Rays's

Personal Study Note

of

Mybatis

目 录

[1. MyBatis概述 1](#_Toc15599089)

[1.1. 简介 1](#_Toc15599090)

[1.2. Hello World 1](#_Toc15599091)

[1.2.1. 基于xml文件的Hello World 1](#_Toc15599092)

[1.2.2. 基于代码配置的Hello World 3](#_Toc15599093)

[2. XML配置 4](#_Toc15599094)

# MyBatis概述

## 简介

MyBatis 是一个持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 可以使用简单的 XML 或注解来配置和映射原生类型、接口和 Java 的 POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

## Hello World

mybatis的相关jar包就只有mybatis-xx.jar，只要将需要的版本的包置于工程的类路径即可，使用maven可以在POM中添加如下信息：

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>x.x.x</version>

</dependency>

现在准备一个数据库，和一张表Dog，然后使用mybatise实现一个符合ORM查询方法。

|  |  |
| --- | --- |
| Dog | |
| age | number |
| name | varchar |

和物理库的表对应，我们需要创建一个类Dog

class Dog {

int age;

String name;

// get set method

}

### 基于xml文件的Hello World

mybatis的核心是SqlSessionFactory，这个类的作用是获取数据库连接对象SqlSession。SqlSession用于执行数据库的增删改查语句，同时处理语句和java对象之间映射关系。

首先需要配置数据源，在mybatis-config.xml配置如下：

<configuration>

<properties resource="datasource.properties">

</properties>

<environments default="dev">

<environment id="dev">

<transactionManager type="JDBC" />

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

</environment>

</environments>

<mappers>

<mapper resource="com/ray/xml/DogMapper.xml"/>

</mappers>

</configuration>

environment中配置了数据源的url，用户名，密码等信息。mapper中指定了Dog类和dog表之间的映射。接下来需要编写DogMapper.xml:

<mapper namespace="com.ray.xml.DogMapper">

<select id="queryDog" resultType="com.ray.Dog">

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

</select>

</mapper>

DogMapper中定义了Dog查询的语句，同时定义了一个占位符ID。然后在代码中可以调用这个语句来查询dog表：

// 使用配置文件创建 SqlSessionFactory

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()

.build(Resources.getResourceAsStream(

"com/ray/xml/mybatis-config.xml"));

// 获取 session 使用 session操作数据库

try (SqlSession sess = sqlSessionFactory.openSession()) {

// 调用mapper查询

Dog d = sess.selectOne("com.ray.xml.DogMapper.queryDog", 1);

System.out.println(d.getName() + ", " + d.getAge());

}

直接在main函数中执行上述语句，就能够查询出dog表，并且会自动的将数据字段映射到Dog对象的属性。

### 基于代码配置的Hello World

如果不使用xml，我们可以在代码中配置MyBatis的相关配置：

Properties prop = loadProperties("datasource.properties");

// 创建数据源

DataSource ds = new PooledDataSource(

prop.getProperty("driver"),

prop.getProperty("url"),

prop.getProperty("username"),

prop.getProperty("password"));

// 事务管理器

TransactionFactory tsf = new JdbcTransactionFactory();

// 创建配置环境

Environment environment = new Environment("dev", tsf, ds);

// 创建配置对象，并添加映射器

Configuration configuration = new Configuration(environment);

configuration.addMapper(DogMapper.class);

我们在代码中创建的配置对象和xml中的配置是等效的，配置文件使用到了DogMapper，这是一个接口对应DogMapper.xml的配置：

interface DogMapper {

@Select("select \* from dog where id = #{id}")

public Dog selectDog(int i);

}

Select 注解指定的Dog类和Dog表的关联的查询语句。然后我们在程序中可以使用配置对象来创建SqlSessionFactory，进而创建Session来操作数据库。

SqlSessionFactory sqlSessionFactory =

new SqlSessionFactoryBuilder().build(configuration);

try (SqlSession session = sqlSessionFactory.openSession()) {

DogMapper mapper = session.getMapper(DogMapper.class);

