《Ai Agent》第3-5节：多数据源和Mapper配置

来自： 码农会锁

小傅哥

2025年06月22日 21:25

本章重点： ★★☆☆☆
课程视频： https://t.zsxq.com/eYn4R
代码分支：https://gitcode.net/KnowledgePlanet/ai-agent-station-study/-/tree/3-5-datasource-mappper
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！😄

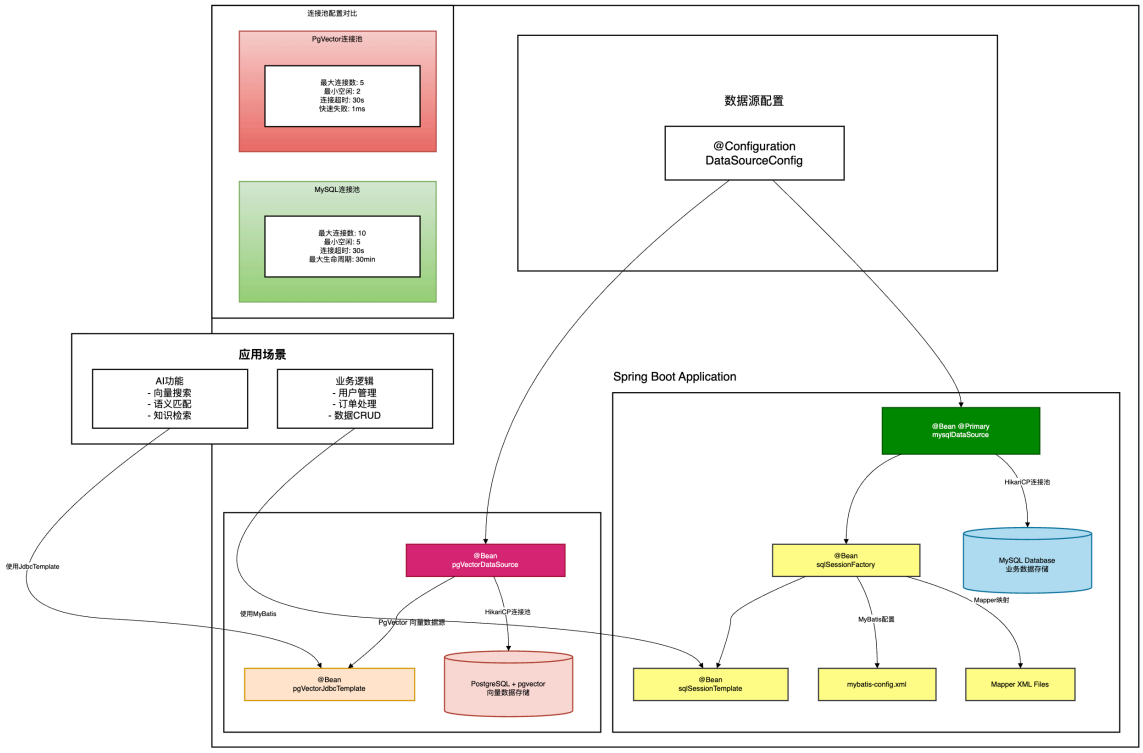
一、本章诉求

为应用程序配置pgvector（向量库）、mysql（业务库）两套数据源，同时基于库表，编写基础设施层 Mapper 操作。

对于数据库表的 Mapper 编写，是一种固定的结构化代码，可以通过 MyBatis 工具生成，也可以使用 AI 编码工具处理。不过对于新人学习来说，更建议在这个阶段，通过手动的方式进行配置编写，这样可以更熟悉库表的设计和字段的理解。尤其是报错后，还可以基于报错排查错误增加编程经验。

