**[mybaits中sql语句新功能(Ibatis3.0以后称Mybatis)](http://blog.csdn.net/chengjiangbo/article/details/11179895)**

分类： [MyBatis](http://blog.csdn.net/chengjiangbo/article/category/1618929) 2013-09-05 21:59 43人阅读 [评论](http://blog.csdn.net/chengjiangbo/article/details/11179895#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/chengjiangbo/article/details/11179895#report)

目录[(?)[+]](http://blog.csdn.net/chengjiangbo/article/details/11179895)

**1、Mybatis主键自动生成：**

在MYSQL、sql server等支持主键自动增长的数据库中！mybatis插入时，对自动生成的字段的写法：

<insert  id=”insertAuthor” parameterType=”类型”useGeneratedKeys=”true”

keyProperty=“自动生成的字段名”>

insert  into(uername, password, email )

      vlaues(#{username}, #{password},#{email})

<insert>

对不支持自动生成功能的数据库，mybatis提供以下写法，不过，此写法生成ID，是随机的！

 <insert id=”insertAuthor” parameterType=“类型”>

<selectKeyresultType=*"java.lang.Long"*order=*"AFTER"*keyProperty=*"id"*>

            SELECT LAST\_INSERT\_ID() AS id

        </selectKey>

Insert into 数据库名

  （id, username, password, email）

Values  (#{id}，#{username}，#{password}，#{email})

</insert>

**2、Mybaits中SQL语句包含**：

 <sql  id=”Columns”> select/update/delete/insert等操作</sql>

<select id=”selectUser” parameterType=”int”resultType=”hashMap”>

  Select  <include  refid=”Columns”/>

     From  表名或where条件

  Where id=#{id}

</select>

**3、Mybatis动态SQL语句**

a、if语句

   select 字段名 from　表名　where state=”1”

     <if  test =”字段名!=null”>

         AND 条件

     </if>

b、choose, when ,otherwise

    select 字段名 from　表名  where state=”1”

      <choose>

         <when test=”字段名!=null”>

             And 条件

         </when>

         <when test=”条件表达式”>

             And 条件

         </when>

         <otherwise >

              And 条件

          </otherwise>

       </choose>

C、trim, where , set

    1, where

       Select 字段名  from表名

          <where>

              条件

          </where>

        注 :加 <where>后则确保一定是 where开头

    2, set

       Update  表名

         <set>

             <if test=”条件”>字段名=#{参数}</if>

         </set>

            Where  条件

D、foreach

       通常构建在in条件中

         Select  字段名  from表名

            Where 字段名 in

               <foreach  item=”参数名” index=”index” collection=”list”

Open=”(” separator=”,” close=”)” >

#{参数名}

</foreach>

 E、作用例：批量删除

 <delete id = "delete" parameterType = "java.util.List">

    <![CDATA[

       delete from tests where id in

        ]]>

     <foreach collection="list" item = "要删除的id" open="(" separator="," close=")">

#{ 要删除的id}

      </foreach>

</delete>

  F、模糊查询：

      select 字段名 from表名 where  字段名 like  "%" #{参数}

**2.2 select**

一个select 元素非常简单。例如：

**Xml代码**

1. <!-- 查询学生，根据id -->

2. **<select** id="getStudent" parameterType="String" resultMap="studentResultMap"**>**

3.     SELECT ST.STUDENT\_ID,

4.                ST.STUDENT\_NAME,

5.                ST.STUDENT\_SEX,

6.                ST.STUDENT\_BIRTHDAY,

7.                ST.CLASS\_ID

8.           FROM STUDENT\_TBL ST

9.          WHERE ST.STUDENT\_ID = #{studentID}

10.**</select>**    
这条语句就叫做‘getStudent，有一个String参数，并返回一个StudentEntity类型的对象。  
注意参数的标识是：#{studentID}。

select 语句属性配置细节：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 描述 | 取值 | 默认 |
| id | 在这个模式下唯一的标识符，可被其它语句引用 |  |  |
| parameterType | 传给此语句的参数的完整类名或别名 |  |  |
| resultType | 语句返回值类型的整类名或别名。注意，如果是集合，那么这里填写的是集合的项的整类名或别名，而不是集合本身的类名。（resultType与resultMap 不能并用） |  |  |
| resultMap | 引用的外部resultMap 名。结果集映射是MyBatis 中最强大的特性。许多复杂的映射都可以轻松解决。（resultType与resultMap 不能并用） |  |  |
| flushCache | 如果设为true，则会在每次语句调用的时候就会清空缓存。select语句默认设为false | true|false | false |
| useCache | 如果设为true，则语句的结果集将被缓存。select语句默认设为false true|false false   timeout 设置驱动器在抛出异常前等待回应的最长时间，默认为不设值，由驱动器自己决定 | true|false | false |
| timeout | 设置驱动器在抛出异常前等待回应的最长时间，默认为不设值，由驱动器自己决定 | 正整数 | 未设置 |
| fetchSize | 设置一个值后，驱动器会在结果集数目达到此数值后，激发返回，默认为不设值，由驱动器自己决定 | 正整数 | 驱动器决定 |
| statementType | statement，preparedstatement，callablestatement。 预准备语句、可调用语句 | STATEMENT   PREPARED   CALLABLE | PREPARED |
| resultSetType | forward\_only，scroll\_sensitive，scroll\_insensitive   只转发，滚动敏感，不区分大小写的滚动 | FORWARD\_ONLY   SCROLL\_SENSITIVE   SCROLL\_INSENSITIVE | 驱动器决定 |

**2.3 insert**

 一个简单的insert语句：

**Xml代码**

1. <!-- 插入学生 -->

2. **<insert** id="insertStudent" parameterType="StudentEntity"**>**

3.         INSERT INTO STUDENT\_TBL (STUDENT\_ID,

4.                                           STUDENT\_NAME,

5.                                           STUDENT\_SEX,

6.                                           STUDENT\_BIRTHDAY,

7.                                           CLASS\_ID)

8.               VALUES   (#{studentID},

9.                           #{studentName},

10.                          #{studentSex},

11.                          #{studentBirthday},

12.                          #{classEntity.classID})

13.**</insert>**

 insert可以使用数据库支持的自动生成主键策略，设置useGeneratedKeys=”true”，然后把keyProperty 设成对应的列，就搞定了。比如说上面的StudentEntity 使用auto-generated 为id 列生成主键.  
 还可以使用selectKey元素。下面例子，使用mysql数据库nextval('student')为自定义函数，用来生成一个key。

**Xml代码**

1. <!-- 插入学生 自动主键-->

2. **<insert** id="insertStudentAutoKey" parameterType="StudentEntity"**>**

3.     **<selectKey** keyProperty="studentID" resultType="String" order="BEFORE"**>**

4.             select nextval('student')

5.     **</selectKey>**

6.         INSERT INTO STUDENT\_TBL (STUDENT\_ID,

7.                                  STUDENT\_NAME,

8.                                  STUDENT\_SEX,

9.                                  STUDENT\_BIRTHDAY,

10.                                 CLASS\_ID)

11.              VALUES   (#{studentID},

12.                        #{studentName},

13.                        #{studentSex},

14.                        #{studentBirthday},

15.                        #{classEntity.classID})

