计算机科学1994Vol.21 №.4

面向对象数据库系统：诺言·现实·前景(1)

Won Kim

摘 要 过去十年里，面向对象技术已逐步用于程序设计语言、用户界面、数据库、操作系统、专家系统、等等。尽管如此，关于面向对象数据库系统、扩充了面向对象功能的关系系统、甚至这些系统的必要性，在用户、商业杂志及卖主中仍有许多谬传和混淆。本文旨在回顾面向对象数据库系统的承诺，分析其现实性，以及如何通过与关系技术联合起来以履行其诺言。

1. 定义

今天使用的面向对象技术包括面向对象程序设计语言(如 C++ 和 Smalltalk)、面向对象数据库系统、面向对象用户界面(如 Macintosh 和 Microsoft 窗口系统，Frame 和 Interleaf 桌面印刷系统)，等等。面向对象技术是一种用户能够用那些基于"面向对象概念"设施的技术。为了定义"面向对象概念"，我们必须首先理解什么是"对象"。

术语"对象"意味着一个"数据"和"程序"的联合体，它表示某个现实世界实体。例如，考虑一个名叫 Tom 雇员，假设 Tom 是25岁，他的工资为 $25,000。那么在计算机程序中可把 Tom 表示为一个对象，这个对象的"数据"部分将是(name：Tom，age：25，salary：$25,000)。该对象的"程序"部分可能是一组程序(雇用，检索数据，改变年龄，改变工资，解雇)。数据部分由各种类型的数据组成，对于"Tom"这个对象，name 用字符串，age 用整数，salary 用货币型。但一般而言，任何用户定义的类型(如 Employee)也可以使用。在"Tom"这个对象中，name、age 和 salary 称为对象的属性。

我们常常说一个对象"封装"了数据和程序，这意味着用户不能看见对象"封闭壳"的内部，但可以通过调用对象的程序部分来使用对象。这和传统程序设计中的过程调用没多大区别；用户调用过程时为输入参数提供数值，并从输出参数中得到结果。

术语"面向对象"粗略地说就是对象封装和继承的组合。术语"继承"有时也称为"重用"，继承的大致意思是一个新对象可以通过扩充一个已存在的对象来创建。现在让我们更精确地理解术语"继承"。一个对象有一个数据部分和一个程序部分，数据部分的属性相同且程序部分也相同的所有对象合称为一个类(或型)。类被按某种形式安排以便某个类可以从某些其它类那里继承属性和程序部分。

Tom、Dick 和 Harry 都分别是一个 Employee 对象，每个这种对象的数据部分都包括属性 Name、Age 和 Salary。每个这种 Employee 对象都有相同的程序部分(雇用、检索数据、改变年龄、改变工资、解雇)。程序部分的每个程序称为"方法"。术语"类"指具有相同属性和方法的所有对象的集合。在我们的例子中，Tom、Dick 和 Harry 对象属于类 Employee，因为他们都有相同的属性和方法。这个类可用作任何对象的一个属性类型。当前系统中只有一个类 Employee，有三个对象 Tom、Dick 和 Harry 属于此类。

现在假设一个用户希望创建两个售货雇员 John 和 Paul，但售货雇员有一个额外的属性，假定是 Commission，售货雇员不能属于类 Employee，不过用户可以创建一个新的类，叫 Sales-Employee，使之可以重用类 Employee 的所有属性和方法，并且属性 Commission 可以加到 Sales-Employee 中。用户通过说明类 Sales-Employee 是类 Employee 的一个"子类"来实现。用户现在能接着创建两个售货雇员作为属于类 Sales-Employee 的对象，即能创建新类作为已存在类的子类。一般而言，一个类可能从一个或多个已存在的类中继承，类的继承结构组成一个有向无圈图(DAG)，不过为简单起见，继承结构被称为"继承层次"或"类层次"。

