Mybatis 框架课程第二天

第1章 Mybatis 框架实现 CRUD 操作

1.1 回顾

1.1.1 自定义 Mybatis 框架的回顾

1.2 Mybatis 实现用户新增

1.2.1 在 UserMapper 类中添加新增方法

为了实现新增操作，我们可以在原有入门示例的 UserMapper.java 类中添加一个用于 saveUser()的方法用于用户新增操作。

UserMapper 类中新增 saveUser()方法，如下：

UserMapper.java

```
1  package com.itheima.mapper;
2
3  import java.util.List;
6
7  public interface UserMapper {
8
9      public List<User> findAll();
10     //新增用户
11     public int saveUser(User user);
12 }
```

1.2.2 在 UserMapper.xml 文件中加入新增配置

在 UserMapper.xml 文件中加入新增用户的配置，如下：

```
<!-- 新增 -->
<insert id="saveUser" parameterType="com.itheima.domain.User">
   insert into user(username,birthday,sex,address)
   values(#{username},#{birthday},#{sex},#{address});
</insert>
```

我们可以发现，这个 sql 语句中使用#{}字符，#{}代表占位符，我们可以理解是原来 jdbc 部分所学的?，它们都是代表占位符， 具体的值是由 User 类的 username 属性来决定的。

parameterType 属性：代表参数的类型，因为我们要传入的是一个类的对象，所以类型就写类的全名称。

注意：

这种方式要求<mapper namespace=”映射接口所在的包名”>,同时还要求<select>,<insert>,<delete>,<update>这些标签中的 id 属性一定与代理接口中的方法名相同。

1.2.3 添加测试类中的测试方法

UserTest.java

```
14 import com.itheima.mapper.UserMapper;
15
16 public class UserTest {
17     @Test
18     public void testSaveUser() throws Exception {
19         InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
20         SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
21
22         SqlSession session = factory.openSession();
23         UserMapper userMapper = session.getMapper(UserMapper.class);
24
25         User user = new User("老王","男",new Date(),"北京");
26         userMapper.saveUser(user);
27
28         session.close();
29         is.close();
30     }
```

测试结果如下：

```
Opening JDBC Connection
Created connection 280744458.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@10bbd20a]
==>  Preparing: insert into user(username,birthday,sex,address) values(?,?,?,?);
==> Parameters: 老王(String), 2018-02-27 14:53:54.228(Timestamp), 男(String), 北京(String)
<==    Updates: 1
```

打开 Mysql 数据库发现并没有添加任何记录，原因是什么？

这一点和 jdbc 是一样的，我们在实现增删改时一定要去控制事务的提交，那么在 mybatis 中如何控制事务提交呢？

可以使用:session.commit();来实现事务提交。加入事务提交后的代码如下：

UserTest.java

```
14 import com.itheima.mapper.UserMapper;
15
16 public class UserTest {
17     @Test
18     public void testSaveUser() throws Exception {
19         InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
20         SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
21
22         SqlSession session = factory.openSession();
23         UserMapper userMapper = session.getMapper(UserMapper.class);
24
25         User user = new User("老王","男",new Date(),"北京");
26         userMapper.saveUser(user);
27
28         session.commit();//提交事务
29         session.close();
30         is.close();
31     }
```

插入数据库表中的记录如下：

36	老王	2018-02-27 15:30:32	男	北京

1.2.4 问题扩展：新增用户 id 的返回值

新增用户后，同时还要返回当前新增用户的 id 值，因为 id 是由数据库的自动增长来实现的，所以就相当于我们要在新增后将自动增长 auto_increment 的值返回。

Mysql 自增主键的返回，配置如下：

```
<!-- 新增 -->
<insert id="saveUser" parameterType="com.itheima.domain.User">
    <!-- keyProperty代表要返回的值名称  order:取值为AFTER代表插入后的行为  resultType代表返回值的类型 -->
    <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">
        select last_insert_id();
    </selectKey>
    insert into user(username,birthday,sex,address)
    values(#{username},#{birthday},#{sex},#{address});

</insert>
```

修改测试类，用于接收 last_insert_id()的返回值。

```
@Test
public void testSaveUser() throws Exception {
    InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
    SqlSession session = factory.openSession();
    UserMapper userMapper = session.getMapper(UserMapper.class);

    User user = new User("老王","男",new Date(),"北京");
    int id = userMapper.saveUser(user);
    System.out.println(user.getId());
    session.commit();//提交事务
    session.close();
    is.close();
}
```

运行后的结果如下：

```
- Created connection 1754638213.
- Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6895a785]
- ==>  Preparing: insert into user(username,birthday,sex,address) values(?,?,?,?);
- ==> Parameters: 老王(String), 2018-02-27 17:48:05.517(Timestamp), 男(String), 北京(String)
- <==    Updates: 1
- ==>  Preparing: select last_insert_id();
- ==> Parameters:
- <==      Total: 1
user.id=1
```

1.3 Mybatis 实现用户修改

1.3.1 在 UserMapper 类中添加更新方法

为了实现更新操作，我们可以在原有入门示例的 UserMapper.java 类中添加一个用于 updateUser()