Dog d = mapper.selectDog(1);

System.out.println(d.getName() + ", " + d.getAge());

}

这里没有使用语句来操作而是通过创建一个DogMapper的代理对象来完成ORM映射。这样基于xml和代码的基本配置就整理完毕，但是关于mybatis还有很多需要学习。

# XML配置

配置文件决定mybatis的行为，配置文件结构如下：

configuration（配置）

properties（属性）

settings（设置）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境配置）

environment（环境变量）

transactionManager（事务管理器）

dataSource（数据源）

databaseIdProvider（数据库厂商标识）

mappers（映射器）

## 属性 properties

mybatis的属性可以从属性文件定义也可以在配置文件中直接定义：

<properties resource="config/datasource.properties">

<property name="username" value="root"/>

<property name="password" value="root"/>

</properties>

包含的属性可以在配置文件中用来替换占位符 value = "${username}"，如果多个地方都定义了属性，那么先读取的属性会被后读取的覆盖。属性的加载顺序如下：

1. properties 元素体内指定的属性

2. resource 属性或根据 url 属性中定义的属性文件中的属性

3.作为方法参数传递的属性

属性可以支持默认值的设置，使用配置${username:root}可以使用默认值功能。这些功能是可以配置的：

<property name="...enable-default-value" value="true"/> <!-- 启用默认值特性 -->

<property name="...default-value-separator" value="?:"/> <!-- 修改默认值分隔符 -->

## 设置 settings

设置中定义mybatis运行时的一些行为，具体的配置和行为可以参考文档。

## 类型别名 typeAliases

类型别名是为java类型设置一个短的名字，目的是减少限定名的冗余。例如一个类 com.ray.persistence.bean.Dog，我们可以为其配置别名：

<typeAliases>

<typeAlias alias="Dog" type="com.ray.persistence.bean.Dog"/>

</typeAliases>

这样在任意地方配置时，如果需要使用com.ray…Dog 可以直接使用Dog。

## 类型处理器 typeHandlers

类型处理器的目的是告诉mybatis数据的一种类型和java的类型之间如果互相转换，mybatis自己自然定义了一些列的处理器。但是对于一些非标准的数据库类型或特殊需求可能需要其他的一些转换，可能需要自定义类型处理器来处理。要实现自定义类型处理器可以实现TypeHanler接口或者继承BaseTypehandler类。如：

@MappedJdbcTypes(JdbcType.VARCHAR)

public class RaysTypeHandler extends BaseTypeHandler<String> {

public void setNonNullParameter(PreparedStatement ps, int i, String parameter, JdbcType jdbcType) {

ps.setString(i, parameter);

}

public String getNullableResult(ResultSet rs, String columnName) {

return rs.getString(columnName);

}

public String getNullableResult(ResultSet rs, int columnIndex) {

return rs.getString(columnIndex);

}

public String getNullableResult(CallableStatement cs, int idx) {

return cs.getString(idx);

}

}

然后在配置文件中配置typeHandler

<typeHandlers>

<typeHandler handler="com.ray.example.RaysTypeHandler"/>

</typeHandlers>

处理器类上的MappedJdbcTypes告诉mybatis这个处理器用于处理何种数据库类型，而其泛型则表示这个处理器处理的java类型。这些也可以在配置文件的typeHandler中配置，也可以通过注解来配置。

## 对象工厂 objectFactory

MyBatis在从结果集中创建实体对象时，会使用一个对象工厂ObjectFactory来初始化对象。默认的对象工厂通过类的构造器或者指定的初始化方法来初始化对象。可以继承DefaultObjectFactory来定义自己的对象工厂。

## 插件

MyBatis允许在一些方法执行前拦截方法调用，这种拦截可以借助插件来完成。允许拦截的方法包括：

Executor(update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

ParameterHandler(getParameterObject, setParameters)

ResultSetHandler(handleResultSets, handleOutputParameters)

StatementHandler(prepare, parameterize, batch, update, query)

