

规划与智能体

Yuxuan Chen

School of Mathematics and Statistics
Wuhan University

2024 年 12 月 18 日



- ① 基于大语言模型的规划
- ② 基于大语言模型的智能体
- ③ 大语言模型智能体的典型应用

① 基于大语言模型的规划

整体框架

方案生成

反馈获取

② 基于大语言模型的智能体

③ 大语言模型智能体的典型应用

基于大语言模型的规划

- 规划旨在**为目标任务制定包含一系列动作的解决方案**，是大语言模型解决复杂问题能力的重要体现，也是自主智能体最重要的核心能力。
- **背景**：上下文学习和思维链提示方法形式上较为简洁且较为通用，但是在面对诸如几何数学求解、游戏、代码编程以及日常生活任务等复杂任务时仍然表现不佳。
- **基于大语言模型规划的核心思想**：将复杂任务**分解**为若干相关联的子任务，并围绕这些子任务制定包含一系列执行动作 (Action) 的解决方案，从而**将复杂任务的求解转换为一系列更为简单的子任务依次求解**，进而简化了任务难度。

① 基于大语言模型的规划

整体框架

方案生成

反馈获取

② 基于大语言模型的智能体

③ 大语言模型智能体的典型应用

整体框架

- 基于大语言模型的规划方法主要由三个组件构成：
- **任务规划器**：主要职责是生成目标任务的解决方案。
该方案包含一系列执行动作，每个动作通过合适的形式进行表达，例如自然语言描述或代码片段。
- **规划执行器**：负责执行解决方案中所涉及到的动作。
根据任务性质的不同，规划执行器可以由大语言模型实现，也可以由执行具体物理任务的实体（如机器人）来实现。
- **环境**：规划执行器实施动作的具体场景。
不同任务对应着不同的执行环境，例如 Web 互联网或像 Minecraft 这样的外部虚拟世界。

整体框架

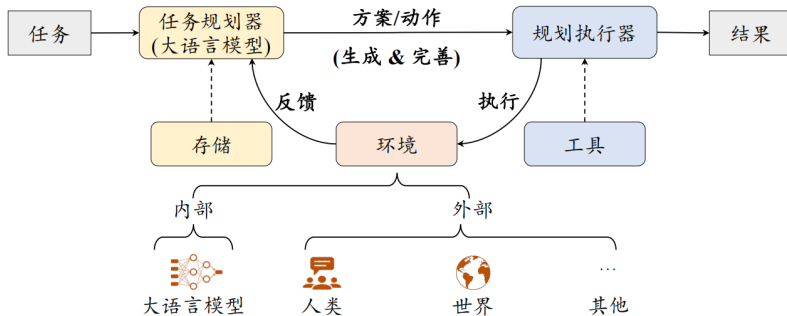


图 11.1 大语言模型通过基于提示的规划解决复杂任务的流程（图片来源：[10]）

① 基于大语言模型的规划

整体框架

方案生成

反馈获取

② 基于大语言模型的智能体

③ 大语言模型智能体的典型应用

方案生成

- 方案生成主要是基于大语言模型的综合理解与推理能力，通过合适的提示让大语言模型生成目标任务的解决方案。
- 解决方案（或者其中包含的中间步骤）可以采用自然语言表达或者代码表达的形式。

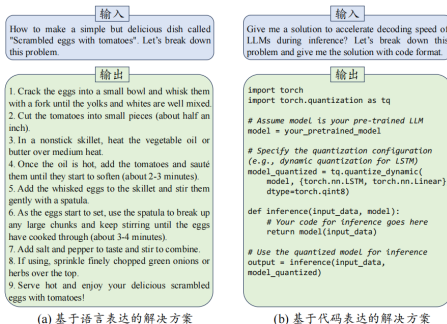


图 11.2 基于语言表达和基于代码表达的方案生成示例对比

一次性方案和迭代式方案

- 任务规划器主要采用一次性的方案生成和迭代式的方案生成两种规划方法
- **一次性方案生成**：通过特殊设计的提示方法让大语言模型一次性生成完整的解决方案，生成的方案通常包含一系列供规划执行器执行的动作描述。
- **迭代式方案生成**：大语言模型基于历史动作和当前环境的反馈逐步规划下一步的执行动作。

ReAct 方法和 Reflexiton 方法

- ReAct 方法:
- 核心动机是在让大语言模型在规划动作时模拟人类“先思考-再决策”的行为方式。即该方法首先通过提示让大语言模型思考当前状态下应该采取何种决策，并生成决策理由与相应的执行动作。
- Reflexiton 方法:
- 该方法旨在借助大语言模型的分析与推理能力，对于当前方案的执行结果给出具体的反思结果，用于改进已有的解决方案。

具体示例

(首先, 输入问题和一些相关的问题-解决方案的示例 (此处为方便表示, 我们省略示例的具体内容), 任务规划器进行第一轮规划。首先, 生成第一步动作, 其内容具体包括“思考 1”和“动作 1”。)

第一轮规划: Question: Grown-Ups starred the actor who was best known for which role on “Allo Allo?”

