《Ai Agent》第3-11节：Agent执行链路设计

来自： 码农会锁

小傅哥

2025年08月08日 08:30

本章重点 ： ★★★★☆
课程视频 ： https://t.zsxq.com/hNFqE
代码分支 ： https://gitcode.net/KnowledgePlanet/ai-agent-station-study/-/tree/3-11-agent-exec-function
工程代码 ： https://gitcode.net/KnowledgePlanet/ai-agent-station-study

作者：小傅哥
博客：https://bugstack.cn

> 沉淀、分享、成长，让自己和他人都能有所收获！😄

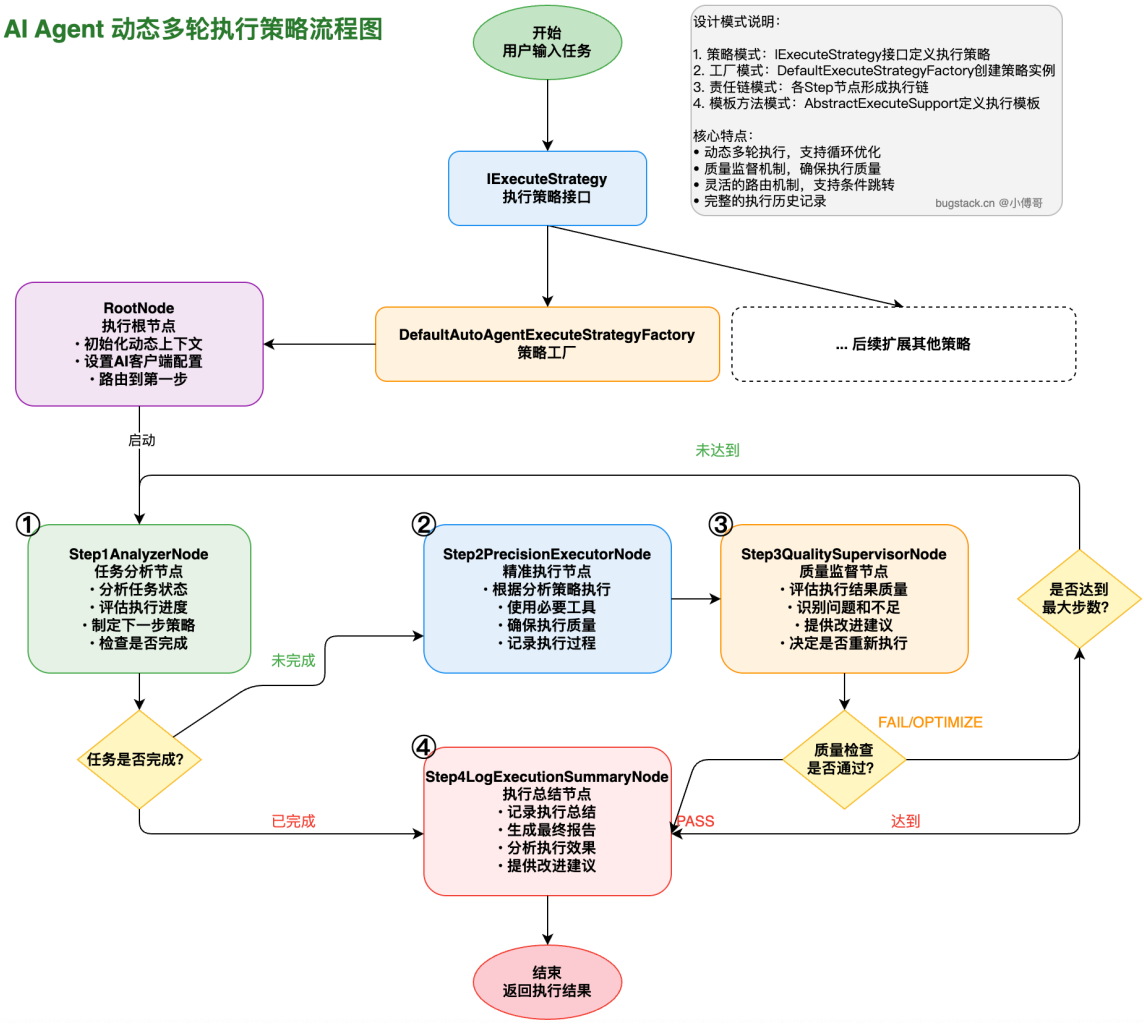
一、本章诉求

