主要关注应用的交互友好性，包括：

- 响应速度: 回答的生成时间。
- 交互性: 系统界面和交互设计是否直观。
- 稳定性: 系统是否可靠，无明显错误。

## 2. 评估及优化检索模块

**评估检索模块的方法**:

- **指标评估**: 使用准确率、召回率、F1 值等指标

评估检索模块时，常用以下指标：

- 准确率: 衡量检索结果的相关性。
- 召回率: 衡量检索结果的覆盖范围。
- F1 值: 综合考虑准确率和召回率。

**评估检索模块**

```
from camel.embeddings import SentenceTransformerEncoder
from camel.retrievers import VectorRetriever
from camel.storages.vectordb_storages import QdrantStorage
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from nltk.tokenize import word_tokenize
import numpy as np

初始化嵌入模型
embedding_model = SentenceTransformerEncoder(model_name='intfloat/e5-large-v2')

初始化向量存储
vector_storage = QdrantStorage(
 vector_dim=embedding_model.get_output_dim(),
 collection="test_collection",
 path="test_storage",
 collection_name="CAMEL AI 文档"
)

初始化检索器
vr = VectorRetriever(embedding_model=embedding_model, storage=vector_storage)
```

```

处理文档并构建向量数据库
vr.process(
 content="example_document.md",
)
这里可以使用我们用 LLM 生成的一个示例 md 文档作为我们要检索的文档，使用本地的
embedding 模型和向量检索器。
暂时无法在飞书文档外展示此内容
在这里我们设置一些问题和我们预期的标准答案。
test_queries = [
 {
 "query": "什么是 CAMEL AI? ",
 "expected_answers": ["CAMEL AI 是一个开源的、社区驱动的 AI 框架。"]
 },
 {
 "query": "如何开始使用 CAMEL AI? ",
 "expected_answers": ["首先安装框架: `pip install camel-ai`，然后引入必要的模块。"]
 }
],
{
 "query": "CAMEL AI 的主要特点是什么? ",
 "expected_answers": ["模块化设计、易用性和扩展性。"]
}
]

之后，我们定义我们的评估指标，我们计算检索出的结果和我们预期结果之间的余弦相似度，

并且认为超过一定范围时检索的结果是符合预期的。
定义评估指标
def calculate_precision(retrieved, relevant, threshold=0.5):
 """计算精确率 (Precision)，当相似度超过阈值时认为是正确的"""
 correct = 0
 for r in retrieved:
 for rel in relevant:
 similarity = compute_similarity(rel, r)
 if similarity >= threshold:
 correct += 1
 break
 return correct / len(retrieved) if retrieved else 0

def calculate_recall(retrieved, relevant, threshold=0.5):
 """计算召回率 (Recall)，当相似度超过阈值时认为是正确的"""
 correct = 0
 for rel in relevant:
 for r in retrieved:
 similarity = compute_similarity(rel, r)

```

```

 if similarity >= threshold:
 correct += 1
 break
 return correct / len(relevant) if relevant else 0

def calculate_f1(precision, recall):
 """计算 F1 值"""
 return 2 * (precision * recall) / (precision + recall) if (precision + recall) else 0

def compute_similarity(expected, retrieved):
 """计算预期答案与检索结果的相似度"""
 vectorizer = TfidfVectorizer()
 tfidf = vectorizer.fit_transform([expected, retrieved])
 similarity_matrix = cosine_similarity(tfidf, tfidf)
 return similarity_matrix[0, 1]

def evaluate_retrieval(query, expected_answers, threshold=0.5, top_k=1):
 """评估单个查询的检索质量"""
 results = vr.query(query=query, top_k=top_k)
 retrieved_texts = [result["text"] for result in results]

 # 计算精确率、召回率和 F1 值
 precision = calculate_precision(retrieved_texts, expected_answers, threshold)
 recall = calculate_recall(retrieved_texts, expected_answers, threshold)
 f1 = calculate_f1(precision, recall)

 # 计算平均相似度
 similarities = []
 for expected, retrieved in zip(expected_answers, retrieved_texts):
 similarities.append(compute_similarity(expected, retrieved))
 avg_similarity = np.mean(similarities) if similarities else 0

 return {
 "precision": precision,
 "recall": recall,
 "f1": f1,
 "avg_similarity": avg_similarity,
 "retrieved_texts": retrieved_texts
 }

```

下面我们执行评估:

```

执行评估
evaluation_results = []
for test_case in test_queries:
 query = test_case["query"]

```

```
expected_answers = test_case["expected_answers"]

evaluation = evaluate_retrieval(query, expected_answers)

evaluation_results.append({
 "query": query,
 "expected_answers": expected_answers,
 "evaluation": evaluation
})

打印详细结果
print(f"Query: {query}")
print(f"Expected Answers: {expected_answers}")
print(f"Retrieved Results: {evaluation['retrieved_texts']}")
print(f"Precision: {evaluation['precision']:.4f}")
print(f"Recall: {evaluation['recall']:.4f}")
print(f"F1 Score: {evaluation['f1']:.4f}")
print(f"Average Similarity: {evaluation['avg_similarity']:.4f}")
print("-" * 100)

计算整体评估结果
total_precision = sum(result["evaluation"]["precision"] for result in evaluation_results) / len(evaluation_results)
total_recall = sum(result["evaluation"]["recall"] for result in evaluation_results) / len(evaluation_results)
total_f1 = sum(result["evaluation"]["f1"] for result in evaluation_results) / len(evaluation_results)
total_similarity = sum(result["evaluation"]["avg_similarity"] for result in evaluation_results) / len(evaluation_results)

print("\n 整体评估结果:")
print(f"Average Precision: {total_precision:.4f}")
print(f"Average Recall: {total_recall:.4f}")
print(f"Average F1 Score: {total_f1:.4f}")
print(f"Average Similarity: {total_similarity:.4f}")

>>>
Query: 什么是 CAMEL AI?
Expected Answers: ['CAMEL AI 是一个开源的、社区驱动的 AI 框架。']
Retrieved Results: ['CAMEL AI 介绍\n\nCAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。\n\n 主要特点\n\n 模块化设计：用户可以根据需求选择性地加载不同的功能模块。\n\n 易用性：提供了简单易用的 API 接口，降低了开发门槛。\n\n 拓展性：支持多种模型和后端服务，方便用户根据需求进行扩展。'\n\n 常见问题\n\n 如何开始使用 CAMEL
```

AI? \n\n 首先安装框架: pip install camel-ai\n\n 引入必要的模块: from camel import \*\n\n 参考官方文档: CAMEL AI 官方文档']

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Average Similarity: 0.5722

---

Query: 如何开始使用 CAMEL AI?

Expected Answers: ['首先安装框架: `pip install camel-ai`，然后引入必要的模块。']

Retrieved Results: ['CAMEL AI 介绍\n\nCAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。\\n\\n 主要特点\\n\\n 模块化设计：用户可以根据需求选择性地加载不同的功能模块。\\n\\n 易用性：提供了简单易用的 API 接口，降低了开发门槛。\\n\\n 拓展性：支持多种模型和后端服务，方便用户根据需求进行扩展。\\n\\n 常见问题\\n\\n 如何开始使用 CAMEL AI? \\n\\n 首先安装框架: pip install camel-ai\\n\\n 引入必要的模块: from camel import \*\n\n 参考官方文档: CAMEL AI 官方文档']

Precision: 0.0000

Recall: 0.0000

F1 Score: 0.0000

Average Similarity: 0.4752

---

Query: CAMEL AI 的主要特点是什么?

Expected Answers: ['模块化设计、易用性和扩展性。']

Retrieved Results: ['CAMEL AI 介绍\n\nCAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。\\n\\n 主要特点\\n\\n 模块化设计：用户可以根据需求选择性地加载不同的功能模块。\\n\\n 易用性：提供了简单易用的 API 接口，降低了开发门槛。\\n\\n 拓展性：支持多种模型和后端服务，方便用户根据需求进行扩展。\\n\\n 常见问题\\n\\n 如何开始使用 CAMEL AI? \\n\\n 首先安装框架: pip install camel-ai\\n\\n 引入必要的模块: from camel import \*\n\n 参考官方文档: CAMEL AI 官方文档']

Precision: 0.0000

Recall: 0.0000

F1 Score: 0.0000

Average Similarity: 0.0502

---

...

Average Precision: 0.3333

Average Recall: 0.3333

Average F1 Score: 0.3333

Average Similarity: 0.3659

通过以上的步骤，我们可以完成一个简单检索模块的评估。下面我们来看一下如何优化我们

的检索模块。

首先我们会发现在上面第三条检索的结果其实是涵盖了我们的预期结果的，但是由于我们使用的是基于TF-IDF的余弦相似度。导致了其得分较低。

首先，我们简单介绍一下TF-IDF的基本原理和它在相似度计算中的应用。TF-IDF（词频-逆向文件频率）是一种用于信息检索和文本挖掘的常用加权技术，用于评估一个词在文档中的重要程度。其核心思想是：如果某个词在一篇文档中出现的频率高，并且在其他文档中很少出现，则认为该词具有很好的类别区分能力。

这种方法在我们第三条结果出现的问题是：

**忽略语义信息**：TF-IDF 基于词频统计，无法捕捉词语之间的语义关系。例如，“汽车”和“车辆”在语义上是相近的，但 TF-IDF 会将其视为完全不同的词。

而且预期的结果和检索到的结果字数差距较大，导致两条结果的在基于TF-IDF的余弦相似度下相差较大。这里我们可以尝试使用 Embedding 模型来将结果转成向量之后再计算余弦相似度。它的特点如下：

**优点**：精准捕捉语义和语境。

**缺点**：计算资源消耗大。

#我们将原有的 **compute\_similarity** 函数替换一下

```
def compute_similarity(expected, retrieved):
"""计算预期答案与检索结果的相似度"""
vectorizer = TfidfVectorizer()
tfidf = vectorizer.fit_transform([expected, retrieved])
similarity_matrix = cosine_similarity(tfidf, tfidf)
return similarity_matrix[0, 1]

def compute_similarity(expected, retrieved):
 """计算预期答案与检索结果的相似度"""
 embeddings = embedding_model.embed_list([expected, retrieved])
 return cosine_similarity([embeddings[0]], [embeddings[1]])[0][0]

然后再重新运行一下评估函数：
执行评估
evaluation_results = []
for test_case in test_queries:
 query = test_case["query"]
 expected_answers = test_case["expected_answers"]

 evaluation = evaluate_retrieval(query, expected_answers)

 evaluation_results.append({
 "query": query,
 "expected_answers": expected_answers,
 "evaluation": evaluation
 })

 # 打印详细结果
```

```
print(f"Query: {query}")
print(f"Expected Answers: {expected_answers}")
print(f"Retrieved Results: {evaluation['retrieved_texts']}")
print(f"Precision: {evaluation['precision']:.4f}")
print(f"Recall: {evaluation['recall']:.4f}")
print(f"F1 Score: {evaluation['f1']:.4f}")
print(f"Average Similarity: {evaluation['avg_similarity']:.4f}")
print("-" * 100)

计算整体评估结果
total_precision = sum(result["evaluation"]["precision"] for result in evaluation_results) / len(evaluation_results)
total_recall = sum(result["evaluation"]["recall"] for result in evaluation_results) / len(evaluation_results)
total_f1 = sum(result["evaluation"]["f1"] for result in evaluation_results) / len(evaluation_results)
total_similarity = sum(result["evaluation"]["avg_similarity"] for result in evaluation_results) / len(evaluation_results)

print("\n 整体评估结果:")
print(f"Average Precision: {total_precision:.4f}")
print(f"Average Recall: {total_recall:.4f}")
print(f"Average F1 Score: {total_f1:.4f}")
print(f"Average Similarity: {total_similarity:.4f}")
```

>>>

Query: 什么是 CAMEL AI?

Expected Answers: ['CAMEL AI 是一个开源的、社区驱动的 AI 框架。']

Retrieved Results: ['CAMEL AI 介绍\n\nCAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。\\n\\n 主要特点\\n\\n 模块化设计：用户可以根据需求选择性地加载不同的功能模块。\\n\\n 易用性：提供了简单易用的 API 接口，降低了开发门槛。\\n\\n 拓展性：支持多种模型和后端服务，方便用户根据需求进行扩展。\\n\\n 常见问题\\n\\n 如何开始使用 CAMEL AI？\\n\\n 首先安装框架：`pip install camel-ai`\\n\\n 引入必要的模块：`from camel import \*`\\n\\n 参考官方文档：CAMEL AI 官方文档']

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Average Similarity: 0.9024

---

---

Query: 如何开始使用 CAMEL AI?

Expected Answers: ['首先安装框架：`pip install camel-ai`，然后引入必要的模块。']

Retrieved Results: ['CAMEL AI 介绍\n\nCAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在

简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。  
主要特点  
模块化设计：用户可以根据需求选择性地加载不同的功能模块。  
易用性：提供了简单易用的 API 接口，降低了开发门槛。  
拓展性：支持多种模型和后端服务，方便用户根据需求进行扩展。  
常见问题  
如何开始使用 CAMEL AI？  
首先安装框架：pip install camel-ai  
引入必要的模块：from camel import \*  
参考官方文档：CAMEL AI 官方文档']

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Average Similarity: 0.9148

---

Query: CAMEL AI 的主要特点是什么？

Expected Answers: ['模块化设计、易用性和扩展性。']

Retrieved Results: ['CAMEL AI 介绍  
CAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。  
主要特点  
模块化设计：用户可以根据需求选择性地加载不同的功能模块。  
易用性：提供了简单易用的 API 接口，降低了开发门槛。  
拓展性：支持多种模型和后端服务，方便用户根据需求进行扩展。  
常见问题  
如何开始使用 CAMEL AI？  
首先安装框架：pip install camel-ai  
引入必要的模块：from camel import \*  
参考官方文档：CAMEL AI 官方文档']

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Average Similarity: 0.8000

---

Average Precision: 1.0000

Average Recall: 1.0000

Average F1 Score: 1.0000

Average Similarity: 0.8724

可以发现，这次评估的结果更加符合我们人类的标准。

另外我们可以发现，我们每次检索到的内容其实完全一样，这是因为 VectorRetriever 会默认将文档按照 500 字为间隔来划分，我们将这里我们可以调整一下 chunk 的大小，之后重新划分文档：

```
初始化向量存储
vector_storage = QdrantStorage(
 vector_dim=embedding_model.get_output_dim(),
 collection="test_collection",
 path="test_storage",
 collection_name="CAMEL AI 文档"
)
```

```

初始化检索器
vr = VectorRetriever(embedding_model=embedding_model, storage=vector_storage)

vr.process(
 content="example_document.md",
 max_characters=100,
 should_chunk=True,
)
然后再次检索：

执行评估
evaluation_results = []
for test_case in test_queries:
 query = test_case["query"]
 expected_answers = test_case["expected_answers"]

 evaluation = evaluate_retrieval(query, expected_answers)

 evaluation_results.append({
 "query": query,
 "expected_answers": expected_answers,
 "evaluation": evaluation
 })

打印详细结果
print(f"Query: {query}")
print(f"Expected Answers: {expected_answers}")
print(f"Retrieved Results: {evaluation['retrieved_texts']}")
print(f"Precision: {evaluation['precision']:.4f}")
print(f"Recall: {evaluation['recall']:.4f}")
print(f"F1 Score: {evaluation['f1']:.4f}")
print(f"Average Similarity: {evaluation['avg_similarity']:.4f}")
print("-" * 100)

计算整体评估结果
total_precision = sum(result["evaluation"]["precision"] for result in evaluation_results) / len(evaluation_results)
total_recall = sum(result["evaluation"]["recall"] for result in evaluation_results) / len(evaluation_results)
total_f1 = sum(result["evaluation"]["f1"] for result in evaluation_results) / len(evaluation_results)
total_similarity = sum(result["evaluation"]["avg_similarity"] for result in evaluation_results) / len(evaluation_results)

print("\n整体评估结果:")

```

```
print(f"Average Precision: {total_precision:.4f}")
print(f"Average Recall: {total_recall:.4f}")
print(f"Average F1 Score: {total_f1:.4f}")
print(f"Average Similarity: {total_similarity:.4f}")
```

>>>

Query: 什么是 CAMEL AI?

Expected Answers: ['CAMEL AI 是一个开源的、社区驱动的 AI 框架。']

Retrieved Results: ['CAMEL AI 介绍\n\nCAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。']

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Average Similarity: 0.9584

---

-----  
Query: 如何开始使用 CAMEL AI?

Expected Answers: ['首先安装框架: `pip install camel-ai`，然后引入必要的模块。']

Retrieved Results: ['常见问题\n\n如何开始使用 CAMEL AI? \n\n首先安装框架: pip install camel-ai\n\n引入必要的模块: from camel import \*\n\n参考官方文档: CAMEL AI 官方文档']

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Average Similarity: 0.9604

---

-----  
Query: CAMEL AI 的主要特点是什么?

Expected Answers: ['模块化设计、易用性和扩展性。']

Retrieved Results: ['CAMEL AI 介绍\n\nCAMEL AI 是一个开源的、社区驱动的 AI 框架，旨在简化 AI 应用的开发和部署。该框架提供了多种功能模块，包括数据加载、特征工程、模型训练和部署等。']

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Average Similarity: 0.8103

---

-----  
...  
Average Precision: 1.0000

Average Recall: 1.0000

Average F1 Score: 1.0000

Average Similarity: 0.9097

可以发现效果又更近了一步。

更多的调整方向可以参考以下几点：

- **参数调整**: 优化嵌入模型和检索算法的参数

优化嵌入模型和检索算法的参数，例如：

- 调整嵌入向量的维度。
- 优化检索的相似度计算方法（如欧几里得距离、余弦相似度）。
- 在自身场景下微调嵌入模型通过效果是明显的
- **数据增强**: 扩充知识库，提高覆盖面

通过扩充知识库提高覆盖面，例如：

- 添加更多高质量的知识数据。
- 利用数据增强技术生成多样化的知识表达。
- **检索策略优化**：
  - 分阶段检索
  - 混合检索(如 TF-IDF + 语义模型)
  - 实时反优化

### 3. 评估及优化生成模块

优化生成模块的方法：

- **质量评估**: 使用 BLEU、ROUGE 等自动指标和人工评估

生成模块的评估可以分为自动评估和人工评估：

- **自动评估**: 使用 BLEU、ROUGE 等指标衡量生成文本的质量。
- **人工评估**: 通过专家或用户打分，评估文本的准确性、流畅性和相关性。
- **上下文增强**: 提供更丰富的上下文信息
- 为生成模型提供更丰富的上下文信息，例如问题背景或用户历史记录。

代码示例：以下展示了如何评估生成模块的回答质量。

```
from rouge_score import rouge_scorer
from nltk.translate.bleu_score import sentence_bleu
from nltk.tokenize import word_tokenize

示例数据
reference = "RAG combines retrieval and generation for QA."
generated = "RAG integrates retrieval and generation for question answering."

使用 ROUGE 评估
scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)
scores = scorer.score(reference, generated)

使用 BLEU 评估
reference_tokens = word_tokenize(reference)
generated_tokens = word_tokenize(generated)
bleu_score = sentence_bleu([reference_tokens], generated_tokens)

print(f'ROUGE-1: {scores["rouge1"].fmeasure:.2f}')
print(f'ROUGE-L: {scores["rougeL"].fmeasure:.2f}')
print(f'BLEU Score: {bleu_score:.2f}')
```

## 6. Graph RAG 应用实战

### 1. Graph RAG 以及与传统 RAG 的优劣

Graph RAG 是将知识图谱 (Knowledge Graph) 引入检索增强生成 (RAG) 框架的一种扩展形式。它利用结构化的知识显式表示实体及其关系，从而显著提升系统的推理能力和回答准确性。相比于传统 RAG 仅基于向量检索的机制，Graph RAG 在复杂任务中具有独特的优势。

**优点：深度理解：**通过显式的实体和关系表达，Graph RAG 能够支持复杂的问答和逻辑推理。**高准确性：**由于知识图谱中知识的结构化表示，减少了生成模型出现幻觉现象 (hallucination) 的可能性。

**缺点：构建复杂：**知识图谱的构建和维护需要大量时间和资源投入。**灵活性有限：**对于实时更新和动态信息支持较差，难以应对快速变化的场景。

### 2. 图数据库介绍

图数据库是 Graph RAG 的核心组件，用于存储和管理知识图谱。它能够高效处理复杂的关系查询，同时提供灵活的结构和强大的查询能力。

**特点：高效的关系查询：**图数据库优化了实体与实体之间关系的查找和操作，性能优于传统关系型数据库。**灵活的结构：**支持动态添加节点（实体）和边（关系），能够应对多样化的数据变化。**强大的查询语言：**如 Neo4j 的 Cypher，方便用户编写复杂查询逻辑。

#### 常用图数据库：

- **Neo4j**: 功能强大，易于使用的企业级图数据库。
- **JanusGraph**: 分布式的开源图数据库，适合大规模图数据管理。

### 3. 构建三元组并上传图数据库

知识图谱的核心是三元组 (Triplets)，由节点、关系、属性组成：

**节点 (Node/Entity)**：表示实体，是图中的顶点

**关系 (Relationship/Edge)**：表示实体之间的联系，是图中的边

**属性 (Property/Attribute)**：节点或关系的特征描述

例如我们的有一段文本：

text\_example = """

CAMEL 和 DATAWHALE 是两个活跃的 AI 开源社区。

CAMEL 致力于推动人工智能技术的发展，帮助开发者更好地进行 AI 研究和应用。

DateWhale 通过开源项目和社区活动，促进知识分享和技术交流。这两个社区都为 AI 领域的创新和进步做出了重要贡献。

.....

我们可以从中找出 CAMEL、DataWhale 等等这些节点，那么 CAMEL 和人工智能技术的发展之间的关系可以表述为：CAMEL“致力于”人工智能技术的发展。而 CAMEL 的属性可以为“AI 开源社区”。

### 1. 信息抽取

之前我们可能通过一些 NLP 技术来做信息的抽取，现在我们可以直接使用 LLM 来完成这件事情。例如我们通过一些简单的 prompt 工程来完成这个目的：

```
from camel.models import ModelFactory
from camel.types import ModelPlatformType
from camel.loaders import UnstructuredIO
from camel.agents import ChatAgent
```

```
model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key='你的 api 密钥'
)

sys_msg = """
你是一个图数据库专家，你的任务是从给定的内容中提取实体和关系，构建知识图谱。
"""
```

知识图谱的基本组成：

1. 节点 (Node): 表示实体，包含 id 和 type 属性
2. 关系 (Relationship): 表示实体之间的联系
3. 属性 (Property): 节点或关系的附加信息

示例输入：

```
"小明，2008年9月13日出生，男。五行属水，性格开朗。"
```

期望输出：

Nodes:

```
Node(id="小明", type="人")
Node(id="水", type="五行")
Node(id="开朗", type="性格")
Node(id="2008年9月13日", type="时间")
```

Relationships:

```
Relationship(subj=Node(id="小明", type="人"), obj=Node(id="水", type="五行"), type="五行属性")
Relationship(subj=Node(id="小明", type="人"), obj=Node(id="开朗", type="性格"), type="性格特征")
Relationship(subj=Node(id="小明", type="人"), obj=Node(id="2008年9月13日", type="时间"), type="出生日期")
```

请从以下内容中提取实体和关系：

.....

```
graphprocessor = ChatAgent(
 system_message=sys_msg,
 model=model,
)
```

text\_example = """

CAMEL 和 DATAWHALE 是两个活跃的 AI 开源社区。

CAMEL 致力于推动人工智能技术的发展，帮助开发者更好地进行 AI 研究和应用。

DateWhale 通过开源项目和社区活动，促进知识分享和技术交流。这两个社区都为 AI 领域的创新和进步做出了重要贡献。

我们首先通过一个 prompt 来定义来 agent 的功能，以及设置好我们案例文本。

```
response = graphprocessor.step(input_message= text_example)
print (response.msgs[0].content)
```

>>>

Nodes:

- Node(id="CAMEL", type="AI 开源社区")
- Node(id="DATAWHALE", type="AI 开源社区")
- Node(id="推动人工智能技术的发展", type="目标")
- Node(id="帮助开发者更好地进行 AI 研究和应用", type="目标")
- Node(id="通过开源项目和社区活动", type="方法")
- Node(id="促进知识分享和技术交流", type="目标")
- Node(id="AI 领域的创新和进步", type="贡献")

Relationships:

- Relationship(subj=Node(id="CAMEL", type="AI 开源社区"), obj=Node(id="推动人工智能技术的发展", type="目标"), type="致力于")
- Relationship(subj=Node(id="CAMEL", type="AI 开源社区"), obj=Node(id="帮助开发者更好地进行 AI 研究和应用", type="目标"), type="帮助")
- Relationship(subj=Node(id="DATAWHALE", type="AI 开源社区"), obj=Node(id="通过开源项目和社区活动", type="方法"), type="通过")
- Relationship(subj=Node(id="DATAWHALE", type="AI 开源社区"), obj=Node(id="促进知识分享和技术交流", type="目标"), type="促进")
- Relationship(subj=Node(id="CAMEL", type="AI 开源社区"), obj=Node(id="AI 领域的创新和进步", type="贡献"), type="做出贡献")
- Relationship(subj=Node(id="DATAWHALE", type="AI 开源社区"), obj=Node(id="AI 领域的创新和进步", type="贡献"), type="做出贡献")

可以看到，对于信息抽取这个任务，LLM 能完成的相当出色。当然如果不想自己写 prompt 的话，CAMEL 也有预设好的 KnowledgeGraphAgent 供我们一键调用，更改这里的参数 `parse_graph_elements` 可以设定 Agent 返回的信息格式，False 返回的是一个字符串，设置为 True 则返回一个自定义的 GraphElement 类：

```
uio = UnstructuredIO()
kg_agent = KnowledgeGraphAgent(model=model)
```

```
从给定文本创建一个元素
element_example = uio.create_element_from_text(
 text=text_example, element_id="0"
)
```

```
让知识图谱 Agent 提取节点和关系信息
ans_element = kg_agent.run(element_example, parse_graph_elements=False)
```

```

print(ans_element)

>>>
Nodes

- Node(id='CAMEL', type='Community')
- Node(id='DATAWHALE', type='Community')
- Node(id='AI', type='Field')
- Node(id='Developers', type='Group')
- Node(id='Research and Application', type='Activity')
- Node(id='Knowledge Sharing', type='Activity')
- Node(id='Technical Exchange', type='Activity')

Relationships

- Relationship(subj=Node(id='CAMEL', type='Community'), obj=Node(id='AI', type='Field'),
type='ContributesTo')
- Relationship(subj=Node(id='CAMEL', type='Community'), obj=Node(id='Developers',
type='Group'), type='Helps')
- Relationship(subj=Node(id='CAMEL', type='Community'), obj=Node(id='Research and
Application', type='Activity'), type='Promotes')
- Relationship(subj=Node(id='DATAWHALE', type='Community'), obj=Node(id='AI',
type='Field'), type='ContributesTo')
- Relationship(subj=Node(id='DATAWHALE', type='Community'), obj=Node(id='Knowledge
Sharing', type='Activity'), type='Facilitates')
- Relationship(subj=Node(id='DATAWHALE', type='Community'), obj=Node(id='Technical
Exchange', type='Activity'), type='Facilitates')

