**前台**

**后台业务层**

数据库

数据层

业务层

控制层

显示层

**实例分析:**

Emp表(empno,ename,job,hiredate,sal,comm),需求:

1. 【业务层】实现雇员数据的添加,但是需要保证被添加的雇员编号不会重复

|-【数据层】判断要增加的雇员编号是否存在

|-【数据层】如果雇员编号不存在,则进行数据的保存操作

1. 【业务层】实现雇员数据的修改操作

|-【数据层】执行数据的修改操作

1. 【业务层】实现多个雇员数据的删除操作

|-【数据层】执行雇员的限定删除操作

1. 【业务层】可以根据雇员编号查找到一个雇员信息

|-【数据层】根据雇员编号查询指定的雇员数据

1. 【业务层】可以查询所有雇员的信息

|-【数据层】查询全部雇员数据

1. 【业务层】可以实现的分页显示(模糊查询),同时可以返回所有的雇员数量

|-【数据层】雇员数据的分页查询

|-【数据层】使用count()函数统计所有雇员数目

结论:用户提出的所有需求都应该划分为业务层,因为他指的是功能,而开发人员应该

根据业务层,去进行数据层的设计

父包cn.mldn

子包根据不同的功能模块进行划分

## 数据库连接类

保存在dbc子包中

既然要进行数据库的开发,那么就必须进行数据库的连接取得与关闭才能正常操作,几乎所有数据库的连接和关闭都是固定的步骤,那么就可以单独定义一个DatabaseConnection类,这个类主要负责

1. 数据库连接对象的取得

getConnection():Connection

1. 数据库的关闭操作

close():void

|  |
| --- |
| DatabaseConnection |
| DBDRIVER = ” com.mysql.jdbc.Driver”  DBURL = "jdbc:mysql://118.89.54.105:3306/test";  DBUSER = ”lw”;  PASSWORD = ”linwei”;  Connection conn = null |
| +getConnection():Connection  +close():void  +DatabaseConnection |

范例:定义数据库的连接类

|  |
| --- |
| package cn.mldn.dbc;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  /\*\*  \* 本类专门负责数据库的连接与关闭操作,在实例化本类对象时就意味着要进行  \* 数据库的开发,所以在本类的构造方法里要进行数据库驱动加载和数据库连接取得  \*/  public class DatabaseConnection {  private static final String DBDRIVER = " com.mysql.jdbc.Driver";  private static final String DBURL = "jdbc:mysql://118.89.54.105:3306/test";  private static final String DBUSER = "lw";  private static final String PASSWORD = "linwei";  private Connection conn = null;  /\*\*  \* 在构造方法里为Conn对象进行实例化,可以直接取得数据库的连接对象  \* 由于所有的操作都是基于数据库完成了,如果连接失败,所有的操作都可以停止了  \*/  public DatabaseConnection() {  try {  Class.forName(DBDRIVER);  this.conn = DriverManager.getConnection(DBURL, DBUSER, PASSWORD);  }catch (Exception e){//虽然有异常,但是抛出的意义不大  e.printStackTrace();  }  }  /\*\*  \* 取得一个数据库的连接对象  \* @return Connection实例对象  \*/  public Connection getConnection(){  return this.conn;  }  /\*\*  \* 负责数据库的关闭  \*/  public void close(){  if(this.conn != null){ //表示存在有连接对象  try {  this.conn.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  } |

**额外话题:还会考虑到多数据库间移植的问题,此时需要设置一个专门的表示连接标准的接口**

|  |
| --- |
| DatabaseConnection  <<interface>> |
|  |
| +getConnection():Connection  +close():void |

|  |
| --- |
| DatabaseConnectionFactory |
|  |
|  |

dbc.properties

db=mysql

|  |
| --- |
| OracleDatabaseConnection |
|  |
|  |

|  |
| --- |
| MySQLDatabaseConnection |
|  |
|  |

|  |
| --- |
| DatabaseConnection  <<interface>> |
|  |
| +getConnection():Connection  +close():void |

这种方式太麻烦了,第三方框架越来越完善了,就不不自己设计了

## 开发Value Object

数据的结构必须要与表的结构一一对应,自然会想到简单java类

注意:在实际工作中,针对于简单java类的开发,给出如下的要求:

