

实验二：MyBatis 的核心配置

一、 **实验目的：**对 MyBatis 框架中的核心对象，以及映射文件和配置文件有更加深入的了解。

二、 **预习要求：**

- 1、了解 MyBatis 核心对象的作用
- 2、熟悉 MyBatis 配置文件中各个元素的作用
- 3、掌握 MyBatis 映射文件中常用元素的使用

三、 **实验内容：**

四、 **实验方法和步骤：**

配置内容

SqlMapConfig.xml 中配置的内容和顺序如下：

properties（属性）
settings（全局配置参数）
typeAliases（类型别名）
typeHandlers（类型处理器）
objectFactory（对象工厂）
plugins（插件）
environments（环境集合属性对象）
 environment（环境子属性对象）
 transactionManager（事务管理）
 dataSource（数据源）
mappers（映射器）

properties（属性）

SqlMapConfig.xml 可以引用 java 属性文件中的配置信息如下：

在 Resources 下定义 db.properties 文件，如下所示：

db.properties 配置文件内容如下:

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test?characterEncoding=utf-8
jdbc.username=root
jdbc.password=123
```

SqlMapConfig.xml 引用如下:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 是用 resource 属性加载外部配置文件 -->
    <properties resource="db.properties">
        <!-- 在 properties 内部用 property 定义属性 -->
        <!-- 如果外部配置文件有该属性, 则内部定义属性被外部属性覆盖 -->
        <property name="jdbc.username" value="root" />
        <property name="jdbc.password" value="123" />
    </properties>

    <!-- 和 spring 整合后 environments 配置将废除 -->
    <environments default="development">
        <environment id="development">
            <!-- 使用 jdbc 事务管理 -->
            <transactionManager type="JDBC" />
            <!-- 数据库连接池 -->
            <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>

    <!-- 加载映射文件 -->
    <mappers>
        <mapper resource="sqlmap/User.xml" />
        <mapper resource="com/haust/mapper/CustomerMapper.xml" />
    </mappers>
</configuration>
```

注意： MyBatis 将按照下面的顺序来加载属性：

- ◆ 在 `properties` 元素体内定义的属性首先被读取。
- ◆ 然后会读取 `properties` 元素中 `resource` 或 `url` 加载的属性，它会覆盖已读取的同名属性。

typeAliases（类型别名）

mybatis 支持别名：

别名	映射的类型
<code>_byte</code>	<code>byte</code>
<code>_long</code>	<code>long</code>
<code>_short</code>	<code>short</code>
<code>_int</code>	<code>int</code>
<code>_integer</code>	<code>int</code>
<code>_double</code>	<code>double</code>
<code>_float</code>	<code>float</code>
<code>_boolean</code>	<code>boolean</code>
<code>string</code>	<code>String</code>
<code>byte</code>	<code>Byte</code>
<code>long</code>	<code>Long</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>integer</code>	<code>Integer</code>
<code>double</code>	<code>Double</code>
<code>float</code>	<code>Float</code>
<code>boolean</code>	<code>Boolean</code>
<code>date</code>	<code>Date</code>
<code>decimal</code>	<code>BigDecimal</code>
<code>bigdecimal</code>	<code>BigDecimal</code>
<code>map</code>	<code>Map</code>

自定义别名：

在 `SqlMapConfig.xml` 中配置如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
```

```

<!-- 是用 resource 属性加载外部配置文件 -->
<properties resource="db.properties">
    <!-- 在 properties 内部用 property 定义属性 -->
    <property name="jdbc.username" value="root" />
    <property name="jdbc.password" value="123" />
</properties>

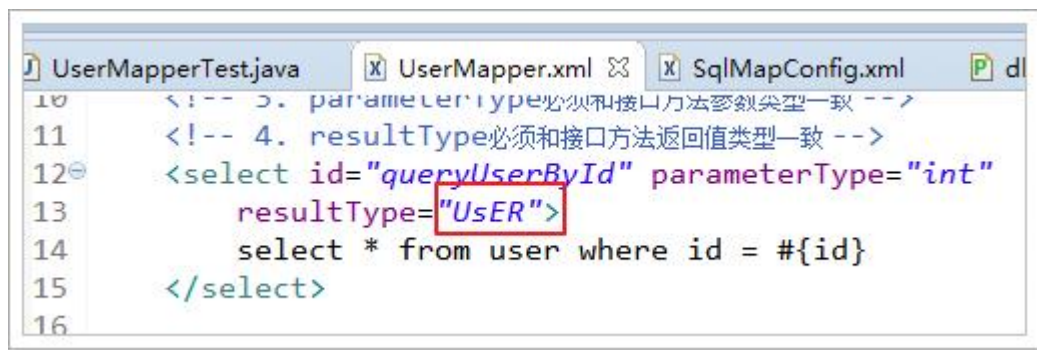
<typeAliases>
    <!-- 单个别名定义 -->
    <typeAlias alias="user" type="com.haust.pojo.User" />
    <!-- 批量别名定义，扫描整个包下的类，别名为类名（大小写不敏感） -->
    <package name="com.haust.pojo" />
    <package name="其它包" />
</typeAliases>

