# MyBatis介绍

MyBatis 的前身是IBatis，是一款优秀的支持自定义SQL查询、存储过程和高级映射的持久层框架，消除了几乎所有的JDBC代码和参数的手动设置以及结果集的检索。MyBatis可以使用XML或注解进行配置和映射，MyBatis通过将参数映射到配置的SQL形成最终执行的SQL语句，最后将执行SQL的结果映射成Java对象返回。

与其他的ORM（对象关系映射）框架不同，MyBatis并没有将Java对象与数据库表关联起来，而是将Java方法与SQL语句关联。MyBatis允许用户充分利用数据库的各种功能，例如存储过程、视图、各种复杂的查询以及某数据库的专有特性。

与JDBC相比，MyBatis简化了相关代码，SQL语句在一行代码中就能执行。MyBatis提供了一个映射引擎，声明式地将SQL语句的执行结果与对象树映射起来。通过使用一种內建的类XML表达式语言，SQL语句可以被动态生成。

# 搭建MyBatis框架

## 环境搭建

在搭建MyBatis框架之前需要下载MyBatis框架所需要的jar包（http://mvnrepository.com/artifact/org.mybatis）。并将jar包导入项目的lib文件夹中。将jar包进行构建路径（Build Path--->Add to Build Path）。

1. 准备数据库

首先创建一个数据库，编码方式设为UTF-8。并创建所需的表和数据。

1. 配置MyBatis

配置MyBatis有很多种方式，这里先用最常用的XML形式进行配置。

在项目中新建一个XML文件（mybatisConfig.xml）

<?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>

<!-- 定义JavaBean所在的包 -->

<typeAliases>

<package name=*"com.uc.pojo"*/>

</typeAliases>

<!-- 定义数据库连接参数 -->

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*></transactionManager>

<dataSource type=*"UNPOOLED"*>

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/javaee?characterEncoding=utf-8"*/>

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

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

</dataSource>

</environment>

</environments>

<!-- 定义需要映射的Mapper文件 -->

<mappers>

<mapper resource=*"mapping/UserMapper.xml"*/>

</mappers>

</configuration>

1. 创建实体类（JavaBean）和Mapper.xml文件

MyBatis是一个结果映射框架，这里创建的实体类实际上是一个数据值对象，在实际应用中，一个表一般会对应一个实体，用于INSERT、UPDATE、DELETE和简单的SELECT操作。（实体类中的属性名与数据库表中的字段名一致）。

**public** **class** User {

**private** **int** id;

**private** String name;

**private** String pwd;

**public** User() {}

**public** User(String name, String pwd) {

**this**.name = name;

**this**.pwd = pwd;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getPwd() {

**return** pwd;

}

**public** **void** setPwd(String pwd) {

**this**.pwd = pwd;

}

}

实体类建好后，需要给实体类新建对应的Mapper.xml文件（该文件用于配置操作表的具体SQL语句等）。

UserMapper.xml

<?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=*"UserMapper"* >

<select id=*"getUserById"* resultType=*"User"*>

select id,name,pwd from t\_user

</select>

</mapper>

1. 编写测试类

写测试类除了通常的使用main方法来运行测试结果，还可以使用JUnit单元测试和Log4j日志结合一起使用进行测试。

**public** **static** **void** main(String[] args) {

**try** {

Reader reader = Resources.*getResourceAsReader*("mybatisConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(reader);

SqlSession sqlSession = sqlSessionFactory.openSession();

List<User> userList = sqlSession.selectList("UserMapper.getUserById");

**for** (User user : userList) {

System.***out***.println(user);

}

reader.close();

sqlSession.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

## 配置说明

### MyBatis配置文件

完整的MyBatis配置文件

<?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">

<!-- 配置文件的根元素 -->

<configuration>

<!-- 属性：定义配置外在化 -->

<properties></properties>

<!-- 设置：定义mybatis的一些全局性设置 -->

<settings>

<!-- 具体的参数名和参数值 -->

<setting name=*""* value=*""*/>

</settings>

<!-- 类型名称：为一些类定义别名 -->

<typeAliases></typeAliases>

<!-- 类型处理器：定义Java类型与数据库中的数据类型之间的转换关系 -->

<typeHandlers></typeHandlers>

<!-- 对象工厂 -->

<objectFactory type=*""*></objectFactory>

<!-- 插件：mybatis的插件,插件可以修改mybatis的内部运行规则 -->

<plugins>

<plugin interceptor=*""*></plugin>

</plugins>

<!-- 环境：配置mybatis的环境 -->

<environments default=*""*>

<!-- 环境变量：可以配置多个环境变量，比如使用多数据源时，就需要配置多个环境变量 -->

<environment id=*""*>

<!-- 事务管理器 -->

<transactionManager type=*""*></transactionManager>

<!-- 数据源 -->

<dataSource type=*""*></dataSource>

</environment>

</environments>

<!-- 数据库厂商标识 -->

<databaseIdProvider type=*""*></databaseIdProvider>

<!-- 映射器：指定映射文件或者映射类 -->

<mappers></mappers>

</configuration>

#### 属性<properties>

properties元素主要是用来定义配置外在化，比如数据库的连接属性等。这些属性都是可外部配置且可动态替换的，既可以在典型的Java配置文件中配置，亦可以通过properties元素的子元素property来传递。resource属性的值为外部配置文件的路径（用于将外部配置文件的值引入配置中）。

例如：

<properties resource=*" jdbc.properties"*>

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

<property name=*"password"* value=*"123456"*/>

</properties>

其中的属性就可以在整个配置文件中使用来替换需要动态配置的属性值。比如在数据源中使用的例子：

<dataSource type=*"POOLED"*>

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

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

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

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

</dataSource>

这个例子中的username和password将会由properties元素中设置的相应值来替换。driver和url属性将会由jdbc.properties文件中对应的值来替换。这样就为配置提供了诸多灵活选择。

其中dataSource元素中子元素property的value属性值可以是直接的数据库连接参数值（这样就不需要定义properties元素了），也可以是如上的${url}的值。${url}值则用做与properties元素中定义的参数值进行映射，或者是与外在的Java配置文件（jdbc.properties）中的属性值作映射。花括弧中的值为properties元素中的name属性的值或者是外在配置文件中的key。

除了上述两种方式外属性也可以通过将Properties对象传递到SqlSessionBuilder.build()方法中

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(*reader*, props);

// ... or ...

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(*reader*, environment, props);

但是，这也就涉及到了优先级的问题，如果属性不只在一个地方配置，那么mybatis将会按照下面的顺序来加载：

1. 在properties元素体内指定的属性首先被读取。
2. 然后根据properties元素中的resource属性读取类路径下属性文件或根据url属性指定的路径读取属性文件，并覆盖已读取的同名属性。
3. 最后读取作为方法参数传递的属性，并覆盖已读取的同名属性。

#### 设置<settings>

settings中的子元素setting指定MyBatis的一些全局配置属性，这是MyBatis中极为重要的调整设置，它们会改变MyBatis的运行时行为，所以我们需要清楚的知道这些属性的作用及默认值。

|  |  |  |  |
| --- | --- | --- | --- |
| 设置参数 | 描述 | 有效值 | 默认值 |
| cacheEnabled | 该配置影响的所有映射器中配置的缓存的全局开关 | true | false | true |
| 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。特定关联关系中可通过fetchType属性来覆盖该项的开关状态 | true | false | false |
| aggressiveLazyLoading | 当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之每种属性将会按需加载 | true | false | true |
| multipleResultSetsEnabled | 是否允许单一语句返回多结果集（需要兼容驱动）。 | true | false | true |
| useColumnLabel | 使用列标签代替列名。不同的驱动在这方面会有不同的表现， 具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。 | true | false | true |
| useGeneratedKeys | 允许 JDBC 支持自动生成主键，需要驱动兼容。 如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。 | true | false | false |
| autoMappingBehavior | 指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论是否嵌套）。 | NONE, PARTIAL, FULL | PARTIAL |
| defaultExecutorType | 配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。 | SIMPLE、REUSE、BATCH | SIMPLE |
| defaultStatementTimeout | 设置超时时间，它决定驱动等待数据库响应的秒数。 | 任何正整数 | Not Set (null) |
| defaultFetchSize | 设置驱动程序提示，以控制返回结果的读取大小。 这个参数值可以被查询设置覆盖。 | 任何正整数 | Not Set (null) |
| safeRowBoundsEnabled | 允许在嵌套语句中使用分页（RowBounds）。 | true | false | false |
| mapUnderscoreToCamelCase | 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射 | true | false | false |
| localCacheScope | MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。 默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。 | SESSION | STATEMENT | SESSION |
| jdbcTypeForNull | 当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。 | JdbcType枚举。 最常见的是：NULL，VARCHAR和OTHER | OTHER |
| lazyLoadTriggerMethods | 指定哪个对象的方法触发一次延迟加载。 | 用逗号分隔的方法名称列表 | equals,clone hashCode |
| defaultScriptingLanguage | 指定动态 SQL 生成的默认语言。 | 类型别名或完全限定的类名称。 | org.apache.ibatis. scripting |
| callSettersOnNulls | 指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有 Map.keySet() 依赖或 null 值初始化的时候是有用的。注意基本类型（int、boolean等）是不能设置成 null 的。 | true | false | false |
| logPrefix | 指定 MyBatis 增加到日志名称的前缀。 | 任何字符串 | Not set |
| logImpl | 指定 MyBatis 所用日志的具体实现，未指定时将自动查找。 | SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING | Not set |
| proxyFactory | 指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。 | CGLIB | JAVASSIST | JAVASSIST (MyBatis 3.3 or above) |

一个完整的settings元素示例如下：

<settings>

<setting name=*"cacheEnabled"* value=*"true"*/>

<setting name=*"lazyLoadingEnabled"* value=*"true"*/>

<setting name=*"multipleResultSetsEnabled"* value=*"true"*/>

<setting name=*"useColumnLabel"* value=*"true"*/>

<setting name=*"useGeneratedKeys"* value=*"false"*/>

<setting name=*"autoMappingBehavior"* value=*"PARTIAL"*/>

<setting name=*"defaultExecutorType"* value=*"SIMPLE"*/>

<setting name=*"defaultStatementTimeout"* value=*"25"*/>

<setting name=*"defaultFetchSize"* value=*"100"*/>

<setting name=*"safeRowBoundsEnabled"* value=*"false"*/>

<setting name=*"mapUnderscoreToCamelCase"* value=*"false"*/>

<setting name=*"localCacheScope"* value=*"SESSION"*/>

<setting name=*"jdbcTypeForNull"* value=*"OTHER"*/>

<setting name=*"lazyLoadTriggerMethods"* value=*"equals,clone,hashCode,toString"*/>

</settings>

#### 类型别名 <typeAliases>

类型别名是为Java类设置一个短的名字。它只和Mapper.xml配置文件有关，存在的意义仅在于用来减少类完全限定名的冗余，例如：

<typeAliases>

<typeAlias alias=*"Author"* type=*"domain.blog.Author"*/>

<typeAlias alias=*"Blog"* type=*"domain.blog.Blog"*/>

<typeAlias alias=*"Comment"* type=*"domain.blog.Comment"*/>

<typeAlias alias=*"Post"* type=*"domain.blog.Post"*/>

<typeAlias alias=*"Section"* type=*"domain.blog.Section"*/>

<typeAlias alias=*"Tag"* type=*"domain.blog.Tag"*/>

</typeAliases>

当这样配置时，Blog可以用在任何使用domain.blog.Blog的地方。

也可以指定一个包名，MyBatis会在包名下搜索需要的JavaBean，比如：

<typeAliases>

<package name=*"domain.blog"*/>

</typeAliases>

每一个在包domain.blog中的JavaBean，在没有注解的情况下，会使用Bean的首字母小写的非限类名来作为它的别名。比如domain.blog.Author的别名为author；若有注解，则别名为注解值。看下面的例子：

@Alias("author")

**public** **class** Author {

...

