文章编号:1001 - 9081(2007)10 - 2498 - 03

一个可动态扩充的数据访问对象模式

方 铖[1,2],曾 平[2]
(1. 武汉工业学院 计算机与信息工程系,武汉 430023; 2. 武汉大学 计算机学院,武汉 430072)
(fangcheng11@163.com)

摘 要:当前已有的数据访问对象(DAO)模式普遍存在诸多不足之处,例如与业务对象的耦合度较大,不能实现软件系统的动态扩充,实现代码重复,系统维护难度较大等。针对这些问题,借鉴数据绑定的有关思想,引入元数据、元模型的概念,利用 XML 语言的独立性,提出了一个独立性更强、可动态扩充的数据访问对象模式,并结合具体应用实例说明了该模式的使用过程。该模式的创新之处在于如果需要增加业务对象,只需要修改映射文件,不必改动 DAOFactory 类的任何代码;而且由于该模式用一个 DAO 实现类完成所有业务对象的数据访问,如果需要修改某个 SQL 语法,只需要修改这个 DAO 实现类,不必逐一修改各业务对象类对应的 DAO 实现类。

关键词:模式;数据访问对象;数据绑定;元模型
中图分类号: TP311.5 **文献标志码**:A

New DAO pattern with dynamic extensibility

FANG Cheng[1,2], ZENG Ping[2]
(1. *Department of Computer and Information Engineering, Wuhan Polytechnic University, Wuhan Hubei* 430023, *China*;
2. *College of Computer Science, Wuhan University, Wuhan Hubei* 430072, *China*)

Abstract: Currently the existing Data Access Object (DAO) patterns have several limitations. First, the interface of the patterns and business objects is tightly-coupled, which seriously affects the dynamic extensibility of software systems. Second, the patterns have duplicated implementation codes, which adds to the difficulty of system maintenance. To solve these problems, a new DAO pattern with stronger independency and dynamic extensibility was proposed in the paper based on the concepts of data binding, meta data and meta model. Finally, an example was given to illustrate the using process of the new DAO pattern. The greatest advantages of the new DAO pattern are as follows. If any business object is needed to add to the system, we don't have to modify any codes of the class DAOFactory. All that we need to do is to modify the mapping file. Furthermore, because we have only one DAO implementation class to accomplish all the data access to business objects, if some SQL statements are needed to be modified, all we need to do is to modify the DAO implementation class but not modify any business objects.

Key words: pattern; Data Access Object (DAO); data binding; meta model

0 引言

数据访问对象(Data Access Object,DAO)模式[1]提供了业务逻辑层(如业务对象)和持久存储层(如数据库资源)之间的抽象,封装了所有对数据源的访问,隐藏了数据源实现细节。当底层数据源发生变化时,DAO 提供的接口保持不变,从而实现了不同供应商产品、不同数据存储类型和不同数据源类型之间方便的移植。一般的 DAO 模式[2]如图 1 所示。

然而,从图 1 中也可以看出,一般的 DAO 模式存在如下一些问题:

1)DAO 模式虽然独立于数据源实现细节,但是不独立于业务对象。每次在软件系统中增加新的业务对象,都要修改 DAOFactory 接口。如要增加 Account 业务对象,除了要引入新的 AccountDAO 接口外,还需给 DAOFactory 接口增加 getAccountDAO 方法。可见,DAOFactory 接口与业务对象的耦合度较大,不能实现软件系统的动态扩充。

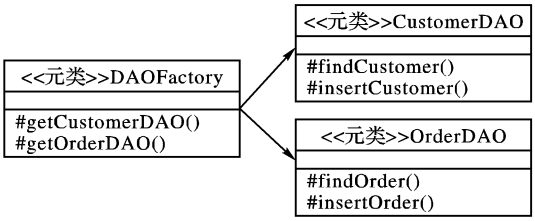

图 1 DAO 模式的一般实现

2)实现代码重复,系统难以维护。在图 1 所示的实现方案中,每一个业务对象都有一个 DAO 实现类与其对应。如 Customer、Order 等业务对象,分别有 CustomerDAO、OrderDAO 等 DAO 实现类与之相对应。在关系数据库实现情况下,这些 DAO 实现类都生成 Select、Insert 等 SQL 语句,这些语句基本类似。如果需要扩充某个 SQL 语法成分,所有业务对象的 DAO 实现类中的 SQL 语句都要进行扩充,系统维护工作量较大。

