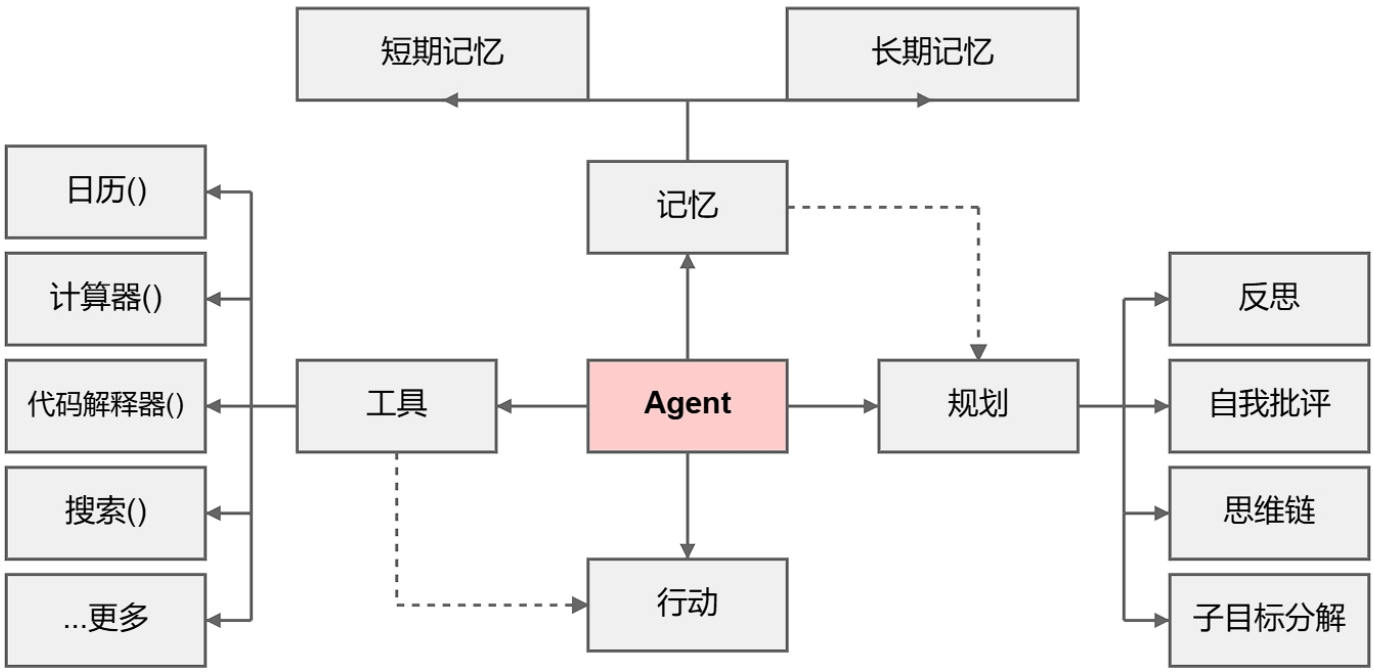
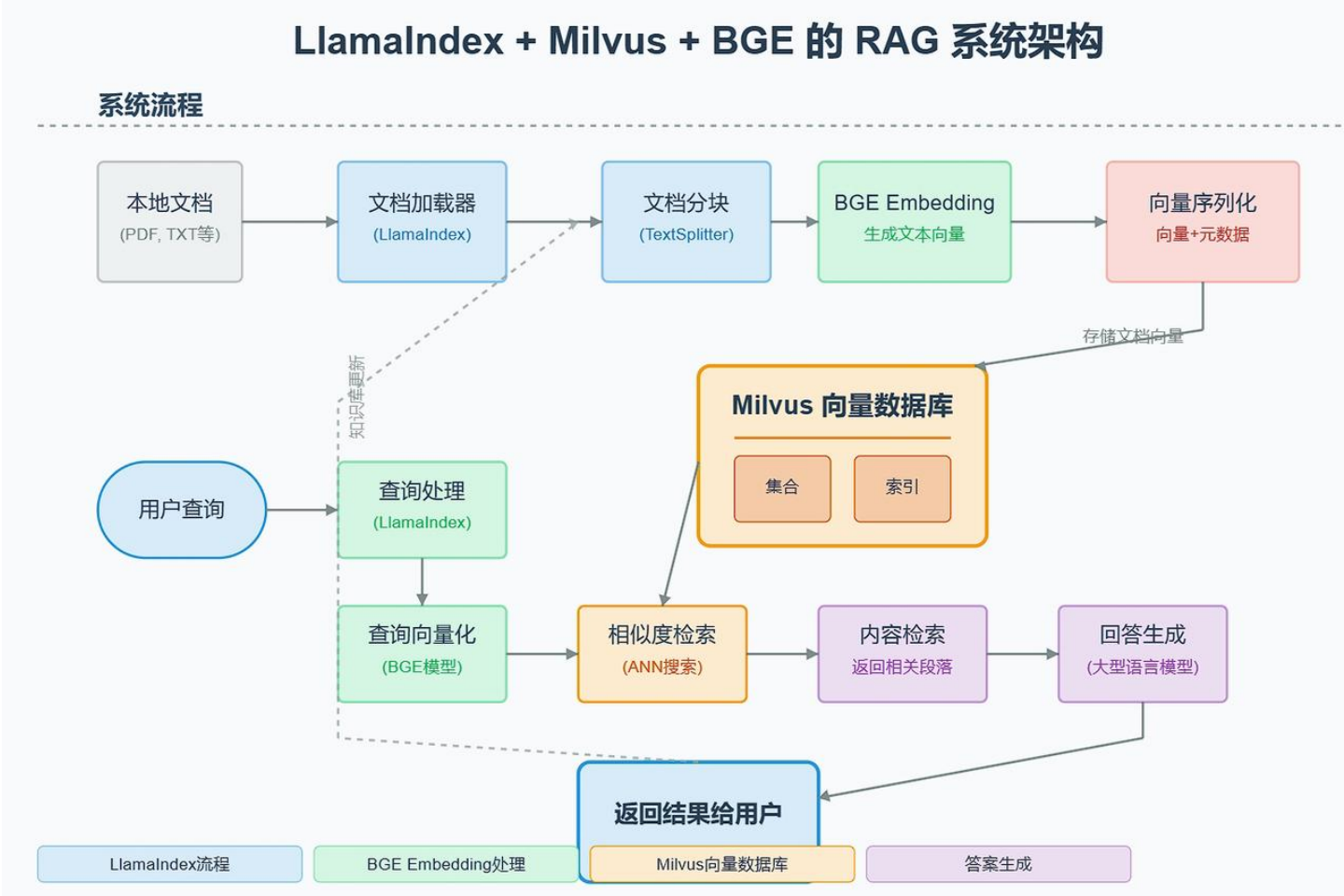
9. Agent +RAG

