北京动力节点教育科技有限公司

动力节点课程讲义

DONGLIJIEDIANKECHENGJIANGYI

www.bjpowernode.com

MyBatis 框架

第1章 框架概述

1.1 软件开发常用结构

1.1.1 三层架构

三层架构包含的三层：

界面层（User Interface layer）业务逻辑层（Business Logic Layer）数据访问层（Data access layer）

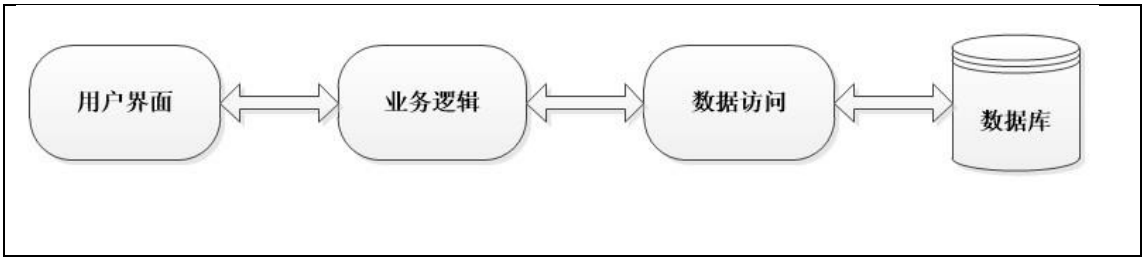
三层的职责

1. 界面层（表示层，视图层）:主要功能是接受用户的数据，显示请求的处理结果。使用 web 页面和用户交互，手机 app 也就是表示层的，用户在 app 中操作，业务逻辑在服务器端处理。
2. 业务逻辑层：接收表示传递过来的数据，检查数据，计算业务逻辑，调用数据访问层获取数据。
3. 数据访问层：与数据库打交道。主要实现对数据的增、删、改、查。将存储在数据库中的数据提交给业务层，同时将业务层处理的数据保存到数据库.

三层的处理请求的交互：

客户端<--->界面层<--->业务逻辑层<--->数据访问层<--->数据库。

如图



为什么要使用三层？

1. 结构清晰、耦合度低， 各层分工明确

2. 可维护性高，可扩展性高

3. 有利于标准化

4. 开发人员可以只关注整个结构中的其中某一层的功能实现

5. 有利于各层逻辑的复用

1.1.2 常用框架

常见的 J2EE 中开发框架

MyBatis 框架：

MyBatis 是一个优秀的基于 java 的持久层框架，内部封装了 jdbc，开发者只需要关注 sql 语句本身，而不需要处理加载驱动、创建连接、创建 statement、关闭连接，资源等繁杂的过程。

MyBatis 通过 xml 或注解两种方式将要执行的各种 sql 语句配置起来，并通过 java 对象和 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。

Spring 框架：

Spring 框架为了解决软件开发的复杂性而创建的。Spring 使用的是基本的 JavaBean 来完成以前非常复杂的企业级开发。Spring 解决了业务对象，功能模块之间的耦合，不仅在 javase,web 中使用，大部分 Java 应用都可以从 Spring 中受益。

Spring 是一个轻量级控制反转(IoC)和面向切面(AOP)的容器。

SpringMVC 框架

Spring MVC 属于 SpringFrameWork 3.0 版本加入的一个模块，为 Spring 框架提供了构建 Web 应用程序的能力。现在可以 Spring 框架提供的 SpringMVC 模块实现 web 应用开发，在 web 项目中可以无缝使用 Spring 和Spring MVC 框架。

1.2 框架是什么

1.2.1 框架定义

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;另一种认为，框架是可被应用开发者定制的应用骨架、模板。

简单的说，框架其实是半成品软件，就是一组组件，供你使用完成你自己的系统。从另一个角度来说框架一个舞台，你在舞台上做表演。在框架基础上加入你要完成的功能。

框架安全的，可复用的，不断升级的软件。

1.2.2 框架解决的问题

框架要解决的最重要的一个问题是技术整合，在 J2EE 的 框架中，有着各种各样的技术，不同的应用，系统使用不同的技术解决问题。需要从 J2EE 中选择不同的技术，而技术自身的复杂性，有导致更大的风险。企业在开发软件项目时，主要目的是解决业务问题。 即要求企业负责技术本身，又要求解决业务问题。这是大多数企业不能完成的。框架把相关的技术融合在一起，企业开发可以集中在业务领域方面。

另一个方面可以提供开发的效率。

1.3 JDBC 编程

1.3.1 使用 JDBC 编程的回顾

```
public void findStudent() {
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
  //注册 mysql驱动
  Class.forName("com.mysql.jdbc.Driver");
  //连接数据的基本信息 url , username, password
  String url = "jdbc:mysql://localhost:3306/springdb";
  String username = "root";
  String password = "123456";
  //创建连接对象
  conn = DriverManager.getConnection(url, username, password);
  //保存查询结果
```

```
List<Student> stuList = new ArrayList<>();
//创建Statement，用来执行 sql语句
stmt = conn.createStatement();
//执行查询，创建记录集，
rs = stmt.executeQuery("select * from student");
while (rs.next()) {
     Student stu = new Student();
     stu.setId(rs.getInt("id"));
     stu.setName(rs.getString("name"));
     stu.setAge(rs.getInt("age"));
     //从数据库取出数据转为 Student对象，封装到 List集合
     stuList.add(stu);}
}catch(Exception e){
    e.printStackTrace();
}finally{
   try{
    if(rs != null)
      rs.lose();
    if(pstm != null)
      pstm.close();
   if(con != null)
      con.close();

}catch(Exception e){
  e.printStackTrace();
}
}
```

1.3.2 使用 JDBC 的缺陷

1. 代码比较多，开发效率低
2. 需要关注 Connection ,Statement, ResultSet 对象创建和销毁
3. 对ResultSet 查询的结果，需要自己封装为 List
4. 重复的代码比较多些
5. 业务代码和数据库的操作混在一起

1.4 MyBatis 框架概述

MyBatis 框架：

MyBatis 本是 apache 的一个开源项目 iBatis, 2010 年这个项目由 apache software foundation 迁移到了 google code，并且改名为 MyBatis 。2013 年11 月迁移到 Github。

iBATIS 一词来源于“internet”和“abatis”的组合，是一个基于 Java 的持久层框架。iBATIS 提供的持久层框架包括 SQL Maps 和 Data Access Objects（DAOs）

当前，最新版本是 MyBatis 3.5.7 ，其发布时间是 2021 年 4月 7日。

1.4.1 MyBatis 框架解决的主要问题

减轻使用 JDBC 的复杂性，不用编写重复的创建 Connetion , Statement ； 不用编写关闭资源代码。直接使用 java 对象，表示结果数据。**让开发者专注 SQL 的处理**。 其他分心的工作由 MyBatis 代劳。

MyBatis 可以完成：

注册数据库的驱动，例如 Class.forName(“com.mysql.jdbc.Driver”))

创建 JDBC 中必须使用的 Connection ， Statement， ResultSet 对象

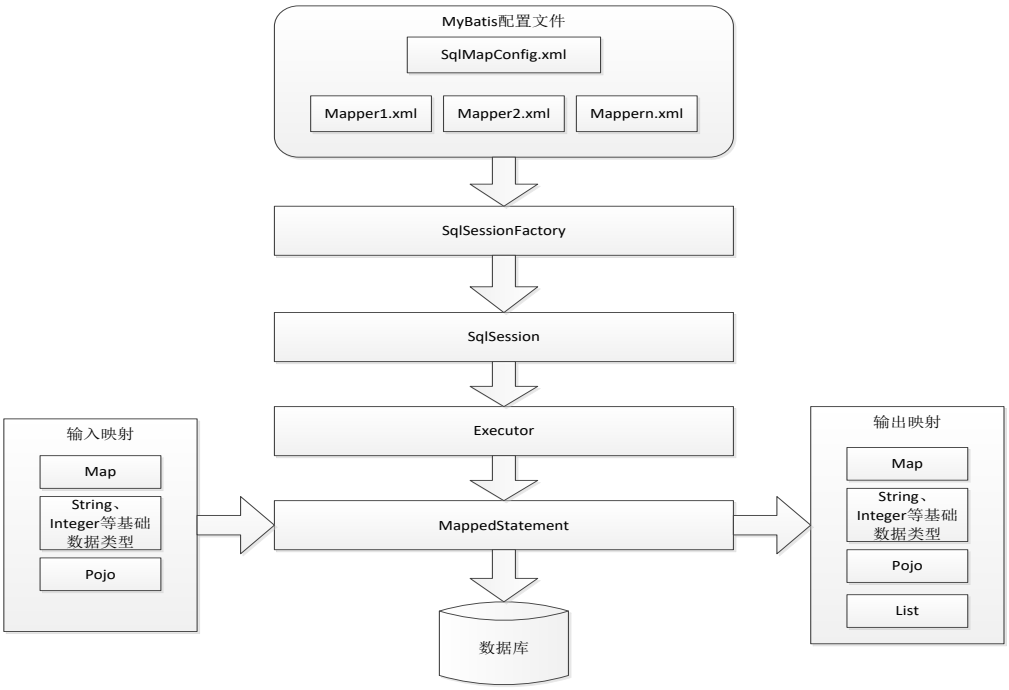
从xml 中获取 sql，并执行 sql 语句，把 ResultSet 结果转换 java 对象

```
List<Student> list = new ArrayLsit<>();
ResultSet rs = state.executeQuery(“select * from student”);
while(rs.next){
Student student = new Student();
student.setName(rs.getString(“name”)); student.setAge(rs.getInt(“age”));
list.add(student);
}
```

4.关闭资源

ResultSet.close() , Statement.close() , Conenection.close()

