MyBatis 的 XxxMapper.xml 映射器的详解

一．XxxMapper.xml 映射器的使用

在 MyBatis 中，将 Dao 层的接口与对应的 Mapper.xml 配置文件进行组合使用，而不是以前的接口实现类处理。 这里着重讲解一下，这个配置文件的使用。

将 XxxMapper.xml 放置在与接口 XxxMapper.java 同级的目录下。

一.一 Schema 约束

```
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
```

当然，前面不要忘记 xml 文件的头部。

```
<?xml version="1.0" encoding="UTF-8"?>
```

一.二 Mapper 根节点

有一个根节点，是 mapper，里面只有一个属性， namespace，命名空间。 后面跟的值一般为其所在的包路径，或者说是 XxxMapper.java 接口所在的包路径。

```
<mapper namespace="com.yjl.mapper.UserMapper" >
```

一.三 下属节点

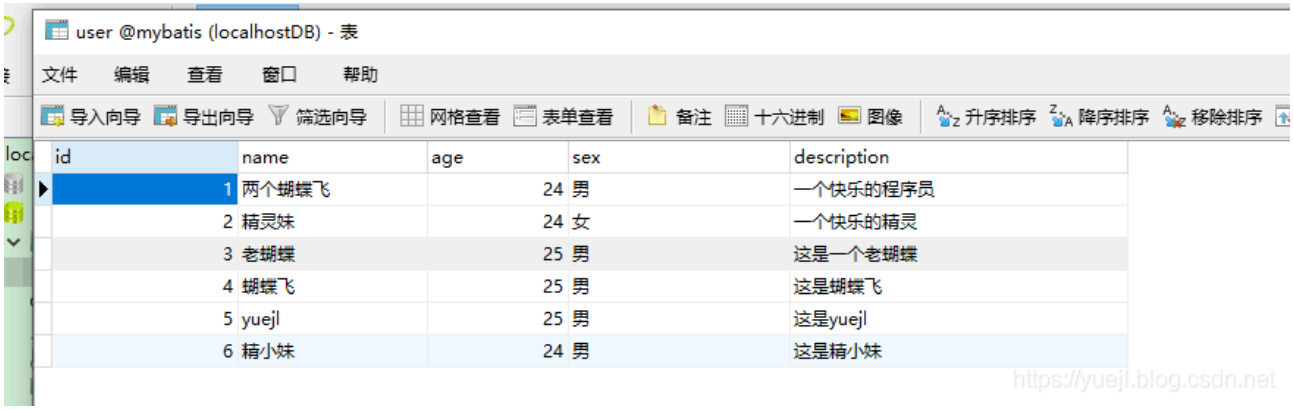
在根节点 mapper 下的节点有:

1. select　　最常用，最复杂的元素之一， 可以自定义参数，返回结果集。
2. insert　　插入语句 返回所影响的行数
3. update　　更新语句 返回所影响的行数
4. delete　　删除语句 返回所影响的行数
5. sql　　定义一部分 sql,构成 sql 片段，然后在各个语句中引用。
6. resultMap　　从结果集中来加载对象，最复杂也是最强大的元素，提供了映射的规则。
7. cache　　给定命名空间的缓存配置
8. cache-ref　　其他命名空间缓存配置的引用

其中，有一个 parameterMap ,已经被放弃使用了。

二. Select 元素的配置

数据库数据为:



id	name	age	sex	description
1	两个蝴蝶飞	24	男	一个快乐的程序员
2	精灵妹	24	女	一个快乐的精灵
3	老蝴蝶	25	男	这是一个老蝴蝶
4	蝴蝶飞	25	男	这是蝴蝶飞
5	yuejl	25	男	这是yuejl
6	精小妹	24	男	这是精小妹

在以前的 mapper 映射中:

```
    <select id="getById"  parameterType="int" resultType="user">
        <!-- 设置别名 -->
        select * from user where id=#{id}
    </select>
```

```
1
2
3
4
5
```

select 下面有 id 属性,parameterType 参数类型, resultType 结果类型属性等节点， 这些不同的节点表示不同的作用与意义。 除了这些之外，还有其他的节点属性。 只需要记住一些常见的节点即可。

