**Mybatis 框架课程第三天** 第**1**章 **Mybatis** 连接池与事务深入 **12**

# 1.1 Mybatis 的连接池技术

我们在前面的WEB课程中也学习过类似的连接池技术，而在Mybatis中也有连接池技术，但是它采用的是自

己的连接池技术。在Mybatis的SqlMapConfig.xml配置文件中，通过<dataSource type=”pooled”>来实现Mybatis中连接池的配置。

### 1.1.1 Mybatis 连接池的分类

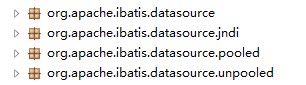
在

Mybatis

中我们将它的数据源

dataSource

分为以下几类：



可以看出

Mybatis

将它自己的数据源分为三

类：

UNPOOLED

不使用连接池的数据源

POOLED

使用连接池的数据源

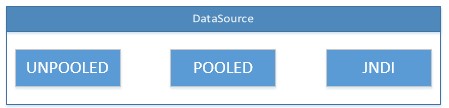
JNDI

使用

JNDI

实现的数据源

具体结构如下：



相应地，

MyBatis

内部分别定义了实现了

java.sql.DataSource

接口的

UnpooledDataSource

，

PooledDataSource

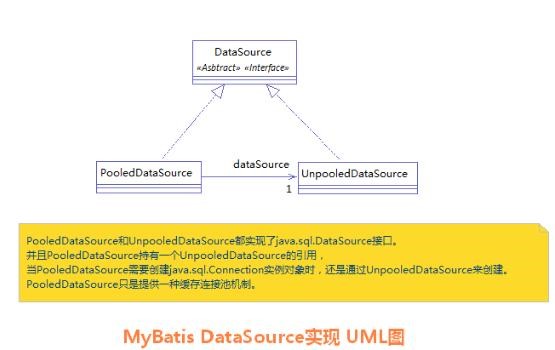
类来表示

UNPOOLED

、

POOLED

类型的数据源。



在这三种数据源中，我们一般采用的是

POOLED

数据源（很多时候我们所说的数据源就是为了更好的管理数据

库连接，也就是我们所

说的连接池技术）

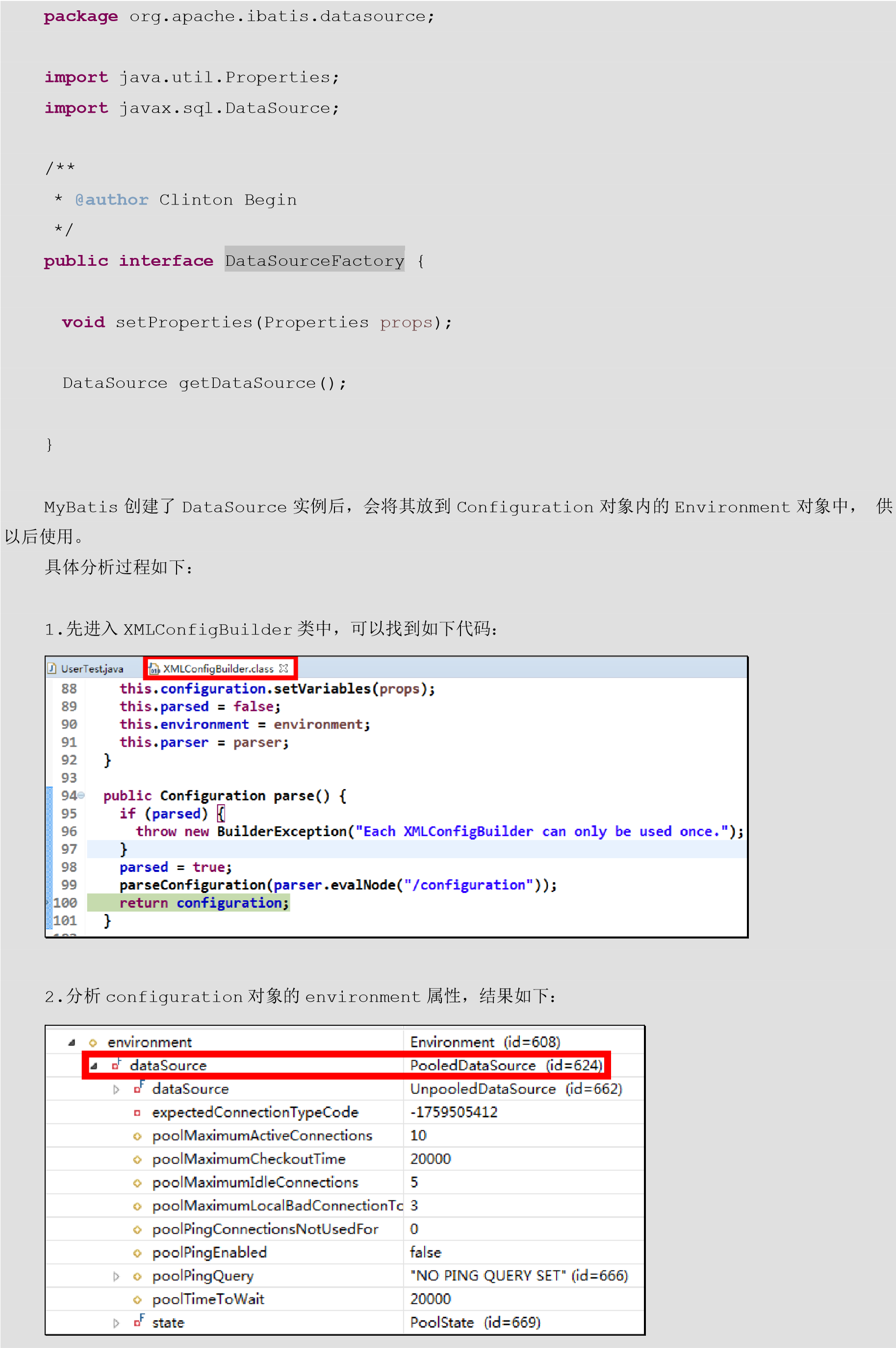
。

### 1.1.2 Mybatis 中数据源的配置

|  |
| --- |
| 我们的数据源配置就是在SqlMapConfig.xml文件中，具体配置如下：  <!-- 配置数据源（连接池）信息 -->  <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>  MyBatis在初始化时，根据<dataSource>的type属性来创建相应类型的的数据源DataSource，即： type=”POOLED”：MyBatis会创建PooledDataSource实例 type=”UNPOOLED” ： MyBatis会创建UnpooledDataSource实例 type=”JNDI”：MyBatis会从JNDI服务上查找DataSource实例，然后返回使用 |