收稿日期:2007 - 04 - 27;**修回日期**:2007 - 06 - 25。

作者简介:方铖(1974 -),男,湖北黄冈人,讲师,硕士,主要研究方向:软件工程、信息工程; 曾平(1966 -),女,湖北武汉人,副教授,主要研究方向:软件工程、信息工程、操作系统安全。

针对这些问题,本文总结 DAO 模式的基本概念,借鉴数据绑定[3]的有关思想,引入元数据、元模型[4]的概念,利用 XML 语言的独立性,提出了一个独立性更强、可动态扩充的 DAO 模式,较好地解决了上述问题。

1　新的 DAO 模式

本文提出的 DAO 模式如图 2 所示。

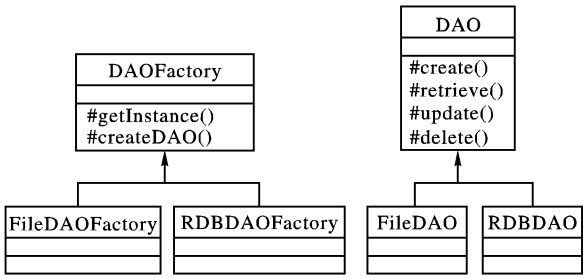

图 2 新的 DAO 模式

在图 2 中,DAOFactory 是抽象类,而不是接口。该类使用了 Singleton 模式[5]。它的 getInstance 方法如下所示。该方法能够根据系统配置文件的 factoryClass 项的内容,在运行时间动态地载入 DAOFactory 的相应子类。DAOFactory 类的 CreateDAO 方法用来创建 DAO 对象,它是抽象方法。DAO 接口完成所有业务对象的数据访问要求。它实行了 CRUD (Create,Retrieve,Update,Delete)模式[6]。其 create( )、retrieve( )、update( )、delete( )方法用来实现新建、查询、修改、删除 DAO 业务对象。这些都是抽象方法。

DAOFactory 类的 getInstance 方法如下:

```
private static DAOFactory m_instance;
public synchronized static final DAOFactory getInstance( ){
  if( m_instance = = null) {
    m_instance = ForClass( System. getProperty( "factoryClass")) .
    instance( ) ;
  }
  return m_instance;
}
```

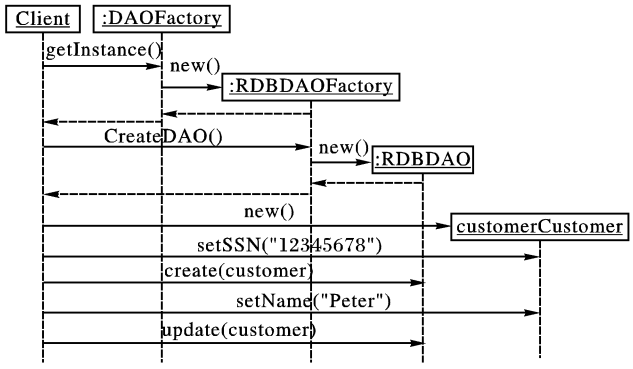

图 3　新的 DAO 模式的一个使用过程

DAO 模式的使用过程如图 3 所示。用户首先调用 DAOFactory 类的 getInstance( )方法,动态载入 DAOFactory 类的子类 RdbDAOFactory 类的对象,再通过该 RdbDAOFactory 对象,创建实际 DAO 类的对象(如图 3 中的 RdbDAO 类对象)。然后,用户便可以根据需要创建业务对象(如图 3 中的 Customer 类对象),并通过 DAO 类对象实现对业务对象的数据访问,如新建、修改、删除等。可见,我们用一个 DAO 类,完成了所有业务对象的数据访问工作。

2　实现举例

在图 4 中,我们给出了一个具体实例,进一步描述新的 DAO 模式。

RdbDAOFactory 类是 DAOFactory 类的子类。它有 2 个私有变量:DataSourceName 表示数据源的名字,MapFileName 表示映射文件的名字。

