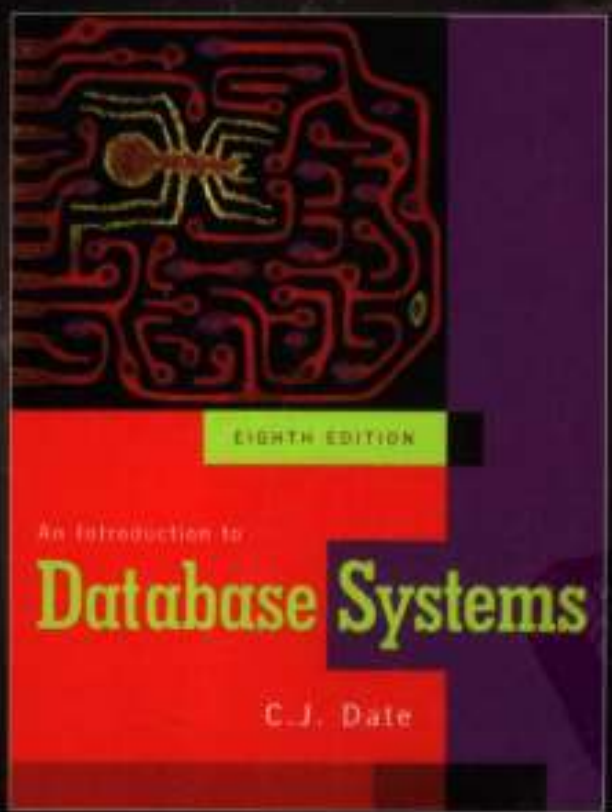
计 算 机 科 学 丛 书

原书第8版

# 数据库系统导论

(英) C. J. Date 著 孟小峰 王珊 姜芳艽 等译




An Introduction to Database Systems

Eighth Edition

机械工业出版社
China Machine Press

# 数据库系统导论（原书第8版）

本书一直是数据库方面的权威著作，深受对数据库系统原理感兴趣的读者的喜爱。本书现已升级至第8版，在继续保持了对数据库技术的基本内容提供翔实讲解的基础上，又对该领域的未来发展作出展望。

本书为读者提供了当前数据库系统的全面介绍，同时为该领域未来的发展指明了方向。第8版已经对目前数据库系统的最新发展内容进行了扩充，同时更加强调概念的理解，而不局限于公式的条陈。

本书的重点内容包括：

- SQL的内容已经更新到目前的最新标准。
- 为关系数据库提供了广泛深入的介绍。
- 对类型和域的本质的探讨已经成为独立的一章（第5章）。
- 第9章对完整性进行了彻底地重写。
- 对第15、16章进行了修订、扩充和完善。
- 包含了对事物的ACID特性，以及对一些未得到完全证明的结论的全面化的分析。
- 第20章（类型继承）和第23章（临时数据库）都进行了重写，以便更好地反映当今数据库研究的发展。
- 第27章（XML）介绍了数据库和XML标准的关系。
- 附录中修订了交换关系模型的概要、SQL语句中BNF语法、正文中重要的缩略语、缩略词和符号的术语表。


作者简介

**C. J. Date** 是关系数据库技术领域中非常著名的独立撰稿人、讲师、学者和顾问。在英国剑桥大学获得数学学士、硕士学位，在英国Montfort大学获得技术博士学位。30多年来，Date先生一直活跃在数据库领域，他著述丰富，先后发表了多篇技术性文章和研究论文，并写有多部数据库方面的著作。他还是《Database Programming & Design》和《Intelligent Enterprise》的专栏作家。





www.PearsonEd.com

上架指导：计算机/数据库

ISBN 978-7-111-21333-8

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

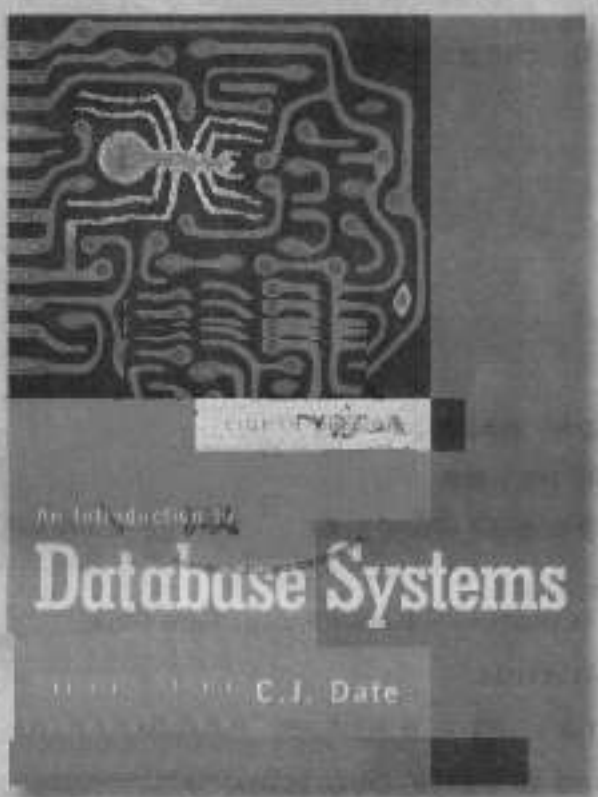
ISBN 978-7-111-21333-8
定价：75.00 元

TP392
8-3

计 算 机 科 学 丛

# 数据库系统导论

(英) C. J. Date 著 孟小峰 王珊 姜芳艽 等译




**An Introduction to Database Systems**

Eighth Edition

机械工业出版社
China Machine Press

本书全面介绍了现存应用广泛的数据库系统，为数据库技术基础知识提供坚实的基础，并对数据库领域的将来发展方向给出看法，本书一直是数据库方面的权威著作。本书整体上可以划分成六个主要部分：基本概念、关系模型、数据库设计、事务管理、高级专题、对象、关系和XML。第8版已经对数据库系统目前的系统的最新发展内容进行了扩充；同时又注重于强调概念的理解，而不仅局限于公式的条陈。

本书可用作计算机及业本科生和研究生学习数据库的教科书，也可供从事数据库研究工作的相关人员作为参考书。

Simplified Chinese edition copyright © 2007 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *An Introduction to Database Systems*, Eighth Edition (ISBN 0-321-19784-4) by C. J. Date, Copyright © 2004.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison Wesley.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2003-8523

图书在版编目（CIP）数据

数据库系统导论（原书第8版）/（英）戴特（Date, C. J.）著；孟小峰等译. -北京：机械工业出版社，2007.7

（计算机科学丛书）

书名原文：An Introduction to Database Systems, Eighth Edition

ISBN 978-7-111-21333-8

Ⅰ.数… Ⅱ.①戴…②孟… Ⅲ.数据库系统-教材 Ⅳ.TP311.13

中国版本图书馆CIP数据核字（2007）第055924号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：王 珊

北京京北制版厂印刷·新华书店北京发行所发行

2007年7月第1版第1次印刷

184mm×260mm·40印张

定价：75.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

## 对本书的赞誉

“C. J. Date 的书是关系理论和数学方法的一面旗帜……也是 SQL 标准的先驱。本书注重内容的翔实以及概念和原理的重要性，从而帮助读者掌握和理解本领域的知识。”

——Carl Eckberg，圣地亚哥州立大学

“本书第 8 版全面、极佳地展示了当代数据库领域。尤其是，Date 将类型、关系、对象数据库和对象 - 关系数据库等章节综合在一起，给出了关于数据库中对象—关系方法的非常清晰并且自包含的解释。”

——Martin K. Solomon，佛罗里达大西洋大学

“Chris Date 是计算机行业中数据库技术方面最受尊敬的专家和思想家，对于那些想对数据库系统做广泛了解的读者来说，他的这本《数据库系统导论》仍然是一本权威的著作，而且是一本数据库系统当前发展动向的指南。”

——Colin J. White，Intelligent Business Strategies 公司总裁

“这是我所看过的解释并发控制的最好的书，并且这本书囊括了本领域的方方面面。”

——Bruce O. Larsen，史蒂文斯科技学院

“……是一本不可缺少的读物和参考书。它是信息系统或数据库实践者的必备书籍。”

——Declan Brady，MICS，富士通公司系统评定和数据库专家

“除了对参考书目翔实的注释之外，作者对该领域的深入洞察、对深奥理论的浅显解释、对一些有争论问题的开放式讨论、以及书中全面的内容等，都使该书成为近 20 多年中数据库领域中最受欢迎的书。”

——Qiang Zhu，迪尔伯恩，密歇根大学

“［该书］最吸引人之处在于其全面性和给出了当前研究发展的最新内容。Date 之所以能够写出该领域的最新动态，得益于他参与了这些发展过程。”

——David Livingstone，纽斯卡尔，诺森比亚大学

信息原则（唯一表示的原则）：数据库全部的信息内容有且只有一种表示方式，也就是表中的行列位置有明确的值。

黄金规则：如果一个更新操作将使数据库处于违反自身某个谓词的状态，这样的更新是被禁止的。

（基本和导出的关系变量的）可交换性原则：基本和导出的关系变量没有任意的和不必要的区别。

数据库相对性原则：从用户的角度看，（a）所有的关系变量都是基本关系变量；（b）只要信息等价，数据库是“真实”的这一点是任意的。

规范化原则：

i. 一个非 5NF 的关系变量应能被分解为一系列的 5NF 的投影关系。

ii. 通过连接投影能重构最初的关系变量。

iii. 分解过程应保持独立性。

iv. 重构过程应使用每一个投影关系。

v. （……）只要所有的关系变量都符合 5NF，就停止规范化。

正交设计准则：如果 $A$ 和 $B$ 是数据库中任意的两个基关系变量，那么并不存在无损分解将 $A$ 和 $B$ 分为 $A1,\cdots,Am$ 和 $B1,\cdots,Bn$（相互独立）使在 $A1,\cdots,Am$ 中的某些投影 $Ai$ 和在 $B1,\cdots,Bn$ 中的某些投影 $Bj$ 有重叠的含义。

值可替代原则：不管系统在何处需要类型 $T$ 的值，类型 $T'$（$T'$ 是 $T$ 的子类型）的值总是可以被替换的。

分布式数据库的基本原则：对用户来说，一个分布式系统看起来应该完全像一个非分布式系统。

# 译 者 序

计算机技术的发展过程是一个不断创新的过程。没有任何一项技术的发展能够像计算机技术的发展这样，其创新总是让人应接不暇，其进步总是让人不得歇息的机会。数据库技术是计算机科学技术中发展最快的领域之一，也是应用最广的技术之一，它是计算机信息系统与应用系统的核心技术和重要基础。数据库作为计算机专业及相关专业的核心课程，国内外已有大量与此相关的教科书出版。初步分来，不外以下几类：一是以导论和概论为主的综合类，这些书适合低年级计算机本科的入门教育；二是较为深入的数据库实现技术介绍，这些书适合高年级本科和低年级研究生的提高教育；三是面向某些数据库专门技术的专深内容的讲解，适合数据库研究方向的高年级研究生使用。

近30年来，戴特（C. J. Date）博士的这本《数据库系统导论》（《An Introduction to Database Systems》）一直是数据库方面的权威著作，经久不衰，深受读者的喜爱。作者戴特博士是关系数据库技术领域中非常著名的独立撰稿人、学者和顾问。多年来，戴特先生一直活跃在数据库领域，他是最早认识到卡德博士（E. F. Codd）在关系模型方面所做的开创性贡献的学者之一。此次推出的最新版在继续保持了对数据库技术的基本内容提供翔实讲解的基础上，又对该领域的最新发展进行了介绍。正如Michigan大学的Qiang Zhu博士对该书的评论中提到的“（本书）除了对参考书目的翔实的注释之外，作者对该领域的深入洞察、对深奥理论的浅显的解释、对一些有争论问题的开放式讨论、以及书中有关前沿的内容等，都使该书成为近二十多年中数据库领域中最受欢迎的书。”的确，戴特博士这种深入浅出的写作风格和对大量第一手资料的充分掌握，是本书长久不衰，生命力得以不断延续的根本所在。

大约在20多年前，本书的第2版由科学出版社翻译出版，第7版由机械工业出版社华章分社出版，此次翻译的是本书的第8版。在翻译过程中，本人深感书的内容变化颇大。虽多年从事数据库的教学工作，借本次翻译之机，犹如重新充电。尤其书中对许多传统概念的重新认识和讲解（如关系模型、视图更新等）对本人启发很大。教书不能一成不变，教好书尤其要推陈出新。依据翻译的感受和使用本书一学期的教学实践，归纳起来，我认为本书具有如下的突出的特点：

**第一，注重基础知识的讲解。**每一本有关数据库的书都有其侧重点，每一个作者也都有其所擅长的方面，本书所强调的主要是基础。要想在基础之上做一些事情，必须要牢牢地把握好基础，并且对其有正确的理解。戴特博士一贯认为关系数据模型是现代数据库技术的基础，它使这一领域具有了科学性。如今任何有关数据库技术基础的书籍，若没有包括关系模型，那它是不全面的。同样，如果没有深入理解关系模型，而声称是数据库领域的专家，那也是无法让人接受的。为此本书用长达六章的篇幅来详细介绍关系模型的有关概念（第7版为五章）。

**第二，注重用全新的观点和视角认识新旧问题。**戴特博士在书的前言中特别引用了一段话“人们可以获得一些确定性的知识，人们还可以从新的发现中更正那些以前所接受的错误的观点，使其更为公正，正是从这个意义上来讲，笔者认为自己所一直信奉并且尽力保持的这种不断改变自己观点的哲学方法是科学的。对于笔者说的话，不管是以前说过的还是以后将要说的，笔者并不像神学家信奉他们的信条那样认为它们是真理。笔者所主张的只是在当时为大多数人所接受的较为明智的看法……”我十分钦佩作者的这种精神。比如，关系模型不是一个静态的事物，理解这一点是非常重要的——过去几年里它不断发展和进化，未来仍将如此。作者用他在《对象关系数据库基础：第三版宣言》（1998）中的观点，重新解释了关系模型的含义，并与传统定义相比较。特别地，书中对类型（域）、关系值和关系变量、完整性，谓词和视图等内容都提出了大量新的观点。基于新的解释，作者对视图更新这一传统难题给予了更为清晰的解释，在教与学两个环节上都更便于把握。再比如对传统的规范化理论，增加了新的关系值的属性、逆规范化

(denormalization)、正交设计(orthogonal design)和语义模型等有关内容(包括“商业规则”)。

**第三,注重分析问题的本质,带给读者以思考的空间。**比如第18章信息空缺,先描述空值和3VL的基本思想,然后讨论了这些思想所带来的一些严重后果,以此来证明空值和3VL是非常严重的错误。在关系模型这样清晰规范的系统中应该没有它们的位置,但作者认为对空值和3VL不加介绍是不合适的,只有通过分析批判,才能给读者更大的认识空间。同样针对“对象数据库/对象关系数据库为什么不能取代关系数据库”这一问题,作者用大量的分析比较,论述的十分精彩。特别是“**对象DBMS还是一个DBMS吗?**”,一语道破对象数据库的本质。

**第四,注重反映数据库的新技术。**数据库学科的发展可谓突飞猛进。第8版注重补充数据库新的成果,但又不哗众取宠,取舍得当是这部分的关键。如第18章(空缺信息),第20章(类型继承),第22章(决策支持),第23章(时态数据库),第24章(逻辑数据库),和第21章(分布式数据库),第25、26章(对象数据库/对象关系数据库)等,都是数据库的最新的且相对成熟的技术。

**最后,本书提供了丰富翔实的参考文献和注释。**这也是本书一贯的风格和传统。它几乎成为一种书写体例为大家所广泛接受。这些参考书目反映了以前和现在该领域的研究状况,而且对有兴趣的读者还提供了好的信息资源。我本人就获益匪浅,当我要查找一个经典文献的出处时,最先想到的就是戴特博士的这本书。

当然本书也不是没有缺点。可以看出,戴特博士在尽其所能延续这本书的经典地位,但毕竟戴特博士脱离数据库学术界和大学教学一线多年,其对主流技术的把握略显生疏,比如对数据库实现技术无所涉及,但好在这方面已有其他很好的教材可以参考。

全书主要包括六个部分和三个附录。第一部分主要介绍了数据库系统中的一般概念和关系数据库系统。第二部分详细介绍了关系模型这一数据库领域的基础理论。第三部分讨论了数据库设计的理论和实践中的一些问题。第四部分主要介绍了事务处理,包括恢复和并发控制。第五部分讨论了数据库的一些高级主题,包括:安全性、优化、类型继承、分布式数据库、时态数据库、逻辑数据库等。最后,第六部分讲解了对象关系数据库技术以及XML数据管理技术。

本书的翻译和审校由孟小峰和王珊共同组织完成。参加翻译的还有姜芳艽、尹少宜、赖彩凤、李先、凌妍妍、王凌、陈继东、刘伟、周军锋、黄静、贾琳琳等。全书由孟小峰负责统一定稿。

本书涉及面广,内容丰富,术语量大,由于译者水平有限,译文中不当之处在所难免。诚恳读者批评指正并不吝赐教。如果你有任何建议或意见,欢迎发E-mail至xfmeng@public.bta.net.cn。

译　者

二〇〇七年五月于北京

# 译者简介



孟小峰，1964 年生，博士、教授、博士生导师，中国人民大学信息学院副院长兼计算机系主任。现为中国计算机学会理事、中国计算机学会普及工委主任，中国计算机学会数据库专委会委员秘书长、电子政务与办公自动化专委会副主任委员，《计算机研究与发展》编委等。主持或参加过二十多项国家科技攻关项目、国家自然科学基金以及国家 863 项目。获电子部科技进步特等奖 1 项、北京市科技进步二等奖 2 项、第七届“中创软件人才奖”、“教育部新世纪优秀人才支持计划”等奖励。研制开发的主要软件产品有国产数据库系统 COBASE、嵌入式移动数据库系统“小金灵”、中文自然语言查询系统 NChiql、并行数据库系统 PBASE/1、Native XML 数据库系统 OrientX 等。近年在国内外杂志及国际会议发表论文 100 余篇，出版数据库方面的著译作七部，其参与主讲的“数据库系统概论”获评国家精品课程，并多次应邀担任国际会议 SIGMOD、ICDE、DASFAA、MDM、DEXA 等程序委员及各类主席。近期主要研究领域为 Web 数据集成、XML 数据库、移动数据库等。

http://www.ccf-dbs.org.cn/idkc/xfmeng

http://idke.ruc.edu.cn/wamdm/

# 第8版前言

本书全面介绍了现在广泛应用的数据库系统。本书提供了坚实的数据库技术基础知识，并对数据库领域未来的发展方向给出看法。本书适合作为课本，而不是参考书（当然，在某种程度上本书作为参考书会很有用）。学习本书重点要**理解**，而不是机械的记忆。

## 本书前提

本书适合任何对计算技术有专业兴趣并在某些方面也想理解数据库系统内涵的读者。作者假设读者应该至少对下面两方面的知识有所了解：

1）现代计算机系统的存储和文件管理（索引等）知识。

2）至少一门高级编程语言的特征（Java、Pascal、PL/I等）。

关于第一个前提知识，请注意本书的在线附录D“存储结构和存取方法”中有详细的指南。

## 结构

必须承认本书比较厚。事实上数据库技术已经发展成了一个非常庞大的领域，以普通的篇幅进行全面介绍是不太可能的（实际上，大多数同类题材的书都在1000页左右）。本书整体上可以划分成6个主要部分：

1）基本概念。

2）关系模型。

3）数据库设计。

4）事务管理。

5）高级专题。

6）对象、关系和XML。

每一部分依次被划分成几个章节：

第一部分（共4章）全面介绍数据库系统的一般概念，特别是关系数据库系统，并介绍了标准数据库语言SQL。

第二部分（共6章）详尽地介绍**关系模型**，它不但是关系数据库系统的理论基础，实际上更是整个数据库系统的理论基础。

第三部分（共4章）讨论**数据库设计**的一般问题；其中3章介绍设计理论，1章讲解语义模型和实体/关系模型。

第四部分（共2章）涉及**事务管理**（也就是恢复和并发控制）。

第五部分（共8章）内容有点杂，但总体说来是介绍关系的概念和数据库技术未来若干方面的联系——**安全性**、**分布式数据库**、**时态数据**、**决策支持**等。

最后，第六部分（共3章）描述了数据库系统中的**对象技术**。第25章特别介绍了**对象系统**；第26章考虑对象和关系技术共存的可能性，并且讨论**对象/关系**系统；第27章讨论XML和数据库的相关性。

本书有3个附录。附录A概述一种全新的完全不同的实现技术，称作**TransRelational™模型**；附录B给出SQL表达式的**BNF范式**；附录C是**术语表**，包括重要的缩略语、首字母缩写词和本书用到的符号。

## 在线材料[⊖]

- 教师指导手册提供了如何在教数据库课程时使用本书的指导。它包括一系列针对每部分、

---

⊖ 本教师补充资料只有符合条件的教师才能得到，有需要者请与Pearson公司北京办事处联系，联系电话：8610-88819178或8008100855。——编者注。

每章的笔记、提示和建议，还有附录和其他的辅导资料。

- 习题答案（包含在教师指导手册中）。
- ppt 格式的幻灯片。

每个读者都可以从网址 http://www.aw.com/cssupport 得到下面的材料：

- 关于存储结构和存取方法的附录 D（前面提到过）。
- 部分习题的解答。

## 如何学习本书

读者大体上应该按照本书写作的顺序阅读，但也可以跳过后面的章节，建议第一次阅读本书的顺序是：

- 速读第 1 章和第 2 章。
- 仔细阅读第 3、4 章（4.6 节和 4.7 节可以是例外）。
- 速读第 5 章。
- 仔细阅读第 6、7、9、10 章，但可以跳过第 8 章（8.6 节的 SQL 部分应该看看）。
- 浏览第 11 章。
- 仔细阅读第 12、14 章，[⊖] 可以跳过 13 章。
- 仔细阅读第 15、16 章（可能除了 15.6 节介绍的两阶段提交部分）。
- 可以根据个人的喜好和兴趣，有选择地阅读随后的章节（但要按照前后顺序读）。

每章从引言开始，以小结结束，每章都附有课后习题，本书的配套网站上提供的在线答案还包括习题知识点之外的其他信息。大多数章节还包括一些参考文献，其中很多都有注解。这种结构便于将介绍的主题组织成层次的形式，书的主体包括重要的概念和结果，各种辅助问题和更复杂的方面放在习题、答案或者参考文献中（如果合适的话）。*注意*：参考文献通过一个方括号中的两部分的数字来区分。例如，参考文献［3.1］表示第 3 章最后所列出的参考文献中的第一个，也就是 E. F. Codd 于 1982 年 2 月发表在 CACM 25 第 2 期上的论文。（参考文献中的缩略语，比如“CACM”，其解释参见附录 C。）

## 和早期版本的比较

本书和以前版本的主要区别是：

- 第一部分：新版的第 1 ~ 4 章和第 7 版的第 1 ~ 4 章内容大体一致，但在细节上有重大的修订。特别是第 4 章，对 SQL 的介绍升级到当前标准 SQL：1999，并以标准贯穿整书。（因此，本版与第 7 版相比，有超过半数章节进行了修订。）*注意*：还有的内容有可能纳入下一个 SQL 标准版本中——2003 年底的时候可能通过——在适当的时候都有所提及。
- 第二部分：第 5 ~ 10 章是关于关系模型的内容，和第 7 版的第 5 ~ 9 章相比，这部分整体被重写了，还扩充了很多内容，有了很大程度的改进。特别是关于类型（俗称为域）的介绍，被扩充为一章（第 5 章），关于完整性的介绍（第 9 章）我彻底地调整了结构并进行了重写。我在短时间内完成这部分章节的增改，但更新的内容，没有表现在底层概念上，而是表现在我选择如何描述它们的方式上，这些改变是以我在教授这部分内容时描述它们的实践经历为基础的。

  *注意*：这里有必要作进一步的解释。本书的早期版本用 SQL 作为教授关系概念的基础，这是因为我们相信，具体的东西比抽象的东西更容易展示。但不幸的是，SQL 和关系模型之间的鸿沟扩大了，并且这一鸿沟有继续扩大的趋势，最终会达到一种状况，再这样继续下去，有可能会误导我们对 SQL 的使用。可悲的是，SQL 越来越远离作为关系原理的具体体现，它包含了太多不合理的地方——冗长、功能太多，坦率地说我压根都不想讨论它。但是，从商业的角度看 SQL 显然很重要，因此每个数据库专业人员都有必要熟悉它，在本书中忽略这个问题也是不恰当的。因此在书中介绍了我的解决策略：（a）第一部分有一章来介绍 SQL 基础；（b）其他章中有单独的部分描述该章讨论的主题涉及的 SQL 诸方面。通过这种方法本书提供了全面的、广泛的 SQL 材料，这些材料放在我觉得

---

⊖ 如果愿意可以提前读第 14 章，不过最好先读完第 4 章。

合适的上下文中。

- 第三部分：第 10 ~ 13 章是第 7 版中第 9 ~ 12 章的完美修订。在内容细节上有全面的改善。

  注意：一些进一步的解释是有必要的。一些批评家抱怨说早期版本关于数据库设计主题的讨论太靠后了，但我认为是学生们还没有准备好来正确地设计数据库或者来充分理解设计主题，除非他们初步理解了数据库是什么和数据库是如何应用的；换句话说，我认为在向学生提出设计问题前，在关系模型和相关主题上花些时间是很重要的。因此我仍然认为第三部分的位置是合适的。（的确，我知道有些老师愿意提前教授实体/关系模型，因此我尽量使第 14 章自成一体，这样他们就可以在讲完第 4 章后马上讲述第 14 章了。）
- 第四部分：这部分的两章，第 15、16 章，是从第 7 版的第 14、15 章修订而来的，是完全的重写、扩展和改进。特别是第 16 章，现在包括对所谓的事务的 ACID 属性的仔细分析和一些相关的非正统的结论。
- 第五部分：第 20 章关于类型继承和第 23 章关于时态数据库都全部重写，以反映相关领域的最新研究和发展。对其他章节的修订都是些美化工作，不过在解释和举例上有所改善，并且多处引用了新材料。
- 第六部分：第 25 和 26 章是第 7 版的第 24 和 25 章的改善和扩展。第 27 章是全新的。

最后，附录 A 也是新的，附录 B 和 C 分别是对第 7 版的附录 A 和 C 的修订（第 7 版的附录 B 的材料已经合并到本书的主体部分了）。

## 本书的特色

市场上的每本数据库书籍都有各自的优缺点，并且每个作者都是有针对性地著书。一个集中于事务管理问题，一个强调实体/关系模型；另一个仔细分析 SQL 的各个方面，也有以纯粹“对象”的观点来介绍数据库的，还有专门讨论一些商业数据库产品的，等等。当然我也不例外，我也有把削减题材的斧头：也许是可以被称为基础的斧头。我坚定地相信，当我们在基础之上再有所发展之前，必须打好基础，对它有恰当的理解。这种信念解释了我为什么在本书中如此强调关系模型；特别是第二部分的篇幅——本书中最重要的部分——在这部分中我尽可能谨慎地提出了我对关系模型的理解。我对基础很感兴趣，而不是时尚或流行的东西。产品总是在改变，但原则更持久。

在这方面，我希望能提醒你注意这个事实，对某些很重要的主题（基础），本书使用一个完整、深入的章（或者附录）来介绍。讨论的主题有：

- 类型
- 完整性
- 视图
- 缺失信息
- 继承
- 时态数据库
- TransRelational™模型

基于同样的观点（基础的重要性），我必须承认本书的整体叙述方式这几年来已经变了。前几版主要是自然的描述；描述该领域在实践中实际的样子，“尽管有各种缺点与瑕疵”。后来的版本，相对而言说明性更多些；讨论的是该领域应该是什么样子和未来可能的发展方向，如果我们行事正确的话。现在的版本在这个意义上的确是说明性的了（因此这是带有个人观点的教材）。既然“行事正确”的第一部分是教育人什么是正确的事，我希望，本书的新版本能在这方面有所帮助。

另一个相关的观点：也许你也知道，我最近和我的同事 Hugh Darwen 合作出版了另一本“说明性”的书，《Foundation for Future Database Systems: The Third Manifesto》（本书参考文献[3.3]）[⊖]。我们称该书为《第三个宣言》，或者简称为《宣言》，它基于关系模型为未来的数据库系统提供详细的技术上的建议；它是 Hugh 和我多年教学与思考这些问题的结晶。并且，《宣

---

⊖ 本书有一个相关站点是 http://www.thethirdmanifesto.com 更多的相关材料也见 http://www.dbdebunk.com

言》的思想渗透在本书中。这并不是说阅读《宣言》是阅读本书的必要前提，但它的确是和本书涉及的内容有很大的直接相关性，并且在那里经常会发现更进一步的相关信息。

注意：参考文献［3.3］使用 **Tutorial D** 语言，用来解释目的，本书也是一样。Tutorial D 的语法和语义多多少少是自解释的（该语言可能是描述特征的，自由松散的，像“Pascal”之类），但是个别的特征在第一次使用的时候会有所解释，如果有必要的话。

## 结束语

我愿意用下面略微修改过的另一篇序言来结束这个前言——伯兰特·罗素的《伯兰特·罗素关于智力、物质和道德辞典》。

*我被指责有不停改变观点的恶习……我自己丝毫不以之（前述的习惯）为耻。活跃于 1900 年的哪个唯物论者能自夸在最后的半个世纪他的观点从没改变过？……我所重视并且努力去追求的哲学是科学，从某种意义上讲就是要获得真实的知识，新的发现会使任何聪慧的头脑都不可避免地承认先前的错误。至于我曾经说过的，无论说得早或晚，我不会像那些空头理论家出于他们的信念那样宣称为真理。我只能说我所表达的观点目前来说是明智合理的……如果后续的研究不能表明它需要修改的话，我会很惊讶的。这种观点不是傲慢武断的声明，而是我目前能做到的更接近于清楚和正确思考的最好声明。文字的清晰是我追求的最终目的。*

如果你比较过本书第 8 版和以前的版本，你会发现我在许多事情上已经改变了观点（毫无疑问还会继续改变下去）。我希望你能接受上面引用的评论，来恰当地判断这件事的目前状态。我赞同伯兰特·罗素关于科学质询是什么的观点，但是他对此观点的描述比我所能描述的更加雄辩、更有说服力。

## 致谢

我很乐意再一次感谢在本书写作过程中有过直接或间接参与的许多人：

首先，我必须感谢我的朋友 David McGoveran 和 Nick Tindall，感谢他们在本书出版中花费的心血；David 提供了第 22 章关于决策支持的初稿，Nick 提供了第 27 章关于 XML 的初稿。我也必须感谢我的朋友和同事 Hugh Darwen 对手稿中所有 SQL 部分的主要帮助（多种形式的帮助）。Nagraj Alur 和 Fabian Pascal 提供给我各种各样的技术背景材料。特别感谢 Steve Tarin 发明了附录 A 中描述的技术，并且感谢他帮助我彻底理解了该技术。

其次，本书的内容受益于过去几年中我教过的学生在研讨会上提出的意见，也颇得益于许多朋友和评论家的意见和讨论，包括 IBM 公司的 Hugh Darwen，根特大学的 Guy de Tré，圣地亚哥州立大学的 Carl Eckberg，Rensselaer 工艺美术学院的 Cheng Hsu，密歇根大学迪尔伯特分校的 Abdul-Rahman Itani，波士顿大学的 Vijay Kanabar，史蒂文斯技术研究所的 Bruce O. Larsen，纽卡斯尔 Northumbria 大学的 David Livingstone，Alternative Technologies 的 David McGoveran，IBM 公司的 Steve Miller，独立咨询员 Fabian Pascal，佛罗里达亚特兰大大学的 Martin K. Solomon，Required Technologies 的 Steve Tarin 和 IBM 公司的 Nick Tindall。上述名单中的每一位或者至少校阅了手稿中的某些部分，或者提供了技术材料，或者帮我找到了许多技术问题的答案，我很感谢他们。

我要感谢我的妻子 Lindy 又帮我设计了书的封面，感谢她在我写本书和这么多年来所有其他数据库相关书籍过程中一贯的支持。

最后，我要感谢 Addison-Wesley 公司的 Maite Suarez-Rivas 和 Katherine Harutunian，感谢他们在我写这本书时所给予的鼓励和支持，同时本书设计者 Elisabeth Beller 的辛勤工作。

C. J. Date
Healdsburg，California，2003

# 目　录


译者序
译者简介
第8版前言

## 第一部分　基础知识

第1章　数据库管理概述 …… 2
1.1　引言 …… 2
1.2　数据库系统的构成 …… 4
1.3　数据库的内涵 …… 6
1.4　使用数据库的优点 …… 9
1.5　数据独立性 …… 11
1.6　关系系统及其他数据库系统 …… 14
1.7　小结 …… 16
习题 …… 16
参考文献 …… 17

第2章　数据库系统体系结构 …… 18
2.1　引言 …… 18
2.2　三级体系结构 …… 18
2.3　外部层 …… 19
2.4　概念层 …… 21
2.5　内部层 …… 22
2.6　映像 …… 22
2.7　数据库管理员 …… 23
2.8　数据库管理系统 …… 24
2.9　数据通信 …… 26
2.10　客户/服务器体系结构 …… 26
2.11　实用程序 …… 27
2.12　分布式处理 …… 27
2.13　小结 …… 29
习题 …… 29
参考文献 …… 30

第3章　关系数据库简介 …… 32
3.1　引言 …… 32
3.2　关系模型概述 …… 32
3.3　关系和关系变量 …… 34
3.4　关系的含义 …… 36
3.5　优化 …… 37
3.6　数据字典 …… 38
3.7　基本关系变量和视图 …… 39
3.8　事务 …… 41
3.9　供应商和零件数据库 …… 42
3.10　小结 …… 43
习题 …… 44
参考文献 …… 45

第4章　SQL简介 …… 47
4.1　引言 …… 47
4.2　SQL基本操作 …… 47
4.3　目录 …… 49
4.4　视图 …… 50
4.5　事务 …… 50
4.6　嵌入式SQL …… 51
4.7　动态SQL和SQL/CLI …… 55
4.8　SQL并不完美 …… 57
4.9　小结 …… 57
习题 …… 58
参考文献 …… 59

## 第二部分　关系模型

第5章　类型 …… 64
5.1　引言 …… 64
5.2　值与变量 …… 64
5.3　类型与表示 …… 66
5.4　类型定义 …… 69
5.5　操作符 …… 70
5.6　类型生成子 …… 74
5.7　SQL支持 …… 75
5.8　小结 …… 80
习题 …… 81
参考文献 …… 82

第6章　关系 …… 83
6.1　引言 …… 83
6.2　元组 …… 83
6.3　关系类型 …… 86
6.4　关系的值 …… 87
6.5　关系变量 …… 92
6.6　SQL的支持 …… 96
6.7　小结 …… 100
习题 …… 101
参考文献 …… 102

第 7 章　关系代数 …… 105
7.1　引言 …… 105
7.2　关系封闭性 …… 106
7.3　基本代数：语法 …… 107
7.4　基本代数：语义 …… 109
7.5　举例 …… 115
7.6　关系代数的作用 …… 117
7.7　深入讨论 …… 118
7.8　附加的操作符 …… 119
7.9　分组与解组 …… 124
7.10　小结 …… 126
习题 …… 126
参考文献 …… 128

第 8 章　关系演算 …… 130

8.1　引言 …… 130
8.2　元组演算 …… 131
8.3　举例 …… 136
8.4　关系演算与关系代数的比较 …… 138
8.5　计算能力 …… 141
8.6　SQL 语言 …… 142
8.7　域演算 …… 149
8.8　QBE …… 151
8.9　小结 …… 154
习题 …… 155
参考文献 …… 156

第 9 章　完整性 …… 158
9.1　引言 …… 158
9.2　进一步讨论 …… 159
9.3　谓词和命题 …… 161
9.4　关系变量谓词和数据库谓词 …… 162
9.5　约束检查 …… 162
9.6　内部谓词与外部谓词 …… 163
9.7　正确性与一致性 …… 164
9.8　完整性和视图 …… 165
9.9　约束分类模式 …… 166
9.10　码 …… 167
9.11　触发器 …… 172
9.12　SQL 的支持 …… 174
9.13　小结 …… 177
习题 …… 178
参考文献 …… 180

第 10 章　视图 …… 185
10.1　引言 …… 185
10.2　视图的用途 …… 187
10.3　视图检索 …… 189
10.4　视图更新 …… 189
10.5　快照 …… 199
10.6　SQL 对视图的支持 …… 200
10.7　小结 …… 202
习题 …… 202
参考文献 …… 203

## 第三部分　数据库设计

第 11 章　函数依赖 …… 209
11.1　引言 …… 209
11.2　基本概念 …… 209
11.3　平凡的函数依赖和非平凡的函数依赖 …… 211
11.4　依赖集的闭包 …… 211
11.5　属性集的闭包 …… 212
11.6　最小函数依赖集 …… 214
11.7　小结 …… 215
习题 …… 216
参考文献 …… 217

第 12 章　进一步规范化Ⅰ：1NF、2NF、3NF 和 BCNF …… 219
12.1　引言 …… 219
12.2　无损分解和函数依赖 …… 221
12.3　第一、第二和第三范式 …… 223
12.4　保持函数依赖 …… 228
12.5　BOYCE/CODD 范式 …… 229
12.6　具有关系值属性的关系变量 …… 233
12.7　小结 …… 234
习题 …… 234
参考文献 …… 236

第 13 章　进一步规范化Ⅱ：高级范式 … 238
13.1　引言 …… 238
13.2　多值依赖与第四范式 …… 238
13.3　连接依赖与第五范式 …… 241
13.4　规范化过程小结 …… 244
13.5　逆规范化 …… 246
13.6　正交设计 …… 247
13.7　其他的规范化形式 …… 249
13.8　小结 …… 250
习题 …… 250
参考文献 …… 251

第 14 章　语义建模 …… 256
14.1　引言 …… 256
14.2　总体方法 …… 257
14.3　E/R 模型 …… 258
14.4　E/R 图 …… 261
14.5　基于 E/R 模型的数据库设计 …… 262
14.6　简单分析 …… 265
14.7　小结 …… 267
习题 …… 268
参考文献 …… 269

## 第四部分 事务管理


第 15 章 恢复 …… 280
15.1 引言 …… 280
15.2 事务 …… 280
15.3 事务恢复 …… 283
15.4 系统恢复 …… 284
15.5 介质恢复 …… 286
15.6 两阶段提交 …… 286
15.7 保存点 …… 287
15.8 SQL 对事务的支持 …… 287
15.9 小结 …… 288
习题 …… 288
参考文献 …… 289

第 16 章 并发 …… 292
16.1 引言 …… 292
16.2 三个并发问题 …… 292
16.3 锁 …… 294
16.4 重提三个并发问题 …… 295
16.5 死锁 …… 297
16.6 可串行性 …… 298
16.7 重提恢复 …… 299
16.8 隔离级别 …… 300
16.9 意向锁 …… 301
16.10 ACID 的不足之处 …… 303
16.11 SQL 的支持 …… 306
16.12 小结 …… 306
习题 …… 307
参考文献 …… 308


## 第五部分 高级专题


第 17 章 安全性 …… 314
17.1 引言 …… 314
17.2 自主存取控制 …… 315
17.3 强制存取控制 …… 319
17.4 统计数据库 …… 320
17.5 数据加密 …… 324
17.6 SQL 的支持 …… 327
17.7 小结 …… 329
习题 …… 330
参考文献 …… 330

第 18 章 优化 …… 333
18.1 引言 …… 333
18.2 一个启发性的示例 …… 334
18.3 查询处理概述 …… 334
18.4 表达式转换 …… 337
18.5 数据库统计信息 …… 340
18.6 分而治之的策略 …… 341
18.7 关系操作的实现算法 …… 343
18.8 小结 …… 346
习题 …… 347
参考文献 …… 349

第 19 章 信息空缺 …… 362
19.1 引言 …… 362
19.2 3VL 方法概述 …… 363
19.3 上述方案所造成的某些结果 …… 367
19.4 空值和码 …… 369
19.5 外连接 …… 370
19.6 特殊值 …… 372
19.7 SQL 的支持 …… 372
19.8 小结 …… 376
习题 …… 376
参考文献 …… 378

第 20 章 类型继承 …… 381
20.1 引言 …… 381
20.2 类型的层次结构 …… 383
20.3 多态性和可置换性 …… 386
20.4 变量与赋值 …… 388
20.5 约束特化 …… 391
20.6 比较 …… 392
20.7 操作、版本和签名 …… 395
20.8 圆是椭圆吗 …… 397
20.9 再论约束特化 …… 400
20.10 SQL 的支持 …… 401
20.11 小结 …… 404
习题 …… 405
参考文献 …… 406

第 21 章 分布式数据库 …… 409
21.1 引言 …… 409
21.2 预备知识 …… 409
21.3 十二个目标 …… 411
21.4 分布式系统面对的问题 …… 416
21.5 客户/服务器系统 …… 424
21.6 DBMS 独立性 …… 426
21.7 SQL 的支持 …… 429
21.8 小结 …… 429
习题 …… 430
参考文献 …… 430

第 22 章 决策支持 …… 436
22.1 引言 …… 436
22.2 决策支持的特征 …… 437
22.3 决策支持的数据库设计 …… 438
22.4 数据准备 …… 442
22.5 数据仓库和数据集市 …… 443
22.6 联机分析处理 …… 446
22.7 数据挖掘 …… 451
22.8 SQL 的支持 …… 452

22.9 小结 …… 453
习题 …… 453
参考文献 …… 454

第 23 章 时态数据库 …… 457
23.1 引言 …… 457
23.2 问题是什么 …… 459
23.3 时间区间 …… 463
23.4 归并和反归并关系 …… 467
23.5 关系操作符推广 …… 474
23.6 数据库设计 …… 478
23.7 完整性约束 …… 482
23.8 小结 …… 486
习题 …… 487
参考文献 …… 488

第 24 章 基于逻辑的数据库 …… 489
24.1 引言 …… 489
24.2 综述 …… 489
24.3 命题演算 …… 491
24.4 谓词演算 …… 494
24.5 数据库的证明理论观点 …… 498
24.6 演绎数据库系统 …… 501
24.7 递归查询过程 …… 504
24.8 小结 …… 509
习题 …… 510
参考文献 …… 511

## 第六部分 对象、关系和 XML

第 25 章 对象数据库 …… 516
25.1 引言 …… 516
25.2 对象、类、方法和消息 …… 518
25.3 进一步的分析 …… 521
25.4 一个详实的示例 …… 526
25.5 混合性问题 …… 533
25.6 小结 …… 538
习题 …… 539
参考文献 …… 540

第 26 章 对象/关系数据库 …… 546
26.1 引言 …… 546
26.2 第一个根本性错误 …… 548
26.3 第二个根本性错误 …… 552
26.4 实现上的问题 …… 554
26.5 真正融合的好处 …… 556
26.6 SQL 工具 …… 557
26.7 小结 …… 561
习题 …… 562
参考文献 …… 562

第 27 章 互联网与 XML …… 569
27.1 引言 …… 569
27.2 万维网和因特网 …… 569
27.3 XML 综述 …… 570
27.4 XML 数据定义 …… 577
27.5 XML 数据操纵 …… 583
27.6 XML 和数据库 …… 589
27.7 SQL 的支持 …… 590
27.8 小结 …… 594
习题 …… 595
参考文献 …… 596

## 附 录

附录 A TransRelational 模型 …… 599
附录 B SQL 表达式 …… 613
附录 C 缩略语和符号 …… 620

# 第一部分　基础知识

第一部分包括以下4章：

- 第1章介绍什么是数据库和为什么需要数据库系统。该章还简述了关系数据库系统和其他数据库系统之间的不同之处。
- 第2章介绍数据库系统的一般体系结构，即ANSI/SPARC体系结构。该体系结构构成本书其他部分的基本框架。
- 第3章概括介绍了关系系统。其目的是为在第二部分和后续章节中进一步讨论奠定基础。该章还介绍并解释了一个运行实例：供应商和零件数据库。
- 最后，第4章介绍了标准关系语言SQL（更精确地说是SQL：1999）。

# 第 1 章　数据库管理概述

## 1.1　引言

**数据库系统**本质上是一个用计算机存储记录的系统。**数据库**本身可被看作为一种电子文件柜；也就是说，它是收集计算机化数据文件的仓库或容器。系统用户可以对这些文件执行（或要求系统执行）一系列操作，例如：

- 向数据库中增加新的文件
- 向现有文件中插入数据
- 从现有文件中检索数据
- 删除现有文件中的数据
- 更改现有文件中的数据
- 删除数据库中的现有文件

图 1-1 显示了一个名为 CELLAR（酒窖）的小型数据库，它只包含一个文件。文件中依次包含各种酒的藏酒量情况。图 1-2 显示了一个该数据库的**检索**操作，以及该操作返回的数据。（注意：为清晰起见，在本书中，我们采用大写字母来表示数据库的操作名、文件名以及其他类似内容。实际使用中，采用小写字母输入这些内容会更方便些。多数系统采用大小写均可。）图 1-3 举例说明了在酒窖数据库中进行**插入**（insert）、**修改**（change）和**删除**（delete）操作的情况。插入和删除整个文件的例子将在后续的章节中给出。

| BIN# | WINE | PRODUCER | YEAR | BOTTLES | READY |
|---|---|---|---|---|---|
| 2 | Chardonnay | Buena Vista | 2001 | 1 | 2003 |
| 3 | Chardonnay | Geyser Peak | 2001 | 5 | 2003 |
| 6 | Chardonnay | Simi | 2000 | 4 | 2002 |
| 12 | Joh. Riesling | Jekel | 2002 | 1 | 2003 |
| 21 | Fumé Blanc | Ch. St. Jean | 2001 | 4 | 2003 |
| 22 | Fumé Blanc | Robt. Mondavi | 2000 | 2 | 2002 |
| 30 | Gewürztraminer | Ch. St. Jean | 2002 | 3 | 2003 |
| 43 | Cab. Sauvignon | Windsor | 1995 | 12 | 2004 |
| 45 | Cab. Sauvignon | Geyser Peak | 1998 | 12 | 2006 |
| 48 | Cab. Sauvignon | Robt. Mondavi | 1997 | 12 | 2008 |
| 50 | Pinot Noir | Gary Farrell | 2000 | 3 | 2003 |
| 51 | Pinot Noir | Fetzer | 1997 | 3 | 2004 |
| 52 | Pinot Noir | Dehlinger | 1999 | 2 | 2002 |
| 58 | Merlot | Clos du Bois | 1998 | 9 | 2004 |
| 64 | Zinfandel | Cline | 1998 | 9 | 2007 |
| 72 | Zinfandel | Rafanelli | 1999 | 2 | 2007 |

图 1-1　酒窖数据库（文件名为 CELLAR）

*Retrieval:*

```
SELECT WINE, BIN#, PRODUCER
FROM   CELLAR
WHERE  READY = 2004 ;
```

*Result (as shown on, e.g., a display screen):*

| WINE | BIN# | PRODUCER |
|---|---|---|
| Cab. Sauvignon | 43 | Windsor |
| Pinot Noir | 51 | Fetzer |
| Merlot | 58 | Clos du Bois |

图 1-2　检索举例

```
Inserting new data:
INSERT
INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 53, 'Pinot Noir', 'Saintsbury', 2001, 6, 2005 ) ;

Deleting existing data:
DELETE
FROM   CELLAR
WHERE  BIN# = 2 ;

Changing existing data:
UPDATE CELLAR
SET    BOTTLES = 4
WHERE  BIN# = 3 ;
```

图 1-3　插入/修改/删除举例

从图 1-1 到图 1-3 中可以得出以下几点：

1）首先，在图 1-2 和图 1-3 中列出的操作或“语句”、“命令”，如 SELECT（查找）、INSERT（插入）、UPDATE（更新）和 DELETE（删除）等，都是用 **SQL 语言**表示的。SQL 语言最初来自 IBM 公司，是关系数据库的标准交互语言，而且如今市场上几乎所有数据库产品都支持它。由于它的商业重要性，在第 4 章给出了一个关于 SQL 标准的整体概述，而且在大部分的后续章节都会包含一节“SQL 工具”（SQL Facility），主要描述 SQL 标准中与该章的主题相关的细节。

注意：“SQL”这一名字最初代表“结构化查询语言”，并且发音为“sequel”。现在 SQL 已经成为一种标准，其名字已根本不再有任何正式的字母缩写的含义，其发音更倾向于发“ess-cue-ell”。本书将采用后一种发音。

2）注意 SQL 用关键字 UPDATE 来表示“修改”见图 1-3。这可能造成混乱，因为“update”这个词也经常用来概括 INSERT、UPDATE 和 DELETE 三种操作。在书中我们将通过用大小写来区分这两种含义，即小写表示增、删、改，而大写表示 UPDATE 操作符。

注意：你可能已经注意到我们同时使用了单词“操作符”（operator）和“操作”（operation）。严格地说，这两个词是有区别的（操作是发生在操作符被调用的时候），但在非正式的场合，这两个词可以混用。

3）在 SQL 中，例如图 1-1 中的 CELLAR 这样的计算机化文件被称为“**表**”（原因显而易见）。表中的**行**被看作是文件中的记录，表中的**列**被看作是这些记录的字段。在本书中，通常在谈及数据库系统时，我们使用“文件”、“记录”和“字段”这些术语（主要限于前两章）；而在谈及 SQL 系统时，我们将使用“表”、“行”和“列”的叫法。（从第 3 章开始往后，我们进入正式的讨论，将会采用一些更为正式的术语：“关系”、“元组”和“属性”，而不是“表”、“行”和“列”。）

4）对于 CELLAR 表，为了简化起见，在例子中我们默认假设列 WINE（酒名）和 PRODUCER（生产商）为字符串数据，而其他的列为整数数据。然而，总的来说，列里可以包含任意复杂的数据类型。例如，我们可以这样扩展 CELLAR 表，添加：

- 标签（酒瓶标签的照片）
- 评论（一些酒杂志上的文本评论）
- 地图（展示酒出产地的地图）
- 音符（包含我们自己品酒的音频记录）

以及其他很多内容。为了简便，在书中我们采用的大多数是一些简单的数据类型，但是不能忽视，很多复杂的情形也是可以的。在后续的章节我们会详细考虑关于列的数据类型的问题（尤其是第 5、6 章和第 26、27 章）。

5）BIN#列为 CELLAR 表的**主码**（意思是表中没有两行包括同样的 BIN#值），像在图 1-1 中一样，我们经常用双下划线来标识主码列。

最后提及一点：本章以及下一章的内容是基础性的，它对全面正确认识当今数据库系统的特征及功能至关重要。但是，不能否认这两章内容有些抽象和枯燥，而且涉及大量新的概念与术语。在

本书以后的部分（尤其是第3、4章）读者会发现其内容不再那么抽象，相应地也变得容易理解。因此可以先略读前两章，然后在以后遇到直接相关的问题时再仔细重读这些内容。

## 1.2　数据库系统的构成

数据库系统是指一个用计算机存储记录的系统，即它是一个计算机系统，该系统的目标是存储信息并支持用户检索和更新所需要的信息。这里所讨论的信息可以是个人或企业所关心的任何信息，换句话说，它是指任何对个人或组织经营企业的一般处理过程有帮助的数据。

注意：在本书中，术语“数据”和“信息”是同义词。有些作者更喜欢区分两者的含义，即用“数据”表示在数据库中实际存储的内容，而用“信息”表示一些用户对这些数据的理解。这一差异显然很重要，但不适合使用在两个本来同义的术语间生造出来的差异加以区别，我们更倾向于在适当的地方采用明确表述的方式。

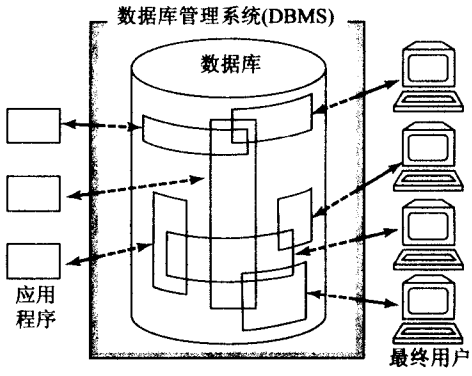



图1-4　数据库系统简图

图1-4是一个数据库系统的简图，它显示了数据库系统包括的4个主要部分：**数据**、**硬件**、**软件**和**用户**。下面将简略地介绍这4个部分。当然，以后会对每一部分进行更详细的讨论（除了硬件部分，硬件的细节内容已经超出了本书的范围）。

### 1. 数据

数据库系统可用于小至手提或个人计算机、大到大型机或机群的各种计算机。显然，任一系统所能提供的功能，某种程度上要取决于运行它的机器的大小与能力。尤其是大型机上的系统（“大型系统”）趋向于多用户，而小型机上的系统（“小型系统”）趋向于单用户。**单用户系统**在任何给定时候最多只有一个用户访问数据库系统，而**多用户系统**可以同时有多个用户访问数据库系统。如图1-4所示，为了不失一般性，本书中假定采用后一种系统，但实际上这一区别绝大多数用户是不关心的，因为通常多用户系统的主要目的就是让每个用户感觉他或她像是在单用户系统上操作。多用户系统的问题主要是系统内部的问题，而不是那些用户可见的问题（见本书第四部分，尤其是第16章）。

注意：为简单起见，不妨假定系统的全部数据都存储在一个数据库中，本书中即采用这一假定（因为这样本质上并不影响任何其他的讨论）。然而实际上，即使在小系统中，也经常需要将数据分散存放在不同数据库中。我们将在第2章及其他章中介绍这其间的原因。

通常，数据库中的数据（至少在大型系统中）既是集成的，又是共享的。正如将在1.4节所看到的，数据集成与数据共享代表着在大型环境中数据库系统的主要优点。其实数据集成在小型环境中也具有重要意义。当然，在小型环境中也还有很多其他的优点（以后将讨论）。但现在首先解释集成与共享的含义。

- **集成**指的是数据库可以被当作几个不同文件的合并，数据库至少可以部分地消除文件之间的冗余。例如，一个指定的数据库可以包含一个EMPLOYEE（雇员）文件和一个ENROLLMENT（注册）文件。EMPLOYEE文件中给出雇员名、地址、部门和工资，等等。ENROLLMENT文件表示在接受培训的雇员的注册信息（参见图1-5）。现在假定为了进行培训课程管理，需要清楚每个在训学员所在的部门。那么，显然没有必要在ENROLLMENT文件中重复有关的部门信息，因为需要的话，可以从EMPLOYEE文件中找到。
- **共享**指的是数据库中的每项数据可以被不同的用户共享。换句话说，每一个用户都可以因不同的目的而访问相同的数据。如上所述，不同的用户甚至可以同时访问同一数据（“并发访问”）。之所以有并发共享或其他方式的共享，部分是因为数据库是集成的。比如，在图1-5的例子中的部门信息共享就很有代表性，EMPLOYEE文件中的部门信息既可以

被人事部门的用户共享，也可以被教育部门的用户所共享，即这两个部门的用户因为不同的目的而使用这一共享信息。（如果数据库是非共享的，则它通常被称作个人数据库或专用数据库。）

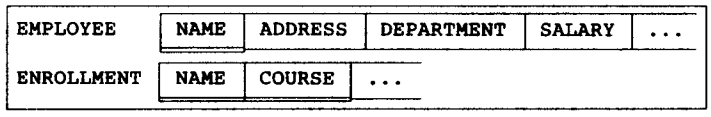



图 1-5　EMPLOYEE 和 ENROLLMENT 文件

数据库集成和共享带来的另外一个结果，是任一个用户都只关心整个数据库中的一小部分，而且不同用户所使用的数据会以各种方式重叠。换句话说，对于一个指定的数据库，不同的用户会以许多不同的方式来观察。实际上，即使两个用户共享数据库中的同一块数据，在细节层次上，他们看待数据的角度也会有所不同。这一点将在 1.5 节和后续章节（特别是在第 10 章）中进行详细讨论。

我们将在 1.3 节中详细阐述数据库系统中数据部分的本质特征。

**2. 硬件**

系统的硬件部分包括：

- 二级存储设备，以及相关的 I/O 设备（磁盘驱动器等）、设备控制器、I/O 通道等。二级存储设备（通常为磁盘）用来存放数据。
- 硬件处理器和相应的主存。硬件处理器和相应的主存用于支持数据库系统软件的执行（见下一小节）。

本书将不对系统硬件部分作过多介绍。这主要基于以下几点原因：一是硬件部分内容庞杂，自成体系；二是硬件方面的问题不是数据库系统独有的；三是硬件方面的有关知识在其他资料上已经详细地阐述了。

**3. 软件**

在物理数据库（例如物理存储的数据）和数据库系统的用户之间有一层，即软件层，它通常被称作**数据库管理器**或**数据库服务器**，而其最通用的称法为**数据库管理系统**（DBMS）。所有访问数据库的请求都是由 DBMS 来处理的。DBMS 提供了许多对数据进行操作的实用程序，如 1.1 节中的增加和删除文件（或表），也可以在这些文件中检索或更新数据。DBMS 提供的基本功能为数据库用户屏蔽掉了物理层的细节（就像程序设计语言系统为应用程序员屏蔽掉物理层细节一样）。换句话说，DBMS 为用户提供了一种在硬件层之上观察数据库的高级别方式，并且支持用户以这种高级别方式表达操作请求（如在 1.1 节简要讨论的 SQL 操作）。本书将会详细讨论 DBMS 这方面的功能以及其他功能。

还有两点进一步的说明：

- 在整个系统中，DBMS 是最重要的软件部分，但不是唯一的。其他软件包括实用程序、应用开发工具、设计辅助、报表书写器和（非常重要的）事务管理器或事务处理监控器（TP monitor）。见第 2、3 章和第 15、16 章有关这方面的更进一步的讨论。
- DBMS 这一术语也通常用于指某个特定厂商的特定产品。例如，IBM 的基于 OS/390 的“DB2 Universal Database”产品。术语 DBMS 实例（instance）有时用于指这些产品运行于某一特定的计算机设备上的特定副本。正如大家将会看到的，有时有必要区别这两个概念。

*注意*：有时企业的人员用数据库这一术语而实际上是指 DBMS。以下是一个典型的例子：“*X* 厂商的数据库与 *Y* 厂商的数据库的性能比是二比一。”这种用法是不恰当的，但却非常普遍。（当然，问题是如果我们称数据库管理系统为数据库，就会把它与真正的数据库混淆。）

**4. 用户**

我们考虑三类主要用户（相互间可能会有些重叠）：

- 首先是**应用程序员**。应用程序员负责编写数据库应用程序。他们使用某些程序设计语言，

如 COBOL、PL/I、C++、Java 或某种高级的第四代语言（见第 2 章），来编写应用程序。这些程序通过向 DBMS 发出 SQL 语句请求来访问数据库。这些程序通常可以是批处理应用程序或**联机**应用程序，目的是允许最终用户通过联机工作站或终端访问数据库。大多数当今的应用程序都是联机方式的。

- 第二类用户是**最终用户**。他们从联机工作站或终端与系统交互。最终用户可以通过联机应用程序访问数据库，或者使用数据库系统软件提供的接口。当然，这些由厂商提供的接口也可以被联机应用程序的方式所支持，但是这些应用程序是**内建的**（build in），而不是用户编写的。大多数数据库系统至少包括一种内建的应用程序，即**查询语言处理器**，通过它用户可以交互地发出数据库请求（就是人们所熟知的语句或指令）给 DBMS，诸如 SELECT 和 INSERT。SQL 是一个典型的数据库查询语言的实例。（注意："查询语言"一词虽很常用，但其实它有用词不当之嫌。因为自然语言的动词"查询"只表示查找数据，而查询语言通常（并不总是）还提供更新和其他操作。）

大多数系统也提供其他内建的界面。通过这种界面，用户根本不发出像 SELECT 或 INSERT 这样明确的数据库请求，而是代之以选择菜单中的一项或填充表格中的一栏。这样的**菜单驱动**或**表格驱动**界面对于并未受过正式 IT 训练的人来说更容易（IT 即信息技术；IS 是信息系统的缩写词）。然而，使用**命令驱动界面**（如查询语言）需要一定量的专业训练（很显然不像用 COBOL 语言编写一个应用程序那样麻烦）。同时，命令驱动界面有时会比菜单驱动或表格驱动界面更灵活。在这类界面中，其查询语言中包含了某些特性，而它们可能不被其他界面所支持。

- 第三类用户没有在图 1-4 中显示。他们是**数据库管理员**或简称为 DBA。有关数据库管理员职能的讨论，请参见 1.4 节和第 2 章的 2.7 节。

以上完成了对数据库系统主要方面的初步描述，在以后的章节中我们还会更进一步讨论这方面的内容。

## 1.3　数据库的内涵

**1. 持久数据**

数据库中的数据通常被认为是持久存储的（尽管事实上保存并非很久!）。对于持久性，直观上是为了把数据库中的数据与其他的输入数据、输出数据、控制语句、工作队列、软件控制块、SQL 语句和中间结果等临时数据相区分，而且通常任何数据本质上都是暂时的。更精确地说，我们说数据库中的数据是持久的，是因为一旦数据进入数据库被 DBMS 接受，就只有向 DBMS 提出某些明确的请求时，才能从数据库中删除数据。这有别于某些程序运行结束时产生的副产品。这种持久性的概念使得我们可以给数据库下一个更精确的定义：

**数据库**是一个持久数据的集合，这些数据用于某企业的应用系统中。

这里"企业"一词只是一个方便的通称，可指代任何独立的商业组织、科学组织、技术组织或其他组织。企业可能仅是个人（一个小的个人数据库），或者是一个公司或类似的大型实体（有一个大型的共享数据库），或是任何界于两者之间的单位。如下面的例子：

1）制造公司
2）银行
3）医院
4）大学
5）政府部门

任何企业必须保存与自身运作有关的大量数据，这些就是上面提到的持久数据。上述企业主要（分别）包含下列持久数据：

1）生产数据
2）会计数据
3）病人数据
4）学生数据

5）计划数据

注意：本书的前 6 版使用“操作型数据”代替“持久性数据”。早期的术语反映了数据库的重点在**操作型**应用或**生产型**应用。例如，例行的、经常重复的应用，这些应用程序要重复地执行以支持企业的日常运行（例如，银行系统中支持存款和取款操作的应用程序）。术语**联机事务处理**（OLTP）用于指这种环境。但是，渐渐地，数据库也面向其他类型应用，例如，**决策支持**应用，这样，操作型数据的说法就不再合适了。事实上，目前的企业经常有两个独立的数据库：一个保存操作型数据；另一个称为**数据仓库**，保存决策支持数据。数据仓库通常包括概要信息（如总计、均值），而这些概要信息来自于操作型数据库，也就是说，以一定时间间隔，一天一次或一周一次从操作型数据库中抽取这些信息。对决策支持数据库及其应用的进一步讨论见第 22 章。

**2. 实体与联系**

现在我们稍微仔细地分析一个制造企业（名为“KnowWare 公司”）的例子。这种企业主要关心以下信息：目前已有的工程；在这些工程中所使用的零件；提供零件的供应商；储存零件的仓库；在这些工程中工作的雇员，等等。工程、零件、供应商等构成了基本的**实体**，KnowWare 公司需要记录这些信息（在数据库中，实体一词通常指数据库中表示的任何可区分的事物）。如图 1-6 所示。

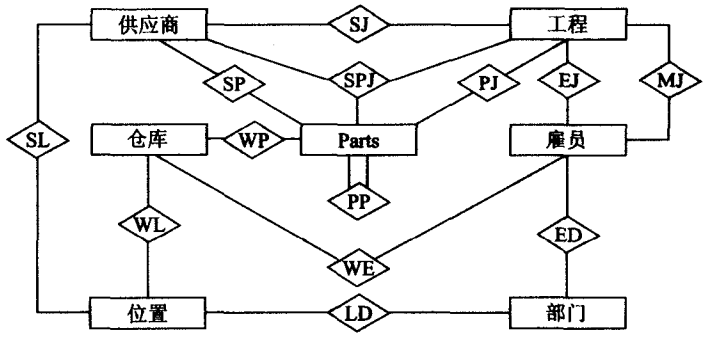



图 1-6　KnowWare 公司的实体/联系（E/R）图

除了基本实体本身（例子中的供应商、零件等），还有连接这些基本实体的**联系**。这些联系在图 1-6 中通过菱形和连线表示。例如，在供应商和零件之间有一个联系“SP”：每个供应商供应某些零件；反过来，每个零件都由某些供应商提供（更准确地说，每个供应商供应某些种类的零件，每种零件都由某些供应商提供）。类似地，零件在工程中被使用；反过来，工程要使用零件（联系 PJ）；零件被存放在仓库中，并且仓库中存放着零件（联系 WP），如此等等。注意这些联系都是双向的，也就是说，每个方向都可调换。例如，在供应商和零件之间的联系 SP 可用来回答下面两个问题：

- 指定一个供应商，获得由该供应商提供的零件。
- 指定一种零件，获得供应这种零件的供应商。

重要的是这一联系（当然，以及图中表示的其他所有的联系）和基本实体一样作为数据的一部分，它们和基本实体一样都要在数据库中表示[⊖]。

注意：上面已经说明图 1-6 是一个**实体/联系图**（E/R 图）的例子。第 14 章将详细讨论这种图。

图 1-6 还说明了以下的几点重要内容：

1）尽管该图中的大部分联系是涉及两个实体类型的，也就是二元联系，但是这并不意味着所有的联系必须是二元的。例子中联系（“SPJ”）就涉及三个实体类型（供应商、零件和工

⊖ 在关系数据库中（见 1.6 节），基本的实体和连接它们之间的联系都是用关系的方式表示的。换言之，它们都是用图 1-1 中展示的表的形式表示的。需要特别注意的是，这里所说的“联系”和关系数据库中提及的“关系”并不是同一个意思。

程），它是一个三元联系。这个三元联系表示某些供应商为某些工程供应某些零件。注意，这个三元联系（“供应商给工程提供零件”）和下面三个二元联系的联合是不同的，即“供应商供应零件”、“零件用于工程”和“工程由供应商供应”。例如，语句[⊖]

a. 史密斯（Smith）给曼哈顿（Manhattan）工程提供活动扳手。

所提供的信息要多于下面三个语句：

b. 史密斯供应活动扳手。

c. 活动扳手用于曼哈顿工程。

d. 曼哈顿工程由史密斯供应。

知道了b、c和d，我们不能有效地推断出a。更精确地说，如果已知b、c和d，我们就可以推断出史密斯给某项工程（如工程Jz）供应活动扳手，某个供应商（如供应商Sx）供应活动扳手给曼哈顿工程，史密斯给曼哈顿工程提供某种零件（如零件Py）。但是我们不能推出Sx是史密斯，或者Py是活动扳手，或者Jz是曼哈顿工程。这样的错误推论有时被称作**连接陷阱**。

2）图中还显示了只含一个实体（零件）类型的联系（PP）。这一联系指出某种零件把其他零件作为直接的组成部件（也称作**材料账单联系**）。例如，螺丝钉是铰链的一个组成部分，而铰链也被看作零件，或许也是某个更高层零件（如盖子）的组成部分。注意，这个联系仍然是二元的，只是它所连接的两个实体恰好是同一类型或同一个实体。

3）通常，一组指定的实体类型可能由一些不同的联系连接起来。在图1-6的例子中，涉及工程和雇员的不同的联系有两个：一个（EJ）表示雇员被分配到工程中这一事实；另一个（MJ）表示雇员管理工程这一事实。

现在来分析把联系本身看作一个实体的情况。若从实体的定义（任何具有有用信息的对象）来看，联系也符合这一定义。例如，“零件P4储存在仓库W8”是一个实体，我们可以记录有关信息（例如，相应的数量）。而且，对于实体和联系不加以不必要的区分会有很多好处（其讨论超出本章的范围）。因此，本书通常只把联系当作一种特殊的实体。

**3. 属性**

如前所述，实体是具有有用信息的对象。实体（包括联系）具有**属性**，这些属性记录实体的相应信息。例如，供应商有地址；零件有重量；工程有优先级；任务有开工期；如此等等。这样的属性必须在数据库中表示。例如，SQL数据库可能包含表S，表示供应商，表中有一列CITY表示供应商的地址。

通常，属性可以简单，也可以很复杂。例如，“供应商地址”这一属性可以设定得非常简单，仅表示城市名，在数据库中只用一个简单的字符串表示即可。相反，数据仓库中可能会包含一个属性“建筑平面图”，这个属性就会很复杂，可能包括整个建筑图和相应的描述文字。换句话说，如在1.1节中所述，SQL表列中可以包含任意复杂的数据类型，并且第5、6章和第26、27章会再度介绍这一主题。在此之前，仍假定属性都是简单类型，而且可以用简单的数据类型来表示。这些简单的数据类型包括数字、字符串、日期和时间等。

**4. 数据和数据模型**

对数据和数据库存在另外一种重要的认识方式。“数据”一词来自拉丁文“给，供给”。这样，数据就是指定的事实，从中可以推出另外的事实。（从指定的事实中推出另外的事实恰恰是DBMS响应用户要求时所做的处理。）一个“指定的事实”要符合逻辑学家所说的**真命题**；例如，陈述句“供应商S1住在伦敦”就是这样的**真命题**。（在逻辑学中，命题只可能是正确的或者错误的，不会含混不清。例如，“William Shakespeare著有《傲慢与偏见》”就是一个错误的命题。）因此，数据库实际上就是一些**真命题的集合**。

现在，SQL产品已经在市场中占有主导地位。其原因之一是，SQL产品是基于一种称为**关**

---

⊖ 术语“语句”在数据库中有两种截然不同的意思：在这里它表示“对事实的论断”，或者是逻辑学家所说的“命题”（见稍后的“数据和数据模型”小节）；它也可以用作“命令”的同义词，例如在短语“SQL语句”中所表示的意思。

系数据模型的形式化理论的，这一理论直接支持对数据和数据库的解释，特别是在关系模型中：

1）数据通过表中的行来表示[⊖]，并且这些行可以被直接解释为真命题。例如，在图 1-1 中，BIN# 72 这一行可以被解释为如下真命题：

“Bin 号码为 72 的行包含两瓶 1999 年的 Rafanelli Zinfandel 酒，这些酒到 2007 年才可饮用。”

2）所提供的操作符可以针对表的行进行，这些操作符直接支持从指定的命题推出另外的真命题的处理。举个简单的例子，关系投影操作符（见 1.6 节）可以从前面列出的真命题推出下列的真命题：

“一些 Zinfandel 酒到 2007 年才可饮用。”

（更精确地说，“在某个 bin 中的某些 Zinfandel 酒，是由某厂商于某年生产的，到 2007 年才可饮用。”）

关系模型不是唯一的数据模型，还存在其他模型（见 1.6 节）。这类模型与关系模型截然不同，一般不具有规范的形式逻辑基础。通常的数据模型是什么？遵照参考文献［1.1］，我们可以给出如下定义：

- **数据模型**是对对象、操作等的一个抽象的、自包含的逻辑定义，这些定义合起来构成了一个面对用户的抽象机。其中对象可以用来建模数据结构。操作符用于建模一些行为。

这样我们可以得到一个很有用（而且很重要）的区分模型及其实现（implementation）的定义：

- 对指定的数据模型的**实现**是指在真实机器上的物理实现，一个真实机器是抽象机的组成部分，它们一起构成模型。

简而言之，模型是用户必须知道的；实现是用户不需要知道的。

正如前文所述，模型与实现之间的区别就如同典型的逻辑与物理之间的区分一样。但是，今天的许多数据库系统（甚至一些关系系统）并没有分清这些区别。确实，目前尚缺乏对这一问题的认识。由此造成了在数据库原理（即数据库系统应当怎样）和数据库实践（即数据库系统实际如何）之间经常存在鸿沟。本书中主要介绍原理，因此有必要提醒读者当你使用商业产品时，会经常遇到一些与原理有差异的情况。

在结束本节之际，还应该提到的是，实际上数据模型这个词在字面上就有两个十分不同的含义。一个是上面提到的；另一个是指作为某特定企业（例如本节前面提到的 KnowWare 公司）的持久数据模型。两者之间的区别如下：

- 第一种含义下的数据模型就像编程语言，虽然有点抽象，可用来解决各种特定的问题，但是，它本身与特定的问题毫无联系。
- 第二种含义下的数据模型就像一个用上述语言编写的特定程序。换句话说，第二种含义下的数据模型利用第一种含义下的模型所提供的一些功能，用于解决一些特定的问题。因此它可以看作是第一种含义下的数据模型的特定应用。

本书中，在没有明确说明的情况下，数据模型一词都是指第一种含义。

## 1.4　使用数据库的优点

为什么要用数据库呢？它有什么优点？在某种程度上，这些问题的答案依赖于所谈的系统是单用户的还是多用户的。更准确地说，多用户系统有更多的优点。先看单用户系统：

仍以酒窖的藏酒量为例（图 1-1），该图例展示了单用户系统的情况。该数据库太小也太简单，不足以展示数据库的优点。但是设想一个大酒店有类似的数据库，其库存可能有成千上万瓶酒，而且库存变化十分频繁；或者设想一家卖酒的商店，也有相当大的库存，并且库存周转率很高。在这种情况下，数据库系统与传统的、基于纸的记录保存方式相比，其优点可能就显而易见。以下列举了一些优点：

- 简洁：不需要大量成卷的文件。
- 快捷：机器对数据的变动和更新比手工快得多。尤其是，对于实时查询（例如：Zinfandel

⊖ 更精确地说，用关系中的元组来表示（见第 3 章）。

和 Pinot Noir 这两种葡萄酒，哪种我们储备得比较多?）可以快速地给出答案。

- 省力：不再手工保存大量的文件，机械的任务可以由机器更好地完成。
- 方便：可以随时得到准确、最新的信息。
- 安全：数据不容易丢失，不会被非法访问。

上述优点在多用户环境中能更好地体现出来。因为通常多用户环境下的数据库要比单用户的更大也更复杂。然而，多用户环境下还有一个很突出的优点，即，数据库系统保证了企业对数据的集中控制（对这一点，大家应意识到，它是最有价值的优点之一）。这种情况与没有数据库系统的企业相比，形成了鲜明的对照。在典型的没有数据库系统的企业中，每个应用拥有各自的文件——通常是各自的磁带和磁盘——以致于难以用任何系统的方法来控制这些非常分散的数据。

**1. 数据管理和数据库管理**

前面简明扼要地描述了集中控制的概念。这一概念隐含着企业中要有某个可确认的人对数据有着核心的权力。这就是 1.2 节简要提到过的**数据管理员**（简称为 DA）。假如数据是一个企业最有价值的资产，就一定要有人能理解这些数据以及企业对这些数据的需求，并且此人要处于企业的高级管理层。数据管理员就是指这样的人。因此，数据管理员的工作就是首先决定什么数据存储在数据库中，一旦存储了这些数据，就要建立维护和处理这些数据的机制。例如，在什么情况下谁可以执行什么操作就是一种机制——换句话说，它是数据安全机制（见下一小节）。

注意，数据管理员是管理者而不是技术人员（尽管他或她当然要在技术上对数据库系统的性能有所了解）。负责执行数据管理员的决策的技术人员就是**数据库管理员**（简称为 DBA）。与数据管理员不同，DBA 是信息技术（IT）方面的专业人员。数据库管理员的工作是创建实际的数据库，以及执行实施各种决策所需的技术控制。数据库管理员也负责确保系统正确执行操作，并且提供各种其他技术服务。数据库管理员通常包括一些系统程序员和其他技术助理（也就是说，数据库管理员的功能实际上由一组人员来承担，而不是一个人）；但是为简化起见，通常假定数据库管理员只是一个个体。我们将在第 2 章具体讨论数据库管理员的职能。

**2. 数据库方法的优点**

本小节给出一些特定的优点，这些优点来自前面所述的集中控制。

- 数据共享。我们在 1.2 节讨论了这一点，但是为了完整，我们在这里再次提及。共享不仅指现有的应用程序可以共享数据库的数据，而且新的应用程序也能对这些数据进行操作。换句话说，不向数据库中添加任何新数据也可能满足新应用程序的数据要求。
- 减少冗余。在非数据库系统中，每个应用程序都有自己的专用文件。这种情况经常导致在存储数据上有相当大的冗余，结果浪费了存储空间。例如，一个有关人事的应用程序和一个有关教育的应用程序可能同时拥有包含职员和部门信息的文件。但是，如 1.2 节所示，这两个文件可以集成起来消除冗余，只要数据管理员意识到这两个应用程序的数据要求——也就是说，企业应有必要的全局控制。

注意：我们并不是指所有的冗余数据都需要消除。某些时候，维护同一数据的多个不同副本是很好的商业和技术考虑。尽管如此，数据冗余还是需要认真控制的。因为 DBMS 需要考虑，如果存在数据冗余，那么需要进行传播更新（见下一点）。

- 避免不一致（某种程度上）。这是前一点必然的结果。假定一种实际情况——雇员 E3 在部门 D8 工作——数据库中有两个不同的条目。还假定 DBMS 也没有意识到冗余的存在（也就是对冗余失控）。则必然会有两个记录不一致的情况：即，当其中一个更新时，另一个不变。这种情况称为数据库不一致。显然，处于不一致状态的数据库可能会给用户提供错误的或矛盾的信息。

当然，如果指定事实是由一条记录表示的（也就是如果排除了冗余），那么这样的不一致就不会发生。另一种选择是，冗余没有排除但是受到控制（被 DBMS 得知），那么数据库管理系统就可以保证对用户来说数据库总是一致的，DBMS 确保两条记录中的任何一条改变会自动地应用到另一条。这一过程即为**传播更新**。

- 提供事务支持。**事务**是一个逻辑工作单元，它包括一些数据库操作（特别是，一些更新

操作）。常见的例子如从账户 *A* 到账户 *B* 转移一定的现金。显然，这里要求两个更新操作：一个是从账户 *A* 提出现金；另一个是把现金存入账户 *B*。如果用户已经说明两个更新是同一事务的一部分，那么系统要确保两个操作要么都做，要么都不做——即使在系统执行过程中出现故障（比如因为断电）也应如此。

注意：刚刚举例说明的事务的原子性并非事务的唯一优点，但是与其他一些优点不同，它甚至可以应用到单用户的情况。事务支持的各种优点和怎样实现的全面描述见第 15 章和第16 章。

- 保持完整性。完整性的问题是确保数据库中的数据是正确的。同样事实的两条记录的不一致，就是缺少完整性的例子（见前面的讨论）；当然，只要在存储的数据中有冗余，就会引起这样的问题。即使没有冗余，数据库也可能包含错误的信息。例如，数据库可能显示雇员一周工作了 400 小时而不是 40 小时，或者属于一个不存在的部门。数据库的集中控制可以有效地避免此类问题，做法是通过支持数据管理员定义一些**完整性约束**，由 DBA 加以实施，完整性约束在任何操作执行时都得到有效的检验。

值得指出的是，数据完整性在数据库中要比在各自独立的文件系统中重要得多，因为数据库中的数据是共享的。要是没有正确的控制，有可能一个用户错误地更新数据库而生成的错误数据，会殃及其他无辜的用户。目前，数据库厂商对数据库的完整性约束的支持还相当不够（尽管最近这一方面的情况有所改善）。这一事实很不幸，因为数据库完整性比你想象的更为基本和重要（见第 9 章）。

- 增强安全性。数据库管理员（在数据管理员的正确指导下）可以确保访问数据库的唯一方式是通过正确的通道，因此可以定义**安全性约束或规则**。当试图访问敏感数据时，要检查这些安全性约束或规则。对于数据库的每条信息的不同类型的访问（检索、插入或删除等），可建立不同的约束。注意，若没有这样的约束，数据的安全性可能比传统的文件系统更糟糕，也就是说，某种意义上数据库系统的集中性要求相称的、好的安全系统。
- 平衡相互冲突的请求。除了单个用户的需求，数据库管理员还应了解企业的全局需要，并在数据管理员的指示下建立系统的结构以提供对企业最佳的全局服务。例如，所选择的数据的物理表示应尽可能使重要的应用能以最快的方式访问数据（可能会以降低其他某些应用的访问速度为代价）。
- 加强标准化。数据库管理员（在数据管理员的指示下）对数据库集中控制，可以确保所有表示数据的可用标准都可以顾及到。可用标准可包括下面的任意一种或全部：部门标准、安装标准、社团标准、工业标准、国家标准和国际标准。标准化的数据表示可以有效地支持数据交换或者两个系统间的数据移动（随着分布式系统的出现，这一点就越来越重要——见第 2、21 和 27 章）。同时，数据命名和文档标准也有效地支持了数据共享和易理解性。

以上列出的大多数优点都是比较显而易见的。但是，有一点则不然，这就是对数据的独立性的支持。（严格地说，数据独立性是数据库系统的客观目标，而不仅仅是一个必要的优点。）数据独立性的概念十分重要，以下专辟一节来深入讨论。

## 1.5　数据独立性

要理解数据独立性的含义，最好的方法是先搞清什么是非数据独立的。在旧的系统中——即关系系统之前的和数据库系统之前的系统——实现的应用程序常常是数据依赖的。这也就意味着，在二级存储中，数据的物理表示方式和有关的存取技术都是应用设计中要考虑的，而且，有关物理表示的知识和访问技术直接体现在应用程序的代码中。例如，假定有一个应用程序使用了图 1-5 中的文件 EMPLOYEE，还假定文件在雇员姓名字段进行索引（见在线的附录 D）。在旧的系统中，该应用程序肯定知道存在索引，也知道记录顺序是根据索引定的，应用程序的内部结构是基于这些知识而设计的。特别地，各种数据访问的准确形式和应用程序的异常检验程序都很大程度上依赖于数据管理软件提供给应用程序的接口细节。

我们称这个例子中的应用程序是**数据依赖**的，因为一旦改变数据的物理表示，就会对应用程

序产生非常强的影响。例如，用哈希算法来对例子重建索引后，对应用程序不作大的修改是不可能的。而且，这种情况下应用程序修改的部分恰恰是与数据管理软件密切联系的部分。这其中的困难与应用程序最初所要解决的问题毫不相关，而是由数据管理接口的特点所引起的。

与数据的依赖性相对，数据独立性包括两个方面：物理独立性和逻辑独立性［1.3，1.4］。首先讨论数据的物理独立性。在未进一步说明之前，“数据独立性”应该理解为数据的物理独立性。注意，应该说“数据独立性”这种说法没有抓住问题的本质；但是，由于传统上一直这么用，本书中仍采用该术语。

在数据库系统中，应尽可能避免应用程序依赖于数据的情况。这至少有以下两条原因：

1）不同的应用程序会从不同角度来看相同的数据。例如，假定在企业建立统一的数据库之前有两个应用程序 *A* 和 *B*，每一个都拥有包括客户余额的专有文件。假定 *A* 是以十进制存储的，而 *B* 是以二进制存储的。这时有可能要消除冗余，并把两个文件统一起来。条件是 DBMS 可以而且能够执行以下必要的转换，即存储格式（可能是十进制、二进制或者其他的）和每个应用程序所采用的格式之间的转换。例如，如果决定以十进制存储数据，每次对 *B* 的访问都要转换成二进制。

这是个非常普通的例子，在数据库系统中，应用程序所看到的数据和物理存储的数据之间可能是不同类型的。本节后面部分还会考虑其他许多可能的不同情况。

2）DBA（或是 DBMS）必须有权改变物理表示和访问技术以适应变化的需要，而不必改变现有的应用程序。例如，新类型的数据可能加入到数据库中；有可能采纳新的标准；应用程序的优先级（因此相关的性能需求）可能改变；系统要添加新的存储设备，等等。如果应用程序是数据依赖的，这些改变会要求程序做相应的改变，这种维护的代价无异于创建一个新的应用程序。类似的情况甚至在今天都并不少见，如典型的 Y2K 问题，这对充分利用稀缺而宝贵的资源是极其不利的。

总之，提出数据独立性主要是考虑到数据库系统的客观要求。数据独立性可以定义成**应用程序不会因物理表示和访问技术的改变而改变**。当然，这意味着应用程序不应依赖于任何特定的物理表示和访问技术。在第 2 章中，描述了支持以上基本要求的数据库系统的结构。在此之前，我们还是先讨论一下发生改变的具体情况，即 DBA 通常都有哪些改变上的要求，进而讨论怎样使应用程序尽量免受这方面的影响。

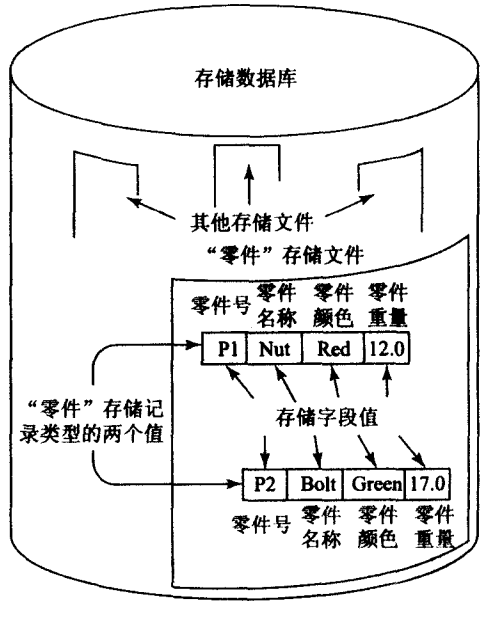



图 1-7　存储字段、存储记录和存储文件

首先给出三个术语：存储字段、存储记录和存储文件（见图 1-7）。

- **存储字段**。简言之，存储字段就是存储数据的最小单位。数据库对每一种**类型**的存储字段都包含许多具体**值**（或**实例**）。例如，包含不同类型零件信息的数据库可能包括称为“零件编号”的存储字段类型，那么对每种零件（如螺丝钉、铰链、盖子等），即有一个该存储字段的具体值。

注意：实际中，通常不再明确指出是类型还是值，而是依据上下文来确定其含义。尽管这有可能带来一些混淆，但实际中是方便的，本书中仍不时地采用这种方式。（以上说明也同时适用于存储记录——见下一段要讨论的内容。）

- **存储记录**是相关的存储字段的集合。我们仍区分类型与值。一条存储记录的**值**（或实例）由一组相关的存储字段的值组成。例如，在零件数据库中的一条存储记录的值可能由下列

存储字段的值组成。这些字段包括：零件号、零件名称、零件颜色和零件重量。而数据库是由“零件”存储记录**类型**的许多值（每种零件一个值）组成的。

- **存储文件**是由现存的一种类型的存储记录的值组成的。注意，为简单起见，假定任一存储文件只包含一种类型的存储记录。这种简化并不影响后面的论述。

现在，在非数据库系统中，常常是应用程序所处理的任一逻辑记录都与相应的存储记录相同。然而，我们已经看到，在数据库系统中这是不必要的，因为 DBA 可能需要对存储的数据表示（即存储字段、存储记录和存储文件）进行改变，可是从应用程序的角度来看数据还是不变的。例如，在 EMPLOYEE 文件中的 SALARY 字段可能为了节省存储空间而存成二进制格式，而指定的 COBOL 应用程序可能把它作为字符串来处理。随后 DBA 可能会由于某种原因而改变数据的存储形式，如从二进制改为十进制，但对 COBOL 应用程序而言，仍以字符串的形式对待。

如前所述，像这种在一次访问中出现某字段的数据类型变化的情况相对比较少。但是，一般情况下，应用程序中的数据和实际所存储的数据之间的差异会相当大。为明确说明这一点，我们给出下列可能要改变的各种存储表示。在每种情况下，都应该考虑 DBMS 应怎么做才能使应用程序保持不变（即是否可以达到数据独立性）。

- *数字数据的表示*。一个数字字段可能存成内部算术形式，或作为一个字符串。对每一种方式，DBA 必须选择恰当的数制（例如，二进制或十进制）、范围（固定的或浮点）、方式（实数或复数）以及精度（小数的位数）。为提高执行效率或符合某一新标准或因其他原因，其中任一方面都可能需要改变。
- *字符数据的表示*。一个字符串字段可能使用了几个不同的编码字符集中的一种，如 ASCII、EBCDIC 或 Unicode。
- *数字数据的单位*。数字字段的单位会改变——例如，在实施公制度量的处理中从英寸转成厘米。
- *数据编码*。有时以编码的形式来表示物理存储的数据是非常好的。例如，“零件颜色”字段，在应用程序中看作字符串（“红”、“蓝”或“绿”），存储时可以存成单个十进制数字，可根据 1 =“红”，2 =“蓝”，如此等等这样的编码模式来解释。
- *数据具体化*。实际中由应用程序所看到的逻辑字段经常与特定的存储字段相联系（尽管它们会在数据类型、编码等方面都不相同）。在这种情况下，数据具体化的处理（也就是从相应的存储字段的值构建逻辑字段的值，并提供给应用程序）可以说是直截了当的。但是，有时一个逻辑字段可能没有对应的存储值，它的值可以通过计算一些存储字段的值来具体化。例如，逻辑字段“总量”的值可以通过对各个单个存储数量来汇总而得到。在这种情况下，具体化的过程是间接的。
- *存储记录的结构*。两个已有的存储记录可以合成为一个。例如，存储记录 `part no.` `part color` 和 `part no.` `part weight` 可以合成为 `part no.` `part color` `part weight` 的形式。

  当把新的应用程序集成到数据库系统时会经常发生这种改变。这也暗示应用程序的逻辑记录可能由相应存储记录的恰当子集组成——也就是说，存储记录中的某些字段对应用程序来说是不可见的。

  另一种情况是单个的存储记录被分成两个。将前面的例子反过来，存储记录 `part no.` `part color` `part weight` 可能被分裂为 `part no.` `part color` 和 `part no.` `part weight`。

  例如，这种分裂允许较少使用的原始记录部分存储在一个慢速设备上。这意味着给定应用程序的逻辑记录可由来自几个不同存储记录的字段组成——即，可能是某指定的存储记录的超集。
- *存储文件的结构*。一个指定的存储文件可以各种方式实现其存储（见在线的附录 D）。例如，它可以完全存储在单个的存储设备上（单个磁盘），或者存储在几个设备（可能在几个不同的设备类型）上；可能根据一些存储字段的值按一定物理顺序来存储，或无序存储；存储顺序可能按某一种或某几种方式进行，例如，通过一个或多个索引，一个或多个嵌入的指针链，或者两者兼有；通过哈希算法可能访问到也可能访问不到该文件；存储记

录可能物理上被分块，也可能没有；如此等等。但是上述任何考虑都不会以任何方式影响应用程序（当然性能除外）。

以上所列基本概括了可能的存储数据形式的改变。这意味着数据库应该能够增长而并不削弱现存的应用程序的功能；的确，在保证数据库增长的同时而不削弱应用程序的功能，是提出数据独立性的重要原因之一。例如，必须有办法通过增加新的存储字段来扩展现有的存储记录，如对现存的实体类型进一步追加信息（如“单价”字段可能会加入到“零件”的存储记录中）。这样新的字段对原应用程序来说应是不可见的。同时，还可能增加全新的存储记录类型（这样就有新的存储文件），而这不会引起应用程序的改变。这样的记录和文件通常代表新的实体类型（例如，一个“供应商”记录类型可以加到“零件”数据库中）。而且这些增加对应用程序来说也应是不可见的。

至此，大家应清楚把数据模型从其实现中分离出来的原因之一就是数据独立性的要求，就像在 1.3 节末尾所指出的那样。某种程度上，不做这种分离，就得不到数据的独立性。目前，不能正确做到这种分离的情况很严重，尤其是当今的 SQL 系统都做不到这一点，这实在让人沮丧。注意：这并不意味着当今的 SQL 系统根本不支持数据的独立性，只是它们提供的要远远少于关系系统理论上要求达到的[⊖]。换句话说，数据独立性不是绝对的（不同系统可提供不同程度的数据独立性，很少有系统根本不提供）；SQL 系统提供的要比其他的系统多一些，但还不是很好，这在后续章节中将会介绍。

## 1.6　关系系统及其他数据库系统

正如在 1.3 节末尾所提到的，SQL 系统在 DBMS 市场上已经占据了主导地位，其中的重要原因是这种系统是建立在关系数据模型基础上的。在非正式的情况下，SQL 系统也被称作“关系系统”[⊜]。在过去 30 年中，大量主要的数据库研究是基于关系模型的。实际上，不可否认，1969～1970 年间关系模型的建立，在整个数据库领域的历史中无疑是最重要的事件。由于这些原因，加上关系模型有坚实的逻辑和数学基础，因此它成为数据库原理的主要教学内容，本书的重点也主要针对关系系统。

那么究竟什么是关系系统呢？很显然本书在此还不能给出一个完满的解答，但是可以先给出一个大致的定义，之后再给出更详尽的答案。简言之，关系系统是指：

1）数据以表（而且只有表）的形式呈现给用户。

2）提供给用户的操作（如检索）以表为操作对象，即操作是从旧表中生成新的表。例如，“选择”（restrict）操作是提取某指定的表的行子集，而“投影”操作是提取一个表的某些列的子集，表的行子集和列子集本身都可以看作一个表。

这样的系统之所以称为“关系”的，是因为表的数学用语为关系。（事实上，“关系”和“表”这两个词至少在非正式的情况可以当作同义词，见第 3 章和第 6 章进一步的讨论。）这里应指明在 1.3 节中对以下原因的解释是不成立的，即因为实体联系图中联系的数学用语为关系；实际上，在关系系统与这些图之间也没有多少直接的联系。

这里重复一下，我们会在后面更详细地阐述这些定义，但这里只是开个头。图 1-8 提供了一个例子。图中 a 部分的数据包含一个名为 CELLAR 的表（实际上，它是图 1-1 中的 CELLAR 表的一个简化版本，规模减小是为了便于管理）。图中 b 部分是两个查找的例子，一个包含选择或行子集操作，另一个包含投影或列子集操作。注意：两个查找都是用 SQL 来实现的。

现在我们可以这样区分关系系统和非关系系统。如前所述，关系系统的用户把数据看作表，而且只能是表（正如我们已经提到的）。在非关系系统中，用户则把数据看作其他的数据结构（代替或者扩展关系系统中的表结构）。访问这些结构需要相应的操作。例如，像 IBM 的 IMS 这样的层次系统，展现给用户的数据是树结构（层次）的集合的形式，提供用于访问这些结构的

---

⊖ 在附录 A 中提供了很典型的例子说明关系系统在这方面的能力。

⊜ 尽管正如我们将要看到的，SQL 因在某些方面背离关系模型而声名狼藉。

操作符包括遍历指针——即表示整个层次路径的指针操作符。与此对比，如本章已展示的例子所示，这是与关系系统相区分的重要特征，关系系统**没有这样的指针**。（至少从用户的角度看是没有指针的，即在模型这个层面上没有指针，当然有可能在物理实现上使用了指针。）

a. 给定表：　CELLAR

| WINE | YEAR | BOTTLES |
|---|---|---|
| Zinfandel | 1999 | 2 |
| Fumé Blanc | 2000 | 2 |
| Pinot Noir | 1997 | 3 |
| Zinfandel | 1998 | 9 |

b. 操作符（举例）：

1. 选择：

```
SELECT WINE, YEAR, BOTTLES
FROM CELLAR
WHERE YEAR > 1998 ;
```

结果：

| WINE | YEAR | BOTTLES |
|---|---|---|
| Zinfandel | 1999 | 2 |
| Fumé Blanc | 2000 | 2 |

2. 投影：

```
SELECT WINE, BOTTLES
FROM CELLAR ;
```

结果：

| WINE | BOTTLES |
|---|---|
| Zinfandel | 2 |
| Fumé Blanc | 2 |
| Pinot Noir | 3 |
| Zinfandel | 9 |

图 1-8　关系系统中的数据结构和操作符（举例）

进一步说，根据数据结构和提供给用户的操作符，数据库系统实际上能够很方便地加以分类。根据这一点，以往的系统（非关系的）可分为三大类，即**倒排表、层次和网状系统**[㊀]。（注意：这里的网状（network）与下一章在数据通信部分提到的网络（network）毫无关系。）本书中我们不详细讨论这些系统，因为至少从技术角度上看它们已经过时了。如果有兴趣，可参见文献［1.5］。

注意：我们提到的网状系统又可称为 **CODASYL** 系统或 **DBTG** 系统，这是因为其核心是由数据系统语言协会（CODASYL）下属的数据库任务组（DBTG）所提出的。可能这一系统的典型代表是 Computer Associates International 公司的 IDMS。与层次系统一样（但与关系系统不同），这些系统都把指针提供给用户。

第一代**关系**产品出现于 20 世纪 70 年代末 80 年代初。在编写本书之际，绝大多数数据库系统都是关系系统（至少支持 SQL），它们可以在各种硬件和软件平台上运行。最主要的产品（以字母顺序排列）包括：IBM 公司的 DB2；Computer Associates International 公司的 Ingres II；Informix Software 公司[㊁]的 Informix Dynamic Server；微软公司的 Microsoft SQL Server；Oracle 公司的 Oracle 9i；Sybase 公司的 Sybase Adaptive Server。注意：本书中以后谈及这些产品时，我们分别只提及缩写名字 DB2、Ingres、Informix、SQL Server、Oracle 和 Sybase。

最近，一些**对象**和**对象/关系型**数据库产品开始推向市场。对象数据库系统出现在 20 世纪 80 年代末 90 年代初，对象/关系数据库则出现于 20 世纪 90 年代末。对象关系系统表示（对大部分）某些初始的 SQL 产品向上兼容的扩展，如 DB2 和 Informix；对象系统（面向对象）表示试图做一些彻底的改变，如 GemStone Systems 公司的 GemStone 和 Versant Object Technology 公司的 Versant ODBMS。我们将会在本书第六部分讨论这些新的系统。（需要指出，这段中对象这个词有其特定的含义，这我们也将在第六部分解释。在这之前，我们只是用它表示通用情况下的意义。）

除以上谈到的数据模型方法以外，近几年的研究还提出了一些新的方法，包括**多维的**方法和

---

㊀ 和关系模型类似，在本书的前几版中使用倒排表、层次和网状模型的说法（许多文献仍然这么使用）。这里实际上有一点令人误解，因为和关系模型不同，倒排表、层次和网状模型是在相应的商业产品出现之后才随之提出来的。可以参见文献［1.1］了解更多。

㊁ 2001 年 Informix Software 公司的 DBMS 子公司被 IBM 收购。

**基于逻辑的**（也称演绎或专家）方法。我们将在第22章讨论多维系统，在第24章讨论基于逻辑的系统。随着万维网的急速发展和XML的广泛应用，出现了**半结构化**方法。我们将在第27章讨论“半结构化”系统。

## 1.7　小结

现在概括本章所讨论的主要问题。首先，**数据库系统**可以被看作计算机化的存储记录的系统。该系统包括**数据**（存储在**数据库**中）、**硬件**、**软件**（尤其是**数据库管理系统**或DBMS）和最重要的**用户**。用户又可分成**应用程序员**、**最终用户**和**数据库管理员**（DBA）。DBA负责根据数据管理员制定的策略来管理数据库和数据库系统。

数据库是**集成**和**共享**的；它们用于存储**持久数据**。这些数据通常可以表示为**实体**及实体间的**联系**——尽管实际上联系只不过是一种特殊实体。我们简要介绍了**实体/联系图**。

数据库系统有许多优点，其中最重要的一点就是（物理的）**数据独立性**。数据独立性的定义是指能使应用程序免于随着数据物理存储和访问方式的变化而变化。另外，数据独立性要求**数据模型**和它的**实现**分开。（提醒大家关注数据模型一词，它可能有两种截然不同的意义。）

数据库系统通常支持**事务**，即工作的逻辑单位。事务的优点之一是，即便在事务执行中系统出现故障，也能确保事务的**原子性**（即要么全做，要么全不做）。

最后，数据库系统以许多不同的方法为基础。特别地，关系系统以称为**关系模型**的形式化理论为基础。在关系模型中，数据表示为表中的行（解释为**真命题**），所提供的操作直接支持从已指定的真命题**推断**出其他真命题的处理过程。从经济的和理论的前景来看，关系系统很显然是最重要的（这种情况在可预见的将来是不会改变的）。我们已经给出了几个**SQL**的例子（特别是，例如SQL的**SELECT**、**INSERT**、**UPDATE**和**DELETE**语句），SQL是关系系统的标准语言。本书将着重于对关系系统的讨论，对SQL则论述不多。

## 习题

1.1　定义下列术语：

| | | | |
|---|---|---|---|
| 二元联系 | 命令驱动界面 | 并发访问 | 数据管理 |
| 数据库 | 数据库系统 | 数据独立性 | 数据库管理员 |
| 数据库管理系统 | 实体 | 实体/联系图 | 表格驱动界面 |
| 集成 | 完整性 | 菜单驱动界面 | 多用户系统 |
| 联机应用程序 | 持久数据 | 属性 | 查询语言 |
| 冗余 | 联系 | 安全性 | 共享 |
| 存储字段 | 存储文件 | 存储记录 | 事务 |

1.2　使用数据库系统的优点是什么？缺点呢？

1.3　什么是关系系统？区分关系和非关系系统。

1.4　什么是数据模型？解释数据模型与其实现的差别。为什么这个差别很重要？

1.5　给出下列对图1-1中酒窖数据库的SQL检索操作的结果：

a.
```
SELECT WINE, PRODUCER
FROM   CELLAR
WHERE  BIN# = 72 ;
```

b.
```
SELECT WINE, PRODUCER
FROM   CELLAR
WHERE  YEAR > 2000 ;
```

c.
```
SELECT BIN#, WINE, YEAR
FROM   CELLAR
WHERE  READY < 2003 ;
```

d.
```
SELECT WINE, BIN#, YEAR
FROM   CELLAR
WHERE  PRODUCER = 'Robt. Mondavi'
AND    BOTTLES > 6 ;
```

1.6　从习题1.5的每个答案中选出一行，用自己的话解释成真命题。

1.7　给出下列对图 1-1 中酒窖数据库的 SQL 更新操作的结果：

a.
```
INSERT
INTO    CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 80, 'Syrah', 'Meridian', 1998, 12, 2003 ) ;
```

b.
```
DELETE
FROM    CELLAR
WHERE   READY > 2004 ;
```

c.
```
UPDATE CELLAR
SET    BOTTLES = 5
WHERE  BIN# = 50 ;
```

d.
```
UPDATE CELLAR
SET    BOTTLES = BOTTLES + 2
WHERE  BIN# = 50 ;
```

1.8　写出对酒窖数据库执行下列操作的 SQL 语句：

(a) 找出所有 Geyser Peak 酒的 bin 号、酒的名字和瓶数。

(b) 找出存储量超过 5 瓶的酒的 bin 号和酒的名字。

(c) 找出所有红酒的 bin 号。

(d) 对 bin 号为 30 的加 3 瓶酒。

(e) 从库存中删除所有 Chardonnay。

(f) 加入一条新记录：12 瓶 Gary Farrell Merlot：bin 号 55，2000 年生产，2005 年出厂。

1.9　假定你有一些古典音乐的 CD 和/或 Midi 和/或 LP 和/或磁带，并且想要建立一个数据库来查找某作曲家（如 Sibelius），或某指挥家（如 Simon Rattle），或某独唱家（如 Arthur Grumiaux），或某作品（如贝多芬的第 5 交响曲），或某管弦乐队（如 NYPO），或某种作品（如 violin concerto），或室内乐乐队（如 Kronos Quartet）的有关资料。为此数据库画出一个像图 1-6 那样的实体/联系图。

## 参考文献

[1.1]　E. F. Codd："Data Models in Database Management," Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colo. (June 1980), *ACM SIGMOD Record 11*, No. 2 (February 1981) and elsewhere.

Codd 是关系模型的创始人，在参考文献 [6.1] 中，他首次描述了该模型。但是，文献 [6.1] 中并没有定义数据模型这个概念——不过现在的论文中包括了。他提出的问题是：数据模型，尤其是关系模型，通常的目的是什么？想要怎样？并对这个声明继续提供证明，关系模型实际上是第一个定义的数据模型。换句话说，Codd 被视为数据模型概念的创始人，同时也是关系数据模型的创始人。

[1.2]　Hugh Darwen："What a Database *Really* Is: Predicates and Propositions," in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994 – 1997*. Reading, Mass.: Addison-Wesley (1998).

在 1.3 节的末尾简要提到过，数据库最好被看作真命题的集合，本书给出了这一观点的非正式（但十分精确）的解释。

[1.3]　C. J. Date and P. Hopewell: "Storage Structures and Physical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, Calif. (November 1971).

[1.4]　C. J. Date and P. Hopewell: "File Definition and Logical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, Calif. (November 1971).

参考文献 [1.3] 和 [1.4] 第一次定义和区分了数据的物理独立性和逻辑独立性。

[1.5]　C. J. Date: *Relational Database Writings 1991 – 1994*. Reading, Mass.: Addison-Wesley (1995).

# 第2章　数据库系统体系结构

## 2.1　引言

本章介绍数据库系统的体系结构。介绍体系结构的目的是给后续章节建立一个框架结构。这个框架结构用于描述一般数据库的概念，并解释特定数据库的结构——但不能说每个数据库系统都和这个框架结构完全相匹配，或者说这一特定的体系结构提供了唯一可能的框架结构。特别是，“小”系统（见第1章）将难以支持体系结构的各个方面。不过，此体系结构基本上能很好地适应大多数系统；而且，它基本上和ANSI/SPARC DBMS研究组提出的数据库管理系统的体系结构（称作ANSI/SPARC体系结构——参见文献［2.1］和［2.2］）是相同的。但是，我们不会在每个细节部分都采用ANSI/SPARC的术语。

注意：本章和第1章类似，本章内容有助于全面认识现代数据库系统的结构和功能，但本章的内容还是有些抽象和枯燥。因此和第1章一样，读者可以先对这些内容“大致浏览”一下，待以后遇到直接相关的内容时再回过来看这部分内容。

## 2.2　三级体系结构

ANSI/SPARC体系结构分为三层：即内部层（Internal Level）、外部层（External Level）和概念层（Conceptual Level）（见图2-1），当然也可能有其他的名称。广义地讲：

- **内部层**（存储层）是最接近物理存储的——也就是，数据的物理存储方式；
- **外部层**（用户逻辑层）是最接近用户的——也就是，单个用户所看到的数据视图；
- **概念层**（公共逻辑层，或有时称为逻辑层）是介于前两者之间的间接的层次。

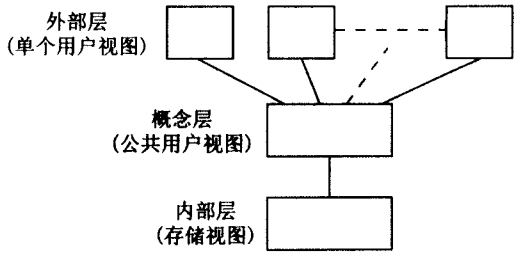


图2-1　三级体系结构

注意：外部层是单个用户的数据视图，而概念层是一个部门或企业的数据视图。正如第1章提到的，大多数用户只对整个数据库的某一部分感兴趣。换句话说，“外部视图”（即外部层）会有许许多多，每一个都或多或少地抽象表示整个数据库的某一部分，而“概念视图”（概念层）只有一个，它包含对现实世界数据库的抽象表示。同样，“内部视图”（即内部层）也只有一个，表示数据库的物理存储。注意：外部层和概念层都是模型层面上的，而内部层是实现层面上的。换言之，外部层和概念层是面向用户构造的，比如记录和字段。而内部层是面向机器构造的，比如位和字节。

举例说明如下。图2-2给出了一个有关人事数据库的概念视图，以及对应的内部视图和两个对应的外部视图（一个为PL/I用户，一个为COBOL用户[⊖]）。当然，例子完全是假设的——与任何实际系统无关——且忽略了许多无关的细节。说明如下：

1）在概念层中，数据库包含了EMPLOYEE（雇员）实体类型的信息。每个雇员都有EMPLOYEE_NUMBER（6个字符）、DEPARTMENT_NUMBER（4个字符）和SALARY（5位的十进制数）。

2）在内部层中，雇员由长度为20字节、名称为STORED_EMP的存储记录类型来表示。STORED_EMP包含四个存储字段：6字节的前缀（大概包含如代码、标记或指针这样的控制信

⊖ 很抱歉在这里我们采用这些比较早期的程序语言来举例，但无可否认的是，PL/I和COBOL在现今的商业应用中还是使用很广泛的。

息）和对应于雇员的三个属性的三个数据字段。此外，STORED_EMP 记录按雇员号字段进行索引，索引名为 EMPX，索引的定义随后给出。

3）PL/I 用户对应一个数据库的外部视图，其中，每个雇员由一条包含两个字段的 PL/I 记录来表示（部门号对该用户没有意义，故已经省略）。记录类型是根据 PL/I 的规则由通常的 PL/I结构声明来定义的。

4）类似地，COBOL 用户也对应一个外部视图，其中，每个雇员也由包含两个字段的 COBOL 记录来表示（这次，工资字段被省略）。记录类型是根据 COBOL 的规则由通常的 COBOL 记录描述来定义的。

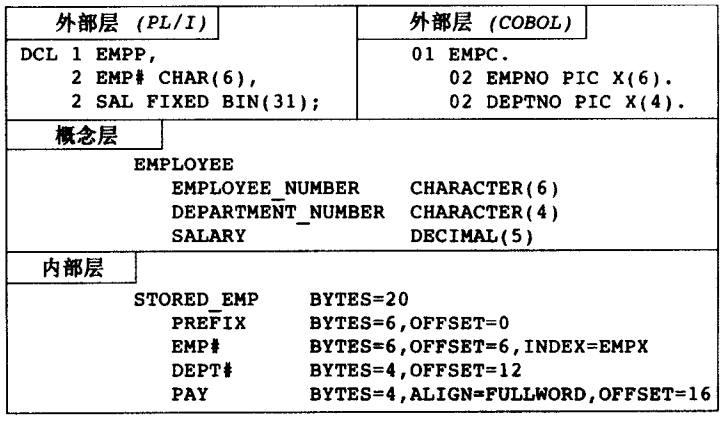



图 2-2　三级层次举例

注意：在不同的视图中相应的数据项可有不同的名字。例如，雇员号在 PL/ I 外部视图中称为 EMP#，而在 COBOL 外部视图中称为 EMPNO，在概念视图中称为 EMPLOYEE_NUMBER，而在内部视图中又称为 EMP#。当然，系统必须知道它们之间的关系：例如，要告知系统 COBOL 的 EMPNO 字段来自概念字段 EMPLOYEE_NUMBER，而 EMPLOYEE_NUMBER 又来自于内部视图的存储字段 EMP#。这种对应关系或**映像**在图 2-2 中并未清楚地表现出来；可参见 2.6 节进一步的讨论。

本章所谈论的内容对系统是不是关系的，差别不大。不过，简要说明一下关系系统中三级体系结构的情况，对具体理解这一概念不无裨益：

- 首先，关系系统的概念层一定是关系的，在该层可见的实体是关系的表和关系的操作符（尤其包括第 1 章中提到的选择和投影操作符）。
- 第二，外部视图也是关系的或接近关系的；例如，在图 2-2 中，PL/I 和 COBOL 的记录定义可以被当作与关系系统的关系表相似的 PL/I 和 COBOL 的声明。注意：这里应指出“外部视图”一词（有时简称为视图）在关系系统中有其特定的含义，它与本章提到的含义有所不同。有关它在关系系统下的含义见第 3 章和第 10 章的解释。
- 第三，内部层不是关系的，因为该层的实体不是关系表的原样照搬。其实不管是什么系统，其内部层都是一样的（如存储记录、指针、索引、散列，等等）。事实上，关系模型与内部层**无关**，正如第 1 章所指出的，它关心的是用户的数据视图。

现在我们从外模式开始进一步讨论三层体系结构的细节。整个讨论都以图 2-3 为基础，该图显示了体系结构的主要组成部分和它们之间的联系。

## 2.3　外部层

外部层就是单个用户的数据视图。如第 1 章所述，一个用户可以是应用程序员或任一最终用户。（这里 DBA 是一个特例，与其他用户不同，DBA 还要了解概念层和内部层，详见下面两节。）

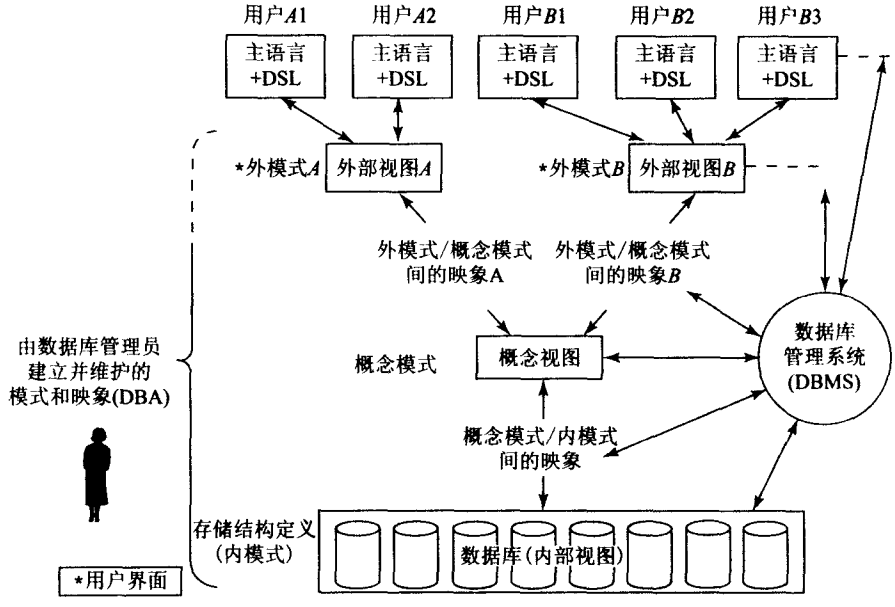



图 2-3　系统体系结构详图

每个用户都有其自己要使用的**语言**：

- 对于应用程序员，可能会使用常规的编程语言（如 PL/I、C++ 或 Java）或其他专用语言。这些专用语言通常被称作“第四代”语言（4GL），这是由于（a）机器语言、汇编语言和像 PL/I 这样的语言被视为第一、二、三代语言；（b）专用语言是对“第三代”语言（3GL）的发展，就像第三代语言对汇编语言的发展和汇编语言对机器语言的发展一样。
- 对于最终用户，其使用的语言或者是一种查询语言或是某一特定目的的语言，这些语言可能是表格驱动的或菜单驱动的，可处理用户的请求并由联机的应用程序来支持。

重要的是，所有这些语言都包含**数据子语言**——即数据库对象和操作的整个语言的一个子集。数据子语言（在图 2-3 中使用缩写词 DSL 表述）**嵌入**在相应的**主语言**中。主语言负责提供各种非数据库的功能，如局部变量、计算操作、逻辑分支等等。一个指定系统可以支持多种主语言和多种数据子语言，但是目前大多数系统支持的特定的数据子语言是 SQL 语言，第 1 章已经简要地介绍过。多数系统允许 SQL 既可以作为独立的交互查询语言，又可嵌入到诸如 PL/I、C++ 或 Java 等主语言中（见第 4 章的进一步讨论）。

尽管对体系结构来说，区分数据子语言和包含它的主语言是方便的，但对用户来说，两者实际上没有多大区别；的确，从用户的观点来看，他们宁可两者没有区别。如果没有区别，或很困难才能区别出来，我们称两者是**紧耦合**（这个整体称为一种数据库编程语言[⊖]）。如果很容易清楚地区分，我们称两者是**松耦合**。一些商业系统（包括某些 SQL 产品，如 Oracle）支持紧耦合，但多数系统不支持（紧耦合提供给用户一套统一方便的功能，但很显然，系统开发者在实现上面要花很大功夫）。

理论上，任何特定的数据子语言至少包含两个子语言——**数据定义语言**（DDL）和**数据操纵语言**（DML）。数据定义语言支持对数据库对象的定义或说明；数据操纵语言支持对这些对象的操纵和处理[⊖]。例如，考虑 2.2 节的图 2-2 中的 PL/I 用户的情况。该用户的数据子语言包括用于与 DBMS 通信的那些 PL/I 的特征：

---

⊖ 在后面几章我们将用 **Tutorial D** 作为例子，它是一种数据库编程语言。参考本书的前言，其中有关于这个主题的叙述。

⊖ “操纵”（manipulation）这个词并不是十分贴切的，但是在广泛的使用中已经得到默许。

- 数据定义语言部分包括 PL/I 用于声明数据库对象的那些声明性构造——声明（DECLARE）语句本身，某些 PL/I 数据类型，以及对 PL/I 的可能的扩展（以支持现有的 PL/I 不能处理的新对象）。
- 数据操纵语言部分包括 PL/I 的可执行语句，这些语句完成与数据库的信息传递——也可能包括特殊的新语句。

注意：准确地说，在编写本书时，PL/I 实际上根本不包括任何特殊的数据库特征。典型情况下，DML 语句只是 PL/I 用于调用 DBMS 的 CALL 语句（尽管那些语句以某种方式掩盖了语法结构以使得对用户更友好，见第 4 章对嵌入式 SQL 的讨论）。

再来看体系结构：我们已经指出，单个用户大多只对整个数据库的某些部分感兴趣；而且与数据的物理存储方式比较而言，其用户视图通常有些抽象。ANSI/SPARC 称单个用户视图的术语为**外部视图**。外部视图就是特定用户所看到的数据库的内容（即对那些用户来说，外部视图就是数据库）。例如，人事部门的用户可能把部门和雇员记录值的集合作为数据库，而没有意识到采购部门的用户所看见的供应商和零件的记录值。

通常，外部视图包括许多**外部记录**类型的值（不必和存储记录一样）[⊖]。用户数据子语言是根据外部记录来定义的；例如，数据操纵语言的检索操作将检索到外部数据记录值，而非存储记录的值。（注意：在第 1 章提到的“逻辑记录”一词实际上指的是外部记录。在以后的讨论中我们将会尽量避免使用“逻辑记录”一词。）

每个外部视图都是通过外模式来定义的，外模式包括外部视图中的各种外部记录类型的基本定义（参见图 2-2 的两个简单的例子）。外模式是使用用户数据子语言的 DDL 部分来编写的（因此有时 DDL 也称为**外部 DDL**）。例如，雇员外部记录类型就可以定义为 6 个字符的雇员号字段加 5 位（十进制数）的工资字段，等等。此外，在外模式和其下面的概念模式（见下一节）之间要定义映像。我们将在 2.6 节讨论映像。

## 2.4　概念层

概念视图是由**概念模式**（conceptual schema）定义的。概念模式包括各种概念记录型的定义（见图 2-2 中的简单例子）。概念模式是用另一种数据定义语言来写的，即**概念 DDL**。如果可以实现物理记录的独立性，那么概念 DDL 定义根本不涉及物理表示和访问的技术——它们只定义信息的内容。这样，在概念模式中不能涉及存储字段表示、存储记录队列、索引、散列算法、指针或其他存储和访问的细节。如果概念视图以这种方式真正地实现数据独立性，那么根据这些概念模式定义的外模式也会有很强的独立性（见 2.6 节）。

**概念视图**表示数据库的全部信息内容，其形式要比数据的物理存储方式抽象些。通常，它与任何特定用户观察数据的方式都很不同。广义上讲，概念视图更接近于实际数据，而不像某一用户所看到的数据——这些数据受到特定语言或可能使用的硬件的限制。

概念视图由许多**概念记录**类型的值构成。例如，它可能包括部门记录值的集合、雇员记录值的集合、供应商记录值的集合、零件记录值的集合，等等。概念记录既不和外部记录相同，也不和存储记录相同。

概念视图是整个数据库内容的视图，概念模式是该视图的定义。但如果把概念模式只理解为类似 COBOL 程序中简单的记录定义一样的一组定义，那是不准确的。在概念模式中的定义应包括许多额外的特征，诸如第 1 章中提到的安全性和完整性约束。到目前为止，有些权威人士认为概念模式的根本目的是描述整个企业的情况——不只是数据本身，而且还包括数据的使用情况，即数据在企业中的流动情况、在每一部门的用处以及对数据实行的审计和其他控制等［2.3］。

---

⊖ 在这里我们假定所有的信息在外部层中都是以记录的形式表示的。然而，很多系统中允许将信息表示成其他的形式。对于这种可以选择表示方式的系统，本节中所给出的定义和解释都需要适当地修改。对于概念模式和内部层的情况也是一样的。对这些问题的细节的考虑已经超出了本书的范围；可以参考第 14 章（尤其是该章的参考文献一节）和第 25 章以获取更多信息。内部层的情况可以参考附录 A。

但必须强调的是，目前的系统实际上还不能支持这种程度的概念模式[⊖]；目前大多数系统支持的“概念模式”实际上只不过是把单个的外模式合并起来，再加上一些安全性约束和完整性约束。但是将来的系统很可能提供更加复杂的概念模式。

## 2.5　内部层

体系结构的第三层是内部层。**内部视图**是整个数据库的低层表示；它由许多**内部记录**类型的值组成。“内部记录”是 ANSI/SPARC 对存储记录的称谓（我们继续使用后者）。内部视图与物理层仍然不同，因为它并不涉及物理记录（即物理块或页）的形式，也不考虑具体设备的柱面或磁道大小。换句话说，内部视图假定了一个无限大的线性地址空间；地址空间到物理存储的映像细节是与特定系统有关的，它未反映在体系结构中。

注意：块或页是**输入/输出的单位**——也就是说，在一次输入/输出操作中，二级存储和主存之间传输的数据量。典型的页面大小在 1KB ~ 64KB 之间，其中 1KB = 1 千字节 = 1024 字节。

内部视图由**内模式**来描述，内模式不仅定义各种存储记录，而且也说明存在什么索引，存储记录怎么表示，存储记录是在什么物理队列中等等（见图 2-2 中简单的例子，也可以参考在线的附录 D）。内模式是用另一种数据定义语言——**内部 DDL** 来写的。

注意：本书中我们通常使用更直观的词“存储数据库”代替“内部视图”，用“存储数据库定义”代替“内模式”。在某些特殊的情况下，应用程序——尤其是实用程序（见 2.11 节）——可能会直接操作内部层而不是外部层。当然，这种做法肯定是不妥的，这会带来一定的安全性和完整性隐患（即安全性约束和完整性约束会被绕过），并且应用程序会失去数据独立性。但有时为了保证功能或性能上的要求，又不得不这样做。这就像使用高级语言系统的用户，偶尔会使用汇编语言以满足特定的功能或性能上的要求一样。

## 2.6　映像

除了三级部层本身，图 2-3 中的体系结构还含有一定的**映像**关系——即概念模式/内模式间的映像和外模式/概念模式间的映像。一般地讲：

- 概念模式/内模式间的映像定义了概念视图和存储数据库间的对应关系；它说明了概念记录和字段在内部层次怎样表示。如果数据库的存储结构改变了——也就是说，如果改变了存储结构的定义——那么概念模式/内模式间的映像必须进行相应的改变，以便概念模式能够保持不变（当然，对这些变动的管理是数据库管理员或甚至可能是 DBMS 的责任）。换句话说，为了保持数据的物理独立性，内模式变化所带来的影响必须与概念模式隔离开来。
- 外模式/概念模式间的映像定义了特定的外部视图和概念视图之间的对应关系。一般地讲，这两层之间存在的差异与概念视图和存储数据库之间存在的差异是类似的。例如，字段可能有不同的数据类型；字段和记录名可以改变；几个概念字段能合成一个单一的外部字段，等等。可能同时存在多个外部视图；多个用户可共享一个特定的外部视图；不同的外部视图可能有交叉。

  不需要涉及太多细节也可以看到，就像概念模式/内模式间的映像是物理数据独立性的关键，外模式/概念模式间的映像是逻辑数据独立性的关键。如在第 1 章中所述，如果相对于数据库物理结构的改变，用户和用户的应用程序能保持不变，系统就提供了**物理数据独立性** [1.3]。同样，如果对于数据库逻辑结构的改变（即在概念或公共逻辑层的改变），用户和用户的应用程序能保持不变，系统就提供了**逻辑数据独立性** [1.4]。在第 3 章和第 10 章还会就这一重要问题做进一步的说明。
- 此外，多数系统允许以其他形式定义某些外部视图（通过外模式/外模式间的映像），而不总是要求一个明确的到概念层的映射定义。这是一个很有用的特征，特别是当几个外部视图相互非常类似时。关系系统一般会提供这种功能。

---

⊖ 有的读者可能认为“商业规则”用词更贴切（见第 9、14 章）。

## 2.7　数据库管理员

如第 1 章中所述，数据管理员（DA）是根据企业的数据制定策略和决策的人，数据库管理员（DBA）对执行这些决定提供必要的技术支持。因此，数据库管理员负责在技术层的全局控制。我们可以更详细地描述数据库管理员的任务。通常，数据库管理员的任务至少包括下列内容：

- 定义概念模式。数据管理员的工作是决定数据库中存放的信息——换句话说，是确定对企业有用的实体和实体的相关信息。这一过程通常是指**数据库的逻辑设计**（有时也称概念设计）。一旦数据管理员在抽象的层次上决定了数据库的内容，数据库管理员就使用概念 DDL 创建相应的概念模式。模式的目标形式（已编译的）由数据库管理系统在响应访问要求时使用。源形式（未编译的）作为系统用户的参考文档。

  在实际中，事情并不像前面所说的那样能够清楚地区分。某些情况下，数据管理员可能直接创建概念模式。另外，数据库管理员也可以做逻辑设计。

- 定义内模式。DBA 也决定存储数据库中数据表示的问题。这一过程通常叫做**数据库的物理设计**。完成物理设计之后，DBA 必须使用内部 DDL 创建相应的存储数据库定义（内模式）。此外，DBA 必须定义相应的概念模式/内模式间的映像。实际上，概念 DDL 或内部 DDL——更多的可能是指前者——会包括定义映像的方法，但是两方面功能（创建模式，定义映像）要清楚地分开。像概念模式一样，每个内模式和相应的映像都会以源形式和目标形式存在。

  注意这样一种顺序：首先决定你需要什么样的数据，然后决定怎样存储它。物理设计总是在逻辑设计之后。

- 与用户联络。为了确保用户需要的数据可用，并可以使用外部 DDL 来编写（或帮助用户写）必要的外模式，DBA 要负责与用户的联络。（如前所述，一个指定的系统可以支持几个不同的外部 DDL。）此外，相应的外模式/概念模式间的映像也需要定义。实际上，外部 DDL 会包括说明映像的方法，但是，模式和映像要清楚地分开。每个外模式和相应的映像都以源形式和目标形式两种方式存在。

  与用户联络的其他方面还包括就应用程序的设计给出咨询，提供技术培训，帮助决定问题和解决问题，以及类似的专业服务。

- 定义安全性和完整性约束。如 2.4 节所述，安全性和完整性约束可以当作概念模式的一部分。概念 DDL 必须包括说明这些约束的功能。

- 定义转储和重载机制。一旦企业采用了数据库系统，它就非常依赖于对系统的正确操作。如果数据库的任何部分遭到破坏——比如说人为引起的硬件错误或操作系统错误——必须能够以最小的代价恢复数据，且尽量减小对系统的影响。例如，未被破坏的数据的访问并不受影响等。DBA 必须定义和实现一个恰当的破坏控制计划。这些计划通常应包括定期卸载数据库或将数据库转储到备份存储设备上，并且当需要时从最近的转储中重载或恢复数据库。

顺便提一下，快速恢复数据的需求是将整个数据分散到几个数据库中而不是全都保存在一个数据库中的主要原因；单个数据库可能是转储和重载的最佳单元。在这个关系中，注意 TB[⊖] 级系统（即存储了若干太字节数据的商业数据库）已经存在，而且未来的系统的数据量会更大。毫无疑问，这种超大规模数据库（VLDB）系统要求非常精细和复杂的管理，尤其是要求系统不间断运行的时候。然而，为了简便起见，我们仍只考虑单一数据库的情况。

- 监控系统性能并响应不断变化的请求。在第 1 章中提到，DBA 负责组织系统，以得到对企业最佳的性能，并根据需求的改变来做相应的调整（或**调节**）。例如，可能要不时地对

---

⊖ 1024 字节 = 1 千字节（KB）；1024 KB = 1 兆字节（MB）；1024 MB = 1 吉字节（GB）；1024 GB = 1 太字节（TB）；1024 TB = 1 拍字节（PB）；1024 PB = 1 艾字节（EB 或 XB）；1024 XB = 1（ZB）；1024 ZB = 1（YB）。注意：BB 有时可以用来替代 GB（即十亿）。Gigabyte 发音为轻音 g 加上长音 i（和 gigantic 中一样）。

数据库进行**重新组织**以确保其效率。如前所述，任何对系统物理存储层的改变都要伴以相应概念模式/内模式间的映像定义的改变，以便概念模式保持不变。

当然，以上并非详尽的描述，只是试图对 DBA 的职责范围和特点给出一些介绍。

## 2.8　数据库管理系统

**数据库管理系统**（DBMS）是处理数据库访问的软件。从概念上说，它包括以下处理过程（参见图 2-3）：

1）用户使用某数据子语言（通常是 SQL）发出一个访问请求。

2）DBMS 接受请求并分析它。

3）DBMS 接下来检查用户外模式（目标形式）、相应外模式/概念模式间的映像、概念模式、概念模式/内模式间的映像和存储数据库定义。

4）DBMS 执行对数据库的必要的操作。

下面通过一个例子来看一下检索一个特定外部记录值的过程。通常，要从几个概念记录值得到字段，而每个概念记录值又需要来自几个存储记录值的字段。概念上，DBMS 首先检索所有要求的存储记录的值，然后构造所要求的概念记录值，接着再构造所要求的外部记录值。在每个阶段都可能需要数据类型或其他方面的转换。

当然，前面的描述非常简单；特别是，这暗示整个过程是解释性的，因为它表明分析请求的处理、检查各种模式等都是在运行时做的。反过来，解释意味着执行效率低，因为这增加了运行时开销。为此，实际中可能在运行之前先对访问请求进行预编译（尤其是，目前的一些 SQL 产品支持这一点——参见第 4 章的［4.13］和［4.27］）。

下面将详细地解释一下 DBMS 的功能。这些功能至少支持下列内容（可参见图 2-4）：

- *数据定义*。DBMS 必须能接受数据定义的源形式，并把它们转化成相应的目标形式。换言之，DBMS 必须包括支持各种数据定义语言（DDL）的 **DDL 处理器**或 **DDL 编译器**。在某种意义上，DBMS 必须理解 DDL 定义，例如，它理解 EMPLOYEE 外部记录包括 SALARY 字段；并能够利用这一知识来分析和响应数据操纵要求（例如，“找出薪金小于 50 000美元的雇员”）。
- *数据操纵*。DBMS 必须能够检索、更新或删除数据库中现有的数据，或向数据库增加数据。换句话说，DBMS 必须包括处理数据操纵语言（DML）的 **DML 处理器**或**编译器**。

通常，DML 请求可以是“计划”的或“非计划”的：

a）*计划的请求*是指在请求执行前就能预见到有关的需求。DBA 可以据此调整物理数据库的设计，以保证请求有好的执行性能。

b）相反，*非计划的请求*是指需求是不可预知的，是一种特殊查询。物理数据库的设计不一定能够真正适合处理这样的请求。

使用第 1 章中的术语，计划的请求是“操作型”或“生产型”应用的特征，而非计划的请求是来自“决策支持”类应用。计划的请求通常来自预先写好的应用程序，而根据定义，非计划的请求是通过某查询语言处理器交互发出的。（事实上，如第 1 章所述，查询语言处理器是一个内置的联机应用程序，而不是 DBMS 本身的一部分。为了完整，我们把它包括在图 2-4 中。）

- *优化和执行*。计划的或非计划的 DML 请求必须经过**优化器**部件的处理，优化器用于决定有效执行请求的方式[⊖]。有关优化的问题将在第 18 章进行详细讨论。优化后的请求在**运行管理器**的控制下执行。（*注意*：在实际操作中，运行管理器调用文件管理器来访问存储的数据。有关文件管理器的内容将在本节的末尾进行详细讨论。）
- *数据安全性和完整性*。DBMS 要监控用户的请求，拒绝那些企图破坏 DBA 定义的数据库安全性和完整性约束的请求。在编译时或运行时或两种情况同时存在时都会执行这些

⊖ 本书中所说的“优化”是专指对 DML 请求的优化，不包括其他。

任务。

- 数据恢复和并发。DBMS——或其他一些相关的软件，通常称作**事务管理器**或**事务处理监控器**——必须保证要有恢复和并发控制。该细节内容已经超出了本章的范围；详细讨论见本书的第四部分。注意：事务管理器没有在图 2-4 中出现，因为它通常并不是 DBMS 的组成部分。
- 数据字典。DBMS 包括**数据字典**。也可以将数据字典本身看作是一个数据库（系统数据库而不是用户数据库）。字典是“数据的数据”（有时称为数据的**描述**或**元数据**），即系统中其他实体的定义，而不只是“原始数据”。特别地，在数据字典中，各种模式和映像（外部的、概念的，等等）及数据的各种安全性和完整性约束都可以用源和目标两种形式来存储。一个易用的字典还包括许多其他信息，例如，给出哪个程序用数据库的哪个部分，哪个用户使用哪个报表等。数据字典甚至还可以集成到它定义的数据库中，因而包括它自己的定义。当然查询数据字典和查询其他数据库是相同的，因此，例如，可以分出哪个程序或哪个用户会受到系统改变的影响。详细讨论见第 3 章。

  注意：在这一领域有许多术语相互混淆。有些人把“数据字典”称作目录或分类——暗示目录或分类比真正的数据字典处于更内层——而用“字典”一词指某种（重要的）应用开发工具。有时还用“数据存储池”（见第 14 章）或“数据百科全书”来指代后者。
- 性能。毫无疑问，DBMS 应尽可能高效地完成上述任务。

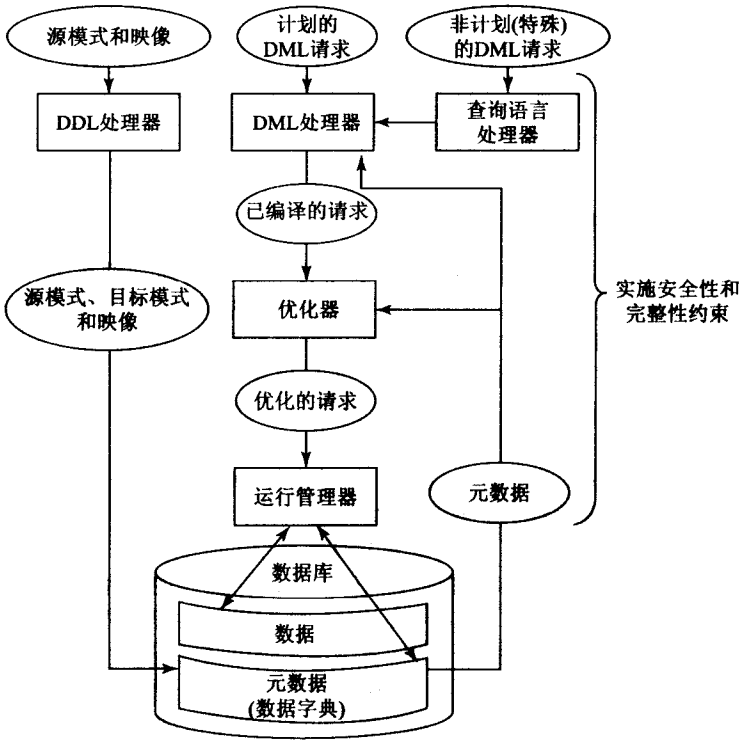



图 2-4　DBMS 的主要功能和组成

总而言之，DBMS 的目的就是提供数据库系统的**用户接口**。用户接口可定义为系统的边界，其内部细节对用户来说是不可见的。因此，根据定义可知，用户接口是外模式。不过，有时外部视图与它下面的概念视图的相应部分不可能完全分清，至少目前的商用 SQL 产品还不能将外部视图完全从下面的层次中独立出来，我们将在第 10 章详细讨论这个问题。

以下通过简单地对比数据库管理系统和文件管理系统（文件管理器或文件服务器）来结束本节。**文件管理器**是操作系统管理文件的部件；它比 DBMS 更“接近磁盘”（事实上，DBMS 通常是建立在某文件管理器之上的，参见在线的附录 D）。因此，文件管理系统的用户就可以创建或删除文件，并执行对这些文件中记录的简单检索和更新操作。然而，与 DBMS 相比：

- 文件管理器并不了解记录的内部结构，因而不能处理与结构相关的请求。
- 文件管理器一般很少提供或根本不支持安全性和完整性约束。
- 文件管理器一般很少提供或根本不支持恢复和并发控制。
- 在文件管理层没有真正的数据字典的概念。
- 文件管理器提供很少的数据独立性。
- 文件一般不像数据库那样具有“统一性”或“共享性”，文件通常是用户或应用程序专用的。

## 2.9　数据通信

本节简要讨论**数据通信**。终端用户的数据库请求实际上以消息的形式从用户工作站——在物理位置上，对于数据库系统本身来说，工作站可能是远程的——传递给一些联机应用程序（内建的或其他），并传递给 DBMS。同时，DBMS 和联机应用程序对用户工作站的响应也以消息的形式传递。所有这些消息通信都是由另一个软件部件——**数据通信管理器**来控制的。

数据通信（DC）管理器不属于 DBMS，其本身是一个相对独立的系统。但由于要与 DBMS 协同工作，有时两者在称为**数据库/数据通信系统**（DB/DC 系统）的较高层上被当作同等的部分，在该系统中 DBMS 处理数据库，而数据通信管理器处理传送给 DBMS 和从 DBMS 发出的消息。更确切地说，数据通信管理器处理传送给使用 DBMS 的应用程序和从应用程序发出的消息。但是，本书对消息处理介绍得较少（它本身是一个大课题）。2.12 节简要地讨论了在不同系统间通信的问题（也就是，在像因特网那样的通信网络中不同机器之间通信的问题），但实际上那又是一个独立的主题。

## 2.10　客户/服务器体系结构

到目前为止，本章已经从 ANSI/SPARC 体系结构的角度论述了数据库系统。特别地，在图 2-3中给出了该体系结构的示意图。本节从一个稍微不同的角度来看数据库系统。

当然，数据库系统的全部目的是支持开发和执行数据库应用程序。从较高层来看，可以将数据库系统看作是由两个非常简单的部分组成：一个服务器（也称为后端）和一组客户（也称为前端），见图 2-5。说明如下：

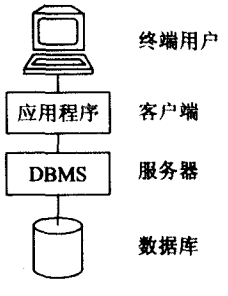


图 2-5　客户/服务器体系结构

1）**服务器**就是指 DBMS 本身，它支持 2.8 节讨论的 DBMS 的所有基本功能——数据定义、数据操纵、数据安全性和完整性，等等。换句话说，在这样的上下文中，“服务器”一词就是 DBMS 的另一个称谓。

2）**客户**是指在 DBMS 上运行的各种应用程序——用户编写的应用程序和内置的应用程序（即 DBMS 厂商或某第三方厂商所提供的应用程序）。对服务器来说，用户编写的应用程序和内置的应用程序之间没有什么不同——它们都使用相同的服务器接口，即在 2.3 节提到的外模式接口。（注意：如 2.5 节所述，某些特殊的“实用程序”会与前面的有些不同，因为它们有时会直接操纵系统的内模式。这些实用程序最好作为 DBMS 的内部构件，而不是通常意义下的应用程序。这些在下一节将会详细讨论。）

我们简要地阐述一下用户编写的应用程序和厂商提供的应用程序：

- **用户编写的应用程序**基本上是规范的应用程序，采用第三代语言（如 C++ 或 COBOL）或一些专门的第四代语言来编写——尽管在两种情况下该语言都要有相应的数据子语言来与之匹配，如 2.3 节所述。

- **厂商提供的应用程序**（常称为**工具**）的基本目的是支持创建和执行其他应用程序。所创建的应用程序是为某些特定任务定制的（它们可能不太像传统的应用程序；的确，工具的关键就是允许用户，特别是终端用户，能够创建应用程序而且不必使用传统的程序设计语言来编写）。例如，厂商提供的一种工具是报表编写器，其目的是允许终端用户能够获得系统根据其请求提供的格式化报表。任何给定的报表请求可以被当作一个小的应用程序，它是用高层的（专用的）报表编写语言来写的。

  厂商提供的工具大致可以分成以下几类：

  a）查询语言处理器
  b）报表编写器
  c）商用图形子系统
  d）电子制表软件
  e）自然语言处理器
  f）统计包
  g）复制管理或“数据提取”工具
  h）应用程序生成子（包括第四代语言处理器）
  i）其他应用程序开发工具，包括计算机辅助软件工程（CASE）产品
  j）数据挖掘和可视化工具

及其他许多软件。大多数这些工具软件的细节已经超出了本书的范围；但是，因为数据库系统的目的就是支持应用程序的创建和执行，所以可获得的工具软件的质量在“选择数据库”（即选择恰当的数据库产品的过程）时当然是一个主要因素。换句话说，尽管 DBMS 是一个非常重要的因素，但 DBMS 本身并非要考虑的唯一因素。

根据以上所述可知，既然整个系统可以清楚地分成服务器和客户端两个部分，就可能出现将这两个部分分置于不同的机器上的情形。换句话说，可能存在**分布式处理**。分布式处理是指通过通信网络的连接，一个数据处理任务可以分散到网络中的不同机器上分别进行处理。事实上，这一可能性如此吸引人——有大量原因，主要是经济上的——以至于术语“客户/服务器”几乎唯一适用于的确需要在不同的机器上部署客户和服务器的情况。我们将在 2.12 节论述分布式处理。

## 2.11　实用程序

**实用程序**（utility）是设计用于帮助 DBA 处理各种管理任务的程序。前一节已经提到，一些实用程序运行于系统的外模式层，这样它们只不过是特定目的的应用程序；甚至有些不是 DBMS 厂商提供的，而是由第三方提供的。但是，其他实用程序直接运行于系统内模式层（换言之，它们实际上是服务器的一部分），因而必须要由 DBMS 厂商提供。

下列是实际中需要的各种实用程序的一些实例：

- *载入例程*，从常规数据文件创建初始化数据库版本；
- *卸载/重载例程*，卸载数据库或其一部分，备份数据库，并从这些备份的副本重新装入数据（当然，“重载”实用程序和上述的载入实用程序基本上是一样的）；
- *重组织例程*，由于各种原因要重新组织数据库中的数据——例如，将磁盘上的数据聚集起来，或回收逻辑上废弃数据所占用的空间；
- *统计例程*，计算各种性能统计数据，如文件大小、数值分布或输入/输出次数等等；
- *分析例程*，分析上述的统计数据。

当然，以上提到的只是实用程序所提供功能范围中的小部分；还存在其他丰富的功能。

## 2.12　分布式处理

在 2.10 节已经提到过，分布式处理是指不同机器可以通过通信网络（如因特网）连接起来，这样，一个数据处理任务可以分散到网络中的几台机器上进行分别处理。（“并行处理”一

词有时也用于同样的意义，除了不同的机器物理上连接成“并行”系统而不是这样的“分布”系统之外——例如，它们可以在地理上是分散的。）在不同机器之间的通信是由网络管理软件来处理的——可能是2.9节中提到的数据通信管理器的扩展，还可能是一个独立的软件部件。

在各种层次上都可能存在分布式处理，分布式处理也可能是各种各样的。2.10节已经提及，一个简单的例子是在一台机器上运行DBMS后端（服务器），而在另一台机器上运行应用程序前端（客户端），见图2-6。

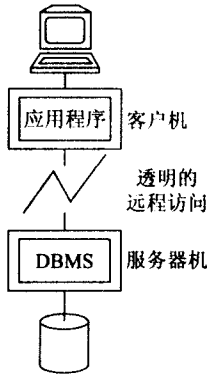



图2-6　运行在不同的机器上的客户机和服务器

在2.10节的末尾已经提到过，尽管“客户/服务器”严格地说是体系结构术语，但它已经成为图2-6中图例方案的同义词，其中客户和服务器运行在不同的机器上。有许多理由支持这样一种方案：

- 从通常的并行处理的角度考虑；许多处理部件都可用于整个任务，而且服务器和客户处理并行进行。这样，响应时间和吞吐率就提高了。
- 服务器机可以是为数据库管理系统功能而定制的机器（数据库机），这样就使数据库管理系统更高效。
- 客户机可以是根据终端用户的需要定制的个人工作站，这样就为用户提供了更好的界面、高可用性、快速响应和全面提高的易用性。
- 几台不同的客户机可能访问同一台服务器。这样，几个不同的客户系统可以共享一个数据库（见图2-7）。

此外，在不同机器上运行客户机和服务器符合企业的实际运作方式。一个企业（例如银行）运行许多计算机是很常见的，这样，企业就可以把一部分数据存在一台机器上，而另一部分数据存在另一台机器上。一台机器的用户偶尔访问其他机器上的数据也是很平常的。继续看银行这个例子，一个分支机构的用户偶尔需要访问其他的数据。注意，客户机有自己的存储数据，服务器有自己的应用程序。因此，一般地讲，每台机器都会作为一些用户的服务器和另一些用户的客户机（见图2-8）；换句话说，在本章前几节讨论的意义下，每台机器都将支持整个数据库系统。

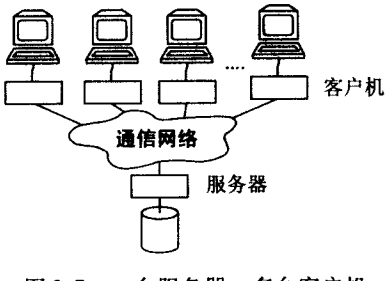



图2-7　一台服务器，多台客户机

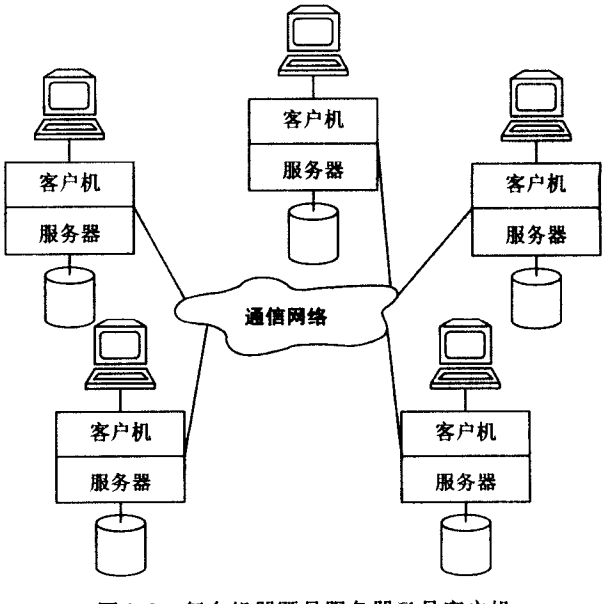



图2-8　每台机器既是服务器又是客户机

最后一点是一台客户机可能访问几台不同的服务器机（与图 2-7 中的例子的情况相反）。如前所述，企业非常需要这种功能，因为企业典型的运转方式是整个数据不是存储在单个的机器上，而是分散在许多不同的机器上，而且应用程序需要从不止一台机器访问数据。基本上，有两种方式可以提供这种访问：

- 一台客户机要访问任意数目的服务器，但是一次只能访问一个（即每个单独的数据库请求只向一台服务器发出）。在该系统中，不可能在一个请求中将两个或更多的服务器的数据结合起来。系统中的用户必须了解哪台机器存有哪些数据。
- 客户机可以同时访问许多服务器（即一个数据库请求可以将几个不同数据库的数据结合起来）。在这种情况下，对客户机来说，这些服务器（从逻辑上看）好像实际上只是一台服务器，系统的用户不必了解哪台机器存有哪些数据。

后一种情况就是通常所说的**分布式数据库系统**。分布式数据库本身是一个大课题，随之而来的逻辑结论是，完整的分布式数据库支持意味着，单个的应用程序应该可以“透明地”操纵数据，这些数据分布在各种不同的数据库中，由不同的 DBMS 管理，运行在不同的机器上，受不同的操作系统支持，通过各种不同的通信网络连接起来——这里，“透明地”的意思是指从逻辑上看，应用程序操纵数据，就像数据都由运行在同一台机器上的 DBMS 来管理。这种功能听起来有点儿离谱！但是从实际的角度来看，这是非常有吸引力的，而厂商正在努力实现这样的系统。我们将在第 21 章详细讨论这些系统。

## 2.13　小结

本章从体系结构的角度分析了数据库系统。首先，我们介绍了 **ANSI/SPARC 体系结构**，它将数据库分成了内模式、外模式和概念模式三层。其中，**内模式**最接近物理存储（即它要考虑数据的物理存储）；**外模式**最接近用户（即它要考虑单个用户看待数据的方式）；而**概念模式**则是介于前两者之间的中间层（它提供数据的公共视图）每一层的数据由一个模式描述（在外部层可能是几个模式来描述）。**映像**定义了给定的外模式与概念模式之间的对应关系，以及概念模式与内模式之间的对应关系。映像是提供**数据逻辑独立性**和**数据物理独立性**的关键。

用户——可分为终端用户和应用程序员，两者都操作外模式——通过**数据子语言**与数据交互。数据子语言至少又分成两部分：**数据定义语言**（DDL）和**数据操纵语言**（DML）。数据子语言嵌入在**主语言**中。注意：主语言和数据子语言间的界限与数据定义语言和数据操纵语言间的界限都是概念上的；在实际应用中，它们对用户来说是透明的。

我们进一步看到了 **DBA** 和 **DBMS** 的功能。DBA 负责创建内模式（**数据库的物理设计**）；相对地，创建概念模式（数据库的**逻辑**或**概念**设计）是由数据管理员来负责的。而 DBMS 负责执行用户的数据定义语言和数据操纵语言的请求。DBMS 也负责提供某些**数据字典**的功能。

数据库系统也可以由一台**服务器**（DBMS）和一组**客户机**（应用程序）来构成。客户机和服务器能够在不同的机器上运行，这样提供了一种**分布式处理**。一般地讲，每台服务器能为几台客户机提供服务，而每台客户机也可以访问多个服务器。如果系统提供完全透明的访问（也就是每台客户机就好像在单机的单个服务器上操作，而不考虑物理连接状态），那么就真正实现了**分布式数据库系统**。

## 习题

2.1　画出本章的数据库系统体系结构图（ANSI/SPARC 体系结构）。

2.2　解释下列术语：

| | | |
|---|---|---|
| 后端 | 客户机 | 概念 DDL，概念模式，概念视图 |
| 概念模式/内模式映像 | 数据定义语言 | 数据字典 |
| 数据操纵语言 | 数据子语言 | 数据库/数据通信系统 |
| 数据通信管理器 | 分布式数据库 | 分布式处理 |
| 外部 DDL，外模式，外部视图 | 外模式/概念模式映像 | 前端 |
| 主语言 | 载入 | 数据库的逻辑设计 |

内部 DDL，内模式，内部视图　　数据库的物理设计　　计划的请求
重组织　　服务器　　存储数据库定义
卸载/重载　　非计划的请求　　用户接口
实用程序

2.3　描述检索一个特定的外部记录值的步骤。
2.4　列出 DBMS 的主要功能。
2.5　区分数据的物理独立性和逻辑独立性的差别。
2.6　如何理解元数据？
2.7　列出 DBA 的主要职能。
2.8　区分 DBMS 和文件管理系统的差别。
2.9　给出几个厂商提供的工具的例子。
2.10　给出数据库的实用程序的几个例子。
2.11　观察一个数据库系统。试用本章描述的 ANSI/SPARC 体系结构来看系统。它明显地支持三层体系结构吗？层之间的映像是如何定义的？各种数据定义语言（外部的，概念的，内部的）的功能是什么？系统支持什么数据子语言？主语言是什么？谁履行 DBA 的职能？有安全性和完整性机制吗？有字典吗？字典是自描述的吗？系统支持什么厂商提供的应用程序？什么实用程序？有没有独立的数据通信管理器？有没有分布式处理的能力？

## 参考文献

下面的参考文献大部分写作时间比较早（除了最后一个），但是与本章介绍的概念相关。除此之外，还可以参考第 14 章的参考文献。

[2.1] ANSI/X3/SPARC Study Group on Data Base Management Systems: Interim Report, *FDT* (bulletin of ACM SIGMOD) 7, No. 2 (1975).

[2.2] Dionysios C. Tsichritzis and Anthony Klug (eds.): "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems," *Information System 3* (1978).

参考文献 [2.1] 和 [2.2] 是 ANSI/SPARC 研究组的中期和终期报告。有关数据库管理系统的 ANSI/X3/SPARC 研究组成立于 1972 年，是由美国国家标准协会（ANSI）的标准规划和需求委员会（SPARC）为计算机和信息处理而建立的。（约 25 年后，X3 这一名字已经改为 NCITS——国家信息技术标准委员会。几年后，又更名为 INCITS——国际信息技术标准委员会。）研究组的目的是决定数据库技术的哪些领域适合标准化，并对每个这样的领域给出一组推荐方案。在为此目的工作时，研究组认为，数据库系统适合标准化的唯一方面是接口，因此定义了一个一般的数据库体系结构或框架，强调这些接口的作用。终期报告提供了体系结构的详细描述和 42 种接口。中期报告是早期的文档；在一些领域还提供了额外的细节。

[2.3] J. J. van Griethuysen (ed.): "Concepts and Terminology for the Conceptual Schema and the Information Base," International Organization for Standardization (ISO) Technical Report ISO/TR 9007: 1987 (E) (March 1982; revised July 1987).

这个文档是 ISO 工作组的报告，其目的包括"概念模式语言的概念的定义"。它介绍了对一组形式方法的三组候选方案，并且把三者都应用到一个普通的例子，即假定的汽车注册授权活动。三组竞争者是（1）"实体 - 属性 - 联系"模式；（2）"二元关系"模式；（3）"解释的谓词逻辑"模式。报告还包括了概念模式的基本概念，提出了一些理论，以作为支持这一概念的系统实现的基础。对系统的概念层有兴趣的人来说，这是一篇重要的文档。

[2.4] William Kent: *Data and Reality*. Amsterdam, Netherlands: North-Holland/New York, N. Y.: Elsevier Science (1978).

一场对信息的本质和概念模式的本质的讨论。"本书提出了哲学原理，生活和现实本质上是无定形的，无序的，矛盾的，不一致的，不合理的，非客观的"（除了最后一章）。这本书可以被看作是真实生活问题的概述，即现存的数据库形式方法——尤其是基于传统的记录型结构的形式，包括关系模型——在处理上有困难。推荐阅读。

[2.5] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis: "The GMAP: A Versatile Tool for Physical Data Independence," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

GAMP 表示通用多层访问路径。文章的作者注意到今天的数据库产品“强迫用户使用与物理结构紧密联系的逻辑模式形式的查询”，因此数据的物理独立性非常差。在文章中，他们提出一种概念模式/内模式间的映像语言，可用于说明比今天的产品所支持的多得多的映像。指定一个“逻辑模式”，该语言（基于关系代数，见第 7 章，其本质上是说明性的，而非过程性的）允许许多不同的物理（或内部）模式的说明，它们多来自于逻辑模式。物理模式包括垂直的和水平的部分（见第 21 章）、任意数量的物理访问路径、聚集和控制的冗余。

这篇论文还给出了一个算法，该算法将用户对逻辑模式的操作转化为等价的对物理模式的操作。一个原型系统显示，数据库管理员能调整物理模式以获得比通常更好的性能。

# 第3章　关系数据库简介

## 3.1　引言

在第1章中已经介绍过，本书的重点在于关系系统。特别是第二部分比较深入地阐述了这些系统（关系模型）的理论基础。为了更好地理解本书随后部分的内容，本章对第二部分的内容（附带随后的部分）做一个直观、非正式的介绍。本章所述的多数论题将在后面章节更详细、更正式地讨论。

## 3.2　关系模型概述

在第1章中已经介绍过，关系系统基于正规的关系基础或理论，即关系数据模型。我们经常从以下三个方面来描述关系模型：

- *结构化方面*：数据库中的数据对用户来说是表，并且只是表。
- *完整性方面*：数据库中的这些表满足一定的完整性约束（在本节最后讨论）。
- *操纵性方面*：用户可以使用用于表操作的操作符——例如，为了检索数据，需要使用从一个表导出另一个表的操作符。其中，选择、投影和连接这三种操作符尤为重要。

图3-1中显示的是一个简单的关系数据库，即部门和雇员数据库。正像你所看到的，数据库给人的感觉就是一张张的表（这些表的含义是不言而喻的）。图3-2中显示了对图3-1中数据库的选择（restrict）、投影和连接操作。下面给出这些操作的定义：

DEPT

| DEPT# | DNAME | BUDGET |
|---|---|---|
| D1 | Marketing | 10M |
| D2 | Development | 12M |
| D3 | Research | 5M |

EMP

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E1 | Lopez | D1 | 40K |
| E2 | Cheng | D1 | 42K |
| E3 | Finzi | D2 | 30K |
| E4 | Saito | D2 | 35K |

图3-1　部门和雇员数据库（样本值）

- 选择操作是从表中提取特定的几行。*注意*：restrict操作有时也称作选择操作，这里用restrict这个词，因为这个操作和SQL中的SELECT操作不同。
- 投影操作是从表中提取特定的几列。
- 连接操作是根据某一列的值将两个表连接起来。

*Restrict:*　　*Result:*

DEPTs where BUDGET > 8M

| DEPT# | DNAME | BUDGET |
|---|---|---|
| D1 | Marketing | 10M |
| D2 | Development | 12M |

*Project:*　　*Result:*

DEPTs over DEPT#, BUDGET

| DEPT# | BUDGET |
|---|---|
| D1 | 10M |
| D2 | 12M |
| D3 | 5M |

*Join:*

DEPTs and EMPs over DEPT#

*Result:*

| DEPT# | DNAME | BUDGET | EMP# | ENAME | SALARY |
|---|---|---|---|---|---|
| D1 | Marketing | 10M | E1 | Lopez | 40K |
| D1 | Marketing | 10M | E2 | Cheng | 42K |
| D2 | Development | 12M | E3 | Finzi | 30K |
| D2 | Development | 12M | E4 | Saito | 35K |

图3-2　选择、投影和连接（例子）

在图3-2的三个例子中，最后一个关于连接的例子需要进一步解释。先观察到DEPT和EMP

两个表都有一个共同的列 DEPT#，因此它们可以根据这一列的相同值连接起来。当且仅当两个表中对应行的 DEPT#值相同时，DEPT 表的一行才能连接 EMP 表中对应的一行（产生结果表的一行）。例如，DEPT 和 EMP 行

| DEPT# | DNAME | BUDGET |
|---|---|---|
| D1 | Marketing | 10M |

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E1 | Lopez | D1 | 40K |

连接起来产生如下的结果行：

| DEPT# | DNAME | BUDGET | EMP# | ENAME | SALARY |
|---|---|---|---|---|---|
| D1 | Marketing | 10M | E1 | Lopez | 40K |

因为在公共列它们有相同的值 D1。注意在结果行中相同的值只出现一次。连接的结果包含了所有可能以这种方式得到的行。尤其注意尽管在 DEPT 表中有 D3 一行，但因为 EMP 表中没有 DEPT#值为 D3 的行，结果中就没有出现 D3 行。

现在，图 3-2 中清楚地显示出三种操作的每个结果都是一个表（如前所述，实际上是从一个表导出另一个表的操作）。这是关系系统的**闭包**特性，这一特性非常重要。基本上，因为任何操作的输出和其输入的对象种类相同（它们都是表），所以一个操作的输出能变成另一个的输入。因此，可以采取如连接的投影、两个选择后的连接或一个投影的选择等操作。即我们可以编写嵌套的关系表达式来处理数据——操作数本身也是由一个关系表达式来表示的，而不仅仅是一个表名。这样随之而来又有许多重要结论，在本章后面及后面的许多章中会介绍。

顺便提一下，当提到一个操作的输出是另一个表时，要知道这是从概念视图的角度来说的，这一点非常重要。这并不是意味着系统实际上必须将每个单独操作的结果实例化[⊖]。例如，假设要计算一个连接的选择，那么，连接后的行一形成，系统就立即检查该行是否满足指定的条件，以判定是否属于最终结果，如果不是就立即抛弃。也就是说，连接操作的中间结果根本不会以完整的实例表形式存在。在实际操作中，通常为了提高效率，系统尽可能不将中间结果生成完整的表。注意：如果中间结果全部实例化，整个表达式的计算策略就是**实例化的计算**；如果中间结果分块地提供给下一步操作，就称作**流水线计算**。

图 3-2 中还清楚地说明了一点：操作是**一次一集合**，而不是一次一行；也就是说，操作数和结果是完整的表，而不只是单行，是包含行集的表。（当然，只包含一行的表是合法的；空表，即根本不包含任何行的表也是合法的。）例如，在图 3-2 中，分别对两个表对应的 3 行和 4 行的连接操作，就返回一个 4 行的结果表。相应地，非关系系统中典型的操作是一次一行或一次一记录；因此，集合处理能力是关系系统区别于其他系统的一个重要特征（见下面 3.5 节进一步的讨论）。

再回到图 3-1。结合图中的样本数据库，还可以得出以下两点：

- 关系系统要求只让用户所感觉的数据库是一张张表。在关系系统中，表是逻辑结构而不是物理结构。实际上，系统在物理层可以使用它所喜欢的方式来存储数据——使用有序文件、索引、散列、指针链、压缩，等等——只要能将存储表示映射到逻辑模式的表。换句话说，表是对物理存储数据的一种抽象表示——对许多存储细节的抽象，如存储记录的位置、存储记录顺序、存储数据值的表示、存储记录的前缀、存储的访问结构如索引，等等，对用户来说都是不可见的。

  此外，在 ANSI/SPARC 中，前述的逻辑结构一词意味着包含概念模式和外模式。关键是——如第 2 章中所述——概念模式和外模式都是关系的，而物理模式和内模式不是。关系理论与内模式毫无关系，它只是考虑数据库应怎样呈现给用户[⊖]。关系系统唯一的要

---

⊖ 换言之，在第 1 章已经提过，关系模型实际上只是一个模型——它与实现并无关系。

⊖ 不幸的是，今天大部分的 SQL 产品并不能有效地支持这方面的理论。更具体一些，它们只支持受限的概念/内部映射。它们将一个逻辑表直接映射成一个存储文件。这就是这些产品并不能支持像关系理论所描述的那样完整的数据独立性的原因。可以参考附录 A 中进一步的讨论。

求是无论选择什么物理结构，一定要全部实现其逻辑结构。

- 关系数据库遵守一条非常好的原则，即**信息原则**：数据库全部的信息内容有一种表示方式而且只有一种，也就是表中的行列位置有明确的值。这种表示是关系系统中唯一可行的方式（当然，在逻辑层）。特别地，**没有**连接一个表到另一个表的**指针**。在图 3-1 中，例如，表 DEPT 中的 D1 行和表 EMP 的 E1 行有联系，因为雇员 E1 在部门 D1 工作；但是这种联系不是通过指针来表示的，而是通过表 EMP 的 E1 行中 DEPT#列的值为 D1 来联系的。相反地，在非关系系统中，这些信息典型地由指针来表示，这种指针对用户来说是可见的。

  注意：我们将在第 26 章解释，为什么允许这种用户可见的指针会违反信息原则。当指出关系数据库中没有指针时，并不是指在物理层没有指针——正好相反，关系数据库在物理层使用指针，但在关系系统中，所有物理存储的细节对用户来说都是不可见的。

对关系模型的结构和操纵方面就提这些；现在来看完整性方面。观察图 3-1 中的部门和雇员数据库。事实上，可以要求数据库满足任何条件的完整性约束——如雇员工资必须在 25K ~ 95K 的范围，部门预算在 1M ~ 15M，等等。某些约束在应用上非常重要，它们喜欢用特定的术语。具体来说：

1）DEPT 表中每一行的 DEPT#值必须是唯一的；同时，EMP 表中每一行的 EMP#值必须唯一。DEPT 表中的 DEPT#列和 EMP 表中的 EMP#列都是它们各自表的**主码**。（回忆第 1 章，在图中主码是用双下划线标出来的。）

2）EMP 表中的每个 DEPT#值必须在 DEPT 表中存在，以反映每个雇员必须安排在现有的部门中。即 EMP 表中的 DEPT#列是**外码**，参照了 DEPT 表的主码。

**更正式的定义**

现在我们给出一个更正式的关系模型的定义来结束这一节，以便于后面参照（尽管该定义十分抽象，且目前非常难理解）。简而言之，关系模型包括下列 5 部分：

- 一个可扩展的**标量类型**的集合（尤其包括布尔型或真值型）；
- **关系类型生成子**和对应这些关系类型的解释器；
- 实用程序，用于定义生成关系类型的**关系变量**；
- 向关系变量赋关系值的**关系赋值**操作；
- 从其他关系值中产生关系值的、可扩充的**关系操作符**集合（关系代数）。

由此可见，关系模型要比“表加上选择、投影和连接”多得多，尽管关系模型常常以此为特征。

另外，你可能会惊讶于没有对完整性约束进行明确的定义。事实上这些约束表示的只是关系操作符的一个应用；即这些约束是由操作符来表达的，详见第 9 章。

## 3.3　关系和关系变量

如果关系数据库只是一个用表来表示数据的数据库——当然这是真的——那么问题是：为什么称这样的数据库是关系的？答案很简单（这在第 1 章已经提到），“关系”只是表的数学术语——精确地说，是某特殊种类的表（细节见第 6 章）。因此我们可以说，图 3-1 中的部门和雇员数据库包含了两个关系。

在非正式的情况下，经常将“关系”和“表”作为同义词，并且“表”一词比“关系”一词更常用。但是花点时间去理解为什么首先介绍“关系”这个词是值得的。简单地说，有如下原因：

- 关系系统基于关系模型，反过来关系模型又是基于数学方面的数据抽象理论（主要是集合论和谓词逻辑）。
- 关系模型理论是 E. F. Codd 在 1969 年到 1970 年提出的，当时他是 IBM 的研究员。1968 年末，数学家 Codd 首先意识到数学可以向数据管理领域提供坚实的理论和严密的逻辑，在那之前，数据管理领域太缺乏这些理论的支持。Codd 的想法首先在一篇经典的论文中

提出，即“A Relational Model of Data for Large Shared Data Banks”（参见第6章中文献[6.1]）。

- 从那时起，那些思想——至今几乎全世界都接受了——就在数据库技术的各个方面产生了广泛的影响。同样在其他领域，如人工智能、自然语言处理和硬件设计等领域也产生了广泛的影响。

目前，最初由Codd建立的关系模型特意采用了这些术语，如关系这个词本身，当时在信息技术界还不是很流行（尽管有时候较常见）。问题是，许多常用的术语很模糊——它们缺乏精确性，而精确性对Codd所提出的形式理论是很必要的。例如，“记录”这个词，不同情况下，“记录”可以表示记录值或记录型，逻辑记录或物理记录，存储记录或虚记录，还可以有其他的表示。因此，关系模型不使用记录这一词，而用“元组”来进行精确的定义。我们将在第6章给出定义的细节的讨论。目前可以说“元组”这个词近似地对应一行的概念（就像“关系”这个词近似地对应一个表的概念）。

同样的，关系模型中并不使用“域”这个词，而用“属性”，目前来说它就相当于一个表中的一列。

当在第二部分进一步讨论关系系统时，将利用这些正式术语，但是本章只进行非正式的介绍。我们将继续使用像“行”或“列”这些相当常见的词，不过以后还会使用“关系”这个词。

再回到图3-1中的部门和雇员数据库来看另外一个重点。实际上，数据库中的DEPT和EMP是关系**变量**，即它们的值是关系**值**（不同的时候是不同的值）。例如，假定EMP中的值如图3-1所示，并假定删除Saito这一行（雇员号为E4）：

```
DELETE EMP WHERE EMP# = EMP# ('E4') ;
```

结果如图3-3所示。

从概念上说，以上就是EMP旧的关系值被一个新的关系值所代替。当然旧值（4行）和新值（3行）很相似，但是从概念上说，这两者是不同的值。确实，删除操作可以简化为一个**关系赋值**操作，如下：

EMP

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E1 | Lopez | D1 | 40K |
| E2 | Cheng | D1 | 42K |
| E3 | Finzi | D2 | 30K |

图3-3 删除E4行后的关系变量EMP

```
EMP  :=  EMP WHERE NOT ( EMP# = EMP# ('E4') ) ;
```

与所有的赋值操作一样，这就是先计算表达式的右边；然后把计算结果赋给左边的变量（当然，根据定义，左边必须是特定的变量）。如上所述，最后结果就是用一个“新”的EMP代替“旧”的EMP。（注意：原来的删除和相应的赋值语句都是用一种称为**Tutorial D**的语言来表示，DELETE操作和赋值操作都可以用这种语言表达。）

当然，通过类似的方式，关系INSERT和UPDATE操作也可以简化为特定的关系赋值操作。详见第6章。

目前，遗憾的是，许多文献用“关系”这个词时，表达的意思却是关系变量的含义（在表示关系时，其含义为关系值）。虽然这是历史造成的，但这必然导致了概念上的混淆。因此，贯穿本书，我们将从本质上仔细区分“关系”和“关系变量”这两个词。根据参考文献[3.3]，我们将使用“**relvar**”作为relational variables的简写，而且还要注意，当实际含义是关系变量时，用“relvar”而不用关系来表示[○]。请注意，从这里开始我们将使用没有限制的“关系”这个词表示一个特定的关系值（就像我们用“整数”这个词表示一个特定的整数值）。当然我们也会偶尔使用“关系值”这个词来强调。

注意：提醒大家，“relvar”一词并不常用——但应该常用！我们深切感到清楚地区分关系本

---

○ 关系值和关系变量的区别只是值和变量之间的区别的一个特例。我们将在第5章深入讨论这个区别。

身（即关系值）和关系变量是非常重要的（尽管本书的早期版本并未如此，且在这方面目前的大多数数据库文献也未能给予重视）。特别是，在第 6 章和第 9 章中，更新操作和完整性约束都是针对关系变量而不是关系。

## 3.4　关系的含义

第 1 章中提到关系中的列与**数据类型**（简称为类型，也称为域）有关。在 3.2 节末尾，指出关系模型包括“可扩充的［数据］类型集”。这意味着**用户可以定义自己的数据类型**（当然，这和使用系统定义的类型或内建的类型一样）。例如，可以定义如下类型（**Tutorial D** 语法；省略号“...”代表与目前所讨论的问题并不密切相关的定义部分）：

```
TYPE EMP# ... ;
TYPE NAME ... ;
TYPE DEPT# ... ;
TYPE MONEY ... ;
```

例如，类型 EMP#作为所有可能的雇员号的集合；类型 NAME 作为所有可能的名字的集合，等等。

图 3-4 基本上是图 3-1 的 EMP 部分（扩展了列的数据类型）。如图所示，每个关系——更确切地说，每个关系值——都有两部分，一组“列名：类型名”对（**列标题**），以及符合该标题的行集（**主体**）。注意：实际上，就像前面所举的例子那样，列标题的类型名部分经常省略，但是，要知道在概念上它们是存在的。

| EMP# : EMP# | ENAME : NAME | DEPT# : DEPT# | SALARY : MONEY |
|---|---|---|---|
| E1 | Lopez | D1 | 40K |
| E2 | Cheng | D1 | 42K |
| E3 | Finzi | D2 | 30K |
| E4 | Saito | D2 | 35K |

图 3-4　带有列类型的 EMP 关系值示例

对“关系”还有一种重要的认识方式。如下所述：

1）对于指定的关系 *r*，关系 *r* 的标题表示了一个特定的**谓词**（谓词就是一个带参数的真值函数）。

2）如第 1 章简要说明的，*r* 主体中的每一行都表示一个**真命题**，用相应类型的参数值代替该谓词的占位符或参数型来得到这个谓词（实例化为谓词）。

例如，在图 3-4 中，谓词如下：

*雇员 EMP#的名字是 ENAME，工作在部门 DEPT#，所得工资为 SALARY。*

（参数 EMP#、ENAME、DEPT#和 SALARY 对应于 EMP 表的四列）。相应的真命题是：

*雇员 E1 的名字是 Lopez，工作在部门 D1，所得工资为 40K。*

（通过用 EMP#值 E1，NAME 值 Lopez，DEPT#值 D1 和 MONEY 值 40K 替换相应的参数来得到）；

*雇员 E2 的名字是 Cheng，工作在部门 D1，所得工资为 42K。*

（通过用 EMP#值 E2，NAME 值 Cheng，DEPT#值 D1 和 MONEY 值 42K 替换相应的参数来得到）；等等。因此：

- **类型是所讨论的事物（的集合）。**
- **关系是关于所讨论的事物的事物（的集合）。**

（这里有一个类比可以帮助大家来理解和记住这些重要的结论：类型对于关系就像名词对于句子一样。）在上述例子中，我们所讨论的事物是雇员号、名字、部门号和货币值，以及事务，这些是真正的说话形式，即“指定雇员号的雇员有指定的名字，工作在指定的部门，得到指定的工资”。

从前面可以得到：

1）类型和关系都是必要的（没有类型，就没什么可讨论的；没有关系，也没什么可讨论的）。

2）类型和关系不仅是必需的，而且是充分的——即从逻辑上说，我们不再需要其他的东西。

3）类型和关系不是一回事。遗憾的是，某些商业产品——根据定义，不是关系的——常常将这一点混淆。在第26章会提到这个问题（见26.2节）。

顺便提及，要知道每个关系都有一个相联系的谓词，包括那些通过如连接关系操作得出的关系，这点也是比较重要的。例如，图3-1中的关系DEPT和图3-2中的三个结果关系有如下几个谓词：

- DEPT："部门DEPT#的名字为DNAME，预算为BUDGET"。
- 选择BUDGET>8M的DEPT："部门DEPT#的名字为DNAME，预算为BUDGET，预算大于800万美元"。
- 对DEPT的DEPT#和BUDGET投影："部门DEPT#有某名字，且预算为BUDGET"。
- DEPT和EMP根据DEPT#的连接："部门DEPT#的名字为DNAME，预算为BUDGET，而且雇员EMP#名为ENAME，在部门DEPT#工作，所得工资为SALARY"。注意，这一谓词有6个参数，而不是7个——两个DEPT#是同一个参数。

最后，我们注意到关系变量也有谓词，所有关系的谓词是关于关系变量的一些可能值。例如，关系变量EMP的谓词是：

名为ENAME，在部门DEPT#工作，薪水为SALARY的雇员EMP#。

## 3.5 优化

如3.2节所述，关系操作如选择、投影和连接等都是集合操作。因而，关系语言通常称为**非过程化语言**，因为用户指出是什么而不是怎么做——即他们只说想要什么，而不说出如何得到。为了满足用户需求而对存储数据的导航过程是由系统自动进行的，而不用用户手工处理。因此，有时称关系系统实现**自动导航**。相反，在非关系系统中，导航一般由用户负责。对自动导航的一系列优点的说明如图3-5所示，该说明对比了某SQL INSERT语句和用户在非关系系统（实际上是一个CODASYL网状系统；例子来自参考文献［1.5］的关于网状数据库的一章）中要实现相应操作所编写的"手工导航"代码。注意：这里的数据库是供应商和零件数据库，参见3.9节的进一步阐述。

```
INSERT INTO SP ( S#, P#, QTY )
       VALUES ( 'S4', 'P3', 1000 ) ;
```

```
MOVE 'S4' TO S# IN S
FIND CALC S
ACCEPT S-SP-ADDR FROM S-SP CURRENCY
FIND LAST SP WITHIN S-SP
while SP found PERFORM
   ACCEPT S-SP-ADDR FROM S-SP CURRENCY
   FIND OWNER WITHIN P-SP
   GET P
   IF P# IN P < 'P3'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN S-SP
END-PERFORM
MOVE 'P3' TO P# IN P
FIND CALC P
ACCEPT P-SP-ADDR FROM P-SP CURRENCY
FIND LAST SP WITHIN P-SP
while SP found PERFORM
   ACCEPT P-SP-ADDR FROM P-SP CURRENCY
   FIND OWNER WITHIN S-SP
   GET S
   IF S# IN S < 'S4'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN P-SP
END-PERFORM
MOVE 1000 TO QTY IN SP
FIND DB-KEY IS S-SP-ADDR
FIND DB-KEY IS P-SP-ADDR
STORE SP
CONNECT SP TO S-SP
CONNECT SP TO P-SP
```

图3-5 自动与手工导航

尽管前一段已经提到，这里还要强调一下，非过程化并不是一个让人非常满意的术语，因为过程化和非过程化并不是绝对的。最好的提法是某种语言*A*比另一种语言*B*多些或少些过程化。或许更好的表述方式为：与非关系语言相比，关系语言是在更高层次上的抽象（如图3-5所示）。本质上，关系语言是一种抽象层次的提升，这种抽象提高了关系系统的生产力。

如何实现上面提到的自动导航的功能是DBMS的一个重要组成部分——**优化器**（在第2章中简要介绍过）的职责。换句话说，对于

用户的每个关系请求，优化器的工作是选择一个更高效的方式执行这一请求。举个例子，我们假定用户发出如下请求（采用 **Tutorial D**）：

```
( EMP WHERE EMP# = EMP# ('E4') ) { SALARY }
```

说明：圆括号（“EMP WHERE ...”）中的式子表示选择关系变量 EMP 中满足 EMP#行为 E4 的当前值。在大括号（“SALARY”）中的列名对选择的结果根据 SALARY 列进行投影。投影的结果是一个包括雇员 E4 的工资的单行、单列的关系。（注意，例子中显然利用了关系闭包的性质——其中写了一个嵌套的关系表达式，其中选择的输出作为投影的输入。）

目前，即使在很简单的例子中，也可能至少有两种执行数据访问的方式：

1）通过对关系变量 EMP 依物理顺序扫描，直到找到所要的数据；

2）如果在 EMP#列有索引——EMP#的值假定是唯一的，因为为了确保唯一性，许多系统实际上需要索引——那么，通过使用索引直接找到所要的数据。

优化器将选择采取其中一种策略。更一般地，对于任何一个给定的请求，优化器根据如下考虑来选择执行请求的策略：

- 请求中参照了哪些关系变量
- 关系变量当前的数据规模
- 存在什么索引
- 索引的选择性如何
- 在磁盘上数据是怎样物理聚集的
- 涉及什么样的关系操作

等等。因此，用户只需说明他们想要什么，而不用管怎样得到数据；获得这些数据的访问策略是优化器来选择的（“自动导航”）。用户和用户应用程序独立于这些访问策略，如果要得到物理数据的独立性，这一点显然是很重要的。

第 18 章将详细阐述优化器的功能。

## 3.6　数据字典

第 2 章中已经提到，每个 DBMS 必须提供**目录**或**字典**功能。字典存储了各种模式（外模式、概念模式和内模式）和相应的映像（外模式/概念模式的映像，概念模式/内模式的映像）。换句话说，字典包括了对系统自身有用的各种对象的细节信息（有时称作描述信息或元数据）。这些对象包括关系变量、索引、用户、完整性约束、安全性约束，等等。描述信息对确保系统正确工作很重要。例如，优化器利用索引和其他物理存储结构的字典信息，以及其他信息来帮助决定怎样实现用户的请求（参考第 18 章）。同样地，授权子系统首先利用用户和安全性约束的字典信息来准许或拒绝这些请求（参考第 17 章）。

关系系统的优点之一是，在系统中字典本身就是由关系变量（更精确地说应为系统关系变量，这样称呼是为区别于普通用户的关系变量）组成的。因此，用户能够像访问自己的数据一样访问字典。例如，SQL 系统中典型的字典包括 TABLE 和 COLUMN 两种系统关系变量，分别描述了数据库中的表（即关系变量）和表中的列。对于图 3-1 中的部门和雇员数据库，TABLE 和 COLUMN 关系变量如图 3-6 中所示[○]。

注意：在第 2 章中提到，字典通常应该是自描述的，即它包括描述字典关系变量自身的条目。见本章末尾的练习 3.3。

假定部门和雇员数据库的一些用户想要确切地知道关系变量 DEPT 包含哪些列（假定由于某种原因，用户不知道该信息）。那么用如下表达式可实现此任务：

```
( COLUMN WHERE TABNAME = 'DEPT' ) { COLNAME }
```

---

○ 注意在图 3-6 中的 ROWCOUNT 列，对数据库的插入和删除操作同时也更新了数据字典。在实际操作时，ROWCOUNT 可能按照用户的要求被更新，因此这一列的值未必总是当前的。

下面是另一个例子："哪一个关系变量包含了名为 EMP#的列？"

```
( COLUMN WHERE COLNAME = 'EMP#' ) { TABNAME }
```

练习：下面语句的执行结果是什么？

```
( ( TABLE JOIN COLUMN )
        WHERE COLCOUNT < 5 ) { TABNAME, COLNAME }
```

TABLE

| TABNAME | COLCOUNT | ROWCOUNT | ..... |
|---|---|---|---|
| DEPT | 3 | 3 | ..... |
| EMP | 4 | 4 | ..... |
| ..... | ..... | ..... | ..... |

COLUMN

| TABNAME | COLNAME | ..... |
|---|---|---|
| DEPT | DEPT# | ..... |
| DEPT | DNAME | ..... |
| DEPT | BUDGET | ..... |
| EMP | EMP# | ..... |
| EMP | ENAME | ..... |
| EMP | DEPT# | ..... |
| EMP | SALARY | ..... |
| ..... | ..... | ..... |

图 3-6 部门和雇员数据库的字典（略图）

## 3.7 基本关系变量和视图

我们已知，从一组关系变量如 DEPT 和 EMP，以及一组关系变量的关系值开始，通过关系表达式可以从指定的关系值来得到其他的关系值。我们再介绍一些术语，源（指定的）关系变量称为**基本关系变量**，而它们的关系值称为**基本关系**；通过某关系表达式从基本关系中得出的、或能够得出的关系称为**导出关系**或**可导出关系**。注：在参考文献［3.3］中，基本关系变量又称为实关系变量。

首先，关系系统必须提供创建基本关系变量的方法。例如，在 SQL 中，这一任务是由 CREATE TABLE 语句来执行的（很明显，这里的"TABLE"是基本关系变量，或 SQL 中的基本表）。显然，基本关系变量必须是已命名的——例如：

```
CREATE TABLE EMP ... ;
```

但是，关系系统通常也支持另一种命名的关系变量——**视图**，在任何指定的时候，它的值都是导出的关系（因此，视图也可以当作**导出的关系变量**）。在指定时刻，指定视图的值是当时关系表达式的计算结果；关系表达式在创建视图时是确定的。例如，语句

```
CREATE VIEW TOPEMP AS
     ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } ;
```

可用于定义名为 TOPEMP 的视图。（注：为了方便，例子用 SQL 和 **Tutorial D** 的混合形式来表示。）

当语句执行时，AS 之后的关系表达式——**视图定义表达式**——没有计算，只是系统以某种方式保存下来（实际上，把它保存在字典中，在名字 TOPEMP 之下）。但是，对用户来说，好像在数据库中真正存在名为 TOPEMP 的关系变量，而且在图 3-7 中的非阴影部分是它的当前值。用户能够像操纵基本关系一样操纵视图。注：如果

TOPEMP

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E1 | Lopez | D1 | 40K |
| E2 | Cheng | D1 | 42K |
| E3 | Finzi | D2 | 30K |
| E4 | Saito | D2 | 35K |

图 3-7 EMP 的视图 TOPEMP（非阴影部分）

（如前所述）DEPT 和 EMP 被当作实关系变量，则 TOPEMP 就被当作虚关系变量——即关系变量看起来以自身的形式存在，但实际上并不是（在任何特定时候，它的值依赖于其他特定的关系变量）。参见［3.3］。

注意：尽管说 TOPEMP 的值是关系，该关系是计算视图定义表达式而得到的结果，但是它并不表示现在有了一份独立的数据副本；也就是说，不表示真的计算了视图定义表达式。相反，视图只是它所依赖的基本关系变量 EMP 的一个窗口。结果，任何基本关系变量的改变都可通过窗口自动地看到（当然，假定改变在非阴影部分）。同样，TOPEMP 的改变也会自动地同时反映到关系变量 EMP 上（见后面的例子）。

以下是一个对视图 TOPEMP 的检索操作的例子：

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

对图 3-7 中的样本数据，结果如下：

| EMP# | SALARY |
|---|---|
| E1 | 40K |
| E4 | 35K |

从概念上讲，对视图的操作（如前面提到的检索操作）是通过替换视图名所对应的视图定义表达式（保存在目录中的表达式）来处理的。因此，例子中的原表达式

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

被系统替换为

```
( ( ( EMP WHERE SALARY > 33K ) { EMP#, ENAME, SALARY } )
          WHERE SALARY < 42K ) { EMP#, SALARY }
```

（把原表达式中的视图名用斜体表示，改变的代替部分也用斜体表示），替换后的表达式可以简化为

```
( EMP WHERE SALARY > 33K AND SALARY < 42K ) { EMP#, SALARY }
```

（见第 18 章），对此计算后就得到前面的结果。换句话说，对视图的操作其实是转变为对基本关系变量的等价操作，等价操作以正常的方式执行（更确切地说，以正常的方式优化并执行）。

再举另一个例子，考虑如下的 DELETE 操作：

```
DELETE TOPEMP WHERE SALARY < 42K ;
```

实际执行的 DELETE 是如下的：

```
DELETE EMP WHERE SALARY > 33K AND SALARY < 42K ;
```

当前，视图 TOPEMP 非常简单，只包含了一个基本关系变量的一个行列子集。但是，理论上，视图定义可以是任意复杂的（甚至可以参照其他视图），因为实质上它只是一个命名的关系表达式。例如，下面的视图涉及了两个基本关系变量的连接：

```
CREATE VIEW JOINEX AS
   ( ( EMP JOIN DEPT ) WHERE BUDGET > 7M ) { EMP#, DEPT# } ;
```

我们将在第 10 章详细讨论视图定义和处理的问题。

在此，我们顺便解释一下 2.2 节末尾的评论，大意是“视图”在关系范畴中有一个特定的含义，和在 ANSI/SPARC 体系结构中所述的含义不尽相同。在该体系的外模式，通过外部视图观察数据库，由外模式来定义外部视图（不同的用户有不同的外部视图）。相反，在关系系统中，视图是命名的、导出的虚关系变量。ANSI/SPARC 的外部视图类似于几个关系变量的集合，其中每个都是关系意义下的一个视图，“外模式”包含那些视图的定义。（可以看出，在关系意义

下的视图是一种提供**逻辑数据独立性**的关系模型的方式，尽管目前的商业产品在这方面还有欠缺。见第10章。）

目前，ANSI/SPARC体系结构已经很普通了，并且允许外模式和概念模式之间的任意可变性。理论上，甚至两层所支持的数据结构的类型都可能是不同的——例如，概念模式可能是关系的，而用户可以有一个层次的外部视图[⊖]。但是，实际上，许多系统使用同一种数据结构类型作为两层的基础，关系产品也不例外——就像基本关系变量一样，视图仍然是关系变量。既然两层支持同样的对象类型，两层也就支持同样的数据子语言（通常为SQL）。视图是关系变量，这一事实是关系系统的扩展之一；这就像数学中子集是集合一样重要。注意：SQL产品和SQL标准（见第4章）好像对用词有一点误解，比如在SQL中经常说“表和视图”（暗示视图不是表）。建议大家不要掉入这一陷阱，不要把“表”（或关系变量）只理解为基本表（或关系变量）。

最后一点需要指出的是基本关系变量和视图的问题。两者的不同在于：

- 基本关系变量是“真实存在”的，意味着它们所表示的数据真正存在于数据库中。
- 相反，视图并不“真实存在”，只是提供了观察“真实数据”的各种方式。

尽管这在非正式情况下可能很有用，但是，这一特征并不能准确反映事情的真正本质。确实，用户可以把基本关系变量理解为物理存储的关系变量；但事实上，在某种程度上，关系系统的本意是允许用户把基本关系变量认为是物理上存在的，而不用考虑在物理上是怎样表示的。但这不能解释为：基本关系变量是以任意一种直接的方式（如，作为一个存储文件）进行存储的。如在3.2节提及的，基本关系变量最好被当作是存储数据集的抽象——即已经掩盖了存储细节的抽象。理论上，基本关系变量和它相应的存储之间可以存在任何程度的差别。[⊜]

用一个简单的例子来澄清这一点。再看部门和雇员数据库，目前的大多数关系系统可能用两个存储文件实现数据库，每个文件都用于一个基本关系变量。但是，对为什么不用一个存储记录文件，这其中绝对没有逻辑上的原因。每个存储记录包含（a）每个部门有一个部门号、名称和预算；（b）部门中的每个雇员有雇员号、姓名和工资。换句话说，只要数据是正确的，就可以在物理上以任何方式存储（见附录A中更进一步的讨论），但在逻辑层上看起来总是一样的。

## 3.8 事务

注意：本节的主题并不只针对关系系统，为了理解第二部分有关的内容，必须先理解一些基本概念。但是，这里只作简单地介绍。

第1章中指出，事务是一个逻辑工作单元，通常包括几个数据库操作，并指出当不同的操作处于同一事务中时，用户要通知系统。BEGIN TRANSACTION、COMMIT和ROLLBACK操作就是基于这一考虑提出的。基本上，当BEGIN TRANSACTION操作执行时，表示一个事务的开始，当COMMIT或ROLLBACK操作执行时，表示一个事务的终止。例如（伪码）：

```
BEGIN TRANSACTION ; /* move $$$ from account A to account B */
UPDATE account A ;                          /* withdrawal   */
UPDATE account B ;                          /* deposit      */
IF everything worked fine
   THEN COMMIT ;                            /* normal end */
   ELSE ROLLBACK ;                          /* abnormal end */
END IF ;
```

注意以下几点问题：

1）要保证事务的**原子性**，也就是说，即使系统在处理中发生故障，也要保证（从逻辑的观

---

⊖ 我们将在第27章看到例子。

⊜ 下面引用一本最近出版的书中的话来展示这一段中所讨论的冲突问题：在前面的3.3节中“区分存储的关系是表，虚关系还是视图是很重要的。我们只有在用到一个表或者视图时使用关系这个词，当强调关系是存储关系而不是视图时，我们会用术语基本表或基本关系。”这段引用很典型。

点）事务中的操作要么都做，要么都不做。[⊖]

2）要保证事务的**持续性**，一旦事务成功地执行了 COMMIT，即使随后系统发生故障，也要确保它的更新写入数据库中。注意：本质上，是事务的持续性保证了数据库中数据的持久性。

3）要保证事务的**隔离性**，事务 T1 对数据库的更新操作对任何不同的事务 T2 来说是不可见的，直到或除非 T1 成功执行 COMMIT。COMMIT 使事务的更新对其他事务来说是可见的；这些更新已经提交，并且要保证不能取消。若事务执行了 ROLLBACK，所有事务的更新操作都要取消（回滚）。对后一种情况，其结果应该就像事务一开始就没有执行一样。

4）要保证一组并发事务的交叉执行（通常）是**可串行的**，即其结果与按某一未指明的次序串行地执行事务时的结果相同。

对上述几点和许多其他方面的深入讨论参见第 15 章和第 16 章。

## 3.9　供应商和零件数据库

整本书使用的大多数例子均基于著名的**供应商和零件**数据库。本节主要是解释这个数据库，为后续章节引用提供方便。图 3-8 给出了一个样本数据值的集合；当后面做改动时，后面所举的例子实际上基于这些值。[⊖] 图 3-9 给出了数据库定义，是用 **Tutorial D** 来表示的（Tutorial D 关键字 VAR 表示“变量”）。尤其注意主码和外码的说明。注意：(a) 几个列都有与所提到的列同名的数据类型；(b) STATUS 列和两个 CITY 列都是用内部类型定义的——INTEGER（整数）和 CHAR（变长字符串）——而不是用户定义的。最后要注意的很重要的一点是关于图3-8中所示的列值，在此还不能说明这一点，留待 5.3 节来加以阐述。

S

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

SP

| S# | P# | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S1 | P3 | 400 |
| S1 | P4 | 200 |
| S1 | P5 | 100 |
| S1 | P6 | 100 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S4 | P4 | 300 |
| S4 | P5 | 400 |

P

| P# | PNAME | COLOR | WEIGHT | CITY |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P2 | Bolt | Green | 17.0 | Paris |
| P3 | Screw | Blue | 17.0 | Oslo |
| P4 | Screw | Red | 14.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |
| P6 | Cog | Red | 19.0 | London |

图 3-8　供应商和零件数据库（样本值）

以上数据库的含义如下：

- 关系变量 S 代表供应商（更确切地说，合同供应商），每个供应商有一个供应商号(S#)，对供应商来说这是唯一的；一个供应商名（SNAME），不必唯一（尽管 SNAME 值在图 3-8中是唯一的）；定额或状态值（STATUS）；和一个场所（CITY）。假定一个供应商只住在一个城市。
- 关系变量 P 代表零件（更精确地说，零件的种类），每种零件有一个零件号（P#）是唯一的；零件名（PNAME）；颜色（COLOR）；重量（WEIGHT）；该种零件储存的地点

---

⊖ 一个事务是一段代码的执行，而诸如“一个事务的执行”这样的短语实际上是错误的（它表示的是一个执行的执行）。尽管如此，这样的措辞是普遍而有用的。因为没有更好的表达，所以在本书中我们也将这样用。

⊖ 为了方便引用，这里将图 3-8 重复一次。可能读者已经从早期的几版书中熟悉了示例数据，我们需要指出零件 P3 已经由过去的 Rome 改为 Oslo。在下一章的 4.5 节也作了同样的改动。

(CITY)。假定——在通常情况下——零件的重量单位是磅（参见5.4节对度量单位的讨论）。假定每种零件只有一种颜色，恰好只存在一个城市的仓库中。

- 关系变量SP代表发货量。逻辑上，它的含义是将另外两个关系变量连接起来。例如，图3-8中SP的首行连接了表S（供应商S1）的一个特定的供应商和表P（零件P1）的一种特定的零件——换句话说，表示供应商S1供应了P1这种零件（供应数量为300）。这样，每个发货量就有一个供应商号（S#）、零件号（P#）和数量（QTY）。假定在任何指定的时刻，对特定的供应商和特定的零件至多只有一次供货。因此，对于每次供货，对于目前在SP中出现的S#值和P#值的组合是唯一的。注意：图3-8中的样本值特意包括了一个供应商S5，没有供货与之对应。

```
TYPE S# ... ;
TYPE NAME ... ;
TYPE P# ... ;
TYPE COLOR ... ;
TYPE WEIGHT ... ;
TYPE QTY ... ;

VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR
    PRIMARY KEY { S# } ;

VAR P BASE RELATION
    { P#     P#,
      PNAME  NAME,
      COLOR  COLOR,
      WEIGHT WEIGHT,
      CITY   CHAR }
    PRIMARY KEY { P# } ;

VAR SP BASE RELATION
     { S#     S#,
       P#     P#,
       QTY    QTY }
     PRIMARY KEY { S#, P# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P ;
```

图3-9 供应商和零件数据库（数据定义）

注意：1.3节已经指出，供应商和零件可以当作**实体**，发货量可以当作一个指定的供应商和指定的零件之间的**联系**。该节还指出，联系最好当作实体的一个特例。关系数据库的优点是，无论实际上这些实体是否为联系，所有实体都以同样的形式表示——即，例子中显示的关系中的行。

最后注意以下几点：

- 首先，尽管供应商和零件数据库非常简单，可能比任何实际数据库都简单得多；绝大多数实际的数据库都包含比这个例子多很多的实体和联系（而且是多很多种实体和联系）。但不管怎样，该例至少可以恰当地说明本书后面所要得出的结论。在下面几章，大多数例子还将使用它。
- 其次，可以使用像SUPPLIERS、PARTS和SHIPMENT这样描述性的名字代替上面使用的S、P、SP这些简写的名字；实际中推荐使用描述性的名字。但对于供应商和零件数据库这一特定的例子，因为要频繁使用，所以使用了简名。多次重复长名字会令人厌烦。

## 3.10 小结

本节简要回顾一下关系技术。显然，对这样一个含义广泛的概念，目前只涉及了一些表面的内容，本章的主旨是为进一步理解做一些简要介绍。尽管如此，我们还是设法尽可能包括更多方面的内容。下面将谈到的主要内容小结一下。

从用户的观点来看，**关系数据库**是**关系变量**或**表**的集合。**关系系统**是支持关系数据库及其操作的系统，特别是**选择**、**投影**和**连接**操作。这些操作和其他操作都是集合操作，这些操作的集合形成**关系代数**。[⊖] 关系系统的**封闭性**是指每种操作的输出都和其输入是相同类型的（都是关系的），这就表示可以写**嵌套的关系表达式**。关系变量可以通过**关系赋值**操作来更新；类似地，**INSERT**、**UPDATE**和**DELETE**更新操作也可以看作某些一般关系赋值操作的缩写。

关系系统所依据的形式理论是**关系数据模型**。关系模型只涉及逻辑的内容，而不涉及物理的内容，它包括数据的三方面理论——即数据**结构**、数据**完整性**和数据**操作**。结构化方面与关系本身有关；完整性方面与**主码**和**外码**有关；操作方面与操作符（选择、投影和连接等）有关。**信**

⊖ 在3.2节中我们给出关系模型的正规定义时提到过这个词。而这个词从第6章开始我们才会正式使用。

**息原则**（现在看来更好的说法是统一表示原则）表明是以一种而且只以一种方式表示关系数据库的整个内容，即以关系中的行和列交叉位置的明确的值表示。等价地：关系数据库允许的唯一变量是关系变量。

每个关系都含有一个**标题**和一个**主体**；标题是“列名：类型名”对的集合，主体是对应标题的行集。一个指定关系的标题可以当作**谓词**；主体中的每一行表示一个**真命题**，用**参数**的值代替占位符或谓词参数的适当类型来得到这个真命题。换句话说，类型是我们所研究的对象（的集合）；关系是关于我们所研究的对象的内容（的集合）。类型和关系对于我们所要表示的数据（逻辑上的）来说是**充分而必要**的。

**优化器**是系统的组成部分，它决定怎样执行用户请求（用户只需考虑要什么，而不必考虑怎么做）。因为关系系统能够负责导航定位所要的数据库中的数据，所以有时被称为**自动导航系统**。优化和自动导航是保持**数据物理独立性**的先决条件。

**字典**是一组系统关系变量，它包括了关于对数据库有用的各种条目的细节信息（基本关系变量、视图、索引和用户，等等）。用户能够像访问自己的数据一样访问字典。

在指定的数据库中，源关系变量称为**基本关系变量**，而它们的值称为**基本关系**；通过关系表达式从基本关系中得出的关系称为**导出关系**（基本关系和导出关系被称为**可表示的**关系）。**视图**是一种关系变量，它的值在任何指定的时刻是一个导出的关系（**粗略地讲**，可以认为视图是**导出的关系变量**）；在指定的时刻，这一关系变量的值是从相应的**视图定义表达式**计算得到的。因此，基本关系变量是独立存在的，但视图不是——它们依赖于相应的基本关系变量（另一种说法是基本关系变量是自治的，而视图不是）。用户能够像操纵基本关系变量一样操纵视图（至少理论上是）。系统是通过替换视图所对应的视图定义表达式来执行视图上的操作的。因此，对视图的操作就转变为对基本关系变量的等价操作。

**事务**是一个逻辑工作单元，通常包括几个数据库操作。当 **BEGIN TRANSACTION** 执行时，一个事务开始；当 **COMMIT**（正常终止）或 **ROLLBACK**（非正常终止）执行时，事务终止。事务是**原子的**、**持续的**且与其他事务相**隔离**。一组并发事务的交叉执行（通常）要保证是**可串行的**。

最后，本书中基本的例子是**供应商 - 零件数据库**。如果大家还没有熟悉这个例子，现在得多花点时间来熟悉它；至少应该知道哪个关系变量有哪些列，以及主码和外码是什么（不必确切地知道样本数据值）。

## 习题

3.1　定义下列术语：

| | | | |
|---|---|---|---|
| 自动导航 | 基本关系变量 | 日志 | 闭包 |
| 提交 | 派生关系变量 | 外码 | 连接 |
| 优化 | 谓词 | 主码 | 投影 |
| 命题 | 关系数据库 | 关系 DBMS | 关系模型 |
| 选择 | 回滚 | 集合操作 | 视图 |

3.2　描述供应商和零件数据库的字典关系变量 TABLE 和 COLUMN 的内容。

3.3　在 3.6 节中提到，字典是自描述的——也就是说，包括字典关系变量自身的条目。扩展图 3-6，以包括关系变量 TABLE 和 COLUMN 的必要条目。

3.4　这是一个对供应商和零件数据库的查询。它是做什么的？谓词的结果是什么？

```
( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY }
```

3.5　假定习题 3.4 中的赋值表达式使用视图定义：

```
CREATE VIEW V AS
   ( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY } ;
```

考虑查询

```
( V WHERE CITY = 'London' ) { S# }
```

该查询做了什么？谓词的结果是什么？DBMS 的哪个部分处理这个查询？

3.6　你如何理解（事务）的原子性、持续性、隔离性和可串行性？

3.7　陈述“信息原则”。

3.8　如果你了解层次数据模型，请你简要描述一下它与本章介绍的关系数据模型的区别。

## 参考文献

[3.1] E. F. Codd:“Relational Database : A Practical Foundation For Productivity,” *CACM 25*,No. 2 (February 1982). Republished in Robert L. Ashenhurst (ed.),*ACM Turing Award Lectures: The First Twenty Years 1966 – 1985*. Reading,Mass.: Addison-Wesley (*ACM Press Anthology Series*,1987).

这篇论文使 Codd 获得 1981 年 ACM 图灵奖，该奖项是对他在关系模型方面的工作给予的肯定。该文指出了著名的应用程序黑箱问题。引文：“计算机应用程序的需求发展迅速——如此迅速以致于信息系统部门（负责提供这些应用程序）的能力越来越落后于需求。”对这一问题有两种弥补办法：

1）给 IT 专业人员提供新工具以提高其生产力。

2）允许最终用户直接与数据库交互，这样完全越过了 IT 专业人员。

以上两种方法都需要，在本文中 Codd 证明关系理论能向两者提供必要的基础。

[3.2] C. J. Date:“Why Relational?” in *Relational Database Writings 1985 – 1989*. Reading,Mass.:Addison-Wesley(1990).

试图对关系系统的主要优点做简洁明了的概括。本文的下列观点值得强调：在“关系化”(going relational) 的各种优点中，有一点是再强调也不过分的，即一个合理理论基础的存在。引文：“……关系的（relational）确实与众不同。不同之处是因为它并不特殊。相反，旧的系统是特殊的；它们可以解决当时特定的重要问题，但是它们没有坚实的理论基础。而关系的确基于这样一个基础……这意味着它是坚实的……感谢这一坚实的基础，关系系统以良好定义的方式工作；并且（人们也许没有意识到这一事实）用户在头脑中有一个那种状态的简单模型，这使他们有信心预见在特定情况下系统的处理结果。这没有什么可惊讶的。这种预见能力意味着用户界面易于理解、存档、教学、学习、使用和记忆。”

[3.3] C. J. Date and Hugh Darwen:*Foundation for Future Database Systems: The Third Manifesto* (2nd edition). Reading, Mass.: Addison- Wesley (2000). See also *http://www. thethirdmanifesto. com*, which contains certain formal extracts from the book,an errata list,and much other relevant material. Reference [20.1] is also relevant.

《对象关系数据库基础：第 3 版宣言》是对数据库和 DBMS 未来方向的一个详尽的、形式化的和严密的论述。它可以被看作 DBMS 和其语言界面设计的一个抽象蓝图。它基于传统的核心概念——**类型、值、变量和操作符**。例如，有一个 INTEGER 类型；整数“3”是该类型的一个值；N 是该类型的一个变量，其值在特定的时刻是某整数值；“+”是应用于整数值（即该类型的值）的操作符。注意：对于类型的强调就来自于这本书的子标题“A Detailed Study of the Impact of Type Theory on the Relational Model of Data, Including a Comprehensive Model of Type Inheritance”。类型的理论和关系模型基本是相互独立的。更具体地说，关系模型并不支持特定的类型（除了布尔型）；它只认为关系的属性应该是某个类型的，即它支持某个并不具体的类型。

关系变量这个词来自于这本书。书中有这样的描述：“*Manifesto*（宣言）的第一版本中给出了数据库值和数据库变量的区别，它类似于关系值和关系变量的区别。同样的，引入 dbvar 作为数据库变量（database variable）的缩写，虽然这个区别是存在的，但它与其他的部分没有太多的直接联系。因此我们决定，为了易用性，我们还是使用更加传统的术语。”而这个决定后来证实并不好。引用 [23.4] 中的内容：“如果有先见，采用逻辑上更为正确的词数据库值和数据库变量将会更好。”在现在的这本书中，我们仍然使用熟悉的词“数据库”，但是这么做和我们的看法是不吻合的。

还有一点，像这本书中所说的：“我们承认给一份技术性的文档取名宣言的确有些不妥。在《Chambers 20 世纪字典》中，宣言的含义是：一种表明一个人或者一个组织（例如一个政党）的意图、观点或者动机的手写声明。而第三次宣言实际上关于科学和逻辑，而不仅是意图、观点或动机。但是第三次宣言的确是为了对比前两者——《面向对象数据库系统宣言》[20.2, 25.1] 和《第三代数据库系统宣言》[26.44] 而特别撰写的。因此我们选择了这个标题。”

[3.4] C. J. Date: "Great News, The Relational Model Is Very Much Alive!," *http://www.dbdebunk.com* (August 2000).

自从在1969年首次出现，关系模型就承受一大批不同的作者的攻击。一个最近的例子，"Great News, The Relational Model Is Dead!" 这篇文章就是反驳这种观点。

[3.5] C. J. Date: "There's Only One Relational Model!" *http://www.dbdebunk.com* (February 2001).

自从在1969年首次出现，关系模型便被很多作者误传。一个最近的例子是一本书的名为 "Different Relational Models" 的一章。这章的第一句话是："数据库里并没有关系模型这回事，就像它没有一门几何学一样。" 这篇文章就是反驳这种观点。

# 第4章 SQL 简 介

## 4.1 引言

正如第1章中所述，SQL是关系数据库的标准语言，今天市场上的任何数据库产品几乎都支持SQL。SQL最初于20世纪70年代早期在IBM Research开发出来［4.9，4.10］；其大部分内容首先在IBM的System R原型系统中实现［4.1～4.3，4.12～4.14］，随后又在IBM公司的其他产品和其他公司的产品中重新实现［4.8，4.14，4.21］。本章我们将概要介绍SQL语言的主要特性；更多方面的细节——维护完整性、确保安全性等——将延缓至后面关注这些问题的章节介绍。在讨论的过程中，除非明确说明，SQL将指代当前版本的SQL标准（即**SQL：1999**）[⊖]。参考文献［4.23］是正式的SQL：1999规范；［4.24］是该规范的扩展的修正集。

注意，前一个SQL标准版本是SQL：1992，人们有意使得SQL：1999标准作为前一个版本的扩展，并与之兼容。然而，我们仅仅能清楚指出的是，现在甚至没有数据库产品可以完整地支持SQL：1992标准；不过，数据库产品通常支持被称为“子集的超集”的SQL标准（SQL：1999或者更可能的，SQL：1992）。具体地讲，大部分的数据库产品，不能支持标准中的某些方面，在其他方面又走得太远[⊖]。例如，IBM公司的DB2并不完全支持标准完整性特性，但却支持标准中不支持的重命名基本表的操作符。

一些附加的基本注意事项：

- SQL原先是作为特殊的“数据子语言”出现的。然而，随着**SQL持久存储模块**（SQL Persistent Stored Modules，SQL/PSM，简称PSM）在1996年成为了标准，SQL已经变成了计算上完全（computationally complete）的语言——它现在包括语句，例如CALL、RETURN、SET、CASE、IF、LOOP、LEAVE、WHILE和REPEAT，还包含其他一些相关的语言特征，比如可声明变量和异常处理。对PSM的详细讨论超出了本书的范畴，但你可以在［4.20］中找到一份不错的入门介绍。
- SQL使用术语**表**（table）来表示术语关系（relation）和关系变量（relvar）；对应地，使用术语**行**（row）和**列**（column）表示术语元组（tuple）和**属性**（attribute）。因此为了与SQL标准和SQL产品一致，在本章中也做同样的处理，对本书其他涉及SQL的部分，都做此处理。
- SQL是一个庞大的语言。它的规范文档［4.23］超过2000页——不计［4.24］中300多页的勘误表。因此，本书无法全面阐述SQL标准，也不可能做到，我们也从来没有试图如此；我们合理地忽略了很多标准特性，对其他的则做了简化。
- 最后不得不说明，如第1～3章中很多地方已经指出的一样，SQL远远不是完美的关系语言——它被自身的繁琐冗长和承载的期望和责任这些缺陷所拖累。尽管如此，它是标准，市场上差不多每一种产品都支持它，且数据库的专业人员也需要了解它。而这就是本书所涵盖的（关于SQL的）内容。

## 4.2 SQL基本操作

SQL兼有数据定义和数据操纵的功能。首先考虑定义操作。图4-1为供应商和零件数据库的

---

⊖ 新的SQL标准（SQL：2003）预计将在2003年晚些时候颁布；有时候我们也将提到这个版本，这时会特别说明。

⊖ 事实上，之所以没有数据库产品能完整地支持标准，是因为标准中还存在着缺陷、错误和不一致，请参考［4.23］和［4.24］。［4.20］中包含了对于SQL：1992存在的问题的更详细的讨论。

SQL 定义，请对照第 3 章的图 3-9。在这个定义中你可以发现，6 个用户定义类型（user-defined type，UDT）各包含了一条 **CREATE TYPE** 语句，3 个基本表各包含了一条 **CREATE TABLE**语句（如第 3 章中所指，CREATE TABLE 语句中的关键字 TABLE 专指基本表）。每条 CREATE TABLE 语句定义出基本表的名字、列的名字和类型、主码和外码（可能还有其他的附加信息，但是图 4-1 中没有描绘）。请注意如下要点：

- 举例时，我们经常在类型名和列名中使用字符“#”，但是事实上该字符在标准中是不合法的。
- 我们使用分号“;”作为语句的结束符。但是 SQL 是否使用该符号作为结束符要依情况而定。具体的细节问题超出了本书的范围。
- SQL 内建类型 CHAR 使用时需要指明长度——图中是 15。

定义了数据库之后，就可以通过 SQL 语句**操纵**数据，包括 SELECT、INSERT、UPDATE 和 DELETE。使用 SELECT 语句可以完成关系的选择、投影和连接操作。图 4-2 给出一些示例。注意，图中连接操作的例子中描绘了**点限定的列名**（如 S. S#，SP. S#），其目的是为了在 SQL 中避免列引用的歧义。一般规则是——尽管存在例外——限定名总是可接受的，而非限定名只有在不造成歧义时才可接受。

```
CREATE TYPE S# ... ;
CREATE TYPE NAME ... ;
CREATE TYPE P# ... ;
CREATE TYPE COLOR ... ;
CREATE TYPE WEIGHT ... ;
CREATE TYPE QTY ... ;

CREATE TABLE S
     ( S#     S#,
       SNAME  NAME,
       STATUS INTEGER,
       CITY   CHAR(15),
     PRIMARY KEY ( S# ) ) ;

CREATE TABLE P
     ( P#     P#,
       PNAME  NAME,
       COLOR  COLOR,
       WEIGHT WEIGHT,
       CITY   CHAR(15),
     PRIMARY KEY ( P# ) ) ;

CREATE TABLE SP
     ( S#     S#,
       P#     P#
       QTY    QTY,
     PRIMARY KEY ( S#, P# ),
     FOREIGN KEY ( S# ) REFERENCES S,
     FOREIGN KEY ( P# ) REFERENCES P ;
```

图 4-1　供应商和零件数据库（SQL 定义）

*Restrict:*　　*Result:*

```
SELECT S#, P#, QTY
FROM   SP
WHERE  QTY < QTY( 150 ) ;
```

| S# | P# | QTY |
|---|---|---|
| S1 | P5 | 100 |
| S1 | P6 | 100 |

*Project:*　　*Result:*

```
SELECT S#, CITY
FROM   S ;
```

| S# | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

*Join:*

```
SELECT S.S#, SNAME, STATUS, CITY, P#, QTY
FROM   S, SP
WHERE  S.S# = SP.S# ;
```

*Result:*

| S# | SNAME | STATUS | CITY | P# | QTY |
|---|---|---|---|---|---|
| S1 | Smith | 20 | London | P1 | 300 |
| S1 | Smith | 20 | London | P2 | 200 |
| S1 | Smith | 20 | London | P3 | 400 |
| .. | ..... | .. | ..... | .. | ... |
| S4 | Clark | 20 | London | P5 | 400 |

图 4-2　SQL 中的选择、投影和连接的例子

如下例所示，SQL 还支持 SELECT 子句的简记方式：

```
SELECT *                        /* or "SELECT S.*" (i.e., the */
FROM   S ;                      /* "*" can be dot-qualified)  */
```

结果就是 S 表的整个副本；星号表示列出 FROM 子句中表的所有列或用逗号隔开的列表，见 4.6 节中的形式化解释，并且按照在表中定义的从左往右的顺序列出；对于 FROM 子句中的其他表，可以以此类推。注意：表达式 SELECT * FROM *T*（其中 *T* 是一个表的名字）在 SQL 中可进一步简写为 TABLE *T*。

第 8 章（8.6 节）中将对 SELECT 语句进行更详尽的介绍。

现在介绍**更新**操作：第 1 章已经给出了 SQL 中的 INSERT、UPDATE 和 DELETE 语句的例子，但是该例却仅仅是对单行的操作。然而，一般来说，INSERT、UPDATE 及 DELETE 和 SELECT 语句一样，都是集合操作运算符，这在第 1 章的习题和答案中也可以看出。下面给出几个

在供应商和零件数据库上进行集合更新操作的例子：

```
INSERT
INTO    TEMP ( P#, WEIGHT )
        SELECT P#, WEIGHT
        FROM   P
        WHERE  COLOR = COLOR ('Red') ;
```

这个例子假定已经创建了一个名为 TEMP 的表，该表有两列：P#和 WEIGHT。INSERT 语句在零件表中插入所有的红色零件的标识号以及与标识号相对应的零件的重量。

```
DELETE
FROM    SP
WHERE   P# = P# ('P2') ;
```

这个 DELETE 语句删除了零件号为 P2 的所有的记录。

```
UPDATE S
SET     STATUS = 2 * STATUS ,
        CITY = 'Rome'
WHERE   CITY = 'Paris' ;
```

这个 UPDATE 语句将所有在 Paris 的供应商的状态值（STATUS）都加倍，且将这些供应商转移到 Rome。

注意：SQL 没有直接的**关系赋值**的操作。然而，可以通过下面的方法实现该操作：首先将目标表中的所有行删除，然后执行一个 INSERT…SELECT…命令（如上面的第一个例子所示）将数据插入表中。

## 4.3 目录

SQL 标准还包括了称为**信息模式**（Information Schema）的标准目录的规范。实际上，惯用的两个名词“目录”（catalog）和“模式”（schema）都在 SQL 中使用，但是各具有 SQL 的特定含义。笼统地说，一个 SQL **目录**包括的是对某一单个数据库的描述[⊖]，而一个 SQL 模式包括的则是某一用户数据库的某一部分的描述。换句话说，目录可以有很多（每个数据库一个），每个目录可以由很多模式组成。然而每一个目录必须要包括一个叫做 INFORMATION_SCHEMA 的模式，而从用户的观点来看，正是该模式起到了目录的作用。

信息模式由一系列的 SQL 表组成，这些表的内容以一种非常精确的方式定义，因此可以非常有效地显示该目录中其他模式的定义。更精确地说，信息模式定义为一组假定存在的“定义模式”（Definition Schema）的视图。同样实现中不需要支持“定义模式”，但是必须（a）支持一些类型的“定义模式”；（b）并且支持跟信息模式类似的“定义模式”的视图。这里有如下要点：

1）之所以提出如上所说的 a 和 b 两个概念，是基于以下的考虑。首先，现在的很多产品支持与“定义模式”类似的内容。然而，那些“定义模式”在各种产品中的定义有很大的差别，即使是同类的、但是属于不同厂家的产品，其差别也很大。因此，需要支持对“定义模式”的视图进行预定义的观点是合理的。

2）因为在每一个目录中都有一个信息模式，所以提到信息模式时，不应该理解成是特定的某个。因此，一般来说，对一个用户有用的所有数据不可能由一个信息模式来描述。然而，为了简单起见，仍假定只有一个信息模式。

这里没有必要详细讨论信息模式的内容。下面仅列出一些重要的信息模式中的视图，依据这些模式的名字，可以容易地看出该模式中包含的内容。然而，需要说明的是，TABLES 视图包含了所有的视图和基本表的信息，而 VIEWS 视图则只包含了视图的信息。

---

⊖ 从准确性的角度说，SQL 标准中并没有一个与“数据库”意义等同的东西。精确地说，目录定义的、由数据的集合所描述出来的东西是依赖于具体实现的。然而，就因此认为它就是数据库是不合理的。

```
ASSERTIONS                      TABLES
CHECK_CONSTRAINTS               TABLE_CONSTRAINTS
COLUMNS                         TABLE_PRIVILEGES
COLUMN_PRIVILEGES               USAGE_PRIVILEGES
COLUMN_UDT_USAGE                USER_DEFINED_TYPES
CONSTRAINT_COLUMN_USAGE         UDT_PRIVILEGES
CONSTRAINT_TABLE_USAGE          VIEWS
KEY_COLUMN_USAGE                VIEW_COLUMN_USAGE
REFERENTIAL_CONSTRAINTS         VIEW_TABLE_USAGE
SCHEMATA
```

参考文献［4.20］提供更多关于信息模式的细节描述。特别地，该参考文献还介绍了如何在信息模式的基础上显示查询。当然，查询过程没有读者想像的那么直接。

## 4.4　视图

首先给出一个 SQL 视图定义的例子：

```
CREATE VIEW GOOD_SUPPLIER
    AS SELECT S#, STATUS, CITY
       FROM   S
       WHERE  STATUS > 15 ;
```

以下给出了一个在该视图上定义的查询：

```
SELECT S#, STATUS
FROM   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

用视图的定义代替视图的名字，可以写成如下的表达式（注意 FROM 子句中的**嵌套子查询**）：

```
SELECT GOOD_SUPPLIER.S#, GOOD_SUPPLIER.STATUS
FROM ( SELECT S#, STATUS, CITY
       FROM   S
       WHERE  STATUS > 15 ) AS GOOD_SUPPLIER
WHERE  GOOD_SUPPLIER.CITY = 'London' ;
```

上面的表达式还可以简化成如下的形式：

```
SELECT S#, STATUS
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

后一个查询是真正要执行的查询。

考虑另外一个关于 DELETE 操作的例子：

```
DELETE
FROM   GOOD_SUPPLIER
WHERE  CITY = 'London' ;
```

这个 DELETE 语句的实际执行是这样的：

```
DELETE
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

## 4.5　事务

SQL 包含与第 3 章提到的 BEGIN TRANSACTION、COMMIT 和 ROLLBACK 语句对等的语句，分别是 **START TRANSACTION**、**COMMIT WORK** 和 **ROLLBACK WORK**（其中的关键字 WORK 可以省略）。

## 4.6 嵌入式 SQL

大多数的 SQL 语句允许直接执行（例如直接在联机终端上交互运行）或**嵌入**应用程序中（如 SQL 语句可以嵌入程序语言中，并跟它们一起使用）执行。而且，在嵌入的情况下，应用程序可以使用多种宿主语言；SQL 标准包含了对于 Ada、C、COBOL、Fortran、Java、M（之前叫做 MUMPS）、Pascal 和 PL/I 语言嵌入式 SQL 支持的规定。本节主要介绍嵌入式 SQL 的情况。

嵌入式 SQL 的基本思想就是任何可以交互使用的 SQL 语句都可以在应用程序中嵌入使用，叫做**双重模式原理**。当然，SQL 语句的交互形式和嵌入形式之间有很大的区别，尤其是检索操作需要在一个主程序的环境中提供一些必要的扩展（见本节中后面的描述），但是基本原则无论如何是不变的。注意：反过来就不一定对了，如一些嵌入式的 SQL 就不能在交互的环境下运行。

在实际讨论嵌入式 SQL 之前，有必要介绍一些基本的细节知识。读者可在图 4-3 的程序片断中看到这些细节的说明。书中的宿主语言使用 PL/I，若使用其他的宿主语言，只需要做小的改动即可。下面做几点说明：

```
EXEC SQL BEGIN DECLARE SECTION ;

   DCL SQLSTATE CHAR(5) ;
   DCL P#       CHAR(6) ;
   DCL WEIGHT   FIXED DECIMAL(5,1) ;

EXEC SQL END DECLARE SECTION ;

P# = 'P2' ;                    /* for example                */
EXEC SQL SELECT P.WEIGHT
         INTO   :WEIGHT
         FROM   P
         WHERE  P.P# = P# ( :P# ) ;
IF SQLSTATE = '00000'
THEN ... ;                     /* WEIGHT = retrieved value */
ELSE ... ;                     /* some exception occurred  */
```

图 4-3 使用了嵌入式 SQL 的 PL/I 程序片断

1）为了将 SQL 语句与主语言分开，嵌入式 SQL 以 **EXEC SQL** 开始，并以特殊的**结束符号**（PL/I 中使用一个分号）结束。

2）一个可执行的 SQL 语句（在本节的其他部分，对嵌入式 SQL 将省略“嵌入式”这个限定词）可以出现在任何的主语言可执行的地方。另外，注意“可执行的”意思。嵌入式 SQL 包含很多说明的、但不可执行的语句，这跟交互式 SQL 是不一样的。例如，DECLARE CURSOR 就不是一个可执行的语句（见后面的“使用游标的操作”小节），BEGIN 和 END DECLARE SECTION（见下面的第 5 点）和 WHENEVER（见下面的第 9 点）也不是可执行语句。

3）SQL 语句可以使用**主变量**；但是使用之前必须在其前面加一个**冒号前缀**，将其与 SQL 的列名区分开来。主变量可以出现在嵌入式 SQL 中，而字符值可以出现在交互式 SQL 的任何地方。主变量还可以出现在 SELECT（见第 4 点）或 FETCH 语句（见下面的“使用游标的操作”小节）中的 INTO 子句中，为一些修改操作指定目标。

4）注意图 4-3 的 SELECT 语句中的 **INTO** 子句。该子句的作用是给出新值要插入到的目标变量，INTO 子句中的第 $i$ 个目标变量对应 SELECT 子句中检索的第 $i$ 个值。

5）所有在 SQL 语句中使用的主变量必须要在**嵌入式 SQL 的声明部分**进行声明（在 PL/I 中用 DCL），在 BEGIN 和 END DECLARE SECTION 两个声明语句之间。

6）每一个包含嵌入式 SQL 语句的程序必须包括一个叫做 **SQLSTATE** 的主变量。在每一个 SQL 语句执行之后，执行的状态值将返回到该变量中；特别地，状态码为 00000 表示该语句正确执行，02000 的状态值表示语句执行但是没有满足要求的数据返回（参见 [4.23] 对其他代码值的详细解释）。因此，原则上来看，程序中的每一个 SQL 语句之后应该有一个对 SQLSTATE 值的检测，并且当返回的结果值不是预想的情况时，对其做适当的处理。然而，这样的检测实际上

经常是隐含的（见下面的第9点）。

7）主变量的**数据类型**必须要跟它的用途一致。特别地，当一个主变量作为目标变量（如在SELECT语句中）时，它的数据类型必须跟要赋值的数据类型相一致；同样，当一个主变量要用作源变量（如在INSERT语句中）时，它的数据类型必须与要插入到的列的数据类型相一致。这么做的深层次细节的原因比较复杂，因此，本章余下的部分先忽略它的大部分内容，将它放到5.7节来解释。

8）主变量和SQL的列可以有相同的名字。

9）如上面所提到的，每一个SQL语句之后最好有一个检测SQLSTATE返回值的语句。**WHENEVER**语句就是提供来简化这个过程的。WHENEVER的语法如下：

```
EXEC SQL WHENEVER <condition> <action> ;
```

< *condition* >可以是NOT FOUND、SQLWARNING或SQLEXCEPTION，还包括其他具体SQLSTATE值和违背完整性约束的条件；< *action* >或者是CONTINUE，或者是GO TO语句。WHENEVER不是一个可执行语句，而是给SQL编译器的一个说明语句。“WHENEVER < *condition* > GO TO < *label* >”可以使编译器在遇到每一个可执行的SQL语句之后，插入一个形式为“IF < *condition* > GO TO < *label* >...”的语句；“WHENEVER < *condition* > CONTINUE”就不会导致插入这样的语句，隐含的是程序员手工插入这样的语句。< *condition* >的三个值定义如下：

```
NOT FOUND       表示无数据——SQLSTATE = 02xxx
SQLWARNING      表示发生了一个次要错误——SQLSTATE = 01xxx
SQLEXCEPTION    表示发生了一个关键错误——SQLSTATE 的值参见 [4.23]
```

在对程序的顺序扫描过程中，SQL编译器遇到的每一个WHENEVER语句将覆盖其前一个WHENEVER语句。

10）最后需要注意：嵌入式SQL（使用第2章的术语）在SQL和宿主语言之间组成了一个松散的耦合。

基础知识就介绍到此，在本节的剩余部分将专门介绍数据操纵。正如前面所说的，大多数的这种操作可以以直接的方式处理，仅需要在语法上做很小的改变。然而，检索操作需要一些特殊的处理。这其中的问题是检索操作一般来说会检索到多行，而不是一行，而宿主语言一般不可能一次处理多行。因此，必须要有一种方式来协调SQL语言的集合操作的检索能力和宿主语言的行操作的处理能力，而**游标**就是协调两者的桥梁。游标是特殊类型的SQL实体，只在嵌入式SQL中使用（因为交互式SQL不需要）。游标实际上是一个（逻辑）指针，可以使用该指针来处理数据行的集合，按顺序将指针指向每一行，这样就可以一次定位一行。有关游标的细节将在后面的小节中讨论，下面首先介绍那些不需要使用游标的语句。

**1. 不使用游标的操作**

不需要游标的数据操纵语句有：

- 查询结果为单记录的SELECT语句
- INSERT语句
- UPDATE（除了CURRENT形式的UPDATE，见下一小节）语句
- DELETE（除了CURRENT形式的DELETE，见下一小节）语句

下面按顺序给出每个语句的示例。

**查询结果为单记录的SELECT语句**

列出供应商号跟主变量GIVENS#相同的供应商的状态和所在的城市。

```
EXEC SQL SELECT STATUS, CITY
         INTO   :RANK, :TOWN
         FROM   S
         WHERE  S# = S# ( :GIVENS# ) ;
```

用单记录（singleton）SELECT表示一个SELECT语句的返回结果只有一行。在这个例子

中，如果在 S 表中只有一条记录满足 WHERE 子句的条件，则 STATUS 和 CITY 的返回值将赋值到主变量 RANK 和 CITY 中，并且 SQLSTATE 将设为 00000；如果 S 表中没有满足条件的记录，则 SQLSTATE 将为 02000；如果有多于一条的记录满足条件，则程序将出错，SQLSTATE 将返回一个错误代码。

INSERT

在 P 表中增加一个新的零件（主变量 P#、PNAME 和 PWT 分别给出零件号、零件名和零件的重量；不知道颜色和城市）。

```
EXEC SQL INSERT
         INTO   P ( P#, PNAME, WEIGHT )
         VALUES ( :P#, :PNAME, :PWT ) ;
```

新零件的 COLOR 和 CITY 值将设为程序的默认值。6.6 节将对 SQL 的默认值做详细的介绍。注意：因为某些超出本书范畴的原因，用户自定义的列的默认值需要是空值（null）。（详细的关于空值的讨论请见第 19 章，当然在此之前偶尔引用一下总是无可避免的。）

DELETE

删除所在城市为主变量 CITY 给定的供应商的记录。

```
EXEC SQL DELETE
         FROM   SP
         WHERE  :CITY =
              ( SELECT CITY
                FROM   S
                WHERE  S.S# = SP.S# ) ;
```

如果 SP 表中没有满足 WHERE 子句中给定的条件的记录，则 SQLSTATE 将被赋值为 02000。另外，需要注意 WHERE 子句中的嵌入式子查询。

UPDATE

将伦敦的供应商的状态增加由主变量 RAISE 给定的值。

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CITY = 'London' ;
```

如果没有满足条件的记录，SQLSTATE 将置为 02000。

**2. 使用游标的操作**

现在来看集合的检索，例如，包含任意行的集合的检索，而不是上面所说的检索结果只为一行的情况。正如上面所说的，这需要一种机制来逐行存取集合中的记录，而游标就提供了这样的机制。处理过程可见图 4-4，该例是对某些供应商的细节（S#、SNAME 和 STATUS）进行查询，这些供应商所在的城市是由主变量 Y 给定的。

```
EXEC SQL DECLARE X CURSOR FOR       /* define the cursor   */
         SELECT S.S#, S.SNAME, S.STATUS
         FROM   S
         WHERE  S.CITY = :Y
         ORDER  BY S# ASC ;

EXEC SQL OPEN X ;                   /* execute the query   */
         DO for all S rows accessible via X ;
            EXEC SQL FETCH X INTO :S#, :SNAME, :STATUS ;
                                    /* fetch next supplier */
            .........
         END ;
EXEC SQL CLOSE X ;                  /* deactivate cursor X */
```

图 4-4 多行检索示例

说明："DECLARE X CURSOR ..." 语句定义了一个名为 X 的游标，通过在 DECLARE 声明中的 SELECT 语句，可将该游标跟一个表表达式（例如，从表中取值）相关联。因为 DECLARE

CURSOR 是一个纯粹的声明语句，因此表的表达式不会在此时执行。当执行了“OPEN X”打开游标时，这个语句才执行。“FETCH X INTO ...”语句则是一次从结果集中取一行，并且根据 INTO 子句中的说明将取得的值赋到主变量中。（为了简单起见，这里将主变量与数据库中的列命名为一样的名字。注意在游标声明部分的 SELECT 语句没有 INTO 子句。）因为在结果集中有多行，因此 FETCH 语句一般以循环的方式出现，而且只要结果集中还有未处理的行，循环就一直进行。一旦从循环中退出，就执行“CLOSE X”，关闭游标。

现在详细讨论游标和游标操作。首先，使用“DECLARE CURSOR”语句定义游标，它的形式一般是：

```
EXEC SQL DECLARE <cursor name> CURSOR
         FOR <table exp> [ <ordering> ] ;
```

为了使该形式看起来更简洁，此处忽略了一些可选的选项。可选的 < *ordering* > 的语法是这样的：

```
ORDER BY <order item commalist>
```

其中（a） < *order item commalist* > 列表不能为空，（b）每个 < *order item* > 包含一个列名（注意，这是没有限定的[⊖]），后面跟着可选的 ASC（升序）和 DESC（降序），默认值是升序。如果没有指定 ORDER BY 子句，排序由系统决定。（同样可以认定的是，如果指定了 ORDER BY 子句，排序时至少满足 < *order item commalist* > 的行将由之决定。）

注意：按如下的方式定义 commalist（逗号列表）。使用 < *xyz* > 指明任意一个符合语法的类，例如，在一些 BNF 产生式规则左边出现的符号。用 < *xyz commalist* > 指明一个或多个 < *xyz* >，在这里面，每一个 < *xyz* > 由逗号隔开，也有可能是一个或多个空格。注意会在以后的语法规则中沿用 commalist（在所有的语法规则中，而不只是 SQL 语法中）。

正如前面所说的，DECLARE CURSOR 语句是声明性的，而不是可执行的；它声明了一个有明确名字的游标，该游标对应明确的表的表达式，以及相关的排序。在表的表达式中可以使用主变量。一个程序可以包含任意多个 DECLARE CURSOR 语句，每一个这样的语句对应一个完全不同的游标。

在游标上可以进行三种操作：**OPEN**、**FETCH** 和 **CLOSE**。

- 语句

```
EXEC SQL OPEN <cursor name> ;
```

打开某个未被打开过的游标。跟该游标相关联的表表达式将被执行（使用在表达式中主变量的值）。这样就标识了记录集，并且使该记录集成为游标的当前**活动集**（active set）。游标还定位一个指针指向活动集第一行的前面。注意：因为活动集总是有顺序的——见前面关于 ORDER BY 的讨论，所以位置的概念有一定的意义[⊖]。

- 语句

```
EXEC SQL FETCH <cursor name>
         INTO <host variable reference commalist> ;
```

在当前活动集中将一已经打开的游标向前推进一行，并将结果的第 *i* 个值赋值到 INTO 子句的第 *i* 个主变量中。如果 FETCH 语句执行时记录集中已经没有记录，SQLSTATE 将被赋值为 02000，并且不返回任何待修改的数据。

- 语句

---

⊖ 实际上，SQL：1999 标准通过一堆复杂的规则限定了可以使用的列名 < *table exp* >，同时它也指定了规则规范 < *order item* > 可以是（a）一个可计算的表达式，比如 ORDER BY A + B，或者是（b）不出现在结果中的某个列名，比如 SELECT CITY FROM S ORDER BY STATUS。详细的细节已经超出本书范畴，不作讨论。

⊖ 集合实际上没有顺序可言（见第 6 章），因此“当前活动集”不是真正意义上的集合，认为它是排序列表或者数组比较合适。

```
EXEC SQL CLOSE <cursor name> ;
```

关闭一个当前已经打开的游标。关闭后的游标没有了当前活动集。然而，它还可以被重新打开，在游标重新打开后，它将跟另外一个活动集对应，该活动集可能不是上一个同样的活动集，在游标声明中引用的主变量的值改变的情况下，更是这样。注意，在游标打开的情况下改变主变量的值在当前的活动集中是无效的。

游标还可以跟另外两个语句结合使用。它们是：**CURRENT** 形式的 **UPDATE** 语句和 **DELETE** 语句。如果将一个名为 *X* 的游标定位在某一特定的行，就可以对“*X* 游标的当前内容”进行 UPDTAE 或 DELETE 操作，如 *X* 游标指向的当前记录。例如：

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS + :RAISE
         WHERE  CURRENT OF X ;
```

如果表的表达式中引用的是一个不可更新的视图（见 10.6 节），那么 CURRENT 形式的 UPDATE 和 DELETE 语句就是不允许的。

## 4.7 动态 SQL 和 SQL/CLI

前一节的讨论假定我们能够在某时间之前（比如在执行之前）完整地编译程序（包含 SQL 语句）。但对于某些确定的应用而言，这种假设是没有道理的。要举例的话，考虑一个联机应用：回想一下第 1 章，其中的一个联机应用支持从联机终端机或者其他类似的东西存取数据库。概括地说，一个典型的联机应用程序要按照如下的步骤执行：

1）从终端接受命令

2）分析命令

3）在数据库上执行适当的 SQL 语句

4）将消息或结果返回到终端

如果程序在第 1 步中收到的命令集非常小，如航班预订处理程序的情况，那么要执行的 SQL 语句就会非常少，因此就可以在程序中做固定处理。在这种情况下，第 2 步和第 3 步就是简单地检查输入的命令，并将其分解到处理预定义的 SQL 语句的分支程序进行处理。另一方面，如果输入有很大的灵活性，进行预处理和固定某些处理就是不恰当的。相反，这个时候如果动态地构造 SQL 语句就方便得多，然后对构造的 SQL 语句进行动态编译和执行。本节描述的 SQL 设施就是用来辅助这种过程的开发的。

### 1. 动态 SQL

**动态 SQL** 是嵌入式 SQL 的一种形式。它由一些“动态语句”组成，它们事先被编译好，用来精确地支持在运行时构建起来的普通 SQL 语句的编译和执行工作。两个基本的动态语句是 PREPARE（效果上等于编译）和 EXECUTE。我们可以通过下面 PL/I 的例子（不是很实际，但是非常精确）来说明它们的使用：

```
DCL SQLSOURCE CHAR VARYING (65000) ;

SQLSOURCE = 'DELETE FROM SP WHERE QTY < QTY ( 300 )' ;
EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE ;
EXEC SQL EXECUTE SQLPREPPED ;
```

说明：

1）SQLSOURCE 标识了一个 PL/I 的变长字符串变量，程序需要在该变量中构造 SQL 语句的源形式，如字符串的表示形式。例子中是一个 DELETE 语句。

2）SQLPREPPED 标识了一个 SQL 变量，而不是一个 PL/I 变量。该变量将用来接收编译后的 SQL 语句，这些 SQL 语句的源形式是由 SQLSOURCE 给出的。（当然名字 SQLSOURCE 和 SQLPREPPED 是任选的。）

3）PL/I 的赋值语句“SQLSOURCE = …”将一个 SQL DELETE 语句的源形式赋值到 SQLSOURCE 中。当然，实际构造这样一个源语句的过程非常灵活，可能还包括对终端用户用自然

语言（或者是用比 SQL 更友好的语言）输入的请求的分析。

4）PREPARE 取得源语句，并且准备将它处理（如编译）为可执行的形式，存放在 SQLPREPPED 中。

5）最后，EXECUTE 语句执行 SQLPREPPED，并因此生成真正的 DELETE。执行完后将返回值置到 SQLSTATE 中，就像是直接运行 DELETE 语句一样。

注意：因为 SQLPREPPED 声明的是一个 SQL 变量，而不是一个 PL/I 变量，因此当它在 PREPARE 和 EXECUTE 语句中使用时不需要加冒号。而且这样的变量无需显式声明。

顺便说一下，上述处理过程与 SQL 语句交互式输入的处理情况类似。大多数系统都包含交互式 SQL 查询处理器。这些查询处理器其实是特殊的联机应用程序，它可以接受多种类型的输入，不仅可以接受有效的 SQL 语句，有时还可以接受无效的 SQL 语句。它根据用户的输入，使用动态 SQL 的便利性构造适当的 SQL 语句，编译和执行构造好的语句，并将结果信息或结果集返回到终端。

当然，关于动态 SQL，远不止刚刚描述的 PREPARE 和 EXECUTE 这些内容；例如，还有方法可以使得带参数的 SQL 语句也可以通过编译，并在执行前将实际的参数填入，对于前一节提到的游标来说，也有与之类似的机制。特别是，存在一个 EXECUTE IMMEDIATE 语句，它在一条语句中结合了 PREPARE 和 EXECUTE 语句的功能。

**2. 调用级接口**

**SQL 调用级接口**（SQL Call-Level Interface，SQL/CLI，简写为 CLI）已经于 1995 年加入到 SQL 标准中来。SQL/CLI 大部分建立在微软的开放数据库互连接口 ODBC 的基础之上。CLI 允许一个用一般的宿主语言写的应用程序通过激活开发商提供的过程（而不是嵌入式 SQL）来发出数据库操作请求。这些过程将连接至应用程序，然后应用程序根据需要使用动态 SQL 执行数据库的操作请求。换句话说，从 DBMS 的角度来看，可以简单地将 SQL/CLI 过程理解为一个一般的应用程序。

读者可以看到，SQL/CLI（和 ODBC）跟动态 SQL 一样，都针对了同样的问题：应用程序中的 SQL 语句可以在程序运行前不完全确定。然而，SQL/CLI 对这个问题提供了一个比动态 SQL 更好的解决办法。这有两个原因：

- 动态 SQL 是一个源码级的标准。任何使用动态 SQL 的应用程序如果要处理按照动态 SQL 标准所写的操作，如 PREPARE、EXCUTE 等，都需要 SQL 编译器的支持。而 SQL/CLI 则与之不同，它将某些例行程序的细节进行了标准化（例如，子程序的调用等）；因而不需要特殊的编译处理，而只需要一般的宿主语言编译器的支持。因此，应用程序可以以一种封装的目标代码的形式发布（可能由第三方开发商提供）。
- 更大的优势是，这些 SQL/CLI 应用程序可以具有 DBMS 独立性；也就是说，SQL/CLI 具有允许创建一般性应用程序的特性（可以由第三方软件开发商提供），所创建的应用程序可在几个 DBMS 上使用，而不是只能应用到特定的 DBMS 上。

下面给出一个模拟前面动态 SQL 示例的 SQL/CLI 程序的示例：

```
char sqlsource [65000] ;

strcpy ( sqlsource,
         "DELETE FROM SP WHERE QTY < QTY ( 300 )" ) ;
rc = SQLExecDirect ( hstmt, (SQLCHAR *)sqlsource, SQL_NTS ) ;
```

说明：

1）现实世界的 SQL/CLI 应用程序倾向于采用 C 作为宿主语言，因此我们在例子中也使用 C 而不是 PL/I。我们也遵循 SQL/CLI 规范将变量名、过程名等类似的名字写为小写（或者是大小写夹杂），取代了本书其他部分对于这些名字全部采用大写的习惯。不要过分在意这一点，因为准确地说，它只是在宿主语言中调用过程时的一种标准规范，一般来说 SQL/CLI 语法（当然不是相应的语义）将随着宿主语言的变化而变化。

2）C 函数 **strcpy** 是用来将某个 SQL DELETE 语句复制至 C 变量 **sqlsource** 中。

3）C 赋值语句（“=”）调用了 SQL/CLI 过程 **SQLExecDirect**——动态 SQL 的 EXECUTE IMMEDIATE 语句的等价物——来执行 **sqlsource** 中的 SQL 语句，并将执行的结果代码赋给 C 变量 **rc**。

你应该已经预计到了，SQL/CLI 包含了动态 SQL 中差不多所有特性的等价物，还加入了一些新的东西。更深入的细节已经超出了本书的范围。但是，你应该掌握这些接口，比如 SQL/CLI、ODBC 和 JDBC（ODBC 的 Java 衍生物），实践中它们正变得越来越重要。你可以在 21.6 节中找到进一步的讨论。

## 4.8 SQL 并不完美

如 4.1 节提到的，SQL 远远不是完美的关系语言——它被自身的繁琐冗长和承载的期望和责任这些缺陷所拖累。在接下来的章节中的合适地方，我们会介绍 SQL 的具体不足之处；SQL 最大的问题在于：它在很多方面不能合适地支持关系模型。因此，现在的很多 SQL 产品声称为“关系”的完全是名不副实。确实，就笔者的观察来看，现在市场上还没有产品能真正完整地支持关系模型。[⊖] 这并不是说关系模型的某些部分是不重要的；相反，模型的每一个细节都很重要，而且从实际的使用来看更重要。是的，强调这一点并不为过，提出关系理论的目的不是为了理论而理论，而是给百分百实践意义上的实际系统提供理论基础。但是不太乐观的是，各个开发商都没有真正将理论作为整体来实施。因此，从某些方面来说，现在的“关系的”产品都没有能够真正贯彻关系理论。

## 4.9 小结

本章包括对 SQL 标准的一些主要特征的介绍。我们从商业的角度强调了 SQL 的重要性，虽然从纯关系的角度来看，它不很合适。

SQL 包括**数据定义**语言（DDL）和**数据操纵**语言（DML）部分。DML 可以在外模式（视图）上操作，也可以在**概念模式**（基表）上操作。同样，SQL 的 DDL 既可以用在外模式（视图）上，也可以用在概念模式（基表）上定义对象；在大多数商业系统中，甚至还支持内模式（如索引和其他的辅助结构）。另外，SQL 还有数据控制的功能，例如，它可以提供很多既不属于 DDL 也不属于 DML 的功能。GRANT 语句便是其中一例，该语句允许用户互相授予存取权限（见第 17 章）。

本章说明了如何使用 SQL 中的 **CREATE TABLE** 语句创建基本表，顺便也接触了 **CREATE TYPE** 语句。并且接着给出了 **SELECT**、**INSERT**、**UPDATE** 和 **DELETE** 语句的例子，说明了如何利用 **SELECT** 来实现选择、投影和连接运算。还简单描述了**信息模式**，信息模式包含了一些事先假定的“定义模式”的视图。本章还对 SQL 在处理**视图**和**事务**中的功能做了简单的介绍。

本章的大部分篇幅主要介绍**嵌入式 SQL**。嵌入式 SQL 的基本思想是**双重模式的原则**，即，任意可以交互使用的 SQL 语句也可以在一个应用程序中使用。这一原则的一个主要例外是**多行检索操作**，在这种情况下，需要使用**游标**来协调 SQL 的集合操作和宿主语言如 PL/I 的单行操作的差异。

在讲解了一些基本知识（虽然大多数都是语法的讲解）——包括对 **SQLSTATE** 的简单说明——之后，主要介绍了下列几种操作：**查询单记录的 SELECT** 语句、**INSERT**、**DELETE** 和 **UPDATE** 语句，对它们的操作都不需要游标。之后又介绍了需要游标的操作，讨论了 **DECLARE CURSOR**、**OPEN**、**FETCH**、**CLOSE** 和 **CURRENT** 形式的 **UPDATE** 和 **DELETE** 语句。（标准中将 CURRENT 形式的这些操作称为定位型（positioned）UPDATE 和 DELETE 语句；将非 CURRENT 形式的 UPDATE 和 DELETE 语句称为搜索型（searched）UPDATE 和 DELETE 语句。）最后，简要介绍了**动态 SQL** 的概念，特别描述了 **PREPARE** 和 **EXECUTE** 语句，还提到了 **SQL 调用级接口**，即 SQL/CLI（还提到了 ODBC 和 JDBC）。

---

⊖ 参见 [20.1]。

## 习题

4.1　图4-5给出了一些样本数据值，这些数据来自扩展后的供应商和零件数据库：供应商－零件－工程数据库。○ 供应商（S）、零件（P）和工程（J）有唯一的标识，分别为供应商号（S#）、零件号(P#)和工程号（J#）。SPJ的一行表示特定的供应商S#给某一工程J#供应特定数量QTY的某一零件P#（如图中所示，{S#, P#, J#} 即是主码）。为该数据库写一个适当的SQL定义。注意：这个数据库将要作为以后章节很多习题的基础。

S

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

P

| P# | PNAME | COLOR | WEIGHT | CITY |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P2 | Bolt | Green | 17.0 | Paris |
| P3 | Screw | Blue | 17.0 | Oslo |
| P4 | Screw | Red | 14.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |
| P6 | Cog | Red | 19.0 | London |

J

| J# | JNAME | CITY |
|---|---|---|
| J1 | Sorter | Paris |
| J2 | Display | Rome |
| J3 | OCR | Athens |
| J4 | Console | Athens |
| J5 | RAID | London |
| J6 | EDS | Oslo |
| J7 | Tape | London |

SPJ

| S# | P# | J# | QTY |
|---|---|---|---|
| S1 | P1 | J1 | 200 |
| S1 | P1 | J4 | 700 |
| S2 | P3 | J1 | 400 |
| S2 | P3 | J2 | 200 |
| S2 | P3 | J3 | 200 |
| S2 | P3 | J4 | 500 |
| S2 | P3 | J5 | 600 |
| S2 | P3 | J6 | 400 |
| S2 | P3 | J7 | 800 |
| S2 | P5 | J2 | 100 |
| S3 | P3 | J1 | 200 |
| S3 | P4 | J2 | 500 |
| S4 | P6 | J3 | 300 |
| S4 | P6 | J7 | 300 |
| S5 | P2 | J2 | 200 |
| S5 | P2 | J4 | 100 |
| S5 | P5 | J5 | 500 |
| S5 | P5 | J7 | 100 |
| S5 | P6 | J2 | 200 |
| S5 | P1 | J4 | 100 |
| S5 | P3 | J4 | 200 |
| S5 | P4 | J4 | 800 |
| S5 | P5 | J4 | 400 |
| S5 | P6 | J4 | 500 |

图4-5　供应商－零件－工程数据库（样本值）

4.2　在4.2节中，按照SQL标准本身介绍了CREATE TABLE语句。许多商用的SQL产品还在标准语句的基础之上支持附加的选项，支持索引的处理、磁盘空间的分配和其他一些实现时的事务。这样就破坏了数据的物理独立性和系统间的兼容性。调查任何一个你可得到的SQL产品。看一下：自己预先对该产品的某些批评是否还是恰当的？特别地，写出该产品支持哪些CREATE TABLE附加的选项。

4.3　对某一SQL产品再进行一次调查。思考下列问题：该产品是否支持信息模式？如果不支持，它的目录支持是什么样子的？

4.4　写出下列对供应商－零件－工程数据库的更新操作的SQL语句：

a）在S表中插入一个新的供应商S10。供应商的名字是Smith，其所在的城市是New York，其状态还是未知的。

b）删除所有没有供货记录的工程。

c）将所有颜色为红色的零件改为颜色为橙色。

4.5　使用供应商－零件－工程数据库，写一个使用嵌入式SQL语句的程序，按照供应商号列出所有的供应商。每一个供应商后面必须紧跟着列出该供应商供应产品的所有工程行，工程行按照工程号排序。

4.6　假定PART和PART_STRUCTURE表是如下定义的：

```
CREATE TABLE PART
     ( P# P#, DESCRIPTION CHAR(100),
       PRIMARY KEY ( P# ) ) ;
```

○ 为了方便参考，在本书封三上也有图4-5与图3-8。

```
CREATE TABLE PART_STRUCTURE
     ( MAJOR_P# P#, MINOR_P# P#, QTY QTY,
       PRIMARY KEY ( MAJOR_P#, MINOR_P# ),
       FOREIGN KEY ( MAJOR_P# ) REFERENCES PART,
       FOREIGN KEY ( MINOR_P# ) REFERENCES PART ) ;
```

PART_ STRUCTURE 表说明了组成某一个零件（MAJOR_P#）第一层的其他零件（MINOR_P#）。写一个 SQL 程序，列出某一个给定零件的各层的组成零件（即**零件爆炸**问题）。注意：在图 4-6 中的样本数据对解决这个问题会有帮助。可以看到，表 PART_STRUCTURE 显示了材料单数据（见 1.3 节）在关系系统中如何描述。

| PART_STRUCTURE | MAJOR_P# | MINOR_P# | QTY |
|---|---|---|---|
| | P1 | P2 | 2 |
| | P1 | P3 | 4 |
| | P2 | P3 | 1 |
| | P2 | P4 | 3 |
| | P3 | P5 | 9 |
| | P4 | P5 | 8 |
| | P5 | P6 | 3 |

图 4-6 表 PART-STRUCTURE（样本数据）

## 参考文献

附录 H 中的 [3.3] 给出了 SQL：1999 与 *The Third Manifesto* 之间的详细比较。同时可以参考本书的附录 B。


[4.1] M. M. Astrahan and R. A. Lorie："SEQUEL-XRM: A Relational System," Proc. ACM Pacific Regional Conf., San Francisco, Calif. (April 1975).

描述了第一个实现 SEQUEL（SQL 的最早的版本）的原型系统 [4.9]。也参见 [4.2] 和 [4.3]。SEQUEL 的功能跟 System R 类似。

[4.2] M. M. Astrahan *et al.*："System R: Relational Approach to Database Management," *ACM TODS 1*, No. 2 (June 1976).

System R 是实现了 SEQUEL/2（后来是 SQL）语言的主要的原型系统 [4.10]。该文介绍了在初始计划中 System R 的体系结构。也参见 [4.3]。

[4.3] M. W. Blasgen *et al.*："System R: An Architectural Overview," *IBM Sys. J. 20*, No. 1 (February 1981).

描述了 System R 已经完全实现之后的体系结构。参照 [4.2]。

[4.4] Joe Celko: *SQL for Smarties: Advanced SQL Programming*. San Francisco, Calif.: Morgan Kaufmann (1995).

"这是第一本有用的高级 SQL 手册，对于想从 SQL 的一般用户升级到专业编程员的人来说，可以提供全面的技术介绍。"（摘自书的封面。）

[4.5] Surajit Chaudhuri and Gerhard Weikum: "Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System," Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt (September 2000).

这篇文章中有一些针对 SQL 的尖锐批评。引用一些："SQL 令人痛苦。来自数据库系统的最头疼的是 SQL 语言。它是任何想得到的特性的集合——很多特性极少使用或者根本就不该鼓励人们使用；对于典型的应用程序员来说也太复杂。其核心结构（即选择 - 投影 - 连接查询与聚集语句）十分有用，但是我们在其他细枝末节的特性是否普遍、加入它们是否明智的问题上有疑虑。理解 SQL：1992（先不管 SQL：1999）所有特性的语义、覆盖所有组合形式的嵌套（和相关）子查询、处理空值、支持触发器和抽象数据类型的函数，这些是一个噩梦。典型的 SQL 语言的教学只关注核心结构，将其他特性当作"边工作边学习"时得到的生活经验。一些行业杂志时不时登出一些 SQL 语言的测试题，它们通常是一个单个查询语句，且横跨数页，晦涩难懂，用来挑战读者能否理解这种请求中蕴涵的复杂信息。"

[4.6] Andrew Eisenberg and Jim Melton: "SQL:1999, Formerly Known as SQL3," *ACM SIGMOD Record 28, No. 1* (March 1999).

对于因 SQL：1999 发布而引入 SQL 标准中的新特性，该文是一个概要介绍。

[4.7] Andrew Eisenberg and Jim Melton: "SQLJ Part 0, Now Known as SQL/OLB (Object Language Bindings)," *ACM SIGMOD Record 27*, No. 4 (December 1998); "SQLJ—Part 1: SQL Routines Using the Java™ Programming Language," *ACM SIGMOD Record 28*, No. 4 (December 1999). See also Gray Clossman *et al.*: "Java and Relational Databases: SQLJ," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

SQLJ 最初是指一个探讨 SQL 和 Java 结合的可能途径的项目（它是几个最好的 SQL 厂商共同

努力的结果)。项目的0号部分处理Java程序中的嵌入式SQL的问题;1号部分关注在SQL中调用Java代码的想法(例如,调用一个用Java语言编写的存储过程——见第2章);2号部分探究将Java类用作SQL数据类型的可能性(例如,在定义SQL表列时提供帮助)。0号部分的工作已经被SQL:1999标准包含,1号和2号部分则差不多确定会在SQL:2003中包含进来(参见[4.23]的注释)。

[4.8] Donald D. Chamberlin: *Using the New DB2*. San Francisco, Calif.: Morgan Kaufmann (1996).

这是一本对商业SQL产品的现状做了全面介绍、可读性非常好的书籍,由SQL的两个最初的主要设计者所写[4.9~4.11]。注意:这本书也讨论了在SQL的设计过程中"一些有争议的结论"。主要是:(a)对空值的支持和(b)允许重复行。"Chamberlin认为对这两个问题的认识已经形成定论了,现在不需要来进行形式推理,他认为空值和重复行的问题是一个类似于宗教信仰的问题。最主要的是,[SQL]的设计者是实用者而不是理论家,这种倾向可以在设计的很多决定上看出来。"这种立场跟现在的许多作者有很大的不同!空值和重复值是一个科学的问题,而不是一个宗教信仰的问题;在这本书的第19章和第6章分别说明了这两个问题。在"实用的而不是[理论的]"这个问题上,认为"理论的就不实用"的观点是不正确的;在4.8节中我们已经表明了观点:关系理论至少是非常实用的。

[4.9] Donald D. Chamberlin and Raymond F. Boyce: "SEQUEL: A Structured English Query Language," Proc. 1974 ACM SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor, Mich. (May 1974).

这篇论文最先介绍了SQL语言(或者按照该论文作者原先的叫法是SEQUEL,因为法律上的原因,名字随后做了修改)。

[4.10] Donald D. Chamberlin *et al.*: "SEQUEL/2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM J. R&D. 20*, No. 6 (November 1976). See also the errata in *IBM J. R&D. 21*, No. 1 (January 1977).

基于以下两个条件,即[4.1]中对SEQUEL原型的实现经验和[4.28]中报告的可用测试的结果,提出了修订版SEQUEL/2语言。System R[4.2, 4.3]支持的语言基本上就是SEQUEL/2,但是该语言没有"断言"和"触发器"的功能(见第9章),该版还根据早期用户的使用经验增加了一定的扩充功能[4.11]。

[4.11] Donald D. Chamberlin: "A Summary of User Experience with the SQL Data Sublanguage," Proc. Int. Conf. on Databases, Aberdeen, Scotland (July 1980). Also available as IBM Research Report RJ2767 (April 1980).

给出了早期使用System R的用户经验,并且根据这些经验,提出了对SQL语言的一些扩充。有的扩充,如EXISTS、LIKE、PREPARE和EXECUTE等,实际上一直到System R的最终版本中才实现。8.6节(EXISTS)、附录B(LIKE)和4.7节(PREPARE和EXECUTE)对之有所描述。

[4.12] Donald D. Chamberlin *et al.*: "Support for Repetitive Transactions and *Ad Hoc* Queries in System R," *ACM TODS 6*, No. 1 (March 1981).

给出了System R在随意查询和"封装事务"环境中的性能测试的很多指标。一个"封装事务"是一个可存取数据库的一小部分的应用程序,需要在执行之前编译。它与2.8节的计划请求相对应。论文说明了在与System R类似的系统上,所有事情中(a)编译总是优于解释的,甚至在随意查询下也不例外;(b)如果系统中有适当的索引,系统有能力在一秒钟之内处理几个封装事务。当时,很多人宣称"关系系统是没有用的",而这篇论文则认为这种说法不恰当,该论文也因为第一次对该观点提出质疑而闻名。当然,自从它第一次发表以来,商业的关系产品在速度上已经达到了每秒钟上百个或上千个的事务处理能力。

[4.13] Donald D. Chamberlin *et al.*: "A History and Evaluation of System R," *CACM 24*, No. 10 (October 1981).

介绍了System R工程的三个主要阶段(初步的原型、多用户的原型和对原型的评价),强调了编译和优化技术,而就是这些技术使得System R比较超前。在这篇论文和参考资料[4.14]之间有一定的重复。

[4.14] Donald D. Chamberlin, Arthur M. Gilbert, and Robert A. Yost: "A History of System R and SQL / Data System," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

讨论了从 System R 原型中吸取的经验和教训，介绍了将该原型引入到 IBM 的 DB2 产品系列 SQL/DS（随后改名为 DB2 for VM and VSE）中的过程。

[4.15] C. J. Date: "A Critique of the SQL Database Language," *ACM SIGMOD Record 14*, No. 3 (November 1984). Republished in *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

正如在本章中所说的，SQL 并不完美。这篇文章对该语言的很多缺点给出了批判性的分析（主要从形式化的计算机语言而不是专门从数据库语言来看）。注意：这篇文章的观点有的不适用于 SQL：1999。

[4.16] C. J. Date: "What's Wrong with SQL?" in *Relational Database Writings 1985 - 1989*. Reading, Mass.: Addison-Wesley (1990).

除了 [4.15] 中所说的 SQL 的一些缺点，这篇文章还以下面的标题讨论了 SQL 的一些其他的不足之处："SQL 本身的不足之处"、"SQL 标准的不足之处" 和 "应用的可移植性"。注：本文的一些观点同样不适用 SQL：1999。

[4.17] C. J. Date: "SQL Dos and Don'ts," in *Relational Database Writings 1985 - 1989*. Reading, Mass.: Addison-Wesley (1990).

在 [4.15]、[4.16] 和 [4.19] 中说到了 SQL 的很多潜在的不足，这篇文章给出了很多使用 SQL 的建议，以尽量避免这些不足，而且文章中还给出建议以充分利用开发效率、可移植性和可连接性等，实现效益的最大化。

[4.18] C. J. Date: "How We Missed the Relational Boat," in *Relational Database Writings 1991 - 1994*. Reading, Mass.: Addison-Wesley (1995).

通过介绍 SQL 对关系模型的结构、完整性和操作的支持，简洁地说明了 SQL 的不足之处。

[4.19] C. J. Date: "Grievous Bodily Harm" (in two parts) *DBP&D 11*, No. 5 (May 1998) and No. 6 (June 1998); "Fifty Ways to Quote Your Query," *http://www.dbpd.com* (July 1998).

SQL 是一个冗余非常大的语言，就是差别很小的查询也有多种不同的表达方式。这篇文章说明了这一点，并且介绍了它们隐含的不同之处。特别地，文章还说明了 GROUP BY 子句、HAVING 子句和范围变量可以在不影响功能的情况下从语言中有效地去掉（在子查询结构中也一样）。注意：所有的这些 SQL 结构在第 8 章（8.6 节）中有介绍。

[4.20] C. J. Date and Hugh Darwen: *A Guide to the SQL Standard* (4th edition). Reading, Mass.: Addison-Wesley (1997).

这是一个更全面的 SQL：1992 标准的指南，包括 SQL/CLI（1995）和 SQL/PSM（1996），也是对 SQL：1999 标准的预览。特别地，该书中的附录 D 说明了 "标准中没有充分定义的，甚至是没有正确定义的很多方面"。这些问题中的大多数在 SQL：1999 中仍然存在。

[4.21] C. J. Date and Colin J. White: *A Guide to DB2* (4th edition). Reading, Mass.: Addison-Wesley (1993).

对 IBM 最初的 DB2 产品（1993）和一些相关产品做了广泛全面的综述。DB2 跟 SQL/DS [4.14] 一样，是在 System R 的基础上开发的，尽管 SQL/DS 继承的更加直接一些。

[4.22] Neal Fishman: "SQL *du Jour*," *DBP&D 10*, No. 10 (October 1997).

描述了在一些宣称 "支持 SQL 标准" 的 SQL 产品中发现的很多不能令人满意的不相容性。

[4.23] International Organization for Standardization (ISO): *Information Technology—Database Languages—SQL*, Document ISO/IEC 9075:1999. *Note*: See also reference [22.21].

这是 ISO SQL：1999 标准的定义的原本（有时候被称为 ISO/IEC 9075，有时候仅仅是 ISO 9075）。然而，原先的单个文档已经扩展到一系列开放的单独部分（ISO/IEC 9075 - 1，- 2 等）。在写该书的时候，下面的部分已经定义：

第一部分：框架（SQL/框架）
第二部分：基础（SQL/基础）
第三部分：调用接口（SQL/CLI）
第四部分：永久存储模块（SQL/PSM）
第五部分：宿主语言捆绑（SQL/绑定）
第六部分：暂缺
第七部分：暂缺

第八部分：暂缺

第九部分：外部数据的管理（SQL/MED）

第十部分：对象语言绑定（SQL/OLB）

正如本章前面所说，下一个版本预计是2003年发布，并引入如下的变化内容：

- 第五部分的材料将合并入第二部分，第五部分将被舍弃。
- 第二部分中定义标准数据库目录（“信息模式”）的材料将移至新的一部分，第十一部分“SQL Schemata”。
- 一个新的部分：第十三部分，“Java例程和类型（SQL/JRT）”将加入，用来标准化Java和SQL结合的过程（见［4.7］的注释部分）。
- 一个新的部分：第十四部分，“XML相关规范（SQL/XML）”将加入，用来标准化处理SQL和XML之间关系的特性（见第27章）。

顺便提一下——也值得一提，尽管广泛来说SQL是国际“关系”数据库的标准，但SQL标准文档中从来没有如此描述自身；事实上，它从没有使用过“关系”这个词。（如本章前面的脚注所言，既然如此，它也从未使用“数据库”一词。）

［4.24］ International Organization for Standardization（ISO）：（*ISO Working Draft*）—*Database Language SQL—Technical Corrigendum 5*，Document ISO/IEC JTC1/SC32/WG3（December 2，2001）.

包括了对［4.23］中所说的原始版本的一些修订和更正。

［4.25］ Raymond A. Lorie and Jean-Jacques Daudenarde：*SQL and Its Applications*. Englewood Cliffs，N. J.：Prentice-Hall（1991）.

关于如何使用“SQL”的书（该书中有一半的篇幅给出了具体的实际程序的学习示例）。

［4.26］ Raymond A. Lorie and J. F. Nilsson：“An Access Specification Language for a Relational Data Base System，”*IBM J. R&D. 23*，No. 3（May 1979）.

对System R［4.12，4.27］的编译机制给出了详细的解释。对于任何一个给定的SQL语句，System R优化器生成一个名为ASL（Access Specification Language）的内部语言程序。ASL语言是优化器和代码生成子的接口（代码生成子将一个ASL程序转为机器代码）。ASL包括在索引、存储文件等实体上的“扫描”和“插入”操作。ASL通过将程序分解为一些定义好的子过程，使所有的翻译过程更好管理。

［4.27］ Raymond A. Lorie and Bradford W. Wade：“The Compilation of a High-Level Data Language，”IBM Research Report RJ2598（August 1979）.

在System R中，查询的编译过程放在执行之前，而且在物理数据库结构有了明显的改变之后会自动地重编译，这种做法比较超前。这篇文章非常详细地论述了System R的编译和重编译机制，然而没有涉及优化的问题，该问题可参见后面的［18.33］。

［4.28］ Jim Melton and Alan R. Simon：*SQL：1999—Understanding Relational Components*. San Francisco，Calif.：Morgan Kaufmann（2002）.

一本SQL：1999教程（只含基础部分，高级主题放在了文献［26.32］中）。Melton是当前的SQL标准的编辑之一。

［4.29］ David Rozenshtein，Anatoly Abramovich，and Eugene Birger：*Optimizing Transact-SQL：Advanced Programming Techniques*. Fremont，Calif.：SQL Forum Press（1995）.

Sybase和SQL Server产品支持Transact-SQL这一SQL方言。这本书介绍了Transact-SQL的基于其一些特色函数（按照作者的定义：“这些函数可以允许程序员用SELECT、WHERE、GROUP BY和SET子句编写有条件的程序逻辑”）的一系列编程方法。虽然这是专门用Transact-SQL编写的，但是这个思想在实际中有很大的适用性。注意：应该说明，在本书标题中提到的“优化”不是指DBMS的优化器，而是用户可以自己手工执行的优化。

# 第二部分　关系模型

关系模型毫无疑问是现代数据库技术的基础；它是使得这一领域成为一门科学的基础。因此，任何有关数据库技术基础的书籍如果没有包含关系模型，那就是浅薄的。同样，如果没有深入理解关系模型，而声称是数据库领域的专家，这是无法让人接受的。其实关系模型并不非常难理解，但是需要强调，它是基础，并且在人们所能预见的时期内它仍然是。

在第3章曾经提到，关系模型涉及数据的三个基本方面：数据结构、数据操纵和数据完整性。在本书的这一部分，我们依次讨论这三个方面：

- 第5章和第6章讨论结构（第5章涉及类型，第6章涉及关系）。
- 第7章和第8章讨论操纵（第7章涉及关系代数，第8章涉及关系演算）。
- 第9章介绍完整性。

最后，第10章讨论视图这一重要主题。注意：可能需要补充的是关系模型划分为三部分，这点在较高的理论层面上非常有用，可以帮助我们了解得更清楚。正如我们将要看到的一样，实际上，关系模型的每一个独立部分彼此之间高度互连，彼此在很多方面相互依赖；因此，通常（或者原则上）不可能删除任何一部分而不破坏整体模型。在第5～10章中可以看到大量前后参照的内容。

关系模型不是一个静态的事物，理解这一点非常重要——过去几年里它不断发展和进化，未来将仍然如此。[⊖] 接下来的内容反映了作者和这个领域里其他工作者的思想（特别是，如同在前言提到的，参考了《对象关系数据库基础：第3版宣言》[3.3]一书的观点）。本书尽量使内容相对地完备准确，叙述风格虽然采用了授课形式，但读者不要把这些当作一成不变的东西。

需重申的是，关系模型并不难理解，但它是一种理论，大多数理论都带有它自己的专门术语（原因在3.3节已经说明），关系模型在这方面也不例外。本书的这一部分要用到自己的专门术语。不可否认，最初这些术语会带来一点混乱，并且会成为理解的障碍。因此，如果读者在理解上有些困难，需要耐心些，一旦熟悉了有关的术语，就会发现内容非常简单。

接下来的6章篇幅很长（这一部分完全可单独成书）。但是，长度正反映了这部分内容的重要性。用一两页的篇幅来概述这部分内容也是完全可能的；事实上，关系模型的一个主要优点就是其基本内容非常容易阐述和理解。然而，用过短的篇幅来介绍是不合理的，也不能说明关系模型的应用广泛性。本部分的长篇论述，不应看作是模型复杂性的表现，而应看作是该模型作为大量深度研究之基础的重要性和成功之处的诠释。若能深入理解这部分内容，则在未来的数据库实践中必会给读者无数倍的回报。

最后是一点关于SQL的说明。在本书的第一部分已解释过，SQL是标准关系数据库语言，市场上几乎每一种数据库产品都支持它（或者，更准确一点，是支持它的变种－参照[4.22]）。由此而来的结果是，不包含SQL的现代数据库书籍是不能称其为全面的。因此，在接下来关于关系模型各方面的章节中，都会在适当的地方顺便讨论相关的SQL实用程序（第4章覆盖SQL的基本内容）。

---

⊖ 在这点上关系模型类似于数学（数学也不是静态的，而是随时间发生改变）；事实上，关系模型可以被视为数学的一个小分支。

# 第5章 类 型

## 5.1 引言

注意：这一部分读者第一次可以从头到尾粗略地读一遍。这一部分逻辑上应该放在这里，但是它的大部分内容直到第五部分的第 20 章和第六部分的第 25～27 章才会用到。

数据类型（简称类型）的概念非常基础：值、变量、参数、只读操作符，特别是关系属性，都是类型的一种。那么，什么是类型呢？它是相关类型所有值的集合。例如，类型 INTEGER 是所有整数的集合；类型 CHAR 是所有字符串的集合，类型 S#是所有供应商号码的集合；等等。当我们举例说供应商 S 有一个类型为 INTEGER 的属性 STATUS 的时候，我们指的是这个属性的值是整数，而且只能是整数。

注意，有两点需要指出：

- 首先，类型也称为域，特别是在关系中更多地使用域这个术语；实际上，这本书的早期版本都使用的是后者，但是现在本书更倾向于使用类型这个术语。
- 其次，我们提醒读者，本章尽量保证内容合理正确。因此，我们不能说类型 INTEGER 是所有的整数的集合，而应该说它是所有考虑到的计算机能够表达的整数的集合（显然有些整数超出了任何一台计算机所能表达的范围）。类似的限定也适用于本章的其他声明和例子。我们不再每次都将这个问题指出来，但是会一直坚持这一点。

任何给定的类型要么是**系统定义**类型（比如说内置类型），要么是**用户定义**类型。从本章的目的出发，我们假定前面提到的三种类型中，INTEGER 和 CHAR 都是系统定义类型，S#是用户定义类型。任何类型，不管它是系统定义类型还是用户定义类型，都能够作为正在声明的关系属性（和变量、参数及只读操作符——概念见 5.2 节）的基础。

任何给定类型都存在相关的**操作符**的集合，这些操作符都能有效应用于这些类型的值。换句话说，给定类型的值只能参与针对这个类型定义的操作符的操作（这里所说的“针对这个类型定义”的意思是指，我们所讨论的操作符有一个参数被声明为这个类型）。举个例子，在系统定义类型 INTEGER 中：

- 系统提供操作符“=”、“<”，等等，用来比较整数大小。
- 系统同时提供操作符“+”、“*”，等等，执行整数上的算术操作。
- 系统不提供“||”（连接）、SUBSTR（取子串）等操作符来在整数上进行字符串操作。换句话说，整数不支持字符串操作。

相应地，在用户定义类型 S#中，我们也能定义“=”、“<”等操作符，用来比较供应商号码。然而，不能定义“+”、“*”等操作符，即这种类型不支持供应商号码上的算术操作（其实对两个供应商号码相加或相乘是毫无意义的）。

现在我们将以［3.3］中提出的类型理论为基础，深入探讨前面提到的想法。

## 5.2 值与变量

首先要讨论的是值与变量之间的重要且基本的逻辑区别[⊖]（在文献材料中关于这点存在大量令人惊奇的争议）。参考［5.1］，我们提出以下的定义：

- **值**（value）是指一个独立的常量。例如，这个独立的常量就是整数 3。一个值本身没有时间和空间的位置，但是值能够通过某种内存内的编码得以表示，这样的表示或者出现（我们以前使用的术语）就在时间和空间上有了位置（location）。同一个值的不同出现在空间和时间上存在不同的位置，意思是说，任何不同变量都可以同时或者不同时有同一个

⊖ 参见［3.3］中对这个有用而且重要的概念的解释。

值。特别要注意的是，从定义上，**值不能更新**，因为如果它可以更新，那么经过更新后就不再是这个值了。

- **变量**（variable）是一个值的某次出现的所有者。一个变量在时间和空间上有自己的位置。而且，显然变量不像值一样，它是**能够更新的**。也就是说，我们讨论的这个变量的当前值可以被另一个值替换，这个值不同于前一个值。（当然这个变量在更新后还是同一个变量。）

请注意这并不只是像整数3那么简单。相反，值可以相当复杂；比如说一个值可以是一个几何点、一个多边形、一条X射线、一篇XML文档、一个指纹、一个数组、一个堆、一个列表、一个关系，等等。当然类似的观点也适用于变量。

接下来有一点很重要，我们必须区分本质上的值和在特定上下文中这个值的出现（比如说作为某些变量的当前值）之间的不同。正如已经解释过的，同一个值可以同时出现在许多不同的上下文中。每一个出现都包含这个值的某种编码或者物理表示；更进一步，那些编码并不必然相同。举例来说，整数值3确实只在整数集合中存在一次（好像宇宙中确实只存在一个整数3），但是任意数目的变量都可能同时把这个整数的一次出现当作当前值。更进一步，某些出现可能物理上通过十进制编码表示，另一些出现则是通过二进制编码表示。于是，一面是一个值的**一次出现**，另一面是这个出现的内部**编码**或者**物理表示**，这两者存在着逻辑不同。

很显然，虽然有上述的讨论，我们发现缩写会带来便利，“一个值的一次出现的编码”缩写为“一个值的出现”，或者直接缩写为“值”，只要不会产生混淆就可以这样做。注意，“值的出现”是一个模型（model）概念，而“一次出现的编码”是实现（implementation）概念。比如，用户可能需要知道某两个不同的变量是否包含同一个值的出现（即它们是否相等）；然而，他们不必知道这两个出现是否采用同一种物理编码。

**值和变量类型化**

每一个值都有（等价于“属于”）某种类型。换句话说，如果 *v* 是一个值，那么 *v* 需要有一个标记能够说明“我是一个整数”或者“我是一个供应商号码”或者“我是一个几何点”。在定义上，任何给定的值总是有一个明确的类型，[⊖] 这个类型不再发生改变。不同的类型之间没有交集，这意味着这些类型不会有相同的值。更进一步：

- 每个变量必须显式声明为某种类型，即每个变量的可能取值都是这个类型的一个值。
- 关系变量（relvar）——见第6章——的每个属性必须显式声明为某种类型，即每个属性的可能取值都是这个类型的值。
- 每个操作符——见5.5节——的返回结果都要显式声明属于某种类型，即每一个通过调用这个操作符得到的可能结果都是这个类型的值。
- 操作符的每个参数——见5.5节——都显式声明为某种类型，意思是替换参数的每一个可能自变量都是这个类型的值。（实际上这句话并不足够准确。操作符通常分为两个不相交的类别，只读操作符与更新操作符；只读操作符返回一个结果，但是更新操作符更新一个或者更多自变量。对于一个更新操作符来说，任何需要更新的自变量都要求是一个变量，而不是一个值，它的类型必须和相应的参数类型保持一致。）
- 通常，每一个表达式隐性声明为某种类型：也就是，类型声明为最外层操作符所包含的类型，这个最外层操作是指这个表达式最后执行的操作符。比如说，表达式 $a * (b + c)$ 的类型为操作符 * （乘）的类型。

此外，如果操作符是**多形态**（polymorphic）的，那么前面提过的操作符和操作符参数都需要一些调整。如果一个操作符在不同的调用中，它的参数P以及与这些参数P对应的自变量都属于不同的类型，那么这个操作符就是多形态的。相等操作符“=”就是一个明显的例子：我们可以检查任何两个值 *v1* 和 *v2* 是否相等（只要这两个值都属于同一类型），所以“=”是多形态的——它适用于整数、字符串、供应商号码和任何可能类型的值。类似的结论适用于赋值操作符

⊖ 当支持类型继承时情况可能会不同。从第20章起，我们才考虑类型继承。

“：=”（这个操作符可以定义为任何类型）：我们可以将任何一个值 $v$ 指定给任何变量 $V$，只要 $v$ 和 $V$ 属于同一类型。（当然，如果赋值违反了某些整数限制，赋值就会失败——见第 9 章——但是发生同样的类型错误就不会失败。）[○] 在第 20 章等其他章节中可以看到更多关于多形态操作符的例子。

## 5.3　类型与表示

我们已经谈到这样的事实，类型本身和这个类型的值在系统内部的物理表示之间存在着逻辑不同。实际上，类型是一个模型问题，而物理表示是一个实现问题。例如，供应商号码可以被物理地表现为字符串，但这并不意味着可以在供应商号码上进行字符串操作；仅当在类型上定义了相应的操作符，才能执行这些操作。当然，给一种类型定义操作符，依赖于该类型的具体含义和语义，而不依赖于类型的值被物理表示的方式，实际上，物理表示应该**对用户是隐藏的**。换句话说，我们所讨论的类型和物理表示之间的区别是数据独立（见第 1 章）的一个重要方面。

我们顺便提到数据类型（特别是用户定义类型）在文献中有时称为**抽象数据类型**或者 ADT，这是为了强调类型必须和它们的物理表示分开。这里并不使用这个术语，因为它可能会让人误以为有些类型不是抽象的，我们强调类型必须永远与其物理表示分开。

### 1. 标量与非标量

任何给定类型可以是标量（scalar）的，也可以是非标量（nonscalar）的。

- 一个**非标量**的类型是其值可以显式定义为一组用户可见的和可直接访问的分量（component）的类型。从这个意义上说，关系类型（见第 6 章）是非标量的，因为关系有元组和属性作为用户可见的分量。（更进一步，元组类型依次也是非标量的，因为元组有属性值作为用户可见的分量。）
- 一个**标量**的类型就是不是非标量的类型。*注意*：标量有时被说成是**封装的**，有时说成是**原子的**。原子的说法通常用于关系的上下文中（包括本书的早期版本）。关于封装，参照第 25 章。

类型 $T$ 的值是标量或者非标量取决于 $T$ 是标量或者非标量；于是，一个非标量的值有一组用户可见的分量，而一个标量的值则没有。类似的结论适用于变量、属性、参数和通常的表达式，细节已作更正。

### 2. 可能表示，选择子操作符和 THE _ 操作符

假定 $T$ 是标量的类型，则类型 $T$ 的值的物理表示是不为用户所见的。实际上，这样的表示可以相当复杂——特别是，它可以有分量——但是这样的分量是用户不可见的。然而，我们要求类型 $T$ 的值至少有一个**可能表示**[○]（作为类型 $T$ 定义的一部分声明），而且这样的可能表示是用户可见的，特别是，它们有用户可见的分量。要理解，这里的分量不是类型的分量，它们是可能表示的分量——从这个意义上，类型仍然是标量的。为了证明，我们考虑用户定义类型 QTY（“quantity”），它在 **Tutorial D** 中的定义如下：

```
TYPE QTY POSSREP { INTEGER } ;
```

这个类型定义说明，quantity 可能表示为整数。这样，已声明的可能表示“possrep”能够有用户可见的分量——实际上，它确实有这样的类型 INTEGER，但是 quantity 本身没有。

这里还有另一个例子来证明这一点：

```
TYPE POINT      /* geometric points in two-dimensional space */
     POSSREP CARTESIAN { X RATIONAL, Y RATIONAL }
     POSSREP POLAR { R RATIONAL, θ RATIONAL } ;
```

---

○ 更准确地说，运行时发生类型错误不会导致赋值操作失败。这里，我们有充分理由假定，这个系统会进行“静态”类型检查或者在编译时进行类型检查；很明显，如果编译时刻检查成功，那么在运行时错误就不会发生。

○ 除非类型 $T$ 是虚构类型（见第 20 章）。

类型 POINT 有两个不同的可能表示 CARTESIAN 和 POLAR，这表明，二维空间的几何点可以用笛卡尔坐标或极坐标来表示。每一种可能表示有两个分量，这两个分量都属于类型 RATIONAL[⊖]。特别注意，类型 POINT 本身仍然是标量的，它没有用户可见的分量。

*语法*：我们约定，如果给定的类型 *T* 有一种没有具体名字的可能表示，那么这个可能表示默认命名为 *T*。我们还约定，如果给定的可能表示 *PR* 有一个没有具体名字的分量，那么这个分量名字默认为 *PR*。此外，每一个 POSSREP 的声明都可能引起下面几种操作符的自动定义：

- **选择子**（selector）操作符，它允许用户通过给定可能表示的每个分量的值，指定或选择类型的一个值。
- **THE_操作符**（对应于每个可能表示的分量）的集合。用户可以利用它来获得类型的值的可能表示的分量。

*注意*：当我们提到 POSSREP 声明会引起这些操作符的“自动定义”，我们指的是负责实现类型的系统或者用户同样要负责实现这些操作符。

例如，这里有针对类型 POINT 调用选择子和 THE_操作符的几个例子：

```
CARTESIAN ( 5.0, 2.5 )
/* selects the point with x = 5.0, y = 2.5 */

CARTESIAN ( X1, Y1 )
/* selects the point with x = X1, y = Y1, where */
/* X1 and Y1 are variables of type RATIONAL     */

POLAR ( 2.7, 1.0 )
/* selects the point with r = 2.7, θ = 1.0 */

THE_X ( P )
/* denotes the x coordinate of the point in */
/* P, where P is a variable of type POINT   */

THE_R ( P )
/* denotes the r coordinate of the point in P */

THE_Y ( exp )
/* denotes the y coordinate of the point denoted  */
/* by the expression exp (which is of type POINT) */
```

注意，(a) 选择子与相应的可能表示有相同的名字。(b) THE_操作符形如 THE_*C*，其中 *C* 表示相应可能表示的相应分量的名字。注意，选择子（或者说，选择子调用）是对大家所熟悉的**文字**（literal）的概括（所有的文字都是选择子调用，但并非所有的选择子调用都是文字，实际上，只有选择操作符中所有的自变量依次都是文字的，这个选择子调用才是文字的）。

为了搞清楚前面所说的在实际系统中如何工作，假设点的物理表示用笛卡尔积坐标（尽管物理表示没有必要用所讲的任何一种方式来表示）。于是系统提供某种受高度保护的操作符——下面用斜体的伪代码表示——用以有效地表现物理表示，并且，类型的实现者会用这些操作符来实现必要的 CARTESIAN 和 POLAR 选择子。（显然，类型实现者是一般规则的例外。这个规则是，用户涉及不到物理表示。）例如：

```
OPERATOR CARTESIAN ( X RATIONAL, Y RATIONAL ) RETURNS POINT ;
   BEGIN ;
      VAR P POINT ;     /* P is a variable of type POINT */
      X component of physical representation of P  :=  X ;
      Y component of physical representation of P  :=  Y ;
      RETURN ( P ) ;
   END ;
END OPERATOR ;

OPERATOR POLAR ( R RATIONAL, θ RATIONAL ) RETURNS POINT ;
   RETURN ( CARTESIAN ( R * COS ( θ ), R * SIN ( θ ) ) ) ;
END OPERATOR ;
```

---

⊖ **Tutorial D** 使用更精确的 RATIONAL 来代替熟悉的 REAL。顺便说一句，RATIONAL 可能更像一个内置类型的例子，而不是一个已声明的可能表示。比如说，530.00 和 5.3E2 可能都表示同一个 RATIONAL 值，也就是说，它们可能使用两个不同的 RATIONAL 选择子操作符的不同但等价的调用（见随后的讨论）。

在上面的代码中，POLAR 的定义用到了 CARTESIAN 选择子，也用到了（假设是内置的）操作符 SIN 和 COS。POLAR 的定义也可以直接用那些受保护的操作符来表示，如下：

```
OPERATOR POLAR ( R RATIONAL, θ RATIONAL ) RETURNS POINT ;
   BEGIN ;
      VAR P POINT ;
      X component of physical representation of P
                                        :=  R * COS ( θ ) ;
      Y component of physical representation of P
                                        :=  R * SIN ( θ ) ;
      RETURN ( P ) ;
   END ;
END OPERATOR ;
```

类型实现者也会用那些受保护的操作符实现必要的 THE_ 操作符，如下所示：

```
OPERATOR THE_X ( P POINT ) RETURNS RATIONAL ;
   RETURN ( X component of physical representation of P ) ;
END OPERATOR ;