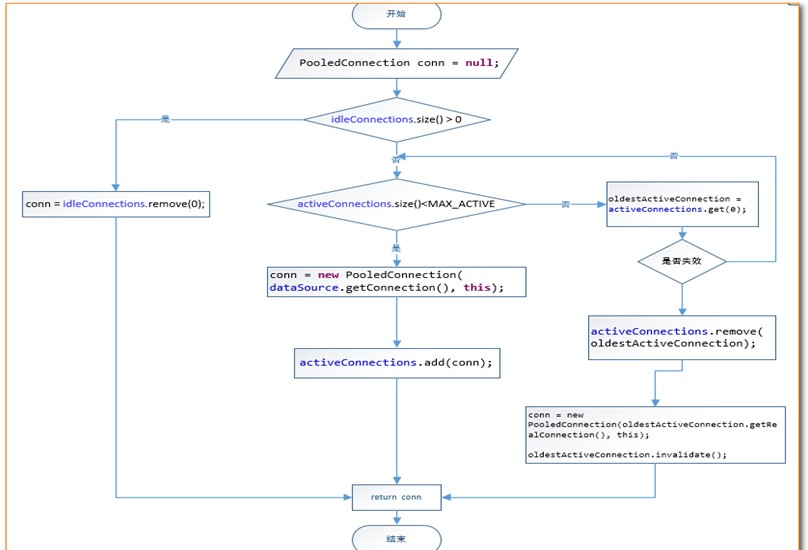
### 1.1.3 Mybatis 中 DataSource 的存取

|  |
| --- |
| MyBatis 是通过工厂模式来创建数据源 DataSource 对象的，MyBatis 定义了抽象的工厂接口:org.apache.ibatis.datasource.DataSourceFactory,通过其 getDataSource()方法返回数据源 DataSource。    下面是**DataSourceFactory**源码，具体如下： |



### 1.1.4 Mybatis 中连接的获取过程分析

|  |  |
| --- | --- |
| 当我们需要创建SqlSession对象并需要执行SQL语句时，这时候MyBatis才会去调用dataSource对象来创建java.sql.Connection对象。也就是说，java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。  @Test  public void testSql() throws Exception {  InputStream in = Resources.getResourceAsStream("SqlMapConfig.xml");  SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);  SqlSession sqlSession = factory.openSession();  List<User> list = sqlSession.selectList("findUserById",41);  System.out.println(list.size());  }  只有当第4句sqlSession.selectList("findUserById")，才会触发MyBatis在底层执行下面这个方  法来创建java.sql.Connection对象。 如何证明它的加载过程呢？  我们可以通过断点调试，在PooledDataSource中找到如下popConnection()方法，如下所示： | |
|  |  |
| 分析源代码，得出PooledDataSource工作原理如下： |



