鲁棒的数据库持久层设计

An AmbySoft Inc. White Paper

Scott W. Ambler

Senior Object-Oriented Consultant

AmbySoft Inc.

Material for this White Paper has been excerpted from Scott W. Ambler's

Building Object Applications That Work

SIGS Books/Cambridge University Press, 1998,

Process Patterns

SIGS Books/Cambridge University Press, August 1998,
and the

Design of a Persistence Layer Series

Software Development, January through April 1998

http://www.ambysoft.com/persistenceLayer.pdf

This Version: August 16, 1998

修改履历：

1998 年 8 月 16 日：

1. 修改了一些语法和拼写的小错误。
2. 扩充了关于持久层需求的部分，在多方面都更加详细。
3. 在文中明显的标注出开发一个持久曾是异常困难的，读者最好是买一个现成的，而不是自己开发。
4. 根据 UML1.1(Rational 版)修改了一些图

致谢

Ben Bovee, Boeing Information Services
Ian Culling, SQL Financials
Jim Comparin, SQL Financials
Klaus Shultz

译注

这篇文章我本来只是想仔细读一读，但是文章太长而自己英文水平又比较差，很担心没有耐心读完。因此我决定动手翻译，希望在翻译的过程中可以更好的理解作者的意图。就这样，每天回家后翻译一段，断断续续大概一个月左右才完成。囿于自己的水平，翻译的过程中发现很多地方难以表达原作真意，只是为了不半途而废，才勉强为之。对于看我这篇译文的人来说，这么做大概属于很不负责任的。但如果读者能因我的初衷而体谅我，并告诉我哪里可以适当修改，那我简直要雀跃了。
原文中的图经过拷贝，都不再是矢量图了，所以我重新用 visio 2002 画了一遍。但因为使用的工具不同，必定与原图有些差异。另外原文图 7 中有个小的错误，误将 DeleteCriteria 写成了 InsertCriteria，这在译文中做了修改。

张笑猛（sum_z@263.net）
2003-1-20

目录

图目录

表目录

在这篇被期望了很久的白皮书中，我给出了一个在面向对象应用中鲁棒的持久层的概要性设计。我已经用不同的语言实现了这个设计的全部或者部分，换而言之，这个设计已经经过了实践的考验。

1 对阅读本文有帮助的一些信息

- 我假设你已经阅读过我的题为 Mapping Objects to Rational Database(Ambler, 1998c)的文章。这篇文章可以从 http://www.ambysoft.com/mappingObjects.html 免费下载。
- 通篇文章我都是使用的统一建模语言（UML）1.1 版（Rational, 1997）来表达我的模型。
- 假设所有的属性都有访问子，也就是 getter 和 setter 方法。
- 所有的属性都是私有的。
- 在我说“类 X 的实例”的时候，实际上隐含的意思是“类 X 或者其任意一个子类的实例”。
- 我并没有给出持久层的代码（我也不打算发布这样的代码），同时我也并未在设计中涉及特定的语言规范的问题。但是在本文的最后，我会讨论一些关于实现方面的问题。

2 持久层的类型

在业务类中硬编码 SQL，导致代码难于维护和扩展

我希望从时下持久化方面比较流行的方法开始我们的讨论。图 1 表达了最常用的，但是也最不爽的持久化方法。在这个方法中，SQL 代码到处出现在你的类代码中。这样的好处是写代码效率很高，对于小型应用程序或者原型，这样是可行的。缺点是直接耦合了你的业务类与关系数据库结构（*译注：结构是我对 schema 的翻译，因为大多数情况下我们说“数据库结构”。下文中凡是出现“数据结构”的情况，都指是 data schema，而不是 data structure*），这意味着任何小的改变（例如对某一列重命名或者移植到另外一种数据库）都导致原代码级的修改。

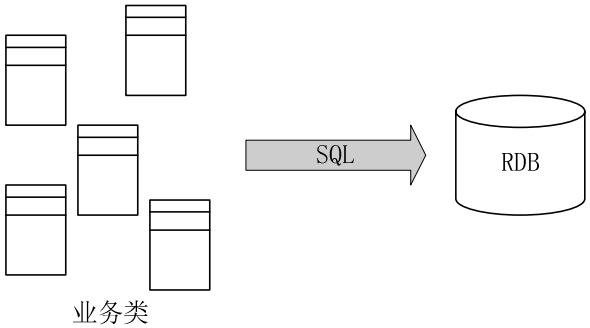

图 1 在业务类中硬编码 SQL

图 2 中的方法稍微好一些，在这种方法中，业务类的 SQL 语句被封装在了一个或者多个“数据类”中。再强调一下，这种方法也只适合原型或者少于 40 或者 50 个业务类的小系统。但是在对数据库进行一点儿改动后，也仍然需要修改和重新编译（数据类）。这种方法的例子包括开发存储过程（用来代替图 2 中的数据类）以及微软的 ActiveX Data Object（ADO）策略。对于这种方法来说，最大的好处莫过于你至少已经将处理交互的部分封装到了单独的数据类中。

在存储过程或者单独的数据类中硬编码 SQL，只算是稍有进步。

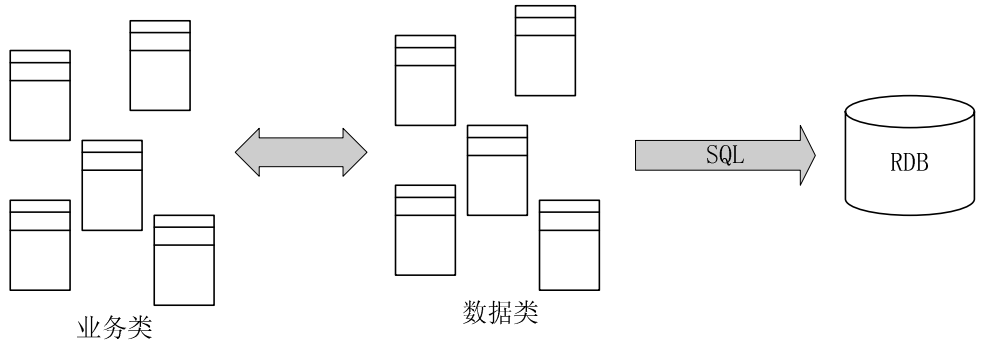

图 2 创建对应业务类的数据类

图 3 中的方法正是本文中所要阐述的，将对象映像到某种持久机制（在这里是关系数据库）并且对关系数据库结构的简单改动并不影响你的面向对象代码的一个鲁棒的持久层。这种方法的好处是你的应用程序开发者不需要了解关系数据库结构，事实上，他们甚至不需要知道对象是保存在关系数据库中。这种方法允许你组织开发大规模的针对关键业务的应用程序。缺点是对你的应用程序有些性能上的影响，尽管如果你做的好的话，影响将很小，但是还是有影响。

图 3 鲁棒的持久层

为了更好的理解这个方法，你必须首先理解 Class-Type 体系结构。

3 Class-Type 体系结构

图 4 展示了程序员在编写应用程序时应该遵循的 Class-Type 体系结构（Ambler, 1998a; Ambler 1998b）。Class-Type 体系结构基于 Layer 模式（Buschmann, Meunier, Rohnert,

Sommerlad, Stal, 1996），基本的概念就是某一层中的一个类，只能与同层的或者相邻层中的类进行交互。通过按照这种方法将代码分层，你就可以更容易的维护以及增强你的代码，因为他们之间的耦合已经极大的降低了。

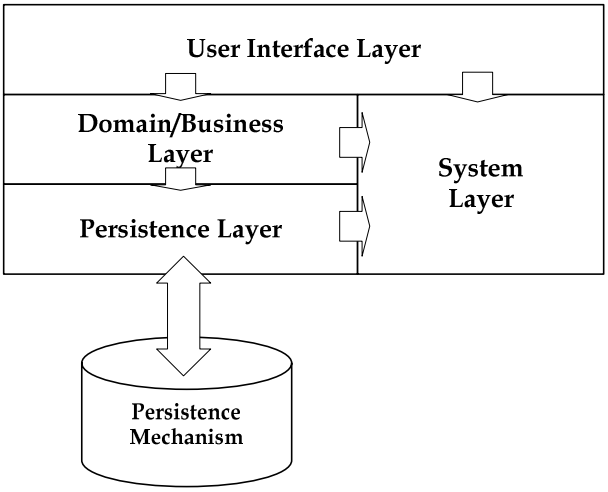

图 4 Class-Type 体系结构

在图 4 中，用户直接与应用程序的用户接口层进行交互。用户接口层主要实现了界面以及报表等。用户接口层的类可以发送消息给业务层和系统层。业务层实现了业务类，例如对于电信公司来说，业务层可能包括客户类和电话呼叫类，系统层实现了提供访问系统功能例如打印和电子邮件的类。业务类可以发送消息给系统层和持久层。持久层封装了存储对象到持久机制的行为，例如对象数据库，文件和关系数据库。

通过遵从 Class-Type 体系结构，你的应用程序会因为减少了内部的耦合而戏剧性的提高鲁棒性。图 4 展示了如果用户接口层要获得信息，必须与业务层的对象交互，然后再通过业务层对象从持久层获得存储在持久机制中的对象。这是 Class-Type 体系结构的重要特点—通过禁止用户层直接访问存在于持久机制的信息，你可以有效的将用户层与持久结构解耦。这也暗示你可以改变对象的存储方式，例如重新组织关系数据库的表或者从某个厂家的数据库产品移植到另外厂家的产品，而不需要重新编写你的界面和报表。