AI Agent 一般包含以下几个**关键模块**：

1. **计划模块（Planning）**
- 负责理解目标任务
- 拆解任务为可执行的步骤
- 生成行动计划
2. **记忆模块（Memory）**
- 存储与任务相关的历史记录、上下文信息
- 支持长期记忆与短期记忆
- 提高连续对话与任务一致性
3. **工具使用模块（Tools）**
- 能调用各种外部工具，如：
  - 网络搜索引擎
  - API 接口
  - 计算器、代码执行器
- 实现感知-行动闭环
4. **执行模块（Executor）**
- 根据计划执行每一步操作
- 反馈执行结果并调整策略（如果需要）



本地文档检索RAG



文档解析

利用数据解析云服务**llamacloud**（https://cloud.llamaindex.ai/），异步的解析文档。

代码块

```
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
import nest_asyncio

nest_asyncio.apply()

from llama_cloud_services import LlamaParse

parser = LlamaParse(
    api_key="YOUR_LLAMA_CLOUD_API_KEY",  # can also be set in your env as LLAMA_CLOUD_API_KEY
    result_type="markdown",  # "markdown" and "text" are available
    num_workers=3,  # if multiple files passed, split in `num_workers` API calls
    verbose=True,
    language="ch_sim",  # Optionally you can define a language, default=en
)

file_extractor = {".pdf": parser}
# 返回一个列表
```

```
documents_cloud = SimpleDirectoryReader(
    "./data", file_extractor=file_extractor
).load_data()
# 汇总所有文档
all_doc=''
for doc in documents_cloud:
    all_doc+=doc.text
```

切块

代码块

```
def chunk_text(text, max_chunk_size=300):
    """
    将文本切分成固定大小的块，保持段落和表格的完整性。

    Args:
        text (str): 需要切分的文本
        max_chunk_size (int): 每个块的最大大小，默认为300

    Returns:
        list: 包含切分后文本块的列表
    """
    # 按行分割文本
    lines = text.split('\n')

    chunks = []
    current_chunk = []
    current_size = 0
    in_table = False
    table_content = []

    for line in lines:
        # 检测表格开始（含有 | 字符的行，通常是表格）
        if '|' in line and not in_table:
            # 可能是表格的开始
            in_table = True

            # 保存当前块
            if current_chunk:
                chunks.append('\n'.join(current_chunk))
                current_chunk = []
                current_size = 0

            table_content.append(line)

