安徽省高等学校“十三五”规划教材

数据库系统及应用

金培权　编著

科学出版社

安徽省高等学校“十三五”规划教材

数据库系统及应用

金培权　编著

科 学 出 版 社

北　京

内 容 简 介

本书以经典的关系数据库理论和技术为基础，介绍了数据库系统概述与体系结构、关系数据模型、结构化查询语言、过程化 SQL、数据库模式设计、数据库设计、数据库应用系统开发、数据库事务、故障恢复、并发控制、数据库完整性、数据库安全性以及数据库技术新发展等内容。

在理论方面，本书重点介绍关系数据模型、关系模式规范化、并发调度的可串性，以及基于锁的并发控制机制等理论提出的背景和动机、优点和缺点，使学生能够明白当前成熟的理论对数据库领域的主要贡献，也明白已有的一些模型和方法在设计上的先进性和存在的问题，从而能够对数据库原理有更深入的认识。在应用方面，本书重点介绍存储过程、触发器等过程化 SQL 编程技术和应用的策略，并结合了 MySQL、Oracle、Microsoft SQL Server 等流行的 DBMS 详细阐述关键技术。

本书可作为高等学校计算机专业、软件工程专业、大数据专业以及其他相关专业本科生的教材，也可作为从事数据库系统与应用工作的管理人员和技术人员的参考书。

图书在版编目(CIP)数据

数据库系统及应用 / 金培权编著. — 北京：科学出版社，2023.6
安徽省高等学校“十三五”规划教材
ISBN 978-7-03-075532-2

Ⅰ. ①数… Ⅱ. ①金… Ⅲ. ①数据库系统－高等学校－教材 Ⅳ. ①TP311.13

中国国家版本馆 CIP 数据核字(2023)第 084501 号

责任编辑：于海云 张丽花 / 责任校对：王 瑞
责任印制：张 伟 / 封面设计：马晓敏

科 学 出 版 社 出版
北京东黄城根北街 16 号
邮政编码：100717
http://www.sciencep.com
北京建宏印刷有限公司 印刷
科学出版社发行 各地新华书店经销
*
2023 年 6 月第 一 版 开本：787×1092 1/16
2023 年 6 月第一次印刷 印张：19
字数：450 000

定价：69.00 元

（如有印装质量问题，我社负责调换）

前　言

本书是作者长期从事本科生和研究生“数据库”课程教学工作的经验总结，涵盖了数据库系统理论和应用的基本内容。本书的所有内容都在中国科学技术大学计算机科学与技术专业的本科生课程中讲授，部分内容曾在软件工程和计算机科学与技术双学位课程中讲授。本书总体上覆盖了经典关系数据库理论的主要内容，全面地介绍了与数据库系统相关的基本概念、理论、模型和方法。本书旨在使读者掌握四个方面的知识：①经典的关系数据库理论；②利用 SQL 回答用户查询的基本方法；③规范化数据库设计的方法和流程；④ DBMS 事务处理的相关理论和方法。学习完本书的内容，读者能基本掌握数据库系统的相关概念、原理和技术，并能够胜任数据库应用系统开发的工作。

本书内容主要涉及数据库系统概述与体系结构、关系数据模型、结构化查询语言、过程化 SQL、数据库模式设计、数据库设计、数据库应用系统开发、数据库事务、故障恢复、并发控制、数据库完整性、数据库安全性以及数据库技术新发展等方面。本书注重理论和技术的前因后果，强调相关的理论和技术提出的背景与动机，从每个知识点的设计思路、优点、不足等方面由浅入深地介绍数据库系统中的基本理论。同时，引入了大量的案例和插图帮助读者理解书中内容，便于读者更好地理解技术原理，并能够学以致用。

“教育、科技、人才是全面建设社会主义现代化国家的基础性、战略性支撑。”为了深入贯彻党的二十大报告精神，落实立德树人根本任务，围绕我国高等学校计算机类专业的教学需求和人才培养目标，对标高水平一流教材建设要求，本书从内容编排上突出系统性、整体性、实用性、前沿性等特色。

⑴ 系统性：按照“理论→应用”的顺序系统安排书中内容，既注重数据库基本概念和理论的介绍，也注重数据库应用与系统特性的讨论；既注重一般的 SQL 使用技能，也注重更高一级的过程化 SQL 和数据库应用系统设计技术；涵盖了不同专业对“数据库”课程的教学要求。

⑵ 整体性：强调了 SQL、数据库设计与软件工程的整体性。数据库设计是软件工程的一部分，而 SQL 是数据库应用系统中的核心编程技术。过去的数据库教材往往注重 SQL、数据库设计这些“点”的教学，而对于这些内容在现实世界中如何与软件工程结合的讨论很少。因此，本书在介绍 SQL、数据库设计的基础上，加入了数据库应用系统分析与设计内容，将 SQL、存储过程、触发器、数据库设计等问题通过软件工程串联为一个整体，增强学生对数据库技术的感性认识，提升学习效果。

⑶ 实用性：本书抛弃了一些过时的内容，如网状数据模型、层次数据模型等，充分考虑现实世界中数据库技术的应用特点，重点突出了 SQL 数据库技术，在数据库设计、数据库安全性、故障恢复等内容上紧紧围绕 SQL 数据库展开介绍。同时，本书中所涉及的过程化 SQL、数据库设计 CASE 工具、数据访问模型等采用流行的技术，与当前企业应用完全一致，可以有效地提升学生的数据库实践技能。

(4)前沿性：本书在内容设计上不仅涵盖了经典的关系数据库内容，还加入了前沿的NoSQL 数据库技术介绍，使学生不仅能够学习到关系数据库的成熟理论和技术，也能够了解国际上数据库领域的最新进展。

本书的内容设计参考了教育部制定的计算机领域本科教育教学改革试点工作计划(简称“101 计划”)对数据库系统课程的教学要求，同时还参考了大量的经典文献和近年最新的资料，力求使读者能够了解数据库技术的起源及最新的进展。

本书由中国科学技术大学金培权编写。衷心地感谢编辑为本书出版付出的辛勤劳动。在本书编写过程中参考了大量的文献资料，在此也向这些文献的作者表示谢意。

由于数据库技术的发展日新月异，加上作者水平有限，书中难免存在疏漏之处，敬请广大读者提出宝贵意见。

作　者

2022 年 12 月

目　录

第 1 章　数据库系统概述

数据库技术作为一种先进的数据管理技术，极大地推进了数据管理技术乃至计算机应用技术的发展，使企业和组织能够高效地存储、管理和使用日益增长的数据。为了深入学习和掌握数据库技术，必须首先明晰数据库领域中的一些基本概念，了解数据库系统的体系结构、数据库管理系统的分类、数据库技术的研究领域以及发展历史等内容，从而为后面的学习奠定基础。

内容提要：本章首先介绍数据库系统的相关基本概念，然后介绍数据库技术提出的背景和动机，接着讨论 DBMS 的功能、分类和架构，之后给出数据库语言的基本知识，最后概述数据库技术的发展历史。

1.1　基 本 概 念

数据库系统是对应用了数据库技术的计算机系统的一种概称，因此数据库技术基本都包含在数据库系统的范畴中。数据库系统涉及了一些最基本的概念。这些概念在现实应用中很容易混淆，也是学习数据库技术必须了解和区分的对象。

1.1.1　数据的定义

数据是数据库中存储和管理的基本对象。数据这个名词在现实生活中并不陌生，“财务数据”“采购数据”等都是在实际生活中经常听到或接触的。但是，什么是数据？下面给出数据的定义。

定义 1-1　数据是人们用来反映客观世界而记录下来的可以鉴别的符号。

这个定义的核心意思是“数据是符号”。之所以强调这一点是因为数据库系统除了存储与管理数据之外，还管理另一些内容(如 1.1.3 节介绍的数据库模式)。

由于现实世界中存在着不同类型的符号，因此数据也可分为两种基本类型：数值数据和非数值数据。数值数据记录了由 0～9 这 10 个阿拉伯数字所构成的数值，例如，职工张三的年龄是一个数值数据，学生李四的英语成绩也是数值数据。非数值数据包括像字符、文字、图像、图形、声音等特殊格式的数据。在实际应用中，非数值数据也很常见，如人的姓名(字符)、照片(图像)等。现有的数据库技术都可以同时支持数值数据和非数值数据的存储与管理。

在实际应用中，如果仅存储数据，一般来说是没有什么意义的。因为数据本身只是符号而已，而同样的符号在不同的应用环境中可能会出现完全不同的解释。例如，“65”这一数据在教学信息管理系统中可能表示某个学生某门课程的成绩，在职工管理系统中可能表示的是某个职工的体重，而在学生管理系统中可能表示计算机系 2011 级的学生人数。因此，数据与其代表的语义是分不开的，在存储数据的同时必须知道数据所代表的语义。

除了“93”“张三”这类表示单一值的简单数据外，现实生活中还存在着复合数据。

复合数据是由若干个简单数据组合而成的。例如，学生记录(李明，197205，中国科学技术大学，1990)就是由简单数据“李明”、“197205”、“中国科学技术大学”和“1990”构成的一个复合数据。复合数据同样和其语义是不可分的。像上面的学生记录，其语义在不同应用环境下可能完全不同，例如，在高校毕业生管理系统中可能表示“学生姓名，出生年月，所在学校，毕业年份”这样的语义，而在另一个系统中则可能表示了另一种语义，如“学生姓名，出生年月，录取大学，入学时间”。

1.1.2　数据库的定义

简单地讲，数据库是一个数据的仓库，它存储了一个数据的集合。但是，这种定义还不够确切，因为数据库中的数据并不是随便存放的，而是有一定的组织和应用特点的。严格的数据库定义如下。

定义 1-2　数据库(Database，DB)是长期储存在计算机内有组织的、可共享的大量数据的集合。

这个定义指出了数据库的几个特点。

(1)数据库是数据的集合，因此数据库只是一个符号的集合，本身是没有语义的。

(2)数据库中的数据不是杂乱无章存放的，而是有组织的，确切地说，是按一定的数据模型组织、描述和存储的。

(3)数据库中存储的数据通常是海量的。如果是少量的数据，通常不需要使用数据库技术来管理，借助文件系统就可以实现管理。实际上，存储的数据量越大，越能体现数据库技术相对于文件系统的优势。

(4)数据库通常是持久存储的，即存储在磁盘等持久存储介质上。

(5)数据库一般是多用户共享的。换句话说，如果一个数据集合只为单用户服务(如手机中的通讯录)，那么依靠传统的文件系统等数据管理技术基本也可以满足要求。只有在多用户共享的环境中，才能充分发挥数据库技术的优点。目前，除了少数专用的数据库产品外，绝大多数的商用数据库产品都是面向多用户应用的。

(6)数据库一般服务于某个特定的应用，因此数据间联系密切，具有最小的冗余度和较高的独立性。

现实世界中有银行数据库、航班数据库、图书数据库等面向特定应用的数据库，但是不存在通用的数据库。即便都是图书数据库，不同的应用环境对数据组织、数据存储等也会有不同的要求。例如，某学校中的图书数据库中需要存储每一种图书的供应商，而另一个学校则可能不需要保存。这些都会影响到数据库中数据的表示和组织方式。因此，数据库一般都是专门针对某个特定应用的。

1.1.3　数据库模式

数据库的语义用另一个概念即数据库模式(Database Schema)来表达。数据库模式的定义如下。

定义 1-3　数据库模式是数据库语义的表达，它是数据库中全体数据的逻辑结构和特征的描述。

图 1-1 显示了数据库与数据库模式之间的关系。两者之间的关系与“数据”和“数据

的语义”之间的关系是类似的。实际上，因为数据库本身就是数据的集合，所以数据库中所有数据的语义就构成了数据库的语义，即数据库模式。

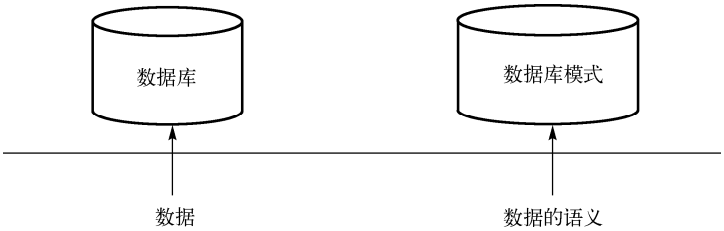


图 1-1　数据库与数据库模式之间的关系

图 1-2 所示为数据库与数据库模式的示例。在这个示例中，假设数据库中只存储了学生数据。图 1-2 中箭头的左边显示了使用关系数据模型表示的数据库结构与内容(如前所述，数据库中的数据一般都是按某种数据模型进行组织的)，右边则分别显示了对应的数据库和数据库模式。关系数据模型是目前最流行的数据模型(现有的商用数据库产品基本都基于关系数据模型)，它的基本数据结构就是图 1-2 中箭头左边显示的二维表格。但是，二维表格本身包含了表头和下面的数据行，从概念上讲，二维表格的表头表示了下方数据行的语义，其所对应的结构就是此二维表格的模式(由于假设数据库中只有这一个表格，因此此模式就是数据库的模式)，而下方的数据行集合则构成了数据库。

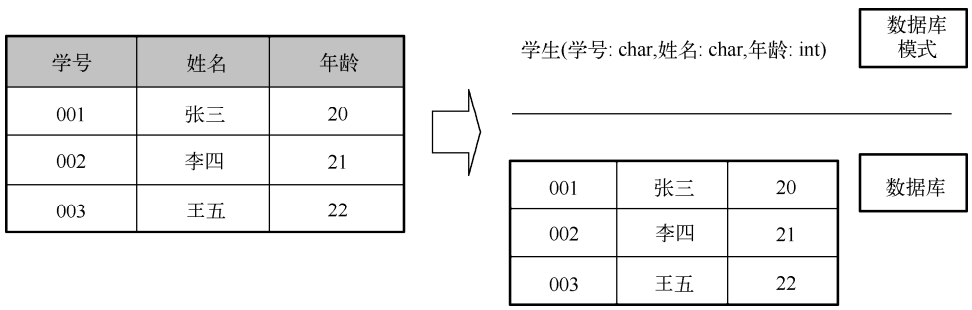


图 1-2　数据库与数据库模式示例

1.1.4　数据库管理系统

定义 1-4　数据库管理系统(Database Management System，DBMS)是一个计算机软件，它用于创建和管理数据库。

数据库管理系统从软件的分类上属于计算机系统软件。系统软件一般是管理计算机资源的软件。常见的系统软件有操作系统、数据库管理系统等。同样是系统软件，操作系统管理计算机中的全部资源，包括处理器、存储器、外设等，而数据库管理系统则只管理计算机中的数据资源。操作系统本身也有数据管理的能力，即文件管理功能，但正是因为操作系统在管理大规模共享数据时容易出现存取性能差、数据不一致等问题，所以才有了今天的数据库管理系统。可以这么理解，数据库管理系统是一种专门用于高效管理数据资源的系统软件。

通常情况下，数据库管理系统运行在操作系统之上，也就是说，当涉及底层的磁盘操

作时，数据库管理系统通常利用操作系统提供的磁盘存取服务来实现底层数据存取。目前，大多数的商用 DBMS 都采取了这种方式。但是，理论上，数据库管理系统也可以完全绕过操作系统提供的数据输入/输出(Input/Output，I/O)服务，直接跟底层的磁盘打交道。现在一些大型的 DBMS(如 IBM DB2、Sybase ASE 等)已经可以支持这种数据访问方式。此外，数据库管理系统通常不直接面向应用。作为系统软件，其职责在于管理和维护数据资源。

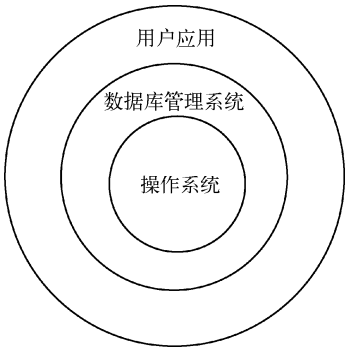


图 1-3　用户应用、数据库管理系统和操作系统之间的层次架构

同时，用户可以在数据库管理系统之上创建直接服务于应用的数据库应用系统(或称为数据库应用软件)，从而构建基于数据库技术的应用软件，满足实际应用的需求。图 1-3 显示了用户应用、DBMS 和操作系统之间的层次架构。

由于数据库中的数据需要按某种逻辑结构进行组织，因此任何一个数据库管理系统在实现时必须基于某种数据模型(数据模型描述了数据的逻辑组织结构、操作等内容，将在后面内容中详细讨论)，以保证所管理的数据库都能够按照统一的逻辑结构进行存储和存取。目前常见的数据库管理系统(如 Oracle、Microsoft SQL Server 等)都是基于关系数据模型的，因此也称为关系 DBMS，而另一些 DBMS(如 Versant O2 等)是基于面向对象数据模型的，人们通常称它们为面向对象 DBMS。正是因为数据模型决定了数据库管理系统中的数据组织和操作方法，所以基于什么样的数据模型成为区分数据库管理系统特征的最主要因素。

在实际应用中常见的一些数据库产品(如 Oracle、Microsoft SQL Server 等)在严格意义上讲都是指 DBMS，但随着计算机软件技术和应用的不断发展，目前的 Oracle、Microsoft SQL Server 已经不单纯是 DBMS，而是一套以 DBMS 为核心的套件，也就是说，它们不仅仅提供了 DBMS 的核心功能，还通常包含一些其他的软件功能，如数据导入/导出、备份管理等。这一点与 C++和 Visual C++的区别有点类似，C++好比 DBMS，而 Visual C++好比 Oracle 等各种大型商用数据库软件。虽然 Visual C++提供了多种多样的开发功能，但 C++是其核心，其本质仍然是 C++开发工具。在实际生活中，只要知道 Oracle、Microsoft SQL Server 这些产品的本质是 DBMS 就可以了。

图 1-4 显示了 DB-Engines 网站上排名前十的 DBMS 列表，可以看到，Oracle、MySQL、Microsoft SQL Server 目前仍是最受欢迎的三大 DBMS 产品。列表中还包括了 MongoDB、Redis 等 NoSQL 数据库系统，本书将在后续内容中对其介绍。

需要注意的是，DB-Engines 主要借助了互联网信息，根据 DBMS 的受关注程度对 DBMS 进行排名。数据的排名依据五个不同的因素。

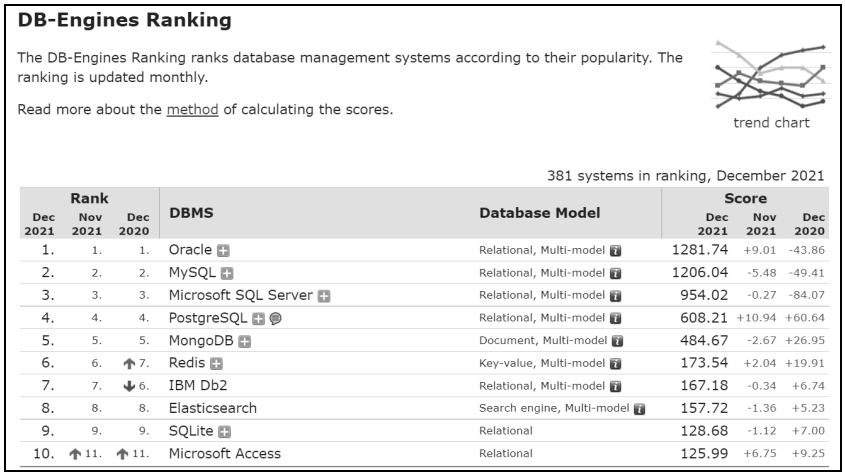
(1) Google 及 Bing 搜索引擎的关键字搜索数量。

(2) Google Trends 的搜索数量。

(3) Indeed 网站的职位搜索量。

(4) LinkedIn 上提到关键字的个人资料数。

(5) Stack Overflow 上的相关问题和关注人数。



DB-Engines Ranking

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

Read more about the method of calculating the scores.

trend chart

381 systems in ranking, December 2021

Rank Dec 2021	Rank Nov 2021	Rank Dec 2020	DBMS	Database Model	Score Dec 2021	Score Nov 2021	Score Dec 2020
1.	1.	1.	Oracle	Relational, Multi-model	1281.74	+9.01	-43.86
2.	2.	2.	MySQL	Relational, Multi-model	1206.04	-5.48	-49.41
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	954.02	-0.27	-84.07
4.	4.	4.	PostgreSQL	Relational, Multi-model	608.21	+10.94	+60.64
5.	5.	5.	MongoDB	Document, Multi-model	484.67	-2.67	+26.95
6.	6.	↑ 7.	Redis	Key-value, Multi-model	173.54	+2.04	+19.91
7.	7.	↓ 6.	IBM Db2	Relational, Multi-model	167.18	-0.34	+6.74
8.	8.	8.	Elasticsearch	Search engine, Multi-model	157.72	-1.36	+5.23
9.	9.	9.	SQLite	Relational	128.68	-1.12	+7.00
10.	↑ 11.	↑ 11.	Microsoft Access	Relational	125.99	+6.75	+9.25

图 1-4　DB-Engines 网站上的 DBMS 排名

因此，DB-Engines 的 DBMS 排名只代表了互联网中各个 DBMS 的受关注程度，某种意义上可以看作“热度排名”。这一排名没有包括实际的 DBMS 产品销售和装机数据，另外，也没有考虑中国市场的影响(例如，没有考虑中国用户主要使用百度搜索引擎)，因此仅具有参考价值。而且想给出一个综合各种因素的 DBMS 综合排名也有较大的难度，因为各类数据的获取存在困难。

1.1.5　数据库系统

数据库系统是一个泛指的概念，其定义如下。

定义 1-5　数据库系统(Database System，DBS)是指在计算机系统中引入了数据库后的系统，即采用了数据库技术的计算机系统。

因此，数据库系统与其他计算机系统的区别在于它是以数据库为基础的。目前社会上所见的许多系统，如银行信息系统、电子政务系统等，都可以称为数据库系统，因为它们背后都有 DBMS 和数据库的支持。随着应用数据类型和数据量的不断增长，在计算机系统中采用数据库技术来管理数据已经成为一种普遍的方案，因此数据库系统在实际应用中已经非常普及。

作为一个计算机系统，数据库系统同样包含了软件、硬件、用户等要素。一个数据库系统通常包括计算机硬件平台、操作系统、DBMS、应用程序及用户。图 1-5 给出了数据库系统的组成。

在图 1-5 中，把数据库系统中的用户区分为两种类型：终端用户与数据库管理员(Database Administrator，DBA)。从图中可以看到，终端用户直接与应用程序交互，而数据库管理员则直接跟 DBMS 打交道。终端用户相当于银行中的前台柜员，而数据库管理员则好比银行数据库系统的管理员和维护人员，他们负责管理和维护数据库系统的正常运行。通常，数据库管理员会通过对 DBMS 的一些参数进行配置和调优，或者利用 DBMS 提供的一些管理功能(如备份、监控、访问控制等)来维护系统。

数据库、数据库管理系统、数据库系统这三个概念在实际应用中容易混用，但是，在上下文清楚的情况下，可以用数据库来指代数据库管理系统或者数据库系统。举个例子，

“你们上机用什么数据库啊？”“Oracle 12g”，这一场景下数据库指的是 DBMS 的概念。此外，在论文或者其他规范化文档的写作中，要求严格区分这些概念，保证论文的严谨性。例如，不能将“高校信息化数据库的设计”混淆为“高校信息化数据库管理系统的设计”，前者是指数据库的设计，而后者是指 DBMS 的设计，两者的内涵和难度不是一个量级的。

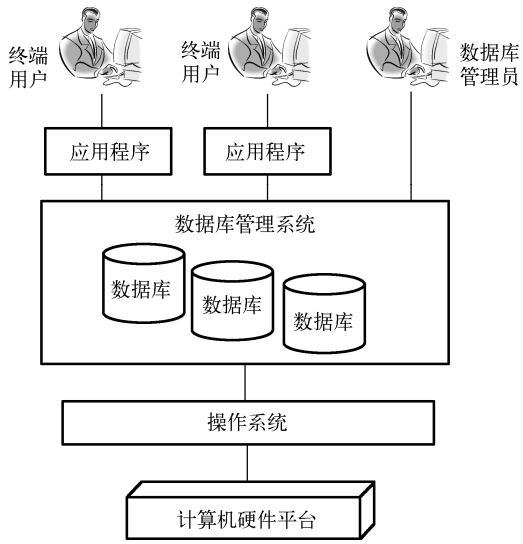


图 1-5　数据库系统的组成

1.2　使用数据库的原因

在数据库技术产生之前，现实应用中也同样存在着数据，也需要对数据进行存储和管理。过去的数据管理一般只能依靠文件系统，例如，通过 Excel 文件、Word 文档等来存储和管理数据。事实上，目前 Excel 等电子表格文件仍然是一些单位或部门中的数据管理手段。为什么现在大型的应用系统中必须使用数据库而不能使用 Excel 文件来存储和管理数据？本节将通过若干示例来加以说明。

1.2.1　利用文件系统管理数据的局限性

在数据库技术产生之前，文件系统是主要的数据存储和管理技术。一旦需要存储新的数据，就创建一个新文件并将数据写入，或者将数据增加到已有的文件当中。这种技术在现实世界中很常见，例如，一个部门可能会将自己的部门职工信息存在一个 Excel 文件中，由于部门职工人数不多，所以增加职工、查询职工等操作也都方便。

利用文件系统管理数据需要考虑两种情形：数据无共享和数据共享。当数据无共享时，每个用户或者应用程序维护自己的数据文件，不需要和其他用户或者应用程序共享数据。而在数据共享环境下，不同用户或者应用程序之间存在着共享的数据文件。

图 1-6 给出了数据无共享时的文件管理示例。图中给出了三个应用：学生管理、课程管理、教师管理(可以假设它们分别对应高校中的学生处、教务处、人事处)，它们分别管理学生数据、课程数据和教师数据。所有这些数据都以独立的文件存在，例如，学生数据

存储在 student.xls 文件中，课程数据存储在 course.xls 中，教师数据存储在 teacher.xls 中。如果数据不需要共享，则学生处、教务处、人事处都只需要维护自己的数据文件。在计算机网络技术产生之前，由于不同应用之间无法进行即时的网络通信，自然也无法实现即时的数据共享，所以，早期的基于文件系统的数据管理都采用如图 1-6 所示的方式，即各个应用各自维护自己的数据文件且相互之间不存在数据共享。

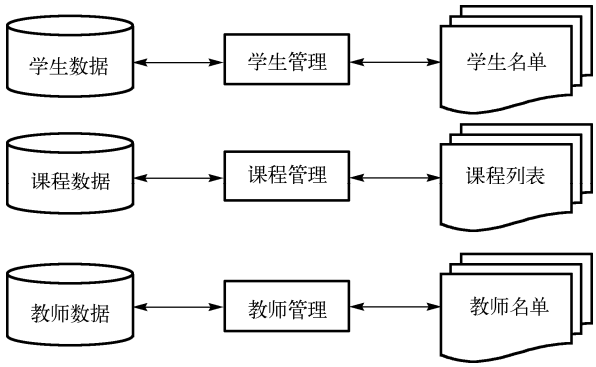


图 1-6　数据无共享时的文件管理示例

随着计算机网络技术的发展，不同用户之间的信息共享成为可能。在以文件形式管理数据的方式下，如果不同用户之间需要共享数据，则需要通过文件复制来实现。例如，教务处在制作课程选课名单时，需要同时用到学生数据、课程数据和教师数据，但由于学生数据在学生处，教师数据在人事处，此时只能从学生处和人事处复制学生数据和教师数据，从而完成课程选课名单的制作。图 1-7 给出了数据共享时的文件管理示例。此时，共享的数据以文件的方式在不同的部门中存有副本。

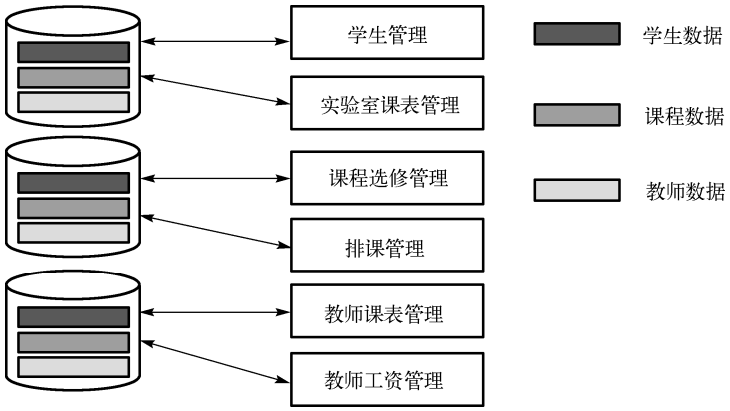


图 1-7　数据共享时的文件管理示例

图 1-7 所示的数据管理方式存在以下的问题。

1. 数据冗余和不一致

数据冗余是指同一份数据在不同部门(服务器)中冗余存储，因此存在着空间浪费。虽然目前磁盘价格不算昂贵，对于企业来说增加一部分存储成本不是大的问题，但数据冗余

的主要问题是其会带来数据不一致。例如，假设教师数据在人事处和教务处都存有副本，如果人事处更新了教师数据，但是教务处的教师数据没有同步更新，则两个部门的数据就出现了不一致。在实际应用中，部门之间的这种数据同步无法自动完成，某一份数据被越多的部门所共享，出现数据不一致的风险就越高。如果一个系统内的某一份数据出现不一致，则这些数据不可靠，无法在实际应用中使用。如果在不同的分行中查询同一个银行账户的余额得到了不同的结果，无论银行还是用户都是无法接受的。

2. 数据孤立和访问困难

采用文件系统管理共享数据时，所有的数据都是被不同的部门独立管理的，这导致整个系统范围内的数据是孤立存储的，相互之间的访问非常困难。这是因为每个部门可能都有着自己的文件安全管理策略，所以进行部门之间的文件访问时，需要了解每个部门的文件访问控制策略，获得授权，并且通过网络进行文件复制。这种方式在大规模数据共享与海量用户并发存取时会带来大量的网络传输和较大的访问代价，性能很差，无法满足实际需要。此外，如果通过程序访问共享数据，在文件系统中需要通过 API 来实现，但不同编程语言的 API 均不相同，这给用户访问数据带来了极大的学习负担。

3. 数据完整性问题

现实世界中的数据往往存在一定的约束，例如，学生的学号不能为空且必须唯一，课程的成绩不能为负，等等。数据应当满足的约束和条件通常称为数据的完整性约束。如果存储在系统中的数据违背了数据完整性约束，它们就不满足现实世界中的数据特征，就是错误的数据。而在文件系统中，一方面很难对数据的完整性进行约束，另一方面由于数据的副本都是被不同的部门独立管理的，会出现同一类数据在不同部门之间的完整性约束不同的情况。这些问题都是实际应用中无法接受的。

4. 数据操作的原子性问题

许多应用中存在着多个操作必须同时完成的情况，即多个操作的执行要满足原子性。例如，银行转账操作中，*A* 账户上扣款和 *B* 账户上进款这两个操作必须同时完成，不能只完成其中一个，否则就会导致转账出错。这一问题在文件系统中很难实现。即使在某个部门中实现了数据操作的原子性，由于其他部门还存在共享数据，很难保证整个系统内的数据原子性。一种可能的方式是先更新所有其他站点中的共享数据，再更新本地的数据，但这种方式首先会带来较大的操作延迟，其次也会带来数据访问中的冲突问题(更新了某个站点中的共享数据，但另一个站点中的同一共享数据此时正在被用户读取)。

5. 数据并发操作问题

由于一个系统内通常存在着多个用户，因此同一时刻就会出现多个用户访问同一数据的情况。这种操作在数据库中称为并发操作。并发操作的主要问题是其会导致并发冲突，例如，用户 *A* 修改完数据 *a*，用户 *B* 接着把数据 *a* 读走了，但用户 *A* 又把数据 *a* 的修改撤销了，此时，用户 *B* 读到的数据就是错误的数据。利用文件系统管理共享数据时，同一数据的并发操作很难进行控制，由此会带来一系列的数据更新丢失、读到过时数据等问题。

6. 数据安全性问题

数据以文件方式进行存储和管理时，其安全性完全依赖于文件系统本身的安全性。文件系统只能提供文件粒度的访问控制，无法对文件内部的数据进行细粒度的访问控制。例如，只要某个用户可以读取某个文件，该用户就能看到该文件的全部信息。而现实应用中，不同用户的数据读取权限显然是有差别的。此外，文件系统的安全性无法防止数据的窜改，只要用户能够访问文件系统(用户登录了某台计算机)，就有可能窜改文件中的内容，而其他用户再访问该文件时，很难发现数据已经被窜改了。总而言之，在文件系统中，数据的安全性很难进行系统性的管理，也无法满足实际应用中不同用户对数据安全性的不同要求。

1.2.2　利用 DBMS 管理数据的优点

数据库技术采用 DBMS 统一组织和管理数据，可以有效地解决文件系统管理数据时的问题。

如图 1-8 所示，数据库技术的核心思想是将所有数据统一组织、统一存储和统一管理。这一功能主要借助 DBMS 来实现。在这一架构下，整个系统范围内的数据都集中存储在一个统一的数据库中，不同应用访问数据时也均通过 DBMS 来实现。

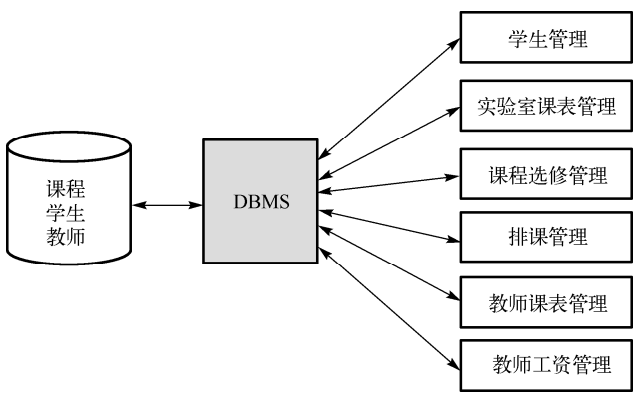


图 1-8　使用数据库技术统一管理数据

与文件系统相比，数据库技术的主要优点可归纳为以下几个方面。

1. 支持数据共享

存储在数据库中的数据可以被不同用户共享存取，用户不必担心共享数据之间的数据冗余、数据不一致等问题。DBMS 承担了数据管理和维护的全部工作，可以保证数据共享存取时的一致性和安全性。

2. 减少冗余

DBMS 对共享数据进行统一组织和存储，因此不会像文件系统那样通过简单的数据复制来支持数据共享。数据库的结构通常基于某种高效的数据模型(如关系数据模型)设计得

到，从而可以在理论上保证数据存储时的低冗余。需要注意的是，DBMS 并不能完全消除数据冗余，而是尽量实现数据的最低冗余存储。

3. 保证数据一致性

数据一致性是现实世界中数据存取的一个关键要求。如果存储在数据库中的数据不一致，则对用户而言就是错误的数据，无法使用。DBMS 中的事务处理、日志与恢复、并发控制等模块通过一系列的先进理论和技术，保证了存储在 DBMS 中的数据的一致性。例如，当用户在数据存取过程中由于各种原因而导致系统出现故障时，DBMS 的日志与恢复模块能够准确地判断数据是否处于不一致状态，并将数据恢复到离故障时刻最近的一致状态。

4. 提供事务支持

目前流行的关系数据库技术提供了事务处理的支持，允许用户将若干数据库操作定义为事务提交给 DBMS 执行。DBMS 的事务处理模块可以保证事务执行时的原子性、一致性、隔离性和持久性，从而用户在执行类似银行转账这类操作时，不必担心发生操作被中断时转账操作只完成了一半的情况。事务支持是目前数据库技术的一个重要特色，它也为数据库技术的实际应用提供了有力的支持。

5. 保证数据完整性

DBMS 的完整性模块提供了对数据完整性的支持。因此，当用户在设计数据库的过程中给出了完整性约束时，DBMS 会保证数据操作过程中这些完整性约束的有效性。由于完整性约束实际上是对现实世界中的数据特征的断言，因此 DBMS 可以保证存储在数据库中的数据始终满足用户定义的数据约束。

6. 增强了数据安全性

相对于文件系统而言，DBMS 提供了自身的安全性控制系统，可以支持针对数据库内部数据的细粒度访问授权和验证，例如，允许用户 *A* 读取数据库的 *a* 表但不能修改，允许用户 *B* 执行某些数据库操作语句，等等。

7. 提供并发控制支持

数据库技术是随着数据共享需求的提出而逐步发展起来的，因此数据的共享访问是数据库技术的一个基本特征。在共享访问的环境中，针对同一数据的多用户并发操作普遍存在。但是，现有研究已经表明，如果不对用户的并发操作进行控制，将会出现一系列的问题，如丢失更新、脏读、不一致分析等。这些问题的出现会破坏数据的一致性，但传统的文件系统缺乏有效的并发控制机制。并发控制功能是当今主流的关系 DBMS 的一个重要功能，它提供了对多用户并发操作的控制，并通过锁机制等有效地保证了并发操作时的数据一致性。并发控制技术的存在，使得现在的银行、证券等系统不必担心大量读写请求同时发生在同一数据上时的数据一致性问题。

8. 标准化

目前主流的关系 DBMS 采用标准化的结构化查询语言(Structured Query Language，SQL)存取数据。SQL 是与前端的编程语言独立的，这意味着不管用户采用何种编程语言开发数据库应用系统，都需要而且只能通过标准的 SQL 访问数据库。这一方面大大降低了用户对数据库访问技术的学习负担，另一方面促进了数据库技术的市场化和实用化。事实上，SQL 已成为全世界最流行的程序设计语言之一。

1.3　DBMS 的功能

DBMS 是数据库系统中的核心部件，它与操作系统、编译程序等被认为是主要的系统软件之一。DBMS 主要的功能就是管理数据库，其具体功能大致可划分为以下几个方面：数据库定义、数据库操纵、数据库保护、数据库维护。

1. 数据库定义

DBMS 首先需要提供创建数据库的功能。从 DBMS 的视角，数据库是由若干对象构成的一个集合，因此 DBMS 需要提供对不同数据库对象的创建和维护能力，包括表、视图、索引、约束、用户、存储过程、触发器等。在目前流行的关系 DBMS 中，这些对象的定义都通过标准的 SQL 语句来完成。

2. 数据库操纵

数据库是为前端应用程序服务的，因此 DBMS 必须提供数据库的存取能力。数据库操纵指对数据库的各类存取操作的实现，主要包括数据库的增、删、改、查。同样，在目前流行的关系 DBMS 中，数据库操纵也通过标准的 SQL 语句来完成。

3. 数据库保护

为了保证数据库的安全可靠，DBMS 必须提供一定的数据库保护功能。数据库保护通常包括两种方式：一是提供数据库故障后的恢复功能；二是提供防止数据库被破坏的技术。具体的数据库保护功能包括数据库恢复、并发控制、完整性控制、安全性控制等。通过这些数据库保护功能，DBMS 可以保证数据库的一致性、安全性和可靠性。

4. 数据库维护

数据库维护功能包括初始数据的转换和装入，数据备份，数据库的重组织、性能监控和分析等，这些功能对于保证 DBMS 的实用性是必不可少的。在实际的数据库应用系统中，系统正式运行之前通常要求数据库中有一些必需的初始数据。这些数据或者来自遗留的旧系统，或者是应用系统运行必需的一些先验数据。数据库的备份、性能监控等也是维护数据库所不可或缺的功能。这些功能通常由 DBMS 提供的一些实用程序完成。

1.4　DBMS 的分类

由于采用不同的数据模型和实现技术，DBMS 也存在着不同的类型。数据库自产生以来形成了许多子分支，如面向对象数据库、时空数据库等，这些新型的数据库技术通常都是因为采用了新的数据模型或者新的数据库技术而形成的。DBMS 可以从几个不同的角度来分类。

1．按数据模型分类

每个 DBMS 的背后都有一个所基于的数据模型，因此按数据模型来划分 DBMS 是最常见的一种方式。

按数据模型分类，DBMS 可以分为以下几种类型。

(1)网状 DBMS：基于网状数据模型的 DBMS。

(2)层次 DBMS：基于层次数据模型的 DBMS。

网状数据模型和层次数据模型都是 20 世纪 60 年代提出的数据模型，对于数据库技术的发展有重要的意义，不过目前已经很少使用。一般把网状 DBMS 和层次 DBMS(或网状数据库和层次数据库)称为第一代 DBMS(或第一代数据库技术)。

(3)关系 DBMS：基于关系数据模型的 DBMS。关系数据模型采用了完全不同于网状数据模型和层次数据模型的数据建模方法，自 20 世纪 70 年代提出后得到了快速发展和广泛应用，到目前为止仍是主流的数据库技术。已有的商用 DBMS(如 Oracle、MySQL、Microsoft SQL Server 等)都是关系 DBMS。一般称关系 DBMS 为第二代 DBMS(或第二代数据库技术)。此外，由于目前的关系 DBMS 普遍以标准的 SQL 作为数据库语言，所以很多时候也以“SQL 数据库技术”来指代关系数据库技术。

(4)面向对象 DBMS：基于面向对象数据模型的 DBMS。面向对象 DBMS 是在面向对象程序设计技术的基础上发展起来的，自 20 世纪 80 年代提出后得到了学术界和企业界的大力支持。但是由于面向对象 DBMS 存在一些固有的问题，目前还难以取代关系数据库技术。20 世纪 90 年代，研究者又将面向对象数据模型的一些特征加入到关系 DBMS 中，推出了对象关系 DBMS。有些学者试图把面向对象 DBMS 和对象关系 DBMS 统称为第三代 DBMS(或第三代数据库技术)，但相比对象关系 DBMS 的发展态势和影响力，面向对象 DBMS 显然要差一些。

(5)NoSQL DBMS：NoSQL 是近年来随着互联网应用的发展而提出的新型数据库技术。大多数人容易误解 NoSQL 为“Not SQL”，但实际上 NoSQL 指的是“Not Only SQL”。关系 DBMS(SQL 数据库技术)虽然应用广泛，有很多优点，但它在复杂数据类型支持、扩展性、支持高吞吐应用等方面依然存在不足。NoSQL 试图摆脱关系 DBMS 对于数据库模式、数据类型等的约束，使其适合分布式环境下的大数据应用。目前还没有关于 NoSQL 的标准定义，一种观点是把所有除 SQL 数据库技术之外的数据库技术都归纳到 NoSQL 范畴，包括传统的网状 DBMS、层次 DBMS 和面向对象 DBMS，但这种划分显然过于粗糙；另一种观点是不对 NoSQL 做明确的定义，这是因为 NoSQL 本身是为了弥补 SQL 数据库技术在表达复杂数据类型以及扩展性方面的不足而提出的，所以一种固定的定义很难满足未来 NoSQL 的不断演化。事实上，现在国内外已经提出了 200 多种 NoSQL 技术。目前流

行的 NoSQL 技术主要有键值数据库技术、文档数据库技术、列存储数据库技术、图数据库技术、时序数据库技术等。随着新型应用和新数据类型的出现，NoSQL 的技术类型依然在不断发展，未来可能会出现更多类型的 NoSQL 产品。

总体来说，第一代 DBMS 是开创者，直接导致了数据库领域的诞生；第二代 DBMS 是推动力，极大地推动了数据库技术的发展和应用，产生了像 Oracle 这样的著名数据库企业；第三代 DBMS 是探索者，对数据库技术在图像、图形、文档等复杂数据管理方面起到了重要的作用。NoSQL 数据库技术是目前互联网等领域的新宠，与关系 DBMS 相比各有优势和劣势，形成了数据库技术在不同领域的互补态势。

2. 按支持的用户数分类

按支持的用户数分类，DBMS 可分为单用户 DBMS 和多用户 DBMS。

(1) 单用户 DBMS：同一时刻只允许一个用户使用数据库。这种 DBMS 一般使用在早期的单机上，只能满足单个部门的数据管理要求；或者用在智能手机等单用户系统中。单用户 DBMS 目前已经很少见。

(2) 多用户 DBMS：可以同时支持多个用户并发访问数据库。目前常见的 DBMS 基本都是多用户 DBMS，包括 Oracle、MySQL 等。多用户 DBMS 适合企业的实际应用情况，因此得到了广泛的应用和发展。

3. 按允许数据库分布的站点数分类

按允许数据库分布的站点数分类，DBMS 可分为集中式 DBMS 和分布式 DBMS。

(1) 集中式 DBMS：数据库存放在单个物理站点。这是目前常见的 DBMS 类型。

(2) 分布式 DBMS：允许数据库分布存储在多个物理站点上。分布式 DBMS 需要考虑如何划分数据、如何将数据库分配到多个物理站点上、如何在多个站点间复制数据等问题，同时在查询处理、恢复、并发控制等方面也都需要研究新的技术。大部分的商用集中式 DBMS 都可以提供专门的分布式模块来实现分布式 DBMS 的功能。此外，NoSQL 数据库系统一般都是分布式 DBMS，这是因为 NoSQL 是在互联网应用背景下发展起来的，而互联网应用天然就是分布式的。

4. 按用途分类

按用途分类，DBMS 可分为通用型 DBMS 和专用型 DBMS。

(1) 通用型 DBMS：可应用于不同类型的应用环境，能够适应不同数据的管理需求。Oracle、MySQL、Microsoft SQL Server 等常见的 DBMS 都是通用型 DBMS，它们可以用于小型企业的信息管理系统，也能够满足银行、跨国企业、学校等不同应用系统的数据管理需求。实际上，由于数据库技术最早提出的动机就是通过统一的数据库服务器对数据进行集中存储和管理，所以采用一个统一的 DBMS 管理所有的数据并支持所有的应用，即 One Size Fits All 思想，一直是数据库领域广泛接受的观点。在 21 世纪之前，由于互联网应用还没有兴起，通用型 DBMS 得到了快速发展。但随着互联网应用的爆炸式增长，通用型 DBMS 在应对复杂多变的业务时捉襟见肘，这也是近年来 NoSQL 数据库技术快速崛起的主要原因之一。

(2) 专用型 DBMS：专门针对某些特殊的数据管理而研制，如针对时间/空间数据管理的时空数据库管理系统、针对多媒体数据管理的多媒体数据库管理系统等。它们对于这些特殊的数据管理可能是高效的，但不一定适合其他类型的数据管理，因此适合应用在某些特殊的应用系统中，如航空航天、遥感监测等。

1.5　DBMS 的架构

从软件系统的角度，DBMS 的架构一般可分成五层，俗称“DBMS 的五层架构”。各层模块实现了对数据库的不同抽象，向上对用户提供查询语言的接口，向下将磁盘中的字节流数据封装为磁盘块(也称为磁盘页)。通过五层架构，DBMS 可以支持用户对数据库的存取操作，也能够在系统的可维护性、独立性、性能、可靠性等方面实现合理的折中。事实上，很多 DBMS 的技术都是在时间性能、空间代价、可维护性等方面的折中，因此是目前技术现状下的最优选择，但不一定一直是最优选择。未来随着技术的发展，如持久性内存等，DBMS 的经典架构和技术依然有可能继续发展，最终目标是使得 DBMS 存储和管理数据库能够更高效、更灵活、更可靠。

如图 1-9 所示，DBMS 的架构明显分成了两个部分：左边的五层架构主要完成了接收用户查询语句并返回查询结果的整个处理过程，而右边的部分则主要负责事务管理、恢复、并发控制等数据库保护工作。这两部分的设计目标和功能要求存在着较大的差别。下面自底向上简要描述各个组件的主要功能。

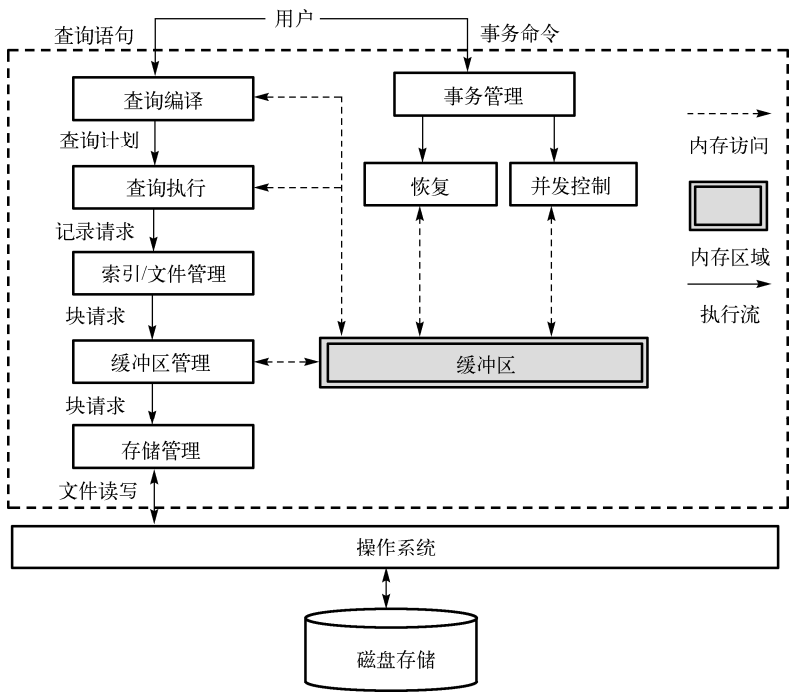


图 1-9　DBMS 架构

1. 存储管理

一般情况下，DBMS 都以文件作为数据库的物理存储形式，因此 DBMS 构建在操作系统之上，依赖操作系统中的文件系统完成文件的操作。需要注意的是，理论上 DBMS 也可以不需要借助文件系统，但这样需要 DBMS 负责底层磁盘存储器的控制和存取。目前一些大型的 DBMS（如 Oracle、DB2 等）提供了这类选项。

存储管理模块（也称为存储管理器）的主要功能是将磁盘中的字节流文件抽象为磁盘块结构。为什么不直接访问字节流文件而是把文件抽象为磁盘块的集合？这主要是为了优化磁盘访问的性能。磁盘最小的物理存取单位是一个扇区（通常是 512 字节）。当需要访问磁盘中的某个扇区时，系统需要执行一系列的寻道（磁头定位到扇区所在磁道）、旋转（旋转盘片让扇区的起始位置到达磁头下方）、传输（扇区通过磁头下方传输数据）等操作。这些操作的延迟相对较高，尤其是寻道操作，一般需要 10ms 左右的时间（具体和磁盘型号相关），是所有操作中延迟最高的。因此，如果每次都以扇区为单位从磁盘中读写数据，当一条记录是 4KB 时，则需要将其转换为 8 个扇区的读写操作，而每个扇区的读写都需要花费寻道、旋转、传输的时间，如此一来，读写一条 4KB 的记录就至少需要花费 8 次寻道时间。如果将数据的空间分配和管理改成以 8 个连续的扇区构成的 4KB 大小的磁盘块为单位，则当需要写入 4KB 数据时，系统只需要执行 1 次寻道操作即可，从而可以大大减少数据访问时的磁盘操作时间。基于这一原因，目前 DBMS 大都采用以磁盘块为单位的空间分配和管理模式，也就是说，虽然数据库数据存储在字节流文件中，但 DBMS 每次访问数据库文件时按照而且只能按照以磁盘块为单位进行。因此，磁盘块是 DBMS 与磁盘之间交互的最小单位，也是唯一允许的单位。

磁盘块是 DBMS 中的一个逻辑单位，因此不同的 DBMS 可以设计不同的磁盘块大小，例如，MySQL 默认磁盘块大小是 16KB，而 Microsoft SQL Server 则是 8KB。大型的 DBMS（如 Oracle、DB2 等）允许用户根据实际需要自行决定数据库的磁盘块大小。例如，如果一条记录是 4.1KB，而且记录不允许跨块存储，如果采用 8KB 的磁盘块，则整个数据库的空间利用率近似为 50%，空间效率很差。如果选择 32KB 的磁盘块，则一个磁盘块大约可以容纳 7 条记录，空间利用率可提升到 90%左右。需要注意的是，不是所有 DBMS 都允许调整磁盘块大小，像 Microsoft SQL Server 就采用了固定磁盘块大小（8KB）的设计。

