SSM 框架整合教程：一、MyBatis——尚硅谷学习笔记 2022 年

一、MyBatis

1、MyBatis 简介

1.1、MyBatis 历史

MyBatis 最初是 Apache 的一个开源项目**iBatis**，2010 年 6 月这个项目由 Apache Software Foundation 迁移到了 Google Code。随着开发团队转投 Google Code 旗下， iBatis3.x 正式更名为 MyBatis。代码于 2013 年 11 月迁移到 Github。

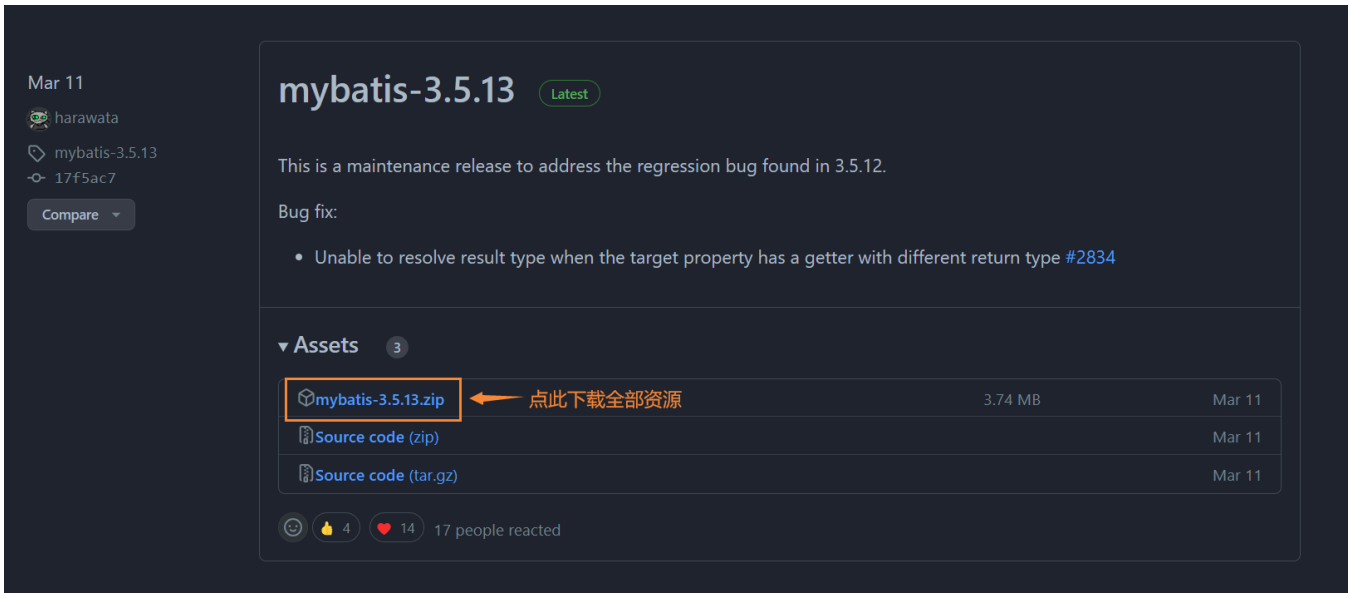
iBatis 一词来源于“internet”和“abatis”的组合，是一个基于 Java 的持久层框架。 iBatis 提供的持久层框架包括 SQL Maps 和 Data Access Objects（DAO）。

1.2、MyBatis 特性

1. MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。
2. MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
3. MyBatis 可以使用简单的 XML 或注解用于配置和原始映射，将接口和 Java 的 POJO（Plain Old Java Objects，普通的 Java 对象）映射成数据库中的记录。
4. MyBatis 是一个 半自动的 ORM（Object Relation Mapping）框架。

1.3、MyBatis 下载

MyBatis 下载地址：https://github.com/mybatis/mybatis-3



1.4、和其它持久化层技术对比

- JDBC
  - SQL 夹杂在 Java 代码中耦合度高，导致硬编码内伤。
  - 维护不易且实际开发需求中 SQL 有变化，频繁修改的情况多见。
  - 代码冗长，开发效率低。
- Hibernate 和 JPA

        - 操作简便，开发效率高。
        - 程序中的长难复杂 SQL 需要绕过框架。
        - 内部自动生产的 SQL，不容易做特殊优化。
        - 基于全映射的全自动框架，大量字段的 POJO 进行部分映射时比较困难。
        - 反射操作太多，导致数据库性能下降。
- MyBatis
        - 轻量级，性能出色。
        - SQL 和 Java 编码分开，功能边界清晰。Java 代码专注业务、SQL 语句专注数据。
        - 开发效率稍逊于 HIbernate，但是完全能够接受。

2、搭建 MyBatis

2.1、开发环境

IDE：idea 2022.3

构建工具：maven 3.8.1

MySQL 版本：MySQL 8

MyBatis 版本：MyBatis 3.5.13

> MySQL 不同版本的注意事项
>
> 1、驱动类 driver-class-name
>
> MySQL 5 版本使用 jdbc5 驱动，驱动类使用：com.mysql.jdbc.Driver
>
> MySQL 8 版本使用 jdbc8 驱动，驱动类使用：com.mysql.cj.jdbc.Driver
>
> 2、连接地址 url
>
> MySQL 5 版本的 url：
>
> jdbc:mysql://localhost:3306/ssm
>
> MySQL 8 版本的 url：

> jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
>
> 否则运行测试用例报告如下错误：
>
> java.sql.SQLException: The server time zone value 'ÖÐ¹ú±ê×¼Ê±¼ä' is unrecognized or represents more

2.2、创建 maven 工程

① 打包方式：jar

② 引入依赖

```
<dependencies>
    <!-- MySQL 驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>

    <!-- Mybatis 核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.13</version>
    </dependency>

    <!-- junit 测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2.3、创建 MyBatis 的核心配置文件

> 习惯上命名为 mybatis-config.xml，这个文件名仅仅只是建议，并非强制要求。将来整合 Spring 之后，这个配置文件可以省略，所以大家操作时可以直接复制、粘贴。

> 核心配置文件主要用于配置连接数据库的环境以及 MyBatis 的全局配置信息。
>
> 核心配置文件存放的位置是 src/main/resources 目录下。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "https://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 配置连接数据库的环境 -->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC"/>
                <property name="username" value="MYXH"/>
                <property name="password" value="520.ILY!"/>
            </dataSource>
        </environment>
    </environments>

    <!--引入 MyBatis 的映射文件 -->
    <mappers>
        <package name="com.myxh.mybatis.mapper"/>
    </mappers>
</configuration>
```

2.4、创建 mapper 接口

> MyBatis 中的 mapper 接口相当于以前的 DAO。但是区别在于，mapper 仅仅是接口，我们不需要提供实现类。

```java
package com.myxh.mybatis.mapper;

import com.myxh.mybatis.pojo.User;

/**
 * @author MYXH
 * @date 2023/8/11
 */
public interface UserMapper
{
    /**
     * 添加用户信息
     *
     * @return 影响的行数
     */
    int insertUser();
}
```

2.5、创建 MyBatis 的映射文件

相关概念：**ORM**（**O**bject **R**elationship **M**apping）对象关系映射。

- 对象：Java 的实体类对象。
- 关系：关系型数据库。
- 映射：二者之间的对应关系。

Java 概念	数据库概念
类	表
属性	字段/列
对象	记录/行

> 1、映射文件的命名规则：
>
> 表所对应的实体类的类名+Mapper.xml
>
> 例如：表 t_user，映射的实体类为 User，所对应的映射文件为 UserMapper.xml
>
> 因此一个映射文件对应一个实体类，对应一张表的操作。

