**Mybatis概述**

**1.什么是Mybatis？**

MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射,是一个半自动的ORM(Object Relation Mapping)框架。

**2.Mybatis的优点？**

1.MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。

2.MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO为数据库中的记录。

3.基于SQL语句编程，相当灵活，SQL与程序代码分离，方便管理。

4.Mybatis底层使用JDBC操作数据库，只要JDBC支持的数据库MyBatis都支持且能与spring很好的集成。

5.对性能的要求很高，或者需求变化较多的项目，Mybatis是不错的选择。

**3.Mybatis的缺点？**

1.由于Mybatis是基于SQL的，而SQL依赖于数据库，因此导致数据库的移植性比较差。

2.SQL的开发工作量比较大，对开发人员编写SQL的能力有要求。

# 如何使用Mybatis操作数据库

## 1.添加Mybatis依赖

<dependency>

  <groupId>org.mybatis</groupId>

  <artifactId>mybatis</artifactId>

  <version>3.5.10</version>

</dependency>

<!--Oracle驱动包-->

<dependency>

  <groupId>com.oracle</groupId>

  <artifactId>ojdbc14</artifactId>

  <version>8.0.0</version>

  <scope>compile</scope>

</dependency>

<!--查看SQL语句等日志信息-->

<dependency>

 <groupId>log4j</groupId>

 <artifactId>log4j</artifactId>

 <version>1.2.17</version>

</dependency>

## 2.配置Mybatis

2.1 配置log4j.properties文件

log4j.rootLogger=debug,stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

2.2 编写jdbc.properties

oracle.driver = oracle.jdbc.driver.OracleDriver

oracle.url = jdbc:oracle:thin:@localhost:test

oracle.username = scott

oracle.password = target

3.编写mybatis-conf.xml文件

<?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>

 <!--

 Mybatis可以使用properties来引入外部的properties配置文件的内容

  resource:引入类路径下的资源

  url:引入网络路径或磁盘路径下的资源

  -->

  <properties resource="conf/jdbc.properties"></properties>

  <!--

   logImpl:指定 MyBatis 所用日志的具体实现，未指定时将自动查找,默认mybatis未指定;

   mapUnderscoreToCamelCase:是否开启驼峰命名自动映射，即从经典数据库列名 A\_COLUMN 映射到经典 Java 属性名 aColumn。默认Mybatis没有开启;

   -->

   <settings>

    <setting name="logImpl" value="LOG4J"/>

    <setting name="mapUnderscoreToCamelCase" value="true"/>

  </settings>

  <!-- 配置数据源 -->

  <environments default="development">

    <environment id="development">

      <transactionManager type="JDBC"/>

      <dataSource type="POOLED">

        <property name="driver" value="${oracle.driver}"/>

        <property name="url" value="${oracle.url}"/>

        <property name="username" value="${oracle.username}"/>

        <property name="password" value="${oracle.password}"/>

      </dataSource>

    </environment>

  </environments>

   <!-- 告诉Mybatis SQL映射文件的位置-->

   <mappers>

    <mapper resource="com/stu/mybatis/mapper/EmpMapper.xml"/>

  </mappers>

</configuration>

## 3.映射 SQL 语句

SQL语句都可以由 XML 或 Annotations 定义。

### 3.1使用XML方式定义SQL

**1.编写SQL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.stu.mybatis.mapper.EmpMapper">

 <!-- 通用查询映射结果 -->

 <resultMap id="BaseResultMap" type="com.stu.mybatis.Emp">

  <id column="EMPNO" property="empno"  />

     <result column="ENAME" property="ename"  />

     <result column="JOB" property="job"  />

     <result column="E\_HOBBY" property="eHobby"  />

 </resultMap>

      <!--查询单条-->

     <select id="selectEmp" resultType="com.stu.mybatis.Emp">

      select \* from emp  where empno = #{empno}

    </select>

 <!--查询多条-->

    <select id="selectEmpList" resultType="com.stu.mybatis.Emp">

      select \* from emp

      <where>

       <if test="ename!=null">

         ename like concat(concat('%',#{ename}),'%')

       </if>

     </where>

    </select>

    <!--插入（Oracle写法）

      keyProperty:查出的主键值封装到JavaBean的那个属性。

         order="BEFORE":当前SQL在插入SQL之前运行。

         resultType:查出数据返回类型。

 -->

    <insert id="insertEmp" parameterType="com.stu.mybatis.Emp">

     <selectKey keyProperty="empno" order="BEFORE" resultType="Long">

      select seq\_emp.nextval from dual

     </selectKey>

     <!--插入时的主键从序列中拿到-->

    insert into emp(empno,ename,job)values(#{empno},#{ename},#{job})

    </insert>

 <!--修改-->

    <update id="updateEmp" parameterType="com.stu.mybatis.Emp">

    update emp set ename=#{ename},job=#{job} where empno= #{empno}

    </update>

   <!--删除-->

    <delete id="deleteEmp" parameterType="Long">

     delete emp where empno=#{empno}

    </delete>

</mapper>

**2.编写Mapper接口,方法名和XML配置文件的Id属性的值保持一致**

public interface EmpMapper {

  Emp selectEmp(Long id);

  Long insertEmp(Emp emp);

  Long updateEmp(Emp emp);

  Long deleteEmp(Long empno);

  List<Emp> selectEmpList(Emp emp);

}

### 3.2使用注解方式定义SQL

public interface EmpMapper {

*//1.查询*

@Select("select \* from emp  where empno = #{empno}")

Emp selectEmp(Long id);

*//2.插入（Oracle）*

@SelectKey(

  keyProperty = "empno",

  before = true,

  resultType = Long.class,

  statement = "select seq\_emp.nextval from dual")

@Insert("insert into emp(empno,ename,job)values(#{empno},#{ename},#{job})")

Long insertEmp(Emp emp);

*//3.修改*

@Update("update emp set ename=#{ename},job=#{job} where empno= #{empno}")

  Long updateEmp(Emp emp);

*//4.删除*

 @Delete("delete emp where empno=#{empno}")

 Long deleteEmp(Long empno);

}

## 4.测试

### 4.1.构建SqlSessionFactory

每个 MyBatis 应用程序都以 SqlSessionFactory 实例为中心。通过XML文件(Mybatis配置文件)来构建 SqlSessionFactory；

### 4.2.从SqlSessionFactory中获取SqlSession

获取一个SqlSession 的实例。SqlSession 包含对数据库执行 SQL 命令所需的所有方法。Mybatis推荐通过SqlSession获取对应Mapper接口的实现对象（代理对象）实现对数据库的操作。

### 4.3.测试实例

public class MybatisTest {

    @Test

    public void testSelectEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

      SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

   SqlSession openSession = sqlSessionFactory.openSession();

   System.out.println(openSession);

*//3.获取接口的实现对象*

   EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

   try {

*//1.查询单条*

    Emp selectOne = mapper.selectEmp(7369L);

    System.out.println(selectOne);

   }catch (Exception e) {

*// TODO: handle exception*

    e.printStackTrace();

   }finally {

    openSession.close();

   }

  }

    @Test

    public void testSelectEmpList() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

*//查询列表*

      List<Emp> selectEmpList = mapper.selectEmpList();

      System.out.println(selectEmpList);

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

    @Test

    public void testInsertEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

*//插入*

      Emp emp = new Emp();

      emp.seteHobby("摄影");

      emp.setEname("oracle");

      emp.setJob("doctor");

      Long insertEmp = mapper.insertEmp(emp);

      openSession.commit();

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

    @Test

    public void testUpdateEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

      Emp e = new Emp();

      e.seteHobby("游泳,打篮球");

      e.setEname("mike1");

      e.setJob("doctor1");

      e.setEmpno(6L);

      Long updateEmp = mapper.updateEmp(e);

      openSession.commit();

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

    @Test

    public void testDeleteEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

     SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

     SqlSession openSession = sqlSessionFactory.openSession();

     System.out.println(openSession);

*//3.获取接口的实现对象*

     EmpMapper mapper = openSession.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

     try {

   mapper.deleteEmp(6L);

      openSession.commit();

     }catch (Exception e) {

*// TODO: handle exception*

      e.printStackTrace();

     }finally {

      openSession.close();

     }

    }

private static SqlSessionFactory getSqlSessionFactory() throws IOException {

 String resource = "conf/mybatis-conf.xml";

 Reader reader = Resources.getResourceAsReader(resource);

 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);

 return sqlSessionFactory;

}

}

## 总结

1.SqlSession代表和数据库一次会话，用完必须关闭;

