[Mybatis](https://so.csdn.net/so/search?q=Mybatis&spm=1001.2101.3001.7020)

**一．概述**

1. mybatis是什么？有什么特点？

它是一款**半自动的ORM**持久层框架，具有较高的SQL灵活性，**支持高级映射(一对一，一对多)，动态SQL，延迟加载和缓存**等特性，但它的数据库无关性较低

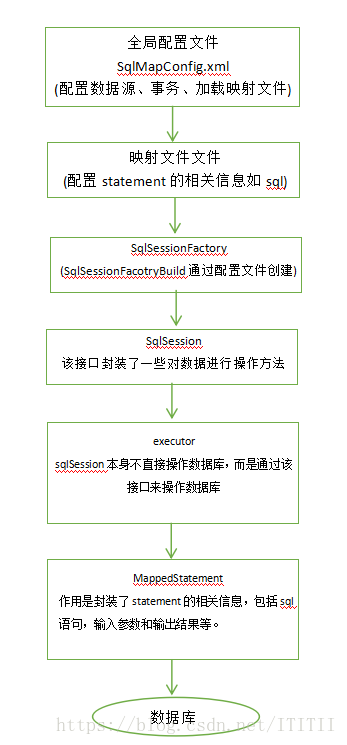
ORM---对象关系映射，指的就是在Java对象和数据库的关系模型之间建立一种对应关系

**二．快速入门**

只需要通过如下几个步骤，即可用mybatis快速进行持久层的开发

1. 编写全局配置文件
2. 编写mapper映射文件
3. 加载全局配置文件，生成SqlSessionFactory
4. 创建SqlSession，调用mapper映射文件中的SQL语句来执行CRUD操