将上一节对 Ai Agent 执行链路的分析，以及对应的 AutoAgentTest 测试代码，使用规则树设计可执行链路节点。

本节是其中的一个 Ai Agent Auto 自动执行策略，后续还要把其他的 Ai Agent 执行策略也加入进来实现。

二、流程设计

如图，Auto Ai Agent 动态多轮会话执行流程图；



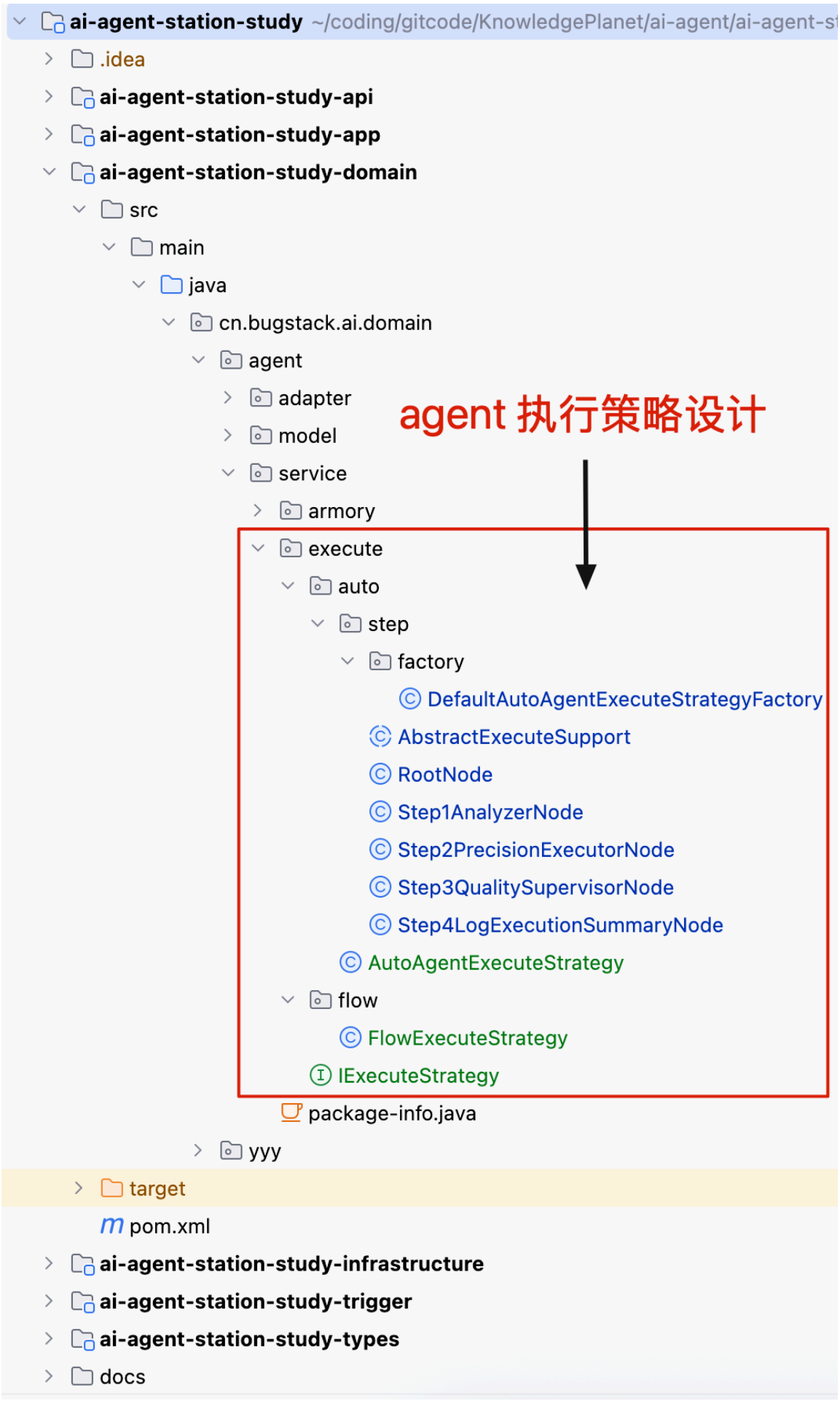
首先，给入口保留一个多策略选择，以适应我们不同场景的多类型 Agent 选择使用，后续会在 agent 配置表增加策略选择属性来区分调用。本节我们先处理一个 AutoAgent 的实现。

之后，进入到关键地方，在上一节 AutoAgentTest 章节，设计了一套自动化 Agent 执行方法，通过 for 循环处理。这里我们通过规则树，分多个多个节点步骤执行，节点间可循环调用，增强整体的灵活性。

