

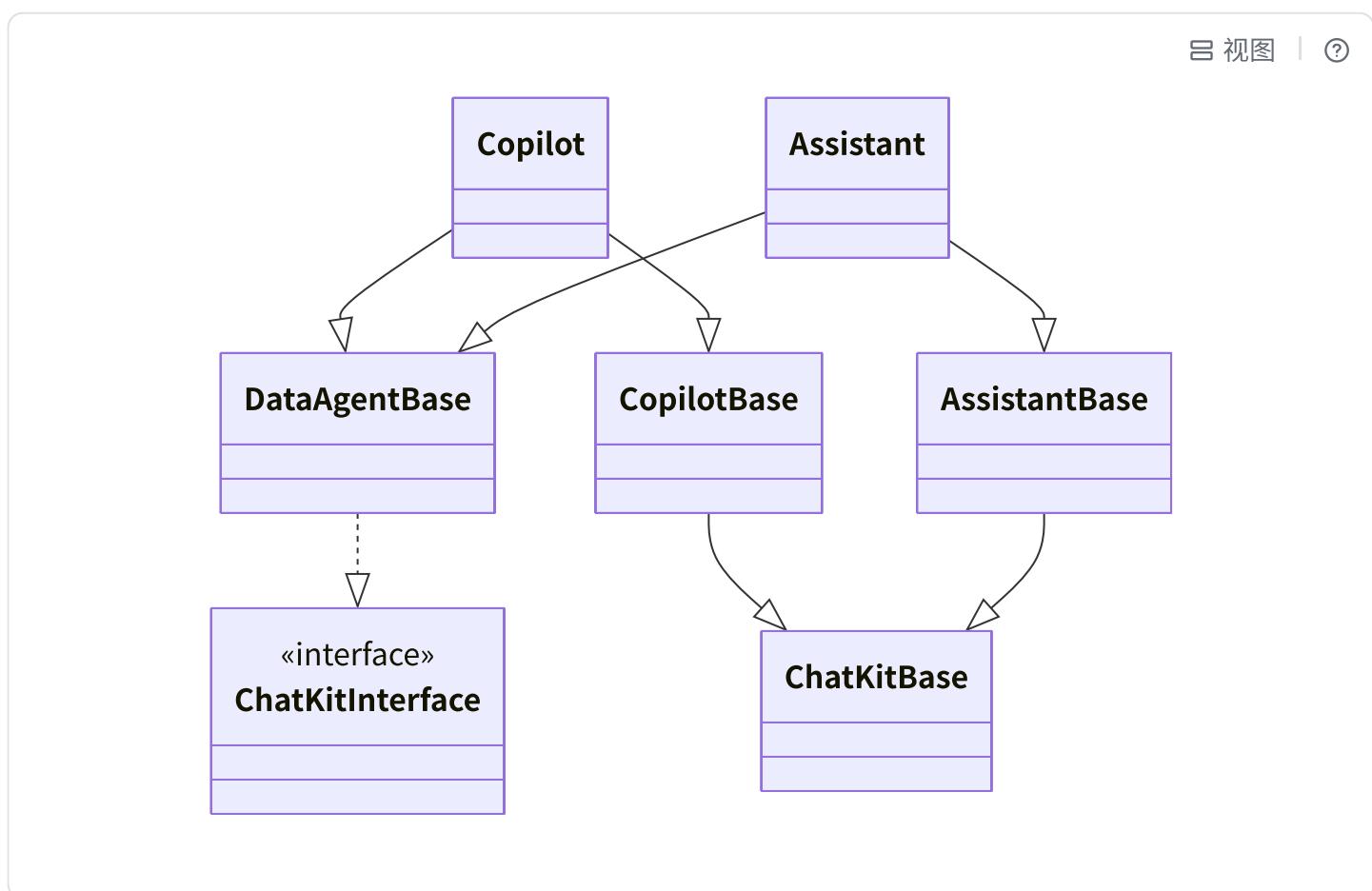
ChatKit for DIP

一、基本概念

ChatKit for DIP 是专门适配 AISHU Data Agent 智能体 API 的 AI 助手。ChatKit for DIP 导出两个组件：

- Copilot：右侧跟随的 AI 助手
- Assistant：主 AI 对话入口

二、组件实现



DataAgentBase 根据 AISHU Data Agent 的 Open API 定义实现 ChatKitInterface 中的抽象接口。

Copilot 是右侧跟随的 AI 助手，继承 DataAgentBase 的 API 实现以及 CopilotBase 的交互逻辑和界面。

Assistant 是主 AI 对话入口，继承 DataAgentBase 的 API 实现以及 AssistantBase 的交互逻辑和界面。

2.1 class ChatKitDataAgent

ChatKitDataAgent 组件实现 ChatKitInterface 以下方法：

- `generateConversation()`：创建新的会话。
- `getOnboardingInfo()`：获取会话开场白信息。
- `sendMessage()`：发送消息给 AI 助手。
- `reduceAssistantMessage()`：从 EventStream 中提取出对 `action` 和 `content`，并根据 `action` 将 `content` 增量更新到 AssistantMessage。
- `shouldRefreshToken()`：判断 API 响应的状态码是否是 401，如果是，则表示需要刷新 Token。
- `terminateConversation()`：终止会话
- `getConversations()`：获取历史会话列表
- `getConversationMessages()`：加载指定历史会话消息。注意，获取会话消息后，需要对 `Messages[i].content` 进行反序列化获取消息结构。