以上是关于存储管理的一些背景。可以看到，存储管理模块需要密切考虑底层磁盘存取的特点，并采用磁盘块的方式进行空间分配、管理和数据访问。因此，存储管理模块的主要职责就是将底层字节流文件转换为磁盘块集合，从而向上层模块（缓冲区管理模块）提供数据库的磁盘块抽象。换句话说，从上层的缓冲区管理模块看，整个数据库就是一个磁盘块的集合，而这一层的抽象由存储管理模块负责实现。

2. 缓冲区管理

缓冲区管理模块（也称为缓冲区管理器）是 DBMS 中不可或缺的一个组件。缓冲区实际上就是一块内存区域，它缓存了最近访问的一部分数据，从而，当用户请求读取数据时，DBMS 首先会检查缓冲区中是否存在所读取的数据。如果存在（称为缓冲区的一次命中），则直接将缓冲区中的数据地址返回给上层应用；如果不存在，则缓冲区管理器会向存储管

理器请求从磁盘读入相应的磁盘块并放入缓冲区中，同时将缓冲区中的数据地址返回给上层应用。

因此，缓冲区内的原始数据都是 DBMS 通过存储管理器读入的。由于存储管理器从磁盘读取是以磁盘块为单位的，所以缓冲区内的数据通常也是以磁盘块的粒度进行组织的。缓冲区管理器接收的数据读取请求来自上层的索引/文件管理器，也是针对磁盘块的请求，所以，缓冲区管理器实际管理的就是在缓冲区内的磁盘块集合，这个集合一般称为“缓冲池”（Buffer Pool）。

由于内存（DRAM）相对于磁盘而言容量小，例如，目前服务器的内存一般是 GB 级，而磁盘通常是 TB 级甚至 PB 级的，因此，缓冲区一般认为是一种有限的资源。随着数据访问的不断累积，缓冲区最终会被最近访问的磁盘块填满。此时，如果再需要从磁盘读入新的磁盘块，则必须在缓冲区中选择某个内存块并将其移出缓冲区，以腾出空间来容纳当前需要读入的磁盘块。按照什么样的策略来选择要换出的内存块，对于缓冲区管理的效率有着重要的影响，相应的算法称为缓冲区置换算法。常见的缓冲区置换算法有 LRU、FIFO、Clock 等，其中 LRU 是目前常用的置换算法。Oracle、Informix 等均采用了 LRU 算法。

3. 索引/文件管理

索引/文件管理模块（也称为索引/文件管理器）的主要任务是将上层面向记录的访问请求转换为对磁盘块的请求，并将对磁盘块的请求发送给下层的缓冲区管理器，所以，它在 DBMS 的整个查询处理过程中是中央的一环。在索引/文件管理器之上，数据库展现的是记录的集合，而在它之下，数据库即转变为磁盘块的集合。

索引/文件管理器接收记录请求，输出该记录所在的磁盘块地址（即磁盘块号）。这个过程在大多数情况下是通过数据库索引来实现的。数据库索引结构主要用于加快根据记录键值来查找记录所在磁盘块地址的过程，其原理与《新华字典》的目录类似：当需要查找“数”这个字时，首先在《新华字典》的目录（如按拼音 A、B、C 开头的索引）中，找到“数”的拼音“shu”所在的目录页，然后得到“shu”后面记录的字典页码，如第 450 页，最后翻到字典第 450 页即可找到“数”这个字。

在上面的示例中，《新华字典》的前几页目录就是“索引”。这些目录也是以页的形式存在的，所以索引在数据库中的存储方式也是页的集合，即文件，称为索引文件。这也是 DBMS 把索引和文件管理组织为一个模块（即索引/文件管理）的原因，因为其本质上都是对文件的搜索。

通过数据库索引，DBMS 可以快速地根据给定的记录键值（如上面《新华字典》查找示例中的“数”字）找到记录所在的磁盘页号。在实际数据库中，索引是数据库设计者根据应用需求而人工设计的。如果不存在数据库索引（想象一下，《新华字典》的前几页拼音目录现在不存在了），此时数据库文件就单纯变成了一个无序的页的集合。为了得到给定记录所在的页号，文件管理器只能一页一页遍历数据库文件。这个过程显然需要较大的时间代价，所以，如果要查找的记录不存在索引，访问代价必然会上升。这也是现在的数据库系统性能依赖数据库索引的主要原因。

目前 DBMS 中最常用的索引结构是 B^+-tree。它是一种多叉、平衡、有序的面向磁盘块

的树形结构。B^+-tree 的每个结点都是一个磁盘块。其他常见的索引包括散列索引、位图索引等。不同的 DBMS 对于索引结构的支持有所不同，并且不同类型的索引对于查询的适用性也不同，例如，Oracle 支持位图索引，但不支持散列索引；散列索引只支持点查询，对于范围查询无效。

需要注意的是，一个数据库索引只对索引的记录键值上的查询优化有效。例如，在学生记录的“学号”字段上面设计了一个索引，当按照“学号”去查找记录时，该索引可以用于加速查询；但如果按照“学生姓名”去查询，则“学号”上的索引没有任何帮助。以前面《新华字典》查找为例，字典的目录(即索引)只能支持按拼音、笔画或者偏旁查找某个汉字，其他类型的查找(如按字形结构查找)均无法利用索引。

实际数据库设计中，在哪些字段上需要设计索引，以及设计哪种索引是重要的问题，设计索引也对数据库系统的性能至关重要。它与数据库的访问模式有着密切的联系。像《新华字典》查找这种访问模式都是针对单个汉字的点查询，而像银行交易记录查找这种访问模式通常是范围查询(如按日期范围或者按金额范围)，这些都需要在索引设计时加以考虑。

4. 查询执行

查询执行模块的主要职责是执行查询编译模块生成的查询计划。查询计划可以大致看成一个 API 或者函数的执行序列。查询执行模块接收到查询计划后，就会按照查询计划给出的执行顺序逐步执行各个操作，包括数据读取、中间结果写出等。当然，实际 DBMS 中的查询计划还涉及很多其他的要素，如中间结果的传递方式等。

当查询执行模块执行查询计划中的操作时，如果需要读写记录，则会调用下层的索引/文件管理器提供的接口，然后把执行流逐步往下推，直到缓冲区管理器从内存中直接返回数据地址或者存储管理器从磁盘中读入磁盘块到缓冲区中并返回内存地址。

5. 查询编译

查询编译模块(也称为查询编译器)是用户与 DBMS 之间的查询接口。它接收用户的数据查询语句，通过一系列的编译和优化处理，将其转换为最优的查询计划，并交给下层的查询执行模块执行。

目前，SQL 是数据库领域最为流行的数据库语言。在 SQL 数据库系统中，所有的数据操作均以 SQL 语句的形式表示。因此，查询编译器接收的用户查询通常就是 SQL 语句。首先，查询编译器需要对用户发出的 SQL 语句进行语法检查，这个过程通常需要借助构建与 SQL 语句对应的语法分析树来完成。之后，DBMS 将基于数学方法对 SQL 查询进行优化，所以，查询编译器会将通过语法检查后的 SQL 查询转换为一个初始的查询计划树。查询计划树是以关系代数操作符为中间结点，以关系为叶结点的树(与关系代数和关系数据模型相关的内容将在第 3 章中详细介绍)。接着，查询编译器对初始的查询计划树进行一系列的优化操作，最终得到一个与初始查询计划等价但是估算的查询代价最低的查询计划，这就是对应用户 SQL 查询的最优查询计划。

查询优化是查询编译器最重要的任务。实际上，如何得到一个 SQL 查询的代价最低的查询计划，是数据库领域几十年来一直关注的一个研究方向。

6. 事务管理

前面介绍的五个 DBMS 模块总体上形成了一个完整的查询处理流程，其目的是快速响应用户的数据查询请求，所以，它们的设计目标均以优化时间性能和降低空间代价为主，尤其是时间性能的优化。但是，事务管理、恢复、并发控制这三个 DBMS 组件是围绕数据库保护的目标而设计的，其主要目的是保证数据库中数据的一致性。这是因为数据现在已经跟资本、人才等一样都是企业的重要资产，所以必须要保证存储在数据库中的数据安全可靠，否则用户不会信任 DBMS，也不敢将关键的数据全部存储在数据库中。事务管理、恢复和并发控制这三个组件构成了目前 DBMS 中主要的数据库保护机制，为数据库中的数据一致性提供了有力的技术保障。在这三个组件中，事务管理是恢复和并发控制的基础，换言之，恢复和并发控制都是以事务管理为基础来设计和实现的。

一个典型的事务示例是银行转账。例如，账户 *A* 向账户 *B* 转账 100 元。这一事务在 SQL 中需要通过两次更新操作来实现：第一次更新操作将账户 *A* 的余额减去 100，第二次操作将账户 *B* 的余额增加 100。在实际应用中，这两次更新操作是不可分的，要么都执行，要么一个也不执行。

事务管理模块(也称为事务管理器)的主要职责是接收用户的事务命令，创建事务(每个事务赋予唯一的事务 ID)，并保证事务的 ACID 性质。事务的 ACID 性质是指事务的原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)和持久性(Durability)。ACID 性质的详细含义将在后面章节中讨论。注意，DBMS 并不知道哪些数据库操作是一个事务，所以事务的定义必须由用户通过事务命令告诉事务管理器。在 SQL 中，事务命令只包含三条语句：Begin transaction、Commit transaction、Rollback transaction。第一条语句告诉事务管理器这是事务的开始，也就是这之后的数据库操作是属于这个事务的。后面两条语句定义了事务的结束标志，表示事务到此结束了。其中，Commit transaction 表示事务正常结束，而 Rollback transaction 表示事务取消了(例如，账户 *A* 向账户 *B* 转账时发现余额不足，无法正常完成事务)。

7. 恢复

恢复模块主要负责当数据库因为各种故障而被破坏时将数据库恢复到最近的一致状态。在正常执行的情况下，事务的 ACID 性质可以保证数据库永远处于一致的状态。但是，如果发生了故障，如宕机、掉电，甚至硬盘故障等，就可能导致数据库处于不一致的状态。此时，DBMS 中的恢复模块将执行一系列的恢复操作，保证数据库的一致性不会因为故障而被破坏。

DBMS 的故障大致可分为介质故障、系统故障和事务故障三大类。其中介质故障一般是磁盘损坏导致整个数据库数据都丢失；另外两种故障只导致内存数据丢失，但磁盘数据依然存在。总体上，目前 DBMS 采用的恢复策略是定期备份加上基于数据库事务日志的恢复。事务日志(Log)按时间顺序记录了每个事务对数据库所执行的更新操作。此处需要注意，事务日志只记录数据库的更新操作，因为查询操作不改变数据库的值，所以也不会破坏数据库的一致状态。当发生 DBMS 故障时，如果是介质故障，则重装系统并根据备份恢复到备份时刻的数据库状态，然后根据事务日志恢复到故障时刻的数

据库状态。如果是系统故障或者事务故障，则一般由 DBMS 自动执行基于日志的恢复过程。

目前常用的事务日志类型包括 Undo 日志、Redo 日志和 Undo/Redo 日志。它们都可以完成基于日志的数据库恢复，差别在于日志登记的规则、日志项的格式以及恢复策略不同。详细的内容将在后续的章节中讨论。

8. 并发控制

并发控制模块用于保证 DBMS 在执行并发操作时的数据库一致性。当多个用户同时访问同一数据时，如果对操作顺序不进行控制，容易出现丢失更新等问题，使数据库一致性遭到破坏。目前 DBMS 普遍采用基于锁的并发控制技术，详细内容将在第 11 章中讨论。

1.6　数据库语言

数据库作为数据集合，是为应用程序服务的。数据库只有被用户使用，才能体现出它的价值。用户如何才能高效地存取数据库？这要求 DBMS 本身必须提供相应的数据存取接口。理论上，数据存取可以有多种方式，如通过 API、通过组件、通过数据库语言等方式。

在数据库领域，目前标准化的方式是通过数据库语言(Database Language)进行数据存取。而且，通过数据库语言是用户存取数据库的唯一方式。在讨论具体的数据库语言之前，有必要先分析 DBMS 中有哪些数据需要用户存取。总体而言，DBMS 需要支持三类数据的存取。

(1)数据库中的数据：用户主要存取的数据。数据库中的数据来源于客户端的各类业务，主要支持前端的各类数据库应用。

(2)数据库模式信息：数据库模式存储了数据库的逻辑结构特征，代表了数据库中的数据语义。在实际应用中，数据库模式的定义也是需要用户来完成的，因为数据的语义与应用相关，只有用户才能准确解释数据的语义。用户定义的数据库模式与数据库中的数据一样存储在 DBMS 中，由 DBMS 管理和维护。因此，DBMS 也需要为用户提供定义和修改数据库模式的功能，即支持数据库模式信息的存取。

(3)数据库访问控制信息：描述了用户在数据库对象上的存取权限，它们同样需要用户根据应用的需求加以定义。因此，DBMS 需要为用户提供数据库访问控制信息的存取功能。

由于数据库语言是 DBMS 与用户之间的唯一接口，因此上述三类数据的存取均需要通过数据库语言来实现。为此，数据库语言通过设计三类子语言来实现，其中每一类子语言对应了一类数据的存取功能。

(1)数据定义语言(Data Definition Language，DDL)：专门用于定义和操纵数据库模式的语言。在第 2 章中将给出数据库模式的三级模式结构，即外模式、概念模式和内模式。这三级模式的定义和修改均通过 DDL 来实现。

(2)数据操纵语言(Data Manipulation Language，DML)：专门用于操纵数据库数据的语言，包括数据的增、删、改、查等。DML 是用户最常用的语言。对于前端的数据库应用程序而言，其主要的功能就是支持数据库数据的存取。在 SQL 中，DML 包含了 Insert、Delete、Update 和 Select 四条语句，分别对应数据的增、删、改、查。

(3) 数据控制语言(Data Control Language，DCL)：专门用于存取数据库访问控制信息的语言，主要用于用户的授权与验证。

早期的数据库语言还包含对数据库语言嵌入到程序设计语言(称为宿主语言)的支持。这是因为早期的编程语言(如 C 语言)并没有提供对数据库访问的支持，因此如果要在 C 程序中实现数据库存取，只能将数据库语言嵌入到 C 程序中，由数据库语言来完成数据库存取。但是，C 语言与数据库语言在语法、标识符等方面都存在差异，因此需要制定专门的规则来解决宿主语言如何支持数据库语言的问题。嵌入式数据库语言在早期数据库应用开发中常使用，但随着面向对象程序设计语言和开发工具的发展，目前除了在一些嵌入式平台上，已经很少使用了。

此外，数据库语言只是从 DBMS 角度为应用程序提供了数据库访问的方式。然而，在实际应用中，如何将用户的查询需求表达为数据库语言仍是一个有难度的问题。这要求数据库应用系统的程序员具有较强的数据库语言表达能力。如何利用数据库语言准确、高效地表达用户查询，是学习和使用数据库技术必须要掌握的主要知识之一。

1.7　数据管理技术发展历史

数据库技术是数据管理技术的一个发展台阶。在数据库技术之前，人们普遍采用文件系统管理数据。但是，随着数据规模的不断增长以及数据共享需求的提出，文件系统越来越难以适应数据管理的要求。另外，数据库技术自 20 世纪 60 年代开始发展以来，经历了网状数据库、层次数据库、关系数据库、面向对象/对象关系数据库等发展阶段，即使到了今天，XML 数据库、NoSQL 数据库等技术也在不断地发展。通过了解数据库技术的发展历史，可以看到整个数据库领域在最近几十年里的历程，对当今一些新的数据库技术的背景有更深入的理解。

1.7.1　早期的数据管理技术

数据库技术从概念上讲只是数据管理技术的一种，只不过因为数据库技术在数据管理方面具有高效、一致等优点，所以才得到了用户的认可。要了解数据库技术的发展历程，首先应当对整个数据管理技术的发展历程有所认识，明白数据库技术的发展背景。

数据管理技术的发展历程大致可归纳为三个阶段，即人工管理阶段、文件系统阶段和数据库管理阶段。

1. 人工管理阶段

人工管理阶段主要集中在 20 世纪 50 年代中期以前。当时计算机刚刚面世，处理能力还非常弱，数据管理能力也非常有限。

人工管理阶段的数据管理(图 1-10)具有以下的特点。

(1) 数据不保存在计算机中。此时还没有出现磁盘这样的二级存储概念，数据都是纯二进制数据，并且以打孔纸带的形式表示。

(2) 应用程序自己管理数据。应用程序根据自己的要求准备打孔纸带形式的数据，这些数据只能被自己使用。不同的应用程序需要各自准备数据。

(3) 数据无共享。

(4) 数据与应用程序之间不具有独立性。如果应用程序发生修改，原先的数据一般不能继续使用，需要再重新准备打孔纸带。同理，如果数据修改了，应用程序同样无法进行处理。

(5) 只有程序概念，没有文件概念。此时还没有文件存储的概念。

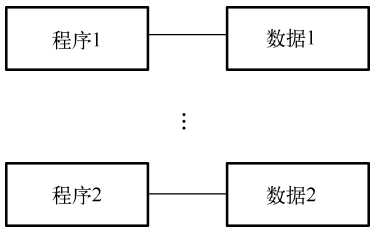


图 1-10　人工管理阶段的数据管理特点

2. 文件系统阶段

在 20 世纪 50 年代中期到 60 年代中期，出现了文件系统形式的数据管理技术。它主要是随着操作系统技术的发展而提出的。

这一阶段的主要特点是数据以文件形式保存和管理。图 1-11 给出了文件系统阶段的数据管理特点。这一阶段的数据管理的主要特点如下。

(1) 数据以文件形式存在，由文件系统管理。

(2) 数据可以长期保存在磁盘上。

(3) 数据共享性差，冗余大。冗余时必须建立不同的文件以满足不同的应用。例如，在一个教学信息管理系统中，教师数据同时被选课、科研管理、人事管理等应用使用，在文件系统阶段，只能将教师数据复制到这些不同的应用中。这一方面带来了数据的冗余存储，另一方面如果某些教师数据发生了修改，则很容易出现数据的不一致。

(4) 数据与应用程序之间具有一定的独立性，但非常有限。应用程序通过文件名即可访问数据，但文件结构改变时必须修改程序。

3. 数据库管理阶段

20 世纪 60 年后期开始，数据管理技术进入了数据库管理阶段。这一阶段引入了 DBMS 以实现数据管理(图 1-12)，其数据管理主要特点如下。

(1) 数据结构化。DBMS 采用了数据模型来组织数据，不仅可以表示数据，还可以表示数据间的联系。

(2) 高共享，低冗余。应用程序之间可以高度共享数据，并且可以保证数据之间的最低冗余。

(3) 数据独立性高。数据的修改不会影响到应用程序的运行，具有高度的数据独立性。

(4) 数据由 DBMS 统一控制，应用系统中所有的数据都由 DBMS 负责存取。

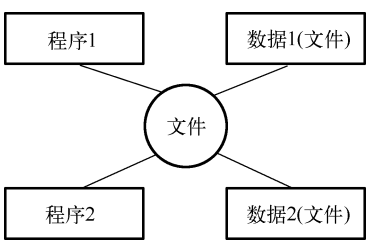


图 1-11　文件系统阶段的数据管理特点

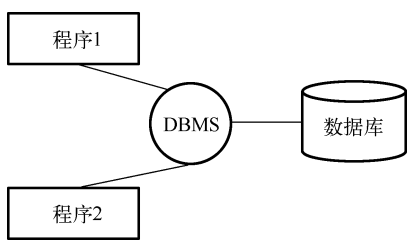


图 1-12　数据库管理阶段的数据管理特点

1.7.2　数据库技术的发展历程

数据库技术从 20 世纪 60 年代后期开始发展，至今仍是计算机领域中一个非常活跃的研究方向。下面简单地总结了数据库技术发展历程中的一些里程碑。

(1) 1961 年，通用电气(GE)公司的 Charles W. Bachman 设计了历史上第一个 DBMS ——网状数据库系统 IDS(Integrated Data Store)。Charles W. Bachman 是一名工业界的研究人员，为了解决 GE 项目中的复杂数据管理问题而设计了 IDS，开创了数据库这一新的研究领域。Charles W. Bachman 因为在网状数据库方面的贡献获得了 1973 年的计算机领域的最高奖项——图灵奖。这是第一位获得图灵奖的数据库研究人员。

(2) 1968 年，IBM 设计了层次数据库系统 IMS(Information Management System)。

(3) 1969 年，CODASYL 的 DBTG(Database Task Group)发表了网状数据模型报告，奠定了网状数据库技术。

(4) 1970 年，IBM 的 Edgar F. Codd 在 *Communications of ACM* 上发表了论文 *A Relational Model of Data for Large Shared Data Banks*，提出了关系数据模型，奠定了关系数据库理论基础。关系数据模型采用了一种简单、高效的二维表形式组织数据，从而开创了数据库技术的新纪元。Edgar F. Codd 因为关系数据模型方面的贡献获得了 1981 年的图灵奖。

(5) 1973～1976 年，Edgar F. Codd 牵头设计了 System R。System R 是数据库历史上第一个关系数据库原型系统，其字母 R 是 Relation 的首字母。之所以称为原型系统而不是产品，是因为 System R 开发完成后并没有及时商业化，从而 Oracle 的后来居上。在这期间，加利福尼亚大学伯克利分校的 Michael Stonebraker 设计了 Ingres。Ingres 是目前开源 DBMS Postgre SQL 的前身。在 20 世纪 70 年代，Ingres 是少数几个能和 IBM 的 IMS 竞争的产品。

(6) 1974 年，IBM 的 Ray Boyce 和 Don Chamberlin 设计了 SQL。SQL 最早是作为 System R 的数据库语言而设计的。经过 Ray Boyce 和 Don Chamberlin 的不断修改和完善，最终形成了现在流行的 SQL。目前，SQL 已经成为 ISO 国际标准。Ingres 就是因为其系统不支持 SQL 而最终没落。

(7) 1976 年，IBM 的 Jim Gray 提出了一致性、锁粒度等设计，奠定了事务处理基础。Jim Gray 因为在事务处理方面的贡献获得了 1998 年的图灵奖。

(8) 1977 年，Larry Ellison 创建了 Oracle 公司，1979 年发布 Oracle 2.0，Larry Ellison 早期在执行美国国防部(Department of Defense，DoD)的一个项目时遇到了数据管理方面的问题，后来他阅读了 Edgar F. Codd 发表的关于 System R 的论文，于是基于 System R 的思想很快地实现了一个数据管理系统，并且将其商业化。Oracle 采取了兼容 SQL 的做法，逐步占领了数据库领域的龙头地位。Oracle 的发展对于数据库技术的商业化起到了十分重要的作用。

(9) 1983 年，IBM 发布 DB2。Oracle 在商业领域的成功，使 IBM 意识到了数据库技术的发展前景。由于其技术实力雄厚，因此马上推出了商业化 DBMS DB2。这一产品至今仍在市场上占据重要的地位。

(10) 1985 年，面向对象数据库技术提出。面向对象数据库技术是随着面向对象程序设计(Object Oriented Programming，OOP)技术提出的，实质上是持久化的 OOP。

(11) 1990 年，Michael Stonebraker 发表《第三代数据库系统宣言》，提出了对象关系数据模型。面向对象数据库技术以及对象关系数据库技术标志着第三代数据库技术的诞生。但从商业应用上看，第三代数据库技术还远远赶不上关系数据库技术。

(12) 1987～1994 年，Sybase 和 Microsoft 合作，发布 Sybase SQL Server 4.2。Sybase 继续发布 Sybase ASE 11.0。

(13) 1996 年，Microsoft 发布 Microsoft SQL Server 6.5。虽然 Microsoft 和 Sybase 解除合作后都拥有了 Sybase SQL Server 的源代码，但 SQL Server 这一名称从此开始属于微软，而 Sybase 启用 ASE 产品名称。

(14) 1996 年，开源的 MySQL 1.0 正式发布。2009 年 MySQL 被 Oracle 收购，目前是数据库领域影响最大的 DBMS 之一。按照最新的 DB-Engines 排名，其流行度仅次于 Oracle，排在全世界第二位。

(15) 1998 年，半结构化数据模型(XML1.0)提出。由于网络数据管理需求的不断增长，XML 数据管理技术在近年受到了重视，至今仍是数据库领域的一个研究热点。

(16) 2005 年，Michael Stonebraker 等开发完成 C-Store。C-Store 是列存储的 DBMS，它完全抛弃了传统基于行记录的数据库存储方式，从而开创了一个全新的研究方向。Michael Stonebraker 因为在数据库系统创新和应用方面的贡献获得了 2015 年的图灵奖。

(17) 2007 年，NoSQL(非关系数据库)在 Web 领域占主导地位。传统的 SQL 数据库技术经过了几十年的发展和应用，在新的应用领域如 Web、云计算等中面临着一些数据表示、查询处理方面的新问题。因此 NoSQL 数据库技术开始提出并且马上得到了多个互联网企业的支持，包括 Amazon(Dynamo)、Google(BigTable)、Facebook(Cassandra)等。目前，NoSQL 在互联网领域得到了广泛的应用，但在传统领域如银行、证券、政府部门等还依然难以替代关系数据库技术。

1.8　本 章 小 结

本章主要介绍了数据库系统中的基本概念，包括数据、数据库、数据库模式、数据库管理系统、数据库系统等，还着重介绍了数据库技术产生的背景。此外，本章还系统介绍了 DBMS 的功能和分类、DBMS 的架构、数据库语言以及数据库技术的发展历史。

通过对本章的学习，读者应能够清晰区分数据库系统中的一些基本概念，了解 DBMS 的分类以及基本架构，并熟悉数据库领域的发展历史。

习　　题

1．数据库和数据库模式之间是什么关系？

2．数据库管理系统和数据库系统之间是什么关系？

3．第一代 DBMS、第二代 DBMS 和第三代 DBMS 分别指什么？

4．DBMS 的基本功能有哪些？

5．描述 DBMS 的基本组成。

第 2 章　数据库系统体系结构

数据库系统既涉及了 DBMS、数据库应用程序等软件，也包含了存储海量数据的数据库。数据库系统体系结构从架构的观点对数据库系统的软件以及数据库进行分析，梳理其内部的层次和相互之间的联系，从而为人们深入理解和应用数据库技术提供帮助。数据库系统体系结构可以从两种不同的角度来理解：一是从 DBMS 的角度看数据库内部的模式结构如何组织；二是从终端用户的角度看数据库系统的软件架构如何组织。

内容提要：本章首先介绍数据库系统体系结构的概况，然后着重介绍数据库三级模式结构和两级映像，最后讨论数据库应用系统的体系结构。

2.1　数据库系统体系结构概述

数据库系统与通常的计算机系统的主要不同之处在于它是以 DBMS 为核心的。DBMS 的引入一方面带来了数据库应用程序软件架构上的变化，另一方面也对如何合理组织和使用数据库提出了新的要求。

从软件架构上看，DBMS 引入之后，系统中开始出现了数据库服务器。由于商用 DBMS 不仅提供了数据存储的功能，还兼有一定的数据库编程能力，也就是说数据库服务器也有一部分的业务处理功能，因此在系统中如何合理地安排数据库服务器以及如何在应用程序和数据库服务器之间合理地分配业务处理能力是搭建数据库系统必须解决的问题。

此外，DBMS 只提供了数据库创建和维护的功能，而不同的应用对数据库的设计和使用要求存在一定的差别。例如，有的应用中数据库模式通常是稳定不变的，而有的应用中数据库模式有可能会变化。由于在软件工程中软件维护的代价越来越高，因此人们在建立数据库系统时不希望因为 DBMS 和数据库的引入而导致软件的维护代价大幅增大。一旦出现这种情形，将十分不利于数据库技术的进一步发展和应用，因为企业可能会宁愿放弃 DBMS 的高效数据管理能力而去寻求其他维护代价较小的解决方法。为此，如何对数据库模式结构进行合理的组织以保证数据库与应用程序之间的独立性并最终尽可能地减小软件维护代价，是数据库技术在实际应用中不得不考虑的一个问题。

以上这两点就是研究数据库系统体系结构的原因，总结就是：①由于 DBMS 的引入，必须从终端用户角度考虑合适的软件架构；②数据库的实际应用必须考虑数据库模式结构的合理组织，以尽可能地减小系统的软件维护代价。

从终端用户的视角看，数据库系统是一个软硬件系统，DBMS 只是其中的一个部件。DBMS 与前端的数据库应用程序一起为用户服务，满足用户的各种业务处理需求。按照这一视角，数据库系统的体系结构是由客户端(数据库应用程序)、服务器(数据库服务器、Web 服务器、应用服务器等)组成的一种架构。同时，由于数据存储方式、业务处理功能实现方式等方面存在着不同选择，数据库系统体系结构也存在多种方案。

从 DBMS 的视角看，数据库系统是以数据库为中心的系统，如何合理地设计数据库模式以及如何将数据库呈现给前端应用程序存在着多种选择。例如，可以直接将整个数据库完整地呈现给所有的用户，也可以为不同的用户封装并一次性呈现用户需要使用的部分数据。这些不同的设计也会直接影响到数据库应用程序的可维护性。这是指一旦数据库模式发生修改，如果不采取合理的数据库模式结构设计，那么很可能会导致前端的数据库应用程序不得不大批量地修改源代码，从而带来极大的软件维护工作量。后面将会看到，合理的数据库模式结构可以保证在数据库模式发生修改时只需要很少的工作量就可以保证前端的应用程序源代码不需要修改，实现数据库与应用程序之间较好的独立性。

2.2　数据库模式结构

本节首先讨论基于 DBMS 视角的数据库系统体系结构。在这种体系结构中，目前广泛采用的是 ANSI/SPARC 体系结构。

ANSI/SPARC 体系结构是 1975 年由美国国家标准学会（American National Standards Institute，ANSI）的计算机与信息处理委员会中的标准计划与需求委员会（Standards Planning and Requirements Committee，SPARC）提出的数据库模式结构。ANSI/SPARC 体系结构定义了标准的数据库模式结构。它不仅可以用来解释已有的商用 DBMS 的数据库模式结构，也可以作为研发新型 DBMS 时的数据库模式组织标准。目前，Oracle、MySQL、Microsoft SQL Server 等商用 DBMS 都遵循和支持 ANSI/SPARC 体系结构。

在详细讨论 ANSI/SPARC 体系结构之前，有必要先说明数据库模式和数据库实例的概念。数据库模式是数据库中全体数据的逻辑结构和特征的描述，它仅仅涉及类型的描述，不涉及具体的值。与数据库模式相对的另一个概念是数据库实例（Database Instance）。数据库实例是数据库模式的一个具体值。在数据库系统中，数据库模式反映了数据的结构及联系，而数据库实例反映的是某一时刻数据库的状态。一个数据库模式可以对应多个数据库实例（例如，不同时刻的数据库状态就对应了一系列的数据库实例）。另外，数据库模式设计完成后相对稳定，一般修改较少，而数据库实例通常是频繁变化的。

图 2-1 给出了数据库模式和数据库实例的一个示例。图中左边是一个教学信息管理系统中的数据库模式，而右边则显示了不同时刻的两个数据库实例。要注意的是，在实际的 DBMS 中，数据库模式的描述要比图 2-1 复杂得多，数据库实例所包含的数据也要多得多，这里只是一个示例而已，目的是使大家明白数据库模式与数据库实例之间的差别。

ANSI/SPARC 体系结构可以用一句话概括——三级模式结构+两级映像。三级模式结构是指数据库模式由外模式（External Schema）、概念模式（Conceptual Schema）、内模式（Internal Schema）来描述；两级映像是指外模式-模式映像和模式-内模式映像，它们定义了三级模式结构之间的相互关联。图 2-2 显示了 ANSI/SPARC 体系结构。

1．外模式

外模式，也称为用户模式（User Schema）或子模式（Sub Schema），它定义了视图层（View Level）的模式结构，它建立在概念模式之上，代表了单个用户所见到的局部数据库数据的逻辑结构。由于数据库系统中一般存在着多个用户，而不同用户对数据库的存取需求是不

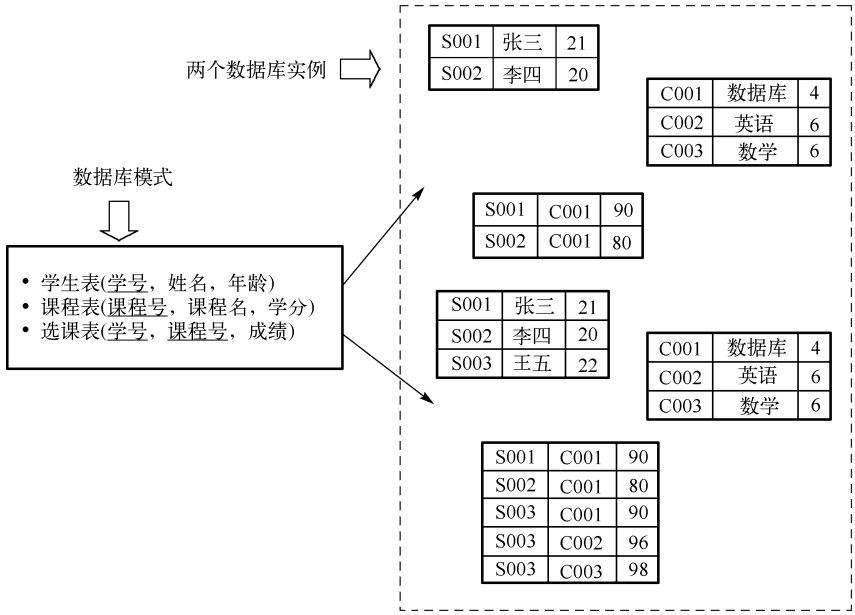


图 2-1　数据库模式与数据库实例示例

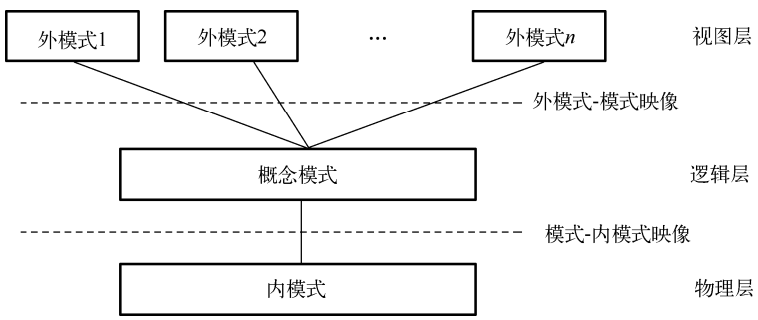


图 2-2　ANSI/SPARC 体系结构

同的，因此每一个用户都应当有自己对应的外模式，即一个概念模式之上可以定义多个外模式。注意，这里的用户实际对应着数据库应用程序，因为在实际系统中是由数据库应用程序来负责存取数据库中的数据的。在数据库系统中，外模式定义了用户与数据库系统之间的数据接口，对用户来说，他们所看到的数据库是由外模式定义的。外模式的引入使得同一个数据库可以给不同用户呈现不同的视图。举个例子，一个图书馆数据库中，借书者眼里的数据库内容与图书馆工作人员眼里的数据库内容可能完全不同——借书者眼里的图书只有图书名、作者、出版社、ISBN 等内容，而图书馆工作人员眼里的图书则还可能包含库存数、购买价格、购买单位等信息。外模式的实例称为外部视图(External View)。在 ANSI/SPARC 体系结构中规定外模式通过外模式 DDL 进行定义。

2. 概念模式

概念模式定义了逻辑层(Logical Level)的模式结构，它表示了整个数据库的逻辑结构，如数据记录由哪些数据项构成，数据项的名字、类型、取值范围，数据之间的联系，数据

的完整性等。概念模式不涉及数据物理存储的细节和硬件环境。一个数据库只有一个概念模式。概念模式的实例称为概念视图(Conceptual View)，它实际上就是特定时刻的整个数据库，是数据和值的集合。在 ANSI/SPARC 体系结构中规定概念模式通过模式 DDL 进行定义。DDL 是数据库语言的一类，它的主要功能是操纵数据库模式。在本书后面介绍 SQL 时将会对其进行详细介绍。另外，通常如果不特别说明，“模式”这一名词往往代表的是概念模式的含义。因此在许多时候也可以直接用“模式”来指代“概念模式”。

3. 内模式

内模式定义了物理层(Physical Level)的模式结构，它描述了数据库物理存储结构和存储方式。例如，数据库记录的存储方式是顺序存储、按 B 树存储还是散列存储，索引按什么方式组织，数据是否加密，数据是否压缩存储，等等。注意，内模式所定义的数据库存储结构是一种基于磁盘块的存储结构，也就是说，它是以磁盘块为基本单位来描述数据库存储结构的。例如，前面提到的 B 树、散列等指的都是磁盘块的组织方法。但是，内模式不涉及磁盘块的大小，也不考虑具体的磁盘物理参数(如柱面数)。因此，内模式与底层真正的存储硬件实际上是独立的。由于 DBMS 通常都建立在操作系统之上，因此这些与具体硬件打交道的工作实际上是由底层的操作系统负责的。与概念模式类似，一个数据库也只有一个内模式。内模式的实例称为内部视图(Internal View)。在 ANSI/SPARC 体系结构中规定内模式通过内模式 DDL 进行定义。

图 2-3 所示为教学信息管理系统中三级模式结构示例。假设在该系统中存在三类用户：学生、选课管理人员、课程评价管理人员。学生只能看到课程信息和自己的选课数据，选课管理人员可以看到课程信息和全部的选课数据，而课程评价管理人员则可以看到课程信息和课程评价数据。在图中为这三类用户分别定义了三个外模式。

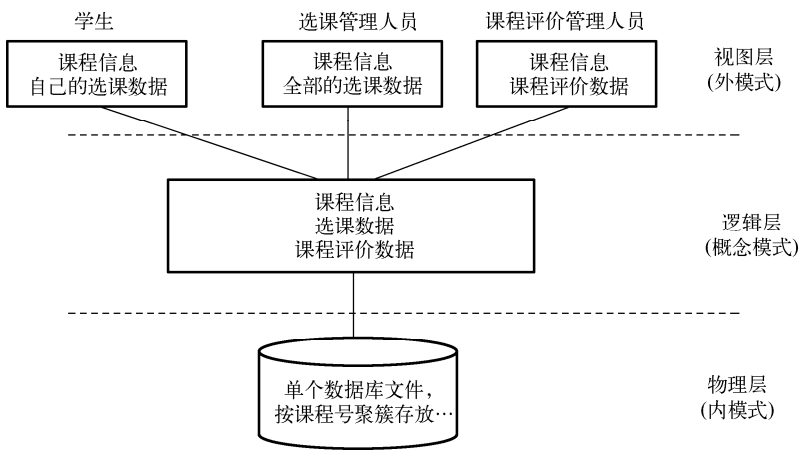


图 2-3　三级模式结构示例

4. 三级模式之间的映像关系

ANSI/SPARC 体系结构中的两级映像实现了三级模式结构间的联系和转换，用户可以逻辑地处理数据，不必关心数据的底层表示方式和存储方式。

外模式-模式映像定义了外模式与概念模式之间的联系。已经知道外模式是建立在概念模式之上的，那么究竟如何建立外模式？答案就是通过外模式-模式映像来建立。外模式-模式映像的引入使得用户可以使用不同于概念模式的属性名称来定义外模式，也可以使用一些属性运算从概念模式中得到外模式的属性。但外模式-模式映像最重要的价值在于它可以实现数据库的逻辑独立性——概念模式发生改变时，只需要修改外模式-模式映像，可保持外模式不变，从而保持用户应用程序不变，保证了数据库与应用程序的逻辑独立性。

模式-内模式映像定义了概念模式与内模式之间的联系。这一级映像的主要作用是保证数据库的物理数据独立性——当数据库的内部存储结构发生改变时，只需要修改模式-内模式映像，可保持概念模式不变，从而保持外模式以及用户应用程序不变，保证了数据库与应用程序之间的物理独立性。

举个例子，假设有概念模式 Employee(E#, D#, Name, Salary)，下面的语句定义了外模式 EMP(Emp, Dept, Name)，同时定义了外模式-模式映像。此处使用的是 SQL 语句，具体细节将在本书后面讨论。

```
Create View EMP(Emp,Dept,Name)
As
Select E# as Emp,D# as Dept,Name
From Employee
```

如果概念模式发生了变化，如属性 E#修改成了 Emp，则新的概念模式为 Employee(Emp, D#, Name, Salary)。此时可以执行下面的操作来修改外模式-模式映像：

```
Drop View EMP;
Create View EMP(Emp,Dept,Name)
As
Select Emp,D# as Dept,Name
From Employee
```

上述操作的含义是先用 Drop View 语句删除原先定义的外模式(即 SQL 中的 View)，然后重新创建一个新的外模式。

执行完这一操作后，应用程序眼里的外模式仍然为 EMP(Emp,Dept,Name)，没有任何变化，因此原先的源程序也不要做任何修改。这就是数据库的逻辑数据独立性。物理数据独立性的思想与此类似。

2.3　数据库应用系统体系结构

从终端用户的角度看，数据库系统体系结构从严格意义上讲是数据库应用系统的体系结构，因为它是从使用者(应用程序)角度来理解数据库系统的部件集成方式的。这类体系结构可以分为单用户结构、主从式结构、分布式结构、客户端/服务器结构、浏览器/服务器结构等。这些结构是随着计算机技术的发展而逐步提出的，目前最常见的是客户端/服务器结构和浏览器/服务器结构。

1. 单用户结构

单用户结构是早期的数据库应用系统体系结构。在这种结构中，整个数据库系统(应用程序、DBMS、数据库等)装在一台计算机上，为一个用户独占，不同机器之间不能共享数据。

在计算机网络技术出现之前，各个部门之间很难直接共享数据，所以单机应用是最常见的方式。但由于一个企业或组织内部的数据一般都需要被多个部门使用，因此单用户结构很容易造成数据冗余、不一致等问题。例如，一个企业的各个部门都使用本部门的机器来管理本部门的数据，各个部门的机器是独立的。由于不同部门之间不能共享数据，因此企业内部存在大量的冗余数据。例如，人事部门、会计部门、技术部门必须重复存放每一名职工的一些基本信息(职工号、姓名等)。

目前，单用户结构在企业应用中已经很少使用，在一些新型应用(如智能手机等)中还在继续使用。理论上讲，如果一个单一的数据库不需要多用户共享，则可以考虑采用单用户结构。

2. 主从式结构

主从式结构也是早期的一种数据库应用系统体系结构。在 20 世纪 90 年代之前，由于计算机的处理能力还比较弱，因此往往采用小型机、中型机，甚至大型机来作为数据存储和处理的主机，如果多个用户同时需要使用主机，则采取连接多个终端的方式来实现。终端往往只具有简单的输入/输出功能和非常弱的处理能力。主从式结构就是在这一环境下发展起来的数据库应用系统体系结构。在这一结构中，一个主机带多个终端，可以支持多用户的数据存取。数据库系统(包括应用程序、DBMS、数据库等)都集中存放在主机上，所有处理任务都由主机来完成，然后各个用户通过主机的终端并发地存取数据库，共享数据资源。

主从式结构的优点是易于管理、控制与维护。它的缺点主要有两点。

(1) 当终端用户数目增加到一定程度后，主机的任务会过分繁重，成为瓶颈，从而使系统性能下降。

(2) 系统的可靠性依赖主机，当主机出现故障时，整个系统都不能使用。这一问题常形象地称为“单点故障”。

在主从式结构中，客户端(终端)基本不具备数据处理能力。随着 PC 的逐步发展，客户端逐渐开始有数据存储、处理等能力，可以将系统的一部分负荷迁移到客户端上，从而使得传统的主从式结构逐步地淡出了应用领域。

3. 分布式结构

分布式结构是随着计算机网络技术的发展而产生的一种数据库应用系统体系结构。在现实生活中，有一些应用系统(如银行、连锁企业等)由于拥有分布在各地的多个分支机构，它们的数据同样是分布在各地的，但是逻辑上整个企业又是一个整体，有的业务需要建立在整个企业的全局数据上，有的业务只需要使用某个分支机构的局部数据。以银行为例，当开设银行账户时，往往只需要访问开户行本地的银行数据库，而当执行异地取款、转账

等操作时则需要访问整个银行的全局数据。分布式结构的产生很好地满足了这类物理上分布的企业的需求。

在分布式结构中，数据库中的数据在逻辑上是一个整体，但在物理上分布在计算机网络连接的不同结点上。“物理上分布，逻辑上一体”是分布式结构的最主要特征。网络中的每个结点都可以独立处理本地数据库中的数据，执行局部应用，也可以同时存取和处理多个异地数据库中的数据，执行全局应用。要注意的是，简单地将多个单机的数据库系统通过网络相连接并不能称为分布式结构，因为虽然它们在物理上分布，但在逻辑上不是一个整体(在用户眼里不是一个单一的数据库)，用户不能透明地存取所有的数据。图 2-4 给出了分布式结构的一个示例。在这个示例中，整个图书数据库在物理上分布在三个校区，但在逻辑上是一个整体，在用户眼里仿佛只有一个数据库。

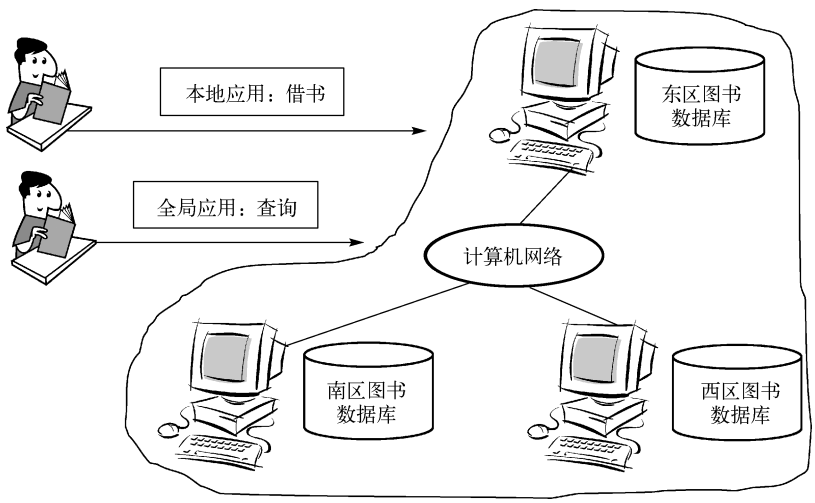


图 2-4　分布式结构示例

分布式结构的优点是适应了地理上分散的企业或组织对于数据库应用的需求。其缺点主要有两点。

(1)数据的分布存放给数据的处理、管理与维护带来困难。

(2)当用户需要经常访问远程数据时，系统效率会明显地受到网络传输的制约。

分布式结构需要 DBMS 有专门进行数据划分、数据分配的技术，另外，由于数据有可能复制在各个结点中，因此如何维护数据的一致性是一个关键的问题。分布式结构由于对各个分布的结点的处理能力的要求较低，在 20 世纪 90 年代得到了广泛应用，但由于分布式结构在维护方面存在的问题，近年来大型企业应用又开始逐步从分布式回归到了集中式(即下面要介绍的客户端/服务器结构)，通过建设大规模数据中心，将企业的数据集中存储在一个位置。

4. 客户端/服务器结构

客户端/服务器(Client/Server，C/S)结构是 20 世纪 90 年代随着个人计算机(PC)的快速发展而产生的一种数据库应用系统体系结构。随着 PC 处理能力的提高，可以将应用系统的业务处理转而放到 PC 上(这与主从式结构不同)，而服务器则提供数据存储和管理等服务。

在客户端/服务器结构的数据库系统中，存在两类结点(图 2-5)：一类称为数据库服务器，主要提供 DBMS 的功能；另一类称为客户端，主要运行应用程序以及一些前端的数据库开发工具。这两类结点通过计算机网络(一般是局域网)相互连接，用户操作客户端上的应用程序进行业务处理，如果需要访问数据库，则通过网络访问数据库服务器。

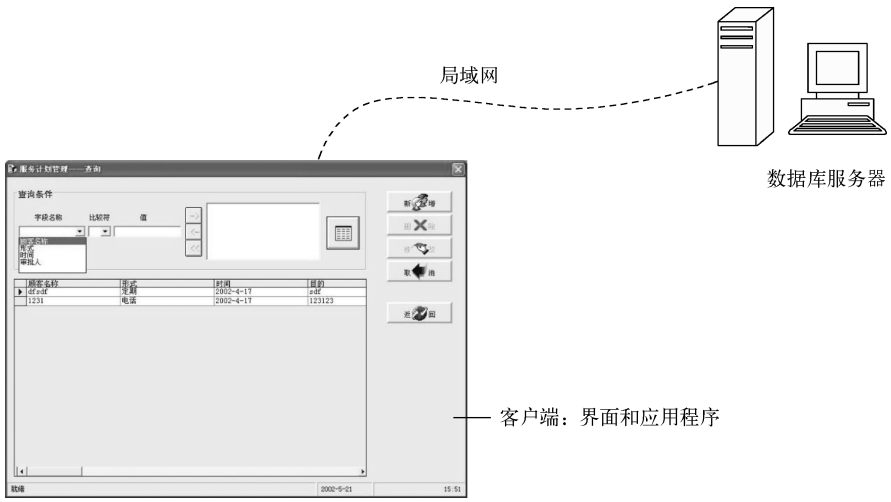


图 2-5　客户端/服务器结构示例

在最早提出的客户端/服务器结构中，一般包含一台数据库服务器和多台客户端，这种最原始的结构现在称为集中式 C/S 结构。集中式 C/S 结构又可以细分为两种：以前端为主的集中式 C/S 结构和以后端为主的集中式 C/S 结构。以前端为主的集中式 C/S 结构强调把所有的业务处理功能放在客户端，服务器只负责存储和维护数据。由于服务器通常具有更高的配置和处理能力，因此研究者希望能够将部分业务处理任务放到服务器上执行，由此提出了以后端为主的集中式 C/S 结构。在这种结构中，服务器不仅负责数据的存储与维护，还承担一部分的业务处理任务。当然，这需要 DBMS 提供一定的编程能力。目前，大部分的商用 DBMS(如 Oracle、Microsoft SQL Server、Sybase 等)都支持服务器的编程。业务处理程序在服务器中需要使用 DBMS 认可的专门的过程化 SQL 来编写，编译好后存放在数据库中，然后前端的客户端应用程序可以在需要时调用它们。目前，在实际较大规模的应用中，以后端为主的集中式 C/S 结构更为普遍一些。

客户端/服务器结构还可以和分布式结构结合形成分布式 C/S 结构。在分布式 C/S 结构中，存在着多台分布的数据库服务器和多台客户端。客户端透明地访问多台分布的数据库服务器，就好像只有一个服务器一样，也就是说，从应用程序这一端是感知不到后台数据库服务器结构的这种变化的。这一点也正是分布式结构的主要特点——逻辑上一体。分布式 C/S 结构需要 DBMS 具有相应的分布式数据管理的功能，一般要求 DBMS 是一个分布式 DBMS。目前商用的 DBMS 通常通过提供需要另外购买的分布式扩展模块来支持分布式结构。

客户端/服务器结构的主要优点如下。

(1) 客户端的用户请求被传送到数据库服务器，数据库服务器进行处理后，只将结果返回给用户，从而显著减少了数据传输量。

(2) 客户端与服务器一般都能在多种不同的硬件和软件平台上运行。

(3) 可以使用不同厂商的数据库应用开发工具。

在 Web 开发技术出现之前，客户端/服务器结构是最流行的数据库应用系统体系结构，主流的软件开发平台也都以 C/S 开发为主，如 VC++、VB、Delphi、PowerBuilder 等。进入 21 世纪后，随着 Web 开发技术的流行，浏览器/服务器结构开始流行，越来越多的 Web 开发平台开始出现，基本形成了 Web 开发与传统 C/S 开发半分江山的局面。尽管如此，客户端/服务器结构仍具有其独特的优点。就目前而言，其是一种主要的数据库应用系统体系结构，在银行、证券、企业内部管理等有区域性应用需求的系统中广泛使用。

客户端/服务器结构的主要缺点如下。

(1) 系统安装维护复杂，工作量大。每台客户端都需要安装专门开发的应用程序(银行柜台里面的应用程序都是需要人工安装的)。

(2) 相同的应用程序要重复安装在每一台客户端上，从系统总体来看，大大浪费了系统资源。

5. 浏览器/服务器结构

客户端/服务器(Browser/Server，B/S)结构的不足在浏览器/服务器结构出现之后得到了显著改善。浏览器/服务器结构可以看成 Web 时代的客户端/服务器结构。两者的不同之处在于，在浏览器/服务器结构中，客户端应用程序的界面是统一的，即浏览器。此外，浏览器/服务器结构的应用系统通常运行在 Internet 之上(当然理论上也可以只在局域网内使用，不过要求支持 TCP/IP，即 Intranet)。

由于要支持浏览器，所以在浏览器/服务器结构中，服务器除了数据库服务器外，增加了 Web 服务器。Web 服务器相当于一个网页的容器，首先客户端的浏览器向 Web 服务器发送请求，然后 Web 服务器根据需要访问数据库服务器，得到结果并形成网页发送给客户端，并且显示在客户端的浏览器上。图 2-6 给出了这种结构的示例。

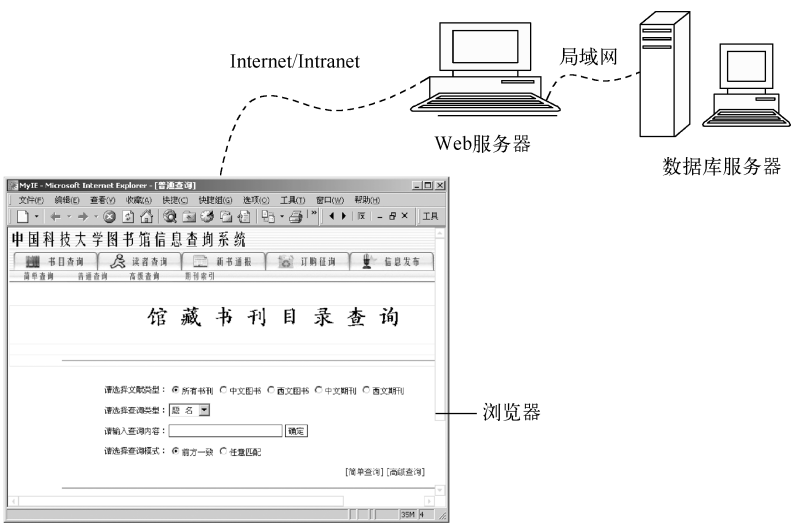


图 2-6　浏览器/服务器结构示例

浏览器/服务器结构可以看成一种适合互联网应用的客户端/服务器结构。与客户端/服务器结构相比，其主要优点如下。

(1) 统一的客户端界面，减少了应用安装和维护的工作量。在浏览器/服务器结构中，客户端只要有浏览器即可(目前 Windows 本身即包含了 IE 浏览器)，不需要安装应用程序。应用程序升级时，也只需要在 Web 服务器中升级，因此大大减小了系统维护的工作量。

(2) 基于 Web 技术，支持互联网应用。互联网应用的优点是可以跨地域运行，传统的 C/S 应用一般只能局限在局域网中。因此，对于像电子商务系统、网上银行等应用，浏览器/服务器结构具有明显的优势。

但浏览器/服务器结构也存在一定的缺点。

(1) 安全性问题，用户访问无地域限制。相比之下，由于 C/S 结构的应用只运行在局域网连接的系统内部，通常是一个部门或者一栋大楼，其用户类型、访问来源、访问数量等都很容易控制，安全性要高很多。

(2) 开发工具的能力相对较弱。

2.4　本 章 小 结

本章主要介绍了数据库系统体系结构的相关概念，包括数据库系统体系结构的研究缘起、ANSI/SPARC 三级模式结构以及基于软件系统视角的数据库应用系统体系结构。本章的重点是 ANSI/SPARC 三级模式结构的概念、含义以及背后隐含的不同级模式之间的映像关系。三级模式结构之间的两级映像保证了数据库与应用程序之间的逻辑独立性和物理独立性，对于提高数据库应用系统的软件可维护性有着重要的意义。

通过对本章的学习，读者应能够清晰区分数据库三级模式的基本概念，并对两类数据独立性有深入的理解，同时对数据库应用系统常用的一些软件体系结构有所了解。

习　　题

1．什么是 ANSI/SPARC 体系结构?