> MyBatis 映射文件用于编写 SQL，访问以及操作表中的数据。
>
> MyBatis 映射文件存放的位置是 src/main/resources/mappers 目录下。
>
> 2、 MyBatis 中可以面向接口操作数据，要保证两个一致：
>
> ① mapper 接口的全类名和映射文件的命名空间（namespace）保持一致。
>
> ② mapper 接口中方法的方法名和映射文件中编写 SQL 的标签的 id 属性保持一致。

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "https://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.myxh.mybatis.mapper.UserMapper">
    <!--
        mapper 接口和映射文件要保证两个一致:
        1、mapper 接口的全类名和映射文件的 namespace 保持一致
        2、mapper 接口中的方法的方法名要和映射文件中的 sql 的 id 保持一致
     -->

    <!-- int insertUser(); -->
    <insert id="insertUser">
        insert into t_user (id, username, password, age, gender, email)
        values (null, 'myxh', '520.ILY!', 21, '男', '1735350920@qq.com'),
               (null, 'root', '000000', 21, '男', 'root@qq.com'),
               (null, 'admin', '123456', 21, '男', 'admin@qq.com');
    </insert>
</mapper>
```

2.6、通过 junit 测试功能

```java
package com.myxh.mybatis.test;

import com.myxh.mybatis.mapper.UserMapper;
import com.myxh.mybatis.pojo.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.Test;

import java.io.IOException;
import java.io.InputStream;

/**
 * @author MYXH
 * @date 2023/8/11
 */
public class MyBatisTest
{
    @Test
    public void testInsert() throws IOException
    {
        // 获取核心配置文件的输入流
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");

        // 获取 sqlSessionFactoryBuilder 对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();

        // 获取 sqlSessionFactory 对象
        SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);

        // 获取 sql 的会话对象 sqlSession(不会自动提交事务)，是 MyBatis 提供的操作数据库的对象
        // SqlSession sqlSession = sqlSessionFactory.openSession();

        // 获取 sql 的会话对象 sqlSession(会自动提交事务)，是 MyBatis 提供的操作数据库的对象
        SqlSession sqlSession = sqlSessionFactory.openSession(true);

        // 获取 userMapper 的代理实现类对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);

        //调用 mapper 接口中的方法，实现添加用户信息的功能
        int result = mapper.insertUser();

        //提供 sql 以及的唯一标识找到 sqL 并执行，唯一标识是 namespace.sqlId
```

```
        // int result = sqlSession.insert("com.myxh.mybatis.mapper.UserMapper.insertUser");

        System.out.println("结果: " + result);

        // 提交事务
        // sqlSession.commit();

        // 关闭 sqlSession
        sqlSession.close();
    }
}
```

> - SqlSession：代表 Java 程序和**数据库**之间的**会话**。（HttpSession 是 Java 程序和浏览器之间的会话）
> - SqlSessionFactory：是“生产”SqlSession 的“工厂”。
> - 工厂模式：如果创建某一个对象，使用的过程基本固定，那么我们就可以把创建这个对象的相关代码封装到一个“工厂类”中，以后都使用这个工厂类来“生产”我们需要的对象。

2.7、加入 log4j 日志功能

2.7.1 ① 加入依赖

```
<!-- log4j 日志 -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

2.7.2 ② 加入 log4j 的配置文件

> log4j 的配置文件名为 log4j.xml，存放的位置是 src/main/resources 目录下。

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <param name="Encoding" value="UTF-8"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p %d{MM-dd HH:mm:ss,SSS} %m (%F:%L) \n"/>
        </layout>
    </appender>

    <logger name="java.sql">
        <level value="debug"/>
    </logger>

    <logger name="org.apache.ibatis">
        <level value="info"/>
    </logger>

    <root>
        <level value="debug"/>
        <appender-ref ref="STDOUT"/>
    </root>
</log4j:configuration>
```

> **日志的级别**
>
> FATAL(致命) > ERROR(错误) > WARN(警告) > INFO(信息) > DEBUG(调试)
>
> 从左到右打印的内容越来越详细。

3、核心配置文件详解

> 核心配置文件中的标签必须按照固定的顺序：
>
> properties?,settings?,typeAliases?,typeHandlers?,objectFactory?,objectWrapperFactory?,reflectorFactory?,plugins?,environments?,databaseIdProvider?,mappers?

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "https://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--
        MyBatis 核心配置文件中的标签必须要按照指定的顺序配置:
        properties?,settings?,typeAliases?,typeHandlers?,
        objectFactory?,objectWrapperFactory?,reflectorFactory?,plugins?,
        environments?,databaseIdProvider?,mappers?
     -->

    <!-- 引入 properties 文件，此后就可以在当前文件中使用 ${key} 的方式访问 value -->
    <properties resource="jdbc.properties"/>

    <!--
        typeAlias: 设置类型别名，即为某个具体的类型设置一个别名
        在 MyBatis 的范围中，就可以使用别名表示一个具体的类型
     -->
    <typeAliases>
        <!--
            typeAlias: 设置某个类型的别名
            属性:
            type: 设置需要设置别名的类型
            alias: 设置某个类型的别名
         -->
        <!-- <typeAlias type="com.myxh.mybatis.pojo.User" alias="User"></typeAlias> -->
        <!-- 若不设置 alias，当前的类型拥有默认的别名，即类名且不区分大小写 -->
        <!-- <typeAlias type="com.myxh.mybatis.pojo.User"></typeAlias> -->

        <!-- 通过包设置类型别名，指定包下所有的类型将全部拥有默认的别名，即类名且不区分大小 -->
        <package name="com.myxh.mybatis.pojo"/>
    </typeAliases>

    <!--
        environments: 配置连接数据库的环境
        属性:
        default: 设置默认使用的环境的 id
     -->
    <environments default="development">
        <!--
            environment: 设置一个具体地连接数据库的环境
            属性:
            id: 设置环境的唯一标识，不能重复
```

```
        -->
        <environment id="development">
            <!--
                transactionManager：设置事务管理器
                属性：
                type：设置事务管理的方式
                type="JDBC|MANAGED"
                JDBC：表示使用 JDBC 中原生的事务管理方式
                MANAGED：被管理，例如 Spring
             -->
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <!--
                    datasource：设置数据源
                    属性：
                    type：设置数据源的类型
                    type="POOLED|UNPOOLED|JNDI"
                    POOLED：表示使用数据库连接池
                    UNPOOLED：表示不使用数据库连接池
                    JNDI：表示使用上下文中的数据源
                 -->
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>

        <environment id="test">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>

    <!--引入 MyBatis 的映射文件 -->
    <mappers>
        <!-- <mapper resource="com/myxh/mybatis/mapper/UserMapper.xml"/> -->
        <!--
            以包的方式引入映射文件，但是必须满足两个条件：
```

```
            1、mapper 接口和映射文件所在的包必须一致
            2、mapper 接口的名字和映射文件的名字必须一致
        -->
        <package name="com.myxh.mybatis.mapper"/>
    </mappers>
</configuration>
```

4、MyBatis 的增删改查

4.1、新增

```
<!-- int insertUser(); -->
<insert id="insertUser">
    insert into t_user (id, username, password, age, gender, email)
    values (null, 'myxh', '520.ILY!', 21, '男', '1735350920@qq.com'),
           (null, 'root', '000000', 21, '男', 'root@qq.com'),
           (null, 'admin', '123456', 21, '男', 'admin@qq.com');
</insert>
```

4.2、删除

```
<!-- int deleteUser(); -->
<delete id="deleteUser">
    delete from t_user where id = 3;
</delete>
```

4.3、修改

```
<!-- int updateUser(); -->
<update id="updateUser">
    update t_user set email = 'denglei_myxh@qq.com' where id = 1;
</update>
```

4.4、查询一个实体类对象

```
<!-- User getUserById(Integer id); -->
<!--
        resultType：设置结果类型，即查询的数据要转换为的 Java 类型
        resultMap：自定义映射，处理一对多或多对一的映射关系
    -->
<select id="getUserById" resultType="User">
    select * from t_user where id = 1;
</select>
```