二、功能流程

如图，两个数据源的配置和使用；

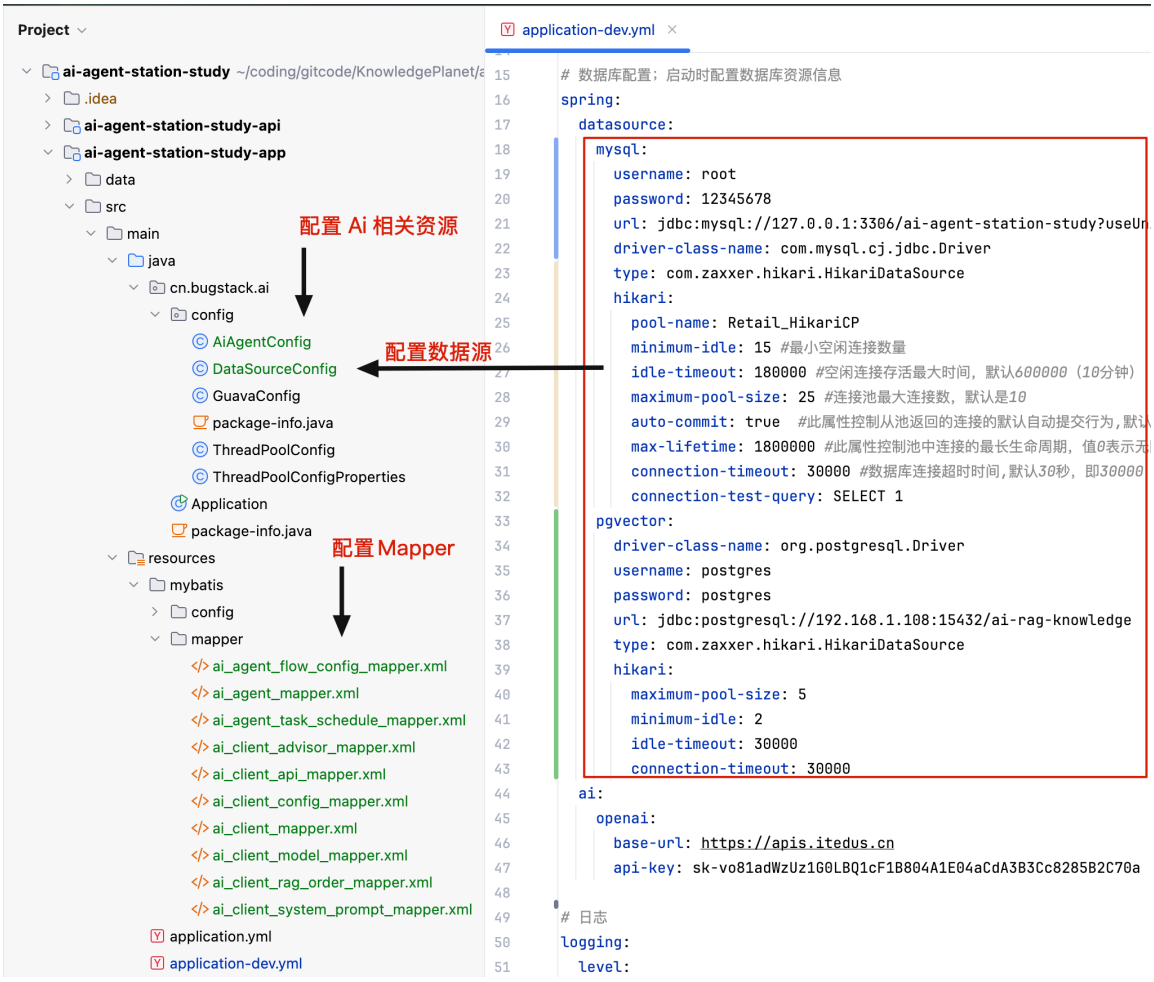


首先，为了让应用程序具备多数据源链接，则需要增加一个扩展的 DataSourceConfig 配置类，来自己实现数据源的加载。这部分会替代原本配置到 yml 文件中，由 Spring 加载数据源的过程。

之后，根据不同类型的数据源，注入到 AI 向量库使用场景和 MyBatis 业务使用场景中。这个过程类似于星球中 DB-Router 路由组件的课程。可以参考：https://bugstack.cn/md/road-map/db-router.html

三、编码实现

1. 工程结构



如图，整个工程分为配置数据源、配置Ai相关资源，以及配置库表 Mapper，同时在工程的 ai-agent-station-study-infrastructure 模块下，还要有 po、dao 的配置。

2. 数据源配置

源码： `ai-agent-station-study-app/src/main/java/cn/bugstack/ai/config/DataSourceConfig.java`
在传统的Spring Boot应用中，我们通常在`application.yml`文件中配置单一数据源，由Spring Boot自动装配。但在AI应用场景下，我们需要同时连接MySQL（业务数据）和PgVector（向量数据），这就需要手动配置多数据源。

2.1 配置类基础结构

```
@Configuration
public class DataSourceConfig {
    // 多数据源配置
}
```