使用插件的方式是实现Interceptor接口，然后加入配置中。

@Intercepts({@Signature(

type = Executor.class,

method = "query",

args = { MappedStatement.class, Object.class,

RowBounds.class, ResultHandler.class}) })

public class RaysInterceptor implements Interceptor {

@Override

public Object intercept(Invocation invocation) throws Throwable {

String name = invocation.getMethod().getName();

System.out.println("befor interceptor " + name);

return invocation.proceed();

}

}

## 环境配置

MyBatis可以配置多个环境，但是同一个SqlSessionFactory只能使用一种环境的配置。比如生产、测试、开发等不同的环境需要不同的配置。配置环境的方式如下：

<environments default="development">

<environment id="development">

<transactionManager type="JDBC">

<property name="..." value="..."/>

</transactionManager>

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

</environment>

</environments>

然后在代码中使用需要使用某种环境，可以这样来构建SqlSessionFactory:

SqlSessionFactory factory =

SqlSessionFactoryBuilder.build(reader, environment,properties)

其中环境和属性的参数是可选的，如果环境参数确实则使用默认环境。

### 事务管理器transactionManager

MyBatis中有两种事务管理器，JDBC和MANAGED。

JDBC事务管理器直接使用JDBC的提交和回滚设置。

MANAGED事务管理器从不提交和回滚连接，而是由容器来管理事务。

### 数据源 datasource

datasource元素用来配置JDBC数据源，type属性定义数据源类型，有三种内置的数据库类型UNPOOLED，POOLED，JNDI。

UNPOOLED数据源就是简单得数据源配置，每次请求都会打开和关闭新的连接。

POOLED数据源其实就是配置一个数据源连接池，这样在配置的时候除了数据按本身的属性，如用户名、密码等还要配置连接池的一些属性，如连接池的大小、连接生存时间、连接超时等等。

JNDI数据源用于EJB服务，这里不做介绍。

可以通过实现接口DataSourceFactory或者继承实现了该接口的相应的一些类如UnpooledDataSourceFactory来为一些特殊的数据库添加数据源支持。

## 数据库厂商标识 databaseIdProvider

厂商标识配置的目的是适配不同的数据库之间的方言差异，我们在配置查询语言时可以指定语句的databaseId，这个ID和数据库的元信息中的名称一致（DatabaseMetaData#getDatabaseProductName() ），那么MyBatis就能实现对不同数据库的适配。一般来说，数据库的元信息都比较长，我们可以通过厂商标识配置将长名称配置为段名称，这样在语句中的databaseId就不至于太长。如：

<databaseIdProvider type="DB\_VENDOR">

<property name="SQL Server" value="sqlserver"/>

</databaseIdProvider>

## 映射器

映射器是MyBatis的主要功能，映射器中定义了实体的操作和对应的sql语句。

映射器配置主要是告诉MyBatis如果去寻找映射器，而具体的映射器配置内容将在后面介绍。

我们告诉MyBatis映射文件的位置：

<!-- 使用相对于类路径的资源引用 -->

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<mapper resource="org/mybatis/builder/BlogMapper.xml"/>

<mapper resource="org/mybatis/builder/PostMapper.xml"/>

</mappers>

也可以告诉MyBatis映射器对象的类名：

<!-- 使用映射器接口实现类的完全限定类名 -->

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

<mapper class="org.mybatis.builder.BlogMapper"/>

<mapper class="org.mybatis.builder.PostMapper"/>

</mappers>

或者告诉MyBatis包名让其自行搜索带注解的类：

<!-- 将包内的映射器接口实现全部注册为映射器 -->

<mappers>

<package name="org.mybatis.builder"/>

</mappers>

# XML映射文件

映射文件中配置实体和数据库之间的映射方式。下面按照配置元素来分别介绍。

## select

select元素对应数据库查询操作。例如我们数据库有一张表Dog (id number, name varchar),我们有一个dao方法需要按照id来查询dog，那么我们可以配置如下的映射：

