第二讲 MyBatis 操作单表

一、框架概述

1. 什么是框架

框架（Framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法;另一种 定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面而后者是从目的方面给出的定义。
简而言之，框架其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。简单说就是使用别 人搭好的舞台，你来做表演。而且，框架一般是成熟的，不断升级的软件。

2. 框架要解决的问题

框架要解决的最重要的一个问题是技术整合的问题，在 J2EE 的 框架中，有着各种各样的技术，不同的软件企业需要从 J2EE 中选择不同的技术，这就使得软件企业最终的应用依赖于这些技术，技术自身的复杂性和技术的风险性将会直接对应用造成冲击。而应用是软件企业的核心，是竞争力的关键所在，因此应该将应用自身的设计和具体的实现技术解耦。这样，软件企业的研发将集中在应用的设计上，而不是具体的技术实现，技术实现是应用的底层支撑，它不应该直接对应用产生影响。

3. 分层开发下的常见框架

解决数据的持久化问题的框架

作为持久层的框架，还有一个封装程度更高的框架就是 Hibernate，但这个框架因为各种原因目前在国内的 流行程度下降太多，现在公司开发也越来越少使用。目前使用 Spring Data

来实现数据持久化也是一种趋势。

4. MyBatis 框架概述

mybatis 是一个优秀的基于 java 的持久层框架，它内部封装了 jdbc，使开发者只需要关注 sql 语句本身， 而不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。

mybatis 通过 xml 或注解的方式将要执行的各种 statement 配置起来，并通过 java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并 返回。

采用 ORM 思想解决了实体和数据库映射的问题，对 jdbc 进行了封装，屏蔽了 jdbc api 底层访问细节，使我 们不用与 jdbc api 打交道，就可以完成对数据库的持久化操作。

二、 Mybatis 框架快速入门

1. 搭建 Mybatis 开发环境

创建 mybatis01 的工程，工程信息如下：

Groupid:club.banyuan

ArtifactId:mybatis01

Packing:jar

2. 添加 Mybatis3.4.5 的坐标

在 pom.xml 文件中添加 Mybatis3.4.5 的坐标，如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
```

```
    <artifactId>mybatis1</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.5</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.10</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.19</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>
    </dependencies>

</project>
```

3. 编写 User 实体类

```
package club.banyuan.entity;
```

```
public class User {
	private Integer id;
	private String loginName;
	private String userName;
	private String password;
	private Integer sex;
	private String email;
	private String mobile;

	public User(){}

	public User(Integer id, String loginName, String userName, String password, Integer
sex, String email, String mobile) {
		this.id = id;
		this.loginName = loginName;
		this.userName = userName;
		this.password = password;
		this.sex = sex;
		this.email = email;
		this.mobile = mobile;
	}

	public Integer getId() {
		return id;
	}
	public void setId(Integer id) {
		this.id = id;
	}
	public String getLoginName() {
		return loginName;
	}
	public void setLoginName(String loginName) {
		this.loginName = loginName;
	}
	public String getUserName() {
		return userName;
```

```
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public Integer getSex() {
        return sex;
    }
    public void setSex(Integer sex) {
        this.sex = sex;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getMobile() {
        return mobile;
    }
    public void setMobile(String mobile) {
        this.mobile = mobile;
    }
}
```

4. 编写持久层接口 IUserDao

```
package club.banyuan.dao;

import club.banyuan.entity.User;

import java.util.List;

public interface UserDao {
    List<User> getAll();
}
```

5. 编写持久层接口的映射文件 IUserDao.xml

要求：

创建位置：必须和持久层接口在相同的包中。

名称：必须以持久层接口名称命名文件名，扩展名是.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="club.banyuan.dao.UserDao">
    <!-- 配置查询所有操作 -->
    <select id="getAll" resultType="club.banyuan.entity.User">
    select * from user
  </select>
</mapper>
```