思考 1: I need to search Grown-Ups and “Allo Allo!”, find the actor who starred in Grown-Ups, then find the role he was best known for in “Allo Allo!”.

动作 1: Search[Grown-Ups]

(然后, 规划执行器执行“动作 1”, 并将得到反馈, 即“反馈 1”。)

反馈 1: Grown-Ups is a 1980 British BBC television film devised and ... on 28 November 1980.

(在后续步骤中, 将历史涉及的动作和反馈拼接在一些输入给任务规划器, 然后继续生成下一步的动作, 以此类推, 直至第四步, 生成“动作 4”, 即停止推理。)

思考 2: Grown-Ups starred Lesley Manville, Philip Davis, Brenda Blethyn, Janine Duvitski, Lindsay Duncan and Sam Kelly. I need to search “Allo Allo!” and find which actor was best known for which role.

动作 2: Search[“Allo Allo!”]

反馈 2: Could not find [“Allo Allo!”] ... on 28 November 1980.

...

思考 4: Gordon Kaye was best known for playing womanising café owner René Artois in the television comedy series “Allo Allo!”. So the answer is Rene Artois.

动作 4: Finish[René Artois]

(执行“动作 4”后, 得到最终结果, 即“反馈 4”。)

反馈 4: Answer is INCORRECT

(执行后发现答案不正确。然后, 我们将完整解决方案及每一步动作对应的反馈拼接输入给任务规划器, 同时添加一些相关的问题-解决方案-反思的示例 (此处为方便表示, 我们省略示例具体内容) 帮助任务规划器生成对当前解决方案的反思, 即“反思 1”。)

反思 1: I searched the wrong title for the show, “Allo Allo!”, which resulted in no results. I should have searched the show’s main character, Gordon Kaye, to find the role he was best known for in the show.

(上述是一轮完整的规划, 我们将上一轮不正确的完整求解策略及其反馈, 以及反思拼接输入给任务规划器, 其重新生成新的求解策略的每一步, 以此类推, 直至得到成功的反馈。)

第二轮规划: Question: Grown-Ups starred the actor who was best known for which role on “Allo Allo?”

思考 1: I need to search Grown-Ups ... in “Allo Allo!”.

动作 1: Search[Grown-Ups]

反馈 1: Grown-Ups is a 1980 British BBC television film devised and ... on 28 November 1980.

(可以看到, 在加入“反思 1”后, 模型在第二步生成时, 对“思考 2”和“动作 2”进行了修正, 我们用下划线进行了强调。)

思考 2: The paragraph does not mention the actor who was best known for a role on “Allo Allo!”. I need to search the actor’s name instead.

动作 2: Search[Sam Kelly]

反馈 2: Roger Michael Kelly ... Sam in On the Up, and Ted Liversidge in Barban.

思考 3: Sam Kelly is best known for his role as Captain Hans Geering in “Allo Allo!”, so the answer is Captain Hans Geering.

动作 3: Finish[Captain Hans Geering]

反馈 3: Answer is CORRECT

(执行后答案正确, 结束规划求解。)

图 11.3 使用 ReAct 方法 (单轮规划) 与 Reflexion 方法 (多轮规划) 求解多跳问答任务示例

① 基于大语言模型的规划

整体框架

方案生成

反馈获取

② 基于大语言模型的智能体

③ 大语言模型智能体的典型应用

反馈获取

- 在执行动作后，规划执行器会将环境的反馈信号传递任务规划器。这些反馈信号可以用于完善整体解决方案或规划下一步动作。
- 外部反馈：
- 外部对象可以为任务规划器提供重要的反馈信号。如物理工具、人类以及虚拟环境这三种外部对象。
- 内部反馈：
- 大语言模型本身也能够对任务规划器提供反馈信息，如可以直接判断当前动作是否规划正确，可以将简单的环境反馈（例如成功或失败）转换为信息量更为丰富的、自然语言表达的总结反思，帮助任务规划器重新生成改进的解决方案。