的方法用于用户更新操作。

UserMapper 类中新增 updateUser()方法，如下：

UserMapper.java

```
1  package com.itheima.mapper;
2
3  import java.util.List;
6
7  public interface UserMapper {
8      //更新
9      public void updateUser(User user);
10
11     //新增用户
12     public int saveUser(User user);
13
14     public List<User> findAll();
15
16 }
17
```

1.3.2 在 UerMapper.xml 文件中加入更新操作配置

在 UserMapper.xml 文件中加入更新用户的配置，如下：

```
<!-- 更新 -->
<update id="updateUser" parameterType="com.itheima.domain.User" >
   update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address} where id=#{id};
</update>
```

上面的更新语句中，同样使用#{}代表占位符，比如#{username}代表占位符，将来这个部分使用 User 对象中的 username 属性的值来作为 sql 语句中 username 字段的值。

1.3.3 加入更新的测试方法

为了更好的实现测试效果，我们加入更新的测试方法 testUpdateUser()方法，具体实现代码如下：

```
public class UserTest {
    @Test
    public void testUpdateUser() throws Exception {
        InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
        SqlSession session = factory.openSession();
        UserMapper userMapper = session.getMapper(UserMapper.class);

        User user = new User(44,"王老二","男",new Date(),"北京昌平");
        userMapper.updateUser(user);
        session.commit();//提交事务
        session.close();
        is.close();
    }
```

测试后，控制台输出结果如下：

```
Created connection 1754638213.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6895a785]
==>  Preparing: update user set username=?,birthday=?,sex=?,address=? where id=?;
==> Parameters: 王老二(String), 2018-03-01 11:19:13.427(Timestamp), 男(String), 北京昌平(String), 44(Integer)
<==    Updates: 1
Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6895a785]
```

可以看出成功实现了记录的更新。

1.4 Mybatis 实现用户删除

1.4.1 在 UserMapper 类中加入删除方法

在 UserMapper 类中加入删除 deleteUser()方法，用于实现用户的删除。

```
public interface UserMapper {
    //删除
    public void deleteUser(Integer id);

    //更新
    public void updateUser(User user);

    //新增用户
    public int saveUser(User user);

    public List<User> findAll();

}
```

1.4.2 在 UserMapper.xml 文件中加入删除操作

加入的删除的映射配置信息如下：

```
<!-- 删除 -->
<delete id="deleteUser" parameterType="int">
    delete from user where id=#{id}
</delete>
```

其中的#{id}是占位符，代表参数的值由方法的参数传入进来的。

注意：

1.此处的#{id}中的 id 其实只是一个形参，所以它的名称是自由定义的，比如定义成#{abc}也是可以的。

2.关于 parameterType 的取值问题，对于基本类型我们可以直接写成 int,short,double.....也可以写成 java.lang.Integer。

3.字符串可以写成 string,也可以写成 java.lang.String

1.4.3 加入删除的测试方法

在原有的 UserTest 类中加入测试方法，如下：

```
public class UserTest {
    @Test
    public void testdeleteUser() throws Exception {
        InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
        SqlSession session = factory.openSession();
        UserMapper userMapper = session.getMapper(UserMapper.class);

        userMapper.deleteUser(44);
        session.commit();//提交事务
        session.close();
        is.close();
    }
```

下面是测试的结果：

```
Created connection 1754638213.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6895a785]
==>  Preparing: delete from user where id=?
==> Parameters: 44(Integer)
<==    Updates: 1
Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6895a785]
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@6895a785]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@6895a785]
Returned connection 1754638213 to pool.
```

1.4.4 Mybatis 默认参数的源码分析

Mybatis 默认情况下支持参数写别名，这就导致了我们前面在看基本类型时，既可以写成 int，也可以写成 java.lang.Integer,默认支持的到底有哪些类型？从源码角度怎么去找?

先回答第一个问题：参考官方文档的说明(第 19 页)

Alias	Mapped Type
_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

这些都是支持的默认别名。我们也可以从源码角度来看它们分别都是如何定义出来的。可以参考 TypeAliasRegistery.class 的源码。

TypeAliasRegistry.class

```
41
42  public TypeAliasRegistry() {
43    registerAlias("string", String.class);
44
45    registerAlias("byte", Byte.class);
46    registerAlias("long", Long.class);
47    registerAlias("short", Short.class);
48    registerAlias("int", Integer.class);
49    registerAlias("integer", Integer.class);
50    registerAlias("double", Double.class);
51    registerAlias("float", Float.class);
52    registerAlias("boolean", Boolean.class);
53
54    registerAlias("byte[]", Byte[].class);
55    registerAlias("long[]", Long[].class);
56    registerAlias("short[]", Short[].class);
57    registerAlias("int[]", Integer[].class);
```