- `deleteConversation()`：删除指定会话。

三、处理 EventStream

1、EventStream 的数据结构

EventStream 由多条 Event Message 组成，每条 Event Message 包含一个 `seq_id` 属性用于标记 Event Message 的顺序。

每一条 Event Mesage 都是一个 JSON 对象，表示一次对 AssistantMessage 对象的更新操作。一条 Event Message 包含 `seq_id`、`key`、`action`、`content` 四个属性：

- `seq_id`：Event Message 序号。
- `key`：要操作的 AssistantMessage 属性的路径的数组表示，需要转换为 JSONPath 后对 AssistantMessage 进行操作。例如：`["message", "content", "middle_answer", "progress", 0]` 转换为 JSONPath 后是 `"message.content.middle_answer.progress[0]"`。
- `action`：表示对 AssistantMessage 执行的操作动作：
 - `upsert` 表示在 JSONPath 路径插入数据
 - `append` 表示在 JSONPath 路径原有数据后追加内容，有两种情况会 append：
 - 如果 JSONPath 路径是一个数组下标，则在数组下标位置插入新的对象
 - 否则 JSONPath 路径表示 AssistantMessage 的某个文本类型的属性，在文本后追加内容

- `end` 表示 EventStream 结束
- `content`：表示要 `upsert` 或 `append` 的内容

2、AssistantMessage 对象的数据结构

AssistantMessage 对象的数据结构与 `agent-app.schemas.yaml#/components/schemas/Message` 的定义保持一致。

