Mybatis笔记

一、第一节——基础配置和DQL

1.需要用到的jar包。

1）mybatis-3.3.0.jar （mybatis核心包）

2）mysql-connector-java-5.1.26-bin.jar（jdbc驱动包）

2.db.properties。数据源配置文件

1）在根目录下创建db.properties。

2）内容如下：

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver  jdbc.url=jdbc\:mysql\://localhsot\:3306/blog\_gp1701?useUnicode\=true&characterEncoding\=UTF-8  jdbc.username=root  jdbc.password= |

3.mybatis-config.xml。Mybatis配置文件

1）在根目录下创建mybatis-config.xml。

2）内容如下：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!-- 引入数据库连接配置 -->  <properties resource=*"db.properties"*></properties>  <!-- 配置类的别名 -->  <typeAliases>  <!-- <typeAlias type="com.zhangfd.mybatis.pojo.Blog" alias="Blog"/> -->  <package name=*"com.zhangfd.mybatis.pojo"*/>  </typeAliases>  <!-- 定义数据源 -->  <environments default=*"development"*>  <environment id=*"development"*>  <transactionManager type=*"JDBC"*/>  <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>  </environment>  <!-- <environment id="test"></environment>  <environment id="production"></environment> -->  </environments>    <!-- 映射文件 -->  <mappers>  <!-- <mapper resource="com/zhangfd/mybatis/mapper/BlogMapper.xml" /> -->  <package name=*"com.zhangfd.mybatis.mapper"*/>  </mappers>  </configuration> |

4.mapper.xml。DAO层映射文件

1）创建文件，一个DAO接口对应一个一个mapper.xml。

2)文件内容：

|  |
| --- |
| <?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">  <!-- namespace是接口的名字，相当于实现类 -->  <mapper namespace=*"com.zhangfd.mybatis.mapper.BlogMapper"*>  <!-- 一个语句就是一个statement -->  <!-- id是方法名 -->  <select id=*"selectBlog"* parameterType=*"int"* resultType=*"com.zhangfd.mybatis.pojo.Blog"*>  select \* from Blog where id = #{id}  </select>  </mapper> |

5.编写接口。（略）

6.创建POJO。（略）

7.编写MybatisUtil工具类

具体看F:\eclipseWorkingSpace\mybatis01\src\com\zhangfd\mybatis\util\MybatisUtil.java

8.创建测试用例

可用DAO接口，也可用session的方法直接查mapper，详情见：F:\eclipseWorkingSpace\mybatis01\test\com\zhangfd\mybatis\mapper\ BlogMapperTest.java

9.字段名和pojo成员名不同的处理方法

1）在mapper的查询语句里as区别名。

2）使用ResultMap：

|  |
| --- |
| <?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">  <!-- namespace是接口的名字，相当于实现类 -->  <mapper namespace=*"com.zhangfd.mybatis.mapper.BlogMapper"*>  <!-- 类似SQL 的as取别名 -->  <resultMap type=*"Blog"* id=*"blogResultMap"*>  <id column=*"id"* property=*"id"* jdbcType=*"INTEGER"*/>  <result column=*"author\_id"* property=*"authorId"* jdbcType=*"INTEGER"*/>  </resultMap>  <select id=*"selectBlog2"* parameterType=*"int"* resultMap=*"blogResultMap"*>  select \* from Blog where id = #{id}  </select>  </mapper> |

10.查询list

1）查询list和查model一样，返回值为list的时候，如果有多条记录，会自动转成list。

11.sql中#和$的区别

1）当只有一个参数时用${}里面只能写value，#{}可以任意写。

2）#{}有占位符预编译功能，${}直接字符串拼接.