最后，以用户提问到所有的步骤执行完成后，进入到结束环节，产生结果。如果你上一节已经高透彻，那么到这里其实会更加容易理解对于节点的拆分。

三、编码实现

1. 工程结构



如图，armory 是装配 agent 所需的元素，execute 是执行 agent 所需的内核。这部分要想清楚。

之后，本节我们先来实现 auto 自动型的 agent 步骤，也就是把 AutoAgentTest 拆成各个节点。节点包括；Root 根节点，用于数据加载，之后是操作步骤节点 Step1~4，分别执行各自的流程步骤。1-任务分析、2-精准执行、3-质量监督、4-执行总结。这块的步骤不要

过于拘泥，如果市面上各大厂还有更好的步骤分享，也可以新增加一个套执行过程步骤。类似你看到的我们用过的一些 Agent 也有是有版本迭代升级的，付费后就更好用。

2. 修改说明

ai_agent_flow_config 表，增加 client_name、client_type，用于 AutoAgent 调度不同类型的对话客户端。同时修改对应程序里的 dao、po、mapper。

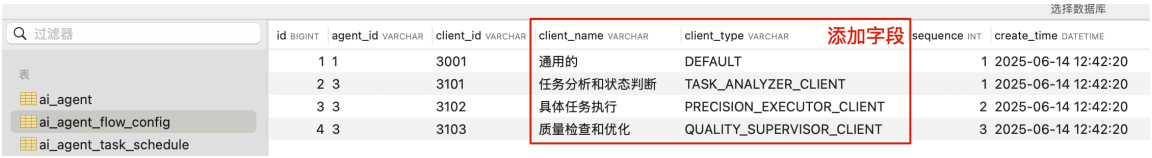
在 IAgentRepository 增加一个 queryAiAgentClientFlowConfig(String aiAgentId) 方法，来查询 Agent 下，配置的 Client 节点。

在 domain agent model 模型下，把 valobj 下的枚举，专门增加一个枚举的包 enums 存放，方便管理。

增加 ExecuteCommandEntity 执行 agent 请求实体对象，包含，aiAgentId、message、sessionId、maxStep

实现 execute 执行包，以规则树方式实现执行过程，包括；RootNode、Step1AnalyzerNode、Step2PrecisionExecutorNode、Step3QualitySupervisorNode、Step4LogExecutionSummaryNode，各项执行节点。这部分可以查看工程代码。

3. 库表修改



添加 client_name、client_type 字段，以及对应的枚举值，以及代码中增加 AiClientTypeEnumVO 枚举类，匹配这里配置的枚举类型。

4. 增加查询 - 客户端端类型

AgentRepository.java

```
@Override
public Map<String, AiAgentClientFlowConfigVO> queryAiAgentClientFlowConfig(String aiAgentId) {
    if (aiAgentId == null || aiAgentId.trim().isEmpty()) {
        return Map.of();
    }
    try {
        // 根据智能体ID查询流程配置列表
        List<AiAgentFlowConfig> flowConfigs = aiAgentFlowConfigDao.queryByAgentId(aiAgentId);

        if (flowConfigs == null || flowConfigs.isEmpty()) {
            return Map.of();
        }

        // 转换为Map结构，key为clientId，value为AiAgentClientFlowConfigVO
        Map<String, AiAgentClientFlowConfigVO> result = new HashMap<>();

        for (AiAgentFlowConfig flowConfig : flowConfigs) {
            AiAgentClientFlowConfigVO configVO = AiAgentClientFlowConfigVO.builder()
                    .clientId(flowConfig.getClientId())
                    .clientName(flowConfig.getClientName())
                    .clientType(flowConfig.getClientType())
                    .sequence(flowConfig.getSequence())
                    .build();

            result.put(flowConfig.getClientType(), configVO);
        }

        return result;
    } catch (NumberFormatException e) {
        log.error("Invalid aiAgentId format: {}", aiAgentId, e);
        return Map.of();
    } catch (Exception e) {
        log.error("Query ai agent client flow config failed, aiAgentId: {}", aiAgentId, e);
        return Map.of();
    }
}
```