1.5 Mybatis 实现用户模糊查询

现在来实现根据用户名查询用户信息，此时如果用户名想用模糊搜索的话，我就可以想到前面 Web 课程中所学的模糊查询来实现。

1.5.1 在 UserMapper 类中添加模糊查询方法

可以在 UserMapper 类中添加一个 findUserByUsername()的方法，如下：

```
public interface UserMapper {
    //模糊查询
    public List<User> findUserByUsername(String username);
```

1.5.2 在 UserMapper.xml 文件中加入模糊查询的配置

下面在 UserMapper.xml 文件中加入模糊查询的配置代码，如下：

```
<!-- 根据用户名查询用户信息 -->
<select id="findUserByUsername" parameterType="string"
        resultType="com.itheima.domain.User">
    select * from user where username like #{username}
</select>
```

注意：此时的#{username}中的因为这时候是普通的参数，所以它的起名是随意的，比如我们改成#{abc}也是可以的。

1.5.3 加入模糊查询的测试方法

在 UserTest 类中加入模糊测试的 testFindLike()方法，如下：

```
@Test
public void testFindLike() throws Exception {
    InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
    SqlSession session = factory.openSession();
    UserMapper userMapper = session.getMapper(UserMapper.class);

    List<User> list = userMapper.findUserByUsername("%王%");
    for(User user :list) {
        System.out.println(user);
    }
    session.commit();//提交事务
    session.close();
    is.close();
}
```

在控制台输出的执行 SQL 语句如下：

```
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1d29cf23]
==>  Preparing: select * from user where username like ?
==> Parameters: %王%(String)
```

我们在 UserMapper.xml 配置文件中没有加入%来作为模糊查询的条件，所以在传入字符串实参时，就需要给定模糊查询的标识%。配置文件中的#{username}也只是一个占位符，所以 SQL 语句显示为“？”。如何将模糊查询的匹配符%写到配置文件中呢？

1.5.4 模糊查询的另一种配置方式

第一步：编写 UserMapper.xml 文件，配置如下：

```
<!-- 根据用户名查询用户信息 -->
 <select id="findUserByUsername" parameterType="string"
         resultType="com.itheima.domain.User">
    select * from user where username like '%${value}%'
</select>
```

我们在上面将原来的#{}占位符，改成了${value}。注意如果用模糊查询的这种写法，那么${value}的写法就是固定的，不能写成其它名字。

第二步：编写测试方法，如下：

```
@Test
public void testFindLike() throws Exception {
    InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
    SqlSession session = factory.openSession();
    UserMapper userMapper = session.getMapper(UserMapper.class);

    List<User> list = userMapper.findUserByUsername("王");
    for(User user :list) {
        System.out.println(user);
    }
    session.commit();//提交事务
    session.close();
    is.close();
}
```

在控制台输出的执行 SQL 语句如下：

```
Created connection 813656972.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@307f6b8c]
==>  Preparing: select * from user where username like '%王%'
```

可以发现，我们在程序代码中就不需要加入模糊查询的匹配符%了，这两种方式的实现效果是一样的，但执行的语句是不一样的。

1.5.5 #{}与${}的区别

- #{}表示一个占位符号

通过#{}可以实现 preparedStatement 向占位符中设置值，自动进行 java 类型和 jdbc 类型转换，#{}可以有效防止 sql 注入。 #{}可以接收简单类型值或 pojo 属性值。 如果 parameterType 传输单个简单类型值，#{}括号中可以是 value 或其它名称。

- ${}表示拼接 sql 串

通过${}可以将 parameterType 传入的内容拼接在 sql 中且不进行 jdbc 类型转换， ${}可以接收简单类型值或 pojo 属性值，如果 parameterType 传输单个简单类型值，${}括号中只能是 value。

1.5.6 模糊查询的${value}源码分析

我们在将模糊查询的匹配符%写到配置文件时，就用到一个固定写法${value}，如下匹配:

```
<!-- 根据用户名查询用户信息 -->
 <select id="findUserByUsername" parameterType="string"
         resultType="com.itheima.domain.User">
    select * from user where username like '%${value}%'
</select>
```

那么为什么一定要写成${value}呢？我们一起来看 TextSqlNode 类的源码:

```
TextSqlNode.class
69      @Override
70      public String handleToken(String content) {
71        Object parameter = context.getBindings().get("_parameter");
72        if (parameter == null) {
73          context.getBindings().put("value", null);
74        } else if (SimpleTypeRegistry.isSimpleType(parameter.getClass())) {
75          context.getBindings().put("value", parameter);
76        }
77        Object value = OgnlCache.getValue(content, context.getBindings());
78        String srtValue = (value == null ? "" : String.valueOf(value)); //
79        checkInjection(srtValue);
80        return srtValue;
81      }
```

这就说明了源码中指定了读取的 key 的名字就是”value”，所以我们在绑定参数时就只能叫 value 的名字了。

1.6 Mybatis 与 JDBC 编程的比较

1.数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在 SqlMapConfig.xml 中配置数据链接池，使用连接池管理数据库链接。

2.Sql 语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。

解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。

3.向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis 自动将 java 对象映射至 sql 语句，通过 statement 中的 parameterType 定义输入参数的类型。