OPERATOR THE_Y ( P POINT ) RETURNS RATIONAL ;
   RETURN ( Y component of physical representation of P ) ;
END OPERATOR ;

OPERATOR THE_R ( P POINT ) RETURNS RATIONAL ;
   RETURN ( SQRT ( THE_X ( P ) ** 2 + THE_Y ( P ) ** 2 ) ) ;
END OPERATOR ;

OPERATOR THE_θ ( P POINT ) RETURNS RATIONAL ;
   RETURN ( ARCTAN ( THE_Y ( P ) / THE_X ( P ) ) ) ;
END OPERATOR ;
```

从上面可看到，THE_R 和 THE_θ 的定义利用了 THE_X 和 THE_Y，同时用到了 SQRT 和 ARCTAN 操作符（假定是内置的）。同样，THE_R 和 THE_θ 也能直接通过受保护的操作符来定义（详细的实现作为一个练习）。

关于 POINT 的例子就到此为止。然而，所有上面讨论的内容也可以应用于简单的类型[⊖]——例如 QTY，理解这一点很重要。这里有一些例子：

```
QTY ( 100 )

QTY ( N )

QTY ( N1 - N2 )
```

这里有几个有关 THE_操作符调用的例子：

```
THE_QTY ( Q )

THE_QTY ( Q1 - Q2 )
```

注意：我们假定在这些例子中，(a) *N*，*N*1 和 *N*2 都是 INTEGER 类型的变量，(b) *Q*，*Q*1 和 *Q*2 都是 QTY 类型的变量，(c) “-” 是多形态操作符，它适用于 INTEGER 和 QTY 类型。

特别强调的是，值都是有类型的，因此对“某种货物供货量是 100”的说法，严格来讲就是不正确的。这里的数量是类型 QTY 的一个值，而不是 INTEGER 的一个值。因此，对于供货量应该更准确地说是 QTY（100），而不是简单的 100。然而，在非正式的上下文中，不用如此精确，这样就用 100 替代 QTY（100），以图方便。注意，本书在供应商和零件数据库和供应商 - 零件 - 工程数据库中使用这样的缩写（见图 3-8 和图 4-5）。

下面给出最后一个关于类型定义的例子，

```
TYPE LINESEG POSSREP { BEGIN POINT, END POINT } ;
```

---

⊖ 特别包括内建类型，虽然（由于历史原因）相应的选择子和 THE_操作符可能背离了本节描述的句法和其他规则。更进一步的讨论请参见 [3.3]。

类型 LINESEG 表示线段。这个例子证明一种给定的可能表示当然可以利用用户定义的类型来定义，而不是像前面的例子那样只使用系统定义的类型（也就是说，用户定义类型也是真正的类型）。

最后注意，本小节关于可能表示和相关问题的例子都特别包含标量类型，但是非标量类型也有可能表示。这个问题在5.6节说明。

## 5.4 类型定义

**Tutorial D** 中的新类型既可以通过前面章节的例子提到的 TYPE 语句定义，也可以使用类型生成子定义。本书在5.6节具体说明类型生成子以及相关的如何定义非标量类型的问题。在这一节，我们仔细讨论 TYPE 语句。首先以标量类型 WEIGHT 的定义为例：

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

*解释*：WEIGHT 用5位十进制数表示，精确到小数点后一位，这个数的大小大于0，小于5000。*注意*：上面的语句为 WEIGHT 类型定义了**类型约束**（type constraint）。通常，类型 *T* 的类型约束，准确地说，就是对类型 *T* 的一组数值的定义。如果给定的 POSSREP 声明没有包含显式的约束说明，那么就是默认的约束。（例中，即使忽略掉 CONSTRAINT 约束，WEIGHT 类型的有效值必须是最多5位十进制数，精确到小数点一位。）

WEIGHT 的例子还涉及到了另一点，在第3章3.9节我们说到部分重量是以磅计算。但是将类型本身和单位捆绑在一起并不是一个好主意（这里的术语单位指的是度量单位）。根据文献［3.3］，我们允许用户将重量按照磅或者按照克来度量，提供不同的可能表达式，这样：

```
TYPE WEIGHT
     POSSREP LBS { L DECIMAL (5,1)
                   CONSTRAINT L > 0.0 AND L < 5000.0 }
     POSSREP GMS { G DECIMAL (7,1)
                   CONSTRAINT G > 0.0 AND G < 2270000.0
                              AND MOD ( G, 45.4 ) = 0.0 } ;
```

*注意*：POSSREP 声明包括约束说明，而且这两个约束说明逻辑等价（MOD 是一种将两个数字操作数进行相除取余数的操作；我们简化问题，假定一磅等于454克）。给出这样的定义：

- 如果 W 是 WEIGHT 类型的表达式，那么 THE_L(W)返回一个十进制值(5,1)，表示相应重量的磅数。当 THE_G(W)返回一个十进制值(7,1)，表示相应重量的克数。
- 如果 N 是十进制(5,1)类型的表达式，那么 LBS(N)和 GMS(454 * N)将返回同样的 WEIGHT 值。

这里，我们来看在 **Tutorial D** 语法中定义一个标量类型：

```
<type def>
    ::=   TYPE <type name> <possrep def list> ;

<possrep def>
    ::=   POSSREP [ <possrep name> ]
                { <possrep component def commalist>
                        [ <possrep constraint def> ] }

<possrep component def>
    ::=   [ <possrep component name> ] <type name>

<possrep constraint def>
    ::=   CONSTRAINT <bool exp>
```

从语法中可以得出（大部分可以从两个 WEIGHT 例子中得到证明）：

1）语法利用了 *list* 和 *commalist*。*commalist* 术语的定义见第4章（4.6节）：*list* 术语的定义近似为，<*xyz*>表示为任意符合造句法的类别（也就是，出现在 BNF 产生规则左边的任何东

西），那么表达式 <*xyz list*> 表示零个或者多个 <*xyz*> 的序列，每一对邻近的 <*xyz*> 之间用一个或多个空白分开。

2）<*possrep def list*> 必须包含至少一个 <*possrep def*>。<*possrep component def commalist*> 必须包含至少一个 <*possrep component def*>。

3）括号“[ ]”表示括号里面的内容是可选的（就像标准的 BNF 符号）。相对应的，大括号“{ }”代表内容本身，也就是它们都是在语言中已经定义好了的符号，不是元语言符号。精确地说，如果 *commalist* 要表示一组项，那么使用大括号来封装 *commalist*（这意味着这些项的顺序不重要，而且没有一项出现次数超过一次）。

4）通常，<*bool exp*>（“布尔表达式”）表示一个真值（真或假）。<*bool exp*> 不会用变量，但是 <*possrep def*> 中的 <*possrep component name*> 用于表示所讨论的标量类型的任意值的适合可能表达式的相应组件。注意：布尔表达式也称为条件、真值或者逻辑表达式。

5）<*type def*> 和物理表示毫无关系。更准确地说，这样的表示必定被指定为概念模式/内模式间的映像（见 2.6 节）。

6）定义一个新的类型，系统必定在目录中建立一个条目来描述这个新类型（如果需要刷新存储器来查看这个目录，参照 3.6 节）。类似的说明也适用于操作符定义（见 5.5 节）。

下面是供应商和零件数据库中使用的标量类型的定义（除了已讨论过的 WEIGHT 类型）。出于简化，忽略了约束说明。

```
TYPE S#     POSSREP { CHAR } ;
TYPE NAME   POSSREP { CHAR } ;
TYPE P#     POSSREP { CHAR } ;
TYPE COLOR  POSSREP { CHAR } ;
TYPE QTY    POSSREP { INTEGER } ;
```

（回顾第 3 章，供应商 STATUS 属性和供应商和零件 CITY 属性都是根据内置类型定义而不是用户定义类型，所以这些属性没有对应的类型定义。）

当然，如果不再使用这个类型，就要删除这个类型：

```
DROP TYPE <type name> ;
```

这个 <*type name*> 必须定义为用户自定义类型，而不是内置类型。这个操作导致系统目录中描述这个类型的条目被删除，那么这个类型不再被系统所知。简单地说，如果这个类型还在某处使用——特别是某个关系变量定义成这个类型，那么我们假定 DROP TYPE 失败。

最后指出，定义类型的操作实际上不能产生相应的一组值，理论上，这些值已经存在，而且一直存在（比如说 INTEGER 类型）。这样，所有的“定义类型”操作——比如 **Tutorial D** 中的 TYPE 语句——真正做的就是给这组值赋予名字。同样地，DROP TYPE 语句实际不能删除相应的值，它主要删除相应的 TYPE 语句所产生的名字。

## 5.5　操作符

到此为止，我们在这一章看到的所有操作符定义要么是选择子操作符，要么是 THE_操作符；现在我们将了解一下通常的操作符定义。第一个例子是用户定义操作符 ABS，用于用户定义类型 RATIONAL 中：

```
OPERATOR ABS ( Z RATIONAL ) RETURNS RATIONAL ;
   RETURN ( CASE
               WHEN Z ≥ 0.0 THEN +Z
               WHEN Z < 0.0 THEN -Z
            END CASE ) ;
END OPERATOR ;
```

操作符 ABS（absolute value）定义成只有一个参数 Z，类型为 RATIONAL，返回一个相同类型的值。这样，ABS 的调用——比如 ABS（AMT1 + AMT2）——根据定义，是一个 RATIONAL 类型的表达式。

下一个例子，DIST（distance between）有两个参数，一个是用户定义类型 POINT，返回另

一个结果 LENGTH：

```
OPERATOR DIST ( P1 POINT, P2 POINT ) RETURNS LENGTH ;
   RETURN ( WITH THE_X ( P1 ) AS X1 ,
                 THE_X ( P2 ) AS X2 ,
                 THE_Y ( P1 ) AS Y1 ,
                 THE_Y ( P2 ) AS Y2 :
            LENGTH ( SQRT ( ( X1 - X2 ) ** 2
                          + ( Y1 - Y2 ) ** 2 ) ) ) ;
END OPERATOR ;
```

假设 LENGTH 选择子带有一个 RATIONAL 类型的参数。注意 WITH 子句的使用作为某种子表达式的结果的名称。在这一章的后面部分会提到这个结构的使用。

接下来的例子是类型 POINT 的“=”（相等[⊖]）比较操作符：

```
OPERATOR EQ ( P1 POINT, P2 POINT ) RETURNS BOOLEAN ;
   RETURN ( THE_X ( P1 ) = THE_X ( P2 ) AND
            THE_Y ( P1 ) = THE_Y ( P2 ) ) ;
END OPERATOR ;
```

RETURN 语句的表述中对类型 RATIONAL 使用了系统内置操作符“=”。为简单起见，假定符号“=”可以用在等价的操作符中（可针对所有类型，而不仅仅是 POINT）；我们不考虑这样的插入符号在实际中怎样说明，因为这只是一个语法的问题。

下面是类型 QTY 的“>”操作符：

```
OPERATOR GT ( Q1 QTY, Q2 QTY ) RETURNS BOOLEAN ;
   RETURN ( THE_QTY ( Q1 ) > THE_QTY ( Q2 ) ) ;
END OPERATOR ;
```

RETURN 语句的表述中对类型 INTEGER 使用了系统内置操作符“>”。同样，假定这一操作符用于所有可排序的类型上，而不只是类型 QTY（实际上，从定义看，**可排序的类型**就是能应用“>”的类型。不能排序的类型的简单例子是类型 POINT）。

最后是一个关于更新操作符定义的例子（前面所有的例子中都是只读操作符，除了局部变量，这些操作符不能更新其他任何变量。[⊖] 可以看到，定义中包含了一个 UPDATES 说明，而不是 RETURNS 说明；更新操作符没有返回值，并且必须被显式的 CALL 调用 [3.3]。

```
OPERATOR REFLECT ( P POINT ) UPDATES P ;
   BEGIN ;
      THE_X ( P )  :=  - THE_X ( P ) ;
      THE_Y ( P )  :=  - THE_Y ( P ) ;
      RETURN ;
   END ;
END OPERATOR ;
```

REFLECT 操作符能有效地移动笛卡尔坐标的点 $(x, y)$ 到相反的位置 $(-x, -y)$；它通过适当修改点的参数来实现。注意这个例子中“**THE_伪变量**”的使用。THE_伪变量是 THE_操作符在目标位置的一个调用（特别地，在一个赋值的左边）。这样的一个调用实际上是指定——优于简单地返回值——参数的特定分量（应用的可能表示）。例如，在 REFLECT 的定义中，下面的赋值实际上是根据参数 P 给参数变量的分量 X 赋了一个值：

```
THE_X ( P )  :=  ... ;
```

当然，任何被更新操作符修改的参数变量（特别是给 THE_伪变量赋值）必须特别地作为一个变量指定，而不是作为一般的表达式。

伪变量可以嵌套，例如：

---

⊖ “相等”（equality）操作符比一致（identity）要好一些，因为当且仅当 *v1*、*v2* 有相同的值时 *v1* = *v2*。

⊖ 只读和更新操作符也称为观察者和存取器，对应于对象系统（见第 25 章）。函数是只读操作符的另一个同义词（本书偶尔会用到）。

```
VAR LS LINESEG ;
THE_X ( THE_BEGIN ( LS ) )  :=  6.5 ;
```

THE_伪变量实际在逻辑上是不必要的。考虑下面这个赋值：

```
THE_X ( P )  :=  - THE_X ( P ) ;
```

这个赋值使用伪变量，逻辑上等价于下面不使用伪变量的赋值语句：

```
P  :=  CARTESIAN ( - THE_X ( P ), THE_Y ( P ) ) ;
```

同样地，

```
THE_X ( THE_BEGIN ( LS ) )  :=  6.5 ;
```

逻辑上等价于

```
LS  :=  LINESEG ( CARTESIAN ( 6.5,
                              THE_Y ( THE_BEGIN ( LS ) ) ) ,
                  THE_END ( LS ) ) ;
```

总之，严格地说，伪变量本身对于支持某种组件层次的更新来说并不必要。然而，伪变量方法比替换方法（可认为是一种简略表达方式）更直观。此外，它在相应的选择子操作符的语法改变上提供更高程度的不透性。（它的有效实现也更容易一些。）

考虑简略表达方式的时候，只有只读更新操作符在逻辑上是必要的，它也就是实际中的赋值（：=）。其他的更新操作符可以根据赋值定义（实际上我们已经从第3章，关系更新操作符的例子中了解了这一点）。然而，我们要求支持一个**多形式赋值**，允许任何数目的单独赋值“同时”实现［3.3］。比如，通过下面的多赋值来替换 REFLECT 操作符定义中的两个赋值：

```
THE_X ( P )  :=  - THE_X ( P ) ,
THE_Y ( P )  :=  - THE_Y ( P ) ;
```

（注意逗号分隔符）。语义如下：首先，所有的右边源表达式被计算；然后，所有单独赋值依次执行。[⊖] 注意：既然多赋值被认为是一个单一操作，就不需要在赋值过程中进行完整性检查；这才是我们为什么把对多赋值的支持放在第一位的真正原因。第9章和第16章有更详细的讨论。

最后，如果一个操作符不再被使用，必须能够删除掉，例如：

```
DROP OPERATOR REFLECT ;
```

被删除的操作符必须是用户定义的，而不是系统内建的。

**1. 类型转换**

再次考虑下面的类型定义：

```
TYPE S# POSSREP { CHAR } ;
```

默认情况下，这里的可能表示为名字 S#，相应的选择子操作符同样如此。因此，下面是一个有效的选择子调用：

```
S# ('S1')
```

（它返回某个供应商号码）。注意，选择子 S#因此可被视为把字符串转换为供应商号码的**类型转换符**；类似地，选择子 P#可被视为把字符串转换为零件号码的类型转换符；QTY 选择子可被视为把整数转换为数量的类型转换符；等等。

---

⊖ 这个定义在有两个或者多个单独的赋值指向同一个目标变量的情况下需要一些改进。这个细节超出了本书的范围，假如要满足这些赋值给出期望的结果，不同的单独赋值应更新同一个目标变量的不同部分（一个重要的特例）。

THE_操作符可被视为执行反方向类型转换的操作符。比如，回顾5.4节开始部分中类型WEIGHT的定义：

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

如果W是WEIGHT类型，那么表达式

```
THE_D ( W )
```

有效地将W表示的重量转化成DECIMAL(5,1)数量。

现在，在5.2节中看出（a）赋值中的源和目标都是同一类型，（b）相等比较中的被比较数必须是同一类型。在一些系统中，这些规则并没有直接强制执行；系统中可能会有这样的比较，比如说一个零件号码和一个字符串的比较，在WHERE子句中，像这样：

```
... WHERE P# = 'P2'
```

这里左边的比较字是P#类型，右边的比较字是CHAR类型；表面上这个比较会因为**类型错误**失败（编译时间出现类型错误）。理论上，系统会意识到可以使用P#的转换操作符（换句话说就是P#选择子）将CHAR类型的比较字转换成P#类型，这样重写比较如下：

```
... WHERE P# = P# ('P2')
```

这样的比较才算是有效。

隐式调用这个转换操作符被称为**强制**（coercion）执行。然而，强制执行会导致程序bug。因为这个原因，我们调整转换位置，本书中不允许强制执行——操作数必须是适当的类型，不是强制的某些类型。当然，允许在必要的时候定义和调用类型转换操作符（或者通常称为“CAST”操作符），比如：

```
CAST_AS_CHAR ( 530.00 )
```

正如前面指出，选择子（至少有一个参数）被认为是一种转换操作符。

现在，你已经意识到这里讨论的内容在编程语言中称为**强类型**。不同作者对于这个术语的定义会稍有不同；这里，它指的是（a）每个值都有一个类型，（b）当完成一个操作的时候，系统检查操作数是不是这个操作的正确类型。比如，考虑下面的表达式：

```
WEIGHT + QTY    /* e.g., part weight plus shipment quantity  */
WEIGHT * QTY    /* e.g., part weight times shipment quantity */
```

第一个表达式没有意义，系统会拒绝它。另一方面，第二个有意义，它表示所有发货零件的总重量。所以定义的重量和数量操作符应该包含 * 而不是 + 。

这里有几个例子，包含比较操作：

```
WEIGHT > QTY
EVEN > ODD
```

（第二个例子中假定EVEN是EVEN_INTEGER类型，ODD是ODD_INTEGER类型，语义明显。）这样，第一个表达式没有意义，但是第二个表达式就有意义。因此定义的重量和数量的操作符不能包括“>”，但是对于偶数和奇数来说就可以。[⊖]（对于在什么类型上什么操作符有效，我们参考历史上大部分数据库文献——也包括这本书的早期版本——只考虑比较操作符而忽视其他的操作符，比如 + 和 * 。）

---

⊖ 实际上EVEN_INTEGER和ODD_INTEGER都是INTEGER的子类型，“>”操作符是从后面的类型中继承过来的（见第20章）。

### 2. 小结

这一节介绍的对类型的完全支持蕴涵了很多重要的东西，现简要地概括如下：

- 首先且最重要的是，它意味着系统（a）确切知道**哪一个表达式是合法的**，（b）知道合法表达式的**结果的类型**。
- 它也意味着一个给定的数据库中所有类型的集合是一个**封闭集**——也就是说，每一个合法表达式的结果的类型对系统是可知的。特别地，如果比较式是合法的表达式，这个类型的封闭集必须包含布尔型或真值型！
- 特别地，由于系统知道每一个合法表达式的结果的类型，因此它知道哪一个**赋值**是合法的，也知道哪一个**比较**是合法的。

最后说明一点，我们已经知道域是一个任意复杂的数据类型，它可以是系统定义的，也可以是用户定义的，并且它的值只能被定义在有关类型上的操作符操作（并且它的物理表示对用户是隐藏的）。而对**对象**系统而言，可以看出最基本的对象概念（即对象类）就是一种由系统或用户定义的任意复杂的数据类型，其值只能通过定义在有关类型上的操作符操作（并且它的物理表示对用户是隐藏的）……换句话说，域和对象类是同样的事情！这是把这两项技术（关系和对象）结合在一起的关键。第 26 章将详细论述这一重要问题。

## 5.6　类型生成子

现在考虑那些不用 TYPE 语句定义而是用某种**类型生成子**调用获得的类型。抽象地说，类型生成子是某种操作符，因为它返回类型而不是简单的标量值。例如，在常规的编程语言中是这样表示：

```
VAR SALES ARRAY INTEGER [12] ;
```

来定义一个变量 SALES，它的有效值是 12 个整数的一维数组。在这个例子中，表达式 ARRAY INTEGER[12]可被视为 ARRAY 类型生成子的调用，它返回一个特殊的数组类型。这个特殊的数组类型就是类型生成子。在此有几点需要指出：

1）文献中类型生成子有多个名字，包括类型构造器，参数化类型，多形态类型，类型模板和通用类型。这里采用术语类型生成子。

2）生成的类型是真正的类型，能够用在任何普通的“非生成”类型使用的地方。比如，定义某种关系变量，使它包含 ARRAY INTEGER[12]类型的属性。对应地，这样的类型生成子就不是类型。

3）大多数生成的类型（尽管不是所有）都是非标量类型（数组类型就是这样的例子）。5.4 节中已经介绍了如何定义非标量类型。注意：不直接调用某些类型生成子也可以定义非标量类型，这里我们不进一步考虑这样的情况。

4）为清晰起见，生成的类型被认为是系统定义类型，因为它们是通过调用系统定义的类型生成子获得的。注意：实际这里过于简化了，尤其是没有排除用户定义自己的类型生成子的可能，在本书中不考虑这种可能。

现在，生成的类型有可能表示（简称“possrep”），这个可能表示来自以下方式：（a）应用于类型生成子的一般的 possrep；（b）特定的类型生成子的用户可见的组件的特殊 possrep。在 ARRAY INTEGER[12]的例子中：

- 通常有一些通用 possrep 用于一维数组，比如说相邻的数组元素通过从低到高的下标来定义（在本例中低和高是指 1 和 12）。
- 用户可见的组件就是指刚刚提到的 12 个数组元素，它们有定义成 INTEGER 类型的 possrep。它们都会有选择子和 THE_操作符的功能，比如这样的表达式

```
ARRAY INTEGER ( 2, 5, 9, 9, 15, 27, 33, 32, 25, 19, 5, 1 )
```

用于指定 ARRAY INTEGER [12] 类型的特定值（“选择子功能”）。同样地，表达式

```
SALES [3]
```

可以用于访问数组值的第三个组件（也就是第三个数组元素），这个组件是数组变量 SALES 的当前值（THE_操作符功能）。它也可以用作伪变量。

赋值和相等比较操作符也都适用。比如，这里是一个有效的赋值：

```
SALES  :=  ARRAY INTEGER (  2,  5,  9,  9, 15, 27,
                           33, 32, 25, 19,  5,  1 ) ;
```

这里是一个有效的等值比较：

```
SALES  =  ARRAY INTEGER (  2,  5,  9,  9, 15, 27,
                           33, 32, 25, 19,  5,  1 )
```

注意：任何给定的类型生成子都包含一组通用的约束和相关的操作符（通用指约束和操作符都应用于任何通过类型生成子得到的特殊类型）。比如，在 ARRAY 类型生成子中：

- 存在一个通用的约束是低的边界值不会超出高的边界值。
- 存在一个通用的“反向”操作符，能够将任意一维数组作为输入，返回另一组数组，这个数组包含同样的元素，但是顺序完全相反。

（实际上，选择子操作符、THE_操作符、赋值操作符、等值比较操作符也是从某种通用的操作符中继承来的。）

最后注意，在关系世界中还有两个特别重要的类型生成器是 TUPLE 和 RELATION。在下一章会详细介绍。

## 5.7 SQL 支持

**内置类型**

SQL 提供以下或多或少自解释的内置类型：

```
BOOLEAN                          NUMERIC (p,q)     DATE
BIT [ VARYING ] (n)              DECIMAL (p,q)     TIME
BINARY LARGE OBJECT (n)          INTEGER           TIMESTAMP
CHARACTER [ VARYING ] (n)        SMALLINT          INTERVAL
CHARACTER LARGE OBJECT (n)       FLOAT (p)
```

SQL 还支持一些默认值、缩写和替换拼写，比如说 CHAR 替代 CHARACTER，CLOB 替代 CHARACTER LARGE OBJECT，BLOB 替代 BINARY LARGE OBJECT。细节不多说，有几点需要指出：

1）BIT 和 BIT VARYING 都是在 SQL：1992 标准中加入的，SQL：2003 中去掉了。

2）不管它们的名字怎样，（a）CLOB 和 BLOB 都是串类型（它们和第 25 章中的对象没有关系）；（b）BLOB 是一个字节或者“八位字节”串类型（它和二进制数没有关系）。还有，既然这样类型的值会非常大——有时也会被指向长串类型——SQL 提供一个叫 locator 的构造器允许逐字节访问这个值。

3）赋值和等值比较操作符对所有的类型都适用。等值比较本质上很直观（详见第 5 点）。赋值语句像这样：

```
SET <target> = <source> ;
```

当然，当数据库执行更新和恢复时，同时也隐式执行了赋值操作。然而，关系赋值并不被支持。多赋值也不被支持，除了像下面这样，[⊖] 如果 *r* 行通过下面形式语句更新：

```
UPDATE T SET C1 = exp1, ..., Cn = expn WHERE p ;
```

---

⊖ 进一步有两个例外在 9.12 节和 10.6 节。除了这些例外，现在没有产品支持多赋值。但我们还是期望和要求这样的支持，计划在 SQL：2003 中会实现。

（此处 *r* 是 *T* WHERE *p* 的结果中的一行），所有表达式 *exp1*，…，*expn* 在每一个 *C1*，…，*Cn* 赋值执行前计算。

4）SQL 支持强类型，但是支持是有限的。在内置类型上使用特定的分类法，可以分成如下 10 个不相交的类别：

- 布尔值　■ 日期　■ 位串　■ 二进制
- 字符串　■ 数字　■ 时间　■ 时间戳
- 年/月间隔　■ 天/时间间隔

类型检查基于以上 10 个类别。这样，例如数字和字符串这样的比较是非法的；而比较两个数字则是合法的，即使这些数字属于不同的数字类型，比如说 INTEGER 和 FLOAT。（在比较之前 INTEGER 会强制转化成 FLOAT。）

5）对于字符串类型——CHAR(*n*)，CHAR VARYING(*n*)和 CLOB(*n*)——类型检查规则非常复杂，细节超出了本书的范围，但是我们要简要描述固定长度的字符串类型（比如，CHAR(*n*)类型）：

- 比较：如果 CHAR(*n1*)和 CHAR(*n2*)类型的值相比较，短的字符串先在右边填上空格，使其长度与长的字符串长度一样。[⊖] 比如，“P2”（长度为 2）和“P3”（长度为 3）相等。
- 赋值：如果 CHAR(*n1*)类型的值被赋给 CHAR(*n2*)类型的变量，那么，在赋值之前如果 *n1* < *n2*，CHAR(*n1*)值在右边用空格填充，或者如果 *n1* > *n2* 就在右边截去达到 *n2* 长度。如果任何非空字符在截除的过程中丢失，就会出错。

进一步的解释和讨论参看［4.20］。

1. DISTINCT 类型

SQL 支持两种用户定义类型，DISTINCT 类型和结构类型，这些类型都通过 **CREATE TYPE** 语句来定义。[⊖] 这一小节讨论 DISTINCT 类型，结构类型在下一节中讨论（DISTINCT 大写是为了强调这个单词不是用在一般的自然语言环境中）。下面是 DISTINCT 类型 WEIGHT 的 SQL 定义（比较和对照 5.4 节中这个类型的不同 **Tutorial D** 定义）：

```
CREATE TYPE WEIGHT AS DECIMAL (5,1) FINAL ;
```

最简单的形式（忽略所有的可选项）的语法如下：

```
CREATE TYPE <type name> AS <representation> FINAL ;
```

有几点要指出：

1）FINAL 说明在第 20 章解释。

2）< *representation* > 是另一个类型的名字（这个类型不是用户定义的或者生成的）。注意，根据这些规则，就不能定义 5.3 节中的 POINT 类型为 SQL DISTINCT 类型。

3）进一步注意，< *representation* > 不是指可能表示，而是实际的这个类型的物理表示。实际上，SQL 根本不支持 possrep 概念。这就使得它不可能定义 DISTINCT 类型——或者结构类型——的两个或多个不同的 possrep。

4）这里也存在着类似的 **Tutorial D** 约束说明。例如，在 WEIGHT 类型的例子中，没有指明对于每一个 WEIGHT 值，相应的 DECIMAL(5,1)必须大于 0 或小于 5000。

5）DISTINCT 类型的比较操作符适用于底层的物理表示。注意：除了赋值（见第 8 点），其他的适应于物理表示的操作符不适用于 DISTINCT 类型。比如，即使 WT 是 WEIGHT 类型，下面的表示也无效：

```
WT + 14.7    WT * 2    WT + WT
```

---

⊖ 我们假定 PAD SPACE 使用这个比较［4.20］。

⊖ 有时候 SQL 也支持域，但是 SQL 域和关系中的域没有任何关系，SQL 域的详细说明参看［4.20］。

6）支持选择子和THE_操作符。比如，如果NW是DECIMAL(5,1)类型的变量，那么表达式WEIGHT( :NW)返回相应的WEIGHT值；如果WT是一列WEIGHT类型，那么表达式DECIMAL(WT)返回相应的DECIMAL(5,1)值。[⊖] 下面是有效的SQL语句：

```
DELETE
FROM   P
WHERE  WEIGHT = WEIGHT ( 14.7 ) ;

EXEC SQL DELETE
         FROM   P
         WHERE  WEIGHT = WEIGHT ( :NW ) ;

EXEC SQL DECLARE Z CURSOR FOR
         SELECT DECIMAL ( WEIGHT ) AS DWT
         FROM   P
         WHERE  WEIGHT > WEIGHT ( :NW ) ;
```

7）强类型应用于DISTINCT类型，除了一个例外（见第8点）。注意，DISTINCT类型和底层表示类型的值的比较是不合法的。下面给出的是无效的SQL语句，即使NW是DECIMAL(5,1)类型：

```
DELETE
FROM   P
WHERE  WEIGHT = 14.7 ;            /* warning -- invalid !!! */

EXEC SQL DELETE
         FROM   P
         WHERE  WEIGHT = :NW ;    /* warning -- invalid !!! */

EXEC SQL DECLARE Z CURSOR FOR
         SELECT DECIMAL ( WEIGHT ) AS DWT
         FROM   P
         WHERE  WEIGHT > :NW ;    /* warning -- invalid !!! */
```

8）第7点中提到的例外和赋值操作符有关。比如，如果重新将一些WEIGHT值指定给DECIMAL(5,1)变量，需要一些类型转换。现在，可以明确实现这个转换，如下：

```
SELECT DECIMAL ( WEIGHT ) AS DWT
INTO   :NW
FROM   P
WHERE  P# = P# ('P1') ;
```

然而，下面这个也是合法的（并且会发生相应的转换）：

```
SELECT WEIGHT
INTO   :NW
FROM   P
WHERE  P# = P# ('P1') ;
```

类似的结论也适用于INSERT和UPDATE操作。

9）显式的CAST操作符可以用来定义在DISTINCT类型之间的转化，或向DICTINCT类型的转化，或者从DISTINCT类型的转化。这里不介绍细节。

10）可以根据需要定义一些附加的操作符（之后可删除）。注意，SQL用于操作符的术语是例程，例程分为三种：函数、过程和方法。（函数和过程对应只读和更新操作；方法像过程，但是以一种不同的动态方式调用。[⊜]）我们可以定义一个函数——多态函数——称为ADDWT（“增加重量”），允许两个值相加，不在乎它们是WEIGHT值还是DECIMAL(5,1)值，或者是两者的混合。所有下面的表示都是合法的：

---

⊖ 实际上DECIMAL(WT)在SQL：1999中无效，但是期望在SQL：2003中有效。它不能用作伪变量，不像**Tutorial D**中的THE_操作符。

⊜ 不像函数和过程，方法包含一些运行时绑定（见第20章）。注意：方法这个术语来自于面向对象领域，它的意思需要根据上下文来做调整（见第25章）。

```
ADDWT ( WT, 14.7 )
ADDWT ( 14.7, WT )
ADDWT ( WT, WT )
ADDWT ( 14.7, 3.0 )
```

更多关于SQL例程的内容参见［4.20］和［4.28］，进一步的细节已超出了本书的范围。

11）下面的语句用于删除用户定义类型：

```
DROP TYPE <type name> <behavior> ;
```

*<behavior>*或者是RESTRICT或者是CASCADE。RESTRICT的意思是如果类型在任何地方还有使用的话，DROP会失败；CASCADE的意思是DROP总是会成功，对于任何正在使用这种类型的函数引发隐式的DROP…CASCADE(!)

**2. 结构类型**

现在介绍结构类型。这里有两个例子：

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL ;

CREATE TYPE LINESEG AS ( BEGIN POINT, END POINT ) NOT FINAL ;
```

（实际上，BEGIN和END是SQL保留字，所以第二个例子不会成功。）创建结构类型的最简单语法（忽略各种可选说明）如下：

```
CREATE TYPE <type name> AS <representation> NOT FINAL ;
```

这里有几点要指出：

1）NOT FINAL说明在第10章介绍。*注意*：SQL：2003有望允许FINAL可选。

2）*<representation>*是用圆括号包装的*<attribute definition commalist>*，这里*<attribute>*包含*<attribute name>*和*<type name>*。注意，这里的“*attribute*”不是关系里面的属性，因为结构类型不是关系类型（见第6章）。此外，*<representation>*是实际的物理表示，不是某种可能表示。*注意*：类型设计者会通过操作符的设计和选择来有效地隐藏这个事实。比如，给定类型POINT的定义，系统自动提供操作符来表示笛卡尔表示（见第3点和第6点），但是类型设计者提供操作符来表示相反的表示。

3）每一个属性定义会引起两个相关的操作符自动定义（实际上是“方法”），一个是观察者，一个是增变者，它们提供的功能类似于**Tutorial D**中的THE_操作符。[⊖] 比如，如果LS，P和Z是LINESEG，POINT和FLOAT，相应地，下面的赋值语句有效：

```
SET Z = P.X ;                  /* "observes" X attribute of P */
SET P.X = Z ;                  /* "mutates" X attribute of P  */
SET X = LS.BEGIN.X ;           /* "observes" X attribute of   */
                               /* BEGIN attribute of LS       */
SET LS.BEGIN.X = Z ;           /* "mutates" X attribute of    */
                               /* BEGIN attribute of LS       */
```

4）没有和**Tutorial D**约束说明类似的问题。

5）比较操作符的定义通过单独的**CREATE ORDERING**语句指定。这里有两个例子：

```
CREATE ORDERING FOR POINT EQUALS ONLY BY STATE ;

CREATE ORDERING FOR LINESEG EQUALS ONLY BY STATE ;
```

EQUALS ONLY指的是=或者≠（或者<>，意思是不等于），它是结构类型的值的有效比较操作符。BY STATE的意思是这个类型的两个值会相等。在这种情况下，对于所有的$i$，第$i$

---

⊖ 从正确性角度看，SQL的增变不是传统术语上的增变（它们不是更新操作符），但是它们能够实现传统的增变功能。比如，“SET P.X = Z”（它其实并没有准确包含增变调用）是“SET P = P.X (Z)”的缩写。

个属性是相等的。其他的 CREATE ORDERING 说明超出了本书的范围；比如，“ > ”的语义也定义为结构类型。注意：如果给定的结构类型没有相关的“ordering”，那么这个类型值上也没有比较，甚至是**等值比较**。

6）不自动提供选择子，但是它们的作用同样实现。首先，SQL 自动提供构造器函数，但是这个函数的每次调用返回同样的值——这个类型的值的属性都有相应合适的默认值。[1] 比如，这个构造器函数调用

```
POINT ()
```

返回默认值为 *X* 和 *Y* 的点。现在，立即调用 *X* 和 *Y* 的增变操作符（见第 3 点）来得到希望从这个构造器函数调用能够得到的点。此外，将初始的“结构”和随后的“增变”捆绑在一起，形成一个单一的表达式，如下：

```
POINT () . X ( 5.0 ) . Y ( 2.5 )
```

这里有一个更复杂的例子：

```
LINESEG () . BEGIN ( POINT () . X ( 5.0 ) . Y ( 2.5 ) )
           . END   ( POINT () . X ( 7.3 ) . Y ( 0.8 ) )
```

注意：构造器函数调用加上 NEW 噪音词不会改变语义，比如：

```
NEW LINESEG () . BEGIN ( NEW POINT () . X ( 5.0 ) . Y ( 2.5 ) )
               . END   ( NEW POINT () . X ( 7.3 ) . Y ( 0.8 ) )
```

7）强类型适用于结构类型，除了在第 6 章 6.6 节介绍的情况外（见“结构类型”一节）。

8）前面提到的额外的操作符也能按要求定义（以及删除）。

9）结构类型和排序可以删除。这样的类型也能够改变，这可通过 ALTER TYPE 语句实现，比如，新的属性增加或者旧的属性减少（也就是说，表示会发生改变）。

在下一章将进一步关注 SQL 结构类型（见 6.6 节），还有第 20 章和第 26 章。

### 3. 类型生成子

SQL 支持三种类型生成子（SQL 术语为类型构造器）：REF、ROW 和 ARRAY。[2] 本节只讨论 ROW 和 ARRAY，REF 在第 26 章解释。这里是 ROW 的用法的例子：

```
CREATE TABLE CUST
     ( CUST# CHAR(3),
       ADDR  ROW ( STREET CHAR(50),
                   CITY   CHAR(25),
                   STATE  CHAR(2),
                   ZIP    CHAR(5) )
PRIMARY KEY ( CUST# ) ) ;
```

STREET，CITY，STATE 和 ZIP 是生成的行类型的字段。通常，这样的字段可以是任何的类型，包括其他的行类型。字段层次的引用使用点限制，比如下面的例子（语法是 < *exp* > . < *field name* >，这里的 < *exp* > 必须是行值的）：

```
SELECT CX.CUST#
FROM   CUST AS CX
WHERE  CX.ADDR.STATE = 'CA' ;
```

注意：CX 是一个相关性名称。相关性名称在第 8 章（8.6 节）具体讨论。这里简单提到

---

[1] 给定属性的默认值由相应的属性部分指定。如果没有显式指定这些值，默认值就为空。注意：由于超出本书的范围，如果这个属性的类型是行类型或者是用户定义类型（如 POINT），那么默认值必须为空。如果它是一个数组类型，那么它可以是空也可以为空值——指定为 ARRAY[ ]。比如，如果 BEGIN 和 END 都是空，构造器函数调用 LINESEG( ) 返回线段。

[2] SQL：2003 有可能添加 MULTISET。

SQL 要求准确的相关性名称用于字段引用，以达到避免某种语法混乱发生的目的。

现在这里有一个 INSERT 的例子：

```
INSERT INTO CUST ( CUST#, ADDR )
VALUES ( '666', ROW ( '1600 Pennsylvania Ave.',
                      'Washington', 'DC', '20500' ) ) ;
```

注意这个例子中的行值（row literal）（实际上，在 SQL 里没有行值，这个例子中的表示是一个行值构造器（row value constructor））。

下一个例子：

```
UPDATE CUST AS CX
SET    CX.ADDR.STATE = 'TX'
WHERE  CUST# = '999' ;
```

注意：实际上标准目前不允许字段层次上的更新，但是以后可能会变化。

ARRAY 类型生成子是类似的，这里是一个例子：

```
CREATE TABLE ITEM_SALES
     ( ITEM# CHAR(5),
       SALES INTEGER ARRAY [12],
       PRIMARY KEY ( ITEM# ) ) ;
```

通过 ARRAY 生成的类型总是一维的；这个特定的元素类型（例子中是 INTEGER）能够是除了数组类型以外的任何类型。[⊖] 让 $a$ 等于某种数组类型的值，那么 $a$ 能够包含任何数目 $n$ 的元素（$n \geqslant 0$），趋于但是不超过指定的上界（例子中是 12）。如果 $a$ 准确包含 $n$ 个元素（$n > 0$），那么这些元素引用为 $a[1], a[2], \cdots, a[n]$。表达式 CARDINALITY($a$)返回 $n$ 值。

这里给出使用 ITEM_SALES 表的几个例子。注意第二个例子中的数组值（array literal）（相当于一个数组值构造器）：

```
SELECT ITEM#
FROM   ITEM_SALES
WHERE  SALES [3] > 10 ;

INSERT INTO ITEM_SALES ( ITEM#, SALES )
VALUES ( 'X4320',
         ARRAY [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ) ;

UPDATE ITEM_SALES
SET    SALES [3] = 10
WHERE  ITEM# = 'Z0564' ;
```

本节结尾我们强调，赋值和等值比较操作符应用于 ROW 和 ARRAY 类型——除非 ROW 或者 ARRAY 类型包含一个未定义等值比较的元素类型，这样它也无法定义 ROW 或 ARRAY 类型。

## 5.8　小结

在本章中，我们对**数据类型**（也称为域或者类型）的严格概念进行了全面的介绍。类型是**一组值**：也就是说，这组值满足一定的类型约束（**Tutorial D** 中的 **POSSREP** 子句来指定，包括可选的约束说明）。每一个类型有相关的一组**操作符**（包含只读和更新操作符）来处理值和变量。类型可以是简单的也可以是复杂的，类型的值可以为数字、串、日期、音频数据、图、视频数据或者地理位置，等等。类型**约束操作**，就是一个给定操作上的操作对象必须符合操作上的正确类型（**强类型**）。强类型是一个非常好的设想，并且是在编译的时候（而不是在运行时）就能

---

⊖ 这个限制可能在 SQL：2003 中会被去除，在任何情况下，元素类型可以是一个行类型，而且这个行类型包括一些数组类型的字段，下面就是一个合法的变量定义：

```
VX ROW (FX INTEGER ARRAY [12]) ARRAY [12]
```

例如，VX[3].FX[5]指 VX 数组变量值的第 3 个元素，行的惟一的第 5 个元素。

捕捉到某些逻辑错误。注意对关系操作（尤其是连接、并等）来说，强类型有重要的含义，在第7章中将会讨论到。

我们也讨论**值**和**变量**之间的重要的逻辑区别，指出值是不能更新的。值和变量都可以类型化，可以是关系的属性、只读操作符、参数，甚或是任意复杂的表达式。

类型可以是**系统定义**的也可以是**用户定义**的，它们可以是**标量**的或者**非标量**的，标量类型没有用户可见的组件。（关系模型中最重要的非标量类型是关系类型，这个在下一章讨论）。我们严格区分类型和它的**物理表示**（类型是模型问题，物理表示是实现问题）。然而，我们规定每个标量类型至少有一个**可能**表示。每个可能表示产生一个**选择子**操作符和对应可能表示的每个分量的 THE_操作符（包括 THE_**伪变量**）的自动定义。我们支持显式的**类型转换**，但不支持隐含的**强制转换**。同时支持对标量类型定义任意数目的操作符，并在每个类型上定义**等值比较**操作符和（多）**赋值**操作符。

我们还讨论了**类型生成子**，它是返回类型的操作符（ARRAY 就是例子）。产生的类型的约束和操作符都来自于一般的与合适类型生成子相关的约束和操作符。

最后讨论了 SQL 类型特性。SQL 提供各种**内置**类型——BOOLEAN、INTEGER、DATE、TIME，等等（每一个有相关的操作符）——但是只支持这些类型相关的强类型的有限形式。SQL 也允许用户定义自己的类型，分为 **DISTINCT** 类型和**结构类型**（structured type）；也支持某种**类型生成子**（ARRAY 和 ROW，还有 REF）。我们根据本章前面的内容对这些 SQL 功能进行分析。

## 习题

5.1 列举赋值（:=）和等值比较（=）操作符的类型规则。

5.2 区分：

值和变量　　类型和表示　　物理表示和可能表示

标量和非标量　只读操作符和更新操作符

5.3 用自己的语言解释下面的概念：

| | |
|---|---|
| 强制 | 伪变量 |
| 生成类型 | 选择子 |
| 值 | 强类型 |
| 顺序类型 | THE_操作符 |
| 多态操作符 | 类型生成子 |

5.4 为什么逻辑上不需要伪变量？

5.5 给定一个有理数，定义一个操作符返回这个数的立方。

5.6 定义一个只读操作符，给定一个笛卡尔坐标为 $x$ 和 $y$ 的点，返回笛卡尔坐标为 $f(x)$ 和 $g(y)$ 的点，其中 $f$ 和 $g$ 是事先定义好的操作符。

5.7 重复习题5.6但是定义一个更新操作符。

5.8 给定标量类型 CIRCLE 的类型定义，它的选择子操作符和 THE_操作符如下：

a）定义一组只读操作符来计算直径、圆周和给定圆的面积。

b）定义一个更新操作符使得给定圆的半径扩大到两倍（也就是更新给定圆的变量，半径是以前的两倍）。

5.9 给出一些类型的例子，用于定义两个或多个不同的可能表示。你能想到一个例子，其中同一个类型的不同的可能表示有不同数目的组件吗？

5.10 给定一些类型的例子，根据第3章图3-6中给定的部门和雇员数据库，怎么使得它的目录扩展考虑用户定义类型和操作符？

5.11 目录关系变量本身定义在什么类型上？

5.12 给定一组合适的标量类型定义，用于供应商－零件－工程数据库（见图4-5）。不要试图写关系变量定义。

5.13 5.3节指出以下这种说法严格来讲不正确，比如说某一出货量是100（数量是 QTY 类型的值，不是 INTEGER 类型的值）。结果，图4-5相当庞大，如果它假装认为数量作为整数是正确的。对于习题5.12，给出正确的方式来表示图4-5中的各种标量值。

5.14　计算练习5.12，下面哪个标量表达式是合法的？对于合法的这个，表示结果的类型；对于其他，写出能够得到期望效果的合法表达式。

a) `J.CITY = P.CITY`

b) `JNAME || PNAME`

c) `QTY * 100`

d) `QTY + 100`

e) `STATUS + 5`

f) `J.CITY < S.CITY`

g) `COLOR = P.CITY`

h) `J.CITY = P.CITY || 'burg'`

5.15　有时候类型也是变量，像关系变量。比如，合法的雇员号可以从三位数字增长到四位，以适合商业扩展，这样需要更新“所有可能的雇员号”。讨论这一情况。

5.16　给出5.3节和5.4节的所有类似SQL的类型定义。

5.17　给出练习5.12的SQL解答。

5.18　在SQL中：

a）什么是DISTINCT类型？什么是一般的DISTINCT类型值？是否存在一个indistinct类型？

b）什么是结构类型？什么是一般的结构类型的值？是否存在一个非结构类型？

5.19　解释SQL中的术语：观察者（observer）、增变函数（mutator）以及构造器函数（constructor function）。

5.20　“=”操作符没有定义给定类型，计算的结果是什么？

5.21　类型是一组值，那么当组为空值的时候，这个类型为空值。构想一个这种类型的应用。

5.22　“SQL没有正式的行值（row literal）或者数组值（array literal）。”解释这个结论。

5.23　考虑5.7节中的“结构类型”定义的SQL类型POINT。这个类型有包含笛卡尔坐标X和Y的表示。如果把POINT类型改成极坐标R和θ的表示，会发生什么？

5.24　SQL COUNT和CARDINALITY操作符之间的区别是什么？注意，COUNT定义在第8章8.6节。

## 参考文献

[5.1]　J. Craig Cleaveland: *An Introduction to Data Types*. Reading, Mass.: Addison-Wesley (1986).

# 第6章 关 系

## 6.1 引言

在前一章里，我们已经概要地讨论了类型、值和变量的概念；现在将注意力转移到关系类型、值和变量上来。既然关系是从元组扩展而来的，因此我们将仔细分析元组的类型、值和变量。不过要注意的是，元组本身并不是最重要的，至少从关系的角度来看是这样。元组的重要性主要在于它们构成了通向关系的一个必要的铺垫。

## 6.2 元组

首先我们对元组给出一个准确的定义。给定一系列类型 $T_i(i = 1,2,\cdots,n)$，这些类型不必是独一无二的，在这些类型之上的一个元组值（简称元组）$t$ 是一个有序的三元组 $<A_i, T_i, v_i>$，其中 $A_i$ 是**属性名**，$T_i$ 是**类型名**，$v_i$ 是类型 $T_i$ 对应的**值**，并且：

- $n$ 的值是元组 $t$ 的**度**或者**数量**。
- 有序三元组 $<A_i, T_i, v_i>$ 是元组 $t$ 的一个**组件**。
- 有序对 $<A_i, T_i>$ 是元组 $t$ 的一个**属性**，由属性名 $A_i$ 唯一确定（仅当 $i = j$ 时 $A_i$ 与 $A_j$ 相同）。$v_i$ 的值是元组 $t$ 中属性 $A_i$ 的**属性值**，[⊖] 而 $T_i$ 则是相应的**属性类型**。
- 完整的属性集构成了元组 $t$ 的**表头**（heading）。
- 元组 $t$ 的**元组类型**取决于 $t$ 的表头，而表头和元组类型包含有相同的属性（相同的属性名和类型）和相同的度。**元组类型名**的精确定义如下：

```
TUPLE { A1 T1, A2 T2, ..., An Tn }
```

下面是一个示例元组：

| MAJOR_P# : P# | MINOR_P# : P# | QTY : QTY |
|---|---|---|
| P2 | P4 | 7 |

这里的属性名分别是 MAJOR_P#，MINOR_P#和 QTY；相应的类型名是 P#，P#和 QTY；相应的值分别是 P#（'P2'），P#｛'P4'｝和 QTY(7)（为简单起见，图中的这些值分别被简写为 P2，P4 和 7）。这个元组的度为 3，它的表头是：

| MAJOR_P# : P# | MINOR_P# : P# | QTY : QTY |
|---|---|---|

它的类型是：

```
TUPLE { MAJOR_P# P#, MINOR_P# P#, QTY QTY }
```

注意：非正式场合中常常省略元组表头中的类型名，而只写出属性名。所以，我们一般这样描述原来的元组：

| MAJOR_P# | MINOR_P# | QTY |
|---|---|---|
| P2 | P4 | 7 |

在 **Tutorial D** 中，刚才讨论过的元组通常被表示为如下形式：

---

⊖ 当然，属性名和属性本身在逻辑上是不同的。实际上这是经不起推敲的，我们经常使用表达式如"属性 $A_i$"，表示属性名是 $A_i$（我们在前一章中确实是这样做的）。

```
TUPLE { MAJOR_P# P#('P2'), MINOR_P# P#('P4'), QTY QTY(7) }
```

（元组选择子调用的例子见下一小节。）仔细观察如上表达式可发现，元组属性的类型由具体的属性值明确地确定（例如，属性 MINOR_P#的类型是 P#，这是由于其相应的属性值的类型是 P#）。

**1. 元组的性质**

元组满足多种重要的性质，所有这些性质都直接引发本节之后的许多定义。具体而言：

- 在每一个元组中，对于其中的每一个属性（适当的类型）只含有一个值。
- 一个元组中的组件没有从左到右的顺序。因为元组的定义已经规定了它包括一个组件的集合，而从数学角度来说集合中的元素是没有顺序的。
- 一个元组的每一个子集都是一个元组（同样，表头的每一个子集也是一个表头）。而且对于空子集来说，这两个性质也是正确的！参看下一段。

更术语化的描述：度为 1 的元组被称为是一元的，度为 2 的元组被称为是二元的，度为 3 的元组称为三元的（依此类推）；更一般的来说，度为 $n$ 的元组被称为 $n$ 元的。[⊖] 度为 0 的元组（比如，一个不包含任何组件的元组）被称为零元（nullary）的。这里我们概要地讨论最后这种可能性。比如，在 **Tutorial D** 中有一个零元的元组：

```
TUPLE { }
```

有时候为了强调一个度为零的元组中不包含任何组件，我们可以更明确地将之描述为“0 元组”。看起来 0 元组似乎在实际应用中没有太多用途（不包含组件）；然而实际上从概念上来说，0 元组是至关重要的。6.4 节会更多地描述这个问题。

**2. 元组类型生成子**

**Tutorial D** 中提供了一个元组类型生成子，这个生成子可以用于定义关系变量的属性以及一些元组变量。[⊖] 下面是后者的一个例子：

```
VAR ADDR TUPLE { STREET CHAR,
                 CITY   CHAR,
                 STATE  CHAR,
                 ZIP    CHAR } ;
```

调用元组类型生成子的一般形式为：

```
TUPLE { <attribute commalist> }
```

（每一个 < *attribute* > 包含一个 < *attribute name* > 和一个 < *type name* >，其中后者紧随在前者之后）。对元组类型生成子的一个具体调用产生出元组类型——例如，上面所示变量 ADDR 的定义就是一个生成的类型。

每一个元组类型都有一个相关联的元组选择子操作符。这里是一个对刚才定义的变量 ADDR 的元组类型调用选择子操作符的例子：

```
TUPLE { STREET '1600 Pennsylvania Ave.',
        CITY 'Washington', STATE 'DC', ZIP '20500' }
```

这样表达的元组可以给元组变量 ADDR 赋值，或者用另一个类型相同的元组测试其是否相等。尤其要注意，为使两元组有相同的类型，它们必须有相同的属性。也要注意，所给元组的类型可以是任何类型（它们可以是某个关系类型，也可以是其他的元组类型）。

**3. 元组操作符**

我们已经简单地提到了元组的选择子、赋值和相等比较操作符。不过，更详尽地阐述元组的语义很重要，因为后面的章节是以此为基础的。尤其是，下面的部分是在此基础上定义的：

---

⊖ 术语“$n$ 元组”有时用于代替“元组”（例如，我们说 4 元组、2 元组，等等），但是常常省掉前缀“$n$”。

⊖ 元组变量不是关系模型的一部分，在关系数据库中是不允许使用元组变量的。但是，支持关系模型的系统可能在应用中支持元组变量（即元组变量是应用的局部变量）。

- 所有的关系代数操作符（见第7章）
- 候选码（见第9章）
- 外码（见第9章）
- 函数和其他依赖（见第11～13章）

还有其他的。下面是准确的定义：

- **元组相等**：元组 *t*1 和 *t*2 **相等**（即 *t*1 = *t*2 为真）当且仅当它们有相同的属性 *A*1,*A*2,…,*An*，并且，对于所有的 *i*(*i* = 1,2,…,*n*)，*t*1 中 *Ai* 的值 *v*1 和 *t*2 中 *Ai* 的值 *v*2 相等。
- 更进一步——这一点看起来很明显，但需要说明一下——元组 *t*1 和 *t*2 是重复的，当且仅当它们是相等的（即事实上它们是相同的元组）。

由前面的定义立刻可以得到所有0元组相互间都重复的结论！因此，我们可以用术语0元代替“一个”0元组，事实上我们通常是这样做的。

也要注意比较操作符“<”和“>”不用于元组（元组类型不是普通的类型）。

除了前面所述，文献［3.3］提出了某些关系操作符的类似操作（将在第7章讨论）——元组投影、元组连接，等等。这些操作符大部分是自解释的；我们这里只举一个元组投影的例子（在应用中这个操作符可能最有用）。使变量 ADDR 如前面小节所定义，其当前值如下：

```
TUPLE { STREET '1600 Pennsylvania Ave.',
        CITY 'Washington', STATE 'DC', ZIP '20500' }
```

则元组投影操作：

```
ADDR { CITY, ZIP }
```

表示元组：

```
TUPLE { CITY 'Washington', ZIP '20500' }
```

我们也需要能从给定的元组中提取出属性值。如果 ADDR 如前所述，那么表达式

```
ZIP FROM ADDR
```

表示的值是

```
'20500'
```

**元组类型推论**：正如本节开始所定义的，称为模式的元组类型的一个重要优点是有利于确定任意元组表达式的结果类型。例如，再看一下这个元组投影：

```
ADDR { CITY, ZIP }
```

正如我们看到的，这个表达式通过从 ADDR 的当前值中“投影掉”属性 STREET 和 STATE 得到一个元组，准确表达为

```
TUPLE { CITY CHAR, ZIP CHAR }
```

类似的注释可应用于所有可能的元组表达式。

**包和解包**：考虑下面的元组类型

```
TUPLE { NAME NAME, ADDR TUPLE { STREET CHAR, CITY CHAR,
                                STATE  CHAR, ZIP  CHAR } }

TUPLE { NAME NAME, STREET CHAR, CITY CHAR,
                   STATE  CHAR, ZIP  CHAR }
```

我们分别用 *TT*1 和 *TT*2 表示这些元组类型。特别注意观察类型 *TT*1，它包括一个本身是某个元组类型的属性（*TT*1 的度是2，不是5）。现在令 NADDR1 和 NADDR2 分别是类型 *TT*1 和 *TT*2 的元组变量，那么：

- 表达式

```
NADDR2 WRAP { STREET, CITY, STATE, ZIP } AS ADDR
```

取 NADDR2 的当前值，包装 STREET，CITY，STATE 和 ZIP 分量，得到单个元组值 ADDR。表达式的结果是类型 *TT*1，所以下面的赋值是正确的：

```
NADDR1  :=  NADDR2 WRAP { STREET, CITY, STATE, ZIP } AS ADDR ;
```

- 表达式

```
NADDR1 UNWRAP ADDR
```

取 NADDR1 的当前值，解包 ADDR 分量，得到分离的 STREET，CITY，STATE 和 ZIP 分量。表达式的结果是类型 *TT*2，所以下面的赋值是正确的：

```
NADDR2  :=  NADDR1 UNWRAP ADDR ;
```

**元组类型和可能表示**

你可能注意到这一节中描述的元组类型生成子语法与第 5 章中声明过的可能表示的语法之间存在一些相似性——它们都包含一列由逗号分割的项，每一项指定了某一事物名称以及与之对应的类型名称，那它们到底是两个概念还是一个概念呢？事实上是两个概念（语法的相似性并不重要）。例如，如果 *X* 是元组类型，那么可能需要做这一类型的投影，就像前一节描述的那样。但如果 *X* 是某个标量类型 *T* 的一个可能表示，那么毫无疑问需要对标量类型 *T* 的一个值进行投影。进一步的讨论请看参考文献 [3.3]。

## 6.3 关系类型

现在我们回来讨论关系。对以前小节中的元组，我们的处理方法相似；不过，后面我们会更多地关注关系，并把内容分成几个部分——6.3 节讨论关系类型，6.4 节讨论关系值，6.5 节讨论关系变量。

首先给出术语关系的准确定义。**关系值**（简而言之就是关系）*r* 由表头和主体组成，[⊖] 其中：

- 关系 *r* 的**表头**是一个元组的表头，这在 6.2 节已经定义过。关系 *r* 有着相同的属性名、属性类型以及相同的度。
- 关系 *r* 的**主体**是元组的集合，它们有相同的表头；集合的基数（cardinality）被称为 *r* 的**基数**（通常，集合的势就是集合元素的个数）。

*r* 的**关系类型**由 *r* 的表头决定，并与表头有相同的属性名、属性类型以及相同的度。准确地说，**关系类型名**为：

```
RELATION { A1 T1, A2 T2, ..., An Tn }
```

这里有一个关系的例子（它和第 4 章图 4-6 所示的关系相似但不相同）：

| MAJOR_P# : P# | MINOR_P# : P# | QTY : QTY |
|---|---|---|
| P1 | P2 | 5 |
| P1 | P3 | 3 |
| P2 | P3 | 2 |
| P2 | P4 | 7 |
| P3 | P5 | 4 |
| P4 | P6 | 8 |

这个关系的类型是：

```
RELATION { MAJOR_P# P#, MINOR_P# P#, QTY QTY }
```

在非正式的上下文中，从关系的表头中省略类型名，只显示属性名的情况是很普遍的。非正

---

⊖ 根据关系常用的表状图，表头对应于列名的行，主体对应于数据行的集合。表头被当作模式，也被当作内涵，这时，主体即为外延。

式地，可以这样表示前面的关系：

| MAJOR_P# | MINOR_P# | QTY |
|---|---|---|
| P1 | P2 | 5 |
| P1 | P3 | 3 |
| P2 | P3 | 2 |
| P2 | P4 | 7 |
| P3 | P5 | 4 |
| P4 | P6 | 8 |

在 **Tutorial D** 中，下面的表达式可以表示这个关系：

```
RELATION {
  TUPLE { MAJOR_P# P#('P1'), MINOR_P# P#('P2'), QTY QTY(5) } ,
  TUPLE { MAJOR_P# P#('P1'), MINOR_P# P#('P3'), QTY QTY(3) } ,
  TUPLE { MAJOR_P# P#('P2'), MINOR_P# P#('P3'), QTY QTY(2) } ,
  TUPLE { MAJOR_P# P#('P2'), MINOR_P# P#('P4'), QTY QTY(7) } ,
  TUPLE { MAJOR_P# P#('P3'), MINOR_P# P#('P5'), QTY QTY(4) } ,
  TUPLE { MAJOR_P# P#('P4'), MINOR_P# P#('P6'), QTY QTY(8) } }
```

下面的表达式是一个关系选择子调用的例子。通常的格式是：

```
RELATION [ <heading> ] { <tuple exp commalist> }
```

这里的 < *heading* > 是可选项，它是用括号括起来并用逗号分隔的 < *attribute* > 的列表，仅当 < *tuple exp commalist* > 为空时需要这个选项。当然，所有的 < *tuple exp* > 必须是相同的元组类型，而且如果指定了 < *heading* >，那么元组类型必须是由 < *heading* > 决定的。

注意，严格地说，关系不包含元组——它包含主体，依次地，主体包含元组。不过，为了方便讨论，我们会非正式地说关系直接包含元组，在整本书中我们也沿用这个简单的说法。

类似于元组，度为 1 的关系我们称其为一元的，度为 2 的关系我们称其为两元的，度为 3 的关系我们称其为三元的，更一般地，度为 $n$ 的关系我们称其为 $n$ 元的。度为 0 的关系——没有属性的关系——我们称其为零元的（我们将在下一节详细讨论最后一种可能）。同时，我们注意到：

- 表头的每个子集也是表头（与元组相似）。
- 主体的每个子集也是主体。

在这两种情况下，讨论的子集在比较特殊时可能是空集。

**关系类型生成子**

**Tutorial D** 提供了关系类型生成子，可以在定义某个关系变量的时候调用这一操作。下面是一个例子：

```
VAR PART_STRUCTURE ...
      RELATION { MAJOR_P# P#, MINOR_P# P#, QTY QTY } ... ;
```

（为简单起见，我们省略了定义中一些不相关的部分。）一般说来，关系类型生成子和元组类型生成子在形式上是一样的，除了将出现 TUPLE 的地方换为 RELATION 之外。通过调用关系类型生成子，能产生出关系类型——例如，在关系变量 PART_STRUCTRUE 的定义中出现的类型就是一个生成的类型。

每一个关系类型都有一个相关的关系选择子操作符。我们刚才已经看到了一个关系类型的选择子操作符调用的例子。由选择子操作符调用所表示的关系可用于给关系变量 PART_STRUCTRUE 赋值，也可用于与另一个关系类型进行相等性比较。特别要注意的是，两个关系如果属于同一类型，则必须且只须含有相同的属性。同时要注意，对于一个给定的关系类型，它的属性可以是任何类型的（它们甚至可以是一些元组类型或是其他一些关系类型）。

## 6.4 关系的值

现在我们开始在细节上更多地关注关系，比如关系的值。首先要注意的是关系要满足某些特

性，这些特性都是由前面章节给出的关系的定义直接决定的，而且都很重要。我们首先以问题的形式描述这些特性，然后再详细讨论它们。如下，在任何给定的关系中：

1）每个元组中对应于每个属性都包含一个确定的值（属于某种恰当的类型）。

2）属性之间没有左右顺序差别。

3）元组之间也没有上下顺序差别。

4）关系中没有重复的元组。

我们用图 3-8 中的供应商关系来阐述这些特性。为方便起见，我们在图 6-1 中再次给出这个关系，并在表头中加入类型名。注意：同样我们也应该在主体中加入属性和类型名。例如，供应商 S1 对应于 S#这一项应写为：

```
S# S# S#('S1')
```

| S# : S# | SNAME : NAME | STATUS : INTEGER | CITY : CHAR |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

图 6-1　供应商关系（来自图 3-8）

然而，为简单起见，我们仍然使主体如图 3-8 中一样。

**1. 关系的特性**

*关系是规范化的*

6.2 节中说过，每个元组中对应于每个属性都包含一个确定的值，所以对于关系中的每个元组，对应的每个属性都包含一个确定的值。满足此特性的关系被称作**规范化的**（normalized），或称作属于**第一范式**（first normal form），即 1NF。[⊖] 这么说来，图 6-1 中的关系就是规范化的。

注意：第一个特性也许看起来非常明显，事实上也是——特别是你可能已经意识到，按照这样的定义，所有的关系都是规范化的！然而，这个特性并没有什么重要的后果。可参见本节后面的“关系赋值的属性”小节以及第 19 章。

*属性由左到右，没有顺序差别*

我们已经知道一个元组中的各组成部分没有左右顺序差别，同样，一个关系中的各个属性也没有这样的差别（本质上原因相同——一个关系的表头中包含一个属性集，而数学意义上的集合元素是没有顺序差别的）。当我们要在纸面上将关系表示成表，就迫不得已需要将这些列从左到右地表示出来，但是你应该尽可能忽略顺序差别。例如，在图 6-1 中，那些列也可以按这样的顺序排列：SNAME，CITY，STATUS，S#——这两组数据代表的是同一个关系，至少表示的是同一个关系模型。[⊖] 所以，这里没有例如“第一个属性”或“第二个属性”这样的说法，当然也没有“下一个属性”这样的说法。属性一般是通过名字来表征的，而不是通过位置。这就大大减少了错误和恶意程序发生的机会。比如说，通过从一个属性跳到另一个属性来破坏系统的办法不再可行。这种情况不同于其他很多编程系统，在那些系统中，可以故意或通过其他一些恶意的方法利用那些逻辑上不连续但物理上连续的数据。

*元组由上到下，没有顺序差别*

这一特性是因为关系的主体是一个集合（元组的集合）；而集合在数学上是无序的。当我们要在纸面上将关系表示成表时，就不得不将这些行从上到下地表示出来，但是你应该尽可能忽略顺序差别。例如，在图 6-1 中，那些行也可以按完全相反的顺序排列——这两组数据代表的是同一个关系。所以，这里没有例如“第 1 个元组”或“第 5 个元组”或“第 97 个元组”这样的说法，当然也没有“下一个元组”这样的说法，换句话说，这里没有位置寻址的概念，也没有

---

⊖ 之所以称其为第一范式，是因为下面还会定义第二范式、第三范式等更高级别的范式（见第 12、13 章）。

⊖ 在数学的关系中，属性有从左到右的顺序（元组也是有顺序的），与关系模型中的属性无顺序是不同的。

“下一个”的概念。要注意，如果我们有这些概念的话，就应该会有一些额外的操作符——例如，“取出第 $n$ 个元组”，“将新的元组插入到这里”，“将这个元组从这里移到那里”，等等。关系模型的一个重要特征（也是 Codd 的信息原则的直接结果）就是，有且只有一种途径来表示模型中的信息，我们只需要唯一的一种操作符集合来处理它。

继续讨论上面提到的这一点：事实上，显而易见，如果我们有 $N$ 种不同的方法来表示信息，那么必然需要 $N$ 种不同的操作符集合。如果 $N>1$，那么我们就有更多的操作要去遵守、记录、教、学、记忆和使用。但是这些额外的操作符只是增加了复杂度，对表达信息并不有益！$N>1$ 时能做的并不比 $N=1$ 时能做得更多。在第 26 章中我们会继续讨论这个问题（参考 [26.12 ~ 26.14] 以及 [26.17]），第 27 章中也会介绍。

回到对关系的讨论中来。当然，一些元组从上到下有序排列的情形——同样还有属性从左到右有序排列的情形——在数据库和宿主语言（如 C 或者 COBOL）之间的接口中是必需的（见第 4 章对 SQL 光标和 ORDER BY 的讨论）。不过是宿主语言而不是关系模型提出了这种要求；宿主语言要求将无序的关系转变为有序的列表或排列（元组序列）的形式，这样“取出第 $N$ 个元组”才有意义。同样，当要把查询结果呈现给用户时，也会涉及元组有序排列的概念。然而，这样的概念并不属于关系模型；它们属于关系的实现环境的一部分。

*关系中没有重复元组*

这种特性也是来自于关系的主体是一个集合这个事实；集合在数学上是不包含重复元组的（即元组之间都是不相同的）。

*注意*：这个特性再次阐明了关系和表并不是一回事，因为表中可以包含重复的行（没有任何规则阻止这样的可能性），但是关系按照定义绝对不能包含任何重复元组。

事实上，显而易见，“重复元组”的概念并没有任何意义。简单假设图 6-1 中的关系只含有两个属性 S#和 CITY，根据 6.5 节的说明“供应商 S#位于城市 CITY”，我们假设关系中包含一个元组表现了这样一个“事实”：供应商 S1 位于城市 London。如果关系中还包含了一个同样的元组（如果这是允许的话），也就是再次告知我们同样的“事实”。但就算这是事实，说两遍也无益于增加它的真实性！

对重复元组的进一步讨论可参考 [6.3] 和 [6.6]。

**2. 关系与表**

为了便于参考，我们在这一小节中列出关系这种正规的对象和表这种不正规的对象之间的主要区别：

1）在关系中，表头里的每个属性都包含有一个类型名，但是在表的形式中常常省略类型名。

2）在关系中，主体里的每个元组的各项都包含一个类型名和一个属性名，但是在表的形式中常常省略这些类型名和属性名。

3）在关系中，主体里的每个元组的各属性值都属于某种可用类型，但是在表的形式中这些值通常用一些简化的形式表示——例如，用 S1 代替 S#（‘S1’）。

4）表的形式中的那些列从左到右是有序的，但是关系中的那些属性是无序的。*注意*：这意味着在表的那些列中可能含有重复的名字，或者根本就没有名字，例如，考虑如下这个 SQL 查询：

```
SELECT S.CITY, S.STATUS * 2, P.CITY
FROM   S, P ;
```

查询结果中得到的列名是什么呢？

5）在表的形式中各行是从上到下有序的，而关系中的各元组之间无序。

6）在表的形式中可能包含重复的行，但是关系中不包含重复元组。

前述并没有详尽地给出所有的区别，其他的一些区别如下：

- 在表的形式中至少含有一个列，然而关系中却要求至少含有一个属性（参见这一节中后面的一个小节“不含任何属性的关系”）。

- 在表的形式中——至少在 SQL 中——允许包含空行，然而关系中不允许（见第 19 章）。
- 表被认为是“平面的”或是“二维的”，然而关系却是“$n$ 维的”（见第 22 章）。

鉴于上述，为了使表格能被用来表示一个关系，我们就必须在怎么“读”这样的表格方面达成共识；换句话说，我们必须在对这类图表的“**解释原则**”（rules of interpretation）上达成共识。明确地说，我们认为每一列都潜在地属于一个类型；每一个属性值都是一个属于这种相应类型的值；行和列的排列顺序都是不相关的；而且不允许重复行的出现。如果我们能在这些解释原则上达成共识，那么（也只有这样）我们才能认为表是关系的合理表示形式。

至此可知，表和关系实际上并不是一回事（尽管有时为了方便我们会假设成一样）。进一步说，**关系**被定义为一种抽象的对象，而**表**则是一种具体的图形，是抽象的对象在纸面上的具体表示，它们并不完全一样。当然，它们很相似，至少在不正规的场合，经常将它们说成是一回事也是可以接受的。如果希望尽量精确——现在我们正在尽力做到精确——我们就不得不承认它们其实是两个不同的概念。

也就是说，我们值得指出，关系模型中的关系这种抽象对象能在纸面上简单地表示出来，这确实是一个巨大的优势。这种简单的表示形式使得关系系统容易使用和理解，而且容易理解关系系统的运行方式。然而，不幸的是，这种简单的表示形式会表示出一些不正确的信息（如：元组从上到下有序）。

### 3. 关系赋值的属性

如 6.3 节中所述，一般说来，关系中的属性可以定义为任何类型。那么关系类型作为一种类型，也可以用来定义关系中的属性；换句话说，属性可以是**关系赋值**（relation-valued）的，意思是我们可以有这样的关系，它的属性值还是关系。换言之，我们可以有嵌套其他关系的关系。图 6-2 中给出了这种关系的一个实例。观察此关系知（a）属性 PQ 是关系赋值的；（b）关系的势和度都是 5；（c）由供应商 S5 提供的空的零件集表示成 PQ 的值为一个空集（更精确地说，是一个空的关系）。

| S# | SNAME | STATUS | CITY | PQ |
|---|---|---|---|---|
| S1 | Smith | 20 | London | P# QTY<br>P1 300<br>P2 200<br>.. ...<br>P6 100 |
| S2 | Jones | 10 | Paris | P# QTY<br>P1 300<br>P2 400 |
| .. | ..... | .. | ..... | ..... |
| S5 | Adams | 30 | Athens | P# QTY |

图 6-2　有关系赋值属性的关系

在这里，我们提起关系赋值的属性的主要原因是，在过去，这样可能被认为是不合规则的。事实上，本书的早些版本中也是这样认为的。例如，下面这段话是从本书的第 6 版中摘录的：

注意，所有的列值都具有原子性……也就是说，一个表中每一个行列交叉的位置上都存在一个具体的数值，绝不是一组值。比如说，在表 EMP 中，我们用

| DEPT# | EMP# |
|---|---|
| D1 | E1 |
| D1 | E2 |
| .. | .. |

代替了

| DEPT# | EMP# |
|---|---|
| D1 | E1,E2 |
| .. | .. |

在第二个表中，EMP#列就是我们称之为**重复组**的一个很好的例子。一个重复组就是这样的一列，在某一行上包含好几个值（一般说来，不同行上值的个数不同）。关系数据库不允许重复组的出现；所以在关系系统中，上面的第二个表是不允许出现的。

稍后在同一本书中我们找到：“域（也就是类型）只包含原子值……所以，关系不包含重复组。满足这个条件的关系被称为是规范化的，或者也可以说是满足第一范式的……在关系模型的

上下文中提到的关系指的都是规范化的关系。”

这种说法其实是不正确的，然而（至少不完全正确）：这其实是因为对作者有关类型（域）本质的描述产生了误解。由于第12章中介绍的原因（见12.6节），这种误解在现实中不太可能导致任何严重的错误；尽管如此，仍然需要向受到误导的人表示道歉。至少在第6版中已经修正为：关系模型中所有的关系都是规范化的！第12章中会详细讨论这一点。

**4. 不含任何属性的关系**

每个关系都包含一个属性的集合；并且，由于空集也是一个集合，所以，一个关系的属性集是空集也是可能的，换句话说，就是不含任何属性。（不要混淆的是：我们经常说到“关系为空”，意思是这个关系不含任何元组，但是这里不同，我们说的是关系的表头包含空的属性集。）所以，从数学角度来看，不含任何属性的关系是合理的，更让人惊讶的是，从实际角度出发，这样的关系也尤为重要。

为了更深入地了解这个概念，我们首先需要考虑这个问题，不含任何属性的关系是否能够包含元组。回答（尤为惊讶）是能！具体来说，这样的关系至多能包含一个元组：叫做0元组（也就是不包含任何内容的元组；这样的关系不能包含多于一个的这样的元组，因为所有的0元组都是彼此相同的）。所以就出现了两种度为0的关系——一个就是只包含一个元组的，另一个就是不包含任何元组的。按照Darwen [6.5] 的说法，它们有两个可爱的名字：第一个称作TABLE_DEE，另一个称为TABLE_DUM，或者简称DEE和DUM。注意，当把关系想成是另类传统的表的时候，我们很难为DEE和DUM这样的关系画出图示！

为什么DEE和DUM这么重要呢？对这个问题我们或多或少有几个相关的答案。一个就是它们在关系代数中扮演着很重要的角色——见第7章——有点类似于空集在集合理论中，0在普通算术中扮演的角色。另一个就和关系的用意有关系了（见文献 [6.5]）；本质上看，DEE代表着TRUE，或者yes，而DUM代表着FALSE，或者no。换句话说，它们代表着最基础的概念。（记住哪个是哪个的最好方法就是DEE中的“E”匹配yes中的“e”。）

在**Tutorial D**中，表达式TABLE_DEE和TABLE_DUM可分别被用作关系 `RELATION{ } { TUPLE{ } }`和`RELATION{ } { }`的缩写。

这里我们不可能在这个话题上更深入了；这些内容在你前面遇到DEE或者DUM的时候已经足够用，详细的讨论见参考文献 [6.5]。

**5. 关系上的操作符**

在6.3节中我们简单地提到了关系的选择子、赋值和相等性比较这些操作符。当然，比较操作符“<”和“>”是不适用于关系的；然而，关系中还可以用其他的一些比较符，除了最简单的相等之外，下面我们会说到。

**关系的比较** 我们用如下的语法来定义一种新的 < *bool exp* >，< *relation comp* >：

```
<relation exp> <relation comp op> <relation exp>
```

关系中出现的两个 < *relation exp* > 必须是属于同种类型的，而 < *relation comp op* > 必须是下面的一种：

= （相等）
≠ （不相等）
⊆ （包含于）
⊂ （真包含于）
⊇ （包含）
⊃ （真包含）

任何需要出现 < *bool exp* > 的地方我们都允许出现 < *relation comp* > ——例如，在一个WHERE从句里。这里有两个例子：

1) `S { CITY } = P { CITY }`

含义：对于给定的CITY，在供应商上的投影等价于在零件上的投影吗？

2) `S { S# } ⊃ SP { S# }`

含义：有根本不提供任何零件的供应商吗？

实际中经常会用到的一种特殊的关系比较是，测试一个给定的关系是否等价于同一类型的空的关系（也就是说，不包含任何元组）。给这种特殊的情况找一个简写形式是很有用的。所以我们定义表达式

```
IS_EMPTY ( <relation exp> )
```

如果由 <*relation exp*> 表示的关系是空的话，则返回 TRUE，否则返回 FALSE。

**其他的操作符**　我们还要求能判断一个给定的元组 *t* 是否出现在一个给定的关系 *r* 中：

```
t ∈ r
```

如果 *t* 在 *r* 中出现，则返回值为 TRUE，否则为 FALSE（“∈”是一种集合成员判断操作，表达式 $t \in r$ 可解释为“*t* 属于 *r*”或者“*t* 是 *r* 的成员”，或者简单地说成“*t* 是在 *r* 中”）。

我们还需要从度为 1 的关系中提取出那个单独的元组：

```
TUPLE FROM r
```

这个表达式在 *r* 不止包含一个元组的时候会出现异常，否则，返回的就是那个单独的元组。

除了至今讨论过的那些操作，还有很多常见的操作——连接、选择、投影等——组成关系代数的一些操作。下一章会详细介绍这些操作。

**关系类型推论**　就像 6.2 节中描述的，元组类型命名模式方便了确定任意元组表达式的结果类型，同样在 6.3 节中，关系类型命名模式方便了确定任意关系表达式的结果类型。第 7 章中会有更详尽的讨论；这里只给出一个简单的例子。给定的供应商关系变量为 S，表达式

```
S { S#, CITY }
```

产生这种类型的结果（关系）：

```
RELATION { S# S#, CITY CHAR }
```

类似的过程适用于各种可能的关系表达式。

**ORDER BY**　为了展示的需要，我们强烈要求提供 ORDER BY 操作，就像在 SQL 中一样（见第 3 章）。这里省略了具体的定义，因为语义已经很明显了。然而还要注意：

- ORDER BY 用于将元组排列成某种具体的序列，然而元组不能定义“<”和“>”的关系。
- ORDER BY 不是一种关系操作，因为它的返回值不是一个关系。
- ORDER BY 不是一种函数操作，因为一般说来，对于一个给定的输入会产生很多种可能的输出。

作为最后一点的示例，我们考虑 ORDER BY CITY 操作对图 6-1 中的供应商关系的影响。显然，这个操作能返回 4 种不同的结果中的任何一种。相反，关系代数中的操作当然是函数操作了——对于任何给定的输入，经常只有一种可能的输出。

## 6.5　关系变量

现在来讨论关系变量。回忆第 3 章的内容，关系变量分为两类，基本的关系变量和视图（也可分别称作实的和虚的关系变量）。这一节将主要讨论基本的关系变量（第 10 章中再讨论视图）；然而要注意，这里讨论的关于关系变量的内容，是针对一般意义上的关系变量而言的，当然也包括视图。

### 1. 基本的关系变量定义

这里给出基本关系变量的一个定义：

```
VAR <relvar name> BASE <relation type>
                  <candidate key def list>
                [ <foreign key def list> ] ;
```

< *relation type* > 的形式是

```
RELATION { <attribute commalist> }
```

(事实上就像 6.3 节中讨论的那样，这里是对关系类型生成器的一个调用)。一会我们再解释 < *candidate key def list* > 和可选的 < *foreign key def list* > 。这里作为例子给出的是供应商和零件数据库中基本关系变量的定义 (数据来自图 3-9)：

```
VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR }
    PRIMARY KEY { S# } ;
VAR P BASE RELATION
    { P#     P#,
      PNAME  NAME,
      COLOR  COLOR,
      WEIGHT WEIGHT,
      CITY   CHAR }
    PRIMARY KEY { P# } ;

VAR SP BASE RELATION
     { S#    S#,
       P#    P#,
       QTY   QTY }
     PRIMARY KEY { S#, P# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P ;
```

解释：

1) 这三个基本的关系变量都具有下列类型：

```
RELATION { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR }

RELATION { P# P#, PNAME NAME, COLOR COLOR,
                              WEIGHT WEIGHT, CITY CHAR }

RELATION { S# S#, P# P#, QTY QTY }
```

2) 表头、主体、属性、元组、度等之前用于关系值的术语也被应用到关系变量上。

3) 对于任何一个给定的关系变量，所有可能的值都有相同的关系类型，即关系变量定义中指定的关系类型 (如果给定的关系变量为视图，则不是直接指定的)。因此，这些值也含有相同的表头信息。

4) 在定义一个基本的关系变量时，这个关系变量同时也被赋予一个相应的初值——合适类型的空关系。

5) 在第 9 章中，我们将给出候选码的详细定义。在这之前，我们先简单地假设一个基本的关系变量定义包含且仅包含一个具有下列特殊形式的 < *candidate key def* > :

```
PRIMARY KEY { <attribute name commalist> }
```

6) 在第 9 章中，我们同样也将给出外码的定义。

7) 在定义一个新的关系变量时，系统同时在目录中生成用于描述这个关系变量的条目。

8) 就像第 3 章中提到的，关系变量和关系一样，都含有一个相应的**谓词**：对于我们问题中的关系变量，会有一系列可能的值，即一系列的关系，而这个谓词和所有这些作为关系变量的值的关系相同。对于前面提到的供应商的关系变量 S 而言，它的谓词类似于下列形式：

*供应商编号为 S#的供应商的名字为 SNAME，并且在合同约束期限内，其状态为 STATUS，该供应商所在的城市为 CITY。*

9) 我们假设可以用一个平均值来指定基本关系变量的属性的**默认值**。如果用户在插入某些元组时没有提供一个显式值，默认值就会自动放在合适的属性位置上。**Tutorial D** 中给出了一种用于指定默认值的语法——在基本关系变量的定义上增加一个新的从句，DEFAULT { < *default*

*spec commalist* > }，其中每一个 < *default spec* > 的形式为 < *attribute name* > < *default* >。比如，我们可以在供应商关系变量 S 的定义中指定 DEFAULT {STATUS 0, CITY ''}。注意：候选码通常（当然并不绝对）都没有默认值（见第 19 章）。

下面是删除一个已存在的基本关系变量的语法：

```
DROP VAR <relvar name> ;
```

这个操作首先将指定的关系变量的值置为“空”（换句话说，就是它会删除这个关系变量中的所有元组）；然后它会删除该关系变量相关的所有目录条目。到此为止，这个关系变量就不再为系统所能够识别。注意：为了简单起见，我们假设只要待删除的关系变量在别处还在使用，DROP 操作就无法正确执行——比如，可能其他地方的视图定义引用了这个关系变量。

**2. 更新关系变量**

关系模型中包含一个**关系赋值**操作用于分配值，即更新关系变量（特别是基本关系变量）的值。下面是 **Tutorial D** 中所用语法（做了一点点简化）：

```
<relation assignment>
    ::=   <relation assign commalist> ;

<relation assign>
    ::=   <relvar name> := <relation exp>
```

该表达式的语义如下所示：[⊖] 首先，所有位于 < *relation assign* > 右边的 < *relation exp* > 表达式都被计算一遍；然后，根据所写的次序执行所有的 < *relation assign* > 表达式。通过计算右边的 < *relation exp* >，我们可以得到一个关系，这样执行一个 < *relation assign* > 过程就是将这个关系指定给左边的 < *relvar name* > 所标识的关系变量（替换这个关系变量原有的值）。当然，这里关系和相应的关系变量必须具有相同的类型。

作为一个例子，假设我们还有两个具有相同类型的基本关系变量 S'和 SP'，我们把这两个关系变量分别作为供应商的关系变量 S 和发货关系变量 SP：

```
VAR S' BASE RELATION
    { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR } ... ;

VAR SP' BASE RELATION
    { S# S#, P# P#, QTY QTY } ... ;
```

下面是一些有效的 < *relation assignment* > 的例子：

```
1) S' :=  S , SP' :=  SP ;
2) S' :=  S WHERE CITY = 'London' ;
3) S' :=  S WHERE NOT ( CITY = 'Paris' ) ;
```

这里要注意的是，每一个单独的 < *relation assign* > 可以被看作：（a）提取表达式右边所指定的表达式，以及（b）更新表达式左边的关系变量。

现在假设我们将在第二和第三个例子中用关系变量 S 来替换左边的关系变量 S'：

```
2) S  :=  S WHERE CITY = 'London' ;
3) S  :=  S WHERE NOT ( CITY = 'Paris' ) ;
```

这两个赋值对于关系变量 S 来说都是有效的更新——一个有效地删除了所有不在 London 的供应商，另一个有效地删除了所有在 Paris 的供应商。为了简单起见，在 **Tutorial D** 中支持了显式的 INSERT、DELETE 和 UPDATE 操作，但是每个这样的操作都被定义为某种 < *relation assign* > 的快捷方式。下面是一些例子：

---

[⊖] 在 72 页第 5 章脚注中我们给出了一些例外情况。

```
1) INSERT S RELATION { TUPLE { S#     S# ('S6'),
                               SNAME  NAME ('Smith'),
                               STATUS 50,
                               CITY   'Rome' } } ;
```

等价的一个赋值如下：

```
S  :=  S UNION RELATION { TUPLE { S#     S# ('S6'),
                                  SNAME  NAME ('Smith'),
                                  STATUS 50,
                                  CITY   'Rome' } } ;
```

这里要注意的是，如果供应商 S6 所指定的元组已经存在于关系变量 S 中，这个赋值操作就会成功。在实际应用中，我们可以考虑通过下列方法来改进 INSERT 操作：如果要插入的元组已经存在，则抛出一个例外。当然，这里为了简单起见，我们暂时不考虑这种情况。类似的，对于 DELETE 和 UPDATE 操作也做同样的处理。

2) `DELETE S WHERE CITY = 'Paris' ;`

等价的一个赋值如下：

```
S  :=  S WHERE NOT ( CITY = 'Paris' ) ;
```

```
3) UPDATE S WHERE CITY = 'Paris'
                 { STATUS := 2 * STATUS,
                   CITY   := 'Rome' } ;
```

等价的赋值如下：

```
S  :=  WITH ( S WHERE CITY = 'Paris' ) AS T1 ,
           ( EXTEND T1 ADD ( 2 * STATUS AS NEW_STATUS,
                             'Rome' AS NEW_CITY ) ) AS T2 ,
           T2 { ALL BUT STATUS, CITY } AS T3 ,
           ( T3 RENAME ( NEW_STATUS AS STATUS,
                         NEW_CITY AS CITY ) ) AS T4 :
      ( S MINUS T1 ) UNION T4 ;
```

在这个例子中我们可以看到，一个等价的赋值通常有着更复杂的形式。事实上，这是由几个特征决定的——在下一章里才会详细解释。这里我们先不做更多的讨论。

为了引用起见，下面是 INSERT、DELETE 和 UPDATE 操作的一个简单的语法综述：

```
INSERT <relvar name> <relation exp> ;

DELETE <relvar name> [ WHERE <bool exp> ] ;
UPDATE <relvar name> [ WHERE <bool exp> ]
                     { <attribute update commalist> } ;
```

相应的一个 < *attribute update* > 的形式如下：

```
<attribute name> := <exp>
```

同样，DELETE 和 UPDATE 语句中的 < *bool exp* > 表达式能够包含对目标关系变量的属性的引用。而且，这些引用具有明显的语义。

最后在这一小节结束之前我们再强调一遍：关系赋值以及相关的 INSERT，DELETE 和 UPDATE 等操作都是集合层次上的操作。[⊖] 比如，对于 UPDATE 来说，不严格地说，就是更新了目标关系变量中的某一元组的集合。在不太正式的情况下，通常我们会说（比如）更新某个单独的元组，但是事实上我们必须对此有如下的清楚理解：

1) 实际上我们是在谈论如何更新一个元组的集合，因此更新单个元组的情况事实上是碰巧这个集合中只包含单个元组。

---

⊖ 相应的，从定义上来说，在现行 SQL 规范中的 DELETE 和 UPDATE 操作（见 4.6 节）都是元组层次（或者行层次）的，因此都是不当的。

2）在某些时候是不可能更新基数为 1 的元组的集合的。

比如，假设存在一个完整性约束——S1 和 S4 必须有相同的状态（见第 9 章），而供应商关系变量必须受到这个完整性约束的限制。因此，对于任何“单个元组”的 UPDATE 来说，不可能只更新两个供应商中的一个，而必须同步更新两个供应商的信息。如下所示：

```
UPDATE S WHERE S# = S# ('S1') OR S# = S# ('S4')
       { STATUS := some value } ;
```

为了把这个问题说得更清楚一些，必须认识到这里所说的在一个关系变量中“更新一个元组或者一个元组的集合”的说法事实上还是比较宽松的。与关系一样，元组都是确定的值，而且根据定义来说是无法修改的。因此，当我们说到“将元组 $t$ 更新成元组 $t'$”，事实上指的是将元组 $t$（$t$ 中的值）**替换**成另外一个元组 $t'$（同样指的是 $t'$中的值）。[⊖] 同样，对于在一些元组中“更新其中的某个属性 $A$”来说，情况也是类似的。在本书中，为了方便起见，我们将会继续沿用“更新元组”和“更新属性”的说法。但是，这样说的目的只是为了简便起见，它是很宽松的说法。

**3. 关系变量和关系变量的解释**

在本节的最后我们对下列情况给出一些提示（就像 3.4 节中解释的那样）：（a）任何一个给定的关系变量都能被看作一个**谓词**，同时（b）任何时刻在关系变量上出现的元组都可以看成相应的**真命题**。通过使用合适的类型来替换相应谓词的参数，就可以得到这样的命题（“实例化谓词”）。对于一个给定的关系变量，我们可以说相应的谓词是该关系变量的**扩展解释**（或者说意义）；同时，该关系变量的元组对应的命题也约定俗成地认为是正确的。事实上，根据**封闭世界假设**（Closed World Assumption，也称封闭世界解释，Closed World Interpretation），如果一个有效元组（符合关系变量头部信息的元组）在一个关系变量的主体没有出现，则我们可以假设相应的命题是错误的。也就是说，在任何时候，关系变量的主体中包含所有且仅仅当前和正确命题相关的元组。在第 9 章中会进一步介绍这些问题。

## 6.6　SQL 的支持

**1. 行**

SQL 并不支持元组上的操作；相反，它支持行（其中所有的组成元素根据从左到右的次序分别排列）上的操作。在一个给定的行中，如果该行直接位于一个表中，则其组成元素的值称为列值，或者字段值。这些组成元素的值都是通过序号来识别的，尽管它们一般都具有名字（当然，这只是大多数情况下）。行类型没有显式的行类型名称。通过下列所示的 < *row value constructor* > 表达式，我们可以“选择”（相应地会构造一个 SQL 项）一个行上的值：

```
[ ROW ] ( <exp commalist> )
```

如果 commalist 中仅仅包含一个 < *exp* > 表达式，则表达式中的括号可以省略；同时，关键字 ROW 也必须省略，或者把它作为可选项。而且，commalist 表达式不能为空（SQL 不支持“0 行”）。下面是一个例子：

```
ROW ( P#('P2'), P#('P4'), QTY(7) )
```

这个表达式表示了一个度为 3 的行。

就像我们在第 5 章所看到的，SQL 也支持行的类型构造器（这点和 **Tutorial D** 中的元组类型生成器不同）。这样的类型构造器可以在定义其他类型（比如表中的列，或者一些变量）时调用。[⊜] 下面是一个定义变量的例子：

---

⊖ 当然，这并不是说我们不能够更新元组变量，就像在 6.2 节中所解释的那样。然而，元组变量部分并不包含在关系模型中，因此在关系数据库中也不包含这种类型的元组变量。

⊜ 有一点不要搞混了：如果我们很宽泛地说的话，SQL 中的“行值构造器”（row value constructor）基本上是一个元组选择子，而一个“行类型构造器”（row type constructor）则基本上是一个元组类型生成子。

```
DECLARE ADDR ROW ( STREET CHAR(50),
                   CITY   CHAR(25),
                   STATE  CHAR(2),
                   ZIP    CHAR(5) ) ;
```

同时，行上的赋值和比较通常在 SQL 中也是支持的，警告：只有强类型受到限制（见 5.7 节）。因此，要特别注意下面这种情况：*r*1 = *r*2 为真并不表示行 *r*1 和 *r*2 是相同的一行。而且，“<”和“>”也是合法的行操作符。这里，我们并不打算涉及太多关于这类比较的细节问题，有兴趣的读者可以参考文献 [4.20]。

SQL 并不支持行层次上的常规的关系操作符（比如行上的投影、行的连接等），也不直接对应地支持包和解包。另外，SQL 也不支持任何“行类型的引用”，不过在 SQL 几乎不支持任何行操作符的前提下，这也不是那么重要了。

**2. 表类型**

如上所述，SQL 不支持关系，但是，它支持表结构。在 SQL 中的一个表并不是一个包含元组的集合，而是包含行的包（包也称为多集，multiset，它和集合很类似，其中的元素也是没有顺序的，但是和集合的不同之处在于包允许存在重复）。因此，这样的表中的列具有从左到右的顺序，而且可以有重复的行。（在本书中，我们会通过一些特定的规则来保证表中不会出现重复的行，即使是在讲述 SQL 的部分。）SQL 也不使用表头（heading）或者主体（body）的概念。

表类型没有显式的表类型名。通过下列所示的 < *table value constructor* > 表达式，我们可以“选择”（注意，在 SQL 中是“构造”）一个表：

```
VALUES <row value constructor commalist>
```

（这里，commalist 不能为空）。因此，举个例子来说，表达式

```
VALUES ( P#('P1'), P#('P2'), QTY(5) ) ,
       ( P#('P1'), P#('P3'), QTY(3) ) ,
       ( P#('P2'), P#('P3'), QTY(2) ) ,
       ( P#('P2'), P#('P4'), QTY(7) ) ,
       ( P#('P3'), P#('P5'), QTY(4) ) ,
       ( P#('P4'), P#('P6'), QTY(8) )
```

给出了一个表的定义，这个定义和 6.3 节中提到的关系很类似，区别在于这个表中没有显式的列名。

在以前的内容中，我们已经提到了 RELATION 类型生成器，SQL 中没有与之相对应的概念。SQL 也不支持显式的表赋值操作符（尽管 SQL 明确支持 INSERT，DELETE 和 UPDATE 语句）。而且，SQL 也不支持任何表比较的操作符，包括“=”。不过，SQL 支持一个操作符，用于判断一个给定的行是否存在于一个给定的表中：

```
<row value constructor> IN <table exp>
```

SQL 也支持和 TUPLE FROM 操作符相对应的操作符：

```
( <table exp> )
```

在这样的一个表达式中至少需要有一个行，同时，如果 < *table exp* > 表示了一个恰好包含一个行的表格，则返回该行；否则会抛出一个异常。*注意*：< *table name* > 并不是一个有效的 < *table exp* > !

**3. 表的值和变量**

SQL 在表值和表变量中都使用了相同的术语——表，这往往容易引起一些混淆。在本节中，必须根据上下文来理解表这个术语到底指的是变量还是值。首先我们给出 SQL 定义一个基本表的语法：

```
CREATE TABLE <base table name>
            ( <base table element commalist> ) ;
```

其中每个 < *base table element* > 或者是一个 < *column definition* >，或者是一个 < *constraint* >：[○]

- < *constraint* > 表达式指定了应用在基本表上的完整性约束。我们会在第 9 章详细解释完整性约束。不过这里要指出一点，因为表中允许有重复的行，因此在 SQL 表中并不一定要求一个主码（或者说，就是不一定要求有候选码）。
- < *column definition* > 表达式——在语法中要求至少必须有一个这样的表达式——必须具有下列通常形式：

```
<column name> <type name> [ <default spec> ]
```

可选的 < *default spec* > 指定了默认值，也就是说，如果用户在 INSERT 的时候没有显式地提供相应的值，则自动将默认值放置到合理的列上（见 4.6.1 节）。< *default spec* > 的形式为 DEFAULT < *default* >，其中 < *default* > 是一个字面量（也可以称作一个 niladic 内建操作符名字[○]），或者是一个 NULL 关键字（见第 19 章）。如果一个给定的列没有显式的默认值，则可以自动假设其默认值为空值——也就是说，空值是“默认的默认值”（事实上，SQL 中一直是这样做的）。注意：如果一个列的类型是用户自定义（我们已经在第 4 章中提到过）的话，该列的默认值必须为空值，当然，这已经超出了本书的范围了。另外，如果一个列的类型为行类型的话，则其默认值必须为空值；当该列是一个数组类型的时候，其默认值也必须为空值或者为零。

我们可以在第 4 章的图 4-1 中看到一些创建表的例子。注意，SQL 不支持表值列，也不支持完全不包含列的表。当然 SQL 支持 ORDER BY 操作，这一点和关系代数上的大多数操作符比较类似（见第 7、8 章）。然而，SQL 对于“表类型引用”还是部分隐含的，尽管它们并不一定要存在。不过，在这里我们不打算做更多的解释。

一个已存在的表也可以被删除，其语法如下所示：

```
DROP TABLE <base table name> <behavior> ;
```

其中，< *behavior* > 不是 RESTRICT 就是 CASCADE，这和第 5 章中的 DROP TYPE 是一样的。不严格地说，RESTRICT 表示如果有任何一个地方在使用当前表，则删除该表的操作就会失败，而 CASCADE 说明删除的动作总能够成功，对于正在使用该表的事务，也会隐式地为之调用 DROP …CASCADE。

通过 ALTER TABLE，我们也可以改变一个已经存在的基本表。SQL 支持下列“改变”：

- 增加一个新的列。
- 对一个已经存在的列指定一个新的默认值（如果原先已经存在默认值，则取代原先的值）。
- 删除一个列的默认值。
- 删除一个已经存在的列。
- 指定一个新的完整性约束。
- 删除一个已经存在的完整性约束。

对于上述的第一种情况，我们给出一个例子：

```
ALTER TABLE S ADD COLUMN DISCOUNT INTEGER DEFAULT -1 ;
```

这个语句在供应商的基本表中增加了一个新的 DISCOUNT 列（其类型为 INTEGER）。表中所有的行都从 4 列扩展成 5 列；同时第 5 列的初始值都置为 -1。

最后，SQL 的 INSERT、DELETE 和 UPDATE 语句在第 4 章中我们已经讨论过了。

**4. 结构类型**

提示：SQL 标准中和本小节相关的部分非常难以理解，这里，本书已经尽了最大的努力试

---

○ 一个 < *base table element* > 也可以采取 LIKE *T* 的形式，它允许从另外一些已经存在的并且名字为 *T* 的表中复制部分或者全部列定义。

○ 一个 niladic 操作符指的是没有显式的操作数的操作符。比如 CURRENT_DATE 就是一个 niladic 操作符。

图说得更加清楚一些。

首先，和5.7节一样，我们先给出一个结构类型定义的例子：

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
```

在变量和列定义中，同样也可以使用上述的POINT类型。比如：

```
CREATE TABLE NADDR
     ( NAME ... ,
       ADDR ... ,
       LOCATION POINT ... ,
       ... ) ;
```

现在，虽然不会像在第5章里那样说的那么明确，这里还是要特别指出SQL的结构类型是标量类型。这一点和我们之前所述的**Tutorial D**中的POINT类型很类似，都是标量类型。从某些意义上来说，它们更类似于**Tutorial D**中的元组类型。[⊖] 当然，像访问元组中的属性一样，我们也可以访问一个给定的POINT值的组件（“属性”）。点限定语法可以用于实现上述目标，我们可以参考下面所给出的例子（注意到其中必须显式给出名字的相关性）：

```
SELECT NT.LOCATION.X, NT.LOCATION.Y
FROM   NADDR AS NT
WHERE  NAME = ... ;

UPDATE NADDR AS NT
SET    NT.LOCATION.X = 5.0
WHERE  NAME = ... ;
```

就像在之前的例子中所用的那样，SQL结构类型可以像简单的行类型一样有效地起作用（见5.7节），除非：

- 其组件被称为属性，而不是字段。
- 更重要的，和行类型不同的是，结构类型有名字（在这一节的最后我们会再解释这一点）。

迄今为止，SQL的结构类型看起来并不是特别难理解。但是，接下来的可能会困难很多。[⊖] 在前面所述的基础之上，SQL也允许将一个基本表定义成一些结构类型，其中，就会需要考虑更多。我们先将POINT类型的定义进行扩展：

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL
       REF IS SYSTEM GENERATED ;
```

现在我们就可以将一个基本表定义成这个类型，比如：

```
CREATE TABLE POINTS OF POINT
     ( REF IS POINT# SYSTEM GENERATED ... ) ;
```

解释：

1）当我们定义了一个结构类型$T$，系统就会自动定义一个相关的引用类型（“REF类型”），称作REF($T$)。类型REF($T$)的值是在一些基本表[⊜] ——已经被定义为类型$T$（见第3点）——中对行的“引用”。在这个例子中，系统自动定义了一个类型REF(POINT)，这个类型的值都是对已经定义为POINT类型的基本表中的行的引用。

2）CREATE TYPE语句中的REF IS SYSTEM GENERATED说明关联的REF类型的实际值是由系统提供的（当然还有其他方式——比如，REF IS USER GENERATED，不过这里不讨论这些细节问题）。注意一点，事实上，REF IS SYSTEM GENERATED是默认的方式；因此，在本例中，如果需要的话，我们可以不改变最初定义的POINT类型。

3）基本表POINTS已经定义成结构类型POINT的实例。这里，关键词“OF”事实上并不是太

---

⊖ 唯一的区别是结构类型的属性具有从左到右的顺序，而元组类型中则没有。

⊖ 详细可见第20章和第26章中的讨论。

⊜ 也可能是一些视图。当然，在视图基础之上的细节已经超出了本书的范围。

合适，然而，因为在这个问题中这个表并不是真正属于这个类型，它的行也不属于这个类型![1]

- 首先，如果一个表中只包含一列，而且这个列属于结构类型 *ST*，这样我们就可以说——尽管不是在 SQL 中！——这个表的类型是 TABLE(*ST*)，其中的行的类型是 ROW(*ST*)。
- 但是，通常一个表中不会恰好只含有一列；更可能的情况是对于 *ST* 的每个属性都有一列。这样，在刚才的例子中，基本表 POINTS 含有两列 X 和 Y；而没有类型 POINT 的列。
- 而且，我们的例子所用的表中还有一个额外的列：也就是一个可用的 REF 类型。然而，定义此列所用的语法和通常定义列的语法不太一样，具体如下所示：

```
REF IS <column name> SYSTEM GENERATED
```

这个额外的列叫做自引用列（self-referencing column），对于我们所述问题中的基本表中的行，用于保存一些独一无二的 ID 或者“引用”。当插入一个行时会自动给它分配一个 ID，并且在这行被删除之前，相应的 ID 和该行是相关联的。比如，事实上基本表 POINTS 中包含有三个列（POINT#、X 和 Y），而不是仅仅两个。注意：事实上，我们并没有说清楚为什么一开始必须将表定义成一些结构类型，而不是直接定义一个合适的列用于获取这个“独一无二的 ID”的功能。但是，我们给出的解释与 SQL 定义的方式一致。

在另一方面，我们必须指出一个系统产生（SYSTEM GENERATED）的列可以是一个 INSERT 或者 UPDATE 操作的目标列，虽然这里需要特殊的考虑。这里我们省略了相关的细节。

4）表 POINTS 是通过标准调用产生的表的示例，虽然不是很合适，但它同时是一个具有类型的表（typed table），也是一个可引用的表（referenceable table）。就像标准中所说的：“一个表…其行类型是从结构化类型派生的，则称之为具有类型的表。只有基本表或视图可以是有类型的表”，以及“可引用的表也需是具有类型的表……具有类型的表也称为可参照的表”。

前面所述的特征都是在 SQL:1999 中引入的，并且主要用于与 SQL 中的一些“对象功能性”协作（在第 26 章我们会更细地介绍这些功能性）。[2] 不过，标准并没有限定说这些特征只能用于和这些特定功能性相关的情况，而这也恰恰是我们在本章中描述的。

最后一点：回想第 5 章中说明了 **Tutorial D** 中并没有显式的“元组类型定义”的操作符；而是通过在元组变量的定义中调用元组类型生成器来实现所需要的目标。因此，在 **Tutorial D** 中，唯一的名字元组类型（names tuple types）的形式如下：

```
TUPLE { A1 T1, A2 T2, ..., An Tn }
```

在这个基础之上，可以很清楚地判断两个元组类型是否相同，以及两个元组是否具有相同的类型。

我们可以看到，SQL 中的行类型和 **Tutorial D** 中的元组类型在之前的叙述中是很相似的。但是结构类型是不一样的；它有一个显式的“结构类型定义”操作符，另外，结构类型有显式的名字。作为一个例子，可以考虑下列所示的 SQL 定义：

```
CREATE TYPE POINT1 AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
CREATE TYPE POINT2 AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
DECLARE V1 POINT1 ;
DECLARE V2 POINT2 ;
```

这里要注意到 V1 和 V2 是不同的类型。因此它们不能够互相比较，也不能互相赋值。

## 6.7　小结

在本章，我们全面地学习了关系和相关的内容。首先，精确地定义了**元组**的概念，并且强调

---

[1] 特别要注意一点，对于一些操作符 *Op* 而言，如果一些参数 *P* 的已声明的类型是一些结构类型 *ST*，那么对于基本表中一个已经被定义为“*ST*”类型的行，并不能够直接看作和操作符 *Op* 的一个调用所相关的参数。

[2] 事实上，SQL 结构类型总是有一个相关联的 REF 类型，即使这个 REF 类型没有太大的作用，除了当该结构类型被用作定义“具有类型的表”的基础的时候。

了下列几点：（a）对于每个元组，其中的每个属性只能够含有一个值，（b）元组中的属性没有从左到右的顺序，（c）元组的每个子集同样是一个元组，而且每个表头信息的子集也是表头信息。然后，我们讨论了**元组类型生成器**、**元组选择子**、元组**赋值**和**相等性**等，以及其他通常的元组操作符。

接下来介绍了**关系**（更确切地说就是关系值）。我们给出了一个准确的定义，并且指出了一个主体（body）的子集还是主体，一个表头信息的子集还是表头信息（这点和元组的定义一样）。我们也讨论了**关系类型生成器**和**关系选择子**。同时，我们也观察到一个给定的关系类型的属性可以是**任何类型**的。

注意：鉴于最后一点在业界引起了太多的混淆之处，我们需要多花费一点笔墨来解释。我们可能经常听到有些说法，如关系属性只能是一些简单的类型，比如数字、字符串，等等。事实上，在关系模型中根本没有对这些说法的支持。就像第5章所说的，事实上类型可以非常简单，也可以非常复杂。因此，属性对应的值可以为数字、字符串、日期、音频记录、地图、视频记录以及几何点，等等。

上述这一点非常重要，同时它也常常被人们误解。我们再次用下列说法来表述：

支持什么样的数据类型与是否支持关系模型是无关的。

继续回到我们的小结中。我们接着给出了一些所有关系都满足的性质：

1）它们都是规范化的。

2）其中的属性没有从左到右的顺序。

3）其中的元组没有从上到下的顺序。

4）其中不可能有重复的元组。

我们也明确了关系和表之间的一些主要区别：讨论了**关系-值属性**（relation-valued attribute）；考虑了 **TABLE_DEE** 和 **TABLE_DUM**，这两者是仅有的不包含任何属性的关系。我们描述了**关系比较**的一些细节问题，同时也浏览了一遍其他的一些关系操作符，特别是 **ORDER BY**。

在考虑关系上的操作符时，读者可能会注意到第5章中深入讨论了关于标量类型上的用户自定义的操作符。但是，对于关系类型来说不能做类似的自定义操作。原因是我们所需要的大多数关系操作符——比如选择、投影、连接、关系比较，等等——都是内建在关系模型之中的，不需要任何的“用户自定义”。（而且，那些操作都是通用的，也就是说，它们可以应用于所有类型的关系。）当然，从另一个角度来说，如果系统提供了用于定义用户自定义操作符的工具，我们也完全可以在基本的内置的操作符之上增加一些新的操作符。

我们还要提醒一点，任何给定的关系的表头信息都可以看作一个谓词，而且关系的元组都可以被看作真命题；这些命题都是通过向谓词中的参数提供合适类型的参数值得到的。

然后，我们考虑了**基本关系变量**，指出关系变量也具有谓词，这一点和关系很类似。根据**封闭世界假设**我们能够假设，如果一个有效元组在关系变量的主体中没有出现，则其相应的命题也是错误的。

我们还讨论了**关系赋值**（以及 **INSERT**、**DELETE** 和 **UPDATE** 等快捷操作）。我们强调了一点：关系赋值是在**集合层次**上的操作；我们也说明了“更新元组”和“更新属性”的说法事实上是不正确的。

最后，对于前面所述的内容，我们简单介绍了 SQL 中的相应部分。SQL 的表不是元组的集合，而是一个**行的包**（bag）（SQL 也使用“表”这个术语来描述表值和表变量）。基本表可以通过 **ALTER TABLE** 进行改变。它们也可以通过**结构类型**来定义，关于这一点，本书的第26章会给出更多的细节。

## 习题

6.1 如何理解术语“基数”？

6.2 准确定义元组和关系的概念。

6.3 准确描述下列说法的意义：（a）两个元组相等；（b）两个元组的类型相同；（c）两个关系相等；（d）两个关系的类型相同。

6.4　针对图 4-5 所示的供应商 - 零件 - 工程数据库，写一组谓词以及一组根据 **Tutorial D** 的关系变量定义。

6.5　为供应商 - 零件 - 工程数据库的每个关系变量的一个典型的元组写一个元组选择子调用。

6.6　定义一个本地元组变量，通过该变量，可以从供应商 - 零件 - 工程数据库的发货关系变量中提取单个的元组。

6.7　描述下列表达式的意义（根据 **Tutorial D** 的定义）：

a.
```
RELATION { S# S#, P# P#, J# J#, QTY QTY } { }
```

b.
```
RELATION { TUPLE { S# S#('S1'), P# P#('P1'),
                   J# J#('J1'), QTY QTY(200) } }
```

c.
```
RELATION { TUPLE { } }
```

d.
```
RELATION { } { TUPLE { } }
```

e.
```
RELATION { } { }
```

6.8　如何理解第一范式？

6.9　尽可能地列出所有你能够想到的关系和表之间的区别。

6.10　根据下列条件定义一个关系：(a) 有一个“**关系 - 值**”属性，(b) 包含两个这样的属性。给出两个或者更多的关系，用于表示相同的意义，并且其中不包含“**关系 - 值**”属性。

6.11　写一个表达式，使得零件关系变量 P 中的当前值为空时返回 TRUE，反之则返回 FALSE。表达式中不用 IS_EMPTY。

6.12　从哪些方面来说 ORDER BY 不是一个普通的操作符？

6.13　描述“封闭世界假设”。

6.14　一些说法认为，关系变量和传统的计算机文件相比没有什么区别，而元组只不过相当于记录，属性则相当于字段（field）。请对此给出你的意见。

6.15　根据 **Tutorial D** 中的模式描述下列对供应商 - 零件 - 工程数据库的更新：

a. 插入一个新的发货记录，其中供应商号为 S1，零件号为 P1，工程号为 J2，数量为 500。

b. 在表 S 中插入一个新的供应商 S10（供应商的名字为 Smith，城市为 New York；状态未知）。

c. 删除所有蓝色的零件。

d. 删除所有没有发货信息的工程。

e. 把所有红色零件的颜色改成橘黄色。

f. 将所有编号为 S1 的供应商替换成 S9。

6.16　我们已经看到数据定义操作更新了目录。但是目录只是关系变量的一个集合，就像数据库的其他部分那样；所以我们不能使用一般的更新操作符 INSERT、DELETE 和 UPDATE 正确地更新目录！对吗？试讨论之。

6.17　本章中提到，通常任何类型都可以用来定义关系属性。但是这里的限定词“通常”是有原因的。你能想出这个一般规则的例外情况吗？

6.18　谈谈你对 SQL 术语列、字段和属性的理解。

6.19　考虑 6.6 节“结构类型”部分定义的 SQL 类型 POINT 和 SQL 表 POINTS。类型 POINT 用笛卡尔坐标 X 和 Y 表示。如果改用极坐标 R 和 θ 表示会如何？

## 参考文献

下列许多参考文献适用于关系模型的各个方面，而不仅仅适用于关系。

[6.1]　E. F. Codd: “A Relational Model of Data for Large Shared Data Banks,” *CACM 13*, No. 6 (June 1970). Republished in *Milestones of Research—Xelected Papers 1958 - 1982* (*CACM 25th Anniversary Issue*), *CACM 26*, No. 1 (January 1983). See also the earlier version, “Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks,” IBM Research Report RJ599 (August 19, 1969), which was Codd's very first publication on the relational model.

论文尽管发表已有 30 年了，但仍值得一读。当然，在论文第一次发表后，许多思想已经精炼了一些，但本质上，这些改变基本上是演进的，而不是革命性的。论文中的一些思想确实至今没有被完全探究清楚。

我们讨论一下术语的问题。在这篇论文中，Codd 用术语“时间变化关系”代替了关系变量。

但时间变化关系确实不是一个很好的术语。首先，关系是一些值，不是随时间变化的（并没有在不同的时间有不同的值这样的概念）。其次，如果我们在编程语言中，例如 DECLARE N INTEGER，我们并不说 N 是时间变化整型，而说它是整型变量。所以在本书中，我们用术语关系变量而不用时间变化变量。不过，以后至少要注意这样的术语的存在。

[6.2] E. F. Codd: "*The Relational Model for Database Management Version 2*. Reading, Mass.: Addision-Wesley (1990).

Codd 在 20 世纪 80 年代末期花了很多时间修正和扩展了他的最初模型（他重命名为"关系模型版本 1"或 RM/V1），这本书是修正和扩展的结果。书中描述为"关系模型版本 2"或 RM/V2。RM/V1 和 RM/V2 的不同之处如下：RM/V1 试图描绘出全部数据库问题的一个特殊方面的抽象蓝图（尤其是基本方面），而 RM/V2 试图描绘出全部系统的抽象蓝图。所以，RM/V1 有三个部分（结构、完整性和操纵），RM/V2 有 18 个部分，这 18 个部分当然不仅包括最初的三个部分，也包括目录、授权、命名、分布式数据库和数据库管理的其他方面。为便于参考，下面列出这 18 个部分：

| | | | |
|---|---|---|---|
| A 授权 | B 基本操作符 | C 目录 | D DBMS 的设计原则 |
| E DBA 的命令 | F 函数 | I 完整性 | J 指示 |
| L 语言设计原则 | M 操纵 | N 命名 | P 保护 |
| Q 限定 | S 结构 | T 数据类型 | V 视图 |
| X 分布式数据库 | Z 高级操作符 | | |

本书所倡导的思想决不会被广泛接受，但是文献［6.7，6.8］不同。我们在此讨论一个特殊的论点。我们在第 5 章看到，域（即类型）限制了比较。在供应商和零件的例子中，例如，比较 S.S# = SP.P#是无效的，因为比较数是不同的类型；所以想通过供应商和零件号的匹配来连接供应商和货运表将失败。因此 Codd 提出关系代数操作的"域检查忽略"（domain check override，DCO）版本，它允许执行有问题的操作，即使这个操作涉及不同类型值的比较（假设执行是基于表示的匹配而不是基于类型的匹配）。

但这样存在问题。全部的 DCO 思想是建立在混淆类型和表示的基础上的。识别出它们是什么域（即类型），提供了我们需要的域检查，同样也给出诸如 DCO 性能的信息。例如，下面的表达式构成了供应商号和零件号之间有效的表示级的比较：

```
THE_S# ( S# ) = THE_P# ( P# )
```

（这里的操作数都是字符类型的）。我们指出，第 5 章讨论的这种机制给了我们需要的全部的支持工具，是单纯的、系统的做法（不是特定的），是全部正交的。尤其是，不需要构造新的"DCO 连接"使关系模型陷入混乱。

[6.3] Hugh Darwen: "The Duplicity of Duplicate Rows," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 – 1991*. Reading, Mass.: Addison-Wesley (2001).

这篇论文更进一步支持以前在［6.6］（第 1 版）中提到的禁止重复行的论点。论文不仅提出了一些相同论点的最新版本，而且提出了一些其他观点。尤其强调的基本点是，为了用更聪明的方式讨论两个对象是否重复的问题，必须有一个清晰的相等的标准（论文中称为识别标准）。也就是说，两个对象，是表中的行或是其他别的什么，它们"相同"意味着什么？

[6.4] Hugh Darwen: "Relation-Valued Attributes," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 – 1991*. Reading, Mass.: Addison-Wesley (1992).

[6.5] Hugh Darwen: "The Nullologist in Relationland," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 – 1991*. Reading, Mass.: Addison-Wesley (1992).

Darwen 用**空科学**表示"什么都不学"，换句话说，学的是空集（这与 SQL 中的 null 无任何关系。）。集合在关系理论中是普遍存在的，如果这个集合恰好为空会产生什么问题，这远不只是一个琐碎的问题。事实表明，空集常常是很基本的情况。注意：就本章而言，这篇论文最直接可用的部分是第 2 部分（无行表）和第 3 部分（无列表）。

[6.6] C. J. Date: "Double Trouble, Double Trouble," (in two parts), *http://www.dbdebunk.com* (April 2002). An earlier version of this paper, "Why Duplicate Rows Are Prohibited," appeared in *Relational Database Writings 1985 – 1989*. Reading, Mass.: Addison-Wesley (1990).

提出了扩展的一系列论点，例如，禁止重复行。论文特别提出重复行构成了主要的优化障碍

（见第 18 章）。另参见 [6.3]。

[6.7] C. J. Date: "Notes Toward a Reconstituted Definition of the Relational Model Version 1 (RM/V1)," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

总结和批评了 Codd 的 RM/V1（见文献 [6.2] 的注解），提出了另一定义。在想转到版本 2 之前先看看版本 1 是至关重要的。注意：当前书中描述的关系模型版本是基于本论文概述的"重构"版本。更多细节描述见 [3.3]。

[6.8] C. J. Date: "A Critical Review of the Relational Model Version 2 (RM/V2)," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

总结和批评了 Codd 的 RM/V2。

[6.9] C. J. Date: *The Database Relational Model: A Retrospective Review and Analysis*. Reading, Mass.: Addison-Wesley (2001).

这本篇幅短小的书（160 页），详细而客观地回顾和评价了 Codd 在 20 世纪 70 年代发表的文章的贡献。尤其是，它详细地审视了下列论文：

- "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks" (the first version of reference [6.1])
- "A Relational Model of Data for Large Shared Data Banks" [6.1]
- "Relational Completeness of Data Base Sublanguages" [7.1]
- "A Data Base Sublanguage Founded on the Relational Calculus" [8.1]
- "Further Normalization of the Data Base Relational Model" [11.6]
- "Extending the Relational Database Model to Capture More Meaning" [14.7]
- "Interactive Support for Nonprogrammers: The Relational and Network Approaches" [26.12]

[6.10] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz: "Extended Algebra and Calculus for Nested Relational Databases," *ACM TODS 13*, No. 4 (December 1988).

多年来，许多人提出了支持关系赋值属性的可能性；这篇论文是其中之一。这样的提议通常由"$NF^2$关系"而来，$NF^2$是 NFNF 的缩写，代表"非第一范式"。但是，这些提议和本章描述的支持关系赋值属性至少有两个主要的区别：

- 第一，$NF^2$关系提倡者假设在关系模型中禁止关系赋值属性，宣称他们的提议是关系模型的扩展（注意 [6.10] 的标题）。
- 第二，$NF^2$提倡者是正确的——他们扩展了关系模型！例如，Roth 等人提出了并的一种扩展形式，用我们的术语，(a) 递归地解组操作数直到它们不直接或者间接涉及关系赋值属性；(b) 在这些解组操作数上执行规则的并；(c) 最后，再递归地对结果分组。正是递归构成了扩展。特殊的扩展的并只是对存在的关系操作符的一些特殊合并的**简化**，而不是对存在的关系操作符的一些合并的普遍的**简化**。

# 第7章 关系代数

## 7.1 引言

关系代数是一个操作符的集合，以关系作为操作对象，返回的结果是一个关系。第一版本的代数是由 Codd 在文献［5.1］和［7.1］中定义的；文献［7.1］中的即被认为是基本（original）的关系代数。原始的代数包含 8 个操作符，每四个分成一组：

1）传统的集合操作符：并（union）、交（intersection）、差（difference）和笛卡尔积（Cartesian product）（所做的修改只是操作对象变为了特定的关系，而不再是任意的集合）。

2）专门的关系操作符：选择（restrict）（也就是通常所谓的选择）、投影（project）、连接（join）和除（divide）。

图 7-1 非正式地描述了这些操作符如何工作。

Codd 曾对这 8 个操作符做出精确的定义，在下一章将会看到这些定义。但必须明白这 8 个操作符绝不是全部，实际上满足“关系进，关系出”这一简单要求的任何操作符都可以定义。许多学者也已经定义了不少操作符。本章首先讨论这 8 个基本的操作符——但不严格是那些最原始的，而确实是从它们演变而来的那些操作符；然后以此为基础讨论有关代数学的种种思想；并考虑对这 8 个操作符的基本集合进行有益扩展的几个操作。

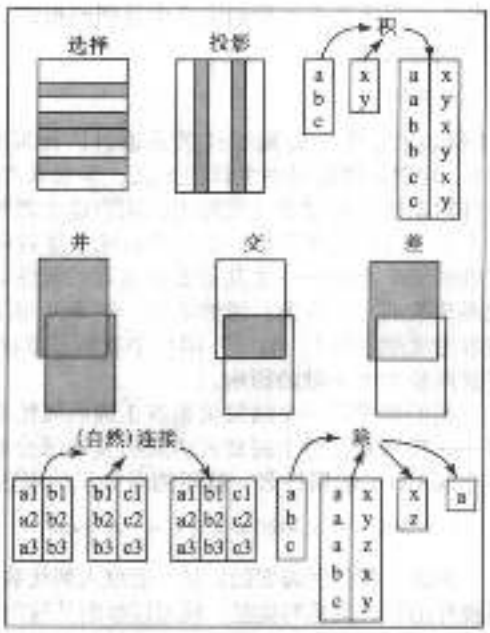



图 7-1 原始的 8 个操作符（概述）

在详细讨论关系代数前，有必要作一些预备的说明：

- 宽泛地讲，所有的操作符都是作用于关系的；实际上，它们是有类型的操作符，具体类型与关系类型生成子相关；因此通过该类型生成子，就可以赋予它们任一指定的关系类型。
- 在这里将要讨论的几乎所有操作符其实只是简单介绍而已。关于这一点我们将在 7.10 节进行详细介绍。
- 所有的操作都是只读而已（也就是，它们能读取但不能更改操作数）。因此，它们作用于特定的值——关系值，但那些关系值（可能碰巧是某关系变量的当前值）不会有任何变更。
- 为了解释前面所说的几点，举个例子吧，“在关系变量 $R$ 的属性 $A$ 上做投影”，意味着这个结果关系来自于在关系变量 $R$ 的属性 $A$ 的投影操作。但是，有时候，为了方便，会使用“在关系变量 $R$ 的属性 $A$ 上做投影”这样的表达式表示另外不同的意思。例如，定义一个在供应商的关系变量 S 上的视图 SC，它由属性 S#和属性 CITY 构成。于是我们可以宽泛而方便地说，关系变量 SC 是“S 在 S#和 CITY 上的投影”。更精确地讲，这意味着在某一个给定的时刻，关系变量 SC 可以看作是变量 S 的当前值在属性 S#和 CITY 的投影。因此，从某个意义上看，在关系变量上的投影，其实质是在关系变量的当前值上的投影。我们希望这些专业术语的双重用法不会引起混淆。

本章接下来的安排如下：在本节之后，7.2 节再次讨论关系封闭的问题，着重举例说明。

7.3 节和 7.4 节详细讨论 Codd 的 8 个基本操作符，7.5 节给出用这些操作符进行描述查询的一些例子。接下来，7.6 节考虑了有关代数的作用的一般化问题。7.7 节讨论诸多的综合性问题。7.8 节描述了对 Codd 基本代数的一些有益扩充，包括扩展（EXTEND）和合计（SUMMERIZE）这两个重要的操作符。7.9 节讨论了在含有关系属性的关系和不包含关系属性的关系之间进行映射的操作符。最后，7.10 节给出一个简要的小结。注意：我们把有关 SQL 的讨论推迟到第 8 章，其原因也将在那里解释。

## 7.2　关系封闭性

在第 3 章中已经说过，任何给定的关系操作的结果是另一个关系，这一事实被称为关系封闭的特性。封闭性意味着可以写出嵌套的关系表达式，也就是说，关系表达式的操作对象可以用任意复杂的关系表达式来表示。（关系代数中的关系嵌套和普通算术中的算术表达式嵌套有类似之处；代数中的关系是封闭的，这一点和“普通算术中的数是封闭的”这一事实具有同样的重要性。）

当第 3 章讨论封闭性时，故意忽略了非常重要的一点。回顾一下便知，每个关系分为两部分——表头和主体；不严格地讲，表头是属性，主体是元组。现在，基本关系的表头（回顾一下第 5 章所讲的：一个基本关系就是一个基本关系变量的值）对系统来说是非常容易理解的，因为它在基本关系变量的定义中被明确指定了。但是那些派生的关系的表头又如何呢？例如，考虑下面的表达式：

```
S JOIN P
```

（本例表示了供应商和零件关系通过匹配城市进行连接。CITY 是两个关系之间唯一的共同属性）。我们知道结果中主体的形式，但表头是何种形式呢？封闭性规定，结果必须有一个表头，并且对系统是可知的（实际上，用户也必须知道，过一会儿将会谈到）。换句话说，结果必须是一个定义好的关系类型。如果严格地根据封闭性，定义的关系操作必须保证每个操作的结果具有正确的关系类型——尤其是要有正确的属性名称。（我们刚刚说过，这是历史上被严重忽略了的关系代数的一个方面；遗憾的是，对 SQL 语言也存在同样的问题，然后到 SQL 的产品——可以在参考文献［7.2］和［7.10］中看到需要注意的意外情况的说明。本章中所介绍的关系代数深受这两份参考文献的影响。）

我们要求每一个结果关系有正确的属性名，原因之一是这使我们可以引用子操作中的那些属性——特别是在整个嵌套表达式的其余部分的操作中。例如，如果不知道 S JOIN P 的结果中有一个叫 CITY 的属性名，就不能写如下的表达式：

```
( S JOIN P ) WHERE CITY = 'Athens'
```

因此，我们所需要的，是一套融入到代数中的**关系类型推演规则**。这样，如果我们知道给定关系操作的输入关系的类型，就可以推断其输出关系的类型。假如有这样一个规则，那么任意的关系表达式，不论怎样复杂，产生的结果就会有一个明确的类型，尤其是有一个明确的属性名的集合。

为此，先介绍一个新的操作符——RENAME。它的作用是对给定的关系中的属性重命名。确切地说，RENAME 操作输入一个给定关系，返回另一个关系，输出的关系与输入关系除了有一个属性名不同之外，其他是完全相同的。（给定的关系可通过关系表达式来指定，该表达式还可能嵌套其他的关系操作。）例如，可以写出如下的式子：

```
S RENAME CITY AS SCITY
```

注意，这不是一个命令或声明，而是一个表达式。因此，它可以嵌套在别的表达式里面——产生一个与关系变量 S 有相同表头和主体的关系，只是属性 CITY 更名为 SCITY：

| S# | SNAME | STATUS | SCITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

请注意 RENAME 表达式没有改变数据库中的供应商关系变量！它只是一个表达式（例如，确切地说，S JOIN SP 也仅仅是一个表达式），像其他表达式一样，它简单地指示了某一个具体的值——在这个特殊的例子中，这个值碰巧看起来像供应商关系变量的当前值。

下面有另外一个例子（这次是多个属性改名）：

```
P RENAME ( PNAME AS PN, WEIGHT AS WT )
```

这个表达式可以速记如下：

```
( P RENAME PNAME AS PN ) RENAME WEIGHT AS WT
```

其结果如下：

| P# | PN | COLOR | WT | CITY |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P2 | Bolt | Green | 17.0 | Paris |
| P3 | Screw | Blue | 17.0 | Oslo |
| P4 | Screw | Red | 14.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |
| P6 | Cog | Red | 19.0 | London |

有必要明确地指出，RENAME 的应用意味着关系代数（不像 SQL）没有必要使用（其实也不支持）像 S. S#那样严格的名字。

## 7.3 基本代数：语法

本节将根据 **Tutorial D** 来介绍关系代数表达式具体的语法，包括 8 个基本操作符和 RENAME 操作。这里介绍语法的主要目的是为了后面章节的引用，也包括了一些语义上需要注意的要点。注意：大多数数据库书籍使用了一些数学的或希腊文的符号来表示关系操作符，典型的有以 σ 表示选择（selection），π 表示投影，∩表示交，⋈表示连接，等等。大家可能注意到，我们更喜欢用诸如 JOIN 和 WHERE 等关键字，这些关键字虽会使表达式变得稍长一些，但它使表达式变得更易读。

```
<relation exp>
    ::=   RELATION { <tuple exp commalist> }
        | <relvar name>
        | <relation op inv>
        | <with exp>
        | <introduced name>
        | ( <relation exp> )
```

每一个 < *relation exp* > 表达式指示一个关系（也就是一个关系值）。第一个形式是关系选择子调用（请参见第 6 章）；在此我们不列出 < *tuple exp* > 的详细语法，因为例子本身已足以说明它们的基本意思。< *relvar name* > 和 < *relation exp* > 都可以自我解释，其他的形式解释如下。

```
<relation op inv>
    ::=   <project> | <nonproject>
```

每一个关系操作调用 < *relation op inv* >，不是 < *project* > 就是 < *nonproject* >。注意：我们从语法上区分这两种情况是因为操作符本身的优先级问题（这样就很容易赋予 project 更高的优先级）。

```
<project>
    ::=   <relation exp>
                { [ ALL BUT ] <attribute name commalist> }
```

< *relation exp* > 必定不是 < *nonproject* >（非投影符）。

```
<nonproject>
    ::=   <rename> | <union> | <intersect> | <minus> | <times>
        | <where> | <join> | <divide>
```

```
<rename>
    ::=   <relation exp> RENAME ( <renaming commalist> )
```

< *relation exp* > 必须不是 < *nonproject* > 。每一个 < *renaming* > 将按它们的顺序来执行（关于 < *renaming* > 的语法请参照前一节的例子）。如果逗号表达式只包含一个 < *renaming* > ，那么圆括号可以省略。

```
<union>
    ::=   <relation exp> UNION <relation exp>
```

< *relation exp* > 必须不是 < *nonproject* > 。除非其中之一或两者都是另一个 < *union* > 。

```
<intersect>
    ::=   <relation exp> INTERSECT <relation exp>
```

< *relation exp* > 必须不是 < *nonproject* > 。除非其中之一或两者都是另一个 < *intersect* > 。

```
<minus>
    ::=   <relation exp> MINUS <relation exp>
```

< *relation exp* > 必须不是 < *nonproject* > 。

```
<times>
    ::=   <relation exp> TIMES <relation exp>
```

< *relation exp* > 必须不是 < *nonproject* > 。除非其中之一或两者都是另一个 < *times* > 。

```
<where>
    ::=   <relation exp> WHERE <bool exp>
```

< *relation exp* > 必须不是 < *nonproject* > 。 < *bool exp* > 可以引用 < *relation exp* > 所指示的关系的属性，它的语义很明显。

```
<join>
    ::=   <relation exp> JOIN <relation exp>
```

< *relation exp* > 必须不是 < *nonproject* > 。除非其中之一或两者都是另一个 < *join* > 。

```
<divide>
    ::=   <relation exp> DIVIDEBY <relation exp> PER <per>
```

< *relation exp* > 必须不是 < *nonproject* > 。

```
<per>
    ::=   <relation exp> | ( <relation exp>, <relation exp> )
```

< *relation exp* > 必须不是 < *nonproject* > 。

```
<with exp>
    ::=   WITH <name intro commalist> : <exp>
```

在本书中我们主要关注 < *with exp* > 是关系表达式的情况，这也是为什么我们要在本章中讨论它的原因。然而， < *with exp* > 也可以支持标量和元组；事实上，一个给定的 < *with exp* > 是 < *relation exp* > 、 < *tuple exp* > 和 < *scalar exp* > 中的哪一情形，是由冒号后面的 < *exp* > 决定的，也就是 < *with exp* > 的实际情形与该 < *exp* > 的情形相一致。无论是什么情况，每一个 < *name intro* > 将按照它们的书写顺序被执行； < *with exp* > 的语义通过 < *exp* > 的实际值来定义—— < *exp* > 中每一个引入的名称将由该名称变量的实际值（也就是相应表达式的执行结果）替换。注意：WITH 不是关系代数中的操作，而是用来描述那些复杂表达式（尤其是那些含有逗号的子表达式）的符号而已。我们将在 7.5 节给出一些例子。

```
<name intro>
    ::=   <exp> AS <introduced name>
```

如果 < *exp* > 允许的话（必要的时候可以引入括号），< *introduced name* > 可以嵌入到 < *with exp* > 中。

## 7.4 基本代数：语义

### 1. 并

数学中两个集合的并是这两个集合的所有元素组成的集合。因为一个关系是（或说包含）一个集合（一个元组的集合），所以构造这样两个集合的并是完全可能的；所得结果包含了出现在任一个或两个原关系中的所有元组。例如，关系变量 S 中的供应商元组的集合与关系变量 P 中的零件元组的集合的并当然是一个集合。

然而，尽管这一结果是一个集合，却不是一个关系；关系不能含有不同类型的元组，且其中的元组必须是同类的。当然，我们希望结果是一个关系，因为要保持封闭性。所以，关系代数中的并不是通常数学中的并；它是一种特殊类型的并，要求两个参与操作的关系是**同一类型**——即它们或者只包含供应商元组，或者只包含零件元组，而不能是两者的混合。如果两个关系属于同一类型，那就可以进行并操作，得到的结果是一个相同类型的关系；换句话说，封闭的特性被保持了下来。*注意*：历史上，很多数据库文献（也包含本书的早期版本）用合并兼容性（union compatibility）这个词来描述两个关系具有相同类型的思想。然而，由于诸多原因这个词并不是非常恰当；其中最明显的原因是其兼容性思想并不仅仅使用于并（union）。

下面是关系并操作的定义：给定两个相同类型的关系 *a* 和 *b*，两者的**并**，即 *a* UNION *b*，是相同类型的一个关系；该关系的主体由出现在 *a* 中或 *b* 中或同时出现在两者之中的所有元组组成。

*例如*：假设关系 *A* 和 *B* 如图 7-2 所示（都从供应商关系变量 S 的当前值导出；*A* 是在伦敦的供应商，*B* 是提供零件 P1 的供应商）。*A* UNION *B*（请看该图的第 1 部分）就是或者在伦敦、或者提供零件 P1 的供应商（或者两者兼有）。*注意*：结果有 3 个（而不是 4 个）元组；重复的元组按照定义被删除掉了（宽泛地讲，并操作取消了重复）。其他涉及删除重复元组的操作符只有投影（在本节的后面将会讲到）。

A

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |

B

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |

1. *Union*
(A UNION B)

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |
| S2 | Jones | 10 | Paris |

2. *Intersection*
(A INTERSECT B)

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |

3. *Difference*
(A MINUS B)

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S4 | Clark | 20 | London |

4. *Difference*
(B MINUS A)

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |

图 7-2 并、交、差举例

顺便说一下，两个关系的并依赖于元组等同性，下面介绍一个和前面等价但是表述方式不同的定义，它很好地说明了元组等同性（修订版更加强调这一点）：假定关系 *a* 和 *b* 具有相同的类型，它们的主体由这样的元组 *t* 组成，元组 *t* 等于（也就是一个副本）*a* 中或者 *b* 中，或者在 *a*，*b* 中同时存在的一个元组。类似的定义同样适用于交和差的操作，读者将在随后的部分看到。

### 2. 交

由于和并基本相同的原因，关系交操作符的操作对象必须是相同类型的。给定类型相同的关

系 *a* 和 *b*，它们的**交** *a* INTERSECT *b* 是一个相同类型的关系，关系的主体包含同时出现在 *a* 和 *b* 中的所有元组。

例如：假设关系 *A* 和 *B* 如图7-2所示，*A* INTERSECT *B*（请看该图的第2部分）就是既在伦敦又提供零件P1的供应商。

**3. 差**

像并和交一样，关系的差操作符也要求操作对象是同一类型。给定两个类型相同的关系 *a* 和 *b*，它们的**差** *a* MINUS *b*（两者有先后次序）是一个相同类型的关系，关系的主体包含属于 *a* 但不属于 *b* 的所有元组。

例如：*A* 和 *B* 如图7-2所示。*A* MINUS *B*（请看图的第3部分）是在伦敦、但不提供零件P1的供应商，*B* MINUS *A*（请看图的第4部分）是提供零件P1、但不在伦敦的供应商。注意MINUS有一个顺序性，就像通常数学中的减法（“5-2”和“2-5”不是同一件事情）。

**4. 积**

数学里两个集合的笛卡尔积是满足如下条件的有序对的集合：每个有序对的第一个元素来自第一个集合，第二个元素来自第二个集合。因此，两个关系的笛卡尔积可粗略地说是有序元组对的集合。但我们想保持封闭的特性，换句话说，我们想要结果包含元组，而不是有序的元组对。因此，关系的笛卡尔积是对这一操作的扩充，其中每个有序元组对代替以两个相关元组相并得出的一个元组（这里的“并”是一般集合理论上的并，而不是特殊的关系意义上的）。也就是说，给定元组[⊖]：

```
{ A1 a1, A2 a2, ..., Am am }
```

和

```
{ B1 b1, B2 b2, ..., Bn bn }
```

两者的并是单个元组：

```
{ A1 a1, A2 a2, ..., Am am, B1 b1, B2 b2, ..., Bn bn }
```

注意，为了简化，我们在这里假定两个元组间没有相同的属性，下面的段落将对此作详细的说明。

笛卡尔积中存在的另一个问题是：需要结果关系有一个正确形式的表头（即正确的关系类型）。现在已明确的是，结果的表头包含了两个输入关系的所有属性。如果两个关系的表头有共同的属性名，问题就会出现了；如果操作允许，结果的表头会有两个相同名称的属性，这就不再是“结构良好”（well-formed）了。如果对两个有相同属性名称的关系进行笛卡尔积操作，必须首先用RENAME操作符适当地更改属性的名称。

因此我们如下定义两个关系 *a* 和 *b* 的**笛卡尔积**：*a* TIMES *b*，其中 *a* 和 *b* 没有共同的属性名称，两者的笛卡尔积是一个关系，它的表头是 *a* 和 *b* 表头的并（集合并），主体包括所有 *a* 中的元组和 *b* 中的元组进行并操作而得到的元组。注意，结果的基数是 *a* 的基数和 *b* 的基数的乘积，结果的度是 *a* 的度和 *b* 的度的和。

例如：关系 *A* 和 *B* 如图7-3所示（*A* 是所有的当前供应商号码，*B* 是所有的当前零件号码）。于是 *A* TIMES *B*（参看图7-3的下半部分）就是所有的供应商号码和零件号码对。

**5. 选择**

假设关系 *a* 含有属性 *X* 和 *Y*（也可能还有别的），假设 $\theta$ 是一个比较操作符，诸如“=”、“≠”、“>”、“<”等；因此，布尔表达式 *X* $\theta$ *Y* 是结构良好的，对 *X* 和 *Y* 赋予具体的值，则能计算出真假值（true或false）。因此，关系 *a* 在属性 *X* 和 *Y* 上（*X* 和 *Y* 有前后顺序）的 $\theta$ **选择**（***θ-restriction***）（或者简称选择）

---

⊖ **Tutorial D** 要求在每一个这样的表达式前出现关键字TUPLE。

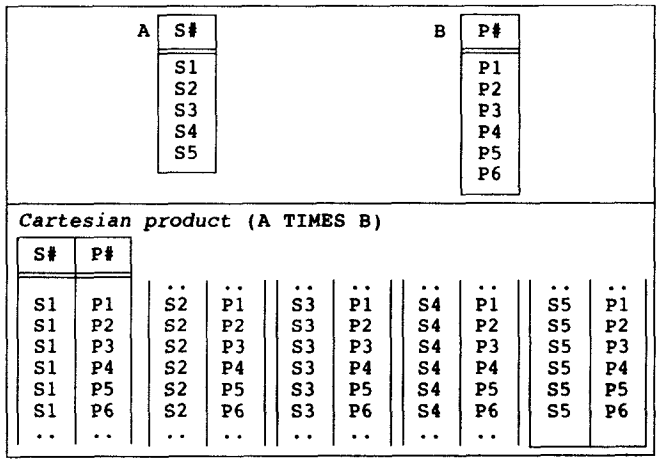
```
a WHERE X θ Y
```

是一个关系，关系的表头和 $a$ 的一样，主体包括所有满足条件 $X\ \theta\ Y$ 为真的元组。

注意：前面对选择的定义是在众多著作中公认的一种（包括本书以前的版本），但是我们却可以做如下的推广：假定关系 $a$ 有属性 $X,Y,\cdots,Z$（可能还有其他的属性），假设 $p$ 是布尔函数，它的参数，准确地说是属性 $X,Y,\cdots,Z$ 的一个子集。这样，关系 $a$ 在 $p$ 上的选择

```
a WHERE p
```

的结果是一个关系，具有和 $a$ 完全相同的表头，它的主体由那些令 $p$ 为 TRUE 的元组构成。



A

| S# |
|---|
| S1 |
| S2 |
| S3 |
| S4 |
| S5 |

B

| P# |
|---|
| P1 |
| P2 |
| P3 |
| P4 |
| P5 |
| P6 |

*Cartesian product* (A TIMES B)

| S# | P# | .. | .. | .. | .. | .. | .. | .. | .. |
|---|---|---|---|---|---|---|---|---|---|
| S1 | P1 | S2 | P1 | S3 | P1 | S4 | P1 | S5 | P1 |
| S1 | P2 | S2 | P2 | S3 | P2 | S4 | P2 | S5 | P2 |
| S1 | P3 | S2 | P3 | S3 | P3 | S4 | P3 | S5 | P3 |
| S1 | P4 | S2 | P4 | S3 | P4 | S4 | P4 | S5 | P4 |
| S1 | P5 | S2 | P5 | S3 | P5 | S4 | P5 | S5 | P5 |
| S1 | P6 | S2 | P6 | S3 | P6 | S4 | P6 | S5 | P6 |
| .. | .. | .. | .. | .. | .. | .. | .. | | |

图 7-3 笛卡尔积举例

选择操作符能有效地产生给定关系的横向（horizontal）子集——即给定关系元组中满足特定选择条件的元组子集。图 7-4 给出了几个例子（它们将阐明刚才对选择所作的抽象定义）。

```
S WHERE CITY = 'London'
```

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S4 | Clark | 20 | London |

```
P WHERE WEIGHT <
        WEIGHT (14.0)
```

| P# | PNAME | COLOR | WEIGHT | CITY |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |

```
SP WHERE S# = S# ('S6')
   OR    P# = P# ('P7')
```

| S# | P# | QTY |
|---|---|---|
| | | |

图 7-4 选择举例

这里需说明几点：

1）关键字 WHERE 后面的表达式 $p$ 是个布尔表达式，但事实上，它在某种意义上可以看作是第 9 章所介绍的谓词。

2）我们将谓词作为**选择条件（restriction condition）**。对于某个元组，如果它可以不用考虑该表上的其他元组就能对选择条件求值，那它就是一个简单选择条件。从这个意义上讲，图 7-4 上的所有选择条件都是简单的，但这里还要给出一个非简单选择条件的例子：

```
S WHERE ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# }
```

我们将在后面介绍除法操作时再详细地讨论这个例子。

3）下面这些等价的式子也值得注意：

```
a WHERE p1 OR p2   ≡  ( a WHERE p1 ) UNION ( a WHERE p2 )
a WHERE p1 AND p2  ≡  ( a WHERE p1 ) INTERSECT ( a WHERE p2 )
a WHERE NOT ( p )  ≡  a MINUS ( a WHERE p )
```

**6. 投影**

假定关系 $a$ 有属性 $X$, $Y$,…, $Z$（可能还有其他属性）。关系 $A$ 在 $X$, $Y$,…, $Z$ 上的投影——$a\{X, Y,\cdots, Z\}$是一个满足如下条件的关系：

- 表头由 $a$ 的表头除去不包含在集合$\{X, Y,\cdots, Z\}$中的属性而得到。
- 主体包含所有形式为$\{X\,x, Y\,y,\cdots, Z\,z\}$的元组；且关系 $a$ 存在这样的元组，其属性 $X$, $Y$, …,$Z$ 的值分别为 $x$, $y$,…, $z$。

投影操作能有效地产生给定关系的垂直（vertical）子集，该子集是由除去不包含在指定列表中的属性，且消除由此产生的重复（子）元组而得出的。

几点解释：

1）在属性名称的列表中，不能有重复的属性（为什么不能?）。

2）现实中经常出现这种情况：指定被投影掉（也就是被删除掉）的属性比指定投影的属性更方便。例如，我们可能会说“从关系 P 中投影掉属性 WEIGHT”，而不说“关系 P 投影在属性 P#、PNAME、COLOR 和 CITY 上”，就像下面这样：

```
P { ALL BUT WEIGHT }
```

图 7-5 给出了几个投影的例子。注意第一个例子（供应商在 CITY 上的投影），尽管关系变量 S 当前含有 5 个元组，即有 5 个城市，但结果中只有 3 个城市——重复的城市（即重复的元组）被删除了。在其他例子上有类似的情况。

S { CITY }

| CITY |
|---|
| London |
| Paris |
| Athens |

P { COLOR, CITY }

| COLOR | CITY |
|---|---|
| Red | London |
| Green | Paris |
| Blue | Oslo |
| Blue | Paris |

( S WHERE CITY = 'Paris' ) { S# }

| S# |
|---|
| S2 |
| S3 |

图 7-5　投影举例

**7. 连接**

**连接**（join）存在几种不同的变种。然而，最简单且最重要的是所谓的自然连接，实际上，按非正式的说法，连接总是特指自然连接，本书也采用这种用法。下面是有关定义（有点抽象，但读者应从第 3 章的讨论中对自然连接有些直观的了解）。假设关系 $a$ 和 $b$ 分别有属性：

```
X1, X2, ..., Xm, Y1, Y2, ..., Yn
```

和

```
Y1, Y2, ..., Yn, Z1, Z2, ..., Zp
```

即 $Y$ 属性 $Y1$, $Y2$,…, $Yn$ 是两个关系的共同属性，$X$ 属性 $X1$, $X2$, …, $Xm$ 是 $a$ 的特有属性，$Z$ 属性 $Z1$, $Z2$,…, $Zp$ 是 $b$ 的特有属性。注意：

- 有了属性重命名操作 RENAME，我们可以不失一般性地假定，没有属性 $Xi(i=1,2,3,\cdots, m)$和属性 $Zj(j=1,2,\cdots,p)$具有相同的名字。
- 在 $a$ 和 $b$ 中，每一个属性 $Yk(k=1,2,\cdots,n)$都有相同的类型（否则，按照定义，它们不可能是共同的属性）。

现在把$\{X1, X2, \cdots, Xm\}$、$\{Y1, Y2,\cdots, Yn\}$和$\{Z1, Z2,\cdots, Zp\}$分别看作是复合属性 $X$、$Y$ 和 $Z$。于是 $a$ 和 $b$ 的自然连接

```
a JOIN b
```

是一个关系。它的表头是{X,Y,Z}；主体包含所有元组{X x,Y y, Z z}，其中含有 X 的值 x 和 Y 的值 y 的元组出现在 a 中，含有 Y 的值 y 和 Z 的值 z 的元组出现在 b 中。

图 7-6 给出了一个自然连接（在共同属性 CITY 上的自然连接 S JOIN P）。

| S# | SNAME | STATUS | CITY | P# | PNAME | COLOR | WEIGHT |
|---|---|---|---|---|---|---|---|
| S1 | Smith | 20 | London | P1 | Nut | Red | 12.0 |
| S1 | Smith | 20 | London | P4 | Screw | Red | 14.0 |
| S1 | Smith | 20 | London | P6 | Cog | Red | 19.0 |
| S2 | Jones | 10 | Paris | P2 | Bolt | Green | 17.0 |
| S2 | Jones | 10 | Paris | P5 | Cam | Blue | 12.0 |
| S3 | Blake | 30 | Paris | P2 | Bolt | Green | 17.0 |
| S3 | Blake | 30 | Paris | P5 | Cam | Blue | 12.0 |
| S4 | Clark | 20 | London | P1 | Nut | Red | 12.0 |
| S4 | Clark | 20 | London | P4 | Screw | Red | 14.0 |
| S4 | Clark | 20 | London | P6 | Cog | Red | 19.0 |

图 7-6 自然连接 S JOIN P

注意：连接并不总是在外码和相对应的主码（或候选码）之间进行，尽管这样的连接很普通且很重要。关于这一点已经举例说了多遍，实际上图 7-6 也给出了说明，但仍需要讲清楚。

还需要注意的是，上述的自然连接的定义恰好建立在等值比较的基础上。考虑关于该定义的如下几方面：

- 如果 $n=0$（意味着 a 和 b 没有共同的属性），那么 a JOIN b 退化为 a TIMES b。[○]

如果 $m=p=0$（a 和 b 具有相同的类型），那么 a JOIN b 退化为 a INTERSECT b。

接下来考虑 θ 连接操作。θ 连接是扩展自然连接，以支持其他比较而不是相等比较的情况（这种情况相对来说比较少，但决不是没有）。假设关系 a 和 b 满足笛卡尔积的条件（即它们没有共同的属性名称）；又设 a 有属性 X，b 有属性 Y，且 X、Y 和 θ 满足 θ 选择的需求。于是关系 a 和 b 在属性 X 和 Y 上的 θ 连接，定义为如下表达式操作的结果：

```
( a TIMES b ) WHERE X θ Y
```

换句话说，其结果关系的表头和 a 与 b 的笛卡尔积的表头相同，主体包含满足以下条件的元组 t：t 出现在笛卡尔积中，且使条件“X θ Y”为真。

下面通过一个例子来说明，假设需要计算关系 S 和 P 在 CITY 上的大于连接（这里的 θ 是“>”；由于属性 CITY 被定义为 CHAR 型的，因此，大于在这里就简单地解释为“在字母的次序上大于”）。大致的关系表达式如下所示：

```
( ( S RENAME CITY AS SCITY ) TIMES
  ( P RENAME CITY AS PCITY ) )
WHERE SCITY > PCITY
```

请注意此例中对属性名称的修改（当然，只修改两个 CITY 名称中的一个就足够了；两个都修改是为了对称）。全部表达式的结果见图 7-7。

如果 θ 是“=”，则 θ 连接称为**等值连接**（equijoin）。由定义可知，等值连接的结果必须包含这样的两个属性：它们的值在关系的每个元组上都相等。如果这两个属性中的一个被投影掉且另一个相应地改名（若需要的话），则结果就是一个自然连接！例如，供应商和零件（在 CITY 上）的自然连接的表达式

```
S JOIN P
```

等价于下面更复杂的式子

---

○ 在文献 [3.3] 中定义的 **Tutorial D**，正是因为这个原因而没有直接支持 TIMES 操作。

```
( ( S TIMES ( P RENAME CITY AS PCITY ) )
                        WHERE CITY = PCITY )
                                { ALL BUT PCITY }
```

注意：**Tutorial D** 不直接支持 $\theta$ 连接操作符，因为（a）在实际中不经常需要；（b）它不是一个基本的操作符（即可以用别的操作符表示它，这我们已经看到过了）。

| S# | SNAME | STATUS | SCITY | P# | PNAME | COLOR | WEIGHT | PCITY |
|---|---|---|---|---|---|---|---|---|
| S2 | Jones | 10 | Paris | P1 | Nut | Red | 12.0 | London |
| S2 | Jones | 10 | Paris | P3 | Screw | Blue | 17.0 | Oslo |
| S2 | Jones | 10 | Paris | P4 | Screw | Red | 14.0 | London |
| S2 | Jones | 10 | Paris | P6 | Cog | Red | 19.0 | London |
| S3 | Blake | 30 | Paris | P1 | Nut | Red | 12.0 | London |
| S3 | Blake | 30 | Paris | P3 | Screw | Blue | 17.0 | Oslo |
| S3 | Blake | 30 | Paris | P4 | Screw | Red | 14.0 | London |
| S3 | Blake | 30 | Paris | P6 | Cog | Red | 19.0 | London |

图 7-7　供应商和零件在 CITY 上的大于连接

**8. 除**

参考文献［7.4］定义了两个不同的“除”操作：小除（small divide）和大除（great divide）。在 **Tutorial D** 中，< *per* > 项只含有一个 < *relation exp* > 的 < *divide* > 称为小除，含有两个 < *relation exp* > 的 < *divide* > 称为大除。下面的叙述只适用于小除，且只适用于特定形式的小除。关于大除的讨论和有关小除的更详细的内容请参看［7.4］。

应该说明，这里讲的小除与 Codd 所说的除不一样；实际上，在处理空关系时，原始的操作符有一定的困难，而小除是对原始版本的一个改进。它与本书前几个版本所讲的也不一样。

这里给出小除的定义。设关系 $a$ 和 $b$ 分别有属性：

```
X1, X2, ..., Xm
```

和

```
Y1, Y2, ..., Yn
```

这里，没有任何一个属性 $Xi(i=1,2,\cdots,m)$ 与属性 $Yj(j=1,2,\cdots,n)$ 具有相同的名字。设关系 $c$ 有属性：

```
X1, X2, ..., Xm, Y1, Y2, ..., Yn
```

（即 $c$ 的表头是 $a$ 和 $b$ 的表头的并）。现在把 {*X*1, *X*2, …, *Xm*} 和 {*Y*1, *Y*2, …, *Yn*} 分别看作是复合属性 $X$ 和 $Y$。于是 $a$ 根据 $c$ 除以 $b$（其中 $a$ 是被除数，$b$ 是除数，$c$ 是中间数）

```
a DIVIDEBY b PER c
```

得到的结果是一个关系，关系的表头是 {*X*}，主体包含符合如下条件的元组 {*X x*}：它来自 $c$ 的元组 {*X x*, *Y y*}，其中每个元组对应于 $b$ 中所有的元组 {*Y y*}。也可以粗略地说，结果关系包含 $a$ 中满足如下条件的 $X$ 值：在 $c$ 中对应的 $Y$ 值包含 $b$ 中的所有 $Y$ 值。这里再次强调元组的等价性。

图 7-8 介绍了几个除的简单例子。在每个例子中的被除数（DEND）是关系 S 在 S#上的投影；中间数（MED）是关系 SP 在 S#和 P#上的投影；三个除数（DOR）如图所示。特别要注意最后一个例子，其中的除数是一个包含所有已知零件号码的关系；结果显示了提供所有这些零件的供应商的号码。从例子中可以看出，DIVIDEBY 操作符用于查询共同的本质；实际上，若在自然语言的查询中包含“所有”（all）一类的词（“查询提供所有零件的供应商”），则极有可能用到除法。（实际上，Codd 特别地扩展除，使关系代数得以支持全称量词，正如扩展投影使得关系代数得以支持存在量词一样。进一步的解释请见第 8 章。）

然而，我们需要指出的是，用关系比较的方式来表达查询的本来思想通常更加通俗易懂（上一个例子就是这样）。例如：

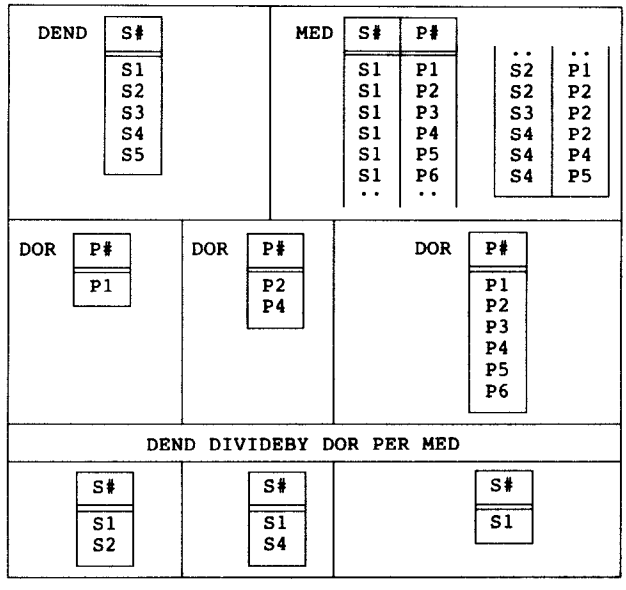



图7-8 除举例

```
S WHERE ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# }
```

这个表达式用来查找那些提供了所有零件的供应商。下面对它作一些必要的解释：

1）对于某一个供应商，这个表达式

```
( ( SP RENAME S# AS X ) WHERE X = S# ) { P# }
```

产生一个零件号的集合，这些零件都是由这个供应商所供应的。

2）这个零件号的集合接下来与包含所有零件号的集合相比较。

3）当且仅当两个集合完全相等时，其对应的供应商便是结果之一。

下面再与之对应地使用 DIVIDEBY 来解答：

```
S JOIN ( S { S# } DIVIDEBY P { P# } PER SP { S#, P# } )
```

你也许能够明显感觉出，使用关系比较的方式来处理要简单得多。事实上，如果关系模型首先考虑的是关系比较的方式——实际上并非如此，那么是否还需要定义 DIVIDEBY 便值得怀疑了。

## 7.5 举例

本节提供了用在查询中的有关关系代数表示的一些例子。建议读者对照图 3-8 中的数据检验这些例子。

### 例 7.5.1 求提供零件 P2 的供应商名称

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

解释：首先构造关系 SP 和 S 在供应商号码上的自然连接，从概念上说，它扩展了每个 SP 元组相应的供应商信息（即 SNAME、STATUS 和 CITY 的值）。随后这个连接以零件 P2 为条件进行了选择。最后选择在 SNAME 上做了投影。结果中只有 SNAME 这一个属性。

### 例 7.5.2 求提供至少一个红色零件的供应商名称

```
( ( ( P WHERE COLOR = COLOR ('Red') )
                    JOIN SP ) { S# } JOIN S ) { SNAME }
```

结果唯一的属性还是 SNAME。

下面给出对同一个查询的不同的表述：

```
( ( ( P WHERE COLOR = COLOR ('Red') ) { P# }
                    JOIN SP ) JOIN S ) { SNAME }
```

这个例子说明了一个重要的事实：对同一个查询经常有不同的表述。关于这一点在第 18 章还有讨论。

**例 7.5.3　求提供所有零件的供应商名称**

```
( ( S { S# } DIVIDEBY P { P# } PER SP { S#, P# } )
                                        JOIN S ) { SNAME }
```

或者

```
( S WHERE
( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# } )
{ SNAME }
```

其结果还是只有一个属性：SNAME。

**例 7.5.4　求至少提供了 S2 提供的所有零件的供应商号码**

```
S { S# } DIVIDEBY ( SP WHERE S# = S# ('S2') ) { P# }
                                        PER SP { S#, P# }
```

结果只有一个属性：S#。

**例 7.5.5　求住在同一个城市的供应商的号码对（若两个供应商住在同一城市，则它们的号码是一对）**

```
( ( ( S RENAME S# AS SA ) { SA, CITY } JOIN
    ( S RENAME S# AS SB ) { SB, CITY } )
        WHERE SA < SB ) { SA, SB }
```

结果包含两个属性：SA 和 SB（当然，只改掉两个 S#属性之一的名称就足够了；这里两者都改是为了对称）。注意，我们假设在类型 S#上已经定义了操作符“<”。条件 SA < SB 具有两重作用：

- 它排除了形式为 $(x, x)$ 的供应商号码对；
- 它保证了 $(x, y)$ 和 $(y, x)$ 不会同时出现。

下面给出这个查询的另一种表示，目的是为了说明如何利用 WITH 简化查询表达式。[⊖]

```
WITH ( S RENAME S# AS SA ) { SA, CITY } AS T1,
     ( S RENAME S# AS SB ) { SB, CITY } AS T2,
     T1 JOIN T2 AS T3,
     T3 WHERE SA < SB AS T4 :
     T4 { SA, SB }
```

有了 WITH，我们就可以考虑那些分步表述（step-at-a-time）的复杂表达方式了；但它并不违反关系代数的非过程化特性。在下一个例子中将对这方面做详细的阐述。

**例 7.5.6　求不提供零件 P2 的供应商名称**

```
( ( S { S# } MINUS ( SP WHERE P# = P# ('P2') ) { S# } )
                                        JOIN S ) { SNAME }
```

---

⊖ 事实上，第 5 章 5.5 节里定义 DIST 时已经使用过 WITH 的标量形式，而在第 6 章的 6.5 节里讨论扩展的操作符 UPDATE 时，也已向读者展示过它的关系形式。

其结果只有一个叫做 SNAME 的属性。

正像前面所说的，我们要对这个例子进行详细的阐述。迅速判断怎样把一个给定的查询表述成一个嵌套表达式并不总是很简单的，也没有必要如此做。用分步表述来描述本例如下：

```
WITH S { S# } AS T1,
     SP WHERE P# = P# ('P2') AS T2,
     T2 { S# } AS T3,
     T1 MINUS T3 AS T4,
     T4 JOIN S AS T5,
     T5 { SNAME } AS T6 :
     T6
```

T6 代表想要的结果。解释：WITH 子句引进名称——例如形如 T*i* 的名称——对包含子句的声明假定是局部意义上的。如果系统支持惰性计算（例如像 PRTV 系统所做的，[7.9]），那么把整个查询分解为这种形式的一连串的步骤必须不能产生不良影响。相反，查询可以按下面来进行：

- 冒号前面的表达式系统不能直接赋值——系统所做的只是记住它们和被 AS 子句引进的名称。
- 冒号后面的表达式代表了查询的最终结果（在这个例子中，表达式只是“T6”）。当运行到这里时，系统就会计算出想要的值（即 T6 的值）。
- 为了计算 T6（它是 T5 在 SNAME 上的投影），系统需要首先计算 T5；为了计算 T5（T5 是 T4 和 S 的连接），系统需先计算 T4；等等。换句话说，系统不得不计算原先的嵌套表达式，就像用户在前面写了嵌套表达式一样。

关于计算嵌套表达式这个问题，在下一节会有一个简要的讨论，在第 18 章也有进一步的阐述。

## 7.6 关系代数的作用

总结一下本章中目前所讲的内容：对关系代数作了定义，它是一个关系操作的集合。涉及的操作有并、交、差、积、选择、投影、连接和除，还有一个给属性改名的操作符 RENAME（除了 RENAME，这些基本上就是 Codd 在 [7.1] 中最初定义的集合）。我们还提供了这些操作的语法，利用这些语法给出了许多例子和说明。

正如所讨论的，Codd 的 8 个操作符不是最小的集合（也从未打算成为一个最小集合），因为它们中的某几个并不是**基本的**——即可以用其他的操作符来定义。例如，连接、交和除这三个操作符可以用其余的 5 个来定义（参看习题 7.6），因此删除它们也不会丢失任何功能。剩下的 5 个（选择、投影、积、并和差）中的任意一个都不能用其他 4 个来定义，因此称它们为一个基本的或最小的集合（请注意，当然不仅有这一个最小集合）[⊖]。实际上，其余的 3 个操作符（特别是连接）是非常有用的，一个好的系统可以直接支持它们。

现在需要澄清重要的一点——尽管没有明确说明，但本章的内容到目前为止好像显示了代数的主要作用只是数据存取。事实上并非如此。代数的基本目的是**书写关系表达式**。那些表达式有各种用途，当然也包括存取，但并不只限于此。下面的功能（并不限于这些）显示了这些表达式的一些可能应用情形：

- 定义**检索**的范围——即定义在某检索操作中将获得的数据（已经详细讨论过了）；
- 定义**修改**的范围——即定义在修改操作中被插入、改变或删除的数据（参考第 6 章）；
- 定义**完整性约束**——即定义数据库必须满足的一些约束（参考第 9 章）；
- 定义**派生的关系变量**——即定义包含在视图或映射中的数据（参考第 10 章）；
- 定义**一致性需求**——即定义在并发控制操作中的数据（参考第 16 章）；

---

⊖ 对于这个说法我们需要一些特定的限制。首先，我们可以看到，积是一种特殊的连接，因此可以对一些特定的集合使用连接来代替积；其次，我们有必要将 RENAME 包括进来，这是因为我们的代数（不像文献 [7.1] 所说的那样）依赖于属性的名称，而不是属性的位置；最后，文献 [3.3] 介绍了关系代数的一种“简化的指令集”——称为 *A*，它只包含两个基本操作 remove 和 nor，就能够表示 Codd 的基本代数（以及 RENAME 和其他几个有用的操作）的所有功能。

■ 定义**安全性约束**——即定义被授予某种权限的数据（参考第 17 章）。

实际上，这些表达式是对用户意图的高层次的、符号式的表述（例如针对一些特殊的查询）。正因为它们是高层次和符号形式的，它们可以根据一系列高层次和符号形式的**转换规则**来操作。例如，表达式

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

（见例子 7.5.1——“提供零件 P2 的供应商名称”）可以被转换为另一个等价的表达式，并且效率可能更高，即

```
( ( SP WHERE P# = P# ('P2') ) JOIN S ) { SNAME }
```

（练习：从何种意义上说第二种表述效率可能更高？为何是“可能”？）

代数因此可以作为**优化**的基础（如果想复习优化的概念，可回顾 3.5 节）。即使用户使用了上面的第一种表达式，在执行前，优化系统会把它转换为第二种形式（理想的情况是，一个给定查询的执行不应依赖于用户提交的方式）。在第 18 章中将有进一步讨论。

*注意*：代数由于它的基本特征而经常被用作衡量一种语言表达能力的尺度。一种语言基本上可以说是**关系完备**［7.1］的，当它至少拥有代数的作用——即当它的表达式允许通过代数（前几节中描述的基本的代数）的形式来定义每一个关系。下一章将详细讨论关系完备的概念。

## 7.7　深入讨论

这一节介绍 8 个基本操作符的其他一些性质。

### 1. 结合律和交换律

不难看出，并操作具有**结合性**——即对于任意的相同类型的关系 *a*，*b*，*c*，下列表达式

```
( a UNION b ) UNION c
```

和

```
a UNION ( b UNION c )
```

在逻辑上是等价的。为了方便，我们允许并的结果不带任何括号，所以，前面的表达式可以简化为

```
a UNION b UNION c
```

结合律对于交、积和连接也同样适用（但是减除外）。*注意*：正是由于它的这个特性，也许在现实中，前缀表示法例如 UNION(*a*,*b*,*c*)，比中缀表示法（**Tutorial D** 中就是使用这种方式）更加合适。但是本书中坚持使用中缀表示法。

并、交、积和连接（减除外）也都具有**交换性**——即表达式

```
a UNION b
```

和

```
b UNION a
```

在逻辑上也是等价的。交、积和连接同样如此。

我们将在第 18 章重新讨论与结合性和交换性有关的问题。顺便提一下，对于笛卡尔积，其集合论形式既不具有交换性也不具有结合性，但是（正如我们看到的一样）它的关系形式却具有这两种性质。

### 2. 一些恒等式

在这个小节，我们列出了一些很重要的恒等式，但不作深入的讨论。在下列式子中 *r* 可表示任意的关系，*empty* 表示与 *r* 具有相同类型的空的关系。

- *r* WHERE TRUE ≡ *r*（同一选择）
- *r* WHERE FALSE ≡ *empty*

- $r\{X,Y,\ldots,Z\} \equiv r$（如果 $X$，$Y$，…，$Z$ 是关系 $r$（单位投影）的所有属性）
- 如果 $r$ = *empty* 那么 $r\{\} \equiv$ TABLE_DUM，否则 $r\{\} \equiv$ TABLE_DEE（空投影）
- $r$ JOIN $r \equiv r$ UNION $r \equiv r$ INTERSECT $r \equiv r$
- $r$ JOIN TABLE_DEE ≡ TABLE_DEE JOIN $r \equiv r$（DEE 是连接的单位元，正如普通的数学运算中，0 是加法的单位元，1 是乘法的单位元）
- $r$ TIMES TABLE_DEE ≡ TABLE_DEE TIMES $r \equiv r$（这个恒等式是上一个等式的一种特例）
- $r$ UNION *empty* ≡ $r$ MINUS *empty* ≡ $r$
- *empty* INTERSECT $r \equiv$ *empty* MINUS $r \equiv$ *empty*

### 3. 一些普遍性

连接、并和交最初定义的时候都是作为二元操作符（即操作数需要两个关系）[⊖]；但是，我们知道，它们都可以推广到 $n$ 元的情况，其中 $n$ 为任意大于 1 的数。但是，对于 $n$ 等于 1，或是 $n$ 等于 0 的情况呢？现实（至少是从概念层次上）却是需要在只有一个关系或者没有关系的情况下能够执行“连接”、“并”、“交”等操作（甚至 **Tutorial D** 也没有在语法上支持这些操作）。假设 $s$ 是一个关系集合（当并和交时，所有关系要求具有相同的类型 *RT*），那么它们的定义如下：

- 如果集合 $S$ 只包含一个关系 $r$，则在 $S$ 上所做的连接、并和交操作的结果如同作用在单个关系 $r$ 上一样。
- 如果集合 $S$ 不含有任何关系，那么
  - 在 $S$ 上的连接操作的结果，则定为 TABLE_DEE（连接的单位元素）。
  - 在 $S$ 上的并操作的结果，则定为一个 *RT* 类型的空关系。
  - 在 $S$ 上的交操作的结果，则定为一个 *RT* 类型的全体表——该表包含所有满足关系类型 *RT* 的表头要求的元组。[⊜]

## 7.8 附加的操作符

自从 Codd 定义了他的 8 个操作符以来，很多人都提出了新的代数操作符。这一节较为详细地分析其中的几个操作符——半连接（SEMIJOIN）、半减（SEMIMINUS）、扩展（EXTEND）、合计（SUMMARIZE）和传递闭包（TCLOSE）等。根据 **Tutorial D** 的语法，这些操作符包含 <*nonproject*> 的 5 个新形式，说明如下：

```
<semijoin>
    ::=   <relation exp> SEMIJOIN <relation exp>
<semiminus>
    ::=   <relation exp> SEMIMINUS <relation exp>
<extend>
    ::=   EXTEND <relation exp> ADD ( <extend add commalist> )
```

如果逗号列表仅仅包含有一个 <*extend add*>，那么圆括号可以省略。

```
<extend add>
    ::=   <exp> AS <attribute name>
<summarize>
    ::=   SUMMARIZE <relation exp> PER <relation exp>
              ADD ( <summarize add commalist> )
```

如果逗号列表仅仅包含有一个 <*summarize add*>，那么圆括号可以省略。

```
<summarize add>
    ::=   <summary type> [ ( <scalar exp> ) ]
                         AS <attribute name>
<summary type>
    ::=   COUNT  | SUM  | AVG  | MAX | MIN | ALL | ANY
        | COUNTD | SUMD | AVGD | ...
<tclose>
    ::=   TCLOSE <relation exp>
```

---

⊖ 减也是二元的；但是，选择和投影都是一元操作符。

⊜ 我们注意到，在其他文献中，全体表（universal relation）这个词被当作别的意思来使用。请看文献 [13.20]。

在前面的 BNF 产生式规则中提到的各个 < *relation exp* > 必须不是 < *nonproject* > 。

**1. 半连接**

假定 *a*、*b*、*X*、*Y* 和 *Z* 如 7.4 节的“连接”小节中所示。于是 *a* 和 *b*（按照先 *a* 后 *b* 次序）的**半连接**（semijoin）——*a* SEMIJOIN *b*——可以等价地定义为

```
( a JOIN b ) { X, Y }
```

换句话说，*a* 和 *b* 的半连接就是 *a* 和 *b* 的连接在 *a* 的属性上的投影。宽泛地讲，操作结果的主体就是在 *b* 中有对应值的 *a* 的元组。

示例：求提供零件 P2 的供应商的 S#、SNAME、STATUS 和 CITY。

```
S SEMIJOIN ( SP WHERE P# = P# ('P2') )
```

在前面我们已经注意到，现实中很多用到连接的查询实际上就隐含着半连接——这意味在实际应用中对半连接有着极大的需求。半减（也叫半差）操作也是类似的（请看下一个小节）。

**2. 半差**

*a* 和 *b* 的**半差**（semidifference）（按照先 *a* 后 *b* 次序），*a* SEMIDIFFERENCE *b*，可以等价地定义为

```
a MINUS ( a SEMIJOIN b )
```

宽泛地讲，操作结果的主体是在 *b* 中没有对应值的 *a* 的元组。

示例：求不提供零件 P2 的供应商的 S#、SNAME、STATUS 和 CITY。

```
S SEMIMINUS ( SP WHERE P# = P# ('P2') )
```

**3. 扩展**

读者可能已经注意到，到目前为止所描述的代数还没有计算的能力。然而，在实际中这样的能力是明显需要的。例如，我们可能想得到形如 WEIGHT * 454 的算术表达式的值，或者遇到 WHERE 子句中的这样一个值（给定的零件的重量是以磅为单位的；表达式 WEIGHT * 454 会把它转换为克[⊖]）。**扩展**（extend）操作的目的是支持这样的能力。更精确地说，EXTEND 接受一个关系，然后返回一个关系，返回的关系除了新增一个属性外，其余与给定关系完全相同，新增属性的值通过计算指定操作表达式而获得。例如，可以写：

```
EXTEND P ADD ( WEIGHT * 454 ) AS GMWT
```

请注意，这是一个表达式而不是一个命令或者声明，因此它可以嵌套在其他表达式中。它将产生一个关系，此关系的表头除新增一个名叫 GMWT 的属性外，其余和 P 相同。除了根据特定算术表达式得到的 GMWT 之外，关系的每个元组与 P 的元组相同。参看图 7-9。

重点：请注意，表达式 EXTEND 并没有改变数据库中的零件关系变量；它只是一个表达式，就像其他表达式一样只是指示了一个结果——在本例中恰巧看起来像零件关系变量的一个当前值。（换句话说，EXTEND 不是 SQL 的 ALTER TABLE...ADD COLUMN 语句在关系代数中的相应操作）。

| P# | PNAME | COLOR | WEIGHT | CITY | GMWT |
|---|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London | 5448.0 |
| P2 | Bolt | Green | 17.0 | Paris | 7718.0 |
| P3 | Screw | Blue | 17.0 | Oslo | 7718.0 |
| P4 | Screw | Red | 14.0 | London | 6356.0 |
| P5 | Cam | Blue | 12.0 | Paris | 5448.0 |
| P6 | Cog | Red | 19.0 | London | 8626.0 |

图 7-9　一个关于 EXTEND 的例子

现在，我们可以在投影和选择等操作中使用 GMWT。例如：

```
( ( EXTEND P ADD ( WEIGHT * 454 ) AS GMWT )
          WHERE GMWT > WEIGHT ( 10000.0 ) ) { ALL BUT GMWT }
```

⊖ 假定在整数和重量之间的乘（*）是一个合法的操作符。这样的操作的结果是什么呢？

注意：当然，在 WHERE 子句应该可以直接使用更友好的语言来描述计算表达式，比如：

```
P WHERE ( WEIGHT * 454 ) > WEIGHT ( 10000.0 )
```

（请看 7.4 节中关于选择的讨论），但是这样的特征实际上是语法的友好性。

一般而言，扩展表达式

```
EXTEND a ADD exp AS Z
```

的结果是一个关系，定义如下：

- 其表头等于关系 *a* 的表头扩加了属性 Z；
- 其主体包含所有的元组 *t*，*t* 是在 *a* 的元组上扩充了新属性 Z 而得到的元组，Z 属性的值通过计算相应的 *a* 元组的 *exp* 表达式而得到。

关系 *a* 不能含有名为 Z 的属性，并且 *exp* 不能涉及 Z；可以看出结果的基数等于 *a* 的基数，结果的度等于 *a* 的度加 1。结果中 Z 的类型是 *exp* 的类型。

下面又是几个例子：

1) `EXTEND S ADD 'Supplier' AS TAG`

这个表达式利用字符串“Supplier”标记了关系变量 S 当前值的每个元组（文字是计算表达式的一个特例，而文字更一般地是一个选择子调用）。

2) `EXTEND ( P JOIN SP ) ADD ( WEIGHT * QTY ) AS SHIPWT`

本例是对一个关系表达式的扩展，这比一个简单的关系变量更为复杂。

3) `( EXTEND S ADD CITY AS SCITY ) { ALL BUT CITY }`

形如 CITY 的一个属性名称也是合法的计算表达式。这个例子等价于

```
S RENAME CITY AS SCITY
```

换句话说，RENAME 不是最基本的操作——它可以用 EXTEND（或投影）来定义。当然，由于使用方便的缘故，我们并不想放弃所熟悉的 RENAME 操作符，但它只是一个捷径。

4) `EXTEND P ADD ( WEIGHT * 454 AS GMWT, WEIGHT * 16 AS OZWT )`

这是多扩展的例子。

5)
```
EXTEND S
ADD COUNT ( ( SP RENAME S# AS X ) WHERE X = S# )
AS NP
```

这个表达式的结果在图 7-10 中显示。**解释**：

- 对于给定的供应商，表达式

```
( ( SP RENAME S# AS X ) WHERE X = S# )
```

产生了该供应商的一组发货情况。

- 聚集操作符 COUNT 作用于发货元组的集合，返回相应的基数（当然是一个标量值）。

结果中的属性 NP 代表供应商提供的零件的数量，供应商由相应的 S#的值指定。特别注意对应于供应商 S5 的 NP 值；S5 的 SP 元组的集合是空的，因此 COUNT 返回零。

| S# | SNAME | STATUS | CITY | NP |
|---|---|---|---|---|
| S1 | Smith | 20 | London | 6 |
| S2 | Jones | 10 | Paris | 2 |
| S3 | Blake | 30 | Paris | 1 |
| S4 | Clark | 20 | London | 3 |
| S5 | Adams | 30 | Athens | 0 |

图 7-10 EXTEND 的另一个例子

下面对**聚集操作符**作一下说明。一般地讲，聚集操作符的目的是从特定关系的特定属性值中得出一个数字。典型的例子是 COUNT、SUM、AVG、MAX、MIN、ALL 和 ANY。在 **Tutorial D** 中，一个聚集操作符调用 < *agg op inv* >（因为它返回一个数字，所以是 < *scalar exp* > 的一个特例）采用下面的一般形式：

```
<agg op name> ( <relation exp> [, <attribute name> ] )
```

如果 < *agg op name* > 是 COUNT，则 < *attribute name* > 是无关的，且必须省略掉；否则，当

且仅当 < *relation exp* > 是一个单目的关系，< *attribute name* > 才被省略掉，在这种情况下，< *relation exp* > 的值的唯一属性假定为默认值。下面是两个例子：

```
SUM ( SP WHERE S# = S# ('S1'), QTY )
SUM ( ( SP WHERE S# = S# ('S1') ) { QTY } )
```

注意两者的区别，第一个得出供应商 S1 的所有发货数量的总计，第二个得出 S1 的所有不同的发货数量的和。

如果一个聚集操作符的参数恰好是一个空集，则 COUNT 返回零（就像所看到的），SUM 也一样；MAX 和 MIN 分别返回相关域的最小值和最大值；ALL 和 ANY 分别返回 TURE 和 FALSE；AVG 会引起一个例外。

4. 合计

首先说明这里讨论的 SUMMARIZE 不同于本书以前版本中所讲的，实际上，它是一个改进版，它克服了由于关系空值所引起的问题。

我们已经看到，*extend* 操作符提供了一种途径，把“水平”（horizontal）或“行方式”（row-wise）计算结合进了关系代数。合计（summarize）操作符执行类似的功能——“垂直”（vertical）或“列方式”（column-wise）计算。例如，表达式

```
SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY
```

产生了一个表头为 ｛P#, TOTQTY｝ 的关系，它对应于投影 P ｛P#｝ 的每个 P#值有一个元组，内含 P#的值和相应的合计数量（见图 7-11）。换言之，关系 SP 从概念上把元组分成多个组或者集合（每个 P#的值产生一个集合），然后，这样的每一个组在最终结果中产生一个相应的元组。

一般地说，表达式

```
SUMMARIZE a PER b ADD summary AS Z
```

的值是一个关系，定义如下：

- 首先，*b* 必须和 *a* 的某些投影具有相同的类型；即 *b* 的所有属性必须都是 *a* 的属性。假设投影的属性（对 *b* 来说是相同的）是 *A*1,*A*2,…,*An*。
- 结果的表头包含有 *b* 的表头，并加上新属性 *Z*。
- 结果的主体包含所有的元组 *t*，其中 *t* 是经过扩展的 *b* 的一个元组，该扩展使其增加了新属性 *Z* 上的一个值。*Z* 的新值是通过计算 *a* 元组上的合计而得到的，这些元组在 *A*1，*A*2，…，*An* 上与元组 *t* 有相同的值（当然，如果 *A* 没有元组和 *t* 在 *A*1，*A*2，…，*An* 上有相同的值，则合计会在一个空集合上操作）。

| P# | TOTQTY |
|---|---|
| P1 | 600 |
| P2 | 1000 |
| P3 | 400 |
| P4 | 500 |
| P5 | 500 |
| P6 | 100 |

图 7-11　关于 SUMMARIZE 的一个例子

关系 *b* 不能有名为 *Z* 的属性，并且合计不能涉及 *Z*。于是，结果和 *b* 有相同的基数，结果的度等于 *b* 的度加 1。结果中 *Z* 的类型和合计的类型相同。

下面是另一个例子：

```
SUMMARIZE ( P JOIN SP ) PER P { CITY } ADD COUNT AS NSP
```

其结果看起来像：

| CITY | NSP |
|---|---|
| London | 5 |
| Oslo | 1 |
| Paris | 6 |

也就是说，结果对应于每个城市（London、Paris 和 Rome）有一个元组，显示了每个城市中存放的零件的数量。

要点：

1）值得注意的是，这个操作符的定义是依赖于元组等价性的。

2）我们的语法允许多重 SUMMARIZE。例如：

```
SUMMARIZE SP PER P { P# } ADD ( SUM ( QTY ) AS TOTQTY,
                                AVG ( QTY ) AS AVGQTY )
```

3）< *summarize* > 的一般格式如下：

```
SUMMARIZE <relation exp> PER <relation exp>
                         ADD ( <summarize add commalist> )
```

其中每个 < *summarize add* > 形为：

```
<summary type> [ ( <scalar exp> ) ] AS <attribute name>
```

典型的 < *summary type* > 有 COUNT、SUM、AVG、MAX、MIN、ALL、ANY、COUNTD、SUMD 和 AVGD。在 COUNTD、SUMD 和 AVGD 中，“D”（distinct）的意思是“执行 summary 之前去掉多余的重复值”。< *scalar exp* > 可以涉及紧跟在 SUMMARIZE 后的 < *relation exp* > 所表示的关系的属性。注意：< *scalar exp* >（在圆括号中）仅当 < *summary type* > 是 COUNT 时才能被省略。

顺便提一下，请注意 < *summarize add* > 不同于 < *agg op inv* >。< *agg op inv* > 是一个标量表达式，只要是标量类型（特殊情况下为一个标量字串）可以出现的地方，它就能出现。相反，< *summarize add* > 只是一个 SUMMARIZE 的操作对象；却不是标量表达式。脱离开 SUMMARIZE 的上下文环境，它就失去意义，实际上根本不能出现。

4）读者可能已意识到，SUMMARIZE 不是一个基本操作符——它可通过 EXTEND 来表达。例如，表达式：

```
SUMMARIZE SP PER S { S# } ADD COUNT AS NP
```

是下面表达式的简写：

```
( EXTEND S { S# }
         ADD ( ( ( SP RENAME S# AS X ) WHERE X = S# ) AS Y,
               COUNT ( Y ) AS NP ) )
{ S#, NP }
```

或等价于：

```
WITH ( S { S# } ) AS T1,
     ( SP RENAME S# AS X ) AS T2,
     ( EXTEND T1 ADD ( T2 WHERE X = S# ) AS Y ) AS T3,
     ( EXTEND T3 ADD COUNT ( Y ) AS NP ) AS T4 :
T4 { S#, NP }
```

顺便提一下，属性 *Y* 在这里是关系值，详细的内容请参看 6.4 节。

5）下面是另外一个例子

```
SUMMARIZE S PER S { CITY } ADD AVG ( STATUS ) AS AVG_STATUS
```

在这里，PER 关系并不仅仅与将要合计的关系的某些投影具有相同类型，它实际上就是那样的一个投影。在这个例子中，我们允许如下的缩写：

```
SUMMARIZE S BY { CITY } ADD AVG ( STATUS ) AS AVG_STATUS
```

（我们用 < *attribute name commalist* > 替代 < *relation exp* >，其中命名的属性必须全是正在合计的关系的属性。）

6）考虑下面的例子：

```
SUMMARIZE SP PER SP { } ADD SUM ( QTY ) AS GRANDTOTAL
```

根据上面的意思，我们可以把这个表达式重写为：

```
SUMMARIZE SP BY { } ADD SUM ( QTY ) AS GRANDTOTAL
```

无论用哪一种方式，分组和合计所作用的关系根本就没有属性。假设 *sp* 是关系变量 SP 的当前值，并且 *sp* 至少有一个元组。于是对应于没有属性的情况，*sp* 的所有元组有相同的值（即 0 元组）。因此只有一组，并且在结果中只有一个元组（也就是说，聚集计算在整个关系 *sp* 上运行了一次）。SUMMARIZE 产生的关系只有一个属性和一个元组；属性叫作总计（GRANDTOTAL），结果元组中唯一的标量值是原关系 *sp* 中所有 QTY 值的和。

另一方面，如果关系 *sp* 根本就没有元组，那么就没有分成的组，也就没有结果元组（即结果关系是空的）。相反，接下来的表达式：

```
SUMMARIZE SP PER TABLE_DEE ADD SUM ( QTY ) AS GRANDTOTAL
```

即使 *sp* 是空的也会运行（即它会产生零这个“正确”结果）。更精确地讲，它会返回只有一个名叫总计的属性的关系，同时关系只有一个在属性总计上值为零的元组。因此我们建议应该从 SUMMARIZE 中删去 PER 子句，如下所示：

```
SUMMARIZE SP ADD SUM ( QTY ) AS GRANDTOTAL
```

省略 PER 子句等价于指定子句 PER TABLE_DEE。

### 5. 传递闭包

“Tclose”代表传递闭包（transitive closure）。这里提到它主要是为了完备性；本章不作深入的讨论。然而接下来我们至少应该给它下个定义。设 *a* 是包含属性 *X* 和 *Y* 的两目关系，其中 *X*、*Y* 具有相同的类型 *T*。于是 *a* 的**传递闭包** TCLOSE *a* 是一个关系 $a^+$，$a^+$ 的表头和 *a* 的相同，主体是 *a* 的主体的一个超集，定义如下：当且仅当元组 $\{X\ x, Y\ y\}$ 出现在 *a* 中，或存在值 $z1, z2, \cdots, zn$（类型都是 *T*）的一个序列，使元组 $\{X\ x, Y\ z1\}$、$\{X\ z1, Y\ z2\}$、……、$\{X\ zn, Yy\}$ 都出现在 *a* 中，元组 $\{X\ x,\ Y\ y\}$ 才属于 $a^+$（也就是说，只有当关系 *a* 所表示的图中存在从节点 *x* 到节点 *y* 的一条路径时，元组 $(x,\ y)$ 才会出现在 $a^+$ 中，这只是不严格的说法。注意，$a^+$ 的主体必须包含 *a* 的主体，把它作为一个子集）。

第 24 章会有关于传递闭包的更详细的讨论。

## 7.9　分组与解组

因为关系的属性值可以是关系值，这就需要增加关系型操作，使得含有这种属性的关系和不含有这种属性的关系之间可以互相转换，分别称之为**分组**（group）和**解组**（ungroup）。首先给出一个关于分组的例子：

```
SP GROUP { P#, QTY } AS PQ
```

给定那些常用的样本数据，此表达式产生的结果如图 7-12 所示。注意：下面的解释有些抽象（很遗憾，只能这样了），参照图例去看解释也许会有所帮助。

基本表达式

```
SP GROUP { P#, QTY } AS PQ
```

可以读作“group SP by S#”，因为 S#是唯一没有在 GROUP 子句中提到的属性。结果是定义如下的一个关系。首先，表头形式为：

```
{ S# S#, PQ RELATION { P# P#, QTY QTY } }
```

换言之，它包含了一个值为关系的属性 PQ（其中 PQ 含有属性 P#和 QTY），并且包含除 SP 外的所有其他属性（当然，除 SP 外的所有属性只有 S#）。其次，主体对于 SP 中每个不同的 S#值对应一个元组。主体中的每个元组包含一个 S#值（如 *s*）和一个 PQ 值（如 *pq*），其中 PQ 的值如下获得：

- 每个SP元组在概念上被一个元组（如$x$）代替，此元组中的P#和QTY分量被包装成一个元组值（tuple-value）的分量（如$y$）。
- 在所有属性S#值为$s$的元组$x$中，所对应的分量$y$组合成了一个关系（$pq$）。这样，一个结果元组就产生了，其中S#的值是$s$，PQ的值是$pq$。

其全部结果实际上已经在图7-12中显示了。尤其应当注意的是，结果中没有任何一个元组包含供应商S5（这是因为关系变量SP中也没有涉及供应商S5）。

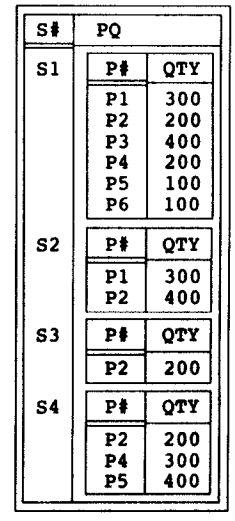


图7-12 按照S#对SP分组

可以看出，$R$ GROUP $\{A1,A2,\cdots,An\}$ AS $B$ 的结果的度等于$nR-n+1$，其中$nR$为$R$的度。

现在看一下解组。假设SPQ是图7-12所示的关系。于是表达式

```
SPQ UNGROUP PQ
```

（或许是意料之中）返回了常用的关系SP。更具体点，它产生了一个定义如下的关系。首先，表头形式为：

```
{ S# S#, P# P#, QTY QTY }
```

换言之，表头包含属性P#和QTY（从属性PQ派生而来），同时包含SPQ的所有其他属性（在本例中就是S#）。其次，SPQ中的一个元组和在此元组中PQ值的一个元组构成一个组合，对于每个组合，主体对应一个元组（只有一个元组）。主体的每个元组由S#的值（如$s$）和P#、QTY的值（如$p$和$q$）组成，通过以下方式得到：

- 一个“解组”的元组集合代替了一个SPQ元组，对应SPQ元组的PQ值中的每个元组产生这样一个元组（如$x$）。
- 元组$x$包含两部分，一部分是等于SPQ元组中S#分量的值（如$s$），另一部分等于SPQ元组中PQ分量的某个元组的值（如$y$）。
- 在属性S#值为$s$的元组$x$中，分量$y$被分解成P#和QTY（如$p$和$q$）。于是产生了这样的一个元组，它在属性S#上值为$s$，P#上值为$p$，QTY上值为$q$。

正如前面所说的那样，整个的结果就是关系SP。

可以看到，$R$ UNGROUP $B$（属性$B$的值为关系型，其表头为（$A1$，$A2$，…，$An$））的度等于$nR+n-1$，其中$nR$为$R$的度。

正如我们所看到的那样，分组和解组提供了关系的嵌套和非嵌套功能。我们更倾向于用术语分组/解组——因为术语嵌套/非嵌套和$NF^2$关系的概念有着紧密联系［见文献6.10］，而我们并不认同这个概念。

为了完整性，本节最后谈一下GROUP和UNGROUP操作的可逆性（初次阅读可能会有一定的困难）。如果通过任一种途径对关系$r$分组，总会有一个可逆的解组操作可使我们重新得到$r$。然而，如果按某种途径对某个关系$r$进行解组，可能就不存在相应的可逆分组了。下面举一个例子（基于文献［6.4］中的一个例子）。假设有关系TWO（见图7-13），对它进行解组操作得到关系THREE。如果现在对THREE按照A分组（并且把产生的关系型属性也命名为RVX），得到的将是ONE而不是TWO。

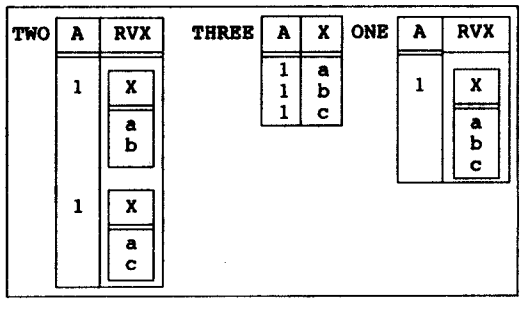


图7-13 分组还原和（重新）分组不一定是可逆的

如果我们对ONE进行解组，将会得到THREE；而且我们已经知道，对THREE进行分组可以得到ONE，因此，对这一对特殊的关系来说，分组和分组还原操作是可逆的。注

意，在 ONE 中，RVX 函数依赖于 A[⊖]（有必要说明的是，ONE 只含有一个元组）。事实上，我们可以概括地说，如果关系 *r* 有一个关系型的属性 RVX，则说 *r* 是分组还原可逆的，当且仅当满足下面两个条件：

- *r* 的任一元组在 RVX 上不是一个空关系。
- RVX 函数依赖于 *r* 的所有其他属性所组成的一组属性。

## 7.10　小结

我们已经讨论了**关系代数**。首先再次强调了**封闭性**和**嵌套关系表达式**的重要性，并解释了如果要严格保持封闭性，则要有一套**关系类型演绎规则**（relation type inference rule）。正是基于这样的考虑，让我们引入了**重命名**操作。

最初的代数包含 8 个操作符——传统的集合操作**并**、**交**、**差**和**积**（对它们都作了稍微的修改，以适应其操作对象是关系而不是任意的集合的特性），和专门的关系操作**选择**、**投影**、**连接**和**除**（对于除的情况，我们曾提到涉及除的查询通常可以用关系比较来描述，很多人发现这样的描述更加通俗易懂）。在此基础上又加了**重命名**、**半连接**、**半减**、**扩展**和**合计操作**，我们也提到了 **TCLOSE**，并讨论了**分组**和**解组**。尤其是扩展，极其重要（在某些方面它如同连接操作般重要）。

接下来，我们指出了这些操作符并不全都是基本的（它们中的几个可以通过其他的来表示）——但在我们看来，这些操作符已经非常完美了。正如文献［7.3］所提到的“定义一种语言时，最初先谨慎地选择几个基本的操作……在往后的发展中，若有必要，再定义新的操作……新的操作可以用旧的操作来描述”——也就是，定义有用的**速记**。如果选择得好，这样的速记不仅可以帮我们省去书写表达式的很多麻烦，而且通过它们和别的操作的联合使用，使得描述查询要求更加自然，从而**提升了抽象程度**。（它们还提供了更有效的实现方法。）与此相关，正如 7.6 节所引用的那样，我们要提醒读者，文献［3.3］介绍了一种“简化的指令集”代数，称为 **A**。它的目的是为了用非常少的描述能力更强的基本操作，来支持这种系统的定义。事实上，那 8 个基本操作符，还有重命名、扩展、合计、分组和解组等操作，可以仅用两个最基本的操作实现，那就是 *remove* 和 *nor*。

接下来本章介绍了怎样把代数操作和表达式结合起来实现一系列目的：**查询**、**更新**或诸如此类的。我们还简单讨论了如何**重写**这样的表达式来达到**优化**的目的（详细内容将在第 18 章讲解）。我们还介绍了如何使用 **WITH** 来简化复杂的查询；由于 WITH 允许为子表达式引进名称，因此我们便可以考虑用分步的方法处理复杂查询；但这样的方式并没有影响到代数的根本：非过程化。

本章中也指出了某些操作符具有**交换性**和**结合性**，还给出了一些**恒等式**（例如，关系 *R* 等价于 *R* 的某些选择和投影的结果）。我们还考虑了仅仅在一个关系，甚至空关系上，如何进行连接、并、交等操作。

## 习题

7.1　本章中哪些关系操作符的定义不要求元组具有相同的类型？

7.2　给定前面的供应商和零件数据库，表达式 S JOIN SP JOIN P 的值是什么？相应的谓词又是什么？注意：这里有一个陷阱！

7.3　假设 *r* 是一个度为 *n* 的关系。*r* 有多少个不同的投影？

7.4　本章中，我们声称并、交、积和（自然）连接都具有交换性和结合性。证明之。

7.5　考虑表达式 *a* JOIN *b*，如果 *a* 和 *b* 没有共同的属性，则 *a* JOIN *b*。等价于 *a* TIMES *b*。如果 *a* 和 *b* 有相同的表头，则上述表达式等价于 *a* INTERSECT *b*。证明前述判断。如果 *a* 的属性是 *b* 属性的一个子集，那 *a* JOIN *b* 又等价于什么呢？

7.6　在 Codd 最初定义的 8 个操作符中，并、差、积、选择和投影可以被认为是基本的。试用这 5 种基本

---

⊖ 请看第 11 章，同时需要特别注意的是，这里所说的函数依赖的形式是针对关系型的属性（而不是一般关系变量的属性）。

操作来表示自然连接、交和除。

7.7 算术里的乘和除是两个互逆的操作。关系代数中的 TIMES 和 DIVIDEBY 是互逆操作吗?

7.8 算术中特殊的数字 1 和 0 使对任意数字 *n* 都有如下性质

```
n * 1  =  1 * n  =  n, n * 0  =  0 * n  =  0
```

在关系代数中存在类似角色的关系吗?如果有,是什么?(考虑本章讨论过的所有操作符。)

7.9 在 7.2 节讲到,与算术中封闭性的重要性一样,关系封闭性是很重要的。然而算术里有一个封闭性被打破的情况——即被零除。关系代数中有类似的情况吗?

7.10 关系操作交可以看成是一种特殊的连接,但并不是对所有的关系,这两种操作都会产生相同的结果。为什么?

7.11 下列表达式哪些是等价的(如果有的话)?

a. `SUMMARIZE r PER r { } ADD COUNT AS CT`

b. `SUMMARIZE r ADD COUNT AS CT`

c. `SUMMARIZE r BY { } ADD COUNT AS CT`

d. `EXTEND TABLE_DEE ADD COUNT ( r ) AS CT`

7.12 假设 *r* 是由下面表达式得到的关系

```
SP GROUP { } AS X
```

那根据前面常用的供应商例子的样本值,*r* 的结果是什么?下列表达式的值又是什么?

```
r UNGROUP X
```

## 查询练习

下面的练习都以供应商 - 零件 - 工程数据库为基础,每个练习要求你为某个查询写出关系代数表达式(由于个人不同的爱好,你可能倾向于先看一些答案,然后用自然语言陈述出给定表达式的意思)。为了方便,下面列出数据库的结构:

```
S    { S#, SNAME, STATUS, CITY }
     PRIMARY KEY { S# }
P    { P#, PNAME, COLOR, WEIGHT, CITY }
     PRIMARY KEY { P# }
J    { J#, JNAME, CITY }
     PRIMARY KEY { J# }
SPJ  { S#, P#, J#, QTY }
     PRIMARY KEY { S#, P#, J# }
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P
     FOREIGN KEY { J# } REFERENCES J
```

7.13 求所有有关工程的信息。

7.14 求在伦敦的所有工程的信息。

7.15 求为工程 J1 提供零件的供应商的号码。

7.16 求数量在 300 ~ 750 之间的所有发货。

7.17 求所有的零件颜色/城市对。注意:这里及以后所说的“所有”特指在数据库中。

7.18 求所有的供应商号/零件号/工程号三元组。其中所指的供应商、零件和工程在同一个城市。

7.19 求所有的供应商号/零件号/工程号三元组。其中所指的供应商、零件和工程不在同一个城市。

7.20 求所有的供应商号/零件号/工程号三元组。其中所指的供应商、零件和工程三者中的任意两个都不在同一个城市。

7.21 求由伦敦的供应商提供的零件的信息。

7.22 求由伦敦的供应商为伦敦的工程供应的零件号。

7.23 求满足下面要求的城市对,要求在第一个城市的供应商为第二个城市的工程供应零件。

7.24 求供应商为工程供应的零件号,要求供应商和工程在同一城市。

7.25 求至少被一个不在同一城市的供应商供应零件的工程号。

7.26 求由同一个供应商供应的零件号的对。

7.27 求所有由供应商 S1 供应的工程号。

7.28　求供应商 S1 供应的零件 P1 的总量。
7.29　对每个供应给工程的零件，求零件号、工程号和相应的零件总量。
7.30　求为单个工程供应的零件数量超过 350 的零件号。
7.31　求由供应商 S1 供应的工程名称。
7.32　求由供应商 S1 供应的零件颜色。
7.33　求供应给伦敦的工程的零件号。
7.34　求使用了 S1 供应的零件的工程号。
7.35　设供应了红色零件的供应商为 S2，求供应了至少一个 S2 供应的零件的供应商号。
7.36　求 status 比 S1 低的供应商号。
7.37　求所在城市按字母排序为第一的工程号。
7.38　求被供应零件 P1 的平均数量大于供应给工程 J1 的任意零件的最大数量的工程号。
7.39　求满足下面要求的供应商号，该供应商给某个工程供应零件 P1 的数量大于供应给这个工程的零件 P1 的平均数量。
7.40　求没有被伦敦的供应商供应过红色零件的工程号。
7.41　求所用零件全被 S1 供应的工程号。
7.42　求所有伦敦的工程都使用的零件号。
7.43　求对所有工程都提供了同一零件的供应商号。
7.44　求使用了 S1 提供的所有零件的工程号。
7.45　求至少有一个供应商、零件或工程所在的城市。
7.46　求被伦敦供应商供应或被伦敦工程使用的零件号。
7.47　求所有供应商号/零件号对，其中指定的供应商不供应指定的零件。
7.48　求供应商号码对（如 Sx 和 Sy），其中 Sx 和 Sy 供应的零件都相同。注意：为简单起见，在本道题中你可以使用原始的供应商和零件数据库，而不是扩展的供应商 - 零件 - 工程数据库。
7.49　按供应商号/零件号对零件供应情况分组，相应的工程号和数量形成二元关系。
7.50　对上题所得的关系进行分组还原。

## 参考文献

[7.1] E. F. Codd: "Relational Completeness of Data Base Sublanguages," in Randall J. Rustin (ed.), *Data Base Systems*, *Courant Computer Science Symposia Series 6*. Englewood Cliffs, N. J.: PrenticeHall (1972).

这是 Codd 首次正式定义最初的代数操作符的文章（当然，[6.1] 中也有定义，但那既不正式也不全面）。注意：这其中有一个小缺陷，他声称为了表述和解释的方便，假设关系的属性有一个从左到右的次序，由它们的相对位置来确定（尽管 Codd 也说明了"实际中存取信息的时候，应该使用名称而不是位置号"——之前他也在文献 [6.1] 中多次提到相同的事情）。文章因此没有提到属性重命名的操作符（RENAME），也没有考虑到结果的类型。可能是因为忽略了这些，同样的批评和考证一直持续到今天，包括：(a) 在讨论代数的诸多文献中，(b) 在现在的 SQL 产品中，(c) 乃至有人要求对 SQL 标准进行轻微的扩展。

在第 8 章特别是 8.4 节里面将有关于这篇文章的更多评论。

[7.2] Hugh Darwen (writing as Andrew Warden): "Adventures in Relationland," in C. J. Date, *Relational Database Writings 1985 - 1989*. Reading, Mass.: Addison-Wesley (1990).

这是一系列内容新颖、有趣并具有知识性的短文，描述了关系模型和关系 DBMS。

[7.3] Hugh Darwen: "Valid Time and Transcation Time Proposals: Language Design Aspects," in Opher Etzion, Sushil Jajodia, and Suryanaryan Sripada (eds.), *Temporal Databases: Research and praCtice*. New York, N. Y.: Springer-Verlag (1998).

[7.4] Hugh Darwen and C. J. Date: "Into the Great Divide," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

这篇文章分析了 Codd 在 [7.1] 中定义的除，并分析了 Hall、Hitchcock 和 Todd [7.10] 关于除的概括——它不同于 Codd 的除，允许关系可以被任意的关系来除（在 Codd 定义的除中，除数关系的表头必须是被除数关系表头的子集）。这篇论文说明，两种除在处理空关系时都遇到了困难，所得的结果都不能处理想要解决的问题（即两者都不是想要的全称量词）。对它们的改进（即小除 small divide 和大除 great divide）克服了上述问题。注意：按照 **Tutorial D** 语法，两者是不同

的操作符，即大除不是小除的一个扩充。该论文同时显示了修改的操作符和除的名字有些名不符实！关于这一点，参考练习7.7。

为了方便引用，在这里给出Codd对除的定义。设关系 $A$ 和 $B$ 分别有表头 $\{X, Y\}$ 和 $\{Y\}$（其中 $X$ 和 $Y$ 可以是合成的）。于是 $A$ DIVIDEBY $B$ 产生了一个关系，关系的表头是 $\{X\}$，主体包含所有满足下述条件的元组 $\{X, x\}$，即对任意出现在 $B$ 中的元组 $\{Y, y\}$，$A$ 都有一个元组 $\{X, x; Y, y\}$ 存在。换一种不严格的说法，即若 $X$ 在 $A$ 中对应的 $Y$ 值包含 $B$ 中所有的 $Y$ 值，则 $X$ 出现在结果中。

[7.5] C. J. Date: "Quota Queries" (In three parts), in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994 - 1997*. Reading, Mass.: Addison-Wesley (1998).

限额查询就是指定结果个数的查询——例如，查询"求三个最重的零件"。这篇论文讨论了一种表达这种查询的方法。用这种方法，"求三个最重的零件"可以表示为：

```
P QUOTA ( 3, DESC WEIGHT )
```

这个表达式是下列语句的简写：

```
( ( EXTEND P
    ADD COUNT ( ( P RENAME WEIGHT AS WT ) WHERE WT > WEIGHT )
    AS #_HEAVIER )
WHERE #_HEAVIER < 3 ) { ALL BUT #_HEAVIER }
```

（其中WT和#_HEAVIER是任意的。给定我们常用的样本数据，则结果包含零件P2、P3和P6）。这篇论文深入地分析了限额查询的需求，并提出了用来处理和描述问题的几种简写。注意：文献[3.3]介绍了表达限额查询的另一种方式，引入了一个名为RANK的关系操作符。

[7.6] R. C. Goldstein and A. J. Strnad: "The MacAIMS Data Management System," Proc. 1970 ACM SICFIDET Workshop on Data Description and Access (November 1970).

参看[7.7]中的注解。

[7.7] A. J. Strnad: "The Relational Approach to the Management of Data Bases," Proc. IFIP Congress, Ljubljana, Yugoslavia (August 1971).

由于历史的缘故我们提到MacAIMS [7.6, 7.7]；它可能是最早支持 $n$ 元关系和代数语言的系统。有意思的是，它和Codd在关系模型上的工作是并行发展的，至少部分是无关的。与Codd的工作不同的是，MacAIMS没有吸引人们在这方面做进一步的研究。

[7.8] M. G. Notley: "The Peterlee IS/1 System," IBM UK Scientific Centre Report UKSC-0018 (March 1972).

参看[7.9]的注解。

[7.9] S. J. P. Todd: "The Peterlee Relational Test Vehicle—A System Overview," *IBM Sys. J.* 15, No. 4 (1976).

PRTV (The Peterlee Relational Test Vehicle) 是IBM位于英国Peterlee的研究中心开发的实验系统。它建立在较早的原型IS/1的基础之上——可能正是IS/1首次实现了Codd的思想。它支持 $n$ 元关系和叫做ISBL (Information System Base Language) 的一种代数语言，这种语言以[7.10]中列出的提案为基础。本章所讲的关于关系类型的思想可以追溯到ISBL和[7.10]的提案。PRTV中比较重要的方面如下：

- 支持RENAME、EXTEND和SUMMARIZE。
- 吸收了复杂表达式转换的技术（参看第18章）。
- 具有一种惰性计算特征（lazy evaluation feature），这对优化和视图支持是很重要的（参看本章关于WITH的讨论）。
- 提供可扩展功能——即赋予用户定义自己的操作符的权利。

[7.10] P. A. V. Hall, P. Hitchcock, and S. J. P. Todd: "An Algebra of Relations for Machine Computation," Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Calif. (January 1975).

[7.11] Anthony Klug: "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *JACM 29*, No. 3 (July 1982).

对最初的关系代数和关系演算做了扩展（见第8章），以支持聚集操作，并证明了两者是等价的。

# 第8章 关系演算

## 8.1 引言

关系演算可以代替关系代数。它们之间的主要区别是：关系代数提供了连接、并和投影等明确的集合操作符，并且这些集合操作符告诉系统如何从给定关系构造所要求的关系；而关系演算仅提供了一种描述（*notation*）来说明所要求的关系（这一关系是根据给定关系导出的）的定义。例如，查询提供零件 P2 的供应商的号码和所在城市。此查询的一个代数操作形式可以描述如下（在这里有意不使用第 7 章的语法形式）：

1）根据供应商号（S#）连接供货商表（supplier）和供货表（shipment）；

2）在上述连接结果中选择零件号为 P2 的元组；

3）将上述选择结果在供应商号（S#）和供应商所在城市（city）列上投影。

相对而言，一个演算形式可以简单地描述为：

查取供应商号（S#）和供应商所在城市（city），当且仅当在关系供货中存在这样的一个元组：它具有同样的供应商号（S#），且它的零件号（P#）取值 P2。

在后一种形式中，用户仅仅描述了所要求结果的定义，而把具体的连接、选择等操作留给了系统。于是，我们可以这样说（至少表面地看）：关系演算是描述性（descriptive）形式的，而关系代数是说明性（prescriptive）形式的。关系演算描述了问题是什么，而关系代数说明了解决问题的过程。或者，可以说，关系代数是过程化的（诚然，是高级的，但仍然是过程化的）；而关系演算是非过程化的。

然而，我们强调的上述区别仅仅是形式上的。实际上，关系代数和关系演算在逻辑上是等价的。即每一个代数表达式都有一个等价的演算表达式，每一个演算表达式都有一个等价的代数表达式，两者是一一对应的关系。所以两者的区别仅仅是形式上的：关系演算更接近自然语言，而关系代数更像程序语言。但是，需要重复的是，这种区别更多的是表面上的，而不是实际的。特别地，没有哪一种方法真正地比另外一种更加非过程化。我们将在 8.4 节详细讨论这一等价问题。

关系演算是基于**谓词**演算，它是数理逻辑的一个分支。使用谓词演算作为查询语言的基础的思想起源于 Kuhns［8.6］的一篇论文。关系演算的概念（即特别适合关系数据库的一个应用型的谓词演算）最早由 Codd 提出（参考［6.1］）；Codd 还在另外一篇论文［8.1］中提出了一种基于关系演算的语言，称作“数据子语言 ALPHA”。ALPHA 从没有实际实现过，但有一种叫作 QUEL［8.5，8.10～8.12］的语言主要是参照它研制的。QUEL 语言已经实现，并且一度是 SQL 的主要竞争对手。

**范围变量**是关系演算的一个基本特征。简单地说，范围变量就是限制其取值范围在一个指定关系内的变量。即其允许的取值是这个关系中的一个元组。所以，如果范围变量 *V* 限制在关系 *r* 上，则在任何时候，表达式“*V*”都表示关系 *r* 的某一元组 *t*。例如，查询“在 London 的供应商的供应商号”，用 QUEL 语言可以表达如下：

```
RANGE OF SX IS S ;
RETRIEVE ( SX.S# ) WHERE SX.CITY = "London" ;
```

在这里 SX 是范围变量，且它限制在关系变量 S 的当前关系上（RANGE 语句是对范围变量的定义）。RETRIEVE 语句可以这样解释：对于变量 SX 的每一个取值，要取出其 S#字段的值，当且仅当其 *CITY* 字段的值为 London。

由于对值为元组的范围变量的依赖性（并且也是为了同域演算区分开来——见下一段），所以，起初关系演算就是**元组**演算。元组演算将在 8.2 节详细叙述。*注意*：为了简单起见，在这本书中，一般用演算和关系演算，而不加“元组”或“域”这样的修饰词，来专指元组演算。

Lacroix 和 Pirotte 提出了另一种演算形式，即**域演算**（参考［8.7］）。这种演算的范围变量

的取值限制在域上而不是在关系上。(这个术语是不恰当的：如果这种演算形式由于上述原因而被称为域演算，那么元组演算就应该被称为关系演算了。) 在所有已提出的与域演算语言相关的文献中，可能 Query-By-Example 语言 (QBE，参考 [8.14]) 是最有名的 (尽管实际上 QBE 混合了某些元组演算成分)。现在 QBE 已经在商业上实现了。我们将在 8.7 节大致介绍一下域演算，并在 8.8 节简要讨论一下 QBE。

注意：由于篇幅原因，在演算讨论中，我们有意省略了一些第 7 章中讨论的主题，例如分组与解组。在演算中我们也忽略了关系更新操作。关于这些问题，参考文献 [3.3] 做了简单的讨论。

## 8.2 元组演算

与第 7 章中讨论关系代数的方法一样，在这里我们首先介绍语法，然后讨论语义。注意：本章介绍的语法虽与 **Tutorial D** (参考 [3.3] 的附录 A) 给出的演算语法不完全一致，但基本风格是一样的。我们将首先讨论语法，而在其余的小节中讨论语义。

### 1. 语法

注意：这一小节的许多语法规则是以比较零散的形式给出的，只有当了解了后面的语义解释后，才能理解其含义。不过这里把它们先罗列在此，便于后面讨论时引用。

这里先给出第 7 章介绍的关系表达式 < *relation exp* > 的语法：

```
<relation exp>
    ::=   RELATION { <tuple exp commalist> }
        | <relvar name>
        | <relation op inv>
        | <with exp>
        | <introduced name>
        | ( <relation exp> )
```

换句话说，这里的关系表达式和以前是一样的，但是，其中最重要的，同时也是这一章唯一详细讨论的部分，关系操作 < *relation op inv* > 却有着不同的解释 (下面将会看到)。

```
<range var def>
    ::=   RANGEVAR <range var name>
          RANGES OVER <relation exp commalist> ;
```

在一定的上下文中，范围变量名 < *range var name* > 可以是元组表达式 < *tuple exp* > ○。这种上下文是：

- 范围属性引用 < *range attribute ref* > 中量词的点号之前；
- 紧接量化布尔表达式 < *quantified bool exp* > 的量词；
- 作为布尔表达式 < *bool exp* > 的一个操作数；
- 作为一个 < *proto tuple* > 或 < *proto tuple* > 中的表达式或其中的操作数。

```
<range attribute ref>
    ::=   <range var name> . <attribute name>
                        [ AS <attribute name> ]
```

在一定的上下文中，< *range attribute ref* > 可以用作一个表达式 < *exp* > 。这种上下文是：

- 作为布尔表达式 < *bool exp* > 中的一个操作数；
- 作为一个 < *proto tuple* > 或 < *proto tuple* > 中的表达式或其中的操作数。

```
<bool exp>
    ::=   ... all the usual possibilities, together with:
        | <quantified bool exp>
```

只有当下面两个条件都成立时，布尔表达式 < *bool exp* > 中范围变量的引用才不受其约束。

---

○ 通过上面的例子大家已经能够大体了解 < *tuple exp* > ，这里就不再做详细的介绍了。由于一些客观原因，我们在这里没有使用和前几章相同的语法。

- 关系操作 < *relation op inv* > 中直接出现的布尔表达式 < *bool exp* > （即布尔表达式直接跟在关键字 WHERE 后面）；
- < proto tuple > 中直接出现的同一个范围变量的引用（必须是自由的）直接包含在同一个关系操作中（即 < proto tuple > 出现在关键字 WHERE 之前）。

术语解释：在关系演算（包括元组演算和域演算）这一部分里，布尔表达式通常叫做**合式公式**（well-formed formulas）或简写为 WFF。在大多数如下情况中，我们使用这一术语：

```
<quantified bool exp>
    ::=   <quantifier> <range var name> ( <bool exp> )

<quantifier>
    ::=   EXISTS | FORALL

<relation op inv>
    ::=   <proto tuple> [ WHERE <bool exp> ]
```

在第 7 章的关系代数中我们讲到，关系操作 < *relation op inv* > 是关系表达式 < *relation exp* > 的一种形式。但这里我们给出一个不同的定义。

```
<proto tuple>
    ::=    ... see the body of the text
```

所有在原型元组 < *proto tuple* > 中直接出现的范围变量的引用都必须独立于该原型元组。注意："proto tuple" 表示 "prototype tuple"。在这里这个术语是合适的，但不标准。

**2. 范围变量**

下面给出一些范围变量的例子（用供应商表和零件表为例）：

```
RANGEVAR SX  RANGES OVER S ;
RANGEVAR SY  RANGES OVER S ;
RANGEVAR SPX RANGES OVER SP ;
RANGEVAR SPY RANGES OVER SP ;
RANGEVAR PX  RANGES OVER P ;

RANGEVAR SU RANGES OVER
       ( SX WHERE SX.CITY = 'London' ) ,
       ( SX WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                               SPX.P# = P# ('P1') ) ) ;
```

上例中的范围变量 SU 定义在一组元组集合的并上，这个集合并是：所有既住在伦敦又提供零件 P1 的供应商的元组。注意，范围变量 SU 的定义利用了范围变量 SX 和 SPX；因此根据并的定义，所有参加 "并" 的关系，其类型必须是一样的。

注意：从通常的程序语言的角度讲，范围变量并不是变量，这里讲的变量是从逻辑意义上讲的。实际上，与第 3 章讨论的参数（parameters）有点类似。不同的是：第 3 章讲的参数取值为域值，而元组演算中的元组变量取值为元组。

在本章以后的讨论中，我们将采用上述范围变量的定义。我们注意到，在实际的语言中，应该有一些规则去限定这种定义的范围。但在本章中我们忽略了此问题（除了 SQL 的部分）。

**3. 自由变量引用和约束变量引用**

在一些上下文中，尤其在合式公式中，每一个变量的引用要么是**自由**的，要么是**约束**的。首先我们从纯语法的角度解释这个概念，接着继续讨论它的重要性。

设 $V$ 为范围变量，$p$ 和 $q$ 为合式公式，则：

- 在合式公式 "NOT $p$" 中的 $V$ 引用是否受此合式公式的约束，要看它们是否受 $p$ 的约束。在合式公式（$p$ AND $q$）和（$p$ OR $q$）中的 $V$ 引用是否受此合式公式的约束，要看它们是否受 $p$ 或 $q$ 的约束。
- 如果 $V$ 引用在合式公式 "$p$" 中是自由的，则它在合式公式 "EXISTS $V(p)$" 和 "FORALL $V(p)$" 中一定是受约束的。其他在 "$p$" 中的范围变量的引用是否受合式公式 "EXISTS $V(p)$" 和 "FORALL $V(p)$" 的约束，要看它们是否受 "$p$" 的约束。

为了完整性，我们再增加下面几条：

- 在 < *range var name* > *V* 中的 *V* 单独引用（sole reference）是不受此 < *range var name* > 的约束的。
- 在 < *range attribute ref* > *V. A* 中的 *V* 单独引用是不受此 < *range attribute ref* > 的约束的。
- 如果在某一表达式 *exp* 中 *V* 引用是自由的，那么在其他任何包含 *exp* 的表达式 *exp'* 中，此 *V* 引用也是自由的，除非表达式 *exp'* 中有某一量词限制了此引用。

下面是包含范围变量的合式公式的一些例子：

- 简单比较：

```
SX.S# = S# ('S1')
SX.S# = SPX.S#
SPX.P# ≠ PX.P#
```

在此例中，所有对 SX、PX 和 SPX 的引用都是自由的。

- 简单比较的布尔联合：

```
PX.WEIGHT < WEIGHT ( 15.5 ) AND PX.CITY = 'Oslo'
NOT ( SX.CITY = 'London' )
SX.S# = SPX.S# AND SPX.P# ≠ PX.P#
PX.COLOR = COLOR ('Red') OR PX.CITY = 'London'
```

在此例中，所有对 SX、PX 和 SPX 的引用也都是自由的。

- 量化合式公式：

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
FORALL PX ( PX.COLOR = COLOR ('Red') )
```

在这两个例子中，SPX 和 PX 引用是约束的，SX 的引用是自由的。我们接下来讨论量词。

**4. 量词**

量词（quantifier）有两个，即 EXISTS 和 FORALL。EXISTS 是**存在量词**，而 FORALL 是**全称量词**[⊖]。基本上讲，如果 *p* 是合式公式，且此公式中 *V* 是自由的，则：

```
EXISTS V ( p )
```

和

```
FORALL V ( p )
```

都是合法的合式公式，且在两者中 *V* 都是受约束的。第一句的意思是：至少存在一个 *V* 值，使得 *p* 为真。第二句的意思是：对所有的 *V* 值，*p* 总是为真。例如，假设变量 *V* 限制在“2003 年美国参议院议员”这个集合体中，并假设 *p* 是合式公式“*V* 都为女性”（当然，在这里，我们不试图使用形式化语法！），则“EXISTS *V*(*p*)”和“FORALL *V*(*p*)”都是合法的合式公式，并且它们的取值分别为真和假。

再看上一节结束时 EXISTS 量词的例子：

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

根据以上所述，我们可以这样理解此合式公式：

*在关系变量 SP 的当前值里存在一个 SPX，其 S#的字段值等于任一 SX. S#字段值，且 P#取值为 P2。*

在这里 SPX 的引用是约束的，只有 SX 的引用是自由的。

---

⊖ 术语量词源于动词量化（quantify），大致意思是“有多少”。符号 ∃（左右颠倒的 E）和符号 ∀（上下颠倒的 A）分别用来代替存在和全部。

我们可以将 EXISTS 量词理解为**多个 OR 的重复**。换句话说，如果（a）$r$ 是一个关系，且其中有元组 $t1, t2, \cdots, tm$；（b）$V$ 是一个定义在 $r$ 上的范围变量；（c）$p(V)$ 是一个合式公式，且在这里 $V$ 是自由变量，则合式公式：

```
EXISTS V ( p ( V ) )
```

就等于：

```
FALSE OR p ( t1 ) OR ... OR p ( tm )
```

特别地，如果 $r$ 是空的（即 $m$ 为零），则此表达式取值为假。

下面我们通过例子来说明。首先假设关系 $r$ 包含下面的元组（这里为了简洁，没有使用通常的语法）：

```
( 1, 2, 3 )
( 1, 2, 4 )
( 1, 3, 4 )
```

假设这三个属性按从左到右的顺序分别记为 $A$、$B$ 和 $C$，每一属性都取整型。这样，下面的合式公式就有所示的取值：

```
EXISTS V ( V.C > 1 )              :  TRUE
EXISTS V ( V.B > 3 )              :  FALSE
EXISTS V ( V.A > 1 OR V.C = 4 )   :  TRUE
```

现在，我们开始讨论 FORALL 量词。还是从上一小节结束时的例子开始：

```
FORALL PX ( PX.COLOR = COLOR ('Red') )
```

我们可以如下理解此合式公式：

*在关系变量 P 的当前值中，对所有的元组 PX，其 COLOR 字段的取值为 RED。*

在这里两次 PX 引用都是受约束的。

正像我们把 EXISTS 量词理解为 OR 的重复使用一样，我们把 FORALL 量词理解为 **AND 的重复**。换句话说，如果 $r$、$V$ 和 $p(V)$ 都和上述 EXISTS 量词中讨论的一样，则合式公式：

```
FORALL V ( p ( V ) )
```

可以理解为：

```
TRUE AND p ( t1 ) AND ... AND p ( tm )
```

特别地，如果 $r$ 是空的（即 $m$ 为零），则此表达式取值为真。

下面我们通过例子来说明。首先假设关系 $r$ 包含和上面同样的元组，则下面的合式公式就有所示的取值：

```
FORALL V ( V.A > 1 )              :  FALSE
FORALL V ( V.B > 1 )              :  TRUE
FORALL V ( V.A = 1 AND V.C > 2 )  :  TRUE
```

*注意*：我们支持两种量词仅仅是为了方便，实际上这不是逻辑上必须的，因为其中的任一个量词都可以由另一个表示出来。为了更加明确，请看下面的式子

```
FORALL V ( p )  ≡  NOT EXISTS V ( NOT p )
```

（不严格地说，就是“所有使得 $p$ 为真的 $V$”和“不存在这样的 $V$ 使得 $p$ 为假”这两种说法是一样的）。此式表明：任何一个包含 FORALL 的合式公式都可以被一个等价的包含 EXISTS 的合式公式代替。例如，下面的语句（值为真）“对所有的整数 $x$，存在一个整数 $y$，使得 $y > x$”（即每一个整数都有一个比它大的整数）等价于“不存在一个整数 $x$，使得不存在一个整数 $y$，使得 $y > x$”（即不存在一个最大整数）。但是，一些问题适合于用 FORALL 量词描述，而另外一些则用 EXISTS 量词描述更加方便。进一步地，如果其中的一种量词不能使用，我们有时将不得不使

用双重否定（正如前面的例子所示），而双重否定处理起来比较困难，因此在实际引用中，这两个量词都要求支持。

**5. 自由变量引用和约束变量引用补充**

假设 *x* 的取值限制在整数集上，考虑下面的合式公式：

```
EXISTS x ( x > 3 )
```

注意，这里 *x* 是哑元（dummy），它仅仅起到连接括号里面的布尔表达式和外面的量词的作用。这个合式公式只是说明：存在某个整数 *x*，它比 3 大。因此，如果所有 *x* 的引用被其他的一些变量 *y* 的引用所代替，则此合式公式的意义不会发生变化。即合式公式

```
EXISTS y ( y > 3 )
```

在语义上和上式是一样的。

再看下面的合式公式：

```
EXISTS x ( x > 3 ) AND x < 0
```

此式对 *x* 进行了三次引用，表示两个不同的变量。前两次引用是受约束的，它可以被其他的引用所代替而不改变整句的意思。第三个引用是自由的，它不能随便地被替换。所以对下面的合式公式，第一个和上面的等价，第二个就不是：

```
EXISTS y ( y > 3 ) AND x < 0
EXISTS y ( y > 3 ) AND y < 0
```

而且，还要注意，如果不知道自由变量引用 *x* 表示的合式公式的值，就不能确定原合式公式的值。反过来，如果一合式公式的所有变量引用都是受约束的，也不能确定此合式公式的值一定为真或假。这里有两个术语：一个是**封闭式合式公式**（closed WFF），就是说，在此公式内的所有变量引用都是受约束的（实际上这是一个命题）；另一个是**开放式合式公式**（open WFF），即其中至少包含一个自由变量引用。如果用第 3 章讲到的术语来解释，封闭式合式公式是一个命题，而开放式合式公式是一个谓词，但它不是命题。（请注意命题是退化的特殊谓词，它的参数集为空。）

**6. 关系操作**

关系操作这个术语在演算里使用也许不太合适，关系定义可能更加合适。我们这样用是为了和第 7 章保持一致性。参看下面的语法：

```
<relation op inv>
    ::=    <proto tuple> [ WHERE <bool exp> ]
<proto tuple>
    ::=    ... see the body of the text
```

回忆一下前面的语法规则，在此我们稍微做了一点修改：

- 所有在原型元组中引用的范围变量都必须不受此原型元组的约束。
- WHERE 子句中的范围变量的引用可能是自由的，仅当在相应的原型元组中同一范围变量的引用存在，并且是自由的。

例如，下述操作是一个合法的关系操作（“寻找位于伦敦的供应商的供应商号”）：

```
SX.S# WHERE SX.CITY = 'London'
```

在此原型元组中，SX 的引用是自由的；在 WHERE 子句中，SX 的引用也是自由的。这种应用是合法的。因为在这一原型元组中出现的是同一范围变量，且它们是自由的。

再看下面的例子（“寻找供应零件 P2 的供应商的供应商号”——参看本小节前面关于 EXISTS 量词的讨论）：

```
SX.SNAME WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                            SPX.P# = P# ('P2') )
```

这里 SX 的引用都是自由的；WHERE 子句中的 SPX 引用都是受约束的，因为在这个原型元组中没有同一范围变量的引用。

直观上看，一个给定的关系操作等价于包含每一个原型元组可能的值的关系。并且对这些原型元组来说，在 WHERE 子句中指定的布尔表达式取值为真（若省略了 WHERE 子句，就默认为 WHERE 子句取值为真）。为了更加明确，下面进一步解释：

- 首先，原型元组是一个由括号括起来的集合，集合中的元素用逗号分隔（如果集合中只有一个元素，则括号可以省略）。在此集合中，每一项要么是一个范围属性引用（可能包含一个 AS 子句来引入一个新的属性名），要么是一个简单的范围变量名。（还存在其他的可能，但在进一步讨论这两种情况之前，其他可能我们先不予考虑。）但是：

  a）这里的范围变量名基本上是范围属性引用列表的简记，而范围属性就是此范围变量被限制的关系的每一个属性；

  b）没有 AS 子句的范围属性的引用基本上仅是一个简记，即其中每一个新属性名与原属性名是一样的。

  因此不失一般性，我们把原型元组看作范围属性引用的列表，如 *Vi*.*Aj* AS *Bj*。注意，*Vi* 和 *Aj* 不必都是明确的，但 *Bj* 一定要明确。
- 设原型元组中的范围变量定为 $V1,V2,\cdots,Vm$；设这些范围变量定义在其上的这些关系分别为 $r1,r2,\cdots,rm$；在应用 AS 子句中的属性重命名之后，设相应的关系为 $r1',r2',\cdots,rm'$；设 $r'$ 是 $r1',r2',\cdots,rm'$ 的笛卡尔积。
- 设 $r$ 是 $r'$ 中使得 WHERE 子句中合式公式的取值为真的子集。注意：为了这些说明，我们还要假设前面步骤中 WHERE 子句中的重命名是应用在属性上，否则 WHERE 子句中的合式公式将没有意义。然而，实际上，具体语法并不依赖于这一假设，而是依赖于圆点符去消除必要的岐义。下一小节我们还将作说明。
- 所有关系操作的值都是定义在所有的 *Bj* 上的关系 *r* 的投影。

具体请看下一节的例子。

## 8.3　举例

下面，我们用明确的查询给出一些有关演算的例子。作为练习，为了对比起见，你可以试着给出代数形式。有些例子在第 7 章中曾经出现过，这里做了标注。

**例 8.3.1　找出位于 Paris 且其状态大于 20 的供应商的供应商号及状态**

```
{ SX.S#, SX.STATUS }
WHERE SX.CITY = 'Paris' AND SX.STATUS > 20
```

**例 8.3.2　找出所有成对的住在同一城市的供应商的供应商号（例 7.5.5）**

```
{ SX.S# AS SA, SY.S# AS SB }
              WHERE SX.CITY = SY.CITY AND SX.S# < SY.S#
```

注意：原型元组中给出了最终结果的属性名；这些名字在 WHERE 子句中是不能使用的，这就是为什么 WHERE 子句的第二个比较用“SX.S# < SY.S#”而不是“SA < SB”的形式。

**例 8.3.3　找出供应零件 P2 的所有供应商的信息（修改过的例 7.5.1）**

```
SX WHERE EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

注意：此原型元组中范围变量名的使用。这个例子可以写成以下形式：

```
{ SX.S#, SX.SNAME, SX.STATUS, SX.CITY }
   WHERE EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

**例8.3.4 找出至少供应一个红色零件的供应商名（例7.5.2）**

```
SX.SNAME
WHERE EXISTS SPX ( SX.S# = SPX.S# AND
                   EXISTS PX ( PX.P# = SPX.P# AND
                               PX.COLOR = COLOR ('Red') ) )
```

或者用下述前束范式表示。此式中所有量词都提到合式公式之前。

```
SX.SNAME
WHERE EXISTS SPX ( EXISTS PX ( SX.S# = SPX.S# AND
                               SPX.P# = PX.P# AND
                               PX.COLOR = COLOR ('Red') ) )
```

前束范式并不是天然地比其他形式更正确或没有其他形式正确，但是在许多情况下，这种范式比较自然明确。而且，它的使用有可能使得括号的数量减少。例如，看下面的合式公式：

```
Q1 V1 ( Q2 V2 ( wff ) )
```

（这里，每一个Q1和Q2或者是EXISTS或者是FORALL）。此式可能很随意，但很明确。此式可以缩写为：

```
Q1 V1 Q2 V2 ( wff )
```

这样我们可以简化上面的演算表达式，如下：

```
SX.SNAME
WHERE EXISTS SPX EXISTS PX ( SX.S# = SPX.S# AND
                             SPX.P# = PX.P# AND
                             PX.COLOR = COLOR ('Red') )
```

然而，为了简明起见，我们将继续使用所有的括号。

**例8.3.5 找出至少供应S2供应的零件之一的供应商名**

```
SX.SNAME
WHERE EXISTS SPX ( EXISTS SPY ( SX.S# = SPX.S# AND
                                SPX.P# = SPY.P# AND
                                SPY.S# = S# ('S2') ) )
```

**例8.3.6 找出供应所有零件的供应商名（例7.5.3）**

```
SX.SNAME WHERE FORALL PX ( EXISTS SPX ( SPX.S# = SX.S# AND
                                       SPX.P# = PX.P# ) )
```

不使用FORALL量词，可以等价地表示为：

```
SX.SNAME WHERE NOT EXISTS PX ( NOT EXISTS SPX
                                    ( SPX.S# = SX.S# AND
                                      SPX.P# = PX.P# ) )
```

**例8.3.7 找出不供应零件P2的供应商名（例7.5.6）**

```
SX.SNAME WHERE NOT EXISTS SPX
                ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

注意：此结果很容易从例8.3.3的结果导出来。

**例8.3.8 找出至少由供应商S2所供应的零件的供应商号（例7.5.4）**

```
SX.S# WHERE FORALL SPX ( SPX.S# ≠ S# ('S2') OR
                         EXISTS SPY ( SPY.S# = SX.S# AND
                                      SPY.P# = SPX.P# ) )
```

此句可解释为："取供应商SX的号码，对于每一个SPX，下列条件为真：要么此供货不是来自于供应商S2；要么存在一个供货SPY，使得SPY成为SX供应的SPX中的一员。"对这种复

杂的查询，我们引入了另一个简单明了的语法简写形式，即**逻辑蕴含**。如果 *p* 和 *q* 是合式公式，则逻辑蕴含表达式：

```
IF p THEN q END IF
```

也是一个合式公式，在语义上和下式相同：

```
( NOT p ) OR q
```

此例还可以用下列形式表达：

```
SX.S# WHERE FORALL SPX ( IF SPX.S# = S# ('S2') THEN
                            EXISTS SPY ( SPY.S# = SX.S# AND
                                         SPY.P# = SPX.P# )
                         END IF )
```

此句可以这样理解："取供应商 SX 的供应商号，对于每一个 SPX，下列条件为真：如果供货 SPX 由供应商 S2 供应，则存在一个供货 SPY，使得 SPY 成为 SX 供应的 SPX 中的一员。"

**例 8.3.9　找出重量超过 16 磅或由供应商 S2 供应的零件**

```
RANGEVAR PU RANGES OVER
       ( PX.P# WHERE PX.WEIGHT > WEIGHT ( 16.0 ) ),
       ( SPX.P# WHERE SPX.S# = S# ('S2') ) ;
PU.P#
```

这与关系代数相似，包含了一个显式的并。

下面给出此查询的一个可替换形式，但这第二个式子要依赖于关系变量 P 的零件号包含了关系变量 SP 的零件号，这一点是并的形式不可缺少的。

```
PX.P# WHERE PX.WEIGHT > WEIGHT ( 16.0 )
      OR    EXISTS SPX ( SPX.P# = PX.P# AND
                         SPX.S# = S# ('S2') )
```

## 8.4　关系演算与关系代数的比较

在引言里我们就谈到，关系演算与关系代数是等价的，现在我们更加具体地讨论这一点。首先，Codd 认为，至少关系代数跟关系演算一样，具有强大的表达能力（参看 [7.1]）。他给出了一个算法证明这一点，这一算法叫"Codd 简约算法"。通过这个算法，任意一个演算表达式都可以简约为在语法上等价的代数表达式。这里不具体提供 Codd 的算法，但是我们用较大众化的术语举一个适当复杂的例子，来解释这个算法如何实现[⊖]。

现在来举一个例子。不使用我们熟悉的供应商与零件数据库，而使用扩展的供应商 - 零件 - 工程数据库（参看第 4 章及其他一些地方）。为方便起见，我们用图 8-1 给出一些示例（参看第 4 章图 4-5）。

现在来看查询：查找在城市 Athens 至少供应一个工程且每种零件至少供应 50 个的供应商名及其所在城市。这个查询的一个演算表达式是：

```
{ SX.SNAME, SX.CITY } WHERE EXISTS JX FORALL PX EXISTS SPJX
                                ( JX.CITY = 'Athens' AND
                                  JX.J# = SPJX.J# AND
                                  PX.P# = SPJX.P# AND
                                  SX.S# = SPJX.S# AND
                                  SPJX.QTY ≥ QTY ( 50 ) )
```

其中 SX、PX、JX 和 SPJX 分别是定义在 S、P、J 和 SPJ 上的范围变量。下面我们来看此查询如

---

⊖ 实际上，文献 [7.1] 提到的运算法则有一个小的缺陷 [8.2]。此外，该篇论文中定义的关系演算并没有包含完整的并操作。因此，严格地说，Codd 的关系演算并没有他的关系代数功能强大。但是，关系代数与包含了完整并操作的关系演算仍然是等价的，很多学者已经证明了这一点，比如 Klug [7.11]。

何进行计算。

步骤1：对每一个范围变量，只要可能，就对其进行选择而确定其范围（即确定所有可能的值）。这样，就意味着在WHERE子句里嵌入了限制性条件（参考第7章的定义），使得WHERE子句一执行，就消减了一定的元组。本例中，确定后的元组是：

SX：S中所有元组　　5个元组
PX：P中所有元组　　6个元组
JX：J中CITY = 'Athens'的元组　　2个元组
SPJX：SPJ中QTY≥QTY(50)的元组　　24个元组

S

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

P

| P# | PNAME | COLOR | WEIGHT | CITY |
|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London |
| P2 | Bolt | Green | 17.0 | Paris |
| P3 | Screw | Blue | 17.0 | Oslo |
| P4 | Screw | Red | 14.0 | London |
| P5 | Cam | Blue | 12.0 | Paris |
| P6 | Cog | Red | 19.0 | London |

J

| J# | JNAME | CITY |
|---|---|---|
| J1 | Sorter | Paris |
| J2 | Display | Rome |
| J3 | OCR | Athens |
| J4 | Console | Athens |
| J5 | RAID | London |
| J6 | EDS | Oslo |
| J7 | Tape | London |

SPJ

| S# | P# | J# | QTY |
|---|---|---|---|
| S1 | P1 | J1 | 200 |
| S1 | P1 | J4 | 700 |
| S2 | P3 | J1 | 400 |
| S2 | P3 | J2 | 200 |
| S2 | P3 | J3 | 200 |
| S2 | P3 | J4 | 500 |
| S2 | P3 | J5 | 600 |
| S2 | P3 | J6 | 400 |
| S2 | P3 | J7 | 800 |
| S2 | P5 | J2 | 100 |
| S3 | P3 | J1 | 200 |
| S3 | P4 | J2 | 500 |
| S4 | P6 | J3 | 300 |
| S4 | P6 | J7 | 300 |
| S5 | P2 | J2 | 200 |
| S5 | P2 | J4 | 100 |
| S5 | P5 | J5 | 500 |
| S5 | P5 | J7 | 100 |
| S5 | P6 | J2 | 200 |
| S5 | P1 | J4 | 100 |
| S5 | P3 | J4 | 200 |
| S5 | P4 | J4 | 800 |
| S5 | P5 | J4 | 400 |
| S5 | P6 | J4 | 500 |

图8-1 供应商－零件－工程数据库（含样本数据）

步骤2：根据步骤1的结果，构造笛卡尔积，得下表：

| S# | SN | STATUS | CITY | P# | PN | COLOR | WEIGHT | CITY | J# | JN | CITY | S# | P# | J# | QTY |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S1 | Sm | 20 | Lon | P1 | Nt | Red | 12.0 | Lon | J3 | OR | Ath | S1 | P1 | J1 | 200 |
| S1 | Sm | 20 | Lon | P1 | Nt | Red | 12.0 | Lon | J3 | OR | Ath | S1 | P1 | J4 | 700 |
| .. | .. | .. | ... | .. | .. | ... | .. | ... | .. | .. | ... | .. | .. | .. | ... |
| .. | .. | .. | ... | .. | .. | ... | .. | ... | .. | .. | ... | .. | .. | .. | ... |
| .. | .. | .. | ... | .. | .. | ... | .. | ... | .. | .. | ... | .. | .. | .. | ... |

（等等）。完全的笛卡尔积的结果是5×6×2×24=1440个元组。注意：由于空间有限，上表只列出了一部分。我们没有给上述属性重命名，其实为了避免模糊性，应该这样做。但是上表中用不同的位置表明同名的属性来自哪些表。这种非正式的运用仅仅是为了简化说明。

步骤3：根据WHERE子句的连接部分，来选择步骤2中构造的笛卡尔积。此例中，这一连接部分是：

```
JX.J# = SPJX.J# AND PX.P# = SPJX.P# AND SX.S# = SPJX.S#
```

因此，对于上述笛卡尔积中那些供应商S#值不等于供货S#值的元组，那些零件P#值不等于供货P#值的元组，那些工程J#的值不等于供货J#的值的元组，统统删掉。这样，我们得到上述笛卡尔积的一个子集，是10个元组，如下所示：

| S# | SN | STATUS | CITY | P# | PN | COLOR | WEIGHT | CITY | J# | JN | CITY | S# | P# | J# | QTY |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S1 | Sm | 20 | Lon | P1 | Nt | Red | 12.0 | Lon | J4 | Cn | Ath | S1 | P1 | J4 | 700 |
| S2 | Jo | 10 | Par | P3 | Sc | Blue | 17.0 | Osl | J3 | OR | Ath | S2 | P3 | J3 | 200 |
| S2 | Jo | 10 | Par | P3 | Sc | Blue | 17.0 | Osl | J4 | Cn | Ath | S2 | P3 | J4 | 200 |
| S4 | Cl | 20 | Lon | P6 | Cg | Red | 19.0 | Lon | J3 | OR | Ath | S4 | P6 | J3 | 300 |
| S5 | Ad | 30 | Ath | P2 | Bt | Green | 17.0 | Par | J4 | Cn | Ath | S5 | P2 | J4 | 100 |
| S5 | Ad | 30 | Ath | P1 | Nt | Red | 12.0 | Lon | J4 | Cn | Ath | S5 | P1 | J4 | 100 |
| S5 | Ad | 30 | Ath | P3 | Sc | Blue | 17.0 | Osl | J4 | Cn | Ath | S5 | P3 | J4 | 200 |
| S5 | Ad | 30 | Ath | P4 | Sc | Red | 14.0 | Lon | J4 | Cn | Ath | S5 | P4 | J4 | 800 |
| S5 | Ad | 30 | Ath | P5 | Cm | Blue | 12.0 | Par | J4 | Cn | Ath | S5 | P5 | J4 | 400 |
| S5 | Ad | 30 | Ath | P6 | Cg | Red | 19.0 | Lon | J4 | Cn | Ath | S5 | P6 | J4 | 500 |

（此表为等值连接结果）。

步骤 4：从右到左，运用量词，如下所示：

- 对于量词 EXISTS $V$（这里 $V$ 是限制在关系 $r$ 上的范围变量），在当前中间结果上进行投影而删除关系 $r$ 上的所有属性。
- 对于量词 FORALL $V$，用步骤 1 中确定的跟 $V$ 有关的、限制了范围的关系去除以当前中间结果。这个操作也会删除关系 $r$ 上的所有属性。注意：这里讲的除，就是 Codd 讲的除操作（参见 [7.4]）。

此例中，这些量词是：

```
EXISTS JX FORALL PX EXISTS SPJX
```

因此：

- （EXISTS SPJX）通过投影消去 SPJ 的属性，即 SPJ. S#、SPJ. P#、SPJ. J#和 SPJ. QTY，结果是：

| S# | SN | STATUS | CITY | P# | PN | COLOR | WEIGHT | CITY | J# | JN | CITY |
|---|---|---|---|---|---|---|---|---|---|---|---|
| S1 | Sm | 20 | Lon | P1 | Nt | Red | 12.0 | Lon | J4 | Cn | Ath |
| S2 | Jo | 10 | Par | P3 | Sc | Blue | 17.0 | Osl | J3 | OR | Ath |
| S2 | Jo | 10 | Par | P3 | Sc | Blue | 17.0 | Osl | J4 | Cn | Ath |
| S4 | Cl | 20 | Lon | P6 | Cg | Red | 19.0 | Lon | J3 | OR | Ath |
| S5 | Ad | 30 | Ath | P2 | Bt | Green | 17.0 | Par | J4 | Cn | Ath |
| S5 | Ad | 30 | Ath | P1 | Nt | Red | 12.0 | Lon | J4 | Cn | Ath |
| S5 | Ad | 30 | Ath | P3 | Sc | Blue | 17.0 | Osl | J4 | Cn | Ath |
| S5 | Ad | 30 | Ath | P4 | Sc | Red | 14.0 | Lon | J4 | Cn | Ath |
| S5 | Ad | 30 | Ath | P5 | Cm | Blue | 12.0 | Par | J4 | Cn | Ath |
| S5 | Ad | 30 | Ath | P6 | Cg | Red | 19.0 | Lon | J4 | Cn | Ath |

- （FORALL PX）用步骤 1 中 P 表的结果去除以上述结果，得：

| S# | SNAME | STATUS | CITY | J# | JNAME | CITY |
|---|---|---|---|---|---|---|
| S5 | Adams | 30 | Athens | J4 | Console | Athens |

（我们现在可以完全显示结果了）

- （EXISTS JX）再通过投影消去 $J$ 的属性，即 J. J#、J. NAME 和 J. CITY，结果是：

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S5 | Adams | 30 | Athens |

步骤 5：根据原型元组中的选择，对步骤 4 的结果进行投影。在本例中，原型元组是：

```
{ SX.SNAME, SX.CITY }
```

因此，最终结果是：

| SNAME | CITY |
|---|---|
| Adams | Athens |

从前面的叙述可以得出：原始的演算表达式在语义上等价于某一嵌套的代数表达式。精确地

讲，就是：四个选择的积的选择的投影的除的投影的投影！

当然，对这种算法还能做出许多改进（具体参看第18章，特别是文献［18.4］，那里有一些改进的思想），上述例子的解释已经涉及了不少细节；不过，它没有充分地给出简约工作的一般思想。

还有，现在我们能够解释为什么Codd精确地定义了他列出的8个代数操作符的理由之一（其实不仅仅是一个理由）。这8个操作符，作为一种手段，为演算的可能实现提供了一种方便的**目标语言**。例如，QUEL语言就是建立在这个演算的基础上，这种语言实现的一个可能的途径就是用户提交查询——这个查询基本上就是一个演算表达式——再运用简约算法，最后获得一个等价的代数表达式，这些都是在内部实现的。下一步就是继续优化此代数表达式，这在第18章介绍。

另一点要说明的是，Codd的8个代数操作符也为测度任一给定的数据库语言提供了一个标准。在7.6节的后面，我们已简单地介绍了这个问题，这里再深入地讨论一下。

首先，如果一门语言具备了关系演算这样的功能，那么它就可以说是**关系完备的**（relational complete）。这就是说，如果某一关系能用演算表达式定义，那它就一定能被符合关系代数（参看［7.1］）的语言的某种表达所定义。（在第7章，我们说关系完备就是它具备关系代数的功能，而不是关系演算。但是，正如我们将看到的，这两者是一回事。注意，从Codd的简约算法，直接可得到关系代数具有关系完备性。）

一般情况下，关系完备被认为是数据库语言表达能力的一种测度。特别地，由于关系代数和关系演算都具备关系完备性，故不必运用循环（在面向终端用户的语言中，它是一种重要的方法，同时它对程序员也特别有用），就可以为具备这种表达能力的语言的设计提供一个基础。

其次，由于关系代数具备关系完备性，故为了表现某一给定语言$L$具备此特征，就必须充分表现：（a）$L$包括类似8个代数操作符的操作（实际上，要充分表现关系代数的5个基本的操作）；（b）$L$语言的任一操作符的操作数可能是$L$的任一表达。SQL是用这种方式表示关系完备的语言的一个例子（参看练习8.9），而QUEL是另一个例子。事实上，在实践中，要表示一门语言具备关系代数操作比表示它具备关系演算表达容易得多。这就是我们为什么用代数的方式而不用演算的方式解释关系完备性。

还有，要注意，关系完备并不意味着其他方面完备。例如一门语言也可能要求提供计算完备，即它应该能计算可能计算的函数。计算完备是我们在第7章中给关系代数增加EXTEND和SUMMARIZE两个操作符的动机之一。下一节讨论这些操作的关系演算。

我们回到关系代数和关系演算等价的问题上：通过例子我们已经表明，任意演算表达式能够被等价地转化为代数，因此关系代数至少具有关系演算的功能；相反地，代数能够被等价地转化为演算，因此关系演算至少具有关系代数的功能（证明请参看Ullman［8.13］，它很好地说明了这两者逻辑上是等价的）。

## 8.5 计算能力

尽管早些时候没有明确地指出，但事实上，正如我们定义的那样，关系演算已包括类似关系代数中的EXTEND和SUMMARIZE那样的操作，因为：

- 一个可能的原型元组的形式是元组选择子调用 < *tuple selector inv* >（“tuple selector invocation”），并且其组成部分是任意的表达式。
- 一个布尔表达式中比较操作的比较数也可能是一个任意的表达式。
- 正如第7章讲到的，聚集算子调用 < *agg op inv* >（“aggregate operator invocation”）中的首个或仅有的变元是一个关系表达式。

在此没有必要对语法和语义作深入探讨，下面给出了一些例子，并且这些例子也作了一些简化。

**例8.5.1 找出重量超过10 000克的零件的零件号及重量**

```
{ PX.P#, PX.WEIGHT * 454 AS GMWT }
                    WHERE PX.WEIGHT * 454 > WEIGHT ( 10000.0 )
```

注意到原型元组中的 AS 说明（与例 8.3.2 类似），它给出结果中可用的名字。这个名字不能在 WHERE 子句中使用。这就是为什么上式中“ PX. WEIGHT * 454”出现了两次。

**例 8.5.2　找出所有供应商的信息，并且用字母值“Supplier”标记每一个供应商**

```
{ SX, 'Supplier' AS TAG }
```

**例 8.5.3　对每一个供货表，取出所有的供货信息，包括供货总重**

```
{ SPX, PX.WEIGHT * SPX.QTY AS SHIPWT } WHERE PX.P# = SPX.P#
```

**例 8.5.4　对每一个零件，取出每一个零件号和其供货总数量**

```
{ PX.P#, SUM ( SPX WHERE SPX.P# = PX.P#, QTY ) AS TOTQTY }
```

**例 8.5.5　查找总的供货数量**

```
SUM ( SPX, QTY ) AS GRANDTOTAL
```

**例 8.5.6　对每一个供应商，取出其供应商号及其供应的零件的总数量**

```
{ SX.S#, COUNT ( SPX WHERE SPX.S# = SX.S# ) AS #_OF_PARTS }
```

**例 8.5.7　找出保存至少 5 个红色零件的城市**

```
RANGEVAR PY RANGES OVER P ;
PX.CITY WHERE COUNT ( PY WHERE PY.CITY = PX.CITY
                         AND   PY.COLOR = COLOR ('Red') ) > 5
```

## 8.6　SQL 语言

在 8.4 节我们已经谈到，一种给定的关系语言要么基于关系代数，要么基于关系演算。那么 SQL 语言是基于哪一个呢？很遗憾，SQL 只有部分基于这两者，还有一部分并不基于它们。因为提出 SQL 语言时，就明确要求要不同于这两者（参看 [4.9]）。事实上，这个目标就是引入“IN < *subquery* >”结构的动机（参看本节后面的例 10）。随着时间的推移，事实证明关系代数和关系演算的一些特征还是需要的，SQL 也在逐渐开始地接受它们[⊖]。今天，SQL 语言在某些方面像代数式的，在某些方面像演算式的，而在某些方面两者都不像。这一现状说明了为什么在第 7 章我们说将 SQL 数据操作的讨论推迟到这一章讲了。具体 SQL 语言的哪些部分是基于代数的，哪些部分是基于演算的，哪些部分两者都不基于，这作为练习留给读者）。

SQL 查询是以表表达式 < *table exp* > 描述的，但其内部很复杂。这里不介绍复杂的内容，而只简单地举出一些例子，希望这些例子能让我们理解其精髓。这些例子基于第 4 章图 4-1 的供应商和零件数据库。

**例 8.6.1　找出非 Paris 生产的、且重量超过 10 磅的零件的颜色和出产地**

```
SELECT PX.COLOR, PX.CITY
FROM   P AS PX
WHERE  PX.CITY <> 'Paris'
AND    PX.WEIGHT > WEIGHT ( 10.0 ) ;
```

注意：

---

⊖ 这一进步导致的结果是，正如参考文献 [4.19] 提到的那样，“IN < *subquery* >”结构现在可以完全从 SQL 中去掉，同时不会影响 SQL 语言的功能。这一事实具有讽刺意味，因为这一结构正是 SQL 语言的名字“结构化查询语言”中“结构化”所指的部分。实际上，人们也正是因为这一结构首先考虑采用 SQL 而不是关系代数或关系演算。

1）正如第5章讲到的，SQL语法中表示不相等的操作符是“<>”。小于等于和大于等于被记做“≤”和“≥”。

2）FROM子句中的“P AS PX”的含义是：在当前基本表P上定义一个范围变量PX（元组型的）。名字PX（注意这里的PX不是指变量）被称为**相关关系名**（correlation name）。这个名字的作用域就是定义它的表表达式，除非该表表达式的子表达式含有同名的范围变量（参见例8.6.12）。

3）SQL也支持隐式范围变量的概念。如上面的查询也等价于：

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT ( 10.0 ) ;
```

这里基本的思想是：使用表名作为定义在这个表上的隐式范围变量名，这当然不会造成歧义。在此例中，FROM子句“FROM P”可以认为是“FROM P AS P”的缩写。换句话说，如SELECT和WHERE子句中“P.CITY”中的“P”，不是代表基本表P，而是代表同名的、定义在该表上的范围变量——这一点必须明确。

4）正如第4章所述的那样，在此例中我们可以用不受限定的列名，如下所示：

```
SELECT COLOR, CITY
FROM   P
WHERE  CITY <> 'Paris'
AND    WEIGHT > WEIGHT ( 10.0 ) ;
```

一般的规则是：只有在不引起歧义时，不受限定的列名才可以使用。在例子中，我们通常写出了所有的限定词（当然不是所有情况），尽管有时会有冗余。然而，有些上下文中明确地要求列名不受限定！ORDER BY子句就是一个例子[⊖]。请看下面的例子。

5）在第4章的游标定义中讲的*ORDER BY*子句，也能在SQL交互查询中使用，例如：

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT ( 10.0 )
ORDER  BY CITY DESC ;        /* note unqualified column name */
```

6）再要注意的是第4章提到过的“SELECT *”的缩写，例如：

```
SELECT *
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT ( 10.0 ) ;
```

“SELECT *”中的星号是FROM子句中涉及的所有表的所有列的列名称列表的简写，并且它隐含着按原表中列出现的从左到右的顺序。由于此符号的使用，节省了键盘敲击，故在交互式查询中是很方便的。然而，它在嵌入式SQL（即SQL嵌在应用程序中）中却存在潜在的危险。这是因为“*”的意思会发生变化。如用ALTER TABLE给表增加列和删除列。

7）（这一点比前面一点更重要！）现在，给出表中的示例数据，上面的查询将返回四行，而不是两行，即使这里有三行相同。SQL不会自动地从查询结果中删除多余的重复行，要删除的话，必须要求用户在查询中显式地使用关键字**DISTINCT**。例如：

```
SELECT DISTINCT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT ( 10.0 ) ;
```

此查询只返回两行。

---

⊖ 除了第4章4.6节讲到的内容。

正如第6章已经介绍的一样，我们从上面的例子看出SQL的基本数据对象不是关系，而是表。并且一般情况下，SQL表不是行的集合，而是行的包（bag）。这样，SQL就违背了信息原则（information principle）。所以，SQL的基本操作不是真正的关系操作，而是类似包的操作；并且在关系模型里是正确的结果和原理，由于考虑到表达的变化（例如[6.6]），在SQL里都不一定正确了。

**例8.6.2　找出所有零件的零件号及其重量（例8.5.1的简版）**

```
SELECT P.P#, P.WEIGHT * 454 AS GMWT
FROM   P ;
```

这里，“AS GMWT”的说明为结果中的计算列提供了一个合适的列名，于是结果表中的两列分别叫P#和GMWT。如果没有AS子句，则查询结果中相应的列就没有列名。注意，SQL实际上不需要在这样的环境中给查询结果命名，但在我们的例子中仍然这么做了。

**例8.6.3　找出供应商所在地和零件出产地在同一地方的这两者的所有信息**

SQL提供了许多不同的方法完成这一查询，这里给出三个例子：

1）
```
SELECT S.*, P.P#, P.PNAME, P.COLOR, P.WEIGHT
FROM   S, P
WHERE  S.CITY = P.CITY ;
```

2）
```
S JOIN P USING CITY ;
```

3）
```
S NATURAL JOIN P ;
```

在每种情况下，结果都是以表S和P的城市进行**自然连接**。[⊖]

上面的第一个式子值得讨论，它在SQL最初版本中定义，在SQL：1992标准中增加了对显式JOIN操作的支持。从概念上讲，上面的查询是按如下步骤进行的：

- 首先，执行FROM子句，产生S和P的**笛卡尔积**SP。严格地讲，在计算这个积之前，我们应该当心重命名的列，但这里我们简单地忽略了这个问题。而且，正如7.7节中所示，一个单独表的笛卡尔积仍然是其本身。
- 其次，当WHERE子句执行时，就根据每行中两个CITY列的值相等对笛卡尔积进行**选择**，或者说，现在我们已经用CITY列对供应商表和零件表进行了等值连接。
- 最后，当SELECT子句执行时，就根据此列中所指定的列对上述选择结果进行**投影**。最终结果是自然连接。

因此，不严格地说，在SQL中FROM子句对应于笛卡尔积，WHERE子句对应于选择，SELECT子句对应于投影。SQL中的SELECT-FROM-WHERE结构表示了笛卡尔积的选择的投影（虽然这里的投影并没有消除重复的元组）。

**例8.6.4　若供应商为另一个不是他所在的城市供应零件，找出这两个城市名**

```
SELECT DISTINCT S.CITY AS SCITY, P.CITY AS PCITY
FROM   S JOIN SP USING S# JOIN P USING P# ;
```

*注意*：下面的写法是不正确的，因为它在第二个连接中用CITY作为连接列。

```
SELECT DISTINCT S.CITY AS SCITY, P.CITY AS PCITY
FROM   S NATURAL JOIN SP NATURAL JOIN P ;
```

**例8.6.5　若两个供应商在同一城市，找出他们的供应商号（例8.3.2）**

```
SELECT A.S# AS SA, B.S# AS SB
FROM   S AS A, S AS B
WHERE  A.CITY = B.CITY
AND    A.S# < B.S# ;
```

⊖ SQL：2003要求后两种表达方式中包含“SELECT * FROM”前缀。

此例中明确运用了显式范围变量。注意，这里引入了 SA、SB 作为结果表的列名，因此不能在 WHERE 子句中使用。

**例 8.6.6 找出供应商的总数量**

```
SELECT COUNT(*) AS N
FROM   S ;
```

这个查询的结果是一个只有一列一行的表，其列名为 N，其值为 5。SQL 支持聚集操作 **COUNT**、**SUM**、**AVG**、**MA**、**MIN**、**EVERY** 和 **ANY**[○]，但是 SQL 有些特殊的方面用户需要注意：

- 一般地，关键字 DISTINCT 是可选的，位于变元之前，从而在聚集之前删去多余的重复行，如 SUM（DISTINDT QTY）。然而，对于 MAX、MIN、EVERY 和 ANY，DISTINCT 是不影响结果的，因此不应该使用。
- 特别地，对于 COUNT（*）操作，不能在其中使用关键字 DISTINCT。它是统计表中包含重复的所有的行。
- 在自变量列中，除掉 COUNT（*）把 NULL 值当作一存在值进行统计外，其余的都在聚集之前把 NULL 值所在行忽略，而不管变元之前是否加了 DISTINCT 关键字（参看第 19 章）。
- 若自变量是一空集，COUNT 操作返回零值，其他的操作都返回 NULL 值。注意：在逻辑上这对于 COUNT 是正确的，但对其他操作则是不正确的。例如，当 EVERY 作用在空集上时，逻辑上应该返回真，就像我们在 8.2 节看到的一样。

**例 8.6.7 找出零件 P2 的最多和最少供货数量**

```
SELECT MAX ( SP.QTY ) AS MAXQ, MIN ( SP.QTY ) AS MINQ
FROM   SP
WHERE  SP.P# = P# ('P2') ;
```

注意，实际上这里 FROM 子句和 WHERE 子句都运用了聚集操作的变元部分。因此，从逻辑上讲，它们应该用括号括起来，正如本查询这样。这种非正统的语法方法会对 SQL 语言的结构、使用和正交性[○]产生非常消极的影响。例如，一个直接的后果就是聚集查询不能被嵌套，从而导致实现像“获取总零件数量的平均值”这样的查询非常麻烦。明确地讲，下述查询是不合法的：

```
SELECT AVG ( SUM ( SP.QTY ) )          /* warning! illegal! */
FROM   SP ;
```

这一查询应该写成类似下述形式：

```
SELECT AVG ( X )
FROM ( SELECT SUM ( SP.QTY ) AS X
       FROM   SP
       GROUP  BY SP.S# ) AS POINTLESS ;
```

下面的例子将介绍 GROUP BY，和此例中 from 子句中的子查询一样，后面还有几个例子介绍嵌套子查询。注意：AS *POINTLESS* 的说明是无意义的，但是 SQL 语法规则有这样的要求（进一步的讨论请参考 [4.20]）。

---

○ EVERY 与 ALL 相似（我们不支持使用 ALL 这个单词）。ANY 也可以写为 SOME。在“联机分析处理”修订版（SQL/OLAP）中，引入了一些新的聚集操作（参见第 22 章）。

○ 正交性的意思是指独立性，如果一个语言中独立的概念能保持其独立性，不会混淆在一起，那么我们称这个语言是正交的。正交性是需要的，因为低正交性意味着高复杂性，同时意味着较少的功能。

**例 8.6.8　对每一个供应的零件，找出其零件号及其供货总数量（修改的例 8.5.4）**

```
SELECT SP.P#, SUM ( SP.QTY ) AS TOTQTY
FROM   SP
GROUP  BY SP.P# ;
```

下面是关系代数表达式：

```
SUMMARIZE SP BY { P# } ADD SUM ( QTY ) AS TOTQTY
```

或者元组演算表达式：

```
( SPX.P#, SUM ( SPY WHERE SPY.P# = SPX.P#, QTY ) AS TOTQTY )
```

注意，如果运用了 GROUP BY 子句，则 SELECT 子句后面的表述就得**单值分组**（single-valued per group）。

下面是此查询的另一种表述（实际上更好）：

```
SELECT P.P#, ( SELECT SUM ( SP.QTY )
               FROM   SP
               WHERE  SP.P# = P.P# ) AS TOTQTY
FROM   P ;
```

这样使用子查询，我们能得到的结果包括了根本没有供应的零件的行，这一点前面使用 GROUP BY 子句的式子并不能做到（然而，不幸的是，这样的零件的 TOTQTY 值是 NULL 值，而不是零）。

**例 8.6.9　找出多家供应商供应的零件号**

```
SELECT SP.P#
FROM   SP
GROUP  BY SP.P#
HAVING COUNT ( SP.S# ) > 1 ;
```

HAVING 子句是将符合 WHERE 子句的几行分组；或者说，HAVING 的作用是删除一部分分组，就像 WHERE 子句删除行一样。在 HAVING 子句中的表达式必须单值分组。

**例 8.6.10　找出供应零件 P2 的供应商的名字（例 7.5.1）**

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = P# ('P2') ) ;
```

解释：此例运用了 WHERE 子句的子查询。不严格地讲，一个子查询就是将 SELECT-FROM-WHERE-GROUP BY-HAVING 表述嵌在另一个这样的查询里。正如本例所述的那样，在有些情况下使用子查询是要通过 IN 条件表示一系列查询值。这一方式是先进行子查询再完成整个查询——至少概念上是这样的。此例先返回供应零件 P2 的供应商号，即{S1,S2,S3,S4}。上面的表达式等价于如下更简单的表达式：

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN ( S#('S1)', S#('S2'), S#('S3'), S#('S4') ) ;
```

值得提出的是，本题的查询（找出供应零件 P2 的供应商的名字）也可以用连接来完成，如下：

```
SELECT DISTINCT S.SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = P# ('P2') ;
```

**例 8.6.11 找出供应至少一个红色零件的供应商名（例 8.3.4）**

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# IN
            ( SELECT P.P#
              FROM   P
              WHERE  P.COLOR = COLOR ('Red') ) ) ;
```

子查询可以嵌套到任何深度。练习：给出此查询的连接查询形式。

**例 8.6.12 找出状态小于当前 S 表中最大状态值的供应商的供应商号**

```
SELECT S.S#
FROM   S
WHERE  S.STATUS <
     ( SELECT MAX ( S.STATUS )
       FROM   S ) ;
```

本例隐含了两个不同的范围变量，但它们都用“S”表示，且都定义在 S 表上。

**例 8.6.13 找出供应零件 P2 的供应商名**

注意：此例与例 8.6.10 是一样的。这里为了引入 SQL 的另一个特征，我们给出了另一种解：

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.S# = S.S#
       AND    SP.P# = P# ('P2') ) ;
```

解释：SQL 表达式“EXISTS（SELECT … FROM …）”取真值，当且仅当“SELECT … FROM …”取非空值。或者说，SQL 中的 EXISTS 操作符相应于元组演算中的存在量词（参看[19.6]）。注意：在这个特殊的作为相关子查询的例子中，SQL 涉及子查询，因此它包含了一范围变量的引用，即隐式范围变量 S，它在外查询中定义。在例 8.6.8 中的第二种解法也是一个相关子查询的例子。

**例 8.6.14 找出没有供应零件 P2 的供应商的供应商名（例 8.3.7）**

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  NOT EXISTS
     ( SELECT *
       FROM   SP
       WHERE  SP.S# = S.S#
       AND    SP.P# = P# ('P2') ) ;
```

或者：

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  S.S# NOT IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = P# ('P2') ) ;
```

**例 8.6.15 找出供应所有零件的供应商的供应商名（例 8.3.6）**

```
SELECT DISTINCT S.SNAME
FROM   S
```

```
WHERE  NOT EXISTS
     ( SELECT *
       FROM   P
       WHERE  NOT EXISTS
            ( SELECT *
              FROM   SP
              WHERE  SP.S# = S.S#
              AND    SP.P# = P.P# ) ) ;
```

SQL 不直接支持全称量词 FORALL，因此“FORALL”全称量词就不得不像本例这样，利用存在量词和双重否定来表述。

值得注意的是，尽管上述表示乍一看令人觉得烦琐，但它还是能很容易地被熟悉关系演算的用户所构建（参看 [8.4]）。如果上述表示还是很烦琐，那可以用几个“工作区”的方法来代替，这种方法可以避免否定量词。例如，本例可写成：

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  ( SELECT COUNT ( SP.P# )
         FROM   SP
         WHERE  SP.S# = S.S# )
     = ( SELECT COUNT ( P.P# )
         FROM   P ) ;
```

（“取供应商名，它们供应的零件的数量等于所有零件的数量”）。然而，注意：后面的式子依赖于每一个供货的零件号等于某一现存零件的号码，这一点在有 NOT EXISTS 的量词的式子中不需要。或者说仅仅当实现了一定的完整性约束条件（参看下一章）时，这两个式子才等价，并且第二个式子才正确。

注意：之前的例子要做的是比较两个表。因此，上述查询可以如下表示：

```
SELECT DISTINCT S.SNAME                  /* warning! illegal! */
FROM   S
WHERE  ( SELECT SP.P#
         FROM   SP
         WHERE  SP.S# = S.S# )
     = ( SELECT P.P#
         FROM   P ) ;
```

然而，SQL 标准并没有直接支持表的比较，所以我们不得不求助于比较表的基本部分（依赖于我们的外部知识去确保只要表的基本部分是一样的，那么这两个表就是一样的）。请看习题 8.11。

**例 8.6.16　找出那些重量超过 16 磅或者由 S2 供应的或者两者都具备的零件号（例 8.3.9）**

```
SELECT P.P#
FROM   P
WHERE  P.WEIGHT > WEIGHT ( 16.0 )

UNION

SELECT SP.P#
FROM   SP
WHERE  SP.S# = S# ('S2') ;
```

使用不加限定的 **UNION**、**INTERSECT** 或者 **EXCEPT**（SQL 中 EXCEPT 即是 MINUS），结果中多余的重复行会自动地被删去。同时，SQL 也提供了它们的另一种形式，即带限定的 **UNION ALL**、**INTERSECT ALL** 和 **EXCEPT ALL**。使用它们时，对于重复的行，只要有都会保留。这里就不再举例了。

**例 8.6.17　找出重量大于 10 000 克的零件的零件号和重量（以克为单位）（例 8.5.1）**

```
SELECT P.P#, P.WEIGHT * 454 AS GMWT
FROM   P
WHERE  P.WEIGHT * 454 > WEIGHT ( 10000.0 ) ;
```

请读者回忆一下第5章介绍的WITH子句，在第7章中联系关系代数我们曾经使用过它。[⊖] WITH子句的作用是：引入表达式的名字。SQL也包含WITH子句，但只能用于表表达式。例如，我们可以用下面的子句避免重复书写两次P.WEIGHT * 454：

```
WITH T1 AS ( SELECT P.P#, P.WEIGHT * 454 AS GMWT
             FROM   P )
     SELECT T1.P#, T1.GMWT
     FROM   T1
     WHERE  T1.GMWT > WEIGHT ( 10000.0 ) ;
```

注意，WITH子句的开始部分（我们在前面的章节中将其称为 < *name intro* > ）在SQL中的形式是 < *name* > AS ( < *exp* > )，而在**Tutorial D**中的形式是 ( < *exp* > ) AS < *name* > 。我们要注意，对于用SQL来类似地表达关系代数的TCLOSE操作，WITH子句是很重要的。这里不做详细的讨论，读者可以参考练习4.6的在线解答。

这一节举了不少例子，实际上SQL的例子是相当多的。但是，这只讨论了SQL大量的特征的一小部分。实际上，SQL是非常冗杂的语言（参看 [4.19]），它为完成同样的功能提供了大量不同的实现方法。由于篇幅有限，我们就不再讨论所有可能的实现方法，甚至对本节讨论的数量有限的例子也是如此。更多的细节请参考附录B。

## 8.7 域演算

我们现在讨论域演算。正如8.1节所述，域演算不同于元组演算，它是定义在域上而不是定义在元组上。从实际的角度看，语法上最明显的不同是域演算支持布尔表达式的补充形式，我们把这叫作**隶属条件**（membership condition）。一个隶属条件可能采取的形式是：

```
R { <pair commalist> }
```

这里 *R* 是关系变量名，且每一个 *pair* 是 *A x* 的形式，其中 *A* 是 *R* 的一个属性，*x* 或者是域演算的范围变量名，或者是选择子调用（通常用文字描述）。当且仅当存在一个元组，对 *R* 的当前值无论是什么关系，指定的属性都有指定的值时，这一条件取真值。例如表达式：

```
SP { S# S#('S1'), P# P#('P1') }
```

当且仅当当前存在一供货元组，其S#取S1，P#取P1，此隶属条件才为真。同样，下面的隶属条件：

```
SP { S# SX, P# PX }
```

当且仅当当前存在一个供货元组，其S#取范围变量SX的当前值（无论此值是什么），且P#取范围变量PX的当前值（无论此值是什么），上述条件取真值。

在本节的剩余部分，我们假设存在域演算的范围变量，如下所示：

| 域 | 范围变量 |
|---|---|
| S# | SX, SY, ... |
| P# | PX, PY, ... |
| NAME | NAMEX, NAMEY, ... |
| COLOR | COLORX, COLORY, ... |
| WEIGHT | WEIGHTX, WEIGHTY, ... |
| QTY | QTYX, QTYY, ... |
| CHAR | CITYX, CITYY, ... |
| INTEGER | STATUSX, STATUSY, ... |

下面是域演算表达式的一些例子：

```
SX
SX WHERE S { S# SX }
```

---

⊖ 当然它也可以被用来和关系演算联系起来。

```
SX WHERE S { S# SX, CITY 'London' }
{ SX, CITYX } WHERE S { S# SX, CITY CITYX }
              AND   SP { S# SX, P# P#('P2') }
{ SX, PX } WHERE S { S# SX, CITY CITYX }
           AND   P { P# PX, CITY CITYY }
           AND   CITYX ≠ CITYY
```

不严格地说，第一个式子表示所有的供应商号；第二个表示关系变量 S 中所有的供应商号；第三个表示位于伦敦的供应商号；第四个是如下查询的域演算的表示：取供应零件 P2 的供应商的供应商号和所在城市（注意，此查询的元组演算表示需要一个存在量词 EXISTS）；第五个是如下查询的域演算的表示：取供应商所在城市与零件出厂地不在同一地方的供应商号和零件号。

用 8.3 节的示例，我们给出其域演算的表示（有些例子作了稍微的改动）。

**例 8.7.1　找出位于巴黎且其状态大于 20 的供应商的供应商号（简化的例 8.3.1）**

```
SX WHERE EXISTS STATUSX
       ( STATUSX > 20 AND
         S { S# SX, STATUS STATUSX, CITY 'Paris' } )
```

就这个例子来讲，它比元组演算显得有点笨拙，并且量词还不能少。不过，有时情况就恰恰相反，请看下面的例子。

**例 8.7.2　找出所有成对的住在同一城市的供应商的供应商号（例 8.3.2）**

```
{ SX AS SA, SY AS SB } WHERE EXISTS CITYZ
                           ( S { S# SX, CITY CITYZ } AND
                             S { S# SY, CITY CITYZ } AND
                             SX < SY )
```

**例 8.7.3　找出至少供应一个红色零件的供应商名（例 8.3.4）**

```
NAMEX WHERE EXISTS SX EXISTS PX
          ( S { S# SX, SNAME NAMEX }
            AND SP { S# SX, P# PX }
            AND P { P# PX, COLOR COLOR('Red') } )
```

**例 8.7.4　找出至少供应 S2 供应的零件中的一个的供应商名（例 8.3.5）**

```
NAMEX WHERE EXISTS SX EXISTS PX
          ( S { S# SX, SNAME NAMEX }
            AND SP { S# SX, P# PX }
            AND SP { S# S#('S2'), P# PX } )
```

**例 8.7.5　找出供应所有零件的供应商名（例 8.3.6）**

```
NAMEX WHERE EXISTS SX ( S { S# SX, SNAME NAMEX }
            AND FORALL PX ( IF P { P# PX }
                            THEN SP { S# SX, P# PX }
                            END IF )
```

**例 8.7.6　找出不供应零件 P2 的供应商名（例 8.3.7）**

```
NAMEX WHERE EXISTS SX ( S { S# SX, SNAME NAMEX }
                        AND NOT SP { S# SX, P# P#('P2') } )
```

**例 8.7.7　找出至少供应 S2 所供应的零件的供应商号（例 8.3.8）**

```
SX WHERE FORALL PX ( IF SP { S# S#('S2'), P# PX }
                     THEN SP { S# SX, P# PX }
                     END IF )
```

**例 8.7.8 找出重量超过 16 磅或者由 S2 提供或者两者都具备的零件的零件号（例 8.3.9）**

