



# Mybatis 框架课程第三天

## 第1章 Mybatis 连接池与事务深入

### 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

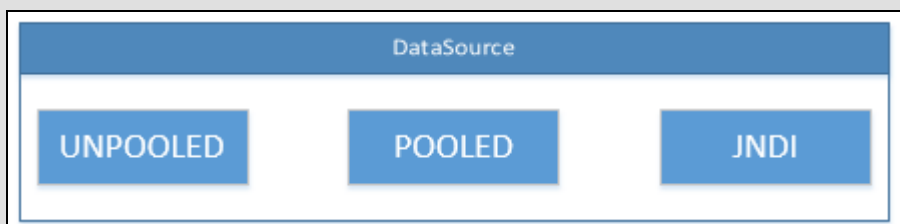
在 Mybatis 中我们将它的数据源 dataSource 分为以下几类：

- ▶ `org.apache.ibatis.datasource`
- ▶ `org.apache.ibatis.datasource.jndi`
- ▶ `org.apache.ibatis.datasource.pooled`
- ▶ `org.apache.ibatis.datasource.unpooled`

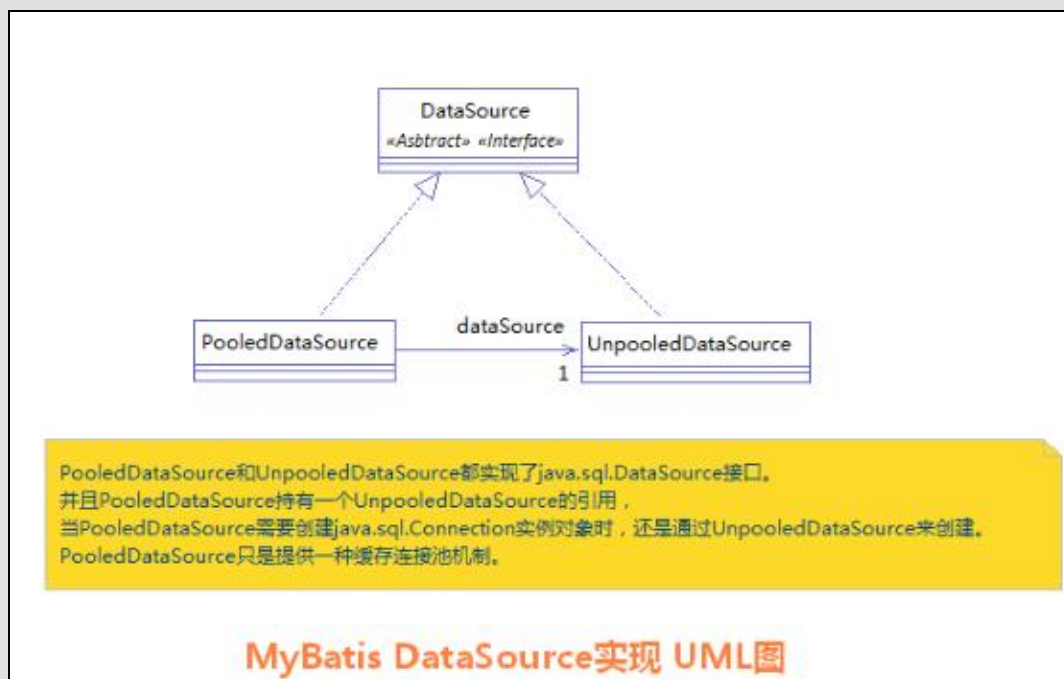
可以看出 Mybatis 将它自己的数据源分为三类：

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：



```
package org.apache.ibatis.datasource;

import java.util.Properties;
import javax.sql.DataSource;

/**
 * @author Clinton Begin
 */
public interface DataSourceFactory {

    void setProperties(Properties props);

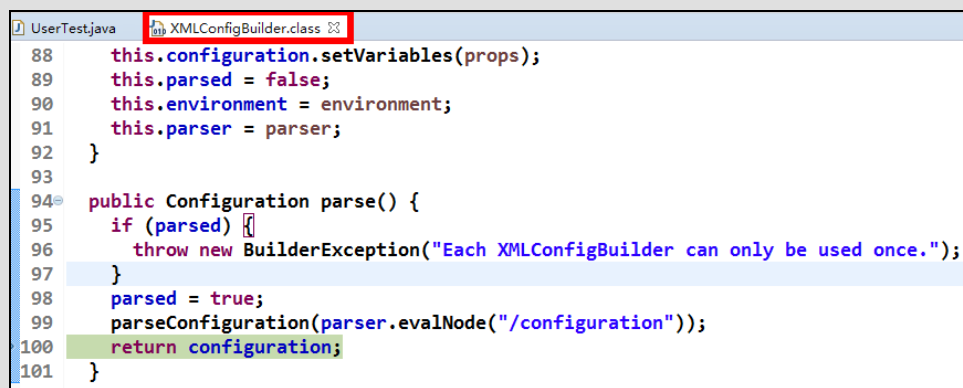
    DataSource getDataSource();

}
```