二.一 所属节点

元　素	说　明	备　注
id	它和 Mapper 的命名空间组合起来是唯一的，提供给 MyBatis 调用	如过命名空间和 id 组合起来不唯一，MyBatis 将抛出异常
parameterType	你可以给出类的全命名，也可以给出类的别名，但使用别名必须是 MyBatis 内部定义或者自定义的	我们可以选择 JavaBean、Map 等复杂的参数类型传递给 SQL
parameterMap	即将废弃的元素，我们不再讨论它	—
resultType	定义类的全路径，在允许自动匹配的情况下，结果集将通过 JavaBean 的规范映射； 或定义为 int、double、float 等参数； 也可以使用别名，但是要符合别名规范，不能和 resultMap 同时使用	它是我们常用的参数之一，比如我们统计总条数就可以把它的值设置为 int
resultMap	它是映射集的引用，将执行强大的映射功能，我们可以使用 resultType 或者 resultMap 其中的一个，resultMap 可以给予我们自定义映射规则的机会	它是 MyBatis 最复杂的元素，可以配置映射规则、级联、typeHandler 等
flushCache	它的作用是在调用 SQL 后，是否要求 MyBatis 清空之前查询的本地缓存和二级缓存	取值为布尔值，true/false。默认值为 false
useCache	启动二级缓存的开关，是否要求 MyBatis 将此次结果缓存	取值为布尔值，true/false。默认值为 true
timeout	设置超时参数，等超时的时候将抛出异常，单位为秒	默认值是数据库厂商提供的 JDBC 驱动所设置的秒数
fetchSize	获取记录的总条数设定	默认值是数据库厂商提供的 JDBC 驱动所设置的条数
statementType	告诉 MyBatis 使用哪个 JDBC 的 Statement 工作，取值为 STATEMENT(Statement)、PREPARED（PreparedStatement）、CallableStatement	默认值为 PREPARED
resultSetType	这是对 JDBC 的 resultSet 接口而言，它的值包括 FORWARD_ONLY（游标允许向前访问）、SCROLL_SENSITIVE（双向滚动，但不及时更新，就是如果数据库里的数据修改过，并不在 resultSet 中反应出来）、SCROLL_INSENSITIVE（双向滚动，并及时跟踪数据库的更新，以便更改 resultSet 中的数据）	默认值是数据库厂商提供的 JDBC 驱动所设置的
databaseId	它的使用请参考第 3 章的 databaseIdProvider 数据库厂商标识这部分内容	提供多种数据库的支持
resultOrdered	这个设置仅适用于嵌套结果集 select 语句。如果为 true，就是假设包含了嵌套结果集或者是分组了。当返回一个主结果行的时候，就不能对前面结果集的引用。这就确保了在获取嵌套的结果集的时候不至于导致内存不够用	取值为布尔值，true/false。默认值为 false
resultSets	适合于多个结果集的情况，它将列出执行 SQL 后每个结果集的名称，每个名称之间用逗号分隔	很少使用

下面就开始讲解 select 元素的常见使用。

二.二 like 的用法查询记录的数量

在 UserMapper.java 接口中:

```
public int countByName(String name);
```

在 UserMapper.xml 配置中其对应的配置语句为:

```
<!-- 传入参数查询数目 -->
	<select id="countByName" parameterType="string" resultType="int">
		<!--单个的，可以用 value 来接收-->
		select count(*) from user where name like '${value}'
	</select>
```

测试方法为:

```
@Test
	public void countByNameTest(){
		SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
		UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
		int count=userMapper.countByName("%蝴蝶%");
		System.out.println("输出数目为:"+count);
	}
```

```
4
5
6
7
```

控制台显示为:

```
15:25:13,909 DEBUG PooledDataSource:406 - Created connection 1208736537.
15:25:13,909 DEBUG JdbcTransaction:101 - Setting autocommit to false on JDBC Connection [com.mysql.jdbc
15:25:13,910 DEBUG countByName:159 - ==>  Preparing: select count(*) from user where name like '%蝴蝶%'
15:25:13,923 DEBUG countByName:159 - ==> Parameters:
15:25:13,934 DEBUG countByName:159 - <==      Total: 1
输出数目为:3
```