3）模糊查询时因为#{}有预编译功能可以不用写’’， ${}可以在传参的时候加或者在sql里面加：

方式一：

|  |
| --- |
| List<Blog> blog = blogMapper.selectBlogByTitle2("'%了%'"); |

方式二：

|  |
| --- |
| <select id=*"selectBlogByTitle2"* parameterType=*"string"* resultMap=*"blogResultMap"*>  select \* from blog where title like '%${value}%'  </select> |

4）注意：尽量使用#{}以防止SQL注入，但是如果参数表示表名或列名的时候只能用${}。

12.模糊查询时忽略大小写

可用sql中的函数lower()或upper().

|  |
| --- |
| <select id=*"selectBlogByTitle2"* parameterType=*"string"* resultMap=*"blogResultMap"*>  select \* from blog where lower(title) like lower('%${value}%')  </select> |

13.排序

1）注意！排序的列如果要用参数传入要用${}，即11点后面注意的地方。

2）如果要达到拼音排序的效果，mysql可用函数convert(‘abc’ USING GBK)实现：

|  |
| --- |
| <select id=*"selectBlogBySort"* parameterType=*"string"* resultMap=*"blogResultMap"*>  select \* from blog order by convert(${value} using GBK)  </select> |

二、第二节——DML与动态sql

1.多参数传递

1）mysql分页：

①方式一：~~使用索引~~（不建议）。

按参数的顺序，从0开始。

接口方法：

|  |
| --- |
| List<Blog> selectBlogByPage(Integer offset, Integer pagesize); |

mapper：

|  |
| --- |
| <select id=*"selectBlogByPage"* resultMap=*"blogResultMap"*>  select \* from blog limit #{0},#{1}  </select> |

②方式二：注解的方式（推荐使用）

接口方法：

|  |
| --- |
| List<Blog> selectBlogByPage2(  @Param(value="offset") Integer offset,  @Param(value="pagesize") Integer pagesize); |

mapper：

|  |
| --- |
| <select id=*"selectBlogByPage"* resultMap=*"blogResultMap"*>  select \* from blog limit #{offset},#{pagesize}  </select> |

③方式三：使用MAP（当参数较多的时候，强烈推荐使用！！！）

单元测试：

|  |
| --- |
| @Test  **public** **void** selectBlogByPage3() {    SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);  Map<String, Object> map = **new** HashMap<String, Object>();  map.put("offset", 1);  map.put("pagesize", 2);  List<Blog> blog = blogMapper.selectBlogByPage3(map);  session.close();  System.***out***.println(blog);  } |

接口方法：

|  |
| --- |
| List<Blog> selectBlogByPage3(Map<String, Object> map); |

mapper：

|  |
| --- |
| <select id=*"selectBlogByPage3"* resultMap=*"blogResultMap"*>  select \* from blog limit #{offset},#{pagesize}  </select> |

2.插入功能和获取刚刚插入的id

1）方式一（在statement节点中设置）

在mapper中配置insert节点的属性：**useGeneratedKeys**=*"true"* **keyColumn**=*"id"。*

①单元测试：

|  |
| --- |
| @Test  **public** **void** testInsertBlog() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);  Blog blog = **new** Blog();  **int** count = blogMapper.insertBlog(blog);  session.commit();  session.close();  System.***out***.println(blog);  System.***out***.println("影响条数：" + count);  } |

②接口：

|  |
| --- |
| int insertBlog(Blog blog); |

③mapper：

|  |
| --- |
| <insert id=*"insertBlog"* parameterType=*"Blog"* useGeneratedKeys=*"true"* keyColumn=*"id"*>  INSERT INTO blog VALUES(NULL,#{title},#{authorId},#{state},#{featured},#{style});  </insert> |

2）方式二（推荐使用）：

全局配置，在mybatis-config.xml的settings节点里添加对应的setting：

|  |
| --- |
| <settings>  <setting name=*"logImpl"* value=*"STDOUT\_LOGGING"* />  <setting name=*"useGeneratedKeys"* value=*"true"*/>  </settings> |

3）方式三（适用于没有自增主键的数据库）

在Oracle中：

|  |
| --- |
| <!-- 非自增插入oracle -->  <insert id=*"insertBlogOracle"* parameterType=*"Blog"*>  <selectKey resultType=*"java.lang.Integer"* order=*"BEFORE"* keyProperty=*"id"*>  select seq.nextval as id from dual  </selectKey>  INSERT INTO blog VALUES(NULL,#{title},#{authorId},#{state},#{featured},#{style});  </insert> |