16.**</insert>**

insert语句属性配置细节：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 描述 | 取值 | 默认 |
| id | 在这个模式下唯一的标识符，可被其它语句引用 |  |  |
| parameterType | 传给此语句的参数的完整类名或别名 |  |  |
| flushCache | 如果设为true，则会在每次语句调用的时候就会清空缓存。select语句默认设为false | true|false | false |
| useCache | 如果设为true，则语句的结果集将被缓存。select语句默认设为false true|false false   timeout 设置驱动器在抛出异常前等待回应的最长时间，默认为不设值，由驱动器自己决定 | true|false | false |
| timeout | 设置驱动器在抛出异常前等待回应的最长时间，默认为不设值，由驱动器自己决定 | 正整数 | 未设置 |
| fetchSize | 设置一个值后，驱动器会在结果集数目达到此数值后，激发返回，默认为不设值，由驱动器自己决定 | 正整数 | 驱动器决定 |
| statementType | statement，preparedstatement，callablestatement。 预准备语句、可调用语句 | STATEMENT   PREPARED   CALLABLE | PREPARED |
| useGeneratedKeys | 告诉MyBatis 使用JDBC 的getGeneratedKeys 方法来获取数据库自己生成的主键（MySQL、SQLSERVER 等  关系型数据库会有自动生成的字段）。默认：false | true|false | false |
| keyProperty | 标识一个将要被MyBatis 设置进getGeneratedKeys 的key 所返回的值，或者为insert 语句使用一个selectKey  子元素。 |  |  |

selectKey语句属性配置细节：

|  |  |  |
| --- | --- | --- |
| 属性 | 描述 | 取值 |
| keyProperty | selectKey 语句生成结果需要设置的属性。 |  |
| resultType | 生成结果类型，MyBatis 允许使用基本的数据类型，包括String 、int类型。 |  |
| order | 可以设成BEFORE 或者AFTER，如果设为BEFORE，那它会先选择主键，然后设置keyProperty，再执行insert语句；如果设为AFTER，它就先运行 insert 语句再运行selectKey 语句，通常是insert语句中内部调用数据库（像Oracle）内嵌的序列机制。 | BEFORE   AFTER |
| statementType | 像上面的那样， MyBatis支持STATEMENT，PREPARED和CALLABLE的语句形式，对应Statement ，PreparedStatement和CallableStatement 响应 | STATEMENT   PREPARED   CALLABLE |

**2.4 update、delete**

一个简单的update：

**Xml代码**

1. <!-- 更新学生信息 -->

2. **<update** id="updateStudent" parameterType="StudentEntity"**>**

3.         UPDATE STUDENT\_TBL

4.             SET STUDENT\_TBL.STUDENT\_NAME = #{studentName},

5.                 STUDENT\_TBL.STUDENT\_SEX = #{studentSex},

6.                 STUDENT\_TBL.STUDENT\_BIRTHDAY = #{studentBirthday},

7.                 STUDENT\_TBL.CLASS\_ID = #{classEntity.classID}

8.          WHERE STUDENT\_TBL.STUDENT\_ID = #{studentID};

9. **</update>**

一个简单的delete：

**Xml代码**

1. <!-- 删除学生 -->

2. **<delete** id="deleteStudent" parameterType="StudentEntity"**>**

3.         DELETE FROM STUDENT\_TBL WHERE STUDENT\_ID = #{studentID}

4. **</delete>**

 update、delete语句属性配置细节：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 描述 | 取值 | 默认 |
| id | 在这个模式下唯一的标识符，可被其它语句引用 |  |  |
| parameterType | 传给此语句的参数的完整类名或别名 |  |  |
| flushCache | 如果设为true，则会在每次语句调用的时候就会清空缓存。select语句默认设为false | true|false | false |
| useCache | 如果设为true，则语句的结果集将被缓存。select语句默认设为false true|false false   timeout 设置驱动器在抛出异常前等待回应的最长时间，默认为不设值，由驱动器自己决定 | true|false | false |
| timeout | 设置驱动器在抛出异常前等待回应的最长时间，默认为不设值，由驱动器自己决定 | 正整数 | 未设置 |
| fetchSize | 设置一个值后，驱动器会在结果集数目达到此数值后，激发返回，默认为不设值，由驱动器自己决定 | 正整数 | 驱动器决定 |
| statementType | statement，preparedstatement，callablestatement。 预准备语句、可调用语句 | STATEMENT   PREPARED   CALLABLE | PREPARED |

**2.5 sql**

Sql元素用来定义一个可以复用的SQL 语句段，供其它语句调用。比如：

**Xml代码**

1. <!-- 复用sql语句  查询student表所有字段 -->

2. **<sql** id="selectStudentAll"**>**

3.         SELECT ST.STUDENT\_ID,

4.                    ST.STUDENT\_NAME,

5.                    ST.STUDENT\_SEX,

6.                    ST.STUDENT\_BIRTHDAY,

7.                    ST.CLASS\_ID

8.               FROM STUDENT\_TBL ST

9. **</sql>**

   这样，在select的语句中就可以直接引用使用了，将上面select语句改成：

**Xml代码**

1. <!-- 查询学生，根据id -->

2. **<select** id="getStudent" parameterType="String" resultMap="studentResultMap"**>**

3.     **<include** refid="selectStudentAll"**/>**

4.             WHERE ST.STUDENT\_ID = #{studentID}

5. **</select>**

**2.6parameters**

       上面很多地方已经用到了参数，比如查询、修改、删除的条件，插入，修改的数据等，MyBatis可以使用的基本数据类型和Java的复杂数据类型。  
        基本数据类型，String，int，date等。  
        但是使用基本数据类型，只能提供一个参数，所以需要使用Java实体类，或Map类型做参数类型。通过#{}可以直接得到其属性。

**2.6.1基本类型参数**

 根据入学时间，检索学生列表：

**Xml代码**

1. <!-- 查询学生list，根据入学时间  -->

2. **<select** id="getStudentListByDate"  parameterType="Date" resultMap="studentResultMap"**>**

3.     SELECT \*

4.       FROM STUDENT\_TBL ST LEFT JOIN CLASS\_TBL CT ON ST.CLASS\_ID = CT.CLASS\_ID

5.      WHERE CT.CLASS\_YEAR = #{classYear};

6. **</select>**

**Java代码**

1. List<StudentEntity> studentList = studentMapper.getStudentListByClassYear(StringUtil.parse("2007-9-1"));

2. **for** (StudentEntity entityTemp : studentList) {

3.     System.out.println(entityTemp.toString());

4. }

**2.6.2Java实体类型参数**

 根据姓名和性别，检索学生列表。使用实体类做参数：

**Xml代码**

1. <!-- 查询学生list，like姓名、=性别，参数entity类型 -->

2. **<select** id="getStudentListWhereEntity" parameterType="StudentEntity" resultMap="studentResultMap"**>**

3.     SELECT \* from STUDENT\_TBL ST

4.         WHERE ST.STUDENT\_NAME LIKE CONCAT(CONCAT('%', #{studentName}),'%')

5.           AND ST.STUDENT\_SEX = #{studentSex}

6. **</select>**

**Java代码**

1. StudentEntity entity = **new** StudentEntity();

2. entity.setStudentName("李");

3. entity.setStudentSex("男");