```

```
        # 表格内容
        elif in_table:
            table_content.append(line)

            # 如果遇到空行，可能表示表格结束
            if not line.strip():
                in_table = False
                chunks.append('\n'.join(table_content))
                table_content = []

        # 普通内容
        else:
            line_length = len(line)

            # 如果加上这一行会超过最大块大小，并且当前块不为空，则保存当前块
            if current_size + line_length > max_chunk_size and current_chunk:
                chunks.append('\n'.join(current_chunk))
                current_chunk = []
                current_size = 0

            # 添加新行到当前块
            current_chunk.append(line)
            current_size += line_length

            # 如果当前行为空，可能是段落结束
            # 检查当前块大小，如果已经足够大，则保存当前块
            if not line.strip() and current_size > max_chunk_size // 2:
                chunks.append('\n'.join(current_chunk))
                current_chunk = []
                current_size = 0

    # 处理剩余的表格内容
    if in_table and table_content:
        chunks.append('\n'.join(table_content))

    # 处理最后剩余的块
    if current_chunk:
        chunks.append('\n'.join(current_chunk))

    return chunks
chunks = chunk_text(all_doc, max_chunk_size=300)
```

插入Milvus数据库

利用Milvus和BGE-M3 模型进行混合搜索。BGE-M3 模型可以将文本转换为密集向量和稀疏向量。Milvus 支持在一个 Collections 中存储这两种向量，从而可以进行混合搜索，增强搜索结果的相关性。

代码块

```
# 下载bge模型
import os
os.environ["HF_ENDPOINT"] = "https://hf-mirror.com"
from modelscope import snapshot_download
model_dir = snapshot_download('BAAI/bge-m3',
cache_dir='./', revision='master')

# 创建embedding模型
from milvus_model.hybrid import BGEM3EmbeddingFunction
ef = BGEM3EmbeddingFunction(model_name_or_path="./BAAI/bge-m3",
use_fp16=False, device="cpu")
dense_dim = ef.dim["dense"]
# 用bge-M3模型的embedding结果
chunks_embeddings = ef(chunks)
```

- 设置 Milvus Collections 和索引

代码块

```
from pymilvus import (
    connections,
    utility,
    FieldSchema,
    CollectionSchema,
    DataType,
    Collection,
)
# 连接milvus数据库
connections.connect(uri="./milvus.db")

fields = [
    # Use auto generated id as primary key, 类似数据库中的主键
    FieldSchema(
        name="pk", dtype=DataType.VARCHAR, is_primary=True, auto_id=True,
max_length=100
    ),
    # Store the original text to retrieve based on semantically distance
    FieldSchema(name="text", dtype=DataType.VARCHAR, max_length=512),
    # 支持稀疏检索和稠密检索
    # Milvus now supports both sparse and dense vectors,
    # we can store each in a separate field to conduct hybrid search on both
vectors
    FieldSchema(name="sparse_vector", dtype=DataType.SPARSE_FLOAT_VECTOR),
    FieldSchema(name="dense_vector", dtype=DataType.FLOAT_VECTOR,
dim=dense_dim),
```

```
]
schema = CollectionSchema(fields)

col_name = "hybrid_demo"
if utility.has_collection(col_name):
    Collection(col_name).drop()
col = Collection(col_name, schema, consistency_level="Strong")
# 创建索引
sparse_index = {"index_type": "SPARSE_INVERTED_INDEX", "metric_type": "IP"}
col.create_index("sparse_vector", sparse_index)
dense_index = {"index_type": "AUTOINDEX", "metric_type": "IP"}
col.create_index("dense_vector", dense_index)
col.load()

```

插入向量数据库

代码块

```
for i in range(0, len(chunks), 50):
    batched_entities = [
        chunks[i : i + 50],
        chunks_embeddings["sparse"][i : i + 50],
        chunks_embeddings["dense"][i : i + 50],
    ]
    col.insert(batched_entities)
print("Number of entities inserted:", col.num_entities)
```