3、处理流程

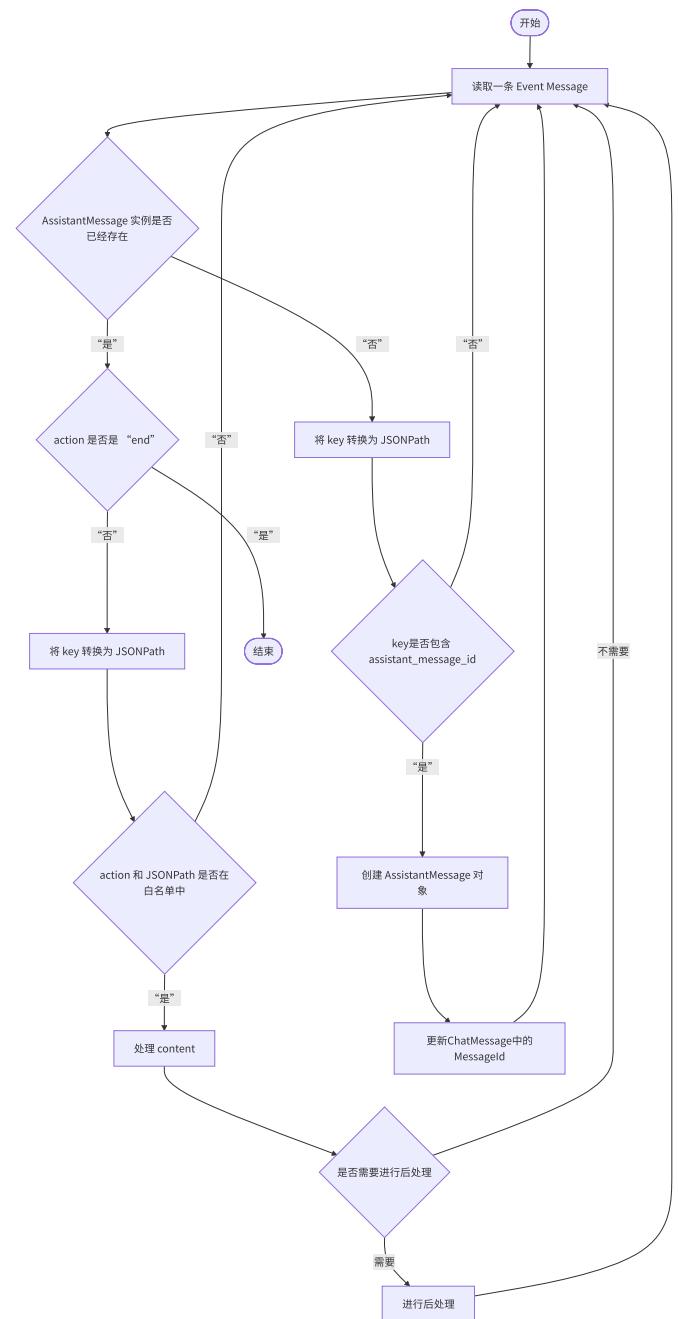
3.1 流程图

调用 Data Agent 的 `chat/completion` 对话接口时，接口以 EventStream 流的形式输出数据。前端需要不断接收 Event Message 并根据 `action` 来将 `content` 增量更新到 AssistantMessage 中，具体处理流程如下：

```

1 flowchart
2
3 Start([开始])
4 Init[创建 AssistantMessage 对象]
5 HasInit{AssistantMessage 实例是否已经存在}
6 Read[读取一条 Event Message]
7 Key2JSONPath[将 key 转换为 JSONPath]
8 Key2JSONPath1[将 key 转换为 JSONPath]
9 CheckWL{action 和 JSONPath 是否在白名单中}
10 ProcContent[处理 content]
11 JsonPathIsAMI{key是否包含 assistant_message_id}
12 UpdateChatMessage[更新ChatMessage中的MessageId]
13 NeedPostProc{是否需要进行后处理}
14 DoPostProc[进行后处理]
15 IsEnd{action 是否是 “end”}
16 End([结束])
17
18 Start([开始]) --> Read
19 Read --> HasInit
20 HasInit --“否”--> Key2JSONPath
21 Key2JSONPath --> JsonPathIsAMI
22 JsonPathIsAMI --“是”--> Init
23 JsonPathIsAMI --“否”--> Read
24 Init -->UpdateChatMessage
25 UpdateChatMessage --> Read
26 HasInit --“是”--> IsEnd
27 IsEnd --“是”--> End
28 IsEnd --“否”--> Key2JSONPath1
29 Key2JSONPath1 --> CheckWL
30 CheckWL --“否”--> Read
31 CheckWL --“是”--> ProcContent
32 ProcContent -->
33 NeedPostProc --"需要"--> DoPostProc
34 DoPostProc --> Read
35 NeedPostProc --"不需要"--> Read
36

```



3.2 Event Message 白名单

判断 Event Message 包含的 action 和 JSONPath 组合是否在以下名单中：

- 如果在，则根据 action 来更新 AssistantMessage 对象，并在需要时执行后处理流程。
- 如果不在，则跳过该条 Event Message。

	JSONPath	如何处理 content	后处理

action			
upsert	error	将 content 赋值到 JSONPath	不需要
upsert	message	将 content 赋值到 JSONPath	不需要
upsert	assistant_message_id	将content赋值给 ChatMessage.messageId	不需要
append	message.content.middle_answer.progress[i]	将 content 赋值到 JSONPath 表示的数组下标位置	<ul style="list-style-type: none"> • 如果 content.stage 是 “skill” : ◦ 如果 content.skill_info.name 是 zhipu_search_tool , 从 content.answer.choices 解析出 WebSearchQuery 结构并调用 appendWebSearchBlock() 将 Web 搜索结果输出到界面 ◦ 如果 content.skill_info.name 是 json2plot , 将 content.skill_info.args 解析出 ChartDataSchema 结构并调用 appendJson2plotBlock() 将结果输出到界面