增加一个查询客户端类型的接口，key 是类型，value 是对象值。用于 Agent 执行中，查询可参考的 Client 节点。

5. 节点流程 - AutoAgent

5.1 RootNode - 数据加载节点

```
@Slf4j
@Service("executeRootNode")
public class RootNode extends AbstractExecuteSupport {

    @Resource
    private Step1AnalyzerNode step1AnalyzerNode;

    @Override
    protected String doApply(ExecuteCommandEntity requestParameter, DefaultAutoAgentExecuteStrategyFactory.I
        log.info("=== 动态多轮执行测试开始 ====");
        log.info("用户输入: {}", requestParameter.getMessage());
        log.info("最大执行步数: {}", requestParameter.getMaxStep());
        log.info("会话ID: {}", requestParameter.getSessionId());

        Map<String, AiAgentClientFlowConfigVO> aiAgentClientFlowConfigVOMap = repository.queryAiAgentClientI

        // 客户端对话组
        dynamicContext.setAiAgentClientFlowConfigVOMap(aiAgentClientFlowConfigVOMap);
```

```
        // 上下文信息
        dynamicContext.setExecutionHistory(new StringBuilder());
        // 当前任务信息
        dynamicContext.setCurrentTask(requestParameter.getMessage());
        // 最大任务步骤
        dynamicContext.setMaxStep(requestParameter.getMaxStep());

        return router(requestParameter, dynamicContext);
    }

    @Override
    public StrategyHandler<ExecuteCommandEntity, DefaultAutoAgentExecuteStrategyFactory.DynamicContext, Str
        return step1AnalyzerNode;
    }

}
```

第一个操作节点，加载数据。并把数据填充到上下文中。

5.2 Step1AnalyzerNode - 任务分析节点

```
@Slf4j
@Service
public class Step1AnalyzerNode extends AbstractExecuteSupport {

    @Override
    protected String doApply(ExecuteCommandEntity requestParameter, DefaultAutoAgentExecuteStrategyFactory.
        log.info("\n🎯 === 执行第 {} 步 ===", dynamicContext.getStep());

        // 第一阶段：任务分析
        log.info("\n📊 阶段1: 任务状态分析");
        String analysisPrompt = String.format("""
                        **原始用户需求:**  %s

                        **当前执行步骤:**  第 %d 步 (最大 %d 步)

                        **历史执行记录:**
                        %s

                        **当前任务:**  %s

                        请分析当前任务状态，评估执行进度，并制定下一步策略。
                        """,
                requestParameter.getMessage(),
                dynamicContext.getStep(),
                dynamicContext.getMaxStep(),
                !dynamicContext.getExecutionHistory().isEmpty() ? dynamicContext.getExecutionHistory().toSt
                dynamicContext.getCurrentTask()
        );

        // 获取对话客户端
        AiAgentClientFlowConfigVO aiAgentClientFlowConfigVO = dynamicContext.getAiAgentClientFlowConfigVOMa
        ChatClient chatClient = getChatClientByClientId(aiAgentClientFlowConfigVO.getClientId());

        String analysisResult = chatClient
                .prompt(analysisPrompt)
                .advisors(a -> a
                        .param(CHAT_MEMORY_CONVERSATION_ID_KEY, requestParameter.getSessionId())
                        .param(CHAT_MEMORY_RETRIEVE_SIZE_KEY, 1024))
                .call().content();
```

```
        assert analysisResult != null;
        parseAnalysisResult(dynamicContext.getStep(), analysisResult);

        // 将分析结果保存到动态上下文中，供下一步使用
        dynamicContext.setValue("analysisResult", analysisResult);

        // 检查是否已完成
        if (analysisResult.contains("任务状态: COMPLETED") ||
                analysisResult.contains("完成度评估: 100%")) {
            dynamicContext.setCompleted(true);
            log.info("✅ 任务分析显示已完成！");
            return router(requestParameter, dynamicContext);
        }

        return router(requestParameter, dynamicContext);
    }

    @Override
    public StrategyHandler<ExecuteCommandEntity, DefaultAutoAgentExecuteStrategyFactory.DynamicContext, Str
        // 如果任务已完成或达到最大步数，进入总结阶段
        if (dynamicContext.isCompleted() || dynamicContext.getStep() > dynamicContext.getMaxStep()) {
            return getBean("step4LogExecutionSummaryNode");
        }

        // 否则继续执行下一步
        return getBean("step2PrecisionExecutorNode");
    }

    private void parseAnalysisResult(int step, String analysisResult) {
        log.info("\n📊 === 第 {} 步分析结果 ===", step);

        String[] lines = analysisResult.split("\n");
        String currentSection = "";

        for (String line : lines) {
            line = line.trim();
            if (line.isEmpty()) continue;

            if (line.contains("任务状态分析:")) {
                currentSection = "status";
                log.info("\n🎯 任务状态分析:");
                continue;
            } else if (line.contains("执行历史评估:")) {
                currentSection = "history";
                log.info("\n📈 执行历史评估:");
                continue;
            } else if (line.contains("下一步策略:")) {
                currentSection = "strategy";
                log.info("\n🚀 下一步策略:");
                continue;
            } else if (line.contains("完成度评估:")) {
                currentSection = "progress";
                String progress = line.substring(line.indexOf(":") + 1).trim();
                log.info("\n📊 完成度评估: {}", progress);
                continue;
            } else if (line.contains("任务状态:")) {
                currentSection = "task_status";
                String status = line.substring(line.indexOf(":") + 1).trim();
                if (status.equals("COMPLETED")) {
                    log.info("\n✅ 任务状态: 已完成");
                } else {
                    log.info("\n🔄 任务状态: 继续执行");
                }
                continue;
```

```
            }

            switch (currentSection) {
                case "status":
                    log.info("   📋 {}", line);
                    break;
                case "history":
                    log.info("   📊 {}", line);
                    break;
                case "strategy":
                    log.info("   🎯 {}", line);
                    break;
                default:
                    log.info("   📝 {}", line);
                    break;
            }
        }
    }

}
```