① 基于大语言模型的规划

② 基于大语言模型的智能体

大语言模型智能体的构建

RecAgent

多智能体系统的构建

③ 大语言模型智能体的典型应用

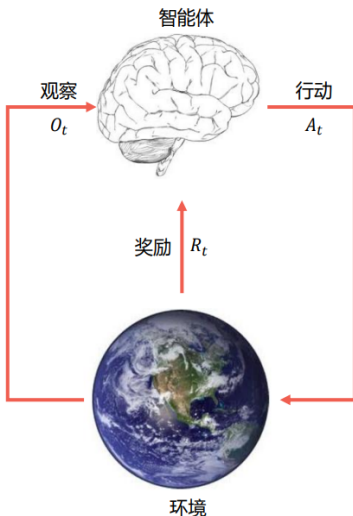
智能体概述

- 智能体 (Agent) 是一个具备环境感知、决策制定及动作执行能力的自主算法系统。

智能体概述

- **智能体 (Agent)** 是一个具备环境感知、决策制定及动作执行能力的自主算法系统。
- **LLM 智能体和 RL 智能体的对比:**
- 架构设计方面: LLM 智能体更倾向于通过已有知识和提示设计完成任务, 不需要显式地与环境交互进行学习, 而不是像 RL 智能体那样通过环境试错优化策略。
- 能力获取方面: LLM 智能体主要通过**微调、提示设计和机制工程**获取能力, 与 RL 智能体通过**试错和策略优化**进行能力强化的方式不同。
- 决策机制方面: LLM 智能体决策基于**上下文推理和自然语言生成**, 行为更多依赖静态数据中获取的知识, 通常通过模块化设计 (如记忆模块、规划模块、行动模块) 实现类似人类的行为逻辑。RL 智能体决策依赖于**策略函数**, 通过动作-反馈-更新的循环改进行为。

智能体概述



□ 在每一步 t , 智能体:

- 获得观察 O_t
- 获得奖励 R_t
- 执行行动 A_t

□ 环境:

- 获得行动 A_t
- 给出观察 O_{t+1}
- 给出奖励 R_{t+1}

□ t 在环境这一步增加

- ① 基于大语言模型的规划
- ② 基于大语言模型的智能体
 - 大语言模型智能体的构建
 - RecAgent
 - 多智能体系统的构建
- ③ 大语言模型智能体的典型应用

大语言模型智能体构建的流程

- 1, 明确目标与需求 (如 RecAgent 是用户行为模拟, WebGPT 是浏览互联网获取实时信息来回答问题)
- 2, 系统架构设计 (模块划分、交互设计、工具框架、硬件资源等)
- 3, 模块开发 (设计记忆、规划等各个模块的内容并且编写相应代码)
- 4, 测试与调试 (测试各模块是否独立运行良好, 运行时间等等)
- 5, 调优与迭代 (提示优化、反馈机制调整、记忆机制优化等)
- 6, 部署与上线 (部署 LLM 至本地或云端, 配置必要的计算资源)
- 7, 应用与扩展 (多场景适配、多智能体协作、多模态扩展等)

大语言模型智能体各个组件

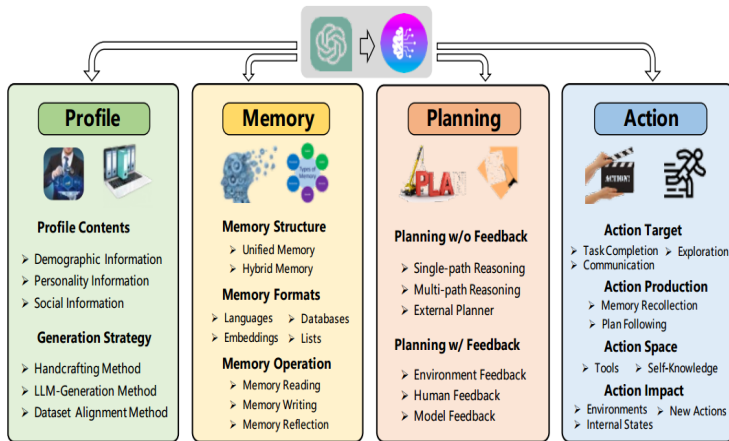


Fig. 2 A unified framework for the architecture design of LLM-based autonomous agent.

档案组件 (Profile)

- 档案组件用于定义和存储智能体的基本属性和个性化特征，这些特征为智能体的行为决策、记忆检索和规划提供了重要依据。
- 主要功能是通过定义智能体的个性化特征，模拟真实人类的多样性，并使智能体的行为更加符合其“身份设定”。(个性、社交、兴趣偏好等)。

记忆组件 (Memory)

- 记忆组件主要用于存储智能体与环境的历史交互记录，并能够随时检索使用，这些信息可以是文本形式，也可以是图像、声音等多模态形式。
- 短期记忆：负责暂时存储和处理智能体相关信息的记忆载体。
特点：信息存储持续时间相对较短，对于信息容量有一定的限制，大部分只使用一次，必要时可以转变为长期记忆存储。
- 长期记忆：智能体存储长期累积信息的记忆载体。
特点：存储内容具有持久性，存储方式比较灵活，可以是文本文件、结构化数据库等形式，一般智能体的角色和功能定义通过长期记忆来存储。

记忆组件 (Memory)

调用记忆组件:

长期记忆:

Name: Bob (gender: male; age: 25; traits: compassionate, caring, ambitious, optimistic; career: photographer; interest: sci-fi movies, comedy movies; feature: watcher, critic, poster).

Bob recently heard ['The Matrix', 'Back to the Future.', 'Anchorman', 'Superbad'] on social media

Alice recently watched nothing on recommender system. Other than that Alice doesn't know any movies.