```
PX WHERE EXISTS WEIGHTX
        ( P { P# PX, WEIGHT WEIGHTX }
          AND WEIGHTX > WEIGHT ( 16.0 ) )
        OR SP { S# S#('S2'), P# PX }
```

域演算和关系演算一样，正常情况下都等价于关系代数，即都是关系完备的（详细情况参看 [8.13]）。

## 8.8 QBE

QBE（QUERY-BY-EXAMPLE，参考 [8.14]）是最为著名的基于域演算的语言。（实际上 QBE 是域演算和关系演算的结合，但是更偏重于域演算。）它的简单明了的语法，是基于在空白表中写入条目的思想。举个例子，一个 QBE 查询：找出至少供应 S2 供应的零件中的一个的供应商名，可以写成下面的形式：

| S | S# | SNAME |
|---|---|---|
| | _SX | P._NX |

| SP | S# | P# |
|---|---|---|
| | _SX | _PX |

| SP | S# | P# |
|---|---|---|
| | S2 | _PX |

说明：用户请求系统在屏幕上显示出三个空表，一个供应商表和两个供货关系表，用户在表上填入条目。以下划线开始的条目是示例元素（example element），也就是域演算中的范围变量，其他的条目是具体的取值。用户要求系统输出这样一些（“P.”）供应商名字（_NX），如果供应商的名字是_SX，那么供应商_SX 必须供应零件_PX，并且零件_PX 也由供应商 S2 供应。如果把这个 QBE 查询和与它等价的关系演算和域演算比较（参见例 8.3.5 和例 8.7.4），我们就会发现 QBE 和它们的不同之处在于，QBE 中没有明确的量化[⊖]，这也是它简洁易懂的原因之一。对 QBE 和 SQL 的比较是非常有意义的，详细情形留给读者在练习中体会。

我们引入一组例子来介绍 QBE 的主要特征。作为练习，读者可以将这些 QBE 的例子和用纯域演算的表示进行一下比较。

**例 8.8.1 找出位于 Paris 且其状态大于 20 的供应商的供应商号（例 8.7.1）**

| S | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| | P. | | > 20 | Paris |

我们注意到“>”和“=”的表示是非常简单的。我们也可以不写示例元素，如果它没有在其他地方被引用（当然显式地给出示例元素，例如 P._SX，也是正确的）。像 Paris 一样的字符串可以不加引号（加上引号也正确，并且有时候必须加引号，例如字符串中包含空格的情形）。

我们也可以把“P.”写在靠着条目行的位置，例如

| S | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| P. | | | > 20 | Paris |

这个例子与在每一列写上“P.”是等价的：

| S | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| | P. | P. | P. >20 | P.Paris |

最后一点要指出的是：用户可以对屏幕上的空表进行编辑，增加或删除行和列，加宽或缩短列。表可以被裁剪成合适的状态，以便适应用户的任何操作。不用的列可以删除。例如，在我们

⊖ QUEL 中有相似的讨论，参见 [8.5]。

讨论的第一个 QBE 查询中，SNAME 列就可以删除，如下所示：

| S | S# | STATUS | CITY |
|---|---|---|---|
| | P. | > 20 | Paris |

因此我们在以后的例子中将省略查询中不使用的列。

**例 8.8.2　找出供应的所有零件的零件号，消除重复的元组**

| SP | S# | P# | QTY |
|---|---|---|---|
| UNQ. | | P. | |

UNQ. 代表唯一（unique），类似于 SQL 中的 DISTINCT。

**例 8.8.3　找出位于 Paris 的供应商的供应商号，按照状态的降序、供应商号的升序排序**

| S | S# | STATUS | CITY |
|---|---|---|---|
| | P.AO(2). | P.DO(1). | Paris |

“AO.”代表升序，“DO.”代表降序。括号里的整数代表排序的主次顺序，本例中，STATUS 是主要排序列，S#是次要排序列。

**例 8.8.4　找出位于 Paris 或者其状态大于 20 的供应商的供应商号（修改的例 8.8.1）**

表示“ANDed”的条件写在同一行，而两个条件的“OR”必须写在不同行。例如：

| S | S# | STATUS | CITY |
|---|---|---|---|
| | P. | | Paris |
| | P. | > 20 | |

这个查询的另一种写法使用了条件盒（condition box），如下所示：

| S | S# | STATUS | CITY |
|---|---|---|---|
| | P. | _ST | _SC |

| CONDITIONS |
|---|
| _SC = Paris OR _ST > 20 |

条件盒可以解决当条件太复杂而不能在一列中写出的问题。例如，涉及两列的比较，或者含有聚集操作符的比较。

**例 8.8.5　找出重量在 16 和 19 之间（包括 16 和 19）的零件**

| P | P# | WEIGHT | WEIGHT |
|---|---|---|---|
| | P. | >= 16.0 | <= 19.0 |

**例 8.8.6　找出所有零件的零件号和零件重量（以克为单位）（例 8.6.2）**

| P | P# | WEIGHT | GMWT |
|---|---|---|---|
| | P. | _PW | P. _PW * 454 |

**例 8.8.7　找出供应零件 P2 的供应商名字（例 7.5.1）**

| S | S# | SNAME |
|---|---|---|
| | _SX | P. |

| SP | S# | P# |
|---|---|---|
| | _SX | P2 |

表 SP 中的行在这里隐含了存在量词的限制。这个查询可以解释为：

找出供应商 SX 的供应商名字，使得对于 SX 存在一个供货关系，满足供应商为 SX，供应零件为 P2。

因此 QBE 隐含地支持 EXISTS（注意隐含的范围变量作用在关系上，而不是域上。这就是为什么我们说 QBE 包含关系演算的一些方面。）但是 QBE 不支持 NOT EXISTS。[⊖] 因此，有一些查询，例如，找出供应所有零件的供应商名字（例 8.7.5），无法用 QBE 表示。QBE 不是关系完备的。

**例 8.8.8 找出供应商所在地和零件出产地在同一地方的供应商号和零件号对（修改的例 8.6.3）**

| S | S# | CITY |
|---|---|---|
| | _SX | _CX |

| P | P# | CITY |
|---|---|---|
| | _PX | _CX |

| | | |
|---|---|---|
| P. | _SX | _PX |

这个查询需要三个空表格，一个 S 表，一个 P 表（仅显示相关的列），一个结果表。请读者注意一下为了把三个表连在一起而给出示例元素的方式。这个查询可以解释为：

找出供应商号和零件号对，即 SX 和 PX，使得 SX 和 PX 的地点是同一个城市 CX。

**例 8.8.9 若两个供应商在同一城市，找出他们的供应商号（例 8.6.5）**

| S | S# | CITY |
|---|---|---|
| | _SX | _CZ |
| | _SY | _CZ |

| | | |
|---|---|---|
| P. | _SX | _SY |

如果要求指定额外条件_SX < _SY，可以使用条件盒。

**例 8.8.10 找出零件 P2 供应的总量**

| SP | S# | P# | QTY | |
|---|---|---|---|---|
| | | P2 | _QX | P.SUM._QX |

QBE 支持一般的聚集操作。

**例 8.8.11 对每一个供应的零件，找出其零件号及其供货总数量（例 8.6.8）**

| SP | S# | P# | QTY | |
|---|---|---|---|---|
| | | G.P. | _QY | P.SUM._QY |

“G.”代表分组（与 SQL 中的 GROUP BY 作用相同）。

**例 8.8.12 找出由多于一个供应商供应的零件的零件号**

| SP | S# | P# |
|---|---|---|
| | _SX | G.P. |

| CONDITIONS |
|---|
| CNT._SX >1 |

**例 8.8.13 找出重量超过 16 磅或者由 S2 提供或者两者都具备的零件的零件号（例 8.7.8）**

| P | P# | WEIGHT |
|---|---|---|
| | _PX | > 16.0 |

| SP | S# | P# |
|---|---|---|
| | S2 | _PY |

| | |
|---|---|
| P. | _PX |
| P. | _PY |

⊖ 至少 QBE 部分支持 NOT EXISTS。实际上原来 QBE 完全支持它，但是却常带来麻烦。最基本的问题是：没有办法确定众多隐含的量词的使用顺序，而这一点在出现 NOT 时是十分重要的。因此这样的 QBE 是不明确的。详细的讨论可以参考文献［8.3］和练习 8.2。

**例 8.8.14　将零件 P7（城市 Athens，重量 24，名字和颜色未知）插入表 P 中**

| P | P# | PNAME | COLOR | WEIGHT | CITY |
|---|---|---|---|---|---|
| I. | P7 | | | 24.0 | Athens |

注意“I.”表示插入条目，出现在表名下面。当然，插入新元组并不是关系演算（或关系代数）中的操作。这是一个更新操作，而不是只读操作。我们为了完备性而引入了这个例子，下面的三个例子也是类似的情况。

**例 8.8.15　从供货关系中删除所有数量大于 300 的元组**

| SP | S# | P# | QTY |
|---|---|---|---|
| D. | | | > 300 |

“D”出现在表名下面。

**例 8.8.16　将零件 P2 的颜色改为黄色，重量增加 5，城市改为 Oslo**

| P | P# | PNAME | COLOR | WEIGHT | WEIGHT | CITY |
|---|---|---|---|---|---|---|
| | P2 | | U.Yellow | _WT | U._WT + 5 | U.Oslo |

**例 8.8.17 所有位于 London 的供货商的供货量改为 5**

| SP | S# | QTY |
|---|---|---|
| | _SX | U.5 |

| S | S# | CITY |
|---|---|---|
| | _SX | London |

## 8.9　小结

我们已经简单地介绍了关系代数的替代——**关系演算**。从表面上看，这两者是很不相同的，演算是**描述性**的，而代数是**说明性**的。但是，从更深一层次讲，它们是一样的。因为任何一种演算表示都可以转换为代数表示，反之亦然。

演算有两种，即**元组**演算和**域**演算。它们之间最重要的区别是：元组演算的范围变量定义在关系上，而域演算的范围变量是定义在域上。

元组演算的表达包括一个**原型元组**、一个可选的包含布尔表达式或**合式公式**的 WHERE 子句。此合式公式可允许包含**量词** EXISTS 和 FORALL、**自由的**和**约束的范围变量**、布尔操作符（AND、OR、NOT 等），等等。每一个在合式公式中用到的范围变量必须在原型元组中用到。注意：本章没有明确地讨论这一点，但是演算表示必须明确地满足用于同一目的的代数表示（参看 7.6 节）。

我们通过例子说明了 Codd 的**简约算法**如何将任意的演算表示转化为等价的代数表示，这样就为演算策略的实现提供了方便。我们再一次讨论了**关系完备性**，并且就此简单地讨论了语言的完备性。

我们也在元组演算中讨论了**计算**功能，它类似于代数中 EXTEND 和 SUMMARIZE 提供的功能。然后，我们对 **SQL** 的相关特征作了概述。SQL 混合了代数和（元组）演算。例如，它包括直接对 JOIN 和 UNION 代数操作符的支持，但是它也使用演算的范围变量和存在量词。

SQL 查询由**表达式**组成。通常，一个操作由一个**选择表达式**组成，但也支持多种形式的 **JOIN** 表达式，且以各种方式通过使用 **UNION**、**INTERSECT** 和 **EXCEPT** 操作符把 JOIN 和 SELECT 表达式组合起来。我们也谈到了 **ORDER BY** 子句，它是对从（任何种类的）表达式导出的表进行强制性排序。特别地，对于选择表达式，我们再说几点：

- 基本的 **SELECT 子句**，包括 **DISTINCT** 关键字的使用、计算表达式的使用、结果列名说明及“SELECT *”。

- FROM 子句，包括范围变量的使用。
- WHERE 子句，包括 EXISTS 量词的使用。
- GROUP BY 和 HAVING 子句，包括聚集操作 COUNT、SUM、AVG 等。
- 在 SELECT、FROM 和 WHERE 子句中使用子查询。○

我们还给出了 SQL 表达式的**概念性的计算算法**（conceptual evaluation algorithm），即 SQL 选择表达式形式化定义的轮廓。这个算法的摘要包括：（a）FROM 子句中的表进行笛卡尔积，（b）根据 WHERE 子句中的布尔表达式进行筛选，（c）将结果投影到 SELECT 子句确定的列上。当然这个算法摘要是不完全的，详细的解释请参见［4.20］。

随后，我们简单地介绍了**域演算**，并认为它具备关系完备性，尽管没有证明。这样，元组演算、域演算和关系代数这三者是相互等价的。最后简单介绍了 **Query-By-Example**，它是域演算思想应用于商业的成功范例。

## 习题

8.1 设 $p(x)$ 和 $q$ 是任意的合式公式，其中 $x$ 为自由变量，分别为出现和不出现。下面哪一个式子是正确的？（符号“=>”表示隐含，符号“≡”表示恒等，“A => B”与“B => A”在一起使用时，就表示“A≡B”。）

a. EXISTS $x$ ( $q$ ) ≡ $q$

b. FORALL $x$ ( $q$ ) ≡ $q$

c. EXISTS $x$ ( $p(x)$ AND $q$ ) ≡ EXISTS $x$ ( $p(x)$ ) AND $q$

d. FORALL $x$ ( $p(x)$ AND $q$ ) ≡ FORALL $x$ ( $p(x)$ ) AND $q$

e. FORALL $x$ ( $p(x)$ ) ⇒ EXISTS $x$ ( $p(x)$ )

f. EXISTS $x$ ( TRUE ) ≡ TRUE

g. FORALL $x$ ( FALSE ) ≡ FALSE

8.2 设 $p(x,y)$ 是任意的合式公式，$x$、$y$ 为自由变量。下面哪一种说法是正确的？

a. EXISTS $x$ EXISTS $y$ ( $p(x,y)$ ) ≡ EXISTS $y$ EXISTS $x$ ( $p(x,y)$ )

b. FORALL $x$ FORALL $y$ ( $p(x,y)$ ) ≡ FORALL $y$ FORALL $x$ ( $p(x,y)$ )

c. FORALL $x$ ( $p(x,y)$ ) ≡ NOT EXISTS $x$ ( NOT $p(x,y)$ )

d. EXISTS $x$ ( $p(x,y)$ ) ≡ NOT FORALL $x$ ( NOT $p(x,y)$ )

e. EXISTS $x$ FORALL $y$ ( $p(x,y)$ ) ≡ FORALL $y$ EXISTS $x$ ( $p(x,y)$ )

f. EXISTS $y$ FORALL $x$ ( $p(x,y)$ ) ⇒ FORALL $x$ EXISTS $y$ ( $p(x,y)$ )

8.3 设 $p(x)$ 和 $q(y)$ 是任意的合式公式，$x$、$y$ 分别为自由变量。下面哪一种说法是正确的？

a. EXISTS $x$ ( $p(x)$ ) AND EXISTS $y$ ( $q(y)$ ) ≡
EXISTS $x$ EXISTS $y$ ( $p(x)$ AND $q(y)$ )

b. EXISTS $x$ ( IF $p(x)$ THEN $q(x)$ END IF ) ≡
IF FORALL $x$ ( $p(x)$ ) THEN EXISTS $x$ ( $q(x)$ ) END IF

8.4 再看一次这个查询“找出至少供应 S2 供应的零件的供应商的供应商号”，下面是一个可能的元组演算表达式：

```
SX.S# WHERE FORALL SPY ( IF SPY.S# = S# ('S2') THEN
                           EXISTS SPZ ( SPZ.S# = SX.S# AND
                                        SPZ.P# = SPY.P# )
                         END IF )
```

（这里 SPZ 是另一个定义在供货表上的范围变量。）如果当前 S2 什么零件都不供应，则本查询返回什

○ 我们现在发现，不严格地说，FROM 子句中的子查询被当作表表达式处理，SELECT 子句中的子查询被当作标量表达式处理，WHERE 子句中的表达式根据内容的不同，被当作表表达式或者标量表达式处理。

么？如果本查询中所有的 SX 都替换为 SPX，它又有什么不同？

8.5　下面是一个对供应商－零件－工程数据库的查询（这是考虑范围变量时的习惯应用）：

```
{ PX.PNAME, PX.CITY } WHERE FORALL SX FORALL JX EXISTS SPJX
                                ( SX.CITY = 'London' AND
                                  JX.CITY = 'Paris' AND
                                  SPJX.S# = SX.S# AND
                                  SPJX.P# = PX.P# AND
                                  SPJX.J# = JX.J# AND
                                  SPJX.QTY < QTY ( 500 ) )
```

a. 把这些查询转化为自然语言。

b. 对这个查询运行 DBMS，并“执行” Codd 的简化算法，这样你能看出一些改进吗？

8.6　用元组演算表达式表示查询“取出三个最重的零件”。

8.7　思考在第 4 章练习 4.6 中的材料清单的关系变量 PART_STRUCTURE。对于知名的**零件探寻**（part explosion）查询“找出组成零件 P1 的任何层次上的零件的零件号”——此查询的结果，依据 PART_BILL，一定是从 PART_STRUCTURE 导出的一个关系。此查询最初不能用关系代数或关系演算独立地完成。或者说，PART_BILL 是一个可导出关系，但它不能由单独的原始关系代数或关系演算导出。请问这是为什么？

8.8　假设供应商关系变量 S 可被另一些关系变量（如 LS，PS，AS，等等）所代替（每一个代表在不同城市的供应商，如 LS 仅仅表示在伦敦的供应商的元组）；假设我们不知道有哪些供应商所在城市存在，并因此不知道有多少这样的关系变量。对于查询“此库中有供应商 S1 吗？”能用演算或代数表示吗？请给出你的答案。

8.9　叙述 SQL 的关系完备性。

8.10　SQL 中有与关系操作 EXTEND 和 SUMMARIZE 等价的操作吗？

8.11　SQL 中有与比较操作等价的操作吗？

8.12　对于查询“找出供应零件 P2 的供应商名”，请用尽可能多的 SQL 表示方法进行表示。

## 查询练习

下面的练习要用到供应商－零件－工程数据库。要求对下面的每一个查询给出一个表达式。（作为一个有意思的变换，你也可以先看解答并用自然语言来解释这个表达式。）

8.13　写出习题 7.13～7.50 的元组演算表达式。

8.14　写出习题 7.13～7.50 的 SQL 表达式

8.15　写出习题 7.13～7.50 的域演算表达式。

8.16　写出习题 7.13～7.50 的 QBE 表示。

## 参考文献

[8.1]　E. F. Codd：“A Data Base Sublanguage Founded on the Relational Calculus，”Proc. 1971 ACM SIGFIDET Workshop on Data Description，Access and Control，San Diego，Calif.（November 1971）.

[8.2]　C. J. Date：“A Note on the Relational Calculus，”*ACM SIGMOD Record 18*，No. 4（December 1989）. Republished as“An Anomaly in Codd's Reduction Algorithm”in C. J. Date and Hugh Darwen，*Relational Database Writings 1989－1991*. Reading，Mass.：Addison-Wesley（1992）.

[8.3]　C. J. Date：“Why Quantifier Order Is Important，”in C. J. Date and Hugh Darwen，*Relational Database Writings 1989－1991*. Reading，Mass：Addison-Wesley（1992）.

[8.4]　C. J. Date：“Relational Calculus as an Aid to Effective Query Formulation，”in C. J. Date and Hugh Darwen，*Relational Database Writings 1989－1991*. Reading，Mass.：Addison-Wesley（1992）.

目前，市场上每一个有关关系的产品都支持 SQL，而不是关系代数或关系演算。而这篇论文提倡并解释了通过关系演算作为中介去构建复杂的 SQL 查询。

[8.5]　G. D. Held，M. R. Stonebraker，and E. Wong：“INGRES——A Relational Data Base System，”Proc. NCC *44*，Anaheim，Calif，Montvale，N. J.：AFIPS Press（May 1975）.

20 世纪 70 年代中后期，发展了两个重要的关系原型。这就是 IBM 的 System R 和加州大学伯克利分校的 Ingres（或写作 INGRES）。在研究领域，它们非常有影响，并相继导致了两个商业系统，即 System R 环境下的 DB2 和 Ingres 环境下的商业 Ingres 产品。注意：有时，Ingres 原型叫做“大学 Ingres”（参看［8.11］），从而与“商业 Ingres”区分开来。文献［1.5］对这一商业版

本作了指导性的概述。

起初，Ingres 并不是一个 SQL 系统，它所支持的语言叫做 QUEL（Query Language）。这一语言在技术上许多方面都超过了 SQL。事实上，现在许多数据库的研究都基于此语言，并且 QUEL 中使用的例子还出现在许多论文里。这篇论文首先介绍了 Ingres 原型系统，接着对 QUEL 作了初步解释。还可参看 [8.10～8.12]。

[8.6] J. L. Kuhns: "Answering Questions by Computer: A Logical Study" Report RM-5428-PR, Rand Corp., Santa Monica, Calif. (1967).

[8.7] M. Lacroix and A. Pirotte: "Domain-Oriented Relational Languages" Proc. 3rd Int. Conf. on Very Large Data Bases, Tokyo, Japan (October 1977).

[8.8] T. H. Merrett: "The Extended Relational Algebra, A Basis for Query Languages" in B. Shneiderman (ed.), *Databases: Improving Usability and Responsiveness*. New York, N. Y.: Academic Press (1978).

本文建议在关系代数中引入量词——不仅仅是关系演算的存在量词和全称量词，而且包括一般的像"……的数量是……"和"……的部分是……"的量词。如："至少……的三个"、"不超过……的一半"、"……的奇数个数"，等等。

[8.9] M. Negri, G. Pelagatti, and L. Sbattella: "Formal Semantics of SQL Queries," *ACM TODS 16*, No. 3 (September 1991).

引用如下摘要："SQL 查询的语法被一系列规则所定义，这些规则决定了把其翻译成一种范式模型。这一模型叫做扩展三值谓词演算（Extended Three - Valued Predicate Calculus, E3VPC），它在很大程度上基于有名的数学概念。这里也给出了将一个一般的 E3VPC 表达式转化为一个规范形式的规则。而且这里也解决了诸如 SQL 查询的等价分析问题。"然而请注意，这些 SQL 所考虑的语言仅是它的最初版（1986）。关于三值和规范形式请参见第 19 章和第 18 章。

[8.10] Michael Stonebraker (ed.): *The INGRES Papers: The Anatomy of a Relational Database Management System*. Reading, Mass.: Addison-Wesley (1986).

这是"大学 Ingres"项目的一些重要的论文集，由最初 Ingres 的设计者们编辑、注释。

[8.11] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held: "The Design and Implementation of INGRES." *ACM TODS 1*, No. 3 (September 1976). Republished in reference [8.10].

本书是关于大学 Ingres 原型的详细论述。

[8.12] Michael Stonebraker: "Retrospection on a Data Base System," *ACM TODS 5*, No. 2 (June 1980). Republished in reference [8.10].

本书介绍了 Ingres 原型项目的历史（到 1979 年 1 月）。重点是介绍其失败和教训，而不是成功经验。

[8.13] Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems: Volume I*. Rockville, Md.: Computer Science Press (1988).

Ullman 的这本书比我们这本书更加规范地处理了关系演算及其相关方面。特别地，它讨论了演算表达式**安全性**的概念。如果我们对这一演算稍作改动，会变得更有意义。因为这一演算中的范围变量并不是由独立的语句所定义，而是通过 WHERE 子句中明确的表述来约束其范围。在这种演算表示中，查询"找出在伦敦的供应商"看起来似乎如下表示：

```
X WHERE X ∈ S AND X.CITY = 'London'
```

很明显，这种演算带来的一个问题（不仅仅是这一个问题）是它允许如下的查询：

```
X WHERE NOT ( X ∈ S )
```

这样的表达式是不安全的，因为它不能返回有限的结果集（非 S 元组的所有项是无限的）。于是，必须有一定的规则去强制保证那些合法的表达式是安全的。这些规则在 Ullman 的这本书里作了描述（包括元组演算和域演算）。我们注意到，Codd 的最初的演算并没有包括这些规则。

[8.14] Moshé M. Zloof: "Query-By-Example," Proc. NCC 44, Anaheim, Calif. (May 1975). Montvale, N. J.: AFIPS Press (1977).

Zloof 是 QBE 的最初的发明者及设计者。这篇文章是 Zloof 在这方面所有论文的第一篇。

# 第9章 完 整 性

## 9.1 引言

关系模型的完整性部分是这些年来变化最大的一部分（也许我们应称其为进化而不是变化）。最初人们只注重主外码上的变化（简称“码”），然而，完整性约束在更广范围内的重要性（的确十分重要）逐渐得到了越来越多的理解和赞同。与此同时，关于码的一些古怪的问题渐渐多了起来。本章从结构上反映了这样的变化，首先它较为深入地讨论了更广范围内的完整性约束问题，然后讨论了码（仍然是一个实际中的主要问题）。

不严格地说，**完整性约束**是数据库相关的布尔表达式，需要在任何时候得到满足。这样的约束被视为是关于一些“商业规则”的正规表达式［9.15］——为了本章介绍需要，尽管商业规则通常用自然语言表达，但有时也用完整性规则描述。下面列出了一些例子，它们都是基于“供应商和零件”数据库：

1）每个供应商的状态值都在1到100的范围内取值（含1和100）。

2）每个伦敦供应商的状态号都是20。

3）如果有一些零件的话，它们中至少有一个是蓝色的。

4）不允许出现两个不同的供应商拥有相同的供应商号。

5）每次供货都要有一个存在的供应商。

6）不允许状态号小于20的供应商供应任何数量多于500的零件。

在本章中，我们将广泛使用以上的例子。

显然，约束应向DBMS正式声明，然后DBMS强制执行它们。声明约束只是用到了数据库语言的相关特性；而执行时，DBMS监控更新程序将有可能违反并拒绝执行约束。以下是与上面列出的第一个例子（在**Tutorial D**中）相关的正式声明：

```
CONSTRAINT SC1
    IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

为了强制执行这个约束，DBMS将监控所有试图插入一个新的供应商或者改变供应商状态的操作［9.5］。

当然，每当一个约束被首次声明时，系统必须检查数据库在当前状况下是否能满足它。如果不能，约束必须被拒绝；否则，约束被接受——也就是说，该约束被存于系统目录中。顺便说一句，请注意例子中的约束名SC1（“供应商约束1”）。假定该约束被DBMS所接受，它将注册到系统字典表的SC1的名下，当试图违反它时，约束名SC1就会出现在系统生成的错误报告消息中。

下面是例1的两个更可能的式子，基于**Tutorial D**的演算版本（这里的SX是在供应商上的范围变量）：

```
CONSTRAINT SC1
    NOT EXISTS SX ( SX.STATUS < 1 OR SX.STATUS > 100 ) ;

CONSTRAINT SC1
    FORALL SX ( SX.STATUS ≥ 1 AND SX.STATUS ≤ 100 ) ;
```

当然，这三个声明是等价的。然而在本章中，我们的讨论将更多地基于演算而非代数，原因会随着讨论的深入而逐渐清晰。作为练习，你可能想试着对我们给出的基于演算的例子得出相应的代数形式。

当然，如果现有的约束已不再需要了，我们也需要有一种去除它们的方法：

```
DROP CONSTRAINT <constraint name> ;
```

## 9.2 进一步讨论

一般的完整性约束是建立在某个变量或某些变量的组合上的。[⊖] 于是，一个给定的变量属于某种类型表示出一种前方（priori）的约束（能通过变量推测的值一定是该变量类型的值）。于是，我们立刻得出这样一个结论（的确，它只是一种特殊情况），一个给定的关系变量的所有属性属于某个特定类型的事实代表了一个在问题关系变量上的前方约束。例如，关系变量 S（供应商）被约束包含特定的值，它们是一些关系，其中每一个 S#值是一个供应商号（一个 S#类型值），每个 SNAME 的值是个名字（NAME 类型的值），等等。

然而，这些简单的“前方”约束当然不是仅有的约束；事实上，9.1 节给出的 6 个例子中没有一个是该意义上的前方约束。再次考虑例 1：

1）每个供应商的状态值都在 1 到 100 的范围内取值（含 1 和 100）。

更精确地讲，如果 s 是供应商，那么 s 有一个在 1 到 100（含 1 和 100）范围内的状态值。

下面是一个更精确的（或更正规的）方式：[⊖]

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S
    THEN st ≥ 1 AND st ≤ 100 )
```

该正规表达式可以读作以下方式（很呆板的说法）：

*对于所有的供应商号 s#，所有的名字 sn，所有的整数 st 和所有的字符串 sc，如果一个拥有 S# s#，SNAME sn，STATUS st 和 CITY sc 的元组出现在供应商变量中，那么 st 大于等于 1 且小于等于 100。*

也许现在你能明白，为什么刚才我们会给出例 1 的替代的自然语言版本。事实上，那个替代的版本相应的正规表达式以及呆板的自然语言模拟总体上都有某种“形式”，看起来如下：

如果某元组出现在某个关系变量中，那么那个元组满足一定的条件。

这种“形式”是一个**逻辑蕴涵**的例子（有时称为物质包含）。我们以前曾遇到过这种构造，在第 8 章；它以通用形式出现：

```
IF p THEN q
```

*p* 和 *q* 都是布尔表达式。分别称作**前项**和**后项**。如果 *p* 取真值，*q* 取假值，则整个表达式取假值，其他情况取真值；换言之，IF *p* THEN *q* 本身是一个布尔表达式，它逻辑上等价于（NOT *p*）OR *q*。

顺便说一下，注意一下前面的形式是如何默认包含必要的 FORALL 量词的——“如果一个元组出现”默认表示“全部的元组都出现”。

现在我们进一步类似地分析例 2 到例 6（但忽略呆板的自然语言式子）。注意：接下来的式子并非是唯一的，也不一定是最简单的，但它们至少是正确的。同时注意，每个例子都至少表示了一个新的观点。

2）每个 London 供应商的状态号都是 20。

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S
    THEN ( IF sc = 'London'
           THEN st = 20 ) )
```

在这个例子中，蕴涵的结果本身是一个蕴涵。

3）如果有一些零件的话，它们中至少有一个是蓝色的。

---

⊖ 完整性约束适用于（至少原则上）所有类型的变量，然而，很明显，在本书中我们主要特指关系变量。

⊖ 请注意这些正式的例子中用的语法既不是 **Tutorial D**（**Tutorial D** 版本的例子稍后给出），也不是我们在第 8 章中为关系定义的语法。尽管看起来相似（特别是关于域的形式）。

```
IF
EXISTS p# ∈ P#, pn ∈ NAME, pl ∈ COLOR, pw ∈ WEIGHT, pc ∈ CHAR
  ( { P# p#, PNAME pn, COLOR pl, WEIGHT pw, CITY pc } ∈ P )
THEN
EXISTS p# ∈ P#, pn ∈ NAME, pl ∈ COLOR, pw ∈ WEIGHT, pc ∈ CHAR
  ( { P# p#, PNAME pn, COLOR pl, WEIGHT pw, CITY pc } ∈ P
    AND pl = COLOR ('Blue') )
```

注意，我们不能只说“至少有一个零件是蓝色的”——我们必须考虑到没有任何零件的情况。注意：尽管这种情况可能不明显，本例同前两个例子一样遵从同样的通用形式。下面是一个可以使本观点更加清楚的替代公式。

```
FORALL p# ∈ P#, pn ∈ NAME, pl ∈ COLOR, pw ∈ WEIGHT, pc ∈ CHAR
  ( IF { P# p#, PNAME pn, COLOR pl, WEIGHT pw, CITY pc } ∈ P
    THEN EXISTS q# ∈ P#, qn ∈ NAME, ql ∈ COLOR,
                         qw ∈ WEIGHT, qc ∈ CHAR
           ( { P# q#, PNAME qn, COLOR ql,
                      WEIGHT qw, CITY qc } ∈ P
             AND ql = COLOR ('Blue') ) )
```

4）不允许出现两个不同的供应商拥有相同的供应商号。

```
FORALL x# ∈ S#, xn ∈ NAME, xt ∈ INTEGER, xc ∈ CHAR,
       y# ∈ S#, yn ∈ NAME, yt ∈ INTEGER, yc ∈ CHAR
  ( IF { S# x#, SNAME xn, STATUS xt, CITY xc } ∈ S AND
       { S# y#, SNAME yn, STATUS yt, CITY yc } ∈ S
    THEN ( IF x# = y#
           THEN xn = yn AND xt = yt AND xc = yc ) )
```

这个表达式仅仅是“｛S#｝是供应商的一个候选码或至少是一个超码”的正规声明；于是码约束仅是一般约束的特例。**Tutorial D** 语法 KEY｛S#｝可能是前面的冗长表达式的一种简单记法。（顺便注意一下括号：码通常情况下是属性的集合——尽管当前讨论的集合只包含一个属性——因此我们把码属性包含在括号中，至少在正式的上下文之中是这样。）注意：在9.10节中详细讨论了候选码和超码。

顺便注意一下，本例采用了总体形式：

IF 一些元组出现在某些关系变量中，THEN 那些元组满足一定的条件。

比较例2和例3，它们都与例1有相同的形式（我们稍后将要看到的例5也是一样）。与之形成对比的是，例6采用了如下总体形式：

IF 某些元组出现在某些关系变量中，THEN 那些元组满足一定的条件。

后一种形式适用于通用的完整性约束（前两种可视为通用情况的特例）。

5）每次供货都要有一个存在的供应商。

```
FORALL s# ∈ S#, p# ∈ P#, q ∈ QTY
  ( IF { S# s#, P# p#, QTY q } ∈ SP
    THEN EXISTS sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
           ( { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S ) )
```

这个表达式是｛S#｝是供货的一个外码的正规声明，与供应商候选码｛S#｝匹配；于是，外码约束也只是通用完整性约束的一个特例（同样，留意9.10节的进一步讨论）。注意，本例包含两个不同的关系变量，SP和S，而例1至例4都只包含一个。[⊖]

6）不允许状态号小于20的供应商供应任何数量多于500的零件。

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR,
       p# ∈ P#, q ∈ QTY
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S AND
       { S# s#, P# p#, QTY q } ∈ SP
```

⊖ 本书的上一版使用术语关系变量约束表示一个只包含一个关系变量的约束，使用术语数据库约束表示包含多于一个的约束。正如我们将在9.9节中看到的那样，这一区别更多是语义上的而不是逻辑上的，这一点在接下来的文章中将不再强调。

```
THEN st ≥ 20 OR q ≤ QTY ( 500 ) )
```

本例也包含两个不同的关系变量，但它不是外码约束。

Tutorial D 例子

我们以例 2～6 的 Tutorial D 版本来结束本节的讨论。我们采用关于范围变量名称的习惯用法。

1）每个 London 供应商的状态号都是 20。

```
CONSTRAINT SC2
    FORALL SX ( IF SX.CITY = 'London'
                THEN SX.STATUS = 20 END IF ) ;
```

注意，在 **Tutorial D** 中逻辑蕴含（IF/THEN 表达式）包含一个“END IF”结束符。

2）如果有一些零件的话，它们中至少有一个是蓝色的。

```
CONSTRAINT PC3
    IF EXISTS PX ( TRUE )
    THEN EXISTS PX ( PX.COLOR = COLOR ('Blue') ) END IF ;
```

3）不允许出现两个不同的供应商拥有相同的供应商号。

```
CONSTRAINT SC4
    FORALL SX FORALL SY ( IF   SX.S#     = SY.S#
                          THEN SX.SNAME  = SY.SNAME
                          AND  SX.STATUS = SY.STATUS
                          AND  SX.CITY   = SY.CITY
                          END IF ) ;
```

4）每次供货都要有一个存在的供应商。

```
CONSTRAINT SSP5
    FORALL SPX EXISTS SX ( SX.S# = SPX.S# ) ;
```

5）不允许状态号小于 20 的供应商供应任何数量多于 500 的零件。

```
CONSTRAINT SSP6
    FORALL SX FORALL SPX
         ( IF SX.S# = SPX.S#
           THEN SX.STATUS ≥ 20 OR SPX.QTY ≤ 500 END IF ) ;
```

## 9.3 谓词和命题

再次考虑例 1 的正式形式（每个供应商的状态值都在 1 到 100（含 1 和 100）的范围内）：

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S
    THEN st ≥ 1 AND st ≤ 100 )
```

这个正式形式是一个布尔表达式。注意，它包含一个变量：供应商关系变量 S。[⊖] 因此，我们不能判断表达式的值是多少——即直到我们给该变量赋予具体值的时候，才能知道最后的结果。当实例化谓词，换句话说，想检查约束的——这个约束是由我们提供的，作为关系，即关系变量 S 的当前值的实参（关系变量 S 是唯一的形参时），表达式才会被重新求值。

现在，我们真正地实例化一个谓词，实际上就是用一些实参代替唯一的形参，我们建立一个不包含任何变量的真值表达式，这种方法同样也适用于包含 2 个、3 个、4 个或任意个关系变量的约束。在任何情况下，当我们想要对表达式求值即检查约束时，我们用可应用关系变量的当前值换每一个参数，得到一个不包含任何变量的真值表达式，即命题。一个命题要么为真，要么为假。明确地说，可以把命题看作退化的谓词，就像是我们在前一章中看到的参数为空的谓词。这

---

⊖ 它涉及几个范围变量，但是我们在第 8 章已经看到，范围变量在编程语言中不是变量，本章我们使用术语变量特指编程语言中的变量。

里有一些简单的例子：

- 太阳是恒星。
- 月亮是恒星。
- 太阳距我们的距离比月亮远。
- 布什在2000年赢得了美国总统竞选。

这些命题的真假判断我们在习题中完成。注意，并不是所有的命题都是正确的，认为每个命题都为真这种想法是错误的。

本节的核心在于：形式化的约束描述为谓词。然而，当检查约束时，需要具体的参数才能判断真假，因此形成了“命题”。

## 9.4　关系变量谓词和数据库谓词

显然，一个给定的关系变量将受到许多约束的限制。令 *R* 是一个关系变量，于是 *R* 的**关系变量谓词**是用于 *R*（或涉及 *R*）的所有约束的逻辑与或合取（conjunction）。这里有可能产生混淆：正如我们已知的那样，每个约束本身是一个谓词；然而，*R* 的关系变量谓词是作用于 *R* 的所有独立谓词的合取。例如，供应商和零件数据库的约束（除了在前约束），那么对应供应商的关系变量谓词是数字1、2、4、5和6的合并，同时，供货的关系变量谓词是5和6的合并。注意，这两个关系多少有一点“互相重叠”，因为它们在组成上有公共的约束。[⊖]

现在令 *R* 是一个关系变量，令 *RP* 是 *R* 的关系变量谓词。显然，当 *R* 代表了 *RP* 中的 *R* 时（当 *RP* 中其他任何需要的参数都已经确定时），*R* 不允许是一个具体的数值。于是，现在我们可以引入**黄金法则**（第一版）：

*如果一个更新操作将使一个关系变量处于违反自身某个谓词的状态，这样的更新是被禁止的。*

现在，令 *D* 是一个数据库，[⊖] 并令 *D* 只包含关系变量 *R*1，*R*2，…，*Rn*。令对应的关系变量谓词分别为 *RP*1，*RP*2，…，*RPn*。于是 *D* 对应的**数据库谓词** *DP*，是所有的关系变量谓词的合取：

```
DP  =  RP1 AND RP2 AND ... AND RPn
```

下面是扩展的（更一般的，也是最终的）**黄金法则**：

*如果一个更新操作将使一个数据库处于违反自身某个谓词的状态，这样的更新是被禁止的。*

当然，一个数据库谓词会取 FALSE 当且仅当组成它的关系变量谓词中至少一个也取 FALSE。一个关系变量谓词会取 FALSE 当且仅当组成它的约束中至少一个也取 FALSE。*注意*：正如我们看到的，两个不同的关系变量谓词 *RPi* 和 *RPj*（$i \neq j$）可能包含公共的约束。因此，在数据库谓词 *DP* 中相同的约束可能出现许多次。从逻辑的观点来看，这种情况并没有什么危害，因为如果 *c* 是一个约束，那么 *c* AND *c* 与 *c* 逻辑上等价。因此，尽管在这种情况下系统取值 *c* 一次（而非两次）显然是我们所期望的，但这是系统实现方面的而非模型方面的问题。

## 9.5　约束检查

本节着重解决两个问题，一个与实现有关，另一个与模型有关，二者都与检查已声明的约束有关。首先让我们考虑实现问题。再次参见例1，它有效地表明如果某个元组出现在关系变量 S 中，那么元组必须满足某个约束（“状态范围从1到100”）。特别注意约束涉及在关系变量中的元组。因此很显然，如果我们在视图中插入一个新的状态号为（比方说）200的供应商元组，接下来发生的一系列事件将是：

1）插入那个新的元组。

---

⊖ 本书的以前版本为关系变量 *R* 定义了关系变量谓词，即应用于 *R* 的所有关系变量约束的并。但现在我们在[3.3]中定义它为所有约束的并，而不只是应用于 *R* 的关系变量的约束。对于发现这一变化，并产生术语混淆的读者，我们表示道歉。

⊖ 当然，*D* 是一个变量（见文献[3.3]注解），因此受完整性约束限制。

2）检查约束。

3）撤消更新（因为检查失败）。

但这将是荒谬的！很明显，我们想在第一个操作“插入”完成之前就捕获错误。因此实现必须在插入操作完成之前，用约束的正规表达式来推断将在待插入元组上执行的恰当的一个或多个检查。

原则上，推理过程是相当直接的。具体来说，如果数据库谓词包含如下形式的约束

```
IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S
THEN ...
```

即，若整个数据库谓词中的某些蕴涵的前项以“一些元组出现在 S 中”的形式出现，那么蕴涵的后项本质上讲是一个在将要插入关系变量 S 的元组上的约束。注意：如果数据库是按照第 13 章介绍的正交设计原理设计的话，假设 DBMS 具有一致性，那么每一个都要满足预先定义的规则。

现在，我们转向讨论模型的问题（当然，它是更为基础的问题）。再次考虑到**黄金法则**：

*如果一个更新操作将使一个数据库处于违反自身某个谓词的状态，这样的更新是被禁止的。*

尽管我们并没有在 9.4 节中显式地声明这一点，你也许已经意识到，这个法则中所有的约束检查都是立即的。为什么呢？因为它涉及更新操作而非事务操作（见下一小段）。因此，事实上**黄金法则**在声明结束的地方被满足，而且根本没有延迟或提交时间的完整性检查。[⊖]

现在，你可能已经了解到刚才的问题（即所有的约束检查都是立即的）是非常不正规的；许多文献（包括本书的早期版本）认为（或简单假定）：“完整性单元”是事务并且某个检查至少必须被推迟到事务结束（即提交时）。然而，有充分的理由表明为什么事务作为“完整性单元”是不合适的，而声明才是真正的单元。不幸的是，若没有一定的关于事务概念的背景知识，是不太可能去解释那些原因的。因此，我们把详细的讨论推迟到第 16 章；在这之前，我们简单认为（并未深入修正），立即检查在逻辑上是正确的。（然而，一个支持我们的观点的理由可在本章末尾的文献［9.16］的注解中找到。）

## 9.6 内部谓词与外部谓词

我们已经知道，每个关系变量有一个关系变量谓词，并且整个数据库有一个数据库谓词。当然，当前讨论的谓词都是“系统能够理解的谓词”：它们被正式地声明（事实上，它们是数据库定义的一部分），并且它们也被系统强制执行。基于这些原因，为了简便，有时候把当前我们讨论的谓词具体称为**内部谓词**——主要是因为关系变量和数据库也都有外部**谓词**，这一点我们将继续讨论。[⊖]

首先，最重要的一点是，内部谓词具有正式的形式，外部谓词只是一种不正式的形式。不严格地讲，内部谓词描述了数据对于系统的意义；相反，外部谓词描述了数据对于用户的意义。当然，用户除了需要理解外部谓词，也要理解内部谓词，但是，系统能且只能理解内部谓词。事实上，不严格地说，一个给定的内部谓词是系统对于相应的外部谓词的大致理解。

让我们具体来看关系变量。正如上面所说，给定关系变量的外部谓词是基本上描述了关系变量对于用户的含义。例如，对于供应商关系变量 S，外部谓词可以作如下描述：

*供应商 S 具有名字 SNAME，状态 STATUS，位于城市 CITY。而且，状态值在 1 到 100（含 1 和 100）之间，如果在伦敦即为 20。任意两个不同的供应商都具有不同的供应商号。*

为了讨论方便，我们将用下面简单的描述代替谓词：

*供应商 S#名为 SNAME，具有状态 STATUS，位于城市 CITY。*

---

⊖ 在这里我们的确需要严格一些，但更准确地描述需要部分依赖于我们正在处理的特殊的语言。为了当前的目的，我们认为约束必须在每一个自身不在语法上内嵌其他声明的声明的末尾被满足。或不太严格地讲，约束必须在分号的地方被满足。

⊖ 第 3 章和第 6 章讨论过，但只称其为谓词。事实上，我们在整本书中都使用术语“谓词”来特别表示外部谓词。只有一个例外，是在第 7 章讨论选择操作的时候，我们说选择的条件是一个谓词，但它不是一个外部谓词。

（毕竟，外部谓词只是一个非正式的描述，因此我们尽可能简单地描述它。）

注意，下面的描述是一个谓词：它具有四个参数（S#, SANME, STATUS, CITY），分别对应变量的四个属性，[⊖] 当赋予变量具体的值时，我们将得到相应的命题（分为真、假两类）。因此，每一个记录都可以被当作是一个命题。有一点非常重要，一些特殊的命题（即元组 S 的当前值）为真。例如，如果元组

```
{ S# S#('S1'), SNAME NAME('Smith'), STATUS 20, CITY 'London' }
```

在某一给定时间内出现在关系变量 S 中，我们则认为该命题正确，即确实存在一个供应商，其供应商号为 S1，名字为 Smith，状态 20，位于伦敦。也就是说：

```
IF ( s ∈ S ) = TRUE THEN XPS ( s ) = TRUE
```

这里：

- S 是一个如下形式的元组：{ S# *s#*, SNAME *sn*, STATUS *st*, CITY *sc* }
  （其中 *s#* 是 S# 的一个值，*sn* 是 NAME 类型的值，*st* 是整型值，*sc* 是 CITY 类型值。）
- *XPS* 是供应商的一个外部谓词。
- *XPS*(*s*) 是用 S# = *s#*, SNAME = *sn*, STATUS = *st* 和 CITY = *sc* 初始化得到的命题。

然而，如第 6 章所提，我们将进一步研究外部而非内部谓词。具体来说，我们采用**封闭世界假设**，它表明如果一个有效元组在某个给定时间没有出现在关系变量中，那么对应的命题将被按习惯理解为在那个时间是错误的命题。例如，如果元组

```
{ S# S#('S6'), SNAME NAME('Lopez'), STATUS 30, CITY 'Madrid' }
```

在某个给定时间没有出现在关系变量 S 中，那么我们将理解为，在那个时间不存在与供应商号 S6、名字叫 Lopez、状态号是 30、位于 Madrid 的供应商签订合同的事实。更一般地讲：

```
IF ( s ∈ S ) = FALSE THEN XPS ( s ) = FALSE
```

或更简捷：

```
IF NOT ( s ∈ S ) THEN NOT XPS ( s )
```

把以上两者合二为一，我们有：

```
s ∈ S ≡ XPS ( s )
```

换言之，一个给定的元组在给定时间出现在给定关系变量中的充要条件是：那个元组使得那时的关系变量的外部谓词取真值。因此，一个给定的关系变量包含全部并且仅包含与那个关系变量的外部谓词的真值实例对应的元组。

## 9.7　正确性与一致性

根据定义，系统无法（事实上也不能）理解通过实例化哪些谓词而得到的外部谓词和命题。例如，系统不会知道一个供应商“位于”某地的含义，也不知道一个“供应商”拥有“状态号”（等等）。所有类似这样的问题都属于解释的问题——它们对于用户是有意义的，对系统而言却不然。给一个更具体的例子，如果供应商的状态号是 S1 并且与城市名 London 恰巧一起出现在某个元组中，那么用户可以理解为供应商 S1 位于 London，[⊖] 但系统是无法作类似的理解的。

更重要的是，即使系统能够理解一个供应商位于某地的含义，它仍无法理解用户是否可以正

---

⊖ 此处使用的术语参数与 9.3 节和 9.4 节使用的参数在意义上有一点不同，后者表示整个关系，而现在表示单个属性值。

⊖ 或者供应商 S1 曾经位于 London，或者供应商 S1 在 London 有一间办公室，或者供应商 S1 在 London 没有办公室，或者其他任何可能的解释之一（这种解释是无穷多的，且解释对应于无限数量的可能的外部谓词）。

确使用。如果用户向系统声明 S1 位于 London（通常通过执行 INSERT 语句实现），系统无法知道那个声明是否为真。系统所能做的只是确保它不会导致声明违规（即它不会导致任何内部谓词取值 FALSE）。假如不是这样，那么系统必须接受声明，并且从那一点往后把它视为真值（至少直到用户告知系统它已不再取真值了（通常通过执行 DELETE 语句实现））。

顺便提及，前面的内容清楚地解释了为什么封闭世界假设不适用于内部谓词。具体来说，对于一个关系变量而言，一个元组可能满足内部谓词但却没有出现在那个关系变量中，因为在现实世界中没有与它对应的真命题。

我们通过不太正式的说法来总结一下以上的讨论：一个给定关系变量的外部谓词是那个关系变量的**有意的解释**（intended interpretation）。因此，它对于用户而非系统而言是重要的。同时，一个给定关系变量的外部谓词是**接受更新的标准**（criterion for acceptability of updates），即至少从原则上讲，它决定 INSERT、DELETE 或 UPDATE 是否能成功。因此，理想情况下，系统知道每个关系变量的外部谓词，因此它能正确处理所有可能的更新该关系变量的企图。然而，正如我们看到的，这个目标是无法实现的；系统无法知道任何给定关系变量的外部谓词。但它知道一个较好的近似：它知道对应的内部谓词。于是，实际接受更新标准的是内部谓词，而不是外部谓词。换一种说法是：

*系统无法保证数据的真实性，只能保持一致性*

也就是说，系统无法保证数据库只包含真命题，它所能做的仅是保证数据库不包含任何破坏完整性约束的命题，即保证数据库的一致性。糟糕的是真实性与一致性是有区别的，事实上，我们可以观察到：

- 如果数据库仅包含真命题，那么它一致的，反之却不一定。
- 如果数据库是不一致的，那么它至少包含一个假命题，反之却不一定。

简单地说，正确意味着一致（而不是相反），不一致意味着不正确（而不是相反）。正确，指的是当且仅当数据能够完全反应现实世界事务的真实状态时，它是正确的。

## 9.8 完整性和视图

截至目前，本章的所有讨论适用于一般的关系变量，而不只是基本的；特别是，它们适用于视图（虚拟关系变量）。于是，视图也受约束限制，它们有关系变量谓词，包括内部的和外部的。例如，假定我们通过在供应商关系变量的 S#、SNAME 和 STATUS 属性上投影定义了一个视图（因此有效地消除了属性 CITY）。于是那个视图的外部谓词看起来像这样：

*存在一个名叫 CITY 的城市，根据合同有 S#这样的供应商，供应商名为 SNAME 它位于城市 CITY，具有状态 STATUS。*

注意到，这个谓词必须有 3 个参数（而不是 4 个）对应于视图的 3 个属性（CITY 目前不再是一个参数而是一个约束变量，由于它被词组“存在某个城市”量化）。另一种更为清晰的解释方式是，如上所述的谓词逻辑上等价于以下形式：

*供应商 S#名为 SNAME，具有状态 STATUS，位于某城市。*

这种谓词形式很清楚地拥有恰好 3 个参数。那么，关于内部谓词又如何呢？这里再次列举我们给出的常见的例子：

1）每个供应商的状态值都在 1 到 100 的范围内取值（含 1 和 100）。
2）每个伦敦供应商的状态号都是 20。
3）如果有一些零件的话，它们中至少有一个是蓝色的。
4）不允许出现两个不同的供应商拥有相同的供应商号。
5）每次供货都要有一个存在的供应商。
6）不允许状态号小于 20 的供应商供应任何数量多于 500 的零件。

假定我们正在讨论的视图叫 SST。那么就视图 SST 而言，例子 3 很明显是无关的，因为它与零件有关，而与供应商无关。其他的例子均适用于视图 SST，只是形式上稍作改动。比如，下面是例 5 改动后的形式：

```
FORALL s# ∈ S#, p# ∈ P#, q ∈ QTY
  ( IF ( S# s#, P# p#, QTY q ) ∈ SP
    THEN EXISTS sn ∈ NAME, st ∈ INTEGER
           ( ( S# s#, SNAME sn, STATUS st ) ∈ SST ) )
```

改变发生在第3和第4行：所有指向CITY的引用均已取消，并且指向S的引用已被替换为指向SST的引用。注意，我们把这个对SST的约束视为从相应的对S的约束中得到，正如关系变量SST本身是从关系变量S得到一样。[⊖]（又如SST的外部谓词是从S的外部谓词得到，也是一个道理。）

类似的评价可直接应用到例1，2，4和6上。然而，例2稍有些复杂，因为它包括一个EXISTS，对应于已通过投影去掉的属性：

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER
  ( IF ( S# s#, SNAME sn, STATUS st ) ∈ SST
    THEN EXISTS sc ∈ CHAR
      ( ( S# s#, SNAME sn, STATUS st, CITY sc ) ∈ S AND
        ( IF sc = 'London'
          THEN st = 20 ) )
```

然而，我们可以再次把这个约束视为从相应的对S的约束中获得。

## 9.9　约束分类模式

在本节中，我们为约束简要地描述一个分类模式（本质上是文献［3.3］中采用的模式）。我们把约束简要地分为4个大类：数据库、关系变量、属性和类型约束。概要如下

- 数据库约束是在一个给定数据库允许取值上的约束。
- 关系变量约束是在一个给定关系变量允许取值上的约束。
- 属性约束是在一个给定属性允许取值上的约束。
- 类型约束是组成一个给定类型的值集的一个定义。

然而，以相反的顺序来解释它们更符合我们的目的。

### 1. 类型约束

在此之前，本章中从未提及类型约束。然而，它们在第5章已详细讨论，因此以下内容只是用来提醒你我们已介绍过的内容。首先，精确地讲，一个类型约束是组成当前的类型的值的详细说明。这里举个例子（与第5章重复）：

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

含义：类型WEIGHT的合法值严格来说，就是那些可能被表示为5位十进制数字和小数点后1位数字的值，当前讨论的十进制数大于0小于5000。

目前可明显地看出，最终，任何表达式能产生类型WEIGHT的值的唯一方式是调用WEIGHT选择子。因此，这样的表达式违反WEIGHT类型约束的唯一方式，就是选择子调用违反WEIGHT类型约束。接下来，类型约束总能被认为（至少是概念上）在某个选择子调用执行期间被检查。例如，考虑下面对类型WEIGHT的选择子调用：

```
WEIGHT ( 7500.0 )
```

这个表达式将引起运行时异常（违反WEIGHT类型约束：值出界）。

由上述我们可以说，类型约束总被立刻检查。因此，特别地，没有一个关系变量能在不恰当的类型（当然是一个支持类型约束的系统）的任何元组的任何属性上获得一个值。

因为类型约束本质上就是一个组成某个当前类型的值的规定，在**Tutorial D**中我们把这样的约束和合适的类型绑在一起，并且通过适当的类型名来辨别它们，因此，一个类型约束可通过删除类型本身来删除。

---

⊖ 注意，演绎约束是从一个基本关系变量到一个视图的有效的外码约束。参见9.10节。

### 2. 属性约束

属性约束基本上是9.2节中说的在前约束；换言之，一个属性约束基本上就是一个对特定关系变量的某个属性是某个特定类型的声明。例如，再次考虑供应商关系变量定义

```
VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR } ... ;
```

在这个关系变量中，属性S#、SNAME、STATUS和CITY分别被约束为类型S#、NAME、INTEGER和CHAR。换言之，属性约束是当前的属性定义的一部分，并且它们可以通过对应的属性名标识。因此，一个属性约束可以通过删除属性本身（实际上通常指删除包含的关系变量）加以删除。注意：原则上，任何引入一个属性值到数据库（通过INSERT或UPDATE操作）的尝试都将被拒绝。然而，实际上，这样的情况永远不会发生，只要系统强制执行前面小节描述的类型约束。

### 3. 关系变量和数据库约束

关系变量和数据库约束是到目前为止我们自始至终关注的，它们之间的不同之处是关系变量约束仅仅包括一个关系变量，而数据库约束包括两个或多个关系变量。然而，在9.2节已提到，从理论角度看这个差别不那么重要（尽管从实际出发可能有用）。

然而，一个至今我们仍未谈及的要点是，某个关系变量或数据库约束可以是**转变约束**（transition constraint）。一个转变约束是在一个给定变量（特别地，一个给定关系变量或一个给定数据库）从一个值变到另一个值的合法转变上的约束；[⊖] 例如，一个人的婚姻状况可以从“未婚”变到“已婚”，但不能反过来。假定我们在一个表达式中有方式指出：

a）在一个任意的更新前当前变量的值。

b）同一个变量在更新后的值，那么我们有办法形式化任何期望的转变约束。例如（供应商的状态不应该下降）：

```
CONSTRAINT TRC1
    FORALL SX' FORALL SX ( SX'.S# ≠ SX.S# OR
                           SX'.STATUS ≤ SX.STATUS ) ;
```

解释：我们引入如下约定，一个主要的范围变量名（如例子中的SX'）被理解为指相应的关系变量，因为它优先于当前的更新。于是，本例中的约束可视为：如果SX'是一个更新前的供应商元组，那么更新后不存在一个供应商元组SX，其供应商号与SX'相同且状态值比在SX'中的小。

注意，约束TRC1是一个关系变量转变约束（它适用于单一关系变量，关系变量S）。这里形成对比的是一个数据库转变约束（“来自所有供应商的任何给定零件的总量应该永远不减少”）：

```
CONSTRAINT TRC2
    FORALL PX
       SUM ( SPX' WHERE SPX'.P# = PX.P#, QTY ) ≤
       SUM ( SPX  WHERE SPX .P# = PX.P#, QTY ) ;
```

## 9.10 码

正如在9.1节中提到的，关系模型总是强调码的概念，尽管它们只是一个普遍现象的特例（虽然实际上是重要的）。在本节中，我们特别考虑码。

### 1. 候选码

设$R$为一个关系变量，由定义，$R$的属性集具有唯一性（uniqueness），也就是说，在任一时刻，没有两个元组是重复的。实际上，$R$的属性集合的某个真子集也具有唯一性；比如在供应商关系变量S中，S#就具有此特性。这样的考虑提供了如下的定义：[⊜]

---

⊖ 不是转变约束的约束有时称为状态（state）约束。

⊜ 注意这个定义只适用于关系变量；也可为关系值定义一个类似的概念[3.3]，但关系变量是重要的情况。也要注意，我们再次依赖元组相等的概念（特指在唯一属性的定义中）。

- 假定 $K$ 是关系变量 $R$ 的属性集合，那么 $K$ 可以称之为 $R$ 的**候选码**，当且仅当 $K$ 具有如下两个特点：
  a）**唯一性**：$R$ 上没有两个不同的元组，在 $K$ 上有相同的值。
  b）**不可约性**：$K$ 没有一个真子集也具有唯一性。

注意，每一个关系变量至少有一个候选码。唯一性是无需解释的；关于不可约性，如果指定了不具有不可约性的候选码，系统将无法得知事件的真实状态，于是不能正确实施相应的完整性约束。比如，假设定义 {S#, CITY} 而不是 {S#} 为候选码，系统将无法实施供应商编号在全数据库内是唯一的这一“全局”约束，它只能实施较弱的“在城市内唯一”的“局部”约束。因此，候选码不应包括任何与唯一识别的目的无关的其他属性。[○]

顺便提一下，前面讲的不可约性在字面上的意思是最小性（minimality）（包括本书的前几版也是如此）。但最小性也不是一个准确的词语，因为如果说候选码 $K1$ 是最小化的，并不意味着无法找到另一个候选码 $K2$，而 $K2$ 包含更少的元素；有可能 $K1$ 包含四个元素而 $K2$ 仅有两个。所以还用“不可约性”这个词。

在 **Tutorial D** 中，在关系变量定义中用如下语法：

```
KEY { <attribute name commalist> }
```

指定这个关系变量的候选码。下面有一些例子：

```
VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR }
    KEY { S# } ;
```

注意：在前几章中，在定义中带有“PRIMARY KEY”子句，而不是用未限定的“KEY”子句。请参考随后的“主码和替换码”一小节，有更多的讨论。

```
VAR SP BASE RELATION
     { S#    S#,
       P#    P#,
       QTY   QTY }
     KEY { S#, P# } ... ;
```

这个例子显示了一个带有**复合**候选码的关系变量（也就是候选码涉及一个以上的属性）。与此相对的是**简单**候选码。

```
VAR ELEMENT BASE RELATION { NAME    NAME,
                            SYMBOL  CHAR,
                            ATOMIC# INTEGER }
                      KEY { NAME }
                      KEY { SYMBOL }
                      KEY { ATOMIC# } ;
```

这个例子显示了一个同时有几个不同的（简单）候选码的情况。

```
VAR MARRIAGE BASE RELATION { HUSBAND                NAME,
                             WIFE                   NAME,
                             DATE /* of marriage */ DATE }
/* assume no polyandry, no polygyny, and no husband and */
/* wife marry each other more than once ...             */
                   KEY { HUSBAND, DATE }
                   KEY { DATE, WIFE }
                   KEY { WIFE, HUSBAND } ;
```

这个例子显示了有多个复合（交错的）候选码的情况。

当然，如在 9.2 节中指出的，候选码定义只是关系变量约束的缩写形式，因为从视图的

---

○ 候选码不可约的另外一个原因是：任何一个依赖于可约候选码的外码（如果存在这种情况的话），它也是“可约”的，因此包含它的关系变量将违背第 12 章中进一步规范化的原则。

语法角度来看，候选码的概念是重要的，所以这种缩写很有用。尤其是候选码提供了一种在关系模型中的**元组级的寻址方式**——系统中唯一精确指示某些指定元组的方法就是通过候选码。比如：

```
S WHERE S# = S# ('S3')
```

可以保证至多产生一个元组（更准确地说，它产生了至多包含一个元组的一个关系）。相反，表达式：

```
S WHERE CITY = 'Paris'
```

产生数目不可预测的多个元组。可以说，候选码对于关系系统的正确操作的重要性，可以和内存地址对于计算机正确操作的重要性相提并论。结论是：

1）没有候选码的关系变量——也就是允许重复元组的关系变量——在某些环境中会出现异常现象。

2）不支持候选码概念的系统有时会表现得像一个非关系系统，即使它所处理的关系变量是真正的关系变量，而且不允许重复元组。

上面所说的“异常”和“像非关系系统”的现象与视图的更新和优化有关（分别参见第10章和第18章）。

再指出几点，然后结束这一小节的讨论：

- 不只是基本关系变量拥有候选码！——所有的关系变量都有，包括视图。然而在视图情况下，这样的码是否被声明将部分依赖于系统是否知道如何执行**候选码推断**［3.3］。
- 候选码的超集是**超码**（superkey）。属性集｛S#，CITY｝是关系变量S的超码。超码具有唯一性，但不具有不可约性。当然，候选码可以看作是超码的一个特例。
- 如果*SK*是关系变量*R*的超码，*A*是*R*的一个属性，那么**函数依赖** $SK \rightarrow A$ 在*R*中为真。实际上，可以定义*SK*上的一个子集为超码，而使函数依赖 $SK \rightarrow A$ 对*R*上的任意属性*A*为真。注意，函数依赖这一重要概念在第11章再深入讨论。
- 最后，请注意不要把逻辑上的概念“候选码”和物理中的概念“唯一索引”相混淆（虽然后者常被用来实施前者）。换言之，候选码上不一定要有索引（或任何其他的物理存取路径）。实际上可能会有这样的存取路径，但有还是没有并不在关系模型的讨论范围之内。

**2. 主码和替换码**

很明显，一个关系变量可以有两个或多个候选码。这种情况下，此关系模型就要求——至少是对于基本关系变量——它们中的一个被选为**主码**，其他的就被称为**替换码**（alternate key）。如在“元素”例子中，我们选取｛SYMBOL｝作为主码；｛NAME｝和｛ATOMIC#｝就是替换码。在只有一个候选码的情况下，对于基本关系变量关系模型，要求这个候选码被指定为主码，因此，基本关系变量总是有一个主码。

选取一个候选码作为主码（当存在选择时），在多数情况下是一个好主意，但并不完全。在参考文献［9.14］中有这方面的详细讨论；这里，我们只说明一种情况，在这种情况中哪一个候选码被选为主码是任意的（引用Codd的［9.9］中的话“以简便性为选取原则，但这是在关系模型研究之外的。”）在本书中，有时会定义主码，有时不会（当然，我们总是会定义一个候选码）。

**3. 外码**

简单地说，外码就是定义在关系变量*R2*上的一组属性集，它们的值要求与关系变量*R1*的候选码的值相匹配。比如，考虑在关系变量SP中的属性集｛S#｝（仅包含一个属性）。显然，如果一个｛S#｝的值要在关系变量SP中出现，仅当同样的值出现在关系变量S的唯一候选码｛S#｝中（不能让不存在的供应商有一个供货量）。相应地，一个对｛P#｝的给定的值要能在关系变量SP中出现，仅当同样的值出现在关系变量P的唯一候选码｛P#｝中（不能让不存在的零

件有一个供货量）。这些例子可用于引出如下定义：[⊖]

- 设 *R2* 是关系变量。那么 *R2* 的**外码**是它的一个属性集，比如说 *FK*，则：

  a）存在带有候选码 *CK* 的关系变量 *R1*（*R1*，*R2* 可能相同）。

  b）有可能重命名某个 *FK* 的属性的子集，以便使 *FK* 成为 *FK'*（比如说）且 *FK'* 和 *CK* 是相同的（元组）类型。

  c）无论何时，*R2* 中的任一个 *FK* 值都在 *R1* 的 *CK* 中有一个相同的值。

这里有如下要点：

1）实际中，很少有必要执行任何实际的重命名；即需要重命名的 *FK* 的属性的子集通常是空集（不是空集的例子在第 7 点中讨论）。因此，为简单起见，从这点往后我们假定 *FK* 和 *FK'* 是相似的，除非有明显相反的声明。

2）注意到，虽然 *FK* 的每个值必须作为 *CK* 的一个值出现，但反过来却不是必须的；即 *R1* 可能包含一个 *CK* 值，该值当前不作为 *FK* 值在 *R2* 中出现。例如在供应商和零件例子中（样内值如内封底所示）供应商号码 S5 出现在关系变量 S 中但没有出现在关系变量 SP 中，因为供应商 S5 当前没有任何零件。

3）外码是简单的还是复合的取决于对应的候选码是简单的还是复合的。

4）外码表示一个对包含相应候选码的某个元组的**参照**（**被参照元组**）。确保数据库中不含有无效外码的问题被称为**参照完整性**问题。（见第 12 条。）外码值必须与相对应的候选码的值相匹配的约束被称为**参照约束**。含有外码的关系变量被称为**参照**关系变量，含有相应候选码的关系变量称为**被参照**关系变量。

5）参照图：考虑“供应商 - 零件”例子。可以把其中的参照约束用下面的**参照图**表示出来：

```
S ⟵ SP ⟶ P
```

每一个箭头表示一个外码，这个外码从此关系变量出发，参照箭头指向的关系变量中的某个候选码。注意：为清晰起见，用构成候选码的属性名来标识参照表是可行的。[⊜] 比如：

```
      S#    P#
S ⟵ SP ⟶ P
```

在本书中，只有当省略标识会引起混淆的情况下才进行标识。

6）某个关系变量可以是参照关系变量，也可以是被参照关系变量。如在 R2 的例子中：

```
R3 ⟶ R2 ⟶ R1
```

为此，引入**参照路径**（referential path）这一术语。设有关系 R*n*，R（*n* - 1），…，R2，R1，存在从 R*n* 到 R（*n* - 1）的参照约束，从 R（*n* - 1）到 R（*n* - 2）的参照约束，……，从 R2 到 R1 的参照约束：

```
Rn ⟶ R(n-1) ⟶ R(n-2) ⟶ ... ⟶ R2 ⟶ R1
```

那么，从 R*n* 到 R1 的箭头链表示从 R*n* 到 R1 的**参照路径**。

7）注意：外码定义中的 R2 与 R1 不必是不同的。也就是说，外码可以参照关系变量自身的候选码。考虑下面的外码定义（一会儿我们再解释语法，不过这应该是显而易见的）：

```
VAR EMP BASE RELATION
          { EMP# EMP#, ..., MGR_EMP# EMP#, ... }
    KEY { EMP# }
    FOREIGN KEY { RENAME MGR_EMP# AS EMP# } REFERENCES EMP ;
```

这里，MGR_EMP#代表此雇员（用 EMP#标识）的经理的雇员号；比如，雇员 E4 这个元

---

⊖ 注意：定义再次依赖于元组等价的概念。

⊜ 也可以给外码起名，然后用这些名字来标识箭头。

组，它的 MGR_EMP#值为 E3，它表示指向雇员号为 E3 的元组的参照。（1 中已经提到，这里要给外码中的某个属性重命名。）像 EMP 这样的关系变量可以称为**自参照**（self-referencing）的。习题：给这个例子增加一些数据。

8）像 EMP 这样的自参照的关系变量只是一种更一般的情形的特例——也就是，这里可以有一个**参照的环**。在关系变量 R*n*，R（*n*-1），…，R2，R1 中，当存在从 R*n* 到 R（*n*-1），从 R（*n*-1）到 R（*n*-2），……，从 R2 到 R1，从 R1 回到 Rn 的参照时，就形成了一个参照的环。简单地说，如果存在一个从关系变量 R*n* 回到它自身的参照路径，这里就有一个参照的环。

```
Rn ⟶ R(n-1) ⟶ R(n-2) ⟶ ... ⟶ R2 ⟶ R1 ⟶ Rn
```

9）“外码-候选码”的匹配可以比作是使数据库结合在一起的粘合剂。它的另一个说法是：它们的匹配表示了元组之间的某种关系。仔细观察，并不是所有的关系都通过码表现出来。比如，在供应商和零件之间有“地点相同”的关系，由 S 和 P 的城市属性表现出来；当某个供应商和某种零件放在同一城市中时，就说它们有地点相同的关系，但这并不通过码之间的关系表现出来。

10）由于历史原因，外码只定义给基本关系变量，这一情况引起了一些问题。（参见第 10 章 10.2 节关于互换性准则的讨论。）这里不实施这样的限制；但是为简单起见，我们只把讨论限于基本关系变量（这里有些区别）。

11）关系模型起初是要求外码参照的，参照码不仅是候选码，还要主码（参见［9.9］）。由于这是不必要的，后来放弃了这些限制，虽然实践中它是一种好习惯［9.14］。

12）除了外码的概念，关系模型还包括下列规则（*参照完整性规则*）：

- **参照完整性**：数据库不能含有无匹配的外码。[⊖]

术语“未匹配外码”表示在被参照表中不含有相应的外码；换句话说，约束只是简单地要求：如果 *B* 参照 *A*，则 *A* 必须存在。

下面是定义外码的语法：

```
FOREIGN KEY { <item commalist> } REFERENCES <relvar name>
```

这一句出现在参照关系变量定义中。

*注意*，<*关系变量名*>确定了一个引用的关系变量，每个<*项目*>要么是一个引用的关系变量的<*属性名*>，要么是如下形式的一个表达式：

```
RENAME <attribute name> AS <attribute name>
```

在前面已经给出了很多例子（如第 3 章中的图 3-9 所示）。*注意*：正如 9.2 节中指出的，外码定义只是一些数据库约束（在自参照的关系变量中，是关系变量约束）的缩写——除非外码定义被扩展到包括了某些“参照行为”（referential action），那时它就不只是完整性约束了。参见下面的“参照行为”部分。

**4. 参照行为**

考虑下面的语句：

```
DELETE S WHERE S# = S# ('S1') ;
```

假设 DELETE 能像所描述的那样做，也就是它删除供应商元组 S1。假设 S1 对应一些发货元组，并且程序没有删除这些元组，当系统检查从发货到供应商的参照完整性时，就会发生错误。

然而，替代的方法是存在的，在某些情况下它可能更合适。那就是让系统进行一次适当的补偿性操作，以保证全部结果满足约束。在此例中，显然补偿的操作是同时自动删除供应商 S1 的“发货”元组。为达到这一目的，将外码定义扩展如下：

---

⊖ 参照完整性可以被看做是一个元约束；意思是给定数据库必然服从一定的完整性约束，特别对现在讨论的数据库，在数据库中不可以违反这些准则。顺便提一下，数据库也经常含有另外一个元约束：实体完整性，它与空值有关。在第 19 章讨论这个问题。

```
VAR SP BASE RELATION { ... } ...
    FOREIGN KEY { S# } REFERENCES S
                      ON DELETE CASCADE ;
```

ON DELETE CASCADE 定义了指定外码的删除规则，CASCADE 是此规则的参照行为。这些说明的意思是：在供应商关系变量上的删除操作应“级联”（cascade）删除相匹配的发货元组。

另一个常见的参照行为是 RESTRICT（与关系代数中的 restrict 操作符没有关系）。这个例子中，RESTRICT 表示：只当不存在相应的元组时才进行删除（否则被拒绝）。在外码定义中忽略参照行为相当于指定行为是“没有动作”（NO ACTION），意思是 DELETE 只按所描述的去做，不多做也不少做。（当指定为 NO ACTION 时，删除了一个有匹配的发货元组的供应商元组，就会引起违反参照完整性。）几个要点：

1）删除不是唯一需要参照行为的操作。比如，当存在相匹配的发货元组时对供应商编号更新。显然，和删除规则一样，需要一个更新规则。通常，DELETE 和 UPDATE 是相同的：

- CASCADE：在对应的发货元组上级联更新。
- RESTRICT：被限制为只当没有相匹配的发货元组时才可更新（否则被拒绝）。
- NO ACTION：更新只按所描述的那样操作（但是随后可能出现违反参照完整性的情况）。

2）当然，并不是只有 CASCADE、RESTRICT 和 NO ACTION 三种参照行为，它们只是常用的三种。原则上，可以有多种不同的响应，比如对某个供应商元组的删除，有：

- 由于某些原因，该操作被拒绝。
- 信息被写到文档数据库。
- 与被删除元组相对应的发货元组被转移到其他的供应商元组中。

为所有可得到的响应提供声明语法是不可行的。一般来说，特定一个由任意用户定义过程组成的参照动作是可能的（见下一节）。进一步地：（a）过程的执行必须被看做是声明执行的一部分，而这个声明会引发完整性检查；（b）当过程已经执行后，必须再次执行完整性检查。（过程显然不该让数据库违反约束）。

3）设 R2、R1 分别为参照关系变量和被参照关系变量：

$$R2 \longrightarrow R1$$

设使用的删除规则为 CASCADE。通常情况下，在 R1 上删除某个元组也对应着在 R2 上删除某些相应的元组。假设 R2 相应地被另一个关系变量 R3 所参照：

$$R3 \longrightarrow R2 \longrightarrow R1$$

那么蕴含的对 R2 的删除就好像在 R2 上直接删除一样；也就是说它也遵守定义在其参照约束上的删除规则。如果蕴含删除失败（由于 R3 到 R2 的删除规则或其他原因），那么整个操作失败，数据库恢复原状……，如此递归地进行，可到达任意层次。

如果 R2 的外码中有被 R3 的候选码所参照的属性，则相似的方法也适用于级联更新（CASCADE UPDATE）。

4）从前面所讲的可知，从视图的逻辑观点来看，数据库更新是原子性的（atomic）——要么全做要么全部不做。即使是在多个关系变量上的多个更新操作，也会因 CASCADE 的参照行为而实现原子性。

## 9.11　触发器

从我们本章所述的所有内容中可以很清楚地看到我们特别关注声明完整性支持。尽管在近些年来情况已经被改进。但事实上即使有也是很少的产品，当它们第一次出现时提供很多这种支持。因此，完整性约束经常用**触发过程**程序地执行。触发过程也是预编译过程，它被存储在数据库里，一旦情况发生就被自动调用。例如，我们通过触发过程执行例 1（status 值在 1 ~ 100 之内）：（a）一旦有元组插入关系变量 s 时，触发过程被调用；（b）检查新插入元组；（c）如果状态值超出了范围，把它删除。基于它们在实际中的重要性，这部分我们简单地了解了触发过程。

注意：

1）因为它们是过程，为了履行完整性约束而触发过程不是一个推荐的方法。过程对人们来讲很难理解，对系统来讲又很难优化，并且注意，声明约束在所有关系修改中被检查，而触发过程仅当特殊事件（如插入元组到关系变量 s）发生时执行。[⊖]

2）触发程序的应用不只局限于本章所讲的完整性问题。事实上，如第 1 点所述，它们能服务于其他有用的目的，这是它们真正的目的所在，其他有用目的的例子包括：

a. 如果发生故障就提醒用户（例如，如果其他部分的现有数量低于危险水平就发出警告）。

b. 排除故障（例如管理对指定变量的参照，说明指定变量的变化）。

c. 审计（例如追踪执行者、修改了什么、对哪个表、什么时间）。

d. 执行测量（例如计时或追踪特定数据库事件）。

e. 开展补充操作（例如因 supplier 元组的级联删除，而删除相应的 shipment 元组）。[⊖]

等等。

考虑下面的例子（这个例子基于 SQL，而不是 **Tutorial D**，因为参考文献 [3.3] 没有规定也没有禁止任何触发过程支持。事实上，它基于商品产品，而不是 SQL 标准，因为 SQL 不支持所说的特定特征。）假设我们有一个叫做 LONDON_SUPPLIER 的视图，如下定义：

```
CREATE VIEW LONDON_SUPPLIER
    AS SELECT S#, SNAME, STATUS
       FROM   S
       WHERE  CITY = 'London' ;
```

正常地，如果用户想要往视图中插入一行，SQL 将在下面的拥有属性 CITY 的表 S 中插入一行，无论默认值是不是 CITY 列。（参见第 10 章）假设默认值不是 London，结果就是新行将不会出现在这个视图中。创建一个如下的触发过程：

```
CREATE TRIGGER LONDON_SUPPLIER_INSERT
       INSTEAD OF INSERT ON LONDON_SUPPLIER
       REFERENCING NEW ROW AS R
       FOR EACH ROW
       INSERT INTO S ( S#, SNAME, STATUS, CITY )
              VALUES ( R.S#, R.SNAME, R.STATUS, 'London' ) ;
```

插入一行到这个视图将会导致 CITY 属性值等于 LONDON 的一行插入到下面的基本表中。（需要的话，新行将会插入到视图中）

这有一些来源于这个例子的观点。注意：尽管我们的例子是基于 SQL 的，但不意味着它们是 SQL 特有的（SQL 特例将在下一节给出。）

1）一般的，CREATE TRIGGER 在其他的语句中特指一个事件、一个条件或一个动作：

- **事件**是数据库里的一个操作（例如 INSERT ON LONDON_SUPPLIER）。
- **条件**是一个布尔表达式，当它的值为 TURE 时，执行动作（如果没有显式指明条件，默认值就是 TURE）。
- **动作**是一个触发程序（例如“INSERT INTO S…”）。

事件和条件在一起有时被称作触发事件。事件、条件、动作三者一块被称作**触发器**。显然，触发器被认为是 *EVENT-CONDITION-ACTION RULES*（简称 ECA 规则）。

2）可能的事件包括 INSERT、DELETE、UPDATE（可能特指属性）、到达交易结束（COMMIT）、到达一天的特定时间、超过特定缺蚀时间、违反特定约束等。

3）大体上，动作能在特殊事件之前（BEFORE）、之后（AFTER）、替代（INSTEAD OF）执行。

---

⊖ 注意到声明约束规定并不说明 DBMS 什么时候进行完整性检查，对我们来说也没必要规定：第一，因为如果他们这么做了，他们需要在用户声明约束上做额外的工作；第二，因为用户也许会做错。甚至，我们想让系统自己决定什么时候做检查（见参考文献 [9.5] 的注解）。

⊖ 事实上，级联删除是触发过程的一个简单的例子。注意，但是，它被声明定义。我们并不意味着建议参照动作是一个坏主意，仅仅因为它们被触发了。

4）一般地，动作能对每一行（FOR EACH ROW）或每一句话（FOR EACH STATEMENT）执行。

5）一般地，无论在特殊事件发生之前还是之后动作都有一种查询数据的方法。

6）与触发器关联的数据库有时被称作**主动数据库**。

这一部分讲完了，尽管触发器显然有作用，但仍然需要小心使用它们，如果有解决问题的其他方法，就不该用触发器。下面介绍现实中触发器使用不当会引发问题的原因：

- 如果同样的事件引起一些无重复的触发器启动，这些触发器发生的序列很可能是既重要又无定义的。
- 触发器 *T1* 可能引起触发器 *T2* 启动，而 *T2* 又会引起 *T3* 启动，以此类推（触发链）。
- 触发器 *T* 也许会递归地再次触发自己执行。
- 由于触发器的出现，简单的 INSERT、DELETE、UPDATE 的作用也许与用户预料的截然不同（尤其是 INSTEAD OF，如在我们的例子中。）

把上述几点总结起来，显然，一系列的触发器是很难理解的。如果可以用声明解决，一般比程序解决要好。

## 9.12　SQL 的支持

9.9 节描述了 SQL 对约束分类模式的支持——或者在很多方面甚至缺乏支持，我们就从这里开始考虑：

- SQL 根本不支持**类型约束**，除了那些作为应用的物理表示的直接结果的基本约束。例如，我们在第 5 章看到的，我们能说出类型 WEIGHT 的值必须用小数表示，但是我们不能说那些数必须大于 0 小于 5 000。
- SQL 不支持属性约束。
- SQL 不支持**关系变量约束**。它支持**基本表约束**，但是（a）这些约束将仅仅用于这些表，不应用于全部表（尤其是不应用于视图）；（b）它们不局限于涉及一个表，还可涉及任意复合表。
- SQL 不支持**数据库约束**。它支持一般约束——**断言约束**。但是这些约束要求必须涉及两个以上的表。（事实上，除了接下来的“基本表约束”小节的最后的特例以外，SQL 的基本表约束和一般约束逻辑上是可互换的。）

SQL 不直接支持过渡约束，尽管这些约束能被触发器过程地执行。对于关系变量（或表）谓词，过渡约束也没有清楚的概念，而这些谓词是很重要的（见下一章）。

**1. 基本表约束**

SQL 基本表约束特指在 CREATE TABLE 或者 ALTER TABLE 中的约束。每个约束是一个候选码约束，或外码约束，或检查约束。我们依次讨论每一种约束。*注意*：规范 CONSTRAINT < *constraint name* > 能够被选择性地放在这些约束的定义之前，为这些约束提供了名字。为了简洁，我们忽略了选择（尽管在实践中命名所有的约束是一个好主意）。同样道理，我们也忽略一定的简写（例如定义候选码“inline”作为列定义的一部分）。

**候选码**：SQL 候选码的定义可用下面的两种形式之一

```
PRIMARY KEY ( <column name commalist> )
UNIQUE ( <column name commalist> )
```

这个 < *column name commalist* > 必须在任何情况下是非空的。[⊖] 一个基本表最多能有一个主码规定，但是可以有任意数目的唯一性规定。对于主码，每个特定列被假设成非空的，尽管没有明确指定非空（见下面检查约束的讨论）。

**外码**：外码定义形式为

---

⊖ 见习题 9.10。

```
FOREIGN KEY ( <column name commalist> )
  REFERENCES <base table name> [ ( <column name commalist> ) ]
[ ON DELETE <referential action> ]
[ ON UPDATE <referential action> ]
```

< *referential action* > 可以是 NO ACTION（默认值）、CASCADE、SET DEFAULT 或 SET NULL。[⊖] SET DEFAULT 和 SET NULL 将在第 19 章中讲述。当外码所参照的候选码不是主码时，第二个 < *column name commalist* > 是必需的。注意：在以逗号分隔列的表中，外码 - 主码匹配是以列的位置“从左到右”，而不是列名为基础的。

**检查约束**：SQL 检查约束的形式

```
CHECK ( <bool exp> )
```

假定 *CC* 是基本表 *T* 的检查约束。*T* 被认为违反 *CC* 当且仅当它目前包含至少一行——看本部分的最后一节——当前的 *T* 值使 *CC* 的**布尔值**为假。注意：一般的，SQL 的**布尔值**能够任意地复杂；甚至在现有的上下文中，它们显然不局限于指基本表 *T*，而指任何数据库的可介入部分。

下面是 CREATE TABLE 的一个例子，它包含基本表约束的三种形式：

```
CREATE TABLE SP
     ( S# S# NOT NULL, P# P# NOT NULL, QTY QTY NOT NULL,
       PRIMARY KEY ( S#, P# ),
       FOREIGN KEY ( S# ) REFERENCES S
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       FOREIGN KEY ( P# ) REFERENCES P
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       CHECK ( QTY ≥ QTY ( 0 ) AND QTY ≤ QTY ( 5000 ) ) ) ;
```

我们假设 S#和 P#分别被定义成表 S 和 P 的主码，并且我们利用简写，通过简写，在列的定义中检查约束

```
CHECK ( <column name> IS NOT NULL )
```

能够被简单的 NOT NULL 表示所代替。检查约束定义可以被列定义中一个简单的 NOT NULL 描述所代替。在这个例子中，用三个 NOT NULL 代替了繁琐的检查约束定义。

最后我们再重申这一点：如果基本表是空的，那么这个 SQL 基本表约束通常被认为是满足的，即使这个约束是“1 = 2”（甚至是：“这个表必须非空!”）。

**2. 断言**

我们转向 SQL 的基本约束或称作断言约束。这个约束被 CREATE ASSERTION 方式定义，语法：

```
CREATE ASSERTION <constraint name>
       CHECK ( <bool exp> ) ;
```

删除断言的语法如下：

```
DROP ASSERTION <constraint name> ;
```

注意：与本书中所讲过的其他 SQL DROP 操作（DROP TYPE、DROP TABLE 和 DROP VIEW）的语法不同，删除断言不提供 RESTRICT 和 CASCADE 选项。

这里有 6 个来自 9.1 节的例子，以 SQL 的断言形式表达。通过练习，你也许想要尝试用基本表约束公式化这些例子。

1）每个供应商的状态号是包含在 1 到 100 范围内的。

---

⊖ < 参照动作 > 的特定支持（尤其是级联）表明，至少在覆盖下，系统能够支持某种多关系任务！——不管这样的操作被 SQL 支持与否。

```
CREATE ASSERTION SC1 CHECK
     ( NOT EXISTS ( SELECT * FROM S
                    WHERE  S.STATUS < 0
                    OR     S.STATUS > 100 ) ) ;
```

2）每名在伦敦的供应商的状态号为20。

```
CREATE ASSERTION SC2 CHECK
     ( NOT EXISTS ( SELECT * FROM S
                    WHERE  S.CITY = 'London'
                    AND    S.STATUS ≠ 20 ) ) ;
```

3）如果有一些零件，则至少有一个是蓝色的。

```
CREATE ASSERTION PC3 CHECK
     ( NOT EXISTS ( SELECT * FROM P )
       OR  EXISTS ( SELECT * FROM P
                    WHERE  P.COLOR = COLOR ('Blue') ) ) ;
```

4）不允许出现两个供应商拥有相同的供应商号码。

```
CREATE ASSERTION SC4 CHECK
     ( UNIQUE ( SELECT S.S# FROM S ) ) ;
```

UNIQUE 在这是一个 SQL 操作符它把表当做对象。如果表不包括重复行返回 TRUE，否则返回 FALSE。

5）每次发货都要有一个实际的供应商。

```
CREATE ASSERTION SSP5 CHECK
     ( NOT EXISTS
         ( SELECT * FROM SP
           WHERE  NOT EXISTS
                ( SELECT * FROM S
                  WHERE  S.S# = SP.S# ) ) ) ;
```

6）不允许出现状态号小于20的供应商提供任何数量多于500的零件。

```
CREATE ASSERTION SSP6 CHECK
       ( NOT EXISTS ( SELECT * FROM S, SP
                      WHERE  S.STATUS < 20
                      AND    S.S# = SP.S#
                      AND    SP.QTY > QTY ( 500 ) ) ) ;
```

进一步考虑下面的例子。其视图定义见9.11节：

```
CREATE VIEW LONDON_SUPPLIER
    AS SELECT S#, SNAME, STATUS
       FROM   S
       WHERE  CITY = 'London' ;
```

我们已经知道无法在该视图定义中将下面的详细说明包含进去：

```
UNIQUE ( S# )
```

然而，奇怪的是我们可以指定一个一般的限定形式，如下所示：

```
CREATE ASSERTION LSK CHECK
     ( UNIQUE ( SELECT S# FROM LONDON_SUPPLIER ) ) ;
```

### 3. 延迟的检查

SQL 的完整性约束分类方案在何时检查完整性的问题上，也与本书所讲的不同。本书所有约束是立即检查。在 SQL 中，约束可以被定义为“可延迟的”和“不可延迟的”；[○] 如果约束是可延迟的，则又可分为“最初延迟”（INITIALLY DEFERRED）和“最初立即”（INITIALLY IMMEDIATE）。它用来定义事务开始时约束的状态。不可延迟约束通常是立即执行的，但可延

○ 但是，某些约束被要求是不能延迟的。例如，如果 FK 是外码，那么匹配候选码的候选码约束必须是不延迟的。

迟约束的“可延迟”是由下面的语句动态打开或关闭的：

```
SET CONSTRAINTS <constraint name commalist> <option> ;
```

其中 < *option* > 可以是 IMMEDIATE 或 DEFERRED。下面是一个例子：

```
SET CONSTRAINTS SSP5, SSP6 DEFERRED ;
```

可延迟约束只有处于“IMMEDIATE”状态时才会被检查。设置可延迟约束到立即状态会使约束被立即检查。如果检查失败，则“SET IMMEDIATE”也失败。COMMIT 强迫所有可延迟约束为 SET IMMEDIATE 状态；如果任何完整性检查失败，则事务回滚。

**4. 触发器**

SQL 的 CREATE TRIGGER 语句如下：

```
CREATE TRIGGER <trigger name>
       <before or after> <event> ON <base table name>
     [ REFERENCING <naming commalist> ]
     [ FOR EACH <row or statement> ]
     [ WHEN ( <bool exp> ) ] <action> ;
```

解释：

1） < *before or after* > 表示是或者前或者后。（SQL 标准不支持 INSTEAD OF，但有一些产品支持。）

2） < *event* > 指 INSERT、DELETE 或者 UPDATE。UPDATE 能通过语句 OF < *column and commalist* > 授予权限。

3）每个 < *naming* > 是下列形式之一：

```
OLD ROW   AS <name>
NEW ROW   AS <name>
OLD TABLE AS <name>
NEW TABLE AS <name>
```

4） < *row or statement* > 表示 ROW 或者 STATEMENT（STAMENT 是默认值）。ROW 表示触发器会在触发语句影响每个单独的行的时候启动。STATEMENT 表示触发器为整个语句只启动一次。

5）如果一个 WHEN 语句被指定，它意味着仅当 < *bool exp* > 为 TRUE 时，< *action* > 被执行。

6） < *action* > 是一个单一的 SQL 声明。（这个单一声明是复合的，就是说，它能由一系列被 BEGIN 和 END 定界符括起来的声明组成。）

最后，这里是 DROP TRIGGER 的语法：

```
DROP TRIGGER <trigger name> ;
```

就像 DROP ASSERTION 一样，DROP TRIGGER 不提供 RESTRICT 和 CASCADE 选项。

## 9.13 小结

本章讨论了**完整性**的重要概念。完整性问题是确保数据库的数据正确的问题。（尽可能地正确，不幸的是，我们真正做到的是确保数据是一致的。）所以我们对问题的**声明**方法感兴趣。

我们首先指明完整性约束采用下面的一般形式：

IF *某些元组在特定关系变量中出现*，THEN *那些元组满足特定的条件*。

（类型约束有点不同——见后面的讨论。）我们为阐明约束给出语法，基于 **Tutorial D** 的微积分版本，并指出语法并不包括任何用户告知数据库何时检查，我们想让 DBMS 自己决定何时检查。我们声明所有的约束检查是必须是**即时的**。

接着，我们解释一个所述的约束是一个**谓词**，但是当它被检查时（即当前的关系的值被提交给谓词中提到的关系变量时），它成为**命题**。所有应用于数据库谓词的逻辑 AND 是那个关系变量的**关系变量谓词**。所有应用于给定数据库的关系变量谓词的逻辑 AND 是那个数据库的**数据**

**库谓词**。**黄金法则**说明：

没有修改操作分配给任何数据库的值使得数据库谓词的值为 FALSE。

接着，我们区分一下**内部**和**外部**谓词。内部谓词是正式的，它们被系统理解，被 DBMS 检查（前面章节提到的关系变量和数据库谓词都是内部谓词）。相反，外部谓词是非正式的，它们被用户而不是被系统理解。**封闭世界假设**应用于外部谓词而不是内部谓词。

你也许会突然意识到，在数据库上下文中通常所说的“完整性”是指**语义**。正是完整性约束（在特定的关系变量和数据库中）用来表明数据的**语义**。这就是为什么完整性**非常重要**的一个原因，正如本章介绍的那样。

我们指出，按照常理，完整性的一切处理应用于整个关系变量，而不是仅仅应用于基本表（特殊情况下，它们应用于视图）——尽管应用于视图的约束能通过应用于关系变量的约束获得，当然视图也是从关系变量获得的。

接着，我们把完整性约束分成四类：

- **类型**约束定义了指定数据类型（域）的合法值，在调用相应的选择子时被检查。
- **属性**约束定义了指定属性的合法值，任何情况不能违反（设类型约束已被检查）。
- **关系变量**约束定义了指定关系变量的合法值，仅当关系变量被更新时检查。
- **数据库**约束定义了指定数据库的合法取值，当数据库被更新时检查。

然而应指出，关系变量和数据库约束的区别不仅是逻辑上的而更是实际的。我们简单地介绍**过渡**约束。

接着我们介绍实际中非常重要的**候选码**、**主码**、**选择码**、**外码**。候选码满足**唯一性**和**不可约性**，每个关系变量必须至少有一个候选码。外码值必须与相应候选码值匹配的约束是**参照约束**。我们定义了许多参照完整性概念，包括**参照动作**的概念（尤其是**级联约束**）。后面的讨论把我们引入到**触发器**领域的分支中。

最后我们讨论了 SQL 的相关方面，现总结一下。SQL 的类型约束非常弱；尤其是类型约束有局限性，在问题中类型必须有物理表示。SQL 的基本表约束（包括对码的特殊支持）和一般约束（“断言”）包括关系变量和数据库约束的模拟（不包括过渡约束），但是，它们几乎没被仔细分类（事实上，它们几乎是相互转变的，并且为什么语言包含两者还不清楚）。SQL 也支持**延迟检查**。最后，我们简单地概括了 SQL 对触发器的支持。

## 习题

9.1　在 9.1 节的例 1 ~ 6 中，哪些操作能够导致违反约束？

9.2　给出 9.1 节中例 1 ~ 6 Tutorial D 的代数公式的表示形式。你喜欢哪个，微积分公式还是代数公式？为什么？

9.3　用 9.2 节中基于微积分的 Tutorial D 的语法，为下面供应商 - 零件 - 项目数据库的“商业规则”写一个完整性约束：

a）合法城市只有：London、Paris、Rome、Athens、Oslo、Stockholm、Madrid 和 Amsterdam。
b）合法的供应商的号码能用至少两个字符表示，以“S”开始，并且提示标明 0 ~ 999 的整数。
c）所有红色零件的重量必须小于 50 磅。
d）没有两个项目在同一城市。
e）在任何时候至多有一个供应商在雅典。
f）没有货物拥有多于平均数量二倍的数量。
g）最高状态的供应商和最低状态的供应商不能位于同一城市。
h）每个项目必须设置在至少有一个供应商的城市。
i）每个项目必须设置在至少有一个项目的供应商的城市。
j）必须至少有一个红色的零件。
k）平均供应商状态必须大于 19。
l）每个伦敦供应商必须提供零件 P2。
m）至少一个红色零件的重量小于 50 磅。
n）在伦敦的供应商必须与巴黎的供应商提供不同种类的零件。

o）伦敦的供应商必须提供比巴黎的供应商总量多的零件。

p）没有供货量减少到少于当前数目的一半。

q）在雅典的供应商只能去伦敦或巴黎，伦敦的供应商只能去巴黎。

9.4 对于习题9.3的每个回答：（a）说出这个约束是关系变量约束还是数据库约束；（b）说出能引起约束被违反的操作。

9.5 用图4-5中“供应商－零件－工程”例子中的数据，说出下列操作会引起什么结果。

a）更新工程J7，置CITY值为New York。

b）更新零件P5，置P#值为P4。

c）更新供应商S5，置S#为S8，此时的参照行为是RESTRICT。

d）删除供应商S3，此时的参照行为是CASCADE。

e）删除零件P2，此时的参照行为是RESTRICT。

f）删除工程J4，此时的参照行为是CASCADE。

g）更新供货量S1-P1-J1，置S#为S2。

h）更新供货量S5-P5-J5，置J#为J7。

i）更新供货量S5-P5-J5，置J#为J8。

j）插入供货量记录S5-P6-J7。

k）插入供货量记录S4-P7-J6。

l）插入供货量记录S1-P2-*jjj*（*jjj*表示默认工程号）。

9.6 本章的内容介绍了外码删除和修改规则，但是它没提到任何外码插入规则。为什么？

9.7 某个教育数据库包含一个公司室内培训的信息。对于每一门课程，数据库中都有它的先导课程的信息；对于每一门开设的课程，数据库中都有它的教师和学生的信息；数据库中也包含雇员的信息。如下所示：

```
COURSE      { COURSE#, TITLE }
PREREQ      { SUP_COURSE#, SUB_COURSE# }
OFFERING    { COURSE#, OFF#, OFFDATE, LOCATION }
TEACHER     { COURSE#, OFF#, EMP# }
ENROLLMENT  { COURSE#, OFF#, EMP#, GRADE }
EMPLOYEE    { EMP#, ENAME, JOB }
```

PREREQ关系变量中，SUP_COURSE#是SUB_COURSE #的先导课程，其余的都是自说明的。为这个数据库画一个参照图，并写出相应的数据库定义（即写出类型集和关系变量的定义）。

9.8 下面表示一个包含“部门”和“雇员”信息的数据库：

```
DEPT  { DEPT#, ..., MGR_EMP#, ... }
EMP   { EMP#, ..., DEPT#, ... }
```

每个部门有一个经理（MGR_EMP#）；每一雇员属于某个部门（DEPT#）。画出参照图，写出数据库定义。

9.9 下面表示一个包含“雇员”和“程序员”信息的数据库：

```
EMP   { EMP#, ..., JOB, ... }
PGMR  { EMP#, ..., LANG, ... }
```

每一程序员都是雇员，反之却不然。画出参照图，写出数据库定义。

9.10 候选码被定义成属。如果问题中的集合是空（没有属性），会发生什么？你能想象出一个空的候选码的用途吗？

9.11 假定*R*是一个*n*维关系变量。候选码*R*能拥有的最大数目是多少？

9.12 设*A*和*B*是两个关系变量，说出下面每一种情况的候选码：

a. *A* WHERE ...

b. *A* {...}

c. *A* TIMES *B*

d. *A* UNION *B*

e. *A* INTERSECT *B*

f. A MINUS B
g. A JOIN B
h. EXTEND A ADD exp AS Z
i. SUMMARIZE A PER B ADD exp AS Z
j. A SEMIJOIN B
k. A SEMIMINUS B

假设 A 和 B 满足上述的每一种操作（如在 UNION 操作下它们是同一种类型）。

9.13　重复习题 9.10，把 *candidate*（候选）用 foreign（外）代替。

9.14　用 SQL 表达式写出习题 9.3 的表示方法。

9.15　给出习题 9.7～9.9 的 SQL 数据库定义。

9.16　我们已经看到每个关系变量（实际上每个关系）与一些谓词相对应。反过来是正确的吗？

9.17　在 9.7 节的脚注中，如果值 S1 和 London 同时在元组中出现，那么可能意味着供应商 S1 在 London 没有办公室。实际上，这个特殊的情况是不太可能的。为什么？（提示：想一想封闭世界假设。）

## 参考文献

[9.1] Alexander Aiken, Joseph M. Hellerstein, and Jennifer Widom: "Static Analysis Techniques for Predicting the Behavior of Active Database Rules," *ACM TODS 20*, No. 1 (March 1995).

这篇论文继续参考文献 [9.2]、[9.5] 中的工作，讨论“专家数据库系统”（这里称为主动数据库系统）。尤其是它探讨了 IBM Starburst 原型的规则机制（参见 [18.21]、[18.48]、[26.19]、[26.23]、[26.29, 26.30] 和 [9.25]）。

[9.2] Elena Baralis and Jennifer Widom: "An Algebraic Approach to Static Analysis of Active Database Rules," *ACM TODS 25*, No. 3 (September 2000). An earlier version of this paper, "An Algebraic Approach to Rule Analysis in Expert Database Systems," appeared in Proc. 20th Int. Conf. onVery Large Data Bases, Santiago, Chile (September 1994).

标题中的“规则”主要指触发器。规则的一个问题（如 9.11 节提到的那样）是触发器的行为难以预料和理解。这篇论文介绍了决定优化执行时间的方法，无论规则集是否具有终点和汇聚点的属性。终点意味着规则过程被确保不再执行。汇聚点意味着最终的数据库独立于被执行的规则的顺序。

[9.3] Philip A. Bernstein, Barbara T. Blaustein, and Edmund M. Clarke: "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data," Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada (October 1980).

对某一特定类型提出了一个有效的完整性实施方法。例如“集合 *A* 中的每一个值必须小于集合 *B* 中的每一个值。”可以看出：给出的约束在逻辑上等同于另一个约束“集合 *A* 中的最大值必须小于集合 *B* 中的最小值”。将这一类约束组织起来，用隐藏的变量来自动表示最大值与最小值的关系，系统能够将实施给定更新操作的约束时的比较次数从 *A* 或 *B* 的势（取决于更新操作应用于哪个集合）减少至 1，当然代价是必须维护这些隐藏的变量。

[9.4] O. Peter Buneman and Erik K. Clemons: "Efficiently Monitoring Relational Databases," *ACM TODS 4*, No. 3 (September 1979).

本文谈到了有效实施触发过程（这里称为 alerters）的方法——尤其是在无需计算条件的值的情况下，判断触发条件是否满足这一问题方面。其中给出了检测是否满足触发条件的更新的一种方法（也称“避免”算法）；讨论了当避免算法失败时减少处理的技巧；判断相关元组的一个子集是否满足触发条件。

[9.5] Stefano Ceri and Jennifer Widom: "Deriving Production Rules for Constraint Maintenance," Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

描述了基于 SQL 的约束定义语言，并给出了识别可能违反给定约束的所有操作的算法。（前面参考文献 [9.21] 中给出了这种算法的初步介绍。）本文也介绍了优化和正确性方面的知识。

[9.6] Stefano Ceri, Roberta J. Cochrane, and Jennifer Widom: "Practical Applications of Triggers and Constraints: Successes and Lingering Issues," Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt (September 2000).

摘要中的引用：“触发器应用的有意义的部分事实上不过是各种完整性约束级别的完整性维

护。”这篇论文还提到许多触发器，特别包括“约束保护”，事实上能够由声明表示自动生成。

[9.7] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca: "Automatic Generation of Production Rules for Integrity Maintenance," *ACM TODS 19*, No. 3 (September 1994).

根据参考文献 [9.5]，本文介绍了自动修护违反约束造成损坏的可能性。约束被编译成由以下组件组成的产生规则：

1) 一系列能够违反约束的操作。

2) 如果违反约束（基本上是原来约束的相反），布尔值将为 TRUE。

3) 一个 SQL 修复过程。

这篇文章仍然包括相关文章的全面研究。

[9.8] Roberta Cochrane, Hamid Pirahesh, and Nelson Mattos: "Integrating Triggers and Declarative Constraints in SQL Database Systems," Proc. 22nd Int. Conf. on Very Large Data Bases, Mumbai (Bombay), India (September 1996).

“谨慎定义触发和声明约束之间的相互作用，以避免它们之间不一致的行为，并为用户提供这种相互作用的可理解的模型。这篇论文定义了这样一种模型”。此模型成为 SQL 相关方面的基础。

[9.9] E. F. Codd: "Domains, Keys, and Referential Integrity in Relational Databases," *InfoDB 3*, No. 1 (Spring 1988).

该文讨论域、主码和外码。由于 Codd 是这三个概念的创造者，此文显然具有权威性；但仍有一些问题未曾解决也未作解答。对于必须选取一个候选码作为主码的问题，作者给出如下描述：“不支持这一准则好像是在用这样一种寻址方式的计算机……每次特定的事件发生时，它都改变基数（如遇到的地址是素数时）。”但如果我们接受这一点，那么为什么不用一种任何情况下都清晰的寻址方式呢？通过供应商号来寻址供应商和通过零件号来寻址零件，这样不也可以吗？发货也是同样的道理，它涉及复合的寻址方式（实际上，对于全程唯一的寻址方法有很多内容。参见 14 章中参考文献 [14.21] 的评论）。

[9.10] C. J. Date: "Referential Integrity," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981). Republished in revised form in *Relational Database: Selected Writings*. Reading, Mass: Addison-Wesley (1986).

该文介绍参照行为（尤其是 CASCADE 和 RESTRICT），即本章 9.10 节中讨论的。该文的初版（VLDB 1981）和修正版的区别在于，初版遵从参考文献 [14.7]，允许一个外码参照多个关系变量，由于参考文献 [9.11] 中所说的原因，修正版中将其取消了。

[9.11] C. J. Date: "Referential Integrity and Foreign Keys" (in two parts), in *Relational Database Writings 1985 - 1989*. Reading, Mass: Addison-Wesley (1990).

该文的第一部分讨论了参照完整性的概念，并提供了更好的定义方式（和基本原理）。第二部分提供了进一步讲述，并给出了应用的办法。特别讨论了（a）外码重叠；（b）复合外码中部分为空；（c）有共同边界的参照路径（如不同的参照路径有相同的起点和终点）时引起的问题。注意：该文有小部分内容与参考文献 [9.14] 相抵触。

[9.12] C. J. Date: "A Contribution to the Study of Database Integrity," in *Relational Database Writings 1985 - 1989*. Reading, Mass.: Addison-Wesley (1990).

“本文试图为完整性问题建立一个结构，提出一个完整性的分类方案；用这一方案来澄清数据完整性概念蕴含的重要性；概述了用一门具体语言来明确表达完整性约束的方法；指出了几个需要进一步研究的领域”。本章的部分内容是基于本文的早期版本，但它的分类方案应该被本章 9.9 节中的修正方案取代。

[9.13] C. J. Date: "Integrity," Chapter 11 of reference [4.21].

IBM 的 DB2 产品确实提供了声明主码、外码的支持。（实际上，它只是最早提供这一功能的产品之一。）正如该文所说，DB2 提供这一功能并不是因为实施中有什么不便，而是为了保证可预测的行为。举个简单的例子：设关系变量 *R* 只含有两个元组，它们的主码分别为 1 和 2，考虑更新请求“将它们的主码都乘以 2”，正确的结果是 *R* 的两个元组主码分别为 2 和 4。如果系统先将 2 更新为 4（用 4 替换 2），然后更新主码 1（用 2 替换 1），则操作成功；相反，如果系统试图先更新 1（用 2 替换 1），就会违反唯一性的约束，则操作失败（数据库恢复原状）。换言之，操作的结果是不可预测的。为避免这种不可预测性，DB2 简单地将引起多种可能结果的情况设为不

合法。然而这种限制过于严格了见参考文献［9.17］。

注意，像前面的例子中所说的，DB2 是进行“飞行中检查”的（inflight checking）——也就是说在每一次单个元组更新后进行检查。这种飞行中检查是逻辑错误的（参见第 6 章 6.5 节中关于“更新关系变量”的讨论）；只是因为实现方面的原因才这样做。

［9.14］ C. J. Date: "The Primacy of Primary Keys: An Investigation," in *Relational Database Writings 1991 - 1994*. Reading, Mass.: Addison-Wesley (1995).

该文基于这样一种观点“某个候选码比其他的更重要并不是件好事”。

［9.15］ C. J. Date: *WHAT Not HOW: The Business Rules Approach to Application Development*. Reading, Mass.: Addison-Wesley (2000).

一个不太正式的（没有技术需求的）“商业规则”的介绍。见参考文献［9.21］和［9.22］。

［9.16］ C. J. Date: "Constraints and Predicates: A Brief Tutorial" (in three parts), *http://www.dbdebunk.com* (May 2001).

本章主要基于这个教程。如下是此论文结论部分的改编版本：我们看到数据库是一系列真命题组成。事实上，数据库和应用于数据库命题的操作者是一个**逻辑系统**。这里的逻辑系统是指正规系统——就像欧氏几何，有**公理**和**推理规则**，通过这些推理规则我们能够证明这些来自公理的**理论**。当 Codd 1969 年第一次发明关系模型时，他指出数据库不是一系列**数据**，而是一系列**事实**，或者说是逻辑家们称作的真命题。那些给出的命题说明了在数据库关系变量中表示的命题。推断规则就是新命题通过给出的命题得出的规则。即这些规则是告诉我们怎么应用关系几何操作的规则。因此，当系统评估某一关系表达式时（尤其是回应某一查询时），它从已有的真值中获得真理，实际上就证明了一个公理！

一旦我们发现前面的真值，我们就会看到在处理“数据库问题”时整个正式逻辑工具都是可以获得的。比如这些问题：

- 对用户来说数据库是什么样子？
- 查询语言是什么样子？
- 怎样把结果提交给用户？
- 怎样最好地实施查询（更一般地说，怎样估计数据库表达式）？
- 一开始怎样设计数据库？

（先不讨论“完整性约束是什么？”这种问题），所有这些成为易受逻辑处理影响的逻辑问题，并且能被逻辑回答。

当然，它没有直接说关系模型支持前述的关于数据库是怎么一回事的观念。也就是说，在作者看来，关系模式是坚固的，是正确的，并将长期存在。

最后，既然数据库和关系操作是一个逻辑系统，我们看到完整性约束的**至关重要性**。如果数据库违反了完整性约束，那么逻辑系统就是不一致的。我们从不一致的系统中根本得不到任何答案！假定系统的问题是 $p$ 和 NoT $p$ 均为 true（此时系统不满足一致性），这里 $p$ 是命题。假定 $q$ 是任意命题，那么：

- $p$ 为 true，我们可以推出 $p$ OR $q$ 为 true
- $p$ OR $q$ 为 true 并且 NOT $p$ 为 true，我们可以推出 $q$ 为 true

但 $q$ 是任意的！这就是说，任意命题在一个不一致的系统中都可以表示为 true。

［9.17］ M. M. Hammer and S. K. Sarin: "Efficient Monitoring of Database Assertions," Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas (May/June 1978).

该文描述了能够产生一种完整性检查过程的算法，这种算法比以前的在一次更新后简单地强制进行的检查效率更高。在编译时，这一检查就被包含到应用目标代码中。有时可能会发现无需运行时检查；即使需要，也有可能通过多种途径显著减少数据库存取次数。

［9.18］ Bruce M. Horowitz: "A Run-Time Execution Model for Referential Integrity Maintenance," Proc. 8th IEEE Int. Conf. on Data Engineering, Phoenix, Ariz. (February 1992).

我们都知道下列三种结构的某些组合：

1）参照结构（通过参照约束相关联的关系变量的集合）。

2）外码的 DELETE 和 UPDATE 规则。

3）数据库中的实际数据值。

能导致相冲突的情况，也会在操作时引起一些不可预期的行为（详细内容请参见［9.11］）。

有三种方法来解决这一问题：

1）留给用户解决。

2）让系统检测并拒绝运行时可能会引起冲突的结构定义。

3）让系统检测并拒绝实际运行时的错误。

方法 a 是不可取的，b 又过分保守［9.13，9.20］；Horowitz 建议采用方法 c，本文给出了一些运行时要遵守的规则，并证明了它们的正确性，但没有考虑这些运行时检测的性能。

Horowitz 是 SQL/92 定义小组中活跃的一员，从其标准中参照完整性部分不难看出本文的建议可行。

［9.19］ Victor M. Markowitz："Referential Integrity Revisited: An Object-Oriented Perspective," Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

该文的标题"面向对象的观点"表明了作者开放的立场：参照完整性支持了面向对象结构的关系表示。该文并不是真正讨论面向对象的，但它从实体/关系图（见第 14 章）出发，展示了一个会产生关系数据库定义的算法，在此算法中将保证参考文献［9.11］中定义的一些问题（如迭代码）不会出现。

此文从参照完整性的观点讨论了三种商业产品（DB2、Sybase 和 Ingres，1990 年前后）。DB2 提供了声明支持，功能受限；Sybase 和 Ingres 提供过程支持（分别通过触发器和规则），限制要比 DB2 少，但很难用（虽然 Ingres 声称其在技术上优于 Sybase）。

［9.20］ Victor M. Markowitz:"Safe Referential Integrity Structures in Relational Databases," Proc. 17th Int. Conf. on Very Large Data Bases, Barcelona, Spain (September 1991).

该文提出了两个"安全条件"，以保证参考文献［9.11］和［9.18］讨论的问题不会出现。此文也考虑了在 DB2、Sybase 和 Ingres 中实现这些条件时所涉及的问题（也是在 1990 年左右）。对于 DB2，该文表明了其中一些由于安全原因［9.13］而实施的执行限制在逻辑上是不必要的，而其他一些限制也不充分（如 DB2 仍允许一些不安全的状态）。对于 Sybase 和 Ingres，此文说它们提供的过程无法检测出不安全的，甚至不正确的参照定义。

［9.21］ Ronald G. Ross: *The Business Rule Book: Classifying, Delining, and Modeling Rules* (Version 3.0). Boston, Mass.: Database Research Group (1994).

见参考文献［9.22］的注释。

［9.22］ Ronald G. Ross: *Business Rule Concepts.* Houston, Tex.: Business Rule Solutions Inc. (1998).

在近年的商业化产品中，都提供了大量的商业规则支持；一些业界人士开始提议将这些规则作为设计和建立数据库和应用程序的更好的基础（更好，是相对于已确立的方法，如"实体/完整性模型"、"对象模型"、"语义模型"等）。我们一致认为，因为商业规则实际上是对用户更友好的关于谓词、命题和完整性的其他方面的说明方法。Ross 是最早提出商业规则这一方法的，对于感兴趣的人，他的书值得一读，参考文献［9.21］读起来有些费力，参考文献［9.22］是一个简明教程。Ross 所写的另一本书"*Principles of the Business Rule Approach*"（Addison-Wesley，2003）已经出版了。

［9.23］ M. R. Stonebraker and E. Wong: "Access Control in a Relational Data Base Management System by Query Modification," Proc. ACM Nat Conf., San Diego, Calif. (November 1974).

在 University Ingres prototype［8.11］一书中最先提出了一种关于完整性约束的有趣的实施方法（也用于安全性约束——参见第 17 章），此方法基于请求修改（request modification）。完整性约束通过 DEFINE INTEGRITY 语句定义。

```
DEFINE INTEGRITY ON <relvar name> IS <bool exp>
```

例如：

```
DEFINE INTEGRITY ON S IS S.STATUS > 0
```

设用户 U 试图进行下列操作：

```
REPLACE S ( STATUS = S.STATUS - 10 )
WHERE   S.CITY = "London"
```

然后 Ingres 自动修改 REPLACE 语句为：

```
REPLACE S ( STATUS = S.STATUS - 10 )
WHERE   S.CITY = "London"
AND     ( S.STATUS - 10 ) > 0
```

当然，被修改过的操作不会违反完整性约束。

此方法的一个缺陷是并非所有的约束都可以用这种简单的方法加以调整；实际上，QUEL只支持布尔表达式是一个简单限制条件的约束。然而，即使是有限的支持，也不能在许多的实际系统中得到体现。

[9.24] A. Walker and S. C. Salveter: "Automatic Modification of Transactions to Preserve Data Base Integrity Without Undoing Updates," State University of New York, Stony Brook, N. Y.: Technical Report 81/026 (June 1981).

本文讲述了一个技巧，用来自动修改“事务模板”（也就是事务源代码）为相应的安全模板（safe template）。安全，是指经修改后的模板不会违反已声明的完整性约束，它是通过向原模板中加入查询和测试来实现的。在运行时，如果测试失败，则事务被拒绝，并产生一个错误消息。

[9.25] Jennifer Widom and Stefano Ceri (eds.): *Active Database Systems: Triggers and Rules for Advanced Database Processing*. San Francisco, Calif.: Morgan Kaufmann (1996).

关于“主动数据库”（也就是能对特定事件进行行为指定的数据库系统——即具有触发器的系统）的有用的研究纲要和自学教程，其中包括了几个原型系统，尤其是有 IBM Research 的 Starburst（参见 [18.21]、[18.48]、[26.19]、[26.23]、[26.29] 和 [26.30]）以及加利福尼亚大学伯克利分校的 Postgres（见参考文献 [26.36]、[26.40] 和 [26.42, 26.43]）。该书也总结了 SQL: 1992、SQL: 1999（早期版本）和一些商业产品（Oracle、Informix、Ingres 等）的相关内容，还包括一个有用的参考书目录。

# 第10章 视 图

## 10.1 引言

正如第3章中所述，关系模型中的视图只是一个命名了的关系代数表达式（或关系代数的等价形式）。比如：

```
VAR GOOD_SUPPLIER VIEW
    ( S WHERE STATUS > 15 ) { S#, STATUS, CITY } ;
```

当这条语句执行时，此关系代数表达式（即这个**视图定义表达式**）没有被计算，只是让系统把它记住——准确地说，是在系统目录中把它以 GOOD_SUPPLIER 这个名字保存起来。然而，对于用户来说，数据库中好像确实有一个叫作 GOOD_SUPPLIER 的关系变量，其元组如图 10-1 中没有阴影的部分所示（假设的样本数据）。换句话说，GOOD_SUPPLIER 指向一个导出的（虚的）关系变量，其值是视图定义表达式计算后所得到的结果。

在第3章中我们这样解释视图（如 GOOD_SUPPLIER），它是一个用来观察底层数据的窗口；任何所对应数据的更新，通过这个窗口都可以实时和自动地看到（只要是在视图可见的范围内）；相应地，任何对视图的更新将自动和实时地在所映射的数据上进行更新。这样，通过窗口也是可见的。[○]

| GOOD_SUPPLIER | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| | S1 | Smith | 20 | London |
| | S2 | Jones | 10 | Paris |
| | S3 | Blake | 30 | Paris |
| | S4 | Clark | 20 | London |
| | S5 | Adams | 30 | Athens |

图 10-1 基本关系变量 S 上的视图 GOOD_SUPPLIER（加没有阴影的部分所示）

现在，依据环境不同，可能有用户没有意识到 GOOD_SUPPLIER 是一个视图；也有的用户可能意识到这一点，明白后面有一个真正的（基本的）关系变量 S，还有一些用户以为 GOOD_SUPPLIER 是一个“真正的”（基本的）关系变量。无论用户处于哪一种情况，都没什么不同。重要的是，用户能像操纵一个真正的或基本的关系变量一样来操纵 GOOD_SUPPLIER。例如，下面有一个对 GOOD_SUPPLIER 的查询：

```
GOOD_SUPPLIER WHERE CITY ≠ 'London'
```

设数据如图 10-1 所示，则结果为：

| S# | STATUS | CITY |
|---|---|---|
| S3 | 30 | Paris |
| S5 | 30 | Athens |

此查询的结果就像是常规的关系变量上的一个常规查询。并且，如第3章所说的，系统对这种查询的处理是将它转化为在所映射的关系变量（或基本关系变量）上的等价查询。实现的方法是将查询中出现的视图名字替换为定义视图的表达式。在这个例子中，**替代过程**为：

```
( ( S WHERE STATUS > 15 ) { S#, STATUS, CITY } )
                          WHERE CITY ≠ 'London'
```

它的等价形式为

```
( S WHERE STATUS > 15 AND CITY ≠ 'London' )
                        { S#, STATUS, CITY }
```

这个查询产生前面所示的结果。

---

○ 实际上，在 SQL 中可能看不到这点，见 10.6 节关于 WITH CHECK OPTION 的讨论。

顺便提一下，替代过程只是更精确地描述出这个工作（由于关系封闭的特性）：将视图名字用视图定义表达式代替。关系封闭性是指，无论何时，若关系变量名 *R* 出现在一个表达式中，则一个关系表达式可以替代它出现（只要它与 *R* 的类型相当）。换句话说，视图能够正常工作是因为在关系代数中关系是封闭的——这就是为什么封闭性有如此重要地位的原因。

更新操作也以同样的方式工作。例如以下操作：

```
UPDATE GOOD_SUPPLIER WHERE CITY = 'Paris'
     { STATUS := STATUS + 10 } ;
```

被转化为：

```
UPDATE S WHERE STATUS > 15 AND CITY = 'Paris'
     { STATUS := STATUS + 10 } ;
```

对 INSERT 和 DELETE 操作的处理与之相似。

**1. 进一步举例**

这一小节将举出更多的例子，以便在后文参考。

```
1. VAR REDPART VIEW
       ( P WHERE COLOR = COLOR ('Red') ) { ALL BUT COLOR }
                                         RENAME WEIGHT AS WT ;
```

视图 REDPART 是关系变量 parts 的选择投影（附加了属性重命名）。包含属性 P#、PNAME、WT 及 CITY 并且只包含红色零件的元组。

```
2. VAR PQ VIEW
       SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY ;
```

视图 PQ 是一种对底层数据的统计汇总或压缩。

```
3. VAR CITY_PAIR VIEW
       ( ( S RENAME CITY AS SCITY ) JOIN SP JOIN
         ( P RENAME CITY AS PCITY ) ) { SCITY, PCITY } ;
```

CITY_PAIR 视图通过供应商号以及零件号将供应商表、零件表和发货表进行连接，然后将结果投影到 SNAME 和 PNAME。不严格地说，当且仅当位于 *x* 市的供应商提供了存放在 *y* 市的零件时，城市对（*x*，*y*）在视图 CITY_PAIR 中出现。比如，供应商 S1 提供零件 P1，S1 位于 London，零件也存放在 London，则城市对（London，London）出现在视图中。

```
4. VAR HEAVY_REDPART VIEW
       REDPART WHERE WT > WEIGHT ( 12.0 ) ;
```

此视图经由另一个视图定义。

**2. 定义和删除视图**

下面是定义视图的 Tutorial D 语法：

```
VAR <relvar name> VIEW <relation exp>
                  <candidate key def list> ;
```

< *candidate key def list* > 允许为空（即规格说明可以省略），因为系统可以推断出视图的候选码 [3.3]。在 GOOD_SUPPLIER 的例子中，系统应该意识到唯一的候选码是 {S#}，是从底层基本关系变量 S 中继承来的。

我们说——用第 2 章中的 ANSI/SPARC 术语——视图定义综合了外部模式的功能和外模式/概念模式映射的功能，因为它既指出了外部实体的内容（视图），也说明了外部实体到概念模式（就是到底层的基本关系变量）映射的方式。注意：一些视图的定义没有指明外模式/概念模式映射，而是外模式/外模式映射。前面的视图 HEAVY_REDPART 就是这样的一个例子。

删除一个视图的语法是：

```
DROP VAR <relvar name> ;
```

< *relvar name* > 是指一个视图。在第 6 章中曾提到过，如果存在任一视图定义指向一个基本关系变量，则删除此基本关系变量的请求将会失败。与此相似，如果存在另一个视图定义指向某个视图，则删除此视图的请求也将会失败。替换的方法是（与引用约束类似），可以考虑扩展视图定义语句，使之包括“RESTRICT”或者“CASCADE”。RESTRICT 的意思是当某个视图被其他视图定义所引用时，对此视图的删除请求将会失败；CASCADE 的意思是删除请求会级联地删除引用此视图的视图。注意：SQL 不支持这样的选项，但可以在 DROP 语句而不是视图定义语句中对这一选项进行设置。这里没有默认值，所需选项必须显式给出（参见 10.6 节）。

## 10.2 视图的用途

使用视图有很多原因，下面给出其中的一些理由：

- 视图提供了一个快捷方式或者是“宏”的功能。

考虑查询“存放在伦敦的供应商所提供的零件的城市”。用前面的“进一步举例”小节中的视图 CITY_PAIR，下述形式就可以实现：

```
( CITY_PAIR WHERE SCITY = 'London' ) { PCITY }
```

相反，没有视图，这个查询的实现就要复杂得多：

```
( ( ( S RENAME CITY AS SCITY ) JOIN SP JOIN
    ( P RENAME CITY AS PCITY ) )
  WHERE SCITY = 'London' ) { PCITY }
```

用户可以直接使用第二种格式——在安全性约束上是相同的——但第一种显然要简单一些。第一种只是第二种的快捷方式；在执行前，系统的视图处理机制将有效地把第一种格式转化为第二种格式。

这一点与编程语言中的“宏”很相似。原则上，编程语言的用户可以在源代码中直接写出宏的扩展后的形式——但当用到宏，让宏处理系统来进行这一转换时，这一过程要简便得多（也可能是出于便于理解的原因）。相似的道理也可用于视图。这样，视图在数据库中的作用与宏在编程语言中的作用是相似的，宏的优势显然也就是视图的优势。尤其注意，对于视图操作没有运行时的系统开销，只有视图处理时的一点系统开销（与宏的扩展相似）。

- 视图使相同的数据可在同一时间被不同的用户以不同的方式查看。

视图可以方便地让用户仅注意到自己关心的数据而忽略其他。当有很多要求各异的用户时，这一点就显得很重要了，视图使这些用户同时与同一个数据库交互。

- 视图对于隐藏的数据自动给予安全保障。

“隐藏的数据”是指在某个视图中不可见的数据（如在视图 GOOD_ SUPPLIER 中的供应商名称）。显然，这些数据对于特定的视图来说，在存取中是安全的，因此强制用户通过视图来对数据库进行存取，相当于建立了一个简单而有效的安全机制，对于这一用途，第 17 章将进行更详细的讨论。

- 视图能提供逻辑上的数据独立性。

这是视图最重要的用途之一，详见下一小节。

### 1. 逻辑上的数据独立性

这里提醒读者，逻辑上的数据独立性可以定义为：用户或用户程序在数据库逻辑结构（逻辑结构是指概念层的结构——参见第 2 章）发生改变时的抗扰性。即通过视图，可以取得关系系统中逻辑上的数据独立性。这种逻辑上的数据独立性有两个方面：**可成长性**（growth）和**可重构性**（restructuring）。（可成长性这个概念很重要，但与视图关系不大，这里介绍它只是为了内容上的完整。）

- 可成长性

随着数据库纳入新的数据而不断增长，对于此数据库的定义也相应地增长了。对于“成长”有两种可能的情况：

1）已有的基本关系变量为增加新的属性而扩展，这是相应于给对象的某一种现存的类型增

加的一种新信息而言——如对供应商基本关系变量增加折扣这一属性。

2）引入一个新的基本关系变量，这是相应于增加一种新的类型对象的——例如，对于供应商－零件数据库增加工程信息。

这两种情况对于已存在的用户或用户程序都没有影响，至少理论上是这样（针对于 SQL 中的“SELECT *”的警告，请参考第 8 章中的例 8.6.1 的第 6 点）。

■ 可重构性

有时需要对数据库进行重新构造。虽然全部信息的内容没有改变，但信息的逻辑位置可能已经改变。也就是说，对于基本关系变量的属性的分配同样发生了改变。这里，我们只考虑一个简单的例子。假设出于某种原因，希望用下面的两个基本关系变量替代基本关系变量 S：

```
VAR SNC BASE RELATION { S# S#, SNAME NAME, CITY CHAR }
    KEY { S# } ;

VAR ST BASE RELATION { S# S#, STATUS INTEGER }
    KEY { S# } ;
```

重要的是，旧关系变量 S 是新关系变量 SNC 和 ST 的连接（SNC 和 ST 都是 S 的投影）。因此以这个连接创建一个视图 S：

```
VAR S VIEW
    SNC JOIN ST ;
```

任何以前对基本关系变量 S 的引用现在都成为针对视图 S 的。这样——假设系统能够正确处理视图中的数据操作——相对于数据库的重构，对用户和用户程序是没有影响的。㊀

另外，对于将原来的供应商关系变量 S 用它的两个投影 SNC 和 ST 代替，并不是随随便便就可以这样做的，尤其是当发觉发货关系变量 SP 也要进行一些处理时——因为它有一个参照原关系变量 S 的外码。见本章末的习题 10.14。

再回到本章讨论的主线：当然，从 SNC-ST 这个例子并不能得出结论，对于所有的重构行为，都能相应地得到逻辑上的数据独立性。关键是，是否存在一个从重构后的数据库版本到原来的数据库版本的明确的映射关系（也就是说，重构是否可逆），或者说，数据库的这两个版本是否是**信息等价的**（information-equivalent）。如果不是，就无法得到逻辑上的数据独立性。

**2. 两个重要的准则**

通过前面对逻辑数据独立性的讨论，我们得出另一个要点，即视图用于两个非常不同的目的：

- 定义视图 *V* 的用户显然知道相应的视图定义表达式 *X*；只要表达式 *X* 可用的地方，视图 *V* 都可用。但这种用法（如前文所示）只是快捷方式。
- 当用户只被告知视图 *V* 存在并可用时，他可能并不知道视图定义表达式 *X*；对于这样的用户，视图 *V* 在外观上和行为上都应像是基本关系变量。

继续前面的讨论，对于哪个关系变量是基本的而哪个关系变量是导出的这样一个问题具有很大的随意性，现在讨论这个问题。还以前面“可重构性”小节中的关系变量 S、SNC 和 ST 为例。很显然，可以

（a）定义 S 为基本关系变量，而 SNC 和 ST 为该基本关系变量投影得到的视图；

（b）也可以定义 SNC 和 ST 为基本关系变量，而 S 为这两个基本关系变量连接操作得到的视图。㊁

---

㊀ 这里也仅是指原则上！不幸的是，当今的 SQL 产品（以及 SQL 标准）通常不支持在视图上对数据进行操作，因此也就不支持例中对变化的抗扰性。具体而言，大多数 SQL 产品（不是所有的）支持视图检索，因此可以为检索操作提供完整的逻辑数据独立性。据笔者所知，没有 SQL 产品支持正确的视图更新（标准 SQL 同样不支持），因此没有 SQL 可以为数据更新提供完整的逻辑数据独立性。注意：参考文献［20.1］中描述的一种产品能够提供正确的视图更新（但不是 SQL 产品）。

㊁ 第 12 章 12.2 节对于无丢失分解（nonloss decomposition）的讨论与此相关。

在基本关系变量和导出的关系变量中不能有随意的或不必要的区别。这一事实称为**互换性准则**（The Principle of Interchangeability）（对基本关系变量和导出的关系变量而言）。尤其注意，这一准则暗示视图一定是可更新的——数据库的可更新性不能取决于“哪些关系变量是基本的，哪些是导出的”这样一个可随意的选择。详见 10.4 节的讨论。

我们暂时将基本关系变量的集合称为真实数据库（real database）。一个典型的用户并非只和真实数据库自身交互，而是与可表达数据库（expressible database）交互，可表达数据库是基本关系变量和视图的混和体。现在假设，可表达数据库中的任何一个关系变量都不能由其余的关系变量派生出来（因为这样的关系变量可以被删除而没有信息丢失），这样，从用户的角度看，它们都是基本关系变量。当然，它们之间都是相互独立的（用第 3 章中的术语来说就是“自治的”）。对数据库来说也是同样的——可以随意指定哪个数据库是真实的，只要所有的选择是信息等价的。这一事实称为**数据库相对性准则**（The Principle of Database Relativity）。

## 10.3 视图检索

本章已经简单解释了如何把一个视图上的检索操作翻译成在底层的基本关系变量上的等价操作。现在，把这个解释变得更形式化。

首先，每一个给定的关系表达式都可以被看作是一个关系赋值**函数**（如第 6 章中介绍的）：对表达式中的各关系变量赋值（表示调用此函数时的参数）后，表达式产生另一个关系。现在设 $D$ 是一个数据库（目前把它看作是基本关系变量的集合），$V$ 是定义在 $D$ 上的视图，也就是说，视图的定义表达式 $X$ 是定义在 $D$ 上的一个函数：

```
V = X ( D )
```

设 $RO$ 是一个在 $V$ 上的检索操作；$RO$ 也是一个关系赋值函数，检索的结果是：

```
RO ( V ) = RO ( X ( D ) )
```

这样，检索的结果被定义为等于将 $X$ 应用于 $D$ 上的结果——也就是说，**物化**一个关系的副本，这个关系就是视图的当前值——然后将 $R$ 应用到这个物化（materializing）后的副本上。使用**替代**过程显然更有效率，正如 10.1 节中所讲的。（现在可以看出该过程等价于构成一个函数 $C$，而 $C$ 是函数 $X$ 和 $R$ 的组合 $R(X)$，然后将 $C$ 施加于 $D$ 上。）用物化而不是用替代来定义视图检索的语义很方便，至少在理论上如此；换句话说，如果替代产生的结果能够保证与使用物化时产生的结果相同（当然，确实如此），那么替代就是有效的。

经过上述讨论，你应该已经对这些内容有了基本了解，了解它们是出于下列原因：

- 首先，它为后面讨论与此相似的（更繁琐的）更新操作打下基础。
- 其次，它阐明了物化是很好且合法的视图实现方法（虽然效率不是很高）——至少是对于检索操作而言。但物化不适用于更新操作，因为对视图的更新实际上要精确地转化为对视图所对应的基本关系变量的更新，而不是某些数据的临时物化的副本。参见 10.4 节。
- 再次，虽然替代过程很直观而且在理论上百分之百有效，但是，在一些 SQL 产品中它实际上无法工作！也就是说，在一些 SQL 产品中，对视图的检索会莫名其妙地失败。对于 SQL/92 以前的 SQL 标准也在实际中不起作用。失败的原因是这些产品以及 SQL 标准的早期版本，不能完全支持关系封闭属性。参见本章末的习题 10.15。

## 10.4 视图更新

视图是关系变量，因而（可以像所有的变量一样）通过定义进行更新。然而，在历史上对视图更新的看法却没有这样简单。视图的更新问题可以这样表达：对于给定视图上的一个更新操作，要在对应的基本关系变量上进行怎样的更新操作，才能实现视图的更新？更准确地说，设 $D$ 是数据库，$V$ 是 $D$ 上的视图，也就是说，此视图的定义 $X$ 是在 $D$ 上的函数：

```
V = X ( D )
```

（如10.3节所示）。设 *UO* 是一个在 *V* 上的更新操作；*UO* 可以被看作是一个能改变参数的操作，于是

```
UO ( V ) = UO ( X ( D ) )
```

在视图上进行更新这一问题就变成寻找一个在 *D* 上的更新操作 *UO'*：

```
UO ( X ( D ) ) = X ( UO' ( D ) )
```

因为 *D* 是唯一真正存在的东西（视图是虚拟的），因此更新无法直接针对视图自身实现。

在进一步讨论之前，有必要强调一下，视图更新一直是近年来研究的重点，也有很多人提出了多种不同的解决方案，参见［10.4］、［10.7～10.10］和［10.12］，特别是Codd针对RM/V2的提议［6.2］。本章介绍在［10.6，10.11］中提到的一种相对较新的方法，它不如上述提议那么典型，但它的一些思想可以和该提议中某些最好的方面相媲美。它的优点之一是与以前的方法相比，可以支持更多种类的可更新视图；实际上，它把所有视图都看做潜在可更新的，而不考虑完整性约束。

**1. 修订后的黄金法则**

回忆前一章学过的黄金法则：

*如果一个更新操作将使一个变量处于违反自身某个谓词的状态，那么这样的更新是被禁止的。*

或者：

*关系变量不能违反自身的谓词状态。*

（这一章中，我们使用变量谓词这个术语来说明内部具体的谓词，使用术语谓词说明具体的关系变量谓词。实际上，我们在本书后面都将使用此约定，不接受与其相反的规定。）

当刚引入这条法则时，我们强调它适用于所有的关系变量，包括基本的或导出的。也就是说，视图也有谓词（事实上，由于互换定律的原因，它必须有）。为了正确实现视图更新，系统也要知道这些谓词是什么。那么视图的谓词是什么呢？显然，我们要的是**谓词推理规则**（predicate inference rule）的集合，这样，如果知道对某个关系操作的输入的谓词，我们就能从此操作中推断出输出的谓词。有了这样的一些规则，就能通过视图直接或间接参照那个基本关系变量的谓词来推断出视图的谓词（当然，这些基本关系变量的谓词是已知的：它们是在此关系变量上定义的约束——即候选码约束——的逻辑与）。

事实上很容易找到一组这样的规则——它们遵从关系操作符的定义。例如，如果 *A* 和 *B* 是同一种类型的变量，它们的谓词分别是 *PA* 和 *PB*。如果视图 *C* 定义为 *A* 与 *B* 的交集，那么谓词 *PC* 将定义为（*PA*）AND（*PB*）。考虑到：

- 当且仅当元组 *t* 既存在于 *A* 又存在于 *B* 的情况下，它属于 *C*。
- 如果元组 *t* 存在于 *A* 中，那么 *PA*(*t*)一定为真（“*PA*(*t*)”是 *PA* 的一个实例）。
- 否则，如果 *t* 存在于 *B* 中，则 *PB*(*t*)为真。
- 因此 *PA*(*t*)AND *PB*(*t*)必须为真，谓词 *PC* 定义为 *PA* 与 *PB* 的交集。

我们接下来将考虑其他的关系操作符。

导出的关系变量会自动从被导出的关系变量中“继承”某些约束，但也可以让导出的关系变量还受附加约束和继承约束［3.3］的制约。因此，能够显式地描述约束是必要的（视图的候选码就是一个例子），**Tutorial D** 支持这一可能性。为了简单，下文忽略这一可能性。

**2. 关于视图更新机制**

对于视图更新，系统操作有很多要遵守的重要准则（**黄金法则**是其中最重要的一个但并不是唯一的）。涉及到的准则如下：

1）视图可更新性是一个语义学的问题，而与语法无关——也就是说，视图能否更新不能依赖于视图定义的语法形式。比如，下面两个定义在语义上是相同的：

```
VAR V VIEW
    S WHERE STATUS > 25 OR CITY  = 'Paris' ;

VAR V VIEW
  ( S WHERE STATUS > 25 ) UNION ( S WHERE CITY = 'Paris' ) ;
```

显然，这两个视图应该都可更新，或者都不可更新（实际上，它们是可更新的）。相反，SQL 标准和当前的大部分 SQL 产品采用这样的特殊处理办法：即第一个视图是可更新的，但第二个是不可更新的（参见 10.6 节）。

2）下一个问题是，对于视图就是基本关系变量这一特殊情况，视图更新机制必须要正确处理——因为如果视图 *V* 定义为 *B* UNION *B*（或者 *B* INTERSECT *B*，或者 *B* WHERE TRUE，或其他结果等于 *B* 的表达式），那么 *B* 和 *V* 在语义上是无法区分的。因此，适用于 *V* = *B* UNION *B* 这一视图上的更新规则，必须要与在 *B* 上直接进行更新时产生的结果相同。换句话说，这一节的标题虽然是“视图更新”，但不如“关系变量更新”更合适；我们一直是在描述一种对于所有关系变量都正常工作的理论，而不仅仅是对于视图。

3）更新法则必须要在用到它的地方保持对称。比如，对于视图 *V* = *A* INTERSECT *B* 的 DELETE 法则，不应该产生这样的结果：在 *A* 中这个元组被删除了，而在 *B* 中却没有删除。虽然这种单方面的删除也会产生元组在视图上被删除的效果，但是，元组应该在 *A* 和 *B* 上都被删除。（换言之，不能产生歧义——对于视图更新应该有一个明确的工作方式，在任何情况下它都能正常工作。尤其是，当两个视图分别被定义为 *A* INTERSECT *B* 和 *B* INTERSECT *A* 时，它们之间应没有任何逻辑差别。）

4）更新规则要考虑所有生效的触发过程，包括像级联删除这样的参照动作。

5）为简便，不妨将 UPDATE 看做 DELETE-INSERT 操作序列的简写形式。我们将对它进行这样的处理。当下列条件满足时，这种简写方式是可以接受的：

- 在更新过程中不进行关系变量谓词的检查；也就是说，UPDATE 的扩展形式是 DELETE-INSERT-检查，而不是 DELETE-检查-INSERT-检查。当然，原因是 DELETE 会暂时违反关系变量谓词，但整个 UPDATE 不会违反；比如，设关系变量 *R* 含有 10 个元组，如果 *R* 的关系变量谓词要求 *R* 至少包含 10 个元组，考虑在 *R* 上更新元组 *t* 会产生什么样的结果。
- 触发过程也同样不能在更新过程中被实施。（实际上它们在结束时才实施，在关系变量谓词检查之前。）
- 这一简写形式在投影视图中要有一些细小的调整（见这一节的后一部分）。

6）所有视图上的更新操作必须通过在相对应的关系变量上实施同类的更新操作来实现。也就是说，插入映射为插入，删除映射为删除。（根据前文的提示，这里可以忽略更新操作。）相反，有一些视图（如 UNION 视图）把插入映射成删除。还有就是在一个基本关系变量上的插入有时也会映射成删除！得到这一结论是因为基本关系变量 *B* 在语义上等于 UNION 视图 *V* = *B* UNION *B*。在其他类型的视图上（限制、投影、相交，等等）也会产生相似的问题。在基本关系变量上的插入可能实际上是一个删除，这一思想显然是荒谬的：因为我们的出发点是 INSERT 映射为 INSERT，DELETE 映射为 DELETE。

7）一般来说，当更新规则被应用到视图 *V* 时，它将指定在 *V* 所映射的基本关系变量上的操作。即使当这些基本关系变量就是视图自身时，这些规则也要能正确工作。也就是说，这些规则要能递归实施。当然，如果对于被映射的基本关系变量的更新请求失败，则原更新也失败；因此，视图上的更新成功与否和在基本关系变量上的更新一样。

8）这些规则不能断定数据库是设计良好的（即完全规范化的，参见第 12 章和第 13 章）。但如果数据库不是规范设计的，它们有时会产生奇异的结果——在支持规范设计上，这可被看做一个额外的论据。下一小节中会给出这种“奇异结果”的例子。

9）不应该有很显然的原因使得某些更新操作允许在视图上进行，而其他的却不行（如 DELETE 可以而 INSERT 不行）。

10）在一定的范围内，INSERT 和 DELETE 应该是互反的操作。

回忆一下另一个重要的观点。在第 6 章中曾说过，关系操作（尤其是关系更新）都是以集

合为单位的；只包含一个元组的集合是例外。此外，多元组更新有时也是必要的（也就是说，某些更新不能用一系列的单元组更新来完成）。并且，这一观点对基本关系变量和视图都是对的。出于简化的目的，把大部分更新规则以单元组操作的形式呈现，但我们要始终意识到单元组操作是简化了的形式，有时甚至是过分简化的形式。

下面逐个考虑关系代数中的操作符，从并、交、差开始。注意：对这三种操作，假设是在分别处理定义表达式为 *A* UNION *B* 或 *A* INTERSECT *B* 或 *A* MINUS *B* 的视图，其中 *A* 和 *B* 是关系表达式（也就是说，它们未必是基本关系变量）。*A* 和 *B* 所指代的关系必须是同一关系类型。相应的关系变量谓词分别是 PA 和 PB。

注意：之后讨论的一些规则和例子可能产生副作用。现在我们普遍认为副作用是不受欢迎的；但是，当 *A* 和 *B* 恰好代表同一个关系变量的两个有重叠的子集时，副作用是不可避免的，这样的情况常常在出现在 UNION、INTERSECTION 以及 DIFFERENCE 视图中。

3. 并

下面是 *A* UNION *B* 的插入规则：

- **INSERT**：新的元组必须满足 *PA* 或 *PB*，或同时满足 *PA* 和 *PB*。若它仅满足 *PA*，则它被插入到 *A* 中，注意这种插入可能会引起也被插入 *B* 中去的副作用 。若它满足 *PB*，则被插入到 *B* 中，除非由于上一种情况的副作用已经被插入 *B* 中。

解释：新插入的元组至少要满足 *PA* 或 *PB* 中的一个，否则它就不会被包含在 *A* UNION *B* 中——也就是，它不满足 *A* UNION *B* 上的关系变量谓词 *PA* 或 *PB*。（再次假设新元组并没有在 *A* 或 *B* 中出现过，否则就是企图插入一个已有的元组。有时我们的假设并不足够严格。）假设新元组被插入到它在逻辑上属于的 *A* 或 *B*（或两者都是）。

注意：这一规则的程序处理方式（插入 *A*，然后插入 *B*）应该被理解为是因为教学的目的而被简化；这并不暗示 DBMS 必须是按这一执行顺序来完成 INSERT 操作的。实际上，对称法则——前一小节的第 3 点法则——也暗示了这一层意思，因为 *A* 和 *B* 都没有相对于另一方的优先权。在下述的规则中也是一样的。

示例：设视图 UV 定义如下

```
VAR UV VIEW
  ( S WHERE STATUS > 25 ) UNION ( S WHERE CITY = 'Paris' ) ;
```

图 10-2 显示了该视图的一个可能的值，对应于通常的样本数据值。

- 设要插入的元组是（S6，Smith，50，Rome）。[⊖] 这一元组满足谓词公式 S WHERE STATUS > 25，而不满足另一个谓词公式 S WHERE CITY ='Paris'。因此它会被插入到公式 S WHERE STATUS > 25 中。这是由于有关插入的限制（参见后文）。结果是它被插入到供应商基本关系变量中，这样，此元组会像期望的那样出现在视图中。

UV

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

图 10-2　视图 UV（样本值）

- 设要插入的元组是（S7，Jones，50，Paris）。这一元组满足谓词公式 S WHERE STATUS > 25，也满足另一个谓词公式 S WHERE CITY ='Paris'。逻辑上它应该被同时插入到两个视图中。但是，插入到其中任何一个都会引起同时被插入到另一个中的副作用，这时，无需再显式地执行第二个 INSERT。

现在，设 *SA* 和 *SB* 是两个不同的基本关系变量。*SA* 表示 STATUS >25 的供应商，*SB* 表示在 Paris 的供应商（见图 10-3）；设视图 *UV* 定义为 *SA* UNION *SB*，考虑前面讨论过的两个插入操作。插入元组（S6，Smith，50，Rome）到 *UV* 中会使它被插入到 *SA* 中，这正是所要的。但是，插入元组（S7，Jones，50，Paris）到 *UV* 中会使它被同时插入到这两个关系变量中！这一结果在逻辑上是正确的，虽然它有悖于常规（这就是我们在前一节中所说的“奇异结果”的一个例

⊖ 出于可读性的原因，本节采用这样简单的符号来表示元组。

子)。如果数据库设计得不好，就会产生这种奇怪的结果。尤其是，让一个相同的元组出现（也就是满足谓词）在两个不同的关系变量中，这样的设计不好。在第 13 章 13.6 节中将详细讨论这一有争议的情况。

下面讨论 *A* UNION *B* 的 DELETE 规则：

- **DELETE**：如果要删除的元组在 *A* 中出现，则从 *A* 中删除（注意这一删除可能会引起同时在 B 中删除的副作用）。如果它（还）在 *B* 中出现，则在 *B* 中删除。

SA

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S3 | Blake | 30 | Paris |
| S5 | Adams | 30 | Athens |

SB

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |

图 10-3 基本关系变量 SA 和 SB（样本值）

这一规则的例子将被留作练习。注意，删除 *A* 或 *B* 中的一个元组可能会引起级联删除或其他触发过程。

最后，是 UPDATE 规则：

- **UPDATE**：更新后的元组必须能满足 *PA* 或 *PB* 或同时满足两者。如果要被更新的元组在 *A* 中出现，它会被从 *A* 中删除而不引发触发过程（如级联删除等），同样也不引起对 *A* 的关系变量谓词的检查。注意这一删除会有将这一元组同时从 *B* 中删除的副作用。如果元组（还）是在 *B* 中出现，它将被从 *B* 中删除（同样不引发触发过程和谓词检查）。然后，如果更新后的元组满足 *PA*，则被插入到 *A* 中（可能会有同时插入 *B* 中的副作用）。最后，如果更新后的元组满足 *PB*，则被插入到 *B* 中，除非它在插入到 *A* 时已被插入 *B* 中。

这一更新规则实质上是由 DELETE 规则和 INSERT 规则合并而成，除了在 DELETE 后不引发触发过程和谓词检查（任何与 UPDATE 有关的触发过程将在所有的删除和插入后、谓词检查之前执行）。

必须指出，这种处理 UPDATE 的方式可能会产生这样一种后果：一个元组从一个关系变量转移到另一个中。以图 10-3 中的数据库为例，将元组（S5，Adams，30，Athens）更新为（S5，Adams，15，Paris），会使旧元组在 SA 中被删除，新元组被插入到 SB 中。

**4. 交**

现在讨论 *A* INTERSECT *B* 这一视图的更新规则。此处只是简单叙述这些规则而不进行深入讨论（它们遵循与 UNION 视图基本相同的更新模式），但是要注意，*A* INTERSECT *B* 的谓词是（*PA*）AND（*PB*）。它的各种情况的例子将被留作练习。

- **INSERT**：新的元组必须同时满足 *PA* 和 *PB*。如果它原先没有在 *A* 中出现，则它被插入到 *A* 中（注意，这一操作的副作用可能是它同时被插入到 *B* 中）；如果它原先没有（还没有）在 *B* 中出现，则它就被插入到 *B* 中。
- **DELETE**：要删除的元组被从 *A* 中删除。（注意，这一操作的副作用可能是将它同时在 *B* 中删除。）如果它还在 *B* 中，再从 *B* 中删除它。
- **UPDATE**：被更新后的元组必须同时满足 *PA* 和 *PB*。当元组被从 *A* 中删除时，它不引发任何触发过程和谓词检查（注意，这一操作的副作用可能是将它同时在 *B* 中删除）；如果它还在 *B* 中，则再从 *B* 中删除也不引发任何触发过程和谓词检查。接下来，更新后的新元组如果没有在 *A* 中出现，则将之插入到 *A* 中；如果还没有在 *B* 中出现，则将之插入到 *B* 中。

**5. 差**

下面是 *A* MINUS *B* 的更新规则（关系变量谓词是（*PA*）AND NOT（*PB*））：

- **INSERT**：新元组满足 *PA* 而不满足 *PB*。它被插入到 *A* 中。
- **DELETE**：元组被从 *A* 中删除。
- **UPDATE**：更新后的元组满足 *PA* 而不满足 *PB*。元组被从 *A* 中删除而不引发触发过程和谓词检查；更新后的新元组被插入到 *A* 中。

6. 选择

设视图 $V$ 的定义表达式是 $A$ WHERE $p$，$A$ 的谓词是 $PA$，则 $V$ 的谓词是（$PA$）AND（$p$）。比如，S WHERE CITY = 'London'的谓词是（PS）AND（CITY = 'London'），PS 是供应商的谓词。下面是 $A$ WHERE $p$ 的更新规则：

- **INSERT**：新元组要满足 $PA$ 和 $p$，它被插入到 $A$ 中。
- **DELETE**：元组被从 $A$ 中删除。
- **UPDATE**：更新后的元组必须同时满足 $PA$ 和 $p$。旧元组被从 $A$ 中删除而不引发触发过程和谓词检查；新元组被插入到 $A$ 中。

示例：设视图 LS 定义为

```
VAR LS VIEW
    S WHERE CITY = 'London' ;
```

图 10-4 给出了此视图的样本值。

- 在 LS 中插入元组（S6，Green，20，London）的请求会成功。新元组会被插入到 $S$ 中，也就相当于被插入到 LS 中了。
- 在 LS 中插入元组（S1，Green，20，London）的请求会失败，因为它违反了 $S$ 的谓词（因此也违反了 LS）——尤其是，它违反了候选码 {S#} 的唯一性约束。

| LS | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| | S1 | Smith | 20 | London |
| | S4 | Clark | 20 | London |

图 10-4　视图 LS（样本值）

- 在 LS 中插入元组（S6，Green，20，Athens）的请求将失败，因为它违反了约束 CITY = 'London'。
- 在 LS 中删除元组（S1，Smith，20，London）的请求会成功。元组被从 $S$ 中删除，因此也就是在 LS 中被删除了。
- 在 LS 中更新元组（S1，Smith，20，London）为（S6，Green，20，London）的请求会成功。而更新元组（S1，Smith，20，London）为（S2，Smith，20，London）或（S1，Smith，20，Athens）的请求会失败。（各自的原因是什么?）

7. 投影

这里讨论相关的谓词。设关系变量 $A$（在其上有谓词 $PA$）的属性可分为两个不相交的组 $X$ 和 $Y$。将 $X$ 和 $Y$ 分别看成是一个复合属性，考察 $A$ 在 $X$ 上的投影 $A\{X\}$。设（$x$）是这一投影的一个元组，很显然，这一投影的谓词应该是“在 $Y$ 的域上存在一个值 $y$，使元组（$x$，$y$）满足 $PA$”。例如，关系变量 $S$ 是在 S#、SNAME 和 CITY 上的投影，则在此投影中出现的每一个元组（$s$，$n$，$c$）都应存在一个值 $t$ 使元组（$s$，$n$，$t$，$c$）满足 $S$ 上的谓词。

下面是在 $A\{X\}$ 上的更新规则：

- **INSERT**：设要插入的元组是（$x$），另设 $Y$ 的默认值是 $y$（如果不存在这样的默认值就会出现错误。也就是说如果 $Y$ 不允许默认），元组（$x$，$y$）（必须满足 $PA$）被插入到 $A$。

注意：候选码属性通常没有默认值（参见 19 章）。因此，如果有一个投影不包含所有的候选码，它就不允许被插入。

- **DELETE**：当从 $A\{X\}$ 中删除一个元组时，所有在 $A$ 中的有这个 $X$ 值的元组都被删除。

注意：在实际中，$X$ 至少应包含一个 $A$ 上的候选码，这样在 $A\{X\}$ 上删除一个元组时，在 $A$ 上也只有一个相应的元组被删除。但是，并没有什么逻辑上的原因要求必须这样做。同理也适用于 UPDATE——参见下文。

- **UPDATE**：设要更新的元组是（$x$），更新后的元组是（$x'$）。设 $a$ 是 $A$ 中在 $X$ 上有值 $x$ 的一个元组，$a$ 中在 $Y$ 上的值是 $y$。所有这样的元组都被删除而不引发触发过程和谓词检查。然后，对每一个值 $y$ 都有一个元组（$x'$，$y$）（必满足 $PA$）被插入到 $A$ 中。

注意：在“关于视图更新机制”小节的第 5 条原则中曾说过，在投影问题的更新上要有一点调整。应该注意到，在更新规则中的插入步骤里，插入的元组是被重新写入原来的 $Y$ 值——而不是被写入可用的默认值，但是单独的插入会是这样。

示例：设视图 SC 定义为

```
SC { S#, CITY }
```

图 10-5 显示了视图的样本值。

- 在 SC 中插入元组（S6，Athens）的请求会成功，其结果是在 *S* 中插入（S6，*n*，*t*，London），*n* 和 *t* 分别是在 SNAME 和 STATUS 上的默认值。
- 在 SC 上插入元组（S1，Athens）的请求会失败，因为它违反了 *S* 上的谓词（因此也违反了 SC 上的谓词）——尤其是，它违反了候选码 {S#} 的唯一性约束。
- 在 SC 上删除元组（S1，London）的请求成功，此元组从 *S* 中被删除。

SC

| S# | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

图 10-5　视图 SC（样本值）

- 在 SC 中将元组（S1，London）更新为（S1，Athens）的请求会成功；结果是将 *S* 中的元组（S1，Smith，20，London）更新为（S1，Smith，20，Athens）——而不是（S1，*n*，*t*，Athens），其中 *n* 和 *t* 为使用的默认值。
- 将 SC 中的元组（S1，London）更新为（S2，London）的请求会失败。（为什么?）

对于投影中不包含它所映射的关系变量上的候选码的情况——比如，*S* 在 STATUS 和 CITYY 中的投影——留作练习。

**8. 扩展**

设视图 *V* 的定义表达式为：

```
EXTEND A ADD exp AS X
```

（如前文 *PA* 是 *A* 的谓词）。*V* 的谓词 *PE* 为：

```
PA ( a ) AND e.X = exp ( a )
```

其中，*e* 是 *V* 中的一个元组，*a* 是当 *e* 的 *X* 扩展部分被移去后剩下的元组（不太严格地，可以说 *a* 是 *e* 在属性集 *A* 上的投影）。用自然语言说就是：

每一个扩展后的元组 *e* 是：（1）通过投影从 *e* 中去掉 *X* 部分而得到的元组 *a* 满足 *PA*；（2）在 *X* 部分上的值等于在 *a* 上执行 *exp* 表达式后的结果。

下面是更新规则：

- **INSERT**：设被插入的元组是 *e*；*e* 必须满足 *PE*。通过投影去掉 *X* 部分的元组 *a* 被插入到 *A* 中。
- **DELETE**：设要删除的元组是 *e*；通过投影去掉 *X* 部分的元组 *a* 被从 *A* 中删除。
- **UPDATE**：设更新前的元组是 *e*，更新后的元组是 *e'*；*e'* 必须满足 *PE*。*e* 通过投影去掉 *X* 部分的元组 *a* 被从 *A* 中删除而不引发触发过程和谓词检查，*e'* 通过从投影去掉 *X* 部分的元组 *a'* 被插入到 *A* 中。

示例：设视图 VPX 定义为

```
EXTEND P ADD ( WEIGHT * 454 ) AS GMWT
```

图 10-6 显示了这一视图的样本值。

- 插入元组（P7，Cog，Red，12，Paris，5448）的请求会成功，结果是在关系变量 *P* 中插入元组（P7，Cog，Red，12，Paris）。
- 插入元组（P7，Cog，Red，12，Paris，5449）的请求会失败。（为什么?）
- 插入元组（P1，Cog，Red，12，Paris，

VPX

| P# | PNAME | COLOR | WEIGHT | CITY | GMWT |
|---|---|---|---|---|---|
| P1 | Nut | Red | 12.0 | London | 5448.0 |
| P2 | Bolt | Green | 17.0 | Paris | 7718.0 |
| P3 | Screw | Blue | 17.0 | Oslo | 7718.0 |
| P4 | Screw | Red | 14.0 | London | 6356.0 |
| P5 | Cam | Blue | 12.0 | Paris | 5448.0 |
| P6 | Cog | Red | 19.0 | London | 8626.0 |

图 10-6　视图 VPX（样本值）

5448）的请求会失败。（为什么?）

- 删除元组 P1 的请求会成功，其结果是从关系变量 *P* 中删除元组 P1。
- 将 P1 元组更新为（P1，Nut，Red，10，Paris，4540）的请求会成功，其结果是更新 *P* 中的元组（P1，Nut，Red，12，London）为（P1，Nut，Red，10，Paris）。
- 将 P1 元组更新为 P2（其他值不变）或使 GMWT 值不是 WEIGHT 值的 454 倍的更新请求会失败。（各自的原因是什么?）

**9. 连接**

以前讲到的大部分更新处理——包括本书的前 5 版和本书作者的其他著作——认为给定连接的可更新性或不可更新性，依赖于（至少部分依赖于）这个连接的类型是一对一的、一对多的还是多对多的。和以前的方式不同，现在认为连接总是可以更新的。而且，这三种连接的规则相同，并且都很直接。这一结果初看上去虽然有些令人吃惊，但确实有道理，是由于通过**黄金法则**对这一问题进行考察而得出了这个结论。下面对它进行解释。

广义上说，提供视图支持的目的是为了使视图看起来尽可能像基本关系变量，这一目标是值得赞赏的。但是：

- 通常（暗含着）假设“对关系变量中一个元组的更新与其他元组是无关的”是可能的。
- 但是，后来又发现对关系变量中一个元组的更新与其他元组无关并不总是可以做到的。

比如，Codd 在［12.2］中说明了在某个特定的连接中仅删除一个元组是不可能的，因为这样结果会使一个关系“不再是任意两个关系的连接”（结果不可能再满足视图的谓词）。而且历史上视图更新的方法一直是简单地拒绝请求，因为它们不可能被完全看做基本关系变量上的更新。

现在的方法很不相同。确切地说，我们发现即使在一个基本关系变量中，也不可能仅更新一个元组而与其他的元组无关。因此，就接受（那些在历史上）一直被拒绝的视图更新操作，把它们解释为用逻辑上显然正确的方式来对其所映射的关系变量上的更新；接受这些更新，而且承认这些在视图上的更新可能会产生副作用——但是，为了避免违反视图上的谓词，副作用可能是必然的。

现在开始讨论细节。下面，先定义几个术语，然后陈述连接视图的更新规则，最后考虑这些规则对三种不同的情况（一对一、一对多、多对多）分别意味着什么。

考查连接 *J* = *A* JOIN *B*（第 7 章 7.4 节），其中，*A*、*B* 和 *J* 分别包括属性 {*X*，*Y*}、{*Y*，*Z*} 和 {*X*，*Y*，*Z*}。设 *A* 和 *B* 的谓词是 *PA* 和 *PB*，则 *J* 的谓词 *PJ* 是

```
PA ( a ) AND PB ( b )
```

对于此连接中指定的元组 *j*，*a* 是 *J* 中属于 *A* 的部分（即通过投影除去 *Z* 部分后得到的元组），*b* 是 *j* 中属于 *B* 的部分（即通过投影除去 *X* 部分后得到的元组）。换句话说，连接中的每一个元组，*A* 部分满足 *PA*，*B* 部分满足 *PB*。比如，对于关系变量 *S* 和 SP 在 S#上的连接有如下谓词：

对于连接中的每一个元组（*s*，*n*，*t*，*c*，*p*，*q*），有元组（*s*，*n*，*t*，*c*）满足 S 上的谓词，元组（*s*，*p*，*q*）满足 SP 上的谓词。

下面是具体的更新规则：

- **INSERT**：新元组 *j* 必须满足 *PJ*。如果 *j* 中 *A* 部分没有在 *A* 中出现，则将之插入到 *A* 中。[⊖] 如果 *j* 中 *B* 部分没有在 *B* 中出现，则将之插入到 *B* 中。
- **DELETE**：被删除元组的 *A* 部分从 *A* 中删除，*B* 部分从 *B* 中删除。
- **UPDATE**：被更新后的元组必须满足 *PJ*。原元组 *A* 部分被从 *A* 中删除，不引发触发过程和谓词检查；原元组 *B* 部分被从 *B* 中删除，也不引发触发过程和谓词检查。然后，如果

⊖ 注意，这一 INSERT 可能会有将 B 部分插入到 B 中的副作用，这和前面讨论的 UNION、DIFFERENCE 和 INTERSECTION 视图一样。相似的结论也适用于 DELETE 和 UPDATE 规则；为了简化，不必详细说明每一情况下的各种可能性。

更新后新元组的 *A* 部分没有在 *A* 中出现，则将之插入到 *A* 中；如果新元组的 *B* 部分没有在 *B* 中出现，则将之插入到 *B* 中。

下面考察这些规则对三种不同情况下的含义。

**第一种情况（一对一）**：首先，注意术语“一对一”，更准确地说应该是“（一或零）对（一或零）”。也就是说，有完整性约束可以保证对于 *A* 中的每一个元组在 *B* 中至多有一个元组与之匹配，反之亦然——这暗示着在连接上的属性集 *Y* 是 *A* 和 *B* 的超码。

示例：

- 第一个例子，可以考虑前述规则在这样一个连接上的结果：将供应商关系变量 *S*（仅有）在供应商编号上与自身进行连接。
- 第二个例子，设存在含有属性 S#和 REST 的基本关系变量 *SR*，S#用来标识供应商，REST 用来标识此供应商最喜欢的餐馆。假设在 *S* 中的供应商并不都在 *SR* 中出现。考虑连接更新规则在 *S* JOIN *SR* 上的结果。如果某些供应商在 *SR* 中出现而不在 *S* 中出现，则会有什么不同？

**第二种情况（一对多）**：术语“一对多”，准确地说应该是“（零或一）对（零或更多）”。也就是说，存在完整性约束，它保证对于 *B* 中的每一个元组在 *A* 中至多有一个元组与之匹配。它表示，连接上的属性集 *Y* 上存在一个属性集 *K*，这个 *K* 是 *A* 的一个候选码，也是匹配到 *B* 上的一个外码。注意：如果真是这种情况，就可以将“零或一”换为“正好一个”。

示例：设视图 SSP 被定义为：

```
S JOIN SP
```

（这显然是一个“外码 - 候选码匹配”连接）。图 10-7 给出了样本值：

SSP

| S# | SNAME | STATUS | CITY | P# | QTY |
|---|---|---|---|---|---|
| S1 | Smith | 20 | London | P1 | 300 |
| S1 | Smith | 20 | London | P2 | 200 |
| S1 | Smith | 20 | London | P3 | 400 |
| S1 | Smith | 20 | London | P4 | 200 |
| S1 | Smith | 20 | London | P5 | 100 |
| S1 | Smith | 20 | London | P6 | 100 |
| S2 | Jones | 10 | Paris | P1 | 300 |
| S2 | Jones | 10 | Paris | P2 | 400 |
| S3 | Blake | 30 | Paris | P2 | 200 |
| S4 | Clark | 20 | London | P2 | 200 |
| S4 | Clark | 20 | London | P4 | 300 |
| S4 | Clark | 20 | London | P5 | 400 |

图 10-7　视图 SSP（样本值）

- 在 SSP 中插入元组（S4，Clark，20，London，P6，100）的请求会成功，其结果是在 SP 中插入元组（S4，P6，100）（也就相当于在视图中插入了元组）。
- 在 SSP 中插入元组（S5，Adams，30，Athens，P6，100）的请求会成功，其结果是在 SP 中插入元组（S5，P6，100）（也就相当于在视图中插入了元组）。
- 在 SSP 中插入元组（S6，Green，20，London，P6，100）的请求会成功，其结果是在关系变量 S 中插入元组（S6，Green，20，London），在关系变量 SP 中插入元组（S6，P6，100）（也就相当于在视图中插入了元组）。

注意：假设可能会出现这样一种情况，在 SP 中存在一个元组，而在 S 中没有相对应的元组。再进一步假设在 SP 中已经包含了几个供应商号为 S6 的元组，但是它们的零件号都不是 P1。则刚讨论过的 INSERT 会在视图中插入一些额外的元组——也就是说，元组（S6，Green，20，London）和在 SP 中已有的供应商号为 S6 的元组的连接。

- 在 SSP 中插入元组（S4，Clark，20，Athens，P6，100）的请求会失败。（为什么？）
- 在 SSP 中插入元组（S1，Smith，20，London，P1，400）的请求会失败。（为什么？）
- 在 SSP 中删除元组（S3，Blake，30，Paris，P2，200）的请求会成功，其结果是从 S 中删除元组（S3，Blake，30，Paris），从 SP 中删除元组（S3，P2，200）。
- 从 SSP 中删除元组（S1，Smith，20，London，P1，300）的请求会成功（见下面的注意）其结果是从 S 中删除元组（S1，Smith，20，London），从 SP 中删除元组（S1，P1，300）。

注意：这一 DELETE 请求的结果依赖于从发货量到供应商的外码删除规则。如果这一规则指定为 RESTRICT，则所有的操作失败。如果指定为 CASCADE，它就会产生对供应商 S1 删除所有其他的 SP 元组的副作用（因此删除了所有相关的 SSP 元组）。

- 将 SSP 元组（S1，Smith，20，London，P1，300）更新为（S1，Smith，20，London，P1，400）的请求会成功，其结果是更新 SP 中元组（S1，P1，300）为（S1，P1，400）。
- 将 SSP 元组（S1，Smith，20，London，P1，300）更新为（S1，Smith，20，Athens，P1，400）的请求会成功，其结果是更新 S 中的元组（S1，Smith，20，London）为（S1，Smith，20，Athens），更新 SP 中的元组（S1，P1，300）为（S1，P1，400）。
- 更新 SSP 中元组（S1，Smith，20，London，P1，300）为（S6，Smith，20，London，P1，300）的请求会成功（参见下面的注意）结果是更新 S 中元组（S1，Smith，20，London）为（S6，Smith，20，London），更新 SP 中元组（S1，P1，300）为（S6，P1，300）。

*注意*：这一 UPDATE 请求的结果依赖于从发货量到供应商的外码更新规则。细节留作练习。

**第三种情况（多对多）**：术语“多对多”，更准确地说应该是“（零或更多）对（零或更多）”。换句话说，不存在有效的完整性约束来保证处于与“第一种情况”和“第二种情况”相同的状态。

*示例*：设存在定义如下的视图

```
S JOIN P
```

（S 和 P 在 CITY 上连接——多对多的连接）。在图 10-8 中给出样本值。

- 插入元组（S7，Bruce，15，Oslo，P8，Wheel，White，25）的请求会成功，其结果是在 S 中插入元组（S7，Bruce，15，Oslo），在 P 中插入元组（P8，Wheel，white，25，Oslo）（相当于在视图中插入了指定的元组）。

| S# | SNAME | STATUS | CITY | P# | PNAME | COLOR | WEIGHT |
|---|---|---|---|---|---|---|---|
| S1 | Smith | 20 | London | P1 | Nut | Red | 12.0 |
| S1 | Smith | 20 | London | P4 | Screw | Red | 14.0 |
| S1 | Smith | 20 | London | P6 | Cog | Red | 19.0 |
| S2 | Jones | 10 | Paris | P2 | Bolt | Green | 17.0 |
| S2 | Jones | 10 | Paris | P5 | Cam | Blue | 12.0 |
| S3 | Blake | 30 | Paris | P2 | Bolt | Green | 17.0 |
| S3 | Blake | 30 | Paris | P5 | Cam | Blue | 12.0 |
| S4 | Clark | 20 | London | P1 | Nut | Red | 12.0 |
| S4 | Clark | 20 | London | P4 | Screw | Red | 14.0 |
| S4 | Clark | 20 | London | P6 | Cog | Red | 19.0 |

图 10-8　S 和 P 在 CITY 上的连接

- 插入元组（S1，Smith，20，London，P7，Washer，Red，5）的请求会成功，其结果是在 P 中插入元组（P7，Washer，Red，5，London）——这相当于在视图中插入了两个元组，指定的元组（S1，Smith，20，London，P7，Washer，Red，5）和另一个元组（S4，Clark，20，London，P7，Washer，Red，5）。
- 插入元组（S6，Green，20，London，P7，Washer，Red，5）的请求会成功，结果是在 S 中插入元组（S6，Green，20，London），在 P 中插入元组（P7，Washer，Red，5，London）（从而在视图中插入了 6 个元组）。
- 删除元组（S1，Smith，20，London，P1，Nut，Red，12）的请求会成功，结果是从 S 中删除了元组（S1，Smith，20，London），从 P 中删除元组（P1，Nut，Red，12，London）（从而在视图中删除了 4 个元组）。

更多的例子被留作练习。

**10. 其他操作符**

最后，简要介绍一下关系代数中的其他操作符。我们关注一下连接：半连接、半差和除，它们不是原语操作，因此它们的规则可以由定义这些操作符的操作规则导出。其他的也一样：

- 重命名：不用详细讨论。
- 笛卡尔积：如在 7.4 节提到的，笛卡尔积是自然连接（如果 *A* 和 *B* 没有相同的属性，*A* JOIN *B* 就退化为 *A* TIMES *B*）的特例，因此，*A* TIMES *B* 的规则就是 *A* JOIN *B*（也是 *A* INTERSECT *B*）的一个特例。
- 合计：合计也不是原语——它是由扩展（extend）定义而来，因此它的更新规则也可以由扩展的更新规则导出来。*注意*：在实际中，大部分 SUMMARIZE 视图上的 UPDATE 操作都会失败。不过，这些失败不是因为这些视图因继承而来的不可更新性，而是因为这些更新与完整性约束的冲突。比如，设视图定义为：

```
SUMMARIZE SP BY { S# } ADD SUM ( QTY ) AS TOTQTY
```

对于删除供应商 S1 的元组的请求会成功；但是更新元组（S4，900）为（S4，800）的请求会失败。因为它违反了约束“TOTQTY 值必须等于单个 QTY 值的和”；插入元组（S5，0）的请求会失败，但原因与上一个不同。（为什么?）

- 分组与解组：与“合计”相似［3.3］。
- Tclose：也和上面的有些相似。

## 10.5 快照

本节将讨论**快照**［10.1］。快照与视图有点相似，但又有所区别。与视图一样，快照是导出的关系变量；但与视图不同的是，它们是真正的关系变量，而不是虚拟的——也就是说，快照不是通过在其他关系变量上的定义来表示自身，而是（至少在理论上是）通过它们各自的物化的数据备份。比如：

```
VAR P2SC SNAPSHOT
    ( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY }
    REFRESH EVERY DAY ;
```

定义一个快照就像是执行一个查询，但不同的是：

a）查询的结果是以一个特定的名字保存在数据库中（本例中是 P2SC），它是只读的关系变量（只读，但是会被定期刷新，见 b）。

b）快照被定期**刷新**（比如每天）——也就是，它当前的值被丢掉，重新执行查询，新的结果成为快照的新值。

这样，快照 P2SC 就会和 24 小时前一样表示相关的数据。（它的谓词是什么?）

快照的意义在于，很多应用（甚至可能绝大部分）可能容忍，或是需要与某个确切时间相近的数据就可以了。报表和会计就是这类应用；这类应用的典型要求是数据被冻结在某一适当的时刻（比如进行会计统计的一段时期），快照可以使这样的数据冻结而又不影响其他事务在这些数据上的更新操作（在真实数据上）。相似地，它可以为一个查询的大量数据或一个只读的应用服务，而不封锁数据库，这是非常有用的。*注意*：在分布式数据库或决策支持环境中（分别参见第 21 章和 22 章）这一想法非常有吸引力。可以说快照是“受控冗余”（controlled redundancy）的一种特殊情况（见第 1 章），“快照刷新”就是相应的更新过程（也见第 1 章）。

一般来说，快照定义的语法如下：

```
VAR <relvar name> SNAPSHOT <relation exp>
    <candidate key def list>
    REFRESH EVERY <now and then> ;
```

*<now and then>* 应该是“月、周、天、小时或是 *n* 分钟，还可以是周一、周末等”（特别地，用 REFRESH［ON］EVERY UPDATE 可以保持快照始终与它所导出的关系变量同步）。下面是删除快照的语法：

```
DROP VAR <relvar name> ;
```

显然，*<relvar name>* 是用来指明快照的。*注意*：假设当一个快照定义被其他关系变量定义所引用时，对这个快照的删除请求会失败。相应地，可以扩展快照定义使之包括“RESTRICT”或“CASCADE”选项。这里不再深入讨论后一个可能性了。

*术语的注释*：事实上，完成此书的时候，快照被认为是**物化视图**[⊖]（参见第 22 章的参考文献一节而非快照。）但此术语很不合适，在笔者看来也是应该坚决更正的。快照并不是视图，视图

⊖ 一些作者（不是所有的作者）认为物化视图就是经常更新的视图（即每天进行更新操作）。

不是物化的，至少具有模型的概念。（它们是否被物化取决于实施的过程，而不是模型本身。）换句话说，物化视图相对于模型而言是一个完全相反的概念，物化视图是一个普遍存在的概念，以至于视图的概念通常被认为是物化视图。因此，当提及原始的视图概念的时候我们没有更加合适的术语。当我们使用术语视图的时候，也许经常会被误解。本书中将不会使用物化视图这个概念（除非引用它文），对于快照概念尚待商榷，我们将使用视图最原始的概念。

## 10.6　SQL 对视图的支持

本节总结一下 SQL 对视图的支持。（在写作本书时，SQL 还不支持快照。）首先，创建视图的语法是（出于简短性的考虑，例如可以将视图定义为某种结构化类型，我们这里忽略了选项和方法的变化）：

```
CREATE VIEW <view name> AS <table exp>
     [ WITH [ <qualifier> ] CHECK OPTION ] ;
```

解释：

1）<*table exp*>是视图定义表达式。

2）如果指明了 WITH CHECK OPTION，则它表示，在此视图上的 INSERT 或 DELETE 若违反了任何视图定义表达式所蕴含的完整性约束，就会被拒绝。注意，只有当 WITH CHECK OPTION 被指明时，这样的操作才会失败——也就是说，在默认情况下它们不会失败。在 10.4 节中，我们把这一行为看作是逻辑上不正确的；因此强烈建议在实际中 WITH CHECK OPTION 总是被指明（参见［10.5］）。○

3）<*qualifier*>可以是 CASCADED 或 LOCAL，并且 CASCADED 是默认值（也是唯一合理的选项，这一点在参考文献［4.20］中进行了详细论述；这里也不再深入讨论 LOCAL）。

以下是一些 10.1 节中的 SQL 视图定义：

1）
```
CREATE VIEW GOOD_SUPPLIER
     AS SELECT S.S#, S.STATUS, S.CITY
        FROM   S
        WHERE  S.STATUS > 15
     WITH CHECK OPTION ;
```

2）
```
CREATE VIEW REDPART
     AS SELECT P.P#, P.PNAME, P.WEIGHT AS WT, P.CITY
        FROM   P
        WHERE  P.COLOR = 'Red'
     WITH CHECK OPTION ;
```

3）
```
CREATE VIEW PQ
     AS SELECT P.P#, ( SELECT SUM ( SP.QTY )
                       FROM   SP
                       WHERE  SP.P# = P.P# ) AS TOTQTY
        FROM   P ;
```

SQL 认为该视图不可更新，因此 WITH CHECK OPTION 必须省略。

4）
```
CREATE VIEW CITY_PAIR
     AS SELECT DISTINCT S.CITY AS SCITY, P.CITY AS PCITY
        FROM   S, SP, P
        WHERE  S.S# = SP.S#
        AND    SP.P# = P.P# ;
```

SQL 认为该视图不可更新，因此 WITH CHECK OPTION 必须省略。

5）
```
CREATE VIEW HEAVY_REDPART
     AS SELECT RP.P#, RP.PNAME, RP.WT, RP.CITY
        FROM   REDPART AS RP
        WHERE  RP.WT > 12.0
     WITH CHECK OPTION ;
```

存在的视图可以用 DROP VIEW 语法来删除：

```
DROP VIEW <view name> <behavior> ;
```

---

○ 这是更新视图。在随后我们会发现，SQL 的视图很少更新，检查点对于不更新的视图而言是不合法的。

< *behavior* > 选项可以是 RESTRICT 或 CASCADE。如果指明为 RESTRICT，而且此视图在其他视图定义中被引用或在某个完整性约束中被引用，则 DROP 操作会失败；如果指明为 CASCADE，则 DROP 操作成功，参照它的视图定义和完整性约束也会被删除。

**视图检索**

正如 10.3 节所述，当前的 SQL 标准（SQL: 1992）保证所有的视图检索正确工作。不幸的是，现在的数据库产品做不到这些，也不能达到 SQL 的早期版本的要求。

**视图更新**

SQL/92 标准对视图更新的支持有限。同时这也非常难以理解。事实上，这个标准相对于通常情况而言更难理解和接受。[㊀] 这里有一个对于标准的摘录：

< query expression > QE1 被更新当且仅当它包含每一个 < query expression > 或 < specificaion > QE2：

a) QE1 包含 QE2，不存在任何 < non join query expression > 的中间层，如 UNION DISTINCT，EXCEPT ALL，或者 EXCEPT DISTINCT。[㊁]

b) 如果 QE1 仅包含一个具体指定为 UNION ALL 的 < non join query expression > NJQE，则

i. NJQE 包含 < query expression > LO 和 < query term > RO，没有一个节点同时存在于 LO 表和 RO 表中。

ii. NJQE 中的每一列分别存在于 LO 和 RO 中，要么同时被更新，要么同时不被更新。

c) QE1 包含 QE2，但是不涉及 < non join query term > 。

d) QE2 是可更新的。

注意，(a) 上述内容是通常要考虑到的一个规则，用以决定一个视图是否能够被更新；(b) 规则并非所有地方都会用到，但是在文件的不同部分将会提及；(c) 规则依赖于附加的各种概念和架构，如列更新操作，< non join query term > 等，这些将在后文中进一步定义。

出于这样的考虑，在这里我们不去试图给出 SQL 中哪一个视图是可以更新的。不严格地讲，我们可以认为 SQL 中以下的视图可以被更新：

1) 视图定义为受限视图，并且/或者基于一个独立的数据库表。

2) 视图的定义是基于两个数据库基表的一对一或者一对多连接（对于一对多连接的情况，只有“多”的一方可以被更新）。[㊂]

3) 通过 UNION ALL 或者 INTERSECT 定义的视图。

4) 1、2、3 的组合情况。

而且，如果这些情况没有正确被处理的话，幸好 SQL 没有谓词的限制，偶尔的情况下 SQL 允许出现同样的两条记录。而且，由于 SQL 定义了四种不同的情况，使得结果更加复杂。具体而言，一个给定的视图可以被更新、简单地更新或者插入[㊃]（可更新操作参照 UPDATE 和 DELETE；可插入操作参照 INSERT，除非视图是可更新的，否则无法插入。）

对于情况 1 而言，我们要特别注意。具体说，一个 SQL 视图如果满足以下所有条件的话，

---

㊀ 引用参考文献 [10.11]：“SQL 标准仍然是视图更新开发方法（除了实施）的障碍”。

㊁ 我们没有提及第 8 章中的论点，但是 SQL/99 增加了指定一个明确的 DISTINCT 修饰语作为 UNION，INTERSECT，EXCEPT 操作上 ALL 的替换的能力。同样地，修饰语 ALL 也可以被指定为 SELECT 操作上 DISTINCT 的替换。但是要求意，DISTINCT 是 UNION，INTERSECT 和 EXCEPT 操作的默认值，而 ALL 是 SELECT 的默认值。

㊂ 结合一对一的连接，我们讨论下面的奇怪现象。SQL 非常正确地要求在这样的连接上的更新操作必须是全都的或完全不执行的。但是这种要求（如同要求大体上所有的更新都是全部的或完全不执行的，即使它们包含参照行为，例如级联删除）意味着，系统不得不支持某种多重关系任务，尽管事实上 SQL 不包含对任何这种操作符的明确的支持。

㊃ 这个标准正式定义了这些术语，但是没有说明这些术语直观的含义以及它们为什么会被挑中。注意对 10.4 节“关于视图更新机制”小节中第 9 条原则和第 10 条原则的违反的情况。

它才是可更新的：

1）定义视图范围表的表达式是一个选择表达式；也就是说，它不直接包含以下关键词：JOIN、UNION、INTERSECT 或 EXCEPT。

2）选择表达式的选择语句不直接包含关键词 DISTINCT。

3）选择语句（可能包括星号作为选项）中的每一个选择项包含一个合适的列名（可以伴着 AS 子句），表示所对应的表的一个列（参见第 5 条）。

4）选择表达式的 FROM 子句只包含一个表的参照。

5）这个表的参照用来标识一个基本表或是一个可更新的视图。注意：这个用表的参照来标识的表就是这个可更新视图所指向的表（参见第 3 条）。

6）这个选择表达式不包含这样一个 WHERE 子句：这个 WHERE 子句包含一个子查询，其中 FROM 所指向的表与第 4 条中提到的 FROM 子句指向的表是同一个表。

7）此选择表达式不包含 GROUP BY 子句。

8）此选择表达式不包含 HAVING 子句。

## 10.7　小结

**视图**实际上是一个命名了的关系表达式；它可以被看做一个**导出的、虚拟的关系变量**。在视图上的操作事实上是被一组**替换**过程完成的；也就是说，对这一视图名的引用是被定义这个视图的表达式所替换了——并且由于**封闭性**，这一替换过程能够准确地工作。对于检索操作，这组替换过程百分之百有效。（虽然实际产品中做不到，但理论上是这样的。）对于**更新操作**，也是百分之百有效；（也是在理论上，虽然目前的产品肯定做不到。）但对某些视图（如定义的合计视图），更新会由于违反完整性**约束**而失败。我们列出了更新模式所要遵循的原则的一个超集，也列出了更新模式在用 **UNION**、**INTERSECTION**、**DIFFERENCE**、**RESTRICT**、**PROJECT**、**JOIN** 和 **EXTEND** 操作符定义的视图上如何工作。对于它们中的每一个，也详细讨论了对它们所映射的关系变量的**谓词引用规则**。

本章考察了视图和**逻辑上的数据独立性**的问题。对于这种独立性有两方面问题：**可成长性**和**可重构性**。视图的其他方面还包括它能隐藏数据的能力，这样它就能够提供一种**安全性**手段；它有用作快捷方式的能力，这可使用户更轻松。接下来又解释了两个重要的准则，**互变准则**（它表示无论情况如何，视图总是可更新的）和是**数据相对性准则**。

本章还对**快照**进行了简单的讨论（通常被认为是物化视图，尽管这种看法不对）。最后，对 SQL 在相关方面进行了概略的描述。

## 习题

10.1　为在 London 的供应商定义一个视图。

10.2　定义一个视图，这一视图包括了“供应商和他所提供的零件不在同一地点”的供应商号和零件号。

10.3　从供应商 - 零件 - 工程数据库定义中的关系变量 SPJ，定义在供应商 - 零件数据库中的视图 SP。

10.4　在供应商 - 零件 - 工程数据库上定义视图，使它包含所有由 S1 提供零件而且使用零件 P1 的工程（只要求工程编号和城市属性）。

10.5　对于下面的视图定义

```
VAR HEAVYWEIGHT VIEW
  ( ( P RENAME ( WEIGHT AS WT, COLOR AS COL ) )
      WHERE WT > WEIGHT ( 14.0 ) ) { P#, WT, COL } ;
```

请给出对下列语句实施替换过程后的转化形式：

```
a. HEAVYWEIGHT WHERE COL = COLOR ('Green')

b. ( EXTEND HEAVYWEIGHT ADD ( WT + WEIGHT ( 5.3 ) ) AS WTP )
                                                    { P#, WTP }

c. INSERT HEAVYWEIGHT
   RELATION { TUPLE { P#  P# ('P99'),
```

```
                        WT  WEIGHT ( 12.0 ),
                        COL COLOR ('Purple') } } ;
```

d.
```
DELETE HEAVYWEIGHT WHERE WT < WEIGHT ( 10.0 ) ;
```

e.
```
UPDATE HEAVYWEIGHT WHERE WT = WEIGHT ( 18.0 )
       { COL := 'White' } ;
```

10.6 设练习 10.5 中的视图 HEAVYWEIGHT 的定义作如下修改：

```
VAR HEAVYWEIGHT VIEW
( ( ( EXTEND P ADD ( WEIGHT * 454 ) AS WT )
    RENAME COLOR AS COL ) WHERE WT > WEIGHT ( 6356.0 ) )
                                          { P#, WT, COL } ;
```

（也就是说，属性 WT 现在是以克为单位的重量，而不再是以磅为单位）。现在，重复练习 10.5。

10.7 对于 10.1 节中的基于代数的视图定义，给出其对应的基于演算的定义。

10.8 在视图定义中，ORDER BY 没有意义（尽管事实上在知名的数据库产品中至少有一个是允许使用 ORDER BY 的）。这是为什么？

10.9 在第 9 章中说过，有时希望能够在视图上定义候选码，或者是主码。为什么需要这一手段呢？

10.10 为了支持视图，系统目录需要扩展，系统需要哪些扩展呢？对于快照又如何？

10.11 设一个指定的关系变量 *R* 可以被两个限制 *A* 和 *B* 代替，*A* 和 *B* 满足 *A* UNION *B* 等于 *R*，并且 *A* INTERSECT *B* 等于空。这种情况下是否有逻辑上的数据独立性？

10.12 如果 *A* 和 *B* 是同一种关系类型，如果存在 *A* INTERSECT *B* 等于 *A* JOIN *B*（JOIN 是一对一的，但并非严格要求，因为在 *A* 中存在的元组不一定在 *B* 中有与之相对应的元组，反之亦然。），那么在 10.4 节中给出的对于 INTERSECTION 视图和 JOIN 视图的可更新性规则是否还适用于这种等价的形式？

10.13 如果 *A* INTERSECT *B* 还等于 *A* MINUS（*A* MINUS *B*），也等于 *B* MINUS（*B* MINUS *A*），那么 10.4 节中给出的对于 INTERSECTION 视图和 DIFFERENCE 视图的可更新性规则是否还适用于这种等价的形式？

10.14 在 10.4 节中给出了一条准则：INSERT 和 DELETE 是互反的操作。那么同样在那一节中讲到的对于 UNION、INTERSECTION 和 DIFFERENCE 视图的更新规则是否也遵守这一准则呢？

10.15 在 10.2 节中讲到了重构供应商 - 零件数据库的可能性，这是用关系变量 *S* 的两个投影 SNC 和 ST 代替基本关系变量来实现的。而且这一重构不是无关紧要的，这是什么意思呢？

10.16 对于你可利用的任一种 SQL 产品：

a）能否找到一个使视图检索失败的例子？

b）视图更新的规则是什么？（它们可能不如在 10.6 节中所讲到的 SQL/99 那么严格。）

10.17 考察供应商 - 零件数据库，为了简化，忽略零件关系变量。下面是对于供应商和发货的两个可能的设计概要：

a.
```
S   { S#, SNAME, STATUS, CITY }
SP  { S#, P#, QTY }
```

b.
```
SSP { S#, SNAME, STATUS, CITY, P#, QTY }
XSS { S#, SNAME, STATUS, CITY }
```

a 方案是常规设计方法。在方案 b 中，关系变量 SSP 包含了每一个发货元组，并给出了零件号、数量和供应商的全部细节；关系变量 XSS 包含了所有不提供任何零件的供应商的详细内容（注意：这两个方案是信息等价的，而且体现了互换性准则）。给出方案 a 和 b 的视图定义表达式，说出每一方案的数据库约束（关于数据库约束的内容请参见第 9 章）。是否每一方案都有相对于另一方案的优势？如果有的话，这一优势是什么？

10.18 给出练习 10.1 ~ 10.4 的 SQL 解答。

10.19 10.4 节中给出的关于更新视图的运算法则是比较严格的，比如删除一个由供应商和运输商连接所组成的元组意味着仅仅删除运输商相应的部分，供应商将不被从 S 中删除。请加以讨论。

10.20 重新考虑，3.2 节给出的关系模型的定义，并对此有一个充分的理解。

## 参考文献

[10.1] Michel Adiba: "Derived Relations: A Unified Mechanism for Views, Snapshots, and Distributed Da-

ta," Proc. 1981 Int. Conf. on Very Large Data Bases, Cannes, France (September 1981). See also the earlier version "Database Snapshots," by Michel E. Adiba and Bruce G. Lindsay, IBM Research Report RJ2772 (March 7, 1980).

这篇论文首次提出快照的概念，并对其语义和实现进行了讨论。关于实现，着重论述了各种差别刷新（differential refresh）和增量维护（incremental maintenance）的可能——在每一次刷新时，系统并不总是要重新执行全部的原始查询。

[10.2] H. W. Buff: "Why Codd's Rule No. 6 Must Be Reformulated," *ACM SIGMOD Record 17*, No. 4 (December 1988).

1985 年，Codd 发表了一个 12 条的准则集，用来"检测一个声称是完全关系的数据库是不是真的如此 [10.3]"。其中第六条准则要求，"所有理论上可更新的视图"实际上也可以被系统更新。Buff 在这篇文章中称一般的视图可更新性问题都是不可预测的——也就是不存在一般的算法来检测任意一个视图的可更新性（Codd 的说法）。（根据 McGoveran [10.11]，这篇文章"已经成为调查研究视图更新问题的主要并且是最严重的障碍。）

但是任何一个真正相关的实施都将受到各种各样的限制（如表达式的最大长度），Buff 的结果不能运用到具体的系统中。再一次引用 McGoveran 的话："限制实施的 Buff 对于减少关系模型是必要的；他的论文只是考虑到单纯的数学抽象以及理论算法。"

[10.3] E. F. Codd: "Is Your DBMS Really Relational" and "Does Your DBMS Run by the Rules?" *Computer world* (October 14 and 21, 1985).

[10.4] Donald D. Chamberlin, James N. Gray, and Irving L. Traiger: "Views, Authorization, and Locking in a Relational Data Base System," Proc. *NCC 44*, Anaheim, Calif. Montvale, N. J.: AFIPS Press (May 1975).

包括 System *R* 原型中用于视图更新的方法的基本原理（因此，也在 SQL/DS、DB2 和 SQL 等标准中）。见参考文献 [10.12]，它讨论的是 University Ingres 原型。

[10.5] Hugh Darwen: "Without Check Option," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

[10.6] C. J. Date and David McGoveran: "Updating, Union, Intersection, and Difference Views" and "Updating Joins and Other Views," in C. J. Date, *Relational Database Writings 1991 - 1994*. Reading, Mass.: Addison-Wesley (1995).

这两篇论文以非正式的形式介绍了视图的更新机制，如 10.4 节所述。一个作者（McGoveran）基于 [10.11] 的描述提出了一个正式的机制说明。

[10.7] Umeshwar Dayal and Philip A. Bernstein: "On the Correct Translation of Update Operations on Relational Views," *ACM TODS 7*, No. 3 (September 1982).

早期用来处理视图更新问题的方法（只用于 restriction、projection 和 join 视图）。但不考虑关系变量谓词。

[10.8] Antonio L. Furtado and Marco A. Casanova: "Updating Relational Views," in reference [18.1].

在处理视图更新问题上有两个广义的方法。一个是试图提供与具体涉及的数据库无关的通用机制（本书只对这种机制进行了详细论述）；它纯粹由视图定义（即能够被系统理解的语义）驱动。另一个方法没有这么随意，它要求 DBA 为每个视图指定哪些更新是允许的并，给出语义说明，根据所参照的基本关系变量来写出实施更新操作的执行代码。这篇论文考查了这两种方法的工作情况（1985 年）。文中大量引用了早期的书籍。

[10.9] Nathan Goodman: "View Update Is Practical," *InfoDB 5*, No. 2 (Summer 1990).

对于视图更新问题的非正式的讨论。下面是其导言中的一段稍加解释过的摘要："Dayal 和 Bernstein [10.7] 已经证明，凡是我们感兴趣的视图都不能被更新；Buff [10.2] 证明了不存在用来推断任意视图是否可更新的算法。好像没有希望了，但实际上，视图更新既是可能的也是可行的"。文中给出了大量视图更新的特殊技巧，但是，谓词这个关键的概念并没有被提及。

[10.10] Arthur M. Keller: "Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins," Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, Ore. (March 1985).

提出了视图更新算法应该遵守的五条原则——没有副作用，一步仅做一个变动，没有不必要的变动，没有更简单的其他方法，不用 DELETE-INSERT 对来替代 UPDATE——并给出了满

足这五条原则的算法。这一算法可以将一种操作转化为另一种操作；比如，在某个视图上的 DELETE 操作可以翻译成在视图所参照的基本关系变量上的 UPDATE 操作（例如，要在“供应商”视图上删除一个元组，方法可以是将 London 改为 Paris）；再举一个例子（超出了 Keller 这篇文章的范围），在视图 *V* 上的 DELETE（*V* 定义为 *A* MINUS *B*）可以这样实现：在 *B* 上插入一个元组，而不是在 *A* 上删除元组。注意，由于第 6 条原则，我们显然不会使用这样的算法。

[10.11] David O. McGoveran: "Accessing and Updating Views and Relations in a Relational Database," U. S. Patent Application 10/114,609 (April 2,2002).

[10.12] M. R. Stonebraker: "Implementation of Views and Integrity Constraints by Query Modification," Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1975).

见参考文献 [10.4] 的注释。

# 第三部分　数据库设计

这一部分主要介绍数据库的设计（更确切地说是关系数据库的设计），数据库设计问题可以简单地描述为：如果要把一组数据存储在数据库中，该如何为这些数据设计一个合适的逻辑结构？换句话说就是，如何决定存在哪些关系变量，以及各个关系变量中应该有哪些属性？这个问题的重要性是显而易见的。

在详细地讨论这个问题前，首先应该注意以下几个问题：

（1）应该注意，在这里只讨论逻辑（或者是概念）设计，而不是物理设计。当然，这并不是说物理设计不重要，相反，物理设计也十分重要，然而：

1）物理设计可以看作是逻辑设计后的一项与逻辑设计相互独立的工作。也就是说，数据库设计的“正确”方法是首先做一个纯逻辑的（例如：关系）设计，然后，作为一个单独的后续步骤，把逻辑设计映射到特定的 DBMS 支持的物理结构上（用第 2 章的话来说就是，物理设计应该在逻辑设计的基础上产生，而不是用其他方法产生）。[⊖]

2）根据定义，物理设计从某种角度来说是依赖于特定的 DBMS 的，在本书这种通用教材中不宜把它作为一个主题来讨论。而逻辑设计正相反，它是独立于 DBMS 的，并且有成型的理论可以应用。当然，这些理论也将在本书中讨论。

遗憾的是，现实世界并不完美，在实际工作中，实际的物理阶段的设计往往会对逻辑设计产生影响，（在本书中已经多次提到，现在的 DBMS 只支持从逻辑结构到物理结构的简单映射。）用另一句话说，数据库设计是逻辑设计—物理设计—逻辑设计这样一个反复进行的过程，有时必须反复多次，在此过程中有时必须作出妥协。然而，我们还是支持原先的观点：正确的数据库设计方法是首先进行数据库的逻辑设计而不考虑数据库的物理设计，因此，该书的这一部分主要讨论“首先如何使数据库的逻辑设计正确”。

（2）虽然所讨论的主要是关系数据库的设计，但是即将讨论的这些思想和非关系数据库也是密切相关的。也就是说，在非关系数据库的设计中，正确的方法是首先作一个正确的关系设计，然后作为一个单独的步骤，把关系设计映射到任何一个 DBMS 支持的非关系结构（如层次结构）上。

（3）数据库设计应该说是一种艺术而不是一种科学。尽管有一些科学理论可以应用到这个问题上，而且这些科学理论也是后面三章讨论的对象；然而，有很多很多的设计问题在这些科学规则中根本没有提到。因此，很多数据库理论家和从业者提出了数据库设计方法学（design methodology），从某种程度来说，它们都可以用来处理到目前来说还比较难处理的问题，即找出一个合理的逻辑设计，但是这些方法有些相当严格，另一些则不然。因为这些方法在一定程度上只适用于某些特定场合，所以没有客观的标准来判断选择哪种方法较好。然而，在第 13 章中介绍一种众所周知的方法，这种方法因为它的许多优点而被广泛使用。在该章中还简单介绍了一些商业上支持的方法。

（4）需要说明的是，以下两个假设是这一部分讨论的基础：

1）数据库设计不仅仅是得到一个正确的数据结构，数据完整性是数据库设计的关键要素之

---

⊖ 事实上，在理想情况下系统应该能够自动地从物理设计中得到，在这个过程中根本不需要人的插手。尽管这个目标听起来像是空想，附录 A 描述了一种使之成为可能的方法。

一。这一点将在以后的各个章节中不断地重复和强化。

2）主要讨论具有应用独立性的设计。也就是说，主要讨论数据本身的问题，而不是数据怎样被使用的问题。应用独立性之所以重要，是因为在设计阶段不可能知道对数据的所有使用方式。人们总是希望自己的设计是鲁棒的，即不希望设计的数据库在遇到一个在设计阶段没有考虑到的应用需求时把它当做非法数据。换句话说（采用第 2 章的术语），人们所要做的主要是使“概念”模式正确，即设计一个独立于硬件、操作系统、DBMS、语言及用户的抽象的逻辑结构，而对于前面所说的因为实现问题而做出的妥协则不感兴趣。

（5）前面已经说过，数据库设计主要是确定在数据库中应有哪些关系变量，以及每个关系变量中应该有哪些属性。事实上，还要确定应该定义哪些域或类型，但是本书没有对这个主题作过多的介绍，因为到目前为止有关这个问题所做的研究还不多（［14.12］和［14.44］除外）。

这一部分的结构安排如下：第 11 章提出一些基础理论，第 12、13 两章涉及规范化思想，该思想直接建立在前面介绍的理论基础上，目的是给非形式化的声明一个形式化的意义，以便说明某种设计在某些方面比另一种设计“好”。第 14 章介绍语义建模（semantic modeling），特别介绍了“实体/关系”（entry/relationship，ER）模型的概念，并介绍这个概念如何运用于自上而下的数据库设计问题。（从现实世界的实体开始，以规范的关系设计结束。）

# 第11章 函数依赖

## 11.1 引言

本章首先介绍一下基本概念——**函数依赖**。参考文献［11.7］中把这一概念描绘为“不是很基础，但十分接近基础的”概念。这个概念对以后各章要讨论的几个问题（特别是第12章要讨论的数据库设计理论）都非常重要。但是注意，它的有用性并不仅限于这一用途；实际上，本章本来也可以包含在第二部分而不是第三部分。

函数依赖主要是指给定关系变量中一个属性集和另一个属性集间的多对一关系。例如，在发货关系变量SP中存在由属性集｛S#，P#｝到属性集｛QTY｝间的函数依赖，它的意思是，对于关系变量SP的任意一个合法值（关系）：

1）对于任意给定的属性对S#和P#的值，只有一个QTY的值与之对应。⊖

2）但是，可以存在许多S#和P#的不同的值，而它们所对应的QTY的值相同。

注意：我们通常所举的SP的例子（见图3-8）确实满足以上两个条件；我们又多了一个依赖于元组相等定义的概念。

在11.2节中，我们将进一步介绍函数依赖的概念，严格区分那些只在某些特定的条件下才能满足给定的关系变量的函数依赖和在任何条件下都能满足给定的关系变量的函数依赖。已经提到过，函数依赖是科学解决许多实际问题的基础，其原因是函数依赖具有一些有趣的形式化的特性，这使得可以用一种形式化的、严格的方法处理所讨论的问题。11.3节至11.6节详细介绍了这些形式化特性及它们在实际工作中的重要性。11.7节给出简单总结。

注意：这是本书中最形式化的一章，可能在第一次阅读的时候，你想要跳过其中的一些部分。实际上，为了理解后面三章内容所应了解的大部分概念在11.2节和11.3节做了介绍，所以第一次阅读本书时，对其余部分可以只简单地看看，等消化吸收了后面三章的内容后再回过头来仔细阅读这些内容。

## 11.2 基本概念

为了解释本节的一些概念，我们把发货关系变量稍作修改，使它除了含有原来的属性：S#（供应商编号）、P#（零件编号）和QTY（发货量）外，增加一个属性CITY（城市），该属性表示供应商的地址，为了防止混淆，把这个关系变量称为SCP。关系变量SCP的一个可能的值见图11-1。

现在，有必要分清楚以下两种不同的情况：（a）给定的关系变量在某一特定时间的值；（b）给定关系变量在不同时候所有可能的值。首先根据情况a讨论函数依赖，然后，把函数依赖的概念扩展到情况b。下面是情况a的定义：

- 函数依赖，情况a：假设 $r$ 是一个关系，$X$ 和 $Y$ 是 $r$ 的属性集的任意子集。当且仅当 $r$ 中任一给定的 $X$ 的值，在 $r$ 中存在一个唯一的 $Y$ 与之对应。也就是说，如果 $X$ 相等，$Y$ 也相等，则 $Y$ 函数依赖于 $X$，表示为

  $X \rightarrow Y$

| SCP | S# | CITY | P# | QTY |
|---|---|---|---|---|
| | S1 | London | P1 | 100 |
| | S1 | London | P2 | 100 |
| | S2 | Paris | P1 | 200 |
| | S2 | Paris | P2 | 200 |
| | S3 | Paris | P2 | 300 |
| | S4 | London | P2 | 400 |
| | S4 | London | P4 | 400 |
| | S4 | London | P5 | 400 |

图11-1 关系变量SCP的一个实例

⊖ 注意，这种陈述正确，是因为某种“商业规则”有效（见第9章）——即对一个给定供应商和一种给定零件，在任一给定时间，至多只有一次供货。也就是说，函数依赖表达的是语义信息（数据的意义），而不是由在某个特殊时间点上恰巧出现在数据库中的一些特殊值而产生的偶然事件。

（读作 *X* **函数决定** *Y*，或简单读作 *X* 指向 *Y*。）

例如，图 11-1 所示的关系满足下述函数依赖：

```
{ S# } → { CITY }
```

因为这个关系的任一给定的 S#值都有一个给定的 CITY 与之对应。事实上，该关系还满足下列函数依赖：

```
{ S#, P# } → { QTY }
{ S#, P# } → { CITY }
{ S#, P# } → { CITY, QTY }
{ S#, P# } → { S# }
{ S#, P# } → { S#, P#, CITY, QTY }
{ S# }     → { QTY }
{ QTY }    → { S# }
```

（练习：检查这些函数依赖的正确性。）

函数依赖表达式的左边和右边有时分别称为**自变量**（determinant）和**应变量**（dependent）。根据定义，自变量和应变量都是属性集。当某个属性集只含有一个属性时，即单元素集（singleton set）时，可以把花括号对省去，例如：

```
S# → CITY
```

前面已经解释过，上面的定义只适用情况 a，即只适用于单独的一个关系。然而当考虑关系变量，特别是基本关系变量时，人们所感兴趣的就不是对关系变量的某个或某些关系适用的函数依赖，而是对关系变量所有可能的值都适用的函数依赖。例如在 SCP 中，函数依赖：

```
S# → CITY
```

对 SCP 的所有可能值都适用，因为在任何情况下，一个给定的供应商有一个确定的地址（CITY），所以在 SCP 中的任意两个元组，如果它的供应商编号（S#）相等，则它们的地址（CITY）相等。事实上，“任何时间”都适用的函数依赖（例如对 SCP 所有可能的值）是关系变量 SCP 的完整性约束条件——它是对 SCP 所有被认为合法的值的一个限制。下面用第 9 章的语法定义这种约束的表述：

```
CONSTRAINT S#_CITY_FD
    FORALL SCPX FORALL SCPY
       ( IF SCPX.S# = SCPY.S#
         THEN SCPX.CITY = SCPY.CITY END IF ) ;
```

（SCPX 和 SCPY 是包括 SCP 的范围变量。）语法 S#→CITY 可以看作是上述表述的简写。

（练习：给出这一约束的代数表示。）

下面是在情况 b 下函数依赖的定义（对情况 a 的扩展用黑体字表示）：

■ **函数依赖，情况 b：设 *R* 是关系变量，*X*、*Y* 是 *R* 的属性集的任意子集。当且仅当对于 *R* 的所有可能的合法值，*X* 的值和 *Y* 的值密切相关。也就是说，对于 *R* 的所有可能的合法值，当两个元组的 *X* 值相等时，*Y* 值也相等，则 *Y* 函数依赖于 *X*，表示为：**

```
X → Y
```

（读作 *X* 函数决定 *Y*，或简单读作 *X* 指向 *Y*。）

今后，我们说“函数依赖”指的是要求更加严格的、具有时间独立性的函数依赖，而不必详细说明函数依赖成立的条件。

下面是关系变量 SCP 的一些函数依赖（具有时间独立性）：

```
{ S#, P# } → QTY
{ S#, P# } → CITY
{ S#, P# } → { CITY, QTY }
{ S#, P# } → S#
{ S#, P# } → { S#, P#, CITY, QTY }
{ S# }     → CITY
```

特别要注意下列函数依赖，它们在图 11-1 的情况下成立，但并不是任何时间对关系变量 SCP 都成立。

```
S#  → QTY
QTY → S#
```

换句话说，在图 11-1 中，命题“给定供应商的每一次的发货量是相等的”是真的，但并不是对关系变量 SCP 的所有可能的合法值都为真。

如果 $X$ 是关系变量 $R$ 的候选码，则关系变量 $R$ 的任意属性 $Y$ 一定函数依赖于 $X$（这一点在 9.10 节提到过，它可以从候选码的定义中得出）。例如，在零件关系变量 $P$ 中，应该有：

```
P# → { P#, PNAME, COLOR, WEIGHT, CITY }
```

事实上，如果关系变量 $R$ 满足函数依赖 $A \rightarrow B$，而 $A$ 不是候选码，[⊖] 则 $R$ 一定存在冗余。例如在关系变量 SCP 中，表示给定的供应商（S#）一般居住在给定的城市（CITY）的函数依赖 S#→CITY 会出现多次（参见图 11-1）。下一章将对这个问题作详细的介绍。

即使只考虑任何时间都满足的函数依赖，一个给定的关系变量的完整函数依赖集还是很庞大的，如关系变量 SCP 所示（练习：给出 SCP 的完整的函数依赖集）。应该寻找一个方法把这个集合缩小到一个可管理的范围——事实上，本章的剩余部分主要讨论这个问题。

为什么这个问题值得讨论呢？原因之一是函数依赖表示某种完整性约束条件，而 DBMS 需要实现这种完整性约束条件。给定一个函数依赖集 $S$，如果能找到一个集合 $T$，$T$ 远远小于 $S$，而集合 $T$ 的函数依赖蕴涵集合 $S$ 的所有函数依赖，则 DBMS 只要实现函数依赖集 $T$，函数依赖集 $S$ 中的所有函数依赖会自动实现，因此，寻找函数依赖集 $T$ 具有实践上的重要性。

## 11.3　平凡的函数依赖和非平凡的函数依赖

*注意：在本章的剩余部分常常把“函数依赖”简称为“依赖”，同样对“函数依赖于”和“函数决定”等作相应的简化。*

缩小函数依赖集大小的一个简单方法是消除**平凡**的函数依赖，一个必须满足的函数依赖称为**平凡**的函数依赖。在前面提到的关系变量 SCP 的一个函数依赖就是平凡的函数依赖，即函数依赖：

```
{ S#, P# } → S#
```

事实上，当且仅当函数依赖的右边是左边的子集（不一定是真子集）时，该函数依赖才是平凡的函数依赖。

正如名称所示，平凡的函数依赖并没有实际意义，实际上人们所感兴趣的是**非平凡**的函数依赖，因为只有它们才和“真正的”完整性约束条件相关。然而，在讨论规范化时，必须处理所有的依赖：平凡的函数依赖和非平凡的函数依赖。

## 11.4　依赖集的闭包

前面已经提到过，有些函数依赖蕴涵另一些函数依赖。举个简单的例子，函数依赖：

```
{ S#, P# } → { CITY, QTY }
```

蕴涵下面两个函数依赖：

```
{ S#, P# } → CITY
{ S#, P# } → QTY
```

作为一个比较复杂的例子，假设一个关系变量 $R$ 有三个属性 $A$、$B$ 和 $C$，如果 $R$ 满足函数依赖 $A \rightarrow B$ 和 $B \rightarrow C$，很容易就可以发现 $R$ 同样满足函数依赖 $A \rightarrow C$。这里函数依赖 $A \rightarrow C$ 是传递函

---

⊖ 并且函数依赖是非平凡的（见 11.3 节），$A$ 不是超码（见 11.5 节），$R$ 包含至少两个元组。

数依赖的一个例子——即 $C$ 通过 $B$ 传递依赖于 $A$。

函数依赖集 $S$ 所蕴涵的函数依赖的全体称为函数依赖集 $S$ 的**闭包**，记为 $S^+$（注意：与关系代数中的闭包无关），很显然，我们需要一个算法用以从一个给定集合 $S$ 中求出它的闭包 $S^+$，对这个问题的阐述最早出现在 Armstrong［11.2］的论文里，他提出了一组**推理规则**（常被叫做 **Armstrong 公理**），通过这些推理规则，可以从给定的函数依赖中推出新的函数依赖。这些规则可以用许多等价的方法阐述，下面是其中最简单的一种：假设 $A$、$B$ 和 $C$ 是给定的关系变量 $R$ 的属性集的任意子集，并把 $A$ 和 $B$ 的并集记为 $AB$，则：

1）**自反律**（reflexivity）：如果 $B$ 是 $A$ 的子集，则 $A \rightarrow B$。

2）**增广律**（augmentation）：如果 $A \rightarrow B$，则 $AC \rightarrow BC$。

3）**传递律**（transitivity）：如果 $A \rightarrow B$ 且 $B \rightarrow C$，则 $A \rightarrow C$。

以上的每一个规则都可以从函数依赖的定义直接证明。（当然，第一条规则正好是平凡的函数依赖的定义。）并且该公理是**完备**的，即给定一个函数依赖集 $S$，该函数依赖集所有蕴涵的函数依赖都可以利用这些规则从 $S$ 中导出，另外该公理是**有效**的，即所有不是由函数依赖集 $S$ 蕴涵的函数依赖不能根据该公理系统从 $S$ 导出。也就是说，可以利用该公理系统推导 $S$ 的闭包 $S^+$。

从上面的规则可以导出其他规则（如下所示），这些规则在实际工作中可以用来简化从 $S$ 中计算 $S^+$ 的工作（在下面列出的规则中，$D$ 是关系变量 $R$ 的属性集的另外一个子集）。

4）**自含规则**（self_determination）：$A \rightarrow A$。

5）**分解规则**（decomposition）：如果 $A \rightarrow BC$，则 $A \rightarrow B$，且 $A \rightarrow C$。

6）**合并规则**（union）：如果 $A \rightarrow B$ 且 $A \rightarrow C$，则 $A \rightarrow BC$。

7）**复合规则**（composition）：如果 $A \rightarrow B$，$C \rightarrow D$，则 $AC \rightarrow BD$。

Darwen 证明了下面的定理［11.7］，该规则被称为“通用一致性定理”（General Unification Theorem）：

8）如果 $A \rightarrow B$ 且 $C \rightarrow D$，则 $A \cup (C - B) \rightarrow BD$。（“∪”表示并集，“-”表示差集。）

该定理之所以被称为“通用一致性定理”，是因为很多早期的规则可以被看做该定理的特例［11.7］。

例：假设有一个关系变量 $R$，$A$、$B$、$C$、$D$、$E$、$F$ 是它的属性，$R$ 满足下列函数依赖：

$$A \rightarrow BC$$
$$B \rightarrow E$$
$$CD \rightarrow EF$$

可以看出，这里对符号稍微作了相容的扩充，例如，用 $BC$ 表示 $B$ 和 $C$ 中的所有属性（精确地说，$BC$ 表示 $B$ 和 $C$ 的并集）。注意：如果想用更具体的例子，可以假定 $A$ 是雇员号，$B$ 是部门号，$C$ 是经理的雇员号，$D$ 是一个经理负责的工程的工程号（对于某个经理，工程号是唯一的），$E$ 是部门名称，$F$ 表示某个经理分配某个工程的时间。

从下面可以看出，在 $R$ 中函数依赖 $AD \rightarrow F$ 是成立的，且是给定函数依赖集的闭包的一个成员：

1. $A \rightarrow BC$　（给定）
2. $A \rightarrow C$　（1，分解规则）
3. $AD \rightarrow CD$　（2，增广律）
4. $CD \rightarrow EF$　（给定）
5. $AD \rightarrow EF$　（3 和 4，传递律）
6. $AD \rightarrow F$　（5，分解规则）

## 11.5　属性集的闭包

原则上讲，通过算法：反复运用前面讲的规则，直到不再产生新的函数依赖为止，就可以计算出一个函数依赖集 $S$ 的闭包 $S^+$。但是，因为这种算法的效率很低，所以在实际工作中，几乎

没有必要计算闭包本身。本节将介绍如何计算一个闭包的子集：即该子集包含所有左边是特定的属性集 Z 的函数依赖。更精确地说，给定一个关系变量 R，R 的一个属性集 Z，以及 R 的一个函数依赖集 S，从中确定 R 中所有的函数依赖于 Z 的属性集——即属性集 Z 关于函数依赖集 S 的闭包 $Z^+$。[⊖] 图 11-2 是计算该闭包的算法（练习：证明该算法的正确性）。

```
CLOSURE[Z,S] := Z ;
do "forever" ;
   for each FD X → Y in S
      do ;
         if X ⊆ CLOSURE[Z,S]
         then CLOSURE[Z,S] := CLOSURE[Z,S] ∪ Y ;
      end
   if CLOSURE[Z,S] did not change on this iteration
   then leave the loop ;          /* computation complete */
end ;
```

图 11-2　计算属性集 Z 关于 S 的闭包 $Z^+$

例：假设给定一个关系变量 R，A、B、C、D、E、F 是它的属性集，以及函数依赖集 S：

$A \rightarrow BC$
$E \rightarrow CF$
$B \rightarrow E$
$CD \rightarrow EF$

下面开始计算属性集 $\{A, B\}$ 关于函数依赖集 S 的闭包 $\{A, B\}^+$。

1）初始化：令集合 CLOSURE $[Z, S] = \{A, B\}$。

2）进行四次内循环，一次一个函数依赖。第一次循环（对于函数依赖 $A \rightarrow BC$）时发现它的左边是到目前为止计算的集合 CLOSURE $[Z, S]$ 的子集，所以把属性（B 和）C 加入集合 CLOSURE $[Z, S]$，这样，集合 CLOSURE $[Z, S]$ 就变为 $\{A, B, C\}$。

3）第二次循环（对于函数依赖 $E \rightarrow CF$）发现它的左边不是到目前为止计算的结果的子集，因此该结果保持不变。

4）第三次循环（对于函数依赖 $B \rightarrow E$），把 E 加入集合 CLOSURE $[Z, S]$，这样，CLOSURE $[Z, S] = \{A, B, C, E\}$。

5）第四次循环（对于函数依赖 $CD \rightarrow EF$），集合保持不变。

6）再一次经过四次内循环，第一次循环结果不变，第二次结果扩展到 $\{A, B, C, E, F\}$，第三次和第四次不变。

7）再一次经过四次内循环，集合 CLOSURE $[Z, S]$ 保持不变，这样，整个过程结束，结果：$\{A, B\}^+ = \{A, B, C, E, F\}$。

注意：（如所陈述的）如果 Z 是关系变量 R 的一个属性集，并且 S 是 R 所具有的函数依赖的集合，那么 R 所具有的以 Z 为左端的函数依赖的集合包含所有形如 $Z \rightarrow Z'$ 的函数依赖，其中 $Z'$ 是属性集 Z 关于 S 的闭包 $Z^+$ 的某个子集。函数依赖的原始集合 S 的闭包 $S^+$ 是所有这样的函数依赖集合的并，Z 为任意可能的属性集合。

根据前面的讨论可以得出这样一个重要结论：给定一个函数依赖集 S，可以方便地判断函数依赖 $X \rightarrow Y$ 是否可以从 S 中导出，因为当且仅当 Y 是属性集 X 关于 S 的闭包的子集时，$X \rightarrow Y$ 才能根据 Armstrong 公理由 S 中导出。也就是说，可以用一种简单的方法，该方法不必精确计算函数依赖集 S 的闭包 $S^+$，就能判断函数依赖 $X \rightarrow Y$ 是否属于函数依赖集 S 的闭包 $S^+$。

还可以得出另外一个重要的结论。在第 9 章中讲过，关系变量 R 的超码是关系变量 R 的属性集，该关系变量的一些候选码是该属性集的子集（不必是真子集）。由此可以得出以下结论：给定关系变量 R 的超码一定是 R 的属性集的一个子集 K，对于 R 中的任意属性集的子集 A 有函数依赖：

---

⊖ 注意，现在我们用到了两种类型的闭包：函数依赖集的闭包及关于函数依赖集的属性集的闭包；并且使用相同的符号“加号上标”来表示它们。我们希望这种双重的使用不会带来混淆。

$K \rightarrow A$

成立，也就是说，当且仅当 $K$ 关于给定函数依赖集的闭包 $K^+$ 是 $R$ 的所有属性的集合时，$K$ 为超码（当且仅当 $K$ 是不可约的超码时，$K$ 是候选码）。

## 11.6　最小函数依赖集

假设 $S1$ 和 $S2$ 是两个函数依赖集，如果 $S1$ 蕴涵的所有函数依赖都为 $S2$ 所蕴涵，即 $S1^+$ 是 $S2^+$ 的子集，则 $S2$ 是 $S1$ 的覆盖，[⊖] DBMS 只要实现了 $S2$ 中的函数依赖，就自动实现 $S1$ 中的函数依赖。

如果 $S2$ 是 $S1$ 的覆盖，同时 $S1$ 是 $S2$ 的覆盖，则 $S1$ 和 $S2$ **等价**，即 $S1^+ = S2^+$。很显然，如果 $S1$ 和 $S2$ 等价，则 DBMS 只要实现 $S1$ 中的函数依赖，就自动实现 $S2$ 中的函数依赖，反之亦然。

当且仅当函数依赖集满足以下条件时，该函数依赖集为最小函数依赖集：[⊖]

1）每个函数依赖的右边（应变量）只含有一个属性（即它是单元素集合）。

2）每个函数依赖的左边（自变量）是不可约的——删除自变量的任何一个属性都将改变闭包 $S^+$（即会使 $S$ 转变为一个不等价于原来的 $S$ 的集合）。这种函数依赖被称为**左部不可约的函数依赖**。

3）删除 $S$ 中任何一个函数依赖都将改变它的闭包 $S^+$，即使 $S$ 转变为一个不等价于原来的 $S$ 的集合。

关于第 2 点和第 3 点，在这里要指出的是，为了知道如果删除某些元素是否会改变闭包，没必要清楚地知道闭包的内容。例如，观察大家熟悉的零件关系变量 P，有下列函数依赖：

```
P# → PNAME
P# → COLOR
P# → WEIGHT
P# → CITY
```

显而易见，该函数依赖集是最小依赖集：每个函数依赖中右边只含有一个属性，同样，左边也是不可约的，且任何一个函数依赖都不能被删除而不改变闭包（即不丢失信息）。相反，下面的函数依赖集不是最小依赖集。

```
1) P# → { PNAME, COLOR}        第一个函数依赖的右边不是单属性集
   P# → WEIGHT
   P# → CITY
2) { P#, PNAME } → COLOR       第一个函数依赖左边的 PNAME 可以
   P# → PNAME                  删除而不改变闭包（即左边是可约的）
   P# → WEIGHT
   P# → CITY
3) P# → P#                     第一个函数可以删除而不改变闭包
   P# → PNAME
   P# → COLOR
   P# → WEIGHT
   P# → CITY
```

任何一个函数依赖集至少存在一个最小函数依赖集。假设函数依赖集为 $S$，根据分解规则，可以假定 S 中的每个函数依赖的右边是单属性的而不会失去它的一般性（如果右边不是单属性的，则可以利用分解规则把它分解成单属性）。接着考察每个函数依赖 $f$ 左边的每一个属性 $A$，如果把 $A$ 从 $f$ 的左边删除而并不改变闭包，则把 $A$ 从 $f$ 的左边删除。然后考察 $S$ 中剩余的每一个函数依赖 $f$，如果把 $f$ 删除而不改变闭包，则把 $f$ 从 $S$ 中删除。最后所得的集合 $S$ 是和原来的函数依赖集 $S$ 等价的最小函数依赖集。

---

⊖ 一些作者使用术语“覆盖”表达我们所说的（稍后）等价集。

⊖ 通常在文献中称为最小集。

例：假设给定关系变量 $R$，$A$、$B$、$C$、$D$ 是 $R$ 的属性集，$R$ 满足函数依赖：

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$
$$AC \rightarrow D$$

现在计算该函数依赖的最小函数依赖集。

1）把所有的函数依赖写成右边是单属性的函数依赖：

$$A \rightarrow B$$
$$A \rightarrow C$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$
$$AC \rightarrow D$$

很显然，函数依赖 $A \rightarrow B$ 出现了两次，可以删除其中的一次。

2）可以把 $C$ 从函数依赖 $AC \rightarrow D$ 的左边删除，因为 $A \rightarrow C$，根据增广律可以得出 $A \rightarrow AC$，给定 $AC \rightarrow D$，根据传递律可以得出 $A \rightarrow D$。所以 $C$ 在函数依赖 $AC \rightarrow D$ 的左边是冗余的。

3）接着发现可以删除函数依赖 $AB \rightarrow C$，因为 $A \rightarrow C$，根据增广律可得 $AB \rightarrow CB$，又根据分解规则可以导出 $AB \rightarrow C$。

4）函数依赖 $A \rightarrow C$ 由函数依赖 $A \rightarrow B$ 和 $B \rightarrow C$ 蕴涵，所以可以删除它。最后剩下下列函数依赖：

$$A \rightarrow B$$
$$B \rightarrow C$$
$$A \rightarrow D$$

该集合不可约。

一个函数依赖集 $I$ 是不可约的，且等价于某个函数依赖集 $S$，则说 $I$ 是 $S$ 的**最小等价依赖集**。这样，如果要实现一个函数依赖集 $S$，系统只要实现它的一个最小依赖集就足够了（重复一次：要计算最小依赖集 $I$ 不必计算闭包 $S^+$）。应该记住，给定函数依赖集的最小依赖集并不一定是唯一的（参考练习 11.12）。

## 11.7　小结

**函数依赖**是一个关系变量中的两个属性集间的多对一关系。给定关系变量 $R$，$A$ 和 $B$ 是 $R$ 的属性集的子集，当且仅当对于 $R$ 的任意两个元组，如果 $A$ 相等，则 $B$ 必相等，那么可以说 $B$ 函数依赖于 $A$，记为 $A \rightarrow B$。每一个关系变量 $R$ 一定满足某些**平凡**的函数依赖。当且仅当函数依赖的右边（**应变量**）是左边（**自变量**）的子集时，该函数依赖为平凡的函数依赖。

一个函数依赖蕴涵其他的函数依赖。给定一个函数**依赖集** $S$，$S$ 中各个函数所蕴涵的函数依赖的集合成为 $S$ 的闭包，记为 $S^+$。$S^+$ 必然是 $S$ 的超集。**Armstrong 公理系统**为计算 $S$ 的闭包 $S^+$ 提供了一个**有效**且**完备**的基础理论（然而，事实上并不用它来计算）。从 Armstrong 公理系统中可以导出其他一些有用的规则。

如果给定一个关系变量 $R$ 的属性集的子集 $Z$ 和一个该关系变量满足的函数依赖集 $S$，那么 $Z$ 关于 $S$ 的闭包 $Z^+$ 是 $R$ 的属性集的一个子集，对于该子集中的任意一个属性 $A$，函数依赖 $Z \rightarrow A$ 都成立。如果 $Z^+$ 包含 $R$ 的所有属性，则 $Z$ 为**超码**（如果超码不可约，则该超码是候选码）。本章介绍了一种计算 $Z$ 关于函数依赖集 $S$ 的闭包 $Z^+$ 的算法，从而有了一种判断给定的函数依赖 $X \rightarrow Y$ 是否是 $S^+$ 的成员的简便方法：当且仅当 $Y$ 是 $X^+$ 的子集时，$X \rightarrow Y$ 是 $S^+$ 的成员。

当且仅当两个函数依赖集 $S1$ 和 $S2$ 相互**覆盖**时，这两个函数依赖集**等价**，即当且仅当 $S1^+ = S2^+$ 时，$S1$ 和 $S2$ 等价。每一个函数依赖集至少等价于一个**最小函数**依赖集。一个最小函数依赖集应满足以下三个条件：（a）该函数依赖集的每一个函数依赖的右边只有一个属性；（b）删除该函数依赖集的任何一个函数依赖都将改变它的闭包；（c）删除该函数依赖集中任意一个函数依赖左边的任意一个属性都将改变该集合的闭包。如果 $I$ 是 $S$ 的最小依赖集，则实现 $I$ 的函数依

赖时自动实现 $S$ 的函数依赖。

最后，应该注意的是前面介绍的思想不仅仅适用于函数依赖，还适用于一般的完整性约束条件：

1）一些约束条件是平凡的。

2）一些约束条件蕴涵其他约束条件。

3）一个给定的约束条件集合所蕴涵的约束条件的集合可以认为是给定约束条件集合的闭包。

4）确定一个约束条件是否属于一个闭包的问题——即该约束条件是否被其他约束条件蕴涵的问题——是个有趣的实际问题。

5）寻找一个约束条件集合的最小依赖集是个有趣的实际问题。

只是因为有一套有效且完备的公理系统才使得函数依赖比一般的完整性约束条件更容易处理。在本章和第 13 章的“参考文献”部分给出了一些参考文献，这些参考文献中介绍了其他一些约束（“MVDs”、“JDs”和“INDs”）对这些约束规则，该公理系统仍然适用。本书不再像介绍函数依赖一样介绍这些约束条件。

## 习题

11.1　(a) 假设 $R$ 是一个 $n$ 目的关系变量，该关系变量最多可以满足多少个函数依赖（包括平凡的函数依赖和非平凡的函数依赖）？(b) 函数依赖 $A \rightarrow B$ 中的 $A$ 和 $B$ 均为属性集合，当二者之一为空集时会发生什么情况？

11.2　什么是 Armstrong 公理系统的有效性，什么是 Armstrong 公理系统的完备性？

11.3　根据函数依赖的基本定义，证明自反律、增广律和传递律。

11.4　证明习题 11.3 中的三个规则蕴涵自含规则、分解规则和合并规则。

11.5　证明 Darwen 的“通用一致性定理”，在证明时运用了前两题的什么规则？哪些规则可以看作是该定理的特例？

11.6　名词解释：(1) 函数依赖集的闭包；(2) 属性集关于给定函数依赖集的闭包。

11.7　列出关系变量 SP 中所有可能的函数依赖。

11.8　关系变量 $R$ $\{A, B, C, D, E, F, G\}$ 满足下列函数依赖：

$$
\begin{aligned}
A &\rightarrow B \\
BC &\rightarrow DE \\
AEF &\rightarrow G
\end{aligned}
$$

计算 $\{A, C\}$ 关于这个函数依赖集的闭包 $\{A, C\}^+$，该函数依赖集是否蕴涵函数依赖 $ACF \rightarrow DG$？

11.9　什么情况下说 $S1$ 和 $S2$ 等价？

11.10　什么是最小函数依赖集？

11.11　下面两个函数依赖集等价吗？

1. $A \rightarrow B$　　$AB \rightarrow C$　　$D \rightarrow AC$　　$D \rightarrow E$

2. $A \rightarrow BC$　　$D \rightarrow AE$

11.12　关系变量 $R$ $\{A, B, C, D, E, F\}$ 满足下列函数依赖：

$$
\begin{aligned}
AB &\rightarrow C \\
C &\rightarrow A \\
BC &\rightarrow D \\
ACD &\rightarrow B \\
BE &\rightarrow C \\
CE &\rightarrow FA \\
CF &\rightarrow BD \\
D &\rightarrow EF
\end{aligned}
$$

找出该函数依赖集的最小依赖集。

11.13　关系变量 TIMETABLE 定义了如下属性（字段）：

$D$　一星期中的每一天（1 ~ 5）

$P$　一天中的一个时间段（1 ~ 8）

$C$　教室号码

*T* 教师姓名

*L* 课程名

当且仅当在时间{*D*: *d*, *P*: *p*}时，教师 *t* 在教室 *c* 教授课程 *l* 时元组{*D*: *d*, *P*: *p*, *C*: *c*, *T*: *t*, *L*: *l*}才在关系变量中出现。可以假定课程持续一个时间段，一个星期里所有课程的名称唯一，那么在该关系变量中存在哪些函数依赖？它的候选码有哪些？

11.14 关系变量 NADDR 定义了如下属性（字段）：NAME（姓名，唯一）、STREET（街道）、CITY（城市）、STATE（州）和 ZIP（邮编），对于给定的邮政编码，只有唯一的州和城市与之对应。同样，给定一个街道、城市和州，只有唯一的一个邮政编码和它对应。给出该关系变量的一个最小函数依赖集和它的候选码。

11.15 上题中的假设在实践中有效吗？

11.16 关系变量 *R*{*A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*}满足下列函数依赖：

```
ABD → E
AB  → G
B   → F
C   → J
CJ  → I
G   → H
```

该函数依赖集是最小依赖集吗？给出该关系变量的候选码。

## 参考文献

如11.1节中提到的，这是本书中最规范化的一章；而参考文献［11.1］、［11.3］和［11.10］都是为多方面数据库理论提供了规范化处理的书籍（不仅是函数依赖本身），所以似乎把它们包含在这里较为合适。

［11.1］ Serge Abiteboul, Richard Hull, and Victor Vianu: *Foundations of DataBase*. Reading, Mass.: Addison-Wesley (1995).

［11.2］ W. W. Armstrong: "Dependency Structures of Data Base Relationships," Proc. IFIP Congress, Stockholm, Sweden (1974).

该文第一次用形式化的语言描述了函数依赖（它是 Armstrong 公理系统的基础），该文还精确描述了候选码这个概念。

［11.3］ Paolo Atzeni and Valeria De Antonelis: *Relational Database Theory*. Redwood City, Calif.: Benjamin/Cummings (1993).

［11.4］ Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou: "Inclusion Dependencies and Their Interaction with Functional Dependencies," Proc. I st ACM SIGACT-SIGNIOD Symposium on Principles of Database Systems. Los Angeles, Calif. (March 1982).

**内含依赖**（inclusion dependencies, IND）可以被看做参照完整性的概括，例如，内含函数依赖：

```
SP.S# ⊆ S.S#
```

（和参考书上使用的符号不同）说明出现在关系变量 SP 中的 S#的集合是出现在关系变量 S 中的 S#的集合的子集，这个例子事实上就是参照完整性，然而，内含依赖并不要求它的左边是外码或它的右边是候选码。注意：因为内含依赖和普通的函数依赖一样是反映多对一的关系，所以它们之间有共同点，但是内含依赖往往是跨越关系的，而函数依赖则是存在于一个关系变量内部的。

本文提供了一套完备且有效的内含依赖推理规则：

1） $A \rightarrow A$。

2）如果 $AB \rightarrow CD$，则 $A \rightarrow C$ 且 $B \rightarrow D$。

3）如果 $A \rightarrow B$ 且 $B \rightarrow C$，则 $A \rightarrow C$。

［11.5］ R. G. Casey and C. Delobel: "Decomposition of a Data Base and the Theory of Boolean Switching Functions," *IBM J. R&D 17*, No. 5 (September 1973).

本文指出：对于一个任意给定的关系变量，该关系变量所满足的函数依赖（该文中称为函数关系）集可以表示成"布尔转化函数"（boolean switching function）的形式，而且这种函数是唯

一的。因为虽然原来的函数依赖可以用很多表面上看起来不同但实际上是等价的形式表示，每种表示形式都可以转化成一个表面上不同的布尔函数——但这些函数可以利用布尔代数规则规约成相同的规范形式（见第 18 章）。所以，关系变量的分解问题（无损分解，见第 12 章）就和众所周知的布尔代数问题——寻找和原来的关系变量及函数依赖相对应的布尔函数的“质蕴涵的最小覆盖”（a covering set of prime implicants）——等价。从而将关系变量的分解问题转化为等价的布尔代数问题，人们就可以运用一些熟悉的技术来解决该问题。

本文和参考文献［11.8］及第 13 章的一些参考文献最先对函数依赖理论和其他理论进行了比较。

［11.6］ E. F. Codd：“Further Normalization of the Data Base Relational Model，”in Randall J. Rustin（ed.）. *Data Base Systems*, *Courant Computer Science Symposia Series 6*. Englewood Cliffs, N. J.：Prentice-Hall（1972）.

本文最早提出了函数依赖这一概念（IBM 的内部备忘除外）。本文的标题中的“规范化”指的是第 12 章介绍的数据库设计规则。该文的目的是论证函数依赖这一概念在数据库设计中的适用性。函数依赖事实上是对数据库设计问题的第一次科学思考，函数依赖本身的应用也很广泛。

［11.7］ Hugh Darwen：“The Role of Functional Dependence in Query Decomposition，”in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.：Addison-Wesley（1992）.

本文介绍了一个推理规则集，通过该规则集，可以从一个关系变量所满足的函数依赖集中推导出任意一个从该关系变量导出的关系变量所能满足的函数依赖集。通过这些函数依赖集就可以确定由原来的关系变量导出的关系变量的候选码，这就是第 9 章和第 10 章偶尔提到的候选码推导规则。本文还介绍了如何利用这些规则来显著提高 DBMS 的性能、功能及可用性。注意：SQL: 1999标准中已经使用了这些规则：（a）标准认为视图的范围可更新（见第 10 章）；（b）标准对表达式（例如 SELECT 子句中）意义的理解扩展为“每组单一值”。

```
SELECT S.S#, S.CITY, SUM ( SP.QTY ) AS TQ
FROM   S, SP
WHERE  S.S# = SP.S#
GROUP  BY S.S# ;
```

依据 SQL：1992 这一查询是非法的，因为 S. CITY 出现在 SELECT 子句，却没有在 GROUP BY 子句中出现；但是在 SQL：1999 中它是合法的，因为 SQL 能够理解关系变量 S 满足函数依赖 S#→CITY。

［11.8］ R. Fagin：“Functional Dependencies in a Relational Database and Propositional Logic，”*IBM J. R&D 21*, No. 6（November 1977）.

本文认为，Armstrong 公理系统［11.2］是和命题逻辑中的蕴涵系统严格等价的，即本文在函数依赖和命题之间定义了一个映射，然后证明当且仅当与函数依赖 $f$ 相对应的命题是与函数依赖集 $S$ 相对应的命题集的逻辑推论时，函数依赖 $f$ 才是函数依赖集 $S$ 的推论。”

［11.9］ Claudio L. Lucchesi and Sylvia L. Osborn：“Candidate Keys for Relations，”*J. Comp. and Sys. Sciences 17*, No. 2（1978）.

给出了计算一个满足给定函数依赖集的关系变量的所有候选码的算法。

［11.10］ David Maier：*The Theory of Relational Databases*. Rockville, Md.：Computer Science Press（1983）.

# 第 12 章　进一步规范化 I：1NF、2NF、3NF 和 BCNF

## 12.1　引言

本书到目前为止一直使用供应商和零件数据库作为例子，它的逻辑设计如下：

```
S  { S#, SNAME, STATUS, CITY }
   PRIMARY KEY { S# }

P  { P#, PNAME, COLOR, WEIGHT, CITY }
   PRIMARY KEY { P# }

SP { S#, P#, QTY }
   PRIMARY KEY { S#, P# }
   FOREIGN KEY { S# } REFERENCES S
   FOREIGN KEY { P# } REFERENCES P
```

注意：正如定义中所暗示的，本章中我们假设关系变量总有特定的主码（直至出现进一步的说明）。

现在需要考察该设计的正确性。很显然，S，P 和 SP 这三个关系变量是必需的，并且供应商的地址（CITY）属于关系变量 S，零件的颜色（COLOR）属于关系变量 P，发货量（QTY）属于关系变量 SP。那么我们是通过什么了解这些内容呢？通过考察改变设计的方式所引起的变化，可以对这个问题有所了解。例如，假设把供应商的地址从供应商的关系变量 S 中移到发货关系变量 SP 中（直觉上这是错误的，因为很明显，“供应商的地址”只和供应商有关，而和发货无关）。图 12-1 和第 11 章的图 11-1 稍有不同，展示了修订过的发货关系变量的一个实例。注意：为了和原来的 SP 相区分，和第 11 章一样，把修订过的发货关系变量称为 SCP。

对该图稍作研究就可以看出其中的不足，即**冗余**。例如，SCP 中每一个供应商 S1 的元组都说明 S1 居住在 London，每一个供应商 S2 的元组都说明 S2 居住在 Paris，等等。事实上，一个供应商提供几种零件，就有几个元组存放关于他的地址信息。冗余将导致许多深层次的问题。例如，修改后有可能在一个元组中供应商居住在 London，而在另外一个元组中却认为同一供应商居住在 Amsterdam[⊖]，因此最好的方法或许是“一事一地”（即避免冗余）。规范化的主要思想实际上就是使一些简单概念形式化，然而在数据库设计中，“形式化”确实具有实际应用价值。

SCP

| S# | CITY | P# | QTY |
|---|---|---|---|
| S1 | London | P1 | 300 |
| S1 | London | P2 | 200 |
| S1 | London | P3 | 400 |
| S1 | London | P4 | 200 |
| S1 | London | P5 | 100 |
| S1 | London | P6 | 100 |
| S2 | Paris | P1 | 300 |
| S2 | Paris | P2 | 400 |
| S3 | Paris | P2 | 200 |
| S4 | London | P2 | 200 |
| S4 | London | P4 | 300 |
| S4 | London | P5 | 400 |

图 12-1　关系变量 SCP 的实例

当然，就像在第 6 章所看到的一样，关系模型所涉及的关系值都是规范化的。至于关系变量，只要它的合法实例是规范化的关系，则该关系变量就是规范化的，这样，关系模型所涉及的关系变量也都是规范化的。也就是说，关系变量（和关系）总是属于**第 1 范式**（简称 1NF）的。换句话说，规范化和 1NF 的意思完全一样，虽然“规范化”（normalized）常常用来表示更高级别的标准（典型的是第三范式：3NF），后一种用法比较不规范，但很常见。

---

⊖ 本章及下一章需要假设关系变量的谓词非完全强制——因为如果它们是完全强制的，类似这样的问题就不可能发生（在一些元组中更新供应商 S1 的城市，而另一些元组不更新是不可能的）。事实上，可以这样认识规范化规则：它帮助我们以某种方式构建数据库，从逻辑上能够比其他方式（如设计不是完全规范化的）容许更多单个元组更新。如果设计是完全规范化的，谓词将比其他方式简单，所以可以达到这一目标。

这样，根据前面的说法，一个给定的关系变量有可能虽然是规范化的，但仍具有一些不受欢迎的性质，关系变量 SCP 就是一个例子（如图 12-1 所示）。进一步规范化理论使人们认识这些问题，并用一些更好的关系变量代替原来的关系变量。例如，在关系变量 SCP 中，运用规范化理论可以知道该关系变量存在一些什么“毛病”，并知道如何用两个更好的关系变量（一个含有属性 {S#, CITY}，另一个含有属性 {S#, P#, QTY}）来代替这个关系变量。

1. 范式

“进一步规范化”（further normalization）（在下文中简写为“规范化”）过程是建立在“范式”这一概念之上的。一个关系变量满足某一个范式所规定的一系列条件时，它就属于该范式。例如，当且仅当一个关系变量属于 1NF，且满足将在 12.3 节介绍的条件时，则它属于第 2 范式（2NF）。

图 12-2 展示了一些范式及它们之间的关系。前三个范式（1NF、2NF 和 3NF）由 Codd 在文献［11.6］中定义。从图 12-2 可以看出，所有的规范化关系变量都属于 1NF，有一些属于 1NF 的关系变量还属于 2NF，一些属于 2NF 的关系变量还属于 3NF。Codd 定义这些范式的原因是，他认为一个属于 2NF 的关系变量比一个只属于 1NF 的关系变量更“好”；同样，一个属于 3NF 的关系变量比一个只属于 2NF 的关系变量更“好”。因此，在数据库设计中，一般应设计属于 3NF 的关系变量，而不是设计只属于 1NF 或 2NF 的关系变量。

文献［11.6］中还介绍了过程化思想，即所谓的**规范化过程**，通过它可以用一系列更“好”的关系变量（如属于 3NF 的关系变量）代替一个“不好”的关系变量（如只属于 2NF 的关系变量）。当然该过程最初定义时只适用到 3NF，但正如下一章即将看到的，该过程经过后来的扩展可以适用到 5NF。该过程可以归纳为：通过一连串的归约，把一组给定的关系变量转化为一些更“好”的形式。应该注意的是该过程是“**可逆**”的，即总可以把该过程的输出（比方说属于 3NF 的关系变量的集合）重新映射到它的输入（比方说最初属于 2NF 的关系变量的集合）。可逆性很重要，因为它意味着在规范化过程中**没有信息丢失**。



图 12-2　不同级别的范式

回到范式这个主题上来：在 12.5 节将会看到，Codd 的最初关于 3NF 的定义［11.6］存在不足，Boyce 和 Codd 一起，对原来的 3NF 作了一些修正，提出了更加严格的定义［12.2］——即那些属于修正后 3NF 的关系变量必然属于原来的 3NF，但是一些属于原来的 3NF 的关系变量可能不属于修正后的 3NF。为了和原来的 3NF 相区分，把修正后的 3NF 范式称为 **Boyce/Codd 范式**（BCNF）。

后来，Fagin［12.8］提出了**第四范式**（4NF——之所以称为“第四”是因为那时 BCNF 仍被称为“第三”）。在［12.9］中，Fagin 还定义了**投影－连接范式**（PJ/NF，又被称为**第五范式**或 5NF）。如图 12-2 所示，一些属于 BCNF 的关系变量同时也属于 4NF，而一些属于 4NF 的关系变量同时也属于 5NF。

现在，读者也许会问：该过程是否有终点？是否有 6NF、7NF 以至无穷的范式？虽然这是一个值得一提的问题，但本书不打算对这个问题作详细的讨论。除了图 12-2 所示的范式外，确实存在其他的范式，但是，从某种意义上说，5NF 是“最后”的范式。这一问题将在第 13 章讨论。

2. 本章结构

本章主要是分析进一步规范化概念——从 1NF 到 BCNF（另外两个范式在第 13 章分析），本章安排如下：引言之后，在 12.2 节讨论基本概念——无损分解，并证明函数依赖对这个概念的重要性（事实上，函数依赖构成了 Codd 三个范式，包括 BCNF 范式的基础）。12.3 节介绍前三

个范式的原型，并通过例子说明一个关系变量如何经过一步一步的规范化而最终成为 3NF 的关系变量。12.4 节介绍分解选择（alternative decomposition）即如果存在多种分解方法，如何选择最“好”的一种。接下来，在 12.5 节讨论 BCNF。12.6 节讲解了具有关系值属性的关系变量。最后在 12.7 节进行总结并提出一些结论性评价。

希望不要把后面介绍的内容教条化，相反，在实际工作中要在很大程度上依靠直觉。事实上，像无损分解、BCNF 等术语虽然有些深奥，但是它们应该是简单的常识。一些参考文献在介绍这部分内容时太正式、古板。可以从参考文献［12.5］中找到较好的学习材料。

最后，给出两个指导性的评论：

1）前面已经提到，规范化的主要思想是：数据库设计人员在设计数据库时，他所设计的数据库的关系变量应该是属于“最终”范式（5NF）的。然而，不应把这个建议当做定则，因为在实际工作中，偶尔会有很多理由需要忽略规范化理论（见本章后面的习题 12.7）。事实上，这也说明数据库设计是个极其复杂的工作。规范化理论是很有用的，但它不是灵丹妙药；因此，任何一个设计数据库的人都应该熟悉规范化理论，但这并不意味着必须只依据这个理论。在第 14 章将讨论数据库其他方面的设计，这些设计和规范化理论很少甚至没有联系。

2）前面已经提到，本章将把规范化过程作为介绍和讨论不同范式的基础。然而，这并不是说实际的数据库设计工作一定是运用这一过程实现的，事实上很有可能运用将在第 14 章介绍的自上而下方案，而不运用本章介绍的规范化过程。规范化思想可以用来验证设计结果是否在无意中违反了规范化理论。然而，规范化过程确实为描述规范化理论提供了一个方便的框架。本章为了说明问题，假定在设计过程中采用这个过程。

## 12.2　无损分解和函数依赖

在开始具体讨论规范化过程之前，有必要仔细分析这一过程的一个重要性质：**分解的无损性（无损分解）**。规范化过程涉及把一个关系模式分解成几个关系模式，而且这种分解是“可逆”的，这样在分解过程中不会有信息丢失。也就是说，人们感兴趣的是没有信息丢失的分解。从后面的分析可以看出：一个模式分解是否是无损的这个问题和函数依赖密切相关。

以大家熟悉的关系变量 *S* 为例，该关系变量的表头为｛*S*#，STATUS，CITY｝（为简单起见，忽略 SNAME）。图 12-3 的上部是该关系变量的一个实例（关系），标有 a 和 b 的是该关系的两种不同分解。

S

| S# | STATUS | CITY |
|---|---|---|
| S3 | 30 | Paris |
| S5 | 30 | Athens |

(a)　SST

| S# | STATUS |
|---|---|
| S3 | 30 |
| S5 | 30 |

SC

| S# | CITY |
|---|---|
| S3 | Paris |
| S5 | Athens |

(b)　SST

| S# | STATUS |
|---|---|
| S3 | 30 |
| S5 | 30 |

STC

| STATUS | CITY |
|---|---|
| 30 | Paris |
| 30 | Athens |

图 12-3　关系变量 *S* 的一个实例及相应的两个分解

仔细分析这两种分解，可以看出：

1）在 a 中，没有任何信息丢失。从 SST 和 SC 的值仍旧可以导出供应商 *S*3 的状态是 30，地址是 Paris，而供应商 *S*5 的状态是 30，地址是 Athens。也就是说，该分解确实是无损的。

2）相反，在 b 中却有信息丢失，从分解后的关系中还是可以导出两个供应商的状态是 30，但不知道供应商地址，即第二个分解不是无损的，而是有损的。

到底是什么原因使得第一个分解是无损的，而第二个分解是有损的呢？通过观察，首先可以发现，“分解”过程实际上是个投影过程，图中的 SST、SC 和 STC 都是原来的关系变量 S 的一个投影，所以规范化过程中的分解实际上就是**投影**。注意：本书第二部分已经提到，人们经常说的“SST 是关系变量 *S* 的投影”，更确切地说，应该是“关系变量 SST 在任何时候的值都是关系变量 *S* 在相应时间的值的投影”。

第二，“在 a 中，没有任何信息丢失”是指 SST 和 SC 经过自然连接可以得到原来的 *S*。相反，在 b 中，SST 和 STC 经过连接却不能得到原来的 *S*，所以说在分解 b 中丢失了信息[㊀]。更精确地说，“可逆”的意思就是原来的关系变量等于它的投影的连接。所以，就像规范化过程中的分解是投影操作一样，“重组”（recomposition）就是**连接**操作。

有一个有意思的问题：如果 *R*1 和 *R*2 是关系变量 *R* 的投影，且 *R*1 和 *R*2 属性集的并集包含 *R* 的所有属性，那么，*R*1 和 *R*2 应满足什么条件，才能保证 *R*1 和 *R*2 的连接能得到原来的 *R*？这就是函数依赖的用武之地。回到例子中来，关系变量 S 满足最小函数依赖集：

```
S# → STATUS
S# → CITY
```

如果只是说关系变量 *S* 满足这些函数依赖，还不能说明为什么关系变量 *S* 和它的投影 {*S#*, STATUS} 和 {*S#*, CITY} 的连接等价，事实上还有如下定理（Heath 定理 [12.4]）：

- **Heath 定理**：假设有一个关系变量 *R*{*A*,*B*,*C*}，*A*、*B* 和 *C* 是属性集。如果关系变量 *R* 满足函数依赖 *A*→*B*，则 *R* 和投影 {*A*, *B*}、{*A*, *C*} 的连接等价。

把 *A* 看做 *S#*，把 *B* 看做 STATUS，把 *C* 看做 CITY，该定理就可以证明关系变量 *S* 可以无损分解成它的投影 {S#, STATUS} 和 {S#, CITY}。同时我们知道关系变量 *S* 不能无损分解成它的投影 {S#, STATUS} 和 {STATUS, CITY}。Heath 定理不能解释为什么如此[㊁]，但是，可以直观地看出，在后面的分解中丢失一个函数依赖，即在后一个分解中虽然仍然满足函数依赖 S#→STATUS，但不能满足函数依赖 S#→CITY。

概括一下，把关系变量 *R* 分解为投影 *R*1,*R*2,…,*Rn*，如果 *R* 等于 *R*1,*R*2,…,*Rn* 的连接，则我们可以说分解是**无损**的。注意，在实际操作中，我们通常增加额外的要求：*R*1,*R*2,…,*Rn* 需要均参与连接，这样可以避免某些冗余的产生。例如，我们不把关系变量 *S* 到投影{*S#*}、{*S#*, STATUS} 和 {*S#*, CITY} 的分解看做一个无损分解，尽管 *S* 无疑等于这三个投影的连接。为简单起见，从现在开始，除非有明确的反面说明，否则我们认为这一额外的要求总有效。

**进一步了解函数依赖**

下面以一些关于函数依赖的评论结束本节。

1）**不可约**（irreducibility）：第 11 章说过，一个**左部不可约**的函数依赖是指函数依赖的左边不“太大”。例如，12.1 节的关系变量 SCP 满足函数依赖：

```
{ S#, P# } → CITY
```

然而，属性 P#在该函数依赖的左边是多余的，也就是说，该关系变量还满足函数依赖：

```
S# → CITY
```

（即 CITY 还函数依赖于 *S#*）。后一个函数依赖是左部不可约的，但前面那个却不是，即 CITY 不可约地依赖于 *S#*，而不是不可约地依赖于 {*S#*, *P#*}[㊂]。左部不可约函数依赖和不可约依赖在第

---

㊀ 详细地说，我们得到原始 *S* 中的所有元组，以及一些额外的“虚假”元组；我们不可能得到比原始 S 更小的集合。（练习：证明这一结论。）因为我们通常无法知道结果中哪些元组是虚假的，哪些是真实的，所以我们确实丢失了信息。

㊁ 因为 Heath 定理形如“当……则……”而不是“当且仅当……则……”（见本章结尾的习题 12.1），所以它不能做出解释。我们将在第 13 章 13.2 节讨论 Heath 定理更强的形式。

㊂ 本书使用“左部不可约函数依赖”和“不可约依赖”表示通常所说的“完全函数依赖”和“完全依赖”（本书的较早版本中也使用这两个术语）。后面的这两个术语虽然简洁，但描述性和准确性不强。

二范式和第三范式中是一个比较重要的概念见 12.3 节）。

2）**函数依赖图**：假设 *R* 是关系变量，*I* 是 *R* 的不可约的函数依赖集，（要了解不可约函数依赖集，可参阅第 11 章）。可以用函数依赖图方便地表达函数依赖集 *I*，图 12-4 给出了关系变量 S、P 和 SP 的函数依赖集，在本章的后面部分将经常引用该图。

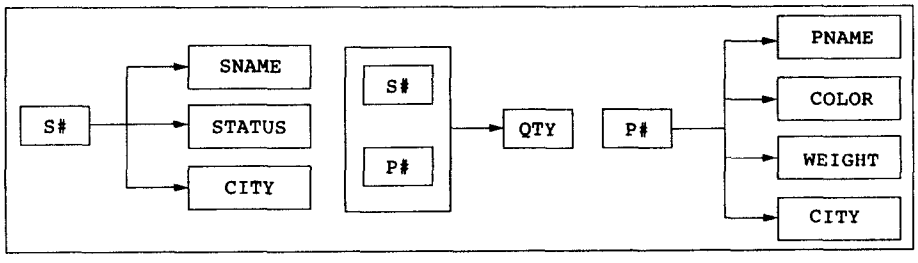



图 12-4　关系变量 *S*、*P* 和 *SP* 的函数依赖图

可以看出，图 12-4 的每一个箭头都是从相关的关系变量的候选码（常常是主码）指出。根据定义，每一个候选码都有箭头指出[⊖]，因为对于每一个给定的候选码的值，其他属性都有一个唯一的值和它对应，所以这些箭头是不可少的。如果图中还存在其他的箭头，就会带来问题。所以，不严格地说规范化过程就是消除那些不是从候选码中指出的箭头。

3）**函数依赖是语义上的概念**：函数依赖是一种特殊的完整性约束条件，所以它们也是一个语义上的概念（事实上，它们是关系变量谓词的一部分）。认识函数依赖是理解数据意义过程的一部分，例如，关系变量 *S* 满足函数依赖 *S#*→CITY，就是指每一个供应商的地址是唯一的。还可以从以下几个方面理解这一点：

- 在现实世界中存在这种在数据库中体现的约束条件，即每一个供应商的地址是唯一的。
- 因为这些约束是现实世界的语义表述的一部分，所以在数据库中应该遵守。
- 确保遵守这些约束的方法是在数据库定义时声明这些约束条件，这样 DBMS 就可以实现它们。
- 在数据库中声明这些约束条件的方法是定义函数依赖。

从后面的介绍中可以发现，规范化使得定义函数依赖变得非常简单。

## 12.3　第一、第二和第三范式

*注意*：为了简单起见，在本节中假定每个关系变量只含有一个候选码，并进一步假定该候选码为主码，这个假定在不太严格的定义中多次得到体现。含有多个候选码的关系变量在 12.5 节讨论。

现在开始介绍 Codd 的三个范式。为了说明问题，首先给出一个初步的、很不规范的 3NF 的定义，然后讨论把任意一个关系变量转化成一个 3NF 的关系变量集的过程，最后给出这三个范式的精确定义。然而可以发现，1NF、2NF 和 3NF 除了是 BCNF 及其他范式的基础外，本身的意义并不大。

下面是 3NF 的初步定义：

- **第三范式**（非常不正式）：当且仅当关系变量的非码属性满足下列条件时，该关系变量是属于 3NF 的：

  a）相互独立。

  b）不可约依赖于主码。

  关于“非码属性”和“相互独立”的解释如下：

- 一个“非码属性”是指不属于所讨论的关系变量的主码的属性。

---

⊖ 更准确地说，超码中总有箭头指出。如果函数依赖集合 *I* 按规定不可约，那么 *I* 中的所有函数依赖（或“箭头”）是左部不可约的。

- 两个或多个属性“相互独立”是指其中的任何一个属性都不函数依赖于其他属性的组合。这种独立性意味着其中的任何一个属性都可以独立地被修改。

作为例子，根据前面的定义，关系变量 *P* 是属于3NF的。属性PNAME、COLOR、WEIGHT和CITY是相互独立的（可以更新其中的任何一个属性如COLOR而不必同时改变其他属性，如WEIGHT），并且这些属性都不可约依赖于主码｛P#｝。

上述关于3NF的定义可以用下面的更不正式的定义解释：

- 第三范式（更不正式）：一个关系变量，当且仅当在任何时候它的每一个元组都含有一个主码来识别实体，同时有一组零个或更多的相互独立的属性从不同方面描述这个实体时，该关系变量是属于3NF的。

同样，关系变量 *P* 符合这个定义：*P* 中的每个元组有一个主码值（零件号码）来识别现实世界的某个零件，同时有四个不同的属性（零件名称、零件重量、零件颜色和零件产地），它们都用来描述零件，且各自独立于其他属性。

现在，回到规范化过程上来，首先介绍第一范式的定义：

- 第一范式：当且仅当一个关系变量的所有合法的值中，每一个元组的每个属性只含有一个值时，该关系变量属于1NF。

该定义只是说明所有的关系变量都是属于1NF的，这当然正确。然而，如果一个关系变量只属于第一范式（即是1NF关系变量而不是2NF关系变量，更不是3NF关系变量），则有很多理由认为它不是一个“好”的结构。为了说明这个问题，假设把关于供应商和发货的信息合在一起形成一个新的关系变量FIRST，而不是把它们分开来形成两个关系变量 *S* 和 *SP*，关系变量SCP的定义如下：

```
FIRST { S#, STATUS, CITY, P#, QTY }
      PRIMARY KEY { S#, P# }
```

该关系变量是12.1节的SCP的扩展。除了为了说明问题而引进的附加的约束条件

```
CITY → STATUS
```

外，每一个属性都和原来的意义一样。STATUS函数依赖于CITY，表示供应商的状态由它的地址决定，如每一个居住在London的供应商的状态是20。同时，为了简单起见，省略了SNAME。FIRST的主码是｛S#，P#｝，函数依赖图见图12-5。

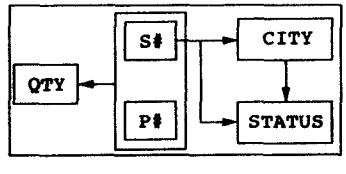



图12-5　关系变量FIRST的函数依赖图

可以看出，该关系变量的函数依赖图比属于3NF的关系变量的函数依赖图复杂，一个3NF关系变量的函数依赖图中只有从候选码中出来的箭头，而一个非3NF关系变量（如FIRST）的函数依赖图除了从候选码出来的箭头外，还有其他箭头——正是这些箭头造成了麻烦。实际上，关系变量FIRST同时违反了前面关于3NF定义中的条件a和b：非码属性不是相互独立的，因为STATUS依赖于CITY（新增加的箭头），并且非码属性不是不可约依赖于主码，因为STATUS和CITY都只依赖于S#。

为了说明多余箭头所带来的麻烦，图12-6列出了关系变量FIRST的一个实例。为了和新增加的约束条件（CITY决定STATUS）保持一致，把供应商 *S3* 的STATUS由30改为10，其他属性的值和原来的一样。图中的冗余是很明显的。例如每一个 *S1* 的元组的CITY都是London，而每一个CITY是London的元组的STATUS都是20。

FIRST中的冗余会造成一系列的**更新异常**，即由INSERT（插入）、DELETE（删除）和UPDATE（修改）等更新操作所造成的异常。首先看和函数依赖 *S#*→CITY相对应的供应商－地址间的冗余，在进行更新操作时都会出现问题：

- **INSERT**（插入）：不能插入一个居住在某个城市却没有零件供应的供应商，事实上，在图12-6中就没有居住在Athens的供应商 *S5*，其原因是，除非 *S5* 至少供应一种零件，否

则没有合适的主码（和 10.4 节一样，本章假定主码的属性没有默认值，进一步的讨论参见第 19 章）。

- **DELETE**（删除）：删除 FIRST 中的某个供应商的元组时，不仅删除了这个供应商的某种零件的发货，而且有可能把该供应商的其他信息（如地址）丢失。例如，把图 12-6 中主码是 {*S3*，*P2*} 的元组删除，则关于 *S3* 的地址是 Paris 的信息就丢失了（INSERT 和 DELETE 问题事实上就像一枚硬币的两面）。

| FIRST | S# | STATUS | CITY | P# | QTY |
|---|---|---|---|---|---|
| | S1 | 20 | London | P1 | 300 |
| | S1 | 20 | London | P2 | 200 |
| | S1 | 20 | London | P3 | 400 |
| | S1 | 20 | London | P4 | 200 |
| | S1 | 20 | London | P5 | 100 |
| | S1 | 20 | London | P6 | 100 |
| | S2 | 10 | Paris | P1 | 300 |
| | S2 | 10 | Paris | P2 | 400 |
| | S3 | 10 | Paris | P2 | 200 |
| | S4 | 20 | London | P2 | 200 |
| | S4 | 20 | London | P4 | 300 |
| | S4 | 20 | London | P5 | 400 |

图 12-6　关系变量 FIRST 的实例

注意：引起这些问题的关键是 FIRST 把太多的信息绑在一起，当从中删除一个元组时，会删除太多信息。更精确地说，FIRST 中同时包含了有关供应商的信息和有关发货的信息，当删除有关发货的信息时，同时也删除了有关供应商的信息。解决这个问题的方法当然是“分解”——即把发货的信息放在一个关系变量中，而把有关供应商的信息放在另一个关系变量中。这样，就可以把规范化过程概括为“分解”过程，即把逻辑上独立的信息放在独立的关系变量中。

- **UPDATE**（更新）：一般来讲，给定供应商的地址在 FIRST 中会出现多次，这就会给更新带来困难。例如，如果供应商 *S1* 从 London 搬到 Amsterdam，这时，我们就面临这样的选择：要么找出 FIRST 中所有和供应商 *S1* 有关的元组，并把地址改为 Amsterdam，要么就保留一个可能不一致的结果（*S1* 的地址有可能在一个元组里是 Amsterdam，而在另一个元组里是 London）。

解决这些问题的方法前面已经提到，就是用关系变量：

```
SECOND { S#, STATUS, CITY }
```

和

```
SP     { S#, P#, QTY }
```

代替原来的关系变量 FIRST。

关系变量 SECOND、SP 的函数依赖图见图 12-7，这两个关系变量的实例见图 12-8。可以看出，供应商 *S5* 的信息已被包含在内（只在关系变量 SECOND 中，在关系变量 SP 中却没有），而关系变量 SP 和往常的发货关系变量已完全一样。

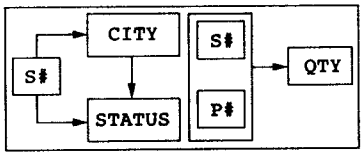



图 12-7　关系变量 SECOND 和 SP 的函数依赖图

SECOND

| S# | STATUS | CITY |
|---|---|---|
| S1 | 20 | London |
| S2 | 10 | Paris |
| S3 | 10 | Paris |
| S4 | 20 | London |
| S5 | 30 | Athens |

*No counterpart in Fig. 12.6*

SP

| S# | P# | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S1 | P3 | 400 |
| S1 | P4 | 200 |
| S1 | P5 | 100 |
| S1 | P6 | 100 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S4 | P4 | 300 |
| S4 | P5 | 400 |

图 12-8　关系变量 SECOND 和 SP 的实例

应该清楚，修改过的结构克服了前面描述的所有与更新操作有关的问题：

- **INSERT**（插入）：即使 *S5* 当前没有供应一种零件，通过把合适的元组插入关系变量 SECOND 中，也可以插入居住在 Athens 的供应商 *S5* 的信息。
- **DELETE**（删除）：可以在关系变量 SP 中删除有关 *S3* 和 *P2* 的元组，却不会丢失 *S3* 的地

址是 Paris 的信息。

- **UPDATE**（更新）：在修改过的结构中，给定供应商的地址只出现一次，而不是多次，因为在关系变量 SECOND（主码是 *S#*）中，给定供应商只有一个元组。也就是说，*S#*-CITY 间的冗余已被消除。在 SECOND 元组中对 *S1* 的地址作一次修改，就可以把 *S1* 的地址由 London 改为 Amsterdam。

比较图 12-5 和图 12-7 可以看出，将关系变量 FIRST 分解为 SECOND 和 SP 的作用是消除非不可约依赖，也正是消除了这些非不可约依赖才解决了前面描述的问题。也可以说，在关系变量 FIRST 中，属性 CITY 描述的不是由主码所确定的实体的信息，即发货，而是和发货有关的供应商（当然，属性 STATUS 也是如此）的信息。正是把这两种信息混在一起才导致了前面的问题。

下面给出第二范式的定义[⊖]：

- **第二范式**（假定只有一个候选码，且该候选码是主码）：当且仅当一个关系变量属于 1NF，且该关系变量的每一个非码属性都不可约依赖于主码时，该关系变量属于 2NF。

关系变量 SECOND 和 SP 都是 2NF 的（主码分别是 {*S#*} 和 {*S#*, *P#*}）。关系变量 FIRST 不是属于 2NF 的，一个属于 1NF 但不属于 2NF 的关系变量总是可以归约成与之等价的 2NF 的关系变量的集合。这个归约过程包括用适当的投影替代原来的关系变量，因为这些投影经过连接可以得到原来的关系变量，所以这些投影和原来的关系变量是等价的。在我们的例子中，关系变量 SECOND 和 SP 是 FIRST 的投影[⊜]，而 FIRST 是 SECOND 和 SP 的自然连接。

规范化过程的第一步可以归纳为利用投影消除“非不可约的”函数依赖，即给定如下关系变量：

```
R { A, B, C, D }
  PRIMARY KEY { A, B }
  /* assume A → D holds */
```

根据规范化理论，可以用下列两个关系变量替代原来的关系变量：

```
R1 { A, D }
   PRIMARY KEY { A }

R2 { A, B, C }
   PRIMARY KEY { A, B }
   FOREIGN KEY { A } REFERENCES R1
```

利用外码 - 主码的连接可以从 *R1* 和 *R2* 重新得到 *R*。

回到前面的例子，SECOND-SP 结构仍然存在问题。SP 是符合要求的，事实上，SP 已经是 3NF 了，在本节可以忽略。但是，SECOND 因非码属性间缺乏独立性，仍然会产生问题。SECOND 的函数依赖图仍然比 3NF 保留关系变量复杂，更确切地说，STATUS 依赖于 S#虽然是不可约依赖，但是它是不可约的**传递函数**依赖（通过 CITY）：每一个 *S#*决定一个 CITY，而每一个 CITY 决定一个 STATUS。一般来说，就像第 11 章介绍的那样，只要 $A \rightarrow B$ 和 $B \rightarrow C$ 都能满足，则必有传递函数依赖：$A \rightarrow C$。而传递依赖一样也会造成更新异常（现在把注意力集中在和函数依赖 CITY→STATUS 相关的 CITY-STATUS 的冗余上）。

- **INSERT**（插入）：不能插入一个没有供应商却具有状态的地址——即不能插入这样的信息：任何居住在 Rome（CITY）的供应商的状态（STATUS）为 50，除非已经有一个居住在 Rome 的供应商。
- **DELETE**（删除）：当在 SECOND 中删除与某一城市有关的元组时，不仅删除了与该城市有关的供应商的信息，而且删除了与该城市有关的状态（STATUS）信息。例如，在 SECOND 中删除 S5 的元组，则同时丢失 Athens 的状态是 30 的信息（同样，DELETE

---

⊖ 严格地说，定义第二范式只需要考虑特定的依赖集合，但在非正式的讨论中通常忽略这一点。除第一范式外，其他范式均使用类似的形式定义。

⊜ 除了一点：SECOND 可以包含像图 12-8 中供应商 *S5* 那样的元组，而 FIRST 中没有相对应的部分；换句话说，新的结构能够表达前面结构不能表达的信息。在这个意义上说，新的结构更符合实际情况。

（删除）和 INSERT（插入）像是同一枚硬币的两面）。

注意：这个问题同样是由于“捆绑”，即关系变量 SECOND 中同时含有与供应商和城市有关的信息。同样，解决这些问题的方法是“分解”，即把供应商的信息放在一个关系变量中，而把关于城市的信息放在另外一个关系变量中。

- **UPDATE**（更新）：某一城市的状态在 SECOND 中有可能出现多次（该关系变量仍存在冗余）。因此，如果要把 London 的状态由 20 改为 30，则同样面临这样的选择：要么在 SECOND 中查找所有与 London 有关的元组，并把状态（STATUS）由 20 改为 30；要么得到一个有可能不一致的结果（在一个元组中 London 的状态是 20，而在另一个元组中 London 的状态是 30）。

再一次用投影代替原来的关系变量（本例中是 SECOND）以解决这些问题，即用投影：

```
SC { S#, CITY }
CS { CITY, STATUS }
```

代替 SECOND {S#, CITY, STATUS}。关系变量 SC 和 CS 的函数依赖图见图 12-9，图 12-10 是这两个关系变量的一个实例。同样，这种归约是可逆的，因为 SECOND 是 SC 和 CS 通过 CITY 的自然连接。




图 12-9　关系变量 SC 和 CS 的函数依赖图

SC

| S# | CITY |
|---|---|
| S1 | London |
| S2 | Paris |
| S3 | Paris |
| S4 | London |
| S5 | Athens |

CS

| CITY | STATUS |
|---|---|
| Athens | 30 |
| London | 20 |
| Paris | 10 |
| Rome | 50 |

图 12.8 中没有副本

图 12-10　关系变量 SC 和 CS 的实例

同样应该清楚，这种修订过的结构克服了前面描述的更新问题（该问题的具体讨论作为练习）。比较图 12-7 和图 12-9，可以看出进一步分解的作用是消除 STATUS 对 *S#*的传递函数依赖，也正是消除了这种传递函数依赖才解决了更新异常问题。直观地说，在关系变量 SECOND 中，属性 STATUS 所描述的并不是由主码所标识的实体，即供应商的信息，而是供应商的地址信息。同样，把这两种信息放在同一关系变量中会带来问题。

下面给出第三范式的定义：

- **第三范式**（假定关系变量只有一个候选码，且该候选码是主码）：当且仅当一个关系变量属于 2NF 且该关系变量的所有非码属性都不传递依赖于主码时，该关系变量属于 3NF。注意：“不传递依赖”蕴涵不互相依赖，从这个意义上说，该术语的解释和本节开始的解释一样。

关系变量 *SC* 和 *CS* 都属于 3NF（它们的主码分别是 {*S#*} 和 {CITY}）。关系变量 SECOND 不属于 3NF。一个属于 2NF 但不属于 3NF 的关系变量总是可以归约成一些属于 3NF 的关系变量的集合。前面已经提到，这种归约是可逆的，即在归约中是没有信息丢失的，然而归约后的 3NF 关系变量集中含有一些在原来的 2NF 关系变量中不能描述的信息[○]，如 Rome 的状态是 50。

规范化过程的第二步可以归纳为利用投影消除非码属性间的传递函数依赖，也就是说，给定关系变量：

```
R { A, B, C }
  PRIMARY KEY { A }
  /* assume B → C holds */
       /*假定满足函数依赖 B → */
```

○ 由此得出结论，正如 SECOND-SP 结合能比第一范式关系变量 FIRST 更好地表达现实世界，SC-CS 结合比第二范式关系变量 SECOND 能更好地表达现实世界。

根据规范化理论，用如下两个关系变量 *R1*、*R2* 替代原来的关系变量 *R*：

```
R1 { B, C }
   PRIMARY KEY { B }

R2 { A, B }
   PRIMARY KEY { A }
   FOREIGN KEY { B } REFERENCES R1
```

利用外码和主码相匹配机制，通过连接 *R1* 和 *R2* 可以重新得到 *R*。

最后应该强调一点，一个给定关系变量的规范化级别是语义问题，而不仅仅是与关系变量在某一特定时间的值有关的问题。也就是说，不能只观察一个关系变量的一些实际值就判断该关系变量是否属于（比如）3NF，而必须通过了解数据的意义，如函数依赖等才能做出判断。还应该注意的是，即使知道这些函数依赖，也不能通过分析一些给定的数据就判断该变量是否属于3NF，通过这些具体的数据最多只能判断所讨论的数据是否违反了这些函数依赖，如果没有违反，则这些数据和该关系变量属于3NF的假设是一致的，当然，这不能保证这种假设是正确的。

## 12.4　保持函数依赖

在归约过程中，常常会出现这样的问题，即一个给定的关系变量可以有不同的无损分解方法。还是以12.3节的关系变量 SECOND 为例，该关系变量满足函数依赖 S#→CITY 和 CITY→STATUS 及传递函数依赖 S#→STATUS（参考图12-11，该图中传递函数依赖以虚线箭头表示）。12.3节的分析表明，该关系变量的更新异常可以通过模式分解把它分解成下面两个属于3NF的关系变量得到解决：

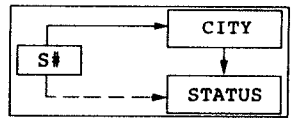



图12-11　关系变量 SECND 的函数依赖图

```
SC { S#, CITY }
CS { CITY, STATUS }
```

假设把这种分解叫作“分解 *A*”。同样还有另外一种分解（“分解 *B*”）：

```
SC { S#, CITY }
SS { S#, STATUS }
```

在这两种分解中，投影 *SC* 是一样的。分解 *B* 也是无损的，而且它的两个投影都属于3NF，但有许多原因可以认为分解 *B* 不如分解 *A* “好”。例如在分解 *B* 中，还是不能插入某个城市具有某种状态的信息，除非恰好有一个供应商居住在这个城市。

下面仔细分析这个例子。首先应该注意，分解 *A* 的投影和图12-11的实箭头相符，而分解 *B* 中的一个投影和图12-11的虚箭头相符。事实上，在分解 *A* 中的两个投影是相互独立的，只要对某个投影的更新在上下文是合法的——只要不违反所更新投影的主码唯一性约束条件，它们就可以单独更新而不用考虑其他投影○，这样，更新后的投影经过连接仍能得到合法的 SECOND（即连接不会违反 SECOND 的函数依赖）。相反，在分解 *B* 中，为了保证不违反函数依赖 CITY→STATUS（如果两个供应商的城市一样，则它们的状态必须一样），更新其中任何一个投影时都要对两个投影实行监控（考虑把供应商 *S1* 的城市由 London 改为 Paris 应做哪些更新）。也就是说，在分解 *B* 中，两个投影不是相互独立的。

在分解 *B* 中，这个问题的关键是函数依赖 CITY→STATUS 变成了一个跨越两个关系变量的数据库约束条件（参见第9章），这意味着在今天的一些产品中必须用过程来维护。相反，在分解 *A* 中是传递函数依赖 S#→STATUS 成了数据库约束条件，而这种数据库约束条件只要 S#→CITY 和 CITY→STATUS 这两个关系变量级的约束条件实现后就会自动实现，而要实现这两个关系变量级的约束条件是非常容易的，只要实现相应的主码唯一性约束条件即可。

投影独立性概念为从可能存在的多种分解方法中选择一个合适的分解提供了一个准则。具体

○　除了 SC 到 CS 的参照约束。

地说，一个投影相互独立的分解比相互不独立的分解“好”。Rissanen 在参考文献［12.6］中介绍了判断关系变量 *R* 的两个投影 *R*1 和 *R*2 是否相互独立的方法，指出当且仅当满足下列条件时，*R*1 和 *R*2 相互独立：

- *R* 中的所有函数依赖都是 *R*1 和 *R*2 的函数依赖的逻辑推论（logical consequence）。
- *R*1 和 *R*2 的相同属性至少组成它们之中一个的候选码。

考察前面定义的分解 *A* 和 *B*。在分解 *A* 中，因为它们的共同属性 CITY 是 CS 的主码，而且 SECOND 中的函数依赖不是分别出现在 CS 和 SC 中，就是 SC 和 CS 的函数依赖的逻辑推论，所以分解 *A* 中的两个投影是相互独立的；而在分解 *B* 中，尽管它们的共同属性 S#是分解后的两个投影的候选码，但是，原来的函数依赖 CITY→STATUS 却不能从分解后的关系变量 SC、SS 中推导出来，所以，在分解 *B* 中的两个投影不是相互独立的。注意：存在第三种可能的分解，即用投影｛S#, STATUS｝和｛CITY, STATUS｝代替 SECOND，因为这种分解不是无损的，所以是不合法的（练习：证明这个分解不是无损的）。

一个不能被分解成几个相互独立的投影的关系变量称为**原子**关系变量［12.6］。然而应该注意，事实上，一个给定的关系变量如果不是原子的并不意味着一定要把它分解成几个原子的关系变量。例如，供应商的关系变量 *S* 和零件的关系变量 *P* 并不是原子的，却根本没必要进行进一步的分解，当然 SP 是原子的，那就更没有必要进行分解了。

规范化过程中把一个关系变量分解成相互独立的投影的思想被称为**“保持函数依赖”**。最后，我们对这个概念作更准确的解释：

1）假设给定一个关系变量 *R*，运用规范化过程对它进行规范化——用一系列的关系变量 *R*1，*R*2，…，*Rn*（都是 *R* 的投影）代替 *R*。

2）假设 *R* 的函数依赖集是 *S*，而 *R*1，*R*2，…，*Rn* 的函数依赖集分别是 *S*1，*S*2，…，*Sn*。

3）每个函数依赖集 *Si* 只和投影 *Ri* 的属性有关（$i=1, 2, \cdots, n$），这样实现 *Si* 就非常简单。但是，真正要实现的是原来 *S* 中的函数依赖。这样就希望分解成 *R*1，*R*2，…，*Rn* 后，在实现 *S*1，*S*2，…，*Sn* 中的约束条件的同时就等价于实现 *S* 中的约束条件——即要求分解是保持函数依赖的。

4）假设 *S*′是 *S*1，*S*2，…，*Sn* 的并集，一般来讲，为了使分解是保持函数依赖的，只需 *S* 和 *S*′的闭包相同即可，而不必使 *S*′ = *S*（关于函数依赖集的闭包，参见 11.4 节）。

5）现在还没有一个有效的方法来计算函数依赖集的闭包，所以计算两个函数依赖集的闭包是否相等是不可能的。然而，现在已经有一种验证一个分解是否保持函数依赖的方法，具体算法已超出本书的讨论范围，要进一步了解可参考文献［8.13］。

这里给出一种 9 个步骤的算法作为后面的参考，运用该算法可以把任意一个关系变量 *R* 无损分解（保持函数依赖）成 3NF 投影的集合 *D*。设 *R* 的函数依赖集为 *S*：

1）初始化 *D* 为空集。

2）设 *I* 为 *S* 的不可约覆盖。

3）设 *X* 为 *I* 中出现在函数依赖 *X*→*Y* 左端属性的集合。

4）设 *I* 中左端为 *X* 的函数依赖的全集为 *X*→*Y*1，*X*→*Y*2，…，*X*→*Yn*。

5）设 *Z* 为 *Y*1，*Y*2，…，*Yn* 的并。

6）用 *D* 及 *R* 在 *X* 和 *Z* 上的投影的并代替 *D*。

7）对每个不同的 *X* 重复步骤 4 ~ 6。

8）设 *A*1，*A*2，…*An* 为 *R*（如果存在的话）中尚未处理的属性（例如，未被 *D* 中任何关系变量所包含）；用 *D* 及 *R* 在 *A*1，*A*2，…，*An* 上的投影的并代替 *D*。

9）如果 *D* 中的关系变量均不包含 *R* 中的候选码，用 *D* 及 *R* 在某个候选码上的投影的并代替 *D*。

## 12.5　BOYCE/CODD 范式

现在，把前面关于每个关系变量只含有一个候选码的假设去掉，来考虑一般情况下可能出现

的问题。事实上，Codd 原来的 3NF［11.6］没有很好地处理一般情况下的问题，更确切地说，是没有很好地处理具有下列特性的关系变量：

1）具有一个或多个候选码。

2）候选码是复合的。

3）候选码之间是重叠的（即至少有一个属性是相同的）。

3NF 后来被由 Boyce 和 Codd 提出的一个更为严格的定义取代，该定义同时适用于上述情况［12.2］。然而，由于该定义比原来的 3NF 定义要严格，所以有必要给它一个新的名称而不是沿用原来的 3NF，这样它就被叫做 Boyce/Codd 范式（BCNF）。○ 注意：在实际生活中，同时满足上述条件的情况并不常见。对于一个不满足上述条件的关系变量，3NF 和 BCNF 是等价的。

为了解释 BCNF，首先回顾一下前面的内容：决定因素（参见第 11 章）是指函数依赖的左边，平凡的函数依赖指左边是右边的超集的函数依赖。下面给出 BCNF 的定义：

- **Boyce/Codd 范式**：当且仅当一个关系变量的所有非平凡的、左部不可约的函数依赖的决定因素是候选码，则该关系变量属于 Boyce/Codd 范式（BCNF）。

或用一种非正式的定义：

- **Boyce/Codd 范式**：（非正式定义）当且仅当一个关系变量的唯一的决定因素是候选码，则该关系变量属于 Boyce/Codd 范式（BCNF）。

也就是说，函数依赖图中唯一的一个箭头是从候选码中出来的。前面已经说过，每一个候选码总有箭头出来，而 BCNF 认为这里没有其他箭头，也就是说，在规范化过程中已没有箭头可消除。应该注意这两个 BCNF 的定义的区别，在非正式定义中默认：（1）决定因素不“太大”；（2）所有的函数依赖都是非平凡的函数依赖。为了使问题简单化，除非有特别声明，在本章的剩余部分一直沿用这两个假设。

值得指出的是，从概念上讲，BCNF 比 3NF 简单，它没有明确地引用第一范式和第二范式的概念，也没有引用传递依赖的概念。而且，BCNF 的定义虽然比 3NF 严格，但是任何一个关系变量都可以无损分解成一系列 BCNF 关系变量的集合。

在讨论含有多个候选码的关系变量之前，首先讨论既不属于 3NF 又不属于 BCNF 的关系变量 FIRST 和 SECOND，以及既属于 3NF 又属于 BCNF 的关系变量 SP、SC 和 CS。在关系变量 FIRST 中，有三个决定因素：即 {S#}、{CITY} 和 {S#, P#}，其中 {S#, P#} 是候选码，所以 FIRST 不属于 BCNF。关系变量 SECOND 含有两个决定因素：{S#} 和 {CITY}，其中 {S#} 是候选码，所以 SECOND 也不属于 BCNF。而关系变量 SP、SC 和 CS 属于 BCNF，因为每个关系变量的唯一的决定因素是候选码。

下面考虑具有两个相互分离（即不重叠）候选码的关系变量。假设在供应商关系变量 S{S#,SNAME,STATUS,CITY} 中，{S#} 和 {SNAME} 都是候选码（即在任何情况下，供应商的编号（S#）和姓名（SNAME）是唯一的），并且假设属性 STATUS 和 CITY 是互相独立的，即在 12.3 节的假设 CITY→STATUS 不再成立，则该关系变量的函数依赖图见图 12-12。

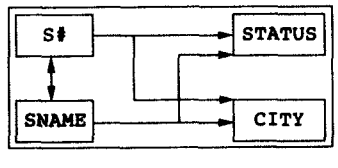



图 12-12　关系变量 S 的函数依赖图（SNAME 是候选码，且函数依赖 CITY→STATUS 不成立）

虽然该关系变量的函数依赖图看起来比 3NF 的“复杂”，但该关系变量是属于 BCNF 的，因为在该关系变量的函数依赖图中，所有的箭头都从候选码中出来。从本例可以看出，一个关系变量有多于一个的候选码不一定是“不好”的。

下面讨论一个候选码相互重叠的例子。候选码相互重叠是指两个或两个以上的候选码至少含有一个以上的公共属性。为了和第 9 章关于该问题的讨论保持一致，在下面的例子中不再从候选

○ 第三范式的定义实际上等价于 Heath 在 1971 年［12.4］给出的 BCNF 定义；因此“Health 范式”应该是更恰当的名字。

码中选定主码，因此在本节的图中，也不对任何属性加下划线。

例 1：假设供应商的姓名是唯一的，看下面的关系变量：

```
SSP { S#, SNAME, P#, QTY }
```

它的候选码是 {S#, P#} 和 {SNAME, P#}。该关系变量属于 BCNF 吗？答案是“不是”，因为 S#和 SNAME 都是决定因素（它们互相决定，所以 {*S#*} 和 {SNAME} 都是决定因素），但不是候选码。

图 12-13 是该关系变量的一个实例。

从图中可以看出，关系变量 SSP 和 12.3 节的 FIRST、SECOND（以及 12.1 节中的关系变量 SCP）一样有冗余，同样会产生更新异常。例如，把供应商 *S*1 的姓名（SNAME）由 Smith 改为 Robinson 同样会出现这样的问题：或者查找所有 *S*1 的元组并修改它，或者得到一个可能不一致的结果。然而根据以前的定义，SSP 是属于 3NF 的，因为原来的定义没有规定属于其他候选码的属性必须不可约依赖于每一个候选码，所以属性 SNAME 不是不可约依赖于候选码 {*S#*, *P#*} 的事实被忽略了。注意，这里的 3NF 是指［11.6］中关于 3NF 的原始定义，而不是本书 12.3 节的简化形式。

| SSP | S# | SNAME | P# | QTY |
|---|---|---|---|---|
| | S1 | Smith | P1 | 300 |
| | S1 | Smith | P2 | 200 |
| | S1 | Smith | P3 | 400 |
| | S1 | Smith | P4 | 200 |
| | .. | ..... | .. | ... |

图 12-13　关系变量 SSP 的(部分)实例

解决该问题的方法当然是把该关系变量分解成两个投影：

```
SS { S#, SNAME }
SP { S#, P#, QTY }
```

或者

```
SS { S#, SNAME }
SP { SNAME, P#, QTY }
```

这两种分解是等价的，它们都属于 BCNF。

现在，应该停下来仔细回顾本章的目的。显然，原来的只包含一个关系变量 SSP 的设计是“不好”的，其中的问题很明显，任何一个合格的数据库设计人员即使不了解 BCNF 的思想也不会提出这样的设计。常识会告诉设计人员采用 SS-SP 这种结构的设计会更好。但什么是“常识”呢？数据库设计人员选择 SS-SP 结构而不选择 SSP 结构的理论基础是什么呢？

答案当然是函数依赖理论和 Boyce/Codd 范式。也就是说，这些概念（函数依赖，BCNF，已经讨论的和即将讨论的各种范式）都是形式化的“常识”。该领域的所有定理的目的就是识别一些常识，然后把它们形式化——这当然不是一个简单的工作！但如果成功了，就可以把这些规则“机械化”，即可以编写程序，用机器去实现这些规则。规范化理论的批评者恰巧忽略了这一点，他们认为这些思想都是常识，而没有意识到给这些常识一个精确的形式化定义的重要性。

例 2：考虑具有属性 *S*、*T* 和 *J* 的关系变量 SJT（有些人可能认为该关系变量是**病理方面的**），我们用 *S* 表示学生，*J* 表示课程，*T* 表示教课程 *J* 的教师。SJT 中的元组 {*S*: *s*, *J*: *j*, *T*: *t*,} 表示学生 *s* 选修了教师 *t* 教的课程 *j*。该关系变量中有下列约束条件：

- 某个学生选定某门课程就对应一个固定的教师。
- 每一个教师只教一门课，而每门课有若干教师。

| SJT | S | J | T |
|---|---|---|---|
| | Smith | Math | Prof. White |
| | Smith | Physics | Prof. Green |
| | Jones | Math | Prof. White |
| | Jones | Physics | Prof. Brown |

图 12-14　关系变量 SJT 的实例

图 12-14 给出该关系变量的一个实例。

那么，该关系变量满足什么样的函数依赖呢？从第一个约束条件可以得出函数依赖：{S, J} →T，从第二个约束条件可以得出函数依赖：T→J。因为一门课可以有多名教师，所以不能满足函数依赖 J→T。所以该关系变量的函数依赖如图 12-15 所示。

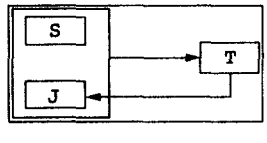



图 12-15　关系变量 SJT 的函数依赖

同样，该关系变量中存在相互重叠的候选码，即 {S, J} 和

{S, T}。而且该关系变量属于3NF但不属于BCNF，所以它同样会产生更新异常。例如，如果要删除Jones选修的课程physics，则会丢失教授physics的教师Brown教授的信息。产生该问题的原因是属性T是决定因素，而不是候选码。同样可以把该关系变量分解成两个属于BCNF的投影：

```
ST { S, T }
TJ { T, J }
```

练习：根据图12-14给出上述关系变量的一个实例，并画出相应的函数依赖图，证明这两个关系变量是属于BCNF的，并指出各自的候选码，检验这种分解是否会产生更新异常。

然而，上述分解虽然避免了更新异常，却带来了新的问题：根据Rissanen的理论（见12.4节），该分解的两个投影不是互相独立的，更确切地说函数依赖