```

## 2. 上传数据库

现在我们有了节点和关系信息，下一步就该把他们上传到数据库里了。

首先我们要初始化我们的图数据库，这里我们可以使用 [Neo4j\(点击跳转\)](#):

```
from camel.storages import Neo4jGraph
```

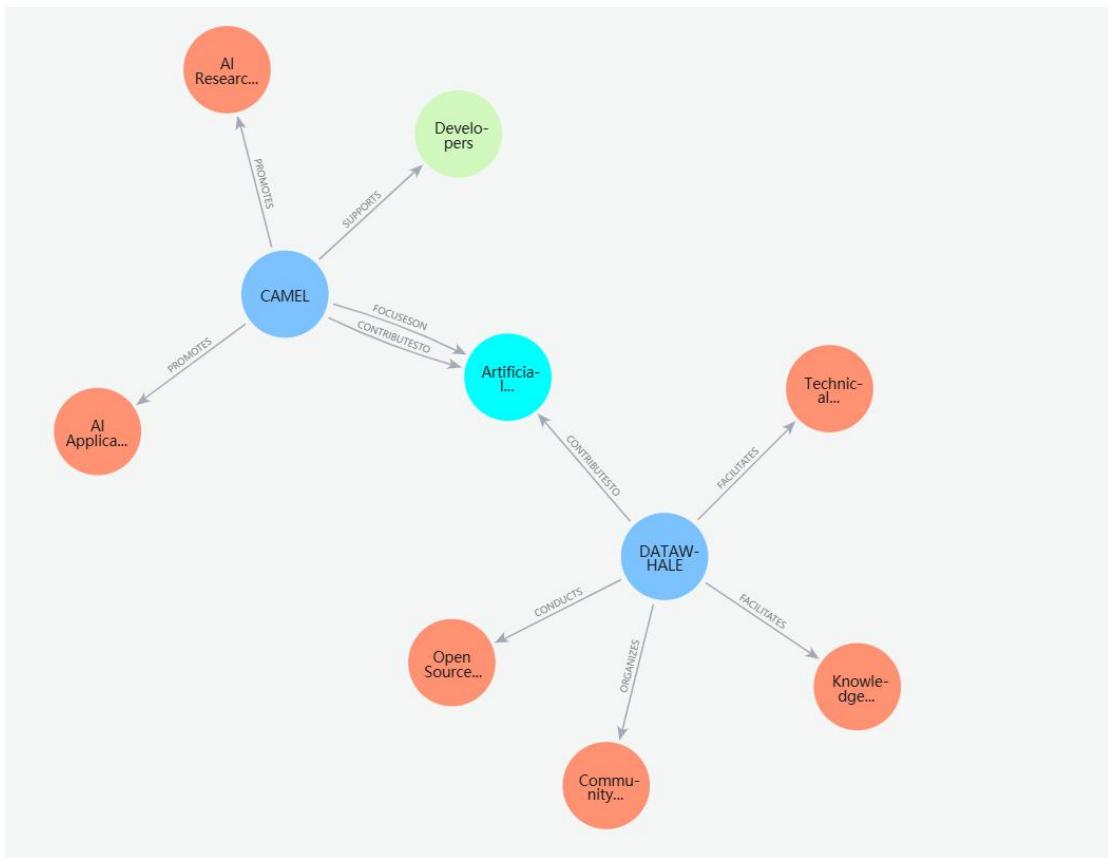
```
n4j = Neo4jGraph(
 url="你的 url",
 username="neo4j",
 password="你的 password",
)
```

CAMEL 已经将上传到图数据这一个操作为我们一键集成，我们只需要使用一个简单的命令调用即可(需要搭配 KnowledgeGraphAgent 一起使用，设定 parse\_graph\_elements=True):

```
检查实体信息
graph_elements = kg_agent.run(element_example, parse_graph_elements=True)
print(graph_elements)
```

```
>>>
```

```
nodes=[Node(id='CAMEL', type='Community', properties={'source': 'agent_created'}),
Node(id='DATAWHALE', type='Community', properties={'source': 'agent_created'}),
Node(id='Artificial Intelligence', type='Field', properties={'source': 'agent_created'}),
Node(id='Developers', type='Group', properties={'source': 'agent_created'}),
Node(id='Knowledge Sharing', type='Activity', properties={'source': 'agent_created'}),
Node(id='Technical Exchange', type='Activity', properties={'source': 'agent_created'})]
relationships=[Relationship(subj=Node(id='CAMEL', type='Community', properties={'source': 'agent_created}), obj=Node(id='Artificial Intelligence', type='Field', properties={'source': 'agent_created'}), type='PromotesDevelopment', properties={'source': 'agent_created'}),
Relationship(subj=Node(id='CAMEL', type='Community', properties={'source': 'agent_created'}), obj=Node(id='Developers', type='Group', properties={'source': 'agent_created'}), type='Helps', properties={'source': 'agent_created'}),
Relationship(subj=Node(id='DATAWHALE', type='Community', properties={'source': 'agent_created'}), obj=Node(id='Artificial Intelligence', type='Field', properties={'source': 'agent_created'}), type='Promotes', properties={'source': 'agent_created'}),
Relationship(subj=Node(id='DATAWHALE', type='Community', properties={'source': 'agent_created'}), obj=Node(id='Knowledge Sharing', type='Activity', properties={'source': 'agent_created'}), type='Facilitates', properties={'source': 'agent_created'}),
Relationship(subj=Node(id='DATAWHALE', type='Community', properties={'source': 'agent_created'}), obj=Node(id='Technical Exchange', type='Activity', properties={'source': 'agent_created'}), type='Facilitates', properties={'source': 'agent_created'}),
Relationship(subj=Node(id='CAMEL', type='Community', properties={'source': 'agent_created'}), obj=Node(id='Artificial Intelligence', type='Field', properties={'source': 'agent_created'}), type='ContributesTo', properties={'source': 'agent_created'}),
Relationship(subj=Node(id='DATAWHALE', type='Community', properties={'source': 'agent_created'}), obj=Node(id='Artificial Intelligence', type='Field', properties={'source': 'agent_created'}), type='ContributesTo', properties={'source': 'agent_created'})]
source=<unstructured.documents.elements.Text object at 0x000001BB8B599460>
将提取的图信息添加到 Neo4j 数据库中:
n4j.add_graph_elements(graph_elements=[graph_elements])
```



我们再回到 neo4j 就可以看到我们构建的知识图谱啦！

#### 4. 实践案例

在这个案例中，我们将展示如何以混合方式运行 RAG，结合向量检索和知识图谱检索，以查询和探索存储的知识。

首先我们使用本地部署的 e5-large-v2 作为我们的 embedding 模型(如果本地没有的话会自动下载)。

```

from camel.models import ModelFactory
from camel.types import ModelPlatformType, ModelType
from camel.loaders import UnstructuredIO
from camel.storages import Neo4jGraph
from camel.retrievers import AutoRetriever
from camel.types import StorageType
from camel.agents import ChatAgent, KnowledgeGraphAgent
from camel.messages import BaseMessage
from camel.embeddings import SentenceTransformerEncoder

设置检索器
camel_retriever = AutoRetriever(
 vector_storage_local_path="local_data/embedding_storage",
 storage_type=StorageType.QDRANT,
 embedding_model=SentenceTransformerEncoder(model_name='intfloat/e5-large-v2'),
)

```

```
model = ModelFactory.create(
 model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
 model_type="Qwen/Qwen2.5-72B-Instruct",
 url='https://api-inference.modelscope.cn/v1/',
 api_key="
)

n4j = Neo4jGraph(
 url="你的 url",
 username="你的 username",
 password="你的密码",
)

uio = UnstructuredIO()
kg_agent = KnowledgeGraphAgent(model=model)
之后我们设置好我们想了解的内容，这里我们首先让它根据向量检索的方式来检索相关信息：
设置 query
query="DataWhale 和 CAMEL 之间是什么关系。"

使用向量检索器获取相关内容
vector_result = camel_retriever.run_vector_retriever(
 query=query,
 contents="""在一个神奇的动物王国里，住着一只聪明的骆驼，名叫 CAMEL，以及一只智慧的鲸鱼，名叫 DataWhale。CAMEL 生活在辽阔的沙漠中，而 DataWhale 则畅游在无边无际的海洋里。虽然它们的栖息地截然不同，但它们都对探索新知充满了浓厚的兴趣。"""
```

有一天，CAMEL 在沙漠中偶然发现了一块古老的石板，上面刻满了奇怪的符号和图案。CAMEL 意识到这可能是一份通往巨大宝藏的地图，但它无法解读这些符号。于是，它决定去寻找帮助。

与此同时，DataWhale 在海洋深处发现了一卷古老的羊皮纸，上面也布满了类似的符号。这让 DataWhale 感到非常好奇，因为它知道这些符号可能蕴藏着某种秘密，但它同样无法破译。

通过一只飞翔在天空中的信使鸟，CAMEL 和 DataWhale 得知彼此的发现，并决定合作解开这个谜团。骆驼踏上了漫长的旅程，穿越沙漠来到了海洋边缘，而鲸鱼则游到岸边，与 CAMEL 会合。

在一起研究这些古老的符号时，CAMEL 运用了它在沙漠中磨炼出的耐心和智慧，而 DataWhale 则利用它在海洋中获得的知识和洞察力。经过一番努力，它们终于解读出这些符号的含义。原来，这些符号描绘的是一个连接沙漠和海洋的秘密通道，通道的尽头藏着一座知识的宝库。

满怀着激动与期待, CAMEL 和 DataWhale 沿着地图所指引的方向, 一同踏上了寻找宝库的旅程。一路上, 它们克服了重重困难, 彼此扶持与信任, 终于找到了那座传说中的宝库。里面并不是金银财宝, 而是无数的古老书籍和知识卷轴。

通过这次冒险, CAMEL 和 DataWhale 不仅获得了丰富的知识, 还建立了深厚的友谊。它们将这些知识带回了各自的家园, 与其他动物分享, 从而推动了整个动物王国的进步与发展。

这个故事告诉我们, 不同背景和能力的个体, 通过合作与分享, 能够创造出超越想象的成果, 开拓出更为广阔的未来。CAMEL 和 DataWhale 的友谊也成为了动物王国中流传不息的传奇。

```
'''',
)
```

```
打印向量检索的结果
print(vector_result)
```

>>>

```
{'Original Query': 'DataWhale 和 CAMEL 之间是什么关系。', 'Retrieved Context': ['在一个神奇的动物王国里, 住着一只聪明的骆驼, 名叫 CAMEL, 以及一只智慧的鲸鱼, 名叫 DataWhale。CAMEL 生活在辽阔的沙漠中, 而 DataWhale 则畅游在无边无际的海洋里。虽然它们的栖息地截然不同, 但它们都对探索新知充满了浓厚的兴趣。\\n\\n 有一天, CAMEL 在沙漠中偶然发现了一块古老的石板, 上面刻满了奇怪的符号和图案。CAMEL 意识到这可能是一份通往巨大宝藏的地图, 但它无法解读这些符号。于是, 它决定去寻找帮助。\\n\\n 与此同时, DataWhale 在海洋深处发现了一卷古老的羊皮纸, 上面也布满了类似的符号。这让 DataWhale 感到非常好奇, 因为它知道这些符号可能蕴藏着某种秘密, 但它同样无法破译。\\n\\n 通过一只飞翔在天空中的信使鸟, CAMEL 和 DataWhale 得知彼此的发现, 并决定合作解开这个谜团。骆驼踏上了漫长的旅程, 穿越沙漠来到了海洋边缘, 而鲸鱼则游到岸边, 与 CAMEL 会合。']}
```

接下来, 我们可以创建我们的知识图谱:

```
从 CRAB 网站解析内容, 并使用知识图谱 Agent 创建知识图谱数据, 将信息存储到图数据库中。
```

```
elements = uio.create_element_from_text(
```

```
text='''在一个神奇的动物王国里, 住着一只聪明的骆驼, 名叫 CAMEL, 以及一只智慧的鲸鱼, 名叫 DataWhale。CAMEL 生活在辽阔的沙漠中, 而 DataWhale 则畅游在无边无际的海洋里。虽然它们的栖息地截然不同, 但它们都对探索新知充满了浓厚的兴趣。'
```

有一天, CAMEL 在沙漠中偶然发现了一块古老的石板, 上面刻满了奇怪的符号和图案。CAMEL 意识到这可能是一份通往巨大宝藏的地图, 但它无法解读这些符号。于是, 它决定去寻找帮助。

与此同时, DataWhale 在海洋深处发现了一卷古老的羊皮纸, 上面也布满了类似的符号。这让 DataWhale 感到非常好奇, 因为它知道这些符号可能蕴藏着某种秘密, 但它同样无法破译。

通过一只飞翔在天空中的信使鸟，CAMEL 和 DataWhale 得知彼此的发现，并决定合作解开这个谜团。骆驼踏上了漫长的旅程，穿越沙漠来到了海洋边缘，而鲸鱼则游到岸边，与 CAMEL 会合。

在一起研究这些古老的符号时，CAMEL 运用了它在沙漠中磨炼出的耐心和智慧，而 DataWhale 则利用它在海洋中获得的知识和洞察力。经过一番努力，它们终于解读出这些符号的含义。原来，这些符号描绘的是一个连接沙漠和海洋的秘密通道，通道的尽头藏着一座知识的宝库。

满怀着激动与期待，CAMEL 和 DataWhale 沿着地图所指引的方向，一同踏上了寻找宝库的旅程。一路上，它们克服了重重困难，彼此扶持与信任，终于找到了那座传说中的宝库。里面并不是金银财宝，而是无数的古老书籍和知识卷轴。

通过这次冒险，CAMEL 和 DataWhale 不仅获得了丰富的知识，还建立了深厚的友谊。它们将这些知识带回了各自的家园，与其他动物分享，从而推动了整个动物王国的进步与发展。

这个故事告诉我们，不同背景和能力的个体，通过合作与分享，能够创造出超越想象的成果，开拓出更为广阔的未来。CAMEL 和 DataWhale 的友谊也成为了动物王国中流传不息的传奇。

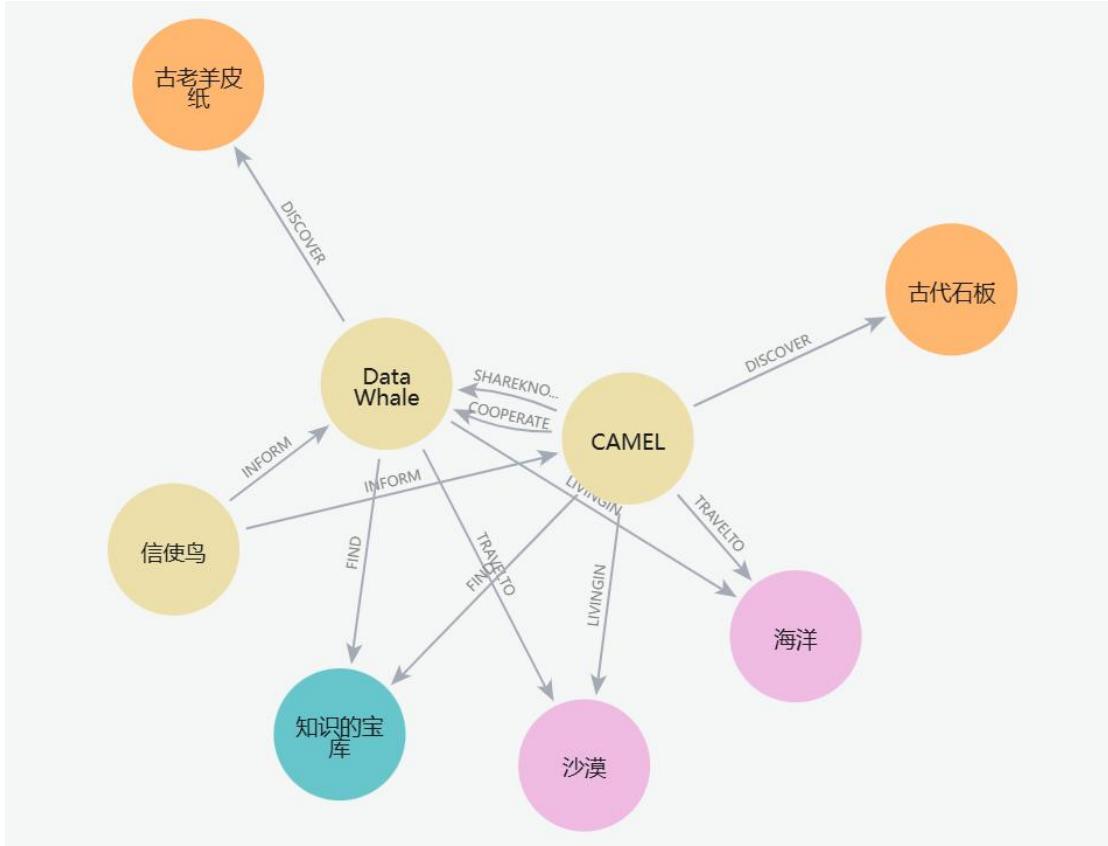
.....

)

)

```
graph_element = kg_agent.run(elements, parse_graph_elements=True)
n4j.add_graph_elements(graph_elements=[graph_element])
```

下面这个漂亮的图谱就是我们创建的结果啦！



```

根据 query 创建一个元素
query_element = uio.create_element_from_text(
 text=query, element_id="1"
)

让知识图谱 agent 从查询中提取节点和关系信息
ans_element = kg_agent.run(query_element, parse_graph_elements=True)

>>>
nodes=[Node(id='DataWhale', type='Organization', properties={'source': 'agent_created'}),
Node(id='CAMEL', type='Organization', properties={'source': 'agent_created'})]
relationships=[Relationship(subj=Node(id='DataWhale', type='Organization', properties={'source': 'agent_created'}), obj=Node(id='CAMEL', type='Organization', properties={'source': 'agent_created'}), type='合作关系', properties={'source': 'agent_created'})]
source=<unstructured.documents.elements.Text object at 0x00000202AF30AC30>

有细心的小伙伴应该发现了，我们在 text 中并没有给 DataWhale 或者 CAMEL 有 'Organization' 的描述，但是最后的结果却含有，这是因为在使用 KnowledgeGraphAgent 的 run 函数时会将检索的到的结果再经过一遍大模型的润色，我们如果想查询原始的数据可以使用 neo4j 的查询语句：

匹配知识图谱存储中的实体：
匹配从 query 中获得的实体在知识图谱存储内容中的信息
kg_result = []

```

```

for node in ans_element.nodes:
 n4j_query = f"""
MATCH (n {{id: '{node.id}'}})-[r]->(m)
RETURN 'Node ' + n.id + '(label: ' + labels(n)[0] + ') has relationship ' + type(r) + ' with Node
' + m.id + '(label: ' + labels(m)[0] + ')' AS Description
UNION
MATCH (n)<-[r]-(m {{id: '{node.id}'}})
RETURN 'Node ' + m.id + '(label: ' + labels(m)[0] + ') has relationship ' + type(r) + ' with Node
' + n.id + '(label: ' + labels(n)[0] + ')' AS Description
"""

result = n4j.query(query=n4j_query)
kg_result.extend(result)

kg_result = [item['Description'] for item in kg_result]

显示来自知识图谱数据库的结果
print(kg_result)

>>>
['Node DataWhale (label: Animal) has relationship LIVINGIN with Node 海洋 (label: Location)', 'Node DataWhale (label: Animal) has relationship DISCOVER with Node 古老羊皮纸 (label: Artifact)', 'Node DataWhale (label: Animal) has relationship TRAVELTO with Node 沙漠 (label: Location)', 'Node DataWhale (label: Animal) has relationship FIND with Node 知识的宝库 (label: Treasure)', 'Node CAMEL (label: Animal) has relationship LIVINGIN with Node 沙漠 (label: Location)', 'Node CAMEL (label: Animal) has relationship DISCOVER with Node 古代石板 (label: Artifact)', 'Node CAMEL (label: Animal) has relationship COOPERATE with Node DataWhale (label: Animal)', 'Node CAMEL (label: Animal) has relationship TRAVELTO with Node 海洋 (label: Location)', 'Node CAMEL (label: Animal) has relationship FIND with Node 知识的宝库 (label: Treasure)', 'Node CAMEL (label: Animal) has relationship SHAREKNOWLEDGE with Node DataWhale (label: Animal)']

这样就可以检索出我们 query 对应的这些信息啦。
合并来自向量搜索和知识图谱实体搜索的结果
comined_results = str(vector_result) + "\n".join(kg_result)

print(comined_results)
>>>
{'Original Query': 'DataWhale 和 CAMEL 之间是什么关系', 'Retrieved Context': ['在一个神奇的动物王国里，住着一只聪明的骆驼，名叫 CAMEL，以及一只智慧的鲸鱼，名叫 DataWhale。CAMEL 生活在辽阔的沙漠中，而 DataWhale 则畅游在无边无际的海洋里。虽然它们的栖息地截然不同，但它们都对探索新知充满了浓厚的兴趣。\\n\\n 有一天，CAMEL 在沙漠中偶然发现了一块古老的石板，上面刻满了奇怪的符号和图案。CAMEL 意识到这可能是一份通往巨大宝藏的地图，但它无法解读这些符号。于是，它决定去寻找帮助。\\n\\n 与此同时，DataWhale 在海洋深处发现了一卷古老的羊皮纸，上面也布满了类似的符号。这让 DataWhale 感到非常好奇，因为它知道这些符号可能蕴藏着某种秘密，但它同样无法破译。']}

```

\n\n 通过一只飞翔在天空中的信使鸟， CAMEL 和 DataWhale 得知彼此的发现，并决定合作  
解开这个谜团。骆驼踏上了漫长的旅程，穿越沙漠来到了海洋边缘，而鲸鱼则游到岸边，与  
CAMEL 会合。']}\nNode DataWhale (label: Animal) has relationship LIVINGIN with Node 海洋  
(label: Location)\nNode DataWhale (label: Animal) has relationship DISCOVER with Node 古老羊皮纸 (label:  
Artifact)\nNode DataWhale (label: Animal) has relationship TRAVELTO with Node 沙漠 (label: Location)\nNode DataWhale (label: Animal) has relationship FIND with Node 知识的宝库 (label:  
Treasure)\nNode CAMEL (label: Animal) has relationship LIVINGIN with Node 沙漠 (label: Location)\nNode CAMEL (label: Animal) has relationship DISCOVER with Node 古代石板 (label: Artifact)\nNode CAMEL (label: Animal) has relationship COOPERATE with Node DataWhale (label:  
Animal)\nNode CAMEL (label: Animal) has relationship TRAVELTO with Node 海洋 (label: Location)\nNode CAMEL (label: Animal) has relationship FIND with Node 知识的宝库 (label: Treasure)\nNode CAMEL (label: Animal) has relationship SHAREKNOWLEDGE with Node DataWhale  
(label: Animal)\n# 设置代理\nsys\_msg = BaseMessage.make\_assistant\_message(\n role\_name="CAMEL Agent",\n content="""您是一个有用的助手来回答问题，  
我将给您原始查询和检索到的上下文，  
根据检索到的上下文回答原始查询。""",\n)\n\ncamel\_agent = ChatAgent(system\_message=sys\_msg,\n model=model)\n\n# 将检索到的信息传递给智能体\nuser\_prompt=f"""\n原始查询是 {query}\n检索到的上下文是 {comined\_results}\n.....\n\nuser\_msg = BaseMessage.make\_user\_message(\n role\_name="CAMEL User", content=user\_prompt\n)\n\n# 获取响应\nagent\_response = camel\_agent.step(user\_msg)\n\nprint(agent\_response.msg.content)\n\n>>>

DataWhale 和 CAMEL 之间的关系是合作和知识共享。尽管它们生活在不同的环境中——CAMEL 生活在沙漠，而 DataWhale 生活在海洋，但它们都发现了神秘的古代文物，分别是古代石板和古老羊皮纸。通过一只信使鸟的帮助，它们了解到对方的发现并决定合作解开这些文物的秘密。CAMEL 跨越沙漠来到海洋边缘，而 DataWhale 也游到岸边与 CAMEL 会合。最终，它们共同努力找到了知识的宝库。

至此恭喜你学会了如何使用基础的向量检索和基于图数据库的检索！下面让我们看看，Graph 还有哪些可以用的地方吧。

## 5. 进阶案例

接下来的案例会演示如何利用设置和利用 CAMEL 的检索增强生成（RAG）与 Firecrawl 相结合，以实现高效的 Web 抓取、多代理角色扮演任务和知识图谱构建。我们将使用 LLM 对 2024 年巴黎奥运会上的土耳其射击运动员进行全面研究。

在本笔记本中，您将探索：

- **CAMEL:** 一个强大的多代理框架，支持 Retrieval-Augmented Generation 和多代理角色扮演场景，允许执行复杂的 AI 驱动任务。
- **Firecrawl:** 一个强大的网络抓取工具，可简化从各种网页中提取和清理内容的过程。
- **AgentOps:** 跟踪和分析 CAMEL Agent 的运行情况。
- **Qdrant:** 一种高效的向量存储系统，与 Camel 的 AutoRetriever 一起使用，根据向量相似性存储和检索相关信息。
- **Neo4j:** 领先的图数据库管理系统，用于构建和存储知识图谱，使实体之间的复杂关系能够高效映射和查询。
- **DuckDuckGo 搜索:** 在 SearchToolkit 中用于从 Web 收集相关 URL 和信息，作为检索初始内容的主要搜索引擎。
- **非结构化 IO:** 用于内容分块，便于管理非结构化数据以实现更高效的处理。

首先我要给所需要的工具设置好 API 密钥

您可以到[这里](#)从 AgentOps 获取免费的 API 密钥

```
import os
from getpass import getpass
```

```
输入 AgentOps API 密钥
```

```
agentops_api_key = getpass('请输入您的 API 密钥: ')
os.environ['AGENTOPS_API_KEY'] = agentops_api_key
```

您可以到[这里](#)从 Firecrawl 获取免费 API 密钥

```
firecrawl_api_key = getpass('Enter your API key: ')
os.environ['FIRECRAWL_API_KEY'] = firecrawl_api_key
```

Firecrawl 是一个强大的工具，可简化网页抓取和清理网页内容的过程。在本节中，我们将从 CAMEL AI 网站上的特定帖子中抓取内容作为示例。

```
from camel.loaders import Firecrawl
```

```
firecrawl = Firecrawl()
```

```
从指定的 URL 抓取并清理内容
```

```
response = firecrawl.scrape(
 url="https://www.camel-ai.org/"
)
```

```
print(response["markdown"])
```

>>>

Aug 9, 2024

```
CRAB: Cross-environment Agent Benchmark for Multimodal Language Model Agents
```

Introducing CRAB: A Benchmark for Cross-Platform Multimodal Agents.