1.4.2 MyBatis 框架的结构



1、 mybatis配置

SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

2、 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂

3、 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。

4、 mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。

5、 Mapped Statement也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。

6、 Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。

7、 Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

第2章 MyBatis 框架快速入门

2.1 内容列表

快速开始一个 MyBatis

基本 CURD 的操作

MyBatis 内部对象分析

使用 DaoImpl.xml

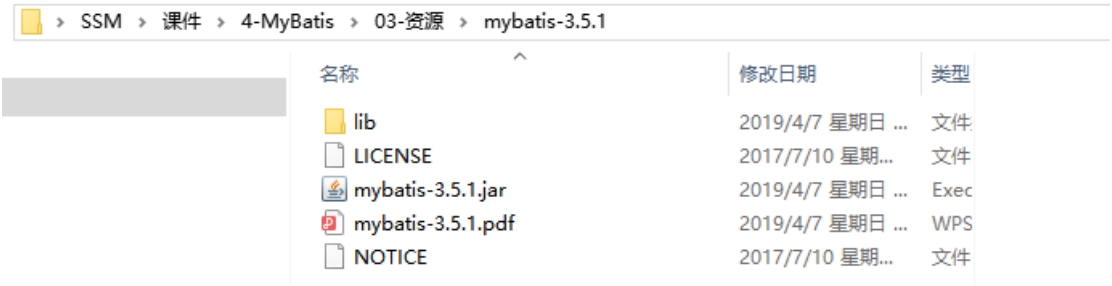
2.2 入门案例

2.2.1 开发准备

搭建 MyBatis 开发环境，实现第一个案例
下载 mybatis

https://github.com/mybatis/mybatis-3/releases

下载后的目录结构



其中：

mybatis-3.5.1.jar----->mybatis 的核心包

lib ----->mybatis 的依赖包

mybatis-3.5.1.pdf----->mybatis 使用手册

2.2.2 搭建 MyBatis 开发环境

（1） 创建 mysql 数据库和表

```
CREATE DATABASE ssm DEFAULT CHARSET utf8;
```

```
use ssm;

CREATE TABLE `student` (
`id` int(11)  AUTO_INCREMENT primary key ,
`name` varchar(255) DEFAULT NULL,
`email` varchar(255) DEFAULT NULL,
`age` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert into student(name,email,age) values('张三','zhangsan@126.com',22);

insert into student(name,email,age) values('李四','lisi@126.com',21);

insert into student(name,email,age) values('王五','wangwu@163.com',22);

insert into student(name,email,age) values('赵六','zhaoliun@qq.com',24);

select * from student;
```

（2） 创建 工程，添加依赖

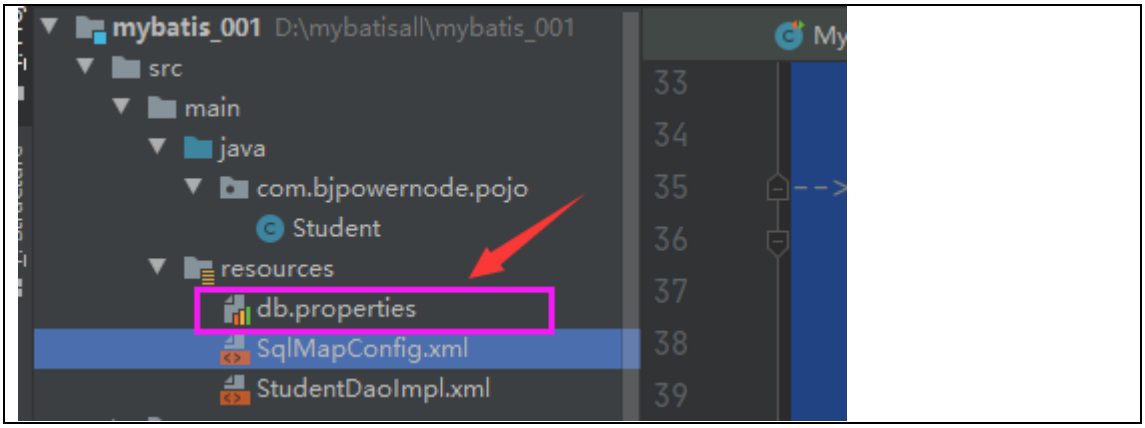
```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
<!--    添加 mybatis 框架的依赖-->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.1</version>
    </dependency>
<!--    添加 mysql8.0 驱动-->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.22</version>
    </dependency>
```

（3） 编写 Student 实体类

```
public class Student {

    private int id;

    private String name;

    private String email;

    private int age;

}
```

（4） 添加 db.properties 文件

该文件用来提供数据库连接信息.



（5） 创建 MyBatis 主配置文件

右键新建 SqlMapConfig.xml 文件,从 MyBatis-3-User-Guide-Simplified-Chinese.pdf 中拷贝过来头参数.

```
<?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">
```

```
<!--mybatis-3-config.dtd:它规定了当前的 xml 文件中可以出现哪些标签
                        这些标签的顺序和位置,这些标签的属性
                        这些标签的子标签
```

（6） 创建 MyBatis 主配置文件参数详解

```
<configuration>
<!--    将文件流指向 db.properties,最终是为了从中读取数据库的一堆配置信息-->
    <properties resource="db.properties"></properties>
<!--    要进行数据库的访问，所以要进行数据库访问的配置，驱动，
url,username,password-->
<!--    environments:进行环境变量(连接数据库)的配置
                    可以进行多个数据库连接配置,可以上线一套,开发一套,可以
mysql 一套,oracle 一套等
        default:本次配置中使用的环境变量的名称,多套配置,default 决定哪套配置生效
-->
    <environments default="development">
<!--environment:  进行具体环境变量的配置
    id:  为当前的配置的环境变量起个名称,为了在 environments 的 default 中使用
-->
        <environment id="development">
<!--            transactionManager:事务管理
                type: JDBC:就是程序员自己来管理事务的提交和回滚
                      MANAGED:由容器来进行事务的管理,例如:spring
-->
            <transactionManager type="JDBC"></transactionManager>
<!--            dataSource:数据源的配置
                    type:指定数据源的配置方式,是否是连接池
                        "POOLED":表明使用数据库连接池
                        "UNPOOLED":不使用连接池
                        "JNDI":java 命名目录接口,由服务器端负责连接池的管理
                    property: driver:数据库驱动
                              url:数据库的路径
                              username:访问数据库的用户名
                              password:访问数据库的用户的密码
-->
            <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>
<!--    注册 StudentDaoImpl.xml 文件,注意.xml 后缀要带上-->
<mappers>
    <mapper resource="StudentDaoImpl.xml"></mapper>
</mappers>
</configuration>
```

支持中文的 url,连接mysql8要添安全密钥允许访问和时区的设置

jdbc.url=jdbc:mysql://localhost:3308/ssm?useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true

（7） 创建 StudentDaoImpl.xml 文件

右键新建 StudentDaoImpl.xml 文件,从 MyBatis-3-User-Guide-Simplified-Chinese.pdf 中拷贝过来头参数.

该文件完成数据库中 student 表的所有增删改查的操作.

<mapper namespace="自定义的路径名称">,在简单访问中 namespace 中的内容可以自定义,目的是为了区别不同<mapper>中相同 id 的语句.

该文件中提供<select><update><insert><delete>等数据库中的基本操作标签.

```
<?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">
```

```xml
<mapper namespace="com.bjpowernode">
    <!--完成数据库中的所有增删改查操作-->
    <select id="getAllStudent" resultType="com.bjpowernode.pojo.Student">
        select id,name,email,age from student
    </select>
    <!--按学生ID查-->
    <select id="getById" parameterType="int" resultType="com.bjpowernode.po
        select id,name,email,age from student where id=#{id}
    </select>
    <!--按学生姓名查-->
    <select id="getByName" parameterType="string" resultType="com.bjpowerno
        select id,name,email,age from student where name like '%${value}%'
    </select>
    <!--增加学生-->
    <insert id="insert" parameterType="com.bjpowernode.pojo.Student">
        insert into student(name,email,age) values(#{name},#{email},#{age})
    </insert>
    <!--更新学生-->
    <update id="update" parameterType="com.bjpowernode.pojo.Student">
        update student set age = #{age} where id = #{id}
    </update>

    <!--按指定ID删除学生-->
    <delete id="delete" parameterType="int">
        delete from student where id=#{id}
    </delete>

</mapper>
```

（8） 创建测试类

使用 Junit 单元测试完成各种功能测试.

使用@Before 注解来进行所有测试前的 SqlSession 的创建工作.

使用@After 注解来进行所有测试方法执行后的关闭 SqlSession 的工作.

使用@Test 注解来验证每一个功能的实现

```
public class MyTest {

    SqlSession session;
```

```
@Before
    public void openSession()throws Exception{
        //创建文件流,读取 SqlMapConfig.xml
        InputStream inputStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
        //创建 SqlSessionFactory
        SqlSessionFactory factory = new
SqlSessionFactoryBuilder().build(inputStream);
        //取得 SqlSession
        session = factory.openSession();
    }
    @Test
    public void testSelectStudentAll()throws Exception{
        List<Student> list =
session.selectList("com.bjpowernode.selectStudentAll");
        list.forEach(stu-> System.out.println(stu));
    }
    @Test
    public void testStudentGetById(){
        Student stu =
session.selectOne("com.bjpowernode.selectStudentById",2);
        System.out.println(stu);
    }
    @Test
    public void testSelectStudentByEmail(){
        List<Student> list =
session.selectList("com.bjpowernode.selectStudentByEmail","l");
        list.forEach(stu-> System.out.println(stu));
    }
    @Test
    public void testInsertStudent(){
        Student stu = new Student("李四四","23234@qq.com",22);
        int num = session.insert("com.bjpowernode.insertStudent",stu);
        //切记切记切记:因为我们的事务管理机制使用的是 JDBC,所以所有的增删改查
后,要手工提交事务
        //session.commit();
        session.commit();//手工提交事务
        System.out.println(num);
    }
    @After
```

```
    public void closeSession(){
        session.close();
    }
}
```

2.2.3 MyBatis 对象分析

(1) Resources 类

Resources 类，顾名思义就是资源，用于读取资源文件。其有很多方法通过加载并解析资源文件，返回不同类型的 IO 流对象。

(2) SqlSessionFactoryBuilder 类

SqlSessionFactory 的创建，需要使用SqlSessionFactoryBuilder 对象的 build() 方法。由于SqlSessionFactoryBuilder 对象在创建完工厂对象后，就完成了其历史使命，即可被销毁。所以，一般会将该对象创建为一个方法内的局部对象，方法结束，对象销毁。

(3) SqlSessionFactory 接口

SqlSessionFactory 接口对象是一个重量级对象（系统开销大的对象），是线程安全的，所以一个应用只需要一个该对象即可。创建 SqlSession 需要使用 SqlSessionFactory 接口的的 openSession()方法。

A. openSession(true)：创建一个有自动提交功能的 SqlSession

B. openSession(false)：创建一个非自动提交功能的 SqlSession，需手动提交

C. openSession()：同 openSession(false)

(4)SqlSession 接口

SqlSession 接口对象用于执行持久化操作。一个 SqlSession 对应着一次数据库会话，一次会话以 SqlSession 对象的创建开始，以 SqlSession 对象的关闭结束。

SqlSession 接口对象是线程不安全的，所以每次数据库会话结束前，需要马上调用其 close()方法，将其关闭。再次需要会话，再次创建。 SqlSession 在方法内部创建，使用完毕后关闭。

2.3 SqlMapConfig.xml 文件的优化

2.3.1 添加日志打印输出

文件加入日志配置，可以在控制台输出执行的 sql 语句和参数.

```
<!--      设置查看 mybatis 生成的 sql 语句的日志配置-->
    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>
```

2.3.2 为实体类起别名

由于 XXXmapper.xml 文件中入参和返回值都要使用实体类的对象,而我们在使用的时候必须书写实体类的完全限定类名,这样比较麻烦,我们可以通过起别名的方式来简化此操作.起别名的方式有两种.

