

第一部分 基础

第1章 概述

本章我们将简单讨论对象-关系模型，并概述数据库行业从 60和70年代的展开文件结构到数据库理论最近趋势的发展。我们也将针对统一建模语言（UML）、Oracle环境的概观以及 Oracle 8i中可利用的新特征作简单介绍。

1.1 对象-关系数据库的发展

当从事于信息系统技术的人们开始利用数据库工作时，数据库由一系列文件构成。我们主要用CODASYL语言的COBOL扩展来访问这些结构。在处理文件时，CODASYL给了我们利用FIRST_RECORD、NEXT_RECORD和LAST_RECORD这类命令的能力，这是我们第一次利用数据库游标。在处理数据文件方面，CODASYL使我们能够比通过阅读text文件更有效地工作。我们对这些结构使用索引、ISAM和VSAM文件，这类结构建立在链接表基础上且与索引结构分离。

第一次我们有了一个数据库管理系统（DBMS），在数据和程序之间具有了一定程度上的独立性。不幸的是，其中仍旧存在问题：程序仍然很大，甚至系统中微小的变化可能也需要数百（如果不是数千）小时的人工。你只需看看现在扰人的2000年问题，就可认识到这种系统的局限性。

关系系统以其巨大的理论优势胜过传统的以CODASYL为基础的系统。开发者能够把文件看作是简单的逻辑展开文件，所有的索引和SQL查询语句分析都可由关系DBMS（RDBMS）来进行。

不幸的是，仍然存在许多问题。查询运行得如此缓慢以致于开发者不得不取代缺省的SQL语法分析算法。80年代，要想获得好的性能是如此困难，以致于一些数据库设计者感觉到很有必要回到非规范化数据模型。

1.1.1 关系数据库

关系模型是一种非常精巧、清楚的模型，它已经为数据库行业提供了近20年的基础。关系理论中较少的概念已使关系数据库成为行业标准。关系数据库厂家已经能够主要从系统逻辑设计上孤立数据库物理实施的复杂性，由此提供了一个简单的应用开发者接口。

在过去的20年里，数据库已经作为一个行业发展成熟起来。许多人已经感觉到使建模环境更加丰富的必要性，这种环境更能适应向类属建模的发展。而且，我们认识到面向对象理论的一些概念能够被引入数据库行业，甚至可以带来更高的效率。

关系数据库的词汇量相对有限。我们通过精心设计展开文件来实现结构的设计，并将一些不同类型的索引放在那些文件的不同字段中。就访问而言，表之间的连接只被逻辑指明，

所以参照指针通过外码执行，表之间确实无显式连接。参照完整性约束仅仅是一些阻止特殊种类的无效数据放在数据模型中的代码段。当一个记录被插入、更新或删除时，可以将触发器加到表中，以便明确地执行一些操作，并且可将程序单元存储到数据库中。最后，可用簇表来改善性能。这些策略仍旧会限制我们以一个相当有限的方式思考。在关系数据库中，每个表事实上是一个独立的结构。

在逻辑实体关系（ER）模型中，有作为其他实体子集的实体。例如，领薪金者雇员是所有雇员的子集。同样，有基于其他实体的实体。例如 PO明细，它是基于购货单（Purchase Order）的。但是，在关系数据库范例中表达这种结构的能力是有限的。

看似矛盾的是，面向对象数据库保持了早期数据库理论的一些概念。正如在 CODASYL 时期，当把对象定位带入关系数据库世界时，人们发现遇到了“老朋友”，如链接表和指针。

重要的是不要忘记最初放弃链接表和指针的理由，要记住在 80 年代早期关系数据库出现之前数据库是怎样的。目前仍有一些高性能的数据库（至少要到 2000 年问题将它们消灭为止），它们的运行主要使用 COBOL 编写的 CODASYL、ISAM 和 VSAM 文件。CODASYL 数据库的修改通常需要花费几个月的精力。我们早期遇到过一个 COBOL 项目，其中有一个将一个字段的长度从 10 个字符改为 12 个字符的简单需求。这个转换花费了几百个小时的编程。现在，在关系数据库环境中，这样一个转换最多仅需几天时间。如果应用程序是用 Oracle Designer（或其他一些集成 CASE 工具）产生的，这样一个转换即使在一个大系统中也只需执行 1~2 天就能实现。

在关系数据库之前的时代，支持报表需求是很困难的。如果一个特殊的报表并非为了系统的最初设计规范而计划，那就能够容易地花费几周时间去编写一个新的报表（假定编写这个报表是可能的）。对基础性数据结构的修改是很麻烦的，似乎微小的变化就得需要整个大系统重新设计。虽不能断言所有这些问题已随着实体关系图（ERD）和关系数据库的出现而消失，但状况显然已得到改善。

早期的关系数据库看起来很像展开文件的前身，规范化被认为是一件纯粹满足好奇心的事。直到最后，至少通过第三范式，才发现规范化并不是一个坏想法。由于 80 年代的严重不规范化，在处理关系结构时遇到了一些处理展开文件时遇到的同样问题。数据结构的修改仍然很难且很昂贵，就像应用程序的修改一样。截止到 1990 年，大多数数据模型制作者已经认识到，当 80 年代的那些系统需要修改时，其严重的非规范化会引起许许多多的问题。规范化最终成为“流行时尚”。

90 年代中期，一些早期的面向对象的思考开始转向关系数据库。在 Oracle 世界里，这涉及到通过创建更抽象的结构构造模型，但仍然在关系数据环境中操作。在最近几年里，一些关系数据库已经包含了面向对象的思想。

现在，在大部分系统中看到某种级别的类属建模是很普通的。例如，把组织单位表示为一个简单的递归结构是行业标准，而不是用单独的表表示区域、部门和科室（或代表特殊组织的任一种结构）。

更抽象的模型正变得很平常。在最近的一次会议中，设计数据库调查表的人问如何支持一个有几百列的表，且每一份调查表都有几百个问题。被调查者中的一些人回答说应通过将答案放在一个单独的表中和将调查表的结构作为数据存入数据库来为调查表建立模型。一个无争议的解决办法因此而产生，面向对象的思想进入了数据模型。

最近，面向对象的发展更进了两步。首先，数据库本身已包括新的面向对象的特点。其次，

在UML中有了比在ERD中更新、更丰富的模型语言。如前所述，这些新概念实际上包括一些关系思想之前的老概念，如链接表和指针。这种新老集成提供了一个很好的机会，但必须谨慎地对待。现在能够构造操作更有效的关系结构，但如果不仔细，可能会失去一些由关系数据库范例而带来的结构灵活性。

1.1.2 对象-关系数据库

对象-关系数据库正向我们走来。近10年来，“面向对象”已成为计算机行业最时髦的语言。在计算机行业中随便捡起一本杂志就会看到一些有关面向对象的文章。面向对象通常被错用来作为可重用代码的同义词，前者的含义是指一组准确的、推理性的计算机科学结构，它能影响系统的每个方面，包括编程语言、工具和数据库。

这种技术的主要影响表现在编程上，面向对象有助于转变编程的过程。但是，应该认识到，并非每个C++程序都是面向对象的。系统中有关面向对象的最重要的方面就是设计组的思想。程序编写时没有可重用组件和标准是很平常的。前一段时间，我们核查了一个用C++编写的大系统的失败，其中，每个程序员都设定了自己的标准，整个系统只有少部分是利用面向对象的编程标准来编写的。

在一定程度上，面向对象的革命已经对数据库界产生了一定的影响。由主要数据库模型制作者支持的类属模型的发展如此强烈就是这场革命的反映。同样，在应用开发中，拥有可重用对象组件的模板的广泛应用正在成为主要编程厂家的规范。从这一点上说，为了更好地支持面向对象，Oracle现在正在改进RDBMS、Oracle Designer和Oracle Developer。

1.1.3 “对象-关系”的真正含义

如果你问六位专家“对象-关系”是什么意思，那你将会得到六个答案。现在仍然在试图结合关系范例和对象范例，结合两者将不是一件容易的事。

真正的对象数据库是市场的很小一部分，对象-关系数据库还处在发展阶段。问题是要确定在这个新的范例中究竟想得到什么。关系已有了20年的历史，在这期间，关系技术已成熟发展到了一个复杂的环境。最终，人们有了足够的参照完整性、存储程序和触发器，一些人已算出如何避免变化的表。同样，面向对象编程几乎和关系数据库同时起步，也已有了20年的历史。