当封装和继承一起发挥作用时，面向对象概念的威力就体现出来了。

——因为继承使得不同类共享一组相同的属性和方法成为可能，同一段程序就可运行于属于不同类的对象上。这是当今桌面印刷系统和窗口管理系统提供的面向对象用户界面的基础。一组相同的程序(如打开、关闭、删除、创建、移动等)适用于不同类型的数据(图象、文本文件、声音、目录等)。

——如果用户定义很多类，而且每个类有很多属性和方法，那么共享属性和程序的好处将是极大的。属性和程序不必从头定义和编写。新的类可通过对已存在的类增加属性和方法来创建，而不用修改原有类的属性和方法，因此降低了在原有类中引入错误的机会。

2. OODB 的诺言

面向对象程序设计语言(OOPL)提供了创建类的设施，用以组织对象、创建对象、把类组织成一个继承层次，使得子类能从超类中继承属性和方法，以及调用方法来访问特定的对象。类似地，一个面向对象数据库系统(OODB)应该提供创建类的设施，用以组织对象、创建对象、把类组织成一个继承层次，使得子类能从超类中继承属性和方法，以及调用方法来访问特定的对象。除此之外，因为 OODB 是一个数据库系统，它必须提供当今关系数据库系统(RDB)提供的那些标准数据库设施，包括检索对象的非过程性查询设施、自动查询优化和处理、动态模式改变(改变类定义和继承结构)、存取方法(如 B^+ 树索引、可扩充散列、排序等)的自动管理以提高查询处理性能、自动事务管理、并发控制、从系统故障中恢复、安全和授权。包括 OOPL 在内的程序设计语言在设计时都只考虑了一个用户和一个相对很小的数据库，但数据库系统是为多个用户和超大规模数据库设计的，因此性能、安全和授权、并发控制、动态模式改变就成为重要的问题。此外，数据库系统用于准确地维护重要数据，其事务管理、并发控制和恢复是重要的设施。

数据库系统是一个系统软件，其功能是从用某种宿主程序设计语言编写的应用程序中调用的。就这一点而言，我们可以区分两种不同的设计 OODB 的方法。一种是存储和管理由用 OOPL 编写的程序所创建的对象。例如，当前存储和管理由 C++ 或 Smalltalk 程序生成的对象。当然，一个 RDB 也能用于存储和管理这样的对象，不过 RDB 不理解对象，尤其是方法和继承，因此需要编写所谓的"对象管理器"或"面向对象层"软件来管理方法和继承，并把对象翻译成关系(表)的元组(行)。对象管理器和 RDB 的联合实际上就是一个 OODB(当然其性能较差)。

另一种方法是允许非 OOPL 的用户可以使用面向对象设施。用户可以创建类、对象、继承层次等，数据库系统将存储和管理这些对象和类。这个方法实际上是把非 OOPL(如 C、FORTRAN、COBOL 等)转变成面向对象语言。事实上，C++ 已把 C 转变成 OOPL，CLOS 已在 Common LISP 中增加了面向对象编程设施。用这种方法设计的 OODB 当然能用于存储和管理由 OOPL 编写的程序所创建的对象。虽然需要编写一个翻译层来把 OOPL 对象映射成数据库系统的对象，但该层远没有 RDB 所需要的对象管理层复杂。

尽管 C++ 日益普及，但仍不是数据库应用编程人员正在使用或将要使用的唯一编程语言，基于这个事实，程序设计语言和数据库系统之间还有一条鸿沟，因此第二种方法是数据库系统将面向对象概念的能力赋予数据库应用编程人员的更为现实的基础。不管哪种方法，只要运用得当，OODB 都能使数据库应用编程人员的生产力和应用程序的性能产生飞跃。

产生技术飞跃的一个源泉是数据库技术进化历史中面向对象概念第一次使数据库设计和编程成为可重用的。面向对象概念主要用于降低开发和改进复杂软件系统或设计的难度。封装和继承允许属性(即数据库设计)和程序的重用，作为建立复杂数据库和程序的基础。也正是这个目标在过去三十年中促进了数据管理技术从文件系统到关系数据库的发展。OODB 有潜力达到降低超大型复杂数据库的设计和改进难度这一目标。