**三．原理**



**四．Mybatis全局配置文件**

SqlMapConfig.xml是Mybatis的全局配置文件，它的名称可以是任意，但是一般命名都为（SqlMapConfig）

##### 4.1.全局配置文件的类容和顺序

**Properties(属性)**

**Settings（全局参数设置）**

**typeAliases（类型别名）**

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境信息集合）

environment（单个环境信息）

transactionManager（事物）

dataSource（数据源）

**mappers（映射器）**

##### 4.2.常见配置详解

**properties标签：**

Mybatis可以通过该标签来读取java配置信息：

例如在工程中对数据源信息写在**db.properties**文件中，可以通过properties标签来加载该文件。

**db.driver=com.mysql.jdbc.Driver**

**db.url=jdbc:mysql://localhost:3306/mybatis**

**db.username=root**

**db.password=12345678**

**SqlMapConfig.xml使用properties标签：**

**<!-- 通过properties标签，读取java配置文件的内容 -->**

**<properties resource="db.properties" />**

**<!-- 配置mybatis的环境信息 -->**

**<environments default="development">**

**<environment id="development">**

**<!-- 配置JDBC事务控制，由mybatis进行管理 -->**

**<transactionManager type="JDBC"></transactionManager>**

**<!-- 配置数据源，采用dbcp连接池 -->**

**<dataSource type="POOLED">**

**<property name="driver" value="${db.driver}"/>**

**<property name="url" value="${db.url}"/>**

**<property name="username" value="${db.username}"/>**

**<property name="password" value="${db.password}"/>**

**</dataSource>**

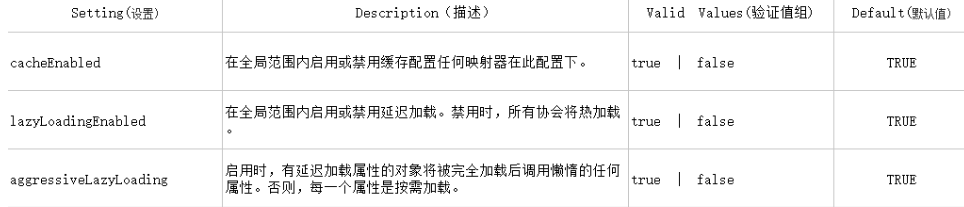
**</environment>**

**</environments>**

**settings标签：**

该标签时mybatis的全局设置，该设置会影响mybatis的运行。

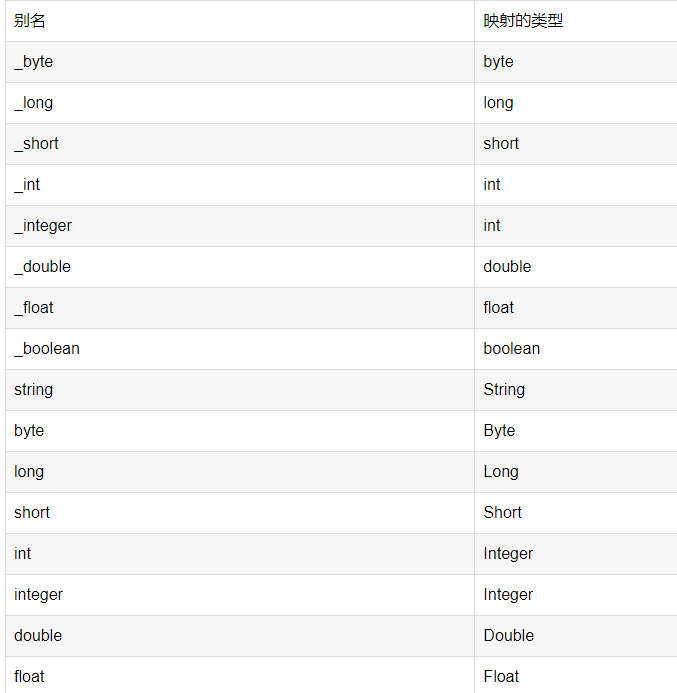
一般我们使用使用该标签来**开启二级缓存和懒加载**。



**typeAliases标签**

该标签是对po类进行别名设置，这样，在后面使用po类的时候就可以直接通过别名引用，而不需要通过po类的全限定名来引用。这样可以提高我们的开发效率

首先介绍下Mybatis的默认提供的别名有：



**自定义单个别名**：这种方式只能定义单个类的别名。

下面的代码就是把com.lc.mybatis.po.User类定义为user的别名

<typeAliases>

<!-- 设置单个别名 -->

<typeAlias type="com.lc.mybatis.po.User" alias="user"/>

</typeAliases>

**自定义之批量定义别名**：

下面代码是把com.lc.mybatis.po类下的所有类都声明别名，默认的别名就是类名（类名大小写都可以）

<!-- 设置别名 -->

<typeAliases>

<!-- 批量设置别名 -->

<!-- package:指定包名称来为该包下的po类声明别名，默认的别名就是类名（类名首字母大小写都可以） -->

<package name="com.lc.mybatis.po"/>

</typeAliases>

**mappers标签**

该标签的作用是加载映射文件

方式一：<mapper resource=""/>

该方式是加载相对于类路径下的映射文件：

**<mappers>**

**<mapper resource="sqlmap/User.xml"/>**

**</mappers>**

方式二：<mapper url=""/>

该方式使用全限定路径

**<mapper url="file:///D:\workspace\_spingmvc\mybatis\_01\config\sqlmap\User.xml" />**

方式三：<mapper class=""/>

该方式使用mapper接口的全限定类名

**<mapper class="cn.itcast.lc.mapper.UserMapper"/>**

此方式要求：

**Mapper接口Mapper映射文件名称相同且在同一个目录下。**

方式四：<package name=""/>

该方式是加载指定包下的所有映射文件

**<package name="cn.lc.mybatis.mapper"/>**

此方式要求：

    Mapper接口Mapper映射文件名称相同且在同一个目录下

**五、映射文件**

**5.1.输入映射parameterType**

**第一种：简单类型**

**#{}表示占位符**?，parameterType接收简单类型的参数时，里面的名称可以任意

**<select id="findUserById" parameterType="java.lang.Integer" resultType="user">**

**SELECT \* FROM USER WHERE id = #{id}**

**</select>**

**${}表示拼接符**，parameterType接收简单类型的参数时，里面的名称**必须是value**

**<select id="findUsersByName" parameterType="java.lang.String" resultType="com.lc.mybatis.po.User">**

**SELECT \* FROM USER WHERE username LIKE "%${value}%"**

**</select>**

**第二种：pojo类型**

这里通过用户的用户名进行模糊查询演示[pojo](https://so.csdn.net/so/search?q=pojo&spm=1001.2101.3001.7020)类型

