#### 第十四题

|  |
| --- |
| 题号：第十四题  类型：编程题  题干：  请完善自定义持久层框架IPersistence，在现有代码基础上添加修改及删除功能  答案：   1. 修改XMLMapperBuilder的parse方法：解析<update><delete>标签 2. 在SqlSession及DefalutSqlSession添加update及delete方法 3. 在getMapper方法中对当前执行的sql语句进行判断，决定调用增删改查的那个方法 4. 在Executor及simpleExecutor中添加update及delete方法   代码详情：见附件 |

XMLMapperBuilder:

|  |
| --- |
| package com.lagou.config;  import java.io.InputStream;  import java.util.List;  public class XMLMapperBuilder {  private Configuration configuration;  public XMLMapperBuilder(Configuration configuration) {  this.configuration =configuration;  }  public void parse(InputStream inputStream) throws DocumentException {  Document document = new SAXReader().read(inputStream);  Element rootElement = document.getRootElement();  String namespace = rootElement.attributeValue("namespace");  List<Element> selectlist = rootElement.selectNodes("//select");  Elementforeach(selectlist,namespace);  List<Element> updatelist = rootElement.selectNodes("//update");  Elementforeach(updatelist,namespace);  List<Element> deletelist = rootElement.selectNodes("//delete");  Elementforeach(deletelist,namespace);  }  public void Elementforeach(List<Element> list,String namespace){  for (Element element : list) {  String id = element.attributeValue("id");  String resultType = element.attributeValue("resultType");  String paramterType = element.attributeValue("paramterType");  String sqlText = element.getTextTrim();  MappedStatement mappedStatement = new MappedStatement();  mappedStatement.setId(id);  mappedStatement.setResultType(resultType);  mappedStatement.setParamterType(paramterType);  mappedStatement.setSql(sqlText);  String key = namespace+"."+id;  configuration.getMappedStatementMap().put(key,mappedStatement);  }  }  } |

SqlSession

|  |
| --- |
| //根据条件进行修改  public Integer update(String statementid,Object... params) throws Exception;  //根据条件进行删除  public Integer delete(String statementid,Object... params) throws Exception; |

DefaultSqlSession

|  |
| --- |
| public class DefaultSqlSession implements SqlSession {  private Configuration configuration;  public DefaultSqlSession(Configuration configuration) {  this.configuration = configuration;  }  @Override  public <E> List<E> selectList(String statementid, Object... params) throws Exception {  //将要去完成对simpleExecutor里的query方法的调用  simpleExecutor simpleExecutor = new simpleExecutor();  MappedStatement mappedStatement = configuration.getMappedStatementMap().get(statementid);  List<Object> list = simpleExecutor.query(configuration, mappedStatement, params);  return (List<E>) list;  }  @Override  public <T> T selectOne(String statementid, Object... params) throws Exception {  List<Object> objects = selectList(statementid, params);  if(objects.size()==1){  return (T) objects.get(0);  }else {  throw new RuntimeException("查询结果为空或者返回结果过多");  }  }  @Override  public Integer update(String statementid, Object... params) throws Exception {  Executor simpleExecutor = new simpleExecutor();  MappedStatement mappedStatement = configuration.getMappedStatementMap().get(statementid);  Integer i = simpleExecutor.update(configuration, mappedStatement, params);  return i;  }  @Override  public Integer delete(String statementid, Object... params) throws Exception {  Executor simpleExecutor = new simpleExecutor();  MappedStatement mappedStatement = configuration.getMappedStatementMap().get(statementid);  Integer i = simpleExecutor.delete(configuration, mappedStatement, params);  return i;  }  @Override  public <T> T getMapper(Class<?> mapperClass) {  // 使用JDK动态代理来为Dao接口生成代理对象，并返回  Object proxyInstance = Proxy.newProxyInstance(DefaultSqlSession.class.getClassLoader(), new Class[]{mapperClass}, new InvocationHandler() {  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  // 底层都还是去执行JDBC代码 //根据不同情况，来调用selctList或者selectOne  // 准备参数 1：statmentid :sql语句的唯一标识：namespace.id= 接口全限定名.方法名  // 方法名：findAll  String methodName = method.getName();  String className = method.getDeclaringClass().getName();  String statementId = className+"."+methodName;  // 准备参数2：params:args  // 获取被调用方法的返回值类型  Type genericReturnType = method.getGenericReturnType();  // 判断是否进行了 泛型类型参数化  if(genericReturnType instanceof ParameterizedType){  List<Object> objects = selectList(statementId, args);  return objects;  }  MappedStatement mappedStatement = configuration.getMappedStatementMap().get(statementId);  String sql = mappedStatement.getSql();  if(sql.startsWith("update")){  return update(statementId,args);  }  if(sql.startsWith("delete")){  return delete(statementId,args);  }  return selectOne(statementId,args);  }  });  return (T) proxyInstance;  }  } |