<!-- 和 spring 整合后 environments 配置将废除 -->
<environments default="development">
    <environment id="development">
        <!-- 使用 jdbc 事务管理 -->
        <transactionManager type="JDBC" />
        <!-- 数据库连接池 -->
        <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>

<!-- 加载映射文件 -->
<mappers>
    <mapper resource="sqlmap/User.xml" />
    <mapper resource="com/haust/mapper/CustomerMapper.xml" />
</mappers>
</configuration>

```

在 mapper.xml 配置文件中，就可以使用设置的别名了
别名大小写不敏感



mappers（映射器）

Mapper 配置的几种方法：

<mapper resource=" " />

使用相对于类路径的资源（现在的使用方式）

如：<mapper resource="com/haust/mapper/CustomMapper.xml" />

<mapper class=" " />

使用 mapper 接口类路径

如：<mapper class=com.haust.mapper.CustomMapper" />

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

<package name="" />

注册指定包下的所有 mapper 接口

如：<package name="com.haust.mapper"/>

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

输入映射和输出映射

Mapper.xml 映射文件中定义了操作数据库的 sql，每个 sql 是一个 statement，映射文件是 mybatis 的核心。

parameterType(输入类型)

传递简单类型

参考前面实验内容

传递 pojo 对象

Mybatis 使用 ognl 表达式解析对象字段的值，#{ }或者\${ }括号中的值为 pojo 属性名称。

传递 pojo 包装对象

开发中通过 pojo 传递查询条件，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。

Pojo 类中包含 pojo。

需求：根据用户名查询用户信息，查询条件放到 QueryVo 的 user 属性中。

QueryVo

```
public class QueryVo {  
  
    private User user;  
  
    public User getUser() {  
        return user;  
    }  
  
    public void setUser(User user) {  
        this.user = user;  
    }  
}
```

Sql 语句

```
SELECT * FROM user where username like '%刘%'
```

Mapper 文件

```
<!-- 使用包装类型查询用户
      使用 ognl 从对象中取属性值，如果是包装对象可以使用.操作符来取内容部的属性
-->
<select id="findUserByQueryVo" parameterType="queryvo" resultType="user">
    SELECT * FROM user where username like '%${user.username}%'
</select>
```

接口

```
6  */
7  public interface UserMapper {
8
9      User findUserById(int id) throws Exception;
10     void insertUser(User user) throws Exception;
11     List<User> findUserByQueryVo(QueryVo queryVo) throws Exception;
12 }
13
```

测试方法

```
@Test
    public void testFindUserByQueryVo() throws Exception {
        SqlSession sqlSession = sessionFactory.openSession();
        //获得 mapper 的代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        //创建 QueryVo 对象
        QueryVo queryVo = new QueryVo();
        //创建 user 对象
        User user = new User();
        user.setUsername("刘");
        queryVo.setUser(user);
        //根据 queryvo 查询用户
        List<User> list = userMapper.findUserByQueryVo(queryVo);
        System.out.println(list);
        sqlSession.close();
    }
```

resultType(输出类型)

输出简单类型

参考 getnow 输出日期类型，看下边的例子输出整型：

Mapper.xml 文件

```
<!-- 获取用户列表总数 -->
<select id="findUserCount" resultType="int">
    select count(1) from user
</select>
```

Mapper 接口

```
public int findUserCount() throws Exception;
```

调用：

```
Public void testFindUserCount() throws Exception{
    //获取session
    SqlSession session = sqlSessionSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);

    //传递HashMap对象查询用户列表
    int count = userMapper.findUserCount();

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

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。使用 session 的 selectOne 可查询单条记录。

输出 pojo 对象

参考实验内容

输出 pojo 列表

参考实验内容

resultMap

resultType 可以指定 pojo 将查询结果映射为 pojo，但需要 pojo 的属性名和 sql 查询的列名一致方可映射成功。

如果 sql 查询字段名和 pojo 的属性名不一致，可以通过 resultMap 将字段名和属性名作一个对应关系，resultMap 实质上还需要将查询结果映射到 pojo 对象中。

resultMap 可以实现将查询结果映射为复杂类型的 pojo，比如在查询结果映射对象中包含 pojo 和 list 实现一对一查询和一对多查询。

Mapper.xml 定义

```
<!-- 查询用户列表 根据用户名称和用户性别查询用户列表 -->
<select id="findUserListResultMap" parameterType="queryVo" resultMap="userListResultMap">
    select id id_,username username_,birthday birthday_ from user

    <!-- where自动将第一个and去掉 -->
    <where>
        <!--
        refid: 指定 sql片段的id, 如果要引用其它命名空间的sql片段, 需要前边加namespace
        -->
        <include refid="query_user_where"/>
    </where>
</select>
```

