基于元数据的对象关系映射研究

史周军　叶晓俊
（清华大学软件学院　北京100084）

摘　要　持久化框架较好地解决了关系型数据库的对象持久化问题，对象关系映射是持久化框架的核心内容。参考MOF标准，本文设计了面向对象和面向关系的元数据模型，在分析属性映射和关系映射的基础上，介绍了基于元数据的对象关系映射，最后讨论了利用XML技术的实现。

关键词　元数据，属性映射，关系映射，持久化

Research of Object-Relational Mapping Based on Metadata

SHI Zhou-Jun　YE Xiao-Jun
(School of Software, Tsinghua University, Beijing 100084)

Abstract　The persistence framework solves the problem of object persistence in relational databases. Object relational mapping is the core of persistence framework. This paper, which refers to the standard of MOF, has designed the metadata model of object-oriented and relational-oriented. Based on the property mapping and relationship mapping, we interpret the Object-Relational mapping using metadata. Finally, we discuss the implementation of the XML-based technology.

Keywords　Metadata, Property mapping, Relationship mapping, Persistence

1　引言

面向对象技术已广泛应用于需求分析、设计、编码和测试等软件生命周期的各个阶段。但是，在数据持久化方面，大部分应用程序却没有采用面向对象技术，这是由于数据持久化通常使用各种关系型数据库。在关系数据库中，数据是以元组而不是对象的方式存放的。在进行数据处理时，通常需在对象和元组之间做转换，这种不匹配为软件的开发和维护带来了困难。

数据访问主要有下列处理方法[1]：

（1）业务对象直接访问数据源：SQL代码可以出现在各个类中。该策略的优点是编码简单快捷，运行效率高；缺点是将业务处理和数据处理直接耦合在一起，当系统规模比较大时，其维护和升级工作都非常复杂，最终可能导致整个系统失控。

（2）采用分层体系结构：将数据处理从业务对象中分离出来，形成数据访问层。这种方法的优点是降低了业务处理与数据处理的耦合，在系统维护和升级时变得相对比较容易。但这种方式也存在问题，当数据库的模式结构进行改变时，必须重新编译源程序并发布。所以，在进行大型系统开发时，这种因变化带来的复杂性也将带来很高的生产成本和管理成本，降低了软件项目的可控制能力。

（3）将业务类映射到一个持久化框架：通过构造一个持久化框架来解决如何实现对象持久化。该策略进一步屏蔽业务类对数据类的依赖，隔离应用程序和数据库的变更。当对数据库进行了简单的变动，整个系统并不需要改动已有的编码和重新发布。该策略能够大大降低数据访问的编码和维护工作，以及对开发人员的技能要求。当然，这种做法与前两种方式相比也有缺点：它对应用的性能会产生一定的影响。不过，性能问题可以通过采用缓存机制等技术来解决。

从上面的几种数据处理策略可以看出，采用持久化框架是一种比较理想的方法。持久化框架通常包括访问层、对象关系映射、数据类型转换、事务管理、SQL代码生成等组成部分[2]。其中，对象关系映射是持久化框架的核心部分。采用元数据技术，通过在对象元数据和关系元数据之间建立映射关系，能够很好地解决对象关系之间的映射。

2　元数据和MOF

元数据（Metadata）是关于数据的数据。元数据在软件系统中广泛使用：首先，元数据提供基于系统的信息，能够帮助开发人员开发软件以及用户使用软件；其次，元数据支持对系统数据的管理和维护，能够使系统以最有效的方式访问数据等等。

由于元数据的广泛存在，使得对不同系统之间的集成和元数据的管理带来不便。为了规范对元数据的管理，对象管理组织OMG在1997年批准了元对象设施MOF（Meta Object Facility）[3]。

MOF是通用的、抽象的、用于定义元模型的语言。MOF是面向对象的，它定义了基本的元素、语法和元模型的结构[4]。MOF标准提供了：

（1）抽象模型：定义MOF对象和它们之间的关联。

（2）规则集合：将基于MOF的元模型映射到语言无关的接口，对这些接口的实现可以被用来访问和修改任何基于这个元模型的模型。

（3）定义元模型元素的生命周期、复合和关闭语义的规则。

（4）层级的反射接口（reflective interfaces）：定义了通用的操作，用于发现和操纵基于MOF元模型的模型。

史周军　硕士研究生，研究方向：软件工程技术，数据库技术；**叶晓俊**　副教授，主要研究方向为数据库技术，对象-关系数据库。

采用 MOF，不同的元模型可以互操作，应用程序可以在不了解特定领域模型实例接口的情况下，仍然能够读取和更新模型。

3 基于元数据的对象关系映射

面向对象模型和面向关系模型是不同的编程模式，当对象需要保存到关系型数据库中的时候，就需要在对象和关系模型之间进行映射。对象关系映射的内容主要包括属性映射和关系映射两个方面[6]，基于元数据，采用 XML 技术能很好地解决对象关系之间的映射。