}

已经为许多常见的Java类型内建了相应的类型别名。它们都是大小写不敏感的，需要注意的是有基本类型名称重复导致的特殊处理。

|  |  |
| --- | --- |
| 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 |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

#### 类型处理器<typeHandlers>

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

|  |  |  |
| --- | --- | --- |
| 类型处理器 | Java 类型 | JDBC 类型 |
| BooleanTypeHandler | java.lang.Boolean, boolean | 数据库兼容的 BOOLEAN |
| ByteTypeHandler | java.lang.Byte, byte | 数据库兼容的 NUMERIC 或 BYTE |
| ShortTypeHandler | java.lang.Short, short | 数据库兼容的 NUMERIC 或 SHORT INTEGER |
| IntegerTypeHandler | java.lang.Integer, int | 数据库兼容的 NUMERIC 或 INTEGER |
| LongTypeHandler | java.lang.Long, long | 数据库兼容的 NUMERIC 或 LONG INTEGER |
| FloatTypeHandler | java.lang.Float, float | 数据库兼容的 NUMERIC 或 FLOAT |
| DoubleTypeHandler | java.lang.Double, double | 数据库兼容的 NUMERIC 或 DOUBLE |
| BigDecimalTypeHandler | java.math.BigDecimal | 数据库兼容的 NUMERIC 或 DECIMAL |
| StringTypeHandler | java.lang.String | CHAR, VARCHAR |
| ClobTypeHandler | java.lang.String | CLOB, LONGVARCHAR |
| NStringTypeHandler | java.lang.String | NVARCHAR, NCHAR |
| NClobTypeHandler | java.lang.String | NCLOB |
| ByteArrayTypeHandler | byte[] | 数据库兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB, LONGVARBINARY |
| DateTypeHandler | java.util.Date | TIMESTAMP |
| DateOnlyTypeHandler | java.util.Date | DATE |
| TimeOnlyTypeHandler | java.util.Date | TIME |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP |
| SqlDateTypeHandler | java.sql.Date | DATE |
| SqlTimeTypeHandler | java.sql.Time | TIME |
| ObjectTypeHandler | Any | OTHER 或未指定类型 |
| EnumTypeHandler | Enumeration Type | VARCHAR-任何兼容的字符串类型，存储枚举的名称（而不是索引） |
| EnumOrdinalTypeHandler | Enumeration Type | 任何兼容的 NUMERIC 或 DOUBLE 类型，存储枚举的索引（而不是名称）。 |

可以重写类型处理器或创建自己的类型处理器来处理不支持的或非标准的类型。具体的做法为：实现org.apache.ibatis.type.TypeHandler接口，或继承一个很便利的类org.apache.ibatis.type.BaseTypeHandler,然后可以选择性地将它映射到一个JDBC类型。比如：

@MappedJdbcTypes(JdbcType.VARCHAR)

**public** **class** ExampleTypeHandler **extends** BaseTypeHandler<String>{

@Override

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

ps.setString(i, parameter);

}

@Override

**public** String getNullableResult(ResultSet rs, String columnName) **throws** SQLException {

**return** rs.getString(columnName);

}

@Override

**public** String getNullableResult(ResultSet rs, **int** columnIndex) **throws** SQLException {

**return** rs.getString(columnIndex);

}

@Override

**public** String getNullableResult(CallableStatement cs, **int** columnIndex) **throws** SQLException {

**return** cs.getString(columnIndex);

}

}

并且还需要在配置文件里面加上：

<!-- mybatis-config.xml -->

<typeHandlers>

<typeHandler handler=*"org.mybatis.example.ExampleTypeHandler"*/>

</typeHandlers>

使用这个的类型处理器将会覆盖已经存在的处理Java的String类型属性和VARCHAR参数及结果的类型处理器。要注意MyBatis不会窥探数据库元信息来决定使用哪种类型，所以必须在参数和结果映射中指明是VARCHAR类型字段，以使其能绑定到正确的类型处理器上。这是因为，MyBatis直到语句被执行才清楚数据类型。

通过类型处理器的泛型，MyBatis可以得知该类型处理器的Java类型，不过这种行为可以通过两种方法改变：

1. 在类型处理器的元素（typeHandler element）上增加一个javaType属性（比如，javaType="String"）;
2. 在类型处理器的类上（TypeHandler class）增加一个@MappedTypes注解来指定与其关联的Java类型列表。如果在javaType属性中也同时制定，则注解方式将被忽略。

可以通过两种方式来指定被关联的JDBC类型：

1. 在类型处理器的配置元素上增加一个javaType属性（比如：javaType="VARCHAR"）;
2. 在类型处理器的类上（TypeHandler class）增加一个@MappedJdbcTypes注解来指定与其关联的JDBC类型列表。如果在javaType属性中也同时指定，则注解方式将被忽略。

最后，还可以让MyBatis查找类型处理器：

<!-- mybatis-config.xml -->

<typeHandlers>

<package name=*"org.mybatis.example"*/>

</typeHandlers>

注意在使用自动检索（autodiscovery）功能的时候，只能通过注解的方式来指定JDBC类型。

你能创建一个泛型类型处理器，它可以处理多于一个类。为达到此目的，需要增加一个接收该类作为参数的构造器，这样在构造一个类型处理器的时候MyBatis就会传入一个具体的类。

//GenericTypeHandler.java

**public** **class** GenericTypeHandler<E **extends** MyObject> **extends** BaseTypeHandler<E> {

**private** Class<E> type;

**public** GenericTypeHandler(Class<E> type){

**if** (type == **null**) **throw** **new** IllegalArgumentException("Type argument cannot be null");

**this**.type = type;

}

...

}

EnumTypeHandler和EnumOrdinalTypeHandler都是泛型处理器（generic TypeHandlers），接下来的部分详细探讨。

**处理枚举类型**

若想映射枚举类型Enum，则需要从EnumTypeHandler或者EnumOrdinalTypeHandler中选一个来使用

比如说我们想存储近似值时用到的舍入模式。默认情况下，MyBatis会利用EnumTypeHandler来把Enum值转换成对应的名字。

注意EnumTypeHandler在某种意义上来说是比较特别的，其他的处理器只针对某个特定的类，而它不同，它会处理任意继承了Enum的类。

不过，我们可能不想存储名字，相反我们的DBA会坚持使用整形值代码。那也一样轻而易举；在配置文件中把EnumOrdinalTypeHandler加到typeHandlers中即可，这样每个RoundingMode将通过他们的序数值来映射成对应的整形。

<!-- mybatis-config.xml -->

<typeHandlers>

<typeHandler handler=*"org.apache.ibatis.type.EnumOrdinalTypeHandler"* javaType=*"java.math.RoundingMode"*/>

</typeHandlers>

但是怎么样能将同样的Enum既映射成字符串又映射成整形呢？

自动映射器（auto-mapper）会自动选用EnumOrdinalTypeHandler来处理，所以如果我们想用普通的EnumTypeHandler，就非要为那些SQL语句显示地设置要用到的类型处理器不可。

<mapper namespace=*"org.apache.ibatis.submitted.rounding.Mapper"*>

<resultMap type=*"org.apache.ibatis.submitted.rounding.User"* id=*"usermap"*>

<id column=*"id"* property=*"id"*/>

<result column=*"name"* property=*"name"*/>

<result column=*"funkyNumber"* property=*"funkyNumber"*/>

<result column=*"roundingMode"* property=*"roundingMode"*/>

</resultMap>

<select id=*"getUser"* resultMap=*"usermap"*>

select \* from users

</select>

<insert id=*"insert"*> insert into users (id, name, funkyNumber, roundingMode) values

( #{id}, #{name}, #{funkyNumber},

#{roundingMode})

</insert>

<resultMap type=*"org.apache.ibatis.submitted.rounding.User"* id=*"usermap2"*>

<id column=*"id"* property=*"id"*/>

<result column=*"name"* property=*"name"*/>

<result column=*"funkyNumber"* property=*"funkyNumber"*/>

<result column=*"roundingMode"* property=*"roundingMode"* typeHandler=*"org.apache.ibatis.type.EnumTypeHandler"*/>

</resultMap>

<select id=*"getUser2"* resultMap=*"usermap2"*>

select \* from users2

</select>

<insert id=*"insert2"*> insert into users2 (id, name, funkyNumber, roundingMode) values(

#{id}, #{name}, #{funkyNumber}, #{roundingMode, typeHandler=

org.apache.ibatis.type.EnumTypeHandler})

</insert>

</mapper>

注意，这里的select语句强制使用resultMap来代替resultType。

#### 对象工厂（<objectFactory>）

MyBatis每次创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成。默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。如果想覆盖对象工厂的行为，则可以通过创建自己的对象工厂来实现，比如：

**public** **class** ExampleObjectFactory **extends** DefaultObjectFactory {

**public** Object create(Class type){

**return** **super**.create(type);

}

**public** Object create(Class type, List<Class> constructorArgTypes, List<Object> constructorArgs) {

**return** **super**.create(type, constructorArgTypes, constructorArgs);

}

**public** **void** setProperties(Properties properties) {

**super**.setProperties(properties);

}

**public** <T> **boolean** isCollection(Class<T> type){

**return** Collection.**class**.isAssignableFrom(type);

}

}

<!-- mybatis-config.xml -->

<objectFactory type=*"org.mybatis.example.ExampleObjectFactory"*>

<property name=*"someProperty"* value=*"100"*/>

</objectFactory>

ObjectFactory接口很简单，它包含两个创建用的方法，一个是处理默认构造方法的，另外一个是处理带参数的构造方法。最后setProperties方法可以被用来配置ObjectFactory,初始化你的ObjectFactory实例后，objectFactory元素体内定义的属性会被传递给setProperties方法。

#### 插件（<plugins>）

1. MyBatis允许你在已映射的语句执行过程中的某一点进行拦截调用。默认情况下，Mybatis允许使用插件来拦截的方法调用包括：
2. Executor(update,query,flushStatements,commit,rollback,getTransaction,close,isClosed)
3. ParameterHandler(getParameterObejct,setParameters)
4. ResultSetHandler(handlerResultSets,handlerOutputParameters)
5. StatementHandler(prepare,parameterize,batch,update,query)

这些类中方法的细节可以通过查看每个方法的签名来发现，或者直接查看MyBatis的发行包中的源代码。假设你想做的不仅仅是方法的调用，那么你应该很好的了解正在重写的方法的行为。因为如果在视图修改或重写已有方法的行为的时候，你很有可能在破坏MyBatis的核心模块。这些都是更低层的类和方法，所以使用插件的时候要特别担心。

通过MyBatis提供强大的机制，使用插件是非常简单的，只需要实现Interceptor接口，并指定想要拦截的方法签名即可。

@Intercepts({@Signature(

type= Executor.**class**,

method = "update",

args = {MappedStatement.**class**,Object.**class**}

)})

**public** **class** ExamplePlugin **implements** Interceptor{

**public** Object intercept(Invocation invocation) **throws** Throwable{

**return** invocation.proceed();

}

**public** Object plugin(Object target){

**return** Plugin.wrap(target, **this**);

}

**public** **void** setProperties(Properties properties){}

}

<!-- mybatis-config.xml -->

<plugins>

<plugin interceptor=*"org.mybatis.example.ExamplePlugin"*>

<property name=*"someProperty"* value=*"100"*/>

</plugin>

</plugins>

上面的插件将会拦截Executor实例中所有的“update”方法调用，这里的Executor是负责执行底层映射语句的内部对象。

覆盖配置类除了用插件来修改MyBatis核心行为之外，还可以通过完全覆盖配置类来达到目的。只需继承后覆盖其中的每个方法，再把它传递到sqlSessionFactoryBuilder.build(myConfig)方法即可。再次重申，这可能会严重影响Mybatis的行为，务请慎之又慎！

#### 环境（<environments>）

##### 环境变量<environment>

MyBatis可以配置成适应多种环境，这种机制有助于将sql映射应用于多种数据库中，现实情况下有多种理由需要这么做。例如，开发、测试和生产环境需要有不同的配置；或者共享相同的Schema的多个生产数据库，想使用相同的sql映射。许多类似的用例。

尽管可以配置多个环境，但是每个SqlSessionFactory实例只能选择其一。

所以，如果想连接两个数据库，就需要创建两个SqlSessionFactory实例，每个数据库对应一个。而如果是三个数据库，就需要三个实例，依此类推。

**每个数据库对应一个SqlSessionFactory实例**

为了指定创建哪种环境，只要将它作为可选参数传递给SqlSessionFactoryBuilder即可。可以接受环境配置的两个方法签名是：

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(*reader*, environment);