6. 编写 SqlMapConfig.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 配置 mybatis 的环境 -->
    <environments default="mysql">
        <!-- 配置 mysql 的环境 -->
        <environment id="mysql">
            <!-- 配置事务的类型 -->
            <transactionManager type="JDBC">
            </transactionManager>
            <!-- 配置连接数据库的信息：用的是数据源(连接池) -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/shoppingstreet?&amp;useSSL=false&amp;serverTim
e"/>
                <property name="username" value="root"/>
                <property name="password" value="rootroot"/>
            </dataSource>
        </environment>
    </environments>
    <!-- 告知 mybatis 映射配置的位置 -->
    <mappers>
        <mapper resource="club/banyuan/dao/UserDao.xml"/>
    </mappers>
</configuration>
```

7. 编写测试类

```
 import club.banyuan.dao.UserDao;
import club.banyuan.entity.User;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;
import java.util.List;

public class MybatisTest {
    public static void main(String[] args)throws Exception {
        //1.读取配置文件
        InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");
        //2.创建 SqlSessionFactory 的构建者对象
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        //3.使用构建者创建工厂对象
        SqlSessionFactory factory = builder.build(in);
        //4.使用 SqlSessionFactory 生产 SqlSession 对象
        SqlSession session = factory.openSession();
        //5.使用 SqlSession 创建 dao 接口的代理对象
        UserDao userDao = session.getMapper(UserDao.class);
        //6.使用代理对象执行查询所有方法
        List<User> users = userDao.getAll();
        for(User user : users) {
            System.out.println(user);
        }
        //7.释放资源
        session.close();
        in.close();
    }
}
```

8. 总结

- 持久层接口和持久层接口的映射配置必须在相同的包下
- 持久层映射配置中 mapper 标签的 namespace 属性取值必须是持久层接口的全限定类名
- SQL 语句的配置标签<select>,<insert>,<delete>,<update>的 id 属性必须和持久层接口的 方法名相同。

三、实现 CRUD 操作

1. 根据 ID 查询

- 在持久层接口中添加 findById 方法

```
/**
 * 根据 id 查询
 * @param userId
 * @return
 */
User findById(Integer userId);
```

- 在用户的映射配置文件中配置

```
<!-- 根据 id 查询 -->
<select id="findById" resultType="club.banyuan.entity.User" parameterType="int">
 select * from user where id = #{uid}
</select>
```

细节：

resultType 属性： 用于指定结果集的类型。

parameterType 属性： 用于指定传入参数的类型。

sql 语句中使用#{}字符： 它代表占位符，相当于原来 jdbc 部分所学的?，都是用于执行语句时替换实际的数据。 具体的数据是由#{}里面的内容决定的。

#{}中内容的写法： 由于数据类型是基本类型，所以此处可以随意写。

- 在测试类添加测试

```
public class MybastisCRUDTest {
   private InputStream in ;
   private SqlSessionFactory factory;
   private SqlSession session;
   private IUserDao userDao;
      @Test
   public void testFindOne()   {
   //6.执行操作
    User user = userDao.findById(41);
   System.out.println(user);
       }
 @Before
//在测试方法执行之前执行
   public void init()throws Exception {
    //1.读取配置文件
    in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2.创建构建者对象
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    //3.创建 SqlSession 工厂对象
    factory = builder.build(in);
   //4.创建 SqlSession 对象
    session = factory.openSession();
   //5.创建 Dao 的代理对象
    userDao = session.getMapper(IUserDao.class);
   }
 @After
//在测试方法执行完成之后执行
   public void destroy() throws Exception{
   session.commit();
   //7.释放资源
    session.close();
   in.close();
   }
 }
```

2. 保存操作

- 在持久层接口中添加新增方法

```
/**
  * 保存用户
   * @param user
   * @return 影响数据库记录的行数
   */
  int saveUser(User user);
```

- 在用户的映射配置文件中配置