4. List<StudentEntity> studentList = studentMapper.getStudentListWhereEntity(entity);

5. **for** (StudentEntity entityTemp : studentList) {

6.     System.out.println(entityTemp.toString());

7. }

**2.6.3Map参数**

根据姓名和性别，检索学生列表。使用Map做参数：

**Xml代码**

1. <!-- 查询学生list，=性别，参数map类型 -->

2. **<select** id="getStudentListWhereMap" parameterType="Map" resultMap="studentResultMap"**>**

3.     SELECT \* from STUDENT\_TBL ST

4.      WHERE ST.STUDENT\_SEX = #{sex}

5.           AND ST.STUDENT\_SEX = #{sex}

6. **</select>**

**Java代码**

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

2. map.put("sex", "女");

3. map.put("name", "李");

4. List<StudentEntity> studentList = studentMapper.getStudentListWhereMap(map);

5. **for** (StudentEntity entityTemp : studentList) {

6.     System.out.println(entityTemp.toString());

7. }

**2.6.4多参数的实现**

 如果想传入多个参数，则需要在接口的参数上添加@Param注解。给出一个实例：  
 接口写法：

**Java代码**

1. **public** List<StudentEntity> getStudentListWhereParam(@Param(value = "name") String name, @Param(value = "sex") String sex, @Param(value = "birthday") Date birthdar, @Param(value = "classEntity") ClassEntity classEntity);

SQL写法：

**Xml代码**

1. <!-- 查询学生list，like姓名、=性别、=生日、=班级，多参数方式 -->

2. **<select** id="getStudentListWhereParam" resultMap="studentResultMap"**>**

3.     SELECT \* from STUDENT\_TBL ST

4.     **<where>**

5.         **<if** test="name!=null and name!='' "**>**

6.             ST.STUDENT\_NAME LIKE CONCAT(CONCAT('%', #{name}),'%')

7.         **</if>**

8.         **<if** test="sex!= null and sex!= '' "**>**

9.             AND ST.STUDENT\_SEX = #{sex}

10.        **</if>**

11.        **<if** test="birthday!=null"**>**

12.            AND ST.STUDENT\_BIRTHDAY = #{birthday}

13.        **</if>**

14.        **<if** test="classEntity!=null and classEntity.classID !=null and classEntity.classID!='' "**>**

15.            AND ST.CLASS\_ID = #{classEntity.classID}

16.        **</if>**

17.    **</where>**

18.**</select>**

进行查询：

**Java代码**

1. List<StudentEntity> studentList = studentMapper.getStudentListWhereParam("", "",StringUtil.parse("1985-05-28"), classMapper.getClassByID("20000002"));

2. **for** (StudentEntity entityTemp : studentList) {

3.     System.out.println(entityTemp.toString());

4. }

**2.6.5字符串代入法**

       默认的情况下，使用#{}语法会促使 MyBatis 生成PreparedStatement 属性并且使用PreparedStatement 的参数（=？）来安全的设置值。尽量这些是快捷安全，也是经常使用的。但有时候你可能想直接未更改的字符串代入到SQL 语句中。比如说，对于ORDER BY，你可能会这样使用：ORDER BY ${columnName}但MyBatis 不会修改和规避掉这个字符串。  
        注意：这样地接收和应用一个用户输入到未更改的语句中，是非常不安全的。这会让用户能植入破坏代码，所以，要么要求字段不要允许客户输入，要么你直接来检测他的合法性 。

**2.7 cache缓存**

        MyBatis 包 含一个强在的、可配置、可定制的缓存机制。MyBatis 3 的缓存实现有了许多改进，既强劲也更容易配置。默认的情况，缓存是没有开启，除了会话缓存以外，它可以提高性能，且能解决全局依赖。开启二级缓存，你只需 要在SQL 映射文件中加入简单的一行：<cache/>

这句简单的语句的作用如下：

1. 所有在映射文件里的select 语句都将被缓存。  
2. 所有在映射文件里insert,update 和delete语句会清空缓存。  
3. 缓存使用“最近很少使用”算法来回收  
4. 缓存不会被设定的时间所清空。  
5. 每个缓存可以存储1024 个列表或对象的引用（不管查询出来的结果是什么）。  
6. 缓存将作为“读/写”缓存，意味着获取的对象不是共享的且对调用者是安全的。不会有其它的调用  
7. 者或线程潜在修改。

例如，创建一个FIFO 缓存让60 秒就清空一次，存储512 个对象结果或列表引用，并且返回的结果是只读。因为在不用的线程里的两个调用者修改它们可能会导致引用冲突。

**Xml代码**

1. **<cache** eviction="FIFO" flushInterval="60000" size="512" readOnly="true"**>**

2. **</cache>**

    还可以在不同的命名空间里共享同一个缓存配置或者实例。在这种情况下，你就可以使用cache-ref 来引用另外一个缓存。

**Xml代码**

1. **<cache-ref** namespace="com.liming.manager.data.StudentMapper"**/>**

Cache 语句属性配置细节：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 说明 | 取值 | 默认值 |
| eviction | 缓存策略：   LRU - 最近最少使用法：移出最近较长周期内都没有被使用的对象。   FIFI- 先进先出：移出队列里较早的对象   SOFT - 软引用：基于软引用规则，使用垃圾回收机制来移出对象   WEAK - 弱引用：基于弱引用规则，使用垃圾回收机制来强制性地移出对象 | LRU   FIFI   SOFT   WEAK | LRU |
| flushInterval | 代表一个合理的毫秒总计时间。默认是不设置，因此使用无间隔清空即只能调用语句来清空。 | 正整数 | 不设置 |
| size | 缓存的对象的大小 | 正整数 | 1024 |
| readOnly | 只读缓存将对所有调用者返回同一个实例。因此都不能被修改，这可以极大的提高性能。可写的缓存将通过序列  化来返回一个缓存对象的拷贝。这会比较慢，但是比较安全。所以默认值是false。 | true|false | false |

今天弄了一下mybatis，发现网上关于mybatis模糊查询的很多人不知道，好像也没人说，所以我就把我关于mybatis模糊查询的用法写出来供有需要的人参考一下，希望对有需要的人能有所帮助！

## Xml代码  [收藏代码](javascript:void())

1. <select id="selectByName" parameterType="String" resultType="Student">
2. select \* from Student s where s.name like "%"#{name}"%";
3. </select>

这里我再补充一下，如果你的student.xml文件里面针对这个查询的配置只用到了name的话，那么你直接给它传一个String类型的 name是没有问题的，如果有多个条件的话你也可以给它传多个参数，当然你也可以给它传一个对象，而该对象所对应的属性就是你所要查的属性，像上面那样你 就可以在你的程序里面这样写

Java代码  [收藏代码](javascript:void())

1. @Override
2. public List<Student> findAllByName(String name) {
3. Student student = new Student();
4. student.setName(name);
5. List<Student> studentList = session.selectList("selectByName", student);
6. return studentList;
7. }

每 一 个 MyBatis 的 应 用 程 序 都 以 一 个 SqlSessionFactory 对 象 的 实 例 为 核 心 。SqlSessionFactory本身是由SqlSessionFactoryBuilder创建的，一般而言，在一个应用中，一个数据库只会对应一 个SqlSessionFactory，所以一般我们都把SqlSessionFactory定义成单例模式，或通过Spring等进行注入。

SqlSessionFactoryBuilder创建SqlSessionFactory的方法有：

Java代码  [收藏代码](javascript:void())