1. 可能出现分布式的问题,所以简单java类必须实现java.io.Serializable接口
2. 简单Java类的名称必须与表名称一致,例如表名称student\_info,类名称:StudentInfo
3. 类中的属性不允许使用基本数据类型,必须使用基本类型的包装类,因为基本类型的默认值为0,包装类型默认值是null
4. 类中可以定义有多个构造方法,但是要保留一个无参构造方法

|  |
| --- |
| package cn.mldn.vo;  import java.io.Serializable;  import java.util.Date;  public class Emp implements Serializable {  private Integer empno;  private String name;  private String job;  private Date hiredate;  private Double sal;  private Double comm;  public Integer getEmpno() {  return empno;  }  public void setEmpno(Integer empno) {  this.empno = empno;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getJob() {  return job;  }  public void setJob(String job) {  this.job = job;  }  public Date getHiredate() {  return hiredate;  }  public void setHiredate(Date hiredate) {  this.hiredate = hiredate;  }  public Double getSal() {  return sal;  }  public void setSal(Double sal) {  this.sal = sal;  }  public Double getComm() {  return comm;  }  public void setComm(Double comm) {  this.comm = comm;  }  } |

不管有多少张表,只要是实体表,那么一定要写简单java类,而且不要

试图想着一次性将所有表都转换到位

## 开发数据层

数据层最终是交给业务层进行调用的,所以业务层必须知道数据层执行标准,

即:业务层需要明确的知道数据层的操作方法,但是不需要知道它的具体实现

数据层

数据层标准

（接口）

业务层

|  |
| --- |
| IEmpDAO  <<interface>> |
|  |
| +doCreate(Emp vo):boolean  +doUpdate(Emp vo):boolean  +doRemoveBatch(Set<Integer> ids):boolean  +findById(Integer id):Emp  +finaAll():List<Emp>  +findAllSplit(Integer currentPage,Integer lineSize,String column,String keyWord): List<Emp>  + getAllcount:Integer |

|  |
| --- |
| EmpDAOImpl |
|  |
| EmpDAOImpl(Connection conn):Connection |

|  |
| --- |
| Emp |
|  |
|  |

|  |
| --- |
| Connection |
|  |
|  |

|  |
| --- |
| DAOFactory |
|  |
| getIEmpDAOInstance(Connection conn):IEmpDAO |

业务层

### 3.1开发数据层标准

不同层如果要进行访问,那么要提供接口

对于数据层的接口的要求:

1. 既然是进行数据操作的,那么就将其保存在dao包下
2. 既然是不同数据表的操作有可能使用不同数据层开发,那么就针对数据表命名

|-emp表,那么数据层的接口定义为IEmpDAO;

1. 对于整个数据层的开发严格来讲就只有两类功能

|-数据更新:建议它的操作方法以doXxx()方式命名,例:doCreate(),doUpdate(),doRemove()

|-数据查询:分为两种形式,查询表中的数据:以findXxx()形式,findById(),以及统计表中的数据，以getXxx()形式命名, 例:getAllCount()

范例:定义IEmpDAO接口

|  |
| --- |
| package cn.mldn.dao;  import cn.mldn.vo.Emp;  import java.util.List;  import java.util.Set;  /\*\*  \* 定义emp表的数据层操作标准(接口)  \*/  public interface IEmpDAO{  /\*\*  \* 实现数据的增加操作  \* @param vo 包含了要增加数据的VO对象  \* @return 数据保存成功返回true,否则返回false  \* @throws Exception SQL执行异常  \*/  public boolean doCreate(Emp vo) throws Exception;  /\*\*  \* 实现数据库的修改操作,本次修改的是根据id进行全部字段数据的修改  \* @param vo 包含了要修改数据的信息,一定要提供有ID内容  \* @return 数据修改成功返回true,否则返回false  \* @throws Exception SQL执行异常  \*/  public boolean doUpdate(Emp vo) throws Exception;  /\*\*  \* 执行数据的批量删除操作,所有要删除的数据以Set集合保存  \* @param ids 包含了要删除的ID,不包含重复内容  \* @return 删除成功返回true(删除的数据个数要与删除的数据个数相同),否则返回false  \* @throws Exception SQL执行异常  \*/  public boolean doRemoveBatch(Set<Integer> ids) throws Exception;  /\*\*  \* 根据雇员编号查看指定的信息  \* @param id 要查询的雇员信息  \* @return 如果雇员信息存在,则数据以VO类对象的形式返回,如果雇员不存在返回null  \* @throws Exception SQL执行异常  \*/  public Emp findById(Integer id) throws Exception;  /\*\*  \* 查询指定数据的全部信息,并以集合的形式保存  \* @return 如果表中有数据,则所有的数据会封装为VO对象而后利用List集合返回  \* 如果没有数据,那么集合的长度为0,不是为null  \* @throws Exception SQL执行异常  \*/  public List<Emp> finaAll() throws Exception;  /\*\*  \* 分页进行数据的模糊查询,查询的结果以集合形式返回  \* @param currentPage 当前所在的页  \* @param lineSize 每页所在的数据行数  \* @param column 要进行模糊查询的数据列  \* @param keyWord 模糊查询的关键字  \* @return 如果表中有数据,则所有的数据会封装为VO对象而后利用List集合返回  \* 如果没有数据,那么集合的长度为0,不是为null  \* @throws Exception SQL执行异常  \*/  public List<Emp> findAllSplit(Integer currentPage,Integer lineSize,String column,String keyWord) throws Exception;  /\*\*  \* 进行模糊查询的统计,如果表中没有记录统计的结果为0  \* @param column 要进行模糊查询的数据列  \* @param keyWord 模糊查询的关键字  \* @return 返回表中的数据量,如果没有数据返回0  \* @throws Exception SQL执行异常  \*/  public Integer getAllcount(String column,String keyWord) throws Exception;  } |