- 如果
 `content.skill_info.name`
 是 `execute_code` , 从
 `content.skill_info.args` 解
 析出
 `ExecuteCodeResult.input` ,
 从
 `content.answer.result.resu`
 `lt.stdout` 解析出
 `ExecuteCodeResult.output` ,
 然后调用
 `appendExecuteCode()` 将代码
 工具执行结果输出到界面
- 如果
 `content.skill_info.name`
 是 `text2sql` , 从
 `content.skill_info.args` 解
 析出 `Text2sqlResult` , 然后调
 用 `appendText2sql()` 结果输出
 到界面
- 如果
 `content.skill_info.name`
 是 `text2metric` , 从
 `content.skill_info.args` 解
 析出 `text2metricResult` , 然
 后调用 `appendText2metric()`
 结果输出到界面
- 如果
 `content.skill_info.name`
 是 `af_sailor` , 从
 `content.answer.result` 中解
 析出 `cites` , 然后调用
 `AppendAfSailor()` 结果输出到
 界面
- 如果
 `content.skill_info.name`
 是 `datasource_filter` , 从
 `content.answer.result` 中解
 析出 `result` , 然后调用
 `AppendDatasourceFilter()`
 结果输出到界面

- 如果 `content.skill_info.name.toLowerCase()` 是 `"search_memory"`, `"_date"`, `"build_memory"` 直接忽略, 不输出到界面
- 否则将 `content.skill_info.name` 输出到界面
- 如果 `content.stage` 是 `"llm"`, 调用 `appendMarkdownBlock(message.content.middle_answer.progress[i].answer)` 将内容输出到界面

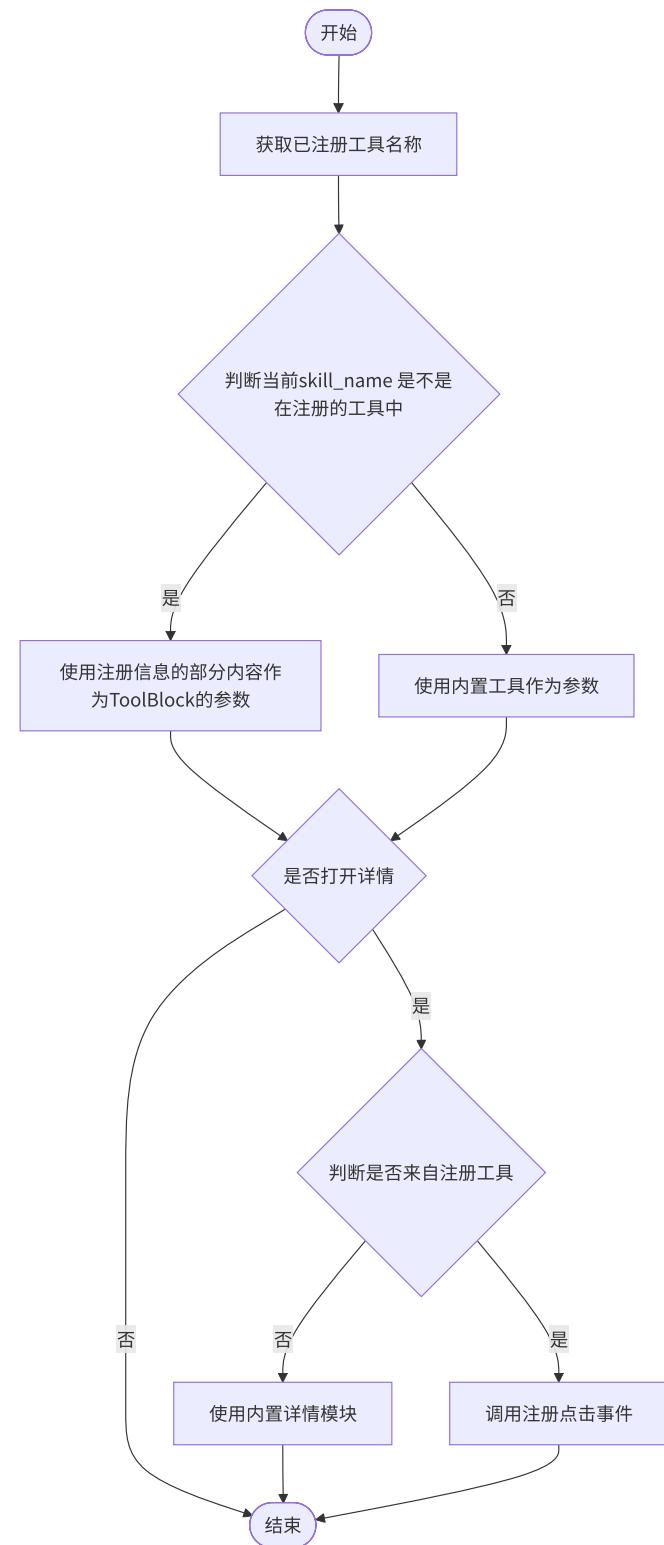
<code>app end</code>	<code>message.content.middle_answer.progress[i].answer</code>	将 content 追加到 JSONPath 现有内容后	• 调用 <code>appendMarkdownBlock(message.content.middle_answer.progress[i].answer)</code> 将内容输出到界面
--------------------------	---	------------------------------	--