下面是连接获取的源代码：



最后我们可以发现，真正连接打开的时间点，只是在我们执行

SQL

语句时，才会进行。其实这

样做我们也可以

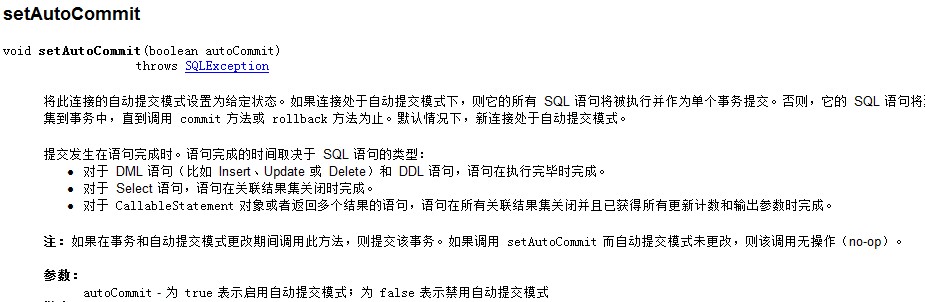
进一步发现，数据库连接是我们最为宝贵的资源，只有在要用到的时候，才去获取并打开连接，当我们用完了就再

立即将数据库连接归还到连接池中。

# 1.2 Mybatis 的事务控制

### 1.2.1 JDBC 中事务的回顾

在JDBC中我们可以通过手动方式将事务的提交改为手动方式，通过setAutoCommit()方法就可以调整。 通过JDK文档，我们找到该方法如下：



那么我们的

Mybatis

框架因为是对

JDBC

的封装，所以

Mybatis

框架的事务控制方式，本身也是用

JDBC

的

setAutoCommit()

方法来设置事务提交方式的

。

### 1.2.2 Mybatis 中事务提交方式



|  |
| --- |
| session.commit();  //8.释放资源  session.close(); in.close();  }  观察在它在控制台输出的结果： |
|  |
| 这是我们的Connection的整个变化过程，通过分析我们能够发现之前的CUD操作过程中，我们都要手动进行事务的提交，原因是setAutoCommit()方法，在执行时它的值被设置为false了，所以我们在CUD操作中，必须通过sqlSession.commit()方法来执行提交操作。 |

### 1.2.3 Mybatis 自动提交事务的设置

通过上面的研究和分析，现在我们一起思考，为什么

CUD

过程中必须使用

sql

Session.commit()

提交事

务？主要原因就是在连接池中取出的连接，都会将调用

connection.setAutoCommit(false)

方法，这样我们

就必须使用

sqlSession.commit()

方法

，相当于使用了

JDBC

中的

connection.commit()

方法实现事务提

交

。

明白这一点后，我们现在一起尝试不进行手动提交，一样实现

CUD

操作。

@Before

//

在测试方法执行之前执行

**public**

**void**

i

nit()

**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(**

**true**

**;**

**)**

//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{

//7.

释放资源

|  |  |
| --- | --- |
| session.close(); in.close();  }  所对应的DefaultSqlSessionFactory类的源代码：    运行的结果如下： | |
|  |  |
| 我们发现，此时事务就设置为自动提交了，同样可以实现CUD 操作时记录的保存。虽然这也是一种方式，但就编程而言，设置为自动提交方式为false 再根据情况决定是否进行提交，这种方式更常用。因为我们可以根据业务情况来决定提交是否进行提交。 |

第**2**章 **Mybatis** 的动态 **SQL** 语句

|  |  |
| --- | --- |
| Mybatis的映射文件中，前面我们的SQL都是比较简单的，有些时候业务逻辑复杂时，我们的SQL是动态变化的，此时在前面的学习中我们的SQL就不能满足要求了。 参考的官方文档，描述如下： | |
|  |  |

# 2.1 动态 SQL 之<if>标签

我们根据实体类的不同取值，使用不同的SQL语句来进行查询。比如在id如果不为空时可以根据id查询，

如果username不同空时还要加入用户名作为条件。这种情况在我们的多条件组合查询中经常会碰到。

### 2.1.1 持久层 Dao 接口

/\*\*

* 根据用户信息，查询用户列表
* **@param** user
* **@return**

\*/

List<User> findByUser(User user);

### 2.1.2 持久层 Dao 映射配置

|  |  |  |
| --- | --- | --- |
| <              < | select | id=*"findByUser"* resultType=*"user"* parameterType=*"user"*>  select \* from user where 1=1 test=*"username!=null and username != '' "*>  and username like #{username}  > test=*"address != null"*>  and address like #{address}  >    >  >标签的test属性中写的是对象的属性名，如果是包装类的对象要使用OGNL表达式的写法。  where 1=1 的作用~！ |
| <if    </ if  <if    </ if    / select 注意：< if 另外要注意 |

### 2.1.3 测试

|  |
| --- |
| @Test  **public** **void** testFindByUser() {  User u = **new** User();  u.setUsername("%王%");  u.setAddress("%顺义%");  //6.执行操作  List<User> users = userDao.findByUser(u); **for**(User user : users) {  System.***out***.println(user);  }  } |

# 2.2 动态 SQL 之<where>标签

为了简化上面where 1=1的条件拼装，我们可以采用<where>标签来简化开发。

### 2.2.1 持久层 Dao 映射配置

<!