用户接口类不应该直接访问持久机制

通过将程序的业务逻辑封装到业务类中而不是用户接口类中，你可以在多处使用这些业务逻辑。例如，假设你通过一个业务类 Invoice 的实例开发一个界面来显示总产量，在报表中也做同样的事。如果计算总量的逻辑发生了变化，比如增加了复杂的折扣逻辑，这时候你只需要更新 Invoice 内部的代码就可以同时对界面和报表进行修正。而如果你是在用户接口中实现的总量计算逻辑，你就不得不修改两个地方。

业务对象不应该直接访问持久机制

就象你不希望允许用户接口类直接访问持久机制中的信息一样，你也不希望业务类这样做。我们将在下一节中看到一个好的持久层可以保护你的应用程序不受持久机制变化的影响。如果一个数据库管理员决定重新组织持久机制的结构，你不用重新写代码来反映这些变化。

Class-Type 体系结构与硬件/网络结构无关

一个需要理解的重要问题是，Class-Type 体系结构是与你的硬件/网络结构完全没有关系的。表 1 展示了不同的 Class type 是如何在常见的硬件/网络结构中实现的。例如，可以看到在客户机-服务器模式的瘦客户机方

式中，用户接口类和系统类在客户端实现，而业务类、持久类和系统类在服务器实现。因为系统类包装了对网络通信协议的访问，你要保证某些系统类在客户机和服务器上都被实现。

Class Type	独立程序	瘦客户机	胖客户机	多层结构	分布式对象
用户接口	客户机	客户机	客户机	客户机	客户机
业务层	客户机	服务器	客户机	应用服务器	无所谓
持久层	客户机	服务器	服务器	数据库服务器	无所谓
系统层	客户机	所有机器	所有机器	所有机器	所有机器

表格 1 不同硬件/网络结构中 Class type 的部署策略

4 持久层的需求

我一直坚信在开发软件之前应该首先定义需求。这里罗列的需求反映了这些年我在构建和使用持久层方面的经验。我从 1991 年开始从事对象典范（paradigm）方面的工作，从那时起，我使用 C++、Smalltalk 和 Java 开发了金融、外包、军事和电信工业的许多系统。这些系统有的很小，只需要一个人，有的则需要几百个开发者。有的是简单的事务处理，有的则牵扯复杂的领域。简要的说，这些需求反映了我在不同领域项目中的经验。

一个持久层封装了使对象持久化的行为，或者说从（或者向）永久存储中读取、写入、删除对象。一个鲁棒的持久层应该支持：

1 **多种持久机制。**一个持久机制是一种可以永久保存对象使其可以在以后被更新、获取和删除的技术。可能的持久机制包括文本文件、关系数据库、对象-关系数据库、层次数据库（hierarchical database）、网络数据库和对象数据库。在本文中，主要讨论集中在针对关系数据库的持久层方面，另外，通过使用微软的 OLE-DB 技术也可以支持多种持久机制。

2 **对持久机制进行完整的封装。**理想情况下，只需要发送 save、delete、retrieve 消息给一个对象，就可以保存、删除或者获取它，持久层完成其它的工作。进一步讲，除了对已知异常的处理，不必写任何其它关于持久层的代码。

3 **基于条件的多对象操作。**因为一次返回多个对象是很常见的，例如一个报表或者一次用户定制的搜索，一个鲁棒的持久层必须可以支持根据一定条件同时返回多个对象的情况。对于根据一定条件删除多个对象也同样应该支持。

4 **基于关联的多对象操作。**就象在关系数据库中可以进行关联的删除、获取和保存一样，这些操作也应该可以针对对象。一个最好的例子就是一个 Invoice 对象拥有多个 LineItem 对象实例，在获取、删除或者保存一个 Invoice 对象的时候，应该自动的获取、删除或者保存它的 LineItem 对象。

5 **事务。**与需求 3 相关的是支持事务，也就是针对多个对象的一组操作。一个事务应该由保存、获取和/或删除的任意组合组成。事务可以是简单的，一种“要么全部要么没有”的方式，操作必须全部成功或者全部回滚（取消）。或者也可以是嵌套的，在外层的事务失败的时候，嵌套的事务仍然提交而不回滚。事务可以是短期的，只运行千分之一秒，或者是长期的，花费几小时、几天、几周甚至几个月。

6 **扩展性。**你可以增加新的类或者属性，并且能容易的变更持久机制（例如从一个厂商的产品移植到另一个产品，或者是升级）。换句话说，持久层必须可以足够灵活以允许应用程序开发者和持久机制管理员做他们需要做的事。

7 **对象标识符。**对象标识符（Ambler, 1998c），或者缩写为 OID，是一个属性，通常是一个编号，用来唯一标识一个对象。OID 是关系理论中在一个表中唯一标识一行的列—“键（key）”的等价物。OID 可以通过 GUID 和 UUID 等方法实现。

8 **组合键。**传统的数据库结构经常使用由两列或多列构成的组合键来唯一标识一行。尽管这样使数据库结构和持久层都变得复杂（Ambler, 1998c），但不幸的是现实即是如此，我们不得不与这些旧的设计打交道。

9 **多键。**传统的数据库结构中，一个表中经常包括多个键（表可以有一个主键和多个次键）。持久层应该利用这些额外的键带来的性能方面的好处。

10 **游标。**一个支持从一个命令获取多个对象的持久层，也应该支持获取游标。这是一个效率问题：你是否希望用户一次只从持久机制的百万个 person 对象中获得一个呢？当然不。游标就是关系世界中这样的一个有趣概念。一个游标是一个到持久机制的逻辑连接，从这个连接，可以可控的获取对象，通常一次多个。这样常常比一次返回上百个甚至上千个对象要有效率，因为用户可能并不立即需要所有的这些对象（可能通过一个列表翻滚）。

11 **评估结果集大小。**在执行一个多对象操作的时候，你希望先确定有多少持久对象会被这个操作影响。例如，一个用户可能输入一个蹩脚的可能导致几千条结果的查询条件。你应该希望在执行操作前告诉他以使用户决定是否重新定义条件。

12 **代理。**对游标的一个补充就是“代理”。代理代表另外一个对象，但是又不会导致与被代表的对象同样的开销。代理只包括足以让计算机和用户标识它的信息，而没有其它的内容。例如，一个 person 对象的代理包含它的 OID，因此应用程序可以标识它，也包含姓、名以便用户可以分辨出这个代理代表的是谁。代理通常在用户在一个显示出来的结果集列表中选择一条或者两条的时候使用。在用户从列表中选择了代理对象的时候，真正的，比代理对象大的对象被自动的从持久机制中获取出来。例如，完整的 person 对象可能包括地址和个人照片。通过使用代理，你不必在网络上传递每个 person 对象的所有信息，而只传递那些将被实际用到的。这也被称为 lazy read。

13 **记录。**目前市面上大量的报表工具都是以数据库记录作为输入，而不是对象。如果你所在的组织正在一个面向对象的应用程序中使用这样的工具进行报表处理，你的持久层应该支持在 retrieve 请求后返回记录的能力。这样可以避免数据库记录转换成对象后再转换回记录产生的开销。

14 **多种体系结构。**当计算从集中的主机迁移到两层的客户机/服务器结构再到多层结构再到分布式对象的时候，持久层应该可以支持这些变化。要点是你必须假设在某些时候持久层可能存在于一个可能非常复杂的环境中。

15 **不同数据库版本和厂商。**你会升级你的数据库或者移植到其它的数据库。持久层应该支持在不对应用程序造成影响的前提下，简单的变更持久机制。因此，持久层应该支持多种数据库版本和厂商。

16 **多连接。**多数组织使用多种来自不同厂商的数据库，而这些数据库又会被同一个对象应用程序访问。这等于说要求持久层应该可以对每种持久机制都提供多个、同时的连接。

即使一些象从某个持久机制拷贝一个对象到另外一个这样简单的事情，比如从一个中心数据库拷贝到本地数据库，都至少同时需要两个连接，每个数据库一个。

17 **连接池。**持久层应该可以共享数据库的连接，尤其是如果将被部署为应用服务器的一部分的时候。在这种情况下，几百个甚至几千个客户会连接到一个应用服务器。因为每个连接都需要内存并且并非每个客户都需要连续访问一个固定的持久机制，所以在持久层维护一个连接池，并且在所有客户当中共享连接是有必要的。

18 **专用(native)和非专用驱动。**数据库通常提供多种访问方式，一个好的持久层应该支持这些常用的方式。连接方式包括使用 ODBC、JDBC 和由数据库厂商或者第三方提供的专用驱动等。

19 **语言绑定。**你的软件需要能够访问你的持久层，因此持久层需要能够支持你所选择的开发语言。

20 **映射的灵活性。**因为映射类层次结构有三种主要的方式（Ambler, 1998c），所以持久层应该可以支持你所选择的不同映射方式。

21 **生成数据库结构。**如果你的 CASE 工具不支持从对象结构产生数据库结构并且也不支持从数据库结构产生对象结构，持久层就应该可以做到。尽管我已经指出（Ambler, 1998c）了这样做的问题，仍然有很多开发者选择了从对象/数据库结构自动生成数据库/对象结构。

22 **关系遍历。**持久层应该支持在对象之间遍历关系，也包括自动的更新/创建/删除关系（包括聚集关系）。

23 **SQL 查询。**在面向对象的代码中写 SQL 查询是对封装原则的公然违背，这样会直接耦合应用程序与数据库结构。然而，为了性能的原因，有时候需要这样做。在代码中的硬编码 SQL 应该是异常情况，而不是正常情况，并且在发生这样的异常前，应该仔细权衡。不管怎样，持久层还是需要支持这种情况。