这一步是把 AutoAgentTest 第一步提取出来，之后设置执行步骤。analysisPrompt 这一部分操作，比较吃大模型的能力，有些模型上下文 token 能力不足，也可能出现幻觉。

之后进行客户端对话，使用 chatClient 进行对话操作。对话后 parseAnalysisResult 分析下一步执行结果，这一步是 for 循环操作。

最后进行完成度的检查，如果完成了，`analysisResult.contains("任务状态: COMPLETED")` 则设置 `dynamicContext.setCompleted(true);` 完成。之后路由后续节点。

5.3 Step2PrecisionExecutorNode - 精准执行节点

```
@Slf4j
@Service
public class Step2PrecisionExecutorNode extends AbstractExecuteSupport{

    @Override
    protected String doApply(ExecuteCommandEntity requestParameter, DefaultAutoAgentExecuteStrategyFactory.I
        log.info("\n⚡ 阶段2: 精准任务执行");

        // 从动态上下文中获取分析结果
        String analysisResult = dynamicContext.getValue("analysisResult");
        if (analysisResult == null || analysisResult.trim().isEmpty()) {
            log.warn("⚠ 分析结果为空，使用默认执行策略");
            analysisResult = "执行当前任务步骤";
        }

        String executionPrompt = String.format("""
                **分析师策略:**  %s

                **执行指令:**  根据上述分析师的策略，执行具体的任务步骤。

                **执行要求:**
                1. 严格按照策略执行
                2. 使用必要的工具
                3. 确保执行质量
                4. 详细记录过程

                **输出格式:**
                执行目标: [明确的执行目标]
                执行过程: [详细的执行步骤]
                执行结果: [具体的执行成果]
                质量检查: [自我质量评估]
```

```
                """, analysisResult);

        // 获取对话客户端
        AiAgentClientFlowConfigVO aiAgentClientFlowConfigVO = dynamicContext.getAiAgentClientFlowConfigVOMa
        ChatClient chatClient = getChatClientByClientId(aiAgentClientFlowConfigVO.getClientId());

        String executionResult = chatClient
                .prompt(executionPrompt)
                .advisors(a -> a
                        .param(CHAT_MEMORY_CONVERSATION_ID_KEY, requestParameter.getSessionId())
                        .param(CHAT_MEMORY_RETRIEVE_SIZE_KEY, 1024))
                .call().content();

        parseExecutionResult(dynamicContext.getStep(), executionResult);

        // 将执行结果保存到动态上下文中，供下一步使用
        dynamicContext.setValue("executionResult", executionResult);

        // 更新执行历史
        String stepSummary = String.format("""
                === 第 %d 步执行记录 ===
                【分析阶段】%s
                【执行阶段】%s
                """, dynamicContext.getStep(), analysisResult, executionResult);

        dynamicContext.getExecutionHistory().append(stepSummary);

        return router(requestParameter, dynamicContext);
    }

    @Override
    public StrategyHandler<ExecuteCommandEntity, DefaultAutoAgentExecuteStrategyFactory.DynamicContext, Str
        return getBean("step3QualitySupervisorNode");
    }

    /**
     * 解析执行结果
     */
    private void parseExecutionResult(int step, String executionResult) {
        log.info("\n⚡ === 第 {} 步执行结果 ===", step);

        String[] lines = executionResult.split("\n");
        String currentSection = "";

        for (String line : lines) {
            line = line.trim();
            if (line.isEmpty()) continue;

            if (line.contains("执行目标:")) {
                currentSection = "target";
                log.info("\n🎯 执行目标:");
                continue;
            } else if (line.contains("执行过程:")) {
                currentSection = "process";
                log.info("\n🔧 执行过程:");
                continue;
            } else if (line.contains("执行结果:")) {
                currentSection = "result";
                log.info("\n📈 执行结果:");
                continue;
            } else if (line.contains("质量检查:")) {
                currentSection = "quality";
                log.info("\n🔍 质量检查:");
                continue;
```

```
            }

            switch (currentSection) {
                case "target":
                    log.info("   🎯 {}", line);
                    break;
                case "process":
                    log.info("   ⚙️ {}", line);
                    break;
                case "result":
                    log.info("   📊 {}", line);
                    break;
                case "quality":
                    log.info("   ✅ {}", line);
                    break;
                default:
                    log.info("   📝 {}", line);
                    break;
            }
        }
    }

}
```