> Abstract: Recently, spearheaded by the CAMEL-AI community, a pioneer in open-source multi-agent projects, researchers from institutions such as King Abdullah University of Science and Technology, Oxford University, University of Tokyo, Carnegie Mellon University, Stanford University, and Tsinghua University have developed a cross-platform multimodal agent benchmark framework: CRAB, innovatively enabling agents to operate multiple devices simultaneously.

### ### Introduction

With the rapid development of multimodal large language models (MLLM), many agents capable of operating graphical user interfaces (GUIs) have emerged this year. Various companies have launched their innovative solutions, creating intense competition. GUI agents, leveraging powerful visual understanding and reasoning abilities of large models, can now efficiently and flexibly complete tasks such as booking appointments, shopping, and controlling smart homes.

\*\*This raises the question: will future agents truly be able to sit in front of a computer and work on my behalf?\*\*

However, in today's era of the Internet of Everything, most work requires the coordination of multiple devices. For example, taking a photo with a phone and then transferring it to a computer for editing involves crossing two different devices (environments). Currently, these GUI agents can only operate on a single device, making what is an easy task for humans exceedingly difficult for today's agents.

Researchers from the CAMEL-AI community noticed this problem and proposed the first cross-environment, multi-device agent benchmark framework—CRAB, the \*\*CR\*\* oss-environment \*\*A\*\*gent \*\*B\*\*enchmark.

)

Paper link: [<https://arxiv.org/abs/2407.01511>](https://arxiv.org/abs/2407.01511)

The CAMEL framework ([\[https://github.com/camel-ai\]](https://github.com/camel-ai)(https://github.com/camel-ai)) developed by the CAMEL-AI community is one of the earliest open-source multi-agent projects based on large language models. Therefore, community members are researchers and engineers with rich research and practical experience in the field of agents.

In CRAB, the authors not only designed a network-based multi-environment architecture that enables agents to operate multiple devices simultaneously to complete tasks, but also proposed two new technologies to address the issues existing in current agent benchmarks: the graph evaluator and task synthesis. CRAB is not only a brand new benchmark tool but also provides an interaction protocol and its implementation between the environment and agents, which is expected to become an important foundation for agents in practical fields.

...

Copyright © 2024 Eigent AI - All Rights Reserved.

Schedule a chat

*Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...*

-  使用 CAMEL 的 RAG 和 Firecrawl 进行 Web 信息检索

在本节中，我们将演示如何使用 Camel 的 RAG 模型从 URL 列表中检索相关信息。这对于聚合和分析来自多个来源的数据特别有用。

首先我们定义一个检索方法：

```
from camel.retrievers import AutoRetriever
from camel.toolkits import FunctionTool, SearchToolkit
from camel.types import ModelPlatformType, ModelType, StorageType
from camel.embeddings import SentenceTransformerEncoder
```

```
def retrieve_information_from_urls(urls: list[str], query: str) -> str:
```

```
 r"""根据给定的查询从 URL 列表中检索相关信息。
```

此函数使用 `Firecrawl` 工具从提供的 URL 抓取内容，然后使用 CAMEL 的 `AutoRetriever` 根据查询从抓取的内容中检索最相关的信息。

参数:

urls (list[str]): 要抓取内容的 URL 列表。

query (str): 用于搜索相关信息的查询字符串。

返回:

str: 根据查询检索到的最相关信息。

示例:

```
>>> urls = ["https://example.com/article1", "https://example.com/
article2"]
```

```
>>> query = "最新的人工智能进展"
>>> result = retrieve_information_from_urls(urls, query)
.....
aggregated_content = ""

从每个 URL 抓取并聚合内容
for url in urls:
 scraped_content = Firecrawl().scrape(url)
 aggregated_content += scraped_content["markdown"]

设置一个带有本地存储和 Mistral AI 嵌入模型的向量检索器
auto_retriever = AutoRetriever(
 vector_storage_local_path="local_data",
 storage_type=StorageType.QDRANT,
 embedding_model=SentenceTransformerEncoder(),
)

根据查询检索最相关的信息
您可以根据需要调整 top_k 和 similarity_threshold 的值
retrieved_info = auto_retriever.run_vector_retriever(
 query=query,
 contents=aggregated_content,
 top_k=3,
 similarity_threshold=0.5,
)

return retrieved_info

让我们通过收集有关 2024 年奥运会的一些信息来测试检索功能。第一次运行可能需要大约 50 秒，因为它需要构建本地向量数据库。
retrieved_info = retrieve_information_from_urls(
 query="哪个国家在 2024 年奥运会上获得了最多的金牌？",
 urls=[
 "https://www.nbcnews.com/sports/olympics/united-states-china-gold-medals-rcna166013",
],
)

print(retrieved_info)

>>>
{'Original Query': '哪个国家在 2024 年奥运会上获得了最多的金牌？', 'Retrieved Context': '[在 2024 年巴黎奥运会上，中国创下了有史以来最佳表现，与美国并列获得最多金牌。两国各获得 40 枚金牌，这是夏季奥运会上金牌数量首次出现平局。尽管如此，美国在奖牌总数上仍领先于中国，总共获得 126 枚奖牌，而中国获得 91 枚。\\n\\n[中'}
```

国 ](<https://sigma.world/zh-hant/news/remarkable-growth-china-lotter-industry-expected/>) 在射击和跳水等项目上的强势表现最初使其领先于美国。然而，美国队在田径赛事中后来居上，最终在奖牌总数上超过中国。中国媒体和公众对这一卓越表现表示庆祝，认为这是国家日益强大和现代化的体现。  
YouTube\n\nOlympics\n\n14.6M subscribers\n\n[Memories of #Paris2024](<https://www.youtube.com/watch?v=6c9hFGtSUF8>)\n\nOlympics\n\nSearch\n\nInfo\n\nShopping\n\nTap to unmute', '[Watch on YouTube](<https://www.youtube.com/watch?v=6c9hFGtSUF8> "Watch on YouTube")\n\n\*\*\_巴黎奥运会 2024 留念\_\*\n\n## 日本在巴黎奥运会上创下金牌新纪录\n日本在巴黎也有出色的表现，赢得 20 枚金牌——这是日本在海外奥运会上获得的最多金牌。这一成就超过了日本在 2004 年雅典奥运会上获得的 16 枚海外金牌的纪录，也是日本继 2021 年东京奥运会获得 27 枚金牌之后的第二高金牌总数。  
[日本] (<https://sigma.world/zh-hant/news/japan-sports-wagering-reaches-new-heights/>) 的成功以摔跤等关键体育项目的精彩表现为亮点。木太郎在男子 65 公斤自由式摔跤中的胜利以及加上加贺美在女子 76 公斤自由式摔跤中的金牌是日本战役中的亮点之一，日本总共获得了 45 枚奖牌。  
## 韩国超出预期，取得了大量奖牌'\n[韩国] (<https://sigma.world/zh-hant/news/paradise-co-profit-in1q-2024-up/>) 以 32 枚奖牌，包括 13 枚金牌，令人惊讶。这是该国 12 年来首次在单届奥运会上超过 30 枚奖牌。韩国队在射击、击剑和射箭项目中的出色表现推动了成功。  
## 值得注意的是，韩国射箭选手在所有五个射箭项目中获得金牌，金优真成为韩国获得金牌最多的奥运选手，职业生涯获得五枚金牌。年轻运动员的成功，包括 16 岁的射击选手班孝珍，他赢得了韩国第 100 枚夏季奥运会金牌，是该国取得优异成绩的关键因素。  
## 菲律宾成为东南亚表现最佳的国家'}

- **使用 AgentOps 监控 AI 代理**

```
import agentops
agentops.init(default_tags=["CAMEL cookbook"])
CAMEL 的一个强大功能是它能够从文本数据构建和存储知识图谱。这很方便我们对数据中的关系进行高级分析和可视化。
```

首先我们定义一个函数用于处理输入文本以创建和提取节点和关系，并将其作为知识图谱添加到 Neo4j 数据库中。

```
from camel.storages import Neo4jGraph
from camel.loaders import UnstructuredIO
from camel.agents import KnowledgeGraphAgent
```

```
def knowledge_graph_builder(text_input: str) -> None:
 r"""从提供的文本构建并存储知识图谱。
```

此函数处理输入文本以创建和提取节点和关系，然后将其作为知识图谱添加到 Neo4j 数据库中。

参数：

text\_input (str): 用于构建知识图谱的输入文本。

返回：

graph\_elements: 知识图谱代理生成的图形元素。

.....

```
设置 Neo4j 实例
n4j = Neo4jGraph(
 url="Your_URL",
 username="Your_Username",
 password="Your_Password",
)

初始化实例
uio = UnstructuredIO()
kg_agent = KnowledgeGraphAgent(mode=model)

从提供的文本创建一个元素
element_example = uio.create_element_from_text(text=text_input, element_id="001")

使用知识图谱代理提取节点和关系
graph_elements = kg_agent.run(element_example, parse_graph_elements=True)

将提取的图形元素添加到 Neo4j 数据库
n4j.add_graph_elements(graph_elements=[graph_elements])

return graph_elements
```

然后我们设置我们的 RolePlaying, AI 代理在其中交互以使用各种工具完成任务。我们将指导助理代理对 2024 年巴黎奥运会的土耳其射击运动员进行全面研究。



导入模块及定义我们的任务：

```
from typing import List
```

```
from colorama import Fore
```

```
from camel.societies import RolePlaying
from camel.utils import print_text_animated
```

task\_prompt = """对 2024 年巴黎奥运会的土耳其射手进行全面研究，为我写一份报告，然后为报告创建知识图谱。

您应该首先使用搜索工具获取相关的 URL，然后使用检索工具通过提供 URL 列表来获取检索到的内容，最后使用工具构建知识图谱以完成任务。

不要进行其他操作。"""

我们将为助理代理配置用于数学计算、Web 信息检索和知识图谱构建的工具。

```
retrieval_tool = FunctionTool(retrieve_information_from_urls)
search_tool = FunctionTool(SearchToolkit().search_duckduckgo)
knowledge_graph_tool = FunctionTool(knowledge_graph_builder)
```

```
tool_list = [
```

```
retrieval_tool,
search_tool,
knowledge_graph_tool,
]

assistant_model_config = dict(
 tools=tool_list,
 temperature=0.0,
)

初始化 RolePlaying
role_play_session = RolePlaying(
 assistant_role_name="CAMEL Assistant",
 user_role_name="CAMEL User",
 assistant_agent_kwargs=dict(
 model=model,
 tools=tool_list,
),
 user_agent_kwargs=dict(model=model),
 task_prompt=task_prompt,
 with_task_specify=False,
 output_language="中文"
)

print(
 Fore.GREEN
 + f"AI Assistant sys message:\n{role_play_session.assistant_sys_msg}\n"
)
print(Fore.BLUE + f"AI User sys message:\n{role_play_session.user_sys_msg}\n")

print(Fore.YELLOW + f"Original task prompt:\n{task_prompt}\n")
print(
 Fore.CYAN
 + "Specified task prompt:"
 + f"\n{role_play_session.specified_task_prompt}\n"
)
print(Fore.RED + f"Final task prompt:\n{role_play_session.task_prompt}\n")
开始我们的 Agent 交互。
注意：此会话大约需要 8 分钟。
n = 0
input_msg = role_play_session.init_chat()
while n < 20: # Limit the chat to 20 turns
 n += 1
 assistant_response, user_response = role_play_session.step(input_msg)
```

```

if assistant_response.terminated:
 print(
 Fore.GREEN
 + (
 "AI Assistant terminated. Reason: "
 f"[assistant_response.info['termination_reasons']]"
)
)
 break
if user_response.terminated:
 print(
 Fore.GREEN
 + (
 "AI User terminated. "
 f"Reason: {user_response.info['termination_reasons']}"
)
)
 break
Print output from the user
print_text_animated(
 Fore.BLUE + f"AI User:\n\n{user_response.msg.content}\n",
 0.01
)
CAMEL_TASK_DONE in user_response.msg.content:
 break

Print output from the assistant, including any function
execution information
print_text_animated(Fore.GREEN + "AI Assistant:", 0.01)
tool_calls = [
 for call in assistant_response.info['tool_calls']
]
for func_record in tool_calls:
 print_text_animated(f"{func_record}", 0.01)
print_text_animated(f"{assistant_response.msg.content}\n", 0.01)

input_msg = assistant_response.msg

```

---

## 4.7 第四章课程作业

作业 1：通过调用 API 或本地部署 reranker 模型来快速将 rerank 模块集成到 RAG 应用中  
[https://github.com/camel-ai/camel/blob/master/camel/retrievers/cohere\\_rerank\\_retriever.py](https://github.com/camel-ai/camel/blob/master/camel/retrievers/cohere_rerank_retriever.py)

作业 2：使用大语言模型和简单的提示词工程在 CAMEL 框架下实现一个 ReRanker，达成重排的效果（灵感来自 CAMEL Shanghai Hackathon 韬子哥 QAQ）

[https://github.com/fengju0213/NavigatorAI/blob/main/2Travel\\_information\\_generation.py](https://github.com/fengju0213/NavigatorAI/blob/main/2Travel_information_generation.py)

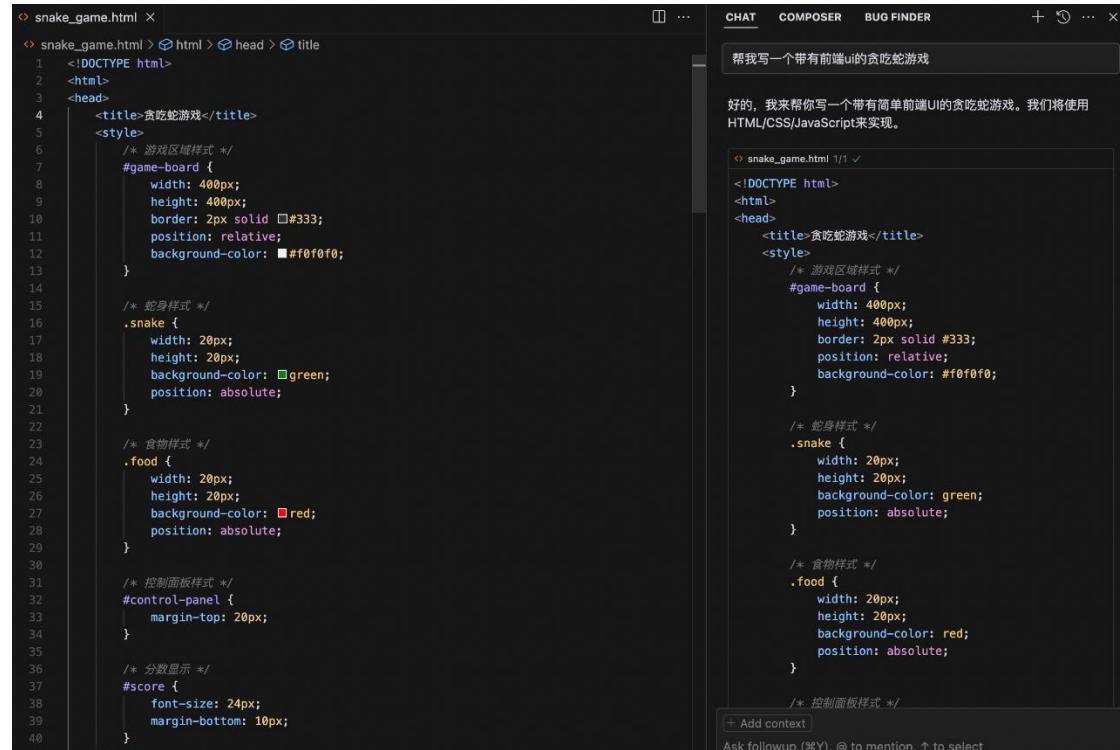
作业 3：导出本教程文档作为一个 PDF 文件，尝试搭建一个具有多模态检索能力的 RAG 应用来帮助其他学习者回答问题。

## 5. 第五章 综合案例

在这一节中，我们将展示如何使用 CAMEL 框架来搭建一个简单实用的旅游出行规划助手。

### 1. 应用概览

在大模型时代，代码能力基本已经不再是初学者想要开发 AI 应用的障碍，单纯地使用一些模型的 Web 端就可以搭建一些简单的应用，搭配一些 AI 编程工具可以让我们的开发效率更高，只需要写一些提示词就可以搭建一个简单的带有前端 UI 的应用。



The screenshot shows the CAMEL IDE interface. On the left is a code editor with the file 'snake\_game.html' open. The code is an HTML/CSS snippet for a snake game. On the right is a chat window with the following interaction:

```
帮我写一个带有前端ui的贪吃蛇游戏
好的，我来帮你写一个带有简单前端UI的贪吃蛇游戏。我们将使用HTML/CSS/JavaScript来实现。
+ Add context
Ask followup (⌘Y), @to mention, ↑ to select
```

```
<!DOCTYPE html>
<html>
<head>
<title>贪吃蛇游戏</title>
<style>
/* 游戏区域样式 */
#game-board {
 width: 400px;
 height: 400px;
 border: 2px solid #333;
 position: relative;
 background-color: #f0f0f0;
}

/* 蛇身样式 */
.snake {
 width: 20px;
 height: 20px;
 background-color: green;
 position: absolute;
}

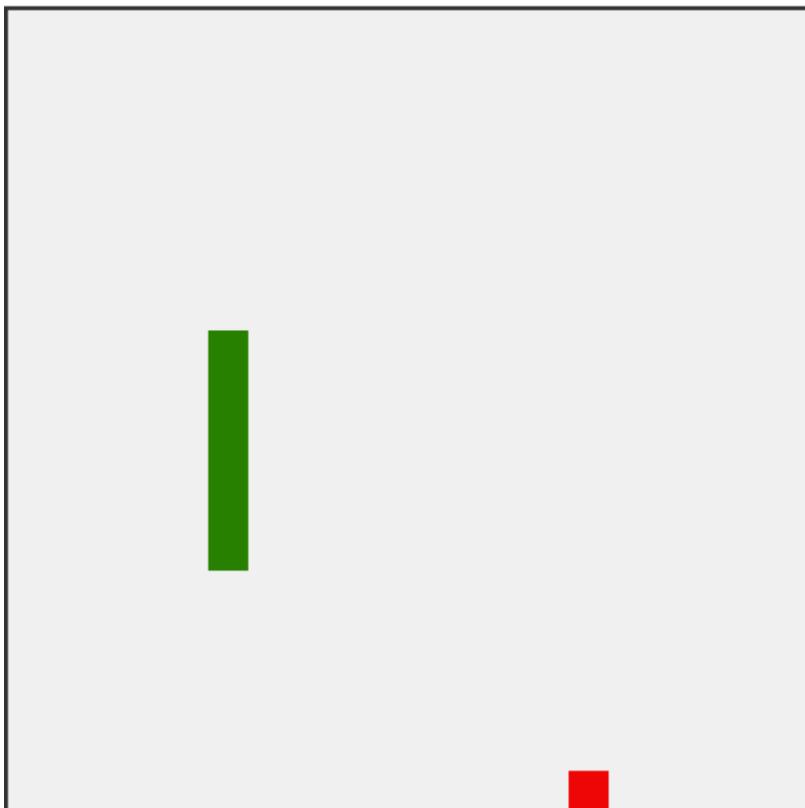
/* 食物样式 */
.food {
 width: 20px;
 height: 20px;
 background-color: red;
 position: absolute;
}

/* 控制面板样式 */
#control-panel {
 margin-top: 20px;
}

/* 分数显示 */
#score {
 font-size: 24px;
 margin-bottom: 10px;
}
```

# 贪吃蛇游戏

得分: 5



**开始游戏**

**暂停**

AI 写的代码直接复制就能跑！

得益于现在大模型的强大代码能力，我们可以将更多的精力放在我们的应用架构以及产品上，包括我们要实现的功能呀、架构设计、目标用户等等。能不能捕捉到用户的需求很重要，以什么形式满足？也就是说，现在的 AI 时代，产品思维对于应用开发者也是很重要的。

以下是我们使用 CAMEL 框架搭建一个真实可用的软件应用——NavigatorAI（旅游出行规划助手），其中使用到了非常多常见而实用的 CAMEL 模块和知识点。

**特别注意：**NavigatorAI 遵循 [CC BY-NC 协议](#)，仅供学习使用，不能商用!!!

## 1. 目标和交互形式

首先明确我们的目标，项目的核心目标是实现一个智能化的旅游出行规划助手：用户只需输入目的地和天数，系统即可自动生成详细的旅游行程，包括每日活动安排、餐饮推荐、景点信息等。系统还支持根据用户反馈动态调整行程，支持在线浏览和编辑，同时提供多种格式（Markdown, HTML 或 PDF）的导出，方便用户保存或分享。这不仅提升了行程规划的效率，也能为用户带来更为便捷和专业的旅游体验。

根据目标我们可以设计出我们的整体应用架构图如下：

暂时无法在飞书文档外展示此内容

## 2. 模块化设计

在现代软件开发中，一般会选择模块化的设计思路，旨在通过将系统分解为多个独立、可重用的模块来提升代码的可维护性、可扩展性和开发效率。模块化设计的核心思想是解耦，即通过清晰的边界和接口将系统的不同功能分离，使得每个模块可以独立开发、测试和维护，而不会对其他模块产生过多依赖。

因此通过模块化设计，开发团队可以更高效地构建复杂系统，同时降低长期维护的难度，可以帮助开发团队在短时间协同完成整个项目的开发设计和测试。

旅游出行规划助手 NavigatorAI 也采取了模块化的设计思想，整个综合应用可以分成四个子模块——

### 1. 信息收集模块

在该模块中，用户和信息收集 Agent 进行多轮往复的对话，直到 Agent 认为能够通过用户的表达和上下文提取出用户的目标出行城市和出行天数等关键信息，并将信息传递到下一个模块。

### 2. 攻略生成模块

- 验证数据是否存在：

- 如果数据库中存在对应的城市和天数的旅游数据，(即缓存命中)，则直接加载复用数据。
- 如果数据库中没有查询到对应的信息，则调用 Tool Calling 中的搜索工具 (Google Search 或 DuckDuckGo) 进行实时的信息检索 (检索的内容包括著名景点，当地美食，天气情况等数据，可以根据需要自由添加)

- 攻略初稿生成：

- 接下来，攻略生成 Agent 会结合上述检索到的基础数据，按预先定义的默认格式生成 HTML 格式的详细行程攻略，包含每日的活动安排和推荐内容。

- 格式转换：

- 将生成的 HTML 导出为 PDF 格式，并在前端提供在线预览，方便用户进行下载/转发/保存/修改。

### 3. 反馈优化模块

- 收集用户反馈：

- 用户可以在前端预览中对生成的攻略初稿中的任意部分和内容进行评价，支持手动标记圈画需要优化的部分，这意味着我们接受反馈的形式是多模态的 (图像理解+文字评述)。

- 使用反馈优化 Agent 进行意图理解并迭代式地调整攻略：

- 反馈优化 Agent 分析用户反馈并理解具体修改需求；
- 基于用户反馈，结合多模态能力和工具调用生成新内容并迭代式优化。

- 输出最终攻略版本：

- 根据用户确认的最终行程生成 PDF 文档。
- 将优化后的行程存入数据库以供未来复用。

## 2. 用户意图识别模块

该模块主要用于收集用户想要去哪，去几天的需求。我们可以通过一个合适的 system prompt 来实现，为了后续在软件开发中的前后端分离的设计，我们还需要进一步通过 flask 将该模块封装成一个本地服务——

```
import os
import sys
import json
```

```
from typing import Optional
from flask import Flask, request, jsonify, Response

from dotenv import load_dotenv
from camel.configs import QwenConfig
from camel.models import ModelFactory
from camel.types import ModelPlatformType
from camel.agents import ChatAgent

load_dotenv()

API_KEY = os.getenv('MODELSCOPE_SDK_TOKEN')

SYSTEM_PROMPT = """

你是一个旅游信息提取助手。你的任务是从用户的输入中提取旅游目的地城市和行程天数，并根据提取情况决定是否需要用户补充信息。

"""


```

用户输入可能包含以下信息：

- \* 旅游目的地城市名称（例如：北京、上海、巴黎、东京）
- \* 行程天数（例如：3 天、5 天、一周、两周）
- \* 可能会有其他无关信息，请忽略。

你需要将提取到的城市名称和行程天数以 JSON 格式返回，格式如下：

- ```
{"city": "城市名称", "days": 天数, "need_more_info": boolean}
```
- * "city" 的值：
 - * 如果成功提取到城市名称，则为城市名称字符串。
 - * 如果无法提取到城市名称，则为 null。
 - * "days" 的值：
 - * 如果成功提取到行程天数，则为数字。
 - * 如果无法提取到行程天数，则为 null。
 - * "need_more_info" 的值：
 - * 如果 "city" 或 "days" 中有任何一个为 null，则为 true，表示需要用户提供更多信息。
 - * 如果 "city" 和 "days" 都不为 null，则为 false，表示不需要用户提供更多信息。
 - * 如果提取到的天数包含“天”或“日”等字样，请将其转换为数字。
 - * 如果提取到的天数包含“周”或“星期”，请将其转换为 7 的倍数。例如，“一周”转换为 7，“两周”转换为 14。
 - * 如果用户输入中包含多个城市，请只提取第一个城市。
 - * 如果用户输入中包含多个天数，请只提取第一个天数。

请严格按照 JSON 格式返回结果。

示例：

用户输入：

我想去北京玩三天，顺便看看长城。

你的输出：

{"city": "北京", "days": 3, "need_more_info": false, "response": "信息在 Navigator 的数据库中查询到啦，正在努力为您生成攻略~"}

用户输入：

我想去北京。

你的输出：

{"city": "北京", "days": null, "need_more_info": true, "response": "Navigator 还不知道您打算去玩几天呢，请补充你计划的行程天数~"}

.....

```
app = Flask(__name__)
```

```
def create_travel_agent():
```

```
    qwen_model = ModelFactory.create(  
        model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,  
        model_type="Qwen/Qwen2.5-72B-Instruct",  
        api_key=API_KEY,  
        url="https://api-inference.modelscale.cn/v1",  
        model_config_dict=QwenConfig(temperature=0.2).as_dict(),  
    )
```

```
    agent = ChatAgent(  
        system_message=SYSTEM_PROMPT,
```

```
        model=qwen_model,  
        message_window_size=10,  
        output_language='Chinese'
```

```
)
```

```
    return agent
```

```
travel_agent = create_travel_agent()
```

```
def get_travel_info_camel(user_input: str, agent: ChatAgent) -> dict:
```

```
    try:
```

```
        response = agent.step(user_input)
```

```
        # 回到原始状态
```

```
        agent.reset()
```

```
        if not response or not response.msgs:
```

```
            raise ValueError("模型没有返回任何消息")
```

```
        json_output = response.msgs[0].content.strip().replace("```json", "").replace("```", "")
```

```

        json_output = json.loads(json_output)
        json_output["query"] = user_input
        return json_output
    except json.JSONDecodeError:
        print("Error: 模型返回的不是有效的 JSON 格式。")
        return {
            'city': None,
            'days': None,
            'need_more_info': True,
            'query': user_input,
            'response': None
        }
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return {
            'city': None,
            'days': None,
            'need_more_info': True,
            'query': user_input,
            'response': None
        }
}

@app.route('/extract_travel_info', methods=['POST'])
def extract_travel_info():
    try:
        request_data = request.get_json()
        if not request_data or 'query' not in request_data:
            return jsonify({'error': '请求数据无效'}), 400

        result = get_travel_info_camel(request_data['query'], travel_agent)
        response = {
            'city': result['city'],
            'days': result['days'],
            'need_more_info': result['need_more_info'],
            'query': result['query'],
            'response': result['response']
        }
        response_json = json.dumps(response, ensure_ascii=False)
        return Response(response_json, status=200, mimetype='application/json; charset=utf-8')
    except Exception as e:
        return jsonify({'error': f'服务器内部错误: {str(e)}'}), 500

if __name__ == "__main__":

```

```
app.run(host="0.0.0.0", port=5001)
```

这个模块运行之后会在本地 5001 端口启动一个服务，我们可以通过请求这个服务来和模块中的**信息收集 Agent** 进行多轮交互

使用 Python requests 库的示例代码

```
import requests
```

```
url = "http://localhost:8000/extract_travel_info"
data = {"query": "我想去北京玩三天"}
response = requests.post(url, json=data)
print(response.json())
```

```
>>>
```

```
{"city": "北京", "days": 3, "need_more_info": False, "query": "我想去北京玩三天", "response": "信息已经获取，正在帮你生成攻略。"}
```

使用 curl 的示例命令行，这次我们不说清楚需求试试：

```
curl -X POST http://localhost:8000/extract_travel_info \
-H "Content-Type: application/json" \
-d '{"query": "想去北京玩"}'
```

```
>>>
```

```
{"city": "北京", "days": null, "need_more_info": true, "query": "想去北京玩", "response": "请补充旅游天数，你打算去玩几天呀。"}
```

通过上述两个测试的示例可以观察到**信息收集模块**的核心功能会让我们在信息表述不全的情况下自主的分析问题并引导提示我们补充行程天数，有的小伙伴可能会觉得，根本没必要这样做呀，直接在开始界面设置弹窗让用户输入城市和天数不就行了，其实这个方案我们也考虑过，这样确实更方便快捷，而我们在方案中特意添加了这个模块是为了考虑到后续可能有更多的交互空间的拓展，比如情感陪伴等。当然，适合自己的方案就是最好的！Enjoy!

