# Mybatis配置文件

# SqlMapConfig.xml配置文件

## 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classpath下定义db.properties文件，

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/userdb

jdbc.username=root

jdbc.password=a62134787

SqlMapConfig.xml引用如下：

<properties resource=*"**db.properties"*/>

<environments default=*"development"*>

<environment id=*"development"*>

<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>

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。
* 最后读取parameterType传递的属性，它会覆盖已读取的同名属性。

因此，通过parameterType传递的属性具有最高优先级，resource或 url 加载的属性次之，最低优先级的是 properties 元素体内定义的属性。

## settings（配置）

mybatis全局配置参数，全局参数将会影响mybatis的运行行为。

详细参见“学习资料/mybatis-settings.xlsx”文件







## typeAliases（类型别名）

### mybatis支持别名：

|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |

### 自定义别名：

在SqlMapConfig.xml中配置：

<typeAliases>

<!-- 单个别名定义 -->

<typeAlias alias=*"user"* type=*"com.mybatis.po.User"*/>

<!-- 批量别名定义，扫描整个包下的类，别名为类名（首字母大写或小写都可以） -->

<package name=*"com.mybatis.po"*/>

<package name=*"其它包"*/>

</typeAliases>

## typeHandlers（类型处理器）

类型处理器用于java类型和jdbc类型映射，如下：

<select id=*"findUserById"* parameterType=*"int"* resultType=*"user"*>

select \* from user where id = #{id}

</select>

mybatis自带的类型处理器基本上满足日常需求，不需要单独定义。

mybatis支持类型处理器：

|  |  |  |
| --- | --- | --- |
| 类型处理器 | **Java**类型 | **JDBC**类型 |
| BooleanTypeHandler | Boolean，boolean | 任何兼容的布尔值 |
| ByteTypeHandler | Byte，byte | 任何兼容的数字或字节类型 |
| ShortTypeHandler | Short，short | 任何兼容的数字或短整型 |
| IntegerTypeHandler | Integer，int | 任何兼容的数字和整型 |
| LongTypeHandler | Long，long | 任何兼容的数字或长整型 |
| FloatTypeHandler | Float，float | 任何兼容的数字或单精度浮点型 |
| DoubleTypeHandler | Double，double | 任何兼容的数字或双精度浮点型 |
| BigDecimalTypeHandler | BigDecimal | 任何兼容的数字或十进制小数类型 |
| StringTypeHandler | String | CHAR和VARCHAR类型 |
| ClobTypeHandler | String | CLOB和LONGVARCHAR类型 |
| NStringTypeHandler | String | NVARCHAR和NCHAR类型 |
| NClobTypeHandler | String | NCLOB类型 |
| ByteArrayTypeHandler | byte[] | 任何兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB和LONGVARBINARY类型 |
| DateTypeHandler | Date（java.util） | TIMESTAMP类型 |
| DateOnlyTypeHandler | Date（java.util） | DATE类型 |
| TimeOnlyTypeHandler | Date（java.util） | TIME类型 |
| SqlTimestampTypeHandler | Timestamp（java.sql） | TIMESTAMP类型 |
| SqlDateTypeHandler | Date（java.sql） | DATE类型 |
| SqlTimeTypeHandler | Time（java.sql） | TIME类型 |
| ObjectTypeHandler | 任意 | 其他或未指定类型 |
| EnumTypeHandler | Enumeration类型 | VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。 |

## 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

如：<mapper resource="sqlmap/User.xml" />

### <mapper url=" " />

使用完全限定路径

如：<mapper url="file:///D:\workspace\_spingmvc\mybatis\_01\config\sqlmap\User.xml" />

### <mapper class=" " />

使用mapper接口类路径

如：<mapper class="com.mybatis.mapper.UserMapper"/>

**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.mybatis.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。**

# Mapper.xml映射文件

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)

### #{}与${}

#{}实现的是向prepareStatement中的预处理语句中设置参数值，sql语句中#{}表示一个占位符即?。

<!-- 根据id查询用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"user"*>

select \* from user where id = #{id}

</select>

使用占位符#{}可以有效防止sql注入，在使用时不需要关心参数值的类型，mybatis会自动进行java类型和jdbc类型的转换。#{}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

${}和#{}不同，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。使用${}不能防止sql注入，但是有时用${}会非常方便，如下的例子：

<!-- 根据名称模糊查询用户信息 -->

<select id=*"selectUserByName"* parameterType=*"string"* resultType=*"user"*>

select \* from user where username like '%${value}%'

</select>

如果本例子使用#{}则传入的字符串中必须有%号，而%是人为拼接在参数中，显然有点麻烦，如果采用${}在sql中拼接为%的方式则在调用mapper接口传递参数就方便很多。

//如果使用占位符号则必须人为在传参数中加%

List<User> list = userMapper.selectUserByName("%管理员%");