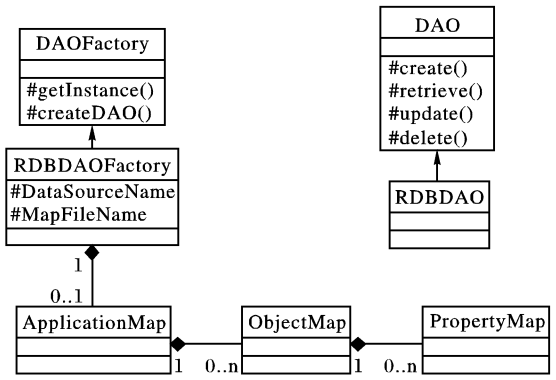

图 4　新的 DAO 模式的一个实现例子

下面给出了业务对象 Customer 和数据库表 Customer 之间的映射关系。它用 XML 形式描述了业务对象与数据库中的表/视图、业务对象属性与表属性之间的映射关系。业务对象 Customer 中有 Name、SSN、Address 三个属性,分别和表 Customer 中的 Name、SSN、Address 相对应。其中 SSN 是主关键字。

```
< ApplicationMap >
    < ObjectMap objectName = "Customer" tableName =
      "Customer" >
       < PropertyMap propertyName = "Name" ColumnName =
         "Name"columnType = "VARCHAR" key = "false" >
       < PropertyMap propertyName = "SSN" ColumnName =
         "SSN"columnType = "VARCHAR" key = "true" >
       < PropertyMap propertyName = "Address" ColumnName =
         "Address"columnType = "VARCHAR"key = "false" >
    </ObjectMap >
</ApplicationMap >
```

根据映射文件,可在 RDBDAOFactory 对象生成时,通过 Castor[3]或 Digester 等数据绑定工具,产生 ApplicationMap、ObjectMap、PropertyMap 类的对象。ApplicationMap 类封装了本应用程序的映射信息,ObjectMap 类封装了业务对象映射信息,PropertyMap 类封装了属性映射信息。一个 ApplicationMap 对象中含有多个 ObjectMap 对象,一个 ObjectMap 对象中含有多个 PropertyMap 对象。

图 4 中的 RdbDAO 类是 DAO 类的子类,实现具体的 create( )、retrieve( )、update( )、delete( )方法。因篇幅关系,本文只给出 update( )方法的实现代码,如下所示。

```
public void update ( Object o) throws DAOException{
  //获得 objectMap 对象
  ObjectMap objectMap = ( ObjectMap) ( getDomainMap. get( o.
    getClass( ) . getName( ) ) ) ;
  //根据映射信息及业务对象类的元信息,生成 SQL 语句
  StringBuffer dmlBuffer = new StringBuffer( "UPDATE") ;
  StringBuffer whereBuffer = new StringBuffer( "WHERE") ;
  dmlBuffer. append( objectMap. getTableName( ) ) ;
  dmlBuffer. append( "SET") ;
  for ( Iterator I = objectMap. getPropertyMap( ) ; i. hasNext( ) ; ) {
    PropertyMap propertyMap = ( PropertyMap) i. next( ) ;
```

```
    String value = null;
    try {
      Method method = o.getClass().getMethod("get" +
          propertyMap.getAttributeName(), null);
      value = method.invoke(o, null).toString();
    } catch (Exception ex) { throws new DAOException();}
    if (propertyMap.isKeyProperty()) {
      whereBuffer.append(propertyMap.getColumnName()).append
          ("=").append(value)whereBuffer.append(",");
    } else {
      dmlBuffer.append(propertyMap.getColumnName()).append
          ("=").append(value);
      dmlBuffer.append(",");
    }
  }
  String dml = dmlBuffer.substring(0, dmlBuffer,
    lastIndexOf(",")) + dmlBuffer.substring(0, dmlBuffer,
    lastIndexOf(",")) + ";";
  //执行 dml 指定的 SQL 语句
  try {
    execute_sql(dml);
  } catch (Exception ex) { throws new DAOException(); }
  return;
}
```

最后,给出应用程序片断,如下所示,以说明新的 DAO 模式是怎样使用的。

```
DAOFactory daoFactory = DAOFactory.getInstance();
DAO dao = daoFactory.createDAO();
Customer customer = new CustomerDAO();
Customer.setSSN("12345678");
dao.create(customer);                          // 建立一个 DAO 对象
…
customer.setName("Peter");
dao.update(customer);                          //修改一个 DAO 对象
…
dao.delete(customer);                          //删除一个 DAO 对象
…
```