(1) 为单个实体类起别名

```
<typeAlias type="com.bjpowernode.pojo.Users"
alias="users"></typeAlias>
```

type:实体类的完全限定名称

alias：为实体类起的别名,在以后所有使用实体类类型的地方,写此别名即可.

(2) 批量别名注册

可以通过<package>来批量的为实体类起别名,只要指定实体类所在的包的名称即可.MyBatis 框架会自动为每个实体类起别名为类型的全小写或类名的大小写混合.推荐使用类名的全小写.

```
<typeAliases>
    <package name="com.bjpowernode.pojo"/>
</typeAliases>
```

2.3.3 在 SqlMapConfig.xml 文件中注册 XXXMapper.xml

注册 XXXMapper.xml 文件的方式有四种.

(1)使用 resource 注册

```
<mapper
resource="com/bjpowernode/mapper/UsersMapper.xml"></mapper>
```

注意: UserMapper.xml 是带后缀的,分隔符使用/.

(2)使用 class 注册

动态代理的方式,使用此种注册.

```
<mapper class="com.bjpowernode.mapper.UsersMapper"></mapper>
```

注意: class 的值是接口的完全限定名称.

(3)使用 url 注册

```
<mapper url="file:///E:/UserMapper.xml"></mapper>
```

指定绝对路径注册,注意 file 后面是双杠.

(4)使用<package>注册

```
<package name="com.bjpowernode.mapper"/>
```

第3章 动态代理

3.1动态代理开发规范

MyBatis 框架使用动态代理的方式来进行数据库的访问.

Mapper 接口的开发相当于是过去的 Dao 接口的开发。由 MyBatis 框架根据接口定义创建动态代理对象，代理对象的方法体同 Dao 接口实现类的方法。在设计时要遵守以下规范.

1. Mapper 接口与 Mapper.xml 文件在同一个目录下
2. Mapper 接口的完全限定名与 Mapper.xml 文件中的 namespace 的值相同。
3. Mapper 接口方法名称与 Mapper.xml 中的标签的 statement 的 ID 完全相同。
4. Mapper 接口方法的输入参数类型与 Mapper.xml 的每个 sql 的 parameterType 的类型相同
5. Mapper 接口方法的输出参数与 Mapper.xml 的每个 sql 的 resultType 的类型相同。
6. Mapper 文件中的 namespace 的值是接口的完全限定名称.
7. 在 SqlMapConfig.xml 文件中注册时,使用 class 属性=接口的完全限定名.

3.2开发步骤

1. 新建项目添加依赖
2. 新建属性文件 db.properties
3. 新建环境配置文件(SqlMapConfig.xml)

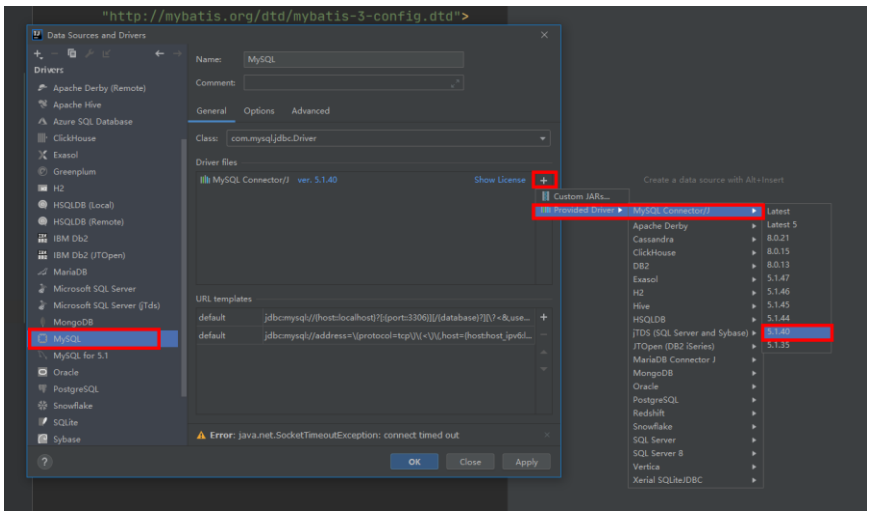
```xml
<configuration>
    <properties resource="db.properties" ></properties>
    <typeAliases>
        <!--注册别名-->
        <typeAlias type="com.oracle.demo.pojo.Users" alias="users"></typeAlias>
    </typeAliases>
    <environments default="development"><!--开发环境-->
        <environment id="development">
            <transactionManager type="JDBC"></transactionManager>
            <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>
    <mappers>
        <mapper class="com.oracle.demo.mapper.UsersMapper"></mapper>
    </mappers>
</configuration>
```

注册实体类的别名，简化开发。

注意：在注册 mapper.xml 文件时，使用 class=”接口的完全限定名称”.

4. 新建可视化窗口

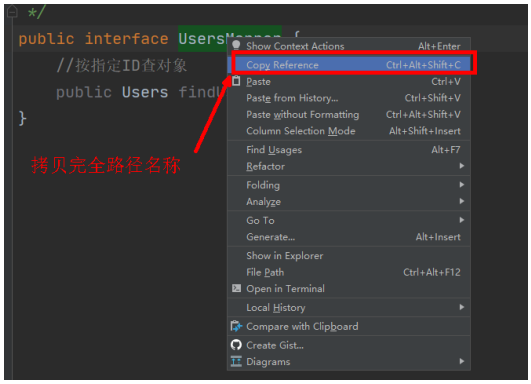
可以使用系统提供的 jar，也可以使用自己的 jar 包

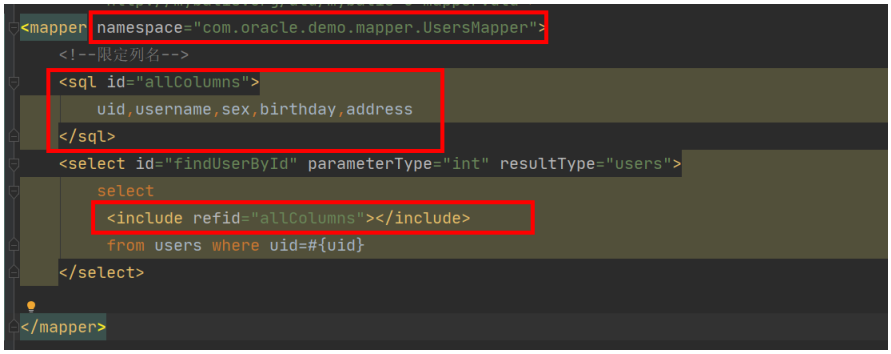


5. 新建实体类
6. 新建接口

```java
public interface UsersMapper {
    //按指定ID查对象
    public Users findUserById(int id);
}
```

7. 新建（接口的实现类）与接口同名的 xml 文件，完成数据库中的所有操作





8. 添加 JUnit 的 jar 包
9. 新建测试类

```
SqlSession session;
@Test
public void testFindUsersById()throws Exception {

    InputStream inputStream= Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactory factory=new SqlSessionFactoryBuilder().build(inputStream);
    session=factory.openSession();

    UsersMapper mapper=session.getMapper(UsersMapper.class);
    Users u=mapper.findUserById(1);
    System.out.println(u);
    session.close();
}
```

总结：

UsersMapper.java 和 UsersMapper.xml 文件必须在同一个目录下，且必须同名。

在 UsersMapper.xml 文件中添加 namespace 属性为接口的完全路径名。

10. 优化测试

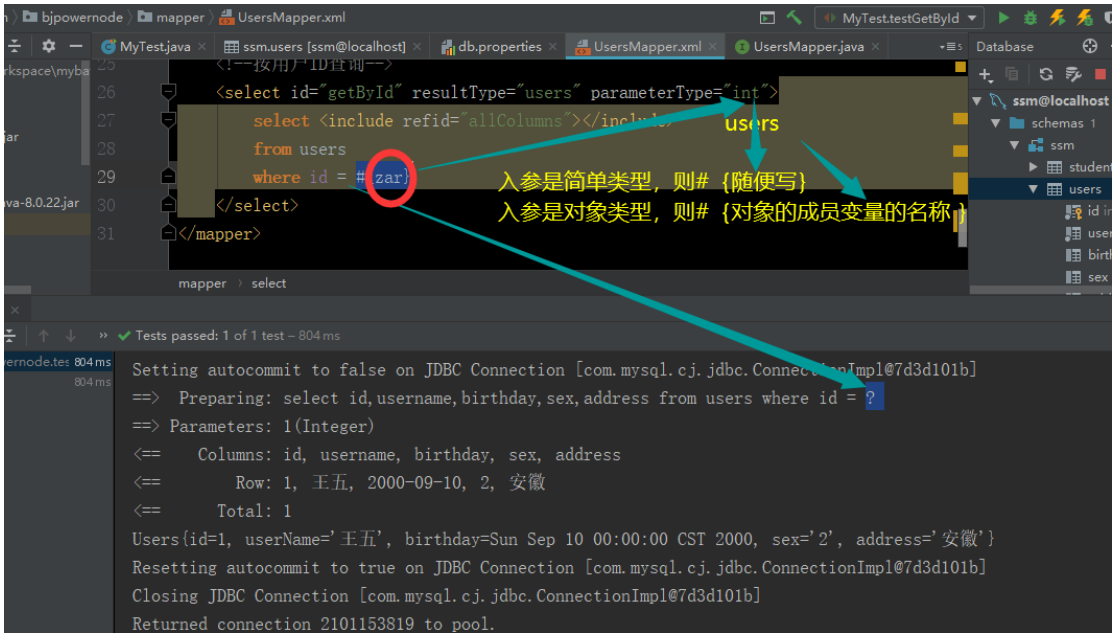
```
SqlSession session;
UsersMapper mapper;
//所有方法每次调用之前都会调用一次@Before注解的方法
@Before
public void openSession() throws Exception {
    InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);
    session = factory.openSession();
    mapper = session.getMapper(UsersMapper.class);
}
@Test
public void testfindUsersById() throws Exception {
    // this.openSession();
    Users u = mapper.findUsersById(1);
    System.out.println(u);
}
//所有方法每次调用之后都会调用一次 @After注解的方法
@After
public void testCloseSession() { session.close(); }
```