这一个节点，执行的是具体的任务步骤操作，executionPrompt 同时明确输出格式。这一步限定好以后，开始进行对话操作，并拿到 executionResult 结果。

这根据结果获取到；执行目标、执行过程、执行结果、质量检查，并做输出。后续这部分的输出是可以传递给前端页面的。

记录好，stepSummary 后，路由到下一个节点，step3QualitySupervisorNode

5.4 Step3QualitySupervisorNode - 质量监督节点

```
@Slf4j
@Service
public class Step3QualitySupervisorNode extends AbstractExecuteSupport {

    @Override
    protected String doApply(ExecuteCommandEntity requestParameter, DefaultAutoAgentExecuteStrategyFactory.I
        // 第三阶段：质量监督
        log.info("\n🔍 阶段3: 质量监督检查");

        // 从动态上下文中获取执行结果
        String executionResult = dynamicContext.getValue("executionResult");
        if (executionResult == null || executionResult.trim().isEmpty()) {
            log.warn("⚠️ 执行结果为空，跳过质量监督");
            return "质量监督跳过";
        }

        String supervisionPrompt = String.format("""
                **用户原始需求:**  %s

                **执行结果:**  %s

                **监督要求:**  请评估执行结果的质量，识别问题，并提供改进建议。

                **输出格式:**
                质量评估: [对执行结果的整体评估]
                问题识别: [发现的问题和不足]
                改进建议: [具体的改进建议]
                质量评分: [1-10分的质量评分]
                是否通过: [PASS/FAIL/OPTIMIZE]
```

```
                """, requestParameter.getMessage(), executionResult);

        // 获取对话客户端
        AiAgentClientFlowConfigVO aiAgentClientFlowConfigVO = dynamicContext.getAiAgentClientFlowConfigVOMa
        ChatClient chatClient = getChatClientByClientId(aiAgentClientFlowConfigVO.getClientId());

        String supervisionResult = chatClient
                .prompt(supervisionPrompt)
                .advisors(a -> a
                        .param(CHAT_MEMORY_CONVERSATION_ID_KEY, requestParameter.getSessionId())
                        .param(CHAT_MEMORY_RETRIEVE_SIZE_KEY, 1024))
                .call().content();

        parseSupervisionResult(dynamicContext.getStep(), supervisionResult);

        // 将监督结果保存到动态上下文中
        dynamicContext.setValue("supervisionResult", supervisionResult);

        // 根据监督结果决定是否需要重新执行
        if (supervisionResult.contains("是否通过: FAIL")) {
            log.info("❌ 质量检查未通过，需要重新执行");
            dynamicContext.setCurrentTask("根据质量监督的建议重新执行任务");
        } else if (supervisionResult.contains("是否通过: OPTIMIZE")) {
            log.info("🔧 质量检查建议优化，继续改进");
            dynamicContext.setCurrentTask("根据质量监督的建议优化执行结果");
        } else {
            log.info("✅ 质量检查通过");
            dynamicContext.setCompleted(true);
        }

        // 更新执行历史
        String stepSummary = String.format("""
                === 第 %d 步完整记录 ===
                【分析阶段】%s
                【执行阶段】%s
                【监督阶段】%s
                """, dynamicContext.getStep(),
                dynamicContext.getValue("analysisResult"),
                executionResult,
                supervisionResult);

        dynamicContext.getExecutionHistory().append(stepSummary);

        // 增加步骤计数
        dynamicContext.setStep(dynamicContext.getStep() + 1);

        // 如果任务已完成或达到最大步数，进入总结阶段
        if (dynamicContext.isCompleted() || dynamicContext.getStep() > dynamicContext.getMaxStep()) {
            return router(requestParameter, dynamicContext);
        }

        // 否则继续下一轮执行，返回到Step1AnalyzerNode
        return router(requestParameter, dynamicContext);
    }

    @Override
    public StrategyHandler<ExecuteCommandEntity, DefaultAutoAgentExecuteStrategyFactory.DynamicContext, Str
        // 如果任务已完成或达到最大步数，进入总结阶段
        if (dynamicContext.isCompleted() || dynamicContext.getStep() > dynamicContext.getMaxStep()) {
            return getBean("step4LogExecutionSummaryNode");
        }

        // 否则返回到Step1AnalyzerNode进行下一轮分析
        return getBean("step1AnalyzerNode");
```

```
    }

    /**
     * 解析监督结果
     */
    private void parseSupervisionResult(int step, String supervisionResult) {
        log.info("\n🔍 === 第 {} 步监督结果 ===", step);

        String[] lines = supervisionResult.split("\n");
        String currentSection = "";

        for (String line : lines) {
            line = line.trim();
            if (line.isEmpty()) continue;

            if (line.contains("质量评估:")) {
                currentSection = "assessment";
                log.info("\n📊 质量评估:");
                continue;
            } else if (line.contains("问题识别:")) {
                currentSection = "issues";
                log.info("\n⚠️ 问题识别:");
                continue;
            } else if (line.contains("改进建议:")) {
                currentSection = "suggestions";
                log.info("\n💡 改进建议:");
                continue;
            } else if (line.contains("质量评分:")) {
                currentSection = "score";
                String score = line.substring(line.indexOf(":") + 1).trim();
                log.info("\n📊 质量评分: {}", score);
                continue;
            } else if (line.contains("是否通过:")) {
                currentSection = "pass";
                String status = line.substring(line.indexOf(":") + 1).trim();
                if (status.equals("PASS")) {
                    log.info("\n✅ 检查结果: 通过");
                } else if (status.equals("FAIL")) {
                    log.info("\n❌ 检查结果: 未通过");
                } else {
                    log.info("\n🔧 检查结果: 需要优化");
                }
                continue;
            }

            switch (currentSection) {
                case "assessment":
                    log.info("   📋 {}", line);
                    break;
                case "issues":
                    log.info("   ⚠️ {}", line);
                    break;
                case "suggestions":
                    log.info("   💡 {}", line);
                    break;
                default:
                    log.info("   📝 {}", line);
                    break;
            }
        }
    }

}
```