```
<!-- 保存用户-->
 <insert id="saveUser" parameterType="club.banyuan.entity.User">
  insert into user(username,birthday,sex,address)
values(#{username},#{birthday},#{sex},#{address})
 </insert>
```

细节：

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。

sql 语句中使用#{}字符： 它代表占位符，相当于原来 jdbc 部分所学的?，都是用于执行语句时替换实际的数据。 具体的数据是由#{}里面的内容决定的。

#{}中内容的写法： 由于我们保存方法的参数是 一个 User 对象，此处要写 User 对象中的属性名称。 它用的是 ognl 表达式。

ognl 表达式： 它是 apache 提供的一种表达式语言，全称是：

Object Graphic Navigation Language 对象图导航语言

它是按照一定的语法格式来获取数据的。

语法格式就是使用 #{对象.对象}的方式

#{user.username}它会先去找 user 对象，然后在 user 对象中找到 username 属性，并调用 getUsername()方法把值取出来。但是我们在 parameterType 属性上指定了实体类名称，所以可以省略 user. 而直接写 username。

- 添加测试类中的测试方法

```
@Test public void testSave(){
  User user = new User();
  user.setUsername("modify User property");
 user.setAddress("北京市顺义区");
  user.setSex("男");
```

```
  user.setBirthday(new Date());
  System.out.println("保存操作之前："+user);
  //5.执行保存方法
  userDao.saveUser(user);

 System.out.println("保存操作之后："+user);
 }
```

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？

- 使用:session.commit();来实现事务提交。

加入事务提交后的代码如下：

```
@After
//在测试方法执行完成之后执行
 public void destroy() throws Exception{
  session.commit();
  //7.释放资源
  session.close();
  in.close();
 }
```

- 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。

```
<insert id="saveUser" parameterType="USER">
  <!-- 配置保存时获取插入的 id -->
  <selectKey keyColumn="id" keyProperty="id" resultType="int">
   select last_insert_id();
  </selectKey>
  insert into user(username,birthday,sex,address)
values(#{username},#{birthday},#{sex},#{address})
 </insert>
```

3. 用户更新

- 在持久层接口中添加更新方法

```
/**
 * 更新用户
  * @param user
  * @return 影响数据库记录的行数
  */
int updateUser(User user);
```

- 在用户的映射配置文件中配置

```
<!-- 更新用户 -->
 <update id="updateUser" parameterType="club.banyuan.entity.User">
  update user set username=#{username},birthday=#{birthday},sex=#{sex},
address=#{address} where id=#{id}
 </update>
```

- 加入更新的测试方法

```
@Test
 public void testUpdateUser()throws Exception{
  //1.根据 id 查询
  User user = userDao.findById(52);
 //2.更新操作
  user.setAddress("北京市顺义区");
  int res = userDao.updateUser(user);
  System.out.println(res);
}
```

4. 用户删除

- 在持久层接口中添加删除方法

```
/**
 * 根据 id 删除用户
  * @param userId
  * @return
```

```
   */
int deleteUser(Integer userId);
```

- 在用户的映射配置文件中配置

```
<!-- 删除用户 -->
<delete id="deleteUser" parameterType="java.lang.Integer">
   delete from user where id = #{uid}
</delete>
```

- 加入删除的测试方法

```
@Test
public void testDeleteUser() throws Exception {
   //6.执行操作
   int res = userDao.deleteUser(52);
   System.out.println(res);
 }
```

5. 用户模糊查询

① 第一种配置方式

- 在持久层接口中添加模糊查询方法

```
List<User> findByName(String username);
```

- 在用户的映射配置文件中配置

```
<!-- 根据名称模糊查询 -->
 <select id="findByName" resultType="club.banyuan.entity.User"
parameterType="String">
       select * from user where username like #{username}
</select>
```

- 加入模糊查询的测试方法

```
 @Test
       public void testFindByName(){
           //5.执行查询一个方法
            List<User> users = userDao.findByName("%王%");
            for(User user : users){
```

```
            System.out.println(user);
        }
    }
```

② 另一种配置方式

第一步：修改 SQL 语句的配置，配置如下：

```
 <!-- 根据名称模糊查询 -->
<select id="findByName" parameterType="string" resultType="com.itheima.domain.User">
        select * from user where username like '%${value}%'
</select>
```

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：测试，如下：

```
/**
  * 测试模糊查询操作
   */
 @Test
 public void testFindByName(){
     //5.执行查询一个方法
  List<User> users = userDao.findByName("王");
  for(User user : users){
  System.out.println(user);
  }
 }
```

③ #{}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。

6. 使用聚合函数

- 在持久层接口中添加模糊查询方法

