MyBatis

Pionpill [1]

本文档为作者归纳的 MyBatis 及 MyBatis Plus 笔记。

2022 年 10 月 13 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础 (操作系统，数据结构，计算机网络，编译原理等)，本人在撰写此笔记时已有 Java, MySQL 基础。

本文的参考书籍如下:

- MyBatis 部分: 《MyBatis 从入门到精通》刘增辉 2017 第一版。
  - 这是一本很好的带例子的入门书籍，建议看原文敲代码。
  - 本人并没有写后几章内容，分别是插件开发和框架集成，有兴趣请看原书。

本人的编写及开发环境如下：

- Java: Java11
- OS: Windows11
- MySQL: 8.0.3
- MyBatis: 3.5.11

2022 年 10 月 13 日

目录

第一部分

MyBatis

I MyBatis 基础

1 MyBatis 概述

1.1 MyBatis 简介

MyBatis 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。

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 有两种使用方式: 使用 jar 包和通过 maven 构建:

- 直接使用: 将 mybatis-x.x.x.jar 文件置于 classpath 中.
- maven 构建: 提供下列依赖:

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>x.x.x</version>
</dependency>
```

配置 Mybatis 的方法总的来说有三种:

- Java 硬编码: 直接用 Java 手动创建配置，这个方式比较繁琐不常用。
- Xml 配置: 通过 XML 文件让 Mybatis 解析，是主流的方法。
- 注解: 利用反射配置，在 Spring 等框架中常用。

其中第一种不讲，后两种在下两节中说明

2 MyBatis 的 XML 使用方式

2.1 使用纯 XML 方式

使用纯 XML 方式开发需要经历以下几个步骤:

- 配置: 编写 mybatis-config.xml 配置文件。
- 映射: 编写 mapper 文件，统一管理 sql 语句。
- 编码: 在程序中获取并执行 SqlSession 语句。

K)

2.1.1 mybatis-config 配置文件

使用 XML 形式进行配置，需要在 src/main/resources 下创建 mybatis-config.xml 文件[1]:

```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>
    <environments default="development">
        <environment id="development">
            <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="org/mybatis/example/BlogMapper.xml"/>
    </mappers>
</configuration>
```

其中，<environments> 配置了数据库连接，<mappers> 中包含了一个 MyBatis 的 SQL 语句和映射配置文件，我们需要在对应的路径下创建映射文件，默认的，使用 Mapper 作为结尾 (下文 mapper 文件就指的是 mybatis 的映射 xml 文件)。开发者可以根据需求修改或增添配置文件内容。

除此之外，我们还可以在 <configuration> 内添加下面这些元素:

```xml
<!-- 包别名，可以避免繁琐的完全限定名 -->
<typeAliases>
    <package name="learn.mybatis.simple.model"/>
</typeAliases>

<!-- 单独配置 sql 并引入 -->
<properties resource="sql.properties"/>

<!-- 使用日志 -->
<settings>
    <setting name="logImpl" value="LOG4J"/>
</settings>
```

2.1.2 mapper 映射文件

在对应的映射文件中，我们可以添加自定义 sql 语句, 下面是一个例子:

[1]该配置来自 MyBatis 官网，包含了必要的配置信息，详细标签: https://mybatis.net.cn/configuration.html

```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="org.mybatis.example.BlogMapper">
    <select id="selectAll" resultType="Country">
        select id, country_name, country_code from country
    </select>
</mapper>
```

其中，<mapper> 是根元素，后面紧跟当前 xml 的命名空间。<select> 代表这是一个查询操作，id 是唯一标识，resultType 对应我们自己写的实体类。相应的，我们还可以使用 update, delete 等其他元素，这里不一一介绍。

有了这些配置，我们就可以在代码中进行操作了:

```java
// 通过 xml 获取 SessionFactory
Reader reader = Resources.getResourcesAsReader("mybatis-config.xml")
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
reader.close();
// 通过 SessionFactory 使用已书写的 sql 语句
SqlSession sqlSession = sqlSessionFactory.openSession();
List<Country> countryList = sqlSession.selectList("selectAll");
sqlSession.close();
```

上述方式是完全使用 xml 来配置 Mybatis, 这种方式已经不常用了，下文不再说明，MyBatis3 提供了接口来调用方法。

2.2 Mapper 代理开发

Mapper 代理开发是指结合映射文件与接口的开发方式，它不需要通过字面值 (上文的 selectAll) 这种硬编码方式执行，安全性更高，更加面向接口编程，可以看作是 MyBatis3 对 XML 方式的一种改进，是目前主流的开发方式之一。

如果映射文件过多，我们还可以在 <mappers> 中使用 <package> 元素包含对应包下的所有映射文件。下面两种写法效果相同:

使用这种方式开发需要经历以下几步:

- 配置: 编写 mybatis-config.xml 配置文件，定义与映射文件名相同的 Mapper 接口，并且映射文件和接口文件必须在同一目录下。
- 映射: 设置映射文件的名称空间为 Mapper 接口的完全限定名。
- 接口: 在接口中定义方法，方法名就是映射文件中的 id, 保持参数类型和返回值类型一致。
- 编码: 通过 SqlSession 获取 Mapper 接口的代理对象并执行 sql 操作。

在配置过程中，使用代理开发的多个同目录的 mapper 映射可以改用 package 元素同一加载:

```xml
<mappers>
    <mapper resource="learn/mybatis/simple/mapper/UserMapper.xml"/>
    <mapper resource="learn/mybatis/simple/mapper/RoleMapper.xml"/>
    <mapper resource="learn/mybatis/simple/mapper/PrivilegeMapper.xml"/>
    <mapper resource="learn/mybatis/simple/mapper/UserRoleMapper.xml"/>
    <mapper resource="learn/mybatis/simple/mapper/RolePrivilegeMapper.xml"/>
</mappers>
<!-- 等效写法 -->
<mappers>
    <package name="learn/mybatis/simple/mapper"/>
</mappers>
```

这种配置方法会先查找对应 java 包下的所有接口，并加载接口对应的 XML 映射文件。同样，mapper 文件的 namespace 也需要修改:

```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="learn.mybatis.simple.mapper.PrivilegeMapper"/>
```

2.3 SELECT 用法

2.3.1 查询单个结果

首先的，我们需要在 UserMapper.java 接口中声明 SELECT 方法，例如:

```java
public interface UserMapper {
    SysUser selectById(long id);
}
```

然后在对应的 UserMapper.xml 中添加下面代码:

```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="learn.mybatis.simple.mapper.UserMapper">
    <resultMap id="userMap" type="learn.mybatis.simple.mapper.SysUser">
        <id property="id" column="id"/>
        <result property="userName" column="user_name"/>
        <result property="userPassword" column="user_password"/>
        <result property="userEmail" column="user_email"/>
        <result property="userInfo" column="user_info"/>
        <result property="headImg" column="head_img" jdbcType="BLOB"/>
        <result property="createTime" column="create_time" jdbcType="TIMESTAMP"/>
    </resultMap>

    <select id="selectById" resultMap="userMap">
        select * from sys_user where id = #{id}
    </select>
</mapper>
```

那么映射和接口文件/方法是怎么样绑定的呢?

- 文件: 通过映射文件的 namespace 设置为接口的全限定名进行关联。如果不使用接口，则 namespace 可以随便写。
- 方法: 通过 <select> 标签的 id 属性值绑定。

映射文件中重要的标签和属性:

- <select>: 映射查询语句使用的标签，id 属性与接口中的方法名对应。
- <resultMap>: 设置返回值的类型和映射关系。
- # id: MyBatis SQL 中使用预编译参数的一种方式，大括号中的 id 是传入的参数名。

其中 <resultMap> 是最为重要的标签，它所包含的标签如下:

- <constructor>: 配置使用构造方法注入结果，包含以下两个子标签。
  - <idArg>: id 参数，标记结果作为 id，可以帮助提高性能。
  - <arg>: 注入到构造方法的一个普通结果。
- <id>: 一个 id 结果，标记结果作为 id, 可以帮助提高性能。
- <result>: 注入到 Java 对象属性的普通结果。
- <association>: 一个复杂的类型关联，许多结果将包成这种类型。
- <collection>: 复杂类型的集合。
- <discriminator>: 根据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。

更多标签的作用请查看官网文档，这里不再说明。

在使用时，我们只需要获通过 sqlSession 的 getMapper 获取接口对应的对象即可。

```
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
userMapper.selectAll();
```

这里我们明明没有手动实现 UserMapper 接口，为什么就能用了呢? 当然是动态代理，具体实现方法另查资料。

2.3.2 查询多个结果

加入我们查询的结果不是单个对象而是对象集, 例如在 UserMapper.java 中添加如下方法:

```
List<SysUser> selectAll();
```

对应的映射文件需要添加如下代码:

```
<select id="selectAll" resultType="learn.mybatis.simple.model.SysUser">
    SELECT id,
           user_name    userName,
           user_password userPassword,
           user_email   userEmail,
           user_info    userInfo,
           head_img     headImg,
```

```
        create_time  createTime
    FROM sys_user
</select>
```

当返回结果大于 1 个时，必须使用 List<SysUser> 或 SysUser[] 作为返回类型，否则会抛 TooManyResultsException 异常。

此外，在映射文件中我们使用了 resultType 返回结果类型，配合别名可以实现类型的自动映射，而不需要手动设置 <resultMap>

名称映射规则 如前文所述，MyBatis 有两种名称映射规则:

- 通过 resultMap 手动设置结果类型。
- 通过 SQL 别名配合 resultType 实现类型自动映射。

注意，<result> 中的 property 要和对象中的属性名相同，而实际上，MyBatis 会将这两者都转换为大写进行判断。

我们知道，sql 语句是不区分大小写的，采用下划线命名，而 Java 属性一般采用小驼峰命名。MyBatis 十分贴心地为提供了一个全局属性 mapUnderscoreToCamelCase, 通过配置这个属性可以自动映射这两种命名。在 mybatis-config 文件中可以开启该属性:

```
<settings>
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

于是，我们就可以这样写了:

```
<select id="selectAll" resultType="learn.mybatis.simple.model.SysUser">
    SELECT id, user_name, user_password, user_email, user_info, head_img, create_time
    FROM sys_user
</select>
```

当然，也可以直接 select *, 但出于性能考虑，最好不要这样做。

2.3.3 处理复杂返回值

前面的例子返回的都是一个表中的单个或多个数据，如果返回的数据中包含不止一个表中的结果，那么应该如何处理 resultType 呢? 例如如下语句:

```
SELECT r.id, r.role_name, r.enabled, r.create_by, r.create_time, u.user_name, u.user_email
FROM sys_user AS u,
     sys_user_role AS ur,
     sys_role AS r
         INNER JOIN ur on u.id = ur.user_id
         INNER JOIN r on ur.role_id = r.id
WHERE u.id = #{userId}
```

这里有两种处理方式，一种是在 SysRole 对象中直接添加 userName 属性，这样仍然可以返回，但破坏了原对象，更推荐的方法是继承一个新的对象:

```java
public class SysRoleExtend extends SysRole{
    private String userName;
    // getter setter 方法
}
```

这种方式比较适合在需要少量字段时使用，但如果需要大量字段，就不适用了。

第二种方法，是直接让 SysRole 包含一个 SysUser 对象，然后修改对应的 XML 配置:

```java
public class SysRole {
    // 其他字段
    private SysUser user;
}
```

```xml
<select id="selectRolesByUserId" resultType="learn.mybatis.simple.model.SysRole">
    SELECT r.id, r.role_name, r.enabled, r.create_by, r.create_time, u.user_name as
        "user.userName", u.user_email as "user.userEmail"
    FROM sys_user AS u,
        sys_user_role AS ur,
        sys_role AS r
            INNER JOIN ur on u.id = ur.user_id
            INNER JOIN r on ur.role_id = r.id
    WHERE u.id = #{userId}
</select>
```

2.4 INSERT 用法

INSERT 的用法和 SELECT 类似:

```java
int insert(SysUser sysUser);
```

```xml
<insert id="insert">
    insert into sys_user(id, user_name, user_password, user_email, user_info, head_img,
        create_time)
    values (#{id}, #{userName}, #{userPassword}, #{userEmail}, #{userInfo}, #{headImg,
        jdbcType=BLOB}, #{createTime, jdbcType=TIMESTAMP})
</insert>
```

```java
@Test
public void testInsert() {
    SqlSession sqlSession = this.getSqlSession();
    try {
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        SysUser user = new SysUser();
        user.setUserName("test");
        user.setUserPassword("123456");
        user.setUserEmail("test@mybatis.com");
        user.setUserInfo("test info");
        user.setHeadImg(new byte[]{1,2,3});
        user.setCreateTime(new Date());
```

```
        int result = userMapper.insert(user);
        Assert.assertEquals(1, result);
        Assert.assertNull(user.getId());
    } finally {
        // 不影响其他测试，这里回滚
        sqlSession.rollback();
        sqlSession.close();
    }
}
```

这里有两个要点，一是在映射文件中，为了防止类型错误，对于一些特殊的数据类型，建议指定具体的 jdbcType 值。例如 headImg 对应 BLOB, createTime 对应 TIMESTAMP。

- BLOB 对应的类型是 ByteArrayInputStream，就是二进制数据流。
- date、time、datetime 对应的 JDBC 类型分别为 DATE、TIME、TIMESTAMP。

此外的，我们会发现 insert 的默认返回值类型为 int，也即插入的数据条数。

有些情况下，我们需要指定返回值以方便后续操作，例如主键自增的情况下，可以这样编写映射文件:

```
<insert id="insert" useGeneratedKeys="true" keyProperty="id">
    <!-- 由于主键自增，删除insert 中的 id -->
</insert>
```

useGeneratedKeys 设置为 true 后，MyBatis 会使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键。获得主键值后将其赋值给 keyProperty 配置的 id 属性。当需要设置多个属性时，使用逗号隔开，这种情况下通常还需要设置 keyColumn 属性，按顺序指定数据库的列，这里列的值会和 keyProperty 配置的属性一一对应。

但这种写法只适用于支持自增的数据库，比如 Oracle 就不支持主键自增，而是使用序列获取 id。对此可以使用另一种方法 <selectKey>，可以处理自增和序列赋值两种情况。

```
<insert id="insert3">
    insert into sys_user(user_name, user_password, user_email, user_info, head_img,
        create_time)
    values (#{userName}, #{userPassword}, #{userEmail}, #{userInfo}, #{headImg,
        jdbcType=BLOB}, #{createTime, jdbcType=TIMESTAMP}
    <selectKey keyColumn="id" resultType="long" keyProperty="id" order="AFTER">
        SELECT LAST_INSERT_ID()
    </selectKey>
</insert>
```

其他属性不再说明，order 属性的设置和使用的数据库有关。在 MySQL 数据库中，order 属性设置的值是 AFTER，因为当前记录的主键值在 insert 语句执行成功后才能获取到。而在 Oracle 数据库中，order 的值要设置为 BEFORE，这是因为 Oracle 中需要先从序列获取值，然后将值作为主键插入到数据库中。

MySQL 中的 SQL 语句 SELECT LAST_INSERT_ID（）用于获取数据库中最后插入的数据的 ID 值。其他数据对应的语句需要自行查阅。

2.5 UPDATE 与 DELETE 用法

UPDATE 用法和前面类似，这里只给出映射文件片段:

```
<update id="updateById">
    update sys_user
    set user_name    = #{userName},
        user_password = #{userPassword},
        user_email   = #{userEmail},
        user_info    = #{userInfo},
        head_img     = #{headImg, jdbcType=BLOB},
        create_time=#{createTime, jdbcType=TIMESTAMP}
    where id = #{id}
</update>
```

DELETE 更简单:

```
<delete id="deleteById">
    delete from sys_user where id = #{id}
</delete>
```

2.6 多个接口参数的用法

参数的类型可以分为两种: 一种是基本数据类型，一种是 JavaBean。

- 基本类型: 在 XML 文件中对应的 SQL 语句只会使用一个参数，例如 delete 方法。
- JavaBean: 在 XML 文件中对应的 SQL 语句会有多个参数，例如 insert, update 方法。

之前我们都是将多个参数合并到一个 JavaBean 中，并使用这个 JavaBean 作为接口方法的参数。参数较少的时候，为此创建一个新的 JavaBean 就显得浪费，有两种方式解决这个问题: 使用 Map 类型作为参数或者使用 @Param 注解。

使用 Map 类型作为参数的方法，就是在 Map 中通过 key 来映射 XML 中 SQL 使用的参数值名字，value 用来存放参数值，需要多个参数时，通过 Map 的 key-value 方式传递参数值，由于这种方式还需要自己手动创建 Map 以及对参数进行赋值，其实并不简洁，所以对这种方式掠过。

首先对不加 @param 的代码进行测试:

```
List<SysRole> selectRolesByUserIdAndRoleEnabled(Long userId, Integer enabled);
```

```
<select id="selectRolesByUserIdAndRoleEnabled"
     resultType="learn.mybatis.simple.model.SysRole">
    select r.id, r.role_name, r.enabled, r.create_by, r.reate_time
    from sys_user as u,
        sys_user_role as ur,
        sys_role as r
            inner join ur on u.id = ur.user_id
            inner join r on ur.role_id = r.id
```

```
    where u.id = #{userId}
        and r.enabled = #{enabled}
</select>
```

对他进行测试会显示如下错误:

```
Parameter 'userId' not found
Available parameters are [0,1,param1,param2]
```

这时如果将 XML 中的＃ userId 改为＃ 0 或＃ param1，将＃ enabled 改为＃ 1 或＃ param2，这个方法就可以被正常调用了。但并不推荐这样做

正确的方法应该是在接口中加入 @Param 注解:

```
List<SysRole> selectRolesByUserIdAndRoleEnabled(
    @Param("userId") Long userId,
    @Param("enabled") Integer enabled);
```

这时的 XML 文件中对应的 SQL 的可用参数变成了 [userId，enabled，param1，param2]，如果把＃ userId 改为＃ param1，把＃ enabled 改为＃ param2，测试也可以通过。

给参数配置 @Param 注解后，MyBatis 就会自动将参数封装成 Map 类型，@Param 注解值会作为 Map 中的 key，因此在 SQL 部分就可以通过配置的注解值来使用参数。

当只有一个参数（基本类型或拥有 TypeHandler 配置的类型）的时候，为什么可以不使用注解？这是因为在这种情况下（除集合和数组外），MyBatis 不关心这个参数叫什么名字就会直接把这个唯一的参数值拿来使用。

但！如果包含多个 JavaBean 对象，比如:

```
List<SysRole> selectRolesByUserIdAndRoleEnabled(
    @Param("user") SysUser user,
    @Param("role") SysRole role);
```

就不能直接在映射文件中使用 #userId 和 #enabled，毕竟没法确定这些属性只有一个，需要使用 #user.id 和 #role.enabled 从两个 JavaBean 中去除指定属性的值。

3 MyBatis 的注解使用方式

MyBatis 注解方式就是将 SQL 语句直接写在接口上。这种方式的优点是，对于需求比较简单的系统，效率较高。缺点是，当 SQL 有变化时都需要重新编译代码。更全面地使用注解开发应该学习 MyBatis-Plus。

3.1 @Select 注解

注解有多方便:

```
public interface RoleMapper {
    @Select("SELECT id, role_name, enabled, create_by, create_time FROM sys_role WHERE id =
        #{id}")
    SysRole selectById(Long id);
}
```

然后就可以直接在测试类中运行了。

之前我们提到过 XML 配置有个问题: 如何实现字段映射? 同样有三种方式:

- SQL 语句使用别名;
- mybatis-plus 配置文件中开启 mapUnderscoreToCamelCase 配置。
- resultMap 方式。

前两种没什么好说的，和之前的操作一样，最后一种要用到 @Result 注解:

```
@Results(id = "roleResultMap", value = {
        @Result(property = "id", column = "id", id = true),
        @Result(property = "roleName", column = "role_name"),
        @Result(property = "enabled", column = "enabled"),
        @Result(property = "createBy", column = "create_by"),
        @Result(property = "createTime", column = "create_time")
})
@Select("SELECT id, role_name, enabled, create_by, create_time FROM sys_role WHERE id =
    #{id}")
SysRole selectById(Long id);
```