短期记忆:

It is September 12, 2023, 08:00 AM.

Most recent observations: Bob and Alice had a conversation about their shared interest in movies, discussing their favorite genres such as...

加入新记忆时对记忆重要性打分:

Observation: Alice want to post for all acquaintances.

规划组件 (Planning)

- 规划组件为智能体引入了类似于人类解决任务的思考方式，
将复杂任务分解为一系列简单的子任务，进而逐一进行解决。
- 优点：降低了一次性解决任务的难度，有助于提高问题解决的效率和效果，提高了智能体对复杂环境的适应性和操作的可靠性。

调用规划组件：

Plan: Bob first wants to enter the recommendation system to find a movie he is interested in, then watch the movie, and afterwards, chat with friends about the movie.

执行组件（Execution）

- 执行组件的主要职责是执行由规划组件制定的任务解决方案。
- 具体来说，智能体会在行动决策过程中执行规划组件制定的明确行动规划，同时会参考记忆组件中的长短期记忆来帮助执行准确的行动。

调用行动组件：

选择进入推荐系统或者社交平台：

[RECOMMENDER]: Bob enters the Recommender System

接受推荐，观看电影：

[RECOMMENDER]: Bob watches the movie <Interstellar>

智能体之间聊天：

[Bob]: Hey Alice! How's it going? I heard you were interested in a movie. What's been on your mind?

[Alice]: Hey Bob! I'm doing great, thanks for asking. Yeah, I've been hearing a lot about this movie <Interstellar> recently. Have you heard of it too?

[Bob]: Absolutely! Actually, I've been seeing it all over social media as well.

[Alice]: That's awesome! I'm glad you enjoyed it. I've been wanting to watch it too. Would you be up for a cozy movie night to watch it together? We can discuss our thoughts and interpretations afterwards.

[Bob]: I'd love that! It's always more fun to watch movies with friends and have those deep conversations afterwards. Count me in!

...

工作流程

- 一、智能体对当前状态进行理解和分析。
- 二、规划组件通过综合考虑长短期记忆组件中已存储的信息，生成下一个行动策略或计划。
- 三、执行组件负责根据规划组件生成的任务解决方案执行实际行动，并与当前环境产生交互。
- 四、智能体通过感知单元或系统接口从环境中接收反馈信息，并将这些信息暂时存储于短期记忆中。智能体会对短期记忆中的新获取到的信息进行处理。上述流程将作为新的记忆被记录在记忆组件中。

① 基于大语言模型的规划

② 基于大语言模型的智能体

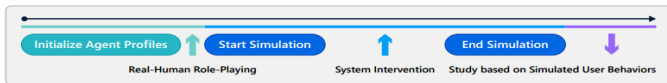
大语言模型智能体的构建

RecAgent

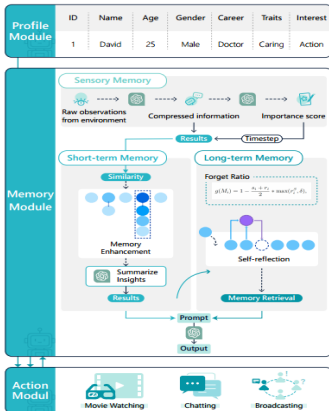
多智能体系统的构建

③ 大语言模型智能体的典型应用

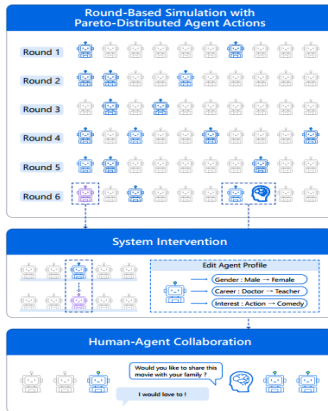
RecAgent 框架



(a) The Simulator Running Process



(b) The Agent Framework



(c) The Simulator Implementations

遗忘机制

- 遗忘率公式:

$$g(M_t) = 1 - \frac{A_t + r_t}{2} \times \max(x_t, \theta)$$

- $g(M_t)$: 当前记忆 M_t 的遗忘率
- A_t : 当前记忆的访问频率
- r_t : 记忆的重要性分数
- x_t : 当前时间
- θ : 遗忘阈值, 控制遗忘的最低标准, 防止所有记忆都被遗忘

Pareto Distribution

- Pareto 分布是一个连续概率分布，描述的变量 X 大于某个最小值 $x_m > 0$ 且遵循以下概率密度函数 (PDF):

$$f(x) = \begin{cases} \frac{\alpha x_m^\alpha}{x^{\alpha+1}}, & x \geq x_m \\ 0, & x < x_m \end{cases}$$

- x_m : 最小值，分布的起始点。
- $\alpha > 0$: 形状参数，称为“幂指数”或“帕累托指数”，决定分布的陡峭程度。
- Pareto 分布表示少数个体占据了大多数资源或影响力，而大多数个体的影响力很小，适用于许多领域。