```
/**
 * 查询总记录条数
  * @return
  */
int findTotal();
```

- 在用户的映射配置文件中配置

```
<!-- 查询总记录条数 -->
<select id="findTotal" resultType="int">
  select count(*) from user;
</select>
```

- 加入聚合查询的测试方法

```
@Test public void testFindTotal() throws Exception {
  //6.执行操作
  int res = userDao.findTotal();
  System.out.println(res);
 }
```

四、 Mybatis 的参数深入

parameterType 配置参数

1. MyBatis 内置类型别名

基本类型和 String 我们可以直接写类型名称,也可以使用包名.类名的方式,例如:

java.lang.String。

实体类类型，目前我们只能使用全限定类名。

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

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

2. 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。

- 编写 QueryVo

```
public class QueryVo implements Serializable {

 private User user;

 public User getUser() {
    return user;
  }

 public void setUser(User user) {
  this.user = user;
  }
}
```

- 编写持久层接口

```
public interface IUserDao {      /**
  * 根据 QueryVo 中的条件查询用户     * @param vo     * @return     */   List<User>
findByVo(QueryVo vo); }
```

- 持久层接口的映射文件

```
<!-- 根据用户名称模糊查询,参数变成一个 QueryVo 对象了 --> <select id="findByVo"
resultType="com.itheima.domain.User"
parameterType="com.itheima.domain.QueryVo">      select * from user where
username like #{user.username}; </select>
```

- 测试包装类作为参数

```
@Test
public void testFindByQueryVo() {
  QueryVo vo = new QueryVo();
  User user = new User();
  user.setUserName("%王%");
```

```
    vo.setUser(user);
    List<User> users = userDao.findByVo(vo);
    for(User u : users) {
      System.out.println(u);
    }
  }
```

五、 Mybatis 的输出结果封装

1. resultType 配置结果类型

resultType 属性可以指定结果集的类型，它支持基本类型和实体类类型。和 parameterType 一样，如果注册过类型别名的，可以直接使用别名。没有注册过的必须 使用全限定类名。例如：我们的实体类此时必须是全限定类名。 同时，当是实体类名称是，还有一个要求，实体类中的属性名称必须和查询语句中的列名保持一致，否则无法 实现封装。

① 基本类型示例

- Dao 接口

```
/**
 * 查询总记录条数
 * @return
  */
int findTotal();
```

- 映射配置

```
<!-- 查询总记录条数 -->
<select id="findTotal" resultType="int">
  select count(*) from user;
</select>
```

② 实体类类型示例

- Dao 接口

```
/**
 * 查询所有用户
  * @return
 */
List<User> findAll();
```

- 映射配置

```
<!-- 配置查询所有操作 -->
 <select id="findAll" resultType="club.banyuan.entity.User">
  select * from user
 </select>
```

- 修改实体类

的实体类属性和数据库表的列名已经不一致

```
public class User implements Serializable {
 private Integer userId;
  private String userName;
  private Date userBirthday;
  private String userSex;
  private String userAddress;

 public Integer getUserId() {
    return userId;
  }
  public void setUserId(Integer userId) {
    this.userId = userId;
  }
 public String getUserName() {
    return userName;
  }
  public void setUserName(String userName) {
    this.userName = userName;
  }
```

```
 public Date getUserBirthday() {
    return userBirthday;
  }
  public void setUserBirthday(Date userBirthday) {
 this.userBirthday = userBirthday;
  }
  public String getUserSex() {
   return userSex;
  }
  public void setUserSex(String userSex) {
   this.userSex = userSex;
  }
  public String getUserAddress() {
   return userAddress;
  }
  public void setUserAddress(String userAddress) {
   this.userAddress = userAddress;
  }
  @Override
  public String toString() {
   return "User [userId=" + userId + ", userName=" + userName + ", userBirthday="
+ userBirthday + ", userSex="       + userSex + ", userAddress=" + userAddress +
"]";
 }
   }
```

- Dao 接口

