第三部分　数据库集成和事务处理

这部分探讨如何把CORBA和已有的数据库管理系统集成，以建立基于CORBA的事务处理系统。

首先，简要介绍对象持久性(第9章)。这部分的介绍和CORBA完全无关，但是以后章节将用到这里所讨论的技术。基于对象持久性的简要介绍，本书进一步探讨CORBA和数据库的集成，以及如何实现持久CORBA对象(第10章)。然后探讨CORBA环境下实现事务的不同方法(第11章)。其中一个方法直接引出分布式事务处理并详细讨论(第12章)。这里介绍的一个重要技术是基于CORBA的事务服务——对象事务服务(OTS)。既然OTS是建立基于CORBA的事务处理系统的核心技术，本书将解释如何用OTS编程，以及OTS如何影响ACID属性和并发控制等等。这部分还介绍了高级的事务概念，如多事务、Saga事务和嵌套事务。与这些主题相关的还有会话。CORBA环境下的会话非常复杂，经常涉及在数据库提供的标准机制上的额外并发控制机制。因此，在这里讨论不同的并发控制机制，以及数据库级的并发控制如何与应用程序级的并发控制结合。最后基于这些讨论研究一个实例(第13章)。

第9章　对象的持久性

现在的数据可以用各种不同的格式储存。通用数据储存机制包括大型机领域的层次和网状数据库，更新的关系和面向对象的数据库。另外，大量的数据储存在文件里，例如：配置文件、HTML文件、字处理文件和表格文件。

本章探讨如何通过对象访问这些不同数据格式，或者说，如何把对象储存为不同数据格式。我们简要地探讨对象模型和不同数据模型间的映射，主要侧重于对象关系映射技术。这反映着一个事实：今天很多工程都是结合了面向对象程序语言和关系数据库的。

注意，本章与CORBA对象和持久性完全无关，但它提供了以后章节讨论CORBA和数据库集成以及事务的基础。

9.1　简介

在简略描述关系和面向对象数据库管理系统的特性后，本章探讨阻抗失配现象，及其在持久对象系统中对于不同类型DBMS影响的严重性。基于这些讨论，定义了对象持久性服务的共同特性。

假定关系与对象模型间阻抗失配通常十分严重之后，我们探讨了对象/关系映射技术。此后，简单讨论了面向对象数据库。

提供保持数据一致性的支持，是任何持久性服务的主要要求之一。因此，本章还将讨论数据一致性，包括分布式系统中的事务、并发控制和数据同步。

9.1.1 RDBMS与ODBMS

通常，对象数据库管理系统 (ODBMS)提供了在持久库中储存对象和关系的最自然的方法。对象的持久状态被存在数据库中。数据库客户机提供对象实现。既然状态和实现分离， ODBMS有责任提供一种合并机制以使编程者看来像是在处理现实的持久对象。

大部分商用 ODBMS与特定编程语言紧密配合，通过框架编译器从程序代码中的类定义产生出数据库框架。 ODBMS常在客户库中实现某种形式的缓存。通常，使用某种灵巧指针来使客户机得以访问缓存中的持久对象。若通过这种灵巧指针访问一个持久对象，灵巧指针先检查它是否在本地缓存。若是，则把此访问直接委托给临时实例，它代表了本地缓存中的持久性对象；否则，在委托前必须先从数据库获得对象状态。通常，持久性对象只能在事务上下文中访问，事务提交后，将变动从缓存写回数据库。大多数 ODBMS提供某种数据库服务器，以向数据库客户机提供对象状态和管理功能，如实现对象并发访问的对象锁定表。商用ODBMS的成功例子包括 Versant、ObjectStore、Objectivity和Poet。

关系数据库管理系统 (RDBMS)的理论基础是关系演算。 RDBMS中，数据存在表里。每个数据项用表中的一行表示。每行可能有很多列。每列代表一个像数字或字符串等等的基本数据类型，键标识表中的一行。关系用外键表示，指向相同或不同表中的行。数据库客户机和服务器通过SQL交互。大部分RDBMS提供基本的语言绑定，使数据库客户机可以发送 SQL语句给服务器，以及接收结果。查询结果通常是一个行集，类似于面向对象的迭代器，数据库用游标来操纵变长查询结果。商用 RDBMS的成功例子有Oracle、Sysbase、DB2和Informix。

关系数据储存非常适合于保存海量数据，使用 SQL作为访问数据的灵活有效的语言。不幸的是，RDBMS的关系遍历代价很大。这是 ODBMS的传统领域，可以通过复杂对象图的管理和导航而高度优化。 RDBMS和ODBMS在关系遍历上的关键区别是它们操纵缓存的方式。相对于RDBMS，ODBMS提供了优化特定应用的缓存策略的更复杂机制。例如，对象簇可把对象逻辑地分组，以反映对象间特定应用的依赖关系。

但若说 ODBMS很适于实现持久性对象，为什么很多系统却使用 RDBMS呢？通常，用RDBMS而不是ODBMS的决策不仅受技术因素影响，还受战略因素影响。但战略的考虑不是为什么今天很多工程结合使用面向对象语言和关系数据库的唯一理由，另一个重要原因是在关系技术上的已有投资。