在mysql中也可以，不过不推荐，因为mysql中没有序列，只能自定义一个表+函数来模拟序列，这样会有性能损耗，并且并发不是很好。

3.修改

1）普通修改传进一个对象，与insert类似。过程略。

**注意**，如果没有为对象设置所有的要修改的属性，那么未设置的属性会用成员变量的默认值填充。

2）要部分修改有两种方式：

①方式一。

先查一遍，把对象查出来，在这个对象上修改。

②方式二。

动态sql。详情见：☞*[点我跳转](#部分更新)*☜

4.删除

较简单，过程略。需要注意的是，一定要加where条件，能有效避免删库跑路。

批量删除，详情见：☞*[点我跳转](#批量删除)*☜

5.动态sql之——if

需求：

1.查询已激活的，并且博客的名字是包含某个查询字符串的记录。

2.如果用户没有输入任何的查询字符串，那么久显示所有已激活的博客。

mapper：

|  |
| --- |
| <select id=*"selectActiveBlogByTitle"* parameterType=*"string"* resultMap=*"blogResultMap"*>  select \* from blog  where state='ACTIVE'  <if test=*"value != null and value != ''"*>  and title like '%${value}%'  </if>  </select> |

6. 动态sql之——choose-when-otherwise

需求：

1.查询已激活的。

2.如果用户输入了标题的查询关键字，根据关键字查询。

3.否则根据blog的风格样式查询

4.如果什么都没有输入，则显示推荐的。

mapper:

|  |
| --- |
| <select id=*"selectActiveBlogByTitleOrStyle"* parameterType=*"Blog"* resultMap=*"blogResultMap"*>  select \* from blog  where state='ACTIVE'  <choose>  <when test=*"title != null and title != ''"*>  and lower(title) like lower(#{title})  </when>  <when test=*"style != null and style != ''"*>  and style = #{style}  </when>  <otherwise>  and featured = true  </otherwise>  </choose>  </select> |

7. 动态sql之——where

需求：

多条件查询，根据状态，标题，是否被推荐

问题：

第一个动态条件如果没有的话第二个动态条件前的”**and**”怎么处理？

mapper:

|  |
| --- |
| <select id=*"selectBlogByCondition"* parameterType=*"Blog"*  resultMap=*"blogResultMap"*>  select \* from blog  <where>  <if test=*"state != null and state != ''"*>  state = #{state}  </if>  <if test=*"title != null and title != ''"*>  and lower(title) like lower(#{title})  </if>  <if test=*"featured != null"*>  and featured = #{featured}  </if>  </where>  </select> |

8. 动态sql之——set

实现update部分更新。*[↑返回↑](#点我跳转)*

需求：

按需修改，修改指定的列，未指定的不修改。

mapper:

|  |
| --- |
| <update id=*"updateBlogByCondition"* parameterType=*"Blog"*>  UPDATE  `blog`  <set>  <if test=*"title != null and title != ''"*>`title` = #{title},</if>  <if test=*"authorId != null and authorId != ''"*>`author\_id` = #{authorId},</if>  <if test=*"state != null and state != ''"*>`state` = #{state},</if>  <if test=*"featured != null and featured != ''"*>`featured` = #{featured},</if>  <if test=*"style != null and style != ''"*>`style` = #{style}</if>  </set>  WHERE  `id` = #{id} ;  </update> |

9. 动态sql之——trim

复用7，8的需求。

1）Mapper7-where:

|  |
| --- |
| <select id=*"selectBlogByConditionTrim"* parameterType=*"Blog"*  resultMap=*"blogResultMap"*>  select \* from blog  <trim prefix=*"where"* prefixOverrides=*"and | or"*>  <if test=*"state != null and state != ''"*>  state = #{state}  </if>  <if test=*"title != null and title != ''"*>  and lower(title) like lower(#{title})  </if>  <if test=*"featured != null"*>  and featured = #{featured}  </if>  </trim>  </select> |

2）Mapper8-set:

|  |
| --- |
| <update id=*"updateBlogByConditionTrim"* parameterType=*"Blog"*>  UPDATE  `blog`  <trim prefix=*"set"* prefixOverrides=*","*>  <if test=*"title != null and title != ''"*>`title` = #{title}</if>  <if test=*"authorId != null and authorId != ''"*>,`author\_id` = #{authorId}</if>  <if test=*"state != null and state != ''"*>,`state` = #{state}</if>  <if test=*"featured != null and featured != ''"*>,`featured` = #{featured}</if>  <if test=*"style != null and style != ''"*>,`style` = #{style}</if>  </trim>  WHERE  `id` = #{id} ;  </update> |