@Configuration： 标记这是一个Spring配置类，Spring容器会扫描并加载其中的Bean定义

这个类替代了传统在 application.yml 中的数据源配置方式，提供更灵活的多数据源管理能力

2.2 MySQL业务数据源配置

```
@Bean("mysqlDataSource")
@Primary
public DataSource mysqlDataSource(
    @Value("${spring.datasource.mysql.driver-class-name}") String driverClassName,
    @Value("${spring.datasource.mysql.url}") String url,
    @Value("${spring.datasource.mysql.username}") String username,
    @Value("${spring.datasource.mysql.password}") String password,
    @Value("${spring.datasource.mysql.hikari.maximum-pool-size:10}") int maximumPoolSize,
    @Value("${spring.datasource.mysql.hikari.minimum-idle:5}") int minimumIdle,
    @Value("${spring.datasource.mysql.hikari.idle-timeout:30000}") long idleTimeout,
    @Value("${spring.datasource.mysql.hikari.connection-timeout:30000}") long
    connectionTimeout,
    @Value("${spring.datasource.mysql.hikari.max-lifetime:1800000}") long maxLifetime) {

    // 创建HikariCP连接池
    HikariDataSource dataSource = new HikariDataSource();
    dataSource.setDriverClassName(driverClassName);
    dataSource.setJdbcUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);

    // 连接池参数配置
    dataSource.setMaximumPoolSize(maximumPoolSize);     // 最大连接数：10
    dataSource.setMinimumIdle(minimumIdle);             // 最小空闲连接数：5
    dataSource.setIdleTimeout(idleTimeout);             // 空闲超时时间：30秒
    dataSource.setConnectionTimeout(connectionTimeout); // 连接超时时间：30秒
    dataSource.setMaxLifetime(maxLifetime);             // 连接最大生命周期：30分钟
    dataSource.setPoolName("MainHikariPool");           // 连接池名称

    return dataSource;
}
```

1. 注解说明：

   @Bean("mysqlDataSource")： 将方法返回值注册为Spring Bean，Bean名称为"mysqlDataSource"

   @Primary： 标记为主数据源，当有多个同类型Bean时，Spring会优先选择这个

2. 参数注入机制：

   @Value("${配置项}")： 从配置文件中读取属性值，实现配置外部化

:10 、 :5 等表示默认值，当配置文件中没有对应属性时使用

这种方式比硬编码更灵活，便于不同环境的配置管理

3. HikariCP连接池优势：

HikariCP是目前性能最优的Java连接池，Spring Boot 2.x默认使用

maximumPoolSize(10)： 业务场景下设置较大连接数，支持高并发访问

minimumIdle(5)： 保持一定数量的空闲连接，提升响应速度

idleTimeout(30秒)： 空闲连接超时释放，避免资源浪费

connectionTimeout(30秒)： 获取连接的最大等待时间

maxLifetime(30分钟)： 连接的最大存活时间，防止长连接问题

2.3 MyBatis集成配置

```
@Bean("sqlSessionFactory")
public SqlSessionFactoryBean sqlSessionFactory(@Qualifier("mysqlDataSource") DataSource mysqlDataSource) thr
    SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
    sqlSessionFactoryBean.setDataSource(mysqlDataSource);

    // 设置MyBatis配置文件位置
    PathMatchingResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    sqlSessionFactoryBean.setConfigLocation(resolver.getResource("classpath:/mybatis/config/mybatis-config.x

    // 设置Mapper XML文件位置
    sqlSessionFactoryBean.setMapperLocations(resolver.getResources("classpath:/mybatis/mapper/*.xml"));

    return sqlSessionFactoryBean;
}

@Bean("sqlSessionTemplate")
public SqlSessionTemplate sqlSessionTemplate(@Qualifier("sqlSessionFactory") SqlSessionFactoryBean sqlSessio
    return new SqlSessionTemplate(Objects.requireNonNull(sqlSessionFactory.getObject()));
}
```