4.5、查询 list 集合

```
<!-- List<User> getAllUser(); -->
<select id="getAllUser" resultType="User">
    select * from t_user;
</select>
```

> 注意：
>
> 查询的标签 select 必须设置属性 resultType 或 resultMap，用于设置实体类和数据库表的映射关系。
>
> resultType：自动映射，用于属性名和表中字段名一致的情况。
>
> resultMap：自定义映射，用于一对多或多对一或字段名和属性名不一致的情况。

5、MyBatis 获取参数值的两种方式

> MyBatis 获取参数值的两种方式：**${}** 和 **#{}**
>
> ${} 的本质就是字符串拼接，#{} 的本质就是占位符赋值。
>
> ${} 使用字符串拼接的方式拼接 sql，若为字符串类型或日期类型的字段进行赋值时，需要手动加单引号；但是 #{} 使用占位符赋值的方式拼接 sql，此时为字符串类型或日期类型的字段进行赋值时，可以自动添加单引号。

5.1、单个字面量类型的参数

> 若 mapper 接口中的方法参数为单个的字面量类型，此时可以使用 ${} 和 #{} 以任意的名称获取参数的值，注意 $ {} 需要手动加单引号。

5.2、多个字面量类型的参数

> 若 mapper 接口中的方法参数为多个时，此时 MyBatis 会自动将这些参数放在一个 map 集合中，以 arg0, arg1 … 为键，以参数为值；以 param1, param2 … 为键，以参数为值；因此只需要通过 ${} 和 #{} 访问 map 集合的键就可以获取相对应的值，注意 ${} 需要手动加单引号。

5.3、map 集合类型的参数

> 若 mapper 接口中的方法需要的参数为多个时，此时可以手动创建 map 集合，将这些数据放在 map 中只需要通过 ${} 和 #{} 访问 map 集合的键就可以获取相对应的值，注意${}需要手动加单引号。

5.4、实体类类型的参数

> 若 mapper 接口中的方法参数为实体类对象时，此时可以使用 ${} 和 #{}，通过访问实体类对象中的属性名获取属性值，注意${}需要手动加单引号。

5.5、使用@Param 标识参数

> 可以通过@Param 注解标识 mapper 接口中的方法参数，此时会将这些参数放在 map 集合中，以@Param 注解的 value 属性值为键，以参数为值；以 param1, param2 … 为键，以参数为值；只需要通过 ${} 和 #{} 访问 map 集合的键就可以获取相对应的值，注意 ${} 需要手动加单引号。

6、MyBatis 的各种查询功能

6.1、查询一个实体类对象

```
/**
 * 根据 id 查询用户信息
 *
 * @param id 用户 id
 * @return 一个用户信息
 */
User getUserById(@Param("id") Integer id);
```

```
<!-- User getUserById(@Param("id") Integer id); -->
<select id="getUserById" resultType="User">
    select * from t_user where id = #{id};
</select>
```

6.2、查询一个 list 集合

```
/**
 * 查询所有的用户信息
 *
 * @return 所有用户信息
 */
List<User> getAllUser();
```

```
<!-- List<User> getAllUser(); -->
<select id="getAllUser" resultType="User">
    select * from t_user;
</select>
```

> 当查询的数据为多条时，不能使用实体类作为返回值，否则会抛出异常 TooManyResultsException；但是若查询的数据只有一条，可以使用实体类或集合作为返回值。

6.3、查询单个数据

```java
/**
 * 查询用户的总数量
 *
 * @return 用户的总数量
 */
Integer getUserCount();
```

```xml
<!-- Integer getUserCount(); -->
<!--
MyBatis 中为 Java 中常用的类型设置了类型别名：
Integer: Integer, int
int: _int, _integer
Map: map
String: string
    -->
<select id="getUserCount" resultType="Integer">
    select count(*) from t_user;
</select>
```

6.4、查询一条数据为 map 集合

```java
/**
 * 根据 id 查询用户信息为 userMap 集合
 *
 * @param id 用户 id
 * @return 一个用户信息为 userMap 集合
 */
Map<String, Object> getUserByIdToUserMap(@Param("id") Integer id);
```

```xml
<!-- Map<String, Object> getUserByIdToUserMap(@Param("id") Integer id); -->
<select id="getUserByIdToUserMap" resultType="Map">
    select * from t_user where id = #{id};
</select>
```

6.5、查询多条数据为 map 集合

6.5.1 ① 方式一

```
/**
 * 查询所有的用户信息为 userMap 集合
 *
 * @return 所有用户信息为 userMap 集合
 */
List<Map<String, Object>> getAllUserToUserMap();
```

```
<!-- Map<String, Object> getAllUserToUserMap(); -->
<select id="getAllUserToUserMap" resultType="Map">
    select * from t_user;
</select>
```

6.5.2 ② 方式二

```
/**
 * 查询所有的用户信息为 userMap 集合
 * 若查询的数据有多条时，并且要将每条数据转换为 map 集合，
 * 此时有两种解决方案：
 * 1、将 mapper 接口方法的返回值设置为泛型是 Map 的 List 集合
 * List<Map<string, object>> getAllUserToUserMap();
 * 结果：[{password=520.ILY!, gender=男, id=1, age=21, email=1735350920@qq.com, username=MYXH}, {pa
 * 2、可以将每条数据转换的 map 集合放在一个大的 map 中，但是必须要通过 @MapKey 注解，
 * 将查询的某个字段的值作为大的 map 的键
 * @MapKey("id") Map<String, Object> getAllUserToUserMap();
 * 结果：
 * {
 *      1={password=520.ILY!, gender=男, id=1, age=21, email=1735350920@qq.com, username=MYXH},
 *      2={password=000000, gender=男, id=2, age=21, email=root@qq.com, username=root},
 *      3={password=123456, gender=男, id=3, age=21, email=admin@qq.com, username=admin}
 * }
 *
 * @return 所有用户信息为 userMap 集合
 */
@MapKey("id")
Map<String, Object> getAllUserToUserMap();
```

```xml
<!-- Map<String, Object> getAllUserToUserMap(); -->
<select id="getAllUserToUserMap" resultType="Map">
    select * from t_user;
</select>
```

7、特殊 SQL 的执行

7.1、模糊查询

```java
/**
 * 通过用户名模糊查询用户信息
 *
 * @param vagueUsername 模糊用户名
 * @return 一些用户信息
 */
List<User> getUserByLikeUsername(@Param("vagueUsername") String vagueUsername);
```

```xml
<!-- List<User> getUserByLikeUsername(@Param("username") String vagueUsername); -->
<select id="getUserByLikeUsername" resultType="User">
    <!-- select * from t_user where username like '%${vagueUsername}%'; -->
    <!-- select * from t_user where username like concat('%', #{vagueUsername}, '%'); -->
    select * from t_user where username like "%"#{vagueUsername}"%";
</select>
```

7.2、批量删除

```java
/**
 * 批量删除用户信息
 *
 * @param ids 一些用户 id
 * @return 影响的行数
 */
int deleteMoreUserByIds(@Param("ids") String ids);
```

```
<!-- int deleteMoreUserByIds(@Param("ids") String ids); -->
<delete id="deleteMoreUserByIds">
    <!-- ids: 4, 5 -->
    delete from t_user where id in(${ids});
</delete>
```

7.3、动态设置表名

```
/**
 * 动态设置表名，查询用户信息
 *
 * @param tableName 动态表名
 * @return 一些用户信息
 */
List<User> getUserByTableNameToList(@Param("tableName") String tableName);
```

```
<!-- List<User> getUserByTableNameToList(@Param("tableName") String tableName); -->
<select id="getUserByTableNameToList" resultType="User">
    select * from ${tableName};
</select>
```