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(*reader*, environment, properties);

如果忽略了环境参数，那么默认环境将会被加载，如下所示：

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(*reader*);

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(*reader*,properties);

环境元素定义了如何配置环境

<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>

注意这里的关键点：

1. 默认环境的ID（比如：default="development"）
2. 每个environment元素定义的环境ID（比如：id="development"）
3. 事务管理器的配置（比如：type="JDBC"）
4. 数据源的配置（比如：type="POOLED"）

默认的环境和环境ID是一目了然的。随你怎么命名，只要保证默认环境要匹配其中一个环境ID

##### 事务管理器（transactionManager）

在MyBatis中有两种类型的事务管理器（也就是 type="[JDBC|MANAGED]"）

1. JDBC --- 这个配置就是直接使用了JDBC的提交和回滚设置，它依赖于从数据源得到的连接来管理事务范围。
2. MANAGED --- 这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如JEE应用服务器上下文）。默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将closeConnection属性设置为false来阻止它默认的行为。例如

<transactionManager type=*"MANAGED"*>

<property name=*"closeConnection"* value=*"false"*/>

</transactionManager>

如果正在使用Spring+MyBatis，则没有必要配置事务管理器，因为Spring模块会使用自带的管理器来覆盖前面的配置。

这两种事务管理器类型都不需要任何属性。它们只不过是类型别名，换句话说，你可以使用TransactionFactory接口的实现类的完全限定名或类型别名替代它们。

**public** **interface** TransactionFactory{

**void** setProperties(Properties props);

Transaction newTransaction(Connection conn);

Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level, **boolean** autoCommit);

}

任何在xml中配置的属性在实例化之后将会被传递给setProperties方法。你也需要创建一个Transaction接口的实现类，这个接口也很简单。

**public** **interface** Transaction{

Connection getConnection() **throws** SQLException;

**void** commit() **throws** SQLException;

**void** rollback() **throws** SQLException;

**void** close() **throws** SQLException;

使用这两个接口，完全可以自定义MyBatis对事务的处理。

##### 数据源（dataSource）

dataSource元素使用了标准的JDBC数据源接口来配置JDBC连接对象的资源。

1. 许多MyBatis的应用程序将会按示例中的例子来配置数据源。然而它并不是必须的。要知道为了方便使用延迟加载，数据源才是必须的。

有三种内建的数据源类型（也就是 type="[]UNPOOLED|POOLED|JNDI]"）；

**UNPOOLED** --- 这个数据源的实现只是被请求时打开和关闭连接。虽然有一点慢，它对在及时可用连接方面没有性能要求的简单应用是一个很好的选择。不同的数据库在这方面表现也是不一样的，所以对某些数据库来说使用连接池并不重要，这个配置也是理想。UNPOOLED类型的数据源仅仅需要配置以下5种属性：

1. driver -- 这是JDBC驱动的Java类的完全限定名（并不是JDBC驱动中可能包含的数据源类）
2. url -- 这是数据库的JDBC URL 地址。
3. username -- 登录数据库的用户名。
4. password -- 登录数据库的密码。
5. defaultTransactionIsolationLevel -- 默认的连接事务隔离级别。

作为可选项，可以传递属性给数据库驱动。要这样做，属性的前缀为"driver."，例如：

1. driver.encoding=UTF-8

这将通过DriverManager,getConnection(url,driverProperties)方法传递值为UTF-8的encoding属性给数据库驱动。

**POOLED** --- 这种数据源的实现利用“池”的概念将JDBC连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。这是一种使得并发web应用快速响应请求的流行处理方式。

除了上述提到UNPOOLED下的属性外，会有更多属性用来配置POOLED的数据源：

1. poolMaximumActiveConnections -- 在任意时间可以存在的活动（也就是正在使用）连接数量，默认值10
2. poolMaximumIdleConnections -- 任意时间可能存在的空闲连接数。
3. poolMaximumCheckoutTime -- 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000毫秒（即20秒）
4. poolTimeToWait -- 这是一个底层设置，如果获取连接花费的相当长的时间，它会给连接池打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直安静的失败），默认值20000毫秒（即20秒）。
5. poolPingQuery -- 发送到数据库的侦测查询，用来检验连接是否处在正常的工作秩序中，并且准备接受请求。默认是"NOT PING QUERY SET"，这会导致多数数据库连接失败时带有一个恰当的错误信息。
6. poolPingEnabled -- 是否启用侦测。若开启，也必须使用一个可执行的SQL语句设置poolPingQuery属性（最好是一个非常快的SQL）,默认值：false。
7. poolPingConnectionsNotUsedFor -- 配置poolPingQuery使用的频度。这可以被设置成匹配具体的数据库连接超时时间，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 -- 当然仅当 poolPingEnabled为true时适用）。

**JNDI** -- 这个数据源的实现是为了能在如EJB或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个JNDI上下文的引用。这种数据源配置只要两个属性：

1. initial\_context -- 这个属性用来在InitialContext中寻找上下文（即，initialContext.lookup(initial\_context)）。这是个可选属性，如果忽略，那么data\_source属性将会直接从InitialContext中寻找。
2. data\_source -- 这是引用数据源实例位置的上下文的路径。提供了 initial\_context配置时会在其返回的上下文中进行查找，没有提供时则直接在InitialContext中查找。

和其他数据源配置类似，可以通过添加前缀"env."直接把属性传递给初始上下文。比如：

1. env.encoding=UTF-8

这会在初始上下文（InitialContext）实例化时往它的构造方法传递值为UTF-8的encoding属性。

通过需要实现接口 org.apache.ibatis.datasource.DataSourceFactory，也可使用任何第三方数据源，：

**public** **interface** DataSourceFactory{

**void** setProperties(Properties props);

DataSource getDataSource();

org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory 可被用作父类来构建新的数据源适配器，比如下面这段插入C3P0所必需的代码：

**import** org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory;

**import** com.mchange.v2.c3p0.ComboPooledDataSource;

**public** **class** C3P0DataSourceFactory **extends** UnpooledDataSourceFactory {

**public** C3P0DataSourceFactory(){

**this**.dataSource = **new** ComboPooledDataSource();

}

}

为了令其工作，为每个需要MyBatis调用的setter方法中增加一个属性。下面是一个可以连接到PostgreSQL数据库的例子：

<dataSource type=*"org.myproject.C3P0DataSourceFactory"*>

<property name=*"driver"* value=*"org.postgresql.Driver"*/>

<property name=*"url"* value=*"jdbc:postgresql:mydb"*/>

<property name=*"username"* value=*"postgres"*/>

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

</dataSource>

#### 数据库厂商标识<databaseIdProvider>

MyBatis可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的databaseId属性。MyBatis会加载不带databaseId属性和带有匹配当前数据库databaseId属性的所有语句。如果同时找到带有databaseId和不带databaseId的相同语句，则后者被舍弃。为支持多厂商特性，只要像下面这样在mybatis-config.xml文件中加入databaseIdProvider即可：

<databaseIdProvider type=*"DB\_VENDOR"* />

这里的DB\_VENDOR会通过DatabaseMetaData#getDatabaseProductName()返回的字符串进行设置。由于通常情况下这个字符串都非常长而且相同产品的不同版本会返回不同的值，所以最好通过设置属性别名来使其变短，如下：

<databaseIdProvider type=*"DB\_VENDOR"*>

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

<property name=*"DB2"* value=*"db2"*/>

<property name=*"Oracle"* value=*"oracle"* />

</databaseIdProvider>

在有properties时，DB\_VENDOR databaseIdProvider的将被设置为第一个能匹配数据库产品名称的属性键值对应的值，如果没有匹配的属性将会设置为”null“。在这个例子中，如果getDatabaseProductName()返回”Oracle(DataDirect)“，databaseId将被设置为"oracle"。

可以通过实现接口org.apache.ibatis.mapping.DatabaseIdProvider并在mybatis-config.xml中注册来构建自己的DatabaseIdProvider：

**public** **interface** DatabaseIdProvider{

**void** setProperties(Properties p);

String getDatabaseId(DataSource dataSource) **throws** SQLException;

}

#### 映射器（<mappers>）

既然MyBatis的行为已经由上述元素配置完了，现在就要定义SQL映射语句了。但是首先需要告诉MyBatis到哪里去找到这些语句。Java在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉MyBatis到哪里去找映射文件。可以使用相对于类路径的资源引用、或完全限定资源定位符（包括file:///的URL），或类名和包名等等。例如：

<!-- Using classpath relative resources -->

<mappers>

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

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

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

</mappers>

<!-- Using url fully qualified paths -->

<mappers>

<mapper url=*"file:///var/mappers/AuthorMapper.xml"*/>

<mapper url=*"file:///var/mappers/BlogMapper.xml"*/>

<mapper url=*"file:///var/mappers/PostMapper.xml"*/>

</mappers>

<!-- Using mapper interface classes -->

<mappers>

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

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

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

</mappers>

<!-- Register all interfaces in a package as mappers -->

<mappers>

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

</mappers>

这些配置会告诉了MyBatis去哪里找映射文件，剩下的细节就应该是每个SQL映射文件了。

### Mapper配置文件

MyBatis 真正的力量是在映射语句中。这里是奇迹发生的地方。对于所有的力量,SQL 映射的 XML 文件是相当的简单。当然如果你将它们和对等功能的 JDBC 代码来比较,你会发现映射文件节省了大约 95%的代码量。MyBatis 的构建就是聚焦于 SQL 的,使其远离于普通的方式。

SQL 映射文件的根元素为<mapper>，它有很少的顶级子元素(按照它们应该被定义的顺序):

1. <cache> – 配置给定命名空间的缓存。
2. <cache-ref> – 从其他命名空间引用缓存配置。
3. <resultMap> – 最复杂,也是最有力量的元素,用来描述如何从数据库结果集中来加载你的对象。
4. <parameterMap> – 已经被废弃了!老式风格的参数映射。内联参数是首选,这个元素可能在将来被移除。这里不会记录。
5. <sql> – 可以重用的 SQL 块,也可以被其他语句引用。
6. <insert> – 映射插入语句
7. <update> – 映射更新语句
8. <delete> – 映射删除语句
9. <select> – 映射查询语句

#### 查询(select)

查询语句是使用 MyBatis 时最常用的元素之一。直到你从数据库取出数据时才会发现将数据存在数据库中是多么的有价值, 所以许多应用程序查询要比更改数据多的多。对于每次插入,更新或删除,那也会有很多的查询。这是 MyBatis 的一个基本原则,也是将重心和努力放到查询和结果映射的原因。对简单类别的查询元素是非常简单的。比如:

<select id=*"selectPerson"* parameterType=*"int"* resultType=*"hashmap"*>

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

</select>

这个语句被称作 selectPerson, 使用一个 int (或 Integer) 类型的参数, 并返回一个 HashMap 类型的对象,其中的键是列名,值是列对应的值。

注意参数注释:

#{id}

这就告诉 MyBatis 创建一个预处理语句参数。使用 JDBC, 这样的一个参数在 SQL 中会由一个“?”来标识,并被传递到一个新的预处理语句中,就像这样:

// Similar JDBC code, NOT MyBatis… String selectPerson = "SELECT \* FROM PERSON WHERE ID=?"; PreparedStatement ps = conn.prepareStatement(selectPerson);

ps.setInt(1,id);