我们不应该指望面向对象是灵丹妙药。面向对象编程还没有解决编程人员的所有问题。相反，对象结构要求用一种完全不同的方法考虑编程。面向对象将另外一层复杂性增加到环境中。许多编程厂家在向面向对象的转换过程中遇到了很大困难，而其他一些厂家则报告说他们的编程效率有了极大的提高。

面向对象数据库把对象的存在作为一种不易变的结构来考虑。宁愿用参照指针，也不愿通过参照完整性来连接对象。继承性是被完全支持的，访问对象严格限于通过使用相关的方法和操作来进行。

对象-关系系统正试图结合两者的优点，即对象标识符和主码。对象将通过对象引用和参照完整性联系起来。这种系统中还同时具备触发器和方法。表可独立构造，也可从一些父结构（称为类）继承属性和方法。

但是，很难将这些实施到一个关系模型中。目前，Oracle 8i有了对象标识符和方法，但只有有限的继承性。

从开发者的前景来看，最大的挑战是如何最好地利用这些新功能。关系数据库中的所有标准工具都已和对象数据库中的所有新结构加在了一起，那么我们应该在哪里和怎样利用这些新结构呢？性能、设计和维护的影响是什么呢？

本书将试图回答所有这些问题。然而，技术总是在朝前发展，在你读这本书时，给出的建议大概已经过时了。因此，这里不仅要试图给出最好方法，而且要讨论一般情况下设计和制作模型的最好方法，以便使这些工具能够跟得上我们的想象力，使你们愿意使用这个新技术。

1.1.4 创建对象-关系系统

我们不应该通过目前 Oracle 8i 的执行情况来判断对象-关系范例的有效性，Oracle 8i 中的对象相对较新。可以假设对象支持的水平和质量将会在今后几年里有重大改进。这里将要讨论一个对象-关系数据库的完全实施可能会为我们提供什么。

对象-关系范例具备哪些简单关系数据库中所不具备的优势呢？

能够以更快的速度和更少的花费创建数据库。

能够更容易地、以更少的花费维护数据库。

系统将会更加灵活和健壮，因而事务中的变化或新发现的需求能够更容易地、以更少的花费被修改。

这就是对象-关系方法的前景所在。从目前 Oracle 8i 中对象扩展的状态来看，还不能完全实现这些好处，但随着产品的成熟，我们确实期望这些前景会得到实现。

对象-关系方法究竟是利用什么来提供这些好处的呢？对象-关系方法是传统关系方法的一种改进。利用对象-关系方法，不仅应知道这个模型的组织结构的好处是什么，而且应试图将这些好处加以合适推广。

一些人已经开始以面向对象的方式来思考问题，每当建立一个超类/子类模型时，面向对象的思想就得到了使用。在传统的人这个超类和顾客与雇员子类中，我们认识到人是人和顾客的一个概化。然而，在关系模型中并没有明确的方法实施子类，要么把所有的属性放到一个表中，要么将它们分成两个或更多个表。在关系数据库中并没有明确的方法表达“概化”的意思。

关系和对象思想之间理论上一个主要的不同是：在关系数据库中，分开考虑数据和应用，而在对象思想中，合起来考虑数据和与它关联的操作和方法。

在 Oracle 领域中，经常分开考虑数据和应用。在 Oracle Designer 中，通过交互矩阵将功能连接到实体中。在 Oracle Developer 中，应用被连接到数据中，其方法与程序和数据结构相互作用的方法一样。逻辑上，数据和代码不应该分开考虑。数据对象和它们各自的操作与方法结合起来考虑的面向对象的方法是一种更自然、更符合逻辑的方法。

利用这种更自然的制作模型的方法，开发与维护速度可以期望得到改进。系统与所使用工具的分离越大，开发过程就会越长。如果能够使 RDBMS 及其工具与面向对象的方法更加一致，那就能够更快地构造更好的系统。

希望改善的第二个原因是数据模型更小了（表减少了）、更健壮了，而且更加易于开发。本章最后的例子将会解释这一点。

1.2 什么是 UML

随着日益复杂的系统的出现，一个用来描述它们的清晰、简洁的方法正变得越来越重要。

针对这种需要，Grady Booch、Jim Rumbaugh和Ivar Jacobson开发出了统一建模语言（UML）。在建立一个信息管理和事务处理的模型及文档系统时，可以利用面向对象的分析和设计方法建立UML。为了构造成功的系统，一个健全的模型是很重要的，它将总的系统计划和整个开发组联系在了一起。像其他任何语言一样，UML有它自己的组成和原则且将用于整本书，用来阐明构造不同模型模式和实际系统的例子。

设计UML的基本目标如下所示：

向用户提供了一种随时可用的可视化模型语言以便他们能够开发和交换有意义的模型。

为扩展核心概念提供了扩展和规范机制。

独立于特定的编程语言和开发过程。

为理解模型语言提供了正式基础。

鼓励面向对象工具市场的发展。

支持较高层次的开发概念，如合作、框架、模式和组件。

集成最好的实践经验。

基于以上所有理由，UML是设计对象-关系系统的可选择语言。

UML并不只是实体关系图的替代物。UML包含了许多组成部分，它们结合在一起提供了一个完整的面向对象的开发环境。UML中处理数据模型的部分是类图，本书将专门讨论UML中的类图，同时简单地提及UML中的任何其他部分。应该注意，UML包括整个系统设计环境，而不只是数据模型。关于UML的完整讨论可在任何一本能够获得UML的书中找到。

UML的组成部分包括：

类图（Class diagram）。这是数据模型图示语言，它与ER模型的作用范围差不多。

对象图（Object diagram）。这是一种只有一组对象的类图。可把它看作是一种显示例子数据而不是总的数据模型的一种数据模型，它对解释复杂的图是很有用的。

用例图（Use case diagram）。用例的思想类似于Oracle的CASE方法中的“函数”。用例图显示了各种角色（例如顾客、雇员）和使用情况的集成。在Oracle方法学中没有这种图的模拟。

时序图（Sequence diagram）。时序图显示了按时间顺序安排的对象交互过程。它类似于Oracle Designer中的过程流程图。

合作图（Collaboration diagram）。为了执行某些功能，对象之间需要传递信息，合作图显示了这些对象及其相关信息。Oracle方法学中也没有这种图的模拟。

状态图（Statechart diagram）。状态图是标准的状态转换图。它们表明一个对象可处于什么状态，什么原因可使对象转换状态。Oracle方法学中也没有这种图的模拟。

行为图（Activity diagram）。行为图是一种流程图，它表示操作及其判定点。它类似于Oracle Designer中的数据流程图。

执行图（Implementation diagram）。执行图表明了系统组件以及它们之间的交互作用。执行图能够表示系统中的软件或硬件组成部分。

使用UML

对于熟悉ER模型的人来说，重要的是要认识到向UML类图的转换并不令人惊奇。严格的模型制作者更加熟悉实体关系图（ERD）的受限之处。UML消除了部分而绝不是全部的限制，

在ERD中我们不喜欢的一些内容在UML中仍然存在。

搞清楚关系和对象模型的术语是很重要的。需要记住的是，ER模型中的“实体”概念应被UML中的“类”概念所代替。类的正确定义要比实体定义的范围广。也可以使用对象描述处理信息。但是，在下面的讨论中，对象一词被用来表示上下文中的数据对象。

一个类就是一些有意义的事物的集合，或是有意义的事物的一种分类。关键点是一个实体或类经常代表现实生活中的某些东西，这是一个可以检验数据模型有效性的关键机制。该数据模型能准确表达类中的每个对象所对应的现实生活中的“东西”吗？如果不能被准确表达，那就没有一个有效的数据模型。

对ER模型制作者来说，类和实体一般认为是等同的。但是，它们之间有一个重要的分别，即类和“方法”是有关系的，方法可认为是PL/SQL、Java或与表相互作用的函数。例如，对一个雇员来说，相关联的方法可能有雇佣、解雇、提升、分配至部门、分配至委员会等。穷尽地开发出一系列方法，以便使开发者只需要与这些方法相互作用而不必直接操作类中的数据是完全可能的。将开发者从类的物理结构中分离出来被认为是面向对象技术的基本优点。但是，这却需要仔细考虑怎样为自己的开发组建立结构。如果开发者构造方法和开发应用程序一样，那么封装将不会带来什么好处。

