第 5 章 DAO 层组件的封装设计

DAO（Data Access Object）是数据访问对象。DAO层是业务逻辑层与数据库层之间的中间层，它进一步封装了数据库的底层操作，屏蔽了具体数据存取技术的实现细节。上层业务逻辑代码通过调用DAO层组件来实现与数据库的交互。

DAO层在设计时通常包含两个部分：DAO接口和DAO实现类。开发人员的业务代码面向DAO接口，当需要更新接口的具体实现时，可以定义一个新的实现类，现有的业务代码不会受到任何影响，便于系统的扩展。

本章介绍DAO层组件的封装设计与具体实现。

视频讲解

5.1 基于泛型的通用 DAO 接口设计

在实际的系统业务中，有很多业务逻辑都是通过对数据库表的增、删、改、查等操作来实现的，例如考生注册新用户时需要在数据库表student中插入一条记录，管理员登录系统时需要查询数据库表admin中的记录，考生用户修改报考信息时需要更新数据库表enroll中的记录。DAO接口作为中间层，要能够提供一些适用于操作不同数据库表的通用方法，从而简化上层业务逻辑代码的编写。基于泛型技术来设计DAO接口无疑是一个很好的选择。下面我们将基于泛型技术设计一个通用的DAO接口，并在这个通用DAO接口中提供比较全面的用于操作数据库的基本方法。

首先在src目录下创建java包“cn.bmxt.dao.common”，然后在该包中创建一个接口BaseDao，BaseDao的设计代码如5.1-01所示。

【代码5.1-01】文件/src/cn.bmxt.dao.common/BaseDao.java，版本0.01。

```
01 package cn.bmxt.dao.common;
02 import cn.bmxt.entity.common.Entity;
03 import java.util.List;
04 import java.util.Map;
05 public interface BaseDao<T extends Entity> {
06   public T findOneById(long id);
07   public T findOneByField(String fieldName, Object fieldValue);
08   public T findOneBySql(String sql, Object... sqlParas);
09   public Map<String, Object> findMapBySql(String sql, Object...
```

```
   sqlParas);
10   public List<T> findAll();
11   public List<T> findAllByField(String fieldName, Object fieldValue);
12   public List<T> findAllBySql(String sql, Object... sqlParas);
13   public List<Map<String, Object>> findMapListBySql(String sql,
   Object... sqlParams);
14   public int save(T entity);
15   public int save(Map<String, Object> entityFieldMap);
16   public int deleteById(long id);
17   public int deleteByField(String fieldName, Object fieldValue);
18   public int deleteBySql(String sql, Object... sqlParas);
19   public int updateById(long id, Map<String, Object>
   entityFieldMap);
20   public int updateByField(String fieldName, Object fieldValue);
21   public int updateBySql(String sql, Object... sqlParas);
22   public long count();
23   public long count(String sql, Object... sqlParas);
24 }
```

代码5.1-01说明如下：

（1）第05行，BaseDao接口的声明行，其中定义的泛型T代表的是Entity类的子类型，Entity类的子类型就是4.2节中定义的10个实体类。

（2）第06行~第08行，定义了3个常用的查询实体的方法。第06行中定义的findOneById方法通过给定的id值查询得到一个实体。第07行中定义的findOneByField方法通过给定的字段名称和字段值查询得到一个实体。第08行中定义的findOneBySql方法通过给定一个预编译SQL语句及其参数数组查询得到一个实体。

（3）第09行，定义了findMapBySql方法，通过给定一个预编译SQL语句及其参数数组查询一条记录，将结果记录转换为Map类型的数据后返回。

（4）第10行~第12行，定义了3个常用的查询实体集合的方法。第10行中定义的findAll方法用于查询一个表中的所有记录，并封装为与之对应的实体集合。第11行中定义的findAllByField方法通过给定的字段名称和字段值查询得到一个实体列表集合。第12行中定义的findAllBySql方法则通过给定一个预编译SQL语句及其参数数组查询得到一个实体列表集合。

（5）第13行，定义了findMapListBySql方法。通过给定一个预编译SQL语句及其参数数组查询多条记录，将其中的记录转换为Map类型的数据，然后再将这些Map类型的数据组织在一个列表集合中返回。

（6）第14行~第15行，定义了重载的两个save方法，用于在数据库表中保存一条记录，方法入参可以是一个实体类，也可以是一个Map类型的数据。

（7）第16行~第18行，定义了3个常用的删除记录的方法。第16行中定义的deleteById方法删除某个id值对应的记录。第17行中定义的deleteByField方法用于删除某个字段（fieldName）等于某个值（fieldValue）的所有记录。第18行中定义的deleteBySql方法则通过给定一个预编译SQL语句及其参数数组来执行删除操作。

（8）第19行~第21行，定义了3个常用的更新数据的方法。第19行中定义的updateById方

法用于更新某个id值对应的记录，记录中需要更新的字段和值则通过一个Map类型的数据提供。第20行中定义的updateByField方法用于对一个字段（fieldName）的值进行修改，将其修改为参数中给定的值（fieldValue）。第21行中定义的updateBySql方法则通过给定一个预编译SQL语句及其参数数组来执行更新操作。