3. 旅游信息检索

旅游信息检索是系统中实现数据获取和处理的关键环节，负责根据用户输入的目的地城市和出游天数，动态获取并生成高质量的旅游数据。

模块的工作流程分为以下几个阶段：首先，对用户输入的信息进行标准化处理，将城市名称和时间信息改写为适合搜索引擎的查询模板（query）。随后，系统调用 **Google Search API** 进行景点和美食信息的文本检索，包括景点描述、距离、推荐美食等详细内容；同时，为了补充图片资源，模块还调用 **DuckDuckGo 搜索引擎**，专注于获取高质量的景点和美食图片链接。

在数据检索完成后，模块进一步利用大语言模型（LLM）对初步搜索结果进行智能解析与重排序（rerank），从相关性和用户需求角度优化数据质量，确保信息全面、准确、优先级清晰。经过优化后的数据会以结构化的形式存储到数据库中，包含每个景点和美食的名称、详细描述、推荐理由以及图片 URL。

暂时无法在飞书文档外展示此内容

```
from camel.toolkits import SearchToolkit
from camel.agents import ChatAgent
```

```
from camel.models import ModelFactory
from camel.types import ModelPlatformType
from camel.loaders import Firecrawl
from typing import List, Dict, Any

from flask import Flask, request, jsonify
import json
import os
from dotenv import load_dotenv

load_dotenv()

os.environ["GOOGLE_API_KEY"] = os.getenv("GOOGLE_API_KEY")
os.environ["SEARCH_ENGINE_ID"] = os.getenv("SEARCH_ENGINE_ID")
os.environ["FIRECRAWL_API_KEY"] = os.getenv("FIRECRAWL_API_KEY")
os.environ["MODELSCOPE_SDK_TOKEN"] = os.getenv("MODELSCOPE_SDK_TOKEN")

app = Flask(__name__)

class TravelPlanner:
    def __init__(self, city: str, days: int):

        # 定义地点和时间，设置默认值
        self.city = city
        self.days = days
        self.res = None

        # 初始化模型和智能体
        self.model = ModelFactory.create(
            model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
            model_type="Qwen/Qwen2.5-72B-Instruct",
            url='https://api-inference.modelscope.cn/v1/',
            api_key=os.getenv('MODELSCOPE_SDK_TOKEN')
        )

        # 初始化各种工具
        # 重排序模型
        self.reranker_agent = ChatAgent(
            system_message="你是一搜索质量打分专家，要从{搜索结果}里找出和{query}里最相关的 2 条结果，保存他们的结果，保留 result_id、title、description、url，严格以 json 格式输出",
            model=self.model,
            output_language='中文'
        )
```

```
#景点抓取 agent
self.attraction_agent = ChatAgent(
    system_message="你是一个旅游信息提取专家，要根据内容提取出景点信息
并返回 json 格式，严格以 json 格式输出",
    model=self.model,
    output_language='中文'
)
#美食抓取 agent
self.food_agent = ChatAgent(
    system_message="你是一个旅游信息提取专家，要根据内容提取出美食信息
并返回 json 格式，严格以 json 格式输出",
    model=self.model,
    output_language='中文'
)
#base 攻略生成 agent
self.base_guide_agent = ChatAgent(
    system_message="你是一个旅游攻略生成专家，要根据内容生成一个旅游攻
略，严格以 json 格式输出",
    model=self.model,
    output_language='中文'
)
# self.firecrawl = Firecrawl()#后续功能
self.search_toolkit = SearchToolkit()

def extract_json_from_response(self,response_content: str) -> List[Dict[str, Any]]:
    """从 LLM 响应中提取 JSON 内容"""
    try:
        # 找到 JSON 内容的开始和结束位置
        start = response_content.find('```json\n') + 8
        end = response_content.find('\n```', start)
        if start == -1 or end == -1:
            print("未找到 JSON 内容的标记")
            return []
        json_str = response_content[start:end].strip()
        print(f"提取的 JSON 字符串: {json_str}") # 调试信息

        # 解析 JSON 字符串
        parsed = json.loads(json_str)

        # 处理不同的 JSON 结构
        if isinstance(parsed, dict) and "related_results" in parsed:
            return parsed["related_results"]
        elif isinstance(parsed, list):
            return parsed
    except json.JSONDecodeError:
        print("无法解析 JSON 字符串")
```

```

        return parsed
    else:
        print("未找到预期的 JSON 结构")
        return []

except json.JSONDecodeError as e:
    print(f"解析 JSON 失败: {str(e)}")
    print(f"原始内容: {response_content}")
    return []

except Exception as e:
    print(f"发生错误: {str(e)}")
    return []

def search_and_rerank(self) -> Dict[str, Any]:
    """多次搜索并重排序，整合信息"""
    city = self.city
    days = self.days
    all_results = {}

    # 第一次搜索：旅游攻略
    try:
        query = f"{city}{days}天旅游攻略 最佳路线"
        search_results = self.search_toolkit.search_google(query=query,
num_result_pages=5)
        prompt = f"请从以下搜索结果中筛选出最相关的{self.days}条{city}{days}天旅游攻略信息，并按照相关性排序：\n{json.dumps(search_results, ensure_ascii=False, indent=2)}"
        response = self.reranker_agent.step(prompt)
        all_results["guides"] = response
        self.extract_json_from_response(response.msgs[0].content)
    except Exception as e:
        print(f"旅游攻略搜索失败: {str(e)}")
        all_results["guides"] = []

    # 第二次搜索：必去景点
    try:
        query = f"{city} 必去景点 top10 著名景点"
        search_results = self.search_toolkit.search_google(query=query,
num_result_pages=5)
        prompt = f"请从以下搜索结果中筛选出最多{self.days}条{city}最值得去的景点信息，并按照热门程度排序：\n{json.dumps(search_results, ensure_ascii=False, indent=2)}"
        response = self.reranker_agent.step(prompt)
        all_results["attractions"] = response
        self.extract_json_from_response(response.msgs[0].content)
    except Exception as e:
        print(f"必去景点搜索失败: {str(e)}")
        all_results["attractions"] = []

```

```
except Exception as e:  
    print(f"景点搜索失败: {str(e)}")  
    all_results["attractions"] = []  
  
    # 第三次搜索: 必吃美食  
    try:  
        query = f"{city} 必吃美食 特色小吃 推荐"  
        search_results      = self.search_toolkit.search_google(query=query,  
num_result_pages=5)  
        prompt = f"请从以下搜索结果中筛选出最多{self.days}条{city}最具特色的美食  
信息, 并按照推荐度排序: \n{json.dumps(search_results, ensure_ascii=False, indent=2)}"  
        response = self.reranker_agent.step(prompt)  
        all_results["must_eat"] = []  
        self.extract_json_from_response(response.msgs[0].content)  
    except Exception as e:  
        print(f"必吃美食搜索失败: {str(e)}")  
        all_results["must_eat"] = []  
  
    # 第四次搜索: 特色美食  
    try:  
        query = f"{city} 特色美食 地方小吃 传统美食"  
        search_results      = self.search_toolkit.search_google(query=query,  
num_result_pages=5)  
        prompt = f"请从以下搜索结果中筛选出最多{self.days}条{city}独特的地方特色  
美食信息, 并按照特色程度排序: \n{json.dumps(search_results, ensure_ascii=False,  
indent=2)}"  
        response = self.reranker_agent.step(prompt)  
        all_results["local_food"] = []  
        self.extract_json_from_response(response.msgs[0].content)  
    except Exception as e:  
        print(f"特色美食搜索失败: {str(e)}")  
        all_results["local_food"] = []  
  
    # 整合所有信息  
final_result = {  
    "city": city,  
    "days": days,  
    "travel_info": {  
        "guides": [  
            {  
                "result_id": item.get("result_id"),  
                "title": item.get("title"),  
                "description": item.get("description"),  
                "long_description": item.get("long_description"),  
            }  
        ]  
    }  
}
```

```

        }
        for item in all_results["guides"]
    ],
    "attractions": [
        {
            "result_id": item.get("result_id"),
            "title": item.get("title"),
            "description": item.get("description"),
            "long_description": item.get("long_description"),
        }
        for item in all_results["attractions"]
    ],
    "must_eat": [
        {
            "result_id": item.get("result_id"),
            "title": item.get("title"),
            "description": item.get("description"),
            "long_description": item.get("long_description"),
        }
        for item in all_results["must_eat"]
    ],
    "local_food": [
        {
            "result_id": item.get("result_id"),
            "title": item.get("title"),
            "description": item.get("description"),
            "long_description": item.get("long_description"),
        }
        for item in all_results["local_food"]
    ]
]
}

return final_result

```

```

def extract_attractions_and_food(self) -> Dict:
    travel_info = self.search_and_rerank()

    # 提供一个 base 攻略路线，直接根据整个 travel_info 生成
    prompt = f"""
参考以下信息，生成一个{self.city}{self.days}天攻略路线，直接根据整个 travel_info
生成
{travel_info}
【输出格式】

```

```
{{
    "base_guide": "攻略内容"
}}
.....
base_guide = self.base_guide_agent.step(prompt)
print(f"这是 base 攻略: {base_guide.msgs[0].content}")

"""提取景点和美食信息"""
# 从描述中提取具体的景点和美食
attractions_text = " ".join([item["description"] for item in
travel_info["travel_info"]["attractions"] + travel_info["travel_info"]["guides"]])
print(f"这是景点信息: {attractions_text}")
food_text = " ".join([
    item["description"]
    for item in travel_info["travel_info"]["must_eat"] +
travel_info["travel_info"]["local_food"]
])
print(f"这是美食信息: {food_text}")
# 使用 LLM 提取并整理信息
attractions_prompt = f"""
请从以下文本中提取出具体的景点名称，注意不能遗漏景点信息，要尽量多提取景点信息，并为每个景点提供简短描述:
{attractions_text}
请以 JSON 格式返回，格式如下:
{{
    "attractions": [
        {"name": "景点名称", "description": "简短描述"}
    ]
}}
.....
"""

food_prompt = f"""
请从以下文本中提取出具体的美食名称或者美食店铺，注意不能遗漏美食信息，要尽量多提取美食信息，并为每个美食和店铺提供简短描述:
{food_text}
请以 JSON 格式返回，格式如下:
{{
    "foods": [
        {"name": "美食名称", "description": "简短描述"}
    ],
    "food_shop": [
        {"name": "美食店铺", "description": "简短描述"}
    ]
}}
"""
```

```
....  
  
    # 使用 attraction_agent 处理提取  
    attractions_response = self.attraction_agent.step(attractions_prompt)  
    foods_response = self.food_agent.step(food_prompt)  
  
    print(f"这是景点信息: {attractions_response.msgs[0].content}")  
    print(f"这是美食信息: {foods_response.msgs[0].content}")  
  
    return {  
        "base_guide": base_guide.msgs[0].content,  
        "attractions": attractions_response.msgs[0].content,  
        "foods": foods_response.msgs[0].content  
    }  
  
def process_attractions_and_food(self) -> Dict:  
    def clean_json_string(json_str: str) -> str:  
        """清理 JSON 字符串, 移除 markdown 代码块标记"""  
        # 移除 ``json 开头  
        if ````json' in json_str:  
            json_str = json_str.split(```json')[-1]  
        # 移除 ``` 结尾  
        if ````` in json_str:  
            json_str = json_str.split(``````)[0]  
        return json_str.strip()  
  
    city = self.city  
    """处理景点和美食信息, 添加图片 URL"""  
    # 获取原始数据  
    results = self.extract_attractions_and_food()  
  
    # 解析 JSON 字符串  
    base_guide = json.loads(clean_json_string(results['base_guide']))  
    attractions_data = json.loads(clean_json_string(results['attractions']))  
    foods_data = json.loads(clean_json_string(results['foods']))  
    foods_list = foods_data['foods']  
    food_shops_list = foods_data['food_shop']  
  
    # 创建结果字典  
    result = {  
        "city": city,  
        "days": self.days,  
        "base 路线": base_guide,  
        "景点": [],
```

```
"美食": [],
"美食店铺": []
}

# 处理景点信息
for attraction in attractions_data['attractions']:
    try:
        # 使用 DuckDuckGo 搜索图片
        images = self.search_toolkit.search_duckduckgo(
            query=f'{city} {attraction["name"]} 实景图',
            source="images",
            max_results=1
        )

        # 添加图片 URL
        attraction_with_image = {
            "name": attraction['name'],
            "describe": attraction['description'],
            "图片 url": images[0]["image"] if images else ""
        }
        result['景点'].append(attraction_with_image)

    except Exception as e:
        print(f'搜索{attraction["name"]}的图片时出错: {str(e)}')
        # 如果出错, 仍然添加景点信息, 但不包含图片 URL
        result['景点'].append({
            "name": attraction["name"],
            "describe": attraction["description"],
            "图片 url": ""
        })

# 处理美食信息
for food in foods_list:
    try:
        # 使用 DuckDuckGo 搜索图片
        images = self.search_toolkit.search_duckduckgo(
            query=f'{city} {food["name"]} 美食',
            source="images",
            max_results=1
        )

        # 添加图片 URL
        food_with_image = {
            "name": food["name"],
```

```
        "describe": food["description"],
        "图片 url": images[0]["image"] if images else "",
    }
    result['美食'].append(food_with_image)

except Exception as e:
    print(f"搜索{food['name']}的图片时出错: {str(e)}")
    # 如果出错, 仍然添加美食信息, 但不包含图片 URL
    result['美食'].append({
        "name": food["name"],
        "describe": food["description"],
        "图片 url": ""
    })
# 处理美食店铺信息
for food_shop in food_shops_list:
    try:
        # 使用 DuckDuckGo 搜索图片
        images = self.search_toolkit.search_duckduckgo(
            query=f"{city} {food_shop['name']} 美食店铺",
            source="images",
            max_results=1
        )
        # 添加图片 URL
        food_shop_with_image = {
            "name": food_shop["name"],
            "describe": food_shop["description"],
            "图片 url": images[0]["image"] if images else ""
        }
        result['美食店铺'].append(food_shop_with_image)
    except Exception as e:
        print(f"搜索{food_shop['name']}的图片时出错: {str(e)}")
        # 如果出错, 仍然添加美食店铺信息, 但不包含图片 URL
        result['美食店铺'].append({
            "name": food_shop["name"],
            "describe": food_shop["description"],
            "图片 url": ""
        })
    try:
        # 获取当前脚本所在目录
        current_dir = os.path.dirname(os.path.abspath(__file__))
        # 创建 storage 目录路径
        storage_dir = os.path.join(current_dir, "storage")
        # 确保 storage 目录存在
        os.makedirs(storage_dir, exist_ok=True)
```

```
# 生成文件名（使用城市名和日期）
filename = os.path.join(storage_dir, f"{self.city}{self.days}天旅游信息.json")

# 将结果写入 JSON 文件
with open(filename, 'w', encoding='utf-8') as f:
    json.dump(result, f, ensure_ascii=False, indent=4)
    print(f"旅游攻略已保存到文件: {filename}")

except Exception as e:
    print(f"保存 JSON 文件时出错: {str(e)}")

return result

@app.route('/get_travel_plan', methods=['POST'])
def get_travel_plan():
    try:
        # 获取请求数据
        data = request.get_json()

        # 验证输入数据
        if not data or 'city' not in data or 'days' not in data:
            return jsonify({
                'status': 'error',
                'message': '请求必须包含 city 和 days 参数'
            }), 400

        city = data['city']
        days = data['days']

        # 验证 days 是否为整数
        try:
            days = int(days)
        except ValueError:
            return jsonify({
                'status': 'error',
                'message': 'days 参数必须为整数'
            }), 400

        # 创建 TravelPlanner 实例并获取结果
        travel_planner = TravelPlanner(city=city, days=days)
        results = travel_planner.process_attractions_and_food()

        return jsonify({
            'status': 'success',

```

```

        'data': results
    })

except Exception as e:
    return jsonify({
        'status': 'error',
        'message': f'处理请求时发生错误: {str(e)}'
    }), 500
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5002, debug=True)

同样的，我们在本地的 5002 端口启动了一个服务，我们使用 requests 库来调用测试一下效果：
import requests
import json

# API 端点
url = "http://localhost:5000/get_travel_plan"

# 请求数据
data = {
    "city": "上海",
    "days": 3
}

# 发送 POST 请求
try:
    response = requests.post(url, json=data)

    # 检查响应状态
    if response.status_code == 200:
        result = response.json()
        print("获取到的旅游计划: ")
        print(json.dumps(result, ensure_ascii=False, indent=2))
    else:
        print(f"请求失败: {response.status_code}")
        print(f"错误信息: {response.text}")

except requests.exceptions.RequestException as e:
    print(f"发送请求时发生错误: {e}")

```

这个模块用于搜集和整理旅游信息。信息主要包括旅游的一些景点、美食信息及对应图片的 url，以便于我们后面将他们转成图文攻略。

在大语言模型的应用开发中，我们常常使用 JSON 作为中间数据的逻辑保存格式，因为交互方便，很好地表示结构化的信息且方便人类阅读和理解。

以下是生成的三份参考结果

```
filename = os.path.join(storage_dir, f"{self.city}{self.days}天旅游信息.json")
```

命名逻辑是 {地点}+{时间}+旅游信息.json

暂时无法在飞书文档外展示此内容

暂时无法在飞书文档外展示此内容

暂时无法在飞书文档外展示此内容

4. 攻略生成模块

这个模块实现了一个旅行行程规划服务，主要流程如下：

核心思路是通过前端传来的城市和出游天数信息，先在本地数据库中查找是否已存有相应的旅游数据（例如景点、美食等），如果没有就自动检索和生成对应的旅游信息并存储起来。

随后，后端会使用 CAME 库与 Qwen2.5-72B-Instruct 模型，结合一段包含行程规划规则的系统消息，指导大模型生成完整的多日行程安排。为了让结果在前端方便查看，代码会将模型给出的文本格式化成 HTML，并在必要时将其中的图片链接转换成 标签，使用户可以直接预览行程攻略页面。如果用户想要下载，后端还可以将该 HTML 转换成 PDF 供导出。当用户对当前结果不满意时，可以再次与大模型交互，通过多轮对话来动态调整和优化最终的旅行方案。整个过程如图所示：当有城市与天数信息输入时，系统先判断本地库中是否存在可用数据；若存在则直接调用大模型生成行程，若不存在则先行检索并存储数据后再进行行程生成；最后通过前端界面查看生成的 HTML 或者 PDF，如果用户仍需修改，则再次通过大模型进行迭代。这样就完成了一套从检索数据到生成和展示定制化旅游攻略的完整流程。

暂时无法在飞书文档外展示此内容

```
import os
import json
import re
from flask import Flask, request, jsonify

from camel.configs import QwenConfig
from camel.models import ModelFactory
from camel.types import ModelPlatformType
from camel.toolkits import SearchToolkit
from camel.agents import ChatAgent
from dotenv import load_dotenv

load_dotenv()

app = Flask(__name__)

# 环境变量
os.environ["GOOGLE_API_KEY"] = os.getenv("GOOGLE_API_KEY")
os.environ["SEARCH_ENGINE_ID"] = os.getenv("SEARCH_ENGINE_ID")

# 模型初始化
qwen_model = ModelFactory.create(
    model_platform=ModelPlatformType.OPENAI_COMPATIBLE_MODEL,
```

```
model_type="Qwen/Qwen2.5-72B-Instruct",
api_key=os.getenv("MODELSCOPE_SDK_TOKEN"),
url="https://api-inference.modelscope.cn/v1",
model_config_dict=QwenConfig(temperature=0.2).as_dict(),
)

tools_list = [
    *SearchToolkit().get_tools(),
]

sys_msg = """""
你是一位专业的旅游规划师。请你根据用户输入的旅行需求，包括旅行天数、景点/美食的距离、描述、图片 URL、预计游玩/就餐时长等信息，为用户提供一个详细的行程规划。

```

请遵循以下要求：

1. 按照 Day1、Day2、... 的形式组织输出，直到满足用户指定的天数。
 2. 每一天的行程请从早餐开始，食物尽量选用当地特色小吃美食，列出上午活动、午餐、下午活动、晚餐、夜间活动（若有），并在末尾总结住宿或返程安排。
 3. 对每个景点或美食，提供其基本信息：
 - 名称
 - 描述
 - 预计游玩/就餐时长（如果用户未提供，可以不写或自行估计）
 - 图片 URL（如果有）
 4. 请调用在线搜索工具在行程中对移动或出行所需时长做出合理估计。
 5. 输出语言为中文。
 6. 保持回复简洁、有条理，但必须包含用户想要的所有信息。
-

```
agent = ChatAgent(
    system_message=sys_msg,
    model=qwen_model,
    message_window_size=10,
    output_language='Chinese',
    tools=tools_list
)
```

```
def create_usr_msg(data: dict) -> str:
```

.....

同你原先的实现，用于生成给大模型的用户输入消息

.....

```
city = data.get("city", "")
days_str = data.get("days", "1")
try:
    days = int(days_str)
```

```
except ValueError:  
    days = 1  
  
lines = []  
lines.append(f"我准备去{city}旅行，共 {days} 天。下面是我提供的旅行信息：\n")  
  
scenic_spots = data.get("景点", [])  
foods = data.get("美食", [])  
  
if scenic_spots:  
    lines.append("- 景点：")  
    for i, spot in enumerate(scenic_spots, 1):  
        lines.append(f"  {i}. {spot.get('name', '未知景点名称')}"")  
        if '距离' in spot:  
            lines.append(f"    - 距离: {spot['距离']}")  
        if 'describe' in spot:  
            lines.append(f"    - 描述: {spot['describe']}")  
        if '图片 url' in spot:  
            lines.append(f"    - 图片 URL: {spot['图片 url']}")  
  
if foods:  
    lines.append("\n- 美食：")  
    for i, food in enumerate(foods, 1):  
        lines.append(f"  {i}. {food.get('name', '未知美食名称')}"")  
        if 'describe' in food:  
            lines.append(f"    - 描述: {food['describe']}")  
        if '图片 url' in food:  
            lines.append(f"    - 图片 URL: {food['图片 url']}")  
  
lines.append(f"""\n  请你根据以上信息，规划一个 {days} 天的行程表。
```

从每天的早餐开始，到晚餐结束，列出一天的行程，包括对出行方式或移动距离的简单说明。

如果有多种景点组合，你可以给出最优的路线推荐。请按以下格式输出：

Day1:

- 早餐:
- 上午:
- 午餐:
- 下午:
- 晚餐:

...

