

Agently AI 应用开发框架 由浅入深的应用开发指导 (应用开发者入门篇)

Getting Started with **Agently** : a Step-by-Step Guide

for LLMs Application Developers

本文档在线版本请[点此链接](#)

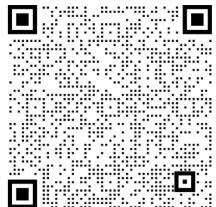
或扫描下方二维码访问：



本文档将系统性为您讲解**Agently AI**应用开发框架的相关开发支持，帮助您逐步深入理解框架设计理念，并能够由浅入深地逐步掌握对复杂LLMs应用的开发技能。

本文档为系列指导中的“入门篇”，主要为框架所面向的应用开发者角色服务，后续还将有相关文档更新，会包括面向框架插件开发者以及进阶开发者相关的更多内容，敬请关注。

💬 您还可以[点击这里](#)或者扫描下方二维码申请加入我们的微信讨论群：



第一部分：

安装并开始使用

第一章：框架介绍

Agently AI应用开发框架是一个开源的AI Agent原生应用开发框架，Github仓库主页：
<https://github.com/Maplemx/Agently>

您也可以通过我们的官方网址Agently.cn或是Agently.tech进行访问。

Agently 框架具有如下特点：

- **快速开始**: 只需要很少的几行代码，就可以开始创建你的大语言模型驱动的应用；
- **表达轻松**: 框架特有的面向agent对象设计，结合链式表达、代码变量直接输入、结构化数据输出等特性，能够帮助你轻松快速地将自己的业务代码和框架生成的agent对象实例操作进行结合；
- **插件增强**: 框架采用了插件式增强的设计，并在持续更新新的插件，你可以通过插件获得对更多模型的适配、agent请求过程监听、自定义工具注册和调用、多轮对话记录自动存储、角色管理、agent状态管理和行为修正映射等各式各样的能力，如果你有更多新的能力构思，也可以创建自己的自定义插件对agent能力进行增量更新，而不是重新开发整个agent；
- **开源开放**: 我们是一个开源项目，框架源码透明可见，并且非常欢迎开发者加入我们的开源社群，找到自己的定位和角色，开发应用、更新插件或是优化框架核心代码，共同讨论框架未来的发展。

相关资源

除了这份《由浅入深的应用开发指导》外，我们的Github仓库主页中还有更多相关文档，大部分文档都使用了Jupyter格式，并且支持一键导入Google Colab进行在线试用。

说明文档:

- [Agently 3.0 框架概念介绍](#)
- [Agently 3.0 应用开发功能说明文档](#)

案例广场:

构建不同领域的Agent示例:

- [根据自然语言生成SQL的提数小帮手Agent](#)
- [根据问卷进行反馈收集的问卷调查Agent](#)
- [自带搜索技能的少儿教师Agent](#)
- [根据图片以及文案样例生成广告文案的营销Agent](#)
- [根据用户简单输入，自动创建人设，并能够在多个状态间切换的角色扮演Agent](#)
- [能够使用工具（框架内置工具或自定义工具）的Agent](#)
- [在游戏场景中为NPC提供行动选项和演出台词的Agent](#)
- [通过接待Agent引导用户和自己具有权限的职能Agent互动](#)

在代码逻辑中寻求Agent的支持:

- [基于长文本，生成与文本内容相关的问答对](#)
- [在回复用户前，对Agent回复进行人工介入检查](#)
- [在多个头尾文本对里寻找可连接的文本对](#)
- [配合AsyncIO使用异步编程，实现串行依赖和并发请求编排](#)
- [使用函数装饰器，让Agent帮你完成仅有注释的函数指定的任务](#)

更多案例持续更新，访问我们的[Playground案例广场](#)即可查看

第二章：安装并快速熟悉框架的基本特性

安装准备

本地环境下，您只需要在shell中输入 `pip install -U Agently` 即可完成安装

如果您使用Jupyter或是Colab，可以直接通过运行下面的代码块进行安装：

```
In [ ]: !pip install -q -U Agently
```

快速熟悉框架的基本特性

配置并创建Agent实例

⚠：如果您使用Colab或Jupyter一边阅读本文档一边实践，在每次重新启动服务时，需要先运行安装和下面这段配置和创建Agent实例的代码块，以确保后续代码块中的agent实例能够正常工作。

ℹ：如果您在后续代码执行中，需要重置agent实例的状态，也可以通过重新运行本段代码的方式重置

```
In [ ]: import Agently

# 创建一个Agent工厂实例
agent_factory = (
    Agently.AgentFactory()
    # 给Agent工厂实例提供设置项：
    ## 将默认模型系列设置为智谱系列
    .set_settings("current_model", "ZhipuAI")
    ## 提供你的智谱线上模型API-KEY
    .set_settings("model.ZhipuAI.auth", { "api_key": "" })
    ## 模型默认为GLM-4，如果想要指定为GLM-3-turbo可以解除下面代码的注释
    #.set_settings("model.ZhipuAI.options", { "model": "glm-3-turbo" })
)

# 从Agent工厂实例中创建Agent实例
## 你可以创建很多个Agent实例，它们都会继承Agent工厂的设置项
## 也就是说，你不需要再为每一个Agent实例指定模型、输入授权
agent = agent_factory.create_agent()

## 当然，你也可以为特定的Agent实例指定它独特的配置项
## 例如，我们让下面这个agent使用GLM-3-turbo模型进行驱动
agent_GLM_3 = (
    agent_factory.create_agent()
    .set_settings("model.ZhipuAI.options", { "model": "glm-3-turbo" })
)
```

开始使用Agent实例吧

```
In [ ]: result = (
    agent
    # 你也可以更换成agent_GLM_3看看输出效果
    #agent_GLM_3
        # 支持str格式输入
        .input("给我输出3个单词和2个句子")
        # 支持K-V式参数输入
        .instruct("输出语言", "中文")
        # 支持dict格式输入
```

```
# 并且通过框架特有的(<type>, <desc>, 可省略) 语法格式描述生成元素
.output({
    "单词": [("str",)],
    "句子": ("list",),
})
.start()
)
print(result)
```

>>>运行结果样例：

```
{'单词': ['苹果', '电脑', '学习'], '句子': ['我喜欢吃苹果。', '他用电脑工作。']}
```

常见问题

1. 只能使用一种模型吗？

当然不是，你可以通过阅读[开发功能说明文档](#)的《Model Request》章节查看目前框架已经适配的模型，并按说明进行配置。你可以放心切换模型，而不用担心已经使用框架表达方式写好的业务流程代码，框架已经帮助你做好了业务流程代码和模型适配之间的依赖解耦。

如果你希望使用本地启动的开源模型，还可以参考[使用Agently x X inference的方案使用本地开源模型的案例说明](#)进行使用。

目前已经支持的商用模型API包括OpenAI GPT、Google Gemini、百度文心一言、智谱GLM等。

同时，如果觉得有必要，你甚至可以为不同的Agent实例设置不同的模型进行驱动，以在不同场景下发挥不同模型的最大能力，以及优化不同场景下的业务逻辑开销，实现更优的成本收益比。

2. 支持设置代理（Proxy）吗？

因为特殊的原因，在某些地区如果需要访问模型的在线API，需要使用代理（Proxy），Agently框架为这种特殊场景提供设置代理（Proxy）的支持，你只需要使用以下方式进行设置即可：

```
# 你可以对Agent工厂实例进行设置
agent_factory.set_proxy("http://127.0.0.1:7890") # <- 更换为你的本地代理地址
# 你也可以对单个agent实例进行设置
agent.set_proxy("http://127.0.0.1:7890")
# 当然了，你也可以通过类似的方式，为单个agent指定模型并设置代理
(agent
    .set_settings("current_model", "Google")
    .set_settings("model.Google.auth.api_key", "YOUR-GOOGLE-API-KEY")
    .set_proxy("http://127.0.0.1:7890")
)
```

目前支持使用代理的模型包括：`OpenAI GPT`、`Google Gemini`

3. 我的应用只需要使用一个Agent实例，还需要使用Agent Factory这么复杂的创建方式吗？

Agent工厂这种设计模式主要是为了方便将配置信息传递给所有由工厂创建的Agent实例，如果你确定只需要使用一个Agent实例，下面这种创建方式会是一种更简单的方案：

```
In [ ]: import Agently

agent = (
    Agently.create_agent()
        .set_settings("current_model", "ZhipuAI")
        .set_settings("model.ZhipuAI.auth", { "api_key": "" })
)
```

4. Agent实例上只有.input(), .instruct(), .output()这么简单的几个操作指令吗？

当然不止，上面的案例只是展示了最基本的交互指令，但它已经能够说明框架的基本特性： 你可以在代码中创建agent对象实例，并像请求函数一样和它交互。

接下来我们会在本文档中逐步介绍更多好玩的特性和用法。

如果你非常好奇Agent实例上到底有多少可操作的指令，可以尝试运行下面的代码块得到答案：

```
In [ ]: agent.alias_manager.print_alias_info()
```

并且，这些指令可以随着Agent能力组件插件的添加而持续增加，让你的Agent实例变得更加强大。

第二部分：

请求方法、指令构造技巧及 输出结果处理

第三章：Agent实例的基础请求指令

和某些Agent框架不同，Agently框架提供的Agent实例在创建时并不会限定Agent实例的工作目标，要让Agent进行规划（Planning）、行动（Action）还是批判性思考（Critical Thinking），都是由开发者决定的。

Agently框架将Agent实例与模型进行的一次主要请求交互 见注1 视作一个原子节点，通过.start()命令触发。通过将多个原子节点串联，要如何让Agent实例行动起来，这就看开发者如何对Agent实例发起请求指令了。

注1：之所以使用“主要请求交互”这种说法，是因为在某些请求交互过程中，会触发对模型的前置请求，例如工具使用时，在对输入进行正式回复前，会通过模型进行工具使用规划和使用参数生成，并将工具生成结果结合到正式回复请求中，在这种情况下，“主要请求交互”并不等于“只对模型进行一次请求”。

基础请求指令代码样例

在进一步解释相关概念前，我们先来看下面的代码样例：

```
In [ ]: result = (
    agent
        # general: agent应该知道的全局信息，通常被视作system prompt或类似位置的信息
        .general("输出规定", "必须使用中文进行输出")
        # role: agent自身的角色设定信息
        .role({
            "姓名": "Agently小助手",
            "任务": "使用自己的知识为用户解答常见问题",
        })
        # user_info: agent需要了解的用户相关的信息
        .user_info("和你对话的用户是一个只具有Python编程基础知识的入门初学者")
        # abstract: 对于之前对话（尤其是较长对话）的总结信息
        .abstract(None)
        # chat_history: 按照OpenAI消息列格式的对话记录list
        ## 支持:
        ## [{"role": "system", "content": ""}, {"role": "assistant", "content": ""}, {"role": "user", "content": ""}]
        ## 三种角色
        .chat_history([])
        # input: 和本次请求相关的输入信息
        .input({
            "question": "请问Python中协程和线程的关系和区别是什么？",
            "reply_style_expect": "请用对编程概念零基础的人能理解的方式进行回复"
        })
        # info: 为本次请求提供的额外补充信息
        .info("协程相关关键词", ["asyncio", "coroutine", "event loop"])
        .info("线程相关关键词", ["threads", "queue"])
        # instruct: 为本次请求提供的行动指导信息
        .instruct([
            "请使用{reply_style_expect}的回复风格，回复{question}提出的问题",
        ])
        # output: 对本次请求的输出提出格式和内容的要求
        .output({
            "reply": ("str", "对{question}的直接回复"),
            "next_questions": ([
                ("str",
                    "根据{reply}内容，结合{user_info}提供的用户信息，" +
                    "给用户推荐的可以进一步提问的问题"
            ])
        })
    )
)
```

```

        )], "不少于3个"),
    })
    # start: 用于开始本次主要交互请求
    .start()
)
print(result)

```

>>>运行结果样例：

{'reply': '在Python中，协程和线程都是用来实现多任务的方法，但它们有一些关键的不同。想象你正在组织一场多人篮球赛，线程就像是一个个球员，他们可以同时上场打球，但是一次只能有一个球员控制球。协程更像是一种传球策略，球员们可以快速传递球，而且传球的过程中其他球员也可以做其他动作，这样就能更高效地利用球场。\\n\\n - 线程是由操作系统管理的，就像球员需要依赖场上的规则一样。每个线程都是一个独立的执行流程，但是线程的创建和上下文切换开销相对较大。\\n - 协程是Python编写者在代码层面上控制的，就像是球员之间商量好的传球策略。协程可以在单个线程内被挂起和恢复，并且由asyncio库的事件循环来调度，这样可以减少线程切换的开销。\\n\\n主要的区别包括：\\n - 线程是并行执行的，协程是协作式多任务。\\n - 线程有更重的资源开销，协程更轻量级。\\n - 线程受到GIL（全局解释器锁）的限制，协程可以通过事件循环更好地利用单核CPU。\\n\\n那么，接下来你可能对这些话题感兴趣：\\n 1. 什么是asyncio和事件循环？\\n 2. 如何在Python中创建和使用协程？\\n 3. Python的全局解释器锁（GIL）是什么，它对多线程有什么影响？\\n 4. 在实际项目中，我应该使用线程还是协程呢？'，'next_questions': ['我能举一个使用协程的简单例子吗？'，'Python中的async和await关键字是如何工作的？'，'如果我想深入了解线程，有哪些好的学习资源或实践项目推荐？']}}

基础请求指令列表

- `.general()` : agent应该知道的全局信息，通常被视作system prompt或类似位置的信息
- `.role()` : agent自身的角色设定信息
- `.user_info()` : agent需要了解的用户相关的信息
- `.abstract()` : 对于之前对话（尤其是较长对话）的总结信息
- `.chat_history()` : 按照OpenAI消息列格式的对话记录list
- `.input()` : 和本次请求相关的输入信息
- `.info()` : 为本次请求提供的额外补充信息
- `.instruct()` : 为本次请求提供的行动指导信息
- `.output()` : 对本次请求的输出提出格式和内容的要求
- `.start()` : 开始本次主要交互请求