1.3 Oracle的面向对象产品

Oracle 8i并不是第一次使用具有面向对象思想的Oracle产品。在Form 4.5中通过使用属性类曾对面向对象做了第一次尝试。可对应用中的任何对象设置和执行属性的标准设置。利用库和对象组的PL/SQL是一组极为丰富的特性，可提供一些创建可重用程序组件的能力。一些人甚至将应用程序中的所有部件都封装到可重用结构中。现在的产品正开始向更好地支持面向对象思想的方向发展。

在RDBMS中——从8.0版起步，Oracle第一次提供了对象-关系数据库。8i版已经对这次编写工作作了测试，期望在1998年年底可得到使用。在对象扩展中，主码补充上了对象标识符（OID），传统的参照完整性利用对象指针进行扩展，且增加了两个非第一范式结构：嵌套表和参考表。Oracle Developer 2.0首次于1998年3月发表，提供了显式模板和对象库支持（多年来，许多人一直在为产生这个模板功能而努力）。这个继承模型被改进以便引用的对象能够被修改。最后，挂接块到PL/SQL过程的能力也增加了进去，这对于将块连接到Oracle 8i对象扩展中起到了先驱作用。

Oracle Designer 2.1版（1998年6月）通过新增加一个称为对象数据库设计器（Object Database Designer，ODD）的新组件提出了第一个对象扩展。这个组件提供了利用UML类图的子集产生Oracle 8i对象扩展DDL代码的能力。

随着RDBMS中对象的扩展，Oracle Developer开始支持编写针对对象扩展的应用程序，同时Oracle Designer开始支持DDL的产生。现在可以创建对象系统了吗？这倒不一定。到目前为止，关系和面向对象数据库的结合仍然很新。纵然是这次编写工作，我们仍然不能对Oracle 8i中的属性和方法完全继承，Oracle 8i仅提供了有限的可继承性。在Oracle Designer的ODD中，UML的许多重要特征还未被实施。

这并不意味着对象-关系数据库是一个坏想法，只是因为它们仍在发展中。在今后的几年内，对象-关系数据库将会作为新的系统标准完全代替传统的关系数据库。

很明显，对象-关系数据库是未来的发展趋势。但是，目前还不准备构造对象-关系系统的产品。首先，Oracle 8i中的对象扩展需要经历另一个反复的过程，这至少需要一个完全的可继承性以便能够说明已经有了面向对象系统。在 Oracle 8i中，可继承性目前有一个语言接口问题。这个问题是：各种语言对“继承性”含义的理解并不一致。C++支持多重继承性，Java则不支持。Oracle语言接口（C++，Java）中的每一种都完全支持语言的继承性模型。（在 Oracle 8i中，Java的继承性在服务器端是可以利用的。）

在Oracle Designer方面，需要UML类图在ODD中的完全执行，也需要UML中的其他一些部分在产品中得到应用，此外还需要能产生与对象结构共同运行的模块。Oracle Designer支持静态数据库UML建模。据估计，UML用户中的85%将他们自己局限在静态建模中。Oracle Designer支持适合产生Oracle 8i结构的UML子集，这应该可以使大多数用户得到满足。但是，本书提倡使用全部UML类图语法设计数据模型。当设计者增加更多的UML支持时，可利用第三方工具和Oracle 8i中的对象-关系特征得到完整的UML模型结果。

1.4 Oracle 8i 中的新特征

Oracle 8i包括一些新特征，其中一些专门属于对象-关系数据结构的可用性。本节将提供许多新的Oracle 8i对象特征的概况，这些特征的大部分扩展到本书更为详细的例子中。

1.4.1 对象-关系数据结构

我们将先开始讨论与对象层相关的 Oracle 8i的新特征，因为这是改进最重要的一个领域。虽然Oracle 8i中的改进并不全与对象相关，但它们中的大多数都与对象-关系结构有关。

1. 列与行类型

类型是一种物理结构，用它来作为其他类型和/或表的模板或构造块。列类型是用来定义有一个或多个属性或成员的组织结构的类型，这些属性或成员共同描述了有一个或多个列对象的结构。代码1-1定义了一个列类型NAME_T，它描述了存储姓名信息的列结构，例如人的姓和名。

代码 1-1

```
CREATE OR REPLACE TYPE NAME_T AS OBJECT
(COL          VARCHAR2 (50))
/
```

行类型是用来定义一个对象表的整个结构的类型。代码 1-2定义了一个类型 PERSON_T。PERSON_T类型包括4个属性：PERSON_ID（即产生的唯一标识序列 UID）、LNAME_TX（即姓）、FNAME_TX（即名）和BIRTH_DATE（即出生日期）。

代码 1-2a

```
--example_01_01 is prereq
CREATE OR REPLACE TYPE PERSON_T
AS OBJECT(
PERSON_ID      NUMBER (10),
LNAME_TX      NAME_T,
FNAME_TX      NAME_T,
BIRTH_DATE    DATE)
/
```

注意，既然 LNAME_TX 和 FNAME_TX 列的数据类型说明不能显式地定义为 VARCHAR (50)，DML 语句的典型关系语法就不会起作用。相反，数据类型引用了在代码 1-1 中定义的新列类型 NAME_T。同样，任何 DML 语句都必须指定列类型。

列类型类似于 Oracle Designer 中的域，主要是由于它们将列结构归类。如果还需要修改系统中每个姓名字段的数据类型或长度，则只需修改 NAME_T 的列类型，因为它的结构被每一个姓名字段所共享。

同样，行类型产生了对象表结构。代码 1-2a 中 PERSON_T 行类型描述了存储人的信息的表结构，代码 1-2b 定义了一个基于 PERSON_T 行类型的表 EMP。

代码 1-2b

```
CREATE TABLE EMP OF PERSON_T (  
  PERSON_ID      NOT NULL,  
  LNAME_TX       NOT NULL,  
  FNAME_TX       NOT NULL,  
  BIRTH_DATE     NOT NULL,  
  PRIMARY KEY (PERSON_ID))  
/
```

基于行类型的表的唯一缺点是，这些表严格地受限于它们所相关的行类型的结构。换句话说，对象表不能偏离它们相关的类型结构，结构的任何修改必须依其类型执行，且会被表继承。

注意，EMP 表中的列已在 CREATE TABLE 语句中被重新声明。当列被来自 PERSON_T 行类型成员中的 EMP 表继承时，定义在表这一级的成员的实际实施就会具有其他特征。

代码 1-2b 已声明所有四个成员（即 PERSON_ID、LNAME_TX、FNAME_TX 和 BIRTH_DATE）不能为空，在 CREATE TABLE 语句中也必须指明完整性约束和缺省值。

缺省值和可选项之类的条款在类型级上将是有用的参数，可像即将发行的 Oracle 8i 版本中那样来使用。列类型不限于单独的属性使用，例如，也许想创建下面代码 1-3 中的 ADDRESS 列类型。

代码 1-3

```
CREATE OR REPLACE TYPE ADDRESS_T AS OBJECT  
(ADDR1          VARCHAR2 (50),  
 ADDR2          VARCHAR2 (50),  
 CITY_TX        VARCHAR2 (50),  
 ST_CD          VARCHAR2 (2),  
 CTRY_CD        VARCHAR2 (3),  
 ZIP_CD         VARCHAR2 (9))  
/
```

在包含地址信息的任何对象类型的创建过程中，你可能最想引用 ADDRESS_T 对象类型。例如，CONTACT_T 表可能包括用于和某一给定联系人通讯的主要地址。可以定义一个 CONTACT_T 对象类型，并将它建立在 PERSON_T 和 ADDRESS_T 数据类型的基础上，如代码 1-4 所示。

对象表结构是在对象类型的基础上建立起来的。如果基于对象类型创建一个称为 EMP 的表存储雇员信息，这两种结构看起来可能会像下面的代码 1-5。

代码 1-4

```
--EXAMPLE_01_03 IS PREREQ
--members of multi-member types not supported as key components
CREATE OR REPLACE TYPE CONTACT_T AS OBJECT
(PERSON          PERSON_T,
ADDRESS          ADDRESS_T)
/

CREATE TABLE CONTACT OF CONTACT_T (
PERSON.PERSON_ID NOT NULL)
/
```

代码 1-5

```
--example_01_01 is prereq
CREATE OR REPLACE TYPE PERSON_T
AS OBJECT(
LNAME_TX          NAME_T,
FNAME_TX          NAME_T,
BIRTH_DATE        DATE)
/

CREATE OR REPLACE TYPE EMP_T AS OBJECT
(EMP_ID           NUMBER (10),
PERSON            PERSON_T,
HIRE_DATE         DATE)
/

CREATE TABLE EMP OF EMP_T
(EMP_ID           NOT NULL PRIMARY KEY)
/
```