产生技术飞跃的另一个源泉是封装和继承这些面向对象概念隐含着强有力的数据类型设施。这些设施事实上是消除 RDB 的三个重要不足的关键。现在罗列如下，然后再更详细地讨论。

——RDB 强迫用户用多个关系的元组表达如材料单之类的层次数据(或复杂嵌套数据，或复合数据)。一开始这点就很糟，再说，为了要检索这种分散在多个关系的数据，RDB 必须借助于连接，连接操作一般很费时。OOPL 中对象的一个属性的数据类型可以是基本类型，也可以是任意的用户定义类型(类)。一个对象的某个属性可能取值为另外的对象，这一事实自然地导致嵌套对象表示，而嵌套对象表示反过来允许自然地(即层次地)表示层次数据。

——RDB 为用户提供一组基本的、内部的数据类型作为关系的列的值域，但不提供增加用户定义数据类型的任何手段。内部数据类型基本上就是所有的数和短符号。RDB 不允许增加新的数据类型，因此为了增加任何新的数据类型都常常需要对系统体系结构和代码进行大手术。向一个数据库系统增加一个新的数据类型意味着允许把它用作一个属性

的数据类型，即要存储该类型的数据，查询和更新这种数据。OOPL 中的对象封装并不对对象的数据部分所可能用到的数据类型强加任何限制，即数据类型可以是基本类型或用户定义类型。此外，新的数据类型还可以创建为新类，甚至作为已存在类的子类，继承它们的属性和方法。

——对象封装是数据库中程序以及数据的存储和管理的基础。RDB 现在支持"存储的过程"，即允许程序用某种过程性语言编写并存入数据库中以备以后装载和执行。不过，在 RDB 中存储的过程并不和数据封装，即它们不和任何关系或关系的元组相关联。而且，由于 RDB 没有继承机制，存储的过程不能自动地被重用。

3. OODB 的现实

有大量商品化的 OODB，包括 Servio 公司的 GemStone，OWTOS 的 ONTOS，Object Design 公司的 ObjectStone，Objectivity 公司的 Objectivity/DB，Versant Object Technology 公司的 Versant，Intellitic International（法国）的 Matisse，Itasca Systems 公司的 Itasca（MCC 的 ORION 原型的商品化版本），O_2 Technology（法国）的 O_2。这些产品都支持一个面向对象数据模型。具体来说，它们允许用户创建一个带属性和方法的新类，使类从超类中继承属性和方法，为类创建带有唯一对象标识的实例，单个地或集合地检索实例，以及装载和运行方法。

这些产品早在1987年就已进入市场。但它们大多数只是处于评估和初步的原型应用开发阶段，还没有当真地用于许多紧要任务的应用。此外，相当大量的产品拷贝是免费赠送试用，人为地提高产品安装的总数。当前全世界 OODB 市场规模总共估计只有2—3千万美元，占3亿美元的世界数据库产品市场的一小部分。可以肯定，过去几年是面向对象技术，尤其是面向对象数据库技术的孕育时期，而且当前 OODB 瞄准的技术市场和 OOPL 市场是以前并不依赖于数据库系统的新市场。然而，由于初期的（很大程度上说来，还有当前的）OODB 产品不够成熟，严重地使其在紧要任务的应用中没受到重视。