9.1.2 阻抗失配问题

传统上，术语阻抗失配用来描述数据库和访问数据库的程序语言两者数据结构间的差异。

实现持久性对象的系统通常基于三种不同模型：设计对象模型、实现对象模型和持久对象模型。这三种不同模型间常发生冲突。本书中 ，这种面向对象系统中的三方冲突，被称为阻抗失配。图9-1显示了面向对象系统中阻抗失配的三角冲突。这说明不仅在本质不同的模型世界之间(如面向对象和关系模型之间)存在失配现象，不同的对象模型间也有。例如，对象的设计模型可能基于多继承，而实现语言却可能不支持。或者更糟的是，模型是面向对象的，而程序语言

完全不支持面向对象。例如，MOTIF GUI类库是用面向对象风格实现的，却使用非面向对象的C语言。

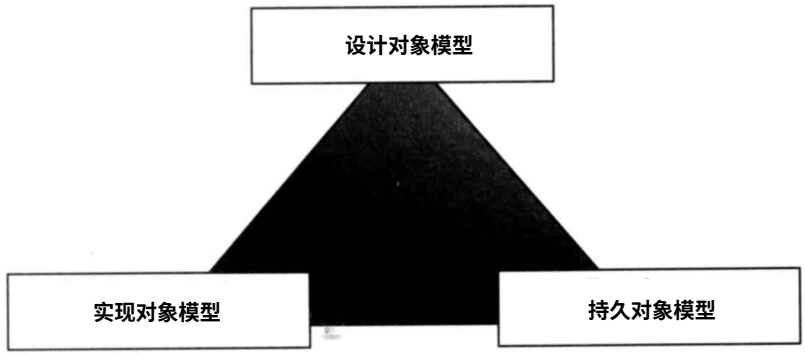

图9-1 OO阻抗失配冲突三角

下面讨论两个阻抗失配的例子，一个是中度失配的，另一个则是极度失配的。

假如从草图建立一个应用程序，使用的是面向对象建模语言UML、面向对象编程语言（如C++）和ODBMS（如Versant或ObjectStore）。下面从对象模型开始。也许很幸运，建模工具有代码生成特性，可以从对象模型生成框架实现，包括类文件和空方法体。也许更幸运，建模工具支持某种逆向工程，可以把实现模型所作的改变反馈回设计模型。然而，更可能的是存在轻度失配，因为通常建模工具不能表现所有程序语言特性，反之亦然。

使用ODBMS时，实现对象模型和具体说明持久对象框架通常是一起进行的。但是，一个常见的问题是，ODBMS要求使用DBMS指定的数据类型，它可能不被建模工具所支持。因此，实现模型很可能更接近于ODBMS强加的模型。而且，ODBMS可能不支持特定编程语言所支持的所有特性。例如，编程语言可能支持多继承，而ODBMS不支持。

在第一个例子中，设计模型和实现模型之间只有轻度失配。实现模型和数据库框架间的失配将取决于编程语言和ODBMS的适配程度。如果ODBMS是专为特定的OO编程语言设计的，那么失配将极其可能是很小的。

第二个例子假定使用RDBMS。给定一个高度规范化的数据库框架，它根本不是面向对象的。框架描述的数据库被已有的应用程序使用。所以，不能为新的应用程序的需要而改造它。现在可以对这个框架进行面向对象分析，并得到一个对象模型，我们认为它一方面表述了数据库框架，另一方面表述了要实现的业务对象模型。从对象模型中可以派生出上面描述的实现类。现在最大的问题是，在多大程度上需要把对象映射到数据库中的行，这通常是一个很困难的工作，尤其是在框架已被高度规范化的情况下。实现模型中的一些类和数据库中的表可能存在联系，但如果框架是规范化了的话，类和表常常是正交的。在这种情况下，实现映射通常是视条件而定的，要寻求一种结构化的方法来桥接失配并不容易。

通常，在使用OO编程来和RDBMS连接时，阻抗失配问题是特别严重的。因此，在9.2节“访问关系数据库”里会进一步探讨如何从OO编程访问RDBMS。但首先要定义对象持久性服务的标准特征。

9.1.3 对象持久性服务的特征

实现持久对象有不同的途径。有时 DBMS直接支持对象持久性。其他情况下，可以在标准DBMS服务之上拥有对象持久性层，以桥接编程语言的对象模型和 DBMS的对象模型之间的阻抗失配。对象持久性服务一般需要的特征有：

- 支持面向对象　持久性服务应该支持重要的面向对象特征，如封装、对象标识、对象关系、继承和多态。
- 基本数据库操作　应该支持对象创建、修改和删除的基本操作。在关系世界中，这些基本操作通常称为CRUD操作(创建/读取/更新/删除)。
- 查询和获取　通常用某种语言来表示查询。集合或迭代器可用于处理从查询获得的结果。
- 多对象行为　需要支持多对象复合行为。
- 并发访问　应该支持并发数据访问的机制，通常基于事务概念或者对象版本。
- 语言绑定　应该有不同编程语言的绑定。例如，C++、Smalltalk 和Java。

9.2 访问关系数据库

数据库客户机通过向服务器发送SQL查询来和关系数据库服务器交互。数据库服务器可能根据查询来更新一些数据或者返回查询结果，查询和查询结果通常以名/值对的集合形式提供。一种关系数据库的大部分语言绑定是通过把名/值对映射成编程语言变量来处理的，反之亦然。这使得客户机能够向服务器发送查询，并处理查询结果。这一节看看关系数据库的一些普通的语言绑定。