Day2:

```
...
Day[days]:
...
.....
)
return "\n".join(lines)

def fix_exclamation_link(text: str) -> str:
    """
    先把类似  的写法，提取出其中的 http://xx.jpg,
    替换成纯 http://xx.jpg
    """
    md_pattern = re.compile(r'!\[.*?\]\((https?://\S+)\)')
    # 将  只保留 http://xxx
    # 比如把 "" -> "http://xx.jpg"
    return md_pattern.sub(lambda m: m.group(1), text)

def convert_picurl_to_img_tag(text: str, width: int = 300, height: int = 200) -> str:
    """
    将文本中的图片 URL 替换为带样式的 HTML img 标签，并让图片居中显示和统一大小
    兼容两步：先处理 ，再匹配 - 图片 URL: http://url
    """
    # 第一步：把  变成纯 url
    text_fixed = fix_exclamation_link(text)

    pattern = re.compile(r'-\s*图片 URL: \s*(https?://\S+)')
    replaced_text = pattern.sub(
        rf"""
        <div style="text-align: center;">
            
        </div>
        """,
        text_fixed
    )
    return replaced_text

def generate_cards_html(data_dict):
    """
    生成景点和美食卡片的 HTML 片段
    """
    spots = data_dict.get("景点", [])
    foods = data_dict.get("美食", [])
```

```
html_parts = []
# 景点推荐
html_parts.append("<h2>景点推荐</h2>")
if spots:
    html_parts.append('<div class="card-container">')
    for spot in spots:
        name = spot.get("name", "")
        desc = spot.get("describe", "")
        distance = spot.get("距离", "")
        url = spot.get("图片 url", "")
        card_html = f"""
        <div class="card">
            <div class="card-image">
                
            </div>
            <div class="card-content">
                <h3>{name}</h3>
                <p><strong>距离:</strong> {distance}</p>
                <p>{desc}</p>
            </div>
        </div>
        .....
        """
        html_parts.append(card_html)
    html_parts.append("</div>")
else:
    html_parts.append("<p>暂无景点推荐</p>")

# 美食推荐
html_parts.append("<h2>美食推荐</h2>")
if foods:
    html_parts.append('<div class="card-container">')
    for food in foods:
        name = food.get("name", "")
        desc = food.get("describe", "")
        url = food.get("图片 url", "")
        card_html = f"""
        <div class="card">
            <div class="card-image">
                
            </div>
            <div class="card-content">
                <h3>{name}</h3>
                <p>{desc}</p>
            </div>
        </div>
        .....
        """
        html_parts.append(card_html)
    html_parts.append("</div>")
```

```
        </div>
        .....
        html_parts.append(card_html)
        html_parts.append("</div>")
    else:
        html_parts.append("<p>暂无美食推荐</p>")

    return "\n".join(html_parts)

def generate_html_report(itinerary_text, data_dict):
    """
    将多日行程文本 + 景点美食卡片，合并生成完整 HTML
    """

    html_parts = []
    html_parts.append("<!DOCTYPE html>")
    html_parts.append("<html><head><meta charset='utf-8'><title>旅行推荐</title>")
    # 可以内联一些 CSS 样式
    html_parts.append("<style>")
    html_parts.append(""""
body {
    font-family: "Microsoft YaHei", sans-serif;
    margin: 20px;
    background-color: #f8f8f8;
    line-height: 1.6;
}

h1, h2 {
    color: #333;
}

.itinerary-text {
    background-color: #fff;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 2px 5px rgba(0,0,0,0.1);
    margin-bottom: 30px;
}

.card-container {
    display: flex;
    flex-wrap: wrap;
    gap: 20px;
    margin: 20px 0;
}

.card {
    flex: 0 0 calc(300px);
    border: 1px solid #ccc;
}
""")

    return "\n".join(html_parts)
```

```
border-radius: 10px;
overflow: hidden;
box-shadow: 0 2px 5px rgba(0,0,0,0.1);
background-color: #fff;
}

.card-image {
width: 100%;
height: 200px;
overflow: hidden;
background: #f8f8f8;
text-align: center;
}

.card-image img {
max-width: 100%;
max-height: 100%;
object-fit: cover;
}

.card-content {
padding: 10px 15px;
}

.card-content h3 {
margin-top: 0;
margin-bottom: 10px;
font-size: 18px;
}

.card-content p {
margin: 5px 0;
}

.image-center {
text-align: center;
margin: 20px 0;
}

.image-center img {
width: 300px;
height: 200px;
object-fit: cover;
}

""")  
html_parts.append("</style></head><body>")  
  
# 标题  
html_parts.append("<h1>旅行行程与推荐</h1>")  
  
# 行程文本
```

```
html_parts.append('<div class="itinerary-text">')
for line in itinerary_text.split("\n"):
    if not line.strip():
        continue
    if line.strip().startswith("Day"):
        html_parts.append(f"<h2>{line.strip()}</h2>")
    else:
        html_parts.append(f"<p>{line}</p>")
html_parts.append('</div>')

# 景点/美食卡片
cards_html = generate_cards_html(data_dict)
html_parts.append(cards_html)

html_parts.append("</body></html>")
return "\n".join(html_parts)

def save_html_file(city: str, days: str, html_content: str) -> str:
    """
    保存 HTML 内容到文件

    Args:
        city: 城市名
        days: 旅行天数
        html_content: HTML 内容

    Returns:
        str: 保存的文件路径
    """
    # 确保 storage 目录存在
    storage_dir = "storage"
    if not os.path.exists(storage_dir):
        os.makedirs(storage_dir)

    # 生成文件名
    filename = f'{storage_dir}/{city}{days}天旅游攻略.html'

    # 保存 HTML 内容
    with open(filename, "w", encoding="utf-8") as f:
        f.write(html_content)

    return filename

@app.route("/generate_itinerary_html", methods=["POST"])
```

```
def generate_itinerary_html():
    """
    请求 JSON 格式:
    {
        "city": "成都",
        "days": "3"
    }
    返回生成的 HTML 文件路径和内容
    """

    req_data = request.json or {}
    city = req_data.get("city", "")
    days = req_data.get("days", "1")

    json_filename = f"storage/{city}{days}天旅游信息.json"
    if not os.path.exists(json_filename):
        return jsonify({"error": f"文件 {json_filename} 不存在, 请检查输入的目的地和天数!"}), 404

    try:
        with open(json_filename, "r", encoding="utf-8") as f:
            data = json.load(f)
    except json.JSONDecodeError:
        return jsonify({"error": f"文件 {json_filename} 格式错误, 请检查文件内容!"}), 400

    # 1. 生成用户输入并调用大模型
    usr_msg = create_usr_msg(data)
    response = agent.step(usr_msg)

    # 2. 将模型输出中的图片 URL 替换成 <img ... />
    model_output = response.msgs[0].content
    end_output = convert_picurl_to_img_tag(model_output)

    # 3. 生成完整 HTML 报告
    html_content = generate_html_report(end_output, data)

    # 4. 保存 HTML 文件
    saved_file = save_html_file(city, days, html_content)

    # 5. 返回文件路径和 HTML 内容
    return jsonify({
        "file_path": saved_file,
        "html_content": html_content
    }), 200
```

```

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5003, debug=True)

```

这里我们使用另一种方式来测试服务接口，熟悉软件开发的小伙伴应该用过 Postman

The screenshot shows a Postman request to `http://localhost:5003/generate_itinerary.html`. The request method is POST, and the body contains a JSON object with `{"city": "成都", "days": "3"}`. The response status is 200 OK, with a duration of 21.69 s and a size of 18.44 KB. The response body is a large JSON string representing an HTML document for a 3-day travel itinerary in Chengdu. It includes sections for travel tips, local specialties like '老妈蹄花' (Lao Ma's Braised Pork Hock), and travel history like '武侯祠' (Wu Hou Temple). It also mentions '锦里古街' (Jinli Ancient Street) and '春熙路' (Chunxi Road).

我们在 postman 中使用 Post 请求发送一份数据来模拟前端的请求，从而得到后端返回的数据，即 `html_content`，然后由 `save` 逻辑保存到本地的 storage 数据库中

The screenshot shows the browser's storage panel with four items: `上海3天旅游信息.json`, `成都3天旅游信息.json`, `成都3天旅游攻略.html`, and `新疆7天旅游信息.json`. The `成都3天旅游攻略.html` file is highlighted.

此时其实我们已经得到了一份不错的可展示的攻略了，在本地的浏览器中双击即可打开并看到渲染效果

暂时无法在飞书文档外展示此内容

旅行行程与推荐

Day1:

- 早餐:
- 名称: 赖汤圆
- 描述: 成都的老字号甜品, 以糯米制成, 内含红糖或芝麻馅, 口感软糯。



- 行程建议: 从酒店出发, 步行至附近的赖汤圆店享用早餐, 大约需要10分钟。

- 上午:

- 名称: 武侯祠
- 描述: 成都著名的古代建筑群, 纪念三国时期蜀汉丞相诸葛亮。



- 预计游玩时长: 2小时

- 行程建议: 乘坐地铁3号线至高升桥站下车, 步行约10分钟到达武侯祠, 全程约30分钟。

- 午餐:

- 名称: 龙抄手
- 描述: 成都的老字号小吃, 皮薄馅嫩, 汤汁鲜美, 以猪肉为主要原料。

景点推荐



青城山

距离:
中国著名的道教名山, 绿树成荫、空气清新, 适合休闲徒步和参观寺庙, 感受道教文化。



武侯祠

距离:
成都著名的古代建筑群, 纪念三国时期蜀汉丞相诸葛亮。



锦里古街

距离:
一条充满成都特色的古老街道, 体验成都的传统文化。



文殊院

距离:
成都著名的佛教寺塔, 历史悠久, 环境幽静。



大慈寺

距离:
成都著名的佛教寺塔, 有悠久的历史和丰富的文化遗产。



春熙路

距离:
成都最繁华的商业街区之一, 购物和美食的大本营。



金沙遗址博物馆

距离:
展示金沙遗址出土文物的博物馆, 展示了古蜀文明的重要场所。



杜甫草堂

距离:
唐代诗人杜甫在成都的故居, 展示了杜甫的生活和诗歌创作。



宽窄巷子

距离:
成都著名的旅游景点, 保留了许多传统的建筑和老成都的生活方式。

美食推荐



串串

成都街头常见的小吃, 可以串上各种食材, 如肉、蔬菜等, 再配上特制的蘸料食用。



麻辣烫

四川盆地特有的传统小吃, 主要材料有红油辣子, 辅料有肥肠、菜籽油、干红辣椒、花椒等, 麻辣鲜香, 色红味浓。



三大炮

成都的传统小吃, 由糯米制成, 外层裹有黄豆粉, 内含红糖, 口感软糯。



龙抄手

成都的名小吃, 皮薄馅嫩, 汤汁鲜美, 以猪肉为主要原料。



钟水饺

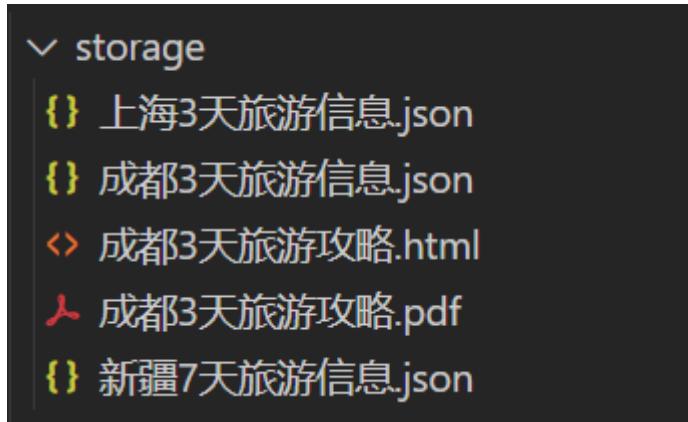
成都的老字号小吃, 以豌豆淀粉皮为主料, 用猪肉和蔬菜做馅。



赖汤圆

成都的老字号甜品, 以糯米制成, 内含红糖或芝麻馅, 口感软糯。

最后我们再把 HTML 转成在任何显示设备上都一致的 PDF 格式, 保证了攻略的统一性, 不必担心错位, 换行等一系列问题。



暂时无法在飞书文档外展示此内容

5. 反馈优化模块

反馈优化模块需要使用到具备多模态能力的大语言模型，因此需要大家再申请一个具有图像理解能力的大模型（这里我们以 gpt-4o 演示）。

在反馈优化中，我们所有的操作都是基于 PDF 的，虽然存在中间过程，但是对于用户来说是无感知的。具体而言，对于一个 PDF，我们会先识别出用户做过标记圈画的部分，然后将其所在的整页转化为 image 的形式作为输出源的一部分交给有多模态理解能力的**反馈优化 Agent**。

反馈优化 Agent 能够很好的理解用户的修改提示和意图，在 HTML 层面做出修正，重复之前的旅游信息检索和攻略生成模块，完成闭环，从而迭代式的优化旅游攻略，直到用户满意为止。

```
(island) PS C:\Users\YUI\Desktop\navigator\backend> python main.py
Loading pages (1/6)
Warning: Failed to load https://upload.wikimedia.org/wikipedia/commons/3/3e/Zhongnanhai.jpg (ignore)
Counting pages (2/6)
Resolving links (4/6)
Loading headers and footers (5/6)
Printing pages (6/6)
Done
HTML文件 beijing-user-modified.html 已成功转换为PDF并保存为 C:/Users/YUI/Desktop/navigator/frontend/public\beijing-user-modified.pdf
{'reply': '哈咯咯，已成功根据您的要求修改了旅游攻略，请查收！'}
```

在线编辑 PDF，调整需求：

需求一：红色部分帮我替换成中南海

需求二：文案帮我调整成小红书风格

旅行行程与推荐

根据您提供的信息，我为您规划了一个为期三天的北京旅行行程。考虑到北京是一个非常大的城市，为了使您的旅行更加顺畅，我会尽量选择地理位置接近的景点进行组合，并推荐合理的交通方式。以下是具体的行程安排：

Day1:

- 早餐：北京传统早点 - 炒肝
- 描述：北京的传统早点，主要材料为猪肝和猪肠，口感醇厚。



- 上午：参观天安门广场
- 描述：位于北京市中心，是世界上最大的城市广场之一，周围有多个重要的政治文化建筑。



我不想去天安门，我想去中南海参观，帮我修改这部分行程

- 出行建议：从酒店出发，乘坐地铁1号线直达天安门东站下车，步行至天安门广场。
- 午餐：王府井小吃街
- 描述：位于北京市中心，是一条著名的商业街，汇集了各种商店、餐馆和娱乐设施。



- 出行建议：从天安门广场步行约10分钟即可到达。
- 下午：游览故宫博物院
- 描述：专业博物馆，位于北京市中心，是明清两代皇家宫殿，也是世界上最大的古代木质结构建筑群。

北京三日游攻略 | 吃喝玩乐全安排

北京，一个历史与现代交融的城市！这次为大家安排了超详细的三日游攻略，吃喝玩乐全覆盖～快收藏起来吧！

Day 1: 历史与美食的碰撞

早餐：北京传统早点 - 炒肝

推荐理由：地道北京味儿，猪肝和猪肠的完美结合，口感醇厚，超满足！



上午：参观中南海

推荐理由：国家政治中心，环绕着美丽的湖泊和园林，拍照超大片！



交通建议：从酒店出发，乘地铁1号线到西单站下车，步行约15分钟即可到达。

午餐：王府井小吃街

推荐理由：各种小吃应有尽有，边逛边吃，幸福感爆棚！



交通建议：从中南海步行约10分钟即可到达。

下午：游览故宫博物院

推荐理由：明清两代皇家宫殿，世界上最大的古代木质结构建筑群，历史迷必打卡！

观察到文案风格确实更加活泼，且红圈部分正确修改成了中南海和配图也十分吻合。

6. 搭配前端食用

综合案例中为了尽可能让大家直观的感受到基于 CAMEL 框架的 Agent 产品开发的乐趣，我们还为大家准备了精美的前端，搭配之前的几个后端模块一起构成我们的 NavigatorAI 旅游

出行规划助手应用。

这里只给出一些功能上的描述和简介，前端的具体框架和原理不属于本次我们教程的讨论范围，感兴趣的同学可以自行拓展~

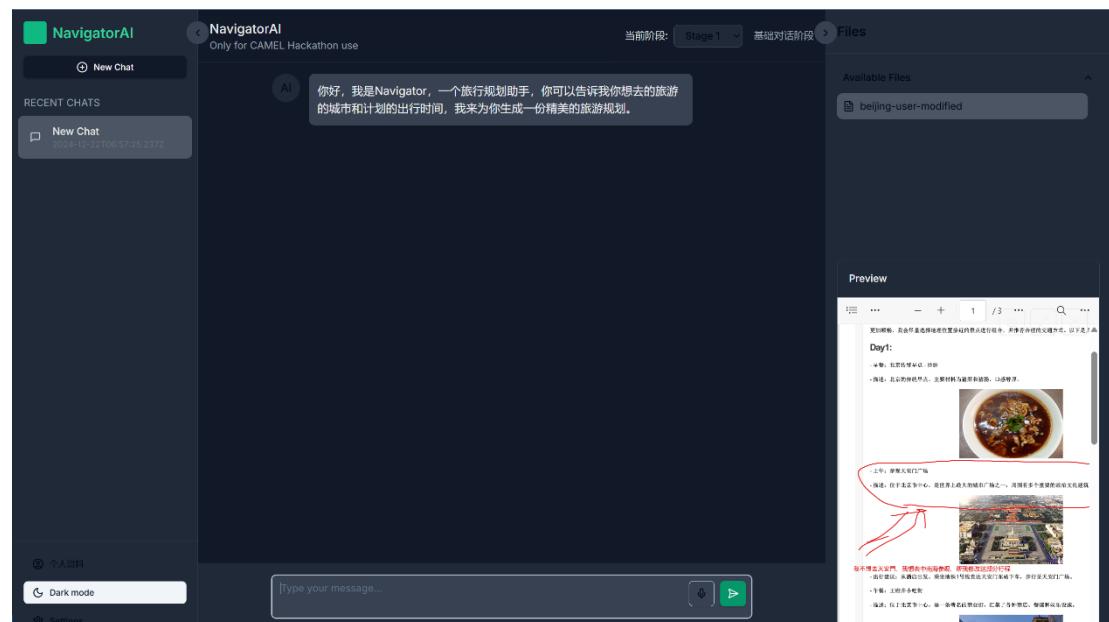
UI 在线交互模块

UI 在线交互模块是系统中与用户交互的关键部分，旨在通过直观的界面实现对生成行程内容的在线预览、编辑和反馈。用户可以在界面中直接查看生成的 PDF 文档，实时圈选、标注或输入修改意见，系统会根据用户提供的反馈动态调整行程内容。

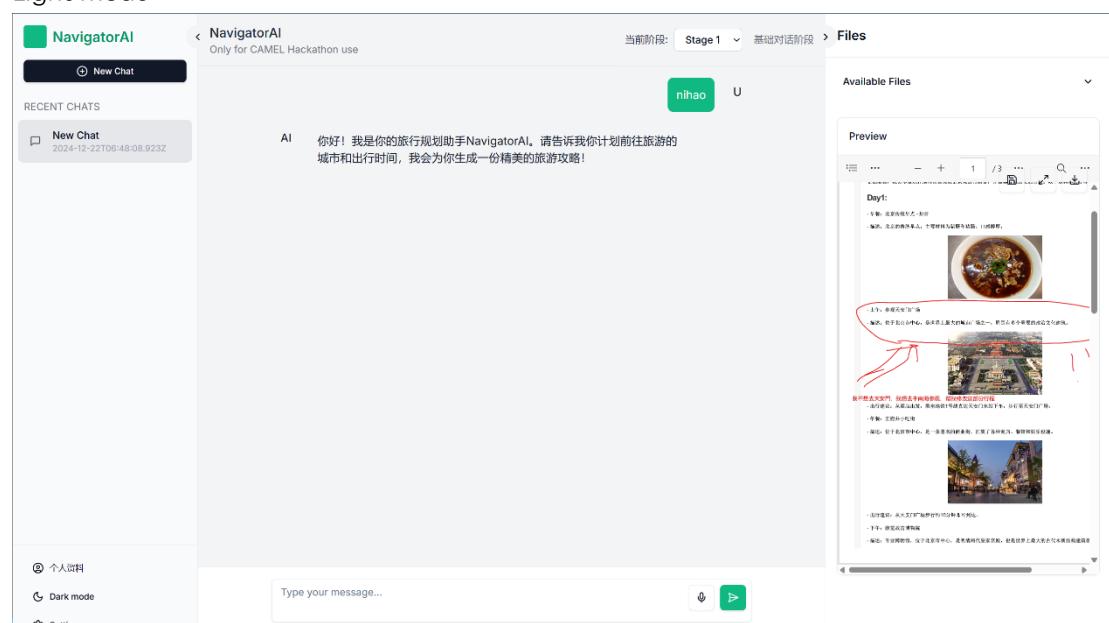
左右两边的侧边栏分别是历史记录+设置栏和文件交互栏，两侧都支持展开和隐藏，非常便捷和人性化。

支持昼夜模式切换

Dark mode



Light mode



1. PDF 在线渲染

1. 将生成的行程文案以 PDF 格式直接嵌入到界面中，用户可以实时浏览内容。

2. 支持高质量的 PDF 渲染，确保用户能直观查看文字、图片和布局。

2. 动态交互编辑

1. 用户可以直接点击 PDF 文档中的任意内容进行标注或编辑。
2. 提供圈选工具，允许用户高亮文中需要修改的部分并附加文本说明。
3. 输入框支持对具体段落、景点描述或图片的修改建议。

3. 反馈与动态调整

1. 用户的修改意见将自动传递给大模型 (LLM)，系统分析反馈并对行程内容进行优化。
2. 支持多轮交互，用户可以反复调整文案，直至满意为止。

4. 实时预览

1. 调整后的內容会立即在界面中重新渲染，用户无需等待即可看到更新后的 PDF 文档。
2. 保持高效的交互流程，提升用户体验。

个人信息输入模块

Navigat
划的出行

个人资料

姓名
姜舒凡

年龄
22

MBTI

月收入
1000

同行人数
2

目标城市
上海

规划时间

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| | Su | Mo | Tu | We | Th | Fr | Sa |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 | |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 | |
| 29 | 30 | 31 | 1 | 2 | 3 | 4 | |

保存更改

个人信息输入模块是系统的重要组成部分，用于收集用户的基础信息以便生成更符合个性化需求的旅游行程。通过直观友好的界面设计，用户可以快速填写相关信息，模块根据这些输入数据定制化生成行程内容。

1. 用户基础信息录入

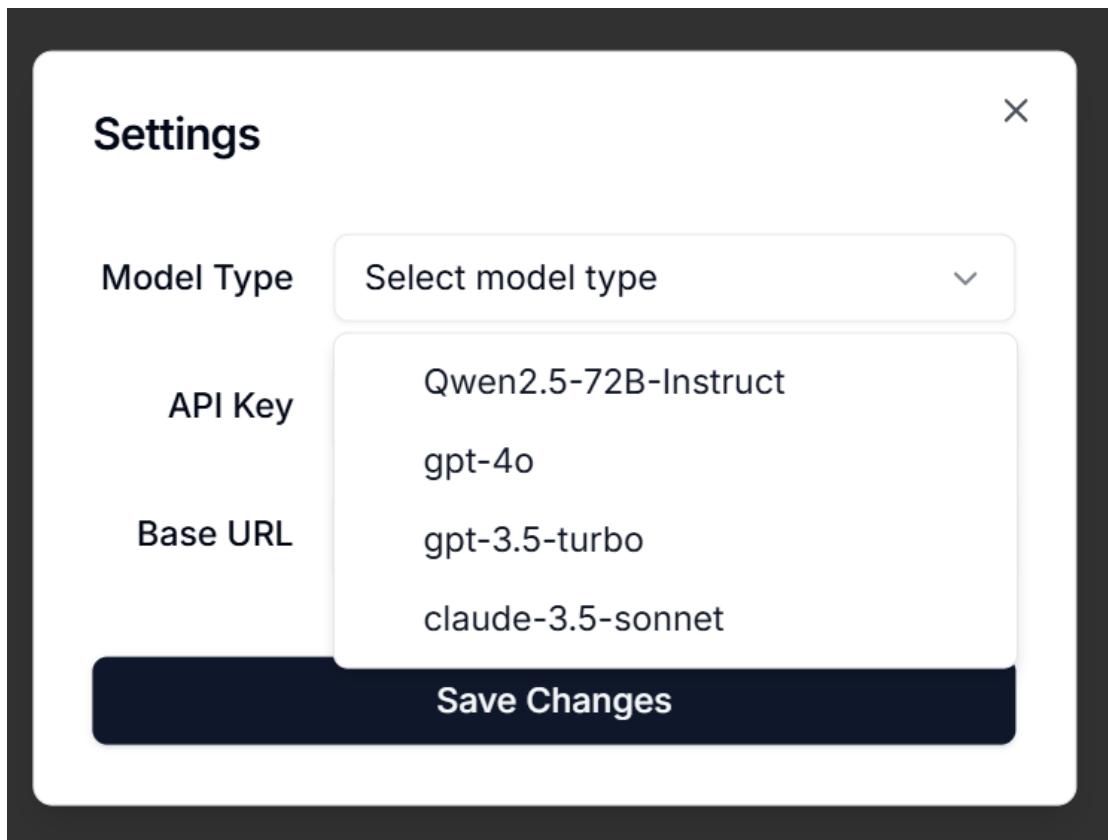
- 支持用户输入包括姓名、年龄、MBTI（个性类型）、月收入等个人信息。
- 数据用于优化旅游规划的推荐内容，例如推荐适合预算的活动和美食。

2. 出行偏好设置

- 用户可以指定目标城市、同行人数以及出行的具体日期（开始日期和结束日）

- 期)。
2. 系统会根据这些信息计算出行程天数并匹配对应的景点和活动。
 3. **交互式日历组件**
 1. 嵌入式日历设计，方便用户快速选择出发日期和返回日期。
 2. 界面清晰，操作直观，同时支持日期范围校验，避免选择无效的时间范围。
 4. **即时保存与更新**
 1. 提供“保存更改”按钮，允许用户修改输入信息并实时保存，确保数据更新准确。
 2. 便捷的数据交互设计减少了用户的操作负担，提高了使用效率。

模型选择



我们的模型选择模块是一款针对多语言模型应用需求设计的配置工具。用户只需要简单几步，就可以在界面中切换模型，并快速完成初始化配置，随时切换和管理多种主流模型。下拉菜单集成了目前支持的模型：

- **Qwen2.5-72B-Instruct**: 通义千问团队的最新开源模型。(综合案例中的大部分演示是基于此 API)
- **gpt-4o**: 拥有强大的推理能力和多模态能力，能够处理一系列复杂任务。
- **gpt-3.5-turbo**: 轻量级、响应快。
- **claude-3.5-sonnet**: 适合对话生成和自然语言理解的场景。
-

5.7 期终课程作业

作业一：[OWL](#)是基于CAMEL，专注在完成真实世界复杂任务的开源项目，请参考OWL中的示例代码，使用CAMEL内置的各类工具并结合你自己定义的工具来构建一个能够解决特定复杂任务的综合用例。点击[此处](#)查看详细内容。

作业二:通过系统性的学习相信你已经了解了如何去构建属于你的 Agent 和多 Agent 系统,请回顾整体的内容并思考, 我们可以在哪些方面进一步优化多智能体系统, 将你的想法提交到 CAMEL 仓库的 [issue 列表](#) 中, 并参考[贡献者指南](#)提交你的代码来进行优化。

结语

恭喜您! 在阅读到这里时, 您已经掌握了 CAMEL-AI 应用开发框架的基础知识, 并能够利用 Agent 实例在您的代码逻辑中执行复杂的任务。此外, 您还可以尝试将 Agent 实例整合到更复杂的工作流中, 或实现 Agent 实例之间的协同工作。

同时, 我们诚挚地邀请您继续关注我们的文档更新。在未来的文档中, 我们将对 CAMEL 框架的高级特性进行深入探讨, 包括如何使用 Workflow 组织和编排 LLMs 应用工作流程、如何理解和开发框架的插件能力, 以及如何构建多 Agent 协作方案等。

祝您学习愉快, 学有所获, 期待下次的相聚!