（9）第22行~第23行，定义了重载的两个count方法，用于获取查询结果中的记录条数，其中，第22行中定义的无参的count方法获取的是某个表中记录的总条数，第23行中定义的count方法获取的是某个查询语句执行后的结果记录条数。


视频讲解

5.2 基于泛型的通用DAO接口实现类设计

在5.1节中，我们创建了一个基于泛型的通用DAO接口，在其中定义了一些常用的操作数据库的方法，接着就需要创建其实现类，在实现类中实现接口中定义的方法。

首先在src目录下创建java包“cn.bmxt.dao.common.impl”，然后在该包中创建一个类BaseDaoImpl，并实现接口BaseDao。实现类BaseDaoImpl的设计代码如5.2-01所示。

【代码5.2-01】文件/src/cn.bmxt.dao.common.impl/BaseDaoImpl.java，版本0.01。

```
01 package cn.bmxt.dao.common.impl;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.common.Entity;
04 import cn.bmxt.util.DbUtil;
05 import java.lang.reflect.ParameterizedType;
06 import java.util.List;
07 import java.util.Map;
08 public class BaseDaoImpl<T extends Entity> implements BaseDao<T> {
09   protected Class<T> clazz;
10   {
11     ParameterizedType parameterizedType = (ParameterizedType) this.
   getClass().getGenericSuperclass();
12     clazz = (Class<T>) parameterizedType.getActualTypeArguments()
   [0];
13   }
14   public T findOneById(long id) {
15     return DbUtil.queryEntity(clazz, "select * from " + DbUtil.
   getTableName(clazz) + " where id=?", id);
16   }
17   public T findOneByField(String fieldName, Object fieldValue) {
18     return DbUtil.queryEntity(clazz, "select * from " + DbUtil.
   getTableName(clazz) + " where " + fieldName + "=?", fieldValue);
19   }
20   public T findOneBySql(String sql, Object... sqlParas) {
21     return DbUtil.queryEntity(clazz, sql, sqlParas);
22   }
```

```
23   public List<T> findAll() {
24     return DbUtil.queryEntityList(clazz, "select * from " + DbUtil.
   getTableName(clazz));
25   }
26   public List<T> findAllByField(String fieldName, Object fieldValue) {
27     return DbUtil.queryEntityList(clazz, "select * from " + DbUtil.
   getTableName(clazz) + " where " + fieldName + "=?", fieldValue);
28   }
29   public List<T> findAllBySql(String sql, Object... sqlParas) {
30     return DbUtil.queryEntityList(clazz, sql, sqlParas);
31   }
32   public Map<String, Object> findMapBySql(String sql, Object...
   sqlParas) {
33     return DbUtil.queryMap(sql, sqlParas);
34   }
35   public List<Map<String, Object>> findMapListBySql(String sql,
   Object... sqlParams) {
36     return DbUtil.queryMapList(sql, sqlParams);
37   }
38   public int save(T entity) {
39     return DbUtil.insertEntity(entity);
40   }
41   public int save(Map<String, Object> entityFieldMap) {
42     return DbUtil.insertEntity(clazz, entityFieldMap);
43   }
44   public int deleteById(long id) {
45     return DbUtil.deleteEntity(clazz, id);
46   }
47   public int deleteByField(String fieldName, Object fieldValue) {
48     return DbUtil.updateBySql("delete from " + DbUtil.
   getTableName(clazz) + " where " + fieldName + "=?", fieldValue);
49   }
50   public int deleteBySql(String sql, Object... sqlParas) {
51     return DbUtil.updateBySql(sql, sqlParas);
52   }
53   public int updateById(long id, Map<String, Object>
   entityFieldMap) {
54     return DbUtil.updateEntity(clazz, id, entityFieldMap);
55   }
56   public int updateByField(String fieldName, Object fieldValue) {
57     return DbUtil.updateBySql("update " + DbUtil.
   getTableName(clazz) + " set " + fieldName + "=?", fieldValue);
58   }
59   public int updateBySql(String sql, Object... sqlParas) {
60     return DbUtil.updateBySql(sql, sqlParas);
```

```
61    }
62    public long count() {
63      return DbUtil.count(clazz);
64    }
65    public long count(String sql, Object... sqlParas) {
66      return DbUtil.count(sql, sqlParas);
67    }
68 }
```

代码5.2-01说明如下：

（1）第08行，BaseDaoImpl类的声明行，其中定义的泛型T代表Entity类的子类型，Entity类的子类型就是4.2节中定义的10个实体类。BaseDaoImpl类实现了BaseDao接口，因此需要在BaseDaoImpl类中实现BaseDao接口中定义的所有方法。