此节点主要为了检测整个过程生成的内容质量是否可靠的。他会进行评分、对话（client）、监督，之后更新历史步骤 stepSummary。

注意，router 做完路由后，要判断是走到最终节点，还是继续回到 step1AnalyzerNode 继续执行，这个过程是根据生成内容是否完成目标来决定的（也包括是否超过最大步骤）。

另外，get 方法里里的路由操作，是分不同的类型获取 bean 对象。

5.5 Step4LogExecutionSummaryNode - 执行总结节点

```
@Slf4j
@Service
public class Step4LogExecutionSummaryNode extends AbstractExecuteSupport {

    @Override
    protected String doApply(ExecuteCommandEntity requestParameter, DefaultAutoAgentExecuteStrategyFactory.I
        log.info("\n📊 === 执行第 {} 步 ===", dynamicContext.getStep());

        // 第四阶段：执行总结
        log.info("\n📊 阶段4: 执行总结分析");

        // 记录执行总结
        logExecutionSummary(dynamicContext.getMaxStep(), dynamicContext.getExecutionHistory(), dynamicConte

        // 如果任务未完成，生成最终总结报告
        if (!dynamicContext.isCompleted()) {
            generateFinalReport(requestParameter, dynamicContext);
        }

        log.info("\n🏁 === 动态多轮执行测试结束 ====");

        return "ai agent execution summary completed!";
    }

    @Override
    public StrategyHandler<ExecuteCommandEntity, DefaultAutoAgentExecuteStrategyFactory.DynamicContext, Str
        // 总结节点是最后一个节点，返回null表示执行结束
        return defaultStrategyHandler;
    }

    /**
     * 记录执行总结
     */
    private void logExecutionSummary(int maxSteps, StringBuilder executionHistory, boolean isCompleted) {
        log.info("\n📊 === 动态多轮执行总结 ====");

        int actualSteps = Math.min(maxSteps, executionHistory.toString().split("=== 第").length - 1);
        log.info("📈 总执行步数: {} 步", actualSteps);

        if (isCompleted) {
            log.info("✅ 任务完成状态: 已完成");
        } else {
            log.info("⏸️ 任务完成状态: 未完成（达到最大步数限制）");
        }

        // 计算执行效率
        double efficiency = isCompleted ? 100.0 : (double) actualSteps / maxSteps * 100;
        log.info("📊 执行效率: {:.1f}%", efficiency);
    }

    /**
     * 生成最终总结报告
     */
```

```
private void generateFinalReport(ExecuteCommandEntity requestParameter, DefaultAutoAgentExecuteStrategyI
    try {
        log.info("\n--- 生成未完成任务的总结报告 ---");

        String summaryPrompt = String.format("""
                请对以下未完成的任务执行过程进行总结分析：

                **原始用户需求:**  %s

                **执行历史:**
                %s

                **分析要求:**
                1. 总结已完成的工作内容
                2. 分析未完成的原因
                3. 提出完成剩余任务的建议
                4. 评估整体执行效果
                """,
                requestParameter.getMessage(),
                dynamicContext.getExecutionHistory().toString());

        // 获取对话客户端 - 使用任务分析客户端进行总结
        AiAgentClientFlowConfigVO aiAgentClientFlowConfigVO = dynamicContext.getAiAgentClientFlowConfig
        ChatClient chatClient = getChatClientByClientId(aiAgentClientFlowConfigVO.getClientId());

        String summaryResult = chatClient
                .prompt(summaryPrompt)
                .advisors(a -> a
                        .param(CHAT_MEMORY_CONVERSATION_ID_KEY, requestParameter.getSessionId() + "-sum
                        .param(CHAT_MEMORY_RETRIEVE_SIZE_KEY, 50))
                .call().content();