3.3 Block工具处理流程

```

1 flowchart
2
3 Start([开始])
4 getAllToolNames[获取已注册工具名称]
5 checkSkillName{判断当前skill_name 是不是在
6 loadToolBlockUseRegistry[使用注册信息的部
7 LoadToolBlockUseDefault[使用内置工具作为参
8 isUseDetail{是否打开详情}
9 useDefaultDetail[使用内置详情模块]
10 execRegistryOnClick[调用注册点击事件]
11 isRegitryLoadDetail{判断是否来自注册工具}
12 End([结束])
13
14 Start([开始]) --> getAllToolNames
15 getAllToolNames --> checkSkillName
16 checkSkillName -- "是" --> loadToolB
17 checkSkillName -- "否" --> LoadToolB
18 loadToolBlockUseRegistry -->isUseDet
19 LoadToolBlockUseDefault --> isUseDet
20 isUseDetail -- "否" --> End
21 isUseDetail -- "是" --> isRegitryLo
22 isRegitryLoadDetail -- "否" --> useD
23 useDefaultDetail --> End
24 isRegitryLoadDetail -- "是" --> execR
25 execRegistryOnClick --> End
26
27
28
29
30
31

```



4、示例

4.1 插入对象

操作前：

代码块

```
1  { }
```

Event Message:

代码块

```
1  {
2      "seq": 0,
3      "key": ["message"],
4      "action": "upsert",
5      "content": {
6          "content": {
7              "middle_answer": {
8                  "progress": []
9              }
10         }
11     }
12 }
```

操作后:

代码块

```
1  {
2      "message": {
3          "content": {
4              "middle_answer": {
5                  "progress": []
6              }
7          }
8      }
9 }
```

4.2 追加到数组

操作前:

代码块

```
1  {
2      "message": {
3          "content": {
4              "middle_answer": {
5                  "progress": []
6              }
7          }
8      }
9 }
```

```
7      }
8      }
9  }
```

Event Message:

代码块

```
1  {
2    "seq": 2,
3    "key": ["message", "content", "middle_answer", "progress", 0],
4    "action": "append",
5    "content": {
6      "stage": "llm",
7      "answer": "我来帮您"
8    }
9  }
```

操作后：

代码块

```
1  {
2    "message": {
3      "content": {
4        "middle_answer": {
5          "progress": [
6            {
7              "stage": "llm",
8              "answer": "我来帮您"
9            }
10           ]
11         }
12       }
13     }
14  }
```

四、解析历史对话消息

读取指定会话 ID 的消息列表后，如果匹配到 `origin` 为 `assistant` 的消息，按以下流程解析处理内容：

- 首先需要对 `content` 进行 JSON 反序列化，解析出 `middle_answer`。

2. 遍历 `middle_answer.progress` 数组，对其中每个元素按照 Event Message 白名单表格中的“后处理”流程渲染到界面。