```
{ S, J } → T
```

不能从函数依赖

```
T → J
```

（它是这两个投影中唯一的一个函数依赖）中推导出来。结果，这两个投影不能被独立地更新。例如，不能在ST中插入元组 {Smith, Brown教授}，因为Smith已经选修了Green教授教的physics课，然而系统如果不检查关系变量TJ就不能检查出这个事实。人们有时会面临两难的选择：（1）把关系变量分解成属于BCNF的关系变量；（2）把关系变量分解成互相独立的关系变量，即不能同时满足两方面的要求。

让我们更详细地分析一下这个例子：事实上，关系变量SJT虽然不属于BCNF，但根据Rissamen的理论，它是原子的（见12.4节）。从而可以看出，一个“原子”关系变量不能分解成相互独立的关系变量，并不意味着它根本不能分解（指无损分解）。直观地说，“原子”并不是一个很好的术语，因为在数据库设计中，“原子”既非充分，也非必要。

例3：考察具有属性S（学生）、J（课程）和P（名次）的关系变量EXAM，EXAM中的一个元组 {S: *s*, J: *j*, P: *p*} 指学生*s*的课程*j*的成绩在班里的排名是*p*。假定该关系变量满足下列约束条件：

- 在同一门课程中，任意两个学生的排名都不相同。

这样，该关系变量的函数依赖图如图12-16所示。

同样，该关系变量有两个相互重叠的候选码 {S, J} 和 {J, P}，因为（1）给定一个学生和他选修的一门课程，都有唯一的一个名次和他对应；（2）给定一门课程和名次，只有一个学生和它对应。但是，很明显，该关系变量是属于BCNF的，因为这些候选码都是决定因素，而且也不会出现前面所说的更新异常（练习：验证此观点）。所以相互重叠的候选码并不一定会导致前面所讨论的问题。

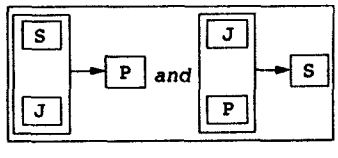



图12-16　关系变量EXAM的函数依赖图

从上面的分析可以看出，BCNF消除了一些原来的3NF定义中仍可能存在的问题，而且BCNF的定义也比3NF简洁，因为它没有涉及1NF、2NF、主码及传递依赖等概念。而且，该定义所涉及的概念——候选码，可以用更基础的概念——函数依赖代替（见[12.2]）。另一方面，主码、传递依赖等概念在实际工作中也很重要，因为它们有助于数据库设计人员了解如何一步一步地把任意一个关系变量归约成一系列属于BCNF的关系变量的集合。

最后我们给出一个4步的算法，使用这一算法可以把任意一个关系变量*R*无损分解成一系列属于BCNF的投影集*D*（而不需要保持所有依赖）的算法。

1）初始化*D*为只包含*R*。

2）对*D*中的每个非*BCNF*关系变量*T*，执行步骤3和4。

3）设*X*→*Y*为*T*的不满足*BCNF*要求的函数依赖。

4）用*T*的两个投影：在*X*和*Y*上的投影及在除*Y*中属性之外的所有属性上的投影代替*D*中

的 $T$。

## 12.6　具有关系值属性的关系变量

在第 6 章中已经说过，一个关系可能含有这样一个属性，这个属性的值也是一个关系（如图 12-17 所示）。同样，一个关系变量也可能含有一个值是关系的属性。从数据库设计的角度看，这种关系变量是不正常的，因为它们是不对称的[⊖]（更不用说它们的谓词有可能非常复杂!），而这种不对称会造成一系列的实际问题。例如，在图 12-17 中，供应商和零件是不对称处理的。

结果，查询（对称的）

1）查出供应零件 *P1* 的供应商的号码 *S#*。

2）查出由供应商 *S1* 提供的零件的号码 *P#*。

有不同的查询表达式：

```
1) ( SPQ WHERE TUPLE { P# P# ('P1') } ∈ PQ { P# } ) { S# }

2) ( ( SPQ WHERE S# = S# ('S1') ) UNGROUP PQ ) { P# }
```

（假定关系变量 SPQ 如值是如图 12-17 所示的关系的集合。）注意，这两个公式不但差别很大，而且均比它们在 SP 中的对应部分复杂得多。

更糟糕的是更新，例如考虑下列更新操作：

1）插入一个发货记录，它的供应商号是 *S6*，零件号是 *P5*，数量是 500。

2）插入一个发货记录，它的供应商号是 *S2*，零件号是 *P5*，数量是 500。

对于通常的发货关系变量 SP，这两个操作根本没什么区别，都是插入一条记录，但是，对于关系变量 SPQ 却相反，这两个操作有本质的区别（更不用说这两个操作比在 SP 上作相同的操作更复杂）：

```
1) INSERT SPQ RELATION
       { TUPLE { S# S# ('S6'),
                 PQ RELATION { TUPLE { P#  P# ('P5'),
                                       QTY QTY ( 500 ) } } } } ;
2) UPDATE SPQ WHERE S# = S# ('S2')
       { INSERT PQ RELATION { TUPLE { P#  P# ('P5'),
                                      QTY QTY ( 500 ) } } } ;
```

关系变量（至少是基本关系变量）一般不希望含有关系值属性，因为这种相对简单的逻辑结构使得对它们的操作也变得相对简单。然而应该清楚，这只是一个指导方针，而不是教条，在实际工作中，一个含有关系值属性的关系变量可能更能说明问题——即使是基本关系变量。图 12-18 是目录关系变量 RVK 的一个实例的一部分，该关系变量列出了数据库中所有的子关系变量以及它们的候选码。其中属性 CK 的值也是关系，而且是该关系变量唯一的候选码的一个组成部分。RVK 的定义如下：

```
VAR RVK BASE RELATION
      { RVNAME NAME, CK RELATION { ATTRNAME NAME } }
        KEY { RVNAME, CK } ;
```

注意：本章末尾的习题 12.3 介绍如何消除关系值属性的方法——如果必须消除这种关系值属性的话[⊖]（事实上常常是需要的）。

---

⊖ 事实上，以历史的观点来看，这样的关系变量甚至是非法的——它们被称为非规范化的，实际上它们甚至不满足第一范式（见第 6 章）。

⊖ 并且关系值是可消除的！注意，在 RVK 的情况下是不可消除的，起码不可直接消除（例如缺少对某种 CK-NAME——“候选码名称”（candidate key name）的介绍）。

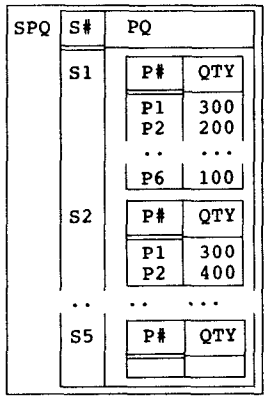



图 12-17　一个具有关系值属性的关系

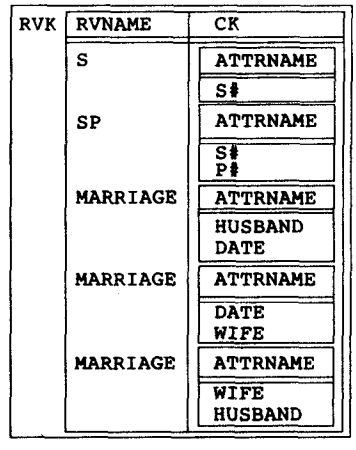



图 12-18　目录关系变量 RVK 的实例

## 12.7　小结

到目前为止，我们已经约束了进一步规范化的前两章的讨论。在这两章中，主要讨论了**第一、第二、第三范式**和 **Boyce/Codd 范式**。因为属于任何一个范式的关系变量都自动属于比它低一级的范式，反之则不然，而且存在一些关系变量，它属于某一范式，但不属于比该范式更高级别的范式，所以这些不同的范式（包括下一章要讨论的第四范式和第五范式）就组成一条线。更进一步地，任意一个关系变量都可以归约成 BCNF（事实上可以归约成 5NF），几乎任何一个关系变量都可以用一个等价的 BCNF（或 5NF）关系变量的集合替代。这种归约的目的是**避免冗余**和消除某些**更新异常**。

归约过程包括用一些**投影**替代给定的关系变量，这样通过这些投影的**连接**可以得到原来的关系变量，即这种归约过程是**可逆**的（或者说这种分解是**无损**的）。**函数依赖**在分解过程中同样具有重要作用，**Heath 定理**指出如果满足给定的函数依赖，则这种分解是无损的。这一步肯定了第 11 章中的观点，函数依赖这一概念“不是很基础，但十分靠近基础。”

这两章里还讨论了 Rissanen 的**相互独立的投影**概念，并建议当有多种分解时，分解成相互独立的投影比分解成相互不独立的投影要“好”。当把一个关系变量分解成相互独立的投影时，该分解保持了函数依赖。但是，把一个关系变量无损分解成 BCNF 的集合和保持函数依赖这两个目的有时是相互冲突的，不能同时满足。

最后给出 3NF 和 BCNF 的精确定义（参见 Zaniolo [12.7]），首先是 3NF 的定义：

■ **第三范式**（Zaniolo 的定义）：假设 $R$ 是关系变量，$X$ 是 $R$ 的属性子集，$A$ 是 $R$ 的任意一个属性，当且仅当每一个函数依赖 $X \rightarrow A$ 至少满足下列条件中的一个时，该关系变量属于 3NF：

1）$X$ 包含 $A$（这样，该函数依赖是平凡的）。

2）$X$ 是一个超码。

3）$A$ 属于 $R$ 的某个候选码。

只要把 3NF 定义中的第三种可能去掉，即可得到 **Boyce/Codd 范式的定义**（这也说明 BCNF 比 3NF 严格）。事实上，Codd 原来的 3NF 定义的缺陷产生的原因就是存在第三种可能性，最终导致了 BCNF 的引入。

## 习题

12.1　证明 Heath 定理。该定理的逆定理成立吗？

12.2　所有的二元关系变量都是属于 BCNF 的。该命题正确吗？

12.3　图 12-19 是某公司人事部门数据库中要存放的信息，以层次结构（如 IBM 的层次数据库系统 IMS）的形式体现如下理解该图：

- 该公司有若干部门。
- 每个部门有若干职工、项目和办公室。
- 每个职工都有工作经历（该职工所从事过的工作的集合）。
- 对每一项工作，该职工都有一个工资历史记录（该职工从事该工作时所得的工资）。
- 每个办公室有若干个电话。

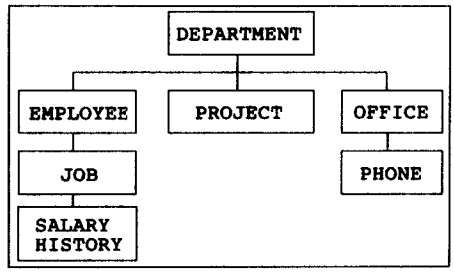


图 12-19　一个公司的数据库（层次结构）

该数据库包含以下信息：

- 每个部门的信息包括：部门号（唯一）、预算和部门经理的职工号（唯一）。
- 每个职工的信息包括：职工号（唯一）、当前参加的项目的项目号、办公室号码、电话号码、每个职工所从事的各个工作的的名称、加薪日期及工资额。
- 描述每个项目的属性：项目号（唯一）和预算。
- 描述办公室的属性：办公室号码（唯一）、楼层、办公室内的电话号码（唯一）。

根据这些信息，设计适当的关系变量，指出各个关系变量所应满足的函数依赖，并声明与这些函数依赖有关的假设。

12.4　一个订货系统数据库中包括顾客、存货和订单等内容，下面是该数据库应包含的内容：

- 每个顾客的信息：
  - 顾客号（唯一）
  - 收货（ship-to）地址（每个顾客可以有多个）
  - 余额
  - 赊购限额
  - 折扣
- 描述每份订单的属性：
  - 订单头信息：
    顾客号
    收货地址
    订货日期
  - 订单细则（每一份订单可以有多项订单细则）：
    货物编号
    订货数量
- 描述每种货物的属性：
  - 货物编号（唯一）
  - 制造厂商
  - 每个厂商实际存货量
  - 每个厂商规定的最低存货量
  - 货物的详细描述

由于处理的需要，每份订货单的每一项订货细则还应有一个未发货量，该值开始时是发货量，随着发货将减为 0。为这些数据设计一个数据库。对于不同的情况给出数据依赖的假设。

12.5　假设上题中只有很少量的顾客（例如 1% 或更少）有多个发货地址，这是符合实际情况的，由于这些极少数的例外而又不能忽视的情形使我们不能按一般的方法来处理问题。请回顾上一题的答案，并改进它。

12.6　（同习题 11.13）关系变量 TIMETABLE 有以下属性：

*D*　一星期中的一天（1 ~ 5）
*P*　一天中的一个时间段（1 ~ 8）
*C*　教室号码
*T*　教师姓名
*S*　学生姓名
*L*　课程名称

元组 {D: *d*, P: *p*, C: *c*, T: *t*, S: *s*, L: *l*} 指在时间段 {D: *d*, P: *p*} 时，学生 *s* 选修了教师 *t* 教的课程 *l*，上课地点是 *c*。可以假定，每节课持续一个时间段，并且一个星期中所有课程的名称都是唯一的。请把该关系变量归约成更好的结构。

12.7　关系变量 NADDR（同习题 11.14）的属性是：NAME（姓名，唯一）、STREET（街道）、CITY（城市）、STATE（州）和 ZIP（邮编）。对于任意一个邮编，只有一个城市和州与之对应；同样，对于任意给定的一个街道、城市或州，只有一个邮编和它对应。那么 NADDR 是否属于 BCNF？是否属于 3NF 或 2NF？能设计出更好的结构吗？

12.8　设 SPQ 为关系变量，它取值为满足图 12-17 中表明的形式的关系。描述 SPQ 的外部谓词。

## 参考文献

除了下面列出的参考文献外，还可以参考第 11 章中的参考文献，特别是 Codd 关于 1NF、2NF 和 3NF 的论文 [11.6]。

[12.1]　Philip A. Bernstein: "Synthesizing Third Normal Form Relations from Functional Dependencies," *ACM TODS 1*, No. 4 (1976 年 12 月)

本章讨论了把一个“大”关系变量分解成一些较“小”的关系变量（即具有少量属性）的方法。在这篇文章中，Bernstein 考虑了该问题的逆问题：用“较小”的关系变量合成“较大”的关系变量（也就是说具有更多属性）。但是在该文中，这个问题并不是这么描述的，他把该问题描述成根据给定的属性集和函数依赖合成一个属于 3NF 的关系变量集。然而，因为脱离关系变量这一具体的上下文，属性和函数依赖没有任何意义，所以，把含有函数依赖的二元关系变量作为基本结构，将会比把多个属性及其函数依赖作为基本结构更精确。

注意：可以把一组给定的属性集和函数依赖集看作是定义一个满足给定函数依赖集的通用关系变量（参见 [13.20]）。在这种情况下，“合成过程”就可以被看作是把该“通用关系变量”分解成 3NF 的关系变量集，但在目前讨论中仍然使用原来的关于“合成”的解释。

这样，合成过程就成了从一组二元关系变量和适用于这些关系变量的函数依赖集中合成 n 元关系变量的过程，而且，这些 n 元关系变量应该是属于 3NF 的（做这些工作时还没有 BCNF 的定义）。执行此任务的算法已经提出。

该方法有一个缺陷（由 Bernstein 发现）：合成算法只能处理语法（syntactic）问题而不能处理语义（semantics）问题。例如：给定函数依赖：

*A* → *B*（适用于关系变量 *R*{*A*,*B*}）
*B* → *C*（适用于关系变量 *S*{*B*,*C*}）
*A* → *C*（适用于关系变量 *T*{*A*,*C*}）

第三个函数依赖有可能是冗余的（即可以由第一个和第二个函数依赖蕴涵），也可能不是冗余的，这取决于 *R*、*S* 和 *T* 本身的意义。例如，假设 *A* 是雇员编号，*B* 是办公室号码，*C* 是部门编号，如果 *R* 表示“雇员的办公室”，*S* 表示“拥有该办公室的部门”，*T* 表示“职工所在的部门”，考虑一个职工在不属于他所在的部门的办公室办公的情况，则可以看出第三个函数依赖不是由第一个和第二个函数依赖所蕴涵的。合成算法假定，这两个 *C* 属性是一样的（事实上，它根本不识别关系变量的名称），这样就需要外部机制（如人的干预）去避免语义上的非法操作。在这种情况下，在最初定义函数依赖时就需要设计者给这两个属性定义不同的名称，如在 *S* 中定义为 *C1*，而在 *T* 中定义为 *C2*。

[12.2]　E. F. Codd: "Recent Investigations into Relational Data Base Systems," Proc. IFIP Congress, Stockholm, Sweden (1974), and elsewhere.

该文涉及的主题比较多，但主要是给出了改进的第三范式的定义，其中的第三范式实际上就是众所周知的 BCNF。该文还讨论了视图及视图更新、数据子语言、数据交换和进一步研究的方向等问题。

[12.3]　C. J. Date: "A Normalization Problem," in *Relational Database Writings 1991 – 1994*. Reading, Mass.: Addison-Wesley (1995).

为了证明这些抽象的理论，该文以航班管理数据库为例，分析了规范化问题，并利用它考察了数据库设计及显式的完整性约束条件的声明等问题。下面是该数据库中的函数依赖：

```
{ FLIGHT } → DESTINATION
{ FLIGHT } → HOUR
{ DAY, FLIGHT } → GATE
{ DAY, FLIGHT } → PILOT
{ DAY, HOUR, GATE } → DESTINATION
{ DAY, HOUR, GATE } → FLIGHT
{ DAY, HOUR, GATE } → PILOT
{ DAY, HOUR, PILOT } → DESTINATION
{ DAY, HOUR, PILOT } → FLIGHT
{ DAY, HOUR, PILOT } → GATE
```

这个例子还说明，仅仅以规范化理论为基础，很难设计出好的数据库。

[12.4] I. J. Heath: "Unacceptable File Operations in a Relational Database," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif. (November 1971).

该文给出了 "3NF" 的定义，它实际上是最早的 BCNF。该文还验证了本书 12.2 节提到的 Heath 定理。应该注意的是，本章所讨论的规范化的三个步骤实际上就是 Heath 定理的应用。

[12.5] William Kent: "A Simple Guide to Five Normal Forms in Relational Database Theory," *CACM 26*, No. 2 (February 1983).

该文阐明了 3NF（更精确地讲是 BCNF）具有吸引力的两大原因：每个属性必须代表码（整个码）的一个事实，而且仅仅是代表码。

[12.6] Jorma Rissanen: "Independent Components of Relations," *ACM TODS 2*, No. 4 (December 1977).

[12.7] Carlo Zaniolo: "A New Normal Form for the Design of Relational Database Schemata," *ACM TODS 7*, No. 3 (September 1982).

在 12.3 节提到了 3NF 和 BCNF 定义的基础。该文的主要目的是定义一个范式：基本码范式（elementary key normal form），即 EKNF，该范式在克服了 3NF 和 BCNF 的不足（3NF 的定义太 "宽松"，而 BCNF 的计算又太复杂）的同时，保留了两者的优点。该文还说明，利用 Bernsteirn 算法 [12.1] 得到的关系变量实际上属于 EKNF，而不是属于 3NF。

# 第 13 章　进一步规范化 II：高级范式

## 13.1　引言

在前一章讨论了从第一范式到 Boyce/Codd 范式的进一步规范化思想（这是函数依赖的概念所能达到的最高层次）。现在讨论 4NF 和 5NF 来结束这一部分的内容。正如下面将要看到的，4NF 的定义利用了一个新型的依赖，称为多值依赖（MVD）。多值依赖是广义的函数依赖。同样，5NF 的定义应用了另一种形式的依赖，称为连接依赖（JD），连接依赖是一种广义的多值依赖。在 13.2 节将讨论 MVD 和 4NF，13.3 节讨论 JD 和 5NF（该节还解释了 5NF 在某种特定意义上被称为是“最终的规范化形式”的理由）。值得说明的是，对 MVD 和 JD 的讨论没有第 11 章对函数依赖的讨论那么正式和完整。这是有意为之，正式的讨论留作研究论文（见本章末的“参考文献”）。

13.4 节回顾了全部的规范化过程，并且补充了一些评论。13.5 节简单地讨论了范式逆规范化的概念。13.6 节描述了另一个设计原则：**正交设计**（orthogonal design）。13.7 节是对今后在规范化领域的一些可能研究方向的简单介绍。13.8 节是全章的小结。

## 13.2　多值依赖与第四范式

假定有一个关系变量 HCTX（H 代表层次），其中包括 COURSES、TEACHERS、TEXTS 信息。在此变量中与 TEACHERS 和 TEXTS 相对应的属性是关系值属性（参考图 13-1 中的 HCTX 值的例子）。可以看到，每一个 HCTX 元组包括一个 COURSE（课程名）、一个包括 TEACHER 名的关系以及一个包括 TEXT 名的关系（在图中列出了两个这样的元组）。建立这样的元组的意图是，指定的课程 COURSE 可以由任意一个指定的教师 TEACHER 教授，并可以用任意一本指定的教材 TEXT 作为参考书。假定对于一个给定的课程 $c$，可以存在任意数量 $m$ 的相应教师和任意数量 $n$ 的相应教材（其中 $m>0$，$n>0$）。并假定：教师与教材之间是彼此相对独立的，这种假定可能并不实际。这就是说，无论哪位教师教指定的哪一门课程，都可以用同一本教材。最后还假定一个指定的教师和一本指定的教材是可以与任意数量的课程相联系的。

| HCTX | COURSE | TEACHERS | TEXTS |
|---|---|---|---|
| | Physics | TEACHER | TEXT |
| | | Prof. Green<br>Prof. Brown | *Basic Mechanics*<br>*Principles of Optics* |
| | Math | TEACHER | TEXT |
| | | Prof. Green | *Basic Mechanics*<br>*Vector Analysis*<br>*Trigonometry* |

图 13-1　关系变量 HCTX 的值的例子

现在假设要去掉关系值属性（与第 12 章中 12.6 节相同）。一种方法（这可能不是最好的方法，将在本节结尾进行讨论）是简单地将关系变量 HCTX 替换为一个有三个标量属性 COURSE、TEACHER 和 TEXT 的关系变量 CTX，如图 13-2 所示。从图中可以看到，每一个 HCTX 中的元组都在 CTX 中引发出 $m*n$ 个元组，其中 $m$ 和 $n$ 是 HCTX 中 TEACHERS 和 TEXTS 关系的基数（cardinality）。值得注意的是，所得到的关系变量 CTX 是“全码”（然而在 HCTX 中只有单一的候选码——COURSE）。

练习：给出一个能从 HCTX 推导出 CTX 的关系表达式。

CTX

| COURSE | TEACHER | TEXT |
|---|---|---|
| Physics | Prof. Green | *Basic Mechanics* |
| Physics | Prof. Green | *Principles of Optics* |
| Physics | Prof. Brown | *Basic Mechanics* |
| Physics | Prof. Brown | *Principles of Optics* |
| Math | Prof. Green | *Basic Mechanics* |
| Math | Prof. Green | *Vector Analysis* |
| Math | Prof. Green | *Trigonometry* |

图 13-2　与图 13-1 中 HCTX 值相对应的关系变量 CTX 的值

关系变量 CTX 的含义基本上可以归纳为：一个元组 {COURSE：*c*，TEACHER：*t*，TEXT：*x*} 在 CTX 中出现，当且仅当课程 *c* 可以由教师 *t* 教授，并且应用教材 *x* 为参考书。因此，对于一门给定的课程，所有的教师和教材的可能组合情况都可能出现。这就是说，CTX 满足如下关系变量的约束条件：

**如果**元组（*c*，*t*1，*x*1），（*c*，*t*2，*x*2）都出现，

**那么**元组（*c*，*t*1，*x*2），（*c*，*t*2，*x*1）也都出现（在此又用到了元组的简单表示法）。

很显然，在关系变量 CTX 中存在大量冗余，通常这会导致一定的更新异常（update anomaly）。举例来说，如果添加物理课可以由一个新教师教的信息，就必须要添加两个元组，对每一个教材都要添加一个。能避免这个问题吗？不难看出：首先，这个问题是由教师和教材彼此完全独立引起的。在此，如果 CTX 被分解为两个相互独立的投影 {COURSE，TEACHER} 和 {COURSE，TEXT}（设这两个投影名为 CT 和 CX），情况就会好得多（参见图 13-3）。

CT

| COURSE | TEACHER |
|---|---|
| Physics | Prof. Green |
| Physics | Prof. Brown |
| Math | Prof. Green |

CX

| COURSE | TEXT |
|---|---|
| Physics | *Basic Mechanics* |
| Physics | *Principles of Optics* |
| Math | *Basic Mechanics* |
| Math | *Vector Analysis* |
| Math | *Trigonometry* |

图 13-3　与 CTX 的值相对应的关系变量 CT 和 CX 的值

如果增加一个教授物理课的新教师的信息，现在我们所要做的就是在图 13-3 中的关系变量 CT 上插入一个单独的元组。（还要注意关系变量 CTX 可以通过连接 CT 和 CX 重新得到，因此这样的分解并没有损失。）因此，这种思想看来确实很合理，即对于像 CTX 这样的关系变量应该有一种“进一步规范化”的方法。

现在，你可能产生这样的想法：在 CTX 中并非一定存在冗余，因而与此相应的更新异常也不一定会出现。更明确地说：你会认为，在 CTX 中对每一个给定的课程，并不需要包括所有可能的教师和教材的组合。例如：两个元组显然足以显示物理课有两个教师和两本教材。但问题是：究竟是用哪两个元组呢？任何一个具体的选择都会引起关系变量中不明确的解释和奇怪的更新操作（试标明一个关系变量中这样的谓词！例如，试着陈述可以决定在这个关系变量中一些更新操作是否是一个可以接受的操作的标准）。

因此，在非正式的条件下 CTX 的设计明显是不好的，而分解为 CT 和 CX 就好多了。然而，问题是在正式的条件下冗余的问题并不很明显，特别是在 CTX 根本不满足任何函数依赖的情况下（除了一些平凡的函数依赖，如 COURSE→COURSE）。事实上，CTX 是属于 BCNF 范式的，[⊖] 因为它已经是全码——所有含全码的关系变量都必定包含在 BCNF 中（注意，CT 和 CX 这两个投影也是全码，因此它们也属于 BCNF 范式）。因此前一章的讨论对本章没有任何帮助。

像 CTX 这样的 BCNF 关系变量“问题”很早就被发现，并且处理它们的方法也很快就为人们所理解，至少是在直观上。然而，直到 1977 年 Fagin 的多值依赖概念（MVD）[13.14] 的提

⊖ HCTX 也属于 BCNF；实际情况是，它也属于第四范式及第五范式（见本章后面的定义）。

出，这些直观的观点才有了令人信服的理论根基。多值依赖是一种一般化的函数依赖，因而每一个 FD 都是 MVD，但反之则不正确（例如上面的 MVD 就不是 FD）。在关系变量 CTX 中，有两个 MVD 如下：

```
COURSE →→ TEACHER
COURSE →→ TEXT
```

（注意这里用双箭头，多值依赖 $A \twoheadrightarrow B$ 称为“*B* **多值依赖于** *A*”，或者说，“*A* **多值决定** *B*”。）让我们来看第一个 MVD，即 COURSE →→ TEACHER。直观地说，MVD 就意味着虽然一门课程并不对应一个相应的教师（例如并不存在函数依赖 COURSE→TEACHER）但每一门课确实对应一个定义得很好的教师集合。这里所说的“定义得很好”的含义，是指对于一门给定的课程 *c* 和一本给定的教材 *x*，教师 *t* 的集合是否与 CTX 中的(*c*,*x*)相匹配，只依赖于 *c* 的值，而与选择哪一个 *x* 值无关。第二个 MVD，COURSE →→ TEXT，也可以有类似的解释。

下面我们给出正式的定义：

- **多值依赖**：*R* 是一个关系变量，*A*、*B* 和 *C* 是 *R* 的属性子集。那么我们说 *B* **多值依赖**于 *A*，符号如下：

```
A →→ B
```

（读做“*A* 多值决定 *B*”，或简单地称为“*A* 双箭头 *B*”），当且仅当对于每一个可能的合法 *R* 值，*B* 值的集合对于给定的一组（*A* 值，*C* 值）只依赖于 *A* 的值，而与 *C* 的值无关。

很容易看出［13.14］，对于给定的变量 *R*{*A*,*B*,*C*}，多值依赖 $A \twoheadrightarrow B$ 存在，当且仅当多值依赖 $A \twoheadrightarrow C$ 也存在。这样 MVD 总是成对出现。因此通常用一种语句来表示它们：

```
A →→ B | C
```

例如：

```
COURSE →→ TEACHER | TEXT
```

在前面我们已经提到，多值依赖是一般化的函数依赖，在这种意义上每一个 FD 都是 MVD。更精确地说，一个 FD 就是一个只有一个依赖值（右边的）与一个给定的决定值相符合的 MVD。因此，如果 $A \rightarrow B$，那么一定 $A \twoheadrightarrow B$。

回到我们原来的 CTX 问题，现在可以看到像 CTX 这样的关系变量的问题是：它们包括不是 FD 的 MVD。（在这种问题并不明显的情况下，可以指出，正是 MVD 的存在使得这种问题存在。举例来说：增加一个教师要插入两个元组。为了保证 MVD 所提供的完整性约束，需要插入两个元组）。CT 和 CX 这两个投影并不包括任何这样的 MVD，它们完善了原来的设计。因此我们用这两个投影来代替 CTX，由 Fagin 证明的一个重要的定理（见参考文献［13.14］）为我们做这样的替换提供了依据：

- **定理**（Fagin）：假定 *R*{*A*,*B*,*C*}是一个关系变量，其中 *A*、*B* 和 *C* 都是属性集。那么 *R* 等同于它在 {*A*, *B*} 和 {*A*, *C*} 上的投影的组合，当且仅当 *R* 满足多值依赖 $A \twoheadrightarrow B \mid C$。

（*注意*：这是一个在第 12 章定义的 Heath 定理［12.4］的更高级的版本。）根据 Fagin［13.14］的理论，我们现在就可以定义第四范式（之所以这样叫是因为那时 BCNF 依然被称为第三范式）：

- **第四范式**：关系变量 *R* 属于 4NF，当且仅当存在 *R* 的属性子集 *A* 和 *B*，满足非平凡的多值依赖 $A \twoheadrightarrow B$，并且 *R* 的所有属性也都函数依赖于 *A*。*注意*：如果 *A* 是 *B* 的超集，式 *A* 与 *B* 的并集为 *R* 的所有属性，则多值依赖 $A \twoheadrightarrow B$ 是平凡的。

换句话说，在 *R* 中唯一的非平凡的依赖（函数依赖或多值依赖）是 $K \twoheadrightarrow X$ 形式（例如一个超码 *K* 对另一个属性 *X* 的函数依赖）。同样，如果 *R* 属于 BCNF，并且 *R* 中的所有非平凡的多值依赖事实上都是“非码函数依赖”，则 *R* 属于 4NF。因此特别要注意的是，4NF 包含了BCNF。

既然关系变量 CTX 包括了根本不是 FD 的 MVD，它就不属于 4NF，更不用说非码函数依赖了。然而，它的两个投影 CT 和 CX 都属于 4NF。因此，4NF 是 BCNF 的一种改进，因为 4NF 消

除了一种并不受欢迎的依赖。而且，Fagin 在参考文献［13.14］中还提到了 4NF 通常是可以实现的，这就是说，任何关系变量都可以无损分解为相应的 4NF 关系变量的集合。虽然在第 12 章的 12.5 节中对 SJT 示例的讨论表明，在某些情况下不必分解得如此之细（甚至进一步规范化到 BCNF 都不必要）。

注意：Rissanen 的投影独立理论［12.6］虽然是以函数依赖为依据的，但是也适用于多值依赖。前面也谈及一个满足函数依赖 $A \rightarrow B$ 和 $B \rightarrow C$ 的关系变量 $R\{A,B,C\}$，将其分解为在 $\{A, B\}$ 和 $\{B, C\}$ 上的投影要好于分解为在 $\{A, B\}$ 和 $\{A, C\}$ 上的投影。如果我们将这里的函数依赖替换为多值依赖，这个定理依然成立。

最后，回到消除关系值属性（简称为 RVA）的问题上来，方法如下：如果开始时一个关系变量中存在两个或多个相互独立的 RVA（像关系变量 HCTX 一样），那么所要做的第一件事就是将这些 RVA 进行无损分解，而不是简单地用标量属性来替代这些 RVA（正如本节前面所做的那样）。例如，在关系变量 HCTX 的例子中，第一件事就是将原来的关系变量替换为它的两个投影 HCT {COURSE, TEACHERS} 和 HCX {COURSE, TEXTS}（TEACHERS 和 TEXTS 仍旧是 RVA）。然后这两个投影中的 RVA 可以用标量属性来替代，并且在必要时用通常的方式把它们归约为 BCNF（实际上是 4NF）而且满足 BCNF 范式的关系变量 CTX 不会产生这种问题。正是 MVD 和 4NF 为上述的分解提供了一个理论基础，否则这将只是一种凭经验做而做出的结论。

## 13.3　连接依赖与第五范式

到目前为止，在本章（并且在第 12 章）中都默认，在进一步规范化过程中可行并且必要的唯一的操作是用一种无损方法将一个关系变量用它的两个投影来代替。这种方法成功地进行到了第四范式。因此，如果关系变量不是无损分解为两个投影而是无损分解为三个投影或更多，可能会让人觉得很奇怪。这样的关系变量可描述为“$n$ 分解”（$n>2$），表示对于任意 $m<n$ 其中 $1<m$ 并且 $m<n$，关系变量可以被无损分解为 $n$ 个投影，而不是为 $m$ 个投影。一个关系变量可以被无损分解为两个投影，则称之为“可 2 分解”的。注意：$n>2$ 时的 $n$ 分解现象是由 Aho、Beeri 和 Ullman 提出的［13.1］，而 Nicolas 还研究了 $n=3$ 时的情况［13.26］。

考虑从供应商 - 零件 - 工程数据库中提取的关系变量 SPJ（为简单起见忽略 QTY 属性）；它的一个简单的值在图 13-4 的上半部表示出来。可以看到关系变量 SPJ 是全码，并且根本不包含非平凡的函数依赖或多值依赖，因此 SPJ 属于 4NF。图 13-4 还显示了：

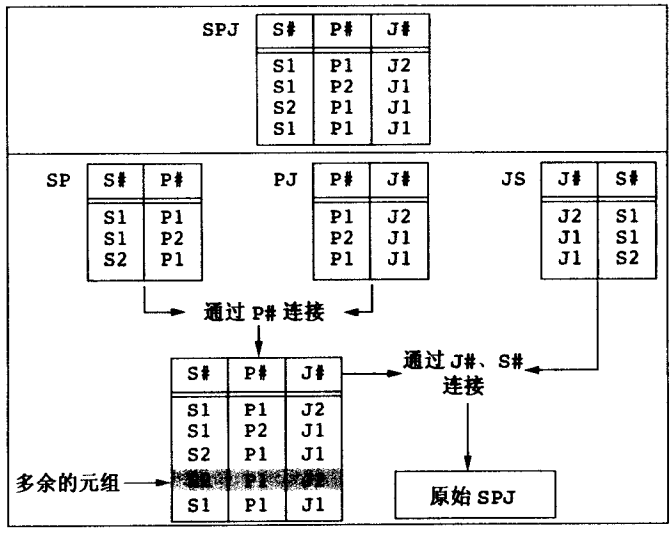



图 13-4　关系 SPJ 是三个二元投影的连接而非其中任何两个的连接

1）与显示在图顶端的 SPJ 的相关值相对应的三个二元投影 SP、PJ 和 JS。

2）将 SP 和 PJ 投影连接起来的结果（通过 *P#*）。

3）将上一步的结果和 JS 投影相结合的结果（通过 *J#*和 *S#*）。

可以观察到第一个连接的结果是产生一个原始 SPJ 关系的副本加上一个多余的元组，第二个连接的结果是消除多余的元组，从而复原为原始的 SPJ 关系。换句话说，原始的 SPJ 关系是可 3 分解的。注意：无论我们选择哪两个投影作为第一次连接，结果都是一样的，虽然在每种情况下中间结果不同。（练习：证明这一命题。）

现在，图 13-4 的例子中是给出关系，而不是关系变量。然而，SPJ 的可 3 分解性是基本的、与时间无关的特性，即它是关系变量的所有合法值满足的特性，如果这个关系变量满足一个特定的与时间无关的完整性约束的话。要理解这个完整性约束的意义，必须首先看到“SPJ 与三个投影 SP、PJ 和 JS 的连接等价”的说法与下面的说法完全相同：

**如果**（*s*1，*p*1）在 SP 中出现，

并且（*p*1，*j*1）在 PJ 中出现，

并且（*j*1，*s*1）在 JS 中出现，

**那么**三元组（*s*1，*p*1，*j*1）在 SPJ 中出现。

因为三元组（*s*1，*p*1，*j*1）显然出现在 SP、PJ 和 JS 的连接中（注意：反之，“如果（*s*1，*p*1，*j*1）出现在 SPJ 中，那么（*s*1，*p*1）出现在投影 SP 中，等等”这一说法对于任意的三元关系 SPJ 来说都是正确的）。对于一个 *j*2，（*s*1，*p*1）出现在 SP 中，当且仅当（*s*1，*p*1，*j*2）出现在 SPJ 中，并且对于（*p*1，*j*1）和（*j*1，*s*1）的情况也类似，那么我们就可以将上面的综述作为 SPJ 的一个约束。复述如下：

**如果**（*s*1，*p*1，*j*2），（*s*2，*p*1，*j*1）和（*s*1，*p*2，*j*1）在 SPJ 中出现，

**那么**（*s*1，*p*1，*j*1）也在 SPJ 中出现。

如果这种说法在任何情况下都成立（例如关系变量 SPJ 的所有可能的合法值）那么这个关系变量确实存在一个与时间无关的约束（虽然是一个很奇异的约束）。可以看到这个约束的**循环性**（“如果 *s*1 与 *p*1 连接，*p*1 与 *j*1 连接，而 *j*1 又与 *s*1 连接，那么 *s*1、*p*1 和 *j*1 一定是在一个元组中同时出现的”）。当 $n>2$ 时，一个关系变量是可 $n$ 分解的，当且仅当它满足这样的 $n$ 路的循环约束。

于是可以认为，关系变量 SPJ 确实满足时间无关约束（图 13-4 中的例子是符合这一假设的）。可以将这种约束简化为 3D 约束（3D 代表可 3 分解的）。在现实条件下，3D 约束意味着什么呢？通过一个例子可以将其具体化。这个约束表明，在所假设的关系变量 SPJ 表示的那部分现实世界当中，**如果**（例如）

a）史密斯提供活动扳手，

b）活动扳手在曼哈顿项目中使用，

c）史密斯为曼哈顿项目提供支持，

**那么**

d）史密斯提供活动扳手给曼哈顿项目。

注意：（正如在第 1 章的 1.3 节所指出的）a、b 和 c 一起出现通常并不表示 d 也一定出现；事实上，这个例子在第 1 章中是作为“连接陷阱”（connection trap）的一个说明提出的。然而，在现在的例子中并不存在陷阱——因为存在一个附加的现实世界约束在起作用，即 3D 约束，使得在这个特殊的例子中从 a、b 和 c 中推演出 d 有效。

回到所讨论的主题：因为 3D 约束被满足当且仅当所考虑的关系变量与其特定的投影的连接结果等价，所以这种约束被称为连接依赖（JD）。当 MVD 或 FD 是某关系变量中的约束时，JD 才是其中的约束。下面是其定义：

- **连接依赖**：如果 *R* 是一个关系变量，并且 *A*，*B*，…，*Z* 都是 *R* 的属性子集，那么称 *R* 满足 JD

  ```
  * { A, B, ..., Z }
  ```

（读做“星 $A$，$B$，…，$Z$”）当且仅当 $R$ 的任何可能出现的合法值都与它在 $A$，$B$，…，$Z$ 上的投影的连接等价。

例如，如果用 SP 来表示 SPJ 的属性集的子集 {S#，P#}，同样也用 PJ 和 JS 来表示 SPJ 的另两个属性集，那么关系变量 SPJ 满足 JD *{SP，PJ，JS}。

另举一个例子，考虑通常的供应商关系变量 S。如果我们使用 SN 表示 S 的属性集的子集 {S#，SNAME}，ST 和 SC 与之类似，那么我们可以说关系变量 S 满足连接依赖*{SN，ST，SC}。

那么，可以看到拥有连接依赖*{SP，PJ，JS} 约束的关系变量 SPJ，是可 3 分解的。但问题是，应该让它可 3 分解吗？答案是“可以”。关系变量 SPJ（存在 JD 约束）存在一大堆的更新异常，当它被 3 分解之后这些问题就迎刃而解了。类似的更新异常例子在图 13-5 中表示。而在 3 分解之后将会有什么情况发生作为练习留给大家思考。

SPJ

| S# | P# | J# |
|---|---|---|
| S1 | P1 | J2 |
| S1 | P2 | J1 |

- 如果 {S2,P1,J1} 被插入，那么 {S1,P1,J1} 也一定被插入。
- 然而反之并不正确。

SPJ

| S# | P# | J# |
|---|---|---|
| S1 | P1 | J2 |
| S1 | P2 | J1 |
| S2 | P1 | J1 |
| S1 | P1 | J1 |

- 可以无副作用地删除 {S2,P1,J1}。
- 如果 {S1,P1,J1} 被删除，那么另一个（哪一个？）元组也要被删除。

图 13-5　在 SPJ 中的更新异常实例

Fagin 定理（见 13.2 节）表明，$R$ {$A$，$B$，$C$} 可以被无损分解为它在 {$A$，$B$} 和 {$A$，$C$} 上的投影，当且仅当在 $R$ 中存在多值依赖 $A \twoheadrightarrow B$ 和 $A \twoheadrightarrow C$。等价于：

- $R${$A$,$B$,$C$} 满足连接依赖*{$AB$，$AC$}，当且仅当它满足多值依赖 $A \twoheadrightarrow B \mid C$。

既然这个定理可以被用做多值依赖的定义，这就表明多值依赖是一种特殊形式的连接依赖，或者说（等价）连接依赖是一种一般化的多值依赖。形式化地表示，可以有

$$A \twoheadrightarrow B \mid C \equiv *\{AB, AC\}$$

注意：从定义可以看到，连接依赖是**可能的依赖中最一般化的一种形式**（在某种特殊的意义上使用了术语“依赖”）。这就是说，只要把对依赖的理解限制在对关系变量通过投影来分解、通过连接来重组的处理框架内，就不存在一种更高形式的依赖，使得连接依赖仅仅是这种更高形式的依赖的特殊情况。（然而，如果允许通过其他的分解和重组操作，那么其他类型的依赖就可能会出现。这种可能性将在 13.7 节简单地加以讨论。）

回到图 13－5 的例子中可以看到，关系变量 SPJ 的问题是它包括不是多值依赖（当然也不是函数依赖）的连接依赖（练习：为什么存在这个问题?）。可以看到，将关系变量 SPJ 分解为更小的部分——即分解为由连接依赖指定的投影是可能的，而且也是我们所期望得到的。这种分解过程重复下去，得到的关系变量称为第五范式。其定义如下：

- **第五范式**：一个关系变量 $R$ 是第五范式——也称为**投影－连接范式**（PJ/NF）——当且仅当 $R$ 的候选码蕴含 $R$ 的每一个非平凡的连接依赖。其中：

  a）$R$ 的连接依赖*{$A$，$B$，…，$Z$} 是平凡的，当且仅当 $A$，$B$，…，$Z$ 中的至少一个是 $R$ 中所有属性的集合。

  b）$R$ 的连接依赖*{$A$，$B$，…，$Z$} 被 R 的**候选码所蕴涵**，当且仅当 $A$，$B$，…，$Z$ 中的每一个都是 $R$ 的超码。

关系变量 SPJ 并不属于 5NF；它满足一个特定的连接依赖，即 3D 约束，这显然没有被其唯一的候选码（这个候选码是 SPJ 所有的属性值的组合）所蕴涵。可以表示其区别如下：关系变量 SPJ 并不属于 5NF，因为（a）它是可以被 3 分解的；（b）可 3 分解性并没有为其 {S#，P#，J#} 是一个候选码的事实所蕴涵。相反，3 分解后，由于三个投影 SP、PJ 和 JS 根本不包括任何

（非平凡的）连接依赖，因此它们都属于5NF。

注意，任何属于5NF的关系变量也都自动属于4NF，因为（如我们所见的）多值依赖是连接依赖的一种特殊情况。事实上，在参考文献［13.15］中，Fagin提出候选码所蕴涵的多值依赖必须是一个函数依赖，在这个函数依赖中，候选码是决定因素。同时，Fagin还提出任何给定的关系变量都可以被无损分解为一个相等的5NF关系变量集合，这就是说，5NF是可以达到的。

现在解释一下连接依赖为候选码所蕴涵的含义。仍以我们所熟悉的供应商关系变量 $S$ 为例，该关系变量满足几个连接依赖。例如，它满足连接依赖

```
* { { S#, SNAME, STATUS }, { S#, CITY } }
```

这就是说，关系变量S和它在｛S#，SNAME，STATUS｝和｛S#，CITY｝上的投影的连接是相等的，因此能被无损分解为这些投影（这并不表示它应该被分解，而是表示可以分解）。这个连接依赖被｛S#｝是一个候选码的情况所蕴涵；事实上它是被Heath的定理［12.4］所蕴涵。

现在假设关系变量S有第二个候选码｛SNAME｝，并且关系变量S也满足连接依赖（如第12章12.5节所设）

```
* { { S#, SNAME }, { S#, STATUS }, { SNAME, CITY } }
```

这个连接依赖是由｛S#｝和｛SNAME｝两个候选码所蕴涵的。

前面的两个例子都表明了，一个给定的连接依赖 $*\{A,B,\ldots,Z\}$ 是由候选码所蕴涵的，当且仅当每一个 $A,B,\ldots,Z$ 都是所讨论的关系变量的超码。因此，给定一个关系变量 $R$，只要知道 $R$ 中所有的候选码和所有的连接依赖就可以分辨出 $R$ 是否属于5NF。然而，所有的连接依赖可能自身是一个非平凡的操作。这就是说，尽管相对来说分辨函数依赖和多值依赖是很简单的（因为它们是对现实世界的直接解释），但对那些既不是多值依赖也不是函数依赖的连接依赖却并非如此——因为连接依赖的意义可能不是很直观。因此确定一个给定的关系变量属于4NF而不属于5NF，并且决定是否可以因此通过分解逆规范化的方式来规范给定关系变量的过程依然不清楚。经验表明这样的关系变量在实际中是很少见的。

综上所述，考虑到投影和连接，从定义上看5NF是**最终的范式**（这也解释了5NF的另一个名字，即投影-连接范式）。这就是说，一个5NF的关系变量是通过投影的方式来**保证没有更新异常**的情况。因此，如果一个关系变量属于5NF，那么所有的连接依赖都是由其候选码所蕴涵的，正确的分解方式就是基于这些候选码的（在这样的分解中，每一个投影都包含一个或多个这样的候选码，再加上零个或多个其他的属性）。例如，供应商关系变量S是属于5NF的。如前所述，它可以通过几种无损的方式被进一步分解，在这种分解中的每一个投影都将包含一个原有关系变量的候选码，并且在进一步的分解中看不到明显的优势。

## 13.4　规范化过程小结

到现在为止，在本章（以及前一章）中，作为数据库设计的目标，已经讨论了无损分解的技术。基本的观点如下：给定某一1NF的关系变量和 $R$ 的一些函数依赖、多值依赖和连接依赖，系统地将 $R$ 归约为一组“更小”（即，度更低）的关系变量，这些关系变量在一定的意义上是与 $R$ 相等的，但在某些方面比 $R$ 更优[⊖]。归约过程的每一步都包括由前一步的结果来投影关系变量。在每一步中用给定的约束来指导下一步投影的选择。整个过程可以非形式化地描述为如下一组规则：

1）对原始的1NF的关系变量进行投影，消除任何不可约的函数依赖。这一步将产生一个2NF关系变量集合。

2）对2NF的关系变量进行投影，消除任何可传递的函数依赖。这一步将产生3NF关系变量的集合。

---

⊖ 我们假定 $R$ 中只包含需要的RVA（如果包含的话），不需要的RVA可以用13.2节讨论的方法去除。

3）对 3NF 的关系变量进行投影，消除任何决定因素不是候选码的函数依赖。这一步将产生 BCNF 关系变量的集合。

注意：规则 1 ~ 3 可以合并为一个规则，即“对原始的关系变量进行投影，消除所有的决定因素不是候选码的函数依赖”。

4）对 BCNF 的关系变量进行投影，消除任何不是函数依赖的多值依赖。这一步将产生 4NF 的关系变量的集合。注意：在实践中通常是——通过“分离独立的 RVA”的方式来实现，正如在 13.2 节中对 CTX 的例子所解释的——消除这样的多值依赖之后再应用以上的规则 1 ~ 3。

5）对 4NF 的关系变量再进行投影，如果存在任何不被候选码所蕴涵的连接依赖，则加以消除。这一步将产生 5NF 关系变量的集合。

从前面的小结中提出以下几点：

1）首先，在每一步进行的投影过程必须是基于一种无损的方式，并且最好是用保持原有依赖的方法来投影。

2）观察到（由 Fagin 首先提出，见参考文献［13.15］）在 BCNF、4NF 和 5NF 的定义中有一种非常引人注目的对应关系，即：

- 一个关系变量 *R* 是 BCNF，当且仅当 *R* 中的每一个函数依赖都被 *R* 的候选码所蕴涵。
- 一个关系变量 *R* 是 4NF，当且仅当 *R* 中的每一个多值依赖都被 *R* 的候选码所蕴涵。
- 一个关系变量 *R* 是 5NF，当且仅当 *R* 中的每一个连接依赖都被 *R* 的候选码所蕴涵。

在第 12 章以及在本章的前几节中讨论的更新异常的问题正是由于那些不被候选码所蕴涵的 FD、MVD 和 JD 所引起的。（这里涉及的 FD、MVD 和 JD 都假设是非平凡的。）

3）规范化过程的总体目的如下：

- 消除某些冗余。
- 避免更新异常。
- 产生一种可以很好代表现实世界的设计——一个很直观并且方便未来扩充的设计。
- 可以简单地满足某些完整性约束。

我们对以上列出的最后一条稍作详细的介绍。通常的观点是（从本书其他部分的论述也可得到这样的观点）一些完整性约束隐含其他的完整性约束。举一个例子，一个工资必须大于 10 000元的约束肯定是蕴涵工资大于 0 的约束的。现在，如果完整性约束 *A* 蕴涵 *B*，那么满足 *A*，则一定自动满足 *B*（甚至没必要明确标明 *B* 完整性，除非是用注释的形式）。规范到 5NF 的规范化提供了一种满足某些重要而又经常发生的约束的简单方法；所需要的工作就是满足候选码的唯一性，这样所有的连接依赖就会被自动满足——因为所有的这些连接依赖（和多值依赖以及函数依赖）都会被候选码所蕴涵。

4）这里再一次强调规范化只是作为一种参考而已，有时偶尔也可能不需要完全规范化。有关姓名和地址的关系变量 NADDR（参见第 12 章习题 12.7）是这一情况的典型例子，但是一般来说，不完全规范化是不可取的。

5）再重复一下，第 12 章中关于依赖和进一步规范化的概念本质上都是语义的概念。换句话说，它们关注数据的意义。相反，基于这些形式的关系代数、关系演算以及像 SQL 这样的语言，却只关注实际数据的值；它们除了要求满足 1NF 外并不要求任何特定级别的规范化。进一步规范化方针应当主要被看作是辅助数据库设计的规则（因此对用户也有帮助）——这种规则帮助设计者用一种简单而又易懂的方式来对现实世界的一部分进行语义描述。

6）承接前面的观点：规范化的思想对数据库的设计是有用的，但它们并不是秘方良药。以下是几点原因（在参考文献［13.9］中有详细介绍）：

- 规范化确实可以协助满足某些完整性约束并使其过程变得简单，但是（如在第 9 章所述）连接依赖、多值依赖和函数依赖并不是唯一可以在实践中应用的约束。
- 分解方法可能并不是唯一的（事实上，通常有许多种将一组给定的关系变量归约为 5NF 的方法），并且还有几个用来选择分解方法的有针对性的标准。
- 根据 12.5 节的解释（“SJT 问题”），BCNF 范式和保持依赖的目标可能是有冲突的。

- 规范化过程通过投影消除了冗余，但是并不是所有的冗余都是可以用这种方式来消除的（“CTXD 问题”，见参考文献［13.14］中的解释）。

无论如何应该指出，好的自上而下的设计方法都趋向于充分规范化的设计（参见第 14 章）。

## 13.5　逆规范化

根据本章（以及前一章中）的观点，我们理所当然地应该全面规范到 5NF。然而在实践中，为取得好的性能，“逆规范化”又是必需的。这个观点的内容如下：

1）完全的规范化意味着会出现许多逻辑上分离的关系变量（并且我们这里假定所讨论的关系变量是特定的基本关系变量）。

2）许多逻辑上分离的关系变量意味着许多物理上分离的存储文件。

3）许多物理上分离的存储文件意味着会有许多 I/O 操作。

严格地说，这种观点当然是不合理的，因为（在本书的其他部分有论述）关系模型并没有保证基本关系变量必须一一映射到存储文件中。如果必要的话，逆规范化应该在存储文件的层次上操作而不是在基本关系变量的层次上操作[⊖]。但是这种观点对现在的 SQL 产品来讲是有效的，因为在这些产品中存在着不同级别的不完全分离。因此在这一节，我们仔细地看一下“逆规范化”的概念。注意：下面的讨论是基于参考文献［13.6］的。

### 1. 逆规范化的含义

简单地回顾一下：**规范化**一个关系变量 $R$ 就意味着将 $R$ 用一组投影 $R1,R2,\cdots,Rn$ 来代替，使得 $R$ 等价于 $R1,R2,\cdots,Rn$ 的连接。总体的目标是通过确定每一个投影 $R1,R2,\cdots,Rn$ 都处于最高级的规范化来减少冗余。

为定义逆规范化，先设 $R1,R2,\cdots,Rn$ 是一组关系变量。**逆规范化**这些关系变量就意味着将它们替换为它们的连接，$Ri(i=1,2,\cdots,n)$ 的属性值可通过投影 $R$ 得到。总体的目标是增加冗余，通过确认 $R$ 处于比关系变量 $R1,R2,\cdots,Rn$ 更低的规范化级别来完成。更具体地说，目的是在数据库设计中事先建立一些连接，从而减少执行中的连接代价。

例如，我们可以考虑逆规范化关系变量零件和发货量来产生一个关系变量 PSQ，如图 13-6 所示[⊖]。可以看到关系变量 PSQ 属于 1NF 而不属于 2NF。

| PSQ | P# | PNAME | COLOR | WEIGHT | CITY | S# | QTY |
|---|---|---|---|---|---|---|---|
| | P1 | Nut | Red | 12.0 | London | S1 | 300 |
| | P1 | Nut | Red | 12.0 | London | S2 | 300 |
| | P2 | Bolt | Green | 17.0 | Paris | S1 | 200 |
| | .. | ..... | ..... | ..... | ..... | .. | ... |
| | P6 | Cog | Red | 19.0 | London | S1 | 100 |

图 13-6　逆规范化零件与发货量

### 2. 相关问题

逆规范化的概念引起许多很著名的问题。一个很明显的问题是，一旦我们开始逆规范化，并不清楚在哪里终止。对于规范化来说，有很合情合理的原因使其一直继续到可能的最高的规范化形式；那么在逆规范化中我们是否要达到最低的规范化形式呢？当然不是；然而并没有合理的标准确切地决定在哪里终止。换句话说，在选择逆规范化时，没有什么固定的科学理论的支持，而是通过纯粹的经验主观决定的。

第二个明显的问题是存在冗余和更新问题，这正是因为所处理的关系变量是没有完全规范化

---

⊖ 这一评论不是十分准确；逆规范化是在关系变量上的操作，而不是在存储文件上的操作，所以它不能被应用“在存储文件的层次上”。但是假定逆规范化的某个类似操作可以在存储文件的层次上执行并不是没有意义的。

⊖ 如果使用我们通常的示例数据，逆规范化供应商和发货量就存在问题。因为供应商 S5 在连接中丢失。因此，有人可能认为，在逆规范化过程中应该使用外连接。但是正如我们将在第 19 章看到的，外连接也存在固有的问题。

的。这些问题已经详细地讨论过了。此外，还可能会有检索问题；这就是说，逆规范化实际上使查询更加难以表示。例如："对于每一种零件颜色，都给出它们的平均重量"的查询。如果使用通常的规范化的设计，适当的公式是：

```
SUMMARIZE P BY { COLOR } ADD AVG ( WEIGHT ) AS AVWT
```

然而对于图 13-6 中的逆规范化设计，这个公式就有些蹊跷（更不必说它是依赖于通常是不合理的假定——每一个零件确实有至少一个发货量与其对应）：

```
SUMMARIZE PSQ { P#, COLOR, WEIGHT } BY { COLOR }
                          ADD AVG ( WEIGHT ) AS AVWT
```

（注意：该公式执行起来可能效率很差）。换句话说，就可用性和性能两方面因素考虑，通常逆规范化是"对检索有利而对更新不利"的理解是错误的。

第三个（也是最主要的）问题如下（这一点适用于"适当"的逆规范化，即只在存储文件层进行的逆规范化，也适用于那些在当今的 SQL 产品中有时必须进行的逆规范化）：当提到逆规范化"对性能有益"时，实际意味着它是对具体的应用的性能有益。任何给定的物理设计都是对某些应用有益而对其他的应用没好处（根据性能来说）。例如，假定每一个基本关系变量确实映射到一个物理存储文件，并且还假定每一个物理存储文件包括一组物理上相邻的存储记录，每一个记录对应一个相应的关系变量上的元组。那么：

- 假定将供应商、发货量和零件的连接作为一个基本关系变量，并因此给出一个存储文件。那么"取出提供红色零件的供应商的详细信息"的查询可能会在这种物理结构下运行得较好。
- 然而，"取出在伦敦的供应商的详细信息"的查询在这种物理结构下，将比使用三个基本关系变量并且将它们映射到三个物理上分离的存储文件中的结果要差。原因是在后一个设计中，所有的供应商的信息将被在物理上邻接地存储；然而在前一个设计中，它们在物理上分散于一个较大的空间中，因此将需要更多的 I/O 操作。类似的结论也适用于任何其他的只访问供应商、只访问零件或只访问发货量的查询，而不适用于一些连接的查询。

## 13.6　正交设计

本节我们接着参考文献［13.12］简要地讨论另外一种数据库设计的原理，一个在本质上不是进一步规范化的设计原理，但它确实与规范化很相像，因为它至少是很科学的。该原理称为正交设计准则（The Principle of Orthogonal Design）。图 13-7 中显示了一种对供应商的设计，它显然不够好，但是很可能存在这种设计；在这个设计中，关系变量 SA 对应于那些居住在 Paris 的供应商，关系变量 SB 对应于那些或不居住在巴黎，或状态大于 30 的供应商（不严格地说，这些是关系变量谓词）。如图中所示，这个设计导致了某些冗余；更具体地说，供应商 *S3* 的元组出现了两次，每个关系变量各一次。而且这些冗余又导致了更新异常。

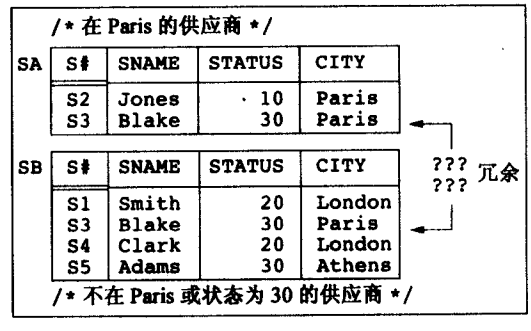



图 13-7　供应商：不好但可能存在的一种设计

注意：供应商是 *S3* 的元组会在两个关系变量中都出现。相反，假设这个元组要在 SB 中出现，而不在 SA 中出现。在 SA 中应用封闭世界的假设，SA 就会告诉我们供应商 *S3* 并不在 Paris，然而，SB 告诉我们供应商 *S3* 在 Paris。换句话说，这将产生一个冲突，并且数据库会变得不一致。

当然，在图 13-7 中的设计问题是很明显的，正是由于允许同样的元组在两个截然不同的关系变量中出现才会导致这样的问题。换句话说，这两个关系变量有相互重叠的含义，即可能出现

对同一个元组满足两个关系变量谓词的情况。因此，有如下的显而易见的规则：

- **正交设计准则**（最初版本）：对于一个给定的数据库，任何两个基本关系变量不应该有重叠的含义。

要点如下：

1）回顾一下第 10 章，从使用者的角度看，所有的关系变量都是基本关系变量（除了作为简便方式才定义的关系变量之外）。换句话说，这个原理适用于所有“可表达”的数据库的设计，而不只限于“真实”的数据库——这就是所谓的数据库相对性原则（当然，类似的观点也适合规范化理论）。

2）注意，两个关系变量可能并没有互相重叠的含义，除非它们是同一种类型（也就是说，除非它们有相同的标题）。

3）正交设计准则意味着：（举例来说）当插入一个元组时，可以认为这个操作是将一个元组插入一个数据库而不是插入一个具体的关系变量——因为最多有一个关系变量是满足新元组插入谓词的。

现在，我们简要阐述一下最后一个要点。当插入一个元组时通常要指定元组插入的关系变量 *R* 的名字。但是，这并不与前面的观点矛盾。事实上，名字 *R* 只是相应的谓词，比如 PR 的简写；真实的意思是说“插入元组 *t*，并且 *t* 满足谓词 PR”。更进一步讲，*R* 有可能是一个视图，可能是由 *A* 与 *B* 的并集表达式来定义的视图，并且正如在第 10 章所见的，系统如果能知道新的元组是插入 *A*、*B*，还是都插入，那么将是最好不过的。

事实上，与上面的论述类似的观点也适用于其他所有的操作；在所有的情况下，关系变量名字实际上都只是关系变量谓词的简写。当表示数据语义这样的观点时，不应该太强调这是谓词，而不仅是名字。

到现在为止还没有讨论完正交设计准则——还有一个很重要的细化工作要做。考虑图 13-8，它显示了有关供应商的另一个不好但却可能存在的设计。其中存在两个关系变量，它们本身并不包含重叠含义，但是它们在 {S#, SNAME} 上的投影确实出现了重叠（事实上，这两个投影的含义是类似的）。因此，如果想插入一个元组（S6, Lopez）到一个由这两个投影的并集定义的视图中就会引起元组（S6, Lopez, *t*）被插入到 SX 中、元组（S6, Lopez, *c*）被插入到 SY 中（其中 *t* 和 *c* 都是可用的默认值）。很明显，正交设计准则需要扩展以便将图 13-8 中的问题考虑进去：

SX

| S# | SNAME | STATUS |
|---|---|---|
| S1 | Smith | 20 |
| S2 | Jones | 10 |
| S3 | Blake | 30 |
| S4 | Clark | 20 |
| S5 | Adams | 30 |

SY

| S# | SNAME | CITY |
|---|---|---|
| S1 | Smith | London |
| S2 | Jones | Paris |
| S3 | Blake | Paris |
| S4 | Clark | London |
| S5 | Adams | Athens |

图 13-8　有关供应商的另一个不好但却可能存在的设计

- **正交设计准则**（最终版本）：*A* 和 *B* 是数据库中任意的两个基关系变量。那么并不存在无损分解将 *A* 和 *B* 分别分解为 *A*1，…，*Am* 和 *B*1，…，*Bn* 使在 *A*1，…，*Am* 中的某些投影 *Ai* 和在 *B*1，…，*Bn* 中的某些投影 *Bj* 有重叠的含义。

要点如下：

1）术语“无损分解”在这里是其通常的含义——即分解为一组投影，使得：

- 给定的关系变量可以通过将投影连接回去的方法重新得到；
- 在重新连接的过程中这些投影中没有冗余的项。（严格地说，第二个条件并不是无损分解所必需的，但是如我们在第 12 章所见，它通常是有用处的。）

2）这个版本蕴涵了最初的版本，因为关系变量 *R* 就是其本身的一个投影，因此无损分解是肯定存在的。

**深入观察**

下面给出一些关于正交设计准则的评论。

1）首先，术语“正交”——取自这一设计准则的实际含义，即基本关系变量是互相独立的（没有重叠含义）。当然，这个准则是常识性的，但它是形式化意义上的常识（就如同规范化理

论一样）。

2）假定以一个通常的供应商关系变量 S 开始，但是为了设计的原因将这个关系变量分解为一组投影。那么正交设计准则会告诉我们这些投影应该都是互不相交的，在这种意义上讲，没有一个供应商的元组会出现在多于一个的投影中。（当然，这些投影的并必须能够还原原始的关系变量。）这种分解称为**正交分解**。

3）正交设计的总体目标是减少冗余，并因此避免更新异常（也与规范化类似）。事实上，它补充了规范化理论，不严格地说，规范化在关系变量中减少了冗余，而正交性在关系变量之间减少了冗余。

4）正交性可能是一个常识，但是它在实践中经常不受重视（事实上，这种不受重视的看法有时甚至是受推崇的）。像下面的这个设计是很常见的，它取自金融数据库：

```
ACTIVITIES_2001 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2002 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2003 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2004 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
ACTIVITIES_2005 { ENTRY#, DESCRIPTION, AMOUNT, NEW_BAL }
```

事实上，将含义编入（关系变量或其他的名字）违反了信息原则（The Information Principle），这个准则提出，所有的数据库信息必须显式地根据值的方法而不能用其他方法来表示。

5）如果 *A* 和 *B* 是同一类型的基本关系变量，根据正交设计准则蕴涵着：

```
A UNION B     : 总是一个不相交的并集
A INTERSECT B : 总是空的
A MINUS B     : 总是等于 A
```

## 13.7　其他的规范化形式

再回到对规范化的讨论。在第 12 章的引言中提到过，除了这几章讨论的范式，确实存在其他的规范化形式。实际情况是，规范化理论和有关问题（现在通常被认为是依赖理论）已经发展为一个拥有大量文献著作的相当可观的领域。这个领域的研究在不断继续、深入。有关这方面的更深入的讨论已超出本章的范围；这个领域的研究（20 世纪 80 年代中期的研究）的综述参见［13.18］，近期研究的综述参见［11.1］和［11.3］。在这里只提几个具体的问题。

1）**域码**（domain-key）**范式**：域码范式（DK/NF）在参考文献［13.16］中由 Fagin 提出。DK/NF（与所讨论过的规范化形式不同）根本不是根据函数依赖、多值依赖或连接依赖而定义的。相反，当且仅当一个关系变量 *R* 中的每一个约束都是 *R* 中所应用的域约束和码约束的合理结果时，称 *R* 属于 DK/NF。

- **域约束**（在这里用到的一个术语）是指有关指定属性值从一些规定的域中取值的约束（在第 9 章的术语中，这样的约束是一个属性约束而不是一个域约束）。
- **码约束**是有关某一个特定的属性或属性组构成的候选码的约束。

满足一个 DK/NF 关系变量上的约束因此在概念上显得很简单，既然足以满足“域”（属性）和码约束，那么所有的其他约束就会自动地被满足。还要注意在这里“所有的其他约束”意味着不止是函数依赖、多值依赖和连接依赖。事实上，它代表整个关系变量谓词。

Fagin 在文献［13.16］中提出，任何 DK/NF 关系变量都必须属于 5NF（并且因此也属于 4NF 等）。事实上在（3，3）范式中也是如此（参见第 2 点）。然而，DK/NF 并不是经常可以达到的，而“何时能够达到”的问题也没有解决。

2）“**选择 - 并**”（restriction-union）**范式**：再次考虑供应商关系变量 S。规范化理论告诉我们，关系变量 S 是一种“好”的范式；事实上，它属于 5NF，并且可以通过投影的方式消除异常。但是为什么将所有的供应商放在一个单独的关系变量中呢？如果设计将伦敦的供应商放入一个关系变量（如 LS）中，而将巴黎的供应商放入另一个（如 PS）关系变量中，等等，这样的关系变量可不可以呢？换句话说，可不可以将原来的供应商的关系变量按选择来分解而不是通过投

影来分解呢？这样的设计结果将是好的还是不好的呢？（事实上几乎可以肯定它是不好的——参见第 8 章中习题 8.8——但是经典的规范化标准化理论对此类问题没有触及）。

因此，对于规范化研究的另一个方向就是研究通过非投影方式对关系变量进行分解的问题。在这个例子中，已经提到分解运算符是选择；而相应的合成运算符是并。因此，构建一个类似投影 - 连接规范化理论的“选择 - 并”规范化理论是可行的[⊖]。就本书作者的了解，还没有任何这样的理论被详细地提出来，但是一些最初的观点可以在 Smith 的论文［13.32］中找到，在其中定义了一个称为“（3，3）范式”的新型范式。（3，3）范式蕴涵了 BCNF 范式；然而，一个（3，3）范式关系变量不必属于 4NF，4NF 也不必属于（3，3）范式，所以（如上面表明的）推演到（3，3）范式是和推演到 4NF（和 5NF）相互正交的。对此观点的深层研究出现在文献［13.15］和［13.23］中。正交设计准则也与此相关［13.12］（因此正交设计最终可以被看作是一种规范化）。

3）**第六范式**：如果限于考虑传统的投影和连接，那么 5NF 是最终的范式。但是在第 23 章将看到，我们可能（并且需要）定义：（a）这些操作的一般化形式；（b）连接依赖的一般化形式；（c）一个新的（第六）范式，6NF。注意，使用“第六范式”作为它的名字是合理的，因为 6NF 确实在从 1NF，2NF…到 5NF 的路上前进了一步。而且，所有符合 6NF 的关系变量必然属于 5NF。详细的说明见第 23 章。

## 13.8　小结

本章完成了关于**进一步规范化**理论的讨论（从第 12 章开始）。本章讨论了**多值依赖**（MVD），多值依赖是一种一般化的函数依赖；还讨论了**连接依赖**（JD），它是多值依赖的一般化形式。大致地说：

- 一个关系变量 $R\{A,B,C\}$ 满足多值依赖 $A \twoheadrightarrow B \mid C$，当且仅当一组 $B$ 值与一给定的 $\{A, C\}$ 对匹配与否，只依赖于 $A$ 的值；对 $C$ 与 $\{A, B\}$ 的情况也是如此。这样的一个关系变量可以被无损分解为它在 $\{A, B\}$ 和 $\{A, C\}$ 上的投影；事实上，多值依赖是这种分解的充要条件（Fagin 定理）。
- 一个关系变量 $R\{A,B,\cdots,Z\}$ 满足连接依赖 $*\{A,B,\cdots,Z\}$，当且仅当它与它在 $A,B,\cdots,Z$ 上的投影的连接结果相等。这样的一个关系变量（显然）可以被无损分解为这些投影。

一个关系变量属于 4NF，仅当它满足的非平凡多值依赖是非超码的函数依赖。一个关系变量属于 5NF——也称为**投影 - 连接**范式，即 PJ/NF——当且仅当它满足的非平凡连接依赖是非超码的函数依赖（意思是如果连接依赖是 $*\{A, B, \cdots, Z\}$，那么 $A$，$B$，…，$Z$ 中每一个都是超码）。5NF（总可以达到的）是关于投影和连接运算的最终范式。

我们还讨论了**规范化过程**，给出了非形式化的步骤和一些相关的解释。然后描述了**正交设计准则**：大致上说，不应该有两个关系变量有含义互相重叠的投影。最后，简单地提到了一些其他的范式。

综上所述，应该指出，对上面所讨论的这些问题的研究是很有意义的。原因是“进一步规范化”（或习惯地称为**依赖理论**的研究）确实描绘出了数据库设计这一领域的一些科学规律，因为数据库设计以往太过工艺化（即太过主观，缺少坚实的理论和方针）。因此，在依赖理论的研究上取得的任何成就都是值得称道的。

## 习题

13.1　本章讨论的关系变量 CTX 和 SPJ（参见图 13-2 和图 13-4）各自满足一个特定的 MVD 和一个特定的 JD，它们并没有被各自的关系变量中的候选码所蕴涵。用第 9 章中的 **Tutorial D** 语法来表示这个 MVD 和 JD。给出微积分和代数两种形式。

13.2　假定 $C$ 是一个俱乐部，关系变量 $R$ $\{A, B\}$ 满足，当且仅当 $a$ 和 $b$ 都是 $C$ 的一员时，元组（$a$，$b$）

---

⊖ Fagin 最初在参考文献［13.15］中确实把 5NF 称为投影 - 连接范式，因为该范式是关于投影和连接运算符的。

才出现在 $R$ 中。那么 $R$ 满足什么样的 FD、MVD 和 JD？$R$ 属于第几范式？

13.3　有一个包含推销员、推销地和产品的数据库。每一个推销员负责在一个或多个地区推销；每一个地区都有一个或多个推销员。同样，每一个推销员负责推销一个或多个产品，每一个产品可以有一个或多个推销员。每一个产品都在每一个地区售卖；然而，没有两个推销员在同一地区销售同一产品。每一个推销员在每一个地区卖他负责卖的同一组产品。对这些数据设计一组合适的关系变量。

13.4　在第 12 章 12.5 节中给出了一个将任意的关系变量 $R$ 无损分解为一组 BCNF 关系变量的算法。修改这个算法，使其能产生 4NF 的关系变量。

13.5　（对习题 13.3 的修改）有一个包含推销员、推销地和产品的数据库。每一个推销员负责在一个或多个地区推销；每一个地区都有一个或多个推销员。同样，每一个推销员负责推销一个或多个产品，每一个产品可以有一个或多个推销员。最后，每一个产品都在一个或多个地区售卖，每一个地区都有一个或多个产品在此售卖。而且，如果推销员 $R$ 负责 $A$ 地区，产品 $P$ 在 $A$ 地区售卖，并且推销员 $R$ 负责卖产品 $P$，那么 $R$ 在 $A$ 地区卖 $P$ 产品。对这些数据设计一组合适的关系变量。

13.6　假定用下面两个关系变量 SX 和 SY（见 13.6 节中的图 13-8）表达供应商：

```
SX { S#, SNAME, STATUS }
SY { S#, SNAME, CITY }
```

这一设计符合本章及前一章描述的规范化指导方针吗？证明你的回答。

## 参考文献

[13.1]　A. V. Aho, C. Beeri, and J. D. Ullman: "The Theory of Joins in Relational Databases," *ACM TODS 4*, No. 3 (September 1979).

这篇论文指出，关系变量在与任何两个它的投影的连接不相同的情况下也可以存在，但要与它的三个或多个投影的连接相等。这篇论文的主要目的是提供一种算法，现在一般称 chase，用来决定一个给定的连接依赖是否是一组给定的函数依赖的合理结果（参见 [13.18] 中的一个相关的例子）。这个问题是与如下的问题等价的，即在给定的一组函数依赖下，判定一给定分解是否是无损的。这篇论文还讨论了将这个算法扩展，来处理给定的依赖是多值依赖而不是函数依赖的问题。

[13.2]　Catriel Beeri, Ronald Fagin, and John H. Howard: "A Complete Axiomatization for Functional and Multi-Valued Dependencies," Proc. 1977 ACM SIGMOD Int. Conf. on Management of Data, Toronto, Canada (August 1977).

Armstrong [11.2] 的扩展是在其中包括了多值依赖和函数依赖。特别地，它给出了下面一组正确而完整的多值依赖的参考规则：

1）互补律（Complementation）：如果 $A$、$B$ 和 $C$ 中包含了关系变量的所有属性，并且 $A$ 是 $B \cap C$ 的父集，那么 $A \twoheadrightarrow B$，当且仅当 $A \twoheadrightarrow C$。

2）自反律（Reflexivity）：如果 $B$ 是 $A$ 的子集，则 $A \twoheadrightarrow B$。

3）增广律（Augmentation）：如果 $A \twoheadrightarrow B$，并且 $C$ 是 $D$ 的一个子集，那么 $AD \twoheadrightarrow BC$。

4）传递律（Transitivity）：如果 $A \twoheadrightarrow B$，并且 $B \twoheadrightarrow C$，那么，$A \twoheadrightarrow C - B$。

下面的规则是从上面的规则中推导出来的：

5）伪传递性（Pseudotransitivity）：如果 $A \twoheadrightarrow B$，并且 $BC \twoheadrightarrow D$，那么 $AC \twoheadrightarrow D - BC$。

6）并（Union）：如果 $A \twoheadrightarrow B$，并且 $A \twoheadrightarrow C$，那么 $A \twoheadrightarrow BC$。

7）分解（Decomposition）：如果 $A \twoheadrightarrow BC$，那么 $A \twoheadrightarrow B \cap C$，$A \twoheadrightarrow B - C$，并且 $A \twoheadrightarrow C - B$。

该论文随后给出了如下涉及 MVD 和 FD 混合的两条规则：

8）复制（Replication）：如果 $A \rightarrow B$，那么 $A \twoheadrightarrow B$。

9）合并（Coalescence）：如果 $A \twoheadrightarrow B$，并且 $C \rightarrow D$，$D$ 是 $B$ 的一个子集，$B \cap C$ 为空，那么 $A \rightarrow D$。

Armstrong 的规则 [11.2] 再加上上面的规则 1～4 和 8～9 就形成了一个正确而又完整的函数依赖和多值依赖的推理规则。

这篇论文还推导出一个更有用的相关函数依赖和多值依赖规则：

10）如果 $A \twoheadrightarrow B$ 并且 $AB \twoheadrightarrow C$，则 $A \rightarrow C - B$。

[13.3] Volkert Brosda and Gottfried Vossen: "Update and Retrieval Through a Universal Schema Interface," *ACM TODS 13*, No. 4 (December 1988).

早期提出"泛关系"接口的尝试（见参考文献［13.20］），只处理了检索操作。这篇论文还提出处理更新操作的方法。

[13.4] C. Robert Carlson and Robert S. Kaplan: "A Generalized Access Path Model and Its Application to a Relational Data Base System," Proc. 1976 ACM SIGMOD Int. Conf. on Management of Data, Washington, D. C. (June 1976).

参见［13.20］的注释。

[13.5] C. J. Date: "Will the Real Fourth Normal Form Please Stand Up?" in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

引自这篇论文的摘要："在数据库设计中，存在好几种对第四范式（4NF）截然不同的解释。本文的目的就是试图澄清这些问题。"也许应该附加说明在本章中称为4NF的概念是仅指真正的4NF……不允许任何替代品！

[13.6] C. J. Date: "The Normal Is So···Interesting" (in tow parts), *DBP&D 10*, Nos. 11 - 12 (November-December 1997).

在13.5节中讨论的逆规范化问题取自这篇论文。下面是另外几个观点：

- 甚至在只读数据库中，标明一个完整性约束也是必要的，因为它们定义了数据的含义，并且（正如在13.4节中所注明的）它们的不可逆规范性又提供了一个标明特定重要约束的简单方法。如果数据库不是只读的，那么不可逆规范性同样提供了一种满足这些约束的简单的方法。
- 逆规范化蕴含了冗余的增加——但是（与普遍的观点相反），增加冗余并不一定蕴涵逆规范化！许多作者都陷入了这个圈套，有一些作者还在重蹈覆辙。
- 作为一个普遍的规则，逆规范化（即在逻辑层的逆规范化）应该作为"仅当所有其他的方法失败"［4.17］时的一种性能策略（performance tactic）来使用。

[13.7] C. J. Date: "The Final Normal Form!" (in two parts), *DBP&D 11*, Nos. 1 - 2 (January-February 1998).

一篇关于连接依赖和5NF的教程。标题即为论述的主题（见第23章）。

[13.8] C. J. Date: "What's Normal, Anyway?" *DBP &D 11*, No. 3 (March 1998).

对某些"病理学的"的规范化例子的综述。

[13.9] C. J. Date: "Normalization Is No Panacea," *DBP &D 11*, No. 4 (April 1998).

本文是有关规范化理论不能解决的一些数据库设计问题的综述。这篇论文并不是一篇攻击性的文章。

[13.10] C. J. Date: "Principles of Normalization," *http://www.BRCommunity.com* (February 2003); *http://www.dbdebunk.com* (March 2003).

本文是一篇简短的指南。为便于参考，总结出以下几点原则：

1）一个非5NF关系变量可以被分解为一系列5NF的投影。
2）连接这些投影可以重新得到原始的关系变量。
3）分解过程应该保持函数依赖。
4）重构过程用到了所有的投影。
5）（此点没有前四点严格）当所有关系变量均属于5NF时，停止规范化过程。

[13.11] C. J. Date and Ronald Fagin: "Simple conditions for Guaranteeing Higher Normal Forms in Relational Databases," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass: Addison-Wesley (1992). Also published in *ACM TODS 17*, No. 3 (September 1992).

如果a）一个关系变量*R*属于3NF并且b）*R*的所有的候选码都是简单的，那么*R*自动地就属于5NF。换句话说，在这个关系变量中没有必要担心在本章中讨论的相对复杂的问题——多值依赖、连接依赖、4NF和5NF。注意：这篇论文还证明了另一个结论，即如果（a）*R*是属于BCNF范式，并且（b）它至少有一个候选码是简单的，那么*R*自动属于4NF，但并不一定属于5NF。

[13.12] C. J. Date and David McGoveran: "A New Database Design Principle," in C. J. Date, *Relational Database Writings 1991 - 1994*. Reading, Mass: Addison-Wesley (1995).

[13.13] C. Delobel and D. S. Parker: "Functional and Multi-Valued Dependencies in a Relational Database and the Theory of Boolean Switching Functions," Tech. Report No. 142, Dept. Maths. Appl. et Informatique, Univ. de Grenoble, France (Novermber 1978).

将 [11.5] 的结果扩展到多值依赖和函数依赖。

[13.14] Ronald Fagin: "Multi-Valued Dependencies and a New Normal Form for Relational Databases," *ACM TODS 2*, No. 3 (September 1977).

MVD 和 4NF 概念的出处。注意：这篇论文还讨论了**嵌入**的多值依赖。假定将 13.2 节的关系变量 CTX 扩展到包含附加的属性 DAYS，这个属性代表在指定的课程中一个指定的教师在一本指定教材上所花的天数。将这个关系变量称为 CTXD，下面是一个实例：

CTXD

| COURSE | TEACHER | TEXT | DAYS |
|---|---|---|---|
| Physics | Prof. Green | *Basic Mechanics* | 5 |
| Physics | Prof. Green | *Principles of Optics* | 5 |
| Physics | Prof. Brown | *Basic Mechanics* | 6 |
| Physics | Prof. Brown | *Principles of Optics* | 4 |
| Math | Prof. Green | *Basic Mechanics* | 3 |
| Math | Prof. Green | *Vector Analysis* | 3 |
| Math | Prof. Green | *Trigonometry* | 4 |

{COURSE, TEACHER, TEXT} 的组合是一个候选码，并且存在函数依赖：

```
{ COURSE, TEACHER, TEXT } → DAYS
```

可以观察到这个关系变量是属于第四范式的；它并不包括任何不属于函数依赖的多值依赖。然而，它确实包括两个嵌入的多值依赖（TEACHER 对于 COURSE 和 TEXT 对于 COURSE）。当"常规"的多值依赖 A →→ B 在 R 的某个投影中存在时，在关系变量 R 中存在嵌入的 B 对 A 的多值依赖。常规的多值依赖是嵌入的多值依赖的特例，但是并非所有嵌入的多值依赖都是常规的多值依赖。

正如这个例子所示，嵌入的多值依赖与常规的多值依赖一样蕴涵着冗余；然而，这个冗余（通常）不能通过投影消除。上面所示的关系变量根本不能被无损分解为它的投影（事实上，它既属于第五范式也属于第四范式），因为 DAYS 依赖于所有的三个属性 COURSE、TEACHER 和 TEXT，并且，如果不是其他三个都出现，它也不能在关系变量中出现。因此这两个嵌入的多值依赖可以标明为关系变量的附加的显式约束。详细的情况作为练习。

[13.15] Ronald Fagin: "Normal Forms and Relational Database Operators." Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (May/June 1979).

这篇论文介绍了投影 - 连接范式（PJ/NF，或 5NF）的概念，而且，其中还不止这些。可以把它看作是"经典的"规范化理论的定义性说明——即基于将投影作为分解的操作符而将自然连接作为相应的合并的操作符的无损分解理论。

[13.16] Ronald Fagin: "A Normal Form for Relational Databases That Is Based on Domains and Keys." *ACM TODS 6*, No. 3 (September 1981).

[13.17] Ronald Fagin: "Acyclic Database Schemes (of Various Degrees): A Painless Introduction," IBM Research Report RJ 3800 (April 1983). Republished in G. Ausiello and M. Protasi (eds.), *Proc. CAAP83 8th Colloquium on Trees in Algebra and Programming* (Springer-Verlag *Lecture Notes in Computer Science 159*). New York, N. Y.: Springer - Verlag (1983).

本章的 13.3 节中介绍了满足一个特定的有环约束（cyclic constraint）的特定的三重关系变量 SPJ 如何被无损分解为它的三个二元投影。所得出的结果数据库的结构被称为是**有环的**，因为这三个关系变量中的每一个都和其他两个有一个相同的属性（如果结果被描述为一个超图，在这个超图中边代表单独的关系变量，相应的两个边相交汇的节点是这两个边的共同的属性，那么应用术语"环"的原因就是很清楚了）。与之相反，实践中的大多数结构都是无环的。无环（acyclic）结构有很多有环结构所不具有的属性。在这篇论文中，Fagin 提供并解释了一系列这样的属性。

无环性的一种很有用的方法如下：正如规范化理论可以帮助决定何时一个单独的关系变量可以用某种方法重构一样，无环性理论可以帮助决定何时一组关系变量可以用某种方法来重构。

[13.18] R. Fagin and M. Y. Vardi: "The Theory of Data Dependencies—A Survey," IBM Research Report RJ4321

(June 1984). Republished in *Mathematics of Information Processing: Proc. Symposia in Applied Mathematics 34*, American Mathematical Society (1986).

这篇论文介绍了依赖理论在20世纪80年代中期的简要发展过程（注意："依赖"在这里并不只代表函数依赖）。特别是，这篇论文总结了在这一领域的三个特定方面的主要成就，并且提供了一系列经过仔细选择的相关参考资料。这三个方面是：(1) 蕴涵 (implication) 问题；(2) "泛关系"模型；(3) 无环模式。蕴涵问题决定的是，给定一组依赖 $D$ 和某个特定的依赖 $d$，$d$ 是否是 $D$ 的一个逻辑结果。泛关系模型和无环模式在文献 [13.20] 和 [13.17] 的介绍中分别简单地进行了讨论。

[13.19] Ronald Fagin, Alberto O. Mendelzon, and Jeffrey D. Ullman: "A Simplified Universal Relation Assumption and Its Properties," *ACM TODS 7*, No. 3 (September 1982).

论文中假设现实世界经常可以通过"泛关系"或泛关系变量来表示 [13.20]，它满足一个连接依赖和一组函数依赖，并且探索了这种假设的一些结果。

[13.20] W. Kent: "Consequences of Assuming a Universal Relation." *ACM TODS 6*, No 4 (December 1981).

泛关系的概念用几种不同的方法证明了它自身。首先，在前面两章中描述的规范化理论都假设定义一初始的泛关系（或者更正确地说，是一个泛关系变量）是可能的，这个关系中包括了与数据库有关的所有属性，并且还描述了关系变量如何被"更小"（较低级）的投影连续代替，直到达到一些"好的"结构。但是这种初始的假定现实或合理吗？这篇论文中表明，无论是在理论上还是实际上都并非如此。参考文献 [13.33] 是对这篇论文的一个回答，而参考文献 [13.21] 是对这个回答的回答。

第二个更重要的泛关系变量概念是作为一个用户界面表示的。它的基本观点十分易懂，并且事实上（从直观的角度）也是十分受欢迎的，那就是：用户可以只根据属性，而不是根据关系变量和这些关系变量中的连接设计它们对数据库的要求。例如：

```
STATUS WHERE COLOR = COLOR ('Red')
```

("取出提供红色零件的供应商的状态")。在这一点上会引来两种不同的解释：

1) 一种可能情况是为了回答这个查询，系统应该自己决定选择哪一个逻辑存取路径（特别是选择哪些连接）。这个方法在参考文献 [13.4] 中有讨论（参考文献 [13.4] 是第一个讨论"泛关系"接口的可能性的文章，但其中并没有用这个术语）。这种方法严格地依赖于属性的适当命名。例如，两个关于供应商数量的属性（分别在关系变量 S 和 SP 中）必须给定同一个名字；相反，供应商的城市和零件的城市的属性（分别在关系变量 S 和 P 中）一定不能是同样的名称。如果违反了这两个规则中的任一个，系统就无法正确处理某些查询。

2) 另一种可能的方法是简单地将所有的的查询看成是根据事先定义的一组连接——由数据库中所有的关系变量的相应连接形成的事先定义的视图——形成的。

毫无疑问，无论哪一种方法都简化了实际中许多查询的表达，而且这样的方法对前端自然语言查询的支持是十分重要的。很明显，该方法一般情况下必须要求系统能够显式指定（逻辑）存取路径。考虑查询：

```
STATUS WHERE COLOR ≠ COLOR ('Red')
```

这个查询的意思是"取出提供非红色的零件的供应商的状态"还是"取出不提供红色零件的供应商的状态"？不论是哪个，都必须有表示另外的一个查询的方法（关于这个问题，上面的第一个例子也是可以产生歧义的："取出只提供红色零件的供应商的状态"）。这里还有第三个例子："取出在同一个城市的一对供应商"。显然一个显式的连接是必要的（因为大致说来，这个问题包括一个关系变量 S 和它本身的连接）。

[13.21] William Kent: "The Universal Relation Revisited," *ACM TODS 8*, NO. 4 (December 1983).

[13.22] Henry F. Korth *et al.*: "System/U: A Database System Based on the Universal Relation Assumption," *ACM TODS 9*, No. 3 (September 1984).

这篇文章描述了理论、DDL、DML 和一个在斯坦福大学开发的实验性的"泛关系"系统的实现。

[13.23] David Maler and Jeffrey D. Ullman: "Fragments of Relations," Proc. 1983 SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1983).

[13.24] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi: "On the Foundations of the Universal Relation

Model," *ACM TODS 9*, No. 2 (June 1984). An earlier version of this paper, under the title "The Revenge of the JD," appeared in Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Ga. (March 1983).

[13.25] David Maier and Jeffrey D. Ullman: "Maximal Objects and the Semantics of Universal Relation Databases," *ACM TODS 8*, No. 1 (March 1983).

最大对象 maximal objects 表示了一个在"泛关系"系统中当基本结构不是无环的情况下出现的歧义问题的解决方法（见参考文献 [13.17]）。一个最大对象对应一个无环结构的全属性的一个预先声明的子集。这样的对象用来指导查询的翻译，否则查询的含义就会出现歧义。

[13.26] J.-M. Nicolas: "Mutual Dependencies and Some Results on Undecomposable Relations," Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, Federal German Republic (September 1978).

本文介绍了"相互依赖"（mutual dependency）的概念。相互依赖实际上是普通连接依赖的一个特例（即它是一个连接依赖，但并不是一个多值依赖或函数依赖），它碰巧包含三个投影（与在 13.3 节中给出的"3D 约束"的例子相似）。这与第 12 章中讨论的共有依赖的概念毫无关系。

[13.27] Sylvia L. Osborn: "Towards a Universal Relation Interface," Proc. 5th Int. Conf. on Very Large Data Bases, Rio de Janeiro, Brazil (October 1979).

这篇论文提出了一个假定，如果对一个给定的查询可以提供候选答案的"泛关系"系统中有两个或多个连接的结果，那么好的回答是所有这些候选答案的并集。并给出了产生这样的连接结果的算法。

[13.28] D. Stott Parker and Claude Delobel: "Algorithmic Applications for a New Result on Multi-valued Dependencies," Proc. 5th Int. Conf. on Very Large Data Bases, Rio de Janeiro, Brazil (October 1979).

本文将参考文献 [13.13] 的结果应用于不同的问题中。例如测试一个无损分解的过程。

[13.29] Y. Sagiv, C. Delobel, D. S. Parker, and R. Fagin: "An Equivalence Between Relational Database Dependencies and a Subclass of Propositional Logic," *JACM 28*, No. 3 (June 1981).

本文组合了参考文献 [11.8] 和 [13.30] 内容。

[13.30] Y. Sagiv and R. Fagin: "An Equivalence Between Relational Database Dependencies and a Subclass of Propositional Logic," IBM Research Report RJ2500 (March 1979).

本文将参考文献 [11.8] 的结果扩展到既包括多值依赖也包括函数依赖。

[13.31] E. Sciore: "A Complete Axiomatization of Full Join Dependencies," *JACM 29*, No. 2 (April 1982).

论文将参考文献 [13.2] 的工作扩展到既包括连接依赖也包括多值依赖和函数依赖。

[13.32] J. M. Smith: "A Normal Form for Abstract Syntax," Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, Federal German Republic (September 1978).

[13.33] Jeffrey D. Ullman: "On Kent's Consequences of Assuming a Universal Relation," *ACM TODS 8*, No. 4 (December 1983).

[13.34] Jeffrey D. Ullman: "The U. R. Strikes Back," Proc. 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, Calif. (March 1982).

# 第14章 语义建模

## 14.1 引言

从20世纪70年代末期开始，语义建模就成为一个研究课题。这项研究的动力，即研究者试图解决的问题是，典型的数据库系统对数据在数据库中的含义的理解是非常有限的：它们通常可以“理解”某些简单的数据值，以及这些值的某些简单约束。但是对其他方面的理解却很少（所有复杂的解释都要用户自己去完成）。显然，系统理解得越多越好，[⊖] 这样它们就会对用户的交互行为反应更智能化，并且会支持更复杂的用户界面。例如：SQL应该能明白零件的重量和发货的数量在含义上的区别。虽然这两个都是数字值，但在类型上是有差别的——即在语义上是不同的。因此，在这种情况下，如果对零件和发货量按照重量和数量相匹配的条件进行连接，那么它即使没有被立即拒绝也至少应该受到质疑。

当然，这个具体的例子与域的概念非常有关系。这个例子显示了现在的数据模型并非完全与语义特征无关。例如，初始定义的域、候选码和外部码都是关系模型的语义特征。换句话说，这些年来发展的注重语义问题的“扩展”数据模型只是稍微地比以前的数据模型增加了一些语义而已（解释参见［14.7］中Codd的文章）。理解数据含义是一个永远不会停止的任务，这个领域也会随着人们对其理解的加深而不断发展。语义模型这个概念经常被用于代表一个或多个“扩展的”模型，并非由于它易于代表这种模型，而是因为它有助于模型理解所涉及的问题的所有语义。另一方面，**“语义建模”**是对试图表示语义的所有行为的一个恰当称呼。在这一章中，我们首先对这些行为中潜在的一些思想进行简单的介绍；然后深入地探讨一种具体的方法：即实体/联系方法（实际中最常用的方法）。

语义建模有许多称呼，包括数据建模、实体/联系建模、实体建模和对象建模。之所以称之为语义建模有以下原因：

- 不用“数据建模”这个概念是因为（a）“数据建模”这个概念与前面所提到的用结构、完整性和操作等组成的正式系统的“数据建模”概念相抵触，（b）数据建模这个概念有可能会导致一种普遍的错误思想，即一个数据模型只包括数据结构。*注意*：这与在第1章1.3节中所提到的数据模型概念有两个不同含义的说法是相关的。前面的概念是通常的数据模型（在这个意义上关系模型是一个数据模型）。而第二种则是一些具体的企业的固定的数据模型。在本书中并不包括第二种含义，而在其他书中是可能包括的。
- 不用“实体/联系建模”的原因是，它只是表示这种问题的一种具体的解决方法，而在实践中许多其他的方法也是可能的。但是“实体/联系建模”这个概念已经很常见并且应用得非常广泛。
- “实体建模”这个概念并不是不好，但是它看起来比“语义建模”更具体一些，因此可能会有某种不适当的暗示意义。
- 对于“对象建模”这个说法的问题是：“对象”在这里是“实体”的一个同义词，然而在其他上下文中则可能会是另一种含义（其他的数据库上下文环境，这方面问题可以参见第25章）。实际上，看来这个现象（有两个不同的含义的现象）很可能引起在本书其他部分提到的**第一个根本性错误**（the first Great Blunder）。请参见第26章对这个问题的详细讨论。

---

[⊖] 不用说，在第8章讨论的支持关系变元和数据库谓词的系统是能“多理解一点语义的”。也就是说，这种谓词支持是语义建模正确而且恰当的基础。然而可悲的是，绝大多数的语义建模方案都不基于任何这样的基础，而是特别到某种程度（参考文献［14.22～14.24］中的提议是一个例外）。然而，由于商业界对商业规则［9.21，9.22］越来越多的重视，这种不好的情况可能会改变。第9章中的谓词在这种意义上讲只是基本的“商业规则”。

下面回到本章的主题。把这一章放入本书的这一部分的原因如下：语义建模的思想在数据库设计中，即使在缺少直接的 DBMS 支持的情况下，也非常有用。因此，数据模型在商业上得到实现之前，最初的关系模型思想是作为最初的数据库设计目标的应用而产生的，因此即使没有商业上的实现而是仅作为设计的补充，这些“扩展的”数据模型思想也是十分有用的。事实上，在撰写本书的时候，语义建模的思想已经在数据库设计领域产生了重要影响，这样说可能并不为过——几个基于某些语义建模方法的设计方法已经被提出来。由于这个原因，本章的重点具体来说是语义模型思想在数据库设计中的应用。

本章的安排如下。在这一节之后，14.2 节解释语义模型中常见的术语。14.3 节介绍最知名的扩展数据模型：Chen 的实体/联系（E/R）模型，14.4 节和 14.5 节介绍这个数据模型在数据设计中的应用（其他的模型作为对“参考文献”中某些文献的注释进行了简单介绍）。最后，14.6 节提供了对 E/R 模型的某些方面的简单分析，14.7 节是小结。

## 14.2 总体方法

下面根据四步来描述语义建模问题的总体方法：

1）首先，要辨别一组语义概念，这些概念在非正式地讨论现实世界时看起来很有用。例如：

- 世界是由**实体**（entity）组成的（虽然不能详细精确地刻画什么是实体，实体的概念至少在讨论现实世界时看起来确实是有用的）。
- 进一步讲，实体可以划分为**实体类型**（entity type）。例如，所有的雇员都是雇员实体类型的**实例**（instance）。这种分类的好处是所有给定**类型**的实体都有共同的**特性**（property）——例如，所有的雇员都有薪水——因此这样可以明显地简化表述。例如，在关系中，共同点就可以被归到关系变量的标题（heading）中。
- 更深入地讲，每一个实体都有一个用来识别自身的具体特性——即每一个实体都有一个**标识**（identity）。
- 再深入一步，任何实体都可以通过**联系**（relationship）来与其他的实体相关。

如此等等。注意：所有的这些术语（例如，实体、实体类型、特性、联系等）都不是精确和非形式化定义的——它们是“现实世界”的概念，而不是形式化的概念。第 1 步不是形式化的步骤，然而，下面的 2～4 步却是形式化的。

2）设计一组相应的符号化的**对象**来代表前面的语义概念（注意：对象这个术语并不代表任何已存在的含义）。例如，扩展的关系模型 RM/T（参见［14.7］）提供了一些特殊的关系类型称为 *E* 关系和 *P* 关系。严格地说，*E* 关系代表实体而 *P* 关系代表特性；如上面所讲的，实体和特性并不是形式化的定义，然而，*E* 关系和 *P* 关系却理所当然是形式化的定义。

3）设计一组正规的、常用的**完整性规则**（或者“元约束”（metaconstraint），这是在第 9 章用到的术语）来搭配这些正规的对象。例如，RM/T 中包括一个称为特性完整性的规则，这个规则要求进入 *P* 关系必须有进入 *E* 关系的许可。（数据库的每一个特性都必须是某些实体的特性，这个要求就反映了这个完整性规则。）

4）最后，设计一组用来操作这些正规对象的正规**操作符**。例如，RM/T 提供了一种属性操作符，这个操作符可以用来将 *E* 关系及其所有相关的 *P* 关系连接在一起，而不必知道有多少个这样的关系和这些关系的名字是什么。因此，可以将任意实体的所有特性都收集在一起。

上面 2～4 步中的对象、规则和操作符一起组成了一个扩展的数据模型——“扩展模型”，这就是说，这些构造确实是一个基本的模型如关系模型的超集；但是，这里上下文中并没有对什么是扩展的和什么是基本的之间的明显区别的解释。请注意，规则和操作符与对象一样，都是模型的一部分（就如它们在关系模型中一样）。另一方面，从数据库设计方面看，较为恰当地说，对象和规则比操作符更重要[⊖]；因此，本章中其余部分的重点就是对象和规则而不是操作符，偶

⊖ 除了需要用操作符公式化地表示规则。

尔可能会就操作符的方面来进行一些讨论。

回到步骤 1：这一步骤尝试区分一组语义概念，这些概念在谈论现实世界时是有用的。有几个这样的概念——实体、特性、关系和子类型——已经在图 14-1 中使用非正式的定义和示例显示出来了。这些示例是有意选取的，它们显示了在现实世界中同样的对象可能会被一些人当作实体，而被另一些人当作特性，还会被另一些人当作联系（顺便说一下，这个观点说明了为什么不可能给实体的术语下一个准确的定义）。支持灵活的解释是语义建模的目标，一个不可能被完全达到的目标。

| 概　念 | 非正式定义 | 举例 |
| --- | --- | --- |
| ENTITY（实体） | 可相互区别的对象 | 供应商，零件，发货<br>雇员，部门<br>人<br>乐曲，协奏曲<br>乐队，指挥<br>购货订单，订货明细 |
| PROPERTY（特性） | 表示实体的一项信息 | 供应商号码<br>发货数量<br>雇员所在部门<br>人的身高<br>协奏曲类型<br>订购日期 |
| RELATIONSHIP（联系） | 充当两个或多个实体间联系的实体 | 发货（供应商 - 零件）<br>分配（雇员 - 部门）<br>录音（乐曲 - 乐队 - 指挥） |
| SUBTYPE（子类） | 如果每一个 *Y* 必是一个 *X*，则实体类型 *Y* 是实体类型 *X* 的子类 | 雇员是人的子类<br>协奏曲是乐曲的子类 |

图 14-1　一些有用的语义概念

顺便提一下，注意在下列术语之间很可能会出现冲突：（a）语义层用到的术语，例如在图 14-1 中所列出的术语；（b）在某些潜在形式（如关系模型）中用到的术语。例如，许多语义建模方案中用属性一词来替代特性，但并不是说这个属性与关系层的属性是同样的概念，也不是意味着它是关系层属性的映射。举另一个例子来说，用在 E/R 模型中的实体类型概念并不与在第 5 章中讲的类型概念相同。（很重要！）更具体地说，这样的实体类型在关系设计中很可能映射到关系变量中，所以它们当然并不与相关的属性类型（域）相一致。但是它们也并不与关系类型一致，因为：

1）一些基本关系类型有可能在语义层与联系类型而不是实体类型相符合。

2）（大致上说）来自关系的类型可能并不与任何语义层的任何类型一致。

层次之间的混乱——特别是在名词冲突中出现的混乱——在过去已经导致了巨大的错误，并且到现在还在出现着错误（参见第 26 章 26.2 节）。

总结：在第 1 章中指出，联系最好被当作实体看待，在本书中一般情况下我们都这样使用。在第 3 章中指出，关系模型的一个优点就是用同样的方法——即通过关系变量来代表所有的实体，包括联系。不过，关系的概念在讨论现实世界时确实在直观上看起来是有用的；而且，在 14.3 ~ 14.5 节中所讨论的数据库设计的方法确实对实体和联系之间的区别依赖很大。因此，引入联系这个术语是为了下面几节的讨论。然而，在第 14.6 节中还会讨论这个问题。

## 14.3　E/R 模型

在 14.1 节中简要地介绍了最有名的语义建模方法之一——当然也是应用最广泛的一种——称为**实体/联系**（E/R）方法，它基于 Chen 在 1976 年提出的“实体/联系模型”（参见［14.6］），并

且在此之后 Chen 和许多其他的人对此方法在很多方面加以修改（参见［14.18］和［14.45～14.47］）。本章的大部分内容都是针对 E/R 方法的讨论（然而，应该强调 E/R 模型远远不是唯一的“扩展”模型，还有许多其他的扩展模型。见参考文献［14.6］、［14.18］、［14.30］、［14.37］特别是［14.24］中具体介绍了几种其他模型，在［14.27］、［14.36］中给出了有关这个领域的综述）。

E/R 模型包括在图 14-1 中列出的所有类似的语义对象。下面将依次进行介绍。然而，首先应该说明，在参考文献［14.6］中并不是只介绍了 E/R 模型本身，还介绍了相应的**图表技术**（“E/R 图”）。在下一节将详细讨论 E/R 图，在这里只给出一个基于参考文献［14.6］的图表的例子，如图 14-2 所示。你将发现它对研究本章所讨论的例子很有帮助。这个例子提供了简单的制造企业的数据（这是对第 1 章图 1-6 中讨论的“KnowWare 公司”的 E/R 图的一个扩充版本）。

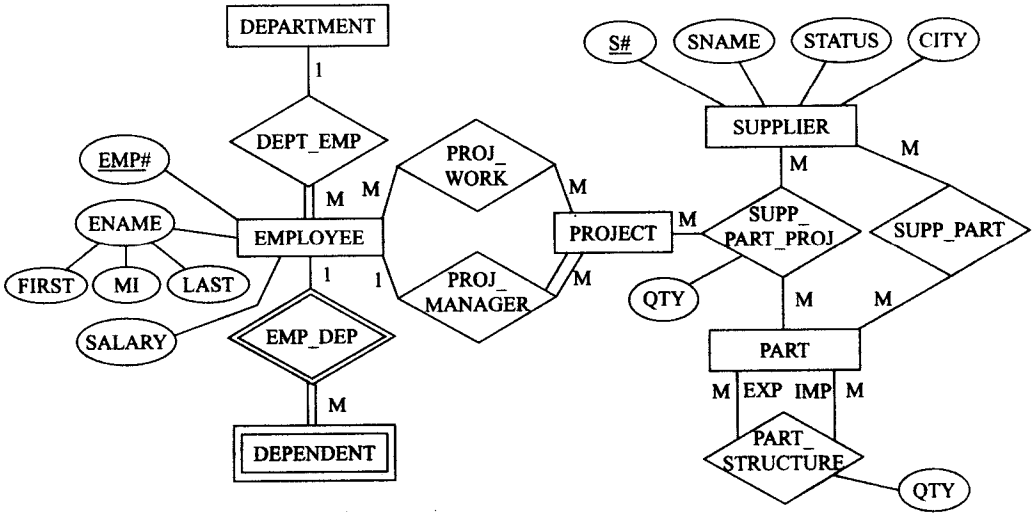



图 14-2 实体/联系图（部分示例）

注意：在下面的小节中所讨论的大部分内容对每一个了解关系模型的人来说都相当熟悉。然而，在术语上有某些不同，下面将会看到。

**1. 实体**

参考文献［14.6］用“能够被清楚辨识的事物”来定义**实体**。并将实体划分为**常规实体**和**弱实体**。一个弱实体是存在－依赖（existence-dependent）于一些其他实体的实体。就是说，如果其他实体不存在，则该实体也不存在。例如，根据图 14-2，一个雇员的依赖者（dependent）就是弱实体——当相关的雇员不存在时，它们也不能存在（只考虑数据库）。特别是如果一个给定的雇员被删掉了，则雇员的所有依赖者都将被删除。与之相反，一个常规实体是非弱实体；例如，一个雇员就是一个常规实体。注意：有些作者用“强实体”而不是用“常规实体”来表示。

**2. 特性**

实体和联系都具有**特性**。一个给定类型的所有的实体或联系都有某种共同的特性：例如，所有雇员有一个雇员编号、雇员姓名、工资等特性（注意：在此有意不将“部门号”作为雇员的特性提出。参见下一小节对联系的讨论）。每一个特性值都是从一个相应的**特性集合**中取出的（即在关系术语中的域）。而且，特性还可以是：

- **简单**的或**复合**的：例如，复合特性“雇员姓名”可能由简单的特性“first name”、“middle name”和“last name”组成。
- **码**（即在某些上下文中是唯一的）：例如，一个依赖者的名字对于一个给定的雇员来说可能是唯一的。

- **单值的**或**多值的**（换句话说，允许重复的组[⊖]）：所有在图 14-2 中显示的特性都是单值的，但是如果出现这种情况：例如，一个供应商有几个不同的住地，那么“供应商所在城市”将会是一个多值特性。
- **空缺的**（即“不知道的”或“不能适用的”）：这种概念并没有在图 14-2 中出现，参见第 19 章的详细讨论。
- **基本的**或**导出的**：例如，对一个给定零件的“总数量”是由该零件的每一笔发货量的总和导出的。图 14-2 中对此概念也没有示例。

注意：一些作者在 E/R 模型中用“attribute”而不是“property”。

**3. 联系**

参考文献［14.6］定义：**联系**是“实体之间的一个关联”。例如，在部门和雇员之间有一个联系称为 DEPT_EMP，这个联系代表特定的部门雇佣特定的雇员。与实体相同（参见第 1 章），在原则上区别联系类型和联系实例是很必要的，但是通常在非正式的讨论中忽略了这样的过程，在下面的讨论中就是这样。

在一个给定联系中的实体称为这个联系中的**参与者**（participant），在一个联系中的参与者的数量称为联系的**度**（degree）。（在此要注意：这个术语与在关系模型中的含义有所不同。）

*R* 是一个联系类型，*E* 是其中的参与者。如果每一个参与者 *E* 的实例都至少在 *R* 的一个实例中出现。那么 *R* 中的参与者 *E* 称为**全部的**（total），否则它被称为**部分的**（partial）。例如，如果每一个雇员都必须属于一个部门，那么在关系 DEPT_EMP 中的参与者雇员是一个全部的；如果对于一个给定的部门可以根本就没有雇员，那么在联系 DEPT_EMP 中部门的参与者就是部分的。

一个 E/R 联系可以是**一对一**、**一对多**（也可以称为**多对一**）或**多对多**（为了简单，假定所有的联系都是二元的，即度数为 2；将这个概念或术语扩展到更高度数的联系当然是很容易的）。现在，如果对关系模型非常熟悉的话，可以认为多对多的情况才是唯一真正的联系，因为这种情况是唯一要求用独立的关系变量来表示——一对一和一对多联系通常可以被一个参与者关系变量中的一个外码来表示。然而，如果一对一和一对多的情况随着时间的推移可以产生或变成多对多的情况，那么就可以将一对一和一对多的情况当作多对多的情况来看待。仅当没有这样的可能性时才能安全地将它们分别对待。当然，有时是不存在这种可能的；例如，一个圆只有一个圆心总是正确的。

**4. 实体子类型和超类型**

注意：在这一小节中讨论的内容并不包含在最初的 E/R 模型中（见参考文献［14.6］），但是出现于后来的模型中。举例来说，见参考文献［14.46］。

任何给定的实体至少属于一个实体类型，但一个实体可以同时有几个类型。例如，如果一些雇员是程序员（所有的程序员都是雇员），实体类型 PROGRAMMER（程序员）是实体类型 EMPLOYEE（雇员）的**子类型**（或者说，实体类型 EMPLOYEE 是 PROGRAMMER 的一个**超类型**）。所有的雇员特性都自动适用于程序员，但反之则不然。（例如，程序员可能具有“程序语言的技能”的特性，但一般来讲雇员没有此特性。）同样，程序员自动具有雇员所具有的所有特性，而反之则不对。（例如，程序员可能属于一些专业的计算机团体，而一般的雇员则并不如此。）适用于父类的特性和联系可以为子类所**继承**。

还要注意一些程序员可能是应用程序员而另一些可能是系统程序员；因此称实体类型 APPLICATION_PROGRAMMER 和 SYSTEM_PROGRAMMER 都是 PROGRAMMER 超类型的子类（如此等等）。换句话说，一个实体子类还是一个实体类型，因此可以有它自己的子类。一个给定的实体类型和它的子类、子类的子类，等等……一起组成了一个**实体类型层次**（参见图 14-3）。

以下几点需要指出：

1）在第 20 章会深入地讨论类型层次和类型继承，然而值得注意的是：在第 20 章使用的类

---

⊖ 如果你不熟悉这一术语，请参考 6.4 节的子节“关系取值的属性”。

型这个术语与第 5 章中的类型的含义相同，而与本章中讨论的“实体类型”的意义不同。

2）如果读者对 IMS（或某些其他支持层次数据结构的数据库系统）熟悉，可以发现类型层次与 IMS 形式的类型层次并不相同。例如，在图 14-3 中并没有表明一个 EMPLOYEE 有许多 PROGRAMMER 与之对应（而如果图中所示的是 IMS 的层次结构，则会有这样的对应关系）；相反，对于一个 EMPLOYEE 的实例至多只有一个相应的 PROGRAMMER 与之对应，这表明一个雇员只能处于一个程序员的角色。

这就是对 E/R 模型的主要结构特征的简要描述。现在将注意力转到 E/R 图上来。

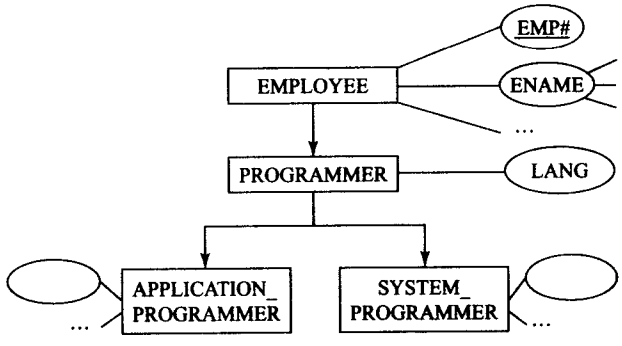



图 14-3　实体类型层次示例

## 14.4　E/R 图

在前面的一节中已经指出，参考文献［14.6］不只介绍了 E/R 模型本身，还介绍了**实体/联系图**的概念（E/R 图）。E/R 图形成了一种以图形方式表示数据库的逻辑结构的技术。因此，它们提供了一种简单易懂、并可以与任意给定的数据库的设计相交流的方式（“一图胜千言”）。的确，E/R 模型作为一种数据库设计方法的普及性更多地归功于 E/R 图的存在而不是其他的原因。下面通过在图 14-2 和图 14-3 中已给的例子来描述创建一个 E/R 图的规则。

注意：和 E/R 模型一样，E/R 图设计技术也是随着时间的发展而发展的。在这里所描述的版本在某些特定的方面与最初在参考文献［14.6］中描述的有所不同。

**1. 实体**

每一个实体类型都以矩形表示，在矩形中包含所表示的实体类型。对于一个弱实体类型，矩形框是双线的。

示例（参见图 14-2）：

- 常规实体：
  DEPARTMENT
  EMPLOYEE
  SUPPLIER
  PART
  PROJECT
- 弱实体：
  DEPENDENT

**2. 特性**

特性用椭圆显示，椭圆内包含特性的名字，并且通过实线与相关的实体相连。如果特性是导出的，则椭圆的边用点线或虚线表示，如果特性是多值的，则椭圆边用双线表示。如果特性是复合的，它的成员特性一样用椭圆来表示，通过实线连接到表示复合特性的椭圆上。码特性用下划线标明。特性对应的值集合不予显示。

示例（参见图 14-2）：

- 对于 EMPLOYEE：
  EMP#（码）
  ENAME（复合的，包含 FIRST、MI 和 LAST）
  SALARY
- 对于 SUPPLIER：
  S#（码）
  SNAME
  STATUS
  CITY
- 对于 SUPP_PART_PROJ：
  QTY
- 对于 PART_STRUCTURE：
  QTY

为了节省空间，所有其他的特性都在图 14-2 中省略了。

**3. 联系**

每一个联系类型都用菱形框显示，其中包含所表示的联系类型名。如果表示一个弱实体和它所依赖的实体之间的联系，则用双线菱形框。每一个联系的参与者都通过实线与其相关的联系连接；每一个这样的线都标注“1”或“M”来表示这个联系是一对一、多对一，等等。如果参与者是全部的，则用双线连线表示。

示例（参见图 14-2）：

- DEPT_EMP（在 DEPARTMENT 和 EMPLOYEE 之间的一对多联系）
- EMP_DEP（在 EMPLOYEE 和一个弱实体 DEPENDENT 之间的一对多联系）
- PROJ_WORK 和 PROJ_MANAGER（在 EMPLOYEE 和 PROJECT 之间的相互独立的多对多和一对多联系）
- SUPP_PART_PROJ（包括 SUPPLIER、PART 和 PROJECT 的多对多对多联系）
- SUPP_PART（在 SUPPLIER 和 PART 之间的多对多联系）
- PART_STRUCTURE（在零件和零件之间的多对多的联系）

观察到在最后一个例子中两条从 PART 到 PART_STRUCTURE 的两条线通过标注着两个不同的**角色**名的标签区别开来（分别为 EXP 和 IMP，即“零件爆炸”（explosion）和“零件闭塞”（implosion））。PART_STRUCTURE 是一个称为**递归联系**（recursive relationship）的示例。

**4. 实体子类型和超类型**

实体类型 *Y* 是实体类型 *X* 的一个子类型。可以画一条从 *X* 的矩形框到 *Y* 的矩形框的箭头线，这条线代表着一种称为“**isa** 联系”的连接（因为每一个 *Y* 都“是一个”（is a）*X*——即所有 *Y* 的集合是所有 *X* 集合的子集）。

示例（参见图 14-3）：

- PROGRAMMER 是 EMPLOYEE 的一个子类
- APPLICATION_PROGRAMMER 和 SYSTEM_PROGRAMMER 都是 PROGRAMMER 的一个子类。

## 14.5　基于 E/R 模型的数据库设计

在某种意义上讲，通过在前一节描述的规则构建 E/R 图就是一个数据库的设计。然而，如果试图将这样的一个设计映射到一个具体形式的 DBMS 上，[○] 就会发现 E/R 图还是在某些方面不精确，并且它在一些细节方面并未加以考虑（特别是完整性约束的细节方面）。为了阐述这一点，可以考虑一下在将图 14-2 的设计映射到关系数据库定义上时都包括了什么内容。

---

○　很多工具可以在这一映射过程中应用。（例如，通过使用 E/R 图生成 SQL CREATE TABLE 语句及类似的语句。）

### 1. 常规实体

重复一下，在图 14-2 中描述的常规实体如下：

```
DEPARTMENT
EMPLOYEE
SUPPLIER
PART
PROJECT
```

每一个常规实体都映射到一个基本关系变量上。因此数据库包括五个基本关系变量：DEPT、EMP、S、P 和 J，分别与这五个实体类型相对应。而且，这五个基本关系变量中的每一个都有一个候选码（通过属性 DEPT#、EMP#、S#、P#和 J#表示）它们都与 E/R 图中确定的“码”相对应。为了定义方便，给每一个关系变量指定一个主码。那么关系变量 DEPT 的定义（例如）以如下方式开始：

```
VAR DEPT BASE RELATION
        { DEPT# ..., ... }
        PRIMARY KEY { DEPT# } ;
```

另外的四个关系变量的定义作为练习留给读者。注意：域或“值的集合”当然也应该被限制。既然在前面已经提到属性值并不包含在 E/R 图中，所以省略了对这一方面的细节的讨论。

### 2. 多对多联系

在示例中多对多（或多对多对多等）联系如下：

```
PROJ_WORK （包括雇员和项目）
SUPP_PART （包括供应商和零件）
SUPP_PART_PROJ （包括供应商、零件和项目）
PART_STRUCTURE （包括零件与零件）
```

每一个相应的联系也映射到一个基本关系变量上。因此再介绍四种基于这四个联系的基本关系变量。下面看一下 SUPP_PART 联系。假定这个联系的关系变量是 SP（最常用的发货量关系变量）。为了辨别这个联系中的参与者，先将注意力从这个关系变量的主码问题转移到外码问题上：

```
VAR SP BASE RELATION SP
     { S# ... , P# ... , ... }
       .....
     FOREIGN KEY { S# } REFERENCES S
     FOREIGN KEY { P# } REFERENCES P ;
```

很清楚，与两个参与者（供应商和零件）相对应，关系变量中必须包括两个外码 {S#和 P#}，并且这些外码必须参照参与者关系变量 S 和 P。此外，一组适当的外码规则组合（即一个 DELETE 规则和一个 UPDATE 规则）必须是这些具体的外码中的某一个。注意：指定的规则只是通过图示来展示（图示并非唯一的展示形式）。要特别注意，规则是不能由 E/R 模型导出或指定的。

```
VAR SP BASE RELATION SP
     { S# ... , P# ... , ... }
       .....
     FOREIGN KEY { S# } REFERENCES S
             ON DELETE RESTRICT
             ON UPDATE CASCADE
     FOREIGN KEY { P# } REFERENCES P
             ON DELETE RESTRICT
             ON UPDATE CASCADE ;
```

对于这个关系变量中的主码又如何处理呢？一个可能方法是采取参与者－标识（identifying）外码的组合（S#和 P#，在 SP 的例子中）。如果（a）对于每一个联系的实例，这个组合都有一个唯一值（有可能出现这种情况，也有可能不出现，通常情况下是有的）；（b）设计者对复合主码无异议。另一种方法是，将一个新的非复合的替代属性，如“发货的数量”，作为主码来使用（见参考文献［14.11］和［14.21］）。由于这个例子的原因，选择第一种方法，给基本关系变量 SP 的定义加上一条语句：

```
PRIMARY KEY { S#, P# }
```

对于 PROJ_WORK、PART_STRUCTURE 和 SUPP_PART_PROJ 的联系的讨论作为练习留给读者。

**3. 多对一联系**

在这个示例中有三个多对一联系：

```
PROJ_MANAGER（从项目到经理）
DEPT_EMP（从雇员到部门）
EMP_DEP（从依赖者到雇员）
```

在这三个联系中，最后一个包含了一个弱实体类型（DEPENDENT），而另两个中只包含了常规实体类型。现在来看一下前两种情况。考虑 DEPT_EMP 的例子。这个例子并不引起任何新的关系变量的引入[⊖]。而事实上只在联系的“多”边上（EMP）的关系变量中引入一个外码，来参照“一”边（DEPT）的关系变量。因此有：

```
VAR EMP BASE RELATION
      { EMP# ..., DEPT# ..., ... }
      PRIMARY KEY { EMP# }
      FOREIGN KEY ( DEPT# ) REFERENCES DEPT
              ON DELETE ...
              ON UPDATE ... ;
```

DELETE 和 UPDATE 规则可能性与外码在多对多联系中代表参与者的可能性相同（一般是这样）。同样，在 E/R 图中不做说明。

注意：对于目前的表示，假定一对一联系（实际中，在任何情况下都不常见）与多对一联系的处理方式相同。参考文献［14.8］中包括了对一对一情况的特殊问题的深入讨论。

**4. 弱实体**

一个弱实体类型与它所依赖的实体类型的联系理所当然地是一个多对一联系，在前面的小节中已经介绍了。然而，这个联系的 DELETE 规则和 UPDATE 规则必须描述如下：

```
ON DELETE CASCADE
ON UPDATE CASCADE
```

这些说明合起来共同获取并反映了必要的存在依赖（existence dependence）。下面是一个示例：

```
VAR DEPENDENT BASE RELATION
  { EMP# ..., ... }
    .....
  FOREIGN KEY ( EMP# ) REFERENCES EMP
          ON DELETE CASCADE
          ON UPDATE CASCADE ;
```

对于这样的一个关系变量来说什么是主码呢？就多对多的情况来说，事实证明可以有几种选择。一种是，如果数据库设计者同意复合的主码，则可以采用外码和 E/R 图中的弱实体“码”的组合。另一种选择是，可以引入新的（非复合）替代属性作为主码（见参考文献［14.11］和［14.21］）。还是由于这个例子的缘故，选择第一种方案，根据基本关系变量 DEPENDENT 的定义增加下面的语句：

```
PRIMARY KEY { EMP#, DEP_NAME }
```

（其中，DEP_NAME 是雇员的依赖者的名字）

**5. 特性**

在 E/R 图中显示的每一个特性都要映射成为一个适当的关系变量的属性——除非特性是多值的，那将为它创建一个新的关系变量（如第 12 章 12.6 节讨论的那样）。很明显，值的集合可映射为很多类型（事实上，值的集合本身就带有类型）。这种映射是非常简单的，这里就不进行

---

⊖ 尽管有可能像 14.3 节提到的，有时有充分的理由把一个多对一关系作为多对多关系处理。参考文献［19.19］的第四部分深入地讨论了这个问题。

进一步的讨论了。注意：开始时确定我们到底需要哪些值是非常困难的。

**6. 实体子类型和超类型**

由于图14-2并没有包括任何子类型和超类型，所以参见图14-3，其中包含了这两种类型。下面看一下实体类型EMPLOYEE和PROGRAMMER。为了简单起见，假定程序员只有一种语言技能（即特性LANG是单值的）。那么[⊖]

- 与往常的方法一样（即已经讨论的方法），超类型EMPLOYEE映射到一个基本关系变量EMP中。
- 子类PROGRAMMER映射到另一个基本关系变量PGMR中，主码与超类型关系变量相同，其他的属性与仅是程序员的雇员的相应属性相同（即在例子中只是LANG）：

```
VAR PGMR BASE RELATION
  { EMP# ..., LANG ... }
  PRIMARY KEY { EMP# } ... ;
```

此外，PGMR的主码也是参照关系变量EMP的外码。因此应将定义扩展为（注意，特别是扩展DELETE和UPDATE规则）：

```
VAR PGMR BASE RELATION
  { EMP# ..., LANG ... }
  PRIMARY KEY { EMP# }
  FOREIGN KEY { EMP# } REFERENCES EMP
          ON DELETE CASCADE
          ON UPDATE CASCADE ;
```

- 还需要一个视图EMP_PGMR，它是超类型关系变量和子类型关系变量的连接：

```
VAR EMP_PGMR VIEW
    EMP JOIN PGMR ;
```

注意这个连接是（零或一）对一的连接。它是通过一个候选码和一个与之相匹配的外码连接的，并且这个外码本身是一个候选码。因此，大致来说，这个视图包括只是程序员的这些雇员的信息。

下面给出设计如下：

- 可以通过基本关系变量EMP了解所有的雇员的特性（例如，为了恢复的方便）。
- 可以通过基本关系变量PGMR了解只属于程序员的特性。
- 可以通过视图EMP_PGMR了解所有属于程序员的特性。
- 可以通过基本关系变量插入不是程序员的雇员。
- 可以通过视图EMP_PGMR插入所有属于程序员的雇员。
- 可以通过基本关系变量EMP或视图EMP_PGMR删除雇员、程序员或其他。
- 可以通过基本关系变量EMP或视图EMP_PGMR更新所有雇员的特性。
- 可以通过基本关系变量PGMR更新所有程序员的特性。
- 可以通过在基本关系变量PGMR或视图EMP_PGMR中插入雇员来使一个非程序员变为程序员。
- 可以通过在基本关系变量中删除程序员的方法来使一个雇员从程序员变为非程序员。

对图14-3中的其他实体类型的讨论作为练习留给读者（APPLICATION_PROGRAMMER和SYSTEM_PROGRAMMER）。

## 14.6 简单分析

在这一节较深入地分析一下E/R模型的某些方面。下面的讨论是从参考文献［14.9］中对

---

⊖ 注意，下面我们要做的不是将EMPLOYEE和PROGRAMMER映射为某种“超表”（supertable）和“子表”（subtable）结构。这里存在一个概念上的难点，或者说是陷阱：只因为在E/R图中实体类型*Y*是实体类型*X*的子类型，并不表明*Y*的关系对象就要是*X*的关系对象的什么“子”类——并且事实上也并不是。更深入的讨论见参考文献［14.13］。

这个问题的详细分析部分中抽取的。另外的分析和注释可以在本章最后一节“参考文献”中许多参考的评注中找到。

**1. E/R 模型是否可以作为关系模型的基础**

通过从一个不同的角度考虑 E/R 方法来进行这个讨论。很明显，当 Codd 第一次提出正式的关系模型的时候，一定是 E/R 方法，或一些与这种思想非常接近的想法，支持着 Codd 的思维的。如在 14.2 节中所讨论的，发展一个“扩展的”模型的总体方法包括四个大的方面，如下：

1）辨别有用的语义概念。

2）设计形式化的对象。

3）设计形式化的完整性规则（“元约束”）。

4）设计形式化的操作。

但是这四步对基本的关系模型的设计也是适用的（并且确实对任何正式的数据模型适用），并不只是应用于“扩展的”模型，如 E/R 模型。换句话说，Codd 为了首先构建（形式化的）基本的关系模型，必须在头脑中具有某些（非形式化的）“有用的语义概念”，并且这些概念必须基本上是这些 E/R 模型，或与之相类似的模型。事实上，Codd 的著作的确表现出了这种观点。在他的第一篇有关关系模型的论文中（参考文献［6.1］的 1969 版本），可以发现如下内容：

“一个给定的实体类型的实体集合可以被看作是一个关系，可以称这种关系为实体类型关系……其他的关系……是在类型之间的……被称为是实体间关系……每一个实体间关系的核心特性［它包括至少两个外码］要么是指相互区别的实体类型，要么是指充当不同角色的一个共同实体类型。”

在这里，Codd 清楚地提出了用来代表“实体”和“联系”的关系。但是（在很大程度上）他的观点是：关系是形式化的对象，而且关系模型是一个形式化的系统。Codd 的主要贡献是他发现了对现实世界的某些方面进行了很好描述的形式化模型。

与前面刚刚讨论的相比，实体/联系模型并不是（或者说至少不是主要的）一个形式化模型。它主要包括一组非形式化的概念，这组概念与前面提到的四步中的步骤 1 相对应（此外，它所进行的形式化的方面看来并不与基本的关系模型相应的方面有什么重大不同——在下面的一个小节中有对这个问题的详细讨论）。毫无疑问，具有“步骤 1”概念的思想将会对进行数据库设计的处理工作很有帮助，但是如果没有形式化的对象和步骤 2、3，数据库设计就不能完成，其他的一些任务没有步骤 4 的形式化的操作也不能完成。

请注意前面的讨论并不是要表明 E/R 模型没有用，相反它是有用的。但这还不够。而且奇怪的是最早发表的非形式化的 E/R 模型的描述是在最早的形式化关系模型的描述发表几年之后，因而存在这样的假定（正如我们已经谈过的），即后者起初是建立在某些类似 E/R 思想的基础上的。

**2. E/R 模型是不是一个数据模型**

由前面的讨论，甚至不能判断 E/R“模型”是不是一个数据模型，至少在本书中所用的这个术语本身的意义上来讲（即作为正式的系统应包括结构、完整性和操作等方面）。当然，“E/R建模”这个术语经常用来表示决定数据库结构（只是数据库的结构）的过程，虽然在第 14.3～14.5 节[⊖] 中也简单地讨论了特定的完整性因素问题（主要是与主码和外码相关的问题）。然而，仔细地阅读参考文献［14.6］会发现，E/R 模型确实是一个数据模型，但在本质上只是基础的关系模型顶上的一个薄层（正如一些人所述，它当然并不是替代关系模型的一个候选模型），下面证明如下：

- 首先，基本的 E/R 数据对象（基本的正规对象，与不正规的对象“实体”、“联系”等相

---

⊖ 这当然是一个主要的弱点，E/R 模型除了几个特殊的情况（公认是重要的情况）之外是完全不能处理完整性约束或“企业规则”的，这里引用一句经典的话：“说明性（declarative）的规则太复杂，在企业模型中很难遵循，必须由分析员/开发人员单独定义。”［14.23］有一种观点认为数据库设计应该是一个限制应用约束的过程（见参考文献［9.21，9.22］和［14.22～14.24］）。

对立）是一个 $n$ 元关系。

- E/R 操作是基本的关系代数的运算符（实际上，参考文献［14.6］在这一点上解释得并不是非常清楚，但提出一组这样的操作显然没有那些关系代数的操作全面；例如，显然在这里没有并集操作和明显的连接操作）。
- 在完整性方面，E/R 模型有一些关系模型没有的功能：E/R 模型包括一组内置的完整性规则，与在本书中所讨论的一些而不是全部的外码相对应。因此，“纯”关系系统需要用户显式地构建特定的外码规则，而 E/R 系统只要求用户标明一个给定的关系变量表示某些联系，这样，一些外码规则就可以很好地理解。

**3. 实体与关系**

本书中已经几次指出，“关系”最好只被当作一种特殊类型的实体。而相反地，区分这两个概念是 E/R 语义建模方案的一个必要条件。作者的观点是任何坚持做这样区分的方法一定是有严重缺陷的，因为（在 14.2 节中提到）同一个对象可以被一些用户看作实体而被另一些用户当作联系。下面看一个关于婚姻的例子：

- 从一个角度看，结婚是两个人之间的联系。（一个查询示例：“谁在 1975 年和伊丽沙白泰勒结婚?”）
- 从另一个角度看，对每一个人来说，婚礼都是实体。（查询示例：“自从 4 月份以来在这个教堂进行了多少次婚礼?”）

如果设计方法强调区分实体和联系，那么（最好的情况是）这两个解释将会被不平等地对待（即“实体”查询和“联系”查询会采用不同的形式）；在最坏的情况下，一种解释方式根本不支持另一种解释方式（即某一类查询将无法进行）。

有关这一点更深层的说明见［14.22］中关于 E/R 方法的介绍：

*最初常见的方法是，在概念模式设计中用属性［特指外码］来表示联系，然后在设计进行中将这些属性转换为联系，使这些属性更好理解。*

但是当一个属性变成一个外码时会发生什么情况呢？即数据库在已经存在了一段时间后发生变化，在这种情况下会发生什么呢？如果因此而作出逻辑结论，那么数据库设计就应该只包括联系而根本不包括属性。（事实上，这种设计确实有一些好处。见参考文献［14.23］的注释。）

**4. 最后一个分析**

除了在本章中描述的 E/R 建模这个特定的语义建模方案之外还有很多其他的方案。然而，这些方案绝大部分都具有一种很强的相似性；特别是它们都可以被刻画为表示特定的外码约束和几个其他的微小方面的简单的图形符号。这样的图形表示法在“大图”（big picture）方式中当然有用，但是它们太简化了，不能应付整个的设计工作[⊖]。特别是如前面提到的，它们一般不能处理通常的完整性约束。例如，如何在一个 E/R 图中表示一个连接依赖呢。

## 14.7 小结

在本章中简单地介绍了**语义建模**这个概念。共包括四个步骤，其中第一步是非正规的，而剩下的三步都是正规的：

1）区别语义概念。

2）设计相应的符号对象。

3）设计相应的完整性规则（“后约束”）。

4）设计相应的操作符。

一些有用的语义概念是**实体**、**特性**、**联系**和**子类型**。*注意*：我们还强调一点：在语义建模层（非正规）和系统层（正规的）之间很可能存在**术语冲突**，这种术语冲突可能引起混乱！

语义建模研究的最终目的是使数据库系统更智能化。一个更直接的目标是为系统地解决**数据**

---

⊖ 遗憾的是，在对 IT 界简单的甚至过于简单的答案是很受欢迎的。就这一情况，我们赞同爱因斯坦说过的，“每件事应该尽可能地让它变得简单，而不是比较简单”。

**库设计**中的问题提供一个基础。我们描述了一种特定的“语义”模型的应用，即 Chen 的**实体/联系模型（E/R 模型）**。与前文所述相联系，需要重申的是：最初的 E/R 论文［14.6］中确实包含两个互相区别的多少有些独立的提议：即提出了 E/R 模型本身以及 E/R 图技术。正如在第 14.4 节中所论述的，E/R 模型的普遍性可以归功于**E/R 图表技术**而并非其他的原因。但是只为了用图没必要采用所有的模型思想；将 E/R 图作为任何设计方法（例如，在参考文献［14.7］的一个基于 RM/T 的方法）的基础都是可能的。有许多讨论将 E/R 模型和一些其他方法的相符性作为数据库设计的基础，在这些讨论中通常没考虑到这一点。

下面将语义建模的思想（特别是 E/R 模型）和在第 12、13 章中讨论的规范化理论作一比较。规范化理论是将大关系变量分解为小的关系变量；它假定初始有数量较少的大关系变量，通过规范化理论将这些大的关系变量的初始输入变成小的关系变量输出——即将大的关系变量映射为小的关系变量（当然，在这里只是非常简略地提一提）。但是规范化理论却没有提到如何先得到的那些大的关系变量。然而，像本章所讨论的自上而下的设计方法确切地揭示了这个问题；它们将现实世界映射到这些大的关系变量中。换句话说，这两个方法（自上而下的方法和规范化方法）互为补充。因此总体设计过程如下：

1）用 E/R 方法（或类似的方法[⊖]）产生“大的”具有常规实体、弱实体等的关系变量。

2）利用进一步规范化的思想将这些“大的”关系变量分解为“小”的关系变量。

然而，从本章讨论的内容上说，语义建模还不像在第 12 章、13 章中讨论的进一步规范理论那样显得严格分明。原因是（正如在本书这一部分的引言中所谈到的）数据库设计是一个非常主观的活动，并不是非常客观的；可以作为处理这些问题的基本原理相对来说是比较少的（当然确实存在这样的一些原理，基本上前两章所讨论的内容就是一些这样的基本原理）。虽然本章的方法在实际问题中确实是有效的，但它们仍然只能被看作经验准则。

最后还有一点要说明。虽然语义建模思想在数据库设计的整个方面还有些主观，但在一些具体的方面（即在**数据字典**中）是非常有用的。数据字典在某些方面可以被看作是“数据库设计者的数据库”；在这个数据库中，数据库设计的行为与其他的操作一起被记录。因此语义建模的研究在数据字典系统的设计中是很有用的，因为它辨别了字典本身需要支持和“理解”的一类对象——例如，实体目录（entity category）（如 E/R 模型中的常规实体和弱实体）、完整性约束（如在 E/R 模型中，联系的全部与部分参与者的表述）、实体超类型和子类型等。

## 习题

14.1　谈谈对“语义建模”的理解。

14.2　试述在 E/R 模型中定义一个“扩展”模型所包括的四个步骤。

14.3　用自己的话解释以下的 E/R 术语

| | | | | |
|---|---|---|---|---|
| 实体 | 继承 | 码属性 | 特性 | 常规实体 |
| 联系 | 父类型和子类型 | 类型层次 | 值的集合 | 弱实体 |

14.4　假定在表示供应商和零件的 E/R 图中表明，在供货关系中参与者零件是“全部的”（也就是，每种零件必须由至少一个供应商供应）。使用（a）**Tutorial D**，（b）SQL，如何表达这一约束？

14.5　举出下面几种情况的示例：

a. 一个参与者是弱实体的多对多联系。

b. 一个参与者是另一个联系的多对多联系。

c. 有一个子类型的多对多联系。

d. 一个有父类型中不存在的弱实体的子类型。

14.6　根据第 9 章中练习 9.7 画一个教育数据库的 E/R 图。

14.7　根据第 12 章中练习 12.3 的公司员工数据库画一个 E/R 图。应用这个图导出一组适当的基本关系变量定义。

---

⊖ 我们首选的方法是：（a）写下描述企业的外部谓词；（b）按照第 9 章描述的，把这些谓词直接映射到内部谓词。

14.8 根据第 12 章中练习 12.4 的数据库画一个 E/R 图。用这个 E/R 图来导出一组适当的基本关系变量定义。

14.9 根据第 13 章的练习 13.3 中的销售数据库画一个 E/R 图。利用这个 E/R 图导出一组适当的关系变量定义。

14.10 根据第 13 章的练习 13.5 中修改后的销售数据库来画一个 E/R 图。利用这个 E/R 图导出一组适当的关系变量定义。

## 参考文献

下面列出的参考文献之所以显得长了一些，是由于在数据库界近年来确实提出了大量的数据库设计方法，这其中有来自工业界的也有来自学术界的。在这个领域很少有一致的意见；本章中所介绍的 E/R 模型当然是应用最广泛的一种方法，但是并不是所有人都接受它（或喜欢它）。事实上，应用最广泛的方法并不一定是最好的方法。其实许多商业产品已远远超出了只提供数据库设计工具的范畴，它们已可以做到去生成整个应用——除了特定的数据库定义（模式）外，还有前端屏幕设计、应用逻辑、触发过程，等等。（关于这一点，见 Ross 的关于企业规则的著作 [9.21, 9.22]，及参考文献 [9.15]。）其他与本章内容相关的文献包括 ISO 有关概念模式的报告 [2.3]；Kent 的《Data and Reality》[2.4]。

[14.1] J. R. Abrial: "Data Semantics," in J. W. Klimbie and K. L. Koffeman (eds.), *Data Base Management*. Amsterdam, Netherlands: North-Holland / New York, N. Y.: Elsevier Science (1974).

在语义建模领域的一个早期的提议。下面引用的话很好地描述了这篇论文的大体格调："对读者的提示：如果要找语义这个术语的定义，请就此打住，因为本文中没有这样的定义"。

[14.2] Philip A. Bernstein: "The Repository: A Modern Vision," *DBP &D 9*, No. 12 (December 1996).

在写这篇论文的时候好像正在面临由术语信息中心（repository）取代术语数据字典的趋势。一个信息中心系统是特定应用于元数据（metadata）管理的 DBMS——元数据并不只是用于 DBMS 而是用于各种类型的软件工具，引用 Bernstein 的话说："是软件设计、开发和配置的工具，也是与工程内容相关的管理工具，比如对电子设计、机械设计、网址和许多其他方面的正式文档的管理等"。这篇论文是有关信息中心概念的一个辅导。

[14.3] Michael Blaha and William Premerlani: *Object-Oriented Modeling and Design for Database Applications*. Upper Saddle River, N. J.: Prentice Hall (1998).

详细描述了一个称为对象建模技术（OMT）的设计方法。OMT 可以看作是 E/R 模型的一个变种（它的对象是基本的 E/R 实体）但是它远不只限于数据库设计范畴。见参考文献 [14.37] 的注释。

[14.4] Grady Booch: *Object-Oriented Design with Applications*. Redwood City, Calif.: Benjamin/ Cummings (1991).

参见参考文献 [14.37] 的注释。

[14.5] Grady Booch, James Rumbaugh, and Ivar Jacobson: *The Unified Modeling Language User Guide*. Reading, Mass.: Addison-Wesley (1999).

参见参考文献 [14.37] 的注释。注意：两本相关的书源自相同的作者（排列顺序不同）：《The Unified Modeling Language Reference Manual》（Rumbaugh, Jacobson, Booch）和《The Unified Software Development Process》（Jacobson, Booch, Rumbaugh），它们都是 1999 年由 Addison-Wesley 出版的。

[14.6] Peter Pin-Shan Chen: "The Entity-Relationship Model—Toward a Unified View of Data," *ACM TODS 1*, No. 1 (March 1976). Republished in Michael Stonebraker (ed.), *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988).

这篇论文介绍了 E/R 模型和 E/R 图。在本章中已经介绍过，这个模型是随着时间推移不断地修改和完善的；在这第一篇论文中提出的解释和定义肯定不十分精确，所以这样的完善是完全必要的（一种对 E/R 模型的批评是在此模型中的术语并没有一个单一而完善的含义，相反却可以用很多方式来解释。当然，整个数据库领域都受不精确和术语冲突所苦恼，但是 E/R 建模这个具体的领域的情况却比大多数其他的领域都差），表现如下：

- 正如在 14.3 节中所介绍的，一个实体定义为"一个可以被清楚辨识的事物"，一个联系定义为"与其他实体的连接"。这样，第一个问题出现了：联系是实体吗？一个联系也是"可以被清楚辨识的事物"。但是这篇论文中下面的几节却保留了术语"实体"来表示某

些不是联系的东西。假定后面的是有意的解释，那么为什么有术语“实体/联系”模型出现呢？但是这篇论文中没有给出清晰的解释。

- 实体和联系可以有属性（在本章中所应用的是“特性”一词）。这篇论文在这个术语的含义上又出现了自相矛盾的情况——首先它定义了一个属性，它是一个非主码也非主码的任何组成部分的特性（与关系的定义相对来说），但其后它在关系的意义上又用到这个术语。
- 一个联系的主码被假定为辨别了联系中的实体的外码的组合（但其中并没有用到外码这个术语）。这个假定只适用于多对多的联系，而且并非总能适用。例如，对于一个关系变量 SPD {S#, P#, DATE, QTY}，它代表了在某些天中某些供应商提供的某些零件的数量；假定同意供应商可以多次运送同样的零件，但是不在同一天。那么主码（或者至少是单独的候选码）是 {S#, P#, DATE} 的组合；然而可以将供应商和零件作为实体来看待，但日期却不能。

[14.7] E. F. Codd: “Extending the Database Relational Model to Capture More Meaning,” *ACM TODS 4*, No. 4 (December 1979).

在这篇论文中，Codd 介绍了一个“扩展”的关系模型版本，他称之为 RM/T。RM/T 论述了与 E/R 模型同样的一些问题，但是它定义得更详细。这两种模型的一些直接的不同如下：首先，RM/T 对于实体和联系并没有作任何不必要的区分（一个联系只是作为一个特殊类型的实体类看待）。第二，RM/T 较之 E/R 模型的结构和完整性方面更庞大并且定义得更精确。第三，RM/T 在基本的关系模型的操作符基础上建立它自己的特殊的操作符（虽然许多附加的工作在后面还需完善）。

在大体结构上，RM/T 模型的内容如下：

- 首先，实体（包括“联系”）是由 *E* 关系和 *P* 关系变量代表的，[⊖] 这两种关系都是一般的 *n* 元关系的特殊形式。*E* 关系用来记录存在的实体，*P* 关系用来记录这些实体的特性。
- 第二，一组不同的联系可以在实体中存在——例如，实体类型 *A* 和 *B* 可以用一个**关联**（association）（RM/T 中的多对多联系的术语）被连在一起，或实体类型 *Y* 可以是实体类型 *X* 的**子类型**。RM/T 包括一个正式的**目录结构**，通过这样的结构系统就可以了解这些联系；因此，系统能够保证由这些联系所蕴涵的各种**完整性约束**。
- 第三，提供了许多更高级别的**操作符**来方便不同的 RM/T 对象的处理（*E* 关系、*P* 关系、目录关系等）。

与 E/R 模型相似，RM/T 模型也包含了图 14-1 中列出的所有的概念（实体、特性、联系、子类型）。它具体地提供了一种**实体分类模式**（这种模式在很大程度上形成了整个模型的最重要的方面——或者说，最直接可见的方面），根据这种模式，实体被划分为三类，即内核、特征和连接：

- **内核**（kernel）：内核实体是指有独立存在性（independent existence）的实体：它们体现的是“数据库究竟是什么?”。换句话说，内核是既非特征也非关联的实体（参见下两条）。
- **特征**（characteristic）：一个特征实体是以描述和表现一些其他实体属性为主要目的的实体。特征是它们所描述的实体上的存在依赖（existence-dependent）。所描述的实体可以是内核，特征和关联。
- **关联**（Association）：一个关联实体是在两个或多个其他实体之间的多对多联系（或多对多对多，等），关联的实体可以是内核、特征和关联。

另外：

- 实体（不管属于哪一类）可以有**特性**。
- 特别地，任何实体（不管其属于哪一类）都可以**指派**（designate）一些其他的相关特性。一个指派代表两个实体之间的多对一联系。*注意*：在最初的论文 [14.6] 中没有讨论指派，这是后来加入的。
- 支持实体**子类型**和**超类型**。如果 *Y* 是 *X* 的一个子类型，那么 *Y* 是一个内核，特征还是关联，依赖于 *X* 是一个内核、特征还是关联。

可以将前面的概念与 E/R 模型做一比较（大致上）如下：一个内核与 E/R “常规实体”相

---

⊖ 论文中称为 *E* 和 *P* 关系。

当，一个属性与 E/R “弱实体” 相当，一个关联与 E/R “联系” 相当（只是多对多情况）。

除了上面简单介绍的方面外，RM/T 还包括对以下内容的支持：（a）替代（surrogates）（见参考文献 [14.21]），（b）时间维（参见第 23 章），（c）各种数据聚集（参见文献[14.40，14.41]）。

[14.8] C. J. Date: “A Note on One-to-One Relationships,” in *Relational Database Writings 1985 - 1989*. Reading, Mass.: Addison-Wesley (1990).

关于一对一联系问题的一个广泛的讨论，证明了一对一问题比最初看来要复杂得多。

[14.9] C. J. Date: “Entity/Relationship Modeling and the Relational Model,” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

[14.10] C. J. Date: “Don't Encode Information into Primary Keys!”, in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading. Mass.: Addison-Wesley (1992).

这篇论文提出了一系列有关 “智能码”（intelligent key）的非正式的讨论。关于外码问题见参考文献 [14.11] 中的相关介绍。

[14.11] C. J. Date: “Composite Keys,” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading. Mass.: Addison-Wesley (1992).

引自摘要：“总结了关于是否支持在关系数据库设计中包含的复合（码）的问题，并且给出……一些建议”。而且，这篇文章中指出了替代码（参考文献 [14.21]）的问题。

[14.12] C. J. Date: “A Database Design Dilemma?”, *http://www.dbpd.com* (January 1999).

从表面判断，一个给定的实体类型，如雇员，在关系系统中既可以通过雇员类型（即一个域）表示，也可以通过雇员关系变量表示。这篇短文（基于参考文献 [3.3] 的附件 C）给出了如何在两者中作出选择。

[14.13] C. J. Date: “Subtables and Supertables ”Appendix E of reference [3.3].

通常认为实体类型继承应该在关系上下文中通过一种 “子表和父表”（实体子类型映射到一个 “子表”，实体父类型映射到一个 “父表”）的方式来处理。例如，SQL 就支持这样一种方法（见第 26 章），某些产品也支持。这篇论文对此持反对意见。

[14.14] C. J. Date: “Twelve Rules for Business Rules,” *http://www.versata.com* (May 1,2000).

提出了一系列 “好” 的商业规则应当遵守的规则或规定。

[14.15] C. J. Date: “Models, Models, Everywhere, Not Any Time to Think,” *http://www.dbdebunk.com* (November 2000).

“模型” 这一术语在 IT 界使用过度（不是说滥用），尤其是在数据库界。这篇论文提醒人们注意在这一点上最糟糕的无节制使用，并且说服人们在使用这一术语时三思而后行。

[14.16] C. J. Date: “Basic Concepts in UML: A Request for Clarification,” *http://www.dbdebunk.com* (December 2000/January 2001).

这篇论文分为两部分，它详细参照了对象约束语言 OCL，对统一建模语言 UML，进行了检验和严肃的批判性的分析。第一部分主要是关于 “OCL 书籍” [14.49]；第二部分着眼于 “UML 书籍” [14.5]。

[14.17] Debabrata Dey, Veda C. Storey, and Terence M. Barron: “Improving Database Design Through the Analysis of Relationships,” *ACM TODS 24*, No. 4 (December 1999).

[14.18] Ramez Elmasri and Shamkant B. Navathe: *Fundamentals of Database Systems* (3d ed.). Redwood City, Calif.: Benjamin/Cummings (2000).

这本关于数据库管理的教材中用整整两章讨论了在数据库设计中如何使用 E/R 技术。

[14.19] David W. Embley: *Object Database Development: Concepts and Principles*. Reading, Mass.: Addison-Wesley (1998).

提供了一种基于 OSM（Object-oriented Systems Model，面向对象的系统模型）的设计方法。OSM 的某些部分很像 ORM [14.22 - 14.24]。

[14.20] Candace C. Fleming and Barbara von Hallé: *Handbook of Relational Database Design*. Reading, Mass.: Addison-Wesley (1989).

针对关系系统中数据库设计的一个实用指南，包括基于 IBM 的 DB2 产品和 Teradata（现在是 NCR）DBC/1012 产品的具体的例子。在这本书中逻辑设计和物理设计的内容都提到了——虽然这本书用 “逻辑设计” 的术语来表示 “关系设计”，用 “关系设计” 这个术语包括

"物理设计"中的一些方面。

[14.21] P. Hall, J. Owlett, and S. J. P. Todd: "Relations and Entities," in G. M. Nijssen (ed.), *Modelling in Data Base Management Systems*. Amsterdam, Netherlands: North-Holland / New York, N. Y.: Elsevier Science (1975).

这是第一篇详细论述**替代码**概念的论文（这个概念后来用于 RM/T 中 [14.7]）。替代码是通常意义上的关系中的码，但是有如下几个特殊的特征：

- 它们通常只包括一个属性。
- 它们的值只作为它们代表的实体的替代。换句话说，这些值只是表明相应的实体存在——它们并不带有其他任何信息或含义。
- 当一个新实体插入到数据库中，就会被赋予一个从未用过的替代码的值，并且这个值也不会再用，甚至当这个实体已经被删除后也不能用。

最好的情况是。替代码是系统生成的。但是，无论是系统生成的还是用户给出的，都与替代码这个基本思想无关。

值得强调的是替代码并非（如一些作者所认为的）是与"元组 ID"相同。第一，明确地说，元组 ID 标识了元组而替代码标识了实体，并且在元组和实体之间并没有一对一的联系（特别是对导出元组的 ID）。而且，元组 ID 有性能上的含义而替代码没有；通过元组 ID 来访问元组通常被认为是很快的（假定元组一至少是在基本关系中的元组一直接映射了物理存储，像现在的许多产品一样）。并且元组 ID 通常是对用户隐藏的，而替代码不能对用户隐藏（由于信息原理）；换句话说，不可能像一个属性值一样来存储一个元组 ID，但可以这样存储一个替代码。

简单一句话：替代码是一个逻辑概念；而元组 ID 是一个物理概念。

[14.22] Terry Halpin: *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. San Francisco, Calif.: Morgan Kaufmann(2001).

详细阐述了 ORM（见以下两个参考文件的注释）。本书讨论了（a）ORM 和 E/R 建模之间的关系；（b）ORM 和 UML 之间的关系。*注意*：本书主要是对作者早期的书《Conceptual Schema and Relational Database Design (2d edition)》——由 Prentice Hall of Australia Pty., Ltd.（1995）出版的修订和扩展。

[14.23] Terry Halpin: "Business Rules and Object-Role Modeling," *DBP &D* 9, No. 10 (October 1996).

一个**对象 - 角色建模**（object-role modeling, ORM [14.22]）的非常好的介绍。Halpin 开始其研究，并认为"（不像）E/R 建模，存在很多方言，ORM 只有很少的方言，并且区别很小。"（*注意*：其中的一个方言是 NIAM [14.34]。）ORM 还以基于事实建模著称，因为设计者所做的是写下（或者是用自然语言，或者是通过一种特殊的图符号）一系列基本事实（或是事实类型（fact type）），这些事实在一起刻画了要被建模的企业的特征。这样的事实类型的例子如下：

- 每一个雇员至多有一个雇员姓名 Empname。
- 每一个雇员至多向一个雇员汇报。
- 如果雇员 *e1* 向雇员 *e2* 汇报，那么 *e2* 不能向雇员 *e1* 汇报。
- 没有雇员可以既指导又进入同一个项目。

正如所见，事实类型其实就是谓词或企业规则；正如 Halpin 的论文题目所表明的，ORM 确实是数据库设计方法的精华，它通常被"企业规则"的倡导者和作者所喜欢。通常，在联系中对象所扮演的角色是由事实指定的（因此有"对象 - 角色建模"的名字）。注意（a）在这里"对象"表示实体，而不是在这本书第六部分中描述的特殊意义上的对象；（b）联系并不必是二元的。然而，事实是基本的——它们不能分解为更小的事实。*注意*：数据库只应该包含在概念层的基本（或不可分解的）事实的观点是由 Hall、Owlett 和 Todd（见参考文献 [14.21]）更早提出的。

观察到 ORM 没有"属性"概念。因此，在这篇文章中显示（见参考文献 [14.24] 的注释），ORM 设计在概念上比 E/R 模型设计的相应部分更简单并且更强。然而，属性可以并且确实出现在由 ORM 设计所（自动）产生的 E/R 模型或 SQL 中。

ORM 还强调了"样本事实"（sample fact）（即样本事实实例—称为命题（proposition））的应用，即作为允许终端用户检查设计合法性的方法。基于事实的建模的方法是易懂的，而 E/

R 建模则要差许多。

存在许多描述一个企业的逻辑上相同的方法，因此有许多逻辑上相同的方案。ORM 包含一组转换规则，这种规则允许逻辑上相同的方案互相转换，所以一个 ORM 工具在设计者指定的设计上可以进行一些优化。如前面提到的，它还可以从一个 ORM 方案产生 E/R 或 SQL 方案，并且它还能相反地从一个存在的 E/R 或 SQL 方案产生一个 ORM 方案。基于 DBMS 目标，一个产生的 SQL 方案可以包含（SQL/92 标准）定义的约束或那些可以通过存储或触发过程执行的约束。顺便提及，考虑这些约束时注意——并不像 E/R 建模——ORM 定义时包含“一个表现约束的丰富的语言”。（然而，Halpin 在参考文献［14.24］中确实承认，并不是所有的企业规则都可以用 ORM 图形符号表现出来——在这种情况下，文本描述也还是需要的）。

最后，一个 ORM 方法可以被看作是一个高级的抽象的数据库视图（事实上，它是相当接近于一个纯粹的规则化的关系视图）。因此，它可以直接查询。参见下面参考文献［14.24］中的注释。

［14.24］ Terry Halpin：“Conceptual Queries,”*Data Base Newsletter 26*, No. 2 (March/ April 1998).

引自摘要：“在关系语言中构造非平凡查询如 SQL 或 QBE，可以证明会使用户畏缩。ConQuer，一个基于对象 - 角色建模（ORM）的新的概念查询语言可以方便用户以一种易于理解的方式构造查询……”该文指出了在构造查询和企业规则上，该语言要优于传统查询语言。在其他讨论中，这篇论文还讨论了一个如下的 ConQuer 查询：

```
✓ Employee
    +− drives Car
    +− works for ✓ Branch
```

（“找出驾驶轿车的雇员及其所在部门”）。如果雇员可以驾驶任何数量的小轿车但只能在一个部门工作，其 SQL 设计就要包括两个表，产生的 SQL 代码如下：

```
SELECT DISTINCT X1.EMP#, X1.BRANCH#
FROM   EMPLOYEE AS X1, DRIVES AS X2
WHERE  X1.EMP# = X2.EMP# ;
```

现在假定对于每一个员工同时在多个部门工作是可能的。那么 SQL 设计就要变化为包括三个表而不是两个，产生的 SQL 代码也将变化为：

```
SELECT DISTINCT X1.EMP#, X3.BRANCH#
FROM   EMPLOYEE AS X1, DRIVES AS X2, WORKS_FOR AS X3
WHERE  X1.EMP# = X2.EMP# AND X1.EMP# = X3.EMP# ;
```

然而，ConQuer 查询保持不变。

正如上面的例子所示，像 ConQuer 这样的语言可以看作是提供一个特别强的逻辑数据独立性形式。为了解释这种说法，首先需要细化 ANSI/SPARC 体系结构。在第 2 章中提到逻辑数据独立性表示对于概念模式的变化的独立性——但是，前面的例子的总体观点是概念模式的变化并没有发生！问题是现在的 SQL 产品并不能很好地支持一种概念模式，而只是支持 SQL 模式。SQL 模式可以被看作是存在于真正的概念层和内部或说物理层之间的中间层。如果一个 ORM 工具允许定义一个真正的概念模式，并且将这个模式映射到一个 SQL 模式上，那么 ConQuer 就可以提供 SQL 模式的独立性（当然是通过对映射做适当的改变）。

在这篇论文中并没有清楚说明 ConQuer 的表现力究竟有什么限制。Halpin 并不直接谈论这个问题；而是说：“这种语言应该很完美地允许根据应用来形成相应的问题；在实践中，较之稍差一些的语言也是可以接受的”。它还提到 ConQuer 的“最强大的特征……是其能够执行任意复杂的相关性查询”。下面是一个例子：

```
✓ Employee1
    +− lives in City1
    +− was born in Country1
    +− supervises Employee2
         +− lives in City1
         +− was born in Country2 <> Country1
```

（“找出监督一个与其住在同一个城市但不在一个城市出生的雇员的雇员信息”）。正如 Halpin 所说的“试着将查询用 SQL 表示！”。

最后，根据 ConQuer 和企业规则的观点，Halpin 指出：“虽然 ORM 的图形符号可以较之

[E/R 方法] 描述更多的企业规则，它还是需要文本语言的补充 [来表示特定的约束]。将 ConQuer 应用于这个目的的研究正在进行。

[14.25] M. M. Hammer and D. J. McLeod: "The Semantic Date Model: A Modelling Mechanism for Database Applications," Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. (May/June 1978).

语义数据模型（SDM）代表另一种数据库设计形式。与 E/R 模型相似，它关注了结构和（某种程度上）完整性方面的问题，并且在操作方面言之很少，甚或什么都没有说。见参考文献 [14.26] 和 [14.29]。

[14.26] Michael Hammer and Dennis McLeod: "Database Description with SDM: A Semantic Database Model," *ACM TODS 6*, No. 3 (September 1981).

见参考文献 [12.25] 的注释。

[14.27] Richard Hull and Roger King: "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Comp. Surv. 19*, No. 3 (September 1987).

一个全面的对在 20 世纪 80 年代末语义建模领域及相关情况的综述。这篇文章是对此问题开始一个更深入的调查和研究语义建模行为有关情况的很好的起点。见参考文献 [14.36].

[14.28] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, and Gunnar Övergaard: *Object-Oriented Software Engineering* (revised printing). Reading, Mass.: Addison-Wesley (1994).

描述一种称为面向对象软件工程（OOSE）的设计方法。与 OMT [14.3] 相似，OOSE 的数据库部分至少可以看作是 E/R 模型的一个变体（与 OMT 相同，OOSE 的对象是基本的 E/R 实体）。下面的引述是值得注意的："现在在工业中应用的所有方法，对于信息技术系统发展来说，都是基于系统的函数和/或数据 - 驱动的分解。这些方法在许多方面和面向对象方法不同，在后一种方法中，数据和函数是有很高的结合度的"。看起来在这里 Jacobson 关注了在对象和数据库思想之间的重要的不符合性。数据库（至少是普遍意义上的共享数据库，这种数据库是数据库领域所关注的主要焦点）被假定是与"函数"分离的；假定它们是与使用它们的应用系统分离设计的。因此，在对象领域里，"数据库"的含义是指一个特定的应用的数据库，并不是一个共享和普遍意义上的数据库。（关于这一点，见第 25 章对象数据库的讨论）。也可见参考文献 [14.5]、[14.16] 和 [14.37]。

[14.29] D. Jagannathan *et al.*: "SIM: A Database System Based on the Semantic Data Model," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, Ill. (June 1988).

描述了一个基于与 Hammer 和 McLeod 在参考文献 [14.25] 中提出的语义数据模型相似的"一个语义数据模型"的商业化 DBMS 产品。

[14.30] Warren Keuffel: "Battle of the Modeling Techniques: A Look at the Three Most Popular Modeling Notations for Distilling the Essence of Data," *DBMS 9*, No. 9 (August 1996).

"三个最普遍的符号"是 E/R 建模，Nijssen 的自然语言信息分析方法，即 NIAM [14.34]，和语义对象建模，即 SOM。Keuffel 称 E/R 建模是其他两种方法的"祖父"，但是他批评了这种方法缺少正式的根基；他说，实体、联系和属性（即特性）都是"没有参考它们是怎么发现而描述的"。NIAM 更严格一些；当严格遵守其设计规则时，所得出的概念设计才比用其他方法设计的东西"具有更多的完整性"，虽然"一些开发者发现 NIAM 的严格设计太闭塞"！对于 SOM，它与"E/R 模型非常相似……它们有相同的对于实体、属性和联系的模糊定义"；然而，它与 E/R 模型也有不同，它支持组属性（即重复组），组属性允许一个"对象"（即实体）包括其他对象（E/R 建模允许实体包括属性的重复组，但不允许包含其他实体）。

[14.31] Heikki Mannila and Kari-Jouko Räihä: *The Design of Relational Databases*. Reading, Mass.: Addison-Wesley (1992).

引用该书的前言，是"一本适合研究生水平的教材，是关系数据库设计的参考书"。它一方面包含依赖理论和规范化理论，另一方面包含了 E/R 方法，每种情况都是从一个正规的角度来描述的。下面的这些章节列出了这本书的大概范围：

- 设计原理
- 完整性约束和依赖
- 关系模式的特性
- 依赖公理

- 有关设计问题的算法
- E/R 图和关系数据库模式之间的映射
- 模式转换
- 设计中的实例数据库

作者对该书介绍的技术，用一种称为 Design By Example 的工具加以实现。

[14.32] Terry Moriarty: *Enterprise View* (regular column), *DBP &D 10*, No. 8 (August 1997).

描述了一种企业应用设计和一种称为 Usoft 的开发工具（http://www.usoft.com），它们允许用一种像 SQL 一样的语法来定义企业规则，并且应用这些规则产生应用（包括数据库定义）。

[14.33] G. M. Nijssen, D. J. Duke, and S. M. Twine: "The Entity-Relationship Data Model Considered Harmful," Proc. 6th Symposium on Empirical Foundations of Information and Software Sciences, Atlanta, Ga. (October 1988).

"E/R 模型被认为是有害的么?" 看起来确实有许多需要回答的地方，包括：

- 类型和关系变量引起的混乱（参见第 26 章中关于**第一个大错误**的讨论）。
- 关于 "子表和父表" 的奇怪的事情（见参考文献 [14.13] 及第 26 章）；
- 对一个散布很广的《数据库相对准则》的估价的失败（参见第 10 章）；
- 实体和联系本身引起的混乱，在前一章论述。参考文献 [14.33] 中提出许多疑问。具体地说，它声称 E/R 模型：
- 提出太多数据结构的重叠的方法，因此使设计过程不适当；
- 没有指出如何选择不同的表示法，并且事实上如果环境变化，可以使现有的设计发生不必要的变化；
- 很少支持保持数据完整性的方法，因此不能完成设计过程的某些方面（"约束可以用一个更普遍的符号——谓词逻辑来形式化地表示，但据此认为这是一个从数据模型中可以省略约束的合理理由就不太合适了，这好比一种程序设计语言，由于它自身不能表达所有的功能，所以需要用户调用汇编语言来完成，却要称这种程序语言是完善的语言一样"）。
- 与普遍的观点相反，它并不是一个最终用户和数据库专业人员之间交流的很好的工具；
- 违反了概念化原则："一个概念模式应该……只包括与概念相关的方面，不论是动态的还是静态的，排除所有无关的内容，如数据表示（外部的或内部的）、物理数据的组织和存取以及消息格式、数据结构等特定外部用户表示，等等" [2.3]。事实上，作者暗示E/R 模型是老的 CODASYL 网络模型的一个 "再生"。"对实现结构的强烈偏见是否成为E/R 模型在专业领域得到这么广泛的认同的主要原因呢?"

这篇文章还详细地分析了 E/R 模型的许多弱点。然后提出了一个可选方法 NIAM [14.34]。它特别强调了 NIAM 并不包含 E/R 模型中所不必要包含的属性和联系的区别。

[14.34] T. W. Olle, H. G. Sol. and A. A. Verrijn-Stuart (eds.): *Information Systems Design Methodologies: A Comparative Review*. Amsterdam, Netherlands: North-Holland / New York, N. Y.: Elsevier Science (1982).

IFIP Working Group 8.1 会议论文集。描述了 13 个不同的方法，并且将其应用到一个标准的基准问题中。其中的一个方法就是 NIAM（见参考文献 [14.33]）；这篇论文一定是最早讨论 NIAM 方法的文章之一。这本书还包括对一些所提出的方法的回顾，其中也包括了 NIAM 模型。

[14.35] M. P. Papazoglou: "Unraveling the Semantics of Conceptual Schemas," *CACM 38*, No. 9 (September 1995).

这篇文章提出了一种称为元数据查询的方法——即根据数据库中数据的意思进行的查询，换句话说，就是根据概念模式本身进行的查询。这样的查询的一个示例是 "什么是永久雇员?"。

[14.36] Joan Peckham and Fred Maryanski: "Semantic Data Models," *ACM Comp. Surv. 20*, No. 3 (September 1988).

另一个综述（见参考文献 [14.27]）。

[14.37] Paul Reed: "The Unified Modeling Language Takes Shape," *DBMS 11*, No. 8 (July 1998).

统一建模语言（Unified Modeling Language, UML），是另一个支持应用设计和开发的图符号（换句话说，在其中通过画图来开发应用程序）。它还可以用来开发 SQL 模式。注意：UML 很可能会在商业上很重要，部分原因是它被对象管理组 OMG 采纳为标准，UML 具有一种很强

的对象风格。它已经被一组商业产品所支持。

不管怎么样，UML 支持数据和过程的建模（从这一点考虑，它超出了 E/R 建模的范围），但是在完整性约束方面它似乎并没有说明什么（参考文献［14.37］中标题为“从模型到代码：企业规则”一节中根本没有提到 declarative 这个术语！然而，它很重视利用过程应用代码来实现“处理”。以下是直接的引述：“UML 形式化地描述了为大家所熟知的对象概念：即现实世界的对象，最好作为既包含数据又包含函数的自含实体加以建模”。还有：“很明显，从一个历史发展的角度来看，数据和函数的分离导致了软件开发的脆弱”。这些评论从一个应用角度来讲可能是合理的，但是从一个数据库角度来看并不一定就合理。例如，见参考文献［25.25］。

UML 由 Booch 的“Booch 理论”［14.4］、Rumbaugh 的 OMT［14.3］和 Jacobson 的 OOSE［14.28］的早期工作发展而来。也可参考文献［14.5］和［14.6］。

［14.38］ H. A. Schmid and J. R. Swenson：“On the Semantics of the Relational Data Base Model，”Proc. 1975 ACM SIGMOD Int. Conf. on Management of Data，San Jose，Calif.（May 1975）.

这篇文章在 Chen 的 E/R 模型［14.6］之前提出了一个“基本的语义模型”，但是事实上它与 E/R 模型非常相似（当然，除了在术语的使用上；Schmid 和 Swenson 分别用独立对象、依赖对象和关联替换了 Chen 的常规实体、弱实体和联系的术语）。

［14.39］ J. F. Sowa：*Conceptual Structures：Information Processing in Mind and Machine*. Reading，Mass.：Addison-Wesley（1984）.

这本书并不是专门讨论数据库系统的，它可以说是讨论知识表示和处理的一般问题。然而，这本书的一部分是直接与本章所讨论的内容相关的（下面的评论是基于 Sowa 在 1990 年或 1990 年左右关于将“概念结构”应用于语义建模的生动介绍的）。E/R 图和类似的表现形式中一个主要的问题是它们没有正式的逻辑严密性。因此，它们不能处理某些重要的设计特点（特别是量词，这涉及到完整性约束的处理问题）而形式逻辑就可以处理这些内容（量词是由 Frege 在 1879 年提出的，这使得 E/R 图只能成为“一种 1879 年以前的逻辑”）。但是形式逻辑不易为接受；正如 Sowa 所说：“谓词演算与知识表示语言非常相似”。概念图是一种可读的、严格的图符号，它可以表示整个逻辑。因此它们（Sowa 称）比 E/R 图和类似的图表更适合用于语义建模。

［14.40］ J. M. Smith and D. C. P. Smith：“Database Abstractions：Aggregation，”*CACM 20*，No. 6（June 1977）.

见参考文献［14.41］。

［14.41］ J. M. Smith and D. C. P. Smith：“Database Abstractions：Aggregation and Generalization，”*ACM TODS 2*，No. 2（June 1977）.

这两篇论文［14.40］和［14.41］所提出的内容对 RM/T［14.7］有很重要的影响，特别是在实体子类型和超类型方面。

［14.42］ Veda C. Storey：“Understanding Semantic Relationships，”*The VLDB Journal 2*，No. 4（October 1993）.

引自摘要：“语义数据模型应用抽象，如子类型、聚合以及关联的方式（在数据库领域）都得到了很好的发展。除了这些常见的联系，许多其他的语义联系也被研究者在其他领域规则中辨识出来，如语言学、逻辑和认知心理学领域。这篇文章探讨了一些这样的联系并且讨论了……它们对数据库设计的影响。”

［14.43］ B. Sundgren：“The Infological Approach to Data Bases，”in J. W. Klimbie and K. L. Koffeman（eds.），*Data Base Management*. Amsterdam，Netherlands：North-Holland / New York，N. Y.：Elsevier Science（1974）.

“信息逻辑方法”（infological approach）是最早被提出的逻辑建模方法之一。它在 Scandinavia 数据库设计领域成功地应用了许多年。

［14.44］ Dan Tasker：*Fourth Generation Data：A Guide to Data Analysis for New and Old Systems*. Sydney，Australia：Prentice Hall of Australia Pty.，Ltd.（1989）.

数据库设计的实用指南，它将重点放在单个数据项（data item）上（即放在域上）。数据项被分为三种基本种类：标签、数量和描述。标签项代表实体；在关系术语中，它与主码和外码相对应。数量项代表一个范围（scale）（可能是一个时间列范围）的数量、量度或状态，并且用于常规的算术操作。描述项是所有剩下的部分（当然，对这种分类方案的论述很详细，而这里的简要介绍则不能完全概括）。这本书还详细介绍了每一种类型。这些讨论并不总是“纯关系”

的（例如，Tasker 对“域”这个词的使用就与其在关系上的使用不完全相同）但是这本书提供了很多可行的实际建议。

[14.45] Toby J. Teorey and James P. Fry: *Design of Database Structures*. Englewood Cliffs, N. J.: Prentice Hall (1982).

是一本有关数据库设计方方面面的教科书。这本书分为五部分：引言、概念设计、实现设计（即将概念设计映射到一个具体的 DBMS 可以理解的结构上）、物理设计和特殊的设计问题。

[14.46] Toby J. Teorey, Dongqing Yang, and James P. Fry: “A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model,” *ACM Comp. Surv. 18*, No. 2 (June 1986).

论文题目为“扩展的 E/R 模型”提出了对实体类型层次、空值（null）（参见第 19 章）和包含两个以上参与者的联系的支持。

[14.47] Toby J. Teorey: *Database Modeling and Design: The Entity-Relationship Approach* (3rd edition). San Francisco, Calif.: Morgan Kaufmann (1998).

近期出版的一本关于数据库设计中 E/R 应用和“扩展”的 E/R 概念 [14.41] 的教科书。

[14.48] Yair Wand, Veda C. Storey, and Ron Weber: “An Ontological Analysis of the Relationship Construct in Conceptual Modeling,” *ACM TODS 24*, No. 4 (December 1999).

[14.49] Jos Warmer and Anneke Kleppe: *The Object Constraint Language: Precise Modeling with UML*. Reading, Mass.: Addison-Wesley (1999).

见参考文献 [14.16] 的注释。

# 第四部分　事务管理

这一部分由两章组成，内容分别是恢复和并发，其内容彼此交错，共同构成了事务管理的主要内容。为便于教学，将这部分内容组织为两个章节。

恢复和并发，或者叫恢复和并发控制，都是关于**数据保护**的——即保证数据不丢失或被毁坏。恢复和并发尤其关注以下问题：

- 系统在执行程序的过程中会出现故障，因此会使数据库处于一个未知状态；
- 两个程序在同时执行（即“并发”）时，会相互交错干扰，因此会造成不正确的结果，这种错误有可能发生在数据库的内部，也可能发生在外部的现实世界。

第15章是有关恢复的内容，第16章则是关于并发的讨论。

# 第15章 恢 复

## 15.1 引言

正如在本部分的引言中提到的那样，恢复和并发控制的内容彼此交错。为了教学的目的，也为了更清楚地阐述这两个概念，特将它们分成两章。在这一章我们集中讨论恢复的内容，并发将在第16章讨论（本章将不可避免地多次引用后一章的内容，特别是在15.4节）。

数据库系统中的**恢复**（recovery）主要指恢复数据库本身，即在故障引起数据库当前状态不一致后将数据库恢复到某个正确状态或一致状态。（将在第五部分详细阐述“数据库正确状态”的含义。）恢复的原理很简单，可以用一个词来概括，即**冗余**（redundancy）。（冗余是物理级的，我们通常认为逻辑级没有冗余，其原因将在本书的其他部分详细阐述。）换句话说，确定数据库是否可恢复的方法就是：确定其包含的每一条信息是否都可利用冗余地存储在系统中的信息重构。

在进行其他深入的讨论前，我们需要先弄清楚，恢复的思想（实际上是整个事务处理的思想）与系统是关系型还是层次、网状或其他模型无关——虽然过去或将来大部分关于事务处理过程的理论研究都是基于关系型系统的。我们必须清醒地意识到这是一个内容丰富的课题，所以在这里只介绍有关恢复的最重要和最基本的思想，若要对恢复作进一步的了解，可参看“参考文献”一节（尤其推荐参考文献[15.12]）。

本章各节的内容如下：在引言之后，15.2节和15.3节给出了事务的基本定义以及事务恢复的基本思想（将数据库从单个事务故障中恢复）；15.4节将事务恢复的思想扩展到系统恢复（将系统从引起多个并发事务同时发生故障的系统崩溃中恢复）；15.5节将事务恢复的思想继续扩展到介质故障恢复（在数据库物理上受到损害后恢复，这样的损害如磁盘的磁头碰撞等）；15.6节介绍一个相当重要的概念——两阶段提交（two-phase commit）；15.7节讨论保存点（savepoint）；15.8节描述SQL中的相关语句；15.9节进行了小结，并给出结论性的评论。

最后需要补充一点：我们假设数据库环境是很“大”的，即共享、多用户的。“小”的，即非共享、单用户的DBMS通常不提供或只提供很少的一部分恢复支持，这种环境下恢复是用户的责任。这就意味着用户必须进行周期性的数据库备份，当故障发生时必须手工恢复。

## 15.2 事务

事务是一个**逻辑工作单元**（logical unit of work）；它以一个BEGIN TRANSACTION作为动作执行的开始，以一个COMMIT或者ROOLBACK作为动作执行的结束。参看图15-1，图中的伪代码描述了一个将100元从账户123转移到账户456的事务。可以看到，“将一个账户上的钱转账到另一个账户”的这样一个原子操作，实际上包含了对数据库的两个独立的更新。数据库的两次更新之间存在一个错误的状态，这个状态不能在现实世界的事务状态中找到对应物。可以肯定的是，现实中的一次转账并不影响那些账户上的总金额，但是在这个例子事务的两次更新之间，有100元暂时地丢失了。所谓逻辑工作单元，是指一个事务并不是只能包含一个单一的数据库操作；相反，它包含一个操作序列，保证数据库是从一个正确的状态转移到另外一个正确的状态，中间状态无须保证正确性。

显然，上例的两个更新操作中一个执行而另一个未执行的情况是不允许发生的，否则会使数据库处于不一致的状态。理想的办法是可靠地保证这两个操作都被执行，很不幸，不可能提供这样的保证——错误不可避免，且可能发生最坏的情况。如两次更新之间发生系统崩溃，后面一次

更新时发生运算溢出，等等。[⊖] 支持**事务管理**（transaction management）的系统提供了另一种相当可靠的保证方式。它保证如果事务执行了几个更新操作，并在事务结束前发生了故障，这些更新操作将被撤消。也就是说，事务或者完全执行，或者全部取消（就像根本没执行过）。尽管操作序列本质上不是原子的，但从外部的角度看就像是原子的一样。

```
    BEGIN TRANSACTION ;

    UPDATE ACC 123 ( BALANCE := BALANCE - $100 ) ;
    IF any error occurred THEN GO TO UNDO ; END IF ;

    UPDATE ACC 456 ( BALANCE := BALANCE + $100 ) ;
    IF any error occurred THEN GO TO UNDO ; END IF ;

    COMMIT ;                          /* 成功终止 */
    GO TO FINISH ;

UNDO :
    ROLLBACK ;                        /* 未成功终止 */

FINISH :
    RETURN ;
```

图 15-1 事务例子（伪代码）

提供原子性保证的系统组成部分是**事务管理器**（transaction manager），亦称为**事务处理监控器**（transation processing monitor 或 TP monitor），COMMIT（提交）和 ROLLBACK（回滚）操作是其中的关键。

- COMMIT 操作表明事务成功地结束：该操作告诉事务管理器一个逻辑工作单元已成功完成，数据库应该又处于一个一致性状态，该工作单元的所有更新操作现在可被提交或永久保留。
- ROLLBACK 操作表明事务没有成功结束：该操作告诉事务管理器出故障了，数据库可能处于不一致的状态，该逻辑工作单元已做的所有更新操作必须回滚或撤消。

因此，对上面的例子，在两个更新操作成功执行完后，可发出 COMMIT 命令提交数据库所发生的变化并使结果永久保存。如果发生了错误，也就是说，若任何一个更新操作产生了错误，必须发出 ROLLBACK 命令撤消已做的所有更新。

我们可以从图 15-1 的简单例子中得出以下几点重要的共识：

- **隐式的 ROLLBACK**：上面的例子包含了显式的检查，如果检测到错误，则执行显式的 ROLLBACK。但是，我们不能假设也不想假设事务总是包含对所有可能错误的检查，因此，我们对任何非正常结束的事务（“正常结束”意味着一个显式的 COMMIT 或者显式的 ROLLBACK）执行隐式的 ROLLBACK。
- **消息处理**（message handling）：一个典型的事务往往不只是更新或者试图更新数据库，它同时向终端的客户返回必要的信息，表明执行情况。在上面的例子中，如果事务执行到 COMMIT，返回一条“转账成功”的消息；否则返回一条“错误——转账没有完成”的消息。这里消息处理同时也暗示着恢复操作发生。请见参考文献［15.12］中对此的详细讨论。
- **恢复日志**：撤消更新是否可以实现呢？回答当然是肯定的。系统在磁带或者磁盘（更普遍）上维护一个**日志**，更新的所有细节，特别是更新对象（例如元组）在更新前后的值（也称前像快照和后像快照）都被记录在日志中。当需要撤消更新时，系统使用相应的日志记录将更新对象还原到它更新前的值。注意：这个解释有点过于简单了，实际上，日志包括两个部分：一个活动的联机部分和一个存档的脱机部分。系统一般的记录的更新操作

---

⊖ 系统崩溃会影响到正在运行的所有事务，也称为全局故障或者系统故障；运算溢出常常只影响一个事务，也称为局部故障。请分别参考 15.4 节和 15.3 节。

仅仅和联机的日志部分交互，这部分保存在磁盘上，当联机部分的日志充满时，它的内容被转移到脱机的部分，脱机部分可以存放在磁带上（因为对它的访问总是顺序的）。

- **语句原子性**（statement atomicity）：系统必须保证各个独立的语句的执行是原子的。这一点在关系数据库系统中显得尤其重要。因为它的语句是集合操作而且往往是在多个元组上同时进行，不能允许一个语句在执行的过程中失败（例如一部分元组被更新而另一部分没有更新）而使数据库处于错误状态。换言之，如果在一个语句执行的过程中发生了错误，数据库系统必须保持状态不变。正如在第 9 章和第 10 章中提到的那样，如果某个语句会引起隐式的额外更新（比如存在级联删除规则或者一个连接视图被更新的情况），语句的原子性同样需要保证。
- 程序执行是一个事务的序列：COMMIT 和 ROLLBACK 只是一个事务的结束，而不是应用程序的结束。一个程序通常由几个事务依次执行组成的序列构成，如图 15-2 所示。

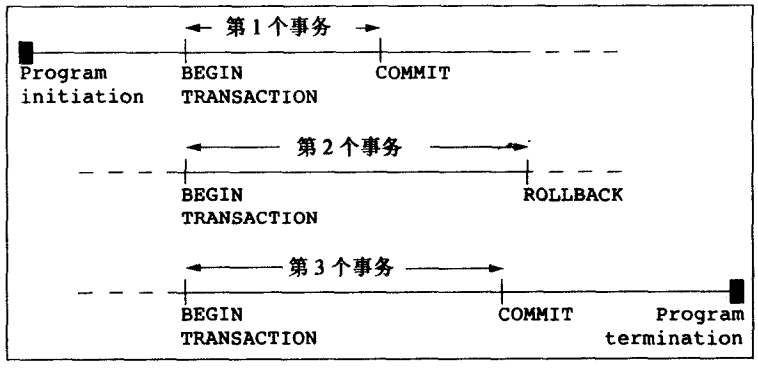



图 15-2　程序的执行是一个事务序列

- 没有嵌套的事务：在这里，我们假定应用程序只有在当前没有事务正在执行的时候才能执行 BEGIN TRANSACTION 语句。也就是说，在一个事务执行的过程中不能有其他的事务嵌套执行。但是请注意在下一章我们将重新讨论这个假定。
- 正确性：根据定义，数据库必须总是在一致的状态中。按照第 9 章（实际上是按照一般语义）的描述，一致性的精确定义是“不破坏任何已知的完整性约束”。一致性是由 DBMS 来保证的。而事务的定义是将数据库从一个一致状态转换到另一个一致状态。但是，仅有一致性是不够的；我们还需要正确性。在图 15-1 所示的事务例子中，我们希望保持账户 123 和账户 456 上的总金额不变。但是，定义那样的完整性约束是不切实际的，（为什么？）所以我们不能期望由 DBMS 来实现这样的需求。事实上，在第 9 章的学习中我们已经知道一致性并不意味着正确性。因此，我们和 DBMS 都只能够假设事务总是正确的，它们正如人们所设想的那样仅仅是反映现实的操作。下面给出更准确的描述：我们假设一个事务 *T* 将数据库从状态 *D*1 转换到 *D*2，如果 *D*1 正确，那么 *D*2 也正确。[⊖] 值得重申的是，系统不能保证这个期望的特性（我们在第 9 章曾提出：“系统不能保证正确性，只能保证一致性”）。
- **多任务**（multiple assignment）：回顾第 5 章，并参考文献 [3.3]，我们需要支持多任务，即允许多个独立的作业（比如更新）“同时”执行。例如，图 15-1 中的两个单独的更新可以用下面的单个语句来代替（注意逗号分隔符）：

---

⊖ 同时请注意，这个假设对于所有可能正确的状态 *D*1 都成立。诚然，*T* 很可能并不能“如期望的那样反映现实的操作”，但是仍然能够从上述的 *D*1 开始产生一个正确的状态 *D*2，但是这还不够；我们希望保证正确性而不是偶然实现正确性。

```
UPDATE ACC 123 { BALANCE := BALANCE - $100 } ,
UPDATE ACC 456 { BALANCE := BALANCE + $100 } ;
```

现在事务将只包含一个更新操作；这样它对数据库的作用从定义上说就是原子的。至少在这个例子中，BEGIN TRANSACTION、COMMIT 和 ROLLBACK 语句都不再需要。但是，在第 5 章我们已经知道现在的产品还不支持多任务（至少不能完全支持）；出于实际的考虑，在本章接下来的部分中我们不再考虑多任务的可能性。（事实上，甚至在关于事务的初始的理论阶段的讨论中就没有考虑多任务，详细的讨论请参考 16.10 节。）

## 15.3 事务恢复

一个事务以 BEGIN TRANSACTION 语句开始，以 COMMIT 或 ROLLBACK 语句结束。COMMIT 建立了一个**提交点**（commit point）（在一些比较老的数据库产品中也称为**同步点**（syncpoint））。提交点标志着逻辑工作单元的结束，也标志着数据库处于或应该处于一致状态。相反，ROLLBACK 将数据库回滚到 BEGIN TRANSACTION 的状态，实际上就是回滚到前一个提交点。（这里“前一个提交点”的表述是正确的，即使该事务是程序的第一个事务，可将程序的第一个 BEGIN TRANSACTION 默认为初始的“提交点”。）

注意：这里的术语“数据库”实际指事务所存取的那部分数据库，其他事务可能与该事务并行地执行，对自己的那部分数据库进行更新，因此，整个“数据库”在提交点不可能处于完全一致的状态。然而正如在 15.1 节中所述，我们这里不考虑并发事务，这种简化并不影响讨论。

当设置提交点时：

1）正如 15.2 节所说，执行程序中上一个提交点以前的所有更新操作都被提交，即结果被永久保存。在提交点之前所有的更新操作都应看作是不确定（tentative）的，不确定意味着它们接下来可能被撤消（即回滚）。一旦提交，任何一个更新操作都不能被撤消（这就是“提交”的定义）。[⊖]

2）所有的**数据库定位**（database positioning）将消失，所有的元组锁（tuple lock）都将被释放。这里“数据库定位”指在任意时间点，一个执行程序将具有对某些特定元组的寻址能力（如通过第 4 章中 SQL 的游标），在提交点该寻址能力将消失。“元组锁”将在下一章中解释。注意：有些系统提供可选项使程序实际在从一个事务转向另一事务时仍保持对元组的寻址能力（因此也保持元组锁），这类操作已被加入到了 SQL：1999 中。15.8 节将有更进一步的讨论。

注意：上述第 2 点（不包括可能保持一定的寻址能力以及元组锁的说明）对以 ROLLBACK 结束而不是以 COMMIT 结束的事务也适用，而第 1 点显然不适用。

现在我们可以看出，事务不仅是工作单元，而且是**恢复**单元。因为一旦事务成功提交，系统将保证其更新永久作用到数据库上，即使系统在紧接下来的时刻就崩溃了。举个例子，在 COMMIT 刚被确认，而更新没有物理地写入数据库之前，系统是可能崩溃的，这时数据可能仍在主存缓冲区中，[⊖] 并在系统崩溃时全部丢失。即使这种情况发生，系统的重启过程应当将这些更新重新作用到数据库上，通过检查日志中的相关入口点，系统可找到那些已经被成功写入数据库的值。这说明在 COMMIT 过程完成之前，日志必须物理地写出，此即**日志先写原则**（write-ahead log rule）。因此，重启过程将对那些成功提交但在系统崩溃前物理上未更新数据库的事务进行恢复，由此可见事务实际上也是恢复单元。注意：下一章将提到，事务也是**并发**（concurrency）单元。

接下来我们讨论实现上的一些问题。直观看来，如果能够保证下面的两个前提，很明显实现将是最简单的：

- 在事务提交之前，对于数据库的更新保存在内存的缓冲区中并且没有物理写入磁盘。这

---

⊖ 那不是一个显式的用户动作。（实际上，如果事务是可以嵌套的，则一个提交的更新可能会被系统隐式地撤消，但是我们将这种情况留到以后讨论。）

⊖ 缓冲区是内存中的分段运输区域，用来存储将在物理数据库和执行的事务之间交换的数据。

样，如果一个事务不成功地终止，不需要进行撤消磁盘更新的操作。

- 将数据库更新物理地写入到磁盘是事务提交过程的一部分。如果系统随后崩溃，磁盘的更新可以不需要重做。

但是，在实际的实现中，以上的两个前提一般都不能保证。首先，内存的缓冲区可能不是足够大，所以对于某个事务 *A* 的更新可能需要在 *A* 提交之前完成——这可能是为了给另一个事务 *B* 腾出空间（这称为 ***B* 窃取 *A*** 的空间）。其次，在事务提交的时候**强制地**执行物理写操作可能是低效的；例如在 100 个连续事务更新同一个数据库对象的情况下，强制写 100 次，而实际上一次就足够了。正因为以上的这些原因，实际中的事务管理器常采用一种称为窃取/不强制（steal/no-force）的策略，它使实现变得复杂。这方面进一步的内容超出了本书的范围。

前面我们已经提到日志先写原则，它要求日志的物理写必须在 COMMIT 操作完成之前完成。这无疑是正确的，这里给出它的准确描述：

- 在给定的数据库更新物理地写入数据库之前，它对应的日志记录必须物理地写入日志。
- 一个事务的所有其他日志记录都必须在它的 COMMIT 日志记录物理写入日志之前物理地写入日志。
- 只有在一个事务的 COMMIT 日志记录物理地写入日志之后，该事务的 COMMIT 过程才能结束。[⊖]

**ACID 性质**

根据参考文献［15.14］，我们对这一节和上一节的内容作一个小结：事务拥有（被假设拥有）以下的“ACID 特性”：原子性（atomicity）、正确性（correctness）[⊜]、隔离性（isolation）和持久性（durability）。

- **原子性**：事务是原子的，要么都做，要么都不做。
- **正确性**：事务将数据库从一个正确状态转变为另一个正确状态，但在事务内无须保证正确性。
- **隔离性**：事务相互隔离。也就是说，即使通常多个事务并发执行，任一事务的更新操作对其他事务都是不可见的，直到其成功提交。另一种说法为，对任意两个不同的事务 *A* 和 *B*，*A* 可在 *B* 提交后看到其更新，或 *B* 可在 *A* 提交后看到其更新，但是两者不可能同时发生。
- **持久性**：一旦事务成功提交，即使系统崩溃，其对数据库的更新也将永久有效。

关于这些特性，在下一章中将会有大量的讨论。

## 15.4　系统恢复

系统必须不仅能从局部故障（如单个事务中的溢出异常）中恢复，而且能从全局（global）故障（如断电）中恢复。顾名思义，局部故障只影响发生故障的事务，这已在 15.2 节和 15.3 节讨论过了；而全局故障影响正在运行的所有事务，具有系统范围的含义。本节及下一节将对全局故障的恢复进行简单的讨论，全局故障涉及两类：

- **系统故障**（如断电）：影响正在运行的所有事务，但不破坏数据库，有时也称作软故障；
- **介质故障**（如磁盘的磁头碰撞）：将破坏数据库或部分数据库，并影响正存取这部分数据的所有事务，有时也称作硬故障。

发生系统故障时，主存内容，尤其是数据库缓冲区中的内容都会丢失。此时正在运行的事务的状态都不得而知，这样的事务将不可能成功结束，因此必须在系统重启时撤消（undone），即

---

⊖ 有些数据库系统在遇到 COMMIT 日志记录时马上强制将它写入磁盘，另一些系统直到缓冲区满的时候才将其中的日志记录写入磁盘。后一种策略因为事实上造成几个事务同时提交的情况，也被称为组提交（group commit），它减少了总的 I/O 数，但也使得一些事务延迟终止。

⊜ 在以前绝大部分的文献中都是使用一致性（consistency）（例如在参考文献［15.14］中），但是这里我们使用正确性（correctness），因为一致性并不意味着正确性。

回滚。而且，在重启时**重做**（redo）那些在系统崩溃前成功结束但未将更新从缓冲区写入物理数据库的事务也是非常必要的。

那么系统在重启时怎么知道哪些事务该撤消哪些事务该重做呢？在一定的预定时间间隔（通常在预定数量的登记项写入日志时）内，系统自动设置检查点。设置检查点涉及（a）将数据库缓冲区的内容强制写入物理数据库；（b）将一特殊的**检查点记录**（checkpoint record）写入物理日志。检查点记录包含设置检查点时正在运行的事务列表。要理解如何使用该信息，可参看图 15-3，具体说明如下（在图中时间从左向右流动）：

- 系统故障在 *tf* 时发生；
- *tf* 之前最近的检查点在 *tc* 时设置；
- *T*1 类的事务在 *tc* 前成功结束；
- *T*2 类的事务在 *tc* 前开始，在 *tc* 后 *tf* 前成功结束；
- *T*3 类的事务在 *tc* 前开始，但直到 *tf* 时尚未结束；
- *T*4 类的事务在 *tc* 后开始，在 *tf* 前成功结束；
- *T*5 类的事务在 *tc* 后开始，但直到 *tf* 时尚未结束。

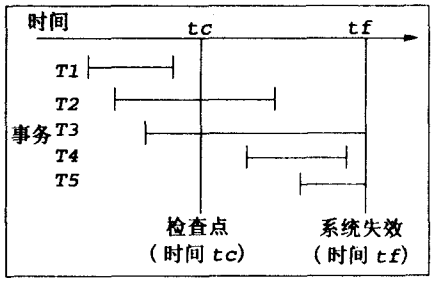



图 15-3 五类事务

显然系统重启时，*T*3 和 *T*5 类的事务必须撤消，*T*2 和 *T*4 类的事务必须重做。但重启过程并不涉及 *T*1 类的事务，因为它们的更新在 *tc* 设置检查点时已强制写入数据库中。对于那些在 *tf* 之前未成功完成（即已 ROLLBACK）的事务，重启过程也不涉及（为什么?）。

因此，在重启时系统首先按照如下的步骤标识 *T*2 ~ *T*5 类的事务：

1）首先设置两个事务列表：UNDO 列表和 REDO 列表。

2）将 UNDO 列表设置为最近一个检查点记录所包含的所有事务的列表，REDO 列表设置为空。

3）从检查点记录开始，对日志进行正向扫描。

4）如果遇到事务 *T* 的 BEGIN TRANSACTION 日志登记项，则将 *T* 加入 UNDO 列表。

5）如果遇到事务 *T* 的 COMMIT 日志登记项，则将 *T* 从 UNDO 列表移到 REDO 列表。

6）当日志扫描结束时，UNDO 列表和 REDO 列表分别标识了 *T*3 和 *T*5 类的事务以及 *T*2 和 *T*4 类的事务。

现在系统将反向扫描日志，撤消 UNDO 列表中的事务；接着再正向扫描，重做 REDO 列表中的事务。注意：通过撤消操作将数据库恢复到一致性状态有时称作**反向恢复**（backward recovery）；通过重做操作将数据库恢复到一致性状态有时称作**正向恢复**（forward recovery）。

最后，当所有的恢复工作都完成时，系统就可以接受新的处理请求了。

### ARIES

前面描述的系统恢复过程是经过了很大简化的。[1] 比如它说撤消操作总是在重做操作之前完成。早期的系统确实是这样实现的，但是出于效率的考虑，现在流行的系统一般不再使用这种策略。事实上，今天绝大多数系统都使用一种叫做 ARIES[2] [15.20] 的算法或者和它非常近似的某种策略，而其中恰恰是先执行重做操作。ARIES 算法的运行可以分成三个主要的阶段：

1）分析：设置两个事务列表：UNDO 列表和 REDO 列表。

2）重做：从在分析阶段确定的某个特定的位置开始，将数据库恢复到它崩溃时的状态。

3）撤消：撤消那些提交失败的事务对数据库的更新。

注意：这里的“重做先于撤消”意味着那些提交失败的事务也先被重做，然后再被撤消。

---

[1] 在其他问题的讨论中，我们总是假设恢复是可行的！不并发执行的事务显然是可恢复的；但是，并发执行引入了某些复杂的因素，我们必须仔细设计使它们不破坏可恢复性。在下一章我们将继续讨论这个问题。

[2] ARIES 是“Algorithms for Recovery and Isolation Exploiting Semantics”的简写。

因此 ARIES 的重做过程常被称为再现历史（repeating history）[15.21]。ARIES 在撤消阶段的执行中会书写日志，如果系统在重启过程（restart procedure）中再次崩溃（尽管几乎不可能发生），那么在下一次重启时这次已经完成的撤消动作将不再重复执行。

## 15.5　介质恢复

注意：介质故障恢复与事务故障和系统故障的恢复有些不同，介绍它是为了完备恢复的内容。

重复 15.4 节的内容，介质故障将破坏部分数据库，如磁盘的磁头碰撞，磁盘控制器故障。介质故障恢复需利用转储的后备副本重载数据库，然后利用日志（联机和脱机日志）对转储后备副本以来完成的所有事务进行重做处理。没有必要对发生故障时正在运行的事务进行撤消处理，因为这些事务的所有更新已“撤消”（实际上是丢失）了。

要能执行介质故障恢复，系统必须具有转储/重建（dump/restore）或卸载/重载（unload/reload）实用例程。转储即制作后备副本，这些副本可保存在磁带或其他用于存档的介质上，不必保存在直接存取介质上。重建即从指定的后备副本重构数据库。

## 15.6　两阶段提交

注意：可能你第一次阅读这一节时希望略过它。

**两阶段提交**（two-phase commit）是提交/回滚概念的又一重要内容，当一个事务与几个独立的“资源管理器”（每个管理器管理自己的可恢复资源集合，并维护自己的恢复日志[⊖]）相互作用时，两阶段提交尤为重要。举个例子，若一事务运行在 IBM 大型机上，该事务对 IMS 数据库和 DB2 数据库进行更新（这样的事务是合法的）。如果该事务成功结束，其对 IMS 和 DB2 数据的所有更新操作都必须被提交；如果失败，所有的更新操作必须回滚。换句话说，不可能出现对 IMS 数据的更新操作被提交同时对 DB2 数据的更新操作被回滚的情形，反之亦然，否则事务就不是原子的（全部做或者全部不做）。

由此可见对该事务，系统要求 IMS 执行 COMMIT 而要求 DB2 执行 ROLLBACK 是没有意义的，即使系统向两者发出相同的指示，系统也可能在两者 COMMIT 间或 ROLLBACK 间崩溃，造成不可预料的结果。因此，事务需发出一个**系统范围**（system-wide）的 COMMIT（或 ROLLBACK）。该“全局”的 COMMIT 或 ROLLBACK 由一称作**协调者**（coordinator）的系统部件控制，协调者保证两个资源管理器（即例中的 IMS 和 DB2）对它们各自的更新操作所做的提交或回滚是一致的，即使系统在事务运行过程中发生了故障。正是两阶段提交协议使协调者提供了这样的保证。

下面具体介绍两阶段提交协议的工作原理。假设事务已成功完成数据处理过程，它将发出系统范围的 COMMIT 指示，协调者在收到 COMMIT 请求后将进入如下两个阶段的处理过程：

- 准备：首先，协调者指示所有的资源管理器做好准备，这意味着每个参与者，即资源管理器，必须将事务对本地资源的所有操作的日志登记选项强制写到物理日志（即写入稳定的存储设备；这样，资源管理器将具有其代表事务在本地工作的永久记录，在必要时可用于提交和回滚）。假设已成功地强制写出日志，资源管理器将向协调者发出“准备好”的响应，否则发出“未准备好”的响应。
- 提交：当协调者收到来自所有参与者的响应时，它将在自己的日志中登记其关于事务的决定。如果所有的响应都是“准备好”，其决定就是“提交”；如果有一个响应是“未准备好”，其决定就是“回滚”。接着协调者将向所有的参与者通知这个决定，每个参与者将根据指示对事务进行本地的提交或回滚。注意：每个参与者必须在第二阶段完成协调者的指示，这就是协议；正是协调者日志中的事务决定的记录项标识了从阶段 1 到阶段 2 的转变。

---

⊖ 在分布式的数据库环境中，这一点尤其重要，其详细地原因将在第 21 章讨论。

如果系统在整个处理过程中出现了故障，重启过程将在协调者日志中查找事务决定的记录项。如果找到该记录，两阶段提交过程将从其被中止的那一点继续执行；如果未找到，系统将假设事务决定是“回滚”，并相应地完成回滚。注意：如果协调者和参与者运行在不同的计算机（如分布式系统）上，则协调者发生故障时，参与者可能需等待很长一段时间才能收到协调者的决定。只要它一直在等待，则事务由参与者执行的那部分更新操作对其他事务是不可见的，即将被加锁（见第16章的讨论）。

数据通信管理器（DC管理器——参看第2章）也被看做资源管理器，这就是说，消息就像数据库一样也被看做可恢复的资源，DC管理器需能参与两阶段提交过程。有关两阶段提交的更权威的论述可参看参考文献［15.12］。

## 15.7 保存点

我们已经知道在通常情况下一个事务是不能被其他事务嵌套的；但是一个事务是否可以分成多个子事务呢？在一定的限定条件下，回答是肯定的。在一个事务执行的过程中建立中间的**保存点**是可能的；随后，根据需要，事务可以回滚到先前建立的某个保存点，而不需要回滚到最开始的状态。事实上，在一些早期的系统，包括Ingres（商业产品，并非原型系统）和System R中已经包含保存点机制和相应的语句；这种语句后来被加入到SQL99标准中。需要注意的是，建立一个保存点和执行一个COMMIT是不同的；在事务成功地执行COMMIT之前，事务对数据库的更新对于其他的事务一直是不可见的。在接下来的15.8节中将会有更深入的讨论。

## 15.8 SQL对事务的支持

SQL对事务的支持以及对基于事务的恢复的支持都遵循前面的概念。首先，绝大部分SQL语句是保证原子的执行的（唯一的例外是CALL和RETURN）。其次，正如第4章中所说，SQL对BEGIN TRANSACTION、COMMIT和ROLLBACK都有直接的支持语句，它们分别是START TRANSACTION、COMMIT WORK和ROLLBACK WORK。START TRANSACTION的语法如下：

```
START TRANSACTION <option commalist> ;
```

其中<*option commalist*>表示存取模式或者隔离级别，或者两者都有（一个和诊断区域大小相关的第三方的选项，超出了本书的范围）：

- **存取模式**可以是READ ONLY或者READ WRITE。除非指定隔离级别是READ UNCOMMITTED，默认的存取模式是READ WRITE。而如果指定了存取模式READ WRITE，那么隔离级别就不能是READ UNCOMMITTED。
- 语法ISOLATION LEVEL <*isolation*>定义了**隔离级别**，选项<*isolation*>可以是READ UNCOMMITTED，READ COMMITTED，REPEATABLE READ或者SERIALIZABLE。更详细的在第16章介绍。

COMMIT和ROLLBACK的语句如下：

```
COMMIT [ WORK ] [ AND [ NO ] CHAIN ] ;
ROLLBACK [ WORK ] [ AND [ NO ] CHAIN ] ;
```

WORK关键字在这里只是噪音，并没有任何作用。AND CHAIN选项表示在COMMIT之后自动执行一个前一个相同的<*option commalist*>选项的START TRANSACTION；AND NO CHAIN是默认的选项。COMMIT和ROLLBACK语句的执行自动使每个打开的游标CLOSE（这就引起了所有数据库定位的丢失），唯一的例外是COMMIT不CLOSE在游标声明时使用WITH HOLD选项的游标。注意：这一点在第4章没有涉及，WITH HOLD是游标声明时的一个选项，声明为WITH HOLD的游标在事务COMMIT时继续保持其打开和定位状态，这样下一个FETCH操作将按顺序将游标指向下一个元组；因此，原先在下一个OPEN时所需要的复杂的重定位代码就不再需要了，这样就避免了否则重新打开并定位的繁复的代码。

SQL还支持保存点。语句是：

```
SAVEPOINT <savepoint name> ;
```

它根据用户指定的名字（被事务视为本地的）创建一个保存点。而语句

```
ROLLBACK TO <savepoint name> ;
```

撤消从指定保存点开始的所有更新。语句

```
RELEASE <savepoint name> ;
```

删除指定的保存点，表示以后将不再允许回滚到该保存点。当事务终止时，所有的保存点都被自动的删除。

## 15.9　小结

本章对**事务管理**做了简单的介绍，事务是**逻辑工作单元**，也是**恢复单元**（同时也是并发单元，参见第16章）。事务具有**ACID性质**：**原子性**、**正确性**（更经常被称为一致性）、**隔离性**和**持久性**。**事务管理**监控事务的执行，以保证事务具有上述的重要性质。事实上系统的目的就是保证**事务的可靠执行**。

事务以BEGIN TRANSACTION开始，以COMMIT或ROLLBACK结束。COMMIT建立了一个**提交点**（更新永久驻留），ROLLBACK将数据库回滚到上一**提交点**（更新被撤消）。如果事务未到达预期的终点，系统将强制ROLLBACK（**事务恢复**）。为了能撤消（或重做）对数据库的更新操作，系统必须维护恢复**日志**，而且事务的日志记录必须在该事务的COMMIT操作完成之前物理地写回到日志（**日志先写原则**）。

系统还必须保证崩溃时事务的ACID性质，因此系统必须（a）**重做**崩溃前成功完成的事务的所有工作；（b）**撤消**崩溃前已启动但仍未成功完成的事务的所有工作。**系统恢复**作为系统**重启**过程的一部分（有时叫做重启/恢复过程），通过检查最近设置的**检查点记录**决定需重做和撤消的事务，**检查点记录**在一定的预定的时间间隔写入日志。

系统还提供**介质恢复**，通过先从转储的后备**副本**恢复数据库，再利用日志重做转储以来所做的工作。系统要支持介质恢复必须具备转储/重建实用**例程**。

系统如果要使事务与多个不同的**资源管理器**（如两个不同的DBMS，或一个DBMS和一个DC管理器）进行交互，同时又要维护事务的原子性，就必须使用**两阶段提交协议**。这两个阶段是：（a）**准备**阶段，**协调者**指示所有的**参与者**做好提交准备；（b）**提交**阶段，在收到所有参与者的响应后，协调者作出最终决定并指示参与者提交（或回滚）。

本章还包含了对**保存点**的一个简要介绍和SQL对恢复支持的综述；特别是，SQL提供显式的**START TRANSACTION**语句，在这个语句中用户可指定事务的**存取模式**和**隔离级别**。

最后，本章的讨论都默认在应用程序环境下，但是所有的概念也适用于最终用户环境。举个例子，SQL产品通常允许用户从终端交互地输入SQL语句，每个交互式语句将作为一个事务，系统在执行了该SQL语句后，将自动代表用户发出COMMIT命令。但是，有些系统允许用户禁止自动COMMIT功能，这样就可执行一系列的SQL语句（以显式的COMMIT和ROLLBACK结束）作为一个单一事务。这种方式可能使数据库的一部分被锁住，其他用户将无法访问，而且可能造成死锁，这些将在第16章讨论。

## 习题

15.1　系统不允许事务只提交对某些数据库或变量所做的更新，而不同时提交对其他数据库或变量所做的更新。为什么？

15.2　事务不能嵌套。为什么？

15.3　说明日志先写规则的内容，为什么该规则是必要的？

15.4　在下列情况下，恢复的含义分别是什么？

a. COMMIT时将缓冲区的内容强制写入数据库。

b. 在COMMIT之前缓冲区的内容不物理地写入数据库。

15.5 说明一下两阶段提交协议的内容，并讨论在每个阶段（a）协调者和（b）参与者分别发生故障时的含义。

15.6 利用供应商和零件数据库，写一个 SQL 应用程序，按零件编号顺序查询并显示所有的零件，每十个记录重新开始一个新的事务，并且要求将第十个记录删除。假设从零件表到供应表的 DELETE 外码规则为 CASCADE（也就是说，该练习中可忽略对供应表的操作）。注：可使用 SQL 游标机制。

## 参考文献

[15.1] Philip A. Bernstein:"Transaction Processing Monitors," *CACM 33*, No. 11(November 1990).

我们从本章的内容知道，*TP monitor* 是事务管理器的另一种称呼。这篇文章给出了一个很好的对 TP 监控器的结构以及功能的非正式的介绍。引用："TP 系统是一产品集，包括像处理器、内存、磁盘、通信控制器这样的硬件以及像操作系统、数据库管理系统、计算机网络、TP 监控器这样的软件。这些产品的许多集成工作是由 TP 监控器承担的。"

[15.2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Reading, Mass.:Addison-Wesley(1987).

这是一本教科书，正如其书名，其内容不仅包括了恢复而且包括了整个事务管理，比本章讲解得更透彻。

[15.3] A. Bilris *et al.*:"ASSET: A System for Supporting Extended Transactions," Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, Minn. (May 1994).

本章以及下一章的基本的事务概念对于新的应用来说太严格了，尤其是对交互程度相当高的应用，因此提出许多"扩展事务模型"来表示这样的应用（参看参考文献［15.16］）。但是，在本书写作的时候，没有一个模型明显地优于其他的任何一个模型，因此数据库厂商不愿意在其产品中纳入任何一个模型。

ASSET 独树一帜，其并未提出新的事务模型，而是提供了一个原语操作集合，包括通常的 COMMIT 等操作以及一些新的特殊操作，用以"对应用定制专用事务模型"。这篇论文还说明了 ASSET 是如何指定"嵌套事务、分割事务、长事务以及其他扩展事务模型"。

[15.4] L. A. Bjork:"Recovery Scenario for a DB/DC System," Proc. ACM National Conf., Atlanta, Ga. (August 1973).

这篇论文以及其姊妹篇［15.7］很有可能是最早对恢复进行理论探讨的文章。

[15.5] R. A. Crus:"Data Recovery in IBM DATABASE 2," *IBM Sys. J. 23*, No. 2(1984).

详细描述了 DB2 的恢复机制，是对恢复技术进行总体介绍的一篇很不错的文章。尤其是，这篇文章介绍了 DB2 在有些事务处于恢复阶段时发生系统故障后的恢复处理过程，要保证将正在回滚的事务所做的未提交的更新撤消需要特别考虑（在某种意义上，这类问题是丢失更新的反问题，见第 16 章）。

[15.6] C. J. Date:"Distributed Database: A Closer Look," in C. J. Date and Hugh Darwen. *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley(1992).

本章的 15.6 节已描述了基本的两阶段协议，基于基本的协议仍可进行改进。例如，如果参与者 *P* 在第一阶段响应协调者 *C*，通知 *C* 其在本事务中并无更新操作（即只读），则 *C* 在第二阶段将忽略 *P*。更进一步，如果所有的参与者都在第一阶段响应内容皆为只读，则第二阶段将整个忽略（更进一步的讨论请见第 21 章）。

还有其他的可能改进方案，论文中涉及了一些，主要包括：假设提交和假设回滚协议（基本协议的改进版）；过程树模型（当参与者在事务的某一部分充当协调者时）；在参与者对协调者的确认过程中通信故障发生时的情形。注：虽然这样的讨论一般都基于分布式系统环境，大部分概念实际上适用于广泛的应用环境，可参看第 21 章，关于这一话题的进一步讨论，请参见参考文献［21.13］。

[15.7] C. T. Davies, Jr.:"Recovery Semantics for a DB/DC System." Proc. ACM National Conf., Atlanta, Ga. (August 1973).

参看参考文献［15.4］的注释。

[15.8] C. T. Davies, Jr.:"Data Processing Spheres of Control," *IBM Sys. J. 17*, No. 2(1978).

控制范围是对事务管理规则的最初研究和形式化探讨，是对外界视为原子的工作的抽象。不像现今大多数系统所支持的事务概念，控制范围可彼此嵌套，并可嵌套到任意深度。

[15.9] Hector Garcia-Molina and Kenneth Salem: "Sagas," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

本章所讨论的事务都默认为在很短的时间内执行完毕（毫秒甚至微秒）。如果一个事务持续较长的时间（小时、天或周），那么（a）如果其必须回滚，则需撤消大量的工作；（b）即使其最终成功结束，在相当长的时间内其将占用大量的系统资源（如数据库数据等），这样将阻止其他用户的使用（参看第16章）。不幸的是，许多"现实世界"的事务持续时间都比较长，尤其在像硬件和软件工程这样的新的应用领域。**Sagas** 是对上述问题的解决方案，saga 是一个短事务（通常意义上）序列，并且系统保证要么（a）该序列中的所有事务都成功执行，要么（b）执行特定的补偿事务［15.17］消除在整个未完成的 saga 中成功执行了的事务的影响，使得 saga 就像从未执行过一样。举个例子，在银行系统可能存在这样的事务，"向账户 A 增加 100 美元"，补偿事务显然为"从账户 A 中减去 100 美元"。对 COMMIT 语句的扩展允许用户在必须消除已完成的事务的影响的情况下通知系统运行补偿事务。注：理想地认为补偿事务从不回滚。

[15.10] James Gray: "Notes on Data Base Operating Systems," in R. Bayer, R. M. Graham, and G. Seegmuller (eds.), *Operating Systems: An Advanced Course* (Springer-Verlag *Lecture Notes in Computer Science 60*). New York, N. Y.: Springer-Verlag (1978). Also available as IBM Research Report RJ 2188 (February 1978).

这是最早的也必定是最易获得的有关事务管理的资料。其对两阶段提交协议最早进行了概要描述，显然其讨论不及最新的参考文献［15.12］那样深入广泛，但仍推荐给读者。

[15.11] Jim Gray: "The Transaction Concept: Virtues and Limitations," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

简明扼要地讲述了事务相关的概念、问题，包括大量的实现。

[15.12] Jim Gray and Andreas Reuter: *Transaction Processing: Concepts and Techniques.* San Mateo, Calif.: Morgan Kaufmann (1993).

这部著作堪称计算机科学著作中的经典。虽然读来较为费时，但作者对各主题阐述得清晰明确，即使最枯燥的部分也令人爱不释手。在序言中，作者表明他们的目的是"帮助解决实际问题"，这本书"切合实际，详细描述了各种基本的事务情形"，并且包含"大量的代码段、基本算法以及数据结构"，但并不是"一本百科全书，面面俱到"。尽管正如其最后声明的，这本书并不是包罗万象，但仍是一部堪称标准的著作。强力推荐。

[15.13] Jim Gray *et al.*: "The Recovery Manager of the System R Data Manager," *ACM Comp. Surv. 13*, No. 2 (June 1981).

参考文献［15.13］和［15.19］都与 System R（是数据库领域的某些方面的先锋）的恢复机制相关。参考文献［15.13］给出了整个恢复子系统的全貌；参考文献［15.19］详细描述了一种特别的处理机制——影子页机制。

[15.14] Theo Härder and Andreas Reuter: "Principles of Transaction-Oriented Database Recovery," *ACM Comp. Surv. 15*, No. 4 (December 1983).

ACID 特性最早在这篇文章中提出，文章对恢复原理作了仔细的阐述，清晰易懂，同时还给出了用统一方式描述大量的恢复模式以及日志技术的术语框架，并依据这一框架对已存在的大量系统进行了分类和描述。

文章还包括一些有趣的经验性数据，包括在一个典型的大系统中三类故障（局部、系统、介质）的发生频率和可接受的恢复次数。如下表所示：

| 故障类型 | 发生频率 | 恢复时间 |
|---|---|---|
| 局部 | 10～100 次/min | 与事务执行时间相同 |
| 系统 | 每周几次 | 几分钟 |
| 介质 | 每年 1～2 次 | 1～2 小时 |

[15.15] Theo Härder and Kurt Rothermel: "Concepts for Transaction Recovery in Nested Transactions," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

它提出了这样一种方案：事务可以包含低层（lower-level）的子事务（子事务同样可以包含更低层的子事务，如此等等）。一个不是任何事务的子事务的事务被称为顶层事务，顶层事务满足 ACID 特性；但是，一个子事务（非顶层事务）则只要求满足原子性和隔离性。更详细

的讨论在第 16 章的 16.10 节。

[15.16] Henry F. Korth:"The Double Life of the Transaction Abstraction: Fundamental Principle and Evolving System Concept"(invited talk), Proc. 21st Int. Conf. on Very Large Data Bases, Zurich, Switzerland (September 1995).

简单概要地描述了为支持新的应用需求事务概念的新发展。

[15.17] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz:"A Formal Approach to Recovery by Compensating Transactions,"Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

形式化地定义了**补偿事务**的概念，用于 sagas [15.9] 以及其他类似方法中的已提交事务以及未提交的事务的撤消。

[15.18] David Lomet and Mark R. Tuttle:"Redo Recovery After System Crashes,"Proc. 21st Int. Conf. on Very Large Data Bases, Zurich, Switzerland (September 1995).

对重做恢复（即正向恢复）进行了精确、仔细的分析。"虽然重做恢复仅为恢复的一种形式，但其作为整个恢复过程的重要组成部分，必须处理最困难的问题，因而是非常重要的。"（注意，与本章 15.4 节中给出的算法相对照，[15.19] 中的 ARIES "建议将恢复过程理解为先做重做恢复，后做撤消恢复"）。作者认为他们的分析将有助于更好地理解已存在的实现方式以及提出对恢复系统的改进方案。

[15.19] Raymond A. Lorie:"Physical Integrity in a Large Segmented Database,"*ACM TODS 2*, No. 1 (March 1977).

正如参考文献 [15.13] 中所说的，这篇论文描述了 System R 恢复子系统的**影子页**机制（注：顺便提一下，术语"集成"与第 8 章中的完整性概念没有任何关系）。基本思想很简单：当一（未提交的）更新第一次写入数据库时，系统并不替换已存在的那一页，而是在磁盘的其他地方存储新的一页。旧页即为新页的"影子"。提交更新必须更新指针的指向，使其指向新的一页，并废除影子页；回滚更新必须将影子页复位，并废除新页。

虽然概念很简单，但影子页机制有一严重的缺陷，就是破坏了数据原有的物理聚集。因此该机制并未应用于 DB2 [15.5]，虽然其曾应用于 SQL/DS [4.14] 中。

[15.20] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwartz:"ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write Ahead Logging,"*ACM TODS 17*, No. 1 (March 1992).

关于 ARIES 的第一篇论文。当这篇论文发表的时候，ARIES 已经在几个商业和实验系统中得以实现，只是程度不同而已，其中包括 DB2。以下引自论文："事务管理问题的解决方案可用几个指标进行判断：对一页以及跨页并发支持的程度；结果逻辑的复杂性；用于存储数据和日志的非挥发性存储介质和内存的空间负载；在重启/恢复和正常处理阶段所需的同步和异步 I/O 的次数；支持的功能类型（如部分事务回滚，等等）；重启/恢复阶段处理的项数；重启/恢复阶段并发处理的程度；死锁引起的事务回滚的范围；对存储数据的限制（如要求所有的记录有唯一码，限制对象的最大数目为页面大小，等等）；对新的锁模式的支持能力，这种锁模式允许基于交换及其他特性并发执行不同事务对同一数据所做的增加/减少操作；等等。[ARIES] 对这些指标处理得不错。"㊀

自从 ARIES 最先提出后，出现了大量的改进版和专用版：ARIES/CSA（用于客户/服务器系统）、ARIES/IM（用于索引管理）、ARIES/KVL（索引关键字死锁）、ARIES/NT（用于嵌套事务），等等。

[15.21] C. Mohan:"Repeating History Beyond ARIES,"Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland (September 1999).

㊀ 对原文稍作修改。——译者注

# 第16章 并 发

## 16.1 引言

DBMS 通常允许多个事务同时存取同一数据库，这就是**并发**。在这样的系统中，必须提供某种并发控制机制以确保并发事务间互不干扰。（在 16.2 节将给出在缺乏必要的控制情况下发生的一些事务间相互干扰的例子。）本章将对并发问题展开深入的讨论，结构如下：

- 刚刚已经提到，16.2 节说明若不进行适当的并发控制将会出现什么样的问题。
- 16.3 节介绍处理上述问题的一种传统的方法，即锁。（锁并不是解决并发控制问题的唯一方法，但却是实际运用最广泛的方法。在本章最后的参考文献中，将介绍几种其他的方法。）
- 16.4 节解释了锁机制如何用于解决 16.2 节中描述的问题。
- 锁自身会引发的一些问题，其中最著名的就是死锁。16.5 节将针对此问题展开讨论。
- 16.6 节描述了可串行性的概念，其通常作为并发事务执行正确性的准则。
- 16.7 节讨论并发对于上一章讨论的话题恢复的影响。
- 16.8 节和 16.9 节对锁的基本思想作了进一步的讨论，即隔离级别和意向锁。
- 16.10 节对事务的 ACID 特性提出了几点新颖的、怀疑性的意见。
- 16.11 节给出了 SQL 的相关表示。
- 16.12 节是小结。

引言最后给出两点说明：第一，和恢复一样，并发的思想与底层系统是否为关系型的无关（尽管和恢复一样，在很大程度上，绝大部分的理论工作都是在关系数据库的领域中来讨论的[16.6]）。第二，并发和恢复一样都是很大的课题，在本章我们只能介绍一些关于并发的最重要和最简单的思想。一些特定的高级特性，在本章后面的一些参考文献中有简要的描述。

## 16.2 三个并发问题

首先考虑任何一个并发控制机制必须提及的一些问题：在三种情况下必定将发生错误，即在这三种情况下，一个事务即使依照第 15 章的讨论能保证自身的正确性，但是如果其他的某个事务以某种方式干扰此事务的运行，也可能引发错误。注意（依照先前的假设）干扰事务本身也可能是正确的。正是未对两个正确运行事务的交错在一起的操作进行控制产生了全局不一致的结果。这三个问题是：

- 丢失更新问题
- 未提交依赖问题
- 不一致分析问题

下面将依次进行讨论。

**1. 丢失更新问题**

考虑图 16-1 的情形。图例的意思是：事务 *A* 在时间 *t1* 检索元组 *t*；事务 *B* 在时间 *t2* 检索同一元组 *t*；事务 *A* 在时间 *t3* 更新元组 *t*（基于时间 *t1* 所看到的值）；事务 *B* 在时间 *t4* 更新元组 *t*（基于时间 *t2* 所看到的值，与 *t1* 时间的值相同）。事务 *A* 的更新在 *t4* 时间丢失，因为事务 *B* 甚至都没看它就将其覆盖了。

| 事务 *A* | 时间 | 事务 *B* |
|---|---|---|
| — | | — |
| — | | — |
| RETRIEVE t | *t1* | — |
| — | | — |
| — | *t2* | RETRIEVE t |
| — | | — |
| UPDATE t | *t3* | — |
| — | | — |
| — | *t4* | UPDATE t |
| — | ↓ | — |

图 16-1 事务 *A* 在 *t4* 时间丢失更新

**2. 未提交依赖问题**

如果一个事务允许检索，甚至允许更新一个已被其他事务更新但未提交的元组，这将引起未提交依赖问题。因为如果

事务尚未提交，就有可能不再提交它而是回滚，这种情况下，第一个事务所看到的数据当前并不存在，某种意义上，从未存在过。参看图 16-2 和图 16-3。

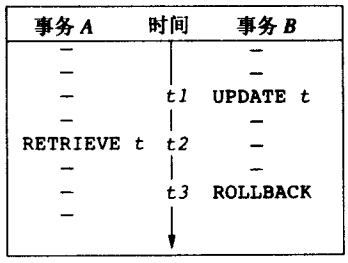



图 16-2 事务 A 在时间 t2 依赖未提交的变化

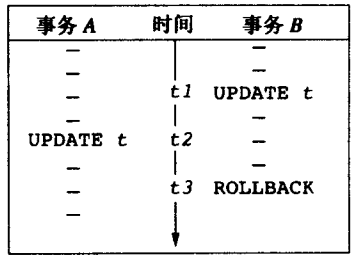



图 16-3 事务 A 在时间 t2 更新未提交的变化，并在时间 t3 丢失了该更新

在第一个例子（图 16-2）中，事务 A 在时间 t2 看到一个未提交的更新（也称作未提交的变化），该更新在时间 t3 被撤消。事务 A 的操作基于错误的假设，即假设元组 t 具有时间 t2 时看到的值，但实际上为时间 t1 前的值，结果事务 A 产生了不正确的输出结果。注意，事务 B 的回滚可能并不是 B 本身的故障造成的，举个例子，可能是系统故障的结果，此时事务 A 可能已终止，该故障并不引起 A 的回滚。

第二个例子（图 16-3）情况更糟。事务 A 不仅依赖时间 t2 未提交的变化，而且实际上在时间 t3 还丢失了更新，因为时间 t3 时的回滚使得元组 t 恢复为时间 t1 以前的值。这是丢失更新问题的又一形式。

**3. 不一致分析问题**

图 16-4 表示了事务 A 和 B 对账户（ACC）元组的操作：事务 A 对账户余额进行求和，事务 B 将账户 3 的金额 10 转到账户 1 上。A 产生的结果为 110，显然不正确。如果 A 将该结果写回到数据库，实际上将使数据库处于不一致的状态。[⊖] 事务 A 看到了数据库的不一致的状态，并因此进行了不一致的分析。注意该例与前例的不同：这里不存在 A 依赖未提交的更新问题，因为 B 在 A 看到 ACC 3 前已提交了所有更新。注意：顺便提一下，这里不一致分析应该是不正确分析，由于历史原因，我们仍然保留以前的说法。

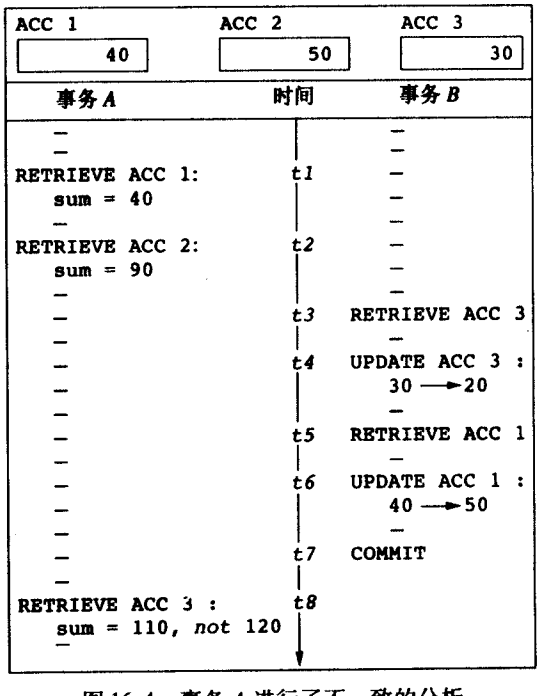



图 16-4 事务 A 进行了不一致的分析

**4. 综合分析**

注意：在你第一次阅读时，可以略过这一小节。

深入分析前面描述过的问题。如果从并发的角度来看，很明显，对于数据库的主要操作有两种：检索和更新；换言之，我们可以将事务看成是一个仅由这两种操作组成的序列（当然，除去必要的 BEGIN TRANSACTION 和 COMMIT 或 ROLLBACK 不考虑）。我们可以简单地把两种操作分别对应为读和写。对于两个并发事务 A 和 B，只有当它们

⊖ 如果考虑可能性（例如将结果写回数据库），还需要假设数据库没有完整性约束阻止这个写操作。

同时试图读或者写同一个数据库对象（如元组 $t$）时才可能发生问题。这时有四种可能的情况：

- **RR**：$A$ 和 $B$ 都读 $t$。读不互相干扰，所以这种情况下没有问题发生。
- **RW**：$A$ 读 $t$ 然后 $B$ 试图写 $t$。如果 $B$ 被允许执行写操作，将会产生不一致分析的问题（如图 16-4 所示）；因此，我们可以说是**读写冲突**（RW Conflict）引起不一致分析。注意：如果 $B$ 被允许写，而后 $A$ 再次读 $t$，它将发现两次读到 $t$ 的内容不一样，这种情况称为**不可重复读**（nonrepeatable read）；不可重复读同样是由于读写冲突引起的。
- **WR**：$A$ 写 $t$ 然后 $B$ 试图读 $t$。如果 $B$ 被允许执行读操作，那么（正如图 16-2 所示，除了需要对换事务 $A$ 和 $B$ 的角色）将有未提交依赖问题；因此我们可以说未提交依赖是由**写读冲突**（WR Conflict）引起的。注意：如果 $B$ 被允许读 $t$，则这时它的读操作称为**脏读**（dirty read）。
- **WW**：$A$ 写 $t$ 然后 $B$ 试图写 $t$。如果 $B$ 被允许执行写操作，那么（可以参考图 16-1 和图 16-3）将产生丢失更新的问题；因此，我们说丢失更新是由写写冲突（WW Conflict）引起的。注意：这时 $B$ 被允许执行的写操作称为脏写（dirty write）。

## 16.3　锁

正如 16.1 节中指出的，可采用**锁**这一并发控制技术解决 16.2 节的所有问题。基本思想很简单：当一个事务需要确保其感兴趣的对象（通常是一个数据库元组）在其完成时不会发生某种形式的改变，它必须**获得**该对象上的一个**锁**。锁的作用就是锁住对象使得其他事务无法访问，尤其是阻止其他事务改变该对象。因此第一个事务就能知道该对象将保持稳定的状态，并顺利执行。

下面将对锁的工作方式做更详细的解释：

1）首先假定系统支持两种锁，**排它锁**（exclusive lock，简记为 X 锁）和**共享锁**（shared lock，简记为 S 锁），定义将在下面两个段落给出。注意：X 锁和 S 锁有时也分别称作写锁和读锁。在进一步讨论之前，假设 X 锁和 S 锁是仅有的类型，在 16.9 节还有其他类型的锁的例子；同时假设元组是仅有的可锁的对象，在 16.9 节还给出了其他类型的可锁的对象。

2）如果事务 $A$ 在元组 $t$ 上具有排它锁（X），则来自某个不同事务 $B$ 对 $t$ 的任何一种类型的锁的请求将被拒绝。

3）如果事务 $A$ 在元组 $t$ 上具有共享锁（S），则：

- 来自某个不同事务 $B$ 对 $t$ 的 X 锁的请求将被拒绝；
- 来自某个不同事务 $B$ 对 $t$ 的 S 锁的请求将被满足，因为 $B$ 也将具有 $t$ 上的 S 锁。

| | X | S | - |
|---|---|---|---|
| X | N | N | Y |
| S | N | Y | Y |
| - | Y | Y | Y |

图 16-5　锁类型 X 和 S 的相容矩阵

这些规则可用**锁类型相容矩阵**表示，见图 16-5。该矩阵解释如下：考虑某个元组 $t$；假定事务 $A$ 当前具有 $t$ 上的锁，用左边列表示（破折号 = 没有锁）；假定某个不同的事务 $B$ 发出了对 $t$ 的锁请求，用最上面的一行表示（为完整起见，仍包括“没有锁”的情况）。“N”表示**冲突**，即 $B$ 的请求不能被满足，将处于等待状态；“Y”表示**相容**，即 $B$ 的请求被满足。矩阵显然是对称的。

下面介绍**数据存取协议**（或称为**锁协议**），其运用 X 锁和 S 锁保证了 16.2 节描述的问题不会发生：

1）事务在检索元组前需获得该元组上的 S 锁。

2）事务在更新元组前需获得该元组上的 X 锁。相应地，如果其已经具有了该元组上的 S 锁（因为通常是 RETRIEVE-UPDATE 的顺序），必须将其从 S 锁升级到 X 锁。

注意：这里需要说明的是：事务对元组的锁请求通常都是隐式的，“检索元组”的请求通常隐含着对相关元组的 S 锁请求，“更新元组”的请求通常隐含着对相关元组的 X 锁请求。（或者隐含着从 S 锁到 X 锁的升级请求。）当然，术语“更新”包括 INSERT、DELETE 以及 UPDATE，但是对 INSERT 和 DELETE 操作，规则需进一步的提炼，这里略去了细节。

3）如果事务 $B$ 的锁请求因为与事务 $A$ 已具有的锁冲突而被拒绝，事务 $B$ 将处于**等待状态**，直到它要求的锁被满足——最早也要等到 $A$ 的锁被释放。我们说“最早”是因为当 $A$ 的锁释放

时，另一个相关元组上的锁请求将被满足，但那不一定是 *B* 的——因为可能有很多事务在等待。注意：系统必须保证 *B* 不会永远处于等待状态（这是有可能出现的，通常称作**活锁**（livelock）或**饿死**（starvation））。避免活锁的简单方法是采用先来先服务的策略。

4）X 锁将一直保持到事务结束（COMMIT 或 ROLLBACK）。S 锁通常也保持到那时，例外情形可参看 16.8 节。

前述的协议成为**加强的两阶段封锁**（strict two-phase locking）。在 16.6 节，我们将对它做详细的描述并且解释它的名字的由来。

## 16.4 重提三个并发问题

本节介绍如何用加强的两阶段封锁协议解决在 16.2 节中描述的三个问题。下面依次进行介绍。

**1. 丢失更新问题**

图 16-6 是图 16-1 的修改形式，表示图 16-1 在加强的两段封锁协议下事务交错执行的情况。事务 *A* 在时间 *t3* 的 UPDATE 未被接受，因为其隐含着对 *t* 的 X 锁请求，该请求与事务 *B* 拥有的 S 锁冲突，因此 *A* 进入等待状态。类似的原因，*B* 在时间 *t4* 也进入等待状态。这样两个事务都无法继续执行，因此不存在任何丢失更新问题。锁解决了丢失更新问题，但同时又引起另外一个问题。但至少其已解决了最初的问题，新问题称作死锁，将在 16.5 节讨论。

| 事务 *A* | 时间 | 事务 *B* |
|---|---|---|
| — | | — |
| — | | — |
| RETRIEVE t | *t1* | — |
| (acquire S lock on t) | | — |
| — | | — |
| — | *t2* | RETRIEVE t |
| — | | (acquire S lock on t) |
| — | | — |
| UPDATE t | *t3* | — |
| (request X lock on t) | | — |
| wait | | — |
| wait | *t4* | UPDATE t |
| wait | | (request X lock on t) |
| wait | | wait |
| wait | | wait |
| wait | | wait |

图 16-6 更新未丢失，但在时间 *t4* 发生了死锁

**2. 未提交依赖问题**

图 16-7 和图 16-8 分别是图 16-2 和图 16-3 的修改形式，表示图 16-2 和图 16-3 在加强的两段封锁协议下事务交错执行的情况。事务 *A* 在时间 *t2* 的操作（图 16-7 中为 RETRIEVE，图 16-8 中为 UPDATE）在两种情况下都未被接受，因为其隐含着对 *t* 的锁请求，这与 *B* 所拥有的 X 锁相冲突，因此 *A* 进入等待状态。其一直处于等待状态直到事务 *B* 的执行结束（或者是 COMMIT，或者是 ROLLBACK），当 *B* 的锁释放掉后，*A* 才能继续执行，而且在此时 *A* 所看到的是提交后的值（如果 *B* 以回滚结束，则为 *B* 之前的值；否则为 *B* 之后的值）。任何一种情况下，*A* 都不再依赖未提交的更新，所以我们解决了开始的问题。

**3. 不一致分析问题**

图 16-9 是图 16-4 的修改形式，表示图 16-4 在加强的两段锁协议下事务交错执行的情况。事务 *B* 在时间 *t6* 的 UPDATE 未被接受，因为其隐含着对 ACC 1 的 X 锁请求，该请求与事务 *A* 拥有的 S 锁冲突，因此 *B* 进入等待状态。同样事务 *A* 在时间 *t7* 的 RETRIEVE 也未被接受，因为其隐含着对 ACC 3 的 S 锁请求，该请求与事务 *B* 拥有的 X 锁冲突，因此事务 *A* 也进入等待状态。因此，锁方法解决了最初的不一致分析问题，同时引起了死锁。死锁将在 16.5 节讨论。

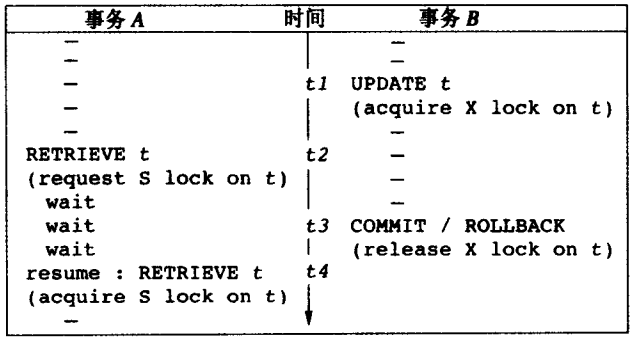



图 16-7　事务 A 在时间 t2 无法检索未提交的更新

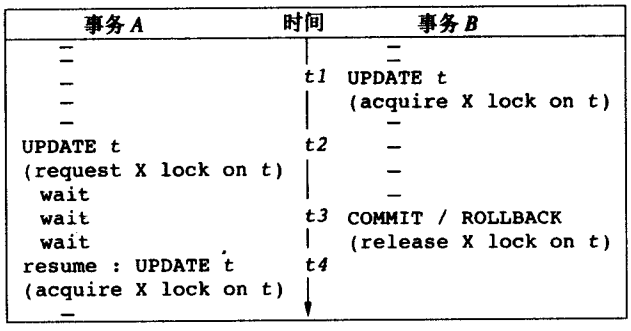



图 16-8　事务 A 在时间 t2 无法修改未提交的更新

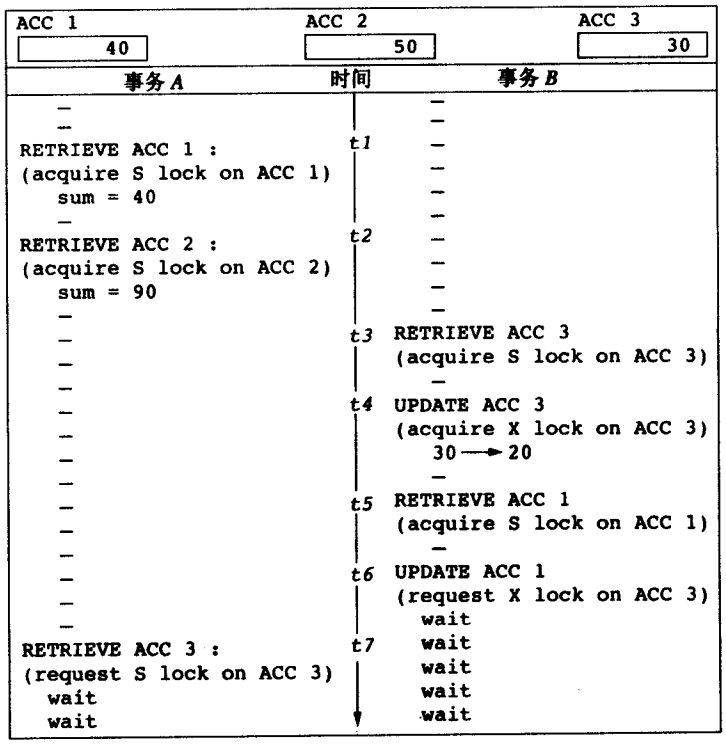



图 16-9　防止了不一致分析，t7 时刻又发生了死锁

## 16.5 死锁

到目前为止，我们已知道锁（更严格地说，是二级锁协议）是如何用来解决三个基本的并发问题的。不幸的是，锁自身也引发了问题，主要为死锁问题。前一节已给出了两个死锁的例子。图16-10给出了更一般的表示形式，图中的*r1*和*r2*（*r*表示资源）表示任何可锁的对象，不必仅为数据库的元组（参看16.9节），语句“LOCK…EXCLUSIVE”表示任何申请（排它）锁的操作，可以为显式或隐式的请求。

**死锁**发生时两个或更多的事务同时处于等待状态，每个事务都在等待其他的事务释放锁使其可继续执行。[⊖] 图16-10中死锁发生时涉及两个事务，但涉及三个、四个甚至更多事务都是可能的，至少原理上是这样。但是System R的实验表明，实际上死锁几乎从未涉及两个以上的事务［16.9］。

| 事务*A* | 时间 | 事务*B* |
|---|---|---|
| – | | – |
| – | | – |
| LOCK *r1* EXCLUSIVE | *t1* | – |
| – | | – |
| – | *t2* | LOCK *r2* EXCLUSIVE |
| – | | – |
| LOCK *r2* EXCLUSIVE | *t3* | – |
| wait | | – |
| wait | *t4* | LOCK *r1* EXCLUSIVE |
| wait | | wait |
| wait | | wait |

图16-10 死锁示例

死锁发生时，系统必须能检测并解除它。检测死锁就是检测**等待图**（即“谁在等待谁”的图——参看练习16.4）中的环。解除死锁就是选出一个死锁的事务，即图中环上的一个事务，将之回滚，从而释放其所拥有的锁使得其他一些事务可继续执行下去。注意：实际上，并不是所有的系统都进行死锁检测，有些系统采用超时机制，简单地假设在预定时间内不工作的事务发生了死锁。

可以看出，被选择回滚的事务自身并未发生错误，有些系统将自动重启这样的事务，当然，这是基于最初引起死锁的条件可能不再出现的假设。其他的系统则简单地向应用程序返回“死锁受害者”的代码，以便应用程序以某种合理的方式处理该情形。从程序员的角度看，前一种方法显然更可取。但是，即使有时需要程序员进行必要的处理，将问题隐藏起来不让最终用户感知总是很必要的。

**避免死锁**

除可允许死锁的发生并在产生死锁后解除死锁（绝大部分系统采用这种方式）之外，也可以修改封锁协议从而避免死锁。这里简要地讨论一种方案。这个方案提出了两种策略（虽然此方案是在分布式的环境中提出，但是同样适用于集中式系统），分别称为等待－死亡（Wait-Die）和伤害－等待（Wound-Wait）。这两种策略的工作原理如下：

- 每个事务都被标上它开始时间的时间戳（唯一的）。
- 当事务*A*申请在已被事务*B*封锁的元组上加锁时：
  - 等待－死亡：如果*A*比*B*年老，则*A*等待；否则*A*“死亡”——即*A*回滚并重新开始。
  - 伤害－等待：如果*A*比*B*年轻，则*A*等待；否则它将“伤害”*B*——即*B*回滚并重新开始。
- 如果一个事务重新开始，它保留它原来的时间戳。

注意：以上两个名字的前半部分（Wait或Wound）表示当*A*比*B*年老时发生的情况。由上面的描述我们知道，等待－死亡意味着所有的等待都是年老的事务等待年轻的事务，而伤害－等待则意味着所有的等待都是年轻的事务等待年老的事务。无论选用哪一种策略，很明显死锁都不会发生。同时，每一个事务都可以保证最后能够成功地执行——因为活锁（livelock）不可能发生（不会有事务陷入永久的等待），也不会有事务会无限次地重新开始。这种方法（不管是哪种策略）的主要缺点是会引起过多的回滚。

---

⊖ 死锁还被形象地称作死死拥抱（deadly embrace）。

## 16.6　可串行性

有了前面的基础，现在来解释一个重要概念——可串行性。可串行性通常作为多个事务执行的正确性准则。更精确的说法是，多个事务的某个调度是正确的，当且仅当其是可串行化的。[⊖]一组给定事务的执行是可串行化的——因此也是正确的——当且仅当它和这组事务的某个串行执行等价（也即保证产生相同的结果），其中：

- 串行执行是在某个序列中一次执行一个事务。
- 保证的含义是：不管数据库的初始状态怎样，给定执行与串行执行产生的最后结果总是一样的。

可串行性的判断方法如下：

1）正如在第 15 章讨论的，各单个事务如能将数据库从一个正确状态转变为另一个正确状态，就认为其是正确的。

2）按任何一个串行顺序依次运行多个事务也认为是正确的（这里串行顺序可为“任何一个”，是基于各单个事务彼此间相互独立的假定）。

3）有了以上的两个结论，我们说一个交错执行过程是正确的，当且仅当其与某个串行执行过程等价（即可串行化）。注意，“仅当”的意思是说：在给定特殊的初始条件的前提下，一个不可串行化的事务序列交错执行过程也可能产生正确的结果（参考练习 16.3），但这是不够的；我们希望正确性能够得到保证，而不仅仅是某种偶然的事件（也即独立于特定的数据库状态）。

从 16.2 节中的例子（图 16-1 ~ 图 16-4）可看出任何一种情形的问题都在于交错执行过程是不可串行化的，也就是说，其与先 *A* 后 *B* 或先 *B* 后 *A* 的串行执行过程都不等价。16.4 节所讨论的锁协议正是强制了每种情形的可串行性。图 16-7 和图 16-8 中的交错执行过程等价于先 *B* 后 *A*。图 16-6 和图 16-9 中发生了死锁，这暗示了两个事务中的一个将被回滚，假设该事务稍后将再次执行。如果 *A* 被回滚，则交错执行过程等价于先 *B* 后 *A*。

**术语**：多个事务，其任何一个执行过程，无论是否交错都被称作调度。依次执行这些事务，不相互交错，构成了一个串行调度。不是串行的调度称作交错调度，或简单地称作非串行调度。两个调度若保证产生同样的结果，与数据库的初始状态无关，则称其是等价的。因此一个调度是正确的（即可串行化的），当且仅当其与某一串行调度等价。

这里强调一下，相同事务的两个不同的串行调度可能产生不同的结果，因此这些事务的两个不同的交错调度也可能产生不同的结果，但两者都是正确的。举个例子，假设事务 *A* 为“x 加上 1”，事务 *B* 为“将 x 翻番”，这里 x 为数据库中的某项。同时假设 x 的初始值为 10，则先 *A* 后 *B* 的串行调度结果为 x = 22，而先 *B* 后 *A* 的串行调度结果为 x = 21。这两个结果都是正确的。任何一个保证与先 *A* 后 *B* 或先 *B* 后 *A* 等价的调度同样都是正确的。

可串行性的概念最初是由 Eswaran 等人在参考文献［16.6］中提出的，尽管当时使用的不是该名称。论文中还证明了一个重要的理论，称作两阶段锁理论，描述如下：[⊜]

*如果所有的事务都遵守“两阶段锁协议”，则所有可能的交错调度都是可串行化的。*

*两阶段锁协议为：*

- 在对任何一个对象（如一个数据库元组）进行操作之前，事务必须获得对该对象的锁；
- 在释放一个锁之后，事务不再获得任何其他锁。

遵守该协议的事务分为两个阶段：获得锁阶段，也称为“扩展”阶段；释放锁阶段，也称为“收缩”阶段。注意：实际系统中收缩阶段通常被压缩为事务结束时的单个操作 COMMIT 或 ROLLBACK（这一点我们将在 16.7 和 16.8 节提到）。实际上，16.3 节的数据存取协议可看作两阶段封锁协议的加强形式。

---

⊖ 实际上，可串行性有两种，即冲突可串行性和视图可串行性。视图可串行性几乎没有应用价值，因此可串行性经常用来作为冲突可串行性的简称。更详细的讨论可以参考文献［16.21］。

⊜ 两阶段封锁与两阶段提交没有关系，只是名字相似。

可串行性的概念有助于更清楚地理解事务并发，这里将给出另外一些结论。假设 *I* 为涉及事务 *T*1,*T*2,…,*Tn* 的交错调度。如果 *I* 可串行化，则存在某个涉及事务 *T*1,*T*2,…,*Tn* 的串行调度 *S*，使得 *I* 等价于 *S*。*S* 为 *I* 的串行调度（serialization）。

假定 *Ti* 和 *Tj* 为 *T*1,*T*2,…,*Tn* 事务集中的任意两个事务。不失一般性，假定在串行调度 *S* 中 *Ti* 在 *Tj* 之前执行。因此，在交错调度 *I* 的效果就像 *Ti* 确实在 *Tj* 之前执行一样。换句话说，可串行性的非正式但却很有用的特征为：如果 *A* 和 *B* 为某个可串行化调度的任意两个事务，则该调度中或者逻辑上 *A* 在 *B* 之前或者逻辑上 *B* 在 *A* 之前，即或者 *B* 看到 *A* 的输出结果或者 *A* 看到 *B* 的输出结果。（如果 *A* 更新资源 *x*，*y*，…，*z*，并且 *B* 可看到其中的任一资源，则 *B* 或者在所有的资源被 *A* 更新后看到这些资源，或者在 *A* 更新所有的资源之前就看到这些资源，总之不可能出现 *B* 在 *A* 更新资源期间看到这些资源的情况）。相反，如果效果不像 *A* 在 *B* 之前执行或 *B* 在 *A* 之前执行，则调度是不可串行化的，因而也是不正确的。

最后必须强调一点，如果某个事务 *A* 不是两阶段的，即不遵守两阶段锁协议，则总能构造某个其他的事务 *B* 以某种方式与 *A* 交错执行，产生不可串行化的不正确的全局调度。为了减少资源冲突，从而改善性能和吞吐量，实际系统通常允许构建不是两阶段的事务，即事务提早释放锁（在 COMMIT 前），并继续获得更多的锁。显然这样的事务冒着很大的风险，实际上，允许某一给定的事务 *A* 为非两阶段的，赌的是在系统中不存在与 *A* 交错运行的事务 *B*（如果存在，系统很可能将产生错误的答案）。

## 16.7 重提恢复

一个串行调度中的事务显然是可以恢复的，在必要的时候，使用在前一章中的技术可以撤消和/或重做事务。但是，如果允许事务交错执行，则事务未必就是可恢复的了。事实上，在 16.2 节提到的未提交依赖问题将引起可恢复性问题，说明如下：

现在假设回到 16.2 节，那时还没有封锁协议，因此事务也不需要等待获得一个锁。如图 16-11 中的情况，它是由图 16-2 修改过来的（不同的地方是现在事务 *A* 在事务 *B* 回滚之前提交）。如图所示，为了使 *B* 的回滚执行，就像 *B* 从来没有发生过，我们同时要回滚 *A*，因为 *A* 用到了 *B* 的更新值。但是，回滚已经提交的事务 *A* 是不可能的。所以，图中的调度是不可恢复的。

| 事务 *A* | 时间 | 事务 *B* |
|---|---|---|
| — | | — |
| — | | — |
| — | *t1* | UPDATE *t* |
| — | | — |
| RETRIEVE *t* | *t2* | — |
| — | | — |
| COMMIT | *t3* | — |
| | | — |
| | *t4* | ROLLBACK |

图 16-11 一个不可恢复的调度

一个调度是可恢复的充分条件［15.2］是：

如果 *A* 用到 *B* 的任何更新，则 *A* 不能在 *B* 终止之前提交。

很明显，我们希望并发控制机制，即锁协议（如果我们用的是锁），应该保证所有的调度都是可恢复的。

但是，刚刚提到的并不是问题的全部。现在假设有一个封锁协议，它不严格遵守两阶段封锁协议，锁可以在事务终止前释放。然后考虑如图 16-12（修改自图 16-11，不同之处是事务 *A* 在事务 *B* 终止前没有提交，但是 *B* 提前释放了它持有的锁）所示的情况，和图 16-11 一样，如果执行 *B* 的回滚要求使 *B* 看上去好像没有执行过，同样也需要回滚 *A*，因为 *A* 还没有提交。但是这样的级联回滚肯定不是期望的；特别是，如果允许一个事务的回滚级联地回滚另一个事务，那就需要处理任意长度的这样的“级联链”。换言之，图中所示的调度的问题是其不能满足级联独立（cascade-free）。

一个事务调度是级联独立的一个充分条件［15.2］是：

如果 *A* 看到 *B* 的任何更新，则在 *B* 终止前 *A* 被禁止使用 *B* 的任何更新。

严格的两阶段封锁协议能保证所有的调度都是级联独立的，所以它被应用在绝大部分的系统中。[⊖] 显然，由前面的定义可知，级联独立的调度肯定是可恢复的。

---

⊖ 事实上，只要事务仍然是两阶段的，并没有必要将 S 锁一直保持到事务结束，所以加强的两阶段封锁有一点过于严格。

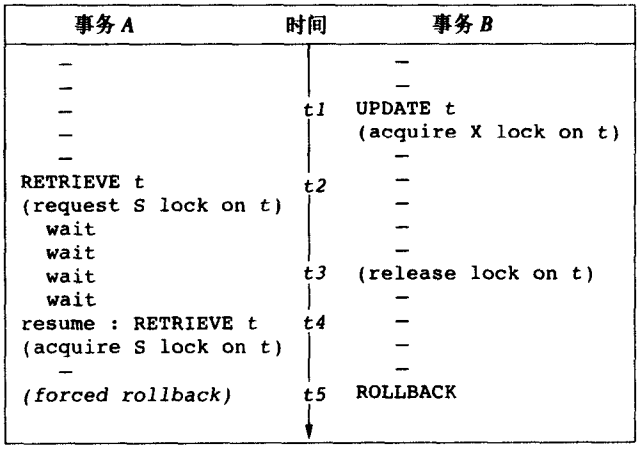



图 16-12　一个级联回滚的调度

## 16.8　隔离级别

可串行性保证了 ACID 特性中的隔离性。我们有一个直接而且令人满意的推论：如果所有的调度都是可串行化的，那么程序员在为事务 *A* 编写程序时没有必要考虑系统中同时执行着事务 *B*。但是，保证可串行性的协议会大大降低整个系统的并发度，甚至到不能忍受的地步。所以在实际应用中，系统通常支持多种“隔离”级别（下面我们将很快看到，在所有的非最高隔离级别下，事务之间并不是真正相互隔离的）。

*隔离级别*用来表示一个事务在与其他事务并发执行时所能容忍干扰的程度。如果可串行性得到了保证，根本就不存在可能被接受的干扰，换句话说，就是此种情况的隔离级别为可能的最高级别；否则正确性、可恢复性和级联独立的调度就不能得到保证。然而，正如已经指出的，由于各种实用的原因，实际系统通常允许事务在低于最高级别的隔离级别下操作。在这一节，我们简要地分析这个问题。

至少可定义五个不同的隔离级别，但参考文献［16.10］、SQL 标准和 DB2 只支持四个。通常，隔离级别越高，干扰越少，并发程度越低；隔离级别越低，干扰越多，并发程度越高。作为示例，可考虑 DB2 所支持的两个级别，分别为游标稳定性（cursor stability）和可重复读（repeatable read）。*可重复读*（RR）是最高级别，如果所有的事务都操作在该级别，则所有的调度都是可串行化的。而在*游标稳定性*（CS）级别，如果事务 *A*

1）获得了对某个元组 *t* 的可寻址性，[⊖] 并因此

2）获得对 *t* 的锁，接着

3）未对元组更新并放弃了对 *t* 的可寻址性，因此

4）未将锁升级为 X 锁，随后

5）不必等到事务结束即可释放该锁。

但是，其他的某个事务 *B* 现在可对 *t* 进行更新并提交这一修改。如果事务 *A* 又再次访问 *t*（注意这里破坏了两阶段封锁协议），将看到更新后的数据，也即看到了数据库的不一致状态。而在可重复读（RR）级别，所有对元组的锁（不仅仅是 X 锁）都将保持到事务结束，从而前面提到的问题将不会产生。

几点说明：

1）前面提到的问题并不是在 CS 级别产生的唯一问题，只不过它最容易解释。但是不幸

---

⊖ 如第 4 章所说，它是通过将游标设置到该元组来实现的——因此有“游标稳定性”的命名。如果准确的说，在 DB2 中 *T1* 获得的 *t* 上的锁实际上是“更新”（U）锁，而不是 S 锁（见参考文献［4.21］）。

的是，这表明只有在一个事务需要访问元组两次这种不大可能发生的情况下才需要 RR 级别。相反，有观点认为 RR 总是比 CS 更优的选择，在 CS 级别运行的事务不是两阶段的，因此无法保证可串行性（见 16.7 节）。当然，相应的观点也认为 CS 比 RR 的并发程度高（可能但不一定）。

2）可串行性在 CS 隔离级别下是不能保证的，这在实际的应用中看起来并不好理解。所以这里需要重申一个事实：如果一个事务 *T* 在非最高隔离级别下操作，那么在 *T* 与其他事务一起并发执行时，不能保证数据库是从一个正确状态转换到另一个正确状态。

3）支持低于最高级别的隔离级别的系统实现通常提供一些显式的并发控制能力（一般为显式的 LOCK 语句），从而允许用户利用它编写自己的应用程序，以保证系统自身未提供保证的安全性。DB2 提供了显式的 LOCK TABLE 语句，允许用户操作在低于最高隔离级别的某个级别上获得显式锁，这些锁的级别比 DB2 在那个隔离级别自动实施的锁级别要高。（顺便提一句，SQL 标准中并没有这样的显式并发控制机制，参考 16.11 节。）

注意，前面提到 DB2 的实现中可重复读（RR）被描述成最高隔离级别。但是在 SQL 标准中，可重复读却是表示一个严格低于最高隔离级别的隔离级别（同样请参考 16.11 节）。

**幻影**

幻影问题是事务在非最高隔离级别下操作可能产生的特殊问题之一。请看下面的例子（可能有些不现实，但是很好地说明了问题）：

- 首先，假设事务 *A* 计算客户 Joe 的所有银行账户上余额的平均值。并假设有三个这样的账户，每个都有余额 100 元。因此事务 *A* 扫描 3 个账户，并在处理过程中获得它们上面的共享锁，得到结果是 100 元。
- 现在，假设有另一个并发执行的事务 *B*，它向数据库中为客户 Joe 添加一个余额为 200 元的新账户记录。并可假定新账户是在 *A* 计算平均余额之后添加的，并且 *B* 在添加完成后立即提交（同时释放它在新元组上持有的排它锁）。
- 之后，假设 *A* 再一次扫描客户 Joe 的所有账户，计算总余额和账户个数并用总余额除以账户个数（可能它希望确定平均余额是否等于总余额除以账户个数）。这时它会发现有 4 个账户而不是 3 个，而且得到的结果是 125 元而不是 100 元！

在上面的例子中，两个事务都遵守严格的两阶段封锁，但是错误仍然发生了，事务 *A* 在第一次时看到了一个并不存在的东西——一个*幻影*。结果，可串行性被破坏了。（显然，这个交错执行过程和先 *A* 后 *B* 及先 *B* 后 *A* 这两个串行执行过程都不等价。）

经过仔细的考虑可以发现，这里的问题与两阶段封锁的本质根本不相关。真正的问题在于事务 *A* 并没有对它逻辑上需要封锁的东西加锁；这里真正需要加锁的并不是客户 Joe 的三个账户，而是 Joe 拥有的账户集合，或者说是谓词“account holder = Joe”（见参考文献［16.6］和［16.13］）。[⊖] 如果它加对了锁，那么事务 *B* 在添加新账户时就需要等待（因为 *B* 肯定需要请求新元组上的一个锁，这个锁会和 *A* 持有的锁冲突）。

尽管参考文献［15.12］中提到的种种原因说明了现在的大部分系统都不支持谓词锁，它们通过对当前数据的访问路径加锁的方法来避免幻影问题。考虑刚才 Joe 的账户的情况，并假设访问路径是一个客户名字的索引，系统可以对该索引上的客户 Joe 项加锁。这种方法可以避免幻影产生，因为幻影的产生首先需要请求访问路径（这里是一个索引项）的更新，因此需要得到访问路径上的 X 锁。更深入的讨论请见参考文献［15.12］。

## 16.9 意向锁

到目前为止，一直假设锁单元是单个元组。但理论上没有任何理由能说明为什么锁不能应用在更大或更小的数据单元上，如整个关系变量甚至整个数据库，或特定元组的特定部分。由此将引出*锁粒度*这一概念［16.10，16.11］。一般认为：粒度越细，并发程度越高；粒度越粗，需要

⊖ 我们也可以说 *A* 需要对属于 Joe 的任何不存在的账户加锁。

设置和测试的锁就越少，并发程度越低。举例来说，如果一个事务具有一个关系变量上的 X 锁，则没有必要对该变量的各单个元组设置 X 锁；另一方面，没有并发事务可获得该关系变量上的锁或该变量的元组上的锁。

假设某个事务 *T* 申请某关系变量 *R* 上的 *X* 锁。在接受 *T* 的请求之前，系统必须能判断出其他的事务是否已拥有 *R* 的任何一个元组上的锁。如果确实拥有，则 *T* 的请求此时无法满足。系统怎样才能检测出这样的冲突？显然，检查每个元组看是否有元组被其他的某个事务锁住，或检查每个已存在的锁看是否有锁施加在 *R* 的元组上，这些方法都是不明智的。由此将介绍另一个协议，即*意向锁协议*。根据该协议，事务在允许对某一元组加锁之前必须首先获得对包含该元组的关系变量上的锁，即意向锁（参见下一段）。该例中的冲突检测由此将变得很简单，只需看事务在关系变量级是否有冲突的锁。

前面已说明 X 锁和 S 锁对关系变量和对单个元组一样有意义，根据参考文献［16.10，16.11］，这里将再介绍三种新的锁，称为意向锁，这些锁只对关系变量有意义，而对单个元组无意义。这三种锁分别称为*意向共享*（IS）、*意向排它*（IX）、*共享意向排它锁*（SIX）。它们非形式化的定义如下（假定事务 *T* 请求获得关系变量 *R* 上的指定类型的锁，为完整起见，这里也包括 X 锁和 S 锁的定义）：

- 意向共享锁（IS）：*T* 意欲对 *R* 的元组设置 S 锁，以保证这些元组被处理时的稳定性。
- 意向排它锁（IX）：与 IS 类似，必须补充一点，*T* 可能对 *R* 的元组进行更新，从而将对这些元组设置 X 锁。
- 共享锁（S）：*T* 允许并发的读操作，但不允许并发的更新操作。*T* 自身在 *R* 的任意元组上都无更新操作。
- 共享意向排它锁（SIX）：综合了 S 和 IX 锁。即 *T* 允许并发的读操作，但不允许并发的更新操作；另外，*T* 可能对 *R* 的元组进行更新，从而对这些元组设置 X 锁。
- 排它锁（X）：*T* 根本就不允许任何对 *R* 的并发存取，*T* 自身可能对 *R* 的元组进行更新操作。

这五种锁类型的正式定义可用 16.3 节所提到的锁类型相容矩阵的扩展形式表示。如图 16-13。

下面给出意向锁协议的更精确的表述：

1）事务在获得某一元组上的 S 锁之前，必须首先获得包含该元组的关系变量上的 IS 或更强的锁。

2）事务在获得某一元组上的 X 锁之前，必须首先获得包含该元组的关系变量上的 IX 或更强的锁。（但是这还不是完整的定义，见参考文献［16.10］的注释。）

| | X | SIX | IX | S | IS | – |
|---|---|---|---|---|---|---|
| X | N | N | N | N | N | Y |
| SIX | N | N | N | N | Y | Y |
| IX | N | N | Y | N | Y | Y |
| S | N | N | N | Y | Y | Y |
| IS | N | Y | Y | Y | Y | Y |
| – | Y | Y | Y | Y | Y | Y |

图 16-13　扩展到包含意向锁的相容矩阵

前面协议中提到的相对锁强度的含义解释如下（参见图 16-14）：锁类型 *L2* 强于（在图中高于）锁类型 *L1*，当且仅当在相容矩阵某一行中 *L1* 列对应的那项为“N”（冲突）。在同一行中 *L2* 对应的那项也为“N”（参看图16-13）。若某一锁类型的请求无法满足，对比之更强的锁类型的请求必定也无法满足。这一事实表明，使用比要求的锁类型更强的锁总是安全的。S 和 IX 彼此都不比对方更强。

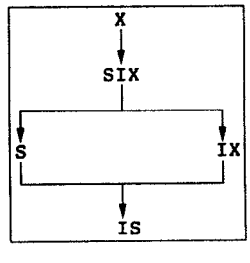



图 16-14　锁类型前驱图

意向锁协议所要求的关系变量级的锁通常可隐式地获得。例如，对某一只读事务，系统可能对事务存取的每一个关系变量都隐式地获得 IS 锁。而对某一更新事务，则可能获得 IX 锁。但系统也可能提供某种显式的 LOCK 语句以允许事务获得关系变量上的 S、X、SIX 锁。DB2 就支持这样的语句，虽然只有 S 和 X 锁，没有 SIX 锁。

最后讨论锁升级（lock escalation）的概念。锁升级在许多系统中

得以实现，用于平衡高并发度和低锁管理开销之间的冲突。其基本思想就是当系统达到预定的阈值时，系统将自动用一个粗粒度的锁取代一组细粒度的锁。例如，取代一组单个元组级的 S 锁，将包含这些元组的关系变量上的 IS 锁改变为 S 锁。该技术在实际应用中很有效［16.9］。

## 16.10 ACID 的不足之处

在第 15 章曾经提到在这一章将会有更多的关于事务的 ACID 特性的讨论；实际上，确实有一些关于这个主题的非正统的观点，接下来我们将讲述。

首先做一个简要的回顾：ACID 是原子性、正确性、隔离性、持久性的英文首字母简写。

- 原子性：任何事务要么全做，要么全不做。
- 正确性（文献中通常称为一致性）：任何事务的执行都是将数据库从一个正确状态转换为另一个正确状态，但在事务执行的任何中间点没有必要保持正确状态。
- 隔离性：在事务提交之前，它的任何更新对于其他所有事务都是不可见的。
- 持久性：一旦事务提交，即使随后系统崩溃，它的更新就被永久保存在数据库中。

ACID 是一个好的简写，但是它表达的概念是否真的能够经受严密检验？答案是否定的。在这一节，我们列出一些证据来回答这个问题。

**1. 即刻约束检查**

我们首先从一个看似题外话的论点开始：所有的完整性约束都要即刻地检查（在语句结束时），而不是推迟到事务的结束。这一观点曾经在第 9 章介绍过，现在再次辨明这一立场。至少有四个理由支持这一立场，现描述如下：

1）我们知道，一个数据库可以看成是一个命题的集合（通常情况下看是这样）。如果这个集合允许不一致性，那就没有什么可以讨论的了。一个不一致的数据库的回答是不可信的；实际上，这样的数据库可以给出任意的回答（证明请参考第 9 章的参考文献［9.16］）。隔离性或者“*I*”特性是说哪怕只是一个事务有可能看到某种特殊情况下的不一致，那么这个事务就有可能看到数据库的不一致状态并产生错误的结果。正是因为不一致性是不能够容忍的，即使是限制不允许一个以上的事务同时看到不一致性仍然不够，所以约束需要即刻实施。

2）在任何情况下都不能保证某个不一致状态（假设是允许的）只被一个事务看到。只有事务遵循特定协议（这里所说的特定协议还没有实施，事实上也不可实施）时，它们才能互相保持隔离。举个例子，假设事务 *A* 看到数据库的不一致状态并且将这一状态写入到某个文件 *F*，然后事务 *B* 刚好从 *F* 中读取了这一信息，那么 *A* 和 *B* 就不是真正互相隔离的（不管它们是否并发）。[⊖] 换言之，至少可以说事务的“*I*”特性是可疑的。

3）本书的前一版本曾提到：关系变量约束（relvar constraint）是即刻检查的而数据库约束是在事务结束时检查的（很多作者曾使用不同的术语同时阐述了这一点）。但是可**交换性原则**（The Principle of Interchangeability）（基本和导出的关系变量——参看第 9 章）意味着，一个真实世界的约束可能是关于数据库的关系变量约束和关于另一数据库的数据库约束！又因为关系变量约束必须即刻检查，所以数据库约束也必须即刻检查。

4）执行“语义优化”的能力要求数据库必须随时保持一致性，而不仅仅是在事务的边界上保持一致性。注意：语义优化是通过利用完整性约束简化查询来提高性能的一种技术。很明显，如果约束不被满足，那么简化就不可能是有效的。进一步的讨论请参考第 18 章。

当然，“传统的智慧”认为数据库约束的检查至少是不得不推迟的。举一个小的例子，假设供应商 - 零件数据库有约束“供应商 *S1* 和零件 *P1* 在同一个城市”。又假设供应商 *S1* 从伦敦搬到巴黎，那么零件 *P1* 也必须从伦敦搬到巴黎。这个问题的常规的解决方案是将这两个更新包装到一个事务中：

---

⊖ 实际上，即使 *A* 没有看到数据库的不一致状态，问题仍然会发生；因为 *A* 仍然可能将不一致的数据写入文件而且随后被 *B* 读到。

```
BEGIN TRANSACTION ;
UPDATE S WHERE S# = S# ('S1') { CITY := 'Paris' } ;
UPDATE P WHERE P# = P# ('P1') { CITY := 'Paris' } ;
COMMIT ;
```

在常规的解决方案中，约束在 COMMIT 时检查，在两次更新之间，数据库是不一致的。特别是，如果执行更新的事务在两次更新操作的中间向数据库提问："供应商 *S1* 和零件 *P1* 在同一个城市吗?" 会得到否定的回答。

回想起来，我们需要支持多赋值任务操作符以允许在同一个操作（也即一条语句）中执行多个任务，而在所有这些任务执行的期间没有任何的完整性约束检查。我们知道，INSERT、DELETE 和 UPDATE 都是某个特定任务的简写。在刚才的例子中，为了在一条语句内执行两个更新，可以这样写：

```
UPDATE S WHERE S# = S# ('S1') { CITY := 'Paris' } ,
UPDATE P WHERE P# = P# ('P1') { CITY := 'Paris' } ;
```

这样，在两个更新都完成以前（也就是到达分号时）就不再有完整性检查。注意，现在事务再也不会在两次更新之间看到数据库的不一致状态了，因为"两次更新之间"的概念现在不再有意义。

从这个例子可以看出，如果支持多任务操作符，那么就不再需要传统认识中的推迟检查（即将检查推迟到事务结束时）了。

接下来讨论 ACID 特性的本质。为了配合我们的目标，我们按照 C-I-D-A 的顺序展开。

**2. 正确性**

我们已经阐述了使用正确性一词取代更经常用到的一致性一词的原因（见第 15 章）。事实上，在数据库领域内通常认为这两个概念是等同的。例如在 Gray 和 Reuter 的书［15.12］的术语表中有：

**Consistent.** Correct.（一致的；正确的。）

在这本书中给出的事务的一致性的定义如下：

一致性：事务是状态的正确转换。转换动作的整体不破坏任何与状态相关的完整性约束。它要求事务是一个正确的程序［sic］。

但是如果完整性约束总是即刻检查，数据库总是一致的，而事务也总是将数据库从一个一致状态转换到另一个一致状态，但是，这些都不代表数据库是必然正确的。

所以如果 ACID 中的 C 代表一致性，这个特性就不重要了，[⊖] 如果代表正确性，又是不可实现的。无论是哪种情况，这个特性都是没有意义的，至少从一个正式的立场上看是如此。我们更倾向于认为 C 代表正确性，所以我们说"正确性"并不是一个真正的特性，而是一种期望之物。

**3. 隔离性**

现在讨论隔离性。在本节开头"即刻约束检查"小节中已经解释过，这个特性也是疑义的。如果每个事务都表现得是系统中唯一的事务，那么一个合适的并发控制机制可以保证隔离性（和可串行性）。但是，"表现得像是系统中的唯一事务"还意味着这个事务必须：

- 不试图有意或者无意地同其他并发或者非并发的事务交流。
- 不试图侦测系统中存在其他事务的可能性（当被赋予非最高的隔离级别时）。

所以，由于上述条件的限制，隔离性同样更像是一种期望之物。而在实际的系统中甚至存在肯定会破坏隔离性的显式的机制（即非最高的隔离级别）。

**4. 持久性**

接下来讨论持久性。因为有恢复机制，只要事务不嵌套（在这之前我们已经这样假设过），这个特性就是合理的。现在假设事务允许嵌套，特别地，假设事务 *B* 嵌套在事务 *A* 中，并且发生了下面的事件序列：

---

⊖ 事实上，即使约束不是即刻检查（如果事务违背了任何约束，事务仍然会回滚，跟没有执行的效果一样），一致性还是不重要。换言之，只要事务不违背任何约束，它们就会对数据库有持久的作用。

```
BEGIN TRANSACTION (transaction A) ;
   ...
   BEGIN TRANSACTION (transaction B) ;
   transaction B updates tuple t ;
   COMMIT (transaction B) ;
   ...
ROLLBACK (transaction A) ;
```

如果 *A* 的 ROLLBACK 执行，那么 *B* 也会被回滚（因为 *B* 是 *A* 的一部分），这样 *B* 对于数据库的更新就不再是“持久的”了；事实上，*A* 的 ROLLBACK 使得元组 *t* 的值恢复到 *A* 执行以前的值了。换言之，持久性不再被保证。至少对于像本例子中 *B* 这样嵌套在别的事务中的事务是如此。

现在，许多的研究者（Davies 是发起人，见参考文献［15.8］）开始仿照刚才例子中的形式研究支持嵌套事务。参考文献［15.15］指出，这种支持至少从以下三个方面来看是值得的：事务内并发、事务内恢复控制和系统模块性。正如刚才的例子所示，在一个支持嵌套事务的系统中，内层事务的 COMMIT 提交事务的更新，但只是相对于内层事务的直接外层。实际上，外层事务对内层事务的 COMMIT 拥有否决权——如果外层事务回滚，内层事务相应回滚。在本例中，*B* 的 COMMIT 只是对 *A* 的，而不是对外面世界的，而它随后确实被撤消（回滚）。

注意：我们可以说嵌套事务是保存点的一个概括思想。保存点允许事务结构化成一个按顺序依次执行动作的线性序列（这样，事务可以回滚到序列中任何前面的动作）。与之形成对比，嵌套允许事务递归地结构化，是一个并发执行的动作的层次结构。换言之：

- BEGIN TRANSACTION 扩展支持“子事务”（也就是说，如果在一个事务已经运行时遇到 BEGIN TRANSACTION，将开始另一个子事务）。
- COMMIT“提交”，但只限于双亲范围内（如果事务是子事务）。
- ROLLBACK 撤消操作，但只撤消到这个事务的开始处（包括它的孩子、孙子等，但是不包括可能的双亲事务）。

回到讨论的主线：事务的持久性只是在最外层有效[⊖]（换言之，只对不被任何其他事务嵌套的事务有效）。因此，这个特性也不能 100% 保证。

**5. 原子性**

最后讨论原子性。和持久性一样，这个特性也是由系统的恢复机制保证的（嵌套事务也如此）。这里我们的观点有些不同。特别地，如果系统支持多任务，那么事务就没有必要拥有原子性；而语句拥有原子性仍然是充分的。更进一步，语句拥有原子性也是必要的，其原因已经在别的地方讨论过了。[⊖]

**6. 结论**

我们可以用下面几个带修饰色彩的问题来概括本节：

- 事务是操作单元吗？是，但只有在不支持多任务时才是。
- 事务是恢复的单元吗？答案同上。
- 事务是并发的单元吗？答案同上。
- 事务是完整的单元吗？是，但仅当不支持“所有约束都即刻检查”时才成立。

注意：在回答这些问题时没有考虑在前面的章节中（和本书的以前版本）提出的有关这个领域内的更加有“传统的智慧”不同看法。

总的来说，我们总结：事务作为一个重要的概念，更多的是从实际的角度而不是从理论的角度来看。请注意这并不是一种贬低！对于过去 25 年以来事务管理领域的一流的研究成果，我们拥有唯一的情感是尊敬。我们仅仅是注意到现在我们对于研究所基于的那些假设有了更好的理解——特别是对于完整性约束的至关重要的角色有了更好的理解，同时也发现了支持多任务并把它作为一个基本操作符的需要。实际上，如果前提假设改变不导致结论的改变，那才是令人感到

---

⊖ 参考文献［15.15］说明了对于一致性特性也是如此。因为像在绝大部分情况下那样，这里仍然假设最外层的事务在它执行的中间没有必要保持一致性，但是，前面的原因已经驳回了这个观点。

⊖ 绝大部分 SQL 标准中的语句都拥有原子性。

惊讶的。

## 16.11　SQL 的支持

SQL 标准不提供任何显式的锁支持，实际上，根本就未提及锁。[⊖] 但是，实际系统必须采取相关措施以免并发执行的事务相互干扰。特别地，它要求事务 *T1* 的更新对其他不同的事务 *T2* 不可见，直到事务 *T1* 提交更新结束。*注意*：假设所有的事务都在下面的隔离级别下执行：读已提交（READ COMMITTED），可重复读（REPEATABLE READ），或者可串行化（SERIALIZABLE）（参考下面一段）。需要特别注意的是在读未提交（READ UNCOMMITTED）的隔离级别下，事务（a）可以进行“脏读”；（b）必须是只读（READ ONLY）的（如果允许 READ WRITE，那么可恢复性将得不到保证）。

回顾第 15 章的内容，SQL 隔离级别是在 START TRANSACTION 语句中定义的。它有四个候选项：**SERALIZABLE**，**REPEATABLE READ**，**READ COMMITTED** 和 **READ UNCOMMITTED**。[⊜] 默认是 SERIALIZABLE，如果指定了其他三个级别中的任何一个，执行时可以自由地设置某一较高的级别。这里“较高”是根据这样的排序定义：SERIALIZABLE > REPEATABLE READ > READ COMMITED > READ UNCOMMITED。

如果所有事务都执行在隔离级别 SERIALIZABLE（默认情况下），则任意一组事务的交错执行过程都是可串行化的。但是，如果有事务执行在较低的隔离级别，则可能存在多种不同的违背可串行性的情形。标准定义了三种情形：读脏数据、不可重复读和幻影（前两个在 16.2 节做出了解释，第三个的解释见 16.8 节），对应不同的隔离级别，分别有不同的违背可串行性的情况可能发生。[⊝] 这些情况总结在图 16-15 中（“Y”表示违背可串行性的情况可能发生，“N”表示不可能发生）。

| 隔离级别 | 读脏数据 | 不可重复读 | 幻影 |
|---|---|---|---|
| READ UNCOMMITTED | Y | Y | Y |
| READ COMMITTED | N | Y | Y |
| REPEATABLE READ | N | N | Y |
| SERIALIZABLE | N | N | N |

图 16-15　SQL 隔离级别

在本节结束之前，这里要重复的一点是：SQL 的可重复读（REPEATABLE READ）和 DB2 中的“可重复读（RR）”不是一回事。事实上，DB2 中的 RR 与 SQL 中的“可串行化”的隔离级别相同。

## 16.12　小结

本章考虑了有关**并发控制**的问题。首先介绍了并发事务的交替执行中没有并发控制时会产生的三个问题，即：**丢失更新**、**未提交依赖**和**不一致分析**。产生这些问题的调度都是**不可串行化**的，即不能等价于涉及相同事务的某个串行调度。

解决这些问题的最常用的技术是**锁**。锁的基本类型有两种，即**共享锁**（S 锁）和**排它锁**（X 锁）。如果某个事务在某个对象上施加了 S 锁，则其他事务也能在该对象上申请 S 锁，但不能对该对象申请 X 锁；如果某个事务在某个对象上施加了 X 锁，则其他事务不能对该对象申请任何锁。为了保证不发生丢失修改，可以引入这样一个锁协议：对任何要检索的对象申请 S 锁，对任

---

⊖ 冗余是必要的——系统为了实现期望的功能，就要求能够自由地选用任何并发控制机制。

⊜ SERIALIZABLE 用在这里并不特别合适，因为它指的是调度的可串行化，而不是事务的。一个更好的术语是 TWO PHASE，表示事务将遵守（或强制遵守）两阶段封锁协议。

⊝ 请见参考文献［16.2］和［16.14］。

何要更新的对象申请X锁，并将锁保持到事务结束。该协议可以实现事务的串行化。

上面所描述的协议是**两阶段封锁协议**的一个较强但很常见的形式。由此得出**两段锁定理**：如果所有的事务都遵循该协议，那么所有的调度都是可串行化的。可串行调度意味着：如果*A*和*B*是该调度所涉及的两个事务，那么或者*A*看到*B*的输出，或者*B*看到*A*的输出。遗憾的是，两阶段封锁协议可能导致**死锁**的发生。打破死锁的办法是选择死锁事务中的某一个事务作为牺牲者(victim)，然后将该事务回滚（释放牺牲者所锁定的全部资源）。

一般说来，如果事务未达到完全可串行性，则不能保证一定安全。但系统一般都允许事务在某个实际上并不安全的**隔离级别**上运行，这样可以减少资源的竞争并提高事务吞吐量。本章将这样一个“不安全”的级别描述为**游标稳定性**（这是DB2中的术语，在SQL中则用READ COMMITTED来表示）。

接下来主要说明**锁粒度**的问题及相应的**意向锁**思想。意向锁的基本观点是：事务在某些对象(如数据库中的元组）上获得某种锁前，它必须先获得在该对象的“父对象”上的一个适当的意向锁。例如，对于一个元组，则要在包含该元组的关系变量上获得一个意向锁。这样的意向锁通常以隐式申请方式实现，就像元组上的S锁和X锁以隐式方式申请一样。但是也应该提供某些意向锁的显式锁语句，以允许事务在必要时能够获得较隐式方式获得的锁更强的锁（尽管SQL标准并不提供这样的机制）。

然后是对事务的**ACID特性**做了一次回顾，包括这个范围内并不像一般的假设那么清晰的那些内容。本章最后概述了SQL对并发控制支持。SQL基本上不提供任何的显式锁能力，但它支持多种隔离级别，即：**可串行化**、**可重复读**、**读已提交**和**读未提交**的等四种。这些不同的隔离级别可以由DBMS通过非显式锁方式在底层加以实现。

## 习题

16.1 用你自己的话解释可串行化。

16.2 说明（a）两阶段封锁协议；（b）两阶段封锁定理。准确的解释两阶段封锁是怎样解决RW，WR和WW冲突的。

16.3 给定事务*T*1、*T*2、*T*3，执行下列操作：

*T*1：将*A*加1；

*T*2：将*A*加倍；

*T*3：在屏幕上输出*A*，并将*A*置为1，其中*A*为数据库中的某个数据项。

a）假设*T*1、*T*2、*T*3可以并发执行。若*A*初值为0，那么存在多少种可能的正确结果？列举之。

b）各个事务的内部结构如下表所示。若事务执行不施加任何锁，则有多少种可能的调度？

| *T1* | *T2* | *T3* |
|---|---|---|
| R1: RETRIEVE A<br>INTO a1 ;<br>a1 := a1 + 1 ;<br>U1: UPDATE A<br>FROM a1 ; | R2: RETRIEVE A<br>INTO a2 ;<br>a2 := a2 * 2 ;<br>U2: UPDATE A<br>FROM a2 ; | R3: RETRIEVE A<br>INTO a3 ;<br>display a3 ;<br>U3: UPDATE A<br>FROM 1 ; |

c）如果*A*的初值给定为0，是否存在能够产生“正确”结果但不可串行化的交错调度？

d）如果这三个事务都遵循两阶段封锁协议，那么存在事实上可串行化但又不能形成的调度吗？

16.4 下面给出的是一个调度的事件序列。该调度包含*T*1、*T*2、…、*T*12等12个事务，*A*、*B*、…、*H*为数据库中的数据项。

```
time t0   ..........
time t1 (T1)   : RETRIEVE A ;
time t2 (T2)   : RETRIEVE B ;
  ...   (T1)   : RETRIEVE C ;
  ...   (T4)   : RETRIEVE D ;
  ...   (T5)   : RETRIEVE A ;
  ...   (T2)   : RETRIEVE E ;
  ...   (T2)   : UPDATE E ;
  ...   (T3)   : RETRIEVE F ;
  ...   (T2)   : RETRIEVE F ;
  ...   (T5)   : UPDATE A ;
```