应用该 DAO 模式,可以实现业务对象的动态扩充。比如要增加 Account 业务对象,只需修改映射文件即可,不必改动 DAOFactory 类的任何代码。另外,因为用一个 DAO 实现类来完成所有业务对象的数据访问工作,如果要修改某个 SQL 语法成分,只需修改相应的 DAO 实现类,而不必逐一修改各业务对象类的 DAO 实现类。

3 结语

在本文中,分析了当前数据访问对象(DAO)模式所存在的问题,借鉴数据绑定的有关思想,引入元数据、元模型的概念,利用 XML 语言的独立性,提出了一个独立性更强、可动态扩充的 DAO 模式。给出了该模式的结构图和使用顺序图,并举例说明了该模式的使用过程,揭示了该模式的动态扩充性。

新的 DAO 模式还需要进一步完善和提高,如数据访问时的事务管理、业务对象和数据库表之间的复杂映射、软件系统执行效率等等问题,都有待我们在下一步工作中加以解决。

参考文献:

[1] ALUR D. J2EE 核心模式[M]. 北京: 机械工业出版社, 2002.

[2] 周颖, 郑国梁. 模型层次与自省思想研究[J]. 计算机应用与软件, 2005, 22(12): 1 - 4.

[3] BIRBECK M. XML 高级编程[M]. 北京: 机械工业出版社, 2002.

[4] 王强, 何克清. 基于 MMF 的互操作性软构件库管理模型的研究[J]. 计算机工程, 2005, 31(16): 57 - 58, 172.

[5] GAMMA E. 设计模式: 可复用面向对象软件的基础[M]. 北京: 机械工业出版社, 2000.

[6] GRAND M. Java Enterprise Design Patterns: Patterns in Java [M]. John Wiley & Sons, 2001.

(上接第 2497 页)

```
State - vector 24 byte, depth reached 64, errors: 0
  77 states, stored (115 visited)
  174 states, matched
  289 transitions ( = visited + matched)
  426 atomic steps
hash conflicts: 0 (resolved)
Stats on memory usage (in Megabytes):
0.002 equivalent memory usage for states (stored * (State - vector +
overhead))
0.291 actual memory usage for states (unsuccessful compression:
  11806.01%)
  State - vector as stored = 3770 byte + 8 byte overhead
2.097 memory used for hash table (-w19)
0.320 memory used for DFS stack (-m10000)
0.148 other (proc and chan stacks)
0.086 memory lost to fragmentation
2.622 total actual memory usage
```

验证结果表明系统模型满足给定约束。

3 结语

使用 PROMELA 对系统进行精确建模比较困难而且很不直观,而 UML 的建模方式则很好地弥补了这一点,因此直接对 UML 模型进行检测成为一个很重要的问题。我们对由类图、状态图和顺序图构成的 UML 模型的检测问题进行了研究,提出并实现了从模型到 ROMELA 的转换算法以及验证顺序图与状态图一致性问题的方法。

参考文献:

[1] CLARKE E M, GRUMBERG O, PELED D A. Model Checking [M]. The MIT Press, 1999.

[2] MCMILLAN K L. Symbolic Model Checking[M]. Kluwer Acadamic Publishers, 1993.

[3] HOLZMANN G J. The model checker SPIN[J]. IEEE Transactions on software engineering, 1997, 23(5): 279 - 295.

[4] BOOCH G, RUMBAUGH J, JACOBSON I. The Unified Modeling Language User guide [M]. Addison-Wesley, Reading, Mass., &c., 1998.

[5] LATELLA D, MAJZIK I, MASSINK M. Towards a formal operational semantics of UML statechart diagrams[C]// Proceedings of FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object - Based Distributed Systems. Boston: Kluwer Academic Publishers, 1999: 15 - 18.

[6] LATELLA D, MAJZIK I, MASSINK M. Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker[J]. Formal Aspects of Computing, 1999, 11(6): 637 - 664.

[7] Object Management Group. OMG unified modeling language specification (Version 1.4)[S]. 2001.