（2）第09行，声明泛型参数的类型变量，用于保存BaseDaoImpl类的声明行中的泛型参数T所代表的实际类型。

（3）第10行~13行，当创建BaseDaoImpl类或者其子类的实例对象时，花括号中的代码块会被执行，其作用是获取被参数化了的泛型T所代表的实际类型。通常不会直接去创建BaseDaoImpl类的实例对象，而是创建其子类的实例对象来应用。子类在声明时会指定它所继承的父类“BaseDaoImpl<T extends Entity>”中的参数T具体代表的是哪个实体类。因此，在构造BaseDaoImpl类的子类的实例对象时，会执行第11行和第12行代码，此时第11行中的this关键字代表的是要构造的子类实例对象，“this.getClass()”获取的就是子类的类型信息；然后调用“getGenericSuperclass()”方法就可获取其带泛型的父类，而这个父类正是现在定义的“BaseDaoImpl<T extends Entity>”，它是具有参数化类型的类，可以转换为ParameterizedType类型。在第12行中调用ParameterizedType的getActualTypeArguments方法就可以获取实际的参数化类型数组，从这个数组中取出第一个（下标为0）参数化类型，就是T代表的实际类型，因为T正是BaseDaoImpl类定义时的第一个参数化类型。最终能够获取T代表的实际类型，它就是4.2节中定义的10个实体类中的某一个。知道了泛型参数T表示的是哪个实体类，就可以通过调用DbUtil工具类中封装好的通用的操作数据库的泛型方法，来实现BaseDao接口中定义的方法了。

（4）第14行~第67行，调用DbUtil工具类中的相关方法，逐一实现BaseDao接口中定义的所有方法。

5.3 对分页数据的封装处理

一般来说，当查询多条记录时可能出现较多的结果条数，此时成百上千条数据在一个页面中显示可能会过于冗长，不便于用户阅读。在这种情况下，通常的做法就是使用分页的形式显示数据，在每个数据页中仅显示少量的数据，然后给用户提供分页导航栏用于数据页的切换。在数据分页导航栏中，用户可以自行设定每页显示的数据条目数，也可以上下进行翻页，或者直接指定要跳转的数据页。实现数据分页导航首先需要设计一个分页模型对象，还需要封装按页次查询数据的方法。

1. 分页模型对象设计


视频讲解

分页模型对象是用于临时存储数据的一种特殊对象。在分页模型对象中，既要存储查询到的当前页的数据集合，也要保存分页的相关信息。数据集合可以使用列表进行存储，列表中存储的具体对象类型并非是确定的，查询考生列表时存储的是考生对象，查询报名信息列表时存储的是报名信息对象，还有一些查询结果得到的是Map类型数据，因此在设计分页模型对象时要使用泛型来描述不同类型的数据集合。分页的相关信息用于描述数据分页情况，包括当前页次、每页显示的条目数、数据的总条数、数据的总页数等。

在“cn.bmxt.entity.common”包中创建分页模型类Page，其设计代码如5.3-01所示。

【代码5.3-01】文件/src/cn.bmxt.entity.common/Page.java，版本1.0。

```
01 package cn.bmxt.entity.common;
02 import java.io.Serializable;
03 import java.util.ArrayList;
04 import java.util.List;
05 public class Page<T> implements Serializable {
06   private static final long serialVersionUID = 1L;
07   private int number = 1; // 当前页次
08   private int size = 10; // 每页显示的条目数
09   private long total = 0; // 数据的总条数
10   private int pages = 1; // 数据总页数
11   private int prev = 1; // 上一页
12   private int next = 1; // 下一页
13   private List<T> items = new ArrayList<T>(); // 存放当前页次的数据集合
14   public Page() { super(); }
15   public Page(int number, int size, long total) {
16     super();
17     this.number = number > 0 ? number : 1;
18     this.size = size > 0 ? size : 10;
19     this.total = total;
20     this.setPages();
21     this.setPrev();
22     this.setNext();
23   }
24   public int getNumber() { return number; }
25   public void setNumber(int number) { this.number = number; }
26   public int getSize() { return size; }
27   public void setSize(int size) { this.size = size; }
28   public long getTotal() { return total; }
29   public void setTotal(long total) { this.total = total; }
30   public int getPages() { return pages; }
31   public void setPages() {
32     if (total == 0) {
33       pages = 1;
```

```
34      } else {
35        pages = (int)((total % size == 0)? total/size : total/size +
    1);
36      }
37    }
38    public int getPrev() { return prev; }
39    public void setPrev() { prev = number > 1 ? number-1 : 1;}
40    public int getNext() { return next; }
41    public void setNext() { next = number < pages ? number+1 :
    pages;}
42    public List<T> getItems() { return items; }
43    public void setItems(List<T> items) { this.items = items; }
44  }
```

代码5.3-01说明如下：

（1）第07行，定义了私有成员变量number，用于存储当前页次，初始化为1。在第24行~第25行中定义了该成员的getter和setter方法。

（2）第08行，定义了私有成员变量size，用于存储每页显示的条目数，初始化为10。在第26行~第27行中定义了该成员的getter和setter方法。

（3）第09行，定义了私有成员变量total，用于存储数据的总条数，初始化为0。在第28行~第29行中定义了该成员的getter和setter方法。

（4）第10行，定义了私有成员变量pages，用于存储数据的总页数，初始化为1。在第30行~第37行中定义了该成员的getter和setter方法。在其setter方法中，pages的值是根据total（数据的总条数）和size（每页显示的条目数）计算出来的，如果数据的总条数total为0，就设置总页数pages为1；如果数据的总条数total不为0，则需要结合每页显示的条目数size进行简单计算，具体求解过程参见第35行代码。