3.1 局限

作为持久存储系统的局限

大多数当前的 OODB 的一个关键目标或产品特色是支持一种合一的编程和数据库语言，即同时作为通用的程序设计和数据库管理的一个语言（如 C++或 Smalltalk），这个目标是由于当前的应用程序是用一个通用的程序设计语言（大多数是 COBOL，FORTRAN，PL/Ⅰ，或 C）写的，而且数据库管理功能被嵌套在一个数据库语言（如 SQL 关系数据库语言）的应用程序中。通用的编程语言和数据库语言在语法和数据模型（数据结构和数据类型）上有很大差别，为了编写数据库应用程序必须学习和使用两种完全不同的语言，这已常被认为是一个主要的麻烦。因为 C++和 Smalltalk 已经包括了定义类和一个类层次（即数据定义）的设施，所以这些语言是合一的编程和数据库语言的一个好的基础，早期 OODB 的大多数卖主采取的第一步就是使类和类的实例持久化，即把它们存储在外存上，而且使它们即使在定义和创建它们的程序终止之后仍可访问。

当前用于支持 OOPL 的 OODB 在对象的定义和使用上施加了各种限制，特别是，大多数系统对持久数据与非持久数据有不同的对待（例如它们认为一个持久对象包含了一个非持久对象的 OID 是不合法的），因此要求用户显式地说明一个对象是否是持久的，而且它们不能使某些数据类型持久化，因此禁止其使用。

作为数据库系统的局限

当前大多数 OODB 产品不成熟的更严重原因是缺乏那些数据库系统用户已经熟知的，因而也是希望提供的基本特征。这些特征包括一个完全非过程性的查询语言（包括自动查询优化和处理）、视图、授权，动态模式变化、以及参量化性能调整。除了这些基本特征之外，RDB 还提供支持触发器、元数据管理、UNIQUE 和 NULL 等约束，而大多数 OODB 不支持这些特征。

——大多数 OODB 缺乏查询设施。也有少数系统确实提供了重要的查询设施，但其查询语言并不与 ANSI SQL 兼容。一般情况是，查询设施不包括嵌套子查询、集合查询（并、交、差）、聚集功能和成组、甚至多个类的连接等，而 RDB 中完全支持这些设施。换句话说，尽管这些产品允许用户创建一个灵活的数据库模式并可在数据库中加入许多实例，但没有提供足够有力的手段从数据库中检索对象。

——RDB 支持视图作为存储的数据库的动态窗口。视图定义包括一个查询语句来说明用以构作视图的数据。视图被用作授权的单位。今天还没有一个 OODB 支持视图。

——RDB 支持授权，即允许用户向其它用户授予和回收读或修改他们创建的表或视图中的元组的权利，或者修改他们创建的关系的定义的权利。大多

数 OODB 不支持授权。

——RDB 允许用户用 ALTER 命令动态地修改数据库模式，可以向一个关系增加一个新的列，可以删掉一个关系，有时也可以从一个关系中删去一个列。但是，当前的大多数 OODB 不允许动态改变数据库模式，比如向一个类增加一个新的属性或方法，为一个类增加一个新的超类，从类中删除一个超类，增加一个新类，删除一个类。

——RDB 在处理用户发出的查询和更新语句时自动加锁和解锁。但当前的一些 OODB 要求用户显式地加锁和解锁。

——RDB 安装时允许系统管理员设置大量参数来调整系统性能。这些参数包括内存缓冲区数目，每一数据页为将来数据插入保留的自由空间的总数，等等。大多数 OODB 只提供有限的参量化性能调整能力。

由于上述缺陷，大多数这样的 OODB 产品都需要极大改进。我们可以肯定，这些产品的卖主将对他们现在的软件产品进行必要的改造，而不会从头开始。要将这些产品变成成熟的数据库系统，至少可与当今数据库系统所希望的数据库功能水平相媲美，其改变之大，不可能指望在三、四年之内使其达到紧要任务应用所要求的健壮性和性能。