在映射文件中添加模糊查询语句：

**<!-- parameterType传递pojo类型 -->**

**<select id="findUsersByPojo" parameterType="com.lc.mybatis.po.User" resultType="com.lc.mybatis.po.User">**

**SELECT \* FROM USER WHERE username LIKE "%${username}%"**

**</select>**

user类

**public class User {**

**private Integer id;**

**private String username;**

**private Date birthday;**

**private String sex;**

**private String address;**

**}**

测试类：

**public class UserDao{**

**//根据用户名进行模糊查询**

**@Override**

**public List<User> findUserByPojo(){**

**//全局配置文件路径：**

**String resource = "SqlMapConfig.xml";**

**InputStream inputStream = Resources.getResourceAsStream(resource);**

**//创建SqlSessionFactory**

**SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);**

**SqlSession sqlSession = sqlSessionFactory.openSession();**

**User user = new User();**

**user.setUsetname("张三");**

**List<User> users = (List<User>)sqlSession.selectList("test.findUsersByPojo",user);//传入pojo**

**System.out.println(users);**

**sqlSession.close();**

**return users;**

**}**

**}**

**第三种：包装类型pojo**

这里通过用户名和用户地址对用户进行查询来演示包装类型pojo：

首先创建包装pojo类

**public class UserVO {**

**private User user;**

**public User getUser() {**

**return user;**

**}**

**public void setUser(User user) {**

**this.user = user;**

**}**

**}**

在映射文件中添加查询语句：

**<!-- parameterType传递pojo包装类型 -->**

**<select id="findUsersByPojo1" parameterType="com.lc.mybatis.po.UserVO" resultType="user">**

**SELECT \* FROM USER WHERE username LIKE "%${user.username}%" AND address=#{user.address}**

**</select>**

测试类：

**public class UserDao{**

**//根据用户名和地址进行查询**

**@Override**

**public List<User> findUserByPojo1(){**

**//全局配置文件路径：**

**String resource = "SqlMapConfig.xml";**

**InputStream inputStream = Resources.getResourceAsStream(resource);**

**//创建SqlSessionFactory**

**SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);**

**SqlSession sqlSession = sqlSessionFactory.openSession();**

**User user = new User();**

**user.setUsetname("张三");**

**user.setAddress("郝汉山");**

**UserVO userVo = new UserVO();**

**userVo.setUser(user);**

**List<User> users = (List<User>)sqlSession.selectList("test.findUsersByPojo1",userVo);//传入pojo包装类**

**System.out.println(users);**

**sqlSession.close();**

**return users;**

**}**

**}**

**第四种：map集合类型**

这里通过查询用户信息演示：

在映射文件中添加该查询语句：

**<!-- parameterType传递hashmap类型 -->**

**<select id="findUsersByMap" parameterType="java.util.Map" resultType="user">**

**SELECT \* FROM USER WHERE username LIKE "%${username}%" AND address=#{address}**

**</select>**

测试方法：

**public class UserDao{**

**//根据用户名和地址进行查询**

**@Override**

**public List<User> findUserByMap(){**

**//全局配置文件路径：**

**String resource = "SqlMapConfig.xml";**

**InputStream inputStream = Resources.getResourceAsStream(resource);**

**//创建SqlSessionFactory**

**SqlSessionFactory sqlSessionFactory=new SqlSessionFactoryBuilder().build(inputStream);**

**SqlSession sqlSession = sqlSessionFactory.openSession();**

**Map<String,String> map = new HashMap<>();**

**map.put("username","张三");**

**map.put("address","郝汉山");**

**List<User> users = (List<User>) sqlSession.selectList("test.findUsersByMap",map);//传入pojo包装类**

**System.out.println(users);**

**sqlSession.close();**

**return users;**

**}**

**}**

**5.2.resultType结果映射**

resultType结果映射要求：**需要查询结果的列名和映射的对象的属性名一致**，这样才能映射成功。如果映射没成功也不会报错，只是映射结果中对象的相应属性没有值，为空。如果映射的列名和对象中的属性名全部不一致，那么映射的对象为空。如果在使用sql语句查询的时候给查询结果列设置了别名，则别名要和映射结果对象的属性名一致，这样才能保证映射成功。