2. 关系和对象-关系表

Oracle 8i中对象-关系数据结构的产生使许多人想知道利用Oracle以前的版本创建的已存在系统的可靠性。Rest保证,对于Oracle或任何其他的人来说,关系数据结构都不会成为一件过时的事物。

Oracle 8i提供了创建关系表和对象-关系表的功能。Oracle Server的未来版本将支持ALTER TABLE命令,该命令将关系表转换为对象表。所以,逻辑上的问题是,还有必要创建关系表吗?结果发现,在某些情况下,关系表的某些特征仍使它使用起来比对象-关系表更方便。

关系表的灵活之处在于用户可任意增加列。至于 Oracle 8i,当然也可以从已存在的表中删除列,但这是一个毫无价值的特点,因为这将导致有无数的地方等着我们为表重新命名,然后使用一个CREATE TABLE...AS SELECT FROM...语句,之后再为原来的表写上 DROP TABLE语句。这时,你会发现完整性约束和索引都需要用适当的名字和位置重新创建。现在所有这些冗余的工作已被删除。

在Oracle 8i中,对象表被严格限于它们所基于的对象类型。你不能直接在对象表中增加列,而必须将它们加在对象表所基于的对象类型中。

关系表可以包括在用户定义的列类型的基础上建立起来的列。例如,代码 1-1 中的NAME_T列类型。因此,关系表和对象表之间的主要不同之处在于它们是否基于一个单一类型。SQL语句能够利用单一查询来引用关系表和对象-关系表。

例如，要创建一个称为 PERSON 的对象表存储人的信息，PERSON 表需要一个唯一的识别符和如下字段：姓、名、出生日期。合适的对象-关系策略应该是代码 1-6 中所显示的对象类型和对象表。

代码 1-6

```
CREATE OR REPLACE TYPE PERSON_T AS OBJECT
PERSON_ID          NUMBER,
LAST_NAME          VARCHAR2 (50),
FIRST_NAME         VARCHAR2 (50),
BIRTH_DATE         DATE)
/

CREATE TABLE PERSON OF PERSON_T (
PERSON_ID NOT NULL PRIMARY KEY,
LAST_NAME NOT NULL,
FIRST_NAME NOT NULL,
BIRTH_DATE NOT NULL)
/
```

如果要在 PERSON 表中另加一列用于存储性别指示符（即 GENDER_MF），则不得不把它加在 PERSON_T 对象类型中。但是，Oracle 8i 版本将会提供一种功能，它可以修改已被其他对象引用的对象类型。

下面的代码 1-7 中，已将 GENDER_MF 列加到了 PERSON 表中。

代码 1-7

```
--EXAMPLE_01_01 IS PREREQ
DROP TABLE PERSON
/

CREATE OR REPLACE TYPE PERSON_T AS OBJECT
(PERSON_ID          NUMBER,
LAST_NAME          NAME_T,
FIRST_NAME         NAME_T,
BIRTH_DATE         DATE,
GENDER_MF          CHAR (1))
/

CREATE TABLE PERSON OF PERSON_T (
PERSON_ID NOT NULL PRIMARY KEY,
LAST_NAME NOT NULL,
FIRST_NAME NOT NULL,
BIRTH_DATE NOT NULL,
GENDER_MF DEFAULT 'F' NOT NULL)
/
```

Oracle 8i 中，将不得不为原来存储在 PERSON 表中的数据升级，并将它重新装载进新的 PERSON 表中。一个通常的方法是把原来的 PERSON 表重命名为别的名字。在产生了一个新的 PERSON 表后，使用 INSERT INTO...SELECT FROM... 语句将数据从原来的 PERSON 表中转移到新的 PERSON 表中。相关的 SQL 语句显示在下面的代码 1-8 中。

现在让我们考虑下面这个问题：怎样创建多个存储关于人的信息的表。 INSERT INTO...

SELECT...FROM语句可以很好地解决这个问题。然而，当使用 CREATE TABLE... SELECT... FROM语法时会出现本质上的缺陷，CREATE TABLE...SELECT...FROM语句将不会自动创建完整性约束，但这些完整性约束恰恰能反映那些表是从哪儿选择出来的。因此，作为 CREATE TABLE语句的一部分，必须保证已经手工声明了目标表上想要得到的每个参照完整性约束。

代码 1-8

```
RENAME PERSON TO ORIG_PERSON;  
CREATE TABLE PERSON OF PERSON_T;  
INSERT INTO PERSON (PERSON_ID, LAST_NAME, FIRST_NAME, BIRTH_DATE)  
SELECT PERSON_ID, LAST_NAME, FIRST_NAME, BIRTH_DATE FROM ORIG_PERSON;
```

对象引用也将在这个过程中丢失，因为它们是由系统产生的、每一行都经过唯一检验的值。当 PERSON 和 ORIG_PERSON 表中行的内容相同时，只要它们是由系统产生的，那么它们的唯一性标识符就不会相同。这是因为 ORIG_PERSON 表中的每一行都已被拷贝，而且已经插入到了 PERSON 表中，结果它们现在都是最初的个别行的拷贝。在 ORIG_PERSON 表的 OID 基础上建立起来的对象引用将会在不稳定的状态下结束，换句话说，引用 ORIG_PERSON 表的表将会包含对新 PERSON 表的无效引用。

Oracle 8i 已引入了一种新的完整性约束，可通过对象引用实施完整性。另一种途径是基于用户定义的列的声明对象引用，如代码 1-8 中的 PERSON_ID。其中，用户定义的列将促使对象引用复制传统的主码/外码检验方面的功能。

在 PERSON_T 对象类型的基础上，可以创建对象表的任何成员。但是，这违反了将在第 5 章中所描述的、对主码列的命名规则（例如，系统产生的 PK 列的名字是 TABLE_NAME||'_ID'），因为这将会导致在多个表中都有一列名为 PERSON_PK 的 PK 列。

相反，我们能够创建关系表。关系表引用 PERSON_T 作为列的数据类型，而不是行的对象类型。我们将首先从 PERSON_T 对象类型的定义中移去 PERSON_ID 成员，因为包含对 PERSON_T 类型的引用并不是所有表所特有的。下面的代码 1-9 使用 PERSON_T 作为数据类型。

代码 1-9

```
CREATE OR REPLACE TYPE PERSON_T AS OBJECT  
(LAST_NAME          VARCHAR2 (50),  
 FIRST_NAME         VARCHAR2 (50),  
 BIRTH_DATE         DATE,  
 GENDER_MF          CHAR (1))  
/  
  
CREATE TABLE EMP (  
EMP_ID              NUMBER (10) NOT NULL PRIMARY KEY,  
EMP                  PERSON_T)  
/  
  
CREATE TABLE CONTACT (  
CONTACT_ID          NUMBER (10) NOT NULL PRIMARY KEY,  
CONTACT              PERSON_T)  
/  
/
```

前面两个例子演示了如何使一种类型既作为列对象类型又作为行类型。注意，在使用多属性类型作为列类型时，语法检查将提示你正在增加一个单独的列（例如，EMP 表中的 EMP

列，因为它引用了 PERSON_T 列类型)。如果你用 SQL*Plus 描述 EMP 表，它看起来就会像下面一样：

```
SQL> desc EMP
Name                               Null?      Type
-----
EMP_ID                             NOT NULL   NUMBER(10)
EMP                                 PERSON_T
```

为了在 EMP 表中为 PERSON_T 数据类型的成员插入一个对象，必须在 INSERT 语句中使用类型构造器，如下面代码 1-10 所示。

代码 1-10

```
SQL> INSERT INTO EMP VALUES
(1, PERSON_T('SMITH', 'JOE', '06-JUN-98', 'M'));
```

在早期的 INSERT 语句中如果引用 PERSON_T 构造器失败的话，就会产生一个错误。

数据库把列对象类型看作是嵌入的对象，因此需要它们在所有的 DML 语句中被显式地引用。SELECT 语句还能利用通配符（如 SELECT*）为所有行返回所有列。

注意 对象表不支持 Oracle 8i 中的复制。

3. 对象标识符（Object Identifiers，OID）

Oracle 8i 中包含 OID，它在关系数据库和 C++/Java 开发环境之间起着桥梁的作用。对于开发本地 Oracle 数据库，SQL、PL/SQL、Forms 和 Reports，ROWID 是数据库中定位特定行的最佳方法。

