将对象映射到关系数据库：对象/关系映射(O/R Mapping)详解

满江红翻译团队声明：本文是在获得Scott W. Ambler的授权后进行翻译的，原文在
http://www.agiledata.org/essays/mappingObjects.html#MappingInheritance
，本文版权属于Ambler，我们放弃除署名权之外的翻译权利，可以在网上随意转载发表，不得用于商业目的，不得修改pdf文件。

本文是从Agile Database Techniques的第 14 章中截取的。

大多数现代商业应用开发项目使用面向对象技术，比如采用Java或者C#来创建应用软件，同时使用关系型数据库来存储数据。但这并不是要说你没有其它选择，也有许多应用程序是使用面向过程的语言开发，比如COBOL，而且也有许多系统使用对象型数据库或者XML数据库来存储数据。然而，因为面向对象和关系数据库技术到目前为止已经成为一种事实上的标准，在本章节中我假设你正在使用这些技术。如果你采用其它的存储技术，本文里的许多概念仍然适用，只需要做一点点修改（不必担心，Realistic XML总括了对象与XML映射的相关问题）。

在项目组通常用来创建以软件为基础的系统时，那些技术中，存在着面向对象技术和关系型技术之间的阻抗失配。不过这种阻抗失配很容易被克服，秘诀在于两点：你需要理解把对象映射到关系型数据库的过程，以及如何去实现这些映射。在本章节里，“映射”一词是用来表示如何把对象和对象之间的关系对应到数据库表以及表之间的关系。你将很快发现这并不像听起来的那样简单易懂，尽管它实际上也不是那么的糟糕。

目录

- 敏捷DBA的角色
- 基本概念
  - Shadow信息
  - 映射元数据
  - 如何使映射适合全过程
- 继承结构的映射
  - 整个层次结构映射到一张表
  - 每个具体类映射到单独的一张表
  - 每个类单独映射到一张表
  - 将类映射为一个通用的表结构
  - 多重继承映射

    - 映射策略之间的比较
- 映射对象关系
    - 关系的类型
    - 如何实现对象关系
    - 如何实现关系数据库中的关系
    - 关系映射
        - 一对一映射
        - 一对多映射
        - 多对多映射
    - 映射有序集合
    - 映射递归关系
- 映射类作用域(Class-Scope)属性
- 性能调优
    - 优化你的映射
    - 延迟读取
- 为什么数据Schema不应该主导对象Schema
- 实现方式对对象的影响
- 模型驱动体系结构(MDA: Model Driven Architecture)的含义
- 映射技术模式化
- 参考文献和阅读推荐

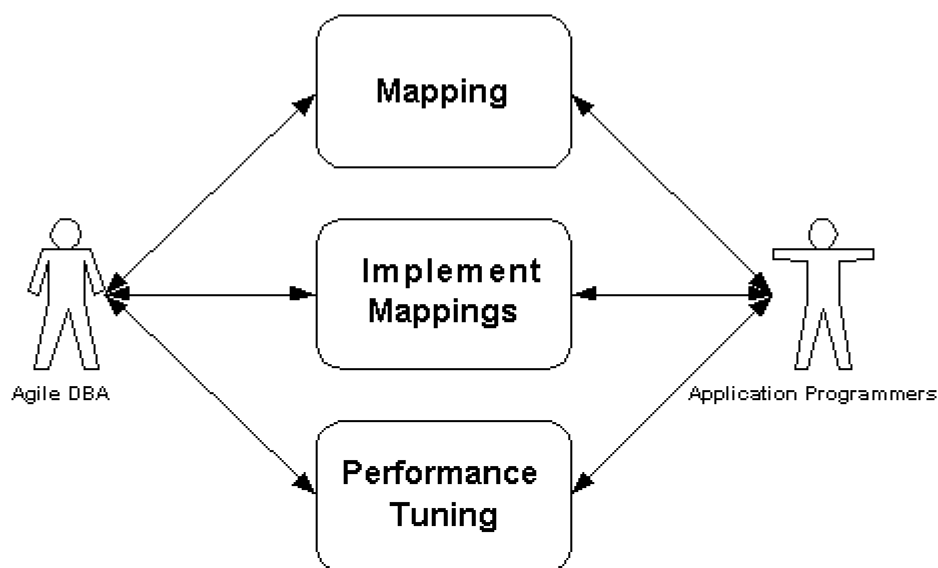
1. 敏捷 DBA 的角色

图 1 显示了一个敏捷 DBA 在映射对象到关系数据库的过程中所扮演的角色。其中我们关心三个主要的活动。

1. **映射**。基本的目标是决定一个有效的策略来持久化对象数据。这包括保存单个对象的属性以及对象之间的关联，同时也包括那些类之间的继承结构。
2. **实现映射**
3. **性能调优**

在图 1 中我们注意到有一个有趣的事情，敏捷的 DBA 和应用程序开发人员在在这三个主要活动中都在一起工作。虽然敏捷的 DBA 应该确保映射的有效性，但他们实际上并不是独自对其负责的。与他人协同工作而不单打独斗正是敏捷软件开发成功的关键所在。

图 1. 映射时敏捷 DBA 的角色。



2. 基本概念

在学习如何把对象映射到关系型数据库的过程中，通常是从映射一个类的数据属性开始的。一个属性可以映射到关系型数据库里 0 个或者多个字段。请记住，不是所有的属性都是持久性的，其中的一些只是用做临时计算的。例如，在你的应用程序中一个 *Student* 对象可能需要有一个平均分（averageMark）属性，但并不需要存储到数据库里，因为它是由应用程序计算得到。一个对象的某些属性可能本身也是对象，比方说一个 *Customer* 对象拥有一个 *Address* 对象作为其属性——这其实反映了两个类之间需要被映射的关系，*Address* 类本身也需要被映射。重要的是这是一个递归的定义：在需要的地方，一个属性将被映射到 0 个或者多个字段。

最简单的映射就是把一个属性映射到一个字段。当双方拥有一样的基本类型的时候，这甚至可以变得更简单。例如，双方都是 date 类型，或者属性是 string 类型而字段是 char 型，或者属性是 number 类型而字段是 float 类型。

映射术语

映射（动词）. 指的是如何把对象和对象之间的关系持久化到永久存储设备（这在里是关系型数据库）中的行为。

映射（名词）.如何将对象的属性或关系持久化到永久存储设备的定义的关系。

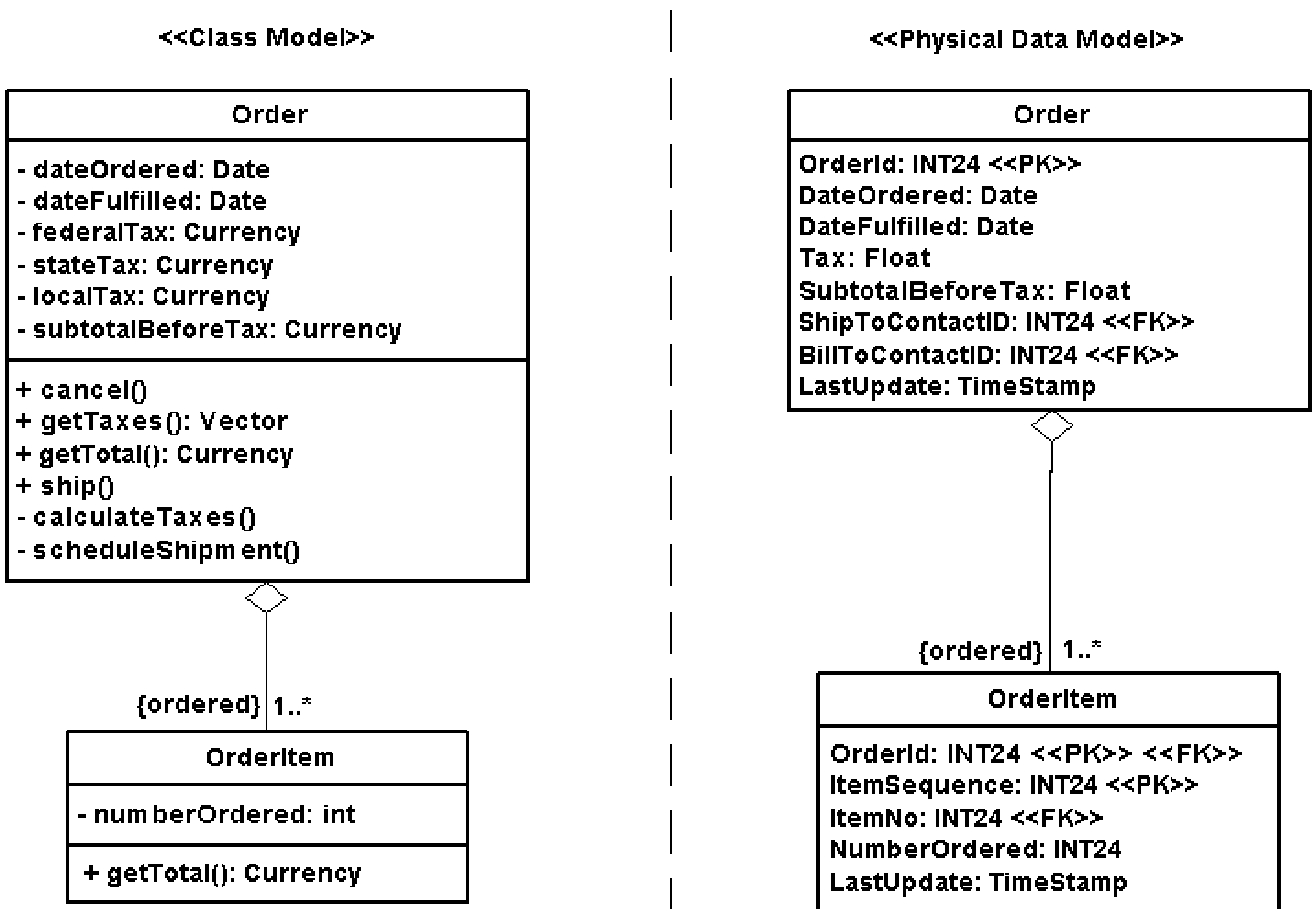
属性. 数据属性，是实际的物理属性，例如一个 firstName 字串；或者是由某个操作实现的虚拟属性，例如 getTotal（）方法返回一个订单的总数。

属性映射．描述如何持久化对象的属性的映射。 **关系映射**．描述如何持久化两个或者更多的对象之间的一个关系（关联，聚合或者组合）。

把类映射到表上会让许多事情思考起来更简单，有时候的确是这样映射的，但并非总是这样直接，除了少数特别简单的数据库，你将不会有机会在类和表之间进行简单的一一映射。在这章的后面你将看到继承映射。但是，就这一整章全面来看，通常对于初始的映射，单类映射到单表是适用的（性能调优可能会促使你对映射进行重构）。

现在，让我们从简单事物开始。图 2 显示了 2 个模型，一个UML的类图和一个遵循UML数据建模规则的物理数据模型。这两张图描绘了一个简单的订单系统。你可以看到如何映射类的属性到数据库的字段上。例如，图里显示*Order*类的*dateFulfilled*属性映射到*Order*表的*dateFulfilled*字段，*OrderItem*类的*numberOrdered*属性映射到*OrderItem*表的*NumberOrdered*字段。

图 2. 简单映射的例子



基本属性的映射很容易确定，有几个原因。首先，两个模型中使用相似的命名规则，这是采用敏捷建模实践“建模标准化”的一个方面；其次，通常是同一群人创建这两个模型。而当人们在不同的团队工作的时候，很容易做出不同的方案，即便是在各个团队本身的工作都很出色的时候也是这样，因为他们沿着不同的方向进行决策；第三，一个模型很容易用来驱动另一个模型的开发。在“不同的项目需要不同策略”一文中我讨论了当你创建一个新系统的时候，你的对象schema应该主导你的数据库schema的开发。