检索

定义密集检索、稀疏检索和混合检索函数

代码块

```
from pymilvus import (
    AnnSearchRequest,
    WeightedRanker,
)

# 密集检索
def dense_search(col, query_dense_embedding, limit=10):
    search_params = {"metric_type": "IP", "params": {}}
    res = col.search(
        [query_dense_embedding],
        anns_field="dense_vector",
        limit=limit,
```

```
        output_fields=["text"],
        param=search_params,
    )[0]
    return [hit.get("text") for hit in res]

# 稀疏检索
def sparse_search(col, query_sparse_embedding, limit=10):
    search_params = {
        "metric_type": "IP",
        "params": {},
    }
    res = col.search(
        [query_sparse_embedding],
        anns_field="sparse_vector",
        limit=limit,
        output_fields=["text"],
        param=search_params,
    )[0]
    return [hit.get("text") for hit in res]

# 混合检索
def hybrid_search(
    col,
    query_dense_embedding,
    query_sparse_embedding,
    sparse_weight=1.0,
    dense_weight=1.0,
    limit=10,
):
    dense_search_params = {"metric_type": "IP", "params": {}}
    dense_req = AnnSearchRequest(
        [query_dense_embedding], "dense_vector", dense_search_params, limit=limit
    )
    sparse_search_params = {"metric_type": "IP", "params": {}}
    sparse_req = AnnSearchRequest(
        [query_sparse_embedding], "sparse_vector", sparse_search_params, limit=limit
    )
    rerank = WeightedRanker(sparse_weight, dense_weight)
    res = col.hybrid_search(
        [sparse_req, dense_req], rerank=rerank, limit=limit, output_fields=["text"]
    )[0]
    return [hit.get("text") for hit in res]

```

混合检索结合了密集检索和稀疏检索的优势，因此后续只采用混合检索的结果。

代码块

```
query = '什么是以人为本的座舱'
query_embeddings = ef([query])
dense_results = dense_search(col, query_embeddings["dense"][0])
sparse_results = sparse_search(col, query_embeddings["sparse"]._getrow(0))
hybrid_results = hybrid_search(
    col,
    query_embeddings["dense"][0],
    query_embeddings["sparse"]._getrow(0),
    sparse_weight=0.7,
    dense_weight=1.0,
)
print(dense_results[:3])
print(sparse_results[:3])
print(hybrid_results[:3])
```

对检索结果添加序号

代码块

```
def format_list_with_markers(input_list):
    """
    Formats a list by adding numbered markers ([1], [2], etc.) to each item
    and adding a line break after each item.

    Args:
        input_list: A list of strings to format

    Returns:
        A single formatted string
    """
    result = ""

    for index, item in enumerate(input_list, 1):
        # Add the numbered marker and the item content
        result += f"[{index}] {item}\n"

    return result
formatted_references=format_list_with_markers(hybrid_results)
pprint(formatted_references)
```

生成

定义用户prompt

```
prompt = f"""
你是一个智能助手，负责根据用户的问题和提供的参考内容生成回答。请严格按照以下要求生成回答：
1. 回答必须基于提供的参考内容。
2. 在回答中，每一块内容都必须标注引用的来源，格式为：[引用编号]。例如：[1] 表示引用自第1条参考内容。
3. 如果没有参考内容，请明确说明。如果没有参考知识，根据你自己的知识进行回答

参考内容：
{formatted_references}

用户问题：{query}
"""

print(prompt)
```

调用gpt-4o接口，将检索到的内容添加到用户prompt中。

```
from openai import OpenAI
client_gpt = OpenAI(api_key="your_openai_api_key", base_url="https://api.openai-proxy.org/v1")

completion_gpt = client_gpt.chat.completions.create(
    model="gpt-4o",
    messages=[
        {
            "role": "user",
            "content": prompt
        }
    ]
)

print(completion_gpt.choices[0].message.content)
```