9.2.1 本地的SQL API

大部分关系数据库厂商提供了传统编程语言 (如C和COBOL)的绑定，少数厂商为 OO编程语言(如C++和Smalltalk)提供了专门的绑定，而极少数还提供了通过 Java的本地访问方式。本地SQL访问产品有Oracle Call Interface(OCI)、Oracle Pro*C(嵌入SQL)和Informix ESQL(嵌入SQL)。注意，调用层接口一方面有时比嵌入 SQL提供更多的灵活性，但另一方面通常更难编程。以下代码片段显示了嵌入式SQL如何为C语言提供语言绑定：

```
// Pro*C example for embedded SQL

void db_set_stock_description(char* symbol, char* desc)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char db_symbol[6];
    char db_desc[20];
    EXEC SQL END DECLARE SECTION;

     // Not shown: copy arguments to db variables

    EXEC SQL UPDATE STOCK
     SET DESCRIPTION = :db_desc
      WHERE CODE = :db_symbol;

    // Not shown: handle errors
}
```

这个例子显示了函数如何为 STOCK表中给定的股票设置新的描述。回忆一下第 4章介绍的STOCK和STOCK_PRICE表。STOCK表有两列：SYMBOL和DESCRIPTION。STOCK_PRICE表使用外键SYMBOL来引用STOCK表。这一章中用这两个表来讨论一些例子。

STOCK表	
String SYMBOL	String DESCRIPTION

9.2.2 开放连接

对标准化关系数据库访问已进行了一些尝试。其思想是定义一个标准 API，可用于访问关系数据库，并为特定数据库提供不同的 API实现。最普遍的开放连接标准是 Microsoft的开放数据库连接(ODBC)。更新的有Java编程语言的类似标准如JDBC，它使Java程序能够以标准方式访问关系数据库。

开放连接工具提供的编程机制介于前面讨论的本地 SQL API和后面介绍的OO SQL封装类之间。

9.2.3 OO SQL封装类

最后看看OO SQL封装类。这是把SQL元素建模为对象的类库。但是注意，这并不意味着数据本身以对象形式暴露。在下面使用RogueWave DBTools.h++的例子中，可以看到SQL元素(如数据库、表、列和结果)如何建模为C++对象。这个例子选择所有价格大于50的股票，并打印结果集。

```
// Rogue Wave DBTools.h++ example:
// Database connection details not shown
RWDBDatabase aDatabase = RWDBManager::database (...);
RWDBTable aTable = aDatabase.table("STOCK_PRICE");
RWDBSelector aSelector = aDatabase.selector();

aSelector << aTable;
aSelector.where(aTable["PRICE"] > 50f);
RWDBResult aResult = aSelector.execute(session);
RWDBReader aReader = aResult.table().reader();

float balance;
RWCString name;
while(aReader()) {
 aReader >> balance >> name;
 cout << name << ":" << balance << endl;
}
```

9.3 对象/关系映射

在前面的讨论中，介绍了一些使我们能处理 SQL的不同语言绑定。所有这些语言绑定都在某种程度上把SQL元素绑定到编程语言上。下面介绍一种叫做对象 /关系映射(O/R映射)的技术。O/R映射和上面的SQL语言绑定之间的根本区别是：O/R映射试图提供关系数据的一个面向对象视图，这意味着数据本身被表示为对象。

9.3.1 基本映射技术

下面解释用于O/R映射的基本映射技术，包括讨论从基本面向对象概念（如类、关系和继承）到关系型世界的映射。

1. 从类映射到表

对象/关系映射的基本思想其实很简单：类映射成表，对象实例映射成行，对象属性映射成关系表中的列。图9-2显示了基本对象/关系映射。

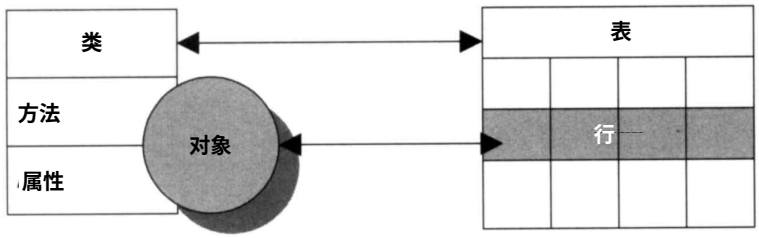

图9-2 基本对象关系映射

通常，这种简单的类和表之间的一对一映射是不够的。例如，如果关系型框架是高度规范化的，用一个类来封装多个表就可能更好，即单个对象能够表示不同表中的多个相关的行。在9.3.5节“自顶而下和自底而上”中将进一步讨论规范化问题。最后，有时还有一些把不同的类映射成同一个表的情况。这种映射的一个例子是映射继承的统一方法（下面将讨论）。

2. 对象ID

每个对象应该有一个唯一的ID，而且在对象生命期中不应该改变。通过给每个对象分配一个叫做对象标识(OID)的属性(通常是一个大整数值)，这就很容易实现。在关系型世界中，这意味着每个表有一个OID列。理想情况下，OID应该没有实际意义。一个有实际意义的列总有可能改变，这不仅影响这个表本身，而且影响到使用这个ID作为到基本表的外键的其他表，还影响到索引。一般地，在所有表中使用RDBMS为标识而产生的代理码提供了一种更标准的访问机制，促进了使用类属码和辅助函数来访问关系库中的数据。