图 2 里面显示的两个 schema 虽然很相似，但还是存在一些区别。这些区别意味着不存在一个完美的映射。2 个 schema 间的不同点有：

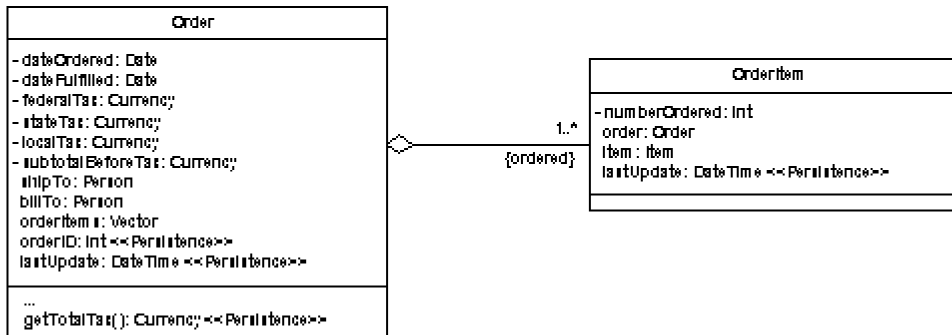
- 在对象 schema 里，tax 有多个属性而数据 schema 里只有一个。当对象被保存时，Order 类里 tax 的 3 个属性将被相加并保存到 tax 字段里。然而，当对象被读进内存时，这 3 个属性将需要被计算（或者使用一个延迟初始化的方式，这样每个属性仅仅在第一次被访问的时候计算）。一个像这样的 schema 上的区别说明数据库 schema 需要重构，把 tax 字段分成 3 个不同的字段。
- 数据Schema标明了键而对象Schema没有。表中的每一行都有一个全表唯一的主键值，行间的关系被用外键实现。而对于对象之间的关系，是通过使用引用而非使用外键。这暗示为了完整的持久化对象和它们的关系，对象需要知道数据库里面用来标识它们的键值。这些额外的信息被称为“shadow 信息”。
- 每个 Schema 里面使用了不同的类型。*Order* 里 *subTotalBeforeTax* 属性是 *Currency* 类型，而 *Order* 表里 *subTotalBeforeTax* 字段是 float 型。当你实现这个映射，你需要在这些数据的表示形式之间进行无损转换。

2.1 Shadow 信息

Shadow信息是指那些为了将对象持久化，而不得不维持的非业务数据。这通常包括主键信息，特别是当主键是没有业务含义的代理键值时；并发控制标识例如时间戳或者增量计数器；以及那些版本号。例如，在图 2 你可以看到*Order*表有一个*OrderID*的字段作为主键，一个*Order*类所没有的*LastUpdate*字段被用来乐观并发控制。为了正确持久化一个order对象，就需要实现包含这些信息的shadow属性。

图 3 显示一个对 *Order* 和 *OrderItem* 进行详细设计的类模型。和图 2 相比，有一些修改。首先，新的图显示了类需要正确持久化自己所需要的 shadow 属性。shadow 属性是实现可见的，在它们的名字前面是一个空格而不是一个“－”号，同时被指定了<<persistence>>进行说明（这不是一个 UML 标准）。第二，它显示需要在两个类之间实现联系所需要的辅助(scaffolding)属性。辅助属性，例如 *Order* 里面的 *orderItems* 列表，同样是实现可见的。第三，一个 *getTotalTax()* 操作需要被加到 Order 类里来计算 *Order* 表中 tax 字段所需要的值。这是为什么我用属性映射这个词来代替属性映射－你所想要做的是映射一个类里面的属性到数据库里的字段上，有时这些属性是通过简单的属性实现的，而其它某些时候是由一个或者多个操作所决定的。

图 3. 在一个类图里包含"shadow 信息"



我还没有讨论的一种 shadow 信息是用一个 boolean 类型的标志来表示当前一个对象是否存在于数据库中。这里的问题是当你把数据保存到一个关系型数据中，如果原先的对象是从数据库中获取出来的，你需要使用一个 SQL update 语句来保存数据，否则应该使用 SQL insert 语句。一个普通的解决方法是为每个类实现一个 *isPersistent* 的 boolean 型信号标志（图 3 里没有显示），当数据是从数据库里面读取的时候把它的值设置成 true，如果对象是新创建的话则设置为 false。

在UML社区里面的一个通用的风格约定是在类图里不显示shadow信息，例如键值或并发标识。类似的，通常也不显示支撑代码。因为每个人都知道你需要做这种事情，所以何必浪费时间去显示这些明显的事实呢？

Shadow信息不必用业务对象（business object）来实现，不过那样你的程序就要在其他地方处理这个问题。例如，在Enterprise JavaBeans (EJBs)里你把主键保存在EJB以外的主键类（primary key class）里，独立对象引用相关的主键对象。而进一步的，Java Data Object（JDO）则是在JDOs里面实现shadow 信息，而不是在业务对象（business object）里。

2.2 映射元数据

图 4 显示了元数据（meta data），这些是代表持久化图 3 里*Order*和*OrderItem*类所需要的属性映射。元数据是关于数据的信息。因如下原因使得图 4 显得很重要的。首先，我们需要某个方式来表现映射。我们可以把 2 个schema并排放在一起，就像图 2 里面那样，然后在它们之间画线，但是这很快会变的非常复杂，而另外一个选择是像图 4 里面这样列表；第二，映射元数据的概念对持久化框架的功能是非常重要的，这是一个可以让敏捷数据库技术发挥作用的数据库封装策略。

图 4.代表属性映射的元数据

Property	Column
Order.orderID	Order.OrderID

Order.dateOrdered	Order.DateOrdered
Order.dateFulfilled	Order.DateFulfilled
Order.getTotalTax()	Order.Tax
Order.subtotalBeforeTax	Order.SubtotalBeforeTax
Order.shipTo.personID	Order.ShipToContactID
Order.billTo.personID	Order.BillToContactID
Order.lastUpdate	Order.LastUpdate
OrderItem.ordered	OrderItem.OrderID
Order.orderItems.position(orderItem)	OrderItem.ItemSequence
OrderItem.item.number	OrderItem.ItemNo
OrderItem.numberOrdered	OrderItem.NumberOrdered
OrderItem.lastUpdate	OrderItem.LastUpdate

我采用的命名规则是非常直接了当的：*Order.dateOrdered* 指得是 *Order* 类里的 *dateOrdered* 属性。类似的还有，*Order.DateOrdered* 指得是 *Order* 表里的 *DateOrdered* 字段。*Order.getTotalTax()* 指得是 *Order* 里的 *getTotalTax()* 操作而 *Order.billTo.personID* 则是被 *Order.billTo 属性引用的* Person 对象里的 *personID* 属性。看起来最难理解的属性是 *Order.orderItems.position(orderItem)*，它指向在将要保存的 *OrderItem* 实例里 *Order.orderItems* 列表中的位置。

图 4 暗示了面向对象技术和关系型技术之间一个最重要的阻抗失配。Class同时实现行为和数据，而关系型数据库的表仅仅保存数据而已。这导致当你映射一个类的属性到关系型数据库时，你也需要映射那些操作到数据库字段上，例如：*getTotalTax()* 和 *position()*。虽然在这个例子里面没有出现，但是你常常需要映射仅代表一个属性两个操作（operation）到一个字段——一个操作是设置值如：*setFirstName()*，而另外一个是获取值，如 *getFirstName()*．　这些操作通常分别被称作setter和getter，或者mutator和accessor.

无论何时，一个键值都要被映射到类里的一个属性上，例如在 *OrderItem.ItemSequence* 和 *Order.orderItems.position(orderItem)* 之间的映射，这实际是关系映射的一部分工作，将在本章的后面进行讨论。这是因为在关系数据库里通过使用键值来实现数据间的联系。

2.3 如何使映射适合全过程

参看文章http://www.agiledata.org/essays/evolutionaryDevelopment.html。

3. 继承结构的映射

由于关系数据库不是生来就支持继承的，这就强制你必须将对象schema的继承结构映射到相应的数据库schema中去。多半是因为不牢靠的基类(译注：不牢靠的基类是指，有时基类很难修改，因为一旦修改基类，子类就容易出错)的原因，

面向对象社区不太提倡使用继承，而我的经验表明：之所以出现这个问题，是因为面向对象的开发者们大多缺乏封装的技巧，而非继承概念本身出了问题（Ambler 2001a）。我想说的是，事实上，你只需要做少量的工作，即可将一个继承层次映射到关系数据库中去，而这并不会影响你在合适的地方运用继承。

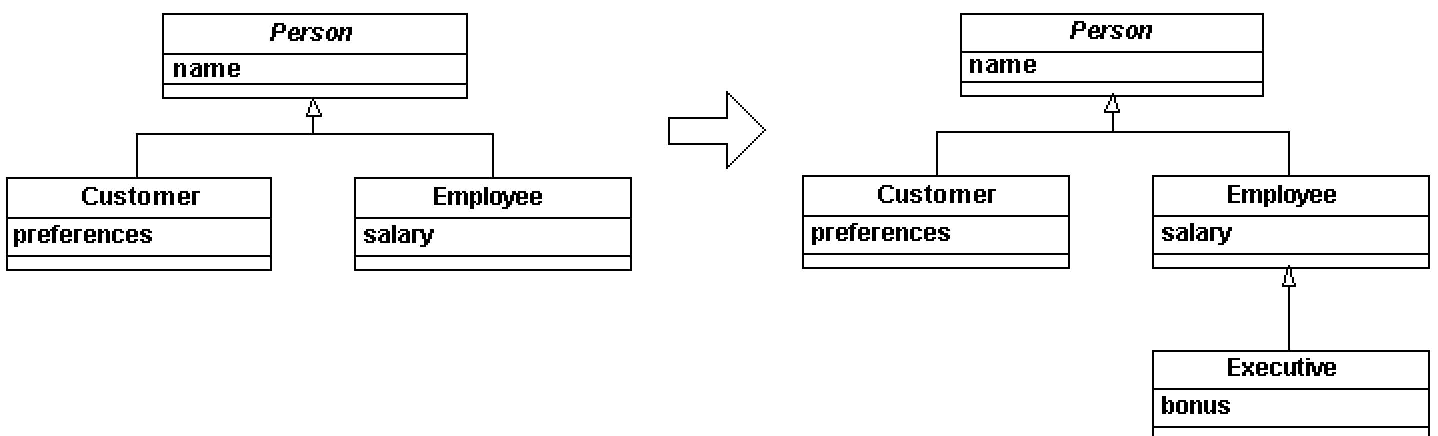
把对象存入关系数据库时，继承的概念会带来一些有趣的变化。如何在数据模型中组织那些通过继承而得到的属性？本节中，你将看到用来解决如何将继承关系映射到数据库的三种基本方法，以及第四种补充方法，它不限于继承映射。这些方法如下所示：

- 整个类层次结构映射到一张表
- 每个具体类单独映射到一张表
- 每个类单独映射到一张表
- 所有类映射到一个通用的表结构

如图 6 所示，有两个版本的类层次结构，我们将通过讨论如何映射这两个结构，来深入了解每种方法。第一个版本描述了三个类：一个抽象类 *Person*，和两个具体类 *Employee* 和 *Customer*。之所以知道 *Person* 是抽象类，是因为在图中它用斜体表示。在较早版本的 UML 中，会用约束"{abstract}"来表示抽象类。第二个版本在第一个版本的基础上，往类层次结构中添加了一个新的具体类 *Executive*。旨在描述，当实现了第一个类层次结构之后，有了一个新的需求，要求为雇员中的执行主管，而非普通雇员，颁发固定的年度分红。类 *Executive* 就是为了满足这一新功能而添加的。

简单起见，我没有对这些类的所有属性、属性的完整签名，以及类的任何操作进行建模。而这幅类图恰好足以满足我的目的，换句话说，这是一个敏捷的模型。此外，这些类层次结构本应该用分析模式中的*Party*模式（Fowler 1997）或*Business Entity*模式（Ambler 1997）。我并没有这么做，因为在这里，我并不是为了说明分析模式的有效应用，而是用一个简单的例子来说明继承层次结构的映射——我总是遵循敏捷建模（AM）的"每次只针对一个目标建模"（*Model With A Purpose*）的原则。

图 6. 一个简单类层次结构的两个版本

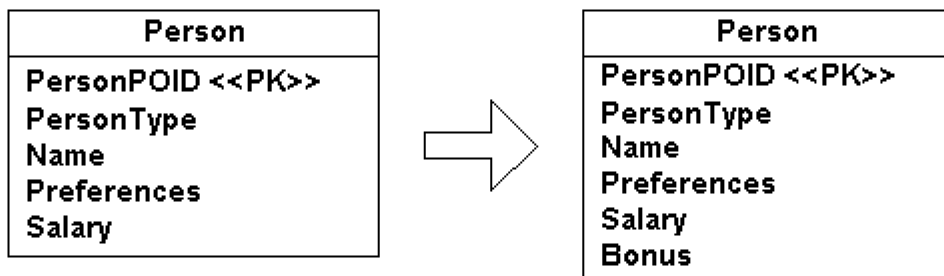


误用继承也会带来问题——比如，图 11.6 的层次结构本应该通过*Party* (Hay 1996, Fowler 1997) 模式或*Business Entity* (Ambler 1997) 模式进行更好的建模。举例来说，有人可能既是雇员又是顾客，为此你要在内存中保留多个对象，而这可能会给你的应用程序带来问题。我选择这个例子，是因为我需要一个简单的，易于理解的类层次结构来进行映射。

