《Ai Agent》第3-6节：数据加载模型设计

来自： 码农会锁

小傅哥

2025年06月29日 11:55

本章重点： ★★★★☆
课程视频： https://t.zsxq.com/USaZS
代码分支：https://gitcode.net/KnowledgePlanet/ai-agent-station-study/-/tree/3-6-load-data-strategy
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！😄

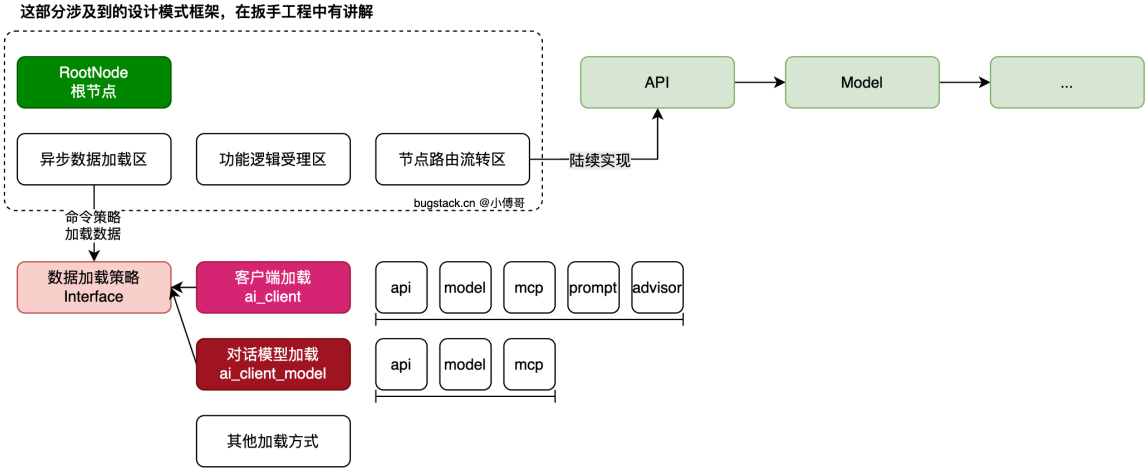
一、本章诉求

在关于 Ai Agent 的功能实现中，有一个非常重要处理步骤，就是要想办法动态的实例化来自于用户配置的；API、对话模型、MCP、顾问角色以及提示词等。这也就是我们前面为什么要基于 ai agent case 案例，把代码抽象出库表配置。

好，那么到这一节，还要思考，怎么让程序来加载和实例化 Ai Agent 所需的各项组件。如，客户端的实例化、对话模型的实例化等。
注意；本节会引入星球组件项目《扳手工程》，通用设计模式框架。可以前置学习：第2节：责任链和规则树通用模型框架