缺点， 上面传入的参数 需要自己手动拼接 %% 连接符， 用户只需要传入参数即可。 真正的拼接，应该放在数据库去完成。 每个数据库都有自己对应的字符串拼接方式。 mysql 可以使用 concat() 函数， oracle 可以使用 || .

```
<!-- 传入参数查询数目 -->
        <select id="countByName" parameterType="string" resultType="int">
                select count(*) from user where name like concat('%',#{name},'%')
        </select>
```

```
1
2
3
4
```

上面是两个%%的形式，如果是前% 为 concat(’ %’ ,#{name}), 后%为 concat(#{name},’ %’);

三. 传参 parameterType

传参可以使用三种方式

1. map 方式的传参(较旧的方式)
2. 注解@Param 的方式 (适用于参数较少的情况，且与实体属性无太太关联)
3. 实体对象 bean 的方式 (适用于插入，更新 等与实体属性关联的，或者属性过多的时候，封装成 bean 传入)

三.一 Map 形式的传参

接口:

```
//多个参数的时候。
        public List<User> findByNameAndSexMap(Map<String,Object> map);
```

xml 配置实现

```
<!-- 查询多个参数的时候 ,map 的形式  parameterType='map' map 为系统定义好的别名-->
        <select id="findByNameAndSexMap" parameterType="map" resultType="user">
                select * from user where name like concat('%',#{name},'%') and
sex=#{sex}
        </select>
```

测试方法:

```
@Test
        public void findByNameAndSexMapTest(){
                SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
                UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
                //定义参数
```

```
        Map<String,Object> paraMap=new HashMap<String,Object>();
        //key 与 xml 中配置的相同
        paraMap.put("name","蝴蝶");
        paraMap.put("sex","男");
        List<User> allList=userMapper.findByNameAndSexMap(paraMap);
        allList.forEach(n ->System.out.println(n));
    }
```

```
15:36:21,632 DEBUG findByNameAndSexMap:159 - ==>  Preparing: select * from user where name like
15:36:21,645 DEBUG findByNameAndSexMap:159 - ==> Parameters: 蝴蝶(String), 男(String)
15:36:21,657 DEBUG findByNameAndSexMap:159 - <==      Total: 3
User [id=1, name=两个蝴蝶飞, age=24, sex=男, description=一个快乐的程序员]
User [id=3, name=老蝴蝶, age=25, sex=男, description=这是一个老蝴蝶]
User [id=4, name=蝴蝶飞, age=25, sex=男, description=这是蝴蝶飞]
```

要注意，map 中的 key 键值 要与 sql 语句中的#{值} 保持一致。

三.二 @param 注解的方式

接口:

```
public List<User> findByNameAndSexAnnotation(@Param(value="name") String name,
                        @Param(value="sex") String sex);
```

sql 语句配置:

```
<!-- 查询多个参数的时候 ,注解 的形式, 没有参数类型  -->
        <select id="findByNameAndSexAnnotation" resultType="user">
                select * from user where name like concat('%',#{name},'%') and
sex=#{sex}
        </select>
```

测试方法， 为传参的形式

```
@Test
        public void findByNameAndSexAnnotationTest(){
                SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
                UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
                List<User> allList=userMapper.findByNameAndSexAnnotation("蝴蝶","男
");
                allList.forEach(n ->System.out.println(n));
        }
```

```
6
7
```

查询的结果与上面的一样。

此时，sql 语句中的值要与 注解的 value 值相同，不一定要与方法中的形参的值相同。 适用于参数较少的情况，一般不超过 5 个。

三.三 对象 bean 的方式

接口:

```
        public List<User> findByNameAndSexBean(User user);
```

```
1
```

sql 语句:

```
<!-- 实体对象 bean 的方式传参， 已经定义好别名 user 了 -->
        <select id="findByNameAndSexBean" parameterType="user" resultType="user">
                select * from user where name like concat('%',#{name},'%') and
sex=#{sex}
        </select>
```

```
1
2
3
4
```

测试方法:

```
@Test
        public void findByNameAndSexBeanTest(){
                SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
                UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
                User user=new User();
                user.setName("蝴蝶");
                user.setSex("男");
```

```
            List<User> allList=userMapper.findByNameAndSexBean(user);
            allList.forEach(n ->System.out.println(n));
        }
```