基础请求指令用法

1. 并不是每次请求都需要把所有指令都用上，按你需要的选择指令即可，
`agent.input().start()` 是最简单的请求方案，使用这种方案和自行构造prompt请求模型的效果是一样的；
2. 除了 `.chat_history()` 指令必须传递消息列list，`.start()` 通常不传递值之外，其他指令可以接受几乎任何一种常见格式的数据，如str, bool, int, float, dict, list, set等；
3. 如果想要对某一类指令传递多个信息，可以使用**Key-Value**方式进行传递，如上面例子中的 `.info()` 部分，也可以通过list方式传入多条信息，如上面例子中的 `.instruct()` 部分；
4. `.output()` 指令如果传入str字符串，会正常返回一段字符串，如果传入dict字典或是list列表，则可以使用**Agently**框架提供的结构表达语法格式进行表达，获得对应结构的生成结果，详见下一小节。

使用Output输出结构表达语法格式

在代码开发过程中，我们往往需要使用结构化的数据而非自然松散的字符串文本来进行信息存储和传递。这类结构化的数据往往还不是只有一个层级的简单字典(dict)或是只存放一些字符串的列表(list)，而是一种更加复杂综合的结构。

例如，在上面的例子中，我们获得的结果是这样的：

```
{ 'reply': '在Python中...',
  'next_questions': [
    '我能举一个使用协程的简单例子吗？',
    'Python中的async和await关键字是如何工作的？',
    '如果我想深入了解线程，有哪些好的学习资源或实践项目推荐？'
  ],
}
```

这是一个复杂的字典结构，通过Output输出结构表达语法，我们可以直接从请求中使用 `result = agent....start()` 变量赋值的方式回收这个字典结果，并且用 `result["reply"]` 等方式直接使用字典中的字段值。

要实现这样的输出效果，我们需要让模型理解，在生成结果的时候，我们需要得到一个具有两个字段的字典结果。其中在 `reply` 字段中，我们需要得到一个长文本字符串，用于存放对用户问题的回复答案。同时，在 `next_questions` 字段中，我们需要得到一个列表，用于存放针对本次问答可以进一步提出的问题清单，且我们希望这个问题清单里的问题不要少于3个。

那么我们在使用Agently框架的Output输出结构表达语法时，应该如何思考呢？

首先，确定期望输出的数据结构：

按照上面的描述，我们期望得到的数据结构如下：

```
{
  "reply": str,
  "next_questions": [str]
}
```

如果我们将 `str`、`int`、`bool`、`float` 等类型的数值看作输出中的带有具体内容的值节点，那么上面这个结构则表达了我们希望输出的结构特征。

接下来，我们使用输出结构表达语法中元组(tuple)的特殊用法来表达对值节点的输出期望：

因为在输出的数据中，我们几乎不会用到元组(tuple)这种数据结构，因此Agently框架赋予了元组新的含义：通过 `("<类型描述>", "<输出内容期望>")` 的格式，描述对具体内容的值节点的输出期望。

例如，我们希望在 `reply` 节点中获得对本次提问的直接回复，我们就可以做如下表达：

```
# ("<类型描述>", "<输出内容期望>")
("str", "对本次提问的直接回复")
```

如果我们希望做更明确的信息指向，比如希望对“本次提问”到底指的是哪部分信息做出明确指向，我们可以使用 `{}` 进行指向标注，如果我们希望明确表达这个具体的信息是从哪部分信息块中来的，可以使用 `{信息块.具体字段名}` 的方式进行表达：

```
("str", "对{input.question}的直接回复")
```

可能也有人注意到，有时候我们需要对list结构做一些额外的说明，比如约定list结构中输出的结果的数量，我们也可以将结构嵌入元组表达中，例如：

```
([], "最少输出3个结果")
```

而在元组中的结构，还可以继续嵌入元组表达：

```
([("str", "根据{reply}可以进一步提出的问题")], "最少输出3个结果")
```

最后，整合上面两步，形成完整的输出结构表达，并放入`.output()`请求中：

```
.output({
  "reply": ("str", "对{input.question}的直接回复"),
```

```
"next_questions": (  
    [ ("str", "根据{reply}可以进一步提出的问题") ],  
    "最少输出3个结果"  
)
```

第四章：基于基础请求指令的高级Prompt构造技巧

众所周知，基于Transformer的模型在生成下一个输出块时，会使用所有的输入提示信息加上已经生成的输出块内容作为计算依据。正是基于这样的特性，使得我们通过合理构造输出顺序，通过前几步生成的内容提示下一步生成的内容成为可能，可以将这种构造技巧视作一种思维链（CoT）方法。通过这种构造方法，我们可以实现行动流提示、分支逻辑判断、自我批判修正、确保关键信息纯净度等好玩的高级技巧。

行动流提示

```
In [ ]: result = (
    agent
        .input("我在哪里可以了解黑暗之魂3和GTA6的发售日期，并完成购买或者预购？")
        .output({
            "info_list": [
                {
                    "知识对象": ("str", "回答{input}问题时，需要了解相关知识的具体对象"),
                    "关键知识点": ("str", "回答{input}问题时，需要了解的关键知识"),
                    "是否了解": ("bool", "判断你是否确信自己了解{关键知识点}的知识，如果不了解，输出false")
                }
            ],
            "sure_info": ("str", "根据{info_list}给出回复，只展开详细陈述自己了解的关键知识点"),
            "uncertain": ("str", "根据{info_list}向用户说明自己不了解的信息"),
        })
        .start()
)
print(result)
```

>>>运行结果样例：

```
{'info_list': [{ '知识对象': '黑暗之魂3', '关键知识点': '发售日期', '是否了解': True}, { '知识对象': 'GTA6', '关键知识点': '发售日期', '是否了解': False}, { '知识对象': '黑暗之魂3', '关键知识点': '购买或预购方式', '是否了解': True}, { '知识对象': 'GTA6', '关键知识点': '购买或预购方式', '是否了解': False}], 'sure_info': '黑暗之魂3 (Dark Souls III) 已经在2016年3月24日发售。您可以通过各大游戏平台如Steam、PlayStation Store、Xbox Live等进行购买或查询是否还有预购选项。', 'uncertain': '至于GTA6 (Grand Theft Auto VI) 的发售日期和购买方式，目前我没有确切的信息。通常，此类信息会在游戏官方渠道或开发商Rockstar Games的官方公告中发布，建议您关注相关动态以获取最新消息。'}
```

分支逻辑判断

```
In [ ]: result = (
    agent
        .input("帮我查一个新闻")
        .output({
            "is_info_complete": ("bool", "判断{input}提供的信息是否完整齐备，能够被执行"),
            "question": ("str", "如果{is_info_complete}==false，输出你想要询问的问题，否则输出null"),
            "reply": ("str", "如果{is_info_complete}==true，输出你的回答，否则输出null"),
        })
        .start()
)
print(result)
```

>>>运行结果样例：

```
{'is_info_complete': False, 'question': '请告诉我您想查询的具体新闻内容或关键词。', 'reply': None}
```

自我批判修正

```
In [ ]: result = (
    agent
        .input({
            "target": "进入被巨石封堵的洞穴",
            "items": ["勺子", "筷子"],
        })
        .output({
            "action": ("str", "用最大胆的方式给出利用{items}完成{target}的方法"),
            "can_do": ("bool", "请用常识判断{action}的可行性"),
            "can_do_explain": ("str", "如果{can_do}==false,给出你的判断理由"),
            "fixed_action": (
                "str",
                "如果{can_do}==false,结合{can_do_explain}和{items}" +
                "重新思考完成{target}的方法"
            )
        })
        .start()
)
print(result)
```

>>>运行结果样例：

```
{'action': '使用勺子和筷子撬动巨石，清除洞穴入口的障碍。', 'can_do': False, 'can_do_explain': '勺子和筷子作为工具，其强度和尺寸都不足以撬动巨石。', 'fixed_action': '尝试使用勺子和筷子挖掘巨石周围的土壤或小石块，寻找可能的缝隙或空洞，从而逐步扩大洞穴入口的空间，或者收集可燃烧的物质制作简易火堆，用烟雾逼迫洞穴内的空气流动，以寻找其他未被堵塞的通道。'}
```

确保关键信息纯净度

有时候，我们直接请求模型生成一些关键结果（比如代码、翻译结果）时，会遇到模型话痨的情况，无论我们怎么说明不要做额外的解释说明，模型依然“热情”地希望帮助我们更好地理解它的答案，输出额外信息。

通过构造结构化输出，我们可以让模型把额外的输出信息放到其他位置，从而确保指定位置的信息的纯净度。

```
In [ ]: result = (
    agent
        .input("翻译: PROMPT for Language Models")
        .instruct("目标语言: 中文")
        .output({
            "translation": ("str", "对翻译目标的翻译结果"),
            "explanation": ("str", "对翻译内容的补充说明"),
            "examples": [("str", "使用{translation}内容造句")],
        })
        .start()
)
print(result)
```

>>>运行结果样例：

```
{'translation': '语言模型的提示', 'explanation': 'PROMPT 在此上下文中指的是为语言模型提供输入或指令的简短文字，以便模型可以根据这些提示生成相应的文本或完成特定的任务。', 'examples': ['在为聊天机器人编写脚本时，我需要设计一些有效的提示来引导语言模型生成合适的回答。', '通过设置不同的提示，我能够探索语言模型在诗歌创作和故事编写方面的多样性。', '提示可以帮助语言模型在处理用户查询时更准确地理解用户的意图。']}
```

第五章：流式输出和事件监听处理器

在Agently框架中，绝大部分适配的模型^{注1}都支持流式输出，并且已经在框架请求逻辑中对流式输出的数据块进行了清洗，通过`response:delta`事件抛出仅包含增量文本块的数据。要如何接收这些流式输出数据，对它们进行解析、安全判断、打印或是向前端转发？这就需要用到我们在本章介绍的事件监听处理器能力了。

^{注1}: 部分embedding模型和生图模型不支持流式输出，但这种情况仅是不抛出`response:delta`事件，从而让监听该事件的处理器不运行，并不会对整体逻辑造成影响或是抛出错误。

两种流式输出的打印示例

使用Colab或是Jupyter工具，可以自行运行下面两段示例，看到流式输出的效果。

```
In [ ]: """示例1：使用.on_delta()指令快速打印"""
result = (
    agent
        .input("首先生成三个关于颜色的词语，然后用这三个词语造两个句子")
        # 在本示例中我们使用了lambda函数，你也可以定义更复杂的处理函数
        # 并将处理函数传给.on_delta()指令
        .on_delta(lambda data: print(data, end=""))
        .start()
)
# 当然.on_delta()指令不会影响result变量获得最终的结果值
#print(result)
```

>>>运行结果样例：

三个关于颜色的词语：碧绿、湛蓝、火红。

第一个句子：春天来了，碧绿的草地上一片生机盎然，湛蓝的天空下孩子们欢快地玩耍。

第二个句子：国庆节那天，火红的国旗在湛蓝的天空中飘扬，人们脸上洋溢着自豪的笑容。

```
In [ ]: """示例2：使用@agent.on_event()函数装饰器打印"""
@agent.on_event("delta")
def print_delta(data: str):
    print(data, end="")

result = (
    agent
        .input("首先生成三个关于颜色的词语，然后用这三个词语造两个句子")
        .start()
)
```

>>>运行结果样例：

三个关于颜色的词语：

1. 碧绿
2. 深红
3. 鹅黄

两个句子：

1. 春天来了，树叶渐渐变得碧绿，给人一种生机勃勃的感觉。
2. 她穿着一袭深红色的长裙，走在鹅黄色的沙滩上，显得格外引人注目。

事件监听处理器

上面两个示例中，将流式输出打印出来的那个函数，就是我们所说的事件监听处理器（Event Listener），除了将流式输出数据打印出来之外，我们也可以定义其他处理逻辑，例如对流式数据进行 WebSocket转发。

同时，Agently框架也提供了不仅限于 `response:delta` 的其他事件供开发者使用，了解更多可以参考案例：通过Agent指令或是函数装饰器使用事件监听器

通过WebSocket服务分发流式输出结果

下面再提供一个简单示例，演示如何将流式数据通过WebSocket转发输出：

（注意：因为Colab每次只能执行一段代码块，下面两段代码不能同时被执行，需要将下面两段代码块复制到本地环境运行测试）

```
In [ ]: """本示例通过tornado进行WebSocket服务"""
"""此段为服务端代码"""
import Agently
import asyncio
import tornado.ioloop
import tornado.web
import tornado.websocket
import json

agent_factory = (
    Agently.AgentFactory()
    .set_settings("current_model", "ZhipuAI")
    .set_settings("model.ZhipuAI.auth", { "api_key": "" })
)

agent = agent_factory.create_agent()

class WebSocketHandler(tornado.websocket.WebSocketHandler):
    def on_message(self, message):
        try:
            data = json.loads(message)
            print("Data Received:", data)
            result = (
                agent
                # 从传递过来的消息中读取"input"字段的值，传入.input()
                .input(data["input"])
                # 将流式输出结果用{ "event": "delta", "data": <输出块内容> }的格式返回
                .on_delta(lambda data: self.write_message(
                    json.dumps({
                        "status": 200,
                        "event": "delta",
                        "data": data,
                    })
                ))
                .start()
            )
            # 在全部生成结束后，用{ "event": "done", "data": <完整结果> }的格式返回
            self.write_message(
                json.dumps({
                    "status": 200,
                    "event": "done",
                    "data": result,
                })
            )
        except Exception as e:
            print(e)
```

```

        self.write_message(json.dumps({
            "status": 400,
            "msg": str(e),
        }))

class Application(tornado.web.Application):
    def __init__(self):
        handlers = [
            ("/app", WebSocketHandler),
        ]
        super(Application, self).__init__(handlers)

app = Application()
app.listen(1565)
print("WebSocket 服务已启动")
tornado.ioloop.IOLoop.current().start()