2.SqlSession和Connection一样是非线程安全的，每次使用都应该去获取对象。

3.Mapper接口没有实现类，但是Mybatis会为Mapper接口生成一个代理对象。（将接口和xml进行绑定）

4.Mybatis全局配置文件包含了数据库连接池信息、事务管理器等系统运行环境信息。SQL映射文件保存了每一个SQL语句的映射信息。

**Mybatis全局配置文件**

MyBatis 的配置文件包含影响 MyBatis行为的设置和属性信息。可以参考官方文档: <https://mybatis.org/mybatis-3/zh/configuration.html>

**1.settings（设置）**

settings是Mybatis重要的调整设置，对Mybatis运行时的行为进行配置;  
常用配置：  
mapUnderscoreToCamelCase：开启驼峰命名自动映射，默认false；解决据库列名 A\_COLUMN 映射到Java 属性名 aColumn；

**2.typeAliases（类型别名）**

typeAliases为Java类型设置一个别名。 别名不区分大小写，它仅用于 XML 配置，意在降低冗余的全限定类名书写。  
1).为某一个Java类型设置别名。

<!--

type：指定要起别名的类型的全类名,默认别名是类名小写(author)；

alias：指定别名

-->

<typeAliases>

  <typeAlias alias="Author" type="domain.blog.Author"/>

  <typeAlias alias="Blog" type="domain.blog.Blog"/>

</typeAliases>

2）批量取别名

使用ypeAlias取别名只能为一个类型取别名，如果有多个Bean的时候工作量比较大。package可以为某个包下的所有类批量去别名，name属性指定包名（为当前包及下面所有的后代包的每一个类都起一个默认的别名，默认别名为类名小写）。

<typeAliases>

<package name="domain.bean"/>

</typeAliases>

3）批量取别名后如果指定的包下（domian.bean）含有多个相同类型的JavaBean,会产生别名冲突，Mybatis会报错，可以在JavaBean上使用@Alias注解在具体的JavaBean指定别名;

4)Mybatis为常见的 Java类型内置了别名。它们都是不区分大小写的。我们自己在取别名的时候一定不要和Mybatis内置的别名冲突，否则Mybatis可能会出现问题；实际开发中建议使用全类型名。

**3.typeHandlers（类型处理器）**

typeHandlers类型处理器，MyBatis 在设置预处理语句（PreparedStatement）中的参数或从结果集中取出一个值时， 都会用类型处理器将获取到的值以合适的方式转换成 Java 类型。是Java类型和数据库类型进行转换的桥梁。

**4.plugins（插件）**

MyBatis 允许你在映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截Executor(),ParameterHandler(),ResultSetHandler(),StatementHandler()方法调用。

**5.environments（环境配置）**

MyBatis 可以配置成适应多种环境，这种机制有助于将 SQL 映射应用于多种数据库之中。 实际开发中交数据源和事务最终交给Spring来做。

**6.databaseIdProvider（数据库厂商标识）**

MyBatis 可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的 databaseId 属性。

**7.mappers（映射器）**

告诉 MyBatis 到哪里去找映射文件。

使用相对于类路径的资源引用

<!-- 使用相对于类路径的资源引用 -->

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<mapper resource="org/mybatis/builder/BlogMapper.xml"/>

<mapper resource="org/mybatis/builder/PostMapper.xml"/>

</mappers>

使用完全限定资源定位符（URL

<!-- 使用完全限定资源定位符（URL） -->

<mappers>

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

<mapper url="file:///var/mappers/BlogMapper.xml"/>

<mapper url="file:///var/mappers/PostMapper.xml"/>

</mappers>

使用映射器接口实现类的完全限定类名

<!-- 使用映射器接口实现类的完全限定类名

1.有SQL映射文件，映射文件名必须和接口同名，并且放在与接口同一目录下；

2.没有SQL映射文件，所有的sql都是用注解的方式写在接口上

3.复杂SQL建议使用SQL映射文件方式编写，简单SQL可以使用注解实现快速开发；

-->

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

<mapper class="org.mybatis.builder.BlogMapper"/>

<mapper class="org.mybatis.builder.PostMapper"/>

</mappers>

批量注册

<!-- 将包内的映射器接口实现全部注册为映射器 -->

<mappers>

<package name="org.mybatis.builder"/>

</mappers>

总结： 1.Mybatis的全局配置文件中标签编写的时候是有顺序的。

**Mybatis参数处理**

**1.单个参数**

**1.单个参数且没有使用@param注解命名参数，Mybatis不会做特殊处理，使用#{参数名}取出参数值。**

public Employee getEmpById(Integer id);

**2.单个参数且参数类型为Collection（List、Set）类型或数组（Array）类型，Mybatis会将传入的Collection或者数组封装到Map中。**

I.)如果参数类型是Collection（List、Set）,Mybatis会将传入的Collection封装到Map中，key为collection，如果参数类型是List，Mybatis还会额外封装一个key为list的数据到Map中。

*//取出第一个id的值：#{collection[0]}*

public List<Employee> selectEmp(Set<Long> id);

*//取出第一个id的值：#{list[0]|collection[0]}*

public List<Employee> selectEmp(List<Long> id);

*// 取出第一个id的值：#{ids[0]}*

public List<Employee> selectEmp(@Param("ids") Set<Long> id);

II.)如果参数的数据类型为数组,Mybatis会将传入的数组封装到Map中，Key为array。

public List<Employee> selectEmp(String [] id); //取出第一个id的值:#{array[0]}

public List<Employee> selectEmp(@Param("ids")String [] id); //取出第一个id的值:#{ids[0]}

**2.多个参数**

**1.多个参数会被封装为一个Map，使用#{key}从map中获取指定的key的值，key为param1...paramN、arg0...argN或者参数的索引。**

**I.)**Mybatis3.4.1以后版本且全局配置useActualParamName=true时，可以使用#{key}从map中获取指定的key的值,key为param1...paramN或arg0...argN。

public List<Employee> selectEmpList(String name,Integer age);

*//name:#{param1|arg0}*

*//age:#{param2|arg1}*

**II.)**Mybatis3.4.1以前的版本和Mybatis3.4.1以后版本且全局配置useActualParamName=false时，可以使用#{key}从map中获取指定的key的值,key为param1...paramN或参数索引。

public List<Employee> selectEmpList(String name,Integer age);

*//name:#{param1|0}*

*//age:#{param2|1}*

**2.多个参数时推荐使用命名参数，使用@param注解明确指定封装参数时Map的key，key为@param注解指定的值，value为参数值，使用#{指定的key}取出对应的参数值。使用了@param注解命名参数后不能使用#{索引、argN}取出对应的参数值。可以使用@param注解指定的key或paramN取出对应的参数值。**

public List<Employee> selectEmpList(@Param("name") String name,@Param("age")Integer age);

*//name:#{param1|name}*

*//age:#{param2|age}*

**3.如果多个参数正好是业务逻辑的数据模型，直接传入Pojo,#{属性名}取出传入Pojo的属性值。**

public List<Employee> selectEmpList(Employee employee);

**4.如果多个参数不是业务逻辑的数据模型，没有对应的Pojo且不经常使用，为了方便，可以传入Map,#{key}取出Map中对应的值。**

public List<Employee> selectEmpList(Map<String,Object> map);

**5.如果多个参数不是业务逻辑的数据模型，但是经常使用，推荐编写一个TO(Transfer Object)数据传输对象。**

public List<Employee> selectEmpList(PageInfo page);

**4.Mybatis源码处理参数**

4.1处理参数源码

public Object getNamedParams(Object[] args) {

    final int paramCount = names.size();

*//1.如果参数为null或无参数直接返回*

    if (args == null || paramCount == 0) {

      return null;

    } else if (!hasParamAnnotation && paramCount == 1) {

*//2.如果只有一个参数,并且没有param注解,args[0]取出参数值;*

      Object value = args[names.firstKey()];

*//根据参数类型封装参数为Map返回*

      return wrapToMapIfCollection(value, useActualParamName ? names.get(0) : null);

    } else {

*//3.多个参数或一个参数且有param注解时，封装为Map;*

      final Map<String, Object> param = new ParamMap<>();

      int i = 0;

      for (Map.Entry<Integer, String> entry : names.entrySet()) {

*//3.1.添加参数名称(arg0,arg1,...),将参数封装为key为argN,value为对应的参数值;*

        param.put(entry.getValue(), args[entry.getKey()]);

*// add generic param names (param1, param2, ...)*

*//3.2添加参数名称(param1,param2,...),将参数封装为key为paramN,value为对应的参数值;*

        final String genericParamName = GENERIC\_NAME\_PREFIX + (i + 1);

*// ensure not to overwrite parameter named with @Param*

        if (!names.containsValue(genericParamName)) {

          param.put(genericParamName, args[entry.getKey()]);

        }

        i++;

      }

      return param;