1. SqlSessionFactory build(InputStream inputStream)
2. SqlSessionFactory build(InputStream inputStream, String environment)
3. SqlSessionFactory build(InputStream inputStream, Properties properties)
4. SqlSessionFactory build(InputStream inputStream, String env, Properties props)
5. SqlSessionFactory build(Configuration config)

这些方法主要设计到的参数有InputStream，environment，properties，其中InputStream是从配置文件中获 取的一个输入流；environment表示在配置文件里面配置的众多的environment中，当前要使用的是哪一个environment，包括数 据源和事务，缺省则使用默认的environment；使用properties，MyBatis则会加载对应的属性或文件，它们可以在配置文件中使 用。

 从XML中构建SqlSessionFactory

 Java代码  [收藏代码](javascript:void())

1. private static SqlSessionFactory sqlSessionFactory = null;
3. static {
4. try {
5. InputStream is = Resources.getResourceAsStream("config/mybatis\_config.xml");
6. sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);
7. } catch (IOException e) {
8. // TODO Auto-generated catch block
9. e.printStackTrace();
10. }
12. }
14. public static SqlSessionFactory getSqlSessionFactory() {
15. return sqlSessionFactory;
16. }

下面讲讲配置文件的基本结构：

mybatis的配置文件一般包括如下几个部分：

properties：properties用于定义或导入属性，然后在后面的环境中使用

settings:settings用于设置一些mybatis在运行时的行为方式，具体的设置信息可以查看mybatis的文档

typeAliases:typeAliases是为系统中的Java类型指定一个较短的别名

environments:MyBatis 可以配置多种环境。这会帮助你将 SQL 映射应用于多种数据库之中。

 ml代码  [收藏代码](javascript:void())

1. <environments default="development">
2. <environment id="development">
3. <transactionManager type="JDBC" />
4. <dataSource type="POOLED">
5. <property name="driver" value="${jdbc.driver}" />
6. <property name="url" value="${jdbc.url}" />
7. <property name="username" value="${jdbc.username}" />
8. <property name="password" value="${jdbc.password}" />
9. </dataSource>
10. </environment>
11. </environments>

由于MyBatis可以配置多个environment，所以可以在创建SqlSessionFactory的时候指定具体的环境来创建特定的环境下的SqlSessionFactory，  不指定则使用默认的环境。

**transactionManager**

在 MyBatis 中有两种事务管理器类型(也就是 type=”[JDBC|MANAGED]”):

* JDBC – 这个配置直接简单使用了 JDBC 的提交和回滚设置。 它依赖于从数据源得 到的连接来管理事务范围。
* MANAGED – 这个配置几乎没做什么。它从来不提交或回滚一个连接。而它会让 容器来管理事务的整个生命周期(比如 Spring 或 JEE 应用服务器的上下文) 默认 情况下它会关闭连接。 然而一些容器并不希望这样, 因此如果你需要从连接中停止 它,将 closeConnection 属性设置为 false。

**dataSource**

dataSource 元素使用基本的 JDBC 数据源接口来配置 JDBC 连接对象的资源。

* 许多 MyBatis 的应用程序将会按示例中的例子来配置数据源。 然而它并不是必须的。 要知道为了方便使用延迟加载,数据源才是必须的。

有三种内建的数据源类型(也就是 type=”???”):

**UNPOOLED** – 这个数据源的实现是每次被请求时简单打开和关闭连接。它有一点慢, 这是对简单应用程序的一个很好的选择, 因为它不需要及时的可用连接。 不同的数据库对这 个的表现也是不一样的, 所以对某些数据库来说配置数据源并不重要, 这个配置也是闲置的。 UNPOOLED 类型的数据源仅仅用来配置以下 5 种属性:

* driver – 这是 JDBC 驱动的 Java 类的完全限定名(如果你的驱动包含,它也不是 数据源类)。
* url – 这是数据库的 JDBC URL 地址。
* username – 登录数据库的用户名。
* password – 登录数据库的密码。
* defaultTransactionIsolationLevel – 默认的连接事务隔离级别。

作为可选项,你可以传递数据库驱动的属性。要这样做,属性的前缀是以“driver.”开 头的,例如:

* driver.encoding=UTF8

这 样 就 会 传 递 以 值 “ UTF8 ” 来 传 递 属 性 “ encoding ”, 它 是 通 过 DriverManager.getConnection(url,driverProperties)方法传递给数据库驱动。

**POOLED** – 这是 JDBC 连接对象的数据源连接池的实现,用来避免创建新的连接实例 时必要的初始连接和认证时间。这是一种当前 Web 应用程序用来快速响应请求很流行的方 法。

除了上述(UNPOOLED)的属性之外,还有很多属性可以用来配置 POOLED 数据源:

* poolMaximumActiveConnections – 在任意时间存在的活动(也就是正在使用)连 接的数量。默认值:10
* poolMaximumIdleConnections – 任意时间存在的空闲连接数。
* poolMaximumCheckoutTime – 在被强制返回之前,池中连接被检查的时间。默认 值:20000 毫秒(也就是 20 秒)
* poolTimeToWait – 这是给连接池一个打印日志状态机会的低层次设置,还有重新 尝试获得连接, 这些情况下往往需要很长时间 为了避免连接池没有配置时静默失 败)。默认值:20000 毫秒(也就是 20 秒)
* poolPingQuery – 发送到数据的侦测查询,用来验证连接是否正常工作,并且准备 接受请求。默认是“NO PING QUERY SET” ,这会引起许多数据库驱动连接由一 个错误信息而导致失败。
* poolPingEnabled – 这是开启或禁用侦测查询。如果开启,你必须用一个合法的 SQL 语句(最好是很快速的)设置 poolPingQuery 属性。默认值:false。
* poolPingConnectionsNotUsedFor – 这是用来配置 poolPingQuery 多次时间被用一次。 这可以被设置匹配标准的数据库连接超时时间, 来避免不必要的侦测。 默认值: 0(也就是所有连接每一时刻都被侦测-但仅仅当 poolPingEnabled 为 true 时适用)。

**JNDI** – 这个数据源的实现是为了使用如 Spring 或应用服务器这类的容器, 容器可以集 中或在外部配置数据源,然后放置一个 JNDI 上下文的引用。这个数据源配置只需要两个属 性:

* initial\_context – 这 个 属 性 用 来 从 初 始 上 下 文 中 寻 找 环 境 ( 也 就 是 initialContext.lookup(initial——context) 。这是个可选属性,如果被忽略,那么 data\_source 属性将会直接以 initialContext 为背景再次寻找。
* data\_source – 这是引用数据源实例位置的上下文的路径。它会以由 initial\_context 查询返回的环境为背景来查找,如果 initial\_context 没有返回结果时,直接以初始 上下文为环境来查找。

再之后就是Mapper了，Mapper就是映射SQL语句的，首先要告诉mybatis要到哪里去找这些SQL语句，即指定资源位置。

Java代码  [收藏代码](javascript:void())

1. <mappers>
2. <mapper resource="com/tiantian/mybatis/model/BlogMapper.xml"/>
3. </mappers>

下面是我在测试过程中的一个简单的配置文件：

Xml代码  [收藏代码](javascript:void())