依赖注入精确控制：

@Qualifier("mysqlDataSource")： 明确指定注入名为"mysqlDataSource"的Bean

确保MyBatis使用MySQL数据源，而不会误用PgVector数据源

配置文件管理：

SqlSessionFactoryBean： MyBatis与Spring集成的核心工厂类

setConfigLocation： 指定MyBatis主配置文件，包含全局设置、类型别名等

setMapperLocations： 指定Mapper XML文件位置，支持通配符批量加载

SqlSessionTemplate：

MyBatis-Spring提供的线程安全SqlSession实现

自动管理SqlSession的生命周期，无需手动关闭

与Spring事务管理完美集成

2.4 PgVector向量数据源配置

```
@Bean("pgVectorDataSource")
public DataSource pgVectorDataSource(
    @Value("${spring.datasource.pgvector.driver-class-name}") String driverClassName,
    @Value("${spring.datasource.pgvector.url}") String url,
    @Value("${spring.datasource.pgvector.username}") String username,
    @Value("${spring.datasource.pgvector.password}") String password,
    @Value("${spring.datasource.pgvector.hikari.maximum-pool-size:5}") int maximumPoolSize,
```

```
    @Value("${spring.datasource.pgvector.hikari.minimum-idle:2}") int minimumIdle,
    @Value("${spring.datasource.pgvector.hikari.idle-timeout:30000}") long idleTimeout,
    @Value("${spring.datasource.pgvector.hikari.connection-timeout:30000}") long connectionTimeout) {

    HikariDataSource dataSource = new HikariDataSource();
    dataSource.setDriverClassName(driverClassName);
    dataSource.setJdbcUrl(url);
    dataSource.setUsername(username);
    dataSource.setPassword(password);

    // 向量库专用连接池配置
    dataSource.setMaximumPoolSize(maximumPoolSize);     // 较小连接数：5
    dataSource.setMinimumIdle(minimumIdle);             // 较少空闲连接：2
    dataSource.setIdleTimeout(idleTimeout);
    dataSource.setConnectionTimeout(connectionTimeout);

    // 向量库特殊配置
    dataSource.setInitializationFailTimeout(1);        // 1ms快速失败
    dataSource.setConnectionTestQuery("SELECT 1");      // 连接测试查询
    dataSource.setAutoCommit(true);                     // 自动提交事务
    dataSource.setPoolName("PgVectorHikariPool");       // 连接池名称

    return dataSource;
}

@Bean("pgVectorJdbcTemplate")
public JdbcTemplate pgVectorJdbcTemplate(@Qualifier("pgVectorDataSource") DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

连接池优化策略：

连接数设置较小（最大5个），因为向量查询通常是计算密集型，不需要大量并发连接

针对AI查询场景的特点进行优化，避免资源浪费

快速失败机制：

setInitializationFailTimeout(1)：设置1毫秒快速失败

避免在向量库不可用时长时间等待，快速发现问题

setConnectionTestQuery("SELECT 1")：简单的连接健康检查

JdbcTemplate选择：

使用JdbcTemplate而不是MyBatis，更适合向量操作的简单SQL

向量查询通常是直接的SQL操作，不需要复杂的ORM映射

性能更优，减少不必要的对象映射开销

2.5 向量库配置

```
@Configuration
public class AiAgentConfig {

    /**
     * -- 删除旧的表（如果存在）
     * DROP TABLE IF EXISTS public.vector_store_openai;
     * <p>
     * -- 创建新的表，使用UUID作为主键
     * CREATE TABLE public.vector_store_openai (
     * id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
     * content TEXT NOT NULL,
     * metadata JSONB,
     * embedding VECTOR(1536)
```

```
     * );
     * <p>
     * SELECT * FROM vector_store_openai
     */
    @Bean("vectorStore")
    public PgVectorStore pgVectorStore(@Value("${spring.ai.openai.base-url}") String baseUrl,
                                       @Value("${spring.ai.openai.api-key}") String apiKey,
                                       @Qualifier("pgVectorJdbcTemplate") JdbcTemplate jdbcTemplate) {

        OpenAiApi openAiApi = OpenAiApi.builder()
                .baseUrl(baseUrl)
                .apiKey(apiKey)
                .build();

        OpenAiEmbeddingModel embeddingModel = new OpenAiEmbeddingModel(openAiApi);
        return PgVectorStore.builder(jdbcTemplate, embeddingModel)
                .vectorTableName("vector_store_openai")
                .build();
    }

    @Bean
    public TokenTextSplitter tokenTextSplitter() {
        return new TokenTextSplitter();
    }