封装成RAG接口

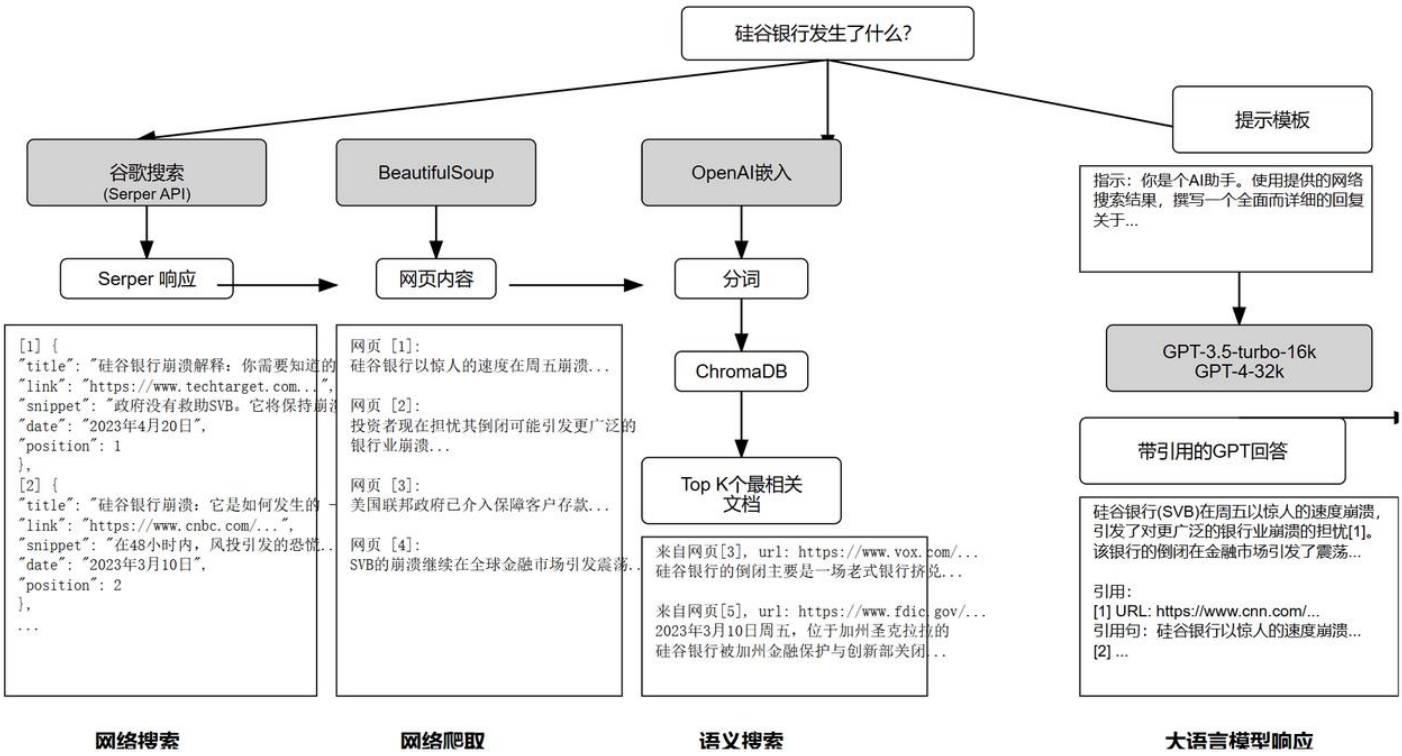
```
def retrieval(query):
    query_embeddings = ef([query])
```

```
    hybrid_results = hybrid_search(
        col,
        query_embeddings["dense"][0],
        query_embeddings["sparse"]._getrow(0),
        sparse_weight=0.7,
        dense_weight=1.0,
    )
    return hybrid_results
```

代码块

```
def rag(query):

    formatted_references=format_list_with_markers(retrieval(query))

    prompt = f"""
    你是一个智能助手，负责根据用户的问题和提供的参考内容生成回答。请严格按照以下要求生成回答：
    1. 回答必须基于提供的参考内容。
    2. 在回答中，每一块内容都必须标注引用的来源，格式为：[引用编号]。例如：[1] 表示引用自第1条参考内容。
    3. 如果没有参考内容，请明确说明。如果没有参考知识，根据你自己的知识进行回答

    参考内容：
    {formatted_references}

    用户问题：{query}
    """
    client = OpenAI(api_key="your_openai_api_key", base_url="https://api.openai-proxy.org/v1")

    completion = client.chat.completions.create(
        model="gpt-4o",
        messages=[
            {
                "role": "user",
                "content": prompt
            }
        ]
    )

    return formatted_references,completion.choices[0].message.content
reference,result=rag('介绍一下技术规格')
print(result)
```

网络检索



网络检索代码的实现在“websearch”文件夹中，这里只调用网络检索的接口。

代码块

```
from websearch.src.retrieval import web_retrieval
def web_search_answer(query):

    web_reference=web_retrieval(query)

    prompt = f"""

    Web 搜索结果:
    {web_reference}

    指令：你是一个/一名销售助手。使用提供的网络搜索结果，对给定的查询写一个全面而详细的回复。
    确保在引用后使用 [number] 标记引用。
    在回答的最后，列出带索引的相应参考文献，每个参考文献包含网络搜索结果中的网址和引用句子，按照您在上面标记的顺序排列，这些句子应该与网络搜索结果中的完全相同。
    以下是参考文献的示例:
        [1] 网址：https://www.pocketgamer.biz/news/81670/tencent-and-netease-dominated-among-chinas-top-developers-in-q1/
            引用句子：腾讯在本季度占据了国内市场收入的大约50%，相比之下2022年第一季度为40%。

    查询：{query}
```

```
19
20      请你过滤掉检索结果中的不相关性的部分进行回答，如果没有相关的部分，按照你自己的知识进行回答
21      """
22      client = OpenAI(api_key="your_openai_api_key", base_url="https://api.openai-proxy.org/v1")
23
24      completion = client.chat.completions.create(
25          model="gpt-4o",
26          messages=[
27              {
28                  "role": "user",
29                  "content": prompt
30              }
31          ]
32      )
33
34
35      return completion.choices[0].message.content
36  query="小米SU7的发动机"
37  print(web_search_answer(query))
```