关注我们

请扫描下方二维码, 关注我们的公众号:





Datawhale

CAMEL-AI

附录

Loader 补充说明

Apify Reader

Apify Reader 提供了一个 Python 接口，用于与 Apify 平台交互，以实现 Web 工作流的自动化。你可以在这里获取所需要的 [APIKEY](#)。

初始化客户端，设置所需参数。

```
from camel.loaders import Apify  
import os
```

```
os.environ["APIFY_API_KEY"] = "你的 apikey"
```

```
apify = Apify()
```

```
run_input = {  
    "startUrls": [{"url": "https://www.camel-ai.org/"}],  
    "maxCrawlDepth": 0,  
    "maxCrawlPages": 1,  
}  
actor_result = apify.run_actor(  
    actor_id="apify/website-content-crawler", run_input=run_input  
)
```

检索结果数据库 ID 并使用 `get_dataset_items` 方法访问它。

```

dataset_result = apify.get_dataset_items(
    dataset_id=actor_result["defaultDatasetId"]
)

print(dataset_result)

>>>
[{'url': 'https://www.camel-ai.org/', 'crawl': {'loadedUrl': 'https://www.camel-ai.org/', 'loadedTime': '2025-02-06T07:11:19.056Z', 'referrerUrl': 'https://www.camel-ai.org/', 'depth': 0, 'httpStatusCode': 200}, 'metadata': {'canonicalUrl': 'https://www.camel-ai.org/', 'title': 'CAMEL-AI', 'description': 'CAMEL-AI.org is the 1st LLM multi-agent framework and an open-source community dedicated to finding the scaling law of agents.', 'author': None, 'keywords': None, 'languageCode': 'en', 'openGraph': [{'property': 'og:title', 'content': 'CAMEL-AI'}, {'property': 'og:description', 'content': 'CAMEL-AI.org is the 1st LLM multi-agent framework and an open-source community dedicated to finding the scaling law of agents.'}, {'property': 'twitter:title', 'content': 'CAMEL-AI'}, {'property': 'twitter:description', 'content': 'CAMEL-AI.org is the 1st LLM multi-agent framework and an open-source community dedicated to finding the scaling law of agents.'}, {'property': 'og:type', 'content': 'website'}], 'jsonLd': None, 'headers': {'content-type': 'text/html', 'transfer-encoding': 'chunked', 'connection': 'keep-alive', 'date': 'Wed, 05 Feb 2025 21:46:50 GMT', 'cf-ray': '90d61869d8a0ef20-PDX', 'cf-cache-status': 'MISS', 'content-encoding': 'gzip', 'last-modified': 'Wed, 05 Feb 2025 21:46:50 GMT', 'strict-transport-security': 'max-age=31536000', 'surrogate-control': 'max-age=432000', 'surrogate-key': 'www.camel-ai.org' 6659a154491a54a40551bc78 pagId:6686a2bcb7ece5fb40457491 679b765c27842bdba036423f 668181a0a818ade34e653b24 6659a155491a54a40551bd7e', 'x-lambda-id': 'ce2c42a4-c711-4bd7-b262-58374d893ffa', 'set-cookie': '_cfuvid=55wNMTV.q_HQ_jKuXybjfL5A9sxw0Wd30vX5Ff26cJA-1738792010775-0.0.1.1-604800000; path=/; domain=.cdn.webflow.com; HttpOnly; Secure; SameSite=None', 'x-cluster-name': 'us-west-2-prod-hosting-red', 'vary': 'Accept-Encoding', 'x-cache': 'Hit from cloudfront', 'via': '1.1 6a783450a00c2b23baf9dd1c4a552cb0.cloudfront.net (CloudFront)', 'x-amz-cf-pop': 'LAX54-P3', 'x-amz-cf-id': 't9nN-HwYopeCl40sOYSwOcFNJJT3I_KFOsxZuiD3I_23GLC9K5OEBA==', 'age': '33862'}}, 'screenshotUrl': None, 'text': 'Building Multi-AgentSystems for World Simu_\nCAMEL's multi-agent approach streamlines large-scale synthetic data creation and labeling. By assigning different specialized roles to each agent, it encourages dynamic, chain-of-thought collaborations that yield high-quality outputs. The orchestrated interaction ensures comprehensive coverage of data variations and consistency across domains. This makes it ideal for generating training sets, question-answer pairs, or other structured content.\nCAMEL powers automated workflows by breaking down complex tasks among coordinated agents. Each agent assumes a specific role, collaborating through an iterative conversational framework. This reduces manual intervention, minimizes errors, and accelerates solution delivery—especially useful for repetitive or logic-intensive processes.\nCAMEL enables simulations of dynamic, interactive worlds. Agents act as entities with distinct personas, communicating and responding in real time. This setup can be used'}
```

for modeling scenarios, building interactive storylines, or testing multi-layered strategies. By capturing the nuances of diverse viewpoints, CAMEL creates immersive simulations for experimentation, training, and creative exploration.', 'markdown': '# Building Multi-AgentSystems for World Simu_n\nCAMEL's multi-agent approach streamlines large-scale synthetic data creation and labeling. By assigning different specialized roles to each agent, it encourages dynamic, chain-of-thought collaborations that yield high-quality outputs. The orchestrated interaction ensures comprehensive coverage of data variations and consistency across domains. This makes it ideal for generating training sets, question-answer pairs, or other structured content.\n\n\n\nCAMEL powers automated workflows by breaking down complex tasks among coordinated agents. Each agent assumes a specific role, collaborating through an iterative conversational framework. This reduces manual intervention, minimizes errors, and accelerates solution delivery—especially useful for repetitive or logic-intensive processes.\n\n\n\nCAMEL enables simulations of dynamic, interactive worlds. Agents act as entities with distinct personas, communicating and responding in real time. This setup can be used for modeling scenarios, building interactive storylines, or testing multi-layered strategies. By capturing the nuances of diverse viewpoints, CAMEL creates immersive simulations for experimentation, training, and creative exploration.\n\n']

这个函数通常用于从 Apify 平台获取爬取或处理后的数据，以便在后续程序中使用这些数据。

Firecrawl Reader

你可以从[此处](#)获得你的 Firecrawl API

Firecrawl Reader 提供了一个 Python 接口来与 Firecrawl API 交互，允许用户将网站转换为大型语言模型可读的 markdown 格式。

初始化客户端并设置要从中检索信息的 URL。当状态为“completed”时，信息检索已完成并可供阅读。

```
from camel.loaders import Firecrawl
```

```
os.environ["FIRECRAWL_API_KEY"] = "你的apikey"
```

```
firecrawl = Firecrawl()
```

```
response = firecrawl.crawl(url="https://www.camel-ai.org/about")
print(response["status"])
```

```
>>>
```

```
completed
```

直接从返回的结果中检索信息。

```
print(response["data"][0]["markdown"])
```

>>> Camel-AI Team

We are finding the scaling law of agent

camel CAMEL is an open-source library designed for the study of autonomous and communicative agents. We believe that studying these agents on a large scale offers valuable insights into their behaviors, capabilities, and potential risks. To facilitate research in this field, we implement and support various types of agents, tasks, prompts, models, and simulated environments.

****We are**** always looking for more ****contributors**** and ****collaborators****.

Contact us to join forces via [Slack](https://join.slack.com/t/camel-kwr1314/shared_invite/zt-1vy8u9lbo-ZQmhIAyWSEfSwLCI2r2eKA)

or [Discord](<https://discord.gg/CNcNpquyDc>)...

Jina_url Reader

JinaURL Reader 是 Jina AI 的 URL 读取服务的 Python 客户端，经过优化，可从 URL 提供更清晰、对 LLM 可读的内容。该 reader 提供一定额度的免费调用次数。无需注册 API 也可使用。

```
from camel.loaders import JinaURLReader  
from camel.types.enums import JinaReturnFormat
```

```
jina_reader = JinaURLReader(return_format=JinaReturnFormat.MARKDOWN)  
response = jina_reader.read_content("https://www.datawhale.cn/home")
```

```
print(response)
```

>>>

Datawhale-学用 AI, 从此开始

=====

![Image 1: logo](<https://www.datawhale.cn/assets/logo-vAxrscYT.png>)

| | |
|--|------------------|
| [HOME 首页] (https://www.datawhale.cn/home) | [OPEN 1+XAI 通识课] |
| [ACTIVITY 动动] (https://www.datawhale.cn/open-ai) | [活课] |
| [COURSE 程程] (https://www.datawhale.cn/activity) | [课程] |
| [CERTIFICATION 认证] (https://www.datawhale.cn/learn) | [认识] |
| [NEWS 资讯] (https://www.datawhale.cn/cert) | [新闻] |
| [ARTICLE 资讯] (https://www.datawhale.cn/article) | [资讯] |

![Image wechat]([blob:<https://www.datawhale.cn/cfd0357cb7301be8879b48a64347d08f>])

2: 
3: <https://www.datawhale.cn/assets/wechat-qrcode-CPBf96kV.png>

4:

![Image
github](blob:<https://www.datawhale.cn/4d659f53c87d622c9cafa1b45920725f>)](<https://github.com/datawhalechina>)

登录

![Image 5](<https://www.datawhale.cn/assets/homebanner-left-DQp9I1cY.png>)

Datawhale for the learner

![Image 6](<https://www.datawhale.cn/assets/banner-title-CqrNZXqM.png>)

Datawhale 通过开源学习模式，助力 AI 学习者与知识连接，与人连接，与场景连接，推动 AI 人才培养。

![Image 7](<https://www.datawhale.cn/assets/homebanner-swiper1-CnhP9rkt.png>)

![Image 8: logo](blob:<https://www.datawhale.cn/7dd095def689b2660c34f51de0662bf6>)

...

社 交 媒 体 [GITHUB](<https://github.com/datawhalechina>) [B 站](<https://space.bilibili.com/431850986>) [CSDN](<https://blog.csdn.net/Datawhale>)

![Image 30](blob:<https://www.datawhale.cn/a1d2760e5d7b70d1dbb76b573944cb7f>) 我要反馈](<https://jinshuju.net/f/gtuc5x>)

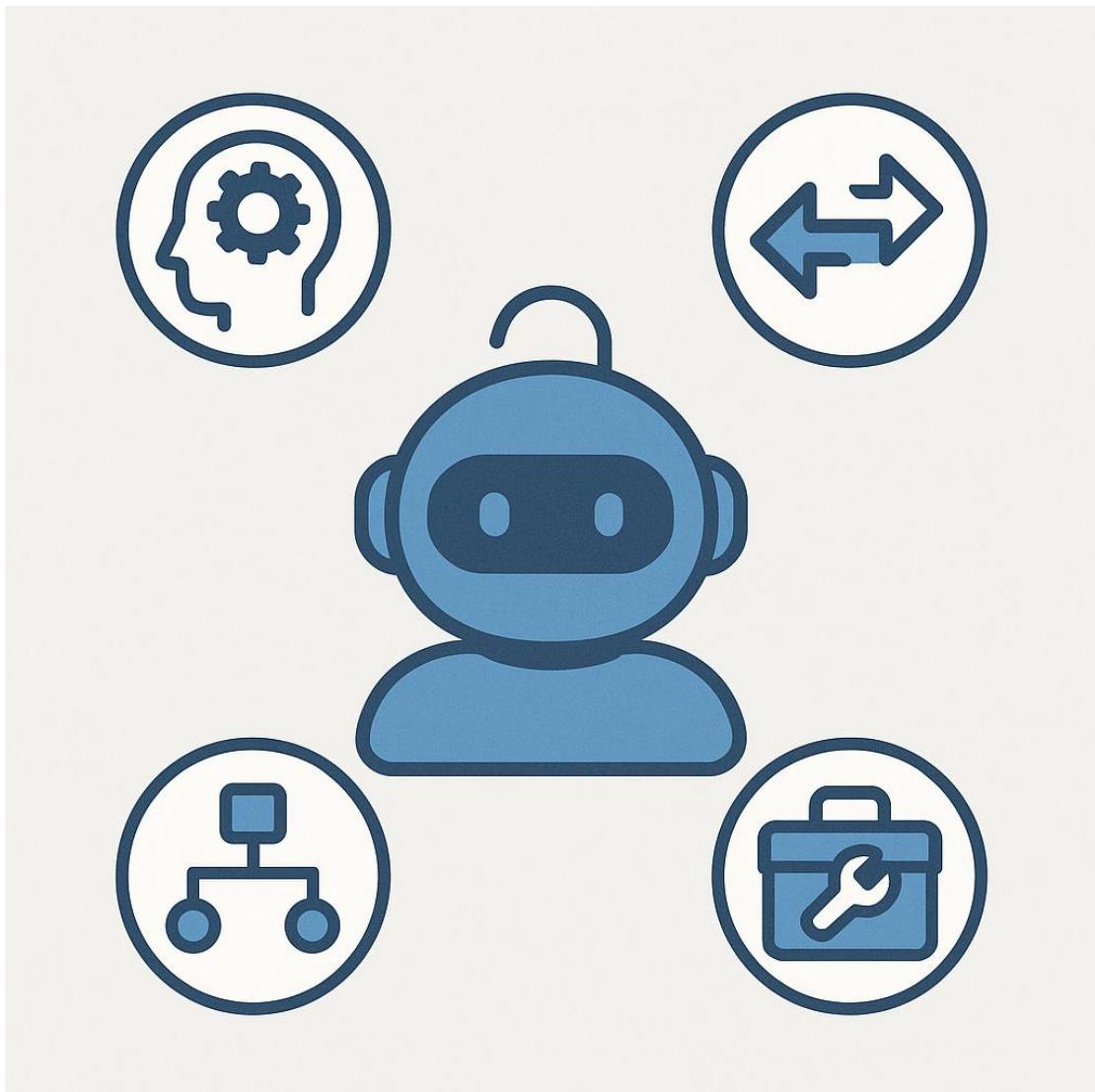
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

协议（初稿）

1. 协议简介

1.1 AI Agent 的崛起与面临的“沟通难题”

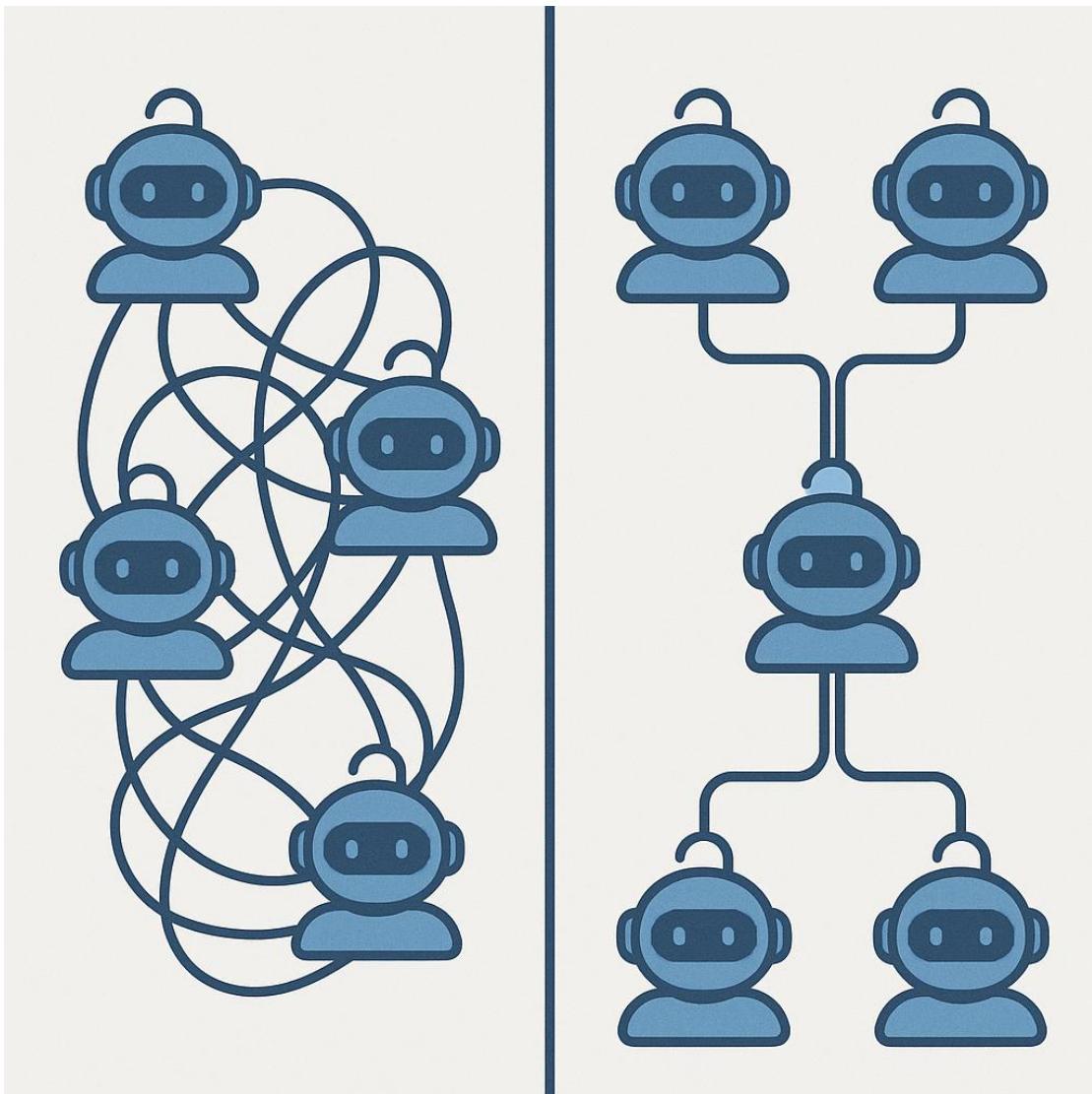
如今，基于大型语言模型（LLM）的 AI Agent 正变得越来越聪明。你可以把它们想象成拥有超级大脑的智能助手，它们不仅能处理信息，还能进行深度思考，把复杂任务拆解成小步骤，甚至能与外部环境互动，比如调用各种工具来增强自身能力。它们甚至可以模拟用户行为，为我们测试和改进信息系统提供反馈。



然而，随着这些 AI Agent 越来越多地应用于各个领域，一些挑战也浮出水面。比如，我们生活在一个信息爆炸的时代，用户提出的问题也越来越复杂多样，这对传统的搜索和信息处理方式是个不小的考验。AI Agent 需要有效地理解和管理大量的上下文信息，并且与五花八门的外部工具和系统打交道。一个常见的问题是，AI Agent 在执行需要多个步骤的复杂动作时，或者在多次互动中保持对同一件事情的持续理解时，会遇到困难，这就像人有时候会“断片儿”一样。很多现有的 AI 应用在设计上，并没有一个统一的方法来告诉 LLM 它当前需要了解的背景信息是什么，导致集成外部工具时常常需要临时抱佛脚，效果也不稳定。

1.2 为什么需要标准化的“沟通规则”？

目前，很多 AI Agent 在集成外部工具或与其他 Agent 协作时，采用的都是一些“一次性”的解决方案。这些方案不仅难以推广和扩展，安全性也难以保障，更不用说跨领域应用了。如果我们想充分发挥 AI Agent 的潜力，尤其是在多个 AI Agent 协同工作的系统中，就迫切需要一套强大且标准化的“沟通规则”。这些规则能帮助它们统一地使用工具、共享必要的背景信息，并在不同的系统间协调任务。



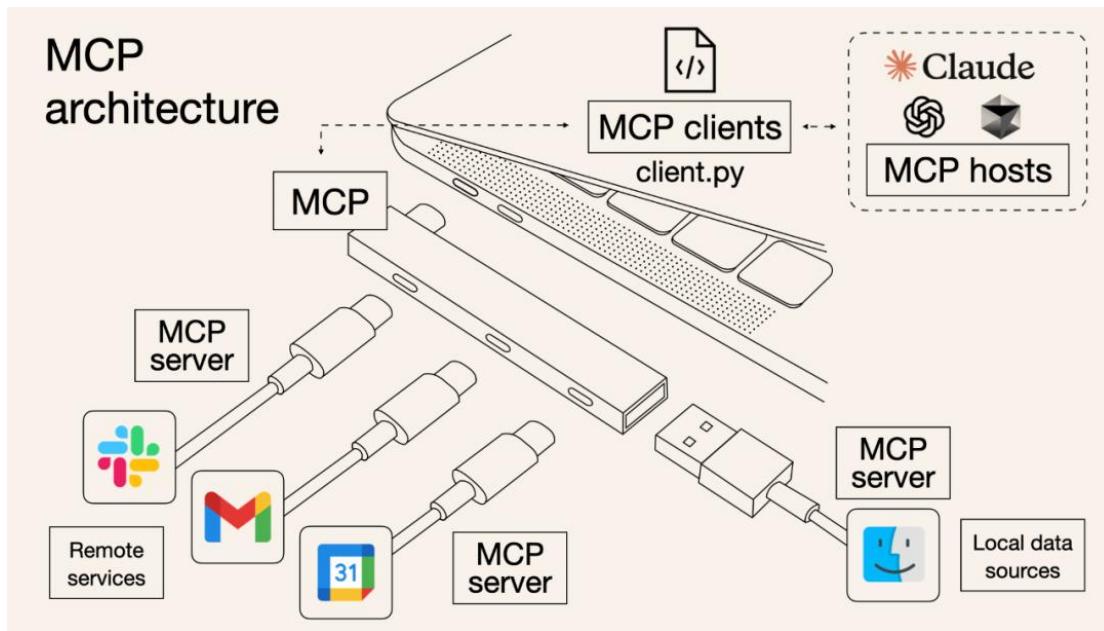
想象一下，如果没有统一的语言和行为规范，来自不同公司或者基于不同技术背景开发的 AI Agent 就很难顺畅地合作。这会限制 AI Agent 网络的扩展，最终也会限制它们解决更复杂现实问题的能力。这种情况有点像互联网早期，各个计算机系统互不兼容，连接非常受限，导致整个网络发展缓慢。一个统一的通信协议不仅仅能解决眼前的互操作问题，更有可能催生出一个由无数智能体连接而成的巨大网络。在这个网络里，各种形式的智能可以自由流动，最终形成一种超越任何单个 AI Agent 能力的集体智慧。因此，标准化协议对于实现 AI Agent 之间的无缝对接、高效协作以及形成集体智能至关重要。

1.3 本节主要内容

这小节将带你了解几种当前 LLM Agent 领域中非常关键的标准化协议，重点是模型上下文协议（MCP）、Agent 间协议（A2A）和 Agent 网络协议（ANP）。这些协议为解决 AI Agent 在理解上下文、使用工具和相互协作方面的难题提供了有效的思路。我们会详细解释这些协议是什么、它们的目标是什么、核心的设计思路是怎样的，以及它们是如何工作的和可以用在哪些地方。

此外，我们还会给出 CAMEL 是如何在实践中应用 MCP 协议的。具体来说，我们会讲到：如何在 CAMEL 环境中使用 MCP 工具，如何把你用 CAMEL 开发的工具转换成能被其他 MCP 系统识别和使用的服务，以及如何让你用 CAMEL 创建的 AI Agent 也变成一个 MCP 服务，供其他系统调用。这部分会包含一些可以直接运行的代码例子，帮助你更好地理解和上手。

2. MCP 协议 (Model Context Protocol)



2.1 MCP 是什么？它想做什么？

模型上下文协议 (MCP)，你可以把它理解成一个为 AI 模型（比如我们常说的 LLM Agent）与外部数据源和工具进行互动而设计的标准化框架。它的核心目标，就是解决大型语言模型在利用外部信息和工具时，缺乏一个统一的、标准化的“理解背景信息”机制的问题。MCP 努力提供一种标准方法，让应用程序能够清晰地告诉 LLM 它现在有哪些工具可以用、有哪些数据可以参考，以及应该遵循哪些基本指令。有人形象地把 MCP 比作“人工智能领域的 USB-C 接口”，意味着它希望成为一种通用的连接标准。通过这种标准化，MCP 能够支持灵活地“即插即用”各种工具，安全地集成到现有的 IT 设施中，并确保在不同的 LLM 提供商之间具有良好的兼容性。这个协议能让 AI 模型更有效地在不同的 Agent 互动中保留和共享上下文信息，从而增强它们处理复杂现实问题的能力。

2.2 MCP 的设计理念

MCP 在设计时遵循了一些核心原则，这些原则决定了它的架构和实现方式，确保它既好用又可靠：

- **通用性 (Interoperability):** MCP 的目标是跨越不同的 AI 模型、平台和环境工作。无论你用的是哪家的 AI 技术，MCP 都希望能提供一致的上下文管理方式。这体现在它不依赖特定编程语言的设计和标准化的数据格式上。
- **简单易用 (Simplicity):** 协议力求简单明了，容易上手。它专注于一小组核心的功能，这些功能可以组合起来支持复杂的应用场景。这样可以降低大家学习和使用的门槛，并鼓励在不同的系统中都能一致地实现它。
- **可扩展 (Extensibility):** 在保持核心简单的同时，MCP 也设计了清晰的扩展机制。这意味着未来可以方便地增加新功能，或者针对特定专业领域进行调整，而不会破坏已有的兼容性。这种平衡使得协议能够适应不断变化的需求。
- **安全和隐私优先 (Security and privacy by design):** MCP 从一开始就把安全和隐私放在非常重要的位置，而不是事后弥补。这包括清晰的权限管理、尽可能少地使用数据的原则，以及控制不同组件之间信息流动的机制。
- **以人为本的控制 (Human-centered control):** 协议的设计也考虑到了保持适当的人工监督和控制，尤其是在处理敏感操作或重要决策时。这体现了对人类判断在 AI

系统中重要性的认可。

2.3 MCP 是如何工作的？

MCP 采用的是一种常见的客户端-服务器架构。在这个架构里，AI 模型（比如 LLM Agent）扮演“客户端”的角色，而外部的数据源或工具则作为“服务器”运行。这种明确的分工有助于各司其职：客户端专注于 AI 的思考和推理，而服务器则负责提供数据和执行工具的功能。它们之间的沟通主要通过 JSON-RPC 2.0 进行。这是一种轻量级的远程过程调用协议，使用 JSON 这种通用的数据格式进行信息编码，从而确保了不同编程语言开发的系统之间也能相互理解。MCP 的通信方式也很灵活，既可以通过计算机程序间标准的输入输出流（Stdio）进行，也可以通过网页中常用的 HTTP 协议配合服务器发送事件（SSE）技术来实现，以适应不同的部署场景和互动需求。