}
```

在 AiAgentConfig 文件中，配置了 pgVectorStore，因为这里要指定库表名称。这个表名 vector_store_openai 也就是工程 docs/dev-ops/pgvector/sql/init.sql 里初始化创建的表。如果没有这个表，那么你要通过语句，在向量库手动创建。

Navicate for PostgreSQL，连接向量库蛮好用的，推荐使用。

2.6 多数据源设计优势

1. 数据隔离性： 业务数据和向量数据完全分离，互不影响，提升系统稳定性
2. 性能优化： 针对不同数据类型和访问模式优化连接池参数
3. 技术栈适配： MySQL使用MyBatis ORM，PgVector使用JdbcTemplate，各取所长
4. 扩展性强： 可以轻松添加更多类型的数据源，如Redis、MongoDB等
5. 配置灵活： 支持外部配置文件，便于不同环境的参数调整
6. 故障隔离： 一个数据源的问题不会影响另一个数据源的正常使用

这种多数据源架构在互联网企业中非常常用，一个应用可以连接多套库，多套Redis，以便于分摊数据应用压力。知道了这样设计，以后做其他类似的场景也可以使用。甚至还可以把这样的配置抽取为独立的技术组件使用。

3. Mapper 配置

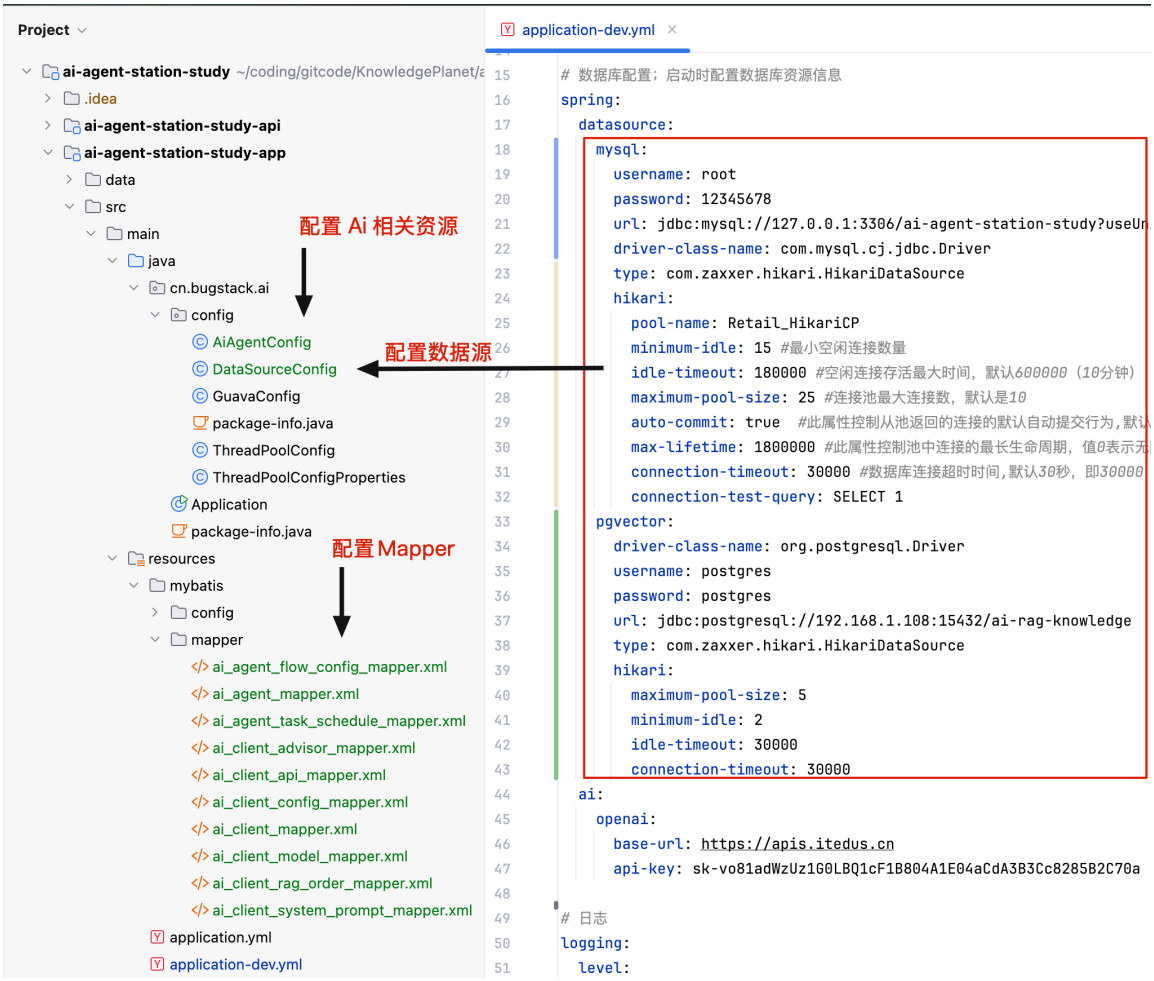
一个数据库表为了可以让程序使用，主要需要 PO 对象，DAO 文件，以及 Mapper 配置数据库语句来映射 PO 和 DAO。如果很感兴趣于 MyBatis 的工作原理，可以在星球学习《手写MyBatis：渐进式源码时间》，可以让你深入的补充这块 ORM 实现技术。
这部分数据库表的配置操作都是很重复的处理，这里只展示其中一个库表，其他的可以参考课程代码。如；ai_agent 表。

3.1 数据库表