4.对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。

解决：Mybatis 自动将 sql 执行结果映射至 java 对象，通过 statement 中的 resultType 定义输出结果的类型。

第2章 Mybatis 的参数深入

Mybatis 的映射文件其实就是与 DAO 相对应，因为 DAO 中的方法有输入参数及返回结果，那么在 Mybatis 的映射文件中自然也就有与之对应的参数和返回结果。

在 Mybatis 的映射文件中参数用 parameterType 来代表，它的值可以是基本类型，也可以是包装的对象，这一点我们第二天学习中就使用过。

在 Mybatis 的映射文件中返回结果用 resultType 或 resultMap 来代表。

2.1 Mybatis 的参数

2.1.1 parameterType(输入类型)

2.1.2 传递简单类型

2.1.3 传递 pojo 对象

Mybatis 使用 ognl 表达式解析对象字段的值，#{}或者${}括号中的值为 pojo 属性名称。

2.1.4 传递 pojo 包装对象

开发中通过 pojo 传递查询条件 ，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件(比如将用户购买商品信息也作为查询条件)，这时可以使用包装对象传递输入参数。

Pojo 类中包含 pojo。

需求：根据用户名查询用户信息，查询条件放到 QueryVo 的 user 属性中。

2.1.4.1 QueryVo

QueryVo.java

```
1  package com.itheima.domain;
2
3  public class QueryVo {
4      private User user;
5      public User getUser() {
6          return user;
7      }
8      public void setUser(User user) {
9          this.user = user;
10     }
11 }
```

2.1.4.2 UserDao 接口

UserDao.java

```
1  package com.itheima.dao;
2
3  import com.itheima.domain.QueryVo;
5
6  public interface UserDao {
7      //根据用户名查询用户对象
8       public User findByVo(QueryVo vo);
9  }
```

2.1.4.3 Mapper 文件

UserDao.xml

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper
3  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5  <mapper namespace="com.itheima.dao.UserDao">
6      <select id="findByVo" parameterType="queryvo" resultType="user">
7          select * from user where username=#{user.username}
8      </select>
9  </mapper>
```

如果我们使用的是包装类作为参数，比如这个示例的 QueryVo 类作为 findByVo()方法的参数，那么在使用时，因为 QueryVo 类中有一个 User 类的 user 对象，而这个 user 对象中才能找到 username 属性，所以我们在访问属性时，就使用 OGNL 表达式才访问对象的属性。

为了简化参数和返回值的编写，我们在 SqlMapConfig.xml 文件中加入了包扫描的配置。

SqlMapConfig.xml

```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>
    <properties resource="jdbc.properties"></properties>
    <typeAliases>
       <package name="com.itheima.domain"/>
    </typeAliases>
    <environments default="mysql">
```

上面的 typeAliases 的子结点 package 就是指定了包扫描，这样我们就可以在映射文件中方便使用了。

2.1.4.4 测试包装类作为参数

```java
@Test
public void testFindByVo() throws Exception{
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);

    QueryVo vo = new QueryVo();
    User user = new User();
    user.setUsername("传智播客");
    vo.setUser(user);

    User user02 = userDao.findByVo(vo);
    System.out.println(user02);
    sqlSession.close();
}
```

2.2 Mybatis 的输出结果封装

2.2.1 resultType(输出类型)

2.2.1.1 输出简单类型

看下边的例子输出整型：

Mapper.xml 文件

```xml
<!-- 获取用户列表总数 -->
    <select id="findUserCount" resultType="int">
       select count(1) from user
    </select>
```

Mapper 接口

```java
public int findUserCount() throws Exception;
```

调用：

```
Public void testFindUserCount() throws Exception{
        //获取session
        SqlSession session = sqlSessionFactory.openSession();
        //获取mapper接口实例
        UserDao userDao = session.getMapper(UserDao.class);

        //传递Hashmap对象查询用户列表
        int count = userDao.findUserCount();
        //关闭session
        session.close();
    }
```

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。
使用 session 的 selectOne 可查询单条记录。

2.2.1.2 输出 pojo 对象

2.2.1.3 输出 pojo 列表

2.3 resultMap 结果类型

resultType 可以指定 pojo 将查询结果映射为 pojo，但需要 pojo 的属性名和 sql 查询的列名一致方可映射成功。

如果 sql 查询字段名和 pojo 的属性名不一致，可以通过 resultMap 将字段名和属性名作一个对应关系 ，resultMap 实质上还需要将查询结果映射到 pojo 对象中。

resultMap 可以实现将查询结果映射为复杂类型的 pojo，比如在查询结果映射对象中包括 pojo 和 list 实现一对一查询和一对多查询。

2.3.1 UserDao.xml 定义

需求：如果返回的列名与实体类的属性不一致时，我们就不能封装结果集到指定的实体对象。

SQL：select id id_,username username_ ,birthday birthday_ ,sex sex_ ,address address_
from user where username='传智播客'

通过改别名的方式，现在返回结果集的列名已经与 User 类的属性名不相同了。