24 **调用存储过程。**尽管本文描述了如何避免使用难于移植的存储过程的方法，但是你会发现在现实中偶尔还是要用到它。因此，持久层必须支持调用存储过程。

25 **缓冲区。**一个提高性能的方法就是缓冲对象。持久层应该实现一个对象缓冲区或者能够访问一个外部的缓冲区。

26 **锁。**应该可以在访问数据的时候指定使用悲观的锁或者乐观的锁。这些锁类型将在本文的后面详细讨论。

27 **支持管理工具。**多数持久层，至少鲁棒的持久层，是运行在元数据上的。元数据需要被维护，理想状态下，用一个管理应用程序维护。

28 **运行时刻配置。**应该可以在运行时刻修改持久方法的某些特性，例如游标大小，持久机制的名字/位置等。

持久层应该让应用程序开发者集中精力于他们所擅长的—开发应用程序，而不用关心他们的对象是如何存储的。进一步说，持久层应该也允许数据库管理员（DBA）做他们最擅长的—管理数据库，而不用担心在现有的应用程序中引入 bug。对于一个好的持久层，DBA 应该可以移动表，重命名表，重命名列以及重新组织表而不影响应用程序对他们的访问。不可能？打赌吗，我的经验是构造满足这些需求的持久层是完全可能的，实际上，设计就在下面。

5 持久层设计

在这一节中，我将呈现给读者一个鲁棒的持久层。在后续的章节里，我会讨论关于这个设计在实现方面的问题。

> **警告！想自己做很危险！危险！**
> 构建一个持久层惊人的困难。尽管我对很多困难的问题都给出了答案，但是如果要做的话，还需要很多工作。进一步说，一旦你的持久层完成了，你还需要维护和支持它。这是一个操作的可行性问题，我在Justify stage过程模式（Ambler, 1998b）中详细的讨论过。
> 我在*http://www.ambysoft.com/mappingObjects.html*提供了一些持久层厂商的连接，所以你应该先买一两个看看。

5.1 设计概要

图 5 展示了一个鲁棒的持久层的高层设计（Ambler, 1998b），表 2 则描述了图中的每个类。这个设计吸引人的地方是应用程序开发者只需要知道下面几个类就可以将他们的对象持久化：PersistentObject、PersistentCriteria 类及其子类、PersistentTransaction 和 Cursor。其它的类并不会由应用程序代码直接访问，但是需要开发和维护它们以支持这些“public”的类。

类	描述
ClassMap	一组类，封装了将类映射到关系数据库的行为。
Cursor	这个类封装了数据库中游标的概念。
PersistenceBroker	维护到诸如数据库或者文本文件等持久机制的连接，并且处理对象应用程序与持久机制之间的通信。
PersistentCriteria	这个类层次封装了根据指定条件进行获取、更新、删除等所需的行为。
PersistenceMechanism	一个封装了对文本文件、关系数据库、对象数据库等的访问方法的类层次。对关系数据库，这个树封装了复杂的类库，例如微软的 ODBC 或者 Java 的 JDBC，这样可以保护你的组织不受这些类库改变的困扰。
PersistentObject	这个类封装了使单个实例持久化的行为，所有需要持久化的业务对象都从这里派生出来。
PersistentTransaction	这个类封装了支持持久机制的简单以及嵌套事务所需的行为。
SqlStatement	这个类层次知道如何根据 ClassMap 对象构造 insert、update、delete 和 select 语句。

表格 2 持久层的类和类层次

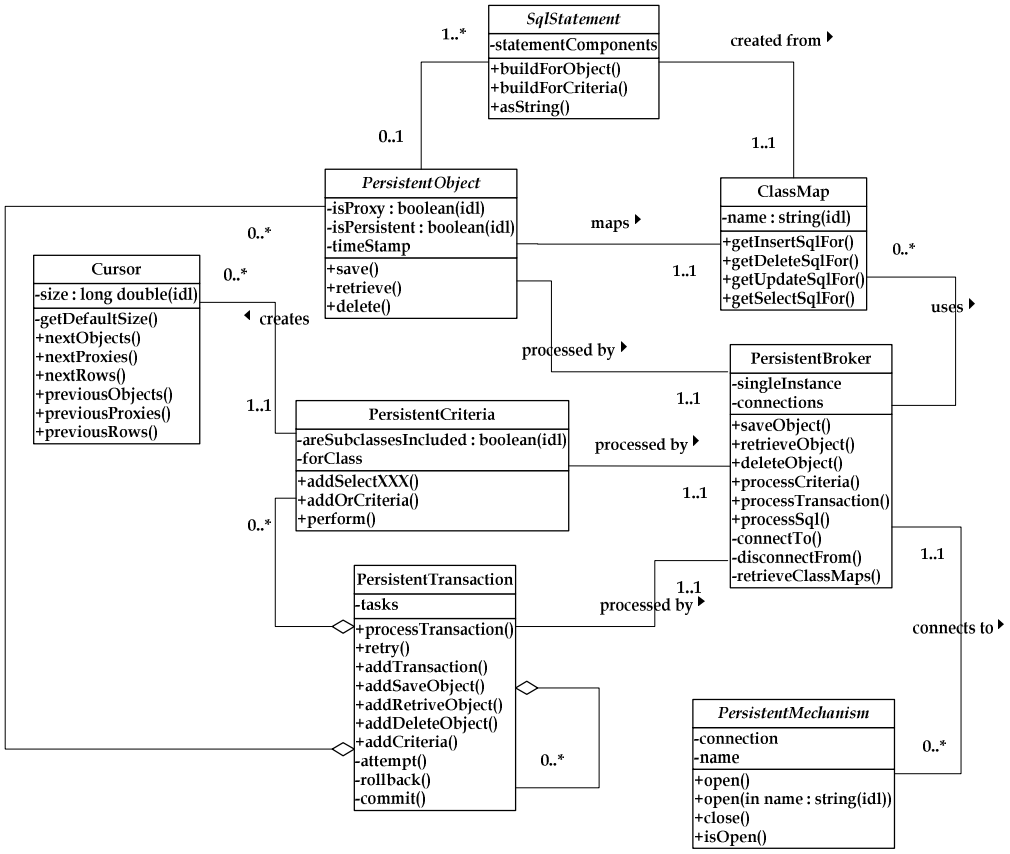

图 5 持久层概要设计

图 5 种的类代表了一种内聚的概念，换句话说，每个类只做一件事，并且做得很好。这是一个好的设计的基础。PersistentObject 封装了使单个对象持久化的行为而 PersistentCriteria 类层次封装了需要与一组持久对象协同工作的行为。进一步说，我愿意指出本文中的设计代表了我使用 Java、C++和 Smalltalk 为不同行业的不同问题域构造持久层的经验。这个设计可以工作并经由广泛的应用程序验证过。

5.1.1 PersistentObject 类

图 6 展示了两个类的设计，PersistentObject 和 OID。PersistentObject 封装了使单个对象持久化的行为，问题/业务域中所有类也都从它派生出来。例如业务类 Customer 将或者直接或者间接的继承自 PersistentObject。OID 类使用 HIGH/LOW 方法封装了对象 ID（在 CORBA 社区中被称为持久 ID）相关的行为。我会将 HIGH/LOW OID 方法的细节增加到 1998 年秋季的白皮书中，现在只管相信下面的 OID 类可以很好的工作即可。

图 6 PersistentObject 和 OID 类的设计

就像你看到的，PersistentObject 类非常简单，它的三个属性 isProxy、isPersistent 和 timeStamp 分别指示一个对象是否是一个代理、是否是从持久机制获取的以及由持久机制记录的最后一次被应用程序访问的时间。代理对象只包括系统和用户标识一个对象所需的最少信息，因为它们比完整的对象小，所以可以减少网络流量。当需要真正的对象的时候，发送给代理 retrieve()消息，可以刷新对象的所有属性。代理在用户只对被获取的对象的一个小的子集感兴趣的时候使用，通常是搜索的结果或者对象的简单列表。属性 isPersistent 的重要性在于一个对象需要知道它已经存在于持久机制中还是新创建的，这将决定在保存对象时生成 insert 还是 update 语句。timeStamp 属性被用于支持持久机制中的乐观锁。当对象被读到内存中时，它在持久机制中的 timeStamp 被更新。当对象随后被写回，timeStamp 第一次被读出来并与初始值比较—如果 timeStamp 的值发生了变化，说明还有另一个工作于这个对象的用户，两个用户的工作发生了冲突，需要解决（典型办法是向用户显示一条消息）。

PersistentObject 实现了三个方法—save()、delete()和 retrieve()，它们是可以将对象持久化的消息。也就是说，应用程序开发者不需要知道任何关于持久策略的知识就可以将对象持久化，只需要向对象发送这些消息，而对象自己完成其它的事情。这就是封装。

构建一个持久层非常困难，最好买一个。

Retrieve()也许是这三个方法中最有意思的，因为如果这个对象的 OID 还没有被设定，就需要基于对象当前的属性创建一个 RetrieveCriteria（见下文）的实例。因为一个 RetrieveCriteria 会返回零个、一个或者多个对象，所以针对这三种情况进行需要不同的处理策略。当返回零个对象时，你知道当前对象还没有被持久化（因此要分配给它一个新的 OID）。当返回一个对象的时候，你希望将返回对象的值交换到当前对象中（在 Smalltalk 中，调用 become:方法）。当返回两个或以上的对象时，你要确定哪个是你问题域中所需要的，例如你可以决定这是一个错误，并进行相应的处理。