**第一种：简单类型**

注意： 如果结果映射为简单类型，则需要查询的结果为一列才能映射成功。

例如：查询用户表中用户的总数。

映射文件为：

**<!-- resultType:输出为简单类型 -->**

**<select id="findUserCount" resultType="int">**

**select count(\*) from user;**

**</select>**

测试代码：

**public class UserDao{**

**@Override**

**public int findUserCount(){**

**//全局配置文件路径：**

**String resource = "SqlMapConfig.xml";**

**InputStream inputStream = Resources.getResourceAsStream(resource);**

**//创建SqlSessionFactory**

**SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);**

**SqlSession sqlSession = sqlSessionFactory.openSession();**

**int userCount= sqlSession.selectOne("test.findUserCount");**

**System.out.println(userCount);**

**sqlSession.close();**

**return userCount;**

**}**

**}**

**第二种：pojo结果映射**

这里操作pojo输入映射。

**5.3.resultMap结果映射**

使用resultMap结果映射时，不需要查询出来的结果集的列名和映射结果对象的属性名相同，但是需要声明一个resultMap，手动的方式来对列名和对象属性进行映射。（resultMap一般用于多表关联映射）

例如：通过查询用户表中的用户，并对查询出来的用户表的列名设置别名。

映射文件添加查询语句：

[id]：定义resultMap的唯一标识

[type]：定义该resultMap最终映射的pojo对象

[id标签]：映射结果集的唯一标识列，如果是多个字段联合唯一，则定义多个id标签

[result标签]：映射结果集的普通列

[column]：SQL查询的列名，如果列有别名，则该处填写别名

[property]：pojo对象的属性名

<!-- 如果查询出来的列名有别名就不能通过resultType来接收输出类型了。需要通过resultMap来声明传出类型（resultMap需要声明） -->

<resultMap type="user" id="userMap">

<!-- id标签：专门查询结果中唯一列映射 -->

<id column="id\_" property="id"/>

<!-- result标签：映射查询结果中的普通列 -->

<result column="username\_" property="username"/>

<result column="address\_" property="address"/>

</resultMap>

<select id="findUserResultMap" parameterType="int" resultMap="userMap">

select id id\_,username username\_,address address\_ from user where id=#{id}

</select>

测试类：

public class UserDao{

@Override

public User findUserResultMap(){

//全局配置文件路径：

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

//创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession sqlSession = sqlSessionFactory.openSession();

User user = sqlSession.selectOne("test.**findUserResultMap**",1);

System.out.println(user);

sqlSession.close();

return user;

}

}

**5.4.动态sql**

在mybatis中提供了一些动态sql标签，可以让我们开发效率更快，这些动态sql语句可以增加我们写的sql语句的重用性，**常用的动态sql语句标签有：if标签、sql片段(需要先定义后使用)、where标签、foreach标签**

**六、Mybatis关联映射**

**Mybatis的三种关联关系：一对一、一对多、多对多**

#### 6.1、mybatis 一对一映射关系

在< resultMap >元素中包含一个**< association >**子元素，MyBatis就是通过该元素来处理一对一关联关系的property：指定映射到的实体类对象属性，与表字段一一对应。

* column：指定表中对应的字段。
* **javaType**：指定映射到实体对象属性的类型。
* select：指定引入嵌套查询的子SQL语句，用于关联映射中的嵌套查询。
* fetchType：指定在关联查询时是否启用延迟加载，有lazy和eager两个属性值，默认值为lazy（默认关联映射延迟加载）。
* < association>元素有如下两种配置方式

< ！- - 方式一：嵌套查询-->

< association property="card" column="card\_id" javaType="com.Jorya.model.StudentIdCard"

select="com.Jorya.mapper.StudentIdCardMapper.findCodeById"/>

<!-- 方式二： 嵌套结果-->

<association property="card" javaType="com.Jorya.Model.StudentIdCard">

<id property="id" column="card\_id"/>

<result property="code" column="code"/>

</association>

MyBatis在映射文件中加载关联关系对象主要通过两种方式**：嵌套查询和嵌套结果**。

**嵌套查询**是指通过**执行另一条SQL映射语句来返回预期的复杂类型**；

**嵌套结果**是**使用嵌套结果映射来处理重复的联合结果的子集**。

举例：

创建持久化类：学生证类StudentIdCard和学生类Student

public class StudentIdCard {

private Integer id;

private String code;

}

public class Student {

private Integer id;

private String name;

private String sex;

private StudentIdCard studentIdCard;

}

**一个学生对应一个学生卡 1对1**

创建学生证映射文件StudentIdCardMapper.xml和学生映射文件StudentMapper.xml，并在两个映射文件中编写一对一关联映射查询的配置信息