二、功能流程

如图，Ai Agent 实现过程，数据加载策略设计；

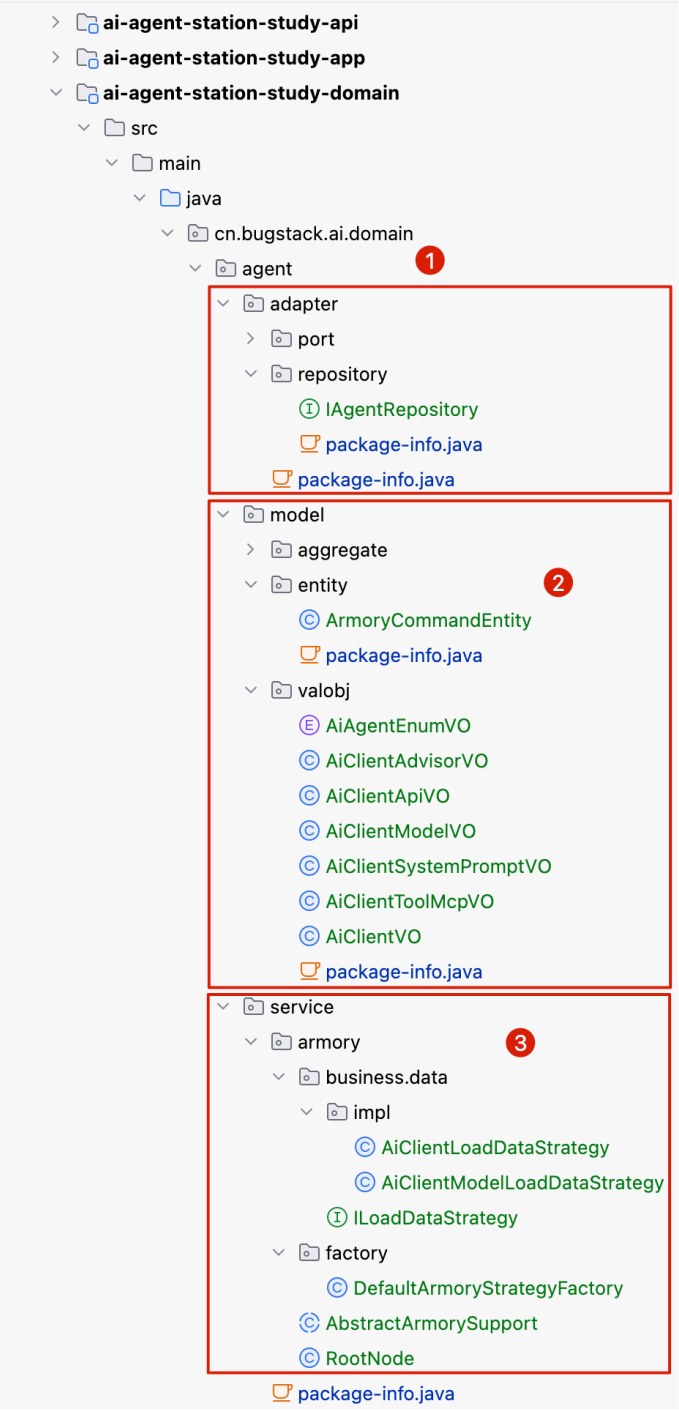


首先，整个 Ai Agent 的实例化过程，就是各项组件的创建和组装的过程。那么，为了让整体的实现代码更易于维护，我们可以把这样的创建过程，通过规则树的方式进行串联实现。而这部分需要的规则树，是不需要重复建设的，因为星球里的《扳手工程组件项目》，已经把这类的共性内容，凝练成了通用的组件，各个业务系统引用使用即可。所以，这部分建议刷下《扳手工程组件项目》，来看第2节：责任链和规则树通用模型框架

之后，本节我们先把目标缩小到关于数据加载部分，因为后续所有的 Ai Agent 组件实例化的过程，都是需要基础数据的提供。所以组装数据就显得尤为重要了。

三、编码实现

1. 工程结构



1. 数据，适配器层。

   也就是整个 agent 需要什么样的数据结构，就定义对应的接口方法。之后由基础设施层，通过 pom 引入 domain 领域层。之后实现适配接口，用 dao、redis、http 等方式完成数据的组装，返回给 domain 领域层。

   这种设计的好处是适配和防腐，如果将来基础层数据有变动，也不会影响到领域层的服务方法。这样的设计是非常好的。

2. 领域对象，这些对象是为了满足 service 服务创建的对象。

3. 具体的逻辑服务实现，这部分要完成 agent 实例化过程。

   本节先完成数据的加载设计。

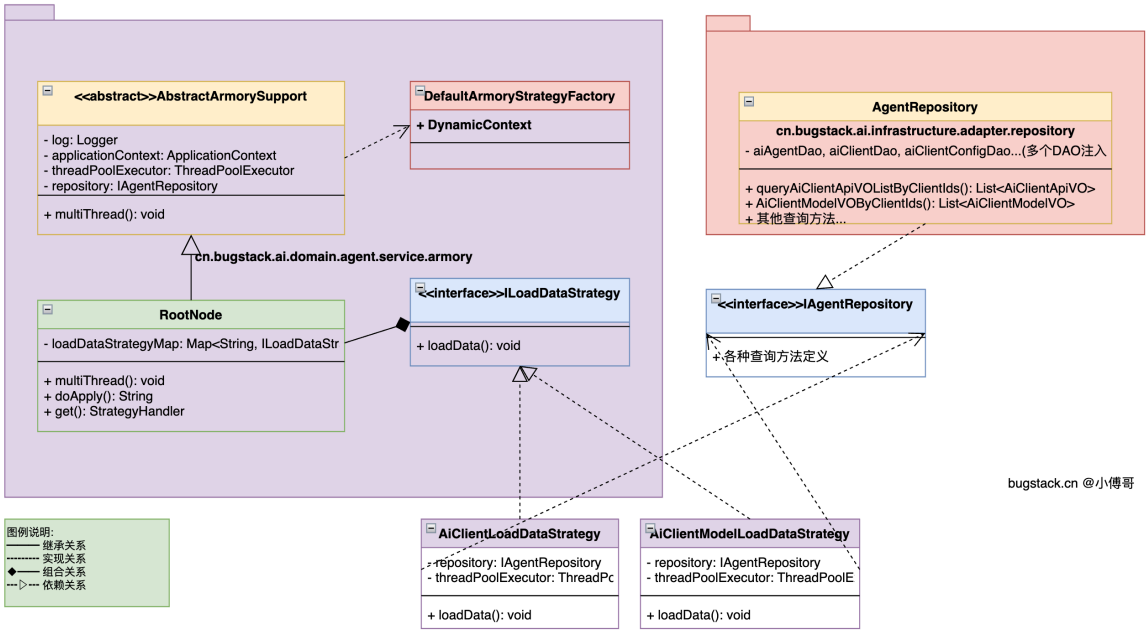
如图，整个 agent 领域，为的是动态化的加载各个模块。所以，这一节要先设计数据的加载设计。