7.4、添加功能获取自增的主键

> 场景模拟：
>
> t_clazz(clazz_id,clazz_name)
>
> t_student(student_id,student_name,clazz_id)
>
> 1、添加班级信息
>
> 2、获取新添加的班级的 id
>
> 3、为班级分配学生，即将某学的班级 id 修改为新添加的班级的 id

```java
/**
 * 添加用户信息并获取自增的主键
 *
 * @param user 用户信息
 * @return 影响的行数
 */
int insertUser(User user);
```

```xml
<!-- int insertUser(User user); -->
<!--
useGeneratedKeys：表示当前添加功能使用自增的主键
keyProperty：将添加的数据的自增主键为实体类类型的参数的属性值
    -->
<insert id="insertUser" useGeneratedKeys="true" keyProperty="id">
    insert into t_user (id, username, password, age, gender, email)
    values (null, #{username}, #{password}, #{age}, #{gender}, #{email});
</insert>
```

8、自定义映射 resultMap

8.1、resultMap 处理字段和属性的映射关系

> 若字段名和实体类中的属性名不一致，则可以通过 resultMap 设置自定义映射。

```
<!--
resultMap：设置自定义的映射关系
id：唯一标识
type：处理映射关系的实体类的类型
常用的标签：
id：处理主键和实体类中属性的映射关系
result：处理普通字段和实体类中属性的映射关系
association：处理多对一的映射关系（处理实体类类型的属性）
collection：处理一对多的映射关系（处理集合类型的属性）
column：设置映射关系中的字段名，必须是 SQL 查询出的某个字段
property：设置映射关系中的属性的属性名，必须是处理的实体类类型中的属性名
    -->
<resultMap id="EmployeeResultMap" type="Employee">
    <id column="employee_id" property="employeeId"/>
    <result column="employee_name" property="employeeName"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="email" property="email"/>
</resultMap>

<!-- Employee getEmployeeByEmployeeId(@Param("employeeId") Integer employeeId); -->
<!--
<select id="getEmployeeByEmployeeId" resultType="Employee">
    select employee_id as employeeId, employee_name as employeeName, age, gender, email from t_emp
    where employee_id = #{employeeId};
</select>
    -->
<!--
<select id="getEmployeeByEmployeeId" resultType="Employee">
    select * from t_employee where employee_id = #{employeeId};
</select>
    -->
<select id="getEmployeeByEmployeeId" resultMap="EmployeeResultMap">
    select * from t_employee where employee_id = #{employeeId};
</select>
```

> 若字段名和实体类中的属性名不一致，但是字段名符合数据库的规则（使用_），实体类中的属性名符合 Java 的规则（使用驼峰），此时也可通过以下两种方式处理字段名和实体类中的属性的映射关系。
>
> 1、可以通过为字段起别名的方式，保证和实体类中的属性名保持一致。
>
> 2、可以在 MyBatis 的核心配置文件中设置一个全局配置信息

> mapUnderscoreToCamelCase，可以在查询表中数据时，自动将_类型的字段名转换为驼峰。
>
> 例如：字段名 employee_name，设置了 mapUnderscoreToCamelCase，此时字段名就会转换为 employeeName。

8.2、多对一映射处理

> 场景模拟：
>
> 查询员工信息以及员工所对应的部门信息。

8.2.1、级联方式处理映射关系

```
<resultMap id="EmployeeAndDepartmentResultMap" type="Employee">
    <id column="employee_id" property="employeeId"></id>
    <result column="employee_name" property="employeeName"></result>
    <result column="age" property="age"></result>
    <result column="gender" property="gender"></result>
    <result column="employee_email" property="email"></result>
    <result column="department_id" property="department.departmentId"></result>
    <result column="department_name" property="department.departmentName"></result>
    <result column="department_email" property="department.email"></result>
</resultMap>

<!-- Employee getEmployeeAndDepartmentByEmployeeId(@Param("employeeId") Integer employeeId); -->
<select id="getEmployeeAndDepartmentByEmployeeId" resultMap="EmployeeAndDepartmentResultMap">
    select
    t_e.employee_id,
    t_e.employee_name,
    t_e.age,
    t_e.gender,
    t_e.email as employee_email,
    t_d.department_id,
    t_d.department_name,
    t_d.email as department_email
    from t_employee as t_e
    left join t_department as t_d
    on t_e.department_id = t_d.department_id
    where t_e.employee_id = #{employeeId};
</select>
```

8.2.2、使用 association 处理映射关系

```
<resultMap id="EmployeeAndDepartmentResultMap" type="Employee">
    <id column="employee_id" property="employeeId"/>
    <result column="employee_name" property="employeeName"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="employee_email" property="email"/>
    <!--
    association：处理多对一的映射关系（处理实体类类型的属性）
    property：设置需要处理映射关系的属性的属性名
    javaType：设置要处理的属性的类型
        -->
    <association property="department" javaType="Department">
        <id column="department_id" property="departmentId"/>
        <result column="department_name" property="departmentName"/>
        <result column="department_email" property="email"/>
    </association>
</resultMap>

<!-- Employee getEmployeeAndDepartmentByEmployeeId(@Param("employeeId") Integer employeeId); -->
<select id="getEmployeeAndDepartmentByEmployeeId" resultMap="EmployeeAndDepartmentResultMap">
    select
    t_e.employee_id,
    t_e.employee_name,
    t_e.age,
    t_e.gender,
    t_e.email as employee_email,
    t_d.department_id,
    t_d.department_name,
    t_d.email as department_email
    from t_employee as t_e
    left join t_department as t_d
    on t_e.department_id = t_d.department_id
    where t_e.employee_id = #{employeeId};
</select>
```

8.2.3、分步查询

8.2.3.1 ① 查询员工信息

```java
/**
 * 通过分步查询查询员工信息以及所对应的部信息的第一步
 *
 * @param employeeId 员工 id
 * @return 一个员工信息
 */
Employee getEmployeeAndDepartmentByStepOne(@Param("employeeId") Integer employeeId);
```

```xml
<resultMap id="EmployeeAndDepartmentByStepResultMap" type="Employee">
    <id column="employee_id" property="employeeId"/>
    <result column="employee_name" property="employeeName"/>
    <result column="age" property="age"/>
    <result column="gender" property="gender"/>
    <result column="email" property="email"/>
    <!--
    property：设置需要处理映射关系的属性的属性名
    fetchType：在开启了延迟加载的环境中，通过该属性设置当前的分步查询是否使用延迟加
    fetchType="eager（立即加载） | lazy（延迟加载）
    select：设置分步查询的 SQL 的唯一标识
    column：将查询出的某个字段作为分步查询的 SQL 的条件
        -->
    <association property="department" fetchType="eager"
                 select="com.myxh.mybatis.mapper.DepartmentMapper.getEmployeeAndDepartmentBySte
                 column="department_id">
    </association>
</resultMap>

<!-- Employee getEmployeeAndDepartmentByStepOne(@Param("employeeId") Integer employeeId); -->
<select id="getEmployeeAndDepartmentByStepOne" resultMap="EmployeeAndDepartmentByStepResultMap">
    select * from t_employee where employee_id = #{employeeId};
</select>
```

8.2.3.2 ② 根据员工所对应的部门 id 查询部门信息

```
/**
 * 通过分步查询查询员工信息以及所对应的部信息的第二步
 *
 * @param departmentId 部门 id
 * @return 一个员工所对应的部门信息
 */
Department getEmployeeAndDepartmentByStepTwo(@Param("departmentId") Integer departmentId);
```

```
<!-- Department getEmployeeAndDepartmentByStepTwo(@Param("departmentId") Integer departmentId); --
<select id="getEmployeeAndDepartmentByStepTwo" resultType="Department">
    select * from t_department where department_id = #{departmentId};
</select>
```