2．举例说明逻辑数据独立性和物理数据独立性。

第 3 章　关系数据模型

数据模型旨在描述现实世界中的实体、实体间的联系以及语义约束。对于数据库，迄今为止最流行也最著名的数据模型是关系数据模型。关系数据模型以其简洁的数据结构和数据操作闻名于世，大大促进了数据库学科的发展。深入了解和理解关系数据模型，是掌握数据库技术的前提。它不仅有利于学习 SQL，还有利于掌握 Microsoft SQL Server 等以关系数据模型为基础的 DBMS，也有利于正确地研究和评价新型的数据库技术。从某种意义上讲，关系数据模型已经成为数据库领域的一个标杆。

内容提要：本章首先介绍数据模型的概述，然后着重介绍关系数据模型的形式化定义、关系数据模型的完整性约束以及关系代数。

3.1　数据模型概述

模型是对现实世界特征的抽象。数据模型也是一种模型，只不过它关心的是现实世界的数据特征。从不同角度去抽象现实世界的数据特征可以得到不同类型的数据模型。为了深入理解数据模型，从多个方面对数据模型进行讨论，包括数据模型的定义、分类以及形式化定义等。

3.1.1　数据模型的定义

模型的概念在实际生活中很常见，如建筑模型、沙盘模型等。通常，模型是对现实世界特征的抽象，例如，建筑模型是对建筑物的特征抽象。

与其他模型不同，数据模型是对现实世界数据特征的抽象，如数据的组成、数据之间的联系等。要注意的是，现实世界中的实体不仅具有数据特征，还具有一些其他特征，如行为特征等。但对于数据模型而言，它只关心实体的数据特征。举个例子，“学生”是现实世界中的一个实体，数据模型关心的是“学生”这个实体由哪些属性来描述(如姓名、学号等)、它与其他实体之间有何联系(例如，“学生”与“课程”之间存在着选课关系)等内容，而不关心“学生”实体会有哪些行为以及如何完成这些行为的细节(例如，“学生”是如何进行选课、如何进行考试的等)。

定义 3-1　数据模型是描述现实世界实体、实体之间的联系以及语义约束的模型。

数据模型的定义明确指出，一个数据模型需要描述现实世界三方面的数据特征：①现实世界存在哪些实体；②这些实体之间存在什么样的联系；③现实世界的实体以及联系表达上具有什么样的语义约束。

3.1.2　数据模型的分类

根据对现实世界数据进行抽象的不同层次，数据模型一般分为两类：概念数据模型和结构数据模型。图 3-1 显示了概念数据模型和结构数据模型之间的关系。

概念数据模型(Conceptual Data Model)又称为语义数据模型(Semantic Data Model)，或者简称概念模型或语义模型。概念数据模型强调从用户的角度来描述现实世界的数据特征，着重于对实际数据需求的获取和表达。在建立概念数据模型时通常不考虑该模型的具体实现细节，因此概念数据模型与具体的计算机系统以及 DBMS 无关。在实际应用中，通常用概念数据模型来表达数据库应用系统的数据需求。

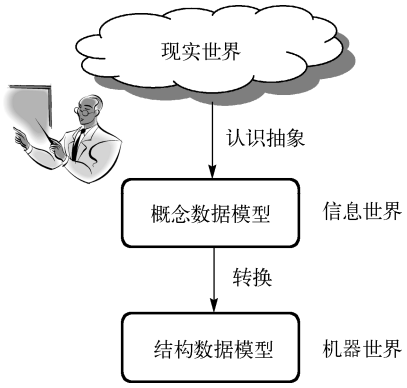


图 3-1　数据抽象的不同层次

结构数据模型强调从计算机系统角度来进行数据建模。它直接面向数据库逻辑结构，因此必须考虑模型的实现细节，例如，将来在什么样的 DBMS 上实现。正是因为这一点，结构数据模型具有严格的形式化定义，包括数据结构、数据操作、数据约束等。结构数据模型是 DBMS 的逻辑基础。任何一个 DBMS 都是基于某种特定的结构数据模型的，可以说，一个 DBMS 就是一个结构数据模型的计算机实现。一般地，在没有特指的情况下，“数据模型”这一名词指的都是结构数据模型。在本书后面的内容中，如果没有特别说明，数据模型都是指结构数据模型。

为什么在数据库领域中要引入概念数据模型和结构数据模型两种类型的数据模型？数据库技术最基本的目标是将现实应用中的数据用一种合理的结构存储到数据库系统中，但是，如果应用中的数据非常复杂，很难直接得到这种合理的结构。为此，引入了一个中间层——概念数据模型，从而可以先将现实世界的数据特征表达为概念数据模型，然后通过特定的方法将概念数据模型转换为面向 DBMS 的结构数据模型。这样就解决了实际应用中的数据建模和数据库建立问题。

以银行存款应用为例，对应图 3-1 的现实世界就是客户存款的需求，信息世界对应了客户存款相关数据的概念模型。图 3-2 给出了对应银行存款应用的一个概念模型，此处以 ER(Entity-Relationship，实体联系)模型为例。图 3-2 所示的概念模型只包含了一些简单的图形符号，例如，矩形框表示了现实世界中的实体，菱形框表示了实体之间的联系(两端的 *M* 和 *N* 表示了联系的类型是多对多)，椭圆框表示了实体和联系的属性。可以看到，概念模型中并没有涉及任何与计算机系统实现相关的内容，与未来在 Oracle 还是其他 DBMS 上实现无关。这么做的目的是便于和用户进行交流。因为概念模型是对现实世界数据需求的获取和抽象，而用户是最了解数据需求的一方，所以必须保证没有计算机和数据库知识的普通用户也能够看得懂概念模型，这样才能进一步进行交流讨论，最终准确、全面地获取应用的数据需求。

但是，图 3-2 所示的概念模型并不能直接在 Oracle 等 DBMS 中实现。为此，需要将其转换为机器世界中的结构数据模型。图 3-3 给出了与图 3-2 概念模型对应的一个关系模型示例。在概念模型中只有两个实体：客户和账户，但在关系模型中有三个关系模式。因为关系模型有严格的定义，并且可以在 Oracle、MySQL 等关系 DBMS 上实现，所以通过概念模型到关系模型的转换，最终完成了现实世界的数据需求在计算机系统中的实现。

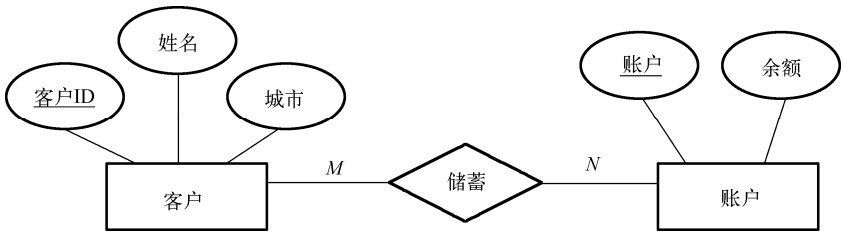


图 3-2　对应银行存款应用的概念模型

客户 ID	姓名	地址
001	张三	北京
002	李四	上海
003	王五	合肥

(a) 客户

账户	余额
1001	2234
2002	5678
3003	9101

(b) 账户

客户 ID	账户
001	1001
002	2002
003	3003

(c) 储蓄

图 3-3　对应图 3-2 的关系模型

3.1.3　数据模型的形式化定义

由于(结构)数据模型面向 DBMS，要求有严格的形式化定义。因此，下面给出一种数据模型的形式化定义方法。

数据模型的形式化定义包括三个要素。

(1) 数据结构：数据模型中用来表示现实世界实体以及实体间联系的数据结构。数据结构的定义规定了数据库数据的逻辑结构。

(2) 数据操作：数据模型中包含的用于操作数据结构的操作集合。因此，数据操作的定义规定了将来数据库中的数据存取方式，如查询、更新等。

(3) 数据约束：数据模型中建立数据结构和执行数据操作时必须遵循的约束条件。由于数据模型要尽可能真实地反映现实世界中数据的真实特征，而实际应用中的数据通常具有各种各样的约束，因此，数据约束的定义使得数据模型可以有效地表达真实的数据特征，满足实际应用的要求。

数据模型的形式化定义一方面有助于理解后面讨论的关系数据模型，另一方面如果未来提出一种新的数据模型，也可以参考这一形式化定义，从数据结构、数据操作、数据约束三方面加以定义。例如，假设需要定义一个针对时空数据(随时间而变化的空间数据)的数据模型，可以从以下三个方面进行定义。

(1) 时空数据结构：现实世界中的时空数据用什么数据结构表示。

(2) 时空数据操作：时空数据如何进行查询和更新。

(3) 时空数据约束：时空数据在表示和操作过程应满足什么样的数据约束。

3.2　关系数据模型概述

关系数据模型(也可以简称关系模型)涉及了许多新的概念和技术，为了全面地掌握关系数据模型，应从数据模型的定义以及三要素(数据结构、数据操作、数据约束)来理解关系数据模型。

3.2.1　关系数据模型的定义

回顾数据模型的定义：描述现实世界实体、实体间的联系和语义约束的模型。可以参照数据模型的定义，给出关系数据模型的定义。

定义 3-2　关系数据模型(Relational Data Model)是以规范化的二维表格结构表示实体集，以外码表示实体间联系，以三类完整性表示语义约束的数据模型。

图 3-4 给出了一个关系数据模型的示例。在关系数据模型中，所有的实体都表示在一个二维表格结构中，每一个实体表示为表格中的一行，称为一个元组(Tuple)。所有元组的集合构成了一个关系(Relation)。表格的表头给出了所有元组的语义，因此它代表了整个关系的模式，在关系数据模型中称为关系模式(Relational Schema)。表格的每一列称为一个属性(Attribute)，元组的每一个分量称为一个属性值。

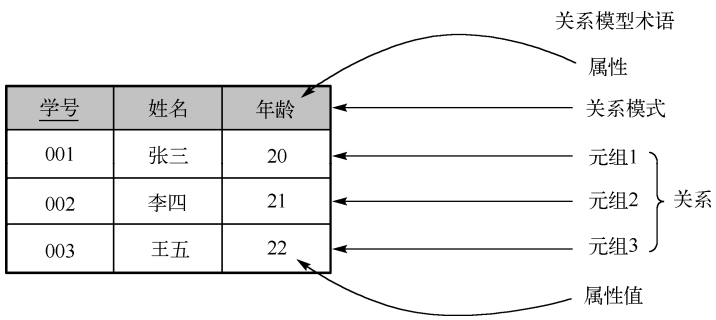


图 3-4　关系数据模型示例

下面给出关系数据模型涉及的一些术语的定义，包括元组、属性、关系、关系模式、关系数据库模式、关系数据库、码等。

1. 元组

二维表格的每一行称为一个元组，它是一个属性值的集合。元组的数目称为关系的基数(Cardinality)。关系数据模型中的元组代表了现实世界中可唯一区分的实体，知道这一点有助于理解关系数据模型的基本性质。

元组在本质上是数据，是值的集合，确切地说，是一系列属性值的集合。

2. 属性

二维表格的每一列称为一个属性。属性有一个属性名以及相应的域(Domain)。域是一组具有相同数据类型的值的集合，它表示了属性取值的范围。属性的数目称为关系模式的度(Degree)。

3. 关系

在关系数据模型中，关系是元组的集合。因此，关系代表了一个实体的集合。与元组一样，关系表达的也是数据的概念，是一个值的集合。

从形式上看，关系是二维表格中除了表头部分的数据行集合。关系是关系数据模型中表示和组织数据的唯一形式。这一结构与网状数据模型的网络结构、层次数据模型的层次结构以及面向对象数据模型的对象结构相比要简单得多。需要说明的是，虽然面向对象数据模型中的基本数据结构对象与关系数据模型中的元组类似，但对象之间存在继承、聚合、引用等复杂联系，因此要比关系数据模型复杂许多。

4. 关系模式

数据和语义是不可分的，脱离了语义的数据是没有实际意义的。关系的语义通过关系模式来定义。关系模式描述了数据关系的逻辑结构和特征，从形式上看它对应了二维表格的表头。

关系模式在关系数据模型中有着严格的定义(将在 3.2.3 节中详细讨论)。在某些情况下，关系模式可以简化表示为一个属性集，例如，学生关系模式可简化表示为 Student(sno, name, age, gender)。

5. 关系数据库模式

关系数据库模式描述了整个关系数据库的逻辑结构和特征。关系数据库模式是一个由若干关系模式构成的集合，其中每一个关系模式都描述了某一类实体的逻辑结构和特征。

因此，如果要针对某个特定应用设计它的关系数据库模式，就需要将该应用所涉及的所有实体的关系模式分别设计出来。所有的关系模式设计完成后，就自然而然地得到了整个关系数据库模式。本书后面的数据库设计部分就遵循了这样的思路。

6. 关系数据库

关系数据库模式的一个实例称为关系数据库，所以关系数据库对应的是数据的概念，是关系的集合。

7. 码

关系数据模型中存在几个码的概念。首先是超码(Super Key)。超码是关系模式中能够唯一区分每个元组的属性集。其次是候选码(Candidate Key)。不含多余属性的超码称为候选码，所以候选码是唯一区分元组的最小属性集。一个关系模式中有可能存在多个候选码，例如，一个学生关系模式中，学号和身份证号都可以唯一区分学生，因此都是候选码。把包含在某个候选码中的属性称为关系模式的主属性(Primary Attribute)，把主属性之外的属性称为关系模式的非主属性(Nonprime Attribute)。主属性和非主属性的概念在数据库模式设计中将会用到，此处不再展开讨论。最后是主码(Primary Key)。在实际的数据库设计中，用户选定作为元组标识的候选码称为主码，其他的候选码则称为替换码(Alternate Key)。替换码在实际应用中很少使用，此处只需要知道这一概念即可。如果一个关系模式只存在

着一个候选码，则此唯一的候选码必定成为主码。主码在本章的图中以属性名下方加下画线表示。

例如，假设有学生关系模式 Student(sno, name, libraryID, age, gender)，其中 sno 是学号，libraryID 是借书证号，则(sno, name)和(libraryID, age)都是超码。当然这里还有许多其他超码，就不一一罗列了。候选码有两个，即 sno 和 libraryID，这两个候选码都可以唯一标识元组，并且不含多余属性。如果用户选择 sno 作为该关系模式的主码，则 libraryID 是替换码；如果选择 libraryID 作为主码，则 sno 是替换码。

3.2.2　关系的基本性质

关系数据模型是以二维表格形式的关系为基本数据结构表示现实世界中的实体的，但关系并不是一般的二维表格，它必须满足一定的规范要求，因此称关系是规范化的二维表格。在关系数据模型中，二维表格的这些规范表现为关系的几个基本性质。

(1)属性值不可分解。

属性值不可分解是指每个属性值都是单一的值，不能是一个值集，通俗地讲，就是不允许关系表中有表。如果允许关系的属性值是一个值集，则会出现更新二义性问题。如图 3-5 所示的选课关系，假设属性值可以是值集，如果现在学号为“002”的学生也选修 C 语言，则 DBMS 在执行这样一次更新时将面临两种方案：一是将第一行元组的“课程”值修改为{数据库, C 语言}，二是将第二行元组的“学号”修改为{001, 002}。这将导致 DBMS 无法继续执行。

学号	课程
001	数据库
002	{数据库, C 语言}
003	离散数学

图 3-5　选课关系

(2)元组不可重复。

元组不可重复是指任一关系中都不允许存在重复元组。这也意味着任何一个关系模式必定存在至少一个超码(或候选码)。极端情况下，关系模式的整个属性集肯定可以唯一标识元组，因此是关系模式的超码。

为什么关系数据模型要求元组不可重复？这是因为关系数据模型的基本思路是用元组来表示现实世界中的实体，而现实世界中的实体都是可唯一区分的。这是它的建模基础。

(3)关系没有行序。

关系没有行序是指任何关系的元组之间没有顺序，因此颠倒某一关系中的元组顺序并不会产生一个新的关系——两者之间是等价的。

关系没有行序这一性质的根源在于：关系是元组的集合，也就是说，关系数据模型是以集合论为基础来表示关系的。集合中的元素是没有顺序的，因此，关系作为元组的集合，其元素当然也是没有顺序要求的。

(4)关系没有列序。

关系没有列序是指任一关系的属性之间没有顺序，因此可以任意变动一个关系的属性顺序而不会使该关系有任何改变。

关系没有列序的根源仍在于关系数据模型基于集合论这一事实。由于一个元组是属性值的一个集合，因此属性值之间的顺序可以随意变换而不会改变集合的值。

3.2.3　关系模式的形式化定义

关系模式描述了关系的逻辑结构和特征。如前所述，关系模式对应着二维表格的表头，它给出了关系中每一个元组以及属性值的语义。在关系数据模型中，关系模式有着严格的形式化定义，它也是关系数据库模式设计的理论基础。

定义 3-3　关系模式可以形式化定义为一个四元组 $R(U, D, \text{Dom}, F)$，其中 R 是关系模式名，U 是一个属性集，D 是 U 中属性的值所来自的域，Dom 是属性向域的映射集合，F 是属性间的数据依赖集。

例如，一个学生关系模式可以形式化定义为 Student(U, D, Dom, F)，其中，

$$U = \{\text{sno, name, age}\}$$

$$D = \{\text{CHAR, INT}\}$$

$$\text{Dom} = \{\text{dom(sno)} = \text{dom(name)} = \text{CHAR,dom(age)} = \text{INT}\}$$

$$F = \{\text{sno}\rightarrow\text{name, sno}\rightarrow\text{age}\}$$

其中，Student 关系模式的数据依赖集表示为 F，它包含了两个函数依赖(函数依赖的概念将在后面介绍)，反映了 Student 的三个属性之间的数据依赖关系。通过关系模式的形式化定义可以很好地描述关系的逻辑结构和特征。一般情况下，如果不需要对关系模式的数据依赖进行处理，则可以将关系模式简写为 $R(U)$，或者 $R(A_1, A_2, \cdots, A_n)$，其中 $U = \{A_1, A_2, \cdots, A_n\}$。例如，Student 关系模式可以简写为 Student(sno, name, age)。

3.3　关系数据模型的形式化定义

关系数据模型的概念以及特点可以通过数据模型的三要素来掌握，即数据结构、数据操作和数据约束。掌握了关系数据模型的这三个要素，就能够很好地理解关系数据模型。因此，本节从数据结构、数据操作和数据约束三个方面给出关系数据模型的形式化定义。

关系数据模型的形式化定义如下。

(1) 数据结构。关系数据模型只有唯一的数据结构，即关系。关系数据库中的全部数据及数据间的联系都以关系来表示。

(2) 数据操作。关系数据模型中，所有数据(即关系)通过关系运算来操作。关系运算有两种类型：关系代数和关系演算。关系代数是以集合操作为基础的，而关系演算是以谓词演算为基础的。关系演算又有两种类型：元组关系演算和域关系演算。元组关系演算以元组为变量，而域关系演算以属性为变量。目前已经证明，元组关系演算和域关系演算是等价的，而关系代数与关系演算也是等价的。在本书中，将着重讨论关系代数。

(3) 数据约束。在关系数据模型中，数据约束通过三类完整性约束来表达。在关系数据模型中，定义了实体完整性、参照完整性和用户自定义完整性三类完整性约束，它们可以很好地表示在关系数据模型中实体和实体间的联系表达时应遵循的语义约束。

3.4　关系数据模型的完整性约束

关系数据模型通过三类完整性约束来表达数据的语义约束。完整性约束(Integral Constraint)也称为完整性规则(Integral Rule)，它是关系模式应满足的一些谓词条件。通过三类完整性约束，关系数据模型可以表达丰富的语义约束条件。

3.4.1　实体完整性

定义 3-4　实体完整性(Entity Integrity)是指关系模式 *R* 的任一关系的主属性值不可为空。

注意，实体完整性要求关系模式的任一实例的所有主属性均不可取空值，而不仅仅是主码不为空。

图 3-6 给出了一个实体完整性的示例。图中显示了一个选课关系，它的唯一候选码也是主码，为“学号, 课程号”，因此主属性为学号和课程号。实体完整性要求关系的每个元组的主属性都不能为空。图 3-6 中的第 2 个元组和第 3 个元组都违背了实体完整性。

学号	课程号	成绩	
001	C001	90	
002		80	违背实体完整性
		85	

图 3-6　实体完整性示例

为什么关系数据模型要求满足实体完整性？可以以图 3-6 为例进行说明。关系数据模型的基本思路是用元组来表示实体，那么一个实体之所以可以称为实体，是因为它在现实世界中是可以唯一通过其码来标识的。因此，当在关系数据模型中插入一个实体(元组)时，要求其主属性值都不能为空，因为一旦为空，就无法将其对应到现实世界中的某个实体。此外，如果主属性值为空，那么关系数据模型所表达的语义也不符合现实世界的真实情况。例如，图 3-6 中第 2 个元组的课程号为空，如果允许这样的元组进入数据库，则意味着某个学生(即 002)没有选修任何一门课程也可以有课程成绩，显然这是不符合实际情况的。

建立数据库的目的就是将现实世界中的真实情况表达到计算机系统中。如果存储在数据库中的数据与现实世界的真实情况矛盾，就会大大降低数据库的可用性，因为对用户来说，这些数据都是虚假的数据。实体完整性为数据库系统提供了一种保证实体可标识性的有效手段。目前商用的 DBMS 都支持实体完整性的定义和实现。

3.4.2　参照完整性

参照完整性定义在两个关系模式之上。要定义参照完整性，首先需要了解外码的概念。回顾关系数据模型的定义，它是以外码来表示实体间的联系。

定义 3-5　关系模式 *R* 的外码(Foreign Key)是它的一个属性集 FK，满足：

(1)存在带有候选码 CK 的关系模式 *S*。

(2)*R* 的任一非空 FK 值都在 *S* 的 CK 中有一个相同的值。

把 *S* 称为被参照关系(Referenced Relation)，*R* 称为参照关系(Referential Relation)。

图 3-7 给出了一个外码的示例。图中有两个关系：学生关系和选课关系，其中选课关系中的属性“学号”是引用学生关系中属性“学号”的外码。

被参照关系*S*：学生关系

学号	姓名	年龄
001	张三	19
002	李四	20
003	王五	19

参照关系*R*：选课关系

学号	课程号	成绩
001	C001	90
002	C001	80
003	C002	85

图 3-7　关系模式的外码示例

外码的定义指出外码的任何非空取值必须存在于它所引用的被参照关系的属性中。例如，图 3-7 中，选课关系的外码“学号”的非空取值必须出现在学生关系的被引用属性“学号”中。

下面给出参照完整性的定义。

定义 3-6　关系模式 *R* 的参照完整性(Referential Integrity)是指 *R* 的任一个外码值必须等于被参照关系 *S* 中所参照的候选码的某个值，或者为空。

参照完整性完全是由关系模式的外码来定义的。参照完整性要求关系模式的外码要么为空(前提是外码不是主属性，因为要满足实体完整性)，要么等于被参照关系中所参照的某个属性值。

以图 3-8 为例说明参照完整性的作用。假设有三个关系：学生关系、选课关系、专业关系，其中选课关系中的外码“学号”参照了学生关系中的主码“学号”，学生关系中的属性“专业号”参照了专业关系的主码“专业号”。

专业关系

专业号	专业名称	学科类别
P001	自动化	1
P002	计算机	2
P003	企业管理	3

学号	课程号	成绩
001	C001	90
002	C001	80
003	C002	85

选课关系

学号	姓名	专业号
001	张三	P001
002	李四	
003	王五	

学生关系

图 3-8　参照完整性示例

如果把选课关系中插入一个新元组{004, C001, 80}，则违背了参照完整性，操作将被 DBMS 拒绝。原因是 004 这一学号没有出现在所参照的“学号”属性值中。将选课

关系的第 3 个元组的学号从 003 改为 004 同样也违背参照完整性。学生关系中，如果要修改 002 学生的专业号，则其值只能是 P001、P002 或者 P003，即必须源自所参照的专业关系的“专业号”。另外，学生关系的外码“专业号”可以为空，表示学生还未确定专业。

参照完整性与实体完整性一样都是为了保证数据库中的数据与现实世界真实情况一致。为什么必须要求选课关系中的“学号”来自学生关系的“学号”属性？实际上，这是因为在实际应用中，必须是本校注册的学生(出现在学生关系中)才能够选课。外码“学号”所定义的参照完整性保证了选课关系中的所有学号都是正式注册的学生。如果没有参照完整性，则选课关系中出现不存在于学生关系中的学号(即非本校学生)，DBMS 将难以发现此类错误，从而导致选课数据与真实世界中的值不一致。至于外码为什么允许为空，可以看图 3-8 学生关系中的外码“专业号”。在实际应用中，学生刚入学时往往是大类招生，不分具体专业，到了大学二年级或三年级才开始选专业，因此学生关系的外码“专业号”应允许为空，表示目前学生还没确定专业。但是，一旦学生确定了专业，则要求专业号必须源自专业关系，也就是说，只能是本校有的专业。因此，参照完整性的实施切实地保证了实体与实体之间的联系应满足的语义约束。

3.4.3　用户自定义完整性

实体完整性和参照完整性给出了针对主码与外码的语义约束，但实际应用还常常要求对一些非码属性添加完整性约束条件。因此，在关系数据模型中，引入了第三类完整性约束，即用户自定义完整性，来表达这类根据应用环境要求而设定的语义约束。

定义 3-7　用户自定义完整性(User-Defined Integrity)是指关系模式针对某一具体数据的要求而制定的约束条件，反映某一具体应用所涉及的数据必须满足的特殊语义。

用户自定义完整性通常以不等式、等式等给出，并且可以通过逻辑操作符连接多个谓词条件。例如，对于选课关系中的“成绩”，某一应用环境可能要求“成绩”取值为[0, 100]，在关系数据模型中可以使用“成绩> = 0 and 成绩< = 100”这样的表达式来定义完整性约束；而在另一应用中，成绩可能采用 5 分制，因此要求成绩取值只能是{1, 2, 3, 4, 5}，同样可以使用“成绩 IN{1, 2, 3, 4, 5}”这样的表达式来定义完整性约束。

3.5　关 系 代 数

关系代数是关系数据模型数据操作的主要实现方式。在关系数据模型中，所有数据都表示为关系，而关系代数则是实现对关系的增、删、改、查等操作的一个操作集合。关系代数通过关系代数表达式来表达用户对关系的操作需求。

3.5.1　关系代数的概念

代数包含一个类型集以及该类型集上的一个操作集合。关系代数首先是一种代数，它的类型集只包含一种类型——关系，而它的操作集合则包含了若干操作，称为关系代数操作。由于关系代数只涉及一种类型，因此在数据库领域常常忽略关系代数的类型集，而直接用关系代数指代关系代数操作。本书的后续内容中，如果不特别指出，关系代数指的就

是关系代数操作。关系代数操作是以关系为运算对象的一组操作集合。由于关系是一个元组的集合，因此关系代数操作是以并、交、差等集合操作为基础的。

关系代数操作可分为两种类型：一元操作(Unary Operation)是指只有一个运算对象的操作，而二元操作(Binary Operation)是指有两个运算对象的操作，传统的并、交、差都是二元操作。

关系代数在关系上是封闭的，即任何关系代数操作在关系上的运算结果仍然是关系。关系代数的封闭性保证了关系代数操作的可嵌套性。在关系数据模型中，正是关系代数的可嵌套性，导致有时用关系代数表达用户的操作需求比较困难，因此关系代数操作层层嵌套，使得关系代数表达式异常复杂。

3.5.2　关系代数的组成

在早期的关系数据模型中，Edgar F. Codd 定义了 4 种集合操作以及 4 种专门的关系代数操作，在本书中将这 8 种操作称为原始的关系代数。与此相对的是附加的关系代数，它们是研究者针对原始关系代数的不足而提出的扩展。无论原始的关系代数还是附加的关系代数，目前在商用 DBMS 上除了一些少数操作外基本都是支持的。因此，将对原始关系代数和附加关系代数都进行详细的讨论。

图 3-9 给出了关系代数的基本组成。从图中可以看到，原始的关系代数包括了 4 个传统集合操作：并、交、差、笛卡儿积，以及 4 个专门的关系代数操作：选择、投影、连接、除。在附加的关系代数中，着重介绍 6 种操作：重命名、扩展投影、聚集、分组、排序、赋值。需要指出的是，附加的关系代数操作是人们后来补充的，并不只有这 6 种操作。

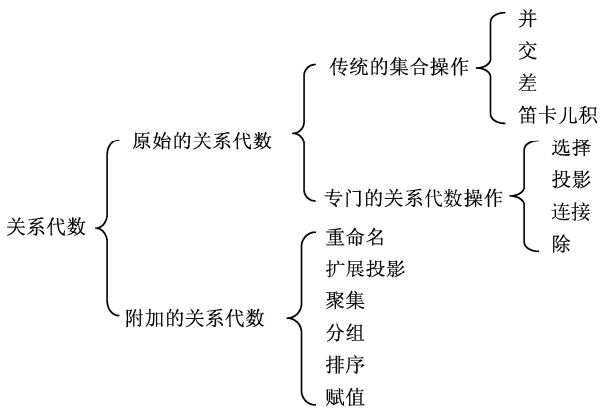


图 3-9　关系代数的基本组成

3.5.3　传统的集合操作

4 个传统的集合操作具体如下。

(1)并(Union)：返回两个关系中所有元组。

(2)交(Intersection)：返回两个关系共同的元组。

(3)差(Difference)：返回属于第一个关系但不属于第二个关系的元组。

(4) 笛卡儿积(Cartesian Product)：返回两个关系的元组的任意组合所得到的元组集合。

下面详细讨论每一个操作的定义和实例。

1. 并

并操作定义如下：

$$R \cup S = \{t \mid t \in R \vee t \in S\}$$

其中，t 是元组变量；R 和 S 是关系代数表达式，R 和 S 的属性个数(度)必须相同，R 和 S 的类型也必须相同。

图 3-10 给出了两个关系并操作的示例。需要注意的是，由于集合不允许有重复的元素，因此在并的过程中会自动剔除重复的元组。

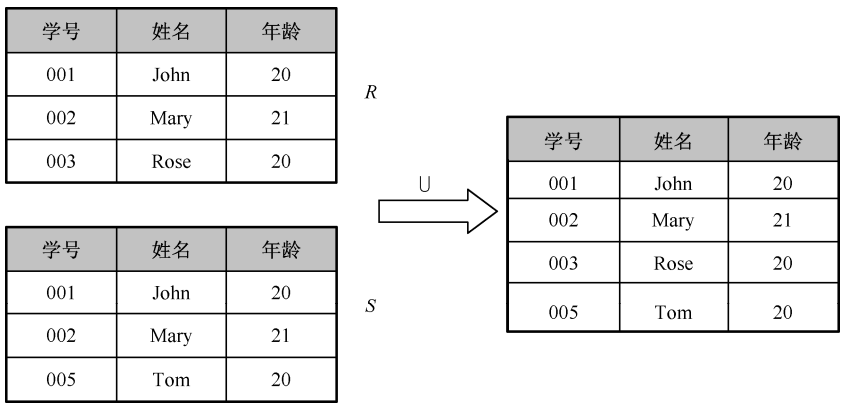


图 3-10　并操作示例

并操作是可交换的，因此 $R \cup S$ 等价于 $S \cup R$。

2. 交

交操作定义如下：

$$R \cap S = \{t \mid t \in R \wedge t \in S\}$$

其中，t 是元组变量；R 和 S 是关系代数表达式，R 和 S 的属性个数(度)必须相同，R 和 S 的类型也必须相同。

图 3-11 给出了两个关系交操作的示例，它返回了 R 和 S 中公共的两个元组。交操作也是可交换的，因此 $R \cap S$ 等价于 $S \cap R$。

3. 差

R 和 S 的差操作返回属于 R 但不属于 S 的元组，定义如下：

$$R - S = \{t \mid t \in R \wedge t \notin S\}$$

其中，t 是元组变量；R 和 S 是关系代数表达式，R 和 S 的属性个数(度)必须相同，R 和 S 的类型也必须相同。

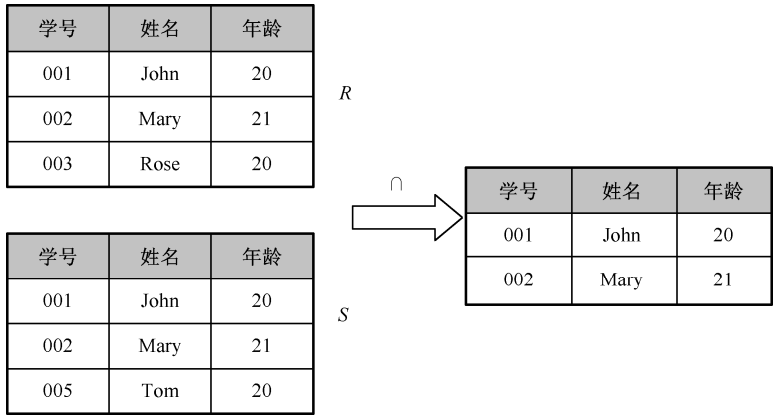


图 3-11　交操作示例

图 3-12 给出了两个关系 *R* 和 *S* 的差操作的示例，它返回了属于 *R* 但不属于 *S* 的元组。

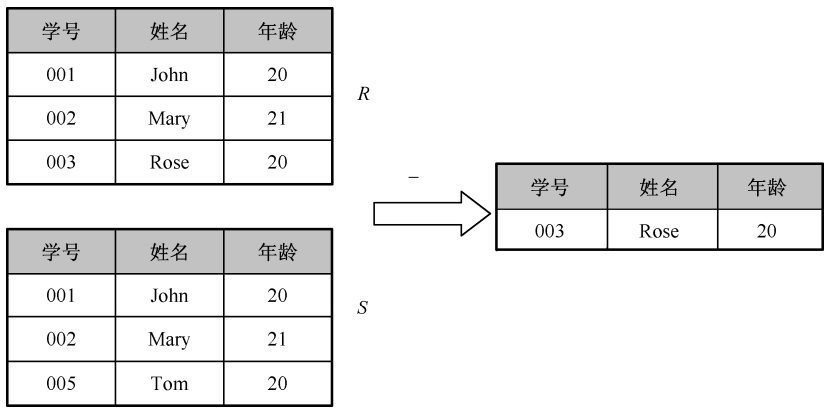


图 3-12　差操作示例

注意，差操作不满足交换律。如图 3-12 所示，*R*–*S* 的结果是元组{‘003’, ‘Rose’, 20}，而 *S*–*R* 的结果则是元组{‘005’, ‘Tom’, 20}。

4. 笛卡儿积

笛卡儿积操作定义如下：

$$R \times S = \{t \mid t = <\text{tr}, \text{ts}> \wedge \text{tr} \in R \wedge \text{ts} \in S\}$$

其中，*t*、tr 和 ts 都是元组变量；*R* 和 *S* 是关系代数表达式。

笛卡儿积的结果元组的前一部分 tr 来自关系 *R*，而后一部分来自关系 *S*。图 3-13 给出了两个关系笛卡儿积操作的示例，它返回了 *R* 的任意元组和 *S* 的任意元组两两组合的结果。笛卡儿积操作也是可交换的，因此 $R \times S$ 等价于 $S \times R$。

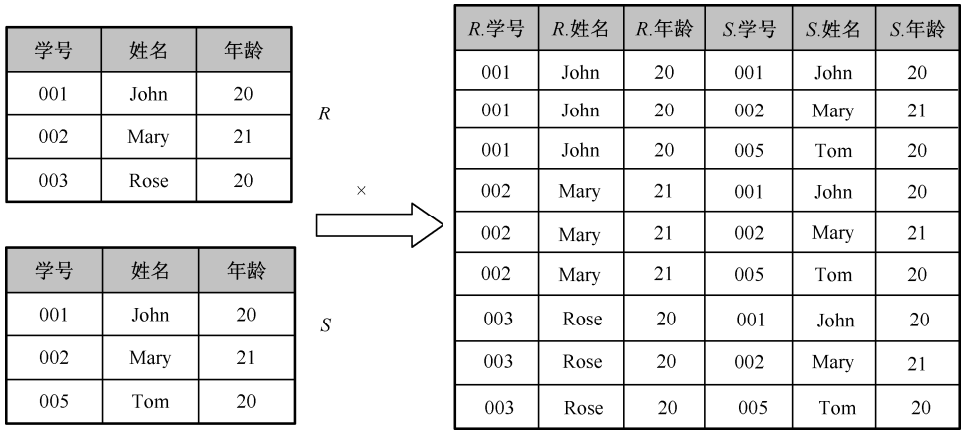
笛卡儿积具有如下几个特点：

(1) 结果关系的元组数目(基数)等于 *R* 的基数与 *S* 的基数的乘积。

(2) 结果关系的属性数目等于 *R* 的属性数目和 *S* 的属性数目之和。

在图 3-13 中，由于 *R* 和 *S* 的属性集相同，因此在结果关系中，通过在属性名之前加上

关系名前缀来区分属性。在关系代数表达式中，如果出现同名的属性，则要求在属性名之前加上关系名。如果属性名是唯一的，则可以不用加前缀。



R

学号	姓名	年龄
001	John	20
002	Mary	21
003	Rose	20

×

S

学号	姓名	年龄
001	John	20
002	Mary	21
005	Tom	20

R.学号	*R*.姓名	*R*.年龄	*S*.学号	*S*.姓名	*S*.年龄
001	John	20	001	John	20
001	John	20	002	Mary	21
001	John	20	005	Tom	20
002	Mary	21	001	John	20
002	Mary	21	002	Mary	21
002	Mary	21	005	Tom	20
003	Rose	20	001	John	20
003	Rose	20	002	Mary	21
003	Rose	20	005	Tom	20

图 3-13　笛卡儿积操作示例

3.5.4　专门的关系代数操作

专门的关系代数操作包括选择、投影、连接和除。

(1) 选择(Select)：返回指定关系中满足给定条件的元组。

(2) 投影(Project)：返回指定关系中去掉若干属性后所得的元组。

(3) 连接(Join)：从两个关系的笛卡儿积中选取属性间满足给定条件的元组。

(4) 除(Divide)：除的结果与第二个关系(除数)的笛卡儿积包含在第一个关系(被除数)中。

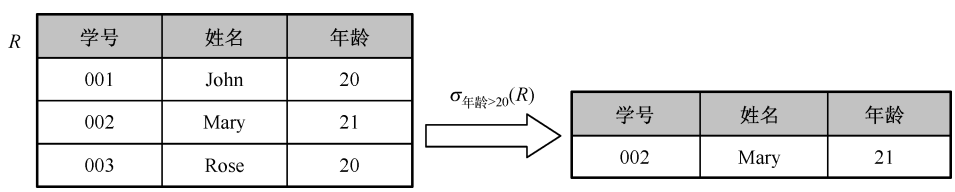
1. 选择

选择操作是一元操作，定义如下：

$$\sigma_F(R) = \{t \mid t \in R \wedge F(t) = \text{true}\}$$

其中，t 是元组变量；F 是一个逻辑表达式，表示 t 应满足的选择条件。F 由逻辑运算符¬(not)、∧(and)和∨(or)连接算术表达式构成。算术表达式形为 $X\theta Y$，其中 θ 可以是≤、≥、≠、=、>和<，X 和 Y 可以是常量、属性名或简单函数。

图 3-14 给出了选择操作的示例，它返回了 R 中满足“年龄>20”的学生元组。



R

学号	姓名	年龄
001	John	20
002	Mary	21
003	Rose	20

$\sigma_{\text{年龄}>20}(R)$

学号	姓名	年龄
002	Mary	21

图 3-14　选择操作示例

选择操作将关系水平划分为两个元组集合，一个元组集合是满足给定的选择条件 F 的，而另一个元组集合则是不满足条件的。选择的结果就是返回满足 F 的元组集合。

但是选择操作所得到的结果与原关系的度是一样的，即选择操作只对关系进行水平分组，不影响关系的模式结构。

在实际应用中，可以通过选择操作来表达用户按行查询的需求。

【例 3-1】 假设有学生关系模式：学生(学号, 姓名, 系, 年龄, 性别)。

(1) 查询计算机系的男学生信息，相应的关系代数表达式为

$$\sigma_{系='计算机系' \wedge 性别='男'}(学生)$$

(2) 查询年龄为 20 岁或 21 岁的学生信息，相应的关系代数表达式为

$$\sigma_{年龄=20 \vee 年龄=21}(学生)$$

2. 投影

投影操作也是一元操作，定义如下：

$$\pi_A(R) = \{t[A] \mid t \in R\}$$

其中，t 是元组变量；A 是 R 的一个属性子集。

图 3-15 给出了投影操作的示例，它返回了所有学生的姓名和年龄(注意，“学号”属性被投影操作去除了)。

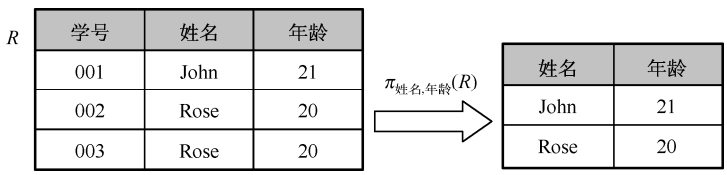


图 3-15　投影操作示例

投影操作是垂直划分整个关系。在许多情况下，用户并不需要返回一个关系中的所有属性，在这种情况下就可以使用投影操作来剔除不需要的属性，只返回想要的属性。此外，虽然投影操作没有指定按行选择的条件，但是投影操作所得到的结果中元组数目(基数)仍有可能少于原关系。其原因在于投影操作有可能会产生重复元组，这些重复元组在生成结果集时被自动剔除。以图 3-15 为例，投影“姓名”和“年龄”两列时，R 的第 2 个元组和第 3 个元组所得到的投影结果均为{‘Rose’, 20}，因此在最后的结果中只保留了一个{‘Rose’, 20}结果，以保证结果满足关系的基本性质，即关系中不存在重复元组。

选择和投影操作是关系代数最常用的操作中的两个。用户的许多查询需求都可以通过选择和投影操作来表达。由于关系代数操作具有可嵌套性，因此选择或投影操作的结果仍可以作为操作对象传递给下一个操作，从而表达较复杂的查询需求。

【例 3-2】 假设有学生关系模式：学生(学号, 姓名, 系, 年龄, 性别)，求计算机系男学生的学号和姓名。

该例题相应的关系代数表达式为

$$\pi_{学号,姓名}(\sigma_{系='计算机系' \wedge 性别='男'}(学生))$$

3. 连接

连接操作是关系代数较为复杂的操作。连接操作是二元操作，其本质语义是表达涉及

现实世界实体与实体之间的联系的查询。在连接操作中，参与计算的两个关系通常代表了两个实体，而连接操作则返回与这两个实体都相关的查询结果。

在关系代数中，连接操作有两种类型：自然连接(Natural Join)和θ连接(Theta Join)。

1) 自然连接

自然连接是最常见的连接操作。设参与自然连接的两个关系R和S的属性集分别为$R(X, Y)$和$S(Y, Z)$，自然连接的定义如下：

$$R \bowtie S = \{t \mid t = <X, Y, Z> \wedge t[X, Y] \in R \wedge t[Y, Z] \in S\}$$

R和S的自然连接的结果相当于在$R \times S$的结果中选择公共属性值Y都相等的元组，并且在最终结果中去掉重复属性，即

$$R \bowtie S = \pi_{X, R.Y, Z}(\sigma_{R.Y = S.Y}(R \times S))$$

图 3-16 给出了自然连接操作的示例。在这个示例中，要查询所有学生的学号、姓名、年龄、所选的课程号以及成绩。由于学号、姓名和年龄信息在学生关系中，而学生的选课信息在选课关系中，因此这里需要将这两个关系进行自然连接。自然连接操作将学生关系中的每一个学生元组与选课关系中学号相同的元组进行拼接(即在笛卡儿积结果上再执行一次“R.学号 = S.学号”的选择操作)，并在最后剔除掉重复的学号(S.学号)。

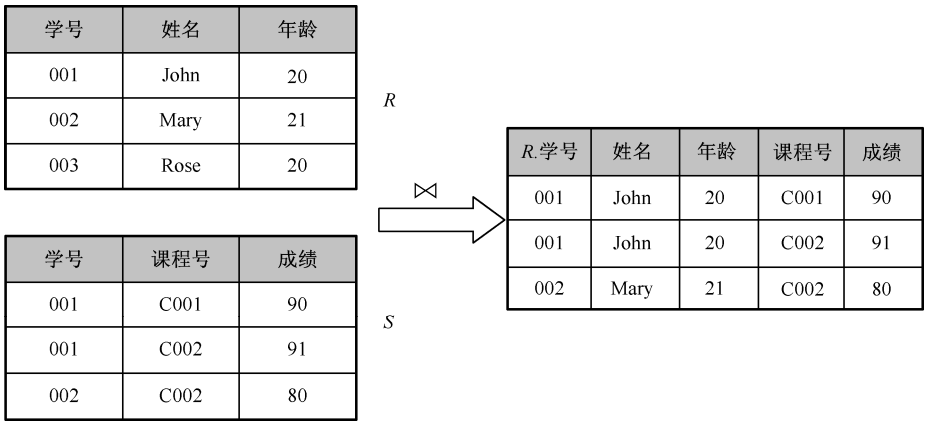


图 3-16　自然连接操作示例

自然连接总是在R和S的公共属性值上执行等值比较，其原因是当对两个关系R和S执行自然连接时，一般情况下对于R中的一个元组t，希望得到t在S中的相应信息，因此只有要求R和S在它们的公共属性值上相等，才能确保返回的所有信息都是关于一个实体的。以图 3-16 的示例为例，对于R的元组{‘001’, ‘John’, 20}，只有将S中学号为 001 的元组与其连接才有意义，因为只有这样的元组才代表 001 的选课信息。

2) θ 连接

虽然自然连接是实际应用中很常见的连接形式，但在某些情况下，希望两个关系之间的联系是一种不等值的比较关系。例如，有一个学生关系和一个教师关系，希望返回那些年龄大于教师的学生和教师对，此时要求在学生的“年龄”属性和教师的“年龄”属性上做一个不等值的比较。θ 连接正是用来回答此类涉及两个实体之间的不等值比较关系的连接查询的。

设两个关系 R 和 S 的属性集分别为 $R(X, Y)$ 和 $S(Y, Z)$，θ 连接的定义如下：

$$R \underset{A\theta B}{\bowtie} S = \{t \mid t = <\text{tr}, \text{ts}> \wedge \text{tr} \in R \wedge \text{ts} \in S \wedge \text{tr}[A]\theta\text{ts}[B]\}$$

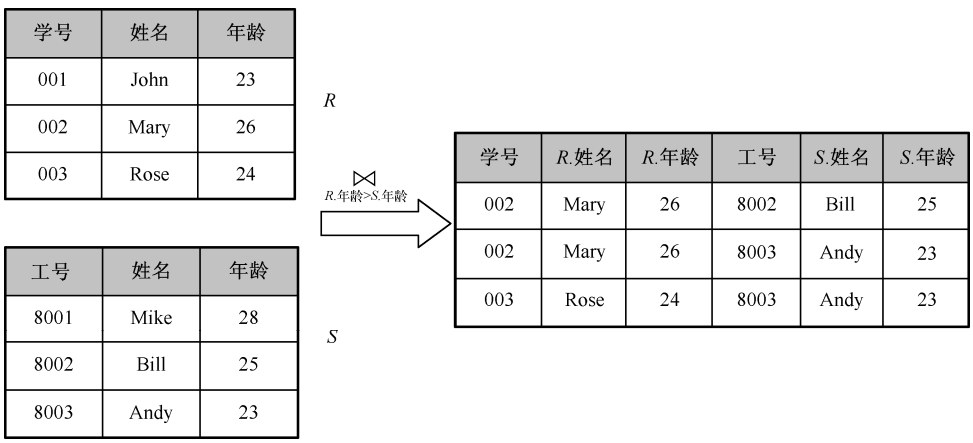
其中，A 和 B 分别是来自 R 和 S 的属性名；θ 是比较符，可以是≤、≥、≠、=、>和<。