查询的结果与上面的一致。

三.四 三种传参方式的比较

map 形式: map 传入进去，是不知道 key 值是什么的，与 sql 语句隔离了，导致了业务可读性的丧失，导致了后续的扩展与维护的困难，应当果断废弃这种方式。

param 注解方式: 受参数个数的影响， 当 n<=5 时，是最好的方式， 比 java bean 还好，应该比 java bean 更直观。 当 n>5 时，多个参数调用会出现困难。

java bean 方式: 当参数个数过多时使用。 >5 时。

四. MyBatis 排序

排序，用 order by 进行排序。

四.一 单个参数排序，用 value 接收

接口:

```
public List<User> orderByAge(String age);
```

sql 语句:

```
<!-- 单个值排序，必须用 value 进行接收 -->
	<select id="orderByAge" parameterType="string" resultType="user">
		<!-- select * from user order by ${value} desc -->
		select * from user order by ${value} desc
	</select>
```

测试方法:

```
	@Test
	public void orderByAgeTest(){
		SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
		UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
		List<User> allList=userMapper.orderByAge("age");
		allList.forEach(n ->System.out.println(n));
	}
```

```
3
4
5
6
7
```

```
15:51:27,181 DEBUG JdbcTransaction:101 - Setting autoCommit to false on JDBC Connection [c
15:51:27,182 DEBUG orderByAge:159 - ==>  Preparing: select * from user order by age desc
15:51:27,195 DEBUG orderByAge:159 - ==> Parameters:
15:51:27,205 DEBUG orderByAge:159 - <==      Total: 6
User [id=3, name=老蝴蝶, age=25, sex=男, description=这是一个老蝴蝶]
User [id=4, name=蝴蝶飞, age=25, sex=男, description=这是蝴蝶飞]
User [id=5, name=yuejl, age=25, sex=男, description=这是yuejl]
User [id=1, name=两个蝴蝶飞, age=24, sex=男, description=一个快乐的程序员]
User [id=2, name=精灵妹, age=24, sex=女, description=一个快乐的精灵]
User [id=6, name=精小妹, age=24, sex=男, description=这是精小妹]
```


四.二 多个参数排序 ，用注解方式传入

接口:

```
public List<User> orderByAgeAndId(@Param("age") String age,@Param("id") String id);
```

```
1
```

sql 语句:

```
<!-- 多个参数排序，可以用注解的方式 -->
        <select id="orderByAgeAndId"  resultType="user">
                select * from user order by ${age} asc,${id} desc
        </select>
```

```
1
2
3
4
```

测试方法:

```
@Test
```

```
	public void orderByAgeAndIdTest(){
		SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
		UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
		List<User> allList=userMapper.orderByAgeAndId("age","id");
		allList.forEach(n ->System.out.println(n));
	}
```

```
15:56:29,377 DEBUG JdbcTransaction:101 - Setting autoCommit to false on JDBC Connection [com.mysql.j
15:56:29,378 DEBUG orderByAgeAndId:159 - ==>  Preparing: select * from user order by age asc,id desc
15:56:29,392 DEBUG orderByAgeAndId:159 - ==> Parameters:
15:56:29,403 DEBUG orderByAgeAndId:159 - <==      Total: 6
User [id=6, name=精小妹, age=24, sex=男, description=这是精小妹]
User [id=2, name=精灵妹, age=24, sex=女, description=一个快乐的精灵]
User [id=1, name=两个蝴蝶飞, age=24, sex=男, description=一个快乐的程序员]
User [id=5, name=yuejl, age=25, sex=男, description=这是yuejl]
User [id=4, name=蝴蝶飞, age=25, sex=男, description=这是蝴蝶飞]
User [id=3, name=老蝴蝶, age=25, sex=男, description=这是一个老蝴蝶]
```


五. 查询某些列的值

在 sql 语句的前面进行查询

接口:

```
public List<User> selectColumn(@Param(value="column1") String column1,
                    @Param(value="column2") String
column2,@Param(value="column3") String column3);
```

sql 语句:

```
<select id="selectColumn" parameterType="map" resultType="user">
		<!-- 也可以写别名进行操作。 -->
		select ${column1},${column2},${column3} from user
	</select>
```