8.3、一对多映射处理

8.3.1、collection

```
/**
 * 根据 id 获取部门信息以及部门中的所有员工信息
 *
 * @param departmentId 部门 id
 * @return 一个部门信息以及部门中的所有员工信息
 */
Department getDepartmentAndEmployeeByDepartmentId(@Param("departmentId") Integer departmentId);
```

```xml
<resultMap id="DepartmentAndEmployeeResultMap" type="Department">
    <id column="department_id" property="departmentId"/>
    <result column="department_name" property="departmentName"/>
    <result column="department_email" property="email"/>
    <!--
    collection：处理一对多的映射关系（处理集合类型的属性）
    ofType：设置要处理的集合类型的属性中存储的数据的类型
        -->
    <collection property="employees" ofType="Employee">
        <id column="employee_id" property="employeeId"/>
        <result column="employee_name" property="employeeName"/>
        <result column="age" property="age"/>
        <result column="gender" property="gender"/>
        <result column="employee_email" property="email"/>
    </collection>
</resultMap>

<!-- Department getDepartmentAndEmployeeByDepartmentId(@Param("departmentId") Integer departmentId); -->
<select id="getDepartmentAndEmployeeByDepartmentId" resultMap="DepartmentAndEmployeeResultMap">
    select
    t_d.department_id,
    t_d.department_name,
    t_d.email as department_email,
    t_e.employee_id,
    t_e.employee_name,
    t_e.age,
    t_e.gender,
    t_e.email as employee_email
    from t_department as t_d
    left join t_employee as t_e
    on t_d.department_id = t_e.department_id
    where t_d.department_id = #{departmentId};
</select>
```

8.3.2、分步查询

8.3.2.1 ① 查询部门信息

```java
/**
 * 通过分步查询查询部门信息以及部门中的所有员工信息的第一步
 *
 * @param departmentId 部门 id
 * @return 一个部门信息
 */
Department getDepartmentAndEmployeeByStepOne(@Param("departmentId") Integer departmentId);
```

```xml
<resultMap id="DepartmentAndEmployeeByStepResultMap" type="Department">
    <id column="department_id" property="departmentId"/>
    <result column="department_name" property="departmentName"/>
    <result column="email" property="email"/>
    <collection property="employees" fetchType="eager"
                select="com.myxh.mybatis.mapper.EmployeeMapper.getDepartmentAndEmployeeByStepTwo"
                column="department_id">
    </collection>
</resultMap>

<!-- Department getDepartmentAndEmployeeByStepOne(@Param("departmentId") Integer departmentId); -->
<select id="getDepartmentAndEmployeeByStepOne" resultMap="DepartmentAndEmployeeByStepResultMap">
    select * from t_department where department_id = #{departmentId};
</select>
```

8.3.2.2 ② 根据部门 id 查询部门中的所有员工

```java
/**
 * 通过分步查询查询部门信息以及部门中的所有员工信息的第二步
 *
 * @param departmentId 部门 id
 * @return 一个部门中的所有员工信息
 */
List<Employee> getDepartmentAndEmployeeByStepTwo(@Param("departmentId") Integer departmentId);
```

```xml
<!-- List<Employee> getDepartmentAndEmployeeByStepTwo(@Param("departmentId") Integer departmentId); -->
<select id="getDepartmentAndEmployeeByStepTwo" resultType="Employee">
    select * from t_employee where department_id = #{departmentId};
</select>
```

> 分步查询的优点：可以实现延迟加载。
>
> 但是必须在核心配置文件中设置全局配置信息：
>
> lazyLoadingEnabled：延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。
>
> aggressiveLazyLoading：当开启时，任何方法的调用都会加载该对象的所有属性。否则，每个属性会按需加载。
>
> 此时就可以实现按需加载，获取的数据是什么，就只会执行相应的 sql。此时可通过 association 和 collection 中的 fetchType 属性设置当前的分步查询是否使用延迟加载，fetchType="lazy(延迟加载)|eager(立即加载)"。

9、动态 SQL

> Mybatis 框架的动态 SQL 技术是一种根据特定条件动态拼装 SQL 语句的功能，它存在的意义是为了解决拼接 SQL 语句字符串时的痛点问题。

9.1、if

> if 标签可通过 test 属性的表达式进行判断，若表达式的结果为 true，则标签中的内容会执行；反之标签中的内容不会执行。

```
<!-- List<Employee> getEmployeeByCondition(Employee employee); -->
<select id="getEmployeeByCondition" resultType="Employee">
        select * from t_employee where 1=1
        <if test="employeeName != null and employeeName != ''">
            and employee_name = #{employeeName}
        </if>
        <if test="age != null">
            and age = #{age}
        </if>
        <if test="gender != null and gender != ''">
            and gender = #{gender}
        </if>
        <if test="email != null and email != ''">
            and email = #{email}
        </if>
        ;
    </select>
```

9.2、where

where 和 if 一般结合使用：

1、若 where 标签中的 if 条件都不满足，则 where 标签没有任何功能，即不会添加 where 关键字。

2、若 where 标签中的 if 条件满足，则 where 标签会自动添加 where 关键字，并将条件最前方多余的 and 去掉。

注意：where 标签不能去掉条件最后多余的 and。

```
<!-- List<Employee> getEmployeeByCondition(Employee employee); -->
<select id="getEmployeeByCondition" resultType="Employee">
    select * from t_employee
    <where>
        <if test="employeeName != null and employeeName != ''">
            employee_name = #{employeeName}
        </if>
        <if test="age != null">
            and age = #{age}
        </if>
        <if test="gender != null and gender != ''">
            and gender = #{gender}
        </if>
        <if test="email != null and email != ''">
            and email = #{email}
        </if>
    </where>
    ;
</select>
```

9.3、trim

trim 用于去掉或添加标签中的内容。

常用属性：

prefix：在 trim 标签中的内容的前面添加某些内容。

prefixOverrides：在 trim 标签中的内容的前面去掉某些内容。

> suffix：在 trim 标签中的内容的后面添加某些内容。
>
> suffixOverrides：在 trim 标签中的内容的后面去掉某些内容。

```
<!-- List<Employee> getEmployeeByCondition(Employee employee); -->
<select id="getEmployeeByCondition" resultType="Employee">
    select * from t_employee
    <trim prefix="where" suffixOverrides="and">
        <if test="employeeName != null and employeeName != ''">
            employee_name = #{employeeName} and
        </if>
        <if test="age != null">
            age = #{age} and
        </if>
        <if test="gender != null and gender != ''">
            gender = #{gender} and
        </if>
        <if test="email != null and email != ''">
            email = #{email}
        </if>
    </trim>
    ;
</select>
```

9.4、choose、when、otherwise

> choose、when、 otherwise 相当于 if … else if … else。

```xml
<!-- List<Employee> getEmployeeByChoose(Employee employee); -->
<select id="getEmployeeByChoose" resultType="Employee">
    select <include refid="employeeColumns"/> from t_employee
    <where>
        <choose>
            <when test="employeeName != null and employeeName != ''">
                employee_name = #{employeeName}
            </when>
            <when test="age != null">
                age = #{age}
            </when>
            <when test="gender != null and gender != ''">
                gender = #{gender}
            </when>
            <when test="email != null and email != ''">
                email = #{email}
            </when>
        </choose>
    </where>
    ;
</select>
```

9.5、foreach

```
<!-- int insertMoreEmployee(@Param("employees") List<Employee> employees); -->
<insert id="insertMoreEmployee">
    insert into t_employee (employee_id, employee_name, age, gender, email, department_id)
    values
    <foreach collection="employees" item="employee" separator=",">
        (null, #{employee.employeeName}, #{employee.age}, #{employee.gender}, #{employee.email}, n
    </foreach>
    ;
</insert>

<!-- int deleteMoreEmployee(@Param("employeeIds") Integer[] employeeIds); -->
<delete id="deleteMoreEmployee">
    delete from t_employee where employee_id in
    <foreach collection="employeeIds" item="employeeId" separator="," open="(" close=")">
        #{employeeId}
    </foreach>
    ;
</delete>

<!-- int deleteMoreEmployee(@Param("employeeIds") Integer[] employeeIds); -->
<delete id="deleteMoreEmployee">
    delete from t_employee where
    <foreach collection="employeeIds" item="employeeId" separator="or">
        employee_id = #{employeeId}
    </foreach>
    ;
</delete>
```