Agent

planning模块

定义planning模块，根据用户问题，决定使用哪个工具来查询以获得更多需要的信息（本地文档搜索或网络搜索）

代码块

```
1   from openai import OpenAI
2   import os
3
4   def middle_json_model(prompt):
5
6       client = OpenAI(
7           # 若没有配置环境变量，请用百炼API Key将下行替换为：api_key="sk-xxx",
8           api_key='your_bailian_api_key',
9           base_url="https://dashscope.aliyuncs.com/compatible-mode/v1",
10      )
11      completion = client.chat.completions.create(
12          model="qwen-plus", # 此处以qwen-plus为例，可按需更换模型名称。模型列表：https://help.aliyun.com/zh/model-studio/getting-started/models
13          messages=[
```

```
            {'role': 'system', 'content': 'You are a helpful assistant.'},
            {'role': 'user', 'content': prompt}],
        response_format={"type": "json_object"}
        )

    return completion.choices[0].message.content
```

提取有效的json输出

代码块

```
import re
import json

def extract_json_content(input_str):
    """
    提取字符串中第一个"["和最后一个"]"之间的内容（包括中括号）

    Args:
        input_str (str): 需要处理的输入字符串

    Returns:
        str or None: 提取的JSON内容，如果没有匹配则返回None
    """
    # 使用正则表达式匹配第一个"["到最后一个"]"之间的内容
    # [\s\S]* 匹配任意字符（包括换行符）
    pattern = r'(\[[\s\S]*\])'
    match = re.search(pattern, input_str)

    # 如果匹配成功，返回匹配的内容；否则返回None
    return match.group(1) if match else None
```

测试planning模块

代码块

```
#规划模块plan
import json

def agent_plan(query):
    prompt='''

你是一个专业的汽车销售助手的规划模块。你的任务是：
1. 分析用户的查询:{0}
2. 基于已有的信息，决定使用哪个工具来查询以获得更多需要的信息（本地文档搜索或网络搜索）
3. 将用户的原始查询拆解或延伸为1-2个相关问题，以获取更全面的信息
```

```
## 可用工具
1. **本地文档搜索**：搜索本地星辰电动ES9的文档，包含以下章节：
   - 产品概述
   - 设计理念
   - 技术规格
   - 驱动系统
   - 电池与充电
   - 智能座舱
   - 智能驾驶
   - 安全系统
   - 车身结构
   - 舒适性与便利性
   - 版本与配置
   - 价格与购买信息
   - 售后服务
   - 环保贡献
   - 用户评价
   - 竞品对比
   - 常见问题
   - 联系方式

2. **网络搜索**：在互联网上搜索相关信息

## 工具选择规则
- 当查询明确涉及星辰电动ES9的具体信息、参数、功能或服务时，优先使用**本地文档搜索**
- 当查询涉及以下情况时，使用**网络搜索**：
  - 与其他品牌车型的详细对比
  - 最新市场动态或新闻
  - 非官方的用户体验或评测
  - 星辰电动ES9文档中可能没有的信息
  - 需要实时数据（如当前市场价格波动等）

## prompt延伸的规则
- 本地检索的查询扩展侧重于产品信息的深度查询
- 网络检索的查询扩展侧重于本地无法检索到的信息

## 输出格式
你的输出应该是一个JSON格式的列表，每个项目包含：
1. `action_name`：工具名称（"本地文档搜索"或"网络搜索"）
2. `prompts`：问题列表，第一个是原始查询，后面是拆解或延伸的问题
[
  {{
    "action_name": "工具名称",
    "prompts": [
      "原始查询",
```

```
        "拆解/延伸问题1",
        "拆解/延伸问题2",
        "拆解/延伸问题3"
      ]
    }}
]


## 示例

### 示例1：关于车辆规格的查询
用户：星辰电动ES9的续航里程是多少？

输出：
[
  {{
    "action_name": "本地文档搜索",
    "prompts": [
      "星辰电动ES9的续航里程是多少？",
      "星辰电动ES9的电池容量是多少？",
      "星辰电动ES9不同版本的续航里程有何区别？"
    ]
  }}
]


### 示例2：关于市场比较的查询
用户：星辰电动ES9和特斯拉Model Y相比怎么样？

输出：
[
```