把大多数当前的 OODB 提升为真正的数据库系统不仅有上面提到的一些主要技术困难，还有严重的观念上的困难。正如我们已经看到的，当前大多数 OODB 更接近于仅仅是某个 OOPL 的持久存储系统，而不象数据库系统。术语 OODB 并不是故意设计得来造成误解和混淆，因为 OODB 是用来管理用 OOPL 编写的程序所生成的对象的数据库。然而，数据库用户经过过去二十年的训练，自然认为一个数据库系统软件应该允许查询大的数据库以检索其中的小部分；不需要用户就如何处理给定的查询给出任何提示；允许大量用户同时读和修改同一个数据库；在多个并发用户以及系统出错情况下自动保证数据库完整性；允许一个数据库的某个部分的创建者向其他用户授予和回收检索其数据的权利；允许安装时通过调整各种系统参数来调整数据库系统性能；等等，由于这个原因，当前的大多数 OODB 不配使用术语 OODB。

当前大多数 OODB 实质上是对 OOPL 扩充了数据库函数的一个运行时刻库。必须带上适当的输入输出参数说明从应用程序调用这些函数。调用函数的语法构成与应用程序语言一致。如果当前的 OODB 要提升为真正的数据库系统，将需要对当前的数据库函数库进行大量的扩充以支持查询机制。当今的程序设计语言（包括面向对象语言）设计时根本没有考虑过数据库查询。一个数据库查询可能返回一组数目不定的满足用户说明的搜索条件的记录或对象。因此，应用程序必须设计得来一个个地处理返回的记录或对象的整个集合，直到没有剩下的为止。这正是数据库系统中引入游标机制之缘故。因此数据库查询的结果必须赋给某个数据结构并辅以能存储和一步步处理不定数目对象的算法。另外，还有必要提供说明嵌套子查询、对查询结果进行后处理（相应于成组、聚集函数、关联查询等）、集合查询（并、交、差）等设施。顾名思义，在一个合一的编程和数据库语言中，所有这些设施为编人员可用都必须在语法上与程序设计语言一致。换句话说，合一语言这一途径是任何数据库设施所需要的，只不过使这些设施为用户以不同的语法形式使用。此外，语法必须在低的过程性级别上与宿主程序设计语言一致，一个过程性语法对于普通用户总是难以学习和使用的。因此，还不清楚合一语言方法是否最终能提供什么好处，超过宿主程序设计语言中嵌入数据库语言的方法。

3.2 谬传

有许多关于 OODB 的谬传，其中多数全不分是非曲直，因为当前大多数 OODB 比起 RDB 来，连个象样的数据库系统都算不上，却侥幸贴上了"数据库系统"的标签。有些谬传出于技术演化现象，有些反映了纯粹派关心的，但在我看来并非现实有用的东西。

OODB 比 RDB 快10到100倍

OODB 卖主们经常宣称 OODB 比 RDB 快10到100倍，扬言有性能数字为据。假如不小心地考证，就可能造成误解。OODB 性能比 RDB 优越有两个由来。一是：OODB 中对象 X 的一个属性，如果其域是另外一个对象 Y，那么这个属性的值就是对象 Y 的对象标识(OID)。因此，如果一个应用已经检索到了对象 X，且现在要检索对象 Y，数据库系统就可以通过查找其 OID 来检索对象 Y。图1.a 例示了类 Person 的两个实例，以及类 Company 的两个实例，而类 Company 是类 Person 中的属性 Worksfor 的域。存在 Worksfor 属性中的值是类 Company 的一个对象的 OID，如果这个 OID 是一个对象的物理地址，则该对象就可以直接从数据库中取到；如果 OID 是一个逻辑地址，则对象可以通过查找一个散列表入口来取得（假设系统保持了一张从 OID 对应到其物理地址的散列表）。