10. 动态sql之——foreach

需求：实现批量删除。

Mapper:

|  |
| --- |
| <delete id=*"deleteBlogList"* parameterType=*"list"*>  delete from blog where id in  <foreach collection=*"list"* item=*"item"* open=*"("* close=*")"* separator=*","*>  #{item}  </foreach>  </delete> |

11.sql片段

Sql片段即定义一个可复用的代码段

Mapper:

|  |
| --- |
| <sql id=*"columBase"*>  `id`,  `title`,  `author\_id` as authorId,  `state`,  `featured`,  `style`  </sql>  <select id=*"selectBlog"* parameterType=*"int"* resultType=*"Blog"*>  select  <include refid=*"columBase"*></include>  from  Blog where id = #{id}  </select> |

三、第三节——逆向工程与缓存

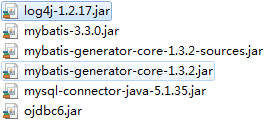
1.逆向工程

1）下载地址

<https://github.com/mybatis/generator/releases/tag/mybatis-generator-1.3.2>

2）工程里包含的内容

①.jar包:



②.入口函数:

|  |
| --- |
| **public** **class** GeneratorSqlmap {  **public** **void** generator() **throws** Exception{  List<String> warnings = **new** ArrayList<String>();  **boolean** overwrite = **true**;  //指定 逆向工程配置文件  File configFile = **new** File("src/generatorConfig.xml");  ConfigurationParser cp = **new** ConfigurationParser(warnings);  Configuration config = cp.parseConfiguration(configFile);  DefaultShellCallback callback = **new** DefaultShellCallback(overwrite);  MyBatisGenerator myBatisGenerator = **new** MyBatisGenerator(config,  callback, warnings);  myBatisGenerator.generate(**null**);  }  **public** **static** **void** main(String[] args) **throws** Exception {  **try** {  GeneratorSqlmap generatorSqlmap = **new** GeneratorSqlmap();  generatorSqlmap.generator();  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

③.generatorConfig.xml配置文件

需要修改的主要配置有：

a.数据库连接信息；

b.pojo,mapper,dao接口生成的位置

c.指定要生成的数据库表

3）xxxExample的用法

xxxExample的方法是修改、查询和删除时附加过滤条件的方法：

例子：

|  |
| --- |
| @Test  **public** **void** testSelectBlog() {    SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    BlogExample example = **new** BlogExample();  BlogExample.Criteria criteria = example.createCriteria();  criteria.andTitleLike("%blog%");  List<Blog> blogList = blogMapper.selectByExample(example);  session.close();    System.***out***.println(blogList);  } |

2.一级缓存

1）默认开启，session级别。

在同一个session范围里执行查询的时候，如果执行相同的查询，那么第二次查询会从缓存中获取数据。

测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogCacheLevelOne1() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    Blog blog = blogMapper.selectBlog(2);  System.***out***.println("结果已查询！");    Blog blog2 = blogMapper.selectBlog(2);  System.***out***.println("结果已查询！从缓存中获取数据");    session.close();  System.***out***.println("session关闭!");  } |

测试结果：只在第一次查询的时候打印sql，第二次直接从缓存中取数据。

2）缓存刷新的情况

如果在同个session范围里执行连续查询两次，第二次会从缓存中取数据。但是，如果在两次查询操作中间执行了一次DML操作，缓存会自动清空，第二次查询会重新查询。

测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogCacheLevelOne2() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    Blog blog = blogMapper.selectBlog(2);  System.***out***.println("结果已查询！");    blog.setFeatured(**true**);  blogMapper.updateBlog(blog);  System.***out***.println("刷新缓存！");    Blog blog2 = blogMapper.selectBlog(2);  System.***out***.println("重新执行查询！");    session.commit();  session.close();  System.***out***.println("session关闭!");  } |