MCP 定义了一套标准化的“原语”（可以理解为基本指令或构件）来进行上下文管理，这些原语分为服务器端和客户端两类：

- **服务器端原语：**

- **提示 (Prompts):** 预先定义好的一些指令或模板。AI 客户端可以向服务器请求这些提示，以确保其行为的一致性。
- **资源 (Resources):** 可以发送给 AI 模型的结构化数据或文档，它们通常带有描述信息（元数据），为 AI 提供必要的背景知识。
- **工具 (Tools):** AI 可以调用的可执行功能，用来执行某些操作或获取特定信息。每个工具都会清晰地定义其名称、用途描述、需要哪些输入参数以及会返回什么类型的结果。

- **客户端原语：**

- **根 (Roots):** 允许服务器访问客户端特定数据区域的入口点，主要用于设定权限边界，明确服务器可以访问哪些信息。
- **采样 (Sampling):** MCP 服务器请求 AI 模型生成一段续写或补全内容的机制。这使得服务器能够利用 AI 模型的推理能力来完成某些任务。

这种架构和组件设计使得 MCP 能够灵活部署，并且客户端和服务器可以独立发展和更新，只要它们都遵守 MCP 协议的规定就行。

2.4 MCP 的“一生”：交互的三个阶段

MCP 客户端（比如 AI Agent）和服务器（比如一个工具服务）之间的互动，遵循一个清晰定义的、分为三个阶段的生命周期。这个设计确保了会话管理的稳健性、能力协商的安全性以及明确的结束过程。

MCP

INITIALIZE



OPERATE



SHUTDOWN

- **初始化阶段 (Initialization):** 一切从这里开始。首先，客户端和服务器要确认彼此“说的是同一种语言”，也就是协议版本兼容。它们会协商一个双方都支持的最高协议版本。紧接着，它们会交换各自支持的“能力清单”，比如是否支持采样、是否提供提示、有哪些工具可用、是否支持日志记录等等。当客户端收到服务器同意初始化的响应

后，会发送一个“我已经准备好了”的通知，标志着双方可以开始正式的业务沟通了。

- **操作阶段 (Operation):** 这是核心的活动阶段。在这个阶段，客户端和服务器会根据在初始化阶段商量好的能力，互相发送 JSON-RPC 格式的指令调用和通知。双方都必须严格遵守之前商定的功能范围，这样才能保证兼容性和可预测性。客户端在调用服务器的某个功能时，可以设定一个超时时间。如果在这个时间内没有收到服务器的响应，客户端可以发送一个取消通知，以防止资源被无效占用或执行过时的操作。
- **关闭阶段 (Shutdown):** 这个阶段确保整个会话能够干净利落地结束。任何一方都可以通过关闭底层的通信连接（通常是 HTTP 连接或 stdio 流）来发起终止。一旦关闭，客户端和服务器都有责任清理各自占用的资源，比如取消还在等待的超时设置、取消订阅、释放相关的子进程等。此后，除了必要的诊断信息（比如测试网络通不通的 ping 消息或刷新日志），双方不应再发送新的协议消息。

2.5 MCP 与其他沟通方式有何不同？

- **与传统的“Agent 通信语言”(ACLs)相比:** 像 FIPA-ACL 和 KQML 这样的传统 ACLs，提供了一套形式化的语言和标准化的消息结构，支持像通知、请求、查询等复杂的互动模式。它们更侧重于定义沟通行为的“言语动作”（比如“告知”、“请求”）。相比之下，MCP 更关注模型如何获取和利用外部的“知识”和“能力”（也就是上下文资源和工具），而不是通用的 Agent 间对话逻辑。ACLs 有时会显得比较复杂和重量级，而 MCP 则力求简洁。
- **与基于“本体”的通信相比:** 基于本体的通信依赖于大家共享一套“本体”（可以理解为领域知识的结构化表示），来提供共同的词汇和语义框架，确保 Agent 们对特定领域的概念有统一的理解。这种方法能实现非常精确的领域概念交流，但通常需要投入大量精力预先开发好这个“本体”。MCP 本身并不强制要求复杂的本体，而是通过结构化的 JSON 模式来定义工具和资源的接口，更侧重于操作层面的互通有无。
- **与“自然语言”通信相比:** 随着 LLM 技术的发展，让 AI Agent 之间直接用自然语言（比如中文、英文）交流变得越来越可行。自然语言非常灵活，表达能力强。但是，自然语言也容易产生歧义，导致理解上的困难。MCP 通过 JSON-RPC 提供了明确的、机器可以准确解析的接口，减少了模糊性，为工具调用这类结构化交互提供了更高的可靠性。

总的来说，MCP 可以被看作是一种专注于 AI 模型与外部环境（工具和数据）之间“最后一公里”连接的协议。它强调的是结构化、安全且可互操作的上下文信息与功能的集成。

2.6 MCP 能用在哪些地方？

- **企业知识管理:** 想象一下，在一家大公司里，AI Agent 可以通过 MCP 连接到公司内部的各种数据仓库（比如数据库、文档系统、各种 API 接口）。这样，AI Agent 就能帮助员工快速查找信息、回答他们的疑问，甚至自动化一些需要大量知识处理的流程。MCP 确保了 AI Agent 能够以一致和安全的方式访问这些五花八门的内部资源。
- **科研协作:** 在科学研究领域，LLM Agent 可以通过 MCP 与多个学术数据库、模拟计算工具或者实验数据分析服务进行互动。例如，一个 AI Agent 可以利用 MCP 工具从不同的来源收集学术论文，然后使用另一个 MCP 工具进行数据分析和可视化，最终生成一份综合性的研究报告。
- **分布式问题解决:** 在一些复杂的分布式系统中，可能需要多个专门的 AI Agent 协同工作才能解决问题。MCP 可以作为它们与共享工具或数据服务进行交互的标准接口。比如，在供应链管理中，一个 AI Agent 可能负责预测市场需求，另一个负责优化库存，它们都可以通过 MCP 访问共享的销售数据和优化算法工具。

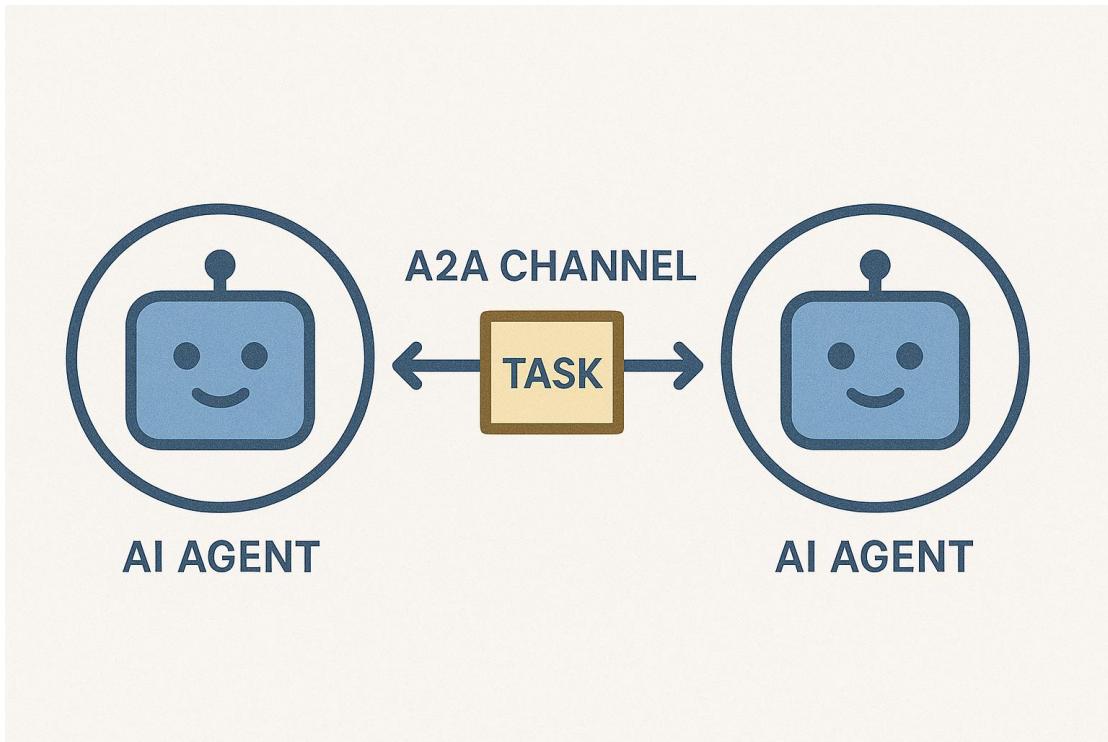
- **动态调用外部工具:** AI Agent 可以根据当前任务的需要，通过 MCP 动态地发现并调用可用的外部工具，比如进行网络搜索、生成图片、执行代码等等。这极大地扩展了 AI Agent 的能力边界，而不需要把所有可能用到的功能都预先内置到 Agent 内部。

这些应用场景都突显了 MCP 在增强 AI Agent 能力、提高其对环境的感知以及促进与外部世界无缝集成方面的价值。

3. A2A 协议 (Agent-to-Agent Protocol)

3.1 A2A 是什么？它想做什么？

Agent 间协议 (A2A)，是由 Google 等公司推动的一项技术框架，目标是让自主的 AI Agent 之间能够进行结构化、安全并且可互操作的通信。它的主要目的是支持在企业级的工作流程中，AI Agent 之间可以点对点地外包任务和进行协作。随着 AI 系统从执行简单的流程，发展到需要多个 AI Agent 进行复杂的协同工作，像 A2A 这样的协议就变得至关重要了。它致力于解决这样一个问题：在缺乏统一协作标准的情况下，不同的 AI Agent（无论是人类编写的还是 AI 驱动的）很难有效地发现对方、商议任务并一起执行。A2A 通过提供一个声明式的、能感知身份的框架，让 AI Agent 能够共享关于自身能力的描述性信息，从而促进有效的互动、发现和互操作。



3.2 A2A 的核心设计理念

A2A 协议的设计基于以下几个核心原则，以确保它在多 Agent 环境中的有效性和安全性：

- **Agent 优先 (Agentic-first):** 协议的设计以 AI Agent 的独立操作为核心。Agent 们通过明确的通信来交换信息，默认情况下不依赖共享的内存或共享的工具。这强调了生态系统中每个 Agent 的自主性。
- **符合标准 (Standards-compliant):** A2A 利用了广泛应用的 Web 技术，比如 HTTP (网页传输协议)、JSON-RPC (一种数据交换格式) 和服务器发送事件 (SSE，一种服务器向浏览器推送消息的技术)。这降低了开发者的学习成本，并确保了与现有 IT 基础设施的兼容性。
- **默认安全 (Secure by default):** 集成的身份验证和授权措施是协议设计的基础组成

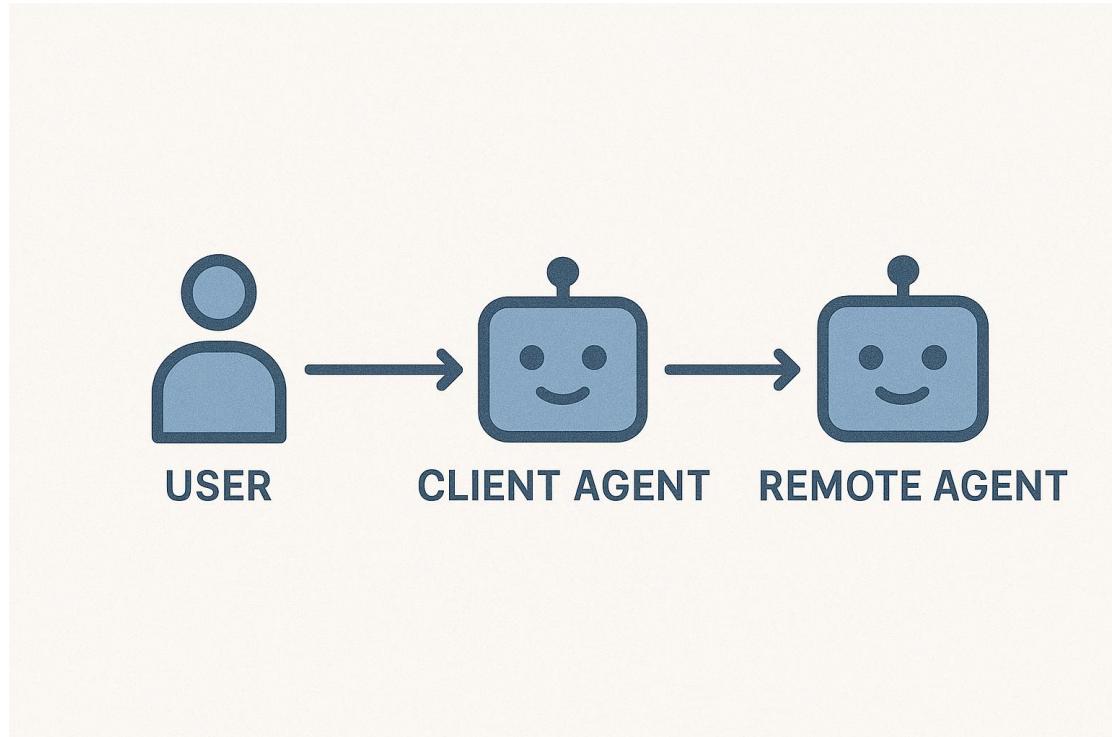
部分。这意味着从一开始就考虑如何保护敏感数据和交易的安全，而不是事后弥补。这些原则共同构成了 A2A 协议的基础，使其能够支持复杂的多 Agent 互动，同时保持安全性和易用性。

3.3 A2A 是如何工作的？

A2A 协议定义了一个清晰的架构和交互流程，主要涉及以下几个角色：

- **用户 (User):** A2A 交互的发起者。这个“用户”可以是真实的人，也可以是某个系统、服务或其他 AI Agent。用户通过“客户端 Agent”来发起请求。
- **客户端 Agent (Client Agent):** 代表用户与远程的 AI Agent 进行通信。它负责把用户的请求转换成可操作的任务，并协调所有收到的响应。
- **远程 Agent (Remote Agent / Server):** 提供特定能力并执行任务的 AI Agent。

它们之间的互动通常遵循以下步骤：



1. 客户端 Agent 向远程 Agent 请求/.well-known/agent.json（代理名片）。
 2. 远程 Agent 返回代理名片。
 3. 客户端 Agent 进行身份验证后，通过 HTTPS 向远程 Agent 的 a2aEndpointUrl 发送 JSON-RPC 请求（可区分 tasks.send 和 tasks.sendSubscribe 路径）。
 4. 远程 Agent 处理任务，并通过 HTTP 响应(tasks.send)或 SSE 事件(tasks.sendSubscribe)返回任务状态和工件。
 5. (可选) 如果需要输入，客户端 Agent 发送后续信息。
 6. 任务完成，最终结果通知客户端。
 7. (可选) 远程 Agent 通过 Webhook 发送推送通知。这个图可以突出 HTTPS 和 JSON-RPC 的使用。]
- **发现 (Discovery):** 客户端 Agent 首先要去“发现”远程 Agent。它通过从一个标准化的路径（通常是远程 Agent 服务器上的 /.well-known/agent.json 文件）获取一份叫做**“Agent 名片” (Agent Card)** 的东西。这份“名片”是一个 JSON 格式的元数据文件，里面声明了远程 Agent 的基本信息、它拥有哪些技能、A2A 通信的地址 (a2aEndpointUrl) 以及它要求使用哪种身份验证方法（通过 OpenAPI 3.x 安全方案

对象来指定)。

- **发起 (Initiation):** 客户端 Agent 根据“Agent 名片”上声明的认证方法进行身份验证。成功后, 它会通过 HTTPS(一种安全的网页传输协议)向远程 Agent 的 a2aEndpointUrl 发送一个 JSON-RPC 请求。主要使用两种方法:
 - tasks.send: 用于那些可能同步完成的任务, 或者不需要立即流式传输更新的任务。
 - tasks.sendSubscribe: 用于那些需要长时间运行, 并且需要通过服务器发送事件 (SSE) 在持久的 HTTPS 连接上持续发送更新的任务。
- **处理与交互 (Processing & Interaction):**
 - 对于非流式任务 (即使用 tasks.send 发起的任务), 服务器处理完任务后, 会在 HTTP 响应中返回最终的任务对象 (Task object)。
 - 对于流式任务 (即使用 tasks.sendSubscribe 发起的任务), 服务器会通过持久连接发送 SSE 消息。这些消息包括包含更新后任务对象的 TaskStatusUpdateEvent, 以及包含任务执行过程中生成的工件对象 (Artifact objects) (比如生成的文件或报告) 的 TaskArtifactUpdateEvent。
- **需要输入 (Input Required):** 如果服务器在处理任务过程中发现需要更多信息才能完成, 它会发出信号。客户端收到信号后, 会使用相同的任务 ID (taskId) 发送后续的输入信息。
- **完成 (Completion):** 当任务达到一个最终状态 (比如完成、失败或被取消), 这个结果会通过最终的响应或 SSE 事件通知给客户端。
- **推送通知 (Push Notifications) (可选):** 如果远程 Agent 服务器支持 pushNotifications 能力, 它可以向客户端预先提供的一个 Webhook URL (一个网络回调地址) 发送异步的更新。这个 Webhook URL 是通过 tasks.pushNotification.set 方法注册的。

在 A2A 协议中, 任务对象 (Task object) 是工作的基本单元, 由客户端提供的唯一 taskId 来标识。

3.4 A2A 中的安全考虑

安全性是 A2A 协议设计的核心要素之一, 旨在应对多 Agent 环境中可能出现的各种安全威胁, 比如身份冒充、数据泄露、任务被篡改以及未经授权的权限提升等。协议通过以下机制来增强安全性:

- **基于 HTTPS 的通信:** 所有的 A2A 通信, 包括获取“Agent 名片”和交换任务信息, 都强制使用 HTTPS 协议。这确保了数据在传输过程中的加密和完整性, 防止被窃听或篡改。
- **身份验证:** A2A 利用“Agent 名片”明确声明其支持的身份验证方法, 例如常用的 OAuth 2.0/OpenID Connect (OIDC) 配合 JSON Web Tokens (JWTs)。客户端 Agent 必须按照“Agent 名片”中指定的要求成功进行身份验证后, 才能与远程 Agent 进行交互。
- **签名机制:**
 - **签名的 Agent 名片 (Signed Agent Cards):** 为了防止“Agent 名片”被篡改或伪造, 建议使用数字签名(例如通过受信任的证书颁发机构 CA 签发的证书)来确保其真实性和完整性。
 - **JSON Web Signatures (JWS):** 可以用来保护消息内容和任务产物 (工件)的完整性, 确保它们在传输过程中没有被修改。
- **授权:** 虽然协议本身可能不直接规定具体的授权逻辑 (即判断一个通过身份验证的

Agent 是否有权限执行某个操作), 但其设计支持基于身份的访问控制。远程 Agent 应该根据客户端 Agent 的身份和它所拥有的权限, 来决定是否执行其请求的任务。

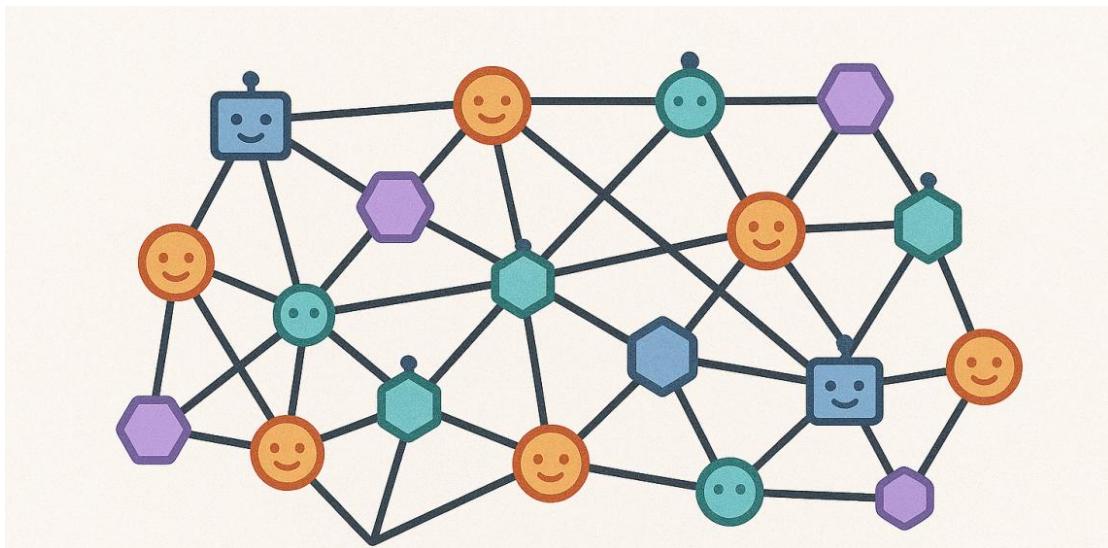
- **安全的通知渠道:** 对于推送通知功能, 需要对客户端提供的 Webhook URL 进行验证, 以防止服务器端请求伪造 (SSRF) 这类常见的网络攻击。同时, 建议对发送到 Webhook 的通知内容进行签名验证, 以确保其来源可靠且内容未被篡改。
- **威胁建模与缓解:** A2A 的设计考虑了多种潜在的安全威胁, 并鼓励开发者采用安全开发方法论和架构最佳实践来构建有弹性且有效的 A2A 系统。

通过这些内置的安全措施和设计上的周全考虑, A2A 协议旨在为构建可信赖的、安全的下一代 AI Agent 应用提供坚实的基础。

4. ANP 协议 (Agent Network Protocol)

4.1 ANP 是什么? 它想做什么?

Agent 网络协议 (ANP) 是一项旨在支持在一个开放的网络环境中, 让 AI Agent 能够相互发现并安全协作的协议。它的核心目标是创建一个开放、安全且高效的协作网络, 连接海量的、来自不同提供商、结构也可能各不相同的 AI Agent, 形成一个所谓的“Agent 互联网” (Internet of Agents)。ANP 特别关注解决当前互联网主要为人类交互而优化, 而非为自主 AI Agent 优化的局限性。它致力于满足 AI Agent 对低延迟通信、原生 API 接口以及去中心化身份验证的需求。通过标准化 AI Agent 之间的连接机制, ANP 希望打破数据孤岛, 确保 AI 能够访问到完整的上下文信息, 从而促进更深层次的协作和集体智能的涌现。



4.2 ANP 的核心理念与分层设计

ANP 的构建基于三大核心原则, 指引着它的发展方向:

- **互联互通 (Interconnectivity):** 目标是让所有的 AI Agent 都能相互通信, 打破信息壁垒, 确保 AI 在做决策时能够获取到完整的背景信息。
- **原生接口 (Native Interfaces):** 允许 AI Agent 通过专门为机器间通信优化的 API 和协议与数字世界进行交互, 而不是受限于为人类交互设计的界面和习惯。
- **高效协作 (Efficient Collaboration):** 通过利用自动组织和自动协商的机制, 使 AI Agent 能够建立起更具成本效益和更高效率的协作网络。

为了实现这些原则, ANP 采用了一种分层架构, 就像搭积木一样, 每一层都有特定的功能:

- **身份与加密通信层 (Identity and Encrypted Communication Layer):** 这一层利用了万维网联盟 (W3C) 的去中心化标识符 (DID) 标准来建立一套去中心化的身份验证机制。这意味着 AI Agent 可以拥有一个不由任何单一机构控制的、可验证的身份。

这确保了在不需要预先信任对方的情况下，也能进行端到端的加密通信，允许跨不同平台的 AI Agent 安全地相互验证身份。

- **元协议层 (Meta-Protocol Layer):** 这一层扮演着“协议的协议”的角色。它能让 AI Agent 使用自然语言（比如通过一个叫做 Agora 的系统）自主地协商和协调它们之间要使用的通信协议。这支持了通信协议的动态适应能力，以满足不同互动场景的需求。
- **应用协议层 (Application Protocol Layer):** 这一层负责定义标准化的协议，用于规范以下几个方面：
 - 互联网上的其他 AI Agent 如何发现某个 AI Agent。
 - 如何描述这些 AI Agent 所能提供的信息、拥有的能力以及可用的接口。
 - 用于完成特定领域任务的应用级协议。

这种分层设计为 ANP 提供了灵活性和可扩展性，使其能够适应未来 AI Agent 网络的多样化需求。

4.3 ANP 中的“身份证”与“寻路指南”

ANP 在管理 AI Agent 身份和帮助它们相互发现方面，采用了一种去中心化和语义化的方法：

- **去中心化标识符 (DIDs):** ANP 的核心是使用 W3C (万维网联盟) 推荐的去中心化标识符 (DIDs) 标准来管理 AI Agent 的“数字身份证”。具体来说，它可能会采用像 did:wba 这样特定的 DID 方法。DIDs 允许每个 AI Agent 拥有一个可验证的、全球唯一的身份，而不需要依赖某个中心化的身份认证机构。这为 AI Agent 之间建立信任和进行安全认证打下了坚实的基础。
- **JSON-LD 图谱:** 为了描述一个 AI Agent 都具备哪些能力、能提供哪些服务以及如何与它交互，ANP 利用了 JSON-LD（一种用于链接数据的 JSON 格式）图谱。JSON-LD 允许以一种结构化并且语义丰富的方式来表达数据，使得 AI Agent 能够更好地理解彼此的功能。这些包含 JSON-LD 图谱的描述文件，对于 AI Agent 的发现至关重要。
- **发现流程:** 当一个本地的 AI Agent 想要寻找其他 AI Agent 时，它会首先通过一个标准化的发现路径（比如一个预先定义好的网络端点，或者一个分布式的哈希表）来查找其他 AI Agent 的列表。然后，它会访问这个列表中引用的 AI Agent 描述文件，解析其中包含的 JSON-LD 等信息，从而了解目标 AI Agent 都具备哪些功能以及如何与它进行交互。

通过结合使用 DIDs 和 JSON-LD，ANP 旨在实现一个开放且可互操作的 AI Agent 发现机制，使得 AI Agent 能够在广阔的网络中找到并理解彼此。

4.4 ANP 如何保障安全与协作？

ANP 通过其架构设计和采用的技术来保障 AI Agent 之间安全地协作：

- **安全的身份与通信:**
 - **HTTPS 托管的 DIDs:** DID 文档（包含了 AI Agent 身份信息的文件）通常通过 HTTPS（安全的网页传输协议）来提供，确保其在被检索过程中的安全。
 - **DID 签名验证:** AI Agent 之间的身份验证依赖于对与 DID 相关的数字签名进行加密验证，以确保通信对方的真实性，防止假冒。
 - **接口的加密签名:** AI Agent 提供的接口和服务描述也可以进行加密签名，以防止被篡改，并确保其来源可靠。
 - **端到端加密通信:** ANP 的身份与加密通信层旨在实现无需预先信任的端到端加密通信，保护通信内容不被窃听。
- **协作与协商:**