--

根据用户信息查询

--

>

<

select

id

=

*"findByUser"*

resultType

=

*"user"*

parameterType

=

*"user"*

>

<

include

refid

=

*"defaultSql"*

/

><

include

>

<

where

>

<

if

test

=

*"username!=null and username != '' "*

>

and

username

like

#{

username

}

<

/

if

>

<

if

test

=

*"address != null"*

>

and address like #{address}

<

/

if

>

<

/

where

>

<

/

select

>

# 2.3 动态标签之<foreach>标签

### 2.3.1 需求

传入多个id查询用户信息，用下边两个sql实现：

SELECT \* FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)

SELECT \* FROM USERS WHERE username LIKE '%张%' AND id IN (10,89,16)

这样我们在进行范围查询时，就要将一个集合中的值，作为参数动态添加进来。 这样我们将如何进行参数的传递？

##### 2.3.1.1 在 QueryVo 中加入一个 List 集合用于封装参数

/\*\*

\*

\*

<

p

>

Title: QueryVo

<

/p

>

\*

<

p

>

Description:

查询的条件

/p

<

>

\*

<

p

>

Company: http://www.itheima.com/

<

/p

>

\*/

**public**

**class**

Quer

yVo

**implements**

Serializable {

**private**

List<Integer>

ids

;

**public** List<Integer> getIds() { **return** ids;

}

**public** **void** setIds(List<Integer> ids) { **this**.ids = ids;

}

}

### 2.3.2 持久层 Dao 接口

/\*\*

* 根据id集合查询用户
* **@param** vo
* **@return**

\*/

List<User> findInIds(QueryVo vo);

### 2.3.3 持久层 Dao 映射配置

<!

--

查询所有用户在

id

的集合之中

--

>

<

select

id

=

*"findInIds"*

resultType

=

*"user"*

parameterType

=

*"queryvo"*

>

<!

--

select \* from user where id in (1,2,3,4,5);

--

>

<

include

refid

=

*"defaultSql"*

><

/

include

>

<

where

>

<

if

test

=

*"ids != null and*

*ids.size() > 0"*

>

<

foreach

collection

=

*"ids"*

open

=

*in*

*"id*

*(*

*"*

close

=

*")"*

item

=

*"uid"*

separator

=

*","*

>

#{

uid

}

/

<

foreach

>

<

/

if

>

<

/

where

>

<

/

select

>

SQL

语句：

s

elect

字段

from user where id in (?

)

foreach

<

>

标签用于遍历集合，它的属性：

collection:

代表要遍历的集合元素

，注意编写时

不要写

#{}

open:

代表语句的开始部分

close:

代表结束部分

item:代表遍历集合的每个元素，生成的变量名 sperator:代表分隔符

##### 2.3.3.1 编写测试方法

|  |
| --- |
| @Test  **public** **void** testFindInIds() {  QueryVo vo = **new** QueryVo();  List<Integer> ids = **new** ArrayList<Integer>(); ids.add(41); ids.add(42); ids.add(43); ids.add(46); ids.add(57); vo.setIds(ids);  //6.执行操作  List<User> users = userDao.findInIds(vo); **for**(User user : users) {  System.***out***.println(user);  }  } |

# 2.4 Mybatis 中简化编写的 SQL 片段

Sql中可将重复的sql提取出来，使用时用include引用即可，最终达到sql重用的目的。

#### **2.4.1** 定义代码片段

|  |
| --- |
| <!-- 抽取重复的语句代码片段 -->  <sql id=*"defaultSql"*> select \* from user  </sql> |

#### **2.4.2** 引用代码片段

<!-- 配置查询所有操作 -->

<select id=*"findAll"* resultType=*"user"*>

<include refid=*"defaultSql"*></include>

</select>

<!-- 根据id查询 -->

<select id=*"findById"* resultType=*"UsEr"* parameterType=*"int"*>

<include refid=*"defaultSql"*></include> where id = #{uid} </select>

第3章 **Mybatis** 多表查询之一对多

**3.1** 一对一查询**(**多对一**)**

本次案例主要以最为简单的用户和账户的模型来分析

Mybatis

多表关系。用户为

User

表，账户为

Account

表。一个用户（

User

）可以有多个账户（

Account

）。具体关系如下：

FK\_Reference\_8



用户表

id

用户名

性别

出生日期

地址

...

int

varchar(20)

varchar(10)

date

varchar(60)

>

<

pk



账户

编号

用户编号

金额

...