θ 连接的定义与笛卡儿积有些类似，唯一的不同在于多了一个条件“$\text{tr}[A]\theta\text{ts}[B]$”，也就是说，$\theta$ 连接是在 $R \times S$ 中选取 R 的属性 A 值与 S 的属性 B 值满足比较关系 θ 的元组，用公式可以表示如下：

$$R \underset{A\theta B}{\bowtie} S = \sigma_{A=B}(R \times S)$$

当 θ 比较符是=时，习惯上将这种特殊的 θ 连接称为“等值连接”。如果在 R 和 S 的所有公共属性值上做等值连接，并且在结果中去掉重复的属性，则该结果等价于 R 和 S 自然连接的结果。

图 3-17 给出了 θ 连接操作的示例。在这个示例中，查询学生年龄大于某个教师的学生和教师信息。在返回的结果中，每一个元组中的学生年龄均大于该元组中的教师年龄。



学号	姓名	年龄
001	John	23
002	Mary	26
003	Rose	24

工号	姓名	年龄
8001	Mike	28
8002	Bill	25
8003	Andy	23

学号	R.姓名	R.年龄	工号	S.姓名	S.年龄
002	Mary	26	8002	Bill	25
002	Mary	26	8003	Andy	23
003	Rose	24	8003	Andy	23

图 3-17　θ 连接操作示例

4. 除

除操作是关系代数中比较特殊的一个操作。下面先给出除操作的定义：设关系 R 的属性集为 $\{X,Y\}$，S 的属性集为 $\{Y\}$，则 $R \div S$ 的结果是一个关系 P，P 的属性集为 $\{X\}$，并且 $P \times S$ 包含在 R 中。

$R \div S$ 的计算方法如下：

(1) $T = \pi_X(R)$；

(2) $W = T \times S - R$；

(3) $V = \pi_X(W)$；

(4) $R \div S = T - V$。

以上的计算过程用文字描述，即 $\{X\}$ 和 $\{Y\}$ 的笛卡儿积不包含在 R 中的 $\{X\}$，R 的 $\{X\}$ 投影(即 $\pi_X(R)$)减去这部分 $\{X\}$，剩下的 $\{X\}$ 与 $\{Y\}$ 的笛卡儿积都包含在 R 中，这一 $\{X\}$ 就是 $R \div S$ 的最终结果。

除操作有两个特点：

(1) 被除数 R 和除数 S 须满足 $S \subseteq R$；

(2) 设 R 的属性集表示为 A_R，则 $A_{R\div S} = A_R - A_S \cdots A_{R\div S} = A_R - A_S$。

一般情况下，当涉及“全部”“所有”之类的查询时，可以考虑用除操作来实现。

【例 3-3】 设学生选课关系模式：选课(学号, 课程号, 成绩)，求选修了学生 001 所选全部课程的学生学号。

第一步，先求出学生 001 所选全部课程的课程号：$\pi_{课程号}(\sigma_{学号='001'}(选课))$。

第二步，求出选修了学生 001 所选全部课程的学生学号：

$$\pi_{学号,课程号}(选课) \div \pi_{课程号}(\sigma_{学号='001'}(选课))$$

这个示例中，要求这些学号对应的学生选修了学生 001 所选的全部课程，也就是说，要求这些学号与学生 001 所选的全部课程的课程号的笛卡儿积出现在选课关系中，因此可以用除操作。同时因为除操作要求 $A_{R\div S} = A_R - A_S$，所以被除数的属性集必须是{学号, 课程号}。

图 3-18 给出了例 3-3 计算过程的示例。

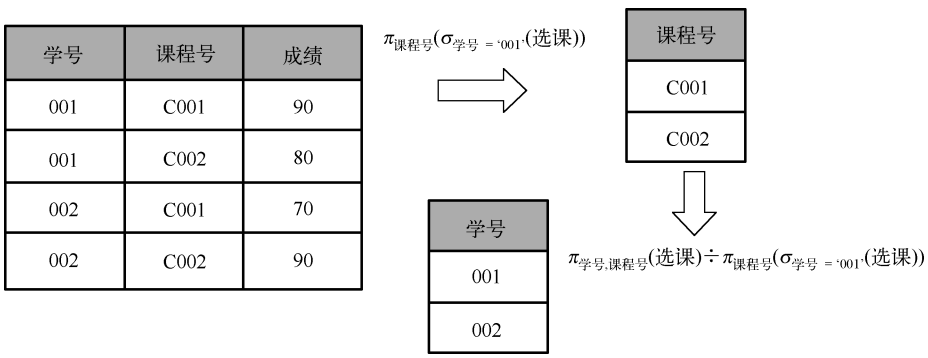


图 3-18　除操作示例

3.5.5　附加的关系代数操作

附加的关系代数操作是对原始的关系代数操作的补充。由于实际应用中有一些特殊的查询难以直接通过原始的关系代数操作表达，因此研究者逐步提出了一些附加的关系代数操作，使得关系代数的表达能力不断增强。

本节将着重讨论 6 种附加关系代数操作。目前，现有的关系 DBMS 中基本都支持这 6 种附加的关系代数操作。这 6 种附加关系代数操作具体如下。

1. 重命名

关系数据库中的关系在关系代数操作中都可以通过关系模式的名字进行引用，但是关系代数操作运算的结果没有可供引用的名字。这会带来两方面的问题：一是会使得在用关系代数表达复杂查询时书写困难；二是对于某些关系代数操作，如自连接(发生在一个关系上的自我连接)，必须使用某种重命名机制以保证操作的顺利进行。

重命名操作的引入使关系代数操作的结果可以方便地在其他位置引用。借助重命名操作，可以将某个关系或者关系代数操作结果进行重命名。

重命名操作使用符号ρ表示，并且有两种运算模式。

(1) $\rho_X(E)$：将关系代数表达式 E 重命名为 X。X 中的属性集与 E 的属性集相同。

(2) $\rho_{X(A_1, A_2, \cdots, A_n)}(E)$：将关系代数表达式 E 重命名为 X，并且 E 的各个属性在 X 中被更名为 $A_1, A_2, \cdots, A_n$。

【例 3-4】 设学生关系模式和选课关系模式分别为学生(学号, 姓名, 年龄)和选课(学号, 课程号, 成绩)，求每个学生的学号、姓名、课程号和成绩。

下面给出了使用了重命名操作的计算结果：

$$\pi_{R.\text{sno}, R.\text{name}, S.\text{cno}, S.\text{score}}(\rho_{R(\text{sno,name,age})}(\text{学生}) \bowtie \rho_{S(\text{sno,cno,score})}(\text{选课}))$$

通过例 3-4 可以看到，在计算学生关系与选课关系的自然连接结果时，将学生关系重命名为了 R(sno, name, age)，将选课关系重命名为了 S(sno, cno, score)，因此在后面的投影操作中就可以直接使用 R 和 S 来进行运算。

2. 扩展投影

原始关系代数中的投影操作$\pi_A(R)$要求投影出来的属性集 A 必须是关系 R 的一个属性子集，但是在实际应用中有的时候希望投影出来的结果是一个与 R 属性值相关的计算结果。例如，员工关系模式中有“工资”属性，用户可能会查询“税后工资”，即“工资×0.95”(假设个人所得税率为 5%)。这种查询在原始的投影操作中是无法实现的，扩展投影的引入就是为了满足类似的查询需求。

扩展投影仍用$\pi_A(R)$来表示，其中投影得到的属性集 A 可以是以下所列出的元素之一。

(1) R 的一个属性。

(2) 形如 $x \to y$ 的表达式，其中，x 和 y 都是属性的名字。元素 $x \to y$ 表示投影出 R 中的 x 属性并重命名为 y。

(3) 形如 $E \to z$ 的表达式，其中，E 是一个涉及 R 的属性、常量、代数运算符或者字符串运算符的表达式，z 是表达式 E 得到的结果属性的新名字。

第(1)种情形对应的就是原始的投影操作，第(2)种情形是允许在投影结果中对属性进行重命名，第(3)种情形表示在投影列表中可以使用表达式。常见的表达式包括字符串表达式、算术表达式、函数表达式等。

借助扩展投影，可以在投影列表中使用由重命名以及表达式构成的元素来表达一些涉及属性运算的查询。例如，元素 $a+b \to x$ 表示 a 和 b 属性的和，并重命名为 x；元素 $c \| d \to e$ 表示连接字符串类型的属性 c 和 d，并重命名为 e。

【例 3-5】 设员工关系模式为 Employee(E#, FirstName, LastName, Salary)，下面的查询使用了扩展投影操作。

(1) 使用字符串表达式：求每个员工的员工号和姓名全名。

$$\pi_{\text{E\#, FirstName} \| \text{ LastName} \to \text{name}}(\text{Employee})$$

(2) 使用算术表达式：求每个员工的员工号和税后工资(设个人所得税率为 3%)。

$$\pi_{\text{E\#, salary*0.97} \to \text{taxed\_salary}}(\text{Employee})$$

(3) 使用函数表达式：求全体员工的平均工资。

$$\pi_{\mathrm{AVG(salary)}\rightarrow \mathrm{avg\_salary}}(\mathrm{Employee})$$

3. 聚集

聚集操作是一类具有统计性质的操作，它在一个值集上进行相应的统计操作并输出一个单一的值作为结果，如求全体学生的人数(输入为学生集合，输出为一个整数值)、求“数据库”课程的平均成绩(输入为“数据库”课程的成绩集合，输出为一个平均成绩)等。在实际企业应用中，聚集查询很普遍，但是在原始的关系代数操作中，聚集查询无法表达，因此引入聚集操作来表达此类查询。

聚集操作通过使用 5 类聚集函数来完成相应的统计查询功能，这些聚集函数的输入都是一个值集，返回值都是一个单一的值。下面给出这 5 类函数的说明。

(1) SUM：求和函数。SUM 函数返回一个值集上的汇总求和结果。

(2) AVG：求均值函数。AVG 函数返回一个值集上的平均值。

(3) MAX：求最大值函数。MAX 函数返回一个值集上的最大值。

(4) MIN：求最小值函数。MIN 函数返回一个值集上的最小值。

(5) COUNT：计数函数。COUNT 函数返回一个值集中的元素个数。

通常情况下，这些聚集函数的参数都是属性名或者表达式。结合聚集函数与扩展投影，就可以表达实际应用中的统计查询。

【例 3-6】 设员工关系模式为 Employee(E#, name, age, salary)，下面的聚集查询给出了聚集函数的一般用法。

(1) SUM：求全体员工的工资总和。

$$\pi_{\mathrm{SUM(salary)}\rightarrow \mathrm{sum\_salary}}(\mathrm{Employee})$$

(2) AVG：求全体员工的平均年龄。

$$\pi_{\mathrm{AVG(age)}\rightarrow \mathrm{avg\_age}}(\mathrm{Employee})$$

(3) MAX/MIN：求所有员工的最高工资和最低工资。

$$\pi_{\mathrm{MAX(salary)}\rightarrow \mathrm{max\_salary},\ \ \mathrm{MIN(salary)}\rightarrow \mathrm{min\_salary}}(\mathrm{Employee})$$

(5) COUNT：求员工的总人数。

$$\pi_{\mathrm{COUNT(E\#)}\rightarrow \mathrm{count\_emp}}(\mathrm{Employee})$$

或

$$\pi_{\mathrm{COUNT(*)}\rightarrow \mathrm{count\_emp}}(\mathrm{Employee})$$

在 5 个聚集函数中，COUNT 函数的输入一般可以是属性名、表达式或者特殊的符号*。符号*在关系代数操作中指代整个元组。由于一个关系中的元组都是不重复的，因此 COUNT(*)就可以表达对关系中所有元组的计数，这与 COUNT(主码)的效果是等价的。

4. 分组

分组操作是对聚集操作的进一步扩展。通常，人们不仅希望对关系的一整列求平均值

或者其他的聚集操作结果，很多情况下还希望先按照某一属性将整个值集分组，然后求出每个分组中的统计结果，例如，在选课关系上求每一门课程的平均成绩(而不是求所有课程的平均成绩)。分组操作通常是和聚集操作一起使用的，因为分组的结果就是为了在各个分组中再求解聚集函数结果。因此许多时候把分组操作和聚集操作合并在一起讨论，并称为“分组/聚集”操作。

分组操作用符号$\gamma_L(R)$表示，其中 L 由两部分构成，而且仅有以下这两部分。

(1)R 的一个或多个属性。这些属性称为“分组属性”，它们是对 R 上的所有元组进行分组的依据。

(2)应用到 R 的某个属性上的聚集函数。聚集函数所作用的属性称为“聚集属性”。在按“分组属性”对 R 的元组集合进行分组后，将分别计算每个分组上的聚集函数值。

分组操作$\gamma_L(R)$返回的关系是这样产生的。

第一步：把关系 R 的元组按 L 中的分组属性分组。每一组由 L 中分组属性上具有特定值的所有元组构成。举个例子，如果 L 中的分组属性上有三个不同的值，那么这一步就将 R 的整个元组集合分成三个组，每个组对应不同的分组属性值。例如，在学生关系中，如果分组属性是“性别”而且“性别”这一列上只有两个值 male 和 female，那么这一步就会将整个学生关系分为两个组，其中一组的“性别”属性值都为 male，而另一组则都是 female。

第二步：对于第一步产生的每个分组，输出一个元组。该元组包含该组的分组属性值以及该组中所有元组在列表 L 的聚集属性上应用聚集操作的结果。例如，分组属性是“性别”，聚集属性是“年龄”，聚集操作是 AVG(年龄)，那么在第一步按“性别”分组后，在第二步将对每一个分组中的元组计算各自的平均年龄，即返回了所有男学生和女学生的平均年龄。

图 3-19 给出了一个分组/聚集操作的示例。在这个示例中，在选课关系上计算每一门课程的平均成绩。在计算过程中，首先将所有的选课元组按分组属性“课程号”分组，然后分别求每个分组中的平均成绩。因为图 3-19 中的选课关系只有三个不同的课程号，因此所有元组最终分成了三个组，课程号 C001、C002 和 C003 各自对应一个组。其中课程号 C001 分组中包含 2 个元组，C002 和 C003 分组各包含一个元组。下一步就分别求这些分组中各自的平均成绩。

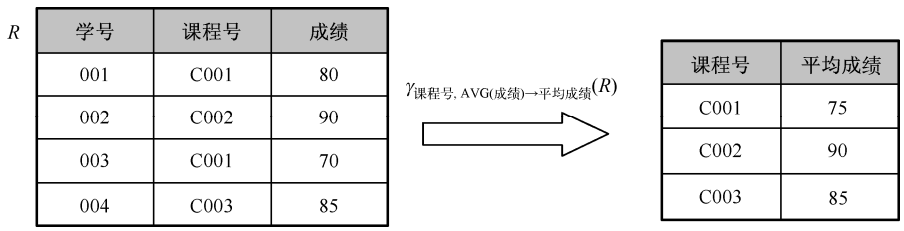


图 3-19　分组/聚集操作示例

5. 排序

排序操作$\tau_L(R)$返回按属性列表 L 排序的关系 R，结果中的所有元组按照 L 中的属性值

排序。如果 L 是 $A_1, A_2, \cdots, A_n$，那么 R 的元组就先按属性 A_1 的值排序，对于 A_1 属性值相等的元组，再按 A_2 的值排序，以此类推。

例如，对学生关系按年龄排序可以表示为 $\tau_{年龄}$(学生)。图 3-20 给出了 $\tau_{年龄}$(学生)操作的运算过程示例。

R

学号	姓名	年龄
001	John	19
002	Mary	20
003	Tom	18
004	Rose	21

$\tau_{年龄}$(学生)

学号	姓名	年龄
003	Tom	18
001	John	19
002	Mary	20
004	Rose	21

图 3-20　排序操作示例

6. 赋值

赋值操作允许给临时关系变量赋值，从而可以用关系代数分步骤来表示一个用户查询。赋值操作的符号为 $R \leftarrow E$，表示把关系代数表达式 E 的结果赋给关系变量 R。

【例 3-7】 设选课关系模式为 R(sno, cno, score)，求选了 PB00001001 所选全部课程的学生学号。

通过赋值操作分步骤来求解这一问题。

(1) 求出 PB00001001 所选全部课程并赋值给关系变量 Temp1；

$$\text{Temp1} \leftarrow \pi_{\text{cno}}(\sigma_{\text{sno}='\text{PB00001001}'}(R))$$

(2) 求出全部的选课信息并赋值给 Temp2；

$$\text{Temp2} \leftarrow \pi_{\text{sno, cno}}(R)$$

(3) 求出选了 PB00001001 所选全部课程的学生学号；

$$\text{Result} = \text{Temp2} \div \text{Temp1}$$

需要注意的是，赋值操作并不把结果显示给用户，最后一个步骤 Result = Temp2 ÷ Temp1 才表示把表达式结果输出。

3.5.6　关系代数的基本操作

在原始的关系代数操作中，并、差、积、选择、投影这 5 个操作称为关系代数的基本操作。基本操作是指这几个操作是关系代数的最小集合，其余的关系代数操作都可以通过这 5 个基本操作进行表示。

下面给出了基于并、差、积、选择、投影这 5 个基本操作的交、自然连接、θ 连接和除等操作的表示。

1) 交

$$R \cap S = R - (R - S)$$

2) 自然连接

设参与自然连接的两个关系模式 R 和 S 的属性集分别为 $R(X, Y)$ 和 $S(Y, Z)$，则有

$$R \bowtie S = \pi_{X, R.Y, Z}(\sigma_{R.Y = S.Y}(R \times S))$$

3) θ连接

设两个关系模式 R 和 S 的属性集分别为 $R(X, Y)$和 $S(Y, Z)$，则有

$$R \underset{A\theta B}{\bowtie} S = \sigma_{A\theta B}(R \times S)$$

4)除

设关系 R 的属性集为$\{X, Y\}$，S 的属性集为$\{Y\}$，则 $R \div S$ 的结果可表示为

$$R \div S = \pi_X(R) - \pi_X(((\pi_X(R)) \times S) - R)$$

3.5.7　关系代数表达式

关系代数是关系数据模型中数据操作的实现方式。关系代数只给出了一个操作集合，在实际回答用户的查询时，关系数据模型是通过关系代数表达式来实现的，也就是说，在关系数据模型中，所有的用户查询或者其他操作最终都是通过关系代数表达式来完成的。

定义 3-8　关系代数表达式的定义如下。

(1)关系代数中的基本表达式是关系代数表达式，基本表达式由如下之一构成：

①数据库中的一个关系；

②一个常量关系。

(2)设 E_1 和 E_2 是关系代数表达式，则下面的都是关系代数表达式：

①$E_1 \cup E_2$、$E_1 - E_2$、$E_1 \times E_2$；

②$\sigma_P(E_1)$，其中 P 是 E_1 中属性上的谓词；

③$\pi_S(E_1)$，其中 S 是 E_1 中某些属性的列表。

定义 3-8 指出，关系代数表达式可以是数据库中的关系、常量关系以及由 5 个关系代数基本操作所得到的结果。由于 5 个基本操作的结果都是关系代数表达式，因此可以递归地应用定义 3-8 得到嵌套的关系代数表达式。例如，$E_1 \cup E_2$ 是关系代数表达式，$\sigma_P(E_1 \cup E_2)$也是关系代数表达式。

下面给出关系代数表达式的一些求解示例。

【例 3-8】　设数据库中有下面的关系模式。

供应商关系模式：S(S#, Sname, City, Status)，其中各个属性分别表示供应商的供应商号、名称、所在城市、城市的状态。

零件关系模式：P(P#, Pname, Color, Weight)，其中各个属性分别表示零件号、名称、颜色、重量。

供应关系模式：SP(S#, P#, QTY)，其中 S#是外码并且参照 S.S#，P#也是外码并且参照 P.P#，QTY 为供应量。

下面是一些只涉及单个关系模式的简单查询示例。

(1)求 London 城市中的供应商的全部信息，相应的关系代数表达式为

$$\sigma_{\text{City = 'London'}}(S)$$

(2)求 London 城市中的供应商号、名称和状态，相应的关系代数表达式为

$$\pi_{\text{S\#, Sname, Status}}(\sigma_{\text{City = 'London'}}(S))$$

(3) 求红色并且重量不超过 15 的零件号和零件名，相应的关系代数表达式为

$$\pi_{\text{P\#, Pname}}(\sigma_{\text{Color}=\text{'Red'}\wedge\text{Weight}\leqslant 15}(P))$$

下面是一些涉及连接操作的复杂查询示例。

(1) 求提供零件 P_2 的供应商名称，相应的关系代数表达式为

$$\pi_{\text{Sname}}(\sigma_{\text{P\#}=\text{'}P_2\text{'}}(S \bowtie \text{SP}))$$

(2) 求提供红色零件的供应商名称，相应的关系代数表达式为

$$\pi_{\text{Sname}}(\sigma_{\text{Color}=\text{'Red'}}(S \bowtie \text{SP} \bowtie P))$$

注意，一个查询对应的关系代数表达式往往可以有多种形式，例如，上面的查询(2)也可以使用另一个关系代数表达式表示：

$$\pi_{\text{Sname}}(S \bowtie \text{SP} \bowtie (\sigma_{\text{Color}=\text{'Red'}}(P)))。$$

(3) 求位于同一城市的供应商号对，相应的关系代数表达式为

$$\pi_{\text{S1\#, S2\#}}(\sigma_{\text{S1\#}<\text{S2\#}}(\rho_{S_1(\text{S1\#, City})}(\pi_{\text{S\#, City}}(S)) \bowtie \rho_{S_2(\text{S2\#, City})}(\pi_{\text{S\#, City}}(S)))))$$

查询(3)是一个自连接查询。在关系数据模型中，由于一个关系对应着一类实体的集合，因此实体之间的联系需要通过关系与关系之间的连接操作来实现。在本查询中，需要查询供应商实体之间的联系，即位于同一城市，因此将关系模式 S 重命名为两个关系模式 S_1 和 S_2，然后在 S_1 和 S_2 之间进行自然连接查询。注意，为了体现“位于同一城市”的查询语义，需要将 S#在两个关系模式中命名为不同的名字。这是因为自然连接操作会在所有共同属性上都进行等值比较，而本查询只需要在 City 属性上做等值比较，因此需要将 S#在 S_1 和 S_2 中重命名为不同的名字以避免在 S#上也进行等值比较。最后，外层的选择操作用于在结果中去除供应商自己与自己连接生成的元组。

下面再给出一些涉及除操作的复杂查询示例。

(1) 求提供所有零件的供应商名称，相应的关系代数表达式为

$$\pi_{\text{Sname}}(S \bowtie (\pi_{\text{S\#, P\#}}(\text{SP}) \div \pi_{\text{P\#}}(P)))$$

在该查询中，关键的操作是其中的除操作：$\pi_{\text{S\#, P\#}}(\text{SP}) \div \pi_{\text{P\#}}(P)$。除操作的结果是返回提供所有零件的供应商号。

(2) 求至少提供了 S_2 提供的所有零件的供应商名称，相应的关系代数表达式为

$$\pi_{\text{Sname}}(S \bowtie (\pi_{\text{S\#, P\#}}(\text{SP}) \div \pi_{\text{P\#}}(\sigma_{\text{S\#}=\text{'}S_2\text{'}}(\text{SP}))))$$

该查询中的除操作返回了提供了 S_2 所提供的全部零件的供应商号，其中也包含了 S_2 自己。

(3) 求不提供零件 P_2 的供应商名称，相应的关系代数表达式为

$$\pi_{\text{Sname}}(S) - \pi_{\text{Sname}}(\sigma_{\text{P\#}=\text{'}P_2\text{'}}(S \bowtie \text{SP}))$$

3.5.8　数据更新的实现

在关系数据模型中，所有的数据操作都是通过关系运算(如关系代数)来实现的。前面讨论了用关系代数表达式来回答用户查询的示例，在这一节中讨论数据更新的实现。

数据更新操作具体包括数据的插入、删除和修改三种操作。数据更新操作仍然通过关系代数操作来实现，但是需要借助附加的一个关系代数操作——赋值操作（见 3.5.5 节）。

1. 数据的插入

数据的插入操作通过赋值和并操作来表达，即

$$R \leftarrow R \cup E$$

其中，R 是关系；E 是关系代数表达式。如果 E 是常量关系，则可以插入单个元组。

【例 3-9】 设关系模式 S_1 和 S_2 分别表示本科生数据和研究生数据，并且具有相同的模式，则把 S_1 中满足条件 P 的本科生插入到 S_2 中可表示为

$$S_2 \leftarrow S_2 \cup \sigma_P(S_1)$$

【例 3-10】 设学生关系模式为 S(sno, sname, age)，则在 S 中插入一个新的学生可表示为

$$S \leftarrow S \cup \{(\text{'001'}, \text{'Rose'}, 19)\}$$

2. 数据的删除

数据删除操作通过赋值和差操作来表达，即

$$R \leftarrow R - E$$

其中，R 是关系；E 是关系代数表达式。

【例 3-11】 设学生关系模式为 S(sno, sname, age)，则在 S 中删除姓名为 Rose 的学生可表示为

$$S \leftarrow S - \sigma_{\text{sname} = \text{'Rose'}}(S)$$

3. 数据的修改

数据的修改操作通过赋值和扩展投影来实现，即

$$R \rightarrow \pi_{F_1, F_2, \cdots, F_n}(R)$$

其中，$F_i(i = 1, 2, \cdots, n)$ 的定义为：当第 i 个属性没有被修改时是 R 的第 i 个属性；当被修改时是第 i 个属性和一个常量的表达式。如果只想修改 R 中满足条件 P 的部分元组，可以用下面的表达式：

$$R \rightarrow \pi_{F_1, F_2, \cdots, F_n}(\sigma_P(R)) \cup (R - \sigma_P(R))$$

【例 3-12】 设学生关系模式为 S(sno, sname, gender, age)，则将每个学生的学号前加上字母 S 可表示为

$$S \leftarrow \pi_{\text{'S'} \,\|\, \text{sno, sname gender, age}}(S)$$

将所有男学生的学号前加上字母 M 可表示为

$$S \leftarrow \pi_{\text{'M'} \,\|\, \text{sno, sname gender, age}}(\sigma_{\text{gender} = \text{'male'}}(S)) \cup (S - \sigma_{\text{gender} = \text{'male'}}(S))$$

3.6 本 章 小 结

本章主要介绍了关系数据模型的相关概念，包括关系、元组、属性、码、关系模式等，

并从数据结构、数据操作和数据约束三个方面详细给出了关系数据模型的形式化定义。本章着重讨论了关系数据模型中关系代数的概念以及关系代数表达式的书写。

通过对本章的学习，读者应掌握关系数据模型的基本概念，了解其中涉及的关系、外码等概念，要求能够从数据模型的三个要素角度深入理解关系数据模型的组成，并能够熟练运用关系代数表达式回答用户的查询。

习　　题

1．什么是数据模型？

2．数据模型通常通过哪些方面来形式化定义？

3．概念数据模型与结构数据模型的主要区别是什么？

4．关系数据模型中关系和关系模式之间是什么关系？

5．关系数据模型的基本性质有哪些？

6．在关系数据模型中，实体、实体间的联系以及语义约束都是如何表达的？

7．关系数据模型的三类完整性规则分别指什么？具体是何含义？

8．关系代数的基本操作有哪些？

9．自然连接和θ连接有何区别？

10．设有下面的关系模式：

图书(图书号: char, 书名: char, 作者: char, 单价: float, 库存量: float)

读者(读者号: char, 姓名: char, 工作单位: char, 地址: char)

借阅(图书号: char, 读者号: char, 借期: datetime, 还期: datetime, 备注: char)

其中，还期为 NULL 表示该书未还。请用关系代数表达式回答下面的查询：

(1)检索工作单位为中国科学技术大学的读者的读者号和姓名；

(2)检索 Ullman 所写的书的书名和单价；

(3)检索借阅了 Ullman 所写的书(包括借阅未还和已还)的读者姓名和借期；

(4)检索未借阅图书《数据库基础》的读者姓名；

(5)检索读者李林借阅过并且已还的图书的作者；

(6)检索借阅图书数目超过 3 本的读者姓名；

(7)检索没有借阅读者李林所借的任何一本书的读者姓名和读者号。

第 4 章　结构化查询语言

数据库查询语言(简称数据库语言)是数据库管理系统与用户之间的唯一交互接口，也就是说，用户对于数据库的所有操作都最终通过数据库查询语言来实现。由于目前关系数据库技术最为流行，因此本章将主要介绍关系数据库查询语言。关系数据库查询语言经过了几十年的发展，现在已经标准化，称为结构化查询语言(SQL)，并得到了 IBM、Oracle、Microsoft 等主流数据库厂商的支持。SQL 的基本功能包括对数据库模式的操作(数据定义)、对数据库的操作(数据操纵)、对数据库访问控制信息的操作(数据控制)以及对程序开发语言的支持(嵌入式 SQL)等。随着程序开发技术的不断进步，嵌入式 SQL 技术已经很少有人使用，因此本章将重点讨论 SQL 的数据定义、数据操纵以及数据控制功能。

需要注意的是，本章重点介绍 SQL 的常用语法，并不给出每条语句完整的用法。一方面是因为一些语法结构平时很少用，所以可以在数据库应用开发需要时再去参考相应的语法手册；另一方面是因为目前 Oracle、MySQL、Microsoft SQL Server 等不同 DBMS 在 SQL 语法上存在一定的差异。

内容提要：本章首先介绍结构化查询语言的发展历史和基本组成，然后着重介绍 SQL 中数据定义、数据更新、数据查询、数据控制等操作的实现，之后讨论 SQL 中视图的用法，最后总结 Oracle、MySQL、Microsoft SQL Server 在 SQL 上的主要差别。

4.1　SQL 概述

SQL[①]是一种结构化的关系数据库查询语言。结构化是指 SQL 在书写时需要遵循一定的结构。关系数据库管理系统作为关系数据模型的物理实现，其数据库查询语言的实现也主要依据了关系运算的定义。因此可以说，SQL 是关系运算的特定实现。但是，SQL 并非关系运算的唯一实现方式，理论上也可以设计别的查询语言。SQL 之所以著名，主要是因为它已经成为 ANSI 和 ISO 建议的关系数据库标准语言。本节将主要介绍数据库语言的概况以及 SQL 的基本情况。

4.1.1　数据库语言概述

DBMS 通过数据库模式定义了数据库的逻辑结构和物理结构，但随之引出了数据库存取问题，包括数据库模式信息、数据库数据以及数据库访问控制信息的存取。其中最重要的问题是数据库数据的存取，即如何将数据存储到数据库中，以及如何修改和查询数据库中的数据。数据库语言作为数据库存取问题的解决手段，通过数据库查询语句来完成数据库的存取。同时，为了简化和统一用户和 DBMS 之间的接口，学术界和企业界一致规定：

① SQL 一般发音/'siːkw(ə)l/，很少发音 Ess-cue-ell，不能发音成 circle/'sɜːk(ə)l/。

数据库语言是用户的 DBMS 交互的唯一方式。这一规定使得各个 DBMS 与用户之间的交互方式得到了统一，也极大地推动了关系数据库技术的发展。

数据库语言一般包括几种子语言。

(1)数据定义语言：专门用于定义和操纵数据库模式结构的语言。

(2)数据操纵语言：专门用于操纵数据库数据的语言，包括数据的增、删、改、查等。DML 是用户最常用的语言。

(3)数据控制语言：专门用于操纵数据库访问控制信息的语言，主要用于用户的授权与验证。早期的数据库语言还包含对数据库语言嵌入到程序设计语言(称为宿主语言)的支持。

嵌入式数据库语言是专门用于嵌入到前端的程序设计语言中完成数据库访问的语言。由于宿主语言与数据库语言的语法、标识符等存在差异，因此需要有专门的嵌入式数据库语言来解决宿主语言如何存取数据库的问题。嵌入式数据库语言在早期数据库应用开发中常使用，但随着面向对象程序设计语言和开发工具的发展，目前除了在一些嵌入式平台上外，已经很少使用了。

关系数据库语言是关系数据模型中数据操作的实现，因此其逻辑基础是关系数据模型。由于关系数据库语言(如 SQL)是在研究和实现关系 DBMS 的过程中提出的，它的功能和语法与关系运算存在着一些差别，主要的差别可以归纳为下面几点。

(1)关系运算是基于集合论的，而关系数据库语言(以 SQL 为例)则是基于包(Bag，也称 Multiset，多集)运算的。集合中不存在重复元素，而包中是允许存在重复元素的。SQL 之所以基于包理论，是因为某些实际应用需要使用重复元素。例如，“打印所有学生的名单”这一查询需要返回所有学生的姓名，包括重名的学生。如果基于集合运算来实现 SQL，则 SQL 将无法实现这一操作。SQL 的这一特性也使得 SQL 中存在一些传统关系运算所没有的操作，如查询结果的去重。理解了 SQL 和关系运算背后的这一本质区别，有助于更好地理解后面的 SQL 语句。

(2)关系运算只定义了对关系数据库数据的操作，而 SQL 不仅支持对数据库的操作，还支持对数据库模式和数据库访问控制的操作。因此，SQL 是关系运算的特定实现，但它在功能上比传统的关系运算做了更切实际的扩展。

(3)关系运算所基于的关系模式一定存在候选码和主码，而 SQL 中与关系对应的基本表不强制要求定义主码。这主要是由第(1)点差别引起的：因为 SQL 中允许基本表中出现重复记录，因此基本表就不能保证一定存在可区分每一条记录的字段集合(即候选码)。

4.1.2　SQL 的发展历史

SQL 最早是 IBM 公司提出的。关系数据模型是由 IBM 的 Edgar F. Codd 在 1970 年提出的。关系模型提出之后，Edgar F. Codd 与他的同事于 1972 年开始研究关系数据模型的系统实现问题，即设计和实现一个基于关系数据模型的 DBMS。他们将这 DBMS 命名为 System R，其中字母 R 是单词 Relational 的首字母，表示这是一个关系数据库系统。System R 开始配置的数据库语言称为 SQUARE(Specifying Queries as Relational Expressions)。就像 SQUARE 的英文表达一样，SQUARE 是通过关系代数表达式来表示查询的，因此使用了大量的数学符号。1974 年，IBM 的 Ray Boyce 和 Don Chamberlin 改进了 SQUARE 语言。他

们去掉了 SQUARE 语言中大量的数学符号，改用英语单词和结构式语法来表达查询。改进后的语言命名为 SEQUEL（Structured English Query Language，结构化英文查询语言），简称 SQL 并沿用至今。

SQL 因其语法简洁、表达能力强大而得到了主流数据库厂商的支持。从 20 世纪 70 年代末开始，Oracle、IBM、Sybase、Microsoft 等纷纷采用 SQL 作为它们推出的 DBMS 中的查询语言。1986 年，SQL 成为美国国家标准学会（ANSI）的标准，次年成为 ISO 标准。习惯上将这第一版的 SQL 标准称为 SQL86。在 1989 年时，SQL 标准得到了扩展，加入了引用完整性特征。修改后的 SQL 标准称为 SQL89。但由于 SQL89 与 SQL86 相比改动不是很大，因此 SQL89 标准在数据库历史上的影响较小。1992 年，ISO 颁布了 SQL92 标准。SQL92 相对 SQL89 做了较大的修改，最主要的修改是加入了对事务和隔离性的规定。SQL92 标准是 SQL 发展历史上影响最大的一个版本，目前的关系 DBMS 在实现时一般都以这一版本的 SQL 为标准。1999 年，ISO 颁布了新的 SQL 标准——SQL1999（也称为 SQL3）。以往的 SQL 标准都是一个文档，但 SQL3 分成了 SQL/Framework、SQL/Foundation、SQL/CLI、SQL/PSM 和 SQL/Bingdings 五个部分。这是因为在 20 世纪末期出现了许多新的技术和应用，如面向对象技术、Web 数据管理等，所以在 SQL3 标准中对于不同的技术和应用定义了单独的文档。SQL3 最主要的扩展是加入对面向对象数据库技术的支持，为研制对象关系 DBMS 提供了数据库语言的实现参考。

SQL 是一种声明性语言（Declarative Language）。它与 C 语言、PASCAL 等过程化语言（Procedural Language）有着较大的区别。像 C 语言这种过程化语言在书写时不仅要求给出程序的需求（如输入/输出参数、长度等），而且要求给出详细的算法（例如，用循环结构还是分支结构来完成处理）。而声明性语言则只要求用户给出操作的需求，即需要什么，而不需要给出如何操作的算法细节。由于 SQL 是声明性语言，因此用户在书写 SQL 语句时只要关注于自己的需求即可，不用去关心 DBMS 底层应该采取什么样的执行算法，SQL 虽然也是一种语言，但 SQL 语句的书写要比程序设计简单一些。由于 SQL 本身的表达能力很强，而应用的数据查询需求又千变万化，因此要想熟练掌握 SQL 还需要大量的学习和训练。

4.1.3　SQL 的基本组成

图 4-1 给出了 SQL 的基本组成。注意，图中只给出 SQL 的主要组成语句，全部的 SQL 语句可参考完整的 SQL92 标准文档。此外，虽然 SQL 有国际标准，不同的 DBMS 在具体实现 SQL 时仍存在着一些差别，如数据类型、语句、语法格式，本书在讨论 SQL 以及第 5 章过程化 SQL 时主要以 Microsoft SQL Server 为例。DB2 和 Oracle 中的 SQL 在表达能力上与 Microsoft SQL Server 没什么差别，只在一些细小的地方有一些不同。例如，变长字符串类型在 SQL Server 中表示为 Varchar，而在 Oracle 中则表示为 Varchar2。但这些

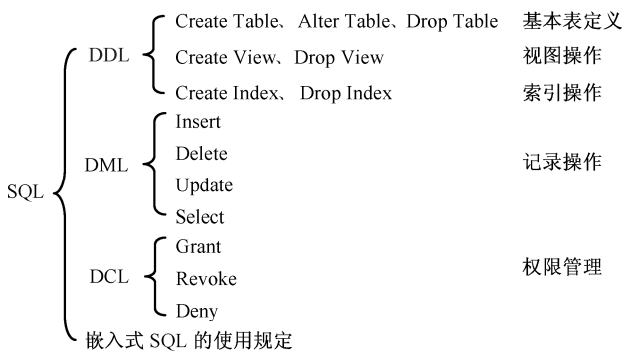


图 4-1　SQL 的基本组成

并不会影响对 SQL 以及本书其他内容的讨论和理解，因为所有这些关系 DBMS 的本质都是一样的，即都是基于关系数据模型的。

从图 4-1 中可以看出，SQL 包含了对数据库模式的操作(DDL)、对数据库的操作(DML)、对访问控制信息的操作(DCL)以及嵌入式 SQL 的使用规定。DDL 语句包括了对数据库概念模式、外模式和内模式的操作。ANSI/SPARC 体系结构指出数据库的三级模式结构是通过模式 DDL、外模式 DDL 和内模式 DDL 来定义的。概念模式在 SQL 数据库(支持 SQL 的关系数据库系统)中表现为基本表的集合，外模式表现为视图，而涉及数据库物理结构的内模式在 SQL 中则主要通过索引结构来反映。从图 4-1 中可以看到，模式 DDL 对应着基本表操作语句，如 Create Table、Alter Table 等，外模式 DDL 对应着视图操作语句(包括 Create View 和 Drop View 语句)，而索引操作则可以看成内模式 DDL。这一分类有助于更好地理解 ANSI/SPARC 体系结构以及 SQL 的体系。

4.2　数 据 定 义

数据定义语言(DDL)的主要功能是定义数据库的模式，包括概念模式、外模式和内模式。在 SQL 中对于不同的模式分别定义了一系列的语句。通过这些语句，数据库管理员(DBA)可以创建和维护数据库模式结构。数据库三级模式结构的核心是概念模式。

4.2.1　基本表的构成

一个基本表(在后面内容中，如果不加说明，“表”即表示基本表)由表名、列和约束构成。其中，表名代表着关系模式的名字，一般要求以字母开头，并可含数字、#、$、_ 等符号。在一个数据库中，表名不能重复，因为在 SQL 中用户需要根据表名来访问数据库中的数据。基本表的核心构成要素为列和约束，下面对这两个要素进行详细讨论。

1. 列

基本表的列对应着关系模式的属性。在关系模式中，属性由属性名和域构成，相应地，在基本表中，列包括列名和列类型两部分。其中列名要求以字母开头，可含数字、#、$、_ 等符号，并且要求不多于 30 个字符。表 4-1 给出了常见的列类型。

表 4-1　常见的列类型

类型	ANSI/ISO	Microsoft SQL Server	MySQL	Oracle
字符型	Char(*n*)	Char(*n*)	Char(*n*)	Char(*n*)
	Character(*n*)			
	Character Varying(*n*)	Varchar(*n*)	Varchar(*n*)	Varchar(*n*)
	Char Varying(*n*)			Varchar2(*n*)
数值型	Numeric	Numeric	Numeric	Number
	Decimal	Decimal	Decimal	
	Integer		Integer	

续表

类型	ANSI/ISO	Microsoft SQL Server	MySQL	Oracle
数值型	Int	Int	Int	Number
	Float	Float	Float	
	Double		Double	
	Real	Real	Real	
日期时间型	Date	Datetime	Date	Date
	Time		Time	
			Datetime	

需要注意的是，Microsoft SQL Server 与 ANSI/ISO SQL 标准之间在类型的标识上有一定的差别。例如，日期时间型在 SQL 标准中是分成两种类型 Date 和 Time 来表示的，而在 Microsoft SQL Server 中则合并成了一种类型 Datetime。另一些 DBMS（如 MySQL）则采用了 Date、Time 和 Datetime 三种类型的设计。事实上，由于同一类型（如 Date）在底层实现时存在着多种选择（例如，Date 类型可以在底层存储为字符串，也可以存储为整数——自 1900 年 1 月 1 日以来的天数），因此同一种类型在不同的 DBMS 产品中也可能在表达和精度上有差别。这些对于学习 SQL 没什么影响，因为 SQL 是声明性语言，人们关心的是如何使用 SQL 语句来表达查询需求，至于底层 DBMS 如何转换不同类型的值以及如何实现数据存储等问题并不需要去了解。

2. 约束

约束是完整性约束的简称。在基本表定义中，约束可以直接定义在每个列定义之后，也可以在全部列定义之后再单独定义。把直接定义在一个列定义之后的约束称为列约束，把定义在全部列定义之后的约束称为表约束。列约束和表约束所表达的语义没有什么差别，只是在基本表定义中的位置不同而已。因此，一个语义约束（如成绩> 0）既可以定义成列约束，也可以定义成表约束，它们所产生的结果是等价的。唯一需要指出的是，由于列约束只能跟在一个列定义后面，因此列约束只能对它所附属的当前列定义约束。如果某个约束定义在多个列之上，则必须通过表约束来实现，因为表约束可以定义涉及多个列的约束。

SQL 中的约束与关系数据模型中的完整性约束概念相同；可以将 SQL 约束看成关系模型中三类完整性约束的实现。但是，根据实际应用的需求，SQL 中共定义了 4 类约束。这些约束既可以定义成列约束，也可以定义成表约束。

（1）主键约束（Primary Key）：要求主码不可为空，也不能重复。基本上，它可以看成关系数据模型中的实体完整性约束在 SQL 中的实现。但是，由于实体完整性要求的主属性不可为空，而主属性不仅可以是主码的属性，还可以是候选码的属性，因此主键约束只定义了主码的约束要求，缺少了对候选码语义的实现。

（2）唯一性约束（Unique）：要求列值不可重复，但是可以为空。唯一性约束是对主键约束的补充，它可以看成对候选码语义的实现。

（3）外键约束（Foreign Key）：关系数据模型中参照完整性约束在 SQL 中的实现。它的语义和参照完整性完全一致，即外键值必须出现在被参照表中，或者为空。

(4)检查约束(Check)：允许用户根据应用环境的要求在某个或某些列上自定义约束条件。它体现的是关系数据模型中用户自定义完整性约束的语义。

因此，可以看到，SQL 中约束的逻辑基础是关系数据模型，它定义的 4 类约束基本遵循了关系模型的 3 类完整性约束。只要理解了关系数据模型，就很容易掌握 SQL 中的 4 类约束。

4.2.2　Create Table 语句

基本表的定义在 SQL 中通过 Create Table 语句实现。如前所述，基本表包括表名、列和约束三个部分，因此 Create Table 语句实际上就是给出这三部分的定义。下面给出 Create Table 语句的一般格式。注意，由于不同 DBMS 在 SQL 语法上存在一些差别，因此完整的 Create Table 语法在实际工作中可以参考 DBMS 的帮助文档。

```
Create Table <基本表名>(
   列名 1  列类型 1  [列约束 1],
   列名 2  列类型 2  [列约束 2],
   …
   [表约束]
)
```

【例 4-1】 下面的 Create Table 语句定义了基本表 Student：

```
Create Table Student(
   S#  Varchar(10) Constraint PK Primary Key,
   Sname Varchar(20),
   Age int,
   Gender Char(1)
)
```

例 4-1 中，Constraint PK Primary Key 是定义在列 S#上的主键约束。它采用了列约束的定义方式，直接跟在 S#定义后面(逗号之前)。需要注意的是，在 SQL 中，基本表不一定必须定义主键，因此上面的示例中如果去掉 Constraint PK Primary Key 也是符合语法格式的。

上面的 Create Table 语句格式只给出了一般的定义方式，下面对基本表的列定义和约束定义进行进一步的讨论。

1. 列定义

Create Table 语句中列定义的完整格式为：

```
<列名><列类型> [ DEFAULT <默认值>] [[NOT] NULL] [<列约束>]
```

其中，中括号表示该部分可选，DEFAULT 定义一个列的默认值，NULL/NOT NULL 表示该列可以为空或者不允许为空。

【例 4-2】 使用了 DEFAULT 和 NOT NULL 的 Create Table 语句：

```
Create Table Student(
   S#  Varchar(10) Constraint PK Primary Key,
   Sname Varchar(20) NOT NULL,
   Age int,
   Gender Char(1) DEFAULT 'F'
)
```

例 4-2 中，列 Sname 上加了 NOT NULL，表示该列不允许为空。实际上，NOT NULL 是一种简化的自定义约束(Check 约束)。在后面讨论 Check 约束时将看到如何用 Check 约束来表示 NOT NULL 的语义。

如果某个列上用 DEFAULT 定义了默认值，则当往表中插入一条新记录时，如果新记录中未指定该列的值，则 DBMS 将自动以默认值进行填充。以例 4-2 定义的表 Student 为例，如果执行 Insert 语句插入一条记录但是不指定 Gender 列的值，则在插入后新记录的 Gender 列值自动填充为 F，如图 4-2 所示。

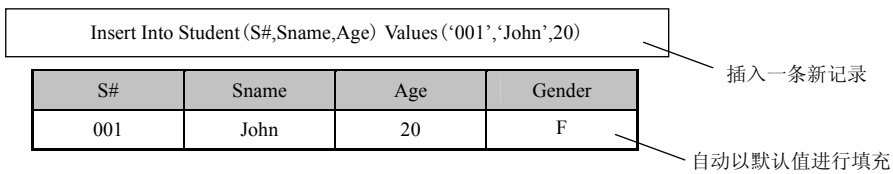


图 4-2　DEFAULT 示例

列定义中列约束的格式为：

```
[Constraint <约束名>]  <约束类型>
```

其中，约束类型可以是 Primary Key、Unique、Foreign Key 或者 Check，约束名由用户自行定义。列约束必须直接跟在列定义之后，只对当前列有效。在列约束定义中，除了约束类型之外，[Constraint <约束名>]是可选的，因此，下面的两种列约束定义方式都是正确的。

列约束定义方式 1：S# char(*n*) Constraint PK_Student Primary Key。

列约束定义方式 2：S# char(*n*) Primary Key。

一般情况下，建议采用方式 1。带 Constraint 关键字的方式 1 允许用户自定义约束的名称。这样做的好处是，当需要删除某个约束时，可以直接通过约束名进行删除。而如果采用方式 2，实际上列约束仍具有名称，只不过此时约束的名称是由 DBMS 自动生成的。因此如果需要删除约束，必须去系统表中查询该约束的名称，然后才能进行删除，因为约束的删除必须要给出约束的名称。一般情况下，可以采用以大写字母作为前缀的名称来定义约束名，这样通过约束名就可以知道约束的类型。例如，Primary Key 约束可以用 PK_Student 这样的名字来命名，表示 Student 表上的 Primary Key 约束。

2. 约束定义

基本表的约束有两种定义方式：列约束和表约束。如果一个约束是定义在单个列上的，则列约束和表约束的方式都可以使用；如果一个约束是定义在多个列上的，则只能采用表约束方式来定义。例 4-3 给出了同时使用列约束和表约束的示例。

【例 4-3】 下面的 Create Table 语句定义了基本表 Student，并且同时使用了列约束和表约束：

```
Create Table Student(
   S#  Varchar(10) Constraint PK_S Primary Key,
   Sname Varchar(20),
   Age int  Constraint CK_S Check (Age>14 and Age<100),
   Gender Char(1),
   Constraint UQ_S Unique(Sname),
   Constraint CK_SS Check (Gender IN ('M','F'))
)
```

在例 4-3 中，S#列上的主键约束使用了列约束方式，而 Sname 列和 Gender 列上的两个约束则使用了表约束方式。与列约束定义格式相比，表约束在约束类型之后需要明确给出所约束的列名称。

无论哪一种定义方式，都可以定义 4 类约束，即主键约束、唯一性约束、外键约束和检查约束。下面详细介绍这 4 类约束的定义。

1）主键约束

主键约束通过 Primary Key 关键字标识，标识列不可为空，也不能有重复值。与关系数据模型的要求一样，一个基本表中也只能存在一个主键，也就是说一个基本表上最多只能定义一个主键约束。

图 4-3 给出了主键约束的定义示例，包括列约束和表约束两种方式。其中列约束方式定义了学生表 Student，表约束方式定义了学生选课表 SC。

```
Create Table Student(
    S# Varchar(10) Constraint PK_S Primary Key,
    Sname Varchar(20),
    Age Int,
    Gender Char(1)
)
```

(a) 列约束

```
Create Table SC(
    S# Varchar(10)
    C# Varchar(20),
    Score Float,
    Constraint PK_SC Primary Key(S#,C#)
)
```

(b) 表约束

图 4-3　主键约束定义示例

2）唯一性约束

唯一性约束表示值可以为空，但不能重复。图 4-4 给出了唯一性约束的定义示例，包括列约束和表约束两种方式。图 4-4(a)中，列 Sname 上用列约束定义了 Unique 约束，表示学生的姓名不可重复，但允许为空；图 4-4(b)中，表约束方式定义的 Unqiue 约束 Unique (Dname, School)表示一个学院名和系名的组合不可重复。

由于 Unique 约束的列允许为空，因此 Unique 约束在空值处理上采取了一个特殊的规则。这一规则为“若 Unique 约束列中有一列不为空，就实施约束；若约束列都为空，则不实施约束”。图 4-5 以图 4-4(b)为例，给出了此规则的说明。

3）外键约束

外键约束表示所约束的列引用了其他表中的主键或者某个定义了 Unique 约束的列，外

键的值或者等于所引用列上的某个值，或者为空(如果外键列上没有定义主键或者 NOT NULL 约束)。外键约束与关系模型中的参照完整性语义是完全一致的。图 4-6 给出了外键约束的示例，包括列约束和表约束两种方式。需要注意的是，由于外键约束涉及两个基本表，所以外键约束的表约束定义方式相对要复杂一些，因此此时它必须给出所引用表和列的详细信息。

```
Create Table Student(
    S# Varchar(10) Constraint PK_S Primary Key,
    Sname Varchar(20) Constraint UQ_S Unique,
    Age Int,
    Gender Char(1)
)
```

(a) 列约束

```
Create Table Department(
    D# Varchar(10),
    Dname Varchar(20),
    School Char(20),
    Constraint UQ_D Unique(Dname, School)
)
```