```
<mapper namespace="com.itheima.dao.UserDao">

    <select id="findByVo" parameterType="queryvo" resultType="user">
       select id id_,username username_ ,birthday birthday_ ,sex sex_ ,address address_
       from user where username=${user.username}
    </select>
```

我们再次运行程序，发现出错如下：

```
Failure Trace
org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Unknown column
### The error may exist in com/itheima/dao/UserDao.xml
### The error may involve com.itheima.dao.UserDao.findByVo-Inline
```

这个错误就是我们返回结果集的列名与对应 User 的属性名不对应造成的，使用 resultMap 可以将建立起结果集的列与实体类的属性名之间的映射，这样就可以解决列名与属性名不相同的问题。

2.3.2 定义 resultMap

由于上边的 mapper.xml 中 sql 查询列和 Users.java 类属性不一致，需要定义 resultMap：userListResultMap 将 sql 查询列和 Users.java 类属性对应起来

UserDao.xml

```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="com.itheima.dao.UserDao">

    <select id="findByVo" parameterType="queryvo" resultMap="userResultMap">
        select id id_,username username_,birthday birthday_,sex sex_ ,address address_
        from user where username=#{user.username}
    </select>

    <!-- type="user"代表User类  id="userResultMap"代表resultMap的名称 -->
    <resultMap type="user" id="userResultMap">
        <id column="id_" property="id"/>
        <result column="username_" property="username"/>
        <result column="birthday_" property="birthday"/>
        <result column="sex_" property="sex"/>
        <result column="address_" property="address"/>
    </resultMap>
</mapper>
```

<id />：此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个<id />。

property：表示 User 类的属性。

column：表示 sql 查询出来的字段名。

column 和 property 放在一块儿表示将 sql 查询出来的字段映射到指定的 pojo 类属性上。

<result />：普通结果，即 pojo 的属性。

2.3.3 测试效果

```
@Test
public void testFindByVo() throws Exception{
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);

    QueryVo vo = new QueryVo();
    User user = new User();
    user.setUsername("传智播客");
    vo.setUser(user);

    User user02 = userDao.findByVo(vo);
    System.out.println(user02);
    sqlSession.close();
}
```

第3章 Mybatis 实现 DAO 层开发

使用 Mybatis 开发 Dao，通常有两个方法，即原始 Dao 开发方式和 Mapper 接口代理开发方式。而现在主流的开发方式是接口代理开发方式，这种方式总体上更加简便。我们的课程讲解也主要以接口代理开发方式为主。

3.1 Mybatis 实现 DAO 的传统开发方式

传统方式开发 DAO 的说明，需求如下：
1、根据用户 id 查询一个用户信息
2、根据用户名称模糊查询用户信息列表
3、添加用户信息

3.1.1 SqlSession 的使用分析

SqlSession 中封装了对数据库的操作，如：查询、插入、更新、删除等。
通过 SqlSessionFactory 创建 SqlSession，而 SqlSessionFactory 是通过 SqlSessionFactoryBuilder 进行创建。

3.1.1.1 SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 用于创建 SqlSessionFacoty，SqlSessionFacoty 一旦创建完成就不需要 SqlSessionFactoryBuilder 了，因为 SqlSession 是通过 SqlSessionFactory 生产，所以可以将 SqlSessionFactoryBuilder 当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

3.1.1.2 SqlSessionFactory

SqlSessionFactory 是一个接口，接口中定义了 openSession 的不同重载方法，SqlSessionFactory 的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理 SqlSessionFactory。

3.1.1.3 SqlSession

SqlSession 是一个面向用户的接口， sqlSession 中定义了数据库操作方法。

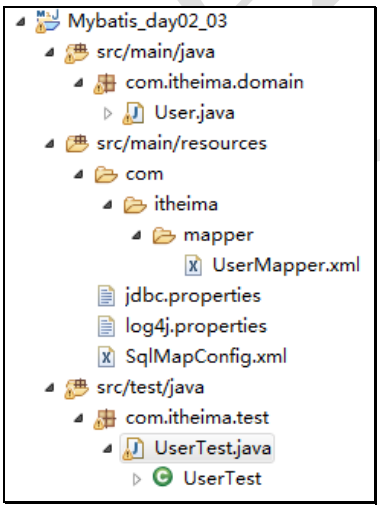
每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不能共享使用，它也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将 SqlSession 实例的引用放在一个类的静态字段或实例字段中。

打开一个 SqlSession；使用完毕就要关闭它。通常把这个关闭操作放到 finally 块中以确保每次都能执行关闭。如下：

```
SqlSession session = sqlSessionFactory.openSession();
try {
        // do work
} finally {
      session.close();
}
```

3.1.2 Mybatis 传统方式开发 DAO

将原有工程复制过来，并修改结构如下：



3.1.2.1 开发 UserMaaper.xml 映射文件

在 UserMapper.xml 映射文件中，按照需求开发相关的配置，具体需求如下：
1、根据用户 id 查询一个用户信息
2、根据用户名称模糊查询用户信息列表
3、添加用户信息
重新配置 UserMapper.xml 文件，配置如下：

UserMapper.xml

