ai-agent-2-剩余功能

在完成预热以后, 我们就正式进入到用户直接面向的核心功能: 对话和定时任务

aiAgentChat

对话分成流式对话和普通对话两种

普通对话-aiAgentChat方法

获取到完整的AI的回答以后再给出返回

1. 获取所有的client的ID

```
    List<Long> aiClientIds =
repository.queryAiClientIdsByAiAgentId(aiAgentId);

    String content = "";
```

2. 根据clientId取出来对应的client Bean对象, 链式调用这个agent对应的client们, 渐进式提问
    a. 这里传入的advisor的参数, `CHAT_MEMORY_CONVERSATION_ID_KEY`是用于标识这次对话的ID, 这个ID是用来区分上下文的, 不同的对话有着不同的ID不同的上下文
    b. `param(CHAT_MEMORY_RETRIEVE_SIZE_KEY, 100))` 这个参数的作用是将对话历史记录的数量限制在100条

```
for (Long aiClientId : aiClientIds) {
    ChatClient chatClient =
defaultArmoryStrategyFactory.chatClient(aiClientId);

    content = chatClient.prompt(message + ", " + content)
        .system(s -> s.param("current_date",
LocalDate.now().toString()))
        .advisors(a -> a
                .param(CHAT_MEMORY_CONVERSATION_ID_KEY, "chatId-
101")
                .param(CHAT_MEMORY_RETRIEVE_SIZE_KEY, 100))
        .call().content();
}
```

> 这里为什么会是链式调用, 一个问题调用一个*Client*进行回答不就行了吗?
>
> 注意我们的方法名不是*clientChat*而是*agentChat*, 一个*agent*能够串联多个*client*, 在这个方法中多*client*协作的工作共同构建最后的回答, 实现功能互补或者质量提升, 亦或是问题分解, 举例:
>
> 场景一：专业分工协作
>
> 假设有*3*个智能体：
> // *aiClientId* = *1*: 文本分析专家
> // *aiClientId* = *2*: 逻辑推理专家
> // *aiClientId* = *3*: 总结归纳专家
>
> // 第一轮：分析专家处理原始问题
> *content* = ""; // 初始为空
> *content* = 分析专家.*process*("用户问题" + "， " + ""); // 得到分析结果
>
> // 第二轮：推理专家基于分析结果进行推理
> *content* = 推理专家.*process*("用户问题" + "， " + "分析结果"); // 得到推理结论
>
> // 第三轮：总结专家整合前面的结果
> *content* = 总结专家.*process*("用户问题" + "， " + "分析结果+推理结论"); // 最终答案
>
> 场景二：渐进式优化

```
// 智能体链逐步完善答案质量
// 第一个智能体：给出初步答案
// 第二个智能体：基于初步答案进行补充和优化
// 第三个智能体：进行最终校验和润色
```

流式对话-aiAgentChatStream方法

获取到一部分返回以后立马呈现给用户

1. 获取配置

```
// 查询模型ID
Long modelId = repository.queryAiClientModelIdByAgentId(aiAgentId);

// 获取对话模型
ChatModel chatModel =
defaultArmoryStrategyFactory.chatModel(modelId);
```

2. 构建messages

a. 如果对话不携带rag, 则messages就只有传入的message

```
else {
            messages.add(new UserMessage(message));
        }
```

b. 如果携带rag

i. 根据ragId获取到tag

ii. 从pg中查询到相近的documents

```
SearchRequest searchRequest =
SearchRequest.builder()
    .query(message)
    .topK(5)
    .filterExpression("knowledge == '" + tag
+ "'")
    .build();

List<Document> documents =
vectorStore.similaritySearch(searchRequest);
```

iii. 构建结构化的rag prompt

```
String documentCollectors =
documents.stream()
    .map(Document::getFormattedContent)
    .collect(Collectors.joining());

Message ragMessage = new
SystemPromptTemplate("""
....
DOCUMENTS:
{documents} """)
.createMessage(Map.of("documents",
documentCollectors));


messages.add(new UserMessage(message));
```

3. 使用messages与第一步获取到的model对话

```
return chatModel.stream(Prompt.builder()
                        .messages(messages)
                        .build());
```

> 为什么这里又不使用链式调用多个*client*了?
>
> 链式调用多个*client*的本质是将当前*client*的输出 + 原问题作为下一个*client*的输入, 如果这样我们必须阻塞式等待*clien*完整回答完毕, 而当前方法实现的是流式回答, 是输出了一部分就给用户呈现一部分, 所以不能链式调用

AiAgentRagService

里面就上传RagFile一个方法, 用于将知识库文件上传

storeRagFile

1. 遍历所有的文件

2. 读取文件
   `TikaDocumentReader documentReader = new TikaDocumentReader(file.getResource());`
3. 将文件切分成一个个document
   `documentList.forEach(doc -> doc.getMetadata().put("knowledge", tag));`
4. 为文件添加知识库标签
   `documentList.forEach(doc -> doc.getMetadata().put("knowledge", tag));`
5. 存储到pg数据库中
   `vectorStore.accept(documentList);`
6. 将这个我们新建的知识库的tag和名字存储进数据库中, 用于chat的时候查询到我们可用的知识库有哪些

```
AiRagOrderVO aiRagOrderVO = new AiRagOrderVO();
aiRagOrderVO.setRagName(name);
aiRagOrderVO.setKnowledgeTag(tag);
repository.createTagOrder(aiRagOrderVO);
```