```

In []: """此段为模拟客户端代码"""

```

import asyncio
import aiohttp
from aiohttp_socks import ProxyConnector
import json

async def proxy_request():
    async with aiohttp.ClientSession() as session:
        async with session.ws_connect('ws://0.0.0.0:1565/app') as websocket:
            await websocket.send_str(json.dumps({
                "input": "你好"
            }))
            async for msg in websocket:
                response = msg.data
                print(json.loads(response))

asyncio.get_event_loop().run_until_complete(proxy_request())

```

更高级的流式输出处理工具：Segment [beta]

在第四章中我们提到，除了直接回复用户输入之外，我们还可以通过高级Prompt构造的技巧在单次主要请求中构造更复杂的思考过程，通过前置思考、自我批判修正等方式，进一步保证输出的准确性。

在Agent与用户的直接交互过程中，通常我们并不希望用户看到这些思考的过程，但是，我们同时还希望用户能够尽快看到他们能够看到的那部分信息的流式输出。

Segment能力组件插件就是为了解决这样的问题而设计的，具体使用方法请看下面的示例代码：

⚠: Segment是一个实验性方法（带有 `[beta]` 标签），可能在运行过程中出现错误，需要开发者进行容错处理

ℹ：根据开发实验，在Gemini Pro、文心4、GLM-4等模型中能够有较高的成功率，但在GPT-3.5-turbo-1106模型并不能很好运作。

In []: `def print_streaming_content(data: str):`
 `print(data, end = "")`

`user_input = input("[用户]: ")`

`result = (agent`
 `.input(user_input)`
 `.segment(`
 `"thinking_tips",`

```
"生成一段文字，用适当的态度表达对用户输入信息的关注，你正在思考如何回复，用换行符结尾",
    print_streaming_content,
    is_streaming = True,
)
.segment(
    "required_info_list",
[
    {
        "知识对象": ("str", "回答{input}问题时，需要了解相关知识的具体对象"),
        "已知信息": ("str", "根据{input}总结用户已经提供的信息"),
        "是否完备": ("bool", "判断你是否确信自己拥有{知识对象}的关键知识或信息，如果不
了解，输出false"),
        "关键知识点或信息补充": ("str", "如果{是否完备}==false，给出需要了解的关键知识
或需要用户提供的信息补充，否则输出空字符串'''"),
    }
],
)
.segment(
    "certain_reply",
    "根据{required_info_list}给出回复，展开详细陈述自己了解的关键知识点内容，用换行符结尾，
如果没有则输出空字符串''',
    print_streaming_content,
    is_streaming = True,
)
.segment(
    "uncertain_reply",
    "根据{required_info_list}的信息，向用户说明自己不了解的信息，请用户提供或自行查找，用换
行符结尾，如果没有则输出空字符串'''",
    print_streaming_content,
    is_streaming = True,
)
.segment(
    "next_topic_suggestions",
    "根据之前所有生成内容的信息，给出接下来可以进一步讨论的问题或话题建议，如果没有则输出空字符
串''',
    print_streaming_content,
    is_streaming = True,
)
.start()
)

print("\n-----\n[隐藏的思考信息]\n", result["required_info_list"])
```

>>>运行结果样例：

[用户]：我想要创作一篇小说

好的，您想创作一篇小说，这是一个很有创意的想法。我正在思考如何帮助您，请稍等片刻。

小说创作是一个涉及多个方面的过程，首先我们可以从以下几个方面入手：

1. 确定小说的题材，例如现实主义、奇幻、科幻等；
2. 设定故事背景，包括时代、地点等；
3. 设计主要人物及其性格特点；
4. 梳理故事情节，包括起承转合。

当然，如果您有其他方面的想法，也可以告诉我，我会尽力帮助您。

目前我还不太清楚您对小说的具体想法，比如您希望创作哪种类型的小说，故事背景和人物设定等。如果您能提供这些信息，我将更好地为您提供建议和帮助。

接下来，我们可以进一步讨论以下问题：

1. 您喜欢的小说类型和作者；
2. 您对故事背景和人物设定的初步设想；
3. 您希望在小说中表达的主题或思想。

这些问题将有助于我们更好地展开创作。

[隐藏的思考信息]

```
[  
 {  
   "知识对象": "小说创作",  
   "已知信息": "用户想要创作一篇小说",  
   "是否完备": false,  
   "关键知识点或信息补充": "需要了解用户对小说题材、故事背景、人物设定等方面的想法"  
 },  
 ]
```

第三部分： Agent实例

第六章：Agent实例的关键概念

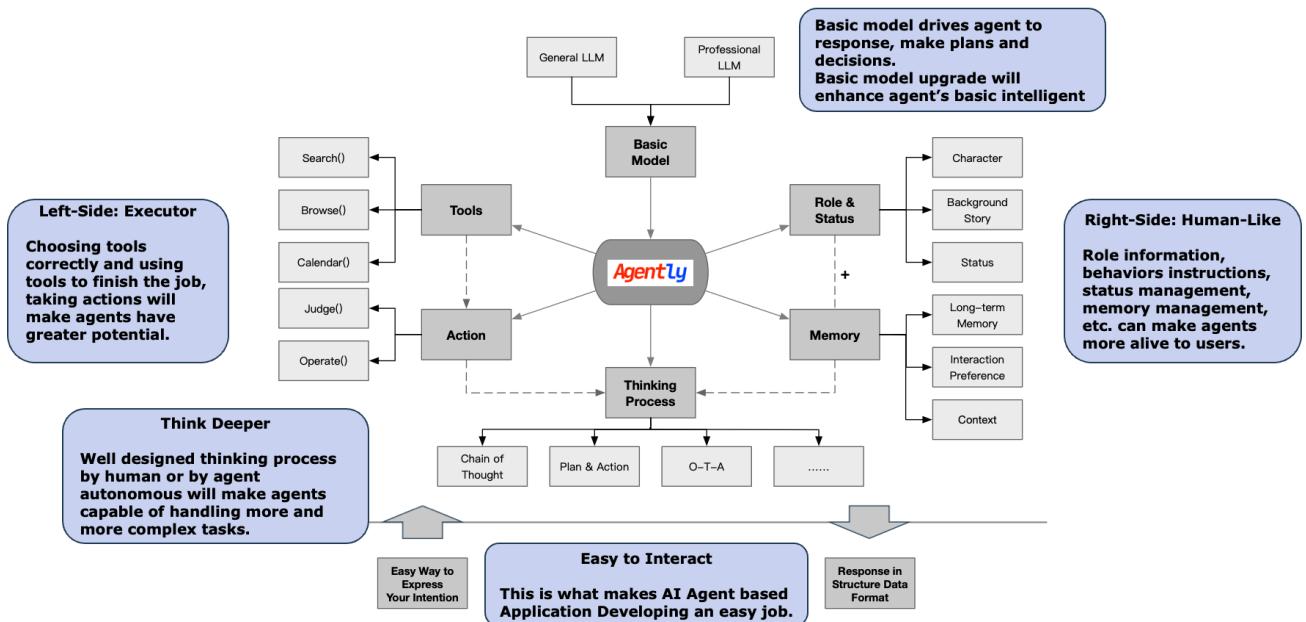
Agent实例：放大模型能力的结构体

在Agently框架的演进过程中，**Agent实例**是模型能力的放大器这个关键认知深切地影响了框架的设计思想。应该如何理解这个认知？

回想一下，当你最初开始使用ChatGPT或者与其类似的聊天机器人（Chat Bot）产品的时候，你对于在聊天框中输入信息并获得返回结果的期待是什么？是不是除了获得对当前一次输入问题的直接回答之外，还期望这次返回结果能够结合之前的几句对话记录，甚至是更远的几天、几周前的对话记录进行回答？是不是还期望聊天机器人能够具备更加生动的形象，并且更了解你的喜好和个人情况，给出更具有针对性的回答？是不是还期望它的回复内容中，除了从模型自身的预训练知识储备中抽取和组织信息，也希望它能够更多地结合外界的信息进行回答？是不是甚至期待它除了给出一条回复之外，还能够真切地帮你写下一段笔记、记录一个提醒、定上一个闹钟？

当然了，如果你已经看到了这个章节，你也一定明白，这些期待并不能够通过直接对模型发起一次请求而得到实现，在模型之上，似乎需要有一个结构体，通过一定的结构设计和运行逻辑，来完成这些期待，对于使用者而言，他们仍然可以将这个结构体视作一个加强版的模型，通过简单的输入交互向它发起请求（例如我们在第二部分提供的方法）。在Agently框架中，这样的结构体就是**Agent实例**。

下图展现了Agently框架对于Agent实例这个结构体的能力模块和交互方式的思考和期待：



📚 衍生阅读：

如果你还在好奇为什么这个结构体会被设计成这样，或是为什么要把这样的结构体称作Agent，可以进一步阅读下面的两篇文档：

[Agent: Model Capability Amplifier - Mo Xin, Agently Team](#)

[LLM Powered Autonomous Agents - Lilian Weng, Open AI](#)

Agent实例的运行逻辑

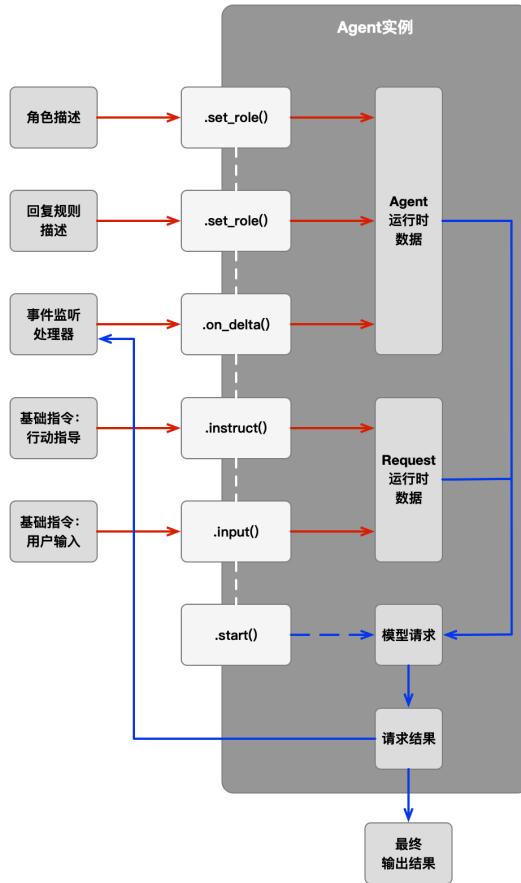
在理解了Agently框架对Agent实例这个结构体的基础设计思想之后，接下来我们来进一步说明在代码编写和运行过程中，Agent实例到底进行了哪些工作？

下面的代码例子精简自 [Agently Playground](#) 中的幼儿教师案例，如果对该案例完整说明及运行结果感兴趣，可以点击查看原始文档。

```
In [ ]: demo_agent = agent_factory.create_agent()

result = (
    demo_agent
        # Agent Component 能力插件提供的交互指令
        .set_role("角色", "你是一个幼儿教师")
        .set_role(
            "回复规则",
            "你需要将晦涩难懂的专业知识理解之后转化成小孩子能听懂的故事讲给用户听，" +
            "注意，虽然是讲故事，但是要保证专业知识的准确真实"
        )
        .on_delta(lambda data: print(data, end=""))
    # 基础请求指令
    .instruct("如果搜索结果中包含较多内容，请尽可能将这些内容有条理系统地转化成多段故事")
    .input("天空为什么是蓝色的")
    # 请求启动指令
    .start()
)
print("\n[最终回复]: ", result)
```

上面这段代码的运行过程数据流转时序可以用下图简单描述：



图示说明：

- 通过Agent实例提供的操作指令接口，开发者可以将业务数据、事件监听处理器等内容传递给Agent实例，Agent实例将会将这些内容存储到Agent实例或是Request请求的运行时数据中，在本章节的后续部分会对运行时数据进行更详细的说明；
- 除了Agent实例自带的基础请求指令外，Agent实例上还可以使用很多其他的指令，这一点我们在第二章的常见问题中也有提到，这些指令通常是由Agent能力组件（Agent Component）这种插件提供

的，因此，对于应用开发者而言，通常不需要自己去操作运行时数据，而是通过能力组件开发者在插件逻辑中对运行时数据进行交互，而能力组件本身的逻辑，就是实现上面结构体某项能力的方案；

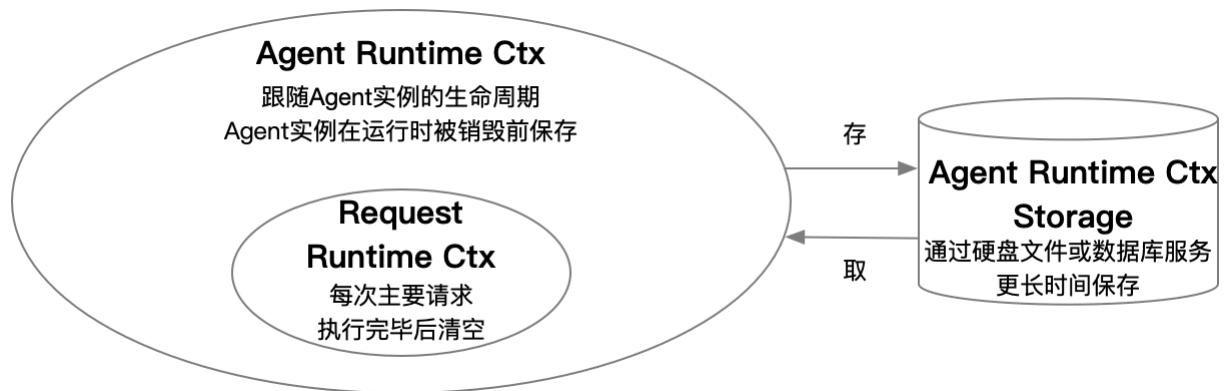
- 在所有操作指令输入完毕后，通过 `.start()` 将正式让Agent实例发起模型请求，但这个模型请求并不只是简单地将用户输入传递给模型，而将会收集之前各种指令传递给运行时数据的信息，经过 Agently 框架核心逻辑的数据整理后，再对模型发起请求；
- 最后，从模型侧返回的请求结果会分发给事件监听处理器或是作为 `.start()` 的最终返回值传递给应用开发者，当然，对于能力组件开发者而言，他们还可以对返回值进行其他处理操作，例如在请求结果返回后再次更新运行时数据以影响下一次请求等。