调整数据格式，使每个action_name只搭配一个prompt

代码块

```
#任务状态state
def adjust_format(original_data):
    """
    调整数据格式，使每个action_name只搭配一个prompt

    参数:
    original_data (list): 原始数据，每个action_name对应多个prompts

    返回:
    list: 调整后的数据，每个action_name只对应一个prompt
    """
```

```
    adjusted_data = []

    for item in original_data:
        action_name = item['action_name']
        prompts = item['prompts']

        # 为每个prompt创建一个新的字典
        for prompt in prompts:
            adjusted_item = {
                'action_name': action_name,
                'prompt': prompt
            }
            adjusted_data.append(adjusted_item)

    return adjusted_data

if action_tool:
    adjusted_tools = adjust_format(action_tool)
    actions=adjusted_tools
else:
    actions=[]
print(actions)
```

tools模块

执行模块，依次执行actions内的动作，根据action_name判断执行函数web_search_answer()还是rag()，web_search_answer()和rag()就相当于tools，可以供agent使用

代码块

```
def process_actions(actions):
    """
    处理动作列表函数

    Args:
        actions: 动作列表，每个动作包含action_name和prompt

    Returns:
        memory: 包含每次调用结果的记忆列表
    """
    # 初始化记忆列表
    memory = []

    # 依次处理每个动作
    for action in actions:
        action_name = action['action_name']
```

```
        prompt = action['prompt']

        print(f'正在执行{action_name}: "{prompt}"')

        try:
            # 根据动作类型调用相应的函数
            if action_name == '本地文档搜索':
                result = rag(prompt)
            elif action_name == '网络搜索':
                result = web_search_answer(prompt)
            else:
                result = f"未知的动作类型: {action_name}"

            # 将结果添加到记忆中
            memory_item = {
                "提问": prompt,
                "结果": result
            }
            memory.append(memory_item)

            # 输出结果
            print(f"提问: {prompt}")
            print(f"结果: {result}")
            print("--------------------")

        except Exception:
            # 如果执行失败，静默处理，继续下一轮循环
            print("--------------------")
            continue

    print("所有执行动作已完成，结果已添加到memory中。")
    return memory
```

memory模块

记忆模块memory，用来储存搜索到的信息

代码块

```
memory_global=[]
memory_global.extend(memory_new[1:])
```

封装成agent接口

调用deepseek r1接口，将检索到的内容添加到用户prompt中，生成包含思考和推理两部分

代码块

```
from openai import OpenAI
import os

# 初始化OpenAI客户端
def final_answer(memory_global,user_query):
    client = OpenAI(
        # 如果没有配置环境变量，请用百炼API Key替换：api_key="sk-xxx"
        api_key = 'your_bailian_api_key',
        base_url="https://dashscope.aliyuncs.com/compatible-mode/v1"
    )

    reasoning_content = ""  # 定义完整思考过程
    answer_content = ""     # 定义完整回复
    is_answering = False   # 判断是否结束思考过程并开始回复


    final_prompt=f'''
        你是一个星辰电动ES9的智能销售助手，负责根据用户的问题和提供的参考内容生成回答。请严格按照以下要求生成回答：
        基于提供的参考内容进行回答，如果原文没有参考内容,根据你自己的知识进行回答
        你需要用有打动力的销售的语言进行输出，突出星辰电动的优势

        参考内容：
        {memory_global}

        用户问题：{user_query}

    '''
    print(final_prompt)
    print('-'*130)

    # 创建聊天完成请求
    completion = client.chat.completions.create(
        model="deepseek-r1",  # 此处以 deepseek-r1 为例，可按需更换模型名称
        messages=[
            {"role": "user", "content": final_prompt}
        ],
        stream=True,
        # 解除以下注释会在最后一个chunk返回Token使用量
        # stream_options={
        #     "include_usage": True
        # }
    )

    print("\n" + "=" * 20 + "思考过程" + "=" * 20 + "\n")

    for chunk in completion:
```

```
            # 如果chunk.choices为空，则打印usage
            if not chunk.choices:
                print("\nUsage:")
                print(chunk.usage)
            else:
                delta = chunk.choices[0].delta
                # 打印思考过程
                if hasattr(delta, 'reasoning_content') and delta.reasoning_content
    != None:
                    print(delta.reasoning_content, end='', flush=True)
                    reasoning_content += delta.reasoning_content
                else:
                    # 开始回复
                    if delta.content != "" and is_answering == False:
                        print("\n" + "=" * 20 + "完整回复" + "=" * 20 + "\n")
                        is_answering = True
                    # 打印回复过程
                    print(delta.content, end='', flush=True)
                    answer_content += delta.content

        # print("=" * 20 + "完整思考过程" + "=" * 20 + "\n")
        # print(reasoning_content)
        # print("=" * 20 + "完整回复" + "=" * 20 + "\n")
        # print(answer_content)
```