3.1 整个层次结构映射到一张表

按照此策略，把所有类的所有属性都存储到一张表中去。当采用这种方法时，图 6 中的类层次结构对应的数据模型如图 7 所示。这是非常直观的方式，每个类的属性都存储到表 *Person* 中，表名最好用类层次结构中根类的名字来命名。

图 7. 映射到一张表

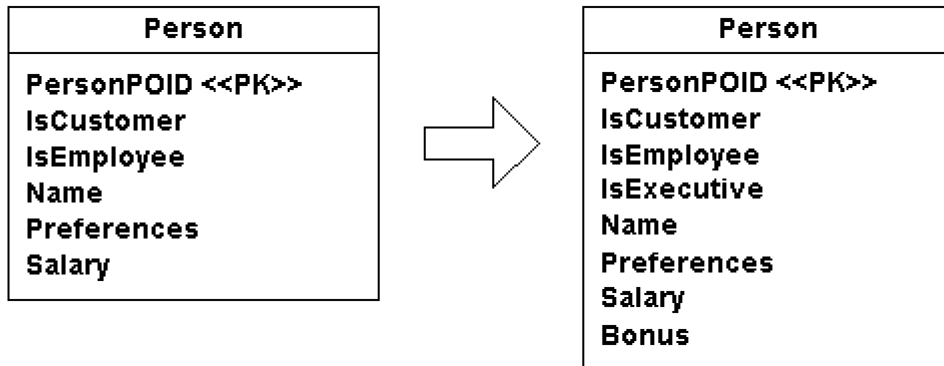


表里多加了两个字段——*PersonPOID* 和 *PersonType* 。图中，衍型(stereotype) <<PK>>说明第一个字段是表的主键，第二个字段是标识代码，用来指明一个人是顾客还是雇员，抑或两者皆是。*PersonPOID*是一个代理键（surrogate key），它是持久化对象的标识（POID，persistent object identifier），通常简称为对象标识（OID，object identifier）。本应该使用可选衍型（stereotype）<<Surrogate>>来标示的，但POID已经暗示了这层意思，这表明，这类衍型（stereotype)只会无谓地使我们的类图更复杂(参见AM实践“简单地描述模型”（Depict Models Simply）)。数据建模101(http://www.agiledata.org/essays/dataModeling101.html)详细讨论了代理键（surrogate keys）的相关内容。

用来识别对象类型的字段*PersonType*是必需的，这个对象可以由给定的数据库中的一行数据实例化而来。例如，取值为*E*表示该人是雇员，*C*表示是顾客，*B*则表示既是雇员又是顾客。这种方法看似直观，但当类型数目和类型间联合越来越多的时候，这种方法就渐渐变的力不从心了。例如，添加执行主管的概念，需要添加一个码值，比如以*X*来代表。对于值*B*来说，它代表的是既是雇员又是顾客，此时就显得有点不伦不类了。此外，有些（类间）联合中可能会包括执行主管，比如，一个人既是执行主管又是顾客也是很合乎情理的事情，这种情形也需要一个码值来表示。对于各种类型联合的情况，应该考虑使用*“用布尔值来代替类型码”（Replace Type Code With Booleans）*的数据库重构技法，如图 8所示。

为了简单，没有包含那些需要并发控制的字段，比如位于图 3 表中的时间戳字段，同时，用于数据版本跟踪的字段也没有包括在内。

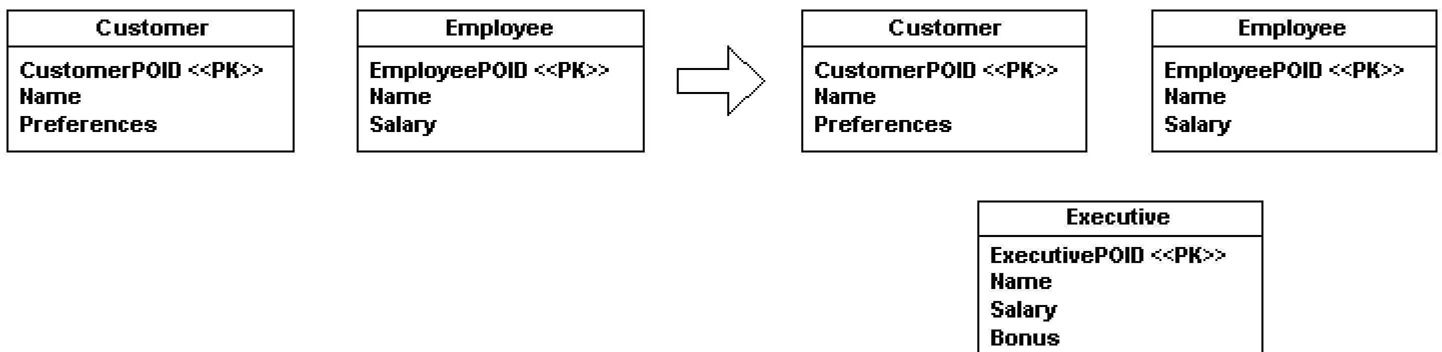
图 8. 一种重构方法



3.2 每个具体类映射到单独的一张表

这种方法为每个具体类创建一张表，每张表既包括对应类自己实现的那些属性，也包括它继承下来的那些属性。采用这种映射方法，图 6 中的类层次结构所对应的数据库物理数据模型如图 9 所示。每个具体类 *Customer* 和 *Employee* 都有对应的映射表，对象从这些表中被实例化，而抽象类 *Person* 则没有对应的表。分别为每张表分配了对应的主键，*customerPOID* 和 *employeePOID*。为了支持后加的类 *Executive*，需要做的全部事情就是添加一张相应的表，表中包括所有 *Executive* 对象所需要的属性。

图 9. 把具体类映射成表

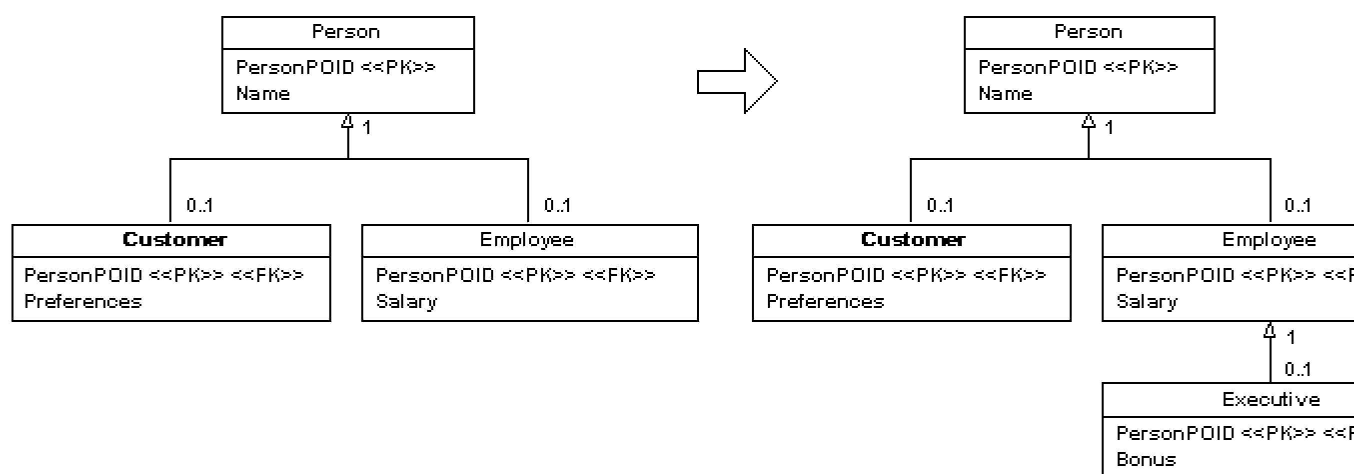


3.3 每个类单独映射到一张表

遵循这个策略，为每个类创建一张表，每个业务属性和任何必须的标识信息都对应于表中的一个字段（还包括并发控制和版本跟踪所需的其他字段）。将每个类都映射成一张单独的表，图 6 中的类层次结构对应的物理数据模型如图 10 所示。类 *Customer* 的数据被存储在两张表 *Customer* 和 *Person* 中，因此要获取这些数据，你需要连接这两张表（或者分两次读取，每张表读一次）。

键的应用很有意思。注意*personPOID*是如何作为所有表的主键来使用的。对于 *Customer*， *Employee*， 和 *Executive* 这些表而言，*personPOID*既是主键又是外键。对于*Customer*表，*personPOID*是它的主键，同时也作为外键来维系与表*Person*之间的关联。这是用<<PK>> 和 <<FK>>两种衍型（stereotype）来表示的。在一些较早版本的UML中，不允许为单个模型元素赋多个衍型（stereotype），但在UML1.4 版中，取消了这一限制。

图 10. 每个类单独映射一张表



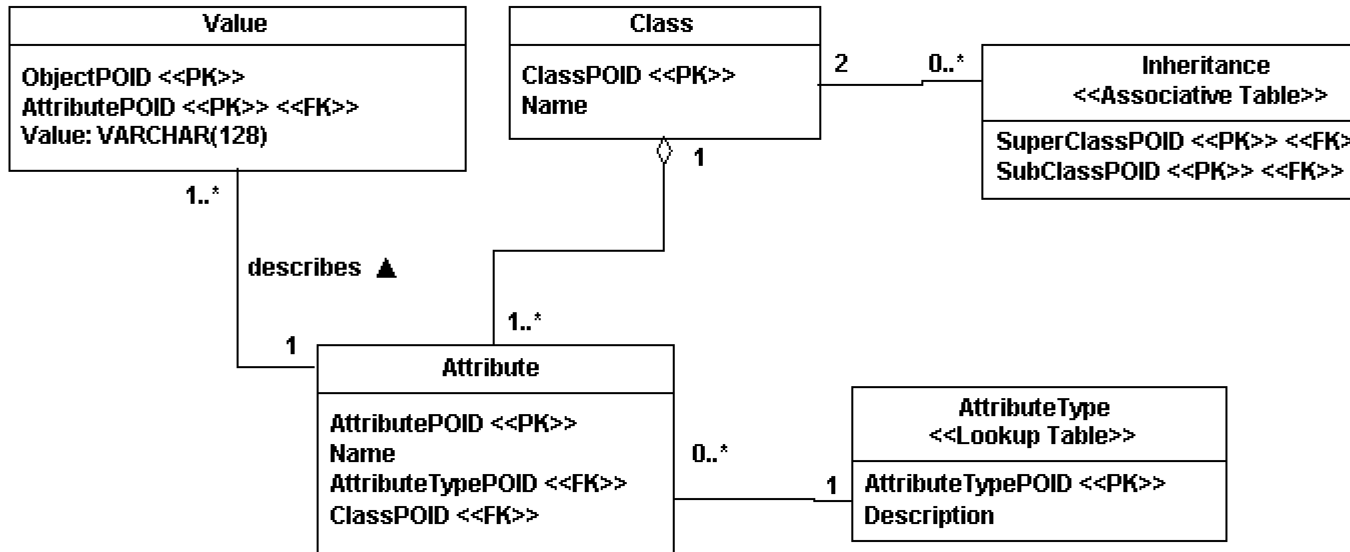
通常你可能会考虑的修改，是往表 *Person* 中添加一个类型字段或者布尔字段，用来表示 person 的可用子类。这个额外的花销将使一些查询变的更容易。在很多情况下，添加额外的视图（view）也是可行的选择，我更倾向于这种方法，因为与附加类型或布尔字段相比，这更易于维护。