int

int

double

>

pk

<

<

fk

>

|  |
| --- |
| 需求  查询所有账户信息，关联查询下单用户信息。 注意： 因为一个账户信息只能供某个用户使用，所以从查询账户信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的账户信息则为一对多查询，因为一个用户可以有多个账户。 |

### 3.1.1 方式一

###### **3.1.1.1** 定义账户信息的实体类

/\*\*

\*

\*

>

p

<

Title: Account

>

/p

<

\*

<

p

>

Description:

账户的实体类

>

<

/p

\*

>

<

p

Company: http://www.itheima.com/

<

>

/p

\*/

**public**

**class**

Account

**implements**

Serializable {

**private**

Integer

id

;

**private**

Integer

uid

;

**private**

Double

money

;

**public** Integer getId() { **return** id;

|  |
| --- |
| }  **public** **void** setId(Integer id) { **this**.id = id;  }  **public** Integer getUid() { **return** uid;  }  **public** **void** setUid(Integer uid) { **this**.uid = uid;  }  **public** Double getMoney() { **return** money;  }  **public** **void** setMoney(Double money) { **this**.money = money;  }  @Override  **public** String toString() {  **return** "Account [id=" + id + ", uid=" + uid + ", money=" + money + "]";  }  } |

**3.1.1.2**

编写

**S**

**ql**

语句

实现查询账户信息时，也要查询账户所对应的用户信息

。

SELECT

account

.\*,

user.username,

user.address

FROM

account

,

user

WHERE

account

.u

id = user.id

在

MySQL

中测试的查询结果如下：



##### 3.1.1.3 定义 AccountUser 类

为了能够封装上面SQL语句的查询结果，定义 AccountCustomer类中要包含账户信息同时还要包含用户信

息，所以我们要在定义AccountUser类时可以继承User类。

/\*\*

\*

\*

p

>

<

Title: AccountUser

/p

>

<

\*

>

<

p

Description:

它是

account

的子类

/p

<

>

\*

<

p

>

Company: http://www.itheima.com/

<

/p

>

\*/

**public**

**class**

AccountUser

**extends**

Account

**implements**

Serializable {

**private**

String

username

;

**private**

String

address

;

**public**

String getUsername() {

**return**

username

;

}

**public**

**void**

setUsername(String

username

)

{

**this**

.

username

=

username

;

}

**public**

String getAddress() {

**return**

address

;

}

**public**

**void**

setAddress(String

address

{

)

**this**

.

address

=

address

;

}

@Override

**p**

**ublic**

String toString() {

**return**

**super**

.toString() +

" AccountUser [username="

+

username

+

",

address="

+

address

+

"]"

;

}

}

###### **3.1.1.4** 定义账户的持久层 **Dao** 接口

/\*\*

* <p>Title: IAccountDao</p>
* <p>Description: 账户的持久层接口</p>
* <p>Company: http://www.itheima.com/ </p>

\*/

**public** **interface** IAccountDao {

/\*\*

* 查询所有账户，同时获取账户的所属用户名称以及它的地址信息
* **@return**

\*/

List<AccountUser> findAll();

}

##### 3.1.1.5 定义 AccountDao.xml 文件中的查询配置信息

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace=*"com.itheima.dao.IAccountDao"*>  <!-- 配置查询所有操作-->  <select id=*"findAll"* resultType=*"accountuser"*>  select a.\*,u.username,u.address from account a,user u where a.uid =u.id; </select> </mapper>    注意：因为上面查询的结果中包含了账户信息同时还包含了用户信息，所以我们的返回值类型 returnType 的值设置为AccountUser类型，这样就可以接收账户信息和用户信息了。 |

##### 3.1.1.6 创建 AccountTest 测试类

/\*\*

\*

\*

p

<

>

Title: MybastisCRUDTest

/p

<

>

\*

>

p

<

Description:

一对多账户的操作

>

/p

<

\*

<

p

>

Company: http://www.itheima.com/

>

/p

<

\*/

**public**

**class**

AccountTest {

**private**

InputStream

in

;

**private**

SqlSessionFactory

factory

;

**private**

SqlSession

session

;

**private**

IAccountDao

accou

ntDao

;

@Test

**public**

**void**

testFindAll() {

//6.

执行操作

List<AccountUser>

accountusers

=

accountDao

.findAll()

;

**for**

AccountUser

(

au

:

accountusers

)

{

System.

***out***

.println(

au

)

;

}

}

@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

.openSessi

on();

//5.

创建

Dao

的代理对象

accountDao

=

session

.getMapper(IAccountDao.