但是，ROWID 在数据库自身之外是没有意义的。OID 在数据库自身的范围之外本质上是用于编程的逻辑指针。一旦一个给定的 C++/Java 应用程序需要使用一些数据，并将这些数据从数据库中提取装入内存，这些语言就会需要一些在对象行之间导航的方法。OID 提供了这个级别上的唯一验证。OID 听起来有些类似于主码和（或）唯一码，它们可由系统产生或基于一个或多个用户定义的列产生。系统产生的 OID 是经过编码的值，它们的内容对于它们自身来说是绝对没有用处的。但需要记住的是，当它们由系统产生时，它们的值必须保证是唯一的，数据库不能从用户定义的列中保证值的唯一性。Oracle 8i 包含一种新的、可避免 OID 冗余现象的完整性约束。

OID 可以是主码组件。实际上，正在经历的演变仅仅是参考完整性约束的一种扩展。这种约束能够有效地用在程序里作为指针使用，而这种程序又是用 Oracle Server 之外的语言编写的。

能够从类型中创建的数据对象中的两种类型是列对象和行对象。列对象在前面代码 1-2b 中做过介绍，那里 EMP 表中创建了一个称为 EMP 的列，它是对 PERSON_T 类型的引用。列对象不是独立的，它们被嵌在其他表中，因而不能接受到一个 OID。行对象是基于对象类型、为对象表建立的。重要的是要记住，为了在外部应用程序中通过 OID 访问数据，必须将数据存储在对象表中或在关系表上创建对象视图，因为 OID 只能通过对象表或视图创建。数据库世界中 OID 的主要好处在于它们大大简化了 WHERE 语句的规范。如果你正在利用 OID 将两个表联系起来，SQL 会向你解释这种关系并自动指明合适的 JOIN 子句。

4. 对象引用

对象引用仅仅是一种能够同步支持参考完整性和外部编程语言的指针。REF 码用于指明

一个对象引用。如果一个对象引用已利用 SCOPE 参数进行了定义，SQL 就会将这个语句作为 JOIN 来执行。产生这种结果的原因是，通过对象引用值的位置，已经为引擎指明了所指向对象的准确对象表。如果未使用 SCOPE 参数，并在给定的对象类型上产生了多个表，优化器就会追踪对象引用值，这与执行 JOIN 命令恰好相反。

下面代码 1-11 利用对 PERSON_T 类型的对象引用创建了一个表，特别地指向了由 PERSON_T 类型创建的 EMP 表中的行。

代码 1-11

```
CREATE OR REPLACE TYPE PERSON_T AS OBJECT
(PERSON_ID          NUMBER,
 LAST_NAME          VARCHAR2 (50),
 FIRST_NAME         VARCHAR2 (50),
 BIRTH_DATE         DATE,
 GENDER_MF          CHAR (1))
/

CREATE TABLE EMP OF PERSON_T
/

CREATE TABLE TASK (
TASK_ID             NUMBER (10),
PERSON              REF PERSON_T  SCOPE IS EMP)
/
```

代码 1-11 定义了类型 PERSON_T，用来作为 EMP 表的模板。TASK 表包含一个指向 PERSON_T 类型的 REF。

如果没有限制 EMP 表的 SCOPE，优化器会自动导航对象引用值来查找所需的数据。为了进行快速数据检索，Oracle 管理内存中的 OID 综合列表。导航访问将通过这组 OID 进行数据库查找，并确定行的位置，以满足所需的查询。但是，既然我们限制了范围，我们已经开始指示引擎执行 JOIN，该命令可在含有索引的情况下被优化。当用户提供了一个字符串如 EMP_ID='23495' 之后，导航访问会被执行；否则，则只能期望引擎执行一个 JOIN。

在可预见的将来，对象引用的执行性能可能要超过 JOIN 的执行。但是，这种优势的实现还需要一段时间。到现在为止，在 Oracle Server 中，JOIN 仍是访问数据的最优途径。

5. DEFERRED/NOVALIDATE (推迟/无效)约束可选项

DEFERRED/NOVALIDATE 可选项是一对相似的特征，两者结合便可操纵由完整性约束执行的有效性。DEFERRED 可选项是 SET CONSTRAINTS 命令的一个参数，由 SQL*Plus 或相对活动的会话产生。这个参数接受的两个值是 IMMEDIATE 和 DEFERRED。

缺省情况下，这个命令被设置为 IMMEDIATE，这意味着 RDBMS 将会使约束立即生效。DEFERRED 提供了这样一种能力：直到 COMMIT 发布，它才使约束生效。DEFERRED 可选项在 PO 和 PO_DTL 的情况下通常是有效的，在那些情况下，你可以在多个有关联的表上同时执行 DML 操作，而对这些事务进行检验的顺序却可不必保证。

NOVALIDATE 约束子句允许只在新记录或修改记录的情况下才能执行约束检查，它特别针对增加新约束或发生大的数据迁移的数据库。

数据迁移的一个最昂贵的组件是未遵循新环境的完整性约束的遗留数据的清除 / 转换组件。
例如：假设你正在迁移一系列联系人的信息，他们居住的州包含在数据集中。

CONTACT_ID	LAST_NAME	FIRST_NAME	STATE_CD
1	SMITH	JOHN	NJ
2	JONES	PAUL	X
3	WHITNEY	JED	
4	KRAUS	LIZZIE	PA

假设新的CONTACT表要求所有这四个字段都是强制型的，STATE_CD字段在REF_STATE表进行检验，那么联系人2和3将会既违反STATE_CD列上的NOT NULL（非空值）约束，也违反STATE_CD列上的外码约束。相应代码显示在代码1-12中。

代码 1-12

```

DROP TABLE CONTACT;
DROP TABLE REF_STATE;

CREATE TABLE REF_STATE (
STATE_CD      VARCHAR2(2) NOT NULL PRIMARY KEY,
DESCR_TX      VARCHAR2 (50) NOT NULL)
/

CREATE TABLE CONTACT (
CONTACT_ID    NUMBER (10) NOT NULL PRIMARY KEY,
LAST_NAME     VARCHAR2 (40) NOT NULL,
FIRST_NAME    VARCHAR2 (40) NOT NULL,
STATE_CD      VARCHAR2 (2) CONSTRAINT STATE_CD_NN NOT NULL DISABLE,
CONSTRAINT CONTACT__STATE_FK FOREIGN KEY (STATE_CD)
REFERENCES REF_STATE(STATE_CD) DISABLE)
/

INSERT INTO REF_STATE (STATE_CD, DESCR_TX) VALUES ('NJ', 'NEW JERSEY');
INSERT INTO REF_STATE (STATE_CD, DESCR_TX) VALUES
('PA', 'PENNSYLVANIA');

INSERT INTO CONTACT (CONTACT_ID, LAST_NAME, FIRST_NAME, STATE_CD)
VALUES (1, 'SMITH', 'JOHN', 'NJ');
INSERT INTO CONTACT (CONTACT_ID, LAST_NAME, FIRST_NAME, STATE_CD)
VALUES (2, 'JONES', 'PAUL', 'X');
INSERT INTO CONTACT (CONTACT_ID, LAST_NAME, FIRST_NAME, STATE_CD)
VALUES (3, 'WHITNEY', 'JED', NULL);
INSERT INTO CONTACT (CONTACT_ID, LAST_NAME, FIRST_NAME, STATE_CD)
VALUES (4, 'KRAUS', 'LIZZIE', 'PA');

ALTER TABLE CONTACT ENABLE NOVALIDATE CONSTRAINT STATE_CD_NN;
ALTER TABLE CONTACT ENABLE NOVALIDATE CONSTRAINT CONTACT__STATE_FK;

SQL> SELECT * FROM CONTACT;

CONTACT_ID LAST_NAME FIRST_NAME ST
-----
1          SMITH      JOHN      NJ

```

2	JONES	PAUL	X
3	WHITNEY	JED	
4	KRAUS	LIZZIE	PA

如上所示，约束现在能像原来移入一样使无效数据保持在表中，简言之，你不能再对你不打算使用的数据执行数据清除。

但是，如果你插入了一个新记录，或对一个违反了给定约束且已存在的记录的一列进行了更新，你输入的值就会被强制进行相应的有效性检查。注意，对于表中那些不违反约束的列可以进行更新。