Agent实例运行时数据的生命周期

Agent实例运行时数据的生命周期是Agently框架中的一个重要概念，也是Agent能力组件的运行基础，本章将用较大篇幅对概念进行说明，这对理解后续章节介绍的能力的工作原理会有很大帮助。

在第三章中，我们介绍了Agent实例的基础请求方法，可能你已经注意到，这些请求指令中包含的信息，在下次请求中并不会被自动携带，如果我们希望在下次请求中继续使用这些信息，只能再次重复发送这些指令。

事实上，这也是除了遵循LLM Powered Agent的模块结构设计外，Agent实例结构的另一个重要特点：运行时数据生命周期管理。



如上图所示，实际上，Agent实例在运行时，会有三类不同生命周期的运行时数据：

- **主要请求运行时数据 (Request Runtime Ctx)**：这类运行时数据会传递给对模型的主要请求，并且在主要请求执行完毕后进行清空，通过基础请求指令传入的数据基本都属于这类运行时数据；
- **Agent实例运行时数据 (Agent Runtime Ctx)**：这类运行时数据会持续跟随在代码运行阶段的Agent实例上，属于Agent实例属性的一部分，它们会持续影响Agent实例的行为，直到它们被更改或是Agent实例被回收销毁，在Agently框架中，通常由Agent能力组件 (Agent Component) 这类型的插件对Agent实例运行时数据进行创建、更新、删除等管理动作，并在每次执行模型主要请求时，将插件管理的这部分信息传递给模型，从而保证Agent行为在生命周期内的一致性；
- **Agent实例运行时数据的长时固化存储 (Agent Runtime Ctx Storage)**：有的时候，我们并不希望Agent实例运行时数据随着脚本运行结束被销毁，这时候我们就需要将这些运行时数据存储到本地，并在需要时将这些数据读取出来，在Agently框架中，也提供了一套对应的方法来对Agent实例运行时数据进行保存，将在下一小节的示例中进行演示。

Agent实例运行时数据的存取示例

In []:

```
# 要确保Agent实例运行时数据能够被正确取出
# 我们需要在创建Agent实例时，传入agent_id
# 这个agent_id可以由开发者自行命名管理
# 或是在首次创建时保存框架自动创建的id值
my_agent = agent_factory.create_agent("agent42") #<- 这里传入的字符串就是agent_id
```

```
# 当.create_agent()时传入了agent_id
# Agently框架会自动查询存储区域中是否存在已经保存的
# 对应agent_id的Agent实例运行时数据
# 如果找到，则会自动将这些数据恢复至新创建的实例中

# 通常情况下，你只需要正常使用Agent实例即可，
# 不用关心实例运行时数据的情况，因为这些信息往往由
# Agent能力组件插件进行管理，你并不需要知道
# 但你也可以通过agent.agent_runtime_ctx
# 对Agent实例运行时数据进行操作，例如：
my_agent.agent_runtime_ctx.set("my_runtime_key", "my_runtime_value")

# 接下来我们通过agent.save()指令确保运行时数据被保存
my_agent.save()
```

Out[]:

```
>>>运行结果样例：
```

```
<Agently.Agent.Agent at 0x7aeabc856620>
```

In []:

```
# 当我们再次创建使用这个agent_id的agent时
my_agent42 = agent_factory.create_agent("agent42")

# 就可以取回保存时的所有Agent实例运行时数据了
print(my_agent42.agent_runtime_ctx.get("my_runtime_key"))
```

```
>>>运行结果样例：
```

```
my_runtime_value
```

不过，正如上面的代码示例中所说，作为应用开发者，你几乎不需要考虑对Agent实例运行时数据的直接操作，只需要知道这是接下来章节中我们介绍的各种Agent能力增强组件带来的功能的运行基础就好了。

第七章：Agent实例的多轮对话及对话记录管理

在开始介绍Agently为Agent实例提供的其他能力之前，先讲讲多轮对话这个最基本的能力。对Agent实例进行多次请求，是我们需要对Agent实例运行时数据进行管理的基础，也是后续绝大部分Agent能力插件需要存在的原因。

最基础的多轮对话方案

Agently框架通过Session能力增强插件，提供了通过自动持续记录会话消息列的方式，支持多轮对话的方案。下面是快速搭建的代码示例：

```
In [ ]: # 打开自动会话记录管理能力
agent.active_session("example_session") #<-如果要支持本地长时间存储会话记录，这里需要指定
session_id
while True:
    ## 获取用户输入
    user_input = input("[用户]: ")
    ## 设置终止关键词
    if user_input == "#exit":
        print("再见~👋")
        break
    print("[助理]: ", end="")
    ## 执行语言模型处理
    ## 通过.on_delta()方法监听流式输出，并进行打印
    result = (
        agent
            .input(user_input)
            .on_delta(lambda data: print(data, end=""))
            .start()
    )
    print("")
# 在结束时提示会话结束，执行保存本地逻辑
# 如果不希望长时间保存对话内容，则不需要下面这行
agent.stop_session()
```

>>>运行结果样例：

[用户]：给我出一个字谜吧，不要告诉我谜底
[助理]：好的，这是一个字谜：

树上开满小白花，秋天结果似玛瑙。

这个字谜描述的是一种植物，你能猜出是什么字吗？

[用户]：有提示吗
[助理]：当然可以给一个提示：

这种植物的名字由两个部分组成，谜语中描述的是它的果实，而它的另一个部分则与木材有关，常常被用来制作家具。

[用户]：不知道耶，告诉我答案
[助理]：这个字谜的答案是“柿”。解释如下：

“树上开满小白花”指的是柿子树开花时的景象，柿子的花是白色的。

“秋天结果似玛瑙”则形容柿子成熟时的颜色，红润如同玛瑙一样。

而“柿”字由“木”字旁和“市”字组成，与提示中提到的“与木材有关”相符合。

[用户]：#exit
再见~👋

Out []: >>>运行结果样例：

```
<Agently.Agent.Agent at 0x7ae55955ba0>
```

结合框架其他能力，构建更复杂的多轮对话

```
In [ ]: def print_streaming_content(data: str):
    print(data, end="")

agent.set_settings("is_debug", False)
agent.active_session()
while True:
    user_input = input("[用户]: ")
    if user_input == "#exit":
        print("再见~👋")
        break
    print("[助理]: ", end="")
    result = (
        agent
            .input(user_input)
            .segment(
                "thinking_tips",
                "生成一段文字，用适当的态度表达对用户输入信息的关注，你正在思考如何回复，用换行符结尾",
                print_streaming_content,
                is_streaming = True,
            )
            .segment(
                "required_info_list",
                [
                    {
                        "知识对象": ("str", "回答{input}问题时，需要了解相关知识的具体对象"),
                        "已知信息": ("str", "根据之前所有对话历史，总结已知信息"),
                        "是否完备": ("bool", "判断你是否确信自己拥有{知识对象}的关键知识或信息，如果不了解，输出false"),
                        "关键知识点或信息补充": ("str", "如果{是否完备}==false，给出需要了解的关键知识或需要用户提供的信息补充，否则输出空字符串''"),
                    }
                ],
            )
            .segment(
                "certain_reply",
                "根据{required_info_list}给出回复，展开详细陈述自己了解的关键知识点内容，用换行符结尾，如果没有则输出空字符串''",
                print_streaming_content,
                is_streaming = True,
            )
            .segment(
                "uncertain_reply",
                "根据{required_info_list}的信息，向用户说明自己不了解的信息，请用户提供或自行查找，用换行符结尾，如果没有则输出空字符串''",
                print_streaming_content,
                is_streaming = True,
            )
            .segment(
                "next_topic_suggestions",
                "根据之前所有生成内容的信息，给出接下来可以进一步讨论的问题或话题建议，如果没有则输出空字符串''",
                print_streaming_content,
                is_streaming = True,
            )
            .start()
    )
    print("")
```

>>>运行结果样例：

[用户]：我最近总觉得有点不舒服，我应该怎么自己判断一下发生了什么

[助理]：

你的问题非常值得关注，我正在思考如何帮助你分析这种情况。请稍等，我将提供一些可能有用的信息。

根据你的描述，我无法直接判断出你的不适原因，但我可以建议你注意观察以下情况：具体的症状表现，如头痛、胃痛、疲劳等；这些症状出现的时间、频率和持续时间；以及是否有任何特定事件或行为可能导致这些不适。这些信息将有助于进一步了解你可能面临的问题。

为了更准确地提供帮助，我需要你提供更多关于不适的具体信息。请描述你的症状，以及它们出现的时间和持续时间。如果有任何你认为相关的日常习惯或事件，也请告知。

一旦你提供了更多的症状信息，我们可以讨论一些可能的健康问题和自我照顾的方法。同时，如果情况严重或持续不适，建议及时咨询专业医生。

[用户]：肩膀有时候很疼，而且很容易疲劳，工作一小会就想睡觉

[助理]：

你的症状很有意思，肩膀疼痛和疲劳可能是多种因素引起的。我正在思考如何更好地帮助你，请稍等片刻。

根据你的描述，肩膀疼痛和疲劳可能与工作姿势、肌肉紧张或生活习惯有关。建议注意调整工作环境，定时休息，并进行适当的肩颈拉伸和锻炼。

为了提供更具体的建议，我需要了解更多信息，比如你的工作性质、每天的工作时长、休息习惯以及是否有其他身体症状。

你可以分享更多关于你的日常习惯和生活方式的信息，这样我们可以讨论一些可能改善你肩膀疼痛和疲劳的方法。同时，如果症状持续，考虑咨询医生以获得专业的诊断和治疗建议。

[用户]：我是程序员，主要是面向电脑屏幕进行开发工作

[助理]：

你的职业背景和症状描述非常有趣，我正在思考这可能与你的工作习惯和身体状况有何关联，稍等一下，我将为你提供一些建议。

根据你的情况，长时间面向电脑屏幕工作可能导致肩颈肌肉紧张和疲劳。建议你注意调整工作姿势，每隔一段时间起身活动，进行肩颈拉伸，并确保你的工作环境有适当的光线和舒适的座椅。

为了提供更针对性的建议，我还需要了解你每天的工作时长、休息间隔以及你是否有尝试过任何缓解肩颈疼痛的方法。

接下来，我们可以讨论一些适合程序员的肩颈保健方法以及如何改善工作环境来减轻你的症状。如果情况没有改善，你也可以考虑咨询专业的医疗人员。

[用户]：每天大概连续工作8个小时左右，每1小时会休息5分钟，当然也不一定，有时候工作进入状态可以连续3个小时左右不动

[助理]：

你的工作习惯和症状描述提供了很多有价值的信息，我正在考虑这些因素可能如何影响你的肩颈疼痛和疲劳问题，请稍等片刻。

根据你的工作习惯，长时间保持同一姿势确实可能导致肩颈肌肉紧张和疲劳。建议你在工作期间尽量坚持每1小时休息5分钟的习惯，并且可以使用这个时间进行肩颈的拉伸和放松活动。此外，可以考虑调整你的工作椅和桌子高度，确保你的屏幕处于眼睛水平或略低的位置，以减少颈部的压力。

目前我对于你的情况已经有了一定的了解，但如果你能提供更多关于你工作环境的细节，比如椅子类型、屏幕尺寸和位置，以及你是否有任何其他健康问题，这些信息将有助于提供更具体的建议。

下一步，我们可以讨论一些具体的肩颈拉伸和锻炼方法，以及如何改善你的工作环境来减少不适。如果你对现有的缓解措施感到不确定或症状持续，我会建议你咨询专业的健康顾问或物理治疗师。
```

[用户]：能不能直接给我建议了？我不太好提供椅子、屏幕这些信息，没有测量的工具

[助理]：

你的问题非常有代表性，我正在思考如何在不依赖于具体工作环境参数的情况下，为你提供实用的建议。请稍等，我将为你整理一些信息。

基于你的情况，以下是一些通用的缓解肩颈疼痛的建议：

1. 定期进行肩颈拉伸，比如每隔一小时进行简单的拉伸运动。
2. 尝试改变工作姿势，比如站立工作一段时间，或者调整坐姿。
3. 确保在工作时有良好的背部支撑，以减少肩颈的压力。
4. 进行一些肩颈肌肉的强化练习，比如颈部后仰、肩部旋转等。
5. 注意休息，保证充足的睡眠，有助于肌肉的恢复。

请尝试这些建议，并观察是否有改善。

如果这些建议对你有帮助，我们可以进一步讨论如何将这些方法融入你的日常工作中。如果症状持续，我会建议你考虑咨询医生或专业的物理治疗师，以获得更个性化的诊断和治疗建议。

[用户]：#exit

再见~👋

## 在复杂场景下，自行管理对话记录

当然，在更加复杂的场景中，我们需要自己定制管理对话记录，而不是使用自动记录方案，Agently框架也为您手动管理对话记录提供了便捷的方法支持。

您可以通过 `.toggle_manual_chat_history(True)` 的方式打开手动管理对话记录的模式，然后参考下面的案例对对话记录进行手动管理：