（5）第11行，定义了私有成员变量prev，用于存储上一页的页次，初始化为1。在第38行~第39行中定义了该成员的getter和setter方法。在其setter方法中，prev的值是根据number（当前页次）计算出来的，如果number大于1，prev就等于“number-1”，否则prev就等于1。

（6）第12行，定义了私有成员变量next，用于存储下一页的页次，初始化为1。在第40行~第41行中定义了该成员的getter和setter方法。在其setter方法中，next的值是根据number（当前页次）和pages（数据的总页数）计算出来的，如果number小于pages，next就等于“number+1”，否则next就等于pages。

（7）第13行，定义了私有成员变量items，用于存储当前页的数据列表，其类型声明为基于泛型的列表集合“List<T>”，并初始化为“ArrayList<T>”类型。在第42行~第43行中定义了该成员的getter和setter方法。

2. 封装查询分页数据的方法


视频讲解

在BaseDao接口中新增3个用于查询分页数据的通用接口方法，更新后的BaseDao接口的完整代码如5.3-02所示。

【代码5.3-02】文件/src/cn.bmxt.dao.common/BaseDao.java，版本1.0，上一个版本参见代码5.1-01。

```
01 package cn.bmxt.dao.common;
02 import cn.bmxt.entity.common.Entity;
03 import cn.bmxt.entity.common.Page;
04 import java.util.List;
05 import java.util.Map;
06 public interface BaseDao<T extends Entity> {
07   ...此处为【代码5.1-01】中的第06行~第23行代码
08   public Page<T> findOnePageBySql(int number, int size, String sql,
   Object... sqlParas);
09   public Page<T> findOnePage(int number, int size);
10   public Page<Map<String, Object>> findOnePageMapListBySql(int
   number, int size, String sql, Object... sqlParas);
11 }
```

代码5.3-02中第08行~第10行为新增的3个查询分页数据的接口方法，方法中都需要传参number和size，其中number为分页的页次，size为每页显示的条目数。第08行中定义的findOnePageBySql方法通过给定一个预编译SQL语句及其参数数组查询得到一页实体列表集合，并将其封装在Page对象中返回。第09行中定义的findOnePage方法用于查询一个表中的某一页记录，并将得到的实体列表集合封装在Page对象中返回。第10行中定义的findOnePageMapListBySql通过给定一个预编译SQL语句及其参数数组查询得到一页Map类型数据的集合，并将其封装在Page对象中返回。

BaseDao接口中新增的3个用于查询分页数据的通用接口方法，需要在其实现类BaseDaoImpl中实现。实现这3个接口方法的BaseDaoImpl类的完整代码如5.3-03所示。

【代码5.3-03】文件/src/cn.bmxt.dao.common.impl/BaseDaoImpl.java，版本1.0，上一个版本参见代码5.2-01。

```
01 package cn.bmxt.dao.common.impl;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.common.Entity;
04 import cn.bmxt.util.DbUtil;
05 import cn.bmxt.entity.common.Page;
06 import org.apache.commons.lang3.ArrayUtils;
07 import java.lang.reflect.ParameterizedType;
08 import java.util.List;
09 import java.util.Map;
10 public class BaseDaoImpl<T extends Entity> implements BaseDao<T> {
11   ...此处为【代码5.2-01】中的第09行~第67行代码
12   public Page<T> findOnePageBySql(int number, int size, String sql,
   Object... sqlParas) {
13     long total = count(sql, sqlParas);
14     Page<T> page = new Page<>(number, size, total);
15     if(total > 0){
```

```
16        String pageSql = sql + " limit ?,? ";
17        Object[] sqlParasLimit = { (number - 1) * size, size };
18        Object[] sqlParasAll = ArrayUtils.addAll(sqlParas,
    sqlParasLimit);
19        page.setItems(findAllBySql(pageSql, sqlParasAll));
20      }
21      return page;
22    }
23    public Page<T> findOnePage(int number, int size) {
24      return findOnePageBySql(number, size, "select * from " + DbUtil.
    getTableName(clazz));
25    }
26    public Page<Map<String, Object>> findOnePageMapListBySql(int
    number, int size, String sql, Object... sqlParas) {
27      long total = count(sql, sqlParas);
28      Page<Map<String, Object>> page = new Page<>(number, size,
    total);
29      if(total > 0){
30        String pageSql = sql + " limit ?,? ";
31        Object[] sqlParasLimit = { (number - 1) * size, size };
32        Object[] sqlParasAll = ArrayUtils.addAll(sqlParas,
    sqlParasLimit);
33        page.setItems(findMapListBySql(pageSql, sqlParasAll));
34      }
35      return page;
36    }
37 }
```