(b) 表约束

图 4-4　唯一性约束定义示例

D#	Dname	School	在 Department 表中顺序插入下面的 D1, D2, …, D8 记录
D1	管理系	商学院	OK!　值唯一
D2	管理系	管理学院	OK!　值唯一
D3	管理系	管理学院	Error!　值重复
D4	管理系		OK!　值唯一
D5		管理学院	OK!　值唯一
D6		管理学院	Error!　实施约束
D7			OK!　约束列允许都空
D8			OK!　约束列都为空，不实施约束

图 4-5　Unique 约束对空值的处理示例

```
Create Table SC(
    S# Varchar(10) Constraint FK_SC References
        Student(S#),
    C# Varchar(20),
    Score Float,
    Constraint PK_SC Primary Key(S#,C#)
)
```

(a) 列约束

```
Create Table SC(
    S# Varchar(10),
    C# Varchar(20),
    Score Float,
    Constraint PK_SC Primary Key(S#,C#),
    Constraint FK_SC Foreign Key(S#) References
        Student(S#))
)
```

(b) 表约束

图 4-6　外键约束定义示例

如果定义了外键约束，则在表中进行记录操作时 DBMS 将实施参照完整性约束检查。以图 4-6 所定义的外键为例，图 4-7 记录操作时实施外键约束的示例。在这个示例中，往选课表 SC 中插入新记录时，由于学号 003 没有出现在被参照表 Student 的 S#列中，因此操作违背了外键定义的参照完整性，DBMS 将拒绝此操作。同理，在 Student 表中删除 001 学生记录时，DBMS 也会报错，因为此操作也违背了参照完整性(一旦 Student 表中的 001 学生记录被删除，SC 表中的记录就会违背参照完整性)。

被参照表(主表)Student，S#是主键

S#	Sname	Age
001	John	20
002	Rose	21

参照表(子表)SC，S#是外键

S#	C#	Score
001	C1	90
002	C2	95

```
Insert Into SC Values('003','C1',85); --Error occurs      !!违背了 SC 表上外键定义的参照完整性
Delete From Student Where S#='001'; - - Error occurs      !!违背了 SC 表上外键定义的参照完整性
```

图 4-7　记录操作时实施外键约束的示例

图 4-7 中，第 2 个操作“从 Student 表中删除 001 学生记录(Delete From Student Where S# = '001')”报错的原因在于参照表中存在着与即将被删除的被参照表记录相关联的记录。如果在定义外键约束时指定了 On Delete Cascade 选项，则此操作也可以顺利完成，但是其后果是 SC 表中所有参照被删除学号的记录也会一起删除。这种删除主表记录时将子表中的关联记录也一并删除的操作，在 SQL 中称为“级联删除”，具体的定义格式如下：

```
Create Table SC(
   S# Varchar(10),
   C# Varchar(20),
   Score Float,
   Constraint PK_SC Primary Key(S#, C#),
   Constraint FK_SC Foreign Key(S#) References Student(S#) On Delete
      Cascade
)
```

在外键约束定义时，可以指定 On Delete Cascade 选项或者 On Delete No Action 选项。其中，默认的是 On Delete No Action 选项。此时，如果试图删除某条记录，而该记录含有由其他表的现有记录中的外键值所引用的字段值，则产生错误并取消删除操作。

4)检查约束

检查约束根据应用环境的特殊要求自定义某些列上的约束。检查约束一般通过逻辑表达式来表示，如成绩 ＞0、性别 IN ('M', 'F')等。

【例 4-4】 下面是检查约束的一些示例：

```
Constraint CK_S1 Check (Age>15) --年龄大于 15 岁
Constraint CK_S2 Check (Gender In ('M', 'F'))       --性别是 M 或者 F
Constraint CK_SC Check (Score> = 0 and Score< = 100)
                                                    --成绩为[0, 100]
Constraint CK_S3 Check (Sname Is NOT NULL)          --姓名不能为空
Constraint CK_S4 Check (Postcode Like '[0-9][0-9][0-9][0-9][0-9][0-9]')
                                                    --6 位数字
```

特别需要指出的是，在例 4-4 中，Constraint CK_S3 Check (Sname Is Not NULL)与直接将 Sname 在列定义时加上 NOT NULL 选项的结果是等价的。因此，NOT NULL 实际上可以看作一种简化的检查约束。

4.2.3　Alter Table 语句

Alter Table 语句完成对基本表的修改。因为基本表包括表名、列和约束，因此 Alter Table 的修改功能也主要包括对列和约束的修改。注意，在 SQL 中表名无法修改。下面给出 Alter Table 语句的语法格式：

```
Alter Table <表名>
   [Add <列定义>] |
   [Alter Column <列定义>] |
   [Drop Column <列名>] |
   [Add <表约束>] |
   [Drop Constraint <约束名>]
```

其中，中括号内的部分为可选项。前三个选项分别对应了列增加、列修改和列删除操作，后两个选项对应了增加约束和删除约束功能。<列定义>和<表约束>的定义格式与 Create Table 语句相同。下面给出了对学生表 Student 的一些修改示例。

(1) 增加一个新列 Dept 并且要求唯一。

```
Alter Table Student Add Dept Varchar(10) Constraint UQ_S3 Unique
```

需要注意的是，如果原表已经存在部分记录，则这些记录的新增列值都为 NULL，所以如果增加的新列有一个 NOT NULL 约束，在原表中已经存在记录的情况下，增加新列的操作是无法执行的。

(2) 删除 Student 表中的列 Age。

```
Alter Table Student Drop Column Age
```

(3) 将 Student 表的 Gender 列修改为 Int 类型并且要求 NOT NULL。

```
Alter Table Student Alter Column Gender Int NOT NULL
```

修改列时列名是不能修改的。另外，如果将某个列修改为 NOT NULL，当原表中已存在部分记录时，该操作可能会失败，该操作能否执行取决于被修改的列上是否存在空值。如果存在空值，操作将无法执行。

(4) 为 Student 表增加一个约束，要求学生年龄大于 15 岁。

```
Alter Table Student WITH NOCHECK Add Constraint CK_Student Check(Age>15)
```

新增约束的定义必须采用表约束的格式。此外，增加约束时有两个选项可以选择：WITH CHECK 和 WITH NOCHECK，默认是 WITH CHECK。WITH CHECK 表示 DBMS 将把新增的约束应用到表中已有的记录上进行验证，WITH NOCHECK 则表示在新增约束时不进行记录验证操作。由于已有的记录可能会违背新增加的约束，因此默认的 WITH CHECK 选项要求在增加约束时，表中的记录必须都符合新约束的要求。例如，如果上面的 Alter Table 语句使用 WITH CHECK 选项，就要求 Student 表中所有记录的 Age 字段值都大于 15，否则该语句无法执行。

(5) 删除 Student 表上的 CK_Student 约束。

```
Alter Table Student Drop Constraint CK_Student
```

由于约束的删除必须要知道约束名称，因此在定义约束时最好使用带 Constraint 的完整格式自定义约束名。

4.2.4　Drop Table 语句

Drop Table 语句的功能是删除基本表(当然表所包含的记录也随之删除)。Drop Table 语句的用法很简单：

```
Drop Table <表名>
```

例如，删除 Student 表的 Drop Table 语句为：

```
Drop Table Student
```

当被删除的表与其他表存在外键关系时，执行 Drop Table 语句有可能会不成功。以图 4-7 所示的 Student 表和 SC 表为例，当删除 Student 表时 DBMS 将拒绝，因为 SC 表上存在着引用 Student 表 S#列的外键，因此如果 Student 表被删除，SC 上的外键所定义的参照完整性就会被破坏。

4.3　数 据 更 新

SQL 中的数据更新语句包括 Insert、Update 和 Delete，分别实现记录的新增、修改和删除功能。与后面要讨论的 Select 语句相比，数据更新语句相对较简单，主要通过一些示例来介绍这几条语句的用法。Insert、Update 和 Delete 语句的完整格式可参考 SQL 标准或者 Microsoft SQL Server 的帮助文档，这里只给出这几条语句的基本格式，也是实际应用中最常用的方式。

4.3.1　Insert 语句

Insert 语句的基本格式如下：

```
Insert Into <表名>[(列名 1,列名 2,…,列名 n)]Values(值 1, 值 2, …, 值 n)
```

Insert 语句有几种使用方法，各有特点。下面以 Student 表为例进行介绍。首先，使用下面的 Create Table 语句创建 Student 表：

```
Create Table Student(
   S#  Varchar(10) Constraint PK Primary Key,
   Sname Varchar(20),
   Age int,
   Gender Char(1) DEFAULT 'F'
)
```

下面给出了在 Student 表中插入一条记录的几种方法。

1）给出完整列名表的记录插入

```
Insert Into Student(S#, Sname, Age, Gender) Values('s1', 'John', 20, 'M')
```

由于给出了列名表，因此 Values 子句后的内容就会自动按列名表顺序进行匹配。因为关系模式中的属性集是无序的，同理在基本表中的列集合也是无序的。如果给出了列名表，那么下面变换了列顺序的 Insert 语句的执行结果与上面语句是等价的：

```
Insert Into Student(S#, Age, Gender, Sname) Values('s1', 20, 'M', 'John')
```

2）省略列名表的记录插入

```
Insert Into Student Values('s2', 'Mike', 20, 'M')
```

上述语句在执行时 Values 子句后的值会根据当前 Student 表的列顺序进行匹配。因此这种方式要求用户对 Student 表的列类型和列顺序比较熟悉。但是，如果基本表的列顺序发生了改变，则这种方式极有可能出错，所以在实际的数据库应用系统开发中，不建议使用这种方式在程序中执行 Insert 语句。一般地，这种方式比较适合用户自行调试 SQL 语句时使用。

3）给出部分列名表的记录插入

```
Insert Into Student(S#, Sname) Values('s3', 'Rose')
```

这种记录插入方式只给出了部分列名，其前提是省略的列名上没有 NOT NULL 约束，否则将会报错（原因很简单，新插入的记录在省略的列上会出现 NULL 值从而违背 NOT NULL 约束）。如果省略的列上有 Default 值，则 DBMS 会自动用默认值来填充相应的列。例如，上面的 Insert 语句的执行结果如图 4-8 所示。

S#	Sname	Age	Gender
s3	Rose		F

图 4-8　给出部分列名表的 Insert 语句执行结果

4.3.2　Update 语句

Update 语句的基本格式如下：

```
Update <表名>Set <列名 1> = <值 1>[, <列名 2> = <值 2>,… Where<条件>]
```

Update 语句的功能是将给定<表名>的表中满足<条件>的记录的一个或多个列设置为新值。Update 语句要求至少更新一个列。<条件>是一个由关系运算符以及 not、and、or 等连接而成的逻辑表达式，具体的格式与后面要讨论的 Select 语句相同，将在讨论 Select 语句时详细讨论条件表达式的情况。

以 Student 表为例，下面给出 Update 语句的几个示例。

【例 4-5】 将学生 John 的性别改为 F，年龄改为 23 岁。

```
Update Student Set Gender = 'F', Age = 23 Where Sname = 'John'
```

【例 4-6】 将所有学生的学号前面加上字母 s。

```
Update Student Set S# = 's' + S#
```

4.3.3　Delete 语句

Delete 语句的基本格式如下：

```
Delete From <表名> [Where <条件>]
```

Delete 语句的含义是从<表名>指定的基本表中删除满足给定<条件>的记录。其中，条件表达式与 Update 语句中的格式相同，也将在 4.4 节讨论 Select 语句时集中进行讨论。

【例 4-7】 从 Student 表中删除学号为 s1 的学生。

```
Delete From Student Where S# = 's1'
```

【例 4-8】 从 Student 表中删除所有的学生。

```
Delete From Student
```

注意，Delete 语句只是对记录的操作，不影响基本表的模式定义；而 Drop Table 语句是删除基本表的模式定义(当然里面的记录也随之删除了)。这两条语句一条是 DML 语句，另一条是 DDL 语句，其功能和定义是完全不同的。

4.4　Select 查询

查询是数据库应用系统中最普遍的操作。同时，由于不同用户对于数据查询的需求往往不同，因此 SQL 查询语句要求能够满足不同的查询需求。数据查询在 SQL 中通过 Select 语句来表达。鉴于查询需求的多样化，Select 查询语句支持多种表达方式。这一方面增强了 Select 语句的表达能力，另一方面也使得 Select 查询语句的书写难度增加。Select 查询语句与关系数据模型中的关系代数表达式有着强烈的对应关系，它的求解思路与关系代数表达式的书写类似，因此只要深入理解了关系数据模型和关系代数表达式，那么书写 Select 查询语句就会容易许多。此外，由于 SQL 是声明性语言，所以 Select 查询语句只要表达查询需求即可，无须给出详细的计算过程。

4.4.1　Select 查询的基本结构

相对而言，Select 语句是 SQL 中最为复杂也最难以掌握好的语句。Select 查询语句的完整语法比较复杂，但在实际应用中常用的基本结构是比较清晰的。在本节中，将给出 Select 查询语句的基本结构。掌握了 Select 查询语句的基本结构后，当遇到一些非常特殊的查询需求时，可以再去参考 Select 查询语句的完整语法格式。一般地，在数据库领域中，“SQL 查询”和“Select 查询”这两个术语具有相同的含义，在本书中它们也具有相同的意思。

Select 查询的基本结构如下：

```
Select <列名表>              --指定希望查看的列
From <表名列表>              --指定要查询的表
[Where <条件>]               --指定查询条件
[Group By <分组列名表>       --指定要分组的列
```

```
        [Having <条件>]]          --指定分组的条件
    [Order By <排序列名表>]       --指定如何排序
```

SQL 称为结构化查询语言，是因为 SQL 语句在书写时有着结构化要求。Select 查询语句的基本结构很好地体现了这种结构化特征。例如，Select 子句必须放在语句的开始，Where 子句必须放在 From 子句后面等。SQL 查询语句就是通过 Select、From、Where 等结构化的子句来表达用户的查询(注意，求解过程无须给出，体现声明性语言的特点)。

Select 查询的基本结构包含了若干个子句，这些子句具有不同的目的和功能。下面进行详细说明。

1. Select 子句

Select 给出了用户希望在结果中查看的列名表，例如，“查询选修了‘数据库’课程的学生学号和姓名”这一查询中，“学号”和“姓名”就是用户希望返回的最终结果，所以它们应该放在 Select 子句后的列名表中。

2. From 子句

From 子句给出了要查询的表名，即告诉 DBMS 从什么地方去查询数据。From 子句后可以跟一个表名，也可以跟一个表名列表，表示所查询的数据涉及多个表。例如，假设有学生表 Student 和选课表 SC，“查询学生 John 的选修课程号和成绩”这一查询就涉及了 Student 和 SC 两个表，此时需要将 Student 和 SC 表都放到 From 子句中。在实际应用中，应当根据查询的实际需要确定所要访问的基本表。

3. Where 子句

Where 子句给出了查询的条件。在关系代数中，选择操作是带有条件的，所以 Where 子句中最普遍的情况就是列出了选择的条件。但是，SQL 与关系代数在表达上有差别，在 Select 查询中，连接的条件(包括自然连接和θ连接)也是放在 Where 子句中的，这一点需要特别注意。

举个例子，学生关系和选课关系的自然连接在关系代数中表达为 Student$\bowtie$SC，但在 SQL 中，必须在 Where 子句中给出它们的自然连接条件：Student.S# = SC.S#。

4. Group By 子句和 Having 子句

Group By 子句和 Having 子句通常是在一起使用的，它们对应着关系代数中的分组/聚集操作。在分组/聚集操作中，存在着分组属性和聚集属性，其中分组属性是用于作为分组依据的属性，而聚集属性则是分组之后用于计算聚集函数值的属性。在 Select 查询中，由于聚集函数值是需要最终作为结果返回的，所以放在 Select 子句中。分组属性则必须放在 Group By 子句中。Having 子句给出了筛选分组结果的条件，它通常包含一个涉及聚集函数的条件表达式，如 COUNT(*)>10。Having 子句用来返回特定的分组，如 COUNT(*)>10 返回记录数在 10 以上的分组。需要注意的是，在使用 Having 子句时必须先给出 Group By 子句(先有分组再有对分组的筛选)，但在使用 Group By 子句分组时如果要返回所有的分组，则可以省略 Having 子句。

5. Order By 子句

Order By 子句用于对查询结果的排序。它和附加关系代数操作中的排序操作语义基本一致，除了 Order By 可以指定升序和降序之外。Order By 后面可以跟一个或多个列名，分别表示第一排序字段、第二排序字段……以此类推，并且对于每个字段都可以分别指定排序的顺序(升序还是降序)。

大多数情况下，Select 查询语句与关系代数表达式之间是一一对应的，但由于 SQL 并非和关系代数一样基于集合论，因此在某些情况下 Select 查询与关系代数表达式很难进行对应，尤其当 Select 查询返回重复结果集时。下面给出了 Select 查询与关系代数表达式之间的对应示例。

【例 4-9】 设有 Student(S#, Sname, Age, Gender)和 SC(S3, C#, Score)，求选修了 C1 课程并且成绩高于 90 分的学生姓名。

该查询的关系代数表达式求解结果为

$$\pi_{\text{Sname}}(\sigma_{\text{C\# = 'C1'} \wedge \text{ Score} > 90}(\text{Student} \bowtie \text{SC}))$$

对应的 Select 查询结果为：

```
Select Sname
From Student, SC
Where Student.S# = SC.S# and C# = 'C1' and Score > 90
```

4.4.2 Select 基本查询

在这一节中，首先讨论只涉及单个表的 Select 基本查询。下面以学生表 Student(S#, Sname, Age, Gender)为例介绍 Select 基本查询。

1. 查询全部记录

返回一个表的全部记录一般用 Select * From <表名>。

【例 4-10】 查询所有的学生信息。对应的 SQL 语句为：

```
Select * From Student
```

其中，*表示返回全部的列。

上面的 SQL 语句与 Select S#, Sname, Age, Gender From Student 结果相同。

2. 返回特定的列

如果查询结果不需要返回全部的列，则直接将要返回的列名表放在 Select 子句即可。

【例 4-11】 查询所有学生的学号和姓名。对应的 SQL 语句为：

```
Select S#, Sname From Student
```

3. 使用别名

在返回查询结果时，Select 语句可以使用关键字 As 为返回的结果列赋别名。别名在实

际应用中是非常有用的。例如，基本表的列名通常是英文的，如果外部的数据库应用程序是中文的，则最好能够将数据库查询结果转换为中文字段输出以提高界面的用户友好性。此时，就可以使用别名将返回的英文列名重命名为中文名称。仍以“查询所有学生的学号和姓名”为例，下面的语句为返回的结果列赋予中文别名：

```
Select  S# AS 学号, Sname AS 姓名 From Student
```

如果别名包含空格，则需要用双引号将别名括起来，像下面的形式：

```
Select  S# AS "Student Number" From Student
```

4. 使用表达式

在关系代数中，表达式主要用于扩展投影操作。扩展投影允许将一个表达式作为投影的结果。Select 语句支持扩展投影，它允许在 Select 子句中使用表达式来作为返回结果。

【例 4-12】 查询所有学生的学号、姓名和出生年份，返回两列信息，其中一列是“学号_姓名”，另一列是“出生年份”。这一查询对应的 SQL 语句为：

```
Select S# + '_' + Sname AS 学生, Year(Getdate()) －Age AS 出生年份 From Student
```

上面的语句中，+表示字符串连接操作，Getdate()和 Year()都是 Microsoft SQL Server 支持的函数，分别返回当前日期和年份。在 SQL 查询中，可以使用的表达式有字符串表达式、算术表达式和函数表达式等。字符串表达式是指由字符串连接操作形成的表达式，算术表达式是涉及+、–、*、/等算术操作的表达式，函数表达式是由函数构成的表达式。在实际查询中，这几类表达式一般情况下都是综合使用的，也不必特意去区分表达式的类型。

在所有表达式中，函数表达式的功能最强，这是因为 DBMS 通常都会支持一个较大的函数集合。常见的函数包括日期函数、字符串函数、聚集函数等。以 Microsoft SQL Server 为例，常用的日期函数如下。

Getdate()：返回当前日期，一般是系统日期。

Year()：返回给定日期中的年。

Month()：返回给定日期中的月。

Day()：返回给定日期中的日。

常用的字符串函数如下。

Str()：将数值数据转换为字符串。

Left()：返回给定字符串中左边指定长度的字符串。

Replace()：字符串替换。

聚集函数在各个 DBMS 中一般都一样，这是因为在关系代数中对聚集操作做了规定，即求和(Sum)、求均值(Avg)、求最大值(Max)、求最小值(Min)以及计数(Count)。聚集函数与其他函数的区别在于：聚集函数的参数是一个值的集合，而其他函数的参数是单个值。因此，聚集函数通常和分组操作 Group By 一起使用，用于对分组的结果集进行统计。

5. 查询特定的记录

前面讨论的几种 Select 基本查询都没有用到 Where 子句。Where 子句一般在指定了查询条件时使用，即返回满足特定条件的记录。

【例 4-13】 查询 20 岁以上学生的学号和姓名，对应的 SQL 语句为：

```
Select S#, Sname From Student Where Age > 20
```

如果没有指定 Where 子句，DBMS 将返回表中的全部记录，否则就返回满足 Where 子句指定条件的记录。Where 子句后的条件表达式在 SQL 查询中有时会非常复杂，因为它可能包含多种关系运算符。这些运算符包括算术运算符(>、<、>=、<=、=、<>)、IN、LIKE、EXISTS、IS NULL 和 IS NOT NULL。下面给出这些关系运算符的示例，其中 EXISTS 通常用在嵌套查询中，所以放在后面嵌套查询中讨论。IN 一般也用在嵌套查询中，但也可以不用在嵌套查询中。

【例 4-14】 查询学号为 s001、s003、s006 和 s008 的四个学生的信息。这一查询可以用 IN 运算符来表示，如下：

```
Select * From Student Where S# IN ('s001', 's003', 's006', 's008')
```

IN 运算符的基本用法为<*A*> IN <*S*>，其中<*A*>可以是字段名，也可以是表达式，但它必须是单个值，不能是一个集合。<*S*>是一个值集。在上面的示例中，这个值集是一个常量。在实际查询中，也可以用 Select 查询来返回一个值集，例如，Select S# From Student 返回了全体学生的学号集合，它也可以用作<*S*>。用 SQL 查询语句来作为 IN 运算符中的<*S*>时就出现了嵌套查询，会在 4.4.4 节中进行详细讨论。

【例 4-15】 查询缺少年龄数据的学生。这一查询可表示为：

```
Select * From Student Where Age IS NULL
```

上述查询使用了 IS NULL 运算符。注意，Where 子句中的条件不能写成 Age = NULL。在数据库中，NULL 值(空值)是一种非常特殊的值。一般情况下，NULL 值表示 Unknown 语义，即没有值，而且也不知道它的类型。由于 DBMS 在编译 SQL 语句时对于>、<、>=、<=、=、<>等运算符要求满足类型一致性(即两个操作数的类型必须是相同或者可以相互转换的)，因此对于 NULL 值不能直接用>、<、>=、<=、=、<>这些运算符去比较，否则会出现 SQL 语法错误。NULL 值的比较必须用 IS NULL 和 IS NOT NULL，前者表示值为空，后者表示值非空。

【例 4-16】 查询姓赵的学生信息。这一查询可表示为：

```
Select * From Student Where Sname LIKE '赵%'
```

上述查询使用了 LIKE 运算符。LIKE 运算符在 SQL 查询中专门用于表达模糊查询，即部分匹配查询。与在 Windows 中模糊查找文件类似，LIKE 查询也使用一些通配符来实现模糊查询，如上述示例中的%。表 4-2 给出了在 Microsoft SQL Server 中 LIKE 查询可使用的通配符及含义。

表 4-2　LIKE 查询可使用的通配符及含义

通配符	含义
%	任意长度的字符串
_	单个字符
[…]	指定范围(如[a-f])或集合(如[abcdef])中的任何单个字符
[^..]	不属于指定范围(如[a-f])或集合(如[abcdef])的任何单个字符

下面再给出 LIKE 查询的几个示例，使大家可以了解这些通配符在模糊查询中的作用。

【例 4-17】 查询姓名的第一个字母为 R 并且倒数第二个字母为 S 的学生。

```
Select * From Student Where Sname LIKE 'R%S_'
```

【例 4-18】 查找姓名以 arsen 结尾且以介于 K 与 T 之间的任何单个字符开始的学生，如 Karsen、Larsen、Parsen 等。

```
Select * From Student Where Sname LIKE '[K-T]arsen'
```

【例 4-19】 查找姓名以 We 开始且其后的字母不为 f 的所有学生。

```
Select * From Student Where Sname LIKE 'We[^f]%'
```

Where 子句后如果有多个条件，则可以用 and、or 或 not 进行连接。

【例 4-20】 查询姓名以字母 R 开头并且年龄大于 19 岁的学生信息。

```
Select * From Student Where Age > 19 and Sname LIKE 'R%'
```

6. 去除重复记录

由于 SQL 基于包而非集合，因此在 SQL 查询中会返回重复结果。关系代数操作是基于集合论的，所以在关系代数表达式中是不会出现重复结果的。SQL 之所以支持重复记录，是因为有的应用有这样的需求，但是有的时候也希望返回的结果中去除重复记录，尤其当进行一些统计操作时(如统计学生人数)。SQL 在 Select 子句中引入了 Distinct 选项来允许用户在结果集中去除重复记录。

以学生选课表 SC(S#, C#, Score)为例，如果希望得到选了课的学生学号，由于一个学生可能选了多门课，所以用下面的 Select 语句将返回许多重复的学号：

```
Select S# From SC
```

进一步，如果希望统计目前选了课的学生人数，则需要用计数函数 Count 来统计返回的学号数：

```
Select Count(S#) From SC
```

这个结果是错误的，因为返回的学号中包含了很多重复值。正确的 Select 语句为：

```
Select Distinct S# From SC
Select Count(Distinct S#) From SC
```

Distinct 选项用来在结果中去除重复记录。需要注意的是，Distinct 选项是针对记录进

行运算的，即去除的是重复记录，而不是某个列的重复值。

7. 查询结果的排序

查询结果的排序使用 Order By 子句。其用法很简单，就是把要排序的列名放在 Order By 子句中即可。Order By 子句提供了两种排序方式：升序和降序，分别用关键字 ASC 和 DESC 表示。如果省略排序方式，则默认为升序。

【例 4-21】 查询所有学生信息并将结果按年龄升序排列，再按姓名降序排列。

```
Select * From Student Order By Age, Sname DESC
```

或

```
Select * From Student Order By Age ASC, Sname DESC
```

上述示例中由于 Age 排序方式是升序，所以在 Order By 子句中可以省略 ASC 关键字。当 Order By 后有多个排序字段时，这些字段的顺序是不能随意调整的。第一个字段代表第一排序字段，以此类推。

8. 分组聚集查询

分组聚集查询通过 Group By 子句来实现，如果必要也会用到 Having 子句，但使用 Having 子句的前提是必须先使用 Group By 子句。如前所述，聚集函数输入一个值的集合，返回一个单一的值作为结果。表 4-3 给出了 SQL 中的聚集函数及其含义。除了 Count 的参数可以有两种类型外，其余函数的参数均为列名。

表 4-3　SQL 中的聚集函数及其含义

聚集函数	含义	聚集函数	含义
Count(列名)	对一列中的值计数	Avg(列名)	求一列值的平均值
Count(*)	计算记录数目	Max(列名)	求一列值的最大值
Sum(列名)	求一列值的总和(数值)	Min(列名)	求一列值的最小值

【例 4-22】 求学生的总人数和平均年龄。这一查询直接在全部学生记录上进行统计运算，相应的 SQL 语句为：

```
Select Count(*) As 总人数, Avg(Age) As 平均年龄 From Student
```

在实际应用中，很多时候需要先对整个记录集进行分组，然后分别对各组进行统计，例如，“统计学生中男生和女生的人数”就要求先将学生记录按性别分成两个集合，然后分别统计男生集合中的记录数和女生集合中的记录数。这种方式的查询称为分组聚集查询。分组聚集查询通常包含两个步骤的计算：第一步按给定字段(分组字段)分组；第二步分别在各组中计算聚集函数的值并返回。

【例 4-23】 统计学生中男生和女生的人数。

```
Select Gender As 性别, Count(S#) As 人数 From Student Group By Gender
```

在分组聚集查询中，用于分组的列称为“分组属性”或“分组字段”，作为聚集函数

参数的列称为“聚集属性”或“聚集字段”。上面的示例中，Gender 是分组属性，而 S#是聚集属性。特别需要注意的是，分组聚集查询在书写时要求出现在 Select 子句中的列除了聚集属性之外必须全部写在 Group By 子句中。例如，上面的示例中如果在 Select 子句中加入 Age 字段，则下面的 SQL 语句是不符合分组聚集查询的书写要求的：

```
Select Age, Gender, Count(S#) As 人数 From Student Group By Gender
```

正确的写法是：

```
Select Age, Gender, Count(S#) As 人数 From Student Group By Age, Gender
```

实际上原因也很容易理解。如果按第一种写法，在执行分组聚集查询时，学生集合先按 Gender 分组，然后计算 Count(S#)，但是最后返回时还要求提供一个 Age 值，由于一个分组中的 Age 值可能有多个(不同学生的年龄可能不同)，所以此时 DBMS 就无法决定到底在返回的结果中使用哪个 Age 值。而第二种写法中，每个分组中的 Age 值和 Gender 值是唯一的，所以就不会出现无法求解的问题。

此外，在执行分组聚集查询时，NULL 值是作为一个新的值来处理的，也就是说如果分组字段中出现 NULL 值，那么所有 NULL 值的记录将会被分为一个组。

以例 4-23 为例，假设现在 Student 表中有如图 4-9(a)所示的记录，执行示例中的分组聚集查询语句后的结果显示在图 4-9(b)中。可以看到，由于 Gender 列上存在 NULL 值，在求解过程中整个记录集被分成了三个组并分别计算了聚集函数 Count(S#)。

S#	Sname	Age	Gender
s1	Rose	18	F
s2	John	19	
s3	Mary	20	M
s4	Mike	20	M

(a) Student 表

性别	人数
F	1
M	2
	1

(b) 分组聚集查询结果

图 4-9　Student 表按性别分组的结果

在分组聚集查询中，如果希望最后的结果只返回特定的分组运算结果，则需要用 Having 子句来指定分组筛选的条件。如果 Group By 子句不带 Having 子句，则在运算之后系统将返回所有分组的计算结果。

【例 4-24】 查询不同年龄的学生人数，并返回人数在 5 以上的结果。

```
Select Age,Count(*) as 人数 From Student Group By Age Having Count(*) > 5
```

在这一查询中，用 Having 子句对最终的分组结果进行筛选，使得最终只返回了那些人数大于 5 的分组结果。Having 子句后面必须是带聚集函数的条件表达式(不带聚集函数的条件表达式放在 Where 子句中)。正因为 Having 子句后的条件表达式带有聚集函数，所以它们不能像通常的条件表达式一样放到 Where 子句中。最根本的原因在于两者的底层执行方式不同，Where 子句后的条件在 SQL 语句执行时是对表中的每一条记录进行条件检查，而 Having 子句后的条件是对分组后的结果执行条件检查。

Having 子句中的聚集函数也可以跟 Select 子句中的聚集函数不同。

【例 4-25】 设有学生表 Student(S#, Sname, Class, Age)，查询人数在 60 以上的班级的学生平均年龄。

```
Select Class, Avg(Age) From Student Group By Class Having Count(*) > 60
```

9. Top *k* 查询

Top *k* 查询是指返回查询结果中前 *k* 条记录，如“查询全班总分排名前 10 的同学”。这类查询在前面的 SQL 查询语句中均很难实现。对于这一特殊查询，Microsoft SQL Server、Oracle、MySQL 有着不同的实现方法。

Microsoft SQL Server 的语法：Select Top <number><column>From<table>。

Oracle 的语法：Select <column> From <table> Where Rownum < = <number>。

MySQL 的语法：Select <column> From <table> Limit number。

【例 4-26】 给定学生表 Student(S#, Sname, Age, Dept)，返回学生中年龄最小的前 10 名同学姓名。

(1) Microsoft SQL Server 查询。

```
Select Top 10 Sname From Student Order By Age
```

在 Microsoft SQL Server 中可以使用 Top *k* 返回前 *k* 条记录，该查询先执行 Order By 运算，然后返回 Top 10 记录。

(2) Oracle 查询。

```
Select Sname From (Select Sname From Student Order By Age) S Where Rownum
< = 10
```

注意，上述查询在 Oracle 中不能写成：

```
Select Sname From Student Order By Age Where Rownum < = 10
```

这是因为 Where 子句的运算等级在 Oracle 中高于 Order By，所以会先执行 Rownum 比较操作，然后执行 Order By，导致结果错误。

(3) MySQL 查询。

```
Select Sname From Student Order By Age Limit 10
```

上述 MySQL 查询语句在执行时并不是对整个记录集排序后再输出前 10 条记录，而是找到了最小的前 10 条记录后即结束执行，所以，当年龄存在相同值时，上述查询返回的结果可能会出现不同。如果想得到稳定的排序输出结果，一种方法是使用嵌套查询：Select Sname From (Select Sname From Student Order By Age) S Limit 10，即对整个记录集进行排序后再返回前 10 条记录。另一种方法是将额外的字段加到 Order By 列表中一起排序，如学号：Select Sname From Student Order By Age, S# Limit 10。

10. Some、Any 和 All 查询

Some、Any 和 All 这三个查询通常和嵌套查询相结合，用于返回满足某个子查询任一

结果的记录。Any 和 Some 关键字是同义词，表示子查询中任意一条记录满足条件即可，All 表示满足子查询中的所有记录。

【例 4-27】 查询“英才班”中年龄比“实验班”所有学生年龄都要小的学生学号和姓名。

```
Select S#, Sname From Student Where Class = '英才班' and Age < All (Select
Age From Student Where Class = '实验班')
```

【例 4-28】 查询“英才班”中年龄大于“实验班”中某个学生年龄的学生学号和姓名。

```
Select S#, Sname From Student Where Class = '英才班' and Age > Any (Select
Age From Student Where Class = '实验班')
```

或者

```
Select S#, Sname From Student Where Class = '英才班' and Age > Some (Select
Age From Student Where Class = '实验班')
```

4.4.3 连接查询

连接查询是指涉及两个以上基本表的查询。在关系代数中，连接操作有自然连接和θ连接之分，但在 SQL 中这两种连接操作都统一用 Select 语句表达。SQL 连接查询返回连接的表中满足连接条件(对于自然连接是等值比较条件，对于θ连接是不等值比较条件)的记录。

图 4-10 给出了学生表 Student、选课表 SC 和课程表 Course 的示例，下面以这三个表为例讨论连接查询。

Student(S#, Sname, Age, Gender)

S#	Sname	Age	Gender
s1	Rose	18	F
s2	John	19	F
s3	Mary	20	M
s4	Mike	20	M

SC(S#, C#, Score)

S#	C#	Score
s1	c1	80
s1	C2	85
s3	c2	90
s4	c3	90

Course(C#, Cname, Credit)

C#	Cname	Credit
c1	DB	3
c2	MATH	3
c3	C	3
c4	AI	2

图 4-10　Student 表、SC 表和 Course 表

【例 4-29】 查询学生的学号、姓名和所选课程号，相应的 SQL 语句为：

```
Select Student.S#, Sname, SC.C# From Student, SC Where Student.S# = SC.S#
```

该查询返回的学号和姓名来自 Student 表，但学生选课的信息来自 SC 表，所以此查询须涉及 Student 表和 SC 表，是一个连接查询。在查询中涉及多个表时，一般都是自然连接查询的语义，这是因为自然连接在实际应用中最为普遍。与关系代数中的自然连接操作不同，SQL 中的连接查询要求将连接条件显式地书写在 Where 子句中。以上述示例为例，根据自然连接的语义——“连接表的所有公共字段上执行等值比较操作”，得连接条件为 Student.S# = SC.S#。

在连接查询中，如果需要引用某个公共字段(如 S#)，必须要加上表名作为前缀。例如，

在上面的查询示例中，S#同时出现在 From 子句后的 Student 表和 SC 表中，因此在 Select 语句中引用 S#时必须加上表名前缀。但是 Sname 是唯一出现在 Student 表中的字段，所以不加表名前缀也不会影响语句的执行。

如果表名比较长，在 SQL 查询中可以使用别名。表的别名可以和表名一样在 SQL 语句的各个子句中使用。

【例 4-30】 查询学生 Rose 所选的课程号和课程名。

```
Select B.C#, C.Cname From Student A, SC B, Course C
Where A.S# = B.S# and B.C# = C.C# and A.Sname = 'Rose'
```

此查询涉及三个表，这是因为学生姓名信息出现在 Student 表中，学生的选课信息来自 SC 表，同时课程名称信息来自于 Course 表。Where 子句中的 A.S# = B.S# and B.C# = C.C# 为三个表的连接条件，A.Sname = ‘Rose’是选择条件。上面的查询中使用了表别名，分别将 Student、SC 和 Course 赋予了别名 A、B 和 C。从示例中可以看到，在 From 子句中将表名后直接加上空格和别名即可对表进行重命名。表的别名可以和表名同时在 SQL 语句中使用。

连接查询可以和 4.4.2 节的 Select 基本查询相结合，例如，和 Gourp By、Order By 子句等相结合，从而可以回答一些复杂的查询。

【例 4-31】 查询男学生的学号、姓名和所选的课程数，结果按学号升序排列。该查询对应的 SQL 语句为：

```
Select A.S#, B.Sname, Count(B.C#) as c_count
From Student A, SC B
Where A.S# = B.S# and A.Gender = 'M'
Group By A.S#, B.Sname
Order By Student.S#
```

该查询需要统计课程数，所以需要用到分组聚集查询(Group By)。同时，它要求对结果排序，所以需要用 Order By 子句。另外，它还需要查询学生的姓名和选课信息，这两者分别来自 Student 表和 SC 表，所以需要用连接查询。

4.4.4　嵌套查询

嵌套查询是 Select 查询中最复杂的查询。在讨论关系代数时，曾提到“关系代数操作的可嵌套性”，即一个关系代数操作的结果可以作为另一个关系代数操作的参数，从而形成多个关系代数操作的嵌套表达式。关系代数的高表达能力和复杂性主要是由它的可嵌套性决定的。SQL 嵌套查询对应了关系代数的可嵌套性。同理，SQL 查询的高表达能力和复杂性也主要由嵌套查询决定。

嵌套查询是指一个 Select 语句中又包含了其他的 Select 语句。把外层的 Select 语句称为“父查询”，把内层的 Select 语句称为“子查询”。嵌套查询可分为三种类型：无关子查询、相关子查询和联机视图。

1. 无关子查询

无关子查询是指父查询与子查询相互独立，子查询语句不依赖父查询中返回的任何

记录。无关子查询的特点是子查询语句单独拿出来也是符合 SQL 语法的，也可以单独执行。

【例 4-32】 查询没有选修课程的所有学生的学号和姓名。

```
Select S#, Sname From Student Where S# NOT IN (Select Distinct S# From SC)
```

此查询包含了子查询 Select Distinct S# From SC，它返回了选修课程的学生学号集合。该子查询与外层的查询无依赖关系，可以单独执行。

【例 4-33】 查询年龄大于平均年龄的学生。

```
Select * From Student Where Age > (Select Avg(Age) From Student)
```

此查询用一个子查询返回了学生的平均年龄，该子查询也是可以单独执行的，与外层的父查询不存在执行时的依赖关系。

2. 相关子查询

相关子查询与无关子查询是相对的。在相关子查询中，内层的子查询在执行时需要引用外层父查询返回的结果，因此相互之间存在依赖性。相关子查询的另一个特点是，它是不能单独运行的。

【例 4-34】 查询选修课程的学生学号和姓名。此查询当然也可以用一般的连接查询来回答，下面给出用相关子查询来回答的结果：

```
Select S#, Sname From Student
Where EXISTS (Select * From SC Where SC.S# = Student.S#)
```

该查询使用了 EXISTS 关系运算符。EXISTS 后面通常跟一个子查询，当子查询返回结果非空时，EXISTS 返回结果为 True，否则返回 False。上面查询的求解思路是：对于每一条 Student 记录，根据该记录中的学号去 SC 表中进行查找，如果在 SC 中有匹配结果，则说明该学生选过课，EXISTS 返回 True，否则说明没有选课。这个示例中的子查询单独拿出来执行是不符合 SQL 语法的。

3. 联机视图

联机视图(Online View)是指子查询出现在 From 子句中，即子查询的结果构成了一个虚拟的表。这与后面要讨论的视图(View)概念相似。但视图是数据库中确实存在的对象，而联机视图只是在 SQL 执行时才临时建立的一种“视图”，这是“联机视图”这一名称的含义。

联机视图是关系代数操作的可嵌套性最直接的实现。在关系代数中，任何一个关系代数表达式的结果都是一个关系，因此可以作为其他关系代数操作的参数。而关系代数操作的参数就是关系，它在 Select 语句中对应着 From 子句后的内容。因此，联机视图实际上就是用一个 Select 语句(子查询)返回一个关系作为另一个 Select 语句(父查询)的参数。联机视图一般都需要重命名。有了别名之后在 Select 语句中可以和其他表一样使用。

【例 4-35】 查询只选修 1 门或 2 门课程的学生学号、姓名和课程数。

```
Select S.S#, count_c#
```

```
From (Select S#, Count(S#) as count_c# From SC Group by S#) SC2,
     Student S
Where SC2.S# = S.S# and (count_c# = 1 or count_c# = 2)
```

上述查询中定义了一个联机视图 SC2，它的模式可以理解为 SC2(S#, count_c#)，其中 S#是学号，count_c#是该学生选修的课程数。有了这一虚拟表，后面的查询就转变成了 SC2 和 Student 的连接查询与选择查询。

4.4.5　查询结果的拼接

查询结果的拼接是指关系代数中的并、交、差等两个结果集之间的集合运算。并操作在 SQL 中通过 Union 实现，但由于 SQL 允许重复元素，因此 SQL 中还引入了包的并操作，即 Union All。Union 和 Union All 的区别是 Union 返回集合运算结果(无重复元素)，而 Union All 返回包运算结果(保留重复元素)。交操作在 SQL 中通过 Intersects 实现。差操作通过 Minus 实现。关系代数中的除操作在 SQL 中没有专门的实现方式，一方面是因为除操作在实际应用中很少使用，另一方面是因为除操作本身不是关系代数的基本操作，因此它的功能可以通过其他的操作来实现。

1. 查询结果的并

查询结果的并可以用 Union 和 Union All 实现。究竟用 Union 还是 Union All 需要考察查询的语义。Union 和 Union All 在书写时要注意参与并运算的两个集合必须是具有相同类型的记录，这与关系代数中的要求是完全相同的。交和差操作也须满足同样的要求。

【例 4-36】 查询课程平均成绩在 90 分以上或者年龄小于 16 岁的学生学号。

```
(Select S# From Student Where Age<16)
Union
(Select S# From (Select S#, Avg(Score) From SC Group By S# Having
   Avg(Score)>90) SC2)
```

在上面的查询中，由于平均成绩在 90 分以上和年龄小于 16 岁的学生之间有重叠，但不需要保留多个重复的学号，因此应当用 Union 而不是 Union All。

2. 查询结果的交

交操作用 Intersects 表示，返回两个查询结果的交集。

【例 4-37】 查询课程平均成绩在 90 分以上并且年龄小于 16 岁的学生学号。

```
(Select S# From Student Where Age<16)
Intersects
(Select S# From (Select S#, Avg(Score) From SC Group By S# Having
   Avg(Score)>90) SC2)
```

3. 查询结果的差

差操作用 Minus 表示，对于两个查询记录集 *A* 和 *B*，*A* Minus *B* 返回属于 *A* 但不属于 *B* 的记录。

【例 4-38】 查询未选修课程的学生学号。

```
(Select S# From Student)
Minus
(Select Distinct S# From SC)
```

4.5 数 据 控 制

数据控制语言(DCL)用于完成用户权限的管理。基本的数据控制语句包括 Grant、Revoke 和 Deny 语句，分别完成用户权限的授予、废除和彻底废除。其中 Grant 和 Revoke 语句在 Oracle、MySQL、Microsoft SQL Server 中都支持，但是 Deny 语句只有 Microsoft SQL Server 支持。

4.5.1 Grant 语句

Grant 语句实现存取权限的授予。在关系数据库中，存取权限包括数据对象和操作类型两个部分，例如，“学生表上的删除权限”中的数据对象是“学生表”，操作类型是“删除”。表 4-4 列出了关系数据库中基本的数据对象和操作类型。

表 4-4　SQL 存取权限包括的数据对象和操作类型

类别	数据对象	操作类型
模式	外模式	建立、修改、删除、查看
	概念模式	建立、修改、删除、查看
	内模式	建立、删除、查看
数据	表	查看、插入、修改、删除
	属性	查看、插入、修改、删除

根据所授予的权限的不同，Grant 语句有两种用法。

1. 授予语句权限

授予语句权限相应的 Grant 语句格式为：

```
Grant { ALL | <语句 1> [ ,…<语句 n> ] } To <用户 1> [ ,…<用户 n> ]
```

例如，下面的 Grant 语句把 Create Table 与 Alter Table 语句的执行权限授给了用户 Mary 和 John：

```
Grant Create Table, Alter Table To Mary, John
```

如果要把全部的语句权限都授给用户，则用 Grant All。例如，下面的语句把全部语句权限都授给了用户 Rose：

```
Grant All To Rose
```

2. 授予对象权限

授予对象权限相应的 Grant 语句格式为：

```
Grant{ALL|<权限 1>[,…<权限 n>]} On <对象> To <用户 1>[,…<用户 n>]
```

它表示把<对象>上的<权限 1>, …, <权限 *n*>授给<用户 1>, …, <用户 *n*>。例如，下面的语句把 Authors 表上的 Insert、Delete、Update 权限授给了用户 Mary、John 和 Tom：

```
Grant Insert, Delete, Update On Authors To Mary, John, Tom
```

4.5.2 Revoke 语句

Revoke 语句与 Grant 语句的功能正好相反，它用于废除已授予用户的权限。由于 Grant 授权时有语句权限和对象权限之分，所以 Revoke 权限时对于这两种权限的语句语法也不一样。

1. 废除语句权限

废除语句权限相应的 Revoke 语句格式为：

```
Revoke { ALL | <语句 1> [ ,…<语句 n> ] } From <用户 1> [ ,…<用户 n> ]
```

例如，下面的 Revoke 语句废除了用户 Joe 拥有的 Create Table 语句权限：

```
Revoke Create Table From Joe
```

2. 废除对象权限

废除对象权限相应的 Revoke 语句格式为：

```
Revoke{ALL|<权限 1>[,…<权限 n>]} On <对象>} From <用户 1>[,…<用户 n>]
```

例如，下面的 Revoke 语句废除了用户 Mary 拥有的对基本表 Authors 的 Insert 和 Delete 权限：

```
Revoke Insert, Delete On Authors From Mary
```

4.5.3 Deny 语句

由于目前一般的商用 DBMS 都采用基于角色的访问控制机制，一个用户有可能属于多个角色，因此有时候虽然用 Revoke 语句废除了用户拥有的显式的权限，但并不能阻止用户从他所属于的角色那里通过继承得到权限。这样一来，仅用 Revoke 语句实际无法彻底废除用户的权限。举个例子，假设将 Employee 表上的 Select 权限从用户 Mary 上面废除，从而使 Mary 不能再查看该表。但当 Mary 是 Administration 角色的成员时，如果以后将 Employee 上的 Select 权限授予了 Administration，则 Mary 可以通过 Administration 角色获得 Employee 表上的 Select 权限。

Deny 语句的引入解决了 Revoke 语句不能彻底废除用户权限的问题。Deny 语句的作用是彻底废除用户的某个权限。一旦采用 Deny 语句废除了用户的某个权限后，该用户就不再能通过角色继承的方式获得此权限，这样就达到了彻底废除用户权限的目的。

1. 废除语句权限

废除语句权限相应的 Deny 语句格式为：

```
Deny{ALL|<语句 1>[,…<语句 n>]} To <用户 1>[,…<用户 n>]
```

例如，下面的语句废除了用户 Mary 和 John 执行 Create Table 和 Drop Table 的语句权限：

```
Deny Create Table, Drop Table To Mary, John
```

2. 废除对象权限

废除对象权限相应的 Deny 语句格式为：

```
Deny{ALL|<权限 1>[,…<权限 n>]} On <对象> To <用户 1>[,…<用户 n>]
```

例如，下面的 Deny 语句废除了用户 Rose 对 Employee 表的 Select 和 Update 权限：

```
Deny Select, Update On Employee To Rose
```

4.6 视　　图

目前流行的 SQL 数据库系统（如 Oracle、Microsoft SQL Server 等）都遵循 ANSI/SPARC 体系结构。其中，外模式以及外模式-模式的映像关系是通过视图来实现的。本节主要介绍 SQL 中对视图的支持和相关的操作语句。

4.6.1 视图的概念

视图是从一个或几个基本表中导出的虚拟表，它具有和表一样的逻辑结构定义，但没有实际的数据存储。从使用的角度，视图可以和表一样操作，但视图没有相应的物理存储文件，而每个基本表都有相应的物理存储文件。

视图是外模式在 SQL 数据库中的实现。图 4-11 给出了 SQL 数据库与 ANSI/SPARC 体

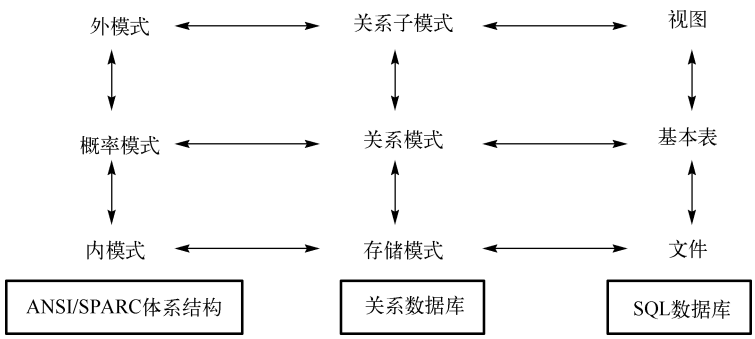


图 4-11　视图与数据库三层体系结构

系结构之间的联系。可以看到，在 SQL 数据库中，视图对应着外模式，基本表对应着概念模式，文件对应着内模式。因此视图的功能与外模式在 ANSI/SPARC 体系结构中的功能是类似的。但是，由于视图是外模式概念在 SQL 数据库中的特定实现，因此它与理论上的外模式概念有些差别。ANSI/SPARC 体系结构包含了三级模式结构和两级映像，而在 SQL 数据库中，视图不仅定义了外模式，还定义了外模式-模式映像关系。

4.6.2　视图的作用

视图在 SQL 中具有十分重要的作用。视图的作用大致可归纳为以下几个方面。

1. 保证逻辑数据独立性

逻辑数据独立性是指当概念模式发生变化时只需要修改外模式-模式映像关系就可以保证外模式不变，从而保证建立在外模式之上的应用程序不需要做任何修改。具体到 SQL 数据库上，当基本表发生变化时，只需要修改视图，就可以保证建立在视图之上的应用程序不必做任何修改。

图 4-12 显示了视图与逻辑数据独立性的一个示例。当基本表发生修改时，只需要修改视图定义，就可以保证视图的逻辑结构不变，从而保证外部应用程序不需要做任何修改。

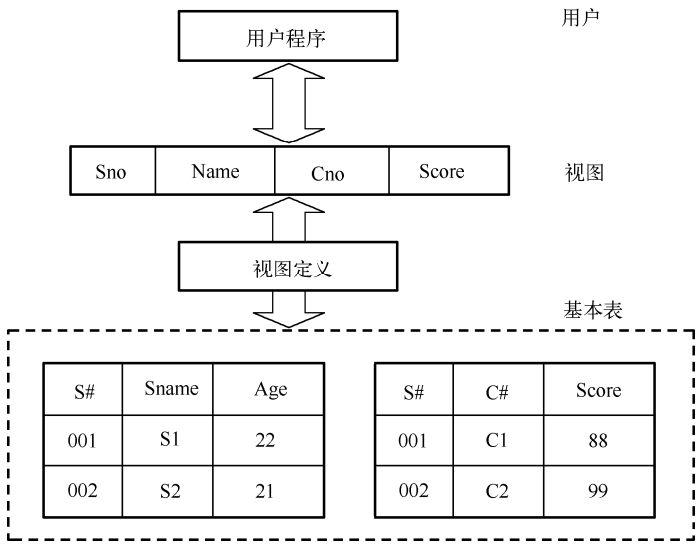


图 4-12　视图与逻辑数据独立性示例

2. 简化数据

一般地，数据库都是被多个用户(应用程序)共享的，但是不同的用户所需要的数据是不一样的。例如，在一个企业数据库中，生产部门可能只需要原材料信息和产品信息，而财务部门则只关心企业的财务往来信息。通过视图，可以为不同的用户定义他们所需的数据，从而简化用户眼中的数据，使用户可以集中于所关心的数据上。