6. 对象视图

对象视图很类似于关系视图。对象视图是基于单一的 SELECT 语句之上的，这个 SELECT 语句能够引用一个或多个关系表和 / 或对象 - 关系表，视图最常见的使用方式主要还是对实际的数据结构的内容提供安全访问。使用对象视图的一个主要好处是它们在关系设计和对象 - 关系设计之间可作为一条迁移路径。可以创建基于对象类型的对象视图，并访问存储在关系表中的数据，这就使对象引用及其方法的使用能与关系数据建立关联，而不必改变任何数据结构。

最根本的是，如果在自己的环境中有一个很好的机会可以比较两种技术的异同点的话，就能够维持已存在的关系的应用程序，并方便地在它们上面增加对象层。对象视图可有效地使用 Oracle 8i 对象而不必迁移已存在的关系数据。

对象视图可以像对象表一样是强类型的。换句话说，对象视图和对象表必须被说明为单一的对象类型，它们的结构不能脱离这种数据类型。尽管文中暗示关系表应该已存在，但我们还是把用于关系表 DEPARTMENTS 和 EMPLOYEES 的 CREATE TABLE 语句加到代码 1-13 中进行演示。所需要的任何结构上的修改必须基于对象类型执行，且可被对象表继承。因此，产生对象视图的第一步是定义对象类型。假设有一个名为 EMPLOYEES 的关系表，它包含雇员信息，现在要在它上面增加一个对象层。EMPLOYEES 表包含 DEPARTMENTS 表的外码，如以下代码 1-13 所示。

代码 1-13

```
CREATE TABLE DEPARTMENTS (
DEPT_ID      NUMBER (10) NOT NULL PRIMARY KEY,
NAME         VARCHAR2 (50) NOT NULL)
/

CREATE OR REPLACE TYPE DEPT_T AS OBJECT (
DEPT_ID      NUMBER (10),
NAME         VARCHAR2 (50))
/

CREATE VIEW DEPT_OV OF DEPT_T WITH OBJECT OID (DEPT_ID)
AS SELECT DEPT_ID, NAME
FROM DEPARTMENTS
/

CREATE TABLE EMPLOYEES (
EMP_ID       NUMBER (10) not null primary key,
LAST_NAME    VARCHAR2 (50) not null,
```

```

FIRST_NAME  VARCHAR2 (50) not null,
HIRE_DATE   DATE not null,
DEPT_ID     NUMBER (10) REFERENCES DEPARTMENTS (DEPT_ID))
/

CREATE OR REPLACE TYPE EMP_T AS OBJECT (
EMP_ID      NUMBER (10),
LAST_NAME   VARCHAR2 (50),
FIRST_NAME  VARCHAR2 (50),
HIRE_DATE   DATE,
DEPT_ID     REF DEPT_OV)
/

CREATE VIEW EMP_OV OF EMP_T WITH OBJECT OID (EMP_ID)
AS
SELECT EMP_ID, LAST_NAME, FIRST_NAME, HIRE_DATE,
MAKE_REF (DEPT_OV, DEPT_ID)
FROM EMPLOYEES
/

```

注意，OID已基于关系表EMPLOYEES中的EMP_ID列值被定义。OID值可由系统产生或由用户定义。MAKE_REF命令类似于一个对象引用，允许对指向一个关系表的主码的指针进行引用。

CAST-MULTISET是一个SQL表达式，可用来从子查询的结果集中创建一个强类型的集合。利用EMP/DEPT的例子，我们可以使用CAST-MULTISET表达式创建EMP_BY_DEPT_OV对象视图，如代码1-14所示。

代码 1-14

```

--example_01_12 is prereq
CREATE TYPE EMPS_T AS object (
EMP_ID      NUMBER(10),
LAST_NAME   VARCHAR2(50),
FIRST_NAME  VARCHAR2(50),
HIRE_DATE   DATE)
/

CREATE TYPE EMPSET_T AS TABLE OF EMPS_T
/

CREATE OR REPLACE TYPE EMP_BY_DEPT_T AS OBJECT (
DEPT_ID     NUMBER(10),
NAME        VARCHAR2(50),
EMP         EMPSET_T)
/

CREATE VIEW EMP_BY_DEPT_OV OF EMP_BY_DEPT_T
WITH OBJECT OID (DEPT_ID)
AS SELECT DEPT_ID, NAME,
CAST (MULTISET(
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, E.HIRE_DATE

```

```
FROM EMPLOYEES E
WHERE E.DEPT_ID = D.DEPT_ID)
AS EMPSET_T)
FROM DEPARTMENTS D
/
```

CAST-MULTISET命令主要用来聚合主记录和相关的细节记录。代码1-14返回了EMPLOYEES表中所有与DEPARTMENTS表中给定的科室记录相关的雇员记录。CAST-MULTISET本质上是嵌入子查询的一个新语法，它能够使用户将相关的行组合成一组以便检索。

在EMP_BY_DEPT_OV对象视图中查询数据将返回与科室相关的雇员记录。CAST-MULTISET子句中的查询称为集合查询（collection query）。注意子查询必须赋予一个对象类型。在这种情况下，我们重新使用代码1-13中创建的EMP_T对象类型。

作为每种对象视图定义的一部分，现在可以加入普通存取数据的方法。第13章中会讲到，Oracle 8i的未来版本将会提供封装方法和列的能力。至于这次编写，我们可以通过创建另外的对象视图来模仿这种功能。事实上，若不去考虑对象类型，对象视图在某种程度上甚至能够支持继承性。

1.4.2 INSTEAD OF 触发器

视图通常由赋予别名的复杂表达式组成。过去，Oracle还不能为复杂视图提供DML支持。这样产生了INSTEAD OF触发器以通过视图支持DML操作。这些新的触发器代替实际的DML语句来触发，能够为视图创建，但不能为表创建。在INSTEAD OF触发器内编写的代码就是要被执行的DML。应用程序现在能够基于视图并对视图发布DML操作，然后，INSTEAD OF触发器中断这些DML操作而去执行嵌入在INSTEAD OF触发器中的代码。INSTEAD OF触发器可对NEW和OLD全局变量进行引用，就像其他行级触发器所做的那样，如以下代码1-15所示。

代码 1-15

```
--example_01_13 is prereq
CREATE OR REPLACE TRIGGER IOI_EMP
INSTEAD OF INSERT ON EMP_OV FOR EACH ROW
DECLARE
INVALID_DEPT_ID    EXCEPTION;
BEGIN
IF :NEW.HIRE_DATE > SYSDATE - 30 THEN
INSERT INTO EMPLOYEES VALUES(:NEW.EMP_ID,
                              :NEW.LAST_NAME,
                              :NEW.FIRST_NAME,
                              :NEW.HIRE_DATE,
                              NEW.DEPT_ID);

END IF;
EXCEPTION
WHEN INVALID_HIRE_DATE THEN
RAISE_APPLICATION_ERROR(-20001, 'HIRE DATE '||
'MUST BE WITHIN LAST THIRTY DAYS OR IN FUTURE. ');
END;
/
```

但是，要记住的是，视图主要是为了检索数据，而不是为了操作。因此，当 DML 借助于 INSTEAD OF 触发器被执行时，视图并不自动支持参照完整性。如果用 INSTEAD OF 触发器创建了一个对象视图，该触发器基于一个含有所定义的参照完整性的关系表，那就不必再手工执行外码有效性检查；否则就得动手编写。

1.4.3 大型对象

大型对象 (Large Object, LOB) 是 Oracle 8.0 版本中可利用的新数据类型。LOB 从根本上提供了比它们的前身 LONG 更大的存储量，LOB 的存储量可达到 4Gb。它们在一行内的存储量可达到 4K，否则就会产生行溢出。有许多不同的数据类型和 LOB 相适应，例如 CLOB (字符 LOB)、NCLOB、BLOB (二进制 LOB) 和 BFILE。BFILE 存储一个指向文件所在磁盘物理位置的指针。当然，BFILE 对这个文件并没有提供安全性，因为该文件是存储于数据库之外的。如果文件的安全性希望由数据库处理，那么这些文件应该利用 LOB 数据类型进行存储。

数据也能够存储在一个 LOB 中，虽然作者并不推荐这种选择。例如，也许想在 LOB 中存储安全数据 (如：工资)，因为 LOB 数据不能直接从 SQL 中得到。将数据存储在未经解释的二进制格式中是很有必要的。要想查看存储在 LOB 中的数据，建议的途径是执行 PL/SQL 方法，PL/SQL 可用来查看或操纵存储在 LOB 中的数据。

还能够进行 LOB 的上下文相关查询，例如请求列的前 500 个字符，而 LONG 则要求返回所有数值或一系列字母数字字符。可以在一个单独的表里拥有多个 LOB，而不像 LONG，后者在一个表中只能有一个 LOB。