代码5.3-03中新增的3个实现方法分别说明如下：

（1）第12行~第22行，实现了findOnePageBySql方法。第13行首先调用count方法获取给定查询语句的结果记录总条数total。第14行根据用户传入的number（页次）、size（每页显示的条目数）以及第13行查询到的记录总条数total构造一个Page对象。第15行判断total是否大于0，如果total大于0，继续执行分页数据的查询。第16行使用limit关键字构建用于分页查询的预编译SQL语句。第17行根据number和size构建limit子句的参数。第18行将用户提供的预编译SQL语句的参数与limit子句的参数进行合并。第19行调用findAllBySql方法，传入用于分页查询的预编译SQL语句以及在第18行中合并后的语句参数，最后将查询得到的实体列表集合封装到page对象的items中。第21行返回page对象。

（2）第23行~第25行，实现了findOnePage方法。查询的是一个表中的某一页记录，因此其查询语句是固定的，通过调用findOnePageBySql方法传入这个固定的查询语句即可实现方法功能。

（3）第26行~第36行，实现了findOnePageMapListBySql方法。与findOnePageBySql方法的不同之处在于，该方法查询得到的是Map类型的数据集合。除此之外，该方法实现时的操作步骤与findOnePageBySql方法的实现步骤基本一致，其中第33行在查询分页数据时调用的方法

需要替换为findMapListBySql方法。

5.4 实体类 DAO 接口及其实现类设计


视频讲解

实体类DAO接口需要继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型。实体类DAO接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，同时实现实体类DAO接口。通过上述继承和实现机制，实体类DAO组件就能够对通用DAO接口中定义的方法进行复用。下面分别对各个实体类DAO接口及其实现类进行设计。

1. 创建SiteDao接口及其实现类

SiteDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Site。在包“cn.bmxt.dao”中创建SiteDao接口，接口的代码如5.4-01所示。

【代码5.4-01】文件/src/cn.bmxt.dao/SiteDao.java，版本1.0。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Site;
04 public interface SiteDao extends BaseDao<Site> {
05 }
```

SiteDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Site，同时实现SiteDao接口。

首先在src目录下创建包“cn.bmxt.dao.impl”，用于组织所有的实体类Dao接口的实现类。接着在包“cn.bmxt.dao.impl”中创建SiteDao接口的实现类SiteDaoImpl，其实现代码如5.4-02所示。

【代码5.4-02】文件/src/cn.bmxt.dao.impl/SiteDaoImpl.java，版本1.0。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.SiteDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Site;
05 public class SiteDaoImpl extends BaseDaoImpl<Site> implements
   SiteDao {
06 }
```

我们看到，通过上述的继承和实现机制，在SiteDao接口中未定义任何新的接口方法，就可以复用那些在通用DAO接口BaseDao中定义的接口方法。同样地，在SiteDao接口的实现类SiteDaoImpl中也不需要实现新的接口方法，就可以复用那些在通用DAO接口实现类BaseDaoImpl中已经实现的接口方法。我们可以编写代码进行测试，在“cn.bmxt.util”包下的JunitTest类中新增一个单元测试方法testSiteDaoImpl，方法代码如5.4-03所示。

【代码5.4-03】测试SiteDaoImpl类继承的方法，代码位置：/src/cn.bmxt.util/JunitTest.java。

```
01 @Test
02 public void testSiteDaoImpl(){
03   SiteDao siteDao = new SiteDaoImpl();
04   System.out.println(siteDao.findOneById(1));
05   System.out.println(siteDao.findMapBySql("select site_school, site_
   name from site where id=?", 1));
06 }
```

代码5.4-03中测试了SiteDaoImpl继承的两个方法，分别是findOneById方法和findMapBySql方法。该单元测试方法执行后的结果如下所示，两个方法都能够成功地查询出site表中id值为1的记录。

```
01 Site(site_school=XXX 大学 , site_name=2023 年高职升本报名系统 ,
   site_location=XXX 市 XXX 路 XXX 号 , site_zip_code=300000, site_
   contact=022-88888888, site_copy= 版权所有 © 2023-2024 津 ICP 备 XXXX 号 )
02 {site_name=2023 年高职升本报名系统 , site_school=XXX 大学 }
```

一般来说，如果在系统上层的某个功能中不涉及特殊的业务需求，通用DAO接口及其实现类中封装的数据操作方法就能够满足需要了。当然，对于系统上层中的一些特殊业务需求，仅通过调用通用DAO接口及其实现类中封装的方法可能无法实现其功能，此时我们可以在相关的实体类DAO接口中定义新的接口方法，然后在实体类DAO接口的实现类中去实现即可。

2. 创建AdminDao接口及其实现类

AdminDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Admin。在包“cn.bmxt.dao”中创建AdminDao接口，接口的代码如5.4-04所示。