从数据库程序员的角度来看，通过视图简化数据也可以简化程序员的工作，使程序员在编程时可以把注意力集中到自己关心的数据上。

3. 提供不同的数据呈现方式

通过视图可以使同一数据库对不同用户提供不同的数据呈现方式。不同的用户数据需求可以通过视图来进行抽象。例如，有的用户需要数据库提供统计信息，那么可以在视图中通过使用分组聚集查询语句定义包含统计结果的视图。视图的引入使得数据库可以以多种方式进行输出，从而满足不同应用程序的需求。

4. 增强数据安全性

数据库安全中最重要的一点就是数据的保密性，即不能让未授权的用户看到保密信息。在一个数据库应用系统中，不同用户对于数据库的权限通常是有差别的。例如，Web查询用户与内部管理用户所看到的数据内容一般是不同的。通过视图，可以为不同的用户定义他们应该看到的数据，并且通过 SQL 中的数据控制语言(DCL)授权用户只能访问它所定义的视图(而不能直接访问基本表)。这样一来，不同的用户就只能访问到他们能够看到的数据(由视图定义)，并且即使用户账户泄露也不会造成其他数据的泄露，因为用户只有访问其相应视图的权限。

4.6.3 Create View 语句

视图的定义通过 Create View 语句来实现。Create View 也属于 DDL 语句。Create View 与 Create Table 语句的区别在于 Create Table 语句用于创建基本表(即概念模式)，而 Create View 用于创建视图(即外模式)。

Create View 语句的基本格式如下：

```
Create View <视图名>[(列名 1, 列名 2, …)] AS <查询> [With Check Option]
```

其中，<查询>是一个 Select 语句，指明视图定义在哪些基本表上，以及定义了什么内容的数据。视图的虚拟表结构就是由<查询>中 Select 子句返回的结果决定的。<列名表>定义了视图的逻辑结构，与<查询>中返回的列相对应。With Check Option 选项表示强制视图上执行的所有数据修改语句都必须符合视图定义中的 Where 条件。

【例 4-39】 以学生表 Student(S#, Sname, Age, Dept)为例，其中 Dept 表示学生所在的系。下面给出了计算机系的学生视图定义。

第一种方式(带视图列名表)：

```
Create View cs_view (sno, name, age)
As
Select S#, Sname, Age From Student Where Dept = '计算机系'
```

上述语句定义的视图逻辑结构为：cs_view (sno, name, age)。

第二种方式(省略视图列名表)：

```
Create View cs_view
As
Select S#, Sname, Age From Student Where Dept = '计算机系'
With Check Option
```

这一种方式定义的视图逻辑结构为：cs_view(S#, Sname, Age)。当省略视图的列名表时，视图就自动获得 Select 查询返回的列名。

【例 4-40】 设学生选课表为 SC(S#, C#, Score)，把每门课程的课程号和平均成绩定义为一个视图。

第一种方式(带视图列名表)：

```
Create View c_view (cno, avg_score)
As
Select C#, AVG(Score)  From SC Group By C#
```

第二种方式(不带视图列名表)：

```
Create View c_view
As
Select C#, AVG(Score) As avg_score From SC Group By C#
```

当 Select 查询中使用了函数时，如果在 Create View 时省略视图列名表，则要求在 Select 语句中必须给函数指定别名。但是如果像第一种方式一样指定了视图列名表，则不要求必须指定函数别名。

4.6.4　视图的查询

视图的查询也是通过 Select 语句来完成的。视图查询的语法与基本表上的 Select 查询没有什么区别。

【例 4-41】 以例 4-40 所定义的视图 c_view 为例。查询平均成绩在 80 分以上的课程号与课程名。该查询若在视图 c_view 上执行，可以表示为下面的 SQL 语句：

```
Select A.C#, A.Cname From Course A, c_view B Where A.C# = B.C# and
B.avg_score> = 80
```

如果不使用视图 c_view，其相应的 SQL 语句为：

```
Select A.C#, A.Cname
From Course A, (Select C#, Avg(Score) as avg_score From SC Group By C#) SC2
Where A.C# = SC2.C# and SC2.avg_score > = 80
```

可以看到，不使用视图时的 SQL 语句要复杂很多。通过预先定义视图，可以大大简化程序员眼里的数据和 SQL 语句。这是视图的主要优点之一。

4.6.5　视图的更新

视图的更新也是通过 Update 语句来实现的。Update 语句的语法与前面讨论的相同，但因为视图是建立在基本表之上的，所以视图上的 Update 语句实际上最终都要转换为基本表上的 Update 操作，并且要求视图上的 Update 语句必须能够转换为基本表上的 Update 操作。也就是说，并非所有视图都是可以更新的。

下面类型的视图都是不可更新的：

(1) 基于连接查询的视图不可更新。

(2) 使用了函数、表达式、Distinct 的视图不可更新。

(3) 使用了分组聚集操作的视图不可更新。

上面这些视图不可更新的原因很简单，就是它们都无法转换成基本表上的 Update 操作。在 SQL 数据库中，可更新的视图必须满足下面的条件：

只有建立在单个表上，而且只去掉了基本表的某些行和列，但保留了主键的视图才是可更新的。

下面给出视图更新的一个示例。

【例 4-42】 以例 4-39 定义的计算机系学生视图 cs_view(sno, name, age)为例。“将计算机系学号为 001 的学生的姓名改为 Rose”在视图上的更新语句为：

```
Update cs_view Set name = 'Rose' Where sno = '001'
```

上面的视图更新语句最终将转换为 Student 表上的 Update 语句执行，因为视图对应的物理数据都是存储在其所关联的基本表上的。

4.6.6 Drop View 语句

Drop View 语句完成视图的删除，其语法与 Drop Table 类似：

```
Drop View <视图名>
```

例如，删除前面定义的计算机系学生视图对应的语句为：

```
Drop View cs_view
```

注意，由于视图只有逻辑结构而没有物理数据存储，所以视图的删除不会对数据库数据产生破坏，也不会对基本表上的数据完整性产生影响。

4.7 本 章 小 结

本章主要介绍了结构化查询语言(SQL)的相关概念，包括 SQL 的发展历史、基本组成以及 DDL、DML 和 DCL 语句，介绍了主要的 SQL 语句的书写语法和实例，并着重讨论了 Select 查询语句的书写方法。

通过对本章的学习，读者应掌握数据库语言和 SQL 的基本概念，了解 SQL 数据库的体系结构，理解 SQL 和关系代数之间的区别和联系，并能够熟练运用 SQL 回答用户的查询。

习　　题

1．SQL 与关系代数之间有何区别与联系？

2．简述 SQL 的基本组成。

3．SQL 数据库体系结构与 ANSI/SPARC 体系结构之间是如何对应的？

4．视图的作用都有哪些？

5．请用 SQL 回答第 3 章的第 10 题。

第 5 章　过程化 SQL

SQL 最初的设计目标是响应用户的数据库存取需求。由于 SQL 是一种声明性语言，因此 DBMS 在执行 SQL 语句时采取的是按语句执行的方式，而且 SQL 语句在 DBMS 服务器上也不能永久保存。事实上，SQL 语句通常嵌入在应用程序的代码中以完成对数据库的存取操作。这与 C 语言这类过程化语言的执行方式有很大的不同。SQL 语句还不能称为程序，因为它不能像 C 语言一样编译为持久存储的程序，并且它也不具备实现算法流程的语句(如循环语句、分支语句等)。但是，随着客户端/服务器(C/S)计算模式的出现，希望在数据库系统中也能够将客户端的计算任务(部分地)迁移到数据库服务器，从而可以发挥服务器的资源优势。为了实现这一目标，SQL 应当能够像程序一样持久存储在数据库服务器中，以便随时被服务器调用，并且 SQL 还应当能够完成一些复杂的计算任务，具备过程化程序设计语言的基本功能，这导致了过程化 SQL 的提出。

内容提要：本章首先介绍过程化 SQL 的概况以及与 SQL 之间的联系与区别，然后着重介绍 Microsoft SQL Server 中的过程化 SQL——T-SQL，最后讨论过程化 SQL 在 DBMS 中的两类主要应用：存储过程和触发器。

5.1　过程化 SQL 概述

过程化 SQL 可以看成 SQL 的过程化扩展，即在声明性 SQL 中添加过程化程序设计语言的要素，使其具备一定的程序设计能力。

5.1.1　过程化 SQL 与 SQL

SQL 是过程化 SQL 的基础。作为 SQL 的扩展，过程化 SQL 支持 SQL 的基本要素，如数据类型、关键字等。但是，过程化 SQL 在 SQL 的基础上，增加了对过程化程序设计的支持，因此与标准 SQL 之间存在着较大的区别。虽然 SQL 已经是国际标准，但是过程化 SQL 并没有标准。这是因为各个 DBMS 厂商在扩展 SQL 以建立过程化 SQL 时采取了不同的实现方式。目前，大型的商用 DBMS 一般都支持过程化 SQL，例如，Microsoft SQL Server 支持的过程化 SQL 称为 Transact-SQL 或简称 T-SQL，Oracle 支持的过程化 SQL 称为 PL/SQL，Informix 支持的过程化 SQL 为 E-SQL，等等。

过程化 SQL 对标准 SQL 所进行的扩展主要表现在对过程化程序设计的支持上。过程化程序设计一般支持三种程序结构：顺序结构、分支结构和循环结构。顺序结构允许定义一个程序块，其中的语句按顺序执行；分支结构允许在程序中进行条件判断并根据判断结果选择程序分支；循环结构允许循环执行某些语句。因此，过程化 SQL 最主要的扩展就是增加了对上述程序结构的支持。此外，它还增加了一般程序设计语言都具有的一些功能，如变量定义、赋值、输入/输出等。

SQL 与过程化 SQL 之间的差别可归纳为以下几点。

(1) SQL 语句只能一条一条地执行，而过程化 SQL 可以将若干语句组织成过程或函数后再执行。

(2) 过程化 SQL 在 SQL 之上扩充了若干语句，使其能够支持过程化程序设计。这些语句包括赋值语句、分支语句、循环语句等。

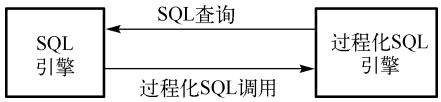


图 5-1　SQL 与过程化 SQL 之间的关系

图 5-1 显示了 SQL 与过程化 SQL 之间的关系。SQL 语句和过程化 SQL 程序都可以在 DBMS 上运行，可以相互调用。

过程化 SQL 同时具备了 SQL 和过程化程序设计语言的特性。从 SQL 的角度看，过程化 SQL 支持数据定义语言(DDL)、数据操纵语言(DML)、数据控制语言(DCL)等标准的 SQL 语句，也支持 SQL 标准中的数据类型、标识符等定义；从过程化程序设计语言的角度看，过程化 SQL 支持变量定义、赋值、语句块、分支语句、循环语句、输入/输出等要素，支持编写可独立存储并在 DBMS 中运行的程序。

过程化 SQL 最主要的用途是实现数据库应用系统的 C/S 结构。C/S 结构要求客户端和服务器协作完成计算任务，即任务可以分散在客户端和服务器一起执行。但在标准 SQL 中，当客户端需要执行数据库操作时，只能通过 SQL 语句与 DBMS 交互，通常一个任务需要多次网络通信，而且客户端需要多次计算，同时服务器只能执行单条 SQL 语句，无法充分利用服务器的高性能计算能力。

以银行转账应用为例，假设要从 *A* 账户转到 *B* 账户 100 元，则客户端大致要执行下面几个步骤的操作(图 5-2)。

(1) 通过 SQL 查询账户 *A* 是否存在(Select 语句)。

(2) 通过 SQL 查询账户 *B* 是否存在(Select 语句)。

(3) 查询账户 *A* 上的余额(Select 语句)。

(4) 修改 *A* 上的余额(Update 语句)。

(5) 修改 *B* 上的余额(Update 语句)。

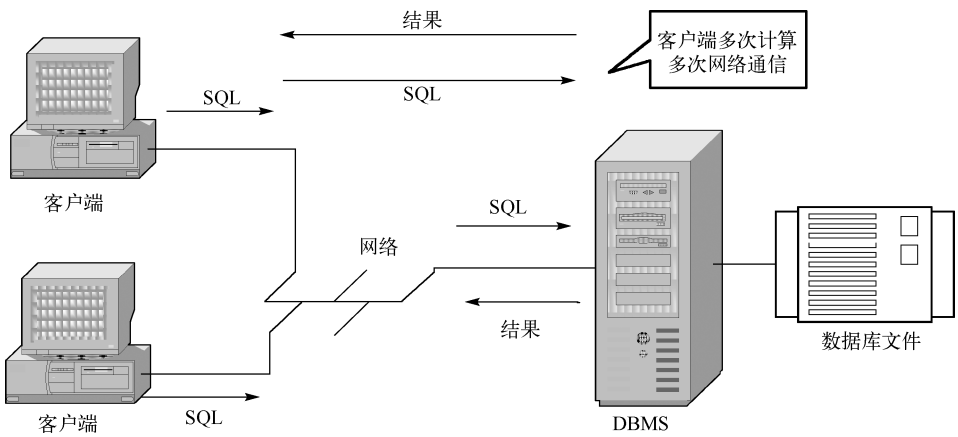


图 5-2　标准 SQL 下的数据库应用执行过程

在这个过程中，客户端需要完成 5 次 SQL 语句的调用，并且需要 5 次网络通信。这种

方式带来了三个主要的问题：一是计算任务主要都加在了客户端上，加重了客户端的编程负担；二是不能充分发挥服务器的高性能计算能力，因为服务器通常配置较高，性能也更好，因此如果能够将计算任务放到服务器上，则可以改善响应性能；三是网络传输过多将导致系统响应性能下降，因为与 CPU、内存、总线等相比，网络传输要慢得多。

如果采用过程化 SQL 来实现银行转账，可以将银行转账用过程化 SQL 预先编写好程序并存储在 DBMS 中，然后客户端只需一次调用就可以执行过程化 SQL 程序。特别需要注意的是，预先编写好的过程化 SQL 程序是在服务器上运行的。这样一来，一方面计算任务可以通过过程化 SQL 转移到服务器上执行，从而使得用户可以采取更适合的方式来平衡客户端和服务器上的负载，另一方面减少了网络通信，有助于响应性能的提升。图 5-3 给出了基于过程化 SQL 程序的数据库应用执行过程。

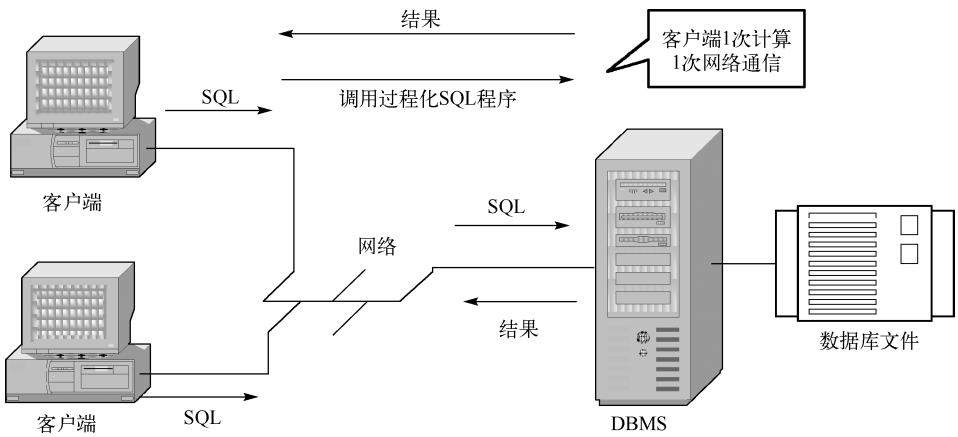


图 5-3　基于过程化 SQL 程序的数据库应用执行过程

表 5-1 总结了 SQL 和过程化 SQL 在数据库应用中的主要区别。

表 5-1　数据库应用中使用 SQL 和过程化 SQL 时的主要区别

使用 SQL	使用过程化 SQL
客户端计算任务多	客户端计算任务少
服务器计算任务少	服务器计算任务多
网络通信多	网络通信少
计算灵活性差	计算灵活性好，可以调整客户端和服务器的计算负载，也可以完成一些 SQL 不能完成的复杂计算

5.1.2　过程化 SQL 的特点

过程化 SQL 是一种非常特殊的程序设计语言。首先，它不同于 SQL，因为它具备过程化程序设计的能力；其次，它也不同于一般的过程化程序设计语言，因为它并非用来编写面向用户的应用程序。过程化 SQL 程序所面向的用户是 DBMS，它最终是由 DBMS 管理和使用的。因此，过程化 SQL 与一般的过程化程序设计语言相比在某些功能上较弱，如输入/输出等。

过程化 SQL 的主要特点如下。

(1)与 SQL 相比，过程化 SQL 支持分支语句、循环语句等过程化程序设计语言的要素，能够编写可以在数据库服务器存储和执行的 SQL 程序。

(2)与 C 语言等传统的过程化程序设计语言相比，过程化 SQL 支持标准 SQL 的数据类型、函数和语句，可以直接与 DBMS 交互实现数据库的存取。

(3)过程化 SQL 可以将数据库应用系统中的数据访问任务从客户端迁移到服务器，在数据库系统中实现灵活的 C/S 结构。

(4)过程化 SQL 使数据库服务器可以通过自定义过程化 SQL 程序的方式实现某些复杂的完整性控制和维护功能，从而提高数据库系统的管理能力。

5.2　过程化 SQL 的程序结构

过程化 SQL 程序由多个支持 SQL 的语句构成。与 C 程序之类传统的过程化程序的不同之处在于，过程化 SQL 程序是需要在 DBMS 支持下才能运行的，因此无法像 C 程序那样通过编译为可执行的 EXE 文件来运行。

在数据库应用程序中，有两种使用过程化 SQL 的方式：会话方式和过程方式。其中会话方式主要用于过程化 SQL 程序的调试和测试，过程方式是实际数据库应用开发中常用的方式。下面分别加以介绍。

5.2.1　会话方式

会话方式是指在一次会话中使用过程化 SQL 语句编程。通常，可以在 DBMS 支持的 SQL 语句调试客户端中输入过程化 SQL 语句(当然，客户端也同样支持标准 SQL 语句)，并且在客户端中查看执行结果。

图 5-4 显示了在 MySQL 客户端(此处为 MySQL Workbench，也可以使用其他 SQL 客户端工具)中使用会话方式输入和调试过程化 SQL 语句的界面。

图 5-4　在 MySQL 客户端中使用会话方式输入和调试过程化 SQL 语句的界面

会话方式主要用于测试过程化 SQL 程序。例如，在图 5-4 的示例中，第 3 行 Set 语句返回了函数 concat 的结果。当使用过程化 SQL 编写了一个存储过程或者函数时，可以在 SQL 客户端中使用会话方式测试过程和函数的运行结果。

MySQL、Oracle、Microsoft SQL Server 对于用会话方式调试过程化 SQL 语句的支持略有不同。表 5-2 总结了三者的主要差异。

表 5-2　三类 DBMS 在用会话方式调试过程化 SQL 语句上的差异

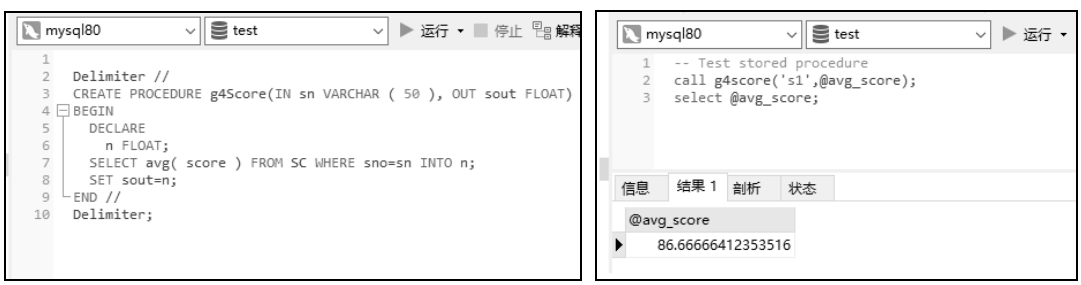
语句类别	MySQL	Oracle	Microsoft SQL Server
赋值语句、输出语句、表达式运算、函数调用	支持	支持	支持
流程控制语句(分支、循环)	不支持	支持	支持
语句块定义	不支持	支持	支持
局部变量定义	不支持	支持	支持

5.2.2　过程方式

过程方式是指在存储过程、函数或者触发器中使用过程化 SQL 编程。这是实际数据库应用开发中常用的方式，也是利用过程化 SQL 实现 C/S 计算模式的主要手段。

在过程方式下，用户使用过程化 SQL 编写存储过程、函数或者触发器，在前端开发语言(如 VB、C#、Java 等)调用存储过程或者函数，或者通过定制的数据库操作启动触发器的执行(触发器不能被前端程序显示调用)。过程方式支持全部的过程化程序设计语言要素，同时还支持事务编程。Oracle、MySQL、Microsoft SQL Server 均支持过程方式，但是在存储过程、函数和触发器的定义与使用上略有差别。

图 5-5 给出了一个在 MySQL 客户端中定义和测试存储过程的示例。图 5-5(a)创建了一个存储过程用于返回给定学生的平均成绩，图 5-5(b)显示了使用会话方式测试存储过程的运行结果。



(a)定义存储过程　　(b)使用会话方式测试存储过程

图 5-5　在 MySQL 客户端中定义和测试存储过程

5.3　过程化 SQL 的语句扩展

过程化程序主要包含顺序结构、分支结构、循环结构三种程序结构，另外还包含输入/输出、变量定义、语句块等功能。因此，过程化 SQL 也主要从这些方面对 SQL 进行扩展。

5.3.1　变量定义

变量定义是过程化语言中的重要功能，所以过程化 SQL 在变量定义方面提供了较多的扩展。总体上，变量可以区分为局部变量和全局变量，不同 DBMS 分别采用了不同的实现方法。

变量定义一般通过 Declare 语句实现，下面分别针对 MySQL、Oracle 和 Microsoft SQL Server 介绍它们对于变量定义的不同支持。

1. MySQL

MySQL 支持三种类型的变量，包括局部变量、用户变量、系统变量。局部变量必须使用 Declare 语句定义。局部变量定义格式为：

```
Declare<变量名><类型>
```

例如：

```
Declare nameVarchar(10);
Declare snoIntDefault 0;
```

上面的第二个变量定义示例中，定义了一个 Int 类型的变量 sno，并且设置其默认值为 0。

变量名使用常规定义，可以包括字母、数字、下画线等，作用域为 Begin…End 之间的程序块。

用户变量在 MySQL 中不需要预先定义，但是变量名前须加一个@符号以进行区分。用户变量的作用域为当前连接，在连接中声明的用户变量会一直有效，直到用户断开与数据库实例的连接。在当前连接中声明的用户变量无法在另一连接中使用。

图 5-5(a)中，变量 *n* 为局部变量。图 5-5(b)中使用了一个用户变量@avg_score。可以看到，用户变量@avg_score 不需要预先使用 Declare 进行定义。由于用户变量不需要预先定义类型，所以用户变量不是类型绑定的。

系统变量是 MySQL 内部定义的变量，变量名前有@@符号。系统变量又分为会话变量和全局变量，这两类变量的区别在于会话变量的作用域为当前连接，而全局变量的作用域为所有客户端连接。会话变量与前面的用户变量的区别在于，会话变量是系统定义的，而用户变量是用户自定义的。全局变量在 MySQL 实例启动的时候初始化为默认值。会话变量在每次用户建立一个新的连接时由 MySQL 初始化，此时 MySQL 会将当前所有全局变量的值复制一份来作为当前连接的会话变量的值。也就是说，如果用户没有修改过会话变量，则当前连接的会话变量集合以及取值与全局变量是完全一样的。全局变量与会话变量的区别在于，对全局变量的修改会影响到整个服务器，但是对会话变量的修改只会影响到当前的会话。另外，系统变量一般只在过程化 SQL 中读取，用于在程序中判断系统当前的某个特定状态值，如@@version。Show global variables 和 Show session variables 语句可以查看 MySQL 中所有的系统变量和当前连接的会话变量。

2. Oracle

Oracle 中的变量分为局部变量和全局变量，两者均通过 Declare 语句定义。Oracle 的 PL/SQL 允许程序块的嵌套，即每个程序块可以包含另一个内部程序块。如果在内部程序块中声明了一个变量，则该变量为局部变量，它只能被内部程序块访问，外部程序块无法访问该内部变量。如果在外部程序块中定义了一个变量，则该变量为全局变量，它可以被外部程序块访问，也可以被所有嵌套的内部程序块访问。下面的示例给出了 Oracle PL/SQL 中局部变量和全局变量的定义区别。

```
DECLARE
   --全局变量
   n1 number := 100;
   n2 number := 200;
BEGIN
   DBMS_OUTPUT.Put_Line('全局变量 n1 的值为:' || n1);
   DBMS_OUTPUT.Put_Line('全局变量 n2 的值为:' || n2);
   DECLARE
      -- 局部变量
      num1 number := 10;
      num2 number := 20;
   BEGIN
      DBMS_OUTPUT.Put_Line('局部变量 num1 的值为:' || num1);
      DBMS_OUTPUT.Put_Line('局部变量 num2 的值为:' || num2);
   END;
END;
```

3. Microsoft SQL Server

Microsoft SQL Server 中的变量同样分为局部变量和全局变量。局部变量使用 Declare 语句定义，而全局变量是系统定义的，用户一般只能读取全局变量值。

局部变量定义的形式为 Declare @local_variable data_type，其中@local_variable 是变量的名称，data_type 是任何由系统提供的或用户定义的数据类型，但不能是 text、ntext 或 image 数据类型。局部变量定义的一个示例：Declare @VAL Char(2)，该语句定义了一个 Char 类型的局部变量@VAL。注意，符号@是局部变量定义的一部分，不能缺少。

Microsoft SQL Server 中的全局变量是系统预定义的，用于返回一些系统信息。全局变量以@@开头，例如，@@ERROR 保存了最后执行的 SQL 语句的错误代码，@@ROWCOUNT 存储了受上一条语句影响的行数。在过程化 SQL 中，用户可以读取特定的全局变量值来判断系统的当前状态。

5.3.2　变量赋值

MySQL、Oracle、Microsoft SQL Server 在变量赋值上有相同的语句，然而各自又都具有自己独特的赋值方式。下面逐一介绍。

1. MySQL

MySQL 的变量赋值有两种方式，一种是使用 Set 语句，另一种是使用 Select…Into…语句。下面是一些变量赋值的示例。

```
Declare status int;
Set status = 1;                          --局部变量，需预先定义
Set @done = 1;                           --会话变量，不需要定义
Select max(score) From SC Into v1;       --局部变量
Select sname From Student Where sno = 's1' Into @name;  --会话变量
```

```
Select v1 Into @name;
Select max(score), min(score) Into n2, n3 FROM SC;
```

注意，如果使用 Select 为多个变量赋值，不能写成下面的错误形式：

```
Select max(score) Into n2, min(score) Into n3 FROM SC;
```

2. Oracle

Oracle 的变量赋值有两种方式，一种是使用:=，另一种是使用 Select…Into…语句。其中，Select…Into…语句的用法与 MySQL 类似。下面是一些变量赋值的示例。

```
Declare s name Varchar2(50);
Declare status Int;
Begin
   status := 1;
   Select name Into s name From Student Where sno = '100';
End;
```

3. Microsoft SQL Server

Microsoft SQL Server 的变量赋值语句有两种：Set 和 Select。两者的功能基本相同，不同之处在于 Select 赋值允许将 Select 查询语句返回的值赋给变量，而 Set 语句则不行。下面是一些变量赋值的示例。

```
Declare @status int;
Declare @score float;
Set @status = 1;                              --使用 Set 语句赋值
Select @score = max(score) From SC;           --使用 Select 语句赋值
```

5.3.3　分支语句

分支语句在过程化 SQL 中一般通过 If…Else…语句来实现。其中，MySQL 和 Oracle 的分支语句格式相同，包括 If 语句和 Case 语句。If 语句的格式如下：

```
If <表达式>Then
    <语句>
ElseIf  <表达式> Then
    <语句>
…
Else
    <语句>
End If;
```

Case 语句一般用于多分支结构，其格式如下：

```
Case <表达式>
  WHEN 'value1' THEN S1;
  WHEN 'value2' THEN S2;
```

```
    WHEN 'value3' THEN S3;
    …
    ELSE Sn;              --default case
End Case;
```

图 5-6 显示了 MySQL 中的 Case 语句示例。

Microsoft SQL Server 的分支语句格式与 MySQL、Oracle 有所不同，主要是少了 Then、Else If 以及 End If，并且也不支持 Case 多分支语句。Microsoft SQL Server 的 If 语句格式如下：

```
If <表达式>
    <语句>
Else
    <语句>
```

```
1  DELIMITER |
2  CREATE PROCEDURE p(IN e INT)
3  BEGIN
4    DECLARE v INT DEFAULT 1;
5    SELECT level INTO V FROM EMP WHERE eno=e;
6    CASE v
7      WHEN 2 THEN SET v=1;
8      WHEN 3 THEN SET v=2;
9      WHEN 1 THEN SET v=3;
10     ELSE
11         SET v=(v-2) mod 3;
12   END CASE;
13 END |
14 DELIMTER ;
```

图 5-6　MySQL 中的 Case 语句示例

下面是一个示例：

```
Declare @var Float;
Select @var = (Select sum(salary) From Employee);
If @var>10000
    If @var<20000
        Print '工资总和为 10000～20000';
    Else
        Print '工资总和大于 20000';
Else
    Print '工资总和小于 10000';
```

5.3.4　循环语句

循环结构是过程化程序中的重要结构。MySQL、Oracle、Microsoft SQL Server 三者对于循环语句的支持也不尽相同，其中 MySQL 和 Oracle 都支持三种循环语句，而 Microsoft SQL Server 只支持一种循环语句。

1. MySQL

MySQL 支持 While、Repeat 和 Loop 三种循环语句。While 循环语句的格式如下(注意 Do 和最后的分号)：

```
While <循环控制条件> Do
    <语句>
      …
End While;
```

图 5-7 给出了 MySQL 中使用 While 循环求 1 到给定数字之间偶数之和的程序示例。

Repeat 循环语句格式如下：

```
Repeat
```

```
-- 计算1到i的偶数和
DELIMITER //
CREATE PROCEDURE even_sum(IN i INT, OUT sum INT)
BEGIN
  DECLARE j INT DEFAULT 1;
  SET sum=0;
  WHILE j<=i DO
    IF j%2=0 THEN
      SET sum=sum+j;
    END IF;
    SET j=j+1;
  END WHILE;
END //
DELIMITER ;
```

```
call even_sum(100,@sum);
select @sum;
```

@sum
2550

图 5-7　MySQL 中的 While 循环语句示例

```
        <语句>
    Until <循环控制条件>

End Repeat;
```

Repeat 语句至少会执行一次循环体，直到满足 Until 语句中定义的循环退出条件为止。图 5-8 给出了使用 Repeat 循环计算 1～*i* 的偶数之和的程序示例。

```
-- 计算1到i的偶数和
DELIMITER //
CREATE PROCEDURE even_sum2(IN i INT, OUT sum INT)
BEGIN
  DECLARE j INT DEFAULT 1;
  SET sum=0;
  REPEAT
    IF j%2=0 THEN
      SET sum=sum+j;
    END IF;
    SET j=j+1;
    UNTIL j>i
  END REPEAT;
END //
DELIMITER ;
```

```
call even_sum2(100,@sum);
select @sum;
```

@sum
2550

图 5-8　MySQL 中的 Repeat 循环语句示例

Loop 循环是无内部控制结构的循环结构，它循环执行循环体中的<语句>，但没有循环控制条件，因此必须在循环体中显式地使用 Leave 语句退出循环。Loop 语句的格式如下：

```
<label>: Loop
<语句>
   If <循环控制条件> Then
       Leave <label>;
   End If;
End Loop <label>;
```

类似地，在图 5-9 中给出了使用 Loop 循环求解 1～*i* 的偶数之和的程序示例。可以发现，三种循环语句都可以用来实现程序的循环执行。在实际开发中，用户可以根据实际情况自行选择合适的循环语句。

2. Oracle

Oracle 支持 While、For 和 Loop 循环。虽然 Oracle 和 MySQL 都支持 While 和 Loop 循环，但语法结构存在差异。

```
-- 计算1到i的偶数和
DELIMITER //
CREATE PROCEDURE even_sum3(IN i INT, OUT sum INT)
BEGIN
  DECLARE j INT DEFAULT 1;
  SET sum=0;
  iter: LOOP
    IF j%2=0 THEN
      SET sum=sum+j;
    END IF;
    SET j=j+1;
    IF j>i THEN
      LEAVE iter;
    END IF;
  END LOOP iter;
END //
DELIMITER ;
```

```
call even_sum3(100,@sum);
select @sum;
```

@sum
2550

图 5-9　MySQL 中的 Loop 循环语句示例

Oracle 的 While 循环语句格式如下：

```
While <循环控制条件> Loop
    <语句>
End Loop;
```

例如，下面的过程化 SQL 实现了 1～100 的求和：

```
Declare
    x Number;
    total Number;
Begin
    x := 1;
    total := 0;
    While x< = 100 Loop
        total := total+x;
        x := x+1;
    End Loop
End;
```

For 循环是 Oracle 特有的循环语句，其格式如下：

```
For <计数变量> In [Reverse] <开始值>…<结束值> Loop
    <语句>
End Loop;
```

在 For 循环中，循环体每执行一次，计数变量自动加 1。如果有 Reverse 关键字，则每次循环计数变量自动减 1。下面的代码用 For 循环实现了 1～100 的求和：

```
Declare
    x Number;
    total Number;
Begin
    total := 0;
    For x In 1..100 Loop
        total := total+x;
    End Loop
```

```
End;
```

Oracle 的 Loop 循环与 MySQL 类似，也是无内部循环控制语句的循环结构，因此也需要用户在循环体中显示结束循环。与 MySQL 的 Leave 结束循环不同，Oracle 中需要用 Exit 语句退出循环。下面给出了 Loop 循环的格式：

```
Loop
    <语句>
End Loop
```

在 Oracle 中，退出 Loop 循环有两种方式：Exit 和 Exit When。下面的代码通过求 1～100 的和，演示了这两种方式的区别。

使用 Exit 退出 Loop 循环	使用 Exit When 退出 Loop 循环

Declare
x Number: = 1;
total Number: = 0;
Begin
Loop
If x< = 100 Then
total: = total+x;
x: = x+1;
Else
Exit;
End If
End Loop
End;
	Declare
	x Number: = 1;
	total Number: = 0;
	Begin
	Loop
	total: = total+x;
	x: = x+1;
	Exit When x>100
	End Loop
	End;

3. Microsoft SQL Server

Microsoft SQL Server 对于循环语句的支持比 MySQL 和 Oracle 要简单，它仅支持 While 循环，其格式如下：

```
While <循环控制条件>
   {SQL 语句或语句块}
   [Break]
   {SQL 语句或语句块}
   [Continue]
   {SQL 语句或语句块}
```

其中，Break 语句是强制终止循环，Continue 语句是跳过循环体内位于 Continue 之后的语句，执行下一次循环控制条件测试。可以看到，Break 和 Continue 语句的含义与 C 语言中的 Break、Continue 相同。

下面的代码示例给出了使用 While 循环求解 1～100 的奇数之和。

```
Declare @i Int, @sum Int;
Set @i = 0, @sum = 0;
While @i> = 0
    Begin
```

```
        Set @i = @i+1;
        If @i< = 100
            If(@i%2) = 0                          --略过偶数
                Continue;
            Else
                Set @sum = @sum+@i;
        Else
            Begin
            Print  '1～100 的奇数和 = ' +str(@sum);
            Break;
        End
    End
```

5.3.5　输入/输出语句

输入/输出语句在传统过程化程序设计语言(如 C 语言)中是重要的程序设计语言要素，这是因为传统的 C 程序需要直接与用户交互，因此必须在输入/输出语句方面提供丰富、灵活的支持。但是，过程化 SQL 程序是驻留在 DBMS 端，被前端开发语言显式调用或者隐式触发的，也就是说，过程化 SQL 程序一般不需要跟用户直接交互。因此，目前的过程化 SQL 一般不提供输入语句方面的支持，仅在输出语句方面提供一定的扩展，供用户在 DBMS 端调试和测试过程化 SQL 使用。

Oracle、MySQL、Microsoft SQL Server 对于输出语句的支持有所不同。在 MySQL 中，过程化 SQL 程序的输出语句依然采用了 Select 语句。Microsoft SQL Server 则提供了 Print 语句用于输出变量值。Oracle 则通过 DBMS_OUTPUT 包(Package)来提供输出支持，例如，DBMS_OUTPUT.PUT_LINE 语句可以输出一行文本。Oracle 的包是一个存储过程和函数的集合，DBMS_OUTPUT 包是 Oracle 系统自带的提供过程化 SQL 程序中输出功能的包。Oracle 也允许用户将自定义的过程化 SQL 程序生成包。

5.4　异 常 处 理

过程化程序设计语言一般都需要有程序异常处理机制。在程序中加入异常处理代码可以捕捉运行中可能出现的错误或者意外情况并加以处理，从而避免程序运行时异常退出，提高程序的健壮性。

C 语言等过程化程序设计语言通常都采用错误陷阱方式处理异常(实际上 Java、VB 等面向对象程序设计语言也同样采用错误陷阱机制)。错误陷阱是指在程序中声明或者设置一个错误捕捉器，一旦程序运行时触发指定的错误就可以捕捉该错误，然后跳转执行相应的错误处理代码(例如，取消操作，返回错误提示信息等)。

过程化 SQL 通常采用错误陷阱的方式处理异常，但是不同的过程化 SQL 对于错误陷阱的实现方式有所区别。

5.4.1　MySQL 的异常处理

MySQL 在过程化 SQL 中的异常处理通过下面的语句实现：

```
Declare <处理方式> Handler For <异常类型><SQL>
```

该语句定义了一个错误陷阱，其含义为：当出现<异常类型>对应的错误时，根据<处理方式>中定义的操作方式执行相应的操作。如果<操作方式>为 Continue（执行下一条 SQL 语句），则执行<SQL>中给定的 SQL 语句。

1. <处理方式>

<处理方式>指明了发生异常时 MySQL 采用的动作，支持三种形式，具体如下。

（1）Continue：继续执行下一条语句。

（2）Exit：直接退出（该操作很少使用）。

（3）Undo：回退（目前 MySQL 暂不支持）。

MySQL 一般采用 Continue 方式处理异常，即发生异常时继续执行<SQL>中的语句。此时，用户需要在可能出现异常的程序代码后面添加相应的语句，判断是否出现了异常并根据判断结果执行相应的动作。

2. <异常类型>

<异常类型>指明了错误陷阱要捕捉的错误类型。MySQL 支持四种异常类型。

1）SQLSTATE 值

SQLSTATE 值是一个长度为 5 位的字符串，用于指示错误类型。当过程化 SQL 程序正常执行时，MySQL 将返回以 00 开头的 SQLSATE 值。具体的 SQLSTATE 值可以参考相应的 MySQL 手册。例如，下面的示例定义了一个 SQLSTATE 值为 42S02 的错误陷阱：

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02' SET @info = 'NO_SUCH_TABLE';
```

2）Error Code（错误码）

错误码是 4 位的数字，它的功能与 5 位字符的 SQLSTATE 值类似，都用于指示特定的错误类型，例如，错误码 1146 表示“表不存在”。具体编程时可以参考 MySQL 手册找到想处理的错误对应的错误码。例如，下面的示例定义了错误码为 1146 的错误陷阱：

```
DECLARE CONTINUE HANDLER FOR 1146 SET @info = 'NO_SUCH_TABLE';
```

3）预定义错误

在实际过程化 SQL 编程中，对于一些常见的错误，MySQL 使用预定义的错误关键字来简化编程。这些预定义的错误关键字包括 SQLWARNING（对应以 01 开头的 SQLSTATE 值，表示 SQL 执行时发生了警告）、NOT FOUND（对应以 02 开头的 SQLSTATE 值，表示游标或 Select 语句没有返回值）或 SQLEXCEPTION（其他的 SQLSTATE 值，表示发生了未知错误）。下面的语句给出了 SQLWARNING、NOT FOUND 和 SQLEXCEPTION 的错误陷阱示例：

```
DECLARE CONTINUE HANDLER FOR SQLWARNING SET @status = 1;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET @status = 1;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET @status = 1;
```

4) 自定义错误

<异常类型>中还允许使用用户自定义错误。这类错误与 MySQL 错误码或 SQLSTATE 值相关联，实际上相当于给特定的错误码或者 SQLSTATE 值定义了一个别名。错误条件需要用 DECLARE CONDITION 语句预先定义。例如，下面的语句先自定义了一个错误，然后声明了一个错误陷阱：

```
DECLARE no_such_table CONDITION FOR 1146;
DECLARE CONTINUE HANDLER FOR no_such_table SET @status = 1;
```

3. <SQL>

<SQL>是当处理方式为 CONTINUE 时 MySQL 在发生异常的情况下执行的 SQL 语句。它可以是单条 SQL 语句，也可以是复合语句。一般情况下，可以在<SQL>中改变若干状态变量的值，从而在后面的代码中可以通过状态变量值来区分究竟发生了什么错误，然后采取相应的动作。

图 5-10 给出了一个捕捉并处理 NOT FOUND 异常的示例。该示例中，使用一个存储过程 error2 根据学号查询学生的年龄；如果学生不存在，则会触发 NOT FOUND 异常。当执行该存储过程 error2 时，如果学生存在，则状态变量 state 返回 0(图 5-10 左下角的图)，若学号不存在，此时会触发 NOT FOUND 异常，因为处理方式是 CONTINUE，所以 MySQL 会执行 DECLARE…HANDLER 语句中定义的<SQL>，即 SET *s* = 1 语句，从而使存储过程返回 state = 1。图 5-10 右下角的测试结果显示了这一错误处理的结果。

图 5-10　MySQL 中捕捉并处理 NOT FOUND 异常的程序示例

图 5-11 给出了 MySQL 中一般的异常处理框架。首先，异常处理语句通常都在存储过程中使用。事实上，存储过程也是过程化 SQL 的主要载体。其次，由于存储过程完成了特定的业务处理，所以可以根据业务处理的特点预先估计可能发生的错误，然后在存储过程中使用 DECLARE…HANDLER 为预期的错误类型都声明相应的错误陷阱(如图 5-11 中的第 6～9 行)。如果业务过程中有一些特殊的错误类型，如“银行账户的余额小于 10 元”，

可以在程序中通过设置相应的状态变量值来表示这类自定义的错误(如图 5-11 中第 16～18 行)。最后，在程序的尾部(图 5-11 中第 21～34 行)对捕捉的所有错误进行统一处理：如果没有触发异常，则正常提交(COMMIT)；如果触发了异常，则根据错误捕捉语句中设定的状态变量值分别进行相应的处理，例如，设置针对性的错误消息文本，同时取消前面的所有业务操作(ROLLBACK)。第 11 行的 START TRANSACTION、第 23 行的 COMMIT 以及第 33 行的 ROLLBACK 语句都是专门的事务处理语句，将在 5.5 节介绍。

```
1   --  一般的错误处理框架，state用于返回错误码
2   DELIMITER //
3   CREATE PROCEDURE error_handler ( IN sn VARCHAR(50) , OUT state INT)
4   BEGIN
5     DECLARE s INT DEFAULT 0;
6     DECLARE CONTINUE HANDLER FOR 1146 SET s = 1; -- 特定错误的捕捉
7     DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02' SET s=2; -- 特定错误的捕捉
8     DECLARE CONTINUE HANDLER FOR NOT FOUND SET s = 3;  -- 如果有查询语句，空集错误的捕捉
9     DECLARE CONTINUE HANDLER FOR SQLEXEPTION SET s = 4;  -- 其余未知错误的捕捉
10    -- 如果有数据更新，则开始事务
11    START TRANSACTION;
12    -- 执行DML语句
13    SELECT age FROM student WHERE sno=sn INTO c;
14    INSERT INTO ...;
15    UPDATE student ...;
16    IF <自定义异常> THEN -- 可以自定义异常，比如余额不足1000
17      SET s=5;
18    END IF;
19
20    -- 下面开始集中处理错误
21    IF s=0 THEN
22      SET state=0;
23      COMMIT;
24    ELSE
25        CASE s -- 根据s值进行错误处理，例如设置state值
26          WHEN 1 THEN
27          WHEN 2 THEN
28          WHEN 3 THEN
29          WHEN 4 THEN
30          ELSE
31
32        END CASE;
33        ROLLBACK; -- 取消所有操作
34    END IF;
35  END //
36  DELIMITER ;
```

图 5-11　MySQL 中一般的异常处理框架

5.4.2　Oracle PL/SQL 的异常处理

Oracle 过程化 SQL，即 Oracle PL/SQL，对于异常处理的总体思路与 MySQL 类似，都是先通过错误陷阱捕捉错误，然后进行处理。但是，在实现机制上，Oracle PL/SQL 与 MySQL 区别较大。

1．Exception

PL/SQL 的异常处理通过 Exception 来实现。Oracle 中的 Exception 语句定义了程序中的一个错误陷阱。一旦程序中给出了 Exception，当 PL/SQL 程序执行出现错误时，Oracle 就会自动跳转到 Exception 开始的程序段并执行，所以 Exception 开始的程序段功能就是错误处理。由于 Oracle 跳转到 Exception 段后会继续顺序执行，所以一般情况下需要把 Exception 段放在程序的尾部，这样一来当 Exception 段执行完毕时整个程序就结束了。

PL/SQL 中的 Exception 段的一般格式如下。可以看到，Exception 后面的语句使用了 When 语句来定义一个个错误，并且每个错误的 Then 后面的<错误处理语句>给出了实际的错误处理操作。

```
Exception
    When <错误名 1> or <错误名 2>…Then
        <错误处理语句>
    When <错误名 1> or<错误名 2>…Then
        <错误处理语句>
    When Others Then
        <错误处理语句>
```

下面的示例演示了从 Student 表里查询学号为 001 的学生姓名时可能发生的错误处理情况。这里，在 Exception 段里定义了一个 NO_DATA_FOUND 异常，用于捕捉当学生不存在时的异常；同时使用 When Others 捕捉其他不可预期的错误。此例中的错误处理语句只是简单地输出了一行错误消息，但在实际开发中可以定义更复杂的操作，例如，返回一个特定的错误代码。

```
Declare
    Name Varchar2(20);
Begin
    Select sname Into name From Student Where s# = '001';
    DBMS_OUTPUT.PUT_LINE('学号 001 的学生姓名是：'|| sname)
Exception
    When NO_DATA_FOUND Then
        DBMS_OUTPUT.PUT_LINE('学号为 001 的学生不存在');
    When Others Then
        DBMS_OUTPUT.PUT_LINE('发生了其他错误');
End;
```

在 Exception 段中，PL/SQL 使用 When 语句捕捉特定的异常。Oracle 中预定义了 20 类标准异常。下面是编程时常用的一些标准异常，其余的可以参考详细的 PL/SQL 编程手册。

(1) NO_DATA_FOUND：当执行 Select 语句却没有返回任何记录时，触发该异常。

(2) TOO_MANY_ROWS：当执行 Select…Into…语句却返回了多行记录时，触发该异常。这是因为 Select…Into…作为变量赋值语句，每次只能为变量赋一个特定的值，所以如果 Select 查询返回了多个值，将无法正确执行赋值操作。在 PL/SQL 编程时，若使用 Select…Into…赋值，需要特别注意 Select 查询是否必定返回单一值，否则就会触发 TOO_MANY_ROWS 异常。

(3) VALUE_ERROR：出现变量赋值错误时触发该异常，原因可能是类型不匹配或者值超出了类型范围等。

(4) ZERO_DIVIDE：出现被零除时触发该异常。

(5) TIMEOUT_ON_RESOURCE：表示资源等待超时，原因可能是网络通信故障、死锁等导致 PL/SQL 语句执行超时。

2. 人工生成异常

除了 Oracle 自定义的标准异常外，很多时候需要自行定义异常。因此，Oracle PL/SQL 提供了人工生成异常的方式。

PL/SQL 支持两种人工生成异常的方式：一种是使用 Raise_Application_Error 语句直接生成并触发异常；另一种是先声明一个自定义异常，然后触发异常。

1) 直接生成并触发异常

Raise_Application_Error 语句可以在 PL/SQL 中直接生成并触发异常，其格式如下：

```
Raise_Application_Error (<自定义错误号>, <错误信息>)
```

其中，<自定义错误号>是–20000～–20999 的整数值，<错误信息>是触发异常时返回的错误文本。

下面的示例演示了当插入一个新学生到 Student 表时，如果该学号已经存在于 Student 表中，则直接生成并触发一个“学生已存在”的异常。SQL%FOUND 是 Oracle 提供的系统状态变量，如果 Select 语句执行返回的结果集非空，则 SQL%FOUND 为 True，否则为 False。

```
--插入一个新学生 001
Declare
    sno Varchar2(20);
Begin
    Select s# Into sno From Student Where s# = '001';
    If SQL%FOUND Then
       Raise_Application_Error(-20001, '学生已存在');
    Else
       Insert Into Student(s#) Values('001');
    End If
End;
```

需要注意的是，一旦 PL/SQL 程序执行 Raise_Application_Error 语句，就会自动终止整个程序的执行。因此，如果要在程序中自定义多种异常，使用 Raise_Application_Error 语句直接触发异常容易导致程序中出现很多的 Raise_Application_Error 语句，并且也使得程序的执行流程变得难以理解。因此，在实际开发时较少使用这种直接触发的方式，而是使用下面这种先声明后统一触发的更灵活、更容易理解的方式。

2) 声明并触发自定义异常

这种自定义异常处理的方式在 PL/SQL 编程中较为常用。它首先定义 Exception 类型的变量，然后在程序段中使用 Raise 语句触发异常，最后在 Exception 段中统一对所有触发的异常(包括系统定义的标准异常和用户自定义的异常)进行处理。

下面的示例演示了声明并触发自定义异常的程序代码。首先定义了一个名为 exp、类型为 Exception 的变量。当插入重复学生记录时，Raise exp 语句就会触发一个异常。注意，Raise 语句并不会终止程序的运行，而是跳转到后面的 Exception 段开始执行错误陷阱中的代码。在 Exception 段中，使用 When exp 就可以捕捉前面触发的 exp 异常并进行处理。例如，示例中使用 Raise_Application_Error 返回了一个错误码–20001 以及一条错误文本“学生已存在”。因此，当把这段程序作为一个存储过程编写完毕后，前端的开发语言(VB、C#等)在调用该存储过程时，如果触发了 exp 异常，就会接收到错误码–20001 以及相应的错误消息，然后前端程序可以根据这一错误码采取相应的操作。

值得一提的是，在 When Others 语句中用 Raise_Application_Error 返回了 SQLCODE 和 SQLERRM。它们均为 Oracle 的系统参数，存储了发生异常时的错误码和错误消息(系统内部定义的错误文本)。