```
CREATE TABLE `ai_client_tool_mcp` (
  `id` bigint NOT NULL AUTO_INCREMENT COMMENT '主键ID',
  `mcp_id` varchar(64) NOT NULL COMMENT 'MCP名称',
  `mcp_name` varchar(50) NOT NULL COMMENT 'MCP名称',
  `transport_type` varchar(20) NOT NULL COMMENT '传输类型(sse/stdio)',
  `transport_config` varchar(1024) DEFAULT NULL COMMENT '传输配置(sse/stdio)',
  `request_timeout` int DEFAULT '180' COMMENT '请求超时时间(分钟)',
  `status` tinyint(1) DEFAULT '1' COMMENT '状态(0:禁用,1:启用)',
  `create_time` datetime DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  `update_time` datetime DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
```

```
  PRIMARY KEY (`id`),
  UNIQUE KEY `uk_mcp_id` (`mcp_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci COMMENT='MCP客户端配置表';
```

基于数据库表有非常多的工具可以生成 PO、DAO、Mapper 文件，建议小白学习阶段尽量先手写，增加一个熟悉库表和字段的过程。如果非常熟悉了，以后是纯自己设计的库表，也可以使用AI来完成。举例；



你可以通过这样的话术，让 AI 工具（Cursor、Trae.ai、IDEA AI 插件）来完成相关类的编写。注意把对应文件夹拖拽进去，这样生成的更准确。关于 Trae.ai 工具的使用，这里有文档；https://bugstack.cn/md/road-map/trae.html

3.2 PO 对象

```
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class AiClientToolMcp {

    /**
     * 主键ID
     */
    private Long id;

    /**
     * MCP名称
     */
    private String mcpId;

    /**
     * MCP名称
     */
    private String mcpName;

    /**
```

```
     * 传输类型(sse/stdio)
     */
    private String transportType;

    /**
     * 传输配置(sse/stdio)
     */
    private String transportConfig;

    /**
     * 请求超时时间(分钟)
     */
    private Integer requestTimeout;

    /**
     * 状态(0:禁用,1:启用)
     */
    private Integer status;

    /**
     * 创建时间
     */
    private LocalDateTime createTime;

    /**
     * 更新时间
     */
    private LocalDateTime updateTime;

}
```

3.3 DAO 文件

```
@Mapper
public interface IAiClientToolMcpDao {

    /**
     * 插入MCP客户端配置
     * @param aiClientToolMcp MCP客户端配置对象
     * @return 影响行数
     */
    int insert(AiClientToolMcp aiClientToolMcp);

    /**
     * 根据ID更新MCP客户端配置
     * @param aiClientToolMcp MCP客户端配置对象
     * @return 影响行数
     */
    int updateById(AiClientToolMcp aiClientToolMcp);

    /**
     * 根据MCP ID更新MCP客户端配置
     * @param aiClientToolMcp MCP客户端配置对象
     * @return 影响行数
     */
    int updateByMcpId(AiClientToolMcp aiClientToolMcp);

    /**
     * 根据ID删除MCP客户端配置
     * @param id 主键ID
     * @return 影响行数
```

```
     */
    int deleteById(Long id);

    /**
     * 根据MCP ID删除MCP客户端配置
     * @param mcpId MCP ID
     * @return 影响行数
     */
    int deleteByMcpId(String mcpId);

    /**
     * 根据ID查询MCP客户端配置
     * @param id 主键ID
     * @return MCP客户端配置对象
     */
    AiClientToolMcp queryById(Long id);

    /**
     * 根据MCP ID查询MCP客户端配置
     * @param mcpId MCP ID
     * @return MCP客户端配置对象
     */
    AiClientToolMcp queryByMcpId(String mcpId);

    /**
     * 查询启用的MCP客户端配置
     * @return MCP客户端配置列表
     */
    List<AiClientToolMcp> queryEnabledMcps();

    /**
     * 根据传输类型查询MCP客户端配置
     * @param transportType 传输类型
     * @return MCP客户端配置列表
     */
    List<AiClientToolMcp> queryByTransportType(String transportType);

    /**
     * 查询所有MCP客户端配置
     * @return MCP客户端配置列表
     */
    List<AiClientToolMcp> queryAll();

}
```

3.4 Mapper 映射

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.bugstack.ai.infrastructure.dao.IAiClientToolMcpDao">

    <resultMap id="AiClientToolMcpMap" type="cn.bugstack.ai.infrastructure.dao.po.AiClientToolMcp">
        <id column="id" property="id"/>
        <result column="mcp_id" property="mcpId"/>
        <result column="mcp_name" property="mcpName"/>
        <result column="transport_type" property="transportType"/>
        <result column="transport_config" property="transportConfig"/>
        <result column="request_timeout" property="requestTimeout"/>
        <result column="status" property="status"/>
        <result column="create_time" property="createTime"/>
        <result column="update_time" property="updateTime"/>
```