有一个 id 属性是用来复用注解的。在 MyBatis3.3.1 及后续版本，可以使用 id 来服用 Results。

```
@ResultMap("roleResultMap")
@Select("select * from sys_role")
List<SysRole> selectAll();
```

同时，如果 XML 配置了 resultMap，还可以根据对应的 id 获取。

3.2 @Insert 注解

Insert 本身是简单的，主要是返回值比较复杂。前面的会了，这里看一下例子就行了:

```
// 返回插入数据量
@Insert("INSERT INTO sys_role(id, role_name, enabled, create_by, create_time) VALUES (#{id},
    #{roleName}, #{enabled}, #{createBy}, #{createTime jdbcType=TIMESTAMP})")
int insert(SysRole sysRole);

// 返回主键，方法 1
@Insert("INSERT INTO sys_role(role_name, enabled, create_by, create_time) VALUES #{roleName},
    #{enabled}, #{createBy}, #{createTime jdbcType=TIMESTAMP})")
@Options(useGeneratedKeys = true, keyProperty = "id")
int insert2(SysRole sysRole);

// 返回主键，方法 2
@Insert("INSERT INTO sys_role(role_name, enabled, create_by, create_time) VALUES #{roleName},
    #{enabled}, #{createBy}, #{createTime jdbcType=TIMESTAMP})")
@SelectKey(statement = "SELECT LAST_INSERT_ID()", keyProperty = "id", resultType =
    Long.class, before = false)
int insert3(SysRole sysRole);
```

3.3 @Update 和 @Delete 注解

这两种更简单了:

```
@Update("UPDATE sys_role SET role_name = #{roleName}, enable = #{enabled}, create_by =
    {createBy}, create_time = #{createTime, jdbcType = TIMESTAMP} WHERE id = #{id}")
int updateById(SysRole sysRole);

@Delete("DELETE FROM sys_role WHERE id = #{id}")
int deleteById(Long id);
```

3.4 Provider 注解

除了上面 4 中注解可以使用简单的 SQL 外，MyBatis 还提供了 4 中 Provider 注解，分别是 @SelectProvoder, @DeleteProvider, @UpdateProvider, @DeleteProvider。

下面来看看 @SelectProvider 的使用方法:

```
// 定义一个新 Provider 类
public class PrivilegeProvider {
    public String selectById(final Long id) {
        return new SQL(){
            {
                SELECT("id, privilege_name, privilege_url");
                FROM("sys_privilege");
                WHERE("id = #{id}");
```

```
            }
        }.toString();
    }
}
```

```
// 使用 @SelectProvider
@SelectProvider(type = PrivilegeProvider.class, method = "selectById")
SysPrivilege selectById(Long id);
```

Provider 提供了两个属性 type 和 method:

- type: 包含 method 方法的类，这个类必须有空的构造方法，这个方法的值就是要执行的 SQL 语句。
- method: 对应方法的放回值必须是 String 类型。

前面的 method 语法比较复杂，但又代码提示，还可以直接写字符串，效果是一样的:

```
public String selectById2(final Long id) {
    return "SELECT id, privilege_name, privilege_url FROM sys_privilege WHERE id = #{id}";
}
```

说白了 Provider 就是将 SQL 语句卸载单独的类中。

II MyBatis 核心

4 MyBatis 动态 SQL

MyBatis3 采用了功能强大的 OGNL（Object-Graph Navigation Language）表达式语言消除了许多其他标签。以下是 MyBatis3 的动态 SQL 在 XML 中支持的几种标签: if, choose(when, otherwise), trim(where, set), foreach, bind。

4.1 if 用法

if 标签通常用于 WHERE，UPDATE，INSERT 语句中用于判断字段。

WHERE 中的 if

如果我们有这样的需求: 实现一个用户管理高级查询功能，根据输入的条件去检索用户信息。

- 当只输入用户名时，需要根据用户名进行模糊查询;
- 当只输入邮箱时，根据邮箱进行完全匹配;
- 当同时输入用户名和邮箱时，用这两个条件去查询匹配的用户。

如果仅使用 MySQL 的语法，无法在一句语句中完成三种条件下的需求，需要通过 Java 程序进行判断后选择合适的 SQL 语句执行。但 MyBatis 使用 if 标签可以在一句语句中完成该任务。

```
<select id="selectByUser" resultType="learn.mybatis.simple.model.SysUser">
    SELECT id, user_name, user_password, user_email, user_info, head_img, create_time
    FROM sys_user
    WHERE 1 = 1
    <if test="userName != null and userName !=''">
        AND user_name like concat('%',#{userName},'%')
    </if>
    <if test="userEmail != null and userEmail != ''">
        AND user_email = #{userEmail}
    </if>
</select>
```

test 的属性值是一个符合 OGNL 要求的判断表达式，表达式的结果可以是 true 或 false，除此之外所有的非 0 值都为 true，只有 0 为 false。建议只用 true 或 false 作为结果。

OGNL 的详细用法会在下文介绍，判断条件 property == null 对任何类型字段都适用，property == '' 仅适用于 String 类型的字段。当有多个判断条件时可以用 and 或 or 进行连接。

注意 1=1 的条件可以避免 SQL 语法错误导致的异常，这种写法将在下文被取代。

UPDATE 中的 if

现在要实现这样一个需求：只更新有变化的字段。需要注意，更新的时候不能将原来有值但没有发生变化的字段更新为空或 null。通过 if 标签可以实现这种动态列更新。

一般情况下，MyBatis 中选择性更新的方法名会以 Selective 作为后缀。在 UserMapper.xml 中添加对应的 SQL 语句，代码如下:

```
<update id="updateByIdSelective">
    UPDATE sys_user
    SET
    <if test="userName != null and userName != ''">
        user_name = #{userName},
    </if>
    <if test="userPassword != null and userPassword != ''">
        user_password = #{userPassword},
    </if>
    <if test="userEmail != null and userEmail != ''">
        user_email = #{userEmail},
    </if>
    <if test="userInfo != null and userInfo != ''">
        user_info = #{userInfo},
    </if>
    <if test="headImg != null">
        head_img = #{headImg, jdbcType = BLOB},
    </if>
    <if test = "createTime != null">
        create_time = #{createTime jdbcType = TIMESTAMP},
    </if>
    id = #{id}
    WHERE id = #{id}
</update>
```

只要注意一下书写顺序，符不符合 sql 语句规范即可。

insert 中的 if

在数据库表中插入数据的时候，如果某一列的参数值不为空，就使用传入的值，如果传入参数为空，就使用数据库中的默认值（通常是空），而不使用传入的空值。使用 if 就可以实现这种动态插入列的功能。

```
<insert id="insert2" useGeneratedKeys="true" keyProperty="id">
    INSERT INTO sys_user (user_name, user_password,
        <if test="userEmail != null and userEmail != ''">
            user_email,
        </if>
        user_info, head_img, create_time)
    VALUES (
```

```
        #{userName}, #{userPassword},
        <if test="userEmail != null and userEmail != ''">
            #{user_email},
        </if>
        #{userEmail}, #{headImg jdbcType=BLOB}, #{createTime jdbcType = TIMESTAMP}
    )
</insert>
```

4.2 choose 用法

<if> 标签只能实现基本的条件判断，无法实现 if else 逻辑。<choose> 标签包含两个标签:

- <when>: 至少一个
- <otherwise>: 0 个或 1 个

现在进行如下查询：当参数 id 有值的时候优先使用 id 查询，当 id 没有值时就去判断用户名 (假定唯一) 是否有值，如果有值就用用户名查询，如果用户名也没有值，就使 SQL 查询无结果。

```
<select id="selectByIdOrUserName" resultType="learn.mybatis.simple.model.SysUser">
    SELECT id, user_name, user_password, user_email, user_info, head_img, create_time
    FROM sys_user
    WHERE 1 = 1
    <choose>
        <when test="id != null">
            AND id = #{id}
        </when>
        <when test="userName != null and userName != ''">
            AND user_name = #{userName}
        </when>
        <otherwise>
            AND 1=2
        </otherwise>
    </choose>
</select>
```

4.3 where, set, trim 用法

这三个标签都是用来解决一些语法问题的，可以让我们避免写比不要的 SQL 子句。

where 用法

where 标签的作用：如果该标签包含的元素中有返回值，就插入一个 where; 如果 where 后面的字符串是以 AND 和 OR 开头的，就将它们 (AND/OR) 剔除，非常智能。

我们修改之前的 selecyByUser 映射代码:

```xml
<select id="selectByUser2" resultType="learn.mybatis.simple.model.SysUser">
    SELECT id, user_name, user_password, user_email, user_info, head_img, create_time
        FROM sys_user
        <where>
            <if test="userName != null and userName != ''">
                AND user_name LIKE CONCAT('%',#{userName},'%')
            </if>
            <if test="userEmail != null and userEmail != ''">
                AND user_email = #{userEmail}
            </if>
        </where>
</select>
```

有了 <WHERE> 标签，我们就不需要 WHERE 1=1 这样的子句。

set 用法

set 标签的作用：如果该标签包含的元素中有返回值，就插入一个 set; 如果 set 后面的字符串是以逗号结尾的，就将这个逗号剔除。

```xml
<update id="updateByIdSelective2">
    UPDATE sys_user
    <set>
        <if test="userName != null and userName != ''">
            user_name = #{userName},
        </if>
        <if test="userPassword != null and userPassword != ''">
            user_password = #{userPassword},
        </if>
        <if test="userEmail != null and userEmail != ''">
            user_email = #{userEmail},
        </if>
        <if test="userInfo != null and userInfo != ''">
            user_info = #{userInfo},
        </if>
        <if test="headImg != null">
            head_img = #{headImg, jdbcType = BLOB},
        </if>
        <if test="createTime != null">
            create_time = #{createTime jdbcType = TIMESTAMP},
        </if>
        id = #{id},
    </set>
    WHERE id = #{id}
</update>
```

trim 用法

where 和 set 标签的功能都可以用 trim 标签来实现，并且在底层就是通过 TrimSqlNode 实现的。

<WHERE> 标签对应的 <trim> 实现如下:

```
<trim prefix="WHERE" prefixOverrides="AND ||OR ">
...
</trim>
```