3. 关系

关系在对象世界和关系世界中都扮演着重要的角色。在关系世界中，引用关系是通过外键来实现的，这导致了双向的关系，因为总能找到匹配主码的外键，反之亦然。使用面向对象编程时，表示关系的直接方式是用指针或者引用。很多编程语言提供了列表和集合类型，能用来表示一对多关系。哈希表也常常用来实现定量的关系。把对象关系映射到关系模型的最普通的技术有：

- 嵌入式外键　关系可以直接映射成外键。这通常是映射一个关系的非常有效的方法。嵌入式外键的一个例子是STOCK表和STOCK_PRICE表之间的关系——STOCK_PRICE表有一个外键SYMBOL，指向STOCK表的一行。
- 不同的连接表　关系还可以通过创建一个额外的表(连接表)来表示。这种方法有时增加了灵活性，但也增加了开销，因为显式的连接表在执行查询时需要额外的连接。通常，连接表只用来表示多对多对象关系。
- 折叠类　最后，可以把类折叠进单个表中以合并它们。通常，这只用于一对一关系和属性。

作为折叠类的一个例子，假定要用一个新的、稍微复杂的类型表示金钱；金钱对象有两个属性：货币和数量。股票价格有一个金钱类型的属性。出于性能考虑，可以直接向STOCK_PRICE表加两列：CURRENCY和AMOUNT，而不用定义一个新表MONEY。

在对象世界中，引用和聚集都是重要的关系。引用关系可以用以上技术之一轻易实现，而聚集关系较为困难。在对象世界中，我们想用复合聚集来实现迭代删除（即聚集体必须作为整体删除，包括所有聚集对象）。在关系世界中，常常用数据库触发器来进行聚集值的迭代删除。

4. 继承

关系模型中并不提供继承的概念。因此，把对象模型的继承层次映射到关系框架中并不容易。对象模型的继承映射到关系模型中有几种可能方式，各有一定的优点和缺点。例如，可以把每个类映射成相应的表，也可以只把类片段映射成表，或者把所有类映射成单个的表。每种方式对于继承层次的变化，都有不同的表现特征和不同程度的灵活性。

9.3.2 水平分割

继承图中的每一个具体类映射成一个单独的表。每个表包含这个具体类的所有属性的列定义，加上它从具体和抽象的基类继承来的所有属性的列定义。抽象类不映射成相应的表。水平分割能支持多继承，列名可能用属性首次定义所在的类的名称。水平分割提供的映射支持了简单的“每个类映射成一个表”的概念，并为强类型控制操作提供了良好的性能。图 9-3的例子显示了水平分割如何查找有两个特例（可选股和优先股）的基类股票。获取所有优先股的请求只需要单个地返回整个集合的简单查询；本质上是发出一个“ SELECT * FROM PREFERRED_STOCK ”

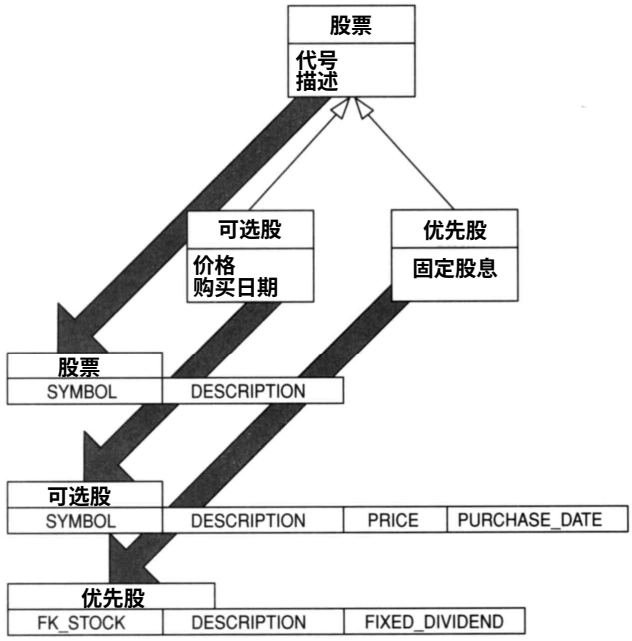

图9-3 水平分割

SQL调用。然而，这也有很多缺点。首先，如果抽象基类之一或者继承图根部附近的具体类发生改变，比如另外加一个属性，那么从这个类派生出的每个具体类都必须改变。从查询的角度看，一个弱类型的查询将需要多表查询以及结果的聚集。例如，为了取得所有股票对象，必须发出三个分别的SQL SELECT查询；一个用来获取所有的股票实例，第二个用来获取所有可选股实例，第三个用来获取所有的优先股实例。这些中间结果集合必须合并起来产生一个包含所有股票的统一集合。也可能通过定义一个覆盖水平分割框架的关系型视图来使这类查询更为有效。

9.3.3 垂直分割

每个派生类映射成一个新的表，只包含此类中显式定义的那些属性。另外，每个表都包含到其基类对应表的外键。通过在派生类的表中包含多个外键列来支持多继承。扩展和修改也很容易实现，因为只须改变一个表。但是，任何获取派生类 /表的实例的查询都需要至少一次连接；当继承层次很深时，还需要多重连接。这影响了操纵数据的所有操作 (插入、查询、更新和删除)的性能。垂直分割的一个例子如图 9-4所示。