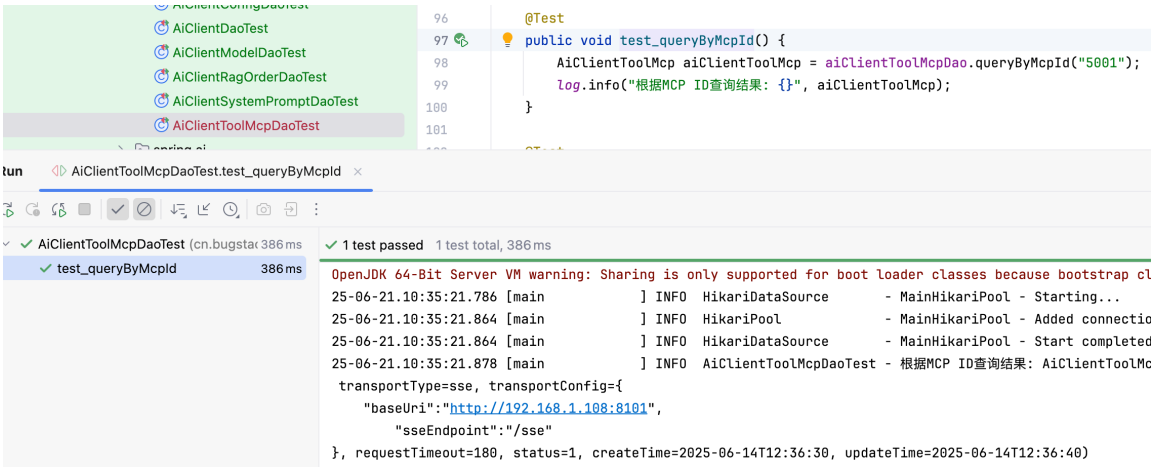
```
</resultMap>

<insert id="insert" parameterType="cn.bugstack.ai.infrastructure.dao.po.AiClientToolMcp" useGeneratedKe
    INSERT INTO ai_client_tool_mcp (
        mcp_id, mcp_name, transport_type, transport_config, request_timeout, status, create_time, updat
    ) VALUES (
        #{mcpId}, #{mcpName}, #{transportType}, #{transportConfig}, #{requestTimeout}, #{status}, #{cre
    )
</insert>

<update id="updateById" parameterType="cn.bugstack.ai.infrastructure.dao.po.AiClientToolMcp">
    UPDATE ai_client_tool_mcp SET
        mcp_id = #{mcpId},
        mcp_name = #{mcpName},
        transport_type = #{transportType},
        transport_config = #{transportConfig},
        request_timeout = #{requestTimeout},
        status = #{status},
        update_time = #{updateTime}
    WHERE id = #{id}
</update>

<update id="updateByMcpId" parameterType="cn.bugstack.ai.infrastructure.dao.po.AiClientToolMcp">
    UPDATE ai_client_tool_mcp SET
        mcp_name = #{mcpName},
        transport_type = #{transportType},
        transport_config = #{transportConfig},
        request_timeout = #{requestTimeout},
        status = #{status},
        update_time = #{updateTime}
    WHERE mcp_id = #{mcpId}
</update>

<delete id="deleteById" parameterType="java.lang.Long">
    DELETE FROM ai_client_tool_mcp WHERE id = #{id}
</delete>

<delete id="deleteByMcpId" parameterType="java.lang.String">
    DELETE FROM ai_client_tool_mcp WHERE mcp_id = #{mcpId}
</delete>

<select id="queryById" parameterType="java.lang.Long" resultMap="AiClientToolMcpMap">
    SELECT id, mcp_id, mcp_name, transport_type, transport_config, request_timeout, status, create_time
    FROM ai_client_tool_mcp
    WHERE id = #{id}
</select>

<select id="queryByMcpId" parameterType="java.lang.String" resultMap="AiClientToolMcpMap">
    SELECT id, mcp_id, mcp_name, transport_type, transport_config, request_timeout, status, create_time
    FROM ai_client_tool_mcp
    WHERE mcp_id = #{mcpId}
</select>

<select id="queryEnabledMcps" resultMap="AiClientToolMcpMap">
    SELECT id, mcp_id, mcp_name, transport_type, transport_config, request_timeout, status, create_time
    FROM ai_client_tool_mcp
    WHERE status = 1
    ORDER BY create_time DESC
</select>

<select id="queryByTransportType" parameterType="java.lang.String" resultMap="AiClientToolMcpMap">
    SELECT id, mcp_id, mcp_name, transport_type, transport_config, request_timeout, status, create_time
    FROM ai_client_tool_mcp
    WHERE transport_type = #{transportType}
```

```
        ORDER BY create_time DESC
    </select>

    <select id="queryAll" resultMap="AiClientToolMcpMap">
        SELECT id, mcp_id, mcp_name, transport_type, transport_config, request_timeout, status, create_time
        FROM ai_client_tool_mcp
        ORDER BY create_time DESC
    </select>

</mapper>
```