```
--重复插入学生 001
Declare
    sno Varchar2(20);
    exp Exception;                          --声明一个 Exception 变量
Begin
    Select s# Into sno From Student Where s# = '001';
    If SQL%FOUND Then
    Raise exp;                              --触发一个异常
    Else
       Insert Into Student(s#) Values('001');
    End If
Exception
    When exp Then
Raise_Application_Error(-20001, '学生已存在');
    When Others Then
Raise_Application_Error(SQLCODE, SQLERRM);
End;
```

这个示例演示的也是 Oracle PL/SQL 一般的异常处理框架。该框架可以总结为如下几个要点：

(1)将自定义的异常声明为 Exception 变量。

(2)在预期触发异常的程序段中使用 Raise 语句触发异常。

(3)在程序的尾部用 Exception 段统一处理所有的异常，包括系统定义的标准异常和用户自定义异常。

(4)Exception 段中对于预期的特定异常使用“When+异常名”的方式进行针对性处理，最后加上 When Others 处理所有未知异常(When Others 是必需的，可以保证错误陷阱的完备性)。

5.4.3　Microsoft SQL Server T-SQL 的异常处理

T-SQL 所采取的异常处理思路也是错误陷阱机制，但是实现方法与 MySQL 和 Oracle 有差别。

T-SQL 处理异常的简单方式是设置一个异常处理器，然后在程序中捕捉异常并跳转到错误处理器统一处理。下面给出了这种方式的基本代码框架：

```
On Error Goto Error_Handler
Begin Transaction
<DML 语句>
If <自定义异常> Goto Error_Handler

Commit Transaction
```

```
    Return

    Error_Handler:
       <异常处理语句>
       Rollback Transaction
    Return
```

在上述框架中，首先定义了一个异常处理器 Error_Handler 用于捕捉并处理所有的异常。在程序开始时，添加 On Error Goto Error_Handler 语句，用于定义错误陷阱并指明一旦发生错误就跳转到标号为 Error_Handler 的代码开始执行后面的语句。这里的 Error_Handler 是用户命名的，在实际开发中可以使用自己习惯的名称。通过将 On Error Goto Error_Handler 和 Error_Hander 两部分结合，可以捕捉所有的系统运行时错误。在 Error_Handler 后面的<异常处理语句>中，可以再根据异常类型执行不同的操作。如果需要自定义异常（例如，成绩低于 60 分，账户余额低于 10 元，等等），这类异常并不会被 Microsoft SQL Server 触发，可以在程序中使用 If 语句判断是否发生了自定义异常，一旦发生，就用 Goto 语句跳转到 Error_Handler 即可。

由于过程化 SQL 一般使用在存储过程中，通常都会使用事务编程，因此，在异常处理时，Error_Handler 中的异常处理操作通常需要执行 Rollback Transaction 语句以取消事务。如果所有程序都正确执行，没有触发任何异常，则需要在 Error_Handler 之前 Commit Transaction 并使用 Return 语句终止程序。注意，Return 语句是必需的，否则程序会接着执行 Error_Handler 后面的语句。

5.5　事 务 编 程

事务（Transaction）是数据库系统中的一个核心概念，它是 DBMS 保护数据一致性的技术基础。关于事务处理的详细内容将在后面内容中介绍，本节重点介绍过程化 SQL 中对于事务编程的支持。

事务是不可分的一个 DML 操作序列。以银行转账为例，假如 A_1 账户要转账到 B_1 账户 100 元，则数据库层面需要执行两次 Update 操作，类似下面的语句：

```
    Update account Set balance = balance - 100 Where account_ID = 'A1';
    Update account Set balance = balance + 100 Where account_ID = 'B1';
```

上面这两个 Update 操作对于银行转账业务来说是不可分的，即要么全部都执行，要么一个也不执行。如果只执行了其中一个 Update 操作，就会导致转账结果出现错误。在数据库应用系统中，如果用户通知 DBMS 上面的两个操作属于一个事务，DBMS 中的事务管理器就可以保证事务的 ACID 性质。事务的 ACID 性质是关系 DBMS 实现数据库保护的重要基础。

虽然 MySQL、Oracle 等主流的 DBMS 都提供了对事务的支持，但是，DBMS 本身无法判断哪些操作应该作为一个事务进行处理。这一点需要由用户根据实际应用的要求进行定义。为此，在数据库应用开发时，需要在过程化 SQL 中为用户提供事务编程的支持。事务编程的主要目的是：告知 DBMS 哪些操作是事务，何时可以提交事务，何时需要取消事务。

事务编程主要涉及三条语句。

(1) Begin Transaction。

当用户需要开始一个事务时，必须用 Begin Transaction 语句告知 DBMS 事务开始了。Begin Transaction 之后的所有 DML 操作将作为一个事务进行处理，DBMS 将保证这些操作的 ACID 性质。

(2) Commit Transaction。

Commit Transaction 语句告知 DBMS 事务正常结束，因此事务对数据库所做的全部修改都写入了持久存储介质，用数据库术语讲，就是事务的持久性已经生效。Commit Transaction 语句的执行表明一个事务执行完成，可以看作事务正常结束的标志。

(3) Rollback Transaction。

Rollback Transaction 语句告知 DBMS 事务因为特殊原因(出现预期或者非预期的各种错误)而取消了，因此事务已经执行的操作将全部取消。这是事务的原子性和一致性的要求。原子性要求事务要么所有操作全部执行，要么一个操作也不执行；一致性要求事务不能停留在事务执行的中间状态。Rollback Transaction 语句可以看成事务取消的标志。

图 5-12 给出了 MySQL 中使用事务编程处理银行转账的示例。在第 8 行，用 Start Transaction 语句告知 DBMS 事务开始，之后的所有 DML 操作都将属于这一个事务。在第 22 行，当所有异常都没有出现(此例中只定义了两种异常：账户不存在和余额不足)时，使用 Commit 语句提交事务。只有执行 Commit 语句，第 18 行和第 19 行的两个 Update 语句对数据库所做的修改才能真正生效，即写到磁盘数据库文件中。如果出现了异常，如账户不存在或者转出账户余额不足，则在第 25 行使用 Rollback 语句取消整个事务。

```
1
2   delimiter //
3   CREATE PROCEDURE transfer(IN id_from INT, IN id_to INT, IN amount INT, OUT state INT)
4   BEGIN
5       DECLARE s INT DEFAULT 0;
6       DECLARE a INT;
7       DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET s = 1;
8       START TRANSACTION;
9       SELECT count(*) FROM account WHERE id = id_from or id=id_to INTO a;
10      IF  a < 2 THEN -- 至少有一个账户不存在
11          SET s = 2;
12      END IF;
13
14      SELECT balance FROM account WHERE id = id_from INTO a;
15      IF  a < amount THEN  -- 余额不足
16          SET s = 3;
17      END IF;
18      UPDATE account SET balance = balance - amount WHERE id = id_from;
19      UPDATE account SET balance = balance + amount WHERE id = id_to;
20      IF s = 0 THEN
21          SET state = 0;
22          COMMIT;
23      ELSE
24          SET state = -1000;
25          ROLLBACK;
26      END IF;
27  END //
28  delimiter;
```

图 5-12　MySQL 中使用事务编程处理银行转账的示例

图 5-13 显示银行转账事务程序的测试结果。图 5-13(a)显示了当账户不存在时的执行结果，此时事务被 Rollback，程序输出–1000 指示了这一错误。图 5-13(b)显示了正确转账时的执行结果，此时 state 值为 0。

其他业务过程中的事务编程思路与图 5-12 的示例类似：首先，用户需要分析业务处理是否需要使用事务编程。如果需要，则使用 Start Transaction 语句开始事务。然后，采用异

常处理框架捕捉可能的异常。最后，如果执行到程序最后没有出现异常，则使用 Commit 语句提交事务，否则，使用 Rollback 取消事务。

(a) 账户不存在导致事务取消　　(b) 转账事务正常完成　　(c) 转账完成前后的表数据对比

图 5-13　基于事务的银行转账测试结果

在事务编程中，关键的问题在于确定是否使用事务。这需要用户熟悉事务的概念以及业务处理的需求。如果应该使用事务的程序中没有采取事务编程，那么程序中就隐含了导致数据库不一致的错误。对于图 5-12 所示的银行转账示例，如果不使用事务，那么第 18 行的 Update 语句就会直接更新数据库中的账户余额。如果执行完第 18 行的 Update 语句但是还没开始执行第 19 行的语句时 DBMS 突然出现了故障，如突然宕机或者掉电了，银行账户数据库中就会出现 *A* 账户的钱少了但是 *B* 账户的钱没有增加的情况。这是现实应用中不允许的。事实上，对于转账操作，数据一致性的要求是两个账户的余额总和应该在转账前后是相等的。如果违背了这一点，意味着数据就处于不一致的状态。对于用户而言，不一致的数据就是错误的数据，无法在实际应用中使用。

ANSI SQL 标准中定义的事务编程语句与 MySQL、Oracle 等有所区别，表 5-3 给出了 MySQL、Oracle、Microsoft SQL Server 与 ANSI SQL 之间的差异。可以看到，Microsoft SQL Server 的事务编程语句与 ANSI SQL 完全相同，MySQL 和 Oracle 略有不同。特别地，Oracle 只提供了 Commit Work（相当于 Commit Transaction）和 Rollback Work（相当于 Rollback Transaction），但不提供 Begin Transaction 语句。这是因为 Oracle PL/SQL 编程中默认任何程序都是以事务形式执行的，所以程序一开始就意味着开始了事务。

表 5-3　不同 DBMS 在事务编程语句格式上的差异

事务编程语句	DBMS			
	MySQL	Oracle	Microsoft SQL Server	ANSI SQL
开始事务	Start Transaction	—	Begin Transaction	Begin Transaction
提交事务	Commit	Commit Work	Commit Transaction	Commit Transaction
取消事务	Rollback	Rollback Work	Rollback Transaction	Rollback Transaction

5.6　游　　标

过程化 SQL 程序中的变量每次只能存储单个值或者单条记录，但由于过程化 SQL 中可能使用了 Select 查询返回了多行记录，此时过程化 SQL 中没有相应的变量类型可以存储这

种动态的记录集。如何在过程化 SQL 程序中支持动态记录集的存储和处理，是过程化 SQL 编程中面临的一个问题。这一问题同样存在于嵌入式 SQL 程序中。如果在 C 程序中嵌入 SQL 语句，当 SQL 查询返回动态记录集时，C 语言同样也无法直接存储和处理记录集。

5.6.1　游标的概念

过程化 SQL 和嵌入式 SQL 目前都采用游标(Cursor)来解决这一问题。游标技术的基本思路是通过在客户端或者数据库服务器开辟一块内存区域来存储 SQL 语句返回的动态记录集。由于记录集被存储在游标指向的内存区域中，可以在程序中根据游标指针访问该内存区域，通过一次读取一条记录的方式，循环扫描游标指向的内存区域，就可以在过程化 SQL 程序中存取 SQL 语句返回的动态记录集。游标技术解决了过程化SQL与声明性的SQL之间的数据交互问题。

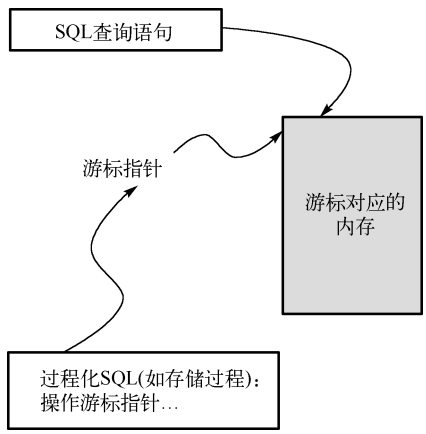


图 5-14　过程化 SQL 程序利用游标存取 SQL 返回的动态记录集的基本思路

图 5-14 给出了过程化 SQL 程序利用游标存取 SQL 返回的动态记录集的基本思路。从编程的角度看，游标可以看成一个指向一块内存区域的指针，称为“游标指针”。当在过程化 SQL 程序中声明了一个对应一条 Select 语句游标(即游标类型的变量)后，DBMS 就会根据 Select 查询返回的记录集大小为游标分配实际的内存区域，然后把记录集放入内存区域，最后让游标指针指向该内存区域的起始地址。之后，过程化 SQL 程序就可以根据游标指针去读取内存区域中的记录。由于过程化 SQL 程序中的变量只能存储单条记录，所以根据游标指针读取游标内的记录需要按照一次一条的方式执行。因此，读取游标内整个记录集一般要通过一次循环来实现。

5.6.2　游标操作

在过程化 SQL 中使用游标需要遵循一定的操作顺序，即声明游标、打开游标、读取游标中的记录、关闭游标。

1. 声明游标

声明游标使用 Declare Cursor 语句，格式如下：

```
Declare Cursor <名称> For <Select 语句> (MySQL 和 Microsoft SQL Server)
Declare Cursor <名称> IS <Select 语句> (Oracle)
```

由于游标的目的是缓存 Select 查询返回的动态记录集，所以 Declare Cursor 语句中 For 或者 IS 后面只能跟 Select 语句。但是，Declare Cursor 语句中的 Select 语句在声明时并不执行，因此游标对应的物理内存区域还未被 DBMS 分配。此时的游标指针为 Null。

例如，下面的示例声明了一个名为 cs_stu 的游标，用于存储 Select 查询返回的学生记

录集。此处给出的是 MySQL 和 Microsoft SQL Server 的示例，Oracle PL/SQL 的游标声明类似，只不过 For 关键字须改成 IS。

```
--声明一个游标，用于存放所有学生记录
Declare Cursor cs_stu For Select * From Student;
```

2. 打开游标

打开游标使用 Open 语句，格式如下(MySQL、Oracle、Microsoft SQL Server 使用相同格式的 Open 语句)：

```
Open <游标名>
```

打开游标时，DBMS 会执行游标声明中给出的 Select 语句，并根据返回的记录集大小为游标分配物理内存，并将记录集放入游标指向的内存区域。

3. 读取游标中的记录

打开游标后，游标变量指向存储了记录集的内存区域的起始地址，即记录集第一条记录的地址。此时，可以使用 Fetch 语句读取游标指向的当前记录，执行 Fetch 语句后游标指针会自动指向下一条记录。如果要读取游标中的所有记录，通常需要使用一个循环结构。

Fetch 语句的格式如下(MySQL、Oracle、Microsoft SQL Server 使用相同格式的 Fetch 语句)：

```
Fetch <游标名> Into <变量表>
```

例如，下面的 MySQL 代码使用游标读取了 CS 系的所有学生记录：

```
--返回所有 CS 系学生记录
Begin
   Declare state Int Default 0;
   Declare s1, s2 Varchar(50);
   Declare Cursor cs_stu For Select sno, sname From Student Where dept
      = 'CS';
   Declare Continue Handler for NOT FOUND Set state = 1;
   Open cs_stu;
   Repeat
      Fetch cs_stu Into s1,s2;

      Until state = 1
   End Repeat;
   …
End
```

Oracle PL/SQL 的代码与此类似。下面的代码给出了 PL/SQL 中使用游标读取 CS 系学生记录的代码示例：

```
--返回所有 CS 系学生记录
Declare
```

```
    Cursor cs_stu IS select * From Student Where dept = 'CS';
    stu Student%ROWTYPE;
Begin
    Open cs_stu;
    Fetch cs_stu Into stu;
        While cs_stu%FOUND Loop
        DBMS_OUTPUT.PUT_LINE(…);
        Fetch cs_stu Into stu;
    End Loop;
    …
End
```

4. 关闭游标

由于游标打开之后会占用内存，所以游标使用完毕后需要使用 Close 语句关闭以释放其占用的内存空间。关闭游标的 Close 语句格式如下：

```
Close <游标名>
```

图 5-15 给出了一个完整的使用游标的示例，左侧图给出了使用游标的存储过程代码，右侧图给出了测试结果。此处以 MySQL 为例，Oracle 和 Microsoft SQL Server 的游标编程与此类似。

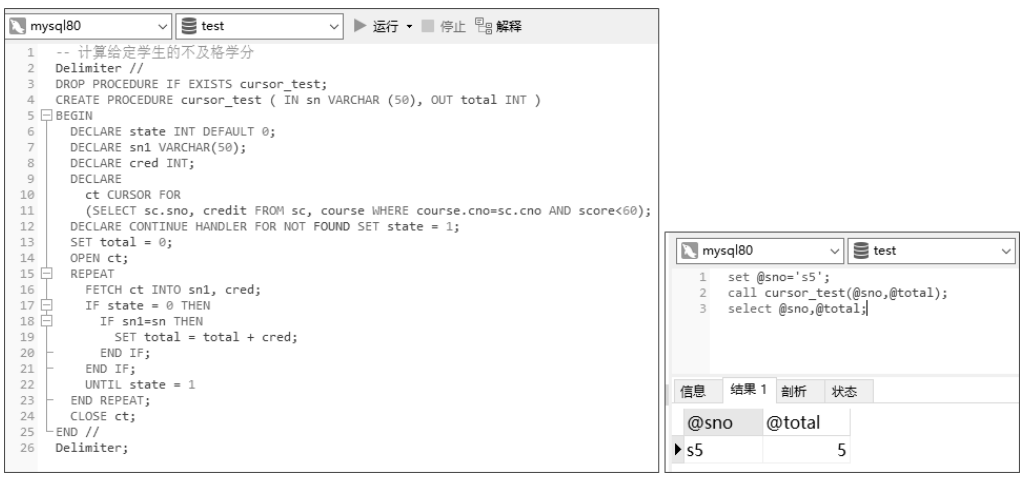


图 5-15　游标编程实例：计算给定学生不及格学分

5.7　存 储 过 程

过程化 SQL 在 DBMS 中最主要的用途之一就是编写存储过程。顾名思义，存储过程就是存储在 DBMS 上的过程。通过存储过程，可以将一些业务处理放到数据库服务器执行，从而实现 C/S 架构中计算任务从客户端到服务器的迁移，均衡客户端和服务器的计算负载。

5.7.1　存储过程的概念

存储过程(Stored Procedure)是在大型数据库系统中，一组为了完成特定功能的 SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数(如果该存储过程带有参数)来执行它。一般的过程化 SQL 程序只是匿名的语句块，由于没有程序名称，因此也无法在 DBMS 上存储。通过创建存储过程，可以将一段过程化 SQL 程序以存储过程的方式预先编译存储在 DBMS 上，并且可以被用户或应用程序反复调用执行。

存储过程的概念与 C 语言中的函数、Basic 语言中的过程类似，都是由多条语句构成的程序体，都具有唯一的名称，都可以持久存储并被反复调用。

目前的 DBMS 一般都支持两种类型的存储过程：系统存储过程和用户自定义的存储过程。系统存储过程是 SQL 自身提供的，用于系统管理、对象管理等，例如，Microsoft SQL Server 中的系统存储过程 sp_password 可以用于修改用户密码。用户自定义的存储过程是用户自行用过程化 SQL 编写的存储过程。

5.7.2　存储过程的作用

存储过程的实质就是部署在数据库端的一组预先定义好的过程化 SQL 代码。使用存储过程有以下的优点。

(1)存储过程大大增强了 SQL 的功能和灵活性。存储过程可以用过程化控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

(2)可保证数据库的安全性。通过存储过程可以使没有权限的用户在控制之下间接地存取数据库，从而保证数据的安全。

(3)可保证数据库的完整性。关系数据库提供的三类完整性约束在某些时候无法满足用户给定的完整性约束条件，此时可以通过存储过程来加以实现。通过存储过程可以使相关的动作在一起发生，从而维护数据库的完整性。

(4)在运行存储过程前，数据库已对其进行了语法和句法分析，并给出了优化执行方案。这种已经编译好的过程可极大地改善 SQL 语句的性能。由于执行 SQL 语句的大部分工作已经完成，所以存储过程能以极快的速度执行。

(5)可以降低网络的通信量。

(6)将体现企业规则的运算程序放入数据库服务器中，以便集中控制。当企业规则发生变化时在服务器中改变存储过程即可，无须修改任何应用程序。企业规则的特点是要经常变化，如果把体现企业规则的运算程序放入应用程序中，则当企业规则发生变化时，就需要修改应用程序，工作量非常大(修改、发行和安装应用程序)。如果把体现企业规则的运算放入存储过程中，则当企业规则发生变化时，只要修改存储过程就可以了，应用程序无需任何变化。

5.7.3　存储过程的创建和删除

存储过程使用 Create Procedure 语句创建，但 MySQL、Oracle、Microsoft SQL Server 在语法上略有不同。

1. MySQL

MySQL 中创建存储过程的 Create Procedure 语句格式如下，其中 Begin…End 之间的<变量定义>、<过程化 SQL>和<异常处理>可以使用前面介绍过的所有语法结构。

```
Create Procedure <名称>(参数表)
Begin
    <变量定义>
    <过程化 SQL>

    <异常处理>
End;
```

2. Oracle

Oracle PL/SQL 创建存储过程的格式如下。可以看到，它和 MySQL 的主要区别在于变量定义的位置以及异常处理的方式。

```
Create Procedure <名称>(参数表)
AS | IS
    <变量定义>
Begin
    <
PL/SQL 语句>
  Exception
    <异常处理>
End;
```

3. Microsoft SQL Server

Microsoft SQL Server T-SQL 创建存储过程的格式如下：

```
CreateProcedure<名称>(参数表)
AS
    <T-SQL 语句块>
```

存储过程的参数有三种类型：输入参数、输出参数以及输入/输出参数。其中 MySQL 和 Oracle 均支持这三种类型的参数，而 Microsoft SQL Server 只支持输入和输出参数(但也能够满足存储过程对参数使用的基本要求)。

(1)在 MySQL 中，输入参数、输出参数和输入/输出参数分别用 IN、OUT 和 INOUT 表示，如 IN name Varchar(50)，OUT result Int。

(2)在 Oracle 中，输入参数、输出参数以及输入/输出参数分别用 IN、OUT 和 IN OUT 表示，如 name IN Varchar2、result OUT number。

(3)在 Microsoft SQL Server 中，默认参数类型为输入参数，如果要表示输出参数，则需要在参数后面添加[OUTPUT]，如(@dept Varchar(50), @num Int OUTPUT。第一个参数@dept 是输入参数，第二个参数@num 是输出参数。

可以看到，MySQL、Oracle 等不同的 DBMS 在存储过程的定义和参数类型等方面存在一些细微的差别。因此，对于过程化 SQL 和存储过程，重点是掌握它们的用途，具体的语法格式可以在实际开发中结合实际使用的 DBMS 进行学习。

【例 5-1】 下面是一个 Microsoft SQL Server 中使用 T-SQL 创建存储过程的示例。该示例假设有基本表 Employee(eno, name, deptID)，根据给定的部门号 deptID 统计该部门的员工数并返回：

```
Create Procedure emps_of_dept(@dno Int, @emps Int OUTPUT)
AS
Select @emps = count(eno) From Employee Where deptID = @eno
```

该存储过程根据用户输入的部门号@dno，计算该部门中的员工总数并赋予输出参数@emps。调用该存储过程时，应用程序就可以直接访问@emps 以取得最后的统计结果。

【例 5-2】 下面的示例演示了使用 Oracle PL/SQL 创建存储过程的示例。该存储过程使用参数插入了一条新的学生记录。

```
Create Procedure AddStudent(v_s# IN Varchar2, v_sname IN Varchar2,
                            v_age IN number)
AS
Begin
  Insert Into Student(s#, sname, age) Values(v_s#, v_sname, v_age);
End;
```

存储过程的删除使用 Drop Procedure 语句，例如，下面的语句删除了存储过程 emps_of_dept：

```
Drop Procedure emps_of_dept
```

5.7.4　函数的调用

函数是具有一个返回值的存储过程。存储过程本身也提供了输出参数用于返回结果，而且函数只能返回一个值，而存储过程则可以通过定义多个输出参数返回多个值。实际开发中，究竟使用存储过程还是函数，可以根据实际需要而定。

函数的定义和存储过程类似，只不过 Create Function 语句替代了 Create Procedure 语句，此外增加了 Return 语句用于返回值。

【例 5-3】 下面的示例给出了 Oracle PL/SQL 编写的返回一个系学生人数的函数。

```
Create or Replace Function StudentCount(DeptNo IN Varchar2)
Return Number
AS
   v_count Number: = 0;
Begin
   select count(s#) Into v_count From Student Where dept = deptno;
   return v_count;
End;
```

【例 5-4】 图 5-16 的示例给出了 MySQL 中计算学生 GPA 的函数实例。此处，GPA 的计算公式为 $\mathrm{GPA}=\dfrac{\sum(\text{课程学分}\times\text{课程学分绩点})}{\sum\text{课程学分}}$。图 5-17 给出了调用函数计算 GPA 的实例。

```
1  -- 计算给定学生的GPA
2  Delimiter //
3  DROP FUNCTION IF EXISTS fun;
4  CREATE FUNCTION fun(sn VARCHAR(50))
5  RETURNS FLOAT
6  READS SQL DATA
7  BEGIN
8     DECLARE state INT DEFAULT 0; -- cursor结束标记
9     DECLARE grade,cred,total_c,total_g FLOAT DEFAULT 0;
10    DECLARE sn1 VARCHAR(50);
11    DECLARE c_count INT;
12    DECLARE t, gpa FLOAT DEFAULT 0;
13    DECLARE
14      ct CURSOR FOR
15      (SELECT score,credit FROM sc,course c WHERE sc.cno=c.cno AND sno=sn AND score IS NOT NULL);
16    DECLARE CONTINUE HANDLER FOR NOT FOUND SET state = 1;
17    OPEN ct;
18    REPEAT
19      FETCH ct INTO grade,cred; -- 每一门课程的成绩和学分
20      IF state = 0 THEN
21        CASE
22          WHEN grade>=95 THEN SET t=4.3;
23          WHEN grade>=90 AND grade<95 THEN SET t=4.0;
24          WHEN grade>=85 AND grade<90 THEN SET t=3.7;
25          WHEN grade>=82 AND grade<85 THEN SET t=3.3;
26          ELSE SET t=3;
27        END CASE;
28        SET total_g=total_g + t*cred; -- 计算总的学分*绩点
29        SET total_c=total_c + cred; -- 计算总的学分
30      END IF;
31      UNTIL state = 1
32    END REPEAT;
33    CLOSE ct;
34    SET gpa=total_g/total_c;
35    RETURN gpa;
36 END //
37 Delimiter;
```

图 5-16　MySQL 函数实例：计算学生 GPA

```
mysql80   test
1  SELECT
2    sno,
3    sname,
4    fun(sno) AS GPA
5  FROM
6    student;
7
```

信息　结果 1　剖析　状态

sno	sname	GPA
s1	a	3.57143
s2	b	3
s3	c	3
s4	d	(Null)
s5	c	3.28571

图 5-17　MySQL 调用 fun() 函数计算 GPA 示例

5.7.5　存储过程的调用

存储过程和函数在 DBMS 端调试时一般用 Execute 语句或者 Call 语句进行调用。其中，Oracle 和 Microsoft SQL Server 支持 Execute 语句，而 MySQL 则使用 Call 语句，格式如下：

```
Execute <存储过程名>
Call <存储过程名>
```

Microsoft SQL Server 的存储过程在调用时具有返回值，该返回值是一个整数，指示存储过程的执行状态。存储过程成功执行时系统默认返回 0。如果想通过存储过程状态值来传达一些关于执行过程的特定信息，则可以使用 Return 语句返回特定的整数值。需要注意的是，–99～0 是系统保留的，因此用户可以通过返回–99～0 之外的整数值来反映存储过程的执行状态。

以例 5-1 中创建的存储过程 emps_of_dept 为例，可以在 Microsoft SQL Server 中按下面的方法进行调用：

```
Declare @eno Int, @emps Int
Decalre @ret Int                    --返回值，必须是整数类型
Set @eno = 100
Execute @ret = emps_of_dept(@eno, @emps OUTPUT)
```

上面这种调用方法一般只在调试存储过程时使用。编写存储过程的目的是为客户端应

用程序服务，因此存储过程一般都在客户端应用程序中通过程序设计语言调用。但在程序设计语言中调用存储过程的方法不尽相同，与各个程序设计语言以及数据访问中间件的语法有关。例如，在微软的 Visual Studio 开发工具中，一般用 ADO（ActiveX Data Objects）进行数据库访问。ADO 中存储过程的调用一般使用 Command 对象。详细的方法可参考一些编程书籍。

5.7.6　存储过程的应用

下面以 Microsoft SQL Server 为例，讨论一个稍微复杂一点的存储过程示例。通过该示例来说明存储过程的一般编写思路。

假设有下面的两个基本表：

```
Employee (EmpID, salary, name, deptID)
Department (DeptID, name, telephone, fax, manager)
```

下面的存储过程根据输入的部门名称，返回该部门的员工的数量和平均工资。

```
CREATEPROCEDURE samp
(@dept Varchar(50), @num IntOUTPUT, @av Float OUTPUT)
AS
   DECALRE @a Int                    --数量
   DECALRE @b Float                  --平均工资
   IF Not Exists(Select * From Department Where name = @dept)
        RETURN -100                  --部门不存在，通过返回-100 表示
   Select @a = (Select Count(*) From Employee, Department
   Where Employee.DeptID = Depatment.DeptID and Department.name = @dept)
   IF @a = 0                         --该部门不存在员工
        RETURN -101
   Select @b = (Select avg(Employee.salary) From Employee, Department
Where Employee.DeptID = Depatment.DeptID and Department.type = @deptID)
SET @num = @a
SET @av = @b
```

上述存储过程在 Microsoft SQL Server 中调试时一般可用下面的方法：

```
DECALRE @empNum Int
DECALRE @empAvgSalary Float
DECALRE @returnstat Int
EXECUTE @returnstat = samp('销售部', @empNum OUTPUT, @empAvgSalary OUTPUT)
IF @returnstat = -100
    PRINT '部门不存在'
ELSE
    IF @returnstat = -101
        Print '员工数为 0'
    ELSE
        BEGIN
            PRINT '员工数 = ' + @empNum
            PRINT '平均工资 = ' + @empAvgsalary
        END
```

在实际数据库应用系统开发中，存储过程通常是被前端开发语言调用的。下面以 Microsoft Visual Basic（简称 VB）为例，说明在前端开发语言中如何调用存储过程。

VB 需要使用微软的 ADO 技术调用存储过程。ADO 是微软推出的数据库访问中间件，目前 Visual Studio 系列的开发工具都支持 ADO 技术，如 VB、VC++、ASP 等。简单讲，ADO 是一个对象集合，它通过不同的对象来完成不同的数据库访问功能，例如，Connection 对象用于建立数据库连接，Recordset 对象用于记录操作，Command 对象用于调用存储过程或执行 SQL 语句。

存储过程的调用主要通过 Command 对象来实现。下面以前面定义的 samp 存储过程为例，给出在 VB 中调用 samp 的代码示例。

```
Sub SP_example()
  Dim cmm as New ADODB.Command      '声明 Command 对象
  Set cmm.ActiveConnection = cnn    'cnn 为数据库连接，在此假设其已建立
  Cmm.CommandText = "samp"          '存储过程名
  Cmm.CommandType = adCmdStoredProc 'Command 类型设为存储过程
  '下面的语句为存储过程调用添加参数
  'Return 参数
  cmm.Parameters.Append cmm.CreateParameter("Return", adInteger,
      adParamReturnValue, 4, 0)
  '输入参数
  cmm.Parameters.Append cmm.CreateParameter("DeptName", adVarchar,
      adParamInput, 50, "")
  '下面这两个是输出参数
  cmm.Parameters.Append cmm.CreateParameter("EmpCount", adInteger,
      adParamOutput, 4, 0)
  cmm.Parameters.Append cmm.CreateParameter("AvgSalary", adNumeric,
      adParamOutput, 8, 0)
  '传递参数值
  Cmm.Parameters("DeptName") = "销售部"
  Cmm.Execute                                  '执行
  If cmm.Parameters("Return") = -100 then
     Msgbox  "部门不存在"
     Exit sub                                  '退出程序
  Elseif cmm.Parameters("Return") = -101
     Msgbox "部门没有员工"
     Exit sub
  End if
  Msgbox  "销售部员工数 = "& cmd.parameters("EmpCount")
  Msgbox  "销售部员工平均工资 = "& cmd.parameters("AvgSalary")
End Sub
```

总体而言，在数据库应用系统开发中使用存储过程和函数比单纯的 SQL 增删改查操作要复杂，编写、调试和使用存储过程也相对较复杂。因此，用户可以在实际开发中根据需求来确定需要编写哪些存储过程和函数。而且，所有开发的存储过程和函数必须书写相应的文档，作为数据库设计报告的一部分提供给前端编程人员参考。

5.8 触 发 器

触发器是过程化 SQL 的另一个主要的用途。一般在数据库应用系统中，过程化 SQL 主要用于编写存储过程和触发器。触发器与存储过程类似，都是用过程化 SQL 编写的有名称的程序。但是，触发器是由用户操作触发执行的，而存储过程一般是由用户手工调用执行的。之所以有这些差别，是因为触发器和存储过程的设计目的不一样。一般地，在系统中使用触发器的目的是实现一些特殊的完整性约束，而存储过程则主要用于实现业务计算任务的迁移。

5.8.1 触发器的概念

触发器是一种特殊类型的存储过程，它不同于前面介绍过的存储过程。触发器主要是通过事件进行触发而执行的，而存储过程可以通过存储过程名字而直接调用。当对某一表或视图进行 Update、Insert、Delete 等操作时，DBMS 就会自动执行触发器所定义的 SQL 语句，从而确保对数据的处理必须符合由这些 SQL 语句所定义的规则。

触发器和存储过程的区别可以归纳为下面几点：

(1)触发器是由用户操作自动触发执行的，而存储过程是通过名字显式调用执行的；

(2)触发器没有参数，而存储过程一般具有参数；

(3)触发器必须关联到某个特定的表或视图上，而存储过程没有这一要求。

触发器一般用于执行数据完整性检查，因此在基本表上执行 Insert、Delete 或 Update 操作才能定义相应的触发器(Select 操作不会破坏数据完整性)。

5.8.2 触发器的作用

触发器的主要作用就是实现由 SQL 提供的四类完整性约束(Primary Key、Foregin Key、Unique、Check)所不能保证的复杂的完整性和数据的一致性。除此之外，触发器还有其他许多不同的作用。

(1)强化约束：触发器能够实现比 Check 语句更为复杂的约束。

(2)跟踪变化：触发器可以侦测数据库内的操作，从而不允许数据库中未经许可的指定更新和变化。

(3)级联运行：触发器可以侦测数据库内的操作，并自动地级联影响整个数据库的各项内容。例如，某个表上的触发器中包含对另一个表的数据操作(如删除、更新、插入)，而该操作又导致该表上的触发器被触发。

由此可见，触发器可以解决高级形式的业务规则或复杂行为限制以及实现定制记录等一些方面的问题。例如，触发器能够找出某个表在数据修改前后状态的差异，并根据这种差异进行一定的处理。此外，在一个表上可以针对不同类型的操作(Insert、Update、Delete)定义多个触发器，也可以对同一个类型操作定义多个触发器，从而能够对数据操作进行多种不同的处理。

例如，假设有一个学生表和一个选课表：Student(sno, sname, age, status)、SC(sno, cno, score)。现在想实现当学生有 3 门功课不及格时，将学生的状态(status)修改为“不合格”。

这一功能可以通过在 SC 表上设计一个触发器来实现：当在 SC 表中插入或者更新记录时，自动检查是否有学生满足“不合格”条件。

但是，触发器的编程调试相对较麻烦。而且，过多的触发器通常会降低数据库应用系统的性能。当运行触发器时，系统处理的大部分时间花费在触发操作的处理上。此外，触发器的使用也会带来潜在的运行错误风险。因此，需要根据应用的要求来设计适当的触发器，以实现数据完整性、性能和可靠性之间的均衡。

5.8.3　触发器的种类

根据触发器执行的顺序以及执行方式等，触发器可以划分为以下几种不同的类型。

1. 按触发器与 DML 语句执行的相对顺序分类

触发器都是 DML 语句所触发的，但是触发器中的过程化 SQL 语句可以在 DML 语句执行之前执行，也可以在 DML 语句执行之后再执行。由此，可以将触发器分为下面三类。

(1) 先触发器(Before Trigger)：触发器在 DML 语句执行之前执行。

(2) 后触发器(After Trigger)：触发器在 DML 语句执行之后执行。

(3) 替代触发器(Instead Trigger)：当 DML 语句触发了触发器时，不执行 DML 语句，而是只执行触发器中的过程化 SQL 语句。

2. 按触发器执行方式分类

按触发器执行方式可以将触发器分为行级触发器和语句级触发器。

(1) 行级触发器：对由触发的 DML 语句所导致变更的每一行触发一次，因此一个触发器可能被一个 DML 语句触发多次。

(2) 语句级触发器：触发器被一个 DML 语句触发执行一次。

3. 特殊的触发器

触发器一般情况下都是被 DML 语句所触发的，但某些 DBMS(如 Oracle)还支持一些特殊的触发器，允许被非 DML 操作触发。这类触发器主要包括两种。

(1) DDL 触发器：执行 DDL 语句时触发的触发器。

(2) DB 事件触发器：当 DBMS 启动、停止服务、登录等事件发生时触发的触发器。

MySQL、Oracle 等不同的 DBMS 所支持的触发器类型有所不同。表 5-4 给出了 MySQL、Oracle 和 Microsoft SQL Server 支持的触发器类型对比。从表中可以看出，Oracle 支持全部的触发器类型，因此功能更为全面。MySQL 相对而言对于触发器的支持较弱，但也可以满足基本的编程需求。

表 5-4　不同 DBMS 支持的触发器类型

触发器类型	DBMS		
	MySQL	Oracle	Microsoft SQL Server
先触发器	√	√	×
后触发器	√	√	√

续表

触发器类型	DBMS		
	MySQL	Oracle	Microsoft SQL Server
替代触发器	×	√	√
行级触发器	√	√	×
语句级触发器	×	√	√
DDL 触发器	×	√	√
DB 事件触发器	×	√	√(仅支持 LOGON 事件)

5.8.4　触发器的创建和删除

触发器通过 Create Trigger 语句创建，删除使用 Drop Trigger 语句。Drop Trigger 语句的格式较简单，语法如下：

```
Drop Trigger <触发器名称>
```

但是，由于 MySQL、Oracle、Microsoft SQL Server 支持的触发器类型不同，因此在 Create Trigger 语句的定义语法上存在差别。

1. MySQL

MySQL 的 Create Trigger 语句基本格式如下：

```
Create Trigger <触发器名称>
    [Before | After | Delete | Insert | Update]  -- 定义触发操作
On<表名>For Each Row  -- 表明是行级触发器
Begin
    <过程化 SQL 程序>  --触发后执行的过程化 SQL 程序
End;
```

其中，Before 和 After 关键字指明是先触发器还是后触发器，Delete、Insert 和 Update 关键字指明触发的操作。触发器一般只针对数据更新操作，不考虑 Select 操作。这是因为触发器的主要目的是实施复杂的数据完整性约束，保证数据的一致性。Select 操作是只读操作，不会破坏数据完整性，因此在触发器中不加考虑。

由于触发器是被动触发执行的，所以触发器与存储过程不同，本身没有参数表，意味着用户无法向触发器传递参数。

此外，MySQL 的一个触发器只允许定义一个触发事件，如 Insert、Delete 或者 Update。如果需要触发多个事件，则必须在一个表上定义多个触发器。这一点与 Oracle 不同：Oracle 允许一个触发器对应多个事件。

2. Oracle

Oracle 对于触发器的支持较强，功能比 MySQL 更为丰富。下面给出了 Oracle PL/SQL 中的 Create Trigger 语句基本格式：

```
Create Trigger <触发器名称>
```

```
    [Before | After | Delete | Insert | Update [of <列名表>] [or
                                --定义触发操作
        Before | After | Delete | Insert | Update [of <列名表>] …]
                                --多个触发操作用 or 连接
    On <表名>  [For Each Row]    --表明是行级触发器，若省略则为语句级触发器
        <PL/SQL 块>
    End;
```

Oracle 和 MySQL 一样支持先触发器和后触发器，但还支持另一些功能，包括：

(1) 当触发操作为 Update 时允许指定列名表，表示只有当更新记录的特定列时才执行触发器；

(2) 一个触发器可以定义多个触发事件，用 or 关键字连接；

(3) 支持语句级触发器。当定义触发器时去掉 For Each Row 选项，表示触发器为语句级触发器。

3. Microsoft SQL Server

Microsoft SQL Server 支持后触发器，但是不支持先触发器，这与 MySQL 和 Oracle 不同。此外，Microsoft SQL Server 支持替代触发器。替代触发器表示触发操作发生时用触发器内的操作代替正常的操作。

Microsoft SQL Server T-SQL 定义触发器的基本格式如下：

```
    Create Trigger<触发器名称>
    On<表名>
        {FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ]}
    As
    {
        <T-SQL 语句块>
    }
```

For 和 After 都表示后触发器，Instead of 表示替代触发器，Insert、Update 和 Delete 可以根据应用的需要定制。与 MySQL 不同，Microsoft SQL Server 中的一个触发器可以定义多个触发操作，例如，使用 After Insert, Delete, Update 定义 Insert、Delete、Update 三个操作都可以触发该触发器的执行。

5.8.5　触发器的使用

不同类型的触发器以及不同的触发操作定义所导致的结果有较大的区别，因此必须清楚触发器执行时的过程和顺序，理解其内部的操作原理，才能结合应用程序的需求定义合适的触发器。本节结合 MySQL、Oracle、Microsoft SQL Server 的不同触发器类型，介绍在实际过程化 SQL 中如何使用触发器。

1. MySQL

由于触发器是由数据更新操作触发的，所以在触发器内部一般需要了解记录更新前后的值，并根据更新前后的记录值确定触发器应该执行的操作。

MySQL 对于行级触发器提供了两个系统变量 old 和 new 分别存储每一条被更新的记录的更新前旧值(old)和更新后新值(new)。表 5-5 给出了这两个系统变量在不同的触发操作时的值。

表 5-5　MySQL 系统变量 old 和 new 在不同触发操作时的值

变量	操作		
	Insert	Delete	Update
old	Null	原记录	原记录
new	新记录	Null	新记录

图 5-18 给出了当插入一条选课记录后更新学生状态的触发器示例。这里假设有一个学生表和一个选课表：Student(sno, sname, age, status)、SC(sno, cno, score)。当学生有 3 门课程不及格时，将学生的状态(status)修改为“不合格”。在此例中，定义了一个针对 Insert 操作的行级后触发器，意味着当用户使用 Insert 语句插入一条记录后，DBMS 将触发执行该触发器中的过程化 SQL 代码。第 5～13 行的过程化 SQL 根据新插入的记录中的学号执行一次聚集查询，统计出成绩低于 60 分的课程数 c_count，并且当 c_count 大于等于 3 时使用 Update 语句修改该学生的状态(status)为“不合格”。注意，第 8 行通过读取系统变量 new 中的 sno 来定位新插入的记录。在第 10、11 行更新学生状态时也通过系统变量 new 选定了要修改的学生记录。

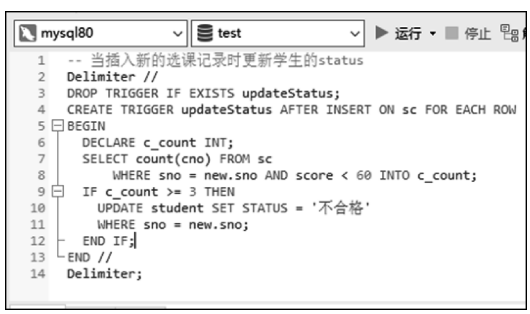


图 5-18　MySQL 触发器示例：自动更新学生状态

图 5-19 给出了测试示例：学号为 s5 的学生已经有两门课程不及格，当插入一门不及格课程成绩时(示例中为 c2 课程)，触发器自动将 Student 表中 s5 的 status 修改为“不合格”。可以看到，通过合理地使用触发器，可以保证数据库中的数据满足现实世界中特殊的完整性约束。在本例中，这一约束为：当学生选课成绩不及格的课程数大于等于 3 时，学生的状态(status)应为“不合格”。

在实际开发中的触发器需要考虑更多的情形。例如，本例中如果考虑学生补考的情况，则需要再增加一个 After Update 触发器。如果学校允许学生通过重修销掉原先不及格的课程，则需要增加 After Delete 触发器。

2. Oracle

Oracle 对于行级触发器也同样提供了两个系统变量:old 和:new 分别存储每一条被更新

的记录的更新前旧值(:old)和更新后新值(:new)。与 MySQL 不同的是，系统变量名前面增加了冒号。表 5-6 给出了这两个系统变量在不同触发操作时值。

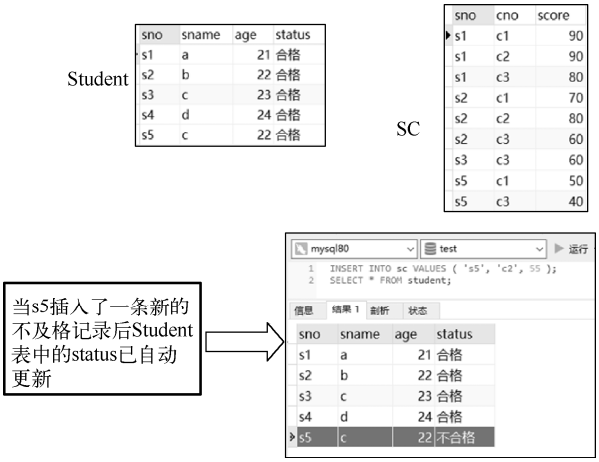


图 5-19　图 5-18 触发器的测试实例

表 5-6　Oracle 系统变量:old 和:new 在不同触发操作时的值

变量	操作		
	Insert	Delete	Update
:old	Null	原记录	原记录
:new	新记录	Null	新记录

下面的代码演示了 Oracle 中自动更新学生状态的行级触发器示例。总体的执行过程与前面 MySQL 触发器类似。

```
Create Trigger SetStatus
After Insert or Update of score on SC
For Each Row
   Declare
    a Number: = 0;
Begin
   Select count(*) Into a From SC Where s#: = :new. s# and score<60;
   If a> = 3 Then
       Update Student Set status = '不合格' Where s# = :new. s#;
   Else
      Update Student Set status = '合格' Where s# = :new. s#;
   End If
End;
```

与 MySQL 不同，Oracle 还支持语句级触发器。语句级触发器对于触发的 DML 语句执行一次，而不是像行级触发器那样对于每一条记录都执行一次。究竟使用行级触发器还是语句级触发器应当根据业务的要求而定。此外，Oracle 还允许在一个触发器中定义多个触发操作，这也与 MySQL 有差异。

下面给出一个 Oracle 中定义在多种触发操作上的语句级触发器示例。在该示例中，假设有两个基本表。

学校表：University（U#, uname, s_count）。

学生表：Student（U#, s#, sname, age）。

其中，University 表中的 s_count 存储了学生人数，它的值应和 Student 表中实际的学生人数一致。因此，当 Student 表中发生人数变化时，需自动更新 s_count 值。这一功能可以通过在 Student 上设计一个针对 Insert 和 Delete 操作的语句级触发器来实现。该触发器的简单实现如下：

```
Create Trigger TotalStudent
After Insert or Delete On Student
Declare
    a Number: = 0;
Begin
    Select count(*) Into a From Student;
    Update University Set s_count = a;
End;
```

上面的示例中都没有考虑异常处理。在实际开发中，触发器与存储过程一样也要加入异常处理语句，以保证程序的健壮性。

3．Microsoft SQL Server

Microsoft SQL Server 使用两个临时表 Inserted 和 Deleted 来存储与触发器相关的更新前后记录值。这与 MySQL、Oracle 使用系统变量的方式不同。

Insert 表和 Deleted 表是 Microsoft SQL Server 中触发器在执行时创建的临时表，可以在触发器代码中使用。两个表的结构与触发器关联的原表完全一致，其中 Inserted 表保存了新插入到表的记录，Deleted 表保存了新删除的表记录。表 5-7 给出了不同触发操作时 Inserted 和 Deleted 表中相应的内容。

表 5-7　Microsoft SQL Server 中 Inserted 和 Deleted 表在不同触发操作时的内容

临时表	操作		
	Insert	Delete	Update
Deleted	Null	原记录	原记录
Inserted	新记录	Null	新记录

下面给出了 Microsoft SQL Server 中由 Update 操作触发的触发器示例。在这个示例中，假设有下面的员工表 Emp 和部门表 Dept：

```
Emp(EmpID, Name, Salary, DepID)
Dept(DepID, Name, EmpCount, SumSalary)
```

要求修改 Dept 表的部门号 DepID 时将 Emp 表中相应的 DepID 也一并修改。这一数据完整性要求可以通过触发器来实现。基本的思路是当更新 Dept 上的 DeptID 时在触发器中同时更新 Emp 中相应的 DeptID。下面给出了触发器的定义代码：

```
Create Trigger Check_DepID On Dept
For Update                                        --Update 操作时触发
As
Declare @old Int
Declare @new Int
Select @old = (Select DepID From Deleted)   --从 Deleted 临时表中获取旧值
Select @new = (Select DepId From Inserted)  --从 Inserted 临时表中获取新值
If Update("DepID") and @@rowcount = 1       --判断 DepID 是否被修改
   Update Emp Set DepID = @new Where DepID = @old
```

Microsoft SQL Server 支持替代触发器，这与 MySQL 和 Oracle 不同。下面给出一个替代触发器的示例。继续使用上面的员工表 Emp 和部门表 Dept。现在要求删除 Dept 记录时，须首先删除 Emp 中关联的员工记录。此约束用替代触发器来实现，在触发器中先删除 Emp 中的相应记录再删除 Dept 记录，代码如下：