    }

  }

4.2一个参数的时，Mybatis处理源码

public static Object wrapToMapIfCollection(Object object, String actualParamName) {

    if (object instanceof Collection) {

*//1.如果是参数是Collection类型,将参数封装为Map,key:collection,value为对应的参数的值;*

      ParamMap<Object> map = new ParamMap<>();

      map.put("collection", object);

*//2.如果参数是List类型key：list,value为对应的参数的值;*

      if (object instanceof List) {

        map.put("list", object);

      }

      Optional.ofNullable(actualParamName).ifPresent(name -> map.put(name, object));

      return map;

    } else if (object != null && object.getClass().isArray()) {

*//3.如果是数组，封装为一个Map，key:array,value为对应的参数的值;*

      ParamMap<Object> map = new ParamMap<>();

      map.put("array", object);

      Optional.ofNullable(actualParamName).ifPresent(name -> map.put(name, object));

      return map;

    }

*//4.如果只有一个参数,不是Collection和Array类型,直接返回参数值;*

    return object;

  }

4.3在ParamNameResolver构造的时候为names的设置值

public ParamNameResolver(Configuration config, Method method) {

    this.useActualParamName = config.isUseActualParamName();

*//1.获取方法列表中参数的数据类型*

    final Class<?>[] paramTypes = method.getParameterTypes();

*//2.获取方法列表中参数的注解类型*

    final Annotation[][] paramAnnotations = method.getParameterAnnotations();

    final SortedMap<Integer, String> map = new TreeMap<>();

    int paramCount = paramAnnotations.length;

*// get names from @Param annotations*

*//3.遍历所有不是RowBounds或者ResultHandler子类的参数*

    for (int paramIndex = 0; paramIndex < paramCount; paramIndex++) {

      if (isSpecialParameter(paramTypes[paramIndex])) {

*// skip special parameters*

        continue;

      }

      String name = null;

      for (Annotation annotation : paramAnnotations[paramIndex]) {

*//判断参数中是否存在@Param注解，存在的话hasParamAnnotation设置true,*

*//获取@param注解设置的值,放入TreeMap中key为参数的下标,value为@param注解设置的值。*

*// selectEmp(@param("id") Long id); {0=id}*

        if (annotation instanceof Param) {

          hasParamAnnotation = true;

          name = ((Param) annotation).value();

          break;

        }

      }

      if (name == null) {

*// @Param was not specified.*

*//如果参数没有@Param注解，但是在全局配置中设置了useActualParamName=true*

*//(3.4.1版本后默认为true),允许使用方法签名中的名称作为语句参数名称.*

*//selectEmp(Long id,String ename);{0=arg0}*

        if (useActualParamName) {

          name = getActualParamName(method, paramIndex);

        }

*//如果3.4.1版本后,且在全局配置中设置了useActualParamName=false*

*//或者3.4.1之前版本,{0=0}*

        if (name == null) {

*// use the parameter index as the name ("0", "1", ...)*

*// gcode issue #71*

          name = String.valueOf(map.size());

        }

      }

      map.put(paramIndex, name);

    }

    names = Collections.unmodifiableSortedMap(map);

  }

**5.#{}和${}的区别**

#{}和${}都可以获取map的值或者pojo对象属性值

1.#{}是以预编译的形式(PreparedStatement)，将参数设置到SQL语句中，可以防止SQL注入。

2.${}取出的值直接拼接在SQL语句中，会有安全问题(SQL注入风险)。

3.大多情况下取参数的值都应该使用#{},在原生Jdbc不支持占位符的地方使用${}进行取值。如在分表(如按照年分表)、排序等情况下使用${}取值。

select \* from ${year}\_salary;

select \* from t\_employee order by ${o\_col} ${order};

4.#{}提供了更丰富的用法，可以规定参数的规则。javaType、jdbcType、typeHandler、numericScale、mode（用于存储过程）、resultMap、jdbcTypeName。

5.jdbcType通常在特定的条件下需要被设置。由于MYbatis全局配置文件中，jdbcTypeForNull默认取值为OTHER,Oracle不支持。使用Oracle保存数据时，当null被当做值来传递，Oracle会报错（jdbcType OTHER：无效的类型）。因为Mybatis默认对所有的null都是映射的是原生Jdbc的OTHER类型，Oracle不能正确处理。

解决方案：

1.为可能为空的参数设置jdbcType设为NULL，如#{ename,jdbcType = NULL}

<insert id="insertEmp" parameterType="com.stu.mybatis.Emp">

     <selectKey keyProperty="empno" order="BEFORE" resultType="Long">

      select seq\_emp.nextval from dual

     </selectKey>

     <!--插入时的主键从序列中拿到-->

    insert into emp(empno,ename,job)values(#{empno},#{ename,jdbcType = NULL},#{job})

    </insert>

2.修改Mybatis的全局运行时配置

<settings>

<setting name="jdbcTypeForNull" value="NULL"/>

</settings>

# MYbatis实现单条插入

## 1.不支持自增的数据库，如Oracle

1.keyProperty:查出的主键值封装到JavaBean的那个属性。

2.order="BEFORE":当前SQL在插入SQL之前运行。

3.resultType:查出数据返回类型。

### 1.使用注解方式

@SelectKey(

        keyProperty = "empno",

        before = true,

        resultType = Long.class,

        statement = "select seq\_emp.nextval from dual"

 )

  @Insert("insert into emp(empno,ename,job)values(#{empno},#{ename},#{job})")

  Long insertEmp(Emp emp);

### 2.使用XML方式

<insert id="insertEmp" parameterType="com.stu.mybatis.Emp">

     <selectKey keyProperty="empno" order="BEFORE" resultType="Long">

      select seq\_emp.nextval from dual

     </selectKey>

     <!--插入时的主键从序列中拿到-->

    insert into emp(empno,ename,job)values(#{empno},#{ename},#{job})

    </insert>

# Mybatis Select元素

## 1.select元素

1.Id:唯一标识：用来引用该SQL语句，需要和接口的方法名一致。

2.parameterType：参数类型，可以不传，Mybatis会根据TypeHandler自动推断。

3.resultType：返回值类型，别名或全类名，如果返回的是集合，定义集合中元素的类型，不能和resultMap同时使用。

<select id="getEmpById" parameterType="Long" resultType="com.itour.Employee">

 select \* from t\_employee where id =#{id}

</select>

## 2.返回List

1.返回集合，定义集合中的元素的类型。

I.Mapper接口

public List<Employee> selectEmpList(Employee employee);

II.XML文件

<select id="selectEmpList" parameterType="com.itour.Employee" resultType="com.itour.Employee">

 select \* from t\_employee where ename like #{ename}

</select>

## 3.返回Map

### 1.返回一条记录的Map

返回一条记录的Map，key是列名，值是列对应的值

I.Mapper接口

public Map<String,Object>(Long id);

II.XML文件

<select id="getEmpById" parameterType="Long" resultType="map">

 select \* from t\_employee where id =#{id}

</select>

### 2.返回多条记录的Map

返回多条记录的Map,key:为这条记录的主键，value：为封装后的javaBean；

I.Mapper接口  
使用@MapKey注解告诉Mybatisf封装这个Map的时候使用那个属性作为Map的key

@MapKey（"id"）

public Map<Integer,Employee> selectEmpList(Employee employee);

II.XML文件

<select id="selectEmpList" parameterType="com.itour.Employee" resultType="com.itour.Employee">

 select \* from t\_employee where ename like #{ename}

</select>

## 4.自动映射

1.Myatis默认开启了自动映射的功能，唯一要求是列名和JavaBean属性名一致；

<settings>

  <setting name="autoMappingBehavior" value="PARTIAL"/>

</settings>

autoMappingBehavior默认为PARTIAL 只会自动映射没有定义嵌套结果映射的字段。 NONE 表示关闭自动映射；

2.如果数据库字段命名规范，POJO属性复合驼峰命名法则（a\_column-->aColumn）,可以开启自动驼峰命名规则映射功能。

<settings>

  <setting name="mapUnderscoreToCamelCase" value="true"/>

</settings>

mapUnderscoreToCamelCase默认为fasle，需要手动开启。

## 4.ResultType属性

1.返回值类型，别名或全类名，如果返回的是集合，定义集合中元素的类型，不能和resultMap同时使用。

2.ResultType是和自动封装有关的属性，在查出数据以后，Mybatis会根据指定的返回类型，将数据封装为对应的Java类型。但是如果在查询出的数据中的列名和JavaBean的属性名不一致，Mybatis不会自动封装。可以使用以下方式解决：