测试结果：打印三次sql。

3.二级缓存

1）二级缓存默认不开启，不同session范围级别。

如果在不同的session范围内，执行相同的数据查询，那么每次查询将会执行独立的数据库检索过程。

测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogCacheLevelTwo1() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    Blog blog = blogMapper.selectBlog(2);  System.***out***.println("结果已查询！");  session.close();    SqlSession session2 = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper2 = session2.getMapper(BlogMapper.**class**);    Blog blog2 = blogMapper2.selectBlog(2);  System.***out***.println("结果已查询！");  session2.close();    System.***out***.println("session关闭!");  } |

结果：输出两次sql。

2）开启二级缓存

**①在对应**Mapper**中配置**：

|  |
| --- |
| <!-- 针对当前表开启二级缓存 -->  <cache/> |

②在对应实体类实现序列化接口：

|  |
| --- |
| public class Blog implements Serializable { |

③测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogCacheLevelTwo1() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    Blog blog = blogMapper.selectBlog(2);  System.***out***.println("结果已查询！");  session.close();    SqlSession session2 = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper2 = session2.getMapper(BlogMapper.**class**);    Blog blog2 = blogMapper2.selectBlog(2);  System.***out***.println("结果已查询！从缓存中获取数据，缓存命中率：0.5");  session2.close();    System.***out***.println("session关闭!");  } |

3）缓存刷新和一级缓存一样

四、第四节——多对一，一对多

1.高级结果映射——嵌套查询（适用于一对一、多对一的情况）

执行流程：

1.先执行一次单表查询

2.再利用单表查询的结果继续执行其他单表查询

3.最后组装结果映射。

1)一对一（对象持有查询）

①修改pojo

使Blog类持有Author对象。

|  |
| --- |
| **public** **class** Blog **implements** Serializable {    **private** Integer id;  **private** String title;  /\*private int authorId;\*/  **private** Author author;  private String state; |

②配置mapper

a.BlogMapper.xml：

<resultMap>:

|  |
| --- |
| <resultMap type=*”Blog”* id=*”blogResultMap”*>  <id column=*”id”* property=*”id”* jdbcType=*”INTEGER”*/>  <association column=*”author\_id”* property=*”author”* javaType=*”Author”* select=*”com.zhangfd.mybatis.mapper.AuthorMapper.selectAuthorById”*></association>  </resultMap> |

<select>:

|  |
| --- |
| <select id=*"selectBlogById"* parameterType=*"int"* resultMap=*"blogResultMap"*>  select \* from blog where id=#{id}  </select> |

b.AuthorMapper.xml

<resultMap>:

|  |
| --- |
| <resultMap type=*"Author"* id=*"authorResultMap"*>  <id column=*"id"* property=*"id"* jdbcType=*"INTEGER"*/>  <result column=*"favourite\_section"* property=*"favouriteSection"* jdbcType=*"VARCHAR"*/>  </resultMap> |

<select>:

|  |
| --- |
| <select id=*"selectAuthorById"* parameterType=*"int"* resultMap=*"authorResultMap"*>  select \* from author where id=#{id}  </select> |

③测试

|  |
| --- |
| @Test  **public** **void** selectBlogById() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);  Blog blog = blogMapper.selectBlogById(3);  session.close();  System.***out***.println(blog);  } |

2）多对一（对象持有查询）

①配置同一对一

②mapper：

|  |
| --- |
| <select id=*"selectBlogList"* parameterType=*"int"* resultMap=*"blogResultMap"*>  select \* from blog  </select> |

③测试：

|  |
| --- |
| @Test  **public** **void** selectBlogList() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);  List<Blog> blogList = blogMapper.selectBlogList();  session.close();  System.***out***.println(blogList);  } |

3）一对多（查询对象并查询关联的list）

①修改pojo

在一的一方实体类中加上多的一方的list：

|  |
| --- |
| **private** List<Comment> commentList; |

②配置mapper：

a.PostMapper.xml（一方）

|  |
| --- |
| <resultMap type=*"Post"* id=*"postResultMap"*>  <id column=*"id"* property=*"id"* jdbcType=*"INTEGER"*/>  <collection property=*"commentList"* column=*"id"* javaType=*"ArrayList"* ofType=*"Comment"* select=*"com.zhangfd.mybatis.mapper.CommentMapper.selectCommentByPostId"*></collection>  </resultMap>    <select id=*"selectPostById"* parameterType=*"int"* resultMap=*"postResultMap"*>  select \* from post where id=#{id}  </select> |

b.CommentMapper.xml（多方）

|  |
| --- |
| <resultMap type=*"Comment"* id=*"commentResultMap"*>  <id column=*"id"* property=*"id"* jdbcType=*"INTEGER"*/>  </resultMap>  <select id=*"selectCommentByPostId"* parameterType=*"int"* resultMap=*"commentResultMap"*>  select \* from Comment where post\_id=#{post\_id}  </select> |