**StudentIdCardMapper.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.Jorya.mapper.StudentIdCardMapper">

<select id="**findStudentIdCardById"** parameterType="Integer" resultType="StudentIdCard">

select \* from oto\_studentidcard where id=#{id}

</select>

</mapper>

**1.42　StudentMapper.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.Jorya.mapper.StudentMapper">

<!-- **嵌套查询**，通过执行一条SQL映射语句来返回预期的特殊类型 -->

<select id="findStudentById" parameterType="Integer"

resultMap="StudentIdCardWithStudentResult">

select \* from oto\_student where id=#{id}

</select>

<resultMap type="Student" id="StudentIdCardWithStudentResult">

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

<result property="name" column="name"/>

<result property="sex" column="sex"/>

**<!-- 一对一，association 使用select 属性引入另一条SQL语句 -->**

**<association property="studentIdCard" column="card\_id" javaType="StudentIdCard"**

**select="com.Jorya.mapper.StudentIdCardMapper.findStudentIdCardById"/>**

</resultMap>

**<!--嵌套查询和嵌套结果查询是两种方法-->**

<!-**-嵌套结果**，通过嵌套结果映射来处理重复的联合结果的子集-->

<select id="findStudentById2" parameterType="Integer"

resultMap="StudentIdCardWithStudentResult2">

select s.\*,sidcard.code

from oto\_student s, oto\_studentidcard sidcard

where s.card\_id=sidcard.id and s.id=#{id}

</select>

<resultMap type="Student" id="StudentIdCardWithStudentResult2">

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

<result property="name" column="name" />

<result property="sex" column="sex" />

**<association property="studentIdCard" javaType="StudentIdCard">**

**<id property="id" column="card id" />**

**<result property="code" column="code" />**

**</association>**

</resultMap>

</mapper>

**测试**

public class MybatisTest {

@Test

public void findUserByNameAndJobs() throws Exception{

//通过工具类生产SqlSession对象

SqlSession sqlSession =MybatisUtil.getSession();

//创建User对象，封装需要组合查询的条件

Student student =sqlSession.selectOne("com.Jorya.mapper.StudentMapper.findStudentById",1);

System.out.println(student.toString());

sqlSession.close();

}

@Test

public void findUserByNameAndJobs02() throws Exception{

//通过工具类生产SqlSession对象

SqlSession sqlSession =MybatisUtil.getSession();

//创建User对象，封装需要组合查询的条件

Student student =sqlSession.selectOne("com.Jorya.mapper.StudentMapper.findStudentById2",1);

System.out.println(student.toString());

sqlSession.close();

}

}

#### 6.2、mybatis 一对多映射关系

例如，一个班级有多个学生，即多个学生属于一个班级

MyBatis是怎么处理这种**一对多**关联关系的呢？**在< resultMap>元素中包含一个< collection>子元素**，MyBatis就是通过该元素来处理一对多关联关系的。

< collection>子元素的属性大部分与< collection>元素相同，但其还包含一个特殊属性—**ofType。ofType属性与javaType属性对应**，用于指定实体对象中集合类属性所包含的元素类型。

< collection>元素可以参考如下两种示例进行配置，具体代码如下

<!--方式一：嵌套查询-->

<collection property="studentList" column="id" ofType="com.Jorya.model.Student"

select ="com.Jorya.mapper.StudentMapper.selectStudent"/>

</collection>

<!--方式二：嵌套结果-->

<collection property="studentList" ofType="com.Jorya.model.Student">

<id property="id" column="student\_id"/>

<result property="username" column="username"/>

</collection>

**创建持久化类：班级类Banji和学生类Student—--------------一个班级多个学生**

public class Banji {

private Integer id;

private String name;

**private List<Student> studentList;**

public class Student {

private Integer id;

private String name;

private String sex;

**创建班级实体映射文件BanjiMapper.xml，并在文件中编写一对多关联映射查询的配置**

<?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.Jorya.mapper.BanjiMapper">

<!--一对多:查看某一班级及其关联的学生信息

注意:若关联查询出的列名相同，则需要使用别名区分-->

<select id="findBanjiwithStudent" parameterType="Integer"

resultMap="BanjiWithStudentResult">

select b.\*,s.id as student\_id,s.name

from otm\_banji b,otm\_student s

where b.id=s.banji\_id and b.id=#{id}

<!--

select b.\*,s.id as student id,s.name

from otm\_banji b,otm\_student s

where b.id=s.banji\_id and b.id=#{id}

-->

</select>

<resultMap type="Banji" id="BanjiWithStudentResult">

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

<result property="name" column="name" />

**<!--对多关联映射: collection**

**ofType表示属性集合中元素的类型List<student>属性，即Student类-->**

**<collection property="studentList" ofType="Student">**

<id property="id" column="student\_id" />

<result property="name" column="name" />

<result property="sex" column="sex" />

**</collection>**

</resultMap>

</mapper>

**测试**

public class MybatisTest {

@Test

public void findBanjiTest() throws Exception{

//通过工具类生产SqlSession对象

SqlSession sqlSession =MybatisUtil.getSession();