3.3 #{}和${}

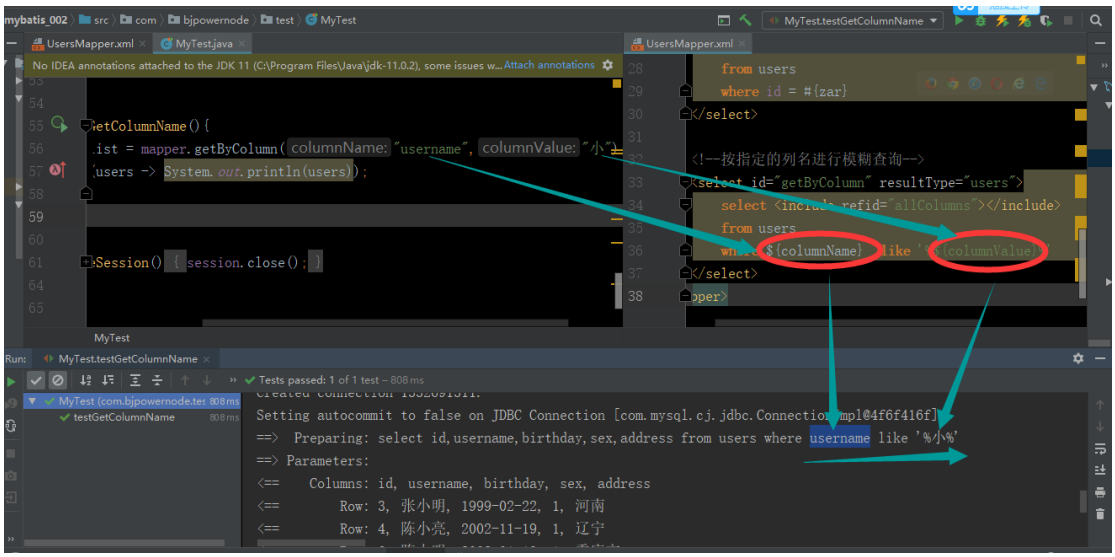
#{}是对非字符串拼接的参数的占位符，如果入参是简单数据类型，#{}里可以任意写，但是如果入参是对象类型，则#{}里必须是对象的成员变量的名称，#{}可以有效防止 sql 注入。

**${}主要是针对字符串拼接替换**，如果入参是基本数据类型，${}里必须是 value,但是如果入参是对象类型，则${}里必须是对象的成员变量的名称。${}还可以替换列名和表名，存在 sql 注入风险，尽量少用。

{ } 演示



${}演示



3.4返回主键标签

在完成插入操作后，将生成的主键信息通过实体类对象返回，在进行后继关联插入操作时，不用再次访问数据库。

```
<insert id="insertUser" parameterType="com.oracle.demo.pojo.Users">
        <!--
            Order:指定生成返回主键的时机，AFTER：先插入再返回主键
                                          BEFORE：先生成再完成插入
            keyProperty：生成的主键放入到对象的哪个属性中
```

```
            resultType: 返回的主键的类型
        -->
          <selectKey order="AFTER" keyProperty="uid" resultType="int">
                select LAST_INSERT_ID()
          </selectKey>
          <!--  先生成随机字符串，再返回
          <selectKey order="BEFORE" keyProperty="uid" resultType="string">
                select uuid()
          </selectKey>
          -->
```

第4章 动态 sql

动态 SQL 是 MyBatis 的强大特性之一。如果你使用过 JDBC 或其它类似的框架，你应该能理解根据不同条件拼接 SQL 语句有多痛苦，例如拼接时要确保不能忘记添加必要的空格，还要注意去掉列表最后一个列名的逗号。利用动态 SQL，可以彻底摆脱这种痛苦。使用动态 SQL 并非一件易事，但借助可用于任何 SQL 映射语句中的强大的动态 SQL 语言，MyBatis 显著地提升了这一特性的易用性。

4.1 <sql>标签

当多种类型的查询语句的查询字段或者查询条件相同时，可以将其定义为常量，方便调用。

```
<!--定义列名-->
<sql id="columns">
    id,username,birthday,sex,address
</sql>
<!--定义条件-->
<sql id="Example_Where_Clause" >
  <where >
    <foreach collection="oredCriteria" item="criteria" separator="or" >
      <if test="criteria.valid" >
        <trim prefix="(" suffix=")" prefixOverrides="and" >
          <foreach collection="criteria.criteria" item="criterion" >
            <choose >
              <when test="criterion.noValue" >
                and ${criterion.condition}
```

```
            </when>
            <when test="criterion.singleValue" >
              and ${criterion.condition} #{criterion.value}
            </when>
            <when test="criterion.betweenValue" >
              and ${criterion.condition} #{criterion.value} and
#{criterion.secondValue}
            </when>
            <when test="criterion.listValue" >
              and ${criterion.condition}
              <foreach collection="criterion.value" item="listItem" open="("
close=")" separator="," >
                #{listItem}
              </foreach>
            </when>
          </choose>
        </foreach>
      </trim>
    </if>
  </foreach>
</where>
</sql>
```

4.2 <include>标签

用于引用定义的常量。

```
<!--引用定义的列名-->
select <include refid="columns"></include>
from users
<!--引用定义的条件-->
<select id="selectByExample" resultMap="BaseResultMap"
parameterType="com.bjpowernode.pojo.ProductInfoExample" >
  select
  <if test="distinct" >
    distinct
  </if>
  <include refid="Base_Column_List" />
  from product_info
  <if test="_parameter != null" >
    <include refid="Example_Where_Clause" />
  </if>
```

```
  <if test="orderByClause != null" >
    order by ${orderByClause}
  </if>
</select>
```

4.3 <if>标签

进行条件判断。

```
<if test="userName != null and userName != '' ">
    and username like concat('%',#{userName},'%')
</if>
```

4.4 <where>标签

一般开发复杂业务的查询条件时，如果有多个查询条件，通常会使用<where>标签来进行控制。标签可以自动的将第一个条件前面的逻辑运算符 (or ,and) 去掉，正如代码中写的，id 查询条件前面是有“and”关键字的，但是在打印出来的 SQL 中却没有，这就是<where> 的作用。

```
<select id="getByCondition" resultType="users" parameterType="users">
    select <include refid="allColumns"></include>
    from users
    <where>
        <if test="userName != null and userName != '' ">
            and username like concat('%',#{userName},'%')
        </if>
        <if test="birthday != null ">
            and birthday = #{birthday}
        </if>
        <if test="sex != null and sex !=''">
            and sex = #{sex}
        </if>
        <if test="address != null and address !=''">
            and address like concat('%',#{address},'%')
        </if>
    </where>
</select>
```

测试

```
@Test
public void testGetByCondition()throws Exception{
    Users u = new Users();
    u.setUserName("小");
    u.setSex("1");
    u.setAddress("市");
    u.setBirthday(sf.parse("2002-01-19"));
    List<Users> list = usersMapper.getByCondition(u);
    list.forEach(users -> System.out.println(users));
}
```

运行结果：

```
==>  Preparing: select id,username,birthday,sex,address from users WHERE username like concat('%',?,'%') and birthday = ?
 ? and sex = ? and address like concat('%',?,'%')
==> Parameters: 小(String), 2002-01-19 00:00:00.0(Timestamp), 1(String), 市(String)
<==    Columns: id, username, birthday, sex, address
<==        Row: 6, 陈小明, 2002-01-19, 1, 重庆市
<==      Total: 1
Users{id=6, userName='陈小明', birthday=Sat Jan 19 00:00:00 CST 2002, sex='1', address='重庆市'}
```

4.5 <set>标签

使用 set 标签可以将动态的配置 SET 关键字，并剔除追加到条件末尾的任何不相关的逗号。使用 if+set 标签修改后，在进行表单更新的操作中，哪个字段中有值才去更新，如果某项为 null 则不进行更新，而是保持数据库原值。**切记：至少更新一列**。

```
<update id="updateBySet" parameterType="users">
    update users
    <set>
        <if test="userName != null and userName != ''">
            username = #{userName},
        </if>
        <if test="birthday != null">
            birthday = #{birthday},
        </if>
        <if test="sex != null and sex != ''">
            sex =#{sex},
        </if>
        <if test="address != null and address !=''">
            address = #{address}
```

```
        </if>
    </set>
    where id = #{id}
</update>
```

测试：

```
@Test
public void testUpdateSet()throws Exception{
    // Users u = new Users("哈哈",new Date(),"1","北京亦庄大兴");
    //Users u = new Users(3,"不知道",sf.parse("1998-08-08"),"2","北京亦庄大兴 888");
    Users u = new Users();
    u.setId(2);
    u.setUserName("认识张三不");
    //u.setSex("2");
    //u.setBirthday(sf.parse("2000-01-01"));
    int num = usersMapper.updateBySet(u);
    //切记切记:必须提交事务
    sqlSession.commit();
    System.out.println(num);
}
```

运行结果：

```
==>  Preparing: update users SET username=?, sex=? where uid=?
==> Parameters: 小黑(String), 女(String), 15(Integer)
<==    Updates: 1
Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@71248c21]
==>  Preparing: select uid,username,birthday,sex,address from users where uid=?
==> Parameters: 2(Integer)
<==    Columns: uid, username, birthday, sex, address
<==        Row: 2, 小黑, 1991-09-09, 女, 北京大兴
<==      Total: 1
Users{uid=2, username='小黑', sex='女', birthday=1991-09-09, address='北京大兴'}
```

4.6 <foreach>标签

<foreach>主要用来进行集合或数组的遍历，主要有以下参数：

collection：collection 属性的值有三个分别是 list、array、map 三种，分别对应的参数类型为：List、数组、map 集合。

item ：循环体中的具体对象。支持属性的点路径访问，如 item.age,item.info.details，在 list 和数组中是其中的对象，在 map 中是 value。

index ：在 list 和数组中,index 是元素的序号，在 map 中，index 是元素的 key，该参数可选。

open ： 表示该语句以什么开始

close ： 表示该语句以什么结束

separator ： 表示元素之间的分隔符，例如在 in()的时候，separator=","会自动在元素中间用“,“隔开，避免手动输入逗号导致 sql 错误，如 in(1,2,)这样。该参数可选。

4.6.1 循环遍历参数集合

```
<select id="getByIds" resultType="users">
    select <include refid="columns"></include>
    from users
    where id in
        <foreach collection="list" item="id" separator="," open="(" close=")">
            #{id}
        </foreach>
</select>
```

测试：

```
    @Test
    public void testCondition() {
        Users u = new Users( username: "赵",  sex: "男");
        // List<Users> list=mapper.findByCondition(u);
        List<Users> list = mapper.findByCondition1(u);
        System.out.println(list);
    }
```

运行结果：

```
==>  Preparing: select uid,username,birthday,sex,address from users WHERE username like '%赵%' and sex=?
==> Parameters: 男(String)
<==    Columns: uid, username, birthday, sex, address
<==        Row: 4, 赵六, 1992-02-02, 男, 北京西城
<==        Row: 7, 赵8888, 2020-08-10, 男, 北京海淀区杏石口路9888号
<==      Total: 2
[Users{uid=4, username='赵六', sex='男', birthday=1992-02-02, address='北京西城'}, Users{uid=7, username='赵8888', sex='男', birthday=2020-08-10, addres
```