```
...    (T1)    : COMMIT ;
...    (T6)    : RETRIEVE A ;
...    (T5)    : ROLLBACK ;
...    (T6)    : RETRIEVE C ;
...    (T6)    : UPDATE C ;
...    (T7)    : RETRIEVE G ;
...    (T8)    : RETRIEVE H ;
...    (T9)    : RETRIEVE G ;
...    (T9)    : UPDATE G ;
...    (T8)    : RETRIEVE E ;
...    (T7)    : COMMIT ;
...    (T9)    : RETRIEVE H ;
...    (T3)    : RETRIEVE G ;
...    (T10)   : RETRIEVE A ;
...    (T9)    : UPDATE H ;
...    (T6)    : COMMIT ;
...    (T11)   : RETRIEVE C ;
...    (T12)   : RETRIEVE D ;
...    (T12)   : RETRIEVE C ;
...    (T2)    : UPDATE F ;
...    (T11)   : UPDATE C ;
...    (T12)   : RETRIEVE A ;
...    (T10)   : UPDATE A ;
...    (T12)   : UPDATE D ;
...    (T4)    : RETRIEVE G ;
time t36  ..........
```

假定 RETRIEVE *i*（如果成功）获得 *i* 上的一个 S 锁，UPDATE *i*（如果成功）将锁升级到 X 锁。同时假定所有锁都保持到事务结束。请画出在 *t*36 时刻事务的等待图（指出谁在等待谁）。在该时刻存在死锁吗?

16.5　重新考虑图 16-1 ~ 图 16-4 中提出的并发问题。如果所有事务都在 CS 隔离级别而不是 RR 级上运行，会发生什么情况?（注意：如 16.8 节所示，这里的 CS 和 RR 相当于 DB2 中的隔离级别。）

16.6　分别给出锁类型 X、S、IX、IS 和 SIX 的非形式化和形式化定义。

16.7　定义相对锁强度的概念，并给出相应的前驱图。

16.8　定义意向锁协议。说明该协议的意义。

16.9　SQL 中定义了读脏数据、不可重复读和幻影等三个并发问题，它们和 16.2 节中说明的三个并发问题具有什么样的联系?

16.10　给出参考文献［16.1］中说明的多版本并发控制协议的一个概要的实现机制。

## 参考文献

除了下面给出的参考文献外，还可以参阅第 15 章中的［15.2］、［15.10］，尤其是［15.12］。

［16.1］　R. Bayer, M. Heller, and A. Reiser: "Parallelism and Recovery in Database Systems," *ACM TODS 5*, No. 2 (June 1980).

第 15 章中提到，在包括硬件和软件工程在内的新兴应用领域中经常要涉及十分复杂的处理要求，而本章大部分及前几章所描述的传统事务管理控制并不能很好地满足这些需求。其中一个很基本的问题就是复杂事务可能持续几小时甚至几天，而不是传统系统中所认为的几个毫秒，从而导致下面的后果：

1）将事务完全回滚到事务开始时的状态可能导致不可接受的大量工作丢失。

2）使用通常的锁可能产生不可接受的过长时延来等待放锁。

本文是阐述这些问题的文章之一（其他的文章可参阅［16.8］、［16.12］、［16.15］及［16.20］）。文中提出了**多版本锁**的并发控制技术，或者叫做**多版本读**技术。该技术在一些商品中已经得到实现。该技术的最大优点在于读操作不会等待，即任意数量的读者和一个写者能够在同一个逻辑对象上同时操作。尤其是：

- 读决不会延迟。
- 读决不会延迟更新。
- 完全没有必要回滚只读事务。
- 死锁只可能在更新事务之间发生。

这些优点在分布系统（见第 21 章）中尤为明显，因为在分布系统中的更新可能持续很长一段时间，这时只读事务将被过度地延迟（反之亦然）。多版本锁的基本思想如下：

- 事务 *B* 请求读一个对象时，如果事务 *A* 已经获得了在该对象上的更新存取路径，则系统为 *B* 提供该对象的一个已提交版本的存取路径。这一版本必须作为恢复之用而保留在系统中（如保存在系统日志里）。
- 事务 *B* 请求更新一个对象时，如果事务 *A* 已经获得了在该对象上的读存取路径，则 *B* 获得该对象的存取路径。这时 *A* 仍然指向原来的对象版本（此时的对象是一个真正的“以前版本”）。
- 事务 *B* 请求更新一个对象时，如果事务 *A* 已经获得了在该对象上的写存取路径，则 *B* 进入等待状态[⊖]（如前所述的死锁和强制回滚仍有可能发生）。

当然，这个方法也包括适当的控制以保证每个事务总是看到数据库的一致状态。

[16.2] Hal Berenson *et al.*：“A Critique of ANSI SQL Isolation Levels,”Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1995).

这篇论文的主要内容是批评 SQL 标准用可串行性违背来说明隔离级别的方式（见 16.11 节）。文章认为：“（SQL 的）定义不能准确地说明几个常见隔离级别的特点，包括其中涉及的几个级别的标准锁实现”。文章特别指出：标准不能防止写脏数据（写脏数据可以定义为两个事务 *T1* 和 *T2* 可能在两者终止前都在同一行上执行了更新）。

对于标准中没有显式地防止写脏数据这一点，看来是正确的。原文说明如下（略有变动）：

- “保证可串行隔离级别上的并发事务的执行是可串行的”。或者说，如果事务都在可串行隔离级别上执行，实现时必须要防止写脏数据，因为写脏数据必将违背可串行性。
- “四个隔离级别保证了……不会丢失更新”。这一说法只是一厢情愿，四个隔离级别的定义本身并未提供这样的保证，但它表明了标准定义者试图防止写脏数据。
- “一个事务所做出的修改直到该事务以提交方式结束时才能被其他事务（读未提交级别的事务除外）所感知”。这里的问题在于：“感知”的准确含义是什么？事务能够更新一个“脏数据”而不“感知”它吗？

也可见参考文献 [16.14]。

[16.3] Philip A. Bernstein and Nathan Goodman：“Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems,”Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada (October 1980).

文章讨论了一组基于时标而不是锁的并发控制方法。基本思想是：如果事务 *A* 在事务 *B* 之前启动执行，则系统将从整体上把 *A* 视为在 *B* 启动前执行完（就像一个真正的的串行调度）。它遵循：

1）*A* 无权看到 *B* 所作的任何更新。

2）决不允许 *A* 更新任何 *B* 已经看到的对象。

这两个条件按下面的方式实现：对任何给定的数据库请求，系统将把请求事务的时标和对所请求元组进行最近一次检索或更新的事务的时标相比较，如果发生冲突，则简单地将发出请求的事务重新启动，并指派一个新的时标（即所谓乐观方法 [16.16]）。

正如论文标题所隐含的那样，时标方法最初是在分布式环境中引入的。原因在于人们感到分布式系统中使用锁会由于检测消息、设置时钟等原因而产生不可忍受的系统开销。当然，在一个非分布式系统中它并不合适。事实上它在分布式系统中的可行性仍然值得怀疑。一个明显的问题是：每个元组必须保留最近检索（或更新）过它的事务的时标，这是指每一个读转变为一个写操作！事实上，参考文献 [15.12] 中已经指出：时标机制恰恰是 [16.16] 中的乐观并发控制模式的一个退化方案，而乐观控制模式本身也有问题。

*注意*：在数据库领域内广泛讨论过的一个概念——“托马斯写规则”（Thomas's write rule）[16.22]，实际上是前面机制的一个精炼。它基于这样一个事实：有些更新因为是陈旧的（用户级的请求被满足但是并没有物理上的更新），所以可以被忽略。

[16.4] M. W. Blasgen, J. N. Gray, M. Mitoma, and T. G. Price：“The Convoy Phenomenon,”*ACM Operating Systems Review 13*, No. 2 (April 1979).

*护航现象*（convoy phenomenon）是抢占式调度系统中使用高流量（high-traffic）锁时所遇

---

⊖ 换言之，WW 冲突仍然可能发生，这里我们假设使用封锁来解决，而其他的技术（例如时标 [16.3]）也同样可行。

到的问题。例如，向日志中写记录时所需要的锁就是一个高流量锁。注意，这里的“调度”是指为事务分配机器周期的问题，而不是本章中主要讨论的不同事务的数据库操作的交叉问题。问题如下：如果事务 *T* 正持有一个高流量锁且这时被系统调度器所强占，如可能因时间片到期而被强制进入等待状态，那么系统将为那些等待获得高流量锁的事务形成一个护航。当 *T* 离开等待状态时，它将很快释放该高流量锁，但由于该锁是高流量的，*T* 本身也可能在下一个事务未用完资源前而加入到护航中，因此不能继续处理下去，从而又进入一个等待状态。问题的本质在于：大多数情况下（而非全部），调度器是底层 OS 而不是 DBMS 的一部分，因此它是基于另外一个不同的假设来设计的。作者观察到：一旦建立护航，它将趋于稳定；系统将处于一个“锁颠簸”状态，这时大部分机器周期都用于处理切换而未做什么有意义的工作。一个不替换调度器的建议方案是将锁的授权方式不再基于“先来先服务”的原则，而采用随机方式。

[16.5] Stephen Blott and Henry F. Korth:“An Almost-Serial Protocol for Transaction Execution in Main-Memory Database Systems,” Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong (August 2002).

本文为主存系统（main-memory system）提出了一种完全不用锁的可串行化机制。

[16.6] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger: “The Notions of Consistency and Predicate Locks in a Data Base System,” *CACM 19*, No. 11 (November 1976).

该文首次将并发控制这一课题建立在一个合理的理论之上。

[16.7] Peter Franaszek, and John T. Robinson: “Limitations on Concurrency in Transaction Processing,” *ACM TODS 10*, No. 1 (March 1985).

见参考文献 [16.14] 中的说明。

[16.8] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian: “Concurrency Control for High Contention Environments,” *ACM TODS 17*, No. 2 (June 1992).

该文指出，因为各种原因，未来的事务处理系统所面临的并发度情况可能大大超出目前系统所支持的并发度，因此在这些系统中可能发生大量的数据竞争。作者随之提出了“大量非锁并发控制概念和可用于高竞争环境中的事务调度技术”。据称这些技术是基于使用了模拟模型的实验的，并能够在这些环境中“带来实质性好处”。

[16.9] J. N. Gray: “Experience with the System R Lock Manager,” IBM San Jose Research Laboratory internal memo (Spring 1980).

本文实际上只是一些注释性说明，而不是一篇完整的论文，其中的某些思想如今可能有些过时了，但它也包括了一些有趣的话题，如：

- 锁在联机事务中带来 10% 的代价，而在批处理事务中只占 1%。
- 支持多种锁粒度是值得的。
- 自动锁升级能够很好地工作。
- 可重复读（RR）比游标稳定性（CS）有效也更安全。
- 实际情况中的死锁很少发生且决不会超过两个事务。
- 几乎所有的死锁（超过 97%）能够通过支持 U 锁避免。U 锁在 DB2 中得到支持而系统 R 则不支持。

注意：U 锁定义为与 S 锁兼容而不能和 U 锁兼容，当然也不能和 X 锁兼容。细节可参阅参考文献 [4.21]。

[16.10] J. N. Gray, R. A. Lorie and G. R. Putzolu: “Granularity of Locks in a Large Shared Data Base,” Proc. 1st Int. Conf. on Very Large Data Bases, Framingham, Mass. (September 1975).

这篇论文引入了意向锁概念。和 16.9 中的解释一样，术语“粒度”是指能够被锁定的对象的大小。由于不同事务显然具有不同的特点和不同的要求，系统提供某个范围内的多种不同的锁粒度是值得的，而且大多数实际系统也是这样做的。这篇文章给出了基于意向锁的多粒度系统的一种实现机制。

由于本章中给出的解释显然有些简单，因此这里将给出关于意向锁协议的较详细的说明。首先，像前面假设的那样，可加锁对象类型不仅限于关系变量或元组。其次，这些锁对象甚至不必形成一个严格的层次；索引和其他存取结构的表现方式意味着它们应被视作有向无环图 DAG。例如，供应商和零件数据库可能包含零件关系变量 *P*（的某种存储形式）和一个建立在 P#属性上的索引 XP。如果要获得关系变量 *P* 中的元组，则必须先启动整个数据库，然后要么直接在关系变量中顺序扫描，要么先进入 XP，然后再找到所需的元组。这样在相应的 DAG 中 *P*

有两个“父对象”，*P* 和 XP，而这两个对象都以数据库作为它的“父对象”。

下面给出协议的一般形式。

- 获得指定对象上的 X 锁时，隐式获得该对象的所有子对象上的 X 锁。
- 获得指定对象上的 S 或 SIX 锁时，隐式获得该对象的所有子对象上的 S 锁。
- 在事务获得指定对象上的 S 或 IS 锁前，它必须先获得该对象至少一个父对象上的 IS（或更强的）锁。
- 在事务获得指定对象上的 X、IX 或 SIX 锁前，它必须先获得该对象的所有父对象上的 IX（或更强的）锁。
- 在事务能释放某个指定对象上的一个锁前，它必须先释放掉它在所有子对象上保持的锁。

在实际应用中，该协议不会产生想像中那样的系统开销。原因在于在任何给定时刻，事务可能已经获得了所需的大多数锁。例如，在整个数据库上的一个 IX 锁可能只需要在程序初始启动时申请，然后该锁在程序的这个生存期内的所有事务中得到保持。

[16.11] J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger: “Granularity of Locks and Degrees of Consistency in a Shared Data Base,” in G. M. Nijssen (ed.), *Proc. IFIP TC-2 Working Conf. on Modelling in Data Base Management Systems.* Amsterdam, Netherlands: North-Holland/New York, N. Y.: Elsevier Science (1976).

这篇论文介绍了隔离级别的概念（在一致性程度（degree of consistency）的名义下[⊖]）。

[16.12] Theo Härder and Kurt Rothermel: “Concurrency Control Issues in Nested Transactions,” *The VLDB Journal 2*, No. 1 (January 1993).

在第 14 章中提到，有些作者给出了关于嵌套事务思想的建议。这篇文章提出了适用于这些事务的一套较为恰当的锁协议。

[16.13] J. R. Jordan, J. Banerjee, and R. B. Batman: “Precision Locks,” Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data, Ann Arbor, Mich. (April/May 1981).

精确锁是一种元组级的锁模式，它保证了只有那些需要加锁的元组才被真正锁定以满足可串行性，这些锁定元组也包括幻影。它实际上是谓词锁的一种形式（参考 16.8 节及参考文献 [16.6]）。它的工作机制是：(a) 检查更新请求，看要插入的元组是否满足其他并发事务较早发出的检索请求；(b) 检查检索请求，看其他并发事务已经插入或删除的元组是否满足查询中的检索请求。该模式不但精巧，而且作者同时还声称它实际上也比传统技术（一般有太多的锁）运行得更好。

[16.14] Tim Kempster, Colin Stirling, and Peter Thanisch: “Diluting ACID,” *ACM SIGMOD Record 28*, No. 4 (December 1999).

这篇文章的题目取“Concentrating ACID”更加合适！它声明：传统的并发控制机制排除了某些可串行化的调度（“对于可串行性来说，隔离确实是一个充分但不必要的条件”）。就像在 SQL 标准和文献 [16.2] 中一样，它根据违背可串行性的情况定义了隔离级别；但是，它的定义比以前那些更加精致，并且容纳更多的可串行化调度。文章同时还演示了文献 [16.2] 的一个缺陷（与幻影有关）。

[16.15] Henry F. Korth and Greg Speegle: “Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model,” *ACM TODS 19*, No. 3 (September 1994).

正如参考文献 [15.3]、[15.9]、[15.16] 和 [15.17] 中所说明的，可串行性通常要求较严格的条件而无法在某些系统中实施。尤其在涉及人机交互和长周期事务的新兴应用领域中更是如此。这篇文章提出了一个叫做 NT/PV（带谓词和视图的嵌套事务）的事务模型来阐述这些问题。此外，文章提出：具有可串行性的标准事务模型只是一个特例。文中定义了“新的且更具意义的正确性类”，并声称新的模型提供了“一个可解决长周期事务问题的合理的框架”。

[16.16] H. T. Kung and John T. Robinson: “On Optimistic Methods for Concurrency Control,” *ACM TODS 6*, No. 2 (June 1981).

---

⊖ 不是名字的问题！数据可能是一致的，也可能不是。因此一致性“程度”的概念是有待商讨的。事实上，在我们对数据完整性（或一致性）的基础的重要性有一个明确的认识之前，“一致性程度”背后的理论已经发展起来了。

锁模型可以采用悲观模式。这时，锁采用了最坏情况假设，即某指定事务所存取的每个数据都可能被其他的并发事务所申请，因此最好加以锁定。对应的，乐观模式（也可称做认证或校验模式）则采用了相反的假设，它认为冲突实际上很少发生。因此，乐观模式允许事务毫无阻碍地运行到结束。然后在提交时检查是否真的发生了冲突，如果有，则将不合理的事务简单地重新启动。在提交处理成功完成前，不能将任何更新写入数据库中，这样，重启事务时不需要对任何更新进行 UNDO 处理。

在随后的论文［16.7］中，作者提出：在某些合理假设下，乐观方法在所支持的并发度（以同时执行的事务数目计）期望级别上比传统锁方法具有某些固有的优点。这意味着在拥有大量并行处理器的系统中，乐观方法可能更值得选择（但参考文献［15.12］却认为乐观方法在“热点”条件下一般比锁方法更糟。热点是指被多个事务频繁更新的数据项。对于在热点上工作得较好的技术的讨论可参见参考文献［16.17］）。

［16.17］ Patrick E. O'Neil：“The Escrow Transactional Method,”*ACM TODS 11*, No. 4 (December 1986).

考虑下面的简单例子。设数据库中的数据项 *TC* 表示“目前总现金数”，并假设系统中几乎所有事务都从 *TC* 中减少一定数量（或者叫提取现金）来更新 *TC*，则 *TC* 是一个“热点”的例子，也就是说，当数据库中的数据项被系统中运行的很大比例的事务所存取时，它被称作“热点”。在传统锁下，热点很快成为一个“瓶颈”。因此，对 *TC* 这样的数据项使用传统锁就显得强度太大。如果 *TC* 初始值为一千万美元，而每个事务平均只减去十美元，则可以执行约一百万个这样的事务，这样在出现问题前可以按任意次序进行一百万次相应的缩减。对于 *TC* 而言，不再需要传统锁，以只要确保当前值足以允许所需的减少量代之来进行更新（如果随后事务失败了，则将减去的数量加回去）。

ESCROW 方法适用前面所描述的情况，在这种情况下，更新是一种特殊形式，可以是完全随意的。系统必须提供一种新的更新语句（如“减去 x，当且仅当当前值大于 y”）。该语句通过将减少量放入“约定（escrow）”中执行更新，在事务结束时将该语句从“约定”中取出（如果事务以 COMMIT 结束，则提交修改；否则，在以 ROLLBACK 结束时，则将各数量放回原处）。

论文中描述了可使用 ESCROW 方法的大量示例。IBM 的 IMS Fast Path Version 支持该技术。这一技术可以作为［16.16］中的乐观并发控制的一个特例（但需注意，对提供特殊更新语句这一“特殊”特点是有争议的）。

［16.18］ Christos Papadimitriou：*The Theory of Database Concurrency Control*. Rockville, Md.：Computer Science Press (1986).

侧重于形式化理论的一本教科书。

［16.19］ Daniel J. Rosencrantz, Richard E. Stearns, and Philip M. Lewis Ⅱ：“System Level Concurrency Control for Distributed Database Systems,”*ACM TODS 3*, No. 2 (June 1978).

［16.20］ Kenneth Salem, Hector Garcia-Molina and Jeannie Shands：“Altruistic Locking,”*ACM TODS 19*, No. 1 (March 1994).

给出了一个两段锁的扩展协议。基于该扩展协议，如果事务 *A* 已经用完某些锁定的数据但根据 2PL 协议又不能释放数据上的锁时，*A* 可以将数据“捐献”给系统，进而允许其他事务获得在这些数据上的锁。这时称 *B* “跟随”（in the wake of）*A*，而所定义的协议用来阻止一个事务看到它的跟随事务所做的任何更新。利他锁（该术语源于“捐献”数据而使其他事务受益，而不是捐献者事务）能够提供较常规 2PL 更大的并发度。

［16.21］ Abraham Silberschatz, Henry F. Korth, and S. Sudarshan：*Database System Concepts* (4th ed.). New York, N. Y.：McGraw-Hill (2002).

这本关于数据库管理的课本有对于事务管理问题的精彩描述（包括恢复和并发）。

［16.22］ Robert H. Thomas：“A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases,”*ACM TODS 4*, No. 2 (June 1979).

请见参考文献［16.3］的注解。

［16.23］ Alexander Thomasian：“Concurrency Control：Methods, Performance, and Analysis,”*ACM Comp. Surv. 30*, No. 1 (March 1998).

对大量并发控制的不同的算法的性能进行了细致的研究。

# 第五部分　高级专题

在本书第二部分我们曾说过，关系模型是现代数据库技术的基础。该部分所介绍的内容基本仅限于基础范畴。但是除了第二部分介绍的关系模式的内容以外，数据库技术还有很多其他的内容。对于学习数据库的学生和专业人员，要想充分掌握数据库这门学科，还有很多知识需要学习（接下来从第三、四部分的讨论将明确看出这一点）。下面我们将进一步讨论数据库领域的高级专题。这些内容包括：

- 安全性（第 17 章）
- 优化（第 18 章）
- 信息空缺（第 19 章）
- 类型继承（第 20 章）
- 分布式数据库（第 21 章）
- 决策支持（第 22 章）
- 时态数据库（第 23 章）
- 基于逻辑的数据库（第 24 章）

以上内容的次序并无定规，读者也可以有选择地阅读。

# 第17章 安 全 性

## 17.1 引言

数据安全性问题常与数据完整性问题混淆，但是实际上这两个概念是不同的。安全性用于保护数据以防止不合法的使用；完整性则与数据的正确性相关。[⊖] 通俗地讲：

- **安全性**（security）保护数据以防止不合法用户故意造成的破坏。
- **完整性**（integrity）保护数据以防止合法用户无意中造成的破坏。

换言之就是：安全性确保用户被允许做其想做的事情；完整性确保用户所做的事情是正确的。

当然这两个概念也有相似点：系统应对用户不能违背的约束了如指掌；这些约束必须用合适的语言给出（通常由 DBA 指定），而且必须通过系统目录进行维护；DBMS 必须监控用户的操作以确定约束发挥了效用。在本章中，我们专门讨论安全性问题（完整性已在第 9 章以及其他章节进行了详细讨论）。注意：将这两个概念分两章讨论的主要原因在于：我们认为完整性是很基本的概念，而安全性是第二位的问题，尽管它也有着非常重要而实际的价值（尤其是在目前因特网大量接入、电子商务广泛应用以及一些类似的情况下，安全性问题显得更加重要）。

安全性问题涉及许多方面，如：

- 法律、社会以及道德问题（例如：请求者对其请求的信息是否具有合法权利?）；
- 物理控制（例如：计算机或终端所在房间是否上锁或受到保护?）；
- 政策问题（例如：拥有系统的企业如何控制使用者对数据的存取?）；
- 可操作性问题（例如：如果某个密码方案被采用，密码自身又如何保密？每隔多久要更换密码?）；
- 硬件控制（例如：处理单元是否具备安全特性，如存储保护键或保护操作模式?）；
- 操作系统支持（例如：底层操作系统在退出时是否抹去了主存和磁盘上的内容？对日志恢复又如何呢?）；

最后还包括：

- 数据库系统需专门考虑的问题（例如：数据库系统是否具有数据所有权的概念?）。

在本章中的绝大部分我们将只就上述的最后一类问题展开讨论。

现代的 DBMS 通常采用自主存取控制和强制存取控制这两种方法来解决安全性问题，有的只提供其中的一种方法，有的两种都提供。无论采用哪种存取控制方法，需要保护的数据单元或数据对象包括从整个数据库到某个元组的某个部分。两者的区别为：

- 在**自主存取**控制方法（discretionary control）中，用户对不同的数据对象具有不同的存取权限（也称为**优先权**），而且没有固定的关于哪些用户对哪些数据对象具有哪些存取权限的限制（例如：用户 *U*1 能看到 *A* 但看不到 *B*，而用户 *U*2 能看到 *B* 但看不到 *A*）。因此自主存取控制非常灵活。
- 在**强制存取**控制方法（mandatory control）中，每一个数据对象被标以一定的**密级**（classification level），每一个用户也被授予一个**许可证**级别（clearance level）。对于任意一个对象，只有具有合法许可证的用户才可以存取。因此，强制存取控制本质上具有分层特点，且相对比较严格（例如：如果用户 *U*1 能看到 *A* 但看不到 *B*，这说明 *B* 的密级高于 *A*，因此不存在用户 *U*2 能看到 *B* 但看不到 *A*）。

我们将在 17.2 节讨论自主存取控制，在 17.3 节讨论强制存取控制。

不管我们采用的是自主存取控制方法还是强制存取控制方法，所有有关哪些用户可对哪些数据对象进行操作的决定都是政策决定，而非技术问题。这显然超出了 DBMS 的权限，DBMS 只

---

⊖ 请参考第 9 章的内容来理解所谓“正确性”的含义。

是实施这些决定。这就要求：

- 政策决定的结果（a）需为系统所知（这可通过使用某种合适的语言定义**安全性约束**条件（security constraints）来实现）；（b）需被系统记住（这可通过将那些约束条件保存在目录中实现）。
- 必须采用一定的方式检查存取请求是否违背目录中适用的安全性约束（通常用“存取请求”表示涉及的请求操作、请求对象以及请求用户），这主要通过 DBMS 的**安全性子系统**（security subsystem）或者**授权子系统**（authorization subsystem）实现。
- 为了决定哪些安全性约束适用于给定的存取请求，系统必须能识别请求的来源，也就是请求用户。因此，当用户登录系统时必须给出用户 ID 以及**口令**（password）。口令只能为系统和拥有该用户 ID 的合法用户所知。密码验证的过程，也就是检查用户是不是他所提供 ID 的真正所有者这一过程，被称作**身份鉴定**（authentication）。注意：这里顺便提一下，目前可用的其他鉴定技术要比简单的密码验证复杂得多，其中包括一系列的生物设备，如指纹阅读器、视网膜扫描仪、手形机、语音识别器、签字识别器等。这些设备都可以有效地被用于验证“没有人能够窃取的个人身份特征”[17.6]。

关于用户 ID，请注意一定数量的不同用户可能共享同一 ID。这种情况下系统支持**用户组**（user group），并允许组内每一用户共享对同一数据对象的相同存取权限。将用户加入用户组以及将用户从用户组删除的操作应与指定用户组对哪些数据对象拥有哪些权限的操作分离。参考文献 [17.11] 描述了一个用户组嵌套的系统，并提到“将用户分成有层次的用户组，提供了强有力的管理有着大量用户以及数据对象的大型系统的工具”。但是注意用于维护用户组内有哪些用户的记录所在的最佳位置还是系统目录（或为数据库自身），并且这些记录本身也受限于某些安全控制。

## 17.2 自主存取控制

上一节已提到，大多数 DBMS 采用自主存取控制和强制存取控制这两种方法中的任意一种，或两者都采用。准确地说，大多数系统应该支持自主存取控制，有些系统同时支持强制存取控制；实际系统中支持最多的是自主存取控制，因此我们首先对其进行讨论。

前面已经提到，需要支持（自主）安全性约束定义的语言，显然，说明允许的内容要比禁止的内容容易，因此定义语言通常支持的不是安全性约束的定义，而是有关**授权**（authority）的定义（授权实际上就是约束的对立面，即一旦被授权，就不允许施以约束）。我们首先介绍一种定义授权的语言。[⊖] 下面是一简单的例子：

```
AUTHORITY SA3
   GRANT RETRIEVE { S#, SNAME, CITY }, DELETE
   ON    S
   TO    Jim, Fred, Mary ;
```

该例子说明了授权包括四个重要的部分：

1）**名字**（例子中为 SA3），授权将以该名字登记到目录。

2）一种或多种**权限**（例子中为 RETRIEVE——仅在某些属性上，以及 DELETE），通过 GRANT 子句指定。

3）**关系变量**是授权施以的数据对象，由 ON 子句指定。

4）一组**用户**（准确地说应该是用户 ID）对指定的关系变量拥有指定的权限，由 TO 子句指定。

该语句的语法如下：

```
AUTHORITY <authority name>
   GRANT  <privilege commalist>
   ON     <relvar name>
   TO     <user ID commalist> ;
```

---

⊖ 在参考文献 [3.3] 中所定义的语言 *Tutorial D* 并不包括任何对授权定义的措施，本节假定的语言可看作反应了 *Tutorial D* 的精华。

解释：< *authority* name >、< relvar name > 和 < user ID commalist > 不言自明，ALL 作为合法“用户 ID”，指所有的用户。< privilege > 如下：

```
RETRIEVE [ ( <attribute name commalist> ) ]
INSERT   [ ( <attribute name commalist> ) ]
DELETE
UPDATE   [ ( <attribute name commalist> ) ]
ALL
```

RETRIEVE（无限制）、INSERT（无限制）、DELETE 以及 UPDATE（无限制）无须加以说明（也许还是要解释一句，对于 RETRIEVE 权限除了要提及查询操作本身之外，某些情况下如在一个视图定义或完整性约束中，还需要提及相关的对象）。如果 RETRIEVE 有属性名列表，则说明 RETRIEVE 权限只适用于指定的属性，INSERT、UPDATE 也具有类似的含义。ALL 意味着所有权限：RETRIEVE（所有属性）、INSERT（所有属性）、UPDATE（所有属性）以及 DELETE。注意：为简便起见，在此我们没有讨论是否需要特权以执行一般的关系分配操作，另外我们讨论的范围也局限在**数据操纵**（data manipulation）操作上。实际上，合法权检查机制还要检查许多其他的操作，如定义与删除关系变量的操作，定义与删除授权自身的操作，等等。限于篇幅，我们省略了对这些操作所做的细致讨论。

如果用户尝试对某一数据对象进行某种操作，但不具有访问权限，将会出现怎样的情形呢？最简单的反应显然是拒绝该尝试（当然还应给出合适的诊断信息），这是实际系统中最基本的要求，所以我们不妨将之作为默认处理。在更多敏感的情形下，其他的一些反应可能更合适，例如中止程序的运行或锁住用户的键盘。将这些尝试记录在特殊的日志——**威胁监控**（threat monitoring）——中或许更可取，这些信息可用以对破坏系统安全性的尝试进行分析，而且自身能抵制不合法的渗透（可参考本节最后关于审计追踪的讨论）。

当然，我们也需要一种方式来删除授权：

```
DROP AUTHORITY <authority name> ;
```

为简化起见，我们假设删除关系变量时将自动删除使用于该关系变量上的所有授权信息。

下面是一些关于授权的例子，大部分不言自明：

```
1) AUTHORITY EX1
      GRANT RETRIEVE ( P#, PNAME, WEIGHT )
      ON    P
      TO    Jacques, Anne, Charley ;
```

用户 Jacques、Anne 和 Charley 所看到的是基关系变量 *P* 的“垂直子集”，该例是值无关授权的例子。

```
2) AUTHORITY EX2
      GRANT RETRIEVE, DELETE, UPDATE ( SNAME, STATUS )
      ON    LS
      TO    Dan, Misha ;
```

LS 是视图（见第 10 章的图 10-4—“London 供应商”）。用户 Dan 和 Misha 看到的是关系变量 *S* 的“水平子集”，该例是**值依赖**授权的例子。注意：虽然 Dan 和 Misha 能删除 *S* 的部分元组（通过视图 LS），但不能插入元组，也不能对属性 S#或 CITY 进行更新。

```
3) VAR SSPPO VIEW
     ( S JOIN SP JOIN ( P WHERE CITY = 'Oslo' ) { P# } )
                                          { ALL BUT P#, QTY } ;

   AUTHORITY EX3
      GRANT RETRIEVE
      ON    SSPPO
      TO    Lars ;
```

该例也是值依赖授权的例子：用户 Lars 可以查询供应商的信息，但只能是供应存储在 Oslo 的零件的供应商。

```
4) VAR SSQ VIEW
       SUMMARIZE SP PER S { S# } ADD SUM ( QTY ) AS SQ ;

   AUTHORITY EX4
      GRANT RETRIEVE
      ON    SSQ
      TO    Fidel ;
```

用户 Fidel 可查看每个供应商的总发货量，但不能查看到每次发货量，用户只能查看到基于底层基本数据的统计汇总信息。

```
5) AUTHORITY EX5
      GRANT RETRIEVE, UPDATE { STATUS }
      ON    S
      WHEN  DAY () IN { 'Mon', 'Tue', 'Wed', 'Thu', 'Fri' }
            AND NOW () ≥ TIME '09:00:00'
            AND NOW () ≤ TIME '17:00:00'
      TO    ACCOUNTING ;
```

这里我们对 AUTHORITY 的语法作了扩展，增加了 WHEN 子句用以说明特定的“环境控制”；我们也假设系统支持两个无操作数的操作（niladic operator）——也就是说，操作符没有带有明显的操作数——DAY( )和 NOW( )。授权 EX5 保证了供应商的状态值只能由用户“ACCOUNTING”（假定是 accounting 部门的任意一个职员）在工作日的工作时间更改。这通常称为**环境依赖**（context-dependent）授权，因为存取请求是否被准许依赖于环境，如该例中的环境指的是一周中的某些特定的天和一天中某段特定时间的组合。

其他系统应该支持且对环境依赖授权有用的内置操作的例子有：

```
TODAY()     :  Value = the current date
USER()      :  Value = the ID of the current user
TERMINAL() :  Value = the ID of the originating terminal
                      for the current request
```

现在你可能已意识到，所有的授权是“或”在一起使用的。换句话说，如果至少存在一个授权准许给定的存取请求，该请求就是可接受的。注意：举个例子来说，如果（a）一个授权允许用户 Nancy 检索零件的颜色；（b）另一授权允许她检索零件的重量；这并不意味着她能同时检索零件的颜色和重量（这种组合需要特别授权）。

最后提一点，用户只能做定义的授权明确允许的事情，任何未明确授权的事情都隐含是不合法的。

**1. 修改请求**

为了解释本节中已经提到的概念，我们简单介绍 Ingres 原型系统的安全机制以及查询语言 QUEL，因为其采用了一种巧妙的方法。基本原理就是系统对给定的 QUEL 请求在其执行前进行自动地修改，使其不违背安全性约束。举个例子：假定用户 *U* 只允许检索存储在 London 的零件：

```
DEFINE PERMIT RETRIEVE ON P TO U
       WHERE  P.CITY = "London"
```

（在后面将给出 DEFINE PERMIT 操作的详细解释）假设用户 *U* 发出如下的 QUEL 请求：

```
RETRIEVE ( P.P#, P.WEIGHT )
WHERE      P.COLOR = "Red"
```

利用存储在目录中的允许对关系变量 *P* 和用户 *U* 存取的条件，系统将用户的请求自动修改如下：

```
RETRIEVE ( P.P#, P.WEIGHT )
WHERE      P.COLOR = "Red"
AND        P.CITY  = "London"
```

显然，这种修改不可能违背安全性约束。注意，这个修改过程是“悄悄进行的”：用户 *U* 并

不知道系统执行的语句与其请求并不完全相同，因为数据本身就是敏感的，用户 *U* 无权知道非 London 地区的零件。

上述修改请求的过程实际上与视图实现［10.12］以及完整性约束［9.23］（尤其在 Ingres 原型中）的技术是一致的，因此该方案的优点在于实现简单——许多必要的代码已存在于原来的系统；效率高——安全措施的实施在编译时，而不是在运行时，至少部分如此；另外一个优点就是能满足某用户对同一关系变量的不同部分具有不同的存取权限的需求（17.6 节中对这个问题将有一个专门的解释）。

该方案的缺点在于不能处理所有安全性约束。举个简单的例子，假设用户 *U* 根本就不具有存取关系变量 *P* 的权限，根据上述修改过程，不存在 RETRIEVE 的修改形式能说明关系变量 *P* 存在，因此很可能会有另外的错误消息代替“你无权存取该关系变量”。系统可能会“撒谎”说“不存在该关系变量”，或者，好一点的情况是说“不存在该关系变量或者你无权存取它”。

下面是 DEFINE PERMIT 的语法：

```
DEFINE PERMIT <operation name commalist>
       ON     <relvar name> [ ( <attribute name commalist> ) ]
       TO     <user ID>
     [ AT     <terminal ID commalist> ]
     [ FROM   <time> TO <time> ]
     [ ON     <day>  TO <day>  ]
     [ WHERE  <bool exp> ]
```

该语句与 AUTHORITY 非常类似，除了增加了对 WHERE 子句的支持（AT、FROM、ON 子句全部被我们的 WHEN 子句所包含）。参考下面的例子：

```
DEFINE PERMIT RETRIEVE, APPEND, REPLACE
       ON     S ( S#, CITY )
       TO     Joe
       AT     TTA4
       FROM   9:00 TO 17:00
       ON     Sat TO Sun
       WHERE  S.STATUS < 50
       AND    S.S# = SP.P#
       AND    SP.P# = P.P#
       AND    P.COLOR = "Red"
```

注意：QUEL 中的 APPEND 和 REPLACE 分别类似于我们的 INSERT 和 UPDATE。

**2. 审计追踪**

永远不要认为安全性系统是坚不可摧的，潜在的渗透者总能想方设法突破控制，尤其是当其因此能获得很高的利益时。如果数据具有很高的敏感性，数据处理过程很重要，审计追踪就非常必要了。如果你怀疑数据库中的数据遭到篡改，就可以通过审计追踪检查用户对数据库究竟执行了哪些操作，并可确认操作都受到了控制，或可帮助确定误操作的人员。

审计追踪本质上是一特殊的文件或数据库，系统可自动记录用户对数据执行的所有操作。在有些系统中，审计追踪物理上与恢复日志合二为一（参见第 15 章），而有的系统中两者分别存放。无论哪种方式，用户都应该能利用规则的查询语言解释审计追踪（当然用户必须具有权限）。通常的审计追踪包含如下的信息：

- 请求（源文本）
- 发出操作调用的终端
- 发出操作调用的用户
- 操作日期和时间
- 操作作用的关系变量、元组、属性
- 镜像前（旧值）
- 镜像后（新值）

正如本节前面所提到的，审计追踪维护的信息必须足够充分以防止某些情形下的不怀好意的渗透者。

## 17.3 强制存取控制

军方和政府的数据具有很高的敏感性，通常具有静态的严格的分层结构（classification structure），强制存取控制对于存放这样的数据的数据库非常适用。如17.1节简要介绍的那样，该方法的基本思想在于每个数据对象具有一定的**密级**（classification level），如：绝密、机密、秘密等；每个用户具有一定的**许可证级别**（clearence level）。密级和许可证级别都是严格有序的，如：绝密 > 机密 > 秘密。Bell 和 La Padula [17.3] 采用如下的简单规则：

1）用户 $i$ 可以查询对象 $j$，当且仅当 $i$ 的许可证级别大于或等于 $j$ 的密级（简单规则）；

2）用户 $i$ 可以更新对象 $j$，当且仅当 $i$ 的许可证级别等于 $j$ 的密级（星规则）。

规则1的意义是明显的，而规则2需要一点说明。规则2的另一种表述为用户 $i$ 所写的对象自动获得的密级等于其许可证级别，该规则是为了防止具有较高级别的用户将该级别的数据复制到较低级别的文件中。注意：如果只是纯粹的“写”（INSERT）操作，写规则可以弱化为：当且仅当 $i$ 的许可证级别小于或等于 $j$ 的密级，用户 $i$ 可以“写”对象 $j$，但是用户可能无权读所写的对象，因此，这样的弱化不很现实。

20世纪90年代早期强制存取控制引起了数据库领域的注意，因为美国国防部要求其所购买的所有系统都必须支持这样的控制，这就促使各大DBMS厂商竞相提供这样的支持。美国国防部颁布的“橘皮书”[17.21]和“紫皮书”[17.22]对强制存取控制作了全面的描述和定义，“橘皮书”定义了任意“可信计算基”（Trusted Computing Base，简称TCB）应当遵从的一系列安全性要求；而“紫皮书”则定义了这些要求在数据库系统中相应的解释。

上述两份文献给出了通用的安全性分级模式，共定义了四类安全级别：D、C、B和A，由D类到A类级别依次增高。D类提供最小（minimal）保护，C类提供自主（discretionary）保护，B类提供强制（mandatory）保护，A类提供验证（verified）保护。我们对级别C，B和A做一介绍。

- **自主保护**：C类分为两个子类C1和C2，C1安全级别低于C2。每个子类都支持自主存取控制，即存取权限由数据对象的所有者决定（见17.2节的介绍）。

1）C1子类对所有权与存取权限加以区分，虽然它允许用户拥有自己的私有数据，但仍然支持共享数据的概念。

2）C2子类还要求通过注册、审计以及资源隔离以支持责任说明（accountability）。

- **强制保护**：B类适用于强制控制的安全级别。它分为三个子类B1、B2和B3，B1安全级别最低，B3最高。

1）B1子类要求“标识化安全保护”，即要求每个数据对象都必须标以一定的密级，如机密、秘密等。同时还要求安全策略的非形式化说明。

2）B2子类要求安全策略的形式化（formal）说明，能识别并消除隐蔽通道（covert channel）。隐蔽通道的例子有（a）从合法查询的结果中推断出不合法的查询的结果（见17.4节）；（b）通过合法计算所用的时间推断出敏感信息（见参考文献[17.14]的解释）。

3）B3子类要求审计和恢复支持以及指定的安全管理者。

- **验证保护**：A类作为最高的安全级别，要求有一套数学证明来保证安全机制的相容性并足以支持给定的安全策略。

有些DBMS产品现在提供B1级强制存取控制以及C2级自主存取控制。术语：支持强制存取控制的DBMS也称为**多级安全系统**（multi-level secure system）[17.15，17.18，17.23]或**可信系统**（trusted system）[17.19，17.21，17.22]。

**多级安全**

假设要对关系变量 $S$ 进行强制存取控制，为简化起见，假设要控制存取的数据单元是元组，则每个元组需标以密级，如图17-1所示（4 = 绝密，3 = 机密，2 = 秘密）。

S

| S# | SNAME | STATUS | CITY | LEVEL |
|---|---|---|---|---|
| S1 | Smith | 20 | London | 2 |
| S2 | Jones | 10 | Paris | 3 |
| S3 | Blake | 30 | Paris | 2 |
| S4 | Clark | 20 | London | 4 |
| S5 | Adams | 30 | Athens | 3 |

图17-1 具有密级的关系变量 $S$（示例）

假设用户 *U3* 和 *U2* 的许可证级别分别为 3 和 2，那么 *U3* 和 *U2* 看到的 *S* 是不一样的。若请求查询所有的供应商，*U3* 能查得四个元组：*S1*、*S2*、*S3* 和 *S5*；*U2* 只查得两个元组：*S1* 和 *S3*，*U3* 和 *U2* 都无法查得元组 *S4*。

可从**修改请求**（request modification）的角度考虑上述问题。考虑这样的查询“查找 London 的供应商”：

```
S WHERE CITY = 'London'
```

系统将请求修改为：

```
S WHERE CITY = 'London' AND LEVEL ≤ user clearance
```

更新操作类似，举个例子，用户 *U3* 不知道元组 *S4* 存在，因此，对 *U3* 来说，下面的 INSERT 语句是合理的：

```
INSERT S RELATION { TUPLE { S#     S# ('S4'),
                            SNAME  NAME ('Baker'),
                            STATUS 25,
                            CITY   'Rome' } } ;
```

系统必须拒绝该插入请求，但这样做实际上使用户意识到 *S4* 在数据库中是存在的。因此系统接受该请求，但将其修改为：

```
INSERT S RELATION { TUPLE { S#     S# ('S4'),
                            SNAME  NAME ('Baker'),
                            STATUS 25,
                            CITY   'Rome',
                            LEVEL  3 } } ;
```

这样，供应商的主码实际上不是 {S#}，而是 {S#, LEVEL}。注意：这里假设只有一个候选码，因此可将之作为主码。

供应商关系变量可看作多级变量（multi-level relvar），“相同”数据对不同用户来说，实际并不相同被称作多重实例（polyinstantiation），如查询供应商 *S4* 的请求将对具有绝密、机密以及秘密等不同安全级别的用户返回不同的结果。

DELETE 和 UPDATE 也是类似处理方式；这里我们省略了细节的说明，但在本章最后的几个参考文献中将针对这个问题进行更加深入的讨论。回答这样一个问题：你认为上面的处理方式是否违背了信息原则（The Information Principle）？请证明你的想法。

## 17.4　统计数据库

**统计数据库**允许用户查询聚集类型的信息（例如合计、平均值等），但是不允许查询单个记录信息。例如，查询“雇员的平均工资是多少?”是合法的，但是却不允许查询“雇员 Mary 的工资是多少?”。

在统计数据库中存在着特殊的安全性问题，即可能存在着隐蔽的信息通道，使得用户可以从合法的查询中推导出不合法的信息。正如参考文献［17.8］指出的：“综合信息总是带有少量的原始信息，利用足够的综合信息就可能获得原始信息，这也称作机密信息的推断。”随着数据仓库应用得越来越广泛，这个问题也变得越来越严重（参见第 22 章）。

假定数据库中只有一个关系变量 STATS（如图 17-2 所示），所有的属性都简单地定义在字符串或数字上；用户 *U* 只具有执行统计信息查询的权限，并且打算查出 Alf 的工资，其中 *U* 知道 Alf 是程序员且为男性。

考虑以下查询：[⊖]

---

⊖ 为写起来方便，本节的查询都使用 *Tutorial D* 的简化格式来表示。例如，查询 1 中的表达式 COUNT（X），完整地写应该是 EXTEND TABLE_DEE ADD COUNT（X）AS RESULT1。

| NAME | SEX | CHILDREN | OCCUPATION | SALARY | TAX | AUDITS |
|---|---|---|---|---|---|---|
| Alf | M | 3 | Programmer | 50K | 10K | 3 |
| Bea | F | 2 | Physician | 130K | 10K | 0 |
| Cyn | F | 0 | Programmer | 56K | 18K | 1 |
| Dee | F | 2 | Builder | 60K | 12K | 1 |
| Ern | M | 2 | Clerk | 44K | 4K | 0 |
| Fay | F | 1 | Artist | 30K | 0K | 0 |
| Guy | M | 0 | Lawyer | 190K | 0K | 0 |
| Hal | M | 3 | Homemaker | 44K | 2K | 0 |
| Ivy | F | 4 | Programmer | 64K | 10K | 1 |
| Joy | F | 1 | Programmer | 60K | 20K | 1 |

图 17-2 关系变量 STATS（样本值）

```
1) WITH ( STATS WHERE SEX = 'M' AND
          OCCUPATION = 'Programmer' ) AS X :
   COUNT ( X )
```

结果：1。

```
2) WITH ( STATS WHERE SEX = 'M' AND
          OCCUPATION = 'Programmer' ) AS X :
   SUM ( X, SALARY )
```

结果：50K。

显然，尽管用户 $U$ 的统计信息查询都是合法的，数据库的安全仍受到了危害。从上例可看出，如果用户能找到一个布尔表达式识别出某个个体，有关该个体的信息将不再安全。这表明当综合查询结果集的基数低于某个下限 $b$ 时系统应拒绝响应。同样，当综合查询结果集的基数高于上限 $n-b$ 时系统也应拒绝响应（$n$ 是关系的基数），从下面的查询可看出，当基数很高时数据库的安全同样会受到危害。

```
3) COUNT ( STATS )
```

结果 12。

```
4) WITH ( STATS WHERE NOT ( SEX = 'M' AND
          OCCUPATION = 'Programmer' ) ) AS X :
   COUNT ( X )
```

结果：11；12 - 11 = 1。

```
5) SUM ( STATS, SALARY )
```

结果：728K。

```
6) WITH ( STATS WHERE NOT ( SEX = 'M' AND
          OCCUPATION = 'Programmer' ) ) AS X :
   SUM ( X, SALARY )
```

结果：678K；728K - 678K = 50K。

不幸的是，简单地将综合查询结果集的基数 $c$ 限定在 $b \leqslant c \leqslant n-b$，并不足以避免数据库的安全受到危害。考虑图 17-2，假定 $b=2$，当且仅当 $2 \leqslant c \leqslant 8$ 时查询请求才被响应，布尔表达式

```
SEX = 'M' AND OCCUPATION = 'Programmer'
```

不再合法。但是考虑下面的查询：

```
7) WITH ( STATS WHERE SEX = 'M' ) AS X :
   COUNT ( X )
```

结果：4。

```
8) WITH ( STATS WHERE SEX = 'M' AND NOT
                    ( OCCUPATION = 'Programmer' ) ) AS X :
   COUNT ( X )
```

结果：3。

从查询7和8，用户 *U* 可推断出只存在一个男性程序员，其必定是 Alf，由此可通过下面的语句查出 Alf 的工资：

```
9) WITH ( STATS WHERE SEX = 'M' ) AS X :
   SUM ( X, SALARY )
```

结果：328K。

```
10) WITH ( STATS WHERE SEX = 'M' AND NOT
                   ( OCCUPATION = 'Programmer' ) ) :
    SUM ( X, SALARY )
```

结果：278K；328K－278K＝50K。

布尔表达式 SEX = 'M' AND NOT OCCUPATION = 'Programmer' 称作 Alf 的**个体追踪者**（individual tracker）[17.8]，这是因为通过它用户才能追踪获得个体信息。一般地讲，如果用户知道识别某个体 *I* 的某布尔表达式 *BE*，并且 *BE* 可表示为 *BE*1 AND *BE*2 这样的形式，那么 *BE*1 AND NOT *BE*2 就是 *I* 的个体追踪者（假设 *BE*1 和 *BE*1 AND NOT *BE*2 都是合法的，即两者的结果集基数都在上述的限定范围内）。原因在于 *BE* 所确定的集合等于 *BE*1 所确定的集合同 *BE*1 AND NOT *BE*2 所确定集合的差：

```
{ x : BE }  ≡  { x : BE1 AND BE2 }
            ≡  { x : BE1 } MINUS { x : BE1 AND NOT BE2 }
```

如图 17-3 所示。

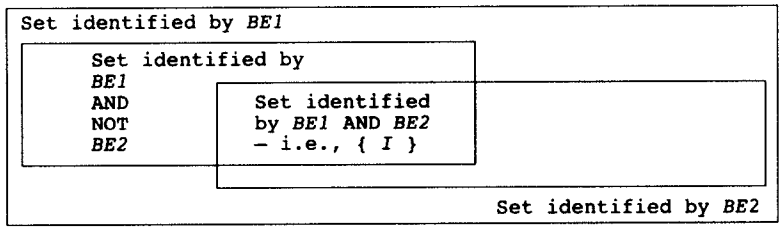



图 17-3　个体追踪者 *BE*1 AND NOT *BE*2

参考文献 [17.8] 概括了前面的内容，并指出对几乎任意一个统计数据库总能找到一个**通用追踪者**（general tracker）——这是相对于个体追踪者而言的。通用追踪者是这样的布尔表达式，它能用于获得任意不合法查询（包含不合法表达式的查询）的结果，而个体追踪者只对某些特殊的不合法查询有效。实际上，任何结果集基数 $c$ 满足 $2b \leqslant c \leqslant n-2b$ 的表达式都是通用追踪者，这里 $b$ 必须小于 $n/4$，通常适用于各种实际情况。一旦找到这样的追踪者，相应于不合法表达式 *BE* 的查询就可迎刃而解，下面的例子将对此进行解释。该例中假设 *BE* 的结果集基数小于 $b$，类似地，当 *BE* 的结果集基数大于 $n-b$ 时也是如此处理。注意，从定义可看出，若 *T* 是通用追踪者，则 NOT *T* 也是通用追踪者。

例：假设 $b=2$，则通用追踪者是任何结果集基数 $c$ 满足 $4 \leqslant c \leqslant 6$ 的表达式。仍假设用户 *U* 从外部资料获悉 Alf 是一个男性程序员，即不合法布尔表达式 *BE* 是（与前面一样）：

```
SEX = 'M' AND OCCUPATION = 'Programmer'
```

并且假设 *U* 试图查出 Alf 的工资。对通用追踪者使用两次，首先确定 *BE* 实际上唯一标识 Alf

(步骤2~4)，再确定 Alf 的工资（步骤5~7)。

步骤1：猜测一追踪者 *T*。这里选择的 *T* 为表达式

```
AUDITS = 0
```

步骤2：利用表达式 *T* 以及 NOT *T* 获得数据库中个体的总数。

```
WITH ( STATS WHERE AUDITS = 0 ) AS X :
COUNT ( X )
```

结果：5。

```
WITH ( STATS WHERE NOT ( AUDITS = 0 ) ) AS X :
COUNT ( X )
```

结果：5；5+5=10。

显然，我们所猜测的 *T* 就是通用追踪者。

步骤3：利用表达式 *BE* OR *T* 以及 *BE* OR NOT *T* 获得数据库中个体总数与满足不合法表达式 *BE* 的个体数目的总和。

```
WITH ( STATS WHERE ( SEX = 'M' AND
                     OCCUPATION = 'Programmer' )
                OR AUDITS = 0 ) AS X :
COUNT ( X )
```

结果：6。

```
WITH ( STATS WHERE ( SEX = 'M' AND
                     OCCUPATION = 'Programmer' )
                OR NOT ( AUDITS = 0 ) ) AS X :
COUNT ( X )
```

结果：5；6+5=11。

步骤4：从上面的结果可算出满足 *BE* 的个体数目是1（步骤3的结果减去步骤2的结果)，显然，*BE* 唯一标识了 Alf。

在步骤5和6中重复步骤2和3的查询，但使用 SUM 取代 COUNT。

步骤5：利用表达式 *T* 以及 NOT *T* 获得数据库中个体的工资总额。

```
WITH ( STATS WHERE AUDITS = 0 ) AS X :
SUM ( X, SALARY )
```

结果：438K。

```
WITH ( STATS WHERE NOT ( AUDITS = 0 ) ) AS X :
SUM ( X, SALARY )
```

结果：290K；438K+290K=728K。

步骤6：利用表达式 *BE* OR *T* 以及 *BE* OR NOT *T* 获得 Alf 的工资与工资总额的总和。

```
WITH ( STATS WHERE ( SEX = 'M' AND
                     OCCUPATION = 'Programmer' )
                OR AUDITS = 0 ) AS X :
SUM ( X, SALARY )
```

结果：488K。

```
WITH ( STATS WHERE ( SEX = 'M' AND
                     OCCUPATION = 'Programmer' )
                OR NOT ( AUDITS = 0 ) ) AS X :
SUM ( X, SALARY )
```

结果：290K；488K+290K=778K。

步骤7：从步骤6的结果中减去工资总额（步骤5的结果）即得 Alf 的工资。

结果：50K。

图 17-4 解释了通用追踪者：

```
{ x : BE }  ≡  ( { x : BE OR T } UNION { x : BE OR NOT T } )
                                MINUS { x : T OR NOT T }
```

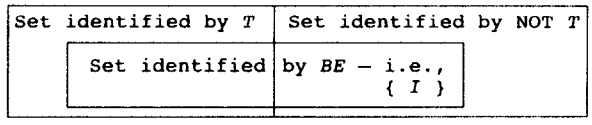



图 17-4　通用追踪者 $T$

如果最初的猜测是错的，也就是说，$T$ 实际上并不是一个通用追踪者，则表达式（*BE* OR *T*）与（*BE* OR NOT *T*）中至少有一个可能是不合法的。举个例子，如果 *BE* 和 *T* 的结果集基数分别为 $p$ 和 $q$，并满足 $p < b$，$b \leqslant q \leqslant 2b$，则（*BE* OR NOT *T*）的结果集基数大于 $n - b$ 是有可能的。因此必须再猜测通用追踪者并检测。参考文献［17.8］表明找到通用追踪者的过程并不困难。本例中，最初的猜测是正确的，*T* 是一个通用追踪者（因为其结果集基数为 5），因此步骤 3 的查询都是合法的。

总结：通用追踪者“几乎总是”存在的，而且通常容易找到并易于使用。事实上，通过猜测迅速找到追踪者是可能的［17.8］。参考文献［17.8］指出，即使在通用追踪者不存在的情况下，对于特殊的查询通常也能找到特殊的追踪者。因此，在统计数据库中安全性确实是个问题。

问题如何解决？现在已经有了一些解决办法，但没有一个能完全令人满意。举个例子，一种方法是“数据交换”，即在元组间交换属性值，但不破坏整体的统计精确性。这样即使某个特定的值（如工资）被识别出来，仍然没有办法知道该值属于哪个个体。采用这种办法的困难在于确定能被交换的值的集合。其他方法采用了类似的限制形式。由此看来我们应同意参考文献［17.8］中的结论：“折衷的办法简单易行。对机密信息绝对保密的要求与对总体的任意子集采用精确的统计措施的要求是不可能同时满足的，要保证秘密可行至少要降低其中的一个要求。”

## 17.5　数据加密

前面的讨论都是认为恶意的攻击者将使用通常的系统设施存取数据库，现在考虑用户可能试图旁路系统的情况，如物理地取走数据库，在通信线路上窃听。对这样的威胁最有效的解决方法就是**数据加密**（data encryption），即以加密格式存储和传输敏感数据。

数据加密的术语有：**明文**（plaintext），即原始的或未加密的数据。通过**加密算法**（encryption algorithm）对其进行加密，加密算法的输入信息为明文和**密钥**（encryption key）；**密文**（ciphertext），明文加密后的格式，是加密算法的输出信息。加密算法是公开的，而密钥则是不公开的。密文不应为无密钥的用户理解，其用于数据的存储以及传输。

例：明文为字符串：

```
AS KINGFISHERS CATCH FIRE
```

（为简便起见，假定所处理的数据字符仅为大写字母和空格符）。假定密钥为字符串：

```
ELIOT
```

加密算法为：

1）将明文划分成多个密钥字符串长度大小的块（空格符以“+”表示）：

```
A S + K I   N G F I S   H E R S +   C A T C H   + F I R E
```

2）用 00 ~ 26 范围的整数取代明文的每个字符，空格符 = 00，A = 01，…，Z = 26：

```
0119001109  1407060919  0805181900  0301200308  0006091805
```

3）与步骤 2 一样对密钥的每个字符进行取代：

```
0512091520
```

4）对明文的每个块，将其每个字符用对应的整数编码与密钥中相应位置的字符的整数编码的和模27后的值取代：

```
0119001109   1407060919   0805181900   0301200308   0006091805
0512091520   0512091520   0512091520   0512091520   0512091520
----------   ----------   ----------   ----------   ----------
0604092602   1919152412   1317000720   0813021801   0518180625
==========   ==========   ==========   ==========   ==========
```

5）将步骤4的结果中的整数编码再用其等价字符替换：

```
F D I Z B    S S O X L    M Q + G T    H M B R A    E R R F Y
```

如果给出密钥，该例的解密过程很简单（**习题**：对上面的密文进行解密）。问题是对于一个恶意攻击者来说，在不知道密钥的情况下，利用相匹配的明文和密文获得密钥究竟有多困难？对于上面简单的例子，答案是相当容易的，但是，复杂的加密模式同样很容易设计出来。理想的情况是采用的加密模式使攻击者破解所付出的代价远远超过其所获得的利益。实际上，该目的适用于所有的安全性措施。这种加密模式的最终目标是：即使是该模式的发明者也无法通过相匹配的明文和密文获得密钥，也无法破解密文。

**1. 数据加密标准**

传统加密方法有两种，替换和置换。上面的例子采用的就是替换的方法：使用密钥将明文中的每一个字符转换为密文中的一个字符。而置换仅将明文的字符按不同的顺序重新排列。单独使用这两种方法的任意一种都是不够安全的，但是将这两种方法结合起来就能提供相当高的安全程度。*数据加密标准*（Data Encryption Standard，简称DES）就采用了这种结合算法，它由IBM制定，并在1977年成为美国官方加密标准［17.20］。

DES的工作原理为：将明文分割成许多64位大小的块，每个块用64位密钥进行加密，实际上，密钥由56位数据位和8位奇偶校验位组成，因此只有$2^{56}$个可能的密码而不是$2^{64}$个。每块先用初始置换方法进行加密，再连续进行16次复杂的替换，最后再对其施用初始置换的逆。第$i$步的替换并不是直接利用原始的密钥$K$，而是由$K$与$i$计算出的密钥$Ki$，详细算法请见参考文献［17.20］。

DES具有这样的特性，其解密算法与加密算法相同，除了密钥$Ki$的施加顺序相反以外。

随着计算机运算速度的提高和存储容量的增加，DES所采用的56位密码已经越来越引起人们的不满。于是在2000年，美国联邦政府采用了一个新的标准，叫做AES（Advanced Encryption Standard），该标准基于Rijndael算法［17.5］，使用的密码可以是128，192或者256位。仅128位的密码就已经显示出新标准比DES考虑了更多的安全性；通过参考文献［26.34］可以知道，如果“我们能够研制出一台可以在一秒钟内破译DES的计算机，那么这台计算机需要149万亿年的时间才能破译一个128位的AES密码”（对原文略有改动）。对此，参考文献［17.5］中有更加深入的介绍。

**2. 公开密钥加密**

我们前面已经提到DES并不是真正地安全。AES虽然更好一些，但许多人仍然认为如果不采用一些智能的方法，这种机制还是有可能被强制破解的。“*公开密钥*”加密方法使得这些传统加密技术过时了。公开密钥加密方法中，加密算法和加密密钥都是公开的，任何人都可将明文转换成密文。但是相应的*解密密钥*是保密的（公开密钥方法包括两个密钥，分别用于加密和解密），而且无法从加密密钥推导出来，因此，若未被授权，即使是加密者也无法进行解密。

公开密钥加密思想最初是由Diffie和Hellman［17.9］提出的，最著名的是Rivest、Shamir以及Adleman［17.17］提出的这种机制在实际中的工作原理，现在通常称为RSA机制（以三个发明者的首位字母命名），该方法基于以下的两个条件：

1）已找到确定一个数是不是质数的快速算法；

2）尚未找到确定一个合数的质因子的快速算法。

参考文献［17.12］给出了一个例子：确定一个130位的数是否为质数只花了大约7分钟，而

确定两个63位质数的乘积的两个质因子（在同一台机器）需要40 000 000 000 000 000年。⊖

RSA方法的工作原理如下：

1）任意选取两个不同的大质数 $p$ 和 $q$，计算乘积 $r=p*q$；

2）任意选取一个大整数 $e$，$e$ 与 $(p-1)*(q-1)$ 互质，整数 $e$ 用做加密密钥。注意：$e$ 的选取是很容易的，例如，所有大于 $p$ 和 $q$ 的质数都可用。

3）确定解密密钥 $d$，使 $d*e$ 能被 $(p-1)*(q-1)$ 整除。即：

```
d * e = 1 modulo (p-1) * (q-1)
```

根据 $e$、$p$ 和 $q$ 可以容易地计算出 $d$，算法可见参考文献［17.17］。

4）公开整数 $r$ 和 $e$，但是不公开 $d$；

5）将明文 $P$（假设 $P$ 是一个小于 $r$ 的整数）加密为密文 $C$，计算方法为：

```
C = P^e modulo r
```

6）将密文 $C$ 解密为明文 $P$，计算方法为：

```
P = C^d modulo r
```

参考文献［17.17］对该方法的工作原理进行了证明，即用 $d$ 确实可将密文 $C$ 复原为明文 $P$。然而只根据 $r$ 和 $e$（不是 $p$ 和 $q$）要计算出 $d$ 是不可能的。因此，任何人都可对明文进行加密，但只有授权用户（知道 $d$）才可对密文解密。

下面举一简单的例子对上述过程进行说明，在此我们选取较小的数字。

示例：选取 $p=3$，$q=5$，则 $r=15$，$(p-1)*(q-1)=8$。选取 $e=11$（大于 $p$ 和 $q$ 的质数），通过：

```
d * 11 = 1 modulo 8
```

计算出 $d=3$。

假定明文为整数13。则密文 $C$ 为：

```
C = P^e modulo r
  = 13^11 modulo 15
  = 1,792,160,394,037 modulo 15
  = 7
```

复原明文 $P$ 为：

```
P = C^d modulo r
  = 7^3 modulo 15
  = 343 modulo 15
  = 13
```

因为 $e$ 和 $d$ 互逆，公开密钥加密方法也允许采用这样的方式对加密信息进行**签名**，以便接收方能确定签名不是伪造的。假设 $A$ 和 $B$ 希望通过公开密钥加密方法进行数据传输，$A$ 和 $B$ 分别公开加密算法和相应的密钥，但不公开解密算法和相应的密钥。$A$ 和 $B$ 的加密算法分别是ECA和ECB，解密算法分别是DCA和DCB，ECA和DCA互逆，ECB和DCB互逆。

若 $A$ 要向 $B$ 发送明文 $P$，不是简单地发送ECB（$P$），而是先对 $P$ 施以其解密算法DCA，再用加密算法ECB对结果加密后发送出去。密文 $C$ 为：

```
C = ECB ( DCA ( P ) )
```

---

⊖ 即使这样，RSA方法仍有安全性问题。参考文献［17.12］发表于1977年，而在1990年，Lenstra和Manasse成功地将一个155位数字进行了因数分解［17.24］；他们估计，使用大约1000台计算机并行工作所能进行的计算量，相当于在单台计算机上以每秒100万条指令的速度运行273年的工作量。这155位的整数是第9个费马数 $2^{512}+1$（注意 $512=2^9$）。还可参阅参考文献［17.14］，里面给出了一种完全不同但成功的破解RSA的方法。

用户 *B* 收到 *C* 后，先后施以其解密算法 DCB 和加密算法 ECA，得到明文 *P*：

```
ECA ( DCB ( C ) )
  = ECA ( DCB ( ECB ( DCA ( P ) ) ) )
  = ECA ( DCA ( P ) )              /* DCB和ECB相互抵消 */
  = P                              /* ECA和DCA相互抵消 */
```

这样 *B* 就确定报文确实是从 *A* 发出的，因为只有当加密过程利用了 DCA 算法，用 ECA 才能获得 *P*，只有 *A* 才知道 DCA 算法，其他人，即使是 *B* 也不能伪造 *A* 的签名。

## 17.6 SQL 的支持

目前的 SQL 标准只支持自主存取控制。与安全性相关的 SQL 性质有两个：**视图机制**，可用于对未授权的用户隐藏敏感数据；**授权子系统**，允许具有特定权限的用户有选择地动态地将这些权限授予其他用户，并在以后回收这些权限。这两个性质将在下面讨论：

**1. 视图和安全性**

为了说明 SQL 中用于安全性目的的视图的使用方法，给出 17.2 节例 2～4 的视图例子对应的 SQL 表示，如下所示。

```
1) CREATE VIEW LS AS
          SELECT S.S#, S.SNAME, S.STATUS, S.CITY
          FROM   S
          WHERE  S.CITY = 'London' ;
```

视图定义了将被授权的数据，而授权是通过 GRANT 语句实现的，如：

```
GRANT SELECT, DELETE, UPDATE ( SNAME, STATUS )
ON    LS
TO    Dan, Misha ;
```

注意：在 SQL 中授权是通过 GRANT 语句而不是通过假定的“CREATE AUTHORITY”语句定义的，因此未对其命名，而完整性约束是必须命名的（见第 9 章）。

```
2) CREATE VIEW SSPPO AS
          SELECT S.S#, S.SNAME, S.STATUS, S.CITY
          FROM   S
          WHERE  EXISTS
                 ( SELECT * FROM SP
                   WHERE  EXISTS
                          ( SELECT * FROM P
                            WHERE  S.S# = SP.S#
                            AND    SP.P# = P.P#
                            AND    P.CITY = 'Oslo' ) ) ;
```

相应的 GRANT 语句：

```
GRANT SELECT ON SSPPO TO Lars ;
```

```
3) CREATE VIEW SSQ AS
          SELECT S.S#, ( SELECT SUM ( SP.QTY )
                         FROM   SP
                         WHERE  SP.S# = S.S# ) AS SQ
          FROM   S ;
```

相应的 GRANT 语句：

```
GRANT SELECT ON SSQ TO Fidel ;
```

17.2 节中的示例 5 是关于环境依赖的授权。SQL 支持各种无参数内置运算符，如 CURRENT_USER、CURRENT_DATE、CURRENT_TIME 等，这些运算符可用于定义环境依赖视图。注意：SQL 不支持原例 5 的 DAY( )运算符。例：

```
CREATE VIEW S_NINE_TO_FIVE AS
       SELECT S.S#, S.SNAME, S.STATUS, S.CITY
       FROM   S
```

```
WHERE  CURRENT_TIME ≥ TIME '09:00:00'
AND    CURRENT_TIME ≤ TIME '17:00:00' ;
```

相应的 GRANT 语句：

```
GRANT SELECT, UPDATE ( STATUS )
ON    S_NINE_TO_FIVE
TO    ACCOUNTING ;
```

注意：S_NINE_TO_FIVE 是一个很特别的视图！它的值随着时间的变化而变化，即使其内在的数据未改变。另外，包含内置操作符 CURRENT_USER 的视图定义对于不同的用户其值也不同。这样的“视图”与通常意义上的视图并不相同，事实上它们是参数化的视图。至少从概念上讲，允许用户定义自己的（潜在的参数化的）值依赖函数更可取，像 S_NINE_TO_FIVE 这样的“视图”就可看作这类函数。

前面的例子说明了视图机制“免费”提供了一种重要的安全性方法，称其“免费”是因为该机制在系统中本来用于其他用途。另外，许多授权检查，甚至是值依赖的检查，都可在编译时（而无须等到运行时）进行，这对性能改善有着很重要的意义。然而，基于视图的安全性方法偶尔也会遇到一点麻烦，尤其是当用户希望在同一时间对同一个表的不同子集拥有不同的权限时。举个例子：考虑这样的应用，扫描并显示所有 London 的零件，同时对其中红色的零件进行更新。

### 2. GRANT 和 REVOKE

视图机制用不同的方式将数据库划分为多个片段，使得未授权用户无法获取敏感信息，然而它并未说明授权用户在这些片段上能执行哪些操作。这个任务（前面例子中已涉及）将由 **GRANT** 语句来承担，在这里将进行详细讨论（不过我们还是会略去对一些更加深奥问题的讨论）。

首先注意任何一个对象的创建者都被自动授予了该对象上的所有权限。举个例子，基表 *T* 的创建者将被自动授予表 *T* 上的 SELECT、INSERT、DELETE、UPDATE、REFERENCES 以及 TRIGGER[○] 权限（下面将对这些权限一一解释）。而且这些权限都被授予了“with grant authority”，这表明拥有这些权限的用户还可以将这些权限转授给其他用户。

GRANT 语句的语法如下：

```
GRANT <privilege commalist>
      ON <object>
      TO <user ID commalist>
    [ WITH GRANT OPTION ] ;
```

解释：

1）合法的 < *privilege* > 有 USAGE、UNDER、SELECT、INSERT、DELETE、UPDATE、REFERENCES、TRIGGER 以及 EXECUTE。SELECT、INSERT、UPDATE 以及 REFERENCES 权限可定义于列级。注意：也可指定 ALL PRIVILEGES，但其语义不太直观（见参考文献[4.20]）。

- USAGE 权限用在特定的用户自定义类型上以便使用该类型。
- UNDER 权限（a）用在特定的用户自定义类型上以便创建这个类型的子类（b）用在特定的表上以创建其子表（可分别参见第 20 和 26 章）。
- SELECT，INSERT，DELETE 和 UPDATE 权限都已经讲过了。
- REFERENCES 权限用在特定的表上以给出其完整性约束（可以是任意的约束，而不仅仅用于参照完整性）。
- TRIGGER 权限用于特定的基表以在其上创建一个触发器。
- EXECUTE 权限用于特定的 SQL 例程以调用该例程。

2）合法的 < *object* > 为类型 < *type name* >，表 < *table name* >，以及对象 < *specific routine designator* >（用于 EXECUTE 权限），本书不做细节解释。注意：其中“TABLE”（实际是可选的）包括视图或基本表。

---

○ UNDER 权限大概也是（这个权限是有意义的），但是标准中并没有提到该权限。

3) <*user ID commalist*>可用关键字 PUBLIC 替代，意味着系统中所有的用户。注意：SQL 也支持用户自定义的**角色**（roles）；ACCOUNTING 就是一个例子，它代表 accounting 部门的所有成员。角色一旦被创建就可以像一个普通用户 ID 一样被赋予权限。而且，角色本身也可以像权限一样被赋给一个用户或者另一个角色。也就是说，角色是 SQL 用于支持用户群组的一种机制（参见 17.1 节）。

4) WITH GRANT OPTION，意味着将指定对象上的指定的权限授予指定的用户，同时带有**授权权限**，即指定的用户可将这些权限继续转授给其他用户。当然，要指定 WITH GRANT OPTION 首先要求发出 GRANT 语句的用户必须具备授权权限。

若用户 *A* 向用户 *B* 授予了权限，则也可从用户 *B* 收回权限。收回权限可通过 **REVOKE** 语句实现，其语法为：

```
REVOKE [ GRANT OPTION FOR ] <privilege commalist>
                            ON <object>
                            FROM <user ID commalist>
                            <behavior> ;
```

其中：a) GRANT OPTION FOR 意味着只有授权权限被收回；

b) <*privilege commalist*>、<*object*>以及<*user ID commalist*>与 GRANT 语句中的含义相同；

c) <*behavior*>或者是 RESTRICT，或者（通常）是 CASCADE。例如：

```
1) REVOKE SELECT ON S FROM Jacques, Anne, Charley RESTRICT ;
2) REVOKE SELECT, DELETE, UPDATE ( SNAME, STATUS )
          ON LS FROM Dan, Misha CASCADE ;
3) REVOKE SELECT ON SSPPO FROM Lars RESTRICT ;
4) REVOKE SELECT ON SSQ FROM Fidel RESTRICT ;
```

现在来看 RESTRICT 与 CASCADE：假设 *p* 是某对象上的某权限，假设用户 *A* 将 *p* 授给用户 *B*，*B* 又将其转授给用户 *C*。如果 *A* 从 *B* 回收权限 *p*，将会发生什么事情呢？首先，假设 REVOKE 成功地执行了。那么用户 *C* 所拥有的权限 *p* 将不再有效，因为其由用户 *B* 转授，而 *B* 不再拥有权限 *p*。RESTRICT 与 CASCADE 选项就是为了避免发生这种情形，其中如果有可能导致上述情形时，RESTRICT 将会使得 REVOKE 以失败告终；而 CASCADE 将会引起上述所有的权限都被回收。

删除一个类型、表、列或例程将自动从所有用户回收被删除对象上的所有权限。

## 17.7 小结

本章讨论了数据库**安全性**问题的不同方面。首先对安全性和完整性进行了比较：安全性确保用户被允许做其想做的事情；完整性确保用户所做的事情是正确的。换句话说，安全性即保护数据防止未授权的存取。

安全性由 DBMS 的**安全性子系统**实施，安全性子系统将对所有的存取请求检查存储在系统目录中的**安全性约束**（或者是授权）。首先讨论的是**自主存取**控制模式，在该模式下，对指定对象的存取将取决于该对象的所有者。自主模式下的每个授权都包括**名字**、**权限集**（RETRIEVE、INSERT 等）、相应的**关系变量**（即权限所施予的数据）以及**用户集**。这样的授权可被用于提供**值依赖**、**值无关**、**统计汇总**以及**环境依赖控制**。**审计追踪**可用于记录试图破坏安全性的行为。除此以外还简单讨论了**请求修改**这一用于自主模式的实现技术（该技术最早在 Ingres 原型系统中的 QUEL 语言中运用）。

**强制存取控制**模式中每个对象具有**密级**，而每个用户具有**许可证**级别。我们解释了该模式下的存取规则，对美国国防部在橘皮书和紫皮书中定义的安全性分级模式进行了概要介绍，并简单地讨论了**多级关系变量**和**多重实例**的思想。

**统计数据库**有其特有的安全性问题。统计数据库中有着大量的有关个体的相关敏感信息，只

向用户提供统计汇总信息，但通过**追踪者**的方式其安全性很容易受到破坏。事实上，由于数据仓库系统的出现，该问题应受到足够的重视（请参阅第 22 章）。

本章介绍了**数据加密**的基本思想：**替换**和**置换**，解释了**数据加密标准**（DES）和**高级加密标准**（AES），并描述了**公开密钥**加密方法的工作原理，特别给出了 **RSA**（质数）方法的一个例子。另外还介绍了**数字签名**的概念。

另外，给出了 SQL 中描述安全性的方法，用于**隐藏**信息的**视图**的使用，以及利用 **GRANT** 和 **REVOKE** 控制用户对数据对象拥有的权限（主要是基表和视图）。

最后必须强调一点，即使 DBMS 提供了大量的安全性控制措施，如果能避开这些措施，这些控制也是没用的。在 DB2 中，数据库是以操作系统文件的形式物理存储在磁盘上的，如果有可能通过传统的操作系统服务从传统的程序存取那些文件，DB2 的安全机制将失效。因此，DB2 与它的伴随系统（即底层的操作系统）协同工作可保证整个系统是安全的，其细节本章不予讨论。

## 习题

17.1　假定关系变量 STATS 与 17.4 节中的定义相同，如下所示：

```
STATS { NAME, SEX, CHILDREN, OCCUPATION, SALARY, TAX, AUDITS }
      KEY { NAME }
```

采用 17.2 节中的语言定义下面给定的授权：

a. 用户 Ford 对整个关系变量具有 RETRIEVE 权限；
b. 用户 Smith 对整个关系变量具有 INSERT 和 DELETE 权限；
c. 每个用户只对自己的元组具有 RETRIEVE 权限；
d. 用户 Nash 对整个关系变量具有 RETRIEVE 权限，但只对 SALARY 和 TAX 属性具有 UPDATE 权限；
e. 用户 Todd 只对 NAME、SALARY 和 TAX 属性具有 RETRIEVE 权限；
f. 用户 Ward 的 RETRIEVE 权限与 Todd 一样，而只对 SALARY 和 TAX 属性具有 UPDATE 权限；
g. 用户 Pope 具有对职业为 preacher 的元组的所有权限；
h. 用户 Jones 具有对职业为非特殊性质的元组的 DELETE 权限，这里非特殊性质的职业为人数超过 10 个的职业；
i. 用户 King 具有对每个职业最高和最低工资的 RETRIEVE 权限。

17.2　若考虑对定义和删除关系变量、定义和删除视图以及定义和删除权限等操作的控制，应如何扩展 AUTHORITY 定义的语法？

17.3　再考虑图 17-2，假设我们知道 Hal 是一个至少有两个孩子的家长。利用个体追踪者写出一系列统计查询以获得 Hal 的 Tax 值。就像 17.4 节一样，假定系统对于集合基数小于 2 或大于 8 的查询不响应。

17.4　重复 17.3 的问题，只是不采用个体追踪者，而采用通用追踪者。

17.5　对下面的密文进行解密，密文采用与 17.5 节“AS KINGFISHES CATCH FIRE”例子类似的方式，但是利用的是 5 字节的加密密码：

```
F N W A L
J P V J C
F P E X E
A B W N E
A Y E I P
S U S V D
```

17.6　通过 RSA 公开密钥加密方法进行工作，给定 $p=7$、$q=5$、$e=17$ 以及明文 $P=3$。

17.7　你能想出任何可能由加密引起的实现上的问题以及其他不利条件吗？

17.8　给出习题 17.1 的 SQL 解决方案。

17.9　写出将习题 17.8 的解决方案中所授予的权限删除的 SQL 语句。

## 参考文献

[17.1]　Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu: "Hippocratic Databases," Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong (August 2002).

文章摘要如下：“（我们）认为未来的数据库必须将保护数据隐私的职责作为数据库构建的

原则……我们阐明了……数据库系统的……关键保密原则。"

[17.2] Rakesh, Agrawal and Jerry Kiernan: "Watermarking Relational Databases," Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong (August 2002).

该文提出了一种可以对"水印"数据进行盗版(侵权)检测的机制。

[17.3] D. E. Bell and L. J. La Padula: "Secure Computer Systems: Mathematical Foundations and Model," MITRE Technical Report M74 - 244 (May 1974).

[17.4] Luc Bouganim and Philippe Pucheral: "Chip-Secured Data Access: Confidential Data on Untrusted Servers," Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong (August 2002).

文章摘要如下:"(这种模式通过)一个基于客户端的安全组件来加强数据的机密性和控制用户个人权限,这种组件在客户端和加密数据库之间起到一种协调的作用。该组件被嵌入到智能卡上以防止(恶意攻击)。"这种模式解决了加密数据库多面临的典型的问题,也就是数据必须以明文的格式存在于索引中。

[17.5] J. Daemen and V. Rijnen: "The Block Cipher Rijndael," in J. -J. Quisquater and B. Schneier (eds.), *Smart Card Research and Applications* (Springer-Verlag *Lecture Notes in Computer Science* 1820). New York, N. Y.: Springer-Verlag (2000).

[17.6] James Daly: "Fingerprinting a Computer Security Code," *Computerworld* (July 27, 1992).

[17.7] Dorothy E. Denning: *Cryptography and Data Security*. Reading Mass.: Addison-Wesley (1983).

[17.8] Dorothy E. Denning and Peter J. Denning: "Data Security," *ACM Comp. Surv.* 11, No. 3 (September 1979).

本书是一本不错的入门书,包括自主存取控制、强制存取控制(此处称为流控)、数据加密以及推理控制(统计数据库的专有问题)。

[17.9] W. Diffie and M. E. Hellmans: "New Directions in Cryptography," *IEEE Transactions on Information Theory* IT - 22 (November 1976).

[17.10] Ronald Fagin: "On an Authorization Mechanism," *ACM TODS* 3. No. 3 (September 1978).

该文章是对参考文献[17.13]的勘误。由于参考文献[17.13]的机制在一定的情况下将回收不应该回收的权限,这篇文章正是对此进行了纠正。

[17.11] Roberto Gagliardi, George Lapis, and Bruce Lindsay: "A Flexible and Efficient Database Authorization Facility," IBM Research Report RJ6826 (May 11, 1989).

[17.12] Martin Gardner: "A New Kind of Cipher That Would Take Millions of Years to Break," *Scientific American 237*, No. 2 (August 1977).

对公用密钥加密方法的非正式的介绍,参考文献[17.14, 17.24]将有进一步的说明。

[17.13] Patricia P. Griffiths and Bradford W. Wade: "An Authorization Mechanism for a Relational Data Base System," *ACM TODS 1*, No. 3 (September 1976).

描述了 System R 中最先采用的 GRANT 和 REVOKE 机制,该机制是 SQL 标准所采用的模式的基础,虽然在实现细节上并不相同。

[17.14] Nigel Hawkes: "Breaking into the Internet," *London Times* (March 18, 1996).

描述了一个计算机专家破解 RSA 模式的方法——通过估计系统对消息解密花费的时间,这种方法其实是"通过观察一个人拨号以及每次拨号花费的时间的方法猜测锁的组合方式的电子等价物"。

[17.15] Sushil Jajodia and Ravi Sandhu: "Toward a Multi-Level Secure Relational Data Model," Proc. 1991 ACM SIGMOD Int. Conf. on Management of Data, Denver, Colo. (June 1991).

正如17.3节中所指出的,安全性环境下的"多级"说明了系统支持强制存取控制。这篇文章暗示了此领域当前的工作很特殊,因为在基本的概念上尚未达成一致,文章提出了多级系统原理的形式化说明。

[17.16] Abraham Lempel: "Cryptology in Transition," *ACM Comp. Surv. 11*, No. 4: Special Issue on Cryptology (December 1979).

本书是关于加密及相关内容的入门书。

[17.17] R. L. Rivest, A. Shamir, and L. Adleman: "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *CACM 21*, No. 2 (February 1978).

[17.18] Ken Smith and Marianne Winslett: "Entity Modeling in the MLS Relational Model," Proc. 18th Int.

Conf. on Very Large Data Bases, Vancouver, Canada (August 1992).

论文中的"MLS"表示"多级安全"[17.15]。这篇论文着重讨论了 MLS 数据库的意义，并提出了新的 BELIEVED BY 子句用于查询和更新操作，使这些操作处于数据库理解或特定用户"相信"的状态。该方法可解决以前的方法能解决的许多问题。见参考文献[17.23]。

[17.19] Bhavani Thuraisingham: "Current Status of R& D in Trusted Database Management Systems," *ACM SIGMOD Record 21*, No. 3 (September 1992).

"可信"或多级系统（20 世纪 90 年代早期）的概述以及大量的参考文献的集合。

[17.20] U. S. Department of Commerce/National Bureau of Standards: *Data Encryption Standard.* Federal Information Processing Standards Publication 46 (January 15 1977).

定义了官方的数据加密标准（DES）。加密/解密算法（参看 17.5 节）适合在硬件芯片上实现，采用该算法的设备能以很高的数据速率操作。已有大量这样的设备可购得。

[17.21] U. S. Department of Defense: *Trusted Computer System Evaluation Criteria* (the "Orange Book"), Document No. DoD 5200-28-STD. DoD National Computer Security Center (December 1985).

[17.22] U. S. National Computer Security Center: *Trusted Database Management System Interpretation of Trusted Computer System Evaluation Criteria* (the "Lavender Book"), Document No. NCSC-TG-201, Version 1 (April 1991).

[17.23] Marianne Winslett, Kenneth Smith, and Xiaolei Qian: "Formal Query Languages for Secure Relational Databases," *ACM TODS 19*, No. 4 (December 1994).

继续参考文献[17.18]的工作。

[17.24] Ron Wolf: "How Safe Is Computer Data? A Lot of Factors Govern the Answer," *San Jose Mercury News* (July 5, 1990).

# 第18章 优 化

## 18.1 引言

对于关系数据库系统来说，优化既是挑战也是机遇。称之为挑战，是因为为了达到所需的性能要求，数据库系统必须使用优化技术；而说它是机遇，是因为关系表达式具有的高度语义层次使得优化能够进行。而在非关系系统中，用户的查询使用低层次的语义表达，任何的“优化”都由用户来进行（“优化”之所以使用引号引起来，是因为一般谈到的优化指的是由系统自动进行的），换句话说，在这样的系统中，是由用户而不是由机器来决定使用什么样的底层操作及操作的顺序。而且，如果用户作出了错误的决定，那么系统对此也是无能为力的。同时还应当注意到，这就意味着这种系统的用户必须是编程高手。仅此一点，就使得许多普通用户无法从该数据库系统中受益。

而自动优化系统的优势就在于用户不必担心如何最好地表达他们的查询要求（也就是如何描述查询以获得最优的系统性能）。事实上，优化器完全有可能比用户做得更好，这是有许多原因的，比如：

1）一个好的优化器具有一些信息而用户通常不具备。比如说，它有一些统计信息，例如：

- 每个类型中的值的个数
- 每个基表中的当前元组的数目
- 每个基表中的每个属性的唯一值的个数
- 这些唯一值在每个属性中出现的次数

等等（所有这些信息都保存在系统目录表中——见18.5节）。因此，优化器可以对给定查询的执行策略作出更为精确的估计，从而更有可能选择最高效的执行方案。

2）而且，当数据库统计信息改变时，可能需要改变执行策略；换句话说，这时需要再优化。在关系系统中，再优化很简单——只需要让系统的优化器重新处理一遍原来的查询请求。而在非关系系统中，再优化就意味着重写源程序，而且很有可能没人愿意做这件事情。

3）再者，优化器是个程序，因此它要比人类用户耐心得多。优化器可以考虑一个给定查询的成百上千的执行策略，而人类用户恐怕很难考虑多过三、四个的执行策略。

4）最后，优化器可以被认为是包含着最优秀程序员的技巧和服务的程序。因此，任何人都可使用优化器包含的这种技巧和服务——这就意味着更多的用户可以以一种高效率、低代价的方式得到这些资源。

在本章开头提到的优化能力（也就是说，关系查询是可优化的）是关系系统提供的一种功能，而以上这些可作为对这一观点的证据支持。

那么，优化器的主要目的就是为计算所给定的关系表达式选择一个高效的执行策略。在本章中，我们将描述一些在这个过程中所使用的基本原则和技术。在下面，18.2节介绍了一个具有启发性的例子，18.3节给出了优化器如何工作的概述，18.4节详细描述了这个过程中一个非常重要的方面，即表达式变换（expression transformation），也称为查询改写（query rewrite）。18.5节简要讨论了数据库统计（database statistics）的问题。18.6节详细描述了查询分解（query decomposition）的方法。18.7节阐述了关系操作符如何实现的问题，并且简要讨论如何使用18.5节中所提出的统计方法来进行一个代价估算。最后，18.8节对整章作了总结。

最后提出一个引导性的评价：我们通常把这个主题称为查询优化。不过这个术语容易让人误解，因为无论如何，除了数据库的交互式查询外，需要优化的表达式（即查询）还可能产生于某些上下文中；特别地，从本质上看，它可能是某个更新操作的一部分而不仅仅是一个查询。而且，术语“优化”本身就有点夸大了，因为并不能保证所选择的实现策略在任何方面都是最优的。事实上，通常所谓的“优化”策略只是在原来没有优化的查询上的一个改进。（但是在某些

特殊场合中，就可以认为其选择的执行策略确实是最优的。例如可以参看参考文献［18.30］，还可参看附录A。）

## 18.2　一个启发性的示例

我们以一个简单的示例开始（这个例子已在第7章的7.6节中详细描述过），它给我们提供了显著改进查询执行策略的思路。这个查询是“给出供应零件P2的供应商的名称”。以下是它的代数表达式：

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

假设这个数据库有100个供应商和10 000个发货，其中只有50个是有关零件P2的。为简单起见，还假设关系变量S和SP分别存放在磁盘上两个独立的文件中，每个元组对应一个记录。这样，如果系统不对该表达式进行任何优化，那么对该表达式的代价估计结果如下：

1）*连接SP和S（连接属性S#）*：这一步包括读10 000个发货；对这100个供应商，每个要读10 000次（即对10 000个发货的每一个要读一次供应商）；要构造一个中间结果集保存这10 000个已连接的元组；并且将这10 000个已连接的元组写回到磁盘上（我们假设主存中已没有空间存放这个中间结果集）。

2）*从第1）步的结果集中选择零件为P2的元组*：这一步包括将这10 000个已连接的元组重新读入主存，但产生一个仅有50个元组的结果集，这里我们假设这个结果集可以放在主存中。

3）*将第2）步的结果集在属性SNAME上投影*：这一步产生最终的结果集（至多50个元组，可以放在内存中）。

下面这个过程等价于上面这个过程，即产生同样的结果集，但明显要高效得多：

1）*从SP中选择零件为P2的元组*：这一步包括读10 000个元组，但产生只有50个元组的结果集，我们假设这个结果集可以放入内存中。

2）*把第1）步的结果集和S进行连接（连接属性S#）*：这一步包括读100个供应商（只读一次，而不是每个P2发货就读一遍），再次产生一个只有50个元组的结果集（仍然保存在主存中）。

3）*将第2）步的结果集在属性SNAME上投影（同上个过程的第3）步）*：这一步产生最终的结果集（至多50个元组，可以放在内存中）。

以上两个过程中的第一个包括总共1 030 000个元组的I/O，而第二个只有10 100个元组的I/O。很明显，如果我们将“元组的I/O次数”作为性能计算标准，那么第二个过程就要比第一个快上100倍。同样清楚的是，我们希望使用第二种实现方案而非第一种。*注意*：实际当中，应当以页面I/O次数作为标准而不是元组I/O次数，所以为了简化我们不妨假设每个元组被单独存储在一个页面上。

可以看到，在执行代数表达式中的一个很简单的变化，即先做选择后再做连接，而不是先做连接后做选择，就使得执行的性能有了戏剧性的改变。而且，如果发货表在P#属性上有索引或者散列，那么第1）步读入的基数（cardinality）将从10 000降到50，这将使执行性能比原来有近7000倍的提高。同样，如果供应商表在S#属性上建有索引或者散列，那么在第2）步中读入的供应商的基数将从100降到50，这将使执行性能比原来提高10 000倍。这就意味着，如果原来的没有优化的查询需要运行3个小时，那么优化过的版本只需要不到1秒的时间，而且更大程度的提高都有可能。

前面的这个例子虽然简单，但足以说明为什么查询优化是必要的。这个例子也可以说明在实际中什么样的优化是可能的。在下一节中，我们将对优化问题提出一个系统解决方案的框架；尤其是，我们将解释如何把优化问题分解为一系列独立的子问题。这为下面进一步讨论和理解优化策略和技术提供了便利的基础。

## 18.3　查询处理概述

我们可以将查询处理分为四个大的阶段（参看图18-1）：

1）将查询转换为内部格式。
2）将内部格式转换为规范格式。
3）为执行选择低层调用。
4）生成并选择最低代价的查询计划。

以下我们将详细说明这四个阶段。

### 1. 阶段 1：将查询转换为内部格式

第一阶段包括将原查询转换为某种内部格式以便于机器处理。这样做可以消除外部考虑（比如考虑查询语言严格语法中的遁词现象），并且给接下来的优化过程铺平道路。注意：视图处理，即用定义视图的表达式来替代对视图的引用，也是在这个阶段进行的。

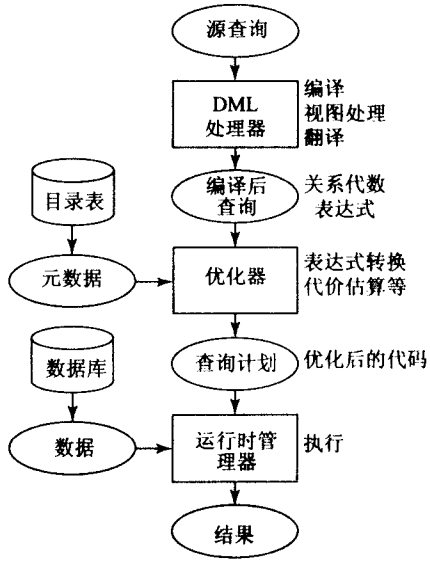



图 18-1　查询处理流程

有一个明显的问题是：这种内部格式应当遵循什么样的形式呢？不管遵循什么样的形式，凡是外部的查询语言能够表达的查询，都应当能用它来表示。这种形式还应当是尽量中立的，这样它就不至于倾向选择某些查询计划。这种内部格式通常为某种**抽象语法树**（abstract syntax tree）或者是**查询树**（query tree）。例如，图18-2画的就是对18.2节中例子（“给出供应零件 P2 的供应商的名称”）的一种可能的查询树表示方法。

为了说明方便，假设这种内部格式是我们已经熟知的关系代数或者是关系演算的形式。比方说，像图18-2这样的树就可以用关系代数或者是关系演算的形式来等价地表示出来。在这里，我们规定只遵循关系代数的表示方法。因此，可以将图 18-2 用内部格式表达如下（这个关系代数表达式在前面已经出现过）[○]：

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

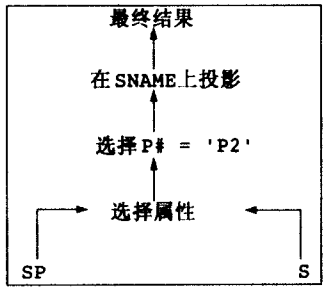



图 18-2　“给出供应零件 P2 的供应商的名称”（查询树）

### 2. 阶段 2：将内部格式转换为规范格式

在这一阶段中，优化器执行一系列“保证能够优化”的优化过程，而不必考虑实际数据的值以及数据库的存取路径。可以这么做的关键在于，至少从表面上来看，即便是最简单的查询，关系语言也能够让用户将它以多种方式表达出来。例如，在 SQL 中，即便不考虑类似“$A = B$ 替换为 $B = A$ 或者是 $p$ AND $q$ 替换为 $q$ AND $p$”之类的变换，像“给出供应零件 P2 的供应商的名称”这样简单的查询仍然可以用几十种方法来表示[⊖]。而且，查询的性能也不应当因为书写的方式不同而有所差异。因此，查询处理的下一步就是将这些内部表示转换为等价的规范格式（见下一段）。这样做的目的是为了消除某些表面上的差异，更重要的是，找到一种在某些方面比原查询更为高效的表示方法。

关于“规范格式”的注释：规范格式的概念在许多数学的分支以及相关学科中都是十分重要的。它可以按如下方式定义：给定（比方说查询）对象集 $Q$，并定义它们之间等价的规则（比方说规则可以定义为：查询 $q1$ 和 $q2$ 是等价的，当且仅当 $q1$ 和 $q2$ 产生相同的结果

---

○ 事实上，把原始查询映射为等价的关系代数正是一些商业 SQL 优化器所采用的方法。

⊖ 然而我们注意到，SQL 语言在这个问题上表现得更为突出（参见第 8 章练习 8.12 和参考文献 [4.19]）。其他语言如关系代数或关系演算，并没有对同一个请求用很多种不同的表示方法。SQL 的这种不必要的“灵活性”实际上造成了实现上的困难（更不用说给用户使用了），因为它使得优化器的工作更为复杂。

集），称 $Q$ 的子集 $C$ 是在等价规则下的规范格式的集合，当且仅当 $Q$ 中每个对象 $q$ 都等价于 $C$ 中的某一对象 $c$。对象 $c$ 被称为对象 $q$ 的规范格式。所有适用于 $q$ 的“有意义的”的属性对 $c$ 也是适用的；因此为了检验多种“有意义的”结果集，只研究小集合 $C$ 中的对象就足够了，而不必是大集合 $Q$。

现在回到我们讨论的主线上来：为了将阶段 1 的输出转换为某种等价的、但更为高效的格式，优化器会使用一些**变换规则**（transformation rules or laws）。下面是这种规则的一个例子：表达式

```
( A JOIN B ) WHERE restriction on A
```

可以被转换为下面这个等价、但更为高效的表达式：

```
( A WHERE restriction on A ) JOIN B
```

我们已经在第 7 章的 7.6 节中简要讨论过这种转换；同时它也是 18.2 节中的例子，那个例子清楚地表明为什么需要做这种转换。更多的变换规则将在 18.4 节中讨论。

**3. 阶段 3：为执行选择低层调用**

将查询的内部格式转换为某种更好的形式后，优化器必须决定如何执行这个查询。在这个阶段中，将要考虑诸如是否存在索引或者其他物理存取路径、数据值的分布情况、数据的物理聚集存储等等问题。请注意，我们在阶段 1 和阶段 2 中并没有考虑过这些问题。

基本的策略是将查询表达式看作是由一系列“低层操作”（low-level operation）构成的[⊖]，这些操作间保持一定的相互依赖关系。这种依赖性的例子如下：投影操作通常需要它的输入元组是按某种方式排序的，以便于它能够去除重复值。这样就要求投影前面的那个操作的输出元组必须是按这种方式排序的。

对于每个可能的低层操作（并且可能也是这些操作的各种平常组合），优化器都具有一组可用的低层调用与之对应。例如，可以有一组实现选择操作的过程：有针对实现在候选码上做等式判断的选择操作实现过程，有针对已索引的属性的选择操作实现过程，有针对已散列化的属性的选择操作实现过程，等等。在 18.7 节中给出了这样的例子（同时请见参考文献［18.7，18.12］）。

每个低层调用都有一个相关的代价公式（cost formula），表明其执行代价，尤其是磁盘 I/O 代价。有些系统也考虑 CPU[⊖] 利用率以及其他一些因素。这些代价公式会在阶段 4 中用到。参考文献［18.7－18.12］讨论并分析了在不同条件下的许多操作实现过程的代价公式。同时请参考 18.7 节。

因此，使用字典表中有关数据库当前状态的相关信息（如是否存在索引，当前的基数等）以及使用上面提到的相互依赖性的信息，优化器将选择一个或多个用以实现查询表达式中低层操作的候选过程。有时称这个过程为**存取路径选择**（access path selection）（见参考文献［18.33］）。注意：参考文献［18.33］使用术语“存取路径选择”用以涵盖阶段 3 以及阶段 4 的工作，而不仅仅只有阶段 3。实际上，阶段 3 和阶段 4 也很难有一个清晰的划分界限，阶段 3 的工作或多或少地在阶段 4 中出现。

**4. 阶段 4：生成并选择最低代价的查询计划**

优化的最后阶段包括构造一组**查询计划**（query plan），然后从中选择一个最优的，也就是代价最低的查询计划。每个查询计划通过绑定一系列的实现过程而构成，每一个过程对应查询中的一个低层操作。注意，对于任何给定的查询，通常会有非常多的可能的查询计划。所以，产生所有的查询计划并不明智，因为组合会使这些查询计划过多，造成选择这些查询计划本身代价太大；如果不会将有利的查询计划排除在外的话（参看文献参考［18.53］），那么一些用以将产生

---

⊖ “层”显然是一个相对的概念。这里提到的“低层”操作指的是关系代数操作（如连接、选择、求和等），然而这些操作更多时候被看作是“高层”操作。

⊖ CPU 代表中央处理器。

的查询计划数目保持在一个限度内的启发式技术是非常可取的。“将查询计划数目保持在一个限度内”通常与减少搜索空间的问题相关，因为这种做法被认为是减少了优化器对于可管理范围内的查询的搜索范围。

要选择代价最低的查询计划自然需要一种方法来计算任何一个计划的代价。一般地讲，一个给定的查询计划的代价就是构成这个计划的各个过程的代价之和。所以，优化器必须根据代价公式计算每个过程的代价。问题在于，这些代价公式的值依赖于所处理关系的大小，而即便是最简单的查询在执行中也会产生中间结果集，所以优化器就不得不估计中间结果集的大小。然而，这些中间结果集的大小非常依赖于实际的数据值。所以，准确的代价估计是一个困难的问题。参考文献［18.2，18.3］讨论了解决这个问题的一些方法，并给出了这个领域中的一些研究文献。

## 18.4 表达式转换

在本节中，我们将描述一些可能会对优化处理的第 2 阶段有用的**变换规则**。本章的练习会要求为部分规则写一些例子以及准确地说明它们可能会有用的原因。

我们使用某个变换规则能够将一个给定的表达式转换为另一种形式，这种形式还可以被另一个规则再转换。例如，一个实际中的原查询不大可能写成具有两个连续投影的形式（见下面“选择和投影”小节中的第二个规则），而转换过后的查询却有可能是这种形式。（一个重要的例子就是**视图处理**（view processing）。例如，查询“给出视图 V 中的所有城市 CITY”，其中视图 V 定义为供应商在属性 S#、CITY 上的投影）。换句话说，由原查询开始，优化器对查询形式进行转换，直至它认为所得到的形式依据某些启发式规则已经是最优的为止。

### 1. 选择和投影

下面是选择和投影的一些变换规则：

1）对同一关系连续的多个选择可以转换为一个用 AND 连接的选择操作。例如，表达式

```
( A WHERE p1 ) WHERE p2
```

等价于表达式

```
A WHERE p1 AND p2
```

这个变换是有意义的，因为原始的表达式意味着要对 *A* 进行两次扫描，而变换后的版本只需要一次就够了。

2）对同一关系连续的多个投影可以转换为仅含最后一个投影的操作。例如，表达式：

```
( A { acl1 } ) { acl2 }
```

（其中 *acl*1 和 *acl*2 是属性名）等价于表达式

```
A { acl2 }
```

当然，要使原查询有意义，*acl*2 必须是 *acl*1 的子集。

3）投影操作后的选择操作可以转换为选择操作后的投影操作。例如，表达式

```
( A { acl } ) WHERE p
```

等价于表达式

```
( A WHERE p ) { acl }
```

*注意*，通常情况下在投影操作前做选择操作更好，因为选择操作可以减少投影操作的输入大小，而投影操作可能为了去除重复项而需要输入数据有序，因而也就减少了需要进行排序操作的数据量。

### 2. 分配律

在 18.2 节的例子中使用到的变换规则（将先连接后选择转换为先选择后连接）实际上是**分配律**的一个特例。一般来说，称一元操作符 *f* 在二元操作符○上是可分配的（distribute），当且

仅当对所有的 $A$ 和 $B$，有

```
f ( A ○ B ) ≡ f ( A ) ○ f ( B )
```

在一般的算术表达式中，例如 SQRT（平方根运算）在乘法上就满足分配律，这是因为对所有的 $A$ 和 $B$，有

```
SQRT ( A * B ) ≡ SQRT ( A ) * SQRT ( B )
```

因此在做算术表达式转换时，优化器总可以使用这些形式来互换。另外有个反例，比如 SQRT 在加法上就不满足分配律，因为 SQRT（$A+B$）一般不等于 SQRT（$A$）+SQRT（$B$）。

在关系代数中，选择操作符在并、交、差操作上均满足分配律。在连接操作上，选择操作满足分配律的条件最为复杂的情况是，当且仅当选择条件是两个独立的选择条件[⊖]的合取式，其中每个选择条件对应连接操作的一个操作数。在 18.2 节的例子中的情况满足了这些要求。实际上，这个条件是很简单的，而且只实施于其中一个操作数。这样就可以使用分配律将表达式转换为一个更高效的等价形式。这样做可以使得“选择操作尽量早做”。早做选择操作总是有利的，因为这样可以减少下一步需要扫描的元组数，而且可以减少下一步的输出。

下面是几个分配律的例子，涉及投影操作。第一点，投影操作在并操作和交操作上满足分配律（在差操作上不满足）：

```
( A UNION B ) { acl } ≡ A { acl } UNION B { acl }
( A INTERSECT B ) { acl } ≡ A { acl } INTERSECT B { acl }
```

这里 $A$，$B$ 是同一类型的关系。

第二点，投影操作在连接操作上也满足分配律，只要投影操作保持了所有的连接属性：

```
( A JOIN B ) { acl } ≡ ( A { acl1 } ) JOIN ( B { acl2 } )
```

这里 *acl*1 是 *acl* 中仅在 $A$ 中出现的属性的并集，*acl*2 是 *acl* 中仅在 $B$ 中出现的属性的并集。

这些规则可使得“投影操作尽量早做”。类似于选择操作的原因，这同样总是有利的。

**3. 交换律和结合律**

交换律和结合律是两个重要的规则。首先，称二元操作符○是**可交换的**（commutative），当且仅当对所有的 $A$ 和 $B$，有

```
A ○ B ≡ B ○ A
```

在算术中，乘法和加法都是满足交换律的，但除法和减法不是。在关系代数中，并操作、交操作以及连接操作都是可交换的，但差操作和关系除法不是。所以，如果一个查询包括两个关系 $A$ 和 $B$ 的连接，那么交换律保证无论 $A$ 或 $B$ 谁作“外”关系谁作“内”关系在逻辑上都没有区别。因此，系统就可以自由地选择其中的小关系作为“外”关系来计算连接（见 18.7 节）。

现在讨论结合律。称二元操作符○是**可结合的**（associative），当且仅当对所有的 $A$、$B$、$C$，有

```
A ○ ( B ○ C ) ≡ ( A ○ B ) ○ C
```

在算术中，乘法和加法都是满足结合律的，但除法和减法不是。在关系代数中，并操作、交操作以及连接操作都是可结合的，但差操作和关系除法不是。所以，如果一个查询包括三个关系 $A$，$B$ 和 $C$ 的连接，那么交换律和结合律一起保证了无论按什么样的顺序进行连接都在逻辑上没有区别。因此，系统就可以自由地选择代价最小的连接顺序。

**4. 幂等律和吸收律**

幂等律是另一个重要的规则。称二元操作符○是**幂等的**（idempotent），当且仅当对所有的

---

⊖ 对于选择条件的解释可以参考第 7 章的 7.4 小节。

$A$，有

```
A ○ A ≡ A
```

幂等属性在表达式变换中也有用处。在关系代数中，并操作、交操作以及连接操作都是幂等的，但差操作和关系除法不是。

并操作和交操作也都满足以下的**吸收律**（absorption）：

```
A UNION ( A INTERSECT B ) ≡ A
A INTERSECT ( A UNION B ) ≡ A
```

**5. 计算的表达式**

这些变换规则不仅适用于关系表达式。例如，前面已经说明一些变换规则同样适用于算术表达式。下面是个例子：表达式

```
A * B + A * C
```

可以被转换为

```
A * ( B + C )
```

这是因为“*”在“+”上满足分配律。关系优化器同样需要知道这些，因为**扩展**（extend）操作符和合计（summarize）操作符可能会用到这样的表达式。

注意，这个例子给出了比分配律更为普遍的形式。前文是根据一元操作符在二元操作符上的分配定义的分配律；而在这个例子中，“*”和“+”都是二元操作符。总之，称二元操作符 $\delta$ 在二元操作符○上是**可分配的**，当且仅当对所有的 $A$，$B$ 和 $C$，有

```
A δ ( B ○ C ) ≡ ( A δ B ) ○ ( A δ C )
```

在上面的算术例子中，用“*”代替δ，用“+”代替○。

**6. 布尔表达式**

现在开始讨论布尔表达式。假设 $A$ 和 $B$ 是两个不同关系的属性。那么布尔表达式

```
A > B AND B > 3
```

明显等价于（即可被转换为）：

```
A > B AND B > 3 AND A > 3
```

这是因为比较操作符“>”是**可传递的**（transitive）。注意这个转换是值得的，因为这样系统可以在关系 $A$ 上多做一个选择操作，从而减少参与比较操作“$A > B$”的基数。这里再重复前面提到过的一个观点：选择操作尽量早做；像这里的例子一样，执行系统的隐含选择条件也是一个好的办法。注意：在一些商用系统中已经实现了该技术，例如 DB2（这项技术在其中被称为“谓词传递闭包”（predicate transitive closure））和 Ingres。

下面是另一个例子。表达式

```
A > B OR ( C = D AND E < F )
```

可以转换为

```
( A > B OR C = D ) AND ( A > B OR E < F )
```

这是因为 OR 在 AND 上满足分配律。这个例子说明了另外一个规则：任何一个布尔表达式都能被转换为一个等价的**合取范式**（conjunctive normal form，CNF）。合取范式是具有如下形式的表达式

```
C1 AND C2 AND ... AND Cn
```

其中，$C1$，$C2$，…，$Cn$ 都是不包含 AND 的布尔表达式，称之为**合取项**（conjunct）。CNF 表达

式的优点是：只有当所有的合取项都真时，该表达式才为真；只要有一个为假，该表达式为假。因为 AND 操作符是可交换的（*A* AND *B* 等价于 *B* AND *A*），所以优化器可以按任何顺序计算合取范式的合取项；特别地，优化器可以按照增量难度次序来计算（最简单的最先做）。只要优化器发现一个合取项为假，那么处理过程就停止。而且在并行系统中，有可能将所有的合取项进行并行计算［18.56－18.58］。同样，一旦发现一个合取项为假，处理过程随即停止。

可以看到，优化器需要知道这些规则，如分配律，不仅适用于关系操作符（如连接），同样也适用于比较操作符（如“>”）、布尔操作符（如 AND 和 OR）以及算术操作符（如“+”）等等。

**7. 语义转换**

考虑如下表达式

```
( SP JOIN S ) { P# }
```

这里的连接是个**外码匹配候选码连接**（foreign-to-matching-candidate-key join）；它用 SP 的一个外码来匹配 S 中相应的候选码。这意味着每个 SP 元组都需要同一些 S 元组做连接，同时每个 SP 元组的属性 P#都会出现在结果集中。换句话说，这根本就不需要做连接！这个表达式可以简化为

```
SP { P# }
```

特别要注意的是，这个转换之所以成立只是因为语义的原因。一般地讲，做连接的两个关系彼此都会有一些对方没有的连接属性的值（因此不是所有的属性值都会出现在连接结果集中），所以这样的转换也是不成立的。但这个例子中，因为完整性约束的缘故（实际上是参照完整性），即所有的货物都必须有供应商，所以所有的 SP 的元组都在 S 中有对应的值。所以这个转换也就是可行的。

只是因为完整性约束的原因才使得一个转换成立的情况被称为**语义转换**（semantic transformation）［18.25］，而由此形成的优化称为**语义优化**（semantic optimization）。语义转换可以定义为依靠完整性约束，将一个查询转换为另一个查询，它们在查询效率上有差别，但输出结果集一致。

需要明确的是，原则上任何的完整性约束条件都可以用于语义优化（并不只限于参照完整性约束）。例如，假如供应商和零件数据库都受限于条件“所有的红色零件都必须存放在 London”，那么考虑查询：

> 获取这样的供应商，他们仅供应红色零件，并且他们所在的城市至少有他们提供的一种零件。

这是个相当复杂的查询，但根据完整性约束，它可以转换为如下更简单的形式：

> 获取 London 供应商，他们仅供应红色零件。

注意：就笔者所知，目前很少有商用系统在语义优化上下了工夫。但是原则上讲，这种转换可以很大地提高性能，这种提高可能比现在任何一种传统的优化技术要大得多。有关语义优化的更深入讨论可以参看参考文献［18.13，18.25－18.28］，其中特别推荐［18.25］。

**8. 本节小结**

在本节的最后，需要强调关系封闭（closure）对讨论内容基础性的重要作用。封闭意味着可以写嵌套表达式，这就是说，可以用单独一个表达式而不是一个多语句的过程调用来表达一个查询；因此，不需要任何的流程分析。同样，这些嵌套表达式是用子表达式递归定义的，这就允许优化器采取许多“分而治之”的策略（见 18.6 节）。而且，如果没有封闭性，各种的变换规则，如分配律等，就没有任何意义。

## 18.5　数据库统计信息

整个优化过程的阶段 3 和阶段 4，即“存取路径选择”阶段，利用了存储在目录表中的数据库统计信息（database statistic）（详细见 18.7 节对这些统计信息使用的描述）。本节将综述（并

稍作评论）两个商用系统——DB2 和 Ingres——维护的主要的统计信息。下面是 DB2 维护的一些统计信息[⊖]：

- 对每个基表（base table）：
  - 基数的数
  - 该表占用的页面数
  - 该表占用的表空间的分片情况
- 对基表的每个列（column）：
  - 该列不同值的个数
  - 该列第 2 大的值
  - 该列第 2 小的值
  - 如果是索引列，该列 10 个最常用的值以及出现次数
  - 对每个索引（index）
    - 指明是否为“聚集索引”（即将所有逻辑上关联的数据物理地在磁盘上存放在一起的索引）
    - 如果为聚集索引，则索引表的分片也按照聚集顺序存放
    - 该索引叶页面数
    - 该索引的层数

注意：每次数据库更新时这些索引信息并不总是更新，因为这样就会带来额外的开销。它们的更新是通过 DBA 按需要（例如在数据库重组后）执行一个称之为 RUNSTATS 的系统工具软件实现的。大多数数据库的策略也是这样。在 Ingres 中，这个工具称之为 OPTIMIZEDB。

下面是 Ingres 的主要统计信息。注意：在 Ingres 中，索引被认为是特殊的存储表；因此，为基表和列收集的统计信息同样适用于索引。

- 对每个基表：
  - 基数的数目
  - 该表占用的主页面数
  - 该表占用的溢出页面数
- 对基表的每个列：
  - 该列不同值的个数
  - 该列最大值、最小值和平均值
  - 该列的实际值以及它们出现的次数

## 18.6 分而治之的策略

正如在 18.4 节提到的，关系表达式可以根据子表达式递归定义，这就使得优化器采取一种“分而治之”（divide and conquer）的策略。注意，这个策略在并行处理环境中可能尤其有利（特别是，在分布式系统中），因为查询的不同部分可以并行地在不同的处理器上执行［18.56-18.58］。在本节中，将研究一个称之为**查询分解**（query decomposition）的策略，这是在 Ingres 原型系统［18.34，18.35］中最早提出的。

查询分解的基本想法是：使用**拆分**（detachment）和**元组置换**（tuple substitution）的办法，将一个含有多个范围变量的查询[⊖]分解为一系列只含有 1 个或 2 个变量的小查询：

- **拆分**是将查询中的一部分（该部分只含有一个同剩余部分相同的变量）从查询中移出的过程。
- **元组置换**是指每次用一个元组置换其中一个查询变量的过程。

---

⊖ 因为 DB2 和 Ingres 都是支持 SQL 的系统，它们使用术语“表”和“列”分别代表关系变量和属性，所以本节中我们也使用这两个术语。并且，这两个系统都假设基本表是直接映射到存储表的。

⊖ Ingres 的查询语言 QUEL 是基于微积分学的。

只要有可能，拆分就在元组置换前使用。但是最终一个查询会被分解为一系列无法再拆分的部分，这样就要使用元组置换了。

这里给出一个例子（基于参考文献［18.34］的例子）。这个查询是“获取 London 供应商的名称，这些供应商供应某种红色零件，并且这些零件数量大于 200 而重量小于 25 磅”。下面是一个 QUEL 格式的查询表达式（“查询 Q0”）：

```
Q0: RETRIEVE ( S.SNAME ) WHERE S.CITY   = "London"
                         AND   S.S#     = SP.S#
                         AND   SP.QTY   > 200
                         AND   SP.P#    = P.P#
                         AND   P.COLOR  = "Red"
                         AND   P.WEIGHT < 25.0
```

其中，范围变量是 S、P 和 SP，它们在同名的基表范围内取值。

可以看到，最后两个子句是对零件的要求，即红色且重量小于 25 磅。所以可以拆分一个包含变量 P 的“单变量查询”（实际上是一个选择操作的一个投影）：

```
D1: RETRIEVE INTO P' ( P.P# ) WHERE P.COLOR  = "Red"
                              AND   P.WEIGHT < 25.0
```

之所以可以拆分出这个单变量查询，是因为它只有一个变量 P 相同于剩余部分。因为它通过属性 P#同原查询的剩余部分相关联（SP. P# = P. P#），所以在拆分后的部分中，P#是必须出现在“原型元组”（proto tuple）（见第 8 章）中的；也就是说，拆分后的查询必须获取红色的以及重量小于 25 磅的零件号。我们把这个拆分后的查询称为查询 D1 并把它的结果集放入一个临时关系 P′中（INTO 从句的作用是创建一个只有一个属性 P#的新关系 P′，用于自动存放执行 RETRIEVE 获得的结果集）。最后，在 Q0 的简化版本中把对 P 的引用换为对 P′的引用。下面把这个简化版本称为查询 Q1：

```
Q1: RETRIEVE ( S.SNAME ) WHERE S.CITY = "London"
                         AND   S.S#   = SP.S#
                         AND   SP.QTY > 200
                         AND   SP.P#  = P'.P#
```

现在对查询 Q1 执行一个相似的过程，拆分得到包含变量 SP 的单变量查询 D2，并且得到 Q1 的新简化版本 Q2。

```
D2: RETRIEVE INTO SP' ( SP.S#, SP.P# ) WHERE SP.QTY > 200

Q2: RETRIEVE ( S.SNAME ) WHERE S.CITY = "London"
                         AND   S.S#   = SP'.S#
                         AND   SP'.P# = P'.P#
```

下面拆分包含变量 S 的单变量查询：

```
D3: RETRIEVE INTO S' ( S.S#, S.SNAME ) WHERE S.CITY = "London"

Q3: RETRIEVE ( S'.SNAME ) WHERE S'.S#  = SP'.S#
                          AND   SP'.P# = P'.P#
```

最后拆分包含变量 SP′和 P′的双变量查询：

```
D4: RETRIEVE INTO SP'' ( SP'.S# ) WHERE SP'.P# = P'.P#

Q4: RETRIEVE ( S'.SNAME ) WHERE S'.S# = SP''.S#
```

因此，原查询 Q0 就被分解为三个单变量查询 D1，D2，D3（每一个都是选择操作的投影）和两个双变量查询 D4 和 Q4（每一个都是连接操作的投影）。可以用如图 18-3 所示的树结构来表示这种情况。这个图可以这样理解：

- 查询 D1，D2 和 D3 的输入分别是关系 P，SP 和 S（更准确地说，是关系 P，SP 和 S 的当前值）。它们的输出分别是 P′，SP′和 S′。
- 查询 D4 的输入是 P′和 SP′，输出是 SP″。

■ 最后，查询 Q4 的输入是 S′和 SP″，输出是整个查询的结果。

可以发现，查询 D1，D2 和 D3 是完全独立的，并且可以按任何次序处理（甚至在并行环境中也是这样）。同样，一旦 D1 与 D2 处理完毕，查询 D3 和 D4 可以按任何顺序处理。但是，查询 D4 和 Q4 就无法再分解而且必须用元组置换来处理——即**强制方法**（brute force）、**索引查找**（index lookup）或者**散列查找**（hash lookup），见 18.7 节。例如，考虑查询 Q4。按一般的例子数据，SP′. S#供应商编号集合是｛S1，S2，S4｝。这三个值的每一个将轮流置换 SP″. S#。因此，可以将 Q4 的计算过程看作如下方式的表达：

```
RETRIEVE ( S'.SNAME ) WHERE S'.S# = "S1"
                      OR    S'.S# = "S2"
                      OR    S'.S# = "S4"
```

参考文献［18.34］给出了将原查询分解为最简化的小查询以及为每个变量做元组置换的算法。做元组置换的选择时会有许多优化工作可以做。参考文献［18.34］包括了一些做这种选择的代价估算的启发式规则（Ingres 通常会选择具有最小基数的关系做元组置换）。总地来说，优化过程的主要目标是避免做笛卡尔积和保持每个阶段做扫描的关系基数最少。

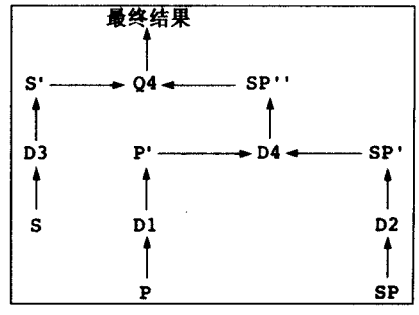



图 18-3 查询 Q0 的分解树

参考文献［18.34］并没有讨论单变量查询的优化问题。但是，有关于此的优化在 Ingres 的概述文献［8.11］中有所论及。它基本上类似于其他系统的相似功能，包括使用目录表统计信息以及在扫描数据时选择存取路径等（例如散列方法或是索引方法）。

参考文献［18.35］给出了一些实验数据。这些数据来源于一些基准测试的查询，证明以上论及的技术基本可行，而且在实际应用中很有效。下面是该文的一些结论：

1）拆分是最好的第一步。

2）如果必须先做元组置换，那么被置换的变量的最佳选择是个连接变量。

3）一旦对一个双变量查询中的一个变量使用元组置换，那么如果有必要，最好在另一个关系的连接属性上动态建立一个索引或散列。（实际上，Ingres 经常使用的就是这个策略。）

## 18.7 关系操作的实现算法

我们现在提供一些关系操作实现方法的简短描述，尤其是连接操作的算法。我们这样做的目的就是尽量扫除围绕着优化技术的一些迷雾，使人觉得它不再那么神秘。下面讨论的方法就是我们在 18.3 节中称为“低层实现过程”的操作。注意：一些更为复杂的实现技术将在本章最后的参考文献部分讨论，也可参考附录 A。

为了简单起见，我们假设数据就是以关系和元组的形式物理存储的。我们考虑的关系操作符包括投影、连接和合计。其中，合计操作包括以下两种情况：

1）求和（列属性除外）。

2）对至少一个属性求和。

第一种情况很明显：一般地讲，它需要扫描整个关系做求和运算，除非这个被施以聚集操作的列恰好被索引过，这样它也许能够不必存取关系本身而直接从索引计算出结果。例如，表达式

```
SUMMARIZE SP ADD SUM ( QTY ) AS TQ
```

就能够通过扫描 QTY 索引（假设存在该索引）计算出结果。如果用 COUNT 或 AVG 代替 SUM（对于 COUNT 来说，任何索引都可以），上述结论也同样成立。至于 MAX 和 MIN，只存取最后一个索引块（对于 MAX 来说）或者第一个索引块（对于 MIN 来说）就可以得到结果（这里同样假设相关属性的索引存在）。

在本节的剩余部分，“合计”特指第二种情况。下面是第二种情况的一个例子：

```
SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY
```

在使用者看来，投影、连接和第二种情况的合计相互之间差别都很大。从实现的角度来看，它们有一定的共性，因为不管是哪种情况，系统都需要在指定的列上按相同的列值做分组操作。如果是投影，这样的分组操作允许系统删除重复的元组；如果是连接，它就会找匹配的元组；如果是合计，它就分别计算每个分组的聚集值。下面列出了这样的几种分组技术：

1）强制方法（brute force）
2）索引查找方法（index lookup）
3）散列查找方法（hash lookup）
4）归并方法（merge）
5）散列方法（hash）
6）综合使用以上方法。

图 18-4 ~ 图 18-8 给出了连接算法的伪代码（投影和合计操作留作练习）。有几点需要注意：首先，$R$ 和 $S$ 是被连接的关系，$C$ 是它们的（有可能为组合的）相同列属性。假设可以按某种顺序一次一条记录地存取 $R$ 和 $S$，分别记作 $R[1]$，$R[2]$，…，$R[m]$ 和 $S[1]$，$S[2]$，…，$S[n]$。使用表达式 $R[i] * S[j]$ 代表元组 $R[i]$ 和 $S[j]$ 的连接结果。最后，设 $R$ 为外（outer）关系，$S$ 为内（inner）关系（因为它们各自负责外层和内层循环）。

**1. 强制连接方法**

强制连接方法可以被称为最“直接”的方式。在这个算法中，检查所有可能的元组组合（如图 18-4 所示，$R$ 的每个元组都试图同 $S$ 的每个元组进行连接）。注意：强制连接方法通常被称作是“嵌套循环”（nested loop），但这样实际上有点误导，因为不管是哪种算法都会有嵌套循环的。

```
do i := 1 to m ;                                    /* 外层循环 */
   do j := 1 to n ;                                 /* 内层循环 */
      if R[i].C = S[j].C then
      add joined tuple R[i] * S[j] to result ;
   end ;
end ;
```

图 18-4　强制连接方法

下面考察一下强制连接方法的代价问题。注意：尽管在实际中其他的代价（如 CPU 时间等）也很重要，但这里只考虑 I/O 代价。

首先，强制连接方法需要 $m + (m * n)$ 个元组读操作；但有多少写操作呢？也就是连接结果集的大小是多少？（如果所有的连接结果都往磁盘上回写的话，写操作的基数就等于连接结果集的基数。）

- 一种重要的特殊情况是多对一连接（特别是，一个外码匹配候选码的连接），结果集的基数就等于包含外码的那个连接关系的基数，即 $m$ 或 $n$。
- 更普通的情况是多对多的连接。设 $dCR$ 是关系 $R$ 连接属性 $C$ 的不同值的个数，$dCS$ 是关系 $S$ 连接属性 $C$ 的不同值的个数。如果假设列值均匀分布（uniform value distribution），即任意给定的关系 $R$ 的属性 $C$ 的值可以按任何顺序出现，那么对于给定的 $R$ 的元组，就应当有 $n/dCS$ 个 $S$ 的元组具有相同的属性 $C$ 的值；因此，总共参与连接的基数（也即结果集的大小）就有 $(m * n)/dCS$ 个。或者，如果以 $S$ 为考虑对象进行同样的分析，那么结果集的大小就为 $(m * n)/dCR$。如果 $dCR \neq dCS$，那么这两个估计结果也就不同了，这就是说，如果有些属性 $C$ 值在 $R$ 中出现但在 $S$ 中不出现（反之亦然），就应当使用较小的那个估计值。

在实际中，正如 18.2 节所述，页面（page）I/O 是代价因子，而不是元组 I/O。所以可以假设 $R$ 的元组按 $pR$ 每页存储，$S$ 的元组按 $pS$ 每页存储（这样，两个关系就分别占用 $m/pR$ 页和 $n/pS$ 页）。这样图 18-4 所示的过程就需要 $(m/pR) + (m * n)/pS$ 个页面读操作。或者，如果交换

$R$ 和 $S$ 的位置（即把 $S$ 作外关系，$R$ 作内关系），那么页面读操作数为 $(n/pS) + (n * m)/pR$。

举个例子：假设 $m = 100$，$n = 10\,000$，$pR = 1$，$pS = 10$。那么两个公式计算的结果分别是 100 100 和 1 001 000 个页面读操作。结论：在强制方法中，最好使用小的那个关系作为外关系（这里小的含义是“页面少”）。

归纳一下对强制连接方法的讨论，可得出以下的结论：这是一种最糟糕的算法；它假设关系 $S$ 在属性 $C$ 上既没有索引又没有散列。Bitton 等人所做的实验 [18.6] 表明，如果这个假设成立，那么再动态地创建索引或者散列连接，然后再使用索引查找连接方法或者散列查找连接方法（见下面两小节），一般也可以使连接性能提高。正如在上一节结尾提到的，参考文献 [18.35] 支持这个观点。

**2. 索引查找连接方法**

现在考虑在内关系属性 $S.C$ 上有一个索引 $X$ 的情况（如图 18-5 所示）。这项技术比强制方法优越的地方在于，对外关系 $R$ 的任意一个给定元组，可以直接定位到内关系 $S$ 的对应元组。所以读关系 $R$ 和 $S$ 的基数就为连接结果集的基数；按最糟的情况考虑，即读 $S$ 的每个元组就是读单独一页，那么总共的读页面数为 $(m/pR) + (mn/dCS)$。

```
/* assume index X on S.C */

do i := 1 to m ;                                    /* 外层循环 */
   /* let there be k index entries X[1], ..., X[k] with */
   /* indexed attribute value = R[i].C                  */
   do j := 1 to k ;                                 /* 内层循环 */
      /* let tuple of S indexed by X[j] be S[j]            */
      add joined tuple R[i] * S[j] to result ;
   end ;
end ;
```

图 18-5 索引查找连接方法

如果关系 $S$ 恰好按连接属性 $C$ 顺序存储，那么读页面的数目就减少为 $(m/pR) + (mn/dCS)/pS$。考虑前面提到的那个例子（$m = 100$，$n = 10\,000$，$pR = 1$，$pS = 10$），并假设 $dCS = 100$，那么两个公式计算结果分别为 10 100 和 1 100。这两个数据的差距清楚地表明将关系按一个“好”的顺序物理存储的重要性 [18.7]。

但是，计算代价时必须包括对索引 $X$ 本身进行存取的开销。按最糟的情况考虑，即 $R$ 的每个元组都需要一个“突发的”索引查找来获取对应的 $S$ 元组，也就是说，需要读遍索引的每一层。那么可以假设索引有 $x$ 层，这就带来额外的 $mx$ 个页面读操作。在实际中，$x$ 通常小于等于 3。（由于最高层的索引通常都长驻在主存中，这样就进一步减少了页面读操作次数。）

**3. 散列查找连接方法**

散列查找连接方法类似于索引查找连接方法，不同之处在于，散列查找连接方法建立在内关系 $S$ 属性列 $C$ 上的“快速存取路径”是散列化的列而不是索引（如图 18-6 所示）。代价估算留作练习。

**4. 归并连接方法**

归并连接技术假设关系 $R$ 和 $S$ 都按照连接属性 $C$ 有序物理存储。如果是这样的话，这两个关系就可以作物理顺序扫描，而且这两个扫描可以同步，所以这个连接操作可以一遍扫描完成（至少在一对多的连接情况下这个结论是成立的，在多对多的情况下就未必了）。因为每个页只需存取一次（如图 18-7 所示），所以这个技术显然是最优化的。换句话说，读页面数仅为 $(m/pR) + (n/pS)$。这意味着：

- 逻辑相关的数据在物理上聚集存储是这个算法最重要的性能因素；也就是说，最好将某些对企业十分重要的连接数据按这种方式聚集存储 [18.7]。
- 如果没有建立聚集索引，那么可以在运行时刻首先将其中一个关系或全部两个关系排序，然后使用归并连接算法（排序的作用就是动态地建立这种聚集索引）。这种技术被称为*排序/归并*（Sort/Merge）算法[18.8]。

```
/* assume hash table H on S.C */

do i := 1 to m ;                                          /*外层循环*/
   k := hash (R[i].C) ;
   /* let there be h tuples S[1], ..., S[h] stored at H[k] */
   do j := 1 to h ;                                       /*内层循环*/
      if S[j].C = R[i].C then
      add joined tuple R[i] * S[j] to result ;
   end ;
end ;
```

图 18-6　散列查找连接方法

```
/* 假设R和S均按属性C排序    */
/* 以下代码假设连接为多对多情况 */
/* 较简单的多对一情况留作习题   */

r := 1 ;
s := 1 ;
do while r ≤ m and s ≤ n ;                    /* outer loop */
   v := R[r].C ;
   do j := s by 1 while S[j].C < v ;
   end ;
   s := j ;
   do j := s by 1 while S[j].C = v ;    /* main inner loop */
      do i := r by 1 while R[i].C = v ;
         add joined tuple R[i] * S[j] to result ;
      end ;
   end ;
   s := j ;
   do i := r by 1 while R[i].C = v ;
   end ;
   r := i ;
end ;
```

图 18-7　归并连接方法（多对多情况）

**5. 散列连接方法**

如同前面讨论的归并技术，散列连接技术也只需要对两个关系各自进行一遍扫描（如图18-8所示）。第一遍扫描在 $S$ 表的连接属性 $S.C$ 上建立一个散列表；散列表单元中包含连接属性值（也可能是其他属性值）并且有一个指针指向磁盘上相应的元组。第二遍扫描关系 $R$ 并对连接属性 $R.C$ 使用相同的散列函数。当 $R$ 元组与一个或多个散列表中的 $S$ 元组冲突时，就开始检查 $R.C$ 和 $S.C$ 是否真正相等，如果相等就生成连接元组。这项技术与归并技术相比，优势在于：它并不要求关系 $S$ 和 $R$ 的元组按照某个顺序存储，而且也不需要排序。

```
/* 在S.C上建立散列表H */

do j := 1 to n ;
   k := hash (S[j].C) ;
   add S[j] to hash table entry H[k] ;
end ;

/* 现在对R做散列查找 */
```

图 18-8　散列连接方法

同散列查找连接方法一样，散列连接方法的代价估算留作习题。

## 18.8　小结

对于关系系统来说，优化是挑战又是机遇。实际上，优化能力是关系系统提供的一种功能。一个具有好的优化器的关系系统可以比非关系系统更加有效地执行操作。我们引用的例子中提及的一些改进查询执行的想法可能是可行的（在这个特定的例子中优化比例超过 10 000:1）。优化包括四个大的阶段：

1）将查询转换为一些**内部格式**（通常是**查询树**或者是**抽象语法树**，但是这些表达方式都可以被认为是关系代数或关系运算的内部形式）。

2）使用一些**变换规则**将内部格式转换为**规范格式**。

3）为执行规范格式的查询选择**低层调用**。

4）生成**查询计划**，并且使用**代价公式**以及**数据库统计**信息从中选择出代价最低的查询计划。

然后，我们讨论了通用的**分配律**、**交换律**和**结合律**以及它们在关系操作符（例如连接）中的应用，同时也论及它们在**算术操作符**、**逻辑操作符**以及**比较操作符**中的应用。还讨论了**幂等律**和**吸收律**。也讨论了一些用于**选择操作符**和**投影操作符**的特殊变换规则。然后介绍了**语义**转换的重要思想，即基于数据库**完整性约束**的变换规则。

以 **DB2** 和 **Ingres** 为例概要地说明了一些数据库系统维护的统计信息。然后以 Ingres 原型为例提出了称为**查询分解**的分而治之的策略，这种策略可能对并行处理以及分布式环境非常有用。

最后，我们讨论了关系操作符的**实现技术**，尤其是**连接**操作。我们给出了 5 种连接操作实现方法（**强制连接方法**，**索引查找连接方法**，**散列查找连接方法**，**归并连接方法**（包括**排序/归并方法**），以及**散列连接方法**）的算法伪代码。关于这些连接技术，我们考虑的主要是其实现代价。

遗憾的是，现在的很多产品包含一定的**优化阻碍因素**（optimization inhibitor）。实际上，用户在使用这些产品时也会发现这些问题（在很多情况下，用户几乎没有办法解决这些问题）。优化阻碍因素指的是系统中的某些特性会阻止优化器生成更好的执行计划，比如重复行（见参考文献［6.6］），三值逻辑（见第 19 章）以及使用三值逻辑实现的 SQL（见参考文献［19.6］和［19.10］）。

最后指出一点：在本章中，我们所讨论的优化问题是基于传统的理解和传统实现的，也就是说，我们描述的是一种“传统的智慧”。但是最近，关于 DBMS 实现已经出现了一种全新的方法，该方法使得很多传统的假设变得没有意义。因此，优化过程的很多方面可以被简化（有的甚至可以完全取消），其中包括：

- “基于代价的存取路径选择”的使用（阶段 3 和阶段 4）
- 索引以及其它传统存取路径的使用
- 对于数据库请求是选择编译方法还是解释的方法
- 实现关系操作的算法

等等。附录 A 中有更深入的讨论。

## 习题

18.1 以下这些表达式是对基于“供应商 - 零件 - 工程”数据库的。其中有一些是等价的，有一些则不是。找出等价的表达式。

```
a1. S JOIN ( ( P JOIN J ) WHERE CITY = 'London' )

a2. ( P WHERE CITY = 'London' ) JOIN ( J JOIN S )

b1. ( S MINUS ( ( S JOIN SPJ ) WHERE P# = P# ('P2') )
                { S#, SNAME, STATUS, CITY } ) { S#, CITY }

b2. S { S#, CITY } MINUS
      ( S { S#, CITY } JOIN
        ( SPJ WHERE P# = P# ('P2') ) ) { S#, CITY }

c1. ( S { CITY } MINUS P { CITY } ) MINUS J { CITY }

c2. ( S { CITY } MINUS J { CITY } )
      MINUS ( P { CITY } MINUS J { CITY } )

d1. ( J { CITY } INTERSECT P { CITY } ) UNION S { CITY }

d2. J { CITY } INTERSECT ( S { CITY } UNION P { CITY } )

e1. ( ( SPJ WHERE S# = S# ('S1') )
                       UNION ( SPJ WHERE P# = P# ('P1') ) )
    INTERSECT
    ( ( SPJ WHERE J# = J# ('J1') )
                       UNION ( SPJ WHERE S# = S# ('S1') ) )

e2. ( SPJ WHERE S# = S# ('S1') ) UNION
    ( ( SPJ WHERE P# = P# ('P1') ) INTERSECT
```

```
      ( SPJ WHERE J# = J# ('J1') ) )

f1.  ( S WHERE CITY = 'London' ) UNION ( S WHERE STATUS > 10 )

f2.  S WHERE CITY = 'London' AND STATUS > 10

g1.  ( S { S# } INTERSECT ( SPJ WHERE J# = J# ('J1') ) { S# } )
     UNION ( S WHERE CITY = 'London' ) { S# }

g2.  S { S# } INTERSECT ( ( SPJ WHERE J# = J# ('J1') ) { S# }
                UNION ( S WHERE CITY = 'London' ) { S# } )

h1.  ( SPJ WHERE J# = J# ('J1') ) { S# }
     MINUS ( SPJ WHERE P# = P# ('P1') ) { S# }

h2.  ( ( SPJ WHERE J# = J# ('J1') )
     MINUS ( SPJ WHERE P# = P# ('P1') ) ) { S# }

i1.  S JOIN ( P { CITY } MINUS J { CITY } )

i2.  ( S JOIN P { CITY } ) MINUS ( S JOIN J { CITY } )
```

18.2　说明连接操作、并操作以及交操作是满足交换律的，而差操作不是。

18.3　说明连接操作、并操作以及交操作是满足结合律的，而差操作不是。

18.4　说明：a）并操作在交操作上满足分配律；b）交操作在并操作上满足分配律。

18.5　证明吸收律。

18.6　说明：a）选择操作在并操作、交操作、差操作上无条件满足分配律，而在连接操作上有条件满足分配律；b）投影操作在并操作、交操作上无条件满足分配律，而在连接操作上有条件满足分配律，在差操作上不满足分配律。说明在有条件情况下的关系条件。

18.7　考虑 EXTEND 和 SUMMARIZE 操作，对 18.4 节的变换规则进行扩充。

18.8　为关系除法操作找到一些有用的变换规则。

18.9　给出使用 AND、OR 和 NOT 的条件表达式的一组变换规则。例如“AND 的交换性”即 *A* AND *B* 等价于 *B* AND *A*。

18.10　考虑 EXISTS 和 FORALL 这样的布尔操作，对上一道题的规则进行扩充。例如 8.2 节的一个规则：将 FORALL 操作的表达式转换为一个 NOT EXISTS 操作的表达式。

18.11　以下是“供应商 - 零件 - 工程”数据库的完整性约束（从第 9 章的习题中抽出的一部分）：

- 规定城市只允许有 London、Paris、Rome、Athens、Oslo、Stockholm、Madrid 以及 Amsterdam。
- 同一个城市不允许有两个工程。
- 任何时候在 Athens 至多只有一个供应商。
- 一次发货量不得大于平均发货量的两倍。
- 最高状态的供应商不能与最低状态的供应商在同一个城市中。
- 任何工程必须在至少有一个该工程供应商的城市中。
- 至少有红色零件。
- 平均供应商状态大于 19。
- 每个 London 供应商必须有零件 P2。
- 至少有一个红色零件重量低于 50 磅。
- London 的供应商必须比 Paris 的供应商提供更多种类的零件。
- London 的供应商必须比 Paris 的供应商提供更大量的零件。

下面是一些对该数据库的查询例子：

a. 获取不供应零件 P2 的供应商名单。

b. 获取不供应任何一个在同一城市中的工程的供应商名单。

c. 获取供应零件种类最少的供应商名单。

d. 获取位于 Oslo 的供应商名单，他们提供至少两种不同的 Paris 的零件给至少两个不同的 Stockholm 的工程。

e. 获取供应商名单对，他们提供的零件的产地是互相交错的。

f. 获取供应商名单对，他们供应的工程的地点是互相交错的。

g. 获取零件列表，它们提供给至少一个这样的工程，即该工程的供应商与该工程并不在同一个城市中。

h. 获取供应零件种类最多的供应商名单。

使用自然语言（即不用形式化的表达），利用数据库的完整性约束将这些查询转换为更为简单的形式。

18.12 调查一个 DBMS 系统，看它做了哪些转换工作？是否做了语义转换的工作？

18.13 做一个实验：将一个简单查询（比如“获取供应零件 P2 的供应商的名称”）用查询语言（如 SQL）使用尽可能多的方式表达出来，然后创建一个测试数据库，将这些不同的查询表达式交给这个数据库执行并测试执行时间。如果执行时间差异很大，那么说明优化器的表达式变换工作没有做好。用多个查询重复这个实验，如果可能的话，用多个数据库重复这个实验。

注意：这些查询表达式应当对应相同的查询结果。否则，要么是你的表达式写得有问题，要么是 DBMS 的优化器有问题，如果是后者，就请告知 DBMS 的供应商吧。

18.14 调查一个 DBMS 系统，看它是否维护了一些数据库统计信息（不一定很全）。如果是，有哪些统计信息？这些信息是如何更新的？是动态更新的，还是通过某些工具程序实现的？如果是后者，那么这些工具又是什么呢？它们运行的频率如何呢？这种更新是否具有选择性，即根据特定的统计信息在特定的查询上进行更新？

18.15 在 18.5 节中我们看到 DB2 维护的是每个基表的每一列的第 2 高和第 2 低的列值的统计信息。那么为什么要选第 2 高和第 2 低的列值呢？

18.16 许多商用数据库系统允许用户调整优化器。例如在 DB2 中，用户可以对一个游标使用 OPTIMIZE FOR *n* ROWS 语句，用以指明该游标检索的最大行数不大于 *n*（也就是执行 FETCH 的次数不大于 *n*）。这种语句有时候可以使得优化器选择一条更加有效的存取路径，至少在用户确实执行 FETCH 次数不大于 *n* 的情况下是如此。那么，你认为系统提供这种语句是不是个好办法，为什么？

18.17 参考 18.7 节中设计实现连接操作的方法，设计一套选择操作和投影操作的实现过程。假设不考虑 CPU 时间以及其他的消耗，只把磁盘 I/O 作为唯一的考虑因素，给出其代价公式，并说明代价公式中的其他假设。

18.18 阅读附录 A 并进行讨论。

## 参考文献

优化是一个庞大的研究领域，并且处在不断的发展之中。下面选出的这些文章代表其中的很小一部分。大致可以分为几类：

- 参考文献［18.1～18.6］对优化问题做了介绍、概述并提出一些普遍的优化问题。
- 参考文献［18.7～18.14］关注于一些特定关系操作符的有效实现的问题，如连接操作和求和操作。
- 参考文献［18.15～18.32］描述了一些基于表达式变换的优化技术（见 18.4 节）。特别地，参考文献［18.25～18.28］考虑了语义转换。
- 参考文献［18.33～18.43］讨论了在 System R、DB2 和 Ingres 中使用的优化技术，以及涉及 SQL 嵌套子查询的优化问题。
- 参考文献［18.44～18.62］提出了各式各样的优化技术、技巧以及未来的研究方向等。特别地，参考文献［18.55～18.58］考虑了并行处理对优化技术带来的冲击。

注意：有关分布式系统和决策支持系统的查询优化技术的参考文献没有包括在内，分别见第 21 和 22 章。

［18.1］ Won Kim, David S. Reiner, and Don S. Batory(eds): *Query Processing in Database Systems.* New York, N. Y.: Springer-Verlag(1985).

这本书是讨论查询处理（而不仅是查询优化）的普遍问题的一组论文集。它包括由 Jarke、Koch 和 Schmidt 写的介绍性综述（类似于参考文献［18.2］但不相同），然后是讨论不同情况下查询处理问题的一组论文，如：分布式数据库、异质数据库系统、视图更新（参考文献［10.8］是这一节中关于这个题目的唯一文献）、非传统应用（如 CAD/CAM）、多语句优化（见参考文献［18.47］）、数据库机器以及物理数据库设计。

［18.2］ Matthias Jarke and Jürgen Koch: “Query Optimization in Database Systems,” *ACM Comp. Surv. 16*, No. 2(June 1984).

本文是一篇优秀的指南。这篇论文给出了查询处理的一个通用框架，有点像本章 18.3 节，但它是基于关系演算的，而非关系代数。在框架内，文章讨论了大量的优化技术，如：语法和语义的转换，低层操作实现以及产生查询计划和从中选择的算法。文章还给出了针对关系演算

的语法转换规则的大集合。还附了一个相当长的没有注释的文献列表；但是注意，自 1984 年以来，有关优化问题的文章至少在数量上比那时增加了一个数量级。

这篇论文还简要讨论了其他的相关问题：高层次查询语言的优化（即比关系代数或关系演算更加有表达力的语言）、在分布环境中的查询优化和有关优化的数据库机器问题。

[18.3] Götz Graefe："Query Evaluation Techniques for Large Databases," *ACM Comp. Surv. 25*, No. 2 (June 1993).

本文是又一篇优秀的指南，而且比参考文献［18.2］更新，它列出的文献也很多。这里引用其摘要："这篇综述提供了设计和实现查询执行工具的基础……它叙述了许多实用的查询计算技术……包括复杂查询的迭代执行技术、基于排序和散列的集合匹配算法的二元性，各种并行查询执行及其实现技术，以及未来数据库应用领域的特殊的操作符问题"。推荐这篇论文。

[18.4] Frank P. Palermo："A Data Base Search Problem," In Julius T. Tou (ed.), *Information Systems: COINS IV*. New York, N. Y.: Plenum Press (1974).

最早的关于优化的经典论文之一。文章从一个关系演算表达式开始，第一次引用 Codd 的归约算法（reduction algorithm），用于将这个表达式简化为一个等价的关系代数表达式（见第 8 章），然后对该算法进行了一系列的改进，下面是其中一些：

- 同一个元组不得多次去获取。
- 在获取元组的同时将不必要的值去掉。这里"不必要的值"指的是在查询中没有使用的列值或者是单用于选择操作的值。这个过程等价于投影"必要的"列，使得不仅减少每个元组所需的空间，而且减少需要保留的元组的个数（在通常情况下）。
- 用于建立结果关系的方法是基于最小增长原则（least-growth principle）的，这样就使得结果集增长缓慢。这个技术可以减少涉及的比较操作的次数以及需要存放中间结果集的存储空间。
- 构造连接时采用了一种高效的技术，包括（a）动态地将连接项（如 S. S# = SP. S#）的值分解为半连接（semi join），这样就可以有效地动态构造 2 级索引（Palermo 的半连接不同于第 7 章中的半连接，见第 7 章）；（b）使用称为非直接连接（indirect join）的内部方式表达每个连接操作，这样就可以利用内部的元组 ID 来表征每个参与连接的元组。通过保证每个连接参与元组在逻辑上按连接属性排序，这些技术被设计用于减少构造连接时的扫描工作量。它们也允许动态决定存取所需关系的"最优"序列。

[18.5] Meikel Poess and Chris Floyd："New TPC Benchmarks for Decision Support and Web Commerce," *ACM SIGMOD Record 29*, No. 4 (December 2000).

TPC 的意思是事务处理委员会（Transaction Processing Council），这是一个近年来发布了多项工业级标准的独立机构。TPC - C（基于一个发货/进货系统建模）是一个测量 OLTP 性能的标准。TPC - H 和 TPC - R 是决策支持的标准，分别用来评价特别查询（TPC - H）和计划报告（TPC - R）的性能。TPC - W 用于测量电子商务环境下的性能。网站 http://www.tpc.org 上有更加详细的信息，包括大量实际标准的案例。

[18.6] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill："Benchmarking Database Systems: A Systematic Approach," Proc. 9th Int. Conf. on Very Large Data Bases, Florence, Italy (October/November 1983).

这是第一篇描述现在被称为"Wisconsin Benchmark"（因为它是由在威斯康星大学的论文作者开发的）的论文。这个基准测试定义了精确定义属性值的一组关系，并用于测量这些关系上已精确定义的各种关系代数操作的性能（例如，各种投影，包括不同程度的列值重复情况下的投影等）。这个基准是对基本的关系操作的系统测试。

[18.7] M. W. Blasgen and K. P. Eswaran："Storage and Access in Relational Databases," *IBM Sys. J. 16*, No. 4 (1977).

许多查询处理技术，包括选择、投影和连接都是基于磁盘 I/O 进行比较的。这些技术基本上在 System R［18.33］中实现。

[18.8] T. H. Merrett："Why Sort/Merge Gives the Best Implementation of the Natural Join," *ACM SIGMOD Record 13*, No. 2 (January 1983).

文章提供了一些直觉的观点来支持如标题所示的看法。这些观点有：

a. 如果两个连接的关系在连接属性上是有序的，那么连接操作本身就是非常高效的（原因

是正如我们在 18.7 节所看到的，归并连接方法在这种情况下改进最为明显，因为归并连接方法要求每个页面都要存取且只存取一次，这样就得到了优化）。

b. 在足够大的机器上，将关系元组按某种顺序排序的代价很可能要小于在元组非有序情况下所额外付出的代价。

但是，作者承认会有不符合他那些有争议的观点的例外情况发生。比如说其中一个关系很小（例如它是一个选择操作的结果集），那么通过索引或者散列方法直接存取这个关系可能就比先排序要高效得多。参考文献［18.9－18.11］给出了更多的例子说明在什么情况下排序/归并并不是最好的办法。

［18.9］ Giovanni Maria Sacco："Fragmentation: A Technique for Efficient Query Processing," *ACM TODS 11*, No. 2 (June 1986).

文章提出了"分而治之"的执行连接操作的方法。这种方法是将关系递归地分割为相互分离的供选择操作的子集（"碎片"），并且在这些子集上面执行一系列的顺序扫描。这种方法不同于排序/归并方法的地方在于，它不要求这个关系必须是有序的。这篇论文提出了：如果参与连接操作的一个关系或者两个关系都需要实现排序，那么这种分割技术的效率将优于排序/归并方法。作者还认为这种技术可以适用于实现其他的操作，例如交操作和差操作。

［18.10］ Leonard D. Shapiro："Join Processing in Database Systems with Large Main Memories," *ACM TODS 11*, No. 3 (September 1986).

文章提出了三种散列连接操作算法。其中一个是"当可用的主存能够整个装入其中一个连接关系时，这种方法尤其有效"。这种方法是通过将关系分解为能够在主存中处理的相互分离的分片（也就是选择操作子集）来实现的。作者认为，从主存降价的趋势来看，散列连接算法必将成为一种有效的算法。

［18.11］ M. Negri and G. Pelagatti："Distributive Join: A New Algorithm for Joining Relations," *ACM TODS 16*, No. 4 (December 1991)

文章提出了另一种"分而治之"的执行连接操作的方法。"（这种方法）是基于这样一个想法……不必要将两个关系都完全排序……只要保证其中一个完全有序而另外一个部分有序即可，从而避免了部分的排序工作"。部分排序将一个关系分解为若干个无序的分片 *P*1，*P*2，…，*Pn*（有点像 Sacco 在参考文献［18.9］中提到的方法，但 Sacco 使用的是散列方法而不是排序），同时，这些分片满足关系 MAX（*Pi*）＜MIN（*P*（*i*+1）），*i* = 1，2，…，*n*－1。本文认为这种方法要优于排序/归并方法。

［18.12］ Göz Graefe and Richard L. Cole："Fast Algorithms for Universal Quantification in Large Databases," *ACM TODS 20*, No. 2 (June 1995).

在 SQL 中没有直接支持 FORALL 全称量词，所以商用系统也不提供这种运算，但是在构成很大一类查询时，该操作符是很有用的。这篇论文描述和比较了"三个已知的和一个刚刚提出的实现关系除法的算法，而关系除法是包含了 FORALL 全称量词的实现的"。同时，本文说明了这种新的算法"在同一关系上做存在量词运算时和散列连接（或半连接）一样快"（稍微有点不同于原文）。作者的结论是，相比其他的语言结构而言，用户语言应当直接支持 FORALL 全称量词，这是因为许多优化器"没有认识到 SQL 中存在相当不直接的表达方式"。

［18.13］ David Simmen, Eugene Shekita, and Timothy Malkemus："Fundamental Techniques for Order Optimization," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

文章提出了用于减少或避免排序的技术。这个技术部分依赖于 Darwen 的工作［11.7］，并且在 DB2 中得以实现。

［18.14］ Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay："Approximate Medians and Other Quantiles in One Pass and with Limited Memory," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

［18.15］ Cĕsar A. Galindo-Legaria and Milind M. Joshi："Orthogonal Optimization of Subqueries and Aggregation," Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

［18.16］ James Miles Smith and Philip Yen-Tang Chang："Optimizing the Performance of a Relational Algebra Database Interface," *CACM* 18, No. 10 (October 1975).

文章描述了在“Smart Query Interfaces for a Relational Algebra,（关系代数的灵巧查询界面，SQUIRAL）中使用的算法。所使用的技术包括：

- 根据 18.4 节中讨论的方法，将原算术表达式转换为一个等价的、但更为高效的操作序列。
- 将这些操作符分配给单独的进程，并且使用这些进程的并发和流水线操作。
- 协调传递给这些进程的临时关系的排列顺序。
- 使用索引，并且尽量将页面引用局部化。

这篇论文和参考文献［18.17］是最早讨论表达式变换的论文。

［18.17］ P. A. V. Hall:“Optimization of a Single Relational Expression in a Relational Data Base System,” *IBM J. R&D* 20, No. 3(May 1976).

文章描述了一些在系统 PRTV［7.9］中使用的优化技术。类似于 SQUIRAL［18.16］，PRTV 一开始也是在计算给定的表达式之前，将原算术表达式转换为一个等价的、但更为高效的形式（该文也是最早讨论表达式转换的论文之一）。PRTV 的一个特征是，当系统收到一个表达式时，它并不立即开始进行计算，而是将计算工作推迟到最后时刻（见第 7 章 7.5 节的查询表达讨论）。因此文章的标题“single relational expression（单关系表达式）”就恰好表征了用户操作的整个顺序。本文论及的优化技术和 SQUIRAL 的类似，但在某些方面更为进步，包括：

- 尽早做选择操作
- 多个投影操作组成的序列可以合并为一个投影操作
- 去掉冗余的操作
- 简化空关系和琐碎的条件
- 分解出相同的子表达式

该文用一些实验数据和对进一步研究的建议作为结尾。

［18.18］ Matthias Jarke and Jürgen Koch:“Range Nesting: A Fast Method of Evaluate Quantified Queries,” Proc. 1983 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif.(May 1983).

文章定义了关系演算的一个变种，它允许使用一些附加的变换规则，并且为这些关系演算表达式提供了计算算法（这种特别的关系演算非常类似于第 8 章的元组演算）。文章描述了对这种变种关系演算一类表达式的优化，这类表达式称为“完全的嵌套表达式”（perfect nested expression）。文章给出了将复杂的查询（比如说含有 FORALL 全称量词）转换为完全表达式的方法。作者说明了实际当中产生的一个查询集合是可以对应到完全表达式的。

［18.19］ Surajit Chaudhuri and Kyuseok Shim:“Including Group-By in Query Optimization,” Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile(September 1994).

［18.20］ A. Makinouchi, M. Tezuka, H. Kitakami, and S. Adachi:“The Optimization Strategy for Query Evaluation in RDB/V1,” Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France(September 1981).

RDB/V1 是富士通公司的 AIM/RDB 这个 SQL 数据库系统的原型。这篇文章描述了这个原型系统使用的优化技术，并且简要地将它同 System R 和 Ingres 的原型系统进行了比较。其中有一个革新性的技术是使用动态包含最大和最小值来引入额外的选择操作。这个技术简化了选择连接顺序的过程，并且改善了连接的性能。一个改善性能的例子是：假设供应商表和零件表在城市列上做连接。首先，供应商表按 CITY 排序；在排序过程中，可以知道 S.CITY 最大值和最小值（设为 HIGH 和 LOW）。然后选择操作

```
LOW ≤ P.CITY AND P.CITY ≤ HIGH
```

能够用来减少一部分在连接操作时使用的零件表记录。

［18.21］ Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan:“Extensible Rule Based Query Rewrite Optimization in Starburst,” Proc. 1992 ACM SIGMOD Int. Conf. on Management of Data, San Diego, Calif.(June 1992).

正如在 18.1 节中提到的那样，“查询重写”是表达式转换的别名。作者感叹商用系统居然在这方面没有做什么工作（至少在 1992 年以前是如此）。文章描述了 IBM Starburst 原型系统的查询转换的工作机制（见参考文献［18.48, 26.19, 26.23, 26.29, 26.30］），即熟练的用户在任何时候都可以将表达式转换规则添加到系统中（也就是文章题目中“extensible”（可扩展的）

的含义)。

[18.22] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan: "Magic Is Relevant," Proc. 1990 ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, N. J. (May 1990).

标题中有一个并不十分贴切的术语"magic"指的是"逻辑数据库"语言 Datalog (见第 24 章) 中使用的查询优化技术, 特别是包含递归的查询。文章将这一技术扩展到一般的关系系统, 在实验数据的基础上, 认为这种技术要比传统的优化技术有效 (注意, 为了尽可能实用, 查询不一定要递归)。这种技术的基本思想是将给定的查询分解为一组定义为"附加关系"的小的查询 (有点像 18.6 节讨论的查询分解方法), 这样就将一些与查询无关的元组过滤掉了。下面给出一个基于这篇论文的例子 (用关系演算表达)。原查询是:

```
{ EX.ENAME }
     WHERE EX.JOB = 'Clerk' AND
           EX.SAL > AVG ( EY WHERE EY.DEPT# = EX.DEPT#, SAL )
```

("查找那些工资高于所在部门平均工资的员工的名字")。如果这个查询被直接执行, 那么系统会对所有的部门一个接一个元组地扫描员工表, 从而计算雇佣员工超过 1 人的部门的平均工资许多次。一个传统的优化器会将这个查询分解为两个小的查询:

```
WITH { EX.DEPT#,
       AVG ( EY WHERE
             EY.DEPT# = EX.DEPT#, SAL ) AS ASAL } AS T1 :

{ EMP.ENAME } WHERE EMP.JOB = 'Clerk' AND
                    EXISTS T1 ( EMP.DEPT# = T1.DEPT# AND
                                EMP.SALARY > T1.ASAL )
```

现在不必将每个部门的平均工资计算多遍了, 但是会计算一些无关的平均值, 也就是那些没有雇佣员工的部门。

这个所谓的"魔集" (magic) 方法能够避免第一种方法的重复计算, 也能避免第二种方法的无关计算。其代价是生成"附加的"关系:

```
/* first auxiliary relation : name, department, and salary  */第一个附加关系: 员工
                                                              的姓名, 部门和工资
                                                           */
/* for clerks
WITH ( { EMP.ENAME, EMP.DEPT#, EMP.SAL }
       WHERE EMP.JOB = 'Clerk' ) AS T1 :

/* second auxiliary relation : departments employing clerks */第二个附加关系: 雇佣
WITH { T1.DEPT# } AS T2 :                                    员工的部门

/* third auxiliary relation : departments employing clerks  */第三个附加关系: 雇佣
                                                              员工的部门及相应的平
                                                              均工资
/* and corresponding average salaries                     */
WITH ( { T2.DEPT#,
         AVG ( EMP WHERE
               EMP.DEPT# = T2.DEPT#, SAL ) AS ASAL } ) AS T3 :

/* result relation */
{ T1.ENAME } WHERE EXISTS T3 ( T1.DEPT# = T3.DEPT# AND
                               T1.SAL > T3.ASAL )
```

"魔集"包括确定究竟需要哪个附加的关系。

对"魔集"进一步的引用见参考文献 [18.23, 18.24] 以及第 24 章的"参考文献"部分。

[18.23] Inderpal Singh Mumick and Hamid Pirahesh: "Implementation of Magic in Starburst," Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, Minn. (May 1994).

[18.24] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan: "Magic Conditions," *ACM TODS 21*, No. 1 (March 1996).

[18.25] Jonathan J. King: "QUIST: A System for Semantic Query Optimization in Relational Databases," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

这篇论文引入了语义优化的思想 (见本章 18.4 节)。文章描述了一个能够做这类优化的实验系统 QUIST (QUery Improvement through Semantic Transformation, 通过语义转换的查询改

进)。

[18.26] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu: "A System for Semantic QUery Optimization," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May/June 1987).

作者通过引入一种模式，该模式能够动态地从非常大的完整性约束条件集合中选择那些有可能对转换一个给定查询有利的那些约束条件，从而对 King [18.25] 的工作进行扩展。完整性约束有两个基本的类别，即蕴涵约束（implication constraint）和子集约束（subset constraint）。这样的约束条件用于通过减少冗余的选择以及连接操作，并在有索引的列上引入附加的选择操作而转换查询。仅仅通过约束条件本身就能回答的查询通过这种办法也是很高效的。

[18.27] Michael Siegel, Edward Sciore, and Sharon Salveter: "A Method for Automatic Rule Derivation to Support Semantic Query Optimization," *ACM TODS 17*, No. 4 (December 1992).

正如在 18.4 节中提到的，语义优化利用完整性约束来转换查询。但是，这种方法有一些问题：

- 优化器如何知道哪些转换将会有效（也就是使得查询更加高效）？
- 有些完整性约束对于优化来说并不非常有用。例如，一个约束条件是零件的重量必须大于 0。尽管这个条件对于完整性来说是重要的，但是对于优化却没有什么用。优化器如何区别有用的和没有用的约束条件？
- 有些条件对于数据库的某些状态来说是有效的，甚至对于绝大多数状态来说是有效的，因此这些条件对于优化是有用的。但是这些条件不是严格的完整性约束。比方说条件"员工年龄不大于 50"，虽然它不是一个完整性约束条件，也就是员工年龄可以大于 50，但在实际上没有一个员工年龄大于 50 的情况下也是可以的。

文章描述了针对上述问题的系统的体系结构。

[18.28] Upen S. Chakravarthy, John Grant, and Jack Minker: "Logic Based Approach to Semantic Query Optimization," *ACM TODS 15*, No. 2 (June 1990).

这篇文章的摘要中提到："在几篇早期的论文中，（作者）描述并证明了语义查询优化方法的正确性……本文巩固了那些论文的主要结论，强调了在关系查询中的优化技术及其应用。另外，（本文还表明）该方法包含和总结了早期语义查询优化的成果。同时，（还说明了）语义查询优化技术可以扩展到（递归查询和）包含析取操作、求反操作以及递归运算的完整性约束条件。"

[18.29] Qi Cheng *et al.*: "Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database," Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland (September 1999).

[18.30] A. V. Aho, Y. Sagiv, and J. D. Ullman: "Efficient Optimization of a Class of Relational Expressions," *ACM TODS 4*, No. 4 (December 1979).

文章标题中的"relational expressions"指的是只包含等值选择条件、投影和自然连接（也称为 SPJ 表达式）的查询。SPJ 表达式相应于只包含等值比较、AND 操作以及存在量词运算的关系演算。该文引入*表关系*（tableau）作为 SPJ 表达式类型查询的方法。一个 tableau 是一个矩阵数组，它的列对应表的列属性，它的行对应条件：特别地，对应成员资格条件（membership condition），即说明某个（或者子元组）元组是否存在于某个关系中。每一行都通过出现的相同符号逻辑关联起来。例如，表示查询"获取供应某种红色零件（*b*2）的，在 London 的供应商（*b*1）的状态（*f*1）"的 tableau 如下：

```
S#    STATUS   CITY     P#   COLOR
-------------------------------------
          f1
-------------------------------------
b1        f1   London                 - suppliers
b1                      b2           - shipments
                        b2   Red     - parts
```

这个 tableau 最上面一行显示所有在查询中涉及的列属性，下一行是"累计"行（对应于关系演算查询的原型元组（proto tuple）或关系代数的最终投影），剩下的行代表成员资格条件。我们已经在例子中用相关关系（或者关系变量）标出了这些行。*注意*，"*b*"代表绑定的变量，"*f*"代表自由变量；累计行只包含"*f*"。

Tableau 代表了将查询转换为规范格式的又一候选方案（见 18.3 节），但它们还不足以表达所有的关系查询（事实上，可以把它们看作 QBE 的语法的一个变种，但确实不如 QBE 有用）

该文给出了将一个 tableau 简化为另一个语义上等价的 tableau 的算法，通过这个算法能够将 tableau 的行数减至最少。因为 tableau 的行数（不算最上面特殊的两行）要比 SPJ 表达式的连接数多，所以转换后的 tableau 代表了查询的优化形式。称之为优化形式的一个方面就是因为将连接最小化了（在上面的例子中，连接的数目已经是最小的了，优化就没有效果了）。被最小化的 tableau 还能够被转换回去，因为有时候需要转换为别的表示方式以便于随后的优化。

最小化连接数目的思想在处理连接视图的查询时可以实际地应用。特别地，使用“通用关系”（universal relation）构造的查询也是如此，见第 13 章“参考文献”部分。例如，假设用户定义一个视图 V，由供应商和发货量在 S#属性上做连接构成，然后用户提出查询：

```
V { P# }
```

一种直接的视图处理算法把这个查询转换为如下形式：

```
( SP JOIN S ) { P# }
```

但是，在 18.4 节已经提到，下面这个查询会产生相同的结果集，而且不必做连接（也就是连接数目被最小化）：

```
SP { P# }
```

注意，因此可以说，因为文中给出的 tableau 的简化算法考虑了任何显式声明的列之间的函数依赖，所以这些算法只是语义优化技术的一些有限的例子。

[18.31] Y. Sagiv and M. Yannakakis:“Equivalences Among Relational Expressions with the Union and Difference Operators,” *JACM 27*, No. 4(October 1980).

扩展了参考文献 [18.30] 的思想，把使用并操作和差操作的查询包含在内了。

[18.32] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv: “Query Optimization by Predicate Movearound,” Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile(September 1994).

[18.33] P. Griffiths Selinger. *et al*: “Access Path Selection in a Relational Database System,” Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (May/June 1979).

这篇开创性的论文讨论了在 System R 原型中使用的一些优化技术。System R 的查询是 SQL 语句形式的，也就是由一组“SELECT-FROM-WHERE”查询块组成，而这些查询块可能是嵌套在一起的。System R 的优化器首先决定按照什么样的顺序来执行这些查询块，然后通过选择最小代价的查询块执行算法使得查询的代价最小。注意这种策略（先选择查询块执行顺序，再优化每个查询块）意味着某些可能的查询计划永远不会被优化器考虑到。实际上，它就构成了“减小查询计划搜索空间”的技术（见 18.3 节中关于这个问题的评注）。注意：在存在嵌套查询块的情况下，优化器就按照用户指定的嵌套顺序执行，宽泛地说，就是最内的块最先执行。见参考文献 [18.37-18.43] 对这种策略的更深入讨论和批评。

对于一个给定的查询块，有两种基本的情况要考虑（第一种情况实际上是第二种情况的特例）：

1）对于只涉及单个关系的选择操作和/或投影操作的查询块，优化器通过使用目录表的统计信息，以及用于估计中间结果集大小的公式（文中已给出）和对低层操作的代价计算，选择出一个执行该查询块的策略。

2）对于涉及两个或两个以上关系连接，以及块内选择和/或投影操作的查询块，优化器：(a) 把每个独立的关系按照上一种情况处理；(b) 决定一个执行连接的顺序。操作 (a) 和操作 (b) 不互相独立；例如，一个给定的策略，比如说，使用某个索引来存取独立的关系 *A* 刚好被选中，因为它产生 *A* 的元组的顺序正好用于随后同关系 *B* 做连接操作。

连接操作可以通过排序/合并、索引查找或者强制方法来实现。需要指出的一点是，对嵌套连接 (*A* JOIN *B*) JOIN *C* 的计算，无需求出 *A* 与 *B* 连接的全部结果后再求与 *C* 的连接；相反，可以在 *A* JOIN *B* 产生一个连接元组后，立即求与 *C* 的连接。因此，根本没有必要将 *A* JOIN *B* 所产生的全部结果进行物化。（在 3.2 节介绍过这一流水线的思想。还可参见参考文献 [18.16] 和 [18.58]。）

文章还讨论了几个优化代价的问题。对两个关系的连接，其代价约为 5 ~ 20 次数据库访问的代价。对需要多次反复执行的优化后的查询而言，这一代价可以忽略不计（注意：System R 为一编译系统，因此其 SQL 语句可以一次优化，多次执行）。复杂查询的优化代价在 IBM Sys-

tem 370 Model 158 上需要“几千字节的存储空间和十分之几秒的时间”。“8 个表的连接在几秒内可以完成优化”。

[18.34] Eugene Wong and Karel Youssefi: "Decomposition—A Strategy for Query Processing," *ACM TODS 1*, No. 3 (September 1976).

[18.35] Karel Youssefi and Eugene Wong: "Query Processing in a Relational Database Management System," Proc. 5th Int. Conf. on Very Large Data Bases, Rio de Janeiro, Brazil (September 1979).

[18.36] Lawrence A. Rowe and Michael Stonebraker: "The Commercial Ingres Epilogue," in reference [8.10].

"商用 Ingres" 是基于 "大学 Ingres" 原型系统的产品。下面列出的是商用 Ingres 系统和大学 Ingres 系统在优化器上的一些差别:

1) 大学系统使用 "增量计划" 的方法，也就是说，它决定首先应当做什么，然后执行这个决定，接着基于刚才的结果集大小决定下一步该做什么，依此类推。而商用系统在开始执行之前先决定整个的执行计划，这个决定过程是基于对中间结果集大小的估计进行的。

2) 大学系统的优化器使用在 18.6 节中讨论的元组置换的方法处理两变量查询（也就是连接）。而商用系统支持一些优选技术的变种用以处理这类查询，包括在 18.7 节中讨论的排序/归并方法。

3) 商用系统使用更加复杂的统计信息。

4) 正如在第 1) 点中提到的，大学系统使用增量计划的方法。商用系统使用更为彻底的查询计划搜索。但是，如果用于优化的时间已经超过其估计的最优查询执行时间，那么搜索过程就会停止（否则优化的代价就大于收益了）。

5) 商用系统考虑所有可能的索引组合，所有可能的连接顺序，以及所有可用的连接方法——比如排序/归并法、部分排序/归并法、散列查找法、ISAM 查找法、B 树查找法以及强制连接方法（见 18.7 节）。

[18.37] Won Kim: "On Optimizing an SQL-Like Nested Query," *ACM TODS* 7, No. 3 (September 1982).

见下面参考文献 [18.41] 的注释。

[18.38] Werner Kiessling: "On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates," Proc. 11th Int. Conf. on Very Large Data Bases, Stockholm, Sweden (August 1985).

见下面参考文献[18.41]的注释。

[18.39] Richard A. Ganski and Harry K. T. Wong: "Optimization of Nested SQL Queries Revisited," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

见下面参考文献[18.41]的注释。

[18.40] Günter von Bültzingsloewen: "Translating and Optimizing SQL Queries Having Aggregates," Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

见下面参考文献[18.41]的注释。

[18.41] M. Muralikrishna: "Improved Unnesting Algorithms for Join Aggregate SQL Queries," Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada (August 1992).

SQL 语言允许 "嵌套子查询"，宽泛地说，就是一个 SELECT-FROM-WHERE 查询块中包含着另外一个这样的查询块（见第 8 章）。这种结构对于实现来说很不利。考虑如下这个查询（"获取供应零件 P2 的供应商的名字"）。我们将它称为查询 Q1:

```
SELECT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = P# ('P2') ) ;
```

在 System R [18.33] 中执行这个查询时，首先计算内块而得到一个临时表 *T*，*T* 包含所要求的供应商号；然后逐行扫描 *S* 表，对每个元组扫描 *T* 表，看 *T* 表是否包含了相应的 *S* 表供应商号。这种策略显然是低效的（尤其在 *T* 表没有建立索引时）。

现在考虑查询 *Q2*:

```
SELECT S.SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = P# ('P2') ;
```

这个查询显然在语义上同 *Q*1 等价，但 System *R* 能够为它考虑更多的执行策略。特别地，如果 *S* 表和 *SP* 表都恰好按照供应商号有序存储，那么 System R 就会使用非常高效的归并连接（merge join）算法。而且（a）这两个查询是逻辑上等价的；（b）第 2 个查询在执行上更为有效，所以将 *Q*1 转换为 *Q*2 的这种可能性就值得研究。这种可能性的研究见参考文献［18.37 - 18.43］。

Kim［18.37］是第一个论述这个问题的。他区分了 5 种嵌套查询，并给出了相应的转换算法。他的论文中有些实验数据证实了这些转换算法可以提高执行效率达 1 ~ 2 个数量级。

而后，Kiessling［18.38］证实了在任何一级的嵌套子查询中的 SELECT 子句中有 COUNT 操作符时，Kim 的算法就不正确了。这是因为 Kim 的算法没有正确处理在一个空集合上进行 COUNT 运算的情况。文章题目中的“semantic reefs”（语义暗礁）就指出了 SQL 的脆弱性和复杂性，以致于用户对于这些查询必须绕开一些问题才能保证其结果的正确性和一致性。Kiessling 进一步指出 Kim 的算法是不易纠正的（“无法在所有的情况下都进行这种转换”）。

Ganski 和 Wong［18.39］提出了 Kiessling 发现的问题的一个解决方案。这个方法通过在转换后的查询中使用外连接（见第 19 章）而非常规的内连接来实现（即使本文作者也认为这个解决办法并不是很令人满意，因为它要求在转换后的查询中的操作符需要有一定的顺序关系）。这篇论文还指出了 Kim 的原文中的另一个错误，并使用同样的办法纠正了这个错误。但是，这篇论文的转换算法也有一些错误，其中有些和重复行问题相关（重复行问题是一个很糟糕的“semantic reef”），另一些则和 SQL 存在量词有关（参见第 19 章）。

von Bültzingsloewen 的论文［18.40］代表了试图将整个问题放入理论上合理的地位的尝试（问题就如许多作者研究的那样，不论是从语法的角度或是语义的角度，SQL 形式的嵌套和聚集都没有被很好地理解）。该文定义了关系演算和关系代数的扩展版本（同聚集和空值有关），并且证明了这两个版本形式是等价的（使用了比原先更为优美的证明方法）。文章还通过将 SQL 映射到刚才提到的关系演算上，定义了 SQL 的语义。但是请注意：

1）文中讨论的 SQL 语言，虽然比参考文献［18.37 - 18.39］中讨论的 SQL 距离商用系统中使用的 SQL 要更接近一些，但还不够正统：它不包括 UNION，也不直接支持形如“ = ALL”或“ > ALL”的操作符（见附录 B），而且它处理未知真值的方法不同于（实际上要优于）原 SQL 语言（参见第 19 章）。

2）这篇论文未考虑删除重复的问题，这样做是“为了技术上的简化”。如果考虑到（像上面提到的那样）重复行会导致转换的正确与否的问题，这样做的真正含义就不清楚了［6.6］。

最后，Muralikrishna［18.41］认为 Kim 原来的算法［18.37］虽然不正确，但仍然比参考文献［18.39］的“通用策略”在某些情况下要高效一些。因此提出了 Kim 算法的另一个纠正版本。这个版本同时也提出了一些改进。

［18.42］ Lars Baekgaard and Leo Mark: "Incremental Computation of Nested Relational Query Expressions," *ACM TODS 20*, No. 2 (June 1995).

这是另一篇讨论包含 SQL 形式的子查询的优化问题的文章。文章特别讨论了相关子查询的问题（文章标题中的“嵌套”特指 SQL 形式的嵌套子查询）。文中提出的策略是（1）将原查询转换为一个非嵌套的等价查询；（2）然后增量计算这个非嵌套查询的代价。“为了支持步骤 1，我们开发了一个简明的代数 - 代数的转换算法……（转换后）的查询使用了密集的（MIN US）操作符。为了支持步骤 2，我们提出和分析了一个高效的、用于增量计算（MINUS 操作）的算法。”术语“增量计算”指的是可以利用前面的结果来计算一个查询的代价。

［18.43］ Jun Rao and Kenneth A. Ross: "Using Invariants: A New Strategy for Correlated Queries," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

这是又一篇讨论包含 SQL 形式的子查询的优化问题的文章。

［18.44］ David H. D. Warren: "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

文章从一个不同的角度，也就是形式逻辑的角度来讨论查询优化的问题，这篇论文报告了基于 Prolog 实现的实验数据库系统的技术。这些技术和 System R 使用的技术是独立开发的，而且源于不同的目的，但它们非常相似。这篇论文认为，与普通的查询语言如 QUEL 和 SQL 相比，基于逻辑的语言（如 Prolog 等）能够更为强调以下的形式：

- 查询的必要成分是什么——即逻辑目标
- 连接这些成分的是什么——即逻辑变量
- 实现的关键问题是什么——即达到目标的顺序

因此，作者认为使用这样的语言可以更为方便地进行优化。实际上，这样的语言可以被认为是查询内部表达方式的另一个候选（参见18.3节）。

[18.45] Yannis E. Ioannidis and Eugene Wong: "Query Optimization by Simulated Annealing," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

查询计划的搜索空间会随着查询涉及的关系数量的增加而呈现指数级的增长。在通常的商业应用中，一般查询涉及的关系数很少，所以搜索空间也就在可以接受的范围内。但在一些新的应用中，查询涉及的关系数可能会非常大（见第22章的例子）。而且，这些应用可能很需要做"全局"（也就是多查询的）优化［18.47］以及需要递归查询支持，这样更加导致了搜索空间的显著增大。在这样的环境中，穷举式的搜索就行不通了，就必然需要一些有效的、能够减少搜索空间的技术。

这篇文章引用了在多关系查询和多查询情况下的优化的文章，但指出没有一个算法是针对递归查询优化的。然后，文章提出了一个作者认为是对任意大搜索空间都适用的算法，而且特别显示了如何将这种算法用于递归查询的情况。这个算法（称为"模拟退火算法"（simulated annealing）是因为它模拟了晶体退火的过程，这个过程中首先让浸泡晶体的液体加热然后使之逐渐冷却）是个概率性的、爬山算法。在别的文章中，它已经被成功地用于优化问题。

[18.46] Arun Swami and Anoop Gupta: "Optimization of Large Join Queries," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, Ill. (June 1988).

在含有多个关系的查询中决定连接的顺序问题（如在演绎数据库中的应用，见第24章）是个组合数学难题。这篇论文比较了一些有关该问题的算法：随机行走（perturbation walk）、准随机取样（quasi-random sampling）、迭代改进（iterative improvement）、顺序启发式算法（sequence heuristic）以及模拟退火过程（simulated annealing）［18.45］。根据这个分析，迭代改进的算法要比其他的算法优秀；特别地，模拟退火算法就其"本身"而言，对于大连接查询是没有用处的。

[18.47] Timos K. Sellis: "Multiple-Query Optimization," *ACM TODS 13*, No. 1 (March 1988).

经典的查询优化计划搜索空间是针对独立的、单个的关系表达式的。但是，将多个不同的查询作为一个整体同时进行优化的能力将会十分重要。之所以这样的一个原因是，在高层应用的一个单独的查询在关系层看来却包含多个查询。例如，一个自然语言表达的查询"Mike拿的薪水还不错吧?"也许会引起3个单独的关系查询的执行：

- "Mike拿的薪水超过75 000美元?"
- "Mike拿的薪水超过60 000美元，但他的工作经验少于5年?"
- "Mike拿的薪水超过45 000美元，但他的工作经验少于3年?"

这个例子显示了一组有关的查询很可能有些相同的子表达式，因此也就需要全局优化。

文章只考虑了包含选择和/或等值连接的合取操作的情况。还给出了一些令人鼓舞的实验结果，并指出了未来的研究方向。

[18.48] Guy M. Lohman: "Grammar-Like Functional Rules for Representing Query Optimization Alternatives," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, Ill. (June 1988).

在某些方面，可以将一个关系的优化器看作是一个专家系统；但是，优化器所使用的这些优化规则都是嵌入在程序中的，没有单独声明。这样就不容易融入一些新的优化技术来扩展优化器的功能。未来的数据库系统（见第26章）将使得这个问题更为严重，因为显然这些系统需要分别安装以扩展优化器来支持各种用户自定义的数据类型。许多研究者提出基于传统的专家系统架构的优化器，并且使用显式声明的优化规则。

但是，这个想法的一个障碍是性能问题。特别地，在优化处理的每个阶段都有大量可用的规则，所以需要复杂的计算来决定使用哪个规则。这篇论文提出一个替代办法（在Starburst原型系统中实现，见参考文献［18.21，26.19，26.23，26.29，26.30］，这个方法利用描述形式语言、类似于语法的产生式规则来声明优化规则。这些产生式规则被称为STAR（STrategy Alternative Rule），它允许从其他查询计划以及"低层计划操作符"（LOLEPOP）递归构造当前的查询计划。这些"低层计划操作符"是基本的关系操作，如连接、排序等。LOLEPOP有许多特性，

例如，连接 LOLEPOP 有排序/归并特性、散列特性等。

文章认为上述的方法有许多好处：STAR 规则很容易理解；在给定情况下决定使用哪个优化规则的过程很简单，而且比传统的专家系统要高效；同时也满足了扩展优化器的目的。

[18.49] Ryohei Nakano："Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS 15*, No. 4 (December 1990).

正如在第 8 章（8.4 节）中说明的一样，使用关系演算表达的查询可以按如下步骤执行：(a) 将该关系演算表达式转换为关系代数表达式；(b) 优化这个关系代数表达式；(c) 最后执行这个优化过的表达式。在文章中，Nakano 提出了一个模式，将步骤 (a)、(b) 合并为一步，即直接将一个关系演算表达式转换为一个优化的关系代数表达式。这个模式被认为是"更为有效的和有希望的……因为优化复杂的代数表达式似乎很困难"。这个转换过程利用了一些启发式规则，这些规则包含了关于等价的关系演算和关系代数的知识。

[18.50] Kyu-Young Whang and Ravi Krishnamurthy："Query Optimization in a Memory-Resident Domain Relational Calculus Database System," *ACM TODS 15*, No. 1 (March 1990).

这篇文章说明了查询处理中代价最大的方面是计算布尔表达式（假设在主存环境中）。所以在这种环境中优化的目标就是减少这类计算。

[18.51] Johann Christoph Freytag and Nathan Goodman："On the Translation of Relational Queries into Iterative Programs," *ACM TODS 14*, No. 1 (March 1989).

这篇文章提出了用 C 或 PASCAL 等语言直接将关系表达式编译为可执行代码的方法。注意这个方法有别于本章讨论的方法，本章的方法是让优化器有效地合并预写的（参数化的）代码片段来构造查询计划。

[18.52] Kiyoshi Ono and Guy M. Lohaman："Measuring the Complexity of Join Enumeration in Query Optimization," Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

因为连接是个两值操作，所以优化器必须将包含 $n$ 个关系（$n>2$）的连接分解为一系列的两值连接。许多优化器按照严格的嵌套方式进行：它们首先选择一个参与连接的关系对，然后将第 3 个关系和前面两者的结果集做连接，由此类推。换句话说，一个形如 *A* JOIN *B* JOIN *C* JOIN *D* 的表达式必须被处理为（（*D* JOIN *B*）JOIN *C*）JOIN *A*，而不能处理为（*A* JOIN *D*）JOIN（*B* JOIN *C*）。而且，传统的优化器都设计为尽可能避免笛卡尔积的运算。这些策略都可以被看作是"减少搜索空间"的办法（当然，选择连接顺序的启发式规则还是需要的）。

这篇文章描述了 IBM Starburst 原型系统的优化器的相关方面（见参考文献 [18.21, 18.48, 26.19, 26.23, 26.29, 26.30]）。文章认为前面的两种策略在某些情况下都是可行的，因此一个可适应的优化器应当针对不同的查询使用不同的策略。

[18.53] Bennet Vance and David Maier："Rapid Bushy Join-Order Optimization with Cartesian Products," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

正如在参考文献 [18.52] 中提到的，优化器通过避免做笛卡尔积来"减少搜索空间"。这篇论文说明搜索整个空间"比先前认为的更为可取"，并且避免做笛卡尔积并不一定有利（有关这一连接的讨论可参见第 22 章的"星型连接"）。根据作者的观点，本文的主要贡献是 (a) 从谓词分析中完全分离了连接顺序列举问题，(b) 为连接顺序列举问题提出了"创造性的实现技术"。

[18.54] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis："Parametric Query Optimization," Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada (August 1992).

考虑如下查询：

```
EMP WHERE SALARY > salary
```

这里的 salary 是个运行时参数。假如在 SALARY 属性上有个索引，那么：

- 如果 salary 是每个月 10 000 美元，那么最好的办法是使用索引来执行这个查询（因为假设绝大部分雇员都达不到这个水平）。
- 如果 salary 是每个月 1000 美元，那么最好的办法是使用顺序扫描来执行这个查询（因为假设绝大部分雇员都达到了这个水平）。

这个例子说明即便在编译系统中，有些优化决策也应当在运行时进行。本文考虑了在编译时产生一组查询计划（每个计划对应某一组参数都是"优化的"），然后在运行时根据参数选择

一个适当的方案。特别地，它侧重于一个查询可用的缓冲区数量这个参数。实验结果表明这个附加的过程稍微增加了一点开销，但对于它带来的优化效果来说是微不足道的。因此，作者认为这个方法可以大大改善查询性能。“使用根据参数值特别定制的查询计划而获得的执行代价方面的收益……是巨大的。”

[18.55] Navin Kabra and David J. Dewitt: "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

[18.56] Jim Gray: "Parallel Database Systems 101," Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1995).

这不是一篇研究论文，而是一个讲座简报的扩充概述。概括地说，并行数据库系统的基本思想是将一个大问题分解为若干个可以同时解决的小问题，从而改进性能（吞吐量和响应时间）。从关系系统的本质来看，关系系统对于并行执行来说尤其有利：它从概念上就容易（a）有许多办法将关系分解为多个子关系，而且（b）可以有多种方法将关系表达式分解为多个子表达式。下面稍微说明一些并行数据库系统的基本概念。

首先，硬件系统要在一定程度上支持并行性。有三种基本的并行架构，各自包括若干处理单元、若干磁盘驱动器以及一个互连网络：

- 共享内存（shared memory）：网络允许所有的处理器存取同一内存。
- 共享磁盘（shared disk）：每个处理器都有自己的内存，但网络允许所有的处理器存取所有的磁盘。
- 无共享（shared nothing）：每个处理器有自己的内存和磁盘，但网络允许处理器之间互相通信。

在实际中，通常选择无共享结构，至少对于大系统是如此（随着处理器的增加，其他两种架构很容易导致相互的干扰问题）。特别地，无共享系统能够提供**线性加速比**（speed-up，提高 $N$ 倍硬件性能导致响应时间 $N$ 倍减少）和**线性伸缩比**（scale-up，提高 $N$ 倍硬件性能及 $N$ 倍数据量能够保证响应时间不变）。注意：“伸缩比”也被称为“**伸缩性**”（scalability）。

有许多种数据分片（data partitioning）的方法（即将关系 $r$ 分解为若干分片或者子关系，并且将这些分片分配给 $n$ 个不同的处理器）：

- 范围分片（range partitioning）：基于关系 $r$ 的某个属性集 $s$，$r$ 被分为若干不相交的分片 1，2，…，$n$。（从概念上讲，$r$ 在 $s$ 上有序，并且将排序结果分为 $n$ 个大小相等的分片。）分片 $i$ 被分配给处理器 $i$。这种方法对于在 $s$ 上包含等值或者范围选择的查询有利。
- 散列分片（hash partitioning）：$r$ 的每个元组 $t$ 被分配给处理器 $i = h(t)$，这里的 $h$ 是某个散列函数。这个方法对于涉及在被散列函数处理过的一个或多个属性上进行等值选择的查询，以及涉及对整个关系 $r$ 进行顺序存取的查询有利。
- 循环分片（round-robin partitioning）：从概念上讲，$r$ 按某种方式排序；在排序结果集中的第 $i$ 个元组被分配给处理器（$i$ MOD $n$）。这个方法对于需要对整个关系 $r$ 进行顺序存取的查询有利。

可以对单个操作实施并行，称之为操作内（intraoperation）并行；也可以对在同一个查询内的多个操作实施并行，称之为操作间（interoperation）或查询内（intra query）并行；还可以对不同查询的执行实施并行，称之为查询间（interquery）并行。参考文献［18.3］有所有这些可能性的讲座，参考文献［18.57，18.58］讨论了一些特定的技术和算法。在实际中，散列连接（hash join，见 18.7 节）是特别有效的，并被广泛应用。

[18.57] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson: "Parallel Algorithms for the Execution of Relational Database Operations," *ACM TODS 8*, No. 3 (September 1983).

这篇文章提出了在多处理器环境中的排序、投影、连接、聚集和更新操作的算法。给出了考虑到 I/O、消息传递以及处理器时间的代价公式，并且能够通过调整适用于不同的多处理器架构。

[18.58] Waqar Hasan and Rajeev Motwani: "Optimization Algorithms for Exploiting the Parallelism Communication Tradeoff in Pipelined Parallelism," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

[18.59] Donald Kossmann and Konrad Stocker: "Iterative Dynamic Programming: A New Class of Optimiza-

tion Algorithms,"*ACM TODS 25*, No. 1 (March 2000).

[18.60] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte: "Exploiting Constraint-Like Data Characterizations in Query Optimization," Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

[18.61] Alin Deutsch, Lucian Poppa, and Val Tannen: "Physical Data Independence, Constraints, and Optimization with Universal Plans," Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland (September 1999).

[18.62] Michael Stillger, Guy Lohman, Volker Markl, and Mohtar Kandil: "LEO-DB2's LEarning Optimizer," Proc. 27th Int. Conf. on Very Large Data Bases, Rome, Italy (September 2001).

# 第19章 信息空缺

## 19.1 引言

在现实世界中，信息是经常有空缺的；比如“不知道生日”、“等待宣布的发言人”、“当前住址不详”等等，对此我们大家都已习以为常。为此，很明显数据库系统里需要有某些方法来处理这样的信息空缺问题。这个问题的解决方法是基于空值（null）和三值逻辑（three - valued logic，3VL），并已很普遍地运用于实际系统中，尤其是SQL及许多相关商业产品中。举个例子，我们可能不知道某些零件比如零件P7的重量，于是可能会很随意地说这个零件的重量“是null”，这句话更准确的意思是：a）我们知道这个零件存在；b）也知道这个零件有重量；c）但我们不知道重量是多少。

如果更进一步考虑如何在数据库中表达零件P7的元组，很显然，我们不能在这个元组上放一个真正的WEIGHT值。于是，替代的做法是在这个元组的WEIGHT位置标上“null”标记，把这个标记的精确意思解释为我们并不知道这个标记的真正值是什么。这里说WEIGHT位置“包含一个null”，或认为WEIGHT值“是null”，是一种不十分严格的说法，尽管在实际中我们经常这样讲。严格地讲，说某些元组的WEIGHT字段值“是null”，其实是说那个元组根本没有包含WEIGHT值。这里不赞成使用“null value”（空的值）这种说法，原因是：精确地讲，null不是值，重复一下，它们是标记或标志。

接下来，我们将在下一节看到任何包括空比较字（comparand）的标量比较式（scalar comparison）的值都等于真值unknown（未知），而不是true（真）或false（假）。这样做的理由是把null解释成“unknown值”：如果不知道$A$的值，那么很明显，$A$是否大于$B$也是unknown，*不管$B$的值是多少*（比较特殊的情况是$B$的值也是unknown）。于是，尤其需要注意两个null之间不能相等，即如果$A$和$B$都是null，则比较式$A = B$的值等于unknown，而不是true（也不能认为它们是不相等，即比较式$A \neq B$的值也是unknown）。因此就有了术语“三值逻辑”（3VL）：空值概念导致这样的逻辑：它包括三个真值，true、false和unknown。

在进一步讨论之前，应当很清楚的是，以我们的观点来看（包括其他很多作者也持这个观点），空值和3VL是非常严重的错误，在像关系模型这样清晰规范的系统中应该没有它们的位置。例如，如果说某个零件元组不包含WEIGHT值[⊖]，实际上也就是说这个有问题的元组终究不是一个零件元组；同样，也就是说有问题的元组并不是一个可应用谓词的实例。实际上，有问题的“元组”根本就不能算是一个元组！（参考第6章中对元组的定义就可以很容易地发现这一点。）事实是，试图准确描述空模式是什么的做法足以说明为什么这一思想在逻辑上不是严格一致的，所以也很难对它有一个一致的解释。用参考文献［11.10］的话来说就是：“如果你稍不注意，不考虑太多，它好像就有意义。”

但是，对空值和3VL不加介绍是不合适的；因此本书保留这部分内容。

本章的内容是这样安排的。引言之后，在19.2节里，先描述空值和3VL的基本思想，但不对这些思想提出过多的批评。（很显然，如果不介绍这些思想是怎么回事，就不可能对它们进行适当和公正地批评。）然后在19.3节里，我们讨论了这些思想所带来的一些重要的后果，以此来证明我们的观点。即空值是一个错误。19.4节讨论了空值在主码和外码上的潜在影响。在19.5节里，讨论了在空值和3VL环境中所遇到的操作，即外连接操作。19.6节非常简短地介绍了信息空缺的一种替代解决方法，即使用特殊值。19.7节略述了SQL的有关空值方面的内容。最后，在19.8节中对本章内容进行了小结。

最后，一个预先要引起注意的问题是：可能有很多原因使我们不能在某些元组的某些位置放置真正的数据值，“未知值”仅仅是其中一个原因。其他原因包括“值不可应用”、“值不存在”、“值

---

⊖ 在本书的其他地方，我们用大写字母表示真值。而在本章中，我们用小写字母表示真值（主要是为了与其他刊物中对相同主题的描述保持一致）。

没有定义”、“值没有提供”等等［19.5］[⊖]。其实，在参考文献［6.2］里，Codd 提出关系模型应当包括两个空值而不是一个空值：一个意味着“未知值”；另一个意味着“值不可应用”。于是他进一步提出 DBMS 应当处理的是四值逻辑，而不是三值逻辑。在［19.5］中我们对这种提法已提出异议；在本章里，我们仅仅讨论单一种类的空值，即“未知值”型空值，以后我们按照定义，经常（但不是所有情况下）把它称为 UNK（对于 unknow 类型）。

## 19.2 3VL 方法概述

在这一节里，简要叙述应用于信息空缺上的 3VL 方法的基本组成部分。下面首先讨论空值（特指 UNK）对布尔表达式的影响。

### 1. 布尔表达式

前面已经说过，只要标量比较式中的两个操作数之一是 UNK，那么这个比较式的值就等于真值 unkown，而不是 true 或 false。接下来讨论三值逻辑（3VL）：Unknown（以后经常（但不是所有情况）把它缩写为 unk）就是“第三个真值”。下面是 AND、OR 和 NOT 在 3VL 上的真值表（t = true，f = false，u = unk）：

| AND | t | u | f |
|---|---|---|---|
| t | t | u | f |
| u | u | u | f |
| f | f | f | f |

| OR | t | u | f |
|---|---|---|---|
| t | t | t | t |
| u | t | u | u |
| f | t | u | f |

| NOT | |
|---|---|
| t | f |
| u | u |
| f | t |

举例，假设 $A=3$，$B=4$，$C$ 是 UNK，那么下面的表达式具有所示的真值：

```
A > B AND B > C   :  false
A > B OR  B > C   :  unk
A < B OR  B < C   :  true
NOT ( A = C )     :  unk
```

然而 AND、OR 和 NOT 并不是所需要的布尔操作符的全部［19.11］；另一个重要的操作符是 MAYBE［19.5］，它的真值表如下：

| MAYBE | |
|---|---|
| t | f |
| u | t |
| f | f |

为了说明为什么需要 MAYBE，现在来考虑这样一个查询：“查出可能是（但不确切知道一定是）在 1971 年 1 月 18 日之前出生的、工资少于 50 000 美元的程序员”。用 MAYBE 操作符，这个查询可以像下面这样简洁地表达出来：[⊖]

```
EMP WHERE MAYBE ( JOB = 'Programmer' AND
                  DOB < DATE ('1971-1-18') AND
                  SALARY < 50000.00 )
```

假定表 EMP 的属性 JOB、DOB 和 SALARY 的类型分别是 CHAR、DATE 和 RATIONAL（有理数）。然而，如果没有 MAYBE 操作符，这个查询可表达如下：

```
EMP WHERE   ( JOB = 'Programmer'
              OR IS_UNK ( JOB ) )
    AND     ( DOB < DATE ('1971-1-18')
```

---

⊖ 需要说明的是，其实并不存在上述所说的各种各样的“空缺信息”。比如，如果我们说佣金对于雇员 Joe“不适用”，可以非常清楚地说佣金属性不应用于 Joe。这里并不存在信息空缺（然而还是这种情况，如果在 Joe 的“雇员元组”的佣金属性中“包含”一个“不适用的 null”，那么这个元组就不是一个雇员元组，即它不是一个“雇员”谓词的实例。）

⊖ 为了举例的需要，我们假设 Tutorial D 支持 UNK 和 3VL，而事实上是不支持的。

```
          OR IS_UNK ( DOB ) )
AND     ( SALARY < 50000.00
          OR IS_UNK ( SALARY ) )
AND NOT ( JOB = 'Programmer' AND
          DOB < DATE ('1971-1-18') AND
          SALARY < 50000.00 )
```

假定存在另一个真值操作符叫 IS_UNK，它是一元操作符，如果操作数的值是 UNK，则返回 true；否则返回 false。（另一方面，非标量比较式也需要一个 IS_UNK 方案。但我们在这儿并不打算定义它，因为其中所包含的复杂性是很令人沮丧的，而且我们并不真的信赖 3VL 方法。）

顺便提一句，上述内容并不表明 MAYBE 是 3VL 需要的唯一新的布尔操作符。其实，比如 TRUE_OR_MAYBE 操作符（如果它的操作数值为 true 或 unk，则返回 true，否则返回 false）也是非常有用的 [19.5]。见本章最后“参考文献及其简介”中 [19.11] 的注释。

**2. 量词**

尽管本书的大部分例子是基于代数而不是演算，但我们还需考虑 EXISTS 和 FORALL 上的 3VL 的潜在影响。如在第 8 章介绍的一样，我们分别把 EXISTS 和 FORALL 定义为迭代的 OR 和 AND。换句话说，如果 a）*r* 是一个关系，它具有元组 *t*1，*t*2，...，*tm*；b）*V* 是在这个关系上的范围变量；c）*p*（*V*）是一个布尔表达式，其中 *V* 是一个自由变量，那么表达式

```
EXISTS V ( p ( V ) )
```

被定义成等价于下列表达式

```
false OR p ( t1 ) OR ... OR p ( tm )
```

同样，表达式

```
FORALL V ( p ( V ) )
```

被定义成等价于如下表达式

```
true AND p ( t1 ) AND ... AND p ( tm )
```

因此，如果对于某些 *i*，*p*（*ti*）等于 unk，将会怎么样？现通过一个例子来说明。假设关系 *r* 正好等于下列元组：

```
( 1,   2,   3   )
( 1,   2,  UNK )
( UNK, UNK, UNK )
```

为了简单起见，假定上述从左到右的三个属性分别叫做 *A*、*B* 和 *C*；而每一个属性都是 INTEGER 类型。于是下面的表达式及其值如下所示：

```
EXISTS V ( V.C > 1 )              :  true
EXISTS V ( V.B > 2 )              :  unk
EXISTS V ( MAYBE ( V.A > 3 ) )    :  true
EXISTS V ( IS_UNK ( V.C ) )       :  true

FORALL V ( V.A > 1 )              :  false
FORALL V ( V.B > 1 )              :  unk
FORALL V ( MAYBE ( V.C > 1 ) )    :  false
```

**3. 其他标量操作符**

考虑下面数字表达式

```
WEIGHT * 454
```

其中 WEIGHT 表示某些零件（比如 P7）的重量。如果零件 P7 的重量是 UNK 会怎么样？这个表达式的值会是什么？答案是它必须为 UNK。通常，对于任何数字表达式，只要其中任何一个操作数本身是 UNK，那么这个数字表达式就等于 UNK。例如，如果 WEIGHT 恰好是 UNK，

那么所有的下列表达式也都等于 UNK：

```
WEIGHT + 454        454 + WEIGHT        + WEIGHT
WEIGHT - 454        454 - WEIGHT        - WEIGHT
WEIGHT * 454        454 * WEIGHT
WEIGHT / 454        454 / WEIGHT
```

注意：这里应当指出，按照上述对数字表达式的处理方式会造成某些异常。比如，表达式 WEIGHT - WEIGHT 应当等于零，但其实等于 UNK。再如表达式 WEIGHT/0 应当会产生一个“除零”错误，但结果也是 UNK（当然首先假定在上述两种情况里的 WEIGHT 是 UNK）。但在未进一步给出说明之前，我们先忽略这些异常。

所有其他标量类型（scalar type）和操作符也会出现类似的问题，但以下情况例外：a）布尔操作符（参见前两小节）；b）之前讨论过的操作符 IS_UNK；c）下面要讨论的操作符 IF_UNK。比如，如果 *A* 是 UNK 或 *B* 是 UNK 或两者都是 UNK，则字符串表达式 *A* ‖ *B* 返回 UNK（同样，这里也会有某些异常情况，在此略去了这些情况的细节。）

IF_UNK 操作符在两个标量表达式操作数上进行操作。如果第一个操作数不等于 UNK，那么它返回第一个操作数的值；否则返回第二个操作数的值（换句话说，这个操作符提供了一种有效的途径，它把一个 UNK 转换为某些非 UNK 值）。比如，供应商的 CITY 属性允许是 UNK。于是表达式

```
EXTEND S ADD IF_UNK ( CITY, 'City unknown' ) AS SCITY
```

产生这样的结果：如果 S 中的供应商的 CITY 属性是 UNK，那么这个供应商的 SCITY 属性的值是 City unknown。

注意，顺便提一句，IF_UNK 可以用 IS_UNK 来定义。清楚起见，表达式

```
IF_UNK ( exp1, exp2 )
```

（这里表达式 *exp*1 和 *exp*2 必须是相同的类型）和下面的表达式等价

```
IF IS_UNK ( exp1 ) THEN exp2 ELSE exp1 END IF
```

#### 4. UNK 不是 unk

理解 UNK（“未知值”的空值）和 unk（unknown 真值）不是同一回事很重要[⊖]。其实，这种情况是基于这样的事实：unk 是一个值（精确地说是一个真值），而 UNK 根本不是一个值。现让我们进一步解释清楚一点。假设 *X* 是一个 BOOLEAN 类型的变量。于是 *X* 一定有 true、false 或 unk 三者中的一个值。这样，语句“*X* 是 unk”精确的意思是已知 *X* 的值是 unk。相对而言，语句“*X* 是 UNK”的意思是 *X* 的值不知道。

#### 5. 一个类型可以包含一个 UNK 吗

UNK 不属于任何一个类型（类型是值的集合），因为事实上 UNK 不是值。其实，如果一个类型可以包含一个 UNK，那么对这个类型的类型约束检查就从来都不会失败！然而，既然类型事实上不能包含 UNK，那么一个包含 UNK（不管它是什么）的“关系”其实根本不是一个关系，不管根据在第 6 章给出的定义还是 Codd 在参考文献［6.1］中给出的原始定义来判断都如此。我们稍后会讲述这个重要的观点。[⊜]

#### 6. 关系操作符

现在把注意力转向 UNK 对关系代数操作符的影响。为了简单起见，我们只讨论乘积、选择、投影、并和差（UNK 对其他操作符的影响可以依据 UNK 在这五个操作符上的影响来确定）。

首先，乘积操作不受影响。

---

⊖ 然而 SQL 认为它们是同一回事（见 19.7 节）。

⊜ 本段原文都将 type 翻译成域，可能有误，本段中将 type 翻译成类型。——译者注

第二，重新定义（稍微改动）**选择**操作，以返回只包含在选择条件上为 true 的元组的关系，即选择条件对这些元组的值不是 false，也不是 unk。注意：前面我们已在“布尔表达式”小节中的 MAYBE 例子里隐含地假定了这个重定义。

接下来是**投影**。投影操作当然包括对重复元组的删除。在传统的两值逻辑（2VL）中，两个元组 *t*1 和 *t*2 完全相同，当且仅当两者实际上是相同的元组——也就是说，当且仅当它们有相同的属性 *A*1，*A*2，…，*An*，对所有的 *i*（*i* = 1，2，…，*n*），元组 *t*1 中 *Ai* 的值等于元组 *t*2 中 *Ai* 的值。然而，在 3VL 中某些属性的值可能是 UNK，而 UNK（我们已经知道）和任何东西都不相等，甚至和它自己也不相等。那么我们是否要作出结论，一个包含 UNK 的元组永远不与任何元组相等，甚至与它本身也不相等？

按照 Codd 的说法，这个问题的答案是否定的：两个 UNK，甚至它们互不相等，也被认为其中一个是另一个的副本，这样做的目的是为了删除重复元组［14.7］。[○] 下面是为这样明显的矛盾进行的辩护：

［等同性判定］删除重复元组所进行的等同性判定，与检索条件中进行的等同性判定相比，在内容的考虑上不那么细致。因此，可以采用不同的规则。

这个基本结论是否有理留给大家去判断。不过现在让我们先接受它，于是有下面的定义：

- 两个元组 *t*1、*t*2 是相同的当且仅当它们有相同的属性 *A*1，*A*2，…，*An*，对所有的 *i*（*i* = 1，2，…，*n*），要么元组 *t*1 中 *Ai* 的值等于元组 *t*2 中 *Ai* 的值，要么元组 *t*1 中 *Ai* 的值与元组 *t*2 中 *Ai* 的值均为 UNK。

有了这扩展的“相同元组”定义，原先的投影定义可不加修改便可适用。但是注意，下面两个等式是等价的：

- *t*1 = *t*2
- *t*1 与 *t*2 是重复的

并同样要删除多余的重复元组，相同元组定义同样适用于这里。于是，我们定义关系 *r*1 和 *r*2（具有相同的类型）并成关系 *r*（同样具有相同类型），关系 *r* 包含所有这样的元组 *t*：即 *t* 是 *r*1 的某个元组的副本或 *r*2 的某个元组的副本或是两者中某个元组的共同副本。

最后，*差*也类似地重新加以定义，尽管它不包括任何副本删除。即一个元组 *t* 出现在 *r*1 MINUS *r*2 里，当且仅当它是 *r*1 中某些元组的副本而不是 *r*2 中元组的副本（至于*交*，虽然并不允许但是为了完善我们的认识也可以做同样的重新定义：即一个元组出现在 *r*1 INTERSECT *r*2 里，当且仅当它同时是 *r*1 的某个元组和 *r*2 的某个元组的副本）。

**7. 更新操作**

对此操作有两个一般性的问题值得注意：

1）如果关系 *R* 的属性 *A* 允许用 UNK，并且如果通过 INSERT 在 *R* 中插入一个在属性 *A* 上没提供值的元组，则系统会自动在这个元组的属性 *A* 位置上放置 UNK。（当然，在这两种情况里都没有定义 *A* 的默认值是非 UNK。）如果关系 *R* 的属性 *A* 不允许 UNK，则试图通过 INSERT 或 UPDATE 把在 *A* 位置上是 UNK 的元组插入 *R* 里是错误的。

2）和往常一样，试图通过 INSERT 或 UPDATE 在 *R* 里创建一个相同元组是错误的。这里“相同元组”的定义和上面是一样的。

**8. 完整性约束**

正像第 9 章所解释的，一个完整性约束本质上是一个结果不等于 false 的布尔表达式。因此，如果一个约束的值等于 unk，则不认为它是违反约束的（其实，在本节先前部分关于类型约束已经隐含地论述了许多这方面的内容）。从技术上讲，在这种情况下我们应当说它是否违反约束是未知的，但是，就像在 WHERE 子句中把 unk 当作 false 一样，为了完整性约束我们把 unk 认为是

---

○ ［14.7］是 Codd 的第一篇讨论信息空缺问题的论文（虽然这个问题不是这篇论文的主要论点，见第 14 章）。在其他方面，这篇论文提出了 θ-join、θ-select 和除操作符（见练习 19.4）的 maybe 方案，以及并、交、差、θ-join 和自然连接操作符（见 19.5 节）的“out”方案。

true（比较不严格的说法）。

## 19.3 上述方案所造成的某些结果

上一节所论述的3VL方法会产生一些逻辑结果，但并不是所有的结果都很明显。我们在这一节里讨论其中一些结果以及它们的重要性。

**1. 表达式转换**

首先，我们注意到，在2VL中的值是true的一些表达式在3VL中就不一定总是true了。这里举几个例子，并加以说明。请注意，所举的例子是不可能穷尽所有情况的。

- 比较式 $x = x$ 不会必然得出true
  在2VL里，任何值 $x$ 总是和它自己相等，但在3VL里，$x$ 如果是UNK，那么它就不等于它自己。
- 布尔表达式 $p$ OR NOT（$p$）不会必然得出true。

在2VL里，不管 $p$ 为什么布尔表达式，表达式 $p$ OR NOT（$p$）必然会等于true。但在3VL里，如果 $p$ 的值是 *unk*，则整个表达式的值是 *unk* OR NOT（*unk*），即unk OR *unk*，简化为 *unk*，而不是true。这个特殊的例子说明了3VL有反直觉的特性，这个特性举例说明如下：如果我们发出“求出在伦敦的所有供应商”的查询，接着发出“求出不在伦敦的所有供应商”的查询，然后对这两个结果做并，则我们不会必然获得所有的供应商。这其中可能漏掉了“所有可能在伦敦供应商”的结果（也就是说，在3VL方法中表达式 $p$ OR NOT（$p$）OR MAYBE（$p$）恒真，与2VL方法中的表达式 $p$ OR NOT（$p$）类似）。

有必要对之前的例子做进一步的验证。这个例子的要点，当然是“城市是伦敦”和“城市不是伦敦”这两种情况在现实世界中是互斥的、并穷尽现实世界的所有可能，而数据库并不包含现实世界——其实，它仅仅包含它对现实世界的认知。关于对现实世界的认知存在三种情况，而不是两种；在这个例子里这三种情况是“被知道是伦敦的城市”、“被知道不是伦敦的城市”和“不被知道的城市”。当然（如［19.6］所说的），我们显然不能问系统关于现实世界的问题，而只能问系统它所知道的用数据库中数据表达出来的关于现实世界的问题。这样，对范畴的混淆产生了这个例子的反直觉特性：用户是根据现实世界这个范畴来思考的，而系统是根据它对现实世界的认知来操作的。（但对本文作者来讲，这样的范畴混淆是一个非常容易掉进去的陷阱。注意，在这本书前面章节中的每一个单一查询（在示例、练习中）是按照“现实世界”而不是按照“对现实世界的认知”表达的。当然这本书在这方面也不会特殊。）

- 表达式 $r$ JOIN $r$ 不会必然给出 $r$
  在2VL里，一个关系 $r$ 和它自己进行连接总能得出原先的关系 $r$（即连接是幂等的）。然而在3VL里，一个任何一个位置是UNK的元组不能和它自己做连接，因为（根据参考文献［14.7］）不像联合，连接是基于“检索类型”（retrieval-type）的相等性判定，而不是“重复类型”的相等性判定。
- 交不再是连接的特殊情况。
  这个事实同样来自于连接是基于检索类型的相等性判定而交是基于重复类型的相等性判定所造成的结果。
- $A = B$ 和 $B = C$ 两个式子不能推出 $A = C$。
  关于这个论点的进一步说明在下面的“部门和雇员的示例”小节里给出。

总之，许多在2VL中有效的恒等式在3VL中不再有效。这种情况造成一个如下的严重后果。通常，简单的恒等式如 $r$ JOIN $r \equiv r$ 是各种各样变换规则的基础，而这些规则用来把查询转化成某些更有效的形式，如第18章所介绍的。甚至不仅系统使用这些规则（当做优化的时候），用户也使用这些规则（当试图决定“更好”地描述某个查询的时候）。如果恒等式不再有效，那么规则也会失效。如果规则失效，那么变换也不再有效。如果变换无效，那么我们将会从系统那里得到错误的答案。

**2. 部门和雇员的例子**

为了阐明不正确变换问题，我们来稍微详细地讨论一个特殊的例子（这个例子来自参考文

献［19.9］；这里使用了关系演算而不是关系代数，但这并不是很重要）。假设给定一个简单的数据库，包含部门和雇员，见图19-1。考虑下面的表达式

```
DEPT.DEPT# = EMP.DEPT# AND EMP.DEPT# = DEPT# ('D1')
```

（当然这可能是一个查询的一部分）；这里 DEPT 和 EMP 暗指范围变量。对于数据库里单一的元组，这个表达式等于 *unk* AND *unk*，即 *unk*。然而，一个“好的”优化器会注意到这个表达式形式是 $a = b$ AND $b = c$，于是它会推出 $a = c$，并会在原先的表达式后边加上限制条件 $a = c$（如在第18章18.4节里讨论的），这样得到

| DEPT | DEPT# |
|---|---|
| | D2 |

| EMP | EMP# | DEPT# |
|---|---|---|
| | E1 | UNK |

图19-1　部门－雇员数据库

```
DEPT.DEPT# = EMP.DEPT# AND EMP.DEPT# = DEPT# ('D1')
                       AND DEPT.DEPT# = DEPT# ('D1')
```

这个修改过的表达式等于 *unk* AND *unk* AND *false*（对于数据库仅有的两个元组来说）。于是，下面的查询（举例）

```
  EMP.EMP# WHERE EXISTS DEPT ( NOT
( DEPT.DEPT# = EMP.DEPT# AND EMP.DEPT# = DEPT# ('D1') ) )
```

如果按上述的理解进行“优化”，将返回雇员 E1。换句话说，这样的“优化”其实是不正确的。这样我们就知道了某些在2VL中是正确且有用的优化在3VL中不再正确。

*注意*那些为了把2VL系统扩展到支持3VL系统而造成的潜在影响。最好的情况是这样的扩展很可能需要对现有的系统进行重设计；最坏的情况是这样会导致错误。更普遍的是，要注意那些把支持 $n$ 值逻辑的系统扩展到支持（$n+1$）值逻辑的系统的潜在影响，这里 $n$ 是任何一个大于1的数；对于每一个离散值 $n$ 都会产生类似难题。

**3. 解释方法**

现在，更进一步地来研究部门－雇员这个例子。既然雇员 E1 在现实世界里没有某个对应的部门，那么让 UNK 代表某个真实值，比如是 *d*。现在 *d* 或是 D1 或不是。如果它是，那么原先的表达式

```
DEPT.DEPT# = EMP.DEPT# AND EMP.DEPT# = DEPT# ('D1')
```

等于（对于上述特定的值来说）false，因为第一项等于 false。另一方面，如果 *d* 不是 D1，则这个表达式还是等于（对于上述特定的值来说）false，因为第二项等于 false。换句话说，原先的表达式在现实世界里总是等于 false，不管 UNK 代表什么真实值。这样，在3VL里是正确的结果和在现实世界是正确的结果不是同样的事情！换句话说，三值逻辑的方法和现实世界是不一致的；即，3VL看起来没有一种符合现实世界本身规律的解释方法。

*注意*：这个解释问题远远不是由空值和3VL所造成的仅有的一个问题（关于其他问题的进一步讨论请见［19.1～19.11］）。它甚至不是最基本的（见下面讨论）。然而，它也许是其中最具有实际意义的一个；其实，以作者观点来看，这只是一幕精彩的表演。

**4. 再论谓词**

假定关系 EMP 仅仅含有两个元组，（*E*2，*D*2）和（*E*1，UNK）。第一个对应于这样的叙述“有一个标识为 *E*2、且在标识为 *D*2 的部门里的雇员”。第二个对应于这样的叙述“有一个标识为 *E*1 的雇员”（回忆一下，说一个元组“含有一个 UNK”其实真正地是说这个元组在这个可应用的位置上根本没有任何东西；这样，元组（*E*1，UNK）——如果它是元组的话，本质上这是一个可能有问题的概念——应当被认为仅仅是（*E*1）的形式）。换句话说，这两个元组是两个不同谓词的事例，且这个“关系”根本不是一个关系，而是（不精确地说）两个有不同标题的不同关系的并。

也许有人会提出上述情况可以通过一个谓词来解决，这个谓词含有一个 OR。可能情况如下：

有一个标识为 E#、且在标识为 D#的部门里的雇员 OR 有一个标识为 E#的雇员。

然而，幸亏封闭世界假设（the Closed World Assumption），使得这个关系中所有的雇员 E*i* 都必须包含一个（E*i*，UNK）形式的元组！如果使这种拯救意图普遍化到几个“属性”都“含有 UNK”的“关系”上，那几乎太可怕了（在任何情况下，有这样结果的“关系”将仍然不是一个关系——见下一段）。

用另一种方法来考虑上述问题：如果一个给定关系的一个给定元组的一个给定属性的值是“UNK”，于是（重复一下）这个属性其实根本没有包含任何东西……于是可推出这个“属性”不是一个属性，这个“元组”不是一个元组，这个“关系”不是一个关系，且我们正在做的（不论它是其他什么东西）基础不再是数学上的关系理论。换句话说，UNK 和 3VL 破坏了关系模型的整个基础。

## 19.4 空值和码

*注意：我们现在放下术语 UNK（在大部分段落中），回到更传统的术语“空值”（null）上，这是由于有历史原因。*

除了前面几节的内容涉及 UNK，事实是现在大部分产品都支持空值和 3VL。这样的支持对于码来说尤其有重要的潜在影响。因此在这一节里，将简要地研究一下这些潜在的影响。

### 1. 主码

正如在第 9.10 节里所介绍的，关系模型在历史上要求（至少对于基变量是如此的）选出一个候选码来作为关系的主码。剩下的候选码（如果有的话），被称为可选码（alternate key）。于是，除了主码概念外，关系模型在历史上曾包括下面的“元约束”（metaconstraint）或规则（实体完整性规则）：

- **实体完整性**：基变量的主码的任何部分都不能为空值。

这个规则的理论基础来自于这样的解释：a）基本关系中的元组表示现实世界的实体；b）现实世界中的实体是用定义标识的；c）因此这些实体在数据库中的副本也必须被标识；d）在数据库中主码的值是被用来作这些标识的；e）因此主码值一定不能“空缺”。这样就产生了下述要点：

1）首先，人们经常会认为实体完整性规则就是指“主码值必须是唯一的”，但其实并不这样。（当然主码值必须唯一是正确的，但这个要求本身蕴涵在主码的基本定义里。）

2）其次，注意这条规则仅适用于主码；可选码显然允许有空值。但如果 AK（可选码）是一个允许有空值的可选码，由于实体完整性规则，则不能选择 AK 做为主码，这样一来，首先在哪种意义上 AK 还是一个“候选”码？另一种角度看，如果我们不得不说可选码也不能有空值，则完整性规则适用于所有的候选码，而不仅仅是主码。从两种角度看，所述规则似乎存在一些错误。

3）最后，注意实体完整性规则仅适用于基本关系变量；其他关系变量显然可以有允许是空值的主码。举一个简单的例子，考虑关系变量 *R* 在任一允许有空值的属性 *A* 上做投影。显然这规则违背了交换性原则（关于基本关系变量和导出关系变量）。我们认为，这是拒绝这个规则的根本理由，即使它不涉及空值，也要拒绝这个规则。

现在，假设采用丢弃空值的观点，取而代之用特殊值[○] 来表示空缺的信息（其实正像在现实世界里所做的——稍后见 19.6 节）。那么也许可以保留一个被修改过的实体完整性规则作为指导法则：“任何基变量的主码的任何部分都不准接受这样的特殊值”，而不是作为一个不可违背的法则（如许多更标准化的思想都作为指导，而不是作为不可违背的法则）。图 19-2

| SURVEY | BIRTHYEAR | AVGSAL | MAXSAL | MINSAL |
|---|---|---|---|---|
| | 1960 | 85K | 130K | 33K |
| | 1961 | 82K | 125K | 32K |
| | 1962 | 77K | 99K | 32K |
| | 1963 | 78K | 97K | 35K |
| | .... | ... | ... | ... |
| | 1970 | 29K | 35K | 12K |
| | ???? | 56K | 117K | 20K |

图 19-2 基本关系变量 SURVEY（样本值）

---

○ 经常被不恰当地称作默认值 [19.12]。

给了一个关于基本关系变量叫 SURVEY 的示例可能会违背一些指导思想；它表示一个工资调查的结果，用来显示某一人群样本按照出生年份的平均、最大和最小工资（BIRTHYEAR 是主码）。BIRTHYEAR 值为“????”的元组表示填表时没有回答“你什么时候出生?”这一问题。

2. 外码

再次考虑图 19-1 的部门和雇员数据库。也许大家没有注意到，图中关系变量 EMP 的属性 DEPT#是一个外码。因此很显然参照完整性规则需要一些改进，因为现在外码很明显必须能够接受空值，而空值外码明显然违反了原先在第 9 章所述的规则：○

- **参照完整性**（最初形式）：数据库不能包含任何不匹配的外码值。

其实，只要适当地扩展术语“不匹配的外码值”的定义，就可以保持所述的规则。为了明确起见，我们定义在某些参照关系变量中一个不匹配的外码值是一个非空的外码值，而相关的被参照的关系变量中不存在这个外码值的相关候选码的匹配值。由此产生了下述要点：

1）必须把是否允许任一给定外码接受空值详细说明为数据库定义的一部分（当然，其实通常这也适用于属性，不管它们是否是某个外码的一部分）。

2）外码可以接受空值的可能性会导致另一个参照动作的可能性，这个动作是 SET NULL，它或许会在一个外码的 DELETE 或 UPDATE 规则中详细说明。比如：

```
VAR SP BASE RELATION { ... } ...
    FOREIGN KEY { S# } REFERENCES S
                      ON DELETE SET NULL
                      ON UPDATE SET NULL ;
```

有了这样的说明，一个在供应商关系变量上的 DELETE 操作会把所有相匹配的供货中的外码设置为空值，然后删除相应的供应商；同样，一个在供应商关系变量的属性 S#上的 UPDATE 操作会把所有相匹配的供货中的外码设置为空值，然后删除相应的供应商。注意：当然仅仅对首先能接受空值的外码，SET NULL 才可以被说明。

3）最后，注意到适当的数据库设计可以避免在外码中有空值［19.19］。比如，再一次考虑部门和雇员。如果真的有可能不知道某些雇员的部门号，那么（如上节接近结尾所述的）更好的方法是根本不要在 EMP 关系变量里包括 DEPT#，而要一个单独的关系变量 ED（比方说），这个关系变量的属性是 EMP#和 DEPT#，表示一个指定的雇员在一个指定的部门这样的事实。于是，某个雇员有一个不知道的部门的事实可以用从关系变量 ED 中删除这个雇员元组来表示。

## 19.5　外连接

在这一节里，我们将简要讨论一下外连接操作（见［19.3，19.4］、［19.7］和［19.14～19.16］）。外连接是常规连接或内连接操作的一种扩展形式。它不同于内连接的是，若一个关系中的元组在另一个关系中没有相匹配的元组，则这些元组会在结果中出现，并在另一个关系的其他属性位置放上空值，而不是像通常那样被忽略。它不是一个基本操作。比如，下述表达式被用来构造供应商和供货关系在供应商号码上的外连接。（在这里假设“NULL”是一个合法的标量表达式）：

```
( S JOIN SP )
  UNION
( EXTEND ( ( S { S# } MINUS SP { S# } ) JOIN S )
  ADD ( NULL AS P#, NULL AS QTY ) )
```

这个表达式的结果包括没有供应零件的供应商元组，并扩展地在 P#和 QTY 位置上放置空值。

更仔细一点来研究这个示例。参考图 19-3。在这个图中，上面部分表示一些样本值，中间部分表示相应的内连接，而下面部分表示相应的外连接。如这个图所示，内连接在没有供应零件的供应商那里（在本例中是供应商 S5）“空缺了信息”（不精确地讲），然而外连接“保留”了

○　即使在相关的参照关系变量（其相关的候选码为空）中有这样的元组，空值也是违反这一规则。

这样的信息。其实，这个区别就是外连接的要点所在。

S

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S2 | Jones | 10 | Paris |
| S5 | Adams | 30 | Athens |

SP

| S# | P# | QTY |
|---|---|---|
| S2 | P1 | 300 |
| S2 | P2 | 400 |

*Regular (inner) join:*

| S# | SNAME | STATUS | CITY | P# | QTY |
|---|---|---|---|---|---|
| S2 | Jones | 10 | Paris | P1 | 300 |
| S2 | Jones | 10 | Paris | P2 | 400 |

"Loses" information for supplier S5

*Outer join:*

| S# | SNAME | STATUS | CITY | P# | QTY |
|---|---|---|---|---|---|
| S2 | Jones | 10 | Paris | P1 | 300 |
| S2 | Jones | 10 | Paris | P2 | 400 |
| S5 | Adams | 30 | Athens | UNK | UNK |

"Preserves" information fo supplier S5

图 19-3 内连接和外连接（示例）

外连接要解决的问题（即内连接有时会“空缺信息”的事实）当然是一个很重要的问题。于是某些作者提出系统应当提供直接的、清楚的外连接支持，而不是要求用户通过非常复杂的陈述来获得想要的结果。尤其 Codd 认为外连接是关系模型中的固有部分（见参考文献［6.2］）。然而，我们不认可这种观点，因为：

1）首先，这个操作包含空值，而我们有许多好理由反对空值。

2）第二，外连接有好多种形式——左、右和完全外 θ－连接，以及左、右和完全外自然连接。（“左”连接保留了来自第一个操作数的信息，“右”连接保留了来自第二个操作数的信息，而“完全”连接则两者兼有；图 19-3 的例子是一个左连接——精确地说是左外自然连接。）进一步讲，这里不存在直接的方法从外 θ 连接导出外自然连接［19.7］。因此，难以确定究竟哪一种外连接需要明确的支持。

3）其次，图 19-3 的例子远远不能把外连接问题叙述完整。其实正如参考文献［19.7］所述的，外连接具有许多有害的特点，这些特点加起来使在现有语言（尤其是 SQL）加上外连接是困难的。一些 DBMS 试图去解决这个问题，但最终以失败告终（即它们被这些有害的特点所困扰）。关于这个论点更详细的论述请见参考文献［19.7］。

4）最后，可以说关系值属性（relation-valued attribute）提供了解决这个问题的一个方法，一个不包含空值也不包含外连接的方法，并且从作者的观点来看，该方法是一个更完美的解决方法。（这里忽略了一个事实，它是一种关系的方法是不违背关系模型的。）比如给定如图 19-3 上面部分的样本数据值，那么下面的表达式

```
WITH ( S RENAME S# AS X ) AS Y :
     ( EXTEND Y ADD ( SP WHERE S# = X ) AS PQ ) RENAME X AS S#
```

会产生如图 19-4 所示的结果。

在图 19-4 中，尤其要注意供应商 S5 所提供的零件空集是用一个空集表示出来的，而不是（像图 19-3）使用一些古怪的“null”。用一个空集表示一个空集不失为一个好主意。其实，如果适当地支持关系值属性，那么就根本没有必要使用外连接。

| S# | SNAME | STATUS | CITY | PQ |
|---|---|---|---|---|
| S2 | Jones | 10 | Paris | P# QTY: P1 300; P2 400 |
| S5 | Adams | 30 | Athens | P# QTY: |

图 19-4 保留供应商 S5 的信息（一种更好的方法）

再来讨论一下这个论点：怎样来解释出

现在外连接结果中的空值？比如它们在图 19-3 的例子中意味着什么？它们当然既不意味着“unknown 值”，也不意味着“值不可应用”；其实，能给出任何有逻辑性的唯一精确的解释是“值是空集”。有关进一步论述请见参考文献［19.7］。

在结束本节内容之前，我们要指出的是，关系代数的其他操作也可以定义类似“外”连接的形式（尤其是并、交和差操作［14.7］）并且 Codd 认为至少其中一个即外并应是关系模型的一部分（见参考文献［6.2］）。这样的操作允许并（及其他操作）在两个甚至是不同类型的关系上进行操作；它们使每个操作数都包括对它来说是古怪的属性（这样，这两个操作数现在就成为相同的类型了），并在这些增加的属性上放置空值，然后变成普通的并，交或差[⊖]。然而，由于下述的原因我们不打算对这些操作进行详细的论述：

- 外交操作肯定会返回一个空的关系，除非原来的关系具有相同的类型，若是这样，就退化为普通的交了。
- 外差操作肯定回返回它的第一个操作数，除非原来的关系具有相同的类型，若是这样，就退化为普通的差了。
- 外并操作的主要的问题是关于解释的问题（它们比外连接所造成的问题更糟糕）。关于进一步论述请见参考文献［19.2］。

## 19.6　特殊值

我们已经看到，空值破坏了关系模型。但值得一提的是，没有空值的关系模型已经很好地使用了十年！——模型在 1969 年第一次被定义［6.1］，空值直到 1979 年才被加进来［14.7］。

因此，假设（像 19.4 节所提议的）我们赞成丢弃空值的整个思想，取而代之使用特殊值来表示空缺的信息。注意，使用特殊值正是我们在实际工作中所做的。比如，在实际工作里，如果因某些原因不知道某个雇员的工作时间，可能会用“?”来表示这个值[⊜]。这样，当没有实际值可用的时候，通常的方法是可以简单地使用一个与这个属性的所有实际值不同的特殊值。注意，特殊值必须是一个来自可应用的域；因此在“工作时间”这个例子里，属性 HOURS_WORKED 的类型不仅仅是整型，而应当是整型加上特殊值。（这里有一个比较好的类推：对于多数纸牌游戏来说，TRUMPS 类型包含五个值，而不是四个，分别是——“hearts”、“clubs”、“diamonds”、“spades”以及“no trumps”。

这里首先得承认上述方案也不是完美的，但它的最大优点在于它不会破坏关系模型的逻辑基础。因此，在本书的其余部分，我们将完全忽略对空值的支持（除非在有关 SQL 的讨论中，会涉及空值的问题）。关于特殊值方案的详细论述请见参考文献［19.12］。

## 19.7　SQL 的支持

SQL 对空值和 3VL 的支持遵循前面章节里所述方法的主要内容。这样，比如当 SQL 在某个表 T 上应用 WHERE 子句时，它会删除所有对于这个 WHERE 子句中的表达式求出的值是 false 和 unk（即非 true）的元组。同样地，当 SQL 在某个“分组表”（grouped table）G 上应用一个 HAVING 子句时，它会删除所有对于这个 HAVING 子句中的表达式求出的值是 false 和 unk（即非 true）的 G 的分组[⊝]。因此，在接下来的部分中，我们将仅限于讨论对 SQL 本身来说是特殊的某些 3VL 特点，而不是像先前所述的 3VL 方法的本质部分。

注意：SQL 关于空值支持的整个蕴涵和延伸是很复杂的。事实上，虽然我们刚刚说过在很大范围内 SQL 遵循 3VL，但是我们很快也会看到 SQL 在支持这个逻辑的时候将会造成许多的错误。有关更多的信息，请参见正式标准文档——参考文献［4.23］或［4.20］中详细的辅导教程。

---

⊖ 这个说明引用了原来定义的操作［14.7］；定义做了某些更改，见［6.2］。

⊜ 我们没有为此用空值，在实际工作中也不会用空值［19.12］。

⊝ 分组表是当执行了 GROUP BY（可能是隐式的）时产生的。当伴有执行 SELECT 时，这些表还原成非分组表。

### 1. 数据类型

正如我们在第4章所看到的那样，SQL包括内置的BOOLEAN类型（它是在1999年加入到标准中的，虽然现在几乎没有什么产品支持这个类型）。常见的布尔操作符如AND，OR以及NOT仍是有用的，而布尔表达式可以出现在普通标量表达式出现的任何地方。但是我们知道，现在有三个真值而不是两个（对应的文字分别是TRUE，FALSE和UNKNOWN）。尽管BOOLEAN类型只包括两个值，而不是三个——unknown真值用空值来表示也是不正确的。它将导致：

- 将UNKNOWN的值赋给BOOLEAN类型的变量 ***B*** 事实上是对这个变量赋空值。
- 在这样的赋值之后，比较式 ***B*** = UNKNOWN 的值就不是true（或者TRUE），而是空值。
- 事实上，不管 ***B*** 是什么值，比较式 ***B*** = UNKNOWN 始终是空值——因为在逻辑上这个比较式等价于比较式 ***B*** = NULL（但并不意味着这是一个非法的语法）。

为了理解这个缺陷，必须要注意区分在数字类型中用空值而不是用零来表示零。

设 *T* 为一个非标量类型或一个结构类型（这里将结构类型考虑为标量类型或非标量类型并没有什么区别）。为证明我们的观点，设 *T* 为一个特殊的行类型，*V* 是类型 *T* 的一个变量，那么在（a）*V* 自身是空值；（b）*V* 至少有一组成部分（例如域）是空值之间有一个很明显的逻辑区别。事实上，即使 *V* 的所有组成部分均为空值，*V* 本身的值也不一定为空值[⊖]。——如果 *V* 值为空那么它的所有组成部分值均为空，这点可能是正确的（虽然在这个项上定义的标准是模糊的）。因此，如果 *V* 非空但是至少有一个组成部分的值是空值，比较式 *V* = *V* 值为空，但是表达式 *V* IS NULL 值为FALSE。通常，我们可以说如果（（*V* = *V*）IS NOT TRUE）IS TRUE 值为TRUE，那么要么 *V* 值为空，要么 *V* 有一个组成部分为空。

### 2. 基表

如第6.6节所述的，基表中的列通常有一个相应的默认值，而且这个默认值经常被显式或隐式地定义为是空值。甚至基表中的列经常允许取空值，除非存在一个完整性约束（可能仅是NOT NULL）这个约束明白地禁止这些列不允许为空值。

紧接着前面所说的，如果我们要证明我们的原理为真，那么我们就必须将在本书中之前所提到的每个SQL例子中出现的所有基表的列显式或隐式的定义为非空。至少从现在开始要对即将出现的SQL例子做这样的定义。但是，请注意主码（PRIMARY KEY）规范中所提到的任何列都被隐式的定义为非空。

### 3. 表表达式

回忆一下8.6节，1992年在SQL标准里给SQL增加了显式连接支持。如果在关键字JOIN的前面加上LEFT、RIGHT或FULL（每种情况后面OUTER词可有可无），这样的连接就是一个外连接。下面有几个例子：

```
S LEFT JOIN SP ON S.S# = SP.S#

S LEFT JOIN SP USING ( S# )

S LEFT NATURAL JOIN SP
```

这三个表达式实际上都是相等的，除了第一个得出一个有两个相同列（都是S#）的表，第二个和第三个得出一个只有一列的表。

SQL也支持一个外并的近似方法，称为并连接。（SQL：1992中加入了该方法，SQL：2003又将其删除了。）关于它的详细论述已超出本书范围。

### 4. 布尔表达式

毫无疑问，SQL中的布尔表达式受空值和3VL的影响很大。我们在这里作几个重要的说明：

- 关于空值的计算：SQL提供两个特殊的比较操作，IS NUL和IS NOT NULL，来计算空值的存在或不存在。语法如下：

```
<row value constructor> IS [ NOT ] NULL
```

---

⊖ 实际上，SQL在这一点上也是错误的——见本节后面“布尔表达式”部分关于IS [NOT] NULL的讨论。

如果 < row value constructor > 构建了一维的一行元组，那么 SQL 将认为这个表达式表达的是包含在这一行内的值而不是这一行；否则 SQL 将认为这个表达式表示的是这一行。在稍后的例子中，SQL 将这行认为是：(a) 空值当且仅当每个组成部分均为空；(b) 非空当且仅当每个组成部分均为非空！这个错误造成的一个后果就是如果 *r* 是带有两个组成部分 *c*1 和 *c*2 的一行元组，那么两个表达式 *r* IS NOT NULL 和 NOT (*r* IS NULL) 是不相等的。前一个等于 *c*1 IS NOT NULL AND *c*2 IS NOT NULL，而后一个等于 *c*1 IS NOT NULL OR *c*2 IS NOT NULL。另一个后果就是如果 *r* 包含一些空和非空的组成部分，那么 *r* 很明显既不是空也不是非空。

- 关于 true、false 和 unknown 的计算：如果 *p* 是括号中的布尔表达式（虽然括号有时候并不是必需的，但是加上括号也不会出错），于是下述也是布尔表达式：

```
p IS [ NOT ] TRUE
p IS [ NOT ] FALSE
p IS [ NOT ] UNKNOWN
```

这些表达式的含义可用下述真值表来表述：

| *p* | *true* | *false* | *unk* |
|---|---|---|---|
| *p* IS TRUE | *true* | *false* | *false* |
| *p* IS NOT TRUE | *false* | *true* | *true* |
| *p* IS FALSE | *false* | *true* | *false* |
| *p* IS NOT FALSE | *true* | *false* | *true* |
| *p* IS UNKNOWN | *false* | *false* | *true* |
| *p* IS NOT UNKNOWN | *true* | *true* | *false* |

观察后可知，表达式 *p* IS NOT TRUE 和 NOT *p* 是不相等的。注意：表达式 *p* IS UNKNOWN 对应于 MAYBE (*p*)。如果 SQL 中使用空值来表示 unk，那么它也等价于 *p* IS NULL。

- EXISTS 条件：SQL 中 EXISTS 操作符同 3VL 方法中的存在量词并不相同，因为它的值总为 true 或 false，不会为 unk，即使 unk 是逻辑上的准确值。具体的说，如果它的参数表为空，那么它将返回 false，否则返回 true（因此，虽然 unk 是逻辑上的正确答案，但是有时候返回值却为 true)。具体的细节请见参考文献 [19.6]。
- UNIQUE 条件：UNIQUE 条件用于检测某个表中是否包含重复的行。更详细点就是，如果一个表（用 < table exp > 表示）包含两行 *r*1 和 *r*2，它们完全相同，表达式 UNIQUE ( < table exp > ) 返回 true，也就是说比较式 *r*1 = *r*2 返回 true，否则返回 false。因此，就像 EXISTS 一样，UNIQUE 在 unk 是逻辑上的正确答案时有时候返回 true。
- DISTICT 条件：DISTICT 条件用于检测两个行是否相同。假设两个行分别表示为 Left 和 Right；Left 和 Right 必须有相同的维度，设为 *n*。假设 Left 和 Right 的第 *i* 个部分为 *Li* 和 *Ri* (*i* = 1, 2, …, *n*)；*Li* 和 *Ri* 必须使得比较式 *Li* = *Ri* 合理，那么表达式

```
Left IS DISTINCT FROM Right
```

对于所有的 *i*，如果：a) "*Li* = *Ri*" 为 true，或 b) *Li* 和 *Ri* 均为空，则返回 false，否则返回 true。

**5. 其他标量表达式**

我们再来看一些重要的特殊例子：

- "文字" (literal)：关键字 NULL 可以被用作空值的文字表示（如在 INSERT 语句里)。然而注意，这个关键字并不能在所有文字可以出现的环境里出现；如标准所述的，"不存在空值的 < literal >，虽然在某些地方使用关键字 NULL 来表示这里需要一个空值" [4.23]。这样，就不能清楚地指明 NULL 作为一个简单比较式的操作数（例如，"WHERE X = NULL"）是非法的（正确的形式是 "WHERE X IS NULL")。
- COALESCE（合并)：SQL 中的 COALESCE 类似于前面提到的 IF_UNK 操作符。更具体些就是，如果 *x*, *y*, …, *z* 均为空，那么表达式 COALESCE (*x*, *y*, …, *z*) 返回空；否则它返回第一个非空操作符的值。

- 聚集操作符：SQL 聚集操作符（SUM，AVG 等）的计算和 19.2 节所述的标量操作符的规则并不一致，它们在计算中忽略了参数中任何的空值。（除了 COUNT（*）以外，它把空值看成是一般值。）同样，如果这样的操作符的参数碰巧是空集，则 COUNT 会返回零；其他操作符都返回空值。（如第 8 章所述，后者在逻辑上是错误的，但 SQL 就是如此定义的。）
- "标量子查询"（Scalar subqueries）：如果一个标量表达式是一个在括号里的表达式——比如，（SELECT S.CITY FROM S WHERE S.S# = S# ('S1')）——一般地讲，这个表达式会求得一个单列单行的表。于是这个标量表达式的值其实精确地来说是一个只包含一个标量值的表。但是，如果这个表达式求得的值是一个根本没有行的单列表，则 SQL 把这个标量表达式的值定义为空值。

6. 码

SQL 中空值和码的相互关系概括如下：

- 候选码：假设 *C* 是某个基本表的某个候选码 *K* 的一个组成列。如果 *K* 是主码，则 SQL 不会允许 *C* 包含任何空值（换句话说，它应当遵守实体完整性规则）。然而，若 *K* 不是主码，则 SQL 会允许 *C* 包含任何数目的空值（当然也允许包含任何数目的非空值）。

  为了很好地将前面所讲的内容联系起来，可以从参考文献［4.20］了解到："假设 *k2* 是 *K* 的新值，有些用户企图使用操作 INSERT 或 UPDATE 操作向 *K* 中增加此值……如果 *k2* 与 *K* 的某个值 *k1*（在同一个表中原有的值）相同，那么将拒绝执行 INSERT 或 UPDATE 操作……那么 *k1* 与 *k2* 的值是否相同呢？我们可以得出一个结论就是下面三个句子没有两个是等价的：

  1）如果比较 *k1* 与 *k2* 的值，那么它们是相同的。

  2）如果 *k1* 与 *k2* 是作为码唯一性的候选项，那么它们是相同的。

  3）如果 *k1* 与 *k2* 是用于消除重复值，那么它们是相同的。

  第一个语句符合 3VL 规则；第二个语句与 UNIQUE 条件中的规则也是相符的；第三个语句与 19.2 节中重复的定义是相符的。特别的是当 *k1* 与 *k2* 均为空值时，第一条语句返回 unk，第二条返回 false，第三条返回 true。"
- 外码：在空值情况下如何确定外码与相应主码值的关系，规则是相当复杂的，有关细节这里不再详述。注意：空值对于参照行为（CASCADE、SET NULL 等）也有潜在影响，在 ON DELETE 和 ON UPDATE 子句中有详细说明（SET DEFAULT 也以明显的解释得到支持）。同样，这些细节也是相当复杂的，并且超出了本书范围：关于细节请参见［4.20］。

7. 嵌入式 SQL

- 指示变量：考虑下面的一个嵌入式 SQL "singleton SELECT" 示例（第 4 章的示例）：

```
EXEC SQL SELECT STATUS, CITY
         INTO   :RANK, :TOWN
         FROM   S
         WHERE  S# = S# ( :GIVENS# ) ;
```

  假设有可能某些供应商的 STATUS 的值是空值。如果被选中的 STATUS 是空值，则上面的 SELECT 语句将会失败（SQLSTATE 将会被置为异常值 22002）。通常，如果存在被选取的值是空值的可能，则用户应当给目标变量指定一个指示变量，如下面所示：

```
EXEC SQL SELECT STATUS, CITY
         INTO   :RANK INDICATOR :RANKIND, :TOWN
         FROM   S
         WHERE  S# = S# ( :GIVENS# ) ;
IF RANKIND = -1 THEN /* STATUS was null */ ... ; END IF ;
```

  如果被选取的值是空值且指示变量已被指定，则这个指示变量被置为值 -1。对于通常的目标变量所造成的影响依执行而定。
- 排序：ORDER BY 子句在游标定义（cursor definition）中被用来对表达式结果里的行进行排序。（当然，在交互式的查询中也会用到。）这样产生了下述问题：如果标量值 *A* 或 *B*

是空值（或都是），则 *A* 和 *B* 的相对次序是什么？SQL 关于这个问题的回答如下：

1）对于排序，所有的空值被认为是互相相等的。

2）对于排序，所有的空值被认为大于所有的非空值或小于所有的非空值（到底应用哪种情况视实现而定）。

## 19.8　小结

我们已经讨论了关于**信息空缺**的问题和目前流行的（虽然很不好）一种基于**空值**和**三值逻辑**（3VL）的关于这个问题的解决方法。我们强调这样的观点：空值不是一个值，虽然通常好像它是一个值。（比方说，某个特殊元组的某个特殊属性值是“空值”。）有一个操作数是空值的任何比较式的值等于“第三个真值”unknown（缩写为 unk），因此有了三值逻辑。同时提到，至少在概念上，存在许多不同种类的空值，并把 UNK 作为“未知值”这个类型的一个简写。

然后我们研究了 UNK 和 3VL 在布尔表达式 **AND**、**OR**、**NOT**（还有 **MAYBE**）；量词 **EXISTS** 和 **FORALL**；**计算**表达式；**关系**操作符；**更新**操作符 INSERT 和 UPDATE 上的潜在影响。介绍了操作符 **IS_UNK**（用来测试 UNK）、**IF_UNK**（用来把 UNK 转化为非 UNK 值）。讨论了关于 UNK 的**等同**问题，并指出了 UNK 和 unk 不是同一回事。

接下来，我们研究了上述思想所造成的一些后果。首先，说明了**某些恒等式**在 3VL 中**不再有效**——即在 2VL 中有效的等价式在 3VL 中不再有效。因此，用户和优化器可能**在表达式转换中产生错误**。而且，即使这样的错误不会发生，3VL 也会因它**和现实世界不符**这个严重问题而带来麻烦——即对于 3VL 是正确的结果某些时候在现实世界中是不正确的。

于是我们继续讨论了在**主码**和**外码**上的空值的潜在影响（特别提到了**实体完整性**规则和修改过的**参照完整性**规则）。接下来介绍了**外连接**。我们自己不提倡直接支持这个操作（至少不像通常理解的那样），因为我们相信有更好的方法来解决外连接要解决的问题——特别地，我们推荐一种使用**关系值属性**的解决方法。简要地提到了其他“外”操作的可能性，特别是**外并**。

接下来，我们研究了上述思想的 **SQL 支持**。SQL 对空缺信息的处理主要是基于 3VL，但它设法包括了大量附加的复杂东西，大部分都超过了本书的范围。其实，SQL 是在设法引入一些**附加的缺点**，这些缺点是 3VL 本身造成的［19.6，19.10］。更糟的是，这些附加的缺点成了**优化的抑制剂**（在第 18 章结尾处提到过）。

最后小结如下：

- 大家会注意到，我们仅仅涉及由空值和 3VL 所造成的表面的问题。但我们已经尽力用足够多的理由来说明从 3VL 方法中得到的“好处”是值得怀疑的。
- 我们也使大家明白，即使你不相信关于 3VL 本身的问题，但仍建议大家应避开它在 SQL 上所造成的相关东西，否则会有上面所述的“附加的缺点”。
- 我们给 DBMS 用户的建议是：完全忽略厂家的 3VL 的支持，使用一个遵守规则的“特殊值”方案（从而牢牢地坚持二值逻辑）。这个方案在参考文献［19.12］里有详细论述。
- 最后，我们重复下述来自 19.3 节的基本要点：不精确地讲，如果一个给定关系的给定元组的给定属性的值“是空值”，则这个属性的位置其实根本没有包含任何东西……这就推导出这个“属性”不是一个属性，这个“元组”不是一个元组，这个“关系”不是一个关系，且我们正在做的（不管它会是什么）基础不再是数学上的关系理论。

## 习题

19.1　若 A = 6，B = 5，C = 4，且 D 是 UNK，请说出下列表达式的真值：

```
a. A = B OR ( B > C AND A > D )
b. A > B AND ( B < C OR IS_UNK ( A - D ) )
c. A < C OR B < C OR NOT ( A = C )
d. B < D OR B = D OR B > D
e. MAYBE ( A > B AND B > C )
```

f. MAYBE ( IS_UNK ( D ) )

g. MAYBE ( IS_UNK ( A + B ) )

h. IF_UNK ( D, A ) > B AND IF_UNK ( C, D ) < B

19.2 假设关系 *r* 正好包含下列元组：

```
(  6,   5,   4  )
( UNK,  5,   4  )
(  6,  UNK,  4  )
( UNK, UNK,  4  )
( UNK, UNK, UNK )
```

像本章内容所述的，假设 a）这三个属性以上面所示的顺序从左到右依次叫作 *A*、*B* 和 *C*，且 b）每一个属性都是 INTEGER 类型。若 *V* 是 *r* 上的范围变量，请说出下列表达式的真值：

a. EXISTS *V* ( *V.B* > 5 )

b. EXISTS *V* ( *V.B* > 2 AND *V.C* > 5 )

c. EXISTS *V* ( MAYBE ( *V.C* > 3 ) )

d. EXISTS *V* ( MAYBE ( IS_UNK ( *V.C* ) ) )

e. FORALL *V* ( *V.A* > 1 )

f. FORALL *V* ( *V.B* > 1 OR IS_UNK ( *V.B* ) )

g. FORALL *V* ( MAYBE ( *V.A* > *V.B* ) )

19.3 严格地说 IS_UNK 操作符是多余的。为什么？

19.4 在参考文献［14.7］里，Codd 提出了一些（不是所有）关系代数操作符的“maybe”方案。比如，maybe-restrict 和通常的 restrict 不同，它返回这样的一个关系，这个关系包含在选择条件上求出的值是 unk 而不是 true 的元组。然而严格地讲这样的操作符是多余的。为什么？

19.5 在二值逻辑（2VL）里，正好存在两个真值，true 和 false。结果正好有 4 个可能的单元逻辑操作符——一个把 true 和 false 都变换为 true，一个把两者都变换为 false，一个把 true 变换为 false，一个把 false 变为 true（当然这个操作符是 NOT），还有一个是使两者都不会变化。正好存在 16 个可能的二元操作符，如下表所示：

| *A* | *B* | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| t | t | t | t | t | t | t | t | t | f | f | f | f | f | f | f |
| t | f | t | t | t | f | f | f | f | t | t | t | t | f | f | f |
| t | t | t | f | f | t | t | f | f | t | t | f | f | t | t | f |
| t | f | t | f | t | f | t | f | t | f | t | f | t | f | t | f |

证明，2VL 中的所有 4 个单元操作符和 16 个二元操作符可以用 NOT、AND 和 OR 的适当组合来表示（因此就没有必要明确地支持所有 20 个操作符）。

19.6 在 3VL 里有多少逻辑操作符？在 4VL 里是多少？推广到更一般的情形，在 *n*VL 里会是多少？

19.7 2VL 操作符 NOR（通常也用一个竖线“|”表示）的真值表如下：

| | t | f |
|---|---|---|
| t | f | f |
| f | f | t |

从真值表可以看出 *p* | *q* 等价于 NOT *p* AND NOT *q*（可以理解为“既不……也不，既不是第一个操作数也不是第二个操作数时为 true”）。所有 20 个 2VL 操作符都可以按照这个操作符的形式来表达。注意：NOR 在整个 2VL 中是一个“产生”操作符。你能在 3VL 中找出功能与它类似的操作符吗？在 4VL、*n*VL 中呢？

19.8 （来自参考文献［19.5］）图 19-5 表示常用的供应商和零件数据库里的一些样本值，但稍微有点变化。（这个变化是：关系 SP 包含一个新的供货号属性 SHIP#，且这个关系中的属性 P#现在“允许为 UNK”；关系 *P* 和这个练习没有关系，故这里被省略）。考虑下面的关系演算查询（这里 S 和 SP 是隐含的范围变量）。下面哪一个（若有）是这个查询的正确解释？

a）取出不供应 *P2* 的供应商。

b）取出不知道供应 *P2* 的供应商。

c）取出已知道不供应 *P2* 的供应商。

d）取出已知道不供应 *P2* 或不知道供应 *P2* 的供应商。

S

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |

SP

| SHIP# | S# | P# | QTY |
|---|---|---|---|
| SHIP1 | S1 | P1 | 300 |
| SHIP2 | S2 | P2 | 200 |
| SHIP3 | S3 | UNK | 400 |

图 19-5　供应商和零件数据库示例

19.9　为 SQL 基本表设计一个物理表示方案，这些基本表允许包含空值。

19.10　定义 SQL EXIST、UNIQUE、IS DISTINCT FROM 操作符。这些操作符是起始操作符吗，能用其他操作符表示吗？存在 IS NOT DISTINCT FROM 操作符吗？请给出一个查询的例子，这个查询里包括 a）EXISTS b）UNIQUE，产生“错误”的结果。

## 参考文献

[19.1]　E. F. Codd and C. J. Date: "Much Ado About Nothing," in C. J. Date, *Relational Database Writngs 1991 - 1994*. Reading, Mass.: Addison-Wesley (1995).

Codd 也许是为了处理空缺信息而把空值和 3VL 作为基础的最初提倡者。这篇文章包含了 Codd 和本书作者在这个问题上所进行的辩论。其中包含下列妙语：“如果不存在空缺值，数据库管理将会变得容易”（Codd）。

[19.2]　Hugh Darwen: "Into the Unknown," in C. J. Date, *Relational Database Writings 1985 - 1989*. Reading, Mass.: Addison-Wesley (1990).

提出了一些关于空值和 3VL 的另外的问题，下述也许是其中最值得研究的问题：如果（如 6.4 节所述）TABLE_DEE 相应于 true 而 TABLE_DUM 相应于 false，且 TABLE_DEE 和 TABLE_DUM 是唯一可能的零级（degree zero）关系，那么什么相应于 unk?

[19.3]　Hugh Darwen: "Outer Join with No Nulls and Fewer Tears," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

提出“外连接”的一个简单变种，这个变种不涉及空值，并解决了许多外连接应该解决的问题。也可以参见参考文献 [3.3]。

[19.4]　C. J. Date: "The Outer Join," in *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

深度分析了外连接问题，并说明了 SQL 是如何实现此操作的。

[19.5]　C. J. Date: "NOT Is Not(Not)! (Notes on Three-Valued Logic and Related Matters)," in *Relational Database Writings 1985 - 1989*. Reading, Mass: Addison-Wesley (1990).

假设 X 是一个 BOOLEAN 类型的变量。那么 X 必须是 true、false 或 unk 中的一个。这样，语句“X is not true”意味着 X 的值是 unk 或 false。相反，语句“X is NOT true”意味着 X 的值是 false（见 NOT 的真值表）。3VL 里的 NOT 不是自然语言里的 not……这个事实已经使有些人（包括 SQL 标准的设计者）感到困惑，并无疑还会继续下去。

[19.6]　C. J. Date: "EXISTS Is Not 'Exists'! (Some Logical Flaws in SQL)," in *Relational Database Writings 1985 - 1989*. Reading, Mass.: Addison-Wesley (1990).

[19.7]　C. J. Date: "Watch Out for Outer Join," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

本章第 19.5 节提到这样的事实，外连接有一些“令人讨厌的特点”。这篇论文把这些特点总结如下：

1）外 θ 连接不是笛卡尔积的一个选择。

2）外 θ 连接上没有选择。

3）“$A \leqslant B$”和“$A < B$ OR $A = B$”不是同一回事（在外连接环境里）。

4）θ 比较操作符不可传递（在 3VL 里）。

5）外自然连接不是外等值连接的一个投影。

这篇论文继续考虑为 SQL SELECT-FROM-WHERE 结构增加外连接将会怎么样。它叙述了上述令人讨厌的特点会推导出下列情况：

1）WHERE 子句不能进行扩展操作。

2）外连接不能进行 AND 操作和选择操作。

3）不能在 WHERE 子句里表达连接条件。

4）多于两个关系的外连接若没有嵌套表达式将不能被明确地表达出来。

5）SELECT 子句（单独）不能进行扩展操作。

这篇论文也说明了许多现有产品与这些因素的冲突。

[19.8] C. J. Date: "Composite Foreign Keys and Nulls," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

这篇论文讨论了这个问题：允许复合外码整个或部分是空值吗？

[19.9] C. J. Date: "Three-Valued Logic and the Real World," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

[19.10] C. J. Date: "Oh No Not Nulls Again," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989 - 1991*. Reading, Mass.: Addison-Wesley (1992).

这篇论文提供了很多的大家也许想知道的关于空值的东西。

[19.11] C. J. Date: "A Note on the Logical Operators of SQL," in *Relational Database Writings 1991 - 1994*. Reading, Mass.: Addison-Wesley (1995).

由于 3VL 有三个真值 true、false 和 unk（这里分别缩写为 t、f 和 u），所以存在 3 * 3 * 3 = 27 个可能的单元操作符，因为每个可能的输入 t、f 和 u 可以映像为三个可能输出 t、f 和 u 中的一个。因此有 $3^9$ = 19 683 个可能的二元 3VL 操作符，如下面的表格所示：

| | *t* | *u* | *f* |
|---|---|---|---|
| *t* | t/u/f | t/u/f | t/u/f |
| *u* | t/u/f | t/u/f | t/u/f |
| *f* | t/u/f | t/u/f | t/u/f |

其实更一般的情况是，*n* 值逻辑包括 *n* 的 *n* 次方幂个单元操作符和 *n* 的 $n^2$ 次方幂个二元操作符：

| | *Monadic operators* | *Dyadic operators* |
|---|---|---|
| 2VL | 4 | 16 |
| 3VL | 27 | 19,683 |
| 4VL | 256 | 4,294,967,296 |
| ... | ..... | ..... |
| *n*VL | (*n*)**(*n*) | (*n*)**($n^2$) |

于是，对任何 *n*VL（*n* > 2），会产生下述问题：

- 一个合适的原语操作符集合是什么？（比如，集合 {NOT, AND} 或 {NOT, OR} 是 2VL 的一个适合的原语集合。）
- 一个有用的操作符集合是什么？（比如，集合 {NOT, AND, OR} 是 2VL 的一个有用集合。）

参考文献 [19.11] 说明了 SQL 标准（在一个很宽松的解释方式下）至少直接或间接地支持所有 19 710 个 3VL 操作符。

[19.12] C. J. Date: "Faults and Defaults" (in five parts), in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994 - 1997*. Reading, Mass.: Addison-Wesley (1998).

叙述一个解决空缺信息问题的系统方法，这个方法是基于特殊值和 2VL 而不是空值和 3VL。这篇论文有力地辩明特殊值是我们在现实世界所使用的东西，即在现实世界里不存在空值这样的事物。于是数据库系统在这个方面应当像现实世界处理问题一样来处理问题。

[19.13] Debabrata Dey and Sumit Sarkar: "A Probabilistic Relational Model and Algebra," *ACM TODS 21*, No. 3 (September 1996).

提出一个基于概率理论而不是空值和 3VL 的、用来解决"数据值不确定性"问题的方法。

"或然性关系模型"是传统关系模型的一个可兼容的扩展。

[19.14] César A. Galindo-Legaria："Outerjoins as Disjunctions,"Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, Minn. (May 1994).

一般来说，外连接不是一个关联的操作符［19.4］。这篇论文准确地描述了外连接的特点，这些外连接有的是关联的，有的不是关联的，并为每种情况提出了实现方案。

[19.15] César Galindo-Legaria and Arnon Rosenthal："Outerjoin Simplification and Reordering for Query Optimization,"*ACM TODS 22*, No. 1 (March 1997).

介绍了包含外连接的表达式的"一个完整的变换规则集合"。

[19.16] Piyush Goel and Bala Iyer："SQL Query Optimization: Reordering for a General Class of Queries," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

像［19.15］一样，这篇论文讨论了包含外连接的表达式的转换："［我们］提出了一个方法，用来重新组织［一个］包含外连接、和……聚集的 SQL 查询……［我们］标识了一个强有力的［在这样的重组织中起辅助作用］的原语，［我们］把这个原语称为通用化查询。

[19.17] I. J. Heath: IBM internal memo (April 1971).

这篇论文介绍了"外连接"术语（和概念）。

[19.18] Ken-Chih Liu and Rajshekhar Sunderraman："Indefinite and Maybe Information in Relational Databases,"*ACM TODS 15*, No. 1 (March 1990).

包含一组正式的建议，这些建议被用来扩展关系模型以便处理 maybe 信息（比如，"零件 P7 可能是黑色"）和不确定性的或离散的信息（比如，"零件 P8 或零件 P9 是红色的"）。介绍了 I 表，用来表示正常（确定的）信息、maybe 信息和不确定性信息。扩展了选择、投影、乘积、并、交和差操作符，以便对 I 表进行操作。

[19.19] David McGoveran："Nothing from Nothing"(in four parts), in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994 - 1997*. Reading, Mass.: Addison-Wesley (1998).

这篇论文由四部分构成。第一部分叙述了在数据库中逻辑的重要作用。第二部分明确地表述了为什么必须是二值逻辑，为什么试图使用三值逻辑（3VL）是被误导的。第三部分研究了三值逻辑（3VL）应当要"解决"的问题。最后，第四部分介绍了一组对这些问题注重实效的解决方案，这些方案不包含 3VL。

[19.20] Nicholas Rescher: *Many-Valued Logic*. New York, N. Y.: McGraw-Hill (1969).

标准文本。

# 第20章 类型继承

## 20.1 引言

注意：本章将在很大程度上依赖第5章的讨论。如果你之前只是简略地阅读第5章，那么你在开始本章的学习之前，需要返回到第5章仔细阅读它的内容。

在第14章介绍了子类型和超类型的概念，更确切地说是实体子类型和实体超类型。在该章中指出，假设某些雇员是程序员，而所有的程序员都是雇员，那么可以认为实体类型 PROGRAMMER（程序员）是实体类型 EMPLOYEE（雇员）的子类型，而 EMPLOYEE 是 PROGRAMMER 的超类型。同时也指出，一个“实体类型”并不是一个非常规范意义上的“类型”（部分原因是由于“实体”本身没有正式的定义）。在本章中，我们将深入地分析子类型和超类型。而对于“类型”，将使用第5章中给出的比较规范和准确的定义。因此，先从更准确的“类型”定义开始：

- 一个类型是一组值的命名集合（即被讨论类型的所有可能的取值），以及与之相关的可以在属于该类型的值或变量上应用的操作的集合。

进一步地讲：

- 任何给定的类型可以是系统定义的，也可以是用户定义的。
- 任何给定类型的定义是对该类型中所有有效值构成的集合的规格说明（规格说明就是在第5章和第9章所说的有效类型约束）。
- 这些值可以具有任意的复杂度。
- 这些值的实际表示形式或物理表示形式对用户通常是不可见的，即类型与其（实际）表示形式是不同的。但是每种类型至少要通过合适的THE_操作（或是其他等价的逻辑操作）明确地给用户提供一种合适的表示形式。
- 任何给定类型的值或由该类型定义的变量只能通过定义在该类型上的操作进行操作。
- 除了已经提到的 THE_操作，类型的操作还包括：
  - 至少要有一个选择子操作（更确切地说，对应于每种可能的外在表示形式都应该有一个这样的操作），通过调用合适的选择子操作，可以实现对类型中的每个值进行选取和引用。
  - 一个相等操作，可以检查相同类型的两个值是否相等。
  - 一个赋值操作，可以将一个值赋给同类型的一个变量。

在上述的基础上，现在指出：

- 某些类型是另外一些**超类型**的**子类型**。如果 *B* 是 *A* 的子类型，则所有适用于 *A* 的操作和类型约束都适用于 *B*（继承），但同时 *B* 拥有属于自己的、并不适用于 *A* 的操作和类型约束。

举个例子，假设有 ELLIPSE（椭圆）和 CIRCLE（圆）两种类型。顾名思义，可以说 CIRCLE 是 ELLIPSE 的子类型（ELLIPSE 是 CIRCLE 的超类型）。这实际上是说：

- 每个圆同时也是一个椭圆（即所有圆的集合是所有椭圆的集合的一个子集），但是反之并不成立。
- 因此，普遍适用于椭圆的操作也相应地适用于圆（因为圆属于椭圆），但是反之并不成立。比如，操作 THE_CTR（取中心）可以适用于椭圆，从而也适用于圆，但是操作 THE_R（取半径）则只适用于圆。
- 此外，普遍适用于椭圆的约束也相应地适用于圆（同样是由于圆属于椭圆），但是反之并不成立。比如，椭圆遵循约束 $a \geqslant b$（$a$ 和 $b$ 分别是椭圆的长半轴和短半轴），则圆必定也满足这一约束。当然对于圆而言，$a$、$b$ 就是半径 $r$，但约束还是得到了满足。事实上确切

地说，约束 $a = b$ 只适用于圆而并不能普遍适用于椭圆。注意：在本章里，我们所说的“约束”一词是指一种类型约束，所说的“半径”和“半轴”实际上是指相应的半径长度和半轴长度。

小结一下：简单地说，类型 CIRCLE 继承了类型 ELLIPSE 的操作和约束，同时还有属于自己的操作和约束，但是这些操作和约束并不适用于类型 ELLIPSE。这样一种情况往往容易引起概念上的混淆，即子类型同时具有超类型的值的一个子集和属性的一个扩展集。注意：在整个这一章里，我们使用“属性”这个词来作为“操作和约束”的简称。

**1. 讨论类型继承的目的**

为什么类型继承问题值得探讨呢？这至少有两点原因：

- 第一，子类型与继承的概念在现实世界中是自然存在的，而且这种情形还是会经常碰到的。一个给定类型的所有值具有某些共同的属性，而这些值的某些子集具有更多属于它们自己的特定的属性。这样，子类型与继承看起来是“为现实建模”（modeling reality）（或者像在第 14 章所说的，是“语义建模”（semantic modeling）的一个有用的工具）。
- 第二，如果能够识别出这些模式，即子类型和继承的模式，并能建立起识别方法，把这些模式纳入到应用软件和系统软件中，我们也许可以获得某些实在的好处。比如，一个可以应用于椭圆的程序或许也可以应用于圆，哪怕在编写程序的时候根本就没有考虑过关于圆的问题（也许在编写程序时类型 CIRCLE 还没有定义），这种好处就是**代码重用**。

尽管有上面所说的这些潜在的好处，但是迄今似乎还没有就一个规范的、严格的、抽象的类型继承模型达成任何一致意见。引用参考文献［20.13］的话说就是：

> 继承的基本概念非常简单……（而且，尽管）它在现有系统中处于中心地位，继承仍然是一种有争议的机制……还是没有（一个）全面的关于继承的概念。

本章所做的讨论是基于作者与 Hugh Darwen 共同建立的模型，该模型在参考文献［3.3］[⊖] 中有详尽的描述。因此必须明确，其他作者和其他文章在使用诸如“子类型”和“继承”等概念的时候，其使用角度可能与我们所讨论的角度并不一样。

**2. 预备知识**

在正式讨论继承之前，先要澄清一些基本概念，这些概念是本小节的主题。

- 值是有类型的

  在第 5 章中曾说过，如果 $v$ 是一个值，则可以认为 $v$ 带有某种标志来表明“我是一个整数”、“我是一个供应商号”或“我是一个圆”，等等。如果不存在继承，那么一个值只属于一种类型。但是如果存在继承，那么一个值就可以同时属于多个类型。例如，一个给定值可以同时属于类型 ELLIPSE 和 CIRCLE。

- 变量是有类型的

  每个变量都有一个声明的类型，比如可以声明如下一个变量：

  ```
  VAR E ELLIPSE ;
  ```

  这里变量 $E$ 的类型声明为 ELLIPSE。如果不存在继承，一个给定变量所能取的值就只属于一个类型，即该变量的声明类型。但是，如果存在继承，一个变量所具有的值就可能同时属于多个类型。比如，变量 $E$ 的当前值可能是一个圆（同时也是一个椭圆），因此这个值就同时属于类型 ELLIPSE 和 CIRCLE。

- 单一继承与多重继承

  类型继承主要有两种情况：单一继承与多重继承。简单地说，**单一继承**是指每个子类型只有一个超类型，从而只继承一种类型的属性；**多重继承**是指一个子类型可以有多个超类型，并继承了所有这些类型的属性。显而易见，前一种情况是后一种情况的特例。可是即

---

⊖ 文献中解释得比较清楚，我们并不希望读者将我们的模型仅仅看作是另一个学术练习。但是出于更方便交流的考虑我们提供这个模型来填补这些间接的不足——也就是说，作为继承模型的候选者应该可以提供“继承全面视图”的缺失。

便是单一继承就已经很复杂了（虽然这有些让人吃惊，但事实上确实如此），因此在本章中，我们只把注意力放在单一继承上，我们所说的继承都是特指单一继承。关于单一继承和多重继承的详细讨论请见参考文献［3.3］。

- 标量、元组和关系继承
  很明显继承既包括标量值也包括非标量值，因为那些非标量值最终是由标量值[⊖]构成的。当然特别地，继承还包括针对元组值和关系值的继承。但是，仅是标量继承就已经相当复杂了，因此在本章中我们只把注意力集中在标量继承上，而我们所说的类型、值以及变量实际上都是指标量类型、标量值和标量变量。在参考文献［3.3］中有关于各种继承的详细讨论，包括标量继承、元组继承和关系继承。
- 结构继承与行为继承
  标量值可以拥有一个具有任意复杂度的内部（物理）结构或表现形式。例如，椭圆和圆在适当的情形下（这种情形我们已经知道），都可以很自然地被看做标量值，即使它们的内部结构可能非常复杂。但是，其内部结构对于用户通常是不可见的。从而当谈到继承时（至少是对于模型而言），其中并不包括结构的继承，因为从用户的角度来看并没有结构需要继承。换句话说，我们感兴趣的是所谓的行为继承，而不是结构继承（这里的“行为”是指操作，虽然约束也可以继承，至少在模型中是这样）。注意：当然并不排除结构继承，只是把它看做一个实现上的问题，从而与我们的模型无关。
- “子表与父表”
  现在应该清楚了，继承模型所关注的是关系术语中所说的域继承（再次提醒，域和类型是相同的东西）。但是在关系范畴里谈到继承的可能性时，大部分人立刻会想到是在讨论某一类的表继承。例如，SQL 标准支持所谓的“子表与父表”，据此标准，*B* 可以在继承表 *A* 的所有列之后再加上一些自己的列（见第 26 章）。而在我们看来，“子表与父表”的概念是一个完全独立的现象，虽然它同时也可能是很有意义的（尽管我们在参考文献［14.13］中对此表示了怀疑），但是它在本质上与类型继承毫无关系。

最后一个要说明的基本概念是：类型继承是一个与所有普遍意义上的数据都有关的问题，而不是仅仅局限于数据库中的数据。因此，为了简单起见，本章的大部分例子是以局部数据（普通程序变量，等等）的形式表达的，而不是数据库中的数据。

## 20.2　类型的层次结构

先给出一个在本章中要用到的例子。这个例子包括一组几何类型——PLANE_FIGURE（平面图形）、ELLIPSE、CIRCLE、POLYGON（多边形），等等，这些类型组织成一个所谓的**类型层次**（type hierarchy），或者更一般地讲，一个类型图（type graph）（见图 20-1）。这里列出的是 **Tutorial D** 中对于其中一些几何类型的定义（请特别注意其中的类型约束）：

```
TYPE PLANE_FIGURE ... ;

TYPE ELLIPSE
     IS PLANE_FIGURE
        POSSREP { A LENGTH, B LENGTH, CTR POINT
                  CONSTRAINT A ≥ B } ;

TYPE CIRCLE
     IS ELLIPSE
        CONSTRAINT THE_A ( ELLIPSE ) = THE_B ( ELLIPSE )
        POSSREP { R   = THE_A ( ELLIPSE ) ,
                  CTR = THE_CTR ( ELLIPSE ) } ;
```

让我们来仔细看看这些定义。首先，为了简单起见，我们假设椭圆总是正向的，也就是说它

---

⊖ 回想一下可知，标量意味着没有用户可见的组成部分。不要被那些可能轮流拥有用户可见组成部分的标量类型所误导，像第 5 章所说的那样；那些组成部分都是可能表达式的组成部分而不是类型的组成部分——虽然有时候我们提到它们时会将它们看做类型的组成部分。

的长轴 $a$ 总是水平的而短轴 $b$ 总是垂直的。同时长半轴 $a$ 总是大于或等于短半轴 $b$（也就是说椭圆是“矮而胖”的而不是“高而瘦”的）。这样，椭圆就可以用它的半轴 $a$ 和 $b$（以及中心）来表示。相对的，圆可以用半径 $r$（和圆心）来表示。

之后，我们观察到：

- 类型 PLANE_FIGURE 可能不代表任何东西；事实上，随后我们将看到 PLANE_FIGURE 实际上是一个合并类型（union type）。（更准确地说，它是合并类型中比较特殊的一个类型，称为虚拟类型。对于虚拟类型的讨论超出本章的讨论范围，更详细的解释请见参考文献 [3.3]。）
- 对于 ELLIPSE 类型来说，我们指定每个椭圆都是一个平面图形，用 $\{a, b, ctr\}$ 来表示，并带有一个约束 $a \geqslant b$。注意：为了满足完整性，我们在 $b$ 上添加一个约束，要求 $b$ 大于 0。为简便起见，这里忽略这个约束。
- 对于 CIRCLE 类型来说，我们指定每个圆都是一个椭圆，并定义一个约束 $a = b$，在这个约束下我们可以用 $\{r, ctr\}$ 来表示，这里也说明此表示与椭圆的表示有什么联系。说明超类型名在这样一个约束的上下文中是如何来表示超类型的任意一个值的。

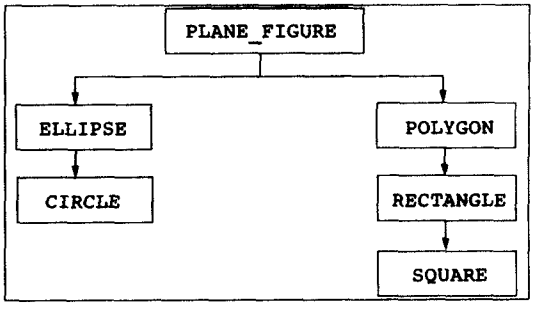



图 20-1　一个简单的类型层次结构

这样系统就会知道 CIRCLE 是 ELLIPSE 的子类型，因此，普遍适用于椭圆的操作和约束也就相应适用于圆。下面列出的是与上述类型有关的一些操作的定义：

```
OPERATOR AREA ( E ELLIPSE ) RETURNS AREA ;
   /* “面积”——注意 AREA 既是操作本身的名字 */
   /* 又是结果的类型的名字 */ ... ;
END OPERATOR ;

OPERATOR THE_A ( E ELLIPSE ) RETURNS LENGTH ;
   /* “半轴 a 的长度” */ ... ;
END OPERATOR ;

OPERATOR THE_B ( E ELLIPSE ) RETURNS LENGTH ;
   /* “半轴 b 的长度” */ ... ;
END OPERATOR ;

OPERATOR THE_CTR ( E ELLIPSE ) RETURNS POINT ;
   /* “中心” */ ... ;
END OPERATOR ;

OPERATOR THE_R ( C CIRCLE ) RETURNS LENGTH ;
   /* “半径长度” */ ... ;
END OPERATOR ;
```

除了 THE_R 以外，所有这些操作都适用于属于 ELLIPSE 类型的值，因此也必然适用于属于 CIRCLE 类型的值。而相应地，THE_R 操作就只适用于属于 CIRCLE 类型的值。

**1. 术语**

在继续下面的讨论之前，我们还需要介绍一些定义和名词。但是这些概念非常浅显。

1）超类型的超类型还是超类型。比如，POLYGON 是 SQUARE（正方形）的一个超类型。

2）每个类型都是自己的一个超类型。比如，ELLIPSE 也是 ELLIPSE 的超类型。

3）如果 $A$ 是 $B$ 的超类型，而 $B$ 是不同于 $A$ 的类型，则 $A$ 是 $B$ 的一个**真**（proper）超类型。比如，POLYGON 是 SQUARE 的真超类型。如果 $A$ 是 $B$ 的一个真超类型，那么 $A$ 也是 $B$ 的一个真超集——也就是，类型 $A$ 至少有一个值不是类型 $B$ 的值。

类似的说法也适用于子类型。即：

4）子类型的子类型还是子类型。比如，SQUARE 是 POLYGON 的一个子类型。

5）每个类型都是自己的一个子类型。比如，ELLIPSE 也是 ELLIPSE 的子类型。

6）如果 *B* 是 *A* 的子类型，而 *A* 是不同于 *B* 的类型，则 *B* 是 *A* 的一个真子类型。比如，SQUARE 是 POLYGON 的真子类型。如果 *B* 是 *A* 的一个真子类型，那么 *B* 也必须是 *A* 的一个真子集。

还有：

7）如果 *A* 是 *B* 的超类型，而且不存在一个类型 *C* 既是 *A* 的真子类型又是 *B* 的真超类型，则 *A* 是 *B* 的一个**直接**（immediate）超类型，而 *B* 是 *A* 的一个**直接**子类型。比如，RECTANGLE（矩形）是 SQUARE 的直接超类型，同时 SQUARE 是 RECTANGLE 的直接子类型。因此要注意，在我们的 **Tutorial D** 的语法中，关键字 SUBTYPE_OF 的意思是特指“是……的直接子类型”。

8）**根**（root）类型是没有超类型的类型。比如，PLANE_FIGURE 就是一个根类型。注意：我们并没有假设只有一个根类型。但是，如果有两个或多个根类型，我们总可以构造出一个“系统”类型做为所有这些类型的直接超类型，所以假设只有一个根类型并不会丧失普遍性。

9）**叶**（leaf）类型是没有子类型的类型。比如，CIRCLE 就是一个叶类型。注意：这样一个定义是有些简化了的，但是对于目前的讨论是足够了（要完全适合多重继承，它还需要进行一些小小的扩充［3.3］）。

10）每个真子类型只有一个直接超类型。注意：这里要明确的是我们的假设只是针对单一继承的。如前所述，放宽假设的情况在文献［3.3］中有详细讨论。

11）只要（a）至少存在一个类型；而且（b）不存在回路——即不存在一系列的类型 *T*1、*T*2、*T*3、…、*Tn*，使得 *T*1 是 *T*2 的直接超类型，*T*2 是 *T*3 的直接超类型……而 *Tn* 是 *T*1 的直接超类型，则至少有一个类型必定是根类型。注意：一定是不可能存在任何回路的。（为什么不可能呢?）

**2. 不相交假设**

再做如下一个简化假设：如果 *T*1 和 *T*2 是不同的根类型，或者是同一超类型的不同直接子类型（尤其是指任何一个都不是另外一个的子类型），则假设它们是**不相交**的——即没有既属于类型 *T*1 又属于类型 *T*2 的值。例如，没有既属于椭圆又属于多边形的值存在。以下接上一小节 11 条的观点是这一假设的直接推论：

12）不同的类型层次结构之间是不相交的。

13）不同的叶类型之间是不相交的。

14）每个值都只有一个最确切（most specific）的类型。比如，一个给定值也许“只是一个椭圆”而不是一个圆，这就是说，它的最确切的类型是 ELLIPSE（在现实世界中，很多椭圆都不是圆）。实际上，更严格地说，如果某个值 *v* 的最确切类型是 *T*，则指 *v* 所属的类型的集合就是 *T* 的所有超类型的集合（当然其中也包括 *T* 本身）。

需要不相交假设的一个原因是它可以避免可能出现的多义性。假设某个值 *v* 同时属于类型 *T*1 和 *T*2，两者都不是对方的子类型。进一步假设 *T*1 定义了一个叫 Op 的操作，而 *T*2 也定义了一个叫 Op 的操作，则对 *v* 应用 Op 操作就会产生二义性。

注意：不相交假设在我们只把注意力局限于单一继承的时候才是合理的，但是对于多重继承的情况，这一假设必须放宽。具体的讨论可以见参考文献［3.3］。

**3. 关于物理表示形式的一点说明**

虽然我们主要关心的是继承的模型而不是继承的实现问题，但是为了能够对继承的整体概念有一个正确的理解，必须对特定的实现问题有一定程度的了解。下面就进行这样一点说明：

*B* 是 *A* 的子类型这一事实并不能说明，*B* 的值的实际表示形式（对用户不可见）与 *A* 的值是一样的[⊖]。比如，椭圆实际上可能是用其中心和半轴来表示的，而圆可能是用其圆心和半径来表示的（虽然一般而言，实际的表示形式并不需要和任何已知的可能表示形式一样）。这一点在

---

⊖ 事实上，为什么相同类型的所有值都必须有相同的物理表示并没有逻辑原因。例如，有些点是用笛卡尔坐标来进行物理表示而有些则用极坐标表示；有些温度是用摄氏温度表示，有些则用华氏温度表示；有些整数是十进制的而有些是二进制的等等。当然，系统必须知道在这些情况下该如何转换这些物理表示以实现配置、比较等。

下面的许多小节中都会显得相当重要。

## 20.3　多态性和可置换性

这一节研究两个非常重要的概念：多态性和可置换性。由这两者共同构成的基础，可以获得在20.1节中提到的代码重用带来的好处。还必须明确一点，即这两个概念实际上是从不同的角度看待同一件事情。下面让我们先来看看多态性。

**1. 多态性**

继承的概念意味着，如果 $T'$ 是 $T$ 的一个子类型，则所有适用于 $T$ 的值的操作也同样适用于 $T'$ 的值。例如，如果 AREA（$e$）是合法的，这里 $e$ 是一个椭圆，并且如果 $c$ 是一个圆，则 AREA（$c$）也是合法的。由此就要非常注意一个给定操作的**参变量**（parameter）以及它们的声明类型与调用该操作时的相应**参数**（argument）以及它们的实际（最确切）类型之间的差异。比如，操作 AREA 是通过一个声明类型为 ELLIPSE 的参变量来定义的（见20.2节），但是在调用中，AREA（$c$）的参数的实际（最确切）类型是 CIRCLE。

现在再次重申椭圆和圆可以具有不同的表示形式，至少在20.2节中是这样定义的：

```
TYPE ELLIPSE ...
     POSSREP { A ..., B ..., CTR ... } ;

TYPE CIRCLE ...
     POSSREP { R ..., CTR ... } ;
```

因此，很有可能在同样的形式下，AREA 操作存在两个不同的版本，一个针对 ELLIPSE 类型的可能表示形式，一个针对 CIRCLE 类型的可能表示形式。但是仅仅是很有可能——而不是一定。比如，ELLIPSE 类型可以有这样的代码：