3.4 将类映射为一个通用的表结构

将继承结构映射到关系数据库中去的第四种选择是采用一种通用的，有时被称为元数据驱动（meta-data driven）的方法来映射你的类，这种方法并不局限于继承结构，它支持所有形式的映射。图 11 中所示的，是用于存储属性值和遍历继承结构的数据 schema。这个 schema 并不完全，例如它还可以被扩展，用来映射关联关系，但对于我们的目的而言是足够用了。单个属性的值存放在 *Value* 表中，因此，如果要保存一个带有十个业务属性的对象，那么需要十条记录，每个属性对应一条记录。字段 *Value.ObjectPOID* 用来存储特定对象的唯一标识(这种方法假定对所有对象采用统一的键生成策略，假若不是这样的话，你必须适当的扩展这个表。）表 *AttributeType* 包含了代表基本数据类型的记录，如数据，字符串，

钱款，整数等等。将对象的属性值转换成为 varchar 类型保存到 *Value.Value* 字段中时，会需要这一信息。

图 11. 一个用于存储对象的通用数据 schema



让我们一起来看一个例子：将单个类映射到这种schema中。若要存储图 3 中的类 *OrderItem* ，表*Value*中要有三条记录，一条存储已订购的订单项的数目，一条存储*OrderPOID*的值，该订单项是相应订单的一部分，还有一条存储*ItemPOID* 的值，这个值用来描述订单项。如果你采用乐观锁的方法进行并发控制，你也可以考虑用第四条记录来储存*lastUpdated*这个shadow属性。表*Class*将为类 *OrderItem* 创建一行记录，表*Attribute*将在数据库中为每个属性创建一行记录（在本例子中，有三行或四行记录）

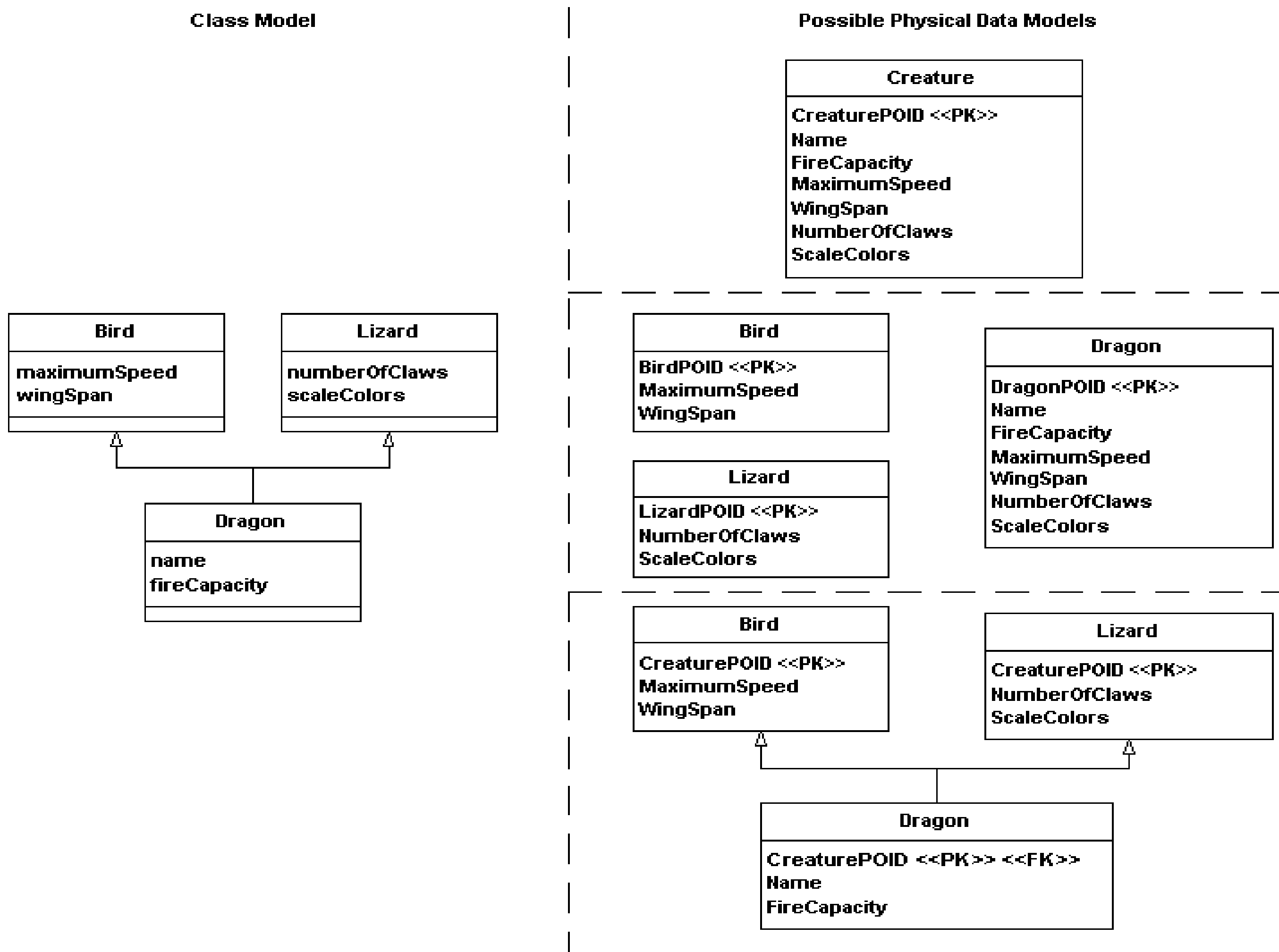
现在，将图 6 所示的 *Person* 和 *Customer* 之间的继承结构映射到这种 schema 中。表 *Inheritance* 是继承映射的关键。每个类将由表 *Class* 中的一行来表示。在表 *Inheritance* 中也有一行，*Inheritance.SuperClassPOID* 的值指向表 *Class* 中表示类 *Person* 的行， *Inheritance.SubClassPOID* 的值指向表中表示类 *Customer* 的行。映射剩余的结构层次关系，需要在 *Inheritance* 表中为每个继承关系都添加一行。

3.5 多重继承映射

到目前为止，我所关注的是单根继承层次结构的话题，所谓的单根继承是指，子类，如 *Customer* ，直接继承单一的父类，如 *Person* 。多重继承是指一个子类有两个或两个以上的直接父类，如图 12 所示，*Dragon* 直接继承了类 *Bird* 和 *Lizard* 。在面向对象语言中，多重继承通常被认为是有问题的，自 1990 年以来，我只见过在一个领域问题中使用多重继承是有意义的，因此，多数语言不支持多重继承。但是像 C++和 Eiffel 这样的语言支持多重继承，所以也存在需要将多重继承层次结构映射到关系数据库中去的情况。

图 12 展示的是，对多重继承分别使用三种继承映射策略而得到的结果数据 schema。如图所示，多重继承映射也是很简单的，跟单根继承映射相比，并没有任何特别的东西。我所经历的最大挑战是，当将整个层次结构映射到同一张表中时，如何为这张表取一个合理的名字，在本例中，用 *Creature* 最合适。

图 12. 映射多重继承



3.6 映射策略之间的比较

如表 1 所示，就这些策略而言，没有一种策略对所有场合都是理想的。我的经验表明，最容易奏效的策略是，先用每个层次结构映射一张表的策略，接下来如果需要，就重构相应的 schema。有时，当我的团队被“纯设计方法”的工作方式所驱动，我会先采用每个类映射一张表的策略。我尽量不使用每个具体类映射一张表的策略，因为这样做最后导致的典型结果是，需要把数据在表之间拷来拷去的，这就强制我在项目的初期就要对它进行合理重构。我很少使用通用 schema 方法，很简单，因为它不具有很好的可伸缩型。

在任何应用程序中，你都可以联合使用前三个映射策略：每个层次结构一个张表、每个具体类一张表和每个类一张表， 理解这一点很重要。你甚至可以在一个大型的层次结构中联合使用这三种策略。

表 1. 继承映射策略的比较

Strategy **策略**	**Advantages** **优势**	**Disadvantages** **缺陷**	**When to Use** **使用的时机**
每个层次结构一张表	方法简单。 添加新类很方便，你只需要为新加的数据添加新的字段即可。 通过简单的修改行的“类型”字段来实现多态。 因为数据在一张表中，因此数据的访问速度很快。 因为所有的数据在同一张表中，特别容易生成专门的报表。	因为所有的类直接关联到同一张表，类层次结构内的耦合度增加。修改一个类将影响整张表，而整张表的改变又会影响类层次结构中的其他类。 数据库空间存在潜在浪费。 如果已有类型之间有较大的重叠，则暗示着类型复杂化了。 对大的层次结构而言，表会非常大。	对那些类层次结构内部的类型间不存在或存在极少重叠情况的简单类层次结构和（或）继承深度比较浅的类层次结构而言，这种策略很好
每个具体类一张表	由于单个类的所有数据都存储在一张表中，所以特别容易生成专门的报表。 访问单个对象数据的性能高。	类修改的时候必须修改它对应的表，同时也必须修改它所有的子类对应的表。举例来说，如果你给类 *Person* 添加了 height 和 weight 两个属性， 那么你需要给表 *Customer*， *Employee*， 和 *Executive* 添加对应的字段。 当一个对象改变了它的角色，比如，你雇用了你的顾客，你需要把对象的数据复制到相应的表中，并为它分配一个新的 POID 值（或者可以重用已有的 POID 值）。 很难既支持多角色又能保持数据的完整性。比如，你把既是顾客又是雇员的人的名字存放到哪里？	当类型很少改变，类型之间极少重叠时，可以选择使用这种策略。
每个类一张	因为是一一对应的映射，	每个类一张表，则在数据库	在类型之间有较大

表	所以容易理解。 每个类型的记录分别在相应的表中，因此可以很好的支持多态。 修改父类，或添加新的子类时，只需要简单的修改或添加一张表。 数据的大小跟对象个数的增长成正比。	中有很多表（还需要附加表用来维护类之间的关系）。 使用这种方法，使得读写数据需要更长的时间，因为需要访问多张表。如果你明智的组织你的数据库，把一个类层次结构中的每张表放入不同的物理驱动盘盘面上（假设驱动器磁头的所有操作相互独立），就可以提高数据的读写速度。 从数据库生成专门的报表特别困难，除非添加视图来模拟所需要的表。	重叠，或类型会频繁的修改的情况下使用这种策略。
通用schema	当用一个稳定的持久化框架来封装数据库的访问，可以工作的非常好。 可以扩展到提供元数据，以支持包括关系映射在内的更大范围的映射。简而言之，它是元数据映射引擎的起点。 非常灵活，可以快速的改变存储对象的方式，因为只需要更新存储在表 *Class*，*Inheritance*，*Attribute* 和 *AttributeType* 中的元数据。	很先进的技术最初可能难以实现。 使用这种方法，需要访问很多数据库的记录来创建一个对象，因此它只适用于少量数据的情况。 你可能需要一个小型的管理程序来维护元数据。 因为要访问多行记录来获取一个对象的数据，因此，用这种数据生成报表显得异常困难。	适用于如下场合： 处理少量数据的复杂应用程序； 不常访问数据的应用程序； 可以预先将数据读入到缓存的应用程序。

4. 映射对象关系

除了属性映射与继承映射，你还需要领会关系映射的艺术。有三种你需要进行映射的对象间关系：关联、聚合以及组合。在这里，我将把这三种关系同等看待——尽管涉及到引用完整性的时候，三者有些微妙差别，但他们的映射方式是一样的。