测试方法:

```
@Test
	public void selectColumnTest(){
		SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
		UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
		List<User>
allList=userMapper.selectColumn("name","sex","description");
		allList.forEach(n ->System.out.println(n));
	}
```

```
16:00:18,596 DEBUG selectColumn:159 - ==> Parameters:
16:00:18,608 DEBUG selectColumn:159 - <==      Total: 6
User [id=null, name=两个蝴蝶飞, age=null, sex=男, description=一个快乐的程序员]
User [id=null, name=精灵妹, age=null, sex=女, description=一个快乐的精灵]
User [id=null, name=老蝴蝶, age=null, sex=男, description=这是一个老蝴蝶]
User [id=null, name=蝴蝶飞, age=null, sex=男, description=这是蝴蝶飞]
User [id=null, name=yuejl, age=null, sex=男, description=这是yuejl]
User [id=null, name=精小妹, age=null, sex=男, description=这是精小妹]
```

只有要查询的列才有值。

六. 查询结果 resultType

六.一 最简单的形式查询

数据表的字段要与实体 Bean 的类属性保持一致。 如，在实体类中是:

```
    private Integer id;
    private String name;
    private Integer age;
    private String sex;
    private String description;
```

数据库中是:

名	类型	长度	小数点	允许空值 (	
id	int	11	0	☐	1
name	varchar	255	0	☑	
age	int	11	0	☑	
sex	varchar	255	0	☑	
description	varchar	255	0	☑	

属性与字体一致的，可以进行相应的查询。

接口:

```
public List<User> findAll();
```

sql 语句:

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

3

测试方法:

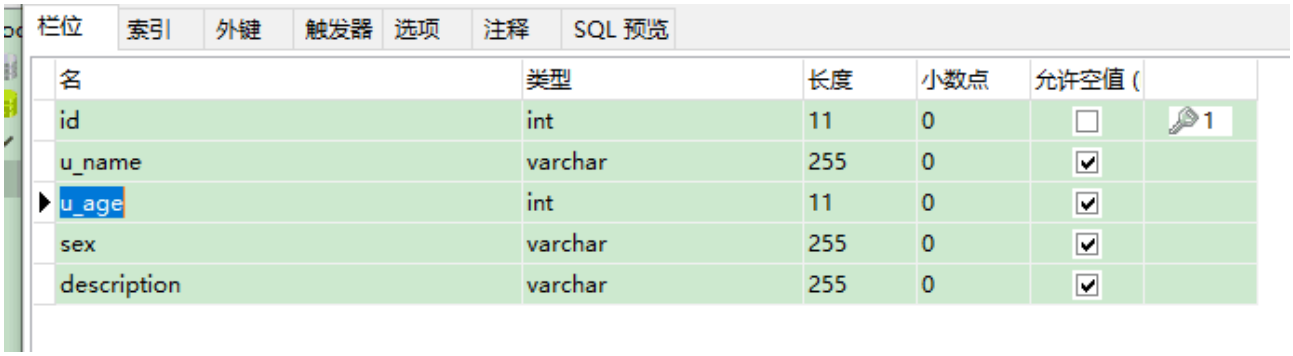
```
@Test
        public void findAllTest(){
                SqlSession sqlSession=SqlSessionFactoryUtils.getSession();
                UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
                List<User> allList=userMapper.findAll();
                allList.forEach(n ->System.out.println(n));
        }
```

1
2
3
4
5
6
7

```
18:49:33,413 DEBUG findAll:159 - ==> Parameters:
18:49:33,442 DEBUG findAll:159 - <==      Total: 6
User [id=1, name=两个蝴蝶飞, age=24, sex=男, description=一个快乐的程序员]
User [id=2, name=精灵妹, age=24, sex=女, description=一个快乐的精灵]
User [id=3, name=老蝴蝶, age=25, sex=男, description=这是一个老蝴蝶]
User [id=4, name=蝴蝶飞, age=25, sex=男, description=这是蝴蝶飞]
User [id=5, name=yuejl, age=25, sex=男, description=这是yuejl]
User [id=6, name=精小妹, age=24, sex=男, description=这是精小妹]
```