9.6、SQL 片段

> sql 片段，可以记录一段公共 sql 片段，在使用的地方通过 include 标签进行引入。

```
<sql id="employeeColumns">
    employee_id, employee_name, age, gender, email, department_id
</sql>
select <include refid="employeeColumns"/> from t_employee;
```

10、MyBatis 的缓存

10.1、MyBatis 的一级缓存

一级缓存是 SqlSession 级别的，通过同一个 SqlSession 查询的数据会被缓存，下次查询相同的数据，就会从缓存中直接获取，不会从数据库重新访问。

使一级缓存失效的四种情况：

1. 不同的 SqlSession 对应不同的一级缓存。
2. 同一个 SqlSession 但是查询条件不同。
3. 同一个 SqlSession 两次查询期间执行了任何一次增删改操作。
4. 同一个 SqlSession 两次查询期间手动清空了缓存。

10.2、MyBatis 的二级缓存

二级缓存是 SqlSessionFactory 级别，通过同一个 SqlSessionFactory 创建的 SqlSession 查询的结果会被缓存；此后若再次执行相同的查询语句，结果就会从缓存中获取。

二级缓存开启的条件：

1、在核心配置文件中，设置全局配置属性 cacheEnabled=“true”，默认为 true，不需要设置。

2、在映射文件中设置标签<cache/>。

3、二级缓存必须在 SqlSession 关闭或提交之后有效。

4、查询的数据所转换的实体类类型必须实现序列化的接口。

使二级缓存失效的情况：

两次查询之间执行了任意的增删改，会使一级和二级缓存同时失效。

10.3、二级缓存的相关配置

在 mapper 配置文件中添加的 cache 标签可以设置一些属性：

① eviction 属性：缓存回收策略，默认的是 LRU。

LRU（Least Recently Used） – 最近最少使用的：移除最长时间不被使用的对象。

FIFO（First in First out） – 先进先出：按对象进入缓存的顺序来移除它们。

SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。

WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。

② flushInterval 属性：刷新间隔，单位毫秒。

默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。

③ size 属性：引用数目，正整数。

代表缓存最多可以存储多少个对象，太大容易导致内存溢出。

④ readOnly 属性：只读， true/false。

true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了 很重要的性能优势。

false：读写缓存；会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是 false。

10.4、MyBatis 缓存查询的顺序

先查询二级缓存，因为二级缓存中可能会有其他程序已经查出来的数据，可以拿来直接使用。

如果二级缓存没有命中，再查询一级缓存。

如果一级缓存也没有命中，则查询数据库。

SqlSession 关闭之后，一级缓存中的数据会写入二级缓存。

10.5、整合第三方缓存 EHCache

10.5.1、添加依赖

```
<!-- Mybatis EHCache 整合包 -->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.3</version>
</dependency>

<!-- slf4j 日志门面的一个具体实现 -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.4.7</version>
</dependency>
```

10.5.2、各 jar 包功能

jar 包名称	作用
mybatis-ehcache	Mybatis 和 EHCache 的整合包
ehcache	EHCache 核心包
slf4j-api	SLF4J 日志门面包
logback-classic	支持 SLF4J 门面接口的一个具体实现

10.5.3、创建 EHCache 的配置文件 ehcache.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
    <!-- 磁盘保存路径 -->
    <diskStore path=".\ehcache"/>
    <defaultCache
            maxElementsInMemory="1000"
            maxElementsOnDisk="10000000"
            eternal="false"
            overflowToDisk="true"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            diskExpiryThreadIntervalSeconds="120"
            memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>
```

10.5.4、设置二级缓存的类型

```
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

10.5.5、加入 logback 日志

> 存在 SLF4J 时，作为简易日志的 log4j 将失效，此时我们需要借助 SLF4J 的具体实现 logback 来打印日志。 创建 logback 的配置文件 logback.xml。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <!-- 指定日志输出的位置 -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- 日志输出的格式 -->
            <!-- 按照顺序分别是：时间、日志级别、线程名称、打印日志的类、日志主体内容、换行 -->
            <pattern>[%d{HH:mm:ss.SSS}] [%-5level] [%thread] [%logger]
                [%msg]%n
            </pattern>
        </encoder>
    </appender>

    <!-- 设置全局日志级别，日志级别按顺序分别是： DEBUG、INFO、WARN、ERROR -->
    <!-- 指定任何一个日志级别都只打印当前级别和后面级别的日志 -->
    <root level="DEBUG">
        <!-- 指定打印日志的 appender，这里通过 "STDOUT" 引用了前面配置的 appender -->
        <appender-ref ref="STDOUT"/>
    </root>

    <!-- 根据特殊需求指定局部日志级别 -->
    <logger name="com.myxh.mybatis.mapper" level="DEBUG"/>
</configuration>
```

10.5.6、EHCache 配置文件说明

属性名	是否必须	作用
maxElementsInMemory	是	在内存中缓存的 element 的最大数目。
maxElementsOnDisk	是	在磁盘上缓存的 element 的最大数目，若是 0 表示无穷大。
eternal	是	设定缓存的 elements 是否永远不过期。如果为 true，则缓存的数据始终有效，如果为 false 那么还要根据 timeToIdleSeconds、timeToLiveSeconds 判断。
overflowToDisk	是	设定当内存缓存溢出的时候是否将过期的 element 缓存到磁盘上。

属性名	是否必须	作用
timeToIdleSeconds	否	当缓存在 EhCache 中的数据前后两次访问的时间超过 timeToIdleSeconds 的属性取值时，这些数据便会删除，默认值是 0,也就是可闲置时间无穷大。
timeToLiveSeconds	否	缓存 element 的有效生命期，默认是 0.,也就是 element 存活时间无穷大。
diskSpoolBufferSizeMB	否	DiskStore(磁盘缓存)的缓存区大小。默认是 30MB。每个 Cache 都应该有自己的一个缓冲区。
diskPersistent	否	在 VM 重启的时候是否启用磁盘保存 EhCache 中的数据，默认是 false。
diskExpiryThreadIntervalSeconds	否	磁盘缓存的清理线程运行间隔，默认是 120 秒。每个 120s， 相应的线程会进行一次 EhCache 中数据的清理工作。
memoryStoreEvictionPolicy	否	当内存缓存达到最大，有新的 element 加入的时候， 移除缓存中 element 的策略。 默认是 LRU （最近最少使用）， 可选的有 LFU （最不常使用）和 FIFO （先进先出）。

11、MyBatis 的逆向工程

> 正向工程：先创建 Java 实体类，由框架负责根据实体类生成数据库表。 Hibernate 是支持正向工程的。
>
> 逆向工程：先创建数据库表，由框架负责根据数据库表，反向生成如下资源：
>
> - Java 实体类
> - Mapper 接口
> - Mapper 映射文件

11.1、创建逆向工程的步骤

11.1.1 ① 添加依赖和插件

```xml
<!-- 依赖 MyBatis 核心包 -->
<dependencies>
    <!-- MySQL 驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>

    <!-- Mybatis 核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.13</version>
    </dependency>

    <!-- junit 测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>

    <!-- log4j 日志 -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
</dependencies>

<!-- 控制 Maven 在构建过程中相关配置 -->
<build>
    <!-- 构建过程中用到的插件 -->
    <plugins>
        <!-- 具体插件，逆向工程的操作是以构建过程中插件形式出现的 -->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.0</version>
            <!-- 插件的依赖 -->
            <dependencies>
                <!-- 逆向工程的核心依赖 -->
```

```
                <dependency>
                    <groupId>org.mybatis.generator</groupId>
                    <artifactId>mybatis-generator-core</artifactId>
                    <version>1.3.2</version>
                </dependency>