**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();

}

}

**3.1.1.7**

小结：

定义专门的 po 类作为输出类型，其中定义了 sql 查询结果集所有的字段。此方法较为简单，企业中使用普

遍。

### 3.1.2 方式二

|  |
| --- |
| 使用resultMap，定义专门的resultMap用于映射一对一查询结果。  通过面向对象的(has a)关系可以得知，我们可以在Account类中加入一个User类的对象来代表这个账户是哪个用户的。 |

**3.1.2.1** 修改 **Account** 类在 Account 类中加入 User 类的对象作为 Account 类的一个属性。

/\*\*

* <p>Title: Account</p>
* <p>Description: 账户的实体类</p>
* <p>Company: http://www.itheima.com/ </p>

\*/

**public** **class** Account **implements** Serializable {

|  |
| --- |
| **private** Integer id; **private** Integer uid; **private** Double money;    **private** User user;    **public** User getUser() { **return** user;  }  **public** **void** setUser(User user) { **this**.user = user;  }    **public** Integer getId() { **return** id;  }  **public** **void** setId(Integer id) { **this**.id = id;  }  **public** Integer getUid() { **return** uid;  }  **public** **void** setUid(Integer uid) { **this**.uid = uid;  }  **public** Double getMoney() { **return** money;  }  **public** **void** setMoney(Double money) { **this**.money = money;  }  @Override  **public** String toString() {  **return** "Account [id=" + id + ", uid=" + uid + ", money=" + money + "]";  }  } |

##### 3.1.2.2 修改 AccountDao 接口中的方法

/\*\*

* <p>Title: IAccountDao</p>
* <p>Description: 账户的持久层接口</p>
* <p>Company: http://www.itheima.com/ </p>

\*/

**public** **interface** IAccountDao {

/\*\*

* 查询所有账户，同时获取账户的所属用户名称以及它的地址信息
* **@return**

\*/

List<Account> findAll();

}

注意：第二种方式，将返回值改 为了Account类型。

因为Account类中包含了一个User类的对象，它可以封装账户所对应的用户信息。

##### 3.1.2.3 重新定义 AccountDao.xml 文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace=*"com.itheima.dao.IAccountDao"*>    <!-- 建立对应关系 -->  <resultMap type=*"account"* id=*"accountMap"*>  <id column=*"aid"* property=*"id"*/>  <result column=*"uid"* property=*"uid"*/>  <result column=*"money"* property=*"money"*/>  <!-- 它是用于指定从表方的引用实体属性的 -->  <association property=*"user"* javaType=*"user"*>  <id column=*"id"* property=*"id"*/>  <result column=*"username"* property=*"username"*/>  <result column=*"sex"* property=*"sex"*/>  <result column=*"birthday"* property=*"birthday"*/>  <result column=*"address"* property=*"address"*/>  </association>  </resultMap>    <select id=*"findAll"* resultMap=*"accountMap"*>  select u.\*,a.id as aid,a.uid,a.money from account a,user u where a.uid =u.id; </select>  </mapper> |

##### 3.1.2.4 在 AccountTest 类中加入测试方法

|  |
| --- |
| @Test  **public** **void** testFindAll() {  List<Account> accounts = accountDao.findAll(); **for**(Account au : accounts) {  System.***out***.println(au);  System.***out***.println(au.getUser());  }  } |

## **3.2** 一对多查询

需求： 查询所有用户信息及用户关联的账户信息。 分析： 用户信息和他的账户信息为一对多关系，并且查询过程中如果用户没有账户信息，此时也要将用户信息

查询出来，我们想到了左外连接查询比较合适。

### 3.2.1 编写 SQL 语句

SELECT

u.\*, acc.id id,

acc.uid,

acc.