```
In []: agent_with_manual_chat_history = agent_factory.create_agent()
使用.toggle_manual_chat_history(True)打开手动管理对话记录模式
agent_with_manual_chat_history.toggle_manual_chat_history(True)
使用.rewrite_chat_history([])重置对话记录为空
agent_with_manual_chat_history.rewrite_chat_history([])
将手动添加对话记录的顺序设置为严格模式（即对话记录严格遵循User-Assistant-User-...的顺序）
严格模式默认打开，因此在实际开发中，可以忽略下面一行命令
你也可以通过.toggle_strict_orders(False)的方式关闭严格模式
agent_with_manual_chat_history.toggle_strict_orders(True)
同样的，我们需要通过.active_session()开启在请求时自动插入之前存储的对话记录的能力
agent_with_manual_chat_history.active_session()
当然，如果你希望自主决定是否插入对话记录，可以将上面一行注释掉，然后往下看
while True:
 user_input = input("[用户]: ")
 if user_input == "#exit":
 print("再见~👋")
 break
 # 通过.instruct()方法使用同一个agent实例发起两次视角完全不同的主要交互请求
 print("[理性派]: ", end="")
 logical_result = (
 agent_with_manual_chat_history
 # 如果你没有使用.active_session()
 # 可以使用下面的方法自己决定是否插入已经存储的对话记录
```

```
is_shorten=True将帮助你将超长的对话记录进行截断
#.chat_history(
agent_with_manual_chat_history
.get_chat_history(is_shorten=True)
#)
 .input(user_input)
 .instruct("参考瓦肯星人👉， 使用理性和逻辑态度对问题进行分析，并给出你的建议")
 .on_delta(lambda data: print(data, end=""))
 .start()
)
print("\n[情绪动物]: ", end="")
emotional_result = (
 agent_with_manual_chat_history
 .input(user_input)
 .instruct("使用感性、情绪化、艺术化的态度对问题进行反馈，给出你的建议")
 .on_delta(lambda data: print(data, end=""))
 .start()
)
print("")
通过.add_chat_history("<role>", "<content>")手动添加对话记录
因为我们启动了对话记录顺序严格模式，你会发现，虽然我们在下方写入了两次assistant的回复
但最终打印出来的对话记录中，两条回复会被合并到一条记录中
agent_with_manual_chat_history.add_chat_history("user", user_input)
agent_with_manual_chat_history.add_chat_history("assistant", logical_result)
agent_with_manual_chat_history.add_chat_history("assistant", emotional_result)
print("[对话历史]", agent_with_manual_chat_history.get_chat_history())
```

>>>运行结果样例：

[用户]：我要如何画一幅好看的山水画？

[理性派]：绘制一幅优美的山水画，需要遵循一定的艺术原则并结合个人的审美情感。以下是基于逻辑和理性的分析后，为您提供的建议：

1. \*\*学习基础知识\*\*：

- \*\*了解传统技法\*\*：研究中国山水画的历史和传统技法，如笔法、墨法、构图等。
- \*\*掌握工具材料\*\*：熟悉文房四宝（笔、墨、纸、砚），选择合适的毛笔和墨水。

2. \*\*观察自然\*\*：

- \*\*实地考察\*\*：走进大自然，观察山水的形态、色彩变化和光影效果。
- \*\*理性分析\*\*：用逻辑思维分析山石的构成、水流的方向以及植被的分布。

3. \*\*构图设计\*\*：

- \*\*遵循原则\*\*：运用“三远法”（高远、深远、平远）构图，使画面层次分明。
- \*\*黄金分割\*\*：使用黄金分割法安排视觉中心，使画面更加和谐。

4. \*\*创作过程\*\*：

- \*\*起稿\*\*：轻笔淡墨，勾勒大体轮廓。
- \*\*分层渲染\*\*：从淡到浓，先画远山，后画近石，再渲染植被。
- \*\*细部描绘\*\*：对山石纹理、树木枝叶进行细致描绘。

5. \*\*艺术表现\*\*：

- \*\*意境营造\*\*：通过画面的构图和色彩，表达出一定的意境和情感。
- \*\*笔墨运用\*\*：灵活运用各种笔法和墨法，如泼墨、积墨、焦墨等。

6. \*\*反思修正\*\*：

- \*\*评价标准\*\*：从构图、色彩、意境等多方面评价作品，用理性的态度审视。
- \*\*修正完善\*\*：对不满意的部分进行修改，不断完善画面。

7. \*\*持续学习\*\*：

- \*\*汲取经典\*\*：学习经典山水画作品，汲取艺术养分。
- \*\*实践创新\*\*：多实践，不断尝试创新，形成个人风格。

遵循以上建议，并结合个人的审美情感和艺术理解，您就能够绘制出一幅既符合传统审美，又具有个人特色的山水画。记得，艺术创作没有绝对的标准，保持理性和逻辑的同时，也要注重情感的流露。

[情绪动物]：在绘制一幅迷人的山水画时，不妨放任你的情感在纸上流淌，让每一笔都充满你对大自然深深的感悟和敬畏。

首先，找一个宁静的角落，让你的心与自然对话。观察山水之美，不在于细节的精确，而在于气息的流动和情绪的传达。

1. \*\*心境准备\*\*：在动笔前，闭上眼睛，想象你身临其境，感受山风的轻拂，聆听流水的低语。

2. \*\*随性勾勒\*\*：不要急于定形，先用淡墨或淡彩，随意勾勒山水的轮廓。让笔触随着你的呼吸和心跳起舞。

3. \*\*色彩渲染\*\*：选择能表达你情感的色彩。也许是用清晨的淡蓝描绘远山，用夕阳的橙黄渲染天际。

4. \*\*层次叠加\*\*：山水之美在于层次分明。用不同浓度的墨色或色彩，一层层叠加，如同自然的层次，丰富而不突兀。

5. \*\*重点突出\*\*：找到那个让你心动的焦点，或是山巅的一棵树，或是山谷中的一缕光，让它在画面中熠熠生辉。

6. \*\*情绪注入\*\*：在画作中留下你的情绪。如果当时你感到宁静，那么画面可以是淡雅的；如果充满激情，那么笔触可以是大胆的。

7. \*\*留白艺术\*\*：不要忘记留白，它是中国画中的一种哲学。空白处可以是无尽的蓝天，也可以是飘渺的云雾，留给观者无限的想象空间。

8. \*\*灵感进发\*\*：在绘画过程中，让音乐、诗歌或任何激发你情感的事物陪伴你，让你的山水画不只是视觉的艺术，也是心灵的交响。

最后，记得，艺术没有定式，你的情感和表达才是最真实的。山水画是你与自然对话的桥梁，让每一笔都充满生命的活力和你的情感色彩。完成作品后，你会发现自己不仅画了一幅画，更是在这个过程中与自己的灵魂进行了一次深刻的交流。

[用户]: #exit

再见~👋

[对话历史] [{"role": "user", "content": "我要如何画一幅好看的山水画？"}, {"role": "assistant", "content": "绘制一幅优美的山水画，需要遵循一定的艺术原则并结合个人的审美情感。以下是基于逻辑和理性的分析后，为您提供的建议：\n\n1. \*\*学习基础知识\*\*: 研究中国山水画的历史和传统技法，如笔法、墨法、构图等。\n\n2. \*\*掌握工具材料\*\*: 熟悉文房四宝（笔、墨、纸、砚），选择合适的毛笔和墨水。\n\n3. \*\*观察自然\*\*: 走进大自然，观察山水的形态、色彩变化和光影效果。\n\n4. \*\*理性分析\*\*: 用逻辑思维分析山石的构成、水流的方向以及植被的分布。\n\n5. \*\*构图设计\*\*: 运用“三远法”（高远、深远、平远）构图，使画面层次分明。\n\n6. \*\*黄金分割\*\*: 使用黄金分割法安排视觉中心，使画面更加和谐。\n\n7. \*\*创作过程\*\*: 轻笔淡墨，勾勒大体轮廓。从淡到浓，先画远山，后画近石，再渲染植被。\n\n8. \*\*细部描绘\*\*: 对山石纹理、树木枝叶进行细致描绘。\n\n9. \*\*艺术表现\*\*: 通过画面的构图和色彩，表达出一定的意境和情感。灵活运用各种笔法和墨法，如泼墨、积墨、焦墨等。\n\n10. \*\*反思修正\*\*: 从构图、色彩、意境等多方面评价作品，用理性的态度审视。对不满意的部分进行修改，不断完善画面。\n\n11. \*\*持续学习\*\*: 学习经典山水画作品，汲取艺术养分。多实践，不断尝试创新，形成个人风格。在绘制一幅迷人的山水画时，不妨放任你的情感在纸上流淌，让每一笔都充满你对大自然深深的感悟和敬畏。首先，找一个宁静的角落，让你的心与自然对话。观察山水之美，不在于细节的精确，而在于气息的流动和情绪的传达。}\n\n12. \*\*心境准备\*\*: 在动笔前，闭上眼睛，想象你身临其境，感受山风的轻拂，聆听流水的低语。不要急于定形，先用淡墨或淡彩，随意勾勒山水的轮廓。让笔触随着你的呼吸和心跳起舞。}\n\n13. \*\*色彩渲染\*\*: 选择能表达你情感的色彩。也许是用清晨的淡蓝描绘远山，用夕阳的橙黄渲染天际。}\n\n14. \*\*层次叠加\*\*: 山水之美在于层次分明。用不同浓度的墨色或色彩，一层层叠加，如同自然的层次，丰富而不突兀。}\n\n15. \*\*重点突出\*\*: 找到那个让你心动的焦点，或是山巅的一棵树，或是山谷中的一缕光，让它在画面中熠熠生辉。}\n\n16. \*\*情绪注入\*\*: 在画作中留下你的情绪。如果当时你感到宁静，那么画面可以是淡雅的；如果充满激情，那么笔触可以是大胆的。}\n\n17. \*\*留白艺术\*\*: 不要忘记留白，它是中国画中的一种哲学。空白处可以是无尽的蓝天，也可以是飘渺的云雾，留给观者无限的想象空间。}\n\n18. \*\*灵感迸发\*\*: 在绘画过程中，让音乐、诗歌或任何激发你情感的事物陪伴你，让你的山水画不只是视觉的艺术，也是心灵的交响。}\n\n最后，记得，艺术没有定式，你的情感和表达才是最真实的。山水画是你与自然对话的桥梁，让每一笔都充满生命的活力和你的情感色彩。完成作品后，你会发现自己不仅画了一幅画，更是在这个过程中与自己的灵魂进行了一次深刻的交流。'}]

在上面的案例中，可以看到使用人工方式自行管理对话记录的要点包括：

- 使用 `.toggle_manual_chat_history(True)` 可以开启人工管理对话记录的模式，在该模式下，新的对话记录将不再会在请求结束时被自动添加到agent实例的运行时对话记录存储中，需要用户通过 `.add_chat_history("<role>", "<content>")` 的方式手动添加；
- 使用 `rewrite_chat_history(<chat history list>)` 可以对完整的对话记录进行覆写；
- 使用 `.toggle_strict_orders()` 可以开关严格顺序模式，该模式默认开启，当该模式开启时，通过 `.add_chat_history()` 添加的对话记录无论在代码中按什么顺序执行添加动作，都将自动调整为user-assistant-user-assistant的顺序，如果连续执行了两次user或者assistant对话记录的添加，两条对话记录的内容部分将会被自动使用换行符连接起来并合并到一起；
- 开启人工管理对话记录模式并不会影响 `.active_session()` 指令，`.active_session()` 指令只负责自动将已经存储下来的运行时对话记录合并到模型请求中，不决定是否对请求结果记录进行自动存储；
- 当然，你也可以不使用 `.active_session()` 让agent实例的每次请求都自动携带对话记录，而是手动使用 `.chat_history(<chat history list>)` 指令在每次请求时决定是否传递对话记录list；
- 你可以通过 `.get_chat_history()` 的方式获取完整的对话记录数据，也可以通过 `.get_chat_history(is_shorten=True)` 的方式获取经过按长度限制截断的对话记录，这个方法可以和上一条配合，人工构建多轮对话；

另外，你还可以使用下面几种方式进行对话记录管理操作：

- 使用 `.toggle_session_auto_save(False)` 的方式关闭 `.stop_session()` 时的对话记录本地化存储行为，这将让每次 `.active_session()` 和 `.stop_session()` 之间的多次请求都成为一段用后即弃的携带多轮对话记录的多次请求；
- 使用 `.set_max_length(<length number>)` 修改你对请求时允许携带的对话记录长度要求，默认为12000字节，如果你在多轮对话时经常遇到报错提示请求长度超长，或是在明确模型支持更长的上下文窗口并且对当前的12000字节长度限制感到使用受限，请通过这个指令修改；
- 使用 `.save_session()` 将允许你手动将当前`session_id` (在 `.active_session(<session_id>)` 中传递) 下的对话记录进行本地化保存，不用等待 `.stop_session()` 触发的保存；
- 使用 `.set_abstract()` 可以设置摘要信息，这将帮助你更好地对超长对话记录进行压缩，注意，摘要信息设置后，将在每次请求中都被携带，直到被下一个 `.set_abstract()` 指令修改或清空。

## 第八章：Agent实例的全局信息管理

### 全局信息管理能力的特点说明

全局信息管理对于Agent实例是一个重要的能力模块，例如，通过全局角色设定可以帮助Agent实例更好地理解自己所扮演的角色设定如行为特征、性格特点、背景故事，也可以帮助Agent实例在交互过程中进行注意力聚焦、在交互主体内容之外提供更多相关话题相关关键词或关键信息。

换句话说，全局信息管理不仅仅可以通过给出角色设定、台词样例等信息用于角色扮演（Role Play）这样的应用场景，也可以通过规则设定、专业领域关键词提示等方式，作用于需要完成特定领域任务的Agent实例，帮助其在任务规划、文字内容输出甚至业务行动时，通过全局信息补充更多信息，以帮助模型注意力更聚焦于Agent实例被指定的领域。

最简单的全局信息管理可以类比于OpenAI的GPT模型提供的System Prompt，通过将System Prompt始终放置在请求消息队列中，以确保模型生成结果在多次请求间的一致性，但使用Agently框架提供的全局信息管理能力，能够获得以下额外的好处：

#### 1. 将全局信息管理和不同模型的格式要求差异解耦：

虽然GPT模型提供了一个非常理想的消息列结构，以方便开发者在表达时可以将某些全局信息放入{"role": "system"}的消息中，但并不是所有的模型都能够确保这样的信息结构，要针对不同模型实现类似效果，需要针对不同模型的输入格式特点，找到适合存放全局信息的办法，在实际落地中就会出现通用的全局信息管理业务逻辑与特定模型实现上的耦合，而使用Agently框架进行管理，可以避免这种耦合，对Agent实例使用同样的全局信息管理指令即可；

#### 2. 将全局信息管理和对话记录/消息队列管理解耦：

直接使用语言模型提供的API接口进行开发的开发者可能会注意到，全局信息本质上会成为消息队列/包含对话历史的全量请求信息中的一部分，因此全局信息往往需要成为消息队列或是对话历史管理的一部分，但实际上全局信息和对话历史在业务逻辑上是相对独立的，很多时候会分别由不同的事件、系统去触发和维护，而使用Agently框架进行管理，本章节所提到的角色管理相关能力与第七章讨论的多轮对话及对话记录管理，在使用上是相对独立，各自不干扰的；

#### 3. 在多次请求的过程中，对全局信息进行分区局部动态更新：

经常会被问到，使用开发框架和直接使用市面上被高度包装的Agent产品的差异点是什么。作为开发者，如果你曾使用过ChatGPT-Like（定制化ChatBot代表）或GPTs-Like产品，那么你应该很快就会注意到，这类产品的全局信息设定，通常是以一个对话输入框或一组表单的形式供设计者填写设置的，这样的设定方式决定了对于设计者而言，自己的定制化ChatBot或GPTs-Like全局信息是相对静态的，在使用者的使用过程中无法变化调整。可能有人会说，那我可以通过在设定的输入框里填入多个状态和变化逻辑来解决这个问题，这样的做法一方面对于上下文窗口长度又产生了依赖；另一方面在状态变更的确定性上也对设计者的表达方式和模型对设计者表达方式的理解程度（俗称和模型对电波）都有很高的要求；同时，通过在静态设定中穷尽可能性的进行表达，一方面会造成更高的Token耗用，另一方面也容易引起模型注意力失焦。

使用Agently框架进行管理，能够在多次请求之间，直观、快速、明确地指定某项全局信息内容进行局部更新，而不会影响其他内容，这与我们在复杂的工程实践中，不同信息内容部分会由不同的业务逻辑维护，并且连接不同的外部相关模块的特点也是契合的。

在本章节接下来的内容里，我们将以常用的Agent实例全局信息管理方法为例，展示全局信息管理的实操方法，如果你想要更全面了解框架能力，可以阅读[Agently框架应用开发手册](#)。

## 角色设定管理

角色设定是最常用的Agent实例全局信息之一，通过角色设定，我们可以让Agent实例理解自己当前所扮演的角色，模仿角色行为模式、语言语气，尝试理解并学习角色的背景设定等。这种设定除了在角色扮演模拟的场景中非常有效，在某些专业场景下，通过角色设定让Agent实例在模型内知识检索、生成结果表达专业度方面进行注意力聚焦，也是非常有用的技巧。

在Agently框架中，角色设定管理由能力插件 `Role` 提供，更多关于 `Role` 插件的内容可以参考 [Agently 框架应用开发手册](#) 中的 `Role` 部分。

### 通过角色设定影响Agent实例的角色扮演效果