//创建User对象，封装需要组合查询的条件

Banji banji =sqlSession.selectOne("com.Jorya.mapper.BanjiMapper.findBanjiwithStudent",1);

System.out.println(banji.toString());

sqlSession.close();

}

}

#### 6.3、mybatis 多对多映射关系

多对多的关联关系是非常常见的。以**学生和课程**为例，一个学生可以选修多门课程，而一门课程又可以被多个学生选修，学生和课程就属于多对多的关联关系

在数据库中，多对多的关联关系通常使用一个**中间表**来维护，中间表选课表

**manytomany**

**创建持久化类课程类Course、Student**

public class Course {

private Integer id;

private String name;

private String code;

private List<Student> studentList;

public class Student {

private Integer id;

private String name;

private String sex;

private List<Course> courseList;

**创建课程实体映射文件CourseMapper.xml和学生实体映射文件StudentMapper.xml**

**3.31 CourseMapper.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.Jorya.mapper.CourseMapper">

<!-- **多对多嵌套查询**：通过执行一条SQL映射语句来返回预期的特殊类型 -->

<select id="findCourseWithStudent" parameterType="Integer"

resultMap="CourseWithStudentResult">

select \* from mtm\_course where id=#{id}

</select>

<resultMap type="Course" id="CourseWithStudentResult">

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

<result property="name" column="name"/>

<result property="code" column="code"/>

**<collection property="studentList" column="id" ofType="Student"**

**select="com.Jorya.mapper.StudentMapper.findStudentById">**

**</collection>**

</resultMap>

<!-- **多对多嵌套结果查询**：查询某课程极其关联的学生详情 -->

<select id="findCourseWithStudent2" parameterType="Integer"

resultMap="CourseWithStudentResult2">

select c.\*,s.id as sid,s.name

from mtm\_course c,otm\_student s,mtm\_electiveCourse ec

where ec.course\_id=s.id and c.id=#{id}

</select>

<resultMap type="Course" id="CourseWithStudentResult2">

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

<result property="name" column="name"/>

<result property="code" column="code"/>

**<collection property="studentList" ofType="Student">**

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

<result property="name" column="name"/>

</collection>

</resultMap>

</mapper>

使用嵌套查询的方式定义了一个id为findCourseWithStudent的select语句来查询课程及其关联的学生信息。在< resultMap>元素中使用了**< collection>元素来映射多对多**的关联关系，其中property属性表示订单持久化类中的课程属性，ofType属性表示集合中的数据为Student类型，而column的属性值会作为参数执行StudentMapper.xml中定义的id为findStudentById的执行语句来查询订单中的学生信息。

**3.32 StudentMapper.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.Jorya.mapper.StudentMapper">

<select id="**findStudentById**" parameterType="Integer" resultType="Student">

select \* from otm\_student where id in(

select student\_id from mtm\_electiveCourse where course\_id=#{id})

</select>

</mapper>

**测试**

在测试类MyBatisAssociatedTest中编写多对多关联查询的测试方法findCourseByIdTest()，其代码如下所示。

public class MybatisTest {

/\*\*

\* 多对多嵌套查询

\* @throws Exception

\*/

@Test

public void findBanjiTest() throws Exception{

//通过工具类生产SqlSession对象

SqlSession sqlSession =MybatisUtil.getSession();

//创建User对象，封装需要组合查询的条件

Course course=sqlSession.**selectOne**("com.Jorya.mapper.CourseMapper.findCourseWithStudent",1);

System.out.println(course.toString());

sqlSession.close();

}

@Test

public void findBanjiTest2() throws Exception{

//通过工具类生产SqlSession对象

SqlSession sqlSession =MybatisUtil.getSession();

//创建User对象，封装需要组合查询的条件

Course course=sqlSession.selectOne("com.Jorya.mapper.CourseMapper.findCourseWithStudent2",1);

System.out.println(course.toString());

sqlSession.close();