4.1. 关系的类型

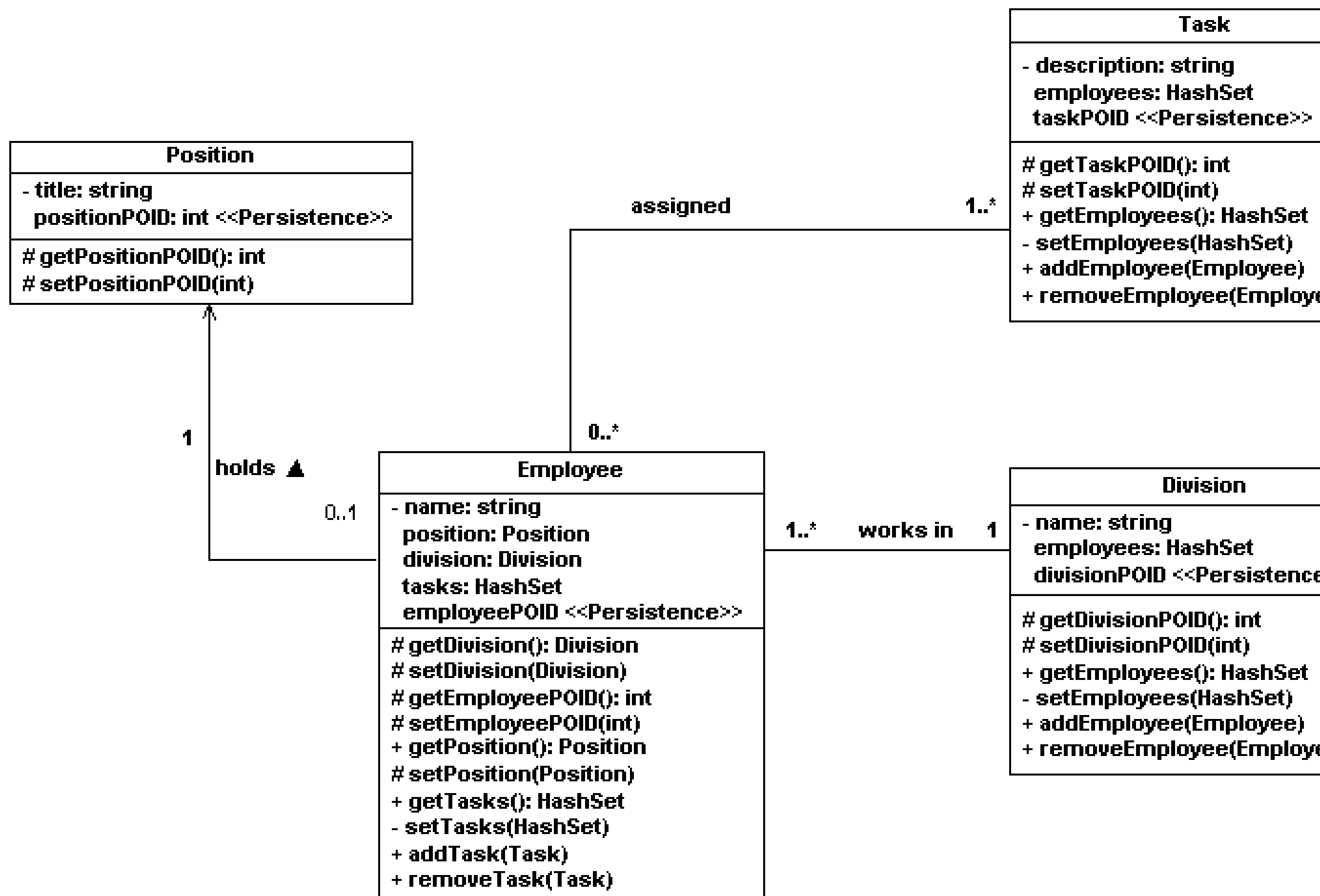
在做映射时，你需要关心两类对象关系。第一类是基于多重性(multiplicity)的，包含三种类型：

- 一对一关系**(One-to-one relationships)**。这是一种两端多重性(multiplicity)最大值都为 1 的关系(译注：两端最多只有一个对象）。举个例子来说就像图 13 中 *Employee* 与 *Position* 之间的*拥有(holds)*关系。每个雇员拥有且仅拥有一个职位，每个职位可能拥有一个雇员（有些职位还可能空缺）。
- 一对多关系**(One-to-many relationships)**。也被称作多对一关系(many-to-one relationship)，这种关系产生于一端多重性(multiplicity)最大为 1，而另一端大于 1 的场合。例如 *Employee* 与 *Division* 之间的*隶属(work in)*关系。每个雇员在一个部门工作，任何给定部门都有一个或者多个雇员在里面工作。
- 多对多关系**(Many-to-many relationships)**。这是一种两端多重性(multiplicity)最大值均大于 1 的关系。例如 *Employee* 与 *Task* 之间的 *分派(assigned)*关系。每个雇员可以被分派一个或多个任务，每个任务可以被指派给 0 个或多个雇员。

第二类是基于方向（directionality）的，包含两种类型：单向关系和双向关系。

- 单向关系(Uni-directional relationships)。单向关系是指一个对象知道与其关联的其他对象，但是其他对象不知道该对象。例如，图 13 中， *Employee* 和 *Position* 之间的*拥有(holds)*关系，图中该关系是用带开口箭头的直线来表示的。*Employee* 对象知道其所拥有的职位，而 *Position* 对象不知道拥有它的雇员是谁（没有需要知道的必要）。不久你就会看到，单向关系要比双向关系容易实现。
- 双向关系(Bi-directional relatinships)。双向关系是指，关联两端的对象都彼此知道对方。例如 *Employee* 与 *Division* 之间的*隶属(work in)*关系。*Employee* 对象知道自己工作的部门，而 *Division 对象*也知道有哪些雇员在本部门工作。

图 13. 对象间的关系



对象schema中有可能包含全部六种关系的组合。然而，关系技术并不支持单向关系的概念——在关系数据库中，所有的关联都是双向的，这也是对象技术与关系技术之间阻抗失配(impedance mismatch)的一个方面。

4.2. 如何实现对象关系

对象 schema 中的关系是通过对象的引用及操作来实现的。当多重性(multiplicity)是 1(比如 0..1 或者 1)的时候，这种关系通过一个对象引用、一个 getter 操作以及一个 setter 操作来实现。举例来说，图 13 中，类 *Employee* 是通过组合 *division* 属性、返回 *division* 属性值的 *getDivision()*操作以及设置 *division* 属性值的 *setDivision()*操作来反映某个雇员隶属于某一部门这个事实的。用来实现对象关系的属性和操作通常称为辅助属性和辅助操作。

多重性(multiplicity)为“多”(比如 N，0..*，1..*)的关系是通过集合属性(collection attribute)（比如 Java 中的 *Array* 或者 *HashSet*），以及操纵该集合的操作来实现的。例如，类 *Division* 定义了名叫 *employees* 的 *HashSet* 属

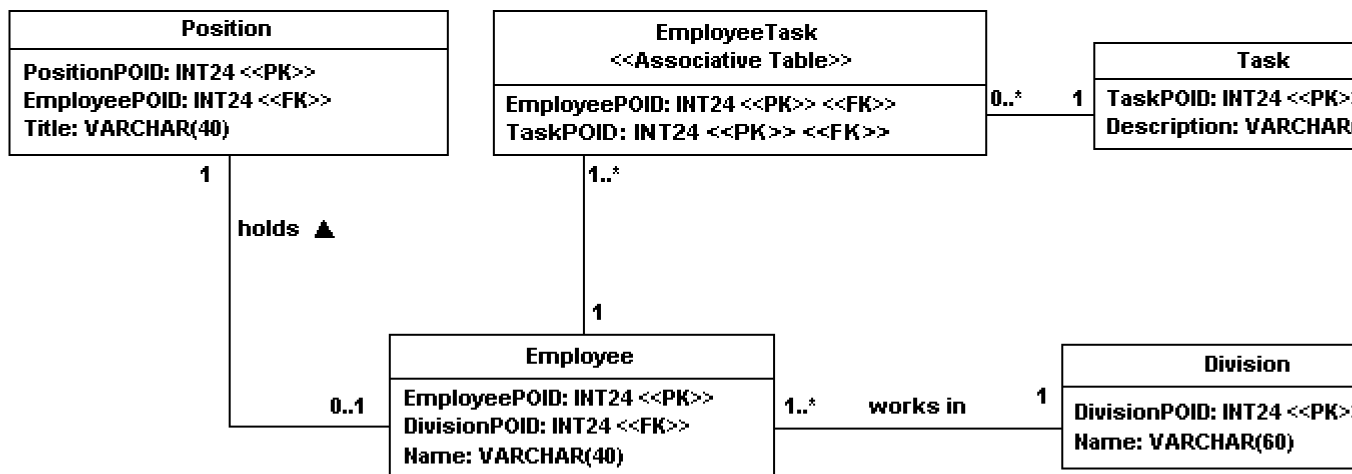
性、用来取值的 *getEmployees()*、用来设值的 *setEmployees()*、将一个雇员对象 (employee)加入到 HashSet 的 *addEmployee()*以及从 *HashSet* 中删除雇员对象(employee)的 *removeEmployee()*。

关系是单向的时候，只需要在“知道其它对象”的对象中实现代码即可。比如 *Employee* 与 *Position* 间的单向关系，只需由类 *Employee* 来实现关联。另一方面，双向关联则需要两个类都实现相关代码，正如你在 *Employee* 与 *Task* 之间的多对多关系(many-to-many relationship)中看到的那样。

4.3. 如何实现关系数据库中的关系

关系数据库中，关系是通过使用外键(foreign key)来维护的。外键的值可能是另一个表（某个行）键值的一部分，也可能就等于那个表（某个行）的键值。一对一关系中需要其中一张表来定义外键。在图 14 中你可以看到表 *Position* 包含了 *EmployeePOID*，它是一个指向 *Employee* 表的外键，用来实现两者的关联。换种方式，在 *Employee* 表中定义 *PositionPOID* 字段，也很简单。

图 14. 关系数据库中的关系



要实现一对多关系，你需要定义一个从"一表"指向"多表"的外键(译注：在 one-to-many 关系中，UML 里所指的 one 和 many 和数据库里的 multiplicity 是反过来的。比如一个 division 有几个 employee，UML 里认为 division 是 one 端，因为一个 division 对应几个 employee；而数据库 Entity-Relationship Diagram 里的 one 端指的是 employee，因为一个 employee 只会参与一个 division-employee 关系，这个值就是所谓的 multiplicity)。 例如 *Employee*（多

方）中就包含一个*DivisionPOID*字段（外键），用于实现与*Division*（一方）的*隶属(work in)*关系(注：同时，字段*DivisionPOID* 在*Division* 中是主键)。你也可以选择增设一个关联表(associative table)来实现一对多关系(one-to-many relationship)，从而很容易实现多对多关系(many-to-many)。

在关系数据库中，有两种方式可以用来实现多对多关联(many-to-many association)。第一种是在每个表中定义多个指向其他表的外键。比如要实现*Employee*与*Task*之间的多对多关系,你可以在*Employee 表*中定义五个*TaskPOID*字段，在*Task*表中定义七个*EmployeePOID*字段。不过当你想要赋予一个雇员超过五项任务，或者一项任务被分配给超过七个雇员时，采用这种方法就会遇到麻烦。更好的方式是实现一张被称为的关联表(associative table)的表。正如图 14 中的*EmployeeTask*所示，关联表包含了与其相关联的表的主键的组合。采用这种方式，你可以将同一项任务分配给五十个人，或者给同一个人指派二十项任务，完全没有问题。这个基本"技巧"的要点在于将多对多关系转化为两个一对多关系，它们都与同一个关联表相关联。

因为外键是用来连接表的，所以关系型数据库中的所有关系都是双向的。这也就是为什么你在哪张表中实现一对一关系都无所谓的原因，连接两表的代码相差无几。比如，图 14 中，对于现有的 schema，连接两者（*Position*和 *Employee*）之间的 holds 关系所使用的 SQL 代码应为

```
SELECT * FROM Position, Employee

WHERE Position.EmployeePOID =
Employee.EmployeePOID
```

假如在 Employee 表中实现外键，SQL 代码应为

```
SELECT * FROM Position, Employee

WHERE Position.PositionPOID =
Employee.PositionPOID
```

在数据库中采用一致的键策略可以极大地简化进行关系映射所需的努力。首先是尽可能使用单字段的键。然后是采用某种全局唯一的代理键(surrogate key)，可以遵从GUID或者HIGH-LOW生成策略。这样的好处是你只需对同一种类型的键值列进行映射。

既然我们明白了如何分别在两种技术(译注：对象技术与关系数据库技术)中实现关系，那么我们来看看怎么对它们做映射。我会从对象关系到关系数据库映射的角度来讲解。记住一件事情，某些情况下你要做出设计决策。再之，谨防"神奇的 CASE 工具栏按钮"，不要幻想它们会为你包办一切。

4.4. 关系映射

关系映射的一条通用的经验规则是你应该使映射前后的多重性保持一致。因此一对一的对象关系映射成一对一的数据关系，一对多的映射成一对多的，多对多的映射成多对多的。不过世事无绝对，你也可以将一对一关系映射成一对多甚至多对多关系。这是因为一对一关系是一对多关系的子集，一对多关系也同样是多对多关系的子集。