1.)可以使用别名方式解决。

2.)如果列名和JavaBean的属性名符合驼峰命名法则，可以在Mybatis中开启驼峰命名法解决。

3.)也可以使用ResultMap自定义结果集。

**Mybatis使用ResultMap自定义映射规则**

自定义ResultMap，实现高级结果集映射，可以完成复杂的映射。

**1.关联查询，级联属性封装结果**

1.查询员工(Employee)的同时查询员工对应的部门

I.Mapper

List<Employee> withDeptResultMap();

2.XML

<!--联合查询，级联属性封装结果-->

 <resultMap id="withDeptResultMap" type="com.itour.model.Employee">

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

        <result column="ename" property="ename" />

        <result column="job" property="job" />

        <result column="email" property="email" />

  <result column="deptno" property="deptno" />

        <result column="did" property="dept.id" />

        <result column="dname" property="dept.dname" />

</resultMap>

<select id="selectEmpListWithDept" resultMap ="withDeptResultMap">

selectv e.id, e.ename,e.job,e.email,d.id did,d.dname from emp,dept where e.deptno = d.id

</select>

**2.关联查询，使用association定义关联对象封装规则**

1.查询员工(Employee)的同时查询员工对应的部门

I.Mapper

List<Employee> withDeptResultMap();

2.XML

<!--使用association指定联合的JavaBean对象

  property="dept":指定那个属性是联合对象

  javaType:指定该属性对象的java类型（不能省略）

-->

 <resultMap id="withDeptResultMap" type="com.itour.model.Employee">

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

        <result column="ename" property="ename" />

        <result column="job" property="job" />

        <result column="email" property="email" />

  <result column="deptno" property="deptno" />

  <association property="dept" javaType="com.itour.model.Dept">

   <result column="did" property="id" />

  <id column="did" property="id" />

        <result column="dname" property="dname" />

  </association>

</resultMap>

<select id="selectEmpListWithDept" resultMap ="withDeptResultMap">

selectv e.id, e.ename,e.job,e.email,d.id did,d.dname from emp,dept where e.deptno = d.id

</select>

**3.使用association进行分部查询**

I.Mapper接口

Employee withDeptResultMap();

2.XML文件

<!--使用association进行分部查询

 1.根据员工Id查询员工信息

 2.根据员工信息中的deptno查询出对应的部门信息

 3.将查询出的部门信息设置到员工中；

-->

 <resultMap id="withDeptResultMap" type="com.itour.model.Employee">

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

        <result column="ename" property="ename" />

        <result column="job" property="job" />

        <result column="email" property="email" />

  <result column="deptno" property="deptno" />

  <!--使用association定义关联对象的封装规则

   property="dept":指定那个属性是联合对象

   select：表明当前属性是调用select指定的方法查询出的结果

   column：指定将那一列的值传给这个方法,多列值传递，将多列的值封装为Map传递，column="{key1=column1,key2=column2,keyN=columnN}"

   总结：使用select指定的方法（传入column指定的这列的参数的值）查出对象，并封装给property指定的属性。

  -->

  <association property="dept" select="com.itour.persist.DeptMapper.getDeptById" column="deptno">

   <result column="did" property="id" />

  <id column="did" property="id" />

        <result column="dname" property="dname" />

  </association>

</resultMap>

<select id="selectEmpById" resultMap ="withDeptResultMap">

select \* from emp where id =#{id}

</select>

**4.延迟查询（懒加载）**

使用association进行分部查询，每次查询Employee对象的时候都会查询出Dept,association支持延迟查询，使用延迟查询可以实现部门信息在使用的时候再去查询部门相关信息。再分步查询的基础上添加两个配置即可。在全局配置文件中配置lazyLoadingEnabled和aggressiveLazyLoading

<settings>

  <!--默认false-->

  <setting name="lazyLoadingEnabled" value="true"/>

  <!--在 3.4.1 及之前的版本中默认为 true）-->

  <setting name="aggressiveLazyLoading" value="false"/>

</settings>

**5.关联查询-collection定义联合集合封装规则**

1.查询部门时，将该部门下的所以员工查询出来(一对多)

<resultMap id="withEmpResultMap" type="com.itour.model.Dept">

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

        <result column="dname" property="dname" />

  <!--collection定义关联集合类型的属性的封装规则

  property：指定关联结合类型的属性

  ofType：指定集合元素的类型

  -->

       <collection property="emp" ofType="com.itour.model.Employee">

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

  <result column="ename" property="ename" />

  <result column="email" property="email" />

  <result column="deptno" property="deptno" />

    </collection>

</resultMap>

<select id="selectDeptById" resultMap ="withEmpResultMap">

select d.id,d.dname,e.id empno,e.ename,e.email,deptno from dept d left join emp e on d.id =e.deptno where d.id =#{id}

</select>

**6.关联查询-collection分步查询**

<resultMap id="withEmpResultMap" type="com.itour.model.Dept">

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

        <result column="dname" property="dname" />

  <!--collection定义关联集合类型的属性的封装规则

  property：指定关联结合类型的属性

  ofType：指定集合元素的类型

  -->

       <collection property="emp" ofType="com.itour.model.Employee">

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

  <result column="ename" property="ename" />

  <result column="email" property="email" />

  <result column="deptno" property="deptno" />

    </collection>

</resultMap>

<select id="selectDeptById" resultMap ="withEmpResultMap">

select d.id,d.dname,e.id empno,e.ename,e.email,deptno from dept d left join emp e on d.id =e.deptno where d.id =#{id}

</select>

**7.discriminator：鉴别器**

鉴别器：Mybatis可以使用discriminator判断某一列的值，然后根据某一列的值改变封装行为。 如封装Employee:如果查询出来的job为SALSMAN，查出对应的部门信息。

<!--使用association进行分部查询

 1.根据员工Id查询员工信息

 2.根据员工信息中的deptno查询出对应的部门信息

 3.将查询出的部门信息设置到员工中；

-->

 <resultMap id="withDeptResultMap" type="com.itour.model.Employee">

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

        <result column="ename" property="ename" />

        <result column="job" property="job" />

        <result column="email" property="email" />

  <result column="deptno" property="deptno" />

  <!--

  column:只当判别的列名

  javaType：列值对应的java类型

  -->

  <discriminator javaType="String" column="job">

   <!--

     job为SALSMAN

     resultType：指定封装结果类型

   -->

         <case value="SALSMAN" resultType="com.itour.model.Dept">

   <association property="dept" select="com.itour.persist.DeptMapper.getDeptById" column="deptno">

   <result column="did" property="id" />

  <id column="did" property="id" />

        <result column="dname" property="dname" />

  </association>

   </case>

        </discriminator>

</resultMap>

<select id="selectEmpById" resultMap ="withDeptResultMap">

select \* from emp where id =#{id}

</select>

# Mybatis动态SQL

动态SQL是Mybatis强大的特性之一，极大的简化了拼装SQL的操作。Mybatis采用基于OGNL表达式来简化操作。OGNL表达式是Apache下的一个开源项目，具体使用可以参考 <https://commons.apache.org/proper/commons-ognl/language-guide.html>

## 1.if-判断

### 1.基本语法

test:判断表达式（OGNL），特殊符号需要写转义字符；

<if test="job != null">

 and job = #{job}

</if>

### 2.使用IF拼装SQL问题

查询时如果某一些条件没带可能SQL拼装会有问题。当job为null，ename不为null的时候，SQL语句：and ename like concat(concat('%',#{ename}),'%')

<select id="selectEmpList" resultType="com.itour.model.Employee">

select \* from tmp where

<if test="job != null">

   job = #{job}

</if>

<if test="ename!=null">

   and ename like concat(concat('%',#{ename}),'%')

</if>

</select>

### 3.解决方案

1.）给where 后面加上1=1 以后的条件用and xxx

<select id="selectEmpList" resultType="com.itour.model.Employee">

select \* from tmp where 1=1

<if test="job != null">

  and job = #{job}

</if>

<if test="ename!=null">

   and ename like concat(concat('%',#{ename}),'%')

</if>

</select>

2.）Mybatis使用where标签来将索引的查询条件包括在内,Mybatis会将where标签中拼接的SQL，多出的and或or去除,where 只会去掉第一个多出来的and或or。

<select id="selectEmpList" resultType="com.itour.model.Employee">

select \* from tmp

<where>

<if test="job != null">

   and job = #{job}

</if>

<if test="ename!=null">

    and ename like concat(concat('%',#{ename}),'%')