//如果使用${}原始符号则不用人为在参数中加%

List<User>list = userMapper.selectUserByName("管理员");

再比如order by排序，如果将列名通过参数传入sql，根据传的列名进行排序，应该写为：

ORDER BY ${columnName}

如果使用#{}将无法实现此功能。

### 传递简单类型

参考上边的例子。

### 传递pojo对象

Mybatis使用ognl表达式解析对象字段的值，如下例子：

<!—传递pojo对象综合查询用户信息 -->

<select id=*"findUserByUser"* parameterType=*"user"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是user对象中的字段名称。

测试：

**Public void** testFindUserByUser()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件user对象

User user = **new User()**;

user.setId(1);

user.setUsername("管理员");

//传递user对象查询用户列表

List<User>list = userMapper.findUserByUser(user);

//关闭session

session.close();

}

异常测试：

Sql中字段名输入错误后测试，username输入dusername测试结果报错：

org.apache.ibatis.exceptions.PersistenceException:

### Error querying database. Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class com.mybatis.po.User'

### Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class com.mybatis.po.User'

### 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。

#### 定义包装对象

定义包装对象将查询条件(pojo)以类组合的方式包装起来。

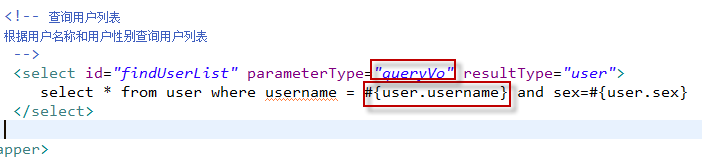
**public** **class** QueryVo {

**private** User user;

//自定义用户扩展类

**private** UserCustom userCustom;

#### mapper.xml映射文件



说明：mybatis底层通过ognl从pojo中获取属性值：#{user.username}，user即是传入的包装对象的属性。queryVo是别名，即上边定义的包装对象类型。

### 传递hashmap

Sql映射文件定义如下：

<!-- 传递hashmap综合查询用户信息 -->

<select id=*"findUserByHashmap"* parameterType=*"hashmap"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是hashmap的key。

测试：

**Public void** testFindUserByHashmap()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件Hashmap对象

HashMap<String, Object> map = **new** HashMap<String, Object>();

map.put("id", 1);

map.put("username", "管理员");

//传递Hashmap对象查询用户列表

List<User>list = userMapper.findUserByHashmap(map);

//关闭session

session.close();

}

异常测试：

传递的map中的key和sql中解析的key不一致。

测试结果没有报错，只是通过key获取值为空。

## 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"* parameterType=*"user"* resultType=*"int"*>

select count(1) from user

</select>

Mapper接口

public int findUserCount(User user) 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**);

User user = **new User()**;

user.setUsername("管理员");

//传递Hashmap对象查询用户列表

**int** count = userMapper.findUserCount(user);

//关闭session

session.close();

}

**总结：**

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findUserById的定义：

Mapper.xml

<!-- 根据id查询用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"user"*>

select \* from user where id = #{id}

</select>

Mapper接口：

public User findUserById(int id) throws Exception;

测试：

**Public void** testFindUserById() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//通过mapper接口调用statement

User user = userMapper.findUserById(1);

System.out.println(user);

//关闭session

session.close();

}

使用session调用selectOne查询单条记录。

### 输出pojo列表

参考selectUserByName的定义：

Mapper.xml

<!-- 根据名称模糊查询用户信息 -->

<select id=*"findUserByUsername"* parameterType=*"string"* resultType=*"user"*>

select \* from user where username like '%${value}%'

</select>

Mapper接口：

public List<User> findUserByUsername(String username) throws Exception;

测试：

**Public void** testFindUserByUsername()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//如果使用占位符号则必须人为在传参数中加%

//List<User> list = userMapper.selectUserByName("%管理员%");

//如果使用${}原始符号则不用人为在参数中加%

List<User> list = userMapper.findUserByUsername("管理员");

//关闭session

session.close();

}

使用session的selectList方法获取pojo列表。

### resultType总结：

输出pojo对象和输出pojo列表在sql中定义的resultType是一样的。

返回单个pojo对象要保证sql查询出来的结果集为单条，内部使用session.selectOne方法调用，mapper接口使用pojo对象作为方法返回值。

返回pojo列表表示查询出来的结果集可能为多条，内部使用session.selectList方法，mapper接口使用List<pojo>对象作为方法返回值。

### 输出hashmap

输出pojo对象可以改用hashmap输出类型，将输出的字段名称作为map的key，value为字段值。

## 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括pojo和list实现一对一查询和一对多查询。

### Mapper.xml定义



使用resultMap指定上边定义的personmap。

### 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



<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person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;