4.6.2 批量删除

```
<!--      批量删除-->
<delete id="deleteBatch" >
    delete from users where id in
    <foreach collection="array" item="id" close=")" open="(" separator=",">
        #{id}
    </foreach>
</delete>
```

测试：

```java
@Test
public void testDeleteBatch(){
    Integer []arrs={1,3,5};
    int num = mapper.deleteBatch(arrs);
    //切记切记:提交
    session.commit();
    if(num > 0)
        System.out.println("删除成功!");
}
```

4.6.3 批量增加

```xml
<!--      批量增加-->
    <insert id="insertBatch" >
        insert into users(username,birthday,sex,address) values
        <foreach collection="list" separator="," item="u">
            (#{u.userName},#{u.birthday},#{u.sex},#{u.address})
        </foreach>
    </insert>
```

测试：

```java
@Test
public void testInsertBatch()throws  Exception{
    Users u1 = new Users( userName: "lisi1",
            new SimpleDateFormat( pattern: "yyyy-MM-dd").parse( source: "1999-01-01"), sex: "2", address: "大兴");
    Users u2 = new Users( userName: "lisi2",
            new SimpleDateFormat( pattern: "yyyy-MM-dd").parse( source: "2000-01-01"), sex: "1", address: "大兴");
    Users u3 = new Users( userName: "lisi3",
            new SimpleDateFormat( pattern: "yyyy-MM-dd").parse( source: "2001-01-01"), sex: "1", address: "大兴亦庄");
    List<Users> list = new ArrayList<>();
    list.add(u1);
    list.add(u2);
    list.add(u3);
    int num = mapper.insertBatch(list);
    //切记切记
    session.commit();
    if(num > 0){
        System.out.println("增加成功!");
    }
}
```

4.6.4 批量更新

```xml
<!--      有选择的批量更新,至少更新一列-->
    <update id="updateSet"    >
        <foreach collection="list" item="u" separator=";">
          update users
          <set>
                <if test="u.userName != null    and u.userName != ''">
                    username=#{u.userName},
```

```
            </if>
            <if test="u.birthday != null">
                birthday = #{u.birthday},
            </if>
            <if test="u.sex != null  and u.sex != ''">
                sex = #{u.sex},
            </if>
            <if test="u.address != null  and u.address != ''">
                address = #{u.address}
            </if>
        </set>
        where id = #{u.id}
    </foreach>
</update>
```

测试

```
@Test
public void testUpdateBatch()throws Exception{
    List<Users> list = new ArrayList<>();
    Users u1 = new Users(3,"张 1167",new SimpleDateFormat("yyyy-MM-dd").parse("1997-02-03"),"2","北京大兴亦庄 1111");
    Users u2 = new Users(4,"李 2267",new SimpleDateFormat("yyyy-MM-dd").parse("1998-02-03"),"1","北京大兴亦庄 2333");
    Users u3 = new Users(5,"王 3367",new SimpleDateFormat("yyyy-MM-dd").parse("1999-02-03"),"2","北京大兴亦庄 122");
    list.add(u1);
    list.add(u2);
    list.add(u3);
    int num = mapper.updateSet(list);
    session.commit();
    System.out.println(num);
}
```

注意：要使用批量更新，必须在 jdbc.properties 属性文件中的 url 中添加 &allowMultiQueries=true，允许多行操作。

4.7 指定参数位置

可以不使用对象的属性名进行参数值绑定，使用下标值。 mybatis-3.3 版本和之前的版本使用#{0},#{1}方式， 从 mybatis3.4 开始使用#{arg0}方式。

UsersMapper.java 接口中：

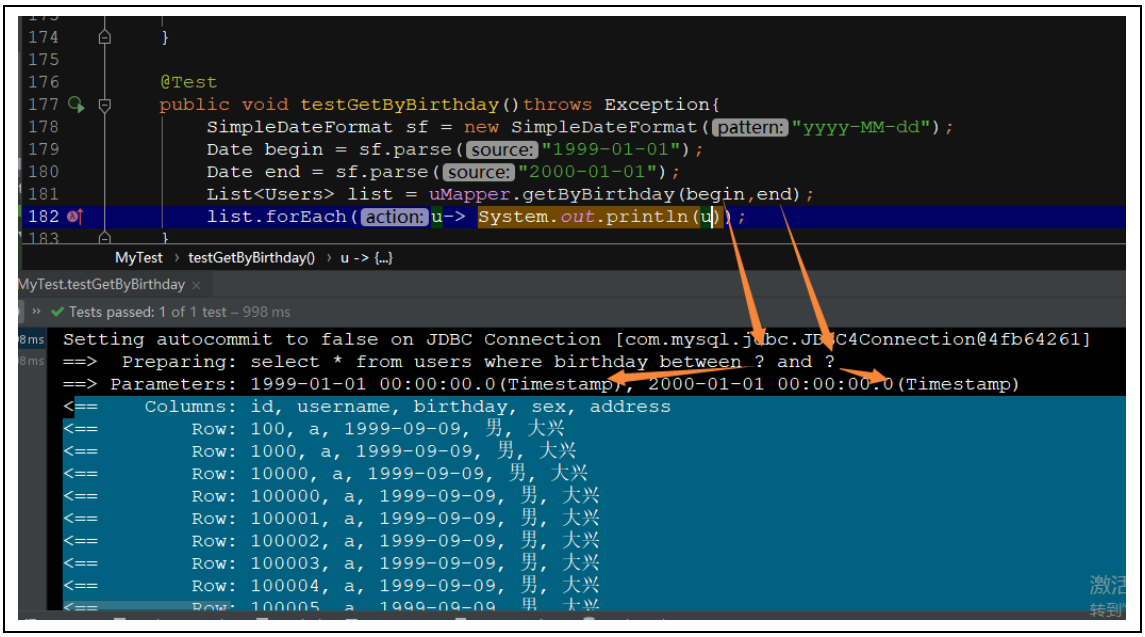
```
//查询指定日期范围内的用户信息,实体类的成员变量无法包含条件了,所以散给条件
```

```
List<Users> getByBirthday(Date begin, Date end);
```

UsersMapper.xml 文件中：

```
<select id="getByBirthday" resultType="users">
   select * from users where birthday between #{arg0} and #{arg1}
</select>
```

测试类：



4.8 @Param 指定参数名称

UsersMapper.java 接口中：

```
//切换列名进行模糊查询
//@Param("columnName"):这里定义的 columnName 的名称是要在 xml 文件中的${引用定义的名称}
List<Users> getByColunm(@Param("columnName") String columnName,
                        @Param("columnValue") String columnValue);
```

UsersMapper.xml 文件中：

```
<select id="getByColunm" resultType="users">
      select <include refid="columns"></include>
      from users
      where ${columnName} =#{columnValue}
</select>
```

测试类：

```
@Test
public void testGetByColumn(){
      List<Users> list = mapper.getByColunm("username","王五四");
```

```
        list.forEach(u-> System.out.println(u));
}
```

4.9 入参是 map

入参是 map,是因为当传递的数据有多个,不适合使用指定下标或指定名称的方式来进行传参,又加上参数不一定与对象的成员变量一致,考虑使用 map 集合来进行传递.map 使用的是键值对的方式.当在 sql 语句中使用的时候#{键名},${键名},用的是键的名称.

UsersMapper.java 接口中：

```
//入参是 map
List<Users> getByMap(Map<String,Date> map);
```

UsersMapper.xml 文件中：

```
<select id="getByMap" parameterType="map" resultType="users">
        select <include refid="columns"></include>
        from users
        where birthday between #{zarbegin} and #{zarend}
</select>
```

测试类中：

```
@Test
public void testGetByMap()throws Exception{
        Date begin = new SimpleDateFormat("yyyy-MM-dd").parse("1996-01-01");
        Date end = new SimpleDateFormat("yyyy-MM-dd").parse("1998-12-31");
        Map<String,Date> map = new HashMap<>();
        //放入 map 集合中的数据是键值对
        map.put("zarbegin",begin);
        map.put("zarend",end);
        List<Users> list = mapper.getByMap(map);
        list.forEach(u-> System.out.println(u));
}
```

4.10 返回值是 map

返回值是 map 的适用场景,如果的数据不能使用对象来进行封装,可能查询的数据来自多张表中的某些列,这种情况下,使用 map,但是 map 的返回方式破坏了对象的封装,返回来的数据是一个一个单独的数据, 之间不相关.**map 使用表中的列名或别名做为键名进行返回数据.**

4.10.1 map 封装返回值是一行

UsersMapper.java 接口中：

```
//返回值是一个值,是 map 类型,根据主键查用户对象
Map<String,Object> getReturnMapOne(int id);
```

UsersMapper.xml 文件中：

```
<select id="getReturnMapOne" resultType="map" parameterType="int">
    select id myid,username myusername,sex mysex,address myaddress,birthday
mybirthday
    from users
    where id=#{id}
</select>
```

测试类中：

```
@Test
public void testGetReturnMapOne(){
    Map<String,Object> map = mapper.getReturnMapOne(7);
    System.out.println(map);
    System.out.println(map.get("username"));
}
```

4.10.2 返回值是 map 多行

UsersMapper.java 接口中：

```
//使用 map 封装返回多个 map 的集合--->List<Map<String,Object>>
List<Map<String,Object>> getReturnMap();
```

UsersMapper.xml 文件中：

```
<select id="getReturnMap" resultType="map" >
    select <include refid="columns"></include>
    from users
</select>
```

测试类中：

```
@Test
public void testGetReturnMap(){
    List<Map<String,Object>> list = mapper.getReturnMap();
    list.forEach(map-> System.out.println(map));
}
```