</if>

</where>

</select>

3.)使用trim标签解决；

1.后面多出的and或or 使用where标签不能解决，此时使用trim标签可以解决

<!--

 prefix=""前缀：trim标签体中是整个字符串拼接后的结果；

          prefix给拼接后的整个字符串加一个前缀。

 prefixOverrides=""前缀覆盖：去掉整个字符串前面多余的字符。

 suffix=""后缀：给拼接后的整个字符串加一个后缀。

 suffixOverrides=""后缀覆盖：去掉整个字符串后面多余的字符。

 <trim prefix="where" prefixOverrides="" suffix="" suffixOverrides="and"></trim>

-->

<select id="selectEmpList" resultType="com.itour.model.Employee">

select \* from tmp

<trim prefix="where" suffixOverrides="and|or">

<if test="job != null">

   job = #{job} and

</if>

<if test="ename!=null">

   ename like concat(concat('%',#{ename}),'%')  and

</if>

</trim>

</select>

## 2. choose(when,otherwise)-分支选择

choose(when,otherwise)相当于java中switch-case。

 <choose>

    <when test="job != null">

      AND job like #{job}

    </when>

    <when test="ename != null">

      AND ename like #{ename}

    </when>

    <otherwise>

       AND id = #{id}

    </otherwise>

  </choose>

## 3.trim字符串截取(where（封装查询条件）,set（封装修改条件）)

### 1.动态更新中遇到的问题；

1.动态修改中Mybatis没有帮忙处理逗号，可能会多出逗号。如果只带了ename，SQL：updat t\_emp set ename =？, where id =?;

<update id="updateEmployee" paramType="com.itour.model.Employee" >

 update t\_emp

  set

  <if test="ename!=null">

  ename =#{ename},

  </if>

  <if test="job!=null">

  job =#{job} ,

  </if>

  <if test="email!=null">

  email =#{email}

  </if>

 where id =#{id}

</update>

### 2.解决方案

1.使用set标签将更新条件包裹起来，set标签可以帮我去除多有逗号(,)

<update id="updateEmployee" paramType="com.itour.model.Employee" >

 update t\_emp

  <set>

   <if test="ename!=null">

   ename =#{ename},

  </if>

  <if test="job!=null">

   job =#{job} ,

  </if>

  <if test="email!=null">

   email =#{email}

  </if>

 </set>

 where id =#{id}

</update>

2.使用trim标签解决

<update id="updateEmployee">

 update t\_emp

  <trim  prefix="set" suffixOverrides=",">

   <if test="ename!=null">

   ename =#{ename},

  </if>

  <if test="job!=null">

   job =#{job} ,

  </if>

  <if test="email!=null">

   email =#{email}

  </if>

 </trim>

 where id =#{id}

</update>

## 4.foreach 遍历集合或Map

### 1.基本语法

1.collection:指定要遍历的集合。

2.item:将当前遍历出的元素赋值给指定的变量

3.#{变量}就能取出变量的值也就是当前遍历的元素。

4.separator：每个元素之间的分割符。

6.open:遍历出所有结果拼接一个开始的字符。

7.close:遍历出所有结果拼接一个结束的字符。

8.index:索引，遍历list的时候index是索引，item是指当前值。 遍历map的时候,index表示的是map的key，item就是map的值。

<foreach collection="ids" item="item" separator="," open="(" close=")" index="index">

#{item}

</foreach>

### 2.查询

1.Mapper接口

public List<Employee> selectEmpList(List<Long>);

2.XML

<select id="selectEmpList" resultType="com.itour.model.Employee">

select \* from t\_emp  where id in

<foreach collection="ids" item="item" separator="," open="(" close=")" >

#{item}

</foreach>

</select>

## 5.使用foreach实现批量保存

### 5.1mysql

1.MYSQL：支持values(),(),()...语法。

insert into emp(ename,job,email,deptno)

valus('tom','salsman','tom@gmail.com',''1379),

('mike','salsman','mike@gmail.com',''1379),

('amy','salsman','amy@gmail.com',''1379)

2.如果需要支持insert...;insert...; 需要在jdbc.url连接属性中配置allowMultiQueries=true;

3.mysql默认不支持多条SQL后使用";"分割,这种方式分割多个SQL可以用于其他批量操作(删除、修改)。

#### 1.使用values(),(),()...语法实现。

1.Mapper接口

public void insertEmpBatch(@param("emps") List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

insert into emp(ename,job,email,deptno)valus

<foreach collection="emps" item="item" separator=",">

(#{item.ename},#{item.job},#{item.email},#{item.deptno})

</foreach>

</insert>

#### 2.insert...;insert...;语法实现。

mysql默认不支持多条SQL后使用";"分割。需要在jdbc.url连接属性中配置allowMultiQueries=true;

1.Mapper接口

public void insertEmpBatch(@param("emps") List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

<foreach collection="emps" item="item" separator=";">

insert into emp(ename,job,email,deptno)valus(#{item.ename},#{item.job},#{item.email},#{item.deptno})

</foreach>

</insert>

### 5.2oracle

1.ORacle：不支持values(),(),()...语法;

2.Oracle支持的批量方式：  
1.)多Insert个sql放在在begin- end;中。

begin

insert into emp(id,ename,job,email,deptno)valus(seq\_emp.nextval,'tom','salsman','tom@gmail.com',''1379);

insert into emp(id,ename,job,email,deptno)valus(seq\_emp.nextval,'mik','salsman','mik@gmail.com',''1379);

end;

2.)利用中间表

insert into emp (id,ename,job,email,deptno)

select emp\_seq.nextval,ename,job,email ,deptno from (

select "amy" ename,'salsman' job,"wwangtao@163.com" email,"1760" deptno from dual

union

select "mike" ename,,'salsman' job,"mike@163.com" email,"1760" deptno from dual

)

#### 1.使用begin- end;语法实现

1.Mapper接口

public void insertEmpBatch(@param("emps")List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

<foreach collection="emps" item="item" open="begin" close="end;">

   insert into emp(id,ename,job,email,deptno)

   valus(seq\_emp.nextval,#{item.ename},#{item.job},#{item.email},#{item.deptno})

</foreach>

</insert>

#### 2.利用中间表来实现

1.Mapper接口

public void insertEmpBatch(@param("emps")List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

insert into emp (id,ename,job,email,deptno)

   select emp\_seq.nextval,ename,job,email ,deptno from (

      <foreach collection="emps" item="item" separator="union">

          select #{item.ename} ename,#{item.job},#{item.email} email,#{item.deptno} deptno from dual

       </foreach>

   )

</insert>

## 6.内置参数

不止是方法传递的参数可以用来判断、取值，mybatis默认提供了两个内置参数。

1.\_paramter:代表整个参数。

a.单个参数：\_paramter就是这个参数

b.多个参数：参数会被封装为一个map：\_paramter就是代表这个map;

2.\_databaseId:如果在Mybatis全局配置文件中配置了databaseIdProvider标签，\_databaseId就是代表当前数据库的别名。

<databaseIdProvider>

<!--为不同的数据库厂商起别名-->

<property name="MySQL" value="mysql"></property>

<property name="Oracle" value="oracle"></property>

<property name="SQL Server" value="sqlserver"></property>

</databaseIdProvider>

3.使用\_databaseId来根据不同的数据库查询对应的表数据；

<select id="getEmps" resultType ="com.itour.model.Employee">

 <if test="\_databaseId == oracle">

  select \* from emp

  <if test="\_paramter !=null">

    where ename like #{ename}

  </if>

 </if>

<if test="\_databaseId == mysql">

  select \* from employee

  <if test="\_paramter !=null">

    where ename like #{ename}

  </if>

 </if>

</select>

## 7.动态SQL绑定：

bind:可以见OGNL表达式的值绑定到一个变量中,方便引用;

<select id="getEmps" resultType ="com.itour.model.Employee">

 <!--bind:可以见OGNL表达式的值绑定到一个变量中,方便引用;

    将value的值赋值个name指定的变量，在需要使用的地方引用变量即可。

 -->

 <bind name="\_ename" value="'%'+ename+'%'"></bind>

 <if test="\_databaseId == oracle">

  select \* from emp

  <if test="\_paramter !=null">

    where ename like #{\_ename}

  </if>

 </if>

<if test="\_databaseId == mysql">

  select \* from employee

  <if test="\_paramter !=null">

    where ename like #{\_ename}

  </if>

 </if>

</select>

## 8.抽取可重用的SQL

1.使用sql标签抽取可重用的SQL,方便使用。

2.使用include标签引入外部SQL片段。

<sql id="selectEmpWhere">

  <if test="job!=null">

   and job =#{job}

  </if>

   <if test="ename!=null">

   and ename  like #{ename}

  </if>

</sql>

<select id="selectEmpList" resultType="com.itour.model.Employee">

   select \* from employee where 1=1 <include refid="selectEmpWhere"></include>

</select>

3.include标签里可以自定义一些property，SQL标签内部就能使用自定义的属性;