这里的 AND 和 OR 后面的空格不能省略，为了避免匹配到 andes、orders 等单词。实际的 prefixeOverrides 包含“AND”、“OR”、“AND\n”、“OR\n”、“AND\r”、“OR\r”、“AND\t”、“OR\t”，不仅仅是上面提到的两个带空格的前缀。

<SET> 标签对应的 <trim> 实现如下:

```
<trim prefix ="SET" suffixOverrides=",">
...
</trims>
```

<trim> 标签有如下属性:

- prefix: 当 teim 元素被包含内容时，会给内容增加 prefix 指定的前缀。
- prefixOverrides: 当 trim 元素内包含内容时，会把内容中匹配的前缀字符串去掉。
- suffix: 当 trim 元素内包含内容时，会给内容增加 suffix 指定的后缀。
- suffixOverrides：当 trim 元素内包含内容时，会把内容中匹配的后缀字符串去掉。

4.4 foreach 用法

foreach 可以对数组、Map 或实现了 Iterable 接口（如 List、Set）的对象进行遍历。数组在处理时会转换为 List 对象，因此 foreach 遍历的对象可以分为两大类：Iterable 类型和 Map 类型。

foreach 实现 in 集合

foreach 实现 in 集合（或数组）是最简单和常用的一种情况，下面介绍如何根据传入的用户 id 集合查询出所有符合条件的用户。

```
List<SysUser> selectByIdList(List<Long> idList);
```

```
<select id="selectByIdList" resultType="learn.mybatis.simple.model.SysUser">
    SELECT id, user_name, user_password, user_email, user_info, head_img, create_time
    FROM sys_user
    WHERE id in
    <foreach collection="list" open="(" close = ")" separator="," item="id" index="i">
        #{id}
```

```
    </foreach>
</select>
```

foreach 包含以下属性:

- collection: 必填，值为要迭代循环的属性名。
- item: 变量名，值为从迭代对象中取出的每一个值。
- index: 索引的属性名，在集合数组情况下值为当前索引值，当迭代对象是 map 时，值为 key。
- open, close: 整个循环内容开头和结尾的字符串。
- separator: 每次循环的分隔符。

下面看一下 MyBatis 是如何处理 collection 的:

以下是 DefaultSqlSession 中的方法，也是默认情况下的处理逻辑:

```
public static Object wrapToMapIfCollection(Object object, String actualParamName) {
    if (object instanceof Collection) {
        ParamMap<Object> map = new ParamMap<>();
        map.put("collection", object);
        if (object instanceof List) {
            map.put("list", object);
        }
        Optional.ofNullable(actualParamName).ifPresent(name -> map.put(name, object));
        return map;
    } else if (object != null && object.getClass().isArray()) {
        ParamMap<Object> map = new ParamMap<>();
        map.put("array", object);
        Optional.ofNullable(actualParamName).ifPresent(name -> map.put(name, object));
        return map;
    }
    return object;
}
```

当参数类型为集合的时候，默认会转换为 Map 类型，并添加一个 key 为 collection 的值，如果参数类型是 List 集合，那么就继续添加一个 key 为 list 的值，这样，当 collection=”list” 时，就能得到这个集合，并对它进行循环操作。

当参数类型为数组的时候，也会转换成 Map 类型，默认的 key 为 array。当采用如下方法使用数组参数时，就需要把 foreach 标签中的 collection 属性值设置为 array。

foreach 实现批量插入

批量插入，说人话，就是 VALUES 中包含多个元组。VALUES 后的元组是同一类元组的循环，所以可以使用 foreach 实现循环插入。

```
int insertList(List<SysUser> userList);
```

```
<insert id="insertList">
    INSERT INTO sys_user (user_name, user_password, user_email, user_info, head_img,
```

```
        create_time)
    VALUES
    <foreach collection="list" item="user" separator=",">
        (#{user.userName}, #{user.userPassword}, #{user.userEmail}, #{user.userInfo},
        #{user.headImg, jdbcType=BLOB}, #{user.createTime, jdbcType=TIMESTAMP})
    </foreach>
</insert>
```

通过 item 指定了循环变量名后，在引用值的时候使用的是“属性. 属性”的方式，如 user.userName。

foreach 动态实现 UPDATE

当参数是 Map 类型的时候，foreach 标签的 index 属性值对应的不是索引值，而是 Map 中的 key，利用这个 key 可以实现动态 UPDATE。

现在需要通过指定的列名和对应的值去更新数据，实现代码如下。

```
<update id="updateByMap">
    UPDATE sys_user SET
    <foreach collection="_parameter" item="val" index="key" separator=",">
        ${key} = #{val}   <!-- ${} 传字符串 -->
    </foreach>
    WHERE id = #{id}
</update>
```

这里的 key 作为列名，对应的值作为该列的值，通过 foreach 将需要更新的字段拼接在 SQL 语句中。

```
int updateByMap(Map<String, Obejct> map);
```

这里没有通过 @Param 注解指定参数名，因而 MyBatis 在内部的上下文中使用了默认值 _parameter 作为该参数的 key，所以在 XML 中也使用了 _parameter 。

4.5 bind 用法

bind 标签可以使用 OGNL 表达式创建一个变量并将其绑定到上下文中。在前面的例子中，UserMapper.xml 有一个 selectByUser 方法，这个方法用到了 like 查询条件，部分代码如下。

```
<if test="userName != null and userName != ''">
    AND user_name LIKE CONCAT('%',#{userName},'%')
</if>
```

使用 concat 函数连接字符串，在 MySQL 中，这个函数支持多个参数，但在 Oracle 中只支持两个参数。由于不同数据库之间的语法差异，如果更换数据库，有些 SQL 语句可能就需要重写。针对这种情况，可以使用 bind 标签来避免由于更换数据库带来的一些麻烦。

```
<if test="userName != null and userName != ''">
```

```
    <bind name="userNameLike" value="'%' + userName + '%'"/>
    AND user_name LIKE #{userNameLike}
</if>
```

bind 标签的两个属性都是必选项，name 为绑定到上下文的变量名，value 为 OGNL 表达式。

4.6 多数据库支持

bind 标签并不能解决更换数据库带来的所有问题，那么还可以通过什么方式支持不同的数据库呢? 这需要用到 if 标签以及由 MyBatis 提供的 databaseIdProvider 数据库厂商标识配置。

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 属性的。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 配置即可。

```
<databaseIdProvider type="DB_VENDOR">
    <property name="SQL Server" value="sqlserver"/>
    <property name="DB2" value="db2"/>
    <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

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，所以通常会通过设置属性别名来使其变短。

除了增加上面的配置外，映射文件也是要变化的。举一个简单例子来介绍 databaseId 属性如何使用。针对 MySQL 和 Oracle 数据库提供下面两个不同版本的 like 查询方法。

```
<select id="selectByUser" databaseId="mysql"
        resultType="learn.mybatis.simple.model.SysUser">
    select * from sys_user where user_name like concat ('%',#{userName},'%')
</select>
<select id="selectByUser" databaseId="oracle"
        resultType="learn.mybatis.simple.model.SysUser">
    select * from sys_user
    where user_name like '%'||#{userName}||'%'
</select>
```

当基于不同数据库运行时，MyBatis 会根据配置找到合适的 SQL 去执行。

也可以这样用:

```
<if test="_databaseId == 'mysql'">
...
</if>
```

4.7 OGNL 简单用法

OGNL（Object-Graph Navigation Language）对象图导航语言。在 MyBatis 的动态 SQL 和 $ 形式的参数中都用到了 OGNL 表达式

常用的 OGNL 表达式如下:

- 逻辑: and, or, !, not
- 比较: ==, !=, eq, neq, lt, lte, gt, gte
- 四则运算: +, -, *, /,
- 对象调用: e.method(args), e.property
- 类 (静态) 调用: @class@method, @class@field
- 索引取值: e1[e2]

5 MyBatis 代码生成器

作为一个优秀的程序员，“懒”是很重要的优点。不仅要会写代码，还要会利用（或自己实现）工具生成代码。MyBatis 的开发团队提供了一个很强大的代码生成器——MyBatis Generator(MBG) 来帮助我们缩减 SQL 语句。

MBG 的版本和 MyBatis 的版本没有直接关系，不同的 MBG 版本包含的参数可能不一样。

5.1 使用 MyBatis Generator

5.1.1 配置文件

在 src/main/resources 中创建 generator-config.xml 内容如下:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <properties resource="sql.properties"/>
    <context id="MySqlContext" targetRuntime="MyBatis3Simple" defaultModelType="flat">
        <property name="beginningDelimiter" value="`"/>
        <property name="endingDelimiter" value="`"/>

        <commentGenerator>
            <property name="suppressDate" value="true"/>
            <property name="addRemarkComments" value="true"/>
        </commentGenerator>

        <jdbcConnection driverClass="${driver}" connectionURL="${url}" userId="${username}"
            password="${password}">
        </jdbcConnection>

        <javaModelGenerator targetPackage="test.model" targetProject="src\main\java">
            <property name="trimStrings" value="true"/>
        </javaModelGenerator>

        <sqlMapGenerator targetPackage="test.xml" targetProject="src\main\resources"/>

        <javaClientGenerator type="XMLMAPPER" targetPackage="test.dao"
            targetProject="src\main\java"/>

        <table tableName="%">
            <generatedKey column="id" sqlStatement="MySql"/>
        </table>
    </context>
</generatorConfiguration>
```

先不管这东西是干嘛的，先运行起来看看怎么用。

5.1.2 运行 MyBatis Generaotr

MBG 提供了多种运行方式，各有优缺点，常用的有以下几种:

- 使用 Java 编写运行代码。
- 从命令提示符运行。
- 使用 MavenPlugin 运行。
- 使用 IDE 插件运行。

这里只介绍第一种方法，其他三种自行查阅资料。

在运行前，需要将对应的 jar 包添加到项目中，可以直接下载 jar 包或者通过 maven 引入依赖。

```
<dependency>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-core</artifactId>
    <version>1.3.3</version>
</dependency>
```

然后创建对应的 Generator 类:

```
public class Generator {
    public static void main(String[] args) throws Exception {
        //MBG 执行过程中的警告信息
        List<String> warnings = new ArrayList<String>();
        //当生成的代码重复时，覆盖原代码
        boolean overwrite = true;
        //读取我们的 MBG 配置文件
        InputStream is =
            Generator.class.getResourceAsStream("/generator/generatorConfig.xml");
        ConfigurationParser cp = new ConfigurationParser(warnings);
        Configuration config = cp.parseConfiguration(is);
        is.close();

        DefaultShellCallback callback = new DefaultShellCallback(overwrite);
        //创建 MBG
        MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config, callback, warnings);
        //执行生成代码
        myBatisGenerator.generate(null);
        //输出警告信息
        for(String warning : warnings){
            System.out.println(warning);
        }
    }
}
```

使用 Java 编码方式运行的好处是，generatorConfig.xml 配置的一些特殊的类（如 commentGenerator 标签中 type 属性配置的 MyCommentGenerator 类）只要在当前项目中，或者在当前项目的 classpath 中，就可以直接使用。使用其他方式时都需要特别配置才能在 MBG 执行过程中找到 MyCommentGenerator 类并实例化，否则都会由于找不到这个类而抛出异常。

使用 Java 编码不方便的地方在于，它和当前项目是绑定在一起的，在 Maven 多子模块的情况下，可能需要增加编写代码量和配置量，配置多个，管理不方便。但是综合来说，这种方式出现的问题最少，配置最容易，因此推荐使用。

5.2 XML 配置详解

配置文件必备的基本信息如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
...
</generatorConfiguration>
```

其中，头文件确定了对应的 DTD 文件用来规范格式。generatorConfiguration 是根节点，他下面有三个子标签: 分别是 properties、classPathEntry 和 context。这三个标签的顺序必须与列举的顺序一致。

properties 标签用于指定一个需要解析的外部属性文件，引入属性文件后，后面需要配置的 JDBC 信息可以通过 ${property} 的形式引入。这个标签最多可以配置 1 个，也可以不配置。properties 标签包含 resource 和 url 两个属性，只能使用其中一个属性来指定，同时出现则会报错。

- resource: 指定 classpath 下的属性文件，类似 com/myproject/generatorConfig.properties 这样的属性值。
- url: 指定文件系统上的特定位置，例如 file:///C:/myfolder/generatorConfig.properties。

第二个是 classPathEntry 标签。这个标签可以配置多个，也可以不配置。classPathEntry 标签最常见的用法是通过属性 location 指定驱动的路径，代码如下。

```
<classPathEntry location="E:\mysql\mysql-connector-java-5.1.29.jar"/>
```

第三个是 context 标签。这个标签是最重要的，该标签至少配置 1 个，可以配置多个。context 标签用于指定生成一组对象的环境。例如指定要连接的数据库，要生成对象的类型和要处理的数据库中的表。运行 MBG 的时候还可以指定要运行的 context。context 标签只有一个必选属性 id，用来唯一确定该标签，此外还有几个可选属性:

- defaultModelType: 定义 MBG 如何生成实体类，有以下选项:
  - defaultModelType: 和 hierarchical 类似，如果一个表的主键只有一个字段，那么不会为该字段生成单独的实体类，而是会将该字段合并到基本实体类中。
  - flat: 该模型只为每张表生成一个实体类。这个实体类包含表中的所有字段。这种模型最简单，推荐使用。
  - hierarchical：如果表有主键，那么该模型会产生一个单独的主键实体类，如果表还

有 BLOB 字段，则会为表生成一个包含所有 BLOB 字段的单独的实体类，然后为所有其他的字段另外生成一个单独的实体类。MBG 会在所有生成的实体类之间维护一个继承关系。

- targetRuntime：此属性用于指定生成的代码的运行时环境，有以下选项:
  - MyBatis3: 默认值。
  - MyBatis3Simple: 这种情况不会生成与 Example 相关的方法。
- introspectedColumnImpl: 该参数可以指定扩展 Introspected Column 类的实现类。

MBG 配置中的其他几个标签基本上都是 context 的子标签，这些子标签（有严格的配置顺序，后面括号中的内容为这些标签可以配置的个数）包括以下几个: property(0+), plugin(0+), commentGenerator(0/1), jdbcConnection(1), javaTypeResolver(0/1), javaModelGenerator(1), sqlMapGenerator(0/1), javaClientGenerator(0/1), table(1+)。

5.2.1 配置标签

property 标签

假设数据库中有一个表，名为 user info，中间有一个空格。在 MySQL 中可以使用反单引号“‘”作为分隔符，例如 ‘user info‘，在 SQL Server 中则是 [user info]。

通过分隔符可以将其中的内容作为一个整体的字符串进行处理，当 SQL 中有数据库关键字时，使用反单引号括住关键字，可以避免数据库产生错误。

property 标签中包含了以下 3 个和分隔符相关的属性。

- autoDelimitKeywords: 自动给关键字添加分隔符，MBG 中维护了一个关键字列表，当数据库的字段或表与这些关键字一样时，MBG 会自动给这些字段或表添加分隔符。
- beginningDelimiter: 配置前置分隔符的属性。
- endingDelimiter: 配置后置分隔符的属性。

```
<property name="autoDelimitKeywords" value="true"/>
<property name="beginningDelimiter" value="`"/>
<property name="endingDelimiter" value="`"/>
```

plugin 标签

plugin 标签用来定义一个插件，用于扩展或修改通过 MBG 生成的代码。该插件将按在配置中配置的顺序执行。MBG 插件使用的情况并不多。

以缓存插件 org.mybatis.generator.plugins.CachePlugin 为例。这个插件可以在生成的 SQL XML 映射文件中增加一个 cache 标签。只有当 targetRuntime 为 MyBatis3 时，该插件才有效。

可以这样进行配置 (具体的插件属性不提及):

```
<plugin type="org.mybatis.generator.plugins.CachePlugin">
    <property name="cache_eviction" value="LRU"/>
    <property name="cache_size" value="1024"/>
```

```
</plugin>
```

增加这个配置后，生成的 Mapper.xml 文件中会增加如下的缓存相关配置:

```
<cache eviction="LRU" size="1024">
...
</cache>
```

commentGenerator 标签

该标签用来配置如何生成注释信息，该标签有一个可选属性 type，可以指定用户的实现类，该类需要实现 org.mybatis.generator.api.CommentGenerator 接口，而且必有一个默认空的构造方法。type 属性接收默认的特殊值 DEFAULT，使用默认的实现类 DefaultCommentGenerator。

默认的实现类中提供了三个可选属性，需要通过 property 属性进行配置。

- suppressAllComments: 阻止生成注释，默认为 false;
- suppressDate: 阻止生成的注释包含时间戳，默认为 false;
- addRemarkComments: 注释是否添加数据库表的备注信息，默认为 false;

一般情况下，由于 MBG 生成的注释信息没有任何价值，而且有时间戳的情况下每次生成的注释都不一样，使用版本控制的时候每次都会提交，因而一般情况下都会屏蔽注释信息，可以如下配置。

```
<commentGenerator>
    <property name="suppressDate" value="true"/>
    <property name="addRemarkComments" value="true"/>
</commentGenerator>
```

自定义注释可以参考这篇文章: `https://blog.csdn.net/u011781521/article/details/78161201`

jdbcConnection 标签

这个是用于连接数据库的标签，基础的四个属性不在说明。该标签还可以接受多个 property 子标签，这里配置的 property 属性都会添加到 JDBC 驱动的属性中。具体的值要看数据库的支持而定。

javaTypeResolver 标签

该标签的配置用来指定 JDBC 类型和 Java 类型如何转换，最多可以配置一个。

该标签提供了一个可选的属性 type。另外，和 commentGenerator 类似，该标签提供了默认的实现 DEFAULT，一般情况下使用默认即可，需要特殊处理的情况可以通过其他标签配置来解决，不建议修改该属性。

javaModelGenerator 标签

该标签用来控制生成的实体类，根据 context 标签中配置的 defaultModelType 属性值的不同，一个表可能会对应生成多个不同的实体类。该标签只有两个必选属性:

- targetPackage: 生成实体类存放的包名。一般就是放在该包下，实际还会受到其他配置的影响。
- targetProject：指定目标项目路径，可以使用相对路径或绝对路径。

该标签还支持以下几个 property 子标签属性:

- constructorBased: 如果设置为 true 使用构造方法入参，默认为 false，使用 setter 方式入参。
- enableSubPackages: 如果为 true，MBG 会根据 catalog 和 schema 来生成子包。如果为 false 就会直接使用 targetPackage 属性。默认为 false。
- immutable: 用来配置实体类属性是否可变。如果设置为 true，那么 constructorBased 不管设置成什么，都会使用构造方法入参，并且不会生成 setter 方法。如果为 false，实体类属性就可以改变。默认为 false。
- rootClass: 设置所有实体类的基类。如果设置，则需要使用类的全限定名称。并且，如果 MBG 能够加载 rootClass，那么 MBG 不会覆盖和父类中完全匹配的属性。
- trimStrings：判断是否对数据库查询结果进行 trim 操作，默认值为 false。如果设置为 true 就会生成如下代码。

```
public void setUserName(String userName) {
    this.userName = userName == null ? null : userName.trim();
}
```