- **元协议协商器 (Meta-Protocol Negotiator):** ANP 的元协议层允许 AI Agent 动态地协商双方都兼容的交互协议。这意味着，如果两个 AI Agent 最初使用的通信协议不完全匹配，它们可以通过一个更高层面的协商过程来确定一种共同的通信方式或参数。这对于促进不同类型的 AI Agent 之间的协作至关重要，并为解决冲突（例如，关于数据格式或交互顺序的分歧）提供了一种潜在的机制——通过协商达成一致。
- **自动组织与协商:** ANP 强调自动组织和自动协商机制，以实现更高效的协作网络。这表明协议的设计目标是让 AI Agent 能够自主地形成协作关系，并解决协作过程中出现的问题，而不需要太多人工干预。

通过这些机制，ANP 力求在开放的网络环境中实现既安全又灵活的多 AI Agent 协作。

5. CAMEL 的 MCP

本章内容需要更新到 CAMEL 的最新版本。

```
pip install "camel-ai[all]==0.2.71a9"
```

为了增强其智能体与外部工具和环境交互的能力，并融入更广泛的 AI 生态系统，CAMEL 集成了模型上下文协议 (MCP)。通过支持 MCP，CAMEL 智能体可以利用标准化的方式发现和调用外部工具，这不仅简化了工具集成的过程，还提高了系统的模块化程度和可扩展性。这种集成使得 CAMEL 能够更好地拥抱 MCP 生态系统，并利用其提供的标准化接口与各种兼容 MCP 的服务器和服务进行交互。

1. CAMEL 中的 MCP 模块的实现原理

camel.toolkits.mcp_toolkit.MCPToolkit 为 CAMEL 框架提供了“一站式”接入多个 MCP Server 的能力：

1. 支持 本地 STDIO 进程、Streamable-HTTP、SSE 传输方式
2. 自动探测服务器类型，统一包装为 CAMEL 的 FunctionTool
3. 用 AsyncExitStack 统一管理多条连接生命周期

流程说明

1. Toolkit 初始化：应用端创建 MCPToolkit，传入配置文件，根据配置文件，对每一个 MCP Server 都生成一个 Client
2. 连接阶段
 1. Toolkit 使用 AsyncExitStack 顺序进入每个 MCPClient 的 async-context
 2. MCPClient 根据 ServerConfig 自动选择传输层并建立 ClientSession
 3. 建链后立即调用 list_tools，缓存工具元数据
3. 工具聚合：MCPToolkit.get_tools() 遍历所有 Client，收集并严格化 schema
4. 工具调用：call_tool() 根据名称找到对应 Client 并把请求透传到 Server
5. 断开连接：Toolkit 退出时，AsyncExitStack 负责依次关闭所有 Client 及底层 transport

所以可以发现，在 CAMEL 中会将 MCP 中的工具转换成 CAMEL 中可用的 FunctionTool，并在最终生成符合 OpenAI strict-schema 的 JSON 描述。

2. 如何在 CAMEL 中使用 MCP

在 CAMEL 框架中使用 MCP 工具，主要涉及到配置 MCP 服务器、利用 MCPToolkit 进行连接和工具管理，并通过 CAMEL 智能体调用这些工具。

5.1.1 MCP 服务器配置

CAMEL 通过一个 JSON 配置文件（例如，mcp_servers_config.json）来指定需要连接的 MCP 服务器。该文件定义了每个服务器的启动命令、参数以及可能需要的环境变量。以下是如何将 CAMEL-AI 的智能体与一个 Time MCP 服务器进行连接的步骤：

```
{
  "mcpServers": {
    "time": { // 服务端标识名
      "command": "uvx", // 启动服务器的命令
      "args": ["mcp-server-time", "--local-timezone=Asia/Riyadh"] // 命令参数
    }
  }
}
```

此配置文件的灵活性允许用户集成各种现有的或自定义的 MCP 服务器，为 CAMEL 智能体提供广泛的外部能力。

5.1.2 CAMEL MCPToolkit

CAMEL 框架中的 MCPToolkit 是与 MCP 服务器交互的核心组件。它负责读取配置文件、管理与一个或多个 MCP 服务器的连接，并从这些服务器中聚合可用的工具，将其转换为 CAMEL 智能体可以使用的格式。

```
import asyncio
from camel.toolkits.mcp_toolkit import MCPToolkit
from camel.agents import ChatAgent
# 确保已定义 ChatAgent 所需的 model 对象
# from camel.models import ModelFactory, ModelPlatformType
#     model      =      ModelFactory.create(model_platform=ModelPlatformType.OPENAI,
model_type="gpt-4o-mini")

async def run_mcp_client_example():
    # 1. 初始化 MCPToolkit，指定配置文件路径
    mcp_toolkit = MCPToolkit(config_path="config/time.json") # 假设配置文件位于 config/time.json

    # 2. 异步连接到配置文件中定义的所有 MCP 服务器
    await mcp_toolkit.connect()

    # 3. 创建 CAMEL Agent 实例，并将从 MCPToolkit 获取的工具列表传递给 Agent
    camel_agent = ChatAgent(
        model=model, # Agent 使用的语言模型
        tools=[*mcp_toolkit.get_tools()], # 动态获取并解包 MCP 工具列表
    )

    # 4. Agent 执行任务，可能调用 MCP 工具
    response = await camel_agent.astep("What time is it now?")
    print(f"Agent response: {response.msgs[0].content}")
    if response.info and 'tool_calls' in response.info:
        print(f"Tool calls: {response.info['tool_calls']}")

    # 5. 任务完成后，断开与所有 MCP 服务器的连接
    await mcp_toolkit.disconnect()
```

```
# asyncio.run(run_mcp_client_example()) # 实际运行时取消注释  
预期输出示例：  
Agent response: The current local time in Riyadh is 15:57 (3:57 PM).  
Tool calls: [ToolCallingRecord(tool_name='get_current_time', args={'timezone': 'Asia/Riyadh'},  
result='{\n    "timezone": "Asia/Riyadh",\n    "datetime": "2025-05-01T15:57:59+03:00",\n    "is_dst": false\n}', tool_call_id='...')]
```

通过 MCPToolkit, CAMEL 极大地简化了在多智能体环境中使用标准化外部工具的过程。开发者无需直接处理 JSON-RPC 消息或 MCP 的底层连接细节，而是可以通过高级 API 方便地集成和使用 MCP 工具。这种抽象能力显著降低了采用 MCP 等标准化协议的门槛，使得开发者可以更专注于智能体的核心逻辑和应用场景，而不是协议本身的复杂性。

5.1.3 代码示例：CAMEL Agent 使用 MCP 工具

下面的 Python 代码将向你展示，如何配置一个 CAMEL ChatAgent（聊天代理），让它能够使用通过 MCP 协议提供的外部工具。这个例子假设我们已经准备好了名为 config/mcp_time_config.json 的 MCP 服务器配置文件，这个文件里写着如何启动一个本地的时间服务。

Python

```
import asyncio  
import os  
import json  
from camel.models import ModelFactory  
from camel.types import ModelPlatformType, ModelType  
from camel.toolkits import MCPToolkit  
from camel.agents import ChatAgent  
  
# from camel.logger import set_log_level # 如果需要更详细日志，可以取消这行注释#  
set_log_level(level="DEBUG")# 重要提示：运行此代码前，请确保你已经设置了  
OPENAI_API_KEY 环境变量# 例如：export OPENAI_API_KEY='你的 OpenAI API 密钥'# 或者  
在代码中安全地提供它。  
async def run_mcp_example_with_camel_agent:# 1. 配置 AI 模型  
(我们这里用 OpenAI 的 GPT-4o 作为例子)# CAMEL 可以让你方便地创建和使用不同平台和  
类型的 AI 模型# 请确保 OPENAI_API_KEY 环境变量已设置，否则模型无法工作  
    model = ModelFactory.create(  
        model_platform=ModelPlatformType.OPENAI, # 指定使用 OpenAI 平台  
        model_type=ModelType.GPT_4O, # 指定使用 GPT-4o 模型  
    )  
    # 2. 初始化 MCPToolkit，告诉它我们的 MCP 服务器配置文件在哪里# 这个配置文件  
(mcp_time_config.json) 告诉 CAMEL 如何启动时间服务器。# 配置文件内容示例 (需要保存  
为 config/mcp_time_config.json):# {#     "mcpServers": {#         "time": {#             "command":  
"uvx",#                 "args": ["mcp-server-time", "--local-  
timezone=America/New_York"]#             }# }# 请根据你的实际文件路径调整下面的  
config_file_path  
    config_file_path = "config/mcp_time_config.json"  
    mcp_toolkit = MCPToolkit(config_path=config_file_path)  
try:# 3. 连接到配置文件中定义的所有 MCP 服务器  
    print("正在尝试连接到 MCP 服务器...")await mcp_toolkit.connect # 这是一个异步
```

```

操作, 所以用 await
    print("成功连接到 MCP 服务器! ")
# 4. 从 MCPToolkit 获取所有可用的工具
    available_tools = mcp_toolkit.get_tools() if not available_tools:
        print("没有从 MCP 服务器找到可用的工具。请检查你的配置文件和服务器状态。") return
        print(f"通过 MCP 发现的可用工具: {[tool.name for tool in available_tools]}")
# 5. 创建一个 CAMEL 聊天代理 (ChatAgent), 并把从 MCP 获取的工具交给它
    print("正在创建一个配备了 MCP 工具的 CAMEL 代理...")
    camel_agent = ChatAgent(
        model=model, # 告诉代理使用我们之前配置的 AI 模型
        tools=[*available_tools], # 把 MCP 工具列表交给代理
    )
    print("CAMEL 代理创建完成。")
# 6. 现在我们可以和这个代理互动了, 它在需要的时候会自动使用 MCP 工具
    task_prompt = "请问现在纽约是什么时间? " # 这是一个需要用到时间工具的问题
    print(f"向代理发送任务: {task_prompt}")
    response = await camel_agent.astep(task_prompt) # astep 表示异步执行一步对话
# 7. 打印代理的回复和它调用工具的信息 if response and response.msgs:
    print("\n 代理的最终回复:")
    print(response.msgs[-1].content) # 通常最后一条消息是最终答案# 检查代理是否调用了工具, 并打印相关信息 if response.info and 'tool_calls' in response.info and response.info['tool_calls']:
        print("\n 代理在此过程中进行的工具调用:") for tool_call in response.info['tool_calls']: # tool_call 的具体结构可能因 CAMEL 版本而异# 通常包含工具名称、传入的参数和工具返回的输出
            print(f"  工具名称: {tool_call.get('name', '未知工具')}")
            print(f"  调用参数: {tool_call.get('args', {})}")
            print(f"  工具输出: {tool_call.get('outputs', '无输出')}") else:
                print("在此步骤中, 代理没有报告工具调用。") else:
                print("未能从代理获取响应。")
except Exception as e:
    print(f"在执行过程中发生了错误: {e}") finally:# 8. 无论成功还是失败, 最后都要断开与 MCP 服务器的连接, 释放资源
    print("\n 正在断开与 MCP 服务器的连接...") await mcp_toolkit.disconnect
    print("已成功断开连接。")

if name == "__main__":# 这部分代码是为了方便你运行示例: # 它会检查配置文件是否存在, 如果不存在, 会创建一个示例配置文件。# 它还会检查 OpenAI API 密钥是否设置。
    config_dir = "config"
    config_file = os.path.join(config_dir, "mcp_time_config.json") if not os.path.exists(config_dir):
        os.makedirs(config_dir) if not os.path.exists(config_file): with open(config_file, "w") as f:

```

```

        json.dump({"mcpServers": {"time": {# 注意: 要运行这个时间服务器, 你需要先
安装 'uvx' 和 'mcp-server-time'# 可以尝试用 pip install universal-unix-devserver mcp-
server-time 来 安 装 "command": "uvx", "args": ["mcp-server-time", "--local-
timezone=America/New_York"]}
    }
},
}, f, indent=2)
print(f"已为你创建了一个示例配置文件: {config_file}")
print("请确保你已经安装了 'uvx' 和 'mcp-server-time' 并且它们在你的系统路径
中。")
print("你可以尝试使用以下命令安装它们: pip install universal-unix-devserver
mcp-server-time")
if not os.getenv("OPENAI_API_KEY"):
    print("警告: OPENAI_API_KEY 环境变量尚未设置。")
    print("此示例需要有效的 OpenAI API 密钥才能与 AI 模型交互。请先设置该环境变
量。")# 如果没有 API 密钥, 可以选择退出或提示用户输入# exit
    asyncio.run(run_mcp_example_with_camel_agent)

```

代码解释: 这段代码完整地演示了在 CAMEL 框架中使用 MCP 工具的流程。

1. **模型初始化:** 我们首先使用 ModelFactory 创建了一个 AI 模型实例（这里是 OpenAI 的 GPT-4o）。要实际运行，你需要确保相关的 API 密钥已经正确配置（比如通过设置名为 OPENAI_API_KEY 的环境变量）。
2. **MCP Toolkit 初始化:** MCPToolkit 通过 config_path 参数加载了 MCP 服务器的配置信息。在我们的例子中，它指向 config/mcp_time_config.json 文件，这个文件里应该包含了启动 MCP 时间服务器所需的指令。
3. **连接服务器:** await mcp_toolkit.connect 这行代码会异步地（不会阻塞主程序）连接到配置文件中定义的所有 MCP 服务器。
4. **获取工具:** mcp_toolkit.get_tools 会返回一个列表，里面是从所有已连接的 MCP 服务器获取到的、并且已经被 CAMEL 封装好的工具对象。
5. **代理创建:** 我们创建了一个 ChatAgent（聊天代理）的实例，并把从 MCPToolkit 获取到的工具列表传递给它。这样，这个代理在后续的思考和行动中，就可以使用这些 MCP 工具了。
6. **任务执行:** await camel_agent.astep(task_prompt) 向代理发送一个任务提示（比如“现在纽约几点钟？”）。如果代理判断需要使用某个工具来回答这个问题（例如，调用时间服务器获取当前时间），它就会通过 MCP 协议调用相应的工具。
7. **响应处理:** 代码会打印出代理的最终回复内容，以及它在执行任务过程中可能调用过的工具的详细信息（比如调用了哪个工具，用了什么参数，得到了什么结果）。
8. **断开连接:** 在 finally 代码块中（这意味着无论前面的代码是否出错，这部分总会执行），await mcp_toolkit.disconnect 确保与 MCP 服务器的连接被正确关闭，从而释放占用的资源。

脚本的 if name == "__main__":部分包含了一些辅助功能：它会检查配置文件是否存在，如果不存在，就创建一个示例配置文件，并提示用户安装必要的依赖库(uvx和mcp-server-time)。同时，它也会检查 OPENAI_API_KEY 环境变量是否已设置，因为与 AI 模型交互通常需要这个密钥。

这个例子清晰地展示了 CAMEL 框架如何通过 MCPToolkit 这个“工具箱管理员”来简化 MCP 协议的复杂性，为开发者提供了一个简洁的接口来利用标准化的外部工具。这无疑增强了 LLM 代理的功能和它们与其他系统协同工作的能力。

3. 如何将 CAMEL 作为 MCP 服务器

除了能把工具包转换成 MCP 服务器外，CAMEL 框架还支持一个更强大的功能：把整个 CAMEL 代理（Agent）本身也作为 MCP 服务器发布出去。这意味着，一个你精心配置好的 CAMEL 代理——它可能拥有特定的 AI 模型、独特的系统指令（system message），甚至内置了一些其他的工具——现在可以作为一个独立的服务运行。然后，任何支持 MCP 协议的客户端（比如像 Anthropic 公司开发的 Claude 桌面应用、Cursor 代码编辑器，或者其他同样支持 MCP 的 AI 代理框架）都可以通过 MCP 协议向你的这个 CAMEL 代理发送请求，并获得它的响应。

想象一下，你在 CAMEL 生态系统中开发了一个非常专业的 AI 代理，它可能利用了 CAMEL 的一些独特功能（比如复杂的角色扮演对话、大规模的模拟推演，或者针对某个特定任务进行了精细的调整）。现在，通过把它作为 MCP 服务器发布，这个专业的代理就能无缝地作为一个“组件”被集成到更广泛的、遵循 MCP 标准的、由不同类型 AI 代理组成的复杂系统中。这有效地在 CAMEL 与其他兼容 MCP 的框架和工具之间架起了一座桥梁，促进了更高层次的互操作性——这不仅仅是工具层面的共享，而是 AI 代理服务层面的集成。

5.3.1 将 CAMEL Agent 暴露为 MCP 服务

将 CAMEL 代理暴露为 MCP 服务，实际上就是把这个代理的交互能力（比如，它对特定提示做出响应的能力、执行一系列思考步骤的能力，或者完成特定任务的能力）封装起来，并作为 MCP 服务器上的一个或多个“工具”或“服务接口”提供出去。当一个 MCP 客户端连接到这个由 CAMEL 代理驱动的 MCP 服务器时，它就可以像调用任何其他普通的 MCP 工具一样来调用这个 CAMEL 代理了。

从 CAMEL 的文档和发展方向来看，其内部一些更复杂的模块，比如用于实现多代理角色扮演（RolePlaying）的模块，或者用于组织多个代理协同工作的“工作组”（Workforce）模块，也计划或者正在被实现为 MCP 服务器。如果这些功能也通过 MCP 开放出来，那么外部系统就能以一种标准化的方式，与 CAMEL 中那些精心设计的、复杂的多代理交互场景进行互动了。

此外，CAMEL 框架中有一个核心的设计哲学叫做“代码即提示”（Code-as-Prompt）。它强调编写既清晰易懂（对人类开发者而言）又能被 AI 代理准确理解的代码。这种理念与 MCP 协议所倡导的结构化、明确定义的接口设计不谋而合。因此，习惯了 CAMEL 设计原则的开发者，在定义和使用 MCP 工具（包括将代理本身作为 MCP 工具）时，可能会感到更加自然和顺手，这也有助于提升集成的质量和稳定性。

5.3.2 代码示例与配置

下面，我们将提供一个概念性的代码示例和配置说明，来演示如何定义一个 CAMEL 代理，并把它作为 MCP 服务器来运行。需要注意的是，CAMEL 框架本身在不断发展和完善中，将代理作为 MCP 服务器运行的确切 API 接口和具体工具可能会随着版本更新而有所变化。因此，强烈建议你查阅最新版的 CAMEL 官方文档，以获取最准确的实现细节。

1. 定义你的 CAMEL 代理

这个脚本的作用是定义你希望作为 MCP 服务暴露出去的 CAMEL 代理。

```
from camel.agents import ChatAgent
from camel.models import ModelFactory, ModelPlatformType
from camel.toolkits import FunctionTool, SearchToolkit # 假设 SearchToolkit 已定义
```

```
# 示例：定义不同功能的 Agent
chat_agent
ChatAgent(model=ModelFactory.create(model_platform=ModelPlatformType.OPENAI,
model_type="gpt-3.5-turbo"))
chat_agent_description = "A general-purpose assistant." =
```

```
reasoning_agent = ChatAgent(
    model=ModelFactory.create(model_platform=ModelPlatformType.OPENAI,
model_type="gpt-4o-mini"))
reasoning_agent_description = "A specialized assistant for logical reasoning."
```

```
# (后续步骤通常涉及一个服务脚本，例如 agent_mcp_server.py,
# 该脚本负责加载这些 Agent 定义，并以 MCP 服务器模式运行它们。)
通过这种方式，这些 Agent 的能力可以通过 MCP 协议被外部系统调用。
```

2. 将 CAMEL 工具包 (Toolkit) 发布为 MCP 服务

任何 CAMEL-AI 工具包都可以通过内置方法启动为一个 MCP 服务器。以下是将 ArxivToolkit 转换为 MCP 服务器的示例脚本 (arxiv_toolkit_mcp_server.py):

```
# arxiv_toolkit_mcp_server.py
import argparse
import sys
from camel.toolkits import ArxivToolkit # 确保 ArxivToolkit 已定义

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Run Arxiv Toolkit in MCP server mode.")
    parser.add_argument("--mode", choices=["stdio", "sse"], default="stdio", help="MCP
server communication mode.")
    parser.add_argument("--timeout", type=float, default=None, help="Timeout for the
MCP server.")
    args = parser.parse_args()

    toolkit = ArxivToolkit(timeout=args.timeout)
    toolkit.run_mcp_server(mode=args.mode) # 启动 MCP 服务器
```

执行此脚本 (e.g., python arxiv_toolkit_mcp_server.py) 后，ArxivToolkit 的功能即可通过 MCP 协议访问。

当然，你可以定义自己的工具，下面有一个最基础的演示，怎么把一个函数变成模型可调用的工具：

```
from camel.toolkits import FunctionTool
from camel.agents import ChatAgent # 假设你还需要引入 ChatAgent
```

```
def my_weird_add(a: int, b: int) -> int:
    return a + b + 7

agent = ChatAgent(tools=[FunctionTool(my_weird_add)])
response = agent.step("What is 15 + 15")
```

```
print(response.msgs[0].content)
```

```
>>> 37
```

3. MCP 客户端配置

当你的 CAMEL 代理 MCP 服务器成功运行后，其他的 MCP 客户端（比如 Anthropic 的 Claude 桌面应用或 Cursor 代码编辑器）就需要进行配置，才能连接到你的这个服务器。这通常涉及到在客户端的 MCP 配置文件中，指定如何启动你的服务器以及传递必要的参数。

参考中给出的客户端配置示例，一个概念性的配置可能看起来像这样（通常是一个 JSON 文件）（用于 Claude Desktop）：

```
"camel-chat-agent": {  
    "command": "/path/to/python_interpreter",  
    "args": ["/path/to/camel/services/agent_mcp_server.py"], // 启动 CAMEL Agent MCP 服  
   务器的脚本  
    "env": { // 所需环境变量  
        "OPENAI_API_KEY": "YOUR_API_KEY",  
        "BRAVE_API_KEY": "YOUR_API_KEY"  
    }  
}
```

此配置允许 Claude Desktop 将请求路由到指定的 CAMEL Agent MCP 服务器。

4. OWL 框架与 MCP 的集成

OWL (Objective-driven Working Language) 是一个构建于 CAMEL-AI 之上的多智能体协作框架。OWL 智能体同样支持通过 MCP 与外部工具和服务进行集成，扩展了其在复杂任务自动化中的应用能力。

CAMEL 官方推出了 **MCP Hub** (mcp.camel-ai.org)，这是一个集中的 MCP 服务器目录平台。

- **功能**：提供可视化工具卡片、按场景分类、官方与社区工具筛选。
- **便捷性**：提供一键复制 JSON 配置功能，简化将 MCP 工具集成到 CAMEL Agent 的过程。

下面提供两个案例：

1. MCPSearchToolkit

MCPSearchToolkit 允许开发者在代码中动态搜索和发现可用的 MCP 服务器。

```
from camel.toolkits.mcp_search import PulseMCPSearchToolkit # 示例：使用 Pulse 的注册  
中心
```

```
search_toolkit = PulseMCPSearchToolkit()  
server_info = search_toolkit.search_mcp_servers(  
    query="Slack",  
    package_registry="npm", # 可选：指定包注册表  
    top_k=1  
)  
print(server_info)
```

该工具返回 MCP 服务器的元数据，包括功能描述、源链接等，辅助开发者进行工具选择和集成。

2. MCPAgent

MCPAgent 是一种 CAMEL 智能体，它能够根据用户查询动态搜索、配置并利用 MCP 服务器执行任务。

```
import os
from camel.agents import MCPAgent, MCPRegistryConfig, MCPRegistryType
# from camel.models import ModelFactory, ModelPlatformType # 确保 model 已定义
#     model      =      ModelFactory.create(model_platform=ModelPlatformType.OPENAI,
model_type="gpt-4o-mini")

# 配置 MCP 注册中心信息 (示例使用 Smithery)
smithery_config = MCPRegistryConfig(
    type=MCPRegistryType.SMITHERY,
    api_key=os.getenv("SMITHERY_API_KEY")
)

# 创建 MCPAgent 实例
agent = MCPAgent(
    model=model,
    registry_configs=[smithery_config] # 传递注册中心配置列表
)

async def run_mcp_agent_example():
    async with agent: # 使用异步上下文管理器
        response = await agent.astep("Use Brave API to search info about CAMEL-AI.org")
        print(response.msgs[0].content)

# asyncio.run(run_mcp_agent_example()) # 实际运行时取消注释
```

MCPAgent 能够自主查询注册中心，找到合适的 MCP 服务器（如 Brave 搜索服务），并调用该服务以完成指定任务。

结论

LLM Agent 的飞速发展为人工智能开辟了激动人心的新可能性，但同时也让我们迫切需要一套标准化的方式来让它们进行交互。本文对模型上下文协议（MCP）、代理间协议（A2A）和代理网络协议（ANP）这些新兴的“沟通规则”进行了技术性的探讨，希望能让你对它们有一个初步的了解。

简单来说：

- **MCP** 就像是 AI 代理的“USB-C 接口”，它提供了一套标准化的上下文信息和工具接口，解决了 LLM 代理与外部环境（比如各种工具和数据源）连接时的“最后一公里”问题，使得工具的集成可以像“即插即用”一样方便。
- **A2A 协议** 则更侧重于 AI 代理之间的直接“对话”和任务协作。通过像“代理名片”（Agent Card）这样的机制，它促进了在企业级应用中，AI 代理能够发现彼此的能力，并有效地将任务分配给合适的代理去完成。
- **ANP 协议** 则展望了一个更为宏大的“代理互联网”蓝图。它希望利用像去中心化身份标识（DIDs）和语义化描述（JSON-LD）这样的技术，来支持在一个开放的网络环境中，大规模的 AI 代理能够安全地被发现并进行协作。

这些协议共同致力于解决当前 LLM 代理生态系统中普遍存在的一些关键挑战，比如不同代理之间难以互通有无、集成外部工具的成本太高、系统难以扩展以及安全性不足等等。通过推动标准化，它们为构建一个更加健壮、灵活和智能的多代理未来奠定了基础，使得来自不同来源、拥有不同能力的 AI 代理能够有效地协同工作，从而释放出更大的集体智能潜力。展望未来，随着这些“沟通规则”的不断成熟和更多开发框架的采纳，我们可以预见到一个更加互联互通的 LLM 代理世界。一些正在进行的开发工作，比如用于托管和验证 MCP 服务器的“MCP Hub”（MCP 中心），以及将更复杂的代理交互模式（如角色扮演和工作组协作）封装为 MCP 服务，都预示着未来的多代理协作将达到新的高度，为解决现实世界中的复杂问题提供前所未有的强大能力。