图 15 描述了图 13 的对象 schema 与图 14 的数据库 schema 之间的属性映射。注意我只需要映射对象的业务属性和 shadow 信息，不需要映射像 *Employee.position* 和 *Employee.tasks* 之类的辅助属性(scaffolding attribute)。这些辅助属性是通过映射到数据库中的 shadow 信息来表示的。当关系信息被读入内存，储存于主键字段中的值就会被解释成对象中相应的 shadow 属性。与此同时，通过为相关对象的辅助属性设置适当的取值，主键字段所表达的关系也会被建立起来。

图 15. 属性映射

Property	**Column**
Position.title	Position.Title
Position.positionPOID	Position.PositionPOID
Employee.name	Employee.Name
Employee.employeePOID	Employee.EmployeePOID
Employee.employeePOID	EmployeeTask.EmployeePOID
Division.name	Division.Name
Division.divisionPOID	Division.DivisionPOID
Task.description	Task.Description
Task.taskPOID	Task.TaskPOID
Task.taskPOID	EmployeeTask.TaskPOID

4.4.1 一对一映射

考察*Employee*与*Position*之间一对一的对象关系。我们可以采取无论何时，只要一个*Position对象*或者一个*Employee*对象被读入内存，应用程序都会自动遍历*拥有(holds)*关系并自动将关联对象也读入内存的方案。另一个可供选择的方案是在代码中手工遍历拥有关系，采用延迟读取方式，即：在应用程序需要的时候才读入另一个对象。两种方案的权衡在实现引用完整性中进行了讨论。图 16 显示了对象关系是如何被映射的。

图 16. 映射关系

对象关系	从	到	基数	自动读取	列	框架属性
拥有(holds)	Employee	Position	One	Yes	Position.EmployeePOID	Employee.position
被拥有(held by)	Position	Employee	One	Yes	Position.EmployeePOID	Employee.position
隶属(works in)	Employee	Division	One	Yes	Employee.DivisionPOID	Employee.division
属下有(has working in it)	Division	Employee	Many	No	Employee.DivisionPOID	Division.employees
被赋予(assigned)	Employee	Task	Many	No	Employee.EmployeePOID EmployeeTask.EmployeePOID	Employee.tasks
赋予(assigned to)	Task	Employee	Many	No	Task.TaskPOID EmployeeTask.TaskPOID	Task.employees

让我们一步一步地完成获取单个 *Position* 对象的工作：

1. 将 *Position* 对象读入内存。
2. 自动遍历*拥有(holds)*关系。
3. *Position.EmployeePOID* 字段的取值被用来标识需要读入内存的那个 *Employee* 对象。
4. 搜索 *Employee* 表，查询 *EmployeePOID* 的值与 *Position.EmployeePOID* 的值一致的记录。
5. 读入该 *Employee* 对象（如果存在的话），并将其实例化。
6. 设定 *Employee.position* 属性的值，使之指向 *Position* 对象。

现在让我们来一步一步完成获取单个 Employee 对象的工作：

1. 将 *Employee* 对象读入内存。
2. 自动遍历拥有*(holds)*关系。
3. *Employee. EmployeePOID* 字段的取值被用来标识需要读入内存的那个 Position 对象。
4. 搜索 *Position* 表，查询 *EmployeePOID* 的值与 *Position.EmployeePOID* 的值一致的记录。
5. 读入该 *Position* 对象，并将其实例化。

6. 设定 *Employee.position* 属性的值，使之指向 *Position* 对象。

现在让我们考虑一下如何将对象保存到数据库中。因为关系是自动遍历的，为了保证引用完整性，需要创建一个事务(transaction)。下一步是将针对每个对象的update语句添加到事务中。每个update语句都包括了图 15 中的业务属性和映射的键值。由于关系是通过外键实现的，而这些外键的取值会被更新到数据库中，于是对象的关系也就会被有效的持久化。然后将该事务提交到数据库并执行。

通过这种方式将拥有关系映射到数据库有个麻烦的问题。尽管在对象schema中，该关系的方向是从*Employee*指向*Position*的，其在数据库中却被实现为从Position指向Employee。这么做并无大碍，却也会烦人。在数据库schema中，你可以随便在哪个表中实现外键，这些实现之间没有任何差别，所以从数据库的角度来看，当选择哪种实现都一样的时候，你可以随机选择。某些变更案例(Bennett 1997, Ambler 2001a)表明，存在着一种潜在的需求，要求将拥有(holds)关系变成一对多关系，那么你就有理由选择合适的外键实现策略以满足该需求。比如，现有数据模型就可以支持一个雇员拥有多个职位。但是，如果对象schema已经被仔细考虑过，并且没有进一步的需求促使你以其他方式建模，那么在*Employee*表中实现外键会更清晰。

4.4.2. 一对多映射

现在我们来考察一下图 13 中*Employee*与*Division*之间的*隶属(works in)*关系。这是一个一对多关系——每个雇员隶属于一个部门，而一个部门包含多个雇员。有趣的是，正如你在图 15 中看到的，该关系应该会被自动地从*Employee*遍历到*Division*（这个常常被称之为级联读取cascading read），反之则不行。同样的，也可能存在级联式的保存与删除，这些会在关于引用完整性的讨论中涉及。

当一个雇员信息被读入内存时，将会自动遍历关系以读取其所工作的部门的信息。你不希望同一个部门的信息在内存中存在多份拷贝，比如你有十个雇员隶属于 IT 部门，你会希望这些雇员对象都引用内存中的同一个 IT 部门对象。这意味着你需要为此实现某种策略，一种选择是实现一个缓存来确保某个对象内存中只存在一个实例，或者简单地让 *Division* 类实现一个集合，这个集合用来管理 *Division* 类自己在内存中的实例（相当于一个微型的缓存）。如果应用程序需要将 *Division* 对象读入内存，那么它将正确设置 *Employee.division* 的取值，使之指向一个合适的 *Division* 对象。同时，*Division.addEmployee()*操作会被调用，以便将 *Employee* 对象添加到它的集合属性中去。

保存一对多关系的方式跟保存一对一的方式一样——当对象被保存的时候，它们的主键跟外键都会被保存，于是相应的关系也就自动地被保存了。

本章的每个例子中，采用的外键都指向其他表的主键，比如 *Employee.DivisionPOID* 指向 *Division.DivisionPOID*。但是这并非必须，有时外键

也可以指向另一个表的候选键（alternate key）。比如，如果图 14 中的 *Employee* 表包含了一个 *SocialSecurityNumber* 字段，那么该字段就是该表的一个候选键（假设所有的雇员都是美国人）。如果存在这种情况，你可以选择使用 *Position.SocialSecurityNumber* 来替换 *Position.EmployeePOID* 字段。

4.4.3. 多对多映射

为了实现多对多关系，你需要（借助于）关联表的概念，该数据实体唯一的用途就是维护关系数据库中两张表或者多张表之间的关系。在图 13 中 *Employee* 与 *Task* 之间存在多对多关系。在图 14 的数据库 schema 中，需要引入关联表 *EmployeeTask* 来实现 *Employee* 与 *Task* 之间的多对多关系。关系数据库中，关联表中包含的属性通常是该关系所涉及的表的键值的组合，本例中便是 *EmployeePOID* 与 *TaskPOID*。一般来说，关联表的名称不是相关表的名称的组合，就是它所表达的关系的名称。本例中我选择了比 *Assigned* 更加贴切的 *EmployeeTask* 作为表名。

注意图 13 中的多重性。如图 14 所示，一个原则是，一旦引入关联表，多重性会发生“互换”。为保持原有关系的整体多重性，在数据库 schema 中关系的“出边”（outside edges）上，多重性总被设置为 1。原有关系指出一个雇员会被分派一个或者多个任务，而一个任务则会被分配给零个或者多个雇员。在数据库 schema 中，尽管采用了关联表来维护该关系，刚刚提到的这个关系依然存在。

假设内存中有一个雇员对象，我们想要列出分配给该雇员的全部任务。应用程序所要执行的步骤包括：

1. 创建一条 SQL Select 语句，连接 *EmployeeTask* 和 Task 这两张表，从中选出满足要求的 EmployeeTask 记录，即 EmployeePOID 的取值等于那位我们正要列出全部任务的雇员。
2. 在数据库中执行该 Select 语句。
3. 将返回的数据记录转化成相应的Task对象。其中要做的部分工作包括检查该Task对象在内存中是否已经存在。如果已经存在，那么就用新的数据刷新旧有对象（这里存在并发问题）
4. 每个 Task 对象都会促使一次 *Employee.addTask()*调用，以便构造 *Task* 对象的集合。

读取所有与某个给定任务相关的雇员信息的过程与此类似。让我们继续从 Employee 对象的角度来看看保存关系的过程需要哪些步骤：

1. 启动一个事务。
2. 为修改过的任务对象生成相应的 Update 语句。
3. 为新创建的任务(task)生成相应的 Insert 语句，向 *Task* 表插入数据。
4. 为新创建的任务生成相应的 Insert 语句，向 *EmployeeTask* 表插入数据。
5. 为被删除的任务生成相应的 Delete 语句，删除 *Task* 表中相应的记录。如果之前已经

单独做过删除对象的动作了，那么这步不是必需的。
6. 为被删除的任务生成相应的 Delete 语句，删除 *EmployeeTask* 表中相应的记录。如果之前已经单独做过删除了，那么这步不是必需的。
7. 为不再属于该雇员(employee)的任务生成相应的 Delete 语句，删除 *EmployeeTask* 表中相应的记录。
8. 运行事务。

关联表的引入使得多对多关系（的映射）变得有趣起来。为了支持这种关系，两个业务类被映射成三个数据表，于是需要多做一些额外的工作。

4.5. 映射有序集合

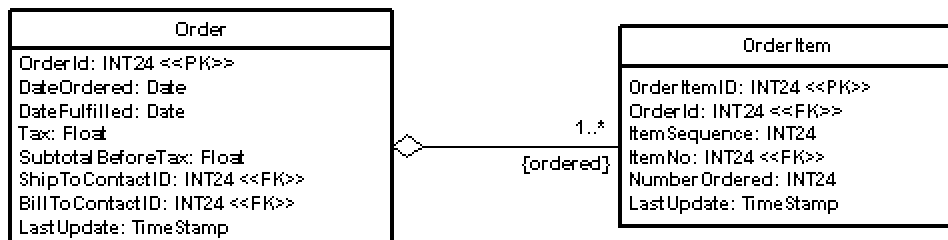
图 2 描述了一种典型的类 *Order* 与类 *OrderItem* 之间带有聚合关联的模型。有意思的地方是位于关系上的{ordered}约束——用户关心订单项出现在订单上的顺序。因此映射到关系数据库的时候需要添加一个额外的字段以记录这个信息。图 2 所示的数据库 schema 便包含了 *OrderItem.ItemSequence* 字段，来持久化这一信息。尽管表面看来这个映射很直观，但还有几个问题需要考虑。当你考虑的基本聚合持久化功能时，这些问题就会变得很明显：

- **按正确的顺序读取数据**。实现这种关系的辅助属性(scaffolding attribute)必须是一个支持按顺序排列的（对元素的）引用（references）的集合，而且在向 Order 添加新的 OrderItem 时，该属性必须能增长。你可以在图 3 中看到我们采用了 Java 集合框架中的 Vector 类来满足这些需求。当你将*订单与订单项*读入内存时，相应的 Vector 对象就必须按正确顺序填充。如果 *OrderItem.ItemSequence* 字段的取值从 1 开始并且每次增加 1 的话，那么你可以简单的使用该字段的取值作为插入位置将订单项插入到集合中。否则，你就必须在提交至数据库的 SQL 语句中使用 ORDER BY 子句来保证结果集中行序的正确性。
- **不要在键中包含序号**。内存中有一个订单对象，它带有五个订单项，并且该订单对象被保存过，数据库中有对应的数据。现在你要在第二个订单项与第三个订单项之间插入一个新的订单项，这样你总共就有了六个订单项。对于图 2 中的数据库 schema，你不得不为那些位于插入位置之后的订单项重新逐一编号，并把他们重新写入数据库，即使这些订单项除了序号以外什么都没有改变。因为序号是 *OrderItem* 表的主键的一部分，如果其他没有出现在图 2 中的表通过包含有 *ItemSequence* 的外键引用了 *OrderItem* 的某些行，重新编号的操作就会造成问题。更好的方式则是如图 17 所示，使用 *OrderItemID* 作为主键。
- **在重排订单项顺序之后何时更新序号？**一旦你重排了某个订单中的订单项的顺序，比如将第四个订单项移到第二行，你就需要更新数据库中的序号。也许你会在内存中缓存这些改变，直到你决定将整个订单写入数据库，尽管，这将存在因为掉电而没有保存正确顺序的风险。
- **删除订单项之后是否更新序号？**如果你删除了六个订单项中的第五个，你是会去更新新的第五号订单项的序号，还是原封不动？（原有）序号依然有意义——它们的值是 1，2，3，4，6——不过你不能将它们当作在集合中进行插入的位置来使用了，