```
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="test">
  <!-- 根据id获取用户信息 -->
    <select id="findUserById" parameterType="int" resultType="com.itheima.domain.User">
        select * from user where id = #{id}
    </select>

    <!-- 根据用户名模糊查询 -->
    <select id="findUserByUsername" parameterType="string"
            resultType="com.itheima.domain.User">
       select * from user where username like '%${value}%'
    </select>

    <!-- 新增 -->
    <insert id="saveUser" parameterType="com.itheima.domain.User">
      <!-- keyProperty代表要返回的值名称  order:取值为AFTER代表插入后的行为  resultType代表返回值的类型 -->
      <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">
          select last_insert_id();
      </selectKey>
       insert into user(username,birthday,sex,address)
       values(#{username},#{birthday},#{sex},#{address});
    </insert>
</mapper>
```

3.1.2.2 开发 UserDao 接口及 UserDaoImpl 实现类

第一步：开发 UserDao 接口

UserDao.java

```
package com.itheima.dao;

import java.util.List;

import com.itheima.domain.User;

public interface UserDao {
    public User findUserById(Integer id);

    public List<User> findUserByUsername(String username);

    public Integer saveUser(User user);
}
```

第二步：开发 UserDaoImpl 实现类

UserDaoImpl.java

```
public class UserDaoImpl implements UserDao {
    //我们将SqlSessionFactory工厂设置成员变量
    private SqlSessionFactory sqlSessionFactory;
    //通过方法实现sqlSessionFactory注值
    public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {
        this.sqlSessionFactory = sqlSessionFactory;
    }
    public User findUserById(Integer id) {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //selectOne()用于查询一条记录,参数：namespace+id的取值
        User user = sqlSession.selectOne("test.findUserById",id);
        sqlSession.close();
        return user;
    }
    public List<User> findUserByUsername(String username) {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        List<User> userList = sqlSession.selectList("test.findUserByUsername",username);
        sqlSession.close();
        return userList;
    }
    public Integer saveUser(User user) {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        Integer id = sqlSession.insert("test.saveUser",user);
        sqlSession.commit();
        sqlSession.close();
        return id;
    }
```

在开发 UserDaoImpl 实现类时，要注意 SqlSessionFactory 是通过构造方法传入进来的。在编写实现类中方法时，要注意所调用的 selectOne()的参数是由 namespace+id 一起来构成，但 namespace 和 id 的取值是我们自己去起名，不要求像我们前面通过 Mapper 代理方式一样，必须与包名和方法相同。这一点是很随意，因为我们自己在代码中要求去手动编码来指定 namespace 和 id。

3.1.2.3 编写测试类

第一步：为了更好的在测试类中将 SqlSessionFactory 对象构造出来，我们特别在测试类中加入一个初始化的 init()方法，这个方法能够实现 SqlSessionFactory 对象初始化工作。

UserTest.java

```
package com.itheima.test;

import java.io.InputStream;

public class UserTest {
    private SqlSessionFactory sqlSessionFactory = null;
    @Before
    public void init() throws Exception {
        InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);
    }
```

第二步：1.编写相关的测试方法，实现根据 id 查询效果如下：

```
@Test
public void testFindById() throws Exception {
    UserDao userDao = new UserDaoImpl(sqlSessionFactory);
    User user = userDao.findUserById(41);
    System.out.println(user);
}
```

测试结果如下：

```
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@3b084709]
==>  Preparing: select * from user where id = ?
==> Parameters: 41(Integer)
<==      Total: 1
```

2.实现根据用户名的模糊查询如下：

```
@Test
public void testFindByLike() throws Exception {
    UserDao userDao = new UserDaoImpl(sqlSessionFactory);
    List<User> userList = userDao.findUserByUsername("老");
    for(User user :userList) {
        System.out.println(user);
    }
}
```

测试结果如下：

```
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@1b68ddbd]
==>  Preparing: select * from user where username like '%老%'
==> Parameters:
<==      Total: 6
```

3.实现添加用户，代码如下：

```
@Test
public void testSaveUser() throws Exception {
    UserDao userDao = new UserDaoImpl(sqlSessionFactory);
    User user = new User("小二王","女",new Date(),"北京金燕龙");
    int id = userDao.saveUser(user);
    System.out.println(id);
}
```

测试效果如下：

```
==>  Preparing: insert into user(username,birthday,sex,address) values(?,?,?,?);
==> Parameters: 小二王(String), 2018-03-02 15:09:37.496(Timestamp), 女(String), 北京金燕龙(String)
<==    Updates: 1
==>  Preparing: select last_insert_id();
==> Parameters:
<==      Total: 1
```

3.1.3 Mybatis 传统开发方式源码分析 SQL 语句执行过程

现在我们可以通过调用 SqlSession 接口中的方法实现 CRUD 操作了，那么 SqlSession 接口中的方法是如何实现的。

下面一起来分析 sqlSession 接口中的 insert()方法的执行过程，我们进行源码跟踪发现了 Mybatis 只是对于 jdbc 的封装。