【代码5.4-04】文件/src/cn.bmxt.dao/AdminDao.java，版本0.01。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Admin;
04 public interface AdminDao extends BaseDao<Admin> {
05 }
```

AdminDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Admin，同时实现AdminDao接口。在包“cn.bmxt.dao.impl”中创建AdminDao接口的实现类AdminDaoImpl，其实现代码如5.4-05所示。

【代码5.4-05】文件/src/cn.bmxt.dao.impl/AdminDaoImpl.java，版本0.01。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.AdminDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Admin;
```

```
05 public class AdminDaoImpl extends BaseDaoImpl<Admin> implements
   AdminDao {
06 }
```

3. 创建PhaseDao接口及其实现类

PhaseDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Phase。在包“cn.bmxt.dao”中创建PhaseDao接口，接口的代码如5.4-06所示。

【代码5.4-06】文件/src/cn.bmxt.dao/PhaseDao.java，版本0.01。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Admin;
04 public interface PhaseDao extends BaseDao<Phase> {
05 }
```

PhaseDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Phase，同时实现PhaseDao接口。在包“cn.bmxt.dao.impl”中创建PhaseDao接口的实现类PhaseDaoImpl，其实现代码如5.4-07所示。

【代码5.4-07】文件/src/cn.bmxt.dao.impl/PhaseDaoImpl.java，版本0.01。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.PhaseDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Phase;
05 public class PhaseDaoImpl extends BaseDaoImpl<Phase> implements
   PhaseDao {
06 }
```

4. 创建MajorDao接口及其实现类

MajorDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Major。在包“cn.bmxt.dao”中创建MajorDao接口，接口的代码如5.4-08所示。

【代码5.4-08】文件/src/cn.bmxt.dao/MajorDao.java，版本1.0。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Major;
04 public interface MajorDao extends BaseDao<Major> {
05 }
```

MajorDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Major，同时实现MajorDao接口。在包“cn.bmxt.dao.impl”中创建MajorDao接口的实现类MajorDaoImpl，其实现代码如5.4-09所示。

【代码5.4-09】文件/src/cn.bmxt.dao.impl/MajorDaoImpl.java，版本1.0。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.MajorDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Major;
05 public class MajorDaoImpl extends BaseDaoImpl<Major> implements
   MajorDao {
06 }
```

5. 创建CourseDao接口及其实现类

CourseDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Course。在包“cn.bmxt.dao”中创建CourseDao接口，接口的代码如5.4-10所示。

【代码5.4-10】文件/src/cn.bmxt.dao/CourseDao.java，版本1.0。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Course;
04 public interface CourseDao extends BaseDao<Course> {
05 }
```

CourseDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Course，同时实现CourseDao接口。在包“cn.bmxt.dao.impl”中创建CourseDao接口的实现类CourseDaoImpl，其实现代码如5.4-11所示。

【代码5.4-11】文件/src/cn.bmxt.dao.impl/CourseDaoImpl.java，版本1.0。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.CourseDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Course;
05 public class CourseDaoImpl extends BaseDaoImpl<Course> implements
   CourseDao {
06 }
```

6. 创建StudentDao接口及其实现类

StudentDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Student。在包“cn.bmxt.dao”中创建StudentDao接口，接口的代码如5.4-12所示。

【代码5.4-12】文件/src/cn.bmxt.dao/StudentDao.java，版本0.01。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Student;
```

```
04 public interface StudentDao extends BaseDao<Student> {
05 }
```

StudentDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Student，同时实现StudentDao接口。在包“cn.bmxt.dao.impl”中创建StudentDao接口的实现类StudentDaoImpl，其实现代码如5.4-13所示。

【代码5.4-13】文件/src/cn.bmxt.dao.impl/StudentDaoImpl.java，版本0.01。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.StudentDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Student;
05 public class StudentDaoImpl extends BaseDaoImpl<Student> implements
   StudentDao {
06 }
```

7. 创建EnrollDao接口及其实现类

EnrollDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Enroll。在包“cn.bmxt.dao”中创建EnrollDao接口，接口的代码如5.4-14所示。

【代码5.4-14】文件/src/cn.bmxt.dao/EnrollDao.java，版本0.01。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Enroll;
04 public interface EnrollDao extends BaseDao<Enroll> {
05 }
```

EnrollDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Enroll，同时实现EnrollDao接口。在包“cn.bmxt.dao.impl”中创建EnrollDao接口的实现类EnrollDaoImpl，其实现代码如5.4-15所示。

【代码5.4-15】文件/src/cn.bmxt.dao.impl/EnrollDaoImpl.java，版本0.01。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.EnrollDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Enroll;
05 public class EnrollDaoImpl extends BaseDaoImpl<Enroll> implements
   EnrollDao {
06 }
```

8. 创建GradeDao接口及其实现类

GradeDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类

型Grade。在包“cn.bmxt.dao”中创建GradeDao接口，接口的代码如5.4-16所示。

【代码5.4-16】文件/src/cn.bmxt.dao/GradeDao.java，版本0.01。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Grade;
04 public interface GradeDao extends BaseDao<Grade> {
05 }
```

GradeDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Grade，同时实现GradeDao接口。在包“cn.bmxt.dao.impl”中创建GradeDao接口的实现类GradeDaoImpl，其实现代码如5.4-17所示。

【代码5.4-17】文件/src/cn.bmxt.dao.impl/GradeDaoImpl.java，版本0.01。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.GradeDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Grade;
05 public class GradeDaoImpl extends BaseDaoImpl<Grade> implements
   GradeDao {
06 }
```