4.include-property取值的正确方式：$(prop)，不能使用#｛prop｝

<sql>

<if test="\_databaseId == oracle">

   empno,ename,job,deptno,${testcol}

 </if>

 <if test="\_databaseId == mysql">

   id,e\_name,e\_job,dept\_no

 </if>

</sql>

<select>

select <include refid="selectEmpWhere">

 <property name="testcol" value="sex"></property>

</include> from emp

</select>

# Mybatis实现批量插入

## 1.不支持自增的数据库(Oracle)

1.）Oracle：不支持values(),(),()...语法;

2.）Oracle支持的批量方式：

1.)多Insert个sql放在在begin- end;中。

begin

insert into emp(id,ename,job,email,deptno)valus(seq\_emp.nextval,'tom','salsman','tom@gmail.com',''1379);

insert into emp(id,ename,job,email,deptno)valus(seq\_emp.nextval,'mik','salsman','mik@gmail.com',''1379);

end;

2.)利用中间表

insert into emp (id,ename,job,email,deptno)

select emp\_seq.nextval,ename,job,email ,deptno from (

select "amy" ename,'salsman' job,"wwangtao@163.com" email,"1760" deptno from dual

union

select "mike" ename,,'salsman' job,"mike@163.com" email,"1760" deptno from dual

)

### 1.）使用begin- end;语法实现

1.Mapper接口

public void insertEmpBatch(@param("emps")List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

<foreach collection="emps" item="item" open="begin" close="end;">

   insert into emp(id,ename,job,email,deptno)

   valus(seq\_emp.nextval,#{item.ename},#{item.job},#{item.email},#{item.deptno})

</foreach>

</insert>

### 2.）利用中间表来实现

1.Mapper接口

public void insertEmpBatch(@param("emps")List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

insert into emp (id,ename,job,email,deptno)

   select emp\_seq.nextval,ename,job,email ,deptno from (

      <foreach collection="emps" item="item" separator="union">

          select #{item.ename} ename,#{item.job},#{item.email} email,#{item.deptno} deptno from dual

       </foreach>

   )

</insert>

## 2.支持自增的数据库(Mysql)

1.MYSQL：支持values(),(),()...语法。

insert into emp(ename,job,email,deptno)

valus('tom','salsman','tom@gmail.com',''1379),

('mike','salsman','mike@gmail.com',''1379),

('amy','salsman','amy@gmail.com',''1379)

2.如果需要支持insert...;insert...; 需要在jdbc.url连接属性中配置allowMultiQueries=true;

3.mysql默认不支持多条SQL后使用";"分割,这种方式分割多个SQL可以用于其他批量操作(删除、修改)。

### 1.）使用values(),(),()...语法实现。

1.Mapper接口

public void insertEmpBatch(@param("emps") List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

insert into emp(ename,job,email,deptno)valus

<foreach collection="emps" item="item" separator=",">

(#{item.ename},#{item.job},#{item.email},#{item.deptno})

</foreach>

</insert>

### 2.）insert...;insert...;语法实现。

mysql默认不支持多条SQL后使用";"分割。需要在jdbc.url连接属性中配置allowMultiQueries=true;

1.Mapper接口

public void insertEmpBatch(@param("emps") List<Employee> employee);

2.XML

<insert id="insertEmpBatch">

<foreach collection="emps" item="item" separator=";">

insert into emp(ename,job,email,deptno)valus(#{item.ename},#{item.job},#{item.email},#{item.deptno})

</foreach>

</insert>

**Mybatis缓存**

Mybatis提供了缓存的功能，可以非常方便的配置和定制，可以极大的提升查询的效率。

**1.Mybatis 缓存**

1.Mybatis默认定义两级缓存，一级缓存和二级缓存。

a.默认情况，只有一级缓存（sqlSession级别的缓存，也称为本地缓存）开启;

b.二级缓存需要手动开启和配置，是基于namespace级别的缓存(全局缓存);

c.为了提高拓展性，Mybatis定义了缓存接口Cache。可以通过实现Cache接口来自定义二级缓存;

**2.一级缓存**

1.一级缓存(本地缓存):与数据库同一次会话期间查询到的数据会放在本地缓存中。以后如果需要获取相同的数据，直接从缓存中获取，无需去查询数据。

2.一级缓存是sqlSession级别的缓存,是sqlSession级别的一个Map，是一直开启的;

**3.一级缓存失效的情况**

1.sqlSession不同；

2.sqlSession相同,查询条件不同(在一级缓存中没有);

3.sqlSession相同，两次查询之间执行了增删改操作（增删改对当前数据可能会造成影响）;

4.sqlSession相同，手动清除了一级缓存（清除缓存）;

**4.二级缓存**

二级缓存（全局缓存）是基于namespace级别的缓存，一个namespace对应一个二级缓存。

**5.二级缓存的工作机制**

1.一个会话，查询一条数据，这个数据就被放到当前会话的一级缓存中;

2.如果会话关闭,一级缓存中的数据会被保存二级缓存中；新的会话查询信息， 就可以参照二级缓存中对应的内容。

3.不同namespace查询出的数据会放在自己对应的缓存(Map)中。

**6.二级缓存的使用步骤**

1.开启全局二级缓存配置。

tip:显示的指定需要更改的配置的值，即是它是默认的，防止版本更新带来问题。

<settings>

  <setting name="cacheEnabled" value="true"/>

</settings>

2.在Mapper.xml中配置使用二级缓存 使用<cache></cache>标签配置;

<cache eviction="lru" flushInterval="6000" readOnly="false" size=""></cache>

   1.eviction:缓存的回收策略, 默认的是lru。

      lru:最近最少使用的，移除最长时间不被使用的对象;

      fifo:先进先出：按照对象进入的顺序来移除他们;

   soft:软引用，移除基于垃圾回收器的状态和软引用规则的对象；

   weak :弱引用：更积极的移除基于垃圾回收器状态和弱引用规则的对象；

   2.flushInterval:缓存刷新间隔。

    缓存多长时间清空一次，默认不清空,设置一个毫秒值

   3.readOnly：缓存是否只读，默认false。

     true:只读;Mybatis认为所有从缓存中获取的数据都是只读操作的，不会修改数据；

       Mybatis为了加速获取速度，直接将引用在缓存中的引用交给用户,不安全，速度快;

  false：非只读;Mybatis觉得获取的数据可能被修改。

        mybatis会利用序列号和反序列化技术克隆一份新数据给用户,安全，速度慢;

  4.size：缓存存放多少元素;

  5.type：指定自定义缓存的全类名；（实现Cach接口即可）

3.在POJO中实现序列化接口(Serializable)；

public class Employee implements Serializable{}

4.测试二级缓存

public class MybatisTest {

    @Test

    public void testSelectEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

      SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

   SqlSession openSession1 = sqlSessionFactory.openSession();

   SqlSession openSession2 = sqlSessionFactory.openSession();

*//3.获取接口的实现对象*

   EmpMapper mapper1 = openSession1.getMapper(EmpMapper.class);

   EmpMapper mapper2 = openSession2.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

   try {

*//1.查询单条*

    Emp selectOne1 = mapper1.selectEmp(7369L);

    System.out.println(selectOne1);

    openSession1.close();

*//第二次查询时从二级缓存中拿到的数据，并没有发新的SQL。*

    Emp selectOne2 = mapper2.selectEmp(7369L);

    System.out.println(selectOne2);

    openSession2.close();

   }catch (Exception e) {

*// TODO: handle exception*

    e.printStackTrace();

   }finally {

   }

  }

tip: 查询出来的数据都会被默认先放到一级缓存中， 只有会话提交或关闭后，一级缓存的数据才会转移到二级缓存中。

public class MybatisTest {

    @Test

    public void testSelectEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

      SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

   SqlSession openSession1 = sqlSessionFactory.openSession();

   SqlSession openSession2 = sqlSessionFactory.openSession();

*//3.获取接口的实现对象*

   EmpMapper mapper1 = openSession1.getMapper(EmpMapper.class);

   EmpMapper mapper2 = openSession2.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

   try {

*//1.查询单条*

    Emp selectOne1 = mapper1.selectEmp(7369L);

    System.out.println(selectOne1);

*//第二次查询时会发新的SQL。*

    Emp selectOne2 = mapper2.selectEmp(7369L);

    System.out.println(selectOne2);

    openSession1.close();

    openSession2.close();

   }catch (Exception e) {

*// TODO: handle exception*

    e.printStackTrace();

   }finally {

   }

  }

private static SqlSessionFactory getSqlSessionFactory() throws IOException {

 String resource = "conf/mybatis-conf.xml";

 Reader reader = Resources.getResourceAsReader(resource);

 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);

 return sqlSessionFactory;