adapter；适配器层，用于完成数据的获取。适配器，也可以理解为水管的接头，适配不同水管的连接。

model；领域对象层，一整个服务的实现，数据需要通过领域对象来传递。基本你需要什么对象，就在这部分创建。关于 DDD 对于这部分的结构，可以扩展学习。https://bugstack.cn/md/road-map/ddd-guide-01.html

service；服务实现层，这部分首先使用到了扳手工程的设计模式组件，用于流程节点的串联，这样会让代码更加干净。其中本节重点是在数据的加载策略上。因为实际使用中，用户可以通过只实例化 api，也可以实例化整个 client，这是不同的操作。所以这块要做一个数据加载策略。

2. 类的关系



首先，抽取后的类的关系结构。以 AbstractArmorySupport 为扩展支撑类，实现第一个 RootNode 节点。

之后，本身 Node 的用途是为了加载数据，但加载数据本身又有很多种类，所以这里要做一个数据加载策略。以 ILoadDataStrategy 接口实现多个数据加载实现类。目前只体现了2个，后续随着功能的开发，再继续增加。

最后，整个数据的加载都是从基础设施层实现类，AgentRepository 通过 dao 操作，完成各项数据的加载。

3. 核心编码

本节的整个编码，只要完成一个线路，其他的就好理解了，因为都是同类的操作。所以，这里只展示核心的代码，其他的可以对照工程代码来看。

3.1 引入POM

```
<!-- 扳手工程（下载后，用 idea 打开，点击 install） https://gitcode.net/KnowledgePlanet/ai-agent-station -->
<dependency>
    <groupId>cn.bugstack.wrench</groupId>
    <artifactId>xfg-wrench-bom</artifactId>
    <version>3.0.0</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>

<!-- 扳手工程，通用设计模式框架 -->
<dependency>
    <groupId>cn.bugstack.wrench</groupId>
    <artifactId>xfg-wrench-starter-design-framework</artifactId>
</dependency>
```

首先在 ai-agent-station-study 根 pom 引入 xfg-wrench-bom，之后在 domain 领域层引入 xfg-wrench-starter-design-framework。

3.2 框架编码（设计模式）

```
public abstract class AbstractArmorySupport extends AbstractMultiThreadStrategyRouter<ArmoryCommandEntity, I

    private final Logger log = LoggerFactory.getLogger(AbstractArmorySupport.class);

    @Resource
    protected ApplicationContext applicationContext;

    @Resource
    protected ThreadPoolExecutor threadPoolExecutor;
```

```
    @Resource
    protected IAgentRepository repository;

    @Override
    protected void multiThread(ArmoryCommandEntity requestParameter, DefaultArmoryStrategyFactory.DynamicCo
        // 缺省的
    }

}

@Slf4j
@Service
public class RootNode extends AbstractArmorySupport {

    private final Map<String, ILoadDataStrategy> loadDataStrategyMap;

    public RootNode(Map<String, ILoadDataStrategy> loadDataStrategyMap) {
        this.loadDataStrategyMap = loadDataStrategyMap;
    }

    @Override
    protected void multiThread(ArmoryCommandEntity requestParameter, DefaultArmoryStrategyFactory.DynamicCo
        // 通过策略加载数据
        String commandType = requestParameter.getCommandType();
        ILoadDataStrategy loadDataStrategy = loadDataStrategyMap.get(commandType);
        loadDataStrategy.loadData(requestParameter, dynamicContext);
    }

    @Override
    protected String doApply(ArmoryCommandEntity requestParameter, DefaultArmoryStrategyFactory.DynamicCont
        return router(requestParameter, dynamicContext);
    }

    @Override
    public StrategyHandler<ArmoryCommandEntity, DefaultArmoryStrategyFactory.DynamicContext, String> get(Ar
        return defaultStrategyHandler;
    }

}
```

这是整个本节反复强调的扳手工程提供的设计模块框架，这部分代码可以阅读扳手工程的第2节深入理解。

3.3 数据加载

3.3.1 定义策略接口

```
public interface ILoadDataStrategy {

    void loadData(ArmoryCommandEntity armoryCommandEntity, DefaultArmoryStrategyFactory.DynamicContext dyna

}
```

```
@Data
public class ArmoryCommandEntity {

    /**
     * 命令类型
     */
```

```
    private String commandType;

    /**
     * 命令索引（clientId、modelId、apiId...）
     */
    private List<String> commandIdList;

}
```