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE configuration
3. PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4. "http://mybatis.org/dtd/mybatis-3-config.dtd">
5. <configuration>
6. <properties resource="config/jdbc.properties"></properties>
7. <typeAliases>
8. <typeAlias alias="Blog" type="com.tiantian.mybatis.model.Blog"/>
9. </typeAliases>
10. <environments default="development">
11. <environment id="development">
12. <transactionManager type="JDBC" />
13. <dataSource type="POOLED">
14. <property name="driver" value="${jdbc.driver}" />
15. <property name="url" value="${jdbc.url}" />
16. <property name="username" value="${jdbc.username}" />
17. <property name="password" value="${jdbc.password}" />
18. </dataSource>
19. </environment>
20. </environments>
21. <mappers>
22. <mapper resource="com/tiantian/mybatis/model/BlogMapper.xml"/>
23. </mappers>
24. </configuration>

在上面配置文件中导入了一个外部的属性文件，MyBatis配置文件中的属性引入可以是直接包含在properties元素中的，也可以是利用 properties元素从外部引入的，还可以是在创建SqlSessionFactory的时候，作为一个参数properties传入。既然 MyBatis配置文件中的属性可以从这么多地方引入，那就牵涉到一个优先级的问题，MyBatis将会按照下面的顺序来寻找它们：

* 先是配置文件中，properties元素体中的属性被读取
* 再是利用properties元素从外部引入的属性文件中的属性被读取，会覆盖前面读取的相同的属性
* 最后是创建SqlSessionFactory时传入的properties中的属性被读取，同样会覆盖前面相同的属性

在有了SqlSessionFactory之后就是获取特定的SqlSession了，在使用mybatis的过程中每一个操作都是离不开 SqlSession的，所以获取SqlSession是相当重要的。此外，SqlSession是不能被共享、线程不安全的，所以在每次需要 SqlSession的时候都应该打开一个，然后在用完了之后再把它关上。

Java代码  [收藏代码](javascript:void())

1. SqlSession session = sqlSessionFactory.openSession();

SqlSessionFactory中湖区SqlSession的方法有：

Java代码  [收藏代码](javascript:void())

1. SqlSession openSession()
2. SqlSession openSession(boolean autoCommit)
3. SqlSession openSession(Connection connection)
4. SqlSession openSession(TransactionIsolationLevel level)
5. SqlSession openSession(ExecutorType execType,TransactionIsolationLevel level)
6. SqlSession openSession(ExecutorType execType)
7. SqlSession openSession(ExecutorType execType, boolean autoCommit)
8. SqlSession openSession(ExecutorType execType, Connection connection)
9. Configuration getConfiguration();

它们的主要区别在于：

* **Transaction (事务)**: 你想为 session 使用事务或者使用自动提交
* **Connection (连接)**: 你想 MyBatis 获得来自配置的数据源的连接还是提供你自己
* **Execution (执行)**: 你想 MyBatis 复用预处理语句和/或批量更新语句(包括插入和 删除)

 默认的opensession方法没有参数，它会创建有如下特性的SqlSession：

* 会开启一个事务，也就是不自动提交
* 连接对象会从当前正在使用的environment中的数据源中得到
* 事务隔离级别将会使用驱动或数据源的默认值
* 预处理语句不会被复用，也不会批量更新语句

ExecutorType有三个值：

* ExecutorType.SIMPLE 它会为每个语句的执行创建一个新的预处理语句
* ExecutorType.REUSE 它会复用预处理语句
* ExecutorType.BATCH 这个执行器会批量执行更新语句

mybatis的基本操作就是增、删、改、查，即insert、delete、update和select。在进行这些基本的操作的时候可以直 接利用SqlSession访问Mapper配置文件里面的映射来进行，也可以利用与Mapper配置文件相对应的Mapper接口来进行操作，条件是 Mapper接口中定义的方法的参数和返回值要与Mapper配置文件中定义的参数和返回值相同。此外，在使用Mapper接口的时候，对应的SQL语句 是可以写在Mapper配置文件中的，也可以直接利用对应的注解在Mapper接口中对应的方法上进行标明，这将在下面的示例代码中看到。

下面是一系列的示例代码：

先贴一个用于获取SqlSessionFactory的工具类：

Java代码  [收藏代码](javascript:void())

1. import java.io.IOException;
2. import java.io.InputStream;
4. import org.apache.ibatis.io.Resources;
5. import org.apache.ibatis.session.SqlSessionFactory;
6. import org.apache.ibatis.session.SqlSessionFactoryBuilder;
8. public class Util {
10. private static SqlSessionFactory sqlSessionFactory = null;
12. static {
13. try {
14. InputStream is = Resources.getResourceAsStream("config/mybatis\_config.xml");
15. sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);
16. } catch (IOException e) {
17. // TODO Auto-generated catch block
18. e.printStackTrace();
19. }
21. }
23. public static SqlSessionFactory getSqlSessionFactory() {
24. return sqlSessionFactory;
25. }
27. }

mybatis的配置文件：

Xml代码  [收藏代码](javascript:void())

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE configuration
3. PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4. "http://mybatis.org/dtd/mybatis-3-config.dtd">
5. <configuration>
6. <properties resource="config/jdbc.properties"></properties>
7. <typeAliases>
8. <typeAlias alias="Blog" type="com.tiantian.mybatis.model.Blog"/>
9. </typeAliases>
10. <environments default="development">
11. <environment id="development">
12. <transactionManager type="JDBC" />
13. <dataSource type="POOLED">
14. <property name="driver" value="${jdbc.driver}" />
15. <property name="url" value="${jdbc.url}" />
16. <property name="username" value="${jdbc.username}" />
17. <property name="password" value="${jdbc.password}" />
18. </dataSource>
19. </environment>
20. </environments>
21. <mappers>
22. <mapper resource="com/tiantian/mybatis/model/BlogMapper.xml"/>
23. </mappers>
24. </configuration>

BlogMapper.xml

Xml代码  [收藏代码](javascript:void())

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE mapper
3. PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4. "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
6. <mapper namespace="com.tiantian.mybatis.model.BlogMapper">
7. <!--  新增记录  -->
8. <insert id="insertBlog" parameterType="Blog">
9. insert into t\_blog(title,content,owner) values(#{title},#{content},#{owner})
10. </insert>
11. <!--  查询单条记录 -->
12. <select id="selectBlog" parameterType="int" resultType="Blog">
13. select \* from t\_blog where id = #{id}
14. </select>
15. <!--  修改记录   -->
16. <update id="updateBlog" parameterType="Blog">
17. update t\_blog set title = #{title},content = #{content},owner = #{owner} where id = #{id}
18. </update>
19. <!--  查询所有记录，查询多条记录即返回结果是一个集合的时候，resultType不是集合类型，而是集合所包含的类型 -->
20. <select id="selectAll" resultType="Blog">
21. select \* from t\_blog
22. </select>
23. <!--  模糊查询   -->
24. <select id="fuzzyQuery" resultType="Blog" parameterType="java.lang.String">
25. select \* from t\_blog where title like "%"#{title}"%"
26. </select>
27. <!--  删除记录   -->
28. <delete id="deleteBlog" parameterType="int">
29. delete from t\_blog where id = #{id}
30. </delete>
31. </mapper>