}

}

**7.缓存相关设置和属性：**

1.cacheEnabled=true表示开启缓存|false表示关闭二级缓存（一级缓存一直可用）

<settings>

<setting name="cacheEnabled" value="true"/>

</settings>

2.每个select标签都有useCache="true"属性,设置为false表示关闭二级缓存（一级缓存依然可以使用，二级缓存不能使用）

<select id="selectEmp" useCache="true"></select>

**3.每个增删改的标签的：flushCache ="true"属性, 增删改完成后就会清除缓存（一级缓存会失效，二级缓存也会被清空）。**

<delete id="delEmpById" flushCache ="true"></delete>

<update id="updateEmpById" flushCache ="true"></update>

4.每个select标签：flushCache ="false",如果改为true,每次查询之前都会清空缓存，导致查询不会被使用。

<select id="selectEmp" flushCache ="false"></select>

5.sqlsession.clearCache()只是清除当前session的一级缓存。

public class MybatisTest {

    @Test

    public void testSelectEmp() throws IOException  {

*//1.获取SqlSessionFactory对象*

      SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();

*//2.获取sqlSession对象*

   SqlSession openSession1 = sqlSessionFactory.openSession();

   SqlSession openSession2 = sqlSessionFactory.openSession();

*//3.获取接口的实现对象*

   EmpMapper mapper1 = openSession1.getMapper(EmpMapper.class);

   EmpMapper mapper2 = openSession2.getMapper(EmpMapper.class);

*//4.调用增删改查方法*

   try {

*//1.查询单条*

    Emp selectOne1 = mapper1.selectEmp(7369L);

    System.out.println(selectOne1);

*//第二次查询时会发新的SQL。*

    Emp selectOne2 = mapper2.selectEmp(7369L);

    System.out.println(selectOne2);

    openSession1.close();

    openSession2.close();

   }catch (Exception e) {

*// TODO: handle exception*

    e.printStackTrace();

   }finally {

   }

  }

private static SqlSessionFactory getSqlSessionFactory() throws IOException {

 String resource = "conf/mybatis-conf.xml";

 Reader reader = Resources.getResourceAsReader(resource);

 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);

 return sqlSessionFactory;

}

}

6.localCacheScop:本地缓存作用域(一级缓存SESSION:当期会话的)STATEMENT:可以禁用一级缓存

<settings>

<setting name="localCacheScope" value="SESSION"/>

</settings>

**Mybatis如何通过配置文件构建SqlSessionFactory**

1.通过配置文件方式构建SqlSessionFactory

**Java代码**

private static SqlSessionFactory getSqlSessionFactory() throws IOException {

 String resource = "conf/mybatis-conf.xml";

 Reader reader = Resources.getResourceAsReader(resource);

 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);

 return sqlSessionFactory;

}

**相关源码**

1.)使用构建者模式构建SqlSessionFactory

public class SqlSessionFactoryBuilder {

*//1.使用builder模式构建SqlSessionFactory*

  public SqlSessionFactory build(Reader reader) {

    return build(reader, null, null);

  }

*//2.通过流构建SqlSessionFactory*

  public SqlSessionFactory build(Reader reader, String environment, Properties properties) {

    try {

*//2.1.构建XMLConfigBuilder解析器,解析XML配置文件.*

      XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);

*//2.2.解析Mybatis的配置文件和SQL映射文件，将配置文件信息保存到Configuration对象中，*

*//并返回包含了Configuration对象的DefaultSqlSessionFactory对象。*

      return build(parser.parse());

    } catch (Exception e) {

      throw ExceptionFactory.wrapException("Error building SqlSession.", e);

    } finally {

      ErrorContext.instance().reset();

      try {

       if (reader != null) {

         reader.close();

       }

      } catch (IOException e) {

*// Intentionally ignore. Prefer previous error.*

      }

    }

  }

*//3.返回SqlSessionFactory的默认实现类DefaultSqlSessionFactory*

  public SqlSessionFactory build(Configuration config) {

    return new DefaultSqlSessionFactory(config);

  }

}

2.)解析Mybatis的配置文件和SQL映射文件，将配置文件信息保存到Configuration对象中，并返回。

public class XMLConfigBuilder extends BaseBuilder {

public Configuration parse() {

    if (parsed) {

      throw new BuilderException("Each XMLConfigBuilder can only be used once.");

    }

    parsed = true;

    parseConfiguration(parser.evalNode("/configuration"));

    return configuration;

  }

private void parseConfiguration(XNode root) {

    try {

*// issue #117 read properties first*

      propertiesElement(root.evalNode("properties"));

      Properties settings = settingsAsProperties(root.evalNode("settings"));

      loadCustomVfs(settings);

      loadCustomLogImpl(settings);

      typeAliasesElement(root.evalNode("typeAliases"));

      pluginElement(root.evalNode("plugins"));

      objectFactoryElement(root.evalNode("objectFactory"));

      objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));

      reflectorFactoryElement(root.evalNode("reflectorFactory"));

      settingsElement(settings);

*// read it after objectFactory and objectWrapperFactory issue #631*

      environmentsElement(root.evalNode("environments"));

      databaseIdProviderElement(root.evalNode("databaseIdProvider"));

      typeHandlerElement(root.evalNode("typeHandlers"));

      mapperElement(root.evalNode("mappers"));

    } catch (Exception e) {

      throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);

    }

  }

}

**Mybatis如何加载Mapper**

**1.解析Mappers标签**

private void mapperElement(XNode parent) throws Exception {

*//1.如果配置了<mappers></mappers>*

    if (parent != null) {

*//2.获取<mappers></mappers>标签的子标签*

      for (XNode child : parent.getChildren()) {

*//3.如果是<package name="com.itour.mapping" />标签*

        if ("package".equals(child.getName())) {

*//4.获取<package name="com.itour.mapping" />标签中name属性的值;*

          String mapperPackage = child.getStringAttribute("name");

*//5.加载mapperPackage包下的所有类*

          configuration.addMappers(mapperPackage);

        } else {

*//3.如果不是package标签，获取mapper标签标签的属性值*

*//4.1获取<mapper resource="com/mapping/AuthorMapper.xml"></mapper>标签的resource属性*

          String resource = child.getStringAttribute("resource");

*//4.2获取<mapper url="file:///var/mappers/AuthorMapper.xml"></mapper>标签的url属性*

          String url = child.getStringAttribute("url");

*//4.3获取<mapper class="com.mapping.AuthorMapper"></mapper>标签的class属性*

          String mapperClass = child.getStringAttribute("class");

          if (resource != null && url == null && mapperClass == null) {

*//5.如果是<mapper resource="" > </mapper>标签.*

            ErrorContext.instance().resource(resource);

            try(InputStream inputStream = Resources.getResourceAsStream(resource)) {

*//6.解析Mapper标签,根据Mapper标签解析对应的Mapper.xml文件*

              XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream, configuration, resource, configuration.getSqlFragments());

              mapperParser.parse();

            }

          } else if (resource == null && url != null && mapperClass == null) {

*//5.如果是<mapper url="" > </mapper>标签.*

            ErrorContext.instance().resource(url);

            try(InputStream inputStream = Resources.getUrlAsStream(url)){

*//6.解析Mapper标签,根据Mapper标签解析对应的Mapper.xml文件*

              XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream, configuration, url, configuration.getSqlFragments());

              mapperParser.parse();

            }

          } else if (resource == null && url == null && mapperClass != null) {

*//5.如果是<mapper class="" > </mapper>标签.*

            Class<?> mapperInterface = Resources.classForName(mapperClass);

*//6.加载对应的类*

            configuration.addMapper(mapperInterface);

          } else {

*//5.如果配置了mapper标签,没有设置url,resource,class属性，抛出异常。*

            throw new BuilderException("A mapper element may only specify a url, resource or class, but not more than one.");

          }

        }

      }

    }

  }

## 2.加载Mapper中的package标签

1.使用Configuration对象中的addMappers()，通过mapperRegistry属性来加载Mapper。

  public class Configuration {

    protected final MapperRegistry mapperRegistry = new MapperRegistry(this);

*//调用MapperRegistry中addMappers()*

    public void addMappers(String packageName) {

     mapperRegistry.addMappers(packageName);

   }

  }

2.MapperRegistry类中主要包含了config和knownMappers两个属性。

public class MapperRegistry {

   private final Configuration config;

   private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap<>();

   public MapperRegistry(Configuration config) {

     this.config = config;

   }

*/\*\**

    \* Adds the mappers.

    \*添加指定包下的Mapper

    \* @param packageName

    \*          the package name

    \* @since 3.2.2

    \*/

   public void addMappers(String packageName) {

*//加载packageName包下的所有类*

     addMappers(packageName, Object.class);

   }

*/\*\**

    \* Adds the mappers.