1.4.4 服务器端代码

Oracle 8i 提供了在数据库内开发和存储服务器端 Java 代码的能力。这些 Java 程序单元可由 PL/SQL 触发器、过程和/或函数执行。与多年在服务器上进行优化的 PL/SQL 相比，未经编译的 Java 代码虽然性能上还要略差一些，但还是表现出了不错的性能。

但是，编译过的 Java 代码的执行速度比 PL/SQL 要快 10 倍。将这个统计结果和 Java 的可移植性特点相结合，就会看到一个新趋势。然而从关系数据结构方面考虑，PL/SQL 还很有用处。已经有太多的系统是基于 PL/SQL 的，而且还有更多这样的系统正在开发中。PL/SQL 是一个健壮的、老资格的编程语言，它已经为大众服务了许多年。

不过要记住的是，有许多世界组织正为使 Java 变得尽可能有效而努力。这些组织的共同努力将最有可能提供一个拥有优越性能和可用性的最终产品，单个组织则很难与之相匹敌。如果日后我们创造出与简单地引用 Java 相比，用 Java 编写的数据库触发器、过程和函数，你千万不要感到惊奇。

1.4.5 集合

集合是一种把与数据库中的对应行相关联的一系列的数据项进行存储的方法。集合不需要必要的完整性约束和单独的数据结构就可以模拟在一行和一个集合之间的一对多关系。

Oracle 8i 提供了两种类型的集合：VARRAYS 和嵌套表。

1. VARRAYS

VARRAYS，或称可变数组，通常同它的拥有行或主行进行串联式存储。VARRAYS 指定

所存值的最大数量。下面代码 1-16中显示了一个 VARRAY 的实现。

代码 1-16

```
CREATE OR REPLACE TYPE TEMP_T AS VARRAY (3) OF NUMBER (5,2)
/
```

TYPE_T 类型定义了一个 VARRAY，维护三个温度记录。DAILY_TEST 表在 TEMP_T 类型的基础上进行扩展，记录每天的三个温度采样，如下面代码 1-17 中所示。

代码 1-17

```
--example_1_15 is prereq
CREATE TABLE DAILY_TEST (
  DAILY_TEST_ID      INTEGER,
  DAY_ID              INTEGER,
  TEMPERATURE_SAMPLE TEMP_T)
/
INSERT INTO DAILY_TEST VALUES(1,1,TEMP_T(1))
/
```

可以用一个 INSERT 语句在 VARRAY 中插入多个值，其语法如代码 1-18 所示。

代码 1-18

```
INSERT INTO DAILY_TEST VALUES(1,1,TEMP_T(75,85,80))
/
```

如果以后要在 VARRAY 中增加值的话，必须求助于 PL/SQL 语法，如下面代码 1-19 中所示。未来的 Oracle 8i 版本可能会提供在 VARRAY 中增加值的 SQL 语法。

代码 1-19

```
CREATE TYPE var_t AS VARRAY(3) OF INTEGER;
CREATE TABLE tab1 (c1 INTEGER, c2 var_t);
INSERT INTO tab1 VALUES(0, var_t(1, 2));
DECLARE
  varvar var_t;
BEGIN
  SELECT c2 INTO varvar FROM tab1 WHERE c1 = 0;
  varvar.EXTEND();
  varvar(varvar.COUNT) := 3;
  UPDATE tab1 SET c2 = varvar WHERE c1 = 0;
END;
```

VARRAY 通常同它的拥有行或主行进行串联式存储，这样做产生了直接的性能收益，因为 VARRAY 值经常同主记录同时存取。当计划跟踪的元素个数较少或希望通过一次 SELECT 调用返回整个集合时，建议使用 VARRAY。如果一个基于 VARRAY 的列存放数据的最大值超过 4K 时，则它的值就作为“串联的”LOB 存储，VARRAY 值中超过 4K 的部分非串联地存放于 LOB 块中，而不超过 4K 的部分与主行进行串联式存储。

2. 嵌套表

嵌套表是第二种集合类型，类似于 Oracle 7 中的群集表概念，只是它们以不同的方式实现。嵌套表本质上是一种附加的关系表。同群集表一样，嵌套表并不存储于主表中。因此，嵌套

表本身并不提供性能收益。下面的例子展示了购货单表（即 PO表）中嵌入的购货单明细（即 PO_DTL）的集合，如代码 1-20所示。

代码 1-20

```
CREATE OR REPLACE TYPE PO_DTL_TYPE
AS OBJECT(
  ITEM_ID  VARCHAR2 (5),
  QTY      NUMBER (5))
/
CREATE OR REPLACE TYPE PO_DTL_TABLE_TYPE
AS TABLE OF PO_DTL_TYPE
/
CREATE OR REPLACE TYPE PO_TYPE
AS OBJECT(
  PO_ID     VARCHAR2 (5),
  DESCR_TX  VARCHAR2 (40),
  VENDOR_ID VARCHAR2 (5),
  PO_DTL PO_DTL_TABLE_TYPE)
/
CREATE TABLE PO
OF PO_TYPE(
  PO_ID     NOT NULL PRIMARY KEY,
  DESCR_TX  NOT NULL,
  VENDOR_ID NOT NULL
  REFERENCES VENDOR (VENDOR_ID))
NESTED TABLE PO_DTL STORE AS PO_DTL
(PRIMARY KEY (NESTED_TABLE_ID, PO_DTL_ID)
 ORGANIZATION INDEX)
RETURN AS LOCATOR
/
```

嵌套表不像 VARRAY 那样受到限制，总的来说查询嵌套表要比查询 VARRAY 效率高，因为在嵌入表中可以建立二级索引。可以独立于主表发送查询以聚集嵌套表中的行。假设希望发送一个查询用来按物品种类返回该物品所卖出的总数量，代码 1-21 所示的 SQL 查询就已足够达到目的了。

代码 1-21

```
SELECT D.ITEM_ID, SUM(D.QTY) QTY
FROM PO P, TABLE(P.PO_DTL) D
GROUP BY D.ITEM_ID
/
```

如你所见，主表（本例中是 PO 表）必须在 FROM 子句中指定，不能简单地基于某个嵌套表发送一个查询而不指定主表。原因是，尽管嵌套表实际上作为非独立的关系表存储，但它不允许对数据进行相同的存取，TABLE (...) 命令指示 SQL 分析程序作用于表 P（P 是 PO 表的别名）的嵌套表 PO_DLT，犹如表 P 是一个典型的表。相同的这个例子可以在 PL/SQL 中执行。

1.4.6 索引组织表

嵌套表可定义为索引组织表（IOT），这种表可以存储在索引结构中或作为索引结构存储，

整个表结构和内容存入内存中，这样就消除了为满足给定查询而识别合适 ROWID值的需要，取而代之的是查询直接在嵌套表的内容上进行。Oracle 8i中对象表不能作为索引组织表存储，虽然以后发行的版本将实现这一功能。当计划通过主行查询具体行的内容时可以使用索引组织表。事实上，IOT这方面的工作性能比群集好。

前面“嵌套表”部分的例子中为嵌套表定义了一个主键 PO_DTL，PO_DTL包括两个组成部分：NESTED_TABLE_ID和PO_DTL_ID。PO_DTL_ID是PO_DTL表的一个列，用于存储系统生成的标识符，类似于从“1”开始的递增整数，存放每一张购货单的标识符。NESTED TABLE ID是系统生成值的别名，作为由嵌套表行指向所引用的主表行的指针。

嵌套表的主码是将该表定义为索引组织表的基础，由 ORGANIZATION INDEX 声明指定该主码。查询中 IOT 返回的默认定位器事实上是存于表中的值，这个例子指定将返回一个定位器而不是一个值。这个定位器是作为一个指针来使用，以支持高性能的数据检索。

1.4.7 表分割

Oracle 8i 提供了将数据子集分割到彼此分离的物理存储地址的功能，目的是为了明显提高执行性能。当前发行的 Oracle 8i 版本只支持列级（例如，YEAR='1997'）的分割，而以后的版本将提供基于数据库函数（例如，START_DATA=Substr ('01-JAN-1997', 8,4)）计算的分割功能。提供这一功能是很有价值的，因为这样可以根据具体情况分割表。当前的实现需要用户存储一个冗余列，这个冗余列将存放事务被提交的年的值，这并不是我们所希望的做法。