<select id="selectDog" parameterType="int" resultType="com.ray.Dog">

SELECT \* FROM DOG WHERE ID = #{id}

</select>

在MyBatis中参数用#{}来标识，等同于jdbc的预处理语句中的 "?"。

select元素有许多属性，这些属性定义了这条语句执行前后MyBatis的行为：

resultType：语句返回的期望类型，如果返回的是集合则是集合的泛型类型。

resultMap：结果集映射，和resultType不能同时使用。

flushCache：配置为true该语句执行时会清空本地缓存和二级缓存。

useCache：本条语句的结果是否被二级缓存保存

fetchSize：尝试让数据库驱动返回的结果的行数和该值相等

statementType：定义语句对象类型，可以是STATEMENT，PREPARED，CALLABLE之一，分别对应JDBC的不同的语句对象。

resultSetType：FORWARD\_ONLY，SCROLL\_SENSITIVE, SCROLL\_INSENSITIVE 或 DEFAULT（等价于 unset） 中的一个，默认值为 unset

databaseId：厂商标识

resultOrdered：针对嵌套的select，如果为 true，就是假设包含了嵌套结果集或是分组，这样的话当返回一个主结果行的时候，就不会发生有对前面结果集的引用的情况。 这就使得在获取嵌套的结果集的时候不至于导致内存不够用。

resultSets：这个设置仅对多结果集的情况适用。它将列出语句执行后返回的结果集并给每个结果集一个名称，名称是逗号分隔的。

## insert，update和delete

数据变更语句 insert，update 和 delete 的实现非常接近：

<insert

id="insertAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

keyProperty=""

keyColumn=""

useGeneratedKeys=""

timeout="20">

<update

id="updateAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

<delete

id="deleteAuthor"

parameterType="domain.blog.Author"

flushCache="true"

statementType="PREPARED"

timeout="20">

一些相关属性如下：

useGeneratedKeys：使用jdbc生成主键

keyProperty：唯一标记一个属性，MyBatis 会通过 getGeneratedKeys 的返回值或者通过 insert 语句的 selectKey 子元素设置它的键值，默认值：未设置（unset）。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。

keyColumn：通过生成的键值设置表中的列名，这个设置仅在某些数据库（像 PostgreSQL）是必须的，当主键列不是表中的第一列的时候需要设置。如果希望使用多个生成的列，也可以设置为逗号分隔的属性名称列表。

我们可以按照如下的方式来配置语句：

<insert id="insertAuthor">

insert into Author (id,username,password,email,bio)

values (#{id},#{username},#{password},#{email},#{bio})

</insert>

<update id="updateAuthor">

update Author set

username = #{username},

password = #{password},

email = #{email},

bio = #{bio}

where id = #{id}

</update>

<delete id="deleteAuthor">

delete from Author where id = #{id}

</delete>

### 主键生成

如果数据库支持自动生成主键，可以使用如下的方式来生成：

<insert id="insertAuthor" useGeneratedKeys="true"

keyProperty="id">

insert into Author (username,password,email,bio)

values (#{username},#{password},#{email},#{bio})

</insert>

对于不支持生成主键的数据库，MyBatis也可以使用其他方式：

<insert id="insertAuthor">

<selectKey keyProperty="id" resultType="int" order="BEFORE">

select next\_val a from ID\_SEQUENCE

</selectKey>

insert into Author

(id, username, password, email,bio, favourite\_section)

values

(#{id}, #{username}, #{password}, #{email}, #{bio}, #{favouriteSection,jdbcType=VARCHAR})

</insert>

这里我们使用了序列来得到主键，然后用于数据库插入。

### 多行插入

多行插入的语句如下：

<insert id="insertAuthor" useGeneratedKeys="true"

keyProperty="id">

insert into Author (username, password, email, bio) values

<foreach item="item" collection="list" separator=",">

(#{item.username}, #{item.password}, #{item.email}, #{item.bio})