2.高级结果映射——嵌套结果（关联查询）

执行流程：

1.先执行关联查询，一次性将所有数据查询出来

2.再讲所有查询出来的列组织成嵌套的结果对象

1）mapper

|  |
| --- |
| <!-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 嵌套结果查询 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* -->    <resultMap type=*"Blog"* id=*"blogResultMapNested"*>  <id column=*"blog\_id"* property=*"id"* />  <result column=*"blog\_title"* property=*"title"* />  <result column=*"blog\_state"* property=*"state"* />  <result column=*"blog\_featured"* property=*"featured"* />  <result column=*"blog\_style"* property=*"style"* />  <association column=*"blog\_author\_id"* property=*"author"* javaType=*"Author"* >  <result column=*"author\_id"* property=*"id"* />  <result column=*"author\_username"* property=*"username"* />  <result column=*"author\_password"* property=*"password"* />  <result column=*"author\_email"* property=*"email"* />  <result column=*"author\_password"* property=*"password"* />  <result column=*"author\_email"* property=*"email"* />  <result column=*"author\_bio"* property=*"bio"* />  <result column=*"author\_favourite\_section"* property=*"favouriteSection"* />  <result column=*"author\_nickname"* property=*"nickname"* />  <result column=*"author\_realname"* property=*"realname"* />  </association>  </resultMap>  <select id=*"selectBlogListNested"* parameterType=*"int"* resultMap=*"blogResultMapNested"*>  SELECT  b.id AS blog\_id,  b.title AS blog\_title,  b.author\_id AS blog\_author\_id,  b.state AS blog\_state,  b.featured AS blog\_featured,  b.style AS blog\_style,  a.id AS author\_id,  a.username AS author\_username,  a.password AS author\_password,  a.email AS author\_email,  a.bio AS author\_bio,  a.favourite\_section AS author\_favourite\_section,  a.nickname AS author\_nickname,  a.realname AS author\_realname  FROM  blog b  LEFT JOIN  author a  ON b.author\_id=a.id  </select> |

2）接口：

|  |
| --- |
| List<Blog> selectBlogListNested(); |

3）测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogListNested() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);  List<Blog> blogList = blogMapper.selectBlogListNested();  session.close();  System.***out***.println(blogList);  } |

4）与嵌套查询对比：

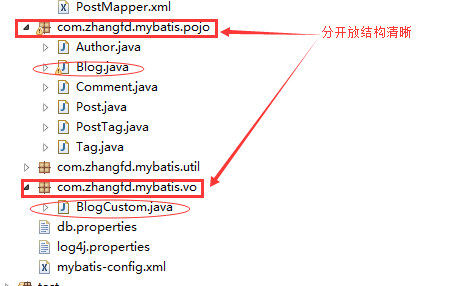
|  |  |  |
| --- | --- | --- |
|  | 嵌套结果 | 嵌套查询 |
| 查询次数 | 1 | N |
| 查询方式 | 关联查询 | 先查一条，然后在根据关联的id去查一次 |
| 查询效率 | 高 | 低，占用大量数据库资源 |

3.高级结果映射——扩展结果集

什么叫扩展结果集？

当查询的时候，有时候不需要把持有的对象整个查询出来，而只需要某个属性的时候，就需要进行扩展结果集。

1）建一个扩展VO(View Object)类继承要查询的类



VO:

|  |
| --- |
| **public** **class** BlogCustom **extends** Blog {    **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **private** String authorUsername;  **public** String getAuthorUsername() {  **return** authorUsername;  }  **public** **void** setAuthorUsername(String authorUsername) {  **this**.authorUsername = authorUsername;  }  } |