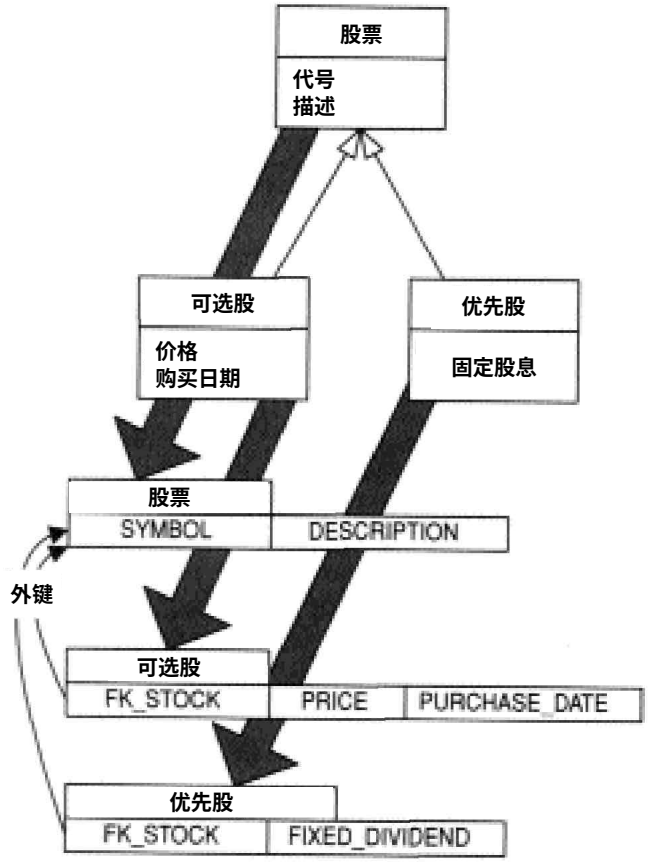

图9-4 垂直分割

9.3.4 统一

在同一个根分支下的各层次所有抽象类和具体类被映射成单个的表。一个增加的列，叫做类型识别器，用来标识每一行的实际类型。大多数情况下，是用实际类名来表示。这种映射不遵从关系型设计的规范化规则。因此，表可能非常大，特别是当一个叶子类定义了大量属性的时候。当用到抽象父类时，这种映射最为有利，即不存在抽象类的实例，虽然它定义的属性出现在每个派生类中。统一方法的一个例子如图 9-5所示。

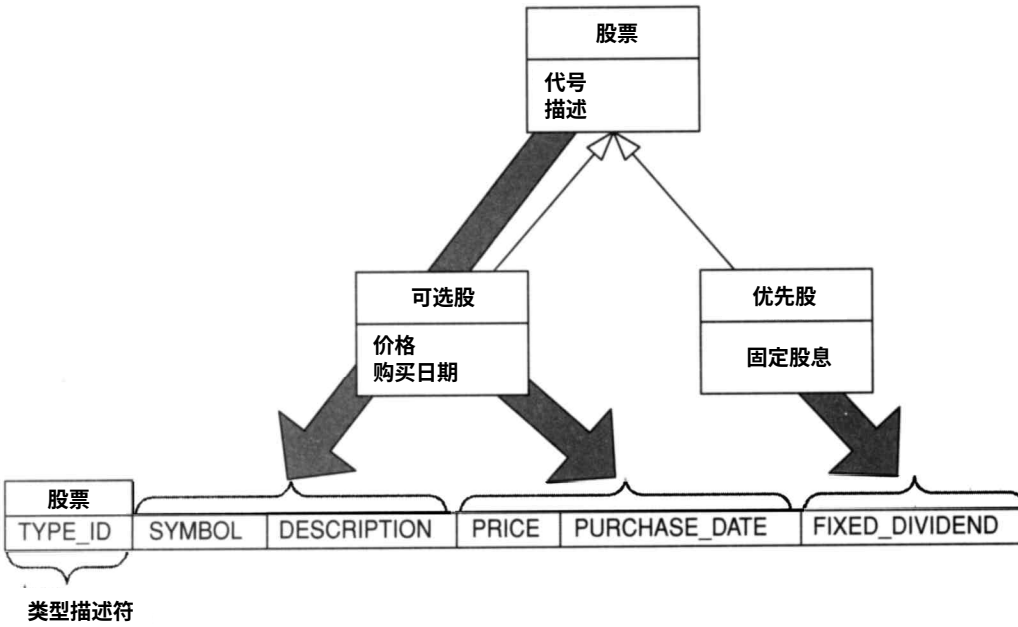

图9-5 统一

9.3.5 自顶而下和自底而上

一般而言，对象/关系映射有两种方式：把对象储存到关系表中，以及通过对象访问关系表。通常，这两种方法分别被称为自顶而下和自底而上。自顶而下方法的思想是从对象模型开始着手，并由此派生出关系型数据框架。自底而上则从关系型数据框架开始，并由此派生出对象模型。

自顶而下方法注重于事务逻辑，例如用 UML来描述通用模型。然后可以从对象模型派生出关系型数据框架。必须注意，设计派生的关系型数据框架时，要保证性能和数据完整性。在类和表之间并不总能够用简单的一对一映射。通常，数据框架必须适应关系型世界的特殊性。例如，规范化可能就是一个问题。关系型数据框架常常被规范化，以避免冗余和更新异常。但是，极端规范化常常可能导致低性能、高度零碎的数据和急剧增加的复杂性。这正是很多人企图避免太严格的规范化的原因。