9. 创建LogDao接口及其实现类

LogDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Log。在包“cn.bmxt.dao”中创建LogDao接口，接口的代码如5.4-18所示。

【代码5.4-18】文件/src/cn.bmxt.dao/LogDao.java，版本0.01。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Log;
04 public interface LogDao extends BaseDao<Log> {
05 }
```

LogDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Log，同时实现LogDao接口。在包“cn.bmxt.dao.impl”中创建LogDao接口的实现类LogDaoImpl，其实现代码如5.4-19所示。

【代码5.4-19】文件/src/cn.bmxt.dao.impl/LogDaoImpl.java，版本0.01。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.LogDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Log;
```

```
05 public class LogDaoImpl extends BaseDaoImpl<Log> implements LogDao {
06 }
```

10. 创建DocDao接口及其实现类

DocDao接口继承通用的BaseDao接口，并提供BaseDao接口中泛型参数T代表的实际类型Doc。在包“cn.bmxt.dao”中创建DocDao接口，接口的代码如5.4-20所示。

【代码5.4-20】文件/src/cn.bmxt.dao/DocDao.java，版本1.0。

```
01 package cn.bmxt.dao;
02 import cn.bmxt.dao.common.BaseDao;
03 import cn.bmxt.entity.Doc;
04 public interface DocDao extends BaseDao<Doc> {
05 }
```

DocDao接口的实现类需要继承BaseDao接口的实现类BaseDaoImpl，提供BaseDaoImpl类中的泛型参数T代表的实际类型Doc，同时实现DocDao接口。在包“cn.bmxt.dao.impl”中创建DocDao接口的实现类DocDaoImpl，其实现代码如5.4-21所示。

【代码5.4-21】文件/src/cn.bmxt.dao.impl/DocDaoImpl.java，版本1.0。

```
01 package cn.bmxt.dao.impl;
02 import cn.bmxt.dao.DocDao;
03 import cn.bmxt.dao.common.impl.BaseDaoImpl;
04 import cn.bmxt.entity.Doc;
05 public class DocDaoImpl extends BaseDaoImpl<Doc> implements DocDao {
06 }
```

5.5 DAO工厂类设计

视频讲解

在Java项目开发时，常常会涉及某个类型下的不同子类型的对象创建，如果在应用程序中直接使用new运算符创建这些对象，将不便于代码的修改与动态扩展。可以采用工厂方法模式来优化代码设计，通过将创建某个类的子类对象所需的类型信息存放在XML、properties等配置文件中，在程序中由某个被称为工厂的类根据相关配置信息来创建并管理对象。这种模式可以降低程序之间的耦合，当系统中需要增加新的子类进行动态扩展时，不必修改已有的程序代码，只需要修改配置文件即可。

本系统在设计开发时，上层业务逻辑是面向DAO层组件进行开发，也就是面向实体类DAO接口进行编程。实际上，程序中的实体类DAO接口引用的是其实现类对象，可以将实体类DAO接口的实现类对象的创建工作交由一个DAO工厂类来统一完成。创建这些对象所需的信息可以配置在一个properties属性文件中，由用户提供需要使用的实体类DAO接口名称，DAO工厂类就能够根据配置文件中的信息对应地创建出所需的DAO接口的实现类对象。因

此，在属性文件中需要配置实体类DAO接口名称与实体类DAO接口的实现类之间的对应关系，其中实体类DAO接口的实现类可以使用类的全限定名称表示，便于DAO工厂类使用Java反射来创建实例对象。

1. 创建属性配置文件

首先创建一个属性配置文件，在src目录上右击，在弹出的菜单中选择“New”→“File”，然后将新创建的File命名为“dao.properties”，接着编辑这个属性文件，在其中配置实体类DAO接口名称与实体类DAO接口的实现类之间的对应关系，配置代码如5.5-01所示。

【代码5.5-01】文件/src/dao.properties，版本1.0。

```
01 SiteDao=cn.bmxt.dao.impl.SiteDaoImpl
02 AdminDao=cn.bmxt.dao.impl.AdminDaoImpl
03 PhaseDao=cn.bmxt.dao.impl.PhaseDaoImpl
04 MajorDao=cn.bmxt.dao.impl.MajorDaoImpl
05 CourseDao=cn.bmxt.dao.impl.CourseDaoImpl
06 StudentDao=cn.bmxt.dao.impl.StudentDaoImpl
07 EnrollDao=cn.bmxt.dao.impl.EnrollDaoImpl
08 GradeDao=cn.bmxt.dao.impl.GradeDaoImpl
09 LogDao=cn.bmxt.dao.impl.LogDaoImpl
10 DocDao=cn.bmxt.dao.impl.DocDaoImpl
```