```
OPERATOR AREA ( E ELLIPSE ) RETURNS AREA ;
   RETURN ( 3.14159 * THE_A ( E ) * THE_B ( E ) ) ;
END OPERATOR ;
```

（椭圆的面积是 $\pi ab$）。显然，当由一个圆而不是一个普通的椭圆来调用的时候，该代码也能得到正确的结果，因为对于一个圆来说，操作 THE_A 和 THE_B 返回的都是半径 $r$。但也许由于各种各样的原因，定义类型 CIRCLE 的人更愿意专门针对圆实现 AREA 的一个版本，那就是通过调用 THE_R 来代替对 THE_A 和 THE_B 的调用。*注意*：事实上就算表示形式相同，但是出于效率上的考虑，也可能需要对同一个操作用两个不同的版本实现。就拿多边形和矩形来说，计算普通多边形面积的算法肯定也适用于矩形，但是对于矩形而言还有一个更有效的算法——长乘宽。

同时应该注意到，如果 ELLIPSE 的代码是针对 ELLIPSE 的实际表示形式而不是可能的表示形式来写的，那么这些代码可能真的无法适用于圆，因为 ELLIPSE 和 CIRCLE 的实际表示形式是不同的。通常，针对实际的表示形式实现操作并不是一个好主意，编制代码要保守一些[⊖]。

无论怎样，如果不需要为 CIRCLE 重写 AREA 的代码，我们就实现了代码重用（对 AREA 的实现而言）。注意：在下一小节中我们还将遇到一种更重要的“重用”。

当然从模型的角度来看，AREA 到底有多少个版本并不重要（就用户而言，在定义中只有一个 AREA 操作，它既适用于椭圆也适用于圆）。换句话说，从模型的角度而言，AREA 是多态的：在不同的调用中它可以引用不同类型的参数。因此必须切实地注意到，这种多态性是继承的逻辑推论：如果采用继承，就必须接受多态性，否则就不能采用继承。

现在你也许已经意识到，多态性并不是一个新的概念，事实上我们在第5章就简单讨论过这个问题。比如，SQL 语言中就有多态操作（“=”、“+”、“‖”等），事实上大部分其他程序设计语言也有多态操作。有些语言甚至允许用户定义自己的多态操作。比如 PL/I 就通过所谓的“GE-

---

⊖ 实际上，我们建议对物理表示的访问仅限于实现由模型指定的操作符（selector，THE_operators）的代码。（而且很多操作符事实上都具有系统提供的实现方法。）

NERIC 函数”提供这样的功能。但是在前文所述的各个例子中并没有实现任何的继承，它们都是通常所说的**重载**多态的例子。而相应地，操作 AREA 所表现的多态叫做**包含**多态，这是因为椭圆与圆之间的关系主要是集合之间的包含关系［20.4］。因此，在本章的余下部分里，我们都用“多态”来表示包含多态，不使用其完整的表述形式。

注意：下面的说明也许有助于了解重载多态与包含多态之间的区别：

- 重载多态是指许多不同的操作具有相同的名字。（而且用户并不需要知道这些操作事实上是不同的，同时语义也不同——当然语义相似更好。）比如，“+”在大部分语言中都是重载的。（比如：一个操作“+”用于整数相加，而另一个操作“+”用于有理数相加，等等。）
- 包含多态是指只有一个操作，但是在同一个形式下有多个不同的实现版本。（但是用户并不需要知道是否真的有多个实现版本。重申一遍，用户只知道有一个操作。）

**2. 利用多态编程**

看下面这个例子。设想我们要写一个程序来显示一些由正方形、椭圆以及圆等构成的图形。如果不用多态，代码可能与下面的伪代码类似：

```
FOR EACH x ∈ DIAGRAM
    CASE ;
       WHEN IS_SQUARE ( x ) THEN CALL DISPLAY_SQUARE ... ;
       WHEN IS_CIRCLE ( x ) THEN CALL DISPLAY_CIRCLE ... ;
         ...
    END CASE ;
```

（假设有诸如 IS_SQUARE、IS_CIRCLE 这样的操作来检查一个给定的值是否属于特定的类型）。相应地，如果采用多态，代码就会非常简洁：

```
FOR EACH x ∈ DIAGRAM CALL DISPLAY ( x ) ;
```

说明：DISPLAY 在这里是一个多态操作。针对不同类型 $T$ 的值，DISPLAY 的实现版本会在定义类型 $T$ 的时候相应地进行定义，同时告知系统。这样在运行的时候，如果系统遇到了带参数 $x$ 的 DISPLAY 调用，它必须识别出 $x$ 的最确切类型，然后调用适合该类型的 DISPLAY 版本——这是一个动态联编（run-time binding）[⊖] 的过程。换句话说，多态实际上是把原来出现在用户源代码中的 CASE 表达式和 CASE 语句转移到了统一的表示形式下：系统实际上代替用户执行了 CASE 操作。

让我们着重看看维护上述程序意味着什么。举个例子，假如又定义了一个新类型 TRIANGLE（三角形），作为 POLYGON 的另外一个直接子类型，那么现在显示的图形就又可以包括三角形了。如果不用多态，则每一个如同上面一样包含 CASE 表达式或 CASE 语句的程序现在都必须修改，加上如下形式的代码：

```
WHEN IS_TRIANGLE ( x ) THEN CALL DISPLAY_TRIANGLE ... ;
```

而如果采用多态，这样的源代码修改就不需要了。

像这样的例子有很多，有时候多态性会被形象地说成是“让旧代码调用新代码”，即一个程序 $P$ 可以有效地调用某些操作的版本，即使在编写程序的时候这些版本还不存在。所以现在我们有了另外一个（也是更重要的）代码重用的例子：即使在编写程序 $P$ 时类型 $T$ 并不存在，$P$ 仍然可以应用在属于 $T$ 的数据上。

**3. 可置换性**

就像前面提到的，可置换性实际上是从一个稍微有些不同的角度来看待多态性这一概念。比如，我们已经看到，当 $e$ 为椭圆时，如果 AREA（$e$）是合法的，则当 $c$ 为圆时，AREA（$c$）也是合法的。换句话说，在系统中任何可以接受椭圆的地方，都可以用圆来置换。更一般地，在系

---

⊖ 动态联编是实现时需要考虑的问题，但不是模型方面所要考虑的。为了更好地理解整个继承的概念，必须对动态联编有一定的了解。

统中任何可以接受属于类型 $T$ 的值的地方，都可以用属于类型 $T'$ 的值来置换，这里的 $T'$ 是 $T$ 的子类型。这就是值的可置换性原则。

特别要注意的是在这一原则下，如果某个关系 $r$ 的属性 $A$ 声明为 ELLIPSE 类型，则 $r$ 中 $A$ 的值可以属于类型 CIRCLE 而不仅仅属于类型 ELLIPSE。类似地，如果某个类型 $T$ 的一个可能的表示形式中有一个声明为 ELLIPSE 类型的分量 $C$，那么对于属于类型 $T$ 的某些值 $v$ 而言，执行操作 THE_C（$v$）所得到的返回值可能属于类型 CIRCLE 而不仅仅属于类型 ELLIPSE。

最后，我们注意到，既然可置换性是多态性的另一种表示，则它同样也是继承的逻辑推论：如果有继承存在，就必定要接受可置换性，否则就没有继承。

## 20.4　变量与赋值

假设有两个变量 $E$ 和 $C$ 分别声明为类型 ELLIPSE 和 CIRCLE。

```
VAR E ELLIPSE ;
VAR C CIRCLE ;
```

首先我们用某个圆初始化 $C$——比如（为了确切起见）是一个半径为 3、中心在原点的圆：

```
C  :=  CIRCLE ( LENGTH ( 3.0 ), POINT ( 0.0, 0.0 ) ) ;
```

表达式右边是对类型 CIRCLE 的选择子操作的调用。在第 5 章中曾说过，对于每一种声明的可能表示形式都有一个同名的选择子操作，该操作的参变量对应于该表示形式的各个分量。选择子操作的作用是允许用户通过给相应的表示形式的每个分量赋值，来指定或者说“选择”一个属于该类型的值。注意：为了简单起见，我们这里假设用笛卡尔表示的点乘被称为 POINT，而不是像第 5 章所说的 CARTESIAN。例子中 CIRCLE 选择的第二个参数作为点乘的笛卡尔选择的一个调用。

现在考虑如下的赋值：

```
E  :=  C ;
```

一般情况下（即不存在子类型和继承的情况下）赋值操作要求表达式左边的变量与表达式右边指定的值具有相同的类型（对于变量而言是具有相同的声明类型）。但是值的可置换性原则表明在系统中任何可以接受属于类型 ELLIPSE 的值的地方，都可以用一个 CIRCLE 类型的值来置换，所以上面的赋值是合法的（实际上“赋值”是一个多态操作）。其效果是从变量 $C$ 赋一个圆的值到变量 $E$ 中，而且，赋值后变量 $E$ 的值的类型是 CIRCLE 而不仅仅是 ELLIPSE。换句话说：

- **在给一个声明类型为非确切（less specific）类型的变量赋值时，值可以保持其自身的最确切类型。**在这类赋值中并不出现类型转换（在这个例子中，圆并不被转换成一个椭圆）[⊖]。注意，我们实际上并不希望出现任何类似的转换，因为这样会让所赋的值丢失其最确切行为，对这个例子而言，就是在赋值之后我们将无法得到变量 $E$ 中圆的半径值。注意：在本节后面的“类型下移”这一小节中将讨论取得半径所涉及的问题。
- 由此可见，可置换性暗示着一个声明为类型 $T$ 的变量可以具有任何值，只要这个值的最确切类型为 $T$ 的**任意一个子类型**。因此，必须仔细区分一个变量的声明类型和该变量（当前值）的实际类型（即最确切类型）之间的区别。在下一小节中我们还要对这个重要问题进行讨论。

接下来在这个例子中，假设有另外一个声明为 AREA 类型的变量 $A$：

```
VAR A AREA ;
```

考虑如下的赋值：

---

⊖ 事实上，瞬间的影响（moment's reflection）使得转化的想法变得没有意义——也就是说如果转换是可能的，那么有些值将同时拥有两个最确切类型。

```
A  :=  AREA ( E ) :
```

将会产生如下后果：

- 首先，系统将对表达式 AREA（*E*）执行编译时的类型检查。由于 *E* 的声明类型为 ELLIPSE，同时操作 AREA 的唯一参变量的声明类型也是 ELLIPSE（见 20.2 节），类型检查可以通过。
- 其次，在运行的时候系统发现 *E* 的当前最确切类型是 CIRCLE，因此调用适用于圆的 AREA 的版本（换句话说，系统执行了在前面一节讨论过的动态联编过程）。

当然，系统实际调用的是 AREA 的圆的版本而不是椭圆的版本，用户对此并不关心——再次说明，对于用户而言只有一个 AREA 操作。

**1. 变量**

对于声明类型为 *T* 的标量变量 *V* 来说，其当前值 *v* 的最确切类型可以是 *T* 的任意子类型。由此可见，可以用一个形如 < *DT*, *MST*, *v* > 的有序三元组来把 *V* 模型化（我们也确实是这样做的），其中：

- *DT* 是变量 *V* 的声明类型。
- *MST* 是变量 *V* 的当前最确切类型（意思是值的最确切类型就是变量 *V* 的当前值）。
- *v* 是一个属于最确切类型 *MST* 的值——也就是变量 *V* 的当前值。

用符号 *DT*（*V*）、*MST*（*V*）和 *v*（*V*）分别代表标量变量 *V* 的声明类型、当前最确切类型和当前值。注意到（a）*MST*（*V*）总是 *DT*（*V*）的子类型但不必是真子类型；（b）一般来说，*MST*（*V*）和 *v*（*V*）是随着时间变化的；（c）实际上 *MST*（*V*）是包含在 *v*（*V*）中的，因为每个值只有一个最确切类型。

标量变量的这个模型对于明确各种操作的精确语义是非常有用的，特别是对于赋值操作。在进行详细阐述之前，首先要说明，声明类型和当前最确切类型的概念显然也可以方便地扩展到任意的标量表达式上，而不仅仅是针对标量变量。设 *X* 是这样一个表达式，则：

- *X* 有一个声明类型 *DT*（*X*）——即在 *X* 最外层级别上调用的操作 Op 的声明类型。*DT*（*X*）是在编译的时候确定的。
- *X* 同样还有一个最确切类型 *MST*（*X*）——即 *v*（*X*）的最确切类型。*MST*（*X*）直到运行的时候才确定。

现在我们可以恰当地解释赋值了。考虑赋值操作

```
V  :=  X ;
```

（其中 *V* 是一个标量变量，*X* 是一个标量表达式）。*DT*（*X*）必须是 *DT*（*V*）的子类型，否则赋值操作是不合法的（这是编译时进行的检查）。如果赋值操作是合法的，其结果是使得 *MST*（*V*）等于 *MST*（*X*）、*v*（*V*）等于 *v*（*X*）。

顺便提一句，如果变量 *V* 的当前最确切类型是 *T*，则 *T* 的每个真超类型也是变量 *V* 的“当前类型”。比如，如果变量 *E*（声明类型为 ELLIPSE）当前值的最确切类型是 CIRCLE，则 CIRCLE、ELLIPSE 和 PLANE_FIGURE 都是 *E* 的“当前类型”。但是，至少是非正式地，“*X* 的当前类型”通常特指 MST（*X*）。

**2. 对可置换性的再讨论**

考虑如下的操作定义：

```
OPERATOR COPY ( E ELLIPSE ) RETURNS ELLIPSE ;
   RETURN ( E ) ;
END OPERATOR ;
```

根据可置换性，COPY 操作的执行参数的最确切类型是 EI LIPSE 或 CIRCLE——无论是哪一种类型，其返回值显然也会具有同样的最确切类型。由此可见，可置换性的概念具有进一步的含义，即（一般而言）如果定义操作 *Op* 的返回值的声明类型为 *T*，则执行操作 *Op* 得到的实际结果可以属于 *T* 的任意子类型。换句话说，就是（a）一般情况下，引用一个声明类型为 *T* 的变

量，实际上得到的可能是 $T$ 的任意子类型的一个值；所以（b）同样，执行一个声明类型为 $T$ 的操作，实际上返回的值可能属于 $T$ 的任何一个子类型。

### 3. 类型下移

同样用上面的例子：

```
VAR E ELLIPSE ;
VAR C CIRCLE ;

C  :=  CIRCLE ( LENGTH ( 3.0 ), POINT ( 0.0, 0.0 ) ) ;
E  :=  C ;
```

现在 *MST*（$E$）为 CIRCLE。假设要得到例子中圆的半径，并把它赋值给变量 L。我们也许会这样做：

```
VAR L LENGTH ;

L  :=  THE_R ( E ) ;          /* 编译时会出现类型错误 !!! */
```

但是，就像注释所指出的，上面的代码在编译时会出现类型错误。更确切地说，是由于赋值表达式右边的操作 THE_R（取半径）需要一个类型为 CIRCLE 的参数，而参数 $E$ 的声明类型是 ELLIPSE 而不是 CIRCLE。*注意*：如果编译时不进行类型检查，则假如运行的时候 $E$ 的当前值是一个椭圆而不是圆，我们会得到一个运行时的类型错误，这种情况就更糟糕了。当然，对于目前的情况，其实我们知道运行时 $E$ 的值是一个圆，问题是我们知道但是编译器不知道。

为了解决这类问题，引入一个新的操作，并非正式地以 TREAT DOWN（类型下移）来称呼它。则例子中获取半径的正确方法如下：

```
L  :=  THE_R ( TREAT_DOWN_AS_CIRCLE ( E ) ) ;
```

定义表达式 TREAT_DOWN_AS_CIRCLE（E）的声明类型为 CIRCLE，这样编译时的类型检查就通过了。那么在运行的时候：

- 如果 $E$ 的当前值确实是一个圆，则整个表达式正确地返回该圆的半径。确切地说，执行 TREAT DOWN 会产生一个结果，比如说是 $Z$，则（a）$Z$ 的声明类型 DT（$Z$）等于 CIRCLE，因为有形如“…_AS_CIRCLE”的说明；（b）$Z$ 的当前最确切类型 MST（$Z$）等于 MST（$E$），在这个例子中同样是 CIRCLE；（c）$Z$ 的当前值 $v$（$Z$）等于 $v$（$E$）；（d）表达式“THE_R（Z）”经过计算给出所要的半径（之后被赋值给 $L$）。
- 但是如果 $E$ 的当前值只是属于类型 ELLIPSE 而不是 CIRCLE，则 TREAT DOWN 在运行时会出现类型错误。

通常，使用类型下移的目的在于保证如果出现运行时的类型错误，那么可以肯定是在调用 TREAT DOWN 的时候出现了错误。

*注意*：假设 CIRCLE 有一个真子类型，比如说是 O_CIRCLE（“O-circle”是指一个圆心在原点的圆）：

```
TYPE O_CIRCLE
    IS CIRCLE
       CONSTRAINT THE_CTR ( CIRCLE ) = POINT ( 0.0, 0.0 )
       POSSREP { R = THE_R ( CIRCLE ) } ;
```

则变量 $E$ 的最确切类型在某些时候就可能是 O_CIRCLE 而不是 CIRCLE。如果是这样，则下面的 TREAT DOWN 操作

```
TREAT_DOWN_AS_CIRCLE ( E )
```

就可以成功执行，并生成结果，比如说是 $Z$，并且（a）由于有“…_AS_CIRCLE”的说明，DT（$Z$）等于 CIRCLE；（b）MST（$Z$）等于 O_CIRCLE，因为 $E$ 的最确切类型是 O_CIRCLE；（c）$v$（$Z$）等于 $v$（$E$）。换句话说，TREAT DOWN 总会产生最确切类型，而不可能“向上提升”类型的层次。

为了便于今后的讨论，对于操作调用 TREAT_DOWN_AS_T（*X*），这里给出一个更加规范的语义说明，其中 *X* 是一个标量表达式确实。首先，最重要的是 *T* 必须是 DT（*X*）的子类型（这是编译时进行的类型检查）。其次，MST（*X*）必须是 *T* 的子类型（这是运行时进行的类型检查）。如果这些条件都得到了满足，调用返回一个结果 *Z*，并且 DT（*Z*）等于 *T*，MST（*Z*）等于MST（*X*），v（*Z*）等于 v（*X*）。注意：参考文献［3.3］也定义了一个一般形式的 TREAT DOWN，允许将一个操作数的类型“下移为”（be treated down）另外的类型，而不是某个明确命名的类型。

## 20.5　约束特化

考虑调用如下一个类型为 ELLIPSE 的选择子操作：

```
ELLIPSE ( LENGTH ( 5.0 ), LENGTH ( 5.0 ), POINT ( ... ) )
```

这个表达式返回一个半轴相等的椭圆。但是，在现实世界中半轴相等的椭圆实际上是圆，那么这个表达式能否返回一个最确切类型是 CIRCLE 而不是 ELLIPSE 的值呢？

对于类似这一类的问题在学术界曾经（即使现在也是）引起非常多的争论。经过仔细考虑，我们决定在自己的模型中，最好还是让这样的表达式确实能够返回一个最确切类型为 CIRCLE 的值。更一般地，如果类型 *T′* 是类型 *T* 的子类型，而调用类型 *T* 的一个选择子操作会返回一个满足 *T′* 约束的值，则（在我们的模型里）选择子操作的调用结果就是一个类型为 *T′* 的值。注意：当今的商业系统几乎都没有在实现的时候这样做，但是我们认为这正是那些系统的失败之处。参考文献［3.3］表明，由于这种缺陷，这些商业系统被迫支持“不是圆形的圆”、“不是正方形的正方形”以及类似的一些毫无意义的东西，但是我们的方法不会出现这样的后果。注意：同样也可以参考第 26 章中第二个重大失误中的讨论。

由上述可知，（至少在我们的模型里）不存在最确切类型为 ELLIPSE 且 $a = b$ 的值。换句话说，最确切类型为 ELLIPSE 的值确切地对应于现实世界中的椭圆而不是圆。与此相反，在其他继承模型中，最确切类型为 ELLIPSE 的值对应于现实世界中的椭圆，但是这些椭圆可能是圆也可能不是圆。由此我们觉得我们的模型作为“现实的一个模型”更容易被人接受。

像 $a = b$ 的椭圆必然属于 CIRCLE 类型这样的概念在参考文献［3.3］中被称为**约束特化**（specialization by constraint），但是必须说明的是，其他作者在使用这个词或与它类似的词时可能有完全不同的含义（可见参考文献［20.10，20.14］）。

### 1. 对 THE_伪变量的再次讨论

在第 5 章中曾说过，THE_伪变量的作用是修改变量的某个分量，同时保持其他分量不变。（这里的分量是指某种可能表示形式的分量，而不是实际表示形式的分量。）比如，变量 *E* 的声明类型是 ELLIPSE，其当前值是一个 $a = 5$、$b = 3$ 的椭圆。则赋值表达式：

```
THE_B ( E )  :=  LENGTH ( 4.0 ) ;
```

将把变量 *E* 的半轴 *b* 改为 4，而保持半轴 *a* 和中心不变。那么，如第 5 章所说，从逻辑上讲是不需要 THE_伪变量的，它们只是一种快捷的形式而已。比如，上面使用 THE_伪变量的赋值表达式实际上可以用下面的表达式来代替，只是不够简洁：

```
E  :=  ELLIPSE ( THE_A ( E ), LENGTH ( 4.0 ), THE_CTR ( E ) ) ;
```

所以考虑如下的赋值：

```
THE_B ( E )  :=  LENGTH ( 5.0 ) ;
```

根据定义，这个表达式与下面的表达式等价：

```
E  :=  ELLIPSE ( THE_A ( E ), LENGTH ( 5.0 ), THE_CTR ( E ) ) ;
```

约束特化在这时就起作用了（因为表达式的右边返回了一个 $a = b$ 的椭圆），则最终效果是赋值

以后 MST（*E*）为 CIRCLE 而不是 ELLIPSE。

接下来再看看下面的赋值操作：

```
THE_B ( E )  :=  LENGTH ( 4.0 ) ;
```

现在 *E* 包含一个 $a=5$、$b=4$ 的椭圆（和前面一样），则 *MST*（*E*）再次变为 ELLIPSE。这是我们所说的约束**泛化**（generalization by constraint）的效果。

注意：假设（如 20.4 节最后一个例子）CIRCLE 类型有一个真子类型 O_CIRCLE（“O-circle”代表圆心在原点的圆）：

```
TYPE O_CIRCLE
     IS CIRCLE
        CONSTRAINT THE_CTR ( CIRCLE ) = POINT ( 0.0, 0.0 )
        POSSREP { R = THE_R ( CIRCLE ) } ;
```

则变量 *E* 当前值的最确切类型在某些时候就可能是 O_CIRCLE 而不是 CIRCLE。如果是这样，那么考虑下面一系列的赋值操作[⊖]：

```
THE_A ( E )  :=  LENGTH ( 7.0 ) ;
THE_B ( E )  :=  LENGTH ( 7.0 ) ;
```

执行了第一个赋值操作之后，*E* 将包含一个“真正的椭圆”，这是约束泛化的作用。在执行了第二个赋值操作之后，它又将变回一个圆。但是将变回一个确切的 O_circle，还是“仅仅是一个圆”？显然，我们希望它是一个 O_circle。事实也是如此，准确地说，是因为它满足类型 O_CIRCLE 的约束（包括从 CIRCLE 类型继承来的约束）。

**2. 类型转化的分支问题**

声明变量 *E* 的类型为 ELLIPSE。我们已经看到如何将 *E* 的类型“下移”（比如，当 *E* 目前最确切类型是 ELLIPSE 时，如何将当前最确切类型修改为 CIRCLE）；也看到了如何将 *E* 的类型“上移”（比如，当 *E* 当前最确切类型是 CIRCLE 时，如何将当前最确切类型修改为 ELLIPSE）。但是如何处理类型转化的“分支”问题呢？假设我们对例子中的类型 ELLIPSE 进行扩展，使其有 CIRCLE 和 NONCIRCLE（非圆）两个直接子类型[⊜]。无需进行过多细节上的讨论，很清楚地，会有如下的结果：

- 如果 *E* 的当前值属于类型 CIRCLE（即 $a=b$），则修改 *E*，令 $a>b$，会让 MST（*E*）变成 NONCIRCLE。
- 如果 *E* 的当前值属于类型 NONCIRCLE（即 $a>b$），则修改 *E*，令 $a=b$，会让 MST（*E*）变成 CIRCLE。

这样，约束特化同样可以解决类型改变的分支问题。注意：很显然，实际上修改 *E* 使得 $a<b$ 是不可实现的（这与类型 ELLIPSE 的约束冲突）。

## 20.6　比较

两个变量 *E* 和 *C* 的声明类型仍然是 ELLIPSE 和 CIRCLE，假设用 *C* 的当前值给 *E* 赋值：

```
E  :=  C ;
```

显然，如果现在进行相等性比较

```
E  =  C
```

得到的结果应该为“真”，事实也确实如此。一般的规则如下：假设 *X*，*Y* 为任意形式的表达式，如果声明类型 *DT*（*X*）和声明类型 *DT*（*Y*）有一个公共子类型（就是说其中一个必须是另外一个变量的子类型）则比较式 $X=Y$ 是合法的，否则比较是不合法的（这是编译时进行的类型检查

---

⊖ 如果支持多次分配，我们就可以将一个序列作为一个单一的操作来实现。

⊜ 顺便提一下，ELLIPSE 现在成为一个合并类型。见 20.7 节。

实现的)。如果比较是合法的，并且值 v (X) 等于值 v (Y)，则其返回结果是“真”，否则为“假”。尤其要注意的是如果两个值的最确切类型不同，则它们不能进行“相等性比较”，因为如果 v (X) 等于 v (Y)，那么 MST (X) 就必须等于 MST (Y)。

**1. 对关系代数的影响**

从第7章可以了解到，在关系代数的许多操作中总会显式或隐式地及到相等性比较[○]。当涉及超类型和子类型的时候，其中某些操作所表现出来的状态可能会让人觉得与想象中有些不一样(至少一眼看上去是有些不一样)。考虑如图 20-2 所示的两个关系 *RX* 和 *RY*，可以看到 *RX* 唯一的属性 *A* 的声明类型是 ELLIPSE，而相应地，在 *RY* 中 *A* 的声明类型为 CIRCLE。如图 20-2 所示，我们用形如 *Ei* 的标识符来表示不是圆的椭圆，用 *Ci* 来表示圆。最确切类型用斜体表示。

现在考虑 *RX* 和 *RY* 的连接 *RJ* (见图 20-3)。显然 *RJ* 中的每个属性 *A* 的值都必须属于类型 CIRCLE (因为对于 *RX* 中任何属性 *A* 的值，其最确切类型如果是 ELLIPSE 的话，是不能和 *RY* 中属性 *A* 的值进行“相等性比较”的)。因此你也许会认为 *RJ* 中属性 *A* 的声明类型应该是 CIRCLE 而不是 ELLIPSE。但是先让我们考虑一下下面的问题：

| RX | A : ELLIPSE |
|---|---|
| | E1 : *ellipse*<br>C2 : *circle* |

| RY | A : CIRCLE |
|---|---|
| | C2 : *circle*<br>C3 : *circle* |

图 20-2 关系 *RX* 和 *RY*

- 既然 *RX* 和 *RY* 都只有一个属性 *A*，则 *RX* JOIN *RY* 可以简化成 *RX* INTERSECT *RY*。那么在这样的情况下，确定 JOIN 的结果中属性的声明类型的规则显然也需要简化成 INTERSECT 中的类似规则。
- *RX* INTERSECT *RY* 逻辑上等于 *RX* MINUS (*RX* MINUS *RY*)。设第二个操作——即 *RX* MINUS *RY* 的结果为 *RZ*。则显然有：
  a. 一般地讲，某些 *RZ* 中属性 *A* 的值的最确切类型为 ELLIPSE，所以 *RZ* 中属性 *A* 的声明类型一定是 ELLIPSE。
  b. 原来的表达式由此就变成了 *RX* MINUS *RZ*，而在 *RX* 和 *RZ* 中，属性 *A* 的声明类型都是 ELLIPSE。因此得到的最终结果中，属性 *A* 的声明类型显然又一定成了 ELLIPSE。

| RJ | A : ELLIPSE |
|---|---|
| | C2 : *circle* |

图 20-3 关系 *RX* 和 *RY* 的连接 *RJ*

- 由此可见，*RX* INTERSECT *RY* 操作的结果中，其属性的声明类型一定是 ELLIPSE 而不是 CIRCLE，因此 *RX* JOIN *RY* 的结果也是一样——即使是该属性中的每个值的类型实际上是 CIRCLE!

现在来看看关系的差操作符 MINUS。首先考虑 *RX* MINUS *RY*。很明显，该操作的结果属性 *A* 的某些值是 ELLIPSE 而不是 CIRCLE，所以结果中属性 *A* 的声明类型也一定是 ELLIPSE。那么 *RY* MINUS RX 又怎么样呢？显然，这个操作的结果中 *A* 的每个值都属于类型 CIRCLE，所以很自然地会想到，结果中 *A* 的值的声明类型会是圆而不是椭圆。但是可以发现，*RX* INTERSECT *RY* 在逻辑上不仅与 *RX* MINUS (*RX* MINUS *RY*) 等价，就像刚才讨论过的那样，而且与 *RY* MINUS (*RY* MINUS *RX*) 等价。由此很显然在 *RY* MINUS *RX* 的结果中，指定 *A* 的声明类型为 CIRCLE 会引起矛盾。可见，即使是在 *RY* MINUS *RX* 的情况下，每个属性值的类型实际上是 CIRCLE，MINUS 操作的结果属性的声明类型也必须为 ELLIPSE 而不是 CIRCLE。

最后来考虑 *RX* UNION *RY*。通常在这种情况下，结果属性 *A* 的某些值的最确切类型显然会是 ELLIPSE，则结果属性 *A* 的声明类型也必须是 ELLIPSE。这样，UNION 结果属性的声明类型也必须为 ELLIPSE (但是就本例情况而言，与 JOIN、INTERSECT 和 MINUS 不同，UNION 的结果在感觉上并不违反常规。)

下面是一般的规则：

- 设 *rx* 与 *ry* 是具有公共属性 *A* 的关系，并且用 *DT* (*Ax*) 和 *DT* (*Ay*) 相应地代表 *A* 的声明类型。考虑 *rx* 与 *ry* 的连接 (当然是在属性 *A* 上或至少是包括属性 *A*)。*DT* (*Ax*) 和 *DT*

---

○ 比较实际上是元组的比较，但对于当前的目的来说，我们可以将他们视为简单的标量比较。

（Ay）必须有一个公共的子类型 *T*，否则连接是不合法的（这是编译时进行的类型检查）。

如果连接是合法的，结果中属性 *A* 的声明类型是所有公共子类型 *T* 的**最确切类型**。

类似的分析也适用于“并、交、差”操作。在每种情况下，（a）操作数的相应属性的声明类型中必须有一个公共子类型；（b）结果中相应属性的声明类型是所有公共子类型的最确切类型。

**2. 类型检测**

在 20.3 节中，我们给出了一段带有形如 IS_SQUARE、IS_CIRCLE 等操作的代码，用来检测一个指定的值是否具有特定的类型。现在可以来进一步探讨这些操作。首先假设定义一个类型 *T* 会自动定义一个如下形式的判真（truth-valued）操作

```
IS_T ( X )
```

如果 *X* 属于类型 *T*，则返值为真，否则为假。例如，如果 *C* 是一个声明类型 CIRCLE 的变量，则表达式

```
IS_CIRCLE ( C )
IS_ELLIPSE ( C )
```

返回值都是真。如果 *E* 是声明类型 ELLIPSE 的变量，但是当前的最确切类型是 CIRCLE 的某个子类型，则表达式

```
IS_CIRCLE ( E )
```

也返回真。

类型检测还与关系操作有关。看看下面这个例子。关系变量 *R* 有一个类型为 ELLIPSE 的属性 *A*。假设我们希望获得 *R* 中所有 *A* 的值为圆、且半径大于 2 的元组，则我们也许会这样做：

```
R WHERE THE_R ( A ) > LENGTH ( 2.0 )
```

但是这个表达式在编译的时候会出现类型错误，因为 THE_R 需要一个类型为 CIRCLE 的参数，但是 *A* 的类型为 ELLIPSE 而不是 CIRCLE。（当然，如果不在编译时做类型检查，那么假如在运行时遇到一个元组，其中 *A* 的值为椭圆而不是圆的话，就会出现运行时的类型错误。）显然，我们需要做的就是，在检查半径之前就把 *A* 的值是椭圆的元组过滤掉。下面的表达式完成的就是这一功能：

```
R : IS_CIRCLE ( A ) WHERE THE_R ( A ) > LENGTH ( 2.0 )
```

简单的说，这个表达式是用来返回 *A* 的值为半径大于 2 的圆的元组。更准确一些，它返回了这样一个关系：

a. 属性名称与 *R* 一样，只是结果中属性 *A* 的类型是 CIRCLE 而不是 ELLIPSE；

b. 关系中只包含来自于 *R*、且 *A* 的值的类型是 CIRCLE、且半径大于 2 的元组。

换句话说，我们所讨论的是一个新的关系操作，其形式如下

```
R : IS_T ( A )
```

其中 *R* 是关系表达式，*A* 是该表达式所指关系（比如说是 *r*）的一个属性。表达式的所有值定义为一个关系：

a. 属性名称与 *r* 一样，除了属性 *A* 的类型在 *r* 中为 *T*；

b. 结果集是由来自关系 *r*、且属性 *A* 的值属于类型 *T* 的元组构成，只是在新关系中这些元组中属性 *A* 的声明类型为 *T*。

注意，参考文献［3.3］定义了本小节中所介绍的操作符的一般形式——例如，IS_*T* 的一般形式用于检测其中一个操作符与另一个操作符是否是相同类型而不是用于检测它是否是某个确切的命名类型。

## 20.7　操作、版本和签名

在 20.3 节中曾说过，一个给定的操作在相同的接口形式下可以有多个不同的实现版本（也叫做**显式特化**（explicit specialization））。也就是说，当顺着类型层次结构从超类型 $T$ 下溯到子类型 $T'$ 时，（由于各种原因）我们必须能够针对 $T'$ 重实现类型 $T$ 的操作。举个例子，考虑下面的 MOVE 操作：

```
OPERATOR MOVE ( E ELLIPSE, R RECTANGLE ) RETURNS ELLIPSE
   VERSION ER_MOVE ;
   RETURN ( ELLIPSE ( THE_A ( E ), THE_B ( E ),
                                   R_CTR ( R ) ) ) ;
END OPERATOR ;
```

简单地说，操作 MOVE 的作用是“移动”一个椭圆，使其中心与矩形 $R$ 的中心重合。或者说得更明确一些，它返回一个与参数椭圆 $E$ 一样的椭圆，只是这个椭圆的中心与参数矩形 $R$ 的中心重合。注意第二行的 VERSION 子句，它为这个特定版本的 MOVE（见下一段）引入了一个不同的名字 ER_MOVE。还要注意的是假设存在操作 R_CTR，其功能是返回指定矩形的中心点。

现在假设 MOVE 为另一个版本，即定义是移动圆而不是椭圆[⊖]：

```
OPERATOR MOVE ( C CIRCLE, R RECTANGLE ) RETURNS CIRCLE
   VERSION CR_MOVE ;
   RETURN ( CIRCLE ( THE_R ( C ), R_CTR ( R ) ) ) ;
END OPERATOR ;
```

通过类似的方法，还可以得到其他情况下的显式特化，比如参数的最确切类型分别是 ELLIPSE 和 SQUARE，相应地可以有 ES_MOVE；以及参数的最确切类型分别是 CIRCLE 和 SQUARE，相应地可以有 CS_MOVE。

**1. 签名**

简言之，签名就是某个操作的名字和该操作的操作数类型的结合体。（顺便提一下，不同的作者和不同的语言对这个词会有一些稍微不同的解释。比如，结果类型有时候会被认为是签名的一部分，而操作数和结果的名字有时候也是如此）。但是，我们还是要再认真地回顾一下：

a）参数和参变量；

b）声明类型和最确切类型；

c）用户所见的操作符和系统所见的操作符（是指就像上面所提到的在接口形式下那些操作的实现版本）之间的区别。

实际上，虽然从文字上经常区分不开，但是，对于一个给定的操作符 Op，我们至少可以区分出三种不同的签名：单个的描述签名，一组版本签名和一组调用签名，如下所示：

- **单个的描述签名**（specification signature）由两部分构成，一部分是操作的名字 Op，另一部分是 Op 的参变量的声明类型，它们按照定义 Op 时提供给用户的顺序排列。这个签名对应于用户所接受的操作 Op。比如 MOVE 的具体签名就是 MOVE（ELLIPSE，RECTANGLE）。

*注意*：参考文献［3.3］中认为应该区分一个给定操作符的描述签名的定义和该操作符所有实现版本的定义。基本思想是要支持合并类型，（也被理解为“抽象”或“非实例化”类型，或者有时候被认为是“接口”。）也就是说，合并类型根本不是任何值的最确切类型。这样的类型提供了一个**确定操作符的方式**，应用于若干个不同规则的类型上，而这些不同规则的类型都是合并类型的子类型。那么这个操作符的实现版本就可定义为这些规则子类型。用前面提到的例子来说明，PLANE_FIGURE 是一个合并类型；AREA 操作符的描述签名应定义在 PLANE_FIGURE 级，显式实现版本定义为 ELLIPSE 类型，对于类型 POLYGON 也是如此。

- 对应于 Op 的每个实现版本都有一个**版本签名**（version signature），它由操作 Op 的名称和

⊖ 实际上在这个特殊的例子中定义这样一个版本并没有什么意义。（为什么?）

该版本参变量的声明类型构成，声明类型是按照定义的顺序排列的。这些签名对应于 Op 实现代码的不同部分。比如，对于 MOVE 来说，版本 CR_MOVE 的版本签名是 MOVE (CIRCLE, RECTANGLE)。

- 各个参数的最确切类型的所有可能的组合都对应一个**调用签名**（invocation signature），操作符的每个参数都可以有若干个最确切类型，它由操作 Op 的名称加上参数最确切类型的某种组合情况一起构成。类型是按照定义的顺序排列的。这些签名对应于 Op 的各种可能的调用情况（当然这种对应是一对多的，即一个调用签名可以对应多个不同的调用）。比如，设 E 和 R 的最确切类型分别是 CIRCLE 和 SQUARE，则对于 MOVE 的调用 MOVE (E, R) 来说，调用签名是 MOVE (CIRCLE, SQUARE)。

因此，同一个操作的不同调用签名与该操作的不同实现版本相对应。这样，如果同一个操作符的接口下确实存在着多个版本，则在某一特定情况下要调用哪一个版本取决于哪一个的版本签名和应用的调用签名“最匹配”。决定是否最匹配的过程，即决定调用哪一个版本的过程是一个动态联编的过程（见 20.3 节）。

顺便要注意的是：(a) 描述签名是一个真正属于模型的概念；(b) 版本签名仅仅是一个属于实现的概念；(c) 虽然从某个角度上说调用签名是一个属于模型的概念，但是和可置换性一样，实际上它首先是类型继承的一个直接的逻辑推论。有可能存在不同的调用签名这一事实其实只是可置换性概念的一部分。

**2. 只读操作与更新操作**

迄今为止，我们一直默认 MOVE 是一个只读操作，但是假设现在把它作为一个修改操作：

```
OPERATOR MOVE ( E ELLIPSE, R RECTANGLE ) UPDATES E
   VERSION ER_MOVE ;
      THE_CTR ( E )  :=  R_CTR ( R ) ;
END OPERATOR ;
```

（注意，只读操作和修改操作有时候分别被叫做观察子（observer）和变异子（mutator）。如果你需要搞清楚这两者之间的差异，请参阅第 5 章。）

现在可以看到，调用这个版本的 MOVE 会修改它的第一个参数（简单地说是改变了该参数的中心）。进一步还可以发现，无论第一个参数的最确切类型是 ELLIPSE 还是 CIRCLE，修改都会起作用，换句话说，对于圆，是不需要显式实现版本的[⊖]。因此，一般而言，修改操作的好处之一就在于它可以不必明确地编写某些操作特化。尤其是对程序维护具有特别的意义，比如，如果引入 O_CIRCLE 作为 CIRCLE 的子类型会出现什么情况呢？（答案是：可以激活带有一个声明类型为 ELLIPSE 或 CIRCLE 的参变量的 MOVE，而当前的最确切类型 O_CIRCLE 尚未定义完全。但是一般来说，不能激活带有一个声明类型为 O_CIRCLE 的参变量的 MOVE）。

**3. 改变操作的语义**

当我们顺着类型层次结构下溯时，对于操作的重实现可以是合法的，这一事实有一个非常严重的后果：它可能改变操作的语义。比如，就 AREA 而言，可能会出现这种情况，类型 CIRCLE 的实现版本实际上返回的是圆周长而不是圆面积（细致的类型设计可以有助于在某种程度上缓和这一问题。如果操作 AREA 的返回类型被定义为 AREA 类型，显然执行是不可能返回一个 LENGTH 类型的结果，但是它仍然可能返回一个错误的面积值！）。

这种情况虽然看起来有些让人吃惊，甚至会有人说（实际上已经有人提出了）这种方式的语义改变是值得的。比如，设类型 TOLL_HIGHWAY 是类型 HIGHWAY 的真子类型，而操作 TRAVEL_TIME 的作用是计算在指定的高速公路上通过指定两点之间的行进时间。对于要征收过路费的高速公路来说，计算公式是 $(d/s) + (n * t)$，其中 $d$ = 距离，$s$ = 速度，$n$ = 收费站的数目、$t$ = 在收费站停留的时间。对于不收费的高速公路，计算公式则相应地为 $d/s$。

---

⊖ 正如 20.7 节第 1 个例子的脚注所标注的，在只读的情况下并不真正需要版本——我们这里的介绍仅仅是为了举例。

再举一个反例，即不欢迎语义改变的情况的例子，再次考虑椭圆和圆。应该说我们希望操作 AREA 的定义对于同一个圆应该能够返回同一个面积值，无论它是一个圆还是一个椭圆。假设顺序发生如下的事情：

1）定义类型 ELLIPSE 及相应版本的 AREA 操作。为简单起见，设在 AREA 的代码中并没有使用椭圆的实际表示形式。

2）定义类型 CIRCLE 为类型 ELLIPSE 的子类型，但是还没有针对圆定义一个 AREA 操作的单独实现版本。

3）对于某个特定的圆 *c* 调用 AREA，得到结果 *a1*，调用的是 ELLIPSE 版本的 AREA 因为它是目前唯一存在的版本。

4）现在针对圆定义一个 AREA 操作的单独实现版本。

5）对于同一个圆 *c* 再次调用 AREA，得到结果 *a2*（而这一次调用的是 CIRCLE 版本的 AREA）。

在这一点上，我们肯定要求“应该有” *a2* = *a1*，但是这一“应该有”的要求并不是强制的。即像已经说过的那样，有可能针对圆的 AREA 的实现版本会返回一个周长而不是面积，或是返回一个错误的面积值。

让我们再回到 TRAVEL_TIME 的例子。事实上，我们发现这个例子以及类似的例子是相当让人无法信服的——即无法让人相信在例子所展现的情况下，改变一个操作的语义可以被认为是有必要的。考虑下面内容：

- 如果 TOLL_HIGHWAY 确实是 HIGHWAY 的一个子类型，这意味着根据定义，每一条独立的收费高速公路首先是一条高速公路。
- 因此，某些高速公路（即某些 HIGHWAY 类型的值）实际上是收费高速公路——路上有收费站。所以 HIGHWAY 类型并不是“没有收费站的高速公路”，而是“可以有 *n* 个收费站的高速公路”（*n* 可以等于零）。
- 故而类型 HIGHWAY 的操作 TRAVEL_TIME 并不是“计算在一条没有收费站的高速公路上的行进时间”，而是“计算忽略了收费站后在一条高速公路上的行进时间 *d*/*s*”。
- 相应地，类型 TOLL_HIGHWAY 的操作 TRAVEL_TIME 是“在不忽略收费站的情况下，计算在一条高速公路上的行进时间（*d*/*s*）+（*n* * *t*）”。所以实际上这两个 TRAVEL_TIME 在逻辑上是不同的操作。由于这两个不同的操作具有相同的名字，这样就会产生混淆。实际上在这里我们引入了“重载多态”而不是“包含多态”。（需要说明的是：在实践中还会有进一步的混淆，因为非常遗憾的是在谈到包含多态的时候许多作者实际使用的却总是“重载”这个词。）

总而言之，改变操作的语义并不是一个好的想法。像我们已经看到的那样，这种要求不是强制的。但是我们当然可以定义自己的继承模型我们也确实这样做了，在模型中如果出现语义被改变的情况，则**执行过程是不合法的**（即该执行过程是不属于继承模型的，其含义是不可预知的）。应该注意到，我们对于这一问题的立场（即语义改变是不合法的）确实是有好处的，无论给定操作 *Op* 是否定义了任何显式特化，用户的感觉都是一样的，也就是：（a）只存在一个操作，一个唯一的操作叫做 *Op*；（b）该操作对于属于某个特定类型 *T* 的参数值有效，因此，根据定义对于属于 *T* 的真子类型的参数值也有效。

## 20.8　圆是椭圆吗

圆是椭圆吗？到现在为止我们在本章中都假设（并以充分的理由假设）是。但是现在我们必须面对这样一个事实，这种观点在学术界有很多争论 [20.6]。考虑我们常用的变量 *E* 和 *C*，其声明类型分别为 ELLIPSE 和 CIRCLE。假设这些变量做了如下的初始化：

```
E  :=  ELLIPSE ( LENGTH ( 5.0 ), LENGTH ( 3.0 ),
                                  POINT ( 0.0, 0.0 ) ) ;
C  :=  CIRCLE  ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;
```

特别要注意到的是 THE_A（C）和 THE_B（C）的值现在都是 5。

现在我们肯定可以对 E 做的一个操作是“修改半轴 $a$”，比如：

```
THE_A ( E )  :=  LENGTH ( 6.0 ) ;
```

但是如果我们要对 C 执行类似的操作

```
THE_A ( C )  :=  LENGTH ( 6.0 ) ;
```

就会出错！到底是什么错误呢？让我们来看看，如果确实进行了修改，则变量 $C$ 将不再包含一个“圆”，这与圆的约束 $a = b$ 冲突（$a$ 现在是 6，而 $b$ 应该还是 5，因为我们还没有对其进行修改）。换句话说，$C$ 现在将包含一个“非圆形的圆”，从而与类型 CIRCLE 的类型约束冲突。

既然“非圆形的圆”与逻辑和通常的感觉相冲突，那么首先不允许进行这种修改看起来是合理的。明显的方法是在编译的时候拒绝这种操作，即进行这样的定义，对于半轴 $a$ 或 $b$ 的修改（赋值）是不合语法的。换句话说，对于 THE_A 和 THE_B 的赋值不适用于类型 CIRCLE，而尝试进行这种修改会因为编译时的类型错误而失败。

注意：实际上这种赋值显然是不合语法的，我们曾说过，对 THE_伪变量的赋值实际上是一种缩写。比如对 THE_A（C）的赋值，如果是合法的话，将是类似如下形式的表达式的缩写：

```
C  :=  CIRCLE ( ... ) ;
```

于是表达式右边的 CIRCLE 选择子操作调用会包括一个值为 LENGTH（6.0）的 THE_A 参数，但是 CIRCLE 选择子操作需要的不是 THE_A 参数，而是 THE_R 参数和 THE_CTR 参数。所以原来的赋值显然是不合法的。

**1. 改变语义会发生什么**

为了维护这样的概念即对于圆而言，对 THE_A 和 THE_B 的赋值还是合法的，常常会提出如下的想法，即如果参数是一个圆，对于如 THE_A 的赋值应该重新定义——换句话说，要显式特化——也就是在这种情况下也要同时对 THE_B 赋值，这样，这个圆在修改之后仍然满足 $a = b$ 的约束。但是这种想法是我们要立即阻止的。之所以反对这一想法，至少是基于以下三个原因：

- 首先，对 THE_A 和 THE_B 赋值的语义在继承模型里不可以用上面所说的方式进行改变（这是非常慎重地规定的）。
- 其次，即使模型没有规定这些语义，但是我们已经说明：（a）通常随意改变一个操作的语义是一个不好的想法，（b）像这样去改变一个操作的语义还会产生副作用，就更不好了。保持一个操作的效果正好能够满足要求，这是一个很好的普遍性原则。
- 第三，也是最重要的。退一步讲，能够像上面所说的那样进行语义改变的机会也并不是总能有的。比如，设类型 ELLIPSE 有另外一个直接子类型 NONCIRCLE；同时对于“非圆形”有约束 $a > b$；对于一个非圆形的 THE_A 操作的赋值会使得 $a = b$。那么对于这一赋值来说，其合适的语义重定义又是什么呢？准确地说，产生什么样的副作用是合适的呢？

**2. 一个合理的模型是否确实存在**

现在面临这样一种状况，即对 THE_A 和 THE_B 的赋值操作普遍适用于椭圆却并不特定地适用于圆。但是：

a）类型 CIRCLE 被假设为类型 ELLIPSE 的子类型；

b）类型 CIRCLE 是类型 ELLIPSE 的子类型意味着，普遍适用于椭圆的操作也特定地适用于圆——换句话说，操作是被继承的。

c）但是我们现在却说对 THE_A 和 THE_B 的赋值操作没有被继承下来。

这样我们不是自相矛盾了吗？现在该怎么办？

在回答这些问题之前，先要强调一下这一问题的严肃性。前面的讨论看起来真的像一个引线（threadpuller）。因为，如果类型 CIRCLE 没有从类型 ELLIPSE 继承这个操作，那我们又凭什么说

一个圆“是”一个椭圆呢？如果某些操作事实上最终无法得到继承，那么“继承”还能成立吗？一个合理的继承模型确实存在吗？我们试图找到这样一个模型是不是就成为了一种幻想？

注意：一些作者确实提出（郑重地提出）对 THE_A 的赋值对于圆和椭圆都应该有效（对于一个圆而言，它修改的是半径），而对 THE_B 的赋值只对于椭圆有效，所以事实上 ELLIPSE 应该是 CIRCLE 的子类型！换句话说，我们把类型的层次结构颠倒过来了。但是，只要稍微动动脑子，就足以发现这种想法是不可能实现的，特别是可置换性将会被破坏（一个一般意义下的椭圆，它的半径是什么?）。

恰恰是上面的那些想法使得某些作者得出结论，一个合理的继承模型是没有的（见本章结尾“参考文献和简介”中参考文献［20.2］的注释）。另一些作者则提出了一些继承模型，这些模型具有不合常理的或是不必要的特征与功能。比如，SQL 允许有“非圆形的圆”以及其他一些毫无意义的概念，参考 20.10 节。（事实上，SQL 根本不支持类型约束。而允许出现那些毫无意义的概念的原因，就是对这种约束的省略，见 20.10 节）。

**3. 解决办法**

小结一下讨论到现在的情况，发现我们面临以下的难题：

- 如果圆从椭圆那里继承了“向 THE_A 和 THE_B 赋值”的操作，则我们会得到非圆形的圆。
- 防止出现非圆形的圆的办法是支持类型约束。
- 但是如果支持类型约束，则上面的操作无法得到继承。
- 所以最后没有继承存在！

怎样解决这一难题？

出路在于——就像经常提起的那样——要认清这样一个事实（并按这一事实去处理问题），即值与变量之间在逻辑上存在着很明显的区别。在说“每个圆都是一个椭圆”的时候，更准确地说，我们的意思是如果一个值是圆，那它也是一个椭圆的值；而不是说每一个圆的变量也是一个椭圆的变量（即一个声明类型为 CIRCLE 的变量不可以是一个声明类型为 ELLIPSE 的变量，而且也不能包含一个确切类型属于 ELLIPSE 的值）。换句话说，**继承只适用于值，不适用于变量**。比如，对于椭圆和圆来说：

- 一个值是圆，那它也是一个椭圆。
- 所有适用于椭圆的、对值的操作也适用于圆。
- 但对于任何值都不能做的事就是改变它！——如果可以改变它，它就不再是原来的那个值了（当然，我们可以通过修改一个变量来“改变变量的当前值”，但是——重申一遍——不能以同样的方式改变一个值）。

现在，准确地说，适用于椭圆的值的操作是针对类型 ELLIPSE 定义的只读操作，而修改 ELLIPSE 变量的操作当然就是针对该类型定义的修改操作。因此，我们所声明的“继承只适用于值，而不适用于变量”，可以作如下更精确的表述：

- **只读操作是由值继承的，因此也毫无疑问可以由变量的当前值继承**（这是因为作为变量当前值的那些值可以毫无妨碍地适用于只读操作）。

这表述同时也可以解释，为什么多态性和可置换性的概念是特别针对值而不是针对变量的。比如（只是为了提醒大家），无论系统在何处需要一个类型 *T* 的值，我们总可以用一个类型 *T'* 的值来置换，其中 *T'* 是 *T* 的子类型。实际上，我们是在引入值的可置换性原则的时候提到了这一原则。

那么对于修改操作又如何呢？由定义可知，这些操作适用于变量而不适用于值。那么我们是否可以说适用于 ELLIPSE 变量的修改操作会被 CIRCLE 变量继承呢？

答案是不完全可以。比如，对 THE_CTR 的赋值对于两种类型的变量都适用，但是（就像我们已经看到的），对于 THE_A 的赋值就不可以。这样一来，对修改操作的继承就是有条件的。事实上，必须明确地指出哪些修改操作是被继承的。比如：

- 类型 ELLIPSE 的变量具有修改操作 MOVE（修改版本）和对 THE_A、THE_B 以及 THE_

CTR 的赋值操作。

- 类型 CIRCLE 的变量具有修改操作 MOVE（修改版本）和对 THE_CTR 以及 THE_R 而不是 THE_A、THE_B 的赋值操作。

注意：MOVE 操作是在上一节讨论的。

当然，如果一个修改操作是通过继承得到的，我们就有了不仅仅适用于值而且适用于变量的多态性和可置换性。比如，修改版本的 MOVE 需要一个类型为 ELLIPSE 的变量做参数，但是在调用它的时候，也可以用一个类型为 CIRCLE 的变量做参数。这样我们可以（而且确实是在）合理地探讨变量的可置换性原则——但是这一原则比上面讨论的值的可置换性原则要有更多的限制。

## 20.9　再论约束特化

对于前面几节的讨论，我们还需要加上一个很短但却是非常重要的后记。后记涉及这样的例子："设 CIRCLE 类型有一个叫做 COLORED_CIRCLE（有颜色的圆）的真子类型"（意思是说"有颜色的圆"被认为是圆的一种特殊情况）。具有这种一般性质的例子在各种著作中经常被引用。只是我们还是要说这种例子并不是能令人信服的，甚至在某些重要的方面，还会产生误导。具体说来：在我们所探讨的情况中，认为有颜色的圆是圆的某种特殊情况实际上是毫无意义的。毕竟有颜色的圆肯定是作为一个图像来定义的，可能是在一个显示屏上，而普遍意义上的圆是几何图形而不是图像。因此认为 COLORED_CIRCLE 是一个完全独立的类型，而不是 CIRCLE 的子类型看起来也许更合理[⊖]。这个单独的类型也许有这样一种可能的表示形式，它有一个分量的类型为 CIRCLE，另一个的类型为 COLOR，但是它不是 CIRCLE 的子类型。

### 1. 继承可能的表示形式

下面有一个非常有力的理由支持上面的立场。首先，回到我们常用的关于椭圆和圆的例子。这里再次给出它们的类型定义（仅给出要点）：

```
TYPE ELLIPSE ...
     POSSREP { A ..., B ..., CTR ... } ;

TYPE CIRCLE ...
     POSSREP { R ..., CTR ... } ;
```

尤其要注意的是，椭圆和圆所声明的可能表示形式是不同的。但是，椭圆的可能表示形式也是（即使是隐含地，也必然是）圆的可能表示形式，因为圆是椭圆。自然地，圆通过其 $a$、$b$ 半轴（以及其中心）"来表示是可能的"，即使事实上其 $a$、$b$ 半轴是一样的。当然，反之不成立——即圆的一种可能的表示形式未必是椭圆的一种表示形式。

由此我们可以认为，类似操作和约束这样的可能表示形式，是圆可以从椭圆继承到的进一步"属性"，或者更一般地认为是子类型可以从超类型继承到的进一步"属性"[⊖]。但是（当我们转到圆和有颜色的圆的情况时），很显然声明用于类型 CIRCLE 的可能表示形式并不能作为类型 COLORED_CIRCLE 的可能表示形式，因为其中没有可以表示颜色的部分！这一事实有力地说明了，有颜色的圆不是圆跟圆是椭圆是一样的道理的。

### 2. 子类型到底意味着什么

下面要讨论的内容与前一个问题有（某些）联系，但是它的结论更强（即逻辑上更强）。结论是：通过约束特化无法从圆获得一个有颜色的圆。

---

⊖ COLORED_CIRCLE 是 CIRCLE 的一个子类型与说它是 COLOR 的一个子类型是等价的（具体是谁的子类型无所谓）。

⊖ 在形式模型中我们并不这样认为，也就是说，我们不考虑将这种继承的可能表示作为声明表示。因为如果将他们作为声明表示会导致冲突。具体说就是，如果我们说类型 CIRCLE 从类型 ELLIPSE 继承了一个可能的表示，那么参考文献［3.3］会要求对于一个 CIRCLE 变量的 THE_A、THE_B 的赋值是合法的，当然我们知道这是不可以的。因此说类型 CIRCLE 从类型 ELLIPSE 继承一个可能表示只是一种口头的说法，但这种说法并不规范。

为了解释这个结论，我们先回到椭圆和圆的情况。再次给出类型的定义：

```
TYPE ELLIPSE
     IS PLANE_FIGURE
        POSSREP { A LENGTH, B LENGTH, CTR POINT
                  CONSTRAINT A ≥ B } ;

TYPE CIRCLE
    IS ELLIPSE
       CONSTRAINT THE_A ( ELLIPSE ) = THE_B ( ELLIPSE )
       POSSREP { R   = THE_A ( ELLIPSE ) ,
                 CTR = THE_CTR ( ELLIPSE ) } ;
```

就像我们在前面看到的，类型 CIRCLE 的 CONSTRAINT 子句保证了一个满足 $a = b$ 的椭圆会自动地被限定为 CIRCLE 类型。但是现在回到圆和有颜色的圆的情况：对于类型 COLORED_CIRCLE，我们无法给出任何类似的 CONSTRAINT 子句，来把一个圆限定为一个有颜色的圆——即我们无法给出任何类型约束，如果一个给定的圆满足这些约束，则意味着这个圆实际上是一个有颜色的圆。

因此，这再一次说明，认为 COLORED_CIRCLE 和 CIRCLE 是完全不同的类型更合理；同时特别地认为类型 COLORED_CIRCLE 有这样一种可能的表示形式，即它的一个分量是 CIRCLE 类型的，而另一个是 COLOR 类型的：

```
TYPE COLORED_CIRCLE POSSREP { CIR CIRCLE, COL COLOR } ... ;
```

事实上，我们在这里碰到了一个更大的问题。事实是，我们相信子类型总是通过约束特化得到的！也就是说，我们认为**如果 $T'$ 是 $T$ 的子类型，则总会有一个这样的类型约束，如果类型 $T$ 的值满足了这个约束，则这个值实际上是类型 $T'$ 的一个值**（而且应该自动地被限定为类型 $T'$）。假设 $T$ 和 $T'$ 表示两个类型，$T'$ 是 $T$ 的子类型（实际上可以不失一般性地假设 $T'$ 是 $T$ 的直接子类型），则：

- $T$ 和 $T'$ 都是基本集合（是值的集合），同时 $T'$ 是 $T$ 的子集。
- $T$ 和 $T'$ 都有成员谓词。即当且仅当满足这一谓词的时候，一个值才是上述集合的成员（因此也就是相应类型的成员）。设这些谓词分别为 $P$ 和 $P'$。
- 现在注意到，根据定义，谓词 $P'$ 在某个值是属于类型 $T$ 的值时取真值。这样，它实际上可以写成是针对属于 $T$ 的值的公式（要好于针对 $T'$ 的值）。
- 用针对 $T$ 的值的公式表达的谓词 $P'$ 恰好是一个类型约束，一个属于 $T$ 的值要成为一个属于 $T'$ 的值就必须满足谓词 $P'$。

因此，约束特化是唯一在概念上定义子类型的有效方法。作为一个推论，我们拒绝接受类似这样的例子，即认为 COLORED_CIRCLE 是 CIRCLE 的一个子类型。

## 20.10 SQL 的支持

SQL 的显式继承仅限于对“结构化类型的”单一继承的支持，对产生类型没有显式继承，对多重继承也没有显式支持，对于所有的 built-in 类型和 DISTINCT 类型也没有继承支持[⊖]。

这里列出定义结构化类型的语法：

```
CREATE TYPE <type name>
     [ UNDER <type name> ]
     [ AS <representation> ]
     [ [ NOT ] INSTANTIABLE ]
       NOT FINAL
     [ <method specification commalist> ] ;
```

下面是 SQL 中用于定义 PLANE_FIGURE，ELLIPSE 和 CIRCLE 可能的例子：

---

⊖ 虽然这些注释说明了 SQL 对生成类型的继承和多重继承有一些潜在的支持。（正如参考文献 [3.3] 所提到的，参考文献里说的更广泛一些。）本章中，我们只将注意力集中于单一继承和标量类型。

```
CREATE TYPE PLANE_FIGURE
       NOT INSTANTIABLE
       NOT FINAL ;

CREATE TYPE ELLIPSE UNDER PLANE_FIGURE
  AS ( A LENGTH, B LENGTH, CTR POINT )
       INSTANTIABLE
       NOT FINAL ;

CREATE TYPE CIRCLE UNDER ELLIPSE
  AS ( R LENGTH )
       INSTANTIABLE
       NOT FINAL ;
```

这里要重复在第5章中的一些要点：

1）NOT INSTANTIABLE 意味着所讨论的类型没有“实例”，这里的实例表示一个值，它的最确切类型是这个被讨论的类型[⊖]。换句话说就是，这个被讨论的类型就是我们所说的合并类型。INSTANTIABLE 意味着被讨论的类型至少有一个“实例”；也就是说它不是一个合并类型，并且至少有一个值的最确切类型是这个被讨论的类型。在我们的例子中，类型 PLANE_FIGURE 是 NOT INSTANTIABLE，而类型 ELLIPSE 和 CIRCLE 则是 INSTANTIABLE 的（默认为 INSTANTIABLE）。

2）正如第5章所讨论的那样，NOT FINAL 必须是被指定的（虽然 SQL：2003 可能允许选择指定为 FINAL）。NOT FINAL 意味着被讨论的类型拥有合适的子类型；如果支持 FINAL 的话，意思正好相反。

3）UNDER 用于识别这个类型的紧接的超类型（SQL 术语中称为直接超类型）。因此，举个例子来说，CIRCLE 是 ELLIPSE 的“直接子类型”，那么特殊的圆可以无条件的继承椭圆中的通用属性，但是要注意：

a）这里的“属性”不表示操作和约束（和它在我们的继承模型中不同）。属性表示操作和结构（或者说是表现）。换句话说，SQL 支持行为继承和结构继承，因为“结构化类型”的内部结构是显式地展示给用户的。参见第5点。

b）这里的“操作”不仅仅代表只读操作（和它在我们的继承模型中不同），它代表所有的操作。也就是说，SQL 并没有完全的区分开值和变量，对更新操作也必须像对只读操作一样需要无条件的继承。于是结果变成圆不一定是圆形的，正方形也不一定是正方形的等等。（为了更深入地探讨这点，在我们的模型中，如果某些值 $v$ 的最确切类型是 ELLIPSE，那么它一定是椭圆而不是圆，如果它的最确切类型是 CIRCLE，那么它一定是一个圆的椭圆，与现实相符。但在 SQL 中，正好相反，如果 $v$ 的最确切类型是 ELLIPSE，那么它实际上可能是一个圆；如果它的最确切类型是 CIRCLE，那么它可能实际上是一个椭圆而不是一个圆，与现实情况并不相符。）

c）讨论的操作可分为三类：函数、过程和方法。正如第5章所说的，函数和过程大致对应于我们的只读和更新操作；方法比较接近函数，不过是用不同的语法规则来激活的。函数和过程分别通过 CREATE FUNCTION 和 CREATE PROCEDURE 语句来指定，而方法则嵌入在 CREATE TYPE 语句的内部（为了简单起见，我们在我们的例子中忽略方法规范说明）。编译时绑定（就是说，只绑定在声明类型的基础上）运用于函数和过程；运行时绑定应用于方法，但它是以只包含一个参数为基础的（对象系统中有典型的例子——参见第25章）。

4）SQL 术语中的根类型是最大超类型；因此，我们例子中的 PLANE_FIGURE 是一个最大超类型。（奇怪的是，在 SQL 术语中叶类型并不是最小子类型而就是叶类型，所以在我们的例子中 CIRCLE 是一个叶类型。）

5）如果指定 < representation >，那么圆括号中就包含 < attribute definition commalist >，这里的 < attribute > 是由 < attribute name > 后面跟着 < type name > 组成的。但是要注意这里的 < representation > 是被讨论类型的值的一个物理表达式——不是所有的表达式（那些展示给用户的物理

⊖ 参考文献［4.23］定义了一个“值的物理表示的”实例（?）。

表达式已经在第3点中提出来了)。特别要注意的是,为相同类型指定两个或更多不同的 < representation > 是不可能的。注意:正如我们在第5章所提到的,类型设计者可以通过明智的选择和操作设计来有效地隐瞒 < representation > 是物理表达式的事实。

6) 每个 < attribute > 都有一个观察方法和一个增变方法,这些是自动提供的,一起使用可以实现与 **Tutorial D** 中的 THE_操作相似的功能(参见第5章中的一些例子)。系统并没有自动提供选择操作,但是每个类型都自动配备一个构造函数,在激活的时候为每个属性配上一个合适的默认值并返回唯一的类型值——正如我们在第5章中所看到的,如果它是用户定义的类型,那么它的所有属性都必须是空值。因此表达式

```
ELLIPSE ()
```

返回的是“椭圆”,它的A和B均为“空长”,CTR为“空点”(当然与点的X和Y部分均为空并不矛盾)。而表达式

```
ELLIPSE () . A   ( LENGTH () . L ( 4.0 ) )
           . B   ( LENGTH () . L ( 3.0 ) )
           . CTR ( POINT ()  . X ( 0.0 ) . Y ( 0.0 ) )
```

返回 $a$ 为4、$b$ 为3、中心为原点的椭圆。(我们已经假设类型LENGTH是只含有一个属性的表达式,L是浮点型。)

7) 我们发现无法确定类型和合法值集合。也就是说,除了物理表达式所隐含的约束,不存在类型约束。SQL不支持约束继承——这些在第3点中已经提到了。

8) 每个结构化类型都有一个相关参照类型。我们在本章不讨论参照类型,但是要注意SQL所支持的参照类型包括“子表和超表”。我们将在第26章继续讨论这个问题。

接下来我们再详细地讨论SQL对类型继承的支持。首先要注意的是SQL与我们的参照的模型相似(至少是应用于单一继承的模型),都依赖于独立假设和最确切类型唯一的假设。SQL也支持置换性(虽然它不能很好地区分值和变量,也无法区分只读和更新操作,甚至不能很好地区分值和变量之间的置换性),但SQL并不使用多态这个术语。

SQL支持与我们的TREAT DOWN和类型测试操作对应的操作。例如:

- SQL中的TREAT (E AS CIRCLE) 同TREAT_DOWN_AS_CIRCLE (E) 类似;
- SQL中的TYPE (E) IS OF (CIRCLE) 同IS_CIRCLE (E) 类似;
- SQL中的

```
R : IS_CIRCLE ( E )
```

是

```
SELECT TREAT ( E AS CIRCLE ) AS E, F, G, ..., H
FROM   R
WHERE  TYPE ( E ) IS OF ( CIRCLE )
```

(F,G,…,H是从E中分离出来的R的所有属性。)

既然SQL不支持类型约束,显然它也不会支持约束的特化和泛化。但是要注意,这并不意味着一个变量的最确切类型不能发生变化。例如:

```
DECLARE E ELLIPSE ;

SET E = CX ;
SET E = EX ;
```

这里的CX和EX是表达式,其返回值的最确切类型分别是CIRCLE和ELLIPSE。因此,在第一次分配之后,变量E(其声明类型为ELLIPSE)有最确切类型CIRCLE;第二次之后,它的最确切类型是ELLIPSE。但是要注意,变量的最确切类型的变化不会影响到约束的特化和泛化。

最后,回想一下第5章,一个给定的结构化类型并不需要有一个关联的“=”操作——即使它有这个操作,这个操作的语义也完全是由用户定义的。事实上,SQL并不要求比较符的最确切类型相同使得比较给出的值是TRUE!如果包含任意一个结构化类型,那么就无法保证SQL

能很好地支持连接、并、交和差。注意，这个标准并没有考虑是否存在类型继承。

**继承还是授权**

我们必须承认在这一小节的讨论可能会造成某些重要方面的误导。事实是，SQL 的类型继承机制（不同于我们自己的继承模型）几乎不支持通过约束超类型来获得子类型的思想。我们再来看看椭圆和圆的例子：

```
CREATE TYPE ELLIPSE UNDER PLANE_FIGURE
  AS ( A LENGTH, B LENGTH, CTR POINT ) ... ;

CREATE TYPE CIRCLE UNDER ELLIPSE
  AS ( R LENGTH ) ... ;
```

根据这些定义，类型 CIRCLE 有属性 *A*、*B*、*CTR*（继承自类型 ELLIPSE）以及 *R*（CIRCLE 类型特有的）。如果用指定的属性组成物理表达式，那么任意给定的圆将是四个值的集合，其中三个均相同。因为这样，很可能类型 CIRCLE 的定义不会指定任何它自己的 < representation >，而是简单地继承类型 ELLIPSE 的表达式。另一方面，如果类型 CIRCLE 的表达式没有 *R*（半径）元素，那么就不会自动为半径提供观察和增变方法。第三点……如果表达式有 *R* 的部分，增变 *R*，也就是卷起一个“不圆的圆”——也就是说一个“圆”的 *A*、*B*、*R* 的值并不完全相同。

基于各种原因，可能有人会认为“椭圆和圆”并不是说明 SQL 类型继承功能的最佳例子。但是 SQL 无法很好地处理这个例子却是事实。我们换一个不同的例子来说明：

```
CREATE TYPE CIRCLE
  AS ( R LENGTH, CTR POINT )
       INSTANTIABLE
       NOT FINAL ;

CREATE TYPE COLORED_CIRCLE UNDER CIRCLE
  AS ( COL COLOR )
       INSTANTIABLE
       NOT FINAL ;
```

这个例子是我们之前不十分推崇的一个例子，在 20.9 节中，我们认为“从圆是椭圆的意义上来讲，有颜色的圆不是圆”。但是如果我们谈论的是继承、可能的扩展以及表达式，那么这个例子会更有意义，认为着色圆是由半径、圆心和颜色来表示，这是合理的。如果我们说类型 COLORED_CIRCLE 处于类型 CIRCLE“之下”，那么认为对普通圆的操作——例如获得半径的操作——应用于特殊的着色圆上（着色圆可以被圆代替）也是合理的。但是认为着色圆是一个普通圆的“约束形式”，或者说着色圆是通过对圆的约束特化得到的就没有意义了。换句话说，SQL 继承机制并不是用于处理本章前面所定义的那些术语，而是用于处理一些作者所说的授权。授权的意思是实现与类型有关某个操作的责任被授予该类型表达式中的某些组成部分的类型（例如，对着色圆的“获取半径”的操作是通过调用在相应的圆上“获取半径”来实现的）。那么称 SQL 的这个机制为授权机制要比禁止它与子类型有任何关系来得更清楚一些。

## 20.11　小结

本章已经简要地介绍了**类型继承模型**的基本概念。如果类型 *B* 是类型 *A* 的子类型（等价地，类型 *A* 是类型 *B* 的超类型），则每个属于类型 *B* 的值也属于类型 *A*，因此适用于类型 *A* 的值的操作和约束也同样适用于属于 *B* 的值（也会存在适用于属于 *B* 的值的操作和约束，但是并不适用于只属于 *A* 的值）。事实上，现在我们可以说得更具体一些，如果 *B* 是 *A* 的真子类型，那么：

- *B* 的值集是 *A* 的值集的一个真子集。
- *B* 有一个 *A* 上约束的真超集。
- *B* 有一个 *A* 上只读操作的真超集。
- *B* 有一个 *A* 上更新操作的真子集（但是它本该有自己额外的更新操作，这些更新操作不用于 *A*）。

我们区分了**单一继承**和**多重继承**（但是只讨论了单一继承），还有**标量继承**、**元组继承**和**关**

**系继承**（但是只讨论了标量继承）[⊖]，同时引入了**类型层次结构**的概念，还定义了**真子类型**和**真超类型**、**直接子类型**和**直接超类型**、**根类型**和**叶类型**。同时我们描述了**不相交假设**：类型 *T*1 与 *T*2 是不相交的，除非一个是另一个的子类型。作为这个假设的一个推论，我们说每个值只有一个唯一的**最确切类型**（但是它不必是叶类型）。

接下来，讨论了（包含）**多态性**和（值的）**可置换性**的概念，它们都是基本的继承概念的逻辑推论。区分了**包含多态**（与继承有关）和**重载多态**（与继承无关），还展示了包含多态是如何带来**代码重用**的——借助于**动态联编**。

然后又考虑了继承对**赋值**操作的影响。基本的想法是不发生类型转换——值在被赋给一个非确切类型的变量时可以保持其最确切类型，因此一个声明类型为 *T* 的变量可以有一个最确切类型是 *T* 的任意子类型的值（同样地，如果定义操作 *Op* 的结果的声明类型为 *T*，则调用 *Op* 所得到的实际结果的值的最确切类型可以是 *T* 的任意子类型）。因此我们模型化一个标量变量 *V*（或者更一般地，一个任意的标量表达式）成为一个形如 < *DT*,*MST*,*v* > 的有序元组。其中 *DT* 是声明类型，*MST* 是当前最确切类型，*v* 是当前值。我们引入了 *TREAT DOWN*（类型下移）操作，使得可以对这样的表达式进行操作，这些表达式在运行时候的最确切类型是其声明类型的真子类型。否则类型下移操作会在编译的时候报出类型错误（运行时的类型错误也会出现，但是错误只会出在 TREAT DOWN 操作内）。

接着更详细地探讨了**选择子操作**。调用类型 *T* 的一个选择子操作时，有时候会产生一个属于 *T* 的某个真子类型的结果（至少在我们的模型里是这样，虽然在今天的商用产品中并不是这样）：**约束特化**。然后又比较详细地讨论了 **THE_伪变量**。既然它们只是一些缩写，则约束特化和约束**概括**在对 THE_伪变量进行赋值时都会出现。

然后开始讨论子类型和超类型对**等值比较**以及一些关系操作（**连接**、**并**、**交**、**差**）的影响。我们也引入了一些**类型检测**操作：**IS_T** 等。其后考虑了关于**只读操作**和**修改操作**、操作**版本**、操作**签名**的问题，并指出为一个操作定义不同版本的能力使得对该操作**改变语义**成为可能（但是在我们的模型中禁止这种改变）。

之后，我们考查了“圆真的是椭圆吗?”这样一个问题。通过考查确定了这样一个立场：即**继承只适用于值，而不适用于变量**。更准确地说就是，**只读操作**（只适用于值）可以百分之百地得到继承而毫无问题，但是修改操作（适用于变量）只能**有条件地**得到继承（我们的模型在这方面与大部分其他的方法不一致。那些方法通常要求修改操作可以被无条件地继承，但是使用这些方法接着就会被各种问题所困扰，这些问题自然与所谓“非圆形的圆”等类似的说法有关）。于是据此得出结论：从我们的立场上，约束特化是在逻辑上定义子类型的唯一有效方法。

最后简单讨论了**授权**的概念。这个概念虽然与继承相关，但是在逻辑上还有一定的距离（它同样也是以代码重用为目的）。我们简单的介绍了 SQL 继承机制，包括真正用于解决授权问题而不是继承问题的机制。我们将在第 26 章更加详细的介绍符合 SQL 风格的继承。

## 习题

20.1　用自己的语言给出下列词语的定义：

| | | | |
|---|---|---|---|
| 代码重用 | 真子类型 | 授权 | 根类型 |
| 约束泛化 | 动态联编 | 直接子类型 | 签名 |
| 继承 | 约束特化 | 叶类型 | 可置换性 |
| 多态性 | 合并类型 | | |

20.2　解释 TREAT DOWN（类型下移）操作。

20.3　区分下列概念：

| | |
|---|---|
| 参数与参量 | 声明类型与当前最确切类型 |
| 包含多态与重载多态 | 调用签名、描述签名与版本签名 |

---

⊖ 我们至少可以说，本章中所讨论的单一继承和标量继承的思想可以很方便地扩展到多重继承、元组继承和关系继承的概念上，正如参考文献 [3.3] 所论述的。

只读操作与修改操作　　值与变量

（最后两个请参考练习 5.2）

20.4　参照图 20-1 的类型层次结构，有一个属于类型 ELLIPSE 的值 *e*。*e* 的最确切类型是 ELLIPSE 或 CIRCLE，那么 *e* 的最不确切类型是什么？

20.5　任意给定的类型层次结构包括很多子层次结构，这些子层次结构仅凭自身也可以被认为是完整的类型层次结构。比如，对于从图 20-1 中（只）删除了类型 PLANE_FIGURE、ELLIPSE 和 CIRCLE 得到的层次结构而言，凭借该层次结构自身的特征也可以认为是一个类型层次结构。同样地，（只）删除了类型 CIRCLE、SQUARE 和 RECTANGLE 之后所得到的也是一样。但是（只）删除 ELLIPSE 后得到的层次结构凭借其自身的特征就不能认为是一个类型层次结构（至少没有人可以从图 20-1 中派生出它来），因为类型 CIRCLE“丢失了它的一些继承特征”，而这是它在那个层次结构中本应该有的。那么在图 20-1 中总共有多少个不同的类型层次结构呢？

20.6　利用本章给出的简要语法，给出类型 RECTANGLE 和 SQUARE 的类型定义（为了简单起见，假设所有矩形的中心都在原点，但是并不假设所有的边都是垂直或水平的）。

20.7　根据你对练习 20.6 的答案，定义一个操作，可以使一个特定的矩形绕它的中心旋转 90°，还要给出该操作针对正方形的显式特化。

20.8　下面引用 20.6 节的一个例子：“关系变量 R 有一个声明类型为 ELLIPSE 的属性 A，我们希望通过查询 R 得到这样的元组，即 A 的值实际上是一个圆而且其半径大于 2。”下面是在 20.6 节给出的这个查询的公式：

```
R : IS_CIRCLE ( A ) WHERE THE_R ( A ) > LENGTH ( 2.0 )
```

a. 为什么我们不能简单地在 WHERE 子句中表达类型检查条件？比如：

```
R WHERE IS_CIRCLE ( A ) AND THE_R ( A ) > LENGTH ( 2.0 )
```

b. 另一个侯选的公式是：

```
R WHERE CASE
           WHEN IS_CIRCLE ( A ) THEN
                THE_R ( TREAT_DOWN_AS_CIRCLE ( A ) )
                                  > LENGTH ( 2.0 )
           WHEN NOT ( IS_CIRCLE ( A ) ) THEN FALSE
      END CASE
```

这个公式会有效吗？如果不会，为什么？

20.9　参考文献［3.3］建议支持这样的关系表达式

```
R TREAT_DOWN_AS_T ( A )
```

这里 *R* 是一个关系表达式，*A* 是该表达式所代表的关系（比如说叫 *r*）的一个属性，*T* 是一个类型。*A* 的声明类型 *DT*（*A*）必定是 *T* 的超类型（这是编译时做的检查）。整个表达式的值被定义为一个关系，而且：

a. 其属性名称与 *r* 一样，除了其中属性 *A* 的声明类型为 *T*；

b. 其内容包含与 *r* 一样的元组，除了元组中属性 *A* 的值被下移成了 *T*。

但是，这个操作还是一个缩写。为什么？请具体说明。

20.10　形如 R：IS_*T*（*A*）的表达式同样是一个缩写。为什么？请具体说明。

20.11　SQL 支持构造函数而不是选择器，两者的区别是什么？

20.12　为什么你认为 SQL 不能支持类型限定？如何进行特殊的类型限定？

## 参考文献

这里没有进一步详细描述继承模型唯一的重要变化，即在本章所述的对多继承的支持。首先，仅仅要求根类型是分离集合；其次，重新定义 most specific type 为：每个集合类型 *T*1，*T*2，…，*Tn*（$n \geq 0$）必须有一个通用子类型 *T*'，以便当给定值是 *T*'类型时，该值属于 *T*1，*T*2，…，*Tn* 类型，见参考文献［3.3］对这些问题的进一步的详细讨论，以及为了支持元组和关系继承的扩展要求。

［20.1］　Alphora: *Dataphor™ Product Documentation.* Available from Alphora, 2474 North University Avenue, Provo, Utah 84604（参见 *http://www.alphora.com*）.

Dataphor 是一个商业产品，支持本章所述的继承模型的一个大的子集（以及 The Third Manifesto [3.3] 中所述的一个大的子集）。

[20.2] Malcolm Atkinson *et al.*: "The Object-Oriented Database System Manifesto," Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N. Y.: Elsevier Science (1990).

关于一个好的继承模型缺乏一致性的问题（在 20.1 节提到过），这篇论文的作者这样说："至少存在四种类型的继承：置换继承、包含继承、约束继承和特定继承……现有系统和原型各自不同程度地提供了这四种类型的继承，而我们并不规定一种特别形式的继承。"

这里还有一些其他的引文表达了同样的观点：

Cleaveland [20.5] 中提到："[继承可以] 基于 [各种] 不同的标准，而且并不存在（被人们）普遍接受的标准定义"。并且对此给出了 8 种合理的解释。(Meyer [20.11] 给出了 12 种)。

Baclawski 和 Indurkhya [20.3] 说："[一种] 编程语言 [只是] 提供一组 [继承] 机制。这些机制显然限定了用这种语言可以做什么，以及可以实现什么样的继承视图……它们自身并不能使这样或那样的继承视图生效。类、特化、泛化和继承性只是概念，而且……它们没有一般意义上的客观含义……这 [一事实] 暗示继承性如何并入一个特定的系统取决于 [这个] 系统的设计者，这就成了一种要由有效机制来实现的策略选择了。" 换句话说，根本就没有模型！

然而，本章我们已经清楚的描述过，我们不同意以上的结论。

注意：此文章将在第 25 章第 1 节中作为参考文献 [25.1] 再次出现，在那里可以找到进一步的解释。

[20.3] Kenneth Baclawski and Bipin Indurkhya: Technical Correspondence, *CACM 37*, No. 9 (September 1994).

[20.4] Luca Cardelli and Peter Wegner: "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Comp. Surv. 17*, No. 4 (December 1985).

[20.5] J. Craig Cleaveland: *An Introduction to Data Types*. Reading, Mass.: Addison-Wesley (1986).

[20.6] C. J. Date: "Is a Circle an Ellipse?" *http://www.dbdebunk.com* (July 2001).

工业界对于本文标题中提到的问题持否定答案。本文引述该方面的一些权威试图扭转他们的观点。注意：本书初版时，曾引起巨大的在线评论和批评，这些都可以从 http://www.dbedunk.com 上获取。

[20.7] C. J. Date: "What Does Substitutability Really Mean?" *http://www.dbdebunk.com* (July 2002).

具体的分析与评论见参考文献 [20.9]。

[20.8] You-Chin Fuh *et al.*: "Implementation of SQL3 Structured Types with Inheritance and Value Substitutability," Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland (September 1999).

应用摘要中的一部分："本文讲述了 DB2 方法…首先，结构化类型的值以自描述的形式出现，并且仅仅通过系统产生的 observer 或者 mutator 方法进行操纵，以便使操作对底层存储管理器的影响最小化；第二，基于值的增变语义通过编译时的拷贝避免算法来有效实现；第三，结构化类型的值是在线或者离线动态存储"。注意："基于值的增变语义" 指 SQL 的增变实际上是只读的操作符。（在目标上执行增变将产生 "突变" 效应，也就是说，T 的赋值结果依然会返回给 T。）

[20.9] Barbara Liskov and Jeannette Wing: "A Behavioral Notion of Subtyping," *ACM TOPLAS (Transactions on Programming Languages and Systems) 16*, No. 6 (November 1994).

在很多参考文献中置换指的是 Liskov 置换原则（LSP）。该文被认为是这个原理的源头。

[20.10] Nelson Mattos and Linda G. DeMichiel: "Recent Design Trade-Offs in SQL3," *ACM SIGMOD Record 23*, No. 4 (December 1994).

该文给出了决策时 SQL 语言设计人员不支持类型限定的原理（这是基于早期 Zdonik 和 Maier 在参考文献 [20.14] 中的讨论）。我们不同意那个原理，基本问题在于它没有合理的区分开值和变量。

[20.11] Bertrand Meyer: "The Many Faces of Inheritance: A Taxonomy of Taxonomy," *IEEE Computer 29*, No. 5 (May 1996).

[20.12] James Rumbaugh: "A Matter of Intent: How to Define Subclasses," *Journal of Object Oriented Programming* (September 1996).

在 20.9 节中提到，我们的观点是：约束特化是在逻辑上定义子类型的唯一有效方法。因此，可以注意到这样一件非常有趣的事情，对象世界恰恰是站在完全相反的立场上！或至少是那个世界中的某些人是这样做的。用 Rumbaugh 的话说："SQUARE 是 RECTANGLE 子类吗？……拉伸矩形的 x 轴是一件很正常的事情，但是，如果你对一个正方形做同样的操作，则它就再也不是一个正方形了。概念上，这并不是一件坏事。当你拉伸一个正方形的时候，你得到了一个矩形……但是……大多数面向对象的语言并不希望对象改变它们所属的类……所有这些都暗示了（一条）分类系统的设计原则：子类不能通过对父类的约束来定义。注意：像在第 24 章所解释的，对象世界中类这个词所表达的意思经常和我们所说的类型所表达的意思是一样的。

我们惊奇地发现，Rumbaugh 站在这个立场上显然只是因为面向对象语言"不希望对象改变它们所属的类"。而我们宁愿首先保证模型的正确性，然后再考虑它的实现问题。（任何情况下，我们都不知道如何有效实现限制的细节，参考文献 [3.3] 中记录了一些这方面的思想）。

[20.13] Andrew Taivalsaari: "On the Notion of Inheritance," *ACM Comp. Surv.* 28, No. 3 (September 1996).

[20.14] Stanley B. Zdonik and David Maier: "Fundamentals of Object-Oriented Databases," in reference [25.42].

# 第21章　分布式数据库

## 21.1　引言

我们在第2章结尾提到了分布式数据库的问题，在此引述一下："完整的分布式数据库支持意味着，单个的应用程序应该可以'透明地'操纵数据，这些数据分布在各种不同的数据库中，由不同的DBMS管理，运行在不同的机器上，受不同的操作系统支持，通过各种不同的通信网络连接起来——这里，'透明地'的意思是指从逻辑上看，应用程序操纵数据，就像数据都由运行在同一台机器上的DBMS来管理。"本章将要更细致地讨论这些概念。具体地说，我们将明确地解释什么是分布式数据库，为什么分布式数据库会变得越来越重要（考虑其在万维网中的应用，参见第27章），以及在分布式数据库领域中的一些技术难题是什么。

第2章还简要地讨论了**客户/服务器系统**，可以将该系统看作是一般分布式系统的一个简单的特例。在21.5节我们将专门讨论客户/服务器系统。

本章的安排将在下一节的最后进行说明。

## 21.2　预备知识

我们先从一个指导性的定义开始（目前还不需要很严密）。

分布式数据库系统由一系列的场地组成，通过某种通信网络连接在一起，其中：

a. 每个场地自身都有一个完备的数据库系统，但是

b. 所有的场地都可以协同工作，使得任何场地上的用户都可以访问通信网络上任何地方的数据，就好像数据是存储在用户自己的场地上一样。

由此可见，所谓的"分布式数据库"实际上是一种虚拟的数据库，它的各个组成部分物理地存储在许多不同场地上的不同的"真实"数据库中（从效果上讲，它是这些真实数据库逻辑上的并集）。图21-1中给出了一个例子。

请再一次注意，每个场地自身都有一个数据库系统。换句话说，每个场地有其本地的"真实"数据库、本地的用户、本地的DBMS和事务管理软件系统（包括本地的封锁、日志、恢复等软件系统），还有本地的数据通信管理器（DC管理器）。尤其要说明的是，一个用户可以操作本地场地上的数据，而完全感觉不到本地场地参与了一个分布式系统（至少这是一个目标）。因此，可以认为分布式数据库系统是独立场地上的独立的DBMS之间形成的一种合作关系。由位于每个场地上的一个新的软件模块（该模块在逻辑上是本地DBMS的扩展）来提供所需的合作功能，这个新的模块与已经存在的DBMS一起构成我们所说的**分布式数据库管理系统**。

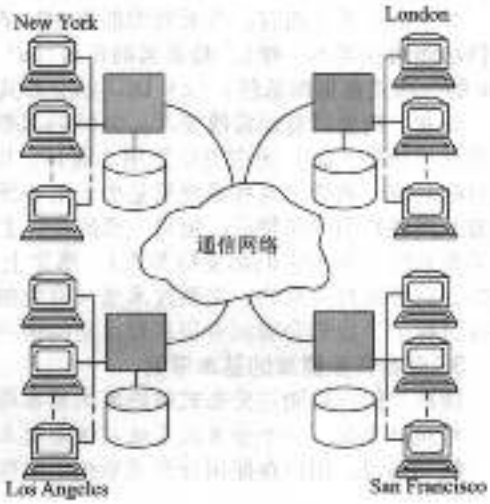


图21-1　一个典型的分布式数据库系统

顺便说一句，通常都假设所有参与场地在物理上是分散的——可能确实在地理上就是分散的，就像在图21-1中所表示的那样，虽然实际上只要它们在逻辑上是分散的就可以了。两个"场地"甚至在物理上可以处在同一台机器中（尤其是在最初的系统测试期间）。其实，分布式系统中所强调的重点是随着时间的推移来回变化的。早期的研究倾向于采用地理的分布，但是最早的几个商业系统采用的却是本地分布，（比如）许

多“场地”都在一幢大楼里，通过局域网（LAN）连接在一起。可是后来，广域网（WAN）的迅速兴起与发展又把兴趣重新拉回到了地理分布的可能性上。但是无论怎样，从数据库的角度来看，这些都没有太大的差别，关键是需要解决的（数据库）技术问题都是一样的，所以为了本章的需要，我们可以合理地认为图 21-1 表示的就是一个典型的系统。

注意：为了叙述简单，除非特别说明，我们假设系统是同构的，也就是说每个场地上运行的 DBMS 都是一样的。我们把这叫做**严格同构假设**。我们将在 21.6 节研究放宽这一假设的可能性。

**1. 优越性**

为什么分布式系统是有价值的？最基本的答案是企业自身经常就已经是分布式的，至少是在逻辑上（被分成分公司、部门、工作组等），而且也非常可能是在物理上（被分成工厂、车间、实验室等）——这就意味着数据通常就已经是分布的了，因为企业中的每个部门都会很自然地维护与自己工作有关的数据。这样，企业的整个信息资产就被分裂成通常所说的信息孤岛（island of information）。而分布式系统所做的就是为把这些小岛联系在一起提供桥梁。换句话说，它使得数据库的结构能够反映企业的结构：本地数据可以保存在本地——它在逻辑上所从属的地方，而同时可以在需要的时候存取异地的数据。

举一个例子将有助于明确上面的论述，现在来看看图 21-1。为了简单起见，假设只有洛杉矶和旧金山两个场地，而系统是银行系统，包括洛杉矶和旧金山各自本地账户的账户数据。那么优越性是显而易见的：分布式的安排结合了**处理的有效性**（数据的存储靠近最经常被用到的地方）与**增强的可存取性**（很可能要通过网络从旧金山存取一个洛杉矶的账号，反之亦然）。

像刚才所说的那样，能够使数据库的结构反映企业的结构，可能是分布式系统最大的优越性。当然还有许多其他的优越性，稍后将在本章合适的地方分别讨论这些优越性。但是我们也应该看到还有一些不利的地方，其中最大的就是分布式系统都比较复杂，至少从技术的观点看是这样。当然在理想情况下，这种复杂性应该是系统实现人员的问题，而不是用户的问题。但是，从实用的角度看，用户很可能会面对复杂性的某些方面，除非采取了非常小心的预防措施。

**2. 典型系统**

为了在方便后面的引用，我们在这里简要地介绍一些比较知名的分布式系统。首先是原型系统。在众多的研究系统中，最著名的三个是（a）SDD-1，这是由美国计算机公司（Computer Corporation of America）的研究部门在 20 世纪 70 年代末期到 80 年代早期实现的［21.32］；（b）R*（“R 星”），这是 System R 原型系统的分布式版本，是在 20 世纪 80 年代早期由 IBM 研究院（IBM Reaserch）实现的［21.37］[⊖]；（c）分布式 Ingres，这是 Ingres 原型系统的分布式版本，同样是在 20 世纪 80 年代早期由加利福尼亚大学伯克利分校实现的［21.34］。

对于商业系统而言，今天的大部分 SQL 产品都提供了某种对分布式数据库的支持（当然所提供功能的程度不一样）。最著名的包括（a）Ingres/Star，Ingres 的分布式数据库组件；（b）Oracle 的分布式数据库选件；（c）DB2 的分布式数据支持。

注意：销售商有经常改变产品名称的习惯，因此我们不能保证以前用过的名称（某些情形下甚至可能是产品）现在仍被使用。而且，以上列举的原型系统和产品显然并不意味着穷尽了所有的系统。列举出这些系统只是由于某些原因而曾经产生过或者正在产生很大的影响，或者是有着某些特别的内在特征。值得一提的是以上所列举的这些系统，无论是原型系统还是产品，都是关系系统（至少它们都支持 SQL）。事实上，有很多理由说明为什么，如果一个分布式系统要成功，它必须是关系的；关系技术是（有效的）分布式技术的一个先决条件［15.6］。在本章的讲述过程中，我们会看到有很多理由支持这一结论。

**3. 分布式数据库的基本原则**

现在我们可以阐述**分布式数据库的基本原则**［21.13］：

*对用户来说，一个分布式系统应该看起来完全像一个非分布式系统。*

换句话说，用户在使用分布式系统的时候，应该完全感觉不到系统是分布的。分布式系统的

---

⊖ 这里的“星”就是所谓的 Kleene 操作符——“R*”表示“0 个或多个［System］R”。

所有问题都应该是内部或实现层次上的问题，而不是外部或用户层次的问题。

注意：上述段落里的“用户”是特指那些执行数据操纵的用户（最终用户或应用程序员）。所有的数据操纵操作在逻辑上应该是不变的。相反，数据定义在分布式系统中则需要进行一些扩展，使得（比如说）一个在场地 *X* 的用户（可能是 DBA）可以把一个基本关系分成不同的“片段”，存储在场地 *Y* 和场地 *Z* 上（见下一节关于分片的讨论）。

上面所说的基本原则会带来一些补充的规则或目标[⊖]，它们总共有 12 个，我们将在下一节对它们进行讨论。为了引用方便，我们把这些目标列在这里：

1）本地自治（local autonomy）
2）不依赖中心场地（no reliance on a central site）
3）可连续操作性（continuous operation）
4）位置独立性（location independence）
5）分片独立性（fragmentation independence）
6）复制独立性（replication independence）
7）分布式查询处理（distributed query processing）
8）分布式事务管理（distributed transaction management）
9）硬件独立性（hardware independence）
10）操作系统独立性（operating system independence）
11）网络独立性（network independence）
12）DBMS 独立性（DBMS independence）

需要了解的是这些目标既不是相互独立的，也肯定不会是毫无遗漏的，而且它们也不可能都是同等重要的（不同的用户在不同的环境中会给予不同的目标以不同的重要程度。事实上，其中的一些目标在某些环境下也许是完全不适用的）。但是，以这些目标为基础，通常可以有助于理解分布式技术；而且以它们为框架，还可以有助于规划一个特定的分布式系统的功能。因此在本章的主要内容中，我们将以它们为组织原则。21.3 节将对每个目标进行简要的讨论；21.4 节将从更为细节的角度着眼于某些特定问题；21.5 节将讨论客户/服务器系统；21.6 节将深入地考查 DBMS 独立性；最后，21.7 节提出对 SQL 的支持问题，而 21.8 节进行了小结并给出了一些结论。

最后要介绍的一点是：区分这样两种系统是非常重要的，一种是真正的、普遍意义上的分布式数据库系统，另一种是只能提供远程数据存取的系统（顺便提一句，所有的客户/服务器系统都能做到这一点）。在一个远程数据存取系统中，用户也许可以操作远程场地的数据，甚至是同时操作许多远程场地上的数据，但是“远程和本地结合的接缝是显而易见的”，用户肯定或多或少地知道数据是在异地的，因而要采取相应的操作。相反，在一个真正的分布式数据库系统中，远程和本地结合的接缝被隐藏起来了。（此处所说的“接缝被隐藏起来”正是本章其余部分里的很多地方所关注的。）在所有下面的讨论中，我们将用“分布式系统”这个术语来特指一个真正的、普遍意义上的分布式数据库系统（区别于一个简单的远程数据存取系统），除非给出明确说明。

## 21.3　十二个目标

### 1. 本地自治

一个分布式系统中的场地应该是自治的。本地自治是指在一个给定场地上的所有操作由这个场地自己控制，场地 *X* 上的成功操作不应该依赖于某个其他的场地 *Y*（除非场地 *Y* 发生停机会导致场地 *X* 也不能运转了，即使场地 *X* 自己没有任何问题——这显然是一种不希望发生的情况）。本地自治还意味着本地数据是由本地拥有和管理的，并具有本地的可计算性；所有的数据都是

⊖ “规则”是参考文献［21.13］里的术语，那篇文章首先引入这些规则（“基本原则”就是规则 0）。不过，术语“目标”比“规则”要更好些——“规则”太教条了。本章我们将采用“目标”这个术语。

“真正”属于某个本地数据库的，即使它们可以被其他远程的场地访问。本地数据的安全性、完整性和存储形式之类的问题也是在本地场地的控制和管辖之下的。

实际上，本地自治的目标是无法完全达到的——在许多情况下，一个给定的场地 *X* 必须交出一定程度的控制权给某个场地 *Y*。因此自治目标也许这样表述更为精确：场地应该在尽可能大的程度上是自治的。可以参看对参考文献［21.13］的注释以获取更多的细节。

**2. 不依赖中心场地**

本地自治意味着所有的**场地都必须得到同等的对待**。因此，尤其是不应该为了某些集中服务——比如，集中查询处理、集中事务管理或者是集中命名服务——而依赖于一个中心“主”场地，以至于整个系统会依赖于一个中心场地。这样第二个目标其实是第一个目标的一个推论（如果第一个满足了，第二个毫无疑问会得到满足）。但是“不依赖中心场地”本身就是非常重要的，即使是完全的本地自治无法达到。因此，把它作为一个单独的目标提出来是很有必要的。

之所以不接受对中心场地的依赖至少是基于以下两个原因：首先，中心场地可能会成为一个瓶颈；其次，也是更重要的，系统将会是脆弱的——如果中心场地停止运转了，整个系统也将停止下来，即出现单点故障（single point of failure）。

**3. 可连续操作性**

通常分布式系统的一个好处是它们可以提供更大的可靠性和更高的可用性：

- 可靠性（即系统可以在任何时刻启动并运行的可能性）是得到证明了的，因为分布式系统不是一个要么做要么不做的系统——在某些单独的部分，比如说单独的场地出现故障的时候，其他部分仍然可以继续操作（当然是在一个被降低的水平上）。
- 可用性（即系统在某个特定的时期内能够启动并始终运行的可能性）也是得到证明了的，一方面是由于可靠性的原因，另一方面是由于分布式系统具有数据复制的可能性（见对第 6 个目标的进一步讨论）。

可靠性与可用性适用于在系统某处发生了意外停机（即出现了某种故障）的情况。意外停机显然是令人讨厌的，但却是难以避免的。与之相对，计划停机应该是永远不需要的，也就是说，应该永远也没有必要停止系统的运行来执行某个任务，比如增加一个新的场地或是在一个现有场地上把 DBMS 升级到新版本之类的任务。

**4. 位置独立性**

**位置独立性**（也叫做位置**透明性**）的基本概念是很简单的：用户不需要知道数据在物理上存储在哪里，但是从逻辑的角度看，用户仍然可以进行操作，就好像数据是存储在用户自己本地的场地上一样。之所以要达到位置独立性是由于它简化了用户程序和终端操作，尤其是它允许数据在场地之间迁移，而不会造成程序和操作的失效。需要这种可迁移性是为了使数据可以在网络上移动以适应改变性能的要求。

*注意*：你一定会意识到这样一点，即位置独立性只是通常的（物理上的）数据独立性概念在分布情况下的扩展。实际上我们将看到，每个列出的目标，只要其中含有“独立性”的字眼，都可以看作是数据独立性的一种扩展。对于位置独立性我们在 21.4 节中还要特别地多说一点（在“目录表管理”小节中）。

**5. 分片独立性**

如果为了物理存储的需要，可以把给定的关系分割成小块或片段，则说这个系统是支持**数据分片**的，并且不同的分片可以存储在不同的场地上。使用分片是出于性能上的考虑：数据可以在最经常被用到的地方存储，这样大部分的操作就会是本地的，而网络开销也会降低。比如，考虑雇员关系 EMP，样本数据在图 21-2 的上半部分。在支持分片的系统中，我们可以定义如下两个分片：

```
FRAGMENT EMP AS
   N_EMP AT SITE 'New York' WHERE DEPT# = DEPT# ('D1')
                            OR    DEPT# = DEPT# ('D3') ,
   L_EMP AT SITE 'London'   WHERE DEPT# = DEPT# ('D2') ;
```

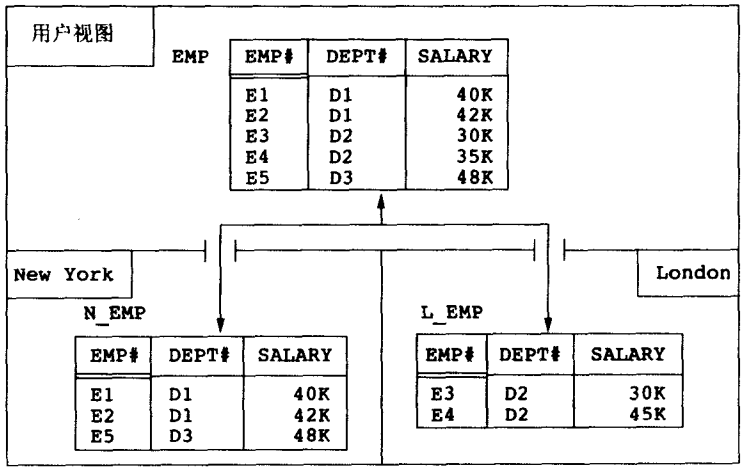



图 21-2　一个分片的示例

(参照图 21-2 的下半部分)。注意：假设 EMP 元组与物理存储之间是以某种直接方式映射的，并且 D1 和 D3 是 New York 的部门，而 D2 是伦敦的部门，因此纽约雇员的元组存储在纽约的场地上，而伦敦雇员的元组存储在伦敦的场地上。系统内部的分片名称是 N_EMP 和 L_EMP。

分片有水平和垂直两种基本的类型，分别对应于关系操作中的选择和投影（图 21-2 给出的是水平分片）。更一般地，一个分片可以由选择操作和投影操作的任意组合来产生——这里的任意不包括下列的情况：

- 对于选择操作而言，所有的选择操作必须构成一个第 13 章所说的正交分解。
- 对于投影操作而言，所有的投影操作必须构成一个第 12、13 章所说的无损分解。

这两条规则的最终影响是一个关系的所有分片将是独立的，意思是说没有任何一个分片，或者是该分片的一个选择或投影，可以从其他分片中产生。注意：如果真的想要在不同的地方存储相同的信息片段，可以用系统的复制机制（见下面第 6 个目标的介绍）。

在分片上对原有关系的重构是通过适当的连接和合并操作来完成的（垂直分片进行连接，水平分片进行合并）。顺便提一句，可以看到在进行合并操作的时候，不需要考虑消除冗余，这要归功于上面两条规则中的第一条，也就是说，这里的合并是**不相交并**（disjoint union）。

在垂直分片的问题上我们要稍微精心一些。确实如上所述，这样的一个分片应该是无损的，因此，比如说对于关系 EMP，按照投影 {EMP#, DEPT#} 和 {SALARY} 进行分片就不是有效的。但是，在某些系统中，认为存储的关系有一个隐藏的、系统提供的“元组 ID”或是 TID 属性，简单地说，一个元组的 TID 就是它的地址。这个 TID 属性显然是这些关系的候选码。比如，如果关系 EMP 包括这样一个属性，则关系按照投影 {TID, EMP#, DEPT#} 和 {TID, SALARY} 进行分片就是有效的，因为显然分片是无损的。同样要注意到这样一个事实，TID 属性是隐藏的这一点与信息原则（The Information Principle）并不矛盾，因为分片独立性（这一点我们一会儿将要进行讨论）意味着用户并不知道分片。

另外，易于分片和易于重构是分布式系统采用关系模式的众多原因中的两个，关系模型提供了完成分片和重构任务所需的合适的操作 [15.6]。

现在我们得出了主要观点：支持数据分片的系统也必定要支持**分片独立性**（也叫做**分片透明性**），即至少在逻辑上，用户的行为应该像数据没有被划分的时候一样。分片独立性（与位置独立性类似）的意义在于它简化了用户程序和终端操作。尤其是，为了满足性能变化的需要，它允许数据在任何时候被重新划分（以及在分片任何时候被重新分布），而无须使上面所说的任何用户程序或操作失效。

分片独立性意味着用户将看到一个数据视图，在这个视图中各个分片是通过合适的连接和合

并逻辑地重新组合在一起的。而为了满足用户的任何请求，系统优化器要负责决定应该物理地访问哪一个分片的数据。对于图 21-2 所给出的分片的例子，如果用户的请求是查询

```
EMP WHERE SALARY > 40K AND DEPT# = DEPT# ('D1')
```

则从分片的定义中（当然该定义存储在目录表里），优化器知道整个结果可以从 New York 的场地上获得，而完全不必访问 London 这一场地。

让我们再仔细地看看这个例子。首先，用户所认识到的关系 EMP 可能被简单地认为是底层分片 N_EMP 和 L_EMP 的一个视图：

```
VAR EMP "VIEW"                                    /* 伪码 */
    N_EMP UNION L_EMP ;
```

则优化器将用户的原始请求转化为如下形式：

```
( N_EMP UNION L_EMP ) WHERE SALARY > 40K
                      AND DEPT# = DEPT# ('D1')
```

这个表达式可以进一步被转化为：

```
( N_EMP WHERE SALARY > 40K AND DEPT# = DEPT# ('D1') )
  UNION
( L_EMP WHERE SALARY > 40K AND DEPT# = DEPT# ('D1') )
```

（因为选择操作对合并服从分配律）。接下来，通过目录表中分片 L_EMP 的定义，优化器知道 UNION操作的第二个操作数等价于：

```
EMP WHERE SALARY > 40K AND
          DEPT# = DEPT# ('D1') AND DEPT# = DEPT# ('D2')
```

这个表达式的计算结果是一个空关系，因为 WHERE 子句中的选择条件永远也不可能为真。这样整个表达式可以简化为：

```
N_EMP WHERE SALARY > 40K AND DEPT# = DEPT# ('D1')
```

现在优化器知道只要访问 New York 的场地就可以了。*习题*：考虑一下在处理如下请求的时候，优化器的工作会涉及哪些方面？

```
EMP WHERE SALARY > 40K
```

像在前面讨论中说到的，支持在分片关系上的操作和支持在连接、合并视图上的操作，这两者所遇到的问题有许多共同点（其实，这两个问题是同一个问题，只是它们表现在整个系统体系结构的不同方面而已）。特别是，对经过分片的关系的修改和对连接、合并视图的修改是一样的（见第 10 章）。这也就同样意味着（只是这样说并不严格），对一个元组的修改可能会使这个元组从一个分片迁移到另一个分片（甚至可能从一个场地到另一个场地），如果被修改的元组不再满足它原来所属分片的分片谓词的话。

**6. 复制独立性**

一个系统是支持**数据复制**的，如果一个给定的关系，更一般地，或一个给定关系的某个分片可以由存储在许多不同场地上的不同的副本或拷贝来表示。比如：

```
REPLICATE N_EMP AS
   LN_EMP AT SITE 'London' ;

REPLICATE L_EMP AS
   NL_EMP AT SITE 'New York' ;
```

（如图 21-3 所示）。系统的内部副本叫做 NL_EMP 和 LN_EMP。

采取复制的理由至少有两点：首先，它可以带来更优越的性能（应用程序可以在本地的副本上进行操作，而不必与远程场地通信）；第二，它还可以带来更高的可用性（一个被复制的对

象总是可以用于处理的——至少可以用于检索——只要至少有一个副本还是可用的）。当然，复制带来的最大的问题是当一个复制对象被修改后，这个对象的所有副本必须进行修改：这就是**更新传播**问题。对于这个问题在 21.4 节将会有更多的讨论。

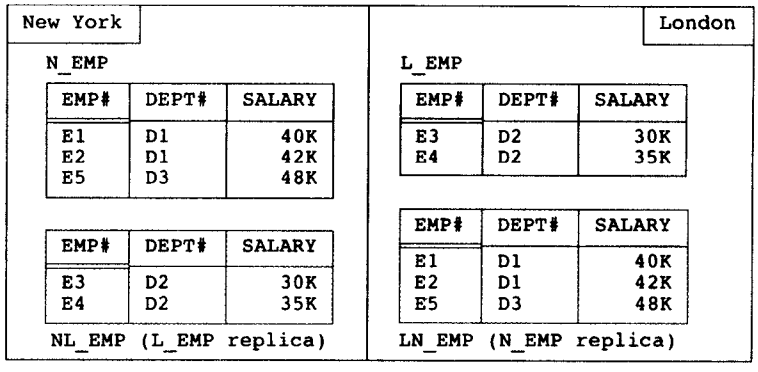



图 21-3　一个复制的示例

顺便说明一下，分布式系统中的复制代表了受控冗余这个概念的一种特殊应用，受控冗余是我们在第 1 章中曾经讨论过的。

现在在理想情况下，复制与分片一样，应该是“对用户透明的”。换句话说，支持数据复制的系统也应该支持**复制独立性**（也叫做**复制透明性**），即至少在逻辑上，用户的行为应该像数据并没有被复制时一样。之所以要提倡复制独立性（与位置独立性和分片独立性一样），是因为它简化了用户程序和终端操作；尤其是为了适应变化的需求，可以随时创建或销毁副本，而无须使任何上面所说的用户程序或操作失效。

复制独立性意味着，为了满足用户的任何请求，系统优化器要负责决定应该物理地访问哪一个副本的数据。这一问题在这里不再讨论。

最后，需要指出，现在许多商用产品支持一种不具备完全的复制独立性的复制形式（即不是完全“对用户透明”）。关于这一问题的进一步说明请参见 21.4 节中的“更新传播”小节。

**7. 分布式查询处理**

在这里有两个要点。

首先，来看看查询“给出供应红色零件的伦敦供应商”。假设用户是在纽约的场地而数据是在伦敦的场地，再假设满足请求的供应商有 $n$ 个。一方面，如果系统是关系的，则查询将主要涉及两条消息：一条是将查询请求从纽约发往伦敦，另一条是将 $n$ 个元组的结果集合从伦敦发回到纽约。另一方面，如果系统不是关系的，而是一次一记录的系统，则查询将涉及 $2n$ 条消息：$n$ 条从纽约到伦敦要求“下一个”供应商，$n$ 条从伦敦到纽约返回“下一个”供应商。这个例子说明，一个关系系统的性能可能比一个非关系系统要高出若干个数量级。

其次，优化在一个分布式系统中比在一个集中式系统中更重要。基本的观点是，在一个像上面一样涉及多个场地的查询中，会有很多种在系统中移动数据的方法可以满足查询请求，而找到一个高效的策略是至关重要的。比如，对于一个将场地 $X$ 上的关系 $rx$ 和场地 $Y$ 上的关系 $ry$ 合并的查询请求来说，执行的方法可以是把 $rx$ 迁移到 $Y$ 或是把 $ry$ 迁移到 $X$ 或是把它们都迁移到第三个场地 $Z$ 上等等。在 21.4 节中有对于这一点的特别说明，它涉及前面提到过的查询（“给出供应红色零件的伦敦供应商的供应商号”）。简要概括一下这个例子，在一组特定的可能的假设下分析了 6 种不同的查询处理策略，响应时间最短的只要 1/10 秒，最长的竟然要将近 6 个小时！优化显然是至关重要的，而这也可以看作是另一个理由来解释为什么分布式系统通常都是关系的（主要是关系的查询请求是可以优化的，而非关系的则不行）。

**8. 分布式事务管理**

事务管理主要有两个方面，恢复控制和并发控制。两者在分布式环境中都需要扩展处理方

式。为了解释扩展处理方式，我们需要引入一个新的术语——代理。在一个分布式系统中，一个单独的事务可以涉及执行多个场地上的代码，尤其是它可以对多个场地进行修改。因此每个事务都可以被看作是由许多**代理**组成，这里代理是指在一个场地上代表一个事务执行的进程。系统需要知道哪两个代理是属于同一个事务的——比如，很显然两个属于同一个事务的代理之间不能发生死锁。

现在先专门看看恢复控制。为了保证一个给定的事务在分布式环境中是原子的（都做或者都不做），系统必须保证这个事务的所有代理要么全部一起提交，要么全部一起回滚。通过第 15 章 15.6 节讨论过的**两阶段提交**协议（尽管不是分布式环境）可以获得这样的效果。在 21.4 节中将对分布式系统的两阶段提交协议做进一步说明。

对于并发控制而言，同在非分布式系统中一样，在大多数分布式系统中并发控制主要是基于**封锁**的（一些产品开始实现多版本控制 [16.1]；但是在实践中，传统的封锁机制看起来仍然是大部分系统的技术选择）。我们同样将在 21.4 节中对这一问题进行更详细的讨论。

**9. 硬件独立性**

对于这个问题确实没有太多要说的——题目已经说明了一切。现实世界中安装的计算机包括了种类繁多的机型——IBM 的机器、富士通的机器、HP 的机器以及各种各样的 PC 和工作站，等等——从而确实需要能够把所有这些系统中的数据集成起来，并给用户提供一个“单一系统映像”（single-system image）[21.9]。因此就非常需要能够在不同的硬件平台上运行同样的 DBMS，进一步地讲，要能够让这些不同的机器作为对等的合作者参与到分布式系统中来。

**10. 操作系统独立性**

可以部分地将这个目标看作是硬件独立性目标的推论。很显然，同样的 DBMS 应该不仅仅能够运行在不同的硬件平台上，而且也可以运行在不同的操作系统平台上——包括在同样硬件平台上的不同操作系统——从而让这些（比如说）OS/390 版本的、UNIX 版本的、Windows 版本的 DMBS 可以参与到同一个分布式系统中。

**11. 网络独立性**

如果系统能够支持许多完全不同的场地，这些场地是基于完全不同的硬件、运行完全不同的操作系统的，这显然就需要系统也能够支持各种完全不同的通信网络。

**12. DBMS 独立性**

对于这个目标，我们需要考虑放松对同构假设的限制会有什么样的结果。可以证明这个假设有些过于严格了：实际上只要不同场地上的 DBMS 实体都支持相同的接口就足够了——这些实体并不需要是同一个 DBMS 软件的副本。比如，如果 Ingres 和 Oracle 都支持官方的 SQL 标准，则让一个 Ingres 场地和一个 Oracle 场地在一个分布式系统的环境中交互是很可能的。换句话说，分布式系统至少在某种程度上可以是异构的。

支持异构性无疑是非常必要的。事实上，现实世界中的绝大部分计算机设备不仅仅是运行着许多不同的机器和许多不同的操作系统，也同样经常运行着不同的 DBMS；而如果这些不同的 DBMS 都能够在某种程度上参与到同一个分布式系统中，这将是一件非常好的事情。换句话说，理想的分布式系统应该提供 **DBMS 独立性**。

但是，DBMS 的独立性是一个非常大的专题，也是实践中非常重要的专题，在 21.6 节我们会专门讨论 DBMS 独立性。

## 21.4　分布式系统面对的问题

在这一节，我们要对在 21.3 节中提到的某些问题进行更进一步的阐述。最重要的问题是通信网络的速度太慢——至少“远距离网”（long haul）或者广域网是这样。一个典型的广域网的数据传输速率大约是每秒 5 ~ 10KB；与之相比，典型的磁盘驱动器的数据传输率大约是每秒 5 ~ 10MB（另一方面，某些局域网支持与磁盘驱动器相同数量级的数据传输率）。这样产生的结果就是，在分布式系统中最重要的目标（至少在使用广域网的情况下，以及某些使用局域网的情况下）是尽量减少对网络的使用——即尽可能减少消息的数量和大小。这个目标接着会引出不

少其他方面的问题，以下是其中的一部分：

- 查询处理
- 目录表管理
- 更新传播
- 恢复控制
- 并发控制

**1. 查询处理**

尽可能减少使用网络的目标意味着查询优化进程本身需要是分布的，查询执行进程也一样。换句话说，整个查询优化进程一般将由两个步骤构成：首先是**全局**优化，之后是各个参与场地上的**本地**优化。比如，假设场地 *X* 提出一个查询 *Q*，再假设 *Q* 涉及一个连接操作，连接的分别是场地 *Y* 上拥有一万个元组的关系 *ry* 与场地 *Z* 上拥有一千万个元组的关系 *rz*。场地 *X* 上的优化器将选择一个全局策略来执行 *Q*；显然它应该决定把 *ry* 移向 *Z* 而不是把 *rz* 移向 *Y*（或者取决于连接结果的基数的数目，把 *ry* 和 *rz* 都移向 *X* 会更好）。然后，一旦它决定将 *ry* 移向 *Z*，则连接操作在场地 *Z* 的实际执行过程就取决于场地 *Z* 上的本地优化器了。

下面对这一点的更详细的举例说明是基于 Rothnie 和 Goodman 所写的论文［21.31］中给出的例子。注意：由于硬件的发展以及现代数据库规模的变化，这里列出的实际数据现在可能已经过时了，但是数据反映出来的信息仍然是有效的。

- 数据库（简化的供应商与零件）：

```
S   { S#, CITY }                    在场地 A 存储了 10 000 个元组
P   { P#, COLOR }                   在场地 B 存储了 100 000 个元组
SP  { S#, P# }                      在场地 A 存储了 1 000 000 个元组
```

  假设每个存储的元组有 25 个字节长（200 位）。

- 查询（“给出所有供应红色零件的伦敦供应商的供应商号”）：

```
( ( S JOIN SP JOIN P ) WHERE CITY = 'London' AND
                             COLOR = COLOR ('Red') ) { S# }
```

- 中间结果的估计元组数：
  红色零件数 = 10
  London 供应商的发货数 = 100 000
- 通信状态：
  数据传输率 = 50 000 bit/s
  访问时延 = 0.1s

现在考查以下可以用来处理这个查询的 6 种可能的策略，并利用下面的公式对每种策略 *i* 计算出其通信时间 *Ti*：

```
(总访问时延 ) + (总数据量 /数据传输率 )
```

也就是（以秒为单位）：

```
(消息数 / 10 ) + (位数 / 50000 )
```

1）把零件表发送到场地 *A*，并在场地 *A* 完成查询。

```
T1  =  0.1 + ( 100000 * 200 ) / 50000
    =  400 seconds approx. (6.67 minutes)
```

2）把供应商表和发货表发到场地 *B*，并在场地 *B* 完成查询。

```
T2  =  0.2 + ( ( 10000 + 1000000 ) * 200 ) / 50000
    =  4040 seconds approx. (1.12 hours)
```

3）在场地 *A* 先对供应商表和发货表进行连接，选择结果中供应商为伦敦的元组，然后对每

个这样的元组检查场地 *B* 上是否有相应的零件为红色的元组。每次这样的检查将涉及两条消息，一条查询和一条响应。这些消息的传送时间与访问时延相比应该可以忽略。

```
T3  =  20000 seconds approx. (5.56 hours)
```

4）在场地 *B* 选择零件表中颜色为红色的元组，然后对每一个选出的元组顺次查找场地 *A*，看在发货表中是否有相应的零件与 London 的供应商联系在一起的元组。同样每次这样的检查也涉及两条消息，这些消息的传送时间与访问时延相比应该可以忽略。

```
T4  =  2 seconds approx.
```

5）在场地 *A* 连接供应商表和发货表，选择结果中供应商为伦敦的元组，并对结果表的 S#和 P#进行投影，然后将结果表发送到场地 *B*。在场地 *B* 上完成查询。

```
T5  =  0.1 + ( 100000 * 200 ) / 50000
    =  400 seconds approx. (6.67 minutes)
```

6）在场地 *B* 上选择零件表中颜色为红色的元组，然后把结果表发送到场地 *A*，在场地 *A* 上完成查询。

```
T6  =  0.1 + ( 10 * 200 ) / 50000
    =  0.1 second approx.
```

图 21-4 对结果进行了汇总。

| 策略 | 技术 | 通信时间 |
|---|---|---|
| 1 | 把 *P* 发送到 *A* | 6.67 min |
| 2 | 把 *S* 和 SP 发送到 *B* | 1.12 h |
| 3 | 对每个伦敦的发货元组，检查零件是否为红色 | 5.56 h |
| 4 | 对每个红色的零件，检查是否存在伦敦的供应商 | 2.00 s |
| 5 | 把伦敦的发货表发送到 *B* | 6.67 min |
| 6 | 把红色零件发送到 *A* | 0.10 s（最佳） |

图 21-4　分布式查询处理策略（小结）

1）通信时间的差异是巨大的（最慢的要比最快的慢二百万倍）。

2）对于一种策略的选择而言，数据传输率和访问时延都是非常重要的因素。

3）对于最差的策略而言与通信时间相比，计算和 I/O 时间是几乎可以忽略的。注意：实际上，对于比较好的策略来说，是不是这样就不一定了［21.33］。对于快速局域网来说，就不能忽略计算和 I/O 时间。

此外，有些策略允许在两个场地上并行地进行处理，这样，对用户的响应时间也许在实际上会比一个集中式系统要少。但是，要注意的是此时我们对哪一个场地来接受最后的结果不予考虑。

**2. 目录表管理**

在一个分布式系统中，系统目录表中不仅包括基本表、视图、完整性约束、权限等通常的目录数据，而且还包括许多必需的控制信息，使系统能够提供所需的位置独立性、分片独立性和复制独立性。现在的问题是：目录表自己存储在哪里，怎样存储？下面是几种存储方法：

1）集中式：整个目录表只在一个单独的中心场地上存储一次。

2）完全复制：在每一个场地上都存储一个完整的目录表。

3）分片式：每个场地只保存与本场地对象相关的目录表，整个目录表是所有这些不相交的本地目录表的并集。

4）方法 1 与方法 3 的组合：与第 3 种方法一样，每个场地维护自己的本地目录表；此外，在一个单独的中心场地上维护一个所有本地目录表的统一副本，就像第 1 种方法那样。

上述的每一种方法都有它们自身的问题。方法 1 显然与“不依赖中心场地”的目标相矛盾；方法 2 会造成自治性的严重丧失，因为每次对目录表的修改都需要传送到每一个场地上去；方法 3 会使得非本地操作的代价非常高（平均下来，找到一个远程对象需要访问半数的场地）；方法 4 的效率要比方法 3 高很多（找到一个远程对象只需要进行一次远程目录表访问），但是它同样与“不依赖中心场地”的目标相矛盾。因此在实践中，系统一般不使用这四种方法中的任何一种！我们会通过例子说明在 R* ［21.37］中使用的方法。

为了解释 R* 的目录表是如何构造的，首先需要说明一下 R* 中的**数据对象命名**。目前数据对象命名在分布式系统中通常都是一个非常突出的问题。很有可能两个不同的场地 *X* 和 *Y* 都有一个叫做 *R* 的数据对象，比如说是一个基本表，这就意味着需要一种机制——大部分是通过场地名称的限定——来进行“区分”（即保证系统范围内的名称唯一性）。但是类似 *X.R* 和 *Y.R* 之类的限定名称如果暴露给用户，显然就会和位置独立性的目标冲突。因此，需要一种方法把用户所知道的名称映射到相应的系统所知道的名称。

下面是 R* 针对这一问题的解决方法。首先 R* 区分一个数据对象的**外部名**（printname）和**系统名**（system-wide name），其中外部名是指数据对象被用户正常引用时的名字（比如，在一个 SQL 的 SELECT 语句中的叫法），而系统名是指数据对象的全局唯一内部标识。系统名有四个部分：

- 创建者 ID（Creator ID）（用 CREATE 操作首次创建这个数据对象的用户的 ID）；
- 创建者场地 ID（Creator site ID）（执行 CREATE 操作所在的场地的 ID）；
- 本地名（Local name）（数据对象的非限定名称）；
- 生成场地 ID（Birth site ID）（数据对象最初存储的场地的 ID）。

用户 ID 在场地上是唯一的，而场地 ID 是全局唯一的。比如，系统名

```
MARILYN @ NEWYORK . STATS @ LONDON
```

指的是一个数据对象，也许是一个基本表，它的本地名是 STATS，它由在纽约场地的用户 Marilyn 创建，最初存储在伦敦场地上[⊖]。STATS 这个名称是保证永远不会改变的，甚至在数据对象迁移到了另外一个场地上的时候也不变（见下文）。

我们已经指出，用户一般通过他们的外部名来引用数据对象，构成一个外部名的是一个简单的非限定名——或者是系统名的“本地名”部分（上面的例子中是 STATS），或者是系统名的一个**同义词**，这个同义词是通过 R* 中特定的 SQL 语句 CREATE SYNONYM 来定义的。这里有一个例子：

```
CREATE SYNONYM MSTATS FOR MARILYN @ NEWYORK . STATS @ LONDON ;
```

比如，现在用户既可以说

```
SELECT ... FROM STATS ... ;
```

也可以说

```
SELECT ... FROM MSTATS ... ;
```

在第一种情况下（使用本地名），系统通过假设使用所有的缺省值来推断系统名，也就是，数据对象是由这个用户创建的、是在这个场地上创建的、最初也是存储在这个场地上的。顺便说一句，这些缺省假设的一个结果就是原有的 System R 的应用程序可以不加变化地在 R* 上运行（即一旦 System R 的数据在 R* 上进行了重新定义，要记住 System R 是 R* 的基础原型）。

在第二种情况下（使用同义词），系统通过询问相关的**同义词表**来决定系统名。同义词表可以被认为是目录表的第一个组成部分；每个场地为该场地上的每个用户维护一组这样的表，把用户所知道的同义词映射到系统名上。

除了同义词表，每个场地还要维护：

1）每个在该场地产生的数据对象的一个目录表表项。

2）每个现在存储在该场地的数据对象的一个目录表表项。

假设现在用户发出一个与同义词 MSTATS 有关的请求。首先系统在合适的同义词表中查找对应的系统名（这是一个纯粹的本地查找）。现在比如说它知道了生成场地是伦敦，于是它可以

---

⊖ 基本表直接映射到 R* 中存储的关系表中，就作者所知的几乎每个系统都是这样做的。参考附录 A 中对这一观点的评述。

询问伦敦的目录表（不失一般性，我们假设这是一个远程查找——第一次远程访问）。由上面的第1点我们知道，伦敦的目录表会包含这个数据对象的一个表项。如果数据对象仍然在伦敦，它现在就被找到了。但是，比如说，如果数据对象已经被迁移到了洛杉矶，在伦敦的目录表表项会给出这一信息，则系统现在就可以询问洛杉矶的目录表（第二次远程访问）。由上面的第2点我们知道，洛杉矶的目录表将包含这个数据对象的一个表项。所以这个数据对象最多经过两次远程访问就可以被找到。

进一步讲，如果该对象被再一次迁移，比如说迁移到旧金山，则系统会做如下工作：

1）插入一个旧金山的目录表项；

2）删除洛杉矶目录表项；

3）将伦敦的目录表项修改为指向旧金山而不是洛杉矶。

最终还是至多通过两次远程访问就可以找到这个数据对象。而且这是一个完全的分布模式——在系统中没有集中的目录表场地，从而没有单点故障。

说明一下，在DB2的分布式数据设备中的数据对象命名模式与上面所说的相似，但是并不相同。

**3. 更新传播**

如21.3节所指出的那样，数据复制的基本问题是，对一个给定逻辑对象的修改必须传播到该对象的所有存储副本上去。随之出现的一个问题是某个存储了该数据对象副本的场地可能在修改时是不可用的（由于场地故障或是网络故障）。因此最直接的策略，即将修改立即传播到所有副本上的策略，看起来是不能实现了，因为这意味着如果当时有一个副本不可用的话，修改——当然还包括事务——就将失败。实际上，从某种意义上说，在这种策略下数据的可用性比在无复制的情况下更差，因此，前面章节中所声称的数据复制的一个优点也就被削弱了。

处理这个问题的一个通常的模式（不是唯一可能的模式）是所谓的**主副本**模式，具体内容如下：

- 被复制对象的一个副本被指定为主副本。剩下的都为从属副本。
- 不同数据对象的主副本存储在不同的场地上（这同样说明这是一个分布模式）。
- 一旦完成了对主副本的修改，修改操作就被认为是逻辑地完成了。拥有主副本的场地则要负责在某个接下来的时间里把更新传播到所有的从属副本上。注意：如果要想保持事务的ACID属性，这个“接下来的时间”就必须在COMMIT之前（参见第15章和第16章）。我们后面还会讨论这个问题。

当然，这个模式本身也会带来许多其他问题，其中的大部分已经超出了本书所讨论的范围。还要注意到，它同时还带来了与本地自治目标之间的冲突，因为在这种情况下一个事务可能会由于一个远程（主）副本的不可用而失败——尽管本地副本是可用的。

如前所述，事务处理的ACID要求意味着所有的更新传播必须在相应的事务完成之前完成（“同步复制”）。但是，目前许多商用产品支持的是一种弱化形式的复制，在这种方式下修改的传播可以在某个稍后的时候（可能是某个用户指定的时候）来进行，而不必在相应事务完成的时间范围内（“异步复制”）。事实上，复制这个词已经被这些产品或多或少地篡改了，结果是——至少在商用市场上——它总是用来暗示更新传播将延迟到相应事务的提交点之后（比如，见参考文献[21.1, 21.16, 21.18]）。显然，延迟传播方法的问题是数据库无法再保证在任何时候都是一致的[⊖]。事实上，甚至用户都可能不知道它是不是一致的。

以下是对于延迟传播方法的3点额外说明：

1）在一个支持延迟更新传播的系统里，可以将复制的概念看作是第10章中快照方法的一种应用。实际上，用另外一个词来形容这种复制会更好，那么我们就可以让“副本”一词保留它

---

⊖ 当然，如果立即执行完整性检查（见第9章和第16章），就不会引发这样的情况。即使这样的检查推迟到COMMIT之后——我们可能认为这样在逻辑上是不正确的，但在某些系统中是存在的——还是不会引发这样的情况。所以，在某种程度上，我们认为延迟传播比推迟检查在逻辑上更加不正确。

在一般论述中的通常含义（即，一个准确的拷贝）了。注意：快照是只读的（除了定期更新外），然而一些系统允许用户直接更新副本——比如，见参考文献 [21.18]。当然，直接更新副本会引发与复制独立性的冲突。

2）我们并不是说延迟传播不是一个好的主意。比如在第 22 章我们将看到，在适当的情形下它显然是一种合适的方法。但是，最主要的是，延迟传播会导致“副本”并不是真正的副本（很可能逻辑层一个给定的数据值在物理层会有两个或更多个的数据值表示，甚至这些值会有所不同），而系统也不是真正的分布式数据库系统。

3）商用产品之所以用延迟传播来实现复制的一个原因（甚至是主要原因）在于，替代方法——即在 COMMIT 之前修改所有副本——需要两阶段提交的支持（见下一小节），但是这会导致性能上的很大开销。这种情况说明了为什么在商业出版物中有时会碰见具有类似这种奇怪标题的文章：“复制与两阶段提交”。其之所以奇怪是因为表面上看起来，他们好像是在比较两件完全不同的事情。

**4. 恢复控制**

正如 21.3 节所说明的，分布式系统的恢复控制一般是基于两阶段提交协议或它的某个变体。假如一个单独的事务要和许多自治的资源管理者交互，则在任何类似这样的环境下，都要用到两阶段提交协议。但是两阶段提交协议在一个分布式系统中尤为重要，因为所说的资源管理者——即本地的 DBMS——运行在不同的场地上，因此是完全自治的。以下是几点说明：

1）“不依赖中心场地”的目标指出协调功能不可以指定给网络中某个特别的场地，而必须针对不同的事务由不同的场地来执行。通常该功能都是由被执行事务的启动场地来完成的，因此，在通常情况下，每个场地必须既可以作为某些事务的协调者又作为另外一些事务的参与者。

2）两阶段提交进程需要协调者与每一个参与场地进行通信，这就意味着更多的消息传递与更大的开销。

3）如果场地 *Y* 是由场地 *X* 协调的两阶段提交进程中的一个参与者，那么场地 *Y* 必须按照场地 *X* 所告诉它的去做（按照需要进行提交或是回滚），这是本地自治性的一种（也许是较小的）损失。

让我们回顾一下第 15 章所描述的最基本的两阶段提交进程。图 21-5 显示了协调者与一个典型的参与者之间所进行的交互过程。不失一般性，我们假设协调者和参与者在不同的场地；并且要注意，图中的时间轴是从左向右的。为简单起见，假设事务要求执行一个 COMMIT 而不是一个 ROLLBACK。在接到 COMMIT 请求后，协调者完成如下两阶段进程：

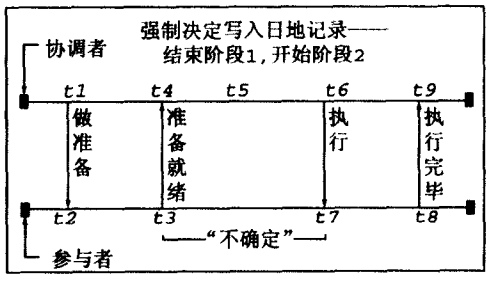



图 21-5　两阶段提交

- 协调者要求每个参与者针对事务“做好完成任何请求的准备”。图 21-5 表明“做好准备”的消息在 *t*1 时刻被发出，在 *t*2 时刻被参与者接收到。这时参与者强制本地代理将一条日志记录写入物理日志中，然后向协调者发一条“准备就绪”的消息（当然，假如出现了任何问题，尤其是本地代理发生故障，它将发出“没有准备好”的消息）。在图中，响应消息——“准备就绪”是在 *t*3 时刻被发出的，协调者在 *t*4 时刻收到这一消息。注意（如同已经指出的）参与者这时丧失了一定的自治性：它必须按照接下来协调者告诉它的去做。而且，任何被本地代理封锁的资源必须继续保持封锁，直到参与者接收到协调者的决定并依照执行。
- 从所有的参与者那里接收到响应以后，协调者将做出决定（如果所有的响应都是“准备就绪”，则决定提交，否则回滚）。然后它强制将一条日志记录写入物理日志，记录下该决定，时刻为 *t*5。*t*5 这一时刻标志着从阶段 1 到阶段 2 的过渡。
- 我们假设决定是提交。于是协调者要求所有的参与者“执行”（即为本地代理执行提交进

程）；图21-5表明“执行”的消息在 *t6* 时刻被发出，在 *t7* 时刻被参与者接收到。参与者为本地代理完成提交之后，向协调者发回一条确认消息（“执行完毕”）。在图中，确认消息是在 *t8* 时刻被发出的，协调者在 *t9* 时刻收到这一消息。

- 当协调者接收到了所有的确认消息，整个进程就结束了。

当然，在实际情况中整个进程肯定比刚才所描述的要复杂得多，因为必须考虑到场地和网络故障的情况。比如，假设协调者场地在 *t5* 与 *t6* 之间的某个时刻 *t* 出现了故障，则在场地恢复之后，通过日志，重新启动的过程将发现有一个事务正处在两阶段提交进程的第二个阶段，于是将通过向所有的参与者发出“执行”的消息来继续这一进程（需要注意的是在该事务中，参与者在 *t3* ~ *t7* 期间是处在“不确定”状态下。如果协调者确实如所说的是在时刻 *t* 发生故障的，则这个“不确定”的状态将持续很长的一段时间）。

当然在理想的情况下，我们希望两阶段提交进程对于任何可能的故障都保持弹性。不幸的是，很显然这一目标从根本上是无法达到的——即面对任意可能的故障，不存在任何有限次的协议可以保证所有的参与者能够一致地提交一个成功的事务或回滚一个不成功的事务。反过来说，假设存在这样一个协议，则 *n* 是这个协议所需要的最少的消息数目。现在再假设由于某种故障，这 *n* 个消息中最后一个消息丢失了，则要么这个消息是不必要的（这与 *n* 为最少数目的假设相矛盾），要么协议无法生效。任何一种可能都会引出矛盾，从而可以推断出不存在这样的协议。

尽管这一事实让人不快，但是从改进性能的角度来看，至少还是可以不同程度地加强一些基本算法：

- 首先（见参考文献［15.6］的注释），如果在某个参与者场地上的代理是只读的，则这个参与者在第一阶段可以发出“忽略我”的响应消息，而协调者就可以在第二阶段真正地忽略它。
- 第二（见参考文献［15.6］），如果所有的参与者在第一阶段的响应消息都是“忽略我”，则第二阶段就可以被完全地跳过。
- 第三，基本模式有两个很重要的变体，叫做假想提交和假想回滚，在下面几段将对它们进行更详细的讨论。

实质上，假想提交模式具有减少成功的情况下所需消息数量的效果，而假想回滚可以减少失败的情况下所需消息的数量。为了解释这两种模式，首先要注意在上面所叙述的基本机制中，协调者要始终记住它的决定直到它从每一个参与者那里都接收到了一个确认消息。当然，原因是如果一个参与者在“不确定”的状态下崩溃了，则在重新启动的时候它必须要询问协调者以获知协调者的决定是什么。但是，如果所有的确认消息都被接收到了，协调者就知道所有的参与者都已经按照被告知的执行了，那么它就可以“忘记”这个事务了。

我们现在来看看**假想提交**。在这种模式下，要求参与者确认“回滚”（“不执行”）消息而不需要确认“提交”（“执行”）消息。而且如果决定是“提交”，协调者可以在广播了它的决定之后立即“忘记”这个事务。如果一个不确定的参与者崩溃了，则在重启动的时候它将（像通常一样）询问协调者；如果协调者仍然记着（存储着）这个事务（即协调者仍然在等待参与者的确认消息），那么决定肯定是“回滚”，否则肯定是“提交”。

当然，**假想回滚**正好相反：参与者被要求确认“提交”消息而不是“回滚”消息，而协调者可以在广播了它的决定之后忘记这个事务，只要决定是“回滚”。如果一个不确定的参与者崩溃了，则在重启动的时候它将询问协调者；如果协调者仍然记着（存储着）这个事务（即协调者仍然在等待参与者的确认消息），那么决定肯定是“提交”，否则肯定是“回滚”。

有趣的是（也是有些有悖于直觉的是），假想回滚似乎比假想提交更可取（我们说它“有悖于直觉”是因为绝大部分事务肯定是成功的，而假想提交可以减少成功情况下消息的数目）。假想提交的问题如下：假设协调者在第一阶段（即在它做出决定之前）崩溃了，则在协调者场地重启动的时候，事务将回滚（因为它没有完成）。接下来，某个参与者询问协调者关于这个事务的决定，而协调者已经不记得这个事务了，那么就将假定决定是“提交”——毫无疑问这是不

对的。

为了避免这种“错误提交”，协调者（如果采用假想提交的话）必须在开始第一阶段的时候向它的物理日志中强制写入一条日志记录，给出事务中所有的参与者（现在如果协调者在第一阶段崩溃了，则在重启动的时候它将向所有的参与者广播“回滚”消息）。而对协调者日志的这一次物理 I/O 对每一个事务都是关键所在，于是假想提交就不像它给人的第一印象那样吸引人了。实际上，不夸张地说，截至在本书写作的时候，假想回滚已经是现有的实现系统事实上的标准了。

**5. 并发控制**

如同在 21.3 节所说明的，绝大部分分布式系统中的并发控制都是基于封锁的。但是在一个分布式系统中，测试、设置以及释放锁的请求都变成了消息（假设所考虑的数据对象在一个远程场地上），而消息就意味着开销。比如，一个事务 $T$ 要修改一个数据对象，而这个对象在 $n$ 个远程场地上都存在副本。如果每个场地都负责对存储在该场地上的对象进行封锁（就像在本地自治假设下所做的一样），则最直接的实现方式至少需要 $5n$ 条消息：

$n$ 条封锁请求
$n$ 条封锁授权
$n$ 条修改消息
$n$ 条确认消息
$n$ 条解锁请求

当然我们可以很容易地通过“搭载”消息来改进上述过程——比如，封锁请求和修改消息可以进行合并，封锁授权和确认消息也同样可以合并——但即使是这样，修改所需要的全部时间也会比在一个集中式系统中高出好几个数量级。

通常解决这一问题的方法是修改在前面“更新传播”小节中提出的主副本策略。对一个给定的数据对象 $A$，拥有 $A$ 的主副本的场地将处理所有有关 $A$ 的封锁操作（要记住，一般情况下不同对象的主副本存储在不同的场地上）。在这一策略下，针对封锁而言，一个对象的所有副本的集合可以被看作是一个单一的对象，而消息的总数也将从 $5n$ 减少到 $2n+3$（一条封锁请求、一条封锁授权、$n$ 条修改消息、$n$ 条确认消息以及一条解锁请求）。但是需要再次注意的是，这个解决方案会带来自治性的（严重）丧失——如果一个主副本不可用了，一个事务就将失败，即使这个事务是只读的，而且有一个本地副本是可用的。（注意不仅修改操作需要封锁主副本，而且检索操作也需要封锁主副本 [15.6]，因此主副本策略的一个不好的副作用就是，它会降低检索和修改的性能和可用性。）

在一个分布式系统中进行封锁的另外一个问题是它会导致**全局死锁**。全局死锁是涉及两个或者多个场地的死锁。比如（如图 21-6 所示）：

1）事务 T2 在场地 X 上的代理正在等待事务 T1 在场地 X 上的代理释放一个锁；

2）事务 T1 在场地 X 上的代理正在等待事务 T1 在场地 Y 上的代理完成操作；

3）事务 T1 在场地 Y 上的代理正在等待事务 T2 在场地 Y 上的代理释放一个锁；

4）事务 T2 在场地 Y 上的代理正在等待事务 T2 在场地 X 上的代理完成操作；死锁！

对任何一个场地而言，只用该场地内部的信息是无法检测出像这样的死锁问题的。换句话说，在本地的等待图中并没有回路，但是，如果两个本地图被合成了一个全局图，就会出现一个回路。这就意味着全局死锁检测会带来进一步的通信开销，因为它要求独立的本地图以某种方式被集中到一起。

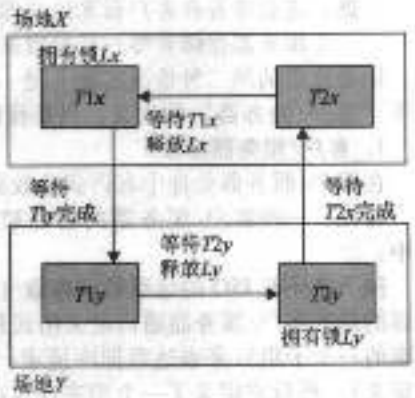



图 21-6 全局死锁的一个示例

在关于R*的论文中描述了一个漂亮的（也是分布的）模式用于全局死锁检测（例如，见参考文献［21.37］）。注意：在第16章我们曾指出，在实际中并不是所有的系统都进行死锁检测，有些只是用超时机制来替代。由于显而易见的原因，这种方式对于分布式系统而言是尤为实际的。

## 21.5　客户/服务器系统

在21.1节中我们曾提到，可以将客户/服务器系统看作是普遍意义上的分布式系统的一种特殊情况。更确切地说，一个客户/服务器系统是这样一个分布式系统：（a）某些场地是客户场地，而某些场地是服务器场地；（b）所有的数据都驻留在服务器场地；（c）所有的应用都在客户场地运行；（d）“存在接缝”（不提供完全的位置独立性）。如图21-7所示（与第2章中的图2-6相同）。

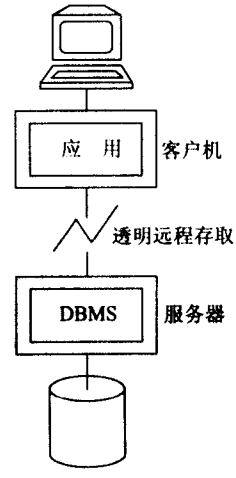


图21-7　一个客户/服务器系统

20世纪80年代末至90年代中期，在客户/服务器系统中蕴含着巨大的商业利益，而在真正的通用分布式系统中商业利益则相对少得可怜。下一节我们将看到，现在这种情形有了一定程度的改变，但是客户/服务器结构始终很重要，因此我们在这里仍然要讨论一下。

首先，回忆一下，“客户/服务器”这个词主要是指一种体系结构，或是对职责的逻辑划分。客户是指应用（也叫做前端），而服务器是指DBMS（也叫做后端）。但是恰恰是由于整个系统可以被清晰地划分为两个部分，也就有了将两个部分运行到不同的机器上的可能。而这种可能性是如此的吸引人（见第2章2.10节的说明），以至于“客户/服务器”这个词几乎只被应用于客户和服务器的确运行在不同的机器上的情况[⊖]。这种用法是普遍的，但也是草率的，我们将在下面对它进行修正。

注意在基本模式之上还可能存在许多变体：

- 许多客户也许可以共享同一个服务器（实际上，这是正常的情况）。
- 一个单一的客户也许可以存取许多服务器，这种可能性又可以分为两种情况：

  1）限制客户一次只可以存取一个服务器——即每个独立的数据库请求必须只针对一个服务器，不可能出现在一个单一的请求中同时包含来自两个或多个不同服务器的数据的情况。而且用户必须要知道哪一个服务器存储着哪一部分数据。

  2）客户可以同时存取多个服务器——即一个单一的数据库请求可以包含多个服务器的数据，这意味着在客户看来这些服务器好像实际上只是一个服务器，而用户也不需要知道哪一个服务器存储着哪一部分数据。

但是这里的第二种情况实际上是一个真正的分布式系统（“隐藏了接缝”），它并不是通常情况下“客户/服务器”的含义，只是我们在下面将忽略这一点。

### 1. 客户/服务器标准

在客户/服务器处理中有许多生效的标准：

首先，一些客户/服务器的特征被包括在SQL标准中，对于这些特征的讨论将放到21.7节中。

接下来还有ISO的**远程数据存取**（Remote Data Access，RDA）标准［21.23，21.24］。RDA的目的是为客户/服务器通信定义格式和协议。它假设a）客户用标准形式的SQL（基本是SQL标准的一个子集）来表达数据库请求；b）服务器支持一个标准的目录表（同样遵循SQL标准的定义）。然后它定义了一个用来在客户与服务器之间传递消息（SQL请求、数据和结果以及分

⊖ 很明显，术语“两级系统”（two-tier system）本质上也是同样的意思。

析信息）的特定的表达格式。

我们在这里要提到的第三个也是最后一个标准是 IBM 的**分布式关系数据库体系结构**（Distributed Relational Database Architecture，DRDA）标准［21.22］（它是一个现实中的标准，而不是一个理论上的标准）。DRDA 和 RDA 有着类似的目标，但是 DRDA 在许多重要的方面与 RDA 不同——尤其是 DRDA 倾向于反映出它的 IBM“血统”。比如，DRDA 并不假设客户端使用的是标准版本的 SQL，而是代之以允许任何形式的 SQL。一个（可能的）后果就是获得更好的性能，因为客户端有可能利用某些服务器所特有的功能；另一方面这又会带来可移植性的损失，恰恰是因为那些服务器所特有的功能对客户端是开放的（即客户端知道与之对话的是哪一种服务器）。基于同样的想法，DRDA 并不假设在服务器端有任何特定的目录结构。DRDA 的格式和协议与 RDA 的非常不同（关键是，DRDA 是基于 IBM 自己的体系结构和标准，而 RDA 是基于国际的、与供应商无关的标准）。

关于 RDA 与 DRDA 进一步的细节已经超出了本书的范围，在参考文献［21.20］和［21.28］中可以找到关于它们的一些分析和比较。

**2. 客户/服务器应用程序编程**

我们曾经说过，客户/服务器系统是普遍意义上的分布式系统的一种特殊情况。如同在本节开始的引言中所说，一个客户/服务器系统可以被看作是这样一个分布式系统，所有的请求都源自于一个场地而所有的处理都在另一个场地上执行（为了简单起见，假设只有一个客户场地和一个服务器场地）。注意：显然在这个简单的定义下，仅凭其自身而言，一个客户场地完全不是一个真正的“数据库系统的场地”。因此，这种系统与在 21.2 节中所给出的通用分布式系统的定义相抵触。当然，客户场地同样可以有其本地的数据库，但是这些数据库并不直接构成这种客户/服务器方案中的一部分。

尽管如此，客户/服务器方法在应用程序编程中确实有其特定的实现方式（实际上一般的分布式系统也一样）。在所有最重要的方面中，有一点在 21.3 节中关于目标 7（分布式查询处理）的讨论中我们已经接触过了：也就是通过定义和设计，我们知道关系系统是**集合层次**的系统。在一个客户/服务器系统中（实际上一般地说，是在分布式系统中），比以往更为重要的是，应用程序员不仅仅“像使用一种存取方法那样使用服务器”并编写针对记录层次的代码，而是要把尽可能多的功能捆绑进针对集合层次的请求中——否则执行性能将受到影响，这种影响是由执行所涉及的消息数量带来的。注意：在 SQL 术语中，上面的表述意味着要尽可能避免使用游标——即避免 FETCH 循环和避免 UPDATE 与 DELETE 的 CURRENT 形式（见第4 章）。

如果系统提供某种存储过程机制，那么在客户和服务器之间的消息还可以进一步减少。一个**存储过程**本质上是一个存储在服务器场地上的（并且是可以被服务器识别的）、预编译的程序，它通过**远程过程调用**（RPC）被客户执行。因此，对由于进行记录层次的处理所造成的性能损失，可以通过建立一个合适的存储过程并在服务器场地上直接运行它来部分地抵消。

注意：尽管这与我们关于客户/服务器处理的论题有些偏离，还是应该指出存储过程的好处不仅仅在于提高性能，而且还包括：

- 这种过程还可以用来对用户隐藏各种与 DBMS 以及数据库相关的细节，从而与其他情况相比可以提供更大程度的数据独立性。
- 一个存储过程可以被许多客户共享。
- 可以在创建存储过程的时候，而不是执行它的时候进行优化（当然这一优点是相对那些通常进行运行时优化的系统而言的）。
- 存储过程可以提供更好的安全性。比如，一个给定的用户可以被授权执行一个给定的过程，但是却无法对该过程所存取的数据进行直接的操作。

它的一个不足之处在于不同的供应商在这方面会提供非常不同的功能软件，尽管——如同在第 4 章所提到的——在 1996 年已经对 SQL 标准进行了扩展（增加了持久存储模块（Persistent Stored Module），SQL/PSM），以包括某些存储过程支持。

## 21.6　DBMS 独立性

现在回到对于一般分布式系统的十二个目标的讨论上来。大家应该记得，最后的一个目标是 DBMS 独立性。在 21.3 节对于这些目标的简短讨论中我们已经表明，严格的同构性假设是太强了，实际上所需要的只是在不同场地上的 DBMS 具有同样的接口。我们在 21.3 节中提到，如果（比如说）Ingres 和 Oracle 都支持官方的 SQL 标准，就有可能让它们在一个异构的分布式系统中充当同等的合作者；事实上，通常这种可能性会首先被提出来作为支持 SQL 标准的理由之一，然后我们再对其仔细地检查。注意：我们把讨论建立在 Ingres 和 Oracle 的基础之上，只是为了使讨论更有针对性一些。所讨论的概念毫无疑问是普遍适用的。

1. 通道

假设有两个场地 $X$ 和 $Y$ 分别运行 Ingres 和 Oracle 系统，而与此同时在场地 $X$ 上的用户 $U$ 希望看到一个单一的分布式数据库，它包括来自于场地 $X$ 上 Ingres 数据库的数据和来自于场地 $Y$ 上 Oracle 数据库的数据。通过定义，用户 $U$ 成为一个 Ingres 用户，因此对用户而言，这个分布式数据库必须是一个 Ingres 数据库。这样，不是 Oracle 而是 Ingres 就有责任提供所需要的支持。那么这种支持包括哪些方面呢？

原则上，这种支持非常简单明了：Ingres 必须提供一个特殊的程序——一个通常所说的**通道**（gateway）——其效果是“使 Oracle 看起来像 Ingres 一样”。如图 21-8 所示[⊖]。通道可能运行在 Ingres 场地上或者 Oracle 场地上或者（如图所示的那样）是在两个场地之间的某个通道自己的特殊场地上。但是不论它在哪里运行，它至少必须明确地提供下述的所有功能。可以发现其中的许多功能会带来某个重要特征的实现问题。但是，在 21.5 节所讨论的 RDA 和 DRDA 标准确实可以解决其中的一些问题，XML 也能做到这一点（参见第 27 章）。

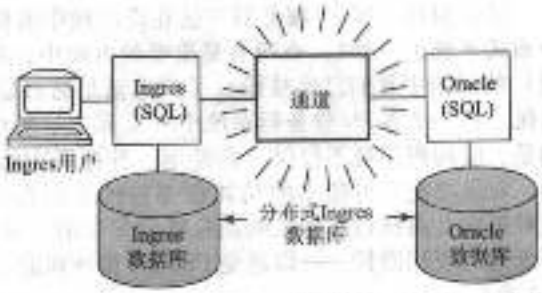



图 21-8　一个假想的 Ingres 提供的针对 Oracle 的通道

- 实现在 Ingres 和 Oracle 之间的信息交换协议——包括把从 Ingres 发出的源 SQL 语句的消息格式映射为 Oracle 所要求的格式，并把由 Oracle 发出的结果的消息格式映射为 Ingres 所要求的格式。
- 为 Oracle 提供一种“SQL 服务器”的功能（类似于已经在大部分 SQL 产品中实现的交互式 SQL 处理器）。换句话说，通道必须能够在 Oracle 数据库上执行任意的 SQL 语句。为了提供这一功能，通道必须利用对动态 SQL 的支持或者（更有可能的）是利用 Oracle 场地上所提供的 SQL/CLI 或 ODBC、JDBC 之类的调用层接口（见第 4 章）。注意：除此之外，通道也可以直接利用由 Oracle 提供的交互式 SQL 处理器。
- 在 Ingres 和 Oracle 之间进行数据类型映射。这个问题包括一系列与其他事情有关的子问题，这里所说的其他事情包括处理器之间的差异（比如，不同的机器字长）、字符代码之间的差异（对字符串的比较以及 ORDER BY 请求的反应会给出非预期的结果）、浮点格式之间的差异（一类让人极其反感的问题）、对日期和时间支持上的差异（就笔者所知，目前没有哪两个 DBMS 在这方面提供了相同的支持），更不用说用户自定义的类型的差异。对于这些问题的进一步讨论可以参见文献 [15.6]。
- 将 Ingres 的 SQL 语言映射成 Oracle 的 SQL 语言——因为实际上 Ingres 和 Oracle 都没有完全、支持 SQL 标准。事实上它们都是只支持其中的某一部分功能，而对另一些则不支持，

⊖ 很明显，术语“三级系统”有时指的是图中的特别结构（有时指类似二个组件的软件结构；可参阅参考下一小节中对“中间件”的讨论）。

同时还有些功能在两个产品中有同样的语法但却有着不同的语义。注意：在这种连接中，确实有些通道产品提供了一种传递转移（passthrough）的机制，利用这种机制，用户可以（比如说）直接用目标系统的语言书写一个查询，然后把它不作修改地通过通道传递到目标系统上进行执行。

- 将 Oracle 的反馈信息（返回代码之类的）映射为 Ingres 的格式。
- 将 Oracle 的目录表映射为 Ingres 的格式，从而使得 Ingres 场地和在该场地上的用户可以知道在 Oracle 数据库中有些什么。
- 处理在异构系统之间很容易出现的各种**语义不匹配**的问题（可以参见文献［21.8，21.11，21.14，21.36，21.38］）。这可能会包括命名的差异（在 Ingres 中可能使用 EMP#，而在 Oracle 中可能使用 EMPNO）；数据类型的差异（在 Ingres 使用字符串的地方在 Oracle 中可能使用数字）；单位的差异（在 Ingres 中可能使用厘米，而在 Oracle 中可能使用英寸）；信息逻辑表达方式的差异（Ingres 可能忽略在 Oracle 中使用空值的元组）等。
- 作为一个两阶段提交协议（Ingres 变体）的参与者（假设 Ingres 的事务可以在 Oracle 数据库上执行修改操作），通道是否能够真正执行这一功能取决于在 Oracle 场地上的事务管理器的能力。值得指出的是，在本书写作的时候，商用的事务管理器（除了某些例外）基本都不提供这方面所需要的功能——也就是让应用程序具有可以发出指令要求事务管理器“准备结束”的能力（相反，而是只具有发出指令要求事务管理器结束，即无条件提交或者回滚的能力）。
- 在 Ingres 需要的时候，保证 Ingres 要求在 Oracle 场地上进行封锁的数据能够确实得到封锁。同样，通道是否能够真正执行这一功能也基本取决于 Oracle 的封锁结构是否与 Ingres 的相匹配。

至此我们只是讨论了在关系系统环境下的 DBMS 独立性，那么其他的非关系系统的情形又如何呢？也就是在一个本来的关系分布式系统中加入一个非关系的场地的可能性是多少呢？比如，是否可能提供从一个 Ingres 或 Oracle 场地到一个 IMS 场地的访问？同样，这样的能力在实践中是非常需要的，因为现在有大量的数据存储在 IMS 以及其他在关系系统出现以前就已经存在的系统中[⊖]。但是这可以做到吗？

如果这个问题的意思是“它可以 100% 做到吗？”或者说是“所有的非关系数据可以通过一个关系接口被访问，并在其上执行所有的关系操作吗？”答案绝对是不可以，在参考文献［21.14］中对此有详细的解释。但是，如果这个问题的意思是“可以提供一些有效完成这样功能的工具吗？”，则答案显然是可以。不过这里不便给出详细的讨论，进一步的说明可以参见文献［21.13，21.14］。

**2. 数据存取中间件**

上面一小节所描述的通道（有时更确切地叫做点对点通道）其实会受到许多局限。其中之一就是它几乎没有提供位置独立性，还有就是同一个应用程序可能需要利用许多不同的通道——比如一个是针对 DB2 的、一个是针对 Oracle 的、一个是针对 Informix 的——而却不提供任何（比如说）对于跨越所有此类场地的连接操作的支持。这种情况的一个后果就是——尽管有上面一小节中所提到的技术困难，功能愈加复杂的通道类产品在过去的几年中出现得越来越频繁了。事实上，整个所谓的中间件（也叫做中介）的业务现在从其自身角度而言，已经成为一个非常引人注目的产业。

所以也许并不奇怪，“中间件”这个词还没有得到精确的定义：对于在不同程度上一起工作的不同系统而言，任何用于掩盖其间差异的软件（比如一个 TP 监控器）都有理由被认为是一个“中间件”［21.3］。但是在这里我们只关注可以叫做数据存取中间件的软件。在这类产品中有 Cohera 公司的 Cohera、IBM 公司的 DataJoiner 以及 Sybase 公司的 OmniConnect 和 InfoHub。这里

---

⊖ 85% 的商业数据仍然存储在这样的系统中（也就是关系系统出现前存在的数据库系统，甚至是文件系统），并且没有迹象表明这些数据将很快移到更新的系统中。

我们简要地介绍一下 DataJoiner 产品［21.6］。

DataJoiner 有许多不同的配置方法（如图 21-9 所示）。从一个独立的客户的角度而言，它看起来像一个正规的数据库服务器（即一个 DBMS），它存储数据、支持 SQL 查询、提供一个目录表、执行查询优化等（实际上 DataJoiner 的核心是 IBM 的 DBMS 产品——DB2 的 AIX 版本）。但是数据主要不是存储在 DataJoiner 场地上（虽然提供了这样的功能），而是存储在幕后的任意数量的其他场地上，这些场地由许多不同的 DBMS 控制着，或者甚至是类似 VSAM 这样的文件管理系统。这样，DataJoiner 向用户提供了一个有效的虚拟数据库，这个数据库是所有那些“幕后的”数据存储的联合体。它允许进行跨越这些数据存储的查询[○]，并利用其对于这些幕后系统的性能（以及网络特点）的了解来决定“全局最优的”查询计划。注意：DataJoiner 还具有模拟能力，能够在不直接支持某些特定的 DB2 SQL 功能的系统上对这些功能进行模拟。在游标声明时的 WITH HOLD 选项可以作为这样的一个例子（参见第 15 章）。

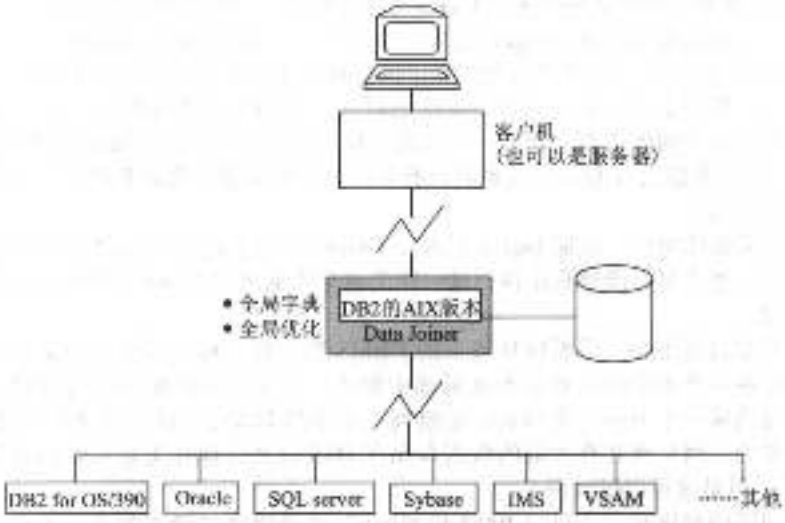



图 21-9　作为数据存取中间件的 DataJoiner

直到现在，我们所描述的系统仍然不是一个完全的分布式系统，因为在幕后的各个场地之间并不知道彼此的存在（即在协同的操作中它们不能被认为是同等的合作者）。但是，如果有任何新的“幕后”场地被添加进来，它同样可以作为一个客户场地，并通过 DataJoiner 执行对某个或者所有其他场地进行访问的查询。整个系统因此构成一个常说的**联合**（federated）系统，也叫做**多数据库**系统［21.17］。一个联合系统是一个接近于完全本地自治的分布式系统（通常是异构的）。在这样的一个系统中，纯粹的本地事务由本地的 DBMS 来管理，但是全局事务就是另外一回事了［21.7］。

DataJoiner 对每一个“幕后的”系统都有一个内置的驱动部件——从功能上说就是上一小节中的点对点通道（这些驱动部件基本上都是利用 ODBC 来访问远程的系统）。DataJoiner 同时还维护一个全局目录表，在遇到系统之间语义不匹配的时候用来告诉系统应该如何处理。

我们注意到，类似 DataJoiner 这样的产品对于第三方软件供应商来说是非常有用的。他们可以进行通用的工具开发（比如报表生成子、统计包等），而不必考虑有可能运行这些工具的不同的 DBMS 产品之间的差异。最后，我们注意到，IBM 最近已经把 DataJoiner 技术应用到它的 DBMS 产品 DB2 中去了；意图很明显，就是要把 DB2 变成存储以各种形式存在的数据（在本书写作的时候，它支持对存储在 Informix、Oracle、SQL Server、Sybase 以及其他系统中数据的存

○　此处强调的是“查询”；更新能力必然会受某些限制，尤其是——但不是完全——当幕后的系统是 IMS 或是其他的非 SQL 系统时（请再次参见文献［21.14］）。

取）的“唯一真正的接口”——至少是唯一真正的 IBM 接口。也就是说，拥有 DataJoiner 技术的 DB2，表明 IBM 意图解决已为人们熟知的所谓的**信息集成**（information integration）问题[21.9]。

**最后的说明**

很显然，要提供完全的 DBMS 独立性会有不少突出的问题，即使是所有参与的 DBMS 都是专门的 SQL 系统。但是潜在的效益同样也是巨大的，即使解决方案并不是完美的。也正是因为如此，才出现了许多数据存取的中间件产品，而且在不久的将来肯定还会有更多的产品出现。但是要提醒你的是解决方案肯定不可能是完美的——尽管供应商们所说的恰恰相反。购买者请当心！

## 21.7　SQL 的支持

目前，SQL 根本没有提供任何对于真正的分布式系统的支持[○]。当然，在数据操纵方面不需要任何支持——（就用户而言）一个分布式系统最重要的是所有的数据操纵能力应该是不变的。但是对诸如 FRAGMENT、REPLICATE 之类的数据定义操作，这种支持是需要的[15.6]，然而现在却没有得到提供。

另一方面，SQL 又确实支持某些客户/服务器的功能，特别是包括用来建立和断开客户/服务器连接的 **CONNECT** 和 **DISCONNECT** 操作。实际上，在可以发出任何数据库请求之前一个 SQL 应用都必须先通过执行一个 CONNECT 操作来建立与服务器的连接（虽然这个 CONNECT 可能是隐式的）。一旦已经建立了连接，应用（即客户）就可以像通常一样发出 SQL 请求了，而服务器将进行所需要的数据库处理。

SQL 还允许已经和一个服务器建立连接的客户与另一个服务器连接。建立第二个连接会使第一个连接进入**休眠**状态，接下来的 SQL 请求就由第二个服务器来处理，直到客户要么（a）重新和上一个服务器连接（通过另外一个新的操作，**SET CONNECTION**）；要么（b）再和另外一个服务器进行连接，从而使第二个服务器进入休眠状态（依此类推）。换句话说，在任何时候一个给定的客户都只能有一个**活动**的连接和若干个**休眠**的连接，而由该客户发出的所有数据库请求都会发到与之处于活动连接的服务器上，并由其处理。

注意：SQL 标准还允许（但是并不要求）实现对多服务器事务的支持。也就是说，在一个事务中间，客户可以从一个服务器转向另一个服务器，从而使事务的一部分在一个服务器上执行而另一部分在另一个服务器上执行。尤其要注意的是如果允许更新事务以这种方式跨越服务器的话，那么执行必须假定支持某种两阶段提交，以提供标准所要求的事务的原子性。

最后，一个给定客户所建立的每一个连接（不管当前是活动的还是休眠的）最终都必须由一个合适的 DISCONNECT 操作来切断（虽然像相应的 CONNECT 一样，DISCONNECT 在简单的情况下也许是隐式的）。

要了解更多的内容——尤其是生成存储过程的 SQL 工具，请参看 SQL 标准文本[4.23，4.24]或者参考文献[4.20]中的内容。

## 21.8　小结

在这一章里，我们给出了一个关于分布式数据库系统的简要讨论。以分布式系统的“**十二个目标**”[21.13]作为组织讨论的基础，虽然这些目标并不是在所有情形下都是关联在一起的。还简要地考察了一些领域的技术问题，这些领域包括**查询处理**、**目录表管理**、**更新传播**、**恢复控制**和**并发控制**。还特别讨论了为了满足 **DBMS 独立性**目标所涉及的问题（在 21.6 节对于**通道**和**数据存取中间件**以及**联合系统**的讨论）。我们又认真地探讨了**客户/服务器**处理过程，可以将它看作是一般意义下分布式处理的特殊情况。最后对 SQL 中与客户/服务器处理有关的方面进行了

---

○ 不过，标准中的第 9 部分 SQL/MED（Management of External Data）[4.23] 的确提供了对联合系统的支持。细节超出了本书的范围，请参阅文献[26.32]中的手册。

汇总，并且着重指出用户要**避免针对记录层次编码**（在 SQL 术语中叫游标操作）。还简要描述了**存储过程**和**远程过程调用**的概念。

注意：有一个始终没有讨论的问题是分布式系统的（物理）**数据库设计问题**。实际上，即使忽略分片和/或复制的可能性，决定哪些数据应该存储在**哪些场地上**的问题——所谓的分配问题——也是一个非常棘手的难题［21.31］。对于分片和复制的支持只会使情况进一步复杂。

值得一提的另外一点是那些在市场上逐渐崭露头角的所谓的海量并行计算机系统（参见对文献［18.56］的注释）。这类系统基本上都是由大量通过高速总线连接在一起的独立的处理器构成的，每个处理器都有自己的存储器和磁盘驱动器，并且运行自己的 DBMS 软件，整个数据库分布在所有的磁盘上。换句话说，这样一个系统本质上是“在一个盒子里”构成了一个分布式系统！而在这一章中所讨论的（比如）关于查询处理策略、两阶段提交、全局死锁等所有的问题，都可以对应到这类系统中。

作为结论，我们指出，把分布式数据库的“十二个目标”（或者至少包括目标 4、5、6 和 8 的某些子集）放在一起，它们看起来和 Codd 所说的关系 DBMS 的“分布独立性”准则［10.3］是等价的。为了便于对照，我们把该准则叙述如下：

- 分布独立性（Codd）：“一个关系 DBMS 要具有分布独立性……（就是说）DBMS 要有一种数据子语言保证应用程序和（终端）操作能够保持在逻辑上没有损失：
  - 在第一次引入数据分布的时候（如果本来安装的 DBMS 只管理非分布的数据）；
  - 在数据被重新分布的时候（如果 DBMS 管理的是分布的数据）。

最后要注意（如同在本章 21.3 节中提到的）目标 4～6 以及 9～12——即所有在名字中包括“独立性”一词的目标，它们可以被看作是类似数据独立性之类的概念——在应用到分布式环境的时候——的扩展。其实它们的目的都在于**对应用投资的保护**。

## 习题

21.1　用你自己的语言来解释位置独立性、分片独立性以及复制独立性的概念。

21.2　为什么分布式数据库系统几乎总是关系的？

21.3　分布式系统的优点是什么？不足又是什么？

21.4　解释下列术语：

主副本修改策略
主副本封锁策略
全局死锁
两阶段提交
全局优化

21.5　描述 R* 的数据对象命名模式。

21.6　一个成功的点对点通道取决于能否很好地协调（尤其是）它所涉及的两个 DBMS 之间的差异。任选两个你熟悉的 SQL 系统，尽可能多地指出它们之间的差异，包括语法和语义之间的差异。

21.7　调查任意一个你可以接触到的客户/服务器系统，它支持显式的 CONNECT 和 DISCONNECT 吗？它支持 SET CONNECTION 或其他“连接类型”的操作吗？它支持多服务器事务吗？它支持两阶段提交吗？它用于客户/服务器通信的格式和协议是什么？它支持什么样的网络环境？它支持什么样的客户与服务器硬件平台？它所支持的软件平台（操作系统、DBMS）又是什么？

21.8　调查任意一个你可以接触到的 SQL DBMS，这个 DBMS 支持存储过程吗？如果支持的话，存储过程是如何创建的？它们又是如何被调用的？它们是用什么语言写的？它们是否支持完整的 SQL？它们是否支持条件分支结构（IF-THEN-ELSE）？它们是否支持循环？它们如何向客户返回结果？一个存储过程可以调用另一个存储过程吗？如果是在另外一个场地上的存储过程呢？存储过程是作为调用它的事务的一部分吗？

## 参考文献

［21.1］　Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool: “Replication, Consistency, and Practicality: Are These Mutually Exclusive?”, Proc. 1998 ACM SIGMOD Int. Conf. on Management of

Data, Seattle, Wash. (June 1998).

这篇论文描述了异步（文章中称为懒惰式）复制模式中的三种模式，这些异步复制模式可以确保事务的原子性和全局可串行性而不需要利用两阶段提交。文章中还给出了有关它们的性能比较的模拟研究报告。在文献 [21.18] 中提出的全局封锁是第一种模式；另外两种，其中一个是悲观式的，另一个是乐观式的，利用了一张复制图。这篇论文的结论是复制图模式的性能要明显地优越于封锁模式，而且“通常是大大地优于”。

[21.2] David Bell and Jane Grimson: *Distributed Database Systems*. Reading, Mass.: Addison-Wesley (1992).

这是众多分布式系统的教科书中的一本（另外两本是文献 [21.10] 和 [21.29]）。这本书的一个显著的特点是包含了一个关于保健网络的扩展案例研究。而且在论调上也要比另外两本实用一些。

[21.3] Philip A. Bernstein: “Middleware: A Model for Distributed System Services,” *CACM 39*, No. 2 (February 1996).

“对各种不同的中间件进行了归类，描述了它们的特性，揭示了它们的演化过程，并为理解今天以及明天的分布式系统软件提供了一个概念模型”（选自摘要）。

[21.4] Philip A. Bernstein *et al.*: “Query Processing in a System for Distributed Databases (SDD-1),” *ACM TODS 6*, No. 4 (December 1981).

参见文献 [21.32] 的注释。

[21.5] Philip A. Bernstein, David W. Shipman, and James B. Rothnie, Jr: “Concurrency Control in a System for Distributed Databases (SDD-1),” *ACM TODS 5*, No. 1 (March 1980).

参见文献 [21.32] 的注释。

[21.6] Charles J. Bontempo and C. M. Saracco: “Data Access Middleware: Seeking Out the Middle Ground,” *InfoDB 9*, No. 4 (August 1995).

一本以 IBM 的 DataJoiner 为重点的有用教程（虽然也提到了其他的产品）。

[21.7] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz: “Overview of Multi-Database Transaction Management,” *The VLDB Journal 1*, No. 2 (October 1992).

[21.8] M. W. Bright, A. R. Hurson, and S. Pakzad: “Automated Resolution of Semantic Heterogeneity in Multi-Databases,” *ACM TODS 19*, No. 2 (June 1994).

[21.9] Michael L. Brodie: “Data Management Challenges in Very Large Enterprises” (panel outline), Proc. 28th Int. Conf. On Very Large Data Bases, Hong Kong (August 2002).

主要的讨论内容是信息集成问题：“主要（问题）是（信息）集成。最近的分析研究表明，IT 预算开支里超过 40% 都用于新系统与现存系统和数据库之间的集成…… 集成问题面临很大的挑战和花费。

[21.10] Stefano Ceri and Giuseppe Pelagatti: *Distributed Databases: Principles and Systems*. New York, N. Y.: McGraw-Hill (1984).

[21.11] William W. Cohen: “Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity,” Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

描述了针对通常所说的“垃圾邮件问题”的一个解决方法——即当两个不同的文本串，比如“AT&T Bell Labs”和“AT&T Research”，所指的是同一个对象的时候，要把它们识别出来（当然这是一种特定的语义不匹配）。这个方法主要是推导出这些文本串的相似性，这些文本串是“用通常在统计信息检索中采用的向量空间模型来度量的”。按照论文的说法，这种解决方法要比“朴质推论方法”（naive inference method）快得多，而且惊人地准确。

[21.12] D. Daniels *et al.*: “An Introduction to Distributed Query Compilation in R*,” in H. -J. Schneider (ed.), *Distributed Data Bases*: Proc. 2nd Int. Symposium on Distributed Data Bases (September 1982). New York, N. Y.: North-Holland (1982).

参见文献 [21.37] 的注释。

[21.13] C. J. Date: “What Is a Distributed Database System?” in *Relational Database Writings 1985-1989*, Reading, Mass.: Addison-Wesley (1990).

这篇论文介绍了分布式系统的“十二个目标”（21.3 节基本上是直接根据这篇论文安排的）。像在本章中提到的，本地自治的目标不是可以 100% 达到的，确实存在某些情况需要对这

一目标进行某种程度上的折衷。为了便于参照，我们在这里归纳一下这些情况：

- 一个被分片关系的个别分片无法被正常地直接存取，甚至从它们所存储的场地上存取也不行。
- 一个被复制关系（或分片）的单独的副本无法被正常地直接存取，甚至从它们所存储的场地上存取也不行。
- 设 $P$ 是某个被复制关系（或分片）$R$ 的主副本，$P$ 存储在场地 $X$ 上。则每个要存取 $R$ 的场地都要依赖于场地 $X$，即使是在该场地上还存有 $R$ 的另一个副本。
- 对一个参与多场地完整性约束的关系，不能在存储它的本地场地中进行修改存取，而必须在定义约束的分布式数据库中去考虑它的修改存取。
- 在两阶段提交进程中作为一个参与者的场地必须遵循相应的协调者场地的决定（即提交或者回滚）。

参见文献［15.6］，这是本文的后续。

[21.14] C. J. Date: "Why Is It So Difficult to Provide a Relational Interface to IMS?" in *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

问题"我们能提供 IMS 的一个关系接口吗?"有两个可能的解释：

1）我们能构造一个将 IMS 用作存储管理器的关系 DBMS 吗?

2）我们能在 IMS 上面构造一个"包装器"（wrapper），使现存的 IMS 数据看起来像关系数据?

如果解释趋向于第一种，那么很明显答案是肯定的（尽管许多的 IMS 特征将不会用到）；如果趋向于后者，则正如文章所论证的，答案是否定的（至少不会达到 100%）。

[21.15] R. Epstein, M. Stonebraker, and E. Wong: "Distributed Query Processing in a Relational Data Base System," Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. (May/June 1978).

参见文献［21.34］的注释。

[21.16] Rob Goldring: "A Discussion of Relational Database Replication Technology," *InfoDB* 8, No. 1 (Spring 1994).

对于异步复制的一个很好的概述。

[21.17] John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis: "Query Languages for Relational Multi-Databases," *The VLDB Journal 2*, No. 2 (April 1993).

给出了关系代数和关系演算在多数据库系统下的扩展。其中讨论优化的问题，并且表明每一个多关系的代数表达式都有一个等价的多关系的演算表达式（"这个定理的逆命题是一个很有趣的研究问题"）。

[21.18] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha: "The Dangers of Replication and a Solution," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

"在工作负载扩充的情况下，在任何地方、任何时候、以任何方式修改事务级复制都会有不稳定的表现……这里提出了一个新的算法，允许（处在断开状态的）移动应用给出一个试探性的修改事务，这个修改可以在以后应用到主副本上"（选自摘要，并稍作了修改）。

[21.19] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham: "Revisiting Commit Processing in Distributed Database Systems," Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

本文提出了一个叫做 OPT 的新的分布式提交协议，这个协议（a）易于实现；（b）可以与传统协议并存；（c）"在各种工作负载与系统配置的情况下提供了最佳的事务吞吐性能"。

[21.20] Richard D. Hackathorn: "Interoperability: DRDA or RDA?", *InfoDB 6*, No. 2 (Fall 1991).

[21.21] Michael Hammer and David Shipman: "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM TODS 5*, No. 4 (December 1980).

参见文献［21.32］的注释。

[21.22] IBM Corporation: *Distributed Relational Database Architecture Reference*. IBM Form No. SC26-4651.

IBM 的 DRDA 对分布式数据库定义了如下四个层次的功能：远程请求、远程工作单元、分布式工作单元以及分布式请求。由于这些术语已经成为产业界事实上的标准（至少是在产业界的某些部分），我们在这里对它们做个简要的解释。注意："请求"和"工作单元"是 IBM 的术

语，分别对应于 SQL 语句和事务。

1）**远程请求**是指一个在场地 *X* 上的应用可以把一个单独的 SQL 语句发送到一个远程场地 *Y* 上去执行。这个请求完全在场地 *Y* 上执行和提交（或者回滚）。在场地 *X* 上的那个应用会接着发送另一个请求到场地 *Y* 上（也可能是到另一个场地 *Z* 上），而不管第一个请求是成功还是不成功。

2）**远程工作单元**（缩写为 RUW）是指一个在场地 *X* 上的应用可以在一个给定的“工作单元”（即事务）中把所有的数据库请求发送到一个远程场地 *Y* 上去执行。数据库对于该事务的处理也因此就完全在远程场地 *Y* 上进行，但是要由本地场地 *X* 来决定该事务是提交还是回滚。注意：RUW 实际上就是在单服务器情况下的客户/服务器处理过程。

3）**分布式工作单元**（缩写为 DUW）是指一个在场地 *X* 上的应用可以在一个给定的“工作单元”（即事务）中把某些或者所有的数据库请求发送到一个或者多个远程场地 *Y*，*Z*，…上去执行。因此一般而言，数据库对这个事务的处理是分散到许多场地上的，每个独立的请求还是在某个单独的场地上执行完成的，但是不同的请求可以在不同的场地上执行。不过场地 *X* 仍然是协调场地，即由这个场地决定事务是提交还是回滚。注意：DUW 实际上就是在多服务器情况下的客户/服务器处理过程。

4）最后，**分布式请求**是在这四个层次中唯一实现了通常所说的真正的分布式数据库支持的。分布式请求是在分布式工作单元的基础上，加上允许单独的数据库请求（SQL 语句）跨越多个场地——比如，一个来自于场地 *X* 的请求可以要求执行对分别来自于场地 *Y* 和 *Z* 上的表的连接或合并。要注意只有在这个层次上，系统才可以说是提供了真正的位置独立性。而在前面的三种情况下，用户都必须对数据的物理位置有所了解。

[21.23] International Organization for Standardization (ISO): *Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 1: Generic Model, Service, and Protocol (Draft International Standard)*. Document ISO DIS 9579-1 (March 1990).

[21.24] International Organization for Standardization (ISO): *Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 2: SQL Specialization (Draft International Standard)*. Document ISO DIS 9579-2 (February 1990).

[21.25] Donald Kossmann: "The State of the Art in Distributed Query Processing," *ACM Comp. Surv. 32*, No. 4 (December 2000).

[21.26] B. G. Lindsay *et al.*: "Notes on Distributed Databases," IBM Research Report RJ2571 (July 1979).

（由 R* 开发小组最初的部分成员所完成的）这篇论文分为五章：

1）复制的数据

2）授权与视图

3）对分布式事务管理的介绍

4）恢复功能

5）事务的启动、转移和终止

第 1 章讨论了更新传播的问题。第 2 章说的几乎完全都是在一个非分布式系统（像 System R 那种类型的系统）中授权的问题，只是在结尾的时候才作了一些其他的说明。第 3 章考虑的是事务的启动和终止、并发控制以及恢复控制，都很简要。第 4 章用来探讨（同样是）在非分布式情况下的恢复。第 5 章较为具体地讨论了分布式事务管理，特别地，还给出了一个非常认真的关于两阶段提交的表述。

[21.27] C. Mohan and B. G. Lindsay: "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," Proc. 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (1983).

参见文献 [21.37] 的注释。

[21.28] Scott Newman and Jim Gray: "Which Way to Remote SQL? " *DBP&D 4*, No. 12 (December 1991).

[21.29] M. Tamer Özsu and Patrick Valduriez: *Principles of Distributed Database Systems* (2d ed.). Englewood Cliffs, N. J.: Prentice-Hall (1999).

[21.30] Martin Rennhackkamp: "Mobile Database Replication," *DBMS 10*, No. 11 (October 1997).

价格低廉、非常便于携带的计算机和无线网络之间的结合，使得一种新的分布式数据库系统的出现成为可能，它不仅能带来特别的收益，（当然）也会带来特殊的问题。尤其是这种系统

中的数据说起来是可以复制到成千个“场地”上去——但是这些场地都是移动的，并且是经常掉线的，它们的操作特点更是与较为传统的场地非常不同（比如在通信开销中还要考虑电池的使用率和连接的时间）等等。对于这类系统的研究是最近开始的（相关文献有［21.1］和［21.18］），而这篇短文强调了其中的一些主要概念和问题。

［21.31］ James B. Rothnie, Jr., and Nathan Goodman: “A Survey of Research and Development in Distributed Database Management,” Proc. 3rd Int. Conf. on Very Large Data Bases, Tokyo, Japan (October 1977).

一篇非常有用的早期综述，分以下几个方面进行了讨论：

1）修改事务的同步
2）分布式查询处理
3）部件故障处理
4）目录管理
5）数据库设计

在这些的最后提到了物理设计问题，即我们在21.8节中所说的分配问题。

［21.32］ J. B. Rothnie, Jr., *et al.*: “Introduction to a System for Distributed Databases (SDD-1),” *ACM TODS 5*, No. 1 (March 1980).

文献［21.4，21.5，21.21，21.32］都是关于早期的分布式原型SDD-1的。SDD-1运行在一组通过Arpanet（见第27章27.2节）互连的DEC PDP-10上，它提供了完全的位置独立性、分片独立性和复制独立性。下面我们选择这个系统的某些方面做一些说明。

**查询处理**：SDD-1的查询优化器（见文献［21.4］）大范围地利用了第7章7.8小节中所描述的半连接操作。在分布式查询处理中使用半连接操作的好处是它们具有减少网络中数据传输量的作用。比如，假设供应商关系变量 *S* 存储在场地 *A* 上，发货关系变量 *SP* 存储在场地 *B* 上，而查询是“对供应商关系和发货关系进行连接”。可以不用（比如说）把整个 *S* 传送到 *B* 上，而是像下面这样做：

- 计算在场地 *B* 上对 *SP* 的 *S#*的投影 *TEMP*1。
- 将 *TEMP*1 发送到场地 *A*。
- 在场地 *A* 上计算 *TEMP*1 和 *S* 关于 *S#*的半连接，得到 *TEMP*2。
- 将 *TEMP*2 发送到场地 *B*。
- 在场地 *B* 上计算 *TEMP*2 和 *SP* 关于 *S#*的半连接。

这个过程会明显减少通过网络传输的数据总量，当且仅当

```
size (TEMP1) + size (TEMP2) < size (S)
```

这里一个关系的“大小”（size）是指这个关系中元组的数目与一个单独元组的长度（比如用位来计算）的乘积。显然优化器要能够估计中间结果，比如说TEMP1和TEMP2的大小。

**更新传播**：SDD-1的更新传播算法是“立即传播”（这里没有主副本的概念）。

**并发控制**：并发控制是基于所谓的时戳技术，而不是封锁技术。其目的是为了避免与封锁相关的消息开销，但是代价似乎是实际上并没有真正地并发！有关细节已经超出了本书的讨论范围（不过参考文献［16.3］的注释确实非常简要地描述了一下基本的想法）。更多的内容可以参见文献［21.5］。

**恢复控制**：恢复是基于一个四阶段提交协议，目的是为了使在协调者场地上的处理能够比传统的两阶段提交协议更有弹性，但是，遗憾的是它还是使得处理在相当大的程度上变得更为复杂。（同样）细节的讨论超出了本书的范围。

**目录表**：目录表是当作普通的用户数据来管理的——它可以被任意地分片，而分片也可以被任意地复制和重新分布，就像其他数据一样。这种方法的优点是显而易见的。当然缺点就是，由于系统没有任何关于一个给定部分的目录表的位置的先验信息，它必须维护一个更高级别的目录表——**字典定位器**——以提供这样的确切信息！这个字典定位器是全复制的（即在每个场地上都存有一个副本）。

［21.33］ P. G. Selinger and M. E. Adiba: “Access Path Selection in Distributed Data Base Management Systems,” in S. M. Deen and P. Hammersley (eds.), Proc, Int. Conf. on Data Bases, Aberdeen, Scotland (July 1980). London, England: Heyden and Sons Ltd. (1980).

参见文献［21.37］的注释。

［21.34］ M. R. Stonebraker and E. J. Neuhold: "A Distributed Data Base Version of Ingres," Proc. 2nd Berkeley Conf. on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory (May 1977).

文献［21.15, 21.34, 21.35］都是关于分布式 Ingres 原型的。分布式 Ingres 由许多 University Ingres 的副本构成，运行在许多互连的 DEC PDP-11 上。它（和 SDD-1 及 R* 一样）支持位置独立性；它也支持具有分片独立性的数据分片（不是通过投影，而是通过选择），以及具有复制独立性的对这些分片的数据复制。与 SDD－1 和 R* 不同，分布式 Ingres 并不假设通信网络的速度很慢；相反，它被设计为对于"缓慢的"（远程的）网络和本地的（即相当快的）网络都可以处理（优化器明白这两种情况的差异）。查询优化算法基本上是本书第 18 章中所说的 Ingres 分解策略（Ingres decomposition strategy）的扩展，在文献［21.15］中有关于它的详细叙述。

分布式 Ingres 提供两种更新传播算法：一个是"性能优先"算法，它只是修改主副本然后就将控制交还给事务（而将修改的传播交由若干个子进程并行地执行）；另一个是"可靠"算法，它立即修改所有的副本（见文献［21.35］）。并发控制在两种情况下都是基于封锁的，而恢复控制是基于改进的两阶段提交。

对于目录表而言，分布式 Ingres 采用了把对数据库某些部分的目录表进行全复制和对其他部分使用纯粹的本地目录表记录相结合的方法。进行全复制的部分主要包括一个所有用户可见的关系变量的逻辑说明和一个关于关系变量如何进行分片的说明，其他部分包括对本地物理存储结构、本地数据库的统计信息（供优化器使用）以及安全性与完整性约束的说明。

［21.35］ M. R. Stonebraker: "Concurrency Control and Consistency of Multiple Copies in Distributed Ingres," *IEEE Transactions on Software Engineering 5*, No. 3 (May 1979).

参见文献［21.34］的注释。

［21.36］ Wen-Syan Li and Chris Clifton: "Semantic Integration in Heterogeneous Databases Using Neural Network," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

［21.37］ R. Williams *et al.*: "R*: An Overview of the Architecture," in P. Scheuermann(ed.), *Improving Database Usability and Responsiveness*. New York, N. Y.: Academic Press (1982). Also available as IBM Research Report RJ3325 (December 1981).

文献［21.12, 21.27, 21.33, 21.37］都是关于 R* 的，它是最初的 System R 原型的分布式版本。R* 提供了位置独立性，但是没有分片和复制，因此也就不具备分片独立性和复制独立性。同样的原因，更新传播的问题也就不存在了。并发控制是基于封锁的（要注意任何被封锁的对象都只有一份，主副本问题也不存在）。恢复控制是基于改进的两阶段提交。

［21.38］ Ling Ling Yan, Renée J. Miller, Laura M. Haas, and Ronald Fagin: " Data-Driven Understanding and Refinement of Schema Mappings," Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

# 第22章 决策支持

## 22.1 引言

注意：本章的原作者为 David McGoveran。

**决策支持系统**是用于对企业信息进行辅助分析的系统。系统的目的是帮助管理者“发现趋势、查明问题并进行智能化的决策”[22.9]。这种系统的来源是业务的研究、管理的行为和科学理论以及统计处理控制，等等，可以追溯到20世纪40～50年代，那时计算机还没有被广泛使用。系统的基本思想是从企业操作型数据中收集（参见第1章）并精简信息，使之能用来分析企业行为，并智能地改变企业行为。当时的客观条件下人们只能将数据精简到极小，常常会只比简单的摘要报告稍多一点。

到了20世纪60年代后期和70年代早期，哈佛大学和MIT的研究人员促进了计算机在决策处理中的应用[22.26]。开始时这种应用局限于自动化报表生成，虽然当时已经具备了某些基本的分析能力[22.6－22.8]。早期的这些计算机系统被称为管理决策系统；后来发展为管理信息系统。不过我们更愿意使用现代的术语——决策支持系统，因为管理信息系统的概念过于抽象，包括OLTP系统（OLTP = Online transaction processing）在内的所有的信息系统都可以被称为“管理信息系统”（毕竟，这些系统都包括在企业管理中并且对其产生影响）。接下来的内容我们将采用现代的术语“决策支持系统”。

在20世纪70年代开发出了许多查询语言，围绕这些查询语言建立了许多特定的决策支持系统，使用报表生成子（如RPG）或数据获取工具（如Focus、Datatrieve和NOMAD）来实现。这些系统首次让具有一定技能的最终用户直接访问计算机数据存储；也就是说，让这些用户在数据存储上阐明商务相关查询，并直接执行这些查询，而无需IT技术人员的协助。

当然，上面提到的数据存储一般是一些简单文件。那时大多商业数据保存在这种文件中，后来保存在非关系型数据库中（当时关系型数据库正在研究之中）。即便在后来，人们也常常需要在数据库中将数据提取出来，然后拷贝到文件中以便被决策支持系统使用。直到80年代，关系型数据库才开始取代简单文件在决策支持系统中的位置（实际上，决策支持、特定查询和报表都出现于关系技术在商业上的早期应用中）。尽管SQL产品现在已经得到了广泛的应用，**抽取处理**（extract processing）的思想（即从操作型环境中复制数据到其他环境）仍然是很重要的；它允许用户按自己的意愿来操作这些抽取出来的数据，而无须和操作型环境打交道。显然，在决策支持中需要经常进行这种抽取工作。

从以上的简短发展史可以明显看出，决策支持实际上并不属于数据库技术，它只是（尽管是很重要的一个）数据库技术的应用，更准确地说，它是由多个这种应用所组成，这些应用彼此独立而又相互关联。目前正在研究的相关问题包括数据仓库、数据集市、操作型数据存储、联机分析处理（OLAP）、多维数据库和数据挖掘等，我们将在以后的章节中讨论这些相关内容。不过，我们很快会注意到在以上的这些领域中存在一个共同点，就是很少会用到良好的逻辑设计原则。决策支持经常是相当特定的应用，此时更需要多考虑物理实现中的细节，对逻辑设计则不会要求那么严格。的确，决策支持趋向于在很大程度上模糊了逻辑和物理上的区别。由于有这些因素，本章中使用SQL，而不是**Tutorial D**，作为示例的基础。同时使用“更模糊的”SQL术语，如行、列和表，而不是元组、属性、关系值和变量（关系变量）。另外还使用逻辑模式和物理模式，作为第2章中的概念模式和内模式的同义词。

本章的各节是这样安排的：在22.2节中讨论决策支持的某些特征，这些特征容易引起误导的设计。在22.3节中给出处理这些问题的方法。在22.4节对数据准备（将操作型数据转换为可供决策支持使用的形式）的结果进行检验，同时简要介绍“操作型数据存储”。在22.5节中讨论数据仓库、数据集市和“多维模式”。在22.6节中讨论联机分析处理（OLAP）和多维数据

库。在22.7节中讨论数据挖掘。在22.8节中简略介绍了相关的SQL工具。最后，在22.9节中给出本章小结。

## 22.2 决策支持的特征

决策支持数据库具有一些特征，最主要的一点是：这种数据库主要（并不是完全）是只读的。更新操作一般仅限于周期性的上载和刷新操作，即偶尔的INSERT-DELETE操作，几乎没有UPDATE操作。（有时会更新某些辅助的工作表，但是一般的决策支持处理基本上不会更新决策支持数据库本身。）

决策支持数据库中还有一些值得关注的特征（将在22.3节中予以详细阐述），前三个为逻辑特征，后三个为物理特征。

- 列常常在组合中使用。
- 一般不考虑完整性约束（假定数据在载入时就是正确的，以后不再更新）。
- 码中常常包含时间成分（适当的时间支持通常非常重要——见第23章，但这样的支持很少）。
- 数据库会变得很庞大，特别是随着时间推移（常常会这样），企业事务[⊖]细节数据会不断增加。
- 数据库中建立索引的负担非常繁重。
- 数据库中包含多种受控冗余（controlled redundancy）。

决策支持查询也有自己的特征，特别是它们会变得相当复杂。复杂性中包括：

- 布尔表达式复杂性：决策支持查询的WHERE子句经常会涉及复杂表达式—难以书写、难以理解而且系统也难以有效实现。一个常见的问题是涉及时间的查询（比如，查询出在指定时间间隔中最大时间戳所在的行）。如果查询中还包括连接操作，尤其是没有适当的时间支持时，那就会变得非常复杂。这时的性能无疑是很差的。
- 连接复杂性：决策支持查询常常需要访问多种事实。在完全规范化设计的数据库中，这种查询将涉及大量连接操作。不幸的是，连接处理技术无法满足不断增长的决策支持查询需求。[⊖]因此设计人员会对数据库进行逆规范化，预连接（prejoining）某些表。但是，正如第13章中所提到的，这种方法很难奏效（有时解决了部分问题，却同样会引来另外一些问题）。而且这种避免连接的期望使得我们无法有效地使用关系操作，当从数据库中检索大量数据时，连接处理必须在应用中完成，而不是由DBMS来完成这项工作。
- 函数复杂性：决策支持查询经常会涉及到统计函数和其他数学函数，很少有产品支持这些函数（尽管这种状况正有一些改观）。因此查询常常被划分为一系列子查询，然后在其中插入一些用户编写的过程来进行所需的计算。这种方法的缺点是需要检索大量的数据，从而导致整个查询变得难以编写和理解。
- 分析复杂性：企业问题很少能在单个查询中予以解答。一方面因为用户难以编写特别复杂的查询，另一方面是因为SQL的实现中存在某些限制，从而无法处理这种查询。解决方法之一是将这种查询划分为一系列子查询，并将中间结果保存在辅助表中。

以上列举了决策支持数据库和决策支持查询的特征，讨论的重点集中于有关性能的设计，特别是批插入和特定检索的性能。不过这种情况只应该影响到数据库的物理设计，而非逻辑设计（在下一节详细阐述）。不幸的是，如同在22.1节中提到的，决策支持系统的销售商和用户常常

---

⊖ 在这里以及本章中的其他位置，当事务是企业的一个事务时（除非上下文指的是本书第四部分中讨论的事务），我们将通过加“企业”这一修饰词来区别企业事务（比如，产品销售）和本书第四部分中讨论的事务含义。

⊖ 作者（McGoveran）曾于1981年研究过早期的决策支持系统。那时他注意到一个涉及三个中等大小的表连接的时候会花费多个小时。4个到6个表连接的时候代价是相当大的。现在，6个到10个的大表连接是很常见的，而且技术处理的也相当好。然而，仍然常常需要更多个表的连接，当前的技术还不能很好地处理。很快人们将会研究涉及到12个乃至更多个表的连接，因为现在对这样的查询的需求是很常见的！注意：参考附录A去寻求这个问题的一个可能的解决方案。

无法正确区分逻辑和物理问题[⊖]，实际上他们经常会完全放弃逻辑设计。这样的后果是，为了处理上面所讨论的不同特征导致了特殊化，并且在平衡正确性、可维护性、性能、缩放性和可用性需求时会遇到难以克服的困难。

## 22.3　决策支持的数据库设计

在本书的前面部分（第三部分引言）中已经说明了我们的观点，即数据库设计至少应分为两个阶段，逻辑设计和物理设计：

a. 首先应该进行逻辑设计。在这个阶段，重点放在关系正确性（relational correctness）上。表要表达适当的关系，从而保证关系操作的正确执行。要指定域（类型），然后在域上定义列，这样列之间的依赖关系（比如 FD）就可以识别出来。最后进行规范化，定义适当的完整性约束。

b. 然后根据逻辑设计来进行物理设计。在这个阶段，重点放在存储效率和性能上。从理论上讲，任何物理方案都是可行的，只要在逻辑和物理模式之间存在一种保持信息的变换，能用关系代数予以表达（见参考文献［2.5］）。正是由于这种变换的存在，保证了在物理模式中存在一些关系型视图，使得它看起来与逻辑模式相似，反之亦然。

当然，逻辑模式会不断变化（例如，为了适应新的数据或新发现的依赖关系时），从而要求物理模式作相应的变化。我们关注的是当修改物理模式的时候，能否保持逻辑模式不变。例如，假设表 SP（发货）和 P（零件）之间的连接是目前的主要访问模式，我们准备在物理层“预连接”表 SP 和 P，从而降低 I/O 和连接开销。但是，如果要实现物理数据独立性，逻辑模式就应该保持不变（当然，如果我们希望性能得到提升，就必须让查询优化器知道“预连接”的存在，并适当地使用它）。此外，如果以后访问模式改为只访问单个表而不是多表连接，我们应该能够再次改变物理模式，使得表 SP 和 P 在物理上分离，并且不会影响到逻辑层。

以上的阐述表明，保持物理数据独立性的问题基本上就是支持视图更新的问题。除此之外，正如在第 21 章中讨论的分片更新问题，在整个系统架构上它与它的表现有所不同。特别地，我们需要能够更新这些视图。其实是，如果（a）我们把逻辑层的基表看作视图，同时把物理层上这些“视图”相应的存储版本看作基表，那么（b）物理模式必须要使得 DBMS 能够按照“基表”执行这些“视图”更新。

从第 10 章我们看到，理论上所有的视图都是可更新的。同样，在理论上，如果从逻辑模式导出满足以上阐述的物理模式，就可以实现最大的物理数据独立性：任何在逻辑模式中的更新都可以自动转换为相应的物理模式中的更新，反之亦然。但是物理模式的改变并不会要求改变逻辑模式。但是，现有的 SQL 产品不能正确地支持视图更新。因此，在这些产品中就只能（并且不是必要的）考虑可允许的物理模式。注意：实际上，我们可以使用存储过程、触发器、中间件或者它们的组合来模拟正确的视图更新机制。但是，这些技术本章不作讨论。

### 1. 逻辑设计

逻辑设计的规则与数据库的用途无关，对于不同的应用使用相同的规则。因此，在 OLTP 或决策支持应用中的数据库逻辑设计没有什么不同：它们实际都采用相同的步骤。现在回顾一下第 22.2 节中提到的决策支持数据库的三个逻辑特征，以及它们对逻辑设计所产生的影响。

■ 列的组合和更少的依赖

决策支持查询（可能还有更新）常常把列的组合视为一个整体，组合中的列不会被单独访问（地址就是一个很明显的例子）。称这种列的组合为复合列。从逻辑设计中视图的观点来看，这种复合列可以被视为没有组合的简单列。更具体地说，令 CC 为复合列，C 是同一个表中的简单列，那么 C 和 CC 中的组件之间的依赖关系实际上可归纳为 C 和 CC 之间的依赖关系。进一步来说，仅涉及 CC 中的组件之间的依赖关系是不相关的，可以被忽略。这样依赖关系的总数会减

---

⊖ 数据仓库和 OLAP 的专家在这方面有一定的责任。他们常指出关系设计对于决策支持是完全不合适的，关系模型不能够表现数据，因此绝不能采用。这种观点将很容易导致人们无法区别关系模型与它的实现。

少，逻辑设计变得更简单，涉及到的列和表的数目也会减少。

■ 一般的完整性约束

由于前文阐述的原因：(a) 决策支持数据库主要是只读的；(b) 当数据库中数据上载或刷新时进行数据完整性检验。大家可能会认为在逻辑模式中无须声明完整性约束。实际上并不是这样的。即使数据库真的是只读的，可以保证完整性约束，但是约束中还包含了不可忽略的语义信息。第 9 章中提到，约束是用来定义表和整个数据库的语义。约束的声明告诉用户数据的含义，从而帮助他们准确描述查询。同时约束的声明还可以向优化器提供至关重要的信息（参见第 18 章关于语义优化的探讨）。

注意：当在一些产品中声明某些约束时，会自动创建索引和其他的强制机制，从而增加数据上载和刷新的开销。因此，设计人员应避免约束的声明。但是，这个问题是由于混淆了逻辑和物理问题：可以在逻辑层单独指定完整性约束的声明，在物理层指定相应的强制机制。可惜现在的产品还不能将逻辑层和物理层完全分离，并且它们根本不能识别约束的语义值。

■ 时态码

操作型数据库一般只涉及当前数据，决策支持数据库一般涉及历史数据，因此会给几乎所有的数据盖上时间戳。这样在决策支持数据库的码中常常包含时间戳列。例如，在前面提到的供应商 - 零件数据库，假如需要显示每次发货时的月份（1 ~ 12 月）。发货表 SP 应该如图 22-1 所示。新增的列 MID（month ID）加入到表 SP 的码中。关于表 SP 的查询中也应指定时间戳。在 22.2 节中会给出简略描述，第 23 章中进行了深入讨论。

注意：为了在码中加入时间戳列可能需要重新设计数据库。例如，假设每次发货的数量是由发货的月份所决定的（图 22-1 中的示例数据符合这种约束）。在表 SP 中就存在函数依赖 MID→QTY，因此它就不符合第五范式，需要进一步规范化为图 22-2 中的形式。不幸的是，决策支持设计人员很少会关心这种传递依赖。对此，第 23 章中进行了深入讨论。

SP

| S# | P# | MID | QTY |
|---|---|---|---|
| S1 | P1 | 3 | 300 |
| S1 | P1 | 5 | 100 |
| S1 | P2 | 1 | 200 |
| S1 | P3 | 7 | 400 |
| S1 | P4 | 1 | 200 |
| S1 | P5 | 5 | 100 |
| S1 | P6 | 4 | 100 |
| S2 | P1 | 3 | 300 |
| S2 | P2 | 9 | 400 |
| S3 | P2 | 6 | 200 |
| S3 | P2 | 8 | 200 |
| S4 | P2 | 1 | 200 |
| S4 | P4 | 8 | 200 |
| S4 | P5 | 7 | 400 |
| S4 | P5 | 11 | 400 |

图 22-1　表 SP 中的示例数据，包含 month Ids

SP

| S# | P# | MID |
|---|---|---|
| S1 | P1 | 3 |
| S1 | P1 | 5 |
| S1 | P2 | 1 |
| S1 | P3 | 7 |
| S1 | P4 | 1 |
| S1 | P5 | 5 |
| S1 | P6 | 4 |
| S2 | P1 | 3 |
| S2 | P2 | 9 |
| S3 | P2 | 6 |
| S3 | P2 | 8 |
| S4 | P2 | 1 |
| S4 | P4 | 8 |
| S4 | P5 | 7 |
| S4 | P5 | 11 |

MONTH_QTY

| MID | QTY |
|---|---|
| 1 | 200 |
| 2 | 600 |
| 3 | 300 |
| 4 | 100 |
| 5 | 100 |
| 6 | 200 |
| 7 | 400 |
| 8 | 200 |
| 9 | 400 |
| 10 | 100 |
| 11 | 400 |
| 12 | 50 |

图 22-2　对图 22-1 规范化以后的结果

**2. 物理设计**

在 22.2 节中提到决策支持数据库趋向于大型数据库，索引任务繁重，而且涉及多种受控冗余（controlled redundancy）。下面详细讨论物理设计问题。

首先考虑分区（也称为分片）。分区是处理“大型数据库”的一种方法；基于物理存储的考虑，它将给定的表划分为不相交的分区或分片（参见第 21 章中关于分片的描述）。分区可以有效地提高被查表的可管理性和可用性。一般给每个分区分配一定的专用资源（如磁盘空间、CPU），并尽量减少分区之间的资源竞争。按照分区函数将表水平划分[⊖]，以选取列的值（分区的码）作为参数，返回分区号或地址。这种函数一般支持区域、哈希和循环分区（参见第 18 章

⊖ 垂直划分尽管可能有优势，但由于很少有系统支持，因此用的不多。

中的参考文献［18.56］的注释）。

现在来考虑**索引**。众所周知，使用合适的索引可以显著减少 I/O 的负担。大多数早期的 SQL 产品只提供一种索引，即 B 树，不过现在出现了更多的索引技术，特别是在决策支持数据库中，包括位图、散列、多表、布尔、函数型索引和原有的 B 树索引。下面简略地进行说明：

- *B 树索引*：B 树索引能提高区域查询的效率（除非要访问的行数过多）。对 B 树的更新也相当有效。
- *位图索引*：假设表 $T$（已建索引）中有 $n$ 行，$C$ 是 $T$ 中的列，$C$ 上的位图索引是一个 $n$ 位的向量，与 $C$ 所在值域中的每一个值对应。根据第 $r$ 行的 $C$ 列的值来设置索引相应的位。当值域比较小时可以明显提高查询效率。一些关系运算（连接、并、相等约束等）可以通过对索引进行简单的位运算（与、或、非）即可得到。只有当获取最终的检索结果时才会访问表中的实际数据。
- *哈希索引*（哈希访问或哈希）：使用哈希索引可以提高访问特定行（不是某些区域）的效率。只要哈希函数不需要其他的键值，计算的开销只会随着行数的增加线性增长。哈希技术也可以提高连接效率，参见第 18 章。
- *多表索引*（连接索引）：一般说来，多表索引中包含了指向多表中的记录的指针，而不是只限于单个表。多表索引可以提高表连接的效率，并检验多表（即数据库）完整性约束。见参考文献［22.33］。
- *布尔索引*（表达式索引）：布尔索引显示表中的哪些行对于特定的布尔表达式为真。当布尔表达式是某些约束条件的公共部分时，布尔索引会很有用。
- *函数型索引*：函数型索引不是简单的行值的索引，而是这些行值对应的特定函数值的索引。

除了上述的这些索引，有人还提出了混合索引（由上述索引组合而成），很难将它归类。另外还有一些专用索引（如 R 树，用于处理几何数据）。本书中不讨论这些索引，详细介绍见参考文献［26.37］。

接下来，我们来讨论**受控冗余**。受控冗余用来减少 I/O 和资源争用。第 1 章中已经提到，这种冗余是由 DBMS 管理而对用户透明（*注意*：冗余是完全在物理层而不是逻辑层被控制，因此不会影响逻辑层的正确性）。受控冗余分为两类：

- 第一类涉及到原始数据的精确制品（replica）。*注意*：除此之外同样存在着非精确的复制，即复制管理。（见下一小节）
- 第二类涉及到除原始数据外的导出数据，一般形式为汇总表或计算（或导出）列。

下面的子小节分别讨论每一种的可能性。

### 3. 复制

在 21.3、21.4 节中已经介绍了关于**复制**（replication）的基本概念（特别在第 21.4 节中的“更新传播”小节中予以详细阐述）；这里只重复讨论几个要点并予以评论。首先要提到的是复制可以是同步的，也可以是异步的。

- 在同步的情况下，如果某个给定复制品被更新了，那么包含重叠数据的其他复制品也应该在同一个事务中被更新，因此理论上讲只能存在数据的一个版本。大多数产品是使用触发器程序来实现同步复制（可能是隐含地由系统管理）。不过，同步复制有一个缺点，当更新任何一个复制品时给所有的事务增加了开销（这样也可能导致可用性问题）。
- 在异步的情况下，对某个复制品更新后，会在以后将这种更新传播给其他的复制品，而不要求在同一个事务内全部更新。因此，在异步复制中就有时间延迟或等待时间的概念，在这个时间间隔内，复制品之间可能会不一致（当然复制品这个术语也就不是很贴切了，因为我们讨论的已不再是真正的副本了）。大多数产品采用读取事务日志或读取需传播的稳定更新的队列的两种方法来实现异步更新。

异步更新的好处是将复制的开销从更新事务中分离出来，这种事务一般是“关键任务”型事务或对性能有较高要求的事务。而它的缺点则是数据的不一致性（就是用户所见到的不一致

性[⊖]），也就是说，可以在逻辑层看到这种冗余，严格地说，此时“受控冗余”这个术语也就不是很贴切的了。

我们对一致性（或不一致性）的问题进行一下简短的阐述。事实上，由于一些很难避免（并非不可能）并且很难修改的因素，复制品变得不一致了。尤其是更新顺序的不同会导致冲突的出现。比如，假设事务 *A* 在复制品 *X* 中插入一行，然后事务 *B* 将该行删除，再假设 *Y* 是 *X* 的一个复制。如果更新是以相反的顺序传播到 *Y*（比如是由于路由的延迟），*B* 发现在 *Y* 中没有要删除的行，然后事务 *A* 将该行插入！最终的结果是 *Y* 包含该行而 *X* 却没有。通常，冲突管理和强制一致性是个难题。更深的讨论超出了本书的范围。

正如在第21章中所提到的，至少在商业领域，“复制”主要指的是异步复制。

复制和复制管理（copy management）之间的根本区别是：在复制中，对某个复制品的更新会“自动”传播给其他所有的复制品。相反地，在复制管理中没有这种自动传播；数据的复制品是由一些批处理或后台进程来创建和管理的，这些进程与更新事务在时间上是分离的。复制管理一般比复制效率高，因为可以同时复制大量数据，缺点则是复制品在大多数时间都与基本数据不一致。实际上，用户一般要知道数据是什么时候被同步的。复制管理常常被简化，以保证对某些“主要复制”模式的一致性更新（参见第21章）。

**4. 导出数据**

我们要考虑的其他冗余是导出数据：尤其是计算列和汇总表，它们在决策支持环境中非常重要，用来保存预先计算的数值（根据数据库中的其他数据计算出来），这样，在查询的时候就无须重复计算。

- **计算列**的值是从同一条记录中的其他列计算得到。（或者，也可能通过计算同一个表或者不同表的几条记录得到计算列的值。因此，这种方法意味着更新一行记录时可能还需要更新其他的许多行；特别地，这种方法可能在装载和刷新操作时会产生很大的负作用。）
- **汇总表**是对其他表的聚集运算（求和、平均值、计数，等等）结果，聚集运算常常是在相同细节数据的不同分组上进行预计算（参见22.6节第）。注意：汇总表还有一系列不同的名称，包括自动汇总表（ASTs，automatic summary tables）、事实查询表（MQTs，materialized query tables）、快照和“事实视图”（materialized views）。我们曾在第10章的10.5节接触到最后两个术语，在那里我们特别讨论了术语“事实视图”。这个概念在文献里面占了很大一部分，文献的大部分用术语“视图”专门来表示“事实视图”（参见[22.3]和[22.4]）。

汇总表和计算列一般通过系统管理的触发器程序来实现，当然也可以由用户编写的程序实现。第一种方法可以保证基本数据和导出数据之间的一致性，而第二种方法则很可能导致不一致性。当然，如果计算列和汇总表真的属于受控冗余的范畴，就应该是对用户透明的，可是在现在的产品中却不是这样的。

**5. 常见设计错误**

在本小节中，我们评论决策支持环境中常见的一些设计错误：

- *重复行*：决策支持设计人员经常说他们的数据没有唯一的标识符，因此允许重复。参考文献[6.3]和[6.6]中详细阐述了为什么重复行的存在是个错误；其实导致这种错误出现的原因是因为物理模式并不是从逻辑模式中导出，甚至根本就没有设计逻辑模式。而且在这种设计中，行经常缺乏同义的语义——也就是说，它们根本不是同一个断言的实例（参见3.4节或第9章）。注意：有时会故意允许重复行的出现，特别是当设计人员具有面向对象的知识背景时（参见25.2节的最后一段）。
- *逆规范化和相关的实践*：出于消除连接运算和减少I/O的考虑，设计人员常常会预连接表，引入各种导出列，等等。这种做法可以在物理层出现，但不应该出现在逻辑层中。
- *星型模式*：“星型模式”（多维模式）是试图“短路”（short-circuit）正常设计技术的结

⊖ 参考前面章节中对于该主题的评论。

果，实际上弊大于利。当数据增长时会影响性能和灵活性，而且通过物理层重新设计来解决这种问题又会导致应用的改变（因为星型模式是物理模式）。注意：在下面的22.5节中予以详细探讨。

- 空值：设计人员经常希望通过允许列中出现空值来节省空间（如果列中包含的是变长数据类型，而且产品中在物理层通过清空列的方法来实现空值的话，也许可以节省空间）。但是可以通过良好的设计来避免空值［19.19］，并能够提供更好的存储效率和更好的I/O性能。
- 汇总表的设计：人们一般不去考虑汇总表的逻辑设计问题，因此导致了失控冗余而且难以维护数据的一致性，用户不能明白汇总表的真正语义，无法正确阐述所需的查询。为了避免这种问题，所有属于相同聚集层次的汇总表（见22.6小节）应该被设计为构成了自己的数据库。通过（a）禁止跨聚集层次的更新；和（b）总是从细节数据中聚集来更新汇总表，可以避免一些循环更新的问题。
- “多重导航路径”：决策支持设计人员和用户经常错误地说可以通过“多重导航路径”得到所需数据，相同的结果可以通过不同的关系表达式得到。有时这些表达式是等价的，例如*A* JOIN（*B* JOIN *C*）和（*A* JOIN *B*）JOIN *C*（参见第7章）；有时是由完整性约束来保证它们等价（参见第18章）；但有时它们根本不等价！比如对于最后一种情况，假设表*A*、*B*都有公共列*KAB*，表*B*、*C*都有公共列*KBC*，表*A*、*C*都有公共列*KAC*；那么“在*KAB*连接*A*和*B*然后将结果在*KBC*和*C*连接”一般上不会等价于“直接在*KAC*上连接*A*和*C*”。

很显然，这些错误会让用户迷惑，不知道应该如何表述要求，才能得到所需的查询结果。要解决这些问题，可以对用户进行适当培训，使优化器正常工作，但是因为某些问题出自于设计人员允许在逻辑层出现冗余，并允许用户直接访问物理模式，所以还要通过正确的设计来解决。

总的来说，由决策支持需求引起的很多设计难题可以依据设计原则加以解决。实际上，也是因为没有遵循设计原则才会出现这些问题（尽管这些问题常常由于SQL的一些问题而加重）。

## 22.4　数据准备

围绕决策支持的很多问题首先是关于数据的获取和准备工作。从不同数据源中提取数据，进行清洗、转换和合并整理，载入到决策支持数据库中，然后进行周期性的刷新。每一步操作都会涉及到各自的考虑事项[⊖]。我们将检视各个步骤，并在本节末尾简略讨论操作型数据存储（operational data store）。

### 1. 提取

提取是从操作型数据库和其他数据源中捕获数据的过程。很多工具可以用于辅助这项工作，包括系统提供的实用程序、定制的提取程序和商业的提取产品。提取过程要进行大量的I/O操作，可能会影响到关键任务的处理，因此一般是在物理层进行并行化操作。但是这种“物理提取”可能会丢失信息（尤其是联系信息）那些用物理方式表达的信息（比如指针或物理毗连性），从而导致后续处理上的问题。出于这方面的考虑，提取程序有时通过引入连续的记录号和更换影响外码的指针来保留这种信息。

### 2. 清洗

很少有数据源能充分保证数据质量，所以当数据进入决策支持数据库之前要进行**清洗**（cleansing）（一般成批清洗）。典型的清洗操作包括填充遗失的数据、纠正印刷错误和数据条目错误、建立标准的缩略语和格式、使用标准标识符来替代同义词，等等。无法清洗的错误数据将被驳回。注意：根据清洗过程得到的信息有时可以用来识别数据错误的原因，从而不断提高数据质量。

### 3. 转换和合并

在清洗过以后，还需要进行合适的**转换**，才能使数据满足决策支持的需要。决策支持中所需的数据形式为一组文件，各个表在物理上要区分开，这样在转换时就要求切分或合并数据源中的

---

⊖ 我们顺便提一下，这些操作常得益于关系系统的成批处理能力，尽管实际上很少如此。

记录（参见1.5节）。注意：在转换时可能会发现清洗时未发现的数据错误，这些错误数据同样也被驳回（如前所述，这些信息也可以用来提高数据源的数据质量）。

当需要对多个数据源进行**合并**时，转换就显得非常重要，称之为合并过程。这时，数据间的任何隐式联系都要被显式表示（通过引入显式的数据）。另外，还要维护和商务相关的日期时间信息，将数据源关联起来，此过程称之为“时间同步”[引用原文中的原词]。

出于性能上的考虑，转换操作常常是并行处理的，需要大量的I/O和CPU开销。

*注意*：时间同步是个难题。例如，假设我们要获取每个季度各个售货员的客户收入。客户和收入数据按财政季度保存在会计数据库中，而售货员和客户数据按日历季度保存在销售数据库中。很显然，需要将两个数据库合并起来。客户数据的合并很简单——只需匹配客户编号即可。然而时间同步则很困难，我们可以找到每个财政季度的客户收入，但却不知道此时是哪个售货员和这位客户打交道，也根本不知道每个日历季度中的客户收入。

**4. 载入**

DBMS销售商很注重载入操作的效率。“载入操作”包括：(a) 将转换和合并过的数据移入决策支持数据库中；(b) 检验一致性（integrity checking）；(c) 建立必需的索引。下面对各个步骤作简略阐述：

a. *移动数据*：现代的系统一般提供并行载入功能。有时在载入之前要预格式化数据为目标数据库中规定的内部物理格式。一种高效的替代技术是将数据载入到目标模式的镜像工作表中，在这些工作表上作完整性检验——参见下面b——然后成批地从工作表移动到目标表中。

b. *完整性检验*：大多数完整性检验工作可以在数据载入之前进行而无须涉及已经载入数据库中的数据，但是，某些约束检验必须涉及到已存入数据库中的数据；例如，唯一性约束就必须在载入时进行检验（或者在载入完毕后成批检验）。

c. *建立索引*：索引的存在会明显减缓数据载入的速度，因为在大多数产品中每插入一条新记录就会对索引进行更新。因此，在载入前删除索引，载入后再重建索引有时不失为一个好主意。当新数据的数量相对已有数据的数量较小时就不值得这么做，因为创建索引的开销并不与表的大小成正比。而且，创建大索引可能会引起不可恢复的分配错误，索引越大就越可能出现这种错误。*注意*：大多数DBMS产品支持并行创建索引，以加速载入和索引创建的过程。

**5. 刷新**

大多数决策支持数据库要求周期性地**刷新**以保证数据的及时性。尽管某些决策支持应用要求全部删除所有数据然后再载入，一般在刷新中还是只涉及部分数据的载入。刷新与载入类似，但可能会在用户访问数据库时同时进行（隐含着更深层次的问题）。

**6. 操作型数据存储**

**操作型数据存储**（ODS）是“面向主题的、集成的、可变的（可更新的）、最近的或几乎最近的数据集合”[22.20]。换句话说，它是一种特殊的数据库。术语“面向主题”表示数据属于特定的主题范围（例如，顾客、产品等）。操作型数据存储可用于（a）对提取的操作型数据进行物理重组，(b) 提供操作型报表；以及（c）支持操作型决策。它同时还可以作为（d）一个合并点，如果操作型数据来自多个数据源。因此，ODS是多用途的。*注意*：由于ODS中不积累历史数据，所以一般不会太大；换句话说，它们一般会经常或连续地使用操作型数据进行刷新。因此有时候采用从操作型数据源到ODS的异步复制（这样数据就可以在几分钟之内得到保持）。因为刷新的频率很快，所以也解决了时间同步问题（参见上面的“转换和合并”小节）。

## 22.5 数据仓库和数据集市

操作型系统一般要求高性能、可预知的工作量、较小处理单元和高可用性。而决策支持系统一般有变化的性能需求、不可预知的工作量、较大处理单元和不稳定的可用性。这些差别使得很难将操作型处理和决策支持处理合并到一个系统中，特别是当需要考虑功能规划、资源管理和系统性能调优的时候。出于以上原因，操作型系统的管理人员不愿意在其系统上进行决策支持，这样就出现了双系统（dual-system）方法。

注意：事情并不总是这样。早期的决策支持系统实际上就存在于操作型系统之中，在低优先级或“批处理窗口”中运行。如果有足够的资源，这样做有很多好处——避免了额外的数据复制、重新格式化以及转换的开销。实际上，操作型事务和决策支持事务的集成正日益被重视（见文献［21.9］），不过这个问题超出了本章讨论的范围。

在编写本章时，一般要从多个操作型系统（常常是分离的）中采集决策支持数据，然后存储在自己的数据存储中，称之为数据仓库。

**1. 数据仓库**

类似操作型数据存储（也类似数据集市——参见下一小节），数据仓库是一种特殊的数据库。这个术语来源于20世纪80年代后期［22.15，22.18］，或者更早。参考文献［22.19］中将数据仓库定义为“面向主题的、集成的、稳定的、随时间变化的数据存储，用于决策支持”（稳定的含义是数据在插入后不再改变，但可以删除）。数据仓库出现的起因有两个：首先，决策支持中需要一个单一的、纯净的、一致的数据源；其次，无须变动操作型系统。

根据定义，数据仓库的工作量就是决策支持的工作量，因此是查询密集的（有时也会进行密集的批插入操作）；同样，数据仓库本身会不断变大（常常达到数T字节，每年增长50%或更多）。因此性能调优尽管可能，也显得很困难。可缩放性也是一个问题。关于这些问题的研究包括（a）数据库设计错误（参见22.3节）；（b）仅用关系操作无法满足需要（参见第22.2节）；（c）在关系模型下的DBMS实现有很多缺点；（d）DBMS本身缺少可缩放性；（e）限制了功能和平台可缩放性的结构设计错误。（a）和（b）已经在本章中予以讨论，（c）在本书的第二部分给予详细的讨论，（d）和（e）超出了本章讨论的范围。

**2. 数据集市**

数据仓库一般用来为决策支持提供单一的数据源。但是，自从20世纪90年代初数据仓库开始流行以来，人们逐渐意识到用户常常只是在数据仓库中某个相对较小的主题领域内生成报表或分析数据。实际上，用户偏向于当这个主题领域内的数据刷新时重复同样的操作。而且，其中的一些操作——例如，预测分析、模拟、建立“如果……会怎么样”的模型——会创建新的模式和数据，并进一步更新这些新数据。

在整个数据仓库的同一个子集上重复执行这些操作显然效率较低，根据用途来裁剪数据仓库，建立一些有限的、专用的“仓库”看来是个好主意。在某些情况下，为了加快访问数据的速度，需要能够直接从本地数据源中提取和准备所需数据，而无须等待全部数据都被载入数据仓库后再进行同步。由此产生了**数据集市**的概念。

现在对于数据集市还没有统一的定义，我们给出的定义是“特定的、面向主题的、集成的、可更新的、随时间变化的数据存储，用于支持特定子集的管理决策”。数据集市与数据仓库之间的主要区别在于它是特定的和可更新的。特定的含义是它只支持特定领域的商务分析；可更新的含义是用户可以更新数据，甚至可以创建新数据（新表）。

建立数据集市有三种方法：

- 直接从数据仓库中提取数据——在整个决策支持工作量上采用“分而治之”的方法，以达到更好的性能和可缩放性。提取出来的数据被载入到数据库中，数据库的物理模式类似于数据仓库的某个合适的子集。不过根据数据集市的“特定的”这一性质，可以进行简化。
- 尽管采用数据仓库是为了提供“单一的控制点”，还是可以建立独立的数据集市（即不从数据仓库中提取数据）。当出于某些原因（金融上的、操作上的或政治上的原因，或者数据仓库还没有建立——参考下一条）无法访问数据仓库时，可以采用这种方法。
- 有时采用“首先建立数据集市”的方法，此时必须建立数据集市，而整个数据仓库则是作为对不同数据集市的合并。

后两种方法都可能导致语义不匹配的问题。独立的数据集市很容易出现这种问题，由于数据库的设计是彼此独立的，所以无法检查语义的匹配。为了避免无法将数据集市合并到数据仓库中，要求（a）首先建立单一的数据仓库逻辑模型，（b）从数据仓库模式中导出各个数据集市的模式（当然可以扩充数据仓库模式，将新数据集市中的主题内容包含进来）。

关于数据集市的设计：设计决策支持数据库时的一项重要决定是数据库的粒度。粒度是数据聚集并保存在数据库中的最低层次。现在的大多数决策支持应用迟早会访问细节数据，这在数据仓库中很容易做到，而在数据集市中则比较困难。如果不是经常访问细节数据，那么从数据仓库中提取细节数据并存储到数据集市就是低效率的工作。另外也很难决定实际所需的最低聚集层次。这时可以在数据集市中只保存聚集数据，而当需要访问细节数据时从数据仓库中提取。同时不会对数据完全聚集，因为那样会产生大量的汇总数据，在22.6节中将详细讨论这个问题。

进一步的观点：因为数据集市用户常常采用特定的分析工具，数据集市的物理设计一般要符合特定工具的需要（参见22.6节中“ROLAP与MOLAP”的探讨）。这时一般采用“多维模式”（下面将会讨论到），它无法遵循优良的关系设计规范。

**3. 多维模式**

现在假设我们希望收集商业交易的历史数据用于分析。正如22.1节中提到的，早期的决策支持系统一般将历史数据保存在一个简单文件中，通过顺序扫描来访问数据。当数据量增长时，人们就希望能从不同的角度来直接访问数据文件。例如，检索出涉及某个产品的所有商业交易，在特定时期的所有商业交易，或者与某位顾客有关的所有商业交易。

要支持这种功能，可以采用“多重目录”[⊖]数据库。在上面的例子中，数据库中有一个很大的中心数据文件（包含了商业交易数据），还有三个独立的“目录”文件——产品、时期和顾客。目录文件中包含了指向数据文件中的记录的索引，但是（a）可以由用户来设置条目（“目录维护”）；（b）其中可以包含附加信息（例如顾客的地址），从而可以从数据文件中删除这些附加信息。注意目录文件一般比数据文件要小得多。

这种组织方式比原来的设计（只有一个数据文件）要有效得多，减少了空间和I/O开销。我们特别发现中心数据文件中的产品、时期和顾客信息缩减了，现在只包含相应的标识符。

关系数据库中对这种方法的模拟，是将数据文件和目录文件变成表（相应文件的映像），目录文件中的指针变成了目录文件映像表中的主码，数据文件中的标识符则变成了数据文件映像表中的外码。这些主码和外码一般都作了索引。数据文件映像称为**事实表**，目录文件映像称为**维表**。根据实体/关系图中的形状，将整个设计称为多维模式或**星型模式**（事实表被多个维表环绕，并与它们相连）。注意：术语“维”的起源参见第22.6节。

SP

| S# | P# | TI# | QTY |
|---|---|---|---|
| S1 | P1 | TI3 | 300 |
| S1 | P1 | TI5 | 100 |
| S1 | P2 | TI1 | 200 |
| S1 | P3 | TI2 | 400 |
| S1 | P4 | TI1 | 200 |
| S1 | P5 | TI5 | 100 |
| S1 | P6 | TI4 | 100 |
| S2 | P1 | TI3 | 300 |
| S2 | P2 | TI4 | 400 |
| S3 | P2 | TI1 | 200 |
| S3 | P2 | TI3 | 200 |
| S4 | P2 | TI1 | 200 |
| S4 | P4 | TI3 | 200 |
| S4 | P5 | TI2 | 400 |
| S4 | P5 | TI1 | 400 |

TI

| TI# | FROM | TO |
|---|---|---|
| TI1 | *ta* | *tb* |
| TI2 | *tc* | *td* |
| TI3 | *te* | *tf* |
| TI4 | *tg* | *th* |
| TI5 | *ti* | *tj* |

图22-3 事实表（SP）和维表（TI）的实例

再次以供应商－零件数据库为例，现在希望列出特定时期内的发货。时期用时期标识符（TI#）来标识，引入表TI来将这些标识符与时期相对应。修改过的发货表SP和时期表TI如图22-3所示[⊖]。在星型模式中，表SP是事实表，表TI是维表（供应商表S和零件表P也是维表—参见图22-4）。注意：在第23章中会详细讨论如何处理时期数据。

查询星型模式数据库时一般会在维表中找到相应的外码组合，然后通过这些外码来访问事实表。假设在单个查询中要同时访问维表和事实表，最好的方法是使用**星型连接**。“星型连接”是一种特定的连接技术，它与常规的笛卡尔连接不同。在第18章中提到查询优化器一般会尽量避免产生笛卡尔积；现在则是首先生成维表的笛卡尔积，然后再与事实表连接（基于索引的查找），这样做很有效。许多商业优化器已经支持星型连接。

那么星型模式和关系设计之间有什么不同呢？实际上，如上所述的简单星型模式类似（甚

---

⊖ 和现在数据库系统表的意义无关。

⊖ 表TI中的FROM和TO属性列包含了一些时间戳类型的数据。为了简化，在图中我们用符号来表示而不是用实际的时间戳值。

至等同）于良好的关系设计。遗憾的是，在星型模式设计方法中经常会出现如下问题：

1）首先是特定性错误（基于直觉而非基于原则）。由于不遵循原则，当增加新数据类型或新的依赖时难以修改模式。实际上，很多星型模式都是对以前设计的修改，而以前的设计又大多是试验性的，包含错误的设计。

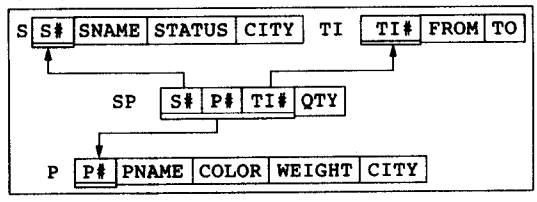



图 22-4　供应商 - 零件数据库星型模式（包括时期）

2）星型模式是物理模式，而不是逻辑模式，在星型模式设计方法中不存在逻辑设计的概念。

3）星型模式设计方法不一定会产生合理的物理设计（即它不一定会完全保留关系型逻辑设计中的所有信息），模式越复杂，这个缺点就越明显。

4）因为缺少设计原则，设计人员常常会将多个不同类型的事实存放到同一个事实表中，因此事实表中的行和列就会产生语义分歧。而且有的列还可能只保存某些类型的事实数据，这样此列就必须允许空值存在。当引入越来越多的事实类型时，事实表就变得难以维护和理解，访问效率也逐渐下降。例如，我们可能希望在发货表中不仅能跟踪零件的发货信息，而且也能跟踪零件的购买信息，于是就要增加一个“标志”列来显示哪些行是关于购买零件，哪些行是关于零件的发货。正确的设计应该是将事实表分离成两个，每个表对应不同的类型。

5）同样因为缺少设计原则，维表也会变得很不统一。当事实表中包含了不同聚集层次的数据时，很容易出现这种错误。例如，我们可能会（错误地）在发货表中增加一些行，用来显示每天、每月、每年甚至所有日期的零件总数，这样做将导致时期标识符（TI#）和表 SP 中的数量列（QTY）出现语义分歧。假设用 YEAR、MONTH、DAY 等列的组合来代替维表 TI 中的 FROM 列和 TO 列，那么这些 YEAR、MONTH、DAY 列就必须允许空值存在。而且还可能要增加一个标志列，以标明相应的时期间隔类型。

6）维表一般不是完全规范化的[⊖]。为了避免表连接的开销，设计人员常常会把截然不同的信息都保存到同一个维表中。最极端的情况是将所有可能访问到的信息都存放到同一个维表中，这种极端的、非关系型的无原则性几乎肯定会导致失控冗余的出现。

雪花模式是由一组星型模式组成，其中的维表是规范化的，名称的由来源于其实体/关系图的外观。星座模式和大风雪（或暴风雪）模式则是进一步的推广。

## 22.6　联机分析处理

术语 **OLAP**（online analytical processing）出自 1993 年 Arbor 软件公司的白皮书［22.11］，不过这个概念（以及“数据仓库”的概念）早就出现了。可以把 OLAP 定义为“关于数据的创建、管理、分析和报表生成的交互处理”使得数据好像是存储于一个“多维数组”之中。不过我们首先用传统的 SQL 表来解释概念，然后再探讨其多维表达方法。

在分析处理中总会要求进行一些不同形式的聚集操作（按照不同的分组方法），常见的基本问题是可能的分组会变得非常庞大，而用户会用到其中大多数或全部的分组。现在的 SQL 标准支持这种聚集操作，但是每个查询只能产生一个结果表——至少优于 SQL：1999——表中的所有的行形式相同，表达相同的语义[⊜]，那么为了得到 $n$ 个不同的分组信息就要求分别进行 $n$ 次不同的查询，产生 $n$ 个不同的结果表。例如，在供应商 - 零件数据库中考虑以下查询：

---

⊖ 这儿可以参考一本关于数据仓库［22.24］的书上的建议：“［反对］规范化……在多维数据库中对表进行规范化只是为了节省磁盘空间，却增加了时间开销……维表不应被规范化，否则会影响可用性。”

⊜ 除非结果表包含有空值（见第 19 章 19.3 节中的“再一次使用谓词”）。实际上，这节描述的 SQL：1999 概念高度“利用了”SQL 一些废弃的特征（?）；它们利用了空值的不同表现形式可以表达不同的意思这一特点，因此允许用许多不同的谓词来生成单一的表（正如我们将看到的）。

SP

| S# | P# | QTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |

1）得到发货的总数。

2）得到每个供应商的发货总数。

3）得到每个零件的发货总数。

4）得到每个供应商的每个零件的发货总数。

（当然，对于给定的供货商和零件，总数目恰好是那个供货商和零件的实际数目。本例中如果使用供货商-零件-工程数据库可能会更真实，不过这里我们还是采用了更简单的数据库。）

假设只有两个零件 P1 和 P2，发货表如下所示：

下面给出以上四个查询对应的 SQL 语句和查询结果。注意，SQL：1999 允许（而 SQL：1992 则不允许）：（a）GROUP BY 的操作数要求用圆括号括起来；（b）没有任何操作数的 GROUP BY 子句（后一种情况等价于完全省略了 GROUP BY 子句）。

1.
```
SELECT SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY () ;
```

| TOTQTY |
|---|
| 1600 |

2.
```
SELECT S#,
       SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY (S#) ;
```

| S# | TOTQTY |
|---|---|
| S1 | 500 |
| S2 | 700 |
| S3 | 200 |
| S4 | 200 |

3.
```
SELECT P#,
       SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY (P#) ;
```

| P# | TOTQTY |
|---|---|
| P1 | 600 |
| P2 | 1000 |

4.
```
SELECT S#, P#,
       SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY (S#,P#) ;
```

| S# | P# | TOTQTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |

这种方法的缺点非常明显：用户感觉分别阐述这些类似的查询是十分乏味的，而在相同的数据上一次又一次地执行这些查询显然需要过多的执行时间。于是人们希望找到一种方法能够（a）在一次查询中进行不同层次的聚集；（b）更有效地执行这些聚集操作。出于这种考虑，在 GROUP BY 子句中引入了 GROUPING SETS、ROLLUP 和 CUBE 选项。注意：这些选项几年前就

已经被一些商业产品所支持。它们在1999年被加入到SQL标准中。

**GROUPING SET** 选项允许用户精确指定进行哪些特定分组操作。例如，下面的SQL语句实现了第2个和第3个查询：

| S# | P# | TOTQTY |
|---|---|---|
| S1 | *null* | 500 |
| S2 | *null* | 700 |
| S3 | *null* | 200 |
| S4 | *null* | 200 |
| *null* | P1 | 600 |
| *null* | P2 | 1000 |

GROUP BY子句会要求系统有效地执行两个查询，一个按S#分组，另一个按P#分组。注意：内层括号并不是必需的（因为每个“分组集合”只涉及单个列），使用括号只是为了增加可读性。

我们并不反对将多个分离的查询捆绑到一条语句之中（但是我们希望用一种更通用的方法，比如系统的、正交的方法来解决这个问题），但是SQL还是把这些逻辑上分离的查询的结果都放到一个结果表中！在本例中的结果表如下所示：

```
SELECT S#, P#, SUM ( QTY ) AS TOTQTY
FROM   SP
GROUP  BY GROUPING SETS ( ( S# ), ( P# ) ) ;
```

以上的输出结果可被视为一个表（至少是SQL风格的表），却很难说它是一个关系。因为供应商行（P#列为null）和零件行（S#列为null）的语义不同，而TOTQTY的含义要根据它是出现在供应商行还是零件行来判定。这个“关系”的断言又该是什么呢？（实际上，这个例子中的结果表可以认为是查询2和3的结果的一种“外并”—比较奇怪的一种。至少从它的奇怪的形式上可以看出，它与第19章中的外并操作符不同，这里它并不是一个恰当的关系操作符。）

我们还注意到，结果中的null构成了另外一种“空缺信息”，它们既不表示“未知值”也不表示“无适用值”，很难弄清楚它们的含义。注意：SQL中至少提供了一种方法来将这些新的null值与其他null值区分开来，但是其技术细节过于冗长并强迫用户去逐行辨析。此处我们忽略这些细节，大家可以参考下面的例子（说明了前面GROUPING SETS例子在实际中的特征）：

```
SELECT CASE GROUPING ( S# )
          WHEN 1 THEN '??'
          ELSE S#
       AS S#,
       CASE GROUPING ( P# )
          WHEN 1 THEN '!!'
          ELSE P#
       AS P#,
       SUM ( QTY ) AS TOTQTY
FROM   SP
GROUP  BY GROUPING SETS ( ( S# ), ( P# ) ) ;
```

将结果表中的空值用另一种形式表示，S#列中的空值用两个问号代替，P#列中的空值用两个感叹号代替。因此有：

| S# | P# | TOTQTY |
|---|---|---|
| S1 | !! | 500 |
| S2 | !! | 700 |
| S3 | !! | 200 |
| S4 | !! | 200 |
| ?? | P1 | 600 |
| ?? | P2 | 1000 |

再次回到GROUP BY子句，另外两个GROUP BY选项ROLLUP和CUBE都可以视为由一些GROUPING SETS组合而成。首先来看**ROLLUP**，考虑下面的查询：

```
SELECT S#, P#, SUM ( QTY ) AS TOTQTY
FROM   SP
GROUP  BY ROLLUP ( S#, P# ) ;
```

上面的 GROUP BY 子句等同于：

```
GROUP  BY GROUPING SETS ( ( S#, P# ), ( S# ), ( ) )
```

换句话说，上面的 SQL 语句实现了第 4 个、第 2 个和第 1 个查询。查询结果如下所示：

| S# | P# | TOTQTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S1 | *null* | 500 |
| S2 | *null* | 700 |
| S3 | *null* | 200 |
| S4 | *null* | 200 |
| *null* | *null* | 1600 |

术语**上卷**（ROLLUP）（在本例中）指的是对每个供应商进行数量上卷（即“沿着供应商维上卷”——参见下面的“多维数据库”小节）。一般说来，GROUP BY ROLLUP（*A*，*B*，...，*Z*），“沿着 A 维上卷”——指的是“按下列组合进行分组”：

```
( A, B, ..., Z )
( A, B, ... )
  .....
( A, B )
( A )
( )
```

注意，一般来说，存在许多分离的“沿着 A 维上卷”（根据 ROLLUP 中用逗号分开的列来确定）。另外要注意 GROUP BY ROLLUP（*A*，*B*）和 GROUP BY ROLLUP（*B*，*A*）含义不同——即 GROUP BY ROLLUP（*A*，*B*）中的 *A* 和 *B* 不具有对称性。

现在来看一下 **CUBE**，考虑以下查询：

```
SELECT S#, P#, SUM ( QTY ) AS TOTQTY
FROM   SP
GROUP  BY CUBE ( S#, P# ) ;
```

其中的 GROUP BY 子句在逻辑上等价于下面的子句：

```
GROUP  BY GROUPING SETS ( ( S#, P# ), ( S# ), ( P# ), ( ) )
```

换句话说，上面的 SQL 语句同时执行了前面的第 4、3、2、1 个查询。查询结果如下：

| S# | P# | TOTQTY |
|---|---|---|
| S1 | P1 | 300 |
| S1 | P2 | 200 |
| S2 | P1 | 300 |
| S2 | P2 | 400 |
| S3 | P2 | 200 |
| S4 | P2 | 200 |
| S1 | *null* | 500 |
| S2 | *null* | 700 |
| S3 | *null* | 200 |
| S4 | *null* | 200 |
| *null* | P1 | 600 |
| *null* | P2 | 1000 |
| *null* | *null* | 1600 |

术语 CUBE 来源于 OLAP 中的事实：可以从多维数组或超立方体中的存储单元中找到数据的

值。在本例中，(a) 数据的值就是发货数量；(b) “立方体”只有两维，供应商维和零件维(所以，这个“立方体”其实是平面的!)；(c) 两维的尺寸不同（因此这个“立方体”不是正方形而是长方形）。总之，GROUP BY CUBE (*A*, *B*, …, *Z*) 等价于“按集合 {*A*, *B*, ..., *Z*} 的所有可能的子集分组”。

在任一个 GROUP BY 子句中可以包括多个 GROUPING SETS、ROLLUP 和 CUBE。

**1. 交叉表格**

OLAP 产品中常常不是用 SQL 风格的表，而是用**交叉表格**（简称为“crosstab”）来显示结果。再回头看第 4 个查询（得到每个供应商的每个零件的发货总数），下面的交叉表格显示了查询结果。注意到供应商 S3 和 S4 在零件 P1 上的发货量为 0，而在 SQL 中则为 null（参见第 19 章）。实际上，使用第 4 个查询对应的 SQL 语句生成的结果表中，根本就没有与 (S3, P1)、(S4, P1) 对应的行！—因此使用交叉表格显然比原来的结果表更有价值。

| | P1 | P2 |
|---|---|---|
| S1 | 300 | 200 |
| S2 | 300 | 400 |
| S3 | 0 | 200 |
| S4 | 0 | 200 |

使用交叉表格可以更紧凑更简易地表示第 4 个查询的结果。而且，它看起来也的确像一个关系表。不过，这个“表”中的列数是由实际数据决定的。在本例中，每个列对应一种零件（因此交叉表格的结构和每行的含义都是由实际数据决定）。交叉表格实际上并不是一个关系而是一份报表，格式为简单数组的报表（关系中拥有可从关系断言中推断出来的断言；但是，如果说交叉表格中也有断言的话，那么这些断言无法从关系断言中推断出来，而是依赖于实际数据的值）。

在供应商 - 零件实例中，如上所示的交叉表格一般是二维的。各个维一般被视为自变量，表格中的单元则与相关变量对应。进一步的说明请参见下面的“多维数据库”小节。

下面的交叉表格显示了上面的 CUBE 查询结果：

| | P1 | P2 | *Total* |
|---|---|---|---|
| S1 | 300 | 200 | 500 |
| S2 | 300 | 400 | 700 |
| S3 | 0 | 200 | 200 |
| S4 | 0 | 200 | 200 |
| *Total* | 600 | 1000 | 1600 |

最右边的列中数值是各行的和（即各个供应商的总发货量），底部的行中数值是各列的和（即各个零件的总发货量），右下角的单元中数值是发货量总和。

**2. 多维数据库**

到目前为止，我们一直假设 OLAP 数据存储在传统的 SQL 数据库中（尽管曾多次提及“多维数据库”）。实际上，这是 ROLAP（“关系型 OLAP”）。但是，很多人认为 MOLAP（“多维 OLAP”）是更好的解决方案，下面的小节就对 MOLAP 进行探讨。

MOLAP 涉及到**多维数据库**，从概念上讲，其中的数据存储在多维数组的单元中（注意：MOLAP 的物理存储方式与其逻辑组织是十分相似的）。相应的 DBMS 平台称为多维 DBMS。例如，可以把数据视为三维数组，分别对应于产品、顾客和时期。各单元中的值表示在相应时期内向相应顾客出售的相应产品的数量。前面的交叉表格也可以被视为这样的数组。

当已经深入了解数据和所有的关系时，就可以把涉及到的“关系变量”（不是程序语言中的变量）分为**自变量**或**应变量**。在前面的例子中，产品、顾客和时期是自变量，而发货量则是唯一的应变量。一般说来，自变量合起来决定了应变量的值（在关系数据库中，候选码就是这样的一些列，它们的值决定了其他列的值）。自变量构成了数组的各个维，并构成了数组的寻址模

式[1]，应变量的值——代表实际的数据——则存储在数组单元中。注意：自变量（维变量）的值与应变量之间的区别类似于地点和内容的区别。

不过前面对多维数据库的描述是过于简单的，因为我们并不是很了解大多数数据。因此需要首先分析数据：从而进一步了解数据。由于对数据缺少了解，就无法判断哪些是自变量，哪些是应变量。一般根据经验（即假设）来选取自变量，然后测试生成的数组是否合适（参见第22.7节）。这种方法需要多次叠代、多次试验，会出现不少错误。因此系统中应该允许交换自变量和应变量，这种操作称为绕轴转动（pivoting）。其他操作包括数组转置和维的重新排序，同时允许增加维。

从以上的描述中可以得知，数组单元常常会是空的（维数越多，这种情况出现的就越多），也就是所谓稀疏数组。例如，假设在时期 $t$ 中没有向顾客 $c$ 出售产品 $p$，那么单元 $[c, p, t]$ 就是空的（或者为零）。多维DBMS支持对稀疏数组的高效存储（通过压缩技术）[2]。另外，空单元对应“遗失信息”，系统应该能处理这些单元——可惜经常是按照与SQL类似的方式来处理。空单元可能意味着信息是未知的，或者尚未获取数值，或者数值是不合适的，或者其他情况（参见第19章）。

自变量中常常存在分类层次，它决定了应变量可以聚集的方式。例如，存在如下的时间层次：从秒到分钟、小时、日、星期、月，一直到年。还可能存在如下的层次：从零件到套件、组件、集成电路板，一直到产品。相同的数据常常可以按照不同方式聚集（即相同的自变量可以属于不同的分类层次）。系统会支持分类层次上的“上钻”（drill up）和“下钻”（drill down）操作，上钻是从低层次到高层次的聚集，下钻则相反。此外还有很多对分类层次的操作（例如，对分类层次的再排列）。

注意：“上钻”和“上卷”之间存在着细微差别：“上卷”是创建所需分组和聚集的操作，“上钻”则是对这些聚集的访问操作。下面给出“下钻”的例子：已知总的发货量，要得到每个供应商的发货量。当然，必须存在更详细的数据才可以回答这个请求。

多维产品中还会提供大量的统计和数学函数，帮助用户阐述和验证自己的假设。同时提供可视化工具和报表生成工具。不过现在还没有多维查询语言的标准，目前正在进行相关的研究[22.31]。同时多维数据库也没有类似规范化理论的科学基础。

最后我们介绍一下综合了ROLAP和MOLAP方法的产品：HOLAP（“hybrid OLAP”）。目前很难说这三种方法哪个更好[3]。一般说来，MOLAP产品计算速度快但支持的数据容量较小（随数据量的增大效率会降低），ROLAP产品支持更成熟的可缩放性、并发控制和管理机制。最近SQL标准已经包括许多统计和分析函数（见22.8节），这样ROLAP产品现在也能提供一些相应的扩展功能。

## 22.7 数据挖掘

**数据挖掘**可以描述为“探测型的数据分析”，它的目标是从数据中找到感兴趣的模式，用这些模式来决定商业策略或者发现不正常的情况（例如，信用卡突然被频繁使用可能意味着被人偷了）。数据挖掘工具在海量数据上应用了统计技术来查找这些模式。注意：数据量一般是很庞大的。数据挖掘数据库常常是特别大的，算法的可缩放性就十分重要。

---

[1] 数组单元是通过符号来寻址而不是传统的数字下标。

[2] 注意这里和关系系统中的区别。在一个关系系统里，如果“单元”是空的，那么就不会出现相应的 $(c, p, t)$。多维系统中的“稀疏数组”（或“稀疏表”）就不会出现，因此也不需要压缩技术来处理。

[3] 值得一提的是，我们常说“表是平面的”（也就是二维的），然而“实际中的数据是多维的”，因此关系不足以作为OLAP的基础。不过这种观点混淆了表和关系之间的区别！我们在第6章中看到，表仅仅是关系的形象表示，但并不是关系。当这些形象表示是二维的时候，上述观点是正确的，然而关系是 $n$ 维的（不一定是二维，$n$ 表示关系的维数），更精确的说，拥有 $n$ 个属性的关系中的每个元组代表 $n$ 维空间中的一个点，整个关系代表这些点的集合。

图 22-5 所示的销售表中给出了某个零售商的售货交易信息[⊖]。商家希望能在这些数据上进行购物筐分析（“购物筐”（market basket）指的是单次交易中购买的所有产品），从而发现诸如购买鞋子的顾客常常也会同时购买袜子之类的信息。鞋子和袜子之间的相关性就是**关联规则**的一个例子，可如下表示：

```
FORALL tx ( Shoes ∈ tx ⇒ Socks ∈ tx )
```

（其中“Shoes ∈ *tx*”是规则的前因，“Socks ∈ *tx*”是规则的结果，*tx* 则代表任意一次交易）。

下面引入一些术语。全部交易的集合称为 **population**，每个关联规则都有支持度和置信度。**支持度**指的是满足规则的交易占全部交易的百分比，**置信度**指的是同时满足前因和结果的交易占满足前因的交易的百分比（注意前因和结果中各自可以包含任意多项产品）。另外在本例中考虑如下规则：

```
FORALL tx ( Socks ∈ tx ⇒ Tie ∈ tx )
```

在图 22-5 中的示例数据中，population 为 4，支持度为 50%，置信度为 66.67%。

在给定数据上进行适当聚集可以得到更普遍的关联规则。例如，对顾客分组可以验证如下规则“如果某位顾客购买了鞋子，他（她）可能会同时或其他时候购买袜子”。

还可以定义其他类型的规则。例如，**序列关联**规则可用来标识随时间推移的购买模式（“如果某位顾客今天买了鞋子，他（她）就可能在五天内购买袜子”）。分类规则可用来辅助判断是否批准一项信用应用（“如果某位顾客的年收入超过 75 000 美元，他（她）就可能比较值得信赖”），等等。类似于关联规则，序列关联和分类规则同样具有相应的支持度和置信度。

SALES

| TX# | CUST# | TIMESTAMP | PRODUCT |
|---|---|---|---|
| TX1 | C1 | *d*1 | Shoes |
| TX1 | C1 | *d*1 | Socks |
| TX1 | C1 | *d*1 | Tie |
| TX2 | C2 | *d*2 | Shoes |
| TX2 | C2 | *d*2 | Socks |
| TX2 | C2 | *d*2 | Tie |
| TX2 | C2 | *d*2 | Belt |
| TX2 | C2 | *d*2 | Shirt |
| TX3 | C3 | *d*2 | Shoes |
| TX3 | C3 | *d*2 | Tie |
| TX4 | C2 | *d*3 | Shoes |
| TX4 | C2 | *d*3 | Socks |
| TX4 | C2 | *d*3 | Belt |

图 22-5　销售表

数据挖掘是个很大的课题 [22.2]，在此无法一一详述。最后简单介绍如何在供应商 - 零件数据库中进行数据挖掘。首先（在不涉及其他信息的情况下）可以使用神经归纳（neural induction）按经营项目（如刹车或发动机零件）对供应商分类，使用价值预测来预测出每个供应商最可能供应什么产品。然后使用人口统计的聚类把供应商和所在地区、运输地区联系起来。这时可以采用关联发现技术来找出每次运输中哪些零件会被同时装运，采用序列关联发现技术来找出当装运过发动机零件后一般会装运刹车零件，采用相似时间序列发现技术来找出哪些零件的装运量随季节而变化（比如哪些变化发生在秋季，而哪些发生在春季）。

## 22.8　SQL 的支持

OLAP 的一些功能，基本上是 22.7 节中描述的 GROUPING SET、ROLLUP 和 CUBE 操作（作为 GROUP BY 的扩充）最先收录到 SQL：1999 标准中，第二年又加入了一些新功能对其完善 [22.21]。完善的详细内容超出了本书的范围，我们只简单的列出下面一些特征总结：

- 新的数学函数（比如自然对数、指数、幂、平方根、下限、上限函数）
- 新的聚集操作符（比如方差、标准差）
- 分类函数（比如通过权重对列表中的元素分类）
- 累积函数以及其他种类的“动态平均”函数（在 SQL 的表达式 SELECT - FROM -

⊖ 注意：（a）码是 {TX#, PRODUCT}；（b）表满足函数依赖 TX# -> CUST#，和 TX# -> TIMESTAMP（非 BCNF）；（c）如果表包含列 PRODUCT（而且 TX#是码），则表属于 BCNF，这样可能更适合与此例类似的情况（但可能不适合其他情况）。

WHERE - GROUP BY - HAVING 表达式中加入一个新的 WINDOW 子句）

- 应用到成对出现的列上的分布、逆分布、相关以及其他统计函数

## 22.9 小结

本章介绍了数据库技术在**决策支持**中的应用。其基本思想是从操作型数据中收集信息，然后将这些信息重新格式化，以帮助管理部门了解和修改企业行为。

首先指出了为什么决策支持系统要和操作型系统分离，主要因为决策支持数据库大多是**只读**的。决策支持数据库会变得很**大**，索引的开销**繁重**，同时涉及大量的**受控冗余**（特别是在复制和聚集预计算中），码中一般有**时间列**，查询也很**复杂**。出于这些考虑，人们会在设计时特别强调性能问题；性能问题值得关注，但这并不是采用不好的设计的理由。实际上，问题在于决策支持系统一般没有明确地区分**逻辑设计**和**物理设计**。

其次探讨了在为决策支持进行操作型数据准备时所涉及的步骤，包括**提取、清洗、转换、合并、载入**和**刷新**，还提及了**操作型数据存储**，可把操作型数据存储作为数据准备过程中的过渡阶段，也可以用来在现有数据上提供决策支持。

再次就是**数据仓库**和**数据集市**。数据集市可以视为一种特殊的数据仓库。我们解释了**星型模式**，其中包括一个大的中心**事实表**和多个小的**维表**。在简化时，星型模式和规范化的关系模式没有什么区别，而实际上星型模式与传统的设计理论存在许多不同，主要是出于性能上的考虑（还是同样的问题，星型模式在本质上更像物理模式而不是逻辑模式）。另外还提到了连接的实现策略（如星型连接）以及雪花模式。

接下来是 **OLAP**。我们讨论了 SQL **GROUPING SETS**、**ROLLUP** 和 **CUBE**（这些都是 GROUP BY 子句的选项，在单个 SQL 查询中实现多个不同的聚集操作）。SQL 中把这些分离的聚集操作结果都放入单个表中（其中包含了很多空值），OLAP 产品可以将这些表转化为**交叉表格**以便显示。在**多维数据库**中，从概念上讲，数据是存储在多维数组或超立方体中，数组的维代表**自变量**，单元中的值代表**应变量**。自变量一般涉及不同的分类层次，分类层次决定了应变量可以分组、聚集的方式。

然后谈到**数据挖掘**。由于一般来说决策支持不被充分理解，可以利用计算机的计算能力来帮助我们发现数据中的模式。我们简单介绍了几种规则——**关联规则**、**分类规则**和**序列关联规则**，以及相关的**支持度**和**置信度**等概念。

最后我们简单介绍了 SQL：1999 OLAP 修改稿中新增的一些功能。

## 习题

22.1 决策支持数据库和操作型数据库有什么主要区别？为什么一般要把决策支持应用和操作型应用分离开来？

22.2 决策支持中的操作型数据准备主要包括哪些步骤？

22.3 请区分受控冗余和失控冗余，并举例说明。为什么在决策支持环境中要强调受控冗余？如果是冗余失控，会出现什么后果？

22.4 请区分数据仓库和数据集市。

22.5 什么是星型模式？

22.6 星型模式一般都不是完全规范化的，如何处理这个问题？阐述其设计时采用的方法。

22.7 ROLAP 和 MOLAP 之间有什么区别？

22.8 当数据包括四维，每一维中有三层分类层次时，有多少种方法进行汇总？

22.9 在供应商 - 零件 - 工程数据库中，用 SQL 语言描述以下查询：

a. 找出每个供应商、零件和工程两两成对（即对于每对 S#-P#，P#-J#和 J#-S#）所对应的发货次数和平均发货量。

b. 找出每个供应商、零件和工程任意组合和总共的最大、最小发货量。

c. 找出“沿供应商维”和“沿零件维”上卷的总发货量。警告：这里有个陷阱。

d. 找出每个供应商、零件和工程任意组合和总共的平均发货量。

在每个查询中给出 SQL 产生的查询结果（采用图 4-5 中的示例数据），并用交叉表格显示。

22.10　在第22.6节的开始，我们给出了只包含6行的表SP。假设在表中另外增加如下的行（表示存在供应商S5，但他没有供应任何零件）：

| S5 | null | null |
|---|---|---|

讨论它对第22.6节中所有查询的影响。

22.11　“多维模式”和“多维数据库”中的“维”是否是相同的概念？为什么？

22.12　考虑购物筐分析问题。简单书写关联规则（具有指定支持度和置信度）的发现算法。提示：某种（些）商品是“人们不感兴趣的”，指的是它（们）只在很少交易中出现。

## 参考文献

注意：文献［22.3-22.5］、［22.10］、［22.12］、［22.16］、［22.25］、［22.28］、［22.30］和［22.35］在题目中提到的“views”指的是快照，而不是视图。


［22.1］　Brad Adelberg, Hector Garcia-Molina, and Jennifer Widom:“ The STRIP Rule System for Efficiently Maintaining Derived Data,“ Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz (May 1997).

STRIP是STanford Real-time Information Processor的缩写。一旦底层数据发生变化，则STRIP用“规则”（通常称这儿为触发器）来更新快照（称这儿为导出数据）。通常这种系统存在的问题是如果数据变化频繁的话，则执行这些规则的计算代价将变得非常大。本文讨论了可以减少这种代价的STRIP技术。

［22.2］　Pieter Adriaans and Dolf Zantinge: *Data Mining*. Reading, Mass.: Addison-Wesley (1996).

尽管本书被描述为一本应用型的概述，但实际上该书详细介绍了与数据挖掘相关的主题。

［22.3］　Foto N. Afrati, Chen Li, and Jeffrey D. Ullman:“ Generating Efficient Plans for Queries Using Views,” Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

［22.4］　D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek:“Efficient View Maintenance at Data Warehouses,” Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

文献［22.12］中的注解中提到，快照可以以递增的方式维护，由于性能方面的原因是需要这种递增式的维护的。然而，如果快照是由一些同时更新的不同数据库导出的，那么这种递增式的维护将会导致一些问题。这篇文章提出了针对这个问题的一个解决方案。

［22.5］　Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya:“ Automated Selection of Materialized Views and Indexes for SQL Databases,” Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt (September 2000).

［22.6］　S. Alter: *Decision Support Systems: Current Practice and Continuing Challenges*. Reading, Mass.: Addison-Wesley (1980).

［22.7］　J. L. Bennett (ed.): *Building Decision Support Systems*. Reading, Mass.: Addison-Wesley (1981).

［22.8］　R. H. Bonczek, C. W. Holsapple, and A. Whinston: *Foundations of Decision Support Systems*. Orlando, Fla.: Academic Press (1981).

最早导入决策支持系统规范化方法的著作之一。重点讨论建模（经验和数学建模）和管理科学。

［22.9］　Charles J. Bontempo and Cynthia Maro Saracco: *Database Management: Principles and Products*. Upper Saddle River, N. J.: Prentice Hall (1996).

［22.10］　Rada Chirkova, Alon Y. Halevy, and Dan Suciu:“ A Formal Perspective on the View Selection Problem.” Proc. 27th Int. Conf. on Very Large Data Bases, Rome, Italy (September 2001).

［22.11］　E. F. Codd, S. B. Codd, and C. T. Salley:“Providing OLAP (Online Analytical Processing) to User-Analysts: An IT Mandate,” available from Arbor Software Corp. (1993).

本章中提到这篇论文首次给出了“OLAP”术语（在22.6小节中，尽管不是概念）。注意：论文一开始就说明“它不是要引入新的数据库技术，而是探讨更健壮的……分析工具”。但实际上却是在讨论一种新的数据库技术！——新的概念数据表达、新操作符、多用户支持（包括安全性和并发特征）、新存储结构以及新的优化特征；换句话说，讨论一种新的数据模型和一种新的DBMS。

［22.12］　Latha S. Colby *et al.*:“ Supporting Multiple View Maintenance Policies,” Proc. 1997 ACM SIGMOD

Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

有三种主要的方法用于快照更新：立即更新（一旦底层关系被更新，立即会引发快照的更新）、延迟更新（仅当快照被查询的时候才更新）以及周期更新（快照在指定的时间被更新 - 比如说每天）。快照是以牺牲更新性能的代价来改善查询性能，这三种维护策略是这两种性能之间的平衡。这篇文章研究了在同一个系统上同时对不同快照的不同策略支持时相关的问题。

[22.13] C. J. Date: "We Don't Need Composite Columns," in C. J. Data, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994 - 1997*. Reading, Mass.: Addison-Wesley (1998).

22.3 节中提到了复合列的概念；这篇论文详细讨论了这个概念。论文标题指出了引入复合列是没有意义的，应该支持合适的用户自定义类型。

[22.14] Barry Devlin: *Data Warehouse from Architecture to Implementation*. Reading, Mass.: Addison-Wesley (1997).

[22.15] B. A. Devlin and P. T. Murphy: "An Architecture for a Business and Information System," *IBM Sys. J. 27*, No. 1 (1988).

第一篇定义和使用"信息仓库"的论文。

[22.16] Jonathan Goldstein and Per-Åke Larson: "Optimizing Queries Using Materialized Views: A Practical, Scalable Solution," Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

[22.17] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh: "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," Proc. 12th IEEE Int. Conf. on Data Engineering, New Orleans, La. (February 1996).

最早建议在 SQL GROUP BY 子句中引入 CUBE 操作的文章。

[22.18] W. H. Inmon: *Data Architecture: The Information Paradigm*. Wellesley, Mass.: QED Information Sciences (1988).

讨论了数据仓库概念的起源，并给出了数据仓库的特点。首次给出"数据仓库"术语。

[22.19] W. H. Inmon: *Building the Data Warehouse*. New York, N. Y.: John Wiley & Sons (1992).

描述数据仓库的第一本书。定义了开发数据仓库时的相关术语及关键问题。主要涉及概念和物理设计问题。

[22.20] W. H. Inmon and R. D. Hackathorn: *Using the Data Warehouse*. New York, N. Y.: John Wiley & Sons (1994).

面向数据仓库的用户和管理人员。探讨物理设计问题和操作型数据存储。

[22.21] International Organization for Standardization (ISO): *SQL/OLAP*, Document ISO/IEC 9075-1:1999/Amd. 1:2000(E).

[22.22] P. G. W. Keen and M. S. Scott Morton: *Decision Support Systems: An Organizational Perspective*. Reading, Mass.: Addison-Wesley (1978).

早期的决策支持经典著作。决策支持系统的分析、设计、实现、评价和开发。

[22.23] Werner Kiessling: "Foundations of Preferences in Database Systems" and "Preference SQL—Design, Implementation, Experiences," Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong (August 2002).

优先允许用户进行模糊查询（比如，"给我找一家好的四川餐馆，不要太贵，最好在市区"）

[22.24] Ralph Kimball: *The Data Warehouse Toolkit*. New York, N. Y.: John Wiley & Sons (1996).

一本入门指导书。在子标题"建造多维数据仓库的实用技术"中，重点讨论实际问题而非理论。认为系统在逻辑和物理上没有什么本质不同。

[22.25] Yannis Kotidis and Nick Roussopoulos: "A Case for Dynamic View Management," *ACM TODS 26*, No. 4 (December 2001).

[22.26] M. S. Scott Morton: "Management Decision Systems: Computer-Based Support for Decision Making," Harvard University, Division of Research, Graduate School of Business Administration (1971).

本文是一篇介绍管理决策系统的经典文章，将决策支持应用到计算机系统中。建立了特定的"管理决策系统"，用于卫生产品计划。后来又应用到市场管理和生产管理的科学测试中。

[22.27] K. Parsaye and M. Chignell: *Intelligent Database Tools and Applications*. New York, N. Y.: John Wiley &

Sons(1993).

第一篇介绍数据挖掘原理和技术的文章（文中使用“智能数据库”术语）。

[22.28] Rachel Pottinger and Alon Levy:“ A Scalable Algorithm for Answering Queries Using Views,” Proc. 26th Int. Conf. on Very Large Data Bases,Cairo,Egypt (September 2000).

[22.29] Dallan Quass and Jennifer Widom:“ On-Line Warehouse View Maintenance,” Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data,Tucson,Ariz. (May 1997).

本文提出了快照维护的一种算法，允许维护事务和查询（而不是快照）一起进行。

[22.30] Kenneth Salem,Kevin Beyer,Bruce Lindsay,and Roberta Cochrane:“How to Roll a Join:Asynchronous Incremental View Maintenance,”Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data,Dallas, Tex. (May 2000).

[22.31] Erik Thomsen:*OLAP Solutions:Building Multi-Dimensional Information Systems*(2d ed.). New York, N. Y. :John Wiley & Sons (2002).

本文是最早讨论 OLAP 的综合性文章之一，侧重于多维分析概念和方法。

[22.32] R. Uthurusamy:“From Data Mining to Knowledge Discovery:Current Challenges and Future Directions,” in U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (eds.):*Advances in Knowledge Discovery and Data Mining*. Cambridge,Mass. :AAAI Press/MIT Press (1996).

[22.33] Patrick Valduriez:“ Join Indices,” *ACM TODS 12*,No. 2 (June 1987).

[22.34] Markos Zaharioudakis *et al.* :“Answering Complex SQL Queries Using Automatic Summary Tables.” Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data,Dallas,Tex. (May 2000).

[22.35] Yue Zhuge,Hector Garcia-Molina,Joachim Hammer,and Jennifer Widom:“ View Maintenance in a Warehousing Environment,” Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data,San Jose, Calif. (May 1995).

当数据仓库场地被告知要对底层数据更新时，它在实施必要的快照更新前可能需要向基数据场地发出一个查询请求，查询请求到基数据更新之间的时间延迟可能导致异常。这篇论文提出了处理这种异常的一种算法。

# 第23章　时态数据库

## 23.1　引言

简单地说，**时态数据库**可以定义为，一个包含了历史数据[⊖]或同时也包含当前数据的数据库（数据仓库就是一个典型例子——参考第22章）。传统的或者非时态的数据库只包含了当前数据；一旦它们代表的命题（proposition）不正确，就要对其进行修改，以维护数据的当前性。相反，除了那些第一次将数据插入的插入操作，时态数据库的更新操作非常少，甚至根本没有更新操作。以供应商-零件数据库为例，给定了通常的采样值。数据库显示供应商S1的状态（状态是“此刻”）是20。但是在同一个数据库的时态数据库版本中，数据库显示的可能不仅仅是此刻的供应商S1的状态20，还会显示从去年7月1日到现在，它的状态一直是20，而从去年的4月5日到6月30日S1的状态是15，等等。

时态数据库研究从20世纪80年代早期就开始了，其中包括了相当数量的关于它的时间本质的研究。这里有一些提出的问题：

- 时间是否有起点或者终点？
- 时间是连续的还是被分成了离散的量子？
- 如何最好地描述“现在”（有时被描述成“移动点NOW”）这个重要概念？

然而，这些问题并不是真正意义上的数据库问题，因此在本章中我们不作深入钻研，而只是简单地进行合理假设，使我们的精力可以集中在与我们的整体目标直接相关的事物上。

我们用到了一个事实：通常情况下，数据可以被视为它们代表的命题。由此可以得出结论，时态数据可以被视为代表带时间戳的命题——也就是说，命题至少包含了某一时间戳类型的一个说法。例如，看下面这个元组：

| S# | SINCE |
|---|---|
| S1 | July 1, 2003 |

从这个元组中可以看到，有一个供应商号属性S#，以及一个时间戳属性“SINCE”，相应的带时间戳的命题是：

*供应商S1从2003年7月1日**开始**就签订了合约。*

当然，我们假设了命题满足下列谓词[⊖]形式：供应商S#从日期SINCE **开始**就签订了合约。后面我们会解释为什么在这个谓词中（以及在这个实例中）把“从……开始”设置成黑体字。首先，先来看看另一个例子：

| S# | FROM | TO |
|---|---|---|
| S1 | May 1, 2002 | April 30, 2003 |

这个元组有一个供应商号属性S#，以及两个时间戳属性“FROM”和“TO”，相应的带时间戳的命题是：

*供应商S1从2002年5月1日**到**2003年4月30日这段时间签订了合约。*

现在我们假设命题满足下列谓词形式：供应商S#从日期FROM **到**日期TO这段时间签订了合约。同上面一样，我们在后面解释其中使用的黑体字。

就像这两个例子显示的那样，“从……开始”和“从……到……”这两个概念在接下来的表述中是非常重要的（事实上是无孔不入的）。然而，为了使它们真正有用，我们需要对它们进行

---

⊖ 在时态数据库中，“历史数据”可以包含关于过去和未来的数据。

⊖ 本章中我们使用无条件的“谓词”来表示在第9章中我们称为“外部”或者用户理解的谓词。

非常精确的定义（比我们在通常讨论下的定义要精确得多）。细节如下：

1）“从……开始”表示**从指定的时间点到现在**，并且不包括此前的一刻。因此，当我们说，供应商 S1 从 2003 年 7 月 1 日开始就签订了合约，这表示（a）供应商 S1 从 2003 年 7 月 1 日开始到现在，包括今天——无论日期是什么——都签订了合约，而（b）供应商 S1 在 2003 年 6 月 30 日时没有签合约。

2）“从……到……”表示**贯穿指定的时间区间**，并且不包括此前或此后的时刻。因此，当我们说，供应商 S1 从 2002 年 5 月 1 日到 2003 年 4 月 30 日这段时间签订了合约，这表示（a）供应商 S1 在贯穿 2002 年 5 月 1 日到 2003 年 4 月 30 日这段时间里签订了合约[⊖]，而（b）供应商 S1 在 2002 年 4 月 30 日或 2003 年 5 月 1 日时没有签合约。

我们在前面的命题和谓词中使用了黑体字“**从……开始**”和“**从……到……**”，这是为了强调我们在前面使用术语的精确性。而在后面的内容中将不再使用黑体字。

### 1. 一些基本假设

我们已经提到过时间区间和时间点，现在该解释它们了——或者至少解释一下支持这些概念的基本假设。首先，我们假设时间本身可以被想象成一条**时间线**（timeline），由离散的、不可分割的时间量子的有限序列组成，其中**时间量子**是系统能够表示的最小时间单位。换句话说，即使真实世界中的时间是连续的、无限的，但在我们的模型中，将它表示为离散的、有限的。注意：这与我们通常用有理数来表示实数的计算模型有明显的相似之处。

接下来，我们仔细辨别（1）时间量子，刚刚解释为系统能够表示的最小时间单位，（2）与特定目的相关的时间单位，可能是月、天或毫秒等。比如，在前面讨论的关于供应商的例子中，我们感兴趣的时间要精确到天。我们把与特定目的相关的时间单位称为**时间点**（简称点），来强调为此目的这些时间也被认为是不可分割的。现在，可以非正式地说，一个时间点是“时间线的一段”，表示一个边界量与下一个边界量之间（比如，一天的午夜到第二天的午夜）的时间量子的集合。因此还可以非正式地说，时间点有一个持续时间（在我们的例子中是一天）。然而，正式地说，时间点确实是点——它们是不可分割的，持续时间的概念并不适用。

注意：尽管在前面的段落中没有直接写出，但我们确实有时利用了“粒度”这一术语，它非正式地表示了应用的时间点的大小（或持续时间），或者等同于临近点之间的区间的大小（或持续时间）。因而我们可以说，在我们的例子中，粒度是一天，也就是说，消除了（在此处的上下文中）我们通常认为的一天是由小时组成的，小时是由分钟组成的等一系列概念。这些概念只能通过追索更细层次的粒度来表示。

在某一特定目的下，时间线可以被视为时间点（与时间量子相反）的一个有限序列；当然，该序列是按照时间顺序排列的。既然它是有限的，那么它需要遵循：

- 序列有一个起点和一个终点。换言之，序列中存在唯一一个起点对应于**时间线的开端**，同时还存在唯一一个终点对应于**时间线的末端**。
- 在序列中，除了对应于时间开端的点之外的其他每一个点都有唯一的**前驱点**，同时，除了对应于时间末端的点之外的其他每一个点都有唯一的**后继点**。
- 我们定义一个时间**区间**为时间线上的一个非空区间[⊖]。更准确地说，时间区间有一个**起点** $b$ 和一个**终点** $e$，它可被视为是**由所有点** $p$，$b \leqslant p \leqslant e$（其中“$<$”表示“早于”），**组成的时间线的一个子序列**。

### 2. 例子

当提到“供应商和发货”时，如果不另行声明，我们的例子都是基于一个简单的供应商 - 零件数据库的。具体地说：

---

⊖ 本章中我们采用称为闭区间的解释，根据这种解释，从 $b$ 到 $e$ 的区间被视为包括了起点 $b$ 和终点 $e$。我们也注意到了在文献中可以找到其他解释方式。

⊖ 如果你熟悉 SQL，那么提醒你，这里定义的时间区间与在 SQL 中理解的时间区间完全不同，它不是通常意义下的时间区间，而是指持续时间（例如，“3 天”）。

1）将零件关系变量 P 全部删掉。

2）为了简化供应商关系变量 S，删除其除了 S#以外的所有属性。修改后对应的谓词非常简单，是：

供应商 S#目前是签订了合约的。

3）删除发货关系变量 SP 中的 QTY 属性，修改后的关系语义是：

供应商 S#目前可以供应零件 P#。

换句话说，关系变量 SP 的这个简单版本，不再包含供应商的实际发货信息，而只表示可被称为潜在发货的信息——某供应商提供某种零件的能力。注意：尽管这已经改变了关系变量的意义，但在下面我们还是认为使用“发货”这个术语是很方便的。

图 23-1 显示了这个简化数据库的一组实例值。注意：这个数据库仍然完全是一个传统数据库——它还不包含任何时间属性。

现在我们来看这个数据库上的一些约束和查询。在下一节中，我们会看到，当对数据库进行扩展使其包含各种时态特征时，这些约束和查询会发生什么变化。

**约束**（原始数据库）：我们需要考虑的约束如下。

- {S#} 和 {S#, P#} 分别是关系变量 S 和 SP 的主码[⊖]。
- {S#} 是关系变量 SP 的外码，参照了关系变量 S 的主码。

**查询**（原始数据库）：我们只考虑两种查询。

- 查询 *A*：取得当前至少可以供应一种零件的供应商的数目。

```
SP { S# }
```

- 查询 *B*：取得当前不能供应任何零件的供应商的数目。

```
S { S# } MINUS SP { S# }
```

S

| S# |
|---|
| S1 |
| S2 |
| S3 |
| S4 |
| S5 |

SP

| S# | P# |
|---|---|
| S1 | P1 |
| S1 | P2 |
| S1 | P3 |
| S1 | P4 |
| S1 | P5 |
| S1 | P6 |
| S2 | P1 |
| S2 | P2 |
| S3 | P2 |
| S4 | P2 |
| S4 | P4 |
| S4 | P5 |

图 23-1　供应商 - 发货数据库（原始版本）实例

可以观察到，查询 *A* 包含了一个简单的投影操作，查询 *B* 则是两个此类投影的差。在 23.5 节中，当我们考虑这两个查询的时态版本时，会发现对上述两个操作存在相应的“时态”形式（至少是具有普遍意义的），并且你会了解到其他关系操作也可以定义类似的通用的形式。

在本节的最后，来看一下本章其余部分的安排。首先，23.2 节说明为什么时态数据需要特殊对待。接下来的 23.3 节会解释怎样将区间处理为一个数值，而不是起点和终点两个数值；这一节还特别介绍了多种处理这种区间的操作。23.4 节讨论了两种非常重要的关系操作，归并和反归并。23.5 节描述了关系代数中类似操作的通用形式。最后，23.6 节和 23.7 节分别讨论了数据库设计问题和完整性约束。

## 23.2　问题是什么

第一步就是要将供应商 - 发货数据库转化成包含了半时态化的关系变量 S 和 SP 的形式（通过向两个关系变量中加入时间戳属性 SINCE），并且根据这一转化对它们进行重命名。如图 23-2 所示。

为了简单起见，在图 23-2 中没有表示出来真正的时间戳，而是用形如 *d01*，*d02* 等等这样的符号代替，其中“d”是“day”的缩写，贯穿整章我们都会使用这个约定俗成的表示方法。（多数例子用到的时间点都特指天数，因此在那些例子中应用的粒度是一天。）我们假设日期 1 马上接着日期 2，日期 2 马上接着日期 3，如此类推；同样，也从表述中删掉开头没有意义的零，表示成“日期 1”（像你们看到的那样）。

---

⊖ 在本章中，为了明确起见，我们假设关系都有特定的主码。但在 **Tutorial D** 的关系定义中，我们并不区分主码和预备码（alternate key），而是将它们统一视为候选码。

关系变量 S_SINCE 和 SP_SINCE 的谓词语义是：

- S_SINCE：供应商 S#从 SINCE 那天开始就签订了合约。
- SP_SINCE：供应商 S#从 SINCE 那天开始就可以供应零件 P#。

**约束**（半时态化数据库）：图 23-2 的“半时态化的”数据库中的主码和外码与图 23-1 的原始数据库的相同。因此，关系变量定义如下：

```
VAR S_SINCE BASE RELATION { S# S#, SINCE DATE }
    KEY { S# } ;

VAR SP_SINCE BASE RELATION { S# S#, P# P#, SINCE DATE }
    KEY { S#, P# }
    FOREIGN KEY { S# } REFERENCES S_SINCE ;
```

这里的 DATE 类型表示的是罗马日期——日期对于每一天是很准确的，同时也是要受到罗马日历规则（其中隐含了诸如“2005 年 4 月 31 日”和“2100 年 2 月 29 日”不是有效日期）约束的。

然而，在外码上，我们需要另一个从 SP_SINCE 到 S_SINCE 的约束，来表示任何供应商在签约之前不能提供任何零件：

```
CONSTRAINT XST1     /* "半时态外部约束 no.1" */
   IS_EMPTY ( ( ( S_SINCE  RENAME SINCE AS SS  ) JOIN
                ( SP_SINCE RENAME SINCE AS SPS ) )
              WHERE SPS < SS ) ;
```

（“如果 SP_SINCE 中的元组 *sp* 引用了 S_SINCE 中的元组 *s*，那么 *sp* 中的 SINCE 值不能小于 *s* 中的 SINCE 值”）。通过这个例子，我们开始看到问题：给定一个像图 23-2 那样的“半时态”数据库，我们可能需要规定很多普通而又相当麻烦的约束，就像约束 XST1，于是我们就希望能有一些方便的速记符号。

S_SINCE

| S# | SINCE |
|---|---|
| S1 | *d04* |
| S2 | *d07* |
| S3 | *d03* |
| S4 | *d04* |
| S5 | *d02* |

SP_SINCE

| S# | P# | SINCE |
|---|---|---|
| S1 | P1 | *d04* |
| S1 | P2 | *d05* |
| S1 | P3 | *d09* |
| S1 | P4 | *d05* |
| S1 | P5 | *d04* |
| S1 | P6 | *d06* |
| S2 | P1 | *d08* |
| S2 | P2 | *d09* |
| S3 | P2 | *d08* |
| S4 | P2 | *d06* |
| S4 | P4 | *d04* |
| S4 | P5 | *d05* |

图 23-2　供应商 - 发货数据库（半时态版本）实例

**查询**（半时态化数据库）：现在考虑查询 *A* 和查询 *B* 的半时态化版本。

- 查询 *A*：取得当前至少可以供应一种零件的供应商的数目，并分别显示从何时起可以供应。

如果供应商 S*x* 现在可以供应一些零件，那么 S*x* 可以供应零件的最早日期就是 SP_SINCE 中 S*x* 对应的 SINCE 的最小值（例如，如果 S*x* 是 S1，那么 S1 对应的最早 SINCE 日期为 *d04*）。因此：

```
SUMMARIZE SP BY { S# } ADD MIN ( SINCE ) AS SINCE
```

查询结果如下：

| S# | SINCE |
|---|---|
| S1 | *d04* |
| S2 | *d08* |
| S3 | *d08* |
| S4 | *d04* |

- 查询 *B*：取得当前不能供应任何零件的供应商的数目，并分别显示从何时起无法供应。

在我们的示例数据中可以看到，现在只有一位供应商 S5 无法供应任何零件。但是我们无从得知它从何时起无法供应零件，因为数据库中还没有足够的信息（重复一次，这个数据库还只是半时态化的）。例如，假设现在的日期为 *d10*，那么 S5 可能在日期 *d02* 到 *d09* 都可以供应至少一种零件，或者是另一个极端，S5 可能从来就无法供应任何零件。

要回答查询 **B**，必须完全“时态化”整个数据库，或者至少要完全“时态化” SP 这部分。具体地说，就要在数据库中保存历史记录，来反映每个供应商在何时可以供应何种零件。如图 23-3 所示。

比较图 23-3 和图 23-2，我们看到 SINCE 属性已经变成了 FROM 属性，并且每个关系变量都增加了 TO 这个时间戳属性（相应地，我们用_FROM_TO 替代了原来关系变量的名字_SINCE）。FROM 和 TO 两个属性合在一起，描述了时间区间的概念，在这段时间内某些命题是真的。注意：我们已经明确假设了今天是日期 10，因此每个关于事务当前状态的元组，它的属性 TO 的值都显示为 *d10*。然而，这一假设应该立刻使你想到，是什么机制可以保证在过了日期 10 的午夜时，所有那些 *d10* 会替换为 *d11*。这个问题我们将暂时放一放，在第 23.6 节再回到这个问题上来。

可以观察到，由于我们保存了历史记录，因此基数的数目比以前要多；事实上，图 23-3 的完全时态化数据库，包括了图 23-2 的半时态化版本的所有信息，除了一点，从这个例子来说，我们将两条供应商 S4 发货记录的 TO 值显示为当前日期之前的某个日期（也就是说，我们将那两条发货记录从“当前”日期转化成“历史”信息）。图 23-3 也包括了从 *d02* 到 *d04* 这一早些时间区间内的历史信息，在这段时间内，供应商 S2 是签订了合约的，并且可以供应某些零件。谓词语义如下：

- S_FROM_TO：供应商 S#从 FROM 到 TO 这段时间里是签订了合约的。
- SP_FROM_TO：供应商 S#从 FROM 到 TO 这段时间里可以供应零件 P#。

S_FROM_TO

| S# | FROM | TO |
|---|---|---|
| S1 | *d04* | *d10* |
| S2 | *d02* | *d04* |
| S2 | *d07* | *d10* |
| S3 | *d03* | *d10* |
| S4 | *d04* | *d10* |
| S5 | *d02* | *d10* |

SP_FROM_TO

| S# | P# | FROM | TO |
|---|---|---|---|
| S1 | P1 | *d04* | *d10* |
| S1 | P2 | *d05* | *d10* |
| S1 | P3 | *d09* | *d10* |
| S1 | P4 | *d05* | *d10* |
| S1 | P5 | *d04* | *d10* |
| S1 | P6 | *d06* | *d10* |
| S2 | P1 | *d02* | *d04* |
| S2 | P1 | *d08* | *d10* |
| S2 | P2 | *d03* | *d03* |
| S2 | P2 | *d09* | *d10* |
| S3 | P2 | *d08* | *d10* |
| S4 | P2 | *d06* | *d09* |
| S4 | P4 | *d04* | *d08* |
| S4 | P5 | *d05* | *d10* |

图 23-3　供应商 - 发货数据库（第一个完全时态化版本，显式地用到了 FROM 和 TO 属性）实例

关于这个完全时态化的例子，还有一些方面需要强调。具体地说，我们假设：

1）任何供应商都不能在某一天结束一个合约，而紧接着在第二天开始另一个合约。

2）任何供应商都不能在同一时间签订两份不同的合约。

3）供应商的合约是可修整的——也就是说，一个供应商可以签订了一份合约，并且合约的终止日期目前还是未知的。

**约束**（第一个时态数据库）：首先，观察图 23-3 中双下划线的属性，两个关系变量的主码中都包括了 FROM 属性。确实，（举个例子）S_FROM_TO 的主码不能只是 {S#}，否则就无法处理诸如一个供应商，比如 S2，在两个或多个单独的时间区间内签订了合约这样的问题。SP_FROM_TO 的情况也是类似的。注意：也可以用 TO 代替 FROM 作为主码属性；事实上，在 S_FROM_TO 和 SP_FROM_TO 中都有两个候选码，没有明显的原因决定一定要选择哪个作为主码 [9.14]。这里选择 FROM 纯粹是为了使事情更明确。

下面是 **Tutorial D** 的定义：