                <!-- MySQL 驱动 -->
                <dependency>
                    <groupId>mysql</groupId>
                    <artifactId>mysql-connector-java</artifactId>
                    <version>8.0.33</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

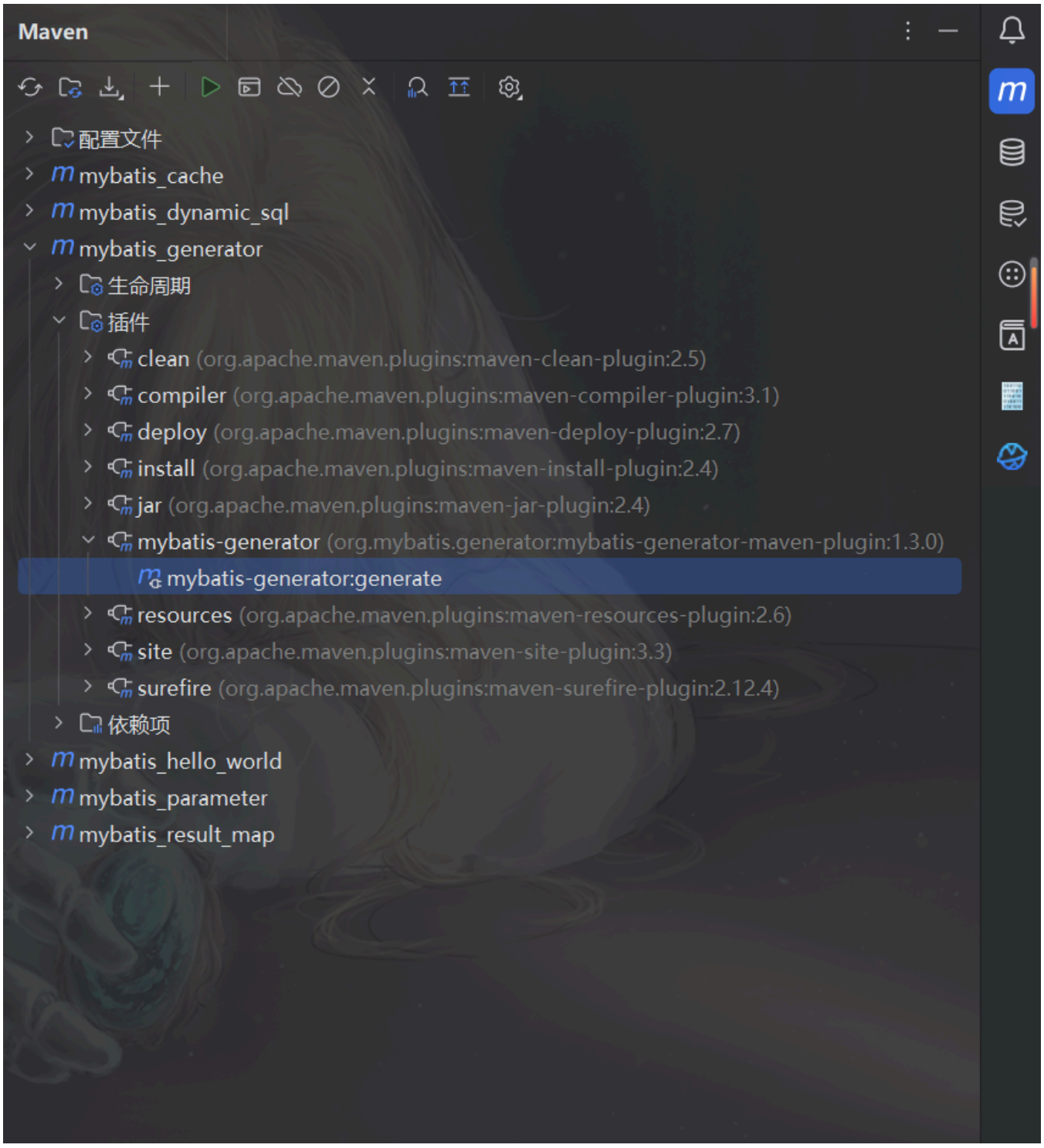
11.1.2 ② 创建 MyBatis 的核心配置文件

11.1.3 ③ 创建逆向工程的配置文件

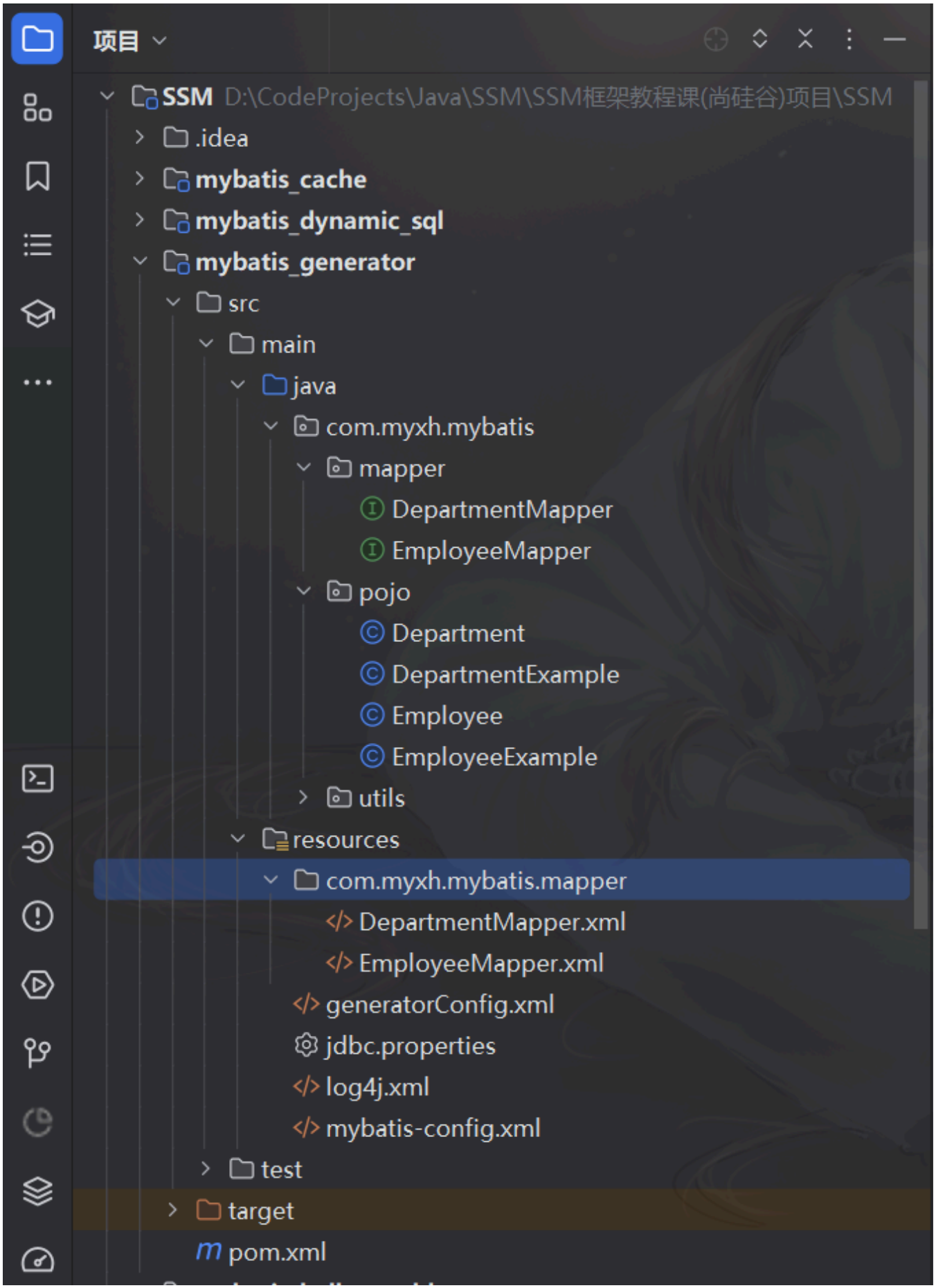
> 文件名必须是：generatorConfig.xml。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!--
    targetRuntime: 执行生成的逆向工程的版本
    MyBatis3Simple: 生成基本的 CRUD（清新简洁版）
    MyBatis3: 生成带条件的 CRUD（奢华尊享版）
     -->
    <context id="DB2Tables" targetRuntime="MyBatis3">
        <!-- 数据库的连接信息 -->
        <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"
                        connectionURL="jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC"
                        userId="MYXH"
                        password="520.ILY!">
        </jdbcConnection>