AgentTaskJob

这里略过了TaskService的两个查询的方法, 一个是查询所有有效的任务, 一个是查询所有无效的任务的ID

init方法-在Bean初始化以后执行

这个方法上有个@PostConstruct注解, 这个注解的功能是在**Spring**容器初始化完成后执行该方法

这个方法的主要功能就是初始化了一个任务调度器

```
public void init() {
    // 初始化任务调度器
    ThreadPoolTaskScheduler scheduler = new
ThreadPoolTaskScheduler(); // 这个类是 Spring 提供的一个线程池任务调度器
    scheduler.setPoolSize(10);
    scheduler.setThreadNamePrefix("agent-task-scheduler-");
    scheduler.setWaitForTasksToCompleteOnShutdown(true); // 设置在关
闭时等待任务完成
    scheduler.setAwaitTerminationSeconds(60); // 设置等待任务完成的时间
    scheduler.initialize();
    this.taskScheduler = scheduler;
}
```

executeTask方法-执行任务

1. 获取任务的参数(json格式)
   `String taskParam = task.getTaskParam();`
2. 执行任务
   `aiAgentChatService.aiAgentChat(task.getAgentId(), taskParam);`

> 这里的执行任务为什么就是*aiAgentChat?*
>
> 在前面的*xfg*的*mcp*章节, 其实就能看到我们让*AI*执行一个任务就是与*AI*对话, 让它调用对应的*MCP*执行特定的任务
>
> 这里也是一样的, 同时*agent*的链式调用*client*还能使得我们这个任务由多个*client*协同执行

scheduleTask方法-调度器执行任务

1. 创建任务调度器
   a. 调度器要执行的方法就是executeTask(task)
   b. 时间设置通过task的cron表达式

```
ScheduledFuture<?> future = taskScheduler.schedule(
    () -> executeTask(task),
    new CronTrigger(task.getCronExpression())
);
```

2. 将这个任务放到类全局map中, 记录这个任务已经被我们调度执行成功了
   `scheduledTasks.put(task.getId(), future);`

refreshTasks方法-移除invalid任务, 执行valid任务

@Scheduled(fixedRate = 60000) // 每分钟执行一次

1. 从taskService中查询到所有有效的任务配置

```
List<AiAgentTaskScheduleVO> taskSchedules =
aiAgentTaskService.queryAllValidTaskSchedule();
```

2. 处理每个任务

   a. 将这个任务的id放入到Map中, 用于后面删除调度器中已经invalid任务
   b. 如果任务不存在调用scheduleTask(task)创建并调度新任务

```
for (AiAgentTaskScheduleVO task : taskSchedules) {
    Long taskId = task.getId();
    currentTaskIds.put(taskId, true);

    // 如果任务已经存在，则跳过
    if (scheduledTasks.containsKey(taskId)) {
        continue;
    }

    // 创建并调度新任务
    scheduleTask(task);
}
```

3. 移除不存在的任务

   a. 获取现在所有在调度器中的任务的keySet, 也就是所有在调度器中的任务的taskId集合A
   b. 上面处理每个任务的时候记录的现在仍然有效的taskId集合B, 如果A不在B中, 说明这个任务已经不存在了
   c. 从调度器task集合中获取任务并将任务移除

```
        scheduledTasks.keySet().removeIf(taskId -> {
            if (!currentTaskIds.containsKey(taskId)) {
                ScheduledFuture<?> future =
        scheduledTasks.remove(taskId);
                if (future != null) {
                    future.cancel(true);
                    log.info("已移除任务，ID: {}", taskId);
                }
                return true;
            }
            return false;
        });
```

cleanInvalidTasks方法-清理已经被标记为无效的任务

@Scheduled(cron = "0 0/10 * * * ?") // 每10分钟执行一次

1. 获取所有已经失效的任务的ID
   `List<Long> invalidTaskIds = aiAgentTaskService.queryAllInvalidTaskScheduleIds();`
2. 从调度器中移除这些任务

```
        for (Long taskId : invalidTaskIds) {
            ScheduledFuture<?> future =
        scheduledTasks.remove(taskId);
            if (future != null) {
                future.cancel(true);
                log.info("已移除无效任务，ID: {}", taskId);
            }
        }
```

总结

chat和task模块可以说是看似简单的实现, 但是具有很强的拓展性, 比如我想设置一个监控数据监控面板然后发现异常微信推送消息的组件

1. 实现能从数据监控面板获取数据的MCP工具
2. 实现发送微信消息的MCP工具
3. 注册一个分析数据监控面板client, 注册一个发送总结分析发送微信消息的client
4. 将两个client串联成一个agent, 设置这个agent的定时任务
5. 这样就实现了监控数据面板在AI分析即将出现问题的时候通知开发者的功能