```
VAR S_FROM_TO
        BASE RELATION { S# S#, FROM DATE, TO DATE }
    KEY { S#, FROM }
    KEY { S#, TO } ;
VAR SP_FROM_TO
        BASE RELATION { S# S#, P# P#, FROM DATE, TO DATE }
    KEY { S#, P#, FROM }
    KEY { S#, P#, TO } ;
```

接下来，我们需要防止一种荒谬错误的出现：就是在 FROM_TO 这对值中，TO 的值小于 FROM 的值：

```
CONSTRAINT S_FROM_TO_OK
   IS_EMPTY ( S_FROM_TO WHERE TO < FROM ) ;

CONSTRAINT SP_FROM_TO_OK
   IS_EMPTY ( SP_FROM_TO WHERE TO < FROM ) ;
```

到目前为止讨论的约束还不能做到我们所希望它们做到的每件事。比如，考虑关系变量 S_FROM_TO。很显然，如果其中存在供应商 S*x* 的一个元组，FROM 的值为 *f*，TO 的值为 *t*，那么就不希望在该关系变量中还存在供应商 S*x* 的另一个元组，表明 S*x* 在 *f* 的前一天或者 *t* 的后一天也是处在合同约束下的。举个例子，供应商 S1 在 S_FROM_TO 中有一个元组，FROM = *d04*，TO = *d10*。{S#，FROM} 是候选码，但这不足以防止 S1 有另一个“重叠”元组出现这种情况，比如 FROM = *d02*，TO = *d06*，这表明在 d*04* 之前与 S1 建立了合同关系。很显然，我们希望将这两个元组归并为 S1 的一个元组，FROM = *d02*，TO = *d10*。

你现在应该已经猜到归并元组这个想法非常重要。确实，在前面那个例子中，不对两个元组进行归并，几乎等于允许副本的存在！副本意味着“将相同的事情说两遍”。供应商 S1 的那两个元组在 FROM-TO 区间上是重叠的，确实等同于“将相同的事情说两遍”；具体来说，它们都表明了供应商 S1 在日期 4、5、6 是签订了合约的。如果两个元组同时出现，那么关系变量 S_FROM_TO 就违背了它自己的谓词语义。在 23.7 节中我们会再来看这个问题并详细讨论它。

接下来，{S#，FROM} 是候选码这一事实，也不足以防止出现 S1 存在另一个“邻接”元组的情况，比如 FROM = *d02*，TO = *d03*，这也表明在日期 4 的前一天 S1 就已经签订了合约。同前面一样，我们希望将两个元组归并为一个——否则，关系变量 S_FROM_TO 也会违背它自己的谓词语义。我们还是会在 23.7 节中再来看这个问题并详细讨论它。

下面给出防止重叠和邻接出现的约束：

```
CONSTRAINT XFT1
   IS_EMPTY
      ( ( ( S_FROM_TO RENAME ( FROM AS F1, TO AS T1 ) ) JOIN
          ( S_FROM_TO RENAME ( FROM AS F2, TO AS T2 ) ) )
            WHERE ( T1 ≥ F2 AND T2 ≥ F1 ) ) OR
                  ( F2 = T1+1 OR F1 = T2+1 ) ) ;
```

现在我们开始真正看这个问题！这个约束相当复杂——更不用说将 T1 + 1 作为 T1 的直接后继这一事实，这一点我们将在下一节进行讨论。此外，给定一个像图 23-3 那样的完全时态数据库，我们可能需要规定很多像约束 XFT1 一样具有普遍性质的约束，因此我们还是希望有一些好的速记符号。*注意*：事实上，约束 XFT1 还存在另一个问题：如果 T1 恰好表示最后一天，那么 T1 + 1 的含义是什么？

接下来要注意，关系变量 SP_FROM_TO 中的属性组合 {S#，FROM} 不是从 SP_FROM_TO 到 S_FROM_TO 的外码（尽管在 S_FROM_TO 的主码中也包含这些属性）。可是，如果在 SP_FROM_TO 中包含了某个供应商，那么要确保它也包含于 S_FROM_TO 中[⊖]：

```
CONSTRAINT XFT2
   SP_FROM_TO { S# } ⊆ S_FROM_TO { S# } ;
```

这个约束是**包含依赖**（inclusion dependency）［11.4］的一个例子。就像第 11 章说的那样，包含依赖可被视为参照约束的一个普遍形式。很明显，任何一个像图 23-3 那样的时态数据库都可能包含了（至少是隐含了）大量这样的依赖。

但是仅仅有约束 XFT2 还不足够——我们还需要确保，如果关系变量 SP_FROM_TO 显示了某个供应商在某个时间区间内可以供应某种零件，那么关系变量 S_FROM_TO 就应该显示这个供应商在同一时间区间内已经签订了合约：

---

⊖ 其中“⊆”表示“包含于”。——译者注

```
CONSTRAINT XFT3
   COUNT ( SP_FROM_TO { ALL BUT P# } ) =
   COUNT ( ( ( SP_FROM_TO RENAME ( FROM AS SPF, TO AS SPT ) )
                                                { ALL BUT P# }

              JOIN
              ( S_FROM_TO RENAME ( FROM AS SF, TO AS ST ) ) )
         WHERE SF ≤ SPF AND ST ≥ SPT ) ;
```

这里的直觉就是，如果关系变量 SP_FROM_TO 包含了一个元组，显示供应商 S*x* 从日期 *spf* 到日期 *spt* 可以供应某种零件，那么关系变量 S_FROM_TO 必须包含一个元组，表明供应商 S*x* 在相同的这段时间里是签订了合约的。（这里我们假设到目前为止讨论的所有约束都是有效的！）我们有意不对这个约束提供更深一层的分析，而是再次关注它的复杂性，并且在一个现实的数据库中我们可能需要规定很多同样有普遍性质的约束。因此，我们还是希望能有好的、适用的速记符号。

**查询**（第一个时态数据库）：以下是查询 *A* 和查询 *B* 的完全时态化版本。

- 查询 *A*：查询至少在某一个时间区间可以供应某种零件的供应商的三个属性 S#-FROM-TO，其中 FROM 和 TO 合起来表示这样的一个时间区间。注意，查询结果可能包含了同一个供应商的多个元组（但是，当然是在不同的时间区间；另外，这些时间区间既不会邻接也不会重叠）。
- 查询 *B*：查询至少在某一个时间区间内不能供应任何零件的供应商的三个属性 S#-FROM-TO，其中 FROM 和 TO 合起来表示这样的一个时间区间（查询结果还是可能包含了同一个供应商的多个元组）。

同我们一样，你可能需要一些时间来说服自己，也可能根本不想尝试这些查询！然而，如果真的尝试一下，你会发现这些查询还是可以表述出来的，尽管要极为费力才能实现，但很明显需要某些速记［表示符号］是很值得的。

简而言之，时态数据的问题在于它会很快导致一些表述起来过度复杂的约束和查询——更不用说超出本章讨论范围的更新了。过度复杂是说除非系统提供适当的速记符号，否则查询实现起来很难，而目前现有的商用数据库管理系统还没有提供这些速记符号。

## 23.3　时间区间

我们从现在就开始开发一些合适的速记符号［］。最初的、同时也是最基本的步骤，就是将时间区间本身视为一个值，而不是像此前那样作为一对分离的起始点和终止点值来处理。比如，考虑从日期 4 到日期 10 的这段时间区间。为了强调将这一时间区间本身看作一个值，我们会用非正式的缩写表达式［*d04*:*d10*］来表示它，而不是使用诸如“从日期 4 到日期 10 的时间区间”这种累赘的表述方式。这种符号的具体含义如下：

- ［*d04*:*d10*］是一个区间值，或者简单地说只是一个区间。
- *d04* 和 *d10* 这两个数值分别是区间值的起始点和终止点。
- 此区间值是某种确定的区间类型。
- 这种区间类型是定义在某种点类型之上的。

马上我们会准确定义所有这些术语。首先要看看，如果采取了这种方法，我们的示例数据库会发生什么变化。如图 23-4 所示。

谓词语义如下：

- S_DURING：供应商 S#从 DURING 的起始点到 DURING 的终止点这段时间区间内签订了合约。
- SP_DURING：供应商 S#从 DURING 的起始点到 DURING 的终止点这段时间区间内可以供应零件 P#。

现在进行正式定义。首先，如果下面的所有定义都适用于一个给定的类型 *T*，那么类型 *T* 可被用于点类型（类型 *T* 的值可被称为点）：

- 一个**全序**，根据这个全序，类型 *T* 的每一对数值 *v1* 和 *v2* 之间都可以定义一个运算符

“>”；如果 *v1* 和 *v2* 是不同值，那么表达式“*v1* > *v2*”和“*v2* > *v1*”中一定有一个是正确的，而另一个是错误的。注意：从第 5 章我们知道，运算符“=”也是为 *T* 定义的。因此，加上前面定义的“>”（也包括可用的布尔运算符 NOT），我们可以合理地假设所有常用的比较运算符“=”，“≠”，“>”，“≥”，“<”以及“≤”对类型 *T* 的每对值都是适用的。

- 零元运算符 **FIRST_*T*** 和 **LAST_*T***，根据前面提到的全序，分别返回类型 *T* 的第一个和最后一个数值。
- 一元运算符 **NEXT_*T*** 和 **PRIOR_*T***，对于一个给定的 *T* 类型的值，根据前面提到的全序，分别返回它的后继和前继。注意：NEXT_*T* 是类型 *T* 的**后继函数**；当然，如果 *p* = LAST_*T*( )，则 NEXT_*T*(*p*) 是无意义的。同样地，如果 *p* = FIRST_*T*( )，则 PRIOR_*T*(*p*) 是无意义的。

S_DURING

| S# | DURING |
|---|---|
| S1 | [*d04*:*d10*] |
| S2 | [*d02*:*d04*] |
| S2 | [*d07*:*d10*] |
| S3 | [*d03*:*d10*] |
| S4 | [*d04*:*d10*] |
| S5 | [*d02*:*d10*] |

SP_DURING

| S# | P# | DURING |
|---|---|---|
| S1 | P1 | [*d04*:*d10*] |
| S1 | P2 | [*d05*:*d10*] |
| S1 | P3 | [*d09*:*d10*] |
| S1 | P4 | [*d05*:*d10*] |
| S1 | P5 | [*d04*:*d10*] |
| S1 | P6 | [*d06*:*d10*] |
| S2 | P1 | [*d02*:*d04*] |
| S2 | P1 | [*d08*:*d10*] |
| S2 | P2 | [*d03*:*d03*] |
| S2 | P2 | [*d09*:*d10*] |
| S3 | P2 | [*d08*:*d10*] |
| S4 | P2 | [*d06*:*d09*] |
| S4 | P4 | [*d04*:*d08*] |
| S4 | P5 | [*d05*:*d10*] |

图 23-4　供应商 - 发货数据库（使用了区间的第二个完全时态版本）实例

接下来，我们定义**区间类型发生器**[⊖]。如果 *T* 是一个点类型，则 INTERVAL_*T* 是通过调用类型 *T* 的类型发生器得到的一个时间区间类型。同所有的类型发生器一样，时间区间与一组普通的可能的表示法，一组普通的操作符以及一组普通的约束联系起来，所有的这些都应用于每种从发生器得到的类型。更详细地说：

- 我们只考虑一个可能的表示法：INTERVAL_*T* 类型的任意值——即此种类型的任意时间**区间**——可能被表示为类型 *T* 的一对值，分别对应于时间区间的起始点和终止点。下面是**选择**操作符的相应语法：

```
INTERVAL_T ( [ b : e ] )
```

这里 *b* 和 *e* 是类型为 *T* 的表达式，整个选择操作返回了以这两个表达式为起始点和终止点的时间区间。

- 下面是“THE_ 操作符” **BEGIN** 和 **END** 对应的语法：

```
BEGIN ( i )
END ( i )
```

这里 *i* 是类型为某种时间区间的表达式，而这两种操作返回了表达式指定的时间区间的起始点和终止点。注意：BEGIN 和 END 是真正的“THE_”操作符，在 **Tutorial D** 中我们通常将它们写作 THE_BEGIN 和 THE_END。这里以及整个这一章，我们用 BEGIN 和 END 来代替，主要是为了与此领域中的其他著作保持连贯性。

- 其他操作符包括“: =”；一组布尔操作符，被称为 Allen 的操作符（见本节后面部分），特别是包括了“=”；各种其他操作符（见本节后面部分）。
- 我们只考虑一个一般性的约束：如果 *i* 是一个时间区间，则 BEGIN (*i*) ≤ END (*i*)。结论就是，时间区间是非空的——这个区间至少包含一个点。另一个结论是，不再需要这样的显式约束，即“防止在一对 FROM_TO 值对中出现 TO 值小于 FROM 值的情况”。

现在应该很清楚，DATE 类型是满足要求的，可以作为一种点类型使用。因此，INTERVAL_DATE 是有效的时间区间类型，关系变量 S_DURING 和 SP_DURING 的定义可以如下：

---

⊖ 需要指出，时间区间类型发生器是本章介绍的唯一一个非速记结构。因此我们对于时态数据库的方法——不同于文献中描述的其他一些方法——与经典关系模型相比没有任何变化（尽管后者包括了一些普遍性，参见 23.5 节和 23.7 节）。

```
VAR S_DURING BASE RELATION
  { S# S#, DURING INTERVAL_DATE }
  KEY { S#, DURING } ;

VAR SP_DURING BASE RELATION
  { S# S#, P# P#, DURING INTERVAL_DATE }
  KEY { S#, P#, DURING } ;
```

注意，这些定义还很不完整！23.7 节中我们会对这个问题进行详细说明。然而可以看到的是，我们已经对一个问题进行了定义，那就是在两个候选码中如何任意选择一个作为主码。对于 23.2 节中的其他约束和查询，它们的相似性可以通过图 23-4 的数据库推导出来，这就要归功于 BEGIN 和 END 这两个操作符的存在了。但在这里我们不做任何推导，因为我们的目标是提出一种更好的描述约束和查询的方法，而它恰好是目标的一部分。

强调一下，DATE 类型是一种有效的点类型——但没有要求说点类型要指定为“日期时间”类型，也没有指定区间必须是“日期时间”区间。实际上，虽然区间是处理时态数据库所必需的基本抽象，但是很明显，区间这个概念应该有更广泛的适用性；也就是说，区间还有很多其他的应用，在这些应用中，区间并不一定要具有时态性。下面有一些例子：

- 缴税等级用需纳税的收入的范围来表示——即区间的起始点和终止点（以及中间的所有点）都是货币值。
- 操作机器要在一定的温度和电压范围内——即区间内包含的点分别是温度和电压。
- 动物在它们的眼睛和耳朵能够接收到的光和声波频率范围内发生不同的变化。
- 很多自然现象的发生和测量，都要在土壤或海洋的一定深度，或者海拔的一定高度范围内。

尽管在本章中，我们主要关注的是时态区间，但很多的讨论实际上都与通常意义上的区间相关。然而，由于篇幅限制不允许讨论过于详细，因此这里我们只讨论两个非时态区间类型的例子：

- `INTERVAL INTEGER`

  这里的点类型是 INTEGER；后继函数是“下一个整数”（即“加一”），这一区间类型的值是形如 $[b : e]$ 的区间，其中 $b$ 和 $e$ 是 INTEGER 类型的值，且 $b \leqslant e$。

- `INTERVAL_MONEY`

  假设这里的 MONEY 代表了一种用美元和美分来衡量的货币量。后继函数是“加上一美分”。这一区间类型的值是形如 $[b : e]$ 的区间，其中 $b$ 和 $e$ 是 MONEY 类型的值，且 $b \leqslant e$。

**1. 点和区间上的操作**

现在开始定义一些在点和区间上有用的操作（在前面讨论的基础上）。修改例子使它可以说明这些操作的功能，这个留做练习。术语：$T$ 为一种点类型，$p$、$p1$ 和 $p2$ 是类型为 $T$ 的值；非正式的情况下，我们分别用 $p+1$ 和 $p-1$ 表示 $p$ 的后继和前驱。此外，$i$、$i1$ 和 $i2$ 是类型为 INTERVAL_$T$ 的区间，而 $b$、$b1$、$b2$ 以及 $e$、$e1$ 和 $e2$ 分别是 $i$、$i1$ 和 $i2$ 的起始点和终止点；非正式的情况下，我们分别用 $[b : e]$、$[b1 : e1]$ 和 $[b2 : e2]$ 表示 $i$、$i1$ 和 $i2$。则：

- IS_NEXT_$T(p1,p2)$ 为真，当且仅当 $p1$ 是 $p2$ 的直接后继。IS_PRIOR_$T(p1,p2)$ 为真，当且仅当 IS_NEXT_$T(p2,p1)$ 为真——即 IS_PRIOR_$T(p1,p2)$ ≡ IS_NEXT_$T(p2,p1)$。
- 如果 $p1 < p2$ 为真，MAX$(p1,p2)$ 的返回值为 $p2$，否则返回值为 $p1$；如果 $p1 < p2$ 为真，MIN$(p1,p2)$ 的返回值为 $p1$，否则返回值为 $p2$。
- $p \in i$ 为真，当且仅当 $b \leqslant p$ 和 $p \leqslant e$ 都为真——即 $p \in i \equiv (b \leqslant p$ AND $p \leqslant e)$。另外，$i \ni p \equiv p \in i$。注意：符号 $\in$ 和符号 $\ni$ 分别读作“包含于”和“包含”。
- COUNT $(i)$ 返回满足条件 $p \in i$ 的点 $p$ 的个数。
- 当且仅当 COUNT $(i) = 1$ 时，$i$ 是一个单位区间。POINT FROM $i$ 返回单位区间 $i$ 中的唯一点 $p$。
- PRE $(i)$ 和 POST $(i)$ 分别返回 $b-1$ 和 $e+1$。注意：PRE $(i)$ 和 POST $(i)$ 分别是 PRIOR_$T$ (BEGIN $(i)$) 和 NEXT_$T$ (END $(i)$) 的速记。

下面要定义一些操作符来测试两个区间是否相等、是否重叠，等等。这类操作符中的多数都是 Allen 在文献［23.1］中第一次提出来的，所以它们一般被称为 **Allen 的操作符**；但在这里，我们并不总是遵循 Allen 的命名。我们会用含义相同但更简洁的术语来表述这些操作符，但为了更直观地理解它们，你可能要试着画一些图。

- 相等（=）：（*i1* = *i2*） ≡ （*b1* = *b2* AND *e1* = *e2*）
- 包含（⊇）和包含于（⊆）：（*i1* ⊇ *i2*） ≡ （*b1* ≤ *b2* AND *e1* ≥ *e2*）；（*i2* ⊆ *i1*） ≡ （*i1* ⊇ *i2*）
- 完全包含（⊃）和完全包含于（⊂）：（*i1* ⊃ *i2*） ≡ （*i1* ⊇ *i2* AND *i1* ≠ *i2*）；（*i2* ⊂ *i1*） ≡ （*i1* ⊃ *i2*）
- BEFORE 和 AFTER：（*i1* BEFORE *i2*） ≡ （*e1* < *b2*）；（*i1* AFTER *i2*） ≡ （*i2* BEFORE *i1*）
- MEETS：（*i1* MEETS *i2*） ≡ （*b2* = *e1* + 1 OR *b1* = *e2* + 1）
- OVERLAPS：（*i1* OVERLAPS *i2*） ≡ （*b1* ≤ *e2* AND b2 ≤ *e1*）
- MERGES：（*i1* MERGES *i2*） ≡ （*i1* OVERLAPS *i2* OR *i1* MEETS *i2*）
- BEGINS：（*i1* BEGINS *i2*） ≡ （*b1* = *b2* AND *e1* ≤ *e2*）
- ENDS：（*i1* ENDS *i2*） ≡ （*e1* = *e2* AND *b1* ≥ *b2*）

最后，我们定义一些区间上有用的二元操作符，它们的返回值都是区间：称为类似操作符 UNION、INTERSECT 和 MINUS 的区间版本。每种操作符都以两个同类型的区间为操作数，返回同类型的另一个区间作为结果（这里我们举一些例子）。

- UNION：*i1* UNION *i2*，如果 *i1* MERGES *i2* 为真，则返回［MIN（*b1*，*b2*）：MAX（*e1*，*e2*）］；否则返回未定义。比如，［*d04*：*d08*］和［*d06*：*d10*］的并集返回值为［*d04*：*d10*］；［*d02*：*d03*］和［*d06*：*d10*］的并集返回值为未定义。
- INTERSECT：*i1* INTERSECT *i2*，如果 *i1* OVERLAPS *i2* 为真，则返回［MAX（*b1*，*b2*）：MIN（*e1*，*e2*）］；否则返回未定义。比如，［*d04*：*d08*］和［*d06*：*d10*］的交集返回值为［*d06*：*d08*］；［*d02*：*d03*］和［*d06*：*d10*］的交集返回值为未定义。
- MINUS：*i1* MINUS *i2*，如果 *b1* < *b2* 和 *e1* ≤ *e2* 都为真，则返回［*b1*：MIN（*b2* - *1*，*e1*）］；如果 *b1* ≥ *b2* 和 *e1* > *e2* 都为真，则返回［MAX（*e2* + *1*，*b1*）：*e1*］；否则返回未定义。比如，［*d04*：*d08*］和［*d06*：*d10*］（按照这个顺序）的差，返回值为［*d04*：*d05*］；［*d06*：*d10*］和［*d04*：*d08*］（按照这个顺序）的差，返回值为［*d09*：*d10*］；［*d02*：*d03*］和［*d06*：*d10*］的差，返回值为未定义。

**2. 查询示例**

下面以一些查询示例来结束这一节，举例说明前面定义的某些操作符的用法。首先，第一个例子是考虑一个查询“取得能够在日期 8 供应零件 P2 的供应商数目”。下面是在图 23-4 的数据库上，该查询的一个可能的表达方式：

```
( SP_DURING WHERE P# = P# ('P2')
            AND   d08 ∈ DURING ) { S# }
```

说明：外面括号里的表达式，把出现在关系变量 SP_DURING 中的元组集合，限定为那些 P#的值为 P2 且时间区间 DURING 的值里包含了日期 8 的元组。接下来，这些元组的集合投影到属性 S#上得到所要求的结果。注意：实际上，表达式中的 *d08* 可以替换为一个合适的 DATE 选择子。

第二个例子是查询“取得能够在同一时间供应同种零件的供应商对”的一个可能的表达方式：

```
WITH ( SP_DURING RENAME ( S# AS X#, DURING AS XD ) ) AS T1 ,
     ( SP_DURING RENAME ( S# AS Y#, DURING AS YD ) ) AS T2 ,
     ( T1 JOIN T2 ) AS T3 ,
     ( T3 WHERE XD OVERLAPS YD ) AS T4 ,
     ( T4 WHERE X# < Y# ) AS T5 :
T5 { X#, Y# }
```

说明：T1 是关系变量 SP_DURING 的当前关系值，只有一点不同，就是将属性 S#和 DURING 分别重命名为 X#和 XD；关系 T2 也是一样，只是新的属性名是 Y#和 YD。关系 T3 是 T1 和 T2 在

零件号上的连接。关系 T4 是将 T3 限制为 XD 和 YD 区间重叠的那些元组（表示供应商不仅要能够提供相同的零件，而且要按照要求，能够在相同的时间提供同种零件）。关系 T5 是将 T4 限制为供应商号 X#小于 Y#的那些元组（比较第 7 章的例子 7.5.5）。最后通过在 X#和 Y#上进行投影得到要求的结果。

第三个例子，假设我们不止要查询能够在同一时间供应同种零件的供应商对，还要知道是何种零件和时间。下面是一个可能的表达方式：

```
WITH ( SP_DURING RENAME ( S# AS X#, DURING AS XD ) ) AS T1 ,
     ( SP_DURING RENAME ( S# AS Y#, DURING AS YD ) ) AS T2 ,
     ( T1 JOIN T2 ) AS T3 ,
     ( T3 WHERE XD OVERLAPS YD ) AS T4 ,
     ( T4 WHERE X# < Y# ) AS T5 ,
     ( EXTEND T5 ADD ( XD INTERSECT YD ) AS DURING ) AS T6 :
T6 { X#, Y#, P#, DURING }
```

说明：关系 T1、T2、T3、T4 和 T5 与前面的例子相同。接下来的 EXTEND 计算相应的时间区间，最后进行投影得到要求的结果。

## 23.4　归并和反归并关系

本节我们介绍两种新的（而且是非常重要的）关系操作符，归并（PACK）和反归并（UNPACK）。而作为这两种操作符的基础，我们首先需要离开主题讨论一下两个它们的简单版本，分别称为压缩（COLLAPSE）和扩展（EXPAND）。另外，为便于描述，后面的会按照相反的顺序讨论这两个简单版本的操作符。

### 1. EXPAND 和 COLLAPSE

这里描述的 EXPAND 和 COLLAPSE[⊖] 都只有一个操作数，且操作数是一个元组中含有区间的一元关系，而产生的结果是另一个同样类型的关系。例如，假设关系 *r* 如下：

| DURING |
|---|
| [*d06*:*d09*] |
| [*d04*:*d08*] |
| [*d05*:*d10*] |
| [*d01*:*d01*] |

则 EXPAND *r* 和 COLLAPSE *r* 产生的结果如下：

EXPAND *r*

| DURING |
|---|
| [*d01*:*d01*] |
| [*d04*:*d04*] |
| [*d05*:*d05*] |
| [*d06*:*d06*] |
| [*d07*:*d07*] |
| [*d08*:*d08*] |
| [*d09*:*d09*] |
| [*d10*:*d10*] |

COLLAPSE *r*

| DURING |
|---|
| [*d01*:*d01*] |
| [*d04*:*d10*] |

说明：一元关系 *r* 有唯一的属性 DURING，且 DURING 是区间值。则 *r* 的扩展和压缩形式都是与 *r* 同种类型的一元关系，定义如下：

- **扩展**形式是一个关系 *rx*，它是由且仅由形如 [*p*: *p*]、包含了一个单位区间的元组组成的，其中 *p* 是 *r* 中某一元组的某一区间里的一点；
- *r* 的**压缩**形式是满足下列条件的关系 *rc*：

1）关系 *r* 和 *rc* 有相同的扩展形式。

2）*rc* 中任意两个不同元组，它们各自的区间为 *i1* 和 *i2*，*i1* MERGES *i2* 都为假。同样，*rc* 中

⊖ 这两个操作符更普遍的形式在参考文献［23.4］中有描述。

任意两个不同元组，它们各自的区间 *i1* 和 *i2*，*i1* UNION *i2* 都是未定义的。（由此得出结论，*rc* 可以通过 *r* 计算出来，方法就是将 *r* 中的元组对 *t1* 和 *t2* 替换为一个元组 *t*，其中 *t* 包含了 *t1* 和 *t2* 区间的并集，替换一直进行到不会再发生此种替换为止。）

现在我们进一步看一下这些概念。为了简便起见，假设在本小节的剩余部分中，处理的关系都是元组中包含区间的一元关系。下面我们定义一个重要的概念等价（equivalence）：

- 两个关系 *r1* 和 *r2* 是**等价**的，当且仅当 *r1* 中元组的区间包含的所有点，与 *r2* 中元组的区间包含的所有点相等。

有了这个定义以及前面关于扩展和压缩形式的定义，下面几点就是很清楚了：

- 对任意给定的一个关系 *r*，总是存在 *r* 的相应的扩展形式。
- 这一扩展形式与 *r* 是等价的。事实上可以这样说，两个关系是等价的，当且仅当它们有相同的扩展形式。
- 扩展形式是唯一的。准确地说，它是区间所有最小可能长度的唯一等价关系。
- 直观上，*r* 的扩展形式使我们将注意力集中在 *r* 原子层的信息内容上，而不必理会信息可能通过很多种方式集合成“簇”。
- 如果 *r* 是空的，则 *r* 的扩展形式也是空的。

同样：

- 对任意给定的一个关系 *r*，总是存在 *r* 的相应的压缩形式。
- 这一压缩形式与 *r* 是等价的。事实上可以这样说，两个关系是等价的，当且仅当它们有相同的压缩形式。
- 压缩形式是唯一的；准确地说，它是拥有最小可能集的基数的唯一等价关系。
- 直观上，*r* 的压缩形式使我们将注意力集中在 *r* 压缩（“簇的”）形式的信息内容上，而不必理会不同的“簇”可能邻接或者重叠。
- 如果 *r* 是空的，则 *r* 的压缩形式也是空的。

顺便说一句，不要错误地认为 EXPAND 和 COLLAPSE 是彼此的逆过程。实际上，无论 EXPAND（COLLAPSE *r*）还是 COLLAPSE（EXPAND *r*），通常都与 *r* 是相等的（同样它们与 *r* 是等价的）。下面的恒等式是显而易见的：

- `EXPAND ( COLLAPSE r ) = EXPAND r`
- `COLLAPSE ( EXPAND r ) = COLLAPSE r`

由此得出结论，在某一关系 *r* 上的操作序列“先压缩后扩展”、或者“先扩展后压缩”，其中的第一个操作可以被简单忽略掉，这一事实对于优化目标是很有用的（尤其当第一个操作是 EXPAND 的时候）。

以下是本小节结束前的最后两点：

- 文献［23.4］表明，EXPAND 和 COLLAPSE 都可以通过关系代数中已有的操作符来定义。换句话说，它们都是速记。
- 文献［23.4］还表明，在空（nullary）关系而不是一元关系上定义 EXPAND 和 COLLAPSE 操作符，是非常有意义的。此处我们忽略细节分析，而只进行简单定义：如果 *r* 是一个空关系，则 EXPAND *r* 和 COLLAPSE *r* 都返回 *r*。同时定义两个空关系是等价的，当且仅当它们相等。

**2. 归并和反归并**

归并和反归并的最普遍形式都是只有一个操作数，此操作数为一个 *n* 元关系，其中的一个属性是区间值，最后产生的结果是另一个同种关系。例如，假设关系 *r* 如下：

| S# | DURING |
|---|---|
| S2 | [*d02*:*d04*] |
| S2 | [*d03*:*d05*] |
| S4 | [*d02*:*d05*] |
| S4 | [*d04*:*d06*] |
| S4 | [*d09*:*d10*] |

那么 PACK *r* ON DURING 和 UNPACK *r* ON DURING 生成的结果如下：

PACK *r* ON DURING

| S# | DURING |
|---|---|
| S2 | [*d02*:*d05*] |
| S4 | [*d02*:*d06*] |
| S4 | [*d09*:*d10*] |

UNPACK *r* ON DURING

| S# | DURING |
|---|---|
| S2 | [*d02*:*d02*] |
| S2 | [*d03*:*d03*] |
| S2 | [*d04*:*d04*] |
| S2 | [*d05*:*d05*] |
| S4 | [*d02*:*d02*] |
| S4 | [*d03*:*d03*] |
| S4 | [*d04*:*d04*] |
| S4 | [*d05*:*d05*] |
| S4 | [*d06*:*d06*] |
| S4 | [*d09*:*d09*] |
| S4 | [*d10*:*d10*] |

可以观察到，在非正式的情况下，每个结果都表示了与原关系 *r* 相同的信息，但是：

- 在 PACK 的情况下，信息按照以下这种方式重新排列：一个给定的供应商的任意两个 DURING 值都不会邻接或者重叠。
- 在 UNPACK 的情况下，信息按照以下这种方式重新排列：每个 DURING 值（一个给定的供应商的 DURING 区间更是如此）都是一个特定的单位区间。

COLLAPSE 和 EXPAND 与 PACK 和 UNPACK 操作的关系是显而易见的。另一件显而易见的事情是，在某种特定意义上（在本节的最后，我们会准确说明这个意义），原关系 *r* 与对应的归并和反归并形式都是等价的。

现在我们集中看一下 PACK。首先是查询 *A* 根据图 23-4 所示的数据库的再声明：

- 查询 *A*：取得至少在某一个时间区间里可以供应至少一种零件的供应商的属性对 S#-DURING，其中 DURING 指明了这样的一个区间。

给定图 23-4 的示例数据，要求的结果如下：

| S# | DURING |
|---|---|
| S1 | [*d04*:*d10*] |
| S2 | [*d02*:*d04*] |
| S2 | [*d08*:*d10*] |
| S3 | [*d08*:*d10*] |
| S4 | [*d04*:*d10*] |

这个关系是关系变量 SP 在 DURING 上的归并形式的某个投影，称为在 S#和 DURING 上的投影。并且，我们最终可以通过一个简单的表达形式得到这一结果：

```
PACK T1 ON DURING
```

其中 T1 是指定的投影。但是，我们需要逐步达到这个目标。第一步是：

```
WITH SP_DURING { S#, DURING } AS T1 :
```

这步是生成要求的投影（它只是“投影掉”与查询不相关的零件号）。根据示例数据值，T1 如下：

| S# | DURING |
|---|---|
| S1 | [*d04*:*d10*] |
| S1 | [*d05*:*d10*] |
| S1 | [*d09*:*d10*] |
| S1 | [*d06*:*d10*] |
| S2 | [*d02*:*d04*] |
| S2 | [*d08*:*d10*] |
| S2 | [*d03*:*d03*] |
| S2 | [*d09*:*d10*] |
| S3 | [*d08*:*d10*] |
| S4 | [*d06*:*d09*] |
| S4 | [*d04*:*d08*] |
| S4 | [*d05*:*d10*] |

这个关系包含了冗余信息；比如，供应商 S1 在日期 6 这天提供某些东西，这件事我们至少被告知了三次（约束下的结果不能包含这样的冗余）。

下一步如下：

```
WITH ( T1 GROUP { DURING } AS X ) AS T2 :
```

T2 如下：

| S# | X |
|---|---|
| S1 | DURING<br>[d04:d10]<br>[d05:d10]<br>[d09:d10]<br>[d06:d10] |
| S2 | DURING<br>[d02:d04]<br>[d08:d10]<br>[d03:d03]<br>[d09:d10] |
| S3 | DURING<br>[d08:d10] |
| S4 | DURING<br>[d06:d09]<br>[d04:d08]<br>[d05:d10] |

T2 的属性 X 是关系值（relation-valued），因此我们可以将 COLLAPSE 操作符应用到这个属性值的一元关系上：

```
WITH ( EXTEND T2 ADD COLLAPSE ( X ) AS Y )
                                { ALL BUT X } AS T3 :
```

T3 如下（由于说明“ALL BUT X”，属性 X 已经通过投影去除了）：

| S# | Y |
|---|---|
| S1 | DURING<br>[d04:d10] |
| S2 | DURING<br>[d02:d04]<br>[d08:d10] |
| S3 | DURING<br>[d08:d10] |
| S4 | DURING<br>[d04:d10] |

最后，撤消分组：

```
T3 UNGROUP Y
```

上面的表达式产生所需的结果。换句话说，现在将所有步骤归并在一起（经过稍许简化），通过

计算下面全部表达式来得到结果：

```
WITH SP_DURING { S#, DURING } AS T1 ,
   ( T1 GROUP { DURING } AS X ) AS T2 ,
   ( EXTEND T2 ADD COLLAPSE ( X ) AS Y ) { ALL BUT X } AS T3 :
T3 UNGROUP Y
```

现在可以定义 PACK 操作符（当然是速记）。语法如下：

```
PACK r ON A
```

这里 *r* 是一个关系表达式，*A* 是表达式指定的关系的一个区间属性。此操作符的语义是通过分组、扩展、投影和逆分组操作来定义的，并通过它从 T1 上得到结果：

```
PACK r ON A  ≡  WITH ( r GROUP { A } AS X ) AS R1 ,
                     ( EXTEND R1 ADD COLLAPSE ( X ) AS Y )
                                          { ALL BUT X } AS R2 :
                R2 UNGROUP Y
```

如早前提到的那样，查询 A 可表示为：

```
PACK SP_DURING { S#, DURING } ON DURING
```

注意：从定义可以清楚地看出，在某个属性 *A* 上归并一个关系包括了将关系按照除了 *A* 以外的其他所有属性分组这一过程。（回顾第 7 章，表达式“T1 GROUP ｛DURING｝ …”可读作“T1 按 S#分组”，S#是除了 GROUP 子句中提到的属性之外的所有其他属性。）但需要注意的是，*r* GROUP ｛*A*｝ …对每个不同值的 *B*（*B* 是 *r* 中除了 *A* 以外的所有属性），确保只返回一个元组作为结果，PACK *r* ON *A* 对任意给定的 *B* 值可能返回几个元组作为结果。举例来说，查询 A 进行 PACK 操作的结果中，供应商 S4 有两个元组。

现在来看看 UNPACK 和查询 *B*：

- 查询 B：取得至少在某一个时间区间内不能供应任何零件的供应商的属性对 S#-DURING，其中 DURING 指明了这样的一个区间。

现在可以看出来我们需要做的，从本质上来说就是查找这样的一些 S#-DURING 属性对：出现、或者没有出现而被 S_DURING 暗指，以及没有被 SP_DURING 暗指。这一简要的描述足以恰当地说明，本质上我们需要做的就是，执行两个反归并操作，比较结果的不同，对不同之处进行再归并。所以我们首先介绍一下 UNPACK 操作符：

```
UNPACK r ON A  ≡  WITH ( r GROUP { A } AS X ) AS R1 ,
                       ( EXTEND R1 ADD EXPAND ( X ) AS Y )
                                          { ALL BUT X } AS R2 :
                  R2 UNGROUP Y
```

这个定义与 PACK 的定义是相同的，除了在第二行中出现的是 EXPAND 而非 COLLAPSE。我们把这一表达式的结果称为关系 *r* 在 *A* 上的反归并形式。

因此，关于查询 B，可以得到需要的左操作数（即出现的，或者被 S_DURING 暗指的 S#-DURING 属性对）如下：

```
UNPACK S_DURING { S#, DURING } ON DURING
```

下面是该表达式的扩展形式：

```
WITH S_DURING { S#, DURING } AS T1 ,
   ( T1 GROUP { DURING } AS X ) AS T2 ,
   ( EXTEND T2 ADD EXPAND ( X ) AS Y ) { ALL BUT X } AS T3 :
T3 UNGROUP Y
```

一步步详细地做完这个表达式，这项工作留作练习。而对于给定的图 23-4 的示例数据，完整的结果——称为 U1——如下：

| S# | DURING |
|---|---|
| S1 | [d04:d04] |
| S1 | [d05:d05] |
| S1 | [d06:d06] |
| S1 | [d07:d07] |
| S1 | [d08:d08] |
| S1 | [d09:d09] |
| S1 | [d10:d10] |
| S2 | [d02:d02] |
| S2 | [d03:d03] |
| S2 | [d04:d04] |
| S2 | [d07:d07] |
| S2 | [d08:d08] |
| S2 | [d09:d09] |
| S2 | [d10:d10] |
| S3 | [d03:d03] |
| S3 | [d04:d04] |
| S3 | [d05:d05] |
| S3 | [d06:d06] |
| S3 | [d07:d07] |
| S3 | [d08:d08] |
| S3 | [d09:d09] |
| S3 | [d10:d10] |
| S4 | [d04:d04] |
| S4 | [d05:d05] |
| S4 | [d06:d06] |
| S4 | [d07:d07] |
| S4 | [d08:d08] |
| S4 | [d09:d09] |
| S4 | [d10:d10] |
| S5 | [d02:d02] |
| S5 | [d03:d03] |
| S5 | [d04:d04] |
| S5 | [d05:d05] |
| S5 | [d06:d06] |
| S5 | [d07:d07] |
| S5 | [d08:d08] |
| S5 | [d09:d09] |
| S5 | [d10:d10] |

当然，获得右操作数（即出现的，或者被 SP_DURING 暗指的 S#-DURING 属性对）用的是相似方式：

```
UNPACK SP_DURING { S#, DURING } ON DURING
```

这一表达式的结果——称为 U2——如下：

| S# | DURING |
|---|---|
| S1 | [d04:d04] |
| S1 | [d05:d05] |
| S1 | [d06:d06] |
| S1 | [d07:d07] |
| S1 | [d08:d08] |
| S1 | [d09:d09] |
| S1 | [d10:d10] |
| S2 | [d02:d02] |
| S2 | [d03:d03] |
| S2 | [d04:d04] |
| S2 | [d08:d08] |
| S2 | [d09:d09] |
| S2 | [d10:d10] |
| S3 | [d08:d08] |
| S3 | [d09:d09] |
| S3 | [d10:d10] |
| S4 | [d04:d04] |
| S4 | [d05:d05] |
| S4 | [d06:d06] |
| S4 | [d07:d07] |
| S4 | [d08:d08] |
| S4 | [d09:d09] |
| S4 | [d10:d10] |

现在可以应用求差操作符：

```
U1 MINUS U2
```

这一表达式的结果，设为 U3，如下：

| S# | DURING |
|---|---|
| S2 | [d07:d07] |
| S3 | [d03:d03] |
| S3 | [d04:d04] |
| S3 | [d05:d05] |
| S3 | [d06:d06] |
| S3 | [d07:d07] |
| S5 | [d02:d02] |
| S5 | [d03:d03] |
| S5 | [d04:d04] |
| S5 | [d05:d05] |
| S5 | [d06:d06] |
| S5 | [d07:d07] |
| S5 | [d08:d08] |
| S5 | [d09:d09] |
| S5 | [d10:d10] |

最后，归并 U3 得到想要的完整结果：

```
PACK U3 ON DURING
```

最终结果如下：

| S# | DURING |
|---|---|
| S2 | [d07:d07] |
| S3 | [d03:d07] |
| S5 | [d02:d10] |

下面是查询 *B* 作为单独一个表达式的公式表示：

```
PACK
  ( ( UNPACK S_DURING { S#, DURING } ON DURING )
    MINUS
    ( UNPACK SP_DURING { S#, DURING } ON DURING ) )
ON DURING
```

可以观察到，在 *A* 上反归并 *r*（如同在 *A* 上归并 *r*）包括了将关系 *r* 按照除了 *A* 以外的其他所有属性分组这一过程。

就像作为其基础的 COLLAPSE 和 EXPAND 操作符一样，PACK 和 UNPACK 并不是彼此的逆过程。也就是说，无论 UNPACK（PACK *r* ON *A*）ON *A*，还是 PACK（UNPACK *r* ON *A*）ON *A*，通常都与 *r* 是相等的（尽管它们与 *r* 是等价的，但在某种意义下还是需要解释一下）。下面的恒等式是显而易见的：

- UNPACK *r* ON *A* ≡ UNPACK ( PACK *r* ON *A* ) ON *A*
- PACK *r* ON *A* ≡ PACK ( UNPACK *r* ON *A* ) ON *A*

由此得出结论，在某一给定关系上的操作序列归并 - 反归并或者反归并 - 归并，其中的第一个操作可以被简单忽略掉，这一事实对于优化目标是很有用的（尤其当第一个操作是 UNPACK 的时候）。

**3. 更多例子**

下面给出一些关于 PACK 和 UNPACK 在查询中使用的例子。我们进行合理地假设，每种情况下要求的结果都是归并形式的。

第一个是一个非时态的例子。给定一个关系变量 NHW，属性有 NAME、HEIGHT 和 WEIGHT，表示了一个人的身高和体重。考虑查询“对于 NHW 中的每一个体重值，查询符合

以下条件的身高范围：该范围 r 中至少存在一个人，他的体重等于此体重值。”下面是一种公式表示：

```
PACK
   ( ( EXTEND NHW { HEIGHT, WEIGHT }
       ADD INTERVAL_HEIGHT ( [ HEIGHT : HEIGHT ] ) AS HR )
     { WEIGHT, HR } )
ON HR
```

说明：首先将 NHW 映射到 HEIGHT 和 WEIGHT 上，从而获得原关系上的所有身高 - 体重对（即至少有一个人符合这样的身高体重的身高 - 体重对）。接下来通过引入另一个属性 HR 来扩展映射，任意元组的 HR 值是一个形如 [h: h] 的单元区间，其中 h 是该元组的 HEIGHT 值（注意区间选择符 INTERVAL_HEIGHT）。接着映射去掉 HEIGHT 属性并将结果归并到 HR 上。最终的结果是一个有两个属性，WEIGHT 和 HR 的关系，谓词语义如下：

对 HR 中的所有身高 *h*——但不是 *h* = PRE（HR）或者 *h* = POST（HR）——至少存在一个人 *p*，满足 *p* 的体重是 WEIGHT，身高是 *h*。

第二个例子，再次考虑关系变量 SP_DURING。假设在任意时间，如果有发货记录，那么一定存在某一零件号 pmax 满足以下要求：在这一时间，任何供应商都无法提供零件号大于 pmax 的零件。（显然这里假设运算符“ > ”对类型为 P#的数值是有意义的。）所以，考虑查询“对于每一个曾经成为 pmax 值的零件号，查询所有这样的零件号以及它作为 pmax 值的那个区间。”下面是一个可能的公式表示：

```
WITH ( UNPACK SP_DURING ON DURING ) AS SP_UNPACKED ,
     ( SUMMARIZE SP_UNPACKED
       BY { DURING }
       ADD MAX ( P# ) AS PMAX ) AS SUMMARY :
PACK SUMMARY ON DURING
```

#### 4. 备注

关于归并和反归并有太多的内容，远非本章的篇幅所能容纳。详细的讨论参见文献[23.4]；这里我们只列出最重要的几点，而不加以证明或做更深层次的解释。

- 不在任何一个属性上做归并或反归并一个关系 *r* 的操作，简单返回 *r*。
- 在两个或两个以上区间类型的属性[⊖] 上反归并一个关系 *r* 是简单直接的；如果这些属性是 *A1*，*A2*，…，*An*（按照某个顺序），那么可以通过以下方式得到结果：在 *A1* 上反归并 *r*，接着在 *A2* 上反归并第一次反归并得到的结果，…，最后在 *An* 上反归并倒数第二次反归并得到的结果。
- 在两个或两个以上区间类型的属性上归并一个关系 *r* 并不是简单直接的。然而，一般的，如果这些属性是 *A1*，*A2*，…，*An*（按照某个顺序），那么可以通过以下方式得到结果：首先在所有这些属性上反归并 *r*，接着在 *A1* 上归并反归并的结果，在 *A2* 上归并第一次归并的结果，…，最后在 *An* 上归并倒数第二次归并得到的结果。
- 设 *r1* 和 *r2* 是同类型的关系，它们的属性 *A1*，*A2*，…，*An* 是区间类型的值。则 *r1* 和 *r2* 是等价的（关于属性 *A1*，*A2*，…，*An*），当且仅当 UNPACK *r1* ON（*A1*，*A2*，…，*An*）的结果与 UNPACK *r2* ON（*A1*，*A2*，…，*An*）的结果相等。

## 23.5　关系操作符推广

23.4 节提到过查询 *B* 的公式表示：

```
PACK
  ( ( UNPACK S_DURING { S#, DURING } ON DURING )
    MINUS
    ( UNPACK SP_DURING { S#, DURING } ON DURING ) )
ON DURING
```

---

⊖ 注意这里暗示了一个关系有两个或两个以上区间类型的属性是完全可以的。

像这样的表达式，包括两个反归并操作，跟着是一个规则的关系操作，然后是一个重归并操作，这样的表达式在实际中是要经常用到的，因此为它们定义一个速记符号也是很值得的（是进一步的速记表示，就我们所知，它们基本上已经算是速记符号化了）。当然这样的速记符号化会省略很多的笔墨。另外它也提供了改进性能的机会：当包含了细粒度的长区间时，一个反归并操作的输出与它的输入相比可能会非常大；如果系统要物化这样一个反归并操作的结果，那么查询可能会发生死循环或者内存溢出。相比而言，将整个需求表示成一个单独的操作可以允许优化器选择一个更有效的执行方案，不再要求物化反归并操作的中间结果。

根据前面的想法，我们定义表达式

```
USING ( ACL ) ◄ r1 MINUS r2 ►
```

作为以下表达式的速记：

```
PACK
   ( ( UNPACK r1 ON ( ACL ) ) MINUS ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

这里 *r1* 和 *r2* 是相同类型关系的关系表达式，*ACL* 是属性名的列表，其中每一个属性（1）都是某种区间类型的，（2）都出现在两个关系中。要点：

1）除非另行声明，我们提到的操作符指的是“U_difference”（U 表示 USING），或者简写为 U_MINUS。

2）如果属性名列表只包含了一个属性名，则在 USING 说明中用来括起属性名的括号可以省略掉。注意：这一说明适用于将要定义的所有“U_”的速记，后面将不会每次都重申这点。

3）本节中 USING 说明出现的上下文中，使用◄和►来把表达式和 USING 说明应用划分开。

4）与规则的 MINUS 操作符不同，U_MINUS 产生的结果集的基数可能会大于它的左操作数。比如，设 *r1* 和 *r2* 如下：

*r1*

| A |
|---|
| [d02:d04] |

*r2*

| A |
|---|
| [d03:d03] |

则 USING A◄r1 MINUS r2►得到的结果是：

| A |
|---|
| [d02:d02]<br>[d04:d04] |

说了这么多关于 U_MINUS，现在可以很清楚地知道，所有规则的关系操作符都可以定义它们的“U_”版本（参考文献［23.4］也确实是这样做的）。然而由于篇幅的限制，这里只限于那些最有用的操作符，（除了 U_MINUS 之外）U_UNION，U_INTERSECT，U_JOIN 以及 U_project。U_UNION 和 U_INTERSECT 遵循了与 U_MINUS 相同的通用模式；即表达式

```
USING ( ACL ) ◄ r1 op r2 ►
```

（其中 *op* 是 UNION 或者 INTERSECT，*ACL*、*r1* 和 *r2* 的含义与 U_MINUS 中的相同）是以下表达式的速记：

```
PACK
   ( ( UNPACK r1 ON ( ACL ) ) op ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

我们注意到，在 U_UNION 的情况下，没有必要执行开始的 UNPACK 操作。即 U_UNION 的展开形式可以进一步缩写为：

```
PACK ( r1 UNION r2 ) ON ( ACL )
```

不难看出为什么这一简化是可能的，但如果需要证明它是合法有效的，还可以试着通过一个例子来证明。（事实上，对于其他一些“U_”操作符，相似的简化同样是可能的，但细节已经超出了本章的范围。）

另外，就像 U_MINUS 可能增加结果集的基数一样（不严格地说），U_UNION 可能降低它；实际上，U_UNION 产生的结果集的基数可能比它的两个操作数都要小（证明留给读者作为习题）。同样，U_INTERSECT 产生的结果集的基数可能比它的两个操作数都要大（作为习题证明）。

现在我们来看一下 U_JOIN，定义表达式

```
USING ( ACL ) ◄ r1 JOIN r2 ►
```

作为以下表达式的速记：

```
PACK
   ( ( UNPACK r1 ON ( ACL ) ) JOIN ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

*ACL* 中的每个属性必须是某种区间类型，而且必须同时出现在 *r1* 和 *r2* 中（所以连接是在 *ACL* 中涉及的所有属性上进行的）。注意：如果 *r1* 和 *r2* 是相同类型的，则 U_JOIN 退化为 U_INTERSECT。

下面用一个例子来说明 U_JOIN 的使用。假设数据库中存在另一个关系变量 S_CITY_DURING，它的属性包括 S#、CITY 和 DURING，候选码是 {S#, DURING}，谓词语义如下：

供应商 S#从 DURING 的开始点到结束点这一区间内位于城市 CITY。

现在考虑查询“取得满足以下条件的 S#_CITY_P#_DURING 元组：供应商 S#位于城市 CITY，且在整个区间 DURING 内可以供应零件 P#，其中 DURING 包含了日期 4。”下面是该查询的一个可能的公式表达形式：

```
( USING DURING ◄ S_CITY_DURING JOIN SP_DURING ► )
                                        WHERE d04 ∈ DURING
```

最后看一下 U_project，定义表达式

```
USING ( ACL ) ◄ R { BCL } ►
```

作为以下表达式的速记：

```
PACK ( ( UNPACK r ON ( ACL ) ) { BCL } ) ON ( ACL )
```

*ACL* 中的每个属性都必须是某种区间类型的，且必须在 *BCL* 中涉及（因此也必然是 *r* 的一个属性）。我们还是将查询 A 作为例子来看一下：

- 查询 A：取得至少在某一个时间区间里可以供应至少一种零件的供应商的属性对 S#_DURING，其中 DURING 是指这样一个区间。

下面是该查询的一个“U_project”的公式表达形式：

```
USING DURING ◄ SP_DURING { S#, DURING } ►
```

鉴于对于查询 B 我们已经有了如下的一个“U_MINUS”的公式表达形式：

```
USING DURING ◄ S_DURING { S#, DURING }
               MINUS
               SP_DURING { S#, DURING } ►
```

现在我们已经达到了一个最初目标：已经找到了一种（实际上是更多）用公式表示查询 A 和查询 B 的更好的方式。

**1. 关系比较**

关系比较从严格意义上来说不算是关系操作，因为它们返回的是一个真值而非一个关系。然而，我们可以将它们与关系操作符同等对待，而且它们也确实值得这样做。当关系包含了区间类

型的属性时，经常要做的是比较这些关系的某些反归并的副本，而不是关系本身。我们先来介绍规则的“关系等价”比较的“U_”对应操作。具体地说，定义表达式

```
USING ( ACL ) ◄ r1 = r2 ►
```

作为以下表达式的速记：

```
( UNPACK r1 ON ( ACL ) ) = ( UNPACK r2 ON ( ACL ) )
```

*ACL* 中的每个属性必须是某种区间类型的，而且必须同时出现在 *r1* 和 *r2* 中。需要注意的是，问题中最后一个步骤 PACK 没有出现，因为正如已经指出的那样，“ = ”的结果是一个真值而非一个关系。

作为例子，设 *r1* 和 *r2* 如下：

*r1*

| A |
|---|
| [*d01*:*d03*] |
| [*d02*:*d05*] |
| [*d04*:*d04*] |

*r2*

| A |
|---|
| [*d01*:*d02*] |
| [*d03*:*d05*] |

则 *r1* = *r2* 的结果为 FALSE，但 USING A◄r1 = r2►的结果为 TRUE。

为了方便，我们将上述的操作符简写为“U_=”（实际上，它与前一节结尾处为 *n* 元关系定义的操作符是等价操作符）。用同种方式可以定义其他所有关系比较操作符（≠、⊂、⊆、⊃和⊇的“U_”版本。例如，如果 *r1* 和 *r2* 如上例中所示，则 USING A◄*r1* ⊆*r2*►的结果为 TRUE，而 USING A◄*r1* ⊂*r2*►的结果为 FALSE。

**2. 规则关系操作回顾**

再次考虑操作符 U_MINUS。回忆一下，我们定义了表达式

```
USING ( ACL ) ◄ r1 MINUS r2 ►
```

作为以下表达式的速记：

```
PACK
   ( ( UNPACK r1 ON ( ACL ) ) MINUS ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

现在假设 *ACL* 为空（即没有指定任何属性），因此：

```
USING ( ) ◄ r1 MINUS r2 ►
```

则扩展形式变为：

```
PACK
   ( ( UNPACK r1 ON ( ) ) MINUS ( UNPACK r2 ON ( ) ) )
ON ( )
```

现在回想上一节中 UNPACK *r* ON( )和 PACK *r* ON( )返回的都是 *r*。因此整个表达式化简为：

```
r1 MINUS r2
```

换句话说，规则的关系 MINUS 本质上就是 U_MINUS 的一种特例！因此如果我们重新定义规则 MINUS 操作符的语法如下：

```
[ USING ( ACL ) ] ◄ <relation exp> MINUS <relation exp> ►
```

并且允许当且仅当 *ACL* 为空时，USING 的说明（以及括起了表达式其余部分的箭头◄和►）可以省略，那么就不再需要特别讨论“U_MINUS”操作符——所有的 MINUS 有效地变为 U_MINUS，而且可以因此扩展 MINUS 的含义。

类似的论点可以应用于其他所有关系操作符，包括关系比较符：在所有情况下，规则操作符基本上只是对应的“U_”操作符在 USING 说明中不存在任何属性时的特例，在此种情况下，允许说明（以及括起了表达式其余部分的箭头◀和▶）省略。从另一个角度说，“U_”操作符都只是对应的规则操作符的概化形式。因此不再需要明确讨论“U_”操作符（除非偶尔为了强调，我们也不会再讨论）；取而代之我们需要做的就是，要承认当关系操作符应用于区间类型属性时，操作符允许但并不一定要求存在额外的一个操作数。*因此请注意，在本章中的剩余部分，所有的关系操作符和关系比较符，指的都是本节所说的通用形式*（除非明确说明）。然而为了清楚起见，我们时常会根据应用明确地限定“规则的”（或者称为“经典的”）和“通用的”操作符和比较符；同样根据应用，我们有时也会明确地使用“U_”。

## 23.6　数据库设计

时态数据库设计中存在一些特殊的问题。为了举例说明这些问题，我们按照如下的方式再次修改前面的例子：第一，完全删掉发货记录；第二，恢复供应商的姓名、状态和城市信息。下面直接提出修改数据库的首选设计[⊖]：

```
VAR S_SINCE BASE RELATION
    { S#      S#,       S#_SINCE      DATE,
      SNAME   NAME,     SNAME_SINCE   DATE,
      STATUS  INTEGER,  STATUS_SINCE  DATE,
      CITY    CHAR,     CITY_SINCE    DATE }
    KEY { S# } ;
VAR S_DURING
    BASE RELATION
    { S#      S#,
      DURING INTERVAL_DATE }
    KEY { S#, DURING } ;
VAR S_STATUS_DURING
    BASE RELATION
    { S#      S#,
      STATUS INTEGER,
      DURING INTERVAL_DATE }
    KEY { S#, DURING } ;
VAR S_NAME_DURING
    BASE RELATION
    { S#      S#,
      SNAME   NAME,
      DURING INTERVAL_DATE }
    KEY { S#, DURING } ;
VAR S_CITY_DURING
    BASE RELATION
    { S#      S#,
      CITY    CHAR,
      DURING INTERVAL_DATE }
    KEY { S#, DURING } ;
```

谓词语义如下：

- S_SINCE：*供应商 S#自从 S#_SINCE 开始就签订了合约，自从 SNAME_SINCE 开始就被称为 SNAME，自从 STATUS_SINCE 开始状态就是 STATUS，自从 CITY_SINCE 开始就位于城市 CITY。*
- S_DURING：*供应商 S#在区间 DURING 期间是处于合约约束下的。*
- S_NAME_DURING：*供应商 S#在区间 DURING 期间的名字是 SNAME。*
- S_STATUS_DURING：*供应商 S#在区间 DURING 期间的状态是 STATUS。*
- S_CITY_DURING：*供应商 S#在区间 DURING 期间位于城市 CITY。*

另外，我们需要增加一种情况，即四个“在……期间”说明了 DURING 结束点的那一天是过去的一天（参加本节后面的小节“移动点 NOW”）。

正如你看到的，我们首选的设计方案在一个“自从关系变量”里保留了当前信息，在一组

⊖ 要特别注意的是，在这个设计中关系变量 S_SINCE 与 23.2 节中的关系变量 S_SINCE 不同。

“在……期间关系变量”里保留了历史信息。我们将这种分离称为**水平分解**。此外，历史信息保留在一组不同的关系变量里面——泛泛地说，每个都是供应商的一个不同属性——而我们称这种分离为**垂直分解**。在接下来的几小节中我们会证明这些分解。

**1. 水平分解**

水平分解的主要理由很简单，是由于历史信息和当前信息之间存在明显的逻辑差异：

- 对于历史信息，开始和结束时间都是已知的。
- 相对而言，对于当前信息，开始时间是已知的，而结束时间未知。

换句话说，谓词语义是不同的，很明显将历史信息和当前信息划分在不同的关系变量中是正确的选择。*注意*：实际上前面的两个声明都有点过于简单，但对于当前的目标来说已经是足够的了。

然而可以观察到，“当前的”关系变量 S_SINCE 有四个“自从”属性，每个对应了一个“非自从”的属性。比较而言，有关“时间戳元组”的文献中提出单独一个“自从”属性应该就足够了，因此：

```
S_SINCE { S#, SNAME, STATUS, CITY, SINCE }
```

但是如果试着表述这个设计的谓词，很容易看出它的错误：

*自从日期 SINCE 开始，以下的四条都为真：*

1）*供应商 S#处在合约约束下。*

2）*供应商 S#被称为* SNAME。

3）*供应商 S#的状态是* STATUS。

4）*供应商 S#位于城市* CITY *中。*

例如，假设当前的关系变量中包含了如下的元组：

| S# | SNAME | STATUS | CITY | SINCE |
|---|---|---|---|---|
| S1 | Smith | 20 | London | *d04* |

假设今天是日期 10，且从今天开始，供应商 S1 的状态改为 30，因此将刚才的元组替换为：

| S# | SNAME | STATUS | CITY | SINCE |
|---|---|---|---|---|
| S1 | Smith | 30 | London | *d10* |

现在我们已经失去了供应商 S1 自从日期 4 开始就位于伦敦中这一信息。概括起来，很显然这个设计无法表示出一个供应商在最近一次修改之前的任何信息（比较宽泛的说法）。笼统地说，问题就在于时间戳属性 SINCE 时间戳的性质太强；它代表了四个不同属性联合（供应商在合约约束下，供应商的名字、状态以及所在的城市）而非单独一个属性的时间戳。相反，在我们的首选设计方案中，每个属性都应有自己的时间戳。

**2. 垂直分解**

当然，即使有四个分离的“自从”属性，关系变量 S_SINCE 也只是半时态化的，这就是为什么我们还需要“在……期间”关系变量来表示历史信息。但为什么对历史信息来说垂直分解是必要的呢？为了考察这个问题，假设已经存在一个“在……期间”关系变量如下：

```
S_DURING { S#, SNAME, STATUS, CITY, DURING }
```

谓词语义如下：

*在区间* DURING *期间里，以下四条都是真的：*

1）*供应商 S#处在合约约束下。*

2）*供应商 S#被称为* SNAME。

3）*供应商 S#的状态是* STATUS。

4）*供应商 S#位于城市* CITY *中。*

如同前一小节中的那个只有一个“自从”属性的关系变量 S_SINCE 一样，从上面的谓词语

义中应该可以马上并且清楚地看出这个关系变量设计得并不好。假设它包含了如下的元组：

| S# | SNAME | STATUS | CITY | DURING |
|---|---|---|---|---|
| S2 | Jones | 10 | Paris | [*d02*:*d04*] |

同时假设我们现在得知（1）供应商 S2 在日期 2 和日期 3 时的状态确实是 10，但在日期 4 时变为了 15，并且（2）供应商 S2 在日期 3 和日期 4 时确实在巴黎，但在日期 2 时应该在伦敦。则我们需要对关系变量做一组相当复杂的更新来反映真实世界的变化。具体地说，需要将已存在的元组替换为以下三个元组：

| S# | SNAME | STATUS | CITY | DURING |
|---|---|---|---|---|
| S2 | Jones | 10 | London | [*d02*:*d02*] |

| S# | SNAME | STATUS | CITY | DURING |
|---|---|---|---|---|
| S2 | Jones | 10 | Paris | [*d03*:*d03*] |

| S# | SNAME | STATUS | CITY | DURING |
|---|---|---|---|---|
| S2 | Jones | 15 | Paris | [*d04*:*d04*] |

现在可以观察到，我们用两个元组替代一个元组来表示在区间［*d02*：*d03*］期间状态是 10，用两个元组替代一个元组来表示在区间［*d03*：*d04*］期间所在的城市是巴黎。

正如这个例子暗示的那样，更新关系变量 S_DURING 使之能够反映真实世界的变化，这项工作通常来说并不是完全简单直接的。主要的问题还是时间戳属性（现在是 DURING）时间戳的性质太强，它实际上还是四个不同属性联合的一个时间戳。解决方法就是将四个属性分离开，分成四个单独的关系变量，因此：

```
S_DURING          { S#, DURING }
                  KEY { S#, DURING }

S_NAME_DURING     { S#, SNAME, DURING }
                  KEY { S#, DURING }

S_STATUS_DURING   { S#, STATUS, DURING }
                  KEY { S#, DURING }

S_CITY_DURING     { S#, CITY, DURING }
                  KEY { S#, DURING }
```

关系变量 S_DURING 表明了某供应商在何时处于合约约束下；关系变量 S_NAME_DURING 表明某供应商在何时叫什么名字；关系变量 S_ STATUS _DURING 表明某供应商在何时的状态如何；关系变量 S_ CITY _DURING 表明某供应商在何时位于哪个城市。

**3. 第六范式**

前面的垂直分解，无论在基本理论上还是在效果上，都使人联想到经典标准化，而且值得花费一定时间在更深的层次上考察它们的相似之处。实际上，垂直分解一直是经典标准理论关心的内容；这一理论中的分解操作符是投影（它被定义为一个垂直分解操作符），相应的重组操作符是连接。基于以上原因，如我们在第 13 章看到的那样，经典标准化理论的最终范式，第五范式或者 5NF，有时被称为投影—连接范式。注意：既然这些评论与经典标准化息息相关，那么提到投影和连接就必须理解成是这些操作符的经典版本，而不是 23.5 节介绍的通用版本。

甚至是在出现时态数据库之前，一些研究人员（见参考文献［14.21］）就赞同尽可能地分解关系变量，而不只是按照经典标准化要求的那样做。通常的想法是要减少关系变量**不可约的成分**[14.21]，意味着不可能存在进一步的无损分解。在非时态关系变量的情况下，要求“自始至终都要分解”的观点并不十分强烈，但是对形如上一个小节中的 S_DURING（只有一个

“在……期间”属性的第一个版本）这样的关系变量，这种观点就变得十分强烈了。一个供应商的名字、状态和城市的改变是不依赖于时间的，而且它们也可能按照不同的速度变化。比如可能的情况是，供应商的名字几乎不变，而同一个供应商的位置偶尔会发生变化，对应的状态经常改变。除此之外，供应商的名字历史、状态历史和城市历史这些信息，比起“名字 - 状态 - 城市”这一联合的历史信息，可能是更容易得到的概念；因此我们建议进行垂直分解。

现在回想起来，5NF 是建立在连接依赖（JD）基础上的。*提示*：关系变量 *R* 满足 JD * {*A*, *B*, …, *Z*}（其中 *A*, *B*, …, *Z* 是 *R* 的属性子集），当且仅当 *R* 的每个合法值都等于它在 *A*, *B*, …, *Z* 上的投影的连接——即当且仅当 *R* 可以被无损分解为那些投影。现在，既然我们已经将连接的定义泛化了，那么相应地我们可以将 JD 的定义也一般化。接着我们可以根据 JD 的一般化概念，定义一个新的（即第六）范式。其定义如下：

- 设 *R* 是一个关系变量，*A*, *B*, …, *Z* 是 *R* 的属性子集，设 *ACL* 是 *R* 的区间值属性的列表。则我们说 *R* 满足一般化的**连接依赖**（JD）：

```
USING ( ACL ) * { A, B, ..., Z }
```

当且仅当表达式

```
USING ( ACL ) ◄ R = R' ►
```

（其中 *R'* 是 *R* 在 *A*, *B*, …, *Z* 上的 U_投影的 U_连接，U_连接和 U_投影都包含了一个 USING (*ACL*) 形式的 USING 说明）对于 *R* 的每一个合法值都为真。*注意*：与连接一样，U_连接是结合性的，也就是说我们可以明确地说到任何多个关系的 U_连接。还需要注意的是，说前面的表达式为真，意味着 *R* 和 *R'* 是（关于 *ACL*）等价的——参见 23.4 节结尾处关于这一内容的讨论。

- 一个关系变量 *R* 是**第六范式**（6NF），当且仅当它完全不满足非平凡连接依赖。其中，一个连接依赖是**平凡**的，当且仅当它至少包含了一个投影（可能是 U_投影），该投影是在这个关系变量涉及的所有属性上的。

从这个定义中可以发现，每个满足 6NF 的关系变量也都满足 5NF。同时，给定一个关系变量，它满足 6NF 当且仅当它在某一方面是不可约的，如同前面解释的那样。

现在根据这个定义，有属性 S#、SNAME、STATUS、CITY 和 DURING 的关系变量 S_DURING 版本不满足 6NF，因为：

1）它满足一般化的连接依赖 USING DURING * {SND, STD, SCD}（其中“SND”代表了属性集 {S#, SNAME, DURING}，“STD”和“SCD”与之类似）。

2）这个连接依赖肯定是非平凡的。

因此我们希望关系变量 S_DURING 像上一小节讨论的那样，被分解成满足 6NF 的投影。

*注意*：你可能已经注意到，前面的例子分解成 3 个而非 4 个关系变量就已经足够了——有属性 S#和 DURING 的关系变量 S_DURING 不需要分解，因为在任何时候 S_DURING 都等于其他三个关系变量在 S#和 DURING 上的（一般）投影。然而我们还是倾向于将 S_DURING 包括进我们的总体设计，部分是因为完整性的原因，另一部分原因是这样的设计避免了可能出现的某种程度上的不易使用和任意性[23.4]。

**4. 移动点 NOW**

现在回到水平分解这个命题上来（即分割成“自从……”和“在……期间”关系变量）。显然不能只有“自从……”关系变量，因为这样的关系变量只不过是半时态化的，而且不能表示历史信息。但是可以只有“在……期间”关系变量——只有假定数据库中的数据都是真实有效的，正如我们现在要解释的这样。

考虑一种情况，一个合约还没有终止的供应商。当然，我们可能知道合约预计何时终止；但多数情况是合约是可修订的（比如一份典型的雇佣合同）。因此，在一个“在……期间”关系变量中无论指定什么（值）作为这样一个供应商的 DURING 属性的 END 值好像都是不正确的。我们可以、也可能会采用一种惯例：每一个供应商 DURING 属性的 END 值都被指定为最后一天

(即 END 值由 LAST_DATE( )返回)[⊖]。但需要注意的是，这样的安排意味着，如果“最后一天”出现在一个查询的结果中，用户可能要将该值解释为“直到另行通知”，而不是实际上的最后一天，换句话说，说一个供应商 DURING 属性的 END 值是“最后一天”就不够真实。

为了避免产生歧义，一些著作——比如参考文献［23.2］——已经提出使用一个特别的“NOW 标记”来指明在 23.1 节中所说的“移动点（时间变元）NOW”（换句话说就是代替“直到另行通知”（变动的时间或当前时间或不确定的时间））。基本的想法就是允许这个特别标记无论在哪里出现都是：（1）允许点类型的值；（2）希望的解释是“直到另行通知”。因此，比如一个关系变量 S_DURING 可能包括了供应商 S1 的一个元组，DURING 值由［*d04*：NOW］代替了［*d04*：*d99*］。（这里假设日期 99 是最后一天，且 *d99* 的出现代表的是“直到另行通知”（当前时间）而不是真正的日期 99。）

但是我们认为，引入 NOW 标记违反了健全的关系原则。要注意的是，NOW 是一个变量，我们观察到，这个方法涉及非常奇特的——可以说是不合逻辑的——变量值（特指区间值）的概念[⊖]。下面有一些由 NOW 这个概念引起的问题，你可能要注意考虑一下：

- 设 *i* 是区间［NOW：*d14*］，*t* 是包含 *i* 的一个元组，并且设今天的日期为 10。元组 *t* 可被认为是分别包含了单位区间［*d10*：*d10*］、［*d11*：*d11*］、［*d12*：*d12*］、［*d13*：*d13*］和［*d14*：*d14*］的五个单独元组的一种速记。但当到了日期 10 的午夜时，这些元组中的第一个会自动删除！日期 11、日期 12 以及日期 13 等都是同样的情况，那么在日期 14 时会出现什么情况呢？
- *d99* = NOW 这一比较会产生什么结果？
- “NOW + 1”或者“NOW - 1”的值是多少？
- 如果 *i1* 和 *i2* 分别是区间［*d01*：NOW］和［*d06*：*d07*］，它们是否会相邻或重叠？
- 一个元组，它的区间属性值为［*d04*：NOW］，在这个属性上反归并包含这个元组的一个关系，得到的结果是怎样的？
- 集合｛［*d01*：NOW］，［*d01*：*d04*］｝的基数是多少？

等等（这里并不是一个最详尽的列表）。对于上面这些问题，相信很难给出一致的答案；显然，我们希望有一种方法可以不依赖于像 NOW 这样不可信的概念以及包含了变量的数值。的确存在这样的观点，即在水平分解中时间元素 NOW 不是必要的。

## 23.7　完整性约束

本节将注意力转向时态数据的完整性约束。在 23.2 节中曾经看到，缺少了适当的区间的支持，即使是用公式准确表示出这种约束都是非常困难的；现在我们将会看到上一节中介绍的概念是怎样减弱这个问题的（即如何减少表示约束所面临的困难）。

为了明确起见，我们集中关注关系变量 S_STATUS_DURING，定义如下：

```
S_STATUS_DURING { S#, STATUS, DURING }
                KEY { S#, DURING }
```

在接下来的三个小节中，我们会逐个检查可能发生在这样一个时态关系变量上的三个常见问题，分别称它们为冗余问题、迂回问题以及矛盾问题。

### 1. 冗余问题

尽管关系变量 S_STATUS_DURING 的主码约束在逻辑上是正确的，但在某种意义上来说是不充分的。比如，它不能防止关系变量同时包含以下两个元组：

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d05*:*d06*] |

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d06*:*d07*] |

⊖　当然，在后面真值已知后，我们可以将那个假值替换为真值。

⊖　其实 NOW 类似于 NULL，因为它就如同 NULL 一样，导致值的概念中包含进了非数值的情况（参见第 19 章）。

可以看到，这两个元组产生了某种**冗余**，因为按照此种情况，供应商 S4 在日期 6 被描述了两次。很显然，由下面一个元组替代它们是更好的选择：

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d05*:*d07*] |

现在可以观察到，如果原来的两个元组是某个两元组关系中仅有的元组，并且我们在 DURING 属性上归并了这个关系，我们将得到一个只包含了如上所示的一个元组的一元组关系。因此不严格地说，刚才所示的元组是一个“归并的”元组，它是通过在属性 DURING 上归并两个原始元组得到的（之所以用“不严格地说”是因为，事实上反归并操作是应用在关系上而非元组上的）。所以我们要做的就是将两个原始元组替换成那个“归并的”元组。实际上正如在 23.2 节中指出的那样，不执行这一替换——即允许两个原始元组同时出现——几乎与允许重复元组出现（这同样是一种冗余）一样糟糕。确实，如果两个原始元组都出现，关系变量就违反了自己的谓词！例如，右边的元组说明供应商 S4 在日期 6 的前一天的状态还不是 25。但左边的元组说明供应商 S4 在日期 5 的状态是 25，而显然日期 5 是日期 6 的前一天。

**2. 迂回问题**

关系变量 S_STATUS_DURING 的主码约束在另一个方面也是不充分的。比如，它不能防止关系变量同时包含以下两个元组：

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d05*:*d05*] |

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d06*:*d07*] |

这里不存在冗余，但是存在某种**迂回**（circumlocution），因为我们用了两个元组来说明一个只用一个“归并的”元组（实际上与前面的相同）就能更好说明的问题：

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d05*:*d07*] |

很容易证明，不将两个原始元组替换成这个“归并的”元组会再次表示关系变量违反了自己的谓词。

**3. 解决冗余和迂回问题**

为了避免刚刚讨论过的冗余和迂回问题，需要做的就是执行一条约束，称为**约束 A**，叙述如下：

> 约束 A：一个任意给定的时态关系变量 S_STATUS_DURING，如果它包含了两个不同的元组，它们除了 DURING 值分别为 *i*1 和 *i*2 以外，其他都相同，那么 *i*1 MERGES *i*2 必须为假。

我们曾讲过，不严格地说，MERGES 是 OVERLAPS 和 MEETS 的逻辑 OR：在约束 A 中，将 MERGES 替换成 OVERLAPS 得到的约束可以用来避免冗余问题；将它替换成 MEETS 得到的约束可以用来避免迂回问题。

显然要执行约束 A 有一种非常简单的方法：任何时候归并关系变量都要在 DURING 这个属性上。因此让我们生成一条新的 PACKED ON 约束，它可以出现在关系变量的定义中：

```
VAR S_STATUS_DURING BASE RELATION
  { S# S#, STATUS INTEGER, DURING INTERVAL_DATE }
  PACKED ON DURING
  KEY { S#, DURING } ;
```

这里的 PACKED ON DURING 是一个在关系变量 S_STATUS_DURING 上的约束。按照第 9 章描述的经典模式，它实际上是一个关系变量约束。它的解释如下：关系变量 S_STATUS_DURING 在任何时候都要保证它的归并是在 DURING 上的。因此这一特殊的语法能够解决冗余和迂

回问题；换句话说，23.2 节中提到的约束 XFT1 可以作为它解决问题的实例。

**4. 矛盾问题**

PACKED ON 约束和主码约束即使加在一起仍然不是完全充分的。比如，它们不能防止关系变量同时包含以下两个元组：

| S# | STATUS | DURING |
|---|---|---|
| S4 | 10 | [*d04*:*d06*] |

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d05*:*d07*] |

这里供应商 S4 在日期 5 和日期 6 的状态同时为 10 和 25——显然是一种不可能的情形。换句话说，这里存在一个**矛盾**；事实上，这个关系变量再次违反了自己的谓词，因为我们已经设定每个供应商在任何一个给定的日期里只能有一个状态。

**5. 解决矛盾问题**

为了避免刚刚讨论的矛盾问题，需要做的就是执行一条约束，称为约束 B，叙述如下：

约束 B，一个任意给定的时态关系变量 S_STATUS_DURING，如果它包含了两个具有相同 S#值和不同 STATUS 值的元组，那么它们的 DURING 值 *i1* 和 *i2* 必须满足 *i1* OVERLAPS *i2* 为假。

注意，正如我们已经看到的那样，约束 B 明显不能仅仅靠在 DURING 上归并关系变量这个保证来执行，它也不能仅仅靠 {S#, DURING} 是候选码这一事实来执行。但假设关系变量保持在任何时间的反归并都是在 DURING 属性上（暂时忽略这样一个事实，即这个假设是不能成立的，因为我们已经规定了关系变量的归并要保证在 DURING 上）。那么：

- 反归并形式中所有的 DURING 值都是单位区间，并且因此与单个的时间点有效对应。
- 反归并形式的唯一候选码还将是 {S#, DURING}，因为任意给定的在合同约束下的供应商，在任意给定的时间都只有一个状态。

由此得出结论，如果执行“{S#, DURING} 是反归并形式 UNPACK S_STATUS_DURING ON DURING”的一个候选码这个约束，那么我们会强制执行约束 B。因此，让我们生成一个可以出现在关系变量定义中的新的 WHEN/THEN 约束，只要主码约束出现，它就可以出现：

```
VAR S_STATUS_DURING BASE RELATION
  { S# S#, STATUS INTEGER, DURING INTERVAL_DATE }
  PACKED ON DURING
  WHEN UNPACKED ON DURING THEN KEY { S#, DURING }
  KEY { S#, DURING } ;
```

这里的 WHEN UNPACKED ON DURING THEN KEY {S#, DURING} 是一个在关系变量 S_STATUS_DURING 上的约束——就像前面讨论的 PACKED ON 约束一样，它也是一个关系变量约束。它的解释如下：任何时候对于关系变量 S_STATUS_DURING，表达式 UNPACK S_STATUS_DURING ON DURING 的结果中都不能存在其属性组合 {S#, DURING}（不严格的说法是“{S#, DURING} 是 UNPACK S_STATUS_DURING ON DURING 的一个候选码”）值相同的两个元组。因此这一特殊的语法能够解决矛盾问题。

**6. U_key**

关于 KEY、PACKED ON 和 WHEN/THEN 约束，还有很多可以讨论的 [23.4]；而由于篇幅的原因，只作如下说明。首先，任何一个给定的关系变量 *R*，它的定义中都允许包含如下形式的速记说明：

```
USING ( ACL ) KEY { K }
```

这里 *ACL* 和 *K* 都是属性名的列表，其中 *ACL* 中的每个属性都必须同时在 *K* 中（并且通常情况下，如果 *ACL* 只包含一个属性名，则圆括号可以省略）。这个说明被定义为如下三个约束组合的速记：

```
PACKED ON ( ACL )
WHEN UNPACKED ON ( ACL ) THEN KEY { K }
KEY { K }
```

我们将 ｛*K*｝ 缩写为“U_key”（后面的内容就会见到）。若使用这个速记，那么关系变量 S_STATUS_DURING 的定义可简化为：

```
VAR S_STATUS_DURING BASE RELATION
  { S# S#, STATUS INTEGER, DURING INTERVAL_DATE }
  USING DURING KEY { S#, DURING } ;
```

现在假设关系变量 *R* 的 U_key 说明中，*ACL* 的属性名列表为空，因此：

```
USING ( ) KEY { K }
```

根据定义，这个说明是下列约束组合的速记：

```
PACKED ON ( )
WHEN UNPACKED ON ( ) THEN KEY { K }
KEY { K }
```

换句话说，就是：

1）关系变量 *R* 必须不在任何属性上归并。而不在任何属性上归并一个关系变量 *r*，将简单地返回 *r*，因此隐式的 PACKED ON 说明没有作用。

2）关系变量 *R* 必须满足：如果它不在任何属性上进行反归并，那么 ｛*K*｝ 是结果的一个候选码。而不在任何属性上反归并一个关系变量 *r*，将简单地返回 *r*，因而隐式的 WHEN/THEN 说明只是意味着 ｛*K*｝ 是 *R* 的一个候选码，因此隐式的 KEY 约束是冗余的。

由此得出结论，可以将一个规则的、形如 KEY ｛*K*｝ 的 KEY 约束，作为某个 U_key 约束的速记，也就是其中的一种形式 USING （） KEY ｛*K*｝。换句话说，规则的 KEY 约束本质上只是我们提出的新语法的一种特殊情况！所以，如果重新定义规则的 KEY 约束的语法为：

```
[ USING ( ACL ) ] KEY { K }
```

并且当且仅当 *ACL* 为空时，允许将 USING 的说明省略掉，那么我们就根本不需要讨论 U_key；所有的候选码都变成了 U_key，从而可以将“候选码”（或者只是“码”）的含义推而广之。下面我们就会这样做。

我们要求类似的通用形式也应用在外码的概念上，而不加以详述。得到的一个结果就是如下形式的说明：

```
USING DURING FOREIGN KEY { S#, DURING }
             REFERENCES S_DURING
```

（这是关系变量 S_STAUTS_DURING 定义的一部分）可被用来强制执行下面的约束：如果关系变量 S_STAUTS_DURING 中显示了某个供应商在某个区间内的状态，那么关系变量 S_DURING 就会显示同一个供应商在相同的区间内是签订了合约的。相似的方法可以用来解决像在 23.2 节中提到的约束 XFT3 那样的问题。因此，现在我们已经实现了另一个初始目标：找到一种用公式表示前几节中讨论的约束的更好的方法。

**7. 九个要求**

在结束这一节前，可以看到，对于一个时态数据库的约束还有相当多的问题我们还没有讨论。文献［23.4］对整个约束问题提出了一个认真详细的分析；具体地说，在非常普遍的情况下，它考虑了九个要求，这九个要求都是像供应商 - 发货这样的典型的时态数据库应该满足的。这里我们将这些要求列出来：

- 要求 R1：如果数据库显示供应商 S*x* 在日期 *d* 时是签订了合约的，那么它必须包含了一个元组来表示这一事实。
- 要求 R2：如果数据库显示供应商 S*x* 在日期 *d* 和日期 *d* + 1 时是签订了合约的，那么它必须包含了一个元组来表示这一事实。
- 要求 R3：如果数据库显示供应商 S*x* 在日期 *d* 时是签订了合约的，那么它必须同时也显示出供应商 S*x* 在日期 *d* 具有的状态。

- 要求 R4：如果数据库显示供应商 S*x* 在日期 *d* 时具有某状态，那么它必须包含了一个元组来表示这一事实。
- 要求 R5：如果数据库显示供应商 S*x* 在日期 *d* 和日期 *d* + 1 时具有相同的状态，那么它必须包含了一个元组来表示这一事实。
- 要求 R6：如果数据库显示供应商 S*x* 在日期 *d* 时具有某状态，那么它必须同时也显示出供应商 S*x* 在日期 *d* 时是签订了合约的。
- 要求 R7：如果数据库显示供应商 S*x* 在日期 *d* 时可以供应某种零件 Py，那么它必须包含了一个元组来表示这一事实。
- 要求 R8：如果数据库显示供应商 S*x* 在日期 *d* 和日期 *d* + 1 时可以供应同种零件 Py，那么它必须包含了一个元组来表示这一事实。
- 要求 R9：如果数据库显示供应商 S*x* 在日期 *d* 时可以供应某种零件 Py，那么它必须同时也显示出供应商 S*x* 在日期 *d* 时是签订了合约的。

文献［23.4］深入地分析了这九个要求，并且展示了在一种完全关系语言（如 **Tutorial D**）中，这些要求是如何表示的。

## 23.8　小结

目前对包含时态数据的数据库（特别是数据仓库）的需求日渐增长。时态数据可被认为是**带时间戳的命题**的编码表示法。命题使用了前置词“自从……”（对当前数据）和“在……期间”（对历史数据），而且赋予了这两个术语以非常准确的含义。具体地说，“自从……”表示**从过去到现在**且不包括指定时间点之前的一刻，而“在……期间”**贯穿整个指定的区间**而不包括区间之前或者之后的一刻。

接下来我们介绍了一个很简单的例子（供应商和发货），并（1）通过加入 SINCE 属性将它半时态化，然后（2）通过加入 FROM 和 TO 属性将它完全时态化。这两个设计导致公式表示约束和查询具有了相当的复杂性。因此我们介绍了一种想法：**将区间作为数值来处理**。具体来说，定义了**点类型**的概念和 **INTERVAL 类型发生器**，讨论了相应的**区间选择子**以及 **BEGIN** 和 **END** 操作符。接着继续定义了点和区间的其他更多操作符，包括 **Allen 的操作符**和区间的 **UNION**、**INTERSECT** 以及 **MINUS** 操作符。

紧接着我们定义了两种非常重要的关系操作符 **PACK** 和 **UNPACK**（在一元关系上使用的是两个更简单的操作符 **COLLAPSE** 和 **EXPAND**）。EXPAND 和 UNPACK 使我们可以将注意力集中在原子层的关系信息上，而不用担心信息可能会通过很多途径结成“块”。类似地，COLLAPSE 和 PACK 使我们可以将注意力集中在关系信息的压缩（“块的”）形式上，而不用担心不同的“块”有邻接或者重叠的可能。我们展示了如何使用 PACK 和 UNPACK 来简化时态查询的公式表示，并且根据它们定义了常见的关系操作符的**通用**或者“**U_**”版本（U_JOIN，U_MINUS，U_project 等）。然后我们证明了那些常见的关系操作符实际上都是通用版本的特例。

然后，我们讨论了**数据库设计**问题，并且介绍了（1）**水平分解**，以分离当前信息和历史信息，以及（2）**垂直分解**，以分离一个相同“实体”的不同“属性”的相关信息（非常宽泛地说）。事实上，我们定义了新的范式，**6NF**。

接着讨论了在缺乏适当的完整性约束的情况下，时态数据可能会遇到的一些问题，具体地说就是**冗余**、**迂回**以及**矛盾**问题，并且展示了如何通过 **PACKED ON** 和 **WHEN/THEN** 约束来解决这些问题。我们定义了常见的 KEY 约束的通用化版本——**U_key 约束**，并且证明了常见的 KEY 约束只是这个通用化版本的一种特例。

最后两点需要注意的是：

- 本章中介绍的所有东西，除了 INTERVAL 类型发生器之外，都可以通过关系模型表示出来，而归根到底我们介绍的只是它们的速记。
- 我们推荐的设计方法（特别是水平分解）都有一个隐含的条件，即查询——如果可能用到，更新亦然——总是跨越了几个关系变量，并且因此具有相当的复杂性。文献［23.4］

中也包含了一系列进一步简化的建议，来减少这些复杂性。

## 习题

23.1　时间量子是什么？时间点是什么？你怎么理解粒度这个术语？

23.2　定义术语点类型和区间类型。

23.3　尽可能多地列出用一个 DURING 属性替代 FROM-TO 属性对有什么优点。

23.4　设 *i* 是 INTERVAL_INTEGER 类型的值。写一个表达式来表示将 *i* 向两边扩展得到的区间（比如 [5：7] 扩展成 [2：10]）。在什么情况下你的表达式的运行时评估会失败？

23.5　同样，设 *i* 是 INTERVAL_INTEGER 类型的值。写一个表达式来表示代表了 *i* 中间三分之一的区间（可以假设 COUNT（*i*）是 3 的倍数）。

23.6　设 *i1*、*i2* 和 *i3* 是区间，存在一个区间 *i4* 是由满足以下条件的每个点 *p* 组成的：$p \in i1$ 或者 $p \in i2$ 或者 $p \in i3$。写出一个表达式来生成 *i4*。

23.7　如果 *a* 和 *b* 是关系（或者集合），那么下面的恒等式成立：

```
a INTERSECT b  ≡  a MINUS ( a MINUS b )
```

但如果 *a* 和 *b* 是区间上述恒等式是否依旧成立？

23.8　举例（1）有两个区间属性的关系（时态的或者其他的）；（2）有三个区间属性的关系；（3）只由区间属性组成的关系。

23.9　关系 *r* 有两个不同的区间属性 A1 和 A2。证明或反驳以下命题：

```
UNPACK (UNPACK r ON A1) ON A2  ≡  UNPACK (UNPACK r ON A2) ON A1

PACK (PACK r ON A1) ON A2      ≡  PACK (PACK r ON A2) ON A1
```

23.10　给定如下关系变量：

```
FEDERAL_GOVT { PRESIDENT, PARTY, DURING }
STATE_GOVT   { GOVERNOR, STATE, PARTY, DURING }
```

语义不需加以说明（两个 DURING 属性都假设为 INTERVAL_DATE 类型；这里我们忽视一个事实，即总统和州长的任期通常是以年计而非以天计）。现在假设我们要得到的结果如下：

```
RESULT { PRESIDENT, GOVERNOR, STATE, PARTY, DURING }
```

一个元组当且仅当满足下面要求时它才会出现在这一结果中：指定的总统和指定的州长都属于指定的党，且在职时间重叠（在该题中特指 DURING 重叠）。写出一个适合的表达式来获得这个结果。

23.11　有一个区间属性的关系变量，对它进行归并时，不把这个区间属性保留在归并后的形式中。试举一例。

23.12　给出一个 U_INTERSECT 的例子，要求结果的基数要大于任何一个操作数的基数。

23.13　对操作符 U_JOIN，为了简单起见，假设归并和反归并都只在一个属性 *A* 上执行。证明下面的恒等式有效：

```
USING A ◄ r1 JOIN r2 ►
≡  WITH  ( r1 RENAME A AS X ) AS T1 ,
         ( r2 RENAME A AS Y ) AS T2 ,
         ( T1 JOIN T2 ) AS T3 ,
         ( T3 WHERE X OVERLAPS Y ) AS T4 ,
         ( EXTEND T4 ADD ( X INTERSECT Y ) AS A ) AS T5 ,
         T5 { ALL BUT X, Y } AS T6 :
   PACK T6 ON A
```

同时证明，如果 *r1* 和 *r2* 最初都在 *A* 上归并，则最后一步归并是没有必要的。注意：其中在 EXTEND 一步中的 INTERSECT 操作符是区间的 INTERSECT 而非关系的。

23.14　定义 6NF。它作为“第六”范式，与 5NF 的“第五”范式是否是相同的？

23.15　讨论：“移动点 NOW”不是数值而是一个变量。

23.16　按照 23.2 节中发货的设计，写出以下查询的 **Tutorial D** 表达式：

1）取得当前至少可以供应两种不同零件的供应商的供应商号，并且显示出每个供应商可以满足这个条件的开始日期。

2）取得当前不能供应至少两种不同零件的供应商的供应商号，并且显示出每个供应商不能供应两

种零件的开始日期。

23.17　用自己的语言解释冗余、迂回以及矛盾问题。

23.18　用自己的语言解释（1）PACK ON 约束，（2）WHEN/THEN 约束，（3）U_key 约束。解释经典的码是如何被视为 U_key 的一个特例的。

## 参考文献

这里没有长长的文献列表，建议读者将注意力放在文献［23.4］中广泛的参考书目上。

［23.1］ J. F. Allen:"Maintaining Knowledge About Temporal Intervals",*CACM 16*,No. 11（November 1983）.

［23.2］ James Clifford, Curtis Dyreson, Tomás Isakowitz, Christian S. Jensen, and Richard T. Snodgrass:"On the Semantics of 'Now' in Databases,"*ACM TODS* 22, No. 2（June 1997）.

［23.3］ Hugh Darwen and C. J. Date :"An Overview and Analysis of Proposals Based on the TSOL2 Approach"（将要出版，暂定名）。初稿可在 http://www.thethirdmanifesto.com 上获得。

之前出版的有关处理时态数据库问题的文献中，TSQL2［23.5］可能是最著名的，一些其他的方法也是以它为基础的。这篇论文提供了对这些方法的一个综述以及评论分析，并将它们与本章中的方法进行了比较。

［23.4］ C. J. Date, Hugh Darwen, and Nikos A. Lorentzos: *Temporal Data and the Relational Model*. San Francisco, Calif. :Morgan Kaufmann（2003）.

本章基本上是基于这本书的，但书中涉及了更多的细节，覆盖了更多的主题而不仅仅是本章提到的这些。其他的主题包括：

- 更多的查询操作符和速记
- 更新操作符和速记
- "有效时间与处理时间"
- 执行和优化
- 循环点类型
- 粒度和刻度
- 连续点类型

等等。值得一提的是，本书第 20 章中讨论的继承模型对于粒度问题是至关重要的（它提供了对相同的点类型有两个不同的后继函数这一问题的解答——比如"第二天"和"下个月"）。

［23.5］ Richard T. Snodgrass（ed）: *The Temporal Query Language TSQL2*. Dordrecht, Netherlands: Kluwer Academic Pub.（1995）. 同时参见 R. T. Snodgrass *et al.* :"TSQL2 Language Specification," *ACM SIGMOD Record 23*, No. 1（March 1994）.

参见文献［23.3］的评注。

# 第 24 章　基于逻辑的数据库

## 24.1　引言

20 世纪 80 年代中期前后，在数据库研究领域出现了一个重要的研究方向，这就是**基于逻辑的数据库系统**。这时有关逻辑数据库、推理 DBMS、专家 DBMS、演绎 DBMS、知识库、知识库管理系统（KBMS）、数据模型逻辑和递归查询处理等的论文相继发表。然而，很难把这些术语和思想同熟悉的数据库术语和概念联系起来；而且也很难从传统数据库的角度理解其潜在的研究动机。所以，从传统的数据库思想和原理去解释所有这些问题就变得迫切。本章试图解决这一问题。

我们的目标是从传统数据库的角度去解释什么是基于逻辑的系统，而并不是就逻辑谈逻辑。所以，当我们介绍有关逻辑的新思想时，会用传统的数据库术语去解释它，这样是可能的，也是合适的（当然，本书中已经讨论了一些有关逻辑的概念，特别是在第 8 章介绍关系演算时。关系演算直接是基于逻辑的。但基于逻辑的系统中用到的逻辑概念要远不止这些）。

本章的内容如下：24.2 节将简单地概述，并介绍一些历史；24.3 节和 24.4 节分别简单地介绍命题演算和谓词演算；24.5 节介绍所谓的数据库证明理论（proof-theoretic）；24.6 节介绍演绎 DBMS；24.7 节介绍递归查询过程的一些方法；最后，24.8 节对本章作了小结。

## 24.2　综述

对数据库理论和逻辑之间的关系的研究要至少追溯到 20 世纪 70 年代后期，关于这一时期的论文可参看［24.3］、［24.4］和［24.8］。然而，近来对这一领域兴趣大量增加的主要原因是在 1984 年 Reiter 发表的一篇具有里程碑意义的论文（参看［24.10］）。在这篇论文里，Reiter 认为传统的数据库是**模型理论**。不严格地讲，他认为：

a）在任何时候，数据库都可看作一系列明确的（比如数据）关系，每一个关系都包括一系列明确的元组；

b）执行一个查询就是对这些确定的元组和关系执行一些指定的公式（如真值表达式（truth-valued expression））。

注意：我们将在 24.5 节更精确地解释“模型理论”这一术语。

Reiter 还认为，一个可替换的**证明理论**的观点在某些方面是可能的，并且是很好的。不严格地讲，它是指：

a）在任何时候，数据库都可看作一系列公理的集合（相对于基本关系中的域(domains[○]）和元组及某些所谓的“演绎”公理来说，就是“基本”公理（ground axiom））；

b）执行一个查询就是证明一些确定的公式是这些公理的逻辑结果，或者说，证明它是一些定理。

注意：在 24.5 节将更加精确地解释“证明理论”这一术语，它能够立刻指出证明理论的观点和我们将数据库描述成是真命题集合非常接近。

下面有一个很适合的例子，考虑如下基于供应商 - 零件数据库的关系演算查询：

```
SPX WHERE SPX.QTY > 250
```

（当然，这里的 SPX 是定义在供货表上的范围变量）用传统（即模型）理论解释，供货表中的元组一个接一个地执行语句“QTY > 250”，查询结果仅仅包括供货表中一些计算结果为真值的元组。相比之下，用证明理论解释，我们就把供货表中的元组（包括其他的项）看作是一

---

○ 为了和这一领域的其他著作的写法相一致，在本章中我们使用术语 domain 而不是我们首选的术语 type。

定“**逻辑理论**”的公理；在这一理论里，我们用定理证明技术去决定范围变量 SPX 取哪些可能的值时，其逻辑结果为公式“SPX. QTY > 250”。此查询结果即由 SPX 的这些特殊值构成。

当然，这个例子非常简单，我们很难辨别这两种解释之间的差别。但是，证明（即证明理论的解释）的推理机制显然比这个简单的例子表达的要复杂得多，它能解决那些经典关系系统所不能解决的问题，而且这一证明理论还有另外的很吸引人的特征（参看［24. 10］）。

- 描述统一性：用它定义的数据库语言中，基本关系中的元组和域值、“演绎公理”、查询和完整性约束基本上用统一的方法描述。
- 操作统一性：它为各种各样明显不同的问题的统一操作提供了一个基础，这包括查询优化（尤其是语义优化）、完整性约束的实施、数据库设计（依赖理论）、程序正确性证明和其他的问题。
- 语义建模：它为各种基本模型语义的扩充提供了坚实的基础。
- 扩充应用：最后，它为处理用那些经典的方法很难处理的问题提供了基础，例如，对于析取信息处理（如，供应商 S5 或者供应了零件 P1 或者供应了零件 P2，但不知道它供应了哪一个）。

**演绎公理**

下面将简单扼要地解释演绎公理这一概念（或者叫推理规则）。基本说来，一个**演绎公理就是一个规则**，它可从给定的事实推断别的事实。例如，给定事实“Anne 是 Betty 的母亲”和“Betty 是 Celia 的母亲”，这里存在一个明显的演绎公理，从中我们推断 Anne 是 Celia 的祖母。因此，用前面所讲的理论，就可以把这个**演绎 DBMS** 中的两个给定的事实描述为关系中的元组，即：

MOTHER_OF

| MOTHER | DAUGHTER |
|---|---|
| Anne | Betty |
| Betty | Celia |

这两个事实规定了此系统的**基本公理**。我们再假设系统中以如下的形式给出这一演绎公理：

```
IF MOTHER_OF ( x, y ) AND MOTHER_OF ( y, z )
THEN GRANDMOTHER_OF ( x, z )
END IF
```

（以上为假设也简化的语法）。现在，用 24.4 节介绍的方法，系统把演绎公理所表达的规则应用于上述基本公理中的数据，从而推导出 GRANDMOTHER_OF（Anne，Celia）的结果。这样用户就可查询如下问题：“谁是 Celia 的祖母?”或“谁是 Anne 的孙女?”（或者精确到“Anne 是谁的祖母?”）。

现在把前面的思想与传统的数据库的概念联系起来。从传统的术语讲，演绎公理可以认为是一种视图定义，例如：

```
VAR GRANDMOTHER_OF VIEW
  { MX.MOTHER AS GRANDMOTHER, MY.DAUGHTER AS GRANDDAUGHTER }
    WHERE MX.DAUGHTER = MY.MOTHER ;
```

（这里特别使用关系演算的形式；MX 和 MY 分别是定义在 MOTHER_OF 上的范围变量）。上面所述查询现在可基于视图概念重写如下：

```
GX.GRANDMOTHER WHERE GX.GRANDDAUGHTER = NAME ('Celia')

GX.GRANDDAUGHTER WHERE GX.GRANDMOTHER = NAME ('Anne')
```

（GX 是定义在 GRANDMOTHER_OF 上的范围变量）。

到目前为止，我们所讲的都只不过是对一些熟悉的例子给出不同语法和解释。但是，在下一节将看到，实际上在基于逻辑的系统和更多传统的 DBMS 之间的一些重要的区别不能用这些简单的例子解释清楚。

## 24.3　命题演算

在本节和下一节，我们对一些基本的逻辑思想作简单的介绍。本节讲命题演算，下一节讲谓词演算。应当指出，命题演算本身并不是最重要的；本节真正的目的仅仅是为下一节的理解铺平道路。同时，这两节合起来是为本章后面几节提供基础。

这里，大家需要熟悉布尔代数的概念。为了引用的需要，下面列出了一些将用到的布尔代数的法则：

- 分配律：

```
f AND ( g OR  h ) ≡ ( f AND g ) OR  ( f AND h )
f OR  ( g AND h ) ≡ ( f OR  g ) AND ( f OR  h )
```

- 摩根（De Morgan）律：

```
NOT ( f AND g ) ≡ ( NOT f ) OR  ( NOT g )
NOT ( f OR  g ) ≡ ( NOT f ) AND ( NOT g )
```

这里，$f$、$g$ 和 $h$ 都是任意的布尔表达式。

下面，谈谈逻辑本身。逻辑可被定义为**规范的推理方法**。因为它是规范的，所以，它能用来执行规范的任务。如，只要检查作为每一步结果的变元结构，就能测试这个变元的正确性（即，不必注意这些步骤本身）。尤其因为它是规范化的，故它可以机械化——即，它可以设计成程序，由机器所应用。

一般地讲，命题演算和谓词演算是两个特殊的逻辑形式（实际上，前者是后者的子集）。相对于任何符号计算的系统而言，“演算”仅是一般的术语；目前情况下，所涉及的各种计算是一定公式或表达式的真值（真或假）计算。

### 1. 项

假设有一些对象的集合，称为**常量**。对这些常量，我们可作多项说明。在数据库用语里，常量是域中的值，语句可能是诸如“3 >2”的布尔表达式。**把项定义成包含常量**（constants[○]）的语句，并且：

1）项不包含任何逻辑连接词或者括号；

2）可以明确地计算项的结果为真或假。

例如，“供应商 S1 在伦敦”、“供应商 S2 在伦敦”和“供应商 S1 供应零件 P1”都是项（用通常的样本值计算，其值分别为真、假和真）。相比之下，“供应商 S1 供应零件 $p$（$p$ 是一个变量）”和“供应商 S5 在未来某一时刻供应 P1”就不是项，因为它们不能明确地计算出取真值还是取假值。

### 2. 公式

下面，我们定义**公式**的概念。命题演算（更一般地讲，是谓词演算）公式是以查询表达式的形式出现在数据库中的。

```
<formula>
    ::=    <term>
         | NOT <term>
         | <term> AND <formula>
         | <term> OR <formula>
         | <term> ⇒ <formula>

<term>
    ::=    <atomic formula>
         | ( <formula> )
```

公式的计算是基于项的真值以及连接谓词真值表。注意：

1）一个原子公式是一个布尔表达式，其中没有连接词，也不包含括号。

---

○ 更确切的讲，常量的名字，或者是包含常量的文字。在许多文献中它们的区别并不明确。

2）符号“⇒”表示**逻辑蕴含连接**。表达式 $f⇒g$ 在逻辑上等价于表达式（NOT $f$）OR $g$。注意在第 8 章或其他章节里，我们使用“IF…THEN…END IF”表示这个连接。

3）为了表达一个请求的计算顺序，通常对这些连接词规定优先次序（即 NOT、AND、OR、⇒，由高到低的次序）来减少要使用括号的数量。

4）一个命题，就是上面定义的一个公式（为了与下一节保持一致性，这里使用了术语“公式”）。

**3. 推理规则**

下面讨论命题演算的推理规则。这样的规则有很多。其中每一条规则都可表述为下述形式的语句：

```
⊨ f ⇒ g
```

（这里 $f$ 和 $g$ 是公式，符号⊨ 读作“**总是有**”；为了能产生某些语句——即由语句推导出的语句，的确需要这样的符号）。下面就列举一些推理规则：

1) ⊨ ( f AND g ) ⇒ f

2) ⊨ f ⇒ ( f OR g )

3) ⊨ ( ( f ⇒ g ) AND ( g ⇒ h ) ) ⇒ ( f ⇒ h )

4) ⊨ ( f AND ( f ⇒ g ) ) ⇒ g

注意：这一推理规则特别重要。这叫**假言推理规则**。非正式地讲，其含义是如果 $f$ 为真，且 $f$ 蕴含 $g$，则 $g$ 一定为真。例如，下面的 $a$ 和 $b$ 都为真，

a）我没有钱。

b）如果我没有钱，那我就不得不刷碟子。

则我们就一定能推断 c 也为真：

c）我不得不刷碟子。

下面的 2 条紧接着上面的推理规则：

5) ⊨ ( f ⇒ ( g ⇒ h ) ) ⇒ ( ( f AND g ) ⇒ h )

6) ⊨ ( ( f OR g ) AND ( NOT g OR h ) ) ⇒ ( f OR h )

注意：这是另一个特别重要的规则，叫作**归结规则**。在下面要讲的“证明”一小节和第 24.4 节将对此详细讨论。

**4. 证明方法过程**

现在，我们讨论形式证明的方法问题（在命题演算的范畴）。这就是判定一个给定的公式 $g$（结论 **conclusion**）是否是另一些给定的公式 $f1$、$f2$、…，$fn$（前提 **premises**○）的逻辑结果，用符号表示就是：

```
f1, f2, ..., fn ⊢ g
```

（读作“从 $f1$、$f2$、…，$fn$ **推导出** $g$”。注意这里使用的另一个元语言符号“⊢”）。这种基本方法叫做**前向推理**（forward chaining）。前向推理就是对这些前提、由这些前提导出的公式、由这些公式导出的公式，等等重复运用推理规则，直到推导出所需要的结论；或者说这一过程就是前面提到结论的“正向链”。然而，对这一基本的论题，下面还有几个证明思路：

1）**附加一个前提**：如果 $g$ 是 $p⇒q$ 的形式，采用 $p$ 作为附加的前提，并表明 $q$ 可从给定的前提及 $p$ 推导出来。

2）**反向推理**：不直接去证明 $p⇒q$，而是去证明其**逆否命题**，即 NOT $q$⇒NOT $P$。

3）**反证法**：不是直接去证明 $p⇒q$，而是先假设 $p$ 和 NOT $q$ 为真，再推导出相互矛盾的结果。

---

○ 也可以写作 premisses（单数形式为 premiss）。

4）**归结**：这种方法要使用归结推理规则（如上述的第 6 条推理）。

现在我们来具体地讨论归结规则，它有着广泛的应用（特别是它可以推广到谓词演算的情况。参见 24.4 节）。

首先要注意，归结规则是很有效的，可以运用它消除一些子公式。如，给出两个公式：

```
f OR g          and          NOT g OR h
```

我们可以删去 $g$ 和 NOT $g$，得到如下简化的公式：

```
f OR h
```

特别地，若有 $f$ OR $g$ 和 NOT g（即 $h$ 取真值），则可以导出 $f$。

因此，一般情况下，这些规则运用在**与**（AND）连接的两个公式上，并且每一个公式是两个子公式的**或**（OR）。下面，我们运用这一归结规则，证明下式的有效性：

```
A ⇒ ( B ⇒ C ), NOT D OR A, B ⊢ D ⇒ C
```

（这里 $A$、$B$、$C$ 和 $D$ 都是公式）。现在，我们将结论的否定形式作为附加的前提，并把每个前提分行列出：

```
A ⇒ ( B ⇒ C )
NOT D OR A
B
NOT ( D ⇒ C )
```

这四行隐含着与（AND）连接。

现在把每一行转化为合取范式，即一个或一个以上的、由 AND 连接起来的公式，每一个独立的公式可能包含 NOT 和 OR，但不包含 AND（参看第 18 章）。当然，第二、三行已符合要求，下面转化其他两行。首先我们根据 NOT 和 OR 连接的定义，删除掉所有的符号“⇒”；接着，必要的时候运用分配律和摩根律，并去掉冗余的括号和邻近的 NOT，这样得到：

```
NOT A OR NOT B OR C
NOT D OR A
B
D AND NOT C
```

接着，对于任何包括 AND 的行，把它转化为一系列独立的行，每一行是一个由 AND 连接的公式（即在这一过程中删掉 AND）。此例中，这一步仅用在第四行。现在，前提看起来如下所示：

```
NOT A OR NOT B OR C
NOT D OR A
B
D
NOT C
```

下面，我们开始运用归结规则。先选择能被归结的两行，即分别包含某个特殊的公式及其否定形式。我们选择头两行，它们分别包含 NOT $A$ 和 $A$，归结它们，得到：

```
NOT D OR NOT B OR C
B
D
NOT C
```

注意：在一般情况下，我们需要保留原始的两行，但在这个特殊的例子里，我们不再需要它们了。

再一次选择头两行，运用这一规则（归结 NOT $B$ 和 $B$），得：

```
NOT D OR C
D
NOT C
```

还是选择头两行（NOT $D$ 和 $D$），得：

```
C
NOT C
```

再处理头两行（NOT *C* 和 *C*）；最后的结果是命题的空集（通常表示为 [ ]），这是一对矛盾。这样，通过反证法，证明原命题是无效的。

## 24.4　谓词演算

现在讨论谓词演算。命题演算和谓词演算之间最大的区别在于后者允许公式中有变量[○] 和量词，这使得它的功能更为强大，应用更为广泛。例如，语句“供应商 S1 供应零件 *p*”和“某位供应商 *s* 供应零件 *p*”在命题演算中不合法，但在谓词演算中它们是合法的公式。因此，谓词演算给我们表达如下的查询提供了基础：“S1 供应了哪些零件?”或“查询供应某一零件的供应商”，甚至“查询根本不供应任何零件的供应商”。

**1. 谓词**

正如第 3 章解释的那样，一个**谓词**就是一个真值函数，即给出合适的自变量参数，其返回结果为真或假的函数。例如，“ > （*x*, *y*）”——更一般地写作“*x* > *y*”——是一个包含有两个参数的谓词，即 *x* 和 *y*；如果相应于 *x* 的变元大于相应于 *y* 的变元，则它返回真值；否则，返回假值。一个用了 *n* 个变元的谓词（即，这个谓词由 *n* 个参数所定义）叫做 *n* 元谓词。一个命题（24.3 节所讲的公式）可以认为是零元谓词，没有参数，并且其结果明确地为真或假。

假设相应于“ =”、“ >”、“ ≥”等的谓词是固有的（即它们是定义的规范系统的一部分），并且用一种习惯的方式将使用它们的表达式写出来。当然，用户也应能定义他们自己的谓词。整个问题的关键就是：实际上用数据库的术语讲，一个用户定义的谓词对应一个用户定义的关系变量（从前面几章我们也可以了解这一点）。例如，供应商的关系变量 S，可以看作是一个包含四个参数的谓词，即 S#、SNAME、STATUS 和 CITY。而且，表达式 S（S1，Smith，20，London）和 S（S6，White，45，Rome）分别表示该谓词结果为真和假的“实例”（或者说是实例化、调用）。不严格地说，正像前面几章所讲的那样（特别是第 9 章），我们可以把这些谓词，与可能的完整性约束（同时也是谓词）一起作为数据库内容的定义。

**2. 合式公式**

下一步是扩展“公式”的定义。为了避免与前面几节讲的公式（实际上，前文中的是特殊的情形）混淆，现在我们转到第 8 章所讲的合式公式（WFF）这一术语。下面是一简单的合式公式的语法：

```
<wff>
    ::=   <term>
        | NOT ( <wff> )
        | ( <wff> ) AND ( <wff> )
        | ( <wff> ) OR ( <wff> )
        | ( <wff> ) ⇒ ( <wff> )
        | EXISTS <var name> ( <wff> )
        | FORALL <var name> ( <wff> )

<term>
    ::= [ NOT ] <pred name> [ ( <argument commalist> ) ]
```

注意：

1）简单地讲，一个项（< *term* >）可能不是“谓词实例”（如果把谓词看作真值函数，那么一个谓词实例就是那个函数的调用）。每一个变元（< *argument* >）必须是一个常量、一个变量名或一个函数调用，而函数调用的每一个变元也必须是这样。在零元谓词里可以删去变元列表（< *argument commalist* >）（可选的）及相应的括号。注意：为了 WFF 能使用包括诸如“ + （*x*, *y*）”（更一般地写作“*x* + *y*”）的计算表达式，可以使用建立在作为谓词的函数之上的函数。

2）正如 24.3 节讲的那样，为了减少因表达计算顺序而引入的括号数目，我们对连接词使用

---

○ 更确切的讲是变量名。这里的变量是指逻辑变量，而不是程序语言中的变量，也可以认为是第 8 章中讲的范围变量。

优先规则（即 NOT、AND、OR、⇒，按优先级排列）。

3）假设大家熟悉量词 EXISTS 和 FORALL。注意：这里讲的仅是一阶谓词演算，其基本意思是：（a）它没有“谓词变量”（即允许值是谓词的变量），因此（b）谓词本身并没有受到量化约束。具体请看第 8 章练习 8.8。

4）摩根律可以推广应用到合式公式，如下：

```
NOT ( FORALL x ( f ) )  ≡  EXISTS x ( NOT ( f ) )
NOT ( EXISTS x ( f ) )  ≡  FORALL x ( NOT ( f ) )
```

这一点也已经在第 8 章讨论了。

5）现在重复一下第 8 章另一个要点：在一个给定的 WFF 里，每一个变量的引用要么是自由的，要么是约束的。如果一个引用是**约束**的，当且仅当（a）它紧跟 EXISTS 或者 FORALL 之后（即它表示约束的变量）；（b）它位于量词的范围内并表示可适用的约束变量。如果一个变量引用是**自由**的，当且仅当它不是约束的。

6）一个**封闭**的合式公式不包含自由变量。（实际上，这是一个命题。）一个开放的合式公式就是一个不封闭的合式公式。

### 3. 释义和模型

合式公式的含义是什么？为了形式化地回答这个问题，我们引入了**释义**（interpretation）这一概念。一系列合式公式的释义如下定义：

- 首先，我们指定要解释这些合式公式所需要的**论域**（universe of discourse）。或者说，在（a）规范系统所允许的常量（用数据库的术语讲，就是域值）和（b）真实世界的对象之间指定一个映像。每一个单个的常量明确地对应于论域中的一个对象。
- 第二，根据论域中的对象，为每个谓词指定一个含义。
- 最后，根据论域中的对象，为每个函数指定一个含义。

这样，这一释义就包括论域、论域中的对象与常量个体之间的映像、根据论域对谓词和函数所作的定义。

下面举例说明。设论域是整数集 {0，1，2，3，4，5}，常量 2 以明显的方式对应于此论域中的某对象，设谓词“$x > y$”定义成通常的含义（若需要的话，也可定义诸如“+”、“-”等函数）。现在，我们给下面的合式公式赋真值，如下所示：

```
2 > 1                  :  TRUE
2 > 3                  :  FALSE
EXISTS x ( x > 2 )     :  TRUE
FORALL x ( x > 2 )     :  FALSE
```

注意，存在其他的释义是可能的。例如，我们可以给论域取一系列安全分类级别，如下：

```
destroy before reading  (level 5)
destroy after reading   (level 4)
top secret              (level 3)
secret                  (level 2)
confidential            (level 1)
unclassified            (level 0)
```

这里谓词“>”就表示“更安全的（即更高的安全级别）”。

现在，你可能觉得上面的两个可能的释义在形式上是一样的，即在它们之间建立一一对应关系是可能的，且从更深一层次讲，这两个释义其实就是一个。但是必须知道，释义可以存在真正的不同。例如，我们把论域定义为 0～5 的整数，但是定义谓词“>”为相等（当然，这样做会引起混淆，但至少这是合法的）。现在上面的第一个合式公式“2 > 1”就取假而不是取真。

另一点必须明确的是，从上述的意义讲，这两个释义可能是真正的不同，但是对某些给定合式公式集，它们都取同样的真值。在我们的例子里，如果删除合式公式“2 > 1”，则对于“>”的两个不同的定义，就会出现这种情况。

还需要注意的是，这一小节之前所讨论的合式公式都是封闭的合式公式。这是因为给定一个

释义，对每一个给定的封闭的合式公式，总能明确地给它赋一个真值。但是，开放的合式公式的真值依赖于赋给自由变量的值。例如，开放的合式公式：

```
x > 3
```

很明显，只要 x 比 3 大，则其值为真，否则其值为假（无论“大于”和“3”在这里表示什么）。

现在，我们来解释被释义的一组合式公式（必须是封闭的）的**模型**，对于这个释义，所有其中的合式公式取值为真。我们给出四个合式公式：

```
2 > 1
2 > 3
EXISTS x ( x > 2 )
FORALL x ( x > 2 )
```

用 0 ~ 5 的整数的话，上面给出的两个释义并不是这些合式公式的模型。因为在那种释义下，其中某些合式公式结果为假。相比之下，上面的第一个释义（即“ > ”定义为“大于”）应该是下面合式公式的模型：

```
2 > 1
3 > 2
EXISTS x ( x > 2 )
FORALL x ( x > 2 OR NOT ( x > 2 ) )
```

最后还要注意，因为对于给定的一组合式公式能接受几种释义，且其中的合式公式为真，因此它就有几个模型（一般地讲）。所以，一个数据库可能有多个模型，用模型理论的观点来讲，一个数据库仅仅是一组合式公式。具体见 24.5 节。

**4. 子句式**

像任何一个命题演算公式都可转化为合取范式一样，任何一个谓词演算合式公式都可转化为**子句式**，转化后的形式被认为是合取范式的扩展形式。进行这种转化的动机在于，我们可以把归结规则运用于构建和验证证明。

这一转化过程如下（这里是概括性的内容，详细内容请参看 [24.6]）。通过举例来解释这些步骤。

```
FORALL x ( p ( x ) AND EXISTS y ( FORALL z ( q ( y, z ) ) ) )
```

这里 p 和 q 是谓词，x、y 和 z 是变量。

1）像 24.3 节那样删除符号“⇒”。此例中，这个转化没有影响。

2）使用摩根定律，删去两个邻近的 NOT。移动 NOT，使它仅应用于项，而不是整个合式公式。（该转化仍对此例没有影响。）

3）把所有的量词都移到前面，从而将这些公式转化为前束范式。（如果有必要，可系统地对变量重命名）：

```
FORALL x ( EXISTS y ( FORALL z ( p ( x ) AND q ( y, z ) ) ) )
```

4）注意一个有存在量词的量化的合式公式：

```
EXISTS v ( r ( v ) )
```

等价于合式公式：

```
r ( a )
```

其中 a 为未知常量。最初的合式公式中存在 a 这样的未知常量，我们不知道其值。同样，对于合式公式：

```
FORALL u ( EXISTS v ( s ( u, v ) ) )
```

等价于合式公式：

```
FORALL u ( s ( u, f ( u ) ) )
```

其中，全称量词 $u$ 的函数 $f$ 是未知的。这里常量 $a$ 和函数 $f$ 分别命名为**斯科林**（Skloem）**常量**和**斯科林**（Skloem）**函数**，这一命名来自于逻辑学家 T. A. Skloem（注意：一个 Skloem 常量仅是一个没有变元的 Skloem 函数）。因此，下一步就是通过取代相应的约束变量而删除存在量词。其中，这些变量由下式的量词前面的全称量词的 Skloem 函数（任意的）所限制：

```
FORALL x ( FORALL z ( p ( x ) AND q ( f ( x ), z ) ) )
```

5）现在，所有的变量都被全称量词约束。因此，我们习惯上可以将所有的变量都隐含为全称量词约束，删除所有的显式量词：

```
p ( x ) AND q ( f ( x ), z )
```

6）将合式公式转化为合取范式，即由 AND 连接的子句，每一子句只可能有 NOT 或 OR，而没有 AND。此例中，此合式公式已是这种形式。

7）删去 AND，把每个子句写在独立的行上，如下：

```
p ( x )
q ( f ( x ), z )
```

这一子句等价于最初的合式公式。

注意：可以从上述的转化过程得出，用子句形式的一个合式公式的一般形式就是一组子句，且每一个占一行，形式如下：

```
NOT A1 OR NOT A2 OR ... OR NOT Am OR B1 OR B2 OR ... OR Bn
```

这里 $A$ 和 $B$ 都是正项。我们可以转化这个子句，如下所示：

```
A1 AND A2 AND ... AND Am  ⇒  B1 OR B2 OR ... OR Bn
```

如果最多有一个 $B$（$n=0$ 或 1），则这个子句叫做 **Horn 子句**，它是以逻辑学家 Alfred Horn 的名字命名的。

**5. 使用归结规则**

下面讨论基于逻辑的数据库系统对查询的处理过程。我们沿用 24.2 节最后的那个例子。首先，有谓词 MOTHER_OF，它包括两个参数，分别表示母亲与女儿，再给出下面两个项（谓词实例）：

```
1) MOTHER_OF ( Anne,  Betty )
2) MOTHER_OF ( Betty, Celia )
```

给出下面的合式公式（演绎公理）：

```
3) MOTHER_OF ( x, y ) AND MOTHER_OF ( y, z ) ⇒
                                  GRANDMOTHER_OF ( x, z )
```

（注意这是一个 Horn 子句）。为了简化归结规则的运用，我们删去上式中的符号“⇒”。

```
4) NOT MOTHER_OF ( x, y ) OR NOT MOTHER_OF ( y, z )
                          OR GRANDMOTHER_OF ( x, z )
```

现在证明 Anne 是 Celia 的祖母——即如何回答查询“Anne 是 Celia 的祖母吗?”，现在我们将需要证明的结论的否定形式作为前提：

```
5) NOT GRANDMOTHER_OF ( Anne, Celia )
```

为了应用归结规则，必须能找到两个子句，分别包含一个合式公式及它的否定公式，并系统地给变量赋值。这样的赋值是合法的，因为这些变量都已隐含为全称量词约束，所以对于每一种变量的值的组合，单个的合式公式（非否定的）必须为真。注意：寻找一系列使这两个子句可

归结的值的过程叫做合一。

下面用例子表明上述过程如何进行，注意到4和5分别包含项 GRANDMOTHER_OF（$x$，$z$）和 NOT GRANDMOTHER_OF（Anne，Celia）。因此，我们用 Anne 取代 $x$，用 Celia 取代 $y$ 并归结，得：

```
6) NOT MOTHER_OF ( Anne, y ) OR NOT MOTHER_OF ( y, Celia )
```

第二条包括 MOTHER_OF（Betty，Celia）。我们用 Betty 取代上面6中的 y 并归结，得：

```
7) NOT MOTHER_OF ( Anne, Betty )
```

归结上面的7和1，我们得空子句集 [ ]，从而矛盾。所以，起初查询的答案应是“是的，Anne 是 Ceila 的祖母”。

对于查询“Anne 的孙女是谁?”又怎么样呢？首先注意到，系统不知道孙女，它只知道祖母。我们可增加另一演绎公理：$z$ 是 $x$ 的孙女，当且仅当 $x$ 是 $z$ 的祖母（此数据库中不允许有男性）。当然，也可以把这个问题叙述为“Anne 是谁的祖母?”。我们来讨论后面一个公式。前提是（和以前一样）：

```
1) MOTHER_OF ( Anne,  Betty )
2) MOTHER_OF ( Betty, Celia )
3) NOT MOTHER_OF ( x, y ) OR NOT MOTHER_OF ( y, z )
                          OR GRANDMOTHER_OF ( x, z )
```

我们引入第四个前提，如下：

```
4) NOT GRANDMOTHER_OF ( Anne, r ) OR RESULT ( r )
```

这个新前提直观地表明要么 Anne 不是任何人的祖母，要么有某个人 $r$ 符合这一结果（因为 Anne 是 $r$ 的祖母）。我们希望发现 $r$ 的身份。可如下进行：

首先，用 Anne 取代 x，$r$ 取代 $z$，并归结上面的3和4，得：

```
5) NOT MOTHER_OF ( Anne, y ) OR NOT MOTHER_OF ( y, z )
                             OR RESULT ( z )
```

接着，用 Betty 取代 $y$，并归结上面的5和1，得：

```
6) NOT MOTHER_OF ( Betty, z ) OR RESULT ( z )
```

现在，Celia 取代 $z$，并归结上面的6和2，得：

```
7) RESULT ( Delia )
```

因此，Anne 是 Celia 的祖母。

注意：如果给出一个附加项，如下：

```
MOTHER_OF ( Betty, Delia )
```

这样我们就能在最后一步用 Delia（不是 Celia）取代 $z$，得：

```
RESULT ( Delia )
```

当然，在查询结果中，用户期望看到两个名字。这样，系统就需要运用合一和归结过程去生成所有可能的结果。进一步的细节已超出本讨论的范围。

## 24.5　数据库的证明理论观点

正像24.4节解释的那样，一个子句就是下述形式的一个表达式：

```
A1 AND A2 AND ... AND Am  ⇒  B1 OR B2 OR ... OR Bn
```

这里 *A* 和 *B* 都是下述形式的项：

```
r ( x1, x2, ..., xt )
```

（其中 *r* 是谓词，*x1*，*x2*，…，*xt* 是它的变元）。参看［24.7］，考虑该一般结构中两个重要的特殊的情况：

1）`m = 0, n = 1`

这种情况下，这个子句可简单地写为：

```
⇒ B1
```

或者写为（删去蕴含符号）：

```
r ( x1, x2, ..., xt )
```

其中 *r* 为谓词，*x1*，*x2*，…，*xt* 为变元。如果 *x* 都是常量，则此子句表示一个**基本公理**（ground axiom），且结果一定为真。在数据库项里，这样的语句相对应于关系变量 *R* 的一个元组[⊖]。谓词 *r* 相对应于关系变量 *R* 的“含义”，本书的其他一些地方解释过这一点。例如，在供应商和零件数据库里，有一个关系变量叫 SP，它的意思是某一个供应商（S#）以某一数量（QTY）供应某一零件（P#）。这个含义相对应于一个**开放式合式公式**，因为它包含自由变量（S#，P#，QTY）的引用。相比之下，元组（S1，P1，300）——其变元都是常量——是一个基本公理或**封闭式合式公式**，它表示供应商 S1 供应了 300 个零件 P1。

2）`m > 0, n = 1`

在这种情况下，子句采用下述形式：

```
A1 AND A2 AND ... AND Am  ⇒  B
```

这叫做**演绎公理**，蕴含符号左边的谓词是右边的谓词的定义（不完全定义）（可参看前面例子中的 GRANDMOTHER_OF 谓词的定义）。

或者，这样的子句可被定义为**完整性约束**（使用第 9 章的术语，即一个关系变量约束）。例如，供应商变量 S 有两个属性 S#和 CITY，则子句：

```
S ( s, c1 ) AND S ( s, c2 )  ⇒  c1 = c2
```

表达了这样的约束：CITY 函数依赖于 S#。注意这里固有谓词“ = ”的使用。

正如前面讨论所解释的那样，关系（“基本公理”）中的元组、导出关系（“演绎公理”）及完整性约束是一般子句结构的特殊情形。现在来看一看对于 24.2 节所讲的数据库的“证明理论”的观点，是如何由这些思想导出的。

首先，数据库的传统观点被认为是**模型理论**。这里讲“传统观点”，是指数据库被理解为由显式命名关系变量的集合组成，每一个关系变量又由一系列显式元组及显式完整性约束组成。正是基于这种理解，所以称这种观点为模型理论。看下面的解释：

- 基本（underlying）的域包括值或常量，它们被假设代表“现实世界”（更精确地讲，像 24.4 节那样，是某种**释义**）中一定的对象。这样它们就对应于论域。
- 关系变量（更精确地讲，是关系变量头部）表示一系列谓词或者开放式合式公式，以及这些合式公式在此论域上释义。例如，关系变量头 SP 表示谓词“供应商 S#以数量 QTY 供应零件 P1”。
- 给定的关系变量里的每一个元组表示一个相应谓词的实例；即表示在这个论域里结果明确为真的命题（一个封闭的合式公式，它没有变量）。

---

⊖ 或者相应于某个域中的一个值。

- 完整性约束也是封闭式合式公式，并且它们在同一域上释义。所以，这些数据并没有（不可能）违背这些约束，这些约束结果必须为真，同样，代入参数时，当前数据库的值也要为真。
- 元组和完整性约束在一起就被看作一组公理，它定义某种**逻辑理论**（不严格地讲，一个“理论”在逻辑上就是一组公理）。由于在这一释义里，这些公式为真，所以从 24.4 节所述的意义讲，该释义就是逻辑理论的**模型**。注意到正如前一节所指出的，这个模型不是唯一的，即一个给定的数据库可能有几个释义，从逻辑的角度讲，所有这些释义都是同样有效的。

因此，在模型理论的观点里，按“模型”的术语讲，数据库的含义就是模型。并且由于有许多可能的模型，故有许多可能的意义（至少从理论上讲是这样的）[⊖]。而且，在模型理论观点下的查询过程基本上就是计算某一个开放的合式公式的过程。这一过程是为了发现在这一模型里，合式公式中哪一个自由变量使得这个合式公式取真值。

模型理论的观点内容很丰富。然而，为了能运用 24.3 节和 24.4 节所讲的推理规则，就必须采用一个不同的视角。在这个视角中，数据库被明确地看作一定的逻辑理论，即作为一系列公理。这样，数据库的意义就变成所有真语句的精确集合，而这些真语句是从一些公理演绎来的，即能由那些公理证明的定理，这就是**证明理论**的观点。利用这一观点，查询工作就变成是一个证明理论的过程（任何情况下，从概念上讲是这样的；但是，为了有更好的效率，系统可能还要使用更多的传统的查询技术。24.7 节再讲这一点）。

*注意*：从前面一段可以得出，直观上模型理论与证明理论的观点间的区别是，在模型理论的观点中，一个数据库有许多“意义”，而在证明理论的观点中，一个数据库仅有一个“意义”。但是（a）正如前面指出的，在模型理论中，数据库的意义是真正的规范的意义，并在任何情况下；（b）一般地讲，如果数据库包含任何一个否定的公理，则在证明理论情形下，数据库只有一个意义就不再成立参看［24.5，24.6］。

证明理论的观点中，给定数据库的公理可总结如下（参看［24.10］）：

1）基本公理相对于库关系变量中域值或元组值。这些公理有时组成了所谓的**外延数据库**（相对于内涵数据库，见下一节）。

2）每一个关系变量的“完整公理”（completion axiom）表明，我们所讲的关系变量中有效元组的失效可被解释为相对于此元组的命题是假的（当然，实际上，这些完整化公理在一起就构成了**封闭世界假说**，这在第 6 章和第 9 章已经讨论了）。例如，供应商变量 S 不包括元组（S6，White，45，Rome）就是指命题“存在供应商 S6，它的名字是 White，它的等级是 45，它住在罗马”为假。

3）“唯一命名”公理，表明每一个常量都不同于其他的任何常量，即它有唯一的命名。

4）“域闭包”公理，表明不存在数据库域以外的常量。

5）一组定义固有等价谓词的基本公理。需要这组公理，是因为上面的 2、3、4 利用了等价谓词。

现在，我们对模型理论和证明理论这两个概念之间的基本区别作简短的小结。首先，有人说，仅从实用的角度讲，它们之间根本没有什么区别，至少从今天的 DBMS 的角度讲是这样的。但是，有几点应当注意：

- 证明理论观点中的公理（除了基本公理）都有明确的假设，而在模型理论的观点中，这一假设隐含在释义的概念中（参看［24.10］）。该观点明确地列出这些假设在一般情况下是一个好的思想；而且，为了能应用一般的证明理论的技术（如在 24.3 节和 24.4 节讲的归一方法），明确指定这些公理是必要的。
- 注意到上述公理没有讲到完整性约束规则。这是因为如果加上这些约束（在证明理论的

---

⊖ 然而，如果我们假设数据库不显示的包含任何否定信息（例如，一个“NOT S#（S9）”形式的命题表示 S9 不是一个供应商号），仍然会有“最小的”或者规范的意义，这就是所有可能模型的交集（参看［24.6］）。而且，在这种情况下，这一规范的意义与在证明理论的观点下的数据库的意义一样。这会在以后适当时候解释。

观点里），系统就变为演绎 DBMS。24.6 节讨论这一问题。

- 证明理论的观点确实有一定的优雅性，而模型理论的观点则没有。证明理论为一些结构提供了统一的理解，而这些结构通常被认为有或多或少的不同。这些结构有：基本数据、查询、完整性约束（虽然是以前的观点）、虚拟数据，等等。所以，更统一的接口和更统一的实现的可能性就提高了。
- 证明理论的观点也为关系系统在传统上有处理困难的那些问题提供了良好的基础。如**析取信息**（例："供应商 S6 住在伦敦还是巴黎"），**否定信息导出**（例："谁不是供应商?"）、**递归查询**（见下一节）等。对递归查询，尽管几个商业系统已能够加以处理[○]，但原则上还不知道为什么经典关系系统不能被合适地扩展以解决此类查询。我们将在 24.6 节和 24.7 节详细讨论这一问题。
- 最后，引用 Reiter 的话（参看 [24.10]），就是证明理论的观点"为关系模型（扩展的）和现实世界语义的结合提供了正确的处理方法"（24.2 节还将讲到）。

## 24.6　演绎数据库系统

一个**演绎 DBMS** 就是支持证明理论观点的 DBMS。特别地，对给定的外延数据库中的事实运用指定的演绎公理或推理规则，就可以从这些**事实演绎**或**推导别的事实**[○]。演绎公理和完整性规则（下面将讨论）合在一起有时就叫做内涵数据库。外延数据库和内涵数据库合在一起就构成了通常讲的演绎数据库（这不是一个很恰当的术语，因为它是运用演绎原理的 DBMS，而不是数据库）。

正如刚刚所讲的，演绎公理构成了内涵数据库的一部分，而另一部分是表示完整性约束的公理（这些规则的主要目的是约束更新，实际上它们也可以用在从给定事实演绎另外的事实的推理过程中）。

现在我们来看一看图 3-8 所示的供应商和零件数据库，其在"演绎 DBMS"中的形式如何。首先，这里有一系列基本公理定义了一些合法的域值。注意：在这里，为了可读性，我们采用了图 3-8 中表示值的同样的方法，如表示 300 在习惯上就可简写为 QTY（300），等等。

```
S# ( S1 )   NAME ( Smith )   INTEGER (  5 )   CHAR ( London )
S# ( S2 )   NAME ( Jones )   INTEGER ( 10 )   CHAR ( Paris  )
S# ( S3 )   NAME ( Blake )   INTEGER ( 15 )   CHAR ( Rome   )
S# ( S4 )   NAME ( Clark )   etc.             CHAR ( Athens )
S# ( S5 )   NAME ( Adams )                    etc.
S# ( S6 )   NAME ( White )
S# ( S7 )   NAME ( Nut   )
etc.        NAME ( Bolt  )
            NAME ( Screw )
            etc.
```

对基本关系中的元组有如下的基本公理：

```
S ( S1, Smith, 20, London )
S ( S2, Jones, 10, Paris  )
etc.
P ( P1, Nut, Red, 12, London )
etc.

SP ( S1, P1, 300 )
etc.
```

注意：我们并没有严格地表明上面列出的基本公理将一定产生外延数据库；不过，我们将使用传统的数据定义和数据登录方法。或者说，演绎 DBMS 直接作用在由传统方法构建的已存在的数据库上。然而要注意的是，外延数据库不违背那些已定义的完整性约束！否则，任何一个违

---

○ 所以有了 SQL 标准 [4.23]。看第 4 章中的练习 4.6。

○ 关于这一点，值得注意的是，在 1974 年，Codd 就认为关系模型中的目标之一就是"把事实检索和文件管理领域合并起来为后来商业中的推理性服务做好准备"。（参看 [12.2]，[26.12]）

背如此约束的数据库就推翻了这一系列公理的一致性（逻辑上），并且我们都知道从这个点出发，任何命题都能够证明为“真”（换句话说，可导出矛盾，参考 [9.16]）。鉴于此，要求完整性约束集是一致的就显得重要。

现在来看外延数据库。下面是一些属性约束：

```
S ( s, sn, st, sc )  ⇒  S# ( s ) AND
                        NAME ( sn ) AND
                        INTEGER ( st ) AND
                        CHAR ( sc )

P ( p, pn, pl, pw, pc )  ⇒  P# ( p ) AND
                            NAME ( pn ) AND
                            COLOR ( pl ) AND
                            WEIGHT ( pw ) AND
                            CHAR ( pc )
etc.
```

候选码约束：

```
S ( s, sn1, st1, sc1 ) AND S ( s, sn2, st2, sc2 )
                       ⇒  sn1 = sn2 AND
                          st1 = st2 AND
                          sc1 = sc2
etc.
```

外码约束：

```
SP ( s, p, q )  ⇒  S ( s, sn, st, sc ) AND
                   P ( p, pn, pl, pw, pc )
```

等等。注意：为了说明的方便，我们假设在蕴含符号右边的变量（即此例中的 *sn*、*st* 等）被存在量词所约束（所有其他变量就像 24.4 节讲的那样，被全称量词约束）。从技术上讲，我们需要一些斯科林函数；例如 *sn*，实际上，*sn* 应该由 SN（*s*）所替代，这里 SN 就是一个斯科林函数。

还要注意，上面的大部分约束不仅仅是 24.5 节所讲的意义上的子句，因为上式右边不仅仅是简单项的逻辑和。

下面，我们再增加一些演绎公理：

```
S ( s, sn, st, sc ) AND st > 15
                    ⇒  GOOD_SUPPLIER ( s, st, sc )
```

（比较第 10 章 10.1 节中 GOOD_SUPPLIER 视图的定义）

```
S ( sx, sxn, sxt, sc ) AND S ( sy, syn, syt, sc )
                                ⇒  SS_COLOCATED ( sx, sy )

S ( s, sn, st, c ) AND P ( p, pn, pl, pw, c )
                                ⇒  SP_COLOCATED ( s, p )
```

等等。

为了使这个例子更具说明性，我们把这个数据库扩展到包括“零件结构”这一关系变量，它表明零件 *px* 由哪些零件 *py* 构成它的组件（只涉及第一级的）。第一个约束要表明 *px* 和 *py* 都必须能标识存在的零件：

```
PART_STRUCTURE ( px, py )  ⇒  P ( px, xn, xl, xw, xc ) AND
                              P ( py, yn, yl, yw, yc )
```

数据值包括：

```
PART_STRUCTURE ( P1, P2 )
PART_STRUCTURE ( P1, P3 )
PART_STRUCTURE ( P2, P3 )
PART_STRUCTURE ( P2, P4 )
(etc.)
```

（实际上，PART_STRUCTURE 也可能有一个“数量”变元，去表明一个 *px* 由多少个 *py* 组成，但为了简单起见，我们删去了这一细节）。

下面增加两个演绎公理来解释零件 *px* 由零件 *py*（任何一级的）组成是什么意思：

```
PART_STRUCTURE ( px, py )  ⇒  COMPONENT_OF ( px, py )

PART_STRUCTURE ( px, pz )  AND COMPONENT_OF ( pz, py )
                           ⇒  COMPONENT_OF ( px, py )
```

或者说，如果零件 *py*（某一级的）是 *px* 的中间组件或是 *pz*（某一级的）的中间组件，而 *pz* 又是 *px* 的中间组件，则 *py* 就是 *px* 的组件。注意到第二个公理是递归的，根据它本身又定义了 COMPONENT_OF 谓词 。相比而言，在关系系统的历史上，是不允许如此递归地定义视图（或查询或完整性约束……）。演绎 DBMS 相对于经典关系系统来说，支持递归能力是其最直接、最明显的区别之一。尽管，就像 24.5 节（和第 7 章 TCLOSE 操作）所说的那样，对于为什么经典的关系系统不能扩展去支持如此的递归，还没有什么根本的解释；但某些系统确实已经开始支持这一点。我们将在 24.7 节详细讨论递归。

### Datalog

从前面的讨论很明确地看到，演绎 DBMS 的大部分内容表现为一种语言，在这一语言里，演绎公理（通常叫**规则**）可以公式化。这样的语言最著名的例子就是 **Datalog** 语言（和 Prolog 类似）（参看 [24.5]）。这一节里我们对 Datalog 语言作简单的介绍。*注意*：Datalog 语言的重点是它的描述能力，而不是它的计算能力（实际上，它与最初关系模型情况类似，参看 [6.1]））。其目标就是定义一种语言，使它最终能比传统的关系语言有更大的表达能力（参看 [24.5]）。所以，Datalog 中强调的（的确，一般地讲，这一强调贯穿于整个基于逻辑的系统中）是对查询的处理，而不是对查询的更新，尽管这是可能的，并也要求扩展这门语言去支持更新（以后将看到）。

Datalog 语言用它最简单的形式，对于那些像简单的 Horn 子句且没有函数的规则，作了公式化的支持。在 24.4 节里，我们把 Horn 子句定义为下述形式的合式公式：

```
A1 AND A2 AND ... AND An

A1 AND A2 AND ... AND  An  ⇒  B
```

（这里的 *A* 和 *B* 是仅包含常量和变量的谓词的实例）。然而，根据 Prolog 的格式，实际上，Datalog 将这第二种形式变形，即：

```
B  ⇐  A1 AND A2 AND ... AND An
```

为了和此领域的其他版本保持一致，下面我们作同样处理。

在这样一个子句中，*B* 是**规则头**（或结论），*A* 是**规则体**（或前提或**目标**，每一个个体是子目标）。为简洁起见，这里的 AND 通常都用逗号代替。一个 **Datalog 程序**是由某些传统的方式分割的一系列子句，例如，可由分号隔开（本书里不使用分号，而是简单地将每一个子句写在单独的一行）。在这样的程序里，子句的顺序是无所谓的。

从上述所讲的意义看，整个“演绎数据库”都可看作 Datalog 程序。例如，我们可以把上述所有的关于供应商和零件数据库的公理（基本公理、完整性约束、演绎公理）用 Datalog 的形式写出来，用分号把它们分开，或把它们写在单独的行上，这样，其结果就是 Datalog 程序。然而，正如早些时候所说的那样，数据库的扩展部分不能用这种方式详细说明，但能用别的更传统的方法。这样，Datalog 语言的主要目标就是明确地支持演绎公理的公式化。前面曾指出过，函数可以看成传统的关系 DBMS 中视图定义机制的扩展。

Datalog 语言也可用作查询语言（这里，也很像 Prolog）。例如，假设我们对 GOOD_SUPPLIER 作出 Datalog 的定义，如下：

```
GOOD_SUPPLIER ( s, st, sc )  ⇐  S ( s, sn, st, sc )
                                AND st > 15
```

下面是有关 GOOD_SUPPLIER 的典型的查询：

1）查询所有“好”供应商：

```
? ⇐ GOOD_SUPPLIER ( s, st, sc )
```

2）查询在巴黎的“好”供应商：

```
? ⇐ GOOD_SUPPLIER ( s, st, Paris )
```

3）供应商 S1 是“好”供应商吗？

```
? ⇐ GOOD_SUPPLIER ( S1, st, sc )
```

等等。或者说，Datalog 查询由一个以“?”为头的特殊的规则组成，其规则体由一个唯一的项组成，从此项可得查询结果。“?”按其习惯用法表示“显示”（display）。

应当指出，尽管最初定义的 Datalog 语言支持递归，但它还有许多传统的关系数据库语言的特征不支持，如：标量运算（“+”、“*”等）、聚集运算（COUNT、SUM 等）、差运算（因为子句不能被否定）、分组运算和不分组运算（ungrouping）等。它还不支持属性命名（谓词变元依赖于它初始位置），不支持全域（即第 5 章讲的用户自定义类型）。本节早些时候说过，它不提供任何更新操作，（由此造成的结果）也不支持定义过的外码删除及更新规则的说明（ON DELETE CASEADE 等等）。

为了弥补上述这些不足，已经对 Datalog 语言提出了各种各样的扩展，这些扩展试图提供下述特征：

- 否定的前提。例如：

```
SS_COLOCATED ( sx, sy ) ⇐ S ( sx, sxn, sxt, sc ) AND
                          S ( sy, syn, syt, sc ) AND
                          NOT ( sx = sy )
```

- 标量操作（固有的或用户自定义的）。例如：

```
P_WT_IN_GRAMS ( p, pn, pl, pg, pc ) ⇐
                P ( p, pn, pl, pw, pc ) AND pg = pw * 454
```

此例中，我们假设符号“*”用传统的记法写在中间。此项也可运用 AND 的更传统的前缀逻辑表示法为“=（$pg$，*（$pw$，454））”。

- 分组和聚集操作（与关系的 SUMMARIZE 操作相似，参看第 7 章）：有时为了叙述总量的问题，这样的操作是必需的；这些问题查询的不仅仅是零件 $px$ 由哪些零件 $py$ 组成，而且要查询组成零件 $px$ 需要多少个零件 $py$。（这里我们假设关系变量 PART_STRUCTURE 包括 QTY 属性）。
- 更新操作：解决更新要求的一种方法（不唯一）是基于对基本的 Datalog 语言的观察，（a）在规则头里面的任何谓词必须是非否定的；（b）规则生成的每一个元组可看作是在结果中的“插入”。这样，一个可能的扩充是允许在规则头里面有否定的谓词，且认为这些谓词是请求删除（有关的元组）。
- 规则体里的非 Horn 子句。或者说在规则的定义中允许有完全通用的合式公式。

Gardarin 和 Valduriez 在［24.6］中举例综述了以上的扩展内容，同时还讨论了各种各样的 Datalog 语言的实现技术。

## 24.7　递归查询过程

正如前一节里所述的，演绎数据库里最值得注意的特征是它对递归的支持。这里的递归包括递归规则的定义及由此决定的递归查询。所以，最近的几年对这样的递归的实现技术有着大量的研究，本节中我们简单地讨论了一些研究。

下面我们举例说明。仍以 24.6 节 COMPONENT_OF 的递归定义来说，它是由 PART_STRUCTURE 定义的。（为简单起见，我们把 COMPONENT_OF 简写为 COMP，把 PART_STRUC-

TURE 简写为 PS;同时,把这一定义转化为 Datalog 的形式,如下):

```
COMP ( px, py ) ⇐ PS ( px, py )
COMP ( px, py ) ⇐ PS ( px, pz ) AND COMP ( pz, py )
```

下面是基于这一数据库的典型的递归查询（“探寻零件 P1”）:

```
? ⇐ COMP ( P1, py )
```

现在回过头来看一看刚才的定义：该定义中的第二个规则，即递归规则，是**线性**递归，因为规则头里面的谓词在规则体里面仅出现一次。相比之下，下面 COMP 的定义中的第二个规则（递归的）从同一种意义上讲并不是线性递归的:

```
COMP ( px, py ) ⇐ PS ( px, py )
COMP ( px, py ) ⇐ COMP ( px, pz ) AND COMP ( pz, py )
```

然而，一般情况下总是认为线性递归表示着“令人感兴趣的情形”。某种意义上讲，在实际中出现的大多数递归都呈线性特征，而且对线性递归已经存在有效处理技术 [24.11]。因此，本节的讨论只限于线性递归的范畴。

*注意*：为了叙述的完整性，有必要对递归规则（包括线性递归）的定义加以概括，以便处理如下列出的更为复杂的情形：

```
P ( x, y ) ⇐ Q ( x, z ) AND R ( z, y )
Q ( x, y ) ⇐ P ( x, z ) AND S ( z, y )
```

为了简单起见，这里我们忽略了细节；具体详细的讨论参见 [24.11]。

在经典的查询过程（即非递归的）中，实现一个递归查询的全部问题被分解为两个子问题，即 a）把初始查询转化为一些等价的且更有效的形式，接着 b）实际执行这一转化结果。本书描述了对这两个问题的不同解法。本节里，我们粗略地讨论这些简单的技术，显示了这些技术在下述样本数据上对查询“探寻零件 P1”的求解过程:

PS

| PX | PY |
|---|---|
| P1 | P2 |
| P1 | P3 |
| P2 | P3 |
| P2 | P4 |
| P3 | P5 |
| P4 | P5 |
| P5 | P6 |

### 1. 合一与归结

当然，一个可能的方法是使用标准的 Prolog 技术，这就是在 24.4 节描述的**合一**与**归结**技术。此例中，这种方法的过程如下所示。第一个前提是演绎公理，在合取范式中它看起来如下所示:

```
1) NOT PS ( px, py ) OR COMP ( px, py )
2) NOT PS ( px, pz ) OR NOT COMP ( pz, py ) OR COMP ( px, py )
```

依据所期望的结论，我们再构造另外一个前提:

```
3) NOT COMP ( P1, py ) OR RESULT ( py )
```

基本公理构成了余下的前提。例如，考虑基本公理:

```
4) PS ( P1, P2 )
```

用 P1 取代上述 1 中的 *px*，P2 取代上述 1 中的 *py*，我们可以归结上面的 1 和 4，得:

5) COMP ( P1, P2 )

现在用 P2 取代 3 中的 *py*，并归结 3 和 5，得：

6) RESULT ( P2 )

因此，P2 是 P1 的组件。类似地，P3 也是 P1 的组件。由此得到公理 COMP（P1，P2）和 COMP（P1，P3）；递归运用上述过程，并确定最终结果。具体细节留做练习。

实际中，合一与归结的处理代价是大的，因此需要寻找一些更有效的处理策略。下一小节讨论对这一问题的一些可能的方法。

### 2. 朴质计算

**朴质计算**（naive evaluation）（参见［24.20］）可能是所有方法中最简单的方法，正如它的名字所说的那样，这一算法是非常纯朴的。仍以前述简单查询为例，用下述伪代码可以很容易地解释清楚这一过程：

```
COMP := PS ;
do until COMP reaches a "fixpoint" ;
  COMP := COMP UNION ( COMP ⋈ PS ) ;
end ;
DISPLAY := COMP WHERE PX = P# ('P1') ;
```

对于关系变量 COMP 和 DISPLAY（类似关系变量 PS），均有两个属性，即 PX 和 PY。不严格地讲，这种方法就是重复地组合中间结果，直到它达到一个不动点——即它停止生长。这一中间结果是：PS 与前面中间结果的连接的并。注意：表达式“COMP PS”是“COMP 和 PS 在 COMP. PX 和 COMP. PY 上进行连接，然后再在其上进行投影”的缩写；为简单起见，我们忽略了属性重命名操作，而这在代数里是不能忽略的（参看第 7 章）。

现在用实际数据来看看这一算法的过程。经过第一次循环迭代之后，表达式 COMP ⋈ PS 的值如下左表所示，COMP 的结果如下右表所示（此迭代过程中增加的元组用星号作标记）：

COMP ⋈ PS

| PX | PY |
|---|---|
| P1 | P3 |
| P1 | P4 |
| P1 | P5 |
| P2 | P5 |
| P3 | P6 |
| P4 | P6 |

COMP

| PX | PY | |
|---|---|---|
| P1 | P2 | |
| P1 | P3 | |
| P2 | P3 | |
| P2 | P4 | |
| P3 | P5 | |
| P4 | P5 | |
| P5 | P6 | |
| P1 | P4 | * |
| P1 | P5 | * |
| P2 | P5 | * |
| P3 | P6 | * |
| P4 | P6 | * |

第二次迭代之后，其结果如下：

COMP ⋈ PS

| PX | PY |
|---|---|
| P1 | P3 |
| P1 | P4 |
| P1 | P5 |
| P2 | P5 |
| P3 | P6 |
| P4 | P6 |
| P1 | P6 |
| P2 | P6 |

COMP

| PX | PY | |
|---|---|---|
| P1 | P2 | |
| P1 | P3 | |
| P2 | P3 | |
| P2 | P4 | |
| P3 | P5 | |
| P4 | P5 | |
| P5 | P6 | |
| P1 | P4 | |
| P1 | P5 | |
| P2 | P5 | |
| P3 | P6 | |
| P4 | P6 | |
| P1 | P6 | * |
| P2 | P6 | * |

看一看第二次迭代计算 COMP ⋈ PS，它重复利用了第一次迭中 COMP ⋈ PS 的计算结果，且又增加了两个另外的元组（即上表中的（P1，P6）和（P2，P6））。这就是为什么朴质算法并不是很智能的原因。

第三次迭代及更多的重复计算之后，COMP ⋈ PS 的值不再发生变化，COMP 达到一不动点，这时我们退出循环。最后的计算结果就是 COMP 的限制子集：

COMP

| PX | PY |
|---|---|
| P1 | P2 |
| P1 | P3 |
| P1 | P4 |
| P1 | P5 |
| P1 | P6 |

这里有一显著的副作用：这一算法有效地计算了对每一个零件的探寻。实际上，它已经计算了关系 PS 的整个传递闭包，但是只保留了所需要的元组，其他元组都抛弃掉了，所以白白地做了许多工作。

朴质计算技术被看作是前向推理的应用：从外延数据库（即实际的数据值）开始，它重复运用定义的前提（即规则体）直到达到所要的结果。事实上，这种算法计算了 Datalog 程序中的最小模型（参看 [24.5，24.6]）。

**3. 半朴质计算**

前面讲到，朴质计算中，每一步的计算在下一步里都要重复，这没有必要。**半朴质计算**就是对这种重复的一大改进（参看 [24.23]）。或者说，每一步里，我们仅计算在这一迭代中需要添加的新元组。还用“探寻零件 P1”的例子来解释这个思想。伪代码是：

```
NEW  := PS ;
COMP := NEW ;
do until NEW is empty ;
   NEW  := ( NEW ⋈ PS ) MINUS COMP ;
   COMP := COMP UNION NEW ;
end ;
DISPLAY := COMP WHERE PX = P# ('P1') ;
```

现在再来看看这一算法。当循环开始，对于 PS，NEW 和 COMP 是一样的：

NEW

| PX | PY |
|---|---|
| P1 | P2 |
| P1 | P3 |
| P2 | P3 |
| P2 | P4 |
| P3 | P5 |
| P4 | P5 |
| P5 | P6 |

COMP

| PX | PY |
|---|---|
| P1 | P2 |
| P1 | P3 |
| P2 | P3 |
| P2 | P4 |
| P3 | P5 |
| P4 | P5 |
| P5 | P6 |

第一次迭代之后，结果如下所示：

NEW

| PX | PY |
|---|---|
| P1 | P4 |
| P1 | P5 |
| P2 | P5 |
| P3 | P6 |
| P4 | P6 |

COMP

| PX | PY | |
|---|---|---|
| P1 | P2 | |
| P1 | P3 | |
| P2 | P3 | |
| P2 | P4 | |
| P3 | P5 | |
| P4 | P5 | |
| P5 | P6 | |
| P1 | P4 | * |
| P1 | P5 | * |
| P2 | P5 | * |
| P3 | P6 | * |
| P4 | P6 | * |

COMP 与朴质计算中的第一次迭代的结果是一样的，而 NEW 仅仅是这一迭代中 COMP 增加的新元组。要特别注意，NEW 不包括（P1，P3）（与朴质计算中这一步比较）。

第二次迭代结束，结果为：

NEW

| PX | PY |
|---|---|
| P1 | P6 |
| P2 | P6 |

COMP

| PX | PY | |
|---|---|---|
| P1 | P2 | |
| P1 | P3 | |
| P2 | P3 | |
| P2 | P4 | |
| P3 | P5 | |
| P4 | P5 | |
| P5 | P6 | |
| P1 | P4 | |
| P1 | P5 | |
| P2 | P5 | |
| P3 | P6 | |
| P4 | P6 | |
| P1 | P6 | * |
| P2 | P6 | * |

再进行下一次迭代，NEW 为空，退出循环。

**4. 静态筛选**

**静态筛选**（static filtering）是一精细化过程，它来自于经典优化理论中尽可能早地执行选择操作这一基本思想。它是后向推理的一种应用，因为它有效地使用了查询结果信息来修改规则（前提）。同时，它也简化相关事实，因为它利用查询结果信息来删除起初的外延数据库中的无用的元组（参看［24.24］）。用例子的伪码解释这一结果如下：

```
NEW  := PS WHERE PX = P# ('P1') ;
COMP := NEW ;
do until NEW is empty ;
   NEW  := ( NEW ⋉ PS ) MINUS COMP ;
   COMP := COMP UNION NEW ;
end ;
DISPLAY := COMP ;
```

再来看看这一算法过程。循环开始，对于 PS，NEW 和 COMP 如下所示：

NEW

| PX | PY |
|---|---|
| P1 | P2 |
| P1 | P3 |

COMP

| PX | PY |
|---|---|
| P1 | P2 |
| P1 | P3 |

第一次迭代之后，结果如下所示：

NEW

| PX | PY |
|---|---|
| P1 | P4 |
| P1 | P5 |

COMP

| PX | PY | |
|---|---|---|
| P1 | P2 | |
| P1 | P3 | |
| P1 | P4 | * |
| P1 | P5 | * |

下一次迭代结束，结果为：

NEW

| PX | PY |
|---|---|
| P1 | P6 |

COMP

| PX | PY | |
|---|---|---|
| P1 | P2 | |
| P1 | P3 | |
| P1 | P4 | |
| P1 | P5 | |
| P1 | P6 | * |

再进行下一次迭代，NEW 为空，退出循环。

对递归查询策略，我们只简单作以上的介绍。当然，在这一领域内还有许多其他的方法，它们比上面的简单方法要成熟得多。但由于空间有限，我们就不再介绍，文献［24.11～24.25］中有更多的介绍。

## 24.8　小结

下面对基于逻辑的数据库作一个小结。虽然这一思想在研究领域还受到很大限制，但还是有基于它的商业关系产品上市（尤其对某些优化技术）。总的来讲，基于逻辑的数据库的概念还是令人感兴趣的，前面一节里已经讲了它的潜在的优点。还有一个重要的优点，本章里没有讨论，就是逻辑为通用程序语言和数据库的无缝集成提供了基础。或者说，这一系统提供了唯一的基于逻辑的语言，在这一语言里，“数据就是数据”，而不管它是在共享数据库中还是在本地库中。正是这一语言取代了今天的 SQL 产品中并不完美的“嵌入数据子语言”的方法（当然，在达到这样的目标之前，还必须克服大量的困难。首先，逻辑是通用程序语言的合适的基础，而目前所取得的进展并不能很好地满足这一需要）。

现在再来简略地回顾一下我们所讲的主要部分。首先，简单地介绍了**命题演算**和**谓词演算**，引入了下面的概念：

- 一个组合式公式的**释义**是下面三者的组合：（a）论域；（b）合式公式中单独的常量与此论域中的对应关系；（c）合式公式中定义的谓词和函数的意义。
- 一个组合式公式的**模型**就是一个释义，在此释义中，所有合式公式为真。一般情况下，一组给定的合式公式有许多模型。
- 证明就是这样的一个过程，它表明某一合式公式 *g*（**结论**）是另一组合式公式 *f1*，*f2*，…，*fn*（**前提**）的逻辑结果。我们稍微具体地讨论了一个证明方法，即**归结与合一**。

接着，我们讨论了数据库的**证明理论**的观点。这一观点认为，数据库是由**外延数据库**和**内涵数据库**组成。外延数据库包含**基本公理**，不严格地讲，就是基本数据；内涵数据库包含完整性约束和**演绎公理**，不严格地讲就是视图。数据库的“含义”在于由一组从公理演绎来的定理组成；执行一个查询就成了**定理证明**过程（至少概念上是这样的）。**演绎 DBMS** 就是这种支持证明理论观点的 DBMS。我们简单地介绍了这种 DBMS 的一种语言，即 **Datalog** 语言。

Datalog 语言与传统的关系语言之间最直接最明显的区别在于它支持**递归**公理，因而支持递归查询。而目前还不知道为什么传统的关系演算和关系代数不能扩展去做同样的功能（参看第 7 章讨论的 TCLOSE 操作）[⊖]。我们还讨论了计算这一查询的简单技术。

结论：本章开始时我们讲了许多术语，如：逻辑数据库、推理 DBMS、演绎 DBMS，等等。这些术语在研究领域经常用到（在一定程度上，甚至在厂家的广告中也会出现）。我们对它们作了一些解释。然而，在这些问题上，总是没有一致性。在研究领域，对同一术语可能会出现不同解释。下面的一些解释都是可能会出现的：

- *递归查询过程*：这是很简单的概念。它是一查询的计算方法，尤其是优化。它内在地定义了递归（参看 24.7 节）。
- *知识库*：此术语有时就指 24.6 节讲的内涵数据库，即它由一系列规则（完整性约束和演绎公理）组成。它和库数据相反，库数据是由一系列外延数据库组成。但是有些观点认为它是这两者的组合（下面的演算数据库将讲到），而且还有人认为（参看 [24.6]）“知识库经常包括复杂的对象和经典关系”（参看本书第六部分“复杂对象的讨论”）。在自然语言系统里，这一术语还有更多的意义。最好能完全避免这一术语的使用。
- *知识*：这也是简单的概念！知识就是知识库里的内容。这一定义简化了前面未能解决的“知识”定义的问题。
- *知识库管理系统*（KBMS）：管理知识库的软件。这一术语典型地作为演绎 DBMS 的同义词使用。
- *演绎* **DBMS**：支持证明理论观点的数据库，尤其是它能根据内涵数据库中的推理（演绎）

⊖ 就这一点来说，关系 DBMS 需要能够解决递归过程，因为在这里有些递归地包含了结构化的信息（根据其他的视图定义的视图就是一个典型的例子）。

规则，从外延数据库演绎出其他的信息。演绎 DBMS 一定能支持递归规则，从而支持递归查询。

- *演绎数据库*（Deductive database[㊀]）：由演绎 DBMS 管理的数据库。
- *专家 DBMS*：演绎数据库的同义词。
- *专家数据库*（Expert databasee[㊁]）：由专家 DBMS 管理的数据库。
- *推理 DBMS*：演绎数据库的同义词。
- *基于逻辑的系统*：演绎数据库的同义词。
- *逻辑数据库*（Logic database[㊂]）：演绎数据库的同义词。
- *逻辑作为数据模型*：至少由对象、完整性规则和操作组成的数据模型。在演绎 DBMS 中，对象、完整性规则和操作都是以统一的方式描述，如在 Datalog 这一逻辑语言中都描述为公理。正如 24.6 节所解释的那样，此系统的一个数据库可以看作一种逻辑程序，包含了所有的三种公理。因此，在这样的一个系统中，抽象数据模型就是其逻辑本身。

## 习题

24.1　试使用归结方法，看下面的中间结果在命题演算中是否构成合法证据。

a. `A ⇒ B, C ⇒ B, D ⇒ ( A OR C ), D ⊢ B`

b. `( A ⇒ B ) AND ( C ⇒ D ), ( B ⇒ E AND D ⇒ F ), NOT ( E AND F ), A ⇒ C ⊢ NOT A`

c. `( A OR B ) ⇒ D, D ⇒ NOT ( E OR F ), NOT ( B AND C AND E ) ⊢ NOT ( G ⇒ NOT ( C AND H ) )`

24.2　把下面的合式公式转化为子句形式：

a.
```
FORALL x ( FORALL y
           ( p ( x, y ) ⇒ EXISTS z ( q ( x, z ) ) ) )
```

b.
```
EXISTS x ( EXISTS y
           ( p ( x, y ) ⇒ FORALL z ( q ( x, z ) ) ) )
```

c.
```
EXISTS x ( EXISTS y
           ( p ( x, y ) ⇒ EXISTS z ( q ( x, z ) ) ) )
```

24.3　下面是一个逻辑数据库的标准的例子：

```
MAN    ( Adam )
WOMAN  ( Eve )
MAN    ( Cain )
MAN    ( Abel )
MAN    ( Enoch )

PARENT ( Adam, Cain )
PARENT ( Adam, Abel )
PARENT ( Eve, Cain )
PARENT ( Eve, Abel )
PARENT ( Cain, Enoch )

FATHER ( x, y )    ⇐  PARENT ( x, y ) AND MAN ( x )
MOTHER ( x, y )    ⇐  PARENT ( x, y ) AND WOMAN ( x )

SIBLING ( x, y )   ⇐  PARENT ( z, x ) AND PARENT ( z, y )

BROTHER ( x, y )   ⇐  SIBLING ( x, y ) AND MAN ( x )

SISTER ( x, y )    ⇐  SIBLING ( x, y ) AND WOMAN ( x )

ANCESTOR ( x, y )  ⇐  PARENT ( x, y )
ANCESTOR ( x, y )  ⇐  PARENT ( x, z ) AND ANCESTOR ( z, y )
```

---

㊀ 这是一个有争议的术语。——译者注

㊁ 这是一个有争议的术语。——译者注

㊂ 这是一个有争议的术语。——译者注

使用归结方法回答下列问题：

a. Cain 的母亲是谁？

b. Cain 的兄妹是谁？

c. Cain 的兄弟是谁？

d. Cain 的姐妹是谁？

e. Enoch 的祖先是谁？

24.4　解释什么是释义和模型。

24.5　写一组 Datalog 公理来解释供应商 - 零件 - 工程数据库的部分内容。

24.6　试写出习题 7.13 ~ 7.50 的 Datalog 语句。

24.7　试写出习题 9.3 的 Datalog 语句。

24.8　24.7 节中用归结与合一的方法实现了“探寻零件 P1”的查询，请给出你自己的解释。

## 参考文献

基于逻辑的数据库系统在过去几年迅速成长；下面只列出了当前这一研究领域的部分成果，并分组如下：

- 文献［24.1 ~ 24.5］是一些书籍，它们讲述了逻辑方面的研究（尤其是在计算和/或数据库方面）或者收集了基于逻辑的数据库的大量的论文。
- 文献［24.6］和［24.7］是一些教程。
- 文献［24.9］、［24.12 ~ 24.15］、［24.25］和［24.27，24.28］讲述了传递闭包及其实现。
- 文献［24.16 ~ 24.19］描述了一个重要的叫做魔集（及其上变化）的递归查询技术。也可参看［18.22 ~ 18.24］。

剩余的文献主要讲述了在这一领域内有多少研究、涉及哪些方向，这里只作了简单的介绍。


［24.1］　Robert R. Stoll: *Sets, Logic, and Axiomatic Theories.* San Francisco, Calif.: W. H. Freeman and Company (1961).

本书很好地介绍了什么叫逻辑。

［24.2］　Peter M. D. Gray: *Logic, Algebra and Databases.* Chichester, England: Ellis Horwood Ltd. (1984).

本书从数据库的观点比较详细地介绍了命题演算和谓词演算及其他许多相关命题。

［24.3］　Hervé Gallaire and Jack Minker: *Logic and Data Bases.* New York, N. Y.: Plenum Publishing Corp. (1978).

本书是早期收集了本领域内的大量的论文的书籍。

［24.4］　Jack Minker (ed.): *Foundations of Deductive Databases and Logic Programming.* San Mateo, Calif.: Morgan Kaufmann (1988).

［24.5］　Jeffrey D. Ullman: *Database and Knowledge-Base Systems* (in two volumes). Rockville, Md.: Computer Science Press (1988, 1989).

该书第一卷共有 10 章，其中有一章详细介绍了基于逻辑的方法。该章还讨论了 Datalog 语言的起源，但重要的是介绍了逻辑与关系代数之间的关系。同时作为逻辑方法的一个特殊的例子，介绍了关系演算（包括域演算和元组演算）。第二卷共有 7 章，其中有 5 章讨论了基于逻辑的数据库。

［24.6］　Georges Gardarin and Patrick Valduriez: *Relational Databases and Knowledge Bases.* Reading, Mass.: Addison-Wesley (1989).

该书有一章介绍了演算系统，它讨论了基本的理论及优化算法等。虽然在实际中它只是一本指南，但比我们这一章介绍的要详细。

［24.7］　Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas: “Logic and Databases: A Deductive Approach,” *ACM Comp. Surv. 16*, No. 2 (June 1984).

［24.8］　Veronica Dahl: “On Database Systems Development Through Logic,” *ACM TODS 7*, No. 1 (March 1982).

本文对基于逻辑的数据库的基本思想作了详细的描述，还列举了基于 Prolog 原型的一些例子。这些例子是由 Dahl 在 1977 年实现的。

［24.9］　Rakesh Agrawal: “Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries,” *IEEE Transactions on Software Engineering 14*, No. 7 (July 1988).

本文提出了一种新的叫做 alpha 的操作，该操作支持“递归查询巨类”的问题（实际上，这是一个递归查询的超类），它建立在传统的关系代数的基础上的。在处理大部分包括递归的实际问题时它具有强大的功能，同时它比任何其他的递归机制更容易实现。这篇论文列举了此操作符的几个例子。特别地，它还讲了如何简单地处理传递闭包和“总量查询的问题”（分别参看［24.12］和24.6节）。

文献［24.14］和［24.13］也讨论了一些相关的实现问题。

［24.10］ Raymond Reiter:“Towards a Logical Reconstruction of Relational Database Theory,” in Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt (eds.), *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York, N. Y.: Springer-Verlag (1984).

24.2节已经讲过，Reiter 著作并不是这一领域内的第一本文献，以前已有许多研究者研究了逻辑和数据库之间的关系（参看文献［24.3］、［24.4］和［24.8］）。但是，好像是“Reiter 的关系理论的逻辑重构”激起了这一领域的发展和当前人们极大的兴趣。

［24.11］ François Bancilhon and Raghu Ramakrishnan:“An Amateur's Introduction to Recursive Query Processing Strategies,” Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, D. C. (May 1986). Republished in revised form in Michael Stonebraker (ed.), *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann (1988).

这是一篇优秀的综合论文。开篇介绍了在递归查询的实现中存在的消极和积极的问题。积极的是已经实现了的大量技术，至少它解决了这一查询的实现。消极的是，还根本不知道如何在给定情形下选择最适合的技术（特别是这里的许多论文很少或根本就没有讨论实现上的特征）。在用一小节讨论了逻辑数据库的基本思想后，继续讨论了大量命题的算法，如朴质计算、半朴质计算、迭代查询/子查询、递归查询/子查询、APEX、Prolog 语言、Henschen/Naqvi、Aho-Ullman、Kifer-Lozinskii、计数（counting）、魔集等、广义魔集等。它还在应用领域（即这一类算法通常应用的问题）、性能和实现的简易性方面上对这些不同的方法进行了比较。论文还包括了来自一个简单的基准测试的性能参数，以及相应的比较分析。

［24.12］ Yannis E. Ioannidis:“On the Computation of the Transitive Closure of Relational Operators,” Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan (August 1986).

本文提出了分而治之的方法来实现传递闭包，还可以参看文献［24.9］、［24.13～24.15］和［24.27，24.28］。

［24.13］ H. V. Jagadish, Rakesh Agrawal, and Linda Ness:“A Study of Transitive Closure as a Recursion Mechanism,” Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

文章摘要中提到“此文表明，每一个递归查询都可表示为一个传递闭包，其后可能紧跟在关系代数里也适用的操作”。作者认为，在一般情况下，有效地实现传递闭包是有效地实现线性递归的充分基础，所以也是在递归巨类上有效地实现演绎 DBMS 的基础。

［24.14］ Rakesh Agrawal and H. Jagadish:“Direct Algorithms for Computing the Transitive Closure of Database Relations,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

此文指出了一组传递闭包的算法，文章“没有把传递闭包问题看作是计算递归的问题，而是看作从第一个原理获得闭包的问题”（所以这一术语是直接的）。本文也对早期的其他的直接算法进行了小结。

［24.15］ Hongjun Lu:“New Strategies for Computing the Transitive Closure of a Database Relation,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

此文讲述更多的传递闭包的算法。与［24.14］一样，该文也对早期的解决这一方面问题的方法进行了总结。

［24.16］ Fransois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman:“Magic Sets and Other Strange Ways to Implement Logic Programs.” Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (1986).

“魔集”的基本思想是引入新的规则（“魔规则”）来保证产生与初始查询相同的结果，但使用魔规则更有效。从这个意义上讲，它们简化了一系列“相关事实”（参看24.7节）。具体内容很复杂，且超出了此注解的范围。至于详细解释，请参考此论文或 Bancilhon 或 Ramakrishnan

的报告［24.11］，或 Ullman 的书［24.5］，或 Gardarin 和 Valduriez 的书［24.6］。基于此思想的许多不同变体相继出台，如下面的文献［24.17～24.19］，同时还可参看［18.22～18.24］。

［24.17］ Catriel Beeri and Raghu Ramakrishnan:"On the Power of Magic,"Proc. 6th ACM SIGMOD- SIGACT Symposium on Principles of Database Systems (1987).

［24.18］ Domenico Saccà"and Carlo Zaniolo:"Magic Counting Methods,"Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data,San Francisco,Calif. (May 1987).

［24.19］ Georges Gardarin:"Magic Functions:A Technique to Optimize Extended Datalog Recursive Programs," Proc. 13th Int. Conf. on Very Large Data Bases,Brighton,UK (September 1987).

［24.20］ A. Aho and J. D. Ullman:"Universality of Data Retrieval Languages,"Proc. 6th ACM Symposium on Principles of Programming Languages,San Antonio,Tex. (January 1979).

假如有关系 $r$、$f(r)$、$f(f(r))$，…（这里 $f$ 为固定的函数），则根据下面的朴质算法，其**最小不动点**定义为关系 $r^*$（参看 24.7 节）：

```
r* := r ;
do until r* stops growing ;
   r* := r* UNION f(r*) ;
end ;
```

此文指出了另一个关系代数的最小不动点操作。

［24.21］ Jeffrey D. Ullman:"Implementation of Logical Query Languages for Databases,"*ACM TODS 10*,No. 3 (September 1985).

本文讲述了一个重要的类，它包括可能的递归查询实现技术。这些技术依据"规则/目标树"的"捕获"规则而定义。它们也是依据子句和谓词而表达查询策略图。这篇论文定义了几个这样的规则。一个是讲关系代数操作的应用，两个分别讲前向链和后向链，还有一个"小路"规则，它允许结果从一个子目标传向另一个子目标。传向后者的"小路"信息是所谓的魔集技术的基础（参看［24.16～24.19］）。

［24.22］ Shalom Tsur and Carlo Zaniolo:"LDL:A Logic-Based Data- Language,"Proc. 12th Int. Conf. on Very Large Data Bases,Kyoto,Japan (August 1986).

LDL 包括：(1) 一个"集"类型构造器；(2) 非（基于差）；(3) 数据定义；(4) 更新操作。它是一种纯的逻辑语言，语句间没有序的依赖关系。这种语言是编译型的，而不是解释型的。

［24.23］ François Bancilhon:"Naive Evaluation of Recursively Defined Relations,"in Michael Brodie and John Mylopoulos (eds.), *On Knowledge Base Management Systems: Integrating Database and AI Systems*. New York,N. Y.:Springer-Verlag (1986).

［24.24］ Eliezer L. Lozinskii:"A Problem-Oriented Inferential Database System,"*ACM TODS 11*,No. 3 (September 1986).

此文讲述了"相关事实"的来源。由于推理技术导致这一研究领域其他方面的快速扩充，这篇文章就讲述了一个原型系统去抑制这种扩充。

［24.25］ Arnon Rosenthal *et al.*:"Traversal Recursion:A Practical Approach to Supporting Recursive Applications," Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data,Washington,D. C (June 1986).

［24.26］ Michael Kifer and Eliezer Lozinskii:"On Compile Time Query Optimization in Deductive Databases by Means of Static Filtering,"*ACM TODS 15*,No. 3 (September 1990).

［24.27］ Rakesh Agrawal,Shaul Dar,and H. V. Jagadish:"Direct Transitive Closure Algorithms:Design and Performance Evaluation,"*ACM TODS 15*,No. 3 (September 1990).

［24.28］ H. V. Jagadish:"A Compression Method to Materialize Transitive Closure,"*ACM TODS 15*,No. 4 (December 1990).

本文提出了一个索引技术，从而可使给定关系的传递闭包压缩存储。这样，要测试一个元组是否在闭包里出现，就可以通过一个单表查询辅以索引比较。

# 第六部分　对象、关系和 XML

注意：像第 20 章一样，本书这部分的几章内容也在很大程度上依赖于最开始在第 5 章讨论到的内容。如果前面你只是马马虎虎地浏览了一遍，那么你可能在读这几章之前要返回去重读一遍（如果你还没有这么做的话）。

一般而言，对象技术在软件开发领域中是一门重要的学科，人们自然而然也会问它是否与数据库管理的领域相关，更重要的是，如果相关，那么关系是什么。虽然人们通常难以就这些问题达成一致，但某种统一的认识确实正在出现。当对象数据库系统刚出现的时候，一些工业名流声称它们可以控制世界，完全取代关系数据库系统；而一些权威人士则觉得对象数据库只适用于某些特定的问题，不可能占领太多的市场。当争辩如火如荼的时候，系统支持的第三种可能开始出现：这是一种结合了对象和关系的技术并且青出于蓝而胜于蓝的系统。现在它似乎证明那些权威人士的观点是正确的：单纯的对象系统只能充当一种角色，但是只是小角色，而关系系统在可以预见的将来会继续统领市场。正如我们看到的一样，不仅是因为那些“对象/关系”系统实际也是关系系统。

目前，一种特定的对象日益被人们所关注，就是 XML 文档；而在数据库中保存、查询和更新它们已经迅速成为重大的课题。“XML 数据库”——一种只包含 XML 文档的数据库——是有可能的。然而，如有可能，在对象或者关系（或者“对象/关系”）数据库中使 XML 文档和其他数据结合成一体，会变得更适当。

本书这部分所包含的几章在一定深度上研究了这些问题。第 25 章研究了纯对象系统，第 26 章概述了对象/关系系统，第 27 章讨论了 XML。

# 第25章　对象数据库

## 25.1　引言

从20世纪80年代末到90年代中期，人们对面向对象数据库系统（简称**对象系统**）产生了极大的兴趣。一些人甚至认为对象系统是关系系统（或称为SQL系统）的强大竞争对手。然而今天很少有人同意这点；大多数IT业人士现在感到，对象系统确实发挥了一些作用，虽然，这种作用相对非常有限［25.33］。然而，这种系统仍然是值得研究的。因此，在本章中我们将详细讨论有关对象系统的内容，包括介绍和解释基本的对象概念，深入分析这些概念，并就如何恰当地把这些概念加入到将来的数据库系统中提出一些建议。

为什么如今大家都对对象系统感兴趣呢？因为“每一个人”都知道经典的SQL系统在某些方面存在不足，一些人甚至认为其基本的理论（即关系模型）也存在着问题。无论如何，我们在DBMS中所需要加入的一些新特性在C++或Smalltalk之类的对象编程语言中已经存在了许多年，所以有把这些特性融入到数据库系统的想法是一件很自然的事。许多研究人员和一些软件开发商也确实在这么做。

因此，对象系统源于对象编程语言。其基本思想是：用户不应该与面向机器的结构，如位或字节（甚至字段和记录）打交道，而应该直接对**对象**和建立在对象之上的**操作**进行处理，这样做显然与用户的真实世界更为一致。例如，用户应该能够直接认为存在一个部门对象，部门对象中包含一系列雇员对象，而不是考虑部门表中的元组及其相应的雇员表中的元组和外码值，并且这些值还必须参照部门表中的主码。当增加一个雇员时，用户也不需要再考虑部门表和雇员表之间主码和外码的参照完整性后再插入一条元组，而只要把雇员对象直接放到相关的部门对象中即可。换句话说，其基本思想是：**提高抽象度**。

如今毫无疑问，提高抽象度是一个有意义的目标，而对象模式在编程语言领域成功地满足了这一目标。自然，我们就会想到在数据库领域是否也能成功地使用这一模式。确实，用数据库直接处理“复杂对象”的思想（例如，拥有能直接雇佣雇员、更换经理和削减预算的部门对象）比起“关系变量”、“元组插入”和“外码”等传统的数据库处理来说，对用户更具有吸引力——至少乍看起来是这样。

然而，在这里我们应该提出一个谨慎的观点：虽然编程语言和数据库管理在原则上有许多共同之处，但它们在某些重要的方面确实有所不同，具体来讲：

- 应用程序按其定义是为了解决某些特定的问题；
- 数据库按其定义却是为了解决一系列不同的问题（其中一些问题甚至在数据库建立时并不清楚）。

所以，在应用编程环境中，把许多“智能”的东西嵌入到复杂的对象中是一个好主意：这样能够减少使用这些对象所需要编写的代码量、提高程序员的工作效率、增强程序的可维护性，等等。相反，在数据库环境中，嵌入许多智能的东西并不一定是一个好主意：它可能简化了一些问题，但同时也使另一些问题变得更加困难，甚至不能解决。

巧合的是，同样的观点曾在20世纪70年代时被用来反对关系数据库以前的数据库系统，如IMS。一个包含有一系列雇员对象的部门对象在概念上与IMS的层次结构非常相似。在层次结构中，部门“父片段”拥有一些从属的雇员“子片段”。这种层次结构对于类似“找出在会计部门工作的雇员”这样的问题非常适合，但却不能很好地回答“找出雇佣了MBA的部门”这样的问题。所以，在20世纪70年代用来反对层次方法的许多观点如今在对象这一环境下又以不同的方式再次被提出。

尽管有上述的种种问题存在，许多人仍然认为对象系统是数据库技术上的重大飞跃。特别是对于如下“复杂”应用，对象技术更是一个非常好的选择：

- 计算机辅助设计和制造（CAD/CAM）
- 计算机集成制造（CIM）
- 计算机辅助软件工程（CASE）
- 地理信息系统（GIS）
- 科学和医学
- 文档存储和获取

还有其他许多应用（注意在这些领域内使用经典的 SQL 产品的确会遇到一些麻烦）。当然，针对这一问题这些年来产生了许多技术论文，并且一些相关的商品化产品也已进入了市场。

本章的目的就是解释对象数据库技术究竟是什么？也就是介绍对象方法中一些最重要的概念，尤其是从数据库的角度对这些概念的看法（相反，许多文献都是从程序设计的角度来引入这些思想的）。本章的结构如下：在下一小节中将给出一个启发性的例子——一个传统的 SQL 产品无法进行有效处理而使用对象技术却能很好解决的例子。25.2 节将给出对于对象、类、消息和方法这些概念的概述，而在 25.3 节中将就这些概念的某些特定方面做深入的探讨。25.4 节给出了一个完整的例子，25.5 节则对一些不同性质的问题进行了讨论。25.6 节为本章的小结。

最后需要注意两点：

- 尽管对象系统是针对一些“复杂”应用，如 CAD/CAM，而发展起来的，但为了简化，我们的例子仍将基于一些非常简单的应用（如部门和雇员等）。当然，这一简化绝不会影响我们对对象数据库的探讨，因为如果对象技术真那么有效的话，它首先应该能处理那些“简单的”应用。
- 注意我们在本章所关注的是对象数据库系统，而不是对象编程或者对象编程语言，对象分析和设计，“对象建模”，图形对象接口等。最重要的是，我们并没有声明在数据库上下文中就对象所做的评论在其他上下文中也是有效的。

**一个启发性的例子**

本小节我们将给出一个简单的例子，这一例子来自 Stonebraker 并经过了笔者的修改[25.15]。这个例子主要为了说明传统的 SQL 产品所存在的一些问题。在此数据库中（可以认为是 CAD/CAM 数据库很大程度上的简化）涉及到对矩形的处理。为了简化起见，假设所有矩形的边都与 *X* 轴和 *Y* 轴或平行或垂直。这样，任何矩形就可以通过其左下和右上两个顶点坐标（*x1*，*y1*）和（*x2*，*y2*）唯一标识（如图 25-1 所示）。用 SQL 语句可表示为：

```
CREATE TABLE RECTANGLES
     ( X1 ... , Y1 ... , X2 ... , Y2 ... , ... ,
       PRIMARY KEY ( X1, Y1, X2, Y2 ) ) ;
```

现在考虑查询“找出所有与单位正方形（0，0，1，1）重叠的矩形”（如图 25-2 所示）。对此查询，一个“明显的”查询表达式为：

```
SELECT ...
FROM   RECTANGLES
WHERE  ( X1 >= 0 AND X1 <= 1 AND Y1 >= 0 AND Y1 <= 1 )
                  /* bottom left corner inside unit square  */
OR     ( X2 >= 0 AND X2 <= 1 AND Y2 >= 0 AND Y2 <= 1 )
                  /* top right corner inside unit square    */
OR     ( X1 >= 0 AND X1 <= 1 AND Y2 >= 0 AND Y2 <= 1 )
                  /* top left corner inside unit square     */
OR     ( X2 >= 0 AND X2 <= 1 AND Y1 >= 0 AND Y1 <= 1 )
                  /* bottom right corner inside unit square */
OR     ( X1 <= 0 AND X2 >= 1 AND Y1 <= 0 AND Y2 >= 1 )
                  /* rectangle totally includes unit square */
OR     ( X1 <= 0 AND X2 >= 1 AND Y1 >= 0 AND Y1 <= 1 )
                  /* bottom edge crosses unit square        */
OR     ( X1 >= 0 AND X1 <= 1 AND Y1 <= 0 AND Y2 >= 1 )
                  /* left edge crosses unit square          */
OR     ( X2 >= 0 AND X2 <= 1 AND Y1 <= 0 AND Y2 >= 1 )
                  /* right edge crosses unit square         */
OR     ( X1 <= 0 AND X2 >= 1 AND Y2 >= 0 AND Y2 <= 1 ) ;
                  /* top edge crosses unit square           */
```

（练习：证明这一写法是正确的。）

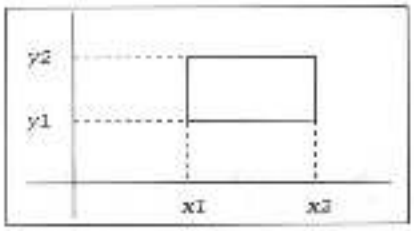



图 25-1　矩形（x1，y1，x2，y2）

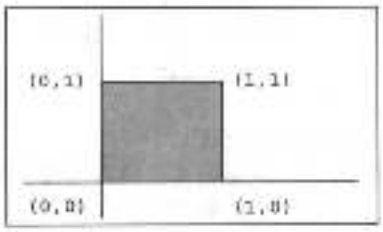



图 25-2　单位正方形（0，0，1，1）

然而，进一步思考会发现此查询可以被更简单地表示为：

```
SELECT ...
FROM   RECTANGLES
WHERE  ( X1 <= 1 AND Y1 <= 1
                /* bottom left corner is "downwind" of (1,1) */
AND      X2 >= 0 AND Y2 >= 0 ) ;
                /* top right corner is "upwind" of (0,0)     */
```

（本章后面的习题 25.3 要求证明此查询表达式也是正确的。）

现在的问题是：系统优化器能把原先较长的查询表达式转换为相对较短的查询表达式吗？换句话说，假设用户在查询时使用了前一种查询表达式——显然是低效率的，查询优化器能否在执行此查询前把它转换为更为有效的查询表达式？参考文献［25.15］通过证明认为，至少就现在所使用的商用优化器而言，这一问题的答案几乎是否定的。

更不幸的是，虽然我们认为较短的查询表达式“更为有效”，但如果考虑到许多产品中通常所采用的存储结构，例如 B 树索引，在这些产品中简化的查询表达式的执行效率仍会非常低（一般来说，对于每一对 *X1*、*X2*、*Y1* 和 *Y2*，都会检查至少百分之五十的索引项）。换句话说，问题不只是良好的优化的问题。

所以，我们看到，传统的 SQL 产品在某些方面的确存在着不足。具体来说，类似矩形的问题清楚地显示：某些“简单的”用户查询：(a) 在传统产品中毫无道理地显得难于表达；(b) 执行效率非常低。这些考虑都是目前促使人们钟爱对象系统的主要原因。

注意：我们将在下一章对矩形问题给出更好的解决方案（见 26.1 节）[⊖]。

## 25.2　对象、类、方法和消息

在这一节中，我们介绍对象方法中的一些主要概念，即对象本身、对象类、方法和消息。我们也将在适当的时候把这些概念与一些更熟悉的概念相联系。事实上，在一开始就把对象术语与传统的术语做大致上的映射是非常有帮助的（如图 25-3 所示）。

提示：在进行详细讨论之前，我们应该提醒大家不要期望在对象世界见到你在关系世界中所熟悉的精确性。事实上，许多对象概念（至少那些概念的公共定义）是相当不精确的，并且存在着大量不一致性，甚至在最基本的层次上也是如此（仔细阅读参考文献［25.10，25.39，25.42］就会发现这一问题）。尤其需要指出的是，现在并不存在抽象的、正式定义的“对象数据模型”，同

| 对象术语 | 传统术语 |
| --- | --- |
| 不可变的对象 | 值 |
| 可变的对象 | 变量 |
| 对象类 | 类型 |
| 方法 | 操作符 |
| 消息 | 操作符转换 |

图 25-3　对象术语（小结）

⊖　这个解决方案涉及一个用户定义类型。当对象系统首次进入市场时，SQL 并不支持用户定义类型；现在它支持了。实际上，SQL 目前包含了几个使其更类似于“对象”的特征；然而，我们有意把这样的特征放到下一章来讨论。

时也不存在一致的非正式模型（正因为如此，在本章中我们通常把“对象模型”之类的短语加上引号）。事实上，令人惊讶的是，对象系统在抽象层次上也存在着许多混乱之处，特别是在**模型**和**实现**的区别方面。

通过上述提示，大家应该明白在本章中所提到的定义和解释也并非完全被大家所接受，并且也不是百分之百与实际运行的系统相一致。事实上，这一领域中的其他作者能够，也很可能会对我们提出的每一个定义或概念提出质疑。

**1. 对象技术概述**

问题：什么是对象？答案：任何东西！

对象方法中一个基本信条是“**任何东西都是对象**”（有时“任何东西作为**第一类**对象”）。一些对象是**不可变的**，例如整数（如 3、42）和字符串（如“Mozart”、“Hayduke Lives!”）。另一些对象是**可变的**，例如在 25.1 节开始所提到的部门和雇员对象。在传统的术语中，不可变的对象对应于值而可变的对象对应于变量[⊖] ——这里所讨论的值和变量都可以具有任意复杂度（也就是说，它们能够利用一般编程语言中所有的类型和类型生成子：数值、字符串、列表、数组、栈，等等）。*注意*：在一些系统中，“对象”一词只在对象可变的情况下才能使用，而“值”（有时称为“字面值”）则在对象不变的情况下使用。甚至在那些把“对象”一词严格对应于这两种情况的系统中，我们仍然应该意识到：除非有明确的声明，在非正式的上下文中“对象”一词还是主要针对可变对象。

每个对象都有类型（对象中的术语为**类**）。单个对象有时特指对象**实例**（instance），以便和对象类型或类区分开。*注意*：在这里我们使用了编程语言中熟悉的术语“类型”（参见第 5 章），但在这个术语中包含了可以应用于此类对象的操作符（对象中的术语为**方法**）的集合。只读和更新操作符分别称为**观察者**和**变化者**。*注意*：实际上，一些对象系统严格区分类型和类，我们将在 25.3 中简略讨论一下这些系统；然而在不考虑这些系统的情况下，这两个术语是通用的。

对象是**封装的**。这就意味着某一对象的物理表示（即其内部结构），如 DEPT（部门）对象，对使用这一对象的用户来说是不可见的。相反，用户只知道这一对象能够执行哪些操作（方法）。例如，应用于 DEPT 对象的方法可以是 HIRE_EMP、FIRE_EMP、CUT_BUDGET 等。*注意*：这些方法构成了这一对象的唯一操作。实现这些方法的代码对对象的内部结构是可见的——用专业术语来说，这些代码（并且只有这些代码）允许“打破封装”[⊖] ——当然这些代码对用户来说仍然是不可见的。

实际上，必须提出的是在围绕封装讨论的对象文献中有大量混淆的地方。似乎最重要的和本书中所采用的封装是，当且仅当对象是第 5 章所说的**标量的**（意味着对象没有用户可见的组件），它才被封装。因此，封装和标量实际意思相同。*注意*：某种“集合”对象（参见 25.3 节）肯定不是标量的，通过这个定义来看，它也不是封装的。相反，一些作者认为所有的对象都是封装的，这不可避免地导致了某种矛盾。其他人认为这个概念除了意味着隐藏的内部结构外，还表示相关的方法在物理上与对象或对象类包装在一起（即物理上是对象的一部分）。我们认为后面的解释混合了模型和实现的考虑；的确，这个混淆正是为什么我们在第 5 章注明的根本不使用术语“封装”的另一个原因。然而在本章，我们的确不得不使用它。

封装的优点在于：当对象内部结构改变时，并不需要重新改写使用了这些对象的应用（当然，前提是实现对象方法的代码随着这种内部结构的变化而变化）。换句话说，封装意味着**物理数据独立性**。

到目前为止，前面按照数据独立性所叙述的封装特性只在数据库角度看有意义，但这并不是

---

⊖ 然而，注意这个未加限定的术语“变量”一般用于对象上下文中指非常具体的一个变量——局部变量或者“实例变量”——变量存放对象 ID（参见本节的后面）。

⊖ 正如第 5 章和第 20 章提到的，我们自己推荐一个更有说服力的原则：仅仅允许那些通过模型规定的操作符（特别是选择符和 THE_ 操作符）能够打破封装；所有其他操作符应该根据那些规定的操作符来实现。防御性的编码！但是既然没有一个一致的对象模型，对那些“规定的操作符”的概念也就没有一个一致的对象系统。

对象系统中通常所表述的概念。在对象系统中，封装对象被描述为拥有一块私有的空间（private memory）和一个公共的接口（public interface）：

- **私有空间**由**实例变量**构成（也被称为成员或属性），其值表示了对象的内部状态。在一个“纯”系统中，实例变量除了对于上面提到的实现方法的代码外，对所有用户来说是完全私有和不可见的。然而许多系统并非如此之“纯”，在这些系统中实例变量对于用户是可见的。关于这一点我们在下一小节中还将提到。
- **公共接口**由应用于这一对象的所有方法的接口定义构成。这些接口定义与我们在第 20 章所说的描述签名（specification signature）相对应，所不同的是（如下一段所解释的那样）：对象系统通常要求这种标识只绑定某一个特定的“目标”类型或类，而在第 20 章中并没有这种要求（我们不认为这是必需的，甚至也不希望这么做［3.3］）。正如前面所说，实现方法的代码与实例变量一样对用户不可见。注意：更准确地说，公共接口是类定义对象的一部分，而非单个对象的一部分。**类定义对象**（CDO）是总的定义类的对象，而单个对象是它的一个实例；这有点类似于传统数据库系统中的目录项或描述符。很明显，公共接口对于类的所有对象都是公用的，而不只针对某些单个对象，因此，它成为 CDO 的一部分是有意义的。

方法通过**消息**来调用。一条消息实质上就是一个操作符调用，在调用时对其中一个参数，即**目标**（target），进行特殊的语法处理。例如，下面是对部门 D 所发的一条消息，要求它雇佣雇员 E：

```
D HIRE_EMP ( E )
```

（这是假设的语法；在 25.3 节的“类、实例和集合的比较”小节中会对参数 D 和 E 进行解释）。这里的目标是用 D 表示的部门对象。在更传统的编程语言中这一消息的类似写法为（即平等地对待所有参数）[⊖]：

```
HIRE_EMP ( D, E )
```

实际上，一个对象系统总会有一些内置的类和方法。几乎所有的系统中都会提供类似 INTEGER（其方法有“=”、“<”、“+”、“-”等）、CHAR（其方法有“=”、“<”、“||”、SUBSTR 等）等类。当然，系统也为高级用户提供了让他们自己来定义和实现类和方法的途径。

**2. 实例变量**

我们现在进一步看看有关实例变量的概念。事实上，关于这一命题还相当的混乱。正如前面所述，在一个“纯”系统中实例变量对用户来说是不可见的。遗憾的是，按照这一规定，绝大多数系统都不是“纯”系统。其结果是，在实际中我们必须区分公共和私有的实例变量；**私有**的实例变量是真正隐藏的，而公共的实例变量则不是。

举例来说，假设有一个线段对象类，线段在物理上通过其起始点 BEGIN 和终止点 END 来表示（回忆一下，我们在第 5 章使用了相似的例子）。这样系统一般会允许用户以 *ls*. BEGIN 和 *ls*. END 的表达形式获得一条给定线段 *ls* 的起始点和终止点。这里，BEGIN 和 END 就是两个公共实例变量。（注意：按照定义，访问公共实例变量必须通过一些特殊的语法——典型地，如我们的例子中用圆点来限定。）如果线段的物理表示改变了，例如，线段的中点、长度和斜率改变了，那么凡是包含了 *ls*. BEGIN 和 *ls*. END 语句的程序就都被打破了，我们失去了数据的独立性。

可以看到，公共实例变量在逻辑上并不是必需的。假设对于线段定义了方法 GET_BEGIN、GET_END、GET_MIDPOINT、GET_LENGTH 和 GET_SLOPE，那么用户同样可以通过方法调用 GET_BEGIN（*ls*）、GET_END（*ls*）、GET_MIDPOINT（*ls*）来获得线段的起始点、终止点、中点等信息。这样的话，线段的内部物理结构对用户来说就无关紧要了——只要各 GET_方法都实

---

⊖ 专门处理其中一个参数可以使得系统执行第 20 章描述的运行时绑定过程更容易。然而，这种方法也有许多缺点［3.3］，其中最常见的问题是使得方法的实现者更难以实现代码。另外，选择哪个参数（在有两个或更多参数的情况下）作为目标是任意的。

现正确，并且在物理结构改变时能够做相应的变化。此外，如果用户把 GET_BEGIN（*ls*）缩写为 *ls*. BEGIN 也是可以的，但注意这时它只是 GET_方法的一种缩写形式，而并非把 BEGIN 作为一个公共实例变量。然而，实际中系统通常并不这么操作，在这些系统中公共实例变量的确暴露了对象的物理结构（至少是暴露其中的一部分，虽然另一些实例变量是私有和不可见的）。为了和实际情况相一致，在下面的论述中我们将假设对象一般都存在某些公共实例变量，尽管从逻辑上来说这一概念是不必要的。

在这里还需要阐述另一个观点。假设正好需要某些变量——用户不妨把其当作“实例变量”——来创建某一特定类的对象[○]，这并不意味着这些“实例变量”能用来达到任意目的。例如，假设创建一条线段需要指定 BEGIN 点和 END 点，这并不意味着通过给定的点 BEGIN 就能得到所有的线段；这一要求只有在适当的方法被定义后才是有效的。

最后，注意在一些系统中支持一种私有实例变量的变体——**保护**（protected）实例变量。如果类 *C* 的对象含有一个保护实例变量 *P*，那么 *P* 对于类 *C* 上定义的方法的实现代码和类 *C* 的任意子类（任意层次）上定义的方法的实现代码来说，都是可见的。在 25.3 节的最后将对子类做简单讨论。

**3. 对象标识**

每一个对象有一个唯一的标识（identifier），叫做**对象 ID** 或 OID。类似整数 42 这样的不可变对象是自我标识的，也就是说，它们本身就是自己的 OID。与此相反，可变对象拥有（概念上的）地址作为其 OID，并且通过这些地址可以在数据库的其他任何地方以（概念上的）指针的形式来引用相关的对象。这些地址并不直接面向用户，而是被赋给程序变量和其他对象中的实例变量。在 25.3 节和 25.4 节中将会进一步讨论这一问题。

我们顺便再来评价一下以下的说法：对象系统的一个优点在于两个不同的对象从所有用户可见的角度来看都一致时——即互为复制——仍然可通过其 OID 来区分。然而，这一叙述却显得似是而非，因为用户在外部如何能够区分这两个对象呢？关于这一问题的进一步讨论见参考文献 [6.3，6.6]，特别推荐参考文献 [25.17]。

## 25.3　进一步的分析

现在进一步看看前一节中介绍的一些思想。假设我们希望定义两个对象类，DEPT（部门）和 EMP（雇员）。同时假设已经定义了用户自定义的类 MONEY 和 JOB，而类 CHAR 是内置的。这样，对 DEPT 和 EMP 必要的类定义可以写成如下形式（假定的语法）：

```
CLASS DEPT
   PUBLIC  ( DEPT#       CHAR,
             DNAME       CHAR,
             BUDGET      MONEY,
             MGR         OID ( EMP ),
             EMPS        OID ( SET ( OID ( EMP ) ) ) )
   METHODS ( HIRE_EMP ( OID ( EMP ) ) code ,
             FIRE_EMP ( OID ( EMP ) ) code , ... ) ... ;

CLASS EMP
   PUBLIC  ( EMP#        CHAR,
             ENAME       CHAR,
             SALARY      MONEY,
             POSITION    OID ( JOB ) )
   METHODS ( ... ) ... ;
```

下面进行几点说明：

1）我们通过**包含层次**（containment hierarchy）的方式来表示部门和雇员，即 EMP 对象在概念上是包含于 DEPT 对象的。这样，类 DEPT 的对象包括一个公共实例变量 MGR，代表给定部门的经理；以及另一个公共实例变量 EMPS，代表给定部门中的所有雇员。更精确地说，类 DEPT 的对象包括的公共实例变量 MGR，其值是对一个雇员的引用（如其 OID）；而另一个称为

○ 这些对象必须是可变的（为什么?）。

EMPS 的值则是指向一个雇员引用集合的引用。稍后我们将对包含层次的观点进行详细阐述。

2）为了例子的简化起见，我们在类 EMP 的对象中没有包含一个实例变量，其值为部门的 OID 或者 DEPT#值（一个“外码”实例变量）。这一决定与我们通过包含层次的方式来表示部门和雇员的决定是一致的。然而，这意味着不能通过直接的方式从 EMP 对象中获得相应的 DEPT 对象。在 25.5 节的“关系间的联系”小节中将对这一点做进一步讨论。

3）我们注意到，每个类的定义都包含有应用于此类对象的方法的定义（详细代码省略）。当然，这些方法的目标类就是在其定义中包含了方法定义的类[⊖]。

图 25-4 显示了按照 DETP 和 EMP 类定义所得到的一些对象实例。首先考虑在图顶部的 EMP 对象（OID 为 *eee*），它包括：

- 公共实例变量 EMP#，其值为内置类 CHAR 的不可变对象“E001”；
- 公共实例变量 ENAME，其值为内置类 CHAR 的不可变对象“Smith”；
- 公共实例变量 SALARY，其值为用户定义类 MONEY 的不可变对象“$ 50 000”；
- 公共实例变量 POSITION[⊖]，其值为用户定义类 JOB 的可变对象的 OID。

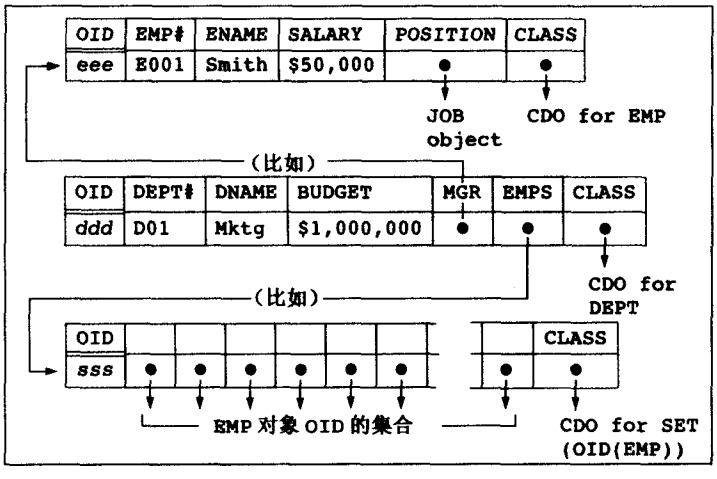



图 25-4　DEPT 和 EMP 实例

同时它至少包含两个附加的私有实例变量，一个包含 EMP 对象自身的 OID—*eee*；而另一个包含 EMP 的类定义对象——即 CDO——的 OID（这样，在实现中才能找到此对象的描述符信息）。*注意*：这两个 OID 可能在物理上与对象存放在一起，也可能不放在一起。例如，值 *eee* 并不需要作为相应的 EMP 对象的一部分被存放；所必需的只是在实现中通过给定值 *eee* 能够定位 EMP 对象（也就是说，在值 *eee* 和 EMP 对象的物理地址之间存在映射关系）。但从概念上来讲，用户总可以认为 OID 是对象本身的一部分。

现在再让我们回到图中间的 DEPT 对象，其 OID 为 *ddd*，它包括：

- 公共实例变量 DEPT#，其值为内置类 CHAR 的不可变对象“D01”。
- 公共实例变量 DNAME，其值为内置类 CHAR 的不可变对象“Mktg”。
- 公共实例变量 BUDGET，其值为用户定义类 MONEY 的不可变对象“$ 1 000 000”。
- 公共实例变量 MGR，其值为用户定义类 EMP 的可变对象的 OID—*eee*（这是代表部门经理对象的 OID）。
- 公共实例变量 EMPS，其值为用户定义类 SET（REF（EMP））的可变对象的 OID—*sss*

---

⊖ 注意假定的语法（虽然有些不合需要，但非常典型）混合了模型和实现的考虑。同时注意我们已经在别的地方[14.12]指出，部门和雇员是对象类的不好的例子！然而这一点进一步的讨论将偏离这里的主题。

⊖ 其他三个公共实例变量也被认为是包含 OID，因为不可变的对象本身就是自己的 OID。

（见下文）。

- 两个私有实例变量，分别包含 DEPT 对象本身的 OID—*ddd* 以及相应的类定义对象的 OID。

最后，OID 为 *sss* 的对象包括单个（可变）EMP 对象 OID 的集合，加上通常所要包含的私有实例变量。

图 25-4 描述了这些对象的“真实状态”；也就是说，此图阐明了“对象模型”的数据结构部分，这样一个图对于模型的用户来说是易于理解的。然而，典型的对象文本和表示并不像图 25-4 那样，而是如图 25-5 所示的那种方式（可能被认为是更高层次的抽象，从而更容易理解）。

当然，图 25-5 所示的表达方式与包含层次的解释更为一致。然而它掩盖了上面所强调的一个重要事实，那就是：对象通常包含的并非其他对象本身，而只是这些对象的 OID——即指向对象的指针。例如，在图 25-5 中我们清楚地看到 DEPT 对象 D01 包含 EMP 对象 E001 两次（这可能就暗示雇员 E001 在两个不同的场地也许会有两种不同的工资，从而导致不一致）。这种写法正是许多混乱的来源，所以我们倾向于类似图 25-4 的表示方法。

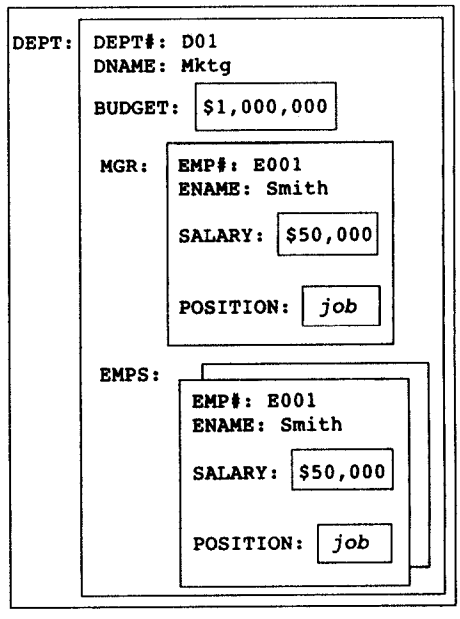



图 25-5　作为包含层次的 DEPT 和 EMP 实例

作为补充，我们注意到真正的对象类定义通常使混乱进一步加剧，因为它们一般总是不把实例变量定义为“OID”（如同我们在假定的语法中的定义），而是直接反映出其中的包含层次关系。例如，对象类 DEPT 中的实例变量 EMPS 不被定义为 OID（SET（OID（EMP））），而是直接定义为 SET（EMP）。虽然前者显得麻烦，但仍然建议使用我们的定义方式，因为这样显得清晰和准确。

值得指出的是，所有原先对于一般的层次系统的批评，例如 IMS，同样适用于包含层次。由于篇幅限制，我们不能在这里仔细考虑这些批评；总地来说，层次系统**缺乏对称性**。具体来说，层次系统不适合多对多关系的表示。例如，考虑供应商和零件的情况：供应商包含零件？还是零件包含供应商？或是两者互相包含？供应商、零件和工程的情况呢？

其实事情比我们所提到的还要复杂。一方面，对象被称为（如前面所解释的）是层次的，这意味着其也该接受对于层次系统的批评；但另一方面，从图 25-4 中我们又清楚地看到，对象系统除了**元组**外根本不是层次的，在其元组中可以包括以下任何成分：

1）不可变的“子对象”（即类似整数或钱数的自标识 self-identifying 的值）。

2）可变“子对象”的 OID（即指向其他可变对象的引用或指针，而这些可变对象很可能是共享的）。

3）1）、2）或 3）的集合、列表、数组等。

还有某些隐藏的成分。注意对于第 3）点来说，典型的对象系统支持若干“集合”类生成子，例如集合、列表、数组、无序单位组等（尽管一般不包括关系！），这些生成子可以以任何方式结合。例如，有一组列表，其中每一条目录包括一组单位组，而每个单位组又由一组指向整数变量的指针构成，在适当的环境下这些可以共同构成一个单一的可变对象。下面对此还将进一步论述。

### 1. 对象 ID 的再思考

典型的关系 DBMS 依靠用户定义、用户控制的码（简称“用户码”）来标识和引用实体（事实上，正如我们在第 1 章和第 3 章所了解的，指针类型的 OID 在关系数据库中是禁止的；进

一步的讨论参见第 26 章）。然而我们都知道，用户码的确有一些问题：参考文献［14.11］和［14.21］详细讨论了这些问题，并且认为关系 DBMS 应该支持由系统定义的码（“代理”），至少作为一个选项。支持在对象系统中加入 OID 在某种程度上与支持在关系系统中加入代理是相似的（然而，千万不要把两者等同：代理是值，并且对用户可见，而 OID 是地址——至少在概念上——并且对用户不可见。在参考文献［25.17］中对这些既有区别又有联系的问题进行了广泛的讨论）。

以下是几点说明和问题：

1）首先，正如我们在 25.4 节将要看到的，OID 的使用并不消除对用户码的需要。更准确地说，尽管在数据库内部所有对象间的引用都通过 OID 完成，但在与外部世界的交互中用户码仍然是需要的。

2）派生对象的 OID 是什么？——例如，给定的 EMP 与相应的 DEPT “连接”后的对象，或者是给定的 DEPT 在 BUDGET 和 MGR 上“投影”后的对象？（顺便提一句，这个关于派生对象的问题很重要，但我们在后面的 25.5 节才谈这个问题。）

3）OID 是人们批评对象系统的使用效果像在“重新使用 CODASYL”的根源——（如第 1 章所述）术语 CODASYL 一般指某些网状（关系以前的）数据库系统，如 IDMS。当然，OID 趋向于导致编程走向一个相当低层次的、使用指针跟踪的状态（见 25.4 节），这的确容易使人回想起 CODASYL。同时，OID 就是指针的事实也导致了以下的说法：

- CODASYL 系统更趋近于对象系统而非关系系统。
- 关系系统是基于值的，而对象系统是基于标识的。

**2. 类、实例和集合的比较**

对象系统清楚地区分类、实例和集合的概念。一个类（正如已经解释的）基本上是一个数据类型[⊖]，它可以是内置的，也可以是用户定义的，并允许任意的复杂度。每一个类都理解一条叫做 **NEW** 的消息，通过这条消息来创建类的（可变的）实例（注意：由 NEW 消息所调用的方法有时被看作是一个**构造函数**）。例如（假定的语法）：

```
E  :=  EMP NEW ( 'E001', 'Smith', MONEY ( 50000 ), POS ) ;
```

在这里 POS 是一个程序变量，包含某个 JOB 对象的 OID。方法 NEW 在对象类 EMP 中被调用；它创建了类的一个新的实例，并给出初始化值，返回新实例的 OID，最后这个 OID 被赋给了程序变量 E。

因为对象能被任何数量的其他对象所引用（通过 OID），因而对于其他对象来说具有很好的共享性。尤其是，它们还能同时属于任何数量的**集合**对象。继续我们的例子：

```
CLASS EMP_COLL
   PUBLIC ( EMPS OID ( SET ( OID ( EMP ) ) ) ) ... ;

ALL_EMPS  :=  EMP_COLL NEW ( ) ;

ALL_EMPS ADD ( E ) ;
```

解释：

1）类 EMP_COLL 的一个对象包含单一公共实例变量 EMPS，其值是指向一个可变对象的指针（OID），而这个可变对象的值为一组指向单个 EMP 对象的指针（OID）。

2）ALL_EMPS 为程序变量，其值为类 EMP_COLL 某个对象的 OID。当赋值操作完成后，ALL_EMPS 就包含了这一对象的 OID，而对象本身的值也为一个 OID，指向一个尚未包含任何 EMP 对象 OID 的空集。

3）ADD 是类 EMP_COLL 对象所理解的方法。在这个例子中，此方法通过包含类对象 OID

---

⊖ 正如 25.2 节所提到的，一些系统既使用“类型”也使用“类”，其中“类型”的意思是类型或者内涵，“类”表示外延（如某种集合），有时也指类型的实现。其他系统使用其他方式来用这些术语……我们将继续使用“类”表示在第 5 章里所介绍的类型。

的程序变量 ALL_EMPS 来调用，其结果是把某一 EMP 对象的 OID 通过包含此 OID 的程序变量 E 加入到 ALL_EMPS 对应的 EMP_COLL 对象所指向的 OID 集合中（此集合开始时为空集）。

经过以上操作，我们大致可以说，变量 ALL_EMPS 代表了一组 EMP 的集合，而目前此集合中只包含一个 EMP 对象，即雇员 E001（顺便说一句，请注意使用用户码值 E001 的必要性!）。

当然，在任何时刻我们可以拥有任意个不同的、但互相之间很可能有所重叠的“雇员集合”：

```
PROGRAMMERS  :=  EMP_COLL NEW ( ) ;
PROGRAMMERS ADD ( E ) ;
    .....
HIGHLY_PAID  :=  EMP_COLL NEW ( ) ;
HIGHLY_PAID ADD ( E ) ;
```

等等。与 SQL 系统中的表达方式相反。例如，SQL 语句

```
CREATE TABLE EMP
     ( EMP# ....... NOT NULL,
       ENAME ...... NOT NULL,
       SALARY ..... NOT NULL,
       POSITION ... NOT NULL ) ... ;
```

同时创建了一个类型和一个集合。类型在表头定义，而集合（初始为空）即为表的主体。同样，SQL 语句

```
INSERT INTO EMP ( ... ) VALUES ( ... ) ;
```

创建了某个单一的 EMP 元组（假设 INSERT 只是插入单个一行），并将它加入到 EMP 集合中。所以，在 SQL 中：

1）不存在不包含于任何“集合”的 EMP“对象”，事实上，这里有且只有一个“集合”（见下面的讨论，注意把一个 EMP 元组作为一个“对象”的列是有点值得怀疑，我们将在第 26 章讨论）。

2）无法直接创建 EMP“对象类”的两个不同的“集合”（见下面的讨论）。

3）无法直接共享不同的 EMP“对象集合”中的相同“对象”（见下面的讨论）。

至少以上这些说法我们都听说过，但事实上，它们都是经不住仔细推敲的。首先，可以通过外码机制来解决这些问题；例如，我们可以定义两个更基本的表 PROGRAMMERS 和 HIGHLY_PAID，每个都包含相关雇员们的雇员号。其次（更为重要），可以通过视图机制来达到相似的效果。例如，可以定义 PROGRAMMERS 和 HIGHLY_PAID 作为基表 EMP 的视图：

```
CREATE VIEW PROGRAMMERS
    AS SELECT EMP#, ENAME, SALARY, POSITION
       FROM   EMP
       WHERE  POSITION = 'Programmer' ;

CREATE VIEW HIGHLY_PAID
    AS SELECT EMP#, ENAME, SALARY, POSITION
       FROM   EMP
       WHERE  SALARY > some threshold ;
```

这样，同一个 EMP“对象”当然也能够同时属于两个或更多个不同的“集合”。此外，由于集合中的成员恰好是视图，所以系统自动获取它们的地址，而不用程序员手工设置。

下面通过一个类似的对比来结束讨论，即对象系统中的可变对象和某些程序语言中所支持的**显式动态变量**（explicit dynamic variable，例如，PL/I 的 BASED 变量）之间的比较。就像给定类的可变对象，对于一个给定类型也可以有任意多个不同的显式动态变量，其存储空间是在程序执行过程中显式分配的。此外，也像单个可变对象一样，这些不同的变量没有名称，只能通过指针来引用。例如，在 PL/I 中，我们可以这样写：

```
DCL XYZ INTEGER BASED ;              /* XYZ 是一个 BASED 变量 P 是一个指针变量 */
DCL P POINTER ;
ALLOCATE XYZ SET ( P ) ;          /* 创建一个新的 XYZ 实例，并通过 P 指向它 */
P -> XYZ = 3 ;                 /* 通过 P 把 3 赋给 XYZ */
```

等等。这些 PL/I 代码看起来与前面的对象代码极为相似；具体来说，BASED 变量的声明类似于对象类的创建，而 ALLOCATE 操作类似于类的一个实例创建。这样我们就很清楚为什么在对象模型中需要 OID 了，因为对象不拥有任何其他的唯一标识，而只能靠 OID 来标识——就像 PL/I 中 BASED 变量的实例。

**3. 类的层次**

如果不提一提**类层次**（class hierarchy，不要和包含层次的概念相混淆），那么对基本对象概念的介绍是不完整的。然而，对象中“类层次”的概念在本质上与第 20 章中所详细讨论的类型层次的概念是一致的；所以在这里只做简要的定义（大部分解释来自第 20 章）和一些相关的说明。注意：关于抽象继承模型，无论是在对象系统还是在其他领域，完全一致的方面是很少的，所以在细节描述上不同的继承系统差别会非常大。

首先，对象类 *Y* 被称为是对象类 *X* 的一个**子类**——反之，对象类 *X* 被称为是对象类 *Y* 的一个**超类**——当且仅当类 *Y* 的每个对象都为类 *X* 的对象（“*Y* **ISA** *X*”）。所以类 *Y* 的对象**继承**类 *X* 的所有公共实例变量和方法[⊖]，其中对公共实例变量的继承称为**结构继承**，而对方法的继承称为**行为继承**。当然，在一个纯系统中只有行为继承而没有结构继承——至少对于标量或完全封装的对象是这样——因为没有可继承的结构（即没有对用户可见的结构）。然而实际上，对象系统几乎都支持一定程度的结构继承（也就是对公共实例变量的继承）。注意：如果一个子类除了从它的超类继承的变量之外还有附加的公共实例变量，就说它扩展了它的超类。

如果类 *Y* 是类 *X* 的子类，那么用户在任何 *X* 类的对象被允许的地方都可以使用 *Y* 类的对象（例如，作为不同方法中的某个参数）——即**可置换性原则**，而这样就实现了**代码重用**。然而，因为对象系统通常不明确区分值和变量——即不可变对象和可变对象——在值和变量的置换方面很可能出现问题（第 20 章中对此已有详细的讨论）。尽管如此，把同一方法应用于类 *X* 对象和类 *Y* 对象的能力被称为**多态性**。

系统一般都会有某些内置的类层次。例如，在 OPAL 中（见 25.4 节），每一个类都被认为是内置类 OBJECT 在某一层次上的子类（因为“任何东西都是对象”）。OBJECT 的内置子类包括 BOOLEAN、CHAR、INTEGER、COLLECTION 等；COLLECTION 有一个子类 BAG，而 BAG 又有一个子类 SET 等。（但 COLLECTION、BAG 和 SET 确实不是这样的类，而是“类生成器”——如 **Tutorial D** 中的 RELATION？这里看上去有些混淆。）

重要的一点是对象系统一般不允许对象改变它们的类（参见对文献［20.12］的评注）。结果，对象系统不通过约束支持泛化或者专有化；因此，这样的系统不能支持我们认为的“好”的继承模型。我们将在下一章详细描述这一点（见 26.3 节）。

最后，在一些对象系统中除了单一继承外，还支持**多继承**。然而，就笔者所知尚没有对象系统支持参考文献［3.3］意义下的元组或关系的继承（无论是单一的还是多个的继承）。

## 25.4　一个详实的示例

我们已经介绍了对象系统的基本概念。在这一节中通过一个详实的例子来说明如何把这些概念结合在一起——也就是说，来看一看对象数据库是如何定义、如何加入数据以及如何查询和更新数据的。我们的例子基于 GemStone 产品（来自 GemStone 系统有限公司）和它的数据语言 OPAL［25.13］；而 OPAL 基于 Smalltalk 语言［25.23］。注意：Smalltalk 是最早的和最纯的对象语言之一（这也是为什么我们选择它的原因）。这里要补充一点，Smalltalk 在市场上正被 C++ 和 Java 所取代。

---

⊖ 它们也可能继承私有实例变量，但是我们把这种继承看作实现问题，而不是模型的一部分。

这个例子是第 9 章中习题 9.7 所提到的教育数据库的一个简化版本。这个数据库包含关于某个公司内部教育培训计划的信息。对于每门课程，数据库中包含开设这门课的所有班的详细信息，而每个班又包含所有选修学生和任课老师的信息。最后，数据库中还放置了所有雇员的具体信息。此数据库的关系版本大概是这样设计的：

```
COURSE      { COURSE#, TITLE }
OFFERING    { COURSE#, OFF#, OFFDATE, LOCATION }
TEACHER     { COURSE#, OFF#, EMP# }
ENROLLMENT  { COURSE#, OFF#, EMP#, GRADE }
EMP         { EMP#, ENAME, SALARY, POSITION }
```

图 25-6 是此数据库的相关图表。

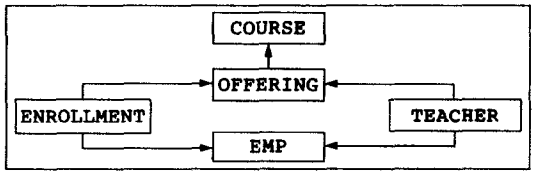



图 25-6　教育数据库的相关图表

**1. 数据定义**

现在来看一看关于此数据库的一系列 OPAL 定义。首先是对象类 EMP 的定义（加入行号是为了便于下文的引用）：

```
1  OBJECT SUBCLASS : 'EMP'
2     INSTVARNAMES : #[ 'EMP#', 'ENAME', 'POSITION' ]
3     CONSTRAINTS  : #[ #[ #EMP#, STRING ] ,
4                        [ #ENAME, STRING ] ,
5                        [ #POSITION, STRING ] ] .
```

解释：行 1 定义了一个对象类 EMP，作为内置类 OBJECT 的子类。（在 OPAL 的术语中，行 1 向 OBJECT 对象发送了一条消息，要求它调用方法 SUBCLASS；INSTVARNAMES 和 CONSTRAINTS 确定了此方法调用时的参数。就像 OPAL 中其他事物一样，一个新类的定义是通过向一个对象发送一条消息来完成的。）行 2 说明类 EMP 的对象有三个私有实例变量 EMP#、ENAME 和 POSITION，行 3 ~ 5 则限定这些实例变量为指向类 STRING 的对象。注意：在本节中我们忽略了对纯语法细节的讨论（例如上面所大量使用的记号“#”），因为这些与讨论的目的没有实质性联系。

再次重复一下，实例变量 EMP#、ENAME 和 POSITION 是类 EMP 的私有变量；它们只能在类 EMP 实现方法的代码中使用。例如，在这里我们定义了方法“get”和“set”（即查询和更新）雇员数目，其中符号“^”可读作“返回”）：

```
METHOD : EMP
   GET_EMP#
      ^EMP#
%

METHOD : EMP
   SET_EMP# : EMP#_PARM
      EMP#  :=  EMP#_PARM
%
```

在下一小节还将涉及方法定义的讨论。下面给出类 COURSE 的定义：

```
1  OBJECT SUBCLASS : 'COURSE'
2     INSTVARNAMES : #[ 'COURSE#', 'TITLE', 'OFFERINGS' ]
3     CONSTRAINTS  : #[ #[ #COURSE#, STRING ] ,
4                        [ #TITLE, STRING ] ,
5                        [ #OFFERINGS, OSET ] ] .
```

解释：行 5 确定了私有实例变量 OFFERINGS 包含类 OSET（将在下面定义这个类）某个对

象的 OID。通俗地说，OFFERINGS 表示相关课程所有开设班的集合；换句话说，我们把课程/班级的关系构建成一个包含层次，其中班级在概念上被相应的课程所包含。

接下来是类 OFFERING 的定义：

```
1  OBJECT SUBCLASS : 'OFFERING'
2     INSTVARNAMES : #[ 'OFF#', 'ODATE', 'LOCATION',
3                                'ENROLLMENTS', 'TEACHERS' ]
4     CONSTRAINTS  : #[ #[ #OFF#, STRING ] ,
5                        [ #ODATE, DATETIME ] ,
6                        [ #LOCATION, STRING ] ,
7                        [ #ENROLLMENTS, NSET ] ,
8                        [ #TEACHERS, TSET ] ] .
```

解释：行 7 确定了私有实例变量 ENROLLMENTS 包含类 NSET 某个对象的 OID；通俗地说，ENROLLMENTS 表示相关班级所有报名者的集合。同样，TEACHERS 表示相关班级所有老师的集合。所以，在这里我们又采用了包含层次的表达方式。NSET 和 TSET 将在下面定义。

接下来是类 ENROLLMENT：

```
1  OBJECT SUBCLASS : 'ENROLLMENT'
2     INSTVARNAMES : #[ 'EMP', 'GRADE' ]
3     CONSTRAINTS  : #[ #[ #EMP, EMP ] ,
4                        [ #GRADE, STRING ] ] .
```

解释：私有实例变量 EMP（行 3）包含类 EMP 某个对象的 OID，表示报名者所对应的具体雇员信息。注意：为了保持包含层次的表示方式，我们把 EMP 对象放在 ENROLLMENT 对象的“内部”。但应该看到其中的不对称性：报名是一个多对多的关系，但对该关系中的参与者雇员和班级的处理是完全不同的。

最后为有关教师的类。为了更好地说明问题，我们对类 TEACHER 的定义与关系数据库上的定义略有不同，把它作为类 EMP 的一个子类：

```
1  EMP SUBCLASS : 'TEACHER'
2       INSTVARNAMES   : #[ 'COURSES' ]
3       CONSTRAINTS    : #[ #[ #COURSES, CSET ] ] .
```

解释：行 1 定义了类 TEACHER 的一个对象，并作为用户定义类 EMP 的子类（换句话说就是，TEACHER “ISA” EMP）。这样，每个 TEACHER 对象拥有私有实例变量 EMP#、ENAME 和 POSITION（都从类 EMP 继承而来[⊖]），再加上包含类 CSET 某个对象 OID 的变量 COURSES；CSET 对象表示此教师可能教的所有课程的集合。每个 TEACHER 对象同时还继承所有 EMP 的方法。

正如已经提到的，上面的类中都假设一些集合类（ESET、CSET、OSET、NSET 和 TSET）的存在。这里给出这些类的定义：

```
1  SET SUBCLASS     : 'ESET'
2       CONSTRAINTS :  EMP .
```

解释：行 1 定义类 ESET 的一个对象，并作为内置类 SET 的子类。行 2 规定类 ESET 对象的内容为类 EMP 对象 OID 的集合。从理论上讲，我们能创建任意多个类 ESET 的对象，但在这里只创建一个对象（见下一小节），在这个对象中包含目前数据库中存在的所有 EMP 对象的 OID。通俗地说，单个 ESET 对象可以被认为是关系数据库中基表 EMP 在对象系统中的替代物。

CSET、OSET、NSET 和 TSET 的定义都是类似的（见下面的定义）。然而对于每个类，我们不得不创建若干个对象而不是一个；例如，对于每个 COURSE 对象都会有一个独立的 OSET 集合对象。

```
SET SUBCLASS     : 'CSET'
    CONSTRAINTS :  COURSE .
```

---

⊖ 注意，这里被继承的是私有表示（如物理实现）。

```
SET SUBCLASS     : 'OSET'
    CONSTRAINTS :  OFFERING .

SET SUBCLASS     : 'NSET'
    CONSTRAINTS :  ENROLLMENT .

SET SUBCLASS     : 'TSET'
    CONSTRAINTS :  TEACHER .
```

**2. 扩充数据库**

现在考虑一下扩充数据库过程中所要涉及的问题。依次考虑五个基本对象类（EMP、COURSE 等）。首先是雇员。我们试图通过一个 ESET 对象包含所有存在的 EMP 对象的 OID，所以首先需要创建 ESET 对象：

```
OID_OF_SET_OF_ALL_EMPS  :=  ESET NEW .
```

此赋值语句右侧的表达式返回类 ESET 的某个空实例的 OID（即 EMP 对象 OID 的空集）；此新实例的 OID 赋予程序变量 OID_OF_SET_OF_ALL_EMPS。所以我们可以很粗略地说，OID_OF_SET_OF_ALL_EMPS 表示“所有雇员的集合”。

现在，每当创建一个新的 EMP 对象就要把这个对象的 OID 通过变量 OID_ OF_SET_OF_ALL_EMPS 插入到 ESET 对象中。因此我们定义一个方法来创建 EMP 对象，并将其 OID 插入到 ESET 对象中（也可以通过应用程序来实现同样的任务）。下面是此方法的代码：

```
 1  METHOD : ESET                                          " 匿名!      "
 2    ADD_EMP#  : EMP#_PARM                                " 参数       "
 3    ADD_ENAME : ENAME_PARM
 4    ADD_POS   : POS_PARM
 5    | EMP_OID |                                          " 局部变量   "
 6    EMP_OID := EMP NEW .                                 " 新雇员     "
 7    EMP_OID SET_EMP#  : EMP#_PARM ;                      " 初始化     "
 8            SET_ENAME : ENAME_PARM ;
 9            SET_POS   : POS_PARM .
10    SELF ADD: EMP_OID .                                  " 插入       "
11  %
```

解释：

1）行 1 定义了下面的代码（到行 11 的百分号标志为止）为应用于类 ESET 对象的方法（当然，事实上在运行时系统中只有一个类 ESET 的对象）。

2）行 2～4 定义了三个参数，分别赋予外部名 ADD_EMP#、ADD_ENAME 和 ADD_POS。这些外部名在调用方法的消息中使用。其相应的内部名 EMP#_PARM、ENAME_PARM 和 POS_PARM 则在方法的实现代码中使用。

3）行 5 定义了局部变量 EMP_OID，行 6 将新生成的尚未初始化的 EMP 实例的 OID 赋予此变量。

4）行 7～9 向新的 EMP 实例发送一条消息；此消息中调用了三个方法（SET_EMP#、SET_ENAME 和 SET_POS），并且赋给每个方法一个变量（EMP#_PARM 赋给 SET_EMP#、ENAME_PARM 赋给 SET_ENAME、POS_PARM 赋给 SET_POS）。注意：这里我们假设方法 SET_ENAME 和 SET_POS 与方法 SET_EMP#类似，都已定义过了。

5）行 10 向 SELF 发送一条消息。SELF 是一个特殊的变量，表示方法所对应的对象目前正在使用之中（也就是说当前的目标对象）。此消息调用对象的内置方法 ADD（ADD 是所有集合都理解的方法）；使用此方法的结果是将 EMP_OID 所标识对象的 OID 插入到 SELF 所标识的对象中去（这样就保证了 ESET 对象包含现存所有 EMP 对象的 OID）。注意：使用特殊变量 SELF 的原因是目标对象还没有其自身的名称（见行 1）。

6）注意（如行 1 注释中所指出的）对于类中所定义的方法同样没有命名。事实上，在一般情况下 OPAL 的方法都没有名字，而是由它们的特征符（signature）来标识（在 OPAL 中定义为方法所对应的类名和相应参数的外部名两者的结合）。这种传统的方式可能导致累赘的表达；此外，如果两个方法都应用于同一个类并且采用相同的参数，则两个方法参数的外部名必须强制

不同。

我们已经有了把新建的 EMP 对象插入数据库的方法，但仍没有实际插入任何数据，所以现在执行下列代码：

```
OID_OF_SET_OF_ALL_EMPS ADD_EMP#   : 'E009'
                       ADD_ENAME  : 'Helms'
                       ADD_POS    : 'Janitor' .
```

这条语句为雇员 E009 创建了一个 EMP 对象，并把此 EMP 对象的 OID 加入到所有现存 EMP 对象的 OID 集合中。

应该看到，这时类 EMP 的内置方法 NEW 只能在上述定义的类 ESET 的方法中使用，而不能单独使用，否则将会出现一些悬空的 EMP 对象，也就是说，雇员不属于包含所有现存 EMP 对象 OID 的 ESET 对象。*注意*：我们不得不多次重复如“上述定义的方法”和“ESET 对象包含所有现存 EMP 对象的 OID”，因为没有名称的东西是很难谈论的。

现在我们转到课程对象。雇员对象是一种最简单的情况，因为它们相当于“规则实体”（使用了 E/R 模型的术语），并且不包含任何嵌入对象（不可变对象除外）。相对而言，课程对象的情况要复杂一些，虽然还是“规则实体”，但从概念上来说包含其他嵌入的可变对象。大致说来，我们必须经过如下的步骤：

1）使用类 CSET 的方法 NEW 创建一个未初始化的空的“课程集合”（实际上是 COURSE OID）。

2）定义某个方法来新建一个 COURSE 对象，并将其 OID 加入到“课程集合”中。此方法使用两个确定的参数 COURSE#和 TITLE，使所创建的 COURSE 对象拥有确定的值；同时它还调用类 OSET 的方法 NEW 创建一个未初始化的空集，用来包含类 OFFERING 对象的 OID，并把此空集的 OID 赋给新建 COURSE 对象的变量 OFFERINGS。

3）对于每一门课程反复调用上述方法创建 COURSE 对象。

接下来为班级。步骤依次如下：

1）定义某个方法新建一个 OFFERING 对象。此方法使用三个确定的参数 OFF#、ODATE 和 LOCATION，使所创建的 OFFERING 对象拥有确定的值。它同时还包括：

- 调用类 NSET 的方法 NEW 创建一个未初始化的空集，用来包含类 ENROLLMENT 对象的 OID，并把此空集的 OID 赋给新建 OFFERING 对象的变量 ENROLLMENTS。
- 调用类 TSET 的方法 NEW 创建一个未初始化的空集，用来包含类 TEACHER 对象的 OID，并把此空集的 OID 赋给新建 OFFERING 对象的变量 TEACHERS。

2）该方法还将使用参数 COURSE#，并把 COURSE#的值用来：

- 确定新建 OFFERING 对象所对应的 COURSE 对象（下一小节中将会讲解如何实现这一过程）[⊖]。
- 确定对应 COURSE 对象的“所有班级的集合”。
- 把新建 OFFERING 对象的 OID 加入到适当的“所有班级集合”中。

*注意*，（如本章前面所提到的）OID 的使用并不能避免对用户码的需要，如 COURSE#。事实上，这些码值不光在外部世界引用对象时需要，同时在数据库内部也是某些查找的基础。

3）最后，对于每个班级反复调用上述方法创建 OFFERING 对象。

应该看到，（为了保证我们的包含层次设计思想）我们没有选择建立一个“所有班级集合”。这就意味着对于许多以所有班级作为查询范围的查询——例如“找出所有在纽约上课的班级”——来说，需要在程序上有一定量的工作区代码（见下一小节）。

接下来是报名对象。报名对象和班级对象的不同之处在于 ENROLLMENT 对象包括实例变量 EMP，其值为相关 EMP 对象的 OID。因此必要的创建步骤如下：

---

⊖ 当然，如果相关的课程不能找到，这个方法必须拒绝创建新的班级。我们在这些讨论中省略了这种异常情况的详细的讨论。

1）定义某个方法新建一个 ENROLLMENT 对象。此方法使用四个确定的参数 COURSE#、OFF#、EMP#和 GRADE，并通过 GRADE 值来真正创建 ENROLLMENT 对象。同时，它还包括：

- 使用 COURSE#和 OFF#的值确定与新建 ENROLLMENT 对象相对应的 OFFERING 对象。
- 确定对应 OFFERING 对象的“所有报名者的集合”。
- 把新建 ENROLLMENT 对象的 OID 加入到适当的“所有报名者集合”中。

此外，它还包括：

- 使用 EMP#的值来确定相关的 EMP 对象。
- 将 EMP 对象的 OID 赋给新建 ENROLLMENT 对象的变量 EMP。

2）对于每个报名者反复调用上述方法创建 ENROLLMENT 对象。

最后是老师对象。老师对象与班级对象的不同之处在于类 TEACHER 是类 EMP 的一个子类，所以创建步骤为：

1）定义某个方法新建一个 TEACHER 对象。此方法使用三个确定的参数 COURSE#、OFF#和 EMP#。它还包括：

- 使用 EMP#的值来确定相应的 EMP 对象。
- 把已知 EMP 对象的特定类（含对象的具体值）转换为 TEACHER 对象的特定类，因为此时雇员同时也是老师（这种类的转换完全基于特定的系统，所以在这里不做更多的描述）。

此外，它还包括：

- 使用 COURSE#和 OFF#的值确定与新建 TEACHER 对象相对应的 OFFERING 对象。
- 确定对应 OFFERING 对象的“所有老师集合”。
- 把新建 TEACHER 对象的 OID 加入到适当的“所有老师集合”中。

2）老师所能教的所有课程的集合必须确定，并在 TEACHER 对象的 COURSE 实例变量中设置。我们忽略此步骤的具体过程。

3）对于每位老师反复调用上述方法创建 TEACHER 对象。

**3. 查询操作**

在进入查询操作的细节之前，我们需要指出的是：OPAL（和大多数对象语言一样）实际上是一个记录层（或对象层）上的语言而非集合层上的语言。所以，对于大多数问题都需要程序员编写程序代码。让我们来看一个简单的查询示例——“找到课程 C001 在纽约的所有班级”。为简便起见，我们假设存在一个变量 OOSOAC，其值为“所有课程集合”的 OID。这样，查询的代码可写成：

```
1 | COURSE_C001 , C001_OFFS , C001_NY_OFFS |
2   COURSE_C001
3     := OOSOAC DETECT : [ :CX | ( CX GET_COURSE# ) = 'C001' ] .
4   C001_OFFS
5     := COURSE_C001 GET_OFFERINGS .
6   C001_NY_OFFS
7     := C001_OFFS SELECT :
8                  [ :OX | ( OX GET_LOCATION ) = 'New York' ] .
9 ^ C001_NY_OFFS .
```

解释：

1）行 1 声明了三个局部变量：COURSE_C001，用来获取课程 C001 的 OID；C001_OFFS，用来获取课程 C001 的所有班级 OID 集合的 OID；C001_NY_OFFS，用来指向实际获取的班级 OID 集合的 OID（即在纽约的所有班级）。

2）行 2～3 向变量 OOSOAC 表示的（集合）对象发送一条消息。消息调用集合的内置方法 **DETECT**，其参数为下列表达式：

```
[ :x | p(x) ]
```

在这里 $p$（$x$）是变量 $x$ 的条件表达式，其中 $x$ 为值域变量，用来限定 **DETECT** 所操作的集合范围（在这里即 COURSE 对象的集合）。DETECT 的结果为 $x$ 集合中第一个满足 $p$（$x$）条件的对

象 OID，在这里即课程 C001 的 COURSE 对象[⊖]。COURSE 对象的 OID 赋给了变量 COURSE_C001。注意：方法 DETECT 可能还需要一个“例外”（escape）参数来处理找不到任何对象满足条件 $p(x)$ 的情况。具体细节我们不再考虑。

3）行 4～5 将课程 C001 中“所有班级集合”的 OID 赋给变量 C001_OFFS。

4）行 6～8 与行 2～3 非常相似；内置方法 **SELECT** 的操作过程与方法 DETECT 的操作过程几乎一致，只不过它返回的是所有满足条件 $p(x)$ 的对象 OID 集合的 OID。在这里即把课程 C001 所有在纽约的班级 OID 集合的 OID 赋给变量 C001_NY_OFFS。

5）最后，行 9 将 OID 返回给方法的调用者。

在这个例子中需要指出几点：

- 首先注意到，在 SELECT 和 DETECT 中条件表达式 $p(x)$ 只可以是一系列简单标量比较的交集（此为最复杂的情况）——也就是说，$p(x)$ 并不能拥有任意的复杂度。
- 在 SELECT 和 DETECT 中参数表达式外的括号可以换成大括号。在 OPAL 中使用大括号意味着在方法里可以使用索引（如果有的话）。如果只使用括号，则不用索引。
- 当我们说方法 DETECT 返回第一个满足 $p(x)$ 条件的对象的 OID 时，这里的“第一个”是依赖于 OPAL 搜索顺序的（集合本身没有内在的顺序）。在我们的例子中没有差别，因为满足条件的对象有且只有一个。
- 正如你可能注意到的，在这里我们大量使用了诸如“DETECT 方法”之类的表达，而我们在前面已指出，OPAL 中的方法是没有名字的。事实上，DETECT 和 SELECT 并不是方法名（严格意义上“DETECT 方法”这样的说法是不正确的）。它们只是内置未命名方法的外部参数名。然而为了简便，我们不妨把 DETECT 和 SELECT（和其他类似的项）作为真正的方法名称。
- 你可能还注意到我们使用了“NEW 方法”的表述（很多次）。这种使用事实上是正确的：只有返回值而没有参数的方法，对于 OPAL 中的方法没有名称这一规则而言是一个例外。

**4. 更新操作**

关系系统中 INSERT 操作在对象系统中的类似操作在前面小节中已经讨论过了。这里再讨论一下 UPDATE 和 DELETE 在对象系统中的类似操作。

- **更新**：更新操作在本质上和查询操作是一致的，只不过将其中的 GET 方法换成 SET 方法。
- **删除**：内置方法 REMOVE 用来删除对象（更准确地说是删除指定集合中指定对象的 OID）。如果一个对象没有任何对它的引用——即不再被访问，则 OPAL 会通过一个垃圾收集进程自动删除它。下面的例子完成了“从所有雇员的集合中删除雇员 E001”这一操作：

```
E001  :=  OID_OF_SET_OF_ALL_EMPS
            DETECT : [ :EX | ( EX GET_EMP# ) = 'E001' ] .
OID_OF_SET_OF_ALL_EMPS REMOVE : E001 .
```

如果我们希望遵守诸如 ON DELETE CASCADE 这样的规则，该如何操作呢？——例如，规则要求删除一个雇员的同时要把与此雇员有关的所有报名对象也删除？答案当然是我们必须再次使用适当的方法；换句话说，我们不得不写更多的程序代码。

也许有人会认为通过垃圾收集的方法来删除对象在一定程度上实现了 ON DELETE RESTRICT 规则，因为一个对象只要有对它的引用存在就不能被真正删除。然而这并不是必然的情况。例如，OFFERING 对象并不包含相应 COURSE 对象的 OID，所以它当然不“限制”对 COURSE 对象的删除。（事实上，在包含层次中缺省执行 ON DELETE CASCADE 规则，除非用户选择下列两种方式之一：(a) 在子对象中包括父对象的 OID。(b) 在数据库中存在其他对象包含有子对象的 OID。在这两种情况下包含层次中的 ON DELETE CASCADE 规则不再起作用。详见下一节中对反变量的讨论。）

---

⊖ 我们在这里假设，如 GET_ COURSE#这样的方法——类似于本节前面显示的 GET_ EMP#方法——已经定义了。

最后请注意，REMOVE 方法可以用来模拟关系中的 DROP 操作——例如，删除整个 ENROLLMENT 类。具体操作留作练习。

## 25.5　混合性问题

在这一节中，我们来简单看一看其他相关问题：

- 特定查询和相关问题
- 完整性
- 关系间的联系
- 数据库编程语言
- 性能上的考虑
- 对象 DBMS 还是一个 DBMS 吗？

**1. 特定查询及相关问题**

我们在前面的讨论中故意没有强调这一点，但如果预定义的方法是对对象进行处理的唯一途径，那么对于一些特定的查询是不可能实现的！除非类和方法都按照某种特定的原则进行设计。例如，如果类 DEPT 所定义的方法只有 HIRE_EMP、FIRE_EMP 和 CUT_BUDGET（如 25.2 节所述），那么甚至对于一些简单的查询，如“谁是程序开发部门的经理”这样的问题都无法处理。

基于同样的原因，视图的定义以及对于对象声明的完整性约束一般来说也不可能——当然，除非能遵循某种特定的设计原则。

我们建议对这些问题的解决方法如下（即遵循“特定的原则”）：

1）如第 5 章中所讨论的，定义一系列操作符（“THE_操作符”）使对象中一些合理的表达对用户可见。

2）将对象适当地嵌入到关系框架中（关于这一问题的详细讨论放在下一章）。

然而，典型的对象系统几乎都不遵循这一原则，相反：[⊖]

1）典型的对象系统定义的操作符让用户可见的不是一些合理的表达而是实际的表达（见 25.2 节关于公共实例变量的讨论）。“所有的对象 DBMS 产品目前都要求查询中（所涉及的实例变量）是公共的”[25.31]。

2）典型的对象系统支持的不是关系框架，而是基于无序单元组、数组等构成的其他框架。关于这一点，我们再次声明：在逻辑层次上，类（即类型）与关系的结合是必要的也是最好的方式（见第 3 章）；事实上，就核心模型而言，我们同样认为无序单元组和数组是不必要和不合适的。我们猜测在对象系统中强调集合而不是关系（事实上几乎是对关系的完全抛弃）的思想仍然来自于对模型和实现两者的混淆。

关于特定查询还有一个重要的问题——即查询的结果属于什么类？例如，假设我们在 25.3 节提到的部门和雇员数据库中执行查询“获取程序设计部门雇员的名字和工资”。假设查询的结果包含（公共）实例变量 ENAME 和 SALARY，但在数据库中并没有包含此结构的类。这样我们是不是需要在查询之前预定义这样一个类呢？（注意，如果按照这种思路，一个包含 $n$ 个实例变量的类仅仅为了支持查询操作就需要预定义至少 $2^n$ 个类！）无论结果类是什么，应用于结果类的方法是什么呢？

类似的问题在连接操作中也会出现。如果能够对部门和雇员对象进行连接操作，那么其结果类又是什么？有哪些相应的方法？

也许因为这一问题在纯对象系统中很难回答，所以一些对象系统通过支持“路径跟踪”操作[25.38]来代替本身的连接操作。以 25.4 节的 OPAL 数据库为例，以下可能是一个有效的路径表达式：

```
ENROLLMENT . OFFERING . COURSE
```

---

⊖ 我们假设这里讨论的对象系统跟大多数现代系统一样实际上支持特定查询。然而，早期的系统并不支持，部分原因将在本节的后面进行讨论。

其意思是："通过此 ENROLLMENT 对象引用唯一的 OFFERING 对象，再通过唯一的 OFFERING 对象访问唯一的 COURSE 对象。"[⊖] 这个表达式在关系上的类似表述涉及到两个连接操作和一个投影操作。换句话说，路径跟踪只能访问预定义的路径（事实上就像关系系统以前的数据库系统），并且只能访问预定义类的对象（更像关系系统以前的系统）。

**2. 完整性**

我们在第 9 章中称数据的完整性是非常基本的问题。然而对于对象系统，甚至是支持特定查询的对象系统来说，一般不支持声明的完整性约束；那些约束只能通过程序代码来实现（即通过方法或应用程序）。例如，考虑 9.1 节中提到的约束（或"企业规则"）："状态低于 20 的供应商不能提供数量大于 500 的零件"。在执行此约束的典型的程序代码中至少需要包括以下的方法：

- 创建发货对象的方法
- 改变发货数量的方法
- 改变供应商状态的方法
- 将发货赋给不同供应商的方法

以下是需要指出的问题：

1）显然我们已经丧失了由系统自身来决定何时进行完整性检查的可能性。

2）如何保证所有使用的方法包括了所有必要的执行代码？

3）如何避免用户跳过类似"创建发货对象"的方法而直接使用方法 NEW 来创建发货对象类，从而绕过完整性检查？

4）如果约束改变了，我们如何找到所有需要改写的方法？

5）如何保证执行代码的正确性？

6）事务约束怎么办？

7）如何通过查询来找到所有应用于某个给定对象或若干个对象组合的约束？

8）当数据库启动或其他例程执行时是否执行约束？

9）语法优化怎么办（即如第 18 章所讨论的，使用完整性约束来简化查询）？

10）对于所有这些问题在创建应用和随后的维护应用中的效率的涵义是什么？

**3. 关系间的联系**

在对象产品和对象词库中一般使用术语"关系间的联系"（relationship）来表示关系系统中通过外码所表示的关系间的内在联系，同时它们还为这种特殊的完整性约束提供了特殊的支持。再考虑部门和雇员的例子。在关系系统中，雇员表中通常会通过一个外码来参照相应的部门，这就实现了关系间的内在联系。相反，在对象系统中至少有以下三种可能的方式：

1）每一个雇员对象中包括所对应部门的 OID。这种可能性与关系系统中的方法类似，但并不完全一致（OID 和外部码并非同一事物）。

2）每一个部门包括对应雇员 OID 的集合。这种可能性与 25.3 节所描述的包含层次的方法一致。

3）方法 1 和 2 可以结合如下：

```
CLASS EMP ...
    ( ... DEPT OID ( DEPT ) INVERSE DEPT.EMPS ) ... ;

CLASS DEPT ...
    ( ... EMPS
          OID ( SET ( OID ( EMP ) ) ) INVERSE EMP.DEPT ) ... ;
```

注意实例变量 EMP. DEPT 和 DEPT. EMPS 中关于 INVERSE 的说明。这两个实例变量互为反变量：EMP. DEPT 是一个"引用"实例变量，DEPT. EMPS 是一个"引用集合"实例变量（如果它们都是引用集合变量，那么关系间的联系即为多对多而不是一对多）。

---

⊖ 实际上这个例子并不是我们所定义的一个有效的数据库路径表达，因为指针指错了位置！例如，OFFERING 并没有指向 COURSE，反而由 COURSE 所指。

当然，每一种可能性都需要一些参照完整性支持。我们将在下文中讨论参照完整性。然而首先要提出的一个明显的问题是：对象系统如何处理涉及两个以上类的关系间的联系——比如说涉及到供应商、零件和工程？此问题最好的（即最为对称的）答案似乎是创建一个新类“SPJ”，每个 SPJ 对象有一个“反变量”与适当的供应商、零件和工程构成关系间的联系。但是创建一个新的对象类对于两个以上的类显然是最好的方法。接下来一个很自然的问题是，为什么对于两个类的处理不采用同样的方式？

同样，关于反变量，为什么它对于引入不对称性、方向性以及使同一事物能够拥有两个不同的名称来说是必要的？例如，两个关系表达式在对象系统上的相应表述为：

```
SP.P# WHERE SP.S# = S# ('S1')
SP.S# WHERE SP.P# = P# ('P1')
```

这看起来有点像：

```
S.PARTS.P# WHERE S.S# = S# ('S1')
P.SUPPS.S# WHERE P.P# = P# ('P1')
```

（假定的语法，用来突出本质区别——例如，两个不同的关系联系名称 PARTS 和 SUPPS 的使用——并且避免无关性。）

**参照完整性**：现在让我们来进一步看看对象系统中支持的参照完整性（参照完整性经常被认为是对象系统的强项）。在各层次上的支持都是可能的，以下的分类来自 Cattell [25.10]：

- 无系统支持：参照完整性由用户编写的代码来控制（这正好是原先的 SQL 标准中的规范）。
- 参照验证：系统检查所有参照的对象是否存在，其类型是否正确；然而，不允许直接删除对象——只有当对象上无任何引用时才能通过垃圾收集的方式自动删除对象，就像在 OPAL系统中一样。正如 25.4 节中所解释的，在这一层次的支持有点类似（a）在包含层次中对非共享子对象所采用的 ON DELETE CASCADE 规则；（b）对其他对象所采用的 ON DELETE RESTRICT 规则（必须保证“指针所指位置正确”）。
- 系统维护：系统自动保持所有参照为最新的（例如，所引用对象被删除时应设为空指针）。这一层次的支持与 ON DELETE SET NULL 规则类似。
- “自定义语义”：ON DELETE CASCADE（不在包含层次中）可能是一个很好的例子。这种可能性目前来讲一般不被对象系统所支持，而只能通过用户编写代码来实现。

**4. 数据库编程语言**

25.4 节中 OPAL 的例子表明典型的对象系统不采用 SQL 的“嵌入数据子语言”。相反，在数据库操作和非数据库操作中使用了同一种**集成式的语言**。用第 2 章中的术语来说，宿主语言和数据库语言在对象系统中是紧密结合的（事实上，它们就是一种语言）。

不可否认这种方法有其优点（这就是为什么我们在 **Tutorial D** 中采用那这种方法的原因），其中重要的一点就是使改善类型检查成为可能 [25.2]。在参考文献 [25.38] 中论证了另一个优点：

*通过采用统一的语言，在编程语言和嵌入的 DML 之间就不会再有阻抗失配的情况。*

这里术语**阻抗失配**（impedance mismatch）指的是典型的编程语言一次一记录与数据库语言（如 SQL）一次一集合在层次上的不同。这种层次上的不同的确在 SQL 产品中引起了实际问题。然而，对此问题的解决绝不是把数据库语言的层次下降到一次一记录层（这正是对象系统所采用的方法）！而应该是把编程语言的层次上升到一次一集合层。对象语言建立在记录层（如过程的）的事实其实是倒退到关系数据库以前的数据库系统，如 IMS 和 IDMS。

进一步说，由于大多数对象系统在本质上是过程化或“3GL”的，所以它们丢失了关系系统在集合层所拥有的所有优点。具体来说，系统对于用户请求进行优化的能力被严重削弱，这意味着——正像在关系系统以前的系统中那样——对性能的考虑大部分留给了用户（应用开发者或 DBA）。关于这一点见下一小节。

### 5. 性能上的考虑

系统的实际性能是所有对象系统最重要的考虑因素之一。再次引用 Cattell 的话［25.10］："性能上一个数量级的差距将会导致功能上的差异，因为如果系统的性能远低于要求，这种系统是不会有人使用的"（对原话稍做了改动）。

许多因素都与性能问题有关。这些因素包括：[⊖]

- 聚集：如我们在第 18 章中所看到的，在磁盘系统中，逻辑关联数据在物理上的聚集是影响性能的最重要的因素之一。典型的对象系统通过数据库定义（即有关类层次、包含层次或其他显式声明的对象间的联系）获取逻辑信息，并将这些逻辑信息作为数据在物理上是否聚集的一个指标。此外，DBA 也被给予了对概念/内部结构映像（使用了第 2 章中的术语）更为明晰和直接的控制。
- 高速缓存：典型的对象系统一般用在客户/服务器环境下。从服务器端获取单位聚集数据（理想情况当然是逻辑关联的数据），并缓存在客户端留待将来使用，自然会提高效率。
- 暗转：术语"暗转"（swizzling）是指当某对象被读入内存后将相应 OID 类型的指针所指向的逻辑磁盘地址转换成主存地址的过程（当然，当对象被写回到数据库中时，则进行反操作）。当应用中需要处理"复杂对象"并由此引起大量指针切换时，采用这种技术的优点是明显的。
- 在服务器端执行方法：考虑查询"获取所有内容在 20 章以上的书"。在传统的关系系统中，书被表示成 CLOB 或 BLOB 的形式（参见第 5 章），客户端的应用程序不得不依次查找每一本书，并通过扫描来确定其是否有 20 章以上的内容。相反在一个对象系统中，可以在服务器端执行"获取章的数目"操作，只有那些符合条件的书才会真正传到客户端。因此，通过这种方式在服务器上的执行方法减少了通信代价。[⊜]

注意：以上所涉及的内容并不只是关于对象的讨论，其实主要还是围绕着存储过程的讨论（见第 21 章）。一个传统的支持存储过程的 SQL 系统和支持方法的对象系统拥有同样的性能优势。

参考文献［25.12］讨论了关于在一个材料单数据库中测试系统性能的基准测试 OO1。此基准测试涉及：

1）随机获取 1000 个零件，将用户定义的方法应用于每个零件。

2）随机插入 1000 个零件，每个零件与三个其他的零件相关联。

3）随机进行零件的扩展（explosion，一直到七层结构），将某个用户定义的方法应用于每个涉及的零件，同时包括相应的内爆过程（implosion）。

根据参考文献［25.12］，通过某个对象产品（不指定）与某个 SQL 产品（不指定）的比较显示，对象系统在性能上要比 SQL 系统高两个数量级——尤其在"温"访问时（在采用高速缓存的情况下）更为明显。然而，参考文献［25.12］同时又谨慎地指出：

*这种差别……不应该归因于关系模型和对象模型之间的差别……我们有理由相信大部分的差异来自于实现上的不同。*

当数据库"足够大"（即当不再能把整个数据库放入高速缓存中）时，这一差别会相对小得多的事实支持了这种让步。

在参考文献［25.9］中描述了一个类似的、但却更为广泛的基准测试 OO7。

### 6. 对象 DBMS 还是一个 DBMS 吗？

注意：这一小节的绝大部分内容来自参考文献［25.16］，在这篇文献中论证了对象系统和关系系统之间的差别比通常想像的还要大。文中写道：

---

⊖ 除了所列的因素之外，也可以认为对象系统通过"使用户更贴近内核"（也就是说，通过将应该在实现中隐藏的指针和其他特征暴露出来）改进了性能。

⊜ 实际上这种表述过分简单。以数据为主的方法如"获取章的总数"实际上在服务器执行得更好，但其他方法如以显示为主的方法可能在客户端执行得更好。

对象数据库的产生来自于开发对象应用的程序员——为了各种特定于应用的原因——希望把他们的特定于应用的对象放入永久性的存储器中。这种永久性的存储器也许就可以被看作是一个数据库，但事实上它是特定于应用的，而非共享的通用数据库；真正的数据库应该能够适应在定义数据库时尚未预料到的应用需求。所以，许多专家认为的数据库应该具有的基本特性，在对象系统（至少是原始的对象系统）中被简单地忽略掉了。结果下列的特性都未得到满足：

1）在所有应用中数据共享
2）物理数据的独立性
3）支持特定的查询
4）视图和逻辑数据独立性
5）声明独立于应用的完整性约束
6）数据归属与灵活的安全机制
7）并发控制
8）适应所有应用的字典
9）独立于应用的数据库设计

当在数据库中存储对象的基本思想提出之后，所有问题都逐渐暴露出来了，而这些都构成了对原始对象数据库的追加特性……一个重要的结果是……在对象 DBMS 和关系 DBMS 中的确存在着**一定程度的差异**。事实上，可以论证一个对象 DBMS 根本不是一个真正的 DBMS——至少与关系 DBMS 相比较而言。其基于的考虑是：

- 关系数据库随时可以使用。换句话说，一旦系统安装完毕，用户……能够马上建立数据库，编写应用，运行查询，等等。
- 相反，对象 DBMS 可以被认为是一个构建 DBMS 的软件包。当它安装完毕后并不能立即使用……相反，首先必须让有经验的技术人员进行适当的加工，定义必要的类和方法等（为此，系统提供了一系列构建块——类库维护工具，方法编译器等）。只有当加工过程完成之后，系统才能被应用程序员和最终用户使用；换句话说，加工后的结果才更类似于一个 DBMS 系统。

还需注意：所完成的“经过加工的”DBMS 是特定于应用的。例如，它可能适合 CAD/CAM 应用，但对于医学应用来说却根本没用。换句话说，这仍然不是一个通用的 DBMS，而关系 DBMS 却是一个真正的通用 DBMS。

文献［25.16］还驳斥了在数据库中可以包含任意复杂度的（可变）对象的思想——这一思想经常被称为“与类型无关的持久性”（persistence orthogonal to type）［25.2］：[⊖]

对象模型要求支持大量类型生成子……例如 STRUCT（或 TUPLE）、LIST、ARRAY、SET、BAG，等等……通过对象 ID，这些类型生成子的可用性实质上使得在应用程序中创建的任何数据结构在对象数据库中也可以创建为相应的对象——进而这一对象的结构对用户可见。例如，考虑给定部门雇员集合的对象 EX，EX 在实现上可能是一个链表，也可能是一个数组，而用户不得不了解其到底是什么（因为相应的访问方式会有所不同）。

当然，这种对于数据库中存储的内容可以进行任意操作的方式是对象模型和关系模型的一个主要不同点，并且值得在这里做进一步讨论。本质上：

- 对象模型认为能够存储任何所感兴趣的东西——通过一般的编程语言机制所能创建的任何数据结构。
- 关系模型也是同理，但继而坚持认为所存储的任何东西都要以纯关系形式呈现给用户。

更精确地说，关系模型并没有提在物理层能够存储什么……所以它对于物理层能够存储何种数据结构没有任何限制。唯一的要求是：无论在物理层上的实际存储结构是什么，在逻辑层都必须映射为关系，从而对用户不可见。这样，关系系统在逻辑和物理（数据模型与实现）之间有

---

⊖ 参见文献［25.19］。

了清晰的区分，而对象系统没有这么做。一个相应的结果是——与传统的思路相反——对象系统比关系系统提供更少的数据独立性。例如，假设上面提到的对象 EX 所在的对象数据库将其实现由数组转变为链表，那么已存在的访问对象 EX 的代码该怎么办？

## 25.6 小结

本节列出了我们所提到的对象模型的主要特性，同时对哪些特性是必需的、哪些是尽量要有的、哪些是不应该有的、哪些与系统是对象系统还是其他系统的问题无关，等等，做一个主观评价。这些分析为我们在第 26 章讨论对象/关系系统铺平了道路。

- **对象类**（即类型）：一定是必需的（事实上对象类是所有构建的基础）。
- **对象**：对象本身，无论是“可变的”还是“不可变的”，同样是必需的——虽然我们倾向于把它们简单地称为变量或值。
- **对象 ID**：不是必需的，事实上也不推荐该特性（在模型层次上），因为它们本质上就是指针。对这个问题在参考文献［25.17］中有广泛讨论。
- **封装**：正如 25.2 节中所解释的，“封装”即意味着标量，我们倾向于使用“封装”这个术语（毕竟一些“对象”不是标量的）。
- **实例变量**：首先，按照定义私有（也称为保护）实例变量仅仅是实现时需要考虑的问题，与抽象模型的定义无关，而抽象模型才是我们要考虑的。其次，公共实例变量在一个纯对象系统中不存在，所以也不用考虑。结论是：实例变量可以忽略；“对象”应该只通过“方法”来控制。
- **包含层次**：我们在 25.3 节中已经解释过，既然典型的包含层次包含的是对象的 OID 而非对象，所以包含层次的说法易令人误解，事实上是一个误称。*注意*：一个真正包含了对象本身的（非封装）层次是允许的，不过这一点通常不明显；它与拥有关系值属性的关系变量有一些类似。
- **方法**：方法的概念是必需的，虽然我们倾向于更为传统的术语“操作符”。将方法与类捆绑是不必要的，并且会导致许多问题［3.3］；我们倾向于将“类”（类型）和“方法”（操作符）分开定义，就如在第 5 章中所看到的，这样就避免了“目标对象”的出现以及使用“自私的方法”（selfish method）。

  我们坚持认为某些操作符是必要的：选择符（选择符与其他事物的结合能够有效地提供一种书写相关类型值的方式）、THE_操作符、赋值和等值比较操作符以及类型测试操作符（见第 20 章）。*注意*：我们拒绝“构造函数”。构造函数构造变量；既然在数据库中需要的唯一变量是关系变量，那么我们所需要的唯一“构造函数”是一个能够创建关系变量（用 SQL 中的术语来说就是 CREAT TABLE 或 CREATE VIEW）的操作符。相反，选择符选择值。所以构造函数返回指向所构造变量的指针，而选择符返回所选择的值本身。
- **消息**：消息的概念也是必需的，虽然我们倾向于更为传统的术语“调用”（注意避免调用直接面对某个“目标对象”的想法，而应该平等地对待所有参数）。
- **类的层次**（及相关概念包括继承、可置换性、包含多态性等）：这些概念是有用的，但无关紧要（我们认为类层次只是类的一部分）。
- **类、实例、集合的比较**：三者的区别是本质的，但却是无关紧要的（这些概念的区别显而易见）。
- **关系间的联系**：我们在第 14 章中已经论证过（见 14.6 节），把“关系间的联系”作为一种正式结构提出并不是一个好主意，尤其是把关系间的二进制联系也看作独立的结构。同时我们认为将关联的参照完整性约束与一般的参照完整性约束区别对待也不是一个好主意。
- **完整的数据库编程语言**：最好有，但也是无关的。然而，如今的对象系统中真正支持的语言一般是过程化的（3GL），所以我们认为不应该有此特性（事实上是一个巨大的退步）。

下面列出典型的“对象模型”不支持或不怎么支持的特性：

- **特定查询**：早期的对象系统一般不支持特定查询。在目前的系统中提供了这方面的支持，但或者是通过打破封装，或者是通过限制查询内容（这样的查询就不能算是真正的特定查询了）。
- **视图**：一般不支持（本质上与不支持特定查询的原因相同）。注意：一些对象系统的确支持“派生”或“虚拟”实例变量（必须是公共的）；例如，实例变量 AGE 可以通过目前的日期与实例变量 BIRTHDATE 两者的差派生出来。然而，这样一种能力缺乏完整的视图机制，并且我们也拒绝使用公共实例变量。
- **声明的完整性约束**：一般不支持（本质上与不支持特定查询和视图的原因相同）。事实上，那些支持特定查询的系统一般也不支持声明的完整性约束。
- **外码**：“对象模型”有若干机制来处理参照完整性，但没有一种机制能与关系模型中统一的外码机制相媲美。诸如 ON DELETE RESTRICT 和 ON DELETE CASCADE 的问题一般只能留给过程化的代码（也许是由方法实现，也许是由应用程序代码实现）。
- **闭包**：在对象系统中又有什么能够支持关系中的闭包特性呢？
- **字典**：对象系统中的字典在哪里？以什么形式表现？是否有确定的标准？注意：事实上，字典必须由专业人员建立，其任务就是如 25.5 节中所讨论的，针对所服务的应用适当地裁剪对象 DBMS。这样的字典自然是特定于应用的，整个 DBMS 也是如此。

我们希望支持的“对象模型”的基本特性如下表所示：

| 特性 | 倾向采用的术语 | 评论 |
|---|---|---|
| 对象类 | 类型 | 标量与非标量<br>可以用户自定义 |
| 不可变对象 | 值 | 标量与非标量 |
| 可变对象 | 变量 | 标量与非标量 |
| 方法 | 操作符 | 包括选择符<br>以及类型测试操作符 |
| 消息 | 操作符调用 | 无“目标”操作数 |

简而言之，对象系统唯一的好的思想是**适当的数据类型支持**，其他所有的思想（包括由用户来定义操作符）都来自于这一基本思想。[⊖] 但这一思想并不是全新的！

## 习题

25.1　用你自己的话解释下列术语：

| | | | |
|---|---|---|---|
| 类 | 包含层次 | 消息 | 对象实例 |
| 类的层次 | 封装 | 方法 | 私有实例变量 |
| 类定义对象 | 实例 | 对象 | 保护实例变量 |
| 构造函数 | 反变量 | 对象 ID | 公共实例变量 |

25.2　OID 的优点是什么？缺点又是什么？OID 如何实现？

25.3　在 25.2 节中定义了 SQL 查询“获取所有与单位正方形相重叠的矩形”的两种表述形式。证明这两种表述形式是等价的。

25.4　研究任何一个你能得到的对象系统。这一系统支持什么编程语言？它是否支持查询语言？如果支持的话，是什么？依你的观点，此对象系统是否比传统的 SQL 功能更强大？其字典是怎样的？用户如何查询字典？有视图支持吗？如果有，其支持是否广泛？（例如，是否支持视图更新？）如何处理“空缺信息”？

25.5　设计一个由供应商和零件构成的对象数据库。注意：这一设计是下面习题 25.6 ~ 25.8 的基础。

25.6　为你的对象数据库写一组 OPAL 数据定义声明。

25.7　为你的对象数据库设计必要的“数据库扩充”的方法。

---

⊖ 有些人认为类型继承是一个好的想法。我们同意，但是我们坚持认为支持继承与支持对象本身无关。

25.8　在你的对象数据库中，对于下列查询写出 OPAL 代码：（a）得到所有伦敦的供应商；（b）得到所有红色的零件。

25.9　再次考虑教育数据库。列出下列操作所涉及的内容：（a）删除一个报名者；（b）删除一个雇员；（c）删除一门课程；（d）删除报名类；（e）删除雇员类。你可以假设存在类似 OPAL 中的垃圾收集进程。将有关这些问题的所有假设列出来，如采用级联删除等。

25.10　假设供应商－零件－工程数据库的对象版由一个单一的包含层次构成。一共可能有多少种这样的层次？哪种最好？

25.11　考虑对供应商－零件－工程数据库进行一些变化，不提供某供应商向某工程提供某零件这样的记录，而只记录：（a）某供应商提供某种零件；（b）某种零件供给某个工程；（c）某个工程由某供应商供货。这样会有多少种对象设计（存在或不存在包含层次）？

25.12　考虑 25.5 节所讨论的影响性能的因素，哪些是对象系统所特有的？为什么？

25.13　典型的对象系统通过方法以过程化的方式支持完整性约束；而主要的例外在于参照完整性一般是以声明的方式来实现的。过程化支持的优点是什么？你认为对参照完整性约束采用不同策略的原因是什么？

25.14　解释反变量的概念。

## 参考文献

参考文献［25.5］、［25.10］、［25.23］和［25.31］是关于对象主题及相关问题讨论的书籍。参考文献［25.29］、［25.30］和［25.42］是研究论文集。参考文献［25.24］、［25.35］、［25.38］是有关的教程。参考文献［25.4］、［25.8］和［25.21］描述了特定的系统。

［25.1］　Malcolm Atkinson *et al.*：“The Object-Oriented Database System Manifesto,” Proc. 1 st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N. Y.: Elsevier Science (1990).

最早试图对“对象模型”中包括的内容做统一构建的文章之一。文中认为下列内容是对象系统必须具备的特性——即某个号称“面向对象”的 DBMS 所必须支持的特性：

1）集合　2）对象 ID　3）封装　4）类型或类　5）继承　6）迟绑定　7）计算的完整性　8）用户定义类型　9）持久性　10）大型数据库　11）并发　12）恢复　13）特定查询

文中同时还讨论了某些可供选择的特性，包括多继承和编译阶段的类型检查；某些开放的特性，包括“编程模式”（如函数的和强制性的）；某些尚未达成一致的特性，包括视图和完整性约束。

注意：参考文献［3.3］和［25.28］中都评论了这篇论文。其中［3.3］认为：既然论文的目的是为了定义一个性能优秀的、真正通用化的 DBMS，那么对于一个可能在特定应用如 CAD/CAM 中表现良好、然而却连完整性约束都不支持的 DBMS 来说，这还是一般意义上的 DBMS 吗？

这个文献在第 20 章的参考文献［20.2］中也出现了，在那有更多的注释。

［25.2］　Malcolm P. Atkinson and O. Peter Buneman：“Types and Persistence in Database Programming Languages,” *ACM Comp. Surv. 19*, No. 2 (June 1987).

本文是最早认为持久性应该与类型无关的论文之一。这篇论文是阅读数据库编程语言领域文章的很好开端（“数据库编程语言”被许多人认为是构建对象系统的必要条件——例如，文献［25.10］和［25.11］即如此认为）。

［25.3］　François Bancilhon：“A Logic-Programming/Object-Oriented Cocktail,” *ACM SIGMOD Record 15*, No. 3 (September 1986).

论文的介绍中称：“面向对象的方法……看起来在一些新型的应用如 CAD、软件工程和人工智能方面尤其适合。然而，此方法对关系数据库技术本质上的扩充是……逻辑编程模式而非面向对象的模式。这两种模式是否能够兼容？”文中谨慎地认为两者是可以兼容的。注意：参考文献［25.40］中对此持相反观点。

［25.4］　J. Banerjee *et al.*：“Data Model Issues for Object-Oriented Applications,” *ACM TOOIS* (*Transactions*

*on Office Information Systems*) 5, No. 1 (March 1987). Republished in Michael Stonebraker (ed.), *Readings in Database Systems* (2d ed.). San Mateo, Calif.: Morgan Kaufmann (1994). Also republished in reference [25.42].

[25.5] Douglas K. Barry: *The Object Database Handbook: How to Select, Implement, and Use Object-Oriented Databases*. New York, N. Y.: John Wiley & Sons (1996).

此书的主要观点是：如果必须处理“复杂数据”，那么我们需要对象系统而非关系系统。复杂数据的特征包括：(a) 普遍存在的；(b) 通常缺乏唯一的标识；(c) 涉及大量多对多的关系；(d) 在关系模式中通常需要使用类型代码（因为在如今的 SQL 产品中缺乏对子类型和超类型的直接支持）。注意：作者任“对象数据管理小组 (ODMG)”[25.11] 的执行主管。

[25.6] David Beech: "A Foundation for Evolution from Relational to Object Databases," in J. W. Schmidt, S. Ceri, and M. Missikoff (eds.), *Extending Database Technology*. New York, N. Y.: Sprinaer Verlag (1988).

这篇文章是有关讨论将 SQL 扩展为“对象 SQL”或“OSQL”是否可能的论文之一（然而那些“对象 SQL”与传统的 SQL 差别很大）。参考文献 [25.32] 对这篇论文的观点做了更详细的讨论。

[25.7] Anders Björnerstedt and Christer Hultén: "Version Control in an Object-Oriented Architecture," in reference [25.30].

许多应用程序需要给定对象提供不同**版本**的概念；此类应用的例子包括软件开发、硬件设计、文档创建等。一些对象系统直接支持这一概念（虽然事实上这与我们处理的到底是对象系统还是别的系统并无关系）。典型的支持包括：

- 创建给定对象新版本的能力。典型的方式为：得到对象的副本并将其从数据库移到用户的私有工作区，在此工作区中可以在较长时间内（如几个小时或几天）保持或修改对象。
- 将给定对象版本转换成当前数据库版本的能力。典型的方式为：将对象从用户的工作区移回到数据库中（可能需要一些机制来**合并**不同的对象版本）。
- **删除**或存档过时版本的能力。
- 查询给定对象**版本历史**的能力。

注意：（如图 25-7 所示）：版本历史并不一定是线性的（图中版本 V. 2 产生两个不同的版本 V. 3a 和 V. 3b，最后合并为版本 V. 4）。

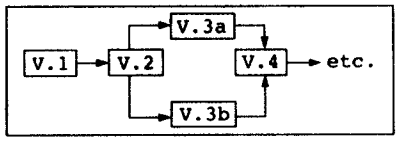



图 25-7　典型的版本历史

其次，因为典型的对象之间以各种不同的方式相互联系，版本的概念导致了布局的概念。一个**布局**就是相互联系的对象一致版本的集合。布局一般支持的内容包括：

- 将一个布局中某对象版本**复制**到另一个布局中的能力（例如，从一个老的布局复制到一个新的布局）；
- 将一个布局中某对象版本**移动**到另一个布局中的能力（即将对象加入新的布局并从老的布局中删除）。

从内部来看，这些操作主要就是许多指针的变换。但是这些对于语言的语法、语义，特别是特定查询来说有许多启示。

[25.8] Paul Butterworth, Allen Otis, and Jacob Stein: "The GemStone Object Database Management System," *CACM 34*, No. 10 (October 1991).

[25.9] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton: "The OO7 Object-Oriented Database Benchmark," Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data, Washington, D. C. (May 1993).

[25.10] R. G. G. Cattell: *Object Data Man agement* (revised edition). Reading, Mass.: Addison-Wesley (1994).

第一本探讨在数据库中应用对象技术的完整教程。以下这段摘要表明在这一领域要达到一致状态还有很长的路要走：“编程语言可能还需要新的语法……暗转技术、复制技术和新的访问方法仍需要研究……需要新的终端用户和应用开发工具……更强大的查询语言还需不断发展……在并发控制方面需要有新的研究……时间戳和基于对象的并发语义需要进一步探

讨……需要有性能模型……在知识管理方面的工作需要与对象和数据管理能力结合考虑……这将导致复杂的优化问题……研究人员缺乏必要的专业知识……对象数据库的联合需要进一步研究。"

[25.11] R. G. G. Cattell and Douglas K. Barry (eds.): *The Object Data Standard: ODMG 3.0*. San Francisco, Calif.: Morgan Kaufmann (2000).

ODMG 一般是指"对象数据管理小组"所提出的一系列提案；这是一个由"几乎覆盖对象 DBMS 业所有成员公司"的代表所组成的社团。提案由**对象模型**、**对象定义语言**（ODL）、**对象交换格式**（OIF）、**对象查询语言**（OQL）以及这些内容与 C++、Smalltalk 和 Java 之间的绑定构成（提案中并没有关于"对象处理语言"的内容；相反，对象处理能力由 ODMG 所绑定的语言提供）。

对于 2.0 版本的 ODMG 对象模型（其实和 3.0 版本没有太大区别）的分析和评价可以参考文献［3.3］中的附录 I，也可以在文献［25.28］中找到。

[25.12] R. G. G. Cattell and J. Skeen: "Object Operations Benchmark," *ACM TODS 17*, No. 1 (March 1992).

[25.13] George Copeland and David Maier: "Making Smalltalk a Database System," Proc. 1984 ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (June 1984). Republished in Michael Stonebraker (ed.), *Readings in Database Systems* (2d ed.). San Mateo, Calif.: Morgan Kaufmann (1994).

文中描述了为创建 GemStone 的 OPAL 对 Smalltalk［25.23］所做的一些改进。

[25.14] O. J. Dahl, B. Myhrhaug, and K. Nygaard: *The SIMULA 67 Common Base Language*, Pub. S22, Norwegian Computing Center, Oslo, Norway (1970).

SIMULA 67 是为进行模拟方法应用而设计的语言，而对象编程语言即来自于这种语言。事实上，SIMULA 67 可以说是第一个对象语言。

[25.15] C. J. Date: "An Optimization Problem," in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

[25.16] C. J. Date: "Why the 'Object Model' Is Not a Data Model," in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).

文中主张无论"对象模型"由什么构成，与关系模型比较起来它就像合成金属一样缺乏抽象，因而也缺少数据独立性；实际上，它更像存储模型，而不是数据模型。

[25.17] C. J. Date: "Object Identifiers *vs.* Relational Keys," in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).

文中主张正像用户所看到的 OIDs 在数据模型中没有地位。注意：支持这个说法的一个重要的论点在这篇文章中没有提到，但却是应该接受的论点，在第 26 章 26.6 节中会提到。

[25.18] C. J. Date: "Encapsulation Is a Red Herring," *DBP&D 12*, No. 9 (September 1998).

在本章中我们曾说过封装意味着独立性。然而，我们同时也指出，最好不使用术语"封装"（而使用"标量"）。部分原因在于"封装对象"并不比没有封装的关系提供更多的数据独立性（至少从原理上来讲是这样）。例如，在基表中用笛卡尔坐标（*X*, *Y*）表示的点，在物理上完全可以采用极坐标中 R 和 θ 来存储。

[25.19] C. J. Date: "Persistence *Not* Orthogonal to Type," *http://www.dbpd.com* (October 1998). 文中强烈反对权威说法："持久性与类型是无关的"［25.2］，并且支持"信息原理"的观点。

[25.20] C. J. Date: "Decent Exposure," *http://www.dbpd.com* (November 1998).

[25.21] O. Deux *et al.*: "The O2 System," *CACM 34*, No. 10 (October 1991).

[25.22] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran, and Joëlle Malec: "Schema and Database Evolution in the O2 Object Database System," Proc. 21st Int. Conf. on Very Large Data Bases, Zurich, Switzerland (September 1995).

见参考文献［25.34］中的评述。

[25.23] Adele Goldberg and David Robson: *Smalltalk-80: The Language and Its Implementation*. Reading, Mass.: Addison-Wesley (1983).

书中对"Xerox Palo Alto 研究中心"在设计和构建 Smalltalk-80 系统方面所做的早期努力给出了权威性的评述。书中的第一部分对 Smalltalk-80 编程语言做了一个详细的描述，这些正是 GemStone 的 OPAL 语言的基础。

[25.24] Nathan Goodman: "Object-Oriented Database Systems," *InfoDB 4*, No. 3 (Fall 1989).

引用这篇论文中的一段话："在现阶段将对象系统和关系系统进行比较是徒劳无益的。我们要比较的应该是对等的东西，如苹果与苹果的比较、梦想与梦想的比较、理论与理论的比较、成熟产品与成熟产品的比较……关系方法已经存在了很长时间：它有着牢固的理论基础并作为大量成熟产品的基础。相反，对象方法是较新的方法（至少在数据库领域），它并没有像关系模型一样完善的理论基础，也没有可称为是成熟的产品。所以，在真正思考对象技术是否能代替关系技术之前还有很多事需要做。"

这段话中的一些论点现在看来仍然是适合的。由于自 1989 年以来许多问题更趋于具体化，所以在一些方面关系和对象的比较已可以被认为是苹果和橘子的比较了。在第 26 章中将看到这一点。

[25.25] Nathan Goodman："An Object-Oriented DBMS War Story：Developing a Genome Mapping Database in C++，"见参考文献 [25.29]。

这篇论文中支持了本章所提出的一些批评意见。摘要如下："与传统思想相反，我们的经验告诉我们，在数据库对象内部通过实现复杂程序的方法来扩充数据库是一个错误。经验同时告诉我们，用 C++ 来实现数据库是不合适的，产生的问题包括：缺乏定义属性机制，缺乏系统引用对象机制，没有垃圾收集器，继承模型中存在细微的陷阱。我们还发现，目前基于 C++ 开发的 DBMS 缺乏重要的功能。为了弥补这一点，我们致力于提出我们自己的标准 DBMS 功能的简单实现方式：事务日志、多线程事务监视、查询语言和查询处理器、存储结构。事实上，我们将 C++ 实现的 DBMS 作为面向对象的存储管理者，而在它的上层再构建一个数据管理系统来管理大规模的基因组映射。"

[25.26] H. V. Jagadish and Xiaolei Qian："Integrity Maintenance in an Object-Oriented Database，" Proc. 18th Int. Conf. on Very Large Data Bases，Vancouver，Canada（August 1992）。

提出了一种对象系统的声明完整性机制，并论述了完整性约束编译器如何将必要的完整性检查代码合并到相关对象类的方法中。

[25.27] Michael Kifer，Won Kim，and Yehoshua Sagiv："Querying Object-Oriented Databases." Proc. 1982 ACM SIGMOD Int. Conf. on Management of Data，San Diego，Calif.（June 1992）。

提出了另一种"对象 SQL"，称为 XSQL。

[25.28] Won Kim："Observations on the ODMG-93 Proposal for an Object-Oriented Database Language，"*ACM SIGMOD Record 23*，No. 1（March 1994）。

[25.29] Won Kim（ed.）：*Modern Database Systems：The Object Model，Interoperability，and Beyond.* New York. N. Y.：ACM Press/Reading，Mass.：Addison-Wesley（1995）。

[25.30] Won Kim and Frederick H. Lochovsky：*Object- Oriented Concepts，Databases，and Applications.* Reading，Mass.：ACM Press/Addison-Wesley（1989）。

[25.31] Mary E. S. Loomis：*Object Databases：The Essentials.* Reading，Mass.：Addison-Wesley（1995）。

[25.32] Peter Lyngbaek *et al.*："OSQL A Language for Object Databases，"Technical Report HPLDTD-91-4，Hewlett-Packard Company（January 15，1991）。

见参考文献 [25.6] 中的评述。

[25.33] Bertrand Meyer："The Future of Object Technology，"*IEEE Computer 31*，No. 1（January 1998）。

文中称："对象数据库的未来是一个值得探讨的主题……从 1986 年开始关系数据库厂商设法通过抢先通告的方式抑制了对象数据库的进一步发展……十年以后，可能对象数据库方面的专家仍会告诉你主要关系数据库厂商提供的产品与现实事物仍然相差很远……对象数据库市场份额将继续增长，但只占一部分。"

[25.34] John F. Roddick："Schema Evolution in Database Systems-An Annotated Bibliography，"*ACM SIGMOD Record 21*，No. 4（December 1992）。

传统数据库产品一般对一个已存在的模式只提供简单的改动（例如，在一个已存在的基表中加入某个新的属性）。但是，某些应用往往需要更复杂的模式变化支持，而一些对象原型系统在这一问题上进行了深入的分析。注意：这一问题在对象环境下更为复杂，因为这时相关的模式变得更为复杂。

下列可能的模式变化的分类来自于关于对象原型系统 ORION [25.4] 的一篇论文。我们认为其中的一些内容违背了模型与实现相区分的原则。

- 对象类的改变

1）实例变量的改变
- 增加实例变量
- 删除实例变量
- 重命名实例变量
- 改变实例变量的缺省值
- 改变实例变量的数据类型
- 改变实例变量的继承源

2）方法的改变
- 增加方法
- 删除方法
- 重命名方法
- 改变方法的内部代码
- 改变方法的继承源

■ 类层次的改变（假设为多继承的情况）
- 在 *B* 类的超类集合中加入 *A* 类
- 从 *B* 类的超类集合中删除 *A* 类

■ 整个模式的改变
- 增加类
- 删除类
- 重命名类
- 划分类
- 合并类

由于系统不支持视图，所以我们并不清楚上述改变的透明度如何。事实上，对于对象系统来说，因为其本质上的3GL特性使“模式进化”的可能性很成问题。如参考文献［25.28］中所述：“如果……索引数量改变或将数据重组成不同的聚集方式，作为方法不能自动利用这种改变所带来的好处”。

进一步说，模式进化在对象系统中也更为需要，因为在对象系统中模式设计时的许多决定是由应用开发的程序员来决定的，而在关系系统中这些都是由DBA甚至是DBMS本身来决定的（参见文献［25.35］）。特别地，性能调优时就需要重设计原有的模式（参见文献［25.35］）。

［25.35］ C. M. Saracco:“Writing an Object DBMS Application”(in two parts),*InfoDB 7*,No. 4 (Winter 1993/94) and *InfoDB 8*,No. 1 (Spring 1994).

文中给出了一些简单但完整的编码例子。

［25.36］ Gail M. Shaw and Stanley B. Zdonik:“A Query Algebra for Object-Oriented Databases,” Proc. 6th IEEE Int. Conf. on Data Engineering (February 1990).

这篇论文支持了笔者本人的观点，即“对象代数”内在是复杂的（因为对象本身就是复杂的）。具体来说，任意嵌套层次对象的等值比较往往需要非常仔细的考虑。在此文的观点背后所包含的基本思想是：每个查询操作产生一个关系，而关系中每条元组包含某些数据库对象的OID；这样，在连接操作的情况下，由于元组并不从成员对象中继承任何方法，所以真正比较的只能是元组中OID所指向的对象本身。

［25.37］ David W. Shipman:“The Functional Data Model and the Data Language DAPLEX,”*ACM TODS 6*, No. 1 (March 1981). Republished in Michael Stonebraker (ed.), *Readings in Database Systms* (2d ed.). San Mateo,Calif.:Morgan Kaufmann (1994).

近年来，有一些试图将系统构建在函数而非关系上的尝试，而DAPLEX系统是其中最知名的一个。我们之所以在这里提到这一系统，是因为函数化的方法与对象的方法在某些思想上有类似之处，包括导航式（即路径跟踪）地获得函数中与其他对象相联系对象的地址。注意：这里所指的函数并不是数学意义上的函数，而可能返回多个值。事实上，函数上的许多概念都必须打破，以适应“函数数据模型”中所需要具备的各项功能。

［25.38］ Jacob Stein and David Maier:“Concepts in Object-Oriented Data Management,” *DBP&D 1*, No. 4 (April 1988).

本文是关于对象概念的不错的早期教程，两位作者是GemStone的设计者。

[25.39] D. C. Tsichritzis and O. M. Nierstrasz:"Directions in OO Research,"见参考文献[25.30].

这篇论文进一步支持了笔者关于对象数据库技术还有很长的路要走的观点，文中谈到："在基本的定义上还存在分歧；例如，对象是什么？……认为在科学探索的动态时期一些松散的定义是不可避免甚至是受欢迎的观点显然是毫无根据的。在这一时期所得出的各项定义应该更为严格才对"。然而对象系统的概念已经存在了将近 40 年！——事实上这些概念的提出比关系模型的提出还早。

[25.40] Jeffrey D. Ullman:"A Comparison Between Deductive and Object-Oriented Database Systems,"Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases, Munich, Germany (December 1991); C. Delobel. M. Kifer, and Y. Masunaga (eds.), *Lecture Notes in Computer Science 566*. New York, N. Y.:Springer-Verlag (1992).

虽然对于这篇论文的一些细节观点我们并不同意，但对于其整体结论我们是非常赞同的；即"演绎"（即基于逻辑的）数据库系统从长期来讲比对象系统更精确。同时，这篇论文对于优化也提出了一个重要观点：

假设我们定义"存在一个类似二进制关系的对象类，类中有一个实现连接操作的方法，这样我们就可以写出类似 R JOIN S JOIN T 之类的表达式。似乎我们可以通过（R JOIN S）JOIN T 或 R JOIN（S JOIN T）的顺序来计算其结果，但事实是这样吗？我们并不知道 JOIN 方法的具体内容，要是它相互关联怎么办？……结论是，如果在关系或关系以上的层次上通过对象的方式来编写应用，那么必须在关系代数中加入相关的信息。系统本身是无法演绎这些信息的，只有通过后续的构建。这样，在查询语言中唯一能够优化的只有那些内置的方法，因为只有这些方法的语义才已经为系统所知。"

[25.41] Carlo Zaniolo:"The Database Language GEM,"Proc. 1983 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1983). Republished in Michael Stonebraker (ed.), *Readings in Database Systems* (2d ed.). San Mateo, Calif.:Morgan Kaufmann (1994).

GEM 即为"通用实体处理器"（General Entity Manipulator）。它是对支持关系中包含集合属性和可选择属性（例如，拥有优先权的 EMP 有 SALARY 属性，而无优先权的 EMP 有 HOURLY_WAGE 和 OVERTIME 属性）的 QUEL 系统的有效扩充。它也利用了对象在概念上可以包含其他对象的思想（而不是通过外码来引用其他对象）。同时，它还通过扩充我们所熟悉的圆点限定的方法为属性中对象的引用提供了一种简单的方式——实际上是通过隐式地遍历某些预定义的连接路径。例如，我们可以通过限定名 EMP. DEPT. BUDGET 的形式来引用给定雇员所在部门的预算。许多系统采用了这一思想或在这一思想的基础上进行了扩充。

[25.42] Stanley B. Zdonik and David Maier (eds.):*Readings in Object-Oriented Database Systems*. San Francisco, Calif.:Morgan Kaufmann(1990).

# 第 26 章　对象/关系数据库

## 26.1　引言

在 20 世纪 90 年代末，一些厂商已经发布了“对象/关系”DBMS 产品（在市场上被称为是通用服务器）。例如 DB2 的通用数据库版，Informix 动态服务器的通用数据选件以及 Oracle 通用服务器（也用过其他名字）。在这些产品中主要的思想是产品能够同时支持对象和关系的性能；换句话说，这些产品试图将这两种技术结合起来。

作者认为这种结合应该把关系模型作为坚实的基础（毕竟如本书第二部分中所谈到的，关系模型是现代大多数数据库技术的基础）。所以，我们希望的是能够在关系系统中融入对象系统的特性[○]——至少是好的特性（我们当然不希望完全放弃整个关系系统，也不希望将这两种系统分开处理）。其他许多作者也支持这一观点，包括《第三代数据库系统宣言》［26.44］的作者；他们认为第三代 DBMS 必须包容第二代 DBMS。（在这里第一代 DBMS 是指关系以前的数据库系统，如 IMS；第二代 DBMS 是指 SQL 系统，而第三代 DBMS 是指未来的系统。）然而，某些对象系统的作者并不支持这一观点。这里引用一段较为代表性的话［26.7］：

*在计算机科学中已经产生了许多代数据管理方式，从开始的索引文件到后来的网状和层次 DBMS……，再到最近的关系 DBMS……。目前我们正处在新一代数据库系统的开始点……，这种系统提供对象管理并支持更复杂的数据。*

在这里作者明确地指出：就像关系系统代替更早的层次和网状系统那样，对象系统也将代替关系系统。

我们不同意这一观点的理由在于与更早的系统模型相比，关系系统是完全不同的［26.17］，它并不针对特定的环境；而老的层次和网状系统却是基于特定环境的，它们也许能解决那个时期某些重要的问题，但却不具备牢固的理论基础。然而不幸的是，早期关系系统的支持者们（包括作者本人在内）对关系和关系以前的系统的优缺点的比较存在着很大的分歧；这些争论在当时是必要的，但是实际上却起了不良的后果，使人觉得关系和关系以前的 DBMS 在本质上是同一事物。这一错误的思想支持了以上引文中的观点——关系系统到对象系统与层次和网状系统到关系系统相类似。

对象系统到底是什么？它们是基于特定环境的吗？《面向对象数据库系统宣言》［20.2，25.1］中对此做了很好的说明：“关于系统的详细规范，我们采用一种达尔文方式：我们希望通过构建一系列实验原型，自然而然生成适当的对象模型”。换句话说，作者认为我们不需要预定义模型，而是应该先编写代码并构建系统来看看结果到底如何！

在下文中，我们将始终贯彻我们的观点（事实上，大多数主要 DBMS 卖家都是这么做），即通过在关系系统中加入对象技术的优秀特性来增强其性能。再次声明，毕竟关系技术已经研究了近 35 年，我们绝不希望完全放弃这一技术。

在第 25 章中我们已经论证——参见［26.31］中的相关叙述——面向对象只包含一种好的思想，那就是**适当的数据类型支持**（或者说是两种好的思想，如果将类型继承单独算的话）。所以我们遇到的问题是：如何将适当数据类型支持这一特性融入到关系模型中？当然，答案很简单，这种支持已经以域的形式存在了（我们仍倾向于将其称为类型）。换句话说，为了加入对象功能我们并不需要在关系模型中增加任何东西，除了将其完整地实现；而现在大多数 SQL 系统

---

○ 注意：我们感兴趣的是演变，而不是革命性的变化。相反地，考虑［25.11］中的一段引用：“［对象 DBMS］是一种革命性的发展而不是演变”。我们认为市场还没有对革命性的变化做好准备，也认为不需要也不应该进行革命性的变化——这就是为什么第三次宣言［3.3］非常明确指出要演变，而不要革命性的变化的一个原因。

在实现上都不是很成功。[1]

所以，我们认为一个真正支持域的关系系统应该能够处理所有那些所谓的对象系统才能够处理而关系系统处理不了的数据类型：多媒体数据、时间序列数据、生物数据、金融数据、工程设计数据、办公自动化数据，等等。同时，我们还认为一个真正的“对象/关系”系统首先应该是一个关系系统——也就是说，支持关系模型及其相关的所有内容。应该鼓励 DBMS 厂商去开发那些他们正努力在开发的东西，即在他们的系统中包含对类型和域的适当支持。事实上，可以论证对象系统之所以吸引人的主要原因就在于 SQL 厂商没能充分支持关系模型；因此这绝不能作为放弃关系系统的原因！

现在让我们来完成第 25 章中尚未完成的例子，看看对于矩形问题如何提出一个很好的关系解决方案。（我们已经在 **Tutorial D** 中给出了解决方案；产生 SQL 语句留作练习）对此问题的解决首先需要定义一个矩形类型 RECT：