### 3.2数据层的实现

保存在dao.impl子包下

范例:EmpDAOImpl子类

|  |
| --- |
| package cn.mldn.dao.impl;  import cn.mldn.dao.IEmpDAO;  import cn.mldn.vo.Emp;  import java.sql.Connection;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.util.ArrayList;  import java.util.Iterator;  import java.util.List;  import java.util.Set;  public class EmpDAOImpl implements IEmpDAO {  private Connection conn; //需要Connection的操作  private PreparedStatement pstmt;  /\*\*  \* 如果要进行数据层进行原子性的操作,必须要实现Connection接口对象  \* 另外,由于开发之中业务层要调用数据层,所以数据库的打开与关闭交由业务层  \* @param conn 表示数据库的连接对象  \*/  public EmpDAOImpl(Connection conn){  this.conn = conn;  }  @Override  public boolean doCreate(Emp vo) throws Exception {  String sql = "INSERT INTO emp(empno,ename,job,hiredate,sal,comm)VALUES(?,?,?,?,?,?)";  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setInt(1,vo.getEmpno());  this.pstmt.setString(2,vo.getEname());  this.pstmt.setString(3,vo.getJob());  this.pstmt.setDate(4,new java.sql.Date(vo.getHiredate().getTime()));  this.pstmt.setDouble(5,vo.getSal());  this.pstmt.setDouble(6,vo.getComm());  return this.pstmt.executeUpdate() > 0;  }  @Override  public boolean doUpdate(Emp vo) throws Exception {  String sql = "UPDATE emp SET ename=?,job=?,hiredate=?,sal=?,comm=? WHERE empno=?";  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setString(1,vo.getEname());  this.pstmt.setString(2,vo.getJob());  this.pstmt.setDate(3,new java.sql.Date(vo.getHiredate().getTime()));  this.pstmt.setDouble(4,vo.getSal());  this.pstmt.setDouble(5,vo.getComm());  this.pstmt.setInt(6,vo.getEmpno());  return this.pstmt.executeUpdate() > 0;  }  @Override  public boolean doRemoveBatch(Set<Integer> ids) throws Exception {  if(ids == null || ids.size()==0){//没有要删除的数据  return false;  }  StringBuffer sql = new StringBuffer();  sql.append("DELETE FROM emp WHERE empno IN(");  Iterator<Integer> iter = ids.iterator();  while(iter.hasNext()){  sql.append(iter.next()).append(",");  }  sql.delete(sql.length()-1,sql.length()).append(")");  this.pstmt = this.conn.prepareStatement(sql.toString());  return this.pstmt.executeUpdate() == ids.size();  }  @Override  public Emp findById(Integer id) throws Exception {  Emp vo = null;  String sql = "SELECT empno,ename,job,hiredate,sal,comm from where empno = ?";  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setInt(1,id);  ResultSet rs = this.pstmt.executeQuery();  if(rs.next()){  vo = new Emp();  vo.setEmpno(rs.getInt(1));  vo.setEname(rs.getString(2));  vo.setJob(rs.getString(3));  vo.setHiredate(rs.getDate(4));  vo.setSal(rs.getDouble(5));  vo.setComm(rs.getDouble(6));  }  return vo;  }  @Override  public List<Emp> finaAll() throws Exception {  List<Emp> all = new ArrayList<Emp>();  String sql = "SELECT empno,ename,job,hiredate,sal,comm from emp";  this.pstmt = this.conn.prepareStatement(sql);  ResultSet rs = this.pstmt.executeQuery();  while (rs.next()){  Emp vo = new Emp();  vo.setEmpno(rs.getInt(1));  vo.setEname(rs.getString(2));  vo.setJob(rs.getString(3));  vo.setHiredate(rs.getDate(4));  vo.setSal(rs.getDouble(5));  vo.setComm(rs.getDouble(6));  all.add(vo);  }  return all;  }  @Override  public List<Emp> findAllSplit(Integer currentPage, Integer lineSize, String column, String keyWord) throws Exception {  List<Emp> all = new ArrayList<Emp>();  String sql = "SELECT empno,ename,job,hiredate,sal,comm from emp WHERE ? LIKE ? LIMIT ?,?";  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setString(1,column);  this.pstmt.setString(2,"%" + keyWord + "%");  this.pstmt.setInt(3,currentPage);  this.pstmt.setInt(4,lineSize);  ResultSet rs = this.pstmt.executeQuery();  while (rs.next()){  Emp vo = new Emp();  vo.setEmpno(rs.getInt(1));  vo.setEname(rs.getString(2));  vo.setJob(rs.getString(3));  vo.setHiredate(rs.getDate(4));  vo.setSal(rs.getDouble(5));  vo.setComm(rs.getDouble(6));  all.add(vo);  }  return all;  }  @Override  public Integer getAllcount(String column, String keyWord) throws Exception {  String sql = "SELECT COUNT(empno) from emp where ? like ?";  this.pstmt = this.conn.prepareStatement(sql);  this.pstmt.setString(1,column);  this.pstmt.setString(2,"%" + keyWord + "%");  ResultSet rs = this.pstmt.executeQuery();  if(rs.next()){  return rs.getInt(1);  }  return null;  }  } |