使用 resultMap 指定上边定义的 userListResultMap。

定义 resultMap

由于上边的 mapper.xml 中 sql 查询列和 Users.java 类属性不一致，需要定义 resultMap：userListResultMap 将 sql 查询列和 Users.java 类属性对应起来

```

<!-- 定义resultMap，将用户查询的字段和user这个pojo的属性名作一个对应关系 -->
<!--
type:最终映射的java对象。
id: resultMap的唯一标识
-->
<resultMap type="user" id="userListResultMap">
  <!-- id标签: 查询结果集的唯一标识列(主键或唯一标识)
  column: sql查询字段名(列名)
  property: pojo的属性名

  resultMap标签: 普通列
  -->

  <id column="id_" property="id"/>
  <result column="username_" property="username"/>
  <result column="birthday_" property="birthday"/>

</resultMap>

```

<id />: 此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个<id />。

Property: 表示 User 类的属性。

Column: 表示 sql 查询出来的字段名。

Column 和 property 放在一块儿表示将 sql 查询出来的字段映射到指定的 pojo 类属性上。

<result />: 普通结果，即 pojo 的属性。

Mapper 接口定义

```
public List<User> findUserListResultMap() throws Exception;
```

1. 传递 pojo 包装对象，并结合动态代理 Mapper 进行测试

开发中通过 pojo 传递查询条件，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。

Pojo 类中包含 pojo。

需求：根据用户名查询用户信息，查询条件放到 QueryVo 的 user 属性中。

QueryVo

```

public class QueryVo {
    private User user;

    public User getUser() {
        return user;
    }

    public void setUser(User user) {

```

```

        this.user = user;
    }
}

```

Sql 语句

SELECT * FROM user where username like '%刘%'

Mapper 文件

```

<!-- 使用包装类型查询用户
      使用 ognl 从对象中取属性值，如果是包装对象可以使用.操作符来取内容部的属性
-->
<select id="findUserByQueryVo" parameterType="queryvo" resultType="user">
    SELECT * FROM user where username like '%${user.username}%'
</select>

```

接

口

```

6  */
7  public interface UserMapper {
8
9      User findUserById(int id) throws Exception;
10     void insertUser(User user) throws Exception;
11     List<User> findUserByQueryVo(QueryVo queryVo) throws Exception;
12 }
13

```

测试方法

```

@Test
    public void testFindUserByQueryVo() throws Exception {
        SqlSession sqlSession = sessionFactory.openSession();
        //获得 mapper 的代理对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        //创建 QueryVo 对象
        QueryVo queryVo = new QueryVo();
        //创建 user 对象
        User user = new User();
        user.setUsername("刘");
        queryVo.setUser(user);
        //根据 queryvo 查询用户
        List<User> list = userMapper.findUserByQueryVo(queryVo);
    }

```

```
        System.out.println(list);

        sqlSession.close();
    }
}
```

2. 输出简单类型，并结合动态代理 Mapper 进行测试

看下边的例子输出整型：

UserMapper.xml 文件

```
<!-- 获取用户列表总数 -->
<select id="findUserCount" resultType="int">
    select count(1) from user
</select>
```

UserMapper 接口

```
public int findUserCount() throws Exception;
```

调用：

```
Public void testFindUserCount() throws Exception{
    //获取session
    SqlSession session = sqlSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);

    //传递HashMap对象查询用户列表
    int count = userMapper.findUserCount();
    System.out.println(count);
    //关闭session
    session.close();
}
```

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。使用 session 的 selectOne 可查询单条记录。

3. 手动映射 resultMap 练习，并结合动态代理 Mapper 进行测试

resultType 可以指定 pojo 将查询结果映射为 pojo，但需要 pojo 的属性名和 sql 查询的列名一致方可映射成功。

如果 sql 查询字段名和 pojo 的属性名不一致，可以通过 resultMap 将字段名和属性名作一个对应关系，resultMap 实质上还需要将查询结果映射到 pojo 对象中。

resultMap 可以实现将查询结果映射为复杂类型的 pojo，比如在查询结果映射对象中包括 pojo 和 list 实现一对一查询和一对多查询。

UserMapper.xml 定义

```
<!-- 查询用户列表 根据用户名和用户性别查询用户列表 -->
<select id="findUserListResultMap" parameterType="queryVo" resultMap="userListResultMap">
    select id id_,username username_,birthday birthday_ from user

    <!-- where自动将第一个and去掉 -->
    <where>
        <!--
        refid: 指定 sql片段的id, 如果要引用其它命名空间的sql片段, 需要前边加namespace

        -->
        <include refid="query_user_where"/>
    </where>
</select>
```