MyBatis 创建了 DataSource 实例后，会将其放到 Configuration 对象内的 Environment 对象中，供以后使用。

具体分析过程如下：

1. 先进入 XMLConfigBuilder 类中，可以找到如下代码：



```
88     this.configuration.setVariables(props);
89     this.parsed = false;
90     this.environment = environment;
91     this.parser = parser;
92 }
93
94 public Configuration parse() {
95     if (parsed) {
96         throw new BuilderException("Each XMLConfigBuilder can only be used once.");
97     }
98     parsed = true;
99     parseConfiguration(parser.evalNode("/configuration"));
100     return configuration;
101 }
```

2. 分析 configuration 对象的 environment 属性，结果如下：

environment	Environment (id=608)
dataSource	PooledDataSource (id=624)
dataSource	UnpooledDataSource (id=662)
expectedConnectionTypeCode	-1759505412
poolMaximumActiveConnections	10
poolMaximumCheckoutTime	20000
poolMaximumIdleConnections	5
poolMaximumLocalBadConnectionTc	3
poolPingConnectionsNotUsedFor	0
poolPingEnabled	false
poolPingQuery	"NO PING QUERY SET" (id=666)
poolTimeToWait	20000
state	PoolState (id=669)



## 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。

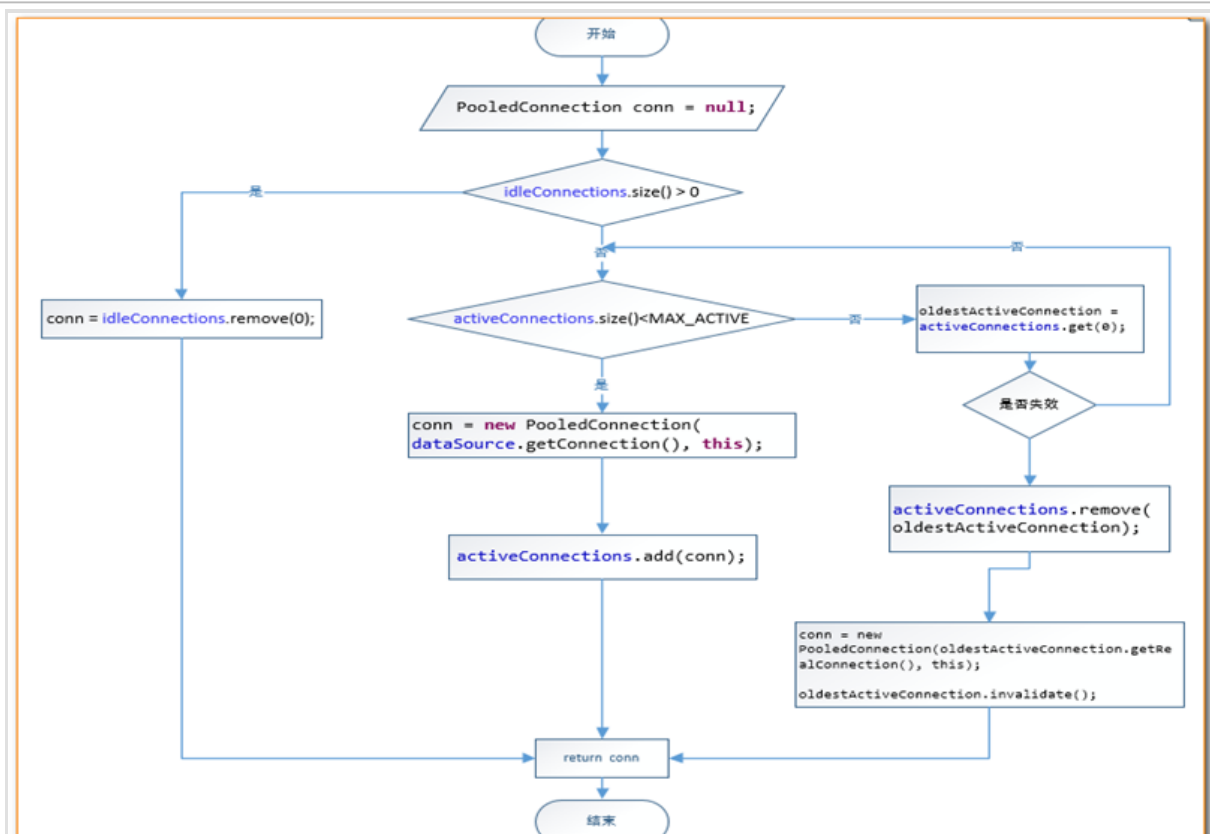
如何证明它的加载过程呢？

我们可以通过断点调试，在 `PooledDataSource` 中找到如下 `popConnection()` 方法，如下所示：

```
private PooledConnection popConnection(String username, String password) throws SQLException {
    boolean countedWait = false;
    PooledConnection conn = null;
    long t = System.currentTimeMillis();
    int localBadConnectionCount = 0;

    while (conn == null) {
        synchronized (state) {
            if (!state.idleConnections.isEmpty()) {
                // Pool has available connection
                conn = state.idleConnections.remove(0);
                if (Log.isDebugEnabled()) {
                    Log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");
                }
            } else {
                // Pool does not have available connection
                if (state.activeConnections.size() < poolMaximumActiveConnections) {
                    // Can create new connection
                    conn = new PooledConnection(dataSource.getConnection(), this);
                    if (Log.isDebugEnabled()) {
                        Log.debug("Created connection " + conn.getRealHashCode() + ".");
                    }
                } else {
                    // Cannot create new connection
                    PooledConnection oldestActiveConnection = state.activeConnections.get(0);
                    long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();
                }
            }
        }
    }
}
```

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：

```

@Override
public Connection getConnection() throws SQLException {
    return popConnection(dataSource.getUsername(), dataSource.getPassword()).getProxyConnection();
}

@Override
public Connection getConnection(String username, String password) throws SQLException {
    return popConnection(username, password).getProxyConnection();
}
  
```

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：



### setAutoCommit

```
void setAutoCommit(boolean autoCommit)
    throws SQLException
```

将此连接的自动提交模式设置为给定状态。如果连接处于自动提交模式下，则它的所有 SQL 语句将被执行并作为单个事务提交。否则，它的 SQL 语句将集到事务中，直到调用 commit 方法或 rollback 方法为止。默认情况下，新连接处于自动提交模式。

提交发生在语句完成时。语句完成的时间取决于 SQL 语句的类型：