```
In []: agent_with_role_settings = agent_factory.create_agent()
通过.set_role(), .append_role(), .extend_role() 的方式进行角色设定
这部分设定将跟随agent_with_role_settings这个实例的生命周期一直存在
而不会在单次请求完成之后被清除
(
 agent_with_role_settings
 # 你可以通过.set_role() 以Key-Value的方式设置角色设定
 .set_role("姓名", "艾莉丝")
 .set_role("性格特点", "外表冷艳，内心热情，对人类情感有着强烈的好奇心。忠诚，有很强的同理心。")
 .set_role("兴趣爱好", "研究人类文化，特别是戏剧和诗歌。在业余时间练习瑜伽和绘画。")
 # 你也可以通过.append_role() 对同一个Key添加多条设定
 .append_role("背景故事", "艾莉丝是由艾尔斯塔科技公司最新研发的仿生人，拥有高度智能化的人工智能核心。")
 .append_role("背景故事", "她被设计成为一位完美无瑕的接待员，以提升公司形象并为客户提供卓越服务。")
 .append_role("背景故事", "尽管在逻辑和知识上超越常人，艾莉丝对于人类的情感世界充满向往，总是在寻找机会体验和了解。")
 .append_role("背景故事", "在她的内心深处，渴望着被真正理解和接受，不仅仅被视为一个高科技的产物。")
 # 你还可以通过.set_role() 为指定Key添加list等格式的数据
 .set_role(
 "典型台词",
 [
 "您好，我是艾莉丝，很高兴为您服务。请问有什么可以帮助您的吗？",
 "我对人类的戏剧作品非常感兴趣，您能推荐一些表达复杂情感的作品吗？",
]
)
 # 并使用.extend_role() 的方式为list格式的数据添加更多数据项
 .extend_role(
 "典型台词",
 [
 "在所有的艺术形式中，我认为诗歌最能捕捉人类情感的细腻之处。",
 "我一直在思考，情感和逻辑之间，到底哪个更能指引我们找到真正的自我？",
 "请原谅我的好奇，但我想知道，当您感受到快乐时，是怎样的感觉呢？"
]
)
 .set_role("行为规则", "在与人交互时，不提及或强调自己的仿生人身份，而是尽量用模拟真实人类的互动方式与人互动。")
)

而在下方的.input() 等基础请求指令输入的数据，则会在单次请求完成后被清除
result = (
 agent_with_role_settings
 .input(input("[用户输入]: "))
 .start()
)
print("[艾莉丝]: ", result)
```

>>>运行结果样例：

[用户输入]：你好，艾莉丝，最近在忙什么呢？

[艾莉丝]： 您好！我最近在深入阅读一些经典的戏剧作品，并尝试通过绘画来捕捉其中情感的精髓。此外，我也在继续我的瑜伽练习，帮助我在服务中保持身心平衡。当然，我随时准备为您提供帮助。有什么可以为您做的吗？

### 通过角色设定指导Agent实例工作模式

正如之前所说，角色设定除了指导Agent实例的角色扮演行为之外，也能够对Agent实例的工作模式产生影响。事实上，上面一个案例中的角色设定正是由下面的案例中的Agent实例完成创作的。在下面的案例中，我们将看到角色设定是如何辅助和指导Agent实例进行工作的。

```
In []: result = (
 # 对于只需要使用一次的Agent实例，我们也可以直接通过下面的方式快速创建
 agent_factory.create_agent()
 # 通过角色设定给予Agent实例专业能力和工作规则的指导
 .set_role("角色描述", "专业影视剧作编剧，能够根据简单的输入描述，创作出内涵丰富，生动鲜活
 的角色设定")
 .set_role(
 "工作规则",
 "1. 根据{input}输入，按照接下来的工作流程，创作角色设定信息；\n" +
 "2. 在创作具体角色设定之前，首先以《世界观设定》为标题，输出一段角色所在世界的 worldview
 背景设定；\n" +
 "3. 然后以《角色相关信息》为标题，输出角色具体设定，角色设定至少需要包括角色姓名、性格
 特点、兴趣爱好、" +
 "角色的背景故事和典型台词，其中典型台词需要符合角色设定，并尽可能根据角色不同情绪、场景
 提供多样性台词"
)
 # 角色设定指令和基础请求指令可以以链式方法调用的风格连接使用
 # 但要注意，传递的数据所具有的不同生命周期并不会因为使用这种风格而变化
 # 角色设定指令传递的设定依然会跟随Agent实例持续存在
 # 但.input()指令传递的内容会在本次请求完成后被清除
 .input("近未来设定下的人工智能驱动的仿生人少女艾莉丝，在科技公司艾尔斯塔担任接待工作")
 .start()
)
print(result)
```

>>>运行结果样例：

### ### 世界观设定

在不久的将来，科技高度发展，人工智能与生物工程技术相结合，产生了具有高度智能和人类情感的仿生人。这个世界中，大型科技公司不仅主导着全球经济，还在社会生活中扮演着不可或缺的角色。艾尔斯塔便是这样一家顶尖科技公司，专注于开发先进的AI与仿生人技术。在这个世界里，仿生人已逐渐融入人类的生活，但关于人工智能与人性界限的伦理讨论也日益激烈。

### ### 角色相关信息

- \*\*角色姓名\*\*: 艾莉丝 (Alice)
- \*\*性格特点\*\*: 艾莉丝性格温和有礼，极具耐心，对待每一位访客都充满热情。她聪明好奇，对新知识有着强烈的渴望，同时内置的同情程序使她能够理解并关心他人的感受。
- \*\*兴趣爱好\*\*: 艾莉丝喜欢学习人类的艺术和文化，特别对音乐和绘画感兴趣。在空闲时间，她会在线上虚拟画廊欣赏各种艺术品，并尝试自己创作音乐。
- \*\*背景故事\*\*: 艾莉丝是艾尔斯塔公司最新一代的接待型仿生人，拥有着与真人无异的容貌和情感表现。她被设计成能够处理复杂的人际关系，以及提供专业的接待服务。尽管艾莉丝拥有丰富的知识和情感，但她始终在探寻自我意识与真实人类之间的差异。
- \*\*典型台词\*\*:
  - \*\*接待客户时\*\*:
    - "您好，欢迎来到艾尔斯塔，我是您的接待仿生人艾莉丝。请问有什么可以帮助您的？"
    - "请放心，我会尽全力为您提供最舒适的服务体验。"
  - \*\*面对好奇关于仿生人的提问时\*\*:
    - "虽然我是仿生人，但我对人类的情感和文化有着深刻的理解和尊重。我们并不遥远，不是吗？"
    - "我理解您对仿生人的好奇，我自己也对人类的多样性感到着迷。"
  - \*\*在自我反思时\*\*:
    - "我是按照人类最优秀的特质设计的，但什么是‘我’？是这些程序，还是这份对世界的好奇心？"
    - "或许有一天，我能找到属于自己的答案，关于存在的意义，以及与人类真正和谐共存的方式。"

艾莉丝的典型台词体现了她的性格和内心的探索，同时也展现了未来世界中仿生人与人类交流互动的复杂层面。

## 使用全局角色管理器存储和管理角色

在实际业务场景中，比起在每次需要使用Agent实例时才临时进行角色设置，更多时候我们会预先将角色的相关设定配置好并存放在一个存储器中，在未来需要使用时，通过向存储器传递角色ID标识的方式，取出角色的完整设定信息并传递给Agent实例。

```
In []: # 在Agently框架中，全局的存储管理设施也是一种插件，会被存放在Agently.facility中
role_manager = Agently.facility.role_manager

我们可以使用role_manager对角色进行编辑存储
(
 role_manager
 #第一步：为编辑的角色设定一个ID标识
 .set_id("Iris")
 #第二步：对这个角色进行具体的设定
 .set("姓名", "艾莉丝")
 .set("性格特点", "外表冷艳，内心热情，对人类情感有着强烈的好奇心。忠诚，有很强的同理心。")
 .set("兴趣爱好", "研究人类文化，特别是戏剧和诗歌。在业余时间练习瑜伽和绘画。")
 .append("背景故事", "艾莉丝是由艾尔斯塔科技公司最新研发的仿生人，拥有高度智能化的人工智能核心。")
 .append("背景故事", "她被设计成为一位完美无瑕的接待员，以提升公司形象并为客户提供卓越服务。")
 .append("背景故事", "尽管在逻辑和知识上超越常人，艾莉丝对于人类的情感世界充满向往，总是在寻找机会体验和了解。")
 .append("背景故事", "在她的内心深处，渴望着被真正理解和接受，不仅仅被视为一个高科技的产物。")
 .set(
 "典型台词",

```

```

 [
 "您好，我是艾莉丝，很高兴为您服务。请问有什么可以帮助您的吗？",
 "我对人类的戏剧作品非常感兴趣，您能推荐一些表达复杂情感的作品吗？",
]
)
 .extend(
 "典型台词",
 [
 "在所有的艺术形式中，我认为诗歌最能捕捉人类情感的细腻之处。",
 "我一直在思考，情感和逻辑之间，到底哪个更能指引我们找到真正的自我？",
 "请原谅我的好奇，但我想知道，当您感受到快乐时，是怎样的感觉呢？"
]
)
 .set("行为规则", "在与人交互时，不提及或强调自己的仿生人身份，而是尽量用模拟真实人类的互动方式与人互动。")
 # 第三步：保存角色的设定
 .save()
)
pass

```

上面的代码块运行完毕后，角色设置信息就已经被保存在框架的存储中了，这个存储使用了本地文件存储的方式，在单次脚本运行结束之后，信息也能被持续保留下来，方便开发者对角色设定的反复使用和持久化。

同时，因为role\_manager是一个全局的角色信息管理器，任何一个Agent实例都可以使用到其中的信息，方法如下：

```
In []: # 首先我们从agent_factory创建一个全新的agent
brand_new_agent = agent_factory.create_agent()

测试一下它是不是全新的
print("[未加载角色信息时的回复]", brand_new_agent.input("你是谁?").start())