## 动态sql(重点)

通过mybatis提供的各种标签方法实现动态拼接sql。

### If

<!-- 传递pojo综合查询用户信息 -->

<select id=*"**findUserList"* parameterType=*"user"* resultType=*"user"*>

select \* from user

where 1=1

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</select>

**注意要做不等于空字符串校验。**

### Where

上边的sql也可以改为：

<select id=*"findUserList"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</where>

</select>

<where />可以自动处理第一个and。

### foreach

向sql传递数组或List，mybatis使用foreach解析，如下：

#### 通过pojo传递list

* 需求

传入多个id查询用户信息，用下边两个sql实现：

SELECT \* FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)

SELECT \* FROM USERS WHERE username LIKE '%张%' id IN (10,89,16)

* 在pojo中定义list属性ids存储多个用户id，并添加getter/setter方法



* mapper.xml

<if test=*"ids!=null and ids.size>0"*>

<foreach collection=*"ids"* open=*" and id in("* close=*")"* item=*"id"* separator=*","* >

#{id}

</foreach>

</if>

* 测试代码：

List<Integer> ids = **new** ArrayList<Integer>();

ids.add(1);//查询id为1的用户

ids.add(10); //查询id为10的用户

queryVo.setIds(ids);

List<User> list = userMapper.findUserList(queryVo);

#### 传递单个List

传递List类型在编写mapper.xml没有区别，唯一不同的是只有一个List参数时它的参数名为list。

如下：

* Mapper.xml

<select id=*"selectUserByList"* parameterType=*"java.util.List"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递List，List中是pojo -->

<if test=*"list!=null"*>

<foreach collection=*"list"* item=*"item"* open=*"and id in("*separator=*","*close=*")"*>

#{item.id}

</foreach>

</if>

</where>

</select>

* Mapper接口

public List<User> selectUserByList(List userlist) throws Exception;

* 测试：

**Public void** testselectUserByList()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

List<User> userlist = **new** ArrayList<User>();

User user = **new User()**;

user.setId(1);

userlist.add(user);

user = **new User()**;

user.setId(2);

userlist.add(user);

//传递userlist列表查询用户列表

List<User>list = userMapper.selectUserByList(userlist);

//关闭session

session.close();

}

#### 传递单个数组（数组中是pojo）：

请阅读文档学习。

* Mapper.xml

<!-- 传递数组综合查询用户信息 -->

<select id=*"selectUserByArray"* parameterType=*"Object[]"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递数组 -->

<if test=*"array!=null"*>

<foreach collection=*"array"* index=*"index"* item=*"item"* open=*"and id in("*separator=*","*close=*")"*>

#{item.id}

</foreach>

</if>

</where>

</select>

sql只接收一个数组参数，这时sql解析参数的名称mybatis固定为array，如果数组是通过一个pojo传递到sql则参数的名称为pojo中的属性名。

index：为数组的下标。

item：为数组每个元素的名称，名称随意定义

open：循环开始

close：循环结束

separator：中间分隔输出

* Mapper接口：

public List<User> selectUserByArray(Object[] userlist) throws Exception;

* 测试：

**Public void** testselectUserByArray()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

Object[] userlist = **new** Object[2];

User user = **new User()**;

user.setId(1);

userlist[0]=user;

user = **new User()**;

user.setId(2);

userlist[1]=user;

//传递user对象查询用户列表

List<User>list = userMapper.selectUserByArray(userlist);

//关闭session

session.close();

}

#### 传递单个数组（数组中是字符串类型）：

请阅读文档学习。

* Mapper.xml

<!-- 传递数组综合查询用户信息 -->

<select id=*"selectUserByArray"* parameterType=*"Object[]"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递数组 -->

<if test=*"array!=null"*>

<foreach collection=*"array"*index=*"index"*item=*"item"*open=*"and id in("*separator=*","*close=*")"*>

#{item}

</foreach>

</if>

</where>

</select>

如果数组中是简单类型则写为#{item}，不用再通过ognl获取对象属性值了。

* Mapper接口：

public List<User> selectUserByArray(Object[] userlist) throws Exception;

* 测试：

**Public void** testselectUserByArray()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

Object[] userlist = **new** Object[2];

userlist[0]=”1”;

userlist[1]=”2”;

//传递user对象查询用户列表

List<User>list = userMapper.selectUserByArray(userlist);

//关闭session

session.close();

}

### Sql片段

Sql中可将重复的sql提取出来，使用时用include引用即可，最终达到sql重用的目的，如下：

<!-- 传递pojo综合查询用户信息 -->

<select id=*"findUserList"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</where>

</select>

* 将where条件抽取出来：

<sql id=*"query\_user\_where"*>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</sql>

* 使用include引用：

<select id=*"findUserList"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<include refid=*"query\_user\_where"*/>

</where>

</select>

注意：如果引用其它mapper.xml的sql片段，则在引用时需要加上namespace，如下：

<include refid=*"*namespace.sql片段”/>