money

FROM

user u

LEFT JOIN account acc ON u.id = acc.uid

测试该

SQL

语句在

MySQL

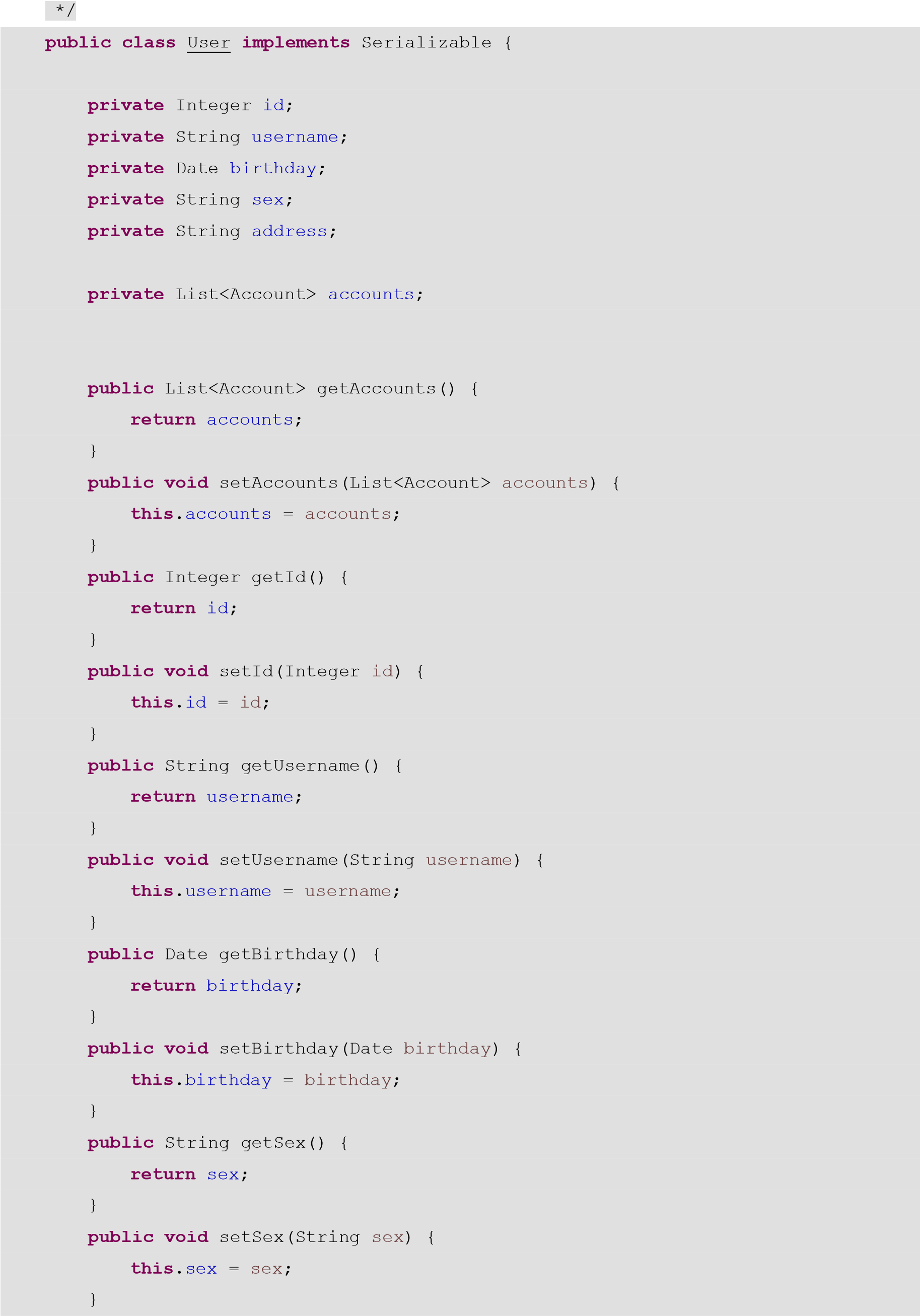
客户端工具的查询结果如下：



### 3.2.2 User 类加入 List<Account>

/\*\*

* <p>Title: User</p>
* <p>Description: 用户的实体类</p> \* <p>Company: http://www.itheima.com/ </p>



**public** String getAddress() { **return** address;

}

|  |
| --- |
| **public** **void** setAddress(String address) { **this**.address = address;  }  @Override  **public** String toString() {  **return** "User [id=" + id + ", username=" + username + ", birthday=" + birthday  + ", sex=" + sex + ", address="  + address + "]";  }  } |

#### **3.2.3** 用户持久层 **Dao** 接口中加入查询方法

/\*\*

* 查询所有用户，同时获取出每个用户下的所有账户信息
* **@return**

\*/

List<User> findAll();

#### **3.2.4** 用户持久层 **Dao** 映射文件配置

<?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.itheima.dao.IUserDao"*>

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

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

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

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

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

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

<!-- collection是用于建立一对多中集合属性的对应关系 ofType用于指定集合元素的数据类型

-->

<collection property=*"accounts"* ofType=*"account"*>

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

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

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

</collection>

</resultMap>

|  |
| --- |
| <!-- 配置查询所有操作 -->  <select id=*"findAll"* resultMap=*"userMap"*>  select u.\*,a.id as aid ,a.uid,a.money from user u left outer join account  a on u.id =a.uid  </select>  </mapper> **collection**  部分定义了用户关联的账户信息。表示关联查询结果集  property=*"accList"*：  关联查询的结果集存储在User对象的上哪个属性。  ofType=*"account"*：  指定关联查询的结果集中的对象类型即List中的对象类型。此处可以使用别名，也可以使用全限定名。 |

### 3.2.5 测试方法

|  |
| --- |
| /\*\*   * <p>Title: MybastisCRUDTest</p> * <p>Description: 一对多的操作</p> * <p>Company: http://www.itheima.com/ </p>   \*/  **public** **class** UserTest {    **private** InputStream in ; **private** SqlSessionFactory factory; **private** SqlSession session; **private** IUserDao userDao;    @Test  **public** **void** testFindAll() {  //6.执行操作  List<User> users = userDao.findAll(); **for**(User user : users) {  System.***out***.println("-------每个用户的内容---------");  System.***out***.println(user);  System.***out***.println(user.getAccounts());  }  }      @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();