**SQL映射语句中一些应该注意的问题：**

1. resultType的值应该是返回类型的完全名或别名，当返回的结果是一个集合的时候，resultType应为集合中所包含的类型，而不是集合类型，如上面的Blog
2. resultType和resultMap都是表示指定返回结果的，但两者不能同时使用
3. 对于Insert映射语句有一个useGeneratedKeys属性，该属性的**默认值为false，**当该属性的值为true时，在进行插入操作时，mybatis会取到当前正在插入的记录在数据库中的自动递增的主键值，并把它设置给指定的实体的属性，这就需要设置一个keyProperty属性，用于指定实体中表示主键的属性

Blog.java

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.mybatis.model;
3. public class Blog {
5. private int id;
7. private String title;
9. private String content;
11. private String owner;
13. public int getId() {
14. return id;
15. }
17. public void setId(int id) {
18. this.id = id;
19. }
21. public String getTitle() {
22. return title;
23. }
25. public void setTitle(String title) {
26. this.title = title;
27. }
29. public String getContent() {
30. return content;
31. }
33. public void setContent(String content) {
34. this.content = content;
35. }
37. public String getOwner() {
38. return owner;
39. }
41. public void setOwner(String owner) {
42. this.owner = owner;
43. }
45. @Override
46. public String toString() {
47. return "id: " + id + ", title: " + title + ", content: " + content
48. + ", owner: " + owner;
49. }
51. }

BlogMapper.java

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.mybatis.model;
3. import java.util.List;
5. import org.apache.ibatis.annotations.Delete;
6. import org.apache.ibatis.annotations.Insert;
7. import org.apache.ibatis.annotations.Select;
8. import org.apache.ibatis.annotations.Update;
10. /\*\*
11. \* 以下的操作1都是把SQL写在配置文件里面的，而操作2都是直接用注解标明要执行的SQL语句
12. \* 因为该Mapper的全名跟BlogMapper.xml文件里面的namespace是一样的，所以不能在这里面
13. \* 用注解定义一个与BlogMapper.xml文件里面同名的映射
14. \* @author andy
15. \*
16. \*/
17. public interface BlogMapper {
19. public Blog selectBlog(int id);
21. @Select("select \* from t\_blog where id = #{id}")
22. public Blog selectBlog2(int id);
24. public void insertBlog(Blog blog);
26. @Insert("insert into t\_blog(title,content,owner) values(#{title},#{content},#{owner})")
27. public void insertBlog2(Blog blog);
29. public void updateBlog(Blog blog);
31. @Update("update t\_blog set title=#{title},content=#{content},owner=#{owner} where id=#{id}")
32. public void updateBlog2(Blog blog);
34. public void deleteBlog(int id);
36. @Delete("delete from t\_blog where id = #{id}")
37. public void deleteBlog2(int id);
39. public List<Blog> selectAll();
41. @Select("select \* from t\_blog")
42. public List<Blog> selectAll2();
44. public List<Blog> fuzzyQuery(String title);
46. @Select("select \* from t\_blog where title like \"%\"#{title}\"%\"")
47. public List<Blog> fuzzyQuery2(String title);
49. }

Test1.java

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.mybatis.test;
3. import java.util.List;
5. import org.apache.ibatis.session.SqlSession;
6. import org.junit.Test;
8. import com.tiantian.mybatis.model.Blog;
9. import com.tiantian.mybatis.util.Util;
11. /\*\*
12. \* 该系列操作是通过把SQL写在配置文件里面，
13. \* 然后利用SqlSession进行操作的
14. \* @author andy
15. \*
16. \*/
17. public class Test1 {
19. /\*\*
20. \* 新增记录
21. \*/
22. @Test
23. public void testInsertBlog() {
24. SqlSession session = Util.getSqlSessionFactory().openSession();
25. Blog blog = new Blog();
26. blog.setTitle("中国人");
27. blog.setContent("五千年的风和雨啊藏了多少梦");
28. blog.setOwner("天天");
29. session.insert("com.tiantian.mybatis.model.BlogMapper.insertBlog", blog);
30. session.commit();
31. session.close();
32. }
34. /\*\*
35. \* 查询单条记录
36. \*/
37. @Test
38. public void testSelectOne() {
39. SqlSession session = Util.getSqlSessionFactory().openSession();
40. Blog blog = (Blog)session.selectOne("com.tiantian.mybatis.model.BlogMapper.selectBlog", 8);
41. System.out.println(blog);
42. session.close();
43. }
45. /\*\*
46. \* 修改记录
47. \*/
48. @Test
49. public void testUpdateBlog() {
50. SqlSession session = Util.getSqlSessionFactory().openSession();
51. Blog blog = new Blog();
52. blog.setId(7);//需要修改的Blog的id
53. blog.setTitle("中国人2");//修改Title
54. blog.setContent("黄色的脸，黑色的眼，不变是笑容");//修改Content
55. blog.setOwner("天天2");//修改Owner
56. session.update("com.tiantian.mybatis.model.BlogMapper.updateBlog", blog);
57. session.commit();
58. session.close();
59. }
61. /\*\*
62. \* 查询所有的记录
63. \*/
64. @Test
65. public void testSelectAll() {
66. SqlSession session = Util.getSqlSessionFactory().openSession();
67. List<Blog> blogs = session.selectList("com.tiantian.mybatis.model.BlogMapper.selectAll");
68. for (Blog blog:blogs)
69. System.out.println(blog);
70. session.close();
71. }
73. /\*\*
74. \* 模糊查询
75. \*/
76. @Test
77. public void testFuzzyQuery() {
78. SqlSession session = Util.getSqlSessionFactory().openSession();
79. String title = "中国";
80. List<Blog> blogs = session.selectList("com.tiantian.mybatis.model.BlogMapper.fuzzyQuery", title);
81. for (Blog blog:blogs)
82. System.out.println(blog);
83. session.close();
84. }
86. /\*\*
87. \* 删除记录
88. \*/
89. @Test
90. public void testDeleteBlog() {
91. SqlSession session = Util.getSqlSessionFactory().openSession();
92. session.delete("com.tiantian.mybatis.model.BlogMapper.deleteBlog", 8);
93. session.commit();
94. session.close();
95. }
97. }