当然, 这需要很多单独的 JDBC 的代码来提取结果并将它们映射到对象实例中, 这就是 MyBatis 节省你时间的地方。我们需要深入了解参数和结果映射。那些细节部分我们下面来了解。

select 元素有很多属性允许你配置,来决定每条语句的作用细节。

<select id=*"selectPerson"* parameterType=*"int"* parameterMap=*"deprecated"* resultType=*"hashmap"* resultMap=*"personResultMap"* flushCache=*"false"* useCache=*"true"* timeout=*"10000"* fetchSize=*"256"* statementType=*"PREPARED"* resultSetType=*"FORWARD\_ONLY"*>

|  |  |
| --- | --- |
| 属性 | 描述 |
| id | 在命名空间中唯一的标识符,可以被用来引用这条语句。 |
| parameterType | 将会传入这条语句的参数类的完全限定名或别名。 |
| parameterMap | 这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。 |
| resultType | 从这条语句中返回的期望类型的类的完全限定名或别名。注意集合情形,那应该是集合可以包含的类型,而不能是集合本身。使用 resultType 或 resultMap,但不能同时使用。 |
| resultMap | 命名引用外部的 resultMap。返回 map 是 MyBatis 最具力量的特性, 对其有一个很好的理解的话, 许多复杂映射的情形就能被解决了。使用 resultMap 或 resultType,但不能同时使用。 |
| flushCache | 将其设置为 true,不论语句什么时候被带哦用,都会导致缓存被清空。默认值:false。 |
| useCache | 将其设置为 true, 将会导致本条语句的结果被缓存。默认值: true。 |
| timeout | 这个设置驱动程序等待数据库返回请求结果,并抛出异常时间的最大等待值。默认不设置(驱动自行处理) |
| fetchSize | 这是暗示驱动程序每次批量返回的结果行数。默认不设置(驱动自行处理)。 |
| statementType | STA TEMENT,PREPARED 或 CALLABLE 的一种。这会让 MyBatis 使用选择使用 Statement,PreparedStatement 或 CallableStatement。默认值:PREPARED。 |
| resultSetType | FORWARD\_ONLY|SCROLL\_SENSITIVE|SCROLL\_INSENSITIVE 中的一种。默认是不设置(驱动自行处理)。 |
| databaseId | 如果存在已配置的databaseIdProvider，则MyBatis将加载所有没有databaseId属性的语句或与当前匹配的databaseId。 如果遇到相同的语句，如果发现有和没有databaseId后者将被丢弃。 |

#### 更新(insert、update and delete)

数据变更语句 insert,update 和 delete 在它们的实现中非常相似:

<insert id=*"insertAuthor"* parameterType=*"domain.blog.Author"* flushCache=*"true"* statementType=*"PREPARED"* keyProperty=*""* keyColumn=*""* useGeneratedKeys=*""* timeout=*"20000"*>

<update id=*"insertAuthor"* parameterType=*"domain.blog.Author"* flushCache=*"true"* statementType=*"PREPARED"* timeout=*"20000"*>

<delete id=*"insertAuthor"* parameterType=*"domain.blog.Author"* flushCache=*"true"* statementType=*"PREPARED"* timeout=*"20000"*>

|  |  |
| --- | --- |
| 属性 | 描述 |
| id | 在命名空间中唯一的标识符,可以被用来引用这条语句。 |
| parameterType | 将会传入这条语句的参数类的完全限定名或别名。 |
| parameterMap | 这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。 |
| flushCache | 将其设置为 true,不论语句什么时候被带哦用,都会导致缓存被清空。默认值:false。 |
| timeout | 这个设置驱动程序等待数据库返回请求结果, 并抛出异常时间的最大等待值。默认不设置(驱动自行处理)。 |
| statementType | STA TEMENT,PREPARED 或 CALLABLE 的一种。这会让 MyBatis 使用选择使用 Statement,PreparedStatement 或 CallableStatement。默认值:PREPARED。 |
| useGeneratedKeys | ( 仅 对 insert 有 用 ) 这 会 告 诉 MyBatis 使 用 JDBC 的 getGeneratedKeys 方法来取出由数据(比如:像 MySQL 和 SQL Server 这样的数据库管理系统的自动递增字段)内部生成的主键。默认值:false。 |
| keyProperty | (仅对 insert 有用) 标记一个属性, MyBatis 会通过 getGeneratedKeys 或者通过 insert 语句的 selectKey 子元素设置它的值。默认: 不设置。 |
| keyColumn | (仅对 insert 有用) 标记一个属性, MyBatis 会通过 getGeneratedKeys 或者通过 insert 语句的 selectKey 子元素设置它的值。默认: 不设置。 |

下面就是 insert,update 和 delete 语句的示例:

<insert id=*"insertAuthor"* parameterType=*"domain.blog.Author"*>

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

</insert>

<update id=*"updateAuthor"* parameterType=*"domain.blog.Author"*>

update Author set username = #{username},password = #{password},email = #{email},bio = #{bio} where id = #{id}

</update>

<delete id=*"deleteAuthor"* parameterType=*"int"*>

delete from Author where id = #{id}

</delete>

如前所述,插入语句有一点多,它有一些属性和子元素用来处理主键的生成。

首先,如果你的数据库支持自动生成主键的字段(比如 MySQL 和 SQL Server) ,那么你可以设置 useGeneratedKeys=”true”,而且设置 keyProperty 到你已经做好的目标属性上。例如,如果上面的 Author 表已经对 id 使用了自动生成的列类型,那么语句可以修改为:

<insert id=*"insertAuthor"* parameterType=*"domain.blog.Author"* useGeneratedKeys=*"true"* keyProperty=*"id"*>

insert into Author (username,password,email,bio)values (#{username},#{password},#{email},#{bio})

</insert>

MyBatis 有另外一种方法来处理数据库不支持自动生成类型,或者可能 JDBC 驱动不支持自动生成主键时的主键生成问题。

这里有一个简单(甚至很傻)的示例,它可以生成一个随机 ID(可能你不会这么做, 但是这展示了 MyBatis 处理问题的灵活性,因为它并不真的关心 ID 的生成)

<insert id=*"insertAuthor"* parameterType=*"domain.blog.Author"*>

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

select CAST(RANDOM()\*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1

</selectKey>

insert into Author(id, username, password, email,bio, favourite\_section)

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

</insert>

在上面的示例中,selectKey 元素将会首先运行,Author 的 id 会被设置,然后插入语句会被调用。这给你了一个简单的行为在你的数据库中来处理自动生成的主键, 而不需要使你的 Java 代码变得复杂。

**<selectKey> 元素描述如下:**

<selectKey keyProperty=*"id"* resultType=*"int"* order=*"BEFORE"* statementType=*"PREPARED"*>

|  |  |
| --- | --- |
| 屬性 | 描述 |
| keyProperty | selectKey 语句结果应该被设置的目标属性。 |
| resultType | 结果的类型。MyBatis 通常可以算出来,但是写上也没有问题。 MyBatis 允许任何简单类型用作主键的类型,包括字符串。 |
| order | 这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE,那 么它会首先选择主键, 设置 keyProperty 然后执行插入语句。 如果 设置为 AFTER,那么先执行插入语句,然后是 selectKey 元素- 这和如 Oracle 数据库相似,可以在插入语句中嵌入序列调用。 |
| statementType | 和前面的相 同,MyBatis 支持 STA TEMENT ,PREPARED 和 CALLABLE 语句的映射类型,分别代表 PreparedStatement 和 CallableStatement 类型。 |

**sql**

这个元素可以被用来定义可重用的 SQL 代码段,可以包含在其他语句中。比如:

<sql id=*"userColumns"*> id,username,password </sql>

这个 SQL 片段可以被包含在其他语句中,例如:

<select id=*"selectUsers"* parameterType=*"int"* resultType=*"hashmap"*>

select <include refid=*"userColumns"*/>

from some\_table

where id = #{id}

</select>

#### 参数(Parameters)

在之前的语句中, 你已经看到了一些简单参数的示例。 MyBatis 中参数是非常强大的 在 元素。对于简单的做法,大概 90%的情况,是不用太多的,比如:

<select id=*"selectUsers"* parameterType=*"int"* resultType=*"User"*>

select id, username, password

from users

where id = #{id}

</select>

上面的这个示例说明了一个非常简单的命名参数映射。参数类型被设置为“int” ,因此 这个参数可以被设置成任何内容。 原生的类型或简单数据类型, 比如整型和没有相关属性的 字符串,因此它会完全用参数来替代。然而,如果你传递了一个复杂的对象,那么 MyBatis 的处理方式就会有一点不同。比如:

<insert id=*"insertUser"* parameterType=*"User"* >

insert into users (id, username, password)

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

</insert>

如果 User 类型的参数对象传递到了语句中, username 和 password 属性将会被查找, id、 然后它们的值就被传递到预处理语句的参数中。

这点对于传递参数到语句中非常好。但是对于参数映射也有一些其他的特性。

首先,像 MyBatis 的其他部分,参数可以指定一个确定的数据类型。

#{property,javaType=int,jdbcType=NUMERIC}

像 MyBatis 的剩余部分,javaType 通常可以从参数对象中来去顶,除非对象是一个 HashMap。那么 javaType 应该被确定来保证使用正确类型处理器。

注意 如果 null 被当作值来传递,对于所有可能为空的列,JDBC Type 是需要的。以可 以自己通过阅读预处理语句的 setNull()方法的 JavaDocs 文档来研究这个。

为了自定义类型处理器,你可以指定一个确定的类型处理器类(或别名), 比如:

#{age,javaType=int,jdbcType=NUMERIC,typeHandler=MyTypeHandler}

尽管它看起来繁琐,但是实际上是你很少设置它们其中之一。

对于数值类型,对于决定有多少数字是相关的,有一个数值范围。

#{height,javaType=double,jdbcType=NUMERIC,numericScale=2}

最后,mode 属性允许你指定 IN,OUT 或 INOUT 参数。如果参数为 OUT 或 INOUT, 参数对象属性的真实值将会被改变,就像你期望你需要你个输出参数。如果 mode 为 OUT (或 INOUT) ,而且 jdbcType 为 CURSOR(也就是 Oracle 的 REFCURSOR) ,你必须指定 一个 resultMap 来映射结果集到参数类型。要注意这里的 javaType 属性是可选的,如果左边 的空白是 jdbcType 的 CURSOR 类型,它会自动地被设置为结果集。

#{department, mode=OUT, jdbcType=CURSOR, javaType=ResultSet, resultMap=departmentResultMap}

MyBatis 也支持很多高级的数据类型,比如结构体,但是当注册 out 参数时你必须告诉 语句类型名称。比如(再次提示,在实际中不要像这样换行):

#{middleInitial, mode=OUT, jdbcType=STRUCT, jdbcTypeName=MY\_TYPE, resultMap=departmentResultMap}

尽管所有这些强大的选项很多时候你只简单指定属性名,MyBatis 会自己计算剩余的。 最多的情况是你为 jdbcType 指定可能为空的列名。

#{firstName}

#{middleInitial,jdbcType=VARCHAR}

#{lastName}

**字符串替换**

默认情况下,使用#{}格式的语法会导致 MyBatis 创建预处理语句属性并以它为背景设 置安全的值(比如?) 。这样做很安全,很迅速也是首选做法,有时你只是想直接在 SQL 语 句中插入一个不改变的字符串。比如,像 ORDER BY,你可以这样来使用:

ORDER BY ${columnName}

这里 MyBatis 不会修改或转义字符串。

重要 接受从用户输出的内容并提供给语句中不变的字符串,这样做是不安全的。这会 导致潜在的 SQL 注入攻击,因此你不应该允许用户输入这些字段,或者通常自行转义并检 查。

#### 结果集(Result Maps)

resultMap 元素是 MyBatis 中最重要最强大的元素。它就是让你远离 90%的需要从结果 集中取出数据的 JDBC 代码的那个东西, 而且在一些情形下允许你做一些 JDBC 不支持的事 情。 事实上, 编写相似于对复杂语句联合映射这些等同的代码, 也许可以跨过上千行的代码。 ResultMap 的设计就是简单语句不需要明确的结果映射,而很多复杂语句确实需要描述它们 的关系。

你已经看到简单映射语句的示例了,但没有明确的 resultMap。比如:

<select id=*"selectUsers"* parameterType=*"int"* resultType=*"hashmap"*>

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

这样一个语句简单作用于所有列被自动映射到 HashMap 的键上,这由 resultType 属性 指定。这在很多情况下是有用的,但是 HashMap 不能很好描述一个领域模型。那样你的应 用程序将会使用 JavaBeans 或 POJOs(Plain Old Java Objects,普通 Java 对象)来作为领域 模型。MyBatis 对两者都支持。看看下面这个 JavaBean:

**package** com.someapp.model;

**public** **class** User {

**private** **int** id;

**private** String username;

**private** String hashedPassword;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** String getHashedPassword() {

**return** hashedPassword;

}

**public** **void** setHashedPassword(String hashedPassword) {

**this**.hashedPassword = hashedPassword;

}

}

基于 JavaBean 的规范,上面这个类有 3 个属性:id,username 和 hashedPassword。这些 在 select 语句中会精确匹配到列名。

这样的一个 JavaBean 可以被映射到结果集,就像映射到 HashMap 一样简单。

<select id=*"selectUsers"* parameterType=*"int"* resultType=*"com.someapp.model.User"*>

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

要记住类型别名是你的伙伴。使用它们你可以不用输入类的全路径。比如:

<!-- In mybatis-config.xml file -->

<typeAlias type=*"com.someapp.model.User"* alias=*"User"*/>

<!-- In SQL Mapping XML file -->

<select id=*"selectUsers"* parameterType=*"int"* resultType=*"User"*>

select id, username, hashedPassword

from some\_table

where id = #{id}

</select>

这些情况下,MyBatis 会在幕后自动创建一个 ResultMap,基于属性名来映射列到 JavaBean 的属性上。如果列名没有精确匹配,你可以在列名上使用 select 字句的别名(一个 基本的 SQL 特性)来匹配标签。比如:

<select id=*"selectUsers"* parameterType=*"int"* resultType=*"User"*>

select

user\_id as "id",

user\_name as "userName",

hashed\_password as "hashedPassword"

from some\_table

where id = #{id}

</select>

ResultMap 最优秀的地方你已经了解了很多了,但是你还没有真正的看到一个。这些简 单的示例不需要比你看到的更多东西。 只是出于示例的原因, 让我们来看看最后一个示例中 外部的 resultMap 是什么样子的,这也是解决列名不匹配的另外一种方式。

<resultMap id=*"userResultMap"* type=*"User"*>

<id property=*"id"* column=*"user\_id"* />

<result property=*"username"* column=*"username"*/>

<result property=*"password"* column=*"password"*/>

</resultMap>

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

<select id=*"selectUsers"* parameterType=*"int"* resultMap=*"userResultMap"*>

select user\_id, user\_name, hashed\_password

from some\_table

where id = #{id}

</select>

如果世界总是这么简单就好了。

**高级结果映射**

MyBatis 创建的一个想法:数据库不用永远是你想要的或需要它们是什么样的。而我们 最喜欢的数据库最好是第三范式或 BCNF 模式,但它们有时不是。如果可能有一个单独的 数据库映射,所有应用程序都可以使用它,这是非常好的,但有时也不是。结果映射就是 MyBatis 提供处理这个问题的答案。

比如,我们如何映射下面这个语句?

<!-- Very Complex Statement -->

<select id=*"selectBlogDetails"* parameterType=*"int"* resultMap=*"detailedBlogResultMap"*>

select

B.id as blog\_id,

B.title as blog\_title,

B.author\_id as blog\_author\_id,

A.id as author\_id,

A.username as author\_username,

A.password as author\_password,

A.email as author\_email,

A.bio as author\_bio,

A.favourite\_section as author\_favourite\_section,

P.id as post\_id,

P.blog\_id as post\_blog\_id,

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

P.created\_on as post\_created\_on,

P.section as post\_section,

P.subject as post\_subject,

P.draft as draft,

P.body as post\_body,

C.id as comment\_id,

C.post\_id as comment\_post\_id,

C.name as comment\_name,

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.post\_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>

你可能想把它映射到一个智能的对象模型,包含一个作者写的博客,有很多的博文,每 篇博文有零条或多条的评论和标签。 下面是一个完整的复杂结果映射例子 (假设作者, 博客, 博文, 评论和标签都是类型的别名) 我们来看看, 。 但是不用紧张, 我们会一步一步来说明。 当天最初它看起来令人生畏,但实际上非常简单。

<!-- Very Complex Result Map -->

<resultMap id=*"detailedBlogResultMap"* 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\_username"*/>

<result property=*"password"* column=*"author\_password"*/>

<result property=*"email"* column=*"author\_email"*/>

<result property=*"bio"* column=*"author\_bio"*/>

<result property=*"favouriteSection"* column=*"author\_favourite\_section"*/>

</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>

resultMap 元素有很多子元素和一个值得讨论的结构。 下面是 resultMap 元素的概念视图

**resultMap**

·constructor---类在实例化时,用来注入结果到构造方法中

·idArg---ID参数；标记结果作为ID可以帮助提高整体效能

·arg-----注入到结构方法的一个普通结果

·id------一个ID结果；标记结果作为ID可以帮助提高整体效能

·result----注入到字段或JavaBean属性的普通结果

·association------一个复杂的类型关联；许多结果将包成这种类型

·嵌入结果映射-----结果映射自身的集，或者参考一个

·discriminator-----使用结果值来决定使用哪个结果映射

·case----基于某些值的结果映射

·嵌入结果映射----这种情形结果也映射它本身，因此可以包含很多相同的元素，或者它可以参照一个外部的结果映射。

|  |  |
| --- | --- |
| 属性 | 描述 |
| id | 当前命名空间中的一个唯一标识，用于标识一个result map. |
| type | 类的全限定名, 或者一个类型别名 (内置的别名可以参考上面的表格). |
| autoMapping | 如果设置这个属性，MyBatis将会为这个ResultMap开启或者关闭自动映射。这个属性会覆盖全局的属性autoMappingBehavior。默认值为：unset。 |

最佳实践 通常逐步建立结果映射。单元测试的真正帮助在这里。如果你尝试创建一次创建一个向上面示例那样的巨大的结果映射, 那么可能会有错误而且很难去控制它来工作。开始简单一些,一步一步的发展。而且要进行单元测试!使用该框架的缺点是它们有时是黑盒(是否可见源代码) 。你确定你实现想要的行为的最好选择是编写单元测试。它也可以你帮助得到提交时的错误。

下面一部分将详细说明每个元素。

**id & result**

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

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

这些是结果映射最基本内容。id 和 result 都映射一个单独列的值到简单数据类型(字符串,整型,双精度浮点数,日期等)的单独属性或字段。

这两者之间的唯一不同是 id 表示的结果将是当比较对象实例时用到的标识属性。这帮助来改进整体表现,特别是缓存和嵌入结果映射(也就是联合映射) 。

每个都有一些属性:

|  |  |
| --- | --- |
| 属性 | 描述 |
| property | 映射到列结果的字段或属性。如果匹配的是存在的,和给定名称相同的 JavaBeans 的属性,那么就会使用。否则 MyBatis 将会寻找给定名称 property 的字段。这两种情形你可以使用通常点式的复杂属性导航。比如,你可以这样映射一些东西: “username” ,或者映射到一些复杂的东西: “address.street.number” 。 |
| column | 从数据库中得到的列名,或者是列名的重命名标签。这也是通常和会传递给 resultSet.getString(columnName)方法参数中相同的字符串。 |
| javaType | 一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列表) 。如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。然而,如果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证所需的行为。 |
| jdbcType | 在这个表格之后的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅需要对插入,更新和删除操作可能为空的列进行处理。这是 JDBC jdbcType 的需要,而不是 MyBatis 的。如果你直接使用 JDBC 编程,你需要指定这个类型-但仅仅对可能为空的值。 |
| typeHandler | 我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默认的类型处理器。这个属性值是类的完全限定名或者是一个类型处理器的实现,或者是类型别名。 |

**支持的 JDBC 类型**

为了未来的参考,MyBatis 通过包含的 jdbcType 枚举型,支持下面的 JDBC 类型。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BIT | FLOAT | CHAR | TIMESTAMP | OTHER | UNDEFINED |
| TINYINT | REAL | VARCHAR | BINARY | BLOB | NVARCHAR |
| SMALLINT | DOUBLE | LONGVARCHAR | VARBINARY | CLOB | NCHAR |
| INTEGER | NUMERIC | DATE | LONGVARBINARY | BOOLEAN | NCLOB |
| BIGINT | DECIMAL | TIME | NULL | CURSOR | ARRAY |

**构造方法**

对于大多数数据传输对象(Data Transfer Object,DTO)类型,属性可以起作用,而且像你绝大多数的领域模型, 指令也许是你想使不可变类的地方。通常包含引用或查询数据的表很少或基本不变的话对不可变类来说是合适的。构造方法注入允许你在初始化时为类设置属性的值,而不用暴露出公有方法。MyBatis 也支持私有属性和私有 JavaBeans 属性来达到这个目的,但是一些人更青睐构造方法注入。构造方法元素支持这个。

看看下面这个构造方法:

**public** **class** User {

//...

**public** User(Integer id, String username, **int** age) {

//...

}

//...

}

为了将结果注入构造方法，MyBatis需要通过某种方式定位相应的构造方法。在下面的例子中，MyBatis搜索一个声明了三个形参的的构造方法，以 java.lang.Integer, java.lang.String and int 的顺序排列。

<constructor>

<idArg column=*"id"* javaType=*"int"*/>

<arg column=*"username"* javaType=*"String"*/>

<arg column=*"age"* javaType=*"\_int"*/>

</constructor>

当你在处理一个带有多个形参的构造方法时，对arg元素顺序的维持是很容易出错的。为了能利用构造方法形参的name来对形参进行引用，你可以添加 @Param 注解或者使用'-parameters'编译选项和启用 useActualParamName （此选项默认开启）来编译工程。下面的例子对于同一个构造方法依然是有效的，尽管第二和第三个形参顺序与构造方法中声明的顺序不匹配。

<constructor>

<idArg column=*"id"* javaType=*"int"* name=*"id"* />

<arg column=*"age"* javaType=*"\_int"* name=*"age"* />

<arg column=*"username"* javaType=*"String"* name=*"username"* />

</constructor>

如果存在具有同名和相同类型的属性，那么它的 javaType 可以省略。

剩余的属性和规则和固定的 id 和 result 元素是相同的。

|  |  |
| --- | --- |
| 属性 | 描述 |
| column | 来自数据库的类名,或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方法的字符串是相同的。 |
| javaType | 一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列表)。如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。然而,如果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证所需的行为。 |
| jdbcType | 在这个表格之前的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅需要对插入, 更新和删除操作可能为空的列进行处理。这是 JDBC 的需要, jdbcType 而不是 MyBatis 的。如果你直接使用 JDBC 编程,你需要指定这个类型-但仅仅对可能为空的值。 |
| typeHandler | 我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默认的类型处理器。这个属性值是类的完全限定名或者是一个类型处理器的实现, 或者是类型别名。 |
| select | 用于加载复杂类型属性的映射语句的ID,从column中检索出来的数据，将作为此select语句的参数。具体请参考Association标签。 |
| resultMap | ResultMap的ID，可以将嵌套的结果集映射到一个合适的对象树中，功能和select属性相似，它可以实现将多表连接操作的结果映射成一个单一的ResultSet。这样的ResultSet将会将包含重复或部分数据重复的结果集正确的映射到嵌套的对象树中。为了实现它, MyBatis允许你 “串联” ResultMap,以便解决嵌套结果集的问题。想了解更多内容，请参考下面的Association元素。 |
| name | 构造方法形参的名字。通过指定具体的名字你可以以任意顺序写入arg元素。参看上面的解释。从3.4.3版本起。 |

关联

<association property=*"author"* column=*"blog\_author\_id"* javaType=*"Author"*>

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

<result property=*"username"* column=*"author\_username"*/>

</association>

关联元素处理“有一个”类型的关系。比如,在我们的示例中,一个博客有一个用户。关联映射就工作于这种结果之上。你指定了目标属性,来获取值的列,属性的 java 类型(很多情况下 MyBatis 可以自己算出来) ,如果需要的话还有 jdbc 类型,如果你想覆盖或获取的结果值还需要类型控制器。

关联中不同的是你需要告诉 MyBatis 如何加载关联。MyBatis 在这方面会有两种不同的方式:

1. 嵌套查询:通过执行另外一个 SQL 映射语句来返回预期的复杂类型。
2. 嵌套结果:使用嵌套结果映射来处理重复的联合结果的子集。首先,然让我们来查看这个元素的属性。所有的你都会看到,它和普通的只由 select 和

resultMap 属性的结果映射不同。

|  |  |
| --- | --- |
| 属性 | 描述 |
| property | 映射到列结果的字段或属性。如果匹配的是存在的,和给定名称相同的 property JavaBeans 的属性, 那么就会使用。否则 MyBatis 将会寻找给定名称的字段。这两种情形你可以使用通常点式的复杂属性导航。比如,你可以这样映射一 些 东 西 :“ username ”, 或 者 映 射 到 一 些 复 杂 的 东 西 : “address.street.number” 。 |
| javaType | 一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列表) 。如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。然而,如 javaType 果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证所需的行为。 |
| jdbcType | 在这个表格之前的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅需要对插入, 更新和删除操作可能为空的列进行处理。这是 JDBC 的需要, jdbcType 而不是 MyBatis 的。如果你直接使用 JDBC 编程,你需要指定这个类型-但仅仅对可能为空的值。 |
| typeHandler | 我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默认的 typeHandler 类型处理器。这个属性值是类的完全限定名或者是一个类型处理器的实现, 或者是类型别名。 |

**关联的嵌套查询**

|  |  |
| --- | --- |
| 属性 | 描述 |
| column | 来自数据库的类名,或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方法的字符串是相同的。 column 注 意 : 要 处 理 复 合 主 键 , 你 可 以 指 定 多 个 列 名 通 过 column= ” {prop1=col1,prop2=col2} ” 这种语法来传递给嵌套查询语 句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。 |
| select | 另外一个映射语句的 ID,可以加载这个属性映射需要的复杂类型。获取的在列属性中指定的列的值将被传递给目标 select 语句作为参数。表格后面有一个详细的示例。 select 注 意 : 要 处 理 复 合 主 键 , 你 可 以 指 定 多 个 列 名 通 过 column= ” {prop1=col1,prop2=col2} ” 这种语法来传递给嵌套查询语 句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。 |
| fetchType | 可选的。有效值为 lazy和eager。 如果使用了，它将取代全局配置参数lazyLoadingEnabled。 |

示例：

<resultMap id=*"blogResult"* type=*"Blog"*>

<association property=*"author"* column=*"author\_id"* javaType=*"Author"* select=*"selectAuthor"*/>

</resultMap>

<select id=*"selectBlog"* resultMap=*"blogResult"*>

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

</select>

<select id=*"selectAuthor"* resultType=*"Author"*>

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

</select>

我们有两个查询语句:一个来加载博客,另外一个来加载作者,而且博客的结果映射描述了“selectAuthor”语句应该被用来加载它的 author 属性。

其他所有的属性将会被自动加载,假设它们的列和属性名相匹配。

这种方式很简单, 但是对于大型数据集合和列表将不会表现很好。问题就是我们熟知的 “N+1 查询问题”。概括地讲,N+1 查询问题可以是这样引起的:

1. 你执行了一个单独的 SQL 语句来获取结果列表(就是“+1”)。
2. 对返回的每条记录,你执行了一个查询语句来为每个加载细节(就是“N”)。

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。

MyBatis 能延迟加载这样的查询就是一个好处,因此你可以分散这些语句同时运行的消耗。然而,如果你加载一个列表,之后迅速迭代来访问嵌套的数据,你会调用所有的延迟加载,这样的行为可能是很糟糕的。

所以还有另外一种方法。

|  |  |
| --- | --- |
| 属性 | 描述 |
| resultMap | 这是结果映射的 ID,可以映射关联的嵌套结果到一个合适的对象图中。这是一种替代方法来调用另外一个查询语句。这允许你联合多个表来合成到 resultMap 一个单独的结果集。这样的结果集可能包含重复,数据的重复组需要被分解,合理映射到一个嵌套的对象图。为了使它变得容易,MyBatis 让你“链接”结果映射,来处理嵌套结果。一个例子会很容易来仿照,这个表格后面也有一个示例。 |
| columnPrefix | 当连接多表时，你将不得不使用列别名来避免ResultSet中的重复列名。指定columnPrefix允许你映射列名到一个外部的结果集中。 请看后面的例子。 |
| notNullColumn | 默认情况下，子对象仅在至少一个列映射到其属性非空时才创建。 通过对这个属性指定非空的列将改变默认行为，这样做之后Mybatis将仅在这些列非空时才创建一个子对象。 可以指定多个列名，使用逗号分隔。默认值：未设置(unset)。 |
| autoMapping | 如果使用了，当映射结果到当前属性时，Mybatis将启用或者禁用自动映射。 该属性覆盖全局的自动映射行为。 注意它对外部结果集无影响，所以在select or resultMap属性中这个是毫无意义的。 默认值：未设置(unset)。 |

在上面你已经看到了一个非常复杂的嵌套关联的示例。下面这个是一个非常简单的示例来说明它如何工作。代替了执行一个分离的语句,我们联合博客表和作者表在一起,就像:

<select id=*"selectBlog"* resultMap=*"blogResult"*>

select

B.id as blog\_id,

B.title as blog\_title,

B.author\_id as blog\_author\_id,

A.id as author\_id,

A.username as author\_username,

A.password as author\_password,

A.email as author\_email,

A.bio as author\_bio

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

where B.id = #{id}

</select>

注意这个联合查询, 以及采取保护来确保所有结果被唯一而且清晰的名字来重命名。这使得映射非常简单。现在我们可以映射这个结果:

<resultMap id=*"blogResult"* type=*"Blog"*>

<id property=*"id"* column=*"blog\_id"* />

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

<association property=*"author"* column=*"blog\_author\_id"* javaType=*"Author"* resultMap=*"authorResult"*/>

</resultMap>

<resultMap id=*"authorResult"* type=*"Author"*>

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

<result property=*"username"* column=*"author\_username"*/>

<result property=*"password"* column=*"author\_password"*/>

<result property=*"email"* column=*"author\_email"*/>

<result property=*"bio"* column=*"author\_bio"*/>

</resultMap>

在上面的示例中你可以看到博客的作者关联代表着“authorResult”结果映射来加载作者实例。

非常重要: id元素在嵌套结果映射中扮演着非常重要的角色。你应该总是指定一个或多个可以唯一标识结果的属性。实际上如果你不指定它的话, MyBatis仍然可以工作,但是会有严重的性能问题。在可以唯一标识结果的情况下, 尽可能少的选择属性。主键是一个显而易见的选择（即使是复合主键）。

现在,上面的示例用了外部的结果映射元素来映射关联。这使得 Author 结果映射可以重用。然而,如果你不需要重用它的话,或者你仅仅引用你所有的结果映射合到一个单独描述的结果映射中。你可以嵌套结果映射。这里给出使用这种方式的相同示例:

<resultMap id=*"blogResult"* type=*"Blog"*>

<id property=*"id"* column=*"blog\_id"* />

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

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

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

<result property=*"username"* column=*"author\_username"*/>

<result property=*"password"* column=*"author\_password"*/>

<result property=*"email"* column=*"author\_email"*/>

<result property=*"bio"* column=*"author\_bio"*/>

</association>

</resultMap>

如果blog有一个co-author怎么办？ select语句将看起来这个样子：

<select id=*"selectBlog"* resultMap=*"blogResult"*>

select

B.id as blog\_id,

B.title as blog\_title,

A.id as author\_id,

A.username as author\_username,

A.password as author\_password,

A.email as author\_email,

A.bio as author\_bio,

CA.id as co\_author\_id,

CA.username as co\_author\_username,

CA.password as co\_author\_password,

CA.email as co\_author\_email,

CA.bio as co\_author\_bio

from Blog B

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

left outer join Author CA on B.co\_author\_id = CA.id

where B.id = #{id}

</select>

再次调用Author的resultMap将定义如下：

<resultMap id=*"authorResult"* type=*"Author"*>

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

<result property=*"username"* column=*"author\_username"*/>

<result property=*"password"* column=*"author\_password"*/>

<result property=*"email"* column=*"author\_email"*/>

<result property=*"bio"* column=*"author\_bio"*/>

</resultMap>

因为结果中的列名与resultMap中的列名不同。 你需要指定columnPrefix去重用映射co-author结果的resultMap。

<resultMap id=*"blogResult"* type=*"Blog"*>

<id property=*"id"* column=*"blog\_id"* />

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

<association property=*"author"*

resultMap=*"authorResult"* />

<association property=*"coAuthor"*

resultMap=*"authorResult"*

columnPrefix=*"co\_"* />

</resultMap>

上面你已经看到了如何处理“有一个”类型关联。但是“有很多个”是怎样的?下面这个部分就是来讨论这个主题的。

**集合**

<collection property=*"posts"* ofType=*"domain.blog.Post"*>

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

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

<result property=*"body"* column=*"post\_body"*/>

</collection>

集合元素的作用几乎和关联是相同的。实际上,它们也很相似,文档的异同是多余的。所以我们更多关注于它们的不同。

我们来继续上面的示例,一个博客只有一个作者。但是博客有很多文章。在博客类中, 这可以由下面这样的写法来表示:

**private** List<Post> posts;

要映射嵌套结果集合到 List 中,我们使用集合元素。就像关联元素一样,我们可以从连接中使用嵌套查询,或者嵌套结果。

**集合的嵌套查询**

首先,让我们看看使用嵌套查询来为博客加载文章。

<resultMap id=*"blogResult"* type=*"Blog"*>

<collection property=*"posts"* javaType=*"ArrayList"* column=*"id"* ofType=*"Post"* select=*"selectPostsForBlog"*/>

</resultMap>

<select id=*"selectBlog"* resultMap=*"blogResult"*>

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

</select>

<select id=*"selectPostsForBlog"* resultType=*"Post"*>

SELECT \* FROM POST WHERE BLOG\_ID = #{id}

</select>

这里你应该注意很多东西,但大部分代码和上面的关联元素是非常相似的。首先,你应该注意我们使用的是集合元素。然后要注意那个新的“ofType”属性。这个属性用来区分 JavaBean(或字段)属性类型和集合包含的类型来说是很重要的。所以你可以读出下面这个映射:

<collection property=*"posts"* javaType=*"ArrayList"* column=*"id"* ofType=*"Post"* select=*"selectPostsForBlog"*/>

读作: “在 Post 类型的 ArrayList 中的 posts 的集合。”

javaType 属性是不需要的,因为 MyBatis 在很多情况下会为你算出来。所以你可以缩短写法:

<collection property=*"posts"* column=*"id"* ofType=*"Post"* select=*"selectPostsForBlog"*/>

**集合的嵌套结果**

至此,你可以猜测集合的嵌套结果是如何来工作的,因为它和关联完全相同,除了它应用了一个“ofType”属性

首先，我们来看看SQL:

<select id=*"selectBlog"* resultMap=*"blogResult"*>

select

B.id as blog\_id,

B.title as blog\_title,

B.author\_id as blog\_author\_id,

P.id as post\_id,

P.subject as post\_subject,

P.body as post\_body,

from Blog B

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

where B.id = #{id}

</select>

我们又一次联合了博客表和文章表,而且关注于保证特性,结果列标签的简单映射。现在用文章映射集合映射博客,可以简单写为:

<resultMap id=*"blogResult"* type=*"Blog"*>

<id property=*"id"* column=*"blog\_id"* />

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

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

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

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

<result property=*"body"* column=*"post\_body"*/>

</collection>

</resultMap>

同样,要记得 id 元素的重要性,如果你不记得了,请阅读上面的关联部分。

同样, 如果你引用更长的形式允许你的结果映射的更多重用, 你可以使用下面这个替代的映射:

<resultMap id=*"blogResult"* type=*"Blog"*>

<id property=*"id"* column=*"blog\_id"* />

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

<collection property=*"posts"* ofType=*"Post"* resultMap=*"blogPostResult"* columnPrefix=*"post\_"*/>

</resultMap>

<resultMap id=*"blogPostResult"* type=*"Post"*>

<id property=*"id"* column=*"id"*/>

<result property=*"subject"* column=*"subject"*/>

<result property=*"body"* column=*"body"*/>

</resultMap>

注意 这个对你所映射的内容没有深度,广度或关联和集合相联合的限制。当映射它们时你应该在大脑中保留它们的表现。你的应用在找到最佳方法前要一直进行的单元测试和性能测试。好在 myBatis 让你后来可以改变想法,而不对你的代码造成很小(或任何)影响。