如果数据表的名称与属性不一致的话，那么此时的查询呢? name 和 age 不一致的情况。

在数据表中，将原先的 name 改成 u_name, age 改成 u_age



栏位 | 索引 | 外键 | 触发器 | 选项 | 注释 | SQL 预览

名	类型	长度	小数点	允许空值 (	
id	int	11	0	☐	1
u_name	varchar	255	0	☑	
u_age	int	11	0	☑	
sex	varchar	255	0	☑	
description	varchar	255	0	☑	

这个时候的查询呢?

```
19:01:53,360 DEBUG findAll:159 - ==>  Preparing: select * from user
19:01:53,372 DEBUG findAll:159 - ==> Parameters:
19:01:53,382 DEBUG findAll:159 - <==      Total: 6
User [id=1, name=null, age=null, sex=男, description=一个快乐的程序员]
User [id=2, name=null, age=null, sex=女, description=一个快乐的精灵]
User [id=3, name=null, age=null, sex=男, description=这是一个老蝴蝶]
User [id=4, name=null, age=null, sex=男, description=这是蝴蝶飞]
User [id=5, name=null, age=null, sex=男, description=这是yuejl]
User [id=6, name=null, age=null, sex=男, description=这是精小妹]
```


发现 name 和 age 是没有值的， 因为无法正确的 setter 和 getter,所以是取不到值的。

六.二 别名的方式查询，使表字段与类属性相同。

```
<!-- 使用别名方式查询，令别名为 类属性 -->
        <select id="findAll" resultType="user">
                select id,u_name as name,u_age as age,sex,description from user
        </select>
```

这个时候运行查询，是正确的。

```
19:04:42,756 DEBUG JdbcTransaction:101 - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC
19:04:42,757 DEBUG findAll:159 - ==>  Preparing: select id,u_name as name,u_age as age,sex,description from
19:04:42,770 DEBUG findAll:159 - ==> Parameters:
19:04:42,779 DEBUG findAll:159 - <==      Total: 6
User [id=1, name=两个蝴蝶飞, age=24, sex=男, description=一个快乐的程序员]
User [id=2, name=精灵妹, age=24, sex=女, description=一个快乐的精灵]
User [id=3, name=老蝴蝶, age=25, sex=男, description=这是一个老蝴蝶]
User [id=4, name=蝴蝶飞, age=25, sex=男, description=这是蝴蝶飞]
User [id=5, name=yuejl, age=25, sex=男, description=这是yuejl]
User [id=6, name=精小妹, age=24, sex=男, description=这是精小妹]
```


如果现在列名改成了，多个 sql 语句查询时，都要设置别名，是不是太麻烦了呢?

六.三 使用 resultMap 来实现

将结果集重新定义，定义成 resultMap 便可以了。 resultMap 的详细用法，在一对一，一对多的关联关系中会仔细讲解。

```
<!-- 定义结果类型的 map 集合形式 -->
        <resultMap type="user" id="userResultMap">
                <!-- 指定主键用 id, column 为表列名， property 为类属性 -->
                <id property="id" column="id"/>
                <!-- 普通属性用 result -->
                <result property="name" column="u_name"/>
                <result property="sex" column="sex"/>
                <result property="age" column="_age"/>
                <result property="description" column="description"/>
        </resultMap>
        <!-- 使用 resultMap 方式查询 -->
        <select id="findAll" resultMap="userResultMap">
                select * from user
        </select>
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