那样会在第五这个位置留下一个空缺。

- **考虑使序号间隔大于一。**可以采用 10，20，30……这样的序号来代替 1，2，3……这样的序号。这样你就不必在每次重排订单项时都要修改 OrderItem.ItemSequence 字段的取值了，因为如果你要将某个订单项移动到 10 与 20 之间的时候，可以指定序号为 15。每隔一段时间，比如经过几次重排之后，你会发现自己置身于试图将订单号插入 17 与 18 之间的境地，这时你需要全面整理更新序号列的值。更大的间隔有助于避免此类情况(比如 50，100，150……)，不过你永远无法完全避免这一问题。

图 17. 改良后的持久化 Order 与 OrderItem 的数据库 Schema

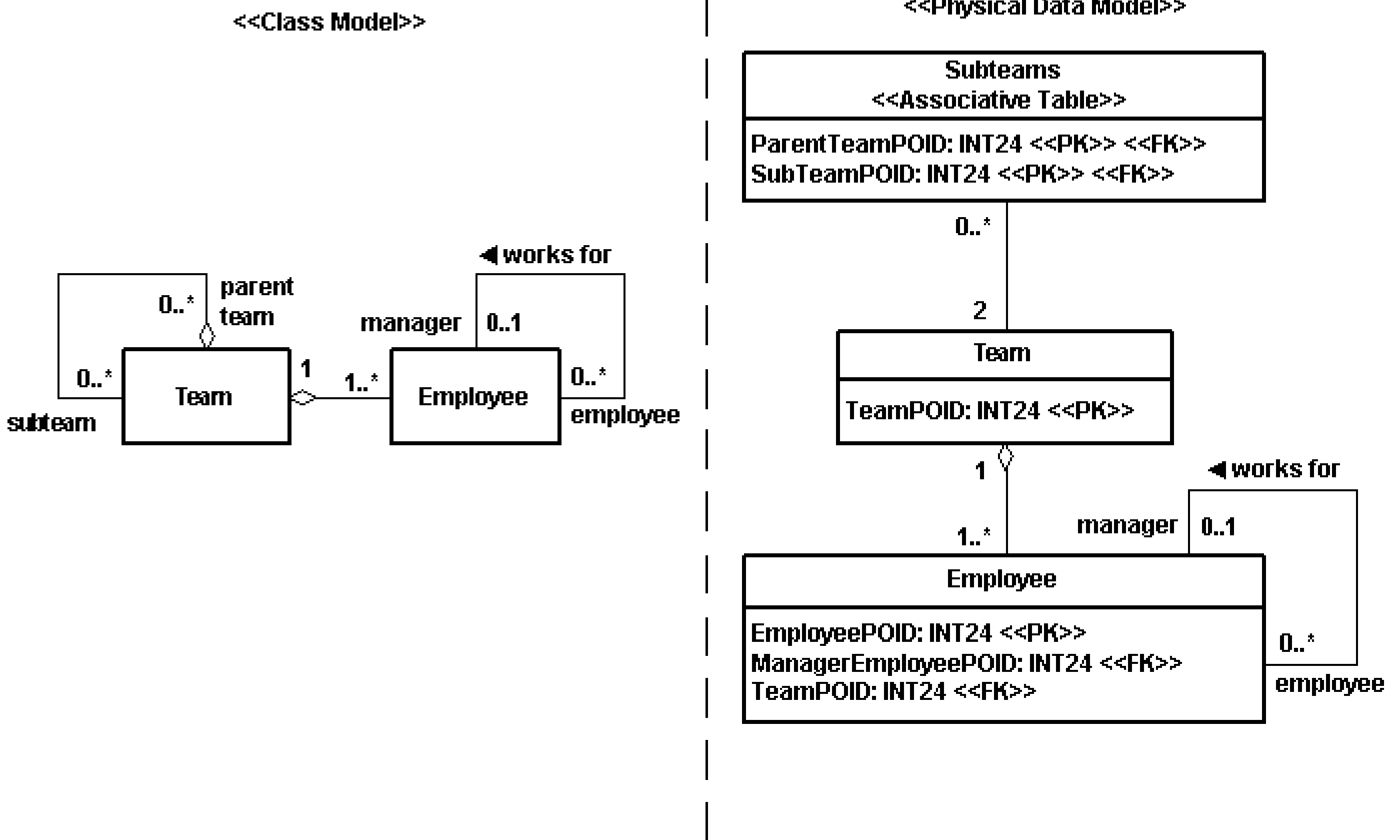


4.6. 映射递归关系

递归关系，也叫自反关系(reflexive relationship)(Reed 2002; Larman 2002)，是指关系的两端指向同一实体(类，数据实体，表，……)。举例来说，图 18 中的*管理(manage)*关系就是递归的，表达了一个雇员可以管理若干其他雇员这样的概念。*Team(团队)*类指向自身的聚合关系也是递归的——一个团队可能是另外的一个或多个团队的一部分。

图 18 描述了一个包含了两个递归关系的类模型，以及对应的映射后的数据模型。简化起见，类模型仅包含了类及其关系，数据模型仅包含了键。跟普通的多对多关系映射一样，多对多递归聚合被映射为Subteams关联表，唯一的区别在于关联表中的两个外键现在指向同一张表。类似的，一对多的*manages*关联的映射方式与普通的一对多关系的映射方式相同，只不过*ManagerEmployeePOID*字段指向的是*Employee*表中的另一行而已，即存放管理者(manager)数据的那一行。

图 18. 映射递归关系



5. 映射类作用域(Class-Scope)属性

有的时候某个类会实现一个为所有该类实例共享的属性(译注：即 Java 中的 static 成员)。图 19 中的 *Customer* 类就实现了一个叫做 *nextCustomerNumber* 的类属性（你可以通过名字下面的下划线知道它是类属性），该属性用来保存下一个客户的编号，以便将之赋予新的客户对象。因为该属性不是每个对象实例一个值，而是整个类所有的实例共享一个值，所以我们需要采用一种不同的方式进行映射。表 2 总结了映射类作用域(class scope)属性的四种基本策略。

图 19. 映射类作用域(class scope)属性

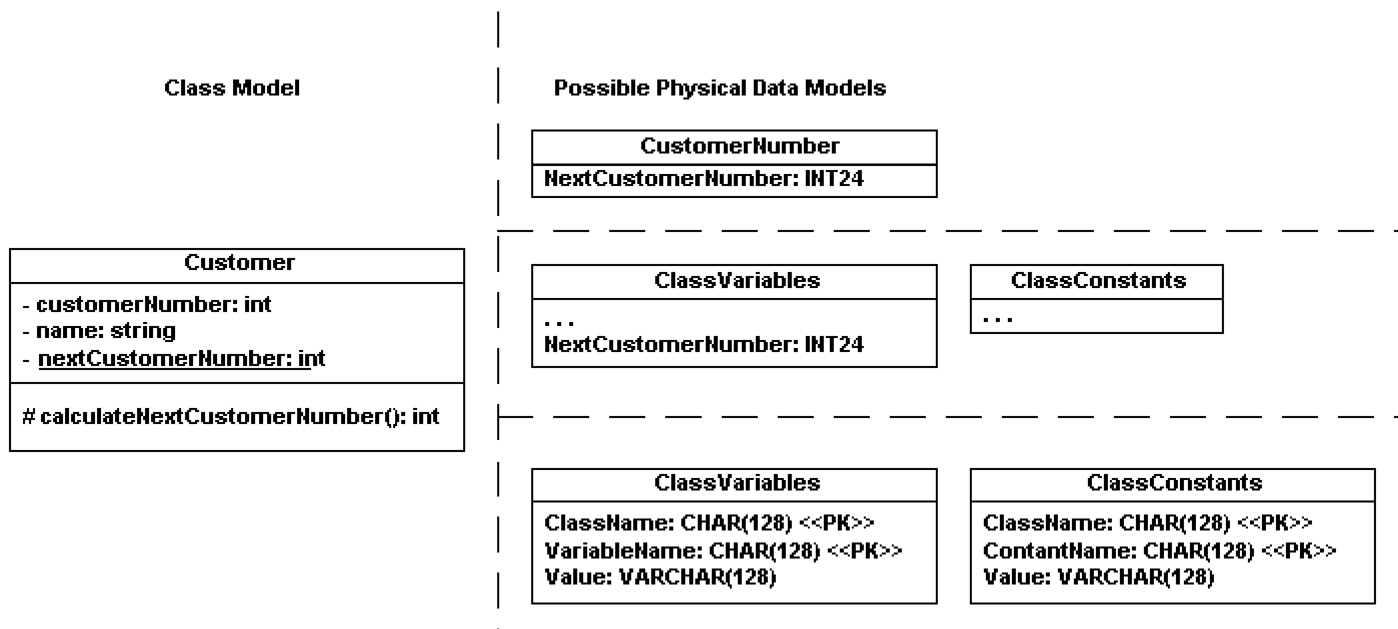


表 2. 类作用域属性的映射策略

策略	举例	优点	缺点
单行单列表	图 19 中的 *CustomerNumber* 表采用此种策略。	简单 访问速度快	可能导致产生大量零碎的表
针对单个类的单行多列表	如果 *Customer* 实现了新的类属性，*CustomerValues* 表可以为每个新的类属性引入一个字段。	简单 访问速度快	同样可能导致产生大量零碎的表，尽管比单行单列的方式要少一些
针对所有类的单行多列表	图 19 中最上边的那个版本的 *ClassVariables* 表。该表为每一个应用程序中的每个类属性都引入一列，因此如果 *Employee* 类有一个 *nextEmployeeNumber* 类属性，那么表中也会有对应一列。	使数据库 schema 中引入表的数目最小	如果很多类需要同时访问类属性数据，可能存在潜在的并发问题。一种解决方案是如图 19 中那样所示引入一个 ClassConstants 表来分离只读的属性跟与可修改的属性。
针对所有类的多行通用 schema	图 19 中最下边的那个 *ClassVariables* 表以及 *ClassConstants* 表。每个系统中的类作用域属性都对应于表中的一行。	使数据库 schema 中引入表的数目最小 减少并发问题（假	需要进行类型转换（比如 *CustomerNumber* 是一个整型数据，但是被保

		设你的数据库支持行锁定）	存成了字符型数据）。 数据库 schema 与你的类名及其类作用域属性名相耦合。 你可以通过使用类似图 11 的更加通用的 schema 来避免这一点。

6. 性能调优

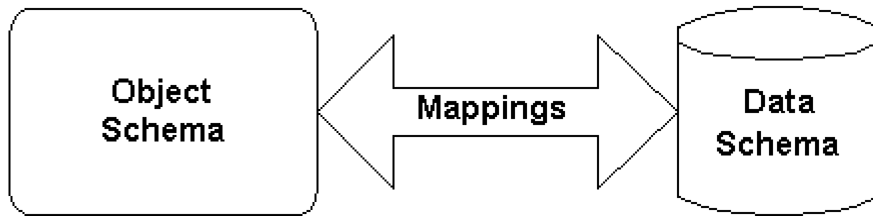
一个敏捷数据库管理员（Agile DBA）对开发团队所能提供的最有价值的服务之一就是性能调优。Shasha 和 Bonnet 写了一本非常精彩的关于数据库性能调优的书《数据库性能调优》（*Database Tuning*）（2003）。对于结构化技术而言，大多数性能调优的工作都是面向数据库的，一般可以分成两类：

1. **数据库性能调优**。这主要**集中在**修改数据库schema上，通常通过对部分schema进行逆范式化处理（denormalize）来实现。其它方法包括修改键字段的类型，例如基于数值型的索引会比基于字符型的更有效率；也可以减少构成组合键的字段数，或者在一个表上引入索引来支持一些常用的连接操作。

2. **数据访问性能调优**。这主要集中在提高访问数据的速度上。常用技术包括，使用存储过程在数据库服务器端处理数据，以减少在网络上传输的结果集；修改 SQL 查询，使之适应于数据库的某些特性；集群数据来应对一般的访问需求；在你的应用程序里面缓存数据来减少访问的次数。