在关系型数据框架已存在，并且改变框架很不可取的时候，自底而上方法是最常用的。在

这种情况下，我们分析关系型框架，并试图派生出既反映业务逻辑又反映关系型框架的对象模型，假如关系型框架已被规范化，这可能特别困难。这时，通常必须把多个表映射成一个类，以便一个对象实例能代表数据行的一个集合。这种方法通常称为封装，因为一个类封装了多个表。

支持遗留系统

如果要建立的应用程序依赖于访问已存在于关系数据库中的数据，那么会发生什么呢？首先，充分观察业务领域，包括从业务对象模型到关系型框架的新映射。这种方法意味着数据要在各种(所有权)的数据储存中复制。但这甚至不是一对一复制；相反，需要对原数据作转换，以把它复制到新应用程序使用的数据库中。甚至更糟的是，如果新应用程序要使它的一些数据用于任何已有的应用程序，那么可能需要编写特别的数据通路。这种方法引起的数据管理的复杂性，使大多数试图用它的人失去信心。不可避免地需要采用其他方法，使得新应用程序可以从已有的框架中访问 (至少一些)数据。我们不再定义从业务对象模型到持久数据框架的映射，而是需要用一种混合的处理方法，先用逆向工程从关系型框架产生出对象表示，然后建立封装器来提供自己的业务对象模型和那些类之间的基本映射。这个抽象层在应用程序本身的对象模型和遗留应用程序的关系型框架之间提供了一个缓冲，方便了底层数据的透明访问，但这种方法有一定的代价。抽象层隐含地理解了两个模型，因而任一个系统的改变都需要它作相应的改变。不但抽象代码难以维护和易于出现错误，而且会影响性能，尤其当应用程序业务对象只是代表表示关系数据的数据访问对象时。总的来说，任何额外的重定向都会降低应用程序的性能和灵活性。

9.3.6 对象/关系代码生成

实现对象/关系映射是一项高度重复的工作。代码生成器很适于这个工作。对象 /关系代码生成器可以从关系数据库框架开始 (自底而上)，或者提供定义对象模型的工具，并由此生成 O/R访问代码(自顶而下)。这两种情况下，代码生成器都需要元信息，即关于要生成的映射的更详细的信息。

对象/关系代码生成器工具的例子包括 Persistence Software 的Persistence和ONTOS的Integrator for C++，以及CocoBase for Java。下面例子假定 Stock和StockPrice类的对象模型用Persistence工具来定义，把它们映射成 STOCK和STOCK_PRICE表。Persistent工具为这种自顶而下方法提供了 GUI，可用于图形化地定义对象模型和用有关类的信息来注解。基于这种信息，Persistence工具自动地生成C++类的集合。这个例子用于简单的应用程序，如使用生成的类来查询股票价格并打印结果。把这个例子和 9.2.3节“ OO SQL封装类 ”中给出的那个例子比较是有启发意义的。

```
// Persistence O/R example
StockPriceCltn result =                                    (1)
    StockPrice::querySQLWhere ("price>50");                (2)

for (int idx=0; idx<result.extent(); idx++)                (3)
{
```

```
        StockPrice* sp = (*result)[idx];                    (4)
        cout << sp->getSymbol() << ": "
           << sp->getDate() << ", "
           << sp->getPrice() << endl;                       (5)
}
```

在(1)中，声明所生成集合类的一个实例，以后用它来保存查询结果。在 (2)中，通过作用于StockPrice类的querySQLWhere函数执行查询。这个类是由 Persistence 工具生成的，可用于访问STOCK_PRICE表，这个类的实例表示表中的一行。在 (3)中，定义for循环来遍历结果集。在 (4)中，使用生成的方括号运算符来访问结果集的元素。其结果是一个股票价格对象，表示查询结果中的一行。最后，在(5)中访问特定的股票价格，并打印一些细节信息。注意， StockPrice类为每个属性提供了生成的访问函数和更改函数 (get/set方法)。

对象粒度

大多数商用 O/R代码生成器实现了非常直接的对象 /关系映射，每个表直接映射成一个类。这种过分单纯化方法的结果通常是相当细粒度的和以数据为中心的对象。为实现业务逻辑，通常有两种选择。可以向产生的类增加方法，以表示一些业务功能。或者像前面的遗留系统中提出的那样，可以用这样细化的数据对象来实现一个额外的抽象层，并在这层实现业务逻辑。第二种方法通常更可取，尤其在数据高度规范化的情况下。

9.3.7 对象/关系映射的局限

不要对直接对象 /关系映射的结果感到失望。回想对象 /关系映射的基本原理：我们尝试为更基本的问题寻找工作区。我们并非正在为支持对象而设计的体系结构上构筑面向对象概念。问题是，只要数据库没有必要的内建的面向对象基础，我们就必须处理工作区，这些工作区并不能提供像从底层就支持面向对象方案那样的无缝性和性能。如果系统确实需要无缝的对象访问和高性能的复杂对象图导航，那么，连接 RDBMS就是一个错误的选择，对象 /关系映射也不会有太大的帮助。相反，应该考虑使用ODBMS，或者近期出现的对象/关系数据库之一。

对象/关系映射导致了细粒度的、以数据为中心的对象。对于实现简单的 CRUD(创建/读取/更新/删除)逻辑的应用程序部分来说，直接的对象 /关系映射可能非常有用，因为它比嵌入 SQL等更容易使用。