设计表分割功能是为了增强存有大量数据的表的数据存取功能。通常方法是将公共行的数据一同存放到独立的物理分区中，这种方法是基于检索的典型方法。

例如，假设一家大银行推出信用卡业务，每个月产生数百万的事务处理。一种方法是

另一种方法就是按事务的 SIC 码分割表，这种划分方法的作用在于市场部能了解是谁在使用信用卡和最普遍的消费类型是什么。

划分TRANSACTIONS表的代码如 1-22所示。

代码 1-22

```
CREATE TABLE TRANSACTIONS (
TRANSACTION_ID          VARCHAR2 (15) NOT NULL,
CARDHOLDER              VARCHAR2 (60) NOT NULL,
SIC                     VARCHAR2 (5)  NOT NULL,
PURCHASE_AMT            NUMBER (9,2)  NOT NULL,
TRANSACTION_DATE DATE              NOT NULL)
/
```

如按年和月分割上述 TRANSACTIONS 表，其中事务开始于 1998 年 1 月并持续两个月，划分方法如代码 1-23 所示。

代码 1-23

```
CREATE TABLE TRANSACTIONS (
TRANSACTION_ID          VARCHAR2 (15)      NOT NULL,
CARDHOLDER              VARCHAR2 (60)      NOT NULL,
SIC                      VARCHAR2 (5)       NOT NULL,
```

```

PURCHASE_AMT          NUMBER (9,2)          NOT NULL,
TRANSACTION_DATE DATE ('MM-DD-YYYY')        NOT NULL)
MONTH_YEAR             VARCHAR2 (6)          NOT NULL)
PARTITION BY RANGE (MONTH_YEAR)
(PARTITION TRANS1 VALUES LESS THAN ('011998')
TABLESPACE TRANS1_TS,
(PARTITION TRANS2 VALUES LESS THAN ('021998')
TABLESPACE TRANS2_TS,
/

```

需要注意的是只需指定每一个分区范围的最大值，Oracle自动确定最小值。

MAXVALUE码用于代替VALUES LESS THAN子句中的具体值，将所有不满足规则的值存放到另外的分区中，正如代码 1-24所示。

代码 1-24

```

ALTER TABLE TRANSACTIONS ADD PARTITION
BY RANGE (MONTH_YEAR)
(PARTITION MAX VALUES LESS THAN (MAXVALUE))
TABLESPACE MAX_TS
/

```

注意，BY RANGE参数要求用一个列作为划分的数据源，这样就需要创建一个冗余属性名为MONTH_YEAR,在该属性中存放一个6个字符的值，前2个表示月份，后4个表示年（例如，MONTH_YEAR='011998'）。不允许在BY RANGE参数中指定函数（例如，PARTITION BY RANGE (SUBSTR (TRANSACTION_DATE , 1 , 2) ||SUBSTR (TRANSACTION_DATE , 7)) ）。

分区索引

可按局部的或全局的索引分割表。局部索引是建立一个索引，该索引按与定义表分割相同的方式分割。如果在TRANSACTIONS表上建立一个局部索引，代码如 1-25所示。

代码 1-25

```

CREATE INDEX TRANSACTIONS_BY_MONTH_IDX
ON TRANSACTIONS (MONTH_YEAR)
LOCAL
(PARTITION TRANS1          TABLESPACE TRANS1_IDX,
PARTITION TRANS2          TABLESPACE TRANS2_IDX,
PARTITION MAX              TABLESPACE MAX_IDX)
/

```

这个索引定义将建立三个独立的索引和一个分区，通过使用LOCAL关键字，为每个分区建立一个独立的索引。

索引创建命令中GLOBAL关键字允许指定索引值的范围，这一点不同于表分割。例如如果创建一个索引，该索引将1月份和2月份的事务合并到一个分区中，并创建第二个分区用于其他所有事务，语句如代码 1-26所示。

代码 1-26

```

CREATE INDEX TRANSACTIONS_BY_MONTH_IDX
ON TRANSACTIONS (MONTH_YEAR)

```

```
GLOBAL PARTITION BY RANGE (MONTH_YEAR)
(PARTITION TRANS1 VALUES LESS THAN ('021998')
TABLESPACE TRANS1_IDX,
PARTITION MAX VALUES LESS THAN (MAXVALUE)
TABLESPACE MAX_IDX)
/
```

1.4.8 方法

方法只不过是与某类型相关的存储过程和函数，建立方法的主要目的是为了数据检索。例如，可以在EMPLOYEE表中创建一个方法，通过减去 SYSDATE_HIRE_DATE的值来确定工作时间。

到目前为止，用于DML操作的唯一方法类型是采用构造器方法，即根据它自己的类型直接自动地创建一一对应。Oracle 8i以后的版本中将提供用户定义的构造器功能，这些用户定义的构造器方法将不直接对表的行执行DML，而是创建将被处理的行，并将其传给分析器。

封装

方法被看作是列，而不是存储过程和函数，因此，方法归入优先权/权限模式的SELECT部分中。这样，如果向一特定用户授予某一给定方法的存取权限，必须使用 GRANT SELECT... 语句。这与以前关于存储代码的思维方式有所区别，但人们很快就可以习惯这种方式。

尽管在以后的版本中会完善，Oracle 8i目前还不能完全支持封装。可能在经过一段相当长的时间后，能够实现将类型成员（列表、嵌入的列类型、方法、集合）标记为私有的和/或受限的，使其有能力拒绝在某些地方使用视图访问数据。而现在实现的功能只是用户能否访问整个对象。以后的工具将允许我们授予类型成员明确的访问权限。

当前的 Oracle 8i 版本提出了一个“调用者权限”的概念。在调用一个方法时，如果调用者在数据库的帐户中被授予了该调用权，则可以执行该方法。而之前只有具有定义该过程的定义者权限时，数据库才允许调用存储过程，这样的要求在开发过程中大概只会引起混乱。

1.5 面向对象方法的优点

从概念上讲，Oracle 8i提供的是与Oracle 6大致相同的环境。利用Oracle 6可以定义参照完整性约束，但并没有强制执行这些约束。同样，Oracle 8i中存在一些面向对象的结构，但没有把它们完全实现。

这并不意味着我们无法进行面向对象的设计，在某些方面，可以利用Oracle 8i中新的结构进行面向对象的设计。只要在思想上转变为面向对象的思考方式，也可以设计得很好，即便在实现面向对象的设计中使用传统的关系结构。

大约在1996年，程序设计人员开始使用面向对象的结构进行数据库的逻辑设计。面向对象方法的引入，使数据库设计的发展得到了很大的进步，特别是使模型中的实体（和最终表）的数量大量减少，这也是面向对象设计的最大优点。因为增强了结构的健壮性和灵活性，当用户提出新的无法回避的要求时，则大大降低了在数据模型上作较大修改的可能性。一些例子可以使更清楚地了解这一优点。

我们在设计一个零售系统时，从结构的角度看，许多小的业务事务处理模块都是相似的，也就是：购货需求、购货单、发货单、提货单、企业内部转运。利用面向对象的思想可

以认识到这些事务都属于货物流动，因此，不必为每一模块建立单独的结构，建立单一的名称为“货物流动”的结构可以使模型大为简化，而且这种模型的健壮性更好。在设计阶段，用户若提出了新的需求，如向供货商退货的功能，因为采用了面向对象方法，因此可在不修改数据模型的情况下就能方便地对程序作调整。在项目后期，客户若又决定再增加一个仓库，同样，不用做任何修改就可以适应这一变化。

在做大型金融方面的项目时，需要通过两种不同的聚集方式存储有价证券，原来使用的系统中通过定义两个结构来实现，一个用于主聚集，而另一个专门用于卷积结构，这两个独立结构使原有数据库含有 81 个实体。使用面向对象结构重新设计系统后，实体的总数量减为 27 个。这样，系统不仅更为精练，增强了可维护性，而且比原有系统增加了灵活性，提高了总体性能。

在一个大型后勤管理系统中，我们设计了购货单的核准程序，之后还为各个工程项目设计了一个相似的核准程序，这时总共需要设计 7~10 个不同的核准程序，而系统为此要在数据模型中增加大约 50 个表。于是我们决定设计一个通用的核准程序结构，而不是设计每一个核准程序，这个通用结构只需要使用 10 个表。

以上所有例子中，面向对象的思想都使得数据模型更为清晰，同时增加了系统的灵活性和可维护性。这些系统的开发工具使用的都是 Oracle Designer1.x，运行在 Oracle 7 上，这说明实现面向对象设计，可以用非完全面向对象工具来开发。