simpleExecutor

|  |
| --- |
| package com.lagou.sqlSession;  import com.lagou.config.BoundSql;  import com.lagou.pojo.Configuration;  import com.lagou.pojo.MappedStatement;  import com.lagou.utils.GenericTokenParser;  import com.lagou.utils.ParameterMapping;  import com.lagou.utils.ParameterMappingTokenHandler;  import java.beans.PropertyDescriptor;  import java.lang.reflect.Field;  import java.lang.reflect.Method;  import java.sql.\*;  import java.util.ArrayList;  import java.util.List;  public class simpleExecutor implements Executor {  @Override //user  public <E> List<E> query(Configuration configuration, MappedStatement mappedStatement, Object... params) throws Exception {  // 1. 注册驱动，获取连接  Connection connection = configuration.getDataSource().getConnection();  // 2. 获取sql语句 : select \* from user where id = #{id} and username = #{username}  //转换sql语句： select \* from user where id = ? and username = ? ，转换的过程中，还需要对#{}里面的值进行解析存储  String sql = mappedStatement.getSql();  BoundSql boundSql = getBoundSql(sql);  // 3.获取预处理对象：preparedStatement  PreparedStatement preparedStatement = connection.prepareStatement(boundSql.getSqlText());  // 4. 设置参数  //获取到了参数的全路径  String paramterType = mappedStatement.getParamterType();  Class<?> paramtertypeClass = getClassType(paramterType);  List<ParameterMapping> parameterMappingList = boundSql.getParameterMappingList();  for (int i = 0; i < parameterMappingList.size(); i++) {  ParameterMapping parameterMapping = parameterMappingList.get(i);  String content = parameterMapping.getContent();  //反射  Field declaredField = paramtertypeClass.getDeclaredField(content);  //暴力访问  declaredField.setAccessible(true);  Object o = declaredField.get(params[0]);  preparedStatement.setObject(i+1,o);  }  // 5. 执行sql  ResultSet resultSet = preparedStatement.executeQuery();  String resultType = mappedStatement.getResultType();  Class<?> resultTypeClass = getClassType(resultType);  ArrayList<Object> objects = new ArrayList<>();  // 6. 封装返回结果集  while (resultSet.next()){  Object o =resultTypeClass.newInstance();  //元数据  ResultSetMetaData metaData = resultSet.getMetaData();  for (int i = 1; i <= metaData.getColumnCount(); i++) {  // 字段名  String columnName = metaData.getColumnName(i);  // 字段的值  Object value = resultSet.getObject(columnName);  //使用反射或者内省，根据数据库表和实体的对应关系，完成封装  PropertyDescriptor propertyDescriptor = new PropertyDescriptor(columnName, resultTypeClass);  Method writeMethod = propertyDescriptor.getWriteMethod();  writeMethod.invoke(o,value);  }  objects.add(o);  }  return (List<E>) objects;  }  @Override  public Integer update(Configuration configuration, MappedStatement mappedStatement, Object[] params) throws Exception {  // 1. 注册驱动，获取连接  Connection connection = configuration.getDataSource().getConnection();  // 2. 获取sql语句 : select \* from user where id = #{id} and username = #{username}  //转换sql语句： select \* from user where id = ? and username = ? ，转换的过程中，还需要对#{}里面的值进行解析存储  String sql = mappedStatement.getSql();  BoundSql boundSql = getBoundSql(sql);  // 3.