ArmoryCommandEntity，对象用于请求加载数据策略。

因为我们本身加载数据，无非就是告诉策略，你要加载哪类的数据策略，之后给一个 ID 集合（无论加载哪类数据，都是这类 String 类型）。

3.3.2 策略1；Client 数据

```
@Slf4j
@Service("aiClientLoadDataStrategy")
public class AiClientLoadDataStrategy implements ILoadDataStrategy {

    @Resource
    private IAgentRepository repository;

    @Resource
    protected ThreadPoolExecutor threadPoolExecutor;

    @Override
    public void loadData(ArmoryCommandEntity armoryCommandEntity, DefaultArmoryStrategyFactory.DynamicContex
        List<String> clientIdList = armoryCommandEntity.getCommandIdList();

        CompletableFuture<List<AiClientApiVO>> aiClientApiListFuture = CompletableFuture.supplyAsync(() -> {
            log.info("查询配置数据(ai_client_api) {}", clientIdList);
            return repository.queryAiClientApiVOListByClientIds(clientIdList);
        }, threadPoolExecutor);

        CompletableFuture<List<AiClientModelVO>> aiClientModelListFuture = CompletableFuture.supplyAsync(()
            log.info("查询配置数据(ai_client_model) {}", clientIdList);
            return repository.AiClientModelVOByClientIds(clientIdList);
        }, threadPoolExecutor);

        CompletableFuture<List<AiClientToolMcpVO>> aiClientToolMcpListFuture = CompletableFuture.supplyAsyn
            log.info("查询配置数据(ai_client_tool_mcp) {}", clientIdList);
            return repository.AiClientToolMcpVOByClientIds(clientIdList);
        }, threadPoolExecutor);

        CompletableFuture<List<AiClientSystemPromptVO>> aiClientSystemPromptListFuture = CompletableFuture.s
            log.info("查询配置数据(ai_client_system_prompt) {}", clientIdList);
            return repository.AiClientSystemPromptVOByClientIds(clientIdList);
        }, threadPoolExecutor);

        CompletableFuture<List<AiClientAdvisorVO>> aiClientAdvisorListFuture = CompletableFuture.supplyAsyn
            log.info("查询配置数据(ai_client_advisor) {}", clientIdList);
            return repository.AiClientAdvisorVOByClientIds(clientIdList);
        }, threadPoolExecutor);

        CompletableFuture<List<AiClientVO>> aiClientListFuture = CompletableFuture.supplyAsync(() -> {
            log.info("查询配置数据(ai_client) {}", clientIdList);
            return repository.AiClientVOByClientIds(clientIdList);
        }, threadPoolExecutor);

    }
```

```
}
```

如前面章节学习，一个 client 加载，要顺序的加载所有的资源。所以这部分的数据加载也是最全面的。

包括；ai_client_api、ai_client_model、ai_client_tool_mcp、ai_client_system_prompt、ai_client_advisor、ai_client。

这部分代码，都是 crud 操作，需要进入到基础设施层，如；AgentRepository.queryAiClientApiVOListByClientIds 查看代码实现。

3.3.3 策略2；Mode 数据

```
@Slf4j
@Service
public class AiClientModelLoadDataStrategy implements ILoadDataStrategy {

    @Resource
    private IAgentRepository repository;

    @Resource
    protected ThreadPoolExecutor threadPoolExecutor;

    @Override
    public void loadData(ArmoryCommandEntity armoryCommandEntity, DefaultArmoryStrategyFactory.DynamicConte
        List<String> modelIdList = armoryCommandEntity.getCommandIdList();

        CompletableFuture<List<AiClientApiVO>> aiClientApiListFuture = CompletableFuture.supplyAsync(() ->
            log.info("查询配置数据(ai_client_api) {}", modelIdList);
            return repository.queryAiClientApiVOListByModelIds(modelIdList);
        }, threadPoolExecutor);

        CompletableFuture<List<AiClientModelVO>> aiClientModelListFuture = CompletableFuture.supplyAsync(()
            log.info("查询配置数据(ai_client_model) {}", modelIdList);
            return repository.AiClientModelVOByModelIds(modelIdList);
        }, threadPoolExecutor);

    }

}
```

Model 数据加载，相对就少一些。

不过，这里有一个 Model 需要的类型，没有加载。这部分是你的作业，我会在后面章节提供。你可以学习到这的时候，补充上这块的代码。

注意：本节先不用运行代码，只需要了解这样的结构实现即可。后面我们会陆续的处理这些流程和完成测试代码。

四、读者作业

简单作业：学习扳手工程中的设计模式，理解本节代码诉求和编写方式。

复杂作业：完成策略2：Mode 数据，中缺少的需要加载的数据，你可以从库表分析，既可以看到都要加载哪些数据。