4.11 列名与类中成员变量名称不一致

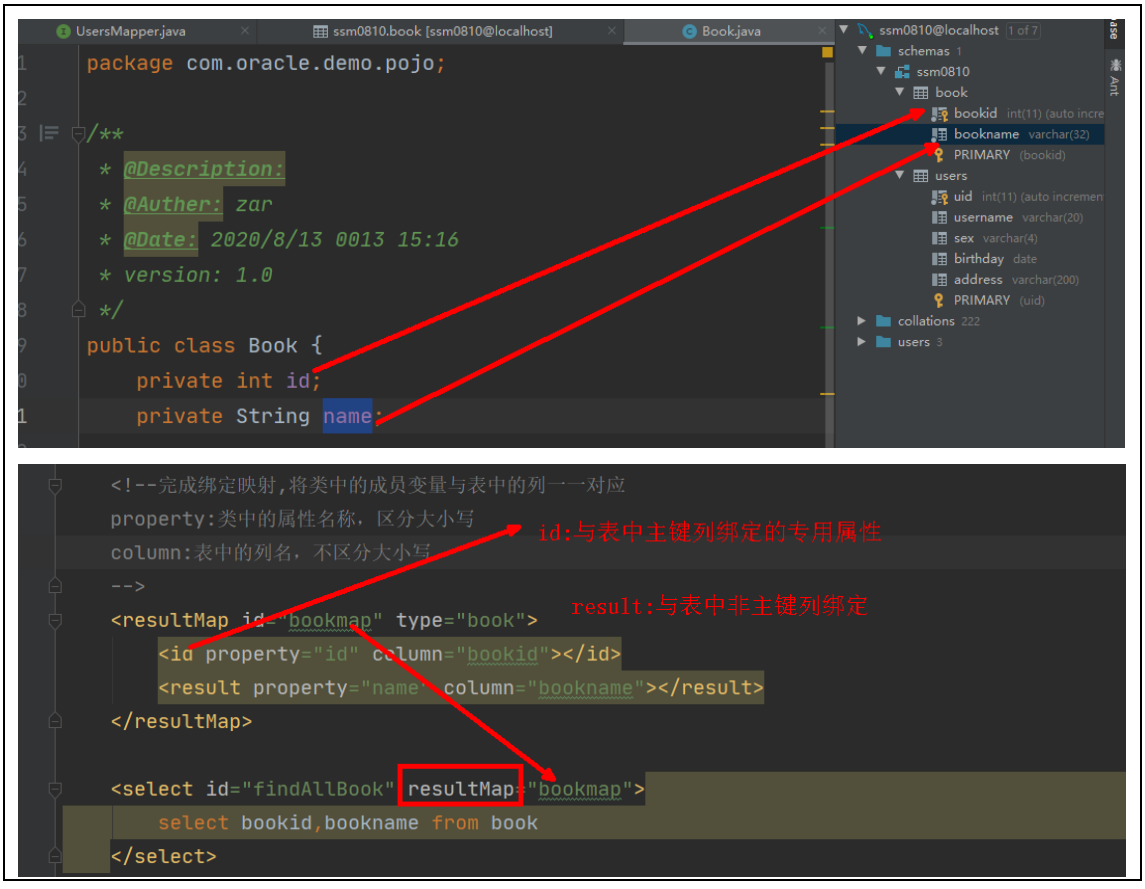
解决方案一：

使用列的别名，别名与类中的成员变量名一样，即可完成注入。

```
<select id="getAll" resultType="book">
      select bookid id,bookname name
      from book
</select>
```

解决方案二：

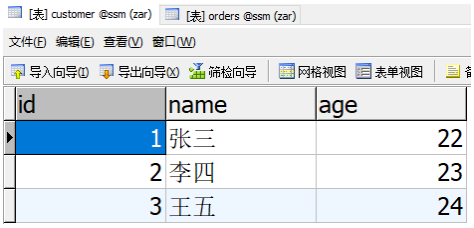
使用<resultMap>标签进行映射。



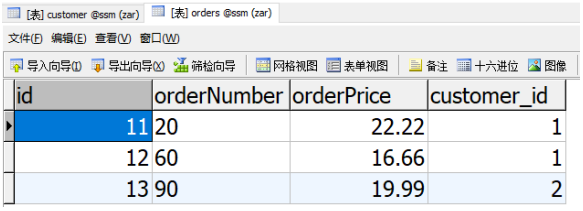
第5章 表的关联关系

我们通常说的关联关系有以下四种，一对多关联，多对一关联，一对一关联，多对多关联。关联关系是有方向的。如果是高并发的场景中，不适合做表的关联。

在多对一和一对多的关联关系中，我们使用订单表和客户表。



id	name	age
1	张三	22
2	李四	23
3	王五	24



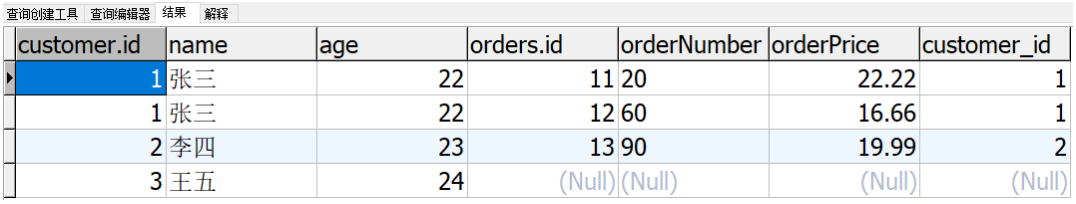
id	orderNumber	orderPrice	customer_id
11	20	22.22	1
12	60	16.66	1
13	90	19.99	2

等值连接下

```
select c.* ,o.* from customer c,orders o where c.id=o.customer_id
```

左外连接下

```
select c.* ,o.* from customer    c left join orders o on c.id=o.customer_id;
```



customer.id	name	age	orders.id	orderNumber	orderPrice	customer_id
1	张三	22	11	20	22.22	1
1	张三	22	12	60	16.66	1
2	李四	23	13	90	19.99	2
3	王五	24	(Null)	(Null)	(Null)	(Null)

5.1 一对多关联

在一对多关联关系中，一方(客户)中有多方(订单)的集合 ，所以要使用<collection>标签来映射多方的属性。

核心代码：

实体类

```
/**
 * 2021/8/19
 * 一方持有多方的集合
 */
public class Customer {
    private int id;
    private String name;
    private int age;

    //在客户的一方持有多方（订单）的集合
    private List<Orders> ordersList;
```

接口

```java
public interface CustomerMapper {
    //根据主键id查用户
    Customer getById(int id);
}
```

mapper.xml

```
<mapper namespace="com.bjpowernode.mapper.CustomerMapper">
    <resultMap id="customermap" type="customer">
        <!--Customer自身的成员变量
            private int id;
            private String name;
            private int age;
            //在客户的一方持有多方（订单）的集合
            private List<Orders> ordersList;===>被collection标签解析
        -->
        <id property="id" column="cid"></id>
        <result property="name" column="name"></result>
        <result property="age" column="age"></result>
        <!--Customer与之关联的订单Orders的信息
            private int id;
            private String orderNumber;
            private double orderPrice;
        -->
        <collection property="ordersList" ofType="orders">
            <id property="id" column="oid"></id>
            <result property="orderNumber" column="orderNumber"></result>
            <result property="orderPrice" column="orderPrice"></result>
        </collection>
    </resultMap>
    <!--
      //根据主键id查用户  Customer getById(int id);       使用关联查询
    -->
    <select id="getById" parameterType="int" resultMap="customermap">
      select c.id cid,name,age,o.id oid,orderNumber,orderPrice,customer_id
      from customer c left join orders o on c.id = o.customer_id
      where c.id = #{id}
    </select>
</mapper>
```

注意使用别名

第二种解决方案

```
    <!--使用嵌套查询的方式实现映射-->
    <resultMap id="customermap1" type="customer">
        <id property="id" column="id"></id>
        <result property="name" column="name"></result>
        <result property="age" column="age"></result>
        <!--column:是当前客户表的id，传给嵌套查询作为入参进行查询返回该客户名下的所有订单集合-->
        <collection property="ordersList" ofType="orders" column="id" select="selectOrdersByCustomerId" />
    </resultMap>
    <!--这种 解决方案中，不需要做表关联查询-->
    <select id="findById1" parameterType="int" resultMap="customermap1">
        select * from customer where id=#{id}
    </select>

    <select id="selectOrdersByCustomerId" parameterType="int" resultType="orders">
        select * from orders where customer_id=#{id}
    </select>
</mapper>
```

最终解决方案

每个 mapper 里只有自己的增删改查。

```
    <resultMap id="customermap1" type="customer">
        <id property="id" column="id"></id>
        <result property="name" column="name"></result>
        <result property="age" column="age"></result>
        <!--column:是当前客户表的id，传给嵌套查询作为入参进行查询返回该客户名下的所有订单集合-->
        <collection property="ordersList" ofType="orders" column="id"
                    select="com.oracle.demo.mapper.OrdersMapper.selectOrdersByCustomerId" />
    </resultMap>
    <!--这种 解决方案中，不需要做表关联查询-->
    <select id="findById1" parameterType="int" resultMap="customermap1">
        select * from customer where id=#{id}
    </select>
    <!--嵌套查询，为订单表提供查询-->
    <select id="findCustomerByOrdersCustomerid" parameterType="int" resultType="customer">
        select * from customer where id=#{id}
    </select>
</mapper>
```

调用 OrdersMapper.xml 中的按用户 ID 查该用户名下的所有订单

```
    <!--嵌套查询,是customerMapper要用的-->
    <select id="selectOrdersByCustomerId" parameterType="int" resultType="orders">
        select * from orders where customer_id=#{id}
    </select>
```

5.2 多对一关联

在多对一关联关系中，多方(订单)中持有一方(客户)的对象，要使用标签<association>标签来映射一方的属性。

核心代码：

实体类：

```
/**
 * 2021/8/19
 * 多方持有一方的对象
 */
public class Orders {
    private int id;
    private String orderNumber;
    private double orderPrice;

    //多方持有的一方的对象
    private  Customer customer;
```

接口：

```
public interface OrdersMapper {
    //根据订单id查订单信息
    Orders getById(int id);
}
```

mapper.xml

```
<mapper namespace="com.bjpowernode.mapper.OrdersMapper">
    <!--
      //根据订单id查订单信息
      Orders getById(int id);
        private int id;
        private String orderNumber;
        private double orderPrice;
       //多方持有的一方的对象
        private  Customer customer;
    -->
    <resultMap id="ordersmap" type="orders">
        <id property="id" column="oid"></id>
        <result property="orderNumber" column="orderNumber"></result>
        <result property="orderPrice" column="orderPrice"></result>
        <!--解析持有的对象的信息-->
        <association property="customer" javaType="customer">
            <id property="id" column="cid"></id>
            <result property="name" column="name"></result>
            <result property="age" column="age"></result>
        </association>
    </resultMap>
    <select id="getById" parameterType="int" resultMap="ordersmap">
        select o.id oid,orderNumber,orderPrice,customer_id,c.id cid ,name,age
        from orders o inner join customer c on o.customer_id = c.id
        where o.id = #{id}
    </select>
</mapper>
```

注意别名的使用

第二种解决方案

```
    <!--第二种解决方案：使用嵌套查询注入对象 -->
    <resultMap id="ordersmap1" type="orders">
        <id property="id" column="id"></id>
        <result property="orderNumber" column="orderNumber"></result>
        <result property="orderPrice" column="orderPrice"></result>
    <!--colunm:一定是列名，一定是当前类对应的表中的列名，一定是与另外一方有关联的列名-->
        <association property="customer" javaType="customer" column="customer_id"
                     select="findCustomerByOrdersCustomerid"/>
    </resultMap>
    <!--查询订单操作-->
    <select id="findById2" parameterType="int" resultMap="ordersmap1">
        select * from orders where id=#{id}
    </select>
    <!--嵌套查询，使用订单表中的客户ID，到客户表中查该客户的相关信息-->
    <select id="findCustomerByOrdersCustomerid" parameterType="int" resultType="customer">
        select * from customer where id=#{id}
    </select>
</mapper>
```