PersistentObject 潜在的维护着一个与 OID 实例之间的关系，这个工作在对象 ID 被用做对象的唯一键时被完成。这是可选的，因为并非总是可以选择对象 ID 作为键，更常见的是必须将对象映射到一个传统结构中。将对象映射到传统结构中的需要是面向对象开发世界中的不幸现实。关于映射是如何实现的，我会在后面讨论。不论怎样，如果你可以完全控制持久机制，让 PersistentObject 在对象创建的时候自动分配 ID 给你的对象是非常简单的。

5.1.2 PersistentCriteria 类层次

尽管 PersistentObject 封装了使单个对象持久化的行为，但是还不够，因为我们常常有很多持久对象。这也就是 PersistentCriteria 类层次在图 7 中出现的地方，它支持一次保存、获取、删除多个对象。

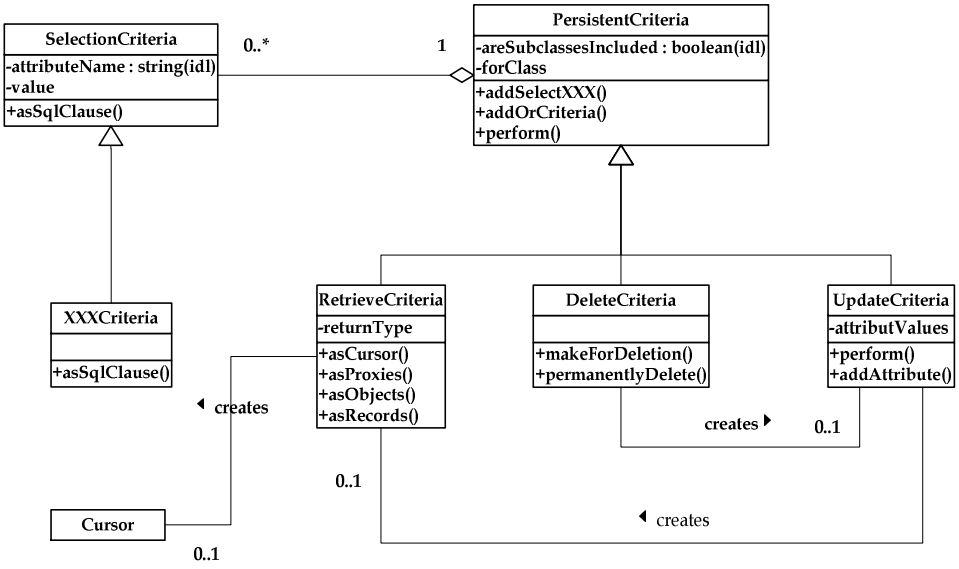

图 7 PersistentCriteria 类层次

PersistentCriteria 是一个抽象类，它捕获子类的公共行为但是不会直接被实例化。这个类允许你定义用以限定对象范围的选择条件。PersistentCriteria 的 addSelectXXX()方法实际上是一组带有两个参数的方法，一个参数是类的属性，另一个是值，返回对应的 SelectionCriteria 的子类实例。SelectionCriteria 类层次封装了用来比较指定值与某个属性的行为。每种比较类型（等于、大于、小于、小于等于、大于等于）一个子类。例如方法 addSelectGreaterThan()创建一个 GreaterThanCriteria 实例而 addSelectEqualTo()创建一个 EqualToCriteria 实例。

我是否已经提到过构建一个持久层是非常困难的？

PersistentObject 类的 forClass 属性指示了被处理对象的类型，例如可能是 Employee 或者 Invoice 对象。isSubclassesIncluded 属性指示了是否条件也应用于 forClass 的子类。这两个属性有效的支持了继承中的多态性。这两个属性的组合以及 addSelectXXX()方法使你可以对名字开头为'J'并且出生日在 1966 年 6 月 14 日到 1967 年 8 月 14 日之间的 Person 对象及其子类的实例进行操作。

因为我们希望可以不只是获取对象，所以 RetrieveCriteria 类支持获取零个或多个对象、代理对象、记录或者一个游标。支持代理是为了减少网络流量，支持记录是因为很多报表类库用记录（而不是对象）作为参数，支持游标可以使你通过只处理结果集的小的子集来提高程序的响应速度。游标类会在后面讨论。

DeleteCriteria 支持一次删除多个对象。这个鲁棒的类既支持我所喜欢的将对象标识为删除，也可以真的删除他们（为存档而清空数据库）。为了标识对象为删除，DeleteCriteria 的实例创建一个 UpdateCriteria 的实例，简单的更新对应表的 deletionDateTime 或者 isDeleted 字段。

UpdateCriteria 类被用来同时更新一组对象的一个或者多个属性。Perform()方法首先通过 RetrieveCriteria 的实例获得要被更新的对象，然后循环将它们的属性赋予新值，最后发送一个 save()消息给每个对象来将对象写回持久机制。你需要获取对象以便能通过 setter 方法来更新它们的属性，setter()方法可以确保在设定新值的时候遵循业务规则。记住，对象封装

了通常无法反映在数据库中的业务规则，因此，你不能简单的产生一个 SQL 语句来一次更新所有对象。

持久条件对象的典型生命周期是首先为其定义零个或多个选择条件，然后通过 perform() 方法来执行（它通过将自己提交给单一实例（Singleton）的 PersistenceBroker 来执行）。SelectionCriteria 的实例通过使用“与”逻辑与其它实例关联。为了支持“或”逻辑，orCriteria() 方法以一个 PersitentCriteria 的实例作为参数并有效的将两个条件连在一起。就象你能猜到的，这样可以构造出更复杂的条件对象。

这个类层次的优点在于它允许应用程序开发者在对存储在持久机制中的对象进行获取、删除和更新的时候，不必了解有关库结构的任何知识。记住，SelectionCriteria 类处理对象的属性，而不是表的字段。这允许应用程序开发者构建不与数据库结构耦合的搜索界面、列表和报表，并能在不知道数据库设计的前提下将持久机制中的信息归档。再说一次，我们的持久层支持对持久机制结构的完全封装。

5.1.3 游标类

图 8 展示了封装数据库游标的游标类的设计。游标允许你从持久机制中获取信息的子集。它的重要性在于通过前面讲到的 RetrieveCriteria 类完成的一个获取动作，可能导致在网络上传输几百或者几千个对象，而通过游标类，你可以获取结果集的一小部分。游标对象允许你在结果集中向前或者向后遍历（大部分数据库支持向前遍历但不一定支持向后遍历，主要是因为服务器端缓冲区的限制），这使用户滚动对象列表变得非常容易。游标类还支持数据库记录、代理对象以及普通的全尺寸对象。

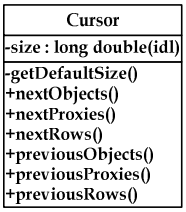

图 8 游标类

游标用一个 size 属性指定一次最多获取多少记录、对象或者代理对象，这个属性一般在 1 和 50 之间。就象你期望的，类的静态方法 getDefaultSize()返回默认的游标大小，我一般设为 1。注意我是对游标默认大小使用了 getter 方法，而将其定义为一个常量（对于 Java 程序员来说是 static final）。通过使用 getter 方法获得这个常量，保留了一个可以计算游标大小值的机会，而不是硬编码为一个常量。我主张信息隐藏的原则既适合于常量也适合于变量，因此我通过对一个常量使用 getter 方法来使我的代码鲁棒性更强。

5.1.4 PersistentTransaction 类

第四个也是最后一个与应用程序直接相关的类（其它的为 PersistentObject、PersistentCriteria 和 Cursor）就是图 9 中的 PersistentTransaction。这个类的实例是由对一些单独的对象进行保存、删除和获取操作构成，也包括 PersistentCriteria 的实例和其它的

PersistentTransaction 对象。

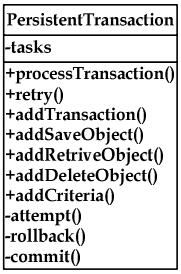

图 9 PersistentTransaction 类

事务的典型生命周期是：被创建，添加一系列操作任务，接受 processTransaction()消息，然后或者提交事务，或者回滚事务，或者重试。你只能在 processTransaction()方法返回成功后，才能提交事务。否则要么回滚，要么当持久机制上的锁已经移除后，重试事务。提交和回滚事务的能力是非常重要的，因为事务是原子的，或者成功或者失败，你只能将所有任务完全回滚或者完成所有任务后完全提交。

任务被按照它们加入到 PersistentTransaction 中的顺序依次处理。如果一个任务失败了，可能是因为它不能删除一个对象，那么，处理过程就中断在这个任务上，同时 processTransaction()方法返回失败。

当一个 PersistentTransaction 实例通过调用 addTransaction()方法被添加到另外一个事务中，这种情况被看作是嵌套的事务。子事务甚至在父事务失败的时候也可以成功，被提交。在嵌套事务中执行到一个子事务的时候，如果它成功了，则在执行任务列表中下一个任务前它将被自动提交，否则，如果它失败了，父事务中断在这个子事务上，并返回失败。

这个类的一个高级版本允许将非持久机制的任务加到事务中来。例如一个事务只能在月圆之夜被执行，所以在你的事务中，会向 Moon 类的实例发送一个 isFull()的消息，如果 isFull()返回真，则事务继续，否则事务失败。