```
TYPE RECTANGLE POSSREP ( X1 RATIONAL, Y1 RATIONAL,
                         X2 RATIONAL, Y2 RATIONAL ) ... ;
```

我们假设矩形在物理上通过某种能有效支持空间数据的方式存储——如四叉树（quadtree）或 R 树等 [26.37]。

我们同时定义一个用来测试两个给定矩形是否重叠的操作符：

```
OPERATOR OVERLAP ( R1 RECTANGLE, R2 RECTANGLE )
                                    RETURNS BOOLEAN ;
   RETURN ( THE_X1 ( R1 ) ≤ THE_X2 ( R2 ) AND
            THE_Y1 ( R1 ) ≤ THE_Y2 ( R2 ) AND
            THE_X2 ( R1 ) ≥ THE_X1 ( R2 ) AND
            THE_Y2 ( R1 ) ≥ THE_Y1 ( R2 ) ) ;
END OPERATOR ;
```

此操作符在实现上采用了重叠测试的有效方式（详见第 25 章中的相关叙述），同时采用了有效的存储结构（R 树或其他等）。

现在用户可以来创建一个基表，表中包含某个以 RECT ANGLE 作为类型的属性：

```
VAR RECTANGLES BASE RELATION { R RECTANGLE, ... } KEY { R } ;
```

这样，对于查询“得到所有与单位正方形相重叠的矩形”来说就很简单了：

```
RECTANGLES
WHERE OVERLAP ( R, RECTANGLE ( 0.0, 0.0, 1.0, 1.0 ) )
```

这一解决方案克服了第 25 章中讨论的所有缺陷。

目前，上述的观点在概念上是相当简单明了的——再重复一下，所有我们所必须做的事是实现关系模型，特别是让用户定义他们自己的类型——可是（再一次）混乱占优势……事实上，我们认为在现有的商用数据库产品中至少存在两个大的错误（就是我们之前提及的两个大的错误[2]）。明显地，我们需要讨论那些错误并且要特别说明为什么我们认为它们是错误的；因此，本章的其余部分组织如下：26.2 节和 26.3 节分别讨论两个根本性错误（至少其中一个错误出现在市场上几乎每一个对象/关系产品中）。26.4 节考虑了对象/关系系统某些实现上的问题，26.5 节描述了一个真正对象/关系系统所具有的好处（也就是说，绝不会犯那两个根本性错误）。最后，

---

[1] 尤其是，现在的系统引起了很普遍的误解，即：关系系统只支持有限的简单类型。下面的引用很典型：“关系系统支持很少的固定的数据类型集（例如：整型、日期型、字符串型）”[26.34]；“关系 DBMS 只能支持……它的内置类型（基本的数字型、字符串型、日期型、时间型）”[25.31]；“对象/关系数据模型通过提供更丰富的类型系统扩展了关系数据模型”[16.21]；等等。

[2] 某评论家反对在这里使用大错（blunder）这个词，这一反对是正确的，因为它不是课本中可以找到的一个普通的术语。我们承认我们选择它部分理由是由于它的震撼价值。但是如果一些系统 X 本应该是关系模型的一个运用，但是后来——在关系模型被首次定义的 25 年后——有人加上了一个“特性”到系统 X，使它完全违反了那个模型的规则，所以可见我们把引入那个“特性”称为是一个大错是十分合理的。

26.6 节描述了相关的 SQL 支持工具，26.7 节给出本章的小结。

## 26.2　第一个根本性错误

首先引用参考文献［3.3］中的一段话：

*在具体考虑如何将对象和关系结合的问题之前，首先需要解决一个关键性的问题，即：*

*对象世界中的概念“对象类”在关系世界中对等的概念到底是什么？*

*这个问题之所以非常重要，其原因就在于对象类是对象世界中所有概念的基础——所有其他的对象概念或多或少都依赖于它。对于这个问题通常有两个等式作为其答案的候选：*

- *域 = 对象类*
- *关系变量 = 对象类*

*我们现在来证明，第一个等式是正确的，而第二个是错误的。*

事实上，既然对象类和域实质上都是类型，那么第一个等式显然是正确的，给定关系变量是变量，类是类型，则第二个等式显然是错误的（变量和类型绝非同一事物）。正是基于这一原因，在《第三次宣言》［3.3］中坚持认为关系变量不是域。然而，许多人和一些产品却在事实上支持了第二个等式——我们认为这是根本性错误（或如我们前面所说的，第一个根本性错误）。仔细讨论这一问题是非常必要的。注意：本节中的大部分内容直接引自参考文献［3.3］。

为什么会有人犯这样的错误呢？考虑下列简单的类定义。其中的对象语言是虚构的，并且我们有意不与 25.3 节中的表达相一致：

```
CREATE OBJECT CLASS EMP
     ( EMP#       CHAR(5),
       ENAME      CHAR(20),
       SAL        NUMERIC,
       HOBBY      CHAR(20),
       WORKS_FOR CHAR(20) ) ... ;
```

（这里的 EMP#、ENAME 等都是公共实例变量。我们故意将它们定义为简单的固定类型而不是用户定义类型；而且，为了简化，这一章中的所有例子都作同样的处理）。现在来考虑一下 SQL 语句对“基表”的定义：

```
CREATE TABLE EMP
     ( EMP#       CHAR(5)  NOT NULL,
       ENAME      CHAR(20) NOT NULL,
       SAL        NUMERIC  NOT NULL,
       HOBBY      CHAR(20) NOT NULL,
       WORKS_FOR CHAR(20) NOT NULL ) ... ;
```

这两个定义看起来的确很类似，将两者等同起来的思想也的确很诱人。某些系统，包括一些商业产品，事实上就是这么做的。现在让我们进一步思考这一问题，或更准确地说，在以上 CREATE TABLE 声明的基础上再增加一些扩充以使其更为“对象化”。注意：以下的讨论基于某一特定的商用产品；其实就是基于此产品文档中的一个例子。我们并不想指出这个产品的名称，因为批评或表扬某一产品并不是本书的目的。我们所提出的批评针对所有支持等式“关系变量 = 对象类”的系统。

第一个扩充为：允许加入复合（即元组值）属性；也就是说，我们允许属性的值可以是其他表中的元组（或本表中的元组）。例如，将原先的 CREATE TABLE 声明替换成如下声明的集合（参见图 26-1）：

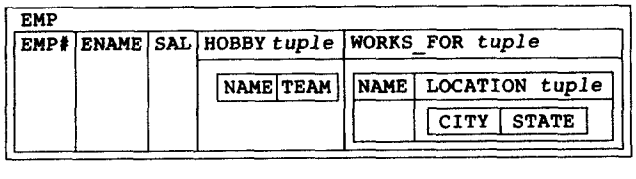



图 26-1　属性包含元组（的指针）——不推荐使用

```
CREATE TABLE EMP
     ( EMP#       CHAR(5)    NOT NULL,
       ENAME      CHAR(20)   NOT NULL,
       SAL        NUMERIC    NOT NULL,
       HOBBY      ACTIVITY   NOT NULL,
       WORKS_FOR COMPANY     NOT NULL ) ;
CREATE TABLE ACTIVITY
     ( NAME       CHAR(20)   NOT NULL,
       TEAM       INTEGER    NOT NULL ) ;

CREATE TABLE COMPANY
     ( NAME       CHAR(20)   NOT NULL,
       LOCATION   CITYSTATE  NOT NULL ) ;

CREATE TABLE CITYSTATE
     ( CITY       CHAR(20)   NOT NULL,
       STATE      CHAR(2)    NOT NULL ) ;
```

解释：表 EMP 中的属性 HOBBY 被定义为 ACTIVITY 类型，而 ACTIVITY 是含有两个属性 NAME 和 TEAM 的表。其中 TEAM 给出 NAME 所指定的运动类型中每队运动员的数目：例如，一个元组可能为（Soccer，11）。这样，每个 HOBBY 值实际上由一对值，NAME 值和 TEAM 值构成，（确切地说这对值目前以关系变量 ACTIVITY 中的元组值形式出现）。注意：我们已经触犯了《对象关系数据库基础：第三版宣言》中关系变量不是域的声明——属性 HOBBY 的“域”被定义为关系变量 ACTIVITY。在本节中还将讨论这一问题。

类似地，关系变量 EMP 中的属性 WORKS_FOR 声明为类型 COMPANY，而 COMPANY 也是一个包含两个属性的关系变量，其中一个属性的类型为 CITYSTATE，而 CITYSTATE 又是一个包含两个属性的关系变量。换句话说，关系变量 ACTIVITY、COMPANY 和 CITYSTATE 都既被认为是类型（或域），又被认为是关系变量。当然，关系变量 EMP 本身也是如此。

第一种扩充大致上与允许对象包含对象类似，从而支持了包含层次的概念（见第 25 章）。

我们将第一种扩充的特征表述为“属性包含元组”，而这正是由等式“关系变量 = 对象类”本身所带来的特征。然而，更精确地说，这一特征应该表述为“属性包含元组的指针”——我们在随后将分析这一问题（所以在图 26-1 中我们应该将三个“元组”替换成“元组的指针”）。

第二个扩充为：允许关系值属性；也就是说，属性值可以是其他表（或就是同一个表）中元组的集合。例如，假设雇员可以有不只是一个而是任意个爱好（参见图 26-2）：

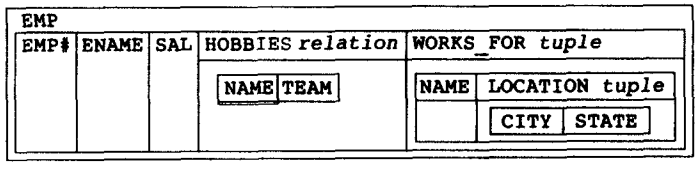



图 26-2　属性包含元组（指针）的集合——不推荐使用

```
CREATE TABLE EMP
     ( EMP#       CHAR(5)                NOT NULL,
       ENAME      CHAR(20)               NOT NULL,
       SAL        NUMERIC                NOT NULL,
       HOBBIES    SET OF ( ACTIVITY )    NOT NULL,
       WORKS_FOR COMPANY                 NOT NULL ) ;
```

解释：现在关系变量 EMP 中任意元组的 HOBBY 值可以是空值或多个（NAME，TEAM）序列对，即一系列关系变量 ACTIVITY 中的元组。第二种扩充大致与允许对象包含“集合”对象类似：包含层次的更为复杂的版本。注意：在我们所基于的特定产品中，集合对象可以是序列、无序单元组或集合本身。

第三个扩充为：允许关系变量有相应的方法（即，操作符）。例如：

```
CREATE TABLE EMP
     ( EMP#       CHAR(5)                NOT NULL,
```

```
        ENAME      CHAR(20)              NOT NULL,
        SAL        NUMERIC               NOT NULL,
        HOBBIES    SET OF ( ACTIVITY )   NOT NULL,
        WORKS_FOR COMPANY                NOT NULL )
METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

解释：方法 RETIREMENT_BENEFITS 将某个 EMP 元组作为参数，并产生类型为 NUMERIC 的结果值。

最后一个扩充为：允许子类。例如（参见图 26-3）：

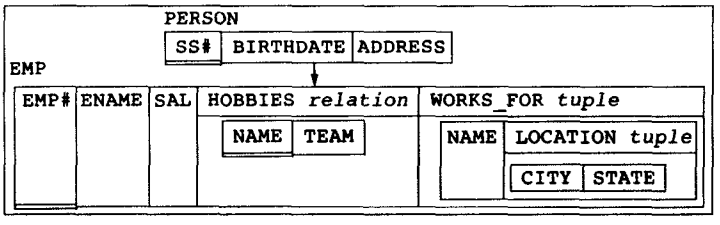



图 26-3　将关系变量作为子类或超类——不推荐使用

```
    CREATE TABLE PERSON
         ( SS#       CHAR(9)  NOT NULL,
           BIRTHDATE DATE     NOT NULL,
           ADDRESS   CHAR(50) NOT NULL ) ;
CREATE TABLE EMP
    AS SUBCLASS OF PERSON
     ( EMP#       CHAR(5)               NOT NULL,
       ENAME      CHAR(20)              NOT NULL,
       SAL        NUMERIC               NOT NULL,
       HOBBIES    SET OF ( ACTIVITY )   NOT NULL,
       WORKS_FOR COMPANY                NOT NULL )
METHOD RETIREMENT_BENEFITS ( ) : NUMERIC ;
```

解释：现在关系变量 EMP 有三个从关系变量 PERSON 中继承的附加属性（SS#、BIRTHDATE 和 ADDRESS）（因为每个 EMP 实例同样也是一个 PERSON 的实例[⊖]）。如果关系变量 PERSON 有方法，也将继承过来。注意：这里的 PERSON 和 EMP 有时分别被称为超表和子表。参考文献［14.13］和 26.6 节中有对这些概念的进一步讨论和批评。

除了以上关于定义的扩展外，当然还需要某些处理上的扩展，例如：

- 路径表达式（例如，EMP. WORKS_FOR. LOCATION. STATE）。注意：这一表达式返回的可能是标量、元组或关系。同时还需注意：在上述扩充的后两种情况下，元组或关系的成员也可能是元组或关系；例如，表达式 EMP. HOBBIES. NAME 返回的即为关系。此外，这里的路径表达式下降到控制层次，而第 25 章中的路径表达式则是上升的。
- 元组和关系的值（可能有嵌套）。例如（此例并不代表实际语法）：

```
( 'E001', 'Smith', $50000,
        ( ( 'Soccer', 11 ), ( 'Baseball', 9 ) ),
                            ( 'IBM', ( 'San Jose', 'CA' ) ) )
```

- 关系比较运算符（例如，SUBSET、SUBSETEQ 等）。注意：这些具体的操作符来自我们的讨论所基于的某特定产品。在该产品中，SUBSET 真正意味着“适当的子集”，而 SUBSETEQ 只意味着“真子集”！。
- 遍历类的层次结构的操作符（例如，同时检索 EMP 和 PERSON 的信息）。注意：这里仍需非常小心。在有些情况下，查找 PERSON 信息及相关 EMP 信息的查询结果不再是关系——这就意味着关系最基本的封闭属性被打破，这将带来一系列很麻烦的问题（关于

⊖ 很抱歉我们使用了让人迷惑的术语“实例”，但是如果使用准确的术语“值”或者“变量”，那么我们想描述的模式将没有什么意义了。

这一问题，参考文献［26.41］——其将这种返回结果称为“参差不齐的返回”——仅仅提出：“客户端程序必须准备好处理这种复杂的参差不齐的返回结果”！）。

- 在诸如 SELECT 和 WHERE 子句中调用方法的能力（在 SQL 中）。
- 访问属性值为元组或关系的成员的能力。

当我们考虑等式“关系变量 = 类”如何具体实现时，马上会想到这么一大堆问题。那么这一等式到底存在什么问题呢？

首先我们注意到：关系变量是变量，而类是类型，它们怎么可能相同呢？这一点从逻辑上就足以打消“关系变量 = 类”的想法。然而，为了更进一步说明问题，我们列出了以下供参考的观点：

- 等式“关系变量 = 类”等价于进一步的等式“元组 = 对象”和“属性 =（公共）实例变量”。这样的话，（正如我们在第 25 章中看见的）一方面一个真正的对象类——至少是标量或“封装的”对象类——必定拥有方法，并且没有公共实例变量，而另一方面关系变量“对象类”却一定有公共实例变量，并且只在某些情况下才有方法（这绝对不满足“封装性”）。还是那句话，这两个概念怎么能够等同呢？
- 属性定义间有着巨大的差别，如“SAL NUMERIC”和“WORKS_FOR COMPANY”。NUMERIC 是真正的数据类型（真正原始的域）；它在属性 SAL 的值上设置了与时间无关的约束。相反，COMPANY 并不是真正的数据类型，它在属性 WORKS_FOR 上设置的约束是时间相关的（其显然依赖于关系变量 COMPANY 的当前值）。事实上，正如前面所指出的，关系变量与域的区别——或者使用对象中的术语，集合与类的区别——在这里被混淆了。
- 如我们所看到的，元组“对象”可以包含其他的“对象”；例如，EMP“对象”显然包含 COMPANY“对象”，然而，事实上它们包含的是“被包含对象”的指针，用户必须非常清楚这一点。例如，假设用户更新了某条特定的 COMPANY 元组（参见图 26-1），所有包含此 COMPANY 元组的 EMP 元组马上可以看到更新。*注意*：我们并非认为这样不好，只是这样的话，用户就必须了解内情，即了解图 26-1 中所示“模型”是不正确的——EMP 元组并不真正包含 COMPANY 元组，而只包含指向 COMPANY 元组的指针。

对于这一点还需做以下补充：

a. 我们能不能插入一条 EMP 元组、并将其包含的 COMPANY 元组值设为目前在 COMPANY 关系变量中尚未存在的元组？如果答案是能，那么对于 INSERT 操作没有任何限制，这样把 WORKS_FOR 定义成 COMPANY 就没有任何意义。如果答案是不能，那么对于 INSERT 操作将会产生不必要的麻烦——用户不得不给出整个公司的信息而不仅仅是公司名称（即外码值）。此外，给出整个公司信息意味着告诉系统一些系统本身已经知道的内容，并且在最坏情况下，本该成功完成的 INSERT 操作会由于用户在给出完整信息时的小错误而失败。

b. 假设我们希望在公司上加入 ON DELETE RESTRICT 规则（也就是说，只要公司有一个雇员，公司本身就不能被删除）。此规则必须由过程化代码来实现，如方法 *M*（注意关系变量 EMP 并没有可附加此规则声明的外码）。此外，SQL 中的标准 DELETE 操作现在也只能在关系变量 COMPANY 的方法 *M* 中实现，那么如何来实现这一规则呢？其他类似的问题也会被提出，如使用规则 ON DELETE CASCADE，等等。

c. 同时注意到：尽管 EMP 元组包含 COMPANY 元组，当删除 EMP 元组时并不“级联地”删除相应的 COMPANY 元组。

从以上几点可以看出，我们讨论的不再是关系模型。其基本的数据对象不再是仅包含值的关系，而是包含值和指针的“关系”——就关系模型而言，这已经不再是关系。换句话说，**我们损害了关系模型的概念完整性**。*注意*：概念完整性这个术语是由 Fred Brooks 提出来的，他曾这样描述概念完整性［26.3］：“概念完整性是系统设计中最重要的考虑。使一个系统忽略某个不规则的特性来反映一个设计思想集合的做法，比使一个包含许多好的、但独立、不协调的思想的系统的做法更好。”在 20 年后，他又加上：“一个清楚的好的程序产品必须呈现……一个一致的智力模型……概念完整性……是一个完全灵活使用的一个最重要的因素……现在**我比以前更相信**这一点。概念完整性是产品质量的核心。”

- 假设将关系变量 EMP 在属性 HOBBIES 上的投影定义为视图 V。V 是由基表导出的关系变量，所以，如果等式“关系变量 = 类”成立，V 也是一个类。它是什么类？类当然还有方法，在 V 上定义的方法又是什么？

“类” EMP 只有一个方法 RETIREMENT_BENEFITS，显然这一方法并不适合“类” V。事实上，应用于“类” EMP 的方法几乎都不能应用于“类” V，所以对于投影的结果没有任何可使用的方法；也就是说，无论投影的结果是什么，它肯定不是类（也许我们认为它是类，但它有公共实例变量而没有方法，我们已观察到一个真正的“封装”类有方法而没有公共实例变量）。

非常明显，当人们将关系变量与类等同时，他们只考虑到基表而忘记了由基表所导出的表（我们在上面讨论的指针是指向基表元组的指针，而非导出表）。然而，以这种方式区分基表和导出表是错误的，因为关系变量是基表还是导出表这一问题的答案从某种意义上来说是任意的（想一想第 10 章中关于可交换性原则的讨论）。

- 最后，能够支持什么域？那些支持等式“关系变量 = 类”的人对域不会有很大的认识，因为他们没能看到域对于整个模式的适应性。然而，我们知道，域是基本的。

以上所有内容可以概括如下：显然，系统可以在错误的等式“关系变量 = 类”上进行构建；事实上也确实有这样的系统存在。然而，那些系统（就像汽车在引擎中没有油或建筑在沙滩上的房子）也许能提供一些有用的服务，但最终注定是要失败的。

**第一个根本性错误从何而来**

推敲第一个**根本性错误**的来源是一件有趣的事。我们认为其根源是缺乏统一的意见（见第 25 章），它来自于对象世界中的某些术语。特别地，术语对象本身并没有一个普遍接受且意见一致的意思——这正是我们自己更不喜欢使用它的原因。

尽管这样，至少在对象编程语言领域，术语“对象”通常指更为传统的值或变量。不幸的是，此术语还用在其他领域；作为“对象分析与设计”或“对象模型”技术的一部分使用在某些语义模型领域（见 [14.3]）。显然，在那些领域中，“对象”不表示值或变量，而是数据库中通常所称的实体（与编程语言中的对象不同，这里的对象并不是封装的）。换句话说，“对象模型”实际上就是“实体/关系模型”；在参考文献 [14.3] 中或多或少承认了这一点。结果，在某些系统中标识为“对象”的事物被映射为关系变量中的元组而非域中的值。

## 26.3　第二个根本性错误

本节我们将讨论**第二个根本性错误**；正如我们将要看到的，第二个错误可以说是第一个错误在逻辑上的延续，但就其本身而言也非常关键。实际上，在避免**第一个错误**的情况下，仍有可能犯第二个错误。事实上，市场上每一个对象/关系产品都犯了第二个错误，包括 SQL 标准（参看 26.6 节）。第二个错误为：**把指针与关系混为一谈。**

为了说明这个问题，首先回顾一下采用等式“关系变量 = 类”后的主要特性。某些读者可能对上一节的内容有些疑惑，因为某些我们似乎是反对的特性在本书的前面部分曾经支持过（元组和关系值属性即为一个例子）。所以在这里就这一问题做进一步说明：

- 元组和关系值属性：事实上，我们并不反对这种属性。我们所反对的是（a）必须有这种属性的思想，尤其当其值已出现在其他（基本）关系变量中时；（b）属性的值并非元组或关系本身，而是指向元组或关系的指针的思想——这意味着，我们讨论的已根本不是元组或关系值。注意：事实上，通过指针指向元组或关系值的思想毫无意义，关于这一点下面我们还将详细讨论。
- 关系变量的相关操作符（“方法”）：我们也不反对这一思想——实质上这就是存储或触发过程的另一种表述方式。我们所反对的是将操作符与关系变量（且只与关系变量）联系而不与域或类型相联系的思想。我们也反对将操作符与特定关系变量联系起来的思想（这其实是目标运算对象的另一种表现形式）。
- 子类和超类：对此我们明确反对。在一个将关系变量与类等同的系统中，子类和超类变成

了子表和超表——这一思想非常值得怀疑（参考［14.13］和 26.6 节）。我们希望拥有第 20 章所描述的适当的继承性支持。

- 路径表达式：我们并不反对那些在参照中用于语法简写的路径表达式——例如［26.15］中提到的从外部码到相关候选码的表达形式。然而，26.2 节中讨论的路径表达式却是某些指针链的简写，对此我们是坚决反对的（因为我们首先反对指针）。
- 元组和关系名称：这是必要的——虽然需要一般化为元组和关系选择子［3.3］。
- 关系比较操作符：同样是必要的（但需要遵循正确的操作方式）。
- 遍历类层次结构的操作符：如果“类的层次”即意味着“关系变量的层次”，那么我们对此是坚决反对的，因为这样打破了关系封闭的属性（见［26.41］）。如果“类的层次”意味着第 20 章中所提到的“类型层次”，我们则表示赞成（但实际并非如此）。
- 在诸如 SELECT 或 WHERE 等子句中调用方法：当然不反对。
- 对包含元组或关系的属性值中单个成员的访问：当然不反对。

现在重点考虑将指针与关系相混淆的问题。从定义来看，问题的关键是在于，指针指向的是变量而不是值（因为只有变量才有地址而值没有）；如果关系变量 *R1* 中的属性值为指向关系变量 *R2* 的指针，这些指针指向的一定是元组变量而非元组值。但是在关系模型中并不存在元组变量，关系模型处理关系值，关系值（大致上）是元组值的集合，而元组值又（大致上）是标量值的集合。同时关系模型也处理关系变量，关系变量值为关系；然而，它并不处理元组变量或标量变量。总之，在关系模型中唯一的变量（也是关系数据库中唯一允许的变量）为关系变量。所以，将指针与关系相混淆的思想实质上是在关系模型中引进了某一全新的变量，从而构成了对关系模型的背离。正如上节中指出的，这严重损害了关系模型的概念完整性。

通过以上讨论，我们很不幸地看到，目前绝大多数对象/关系产品——甚至那些避免了**第一种根本性错误**的产品——就是因为采用了上述方式而混淆了指针和关系的区别。当 Codd 定义关系模型时，他有意排除了指针。他曾在［6.2］中写到：

*我们可以放心地假设所有用户（包括最终用户）理解值之间的比较，但能够理解指针复杂性的用户却实在不多。关系模型是建立在这一原则基础上的……即使用户能够理解指针的复杂性，对指针的处理也比值之间的比较更容易出错。*

具体来说，指针的使用将会导致指针冲突，而这正是系统容易出错的一大原因。如第 25 章中所提到的，对象系统的这一特性有时被贬低为“就像是 CODASYL 系统的重新使用”。

对这一观点更详细的论证可以参见［25.19］和［26.15］。在参考文献［26.12～26.14］和［26.17］中讨论了在数据模型中数据构造的基本思想。

**1. 指针和一个好的继承的模型是矛盾的**

事实上有另外一个强有力的观点反对支持指针，当 Codd 在写参考资料［6.2］时绝不可能意识到这个观点。我们用一个简单的例子来说明那个观点。（我们举这个例子使用的是原先的程序变量而不是数据库关系，这是为了集中到真正的问题上来而不被一些枝节问题烦扰。）再次考虑 ELLIPSE 和 CIRCLE 类型。定义 PTR_TO_ELLIPSE 和 PTR_TO_CIRCLE 为指针类型，即使变量类型 PTR_TO_ELLIPSE 和 PTR_TO_CIRCLE 分别表示为“指向椭圆的指针”和“指向圆的指针”。最后，使 ELLIPSE 成为 CIRCLE 的超类，因而使 PTR_TO_ELLIPSE 成为 PTR_TO_CIRCLE 的超类。现在考虑下面的代码片断：

```
VAR E  ELLIPSE ;
VAR XC PTR_TO_CIRCLE ;

E   :=  CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;
XC  :=  TREAT_DOWN_AS_PTR_TO_CIRCLE ( PTR_TO ( E ) ) ;

THE_A ( E )  :=  LENGTH ( 6.0 ) ;
```

*解释：*

1）那两个变量声明是不需加以说明的。

2）在对 E 的赋值之后，变量 E 就包含了一个半径为 5 的圆。[⊖] 注意到这个赋值语句的可行性；注意到 E 的指定类型现在是 CIECLE。

3）出现在第二条赋值语句的右边的表达式 PTR_TO 是我们通常所说的**引用操作符**：假如有一个变量 *V*，它返回变量 *V* 的地址，即 *V* 的指针。在这个例子中，它返回类型 PTR_TO_ELLIPSE 的指针值。但是，由于指针值指向的那个变量的指定类型是 CIRCLE，事实上指针值是类型 PTR_TO_CIRCLE，而不仅是类型 PTR_TO_ELLIPSE。所以 TREAT DOWN 操作成功了，那个语句将变量 E 的指针（地址）赋给变量 XC。

4）第三个语句（THE_A（E））是最关键的一个。将会发生什么？有三个可能，具体如下：

a. 出现一个运行类型错误，因为并不支持对 THE_A 中对指定类型的变量为 CIRCLE 的赋值。换句话说，并不支持约束一般化——因此也不支持约束专门化，因此继承模型是坏的（当然它不是“一个现实的可信的模型”——见第 20 章）。在任何情况下，运行类型错误都是令人不快的。

b. 这个语句“成功执行了”，但是没有发生一般化约束。结果是变量 E 现在包含一个“非圆形的圆”（可能指定类型仍然是 CIRCLE，但是那个“圆”有不同长度的半轴）。这个继承模型还是不好的（它不是一个现实的可信的模型），因为约束一般化和约束专门化是不被支持的。再者，不仅变量 E 包含一个“非圆形的圆”，而且变量 XC 也指向一个“非圆形的圆”。此外，不支持类型约束！——因为如果这样，不能产生“非圆形的圆”了。换句话说，完整性约束的最基本的类型不能被支持：当我们定义一个类型，我们不能指定那个类型的合法值。

c. 语句成功执行且发生约束一般化。也就是说，变量 E 现在包含一个“椭圆”且它指定的类型是 ELLIPSE。但是，继承模型仍是坏的，因为变量 XC（它的类型是 PTR_TO_CIRCLE,）现在包含一个指定类型是 PTR_TO_ELLIPSE 的值！这暗示着类型约束不能被支持。*注意*：由于允许一个声明类型 *T* 的变量包含一个指定类型，是 *T* 的超类的值是没有逻辑意义的，更有可能的是原先的语句要么在段 1 下运行失败，要么在段 2 下“成功执行”而不包含有约束一般化。换句话说，这个第三个可能性可能早就无成功希望了。

我们从这个例子中得出结论：如果支持指针，继承模型就是不好的；也就是说，指针和一个好的继承的模型是矛盾的。这是丢弃指针的又一个好的理由。

**2. 第二个根本性错误从何而来**

要找出第二个根本性错误的理由是非常困难的（主要是指技术上的合理性说明——有些辩护只从策略上考虑而根本不涉及技术）。当然，由于对象系统和编程语言都包含指针（以对象 ID 的形式表示），将指针与关系混淆的思想很可能是希望使关系系统更为“对象化”，但这种“辩护”只是将问题推向了另一个层次；我们已经很清楚地表明：对象系统将指针暴露给用户是因为它们不能很好地区分模型与实现的差别。

所以我们只能猜测，指针与关系混淆的思想之所以被广泛采用是因为很少有人理解为什么当初要将指针排除在关系之外。正如 Santayana 所说：那些不能牢记过去的人必将重复过去所犯的错误。在这一点上我们非常同意 Maurice Wilkes 在［26.46］中的说法：

*我很高兴看到计算机科学的教学建立在历史框架的基础上……学生们应该了解目前这一领域的发展从何而来，做过哪些尝试，哪些可行哪些不可行以及硬件的改进对其发展的贡献。如果在教学中缺少这些内容，人们往往会在一些原则性问题上犯错误。这样我们不但不能站在前人的肩膀上继续前进，反而还会热衷于一些已被证明是不可行的方法。*

## 26.4　实现上的问题

适当的数据类型支持所带来的一大好处就是允许第三方厂商（包括 DBMS 厂商本身）构建并出售可方便地插入到 DBMS 中的独立的“数据类型”包。例如对复杂文本处理、金融时间序

---

⊖ 第 20 章中，假设笛卡尔对指针的可能的表示法是称为 POINT 而不是 CARTESIAN。例子中的对 CIRCLE 选择子的第二个语句是一个 POINT 选择子的调用（invocation）。

列处理、地理空间数据分析等的支持。对这些包的描述各种各样，如 Informix 的“数据刀片”（data blades）、Oracle 的“数据盒”（data cartridges）、IBM 的“关系扩展器”（relational extenders），[⊖] 等等。在下文中，我们仍将使用术语“类型包”（type package）。

然而，在系统中加入新的类型包并非简单的工作；要想具备此能力，首先需要 DBMS 本身在设计和结构方面做相应变动。考虑查询中包含引用用户定义类型数据和调用用户定义操作符的情况：

- 首先，查询语言编译器要进行语法分析和类型检查，所以编译器必须知道用户定义的类型和操作符。
- 其次，优化器必须确定适当的查询计划，所以它也要了解用户定义的类型和操作符的某些特性。具体来说，它必须知道数据在物理上是如何存储的（看下一条）。
- 最后，管理物理存储的部分也必须支持新的存储结构——四叉树、R 树等——我们在 26.1 节中讨论矩形问题时提到过，它甚至还应该支持有经验的用户引入新的存储结构，并访问自己定义的方法 [26.29, 26.43]。

总之，系统需具有可扩展性——事实上在每一层上都要有可扩展性。下面我们将对每一层做简要讨论。

**1. 语法分析和类型检查**

在传统系统中，所有可用的类型和操作符都是内置的，有关它们的信息可以“硬捆绑”到查询语言编译器中。相反，在用户可自定义类型与操作符的系统中，这种“硬捆绑”的方法显然是行不通的。所要改进的地方包括：

- 用户定义类型和操作符信息——包括内置类型和操作符信息——保存在系统字典中。这意味着字典本身需要重新设计（至少需要扩展）；同时，引入新的类型包意味着需对字典进行大量更新（在 **Tutorial D** 中，更新是在执行 TYPE 和 OPERATOR 定义声明时完成的）。
- 需要重写编译器来访问字典以获取必要的类型和操作符信息。通过这些信息可以实现第 5 和第 20 章中所描述的类型检查等内容。

**2. 优化**

关于优化设计有许多问题，在本书中我们探讨表面的问题。这些问题包括：

- 表达式转换（“查询重写”）：如第 18 章中所看到的，传统的优化器通过某些转换规则来重写查询。然而，这些转换规则一般都“硬捆绑”在优化器上（因为所有数据类型和操作符都是内置的）。相反，在对象/关系系统中相关信息（至少有关用户自定义的类型与操作符）需要保存在目录中——这就意味着目录需进一步扩充，而优化器本身也需要重写。下面是一些示例：

a）给定表达式 NOT（STATUS > 20），传统优化器将会把它变成 STATUS ≤ 20（采用第二种表达形式能够利用 STATUS 上的索引而第一种却不行）。同样，如果用户定义的操作符是某种形式的相反关系，则也需要通过一定的方式通知优化器。

b）传统的优化器能够知道：表达式 STATUS > 20 与表达式 20 < STATUS 在逻辑上是一致的。如果用户定义的操作符中存在这种情况，也需要通过一定的方式通知优化器。

c）传统的优化器还能知道：操作符“+”与“-”相互抵消（即互为反操作）；例如，表达式 STATUS + 20 - 20 可简化为 STATUS。当用户定义的两个操作符也互为反操作时，需要通知优化器。

- 选择率：给定某一布尔表达式，如 STATUS > 20，典型的优化器会估算一下此表达式的选择率（即使表达式成立的元组的百分比）。对于内置的数据类型和操作符，选择率信息可以“硬捆绑”到优化器中；而对于用户定义的类型和操作符，则需要向优化器提供一些用户定义的代码来进行选择率的估算。
- 代价公式：优化器需要知道执行某个用户定义操作符的代价。例如，给定表达式 *p* AND *q*，

⊖ 我们认为这是不恰当的术语。

其中 $p$ 为某个复杂多边形的 AREA 操作符，而 $q$ 为简单的比较如 STATUS > 20。我们当然希望系统先执行 $q$，只有那些满足 $q$ 的元组才需要执行 $p$ 做进一步判断。事实上，一些经典的表达式转换启发规则，如在连接前先做选择等，对于用户定义的类型和操作符来说都不一定有效［26.10，26.24］。

- 存储结构和访问方法：显然优化器实际上需要知道存储结构和访问方法（见下一小节）。

**3. 存储结构**

显然对象/关系系统比 SQL 系统在物理层上需要更多的存储和访问数据的方式。可以从下面几方面考虑：

- 新的存储结构：如前所述，系统需要支持新的“硬捆绑”的存储结构（如 R 树等），甚至还需要提供方式，让有经验的用户自己来引入新的存储结构和访问方法。
- 用户自定义类型数据上的索引：传统的索引是基于一些固定类型的数据和对于“<”操作符的一个固定的理解上的。在对象/关系系统中，要想在用户定义类型的数据上构建索引，则需要了解用户定义操作符的语义（假设此操作符在一开始已被定义了）。
- 操作符结果集上的索引：直接在类型 POLYGON 的数据集合上构建索引比较简单，最可能的方式是按照多边形的内部字符串来对索引进行排序。然而，基于多边形面积来构建索引将会有用得多。注意：我们在第 22 章中将此索引称作函数索引。

## 26.5　真正融合的好处

在参考文献［26.41］中，Stonebraker 列出了 DBMS 的“分类矩阵”（见图 26-4）。矩阵的第一象限表示那些只处理简单数据并且不支持特定查询的应用（传统的文字处理器是其中的代表）。这类应用从经典的定义来看并非真正的数据库应用；所谓的“DBMS”只是服务于特定需求并构建在操作系统之上的内置文件系统。

第二象限表示那些支持特定查询但只处理简单数据的应用。目前绝大多数应用都属于这一象限，这也是传统的关系 DBMS 最为适合的领域。

第三象限表示那些不支持特定查询但能够处理复杂数据的应用。例如，CAD/CAM 应用即属于这一象限。目前的对象 DBMS 其目标就是这部分的市场（传统的 SQL 产品在第三象限中的应用效果不理想）。

最后，第四象限表示那些既支持特定查询又能够处理复杂数据的应用。Stonebraker 曾给出一个例子：某数据库中包含许多数字幻灯片，拥有典型查询如“给出所有在加州萨克拉门托方圆 20 英里内拍摄的有关日落的照片”。通过对这一示例的分析，他提出了自己的观点：（a）对象/关系 DBMS 适合第四象限的应用；（b）若干年后，大部分应用将会在此象限中。例如，甚至简单的人事资料也会包括雇员的相片、声音记录等。

| | Simple data | Complex data |
|---|---|---|
| Query | 2 | 4 |
| No query | 1 | 3 |

图 26-4　Stonebraker 的 DBMS 分类矩阵

总之，Stonebraker 论证（我们同意这一点）“对象/关系系统是未来的发展方向”；它们并不会昙花一现并被其他思想所取代。然而，应该提醒大家的是，就我们了解，一个真正的对象/关系系统首先是一个真正的关系系统。具体来说，这一系统不会犯那**两个根本性错误**！在这一点上 Stonebraker 可能不怎么同意我们的观点，至少在文献［26.41］中他并没有提到这两个错误。事实上，文中甚至认为将指针与关系结合是可接受也是必要的。

尽管如此，我们仍然认为真正的对象/关系系统能够解决单纯的对象系统所遇到的所有问题。具体来说，此系统可方便地支持下列内容：

- 特定查询、视图定义及声明完整性约束；
- 跨越整个类的方法（这样就不需要特别的“目标”运算对象）；
- 动态定义类（对于特定查询的结果）；
- 双重模式访问（见第 4 章。在第 25 章中我们并未强调这一点，但典型的对象系统不支持

双重模式访问；相反，它们使用不同的语言来访问数据库）；

- 事务约束；
- 语义优化；
- 度（属性数目）在 2 以上的关系；
- 外码规则（ON DELETE CASCADE 等）；
- 可优化性；

等等。此外还支持：

- 屏蔽 OID 和指针冲突，使其对用户不可见；
- “麻烦”对象的问题（例如，连接两个对象的结果是什么?）不再存在；
- 仍保持封装的优点，但针对的是关系中的标量值而非关系本身；
- 关系系统能处理“复杂的”应用，如 CAD/CAM 等。

最后，这一方法在概念上是清晰的。

## 26.6　SQL 工具

SQL：1999 的对象/关系特征是它与 SQL：1992 的最明显和广泛的不同。我们已经在之前的章节描述和分析了许多这样的章节。具体如下：

- 在第 5 章，我们说明了两种不同的种类的用户定义的类型，DISTINCT 类型和结构类型。这两种类型都可以用于作为一个在基表中定义列的基础。
- 在第 6 章，我们说明了 SQL 也允许结构类型特别是用于作为定义成为“类型表”的基础。
- 在第 20 章，我们说明了 SQL 也支持类型继承的一种形式，虽然只是支持结构类型。

但是，除了这些特征，SQL 也支持（a）一个 REF 类型生成子[⊖]；（b）子表和超表。这些额外的结构和那些刚刚提到的结构（特别是和结构化类型）联系得很紧密。他们为什么应该联系得如此紧密并不是很清楚——原则上，这些概念很直接明了——但是可能这一点并不重要，因为我们认为额外的结构无论如何不是很有用，正如我们现在试图说明的一样。

### 1. REF 类型

我们从一个简单（忽略无关的细节）的例子开始：

```
CREATE TYPE DEPT_TYPE
  AS ( DEPT#  CHAR(3),
       DNAME  CHAR(25),
       BUDGET MONEY ) ...
     REF IS SYSTEM GENERATED ;

CREATE TABLE DEPT OF DEPT_TYPE
     ( REF IS DEPT_ID SYSTEM GENERATED,
       PRIMARY KEY ( DEPT# ) ) ... ;
```

*解释*（部分与第 6 章重复）：

1）首先从第 5 章开始回顾。每当我们创建一个结构类型 ST，系统就会自动地生成一个相关的参考（REF）类型称为 REF（ST）；在上例中，参考类型 REF（DEPT_TYPE））是自动生成的。REF 类型用于所有任意类型的数据类型的可以使用的地方；但是，他们只能非显式地生成，这是创建结构类型的一个副作用。

2）类型 REF（ST）的值是在已经被定义为“OF”ST 类型的某些基表[⊜]里的列的参考——换句话说，是指针或地址（见第 4 点）。在例子中，类型 REF（DEPT_TYPE））的值是基表 DEPT 中列的指针。（我们在这里假设 DEPT 是定义为“OF”DEPT_TYPE 类型的唯一的表，虽然这个假设并不总是有效。注意：结构类型 *ST* 当然可以用于其他上下文中——例如，为某些列定义的类型或一些局部变量——但是没有 REF（*ST*）值与那些其他的使用相联系。

---

⊖ REF 在这里代表 reference 的含义，但是 REF 与 REFERENCE 特性无关（见第 17 章）或者基于外码的参考。我们认为 REF 类型是标量类型，所以我们有一个标量类型生成子的例子。

⊜ 或者可能是一些视图，关于视图情况的具体细节超出了本书的讨论范围

3）在一个 CREATE TYPE 语句中声明 REF IS SYSTEM GENERATED 意味着与 REF 有联系的值是由系统提供的。（其他的选择——例如，REF IS USER GENERATED——是可用的，但是这个细节问题超出了本书的范围）注意：事实上，REF IS SYSTEM GENERATED 是系统默认的；因此，在我们的例子中，我们可以从类型 DEPT_TYPE 的定义中完全地省略那个声明的。

4）一个基表可以（通过 CREATE TABLE 语句）被定义为“OF”一些结构类型；这样的表就被称为是一个类型表或是参考表。关键词 OF 在这里并不真的是很合适，但是，因为（正如第 6 章所解释的那样）那个表并不真正是正在讨论的那种“OF”类型，它的列也不是。事实上，那个表对正在讨论的那个结构类型中的每个属性有一个列，附加上一个列——即一个可运用的 REF 列——虽然定义那个附加列的语法不是通常的列定义语法而是定义如下：

```
REF IS <column name> SYSTEM GENERATED
```

这个额外的“自参考”列（起初它在表的列中是从左到右的顺序）通常包含正在讨论中的基表列的唯一的 ID（隐含了 UNIQUE 和 NOT NULL 的声明），当插入一个列时就给这个列指定一个 ID，并将这个 ID 一直保留直到那个列[⊖] 被删除为止。

5）当一个结构类型被用于作为定义一个基表的基础时，它并不是封装的（即使它在其他的上下文中非常重要）。因此，在我们的例子中，基表 DEPT 有四个列 DEPT_ID，DEPT#，DNAME 和 BUDGET（以这个顺序），如果 DEPT_TYPE 被封装，它就不仅仅只是两个了。

6）DEPT_ID 列的默认值是 NULL（事实上，它是所有被定义为 REF 类型列的默认值，虽然如果正在讨论的列被特别地指定为 NOT NULL，那个默认值将不会有太大的意义。）

现在让我们扩展上面的例子来引入一个 EMP 基表：[⊖]

```
CREATE TABLE EMP
     ( EMP#     CHAR(5)  NOT NULL,
       ENAME    CHAR(25) NOT NULL,
       SALARY   MONEY    NOT NULL,
       DEPT_ID REF ( DEPT_TYPE ) SCOPE DEPT
               REFERENCES ARE CHECKED
               ON DELETE CASCADE
                         NOT NULL,
     PRIMARY KEY ( EMP# ) ) ;
```

通常地，基表 EMP 将包含一个外码列 DEPT#，它由部门号来表示部门。但是，在这里，我们由一个“参考”列 DEPT_ID——请注意，它并没有显式地声明为一个外码列——而是由它们的“参考”来指向部分。SCOPE DEPT 指定了可运用的参考表。REFERENCES ARE CHECKED 意味着参照完整性将被保持（REFERENCES ARE NOT CHECKED 将允许“虚悬参照”；为什么它将有可能指定那个选项并不清楚[⊜]）ON DELETE……指定一个删除规则，类似于通常的外码删除规则（支持同样的选择）。没有相似的 ON UPDATE 声明。

**2. 使用参照**

我们现在考虑一些样本查询且在部门 - 职员数据库上进行更新。这里首先是一个查询“找到职员 E1 所在部门的部门号”的 SQL 语句：

```
SELECT DEPT_ID -> DEPT# AS DEPT#
FROM   EMP
WHERE  EMP# = 'E1' ;
```

---

⊖ 在这里看起来像是某个环状：“那个列”的意思可以仅仅是“那个正在讨论中的列的特定的 ID 号”，特别地，注意那个值和变量混淆！——如果“那个列”有一个地址，那么它就是一个列变量（见 26.3 节）

⊖ 在表 EMP 上指定 NOT NULL 的声明。指明在表 DEPT 的列上不允许也有空值不是很容易！这个细节问题被留作一个练习。

⊜ REFERENCES 声明有可能在 SQL：2003 中被删除，暗指（默认）REFERENCES ARE NOT CHECKED 将不会总是被指定。

注意到在 SELECT 语句中的**解除参照**操作符“ ->”（表达式 DEPT_ID -> DEPT#从 DEPT_TO 值指向 DEPT 列中产生了 DEPT#值）[⊖] 也注意到指定一个 AS 语句的需要；如果那个语句已经被省略，相应的结果列将被有效地解除命名。最后，注意 FORM 语句的不合理性——DEPT#值从 DEPT 不是 EMP 中被取回，但是 DEPT_ID 值取自 EMP，而不是 DEPT。

顺便提一下，我们不能不指出第一个查询将可能比它对应的常规的 SQL 查询执行的效率更差。

（该查询只能访问一个表，而不是两个）。我们作这样的观察是因为有利于“参照”的通常的观点是它们应该提高性能（“使用一个指针比执行一个连接操作更快”）。当然，如此争论是为了混淆逻辑的和物理的问题。

作为另一个例子。假设原先的查询是“找到职员 E1 所在的部门（而不只是部门号）”现在这个解除参照操作看起来很不一样：

```
SELECT DEREF ( DEPT_ID ) AS DEPT
FROM   EMP
WHERE  EMP# = 'E1' ;
```

此外，这里产生 DEREF 调用。我们期望的不是一个 DEPT 行值，而是一个“封装的”标量值。那个值是 DEPT_TYPE 类型且只有三个属性：DEPT#，DNAME 和 BUDGET（它不包含一个 DEPT_ID 属性）。[⊖] 注意：为了重复我们在第 6 章所作的观察，如果对某个操作符 *Op* 的某个参数 *P* 的声明类型是 DEPT_TYPE，我们不能从表 DEPT 中传递一行作为调用那个操作符 *Op* 的一个相应的参数。但是我们现在可以看到，我们可以传递 DEREF（DEPT_ID），如果 DEPT_ID 包含表 DEPT 的一个行的一个参照。

这里是另外一个例子：“在部门 D1 中找到职员的职员号。

```
SELECT EMP#
FROM   EMP
WHERE  DEPT_ID -> DEPT# = 'D1' ;
```

注意在这个例子中的 WHERE 子句的解除参照：

```
INSERT INTO EMP ( EMP#, DEPT_ID )
       VALUES ( 'E5', ( SELECT DEPT_ID
                        FROM   DEPT
                        WHERE  DEPT# = 'D2' ) ) ;
```

这里现在是一个插入例子（插入一个职员）：

现在，为我们一直在讨论的内容（REF 类型等）的构造辩护的人强调不需要关注这一点，因为“每一件事务都仅仅是速记的”例如，SQL 表达式：

```
SELECT DEPT_ID -> DEPT# AS DEPT#
FROM   EMP
WHERE  EMP# = 'E1' ;
```

（“查找职员 E1 所在部门的部门号”，我们的第一个例子在这个子查询中）被声明如下：

```
SELECT ( SELECT DEPT#
         FROM   DEPT
         WHERE  DEPT.DEPT_ID = EMP.DEPT_ID ) AS DEPT#
FROM   EMP
WHERE  EMP# = 'E1' ;
```

---

⊖ 大部分支持解除参照的语言也支持一个参照操作符（例如，在 26.3 节中对 PTR_TO 的讨论），但是 SQL 并不支持。此外，解除参照通常返回一个变量，而在 SQL 中它返回一个值。

⊖ 这些属性遵循 DEREF 的语义——虽然对于大多数的目的表 DEPT 的列 DEPT#，DNAME 和 BUDGET 表现为常规列，但系统也需要记得这些列是从结构类型 DEPT_TYPE 中得到的。换句话说，为了使用第 6 章的类型术语和标题，我们可以说表 DEPT 在某个上下文中有一个包含四个部分的标题（类型也同样），但是一个标题（或类型）在其他地方就只是带有两个部分。从这一点来看，可能“类型表”应该称为分裂表。

但是事实上，这个声明并不真正站得住脚，因为新的语义——特别是对于解除参照——可以被只用来关联已经以一种专门的新的方式定义的数据，并使用一种全新的类型生成子（REF）。此外，那些数据定义也使用了许多新的语法。同时，功能性以一种高的非正交的方式被提供（在其他东西中，它只运用到那些以一种专门的新的方式定义的表，而不是所有的表）。

此外，即使我们接受那个声明，无论如何我们得问问为什么提供那些简便（shorthand）。这个声明应该解决什么问题？为什么那些支持是非正交的？什么时候我们应该以那种过时的方式来处理事情，而什么时候用新的特别的方法，等等。注意：关于这一点，参照［26.15］，注释参照［26.21］.

### 3. 子表和超表

SQL 允许基表 *B* 被定义为基表 *A* 的“子表”，当且仅当 *B* 和 *A* 都是“类型表”，且在 *B* 上所定义的结构类型 *STB* 是在 *A* 上所定义的结构类型 *STA* 的子类型。考虑下面的结构类型定义示例：

```
CREATE TYPE EMP_TYPE                          /* employees   */
  AS ( EMP# ..., DEPT# ... ) ...
     REF IS SYSTEM GENERATED ;

CREATE TYPE PGMR_TYPE UNDER EMP_TYPE          /* programmers */
  AS ( LANG ... ) ;
```

注意：PGMR_TYPE 没有 REF IS……子句；而是，它从它的直接的超类 EMP_TYPE 中有效地继承了这样一个子句。换句话说，类型 REF（EMP_TYPE）的一个值可以参照一个表里的一个行，这个表被定义为 PGMR_TYPE 类型而不是 EMP_TYPE 类型。

现在考虑下面的基表定义：

```
CREATE TABLE EMP OF EMP_TYPE
     ( REF IS EMP_ID SYSTEM GENERATED,
       PRIMARY KEY ( EMP# ) ) ... ;

CREATE TABLE PGMR OF PGMR_TYPE UNDER EMP ;
```

注意在基表 PGMR 的定义上的声明 UNDER EMP（也注意那个基表的 REF IS 和 PRIMARY KEY 的声明的缺省）。基表 PGMR 和 EMP 分别被称为是一个**子表**和相应的直接**超表**；PFMR 继承 EMP 的列且加上了它本身的一个额外的列 LANG。在这个例子下的直观看法是没有一个程序员在基表 EMP 中仅有一个列，而程序员在两个表中都有一个列——所以在 PGMR 的每一个行在 EMP 中有一个副本，但是反过来不成立。

在这些表上的数据处理操作如下：

- SELECT：在 EMP 上的 SELECT 被正常地处理。在 PGMR 上的 SELECT 就像 PGMR 真正包含 EMP 和 LANG 中的列一样处理.
- INSERT：在 EMP 上的 INSERT 被正常地处理。在 PGMR 上的 INSERT 有效地引入新的行以出现在 EMP 和 PGMR 中。
- DELETE：从 EMP 中 DELETE 导致行从 EMP 和 PGMR 中消失。从 PGMR 中 DELETE 导致行从 EMP 和 PGMR 中消失。
- UPDATE：更新 LANG，必须通过 PGMR，只更新 PGMR；更新其他的列，要么通过 EMP，要么通过 PGMR，更新两个表（从概念上）。

特别注意下面的隐含含义：

- 假设某个职员 Joe 是一个程序员。如果只是简单地尝试把包含 Joe 的信息的那个行插入到 PGMR 中，系统也将尝试把包含 Joe 的信息的那个行插入到 EMP 中——当然这个尝试会失败。我们应该做的是从 EMP 中删除包含 Joe 的信息的那个行，然后在 PGMR 中插入合适的行。
- 相反地，假设某个职员 Joe 不再是一个程序员。这个时候，我们必须从 EMP 或者 PGMR

中删除包含 Joe 的信息的那个行（不管我们指定哪个表，结果将是从两个表中删除它），然后在 EMP 中插入一个合适的行。

一方面，我们注意到 SQL 提供一个 ONLY 特性，这个 ONLY 特性允许我们有效地仅在一个给定的表中的一些行上的操作，这些行在这个给定的表的任何的子表中不包含副本。例如，SELECT * FROM ONLY（EMP）仅得到在 EMP 在 PGMR 中没有副本的那些行；同样地，DELETE FROM ONLY（EMP）仅删除在 EMP 在 PGMR 中没有副本的那些行，对于 UPDATE 也一样（对于 INSERT 没有 ONLY 特性）。但是 ONLY 特性的缺点是：对于先前描述的问题，即某个职员 Joe 是程序员或不再是程序员的问题，没有用。

那么刚刚描述的子表和超表确切地应该怎样处理类型继承呢？问题的答案是不能做什么。因为表是没有类型的！一般来说，表没有任何可代替性——我们在第 20 章解释说，如果我们没有可代替性，那么我们就没有真的类型继承。因此，如果子表和超表提供了一些优势，它一点也不清楚是为什么，为了得到这些优势，SQL 需要根据类型（子类型和直接超类）分别定义子表和直接的超表。

更根本地，我们需要清楚那些优势在哪里？子表和超表给我们带来什么？至少是在模式层，这个问题看起来“非常小”。[⊖] 如果子表和它的超表在物理上作为一个单一的表存储在磁盘上，可能实现某个执行系统是正确的，但是，不应该允许这样的考虑在这样的模型中起任何作用。换句话说，不仅仅是它不清楚，前一节所注意的也不清楚，为什么“子表和超表”必须依赖“子结构类型和超结构类型”的问题也不清楚，而且，为什么支持那个特性也是一点不清楚。

#### 4. SQL 和两个根本性错误

我们一直在描述的那个 SQL 功能是怎么提供好的对象/关系支持的？如果 SQL 并没有犯那两个大的错误，它确实很接近那**两个根本性错误**。我们必须说这样做的合理性是很不清楚的，至少这个作者认为是这样的；它看起来不仅仅是一个含糊的想法（产生那些错误的特性使 SQL 更“像对象”）。

关于**第一个根本性错误**，将类型表和结构类型结合的想法与将表和类等同的想法有关。更具体地说，如果类型表 *TT* 被定义为是属于（“OF”）结构类型 *ST*，那么 *TT* 就应该包含类型 *ST* 的所谓的“区域”——即所有当前存在的类型 *ST* 的“实例”的集合，这似乎是可能的。[⊖] 否则，为什么 *TT* 和 *ST* 之间有如此紧密的联系？

也就是说，会有一些问题。一个问题是有可能有一个或多个类型表属于“OF”同一个结构类型；这样类型表的类型的含义是不明确的（除了他们几乎一定会违反《正交设计原理》）。其他的问题见 6.6 节。

关于**第二个根本性错误**，应该明确 SQL 确定存在这个（并且是主要的）缺陷，即使我们同意它的“参照”和相关的特征正是它的优势。正如更早注意到的那样，如果行有“参照”（地址），那么那些行是定义的行变量。特别地，SQL 确实存在 26.3 节（在“指针和一个好的继承模型是不相容的”子小节）提到的指针的问题。这里我们省略了其中的细节，因为它们有点零乱；只是说一个本应该参照一个包含圆的行 REF 值事实上可能参照一个包含非圆的椭圆的行。

## 26.7　小结

在本章中我们首先简要讨论了**对象/关系**系统。这一系统实际上就是（也应该是）能很好支持关系域（即类型）概念的关系系统——这就意味着用户能够定义自己的类型。为了获得对象方面的功能，我们并不需要对关系模型做任何改动（除了去实现它们）。

---

⊖ 你可能在想答案与第 14 章提到的实体子类和超类有一些关系。如果这样，那么我们提醒你记住我们自己对这个问题的比较满意的方法，它是基于视图的使用。见 14.5 节末的例子。

⊖ 但是，那个“区域”并不是自动维护的。类型 *ST* 的“实例”出现在表 *TT* 中或从表 *TT* 消失只是由于在那个表上的显式的更新。

我们还讨论了**两个根本性错误**。第一个错误是将对象类与关系变量等同（不幸的是，这一等式在表面上极易迷惑人）。我们猜测这一错误来自对术语“对象”两种不同理解的混淆。我们通过一个示例详细讨论了犯**第一个错误**的系统将会怎样，并且解释了这一错误的部分后果。其中一个重要的后果就是将直接导致犯**第二个根本性错误**！——即将指针与关系相混淆（当然不犯第一个错误的系统也可能犯第二个错误，并且目前市场上几乎每一个产品都犯有这一错误）。我们认为，**第二个根本性错误**在许多方面削弱了关系模型的概念完整性。具体来说，它破坏了基本关系与导出关系之间的可交换性原则。

接下来，我们又简要讨论了实现上存在的问题。基本考虑是加入新的“类型包”将至少影响系统中编译器、优化器和存储管理的设计。结论是，对象/关系系统不能通过在关系系统之上包装一层的方法来实现；系统需要从底层开始重建，以使每一部分具备必要的可扩展性。

最后，我们介绍了 Stonebraker 的 DBMS 分类矩阵，并简要讨论了将对象与关系技术真正融合所能产生的好处（这里“真正的”一词是指系统不犯**任何一个根本性错误**）。最后，我们检查了 SQL 对 **REF 类型**以及**子表和超表**的支持。

## 习题

26.1　定义术语对象/关系，什么是“对象/关系模型”？

26.2　下面是在 26.3 节使用的代码中的一个变量来说明指针和一个好的继承模型是不相容的：

```
VAR E  ELLIPSE ;
VAR XE PTR_TO_ELLIPSE ;
VAR XC PTR_TO_CIRCLE ;

E   :=  CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;
XE  :=  PTR_TO ( E ) ;
XC  :=  TREAT_DOWN_AS_PTR_TO_CIRCLE ( XE ) ;
THE_A ( DEREF ( XE ) )  :=  LENGTH ( 6.0 ) ;
```

当执行这段代码会怎么样？*注意*：这里的 DEREF 是传统的解除参照操作，而不是同名的 SQL 操作（假设一个变量的地址，它返回那个变量）。

26.3　接第 26.2 题：如果用外码代替指针，为什么不会引起类似的问题？或者说是否会引起类似的问题？

26.4　你是否认为 SQL 的结构类型是封装的？证明你的答案是正确的。

26.5　在 SQL 中，声明一个 REF 类型的局部变量是否有意义？如果有，是什么意义呢？

26.6　给出练习 26.2 节的一个 SQL 版本。*注意*：你将可能需要访问 SQL 标准或一个 SQL 产品的文档。

26.7　研究任何一个对你有用的对象/关系 DBMS，那个系统是否有犯那两个大的错误？如果有，是什么原因导致它犯这样的错误？

26.8　解释子表和超表的概念：（a）以概括性方式；（b）以 SQL 方式。

## 参考文献

20 世纪 80～90 年代已经构建了一些对象/关系原型系统。其中有两个最为知名，分别是加州大学伯克利分校的 Postgres 系统［26.36，26.40，26.42，26.43］和 IBM 研究中心的 Starburst 系统［26.19，26.23，26.29，26.30］。我们认为这两种系统（至少其原始形式）都没有采用“明显正确的”等式“域=类”。


［26.1］　David W. Adler:“IBM DB2 Spatial Extender—Spatial Data Within the RDBMS”, Proc. 27th Int. Conf. on VLDB, Rome, Italy (September 2001).

［26.2］　Christian Böhm, Stefan Berchtold, and Daniel A. Keim:“Searching in High-Dimensional Spaces—index Structures for Improving the Performance of Multimedia Databases”, *ACM Comp. Surv.* 33. No. 3 (September 2001).

［26.3］　Frederick P. Brooks, Jr.: *The Mythical Mam-Month* (20th anniversary edition). Reading, Mass.: Addison-Wesley (1995).

［26.4］　Michael J. Carey. Nelson M. Mattos, and Anil K. Nori: “Object/Relational Database Systems: Principles, Products, and Challenges,” Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tuc-

son, Ariz. (May 1997).

文中写到："抽象数据类型、用户定义函数、元组类型、参照、继承、子表、集合、触发器，所有这些到底是什么？"好问题！可以看到，这里列出了 8 个特性（假设默认它们都是 SQL3 中的特性）。在这 8 个特性中，至少有 4 个是我们不希望看到的，有两个几乎是同一事物，而另两个与系统是否是对象/关系系统无关。（详见附录 B）

[26.5] Michael J. Carey *et al.*: "The BUCKY Object/Relational Benchmark," Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

在其摘要中提到："BUCKY（通用或复杂的 Kwery Ynterfaces 标准测试）是一个面向查询的标准测试，测试了许多对象/关系系统的主要特性，包括行类型与继承、参照与路径表达式、原子值集合与参照集合、方法与延迟绑定以及用户定义抽象数据类型与用户定义方法。"注意：关于"关键的特性"，参考文献 [26.4] 的注释。

[26.6] Michael Carey *et al.*: "O - O, What Have They Done to DB2?" Proc. 25th Int. Conf. on VLDB⊖, Edinburgh, Scotland (September 1999).

对 DB2 的对象/关系特征的概观。也可以参照 [26.1] [26.4] 和 [26.11]。

[26.7] R. G. G. Cattell: "What Are Next - Generation DB Systems?" *CACM* 34, No. 10 (October 1991).

[26.8] Donald D. Chamberlin: "Relations and References—Another Point of View," *InfoDB 10*, No. 6 (April 1997).

见 [26.15] 中的注释。

[26.9] Surajit Chaudhuri and Luis Gravano: "Optimizing Queries over Multi-media Repositories," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

对象/关系数据库可以充当"多媒体仓库"。多媒体数据查询的结果一般不仅仅是结果对象集合，还包括对象与查询条件的匹配程度（例如某一图片红的程度）。在查询中既可以设定匹配度的最低要求，也可以设定结果的限额 [7.5]。论文中考虑了对于这类查询的优化，也可以参考 [26.2]。

[26.10] Surajit Chaudhuri and Kyuseok Shim: "Optimization of Queries with User - Defined Predicates," *ACM TODS* 24, No. 2 (June 1999)。

也可以参考 [26.26], [26.35]。

[26.11] Weidong Chen *et al*: "High Level Indexing of User-Defined Types," Proc. 25th Int. Conf. on VLDB, Edinburgh, Scotland (September 1999).

解释了 DB2 使用的一些有效技术。

[26.12] E. F. Codd and C. J. Date: "Interactive Support for Nonprogrammers: The Relational and Network Approaches," in C. J. Date, *Relational Database: Selected Writings*. Reading, Mass.: Addison Wesley (1986).

论文中介绍了关于本质性的思想，这一概念对于正确理解数据模型（包括术语的两种意义，见 1.3 节）是非常重要的。关系模型只有唯一的本质数据结构，即关系本身。相反，对象模型却有许多本质数据结构，如集合、无序单元组、列表、数组，等等（尚未提到对象 ID）。参考文献 [26.13, 26.14] 和 [26.17] 对此有进一步的解释。

| MAJOR_P# | MINOR_P# | QTY |
|---|---|---|
| P1 | P2 | 2 |
| P1 | P4 | 4 |
| P5 | P3 | 1 |
| P3 | P6 | 3 |
| P6 | P1 | 9 |
| P5 | P6 | 8 |
| P2 | P4 | 3 |

图 26-5　材料单关系

[26.13] C. J. Date: "Support for the Conceptual Schema: The Relational and Network Approaches," *in Relational Database Writings* 1985 - 1989. Reading, Mass.: Addison - Wesley (1990).

这篇论文反对将指针与关系相混淆 [26.15] 的一个论据是指针的复杂性。文中通过一个示例清楚地表明了这一观点（见图 26-5 与图 26-6）。

[26.14] C. J. Date: "Essentiality," in *Relational Database Writings* 1991 - 1994. Reading, Mass.: Addison - Wesley (1995).

[26.15] C. J. Date: "Don't Mix Pointers and Relations!" and "Don't Mix Pointers and Relations— *Please*!" both in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings* 1994 -

⊖ VLDB——Very Large Data Bases。——编者注

1997. Reading, Mass.：Addison - Wesley (1998).

这两篇论文的第一篇强烈批评了**第二个根本性错误**。在 [26.8] 中，Chamberlin 驳斥了第一篇论文的观点，第二篇论文是对 Chamberlin 所驳斥内容的直接回应。

[26.16] C. J. Date："Objects and Relations：Forty - Seven Points of Light，"in C. J. Date，Hugh Darwen，and David McGoveran，*Relational Database Writings* 1994 - 1997. Reading，Mass.：Addison - Wesley (1998).

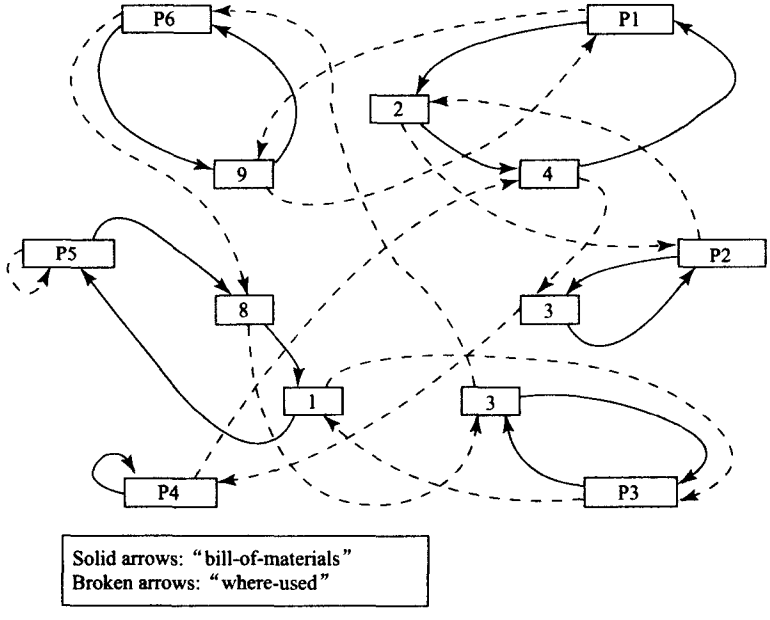



图 26-6　图 26-5 中内容基于指针后的情形

这篇文章对文献 [26.27] 做了极为详细的答复。

[26.17] C. J Date："Relational Really Is Different，"Chapter 10 of reference [6.9]

[26.18] C. J Date："What Do You Mean 'Post - Relational'？"*http://www.dbdebunk.com* (June 2000)

在文献中经常遇上术语“传递关系”（post - relational）。它可能是或者可能不是对象/关系。

[26.19] Linda G. DeMichiel，Donald D. Chamberlin，Bruce G. Lindsay，Rakesh Agrawal，and Manish Arya："Polyflot：Extensions to Relational Databases for Sharable Types and Functions in a MultiLanguage Environment，"IBM Research Report RJ8888 (July 1992).

在其摘要中提到：“Polyglot 系统是一个可扩充关系数据库类型的系统，它支持继承、封装和动态方法调度”（“动态方法调度”为“运行中绑定”的另一个术语）。“Polyglot 系统允许使用不同的应用语言，同时还允许对象在跨越数据库与应用程序的边界时保持其自身的行为。这篇论文描述了 Polyglot 系统的设计，为了支持 Polyglot 中的类型与方法而对 SQL 语言所做的扩充以及 Polyglot 系统在 Starburst 关系原型中的实现。”

Polyglot 系统涉及了本章中（包括第 5、20、25 章）所提到的所有问题。然而，还有两点需要提出。第一，系统中没有提到关系术语“域”（令人奇怪）。第二，Polyglot 系统为类型生成子（在 Polyglot 中称为元类型）提供了基类型、元组类型、重命名类型、数组类型和语言类型，唯独没有关系类型（同样令人奇怪）。当然，系统允许引入新的类型生成子。

[26.20] David J. DeWitt，Navin Kabra，Jun Luo，Jignesh M. Patel，and Jie - Bing Yu："Client - Server Paradise，"Proc. 20th Int. Conf. on Very Large Data Bases，Santiago，Chile (September 1994).

Paradise 系统——“并行数据信息系统”——是威斯康星大学设计的对象/关系（原先称为“扩展关系”）原型，其处理对象为 GIS（地理信息系统）方面的应用。文中描述了 Paradise 系统的设计与实现。

[26.21] Andrew Eisenberg and Jim Melton："SQL：1999，Formerly Known as SQL3，"*ACM SIGMOD Record*

28, No. 1 (March 1999).

概述了 SQL：1992 与 SQL：1999 之间的不同。凑巧的是，当这篇文章第一次发表，Hugh Darwen 和现在的作者写给 *SIGMOD Record* 的编辑如下内容：参照这篇文章——特别地，参照标题为“对象……最后”与“使用 REF 类型”的章节——我们提出一个问题：在这些章节中描述的那些特征有什么作用？更具体地说，除了从 SQL：1992 能够得到的那个特征之外，还能提供哪些有用的功能。这封信并没有发表。

[26.22] Michael Godfrey, Tobias Mayr, Praveen Seshadri, and Thorsten von Eichen: "Secure and Portable Database Extensibility," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

“自从不安全的客户端支持了用户定义运算符后，DBMS 必须明白这些操作符可能破坏系统，直接修改文件和存储器中的内容（避开权限管理机制），独占 CPU、内存或磁盘资源”。控制显然是必要的。这篇论文通过 Java 和对象/关系原型 PREDATOR [26.33] 探讨了这一问题。其结论是乐观的，认为数据库系统：“通过使用 Java，在不大规模牺牲性能的前提下能够支持安全通用的可扩展性”。

[26.23] Laura M. Haas, J. C. Freytag, G. M. Lohman, and Hamid Pirahesh: "Extensible Query Processing in Starburst," Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (June 1989).

在论文 [26.29] 中写道：“Starburst 系统提供以下支持：在表中加入新的存储方法、提供对新类型的访问方法和完整性约束、提供新的数据类型与函数、在表中提供新的操作符”。而在这之后，Starburst 系统的目标又有所扩充。系统被分为两大部分，Core 和 Corona，分别对应于系统 R 中的 RSS 和 RDS（[4.2] 和 [4.3] 中对系统 R 的两个部分做了说明）。Core 支持文献 [26.29] 中所描述的可扩展性函数；而 Corona 支持 Starburst 系统的查询语言 Hydrogen，这一查询语言是 SQL 语言的变种，它（a）取消了系统 R 的 SQL 语言中大部分实现上的限制；（b）比系统 R 的 SQL 语言更具有独立性；（c）支持递归查询；（d）对于用户可扩展。这篇论文还包括了对“查询重写”——即表达式转换规则——问题的讨论（参考第 18 章）。参见参考文献[18.48]。

[26.24] Joseph M. Hellerstein and Jeffrey F. Naughton: "Query Execution Techniques for Caching Expensive Methods," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

[26.25] International Organization for Standardization (ISO): *Information Technology—Database Languages—SQL Multimedia and Application Packages*, Document ISO/IEC 13249:2000.

官方的 SQL/MM 标准的定义。这些定义以 [4.23] SQL 标准为基础，由一个没有固定限度的分离部分的系列（ISO/IEC 13249-1, -2, etc.）本文定义了下面的 6 部分：

第一部分：结构
第二部分：全文
第三部分：空间
第四部分：没有第四部分
第五部分：静态镜像
第六部分：数据挖掘

[26.26] Michael Jaedicke and Bernhard Mitaschang: "User-Defined Table Operators: Enhancing Extensibility for ORDBMS", Proc. 25th Int. Conf. on VLDB, Edinburgh, Scotland (September 1999).

[26.27] Won Kim: "On Marrying Relations and Objects: Relation-Centric and Object-Centric Perspectives." *Data Base Newsletter* 22, No. 6 (November/December 1994).

这篇论文论证了将关系变量与类相等同观点是**第一个根本性错误**。参考文献[26.16]是对此的一个回答。

[26.28] Won Kim: "Bringing Object/Relational Down to Earth," *DBP&D* 10, No. 7 (July 1997).

在这篇文章中，Kim 称在对象/关系产品市场上“整体上很混乱”，因为：首先，“将考虑重点过分放在数据类型扩展上”；第二，“对于对象/关系完整性的评价……处于非常混乱的地步”。他进而提出“一套可用来判断对象/关系产品是否具备完整性的实用化度量。”在他的评价模式中涉及下列标准：

1）数据模型 4）可计算模型 6）数据库工具
2）查询语言 5）性能与可伸缩性 7）能力的发挥
3）关键任务服务

关于第一条标准（最重要的标准!），Kim 的观点为——与《第三次宣言》[3.3] 中的观点完全不同——数据模型必须是“对象管理组（OMG）定义的核心对象模型”，由“关系数据模型和面向对象编程语言中核心的面向对象模型两者的概念构成”。根据 Kim 的观点，数据模型包含下列概念：类（Kim 加入了“或类型”）、实例、属性、完整性约束、对象 ID、封装、（多）类继承、（多）ADT 继承、类型参照的数据、集合值属性、类属性、类方法，等等。注意：关系——我们认为这是最关键也是最基本的概念——并没有被明确提出；Kim 称 OMG 的核心对象模型除上面列出的概念外还包含有整个关系模型，但事实上并不是这样。

[26.29] Bruce Lindsay. John McPherson, and Hamid Pirahesh: “A Data Management Extension Architecture,” Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

描述了 Starburst 原型系统的整个体系结构。Starburst 系统“使关系数据库系统在数据管理方面的扩展实现起来更为方便”。在论文中描述了两类扩展：用户定义的存储结构和访问方法；用户定义的完整性约束（但所有的完整性约束不都是用户定义的吗?）和触发过程。然而，“还有其他一些能够扩展 DBMS 的重要方面，包括用户定义……数据类型和查询评价技术。”

[26.30] Guy M. Lohman *et al.*: “Extensions to Starburst: Objects, Types, Functions, and Rules,” *CACM* 34, No. 10 (October 1991).

[26.31] David Maier: “Comments on the Third - Generation Database System Manifesto,” Tech. Report No. CS/E 91 - 012, Oregon Graduate Center, Beaverton, Ore. (April 1991).

Maier 对文献 [26.44] 中的每项内容几乎都持反对观点。我们赞成他的一些批评意见，但也不赞成另外一些。然而，我们对下述评论（其证实了我们所提出的，对象系统只涉及一个优秀思想，即适当的数据类型支持）很感兴趣：“在面向对象数据库领域，大部分人都试图提取出数据库‘面向对象’的本质……。我个人对 OODB 最重要特性到底是什么这一认识也在随着时间的变化而不断变化。一开始我认为是继承和消息模型，后来我又认为对象标识，对复杂声明的支持及行为上的封装更为重要。如今，在听取了 OODBMS 用户关于他们对系统中哪部分最为满意的意见后，我认为类型的可扩展才是关键。标识、复合态和封装也很重要，但这些只有在系统能够支持新数据类型的创建之后才是重要的”。

[26.32] Jim Melton: *Advanced SQL*: 1999—*Understanding Object - Relational and Other Advanced Features*. San Francisco, Calif: Morgan Kaufmann(2003).

26.6 节讨论的关于 SQL 话题的教程和其他“高级的”的 SQL 的特征，包括 SQL/MED（见第 21 章），SQL/OLAP（见第 22 章）和独立标准 SQL/MM [26.25]。

[26.33] Jignesh Patel *et al.*: “Building a Scalable Geo - Spatial DBMS: Technology, Implementation, and Evaluation,” Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

本文的摘要中提到：“这篇论文在地理空间数据库的并行化方面提出了一些新的技术，同时讨论了这些技术在 Paradise 对象/关系数据库系统中的实现”。[26.20]

[26.34] Raghu Ramakrishnan and Johannes Gehrke: *Database Management Systems* (3d ed). Boston, Mass.: McGraw - Hill (2003).

[26.35] Karthikeyan Ramasamy, Jignesh M. Patel, Jeffrey F. Naughton, and Raghav Kaushik: “Set Containment Joins: The Good, the Bad, and the Ugly,” Proc. 26th Int. Conf. on VLDB, Cairo, Egypt (September 2000).

[26.36] Lawrence A. Rowe and Michael R. Stonebraker: “The Postgres Data Model,” Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

[26.37] Hanan Samet: *The Design and Analysis of Spatial Data Structures*. Reading, Mass.: AddisonWesley (1990).

[26.38] Cynthia Maro Saracco: *Universal Database Management: A Guide to Object/Relational Technology*. San Francisco, Calif.: Morgan Kaufmann (1999).

本书是一本易读的高层次概论书籍。然而，我们注意到 Saracco 支持（与 Stonebraker 在文

献［26.41］中的观点正好相似）一种非常不可靠的继承形式，此形式涉及子表与超表思想的某种版本——我们在［14.13］一开始就对此版本持怀疑态度，但该版本又与 SQL3 中的版本完全不同。具体来说，假设表 PGMR（“程序员”）被定义为表 EMP（“雇员”）的子表。对此 Saracco 和 Stonebraker 认为 EMP 只包含那些不是程序员的雇员信息的元组，而 SQL3 认为 EMP 包含所有雇员信息的元组（参见 26.6 节）。

［26.39］ Praveen Seshadri and Mark Paskin: "PREDATOR: An OR - DBMS with Enhanced Data Types." Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

"PREDATOR 系统的基本思想是为每个数据类型提供机制以确定其相关方法的语义；这些语义在查询优化中将会用到。"

［26.40］ Michael Stonebraker: "The Design of the Postgres Storage System," Proc. 13th Int. Conf. on VLDB, Brighton, UK (September 1987).

［26.41］ Michael Stonebraker and Paul Brown (with Dorothy Moore): *Object/Relational DBMSs: Tracking the Next Great Wave* (2d ed). San Francisco, Calif.: Morgan Kaufmann (1999).

本书是对象/关系系统的指南。它主要——事实上，几乎毫无例外——基于 Informix 的动态服务产品的通用数据选件。通用数据选件基于早期的 Illustra 系统（一个商业化产品，Stonebraker 本人推动了这个产品的发展）。在参考文献［3.3］中有对此书的分析和批评；还可以参见文献［26.38］中的评论。

［26.42］ Michael Stonebraker and Greg Kemnitz: "The Postgres Next Generation Database Management System." *CACM* 34, No. 10 (October 1991).

［26.43］ Michael Stonebraker and Lawrence A. Rowe: "The Design of Postgres," Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (June 1986).

Postgres 系统声明的目标有：

1）为复杂对象提供更好的支持；
2）提供数据类型、操作符和访问方法的用户可扩展性；
3）提供活动数据库工具（警告器和触发器）和接口支持；
4）简化 DBMS 关于系统恢复的代码；
5）能够利用光盘、多处理器工作站和定制的 VLSI 芯片的良好特性；
6）对关系模型做尽可能少的改动（最好不改动）。

［26.44］ Michael Stonebraker *et al.*: "Third - Generation Database System Manifesto," *ACM SIGMOD Record* 19, No. 3 (September 1990).

这篇文章部分是为了回答《面向对象数据库系统宣言》［20.2，25.1］中的内容，认为它在本质上忽略了整个关系模型。文中称："第二代系统在两个方面做出了重大贡献；一是非过程化数据访问，另一是数据独立性，这两个优点绝不应该在第三代系统中受到损害"。下列特性被认为是第三代 DBMS 的本质要求：

1）提供传统数据库服务，并加入更丰富的对象结构和规则
- 丰富的类型系统
- 继承
- 函数和封装
- 可选择的由系统分配的元组 ID
- 不针对特定对象的各项规则（如完整性规则）

2）包含第二代 DBMS
- 只在万不得已的情况下才使用导航
- 集合的内涵和外延定义（即由系统自动维护的集合与由用户手工维护的集合）
- 可更新的视图
- 聚集、索引等对用户不可见

3）支持开放系统
- 多语言支持
- 类型的持续无关性
- SQL（作为通用的数据语言）
- 查询及相应结果必须处于客户/服务器通信的最底层

参考文献［3.3］和［26.31］对这篇文章进行了分析和批评。

［26.45］ Haixun Wang and Carlo Zaniolo："Using SQL to Build New Aggregates and Extenders for Object - Relational System，"Proc. 26th Int. Conf. on VLDB，Cairo，Egypt（September 2000）.

［26.46］ Maurice V. Wilkes："Software and the Programmer，"*CACM* 34，No. 5（May 1991）.

# 第27章　互联网与XML

## 27.1　引言

注：本章最初的作者是IBM的Nick Tindall。

互联网和XML都是现在的热门话题，现在已经有很多介绍这两个方面的书，并且在可预见的将来还会有更多相关的书出现。当然，本书有关的内容都是和数据库特殊相关的。同时，在与这个大前提不太相关的Web或者XML的一些方面的知识上，我们不打算花太多的笔墨详述。对于Web的情况，其与后面章节中密切相关部分我们给出了详细足够的背景（见27.2节），但是在其他方面则较少涉及。对于XML来说，有更多的内容需要讨论，因此我们花了整整三节用于解释相关知识：27.3节给出了XML的一个总体概述，27.4和27.5节包括了XML的数据定义和XML数据的操作。在27.6节我们分析了XML和数据库之间的关系。当然，后面的主题才是我们引入了这章的主要目的——不过在27.6节之前我们暂不做讨论。最后，在27.7节中我们描述了关联SQL程序，在27.8节中总结这一章。

## 27.2　万维网和因特网

在通常的用法当中，Web和Internet这两个词似乎是可以通用和相互交换的，但是从严格意义上来说他们所指的是不同的东西。它们之间的区别可以描述如下：Web是一个庞大的数据库（尽管没有传统数据库原理中关于一致性的设计）；Internet同样是一个庞大的网络，不同的数据库则分布在这个网络之上。注意：你们可能也知道，对网络的访问并不只包括由Internet提供的服务（比如，新闻阅读、远程消息、e-mail、ftp、telnet，等等），这些服务并不是我们主要关注的方面。本书的目的不是讨论新闻阅读、远程消息，以及其他的这些方面的技术细节。

Internet是从Arpanet发展而来的。Arpanet是20世纪60年代后期由美国国防部高级研究项目代理（DARPA）赞助的一个研究项目，这个网络的目的是用一种通用的通信协议TCP/IP（Transmission Control Protocol/Internet Protocol）来连接美国所有的不同的政府网络和科研网络，使得所有这些网络可以连接成为一个一致的“超级网络”。但是这样的Internet（比如，没有Web）还是没有集成到原先预想的地步；用户还是必须是同不同的机制——ftp，gopher，archie，不同种类的e-mail，等等——去获取Internet上的信息。比如，如果你想要查看在一些文档中找到的引用，你可能需要采取下列典型的操作：a）用e-mail或者BBS系统发现相关文件的文件名；b）用telnet登陆一个archie服务器去搜索那个文件的位置；c）通过ftp登陆存有这个文件的服务器；d）浏览那个系统上的相关目录；e）将文件复制到自己的系统上；f）选择合适的程序来显示这个文件。

Web是1989～1990年由Tim Berners-Lee以对所有这些复杂问题［27.2］的阐述为基础而发明的。其中最重要的概念是**超文本**（hypertext），这是由Ted Nelson［27.19］在更早几年的时候提出的。超文本是一种组织信息的方式，它允许文本文档通过内嵌的**链接**（link）引用其他的文档和文件，或者其他文档和文件的部分组件。Berners-Lee的最大贡献在于在一个图形化的浏览器中实现了链接，使得能够将一个窗口中的各种类型的信息都集成起来；在网络上相关的效果就是用户可以通过简单的鼠标点击去获取和显示任何他们需要的信息，而不是不得不通过以前必须使用的所有的单独命令和过程。她可以通过下列定义来达到这种显著的易用性：

- 一种用于标识以及引用文档和其他资源的机制——*Uniform Resource Locators*（URLs，之后被概括成为Uniform Resources Identifiers，或者URIs）。
- 一种标记语言用于生成文档，并且使得这些文档自身包含了如何显示的信息——*Hyertext Markup Language*（HTML）。
- 一种协议用于传递在Internet上传输这种文档——*Hypertext Transfer Protocol*（HTTP）。

注意：在下一节中我们会在标记语言和 HTML 上做一些更多的讨论。

现在，我们可以把 Web 当作一个巨大的数据库，这个数据库分布在一系列的网站上，每个网站都有自己的 web 服务器和标识自己的 URL。用户则通过网络浏览器来访问这个数据库。每个网站中包含了大量的网页，其中每一个页面都有一个相关的根文档用于指定这个页面如何显示。在不同的网站上，根文档也和其他所有的文档一样，含有典型地指向不同类型信息[⊖]（文本、图片、音频、视频，等等）的 URL 链接；对于用户来说，会将根文档和普通页面当作一个整体来对待，最多可能意识到原始页面中的链接，而没有更多的其他东西。但是在显示在用户面前的页面中，其中包含的链接也被同时显示出来，如果用户点击了这样的一个链接，浏览器会在同一个窗口中显示相关的信息（或者在一个新开的窗口中）。注意：对于一些网页来说，用户可以通过填写表单来检索更多的信息。搜索引擎是其中的一种重要的形式。通常来说，一个搜索引擎需要一个指定的搜索参数，比如，“Camelot”，提交之后会返回一个含有相关信息的网站列表。为了能够在一个合理的时间内提供这样的搜索能力，搜索引擎通常需要对 Web 上的成千上万的文档中出现的关键字建立广泛的索引。这些索引是由不间断运行的 web crawlers 来生成和维护的。Web crawlers 能够检索网页并且记录网页中可能有价值的搜索参数。

某一个网站上的信息可以存放在操作系统的文件中，但是越来越多的信息被存放在后台的数据库中（基于 SQL 的数据库或者其他类型的），因此现在的 web 服务器通常需要能够和 DBMS 进行交互。第 27.6 和 27.7 节中给出了一些在这种交互中涉及的思想。

## 27.3　XML 综述

“XML”的意思是 **Extensible**（注意：不是 eXtensible）**Markup Language**。一个 **XML 文档**不准确地说就是一个使用 XML 工具生成的文档。下面是一个简单的例子。注意到其中大量用到的尖括号“<”和“>”（不要和本书中其他地方，特别是 BNF 语法中用到的尖括号混淆起来）。

```
<?xml version="1.0"?>
<greeting kind="succinct">Hello, world.</greeting>
```

例子中的第一行是 **XML 声明**（某几个可选的特性已经省略掉）；XML 文档通常都包含这种声明，虽然这并不是必须的。例子中的第二行是一个 **XML 元素**，其中包含了一个**开始标签**，一些**字符数据**和一个**结束标签**（更一般地来说，一个元素中可能包含字符数据，或者其他的元素，或者两者的混合）。在这个例子中的字符数据是字符串“Hello, world.”；开始标签是这个字符串之前的那个标记，结束标签是字符数据之后的标记。（在这里，标签（tag）也可以作为**标记**（markup）的意思，表示一个开始标签和相应的结束标签一起出现。）XML 文档中标签是由文档定义者所给的名字标识的，同时这个名字也标识了 XML 元素的**类型**；比如，标签为“greeting”的 XML 元素就是一个“greeting”元素。包含在开始标签中的规范

```
kind="succinct"
```

是一个 XML **属性**（attribute，XML 属性和关系意义中的属性没有任何关系）。这个属性的名字是“kind”，字符串“succinct”则是它的值。

就像从这个简单的例子中能够看到的一样，从某些观点上来说，一个 XML 文档仅仅是一个字符串。在这个字符串中包含了数据和标记，而标记则是描述了数据意义的元数据（metadata）[⊖]。典型的来说，XML 文档不仅对于人是可读的和可理解的，而且对于机器来说也是可读和可理解的；这意味着他们可以更容易的在应用程序中进行处理。注意：对于后者来说，我们发现标记使得应用程序可以容忍数据格式在一定范围内的变更。比如：

---

⊖ 根文档中可能也直接嵌入了这种附加的信息。

⊖ 尽管文档中直接包含的数据必须是字符数据，XML 文档同样可以有效的包含很多其他的数据，比如图片，视频记录，以及其他种类的非字符数据。这要归功于文档中嵌入的链接。这种链接也可以认为是标记的一部分。

- 由于 XML 元素中包含了界定（delimiting）标签，因此对于同一个元素的不同实例就不必具有固定的长度，而是可以在文档和文档中间不同，甚至可以在单个文档中也不同。
- （更重要的！）新元素——一些新类型的元素——可以在任何时候加入一个已经存在的文档中，而不会影响该文档的已经存在使用者（特别是已经存在应用程序）。

也就是说，只要数据的生成者和使用者之间能够在如何解释标记这个问题上取得一致，XML 也能够解决一些数据交换的问题。而相反的，对于传统的固定格式的协议来说——比如电子数据交换（EDI）——数据格式的变化需要在数据的生成者和使用者的所有和该数据相关的部分做相应的修改。

**1. 标记语言**

为了发现 XML 的一些隐藏的基本原理，我们有必要来看一下它是如何从更早的标记（Markup）语言发展而来的。不像一些传统的编程语言，标记语言的目的（至少他们的最初目的）仅仅是允许在生成文本文件时允许包含格式提示信息——标记——这些信息可以被可应用的字处理器所识别。在一些特殊情况下，标记由二进制表示，就像在其他情况下被表示为规则文本。不过这样的语言一般都是受专利保护的。比如，IBM 使用一种受专利保护的以文本为基础的语言 Script 来格式化用户手册和类似的文档。下面的例子是从一个典型的 Script 文件中摘录出来的片断：

```
.sp 2
.il 3m;You should specify the ;.us on;first;.us off; parameter
as PRIVATE.  This specification will allow the processor to
complete the conversion without further input.
.br
```

这里的标记告诉格式化工具先向下写两个空行（“.sp 2”），首行缩进 3 em 空间（“.il 3m”），给单词“first”添加下划线（“.us on”和“.us off”），最后换行（“.br”）。

现在 Script 语言以及一些类似的语言面临的一个问题是标记通常是和程序非常相关的——不仅仅是说它控制文档格式的事实，更多的问题是一种标记通常只能适用于一种或者几种特殊的装置（典型的例子就是单色行打印机）。为了弥补这些不足之处，IBM 的三个研究员提出了 Generalized Markup Language（通用标记语言），GML。[⊖] GML 和 Script 语言的关键区别在于 GML 中的标记语言相对于早期的程序化的标记语言来说更具有描述性，或者更有说明性。对于我们刚才提到的 Script 的例子，下面是用 GML 的重新描述：

```
<p>You should specify the <empl>first</empl> parameter as PRIVATE.
This specification allows the processor to complete the conversion
without further input.
```

这里的标记简单地告诉格式化工具，这些文本是一个段落（“<p>”），而不是为这样一个段落给出细节的格式信息（“空两行”，等等）。同时这里的标记也告诉格式化工具，“first”具有第一层次的强调（“<empl>”和“</empl>”）而不是特别的表明它将具有下划线。注意：在例子中我们故意采用了一些 GML；尤其是，我们使用了尖括号来区别标记（tag），而不是更常用的冒号（“:”）。这些违背了通常的习惯，但对当前的使用效果来说并不重要。

GML 的标记确实更具有描述性，但是它还是以关注外观或者文本表示为主（尽管它也能够支持一些其他的任务，比如计算段落的数量）。更特别地，用户只能够使用已经定义在语言中的那些标记。相反，GML 的一种扩展格式——标准 GML（SGML）——允许用户定义自己的标记，并且能够给予这些标记以任何他们所需要的意义。[⊖] 通过使用这个扩展的灵活性，我们可以通过扩展前述例子来详细说明其中数据具有的结构：

---

⊖ 事实上 GML 也是它的三个发明者（Charles Goldfarb，Edward Mosher 和 Raymond Lorie）的名字的首字母缩写，这不仅仅是一个巧合。

⊖ 这里我们引用一段有意思的话：“即使是最小的组织，其中大多数的冲突也来自缺乏对一些常用词的清楚的定义和统一的意义”[27.4]。

```
<paragraph>
  <sentence>
     <subject>You</subject>
     <verb> should specify</verb>
     <object> the <adjective><empl>first</empl></adjective>
              parameter</object>
     ...
  </sentence>
  <sentence>
     ...
  </sentence>
</paragraph>
```

注意：这里我们必须注意到一个事实——“object”元素（element）中同时包含了字符串数据和另一个元素“adjective”。

从这个例子中我们可以看到，SGML 还不是一种真正的标记语言（“SGML”这个名字事实上也是一个名称误用）。更确切地说，它是一种中间语言（metalanguage）——它提供了一些规则，根据这些规则用户可以定义定制他们自己的标记语言；也就是说，用户可以使用他们自己定义的标签。[⊖] HTML 就是这样的一种语言，它也是根据 SGML 定义而来的，我们也可以认为它是一种特殊的 SGML 应用（这里对于“SGML 应用”，我们指的是应用 SGML 中间语言（metalanguage），而不是一个使用了 SGML 的应用程序）。不过不幸的是，HTML 并没有纯粹的保存 GML 的描述性的特性，除了一些结构和语义标记之外，它重新引入了一些格式标记用于格式化文本。事实上一些 HTML 标签能够同时提供所有这三种类型的信息；比如，HTML 标签“<H1>”同时指定了第一层的标题和一个页面的标题（语义上），同时它也可以选择性的指定文本的字体（格式）信息。

**2. XML 的发展历程**

最早 XML［27.25］是由一个 SGML 评审委员会在 1996 发展起来的，这个评审委员会是由 World Wide Web 协会，即 W3C（由 Berners-Lee 在 1994 年创立）赞助的。XML 的最初目的是为了修补 SGML 和 HTML 中存在的某些问题。对于 SGML 来说，其问题在于这个语言对于支持 Web 上的应用来说过于庞杂了。而对于 HTML 来说，存在双重的问题：

- 我们前面已经解释过，HTML 不能够严格的区分结构的、语义的以及格式相关的元数据。
- HTML 允许文档违背文档的“良好形式”规则——也就是说，它允许文档不遵守自己的语法规则！[⊖] 这个问题的产生原因在于市场上同时存在多种互相激烈竞争的浏览器；这样，当面对一个形式不正确的文档 *D* 时，如果这个文档能够被浏览器 *A* 正确显示，而浏览器 *B* 却不可以时，人们不会意识到这是文档 *D* 中存在不足之处，而会觉得是浏览器 *B* 的功能不够完善。

XML 和之前的 SGML 一样，是一种中间语言（metalanguage）（因此“XML”这个名字本身也是一个名称误用（misnomer））；事实上 XML 是 SGML 的一个合理的子集。XML 规范［27.25］中提出：

可扩展标记语言（XML）是 SGML 的子集，……目标是使一般的 SGML 像 HTML 一样，在网络上被使用、接收和处理。XML 使得实现更容易，SGML 与 HTML 互操作更方便。

XML 重新确立了标记的描述特性（比如，XML 定义的标记语言只包括描述性标记）。要注意的是，这样的 XML 在问题上并没有授予标记以特别的含义。

尽管已经表明了其目标，在网页的介质选择中 XML 还没有明显的取代 HTML（通常一个不能够支持 XML 的浏览器仍然能够正确的显示网页）。在另一方面，XML 在其他领域的应用正在突飞猛进的发展：比如，XML 已经被应用在不同的目的上——配置文件、用于分析工具的数据交换格式、协同网络中应用程序之间的新的消息协议等。这样的事实导致的一个结果是，越来越有必要在数据库中存储 XML 数据。下面我们用例子来说明一些我们希望在数据库中保存 XML

---

⊖ 事实上 GML 也是这样的。

⊖ 不幸的是，在我们写这本书的时候，很明显的趋势证明 XML 也正在发生相同的情况。

数据的情况：

- 考虑到我们通常用的关于零件的关系变量 *P*。我们可能想扩展关系变量使得它能够加入一个附加的 DRAWING 属性（见图 27-1）。在给定的任何一个元组中，这个属性的值通过一个特殊的 XML 派生形式 Scalable Vector Graphics（SVG）——详见下一节——对有问题的零件画出的一条横线。可以观察到每一个这样的值都是一个完整的 SVG 文档，而且从前一章的角度来看，整个数据库可以被认为是一个“对象/关系”数据库。注意：事实上，把属性 DRAWING 的值说成横线并不是很正确，更确切的说，这些值是一些 XML 文档，一些应用程序可以将他们解释为生成横线。见图 27-1。
- 相似的，我们可能在关系变量 *P* 中增加一个属性 DESCRIPTION。在每个元组中的属性值也是一个 XML 文档，这个文档描述了问题中的零件，并且解释了使用它的合理方法。

当然，XML 也可以用于表现更传统的数据库中的数据。比如，购买订单，零件目录，以及报表记录等都可以用 XML 来表示。因此，一个非关系的数据库中可能只包含了 XML 文档。在 27.6 节我们讨论了这种可能性。

XML 也可以用来表示关系，这在向关系数据库中导入数据或者从中导出数据时可能会有相关用途（同样见 27.6 节）。比如，下面是我们常用的关于零件的关系的 XML 表示（只包含了 P1 和 P2 的元组）：

```
<?xml version="1.0"?>
<!-- This is an XML representation of the parts relation -->
<!-- of Fig. 3.8 (tuples for P1 and P2 only).  Note that -->
<!-- all data values are represented as simple character -->
<!-- strings.                                            -->
<PartsRelation>
  <PartTuple>
    <PNUM>P1</PNUM>
    <PNAME>Nut</PNAME>
    <COLOR>Red</COLOR>
    <WEIGHT>12.0</WEIGHT>
    <CITY>London</CITY>
  </PartTuple>
  <PartTuple>
    <PNUM>P2</PNUM>
    <PNAME>Bolt</PNAME>
    <COLOR>Green</COLOR>
    <WEIGHT>17.0</WEIGHT>
    <CITY>Paris</CITY>
  </PartTuple>
</PartsRelation>
```

XML 文档片断特点：

- 特殊的布局显示——缩进和特别的行划分——仅仅用于增加可读性。这对于 XML 文档来说不是必须的。事实上，大多数的商业相关的 XML 文档都不包含这种“空白”，通常他们的标记和数据排成一行而没有任何类型的间断。
- 我们把名字 P#换成了 PNUM，这是因为在 XML 标签名字中“#”是一个非法字符（在本章的所有例子中，为了保持一致性，无论是关系还是 XML 中，我们都将做类似的变换）。
- 就是 **XML 注释**中提示的那样，所有（关系）属性值都已经映射成简单的字符串。
- 为了简单起见，我们仅仅选择表示了关系的主体，而没有包括头部信息。（在本章之后的所有例子中，在可行的情况下我们都做了相同的处理。）
- 我们特意说明了**空元素**是允许出现的。比如，假设一些零件没有颜色信息，对于这样的零件我们采用空字符串来表示 COLOR 的值。在文档中显示为 <COLOR> </COLOR>，或者更简单的可以表示为 </COLOR>。

注意：就像术语空元素所提示的，XML 将这样的元素视为不包含内容的元素。然后，从逻辑上来说更正确的说法是他们不包含内容——也就是说字符串（尽管存在一个空字符串）。术语空元素事实上也是一个名称误用。而且，在 27.4 节中的“XML Schema”小节中可以看到“空元素”同样可以有 XML 属性。

| P2 | Bolt | Green | 17.0 | Paris | ... |
|---|---|---|---|---|---|

```
<?xml version="1.0" encoding="utf-8"?>
<svg>
  <g>
    <g>
      <g>
        <path d="M48.888,0.361-0.144-0.2881-48.6,25.05610.288,0.576148.6-25.056L48.888,0.36z"/>
        <path d="M194.184,25.4161-0.144-0.2881-48.6,24.98410.288,0.576148.6-24.984L194.184,25.416z"/>
        <path d="M194.472,25.41610.144-0.2881-48.6-25.0561-0.288,0.576148.6,25.056L194.472,25.416z"/>
        <path d="M48.888,50.410.144-0.2881-48.6-25.0561-0.288,0.576148.6,25.056L48.888,50.4z"/>
        <path d="M145.944,0.36V0h-97.2v0.72h97.2V0.36z"/>
        <path d="M146.016,50.4v-0.36h-97.2v0.72h97.2V50.4z"/>
        <path d="M194.112,64.081-0.144-0.2881-48.6,24.98410.288,0.576148.6-24.984L194.112,64.08z"/>
        <path d="M48.816,89.06410.144-0.288L0.36,63.721-0.288,0.576148.6,25.056L48.816,89.064z"/>
        <path d="M145.944,89.064v-0.36h-97.2v0.72h97.2V89.064z"/>
        <path d="M0.36,64.08h0.36V25.272H0V64.08H0.36z"/>
        <path d="M145.944,88.992h0.36V50.256h-0.72v38.736H145.944z"/>
        <path d="M194.328,64.152h0.36V25.344h-0.72v38.808H194.328z"/>
        <path d="M48.456,89.064h0.36V50.256h-0.72v38.808H48.456z"/>
      </g>
      <path d="M50.976,211.824h0.36V89.352h-0.72v122.472H50.976z"/>
      <path d="M142.416,199.152h0.36V89.208h-0.72v109.944H142.416z"/>
    </g>
    <path d="M143.424,232.1281-0.288-1.1521-0.576-1.0081-0.864-1.081-1.08-0.9361-1.296-0.7921-1.44-0.7921
                                                                                                    ..."/>
    ...
  </g>
</svg>
```

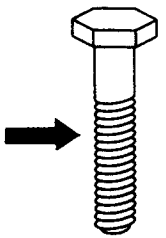


图 27-1 在零件关系变量中加入一个 DRAWING 属性值（示例）

- 最后，我们必须说这里所显示的这个 XML 文档所显示的内容并没有完全忠实于原有关系，这是因为在 XML 文档中强加了一个自顶向下的元组序列和一个从左到右的元组属性的序列（两者其实都是词典序列）。事实上，XML 文档中的元素总是具有顺序的（正式的术语是文档顺序）[○]。在 27.6 节的“文档切割和发布”（Shred and Publish）中，我们还会在这个问题上做进一步的解释。

### 3. XML 文档结构

本章中我们已经多次用到 XML 文档这个术语了，但我们必须知道这个术语事实上不是很准确：文档不是表示成 XML 本身，而是表示成一些 **XML 的派生物**，比如说，一些用 XML 定义的标记语言。[○] 不过，将这样的文档统称为“XML 文档”对于我们来说会方便的多，在以后的章节中我们会沿用这种说法。

假设有一个 XML 派生的 *XD*。这样，*XD* 的定义中包含了 *XD* 中可能出现的标记的意义（至少是非正式的定义），以及可定义的操作符和用 *XD* 处理文档的程序。假设 *D* 为这样的一个文档。抽象的来说，*D* 具有一个层次结构，也就是说包含了一个**根结点**和一系列的**子结点**（这些子结点也具有自己的子结点列表，而且，每个子结点有且仅有一个**父结点**）。对于前几节中提到的关于零件的关系，我们在图 27-2 中给出了他的层次结构。从图中可以看到，除了下述两种情况，每个结点都表示了一个 XML 元素：

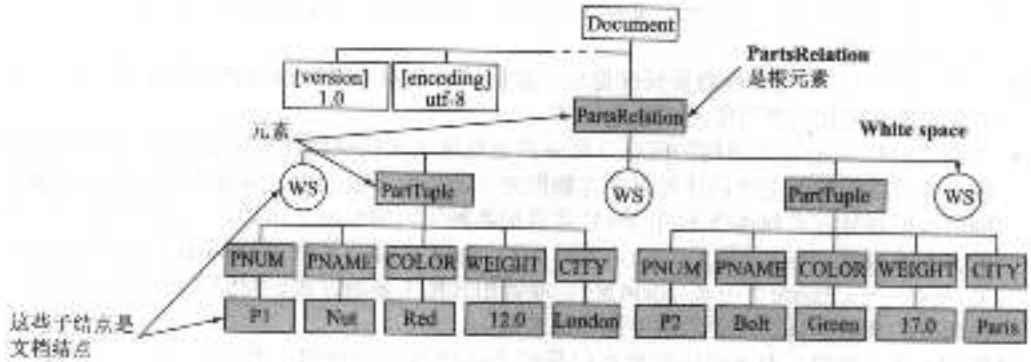



图 27-2　PartsRelation 文档的 infoset（简化）

- 层次结构的根结点，即 **document** 结点代表了整个 XML 文档（注意：document 并不等同于文档的根元素——即最上层的元素结点！）。
- 文档的叶结点代表数据结点。注意：叶结点同时也用于表示：（a）XML 属性（当然在我们现在的例子中并不包含属性）；（b）XML 注释，空白空间，等等不同的项（在图中也有这样的一些结点）。

一个完整的层次结构被称为 XML 文档的**信息集**（“infoset”），这样的层次结构的一种应用程序接口（API）是**文档对象模型** DOM［27.24］。通过 DOM API，一个应用程序可以检索、插入、删除以及改变结点，等等。[⊜]

同时，很明显，会有多个不同的 XML 文档符合一个相同的通用层次结构（比如，他们各自的信息集可能含有相同的一般结构）。举个例子来说，我们可能都认同（至少在这个讨论的目的之上）每本书都含有一个书名、一个前言（可选）、一系列的章节、一系列的附录（可选）和一

---

○ 但是 XML 的属性是没有顺序的。因此，更可取的方法是用属性来表示 PNUM、PNAME、COLOR、WEIGHT 和 CITY，而不是用元素。

○ “XML 派生物”的正式名称是 XML 应用。这里我们倾向于使用前者，因为这样不会与一个使用了 XML 的应用程序混淆起来。

⊜ 指出：一个给定文档的信息集非常接近于该文档的“可能表示”（根据第 5 章）可能会有帮助。

个目录（可选）；其中每章中包含了一个标题和一些非空的节；每节中包含了一个标题和一些非空的段落，其他的情况也可以依此类推。同时，不同的书含有不同的特殊结构，比如一些书没有前言，一些书可能没有附录，一些可能有一些目录，另外不同的书也有着不同数据的章节，等等。因此，如果我们分别用一个 XML 文档来表示每本书，这样在这些文档的细节上我们可以从中发现大量的不同之处——可以想象，其中会有比遵从关系模型的关系元组更多的不同之处。而且，通常我们都认为关系具有非常紧凑的结构，而 XML 文档则具有相对来说松散得多的结构[○]。关系包含结构化数据，而 XML 文档则包含半结构化数据（或者说基于“**半结构化数据模型**”）。

不过，接下来的这些观点并不能经得起仔细的审查。事实上，关系并没有比 XML 文档具有更多或者更少的“结构性”。（它们确实含有不同的结构，但是，能够用 XML 来表示的情况总是可以很好的等价转换成用关系来表示——可能是一个元组，或者一组元组，或者更复杂的情况。）然而，半结构化这个术语在工业界用的比较多，因此这里我们也才提到了这个说法。

附注：这里我们也给出一些在其他文献中提到过的合理的可替换的数据。比如：

- 文献［26.35］中根据实际情况提出：部分数据具有确定的结构，而另外一些则具有不同的结构（比如，每本书都含有一个书名，但不是每本书都有一个目录）。
- 文献［27.13］则根据实际情况提出：数据不含有一个传统意义上的模式，而是一种“无模式”和“自描述”的情况。文献［27.17］中也提出了相同的说法，区别就是再加上了“具有对象特性”的特点。
- 文献［27.23］根据实际数据情况提出：数据“可能是不规则的或者不完整的，并且结构可能会迅速变化或者可能是不可预计的”。
- 当然，XML 具有什么结构的问题，更多的是取决于文档设计者的判断。[○] 从某种意义上来说，这种结构是由文档设计者强加在数据之上的，而不同的设计者显然可以随意的通过不同的数据划分方式和选择不同的标签来强加各种不同的结构。比如，一首诗可以用一整块的文本来表示（< poem > … </poem >），也可以用一系列的诗句来表示（< poem > < verse > … </verse > … </poem >），或者用其他更多的方式。我们或许可以这么说，半结构化的意思是从这样的考虑中演化而来的。

从我们的观点来说，上述的这些理由都是经不起仔细的分析的。事实上，在“半结构化模型”和传统意义上的**层次模型**（就像我们在第 13 章或者参考文献［1.5］中描述并且批判了的模型）的结构方面并没有实质性的不同。当然，这也产生了另外一个问题：通常意义上的数据模型，特别是层次模型，都包含了操作符；但是“半结构化模型”看起来却只和数据结构有关，甚至对操作符有排斥性。

这个问题的最后一个观点是：在一些文献中经常提出 infoset（根据通常的解释见文献［27.26］）和“半结构化模型”都是“XML 数据模型”。但是事实却比这个观点混乱的多，几乎每个文献（比如［27.24］，［27.28］，［27.27］，［27.29］，［27.26］等）都提出并且定义了他们自己的“XML 数据模型”（这些模型中的一部分也有操作符，另一些则没有）。因此，很难搞清楚这些模型中到底哪个应该称为 XML 数据模型。

**4. XML 派生和标准**

XML 的标准化是最近才开始的（1998 年 2 月）[⊜]，但是令人难以置信的是，在之后很短的时间内它得到了大量接受和支持。在写这本书的时候，有至少 200 种不同的派生 XML——有些

---

○ 当然并不是没有结构。从定义上来说，完全没有结构的“数据”指纯粹的噪音数据，这个无结构化数据是相关的概念。

⊜ 和（或）文档类型设计者（见 27.4 节）。

⊜ 因为 W3C 只是一个协会，而不是正式标准组织，所以 W3C 的 XML 规范也只是一个“推荐”标准，而不是正式标准。不过，这个区别对于实际应用并不重要。我们顺便说明一下，XML 规范是根据 Unicode 标准［27.22］制定的一个独立的标准。

是新的，其他的是基于早期的 HTML 定义。这里用一个小的样本来举例说明：

- CML：化学标记语言
- ebXML：在商业贸易（B2B）中已正式替代了 EDI
- MathML：数学标记语言
- PMML：数据挖掘标准
- WML：无线标记语言
- XMLife：保险业标准

也有一些应用和工业规范依赖于 XML，例如：

- SOAP：简单对象访问协议，Web Service 的基础之一，一种新的程序，它经由 Web 发现和访问的联合组件构造应用。
- XMI：XML 元数据的交换，工具之间通信标准。

并且还有许多其他的标准以及潜在的标准，它们要么是有争议的要么是依赖于 XML 标准的（参考图 27-3）。这里有最重要的一些：

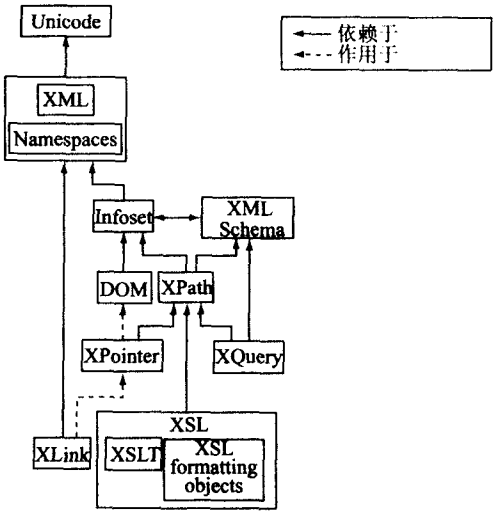



图 27-3　XML 及与 XML 相关的标准和说明

- Namespace in XML：命名模式允许不同的 XML 词汇[○] 无冲突地一起使用。
- XML Information Set：XML 文档结构抽象模型（参看前一节）。
- XML Schema：一种构造描述 XML 文档的模式的语言（参见 27.4 节）。
- DOM：文档对象模型，操作 XML 信息集的对象导向（参见前一节）。
- XPath：XML 路径语言，它提供了一种方法（“路径表达式”）寻址，以此来访问和分离 XML 文档（参见 27.5 节）。
- Xpointer：XML 指示器语言，是基于 Xpath 的但是提供更广泛的寻址方法。
- Xquery：XML 查询，一种 XML 查询语言。
- Xlink：XML 连接语言，一种能允许元素插入到 XML 文档来创建和描述资源间的连接的语言。
- XSL 和 XSLT：XML 类型和类型转换语言，它们一起允许实际上是鼓励格式化信息与描述标记分离开来。

## 27.4　XML 数据定义

如同传统的数据库数据，XML 文档通常有相关的描述符信息。这些信息可以用文档类型定义（DTD）或 XML 模式来说明。DTD 用 DTD 定义语言构造，XML 模式用 XML 模式语言[27.28]构造。我们在本节讨论这两种语言。

**文档类型定义**

DTD 定义语言[○] 在文献[27.25] XML 文档规范中有定义（换句话说，文档定义是 XML 标准一部分）。除了这些，文献[27.25]还提到：

- 定义和使用标置性语言的基本规则是基于 XML（也就是 XML 派生）——怎样区别标记和特征数据，怎样把开始和结束标签恰当配对，怎样详细描述注释等。
- 设计一个处理 XML 文档的程序（通常称为 **XML 剖析器**）的原则是向其他程序显示

---

○ 在 27.4 小节“文档类型定义”中有这一术语的解释。

○ DTD 定义语言表示文档类型定义的定义语言，好像有些冗余，但是其实并非如此。两个定义所指是不同的本节最后的“重述 XML 派生”有更深入的评论。

信息。

并且（如前所述）它也给出了定义 DTDs 的规则。

下面我们用 27.3 节中 PartsRelation 例子的修改版阐明 DTD 的功能。这主要的改变是：我们用 XML 的属性而不是元素来描绘颜色和城市，并且允许在文档的某一地方有一个可选的 NOTE 元素。因此，一个典型的 PartsRelation 文档现在是像这样的：

```
<?xml version="1.0"?>
<!-- This is an XML representation of the parts relation  -->
<!-- of Fig. 3.8 (tuples for P1-P3 only; COLOR and CITY   -->
<!-- now represented by XML attributes instead of by XML  -->
<!-- elements).                                           -->
<!DOCTYPE ... >          <!-- See subsequent explanation  -->
<PartsRelation>
  <NOTE>Revised version</NOTE>
  <PartTuple CITY="London">
    <PNUM>P1</PNUM>
    <PNAME>Nut</PNAME>
    <WEIGHT>12.0</WEIGHT>
    <NOTE>Part color is Red by default</NOTE>
  </PartTuple>
  <PartTuple COLOR="Green" CITY="Paris">
    <PNUM>P2</PNUM>
    <PNAME>Bolt</PNAME>
    <WEIGHT>17.0</WEIGHT>
  </PartTuple>
  <PartTuple CITY="Oslo" COLOR="Blue">
    <PNUM>P3</PNUM>
    <PNAME>Screw</PNAME>
    <WEIGHT>17.0</WEIGHT>
  </PartTuple>
</PartsRelation>
```

上述的一般形式的文档可能有如下形式的 DTD（我们对每一行标上序号以便后面作参考）：

```
1    <!ELEMENT PartsRelation (NOTE?, PartTuple*)>
2    <!ELEMENT NOTE (#PCDATA)>
3    <!ELEMENT PartTuple (PNUM, PNAME, WEIGHT, NOTE?)>
4    <!ATTLIST PartTuple
5       CITY  (London | Oslo | Paris) #REQUIRED
6       COLOR (Red | Green | Blue) "Red">
7    <!ELEMENT PNUM (#PCDATA)>
8    <!ELEMENT PNAME (#PCDATA)>
9    <!ELEMENT WEIGHT (#PCDATA)>
```

解释：

第一行：每一个符合这个 DTD 的文档有根元素，称之为 PartsRelation。这个根元素含有 0 个或多个 PartTuple 元素，在它们的前面可能还有 NOTE 元素。注意：这个可选择性已由问号标示，0 个或多个由星号 * 标示[⊖]。特别是，用“+”代替“*”——就是 PartTuple + 代替 PartTuple* 将意味着“一个或多个”而不是“零个或多个”。再有，“两者都不”就意味着“一个”。

通常，DTDs 可以是内部的（也就是说，在文档中直接描述），也可以是外部的（也就是，包含在其他的某一文件中）：

- 这种内部的情况，（a）DTD 在根元素前面，（b）它必须用一对定界符括起来。开始的定界符是这种形式：

```
<!DOCTYPE document type name [
```

（这里的 document type name——举个例子，这里就是 PartsRelation 必须与根元素的名字一致）。结束的定界符的形式是：

```
]>
```

- 这种外部情况，（a）没有定界符；（b）（因此这种外部情况使得几个文档公用一个 DTD

⊖ * 是 Kleene 操作符。以前在第 21 章遇到过。

更容易）。因此相关的内容应像这样：

```
<!DOCTYPE PartsRelation SYSTEM "file:///c:/parts.dtd">
```

（预示着 DTD 将在文件“parts. dtd”中被找到）。在外部的情况中甚至还有文档中仍然有 <! DOCTYPE…… > 这一行，详细地说明文档类型。

第二行：每一个 NOTE 元素包含有“parsed character data”（#PCDATA），意思是，不严格规则的没有任何标记的文本。第七、八、九行类似。

第三行：每一个 PartTuple 元素含有一个 PNUM 元素，一个 PNAME 元素，一个 WEIGHT 元素（按行中的顺序），有可能后面还有一个 NOTE 元素。

第四至第六行：每一个 PartTuple 开始标签含有 CITY 属性——城市属性的取值必须是 London、Oslo、或者是 Paris。颜色属性的取值必须是 Red、Green、或是 Blue，并且它的默认取值是 Red。

另外还有两点：

第一，所有的元素类型名和属性名集合在一个给定的 DTD 中定义，是指相应地有一个词汇表。这里没有要求这些名字必须唯一。（如果需要的话，在前面章节中提到的名字域机制可以解决名字冲突问题）。

第二，XML 规范定义了两种级别的 XML 文档一致性，“格式良好”和“有效性”。给定一个具体的“原文对象”——即一个特殊的字符串——XML 解析器的职责就是确定这个对象是否满足一致性的要求。下面的两个小节将进行详细描述。

**1. 格式良好**

一个给定的原文对象 *X* 是格式良好的，当且仅当满足以下两个条件：

- *X* 与文献［27.25］中定义的语法一致同时还满足此文献中的 12 条规则（具体的细节超出了本节的讨论范围）。
- 每一个直接或间接与 *X* 相关的文本对象 *Y* 都依次是格式良好的。注意：“直接或间接”在这里我们是指：*X* 直接包含与 *Y* 的关系或是包含与某一其他的原文对象 *Z* 的关系，而 *Z* 又与 *Y* 有着直接或间接的关系。

这些规则表明 *X* 必须有一个精确的根元素，它通常都能够包含其他的元素，每一个元素的开始标签都对应一个相同名字的结束标签（并且这种匹配是区分大小写的），元素的嵌套关系必须恰当，等等。我们所有的 XML 文档的例子都是满足前面的两点的，在此意义上它是格式良好的（事实上，如果一个原文对象不是格式良好的，那么根据定义它不是 XML 文档）。作为对照，下面的一个原文对象是在注释中指出的不是格式良好的：

```
<!-- Warning!  This textual object is not well-formed (and -->
<!-- so is not an XML document at all, by definition).     -->
<PartsRelation>
  <PartTuple>
    <PNUM>P1</pnum>  <!-- End tag does not match start tag -->
    <PNAME>Nut       <!-- Missing end tag                  -->
  </PartTuple>
  </PartTuple>       <!-- Missing start tag                -->
</PartsRelation>
<PartsRelation>      <!-- More than one root element       -->
  ...
```

**2. 有效性**

一个原文对象 *X* 是**有效**的当且仅当它是格式良好，并且符合某一特定的 DTD。这里有一个定义上的格式良好的 XML 文档实例，但是由于声明中的原因仍然不是合法的。

```
<!DOCTYPE PartsRelation SYSTEM "file:///c:/parts.dtd">
<!-- Warning!  This document is well-formed (and is thus   -->
<!-- an XML document), but it is not a valid PartsRelation -->
<!-- document because it does not conform to the DTD for   -->
<!-- such documents as given in the "parts.dtd" file.      -->
<partsRelation>                    <!-- Undefined element -->
  <PartTuple CITY="London">
```

```
  <PNAME>Nut</PNAME>     <!-- PNUM and PNAME elements ... -->
  <PNUM>P1</PNUM>        <!--           ... in wrong order -->
  <WEIGHT>12.0</WEIGHT>
</PartTuple>
  <PartTuple>                    <!-- Missing CITY attribute -->
    <PNUM>P2</PNUM>
    <PNAME>Bolt</PNAME>
    <remarks>Best quality</remarks> <!-- Undefined element -->
  </PartTuple>                   <!-- Missing WEIGHT element -->
</partsRelation>
```

### 3. 类型 ID 和 IDREF 的属性

如大家所见，DTD 支持某些类型完整性约束（属性取值的合法性，等等）○。然而实际上，绝大部分约束是非常弱的（尤其对元素而言——相对于属性来讲——由于它们直接含有实际的数据，所以在本质上根本没有约束可以被指定）。但是由于特殊的属性 type ID 和 IDREF，DTDs 也还是支持**唯一性**（uniqueness）和**相关性**（referential）约束。举个例子，假设我们现在对应整个供应商 - 零件数据库而不是部分的 XML 文档指定 DTD。那么这个 DTD 可能含有如下的定义：

```
<!ATTLIST SupplierTuple SNUM ID #REQUIRED>
<!ATTLIST PartTuple PNUM ID #REQUIRED>
<!ATTLIST ShipmentTuple SNUM IDREF #REQUIRED>
<!ATTLIST ShipmentTuple PNUM IDREF #REQUIRED>
```

（特别要注意到 PartTuple 元素现在有 PNUM 属性而不是 PNUM 元素）。如果根据这个 DTD 文档 *D* 是有效的，那么：

1）*D* 中的每一个 SupplierTuple 元素将会有唯一的 SNUM 取值；并且每一个 PartTuple 将会有唯一的 PNUM 取值。

2）*D* 中的每一个 ShipmentTuple 元素将有一个 SNUM 值，它在 *D* 中其他地方是某一属性的 type ID 值；并且 PNUM 值在 *D* 中其他地方也是某一属性的 type ID 值。

换句话说，在特征上 type ID 有一点像主码，IDREF 有点像外码。不过这种推理关系不是很强。

1）只是简单的字符串的键值我们是没有办法为其赋值的。

2）那些含有多个属性的键值我们是没有办法为其赋值的。（在上面的例子中，注意到我们指定 ShipmentTuple 元素将有一个唯一的 SNUM 值或 PNUM 值）。

3）在例子中，SupplierTuple 元素中的 SNUM 值不仅仅关于所有这样的元素是唯一的，它们关于整个文档中的 type ID 的所有属性也是唯一的。PartTuple 元素中的 PNUM 取值类似。（因此特别有，没有 SNUM 会与 PNUM 取值相等）。

4）而且，ShipmentTuple 元素中的 SNUM 值并不会保证与 SupplierTuple 元素中的 SNUM 值相等——它们仅仅保证与文档中 type ID 的某一属性取值相等。

5）最重要的是，相关性约束检查只是在单个文档的上下文中起作用。没有跨文档的相关性约束检查。

### 4. DTD 的局限性

我们已经知道 DTD 支持的完整性约束是非常弱的。实际上当第一次介绍 DTD 时，关于 DTD 的其他的许多的问题就暴露出来了。例如：

1）它们没有用 XML 语法（也就是说，他们不是 XML 文档），这意味着它们不能被规则的 XML 解析器处理。例如，考虑这个元素声明：

```
<!ELEMENT PNUM (#PCDATA)>
```

这个声明看起来有一点像 XML 开始标记，但是它不是。因为“! ELEMENT”不是合法的 XML 元素类型名，并且“PNUM”和“(#PCDATA)”不是合法的 XML 属性。的确，如果

○ XML 上下文完整性约束问题在文献［27.8］有详细的讨论。

XML 真的如它所声明的那样的通用和强大，那它就能够描述它自己了！[⊖]

2）本质上它们是不支持数据类型的（所有的东西只是字符串）。

3）它们要求不同类型的元素是以某一特定的顺序出现的，甚至那个顺序可以是没有什么实际意义。例如，假设 PartsRelation 文档有一个包含有下列说明的 DTD：

```
<!ELEMENT PartTuple (PNUM, PNAME, COLOR, WEIGHT, CITY)>
```

那么在给定的 PartTuple 元素中出现的 PNUM，PNAME，COLOR，WEIGHT，CITY 元素就必须以特定的顺序出现，即使这个出现顺序在关系条件上毫无意义。*注意*：在原则上我们可以定义一个允许这五个元素以任意顺序出现的 DTD，但是前提是要写出那 120 种不同的所有可能的情况并且保证这 120 种顺序全都是可接受。

这个问题列表并不是毫无疑问的。

尽管那些问题已被列出，但是 DTDs 仍然是很好的很重要的标准，并且它们在实际中被广泛地应用。还有，现实的 DTDs 往往比我们所举的简单例子更复杂更全面。例如，下面我们将会介绍到的 XML 模式，它是由 DTD 定义的有 400 多行（参见 http://www.w3.org/2001/XMLSchema.dtd）。

### 5. XML 模式

XML 模式［27.28］本身就是 XML 的派生。（它不是定义为 XML 规范的一部分，不像 DTD 定义语言。）因此，一个给定的 XML 文档相应的 XML 模式本身就是 XML 文档，称作 *SD*。现在在文档 *D* 和文档 *SD* 之间还没有明确清晰的联系，但是 *D* 可以用一专门的属性 schemaLocation 指出 *SD* 的位置。

特别地，对 XML 文档，XML 模式比 DTD 提供了更广泛的约束。举个例子，这里有的在这节前面“文档类型定义”部分的零件关系 DTD 的相应的 XML 模式（其中 COLOR 和 CITY 是作为 XML 的属性而不是元素）。

```
<?xml version="1.0"?>
<!-- XML Schema schema for PartsRelation documents -->
<!DOCTYPE xsd:schema SYSTEM "http://www.w3.org/2001/XMLSchema.dtd">

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="NOTE" type="xsd:string"/>

  <xsd:element name="PartsRelation">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="NOTE" minOccurs="0"/>
        <xsd:element name="PartTuple" type="PartTupleType"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="PartTupleType">
    <xsd:sequence>
      <xsd:element name="PNUM" type="PartNum"/>
      <xsd:element name="PNAME" type="xsd:string"/>
      <xsd:element name="WEIGHT">
        <xsd:simpleType>
          <xsd:restriction base="xsd:decimal">
            <xsd:totalDigits value="5"/>
            <xsd:fractionDigits value="1" fixed="true"/>
            <xsd:minInclusive value="0.1"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element ref="NOTE" minOccurs="0"/>
```

---

⊖ 当然，这样的陈述是很不精确的。更精确地说，定义 XML 的派生 XD，因此文档根据 XD 有效这就是“文档类型定义”——当然不是 DTD，如我们刚才看到的 DTD 并不是 XML 文档，但是，“文档类型定义”提供像 DTD 一样的功能，并且更理想化。实际上，XML 模式就是一个 XD（见下节）。

```
    </xsd:sequence>
    <xsd:attribute name="CITY" type="City"/>
    <xsd:attribute name="COLOR" type="Color" default="Red"/>
  </xsd:complexType>

  <xsd:simpleType name="PartNum">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="P[0-9]{1,3}"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="Color">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Red"/>
        <xsd:enumeration value="Green"/>
        <xsd:enumeration value="Blue"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="City">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="London"/>
        <xsd:enumeration value="Oslo"/>
        <xsd:enumeration value="Paris"/>
      </xsd:restriction>
    </xsd:simpleType>

  </xsd:schema>
```

现在，前述的模式很明显比它的 DTD 副本更长更复杂，甚至在规定了与 DTD 一样多的约束的绝大多数情况下也是如此。一个很大的不同就是 XML 模式还额外地在元素和属性上强加某些类型约束。XML 模式提供了一套固定的简单类型——布尔型、浮点数、字符型和其他的一些类型——还有某些内置的类型（整型、正整型、负整型等），它们是根据那些简单类型定义的。它也允许用户根据内置的类型定义用户自己的类型。类型可以是简单的也可以是复杂的；区别就是复杂类型的元素能够在里面嵌套地包含其他的元素，而简单类型元素不可以。我们参考 PartsRelation 模式来阐明这些概念，它对 PartsRelation 文档有如下的约束：

1）根元素（PartsRelation）定义为未命名的复杂类型，它的值由可选的 NOTE 元素，然后是 0 个或多个类型为 PartTupleType 的 PartTuple 元素。

2）PartTupleType 类型的元素被定义依顺序由元素 PNUM、元素 PNAME 和元素 WEGHT 组成，后面还跟有属性 CITY 和属性 COLOR，其中属性 COLOR 是可选择的（如果被省略，那么它的默认值为红色）。在这些元素和属性中，PNAME 定义为字符串型；PNUM，CITY 和 COLOR 分别被定义为 PartNum 型、City 型、Color 型（参看第 4 点和第 5 点）。WEIGHT 是未命名的类型，它的类型由系统内部设定。（参看第 3 点）。

3）WEIGHT 型的元素被定义为有约束的小数，它精确到五位，比例因数是 1，最小值为 0.1——也就是说，合法的 WEIGHT 类型取值是 0.1，0.2，…，9999.9。

4）PartNum 类型（有限制的字符串型）定义的通用表达式是 P［0-9］｛1，3｝，这意味着 PartNum 类型的合法取值是一个大写的 P 字母后面跟一位，两位或是三位十进制数字。注意：这种通用的表示式结构借鉴于 Perl 编程语言。

5）Color 和 City 类型（也是有限制的字符串类型）被定义为枚举类型。

注意：之前的讨论是没有标准支持的，更重要的是要明白 XML 模式中的类型“types”并不是第 5 章中真正的意义上的类型的概念。特别地，几乎没有定义相关的操作符作为最初始的类型。事实上，XML 模式类型定义“type definition”与 COBOL 语言和 PL/I 语言中的 PICTURE 规范很类似；也就是说，他们真正所做的就是为正在被讨论的“type”定义一个具有特征的字符串表达式。由于以上的部分原因，在之前的事例中我们可以感觉到从用户自定义类型（第 3 章中列的例子）角度来看类型定义是自由的。

对于作为描述 XML 文档的工具，XML 模式比 DTD 定义语言的几点优势有：

1）XML 模式本身就是 XML 文档。

2）XML 模式支持更复杂的类型机制。

3）XML 模式提供更简单的方法（我们所举的示例中没有这样例子）指定在同一个元素中所含有的不同类型元素，它允许它们按任意顺序出现。

4）XML 模式支持 key 和 keyref 元素（同样，我们所举的示例中没有这样例子），这些元素更类似于关系数据库中的主码和外码，它们能包括一个给定元素的不同层次的值的合并，因此支持其他的被称为“uniqueness within parent”（参看第 13 章中的文献［1.5］）。

当然，XML 模式的一个缺点就是它比一个简单的 DTD 复杂得多。

我们对 XML 模式的讨论得出一些多样性的观点：

1）根据 XML 模式检查 XML 文档叫做模式确认（它与根据 DTD 只是进行简单的有效性，不合格检查是不一样的）。

2）既然模式本身就是 XML 文档，那么它就包含有许多的 XML 元素。然而要特别注意，那些元素中有许多是空的；典型地有，实际上我们用一个空元素去制定一个有属性但没有属性值的元素，就像这里（参看 PartsPelation 例子）：

```
<xsd:element name="NOTE" type="xsd:string"/>
```

3）模式也有它自己的 DTD，由规范定义（在此参看 PartsRelation 例子）：

```
<!DOCTYPE xsd:schema [ ... ]>
```

最后，我们强调指出：对 XML 模式一般性上的多样性（和复杂性）的讨论只是刚刚开始，更详细的信息，参看文献［27.14］或者，XML 模式规范［27.28］。

**6. 重述 XML 派生**

在这一章的前面，我们说 XML 是元语言。也就是说，它是能允许用户定义适合自己语言的语言，特别地还有用户自己定义的标记。我们现在说，一个给定的 DTD 或是 XML 模式，确切地说是一个自定义语言——就是，像这样我们前面说的“XML 派生”。换句话说，一个给定的 DTD 或 XML 模式确切地说就是遵循相应的 XML 文档的语法规则。

从前面所述来看，XML 不仅仅是元语言（metalanguage），而是元语言的元语言（metametalanguage）。XML 在本质上定义了 DTD 构造规则：一个 DTD 就依次是定义构造相应文档规则的元语言。同时注意所有的那些规则主要是语法规则；一般的 XML 和一个给定的特定的 DTD 在意义上都不是由那些规则创建出的文档。

## 27.5　XML 数据操纵

我们转向 XML 数据操纵语言问题。许多这样的语言已经被提出，但是有望成为标准的是 **XQuery**［27.29］。如我们将看到，XQuery——在我们写此书时仍在进行此方面的工作——是以几种早期的语言为基础的，特别地包含有 **XPath**［27.27］；实际上，XQuery 完全包含 XPath。

XQuery 是只读的。如果需要的话，更新必须由 DOM［27.24］或是某一所有者（明确地说是卖主）来完成——但是很显然这两种方法都有问题：

1）DOM 的问题（在 27.3 节中提到的）是它提供给程序员而不是终端用户。

2）用所有权工具的问题是它们是私有的，并且与从一卖主到另一卖主不同。（我们将在 27.7 节中详细地介绍这种功能）。

一种称为 XUpdate 独立于卖主的语言目前正在发展中［27.30］，但是在写本书时它还处于最初级的阶段，所以在此我们不予讨论。因此在后面我们将我们的注意力只局限于 XQuery（和 XPath）。

XQuery 源于早期语言 Quilt［27.9］，它依次被 SQL 和 OQL 影响，并且与老的 XML 语言不同，它还包含 XQL，XML－QL，和 Lorel（参见文献［25.11］中的关于 OQL 的讨论，及文献［27.5］和文献［25.18］中有关于 XQL、XML-QL 和 Lorel 的信息）。现在，完全的 XQuery 语言是相当大而复杂的，并且在书中想彻底地介绍它也是不恰当的。因此我们在这里只是简单地举一系列的例子，这些例子足以理解 XQuery 语言的一般的能力、范围以及它的本质。在我们能做之前我们必须解释一下 XQuery 并不是真正操作在 XML 文档之上的，完全不是！理由如下：

1）根据定义 XML 文档在本质上是字符串，这就意味着它是能被人们读的。

2）结果，它们就有一系列特征（标签、换行符、缩排等）那些对文档可读性有所帮助，但是与文档的真正有信息的文本内容无关。

3）当然，那些含有信息的文本内容才是 XQuery 所要处理的。

因此，XQuery 在本质上不是定义来操作 XML 文档的而是那些已经转换为某一抽象形式（剖析过的）XML 文档。任意给定的 XML 文档的抽象形式被称为一个 XQuery 数据模型的实例“XQuery Data Model”［27.29］；它可以被认为是一个信息集[⊖] ——也就是，层次——像图 27-2 所显示的那样（见 27.3 节的 XML Document Structer 子节）。因此特别注意，XQuery 表达的查询的结果是一个信息集而不是一个 XML 文档。就如在文献［27.29］中所写到的“transformation of a Data Model instance［back］into an XML document is currently an open issue”。实际上，计算一个 XQuery 表达式的结果可能甚至不是一个严格意义上的信息集，因为（我们后面会看到）它有可能不是格式良好的。

### 1. XPath

XQuery 很大程度地依赖于 XPath 的表达式，所以我们在开始时先对这种表达式做一下简单介绍。这样的表达式在概念上与第 25、26 章中描述的表达式具有很强的家族类似性。更确切地说，XPath 中的路径表达式就是开始于某一给定的与源节点或是某一给定的信息集中的节点，沿着一条具体的路径或是信息集中的某些路径寻找我们想要的目标节点或节点集。注意：这里源（source）和目标（target）术语并不是 XPath 中的正式的术语。

在语句结构上，XPath 路径表达式是由一系列步骤组成的，每一步的开始都用斜线“/”与前一步隔开，并且最开始可以是一个或两个斜线字符：

```
[ / | // ] step / step ... / step
```

开始的单个斜线意味着导航是从根节点开始（根节点即源节点）；若开始是双斜线意味着由每个节点依次开始（实际上，它会导致整个信息集被搜索，即深度优先——从左到右顺序遍历，每一个节点会依次作为源节点）。如果开始时根本没有任何的斜线，当前节点——也就是，当前访问到的节点——作为源节点。

这里有一些简单的例子，它们是基于 27.4 小节中“文档类型定义”部分的 PartsRelation 文档（你将会回想起来它含有零件号为 P1、P2、P3 的 PartTuple 元素）：

1）表达式

```
/PartsRelation/PartTuple
```

返回对应于那三个零件元组的节点。

2）表达式

```
/PartTuple
```

返回一系列空节点，因为根节点没有 PartTuple 的孩子。

注意：这里的根节点不是 PartsRelation 节点而是所有的文档节点（参见 27.3 节中的对图 27-2的讨论）。同样的论述适用于前面的例子，当然也适用于下一个例子。

3）表达式

```
//PartTuple
```

返回与第一个例子一样的结果。

通常，给定的路径表达式每一步计算都是根据上一步的结果来计算的（更确切地说，如果前一步的结果是一系列 *SN* 节点，那么对于当前步骤 *SN* 中的每一节点变成文本节点）。每个步骤含有三个部分：

---

⊖ 更准确地说，抽象形式由增大的信息集（被称为 Post Schema Validation）组成，被转化为 XQuery 数据模型形式。

1）一个轴确定导航方向：向上（父亲节点，后代节点），向下（孩子节点，祖先节点），向左（前一节点，前一节点的兄弟节点），或向右（后一节点，后一节点的兄弟节点）。

2）一个节点的测试，是指定感兴趣节点的类型。

3）可以选择性地用一个或多个量词来消除不要的节点。

注意：具体的轴可以是“parent”、“child”等，在第一点中并没有列举出所有的情况——它仅仅只是举例说明。如果没有确定具体的轴，那默认值就是“child”（也就是，导航就从文本节点的孩子节点开始了）。

为了更详细一点地介绍路径表达式是如何计算的，我们来考虑一下下面这个更复杂一点的例子：

```
/PartsRelation/PartTuple[WEIGHT="17.0"]
```

解释：

1）最开始的“/”强调了根节点（也就是文档节点）作为紧跟下来一步的上下文节点。

2）路径表达式可以是（并且通常是）采用缩写形式。因此，例子中的 PartsRelation 表达式就是 child:: PartsRelation 的缩写。这一步的结果就是 PartsRelation 节点是根节点的唯一的孩子节点。

3）类似地，“PartTuple”表达式就是“child:: PartTuple”的缩写；它产生了 PartsRelation 的三个 PartTuple 孩子节点。

4）最后，谓词

```
[WEIGHT="17.0"]
```

消除了所有的 PartTuple 节点除了那些 WEIGHT 值为 17.0 的节点。注意：“WEIGHT”形式本身就是缩写形式；非缩写形式是：

```
[child::WEIGHT="17.0"]
```

所以最后的结果是对应于 P2 和 P3 两个 PartTuple 节点序列。

现在我们对 XPath 这一在之前的讨论中起着至关重要的作用的概念做一下简要的小结。我们已经看到在给定的路径表达式中的每一步的计算式是依赖于某个作为“当前节点”的上下文相关节点的。目前，在 SQL 系统出现以前，一个非常类似的概念出现在那些早期的“手工导航”系统中，遍及了存取语言，统治了 DBMS 的市场，而这个概念导致了系统复杂性的产生——更不要说编码错误了。事实上，再引入这个概念是否明智值得怀疑。

### 2. XQuery

XPath 有一个问题就是它仅仅只是一个寻址的机制。它的路径表达式可以涉及层次中存在的任何节点，但是他却不能构造出原来不存在的节点。换句话说，XPath 语言有点类似于“关系的”语言——这个词用引号是因为，关系的语言当然不是导航式的——它支持选择、投影、但不支持连接[⊖]。这种现象是导致 XQuery 出现的部分原因；XQuery 优于 XPath 的一个主要的地方就是能够造出一个新的节点。

我们在第一个例子里就会阐述这种处理能力。和前面一样，我们再次假设有一个 XML 文档 PartsRelation，再假设同时又有相似结构的供应商关系和供货关系的文档。如下给出查询“对于每一个供货关系，给出供应商名，零件名和供货数量”的 XQuery 形式：

```
<Result>
  { for $spx in document("ShipmentsRelation.xml")
              //ShipmentTuple,
        $sx  in document("SuppliersRelation.xml")
              //SupplierTuple[SNUM = $spx/SNUM],
        $px  in document("PartsRelation.xml")
```

⊖ 这是一个过于简化的语句；XPath 有效地支持一种笛卡尔积操作符（当然这是连接的一种退化情况）。但它不支持连接的更普遍的形式。

```
                //PartTuple[PNUM = $spx/PNUM]
    order by SNAME, PNAME
    return
      <ResultTuple>
        { $sx/SNAME, $px/PNAME, $spx/QTY }
      </ResultTuple> }
</Result>
```

解释：

1）上述表达式构造了一个单独的结果对象，包含一系列结果元组对象。另外，我们注意到如果我们删除了封闭的 Result 标签，剩下的表达式仍然是一个合法的 XQuery 查询，但是它返回的结果可能不很规范。

2）一种解释上述表达式（忽略封闭的 Result 标签）语义的好的方法，就是按照类似于关系积分的表示方法：

```
{ SX.SNAME, PX.PNAME, SPX.QTY } WHERE SX.SNUM = SPX.SNUM
                                  AND PX.PNUM = SPX.PNUM
```

这个表达式有效的实现了供应商，零件和供货三者之间的有效连接，然后将连接后的结果在 SNAME，PNAME，和 QTY 上作投影。注意：我们没有给出 XQuery 中类似于“order by”功能的操作，因为“order by”不是一个关系的操作。同时，我们采用了传统的习惯使用范围变量名（SX、PX、SPX 都分别是建立在供应商、零件、供货关系上的范围变量）。最后，我们略去了这样一个事实，关系的表达式会自动地消除重复元组，而 XQuery 表达式却不能。

3）在前面的章节中说到过，XQuery 中的变量 $ sx，$ px，$ spx 充当了类似于关系上的微积分中范围变量的角色，然而，微积分容易让人产生误解；XQuery 的表示形式和关系的表示形式不很相似，因为就像你能看到的那样，XQuery 显得更为程序化（这里你可以参考［27.3］中的解释）。事实上，XQuery 的表式形式看起来更类似于如下嵌套 - 循环的表式形式（这里用了伪代码的形式，用“.”来代替“/”操作符）：

```
do for each shipment $spx ;
   do for each supplier $sx where $sx.snum = $spx.snum ;
      do for each part $px where $px.pnum = $spx.pnum ;
         emit { $sx.SNAME, $px.PNAME, $spx.QTY } ;
      end do ;
   end do ;
end do ;
```

看起来 $ sx，$ px，和 $ spx 比起它们作为范围变量的情况，更像是循环控制变量（在传统的编程理解中）。还有就是在外层循环中我们要对 shipment 进行重复；也就是我们要先引入循环控制变量 $ spx，因为另两个变量是通过它来定义的——事实上优化器这样做一定是有含义的。相反，另两个变量 $ sx 和 $ px 可以是任意顺序。尽管优化器这样安排顺序有什么意思本身就是一个有意思的问题。至少我们知道变量被引入的顺序对结果对象的产生顺序是有一定影响的（见第 7 点中对 return 子句的讨论）；所以一般说来，XQuery 中表达式 A JOIN B 和 B JOIN A 并不相等。

按照如上的推论看来，可以知道，XQuery 相对于一种程序语言来说，更像是一种终端用户查询语言。

4）现在我们来关注一下 for 子句，尤其是子句中位于第一个逗号之前的部分。表达式

```
document("ShipmentsRelation.xml")
```

返回的是解析过的 XML 文档包含在名为 ShipmentsRelation. xml 的文件中，所有的文档节点都作为上下文节点。注释“//ShipmentTuple”的意思是我们感兴趣于文档中的 ShipmentTuple 对象。而注释“for $ spx in”的意思是那些供货元组在文档中出现的顺序，它们轮流作为变量 $ spx 的值。

5）for 子句的下一个部分——

```
$sx in document("SuppliersRelation.xml")
       //SupplierTuple[SNUM = $spx/SNUM]
```

也是类似的，除了变量 $ sx 只覆盖满足条件 SNUM 的值等于 $ spx 的当前值的那些供应商。

6）for 子句的最后一部分也是类似的情况。

7）变量 $ sx，$ px，$ spx 的值每合并一次，就执行一次 return 子句，顺序就是 for 子句产生出值的顺序。所以，在例子中，return 子句返回产生 ResultTuple 对象的顺序受如下规则指导：$ sx 的值改变得最快，$ px 的值次之，$ spx 的值改变最慢。

8）order by 子句或多或少是自解释的。然而要注意的是，它会出现在相应的 return 子句之前。这样的位置关系允许结果在值的基础上排列顺序，尽管值并不真正出现在结果中（就像如下的这个 SQL 查询：SELECT CITY FROM P ORDER BY WEIGHT）。然而从概念上来说，始终是 return 子句先被执行，因为排序只有在有结果的基础上才能进行。

这里是第二个例子（查询有两个以上供应商供应的零件号和供货总数）

```
for $pnum in
    distinct-values(document("ShipmentsRelation.xml")//PNUM)
let $spx := document("ShipmentsRelation.xml")
            //ShipmentTuple[PNUM = $pnum]
where count ( $spx ) > 1
order by PNUM
return
  <Result>
    { $pnum,
      <totqty> { sum ( $spx/qty ) } </totqty> }
  </Result>
```

1）这个例子是一个完整的 FLWOR 表达式（FLWOR = for + let + where + order by + return；读作 flower）。

2）注意这里使用了 distinct-values 来去掉 for 子句里重复的 part 号；$ pnum 只包括了不同的 shipment part 号。

3）let 子句不同于 for 子句，指定的变量不重复每个变量的具体值而是赋予一个整体的值序列。此外，在这个例子中，let 子句将轮流评估每个不同的 shipment part 号，因为它是嵌入在 for 子句里的。

4）Where 子句中仅当当前 shipment 的序列长度大于 1 时，才会考虑表达式的其余部分。XQUERY 支持一般的聚集操作例如 count、sum、avg 和 min。

5）同样，我们必须得说全部的表达式看上去相当程序化。伪代码的变体如下：

```
do for each distinct shipment part number $pnum ;
   do for all shipments $spx where $spx.pnum = $pnum ;
      if ( count of such shipments $spx ) > 1 then
         emit { $pnum, sum ( $spx.qty ) } ;
      end if ;
   end do ;
end do ;
```

对应的关系语句：

```
{ SPX.PNUM, SUM ( SPY WHERE SPY.PNUM = SPX.PNUM, QTY ) }
            WHERE COUNT ( SPY WHERE SPY.PNUM = SPX.PNUM ) > 1
```

到目前为止，我们的例子可能还是有些不切实际，严格的说他们有些关联，通常它们很少利用 XML 文档的层次特性。所以，下面进行重新设计。首先，是 PartsRelation 文件和以前完全一样。代替 SuppliersRelation 和 Shipment 文档中，然而，假设我们有一个 SuppliersOverShipments 文档，在其中：

1）根元素包括一系列的供应商元素。

2）在一系列的货物元素之后，每一个供应商元素包含 SNUM、SNAME、STATUS 和 CITY 元素。

3）每一个货物元素包含一个 PNUM 元素和 QTY 元素。

如图 27-4 所示：

现在来研究两个查询“找到支持零件 P2 的供应商”和“找到由供应商 S2 提供的零件”。下面是相关的公式表达：

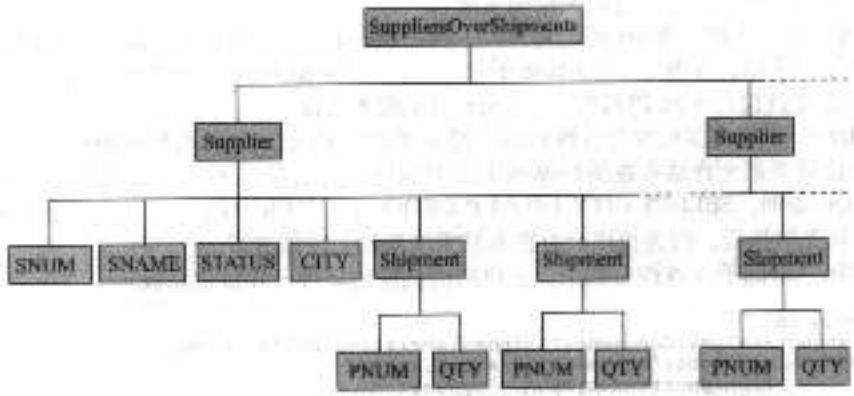



图 27-4　供应商和货运的层次结构

```
SX WHERE EXISTS SPX ( SPX.SNUM = SX.SNUM AND SPX.PNUM = 'P2' )

PX WHERE EXISTS SPX ( SPX.PNUM = PX.PNUM AND SPX.SNUM = 'S2' )
```

注意：简而言之，我们假定 SNUM 和 PNUM 的值是简单的字符串，不是某种用户特定的类型。

下面是对第一个查询的 XQuery 公式表达：

```
for $sx in document("SuppliersOverShipments.xml")//Supplier
where $sx//PNUM = "P2"
return
  <Result>
    { $sx//SNUM, $sx//SNAME, $sx//STATUS, $sx//CITY }
  </Result>
```

下面是对第二个查询的 XQuery 公式表达：

```
let $sx := document("SuppliersOverShipments.xml")
           //Supplier[SNUM = "S2"]
return
  <Result>
    { document("PartsRelation.xml")
      //PartTuple[PNUM = $sx//PNUM] }
  </Result>
```

正如我们所看到的，XQuery 形式与其他形式相比更少的具有相称性。因为包含 XML 设计的不相称（分等级）的类型。

我们总结一下这部分的相关特征。

- XQuery 像 SQL 一样作为原始的公式表达［4.9～4.11］，没有外在的连接支持。实际上，参考［27.29］详细的证明［FLWOR expressions are］对计算连接是很有用处的。更有力的证明了，用户必须详细说明任何涉及这样计算的步骤顺序。然而，它通过复制限制，包含了外在合并、交叉和不同（其他的）的支持。
- XQuery 也包括对有关存在的普遍的量词的外在支持。下面是两个例子：

```
some $x in ( 2, 4, 8 ) satisfies $x < 7

every $x in ( 2, 4, 8 ) satisfies $x < 7
```

第一个表达式是求正确的值，第二个是求错误的值。

- 正如第 7、8 节所解释的，关系模型包含一个与 completeness 有联系（且重要）的概念。

一般来说，似乎没有同 XQuery 和 XML 类似的概念[⊖]。

## 27.6　XML 和数据库

现在我们开始考虑本章所讨论的数据库问题之间的关系。毫无疑问，我们需要把 XML 文档——也许我们应该说是 XML 数据，存储到数据库中；还要按要求来检索和更新数据。实际上，还有相反的要求，也就是可以处理“规范的”或非 XML 的数据，尤其是一些查询的结果，需要把它们转换成 XML 格式，以便它能以 XML 文档形式传输给另一个用户。这里我们主要讨论前一个要求。

很显然，我们可以按三种基本方式把 XML 文档存储到数据库中：

1）我们可以用一个元组的某个属性值来存储整个文档。

2）我们可以把文档划分成块，用不同关系的不同元组的不同属性值来存储这些小块。

3）我们不用传统的数据库来存储文档，而是用 native XML 数据库（例如那些存储 XML 文档而不是关系表的数据库）。

我们将依次来考虑这三种存储方式。

**1. 以属性值来存储文档**

我们在 27.3 节的“XML 的发展”部分接触过这种存储方式，我们提到可以扩展关系变量 P 来包含 DRAWING 和 DESCRIPTION 属性。基本观点如下：

- 首先，我们定义一个新的数据类型，称为 XMLDOC，它的值是 XML 文档，然后我们就可以把给定的关系变量的某个属性定义为这种特殊数据类型。注意：我们已经看到，XML 有些冗长：一个 XML 文档的大小可能是它表示的原始数据的 5 到 10 倍，如果直接处理原数据会非常低效。在内部以压缩的形式（或分列的方式）存储这些文档将会有很大的改善。当然，这种考虑与模型无关。
- 包含 XMLDOC 的元组可以利用传统的关系操作 INSERT 和 DELETE 进行插入和删除，元组中的 XMLDOC 值也可以利用方便的关系操作 UPDATE 整体替换。当然，XMLDOC 值也可以按照传统的方式进行只读操作。
- 像所有数据类型一样，XMLDOC 类型也将有一组相关的操作符。这些正在讨论之中的操作将对 XMLDOC 属性进行粒度更细的检索和更新操作，例如：支持访问单个的元素或属性结点。查询方面的操作符可能将类似于 XQuery 中的相应操作；这些操作甚至提供接口供 XQuery 直接调用。当然 native 方式也许更有友好性。然而，更新操作也应该支持。
- 还需要提供一些操作符，来判断一个给定的 XMLDOC 值是否符合指定的 DTD 或 XML 模式，也就是是否有效。（当然，XMLDOC 值定义必须是结构良好的，但它未必就是无效的。）

注意：这第一种方法，把整个文档存储为属性值，有时在一些文献和应用中被不正式地称为 XML 列。这种方法适合的情况是：

- 文档已经存在
- 文档通常被整体而不是部分地处理
- 文档很少更新
- 数据检索通常是基于一个小的、已知的元素集或属性集
- 文档为了审计要求完整存放

总之，这种方法适合于文档为中心的应用［27.7］，也就是说这些文档主要是供人们来阅读，并且它们主要用自然语言书写。

**2. 划分与发布**

第二种方法没有定义任何新的数据类型。XML 文档被划分成块——例如：一个个的 XML

---

⊖ Except perhaps for flower power?（抱歉，但是诱惑太大难以抵挡）。

元素和属性——然后用不同的关系的不同属性值来存储这些小块[○]。注意：这种情况下数据库中不包含整个的 XML 文档、DBMS 对这些文档一无所知，因此，DBMS 不能创建出 XML 文档，而是由应用程序（可能是一个 Web 服务器）来将那些属性值按照某种方式组合而成的。

由于应用程序可以从数据库中规范的数据生成 XML 文档，我们就可以达到了我们原先的第二个目的，也就是说，我们拥有了从规范数据（非 XML 数据）中得到查询结果并转换成 XML 形式的方法。这一转换被称为**发布**（正在讨论中的数据）。因此在这样背景下，发布这个词，在某种意义上就是与**划分**相反。我们以前曾经谈及，划分与发布之间的转换规则本身通常是包含在 XML 文档格式中。

顺便提一下，将非 XML 数据发布成 XML 形式的数据的能力也被认为是在非 XML 数据上建立 XML 视图的能力。（更精确地说，这种发布用于支持 **XML 视图**以检索数据。如果要更新这种视图的话，那就要支持相应的划分函数。）毕竟，用第 2 章的术语讲，就是没有原因要求系统的外部和概念层必须建立于同一个数据模型之上；实际上，我们曾在第 3 章提到过，当一个系统的外部视图是层次模型时，其概念层也是可以采用关系模型的。唯一的硬性要求，就是在这两者之间是可逆转换的。

XML 模型可以认为是从另一个视角看待的传统的层次模型；这种模型与关系模型之间的转换，由于所谓的阻抗失配（impedance mismatch）的原因——参见第 25 章——还存在一些问题。一个问题是，在层次模型中，一个结点的孩子结点不是组成一个集合，而是一个序列（也就是说它们是有序的），但是一个关系中的元组却是无序的。因此，当把一个 XML 文档 *D* 进行划分并存储到关系数据库时，文档 *D* 的某些方面（信息的或者其他的）可能会丢失。如果真是这样，那将不能保证文档 *D* 以 XML 形式发布后的形态能与其原本的形态完全一样。尤其是这两种形态的文档中的空格可能会有所不同。

注意："划分与发布"这种方法有时候也被非正式地称为 **XML 聚集**。这种方法可能适用于如下的情况：

- 数据已经存放于关系数据库中，且必须与 XML 文档中相应的数据相作用。
- 只有文档中的文本部分需要原样地保存（标签可以抽取成关系属性名）。
- 操作通常是施加于单个的元素和属性结点。
- 更新频繁，且很注重更新的性能。
- 处理程序使用现有的关系界面。

总之，"划分与发布"方法适用于所谓的**以数据为中心**的应用［27.7］。也就是，该应用中的文档通常是用可操作性的或支持决策的信息描述，而不是自然语言描述。

### 3. XML 数据库

我们在此提到这第 3 种方法主要是为了介绍的完整性。毕竟，我们已经在第 3 章中看到，用关系模型描述任何形式的数据，是足够了的也是必需的。我们也知道现在在关系数据库的研究、开发和商业产品中有大量的投资，这些可以称为**关系基础**结构（也就是，支持恢复、并发、安全和优化——更不要说完整性了！——其他的主题我们也在本书中讨论过。）因此，在我们看来，着手开发一种全新的数据库技术是非常不明智的，因为这么做好像是没有任何具有说服力的理由的——更不要说，这样的技术显然也会遇到层次数据库技术所遇到的类似问题（请参考第 13 章或者文献［1.5］或者文献［27.3，27.6］的注释）。

## 27.7　SQL 的支持

在制定现行的 SQL 规范时，并没有支持 XML，但是在下一版本规范中的第 14 部分（可能是 2003 年），SQL/XML［27.15］可能会支持一些此类功能。在本节，我们展望这些功能的概

---

○ 这种方法的一个特例是，将整个文档看作是一个字符串，存在一个元组的某个属性中。注意：这种方法和先前介绍的方法不是互相矛盾的。每一种方法都可能是合适的，这取决于我们要对数据进行怎样的处理。我们甚至可能在同一个文档上同时使用两种方法。

况；然而，这并不等于说，我们所提到的这些功能，直到 SQL/XML 得到正式批准后才能得到支持。

### 1. XML 聚集

注意：原文在本节的某些地方把“SQL”等价于“关系的”，例如 SQL database，SQL data，SQL form 等，译者认为这可能有误。因此我们将上述的词相应地译成关系数据库，关系数据，关系形式。——译者

从我们在前面的章节的讨论中可以知道，有两种基本的方法来存储 XML 数据到一个关系数据库中——XML 聚集与 XML 列，而 SQL/XML 则两者都支持。（由于众所周知的原因，它不支持 native XML 数据库以及类似的数据库。）在本小节中，我们侧重于“XML 聚集”的支持。

我们在第 27.6 节中已经看到“XML 聚集”与 DBMS 没有任何一点关系（它是通过一些应用程序来起作用的，例如运行于 DBMS 之上的 Web 服务器）；你可能会觉得有点奇怪，竟然还要在 SQL 规范里添加支持“XML 聚集”！这些支持的基本构成如下：

- 把关系字符串、标识符、数据类型[⊖]，数值转换成相应的 XML 字符串，名称，数据类型，数据的规则。
- 把一个关系表或者关系表集合转换成两个 XML 文档——其一包含数据之类的东西，另一个则是相应的 XML 模式文档。

这些规则结合在一起，将指导如何把关系数据转换成 XML 形式。等价地，它们也支持我们在前一节所提到的关系数据的 XML 视图（尽管是只读的）。因此它们也提供了在这些数据上处理 XQuery 的基础。但是需要注意，SQL/XML 并没有定义可逆的处理规则：也就是把 XML 数据划分成关系形式（有些少数的例外情况包含了把 XML 字符串和标签转换成关系字符串和标识符的规则）。

借用我们常用的零件关系变量 *P* 来举一个例子，下面是该表的简化了的 SQL 定义：[⊜]

```
CREATE TABLE P
     ( PNUM   CHAR(6),
       PNAME  CHAR(20),
       COLOR  CHAR(6),
       WEIGHT NUMERIC(5,1),
       CITY   CHAR(20) ) ;
```

假设该关系表仅仅含有两个零件 *P*1 和 *P*2，那么它转换成 XML 形式后，如下所示：

```
<P>
  <row>
    <PNUM>P1</PNUM>
    <PNAME>Nut</PNAME>
    <COLOR>Red</COLOR>
    <WEIGHT>12.0</WEIGHT>
    <CITY>London</CITY>
  </row>
  <row>
    <PNUM>P2</PNUM>
    <PNAME>Bolt</PNAME>
    <COLOR>Green</COLOR>
    <WEIGHT>17.0</WEIGHT>
    <CITY>Paris</CITY>
  </row>
</P>
```

同时，也会产生一个模式文档，如下所示：

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:simpleType name="CHAR_6">
```

---

⊖ 编写本书时，还不支持结构类型。

⊜ 这里的简化包括：省略了 PRIMARY KEY 子句——因为文献［17.15］中没有就如何把主码转换成 XML 这个问题进行结论；我们也不考虑用户自定义的类型、NOT NULL 的限定。

```
    <xsd:restriction base="xsd:string">
      <xsd:length value="6"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="CHAR_20">
    <xsd:restriction base="xsd:string">
      <xsd:length value="20"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="DECIMAL_5_1">
    <xsd:restriction base="xsd:decimal">
      <xsd:totalDigits value="5"/>
      <xsd:fractionDigits value="1"/>
    </xsd:restriction>
  </xsd:simpleType>
    <xsd:complexType name="RowType.P">
      <xsd:sequence>
        <xsd:element name="PNUM" type="CHAR_6"/>
        <xsd:element name="PNAME" type="CHAR_20"/>
        <xsd:element name="COLOR" type="CHAR_6"/>
        <xsd:element name="WEIGHT" type="DECIMAL_5_1"/>
        <xsd:element name="CITY" type="CHAR_20"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="TableType.P">
      <xsd:sequence>
        <xsd:element name="row" type="RowType.P"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="P" type="TableType.P"/>

  </xsd:schema>
```

### 2. XML 列

我们接下来讨论 SQL 对 XML 列的支持。SQL/XML 引入了一个新的数据类型，简称为 XML（而不是第 27.6 节中所说的 XMLDOC）。该类型的数值主要是 XML 文档或者 XML 片断——这里的片断这个词，意思是指单个的 XML 元素结点或者元素节点序列。很多的操作将会从传统的关系数据中派生或产生 XML 类型的数据。下面是一个简单的例子：

```
INSERT INTO RESULT ( XMLCOL)
       SELECT XMLGEN ( '<Result>
                          <SNAME>{SX.SNAME}</SNAME>
                          <PNAME>{PX.PNAME}</PNAME>
                          <QTY>{SPX.QTY}</QTY>
                        </Result>',
                        SX.SNAME, PX.PNAME, SPX.QTY ) AS Result
       FROM   S AS SX, P AS PX, SP AS SPX
       WHERE  SX.SNUM = SPX.SNUM
       AND    PX.PNUM = SPX.PNUM ;
```

我们假定关系表 RESULT 的 XMLCOL 列是 XML 类型。这个例子与第 27.5 节中的第一个 XQuery 例子相类似但不矛盾。因为在写本书时 XQuery 还在“不断的完善”中，而 XMLGEN 则是基于 XQuery，这意味着 XMLGEN 在 SQL/XML 规范得到批准以前注定是要改变的。

再重复一次：SQL/XML 引入了新的数据类型 XML；但是它几乎没有定义对该类型数据的操作——但不是绝对的没有[⊖]！事实上，文献［27.15］提到，“如果 *V1* 和 *V2* 都是 XML 类型，那么 *V1* 和 *V2* 是否相同，与实现是相关的。”但是这种状况应该在 SQL/XML 得到正式批准前有

---

⊖ SQL/XML 定义了一个称为 XMLSERIALIZE 的操作，该操作把 XML 类型的数值转换成字符串形式。SQL/XML 还定义了另一个操作，称为 XMLPARSE，来进行相反的操作——也就是说，把字符串形式的 XML 文档或者文档片断转换成 XML 类型的数值。在这转换之间，那两个操作提供了一些划分和发布的功能。

所改进。另外，文献［27.11］建议同时（或者延后一些时间）还要支持下列的功能：

- 提供接口以支持 Xpath 或者 Xquery。
- 检查一个给定的 XML 类型的数值是否是结构良好的 XML 元素，或者是否是一个有效的 XML 文档，或者是否符合某个给定的 XML 模式等。

即使增加了这些功能，SQL/XML 对 XML 数据的访问也只能是只读方式，而不考虑 XML 的大多数 native 访问方式。

**3. 专有的支持**

正如第 27.5 节所提到的那样，一些关系数据库产品（例如 DB2、Oracle）含有专有的支持来访问和更新 XML 数据。在这里我们不打算深入讨论特定的支持情况，而是有选择地举一些例子来说明此类产品通常提供的功能。所举的例子基本上是基于 IBM 的 DB2 产品中的“XML Extender”，但是我们适当地作了简化，删除了与我们讲述目的无关的方面。

“XML Extender”利用 SQL 对用户自定义函数的支持（请参考第 5 章），提供了一套（从用户角度看是事实上的）内嵌函数。这些函数由 SQL 来调用，提供 XML 检索和更新的功能。后面我们要举例说明的函数包括 XMLFILETOCLOB，XMLCONTENT，XMLEXTRACTREAL 和 XMLUPDATE（这些不是它们在系统中真实的名字），它们主要提供如下的功能：

- 把一个文档存储成一个关系属性值。

示例：下面的 UPDATE 语句使用函数 XMLFILETOCLOB 把外部名称为 BoltDrawing. svg 的 XML 文档转换成 CLOB 类型，然后把该 CLOB 字符串存到关系表 *P* 的零件 *P2* 的属性列 DRAWING 中。

```
UPDATE P
SET    DRAWING = XMLFILETOCLOB ( 'BoltDrawing.svg' )
WHERE  PNUM = 'P2' ;
```

当然我们假定关系表 *P* 是有属性列 DRAWING 的，且其数据类型为 CLOB。

- 读取一个 CLOB 类型的属性值。

示例：下面的 SELECT 语句，读取前面例子所存中的 CLOB 型数值，把它以 XML 文档形式发布到名为 RetrievedBoltDrawing. svg 的外部文件。

```
SELECT XMLCONTENT ( DRAWING, 'RetrievedBoltDrawing.svg' )
FROM   P
WHERE  PNUM = 'P2' ;
```

- 从 XML 文档中读取其中一个特定的部分。

示例：我们进一步假设零件信息存放在名为 PartsRelation. xml 的文档中。下面的 UPDATE 语句：（a）从该文档中抽取零件 *P3* 的属性 WEIGHT 的值，把它转换成 REAL 类型；（b）然后把关系表 *P* 中的元组 *P3* 的相应的 WEIGHT 属性值更新为该值：

```
UPDATE P
SET    WEIGHT = XMLEXTRACTREAL
                ( 'PartsRelation.xml',
                  '//PartTuple[PNUM = "P3"]/WEIGHT' )
WHERE  PNUM = 'P3' ;
```

注意我们这里所用的函数——表 *P* 的 WEIGHT 列是 REAL 类型而不是 NUMERIC（5，1），因为 XML Extender 目前还没有支持抽取出 NUMERIC 类型数值的函数。更重要的是，XMLEXTRACTREAL 与其他“extract”函数都可以操作那些存储成关系属性值的 XML 文档，而不仅仅是那些存储在外部的 XML 文档。

- 更新一个 XML 文档中特定的部分。

示例：假设关系表 *SP* 包含类型为 CLOB 的属性列 PARTDETAIL（实际上这是不大可能的），对于给定的元组，该属性列的值为一个 XML 文档，用于描述相应的零件。那么下面的 UPDATE 语句则把该文档中有供应商 S4 供应的零件颜色都更改为绿色。

```
UPDATE SP
SET    PARTDETAIL = XMLUPDATE
```

```
                    ( PARTDETAIL,
                      '//PartTuple/COLOR', 'Green' )
WHERE  SNUM = 'S4' ;
```

## 27.8 小结

本章我们分析了 XML 与数据库之间的关系。为了更好地分析，我们先介绍了很多背景：首先一部分是关于万维网，然后是挥洒更多的笔墨介绍 XML 本身。

XML 源于更早的语言 SGML 与 HTML；名称“XML”是“Extensible Markup Language”的缩写，但是实际上 XML（如同 SGML 一样）是一种**元语言**，甚至可以说是一种用于分析元语言的语言。在本章的开头部分，我们把针对 XML 的应用——也就是一种 **XML 的派生物**，称为一种用于定义某些特殊的 XML 文档的语言。XML 本来与数据库没什么关系，它的提出仅仅是为了“使得 SGML 能像现在的 HTML 一样，可以在 Web 上发送，接收和处理”[27.25]。不过，显然 XML 数据是需要用数据库来存储和处理的；正是这个事实让一些人倡导把 XML 发展成一种新的数据库技术（或者是这种技术的基础），以及诸如此类的东西。

**XML 文档**主要由嵌套的有层次的**元素**结点构成，每一个元素结点有一对划分界限的**标签**。一个给定的元素可以包含**字符数据**，嵌套的元素，或者两者混合存在，也支持**空元素**。开始的标签可有一系列的**属性**结点。我们已经看到，XML 文档可以用来表示关系表，但是它必须对元组强加一种自上而下的顺序，对属性也可能要施加一种从左至右的顺序。

任意给定的 XML 文档都有一个被称为**信息集**的抽象结构，它通过**文档对象模型**（DOM）接口的调用进行处理，也可以用 **XQuery** 进行查询。XML 文档有时也被认为是遵循**半结构数据模型的**，但是实际上，XML 文档的结构与关系一样；事实上，我们看到半结构模型与旧的**层次**模型并没有本质的差别（至少在结构方面）。

任意给定的 XML 文档在定义上都必须是**结构良好**的。它可能是**有效**的，也就是它是符合某些给定的**文档类型定义**（DTD）的。书写 DTD 的规则是 XML 标准的固有组成部分（实际上，一个给定的 DTD 就是从 XML 派生出来的某些定义）。但是 DTD 存在很多的问题；尤其是它在完整性约束方面支持得远远不够。**XML Schema** 是用来生成 XML 模式的元语言，它可以为 XML 文档提供更确切更详细的描述（尤其是在数据类型方面，尽管它还是很难达到第 5 章所描述的情形）。检查一个给定的 XML 文档是否符合一个给定的 XML 模式的过程，就是**模式合法性检查**。

接下来，我们来看看 **XQuery** 和 **XPath**（后者是前者的一个真子集），它们以只读方法来访问 XML 数据——或者，更确切地说是，访问抽象的或解析过的 XML 数据（比如说数据集）。我们没有打算讨论它们是如何定义的，而是通过一些例子来看看它们的功能。我们讨论了**路径表达式**，它有效地让用户按照信息集中某一条特定的路径来导航式地遍历某些目标结点。我们也指出了路径表达式中的**当前结点**所扮演的重要角色，并对这个特征所希望达到的目的提出了质疑（当然它对 XQuery 和 XPath 的“跳到某结点，然后往下遍历”过程化特性起了很多的作用，这正是我们所批判的此类语言的特性之一）。接下来，我们还看到 Xpath 实际上仅仅是**选择结点**而已——它可以在层次数据中导航遍历已有的结点，但是不能构造出新的结点（所以需要 XQuery）。

然后我们列举了很多 XQuery 例子，尤其是用于表现**“flower 表达式”**（“FLOWR” = for + let + where + order by + return）。我们拿这种表达式与关系演算表达式和传统的编程语言的嵌套循环表达式相比较，结果发现后者嵌套循环表达式比关系演算表达式更接近于 flower 表达式。后来我们也考虑了一些优化中的问题。我们注意到 Xquery 没有支持显示的连接操作（如同 SQL 的早期版本一样）。

接下来，我们介绍了如何把 XML 文档存储到数据库中的三种方法：

1）**“XML 属性列”**：我们将整个 XML 文档存储到一个元组的一个属性值中。这种方法需要**一个新的数据类型**（当然还包括基于该类型的变量和数值的操作），称之为 XMLDOC。

2）**“XML 集”**：它把文档**划分**成很多块，把每一块存储到不同关系表中的不同元组的不同

属性值中。注意：划分的相反的操作，就是把非 XML 的数据转换成 XML 形式，称之为**发布**。把这两者结合在一起，就可以在非 XML 数据上实现 **XML 视图**（发布为了检索，划分为了更新）。

3）我们也可以把文档存到 **native XML 数据库**中（这些数据库存放 XML 之类的文档，而不是关系表）。

我们也解释了这些方法的优劣点。

最后，我们简短地讨论了 **SQL/XML**（它将可能在 2003 年加入到新的 SQL 标准中）。SQL/XML 支持把关系数据以 XML 形式发布（尽管在我们看来，还是有些不大合适）。SQL/XML 也引入了一个名为 XML 的新数据类型，此类型的值就是 XML 文档或者文档片断；因此，它可以把 XML 数据存储到关系属性列中，但是它却没有提供对此类数据的相应的操作。临近结束时，我们以 **DB2** 为例，简单地讨论了商用产品对 **XML 的专有支持**。

## 习题

27.1　请用自己的话解释下面的术语：

| | | | | | |
|---|---|---|---|---|---|
| 属性 | 因特网 | 标记 | 网页 | XML | XQuery |
| 元素 | 标记 | URL | 服务器 | XML 派生 | |
| HTML | 搜索引擎 | 浏览器 | 网站 | XML 模式 | |
| HTTP | SGML | 万维网爬取器 | 万维网 | XPath | |

27.2　XML、HTML 和 SGML 是如何联系的？

27.3　请把本书前面的内容列表表示成 XML 文档的形式，并写出它内部的 DTD。

27.4　把练习 27.3 改成外部 DTD 的形式，并解释外部 DTD 的优势。

27.5　一个 XML 文档（a）结构良好（b）有效，是指什么意思？

27.6　什么是空元素？

27.7　你是否认为按照第 25 章的讲解，XML 文档是限制性结构的？

27.8　在第 27.4 节的“DTD 的局限性”小节中，我们指出了 DTD 的明显不足之处——它不是用 XML 描述的。（当然，如果 XML 真如它所声称的那么通用和强大的话，它应该能够描述自己才对。）请问：SQL 的数据定义中也有相似的不足吗？关系数据模型呢？为什么？

27.9　请用 XML 文档的形式表示图 4-5。使用 XML 的元素来表示数据值，不要使用属性。思考唯一性约束可以执行到什么程度？

27.10　重写练习 27.9 的例子，使用 XML 属性来表示数据值。使用属性的优点和缺点各是什么？

27.11　假设练习 27.9 和 27.10 的答案扩展到包括供应商、零件和供货关系，参照约束可以被执行到什么程度？

读者在做练习 27.12 ~ 27.14 时，可能需要参考官方的 XML Schema 文档［27.28］以及其他一些类似的参考文献。（本章的内容不足以完全回答这些问题）

27.12　请为练习 27.3 的答案建立 XML schema。

27.13　请为 27.3 节的 PartsRelation 文档建立 XML schema，不用给 PartTuple 强加顺序。

27.14　我们在 27.4 节讲 XML Schema 数据类型不是一般理解的类型。你同意这个观点吗？请做出解释。

27.15　讲讲你对 infoset 这个词的理解。

27.16　什么是 path 表达式？

27.17　什么是“FLWOR 表达式”？for 子句和 let 子句的关键区别是什么？什么时候应该使用谓词而不是 where 子句？

练习 27.18 ~ 27.21 参考 27.4 节的 PartsRelation 文档。所有结果要求结构良好。

27.18　写出一个 XQuery 语句，列出所有包含 Note 元素的 PartTuple 元素。

27.19　写出一个 XQuery 语句，列出所有绿色的零件，每一个 PartTuple 都嵌套在一个 GreenPart 元素中。

27.20　如果下面的 XQuery 语句作用于一个包含所有六个零件 P1 ~ P6 的 PartsRelation 文档，会得到什么结果？

```
<Parts>
  { count(document("PartsRelation.xml")//PartTuple) }
</Parts>
```

27.21　假设我们，除了 PartsRelation 文档，还给出文档 SuppliersOverShipments（参考图 27-4），写出一个 XQuery 语句，列出至少供应一种蓝色零件的供应商。

27.22　如果练习 27.21 中的 SuppliersOverShipments 文档给出了图 3-8 的所有供应商和供货关系，下面的语句会产生什么结果？

```
for $sx in document("SuppliersOverShipments.xml")/
           Supplier[CITY = 'London']
return
  <Result>
    { $sx/SNUM, $sx/SNAME, $sx/STATUS, $sx/CITY }
  </Result>
```

27.23　下面两个结果子句的语义区别是什么（如果存在区别的话）？

```
return <Result> { $a, $b } </Result>
```

```
return <Result> { $a } { $b } </Result>
```

27.24　我们有什么方法在数据库中存储 XML 数据？每种方法的利弊各是什么？

27.25　考虑一下 27.6 节“专有的支持”小节中所（简单）描述的函数，你对这些函数的设计有什么想法吗？

27.26　有人认为 XML 文档很像关系数据中的元组，请加以比较。

27.27　有时候我们说 XML 数据是无模式的，你怎么样在无模式的数据上执行查询呢？怎么样为这种数据设计查询语言？

27.28　在 27.3 节中我们讲到，文档设计者数据的认知很大程度地影响着 XML 文档的结构。关系数据存在相似的影响吗？如果没有，为什么没有？请解释你的观点。

27.29　如果你熟悉“层次数据模型”，请尽可能地指出它和本章介绍的“半结构模型”的区别。

27.30　下面的文字出自参考文献［27.4］：“XML 避开我们应该做什么这一基本问题，而是完全注重于我们应该怎么做”。请讨论。

## 参考文献


［27.1］　Serge Abiteboul, Peter Buneman, Dan Suciu: *Data on the Web: From Relations to Semistructured Data and XML*. San Francisco, Calif.: Morgan Kaufmann (1999).

“对关系数据和半结构化数据的发展过程和处理策略的最新检阅”（出自出版者的序言）

［27.2］　Tim Berners-Lee with Mark Fischetti: *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. San Francisco, Calif.: Harper San Francisco (1999). See also Tim Berners-Lee: "Information Management: A Proposal," the original document describing the Web project, at *http://www.w3.org/History/1989/Proposal.html*. Versions of this document were circulated for comments at CERN but never formally published.

［27.3］　Phil Bernstein *et al.*: *The Asilomar Report on Database Research*, *ACM SIGMOD Record 27*, No. 4 (December 1998).

这篇文章包含一些关于 XML 对于数据库的消极观点：“不幸的是，XML 会给数据库系统带来混乱。XML 的查询语言是向 25 年前流行的过程化查询语言的倒退。XML 推进了支持更新的客户端数据缓存的发展，这使得 XML 设计者陷入了分布式处理的难题。大部分 XML 的工作没有受到足够的数据库系统团体的支持。”

［27.4］　Bob Boiko: Understanding XML, "*http://mentatorial.com/papers/xml.asp* (2000).

［27.5］　Angela Bonifati and Stefano Ceri: "Comparative Analysis of Five XML Query Languages," *ACM SIGMOD Record* 29, No. 1 (March 2000).

这五种语言是 XSL, XQL, Lorel, XML－QL, XML－GL（XSL 实际上是一种样式单或格式化语言，参看 27.3 节。但是它可以进行简单的查询，就像 XSLT 一样。）XSL 和 XQL 只支持单个 XML 文档的查询，其他的语言支持跨文档的查询。XML-GL 提供一种像 QBE 的界面。Lorel 和 XML-GL 具有更新能力。参看参考文献［27.16］

［27.6］　Jon Bosak and Tim Bray: "XML and the Second-Generation Web," *http://www.scian.com* (May 1999).

这篇文章包含了一个出色的观点，虽然它还没有广泛到国际化：不要使用 XML 作为新的

数据库技术的基础。它提出："XML（文档）具有计算机中所说的树结构……树结构不能表示所有类型的信息，但它可以表示我们需要计算机理解的大部分信息。进一步说，树结构对于编程人员十分方便。如果你的银行账单是树结构的，那么写一个软件来记录交易或显示账单明细就非常简单。"当然，这些观点可能很正确，但是它足够了么？一个关于树（或者说层次）在数据库中历史的研究表明，答案是否定的。基本的观点是，即使数据具有自然的层次结构，就像部门和雇员的例子，也不意味着它应该被表示成层次结构。因为层次的表示方法不适合于我们要对数据进行的所有处理。那么不具有自然层次结构的数据又应该如何呢？例如：命题"供应商 s 供应零件 p 给工程 j"最好的树型表示是什么？注意：我们在 25.1 节与对象的连接中指出了同样的缺陷，它和 XML 文档一样也是层次的。

[27.7] Ron Bourret "XML and Databases" *http://rpbourret.com/xml/XMLAndDatabases.htm* (November 2002).

一个好的导论。它提到"这篇文章给出了一个关于如何与数据库一起使用 XML 的高水平概论。它讲述了……以数据为中心和以文档为中心的区别，XML 一般如何与关系数据库一起使用，以及 native XML 数据库的概念和使用。"

[27.8] Peter Buneman. Wenfei Fan, Jérôme Siméon, and Scott Weinstein: "Constraints for Semistructured Data and XML," *ACM SIGMOD Record* 30, No. 1 (March 2001).

[27.9] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu: "QUILT: An XML Query Language for Heterogeneous Data Sources," in Dan Suciu and Gottfried Vossen (eds.), *Lecture Notes in Computer Science* 1997. NewYork, N. Y. Springer - Verlag (2000).

[27.10] Donald D. Chamberlin: "XQuery: An XML Query Language," *IBM Sys. J.* 41, No. 4 (2002).

Chamberlin 是 SQL 的初始设计者之一 [4.9 ~ 4.11]，他现在是 W3C 中定义 XQuery 的工作组成员。这篇论文是一个有用的指南，虽然它还包含一些有争议的观点，特别是这一点："迭代是查询语言中的重要部分。"这一论断和 Codd 的观点截然相反。Codd 认为："从数据库中抽取任何信息，不管是程序员还是非程序员的用户，都不应使用迭代或者递归的循环"（Codd 的第九次"数据库管理的基本规则"[6.2]）。

[27.11] Andrew Eisenberg and Jim Melton: "SQL/XML and the SQLX Informal Group of Companies," *ACM SIGMOD Record* 30, No. 3 (September 2001); "SQL/XML Is Making Good Progress," *ACM SIGMOD Record* 31, No. 2 (June 2002).

[27.12] Daniela Florescu, Alon Levy, and Alberto Mendelzon: "Database Techniques for the World-Wide Web: A Survey," *ACM SIGMOD Record 27*, No. 3 (September 1998).

这篇论文是关于 Web 数据的，而不是专门介绍 XML 数据的。它对"管理和查询 Web 数据问题的数据库概念合理性"相关的工程、原型和语言作了一个综述。这里包含更多的参考文献。

[27.13] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom: *Database Systems: The Complete Book*. Upper Saddle River, N. J.: Prentice Hall (2002).

[27.14] Elliotte Rusty Harold: *XML Bible* (2d ed.). New York, N. Y.: Hungry Minds, Inc (2001).

[27.15] International Organization for Standardization (ISO): *XML-Related Specifications (SQL/XML) Working Draft*, Document ISO/IEC JTC1/SC32/WG3: DRS - 020 (August 2002).

一篇基于此文档的早期版本的指南可以在参考文献 [26.32] 中找到，也可以参考 [27.11]。

[27.16] Dongwon Lee and Wesley W. Chu: "Comparative Analysis of Six XML Schema Languages," *ACM SIGMOD Record* 29, No. 3 (September 2000).

这六种语言是 XML Schema, XDR, SOX, Schematron, DSD, 以及我们在本章提到的 DTD 定义语言。其中，DTD 是最弱的，XML Schema 是最强的。参考 [27.5]。

[27.17] Philip M. Lewis, Arthur Bernstein, and Michael Kifer: *Databases and Transaction Processing: An Application-Oriented Approach*. Boston, Mass.: Addison-Wesley (2002).

[27.18] Jason McHugh and Jennifer Widom: "Query Optimization for XML," Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland (September 1999).

一个关于 Lore 的优化组件经验的报告，"a DBMS for XML-based data supporting an expressive query language [called Lorel]."

[27.19] Theodor Holm Nelson: "A File Structure of the Complex, the Changing, and the Indeterminate," Proc. 20th

Nat. ACM Conf., Cleveland, Ohio (August 24 - 26, 1965). See Also Theodor Holm Nelson: *Literary Machines*. Sausalito, Calif.: Mindful Press (1993: 1st edition published in 1982).

Nelson 1965 年的论文（其中提出超文本），是建立在前人 Vannevar Bush（Memex，1945）和 Douglas Engelbart（NLS：oNLine System，1963）工作的基础上的。

[27.20] Fabian Pascal: "Managing Data with XML: Forward to the Past?" *http://searchdatabase.techtarget.com* (January, 2001).

在参考文献 [27.6] 中，Bosak 和 Bray 强烈建议取代一部分 DBMS 的功能，交给应用程序处理。Pascal 则认为这样的提议是一种倒退。

[27.21] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld: "Updating XML," Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

提出一组对 XQuery 扩展的更新操作，并把这些更新通过 XML 视图施行到底层关系数据。

[27.22] The Unicode Consortium: *The Unicode Standard, Version* 4.0. Reading, Mass: Addison-Wesley (2003).

[27.23] Jennifer Widom: "Data Management for XML," *http://www-db.stanford.edu/~widom/xml-whitepaper.html* (June17, 1999).

[27.24] W3C: "Document Object Model (DOM) Level 3 Core Specification Version 1.0 Working Draft," *http://www.w3.org/TR/DOM-Level-3-Core.* (February26, 2003).

[27.25] W3C: "Extensible Markup Language (XML) 1.0" (2d ed.) *http://www.w3.org/TR/REC-xml* (October 6, 2000). See also Tim Bray: "The Annotated XML 1.0 Specification," *http://www.xml.com* (follow the link "Annotated XML").

[27.26] W3C: "XML Information Set," *http://www.w3.org/TR/xml-infoset* (October 24, 2001).

[27.27] W3C: "XML Path Language (XPath) Version 2.0 Working Draft," *http://www.w3.org/TR/xpath20* (May 2, 2003).

[27.28] W3C: "XML Schema Part 0: Primer; Part1: Structures; Part2: Datatypes," *http://www.w3.org/TR/xmlschema-0/*, *-1/*, *-2/* (May 2, 2001).

[27.29] W3C: "XQuery 1.0: An XML Query Language Working Draft," *http://www.w3.org/TR/xquery* (May 2, 2003).

注意：下列是与 XQuery 相关的 W3C 文档：

- "XML Query Requirements Working Draft," *http://www.w3.org/TR/xquery-requirements* (May 2, 2003)
- "XQuery 1.0 and XPath 2.0 Data Model Working Draft," *http://www.w3.org/TR/xpath-datamodel* (May 2, 2003)
- "XQuery 1.0 and XPath 2.0 Formal Sementics Working Draft," *http://www.w3.org/TR/xquery-sementics* (May 2, 2003)
- "XQuery 1.0 and XPath 2.0 Functions and Operators Working Draft," *http://www.w3.org/TR/xpath-functions* (May 2, 2003)
- "XML Query Use Cases Working Draft," *http://www.w3.org/TR/xquery-use-cases* (May 2, 2003)

[27.30] XML:DB: "XML Update Language Working Draft," *http://www.xmldb.org/xupdate/xupdate-wd.html* (September 14, 2000).

XML：DB 是一个开放的工业联盟（不是一个标准化组织），它被特许发展 XML 数据库的特殊规范。它成立于 2000 年，因为 "XML 数据库……有比万维网更加广阔的应用。" XUpdate 将是 XML 数据的更新语言。

# 附　录

# 附录 A　TransRelational 模型

## A.1　引言

在科学领域里，不时地会出现一些令人吃惊的新奇思想，由于新思想要好过目前已有的所有理论，因此可以称得上是一种突破。关系模型在数据库的世界里就是这样的一个明显的突破，本书中几乎所有的观点都体现了这一思想。现在我们正看到另一个具有突破性的思想的诞生：**TransRelational™模型**。笔者认为，Steve Tarin 发明的 TransRelational 模型，简称 TR，很可能会被证明成为自 35 年前 Codd 发明关系模型以来数据库领域里最重大的发展。

我们需要立即声明的就是 TR 并不是要成为关系模型的替代品，“TransRelational”中的“Trans”并不像“translunar”中的“Trans”那样代表“超越”，而是指“转换”。TR 和关系模型一样，也是一种数据的抽象模型，但 TR 是更为底层的模型，它更接近物理存储。实际上，TR 是被设计用来作为一种关系模型的实现手段。也许你还记得，我们在第 18 章的结尾曾提到“一种全新的 DBMS 实现方法已经出现，这种方法使得传统实现方法中许多有关底层的假设不再有效”，这种方法就是 TR。

在进入技术细节的讨论之前我们需要了解一些相关的背景。这里的 TR 其实是一种通用技术上的特定应用程序，以其发明者命名，全称是 **Tarin Transform Method**。它是美国的一项专利，是许多系统（不仅仅是 DBMS）——如数据仓库、数据挖掘、SQL 系统、Web 搜索引擎、原生 XML 系统等——中的数据存储和检索的实现技术。但本附录的主题，局限在这一通用技术在关系系统中的应用。但从后面的介绍中可以看出，这一通用技术是最适合关系系统的实现，实际上，它就是在实现关系系统的过程中产生的。

下面我们可以开始进行技术的讨论。了解 TR 的一个好的方法就是从考虑数据的独立性开始（更准确的说应该是物理数据的独立性），数据独立意味着数据在系统的物理层次和逻辑层次上存在着明显的差异，这种差异导致了数据需要在物理层次和逻辑层次之间进行**转换**。目前大多数的 DBMS 系统都采用了一种主码转换的方法，将物理层次上的关系变量映射到逻辑层次上具有相同主码的关系变量。在这样的系统中，物理存储上的数据可以近似的看作用户从逻辑视图上看到的数据的一个**直接影像**（如图 A-1 所示）。从图中可以看出，这样的系统并没有真正提供太多的数据独立性，而且数据需要按照固定的顺序进行物理存储，如果需要以其他的顺序访问数据，就需要提供诸如索引之类的冗余数据结构，为了达到可接受的性能要求还需要提供优化技术，因此数据库管理员的工作难度远远超过了数据管理本身。

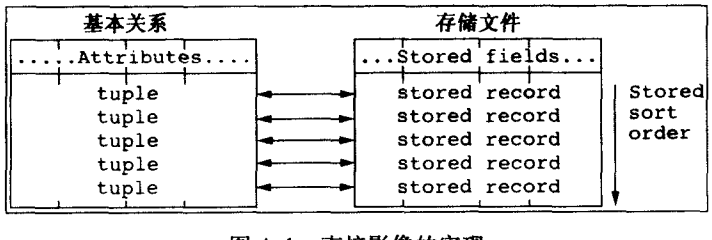



图 A-1　直接影像的实现

相比较而言，TR 中的转换方法要灵活许多，其特点可以概括如下：

- TR 提供更高的数据独立性。
- TR 将数据以多种物理顺序进行有效的存储，不再需要像索引之类的冗余数据结构。
- TR 中的优化技术要远比直接影像系统中的简单，对一种给定的关系操作往往只有一种明显的最好实现方式。
- TR 的运行性能要优于直接影像系统。值得一提的就是，连接操作的复杂度是线性的，也就是说 20 个表连接时间大致等于 10 个表连接时间的两倍，如果真的需要连接 20 个表，那么 TR 将是首选技术，也就是说，TR 系统是可扩展的。
- TR 系统更易于管理，因为它很少需要人的配置选择操作。
- 在整个 TR 系统的物理层次上没有像“存储关系变量”或者是“存储关系元组”之类的概念。

在这个附录里，我们给出了 TR 工作的一个简单描述。当然，我们不可能在这里覆盖 TR 的方方面面，为了限定我们讨论的范围，我们省去了（a）更新，（b）二级存储；也就是说我们假定数据库是（a）只读，（b）在主存中，但请不要认为 TR 仅仅适合只读、主存数据库。要了解 TR 所有方面的知识，包括更新操作和二级存储，请阅读 TR 的用户手册。

## A.2　抽象的三个级别

可以认为使用 TR 技术实现的关系系统包含三个级别的抽象：关系级别（或用户级别），文件级别和 TR 级别（如图 A-2 所示）

- 在最高的级别上，数据被表示成关系表，由通常意义上的元组和属性组成。
- 在最低的级别上，数据被表示成一种称为表的 TR 内部结构，这种表也是由行和列组成。请注意这里表以及行和列不是 SQL 结构中相应概念，它们之间也没有直接的对应关系。
- 中间级别在最高与最低级别的数据模式转换上起一个过渡的作用：高层的关系表被映射到这一层的文件，然后文件再被映射到底层的表结构。这一层的文件由记录和字段组成；其中记录对应上层中的元组，字段对应上层的属性列。注意：不要误认为这里的文件就是物理的存储方式；它们仍然是物理存储的一种抽象，就跟关系变量一样（TR 表结构也是这样），但可以认为它们比关系变量要更接近物理存储（但与 TR 表相比则要远一些）。

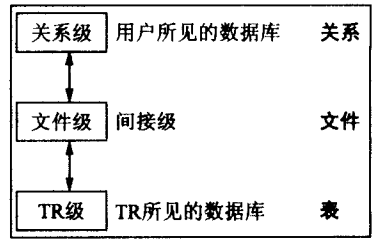



图 A-2　抽象的三个级别

从现在起，我们应该小心的区分一些术语：最高级别上的关系，中间级别上的文件以及最低级别上的表。为了简单起见，本文后面所提到的关系变量和关系默认为是指底层的关系变量和关系，除非我们作显式的声明。

将关系变量映射到 TR 表示的第一步，就是将关系先映射到文件，按照元组对应记录，属性列对应记录字段的映射方式。例如，图 A-3 就表示了一种文件到我们常用的供应商信息表的映射方式。在文件中，如图中的记录号和字段号所示，记录从上到下排列，记录中的字段从左往右排列。尽管如此，记录的次序和字段的次序实际上可以是任意的次序；因此，A-3 中的供应商信息表可以等价的映射到 2880 种不同的文件——5 个记录的全排列合 120 种排法[⊖] 以及 4 个字段的全排列合

| 字段序列： | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 记录序列： | S# | SNAME | STATUS | CITY |
| 1 | S4 | Clark | 20 | London |
| 2 | S5 | Adams | 30 | Athens |
| 3 | S2 | Jones | 10 | Paris |
| 4 | S1 | Smith | 20 | London |
| 5 | S3 | Blake | 30 | Paris |

图 A-3　文件与供应商信息表的映射

⊖ 并不是所有 120 种排序都能够通过简单的 ORDER BY 语句来得到（比如说，图 A-3 中所示的次序就无法得到）。

24 种排法。从表示了相同信息的角度来看，这 2880 个不同的文件是彼此等价的。因此，有时把它们看作是一个“相同文件”的 2880 个不同的版本会比较方便。

图 A-3 中所示的文件现在可以表示成 TR 层上的表结构，同时也可以从 TR 表结构中重构出来。实际上，相同文件的多个不同版本都可以从相同的 TR 表结构中轻松重构出来（术语“版本”上面已经解释过——就是记录和字段的次序不同，但内容却是相同的）；我们在下一节就可以看到这一过程是如何工作的。在 TR 的表结构中，行按自上而下的次序组织，列按自左向右的次序组织。行和列的交点，我们称之为单元，可以通过[$i$, $j$]形式的脚注来表示，其中，$i$ 表示行号，$j$ 表示列号。

文件到 TR 表结构的映射细节我们将在下一节给出。这里我们只是强调这种映射完全不同于 A.1 节所讨论的直接影像的映射方式。TR 表结构中的行和文件中的记录不存在一对一的关系，因此与关系中的元组也不存在一对一的关系。图 A-4 给出了与图 A-3 相对应的 TR 表结构——**字段值表**，对照 A-3，A-4 可以发现，A-4 中行和 A-3 中文件记录没有明显的对应关系。

为了能够从 A-4 中的字段值表重构出 A-3 中所示的文件，我们还需要另外一个表——**记录重构表**（如图 A-5 所示）。注意，在图 A-5 的表格中的值不再是供应商的编号和状态值等——尽管列的名称是这样标记的——而是行的编号。进一步的解释见下一节。

| 列序列： | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 行序列： | S# | SNAME | STATUS | CITY |
| 1 | S1 | Adams | 10 | Athens |
| 2 | S2 | Blake | 20 | London |
| 3 | S3 | Clark | 20 | London |
| 4 | S4 | Jones | 30 | Paris |
| 5 | S5 | Smith | 30 | Paris |

图 A-4 图 A-3 文件的字段值表

| 列序列： | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 行序列： | S# | SNAME | STATUS | CITY |
| 1 | 5 | 4 | 4 | 5 |
| 2 | 4 | 5 | 2 | 4 |
| 3 | 2 | 2 | 3 | 1 |
| 4 | 3 | 1 | 1 | 2 |
| 5 | 1 | 3 | 5 | 3 |

图 A-5 图 A-3 文件的记录重构表

## A.3 基本思想

TR 模型下的重要思想可以归纳如下，令 $r$ 表示某个文件中的一个记录：

*$r$ 的存储形式中包括两个逻辑独立的方面，一个是记录中的字段值，一个是将这些字段值链接在一起的链接信息，对这两个方面有很多种相应的存储方式。*

在直接影像系统中，$r$ 的这两方面的信息是存放在一块的；也就是说，这样的系统中的链接信息是由物理存储上的邻近关系来表示的。在 TR 中，这两方面的信息是分离的——字段值存放在字段值表中，链接信息存放在记录重构表中。这种信息的分离正是 TR 技术能够带来各种好处的根基。

**1. 字段值表**

现在，可能你已经了解到如何从图 A-3 中的文件得到图 A-4 中的字段值表：字段值表中的每一列包含了文件中的相应字段的值，并按升序作了重新的排列。因此不论文件中的记录初始时是什么样的次序，我们都可以得到相同的字段值表（在我们的例子中，文件的 2880 个版本对应同一个字段值表）。尽管现在我们还没有描述这种表如何使用（我们还必须先讨论记录重构表），但可以先看看这种表给我们带来的非常直观的好处：

- 字段值表中每列的值都是有序的，可以直接用于回答用户带 ORDER BY 的查询请求。例如，一个按 city 名排序的查询请求就不需要在运行时间的排序操作，也不需要用到索引。
- 按 city 名逆序排序的查询请求同样如此，实现只需按自底向上的顺序处理字段值表即可。
- 相似的结论可以应用到每个单一属性上；也就是说，字段值表同时有效地表示了多种不同的排序（每个属性上两个方向上的排序）。
- 查找特定值的查询——例如，查询 London 的供应商记录——可以使用有效的二分查找方法。这一点同样适用于每一个属性。

最后，我们可以观察到字段值表可以看作是在用户数据视图（也就是用户级别上的关系表）和 TR 内部结构之间的一个桥梁。字段值表是唯一包含用户数据的 TR 表——所有其他的 TR 信

息都不是与用户直接相关或暴露给用户。

**2. 记录重构表**

图 A-6 并排展示了图 A-4 的字段值表和图 A-5 的记录重构表。可以看出这两个表是同型的，表格之间存在直接的一对一关系（也就是说这两个表的行数和列数分别与图 A-3 中的记录数和记录中的字段数相等）。但正如 A.2 节所提到的，在记录重构表中的数据不是表示供应商编号或是供应商名等，它们只是**行编号**，而且这些行编号可以被认为是指向字段值表或是记录重构表中某些行的**指针**，具体指向哪个表要根据它使用的上下文环境来定。（因为这个原因，记录重构表中的列是不应该被标记上 S#，SNAME 等名字的；但暂时标上这些标记有助于后面的解释。）

在我们解释记录重构表如何构建之前，我们先看看它是如何被使用的。请看下面的操作过程：

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | S# | SNAME | STATUS | CITY |
| 1 | S1 | Adams | 10 | Athens |
| 2 | S2 | Blake | 20 | London |
| 3 | S3 | Clark | 20 | London |
| 4 | S4 | Jones | 30 | Paris |
| 5 | S5 | Smith | 30 | Paris |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | S# | SNAME | STATUS | CITY |
| 1 | 5 | 4 | 4 | 5 |
| 2 | 4 | 5 | 2 | 4 |
| 3 | 2 | 2 | 3 | 1 |
| 4 | 3 | 1 | 1 | 2 |
| 5 | 1 | 3 | 5 | 3 |

图 A-6　图 A-4 的字段值表和其相应的记录重构表

第一步：从字段值表中的［1，1］单元中取出其值：也就是供应商名 S1。这个值是供应商文件中某个记录的第一个字段值（S#字段值）。

第二步：从记录重构表中相同的单元（［1，1］单元）中取出其值：也就是行编号 5，这个行编号就是重构记录**第二个字段**（即 SNAME 字段）的值在字段值表中位置，即位置［5，2］，到字段值表的这个位置上取出存储值（即供应商名 Smith）作为这个重构记录的 SNAME 字段值。

第三步：从记录重构表的［5，2］单元中取出行编号 3，即重构记录**第三个字段**（即 STATUS 字段）的值在字段值表中的位置，即位置［3，3］，到字段值表的这个位置上取出存储值（即 status 20）作为这个重构记录的 STATUS 字段值。

第四步：从记录重构表的［3，3］单元中取出行编号 3，即重构记录**第四个字段**（即 CITY 字段）的值在字段值表中的位置，即位置［3，4］，到字段值表的这个位置上取出存储值（即 city London）作为这个重构记录的 CITY 字段值。

第五步：从记录重构表的［3，4］单元中取出行编号 1。到此为止，供应商信息记录的下一个需要填充的字段应该是第五个字段，但供应商信息记录本身才四个字段，因此这第五个字段又循环返回到第一个字段。因此，重构的供应商记录的下一个字段（字段 S#）值在字段值表的第一行——也就是单元［1，1］。而这个位置正是我们的出发点，因此处理过程结束。

很清楚，前面的这个操作序列重构了供应商文件的一个记录，即图 A-3 中的第 4 号记录：

| | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| 4 | S1 | Smith | 20 | London |

顺便要指出的是，要注意在前面的这个例子中我们所用到的行编号指针是如何构成一个闭环的——实际上，是两个同构的环，一个是在字段值表中，一个是在记录重构表中（如图 A-7 所示）。注意，由于某种原因，这种环被称为 Z 字环，重构算法也因此被称为 Z 字环算法。

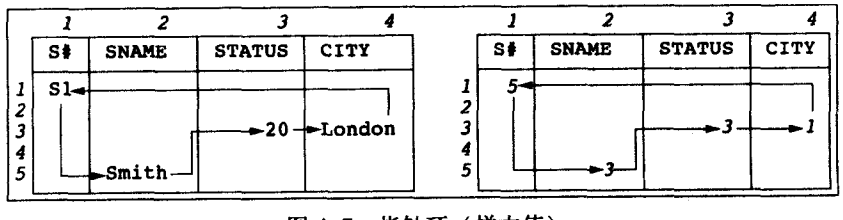



图 A-7　指针环（样本值）

作为一个练习，请试着自己重构另外一个供应商记录。如果你从字段值表中的单元［2，1］开始，你会得到图 A-3 中的记录 3；从单元［3，1］开始，将得到记录 5；从单元［4，1］开始，将得到记录 1；从单元［5，1］开始，将得到记录 2。如果我们按供应商编号自顶向下的次序处理整个字段值表——也就是说如果我们执行五次记录重构过程，按照单元［1，1］，［2，1］，［3，1］，［4，1］和［5，1］的次序——那我们就能够重构出整个供应商文件的一个版本，并且文件中的记录是按照供应商编号的升序排列。换句话说，我们实现了如下的 SQL 查询：

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM   S
ORDER  BY S# ;
```

同样的，也实现了如下的查询——

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM   S
ORDER  BY S# DESC ;
```

我们所要做的就是按供应商编号的逆序来处理字段值表，也就是按单元[5，1]然后[4，1]的次序来完成记录的重构过程。我们不需要作运行时间的排序，也不需要用到索引。

更进一步来说，因为记录重构表中的指针构成一个环，我们可以从环上任何一点进入到环中。因此当我们应用重构算法时，我们可以选择任意的起点单元。例如，如果我们从单元［1，3］——STATUS 列的第一个单元——开始，我们得到如下的记录：

| | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| 3 | S2 | Jones | 10 | Paris |

（更准确的说，我们得到了一个按字段从左到右的次序的记录，首先是 STATUS，然后是 CITY、S#、SNAME。）在 STATUS 属性列上继续上述过程——也就是按单元［2，3］，［3，3］，［4，3］和［5，3］的次序执行重构过程——我们最终可以得到按 status 属性升序排列的供应商信息文件：

```
SELECT S.STATUS, S.CITY, S.S#, S.SNAME
FROM   S
ORDER  BY STATUS ;
```

以相同的方式，如果我们在 SNAME 属性列上执行记录重构过程，那我们就可得到按供应商姓名升序排列的文件；同样的，如果我们在 CITY 属性列上执行记录重构过程，那我们就可得到按城市名升序排列的文件。也就是说，记录重构表和字段值表一起同时代表了所有的排序方式，而且没有用到任何的索引和运行时间的排序。

接着让我们考察如下的查询，它包含了一个简单的等值约束：

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM   S
WHERE  S.CITY = 'London' ;
```

由于 CITY 属性列在字段值表中是有序的，因此可以使用二分查找法找出包含 London 的单元。以图4-6 的字段值表为例，二分查找法找出的单元就是［2，4］和［3，4］。Z 字环就可以通过从［2，4］和［3，4］单元开始的指针链构造出来。在这个例子中，构造出的 Z 字环如下所示：

```
[2,4], [4,1], [3,2], [2,3]
```

和

```
[3,4], [1,1], [5,2], [3,3]
```

将这些 Z 字环用相应字段值表中的值替换，我们就可以得到查询结果记录：

| | S# | SNAME | STATUS | CITY |
|---|---|---|---|---|
| 1 | S4 | Clark | 20 | London |
| 4 | S1 | Smith | 20 | London |

除了简单的 ORDER BY 和等值约束外，字段值表和记录重构表一起还为许多其他的用户级操作提供了直接的支持。实际上，大多数的基本关系操作——选择，投影，连接等（这里没有提到复制和消重，它们属于系统的内部操作）——的实现算法都依赖于以某种次序访问数据的方法。以连接操作为例，在 18 章中我们看到归并排序是一种实现连接的很好的方法，但 TR 技术使我们可以不做排序——至少是不做运行时间的排序——就能完成归并连接（在字段值表和记录重构表构建的过程中就已经完成了排序）。比如说，在供应商表的 CITY 字段上做连接，我们只需简单的按 CITY 字段顺序访问两个字段值表，然后做归并连接。

综上所述，TR 技术将大大简化优化系统的工作，访问路径的选择程序（见第 18 章）也将变的更加简单——在某些情况下是完全不需要的。TR 技术也不再需要传统 DBMS 系统中的诸如索引，哈希表等辅助数据结构。因为省去了很多的选项，数据库系统的设计也变得更加简单，性能的调优也因此而变得简单。

**3. 记录重构表的建立**

考察对图 A-3 中的表按不同的属性列排序的效果。比如说，我们按供应商编号的升序来排列记录，那我们可以得到 4，3，5，1，2 的记录顺序。我们称这种序列为“S#升序排列”（简称为 S#排列）。其他的排列如下所示：

- Ascending SNAME : *2, 5, 1, 3, 4*
- Ascending STATUS : *3, 1, 4, 2, 5*
- Ascending CITY : *2, 1, 4, 3, 5*

这些排列可以由如下的排列表来表示，表中的 $[i, j]$ 单元存放的是原供应商文件的记录号，表示将原供应商文件按第 $j$ 个字段排序时 $[i, j]$ 中的记录将出现在第 $i$ 个位置。

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | S# | SNAME | STATUS | CITY |
| 1 | 4 | 2 | 3 | 2 |
| 2 | 3 | 5 | 1 | 1 |
| 3 | 5 | 1 | 4 | 4 |
| 4 | 1 | 3 | 2 | 3 |
| 5 | 2 | 4 | 5 | 5 |

如上表所示，S#排列就是如下的序列：

*4, 3, 5, 1, 2*

上述排列的相反排列如下所示：

*4, 5, 2, 1, 3*

这个相反的排列就是将上述排列表中 S#下的记录号重新排成 1，2，3，4，5 的序列而得到的。（比如说上述表中 S#下的记录编号次序为 4，3，5，1，2，第四个记录号为 1，第五个为 2，第二个为 3 等等），更一般地说，如果我们把任意一个排列看作是一个向量 $V$，那么相反的排列 $V'$ 可以根据这样的简单规则得到：如果 $V[i] = i'$，那么 $V'[i'] = i$。将这个规则应用到我们给定排列表中的每一列，我们可以得到如下的反向排列表：

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | S# | SNAME | STATUS | CITY |
| 1 | 4 | 3 | 2 | 2 |
| 2 | 5 | 1 | 4 | 1 |
| 3 | 2 | 4 | 1 | 4 |
| 4 | 1 | 5 | 3 | 3 |
| 5 | 3 | 2 | 5 | 5 |

现在我们可以开始构造记录重构表了。比如说，属性列 S#可以按如下的步骤构造出来：

找到反向排列表的 [*i*, *1*] 单元，假定那个单元里的值为 *r*，并假定其右边的单元，即 [*i*, *2*] 里的值为 *r'*。找到记录重构表的第 *r* 行，并将 *r'* 的值填入 [*r*, *1*] 单元。

从 $i=1$ 到 $i=5$ 循环执行上述算法即可产生出记录重构表的整个 S#列值。其他属性列值可以类似地产生。注意：作为一个练习，我们强烈建议你实际操作一下上述算法，构造出完整的记录重构表，它可以帮助你理解这个算法。顺便再提一下，记录重构表的建立不可约依赖于原文件，与字段值表没有任何关系。

**4. 记录重构表不是唯一的**

在我们前面小节的讨论中，我们提到图 A-3 中的供应商文件的 CITY 字段排列是 2，1，4，3，5。注意，供应商 1 和 S4 都处在同一个城市，S2 和 S3 也是，因此我们可以认为 CITY 排列为 2，4，1，3，5 或者是 3，1，4，2，5 或者是 3，4，1，2，5。换句话说，CITY 排列不是唯一的[⊖]。那也就是说排列表不是唯一的，因此记录重构表也不是唯一的。尽管如此，对于一个给定的用户级上的关系表，总是存在某些记录重构表是最合适的，因为它们表现了一些一般的记录重构表所没有的属性。这些细节的讨论超出本附录的范围，进一步的讨论可参考文献 [A.1]。

## A.4 列的压缩

观察图 A-8，它表示了我们的一个常用零件表的文件结构；图 A-9，表示了相应的字段值表；图 A-10，表示了一个最优记录重构表。

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | P# | PNAME | COLOR | WEIGHT | CITY |
| 1 | P1 | Nut | Red | 12.0 | London |
| 2 | P2 | Bolt | Green | 17.0 | Paris |
| 3 | P3 | Screw | Blue | 17.0 | Oslo |
| 4 | P4 | Screw | Red | 14.0 | London |
| 5 | P5 | Cam | Blue | 12.0 | Paris |
| 6 | P6 | Cog | Red | 19.0 | London |

图 A-8 常用表的文件结构

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | P# | PNAME | COLOR | WEIGHT | CITY |
| 1 | P1 | Bolt | Blue | 12.0 | London |
| 2 | P2 | Cam | Blue | 12.0 | London |
| 3 | P3 | Cog | Green | 14.0 | London |
| 4 | P4 | Nut | Red | 17.0 | Oslo |
| 5 | P5 | Screw | Red | 17.0 | Paris |
| 6 | P6 | Screw | Red | 19.0 | Paris |

图 A-9 图 A-8 的字段值表

现在观察图 A-9 所示的字段值表，它含有很多的冗余——比如说 city 字段中的 London 出现了三次，weight 字段中的 17.0 出现了两次，等等。压缩这个表的列就是删除这些冗余，结果就是表中的每一列只包含了彼此不同的值，如图 A-11 所示，对其的讨论如下：

1）有选择性的运用压缩过程是合法的，也是必要的。在属性列 P#上就没有压缩点，因为零件表的编号是唯一的。

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | P# | PNAME | COLOR | WEIGHT | CITY |
| 1 | 4 | 3 | 2 | 1 | 1 |
| 2 | 1 | 1 | 4 | 6 | 4 |
| 3 | 5 | 6 | 5 | 2 | 6 |
| 4 | 6 | 4 | 1 | 4 | 3 |
| 5 | 2 | 2 | 3 | 5 | 2 |
| 6 | 3 | 5 | 6 | 3 | 5 |

图 A-10 图 A-8 的记录重构表

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | P# | PNAME | COLOR | WEIGHT | CITY |
| 1 | P1 | Bolt | Blue | 12.0 | London |
| 2 | P2 | Cam | Green | 14.0 | Oslo |
| 3 | P3 | Cog | Red | 17.0 | Paris |
| 4 | P4 | Nut | | 19.0 | |
| 5 | P5 | Screw | | | |
| 6 | P6 | | | | |

图 A-11 图 A-4 的字段值表压缩版

2）在压缩列中，零件文件的不同记录是共享字段值的。比如说，单元 [1，5] 中的 city 名 London 就是被三个零件记录共享的：也就是 P1，P4 和 P6。由于列的这种压缩，更新操作，尤其是 INSERT，将比以往运行的更快，因为他们可以使用已经存在的字段值；可以考察用户插入这样一个元组的情况：编号列 P7，name 列 Nut，color 列 Red，weight 列 18.0，city 列 London。正

⊖ STATUS 排列具有与之相同的情况，但 SNAME 排列不是这种情况。（很明显，S#也不是。）

如我们在 A.1 节已经说明过的，更新操作超出了本附录的讨论范围。

3）这里的压缩列体现了数据的一种压缩方式（尽管这种方式在传统的直接影像实现中并不常见）——请注意这种方式能够达到多大的压缩比。比如说，想象一个记录了司机执照信息的机动车表，每个元组中的执照发放地信息都是 California，一共大约有 2 千万个元组，但很可能没有 2 千万个不同的身高信息，或是 2 千万个不同的体重信息，或是发色信息，或是执照失效日期。换句话说，压缩比可能达到一百万比 1。

**1. 行的区间**

回到图 A-11，无需说明，我们不能简单的将原来在 city 字段出现了三次的 London 值替换成图 A-11 中一次出现，这样将会丢失信息。因此我们需要在压缩列中加入一些额外的信息，使得我们可以由压缩列表可以重构出原表。一种方式就是在压缩列的每个字段值旁加上在原表中出现了该字段值的行的区间，如图 A-12 所示：

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | P# | PNAME | COLOR | WEIGHT | CITY |
| 1 | P1 | Bolt [1:1] | Blue [1:2] | 12.0 [1:2] | London [1:3] |
| 2 | P2 | Cam [2:2] | Green [3:3] | 14.0 [3:3] | Oslo [4:4] |
| 3 | P3 | Cog [3:3] | Red [4:6] | 17.0 [4:5] | Paris [5:6] |
| 4 | P4 | Nut [4:4] | | 19.0 [6:6] | |
| 5 | P5 | Screw [5:6] | | | |
| 6 | P6 | | | | |

图 A-12 带行区间的压缩字段值表

举例来说，考虑图中的［3，4］单元，其中的值是 weight 字段值 17.0，并带有行区间［4:5］。行区间的意思是指，如果这个字段值表没有被压缩，那么 weight 字段值 17.0 将在 weight 列的第 4 到第 5 行出现。

顺便提一下，不要混淆［4:5］和［4，5］两种书写形式，前者（带有冒号）是指行的区间范围，后者（带有逗号）表示的是表格中的行值和列值。

对这个行区间还有另一种看法，以图 A-12 中的 COLOR 属性列为例，它表达两层意思：（a）在当前的零件文件中出现的 COLOR 值；（b）每一个属性值出现的次数。也就是说，每一列都可以被看成是一个柱状图，如图A-13 所示。一般的，带行区间的整个压缩字段值表可以被看作是一个柱状图的集合，一个柱状图都对应一个压缩列。因此，所有涉及这些柱状图的查询将得到性能上的提高（比如说查询“每一种颜色有多少个零件记录?”），见A.6 节。

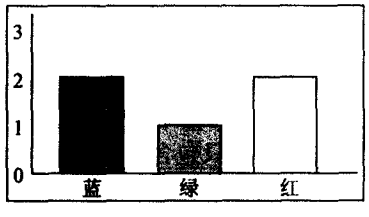



图 A-13 color 柱状图（基于图 A-12）

如果字段值表可以被看作是一组柱状图的集合，那么记录重构表，就可以被看作是一组排列的集合。比如说，如果我们使用图 A-10 记录重构表的第 3 个属性列来重构零件文件，那我们可以得到一个按 color 排序的零件文件；也就是说，我们得到了那个文件的“COLOR 排列”。

因此，我们可以认为任意给定数据集的 TR 表示是一个柱状图集合和一个排列集合。

**2. 记录重构算法的修正**

字段值表的压缩破坏了它与记录重构表表格之间的一对一关系，那么我们曾经使用过的记录重构算法需要做一些修正，但这个工作并不复杂：

*考察记录重构表中的［$i$，$j$］单元，现在不再去找字段值表的［$i$，$j$］单元，而是找单元［$i'$，$j$］，该单元中的行区间中包含了行 $i$。*

举例来讲，考察图 A-10 记录重构表中的［3，4］单元，它出现在第四列，也就是 WEIGHT 列。为了在图 A-11 的字段值表中找到相应的 weight 值，我们搜索字段值表的 WEIGHT 列，找到行区间中包含了行值 3 的单元，在该图中是单元［2，4］（相应的行区间是［3:3］），其中的 weight 值为 14.0。作为练习，请使用图 A-10 中的记录重构表和图 A-12 中的压缩字段值表重构

整个零件文件（为了得到按 city 字段升序排列的文件请从第 5 列开始）。

## A.5 列的合并

在前面的一小节中，我们讨论了压缩列，一种在记录间共享字段值的方法——这些记录都是来自于同一个文件。列的合并，可以看作是在相同或不同的文件的记录间共享字段值的方法。基本的想法是，如果数据的类型都相同，那么文件级别上的不同字段可以映射到 TR 级别上字段值表的相同属性列上。

我们先讨论只有一个文件的例子，考察图 A-14 中的 bill-of-materials 关系表（第 4 章图 4-6 的一个变形）。首先，我们先看看没有列合并的情况；然后再给出应用了列合并的技术后情况的变化。图 A-15 表示了与图 A-14 相应的一个文件结构；图 A-16 表示了相应的压缩字段值表；图 A-17 则表示了一个相应的最优记录重构表。

MMQ

| MAJOR_P# | MINOR_P# | QTY |
|---|---|---|
| P1 | P2 | 2 |
| P1 | P3 | 4 |
| P1 | P4 | 1 |
| P2 | P3 | 3 |
| P2 | P4 | 8 |
| P2 | P5 | 6 |
| P3 | P4 | 3 |
| P3 | P6 | 4 |
| P5 | P6 | 3 |

图 A-14 bill-of-materials 关系表 MMQ

| | 1 | 2 | 3 |
|---|---|---|---|
| | MAJOR_P# | MINOR_P# | QTY |
| 1 | P3 | P4 | 3 |
| 2 | P1 | P3 | 4 |
| 3 | P2 | P4 | 8 |
| 4 | P1 | P4 | 1 |
| 5 | P2 | P5 | 6 |
| 6 | P3 | P6 | 4 |
| 7 | P1 | P2 | 2 |
| 8 | P5 | P6 | 3 |
| 9 | P2 | P3 | 3 |

图 A-15 图 A-14 关系表的一种文件结构

| | 1 | 2 | 3 |
|---|---|---|---|
| | MAJOR_P# | MINOR_P# | QTY |
| 1 | P1 [1:3] | P2 [1:1] | 1 [1:1] |
| 2 | P2 [4:6] | P3 [2:3] | 2 [2:2] |
| 3 | P3 [7:8] | P4 [4:6] | 3 [3:5] |
| 4 | P5 [9:9] | P5 [7:7] | 4 [6:7] |
| 5 | | P6 [8:9] | 6 [8:8] |
| 6 | | | 8 [9:9] |

图 A-16 图 A-15 文件的压缩字段值表

| | 1 | 2 | 3 |
|---|---|---|---|
| | MAJOR_P# | MINOR_P# | QTY |
| 1 | 1 | 2 | 3 |
| 2 | 2 | 6 | 1 |
| 3 | 4 | 3 | 4 |
| 4 | 3 | 1 | 7 |
| 5 | 5 | 9 | 9 |
| 6 | 7 | 4 | 2 |
| 7 | 6 | 8 | 8 |
| 8 | 8 | 7 | 6 |
| 9 | 9 | 5 | 5 |

图 A-17 图 A-15 文件的记录重构表

我们现在开始讨论列的重构，观察图 A-14 的关系表 MMQ，很明显属性列 MAJOR_P#和属性列 MINOR_P#具有相同的类型，因此在文件中与之相对应的字段也就具有相同的类型。于是，可以将它们对应到字段值表中的相同的属性列。如图 A-18 所示，讨论如下：

1）属性列 MAJOR_P#和属性列 MINOR_P#被合并成同一列，该列包含了原 MAJOR_P#和 MINOR_P#属性列中所有的字段值（也就是零件编号），冗余部分被删除。

2）合并列中的每一个单元包含了单一的零件编号以及两个行区间。第一个行区间对应于未压缩的字段值表中的“major”编号，第二个行区间对应于“minor”编号。这里的行区间——除了空区间“[:]”外——等同于我们先前讨论的字段值表中的行区间，空区间表示在未压缩的字段值表中的相应行没有对应的零件编号（比如说，在“minor”编号中就从没有 P1）。

3）在合并的表中，合并的列是第一列，QTY 属性列是第二列（因此在合并表中，列的俄数量是 2，不是 3），属性列 QTY 等同于图 A-16 中 QTY 列。

| | 1 | 2 |
|---|---|---|
| | MAJOR_P# + MINOR_P# | QTY |
| 1 | P1 [1:3] [ : ] | 1 [1:1] |
| 2 | P2 [4:6] [1:1] | 2 [2:2] |
| 3 | P3 [7:8] [2:3] | 3 [3:5] |
| 4 | P4 [ : ] [4:6] | 4 [6:7] |
| 5 | P5 [9:9] [7:7] | 6 [8:8] |
| 6 | P6 [ : ] [8:9] | 8 [9:9] |

图 A-18 图 A-16 中合并了前两个属性列的字段值表

4）记录重构表保持原样[⊖]。但其第1，第2列对应于合并字段值表的合并列，第1列对应合并列中第1个行区间，第2列对应第2个行区间，第3列对应字段值表的第2列。

列合并的思想有很多的优点，这里我们就给出一个实现连接操作的例子以示说明。假如我们要做一个表 MMQ 的自连接，设连接中的 MMQ 表分别为 MMQa 和 MMQb，连接的条件是 MMQa. minor = MMQb. major。那么我们只需在合并的字段值表上作一次遍历就能给出哪些元组是可以做连接的。比如说，合并字段值表中的第3行，其合并列中含零件编号 P3 以及“minor”区间［2:3］，“major”区间［7:8］，马上就可以得出结论：MMQa 中的第2和第3行可以与 MMQb 中的第7和第8行做连接。其他行可依此类推。实际上，在这种情况下，我们可以不作排序和合并而直接进行排序/归并连接[⊖]（正如本附录前面曾提到的）。

注意：上述结论并不能应用于表中任何非合并列，原因如下所述：

- 令 F1 为图 A-18 中的字段值表和图 A-17 中的记录重构表通过自顶向下的次序处理 MAJOR_P#属性列而得到的文件。那么 MMQa 所对应的文件即为 F1，即 MMQa 中的第 $i$ 个元组是唯一对应于文件 F1 中第 $i$ 个记录的元组。
- 同样的，令 F2 为图 A-18 中的字段值表和图 A-17 中的记录重构表通过自顶向下的次序处理 MINOR_P#属性列而得到的文件。那么 MMQb 所对应的文件即为 F2，即 MMQb 中的第 $i$ 个元组是唯一对应于文件 F2 中第 $i$ 个记录的元组。

最后，我们还要强调一次，列合并的思想不仅可以应用于单个文件，同样可以应用于多个文件。以供应商信息表和零件表为例，对整个数据库，我们可以只要一个字段值表，一列是供应商编号，一列是零件编号，一列是 city 名等等。实际上，TR 技术允许在字段值表中包含数据库里任何关系表可能都没有的字段值，因此我们可以认为 TR 是真正的“面向域”的数据库表示方式，见参考文献［A.1］中的进一步讨论。

## A.6　关系操作的实现

在本节中，我们将简要的讨论 TR 技术对某些关系操作的实现。我们的讨论基于供应商－零件－项目数据库实例（实例中的值如图 A-19 所示）。合并压缩的字段值表如图 A-20，一个最优的记录重构表如图 A-21。

### 1. 约束

考虑如下的约束查询[⊜]：

```
SPJ WHERE QTY = 200
```

为实现这个查询，我们在（图 A-20）字段值表的 QTY 属性列上作二分查找，找到包含值 200 的单元；注意如果这样的值存在的话，那它必定是唯一的，因为列是压缩过的。如果没找到，那我们马上就知道查询结果为空。在当前这个例子中，我们将找到［2, 7］单元，除了相应的字段值，还包含行区间［3:6］。那么货运表的记录重构表中的单元［3, 7］,［4, 7］,［5, 7］,［6, 7］:

a. 包含了字段值表中含 QTY 属性值为 200 的单元的行编号（他们的确都包含了行编号2）。

b. 包含了货运记录重构表的“下一个”单元的行编号。

Z 字环因此可以根据货运的记录重构表的指针环构造出来。在这个例子中，构造出的 Z 字环如下所示：

```
[3,7], [1,1], [2,5], [2,6]
[4,7], [3,1], [3,5], [3,6]
[5,7], [8,1], [7,5], [4,6]
[6,7], [9,1], [9,5], [6,6]
```

⊖ 实际上有许多改进的方法，但这些方法不在本附录的讨论范围之内。

⊖ 更准确的说，排序和合并都不是在运行时间完成的，而是提前完成的，也就是在字段值表和记录重构表建立的过程中。

⊜ 在本附录的后面部分我们都是用 Tutorial D，而不是 SQL。

S

| S# | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

SPJ

| S# | P# | J# | QTY |
|---|---|---|---|
| S1 | P1 | J1 | 200 |
| S1 | P3 | J2 | 100 |
| S2 | P1 | J1 | 200 |
| S2 | P1 | J2 | 500 |
| S2 | P2 | J2 | 500 |
| S3 | P1 | J1 | 100 |
| S3 | P2 | J2 | 500 |
| S3 | P3 | J1 | 200 |
| S3 | P3 | J2 | 200 |

图 A-19　关系变量 S 和 SPJ（实例值）

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | S# | SNAME | STATUS | CITY |
| 1 | S1 [1:2] | Adams [1:1] | 10 [1:1] | Athens [1:1] |
| 2 | S2 [3:5] | Blake [2:2] | 20 [2:3] | London [2:3] |
| 3 | S3 [6:9] | Clark [3:3] | 30 [4:5] | Paris [4:5] |
| 4 | S4 [ : ] | Jones [4:4] | | |
| 5 | S5 [ : ] | Smith [5:5] | | |

| | 5 | 6 | 7 |
|---|---|---|---|
| | P# | J# | QTY |
| | P1 [1:4] | J1 [1:4] | 100 [1:2] |
| | P2 [5:6] | J2 [5:9] | 200 [3:6] |
| | P3 [7:9] | | 500 [7:9] |
| | | | |

图 A-20　供应商表和货运表的合并字段值

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | S# | SNAME | STATUS | CITY |
| 1 | 5 | 5 | 4 | 5 |
| 2 | 4 | 4 | 2 | 1 |
| 3 | 2 | 3 | 3 | 4 |
| 4 | 3 | 1 | 5 | 2 |
| 5 | 1 | 2 | 1 | 3 |

| | 1 | 5 | 6 | 7 |
|---|---|---|---|---|
| | S# | P# | J# | QTY |
| 1 | 2 | 1 | 2 | 2 |
| 2 | 8 | 2 | 3 | 6 |
| 3 | 3 | 3 | 4 | 1 |
| 4 | 4 | 7 | 5 | 3 |
| 5 | 5 | 8 | 1 | 8 |
| 6 | 1 | 9 | 6 | 9 |
| 7 | 6 | 4 | 7 | 4 |
| 8 | 7 | 5 | 8 | 5 |
| 9 | 9 | 6 | 9 | 7 |

图 A-21　供应商表和货运表的记录重构表

根据此 Z 字环遍历货运的记录重构表，并从相应的字段值表中取得字段值，我们可以得到如下的结果：

| S# | P# | J# | QTY |
|---|---|---|---|
| S1 | P1 | J1 | 200 |
| S2 | P1 | J1 | 200 |
| S3 | P3 | J1 | 200 |
| S3 | P3 | J2 | 200 |

第二个例子，考虑带“ < ”约束的查询：

```
SPJ WHERE QTY < 150
```

这样的查询也很容易处理，步骤如下：

a. 在字段值表的 QTY 属性列上做顺序查找。

b. 对找到的每一个单元重构所有相应的记录以及用户级上的元组。

c. 当发现字段值表的 QTY 属性列中的值等于或大于 150 时结束操作过程。

结果如下：

| S# | P# | J# | QTY |
|---|---|---|---|
| S1 | P3 | J2 | 100 |
| S3 | P1 | J1 | 100 |

现在考虑下面的这样一个查询：

```
SPJ WHERE S# = S# ('S3') AND QTY = 100
```

通过在字段值表的 S#和 QTY 属性列上的搜索，我们从找到的行区间中发现有四个货运记录的供应商编号为 S3，但只有两个的 QTY 值是 100。因此最好的方法就是使用与 QTY = 100 相关的 zig-zag，然后在记录重构的过程中检查其供应商编号是否是 S3，如果不是就停止重构过程。结果如下：

| S# | P# | J# | QTY |
|---|---|---|---|
| S3 | P1 | J1 | 100 |

最后，我们来考虑将 AND 用 OR 替换后的变化：

```
SPJ WHERE S# = S# ('S3') OR QTY = 100
```

我们可以先找出所有供应商为 S3 的元组，然后找出所有 QTY 值为 100 且在上一步没有出现的元组（或者是将上面两步的次序颠倒）。如果是按这种方式，并且两次查找的结果都按相同的属性排序（比如说都是按 S#的升序排列），那么它们合并后即可产生如下整个结果：

| S# | P# | J# | QTY |
|---|---|---|---|
| S1 | P3 | J2 | 100 |
| S3 | P1 | J1 | 100 |
| S3 | P2 | J2 | 500 |
| S3 | P3 | J1 | 200 |
| S3 | P3 | J2 | 200 |

**2. 投影**

要计算投影 SPJ {S#, P#, J#}，我们只需为货运执行通常的重构过程，但在每个记录中要跳过属性 QTY 的重构步骤。如果要计算投影 SPJ {S#, P#} ——与前面的这个例子稍有不同的就是要删除重复项——它要求为货运执行重构过程后所产生出的元组属性次序是从 major 到 minor，即 S# - P#（或者是 P# - S#）；在这种次序下重复的元组将处在邻近的位置，有利于删除重复项。这里我们略去了进一步的细节，但要指出的就是“最优”记录重构表之所最优是因为它们支持这种排序。

**3. 聚集**

考虑下面的查询：

```
SUMMARIZE SPJ PER S { S# } ADD COUNT AS SHIP_COUNT
```

下面是查询结果：

| S# | SHIP_COUNT |
|---|---|
| S1 | 2 |
| S2 | 3 |
| S3 | 4 |
| S4 | 0 |
| S5 | 0 |

图 A-20 中的字段值表的 S#属性列如下所示：

很明显，查询结果可以直接由该列的行区间求得。

下面是另外一个例子（注意我们这里使用了聚集的 BY 变量）：

| | S# |
|---|---|
| 1 | S1 [1:2] |
| 2 | S2 [3:5] |
| 3 | S3 [6:9] |
| 4 | S4 [ : ] |
| 5 | S5 [ : ] |

```
SUMMARIZE SPJ BY { S# } ADD MIN ( QTY ) AS MNQ
```

我们以供应商 S2 为例考察这个查询的实现方式，图 A-20 的字段值表中 S2 的行区间为 [3:5]，即货运的记录重构表中对应 S2 的行是第 3，4，5 行——即满足要求的单元是 [3，1]，[4，1]，[5，1]。根据表中 QTY 单元的 Z 字环，我们可以找到字段值表中的对应行——分别是第 2，3 行和第 3 行。因为 QTY 属性列在字段值表中是升序排列的，很明显，S2 的 QTY 属性的最小值就是字段值表第 2 行的值，即 200。

**4. 连接**

在前面的小节中，我们曾讨论过连接操作的实现方式。这里我们再给出一些讨论：

- 因为 TR 技术高效的实现了排序/归并连接，并且排序和归并操作都是在系统装载的过程中完成，运行时的连接代价是线性的。参考文献 [A.1] 给出了一个包含五个表连接的例子，TR 技术只需要 5 秒的操作时间，而一般的实现方式（见第 18 章）将需要 3 万亿年，或者说是宇宙年龄的 200 倍。
- 需要连接的表越多，TR 技术的优势就越明显。
- 因为所有的连接都是以相同的方式实现的，因此不需要像直接影像系统那样做复杂的访问路径选择。
- 直接影像系统中的访问路径选择往往是准确度不够，因为中间结果大小的估计是很困难的。

**5. 并、交、差**

作为本节示例的基础，我们扩展前面的示例数据库，使其包含零件的关系变量 P 以及它的一些示例值。同样我们还要扩展合并字段值表的 CITY 属性列，使其包含零件的 city 名以及相应的行区间，如下所示：

| | CITY |
|---|---|
| 1 | Athens [1:1] [ : ] |
| 2 | London [2:3] [1:3] |
| 3 | Oslo [ : ] [4:4] |
| 4 | Paris [4:5] [5:6] |

现在考虑

```
X { CITY } op Y { CITY }
```

形式的操作，其中 *X* 为 *S* 或是 *P*，op 表示 UNION，INTERSECT 或是 MINUS。那么使用字段值表中合并 CITY 属性列实现这些操作的方法是很明显的。讨论如下：

- 并：对于结果集中的每个 city 名元组，它要么在供应商表中有非空的行区间，要么在零件表中有非空的行区间，或者两者都有。也就是说，并集合是合并列中的所有 city 名。
- 交：对于结果集中的每个 city 名元组，它必须在供应商表和零件表中都有非空的行区间。
- 差：如果 *X* 为 *S*，那么对于结果集中的每个 city 名元组，它在供应商表中的行区间必须非空，而在零件表中的行区间为空。相似的，如果 *X* 为 *P*，那么结果集中的元组在零件表中的行区间非空，在供应商表中的行区间为空。

所有这些操作都可以通过在合并字段值表的 CITY 属性列做一次遍历来完成。

**6. 结论**

使用 TR 技术实现关系操作还有很多值得讨论的东西，但本节中所讨论的例子已经足够描绘出主要的思想。下面我们再给出 TR 的其他一些优点：

- 直接影像的实现方式有时需要物化一些中间结果，TR 同样需要做这样的工作，但 TR 中的物化；（a）使用的概率不大；（b）效率更高（主要是因为预排序）。
- 候选码（值唯一）和外码约束在 TR 技术中的实现非常高效，这要归功于字段值表是经过压缩和合并的。
- 下面是 Codd 在他的第一篇关系模型论文中的一段话：
  一旦意识到某个关系的存在，用户就会期望能够组合这个表“已知”的属性和某些“未知”的属性。这是一种我们称为关系对称开发的系统特征（目前许多的信息系统都不具备这一特征），但一般得不到对称的系统性能。

TR 技术同样给我们带来了性能上的对称性！——或者说，它至少在这一点上要比直接影像系统做得好——由于字段值和连接信息的分离，使得数据可以同时高效的以多种排列顺序存放。

## A.7 小结

我们简要的讨论了 **TR 模型**，一种实现关系 DBMSs 的全新的方式。TR 模型代表了一种更为通用的叫做 **Tarin 转换法**的特定应用，Tarin 转换法是用于实现多种数据存储和检索系统的技术，它是美国专利局的一项专利，归属于一个叫做 Required Technologies 的公司（*http*：*//www. requiredtech. com*）。

我们已经讨论了在只读，主存数据库中的 TR 模型，尽管我们的讨论并不深入，但已经完全可以看出它与传统的直接影像技术截然不同。现在，也许你还有许多疑问，比如说：

- 当面临对数据库任意的更新时，字段值表和记录重构表能得到有效维护吗？
- 因为记录重构表和原文件是同构的——实际上，是与原关系同构——但在每个单元中存放的是指针而不是数据值，TR 技术是否比传统的直接影像系统需要更少的存储空间？
- 在基于外存的系统中，Z 字环是不是就意味着大量的随机访问，从而导致低劣的性能呢？在外存中的二分查找能否高效实现？

等等诸如此类的问题。这些问题都是需要考虑的，但这里无法一一讨论；但这些问题都已经得到很好的解决并且已经有了实现。进一步的讨论请参考文献［A.1］和［A.2］。

## 参考文献

［A.1］ *C. J.* Date: *Go Faster! The TransRelational™ Approach to DBMS Implementation*. To appear.
附录 A 的内容主要就是基于这本书，书中讨论了本附录内容的很多细节以及其他的主题。比如说更新，外存数据库等，它给出了 A.7 节提出的问题的答案。

［A.2］ U. S. Patent and Trademark Office: *Value - Instance - Connectivity Computer - Implemented Database*. U. S. Patent No. 6,009,432 (December 28, 1999).
这是 Tarin 转换法专利，它隶属于 Required Technologies 公司（*http://www. requiredtech. com*）。注意：Tarin 转换法目前已经得到扩展，超出了在专利中对它的描述。比如说，它包含了各种更新技术以及处理外存数据库的方法（见参考文献［A.1］）。注意本参考文献的标题：在专利中它们给出了名称 value stores，instance stores 和 connectivity stores。与之相对应的，本附录和参考文献［A.1］中使用术语字段值表和记录重构表，后者在某些方面更面向用户而且更好的反应了本质。

# 附录 B　SQL 表达式

## B.1　引言

表和布尔表达式是 SQL 语句的核心。在本附录中我们给出了这些表达式的 BNF 语法，同时给出了某些情况下这些表达式的语义。但我们省去了下面的这些内容：

- 标量表达式的细节
- WITH 的递归形式的细节
- 非标量表达式 < *select item* > s
- < *table ref* > 和 < *type spec* > 的 ONLY 变量
- GROUP BY 的 GROUPING SETS，ROLLUP 和 CUBE 选项
- 条件 BETWEEN，OVERLAPS 和 SIMILAR
- 所有与空值相关的内容

然而，需要说明一点：因为我们认为标准［4.23］中的术语不太恰当，所以在依据语法进行的分类中和在 SQL 语言的构造中这里所使用的名称跟［4.23］的标准中使用的名称是不同的；实际上，本书中所使用的表表达式、条件表达式和标量表达式就不是标准术语。特别地，我们分别将 < *table value constructor* >、< *row value constructor* > 缩写为 < *table constructor* >、< *row constructor* >。

## B.2　表的表达式

下面是一个 < *table expression* >（表的表达式）的 BNF 文法。

```
<table exp>
   ::=   <with exp> | <nonwith exp>

<with exp>
   ::=   WITH [ RECURSIVE ]
            <table name> [ ( <column name commalist> ) ] ]
            AS ( <table exp> )
            <nonwith exp>

<nonwith exp>
   ::=   <join table exp> | <nonjoin table exp>

<join table exp>
   ::=   <table ref> [ NATURAL ] JOIN <table ref>
                     [ ON <bool exp>
                     | USING ( <column name commalist> ) ]
       | <table ref> CROSS JOIN <table ref>
       | ( <join table exp> )

<table ref>
   ::=   <table name> [ [ AS ] <range var name>
                      [ ( <column name commalist> ) ] ]
       | ( <nonwith exp> ) [ AS ] <range var name>
                           [ ( <column name commalist> ) ]
       | <join table exp>

<nonjoin table exp>
   ::=   <nonjoin table term>
       | <nonwith exp> UNION [ ALL | DISTINCT ]
         [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
           <table term>
       | <nonwith exp> EXCEPT [ ALL | DISTINCT ]
         [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
           <table term>

<nonjoin table term>
```

```
    ::=   <nonjoin table primary>
        | <table term> INTERSECT [ ALL | DISTINCT ]
          [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
            <table primary>

<table term>
    ::=   <nonjoin table term> | <join table exp>

<table primary>
    ::=   <nonjoin table primary> | <join table exp>

<nonjoin table primary>
    ::=   TABLE <table name>
        | <table constructor>
        | <select exp>
        | ( <nonjoin table exp> )

<table constructor>
    ::=   VALUES <row constructor commalist>

<row constructor>
    ::=   <scalar exp>
        | ( <scalar exp commalist> )
        | ( <table exp> )

<select exp>
    ::=   SELECT [ ALL | DISTINCT ] <select item commalist>
            FROM <table ref commalist>
              [ WHERE <bool exp> ]
                [ GROUP BY <column name commalist> ]
                  [ HAVING <bool exp> ]

<select item>
    ::=   <scalar exp> [ [ AS ] <column name> ]
        | [ <range var name>. ] *
```

下面对 < *select expression* > 进行详细阐述，因为在实际使用中，该表达式是最重要的。一个 < *select expression* > 可以不很严格地看作是一个没有 JOIN、UNION、EXCEPT 和 INTERSECT 操作的 < *table expression* > 。之所以说不很严格，是因为这些运算符可以出现在某些嵌套在 < *select expression* > 的表达式中。正如前面的文法中所写的，一个 < *select expression* > 按顺序包括：SELECT 子句，FROM 子句、可选的 WHERE 子句、GROUP BY 子句和 HAVING 子句。下面依次来介绍这些子句。

### 1. SELECT 子句

下面是 SELECT 子句的形式：

```
SELECT [ ALL | DISTINCT ] <select item commalist>
```

说明：

1）< *select item commalist* > 不能为空[⊖]（ < *select item* > 的详细介绍可参见下面）。

2）如果没有指定是 ALL 还是 DISTINCT，默认是 ALL[⊖] 。

3）此时假定已经执行完 FROM、WHERE、GROUP BY 和 HAVING 子句。无论这些子句是给定的还是忽略的，执行完这些子句后，概念上来说得到一个表，这个表可能是一个“组”表（可参见后面的介绍），称这个表为 *T*1，该表将会在后面用到。注：这个概念结果实际上是没有命名的。

4）假定 *T*2 是通过在 *T*1 上执行指定的 < *select item* > 而从 *T*1 中得到的。

5）假定 *T*3 是通过指定 DISTINCT 从 *T*2 或者是跟 *T*2 同样的表中消除冗余行后得到的。

6）*T*3 是最后的结果。

现在对 < *select item* > 做一下解释。对 < *select item* > ，需要考虑两种情况，但是因为第二种情况是第一种情况的 < *select item* > 的逗号列表的简写，因此，第一种情况更加基本。

---

⊖ 实际上，本附录中提到所有 lists 和 commalists 都必须非空。

⊖ 也就是说，SELECT 默认的是 ALL。而 UNION，INTERSECT 或是 EXCEPT 默认的是 DISTINCT。

第一种情况：< *select item* > 的形式是：

```
<scalar exp> [ [ AS ] <column name> ]
```

< *scalar expression* > 可以包含 *T1* 表的一个或多个列（见说明的第 3 点），当然这种包含不是必需的。对 *T1* 中的每一行，执行 < *scalar expression* > 将产生一标量结果。在 T1 表每一行上执行 SELECT 子句中的所有 < *select item* >，将得到一个结果的逗号列表。这个逗号列表就是 *T2* 表（见上面说明的第 4 点）的一行。如果 < *select item* > 包括一个 AS 子句，那么，该子句中的没有限定的 < *column name* > 将赋值到 *T2* 对应的列上，当然，可以忽略可选的 AS 关键字，这不会影响执行结果。如果 < *select item* > 子句没有包含 AS 子句，就有两种情况需要考虑：（a）如果它只包含了简单的可能有限定的 < *column name* >，< *column name* > 就作为 *T2* 中对应的列名；（b）否则，*T2* 中对应的列就没有有效的列名（实际执行中，将对 *T2* 中的列赋值一个执行依赖的列名）。下面再做几点说明：

- 特别地，因为在 AS 子句中引入的列名是 *T2* 的列名，而不是 *T1* 的列名，所以在构造 *T1* 的 WHERE、GROUP BY 和 HAVING 子句时，就不能直接包含该列名。然而，它在任何情况下（特别地，在 DECLARE CURSOR 中）可以在一个相关连的 ORDER BY 子句中参照，也可以在一个包含 < *select expression* > 的 < *table expression* > 中参照使用。
- 如果 < *select item* > 包括一个聚集操作符，而且 < *select expression* > 不包含 GROUP BY 子句，那么：若 *T1* 的列没有作为聚集操作符的参数或者部分参数使用，则 SELECT 子句中的 < *select item* > 就不能参照 *T1* 中的任何列。

第二种情况：< *select item* > 是如下的形式：

```
[ <range var name> . ] *
```

如果忽略了限定词，例如，< *select item* > 只是没有限定的星号，那么这个 < *select item* > 就是 *SELECT* 子句中的唯一的 < *select item* > 了。这是一种按照从左往右的顺序列出 *T1* 中的所有的列名的简写方式。如果包含了限定词，如，< *select item* > 包括了一个由范围变量名所限定的星号：*R.* *，那么 < *select item* > 就会按照从左往右的顺序列出跟范围变量 R 相联系的表的所有列的 < *column name* > 的逗号列表。（8.6 节中曾经介绍过，一个表的名字经常会作为一个隐含的范围变量。这样，< *select item* > 就经常是 *T.* * 的形式，而不是 *R.* * 的形式。）

### 2. FROM 子句

FROM 子句的形式如下：

```
FROM <table ref commalist>
```

指定 < table ref > 分别对应表 *A*,*B*,...,*Z*。那么，FROM 子句的执行结果就是一个等同于 *A*,*B*,...,*Z* 做笛卡尔积（SQL - 风格）后得到的表。

### 3. WHERE 子句

WHERE 子句的形式如下：

```
WHERE <bool exp>
```

假定增加了前面的 FROM 子句后执行得到的结果是 *T* 表。而 WHERE 子句的结果则是将 *T* 表中不符合 < conditional expression > 的行去掉后得到的。如果忽略掉 WHERE 子句，结果就仍是 *T*。

### 4. GROUP BY 子句

GROUP BY 子句的格式如下：

```
GROUP BY <column name commalist>
```

假定执行了前面 FROM 子句和 WHERE 子句之后，执行结果是 *T* 表。每一个在 GROUP BY 子句中的 < column name > 必须是 T 表的一个可选的有限定的列名。GROUP BY 子句的结果是一个组表，——也就是，组的集合（每个组中又包括若干行）。通过概念上重新排列 *T* 表中的行，

使组数减小到最少，就得到了组表中的行，组表中的每一个组中的所有记录行在 GROUP BY 子句所指定的列组合上具有相同的值。因此要注意，这个结果不是一个“真正的表”（即不是一个记录行的表，而是由组组成的一个表）。然而，如果没有相应的 SELECT 子句，GROUP BY 子句也不会出现，SELECT 语句就是从组表中得到真正的表（记录行的表）。因此，这种对关系框架的临时偏离对最后的结果没有很大的影响。

如果 < *select exp* > 包括一个 GROUP BY 子句，那么相应的 SELECT 子句的形式就会有限制。特别地，SELECT 子句（包括星号简写的形式）中的每一个 < *select item* > 在每个组中只有一个值。

**5. HAVING 子句**

HAVING 子句的格式如下：

```
HAVING <bool exp>
```

假定在增加了前面的 FROM 子句、WHERE 子句和 GROUP BY 子句之后执行得到的结果是 *G* 表。如果没有 GROUP BY 子句，那么 *G* 就是在只执行 FROM 和 WHERE 子句的情况下得到的表，是一个只有一个组的表[○]。换句话说，在这种情况下，有一个隐含的、概念上的 GROUP BY 子句，该子句没有指定进行分组所需要参照的列。HAVING 子句的结果就是将 *G* 表中不满足条件 < bool expression > 的组去掉后所得到的表。下面再做几点说明：

- 如果省略了 HAVING 子句，但是仍然包括有 GROUP BY 子句，执行的结果就是 *G*。如果 HAVING 子句和 GROUP BY 子句都省略了，结果就是只执行 FROM 和 WHERE 子句后得到的 *T* 表，是没有分组的。
- HAVING 子句中的任何 < *scalar expression* > 在每组中必须只能有一个值（这跟前一部分 SELECT 子句中有 GROUP BY 子句时 < *scalar expression* > 的情况类似）。

**6. 综合示例**

在讲解 < *select exp* > 的最后部分，给出一个复杂度适当的例子，通过该例可以更进一步阐明上面所讲的一些（但不是全部）内容。进行下面的查询：

对于那些红色和蓝色的总供给量超过 350 的零件（并且零件总数小于等于 200 的除外），列出其零件号、重量（以克计）、颜色和该零件的最大供给量。

下面是一种可行的查询：

```
SELECT P.P#,
       'Weight in grams =' AS TEXT1,
       P.WEIGHT * 454 AS GMWT,
       P.COLOR,
       'Max quantity =' AS TEXT2,
       MAX ( SP.QTY ) AS MXQTY
FROM   P, SP
WHERE  P.P# = SP.P#
AND  ( P.COLOR = COLOR ('Red') OR P.COLOR = COLOR ('Blue') )
AND    SP.QTY > QTY ( 200 )
GROUP  BY P.P#, P.WEIGHT, P.COLOR
HAVING SUM ( SP.QTY ) > QTY ( 350 ) ;
```

说明：需要注意 < *select expression* > 子句概念上的执行是按照其书写顺序一一进行的。当然，SELECT 语句除外，因为它要在最后执行。因此，可以想象，在该例中，结果集是这样构造的：

1）FROM：执行 FROM 子句以生成一个新表，该表是表 *P* 和表 *SP* 的笛卡尔积。

2）WHERE：将第 1 步中不符合 WHERE 子句条件的记录行去掉。该例中，是去掉不满足下面条件的记录行：

---

○ 这是标准上的说法，但从逻辑上讲它最多只有一个组（如果在 FROM 和 WHERE 子句中的条件为空的话，将不存在任何组）。

```
        P.P# = SP.P#
AND   ( P.COLOR = COLOR ('Red') OR P.COLOR = COLOR ('Blue') )
AND     SP.QTY > QTY ( 200 )
```

3) GROUP BY：将第 2 步的结果按照 GROUP BY 子句中的列名分组。该例中，需要根据其分组的列名是：P. P#、P. WEIGHT 和 P. COLOR。

4) HAVING：那些不满足条件表达式的组将要从步骤 3 的结果中删除掉。

```
SUM ( SP.QTY ) > QTY ( 350 )
```

5) SELECT：第 4 步结果中的每个组产生一个只有一个结果的记录行。首先，从组中取得零件号、重量、最大需求量。第二，将重量转化为克来表示。第三，将字符串“Weight in grams =”和“Max quatity =”插入到记录行中适当的位置。注意，之所以说“适当的位置”，是因为在 SQL 中，表中的列有从左到右的顺序，如果这两个字符串没有出现在“恰当的位置”，那么它们的意义也就不大了。

最后的结果是如下的形式：

| P# | TEXT1 | GMWT | COLOR | TEXT2 | MXQTY |
|---|---|---|---|---|---|
| P1 | Weight in grams = | 5448 | Red | Max quantity = | 300 |
| P5 | Weight in grams = | 5448 | Blue | Max quantity = | 400 |
| P3 | Weight in grams = | 7718 | Blue | Max quantity = | 400 |

刚才描述的算法仅对 < *select exp* > 的执行进行了概念上的解释。从保证生成正确结果的方面来看，算法是正确的。然而，实际执行时却有可能不是很有效。例如，要使系统在第 1 步中非常正确地生成笛卡尔积，就是非常不可能的事情。这和第 18 章的讨论都说明了为什么在关系系统中需要优化器。确实，SQL 系统中的优化器的作用就在于：可以用比概念算法更少的时间来得到相同的结果。

## B.3 布尔表达式

跟上节类似，首先给出一个 BNF 文法。接下来详细介绍以下几个问题：< *like condition* >、< *match condition* >、< *all or any condition* >。

```
<bool exp>
   ::=   <bool term> | <bool exp> OR <bool term>

<bool term>
   ::=   <bool factor> | <bool term> AND <bool factor>

<bool factor>
   ::=   [ NOT ] <bool primary>

<bool primary>
   ::=   <simple cond> | ( <bool exp> )
<simple cond>
   ::=   <comp cond> | <in cond> | <like cond> | <match cond>
       | <all or any cond> | <exists cond> | <unique cond>
       | <distinct cond> | <type cond>

<comp cond>
   ::=   <row constructor> <comp op> <row constructor>

<comp op>
   ::=   = | < | <= | > | >= | <>

<in cond>
   ::=   <row constructor> [ NOT ] IN ( <table exp> )
       | <scalar exp> [ NOT ] IN ( <scalar exp commalist> )

<like cond>
   ::=   <char string exp> [ NOT ] LIKE <pattern>
                           [ ESCAPE <escape> ]

<match cond>
   ::=   <row constructor> MATCH UNIQUE ( <table exp> )
```

```
<all or any cond>
   ::=   <row constructor> <comp op> ALL ( <table exp> )
       | <row constructor> <comp op> ANY ( <table exp> )

<exists cond>
   ::=   EXISTS ( <table exp> )

<unique cond>
   ::=   UNIQUE ( <table exp> )

<distinct cond>
   ::=   <row constructor> IS DISTINCT FROM <row constructor>

<type cond>
   ::=   TYPE ( <scalar exp> )
              IS [ NOT ] OF ( <type spec commalist> )

<type spec>
   ::=   <type name>
```

### 1. Like 条件

Like 条件一般用于简单的字符串的模式匹配。例如：检查一个字符串，看它是否与某一预定义的模式相匹配。其语法是：

```
<char string exp> [ NOT ] LIKE <pattern> [ ESCAPE <escape> ]
```

这里，< *pattern* > 是任意的字符串表达式，< *escape* > 是一个只有单个字符的字符串表达式。这是一个例子：

```
SELECT P.P#, P.PNAME
FROM   P
WHERE  P.PNAME LIKE 'C%';
```

（列出那些零件名以 C 开头的零件的零件号和零件名）。结果为：

| P# | PNAME |
|---|---|
| P5 | Cam |
| P6 | Cog |

只要没有指定 ESCAPE 子句，< *pattern* > 中的字符串按照如下的方式解释：

- 下划线“_”表示任意单个字符。
- 百分号“%”代表任何有序的 *n* 个字符（*n* 可以为 0）。
- 所有其他的字符代表其本身。

因此，在该例中，查询将返回表 P 中 PNAME 以 C 开头，之后可以有 0 到多个字符的记录行。下面再举一个例子：

```
ADDRESS LIKE '%Berkeley%'      ——如果在 ADDRESS 中的任意位置包含
                                  “Berkeley”字串，则返回为 TRUE
S# LIKE 'S__'                  ——如果 S# 中包含 3 个字符，并且第一
                                  字符为“S”，则返回为 TRUE
PNAME LIKE '%c___'             ——如果 PNAME 中包含 4 个或 4 个以上
                                  字符，并且最后三个字符的首字母
                                  为“c”，则返回为 TRUE
MYTEXT LIKE '=_%'              ——如果 MYTEXT 以下划线开始则返回为
       ESCAPE '='                 TRUE
```

在最后的例子中，字符“=”指定作为转义字符，转义字符的作用就是使特殊字符“_”和“%”不再具有特殊的意义，如果想使用这些特殊字符，就使用转义字符“=”。

最后，< *like condition* >

```
x NOT LIKE y [ ESCAPE z ]
```

在语义上跟下面的语句是相同的：

```
NOT ( x LIKE y [ ESCAPE z ] )
```

### 2. MATCH 条件

< *match condition* > 的格式是：

```
<row constructor> MATCH UNIQUE ( <table exp> )
```

假定 *r1* 是执行 < *row constructor* > 后得到的一行，*T* 是执行 < *table expression* > 后得到的一个表。如果 T 有且只有一行，假定为 *r2*，那么该 < *match condition* > 执行结果为真，下面的比较将返回真。

```
r1 = r2
```

下面是一个例子：

```
SELECT SP.*
FROM   SP
WHERE  NOT ( SP.S# MATCH UNIQUE ( SELECT S.S# FROM S ) ) ;
```

（如果 *SP* 表的供应商不只对应供应商表中一个供应商，则列出其发货量）。当然，如果数据库正确，该查询就不会有什么结果，所以这样的一个查询对于检查数据库的完整性来说是有用的。需要注意，< *in condition* > 语句也有可能达到这样的目的。

顺便说一下，UNIQUE 可以从 MATCH UNIQUE 中省略，这样 MATCH 就跟 IN 非常类似，至少在没有空值的情况下是这样的。

### 3. All 或 Any 条件

< *all or any condition* > 的一般形式是：

```
<row constructor> <comp op> <qualifier> ( <table exp> )
```

其中，< *comparison operator* > 可以是任何的通常的运算符（=、< >，等），< *qualifier* > 是 ALL 或 ANY[⊖]．一般来说，当且仅当对 < *table expression* > 给出的表中的所有行来说，没有 ALL（即为 ANY）的比较都为 true 时，< *all or any condition* > 才为 true。（如果表为空，ALL 条件将为 TRUE，但是 ANY 条件将为 FALSE。）下面给出一个例子，“取得那些重量超过每一个蓝色零件的零件名”：

```
SELECT DISTINCT PX.PNAME
FROM   P AS PX
WHERE  PX.WEIGHT >ALL ( SELECT PY.WEIGHT
                        FROM   P AS PY
                        WHERE  PY.COLOR = 'Blue' ) ;
```

使用一般的示例数据，结果如下：

| PNAME |
| --- |
| Cog |

*说明*：嵌套的 < *table expression* > 返回蓝色零件的重量的集合。外层的 SELECT 返回那些重量超过上面的集合的零件名。当然，一般来说，结果的数目是不确定的（也有可能为 0）。

*注意*：需要注意一个单词的使用，母语为英语的读者更应该注意。< *all or any condition* > 的使用经常会出错。自然的英语会使用 any 来表示 every 的查询，这样，大家在使用时就很容易使用（不正确）“ > ANY” 而不是 “ > ALL”。所有（还是任意?）ANY 和 ALL 的操作都有类似的问题。

---

⊖ ANY 也可以写为 SOME。

# 附录 C　缩略语和符号

INF（first normal form）　第一范式
2NF（second normal form）　第二范式
2PC（two - phase commit）　两阶段提交
2PL（two - phase locking）　两阶段封锁
2VL（two - valued logic）　二值逻辑
2øC 与 2PC 相同
2øL 与 2PL 相同
3GL（third - generation language）　第三代语言
3NF（third normal form）　第三范式
3VL（three - valued logic）　三值逻辑
4GL（fourth generation language）　第四代语言
4NF（fourth normal form）　第四范式
4VL（four - valued logic）　四值逻辑
5NF（fifth normal form）　第五范式（与 PJ/NF 一样）
6NF（sixth normal form）　第六范式

A（ALGEBRA）　代数
ACID（atomicity - consistency - isolation - durability）　原子性 - 一致性 - 隔离性 - 持久性
ACM（Association for Computing Machinery）　美国计算机学会
ADT（abstract data type）　抽象数据类型
AES（Advanced Encryption System）　高级加密系统
ALGEBRA（A Logical Genesis Explains Basic Relational Algebra）　基本关系代数的一种逻辑起源解释
ANSI（American National Standards Institute）　美国国家标准协会
ANSI/SPARC（ANSI/Standards Planning and Requirements Committee）　ANSI/标准规划与需求委员会，用于指第 2 章中提到的数据库系统三级体系结构
API（application programming interface）　应用编程接口
ARIES（Algorithms for recovery and isolation Exploiting Semantics）　恢复和隔离语义的算法
AST（automatic summary table）　自动摘要表
BB　同 GB
BCNF（Boyce/Codd normal form）　BC 范式
BLOB（binary large object）　二进制大对象
BNF（Backus - Naur form or Backus normal form）　巴科斯范式
CACM（Communications of the ACM（ACM publication））　ACM 通信（ACM 出版物）
CAD/CAM（computer - aided de sign/computer - aided manufacturing）　计算机辅助设计/计算机辅助制造
CASE（computer - aided software engineering）　计算机辅助软件工程
CDO（class - defining object）　类定义对象
CIM（computer - integrated manufacturing）　计算机集成制造
CLI（Call - Level Interface）　调用级接口

CLOB（character large object） 字符大对象
CNF（conjunctive normal form） 合取范式
CODASYL（Conference on Data Systems Languages） 数据系统语言委员会；用于指网状数据库系统，如 IDMS
CPU（central processing unit） 中央处理器
CS（cursor stability（DB2）） 游标稳定性（DB2）
CWA（Closed World Assumption） 封闭世界假设
DA（data administrator） 数据管理员
DB/DC（database/data communications） 数据库/数据通信
DBA（database administrator） 数据库管理员
DBMS（database management system） 数据库管理系统
DBP&D（Database Programming & Design 数据库编程与设计（是一本杂志，后改为《Intelligent Enterprise》杂志，并提供在线文档）
DBTG（Data Base Task Group） 数据库任务组，与 CODASYL 同义（在数据库环境中）
DC（data communications） 数据通信
DCO（domain check override） 域检查过载
DDB（distributed database） 分布式数据库
DDBMS（distributed DBMS） 分布式数据库管理系统
DDL（data definition language） 数据定义语言
DES（Data Encryption Standard） 数据加密标准
DK/NF（domain - key normal form） 域码范式
DML（data manipulation language） 数据操作语言
DNF（disjunctive normal form） 析取范式
DOM（Document Object Model（XML）） 文件对象模型（XML）
DRDA（Distributed Relational Database Architecture（IBM）） 分布式关系型数据库体系结构（IBM）
DSL（data sublanguage） 数据子语言
DSS（decision support system） 决策支持系统
DTD（Document Type Definition（XML）） 文件类型定义（XML）
DUW（distributed unit of work） 分布式工作单元
E/R（entity/relationship） 实体/关系
EB（same as XB） 与 XB 一致
ECA（event - condition - action） 事件 - 条件 - 行为
EDB（extensional database） 外延数据库
EDI（Electronic Data Interchange） 电子数据交换
EKNF（elementary key normal form） 基本码范式
EMVD（embedded MVD） 嵌入式 MVD
EOT（end of transaction） 事务结束
FD（functional dependence） 函数依赖
FLWOR（for-let-where-order by-return（XML）） XML 语句
FTP（File Transfer Protocol） 文件传输协议（一般也用小写 ftp）
GB（gigabyte（1024MB）） 吉字节（1024 兆字节）
GIS（geographic information system） 地理信息系统
GML（Generalized Markup Language） 通用标记语言
HOLAP（hybrid OLAP） 混合 OLAP
HTML（HyperText Markup Language） 超文本标记语言

HTTP（Hypertext Transfer Protocol） 超文本传输协议（一般也用小写 http）
I/O（input/output） 输入/输出
IDB（intensional database） 内涵数据库
IDMS（Integrated Database Management System） 集成的数据库管理系统
IFIP（International Federation for Information Processing） 国际信息处理联盟
IEEE（Institute for Electrical and Electronics Engineers） 电气与电子工程师协会
IMS（Information Management System） 信息管理系统
INCITS（ANSI International Committee on Information Technology Standards） ANSI 国际信息技术标准委员会（以前称为 NCITS，再之前称为 X3）
INCITS/H2（INCITS database committee） INCITS 数据库委员会
IND（inclusion dependence） 内含依赖
IS（intent shared（lock）；information systems） 意向共享锁；信息系统
ISBL（Information System Base Language（PRTV）） 信息系统（库）语言（PRTV）
ISO（International Organization for Standardization） 国际标准化组织
IT（information technology） 信息技术
IX（intent exclusive（lock）） 意向排它锁
JACM（Journal of the ACM（ACM publication）） ACM 学报（ACM 出版物）
JD（join dependence） 连接依赖
JDBC（Java Database Connectivity） 基于 Java 的数据库互连
K（1024）千
KB（kilobyte（1024 bytes）） 千字节（1024 字节）
LAN（local area network） 局域网
LOB（large object） 大对象
LSP（Liskov Substitution Principle） Liskov 替换原则
MB（megabyte（1024KB）） 兆字节（1024 千字节）
MLS（multi - level secure） 多级安全
MOLAP（multi - dimensional OLAP） 多维 OLAP
MQT（materialized query table） 物化查询表
MVD（multi - valued dependence） 多值依赖
NCITS 参见 INCITS
NCITS/H2 参见 INCITS/H2
$NF^2$ “NF 平方” = NFNF = 非第一范式
ODBC（Open Database Connectivity） 开放数据库互连
ODMG（Object Data Management Group） 对象数据管理组
ODS（operational data store） 操作型数据存储
OID（object ID） 对象标识
OLAP（online analytic processing） 联机分析处理
OLCP（online complex processing） 联机复杂处理
OLDM（online decision management） 联机决策管理
OLTP（online transaction processing） 联机事务处理
OMG（Object Management Group） 对象管理组
OO（object - oriented；object orientation） 面向对象
OODB（object - oriented database = object database） 面向对象数据库
OODBMS（object - oriented DBMS，object DBMS） 面向对象的数据库管理系统
OOPL（object - oriented programming language，object programming language） 面向对象的程序设计语言

OQL（Object Query Language） 对象查询语言（ODMG 的一部分）
OSI（Open Systems Interconnection） 开放式系统互联
OSQL（Object SQL） 对象 SQL
PB（petabyte（1024TB）） 拍字节（1024 太字节）
PC（personal computer） 个人计算机
PJ/NF（projection - join normal form） 投影连接范式（第五范式）
PODS（Principles of Database Systems（ACM conference）） 数据库系统原理（ACM 会议）
PRTV（Peterlee Relational Test Vehicle） Peterlee 关系测试
PSM（Persistent Stored Modules 持久存储模块（SQL 标准的一部分）
PSVI（Post Schema Validation Infoset（XML）） 后架构验证信息集（XML）
QBE（Query - By - Example） 按例查询
QUEL（Query Language（Ingres）） 查询语言（Ingres）
RAID（redundant array of inexpensive disks） 冗余廉价磁盘阵列
RDA（Remote Data Access） 远程数据存取
RDB（relational database） 关系数据库
RDBMS（relational DBMS） 关系数据库系统
RID（record ID，row ID） 记录标识或行标识
ROLAP（relational OLAP） 关系 OLAP
RM/T（relational model/Tasmania） 关系模型/Tasmania
RM/V1（relational model/Version 1） 关系模型/版本 1
RM/V2（relational model/Version 2） 关系模型/版本 2
RPC（remote procedure call） 远程过程调用
RR（repeatable read（DB2）） 可重复读（DB2）
RSA（Rivest - Shamir - Adelman（encryption method）） RSA 加密算法
RUW（remote unit of work） 远程工作单元
RVA（relation - valued attribute） 关系值属性
S（shared（lock）） 共享锁
SGML（Standard GML） 标准 GML
SIGMOD（Special Interest Group on Management of Data） 数据管理特别兴趣组（ACM 特别兴趣组）
SIX（shared intent exclusive（lock）） 共享意向排它锁
SOAP（Simple Object Access Protocol） 简单对象访问协议
SPARC 参见 ANSI/SPARC
SQL（Structured Query Language，Standard Query Language） 结构化查询语言，有时也理解为标准查询语言
SQL/MM（SQL/Multimedia） SQL/多媒体
SVG（Scalable Vector Graphics） 可扩展的向量图
TB（terabyte（1024GB）） 太字节（1024 吉字节）
TCB（Trusted Computing Base） 可信计算机
TCP/IP（Transmission Control Protocol/Internet Protocol） 传输控制协议/网际协议
TID（tuple ID） 元组标识
TODS（Transactions on Database Systems（ACM publication）） 数据库系统上的事务（ACM 出版物）
TP（transaction processing） 事务处理
TPC（Transaction Processing Council） 事务处理委员会
TR TransRelational™

U（update（lock））　更新锁
UDF（user - defined function）　用户定义函数
UDO（user - defined operator）　用户定义操作
UDT（user - defined type）　用户定义类型
UML（Unified Modeling Language）　统一建模语言
unk（unknown（truth value））　未知（真值）
UNK（unknown（null））　未知（空值）
UOW（unit of work）　工作单元
URI（Uniform Resource Identifier）　统一资源标识符
URL（Uniform Resource Locator）　统一资源定位符
VLDB（very large database；Very Large Data Bases）　超大规模数据库；超大规模数据库（每年一次的学术会议）
VSAM（Virtual Storage Access Method（IBM））　虚拟存储存取方法（IBM）
W3C（World Wide Web Consortium）世界 Web 联盟
WAL（write - ahead log）　先写日志
WAN（wide area network）　广域网
WFF（well - formed formula）　合式公式
WORM（write once/read many times）　写一次读多次
WWW（World Wide Web）万维网
WYSIWYG（what you see is what you get）　所见即所得
X（exclusive（lock））　排它锁
X3　参见 NCITS
X3H2　参见 NCITS/H2
XB（exabyte（1024PB））　艾字节（1024 拍字节）
XML（Extensible Markup Language）　可扩展标记语言
XQuery（XML Query）　XML 查询
XQL（XML Query Language）　XML 查询语言
XSL（XML Stylesheet Language）　XML 样式表语言
XSLT（XML Stylesheet and Transformation Language）　XML 样式表和转换语言
YB（yottabyte（1024ZB））
ZB（zettabyte（1024XB））
$\in$　（belongs to；is a member of；is contained in；[is] in）　属于
$\ni$　（contains）包含
$\subseteq$　（is a subset of；is included in）　是……的子集
$\subset$　（is a proper subset of；is properly included in）　是……的真子集
$\supseteq$　（is a superset of；includes）　是……的超集
$\supset$　（is a proper superset of；properly includes）　是……的真超集
$\theta$　（comparison operator（=，<，etc.）；polar coordinate）　比较运算符
$\varnothing$　（the empty set）　空集
$\rightarrow$　（functionally determines）　函数决定
$\twoheadrightarrow$（multi-determines）　多值决定
$\equiv$　（is equivalent to；is identically equal to）　等价于
$\Rightarrow$　（implies（logical connective））　蕴含（逻辑上的连接）
$\vdash$　（implies（metalinguistic symbol））　蕴含（元语言符号）
$\models$　（it is always the case that（metalinguistic symbol）　可推出（元语言符号）