Test2.java

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.mybatis.test;
3. import java.util.List;
5. import org.apache.ibatis.session.SqlSession;
6. import org.junit.Test;
8. import com.tiantian.mybatis.model.Blog;
9. import com.tiantian.mybatis.model.BlogMapper;
10. import com.tiantian.mybatis.util.Util;
12. /\*\*
13. \* 该系列操作是将SQL语句写在配置文件里面，
14. \* 然后通过对应Mapper接口来进行操作的
15. \* @author andy
16. \*
17. \*/
18. public class Test2 {
20. /\*\*
21. \* 新增记录
22. \*/
23. @Test
24. public void testInsertBlog() {
25. SqlSession session = Util.getSqlSessionFactory().openSession();
26. Blog blog = new Blog();
27. blog.setTitle("中国人");
28. blog.setContent("五千年的风和雨啊藏了多少梦");
29. blog.setOwner("天天");
30. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
31. blogMapper.insertBlog(blog);
32. session.commit();
33. session.close();
34. }
36. /\*\*
37. \* 查询单条记录
38. \*/
39. @Test
40. public void testSelectOne() {
41. SqlSession session = Util.getSqlSessionFactory().openSession();
42. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
43. Blog blog = blogMapper.selectBlog(7);
44. System.out.println(blog);
45. session.close();
46. }
48. /\*\*
49. \* 修改记录
50. \*/
51. @Test
52. public void testUpdateBlog() {
53. SqlSession session = Util.getSqlSessionFactory().openSession();
54. Blog blog = new Blog();
55. blog.setId(9);// 需要修改的Blog的id
56. blog.setTitle("中国人2");// 修改Title
57. blog.setContent("黄色的脸，黑色的眼，不变是笑容");// 修改Content
58. blog.setOwner("天天2");// 修改Owner
59. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
60. blogMapper.updateBlog(blog);
61. session.commit();
62. session.close();
63. }
65. /\*\*
66. \* 查询所有记录
67. \*/
68. @Test
69. public void testSelectAll() {
70. SqlSession session = Util.getSqlSessionFactory().openSession();
71. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
72. List<Blog> blogs = blogMapper.selectAll();
73. for (Blog blog : blogs)
74. System.out.println(blog);
75. session.close();
76. }
78. /\*\*
79. \* 模糊查询
80. \*/
81. @Test
82. public void testFuzzyQuery() {
83. SqlSession session = Util.getSqlSessionFactory().openSession();
84. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
85. String title = "中国";
86. List<Blog> blogs = blogMapper.fuzzyQuery(title);
87. for (Blog blog : blogs)
88. System.out.println(blog);
89. session.close();
90. }
92. /\*\*
93. \* 删除记录
94. \*/
95. @Test
96. public void testDeleteBlog() {
97. SqlSession session = Util.getSqlSessionFactory().openSession();
98. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
99. blogMapper.deleteBlog(10);
100. session.commit();
101. session.close();
102. }
104. }

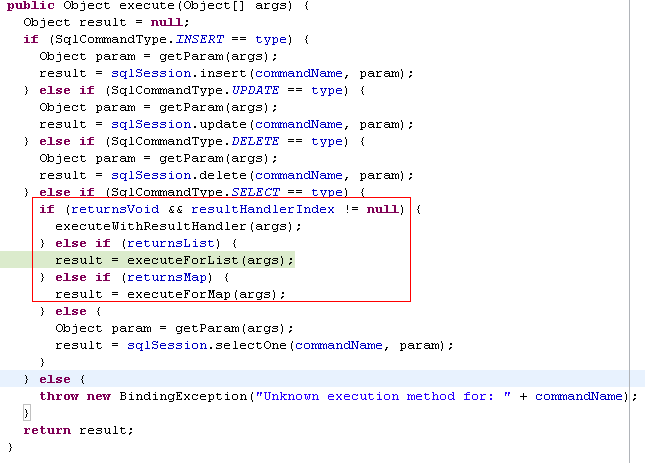
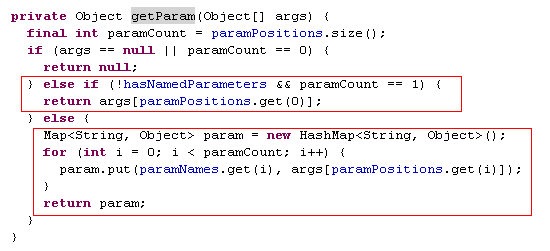
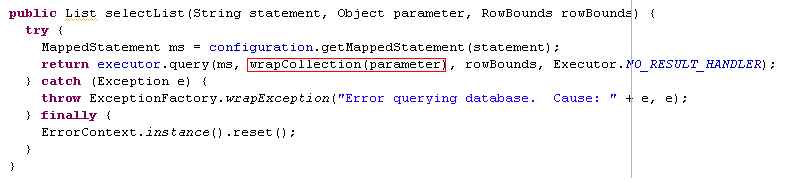
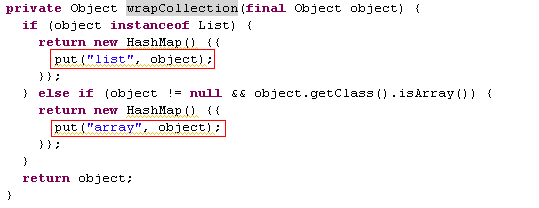
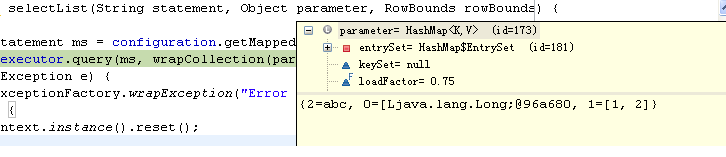
Test3.java

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.mybatis.test;
3. import java.util.List;
5. import org.apache.ibatis.session.SqlSession;
6. import org.junit.Test;
8. import com.tiantian.mybatis.model.Blog;
9. import com.tiantian.mybatis.model.BlogMapper;
10. import com.tiantian.mybatis.util.Util;
12. /\*\*
13. \* 该系列操作是利用Mapper接口来进行的
14. \* ，然而其相应的SQL语句是通过对应的
15. \* 注解Annotation在Mapper中对应的方法上定义的
16. \* @author andy
17. \*
18. \*/
19. public class Test3 {
21. /\*\*
22. \* 新增记录
23. \*/
24. @Test
25. public void testInsert() {
26. SqlSession session = Util.getSqlSessionFactory().openSession();
27. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
28. Blog blog = new Blog();
29. blog.setTitle("title2");
30. blog.setContent("content2");
31. blog.setOwner("owner2");
32. blogMapper.insertBlog2(blog);
33. session.commit();
34. session.close();
35. }
37. /\*\*
38. \* 查找单条记录
39. \*/
40. @Test
41. public void testSelectOne() {
42. SqlSession session = Util.getSqlSessionFactory().openSession();
43. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
44. Blog blog = blogMapper.selectBlog2(1);
45. System.out.println(blog);
46. session.close();
47. }
49. /\*\*
50. \* 查找多条记录，返回结果为一集合
51. \*/
52. @Test
53. public void testSelectAll() {
54. SqlSession session = Util.getSqlSessionFactory().openSession();
55. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
56. List<Blog> blogs = blogMapper.selectAll2();
57. for (Blog blog:blogs)
58. System.out.println(blog);
59. session.close();
60. }
62. /\*\*
63. \* 修改某条记录
64. \*/
65. @Test
66. public void testUpdate() {
67. SqlSession session = Util.getSqlSessionFactory().openSession();
68. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
69. Blog blog = new Blog();
70. blog.setId(3);
71. blog.setTitle("title3");
72. blog.setContent("content3");
73. blog.setOwner("owner3");
74. blogMapper.updateBlog2(blog);
75. session.commit();
76. session.close();
77. }
79. /\*\*
80. \* 删除记录
81. \*/
82. @Test
83. public void testDelete() {
84. SqlSession session = Util.getSqlSessionFactory().openSession();
85. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
86. blogMapper.deleteBlog2(5);
87. session.commit();
88. session.close();
89. }
91. @Test
92. public void testFuzzyQuery() {
93. SqlSession session = Util.getSqlSessionFactory().openSession();
94. BlogMapper blogMapper = session.getMapper(BlogMapper.class);
95. List<Blog> blogs = blogMapper.fuzzyQuery2("中国");
96. for (Blog blog:blogs)
97. System.out.println(blog);
98. session.close();
99. }
101. }

对应的建表语句：

Java代码  [收藏代码](javascript:void())