获取预处理对象：preparedStatement  PreparedStatement preparedStatement = connection.prepareStatement(boundSql.getSqlText());  // 4. 设置参数  //获取到了参数的全路径  String paramterType = mappedStatement.getParamterType();  Class<?> paramtertypeClass = getClassType(paramterType);  List<ParameterMapping> parameterMappingList = boundSql.getParameterMappingList();  for (int i = 0; i < parameterMappingList.size(); i++) {  ParameterMapping parameterMapping = parameterMappingList.get(i);  String content = parameterMapping.getContent();  //反射  Field declaredField = paramtertypeClass.getDeclaredField(content);  //暴力访问  declaredField.setAccessible(true);  Object o = declaredField.get(params[0]);  preparedStatement.setObject(i+1,o);  }  // 5. 执行sql  int i = preparedStatement.executeUpdate();  return i;  }  @Override  public Integer delete(Configuration configuration, MappedStatement mappedStatement, Object[] params) throws Exception {  // 1. 注册驱动，获取连接  Connection connection = configuration.getDataSource().getConnection();  // 2. 获取sql语句 : select \* from user where id = #{id} and username = #{username}  //转换sql语句： select \* from user where id = ? and username = ? ，转换的过程中，还需要对#{}里面的值进行解析存储  String sql = mappedStatement.getSql();  BoundSql boundSql = getBoundSql(sql);  // 3.获取预处理对象：preparedStatement  PreparedStatement preparedStatement = connection.prepareStatement(boundSql.getSqlText());  // 4. 设置参数  //获取到了参数的全路径  String paramterType = mappedStatement.getParamterType();  Class<?> paramtertypeClass = getClassType(paramterType);  if(paramtertypeClass == Integer.class){  preparedStatement.setObject(1, params[0]);  }else {  List<ParameterMapping> parameterMappingList = boundSql.getParameterMappingList();  for (int i = 0; i < parameterMappingList.size(); i++) {  ParameterMapping parameterMapping = parameterMappingList.get(i);  String content = parameterMapping.getContent();  //反射  Field declaredField = paramtertypeClass.getDeclaredField(content);  //暴力访问  declaredField.setAccessible(true);  Object o = declaredField.get(params[0]);  preparedStatement.setObject(i + 1, o);  }  }  // 5. 执行sql  int i = preparedStatement.executeUpdate();  return i;  }  private Class<?> getClassType(String paramterType) throws ClassNotFoundException {  if(paramterType!=null){  Class<?> aClass = Class.forName(paramterType);  return aClass;  }  return null;  }  /\*\*  \* 完成对#{}的解析工作：1.将#{}使用？进行代替，2.解析出#{}里面的值进行存储  \* @param sql  \* @return  \*/  private BoundSql getBoundSql(String sql) {  //标记处理类：配置标记解析器来完成对占位符的解析处理工作  ParameterMappingTokenHandler parameterMappingTokenHandler = new ParameterMappingTokenHandler();  GenericTokenParser genericTokenParser = new GenericTokenParser("#{", "}", parameterMappingTokenHandler);  //解析出来的sql  String parseSql = genericTokenParser.parse(sql);  //#{}里面解析出来的参数名称  List<ParameterMapping> parameterMappings = parameterMappingTokenHandler.getParameterMappings();  BoundSql boundSql = new BoundSql(parseSql,parameterMappings);  return boundSql;  }  } |