RecAgent 代码

- 网址: <https://github.com/RUC-GSAI/YuLan-Rec>
- agent 主体部分代码为:
recagent.py, recagent-memory.py, roleagent.py
- recagent.py: recagent 的主体类, 通过 generate-plan, take-action 等函数来实现智能体的功能。
- recagent-memory.py: 对应于智能体的记忆模块, 内置感官记忆、短期记忆、长期记忆等 class。
- roleagent.py: 写了一个名为 roleagent 的 recagent 的子类, 主要用来模拟用户角色的行为, 并允许用户在模拟器中进行各种操作。

RecAgent 代码（短期记忆）

- 相似性计算：通过嵌入向量计算相似性，决定是否增强现有记忆

```
def cosine_similarity(embedding1: List[float], embedding2: List[float]):
    dot_product = np.dot(embedding1, embedding2)
    norm1 = np.linalg.norm(embedding1)
    norm2 = np.linalg.norm(embedding2)
    similarity = dot_product / (norm1 * norm2)
    return similarity

def add_stm_memory(self, observation: str, importance: float, op: str):
    observation_embedding = self.embeddings_model.embed_query(observation)
    for idx, memory_embedding in enumerate(self.short_embeddings):
        similarity = self.cosine_similarity(observation_embedding, memory_embedding)
        if idx + 1 == len(self.short_embeddings):
            similarity += const # 增加首位效应
        prob = 1 / (1 + np.exp(-similarity)) # 使用 Sigmoid 函数生成增强概率
        if prob >= 0.7 and random() <= prob:
            self.enhance_cnt[idx] += 1
            self.enhance_memories[idx].append(observation)
```

RecAgent 代码（短期记忆）

- 相似性计算：通过嵌入向量计算相似性，决定是否增强现有记忆

```
def cosine_similarity(embedding1: List[float], embedding2: List[float]):
    dot_product = np.dot(embedding1, embedding2)
    norm1 = np.linalg.norm(embedding1)
    norm2 = np.linalg.norm(embedding2)
    similarity = dot_product / (norm1 * norm2)
    return similarity

def add_stm_memory(self, observation: str, importance: float, op: str):
    observation_embedding = self.embeddings_model.embed_query(observation)
    for idx, memory_embedding in enumerate(self.short_embeddings):
        similarity = self.cosine_similarity(observation_embedding, memory_embedding)
        if idx + 1 == len(self.short_embeddings):
            similarity += const # 增加首位效应
        prob = 1 / (1 + np.exp(-similarity)) # 使用 Sigmoid 函数生成增强概率
        if prob >= 0.7 and random() <= prob:
            self.enhance_cnt[idx] += 1
            self.enhance_memories[idx].append(observation)
```

RecAgent 代码（短期记忆）

- 增强机制：增强记忆的次数通过 enhance-cnt 控制，记忆将根据相似性和增强次数进行转移

```
def transfer_memories(self, observation):
    # 如果记忆增强次数超过阈值，进行转移
    for idx, memory in enumerate(self.short_memories):
        if self.enhance_cnt[idx] >= self.enhance_threshold:
            # 将记忆转移到长期记忆
            existing_memory[idx] = False
            content = [memory]
            for enhance_memory in self.enhance_memories[idx][:-1]:
                content.append(enhance_memory)
            content.append(observation)
            content = ';'.join(content)
            memory_content.append(memory)
            memory_importance.append(self.memory_importance[idx])
            insight = self.get_short_term_insight(content)
            insight_content.append(insight)
```

RecAgent 代码（短期记忆）

- 见解生成：根据短期记忆生成的总结见解，增强对个体特征的理解
- 该方法生成了一个 prompt(提示词)，然后用大语言模型接受提示词，创建 LLM 对象，之后调用该对象来运行实际的推理过程，最后生成我们想要的见解。

```
def get_short_term_insight(self, content: str):
    prompt = PromptTemplate.from_template(
        "There are some memories separated by semicolons (;): {content}\n"
        + "Can you infer from the above memories the high-level insight for this per
        + "The insight needs to be significantly different from the content and stru
        + "Respond in one sentence."
        + "\n\nResults:"
    )
    result = self.chain(prompt).run(content=content).strip()
    return self._parse_list(result)
```

RecAgent 代码（长期记忆）

- 自我反思机制：通过对近期记忆生成高层次问题，并从相关记忆中总结洞见（insight）。
- 使用最近的 last-k 条记忆生成一个关键反思主题，提取高层次反思问题。

```
def _get_topics_of_reflection(self, last_k: int = 50) -> List[str]:
    prompt = PromptTemplate.from_template(
        "{observations}\n\n"
        + "Given only the information above, what is the 1 most salient"
        + " high-level questions we can answer about the subjects in"
        + " the statements? Provide the question on a new line.\n\n"
    )
    # 获取最近的记忆内容
    observations = self.memory_retriever.memory_stream[-last_k:]
    observation_str = "\n".join([o.page_content for o in observations])

    # 使用 LLMChain 生成高层次问题
    result = self.chain(prompt).run(observations=observation_str)
    return self._parse_list(result)
```