1. CREATE TABLE `t\_blog` (
2. `id` int(11) NOT NULL AUTO\_INCREMENT,
3. `title` varchar(255) DEFAULT NULL,
4. `content` varchar(255) DEFAULT NULL,
5. `owner` varchar(50) DEFAULT NULL,
6. PRIMARY KEY (`id`)
7. )
8. ，一是应该用单引号把%#{userName}%括起来，二是不能用#号，而是改成$号，所以正确的配置应该是 and user\_name like '%${userName}%'。
9. 第二个思路是，配置文件不该，还为and user\_name like #{userName},而在程序里做手脚，如在给userName赋值的setUserName方法中，使用成this.userName="'%"+userName+"%'"这样
10. 两种方式都可以
11. 但是推荐使用第一种方式，这样不用修改pojo类，不会造成业务逻辑错误
12. **1. sql字符串拼接(1)**
13. SELECT \* FROM tableName WHERE name LIKE '%' || #{text} || '%';
14. **2. sql中字符串拼接(2)**
15. SELECT \* FROM tableName WHERE name LIKE CONCAT(CONCAT('%', #{text}), '%');
16. **3. 使用 ${...} 代替 #{...}**
17. SELECT \* FROM tableName WHERE name LIKE '%${text}%';
18. **4. 程序中拼接**
19. **Java**
20. // or String searchText = "%" + text + "%";
21. String searchText = new StringBuilder("%").append(text).append("%").toString();
22. parameterMap.put("text", searchText);
24. **SqlMap.xml**
25. SELECT \* FROM tableName WHERE name LIKE #{text};
26. [源码解读Mybatis List列表In查询实现的注意事项](http://www.blogjava.net/xmatthew/archive/2011/08/31/355879.html)
27. 在SQL开发过程中，动态构建In集合条件查询是比较常见的用法，在Mybatis中提供了*foreach*功能，该功能比较强大，它允许你指定一个集合，声明集合项和索引变量，它们可以用在元素体内。它也允许你指定开放和关闭的字符串，在迭代之间放置分隔符。这个元素是很智能的，它不会偶然地附加多余的分隔符。下面是一个演示示例：
28. <select id="findByIdsMap" resultMap="BaseResultMap">
29. Select
30. <include refid="Base\_Column\_List" />
31. from jria where ID in
32. **<foreach item="item" index="index" collection="list"**
33. **open="(" separator="," close=")">**
34. **#{item}**
35. **</foreach>**
36. </select>
37. 但由于官方文档对这块的使用，描述的比较简短，细节上也被忽略掉了(可能是开源项目文档一贯的问题吧)，也使用不少同学在使用中遇到了问题。特别是foreach这个函数中，collection属性做什么用，有什么注意事项。由于文档不全，这块只能通过源代码剖析的方式来分析一下各个属性的相关要求。
38. collection属性的用途是接收输入的数组或是List接口实现。但对于其名称的要求，Mybatis在实现中还是有点不好理解的，所以需要特别注意这一点。
39. 下面开始分析源代码(笔记使用的是Mybatis 3.0.5版本)
40. 先找到Mybatis执行SQL配置解析的入口
41. MapperMethod.java类中 **public** Object execute(Object[] args) 该方法是执行的入口.
42. 针对in集合查询，对应用就是 selectForList或SelctForMap方法。
43. 
44. 但不管调用哪个方法，都会对原来JDK传入的参数 Object[]类型，通过 getParam方法转换成一个Object,那这个方法是做什么的呢？分析源码如下：
45. 
46. 上 图中标红的两处，很惊讶的发现，一个参数与多个参数的处理方式是不同的(后续很多同学遇到的问题，就有一大部分出自这个地方)。如果参数个数大于一个，则 会被封装成Map, key值如果使用了Mybatis的 Param注解，则会使用该key值，否则默认统一使用数据序号,从1开始。这个问题先记下，继续分析代码，接下来如果是selectForList操作 (其它操作就对应用相应方法),会调用DefaultSqlSession的**public** List selectList(String statement, Object parameter, RowBounds rowBounds) 方法
47. 又一个发现，见源代码如下：
48. 
49. 上图标红部分，对参数又做了一次封装，我们看一下代码
50. 
51. 现在有点清楚了，如果参数类型是List,则必须在collecion中指定为list, 如果是数据组，则必须在collection属性中指定为 array.
52. 现在就问题就比较清楚了，如果是一个参数的话，collection的值取决于你的参数类型。
53. 如果是多个值的话，除非使用注解Param指定，否则都是数字开头，所以在collection中指定什么值都是无用的。下图是debug显示结果。
54. 
55. 针对上面分析的结果，下面给出了一个使用的解决方案，希望对大家对帮助。
56. **在使用这个功能是需要特别注意以下规则：**
57. 1. 当查询的参数只有一个时
58. findByIds(List<Long> ids)
59. 1.a 如果参数的类型是List, 则在使用时，collection属性要必须指定为 list
60. <select id="findByIdsMap" resultMap="BaseResultMap">
61. Select
62. <include refid="Base\_Column\_List" />
63. from jria where ID in  
                      **<foreach item="item" index="index" collection="list"   
                             open="(" separator="," close=")">  
                            #{item}  
                    </foreach>**  
      </select>
65. findByIds(Long[] ids)
66. 1.b 如果参数的类型是Array,则在使用时，collection属性要必须指定为 array
67. <select id="findByIdsMap" resultMap="BaseResultMap">  
                     select  
                     <include refid="Base\_Column\_List" />  
              from jria where ID in  
                      **<foreach item="item" index="index" collection="array"   
                             open="(" separator="," close=")">  
                            #{item}  
                    </foreach>**  
      </select>
69. 2. 当查询的参数有多个时,例如 findByIds(String name, Long[] ids)
70. 这种情况需要特别注意，在传参数时，一定要改用Map方式, 这样在collection属性可以指定名称
71. 下面是一个示例
72. Map<String, Object> params = new HashMap<String, Object>(2);
73. params.put("name", name);
74. params.put("ids", ids);
75. mapper.findByIdsMap(params);
77. <select id="findByIdsMap" resultMap="BaseResultMap">  
                     select  
                     <include refid="Base\_Column\_List" />  
              from jria where ID in  
                      **<foreach item="item" index="index" collection="ids"   
                             open="(" separator="," close=")">  
                            #{item}  
                    </foreach>**  
       </select>

80. 完整的示例如下:
81. 例如有一个查询功能，Mapper接口文件定义如下方法:
82. List<Jria> findByIds(Long... ids);
83. 使用 in 查询的sql拼装方法如下:
84. <select id="findbyIds" resultMap="BaseResultMap">  
                     select  
                     <include refid="Base\_Column\_List" />  
              from jria where ID in  
                      **<foreach item="item" index="index" collection="array"   
                             open="(" separator="," close=")">  
                            #{item}  
                    </foreach>**  
      </select>
85. 例如有一个查询功能，Mapper接口文件定义如下方法:
86. List<Jria>findByIds(Long... ids);
87. 使用 in 查询的sql拼装方法如下:
88. <select id="findbyIds" resultMap="BaseResultMap">
89. select<include refid="Base\_Column\_List" />
90. from jriawhere ID in
91. <foreach item="item" index="index"collection="array" open="(" separator=","close=")">
92. #{item}
93. </foreach>
94. </select>
95. where";
96. repend="AND" property="peopletype"