第 3-3 课：如何优雅地使用 MyBatis 注解版

自从 Java 1.5 开始引入了注解，注解便被广泛地应用在了各种开源软件中，使用注解大大地降低了系统中的配置项，让编程变得更为优雅。MyBatis 也顺应潮流基于注解推出了 MyBatis 的注解版本，避免开发过程中频繁切换到 XML 或者 Java 代码中，从而让开发者使用 MyBatis 会有统一的开发体验。

因为最初设计时，MyBatis 是一个 XML 驱动的框架，配置信息是基于 XML 的，而且映射语句也是定义在 XML 中的，而到了 MyBatis 3，就有新选择了。MyBatis 3 构建在全面且强大的基于 Java 语言的配置 API 之上，这个配置 API 是基于 XML 的 MyBatis 配置的基础，也是新的基于注解配置的基础。注解提供了一种简单的方式来实现简单映射语句，而不会引入大量的开销。

注解版

注解版的使用方式和 XML 版本相同，只有在构建 SQL 方面有所区别，所以本课重点介绍两者之间的差异部分。

相关配置

注解版在 application.properties 只需要指明实体类的包路径即可，其他保持不变：

```
mybatis.type-aliases-package=com.neo.model

spring.datasource.url=jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnico
de=true&characterEncoding=utf-8&useSSL=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

传参方式

先来介绍一下使用注解版的 MyBatis 如何将参数传递到 SQL 中。

直接使用

```
@Delete("DELETE FROM users WHERE id =#{id}")
void delete(Long id);
```

在 SQL 中使用 #{id} 来接收同名参数。

使用 @Param

如果你的映射方法的形参有多个，这个注解使用在映射方法的参数上就能为它们取自定义名字。若不给出自

定义名字，多参数则先以 "param" 作前缀，再加上它们的参数位置作为参数别名。例如，#{param1}、#{param2}，这个是默认值。如果注解是 @Param("person")，那么参数就会被命名为 #{person}。

```
@Select("SELECT * FROM users WHERE user_sex = #{user_sex}")
List<User> getListByUserSex(@Param("user_sex") String userSex);
```

使用 Map

需要传送多个参数时，可以考虑使用 Map：

```
@Select("SELECT * FROM users WHERE username=#{username} AND user_sex = #{user_sex}
")
List<User> getListByNameAndSex(Map<String, Object> map);
```

使用时将参数依次加入到 Map 中即可：

```
Map param=  new HashMap();
param.put("username","aa");
param.put("user_sex","MAN");
List<User> users = userMapper.getListByNameAndSex(param);
```

使用对象

最常用的使用方式是直接使用对象：

```
@Insert("INSERT INTO users(userName,passWord,user_sex) VALUES(#{userName}, #{passW
ord}, #{userSex})")
void insert(User user);
```

在执行时，系统会自动读取对象的属性并值赋值到同名的 #{xxx} 中。

注解介绍

注解版最大的特点是具体的 SQL 文件需要写在 Mapper 类中，取消了 Mapper 的 XML 配置。

上面介绍参数的时候，已经使用了 @Select、@Delete 等标签，这就是 MyBatis 提供的注解来取代其 XML 文件配置，下面我们一一介绍。

@Select 注解

@Select 主要在查询的时候使用，查询类的注解，所有的查询均使用这个，具体如下：

```
@Select("SELECT * FROM users WHERE user_sex = #{user_sex}")
List<User> getListByUserSex(@Param("user_sex") String userSex);
```

@Insert 注解

@Insert，插入数据库时使用，直接传入实体类会自动解析属性到对应的值，示例如下：

```
@Insert("INSERT INTO users(userName,passWord,user_sex) VALUES(#{userName}, #{passW
ord}, #{userSex})")
void insert(User user);
```

@Update 注解

@Update，所有的更新操作 SQL 都可以使用 @Update。

```
@Update("UPDATE users SET userName=#{userName},nick_name=#{nickName} WHERE id =#{i
d}")
void update(UserEntity user);
```

@Delete 注解

@Delete 处理数据删除。