2. DAO工厂类的代码实现

DAO工厂类用于生产管理实体类DAO接口的实现类对象，它通过读取“dao.properties”文件中的配置信息，然后利用Java反射技术来创建需要的对象实例。在包“cn.bmxt.dao.common”中创建工厂类DaoFactory，其设计代码如5.5-02所示。

【代码5.5-02】文件/src/cn.bmxt.dao.common/DaoFactory.java，版本1.0。

```
01 package cn.bmxt.dao.common;
02 import java.io.IOException;
03 import java.util.Collections;
04 import java.util.HashMap;
05 import java.util.Map;
06 import java.util.Properties;
07 public class DaoFactory {
08   private DaoFactory() {}
09   private static final Map<String, Object> cache = Collections.
   synchronizedMap(new HashMap<String, Object>());
10   private static final Properties props = new Properties();
11   static {
12     try {
13       props.load(Thread.currentThread().getContextClassLoader().
   getResourceAsStream("dao.properties"));
14     } catch (IOException e) {
```

```
15        System.err.println("在classpath下未找到dao.properties配置文
    件！");
16        e.printStackTrace();
17      }
18    }
19    public static Object getInstance(String daoName) {
20      Object obj = cache.get(daoName);
21      if (obj == null) {
22        String className = props.getProperty(daoName);
23        if (className == null || "".equals(className)) {
24          throw new RuntimeException("指定的DAO实现类全限定名在dao.
    properties中未找到！");
25        }
26        try {
27          obj = Class.forName(className).newInstance();
28          cache.put(daoName, obj);
29        } catch (Exception e) {
30          throw new RuntimeException("加载指定DAO实现类的字节出现异常！",
    e);
31        }
32      }
33      return obj;
34    }
35 }
```

代码5.5-02说明如下：

（1）第08行，私有化构造方法，不允许用户创建DaoFactory类的实例对象。DaoFactory类仅对外提供一个静态的工厂方法，直接使用类名调用即可。

（2）第09行，通过调用Collections类的静态方法synchronizedMap获取一个支持同步（线程安全）的映射Map，作为DaoFactory类所维护的DAO实现类实例的缓存cache。

（3）第10行，声明一个静态的私有化常量props，用于保存从“dao.properties”文件中读取的配置信息。

（4）第11行~第18行是一个静态代码块，将“dao.properties”文件中的配置信息加载到静态常量props中。第13行代码首先获取当前线程上下文的类加载器ClassLoader，然后调用getResourceAsStream方法获取文件“dao.properties”的输入流，最后调用Properties类的load方法加载这个输入流，完成“dao.properties”文件的读取工作。第15行，当读取“dao.properties”文件产生异常时，输出对应的错误提示信息。

（5）第19行~第34行是一个静态的工厂方法getInstance，该方法根据传递的DAO接口名称返回一个与之对应的实现类的实例对象。第20行，首先根据用户传递的DAO接口名称在缓存cache中查找实例对象，紧接着在第21行进行判断，如果缓存中存在所需的实例对象，直接执行第33行代码返回该实例对象即可，否则执行第22行~第31行的代码块，这个代码块中的作用就是创建DAO实现类的实例对象并存入到缓存cache中。第22行代码会根据用户传递的DAO接口名称检索属性文件中是否配置了对应的DAO实现类，如果未配置，就在第24行中抛出异

常，否则就会获取配置的DAO实现类的全限定名称，然后在第27行的代码中利用Java反射技术创建出DAO实现类的实例对象。接着第28行代码将这个实例对象添加到缓存cache中，程序最终会跳出代码块执行第33行代码返回新创建的实例对象。当用户第一次调用getInstance方法获取某个DAO实现类的实例时，程序会创建一个新的实例对象并存入缓存；当用户后续再调用getInstance方法请求获取同一个DAO实现类的实例时，将会返回缓存中的实例对象，从而保证了每一个DAO实现类的实例对象在系统运行时仅存在一个。这也是单例模式的一种实现方式，可以节约系统资源，提高系统的性能。

我们可以编写代码来对DaoFactory类中的工厂方法进行测试，在“cn.bmxt.util”包下的JunitTest类中新增一个单元测试方法testDaoFactory，测试代码如5.5-03所示。

【代码5.5-03】DaoFactory类中工厂方法的测试代码，代码位置：/src/cn.bmxt.util/JunitTest.java。

```
01 @Test
02 public void testDaoFactory(){
03   SiteDao siteDao = (SiteDaoImpl) DaoFactory.getInstance("SiteDao");
04   SiteDao siteDao2 = (SiteDaoImpl) DaoFactory.getInstance("SiteDao");
05   System.out.println(siteDao);
06   System.out.println(siteDao2);
07 }
```

代码5.5-03中，第03行和第04行先后两次调用DaoFactory类的工厂方法getInstance来获取SiteDao接口实现类的实例对象，并分别打印，结果如下所示，说明使用DaoFactory类的工厂方法获取的实例对象是同一个。

```
01 siteDao = cn.bmxt.dao.impl.SiteDaoImpl@3581c5f3
02 siteDao2 = cn.bmxt.dao.impl.SiteDaoImpl@3581c5f3
```