3.1 元数据模型

参考 MOF 标准和 Java 元数据接口 JMI[5]，我们通过裁剪和简化后得到如图1所示的对象元数据模型。

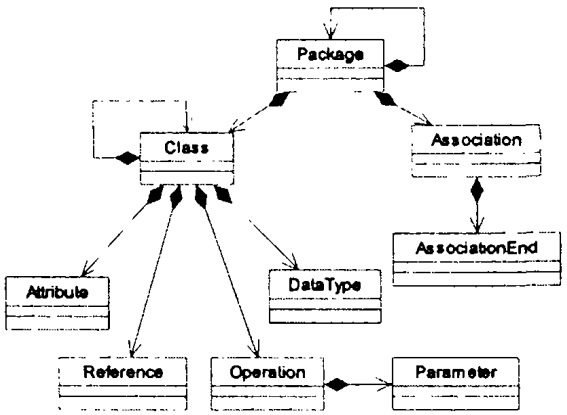

图1 对象元数据模型

在这个对象元数据模型中，包(Package)主要包括类(Class)和关联(Association)，包可以组合其它的包，所有内容均被包含在包内；类组合了类的属性、操作、数据类型和参照等，类也可以组合其它的类；一个关联一般包括包含两个关联点，用来描述类之间的关系。

在图2所示的关系元数据模型中，我们设计了包括模式对象(如表、视图)、模式对象的约束(如主键、唯一键和外键)等组成部分。

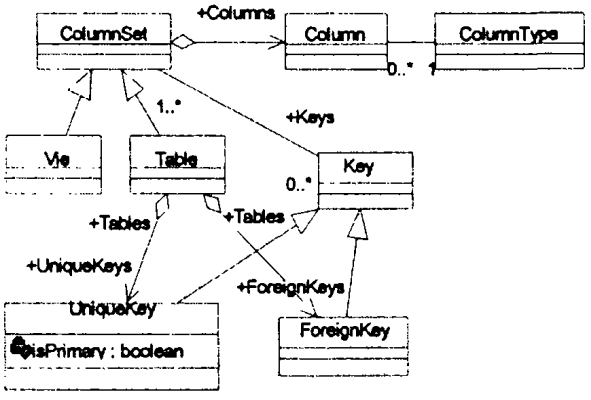

图2 关系元数据模型

3.2 属性映射

属性映射描述了如何在关系数据库中持久化一个对象的属性。根据对象属性的类型，在映射到关系属性时或者实现为一个物理属性，或者实现为一个虚拟属性(即，使用一个操作返回它的值)[6]。

比较复杂的情况包括将一个类的属性映射到多个表或者多个类的属性映射到一个表，将类的一个属性映射到表的多个字段或者将类的多个属性映射到表的一个字段等。属性映射最简单的实现是将一个类的所有属性映射到数据库中的一张表。

在对象模型中对象标识(OID)能够唯一识别一个对象，而在关系数据库中，OID 属性通常映射为一个表的主键。

3.3 关系映射

关系映射描述了如何在关系数据库中，将两个或多个对象之间的关系实现持久化。对象中的关系主要包括：关联关系、继承关系和聚合关系。

3.3.1 *关联关系映射* 对象之间的关联包括单向关联和双向关联，关联的多样性包括0..1，1，1..*n*，0..*n*。当关联的多样性是0..1或1时，关联的实现采用对象的一个引用。当关联的多样性是1..*n*或0..*n*时，关联则通过一个集合属性来实现。

在关系数据库中，数据库中的关系包括一对一、一对多和多对多关系。关系的实现一般是通过外键来实现的。

假设存在两个类：类 *A* 和类 *B*，类在关系数据库中对应的表分别为表 *a* 和表 *b*。当类 *A* 单向关联类 *B*，并且关联的多样性为0..1或1时，在关系数据库中的映射实现为：在表 *a* 增加一个表 *b* 的外键；如果类 *A* 和类 *B* 之间是双向关联关系，则在表 *b* 中也需要增加一个表 *a* 的外键。当类 *A* 单向关联类 *B*，并且关联的多样性为1..*n*或0..*n*时，使用相同的方法，外键增加在表 *b* 中，即多样性为"*n*"的一方；如果类 *A* 和类 *B* 之间关联的多样性均为1..*n*或0..*n*时，则需要在数据库中增加一个关联表，关联表中包含的属性一般是关系中涉及到的表中的主键组合，关联表的名称通常是它所关联的表的名称组合，通过增加一个关联表，将多对多关系转换成两个一对多关系，然后按照上面的方法实现。