```
    insert()方法的执行过程分析
            * DefaultSqlSession----> return update(statement, parameter);185 行
            * 进入 DefaultSqlSession---> public int update(String statement, Object
parameter) {方法定义
            * 进入 198 行--- return executor.update(ms, wrapCollection(parameter));
```

```
            * 找到 executor 对象的 update 方法 ------------ 查看 update() 方法
-----------------BaseExecutor 类的 update()方法
            * 进入 BaseExecutor 类的 update()第 117 行 return doUpdate(ms, parameter);
            * 查看 doUpdate()方法------------------查看 SimpleExecutor 类的 doUpdate()方法
            * 查看 doUpdate()方法------------------stmt = prepareStatement(handler,
ms.getStatementLog());  49 行
            * 查看 prepareStatement()方法----------进入这个方法的 84,85 行
            * 得到代码： Connection connection = getConnection(statementLog);
                  stmt = handler.prepare(connection, transaction.getTimeout());
                    查看代码：
                  stmt = handler.prepare(connection, transaction.getTimeout());
//StatementHandler 的子类 BaseStatementHandler 类

                    查看 BaseStatementHandler 类的 prepare（）方法，该类的 84 行
                   再进入这个类的 88 行,找到代码：
                statement = instantiateStatement(connection);

                  进入 SimpleStatementHandler 类的 instantiateStatement(connection)
            protected Statement instantiateStatement(Connection connection) throws
SQLException {
              if (mappedStatement.getResultSetType() != null) {
                return
connection.createStatement(mappedStatement.getResultSetType().getValue(),
ResultSet.CONCUR_READ_ONLY);
              } else {
                return connection.createStatement();
              }
            }
```

3.2 Mybatis 实现 DAO 代理开发方式

采用 Mybatis 的代理开发方式实现 DAO 层的开发，这种方式是我们后面进入企业的主流方式，所以我们在前面的入门程序实现的 CRUD 就是采用这种方式来做的。现在一起再使用这种方式来开发我们的 DAO。

代理方式开发 DAO 规范总结：

Mapper 接口开发方法只需要程序员编写 Mapper 接口（相当于 Dao 接口），由 Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边 Dao 接口实现类方法。

Mapper 接口开发需要遵循以下规范：

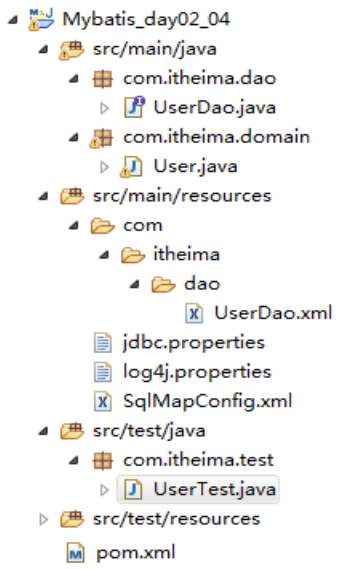
1、 Mapper.xml 文件中的 namespace 与 mapper 接口的类路径相同。

2、 Mapper 接口方法名和 Mapper.xml 中定义的每个 statement 的 id 相同

3、 Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同

4、 Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同

将原来的工程进行备份，并进行修改成如下的结构：



3.2.1 修改原有的 UserMapper.xml 映射文件

将原有的 UserMapper.xml 文件名称修改为 UserDao.xml

UserDao.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="com.itheima.dao.UserDao">
  <!-- 根据id获取用户信息 -->
    <select id="findUserById" parameterType="int" resultType="com.itheima.domain.User">
        select * from user where id = #{id}
    </select>
   <!-- 根据用户名模糊查询 -->
   <select id="findUserByUsername" parameterType="string"
            resultType="com.itheima.domain.User">
       select * from user where username like '%${value}%'
   </select>
   <!-- 新增 -->
   <insert id="saveUser" parameterType="com.itheima.domain.User">
     <!-- keyProperty代表要返回的值名称  order:取值为AFTER代表插入后的行为  resultType代表返回值的类型 -->
     <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">
         select last_insert_id();
     </selectKey>
      insert into user(username,birthday,sex,address)
      values(#{username},#{birthday},#{sex},#{address});
   </insert>
</mapper>
```

就是DAO接口就在的包名.接口名

就是DAO接口的方法名

3.2.2 修改 SqlMapConfig.xml 文件

因为我们已经将 UserMapper.xml 文件名称修改为 UserDao.xml 文件了，所以现在也要将 SqlMapConfig.xml 文件中加载的映射文件进行更新。

```
<!-- 加载映射文件 -->
<mappers>
   <mapper resource="com/itheima/dao/UserDao.xml"/>
</mappers>
```

3.2.3 编写测试方法

在 UserTest 类中加入根据 ID 查询的测试方法

```
@Test
public void testFindUserById() throws Exception{
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = userDao.findUserById(41);
    System.out.println(user);
    sqlSession.close();
}
```

在 UserTest 类中加入模糊查询的测试方法

```
@Test
public void testFindUserByUsername() throws Exception{
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    List<User> userList = userDao.findUserByUsername("老");
    for(User user :userList) {
        System.out.println(user);
    }
    sqlSession.close();
}
```