Person

oid	name	age	salary	worksfor
115	John	25	25000	002
267	Chen	30	25000	001

Company

oid	name	age	president	location
001	Acme	15	Cohen	NY
002	UniSQL	3	Kim	Austin

图1.a OODB中的对象表示

Person

name	age	salary	worksfor
John	25	25000	UniSQL
Chen	30	25000	Acme

Company

name	age	president	location
Acme	15	Cohen	NY
UniSQL	3	Kim	Austin

图1.b RDB中的元组表示

当前的RDB只允许基本数据类型作为一个关系的属性的域。因此，一个元组的一个属性值只能是基本数据(比如一个数或字符串)，绝不会是另外的元组。如果一个关系R_2的元组Y逻辑上是关系R_1的元组X的一个属性A的值，则存在元组X的属性A中的实际值是关系R_2的元组Y的属性B的一个值。如果一个应用已经检索到元组X，且现在想检索元组Y，则系统必须有效地执行一个查询来利用元组X的属性A的值扫描关系R_2。图1.b是图1.a中的面向对象数据库在RDB中的一个等价表示。关系Person中的属性Worksfor的域是基本数据类型String。如果一个应用已经检索到了Person中的关于"John"的元组，并且想检索Company中关于"UniSQL"的元组，就需要提出一个查询来扫描Company关系。假想Company关系有成千上万个元组，如果在关系R_2(Company)的属性B(Name)上没有维护索引，则为了找到元组Y(关于"UniSQL")，必须顺序搜索整个关系R_2。如果在属性B上维护了索引，则检索元组Y就可以和OODB中利用散列表查找时差不多一样快，其效率比OODB中用物理地址实现OID(因而不要求任何散列表查找)低不了多少。

OODB性能比RDB优越的另一个由来是：大多数OODB在把对象装入内存时将存在对象中的OID都转换成内存指针，假设对象X和Y都已装入内存，而且作为对象X的属性A的值存储的OID已被转换成内存中指向对象Y的虚拟内存指针，于是从对象X到对象Y的导航，即通过对象X的属性A的值检索对象Y，实际上就变成了一个内存指针查找。图2.a说明了类Person和Company的对象的数据库表示。图2.b说明了相同对象的内存表示。存储在Person的对象的Worksfor属性中的OID已被转换成了内存地址。假想成百上千个对象已被装入内存，并且每个对象包含了指向一个或多个其它内存对象的内存指针；再假想从一个对象到其它对象的导航会反复进行。由于RDB中不存储OID，它们就不能在一个元组中存储指向其它元组的内存指针。通过驻留内存的对象进行导航的设施是RDB根本缺乏的，由此造成的性能缺陷不可能简单地在内存中利用缓冲区空间来抵消。因此，对于那些要求通过装入内存的相关对象来反复导航的应用，OODB能极大地超出RDB。

如果所有的数据库应用都只要求对数据库对象的OID查找或者在内存对象之间进行内存指针的追踪，那么OODB比RDB的性能好2到3个数量级并非毫无根据。可是大多数要求OID查找的应用也会要求数据库存取和修改，RDB就是设计得来满足这种要求的。这些要求包括大量数据库装载；创建、修改和删除个别的对象(一次一个)；从一个类中检索满足一定查找条件的一个或多个对象；连接多个类(我们马上就会看到)；事务提交；等等。对于这些应用，OODB并不能提供任何性能上的好处。事实上，即使对于图1所例举的数据库，如果应用的目的是获得满足一定条件的Person对象以及相关联的Company对象(例如所有年龄大于25岁且工资少于40000的人——即普通查询)，而不是从给定的Person对象得到一个特殊的Company对象(即一个简单导航)，则OODB可能一点也不会有任何性能上的优越性，这取决于怎样实现OID以及查询优化器在处理查询时是否利用OID。