        logFinalReport(summaryResult);

        // 将总结结果保存到动态上下文中
        dynamicContext.setValue("finalSummary", summaryResult);

    } catch (Exception e) {
        log.error("生成最终总结报告时出现异常: {}", e.getMessage(), e);
    }
}

/**
 * 输出最终总结报告
 */
private void logFinalReport(String summaryResult) {
    log.info("\n📋 === 最终总结报告 ===");

    String[] lines = summaryResult.split("\n");
    for (String line : lines) {
        line = line.trim();
        if (line.isEmpty()) continue;

        // 根据内容类型添加不同图标
        if (line.contains("已完成") || line.contains("完成的工作")) {
            log.info("✅ {}", line);
        } else if (line.contains("未完成") || line.contains("原因")) {
            log.info("❌ {}", line);
        } else if (line.contains("建议") || line.contains("推荐")) {
            log.info("💡 {}", line);
        } else if (line.contains("评估") || line.contains("效果")) {
            log.info("📊 {}", line);
        } else {
            log.info("📝 {}", line);
```

```
            }
        }
    }

}
```

到这一步，就是最终总结产生结果的节点了，生成最终的报告。

四、测试验证

1. 数据配置

注意，要更新课程的最新 SQL 里面配置有对应的数据。

2. 单元测试

```
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest
public class AutoAgentTest {

    @Resource
    private DefaultArmoryStrategyFactory defaultArmoryStrategyFactory;

    @Resource
    private DefaultAutoAgentExecuteStrategyFactory defaultAutoAgentExecuteStrategyFactory;

    @Resource
    private ApplicationContext applicationContext;

    @Before
    public void init() throws Exception {
        StrategyHandler<ArmoryCommandEntity, DefaultArmoryStrategyFactory.DynamicContext, String> armoryStra
                defaultArmoryStrategyFactory.armoryStrategyHandler();

        String apply = armoryStrategyHandler.apply(
                ArmoryCommandEntity.builder()
                        .commandType(AiAgentEnumVO.AI_CLIENT.getCode())
                        .commandIdList(Arrays.asList("3101", "3102", "3103"))
                        .build(),
                new DefaultArmoryStrategyFactory.DynamicContext());

        ChatClient chatClient = (ChatClient) applicationContext.getBean(AiAgentEnumVO.AI_CLIENT.getBeanName
        log.info("客户端构建:{}", chatClient);
    }

    @Test
    public void autoAgent() throws Exception {
        StrategyHandler<ExecuteCommandEntity, DefaultAutoAgentExecuteStrategyFactory.DynamicContext, String
```

```
                = defaultAutoAgentExecuteStrategyFactory.armoryStrategyHandler();

        ExecuteCommandEntity executeCommandEntity = new ExecuteCommandEntity();
        executeCommandEntity.setAiAgentId("3");
        executeCommandEntity.setMessage("搜索小傅哥，技术项目列表。编写成一份文档，说明不同项目的学习目标，以及不同
        executeCommandEntity.setSessionId("session-id-" + System.currentTimeMillis());
        executeCommandEntity.setMaxStep(3);

        String apply = executeHandler.apply(executeCommandEntity, new DefaultAutoAgentExecuteStrategyFactory
        log.info("测试结果:{}", apply);
    }

}
```

温馨提示，最大执行步骤，可以设置的小点。agent 执行比较消耗 token 量。

五、读者作业

简单作业：完成功能的拆解实现，可以运行出对应的结果。

复杂作业：尝试增加 SSE 流式响应接口，与UI对接。