```
@Delete("DELETE FROM users WHERE id =#{id}")
void delete(Long id);
```

以上就是项目中常用的增、删、改、查，但有时候我们有一些特殊的场景需要处理，比如查询的对象返回值属性名和字段名不一致，或者对象的属性中使用了枚举。我们期望查询的返回结果可以将此字段自动转化为对应的类型，MyBatis 提供了另外两个注解来支持：@Results 和 @Result。

@Results 和 @Result 注解

这两个注解配合来使用，主要作用是将数据库中查询到的数值转化为具体的字段，修饰返回的结果集，关联实体类属性和数据库字段一一对应，如果实体类属性和数据库属性名保持一致，就不需要这个属性来修饰。示例如下：

```
@Select("SELECT * FROM users")
@Results({
    @Result(property = "userSex",  column = "user_sex", javaType = UserSexEnum.cla
ss),
    @Result(property = "nickName", column = "nick_name")
})
List<UserEntity> getAll();
```

为了更接近实际项目，特地将 user_sex、nick_name 两个属性加了下划线和实体类属性名不一致，另外 user_sex 使用了枚举，使用 @Results 和 @Result 即可解决这样的问题。

了解更多注解使用请点击这里查看。

注意，使用 # 符号和 `$` 符号的不同：

```
// This example creates a prepared statement, something like select * from teacher
 where name = ?;
@Select("Select * from teacher where name = #{name}")
Teacher selectTeachForGivenName(@Param("name") String name);

// This example creates n inlined statement, something like select * from teacher
where name = 'someName';
@Select("Select * from teacher where name = '${name}'")
Teacher selectTeachForGivenName(@Param("name") String name);
```

同上，上面两个例子可以发现，使用 # 会对 SQL 进行预处理，使用 `$` 时拼接 SQL，建议使用 #，使用 $ 有 SQL 注入的可能性。

动态 SQL

MyBatis 最大的特点是可以灵活的支持动态 SQL，在注解版中提供了两种方式来支持，第一种是使用注解来实现，另一种是提供 SQL 类来支持。

使用注解来实现

用 script 标签包围，然后像 XML 语法一样书写：

```
@Update("<script>
  update users
    <set>
      <if test="userName != null">userName=#{userName},</if>
      <if test="nickName != null">nick_name=#{nickName},</if>
    </set>
  where id=#{id}
</script>")
void update(User user);
```

这种方式就是注解 + XML 的混合使用方式，既有 XML 灵活又有注解的方便，但也有一个缺点需要在 Java 代码中拼接 XML 语法很不方便，因此 MyBatis 又提供了一种更优雅的使用方式来支持动态构建 SQL。

使用 SQL 构建类来支持

以分页为例进行演示，首先定义一个 UserSql 类，提供方法拼接需要执行的 SQL：

```
public class UserSql {
    public String getList(UserParam userParam) {
        StringBuffer sql = new StringBuffer("select id, userName, passWord, user_s
ex as userSex, nick_name as nickName");
        sql.append(" from users where 1=1 ");
        if (userParam != null) {
            if (StringUtils.isNotBlank(userParam.getUserName())) {
                sql.append(" and userName = #{userName}");
            }
            if (StringUtils.isNotBlank(userParam.getUserSex())) {
                sql.append(" and user_sex = #{userSex}");
            }
        }
        sql.append(" order by id desc");
        sql.append(" limit " + userParam.getBeginLine() + "," + userParam.getPageS
ize());
        log.info("getList sql is :" +sql.toString());
        return sql.toString();
    }
}
```

可以看出 UserSql 中有一个方法 getList，使用 StringBuffer 对 SQL 进行拼接，通过 if 判断来动态构建 SQL，最后方法返回需要执行的 SQL 语句。

接下来只需要在 Mapper 中引入这个类和方法即可。

```
@SelectProvider(type = UserSql.class, method = "getList")
List<UserEntity> getList(UserParam userParam);
```

- type：动态生成 SQL 的类
- method：类中具体的方法名

相对于 @SelectProvider 提供查询 SQL 方法导入，还有 @InsertProvider、@UpdateProvider、@DeleteProvider 提供给插入、更新、删除的时候使用。

结构化 SQL

可能你会觉得这样拼接 SQL 很麻烦，SQL 语句太复杂也比较乱，别着急！MyBatis 给我们提供了一种升级的方案：结构化 SQL。

示例如下：