2）建扩展mapper：

|  |
| --- |
| <mapper namespace=*"com.zhangfd.mybatis.mapper.BlogMapperCustom"*>    <sql id=*"baseColum"*>  b.`id`,  b.`title`,  b.`author\_id` as authorId,  b.`state`,  b.`featured`,  b.`style` ,  a.username as authorUsername  </sql>    <select id=*"selectBlogById"* parameterType=*"int"* resultType=*"BlogCustom"*>  select  <include refid=*"baseColum"*></include>  from  blog b  left join  author a  on b.author\_id = a.id  where  b.id = #{id}  </select>    </mapper> |

3）建扩展DAO接口：

|  |
| --- |
| **public** **interface** BlogMapperCustom {    BlogCustom selectBlogById(Integer id);    } |

4）测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogCustom() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapperCustom blogMapperCustom = session.getMapper(BlogMapperCustom.**class**);  BlogCustom blogCustom = blogMapperCustom.selectBlogById(2);  session.close();  System.***out***.println(blogCustom);  } |

4.高级结果映射——构造方法映射

默认情况下mybatis会调用实体类的无参构造方法创建一个实体类，然后再给各个属性赋值，但是有的时候我们可能为实体类生成了有参的构造方法，并且也没有给该实体类生成无参的构造方法，这个时候如果不做特殊配置，resultMap在生成实体类的时候就会报错，因为它没有找到无参构造方法。这时，就需要配置构造方法映射。

Constructor：

|  |
| --- |
| **public** Blog(Integer id, String title) {  **super**();  **this**.id = id;  **this**.title = title;  System.***out***.println("构造方法调用。。。。。。。。");  } |

Mapper:

|  |
| --- |
| <resultMap type=*"Blog"* id=*"blogResultMapConstructor"*>  <constructor>  <idArg column=*"id"* javaType=*"int"*/>  <arg column=*"title"* javaType=*"string"*/>  </constructor>  </resultMap> |

5.鉴别器

作用：

类似培训时讲到设计模式——工厂模式时举例的打印机，配置鉴别条件后，可以自动鉴别出查询出来的是具体哪一个子类的对象。

测试用例

Vehicle(车辆表)表中有字段vehicle\_type(车辆类型), door\_count(车门数量), all\_wheel\_drive(是否全驱车),其中：

|  |  |  |  |
| --- | --- | --- | --- |
|  | vehicle\_type | door\_count | all\_wheel\_drive |
| Car | 1 | 有 | 无 |
| Suv | 2 | 无 | 有 |

Vehicle的子类Car**有**door\_count属性，**没有**all\_wheel\_drive属性；

Vehicle的子类Suv**没有**door\_count属性，**有**all\_wheel\_drive属性

Vehicle类中不用写vehicle\_type、door\_count和all\_wheel\_drive这三个属性。

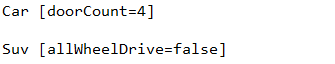
Mapper:

|  |
| --- |
| <resultMap type=*"Vehicle"* id=*"vehicleResultMap"*>  <id column=*"id"* property=*"id"* jdbcType=*"INTEGER"*/>  <discriminator javaType=*"int"* column=*"vehicle\_type"*>  <case value=*"1"* resultType=*"Car"*>  <result column=*"door\_count"* property=*"doorCount"*/>  </case>  <case value=*"2"* resultType=*"Suv"*>  <result column=*"all\_wheel\_drive"* property=*"allWheelDrive"*/>  </case>  </discriminator>  </resultMap>    <select id=*"selectVehicleById"* parameterType=*"int"* resultMap=*"vehicleResultMap"*>  select \* from vehicle where id=#{id}  </select> |

单元测试：

|  |
| --- |
| @Test  **public** **void** testSelectVehicleById() {    SqlSession session = MybatisUtil.*getSqlSession*();  VehicleMapper vehicleMapper = session.getMapper(VehicleMapper.**class**);  Vehicle vehicle = vehicleMapper.selectVehicleById(5);  Vehicle vehicle2 = vehicleMapper.selectVehicleById(7);  session.close();  System.***out***.println(vehicle);  System.***out***.println(vehicle2);  } |