- 对于 DML 语句（比如 Insert、Update 或 Delete）和 DDL 语句，语句在执行完毕时完成。
- 对于 Select 语句，语句在关联结果集关闭时完成。
- 对于 CallableStatement 对象或者返回多个结果的语句，语句在所有关联结果集关闭并且已获得所有更新计数和输出参数时完成。

注：如果在事务和自动提交模式更改期间调用此方法，则提交该事务。如果调用 setAutoCommit 而自动提交模式未更改，则该调用无操作（no-op）。

参数：

autoCommit - 为 true 表示启用自动提交模式；为 false 表示禁用自动提交模式

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

Mybatis 中事务的提交方式，本质上就是调用 JDBC 的 setAutoCommit() 来实现事务控制。

我们运行之前所写的代码：

@Test

```
public void testSaveUser() throws Exception {
    User user = new User();
    user.setUsername("mybatis user09");
    //6.执行操作
    int res = userDao.saveUser(user);
    System.out.println(res);
    System.out.println(user.getId());
}
```

@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{
    //7.提交事务
```



```
session.commit();

//8.释放资源
session.close();
in.close();
}
```

观察它在控制台输出的结果：

```
Opening JDBC Connection
Created connection 982007015,
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@3a883ce7]
==> Preparing: insert into user(username,birthday,sex,address) values(?,?,?,?);
==> Parameters: 小二王(String), 2018-03-04 11:34:34.864(Timestamp), 女(String), 北京金燕龙(String)
<== Updates: 1
==> Preparing: select last_insert_id();
==> Parameters:
<== Total: 1
Committing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3a883ce7]
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@3a883ce7]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3a883ce7]
Returned connection 982007015 to pool.
```

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 sqlSession.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 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 SqlSessionFactory 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:

```
@Override  
public SqlSession openSession(boolean autoCommit) {  
    return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, autoCommit);  
}
```

运行的结果如下:

```
Opening JDBC Connection  
Created connection 982007015.  
==> Preparing: insert into user(username,birthday,sex,address) values(?,?,?,?);  
==> Parameters: 传智播客(String), 2018-03-04 12:04:06.626(Timestamp), 男(String), 北京金燕龙(String)  
<== Updates: 1  
==> Preparing: select last_insert_id();  
==> Parameters:  
<== Total: 1  
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@3a883ce7]  
Returned connection 982007015 to pool.
```

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:

### 5.1 Dynamic SQL

One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

While working with Dynamic SQL will never be a party, MyBatis certainly improves the situation with a powerful Dynamic SQL language that can be used within any mapped SQL statement.

The Dynamic SQL elements should be familiar to anyone who has used JSTL or any similar XML based text processors. In previous versions of MyBatis, there were a lot of elements to know and understand. MyBatis 3 greatly improves upon this, and now there are less than half of those elements to work with. MyBatis employs powerful OGNL based expressions to eliminate most of the other elements:

- if
- choose (when, otherwise)
- trim (where, set)
- foreach





## 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
    <if test="username!=null and username != '' ">
        and username like #{username}
    </if>
    <if test="address != null">
        and address like #{address}
    </if>
</select>
```