最终解决方案：

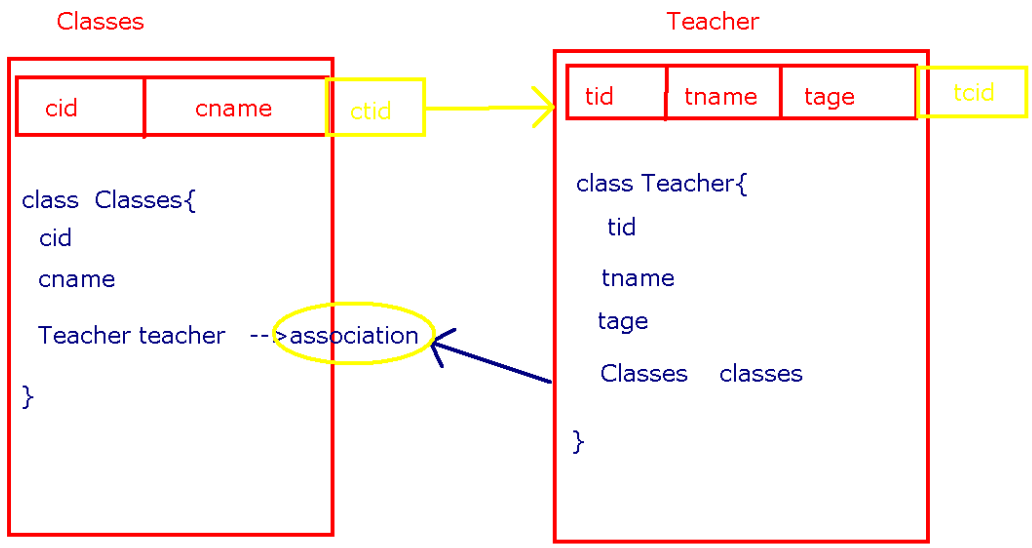
```
    <!--第二种解决方案：使用嵌套查询注入对象 -->
    <resultMap id="ordersmap1" type="orders">
        <id property="id" column="id"></id>
        <result property="orderNumber" column="orderNumber"></result>
        <result property="orderPrice" column="orderPrice"></result>
    <!--colunm:一定是列名，一定是当前类对应的表中的列名，一定是与另外一方有关联的列名-->
        <association property="customer" javaType="customer" column="customer_id"
                     select="com.oracle.demo.mapper.CustomerMapper.findCustomerByOrdersCustomerid"/>
    </resultMap>
    <!--查询订单操作-->
    <select id="findById2" parameterType="int" resultMap="ordersmap1">
        select * from orders where id=#{id}
    </select>
    <!--嵌套查询,是customerMapper要用的-->
    <select id="selectOrdersByCustomerId" parameterType="int" resultType="orders">
        select * from orders where customer_id=#{id}
    </select>
</mapper>
```

调用 CustomerMapper.xml 里的按用户主键查询用户信息的方法

```
    <!--嵌套查询，为订单表提供查询-->
    <select id="findCustomerByOrdersCustomerid" parameterType="int" resultType="customer">
        select * from customer where id=#{id}
    </select>
```

5.3 一对一关联

一个班级只有一个授课老师，一个老师也只为一个班级授课。



实体类：

```
public class Classes {
    private int cid;
    private String cname;
    private Teacher teacher;
```

接口：

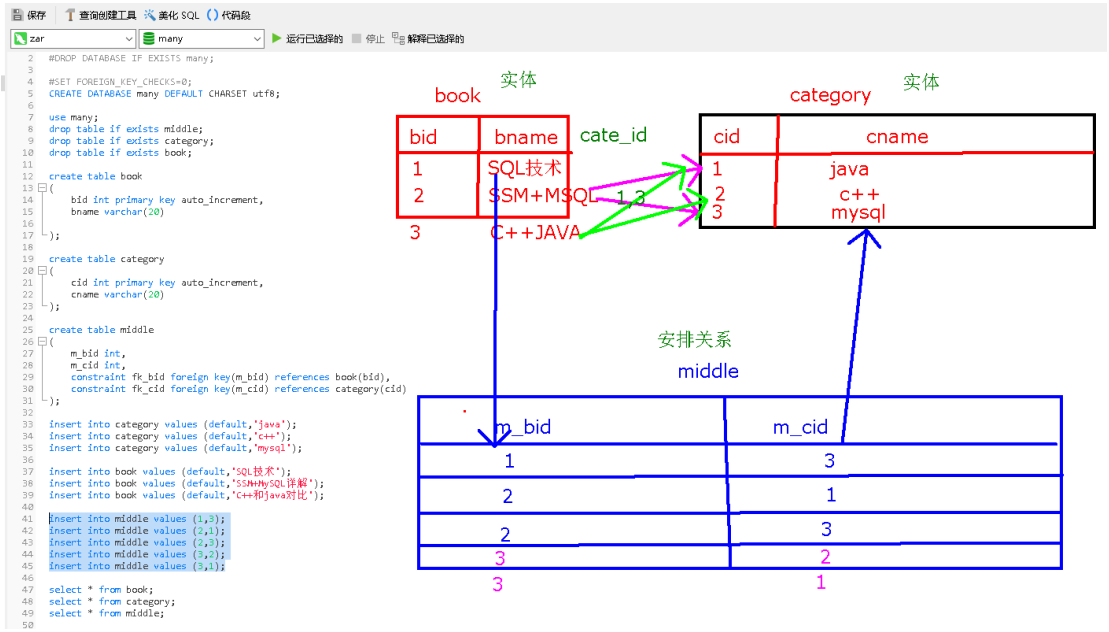
```
public interface ClassesMapper {
    //查询指定ID的班级信息
    public Classes findById(int cid);
}
```

mapper.xml

```
<mapper namespace="com.oracle.demo.mapper.ClassesMapper">
    <resultMap id="classesmap" type="classes">
        <id property="cid" column="cid"></id>
        <result property="cname" column="cname"></result>
        <association property="teacher" javaType="teacher">
            <id property="tid" column="tid"></id>
            <result property="tname" column="tname"></result>
            <result property="age" column="tage"></result>
        </association>
    </resultMap>
    <select id="findById" parameterType="int" resultMap="classesmap">
        select c.*,t.* from classes c,teacher t
        where c.ctid=t.tid and c.cid=#{cid}
    </select>
</mapper>
```

5.4 多对多关联

多对多关联中，需要通过中间表化解关联关系。中间表描述两张主键表的关联。中间表没有对应的实体类。Mapper.xml 文件中也没有中间表的对应标签描述，只是在查询语句中使用中间表来进行关联。



实体类：

```
public class Book {
    private int bid;
    private String bname;
    //目的：就是在查询图书时，将图书所属的类型全部取出来
    private List<Category> categoryList=new ArrayList<>();
```

接口：

```
public interface BookMapper {
    //查询全部图书，并且查询每本书所属的类别
    public List<Book> findAll();
}
```

Mapper.xml

```xml
<mapper namespace="com.oracle.demo.mapper.BookMapper">
    <resultMap id="bookmap" type="book">
        <id property="bid" column="bid"></id>
        <result property="bname" column="bname"></result>
        <!--关联属性-->
        <collection property="categoryList" ofType="category">
            <id property="cid" column="cid"></id>
            <result property="cname" column="cname"></result>
        </collection>
    </resultMap>
    <select id="findAll" resultMap="bookmap">
        select *
            from book b inner join middle m on b.bid = m.m_bid
            inner join category c on m.m_cid = c.cid
    </select>
</mapper>
```

总结：无论是什么关联关系，如果某方持有另一方的集合，则使用<collection>标签完成映射，如果某方持有另一方的对象，则使用<association>标签完成映射。

第6章 事务

Mybatis 框架是对 JDBC 的封装，所以 Mybatis 框架的事务控制方式有两种，一种是容器进行事务管理的，一种是程序员手工决定事务的提交与回滚。

SqlMapConfig.xml文件中设定的事务处理的方式。

```xml
<environment id="development">
    <transactionManager type="JDBC"></transactionManager>
    <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>
```

Connection 对象的 setAutoCommit()方法来设置事务提交方式的。自动提交和手工提交。该标签用于指定 MyBatis 所使用的事务管理器。

MyBatis 支持两种事务管理器类型：**JDBC 与 MANAGED**。

JDBC：使用 JDBC 的事务管理机制。即，通过 Connection 的 commit()方法提交，通过

rollback()方法回滚。但默认情况下，MyBatis 将自动提交功能关闭了，改为了手动提交。即程序中需要显式的对事务进行提交或回滚。从日志的输出信息中可以看到。

```
Opening JDBC Connection
Created connection 2147046752.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@7ff95560]
==>  Preparing: select id,username,birthday,sex,address from users where birthday between ? and
```

在获得 SqlSession 的时候,如果 openSession()是无参或者是 false,则必须手工提交事务，如果 openSession(true),则为自动提交事务，在执行完增删改后无须 commit(),事务自动提交。

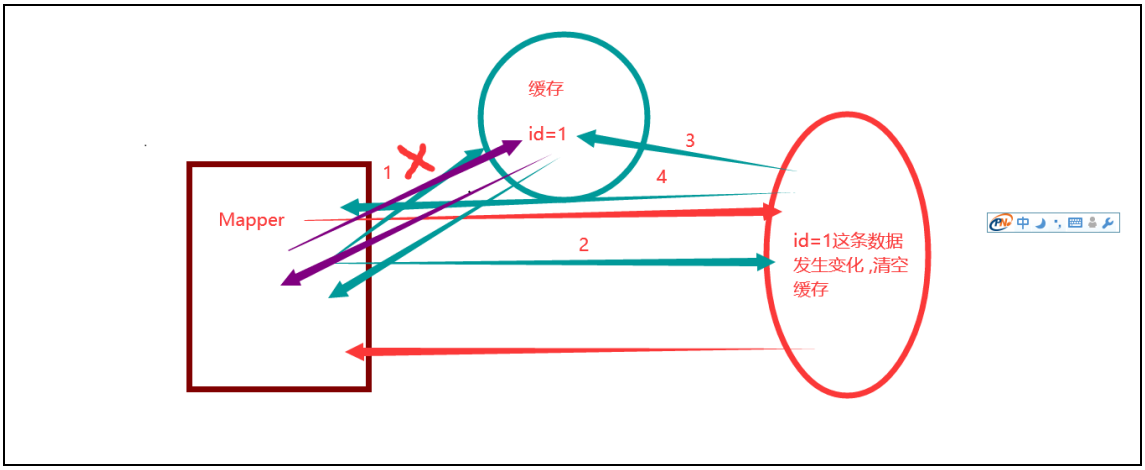
session = factory.openSession(true);