四、测试验证

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

    @Resource
    private IAiClientToolMcpDao aiClientToolMcpDao;

    @Test
    public void test_insert() {
        AiClientToolMcp aiClientToolMcp = AiClientToolMcp.builder()
                .mcpId("test_001")
                .mcpName("测试MCP工具")
                .transportType("sse")
                .transportConfig("{\"baseUri\":\"http://localhost:8080\",\"sseEndpoint\":\"/sse\"}")
                .requestTimeout(180)
                .status(1)
                .createTime(LocalDateTime.now())
                .updateTime(LocalDateTime.now())
                .build();

        int result = aiClientToolMcpDao.insert(aiClientToolMcp);
        log.info("插入结果: {}, 生成ID: {}", result, aiClientToolMcp.getId());
    }

    @Test
    public void test_updateById() {
        AiClientToolMcp aiClientToolMcp = AiClientToolMcp.builder()
                .id(1L)
                .mcpId("test_001")
                .mcpName("更新后的测试MCP工具")
                .transportType("stdio")
                .transportConfig("{\"command\":\"npx\",\"args\":[\"-y\",\"test-mcp\"]}")
                .requestTimeout(300)
                .status(1)
                .updateTime(LocalDateTime.now())
                .build();

        int result = aiClientToolMcpDao.updateById(aiClientToolMcp);
        log.info("更新结果: {}", result);
    }

    @Test
    public void test_updateByMcpId() {
        AiClientToolMcp aiClientToolMcp = AiClientToolMcp.builder()
                .mcpId("test_001")
```

```
                .mcpName("根据MCP ID更新的工具")
                .transportType("sse")
                .transportConfig("{\"baseUri\":\"http://localhost:9090\",\"sseEndpoint\":\"/events\"}")
                .requestTimeout(240)
                .status(0)
                .updateTime(LocalDateTime.now())
                .build();

        int result = aiClientToolMcpDao.updateByMcpId(aiClientToolMcp);
        log.info("根据MCP ID更新结果: {}", result);
    }

    @Test
    public void test_deleteById() {
        int result = aiClientToolMcpDao.deleteById(1L);
        log.info("删除结果: {}", result);
    }

    @Test
    public void test_deleteByMcpId() {
        int result = aiClientToolMcpDao.deleteByMcpId("test_001");
        log.info("根据MCP ID删除结果: {}", result);
    }

    @Test
    public void test_queryById() {
        AiClientToolMcp aiClientToolMcp = aiClientToolMcpDao.queryById(1L);
        log.info("根据ID查询结果: {}", aiClientToolMcp);
    }

    @Test
    public void test_queryByMcpId() {
        AiClientToolMcp aiClientToolMcp = aiClientToolMcpDao.queryByMcpId("5001");
        log.info("根据MCP ID查询结果: {}", aiClientToolMcp);
    }

    @Test
    public void test_queryEnabledMcps() {
        List<AiClientToolMcp> mcpList = aiClientToolMcpDao.queryEnabledMcps();
        log.info("查询启用的MCP工具数量: {}", mcpList.size());
        mcpList.forEach(mcp -> log.info("启用的MCP工具: {}", mcp));
    }

    @Test
    public void test_queryByTransportType() {
        List<AiClientToolMcp> mcpList = aiClientToolMcpDao.queryByTransportType("sse");
        log.info("查询SSE传输类型的MCP工具数量: {}", mcpList.size());
        mcpList.forEach(mcp -> log.info("SSE传输类型的MCP工具: {}", mcp));
    }

    @Test
    public void test_queryAll() {
        List<AiClientToolMcp> mcpList = aiClientToolMcpDao.queryAll();
        log.info("查询所有MCP工具数量: {}", mcpList.size());
        mcpList.forEach(mcp -> log.info("MCP工具: {}", mcp));
    }

}
```



这里提供了很多测试方法，可以分别验证DAO工程。

你可以分别验证各个方法，来熟悉库表。所有的这些熟悉过程都是为了后续的编码能更好的理解库表。