    \*获取指定包下的所有Mapper并加载

    \* @param packageName

    \*          the package name

    \* @param superType

    \*          the super type

    \* @since 3.2.2

    \*/

   public void addMappers(String packageName, Class<?> superType) {

     ResolverUtil<Class<?>> resolverUtil = new ResolverUtil<>();

     resolverUtil.find(new ResolverUtil.IsA(superType), packageName);

     Set<Class<? extends Class<?>>> mapperSet = resolverUtil.getClasses();

     for (Class<?> mapperClass : mapperSet) {

       addMapper(mapperClass);

     }

   }

 public <T> void addMapper(Class<T> type) {

     if (type.isInterface()) {

*//Mybatis配置文件同时支持Mapper和Package标签,但是Mapper标签必须在前面,且Package标签不能包含Mapper。*

       if (hasMapper(type)) {

         throw new BindingException("Type " + type + " is already known to the MapperRegistry.");

       }

       boolean loadCompleted = false;

       try {

*//根据Mapper实例化一个工厂对象,放入Map中，方便使用时获取。*

*//key:interface com.stu.mybatis.mapper.HydaMapper*

*//value:org.apache.ibatis.binding.MapperProxyFactory@5829e4f4*

         knownMappers.put(type, new MapperProxyFactory<>(type));

*// It's important that the type is added before the parser is run*

*// otherwise the binding may automatically be attempted by the*

*// mapper parser. If the type is already known, it won't try.*

*//处理Mapper注解,Mybatis支持使用注解将SQL写入接口中。*

        MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);

         parser.parse();

         loadCompleted = true;

       } finally {

*//如果注解出现异常需要移除*

         if (!loadCompleted) {

           knownMappers.remove(type);

         }

       }

     }

   }

public <T> boolean hasMapper(Class<T> type) {

     return knownMappers.containsKey(type);

    }

 }

**3.加载Mappers中的mapper标签**

public class XMLMapperBuilder extends BaseBuilder {

public void parse() {

    if (!configuration.isResourceLoaded(resource)) {

*//解析Mapper标签*

      configurationElement(parser.evalNode("/mapper"));

      configuration.addLoadedResource(resource);

      bindMapperForNamespace();

    }

    parsePendingResultMaps();

    parsePendingCacheRefs();

    parsePendingStatements();

  }

  private void configurationElement(XNode context) {

    try {

*//namespace必须设置，否则抛异常*

      String namespace = context.getStringAttribute("namespace");

      if (namespace == null || namespace.isEmpty()) {

        throw new BuilderException("Mapper's namespace cannot be empty");

      }

      builderAssistant.setCurrentNamespace(namespace);

      cacheRefElement(context.evalNode("cache-ref"));

*//解析cache标签*

      cacheElement(context.evalNode("cache"));

*//解析入参parameterMap*

      parameterMapElement(context.evalNodes("/mapper/parameterMap"));

*//解析resultMap*

      resultMapElements(context.evalNodes("/mapper/resultMap"));

*//解析SQL*

      sqlElement(context.evalNodes("/mapper/sql"));

*//解析CRUD，构建Stament*

      buildStatementFromContext(context.evalNodes("select|insert|update|delete"));

    } catch (Exception e) {

      throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource + "'. Cause: " + e, e);

    }

  }

}

**Mybatis是如何解析XML**

**1.构建XMLConfigBuilder来解析Mybatis的配置文件**

1.构建SqlSessionFactory之前会构建XMLConfigBuilder来解析Mybatis的配置文件

public class SqlSessionFactoryBuilder {

*//1.创建SqlSessionFactory*

public SqlSessionFactory build(Reader reader, String environment, Properties properties) {

    try {

*//构建XMLConfigBuilder*

      XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);

*//解析Mybatis的配置文件，并返回SqlSessionFactory*

      return build(parser.parse());

    } catch (Exception e) {

      throw ExceptionFactory.wrapException("Error building SqlSession.", e);

    } finally {

      ErrorContext.instance().reset();

      try {

       if (reader != null) {

         reader.close();

       }

      } catch (IOException e) {

*// Intentionally ignore. Prefer previous error.*

      }

    }

  }

}

**Mybatis如何构建XMLConfigBuilder**

1.使用构造函数方式构建XMLConfigBuilde，将XPathParser对象作为属性(parser)放入XMLConfigBuilde对象中

public class XMLConfigBuilder extends BaseBuilder {

*//1.使用构造函数方式构建XMLConfigBuilder*

  public XMLConfigBuilder(Reader reader, String environment, Properties props) {

*//a.构建XPathParser*

    this(new XPathParser(reader, true, props, new XMLMapperEntityResolver()), environment, props);

  }

*//2.将XPathParser放入XMLConfigBuilder对象中*

   private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {

*//创建Configuration对象*

    super(new Configuration());

    ErrorContext.instance().resource("SQL Mapper Configuration");

    this.configuration.setVariables(props);

    this.parsed = false;

    this.environment = environment;

    this.parser = parser;

  }

*//3.解析Mybatis全局配置文件*

   public Configuration parse() {

    if (parsed) {

      throw new BuilderException("Each XMLConfigBuilder can only be used once.");

    }

    parsed = true;

*//解析Mybatis全局配置文件,parser.evalNode("/configuration")获取configuration下的所有节点。*

    parseConfiguration(parser.evalNode("/configuration"));

    return configuration;

  }

*//4.解析Mybatis全局配置文件中的各个元素*

  private void parseConfiguration(XNode root) {

    try {

*// issue #117 read properties first*

      propertiesElement(root.evalNode("properties"));

      Properties settings = settingsAsProperties(root.evalNode("settings"));

      loadCustomVfs(settings);

      loadCustomLogImpl(settings);

      typeAliasesElement(root.evalNode("typeAliases"));

      pluginElement(root.evalNode("plugins"));

      objectFactoryElement(root.evalNode("objectFactory"));

      objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));

      reflectorFactoryElement(root.evalNode("reflectorFactory"));

      settingsElement(settings);

*// read it after objectFactory and objectWrapperFactory issue #631*

      environmentsElement(root.evalNode("environments"));

      databaseIdProviderElement(root.evalNode("databaseIdProvider"));

      typeHandlerElement(root.evalNode("typeHandlers"));

      mapperElement(root.evalNode("mappers"));

    } catch (Exception e) {

      throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);

    }

  }

}

**XPathParser对象构建**

1.使用构造函数方式构建XPathParser;

2.commonConstructor()中实例化XPath对象，作为属性（xpath）保存到XPathParser对象中，用来查询XML节点信息。

3.createDocument()中实例化基于Dom的Document对象并作为属性（document）保存到XPathParser对象中。

public class XPathParser {

*//使用构造函数方式构建XPathParser*

  public XPathParser(Reader reader, boolean validation, Properties variables, EntityResolver entityResolver) {

    commonConstructor(validation, variables, entityResolver);

    this.document = createDocument(new InputSource(reader));

  }

*//实例化XPath对象(用于解析xml)*

  private void commonConstructor(boolean validation, Properties variables, EntityResolver entityResolver) {

    this.validation = validation;

    this.entityResolver = entityResolver;

    this.variables = variables;

*//实例化XPath对象*

    XPathFactory factory = XPathFactory.newInstance();

    this.xpath = factory.newXPath();

  }

*//5.使用Dom方式构建Document对象，用于解析Mybatis配置文件*

   private Document createDocument(InputSource inputSource) {

*// important: this must only be called AFTER common constructor*

    try {

      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

      factory.setFeature(XMLConstants.FEATURE\_SECURE\_PROCESSING, true);

      factory.setValidating(validation);

      factory.setNamespaceAware(false);

      factory.setIgnoringComments(true);

      factory.setIgnoringElementContentWhitespace(false);

      factory.setCoalescing(false);

      factory.setExpandEntityReferences(true);

      DocumentBuilder builder = factory.newDocumentBuilder();

      builder.setEntityResolver(entityResolver);

      builder.setErrorHandler(new ErrorHandler() {

        @Override

        public void error(SAXParseException exception) throws SAXException {

          throw exception;

        }

        @Override

        public void fatalError(SAXParseException exception) throws SAXException {

          throw exception;

        }

        @Override

        public void warning(SAXParseException exception) throws SAXException {

*// NOP*

        }

      });

      return builder.parse(inputSource);

    } catch (Exception e) {

      throw new BuilderException("Error creating document instance.  Cause: " + e, e);

    }

  }

}

**总结**

1.Mybatis底层使用了Dom基于XPath的方式来解析XML。Mybatis对Dom中的Node进行了封装（XNode）。

2.Mybatis通过构建XMLConfigBuilder来实现对Mybatis配置文件元素的解析，然后将解析的内容放入Configuration对象中。