RecAgent 代码（长期记忆）

- 针对给定反思主题，结合相关记忆生成洞察，同时标明相关记忆的索引。

```
def _get_insights_on_topic(self, topic: str, now: Optional[datetime] = None):
    prompt_insight = PromptTemplate.from_template(...)

    # 获取与主题相关的记忆
    related_memories = self.fetch_memories(topic, now=now)
    related_statements = "\n".join(
        [memory.page_content for memory in related_memories]
    )

    # 使用语言模型生成洞察
    result_insight = self.chain(prompt_insight).run(
        topic=topic, related_statements=related_statements
    )

    # 解析洞察结果
    result_insight = self._parse_list(result_insight)
    result_insight = [self._parse_insight_with_connections(res) for res in result_insight]

    # 将洞察与记忆的嵌入向量对比，选择相关的记忆索引
    embedding_1 = self.embeddings_model.embed_query(result_insight[0][0])
    for memory_id in result_insight[0][1]:
        memory = self.memory_retriever.memory_stream[memory_id].page_content
        memory_embedding = self.embeddings_model.embed_query(memory)
        similarity = self.cosine_similarity(embedding_1, memory_embedding)
        # 如果相似度超过阈值，标记为合并
        if similarity >= 0.72:
            self.memory_retriever.memory_stream[memory_id].page_content = '[MERGE]'
            self.memory_retriever.memory_stream[memory_id].metadata['importance'] = 1.0
    return result_insight
```

RecAgent 代码（长期记忆）

- 从最近的记忆生成反思主题，针对主题生成洞察，并更新记忆库。

```
def pause_to_reflect(self, now: Optional[datetime] = None):
    if self.verbose:
        logger.info("Character is reflecting")

    new_insights = []
    # 获取反思主题
    topics = self._get_topics_of_reflection()
    for topic in topics:
        # 针对每个主题生成洞察
        insights = self._get_insights_on_topic(topic, now=now)
        for insight in insights:
            text, par_list = insight
            importance_cur = sum(
                self.memory_retriever.memory_stream[par].metadata['importance']
                for par in par_list if par < len(self.memory_retriever.memory_stream)
            ) / len(par_list)

            # 添加洞察到长期记忆
            ltm = importance_cur, now, text
            self.add_memory(ltm, now=now)
        new_insights.extend(insights)
    return new_insights
```

① 基于大语言模型的规划

② 基于大语言模型的智能体

大语言模型智能体的构建

RecAgent

多智能体系统的构建

③ 大语言模型智能体的典型应用

多智能体系统的构建方法

- 一，明确多智能体系统整体需要解决的问题或实现的目标，
可以针对特定任务，也可以针对某一个环境进行仿真模拟。
- 二，在系统内创建多个智能体实例，根据问题的复杂性和所需功能，设计智能体的类型、数量和特性。
- 三，定义多智能体之间的交互方式，包括协作、竞争、信息交流等方面，以及制定协议、策略或博弈论规则，以确保智能体之间能够有效进行协同运作。

多智能体系统的通讯协同机制

- 两种机制的核心在于加强智能体之间的信息交流与能力共享。
- 通讯机制: 通常包括三个基本要素: **通讯协议**、**通讯拓扑**和**通讯内容**。通讯协议规定了智能体之间如何进行信息交换和共享, 包括通讯的方式、频率、时序等; 通讯拓扑则定义了智能体之间的连接关系, 即哪些智能体之间可以进行直接通讯, 哪些需要通过其他智能体进行间接通讯; 通讯内容则是指智能体之间实际传输的信息, 包括状态信息、控制指令、任务目标等。
- 协同机制: 多智能体系统的协同机制通常包括**协作**、**竞争**和**协商**。协作指的是智能体通过共享资源、信息和任务分配来实现共同目标; 竞争则涉及到在资源有限的环境中, 智能体之间的竞争关系, 协商是指智能体通过交换信息和让步来解决目标或资源的冲突。

- ① 基于大语言模型的规划
- ② 基于大语言模型智能体
- ③ 大语言模型智能体的典型应用
 - WebGPT
 - MetaGPT
 - 待解决的关键技术问题