```
<javaModelGenerator targetPackage="test.model" targetProject="src\main\java">
    <property name="trimStrings" value="true"/>
</javaModelGenerator>
```

sqlMapGenerator 标签

该标签用于配置 SQL 映射文件的属性，如果 targetRuntime 设置为 MyBatis3，则只有当 javaClientGenerator 配置需要 XML 时，该标签才必须配置一个。如果没有配置 javaClientGenerator，则使用以下规则。

- 如果指定了一个 sqlMapGenerator，那么 MBG 将只生成 XML 的 SQL 映射文件和实体类。
- 如果没有指定 sqlMapGenerator，那么 MBG 将只生成实体类。

该标签有两个必要属性:

- targetPackage：生成 SQL 映射文件（XML 文件）存放的包名。一般就是放在该包下，实际还会受到其他配置的影响。
- targetProject：指定目标项目路径，可以使用相对路径或绝对路径。

该标签还有一个可选的 property 子标签属性 enableSubPackages，如果为 true，MBG 会根据 catalog 和 schema 来生成子包。如果为 false 就会直接用 targetPackage 属性，默认为 false。

```
<sqlMapGenerator targetPackage="test.xml" targetProject="src\main\resources"/>
```

javaClientGenerator 标签

该标签用于配置 Java 客户端生成器（Mapper 接口）的属性，如果不配置该标签，就不会生成 Mapper 接口。该标签有以下 3 个必选属性。

- type: 用于选择客户端代码（Mapper 接口）生成器，用户可以自定义实现，需要继承 AbstractJavaClientGenerator 类，必须有一个默认空的构造方法。该属性提供了以下预设的代码生成器，由于 context 的 targetRuntime 的不同，需要指定的属性值也有多不同。
  - XMLMAPPER: 所有的方法都在 XML 中，接口调用依赖 XML 文件。推荐使用，将接口和 XML 完全分离，容易维护，接口中不出现 SQL 语句，只在 XML 中配置 SQL，修改 SQL 时不需要重新编译。
  - ANNOTATEDMAPPER：基于注解的 Mapper 接口，不会有对应的 XML 映射文件。
  - MIXEDMAPPER(MyBatis3)：XML 和注解的混合形式。
- targetPackage：生成 Mapper 接口存放的包名。一般就是放在该包下，实际还会受到其他配置的影响。
- targetProject：指定目标项目路径，可以使用相对路径或绝对路径。

```
<javaClientGenerator type="XMLMAPPER" targetPackage="test.dao" targetProject="src\main\java"/>
```

5.2.2 表标签

table 是最重要的一个标签，该标签用于配置需要通过内省数据库的表，只有在 table 中配置过的表，才能经过上述其他配置生成最终的代码，该标签至少要配置一个，可以配置多个。

table 标签有一个必选属性 tableName，该属性指定要生成的表名，可以使用 SQL 通配符匹配多个表。table 标签包含了多个可选属性:

- schema: 数据库的 schema，可以使用 SQL 通配符匹配。如果设置了该值，生成 SQL 的表名会变成如 schema.tableName 的形式。
- catalog: 数据库的 catalog, 如果设置了该值，生成 SQL 的表名会变成 catalog.tableName 的形式。.....

同时也包含了 <property> 在内的多个子标签。感兴趣自己查吧，太多了不想列。

6 MyBatis 高级查询

6.1 高级结果映射

6.1.1 一对一映射

一对一映射因为不需要考虑是否存在重复数据，因此使用起来很简单，而且可以直接使用 MyBatis 的自动映射。

使用自动映射处理一对一关系 使用自动映射就是通过别名让 MyBatis 自动将值匹配到对应的字段上，简单的别名映射如 user_name 对应 userName。

除此之外 MyBatis 还支持复杂的属性映射，可以多层嵌套，例如将 role.role_name 映射到 role.roleName 上。MyBatis 会先查找 role 属性，如果存在 role 属性就创建 role 对象，然后在 role 对象中继续查找 roleName，将 role_name 的值绑定到 role 对象的 roleName 属性上。

在 SysUser 中添加属性:

```
public class SysUser {
    private SysRole role;
    public SysRole getRole() {
        return role;
    }
    public void setRole(SysRole role) {
        this.role = role;
    }
    // 其他代码
}
```

```
<select id="selectUserAndRoleById" resultType="learn.mybatis.simple.model.SysUser">
    SELECT u.id,
           u.user_name,
           u.user_password,
           u.user_email,
           u.user_info,
           u.head_img,
           u.create_time,
           r.id "role.id",
           r.role_name "role.roleName",
           r.enabled "role.enabled",
           r.create_by "role.createBy",
           r.create_time "role.createBy"
    FROM sys_user AS u,
         sys_role AS r,
         sys_user_role AS ur
            INNER JOIN ur on u.id = ur.user_id
            INNER JOIN r on ur.role_id = r.id
    WHERE u.id = #{id}
</select>
```

注意上述方法中 sys_role 查询列的别名都是 “role.” 前缀，通过这种方式将 role 的属性都映射到了 SysUser 的 role 属性上。通过 SQL 日志可以看到已经查询出的一条数据，MyBatis 将这条数据映射到了两个类中，像这种通过一次查询将结果映射到不同对象的方式，称之为关联的嵌套结果映射。

使用 resultMap 配置一对一映射 resultMap 方式，除了写起来繁，没其他缺点:

```
<resultMap id="userRoleMap" type="learn.mybatis.simple.model.SysUser">
    <id property="id" column="id"/>
    <result property="userName" column="user_name"/>
    <result property="userPassword" column="user_password"/>
    ......
    <result property="role.id" column="role_id">
    <result property="role.roleName" column="role_name">
    <result property="role.enabled" column="enabled">
    ......
</resultMap>
```

通过生成器可以生成一些基础代码后进行扩充可以方便一点。

```
<resultMap id="userRoleMap" type="learn.mybatis.simple.model.SysUser" extends="userMap">
    <result property="role.id" column="role_id"/>
    <result property="role.roleName" column="role_roleName"/>
    <result property="role.enabled" column="role_enabled"/>
    <result property="role.createBy" column="role_createBy"/>
    <result property="role.createTime" column="role_create_ime" jdbcType="TIMESTAMP"/>
</resultMap>
```

使用 resultMap 的 association 标签配置一对一映射 在 resultMap 中，association 标签用于和一个复杂的类型进行关联，即用于一对一的关联配置。

在上面配置的基础上，再做修改，改成 association 标签的配置方式，代码如下:

```
<resultMap id="userRoleMap" type="learn.mybatis.simple.model.SysUser" extends="userMap">
    <association property="role" columnPrefix="role_"
        javaType="learn.mybatis.simple.model.SysRole">
        <result property="id" column="id"/>
        <result property="roleName" column="name"/>
        <result property="enabled" column="enabled"/>
        <result property="createBy" column="createBy"/>
        <result property="createTime" column="create_img" jdbcType="TIMESTAMP"/>
    </association>
</resultMap>
```

<association> 标签包含如下属性:

- property: 对应实体类你中的属性名，必填。
- javaType: 属性对应 java 类型。
- resultMap: 使用现有的 resultMap。

- columnPrefix: 配置前缀，下面就不用写了。

association 标签的嵌套查询 除了前面 3 种通过复杂的 SQL 查询获取结果，还可以利用简单的 SQL 通过多次查询转换为我们需要的结果，这种方式与根据业务逻辑手动执行多次 SQL 的方式相像，最后会将结果组合成一个对象。

association 标签的嵌套查询常用的属性如下:

- select: 另一个映射查询的 id，MyBatis 会额外执行这个查询获取嵌套对象的结果。
- column: 列名，将主查询中列的结果作为嵌套查询的参数，配置方式如: column = prop = col1m prop2 = col2 prop1 和 prop2 将作为嵌套查询的参数。
- fetchType: 数据加载方式，可选值为 lazy 和 eager，分别为延迟加载和积极加载，这个配置会覆盖全局的 lazyLoadingEnabled 配置。

使用嵌套查询的方式配置一个和前面功能一样的方法，首先在 UserMapper.xml 中创建如下的 resultMap。

```
<resultMap id="userRoleMapSelect" type="learn.mybatis.simple.model.SysUser" extends="userMap">
    <association property="role" column="{id = role_id}"
        select="learn.mybatis.simple.mapper.RoleMapper.selectRoleById"/>
</resultMap>
```

然后创建对应的 select 方法:

```
<select id="selectUserAndRoleByIdSelect" resultMap="userRoleMapSelect">
    SELECT u.id,
           u.user_name,
           u.user_password,
           u.user_email,
           u.user_info,
           u.head_img,
           u.create_time,
           ur.role.id
    FROM sys_user AS u,
         sys_user_role AS ur
            INNER JOIN ur on u.id = ur.user_id
    WHERE u.id = #{id}
</select>
```

6.1.2 一对多映射

collection 集合的嵌套结果映射 集合的嵌套结果映射就是指通过一次 SQL 查询将所有的结果查询出来，然后通过配置的结果映射，将数据映射到不同的对象中去。在一对多的关系中，主表的一条数据会对应关联表中的多条数据，因此一般查询时会查询出多个结果，按照一对多的数据结构存储数据的时候，最终的结果数会小于等于查询的总记录数。

在 SysUser 中修改属性:

```
public class SysUser {
```

```
    private List<SysRole> roleList;
    public List<SysRole> getRoleList() {
        return roleList;
    }

    public void setRoleList(List<SysRole> roleList) {
        this.roleList = roleList;
    }
    // 其他成员
}
```

association 和 collection 的属性以及属性的作用完全相同，不再解释。如果我们已经配置过 userMap 和 roleMap, 那么我们可以直接这样写:

```
<resultMap id="userRoleListMap" type="learn.mybatis.simple.model.SysUser">
    <collection property="roleList" columnPrefix="role_"
        resultMap="learn.mybatis.simple.mapper.RoleMapper.roleMapper"/>
</resultMap>
```

对应的映射代码如下:

```
<select id="selectAllUserAndRoles" resultMap="userRoleListMap">
    SELECT u.id,
           u.user_name,
           u.user_password,
           u.user_email,
           u.user_info,
           u.head_img,
           u.create_time,
           r.id         role_id,
           r.role_name  role_role_name,
           r.enabled    role_enabled,
           r.create_by  role_create_by,
           r.create_time role_create_time
    FROM sys_user AS u,
         sys_user_role AS ur,
         sys_role AS r
            INNER JOIN ur ON u.id = ur.user_id
            INNER JOIN r ON ur.role_id = r.id
</select>
```

下面说明 mybatis 如何处理一对多的配置:

MyBatis 在处理结果的时候，会判断结果是否相同，如果是相同的结果，则只会保留第一个结果，所以这个问题的关键点就是 MyBatis 如何判断结果是否相同。MyBatis 判断结果是否相同时，最简单的情况就是在映射配置中至少有一个 id 标签，在 userMap 中配置如下。

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

我们对 id（构造方法中为 idArg）的理解一般是，它配置的字段为表的主键（联合主键时可以配置多个 id 标签），因为 MyBatis 的 resultMap 只用于配置结果如何映射，并不知道这个

表具体如何。id 的唯一作用就是在嵌套的映射配置时判断数据是否相同，当配置 id 标签时，MyBatis 只需要逐条比较所有数据中 id 标签配置的字段值是否相同即可。在配置嵌套结果查询时，配置 id 标签可以提高处理效率。

这样一来，上面的查询就不难理解了。因为前两条数据的 userMap 部分的 id 相同，所以它们属于同一个用户，因此这条数据会合并到同一个用户中。

很可能会出现一种没有配置 id 的情况。没有配置 id 时，MyBatis 就会把 resultMap 中配置的所有字段进行比较，如果所有字段的值都相同就合并，只要有一个字段值不同，就不合并。

在嵌套结果配置 id 属性时，如果查询语句中没有查询 id 属性配置的列，就会导致 id 对应的值为 null。这种情况下，所有值的 id 都相同，因此会使嵌套的集合中只有一条数据。所以在配置 id 列时，查询语句中必须包含该列。

MyBatis 会对嵌套查询的每一级对象都进行属性比较。MyBatis 会首先比较顶层的对象，如果 SysUser 部分相同，就继续比较 SysRole 部分，如果 SysRole 不同，就会增加一个 SysRole，两个 SysRole 相同就保留前一个。假设 SysRole 还有下一级，仍然按照该规则去比较。

6.1.3 鉴别器映射

有时一个单独的数据库查询会返回很多不同数据类型（希望有些关联）的结果集。discriminator 鉴别器标签就是用来处理这种情况的。鉴别器非常容易理解，因为它很像 Java 语言中的 switch 语句。

discriminator 标签常用的两个属性如下。

- column: 用于设置要进行鉴别比较值的列。
- javaType: 用于指定列的类型，保证使用相同的 Java 类型来比较。

discriminator 标签可以有 1 个或多个 case 标签，case 标签包含以下三个属性:

- value: 该值为 discriminator 指定 column 用来匹配的值。
- resultMap, resultType。

```
<resultMap id = "..." type = "...">
    <discriminator column="enabled" javaType="int">
        <case value="1" resultMap="..."/>
        <case value="0" resultMap="..."/>
    </discriminator>
</resultMap>
```

6.2 存储过程

6.2.1 调用存储过程

添加存储过程与对应的映射文件如下:

```
# 根据用户 id 查询用户其他信息
DROP PROCEDURE IF EXISTS `select_user_by_id`;
DELIMITER ;;
CREATE PROCEDURE `select_user_by_id`(
    IN userId BIGINT,
    OUT userName VARCHAR(50),
    OUT userPassword VARCHAR(50),
    OUT userEmail VARCHAR(50),
    OUT userInfo TEXT,
    OUT headImg BLOB,
    OUT createTime DATETIME)
BEGIN
    SELECT user_name, user_password, user_email, user_info, head_img, create_time
    INTO userName, userPassword, userEmail, userInfo, headImg, createTime
    FROM sys_user
    WHERE id = userId;
END ;;
DELIMITER ;
```

```
<select id="selectUserById" statementType="CALLABLE" useCache="false">
    {call select_user_by_id(
        #{id, mode = IN},
        #{userName, mode = OUT, jdbcType = VARCHAR},
        #{userPassword, mode = OUT, jdbcType = VARCHAR},
        #{userEmail, mode = OUT, jdbcType = VARCHAR},
        #{userInfo, mode = OUT, jdbcType = VARCHAR},
        #{headImg, mode = OUT, jdbcType = BLOB, javaType = _byte[]},
        #{createTime, mode = OUT, jdbcType = TIMESTAMP}
    )}
</select>
```

在调用存储过程的方法中，需要把 statementType 设置为 CALLABLE，在使用 select 标签调用存储过程时，由于存储过程方式不支持 MyBatis 的二级缓存，因此为了避免缓存配置出错，直接将 select 标签的 useCache 属性设置为 false。

OUT 模式的参数必须指定 jdbcType。这是因为在 IN 模式下，MyBatis 提供了默认的 jdbcType，在 OUT 模式下没有提供。

headImg 还特别设置了 javaType。在 MyBatis 映射的 Java 类中，不推荐使用基本类型，数据库 BLOB 类型对应的 Java 类型通常都是写成 byte[] 字节数组的形式的，因为 byte[] 数组不存在默认值的问题，所以不影响一般的使用。但是在不指定 javaType 的情况下，MyBatis 默认使用 Byte 类型。由于 byte 是基本类型，所以设置 javaType 时要使用带下画线的方式，在这里就是 _byte[]。_byte 对应的是基本类型，byte 对应的是 Byte 类型。

添加接口及对应的测试如下:

```
void selectUserById(SysUser user);
```

```
@Test
public void testSelectUserById() {
    try(SqlSession sqlSession = this.getSqlSession()) {
```

```
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        SysUser user = new SysUser();
        user.setId(1L);
        userMapper.selectUserById(user);
        Assert.assertNotNull(user.getUserName());
    }
}
```

运行测试，所有的 OUT 属性就被加载到对应的对象中了。

6.2.2 多参数的存储过程

添加存储过程与对应的映射文件如下:

```
# 简单的根据用户名和分页参数进行查询，返回总数和分页数据
DROP PROCEDURE IF EXISTS `select_user_page`;
DELIMITER ;;
CREATE PROCEDURE `select_user_page`(
    IN userName VARCHAR(50),
    IN _offset BIGINT,
    IN _limit BIGINT,
    OUT total BIGINT)
BEGIN
    SELECT count(*) INTO total FROM sys_user WHERE user_name LIKE CONCAT('%', userName, '%');
    SELECT * FROM sys_user WHERE user_name LIKE CONCAT('%', userName, '%') LIMIT
        _offset,_limit;
END ;;
DELIMITER ;
```

```
<select id="selectUserPage" statementType="CALLABLE" useCache="false" resultMap="userMap">
    {call select_user_page(
        #{userName, mode = IN},
        #{offset, mode = IN},
        #{limit, mode = IN},
        #{total, mode = OUT, jdbcType = BIGINT}
    )}
</select>
```

这个映射字段对应的接口需要这样写:

```
List<SysUser> selectUserPage(Map<String, Object> params);
```

由于需要多个入参和一个出参，而入参中除了 userName 属性在 SysUser 中，其他 3 个参数都和 SysUser 无关，因此为了使用 SysUser 而增加 3 个属性也是可以的。这里演示使用 Map 容器作为参数类型。

```
@Test
public void testSelectUserPage() {
    try(SqlSession sqlSession = this.getSqlSession()) {
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        Map<String, Object> params = new HashMap<String, Object>();
```

```
        params.put("userName","ad");
        params.put("offset", 0);
        params.put("limit", 10);
        List<SysUser> userList = userMapper.selectUserPage(params);
        Long total = (Long)params.get("total");
        System.out.println(total);
        for(SysUser user:userList){
            System.out.println(user.getUserName());
        }
    }
}
```

6.3 使用枚举或其他对象

6.3.1 使用 MyBatis 的枚举类型处理器

首先创建枚举类型:

```
public enum Enabled {
    disable, enable;
}
```

添加到对应类中:

```
private Enabled enabled;
public Enabled getEnabled() {
    return enabled;
}

public void setEnabled(Enabled enabled) {
    this.enabled = enabled;
}
```

在数据库中不存在一个和 Enabled 枚举对应的数据库类型，因此在和数据库交互的时候，不能直接使用枚举类型，在查询数据时，需要将数据库 int 类型的值转换为 Java 中的枚举值。在保存、更新数据或者作为查询条件时，需要将枚举值转换为数据库中的 int 类型。

MyBatis 在处理 Java 类型和数据库类型时，使用 TypeHandler（类型处理器）对这两者进行转换。Mybatis 为 Java 和数据库 JDBC 中的基本类型和常用的类型提供了 TypeHandler 接口的实现。MyBatis 在启动时会加载所有的 JDBC 对应的类型处理器，在处理枚举类型时默认使用 org.apache.ibatis.type.EnumTypeHandler 处理器，这个处理器会将枚举类型转换为字符串类型的字面值并使用，对于 Enabled 而言便是 " disabled " 和 " enabled " 字符串。

除了这个枚举类型处理器，MyBatis 还提供了另一个 EnumOrdinalTypeHandler 处理器，这个处理器使用枚举的索引进行处理，可以解决此处遇到的问题。想要使用这个处理器，需要在 mybatis-config.xml 中添加如下配置。

```
<typeHandlers>
```

```
    <typeHandler javaType = "learn.mybatis.simple.type.Enabled" handler =
        "org.apache.ibatis.type.EnumOrdinalTypeHandler"/>
</typeHandlers>
```

6.3.2 自定义枚举类型处理器

如果值既不是枚举的字面值，也不是枚举的索引值，这种情况下就需要自己来实现类型处理器。

修改枚举类型如下:

```
public enum Enabled {
    enabled(1), disable(0);

    private final int value;

    Enabled(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

现在 Enabled 中的值和顺序无关，针对该类，添加新的枚举类型处理器。

```
public class EnabledTypeHandler implements TypeHandler<Enabled> {
    private final Map<Integer, Enabled> enabledMap = new HashMap<Integer, Enabled>();

    public EnabledTypeHandler() {
        for(Enabled enabled: Enabled.values()) {
            enabledMap.put(enabled.getValue(), enabled);
        }
    }

    @Override
    public void setParameter(PreparedStatement ps, int i, Enabled parameter, JdbcType
        jdbcType) throws SQLException {
        ps.setInt(i, parameter.getValue());
    }

    @Override
    public Enabled getResult(ResultSet rs, String columnName) throws SQLException {
        return enabledMap.get(rs.getInt(columnName));
    }

    @Override
    public Enabled getResult(ResultSet rs, int columnIndex) throws SQLException {
        return enabledMap.get(rs.getInt(columnIndex));
    }
```

```
24
25      @Override
26      public Enabled getResult(CallableStatement cs, int columnIndex) throws SQLException {
27          return enabledMap.get(cs.getInt(columnIndex));
28      }
29  }
```