5.1.5 PersistenceBroker 类

在很多方法中，图 10 中的 PersistenceBroker 类都是持久层的关键类。这个类实现了 Singleton 模式，在一个应用程序中，只能有一个实例。在运行时刻，PersistenceBroker 维护到持久机制（数据库、文件。。。）的连接并管理与他们（持久机制）的交互。PersistenceBroker 实际上是在 PersistentObject，PersistentCriteria 和 Transaction 之间扮演了一个协调者，因为它是这些类的实例提交自己并被处理的地方。PersistenceBroker 与 SqlStatement 类层次，映射类和 PersistenceMechanism 类层次进行交互。

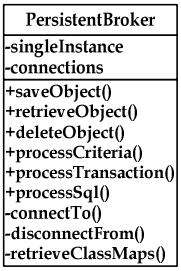

图 10 PersistenceBroker 类

当启动你的应用程序，在开始的时候就要做的事情之一就是让 PersistenceBroker 将创建映射类（ClassMap，AttributeMap,…）实例所需要的信息从持久机制读进来。PersistenceBroker 会在内存中缓冲这些映射类，以便在映射对象到持久机制的时候使用它们。

PersistenceBroker 的一个重要特点就是它的 processSql()方法可以被用来提交硬编码的 SQL 语句。这是一个关键性的方法，因为它允许你在应用程序代码中嵌入 SQL – 当性能是非常重要的时候，你可能会决定重载继承自 PersistentObject 的 save()、delete()和 retrieve()方法，直接提交 SQL 到持久机制。尽管在某些时候常常听起来不错，但也常常无法达到你希望的效果，主要有两个原因：首先这种方式导致耦合了应用程序和持久机制，降低了应用程序的可维护性和扩展性。第二，如果你实际对应用程序进行性能检查，会发现最耗费时间的不是持久层，而是持久机制本身。简单的说，要增进程序的性能，花时间在改进持久结构的设计上比花时间在应用程序代码上更好。

5.1.6 PersistenceMechanism 类层次

图 11 中展示的 PersistenceMechanism 类层次封装了不同持久机制的行为。尽管图中也展示了对对象-关系数据库和文件的支持，实际上我们只关心映射对象到关系数据库。一般来说文本文件相比关系数据库，只提供了顺序读写数据的功能，而对象-关系数据库则提供的更多。

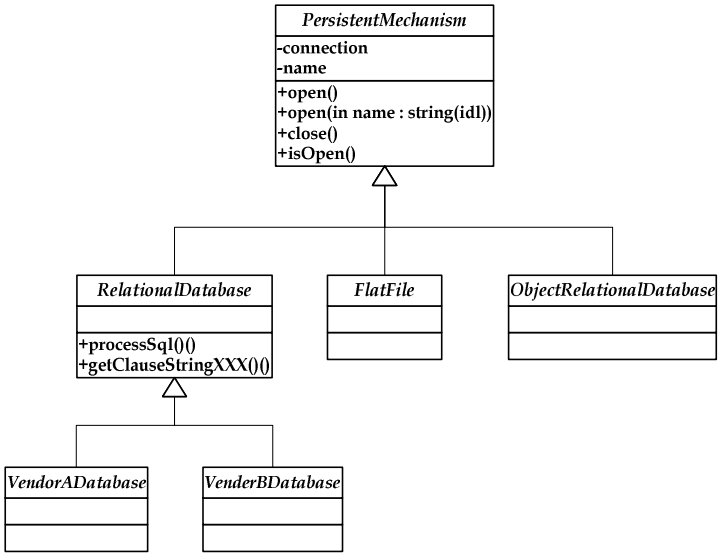

图 11 PersistenceMechanism 类层次

RelationalDatabase 的 getClauseStringXXX()方法代表了一组可以返回一条 SQL 语句的一部分的 getter 方法（这些返回的信息会被 SqlStatement 类层次使用）。例如 XXX 包括：Delete、Select、Insert、OrderBy、Where、And、Or、Clause、EqualTo 和 Between。通常每个方法会有两个版本，例如对于 And，它需要一个 getClauseStringAndBegin()方法返回字符串”AND(”，还需要一个 getClauseStringAndEnd 方法返回”)”以便构造一个完整的 And 语句。这些方法被 SqlStatement 类层次的实例调用，所以他们可以利用每种关系数据库的唯一特性。

RelationalDatabase 支持 ANSI 标准 SQL 子句，尽管它的子类会针对各自对 ANSI SQL 的唯一扩展重载对应的方法。这个类和它的子类包装了诸如微软的 ODBC 和 Java 的 JDBC 等复杂的类库，这样类库的改变就不会影响到你的开发。processSQL()方法带有一个 SQL 语句字符串作为输入，返回零行或者多行的结果集或者一个错误标志。这个方法只被维护到持久机制连接的 PersistenceBroker 调用，而不会被你的应用程序代码调用，应用程序代码根本就不知道这个类的存在，反之亦然。

好消息是，至少对于使用微软环境的开发者来说是，OLE-DB 技术正向这个概念的方向发展。OLE-DB 的基本思路就是使所有使用不同类型持久方式的数据源类型都看起来象个标准的数据源。

5.1.7 映射类

图 12 表达了 ClassMap 组件的类图，一组封装了映射对象到持久机制行为的类。这个设计主要是为了连接对象和关系数据库，尽管你也可以简单的增强使它支持其它的持久机制，诸如文本文件和对象关系数据库等。

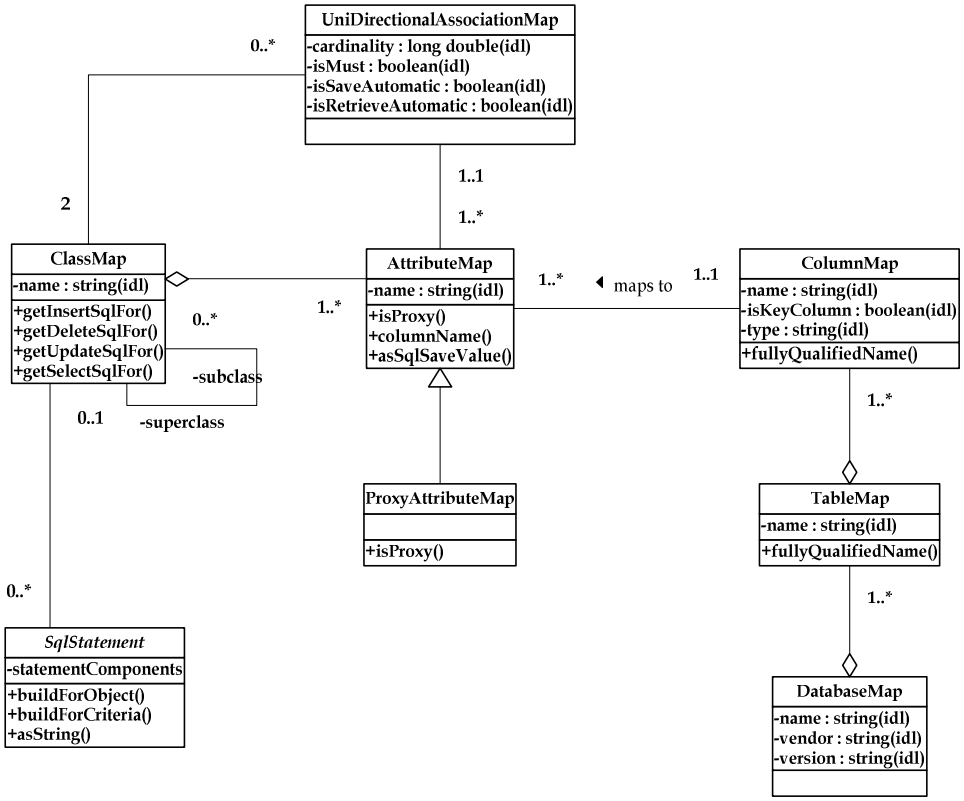

图 12 ClassMap 组件

让我们从 ClassMap 类开始，这个类的实例封装了映射一个类的实例到关系数据库的行为。如果 Customer 类的实例是持久的，就应该有一个 ClassMap 对象用来映射 Customer 对象到数据库。如果一个类的实例不是持久的，例如一个 RadioButton 类的实例，就不需要一个 ClassMap 实例。

ClassMap 对象维护一组 AttributeMap 对象，每个 AttributeMap 对象将一个属性映射到关系表的一列。AttributeMap 对象用于映射简单属性比如存储在数据库中的字符串和数字，或者用于支持 UniDirectionalAssociationMap 类（一会儿再详细说明这个类）的属性集合。AttributeMap 对象知道它们关联到什么 ColumnMap 对象，而 ColumnMap 知道它们的 TableMap 和 DatabaseMap 对象。这四个类的实例用于映射一个对象的一个属性到关系数据库中表的一个列。

ProxyAttributeMap 对象被用于映射一个代理属性，代理属性被用来构造一个对象的代理版本。代理对象只具备用于标识实际对象的信息，而放弃了那些需要大量内存或者网络带宽的属性。ProxyAttributeMap 类在 PersistentCriteria 对象和 Cursor 对象在自动从数据库中获取代理时被用到。

UniDirectionalAssociationMap 类封装了维护两个类之间关系的行为。如果一个关系是双向的，例如一个 Student 对象需要知道它要上的课程，而 Course 对象需要知道来上课的学生，你就需要为这个关系的两个方向各维护一个 UniDirectionalAssociationMap。如果你愿意，你或许应该尝试开发一个 BiDirectionalAssociationMap 类。但是当你考虑这么做的复杂性时，也许会认识到，使用两个 UniDirectionalAssociationMap 的实例更简单。这个映射维护了两个类之间的关系，包括了解是否第二个类应该在第一个类模拟触发器（如果你愿意，也可以