但是，使用自底而上方法经常导致产生一些无意义的对象，即和实际的实体没有关系的对象。这时，我们通常需要在基本的对象 /关系映射上增加若干层。另一方面，从关系的观点来看，自顶而下方法通常过于简单，例如，它通常不能解决规范化和关系型框架优化策略引起的问题。

在以后关于CORBA和数据库集成的章节中，将探讨 CORBA IDL接口如何有助于封装复杂的关系型数据库框架，并通过以行为为中心的 CORBA业务对象输出服务。

9.4 对象数据库

对象数据库结合了 C++等面向对象编程语言的语义和传统数据库系统的数据管理查询工具。

当对象数据库管理系统(ODBMS)与面向对象编程语言集成时，它应该能够直接支持此语言的语义。也就是说，如果编程语言支持继承、封装和多态， ODBMS也应该支持这些特性。因为现在数据的“内存中”表示和“持久性”表示是一致的，所以储存时对象转换的需要就大大减少了。结果，前面章节讨论的阻抗失配通常就不再存在，或者非常小了。特别是，编程语言中创建的对象间关系可以在数据库系统中透明的表示。对象数据库不仅为面向对象编程提供了方便的持久性机制，还增加了并发、事务和查询处理等额外的数据库特征。由于有内存中对象和它们的持久性状态间直接映射的优点，可以期待更多的 ODBMS和面向对象语言一起使用。虽然ODBMS系统的市场不断扩大，但它还不能取代传统的关系型系统。其中一个原因是很多新应用系统使用或增加了已有的关系型系统。但是，在需要管理和遍历极端复杂对象图的很多问题领域，如CAD/CAM系统中，ODBMS已找到了稳定的合适位置。

ODMG

对象数据管理组(ODMG)的定义发展了一系列的所有对象数据库厂商都认可和实现的标准。在ODMG以前，缺乏对象数据库标准限制了它们更广泛的使用。关系数据库的成功不仅是因为比以前有更高层次的数据独立和更简单的数据模型，它们的成功在很大程度上是来自提出的标准。由于接受SQL标准，所以允许了一定程度的系统间可移植性和可互操作性，简化了学习新的关系型DBMS，并代表了关系型方法的广泛认同。 ODMG认识到这些，并着手定义和精炼对象数据库规范。当前ODMG规范版本是2.0，包含了大量重要特征，如 Java持久性标准(扩展了C++和Smalltalk的已有标准)、元对象接口和对象交换格式。 ODMG标准的主要组件有：ODMG对象模型、对象定义语言(ODL)、ODL的不同语言绑定，以及对象查询语言(OQL)：

- ODMG对象模型 ODMG对象模型是基于OMG对象模型的。通过向核心 OMG模型增加ODBMS轮廓，ODMG模型增加了对关系、查询、事务和持久性的支持。
- ODL对象定义语言 可用于ODMG兼容的数据库的规范语言之一就是ODL(对象定义语言)。ODL有点类似基于SQL的、广泛用于关系数据库的数据定义语言(DDL)。但是，像OMG的IDL一样，它不是完整的编程语言，而是接口标记的一种规范语言。
- ODMG语言映射 ODL提供了映射的声明部分。 ODL映射成多种编程语言，如 C++、Smalltalk和Java。在开发者看来，只有一种语言，而不是两种分别的语言。对于 C++，ODL和C++之间的映射表示成类库和标准C++类定义语法的一个扩展。
- OQL(对象查询语言) OQL是类似SQL的声明语言，为查询数据库对象提供了丰富的环境，包括对象集合和结构的高层原语。 OQL提供了SQL-92 SELECT语法的超集。这意味着大多数运行在关系型DBMS的SQL SELECT语句可以用相同的语法和语义工作于ODMG集合类上。OQL还包括了对象扩展，以支持对象标识、复杂对象、操作调用和继承。 OQL的查询能力包括在ODMG语言绑定中调用操作，OQL也能从ODMG语言程序内部调用。

9.5 数据一致性

显然，我们希望对象持久性服务可以使数据能够以对象形式被访问，并为导航关系和执行查询提供高级的机制。但是，期望的最根本的服务还是保持数据的一致性状态。对象持久性服

务必须提供确保数据一致性的机制，甚至在并发访问同一数据的情况下。

下面探讨和数据一致性有关的问题，包括事务、并发控制和数据同步。

9.5.1 事务

保证数据一致性的最自然方法是用事务的概念。几乎所有的数据库管理系统，包括 ODBMS，都是基于事务概念的。事务是逻辑上组合以形成一个原子行为的一系列操作，它把系统持久数据从一个一致性状态转换到另一个。这意味着，在事务执行期间系统状态可能不一致，但事务机制保证事务结束时状态还是一致的。因为在事务执行期间系统状态可能不一致，所以保持事务彼此隔离是重要的，以便能看到它们内部的不一致状态。这也称为事务的串行化(serialization)，因为即使事务并发执行，事务机制也必须保证其效果和串行执行时完全一样。

这些只是事务支持的一些属性，在第 12章“分布式事务处理”中将对其详细探讨。

9.5.2 并发控制

为保证事务串行化， DBMS必须提供使并发事务彼此隔离的机制，也叫并发控制。商用 DBMS上最普通的并发控制技术是加锁。如果事务 T2要访问特定数据项，在访问前它必须得到此数据项的锁。如果另一个事务 T1已经得到了此数据项的锁，并且和 T2需要的锁不相容，就出现了加锁冲突，必须作适当处理。两类最普遍的锁是读锁和写锁。但 DBMS可能提供其他的锁类型。并发控制和加锁将在第 13章“用户会话”中讨论。