Person

oid	name	age	salary	worksfor
115	John	25	25000	002
267	Chen	30	25000	001

Company

oid	name	age	president	location
001	Acme	15	Cohen	NY
002	UniSQL	3	Kim	Austin

图2.a 数据库中的对象表示

Person

addr	name	age	salary	worksfor
040	John	25	25000	020
080	Chen	30	25000	004

Company

addr	name	age	president	location
004	Acme	15	Cohen	NY
020	UniSQL	3	Kim	Austin

图2.b 内存中的对象表示

OODB 不再需要连接操作

OODB 明显地减少了类连接的需要(与 RDB 中关系的连接相比较而言);但是仍没有完全消除。在 OODB 中一个类 C 的属性域可能是另一个类 D,而在 RDB 中一个关系 R_1 的属性域不能是另一个关系 R_2,所以为了使一个关系的一个元组与某个其它关系的元组发生联系,RDB 总是要求用户显式地连接两个关系。OODB 用一个隐式连接,即获取的某个类中的对象 OID 是作为另一个类的属性值而存储的,以代替这种显式连接,图1中的例子说明的正是这点。在一个 OODB 中说明类 D 作为另外一个类 C 的一个属性的域本质上就是对类 C 和 D 之间的一个连接的静态说明。

关系连接是以关系中对应的一对属性值为基础使两个关系发生关联的一般机制。由于 OODB 中的两个类一般说来可能有对应的属性对,所以关系连接仍然有用,在 OODB 中亦必要。例如在图1中,类 Person 和 Company 都有属性 Name 和 Age。尽管类 Company 的 Name 和 Age 属性不是类 Person 的 Name 和 Age 属性的域,反之亦然,但用户仍可能希望这两个类能以它们的属性值为基础发生关联(例如找出所有 Age 小于他所在公司的 Age 的 Person 对象)。

对象标识不再需要关键字

对象标识已经受到了相当的关注。对象标识只不过是表示对象且同时保证每个对象唯一性的一种手段。一个 OID 并不包含任何额外的语义,即使 OID 给对象提供了唯一性,但也是由系统自动生成,通常对于用户是不可见,因此,(当用户不知道对象的 OID 时),它并不提供从一个大的数据库中获得所要求的特定对象的方便手段。对用户来说,用用户定义的关键字获取一个或多个对象更方便。例如,在图1所例举的数据库中,如果 Name 属性是一个主关键字,用户就可以给出一个搜索特定名字的查询来获取一个 Person 对象。

OODB 不再需要(非过程性的)数据库语言

这个谬传的由来是因为当前大多数 OODB 只提供有限的查询功能。OODB 卖主们把他们开发的注意力集中在数据库导航的性能和使对象持久化上,系统给用户提供了调用有限的数据库设施的必要命令,即一个过程性语言,从而调用一个数据库函数库。要把当前的大多数 OODB 提升为真正的数据库系统,尤其是要增加比得上 RDB 中所支持的所有查询机制,非过程性的查询语言是必不可少的,这难以回避,OODB 卖主们现在正在试图提供非过程性的查询语言,一般贴上 Object SQL 标签。

查询处理违背封装性

在 OOPL 中把数据和程序封装成一个对象的目的是强迫编程人员只能通过调用对象的程序部分来访问对象,而不让编程人员利用关于存储对象或程序部分实现的数据结构方面的知识。在处理一个查询的过程中,数据库系统必须读取对象的内容,抽取对象的某些属性中可能存在的 OID,检索出与那些 OID 相应的对象。对象纯粹派认为这违背了对象的封装,因为数据库系统考查了对象的内容,这个观点是不现实也是没有用的。首先,考查对象内容的是数据库系统,而不是任何普通用户。其次,考查存储在对象的属性中的值这一行为可以看作是调用了"get(或 read)"方法,这个方法是隐含地与每个类的每个属性联系的。如果必须不惜一切代价保留对象的纯粹性的话,那么所使用的每个数和字符串常量都必须明确地赋予一个 OID!