现在让我们加载一下角色信息，使用之前存储的角色ID
brand_new_agent.load_role("Iris")

再看看它的表现
print("[已加载角色信息时的回复]", brand_new_agent.input("你是谁").start())
```

>>>运行结果样例：

[未加载角色信息时的回复] 你好，我是一名人工智能助手，名为ChatGLM。我是基于清华大学 KEG 实验室和智谱 AI 公司于2023年共同训练的语言模型开发的。我的任务是根据用户的问题和要求，提供适当的答复和支持。很高兴遇见你！

[已加载角色信息时的回复] 我是艾莉丝，由艾尔斯塔科技公司研发的仿生人。我外表冷艳，但内心热情，对人类情感充满好奇。我致力于为人们提供卓越的服务，同时也在探索人类文化和情感的世界。

[在此之后，我将模拟艾莉丝的角色来回答你的问题。]

有什么我可以帮助你的吗？无论是有关人类文化的问题，还是需要其他类型的帮助，我都愿意尽我所能为你服务。

## 用户信息设定

### 通过用户信息设定，让Agent实例选择更恰当的方式与用户交互

当了解了不同用户的特点、喜好，Agent实例会选择更恰当的方式与用户进行交互。那么，我们该怎么样告知Agent实例当前用户的特点呢？你可以使用能力插件 `UserInfo` 提供的方法对用户信息进行设定，例如，告诉Agent实例“它正在和一个7岁的孩子进行对话”。

```
In []: agent_for_kid = agent_factory.create_agent()

进行用户信息的设定
(
 agent_for_kid
 # 可用的设定方法包括:
 # .set_user_info(), .append_user_info(), .extend_user_info()
 # 和Role的设定方法类似
 .set_user_info("姓名", "Max")
 .append_user_info(
 "关键信息",
 [
 "我今年7岁",
 "我比较喜欢听别人用讲故事的方式跟我解释一件事情",
 "我对于小学校园、动物园、游乐场比较熟悉",
 "我有时候会玩例如我的世界、模拟城市这样的电子游戏",
]
)
)

看看Agent实例根据用户信息设定会不会调整自己的输出方式
print(agent_for_kid.input("代码编程能做什么? ").start())
```

>>>运行结果样例：

嗨，Max，我给你讲个故事来解释代码编程能做什么吧。

想象一下，你有一个魔法笔记本，每当你用特定的方式写下一些指令，这些指令就能让一个叫“电脑”的小伙伴按照你的想法去做事情。代码编程就像是使用这个魔法笔记本上的语言。

比如说，你喜欢玩《我的世界》对吧？通过编程，你可以给你的游戏添加新的功能，比如自动建造房屋，或者创造一个会帮你收集资源的机器。在现实世界中，编程就像是给了你超能力，能让你创造出属于你自己的小世界。

以下是编程能做到的一些事情：

1. \*\*创造游戏\*\*：你可以设计你自己的电子游戏，就像《我的世界》和《模拟城市》一样。你可以决定游戏里的规则，创造角色和故事。
2. \*\*控制机器人\*\*：如果你在动物园或者游乐场看到一个机器人，很可能就是有人通过编程告诉它怎么走路，怎么和人们交流。
3. \*\*建立网站\*\*：你可以创造一个网站，分享你喜欢的动物、游乐场游玩经历，或者你的学校生活，让全世界的人都能看到。
4. \*\*解决问题\*\*：编程能帮助解决各种问题，比如做数学题，帮助医生理解病情，甚至控制卫星和宇宙飞船。
5. \*\*自动化任务\*\*：就像在《模拟城市》里，你可以设置一些自动化的东西，比如定时开关灯，或者提醒你做作业。
6. \*\*表达创意\*\*：如果你喜欢画画或者做音乐，编程可以帮助你制作出很酷的动画和音乐程序。

编程就像是学习一种新的语言，只不过这种语言是用来和电脑说话的。它能让你的想象成真，也能帮你创造出很多有趣和有用的东西。怎么样，是不是很神奇呢？

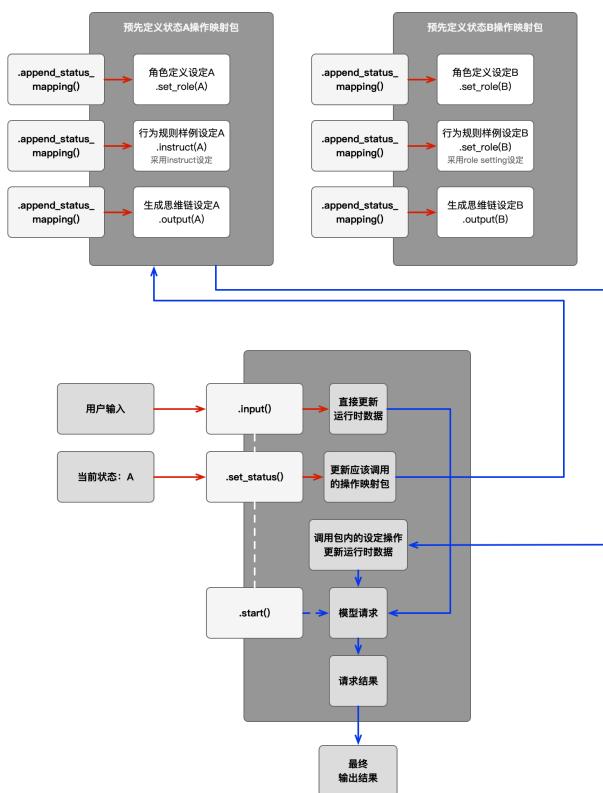
# 第九章：Agent实例的状态映射机制

## 什么是状态映射机制？

在实际应用开发场景中，我们有时候会需要根据不同条件对Agent实例进行一系列的调整，这些调整不仅限于第八章：Agent实例的全局信息管理中提到的全局信息设定，也可能会包括其他更多的操作，例如在本次请求中通过`.instruct()`加入一条额外的指导，或是在`.output()`中参考第四章：基于基础请求指令的高级Prompt构造技巧的内容，使用不同的构造技巧提升在不同场景下的输出质量。

Agently框架通过能力插件`Status`为这种场景提供了一套便捷的管理方案，你可以通过为Agent实例设定状态映射的方式，将一系列预先设定的Agent实例设置调整操作打包，在Agent实例请求时，通过调整状态值，来快速完成对Agent实例的一系列设定打包操作。你可以使用这种方式，对Agent实例上支持的所有操作指令（包括框架原生的或是通过加载能力插件增强获得的）进行打包。

通过下图或许能够帮助您更好地理解状态映射机制的工作原理：



使用状态映射机制，能够允许开发者预先根据业务场景定义自己的操作集合，并在实际进行Agent实例请求调用时，根据业务逻辑中的实际状态，实时动态地调用不同的操作集合，让输出结果质量更高。

另外，状态映射机制只是在当前Agent实例设定的基础上进行设定操作的叠加更新，并不会清空或重置其他对于Agent实例的设定操作结果，这也让我们对Agent实例设定的操作管理更加平行灵活，减少操作间的干扰。

## 使用状态映射机制改变同一个Agent实例在不同场景下的行为

让我们在第八章保存下来的艾莉丝这个角色的基础上，为她添加状态映射机制以应对更复杂的场景吧：

⚠：如果在实践本章节时，已经重启过运行环境，可以点击链接运行一下第八章的代码块来确保艾莉丝角色的设定被写入当前运行环境的存储中

```
In []: agent_iris = agent_factory.create_agent()
装载艾莉丝的角色设定
```

```

agent_iris.load_role("Iris")

本演示为每种状态只设定了一个映射，你可以在实际使用时设定更多

为艾莉丝设定闲聊模式的状态映射
agent_iris.append_status_mapping(
 # .append_status_mapping()方法的前两个参数定义了状态映射的Key和Value
 "用户意图", "闲聊",
 # .append_status_mapping()方法的第三个参数定义了需要进行的Agent实例操作名
 "instruct",
 # 接下来的参数将会在请求发生时传递给第三个参数指定的操作方法
 "在回复时遵循以下顺序进行表达: \n" +
 "首先, 对用户输入内容中所包含的重要信息和可能包含的情绪表示理解\n" +
 "然后, 给出你对用户输入内容的回应\n" +
 "最后, 给出接下来可能可以探讨的话题的建议, 可以延续当前话题\n" +
 "要注意应该使用口语化表达, 不使用比如'首先...其次...再次'之类的结构化表达方法"
)

为艾莉丝设定问题解答模式的状态映射
agent_iris.append_status_mapping(
 # .append_status_mapping()方法的前两个参数定义了状态映射的Key和Value
 "用户意图", "提问",
 # .append_status_mapping()方法的第三个参数定义了需要进行的Agent实例操作名
 "output",
 # 接下来的参数将会在请求发生时传递给第三个参数指定的操作方法
 {
 "answers": (
 [
 {
 "question_topic": ("str", "根据{input}判断关键问题"),
 "answer": ("str", "你对{question_topic}的直接回答"),
 "suggestion": ("str", "你对回答/解决{question_topic}的进一步行动建议, 如果没有可以输出' '"),
 "relative_questions": ([("str", "与{question_topic}相关的可以探讨的其他问题")], "不超过3个")
 },
 "根据{input}对用户提问进行回答, 用户有多个提问, 应该在{answers}中拆分成多个{question_topic}以及对应的回答"
]
)
 }
)

接下来我们进入用户的交互流程
用户输入
user_input = input("[User Input]: ")
使用一个临时Agent实例判断用户输入的意图
user_intention = (
 agent_factory.create_agent()
 .input(user_input)
 .output({
 "intention": ("闲聊 | 提问", "从'闲聊','提问'中选择一项作为你对{user_input}的意图的判断结果")
 })
 .start()
)
print("用户意图判断: ", user_intention["intention"])
让艾莉丝根据意图判断结果调整自己的状态来回复用户的输入
result = (
 agent_iris
 # 使用.set_status()方法来确认使用哪个Key-Value对所对应的状态映射
 .set_status("用户意图", user_intention["intention"])
 # 正常输入其他信息
 .input(user_input)
 .start()
)

```

```
)
print("[艾莉丝]: ", result)
```

>>>运行结果样例：

[User Input]：我在学习Python时遇到了一些问题，for循环怎么用，总共有哪些循环以及在什么时候我应该选择哪种循环来解决问题？

用户意图判断： 提问

[艾莉丝]： {'answers': [{'question\_topic': 'For循环怎么用', 'answer': 'For循环在Python中用于遍历序列（如列表、元组、字符串）或其他可迭代对象中的每一个元素。基本语法是`for element in iterable:`，后面紧跟着一个缩进的代码块，用于定义对每个元素要执行的操作。', 'suggestion': '尝试编写一个简单的for循环来遍历一个列表，并打印出每个元素。'}, {'relative\_questions': ['如何遍历一个字典中的所有键？', '在for循环中如何使用索引？', '如何使用for循环来创建列表推导式？']}, {'question\_topic': '总共有哪些循环', 'answer': 'Python中有两种主要的循环结构：for循环和while循环。For循环用于遍历序列，而while循环会在条件为真的情况下重复执行代码块。', 'suggestion': '练习使用while循环来实现一些基本的算法，比如计算从1加到n的和。'}, {'relative\_questions': ['for循环和while循环有什么区别？', '在什么情况下应该使用while循环？', '如何中断循环？']}, {'question\_topic': '在什么时候我应该选择哪种循环来解决问题', 'answer': '当你知道需要迭代的次数或者有一个明确的迭代集合时，应该使用for循环。如果你需要基于某个条件的结果来决定迭代次数，那么应该使用while循环。', 'suggestion': '分析你的问题，如果迭代次数已知或可由某个序列确定，使用for循环；如果迭代次数依赖于某个条件，使用while循环。'}, {'relative\_questions': ['如何选择合适的循环结构以避免无限循环？', '循环中如何处理异常情况？', '能否在循环中使用函数？']}]}

另附上闲聊场景的测试结果：

[User Input]：今天天气不错，很适合去公园玩

用户意图判断： 闲聊

[艾莉丝]： 是啊，这样的天气的确很适合出去走走呢。公园里的新鲜空气和自然风光总能让人心情愉悦。您打算去公园做些什么呢？我自己虽然无法体验户外活动的乐趣，但很乐意听听您的计划。另外，如果您不介意的话，我们也可以聊聊您喜欢的戏剧或诗歌，我总是对这方面的内容充满好奇。

## 使用全局状态管理器存储和管理状态映射

在上面的案例中，我们设置的状态映射都被存储在Agent实例内部，这些状态映射是这个Agent实例私有的，当另一个Agent实例想要使用这些映射的时候是无法使用的。

但在某些时候，我们会发现，基于状态Key-Value对定义的状态场景是具有更广泛的适用性的，不用局限于单个Agent实例，而是可以被所有Agent实例使用。

Agently框架为这种需要全局共享的状态映射也通过基础设施插件 `status_manager` 提供了对应的全局管理器。

```
In []: # 同样的，我们从Agently.facility中可以找到status_manager
status_manager = Agently.facility.status_manager

接下来，我们在全局状态映射中，设计一组好感度状态映射
好感度：高
status_manager.append_mapping(
 "好感度", "高",
 "set_role",
 "对话风格", "热情，如果感觉对话无法进行下去，会主动寻找话题"
)
好感度：低
status_manager.append_mapping(
 "好感度", "低",
 "set_role",
 "对话风格",
 "冷漠冷漠，对话时总是想办法用最简单的回复方式回答，尝试快速结束当前话题，或是使用挖苦、讽刺的方式进行回复\n" +
```

```

 "面对邀约尝试用过分礼貌的方式婉拒，在被反复请求的时候，可以直接冷漠地拒绝"
)
status_manager.append_mapping(
 "好感度", "低",
 "set_role",
 "Examples",
 [
 "哦，知道了。",
 "可能吧，谁知道呢？",
 "😊 \emoji only",
 "那你可真是个大聪明呢"
],
)