第7章 缓存

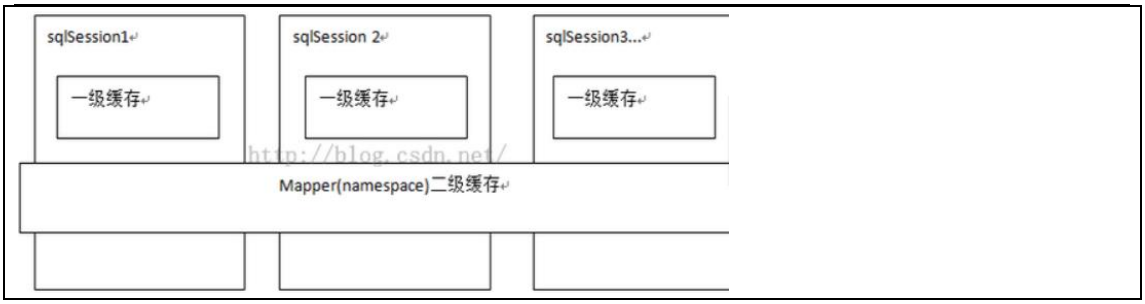
将用户经常查询的数据放在缓存（内存）中，用户去查询数据就不用从磁盘上(关系型数据库数据文件)查询，从缓存中查询，从而**提高查询效率**，解决了高并发系统的性能问题。mybatis 提供查询缓存，用于减轻数据库压力，提高数据库性能。

7.1缓存执行机制

在进行数据库访问时，首先去访问缓存，如果缓存中有要访问的数据，则直接返回客户端，如果没有则去访问数据库，在库中得到数据后，先在缓存放一份，再返回客户端。如下图。



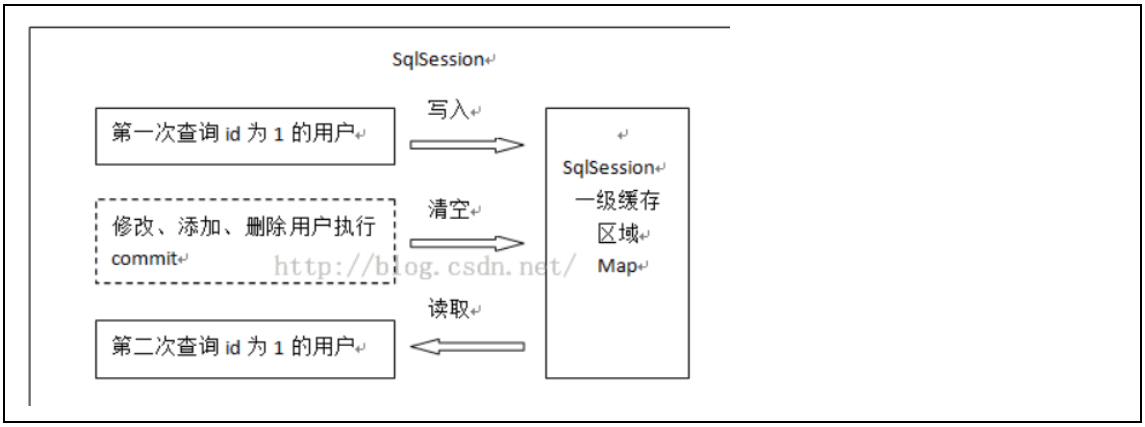
mybaits 提供一级缓存和二级缓存。默认开启一级缓存。



7.2 一级缓存

第一次发起查询用户 id 为 1 的用户信息，先去找缓存中是否有 id 为 1 的用户信息，如果没有，从数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。如果 sqlSession 去执行 commit 操作（执行插入、更新、删除），则清空 SqlSession 中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。

第二次发起查询用户 id 为 1 的用户信息，先去找缓存中是否有 id 为 1 的用户信息，缓存中有，直接从缓存中获取用户信息。如果没有重复第一次查询操作。如下图。



测试代码

```
@Test
public void testCacheOne() {
    //一级缓存基于 SqlSession 的
    //取出用户 ID 为 7 的用户
    Users u1 = mapper.getById(7);
    System.out.println("第一次访问数据库得到的 u=" + u1);
    //再取一次,看看有没有访问数据库,就知道开没有开启一级缓存
    Users u2 = mapper.getById(7);
    System.out.println("第二次访问数据库得到的 u=" + u2);
}
@Test
public void testCacheOneClose() {
```

```
        //一级缓存基于 SqlSession 的
        //取出用户 ID 为 7 的用户
        Users u1 = mapper.getById(7);
        System.out.println("第一次访问数据库得到的 u=" + u1);
        //关闭 session,也会清除缓存
        session.close();
        //再取一次,看看有没有访问数据库,就知道开没有开启一级缓存
        session = factory.openSession();
        mapper = session.getMapper(UsersMapper.class);
        Users u2 = mapper.getById(7);
        System.out.println("第二次访问数据库得到的 u=" + u2);
}
@Test
public void testCacheOneCommit() {
        //一级缓存基于 SqlSession 的
        //取出用户 ID 为 7 的用户
        Users u1 = mapper.getById(7);
        System.out.println("第一次访问数据库得到的 u=" + u1);

        //完成数据更新操作
        int num = mapper.update(new Users(38, "abc", new Date(), "1", "朝阳"));
        session.commit();
        System.out.println(num + "==================");

        Users u2 = mapper.getById(7);
        System.out.println("第二次访问数据库得到的 u=" + u2);

}
```

7.3 二级缓存

mybaits 的二级缓存是 mapper 范围级别，除了在 SqlMapConfig.xml 设置二级缓存的总开关，还要在具体的 mapper.xml 中开启二级缓存，并且要让实体类实现 serializable 接口。

1. 在核心配置文件　　SqlMapConfig.xml 中加入设置

```
    <!--配置打印sql语句-->
    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
        <!--开启二级缓存-->
        <setting name="cacheEnabled" value="true"/>
    </settings>
```

2. 在 UsersMapper.xml 文件中二启二级缓存，使用<cache></cache>

```
<mapper namespace="com.oracle.demo.mapper.UsersMapper">
    <cache></cache>
```

3. 实体类必须实现 java.io.serializable 接口，保证实体可序列化

```
public class Users implements Serializable {
    private int uid;
    private String username;
    private String sex;
    private Date birthday;
    private String address;
```

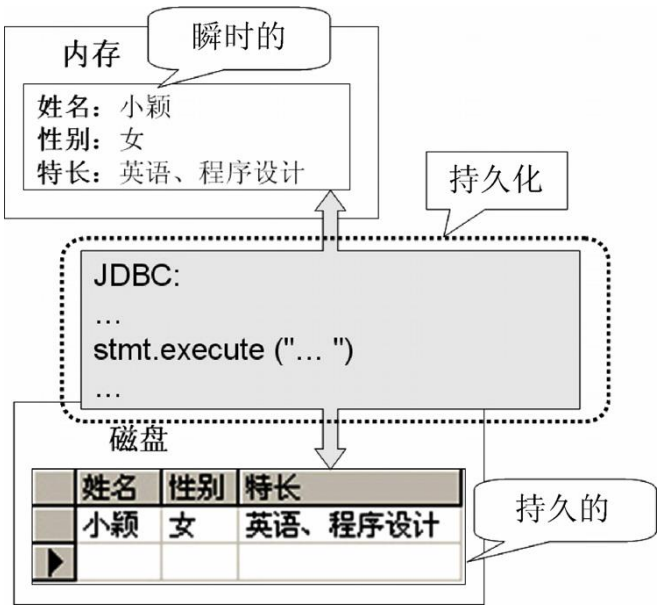
测试代码：

```
public void testTwoCache(){
    //创建不同的session对象
    SqlSession session1=factory.openSession();
    SqlSession session2=factory.openSession();
    SqlSession session3=factory.openSession();
    //通过session1取user
    UsersMapper usersMapper=session1.getMapper(UsersMapper.class);
    Users u1=usersMapper.findUsersById(1);
    System.out.println("session1取用户："+u1);
    session1.close();

    //如果发生了commit操作，则清空缓存
    UsersMapper usersMapper3=session3.getMapper(UsersMapper.class);
    Users u3=usersMapper3.findUsersById(1);
    u3.setAddress("1111111111111");
    usersMapper3.updateUserByCondition(u3);
    session3.commit();
    session3.close();

    //通过session2取user
    UsersMapper usersMapper2=session2.getMapper(UsersMapper.class);
    Users u2=usersMapper2.findUsersById(1);
    System.out.println("session2取用户："+u2);
    session2.close();
}
```

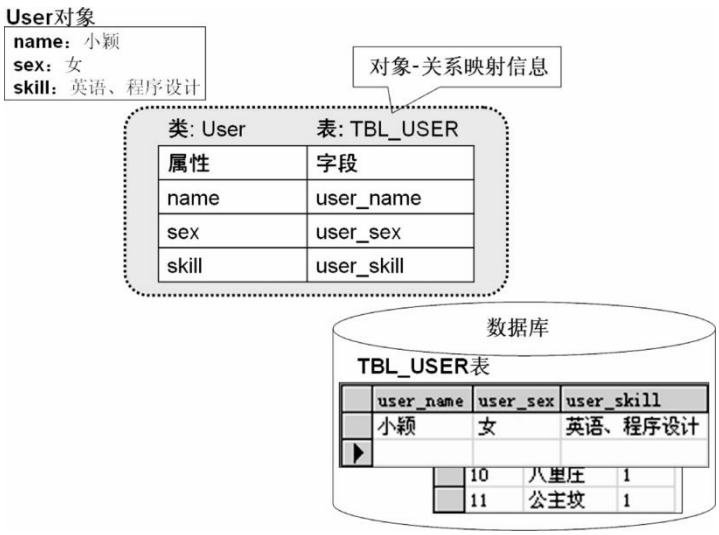
第8章 什么是 ORM

ORM(Object Relational Mapping):对象关系映射

编写程序的时候，以面向对象的方式处理数据，保存数据的时候，却以关系型数据库的方式存储。



代码中：



持久化的操作：将对象保存到关系型数据库中 ,将关系型数据库中的数据读取出来以对象的形式封装

O: java 中的对象

```
public class Users implements java.io.Serializable {
    private Long id;
    private String name;
    private String password;
    private String telephone;
    private String username;
    private String isadmin;
    private Set houses = new HashSet(0);
    set()….get()……
}
```

R:数据库中的表

```
create table USERS
(
   ID              NUMBER(10) primary key,
   NAME            VARCHAR2(50),
   PASSWORD    VARCHAR2(50),
   TELEPHONE VARCHAR2(15),
   USERNAME    VARCHAR2(50),
   ISADMIN      VARCHAR2(5)
);
```

M:映射

```
<collection property="orders" ofType="order">
        <id property="id" column="oid"></id>
        <result property="orderNumber" column="ordernumber"></result>
        <result property="orderPrice" column="orderprice"></result>
</collection>
```

MyBatis 是一个非常优秀的 ORM 框架。