```
/**
 * 查询所有用户
  * @return
 */
List<User> findAll();
```

- 映射配置

```
<!-- 配置查询所有操作 -->
 <select id="findAll" resultType="club.banyuan.entity.User">
```

```
  select * from user
</select>
```

- 测试查询结果

```
@Test public void testFindAll() {
List<User> users = userDao.findAll();
  for(User user : users) {
    System.out.println(user);
  }
 }
```

- 修改映射配置

使用别名查询

```
<!-- 配置查询所有操作 -->
 <select id="findAll" resultType="club.banyuan.entity.User">
  select id as userId,username as userName,birthday as userBirthday,  sex as userSex,address as userAddress from user
 </select>
```

2. resultMap 结果类型

resultMap 标签可以建立查询的列名和实体类的属性名称不一致时建立对应关系。从而实现封装。

在 select 标签中使用 resultMap 属性指定引用即可。同时 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。

- 定义 resultMap

```
<!-- 建立 User 实体和数据库表的对应关系
type 属性：指定实体类的全限定类名
  id 属性：给定一个唯一标识，是给查询 select 标签引用用的。
 -->
<resultMap type="club.banyuan.entity.User" id="userMap">
  <id column="id" property="userId"/>
  <result column="username" property="userName"/>
  <result column="sex" property="userSex"/>
  <result column="address" property="userAddress"/>
```

```
  <result column="birthday" property="userBirthday"/>
 </resultMap>
```

id 标签：用于指定主键字段

result 标签：用于指定非主键字段

column 属性：用于指定数据库列名

property 属性：用于指定实体类属性名称

- 映射配置

```
<!-- 配置查询所有操作 -->
 <select id="findAll" resultMap="userMap">
  select * from user
 </select>
```

- 测试结果

```
@Test
  public void testFindAll() {
  List<User> users = userDao.findAll();
   for(User user : users) {
   System.out.println(user);
  }
 }
```

六、 SqlMapConfig.xml 配置文件

1. properties（属性）

在使用 properties 标签配置时，可以采用两种方式指定属性配置。

① 第一种

```
<properties>
 <property name="jdbc.driver" value="com.mysql.jdbc.Driver"/>
  <property name="jdbc.url" value="jdbc:mysql://localhost:3306/eesy"/>
 <property name="jdbc.username" value="root"/>
  <property name="jdbc.password" value="1234"/>
```

```
</properties>
```

② 第二种

在 classpath 下定义 db.properties 文件

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/eesy
jdbc.username=root
jdbc.password=1234
```

properties 标签配置

```
<!-- 配置连接数据库的信息
    resource 属性：用于指定 properties 配置文件的位置，要求配置文件必须在类路径下
     resource="jdbcConfig.properties"
  url 属性：
    URL： Uniform Resource Locator 统一资源定位符
     http://localhost:8080/mystroe/CategoryServlet  URL
    协议  主机   端口   URI
    URI：Uniform Resource Identifier 统一资源标识符
     /mystroe/CategoryServlet
    它是可以在 web 应用中唯一定位一个资源的路径
 -->
 <properties url=
file:///D:/IdeaProjects/day02_eesy_01mybatisCRUD/src/main/resources/jdbcConfig.prop erties">
 </properties>
```

dataSource 标签就变成了这样配置

```
 <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>
```

2. typeAliases（类型别名）

在前面我们讲的 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。

自定义别名：

在 SqlMapConfig.xml 中配置：

```
<typeAliases>
  <!-- 单个别名定义 -->
  <typeAlias alias="user" type="club.banyuan.entity.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="club.banyuan.entity"/>
  <package name=" 其它包 "/>
 </typeAliases>
```

3. mappers（映射器）

① <mapper resource=" " />

使用相对于类路径的资源 如：<mapper resource="club/banyuan/dao/UserDao.xml" />

② <mapper class=" " />

使用 mapper 接口类路径 如：<mapper class="club.banyuan.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。

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。

4. Mybatis 的连接池技术

① Mybatis 连接池的分类

UNPOOLED　　不使用连接池的数据源
POOLED　　使用连接池的数据源
JNDI　　使用 JNDI 实现的数据源

② Mybatis 中数据源的配置

```
<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>
```