使用 resultMap 指定上边定义的 userListResultMap。

定义 resultMap

由于上边的 UserMapper.xml 中 sql 查询列和 Users.java 类属性不一致，需要定义 resultMap：userListResultMap 将 sql 查询列和 Users.java 类属性对应起来

```
<!-- 定义resultMap, 将用户查询的字段和user这个pojo的属性名作一个对应关系 -->
<!--
type:最终映射的java对象。
id: resultMap的唯一标识
-->
<resultMap type="user" id="userListResultMap">
    <!-- id标签: 查询结果集的唯一标识 列(主键或唯一标识)
    column: sql查询字段名(列名)
    property: pojo的属性名

    resultMap标签: 普通列
    -->

    <id column="id_" property="id"/>
    <result column="username_" property="username"/>
    <result column="birthday_" property="birthday"/>

</resultMap>
```

<id />：此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个<id />。

Property：表示 User 类的属性。

Column：表示 sql 查询出来的字段名。

Column 和 property 放在一块儿表示将 sql 查询出来的字段映射到指定的 pojo 类属性上。

<result />：普通结果，即 pojo 的属性。

UserMapper 接口定义

```
public List<User> findUserListResultMap() throws Exception;
```

调用:

```
Public void testfindUserListResultMap() throws Exception{
    //获取session
    SqlSession session = sqlSessionSessionFactory.openSession();
    //获取mapper接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);

    //传递HashMap对象查询用户列表
    List<User> users = userMapper.findUserListResultMap();
    System.out.println(users);
    //关闭session
    session.close();
}
```

4. 测试添加, 并结合动态代理 Mapper 进行测试

在 UserMapper.xml 中添加:

```
<!-- 添加用户 -->
<insert id="insertUser" parameterType="com.haust.pojo.User">
    <selectKey keyProperty="id" order="AFTER"
resultType="java.lang.Integer">
        select LAST_INSERT_ID()
    </selectKey>
    insert into user(username,birthday,sex,address)
    values(#{username},#{birthday},#{sex},#{address})
</insert>
```

UserMapper 接口定义

```
public void insertUser (User user) throws Exception;
```

测试程序:

```
// 添加用户信息
@Test
public void testInsert() {
    // 数据库会话实例
    SqlSession sqlSession = null;
    try {
        // 创建数据库会话实例sqlSession
```

```

        sqlSession = sqlSessionSessionFactory.openSession();
        // 添加用户信息
        User user = new User();
        user.setUsername("张小明");
        user.setAddress("河南郑州");
        user.setSex("1");
        user.setBirthday(new Date());
        UserMapper userMapper = session.getMapper(UserMapper.class);
        userMapper.insertUser(user);
        //提交事务
        sqlSession.commit();
        System.out.println(user);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (sqlSession != null) {
            sqlSession.close();
        }
    }
}

```

在插入测试中注意主键的以下两个问题，请大家自行测试

(1)mysql 自增主键返回

通过修改 sql 映射文件，可以将 mysql 自增主键返回：

```

<insert id="insertUser" parameterType="com.haust.pojo.User">
    <!-- selectKey将主键返回，需要再返回 -->
    <selectKey keyProperty="id" order="AFTER"
resultType="java.lang.Integer">
        select LAST_INSERT_ID()
    </selectKey>
    insert into user(username,birthday,sex,address)
    values(#{username},#{birthday},#{sex},#{address});
</insert>

```

添加 `selectKey` 实现将主键返回

`keyProperty`: 返回的主键存储在 `pojo` 中的哪个属性

`order`: `selectKey` 的执行顺序，是相对与 `insert` 语句来说，由于 `mysql` 的自增原理执行完 `insert` 语句之后才将主键生成，所以这里 `selectKey` 的执行顺序为 `after`

`resultType`: 返回的主键是什么类型

`LAST_INSERT_ID()`: 是 `mysql` 的函数，返回 `auto_increment` 自增列新记录 `id`

值。

(2)Mysql 使用 uuid 实现主键

需要增加通过 select uuid()得到 uuid 值

```
<insert id="insertUser" parameterType="com.haust.po.User">
<selectKey resultType="java.lang.String" order="BEFORE"
keyProperty="id">
select uuid()
</selectKey>
insert into user(id,username,birthday,sex,address)
values("#{id}",#{username},#{birthday},#{sex},#{address})
</insert>
```

注意这里使用的 order 是 “BEFORE”

五、 思考题：

无

六、 实验作业要求：

- (1)实验目的：
- (2)实验内容：
- (3)实验结果：可以是运行结果截图或其他形式的结果展示
- (4)问题及解决：实验中遇到的问题及解决方法。
- (5)回答思考题提出的问题。