</foreach>

</insert>

### sql片段

对于一些可以重复的语句，可以定义代码片段，来供其他的语句定义使用：

<sql id="cols">${alias}.id,${alias}.username,${alias}.password </sql>

<select id="selectUsers" resultType="map">

select

<include refid="cols">

<property name="alias" value="t1"/>

</include> ,

<include refid="cols">

<property name="alias" value="t2"/>

</include>

from some\_table t1

cross join some\_table t2

</select>

属性值也可以被用在 include 元素的 refid 属性里或 include 元素的内部语句中，例如：

<sql id="sometable">

${prefix}Table

</sql>

<sql id="someinclude">

from

<include refid="${include\_target}"/>

</sql>

<select id="select" resultType="map">

select

field1, field2, field3

<include refid="someinclude">

<property name="prefix" value="Some"/>

<property name="include\_target" value="sometable"/>

</include>

</select>

## 参数

MyBatis支持传入复杂类型的参数，然后框架会根据占位符的名称去需要对应属性的值填充到语句中，如之前的Dog类，有 id 和 name 属性，我们可以这样编写insert语句

<insert id="insertDog" parameterType="com.ray.Dog">

insert into dog (id, name) values (#{id}, #{name})

</insert>

可以为参数指定数据类型：

#{id, javaType=int, jdbcType=Number}

还有一些其他的属性：如typeHandler可以指定类型处理器，numericScale可以指定数值类型的精度，mode可以指定IN,OUT和INOUT参数。

## 结果映射

当数据库的表的字段名和javaBean的属性名一致时，MyBatis可以自动完成结果映射，即将数据库查询结果直接映射到java对象中，如果字段名和java类的属性不完全一致如 Dog 类(id, name) Dog 表是 Dog(DOG\_ID, DOG\_NAME)时，可以使用列别名来辅助映射：

<select id="selectDog" resultType="Dog">

select

dog\_id as "id",

dog\_name as "userName"

from dog

where dog\_id = #{id}

</select>

除此之外，也可以使用resultMap

<resultMap id="dogResultMap" type="Dog">

<id property="id" column="dog\_id" />

<result property="name" column="dog\_name"/>

</resultMap>

而在引用它的语句中使用 resultMap 属性就行了（注意我们去掉了 resultType 属性）。比如:

<select id="selectDog" resultMap="dogResultMap">

select dog\_id, dog\_name from dog where dog\_id = #{id}

</select>

### 高级结果映射

一片博客，有一位作者，每篇博文有多个评论（也可能没有）和标签，这时候查询一条博文，需要关联许多表。



<select id="" resultMap="blogResultMap" >

select B.id as blog\_id,

B.author\_id as blog\_title,

B.aushot\_id as author\_id,

A.author\_name as author\_name,

P.id as post\_id,

P.author\_id as post\_author\_id,

P.body as post\_body,

C.post\_id as comment\_id,

C.comment as comment\_text,

T.id as tag\_id,

T.name as tag\_name

from Blog B

left outer join Author A on B.author\_id = A.id

left outer join Post P on B.id = P.Blog\_id

left outer join Comment C on P.id = C.cost\_id

left outer join Post\_tag PT on PT.post\_id = P.id

left outer join Tag T on PT.tag\_id = T.id

where B.id = #{id}

</select>

映射文件如下：

<!-- 非常复杂的结果映射 -->

<resultMap id="blogResultMap" type="Blog">

<constructor><idArg column="blog\_id" javaType="int"/></constructor>

<result property="title" column="blog\_title"/>

<association property="author" javaType="Author">

<id property="id" column="author\_id"/>

<result property="username" column="author\_name"/>

</association>

<collection property="posts" ofType="Post">

<id property="id" column="post\_id"/>

<result property="subject" column="post\_subject"/>

<association property="author" javaType="Author"/>

<collection property="comments" ofType="Comment">

<id property="id" column="comment\_id"/>

</collection>

<collection property="tags" ofType="Tag" >

<id property="id" column="tag\_id"/>

</collection>

<discriminator javaType="int" column="draft">

<case value="1" resultType="DraftPost"/>

</discriminator>

</collection>

</resultMap>