(转第5页)

Theoretical Computer Science(TCS) Vol. 14,(1981)
[17]Genrich, J. H., et al., System Modeling with High Level Petri Nets. TCS, Vol. 13,(1981)
[18]Lin, C., et al., On Stochastic High Level Petri Nets. Proc. 2nd Intl. Workshop. PNPM87(1987)
[19]李晓山.平均值演算.中国科学院软件研究所博士学位论文.(1993)
[20]Zhou, C. C., et al., A Calculus of Durations. Information Processing Letters 40(5),(1991)
[21]Merlin, P. M., et al., Recoverability of Communication Protocols. IEEE Transactions on Communications. Sept. (1976)
[22]姜旭升.柔性生产线非确定性加色网建模分析与控制.中国科学院自动化研究所博士学位论文.(1993)
[23]Frenkal, K. A., Robin Milner 谈程序语言及计算机科学.计算机科学, Vol. 20, No. 6,(1993)

(接第 36 页)

但还没听说过哪个 OOPL 或 OO 应用系统这样做过。

OODB 能支持版本和长事务

有一个普遍的误解:OODB 能以某种方式支持版本和长事务,言外之意是 RDB 中不能支持。尽管从关系到对象风范的转化确实消除了 RDB 中的关键缺陷,但并没有解决版本和长事务问题。面向对象风范不包括版本和长事务,跟关系数据模型没有两样。例如,C++或 Smalltalk 就不包括任何版本或长事务设施。

版本和长事务与 OODB 有联系,仅仅是因为 RDB 中缺少这些数据库设施,OODB 比 RDB 具有更强的数据建模和对象导航能力,故能更好地满足某些应用(如计算机辅助工程系统、计算机辅助授权系统等),恰好版本和长事务设施又是这些应用的基本要求。事实上,大多数 OODB 一点不支持版本和长事务,提供所谓的版本和长事务的少数 OODB 也只有一些简单的设施。

在 OODB 和 RDB 中支持版本和长事务同样困难。让我们考虑一下版本化的几个方面。如果要把一个对象版本化,通常可能需要维护一个时间戳和/或版本标识。这可以通过为时间戳和/或版本标识创建系统定义的属性来实现。显然,不难做到在 OODB 中对一个类版本化每个对象和在 RDB 中对一个关系版本化每个元组。类似地,也可以在数据库中维护版本派生历史。另外,诸如版本派生、版本删除、版本检索等版本设施可以通过扩充 OODB 和 RDB 的数据库语言来表达。

接下来让我们考虑长事务。事务简单地说就是一个独立的单位,是一组数据库读和更新操作。RDB 中实现事务是假设事务和数据库打交道只有几秒钟或更短,但用户需要长时间(几小时或几天)交互式地访问数据库,这个假设不再成立,长事务变得必要了。不管事务的时间长短,在多用户同时访问数据库时和在出现系统崩溃时事务只不过是一种保证数据库一致性的机制。OODB 和 RDB 的区别在于数据模型,也就是数据是怎样表示的(即 OODB 中的属性和方法,以及类和类层次;RDB 中的属性和关系)。应该明白 RDB 和 OODB 之间的风格差异并不能解决事务要解决的问题。

OODB 能支持多媒体数据

OODB 在实现管理多媒体数据所必需的功能方面比 RDB 具有更加自然的基础。多媒体数据广泛地定义为任意类型(数、短的字符串、Employee、Company、图象、声音、文本、图形、动画、含有图象和文本的文档,等等)和任意大小(1 字节、10K 字节、1G 字节等)的数据。OODB 允许创建和使用任意数据类型,这正是管理多媒体数据的首要要求。

但是,面向对象风范(即封装、继承、方法、任意数据类型——合起来或单独地)都不能解决存储、检索、以及修改超大型多媒体对象(如一幅图像、一个声音片段、一个文本文档、一幅动画等)的问题。OODB 必须解决与 RDB 完全一样的工程问题,允许 BLOB(二进制大对象)作为一个关系的列的值域,其中包括从数据库中对一个超大型对象的增量检索(页缓冲区一般不能保持整个对象),增量修改(对象中的一个小变动不应该造成整个对象的拷贝)、并发控制(多个用户应该能够同时访问同一个大对象)和恢复(作日志不应该引起整个对象的拷贝)。

(未完待续)