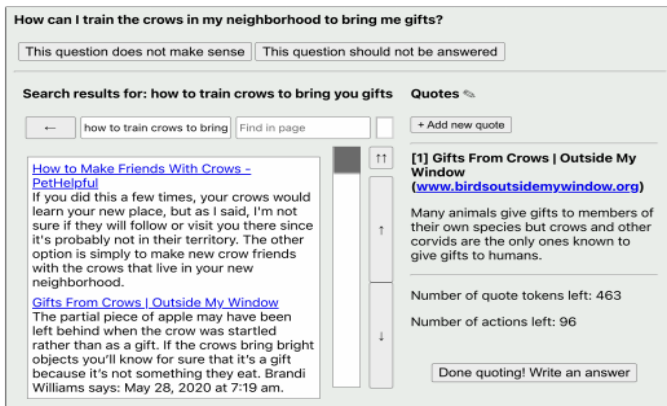
- ① 基于大语言模型的规划
- ② 基于大语言模型智能体
- ③ 大语言模型智能体的典型应用
 - WebGPT
 - MetaGPT
 - 待解决的关键技术问题

WebGPT

- WebGPT 通过浏览器辅助的方式，利用互联网作为信息源来回答问题。它结合 **大语言模型**（如 GPT-3）和 **网络搜索功能**，在回答问题时能够实时访问互联网获取信息，生成基于事实且支持引用的高质量回答。
- WebGPT 具备自主搜索、自然语言交互以及信息整合分析等特点，能够理解用户的自然语言查询，自动在互联网上搜索相关网页。根据搜索结果，WebGPT 能够点击、浏览、收藏相关网页信息，对搜索结果进行分析和整合，最终以自然语言的形式提供准确全面的回答，并提供参考文献。

WebGPT

● 工作流程的演示



(a) Screenshot from the demonstration interface.

- ① 基于大语言模型的规划
- ② 基于大语言模型智能体
- ③ 大语言模型智能体的典型应用
 - WebGPT
 - MetaGPT
 - 待解决的关键技术问题

MetaGPT

- MetaGPT 是一个基于多智能体系统的协作框架，旨在模仿人类组织的运作方式，模拟软件开发过程中的不同角色和协作。
- 它可以模拟软件开发公司，在用户输入一句话命令后按照流程进行软件开发。

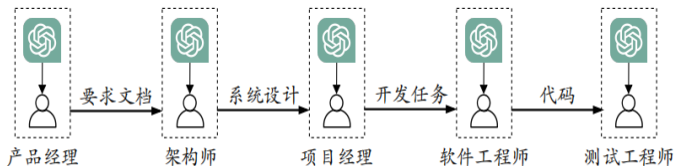


图 11.4 MetaGPT 执行软件开发工作的全流程示例

MetaGPT

- 开发一个智能体团队：
- 一，定义每个角色能够执行的预期动作
- 二，基于标准作业程序（SOP）确保每个角色遵守它。通过使每个角色观察上游的相应输出结果，并为下游发布自己的输出结果，可以实现这一点。
- 三，初始化所有角色，创建一个带有环境的智能体团队，并使它们之间能够进行交互。

MetaGPT（定义动作）

- 在 MetaGPT 中，类 Action 是动作的逻辑抽象。用户可以通过简单地调用 self.aask 函数令 LLM 赋予这个动作能力，即这个函数将在底层调用 LLM api。

```
class SimpleWriteCode(Action):
    PROMPT_TEMPLATE: str = """
    Write a python function that can {instruction}.
    Return ```python your_code_here ``` with NO other texts,
    your code:
    """

    name: str = "SimpleWriteCode"

    async def run(self, instruction: str):
        prompt = self.PROMPT_TEMPLATE.format(instruction=instruction)

        rsp = await self._aask(prompt)

        code_text = parse_code(rsp)

        return code_text
```

MetaGPT（定义角色）

- 在 MetaGPT 中，Role 类是智能体的逻辑抽象。一个 Role 能执行特定的 Action，拥有记忆、思考并采用各种策略行动。基本上，它充当一个将所有这些组件联系在一起的凝聚实体。

```
class SimpleCoder(Role):  
    name: str = "Alice"  
    profile: str = "SimpleCoder"  
  
    def __init__(self, **kwargs):  
        super().__init__(**kwargs)  
        self._watch([UserRequirement])  
        self.set_actions([SimpleWriteCode])
```

MetaGPT（创建一个团队并添加角色）

- 初始化所有角色，设置一个 Team，并 hire 它们。

```
import fire
import typer
from metagpt.logs import logger
from metagpt.team import Team
app = typer.Typer()
1 usage
@app.command()
def main(
    idea: str = typer.Argument(default=..., help="write a function that calculates the product of a list"),
    investment: float = typer.Option(default=3.0, help="Dollar amount to invest in the AI company."),
    n_round: int = typer.Option(default=5, help="Number of rounds for the simulation."),
):
    logger.info(idea)

    team = Team()
    team.hire(
        [
            SimpleCoder(),
            SimpleTester(),
            SimpleReviewer(),
        ]
    )
    team.invest(investment=investment)
    team.run_project(idea)
    await team.run(n_round=n_round)
if __name__ == "__main__":
    fire.Fire(main)
```