}

}

第**4**章 **Mybatis** 多表查询之多对多

# 4.1 实现 Role 到 User 多对多

通过前面的学习，我们使用 Mybatis 实现一对多关系的维护。多对多关系其实我们看成是双向的一对多关

系。

#### **4.1.1** 用户与角色的关系模型

用户与角色的多对多关系模型如下：

FK\_Reference\_9

FK\_Reference\_10



用户表

id

用户名

性别

出生日期

地址

...

int

varchar(20)

varchar(10)

date

varchar(60)

<

pk

>



用户角色

用户编号

角色编号

int

int

1>

pk,fk

<

<

2>

pk,fk



角色表

编号

角色名称

角色描述

...

int

varchar(30)

varchar(60)

<

pk

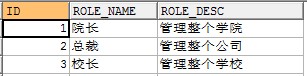
>

在

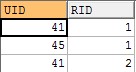
MySQL

数据库中添加角色表，用户角色的中间表。

角色表



用户角色中间表



### 4.1.2 业务要求及实现 SQL

|  |
| --- |
| 需求：  实现查询所有对象并且加载它所分配的用户信息。 分析：  查询角色我们需要用到Role表，但角色分配的用户的信息我们并不能直接找到用户信息，而是要通过中  间表(USER\_ROLE表)才能关联到用户信息。 下面是实现的SQL语句：  SELECT  r.\*,u.id uid,  u.username username,  u.birthday birthday,  u.sex sex,  u.address address  FROM  ROLE r  INNER JOIN  USER\_ROLE ur  ON ( r.id = ur.rid)  INNER JOIN  USER u  ON (ur.uid = u.id); |

#### **4.1.3** 编写角色实体类

/\*\*

\*

**@author**

黑马程序员

\*

**@Company**

http://www.ithiema.com

\*/

**public**

**class**

Role

**implements**

Serializable {

**private**

Integer

roleId

;

**private**

String

roleName

;

**private** String roleDesc;

|  |
| --- |
| //多对多的关系映射：一个角色可以赋予多个用户  **private** List<User> users;    **public** List<User> getUsers() { **return** users;  }  **public** **void** setUsers(List<User> users) { **this**.users = users;  }  **public** Integer getRoleId() { **return** roleId;  }  **public** **void** setRoleId(Integer roleId) { **this**.roleId = roleId;  }  **public** String getRoleName() { **return** roleName;  }  **public** **void** setRoleName(String roleName) { **this**.roleName = roleName;  }  **public** String getRoleDesc() { **return** roleDesc;  }  **public** **void** setRoleDesc(String roleDesc) { **this**.roleDesc = roleDesc;  }    @Override  **public** String toString() { **return** "Role{" +  "roleId=" + roleId +  ", roleName='" + roleName + '\'' + ", roleDesc='" + roleDesc + '\'' +  '}'; |

}

}

### 4.1.4 编写 Role 持久层接口

/\*\*

* **@author** 黑马程序员
* **@Company** http://www.ithiema.com

\*/ **public** **interface** IRoleDao {

/\*\*

* 查询所有角色
* **@return**

\*/

List<Role> findAll();

}

#### **4.1.5** 编写映射文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace=*"com.itheima.dao.IRoleDao"*>    <!--定义role表的ResultMap-->  <resultMap id=*"roleMap"* type=*"role"*>  <id property=*"roleId"* column=*"rid"*></id>  <result property=*"roleName"* column=*"role\_name"*></result> <result property=*"roleDesc"* column=*"role\_desc"*></result>  <collection property=*"users"* ofType=*"user"*>  <id column=*"id"* property=*"id"*></id>  <result column=*"username"* property=*"username"*></result>  <result column=*"address"* property=*"address"*></result>  <result column=*"sex"* property=*"sex"*></result>  <result column=*"birthday"* property=*"birthday"*></result>  </collection>  </resultMap>    <!--查询所有-->  <select id=*"findAll"* resultMap=*"roleMap"*>  select u.\*,r.id as rid,r.role\_name,r.role\_desc from role r |

left outer join user\_role ur on r.id = ur.rid

left outer join user u on u.id = ur.uid

</select>

</mapper>

#### **4.1.6** 编写测试类



**for**(Role role : roles){

System.***out***.println("---每个角色的信息----");

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

System.***out***.println(role.getUsers());

}

}

}

# 4.2 实现 User 到 Role 的多对多

### 4.2.1 User 到 Role 的多对多

从User出发，我们也可以发现一个用户可以具有多个角色，这样用户到角色的关系也还是一对多关系。这样

我们就可以认为User与Role的多对多关系，可以被拆解成两个一对多关系来实现。

### 4.2.2 作业：实现 User 到 Role 的一对多查询

需求：实现查询所有用户信息并关联查询出每个用户的角色列表。