在数据库中完成这个工作）的时候自动保存、删除或者获取。

UniDirectionalAssociationMap 与 AttributeMap 之间关联的实现显示了这个组件中最有趣的部分—有时候 AttributeMap 对象被用来表达一组维护一对多关联的属性。例如，因为一个学生要上一门或者多门课，Student 类和 Course 类之间就有一个一对多的关联。为了维护这个关联，Student 类就应该有一个属性叫做 courses，而它代表了一组 Course 对象。假设 isRetreiveAutomatic 属性被设为真，则当一个 Student 对象被获取的时候，所有与这个学生有关的课程对象都会被获取，它们的引用会自动插入到这个 courses 集合中。类似于在关系数据库中定义触发器的时候你会对你所定义的触发器仔细考虑，使用 UniDirectionalAssociationMap 的 isSaveAutomatic、isRetrieveAutotmatic、isDeleteAutomatic 属性的时候，也应该注意。

为什么你需要这些映射类？简单的说，他们是对对象结构屏蔽持久机制结构的关键（反之易然）。如果你的持久机制结构改变了，可能一个表被改名或者重新组织了，那么你所要做的仅仅是更新映射对象，以后我们会看到，它们都保存在你的数据库中。类似的，如果你重构了应用程序类，那么持久机制结构不需要改变，只有映射对象需要改变。很自然的，如果新增加的特征需要新的属性和列，那么两种结构连同这些映射类都要改变以反映这种变化。

因为性能的原因，ClassMap 的实例维护一组 SqlStatement 对象，缓冲它们来利用每个语句的公共部分。因为同样的原因，尽管我没有展示这种关联，ClassMap 也需要维护一组 DatabaseMap 对象以便 SqlStatement 可以使用 RelationalDatabase 的适当子类，例如 Oracle 8，来获取符合指定数据库规范的语句。没有这个关系，Sqlstatement 就需要遍历映射类之间的关系来获得 RelationalDatabase 的正确子类。

从图 12 中可以学到两个有趣的经验。首先，ClassMap 和 UniDirectionalAssociationMap 之间关联的势（cardinality）为 2 – 我很少指定势的最大值，这是仅有的几次之一（因为一个单向关联永远只能涉及到两个类）。一般建模中使用最大值或最小值不是好办法，因为它们经常变化，而你不会愿意设计依赖最大或最小值。第二，第归关系是很少的几种要在关联中标明角色的情况之一，很多人被第归关系所困惑，例如一个 ClassMap 关联它自己，所以你应该提供额外的信息来帮助他们理解。

5.1.8 SqlStatement 类层次

图 13 展示了用于封装创建 SQL 语句的 SqlStatement 类层次。就象你所期望的，每个子类知道如何根据给定的对象或者 PersistentCriteria 的实例来构建自己。例如 SelectSqlStatement 对象会在获取一个 Customer 对象的时候，通过调用对象的 retreive()方法来创建，或者通过创建一个 PersistentCriteria 的子类 RetrieveCriteria，并调用 perform()方法来创建。

在前面我们已经看到 RelationalDatabase 类层次封装了每个数据库厂家/版本所支持的不同风格 SQL（尽管 SQL 是标准的，但是每个厂家都支持自己的唯一扩展，而我们又想自动使用它们）。SqlStatement 的实例与 ClassMap 的实例合作确定 RelationalDatabase 的子类，通过子类得到 SQL 子句，然后构建自己。

statementComponents 属性对于指定类的对象来说，是一组可重用的字符串。例如 INSERT 语句的属性列表不会随着同一个类的不同实例而变化，对于 INTO 子句来说也一样。

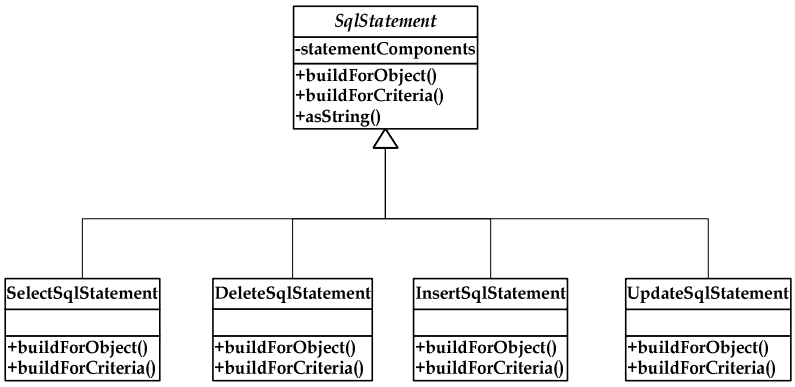

图 13 SqlStatement 类层次

6 持久层的实现

如果你想成功，下面的几个问题就一定要注意：

- 是买还是自己构建？
- 并发、对象和行锁
- 开发语言的问题
- 一个可行的开发计划

6.1 购买 vs 构建

尽管这个白皮书是针对那些想自己构建持久层的人的，但是实际上构建和维护一个持久层是复杂的工作。我的建议是如果你不能彻底完成它，包括在编程结束后的维护和支持，你就不应该开始。

如果你决定不能或者不想构建一个持久层，你就应该考虑买一个。在我的第三本书 Process Patterns(Ambler, 1998b)中，我从经济、技术和操作的可行性等各个角度做了详细的可行性分析。基本理念是你的持久层应该物有所值，应该值得构建或者购买，应该可以长时间的支持和维护（就象前面提到的）。

现在市场上有很多好的持久层产品，我在 http://www.ambysoft.com/mappingObjects.htmls 提供的部分产品连接可以作为你查找的起点。而且在本文中我已经开始着手，至少在一个高层次上，一个持久层的需求列表。你首先要做的应该是充实它们，并根据自己的优先级顺序来判断自己的情况。

6.2 并发，对象和行级锁

从本文的角度来说，“并发”处理与多人同时访问关系数据库中的一条记录有关的问题。因为如果你愿意，允许多用户访问同一条数据库记录也就是同一个对象是可能的，所以你需

要为此确定一种控制策略。关系数据库使用的控制机制是锁，特别是行级锁。行级锁有两种：悲观的和乐观的。

1. 悲观锁。只要记录在内存中，就会一直在持久机制中被锁住。例如，当一个 Cusomter 对象被编辑的时候，在持久机制中就对它上锁，这个对象被读到内存并被编辑，最后再被写回持久机制，这时候对象才被解锁。这个方法保证了一个记录在内存的时候不会被持久机制更新，但是同时不允许其他人在这个对象上工作。悲观锁对于批处理任务是非常理想的。

2. 乐观锁。只有在持久机制中访问记录的时候，才对其上锁。例如，如果一个 Customer 对象被编辑，持久机制在这个对象被读出的时候上锁，然后在读完后立即解锁。在编辑结束后需要保存的时候，它再次被上锁，写回后被解锁。这个方法允许多人同时在一个对象上工作，但是也意味着人们会互相覆盖所修改的内容。乐观锁对于在线处理是最好的方法。

使用乐观锁的情况下，保存记录前，需要花费一些精力确定它是否已经被其他人修改。这可以通过在所有表中增加通用的时戳字段来完成：当你读记录的时候，时戳也被读入；当你打算写回记录的时候，应该将内存中的时戳与数据库中的时戳进行比较，如果他们相同，则更新记录（包括将时戳更新为当前时间）。如果不同，说明有人在你之前更新了记录，你不能覆盖这条记录（因此，可以向用户显示一条消息）。

6.3 开发语言的问题

本文中展示的设计需要“反射”，一种可以在运行时刻动态的操纵对象的能力，以及“内省”，一种运行时刻发现对象接口的能力。反射和内省的必要性体现在根据映射类中包含的元数据动态确定 gettter/setter 方法并正确的调用它们上。反射和内省对于 Smalltalk 和 Java(至少 JDK1.1+)这些语言来说是内建的，但 C++没有。因为 C++缺少这个特性，你需要自己编码完成，典型的，通过结构化的/命名（structured/named）的方法在业务层与持久层之间移动数据集合。就象你能想到的，尽管在一定程度上还是提供了保护，但这样增加了对象结构和数据结构之间的耦合。第二种方法是从映射信息为数据访问类产生源代码，前面已经描述过，然后在需要的时候重新编译。

6.4 开发计划

如果你打算构建一个持久层，这有一个可行的计划可供你选择：

里程碑	任务
1.实现基本的CRUD行为	Ÿ 实现 PersistentObject Ÿ 在 PersistenceBroker 中实现连接管理 Ÿ 实现可以从事先输入好的数据表中读入元数据的映射类（至少是基本的） Ÿ 对于一个对象，实现基本的 SqlStatement 类层次。 Ÿ 实现组织内需要支持的 PesistenceMechanism 类层次。
2．实现对关联的支持	Ÿ 实现 UniDirectionalAssociationMap 类 Ÿ SqlStatement 类层次应该反映关联对构造 SQL 语句带来的复杂

	性。
3．实现对 PersistentCriteria 的支持	Ÿ 实现 PersistentCriteria 类层次，典型的从 RetrieveCriteria 开始，支持查找界面和报表。 Ÿ 更新 PersistenceBroker 使其支持 PersistentCriteria
4．对游标、代理和记录的支持	Ÿ 增加 ProxyAttributeMap 类 Ÿ 增加 Cursor 类 Ÿ 增加 Record 类（如果你的语言不支持） Ÿ 增加 Proxy 类（如果你的语言不支持） Ÿ 修改 PersistenceBroker，使其可以在处理 PersistentCriteria 对象的时候返回对象、行、代理和记录。
5．实现管理工具	Ÿ 参照第 8 节
6．实现事务	Ÿ 实现 Transaction 类 Ÿ 修改 PersistenceBroker 类