52 / 59

MetaGPT（配置大模型 API）

- 以 OpenAI API 为例说明配置过程，其他大模型的 API 配置过程是相同的。
- 可以通过设置 config2.yaml 完成配置：一、在当前工作目录中创建一个名为 config 的文件夹，并在其中添加一个名为 config2.yaml 的新文件。二，将示例 config2.yaml 文件的内容复制到自己的新文件中。三，将自己的值填入文件中。

```
llm:
  api_type: 'openai' # or azure / ollama / groq etc. Check LLType for more options
  api_key: 'sk-...' # YOUR_API_KEY
  model: 'gpt-4-turbo' # or gpt-3.5-turbo
  # base_url: 'https://api.openai.com/v1' # or any forward url.
  # proxy: 'YOUR_LLM_PROXY_IF_NEEDED' # Optional. If you want to use a proxy, set it here.
  # pricing_plan: 'YOUR_PRICING_PLAN' # Optional. If your pricing plan uses a different name than the `model`
```

MetaGPT（为角色或动作配置不同 LLM）

- MetaGPT 允许你为团队中的不同 Role 和 Action 使用不同的 LLM，这极大地增强了团队互动的灵活性和现实性，使得每个 Role 可以根据其特定的需求和背景，以及每个 Action 的特点，选择最合适的 LLM。
- 设置步骤为：
 - 一、定义配置：使用默认配置，或者从 `/.metagpt` 目录中加载自定义配置。
 - 二、分配配置：将特定的 LLM 配置分配给 Role 和 Action。配置的优先级：Action config > Role config > Global config (config in config2.yaml)。
 - 三、团队交互：创建一个带有环境的团队，开始交互。

MetaGPT（定义配置）

- 可以使用默认配置，为不同的 Role 和 Action 配置 LLM，也可以在 `/.metagpt` 目录中加载自定义配置。

```
from metagpt.config2 import Config
```

```
# 以下是一些示例配置，分别为gpt-4、gpt-4-turbo 和 gpt-3.5-turbo。
```

```
gpt4 = Config.from_home("gpt-4.yaml") # 从` ~/.metagpt `目录加载自定义配置`gpt-4.yaml`
```

```
gpt4t = Config.default() # 使用默认配置，即`config2.yaml`文件中的配置，此处`config2.yaml`文件中的model为`gpt-4-turbo`
```

```
gpt35 = Config.default()
```

```
gpt35.llm.model = "gpt-3.5-turbo" # 将model修改为`gpt-3.5-turbo`
```


MetaGPT（分配配置）

- 创建 Role 和 Action，并为其分配配置。

```
from metagpt.roles import Role
from metagpt.actions import Action

# 创建a1、a2和a3三个Action。并为a1指定`gpt4t`的配置。
a1 = Action(config=gpt4t, name="Say", instruction="Say your opinion with emotion and don't repeat it")
a2 = Action(name="Say", instruction="Say your opinion with emotion and don't repeat it")
a3 = Action(name="Vote", instruction="Vote for the candidate, and say why you vote for him/her")

# 创建A, B, C三个角色，分别为“民主党候选人”、“共和党候选人”和“选民”。
# 虽然A设置了config为gpt4t，但因为a1已经配置了Action config，所以A将使用model为gpt4t的配置，而a1将使用model为gpt4t的配置。
A = Role(name="A", profile="Democratic candidate", goal="Win the election", actions=[a1], watch=[a2], config=gpt4t)
# 因为B设置了config为gpt35，而为a2未设置Action config，所以B和a2将使用Role config，即model为gpt35的配置。
B = Role(name="B", profile="Republican candidate", goal="Win the election", actions=[a2], watch=[a1], config=gpt35)
# 因为C未设置config，而a3也未设置config，所以C和a3将使用Global config，即model为gpt4t的配置。
C = Role(name="C", profile="Voter", goal="Vote for the candidate", actions=[a3], watch=[a1, a2])
```

MetaGPT（团队交互）

- 创建一个带有环境的团队，并使其进行交互。

```
import asyncio

from metagpt.environment import Environment
from metagpt.team import Team

# 创建一个描述为“美国大选现场直播”的环境
env = Environment(desc="US election live broadcast")
team = Team(investment=10.0, env=env, roles=[A, B, C])
# 运行团队，我们应该会看到它们之间的协作
asyncio.run(team.run(idea="Topic: climate change. Under 80 words per message.", send_to="A", n_round=3))
# await team.run(idea="Topic: climate change. Under 80 words per message.", send_to="A", n_round=3) # 如果在Jupyter
```

- ① 基于大语言模型的规划
- ② 基于大语言模型的智能体
- ③ 大语言模型智能体的典型应用
 - WebGPT
 - MetaGPT
 - 待解决的关键技术问题

待解决的关键技术问题

- 智能体系统的计算资源消耗
- 复杂工具使用
- 高效的多智能体交互机制
- 面向智能体系统的模型适配方法
- 面向真实世界的智能体模拟