}

}

**七、Mybatis之Mapper接口的开发方式**

该方式开发，不需要写dao层的实现类，而是mybatis根据映射文件等信息对接口进行jdk动态代理生成代理类来实现接口中的方法，因此，采用这种方式，我们只需要编辑接口，而不需要去写实现。

**7.1.需求分析**

根据id查询用户。

**7.2.Mapper开发代理规范**

1、mapper接口的**全限定名**要和mapper映射文件的**namespace**值一致。

2、mapper接口的**方法名称**要和mapper映射文件的**statement的id**一致。

3、mapper接口的**方法参数**类型要和mapper映射文件的**statement的parameterType**的值一致，而且它的参数是一个。

4、mapper接口的方法**返回值类型**要和mapper映射文件的**statement的resultType**的值一致。

**7.3.代码实现**

准备po类：

public class User {

private Integer id;

private String username;

private Date birthday;

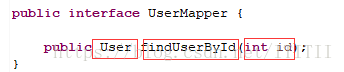
private String sex;

private String address;

//getter和setter方法省略

}

Mapper接口：



UserMapper.xml配置文件：



在全局配置文件SqlMapperConfig中添加该映射文件



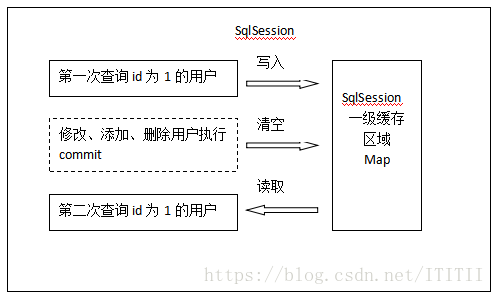
测试代码：



### 八、Mybatis一级缓存

mybatis提供查询缓存，如果缓存中有数据，就不用从数据库中获取，用于减轻数据压力，提高系统性能

**一级缓存**是**sqlSession级别**的缓存，在操作数据库的时候，需要构造sqlSession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据。不同的SqlSession的缓存区域（HashMap）是互相不受影响的。



**mybatis默认是支持一级缓存的。**

**8.1.证明一级缓存的存在：**

**验证方法：**

public void testOneLevelCache() {

SqlSession sqlSession = sqlSessionFactory.openSession();

//获取UserMapper的代理类

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

//第一次查询

User user = userMapper.findUserById(10);

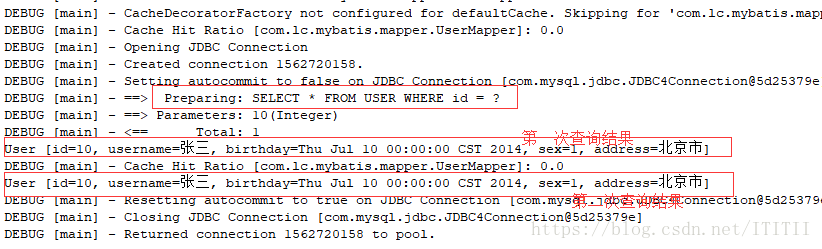
System.out.println(user);

//第二次查询

User user2 = userMapper.findUserById(10);

System.out.println(user2);

通过执行该方法，查看打印日志可以看出就执行了一次sql语句的查询，因此可以判断第二次查询是没有从数据库中进行查询的。



那当什么时候一级缓存会被清空呢？

通过测试发现，

当**在第二次查询数据库之前对数据库进行了写的操作后**，第二次查询就不会从缓存中查询结果，而是直接从数据中查询结果。

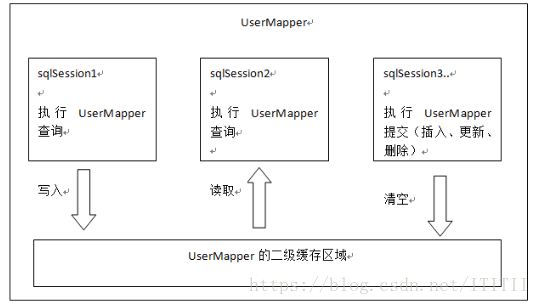
另一种就是在第一查询结果后，**手动的方式清空一级缓存**（使用sqlSession.clearCache();方法。）