EnabledTypeHandler 实现了 TypeHandler 接口，并且针对 4 个接口方法对 Enabled 类型进行了转换。在 TypeHandler 接口实现类中，除了默认无参的构造方法，还有一个隐含的带有一个 Class 参数的构造方法。

```
1  public EnabledTypeHandler(Class<?> type) {
2      this();
3  }
```

当针对特定的接口处理类型时，使用这个构造方法可以写出通用的类型处理器，就像 MyBatis 提供的两个枚举类型处理器一样。有了自己的类型处理器后，还需要在 mybatis-config.xml 中进行如下配置。

```
1  <typeHandlers>
2      <typeHandler javaType = "learn.mybatis.simple.type.Enabled" handler =
           "learn.mybatis.simple.type.EnabledTypeHandler"/>
3  </typeHandlers>
```

7 MyBatis 缓存

使用缓存可以使应用更快地获取数据，避免频繁的数据库交互，尤其是在查询越多、缓存命中率越高的情况下，使用缓存的作用就越明显。

一般提到 MyBatis 缓存的时候，都是指二级缓存。一级缓存（也叫本地缓存）默认会启用，并且不能控制，因此很少会提到。

7.1 一级缓存

如果我们执行下面这段代码:

```
// 准备代码
SysUser user1 = userMapper.selectById(1L);
SysUser user2 = userMapper.selectById(1L);
```

在第一次执行 selectById 方法获取 SysUser 数据时，真正执行了数据库查询，得到了 user1 的结果。第二次执行获取 user2 的时候，如果查日志可以看到，并没有执行 sql 语句，只有一次查询，也就是说第二次查询并没有执行数据库操作。

MyBatis 的一级缓存存在于 SqlSession 的生命周期中，在同一个 SqlSession 中查询时，MyBatis 会把执行的方法和参数通过算法生成缓存的键值，将键值和查询结果存入一个 Map 对象中。如果同一个 SqlSession 中执行的方法和参数完全一致，那么通过算法会生成相同的键值，当 Map 缓存对象中已经存在该键值时，则会返回缓存中的对象。

缓存中的对象和我们得到的结果是同一个对象，反复使用相同参数执行同一个方法时，总是返回同一个对象。如果不想让 selectById 方法使用一级缓存，可以对该方法做如下修改。

```
<select id="selectById" flushCache="true" resultMap="userMap">
    SELECT * FROM sys_user WHERE id = #{id}
</select>
```

修改在原来方法的基础上增加了 flushCache= " true " ，这个属性配置为 true 后，会在查询数据前清空当前的一级缓存。

任何的 INSERT、UPDATE、DELETE 操作都会清空一级缓存。

7.2 二级缓存

MyBatis 的二级缓存非常强大，它不同于一级缓存只存在于 SqlSession 的生命周期中，而是可以理解为存在于 SqlSessionFactory 的生命周期中。虽然目前还没接触过同时存在多个 SqlSessionFactory 的情况，但可以知道，当存在多个 SqlSessionFactory 时，它们的缓存都是绑定在各自对象上的，缓存数据在一般情况下是不相通的。只有在使用如 Redis 这样的缓存数据库时，才可以共享缓存。

7.2.1 配置二级缓存

在 MyBatis 的全局配置 settings 中有一个参数 cacheEnabled，这个参数是二级缓存的全局开关，默认值是 true，初始状态为启用状态。

MyBatis 的二级缓存是和命名空间绑定的，即二级缓存需要配置在 Mapper.xml 映射文件中，或者配置在 Mapper.java 接口中。在映射文件中，命名空间就是 XML 根节点 mapper 的 namespace 属性。在 Mapper 接口中，命名空间就是接口的全限定名称。

映射文件中配置二级缓存 在保证二级缓存的全局配置开启的情况下，开启二级缓存只需要在映射文件中添加 <cache/> 元素即可。

```
<mapper namespace = "xxx">
    <cache/>
    ......
</mapper>
```

默认的二级缓存有如下效果:

- 映射语句文件中的所有 SELECT 语句将被缓存。
- 映射语句文中的所有 INSERT，UPDATE，DELETE 语句会刷新缓存。
- 缓存会使用 LRU(最近最少使用) 算法进行 GC。
- 根据时间表（如 no Flush Interval，没有刷新间隔），缓存不会以任何时间顺序来刷新。
- 缓存会存储集合或对象（无论查询方法返回什么类型的值）的 1024 个引用。
- 缓存会被视为 read/write（可读/可写）的，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。

这些属性可以通过缓存元素的属性来修改:

```
<cache eviction="FIFO" flushInterval="60000" size="512" readOnly="true"/>
```

接口中配置二级缓存 如果想对注解方法启用二级缓存，还需要在 Mapper 接口中进行配置，如果 Mapper 接口也存在对应的 XML 映射文件，两者同时开启缓存时，还需要特殊配置。

```
@CacheNamespace(eviction=FifoCache.class, flushInterval=60000, size=512, readOnly=true)
public interface RoleMapper{
    // 接口方法
}
```

当同时使用注解方式和 XML 映射文件时，如果同时配置了上述的二级缓存，就会抛出异常。

这是因为 Mapper 接口和对应的 XML 文件是相同的命名空间，想使用二级缓存，两者必须同时配置（如果接口不存在使用注解方式的方法，可以只在 XML 中配置），因此按照上面的方式进行配置就会出错，这个时候应该使用参照缓存。

```
@CacheNamespaceRed(RoleMapper.class)
```

```java
public interface RoleMapper {
    // 接口方法
}
```

Mapper 接口可以通过注解引用 XML 映射文件或者其他接口的缓存，在 XML 中也可以配置参照缓存，如可以在 RoleMapper.xml 中进行如下修改。

```xml
<cache-ref namespace="learn.mybatis.simple.mapper.RoleMapper"/>
```

这样配置后，XML 就会引用 Mapper 接口中配置的二级缓存，同样可以避免同时配置二级缓存导致的冲突。

MyBatis 中很少会同时使用 Mapper 接口注解方式和 XML 映射文件，所以参照缓存并不是为了解决这个问题而设计的。参照缓存除了能够通过引用其他缓存减少配置外，主要的作用是解决脏读（后面章节详细介绍）。

7.2.2 使用二级缓存

需要注意的是，由于配置的是可读写的缓存，而 MyBatis 使用 SerializedCache 序列化缓存来实现可读写缓存类，并通过序列化和反序列化来保证通过缓存获取数据时，得到的是一个新的实例。因此，如果配置为只读缓存，MyBatis 就会使用 Map 来存储缓存值，这种情况下，从缓存中获取的对象就是同一个实例。

因为使用可读写缓存，可以使用 SerializedCache 序列化缓存。这个缓存类要求所有被序列化的对象必须实现 Serializable 标记接口。

当调用 close 方法关闭 SqlSession 时，SqlSession 才会保存查询数据到二级缓存中。在这之后二级缓存才有了缓存数据。

在实际使用过程中，如果我们输出日志，会出现 Cache Hit Ratio 字段，代表二级缓存命中率。

MyBatis 默认提供的缓存实现是基于 Map 实现的内存缓存，已经可以满足基本的应用。但是当需要缓存大量的数据时，不能仅仅通过提高内存来使用 MyBatis 的二级缓存，还可以选择一些类似 EhCache 的缓存框架或 Redis 缓存数据库等工具来保存 MyBatis 的二级缓存数据[1]。

二级缓存虽然好处很多，但并不是什么时候都可以使用。在以下场景中，推荐使用二级缓存。

- 以查询为主的应用中，只有尽可能少的增、删、改操作。
- 绝大多数以单表操作存在时，由于很少存在互相关联的情况，因此不会出现脏数据。
- 可以按业务划分对表进行分组时，如关联的表比较少，可以通过参照缓存进行配置。

除了推荐使用的情况，如果脏读对系统没有影响，也可以考虑使用。在无法保证数据不出现脏读的情况下，建议在业务层使用可控制的缓存代替二级缓存。

[1]由于我不会，所以省略原文这部分内容。

7.3 脏数据的产生和避免

二级缓存虽然能提高应用效率，减轻数据库服务器的压力，但是如果使用不当，很容易产生脏数据。这些脏数据会在不知不觉中影响业务逻辑，影响应用的实效。

MyBatis 的二级缓存是和命名空间绑定的，所以通常情况下每一个 Mapper 映射文件都拥有自己的二级缓存，不同 Mapper 的二级缓存互不影响。在常见的数据库操作中，多表联合查询非常常见，由于关系型数据库的设计，使得很多时候需要关联多个表才能获得想要的数据。在关联多表查询时肯定会将该查询放到某个命名空间下的映射文件中，这样一个多表的查询就会缓存在该命名空间的二级缓存中。涉及这些表的增、删、改操作通常不在一个映射文件中，它们的命名空间不同，因此当有数据变化时，多表查询的缓存未必会被清空，这种情况下就会产生脏数据。

假设有这样一种多表查询情形: 有三个不同的 sqlSession 对象，第一个 SqlSession 中获取了用户和关联的角色信息，第二个 SqlSession 中查询角色并修改了角色的信息，第三个 SqlSession 中查询用户和关联的角色信息。

由于三个 sqlSession 对象不同，第二个 sqlSession 查询并修改数据后，第三个 namespace 对应的缓存数据并没有被更新，这时从缓存中直接取出数据，就出现了脏数据，因为角色名称已经修改，但是这里读取到的角色名称仍然是修改前的名字，因此出现了脏读。

该如何避免脏数据的出现呢？这时就需要用到参照缓存了。当某几个表可以作为一个业务整体时，通常是让几个会关联的 ER 表同时使用同一个二级缓存，这样就能解决脏数据问题。