在 UserTest 类中加入保存用户的测试方法

```
@Test
public void testSaveUser() throws Exception{
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = new User("王婆","女",new Date(),"深圳");
    userDao.saveUser(user);
    sqlSession.commit();
    sqlSession.close();
}
```

我们总结一下会发现，采用代理方式开发 DAO 是非常简单的，只需要定义 DAO 接口并开发出它所对应的 sql 映射文件，就可以实现 DAO。我们一起思考一下，这个接口的实现类是没有的，那么 DAO 接口的实现是如何产生的？

3.2.4 Mybatis 代理方式实现 DAO 源码分析

通过对源代码进行跟踪，我们会发现在调用 SqlSession 接口的 getMapper()方法时，它的底层进一步调用了 JDK 动态代理方式来生成代理子类对象的，这个过程如下：

```
    getMapper(Class<T> type, SqlSession sqlSession)
            protected T newInstance(MapperProxy<T> mapperProxy) {
                return  (T)  Proxy.newProxyInstance(mapperInterface.getClassLoader(),
```

```
new Class[] { mapperInterface }, mapperProxy);
              }
              当调用方法时，会去调用 MapperProxy 的 invoke()方法

             这个 invoke()的最后一行： return mapperMethod.execute(sqlSession, args);

             继续跟踪这个方法：execute(sqlSession, args);
             下面是它的方法体
             Object result;
       switch (command.getType()) {
         case INSERT: {
        Object param = method.convertArgsToSqlCommandParam(args);
           result = rowCountResult(sqlSession.insert(command.getName(), param));
           break;
         }
         case UPDATE: {
           Object param = method.convertArgsToSqlCommandParam(args);
           result = rowCountResult(sqlSession.update(command.getName(), param));
           break;
         }
         case DELETE: {
           Object param = method.convertArgsToSqlCommandParam(args);
           result = rowCountResult(sqlSession.delete(command.getName(), param));
           break;
         }
         case SELECT:
           if (method.returnsVoid() && method.hasResultHandler()) {
             executeWithResultHandler(sqlSession, args);
             result = null;
           } else if (method.returnsMany()) {
             result = executeForMany(sqlSession, args);
           } else if (method.returnsMap()) {
             result = executeForMap(sqlSession, args);
           } else if (method.returnsCursor()) {
             result = executeForCursor(sqlSession, args);
           } else {
             Object param = method.convertArgsToSqlCommandParam(args);
             result = sqlSession.selectOne(command.getName(), param);
           }
           break;
         case FLUSH:
           result = sqlSession.flushStatements();
           break;
         default:
```

```
            throw new BindingException("Unknown execution method for: " +
command.getName());
        }
```

第4章 SqlMapConfig.xml配置文件

4.1 配置内容

SqlMapConfig.xml 中配置的内容和顺序如下：

properties（属性）
settings（全局配置参数）
typeAliases（类型别名）
typeHandlers（类型处理器）
objectFactory（对象工厂）
plugins（插件）
environments（环境集合属性对象）
    environment（环境子属性对象）
        transactionManager（事务管理）
        dataSource（数据源）
mappers（映射器）

4.2 properties（属性）

SqlMapConfig.xml 可以引用 java 属性文件中的配置信息如下：

在 classpath 下定义 db.properties 文件，

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8
jdbc.username=root
jdbc.password=root
```

SqlMapConfig.xml 引用如下：

```
<properties resource="db.properties"/>
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
```

```
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}"/>
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
        </dataSource>
    </environment>
</environments>
```

注意： MyBatis 将按照下面的顺序来加载属性：

- 使用 properties 元素加载的外部属性文件优先级最高。
- 然后会读取 properties 元素中 resource 加载的属性，它会覆盖已读取的同名属性。

4.3 typeAliases（类型别名）

4.3.1 mybatis 支持别名：

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

这是之前我们讲的 Mybatis 支持的默认别名，我们也可以采用自定义别名方式来开发。

4.3.2 自定义别名：

在 SqlMapConfig.xml 中配置：

```
<typeAliases>
    <!-- 单个别名定义 -->
    <typeAlias alias="user" type="com.itheima.domain.User"/>
    <!-- 批量别名定义，扫描整个包下的类，别名为类名（首字母大写或小写都可以） -->
    <package name="com.itheima.domain"/>
    <package name="其它包"/>
</typeAliases>
```

4.4 mappers（映射器）

Mapper 配置的几种方法：

4.4.1 <mapper resource=" " />

使用相对于类路径的资源

如：<mapper resource="com/itheima/dao/UserDao.xml" />

4.4.2 <mapper class=" " />

使用 mapper 接口类路径

如：<mapper class="com.itheima.dao.UserDao"/>

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

4.4.3 <package name=""/>

注册指定包下的所有 mapper 接口

如：<package name="cn.itcast.mybatis.mapper"/>

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

小结：通过本次课程，学员应当熟练掌握 Mybatis 的基本操作和基本实现原理。