高级关联和集合映射是一个深度的主题。文档只能给你介绍到这了。加上一点联系,你会很快清楚它们的用法。

**鉴别器**

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

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

</discriminator>

有时一个单独的数据库查询也许返回很多不同 (但是希望有些关联) 数据类型的结果集。鉴别器元素就是被设计来处理这个情况的, 还有包括类的继承层次结构。鉴别器非常容易理解,因为它的表现很像 Java 语言中的 switch 语句。

定义鉴别器指定了 column 和 javaType 属性。列是 MyBatis 查找比较值的地方。 JavaType 是需要被用来保证等价测试的合适类型(尽管字符串在很多情形下都会有用)。比如:

<resultMap id=*"vehicleResult"* type=*"Vehicle"*>

<id property=*"id"* column=*"id"* />

<result property=*"vin"* column=*"vin"*/>

<result property=*"year"* column=*"year"*/>

<result property=*"make"* column=*"make"*/>

<result property=*"model"* column=*"model"*/>

<result property=*"color"* column=*"color"*/>

<discriminator javaType=*"int"* column=*"vehicle\_type"*>

<case value=*"1"* resultMap=*"carResult"*/>

<case value=*"2"* resultMap=*"truckResult"*/>

<case value=*"3"* resultMap=*"vanResult"*/>

<case value=*"4"* resultMap=*"suvResult"*/>

</discriminator>

</resultMap>

在这个示例中, MyBatis 会从结果集中得到每条记录, 然后比较它的 vehicle 类型的值。如果它匹配任何一个鉴别器的实例,那么就使用这个实例指定的结果映射。换句话说,这样做完全是剩余的结果映射被忽略(除非它被扩展,这在第二个示例中讨论) 。如果没有任何一个实例相匹配,那么 MyBatis 仅仅使用鉴别器块外定义的结果映射。所以,如果 carResult 按如下声明:

<resultMap id=*"carResult"* type=*"Car"*>

<result property=*"doorCount"* column=*"door\_count"* />

</resultMap>

那么只有 doorCount 属性会被加载。这步完成后完整地允许鉴别器实例的独立组,尽管和父结果映射可能没有什么关系。这种情况下,我们当然知道 cars 和 vehicles 之间有关系, 如 Car 是一个 Vehicle 实例。因此,我们想要剩余的属性也被加载。我们设置的结果映射的简单改变如下。

<resultMap id=*"carResult"* type=*"Car"* extends=*"vehicleResult"*>

<result property=*"doorCount"* column=*"door\_count"* />

</resultMap>

现在 vehicleResult 和 carResult 的属性都会被加载了。

尽管曾经有些人会发现这个外部映射定义会多少有一些令人厌烦之处。因此还有另外一种语法来做简洁的映射风格。比如:

<resultMap id=*"vehicleResult"* type=*"Vehicle"*>

<id property=*"id"* column=*"id"* />

<result property=*"vin"* column=*"vin"*/>

<result property=*"year"* column=*"year"*/>

<result property=*"make"* column=*"make"*/>

<result property=*"model"* column=*"model"*/>

<result property=*"color"* column=*"color"*/>

<discriminator javaType=*"int"* column=*"vehicle\_type"*>

<case value=*"1"* resultType=*"carResult"*>

<result property=*"doorCount"* column=*"door\_count"* />

</case>

<case value=*"2"* resultType=*"truckResult"*>

<result property=*"boxSize"* column=*"box\_size"* />

<result property=*"extendedCab"* column=*"extended\_cab"* />

</case>

<case value=*"3"* resultType=*"vanResult"*>

<result property=*"powerSlidingDoor"* column=*"power\_sliding\_door"* />

</case>

<case value=*"4"* resultType=*"suvResult"*>

<result property=*"allWheelDrive"* column=*"all\_wheel\_drive"* />

</case>

</discriminator>

</resultMap>

要记得 这些都是结果映射, 如果你不指定任何结果, 那么 MyBatis 将会为你自动匹配列和属性。所以这些例子中的大部分是很冗长的,而其实是不需要的。也就是说,很多数据库是很复杂的,我们不太可能对所有示例都能依靠它。

#### 自动映射(Auto-mapping)

正如你在前面一节看到的，在简单的场景下，MyBatis可以替你自动映射查询结果。 如果遇到复杂的场景，你需要构建一个result map。 但是在本节你将看到，你也可以混合使用这两种策略。 让我们到深一点的层面上看看自动映射是怎样工作的。

当自动映射查询结果时，MyBatis会获取sql返回的列名并在java类中查找相同名字的属性（忽略大小写）。 这意味着如果Mybatis发现了ID列和id属性，Mybatis会将ID的值赋给id。

通常数据库列使用大写单词命名，单词间用下划线分隔；而java属性一般遵循驼峰命名法。 为了在这两种命名方式之间启用自动映射，需要将 mapUnderscoreToCamelCase设置为true。

自动映射甚至在特定的result map下也能工作。在这种情况下，对于每一个result map,所有的ResultSet提供的列， 如果没有被手工映射，则将被自动映射。自动映射处理完毕后手工映射才会被处理。 在接下来的例子中， id 和 userName列将被自动映射， hashed\_password 列将根据配置映射。

<select id=*"selectUsers"* resultMap=*"userResultMap"*>

select

user\_id as "id",

user\_name as "userName",

hashed\_password

from some\_table

where id = #{id}

</select>

<resultMap id=*"userResultMap"* type=*"User"*>

<result property=*"password"* column=*"hashed\_password"*/>

</resultMap>

有三种自动映射等级：

1. NONE - 禁用自动映射。仅设置手动映射属性。
2. PARTIAL - 将自动映射结果除了那些有内部定义内嵌结果映射的(joins).
3. FULL - 自动映射所有。

默认值是PARTIAL，这是有原因的。当使用FULL时，自动映射会在处理join结果时执行，并且join取得若干相同行的不同实体数据，因此这可能导致非预期的映射。下面的例子将展示这种风险：

<select id=*"selectBlog"* resultMap=*"blogResult"*>

select

B.id,

B.title,

A.username,

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

where B.id = #{id}

</select>

<resultMap id=*"blogResult"* type=*"Blog"*>

<association property=*"author"* resultMap=*"authorResult"*/>

</resultMap>

<resultMap id=*"authorResult"* type=*"Author"*>

<result property=*"username"* column=*"author\_username"*/>

</resultMap>

在结果中Blog和Author均将自动映射。但是注意Author有一个id属性，在ResultSet中有一个列名为id， 所以Author的id将被填充为Blog的id，这不是你所期待的。所以需要谨慎使用FULL。

通过添加autoMapping属性可以忽略自动映射等级配置，你可以启用或者禁用自动映射指定的ResultMap。

<resultMap id=*"userResultMap"* type=*"User"* autoMapping=*"false"*>

<result property=*"password"* column=*"hashed\_password"*/>

</resultMap>

#### 缓存(cache)

MyBatis 包含一个非常强大的查询缓存特性,它可以非常方便地配置和定制。MyBatis 3 中的缓存实现的很多改进都已经实现了,使得它更加强大而且易于配置。

默认情况下是没有开启缓存的,除了局部的 session 缓存,可以增强变现而且处理循环依赖也是必须的。要开启二级缓存,你需要在你的 SQL 映射文件中添加一行:

<cache/>

字面上看就是这样。这个简单语句的效果如下:

1. 映射语句文件中的所有 select 语句将会被缓存。
2. 映射语句文件中的所有 insert,update 和 delete 语句会刷新缓存。
3. 缓存会使用 Least Recently Used(LRU,最近最少使用的)算法来收回。
4. 根据时间表(比如 no Flush Interval,没有刷新间隔), 缓存不会以任何时间顺序 来刷新。
5. 缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用。
6. 缓存会被视为是 read/write(可读/可写)的缓存,意味着对象检索不是共享的,而且可以安全地被调用者修改,而不干扰其他调用者或线程所做的潜在修改。

所有的这些属性都可以通过缓存元素的属性来修改。比如:

<cache

eviction=*"FIFO"*

flushInterval=*"60000"*

size=*"512"*

readOnly=*"true"*/>

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会导致冲突。

可用的收回策略有:

1. LRU – 最近最少使用的:移除最长时间不被使用的对象。
2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。
3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
4. 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。

默认的是 LRU。

flushInterval(刷新间隔)可以被设置为任意的正整数,而且它们代表一个合理的毫秒形式的时间段。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。

size(引用数目)可以被设置为任意正整数,要记住你缓存的对象数目和你运行环境的可用内存资源数目。默认值是 1024。

readOnly(只读)属性可以被设置为 true 或 false。只读的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。可读写的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。

**使用自定义缓存**

除了这些自定义缓存的方式, 你也可以通过实现你自己的缓存或为其他第三方缓存方案创建适配器来完全覆盖缓存行为。

<cache type=*"com.domain.something.MyCustomCache"*/>

这个示 例展 示了 如何 使用 一个 自定义 的缓 存实 现。type 属 性指 定的 类必 须实现 org.mybatis.cache.Cache 接口。这个接口是 MyBatis 框架中很多复杂的接口之一,但是简单给定它做什么就行。

**public** **interface** Cache {

String getId();

**int** getSize();

**void** putObject(Object key, Object value);

Object getObject(Object key);

**boolean** hasKey(Object key);

Object removeObject(Object key);

**void** clear();

}

要配置你的缓存, 简单和公有的 JavaBeans 属性来配置你的缓存实现, 而且是通过 cache 元素来传递属性, 比如, 下面代码会在你的缓存实现中调用一个称为 “setCacheFile(String file)” 的方法:

<cache type=*"com.domain.something.MyCustomCache"*>

<property name=*"cacheFile"* value=*"/tmp/my-custom-cache.tmp"*/>

</cache>

你可以使用所有简单类型作为 JavaBeans 的属性,MyBatis 会进行转换。 And you can specify a placeholder(e.g. ${cache.file}) to replace value defined at configuration properties.

从3.4.2版本开始，MyBatis已经支持在所有属性设置完毕以后可以调用一个初始化方法。如果你想要使用这个特性，请在你的自定义缓存类里实现 org.apache.ibatis.builder.InitializingObject 接口。

**public** **interface** InitializingObject {

**void** initialize() **throws** Exception;

}

记得缓存配置和缓存实例是绑定在 SQL 映射文件的命名空间是很重要的。因此,所有在相同命名空间的语句正如绑定的缓存一样。语句可以修改和缓存交互的方式, 或在语句的语句的基础上使用两种简单的属性来完全排除它们。默认情况下,语句可以这样来配置:

<select ... flushCache=*"false"* useCache=*"true"*/>

<insert ... flushCache=*"true"*/>

<update ... flushCache=*"true"*/>

<delete ... flushCache=*"true"*/>

因为那些是默认的,你明显不能明确地以这种方式来配置一条语句。相反,如果你想改变默认的行为,只能设置 flushCache 和 useCache 属性。比如,在一些情况下你也许想排除从缓存中查询特定语句结果,或者你也许想要一个查询语句来刷新缓存。相似地,你也许有一些更新语句依靠执行而不需要刷新缓存。

**参照缓存**

回想一下上一节内容, 这个特殊命名空间的唯一缓存会被使用或者刷新相同命名空间内的语句。也许将来的某个时候,你会想在命名空间中共享相同的缓存配置和实例。在这样的情况下你可以使用 cache-ref 元素来引用另外一个缓存。

<cache-ref namespace=*"com.someone.application.data.SomeMapper"*/>

# MyBatis 的基本用法

MyBatis的真正强大之处在于它的映射语句，这也是它的魔力所在。由于它的映射语句异常强大，映射器的XML文件就显得相对简单。如果将其余具有相同功能的JDBC代码进行对比，立刻就会发现，使用这种方法节省了将近95%的代码量。MyBatis就是针对SQL构建的，并且比普通的方法做的更好。