3.3数据层工厂类—DAOFactory

保存在factory子包下

范例:定义工厂类

|  |
| --- |
| package cn.mldn.factory;  import cn.mldn.dao.IEmpDAO;  import cn.mldn.dao.impl.EmpDAOImpl;  import java.sql.Connection;  public class DAOFactory{  public static IEmpDAO getIEmpDAOInstance(Connection conn){  return new EmpDAOImpl(conn);  }  } |

外层不需要具体的子类

## 业务层

业务层是真正留给外部调用的,可能是控制层,或者是直接调用，

业务层也是有不同层进行调用的,所以也要定义业务层的标准

|  |
| --- |
| Emp |
|  |
|  |

|  |
| --- |
| IEmpService  <<interface>> |
|  |
| +insert(Emp vo):boolean  +update(Emp vo):boolean  +delete(Set<Integer ids):boolean  +get(int id):Emp  +list():List<Emp>  +List(...):Map<String,Object> |

|  |
| --- |
| IEmpServiceImpl |
|  |
| Dbc:DatabaseConnection |

|  |
| --- |
| ServiceFactory |
|  |
| +getIEmpServiceInstance():IEmpService |

控制层

### 4.1开发业务层标准IEmpService

业务层也可以称为Service层,保存在service子包下,但是对业务层的方法

定义没有明确的要求，只不过还是写上有意义的统一名称

范例:IempService

|  |
| --- |
| package cn.mldn.service;  import cn.mldn.vo.Emp;  import java.util.List;  import java.util.Map;  import java.util.Set;  /\*\*  \* 定义emp表的业务层的执行标准,此类还要负责数据库的打开和关闭操作  \* 此类可以用过DAOFactory取得IEmpDAO对象  \*/  public interface IEmpService{  /\*\*  \* 实现雇员数据的增加,本次操作要调用IEmpDAO接口的如下方法:  \* IEmpDAO.findById()方法,判断要增加的id是否已经存在  \* 如果要增加的数据编号不存在,则调用IEmpDAO.doCreate()方法返回一个操作结果  \* @param vo  \* @return 如果增加对象的ID存在或者保存失败,返回false,否则返回true  \* @throws Exception  \*/  public boolean insert(Emp vo) throws Exception;  /\*\*  \* 实现雇员的修改操作,本次要调用IEmpDAO.doUpdate()方法,本次修改数据全部内容的修改  \* @param vo  \* @return 修改成功返回true,否则返回false  \* @throws Exception  \*/  public boolean update(Emp vo) throws Exception;  /\*\*  \* 执行雇员数据的删除操作,可以删除多个雇员信息,调用IEmpDAO.doRemoveBatch()方法  \* @param ids  \* @return  \* @throws Exception  \*/  public boolean delete(Set<Integer> ids) throws Exception;  /\*\*  \* 根据雇员编号查找雇员的完整信息,调用IEmpDAO.findById()方法  \* @param ids  \* @return  \* @throws Exception  \*/  public Emp get(Integer ids) throws Exception;  /\*\*  \* 查询全部信息,调用IEmpDAO.findAll()方法  \* @return 查询结果以List集合的形式返回,如果没有数据则集合长度为0  \* @throws Exception  \*/  public List<Emp> list()throws Exception;  /\*\*  \* 实现数据的模糊查询与统计,要调用IEmpDAO接口的两个方法  \* 1)调用IEmpDAO.findAllSplit()方法,查询出所有的表数据,返回List<Emp>  \* 2)调用IEmpDAO.getAllCount()方法,查询所有的数据量,返回Integer  \* @param currentPage 当前所在页  \* @param lineSize 记录条数  \* @param column 模糊查询的数据列  \* @param keyWord 模糊查询的关键字  \* @return 由于需要返回多种需要类型,使用了Map集合返回,由于包含的类型不统一,  \* 所以value的类型设置为Object,返回类型如下:  \* key=allEmps,value=IEmpDAO.findAllSplit()返回结果  \* key=empCount,value=IEmpDAO.getALlCount()返回结果  \* @throws Exception  \*/  public Map<String,Object> list(int currentPage,int lineSize,String column,String keyWord) throws Exception;  } |