上述这两种优化并不排斥对象技术，不过如图 20 所示，使用它们的复杂度却提高了。一个重要的事情是，记住你的对象 schema也有它自己的结构，因此，修改对象schema会改变从对象schema到数据库schema的映射，进而影响数据库访问代码。例如，假定类*Employee*有一个属性*homePhoneNumber*。一个新的特性要求你实现电话号码的某些特定行为（例如，你的应用程序可以给在家里的人们打电话）。你决定重构*homePhoneNumber*到一个它自己的类，这是一个第三对象范式(注：不同于通常所指的数据库第三范式)的例子，因此你更新你的映射来反映这个变化。这种变化会降低性能，这会促使你去修改那些决定数据访问路径的映射，或者修改数据库schema本身。这暗示了对对象源代码的修改可能导致对数据库schema的修改。有时相反的情况也会发生。这没什么不好的，因为作为一个敏捷的软件开发者，你应当习惯于以渐进的方式进行工作。

图 20. 性能调优的机会



对于性能调优，有两点主要补充你需要了解：映射调优和对象 schema的调优。下面会谈到映射调优。当谈到对象 schema的调优时，常用的重构（Fowler 1999）适用于你对schema所做的大部分修改。然而，一个叫做延迟读取（lazy reading）的技术可以提供很大的帮助。

6.1 优化你的映射

如你所见，在本章中，讨论了不止一种方法来映射对象 schema到数据schema——有四种方法来映射继承结构，两种方式来映射一对一的关系（取决于你在哪边放置外键），四种方法来映射类作用域属性。因为你有不同的映射选择，而每个映射选择又有利有弊，这就给了你机会，让你通过改变映射选择来提高应用程序的数据访问性能。可能你用每个类一张表的方法来映射继承关系，但是发现太慢了，这会促使你用每个类层次结构一张表的方法来重构它。

重要的是要明白，无论你何时更改映射策略，都将要求你修改你的对象 schema 或数据 schema，或者两者都修改。

6.2 延迟读取

我们需要考虑的一个重要的性能问题是，当读取一个对象的时候，是否需要自动地读取它的属性。如果一个属性很大，例如，一个人的照片可能会超过 100k 而其它的属性可能都不到 1k，并且照片很少被访问，那么你可能需要考虑采用一种延迟读取的策略。基本思路是当读取一个对象的时候，并不自动把它的属性通过网络传输，而仅仅是当实际需要这个属性的时候才去获取它。这可以通过一个 getter 方法来实现，这个操作的目的是提供单个属性的值，该方法检查这个属性是否已经被初始化，如果没有，从数据库里面读取。

延迟读取的另一个常见用途是报表。此外，如果你仅仅需要对象数据中一个很小的子集，则作为搜索结果的返回对象集也常常使用延迟读取。

7. 为什么数据 Schema 不应该主导对象 Schema

这个部分的内容已经被重新组织成：为什么数据模型不主导对象模型（反之亦然）具体参见http://www.agiledata.org/essays/drivingForces.html。

8. 实现方式对对象的影响

面向对象技术和关系技术之间的阻抗失配强制你将你的对象schema映射成为相应的数据schema。要实现这种映射，你需要为你的业务对象添加代码，这些代码将对你的应用程序产生影响。对象纯化论者就是利用这种影响，来反对将这两种技术一起使用的。尽管我希望情况有所改观，但事实上，我们目前正将两种技术一起使用，而且极有可能，这种情况将持续很多年。不管我们喜欢与否，都必须接受现实。

我想总结一下映射技术对面向对象技术的影响会非常有价值。在本章和其他章中都有一些这类总结。影响你代码的需求包括：

- 维护shadow信息
- 重构以提高整体性能
- 和遗留数据打交道。和遗留数据库打交道是很普遍的事情，随之而来的经常是重大的数据质量、设计和架构问题。这意味着，你经常需要将你的对象映射到遗留数据库中去，你的对象可能需要实现完成整合与数据清理代码。
- 封装数据库的访问。数据库访问的封装策略决定了你将如何实现你的映射。你所选择的策略会对你的对象构成影响，这种影响可能是在程序中嵌入SQL的代码，也可能是类需要实现持久框架所要求的公共接口。
- 实现并发控制。由于大多数应用程序是多用户的，并且大多数数据库有几个应用程序访问，因此你面临着两个不同进程同时修改相同数据的风险。所以你的对象需要实现并发控制策略来战胜这些挑战。
- 从关系数据库中查找对象。你想要便捷的处理同种类型的对象集合，例如，你或许想在一次操作中列出所有的雇员。
- 实现引用的完整性。在对象和数据库中，有几种实现引用完整性的策略。虽然引用完整性是业务问题，需要在你的业务对象中实现，但实际上，即使不是全部，也还有很多的引用完整性规则是在数据库中实现的。
- 实现安全访问控制。不同的人有不同的信息要访问。这就需要你，在你的对象和数据库内部，实现安全访问控制。
- 实现报表。是让你的业务对象实现基本的报表功能，还是使用单独的直接的访问数据库的报表生成工具，抑或两者结合。
- 实现对象缓存。对象缓存用于提高应用程序的性能，同时确保对象在内存中的唯一性。

9. 模型驱动体系结构（MDA:Model Driven Architecture）的含义

模型驱动体系结构（MDA）(对象管理组2001b)定义了一种建模方法，这种方法将系统功能规格与它在特定技术平台上的实现规格相分离。简而言之，它为以模型的方式表达的结构化规格定义了指导原则。MDA倡导一种方法，即通过辅助的映射标准，或特定平台的个别映射（point mapping），使描述系统功能的同一模型得以在多个平台上实现。同时，MDA也支持那些与不同应用的模型有明显关联的概念，它还支持集成，支持不同应用间的互操作能力，支持系统随着平台技术的发展而不断演化。

虽然MDA是基于统一建模语言（UML）的，而UML还没有正式支持数据模型，但我相信将来那些MDA兼容的CASE工具所支持的最重要的技术中，对象到关系的映射必将占据一席之地。我希望OMG的成员们能够找到方法克服文化上的“阻抗失配”（cultural impedance mismatch），并和数据专家们一起努力，将诸如UML数据建模、对象到关系的映射技术提到日程上来。时间将会证明一切。

10. 映射技术模式化

本章讲述了将对象映射到关系数据库的基础知识，包括一些基本的实现技巧，这些技巧在后续章节将作延伸。如上所述，有若干策略可以将继承结构映射到关系数据库，一旦你了解了这两种技术之间的差异，把对象关系映射到关系数据库是很简单的。上面还分别介绍了映射实例属性和类属性的技巧，它们为你提供了将类属性完整映射到关系数据库的方法。

这章包括了一些方法学的讨论，描述在敏捷软件开发的典型情况，即迭代和增量开发方法中，映射应该如何工作。一个相关概念是：如果你用已有的数据库schema或数据模型来主导对象模型的开发，那么你犯了一个基本的错误。着眼于它们，把它们作为约束，但是如果你能避免，就别让它们对你的设计造成负面影响。

虽然多数作者都喜欢使用平白的语言来叙述这个问题，但是有些作者选择通过模式语言来表达。（参见：www.ambysoft.com/mappingObjects.html列出了大量的有关映射的论文链接）。第一个在这方面卓有成效的是Crossing Chasms模式语言（Brown & Whitenack 1996），而最近的有关于此的著作则是《*企业应用架构模式*》(Fowler et. al. 2003)一书。表 3 以模式的形式，总结了本章列出的要点，尽量采用其他作者建议的名称。

表 3. 映射模式

Pattern	**Description**
“类—表”继承	将继承层次中的每个类分别映射一张单独的表
“具体类—表”继	继承层次中的每个具体类分别映射一张单独的表

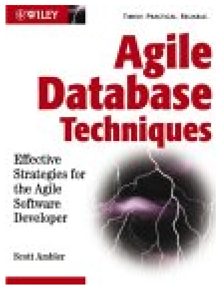
承	
外键映射	关系数据库中，用表中的外键来实现对象之间的关系
标识字段	把对象的主键当成一个属性来维护，这是 Shadow 信息的一个例子
延迟初始化	对象中那些代价高昂的属性，如一张图片，不在你初始化这个对象时读取，而是在第一次访问该属性的时候才将它初始化读入到内存中。
延迟读取	只有在需要某对象的时候，才把它读入到内存中。
遗留数据约束	遗留数据源对你的对象 schema 而言是一种约束，但不能由它们来主导对象 schema 的定义。
映射成类似的类型	在类和对应的表中，使用相近的类型进行映射。如，将一个整型属性映射成一个数值型字段要比将它映射成一个基于字符的字段容易的多。
将简单属性映射成单个字段	最好将对象的属性，如某个定单的合计或者某个雇员的名字，映射到数据库的单个字段中。
基于映射的性能调优	你可以修改对象的 schema、对应数据库的 schema 或两者之间的映射，以提高数据访问的整体性能。
递归关系没什么特别	递归关系的映射和非递归关系的映射方式相同。
用表来表示对象	最好将一个单独的类映射成一张单独的表，但是为了提高性能，你要对设计改进有所准备
将类属性放在单独的表中	引入单独的表来存储类属性
Shadow信息	类需要维持若干属性，以便对自身进行持久化的时候存储数据库键值（比如标识字段）和处理并发的列
单字段代理键	在你的数据库中，你可以采用的最简单的键策略是，给所有的表一个代理键字段，其键值是全局唯一的。
单表继承	将整个继承层次结构的类映射到一张表中
表设计阶段	以对象 schema 为基础，开发你的数据 schema，但要以迭代的方式不断的改进你的设计。
单向键选择	当类 A 到 B 存在一对一的单向关联，要在类 A 相应的表中使用外键来维护这种关联关系。

11. 参考文献和阅读推荐

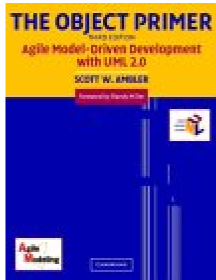
参考文献

在www.ambysoft.com/mappingObjects.html上，我维护了一个链接清单，这些链接指向发布于网络的映射相关的白皮书。

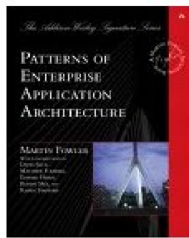
推荐书目：



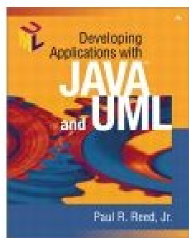
该书描述了,在采用渐进式软件开发过程的项目团队中，开发人员与数据库管理员们为了高效合作所必须掌握的理论与技巧。这些软件过程包括极限编程（XP），Rational 统一过程（RUP），特征驱动开发（FDD），动态系统开发方法（DSDM）或企业统一过程（EUP）等等。本书于 2004 年 3 月获得 Jolt 的生产效率奖。



本书提供了一种针对软件开发的全生命周期的敏捷模型驱动开发方法（AMDD）。本书是少数几本易于理解并且风格一致，内容同时涵盖面向对象与面向数据两种开发技术的书籍之一。书中涵盖的技术包括敏捷建模（AM），全生命周期面向对象测试（FLOOT），30 多个建模技巧，敏捷数据库技术，重构和测试驱动开发（TDD）。如果你想获得必备技能，以敏捷方式来构建执行关键性任务的应用，本书就是为你写的。



该书给出了架构模式的集合，很多模式与持久化相关。强烈推荐本书作为本章内容的补充材料。



这本书与The Object Primer第二版 类似。虽然更多的涉及了java，但还是围绕UML和RUP进行的。本书涵盖了基本的映射概念，包含了更多的源码例子。

12. Let Me Help

12. 关于作者

我和分布在世界各地的客户一起工作，既是教练又是培训讲师，帮助提高他们的IT实践能力。关于我的更完整的信息，诸如我是做什么的，怎么联系我等等，可以在http://www.ambysoft.com/services/找到。

Page last updated on March 8 2005

This site owned by Ambysoft Inc.

Copyright 2002-2005 Scott W. Ambler

译者简介：满江红翻译团队是由各地华人程序员组成的一个网络协作翻译小组，意图将自己工作和学习中遇到的部分好文章和新技术通过网络翻译成中文。本文系列章节由部分成员参与翻译工作，主要参与成员有：莫映，厌倦发呆 ，刘畅，mochow ，郑帅等。有关满江红团队，更多的资料可以访问http://www.redsaga.com/