注意：<if>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。另外要注意 where 1=1 的作用~！

### 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 QueryVo 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="id in ( " close=")" item="uid"  
separator=", ">  
                #{uid}  
            </foreach>  
        </if>  
    </where>  
</select>
```

SQL 语句:

```
select 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  
separator: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">
```



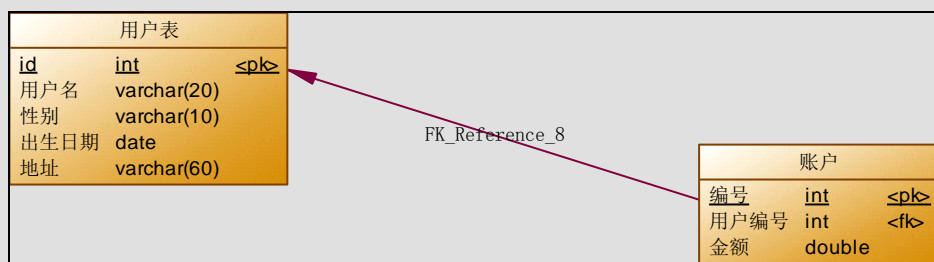
```
<include refid="defaultSql"></include>

where id = #{uid}

</select>
```

## 第3章 Mybatis 多表查询之一对多

本次案例主要以最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：



### 3.1 一对一查询(多对一)

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 Sql 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.uid = user.id
```

在 MySQL 中测试的查询结果如下：

	ID	UID	MONEY	username	address
<input type="checkbox"/>	1	41	1000	老王	北京
<input type="checkbox"/>	2	45	1000	传智播客	北京金燕龙

### 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
    public String toString() {
        return super.toString() + "    AccountUser [username=" + username + ",
address=" + address + "]\n";
    }
}
```

### 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 `resultType` 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 accountDao;

    @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.openSession();
    //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:

	id	username	birthday	sex	address	id	uid	money
<input type="checkbox"/>	41	老王	2018-02-27 17:47:08	男	北京	1	41	1000
<input type="checkbox"/>	41	老王	2018-02-27 17:47:08	男	北京	3	41	2000
<input type="checkbox"/>	42	小二王	2018-03-02 15:09:37	女	北京金燕龙	(NULL)	(NULL)	(NULL)
<input type="checkbox"/>	43	小二王	2018-03-04 11:34:34	女	北京金燕龙	(NULL)	(NULL)	(NULL)
<input type="checkbox"/>	45	传智播客	2018-03-04 12:04:06	男	北京金燕龙	2	45	1000

### 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 class User implements Serializable {

    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    private List<Account> accounts;

    public List<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(List<Account> accounts) {
        this.accounts = accounts;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Date getBirthday() {
        return birthday;
    }
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
}
```



```
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>  
  
    <select id="findAll" resultType="user">  
        select * from user;  
    </select>  
</mapper>
```



```

        </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: MybatisCRUDTest</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：



在MySQL 数据库中添加角色表，用户角色的中间表。

角色表



ID	ROLE_NAME	ROLE_DESC
1	院长	管理整个学院
2	总裁	管理整个公司
3	校长	管理整个学校

用户角色中间表

UID	RID
41	1
45	1
41	2

## 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.itheima.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

```
/**
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 */
public class RoleTest {

    private InputStream in;
    private SqlSession sqlSession;
    private IRoleDao roleDao;

    @Before//用于在测试方法执行之前执行
    public void init() throws Exception{
        //1.读取配置文件，生成字节输入流
        in = Resources.getResourceAsStream("SqlMapConfig.xml");
        //2.获取 SqlSessionFactory
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
        //3.获取 SqlSession 对象
        sqlSession = factory.openSession(true);
        //4.获取 dao 的代理对象
        roleDao = sqlSession.getMapper(IRoleDao.class);
    }

    @After//用于在测试方法执行之后执行
    public void destroy() throws Exception{
        //提交事务
        // sqlSession.commit();
        //6.释放资源
        sqlSession.close();
        in.close();
    }

    /**
     * 测试查询所有
     */
    @Test
    public void testFindAll(){
        List<Role> roles = roleDao.findAll();
    }
}
```



```
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。