隔离级别

加锁通常代价很大，所以隔离经常在性能、并发性和数据一致性之间折衷。为能灵活调整以反映性能、并发性和数据一致性的需求级别，大多数数据库管理系统支持不同级别的隔离。

ANSI/ISO SQL92标准定义了四级隔离。几乎所有商用 DBMS都提供至少一种隔离级别；很多甚至能为不同事务配置使用不同隔离级别。这四种隔离级别中每种都和并发执行的事务之间出现的三种现象之一有关。这三种现象是：

- 脏读取 事务读取的数据已被另一个没提交的事务更改。
- 无重复读取 事务重读先前已读过的数据，并发现在两次读取之间数据已被另一个已提交的事务更改或删除。
- 幽灵读取 事务重复执行查询，返回满足特定查找条件的行集合，并发现另一个已提交的事务已经插入了满足这种特定条件的额外的行。

基于这三种现象， SQL92标准定义了四级隔离。根据事务正在运行的隔离级别，可能不允许这些现象出现，或者允许这些现象的一部分或全部出现：

隔离级别	脏读取	无重复读取	幽灵读取
不提交读取	可能	可能	可能
已提交读取	不可能	可能	可能
重复读取	不可能	不可能	可能
串行	不可能	不可能	不可能

必须注意，只有串行隔离级别提供完全的隔离——即充分保证数据一致性。所有其他的隔离级别都为了增加并发性和有利于性能而接受可能的数据不一致。

9.5.3 数据同步

和数据一致性密切相关的是数据同步。考虑一个场景：用户有一个台式电脑和膝上型电脑，用来读电子邮件和编辑文档。他如何在这两台计算机之间使数据同步呢？文档的标准方法是手工来回地拷贝文件。这常常导致出现有微小差别名字的很多临时文件。另一个问题是，电子邮件程序可能使用专有的文件格式，可能不支持两台计算机之间的简单同步。虽然有很多工具可以帮助解决这些特殊问题，但是这个讨论说明了数据同步的一般问题。

下面从对象持久性的角度探讨数据同步的两个重要部分：缓冲和复制。

1. 缓冲

ODBMS相对于一般RDBMS的主要优点是它们提供了成熟的、特定于业务领域的缓冲机制。RDBMS通常只在数据库服务器提供缓冲，并且这些缓冲是基于表结构而不是基于业务领域逻辑的。相反，ODBMS通常在数据库客户机端提供缓冲机制，并且允许在应用层对缓冲机制进行微调。一个通用技术是定义逻辑上关联的对象簇，它们可以被缓冲机制成组地管理。这是面向对象方式下结构化数据的一个主要优点，因为它使业务对象之间的依赖关系可以在缓冲策略中得以反映。这些特定于应用程序的缓冲策略是 ODBMS在操纵复杂关系图的导航时大大优于RDBMS的一个主要原因。

在9.3.6节“对象/关系代码生成”一节讨论的一些对象/关系工具为数据库客户机端提供了对象缓冲。但是，回想在第 9.3.7节“对象/关系映射的局限”中的讨论： O/R工具提供了工作区，只要数据库服务器不直接支持持久对象的管理，基于 O/R的缓冲结果就不太可能类似于真正的ODBMS。

最后，一个有趣的问题是缓冲和事务相关的程度。一些缓冲机制只在事务周期内缓冲数据。例如，大多数ODBMS只在一次事务上下文内允许访问对象，并在这次事务结束时从本地缓冲区中清除所有对象。

但是，有一些缓冲策略跨事务边界维持缓冲。这时，就必须引入特殊的同步机制来保证数据一致性。通常，在想修改或从缓冲区写回元素时，可以用时间戳机制来比较缓冲版本和数据库版本。既然通常不处理单个缓冲区，而是多个缓冲区，就需要考虑某种缓冲区同步机制。如果数据在数据库中改变了，我们要向相关缓冲区通知这个事件，以便它们能更新缓冲区入口或者使缓冲区入口无效。这样，就减少了由于缓冲区不一致造成的潜在错误数量。

2. 复制

复制的主要目的通常是改善系统可用性和响应时间。无论是持久数据或者活动服务器进程都可以复制。下面讨论数据复制。一个重要问题是复制的数据应该以怎样的程度同步。在 FTP镜像的例子中，我们并不期待复制镜像总能和主服务器完全同步——通常镜像和主服务器之间在一天或者一个星期中同步一次就足够了。

另一方面，我们可能已经复制了要求所有的镜像被实际同步复制的系统。这通常需要分布式事务处理，细节在第 12章讨论。一种常用方法是主/从复制，即在主拷贝上实行更新处理，并

从这里对从拷贝进行更改。复制是一个很复杂的主题，在这里不深入探讨。关于复制的更详细讨论参见Bernstein (1997)。

9.6 小结

本章覆盖的主题包括对象/关系阻抗失配，以及与关系数据库系统有关的问题（例如对象/关系映射方法的局限性），对象数据库和数据一致性。这些主题并不是CORBA特有的，但是和很多CORBA系统有关。理解这些主题有利于阅读本部分其他章节，包括CORBA系统内的数据库集成、事务处理、分布式事务处理和用户会话。