```
public String getCount(UserParam userParam) {
   String sql= new SQL(){{
        SELECT("count(1)");
        FROM("users");
        if (StringUtils.isNotBlank(userParam.getUserName())) {
            WHERE("userName = #{userName}");
        }
        if (StringUtils.isNotBlank(userParam.getUserSex())) {
            WHERE("user_sex = #{userSex}");
        }
        //从这个 toString 可以看出，其内部使用高效的 StringBuilder 实现 SQL 拼接
    }}.toString();

    log.info("getCount sql is :" +sql);
    return sql;
}
```

- SELECT 表示要查询的字段，可以写多行，多行的 SELECT 会智能地进行合并而不会重复。
- FROM 和 WHERE 跟 SELECT 一样，可以写多个参数，也可以在多行重复使用，最终会智能合并而不会报错。这样语句适用于写很长的 SQL，且能够保证 SQL 结构清楚，便于维护、可读性高。

更多结构化的 SQL 语法请参考 SQL 语句构建器类。

> 具体使用和 XML 版本一致，直接注入到使用的类中即可。

多数据源使用

注解版的多数据源使用和 XML 版本的多数据源基本一致。

首先配置多数据源：

```
mybatis.type-aliases-package=com.neo.model

spring.datasource.test1.jdbc-url=jdbc:mysql://localhost:3306/test1?serverTimezone=
UTC&useUnicode=true&characterEncoding=utf-8&useSSL=true
spring.datasource.test1.username=root
spring.datasource.test1.password=root
spring.datasource.test1.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.test2.jdbc-url=jdbc:mysql://localhost:3306/test2?serverTimezone=
UTC&useUnicode=true&characterEncoding=utf-8&useSSL=true
spring.datasource.test2.username=root
spring.datasource.test2.password=root
spring.datasource.test2.driver-class-name=com.mysql.cj.jdbc.Driver
```

分别构建两个不同的数据源。

DataSource1 配置：

```
@Configuration
@MapperScan(basePackages = "com.neo.mapper.test1", sqlSessionTemplateRef  = "test1
SqlSessionTemplate")
public class DataSource1Config {

    @Bean(name = "test1DataSource")
    @ConfigurationProperties(prefix = "spring.datasource.test1")
    @Primary
    public DataSource testDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "test1SqlSessionFactory")
    @Primary
    public SqlSessionFactory testSqlSessionFactory(@Qualifier("test1DataSource") D
ataSource dataSource) throws Exception {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dataSource);
        return bean.getObject();
    }

    @Bean(name = "test1TransactionManager")
    @Primary
    public DataSourceTransactionManager testTransactionManager(@Qualifier("test1Da
taSource") DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean(name = "test1SqlSessionTemplate")
    @Primary
    public SqlSessionTemplate testSqlSessionTemplate(@Qualifier("test1SqlSessionFa
ctory") SqlSessionFactory sqlSessionFactory) throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

DataSource2 配置和 DataSource1 配置基本相同，只是去掉了 @Primary。

将以前的 Userapper 分别复制到 test1 和 test2 目录下，分别作为两个不同数据源的 Mapper 来使用。

测试

分别注入两个不同的 Mapper，想操作哪个数据源就使用哪个数据源的 Mapper 进行操作处理。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private User1Mapper user1Mapper;
    @Autowired
    private User2Mapper user2Mapper;

    @Test
    public void testInsert() throws Exception {
        user1Mapper.insert(new User("aa111", "a123456", UserSexEnum.MAN));
        user1Mapper.insert(new User("bb111", "b123456", UserSexEnum.WOMAN));
        user2Mapper.insert(new User("cc222", "b123456", UserSexEnum.MAN));
    }
}
```

执行测试用例完成后，检查 test1 库中的用户表有两条数据，test2 库中的用户表有 1 条数据证明测试成功。

如何选择

两种模式各有特点，注解版适合简单快速的模式，在微服务架构中，一般微服务都有自己对应的数据库，多表连接查询的需求会大大的降低，会越来越适合注解版。XML 模式比适合大型项目，可以灵活地动态生成 SQL，方便调整 SQL，也有痛痛快快、洋洋洒洒地写 SQL 的感觉。在具体开发过程中，根据公司业务和团队技术基础进行选型即可。

> 点击这里下载源码。