MyBatis3.0相比2.0版本的一个最大变化，就是支持使用接口来调用方法。

以前使用SqlSession通过命名空间调用MyBatis方法时，首先需要用到命名空间和方法id组成的字符串来调用相应的方法。当参数多于1个的时候，需要将所有参数放到一个Map对象中。通过Map传递多个参数，使用起来很不方便，而且还无法避免很多重复的代码。

使用接口调用方式就会方便很多，MyBatis使用Java的动态代理可以直接通过接口来调用相应的方法，不需要提供接口的实现类，更不需要在实现类中使用SqlSession以通过命名空间间接调用。另外，当有多个参数的时候，通过参数注解@Param设置参数的名字省去了手动构造Map参数的过程，尤其在Spring中使用的时候，可以配置为自动扫描所有的接口类，直接将接口注入需要用到的地方。

备注：大家不要认为使用接口就是使用了MyBatis注解，这两者是不同的。接口可以配合XML使用，也可以配合注解来使用。XML可以单独使用，但是注解必须在接口中使用。

## 一个简单的用户权限需求

在这里简单描述一个权限管理的需求：一个用户拥有若干个角色，一个角色拥有若干个权限，权限就是对某个资源（模块）的某种操作（增、删、改、查），这样就构成了“用户—角色—权限”的授权模型。在正中模型中，用户与角色之间、角色与权限之间，一般是多对多的关系。

用户表

用户ID bigint <pk>

用户名 varchar(50)

密码 varchar(50)

创建时间 datetime

用户角色关联表

用户ID bigint <fk1>

角色ID bigint <fk1>

角色表

角色ID bigint <pk>

角色名 varchar(50)

创建时间 datetime

角色权限关联表

角色ID bigint <fk1>

权限ID bigint <fk1>

权限表

权限ID bigint <pk>

权限名 varchar(50)

权限URL varchar(50)

根据上图的关系创建好相应的表及主外键。数据库表创建完成后，并创建数据库表多对应的JavaBean实体类。

例如：

**public** **class** User {

**private** **int** id;

**private** String name;

**private** String pwd;

**private** String createTime;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getPwd() {

**return** pwd;

}

**public** **void** setPwd(String pwd) {

**this**.pwd = pwd;

}

**public** String getCreateTime() {

**return** createTime;

}

**public** **void** setCreateTime(String createTime) {

**this**.createTime = createTime;

}

}

**public** **class** UserRole {

**private** **int** userId;

**private** **int** roleId;

**public** **int** getUserId() {

**return** userId;

}

**public** **void** setUserId(**int** userId) {

**this**.userId = userId;

}

**public** **int** getRoleId() {

**return** roleId;

}

**public** **void** setRoleId(**int** roleId) {

**this**.roleId = roleId;

}

}

**使用MyBatis的XML方式**

首先需要创建5个表各自对应的Mapper.xml文件，分别是UserMapper.xml、RoleMapper.xml、PrivilegeMapper.xml、UserRoleMapper.xml和RolePrivilegeMapper.xml。

例如：

<?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=*"tk.mybatis.simple.mapper.UserMapper"* >

</mapper>

需要注意的是<mapper>根标签的namespace属性。当Mapper接口和XML文件关联的时候，命名空间namespace的值就需要配置成接口的全限定名称。MyBatis内部就是通过这个值将接口和XML关联起来的（接口和XML配合使用的时候）。如果不使用接口，单纯的使用Mapper.xml配置文件，则该属性的值自定义，且唯一。

创建好Mapper.xml文件后还需要创建mybatis-config.xml（MyBatis的配置文件）配置文件中的<mappers>元素中配置所有的<mapper/>。

<mappers>

<mapper resource=*"mapping/UserMapper.xml"*/>

<mapper resource=*"mapping/* *RoleMapper.xml"*/>

<mapper resource=*"mapping/* *PrivilegeMapper.xml"*/>

<mapper resource=*"mapping/* *UserRoleMapper.xml"*/>

<mapper resource=*"mapping/* *RolePrivilegeMapper.xml"*/>

</mappers>

这种配置方式需要将所有映射文件一一列举出来，如果增加了新的映射文件，还需要注意在此处进行配置，

映射文件配置好后，则可以在Mapper.xml文件中写入需要操作数据库表的各种SQL代码，及JavaBean实体类与数据库表字段之间的映射关系。

<mapper namespace=*"UserMapper"* >

<!-- 定义Bean与查询数据库字段之间的映射关系 -->

<resultMap type=*"User"* id=*"userMap"*>

<id property=*"id"* column=*"id"*/>

<result property=*"name"* column=*"name"*/>

<result property=*"pwd"* column=*"pwd"*/>

<result property=*"createTime "* column=*"createTime"*/>

</resultMap>

<!-- 定义查询给定用户名和密码的用户信息 -->

<select id=*"getUserByNameAndPwd"* parameterType=*"User"* resultMap=*"userMap"*>

select id,name,pwd,createTime from t\_user where name=#{name} and pwd=#{pwd}

</select>

</mapper>

测试使用XML方式使用MyBatis

**public** **static** **void** main(String[] args) {

**try** {

//加载MyBatis的配置文件

Reader reader = Resources.*getResourceAsReader*("mybatisConfig.xml");

//公共工厂类创建SqlSessionFactory对象

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(reader);

//调用SqlSessionFactory对象的方法构建SqlSession对象

SqlSession sqlSession = sqlSessionFactory.openSession();

/\*\*

\* SqlSession对象的方法调用Mapper.xml

\* SqlSession对象的方法的第一个参数为对应Mapper.xml文件中<mapper>标签中namespace属性的值加对应SQL语句的id值

\*/

User users = sqlSession.selectOne("UserMapper.getUserByNameAndPwd");

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**使用MyBatis的接口方式**

首先需要创建5个表各自对应的Mapper接口，分别是IUserMapper.java、IRoleMapper.java、IPrivilegeMapper.java、IUserRoleMapper.java和IRolePrivilegeMapper.java。在接口

例如：

**public** **interface** UserMapper {

}

接口建好后需要在接口中定义需要执行SQL语句的对应方法，并在对应的方法上添加注解及SQL语句。

**public** **interface** IUserMapper {

/\*\*

\* 定义根据用户的id值获取用户数据的方法

\* **@param** id 用户的id值

\* **@return** 返回一个用户对象

\*/

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

**public** User getUser(@Param("id")**int** id);

}

测试使用Mapper接口的方式使用MyBatis

public class TestMyBatis {

public static void main(String[] args) {

try {

//加载MyBatis的配置文件

Reader reader = Resources.*getResourceAsReader*("mybatisConfig.xml");

//公共工厂类创建SqlSessionFactory对象

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);

//将Mapper接口加载到sqlSessionFactory对象中

sqlSessionFactory.getConfiguration().addMapper(IUserMapper.class);

//获取SqlSession对象

SqlSession sqlSession = sqlSessionFactory.openSession();

//获取userMapper接口对象

IUserMapper userMapper = sqlSession.getMapper(IUserMapper.class);

//调用接口中的方法

User user = userMapper.getUser(1);

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

## 查询数据

### 单表查询

**XML方式**

在权限系统中有几个常见的业务，我们需要查询出系统中的用户、角色、权限等数据。在使用纯粹的JDBC时，需要写查询语句，并且对结果集进行手动处理，将结果映射到对象的属性中。使用MyBatis时，只需要在XML中添加一个<select>标签，写一个SQL，再做些简单的配置，就可以将查询的结果直接映射到对象中。

<mapper namespace=*"UserMapper"* >

<!-- 定义Bean与查询数据库字段之间的映射关系 -->

<resultMap type=*"User"* id=*"userMap"*>

<id property=*"id"* column=*"id"*/>

<result property=*"name"* column=*"name"*/>

<result property=*"pwd"* column=*"pwd"*/>

<result property=*"createTime"* column=*"createTime"*/>

</resultMap>

<!-- 定义查询给定用户名和密码的用户信息 -->

<select id=*"getUserByNameAndPwd"* parameterType=*"User"* resultMap=*"userMap"*>

select id,name,pwd,createTime from t\_user where name=#{name} and pwd=#{pwd}

</select>

</mapper>

<select>标签的属性：

1. id：（必填）在命名空间中唯一的标识符，可以被用来引用这条语句。通过<mapper>标签中的namespace的(值.id)来调用对应的SQL语句。（如果Mapper接口与XML配合使用，则id属性的值为Mapper接口中对应的方法名）
2. parameterType：（选填）定义参数的数据类型。如果参数是对象且没在MyBatis配置文件中设置类所对应的包或者没有给类设置别名，则此处的值为完全限定名（包.类）。如果设置了包，则此处值为对应的类名称。如果给类设置了别名，则此处的值可以是别名。如果参数是基本类型，则值为基本类型。
3. resultMap：（选填）该属性值为<resultMap>标签中对应的id值。用于将<resultMap>标签中的映射关系应用给结果。该属性与resultType只能同时使用一个。
4. resultType：（选填）定义结果的类型。如果结果类型为基本类型，则值为基本类型。如果结果为对象且没在MyBatis配置文件中设置类所对应的包或者没有给类设置别名，则此处的值为完全限定名（包.类）。如果设置了包，则此处值为对应的类名称。如果给类设置了别名，则此处的值可以是别名。
5. flushCache：（选填）将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：false。
6. useCache：（选填）将其设置为 true，将会导致本条语句的结果被二级缓存，默认值：对 select 元素为 true。
7. timeout：（选填）这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unset（依赖驱动）。
8. fetchSize：（选填）这是尝试影响驱动程序每次批量返回的结果行数和这个设置值相等。默认值为 unset（依赖驱动）。
9. statementType：（选填）STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
10. resultSets：（选填）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每个结果集给一个名称，名称是逗号分隔的。

在定义的SQL代码中#{name}为MyBatis中使用与便于参数的一种方式，大括号中的name是传入的参数名。

**Mapper接口的方式**

使用Mapper接口则没有了Mapper.xml配置文件。首先定义数据表相对应的Mapper接口。然后在接口中定义需要查询数据的方法，然后在方法上添加@Select注解。如果查询需要参数，则将参数定义进方法中。

**public** **interface** IUserMapper {

/\*\*

\* 定义根据用户的id值获取用户数据的方法

\*/

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

**public** User getUser(@Param("id")**int** id);

}

上图实现的查询结果是在基于JavaBean实体类属性与数据库表字段名一致的情况下能实现自动映射。如果实体类属性名与数据库表字段名不一致，则需要手动设置字段名与属性名之间的映射。此时需要用到@Results注解（功能与Mapper.xml中的<resultMap>标签一致）

**public** **interface** UserMapper {

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

@Results({@Result(property = "id", column = "id", id = **true**),

@Result(property = "name", column = "name"),

@Result(property = "pwd", column = "pwd"),

@Result(property = "createTime", column = "createTime")

})

**public** **void** getUser(@Param("id")**int** id);

}

在参数中写上id = true时就对应了<id>元素。MyBatis3.3.0及以前的版本中，注解定义的@Results不能共用，使用很不方便，需要在每个方法上都写一遍映射关系。但从MyBatis3.3.1版本开始，@Results注解增加了一个id属性，设置了id属性后，就可以通过id属性引用用一个@Results配置了。

**public** **interface** UserMapper {

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

@Results(id = "userMapper", value = {@Result(property = "id", column = "id", id = **true**),

@Result(property = "name", column = "name"),

@Result(property = "pwd", column = "pwd"),

@Result(property = "createTime", column = "createTime")

})

**public** **void** getUser(@Param("id")**int** id);

@Select("select \* from t\_user")

@ResultMap("userMapper")

**public** **void** getUsers();

}

通过使用@ResultMap注解引用设置的id值即可。

### 一对一查询

假设在权限系统中，一个用户只能拥有一个角色，则实现了一对一的关系。

**XML方式**

一个用户拥有一个角色，因此先在User类中添加role字段。

**public** **class** User {

**private** Role role;

**public** Role getRoles() {

**return** role;

}

**public** **void** setRoles(Role role) {

**this**.role = role;

}

}

在一对一查询中，在配置主外键之间的映射关系的时候，property属性的值为主表的

注意：虽然接口中的方法可以实现重载，但是Mapper接口中定义的方法名不允许重复。在同一个Mapper中只允许有唯一的方法名。