### 4.2业务层实现类

保存在dao.impl子包中

核心功能:

1. 负责控制数据库的打开和关闭,当存在了业务层对象后就是为了操作数据库,即业务层对象实例化后就必须准备好数据库连接
2. 根据DAOFactory调用getIEmpDAOInstance()方法而后取得IEmpDAO接口对象

|  |
| --- |
| package cn.mldn.service.impl;  import cn.mldn.dbc.DatabaseConnection;  import cn.mldn.factory.DAOFactory;  import cn.mldn.service.IEmpService;  import cn.mldn.vo.Emp;  import java.util.HashMap;  import java.util.List;  import java.util.Map;  import java.util.Set;  public class EmpServiceImpl implements IEmpService{  //在这个类的对象就提供有一个数据库连接类的对象  private DatabaseConnection dbc = new DatabaseConnection();  @Override  public boolean insert(Emp vo) throws Exception {  try{  //如果雇员ID不存在,则findById返回null,null表示可以进行新雇员的保存  if(DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).findById(vo.getEmpno())==null){  return DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).doCreate(vo);  }else{  return false;  }  }catch (Exception e){  throw e;  }finally {  this.dbc.close();  }  }  @Override  public boolean update(Emp vo) throws Exception {  try{  return DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).doUpdate(vo);  }catch (Exception e){  throw e;  }finally {  this.dbc.close();  }  }  @Override  public boolean delete(Set<Integer> ids) throws Exception {  try{  return DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).doRemoveBatch(ids);  }catch (Exception e){  throw e;  }finally {  this.dbc.close();  }  }  @Override  public Emp get(Integer ids) throws Exception {  try{  return DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).findById(ids);  }catch (Exception e){  throw e;  }finally {  this.dbc.close();  }  }  @Override  public List<Emp> list() throws Exception {  try{  return DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).finaAll();  }catch (Exception e){  throw e;  }finally {  this.dbc.close();  }  }  @Override  public Map<String, Object> list(int currentPage, int lineSize, String column, String keyWord) throws Exception {  try{  Map<String,Object> map = new HashMap<String,Object>();  map.put("allEmps",DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).findAllSplit(currentPage,lineSize,column,keyWord));  map.put("empCount",DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).getAllcount(column,keyWord));  return map;  }catch (Exception e){  throw e;  }finally {  this.dbc.close();  }  }  } |

不同层之间的访问就是依靠工厂类和接口进行操作

### 4.3业务层工厂类-ServiceFactory

业务层需要被其他的层所使用,所以需要为其定义工厂类,保存在factory子包下

从实际的开发来将,业务层应该分为两种:

1. 前台业务逻辑:可以将其保存在service.front包中,工厂类:ServiceFrontFactory
2. 后台业务逻辑:可以将其保存在service.back包中,工厂类:ServiceBackFactory

范例:定义ServiceFactory

|  |
| --- |
| package cn.mldn.factory;  import cn.mldn.service.IEmpService;  import cn.mldn.service.impl.EmpServiceImpl;  public class ServiceFactory{  public static IEmpService getIEmpServiceInstance(){  return new EmpServiceImpl();  }  } |

## 代码测试

因为最终的业务层是需要由用户去调用的,所以测试分为两种

调用测试:按照传统方式,而后调用里面的方法

### 5.1调用测试

保存在test子包中

范例:测试增加操作

|  |
| --- |
|  |