```
8
9
10
11
12
13
14
```

查询时，可以正确的查询出来值。

```
19:09:02,691 DEBUG JdbcTransaction:101 - Setting autoCommit to false on JD
19:09:02,692 DEBUG findAll:159 - ==>  Preparing: select * from user
19:09:02,705 DEBUG findAll:159 - ==> Parameters:
19:09:02,714 DEBUG findAll:159 - <==      Total: 6
User [id=1, name=两个蝴蝶飞, age=null, sex=男, description=一个快乐的程序员]
User [id=2, name=精灵妹, age=null, sex=女, description=一个快乐的精灵]
User [id=3, name=老蝴蝶, age=null, sex=男, description=这是一个老蝴蝶]
User [id=4, name=蝴蝶飞, age=null, sex=男, description=这是蝴蝶飞]
User [id=5, name=yuejl, age=null, sex=男, description=这是yuejl]
User [id=6, name=精小妹, age=null, sex=男, description=这是精小妹]
```


resultMap 是很强大的元素， 以后会重点讲解。

七. 插入的元素 insert 属性

属性名称	描　述	备　注
id	它和 Mapper 的命名空间组合起来是唯一的，作为唯一标识提供给 MyBatis 调用	如不唯一，MyBatis 将抛出异常
parameterType	你可以给出类的全命名，也可以是一个别名，但使用别名必须是 MyBatis 内部定义或者自定义的别名。定义方法可以参看第 3 章对 typeAlias 元素的讲解	我们可以选择 JavaBean、Map 等参数类型传递给 SQL
parameterMap	即将废弃的元素，我们不再讨论它	—
flushCache	它的作用是在调用 SQL 后，是否要求 MyBatis 清空之前查询的本地缓存和二级缓存	取值为布尔值，true/false。默认值为 false
timeout	设置超时参数，等超时的时候将抛出异常，单位为秒	默认值是数据库厂商提供的 JDBC 驱动所设置的秒数
statementType	告诉 MyBatis 使用哪个 JDBC 的 Statement 工作，取值为 STATEMENT(Statement)、 PREPARED（PreparedStatement） 和 CallableStatement	默认值为 PREPARED
keyProperty	表示以哪个列作为属性的主键。不能和 keyProperty 同时使用	设置哪个列为主键，如果你是联合主键可以用逗号将其隔开
useGeneratedKeys	这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键，例如，MySQL 和 SQL Server 自动递增字段，Oracle 的序列等，但是使用它就必须要给 keyProperty 或者 keyColumn 赋值	取值为布尔值，true/false。默认值为 false
keyColumn	指明第几列是主键，不能和 keyProperty 同时使用，只接受整形参数	和 keyProperty 一样联合主键，可以用逗号隔开
databaseId	它的使用请参考第 3 章关于 databaseIdProvider 数据库厂商标识这部分内容	提供多种数据库的支持
lang	自定义语言，可使用第三方语言，使用得较少，本书不做介绍	—

关于 insert 的用法，可以参看第三章的知识。

其中，插入的时候，可能并不是以 1 进行插入，会自定义规则。 测试方法与前面的一样，sql 接口是:

```
<!-- 复杂的插入 -->
        <insert id="insertUser" parameterType="user" useGeneratedKeys="true"
        keyProperty="id">
                <selectKey keyProperty="id" resultType="int" order="BEFORE">
                        select if(max(id) is null,1,max(id)+2) as newId from user
                </selectKey>
                insert into user(id,name,sex,age,description)
                values(#{id},#{name},#{sex},#{age},#{description})
        </insert>
```

```
1
2
3
4
5
6
7
8
9
```

八. 更新的元素 update 和删除的元素 delete

关于 update 和 delete 的用法，可以参看第三章的知识。

九. sql 元素

在一个表中， 用 select * from user 中 select * 是不太好的，实际开发中会用属性来代替， 如 select id,u_name,sex,u_age,descrption from user 那么这个前面的查询会写很多个，如果改变了其中的一个字段，那么就要改变多个。 可以先将这个 select id,u_name,sex，u_age,description 先当成 sql 片段定义起来，然后再各个地方进行引用即可。

九.一 普通单个引用

```
<sql id="userSql">
        id,u_name,sex,u_age,description
</sql>
<select id="findAll" resultMap="userResultMap">
        select <include refid="userSql"/> from user
</select>
```

```
1
2
3
4
5
6
```

用 sql 元素来定义片段， 用 include 来引入片段。

九.二 带参数的复杂引用

```
        <!-- 用的是$ 拼接符 -->
        <sql id="userSql">

        ${prefix}.id,${prefix}.u_name,${prefix}.sex,${prefix}.u_age,${prefix}.descr
iption
        </sql>
        <select id="findAll" resultType="user">
                select <include refid="userSql">
                                        <property name="prefix" value="t"/>
                                </include>
                from user t
        </select>
```

```
1
2
3
4
5
6
7
```

```
8
9
10
```

注意，用的是${}，并不是以前常用的#{}．