输出结果（自动鉴别出查询出来的对象是实例化哪个子类的）：



五、延迟加载

延迟加载是什么意思？

延迟加载机制是为了避免一些无谓的性能开销而提出来的，所谓延迟加载就是当在真正需要数据的时候，才真正执行数据加载操作。通俗的说，就是一个查询涉及到多个sql查询的时候，只查询主要对象的sql，对象里持有的其它的类属性的查询sql不一起查询，在有调用的时候才去查询。这样可以极大程度的提高查询效率并减轻数据库压力。

1.没有配置的情况下

测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogByIdLazyLoading() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    System.***out***.println("查询blog");  Blog blog = blogMapper.selectBlogById(3);  session.close();    System.***out***.println("查询blog的Title属性");  System.***out***.println(blog.getTitle());  System.***out***.println("查询blog的author属性");  System.***out***.println(blog.getAuthor().getUsername());  System.***out***.println("查询结束");  } |

日志：

|  |
| --- |
| 查询blog  Opening JDBC Connection  Created connection 161960012.  Setting autocommit to false on JDBC Connection  ==> Preparing: select \* from blog where id=?  ====> Preparing: select \* from author where id=?  Returned connection 161960012 to pool.  查询blog的Title属性  是人性的扭曲？还是道德的沦丧？  查询blog的author属性  jiny  查询结束 |

2.配置了全局延迟加载，不调用属性情况下

配置：

在mybatis-config.xml的<settings></settings>节点中加入：

|  |
| --- |
| <setting name=*"lazyLoadingEnabled"* value=*"true"*/> |

**测试：**

|  |
| --- |
| @Test  **public** **void** testSelectBlogByIdLazyLoading() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    System.***out***.println("查询blog");  Blog blog = blogMapper.selectBlogById(3);  session.close();    // System.out.println("查询blog的Title属性");  // System.out.println(blog.getTitle());  // System.out.println("查询blog的author属性");  // System.out.println(blog.getAuthor().getUsername());  System.***out***.println("查询结束");  } |

**日志**：

|  |
| --- |
| 查询blog  Opening JDBC Connection  Created connection 161960012.  Setting autocommit to false on JDBC Connection  ==> Preparing: select \* from blog where id=?  Returned connection 161960012 to pool.  查询结束 |

3. 配置了全局延迟加载，调用Title属性情况下（积极）

**默认积极的延迟加载，只要有调用任意属性就会进行相关查询，不论调用的属性是否是相关查询的结果。**

测试：

|  |
| --- |
| @Test  **public** **void** testSelectBlogByIdLazyLoading() {  SqlSession session = MybatisUtil.*getSqlSession*();  BlogMapper blogMapper = session.getMapper(BlogMapper.**class**);    System.***out***.println("查询blog");  Blog blog = blogMapper.selectBlogById(3);  session.close();    System.***out***.println("查询blog的Title属性");  System.***out***.println(blog.getTitle());  // System.out.println("查询blog的author属性");  // System.out.println(blog.getAuthor().getUsername());  System.***out***.println("查询结束");  } |

日志：

|  |
| --- |
| 查询blog  Opening JDBC Connection  Created connection 161960012.  Setting autocommit to false on JDBC Connection  ==> Preparing: select \* from blog where id=?  Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@9a7504c]  Returned connection 161960012 to pool.  查询blog的Title属性  Opening JDBC Connection  Checked out connection 161960012 from pool.  ==> Preparing: select \* from author where id=?  Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@9a7504c]  Returned connection 161960012 to pool.  是人性的扭曲？还是道德的沦丧？  查询结束 |

4. 配置了全局延迟加载，调用Title属性情况下（非积极）

配置：

在mybatis-config.xml的<settings></settings>节点中加入：

|  |
| --- |
| <setting name=*"lazyLoadingEnabled"* value=*"true"*/>  <setting name=*"aggressiveLazyLoading"* value=*"false"*/> |

测试同第三点。

日志：

|  |
| --- |
| 查询blog  Opening JDBC Connection  Created connection 161960012.  ==> Preparing: select \* from blog where id=?  Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@9a7504c]  Returned connection 161960012 to pool.  查询blog的Title属性  是人性的扭曲？还是道德的沦丧？  查询结束 |