        <!-- javaBean 的生成策略-->
        <javaModelGenerator targetPackage="com.myxh.mybatis.pojo"
                            targetProject=".\src\main\java">
            <property name="enableSubPackages" value="true"/>
            <property name="trimStrings" value="true"/>
        </javaModelGenerator>

        <!-- SQL 映射文件的生成策略 -->
        <sqlMapGenerator targetPackage="com.myxh.mybatis.mapper"
                         targetProject=".\src\main\resources">
            <property name="enableSubPackages" value="true"/>
        </sqlMapGenerator>

        <!-- Mapper 接口的生成策略 -->
        <javaClientGenerator type="XMLMAPPER"
                             targetPackage="com.myxh.mybatis.mapper" targetProject=".\src\main\jav
            <property name="enableSubPackages" value="true"/>
        </javaClientGenerator>

        <!-- 逆向分析的表 -->
        <!-- tableName 设置为 * 号，可以对应所有表，此时不写 domainObjectName -->
        <!-- domainObjectName 属性指定生成出来的实体类的类名 -->
        <table tableName="t_employee" domainObjectName="Employee"/>
        <table tableName="t_department" domainObjectName="Department"/>
    </context>
</generatorConfiguration>
```

11.1.4 ④ 执行 MBG（myBatis-generate） 插件的 generate 目标



11.1.5 ⑤ 効果


项目
SSM D:\CodeProjects\Java\SSM\SSM框架教程课(尚硅谷)项目\SSM
.idea
mybatis_cache
mybatis_dynamic_sql
mybatis_generator
src
main
java
com.myxh.mybatis
mapper
DepartmentMapper
EmployeeMapper
pojo
Department
DepartmentExample
Employee
EmployeeExample
utils
resources
com.myxh.mybatis.mapper
DepartmentMapper.xml
EmployeeMapper.xml
generatorConfig.xml
jdbc.properties
log4j.xml
mybatis-config.xml
test
target
pom.xml

11.2、QBC（Query By Example 根据样例查询） 查询

```java
package com.myxh.mybatis.test;

import com.myxh.mybatis.mapper.EmployeeMapper;
import com.myxh.mybatis.pojo.Employee;
import com.myxh.mybatis.utils.SqlSessionUtil;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;

/**
 * @author MYXH
 * @date 2023/8/15
 */
public class mybatisGeneratorTest
{
    @Test
    public void testMybatisGenerator()
    {
        SqlSession sqlSession = SqlSessionUtil.getSqlSession();
        EmployeeMapper mapper = sqlSession.getMapper(EmployeeMapper.class);

        // 根据 id 查询数据
        Employee employee = mapper.selectByPrimaryKey(1);
        System.out.println("结果: " + employee);

        // 查询所有数据
        List<Employee> employeeList = mapper.selectByExample(null);
        employeeList.forEach(System.out::println);

        // 根据条件查询数据

        EmployeeExample example = new EmployeeExample();
        example.createCriteria().andEmployeeNameEqualTo("MYXH").andAgeGreaterThanOrEqualTo(20);
        example.or().andGenderEqualTo("男");
        List<Employee> employeeList = mapper.selectByExample(example);
        employeeList.forEach(System.out::println);

        // 测试普通修改功能
        /*
        Employee employee = new Employee(1, "末影小黑xh", 21, "男" , "1735350920@qq.com", null);
        int result = mapper.updateByPrimaryKey(employee);
        System.out.println("结果: " + result);
         */

        // 测试选择性修改功能
```

```
        Employee employee = new Employee(1, "末影小黑xh", 21, "男" , "1735350920@qq.com", null);
        int result = mapper.updateByPrimaryKeySelective(employee);
        System.out.println("结果: " + result);

        sqlSession.close();
    }
}
```

12、分页插件

SQL 语句：

```
SELECT * FROM t_employee
LIMIT index, pageSize;
```

分页代码：

```
// 每页显示的条数
int pageSize;

// 当前页的页码
int pageNum;

// 当前页的起始索引
int index = (pageNum - 1) \* pageSize;

// 总记录数
int count;

// 总页数
int totalPage = count / pageSize;

if(count % pageSize != 0)
{
    totalPage += 1;
}
```

计算规律：

> pageSize = 5，pageNum = 1，index = 0，limit 0, 5

> pageSize = 5，pageNum = 3，index = 10，limit 10, 5
>
> pageSize = 5，pageNum = 6，index = 25，limit 25, 5

分页效果：

> 首页 上一页 2 3 4 5 6 下一页 末页

12.1、分页插件的使用步骤

12.1.1 ① 添加依赖

```
<!-- 分页插件 -->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.3.2</version>
</dependency>
```

12.1.2 ② 配置分页插件

在 MyBatis 的核心配置文件中配置插件。

```
<plugins>
    <!-- 设置分页插件 -->
    <plugin interceptor="com.github.pagehelper.PageInterceptor"/>
</plugins>
```

12.2、分页插件的使用

1、在查询功能之前使用 PageHelper.startPage(int pageNum, int pageSize);开启分页功能。

> pageNum：当前页的页码
>
> pageSize：每页显示的条数

2、在查询获取 list 集合之后，使用 PageInfo pageInfo = new PageInfo<>(List list, int navigatePages);获取分页相关数据。

> list：分页之后的数据

> navigatePages：导航分页的页码数

3、分页相关数据。

> pageInfo = PageInfo{
> pageNum=1,
> pageSize=5,
> size=5,
> startRow=1,
> endRow=5,
> total=30,
> pages=6,
> list=Page{
> count=true,
> pageNum=1,
> pageSize=5,
> startRow=0,
> endRow=5,
> total=30,
> pages=6,
> reasonable=false,
> pageSizeZero=false
> }
> [Employee{employeeId=1, employeeName='MYXH', age=21, gender='男',
> email='1735350920@qq.com', departmentId=1}, Employee{employeeId=2,
> employeeName='张三', age=20, gender='男', email='zhangsan@qq.com',
> departmentId=1}, Employee{employeeId=3, employeeName='李四', age=22, gender='男',
> email='lisi@qq.com', departmentId=2}, Employee{employeeId=4, employeeName='王五',
> age=23, gender='男', email='wangwu@qq.com', departmentId=2},
> Employee{employeeId=5, employeeName='赵六', age=24, gender='男',
> email='zhaoliu@qq.com', departmentId=3}],
> prePage=0,
> nextPage=2,
> isFirstPage=true,
> isLastPage=false,
> hasPreviousPage=false,

```
hasNextPage=true,
navigatePages=5,
navigateFirstPage=1,
navigateLastPage=5,
navigatepageNums=[1, 2, 3, 4, 5]}
```

pageNum：当前页的页码

pageSize：每页显示的条数

size：当前页显示的真实条数

total：总记录数

pages：总页数

prePage：上一页的页码

nextPage：下一页的页码

isFirstPage/isLastPage：是否为第一页/最后一页

hasPreviousPage/hasNextPage：是否存在上一页/下一页

navigatePages：导航分页的页码数

navigatepageNums：导航分页的页码，[1, 2, 3, 4, 5]