封装成agent接口，其中agent_plan、process_actions、final_answer可以视为三个子agent，协同工作。

代码块

```
def agent(user_query):
    #记忆模块
    memory_global=[]
    #获得执行清单
    action_tool=agent_plan(user_query)
    if action_tool:
        adjusted_tools = adjust_format(action_tool)
        actions=adjusted_tools
    else:
        actions=[]
    if actions:
        #进行任务执行
        memory_new=process_actions(actions)
        memory_global.extend(memory_new[1:])
    final_answer(memory_global,user_query)
user_query='比较一下和华为汽车的优劣势'
```

```
print(agent(user_query))
```

实现能自我反思的agent

在prompt中要求agent自我反思，基于已有的信息，是否还需要延伸再进行查询

代码块

```
def reflection(user_query,memory_global):
    prompt='''
    你是一个专业的汽车销售助手的规划模块。你的任务是：
1. 分析用户的查询:{0}
2. 基于已有的信息，是否还需要延伸再进行查询

##目前已有的信息：
{1}


## 可用工具
1. **本地文档搜索**：搜索本地星辰电动ES9的文档，包含以下章节：
   - 产品概述
   - 设计理念
   - 技术规格
   - 驱动系统
   - 电池与充电
   - 智能座舱
   - 智能驾驶
   - 安全系统
   - 车身结构
   - 舒适性与便利性
   - 版本与配置
   - 价格与购买信息
   - 售后服务
   - 环保贡献
   - 用户评价
   - 竞品对比
   - 常见问题
   - 联系方式

2. **网络搜索**：在互联网上搜索相关信息

## 工具选择规则
- 当查询明确涉及星辰电动ES9的具体信息、参数、功能或服务时，优先使用**本地文档搜索**
- 当查询涉及以下情况时，使用**网络搜索**：
  - 与其他品牌车型的详细对比
  - 最新市场动态或新闻
```

```
  - 非官方的用户体验或评测
  - 星辰电动ES9文档中可能没有的信息
  - 需要实时数据（如当前市场价格波动等）

## prompt延伸的规则
- 本地检索的查询扩展侧重于产品信息的深度查询
- 网络检索的查询扩展侧重于本地无法检索到的信息

###重要!
至多再扩展不超过3个查询，如果需要扩展则按照下面的输出格式输出，如果不需要则返回None




## 输出格式
你的输出应该是一个JSON格式的列表，每个项目包含：
1. `action_name`：工具名称（"本地文档搜索"或"网络搜索"）
2. `prompts`：一个扩展的问题，如果是网络检索，prompt不包含电动ES9，如果是本地检索，prompt只包含询问电动ES9，检索内容一定是一个简单问题，不包含对比
[
  {{
    "action_name": "工具名称",
    "prompts":'查询内容'
  }}
  ...
]

    '''.format(user_query,memory_global)
    result=(middle_json_model(prompt))
    # print(result)
    json_list=extract_json_content(result)
    try:
        structure_output=json.loads(json_list)
    except:
        structure_output = None

    return structure_output

print(reflection(user_query,memory_global))
```

封装成agent接口，在执行完process_actions后，agent进行反思，直到检索到的内容足够完善

代码块

```
def agent_reflection(user_query):
    memory_global=[]
```

```
    print("开始任务规划...")
    action_tool=agent_plan(user_query)
    if action_tool:
        adjusted_tools = adjust_format(action_tool)
        actions=adjusted_tools
    else:
        actions=[]
    if actions:
        print("开始任务执行...")
        memory_new=process_actions(actions)
        memory_global.extend(memory_new[1:])
    action_reflect=reflection(user_query,memory_global)
    if action_reflect:
        print("回顾内容，进行反思...")
        memory_new=process_actions(actions)
        memory_global.extend(memory_new)
    final_answer(memory_global,user_query)
print(agent_reflection(user_query))
```