表格 3 开发计划

我一向建议从简单的支持一个数据库开始，如果需要在支持同时多个数据库。第 2 步到第 6 步的顺序可以根据你自己的优先级确定。

7 数据加载

在这一节中我将讨论与加载数据到面向对象应用相关的问题。（*译注：数据加载是我对 data loading 的翻译，实际上这个词组的意思应该是将其它系统的数据迁移到自己系统的 schema 中，Loader 意思与 Oracle SqlLoader 中 loader 这个单词的意思一致*）数据加载是系统开发中的现实问题：你需要从旧数据库迁移到一个新版本，或者你需要从一个外部数据源装入测试/开发用的对象，或者就是普通的数据装载，但是要实时的，从非 OO 的/或外部系统装载数据。我先从传统的数据装载技术开始，然后再提出一个对 OO 应用来说更明智的方案。

7.1 传统数据加载方法

传统数据加载方法就象在图 14 中表现的那样，写一个程序将数据从源数据库中读出，对其进行清理，然后再写到目标库中。清理可以是简单的对数据标准化，或者对单字段的处理，例如从 2 位数的年份转换为 4 位数的年份，也可能是对多字段进行处理，例如某字段的值暗示着另外一个字段的含义（是的，这是源数据中不可想象的糟糕设计，但是在很多以前的数据库中，这都是很平常的事）。对于外键的完整性，也就是保证一条记录中的所有引用所指向的记录都确实存在，也被编码在了数据装载程序中。

图 14 传统的数据装载方法

有很多程序使用这种方法。对于这种方法，首先最重要的，目标数据不再被封装了—如果你的持久机制结构发生了变化，你必须改变你的数据装载程序的代码。如果数据装载工具是在操作元数据（对于结构化的技术，它们实际上有一个持久层）的话，这还可以忍受。第二，你的数据装载工具好象是实现了已经封装在业务对象中的一部分重要逻辑。业务对象虽然不会去解决旧的源数据中的问题，但是它们会被设计为可以保证对象一致性，包括所有的引用完整性问题。这种方法的要点是，你要为很多基本行为在两个地方写代码：一个是你的业务类中，这是本来就应该的，另一个是在数据装载程序中，而这是不应该的。所以说还应该有更好的办法。

7.2 设计良好的数据加载

图 15 描述了一种与对象开发更加协调的数据加载方式。数据加载程序由一组类组成。首先有一些拥护接口类，可能是个管理界面或者是显示日志的界面。第二，会有一些专门针对数据加载的业务类，它们封装了对源数据进行清理的逻辑。你不要将这个功能放到正常的业务类中，因为以后你以前的源数据会消失，而替代它的是现在被改进的目标数据。还会有封装了数据装载过程逻辑的类，它们使用数据装载的业务类读入数据，然后根据读入的数据创建“真正”的业务对象。如果你不是对目标数据进行完全刷新，你需要首先读入已存在的对象，根据源数据更新它们，然后再写回。

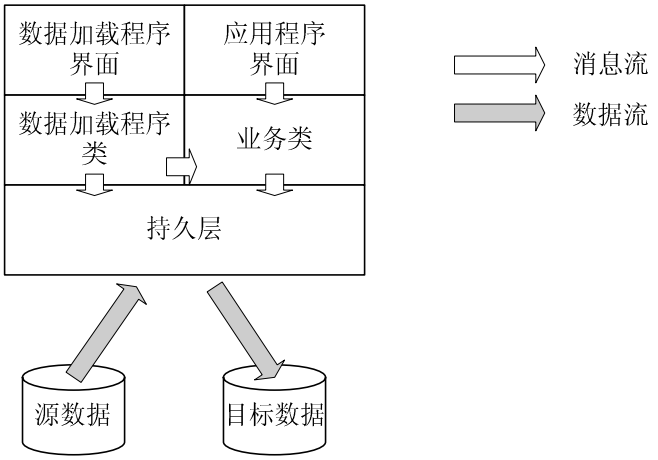

图 15 具有良好设计的数据装载方法

关于图 15，有两个有趣的问题。第一，注意你的“数据装载程序代码”永远不会直接访问源数据—它通过持久层来获得数据。第二，数据装载程序代码即使被移走也不会影响应用程序和业务类，也就是说应用程序不知道也不关心数据装载程序所操纵的源数据。

这种方法有多个好处：

- 数据装载逻辑与目标数据结构无关，允许你根据需要更新目标结构而不需要修改数据装载程序。
- 关键业务逻辑被封装在应用程序的业务类中，这也是它应该存在的地方，这使你可以只在一个地方对其编码。
- 数据清理逻辑被封装在数据装载程序的业务类中，也是它应该存在的地方，使你可以只在一个地方对其编码。

这种方式也有一个缺点，你的组织所购买的昂贵的数据装载工具很可能无法工作在这种结构中，他们大多数都是基于图 14 那种原始的/旧的方式，这会对这些工具的用户产生行政方面的问题。

8 支持持久层

你的组织内如何对持久层进行支持？首先，你需要开发一个提供维护映射类实例的管理系统。这个管理系统可以被负责开发和维护持久结构的人或者维护对象结构的人更新。你还可以为持久层增加一个缓冲以提高性能。

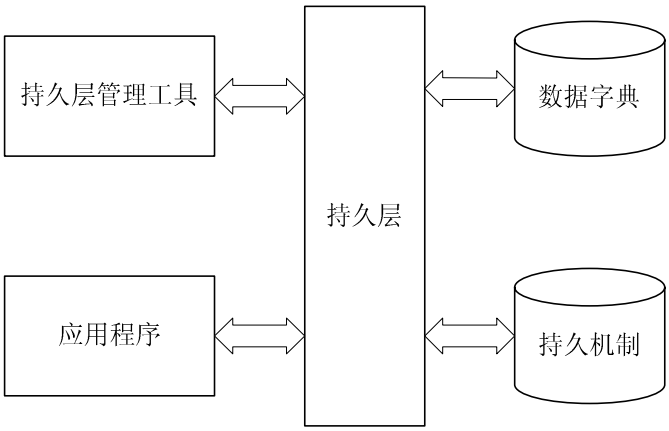

图 16 持久机制是如何工作的

图 16 展示了通过构造一个管理工具来维护 ClassMap 类的实例。这些对象封装了将对象映射到持久机制的行为，包括那些组成了应用程序并构成存储在数据字典中的信息的对象之间复杂的关系。关于成功持久层的秘密是：存储在数据字典中的对象提供映射对象到持久机制的行为。当应用程序或者持久机制的设计发生了变化时，你仅仅需要更新数据字典中的映射对象，而不需要更新应用程序源代码。

这个持久化的方法实际上允许你的数据库管理员（DBA）去做他们最擅长的—管理数据库，而不用要求他们去关心所做的变化会对现有应用程序会有什么影响。只要他们保证数据字典总是最新的，那么他们可以对持久机制做任何他们想做的事。同样，应用程序开发者可以重构他们的对象而不需要关心对持久机制的更新问题，这得宜于他们可以将新版本的对象映射到现有的数据结构上。很自然，如果应用程序中增加（删除）了新的类或者类中增加（删除）了新的属性，持久机制结构也应该做出相应的变化。

9 小结

本文的目的是阐述一个可工作的鲁棒的持久层设计，一个经过实践证明可以工作的设计。面向对象应用程序可以使用关系数据库作为持久机制而不需要在程序中使用嵌入 SQL。类似 JDBC 和 ODBC 之类的技术可以被本文中阐述的设计包裹起来，避免那些在设计的时候没有仔细考虑对持久机制维护和管理的应用程序带来的继承脆弱性。如果你愿意，面向对象应用程序中的持久化可以非常容易。

10 参考书目和推荐读物

Ambler, S.W. (1995). The Object Primer: The Application Developer's Guide To Object Orientation. New York: SIGS Books.

Ambler, S.W. (1998a). Building Object Applications That Work – Your Step-by-Step Handbook for Developing Robust Systems With Object Technology. New York: SIGS Books/Cambridge University Press.

Ambler, S. W. (1998b). Process Patterns: Building Large-Scale Systems Using Object Technology. New York: SIGS Books/Cambridge University Press.

Ambler, S.W. (1998c). Mapping Objects To Relational Databases: An AmbySoft Inc. White Paper.http://www.ambysoft.com/mappingObjects.html.

Ambler, S.W. (1998d). Persistence Layer Requirements, Software Development, January 1998, p70-71.

Ambler, S.W. (1998e). Robust Persistence Layers, Software Development, February 1998, p73-75.

Ambler, S.W. (1998f). Designing a Persistence Layer (Part 3 of 4), Software Development, March 1998, p68-72.

Ambler, S.W. (1998g). Designing a Robust Persistence Layer (Part 4 of 4), Software Development, April 1998, p73-75.

Ambler, S.W. (1998h). Implementing an Object-Oriented Order Screen, Software Development, June 1998, p69-72.

Ambler, S.W. (1998i). More Process Patterns: Delivering Large-Scale Systems Using Object Technology. New York: SIGS Books/Cambridge University Press.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). A Systems of Patterns: Pattern-Oriented Software Architecture. New York: John Wiley & Sons Ltd.

Rational (1997). The Unified Modeling Language for Object-Oriented Development Documentation v1.1. Rational Software Corporation, Monterey California.