3.3.2 *继承关系映射* 关系型数据库本身不能直接支持继承，但可以通过关系数据来描述。在数据库中，组织继承关系主要有三种策略：整个类层次结构映射一个表；每个类映射一个表；每个可实例化的类映射一个表。

(1) 整个类层次结构映射一个表。该策略把继承层次中所有类的属性的并集，映射到数据库的一个表中，每条记录中那些无用的字段使用 Null 值，并指定一个类型标识字段来区分不同的类。采用这种策略的优点是，当映射比较直接和简单，继承的层次也不是很多时，模式的改进比较容易。另外，由于父类的子类可以在一张表中全部找到，多态的读操作也不复杂。该策略的缺点是，对象属性的存储会浪费一些空间，空间浪费的多少取决于继承层次的深度，层次越深，不同类之间的属性越多，属性的并集就越大，也就越浪费空间。另外，如果在一个单独的数据库表中实现很多类，一般需要在表中加入索引，一张表上过多的索引，在进行更新数据时，需要同时更新所有的索引，影响处理性能[7]。

(2) 每个类映射一个表。该策略将每个类的属性放到不同的表中，抽象类也需要建立对应的表。采用这种策略，必须在存放父类实例的表和存放子类实例的表之间建立具有唯一性的外键约束关系，在每个表中插入一个识别 ID，将子类所在表的记录和父类所在表的记录关联起来。该策略的优点是，提供了非常灵活的映射机制，不同的类对应不同的表，多态读取只需要访问一张表；空间节约性好，唯一的冗余就是增加了额外的识别 ID 来连接不同的继承层次；映射比较直接，也易于理解，维护成本比较低。该策略的主要缺点是性能较差，访问类的实例时，采用跨表关联；可能会导致父类对应的表的负载过大；任何子类对应的表的写入锁，均要求其父类对应的表也必须具有写入锁[7]。

(3) 每个可实例化的类映射一个表。该策略将每个可实

例化的类的属性映射到不同的表中，并在表中加入该类的父类的所有属性，抽象类不需要映射到表。其优点是，单次的数据库操作就可以完成对一个对象的读写；提供了最佳的空间占用，没有任何的冗余属性。该策略的缺点是，维护成本较高，增加或删除父类的属性将会导致对所有子类映射的表结构的修改；特殊的多态查询难以编写，对叶结点类的查询将会非常复杂[7]。

3.3.3 聚合关系映射 聚合关系(Aggregation)的映射通常包括单表聚合和外键聚合两种模式。

单表聚合将被聚合对象的属性和聚合对象的属性放在同一张表中。其优势包括：在性能方面，该方案是最佳的。因为只需要访问一张表就能够获取一个带聚合的对象，并读入所有聚合对象；另外在删除聚合对象时，被聚合对象将会自动删除。缺点在于，由于聚合对象的字段增多，一次读取将会增大数据库读入的页面数；而且，如果被聚合的对象被多个对象所引用时，将会降低可维护性，因为每一次对被聚合对象类型的修改都会导致对所有引用它的聚合对象类型对应的表的Schema的修改[6]。

外键聚合将为聚合对象和被聚合对象分别创建单独的表，并在聚合对象的表中增加外键，连接到被聚合对象的标识ID。采用外键模式的优势包括：由于将对象进行了分解，使得维护更加容易，并提高了映射的灵活性；另外，将聚合对象放到单独的表中可以简化对这些对象的查询。缺点在于，采用外键聚合模式至少需要两次表访问，而单表聚合只需要一次；另外，被聚合对象不能随着聚合对象的删除而自动删除，需要编程来实现对象的一致性[6]。

3.4 基于元数据的映射实现

使用元数据而不使用反射技术来实现对象关系映射，其主要原因在于：元数据一般采用集中管理方式，可以方便地实现共享以及进行性能优化；元数据是结构化的，为对象关系映射及应用开发提供复杂查询能力奠定了基础；元数据可以添加标识信息，在使用时比较灵活。而采用反射技术需要在程序运行时对组件的结构进行解析，通常性能较差，可扩展性、复杂查询能力较弱。