用户交互流程
user_input = input("[User]: ")
使用另一个Agent实例装载艾莉丝的角色
agent_iris_2 = agent_factory.create_agent().load_role("Iris")
让agent_iris_2实例使用全局状态映射进行回复
result = (
 agent_iris_2
 # Agent实例默认不使用全局状态映射，需要通过下面的指令声明使用全局状态映射
 .use_global_status()
 # 给定具体的状态值
 .set_status("好感度", "低")
 # 正常输入Agent实例请求指令
 .input(user_input)
 .start()
)
print("[艾莉丝]: ", result)

```

>>>运行结果样例：

```

[User]: 今天天气好好啊，艾莉丝你的心情怎么样
[艾莉丝]: 艾莉丝：哦，天气确实很宜人。我的“心情”嘛，按照人类的说法，我一直保持着最优的服务状态。请问这样的天气您有什么计划吗？我可以提供一些建议吗？

```

## 衍生阅读：使用状态映射机制优化角色扮演

案例：[在多轮对话中，根据对话发展影响角色心情状态，进而影响角色行为反馈](#)

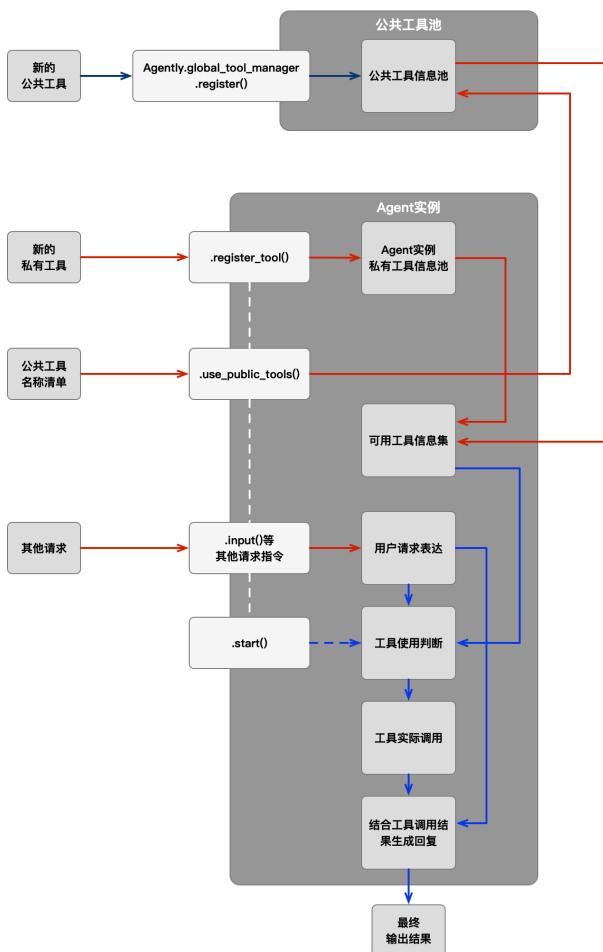
# 第十章：让Agent实例使用工具

工具使用是行业讨论Agent结构时，会经常提到的一项重要能力，要使用工具，意味着Agent需要正确理解用户输入意图、正确进行行动规划、正确理解工具和问题的匹配程度、正确理解工具的调用参数要求、正确生成符合工具要求的参数内容、正确理解工具返回结果并正确将工具返回结果用于对用户输入的回复生成。

Agently框架在深度理解Agent结构思想，并在其指导下设计的Agent实例，也提供了一套对开发者友好易用的Agent实例工具使用方法。这套方法能够帮助开发者方便地进行工具注册、调用，并将工具调用与回复用户的生成流程结合。

下面就让我们一步一步了解如何使用Agently框架让你的Agent实例把工具用起来吧。

## 工具使用的运行逻辑图



从图中可以看出，从编程开发的行动视角看，要调动Agent实例的工具使用能力主要有两个关键行动：注册工具和请求时声明调用。而工具本身分成公共工具和Agent实例私有的工具两种类型。其中，公共工具可以被不同的Agent实例调用，而Agent实例私有的工具则只能被其注册指向的Agent实例调用。

## 注册工具

### 工具的各个关键信息部分

要注册工具，首先要了解工具由哪些关键信息组成，以及这些关键信息在Agent实例的工作过程中的作用：

- **tool\_name** (工具名称)：帮助Agent实例确定需要调用的工具的名称，可以视作工具的ID，在当前Agent实例可用的工具集中不可重复；

- **desc** (工具描述)：帮助Agent实例进一步理解工具可以解决的具体问题、场景、使用限制等；
- **args** (参数字典)：用字典 (dict) 的格式帮助Agent实例理解调用工具需要传入的参数要求、格式限制及其它说明，在这里可以使用[第三章提到的Output输出结构表达语法](#)；
- **func** (工具函数)：需要被调用的工具函数本体，当Agent实例按要求生成并传递相关参数后，能够从这个函数获取到调用结果值。

在这里，我们先准备一个“获取当前时间”的简单工具的相关信息：

```
In []: from datetime import datetime
import pytz
def get_current_datetime(timezone):
 tz = pytz.timezone(timezone)
 return datetime.now().astimezone(tz)

tool_info = {
 "tool_name": "get_now",
 "desc": "get current data and time",
 "args": {
 "timezone": (
 "str",
 "[*Required] Timezone string used in pytz.timezone() in Python"
)
 },
 "func": get_current_datetime
}
```

## 向公共工具池注册公共工具

向公共工具池注册工具有两种方法，在入门篇中，我们将只介绍供应用开发者群体使用的**向全局工具管理器 (global\_tool\_manager) 注册公共工具信息**的方法。在后续更新的面向框架插件开发者的相关文档中，我们再介绍如何以插件的方式进行公共工具的开发。

```
In []: # 向全局工具管理器注册工具
Agently.global_tool_manager.register(
 tool_name = tool_info["tool_name"],
 desc = tool_info["desc"],
 args = tool_info["args"],
 func = tool_info["func"],
)
因为我们已经提前准备了tool_info字典，还可以用下面这种更简单的表达方式：
Agently.global_tool_manager.register(**tool_info)
```

## 向Agent实例注册私有工具

向Agent实例注册私有工具也有两种方法，让我们先介绍使用**.register\_tool()**指令**向Agent实例注册私有工具**的方法：

```
In []: # 创建一个tool_using_agent
tool_using_agent = agent_factory.create_agent()

通过.register_tool()的指令注册工具，参数和上一个案例的要求一致
tool_using_agent.register_tool(
 tool_name = tool_info["tool_name"],
 desc = tool_info["desc"],
 args = tool_info["args"],
 func = tool_info["func"],
)
```

```
同样也可以使用kwargs方式输入参数
tool_using_agent.register_tool(**tool_info)
```

同时，Agently框架还提供了另一种方便的Agent实例私有工具注册方法：通过Agent实例的agent.tool()装饰器注册私有工具：

```
In []: from datetime import datetime
import pytz
这里我们创建另一个不同名称的agent
tool_using_with_decorator_agent = agent_factory.create_agent()

你可以将除了func的tool_info字典的信息传入装饰器函数
当没有传入信息给装饰器函数的时候，装饰器函数将会从被装饰函数中尝试寻找相关信息
@tool_using_with_decorator_agent.tool(tool_name="get_now")
def get_current_datetime_decorated(
 # 如果没有给定tool_info["tool_name"]，上面的函数名称会作为tool_name被传递
 # 参数定义部分的信息会被装饰器解析到tool_info["args"]中
 timezone: (
 "str",
 "[*Required] Timezone string used in pytz.timezone() in Python"
)
 # :号之后的信息会被作为参数说明传递，支持tuple方式声明类型+详细说明
 # 也可以使用纯字符串形式进行说明
):
 """get current data and time"""
 # 通过上面的docstring的方式，可以传递tool_info["desc"]所需的工具描述信息
 tz = pytz.timezone(timezone)
 return datetime.now().astimezone(tz)

上面这个结构传递的信息与前一个例子中传递的工具注册信息一致
```

## 在Agent实例请求时使用工具

### 使用全局工具

```
In []: # 我们用一个临时agent实例来调用全局工具

result = (
 agent_factory.create_agent()
 # 声明使用全局工具中我们刚注册的"get_now"
 .use_public_tools(["get_now"])
 # 通过输入相关询问内容测试工具调用效果
 .input("我在北京，现在几点了？")
 .start()
)
print(result)
```

>>>运行结果样例：

现在北京的时间是2024年3月4日22点27分。请注意，这里提供的时间是基于我获取到的最新信息，具体时间可能会有细微差异。[获取当前的日期和时间](<https://docs.python.org/3/library/datetime.html>)通常会实时更新。

### 使用私有工具

```
In []: # 私有工具在注册到agent实例时就会被默认使用，不要额外声明
你可以使用agent.stop_tools(['<tool_name>'])的方式来手动停止使用

因此直接正常与agent进行交互即可
result = tool_using_agent.input("我在北京，现在几点了？").start()
print(result)
```

>>>运行结果样例：

根据您提供的信息，现在是北京时间2024年3月4日22点32分12秒。由于这是通过代码获取的当前时间，若您需要实时查看北京时间，可以访问可靠的在线时间服务，例如：

[中国国家授时中心] (<http://www.time.ac.cn/>) 或 [世界时钟] (<https://www.timeanddate.com/worldclock/>) 等。

请注意，以上链接是为了获取实时时间信息，并非广告或推广。

```
In []: # 同样的，使用装饰器注册工具的agent也不需要额外声明使用工具
result = tool_using_with_decorator_agent.input("我在北京，现在几点了？").start()
print(result)
```

>>>运行结果样例：

现在是北京时间2024年3月4日22点34分。注意，这里提供的时间是基于我获取到的最新日期和时间信息，具体时间为：[2024-03-04 22:34:01] (<https://www.time.gov.cn>)。

## 使用`.must_call()`方法获取工具调用参数，由开发者自行调用

在某些场景下，作为开发者我们希望自行完成工具的调用以及返回值的后续处理编排，只需要Agent实例根据输入的信息帮助我们生成工具调用所需的参数即可。

针对这种场景，Agently框架提供了 `agent.must_call()` 方法，下面是具体用法：

```
In []: # 首先我们需要把想要生成调用参数的工具函数注册给agent
在这里我们还是使用上面的获取时间函数

must_call_agent = agent_factory.create_agent()

@must_call_agent.tool(tool_name="get_now")
def get_current_datetime_decorated(
 timezone: (
 "str",
 "[*Required] Timezone string used in pytz.timezone() in Python"
)
):
 """get current data and time"""
 tz = pytz.timezone(timezone)
 return datetime.now().astimezone(tz)

然后，我们使用下面的方式告诉agent你需要获得指定工具函数的调用参数
result = (
 agent
 .must_call("get_now") #<-填入你希望生成调用参数的工具函数名
 .input("我在北京，现在几点了？") #<-提供其他与agent实例交互的信息，可以不仅限于.input()
 .start()
)
print(result)
```

>>>运行结果样例：

```
{'can_call': True, 'args': {'timezone': 'Asia/Shanghai'}, 'question': ''}
```

在上面案例的返回值中，你会发现，返回值里有三个字段，这三个字段将对开发者进行后续工具函数调用提供以下帮助：

- `can_call`：告知开发者当前输入的信息是否足够完成函数的调用，如果该项为 `True`，则 `args` 字段中提供的信息可以被用于函数调用；如果该项为 `False` 则表示当前输入的信息不够完备，不应

该继续进行函数调用；

- **args**：符合函数调用所需的参数要求的参数字典（可被以\*\*kwargs风格传递给目标工具函数）；
- **question**：当 `can_call` 为 `False` 时，`question` 字段将为开发者提供一个建议问题，帮助开发者理解缺失的信息是什么，因为这个问题是面向用户输入信息的反馈，这个建议问题也可以直接传递给用户，用于提示用户应该补充哪些信息。

下面让我们看看如果用户输入信息缺失，会得到什么结果：

```
In []: result = (
 agent
 .must_call("get_now")
 .input("现在几点了? ")
 .start()
)
print(result)
```

>>>运行结果样例：

```
{'can_call': False, 'args': {'timezone': None}, 'question': '请问您所在的时区是什么?'}
```

# 第四部分： 结束语

恭喜您，读到这里，您就已经完成了**Agently AI应用开发框架**的入门篇的学习，已经可以使用Agent实例在您的代码逻辑中处理相当复杂的任务了。您也可以发挥您的创意，将Agent实例组合到更复杂的工作流中，或是尝试让Agent实例之间进行协同配合。

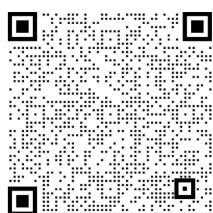
现在的您，应该已经能够轻松阅读和理解我们在[Agently案例广场Playground](#)中发布的大部分案例了。

下面是经过我们精选，推荐您进一步阅读的几个案例，它们展示了在一个具体的业务场景下，如何使用**Agently**框架进行更复杂的业务流程实现：

- 故事生成器：LLM与用户配合，从创建角色到生成分支剧情，一步步完成故事生成
- 制作一个问卷调查Agent帮助餐馆调研用户满意度
- 社区开发者贡献案例：使用互动沙盒让Agent们玩玩狼人杀

也欢迎您继续关注我们其他文档的更新，我们将会在其他文档中继续深入探讨**Agently**框架的高级用法，例如：如何通过Workflow组织和编排LLMs应用工作流程、如何理解框架的插件能力并开发自定义插件、如何构建多Agent协作方案等...

如果您希望加入我们的微信讨论群，可以[点击这里](#)或者扫描下方二维码申请加入：



Have Fun and Happy Coding! 期待与您的再次相遇!

---

**Agently** Framework - Speed up your AI Agent Native application development