11 词汇表

Aggregation – 聚集。表示一种“is-part-of”的关系。
Anti-pattern – 反模式。用于解决某一类问题的方法，这种方法已经被证明是错的或者极其没有效率的。
Application server – 应用服务器。部署业务逻辑的服务器。应用服务器是多层结构中的关键。
Association – 关联。对象之间的关系，例如：客户—购买—产品。
Associative table – 关联表。关系数据库中用于维护两个或多个表之间关系的表。通常用来表达多对多关系。
Client – 客户。一个提供表示服务和适当计算、连通性以及接口的单用户 PC 或者工作站。“client”也被称为前端（front-end）。
Client/server (C/S) architecture – 客户机服务器（C/S）结构。一种通过在客户机和服务器进程之间适当分配应用程序过程来满足业务需要的计算环境。
Composite key – 组合键。由两列或多列组成的键。
Concurrency –并发。与多人同时访问持久机制有关的问题。
Coupling – 耦合。两个单元之间的关联程度。
CRUD – create、retrieve、update、delete 的缩写。持久机制必须支持的基本操作。
Data dictionary – 数据字典。保存数据库、文本文件、类的布局信息以及它们三者之间映射的仓库。
Database proxies – 数据库代理。一个表达存储在数据库中的业务对象的对象。数据库代理看起来就象它所代表的对象，当其它对象向它发送代理消息的时候，它立即将对象从数据库取出来并用取出来的对象代替它自己，并将消息传递给这个新对象。详细内容请参照第四节中的“代理模式”。
Database server – 数据库服务器，一个安装了数据库的服务器。
Distributed objects – 分布式对象。一种面向对象的结构，对象运行于不同的内存空间（例如，不同的计算机），但可以透明的互相交互。
Domain/business classes – 领域/业务类。领域/业务类针对业务领域建模。业务类通常在分析过程中被发现，这种类的例子包括 Customer 和 Account。
Fat-client – 胖客户机。两层结构中同时实现了用户接口和业务逻辑的客户机。典型的服务器则只向客户机提供数据以及一点儿或者干脆没有对数据的处理。
Key – 键。组合起来后可以唯一标识关系数据表中一条记录的一列或多列。
Lock – 锁。表、记录、类、对象等的保留标志，通过上锁保证对一个 item 的操作可以完成。锁的周期为：上锁、完成操作、解锁。
n-Tier client/server – 多层客户机/服务器一种客户机与应用服务器交互，而后者与其他应用服务器或者数据库服务器交互的客户机服务器结构。
Object adapter – 对象适配器。一种既可以将对象转换为持久机制中的记录也可以将记录转换回对象的机制。对象适配器也可以用于转换对象和文本文件中的记录。
Object identifiers (OIDs) – 分配给对象的唯一标识符，通常是一个很大的整数数字。在面向对象中，OID 相当于关系数据库中的键。
ODMG – Object Database Management Group, 一个由 ODBMS 厂家组成的标准化委员会。
OOCRUD – 面向对象的 CRUD。

Optimistic locking – 乐观锁。一种对象只在被数据库存取的时候才加锁的并发机制。例如，如果一个 Customer 对象被编辑，持久机制在这个对象被读出的时候上锁，然后在读完后立即解锁。在编辑结束后需要保存的时候，它再次被上锁，写回后被解锁。这个方法允许多人同时在一个对象上工作，但是也意味着人们会互相覆盖所修改的内容。

OQL – Object Query Languages，对象查询语言。ODMG 对于对象查询建议的标准。它由基本的 SQL 以及面向对象的扩展组成，可以与类或对象而不是关系表和记录一起工作。

Pattern – 模式。/对于一类问题的泛化解决方案，具体问题可以根据它来确定详细解决方案。软件开发模式有很多类，包括但不限于：分析模式、设计模式和过程模式。

Persistence – 持久化。关于如何在永久存储上保存对象的问题。如果你希望在下一次运行程序的时候仍然可以访问以前的对象，它们就应该被持久化。

Persistence classes – 持久类。持久类提供永久存储对象的能力。通过持久类对存储和获取对象进行封装，你可以变换使用不同的存储技术而不影响你的应用程序。

Persistence layer – 持久层。一组可以让业务对象持久化的类。持久层实际上包装了持久机制。

Persistence mechanism – 持久机制。用于对象持久化的永久的存储设施。例如，关系数据库、对象数据库、文本文件和对象/关系数据库。

Pessimistic locking – 悲观锁。一种对对象在内存中的整个周期进行保护的并发机制，只要记录在内存中，就会一直在持久机制中被锁住。例如，当一个 Cusomter 对象被编辑的时候，在持久机制中就对它上锁，这个对象被读到内存并被编辑，最后再被写回持久机制，这时候对象才被解锁。这个方法保证了一个记录在内存的时候不会被持久机制更新，但是同时不允许其他人在这个对象上工作。

Primary key – 主键。表中主要的键。

Process anti-pattern – 过程反模式。描述了开发软件的过程中已经被证明为低效率和错误的一种反模式。

Process pattern – 过程模式。对软件开发过程中被证明为成功的方法或者一系列行为的描述。

Read lock – 读锁。一种用于指示一个表、记录、类、对象正在被其它人并发读的锁。其它人可能也对对象获得读锁，但是除非所有读锁都解开，没有人会获得写锁。

Reading into memory – 读到内存。从持久机制获得一个对象，但并不打算更新它。

Retrieving into memory – 获取到内存。从持久机制获得一个对象，但可能更新它。这也被称为：为更新而读。

Secondary key – 次键。关系表中不是主键的键。

Server – 服务器。一个或多个共享内存的，可以提供连通性、数据库服务和业务需要的接口的多用户处理器。Server 也被称为“后端”（back-end）。

SQL – Structured Query Language。结构化查询语言。一种在关系数据库中使用 CRUD 操纵数据的标准机制。

SQL statement – SQL 语句。一段 SQL 代码。

System layer – 系统层。一组为应用程序提供操作系统规范功能的类，或者由非 OO 应用程序、硬件设备以及非 OO 代码库提供的功能包装。

Thin client – 瘦客户机。在两层结构中只实现用户接口的客户机。

Transaction – 事务。事务是在持久机制中执行的一个任务单元。事务可以由一个或多个对持久机制的更新、一个或多个读、一个或多个删除或者它们的任意组合组成。

User-interface classes – 用户接口类。用户接口类为用户提供了与系统交互的能力。用户接口类通常为应用程序定义图形用户接口，尽管其它接口风格，例如语音命令或者手写输入也

可以通过用户接口类实现。

Wrapping – 包装，包裹。对非 OO 功能的封装，以使其观感类似系统中的其它对象。

Write lock – 写锁。一种用于指示一个表、记录、类、对象正在被其它人并发写的锁。除非这个锁被解开，没有人会获得写锁或读锁。

12 关于作者

Scott W.Ambler 是一个对象开发顾问，生活在距加拿大多伦多以北 60 公里安大略省一个叫做 Sharon 的村子里。他从 1990 年开始就开始了 OO 方面的工作并担任不同的角色：业务架构、系统分析、系统设计、项目经理、Smalltalk 程序员、Java 程序员和 C++程序员。他作为正式的培训教师和对象专家也活跃在教育和培训领域。Scott 是 Software Development（http://www.sdmagazine.com）的编辑以及 Object Magazine（http://www.sigs.com）和 Computing Canada（http://www.plesman.com）的专栏作者。可以通过 scott@ambysoft.com 联系到他，也可以访问他的个人网站：http://www.ambysoft.com。

关于 The Object Primer

The Object Primer 用易于理解的方式介绍了对面向对象分析和设计技术。面向对象是结构化方法之后系统开发的最重要改变。因为 OO 经常用于开发复杂的系统，所以它本身不应该被复杂化。这本书与其它关于面向对象方面的书不同的地方在于—它是从现实中开发者，那些在学习这个令人激动的新方法的过程中遇到无数挫折的人，的角度出发来写的。The Object Primer 的读者已经发现它是目前市场上可以买到的最简单的面向对象开发介绍，他们中的一些人已经与我分享了他们的意见与荣誉。内容包括：CRC 建模、用例、用例情景测试和类图。

关于 Building Object Applications That Work

Building Object Applications That Work 内容包括：构建可维护、可扩展的应用程序；使用 UML 的分析和设计技术；创建独立的、客户机/服务器的和分布环境的应用程序；在持久化中关系和面向对象数据库；OO 度量标准；通过 OO 模式来提高应用程序质量；OO 测试（这个不简单，非常困难）；用户接口设计以及使应用程序易于维护和扩展的编码方法等。

关于 Process Patterns 和 More Process Patterns

Process Patterns 和 More Process Patterns 开辟了对象开发中一个新的令人激动的领域：软件开发的可重用方法。就象设计和分析模式描述了通用建模问题的解决方案，过程模式描述了如何组织管理软件过程的可重用的解决方案。这两本书阐述了一组方法、生命周期、阶段和任务过程模式，它们一起显示了如何开发、维护和支持大规模、针对关键业务的使用对象技术的软件。对象开发是宏观连续，微观迭代，随时间推移发布增进版本的方法。这些书现在可以从剑桥大学出版社预定（细节请参照 http://www.ambysoft.com/processPatterns.html）。

关于 AmbySoft 公司的 Java 编码标准

AmbySoft 公司 Java 编码标准总结了常见的 Java 编码标准，也阐述了一些可以增进代码质量的原则。这是一个 Adobe PDF 格式的文件，可以从 http://www.ambysoft.com 下载。