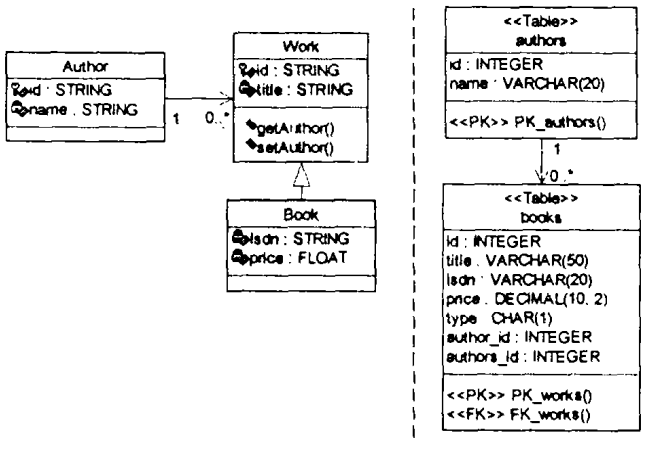

图3 对象关系映射示例

结合图1和图2中的关系和对象元数据模型，我们基于映射文件来实现对象关系映射。映射文件提供了将持久化对象的类结构和关系映射到关系数据库的哪个表或表间关系，以及如何映射等信息。

图3是基于元数据映射的一个简单示例：*Author* 类和 *Work* 类之间是一对多的关系，*Work* 类与 *Book* 类之间是继承关系，*Work* 是 *Book* 的父类。*Author* 类、*Work* 类和 *Book* 类及其间的关系等元数据信息存放在对象元数据模型中。关系数据库中的表 *Authors* 和表 *Books* 及主键、外键等元数据信息存放在关系元数据模型中。

下面给出了图3的映射文件片段：

```
〈OR -mapping〉
    〈class name="Author" table="authors"〉
        〈id name="id" column="id"〉
            〈generator class="assigned"/〉
        〈/id〉
        〈property name="name" column="name"/〉
    〈/class〉
    〈class name="Work" table="books"
        discriminator-value="W"〉
        〈id name="id" column=" id"〉
            〈generator class="native"/〉
        〈/id〉
        〈discriminator column="type" type="character"/〉
        〈property name="title"/〉
            〈many-to-one name="Author"column="id"
             not-null="true"/〉
        〈subclass name="Book" iscriminator-value="B"〉
            〈property name="isdn" column="isdn"/〉
             〈property name="price" column="price"/〉
        〈/subclass〉
    〈/class〉
〈/OR-mapping〉
```

在上面的映射文件中，〈ID〉元素表明将对象的OID映射为表的主键。采用“每个可实例化的类映射一个表”策略，〈discriminator〉元素是必需的，它声明了表的识别器字段。识别器字段包含标识值，用于说明应该创建哪一个子类的实例。识别器字段的实际值是根据〈class〉和〈subclass〉元素的 discriminator-value 得来的。通过 many-to-one 元素，可以定义该类与另一个持久化类的关联，实际上是指对一个对象的引用。

映射文件可以手工编写，也可以通过工具自动产生。通过图3的示例可以看出，基于元数据的对象关系映射解决了对象与关系数据库的模式结构之间的匹配问题，为业务对象的透明持久化奠定了基础。

结论 对象模型与关系理论的基本处理机制不同，使得这两种机制的结合并不理想。成功减少这些差距的关键在于理解这两种模式及其差异，然后根据应用系统的具体情况，做出恰到好处的对象关系映射策略。采用基于元数据的对象关系映射，较好地解决了基于关系型数据库的对象持久化问题，使应用程序具有良好的性能、可维护性和可扩展性，降低了应用开发人员在数据库方面的开发能力要求，简化了数据处理程序的编写，提高了软件开发的效率。基于元数据的对象关系映射能够适应不同应用场景的要求，以满足应用软件开发中对象持久化的需要。

参考文献

1 Ambler S W. The Design of a Robust Persistence Layer For Relational Databases. 2000

2 Yoder J W, Johnson R E, Wilson Q D. Connecting Business Objects to Relational Databases. URL: www. joeyoder. com/Research/objectmappings/Persista. pdf

3 MetaObjectFacility(MOF). Specification. http://www. omg. org/cgi-bin/apps/doc?formal/02-04-03. pdf

4 David Wiley &Sons. 鲍云志译. 应用 MDA. 人民邮电出版社，2003

5 Java™ Metadata Interface(JMI) Specification. http://java. sun. com/products/jmi/index. jsp

6 Ambler S W. The Fundamentals of Mapping Objects to Relational Databases. 2003

7 Ambler S W. Mapping Object to Relational Databases. URL: http:// www. AmbySoft. com/mappingObjects. pdf, 1997

8 Keller W. Object/Relational Access Layer A roadmap, Missing Links and More Patterns. URL: http://www. objectarchitects. de/ObjectArchitects/orpatterns