### 九、Mybatis二级缓存

**二级缓存是Mapper级别的缓存**。多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是夸SqlSession的。

（**二级缓存Mybatis默认是关闭的**，需要自己去手动配置开启或可以自己选择用哪个厂家的缓存来作为二级缓存）



**9.1.开启二级缓存**

首先在全局配置文件中配置开启二级缓存功能：

**<!-- 开启二级缓存总开关 -->**

**<setting name="cacheEnabled" value="true"/>**

**在映射文件中去选择是否该映射文件使用二级缓存**：

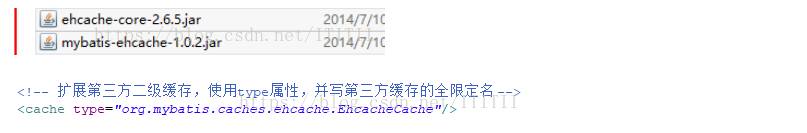
如果**使用**就进行以下配置，如果不是用，就不需要对映射文件做任何修改。

**<!-- 开启本mapper下的namespace的二级缓存，默认使用的是mybatis提供的PerpetualCache -->**

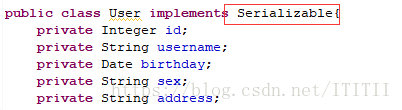
**<cache></cache>**

也可以设置选择二级缓存提工商

以下是我选择使用EhcacheCache缓存（注意：在使用EhcacheCache缓存需要导入jar包）



最后就是，在需要缓存的po类中去实现序列化：



验证二级缓存是否开启：

验证方法如下：

@Test

//测试二级缓存是否开启

public void testTwoLevelCache() {

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

//获取UserMapper的代理类

UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);

UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);

UserMapper userMapper3 = sqlSession3.getMapper(UserMapper.class);

User user1 = userMapper1.findUserById(10);

System.out.println(user1);

//当sqlSession1执行close()的时候，才把sqlSession1查询的结果写到二级缓存中去。

sqlSession1.close();

// User u = new User();

// u.setUsername("小胖");

// u.setAddress("成都");

// u.setSex("2");

// u.setBirthday(new Date());

// userMapper3.insertUser(u);

// //当其中任何一个sqlSession执行commit()方法后将刷新整个缓存区

// sqlSession3.commit();

// sqlSession3.close();

//第二次查询

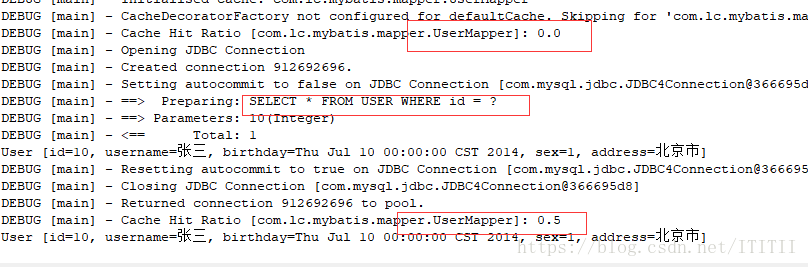
User user2 = userMapper2.findUserById(10);

System.out.println(user2);

sqlSession2.close();

}

测试结果：

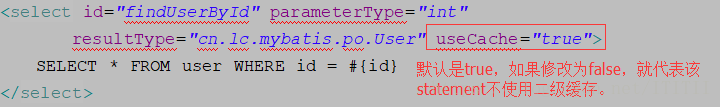


同时通过测试，当在第二次查询的时候，向数据库提交数据后，就会对缓存进行刷新，这样在第二次查询的时候就没有走缓存，而是走的数据库。

**9.2.禁用二级缓存**

由于二级缓存默认是关闭的，如果不开启就不会使用二级缓存。如果，我们开启了二级缓存，而在有些查询结果集中，不需要受到二级缓存影响，该怎么去做呢？

在select查询中，默认是使用了二级缓存，如果不想使用二级缓存，就在select标签中有一个useCache的属性设置为false，就代表不使用二级缓存，每次进行查询数据都不会从缓存总获取，而是直接从数据库中进行查询。**useCache的默认值是true，即代表statement使用二级缓存**。



**9.3.刷新二级缓存**

在statement中设置flushCache=true，可以刷新二级缓存。默认情况下，select语句中的flushCache是false。如果是insert、update、delete语句，那么flushCache的默认值是true。如果将select语句中的flushCache值改为true，就意味着查询语句的二级缓存失效，每次查询都会从数据库进行查询。如果将select语句的flushCache值为false，就代表该查询语句使用了二级缓存，如果在数据库中修改了数据，而二级缓存中的数据还是原来的数据，那么这样就会出现脏读。

flushCache设置如下：