```
Create Trigger Del_All On Dept
InsteadOf Delete                        --替代触发器，由 Delete 操作触发
As
Declare @deptID Int
Selelct @DeptID = (Select DepID From Deleted)
--根据@DeptID 确定 Emp 表中需删除的行
Delete From Emp Where DepID = @DeptID
Delete From Dept Where DepID = @DeptID
```

5.9　本 章 小 结

本章主要介绍了过程化 SQL 的概念以及其与标准 SQL 的区别，讨论了过程化 SQL 在数据库系统中的主要用途，并重点介绍了过程化 SQL 在数据库系统中的两个主要应用——存储过程和触发器。

通过对本章的学习，读者应掌握过程化 SQL 的基本概念，了解过程化 SQL 与标准 SQL 的区别，理解游标、存储过程和触发器的概念，并能够熟练运用过程化 SQL 根据应用需求定义存储过程和触发器。

习　　题

1．过程化 SQL 与标准 SQL 之间有何区别与联系？

2．什么是游标？在过程化 SQL 中操作游标的基本过程是什么？

3．什么是存储过程？存储过程有何作用？

4．什么是触发器？触发器有何作用？

5．给定学生表 Student(sno, sname, age)和选课表 SC(sno, cno, score)，其中 SC 的 sno 字段是引用 Student 表 sno 字段的外键，如果要修改某个学生的学号，该如何实现？

第 6 章　数据库模式设计

前面在讨论 SQL 时都是假设数据库的模式结构已经存在，但是在实际应用中，数据库的模式结构是需要根据应用的需求进行分析和设计的。数据库技术最本质的目的是将现实世界中的数据表达并存储到计算机系统中，但是对于一个具体的应用系统，如何构造适合应用的数据库模式结构？这一问题的解决需要相应的理论和方法。本章以关系数据库为基础，重点讨论数据库模式设计的相关理论和方法，从而为第 7 章数据库设计奠定基础。

内容提要：本章首先介绍数据库模式设计的问题，然后着重介绍函数依赖理论以及模式设计的原则和方法，最后讨论关系模式的范式以及模式分解算法。

6.1　模式设计问题

关系数据库模式是一个关系模式的集合。关系模式的结构是根据数据库应用的需求设计得到的。由于用户需求获取方面常常会出现一些困难，而且不同的设计者在对需求的理解以及方法的使用能力上也有差别，因此针对数据库应用的数据库模式设计常常会出现不同的设计方案。不合理的数据库模式设计往往会导致一系列的问题。如何解决这些问题是数据库模式设计里需要重点研究的内容。在本节中，首先通过示例给出数据库模式设计的四类问题，然后讨论这些问题的基本解决思路。

6.1.1　四类模式设计问题

数据库模式设计中容易出现的问题包括数据冗余、更新异常、插入异常和删除异常。通常，这四类问题都是同时出现的，而且都是关联的，例如，数据冗余往往会带来更新异常。

图 6-1 给出了一个关系模式及实例的示例。在这个示例中，根据教师相关的一些需求设计一个关系模式 R(Tname, Addr, C#, Cname)，分别表示了教师名、地址、所教授的课程号和课程名称信息，并且假设课程号 C#是唯一的，一个 C#对应唯一的课程名称，一个教师只有一个地址，一个教师可教多门课程，但一门课程只有一个任课教师。因此可以得出，关系模式 R 的唯一主码是 C#(关于码的确定将在 6.2 节中讨论)。

Tname	Addr	C#	Cname
T_1	A_1	C_1	N_1
T_1	A_1	C_2	N_2
T_1	A_1	C_3	N_3
T_2	A_2	C_4	N_4
T_2	A_2	C_5	N_5
T_3	A_3	C_6	N_6

图 6-1　关系模式及实例

下面分别讨论关系数据库模式设计中易出现的四类问题。

1. 数据冗余

数据冗余是指同一个数据重复存储。在图 6-1 的示例中，教师 T_1 的地址信息 A_1 被重复存储了 3 次。但是在现实世界中，T_1 只有一个地址 A_1，因此理论上只需要存储一次即可，但由于示例中关系模式设计上的问题，在 T_1 教授了多门课程时，他的地址信息不得不冗余存储。进一步地，如果教师还有其他的一些描述信息，如性别、职称、专业等，这些信息同样需要冗余存储。

数据冗余从表面上看只是多耗费了一些存储空间。但是，冗余存储往往会导致后面的更新异常等问题。而且，在数据库系统中，性能与数据量的大小有着密切的关系。一般来说，数据量越大，同等情况下性能也会下降。因此数据冗余不仅会影响到正常数据库操作，也会影响到系统的存取性能。

2. 更新异常

仍以图 6-1 示例为例，如果教师 T_1 的地址发生了变化，则必须要同时修改图中的三个元组，因为 T_1 教了三门课，他的地址被冗余存储了三次。如果没有一起修改 T_1 三个元组中的地址信息，那么该关系中就会出现教师 T_1 有多个地址的情况。这与现实世界里真实的数据特征是冲突的。这一问题称为更新异常。

理论上，如果更新教师 T_1 的地址时能够保证 T_1 对应的所有元组的地址都一起被修改，则不会出现更新异常问题。但是，DBMS 本身并不能保证这一点。以上面的示例为例，下面的 SQL 语句就会只修改第一个元组的 Addr 而保持 T_1 的后两个元组的 Addr 不变：

```
Update R Set Addr = 'A11' Where Tname = 'T1' and C# = 'C1'
```

上面的这个 Update 语句是一个完全正确的语句，在 DBMS 中执行也没有任何问题。因此，图 6-1 的关系模式理论上会出现更新异常问题，最主要的原因在于模式设计本身存在一定的问题。

3. 插入异常

图 6-1 的关系模式中存储了教师信息、课程信息以及教师的授课信息。如果想增加一名新的教师，如 T_4，但该教师尚未安排课程(这种情况在大学里是很常见的，刚入职的年轻教师往往一开始并不教课)。此时会发现 T_4 这名教师的信息将无法插入到关系中。这是因为关系模式 R 的主码是 C#，按照实体完整性要求，任何一个元组的主码都不可为空，即只有教课的教师才能插入到图 6-1 所示的关系中。而现实世界中，新的教师并不要求必须教课。因此，示例中的关系模式不能满足应用的增加教师的需求。这一问题称为插入异常，即因为关系模式的设计问题而导致现实应用中正常的插入需求被拒绝。

4. 删除异常

删除异常是指在删除信息时将导致额外的信息被删除，从而出现信息丢失的情形。例

如，在图 6-1 的示例中，假设现在教师 T_3 不教课了，则一般的做法是将 T_3 的授课信息置空。在图中，需要将最后一个元组的 C#和 Cname 置为 NULL。这样一来，T_4 的授课信息就被删除了，满足了应用的要求。但是，这一操作是无法完成的。因为 C#是主码，实体完整性要求任何元组的 C#都不能为空。这样一来，如果想删除 T_4 的授课信息，只能将 T_4 的整个元组删除。但这么做的结果一方面是删除了 T_4 的授课信息，另一方面是删除了 T_4 这个教师的信息。这显然并不是用户需求中的一部分。

6.1.2　模式设计问题的解决

上面介绍了模式设计的四类问题，经过分析，发现主要的原因是关系模式中混合了教师信息、课程信息和选课信息。因此，最直接的解决方法是将这些信息分离开来，即通过模式分解方法将原来的关系模式分解为几个关系模式。但是，模式分解存在多种选择。下面给出三种不同的模式分解方法。

方法 1：将关系模式 R 分解为两个关系模式：R_1(Tname, Addr)和 R_2(C#, Cname)。这一分解结果将教师信息和课程信息完全分离，但是教师的授课信息丢失了。现实应用中教师与课程之间是存在授课联系的，如果在分解时丢失了这种联系，将造成信息语义的缺失，当然也不能满足后续应用的操作需求(例如，要查询教师 T_1 的授课信息将无法完成)。

方法 2：因为方法 1 的结果丢失了教师的授课信息，所以可以将 C#加到 R_1 中，从而得到下面的两个关系模式：R_1(Tname, Addr, C#)和 R_2(C#, Cname)。注意，由于一个教师可以教多门课程，而一门课程只有一个任课教师，所以 C#加到 R_1 中 R_1 的主码是 C#而不是 Tname。简单分析 R_1 和 R_2 就可以发现，模式设计的四类问题在 R_1 中依然存在。例如，T_1 教了三门课程，他的地址信息 A_1 同样在 R_1 中被冗余存储三次。

方法 3：将 Tname 加到方法 1 得到的 R_2 中，这样也可以保持教师的授课信息，得到的模式分解结果是 R_1(Tname, Addr)和 R_2(C#, Cname, Tname)。经过分析，发现这一分解结果基本解决了模式设计的问题，两个关系模式之间具有最小的冗余(R_1 的码 Tname 在 R_2 中冗余存储，但非码属性没有冗余存储)，但是，模式分解的结果也导致原来存储在一个关系中的信息被人为分割成了两个关系，因此在查询时会导致较多的连接操作。如前所述，连接操作是关系数据库中代价较高的操作，因此模式分解可能会影响到系统的性能。

从上面的讨论可以看到，不同的关系数据库模式设计具有不同的效果，或者说模式设计是有好坏之分的。但是，究竟什么样的模式是“好模式”，什么样的模式不好？如果模式不好，如何分解才能使它变成一个好的模式？模式分解的标准是什么？如何实现？这些问题都是在做模式设计和优化时必须回答的问题，也是本章要重点讨论的内容。

6.2　函 数 依 赖

关系数据库的模式设计理论是建立在函数依赖之上的，所以本节重点介绍关系数据库中的函数依赖以及相关的一些理论和方法。

6.2.1　函数依赖的概念

简单讲，函数依赖是指一个关系模式中一个属性集和另一个属性集间的多对一关系

(函数关系)。例如，选课关系模式 SC(S#, C#, Score)中，存在由属性集{S#, C#}到属性集{Score}的函数依赖。

(1)对于任意给定的 S#值和 C#值，只有一个 Score 值与其对应。

(2)可以存在多个 S#值和 C#值，它们对应的 Score 值相等。

因为属性之间的这种依赖关系与数学函数的自变量和因变量的关系类似，所以用“函数依赖”这一名词来表示这种依赖关系。

下面给出函数依赖(Functional Dependency，FD)的形式化定义。

定义 6-1(函数依赖)　设关系模式 $R(A_1, A_2, \cdots, A_n)$，简记为 $R(U)$或 R，X 和 Y 是 U 的子集。r 是 R 的任意一个实例(关系)。若 r 的任意两个元组 t_1、t_2，由 $t_1[X] = t_2[X]$可导致 $t_1[Y] = t_2[Y]$，即如果 X 相等，则 Y 也相等，称 Y 函数依赖于 X 或称为 X 函数决定 Y，记作 $X \rightarrow Y$。

上述定义说明，如果 $X \rightarrow Y$，则：

(1)R 的 X 属性集上的值可唯一决定 R 的 Y 属性集上的值。

(2)对于 R 的任意两个元组，X 上的值相等，则 Y 上的值也必相等。

函数依赖是关系模式中的一个固有组成。关系模式是一个四元组 $R(U, D, \text{Dom}, F)$，其中 F 就是属性集之间的数据依赖关系。函数依赖是属性集之间最常见的数据依赖关系，因此在关系数据库中通常以函数依赖来表示关系模式中的 F。

由于函数依赖是关系模式中的固有组成，所以函数依赖是针对整个关系模式而言的，即关系模式的所有实例都要满足其上定义的函数依赖。它代表了属性集之间必须满足的约束条件。例如，Student 关系模式中，一般存在函数依赖{S#}→{Sname}(单个属性可去掉括号，简写成 S# →Sname)，选课关系模式 SC 中，一般有{S#, C#}→{Score}。因此所有的学生元组和选课元组均要满足这些函数依赖。

判断函数依赖是否成立，唯一的办法是仔细考察应用中属性的含义。函数依赖实际上是对现实世界的断言。数据库设计者在设计时把应遵守的函数依赖通知给 DBMS，DBMS 会自动检查关系的合法性。

例如，对于关系模式 R(Tname, Addr, C#, Cname)，如果一门课程只能有一个教师，则有函数依赖 C#→Tname；若一门课程可有多个教师任教，则 C#→Tname 不成立。因此，函数依赖是与具体应用相关的。

函数依赖一般可分为两种类型：平凡函数依赖和不平凡函数依赖。

定义 6-2(平凡函数依赖)　对于函数依赖 $X \rightarrow Y$，且 $Y \subset X$，则 $X \rightarrow Y$ 称为平凡函数依赖，否则是不平凡函数依赖。

平凡函数依赖没有什么实际意义，消除平凡函数依赖是缩小函数依赖集大小的一个简单方法。缩小函数依赖集对于 DBMS 来说是有意义的，因为 DBMS 必须保证用户给出的每个函数依赖的有效性，因此函数依赖集越小，DBMS 的维护负担就越轻。

6.2.2　函数依赖集的逻辑蕴含

函数依赖集的逻辑蕴含定义如下。

定义 6-3(函数依赖集的逻辑蕴含)　设 F 是关系模式 R 的一个函数依赖集，X 和 Y 是 R 的属性子集，若从 F 的函数依赖中能推出 $X \rightarrow Y$，则称 F 逻辑蕴含 $X \rightarrow Y$，记作 $F \models X \rightarrow Y$。

基于函数依赖集的逻辑蕴含概念，可以进一步定义函数依赖集的闭包。

定义 6-4（函数依赖集的闭包）　设 F 是关系模式 R 的一个函数依赖集，被函数依赖集 F 逻辑蕴含的函数依赖的全体构成的集合称为 F 的闭包，记作 F^+。

函数依赖集的逻辑蕴含和闭包提供了一种判断两个属性集之间是否存在函数依赖关系的方法。给定关系模式 R 的一个函数依赖集 F，以及属性子集 X 和 Y，如果 F 逻辑蕴含 $X\rightarrow Y$，或者 $X\rightarrow Y$ 属于 F 的闭包，则说明 $X\rightarrow Y$ 成立，DBMS 需要维护这一函数依赖关系的有效性。

函数依赖集的逻辑蕴含一般通过 Armstrong 公理系统来推理。通过该公理系统，可以从给定的函数依赖中推出新的函数依赖。下面给出了 Armstrong 公理系统。

(1) 自反律(Reflexity)：若 $B\subseteq A$，则 $A\rightarrow B$ 成立。

(2) 增广律(Augmentation)：若 $A\rightarrow B$，则 $AC\rightarrow BC$（AC 表示 $A\cup C$）。

(3) 传递律(Transitivity)：若 $A\rightarrow B$，$B\rightarrow C$，则 $A\rightarrow C$。

(4) 自含律(Self_Determination)：$A\rightarrow A$。

(5) 分解律(Decomposition)：若 $A\rightarrow BC$，则 $A\rightarrow B$，且 $A\rightarrow C$。

(6) 合并律：若 $A\rightarrow B$，$A\rightarrow C$，则 $A\rightarrow BC$。

(7) 复合律(Composition)：若 $A\rightarrow B$，$C\rightarrow D$，则 $AC\rightarrow BD$。

【例 6-1】 给定关系模式 $R(A, B, C, D, E, F)$ 和 R 上的一个函数依赖集 $F = \{A\rightarrow BC, B\rightarrow E, CD\rightarrow EF\}$，$AD\rightarrow F$ 对于函数依赖集 F 是否成立？

可以通过 Armstrong 公理系统来求解这一问题，过程如下：

(1) $A\rightarrow BC$（已知）。

(2) $A\rightarrow C$（分解律）。

(3) $AD\rightarrow CD$（增广律）。

(4) $CD\rightarrow EF$（已知）。

(5) $AD\rightarrow EF$（传递律）。

(6) $AD\rightarrow F$（分解律）。

因此，$AD\rightarrow F$ 对于函数依赖集 F 成立。

但是，在实际的数据库系统中，通过 Armstrong 公理系统来判断 $X\rightarrow Y$ 在给定的函数依赖集 F 上是否成立很难实现，因此引入属性集的闭包概念。

定义 6-5（属性集的闭包）　给定关系模式 $R(U)$，其中 U 是 R 的属性集。设 F 是 R 上的一个函数依赖集，X 是 U 的子集，则称所有用 Armstrong 公理系统推出的函数依赖 $X\rightarrow A$ 中所有 A 的集合为属性集 X 关于 F 的闭包，记作 X^+。

定理 6-1　$X\rightarrow Y$ 能由 Armstrong 公理系统推出的充要条件是 $Y\subseteq X^+$。

借助属性集的闭包定义，可以更容易地判断 $X\rightarrow Y$ 在给定的函数依赖集 F 上是否成立。

【例 6-2】 给定关系模式 $R(A, B, C, D)$，以及 R 上的一个函数依赖集 $F = \{A\rightarrow B, B\rightarrow C, B\rightarrow D, A\rightarrow D\}$，$A\rightarrow CD$ 对于函数依赖集 F 是否成立？

通过属性集的闭包来求解这一问题。因为 $A^+ = ABCD$，所以有 $CD\subseteq A^+$，故 $A\rightarrow CD$ 对于函数依赖集 F 成立。

6.2.3　最小函数依赖集

因为函数依赖代表了实际应用环境中的语义约束，所以 DBMS 必须保证用户给出的函

数依赖的有效性。另外，用户给出的函数依赖中有可能存在冗余的情形。因此，希望能够求出与给定函数依赖集 F 等价的一个最小函数依赖集，这样 DBMS 只要实现了这个最小函数依赖集，就可以保证 F 中的所有语义约束。

最小函数依赖集的概念就是在这样的背景下提出的。要定义最小函数依赖集，首先需要了解函数依赖集的覆盖和等价概念。

定义 6-6（函数依赖集的覆盖）　设 S_1 和 S_2 是两个函数依赖集，若 $S_1^+ \subseteq S_2^+$，则称 S_2 是 S_1 的覆盖（或称 S_2 覆盖 S_1）。

定义 6-7（函数依赖集的等价）　设 S_1 和 S_2 是两个函数依赖集，若 $S_1^+ \subseteq S_2^+$，$S_2^+ \subseteq S_1^+$，则称 S_1 与 S_1 等价。

如果函数依赖集 S_2 是 S_1 的覆盖，说明 DBMS 只要实现 S_2 中的函数依赖，就自动实现了 S_1 中的函数依赖。如果 S_1 与 S_2 等价，则 DBMS 只要实现任意一个函数依赖集，就可自动实现另一个函数依赖集。

因此，对于已知关系模式 R 上的一个函数依赖集 F，如果能够找到与 F 等价的一个最小函数依赖集，则 DBMS 只要实现最小函数依赖集，就可以保证 F 的有效性。下面给出最小函数依赖集的定义。

定义 6-8（最小函数依赖集）　当且仅当函数依赖集 F 满足下面条件时，F 是最小函数依赖集。

（1）F 的每个函数依赖的右边只有一个属性。

（2）F 不可约，指 F 中的每个 $X \rightarrow Y$ 不可删除，即 $F-\{X \rightarrow Y\}$ 与 F 不等价。

（3）F 的每个函数依赖的左部不可约，即删除任何一个函数依赖左边的任何一个属性都会使 F 转变为一个不等价于原来的 F 的集合。

例如，给定学生关系模式 Student（S#, Sname, Age, Sex）：

（1）F_1 = {S#→Sname, S#→Age, S#→Sex}是 Student 的最小函数依赖集。

（2）F_2 = {S#→{S#, Sname}, S#→Age, S#→Sex}不是最小函数依赖集，因为函数依赖 S#→{S#, Sname}的右边不是单属性。

（3）F_3 = {S#→Sname, {S#, Sname}→Age, S#→Sex}不是最小函数依赖集，因为函数依赖{S#, Sname}→Age 的左部可约——属性 Sname 是可删除的，而且删除后的函数依赖集与 F_3 等价。

（4）F_4 = {S#→S#, S#→Sname, S#→Age, S#→Sex}不是最小函数依赖集，因为 S#→S# 是平凡函数依赖，是可删除的，而且删除后的函数依赖集与 F_4 等价。

最小函数依赖集的求解没有特殊的方法，就是按照它的定义对给定的函数依赖集进行约减。下面给出一个求最小函数依赖集的示例。

【例 6-3】　给定关系模式 $R(A, B, C, D)$，以及 R 上的一个函数依赖 $F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C, AC \rightarrow D\}$，求与 F 等价的最小函数依赖集。

具体的求解过程如下。

（1）运用 Armstrong 公理系统中的分解律将右边写出单属性并去除重复函数依赖。

运用分解律后得到

$$F = \{A \rightarrow B, A \rightarrow C, B \rightarrow C, A \rightarrow B, AB \rightarrow C, AC \rightarrow D\}$$

去除重复的函数依赖 $A\rightarrow B$ 后得到

$$F = \{A\rightarrow B, A\rightarrow C, B\rightarrow C, AB\rightarrow C, AC\rightarrow D\}$$

(2)消去左部冗余属性。上一步得到的 F 中左部有可能有冗余属性的只有 $AB\rightarrow C$ 和 $AC\rightarrow D$。先考虑 $AB\rightarrow C$：根据 $A\rightarrow C$，通过增广律可推出 $AB\rightarrow BC$，再通过分解律可得到 $AB\rightarrow C$，所以 $AB\rightarrow C$ 中的 B 是冗余属性。再考虑 $AC\rightarrow D$：根据 $A\rightarrow C$，$AC\rightarrow D$ 可推出 $A\rightarrow AC$(增广律)，$A\rightarrow D$(传递律)，因此 $AC\rightarrow D$ 中的 C 是冗余属性，可去除。消除左部冗余属性得到

$$F = \{A\rightarrow B, A\rightarrow C, B\rightarrow C, A\rightarrow C, A\rightarrow D\}$$

消去重复的函数依赖 $A\rightarrow C$，得到

$$F = \{A\rightarrow B, A\rightarrow C, B\rightarrow C, A\rightarrow D\}。$$

(3)消去冗余函数依赖。上一步得到的 F 中 $A\rightarrow C$ 冗余，因为根据 $A\rightarrow B$，$B\rightarrow C$ 运用传递律推出 $A\rightarrow C$。最终得到下面的函数依赖集：

$$F = \{A\rightarrow B, B\rightarrow C, A\rightarrow D\}$$

通过以上步骤，最终得到了与给定函数依赖集等价的最小函数依赖集。

6.2.4　码的形式化定义

函数依赖在数据库设计中的重要用处是确定关系模式的候选码。在前面的示例中，并没有解释为什么某个属性集是关系模式的码。关系模式的码的确定对于数据库模式设计有着十分重要的作用，因为将在 6.4 节中讨论的关系模式范式都是以码的确定为前提的。

下面给出码的形式化定义。

定义 6-9(关系模式的码)　给定关系模式 $R(U)$，以及 R 的一个函数依赖集 F。设 X 是 U 的一个子集，若有：

(1)$X\rightarrow U\in F^{+}$，则 X 是 R 的一个超码。

(2)若 X 是 R 的一个超码，且不存在 X 的真子集 Y，使得 $Y\rightarrow U$ 成立，则 X 是 R 的一个候选码。

例如，假设关系模式 R(Tname, Addr, C#, Cname)中有函数依赖集 F = {Tname→Addr, C#→Cname, C#→Tname}，则有 C#→{Tname, Addr, C#, Cname}，因此 C#是 R 的候选码。若 C#→Tname 不成立，则候选码为{Tname, C#}。

6.3　模 式 分 解

模式分解是关系数据库中解决模式设计问题的主要方法。但是，模式分解不仅意味着属性集分解，还意味着函数依赖集的分解。因此，在模式分解过程中，必须采取一定的方法保证模式分解过程的正确性。本节将主要讨论关系模式分解的相关概念和方法。

6.3.1　模式分解的概念

定义 6-10(模式分解)　设有关系模式 $R(U)$ 和 $R_1(U_1)$, $R_2(U_2)$, …, $R_k(U_k)$，其中 $U = U_1 \cup U_2 \cup \cdots \cup U_k$，设 $\rho = \{R_1, R_2, \cdots, R_k\}$，则称 ρ 为 R 的一个模式分解。

关系模式的分解隐含了两层含义。

(1) 模式分解是属性集的分解，即 R 的属性集 U 被划分为 $U_1, U_2, \cdots, U_k$。注意，在模式分解中允许这些属性子集之间含有重复属性，但是 U 中的每一个属性必须要投影到至少一个属性子集上。

(2) 模式分解是函数依赖集的分解。例如，$R(A, B, C)$, $F = \{A \rightarrow B, C \rightarrow B\}$，如果分解为 $R_1(A, B)$, $R_2(A, C)$，则丢失了函数依赖 $C \rightarrow B$。因为函数依赖代表了用户给出的语义约束，所以如果在分解过程中丢失了函数依赖，将使得语义完整性被破坏。

模式分解不能以信息丢失和破坏语义完整性为代价，这是应用对模式分解的要求。因此，模式分解过程必须遵循一定的原则和标准。

(1) 无损连接：分解之前的信息应能够通过分解后的模式无损恢复，没有信息丢失。

(2) 保持函数依赖：分解之前的所有函数依赖在分解后应全部保持下来，避免语义完整性被破坏。

一般来说，希望在任何模式分解过程中都能够同时满足这两个标准，但是在实际应用中，有的时候这两个标准无法同时满足，因此有的时候也接受只满足其中一个标准的分解，如只满足无损连接的模式分解。但是，在大多数时候，都能够在模式分解时同时满足这两条标准。

6.3.2　无损连接

无损连接要求在模式分解过程中不丢失任何信息。这是模式分解的一个基本要求。下面首先给出无损连接的动机和概念，然后介绍无损连接的测试。

1. 无损连接的动机和概念

无损连接的动机来自对模式分解后信息可恢复性的考虑。当一个关系模式 R 被分解为几个模式后，如何保证原先包含在 R 中的所有信息都可以按原样被恢复出来，即模式分解的过程应当是可逆的。这是提出并研究无损连接的主要动机。

图 6-2 给出了一个说明无损连接动机的模式分解的示例。在这个示例中，分解之前的信息(两个元组)在分解之后通过自然连接进行恢复时得到了四个元组，因此造成了信息的丢失。例如，在分解之前可以查询得到 S3 所在的城市是 Paris，而分解之后查询 S3 所在的城市将得到 Paris 和 Athens 两个结果，从而无法获知 S3 所在城市的信息。

定义 6-11(无损连接)　设 R 是关系模式，ρ 是 R 的一个模式分解：$\rho = \{R_1, R_2, \cdots, R_k\}$，$F$ 是 R 上的一个函数依赖集。若对于 R 中满足 F 的每个关系 r，都有

$$r = R_1(r) \bowtie R_2(r) \bowtie \cdots \bowtie R_k(r)$$

则称这个分解 ρ 相对于 F 是“无损连接分解”。

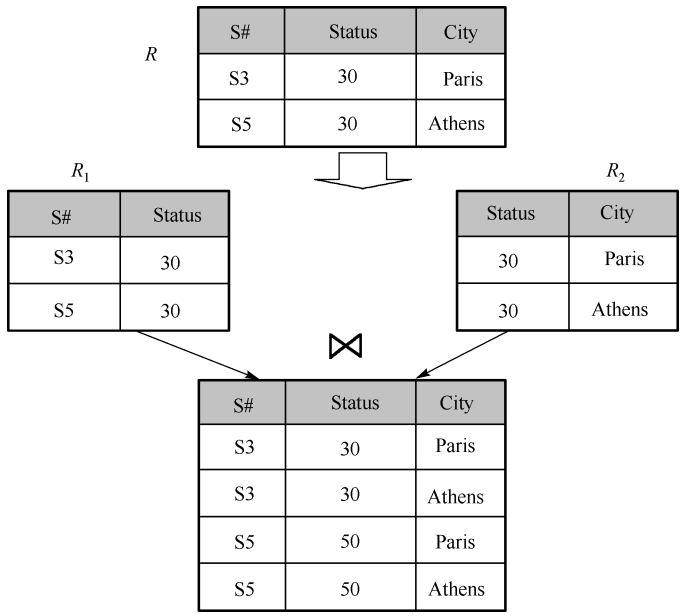


图 6-2　说明无损连接动机的模式分解示例

上述定义指明，R 的每个关系 r 都等于它在 R_i 上的投影的自然连接。模式分解的无损连接保证了 R 分解后还可以通过分解后的模式进行恢复。

一般地，记 $m_\rho(r) = \bowtie_{i=1}^{k} \pi_{R_i}(r)$，则关系模式 R 关于 F 的无损连接条件可表示为

$$r = m_\rho(r)$$

在图 6-2 的示例中，对于给定的关系 r，$m_\rho(r) \neq \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$，因此不满足无损连接。从 SQL 的角度看，这表示下面的两条 SQL 语句返回的结果是不一样的。

(1) Select * From R。

(2) Select * From R_1, R_2 Where R_1.Status = R_2.Status。

2. 无损连接的测试

无损连接的测试方法有两种。一种称为 Chase 方法，适用于一个关系模式分解成三个以上模式时的无损连接测试；另一种则只适用于一个关系模式一分为二时的测试。

Chase 方法的输入为关系模式 $R(A_1, A_2, \cdots, A_n)$、R 上的函数依赖集 F，以及 R 的一个分解 $\rho = \{R_1, R_2, \cdots, R_k\}$。

Chase 方法的执行过程如下。

(1) 构造一个 k 行 n 列的表格，每行对应一个模式 $R_i(1 \leqslant i \leqslant k)$，每列对应一个属性 $A_j(1 \leqslant j \leqslant n)$，若 A_j 在 R_i 中，则在表格的第 i 行第 j 列处填上 a_j，否则填上符号 b_{ij}。

(2) 检查 F 中的每个函数依赖，并修改表格中的元素：对于 F 中的函数依赖 $X \rightarrow Y$，若表格中有两行在 X 分量上相等，在 Y 分量上不相等，则修改 Y：若 Y 的分量中有一个 a_j，则另一个也修改为 a_j；如果没有 a_j，则用其中一个 b_{ij} 替换另一个符号（i 是所有 b 中最小的行数）。

(3)若修改后，表格中有一行全是 a，即 $a_1a_2\cdots a_n$，则ρ 相对于 F 是无损连接的分解，过程结束，否则再执行(2)，直到表格不能修改为止(即重复执行(2)后表格元素保持不变)，说明ρ 相对于 F 不是无损连接的分解。

由于 Chase 方法每次扫描 F 中的函数依赖并修改表格时至少能减少一个符号，而符号有限，因此算法最后必然终止。Chase 方法有两种终止情形：一种是表格中出现了全 a 行，说明该分解是无损连接的；另一种是表格扫描 F 后不再发生任何修改，说明该分解不是无损连接的。

下面通过一个示例来说明 Chase 方法的执行过程。

【例 6-4】 设有关系模式 $R(A, B, C, D, E)$，以及 R 的一个模式分解$\rho = \{R_1(A, D), R_2(A, B), R_3(B, E), R_4(C, D, E), R_5(A, E)\}$。$R$ 上的一个函数依赖集 $F = \{A\to C, B\to C, C\to D, DE\to C, CE\to A\}$。判断模式分解$\rho$ 相对于 F 是否是无损连接的分解。

(1)构造初始表格，如表 6-1 所示。如果某一列出现在某个关系模式中，则相应的单元格填入 a 值，否则填入 b 值。

表 6-1　Chase 方法中构造的初始表格

模式	属性 A	属性 B	属性 C	属性 D	属性 E
$R_1(A, D)$	a_1	b_{12}	b_{13}	a_4	b_{15}
$R_2(A, B)$	a_1	a_2	b_{23}	b_{24}	b_{25}
$R_3(B, E)$	b_{31}	a_2	b_{33}	b_{34}	a_5
$R_4(C, D, E)$	b_{41}	b_{42}	a_3	a_4	a_5
$R_5(A, E)$	a_1	b_{52}	b_{53}	b_{54}	a_5

(2)根据函数依赖集 F 来修改表格。由于 F 中有五个函数依赖，因此需要逐个考虑。修改的规则是：如果当前函数依赖为 $A\to C$，则表格中所有 A 列值相等的行所对应的 C 列要修改为同一个值(如果这些 A 列值相等的行的 C 列上存在一个 a 值，则所有行的 C 列值全修改为 a 值，否则修改为行序最小的 b 值)。具体修改如下。

①$A\to C$：第 1、2、5 行的 A 列值都为 a_1，对应的 C 列中没有 a_3，则将 b_{23}、b_{53} 改为 b_{13}。修改后的表格如表 6-2 所示。

表 6-2　运用 $A\to C$ 后的表格

模式	属性 A	属性 B	属性 C	属性 D	属性 E
$R_1(A, D)$	a_1	b_{12}	b_{13}	a_4	b_{15}
$R_2(A, B)$	a_1	a_2	b_{13}	b_{24}	b_{25}
$R_3(B, E)$	b_{31}	a_2	b_{33}	b_{34}	a_5
$R_4(C, D, E)$	b_{41}	b_{42}	a_3	a_4	a_5
$R_5(A, E)$	a_1	b_{52}	b_{13}	b_{54}	a_5

②$B\to C$：第 2 行和第 3 行的 B 列值相同，则将第 3 行的 C 列上的 b_{33} 改为 b_{13}。修改后的表格如表 6-3 所示。

表 6-3　运用 $B \rightarrow C$ 后的表格

模式	属性 A	属性 B	属性 C	属性 D	属性 E
$R_1(A, D)$	a_1	b_{12}	b_{13}	a_4	b_{15}
$R_2(A, B)$	a_1	a_2	b_{13}	b_{24}	B_{25}
$R_3(B, E)$	b_{31}	a_2	b_{13}	b_{34}	a_5
$R_4(C, D, E)$	b_{41}	b_{42}	a_3	a_4	a_5
$R_5(A, E)$	a_1	b_{52}	b_{13}	b_{54}	a_5

③$C \rightarrow D$：第 1、2、3、5 行的 C 列值相同，由于第 1 行的 D 列为 a_4，将第 2、3、5 行的 b_{24}、b_{34}、b_{54} 改为 a_4。修改后的表格如表 6-4 所示。

表 6-4　运用 $C \rightarrow D$ 后的表格

模式	属性 A	属性 B	属性 C	属性 D	属性 E
$R_1(A, D)$	a_1	b_{12}	b_{13}	a_4	b_{15}
$R_2(A, B)$	a_1	a_2	b_{13}	a_4	b_{25}
$R_3(B, E)$	b_{31}	a_2	b_{13}	a_4	a_5
$R_4(C, D, E)$	b_{41}	b_{42}	a_3	a_4	a_5
$R_5(A, E)$	a_1	b_{52}	b_{13}	a_4	a_5

④$DE \rightarrow C$：第 3、4、5 行的 DE 两列组合值相同，将第 3 行和第 5 行的 C 列值改为 a_3。修改后的结果如表 6-5 所示。

表 6-5　运用 $DE \rightarrow C$ 后的表格

模式	属性 A	属性 B	属性 C	属性 D	属性 E
$R_1(A, D)$	a_1	b_{12}	b_{13}	a_4	b_{15}
$R_2(A, B)$	a_1	a_2	b_{13}	a_4	b_{25}
$R_3(B, E)$	b_{31}	a_2	a_3	a_4	a_5
$R_4(C, D, E)$	b_{41}	b_{42}	a_3	a_4	a_5
$R_5(A, E)$	a_1	b_{52}	a_3	a_4	a_5

⑤$CE \rightarrow A$：第 3、4、5 行的 CE 组合值相同，将第 3 行和第 4 行的 A 列值改为 a_1。修改后的结果如表 6-6 所示。

表 6-6　运用 $CE \rightarrow A$ 后的表格

模式	属性 A	属性 B	属性 C	属性 D	属性 E
$R_1(A, D)$	a_1	b_{12}	b_{13}	a_4	b_{15}
$R_2(A, B)$	a_1	a_2	b_{13}	a_4	b_{25}
$R_3(B, E)$	a_1	a_2	a_3	a_4	a_5
$R_4(C, D, E)$	a_1	b_{42}	a_3	a_4	a_5
$R_5(A, E)$	a_1	b_{52}	a_3	a_4	a_5

到此，完成了对 F 的一轮扫描。扫描后检查表格(表 6-6)，可以发现第 3 行已经全是 a 值，即 $a_1a_2\cdots a_n$，因此算法结束。结论：ρ 相对于 F 是否为无损连接的分解。

如果 F 一轮扫描修改后没有出现全 a 行，则需要继续扫描 F 中的每个函数依赖，并逐个应用到表 6-6 上，直到最后出现了全 a 行或者表格在扫描 F 前后保持不变。

当关系模式 R 分解为两个关系模式 R_1 和 R_2 时，即 $\rho = \{R_1, R_2\}$，有一种简便的方法可以测试无损连接性。

定理 6-2　给定关系模式 R 的一个分解 $\rho = \{R_1, R_2\}$，ρ 是无损连接的分解当且仅当下面两个函数依赖中至少有一个成立：

$$(R_1 \cap R_2) \rightarrow (R_1 - R_2)$$

$$(R_1 \cap R_2) \rightarrow (R_2 - R_1)$$

式中，$R_1 \cap R_2$ 表示模式的交集，返回公共属性集；$R_2 - R_1$ 表示模式的差集，返回属于 R_2 但不属于 R_1 的属性集。

例如，设有关系模式 $R(A, B, C)$，$F = \{A \rightarrow B\}$，则下面的 ρ_1 不是无损连接的分解，而 ρ_2 是无赖连接的分解：

$$\rho_1 = \{R_1(A, B), R_2(B, C)\}$$

$$\rho_2 = \{R_1(A, B), R_2(A, C)\}$$

这是因为 ρ_2 分解满足 $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$，而 ρ_1 则对于 $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$ 和 $(R_1 \cap R_2) \rightarrow (R_2 - R_1)$ 都不满足。

6.3.3　保持函数依赖

简单讲，保持函数依赖要求关系模式 R 的每一个不平凡函数依赖在分解过程中都保留下来。

定义 6-12（保持函数依赖）　给定关系模式 $R(U)$，以及 R 的一个函数依赖集 F，设 Z 是 U 的子集，F 在 Z 上的投影用 $\pi_Z(F)$ 表示，定义为 $\pi_Z(F) = \{X \rightarrow Y \mid X \rightarrow Y \in F^+ \wedge XY \subseteq Z\}$。对于 $R(U)$ 上的一个分解 $\rho = \{R_1, R_2, \cdots, R_k\}$，若满足下面条件，则称分解 ρ 保持函数依赖集 F：

$$\left(\bigcup_{i=1}^{k} \pi_{R_i}(F)\right)^+ = F^+$$

【例 6-5】　给定关系模式 R(City, Street, Zip)，以及 R 上的一个函数依赖集 $F = \{$(City, Street) →Zip, Zip→City$\}$，判断分解 $\rho = \{R_1(\text{Street, Zip}), R_2(\text{City, Zip})\}$ 是否无损连接，以及是否保持函数依赖。

(1) 首先判断 ρ 是否无损连接。

由于这个示例中 R 是一分为二的，因此可以用前面介绍的简单方法来判断 ρ 是否无损连接：

因为 $R_1 \cap R_2 = \{\text{Zip}\}$，$R_2 - R_1 = \{\text{City}\}$，而 Zip→City 已知成立，所以 ρ 是无损连接的分解。

(2) 接着判断 ρ 是否保持函数依赖。

$$\pi_{R_1}(F) = \{\text{按自反律推出的平凡 FD}\}$$

$$\pi_{R_2}(F) = \{\text{Zip→City，以及按自反律推出的平凡 FD}\}$$

$$\pi_{R_1}(F) \cup \pi_{R2}(F) = \{\text{Zip} \rightarrow \text{City}\}^+ \neq F^+ \cup \pi_{R_2}(F) = \{\text{Zip} \rightarrow \text{City}\}^+ \neq F^+$$

因此，模式分解ρ不保持函数依赖。

如果一个模式分解不保持函数依赖，将带来什么问题呢？因为函数依赖代表了应用环境中的语义约束，所以不保持函数依赖从理论上将导致某些特定的语义约束被丢失，从而使得数据库中的数据与现实世界中的真实特征相冲突，破坏数据库的语义完整性。

以例 6-4 中的模式分解为例，这个分解在示例中已经证明是无损连接的。现在在 R_1 中插入元组（'a', '100081'）和（'a', '100082'）。由于 R_1 中没有不平凡的函数依赖，所以这两个元组的插入不违背任何语义约束，可以顺利完成。接下来，在 R_2 中插入元组（'Beijing', '100081'）和（'Beijing', '100082'）。R_2 中的不平凡函数依赖只有一个：Zip→City，而这两个元组的插入都满足这一函数依赖，所以也可以顺利完成。然后，执行一个查询，查询 R 中的 City、Street 和 Zip 信息，即执行 $R_1 \bowtie R_2$，得到的结果如图 6-3 所示。在图中可以看到，Beijing 的同一个街道 *a* 出现了两个邮政编码，而现实应用中一个城市的一个街道只有一个邮政编码（(City, Street)→Zip）。出现这一情况的原因是函数依赖(City, Street)→Zip 在模式分解过程中被丢失了，从而导致语义完整性被破坏。

City	Street	Zip
Beijing	*a*	100081
Beijing	*a*	100082

图 6-3　R_1(Street, Zip) ⋈ R_2(City, Zip)的结果

6.4　关系模式的范式

由于一个关系模式的属性集通常包含较多的属性，因此理论上一个关系模式可以多次分解下去。这里的问题是“分解过程应该持续到何时才能结束?”这个问题的回答需要对模式分解的结果有一个评价标准。如果分解结果满足了定义的标准，那么分解就可以终止了；如果不满足，则需要继续分解下去。

关系模式的范式主要用来解决关系模式的评价问题。通过定义不同的范式要求，可以将关系模式划分到相应的范式中。因为不同的范式代表了对关系模式的不同要求，所以范式可以看成对关系模式的一个衡量标准。模式分解的目的实际上就是想把低级别的范式分解为高级别的范式。

6.4.1　范式与规范化的概念

范式(Normal Form)是指满足特定要求的关系模式。在关系数据库理论中，研究者定义了不同的范式来描述关系模式的不同特性，从而可以将范式看成衡量关系模式好坏程度的一个标准。目前，范式的级别从低到高分别定义了第一范式(1NF)、第二范式(2NF)、第三范式(3NF)、BCNF 范式(BCNF)以及第四范式(4NF)和第五范式(5NF)。如果一个关系模式 R 满足第 x 范式(xNF)，可以记成 $R \in x\text{NF}$。因此，范式可以看成模式特征的统一定义，也可以看成一个满足某种特定要求的关系模式的集合。

在关系模式的各个范式中，第四范式和第五范式一般仅有理论意义，在实际的数据库设计中一般不会涉及，因此在本书中不准备对其进行详细讨论，重点是 1NF、2NF、3NF 和 BCNF 范式。

不同级别的范式之间满足包含关系，如图 6-4 所示。也就是说，如果一个关系模式满足高级别的范式，它必然满足低级别范式的要求，例如，3NF 的关系模式肯定满足 2NF 的要求。

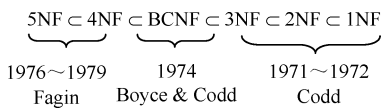


图 6-4　范式之间的包含关系

通常，低级别范式出现模式设计四类问题的可能性要大于高级别范式，所以在数据库设计中希望能够尽量将关系模式设计成满足高级别范式的要求。通过模式分解将低级别范式的关系模式转换为高级别范式的关系模式的过程，称为关系模式的规范化(Normalization)。规范化实际上是一个模式分解和优化的过程。通过关系模式的规范化，可以将“不好的”关系模式分解到“好的”关系模式，从而避免出现模式设计的四类问题，提高数据库模式结构的设计质量。

6.4.2　函数依赖图

为了便于理解关系模式的规范化过程，先引入“函数依赖图”作为工具来描述关系模式的函数依赖。函数依赖图将关系模式中的函数依赖集以有向图的方式进行表示，从而能够更清晰地理解规范化的过程。

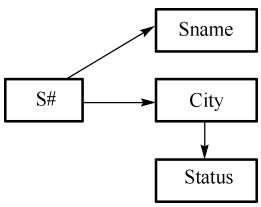


图 6-5　函数依赖图示例

在函数依赖图中，结点是函数依赖中的属性集，用矩形框表示；有向边代表属性集之间的函数依赖关系。例如，假设关系模式 R 的一个函数依赖集为 F = {S#→Sname, S#→City, City)→Status}，则相应的函数依赖图如图 6-5 所示。

6.4.3　1NF

定义 6-13(第一范式 1NF)　对于关系模式 R 的任一实例，若其元组的每一个属性值都只含有一个值，则 R∈1NF。

1NF 是关系模式的基本要求，也就是说，只要是关系模式，就必须满足 1NF。从这个意义上看，1NF 只是保证了理论的完整性。

如果不满足第一范式，则关系的属性值中允许出现值集，俗称“表中有表”。非第一范式的模式不能用关系数据模型来表达，其主要问题是更新困难，会带来更新二义性。如图 6-6 所示，如果是非 1NF 的关系，则属性值中允许出现{数据库, 编译原理}这样的集合。如果现在 S001 也选修了“数据库”，则在更新时将面临两种选择：一种是将第一个元组的课程改成{数据库, 编译原理}；另一种是将第二个元组的学号改成{S001, S002}。在现有的二进制计算方式下，这种具有二义性的操作是 DBMS 无法执行的。

学号	课程
S001	C 语言
S002	{数据库, C 语言}

图 6-6　非 1NF 将导致更新二义性

6.4.4　2NF

定义 6-14(第二范式 2NF)　当且仅当 R∈1NF，且 R 的每一个非主属性都完全函数依赖于主码时，R∈2NF。

2NF 的定义涉及完全函数依赖的概念。

定义 6-15（完全函数依赖）　对于函数依赖 $W \rightarrow A$，若不存在 $X \subset W$，并且 $X \rightarrow A$ 成立，则称 $W \rightarrow A$ 为完全函数依赖，否则为局部函数依赖。

回忆一下，在关系模式中，主属性是指包含在候选码中的属性，而非主属性是指不包含在任何候选码中的属性。

2NF 要求每个非主属性都完全函数依赖于主码。举个例子，设有关系模式 $R(A, B, C, D, E)$，其中$\{A, B\}$为主码，则 A 和 B 是主属性，C、D、E 为非主属性，因此要求 C、D、E 都完全函数依赖 AB，即满足 $AB \rightarrow C$，$AB \rightarrow D$，$AB \rightarrow E$，并同时要求 $A \rightarrow C$，$B \rightarrow C$，$A \rightarrow D$，$B \rightarrow D$，$A \rightarrow E$，$B \rightarrow E$ 中任何一个均不成立（任何一个成立则意味着某个非主属性局部函数依赖于主码）。

【例 6-6】 设关系模式 R 表示了供应关系：R(S#, P#, City, Status, Price, QTY)，R 上的函数依赖集 F = {S#→City, S#→Status, P#→Price, City→Status, {S#, P#} →QTY}，问 R 是否满足 2NF？

要回答这一问题，首先要确定 R 的主码是什么。根据 F，可以知道 R 的唯一候选码为{S#, P#}，因此主码为{S#, P#}。下一步确定非主属性为 City、Status、Price 和 QTY。从 F 中可以看到，除了 QTY 之外，其余非主属性局部函数依赖于主码，所以 R 不满足 2NF。

不满足 2NF 的关系模式会带来什么问题？以例 6-6 中的关系模式 R(S#, P#, City, Status, Price, QTY)为例，模式设计的四类问题都出现 R 中。

(1)数据冗余：同一供应商的 City 被重复存储。

(2)插入异常：没有供应零件的供应商无法插入。这是因为 P#是主码的一部分，实体完整性要求任何元组的 P#不能为空。

(3)删除异常：删除供应商的供货信息有可能会同时删除供应商自身信息。如果某个供应商不再供应零件了，一般来说只将 P#、Price 和 QTY 置空即可，但由于 P#是主码的一部分，因此不能置空，所以只能删除整个元组，这将导致供应商自身信息也被删除。

(4)更新异常：供应商的 City 修改时必须修改多个元组。这是因为一个供应商可以供应多个零件，所以冗余存储的供应商信息(如 City、Status 等)都需要同时修改。如果只修改了其中的部分元组，就会出现更新异常的问题。

解决这些问题的方法就是规范化。以例 6-6 中的关系模式为例，可以将该关系模式的函数依赖集用函数依赖图的方式表示，并去掉其中的局部函数依赖，即把那些由主码部分属性所决定的非主属性连同相应的主属性分解成新的关系模式。图 6-7 表示了这一过程。

对于图 6-7 所示的模式分解，需要用前面讨论的测试方法来测试是否保证了无损连接和保持函数依赖。在后面的内容中，将介绍一种不需要每次模式分解都去测试无损连接和保持函数依赖的分解算法，因此，这里主要是掌握 2NF 的概念。

6.4.5　3NF

定义 6-16（第三范式 3NF）　当且仅当 R 属于 2NF，且 R 的每一个非主属性都不传递依赖于主码时，$R \in$3NF。

3NF 定义用到了传递依赖的概念。

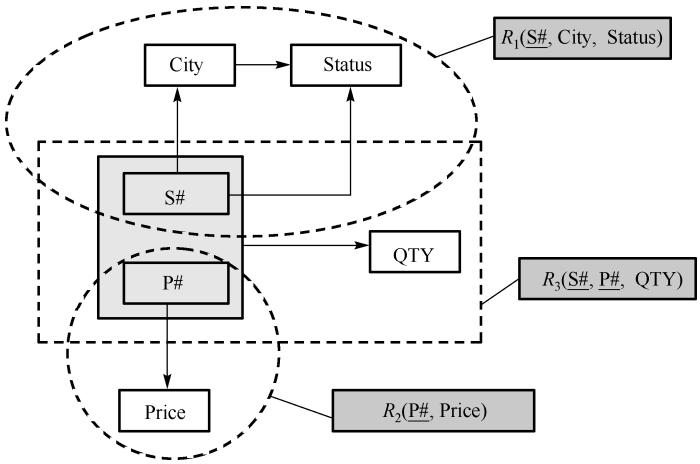


图 6-7　消除局部函数依赖规范化到 2NF

定义 6-17(传递依赖)　若 $Y \rightarrow X$，$X \rightarrow A$，并且 $X \rightarrow Y$ 不成立，A 不是 X 的子集，则称 A 传递依赖于 Y。

以图 6-7 分解得到的 2NF 模式 R_1(S#, City, Status)为例，在 R_1 中存在着函数依赖 S#→City 和 City→Status，因此 Status 和 S#之间存在着传递依赖，所以 R_1 不满足 3NF。

不满足 3NF 同样会带来模式设计的四类问题，以 R_1(S#, City, Status)为例。

(1)数据冗余：同一城市的 Status 冗余存储。

(2)插入异常：不能插入一个具有 Status 但没有供应商的 City，例如，Rome 的 Status 为 50，但除非有一个供应商住在 Rome，否则无法插入。

(3)删除异常：删除供应商时会同时删除与该城市相关的 Status 信息。

(4)更新异常：一个城市中会有多个供应商，因此 Status 更新时要更新多个元组。

可以借助函数依赖图将 2NF 分解到 3NF，图 6-8 给出了说明。分解的主要目的是消除传递依赖。分解后的 R_4 和 R_5 均满足 3NF。

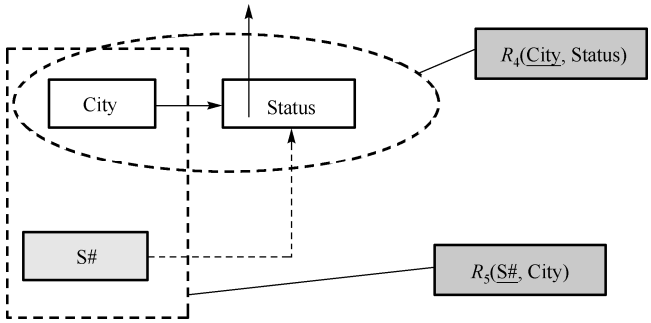


图 6-8　消除传递依赖规范化到 3NF

6.4.6　BCNF

2NF 和 3NF 都只考虑了非主属性与主码之间的函数依赖关系，没有考虑主属性和主码之间的函数依赖关系。当关系模式只有一个候选码时，不会出现主属性和主码之间的不平

凡函数依赖(因为主属性包含在主码当中)。但是，如果一个关系模式中存在着多个候选码，则即使是 3NF，也会出现模式设计的一系列问题。

如图 6-9 所示，假设现有表示供应关系的模式 *R*(S#, Sname, P#, QTY)，并且已知 S# 和 Sname 都是唯一的，因此 *R* 中存在着两个候选码：{S#, P#}和{Sname, P#}。经过分析可以发现，*R* 满足 3NF。因为无论选择{S#, P#}还是{Sname, P#}作为主码，*R* 中都只有一个非主属性 QTY，而 QTY 是完全函数依赖于{S#, P#}和{Sname, P#}的，即{S#, P#}→QTY，{Sname, P#}→QTY 都成立。

S#	Sname	P#	QTY
s1	Intel	p1	300
s1	Intel	p2	200
s1	Intel	P3	400
s2	Acer	p1	200

图 6-9　满足 3NF 但不满足 BCNF 的关系模式示例

但是，很容易发现图 6-9 所示的关系存在一系列的模式设计问题。

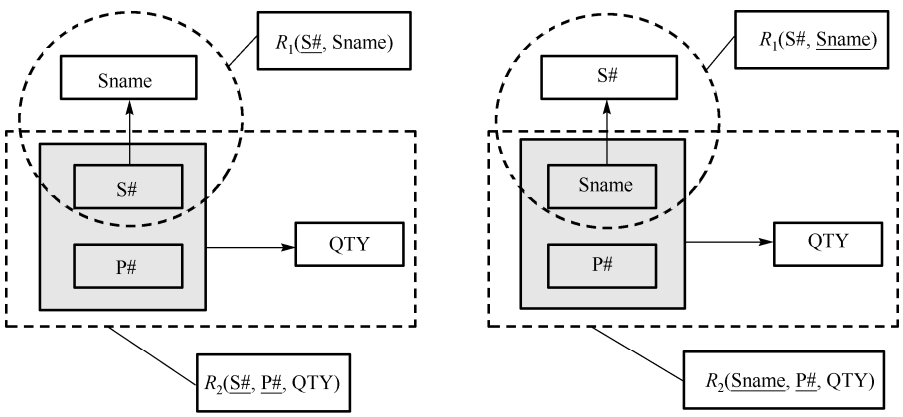
(1) 数据冗余：s1 的名字 Intel 重复存储。

(2) 更新异常：修改 s1 的名字时必须修改多个元组。

(3) 删除异常：若 s2 现在不提供任何零件，则须删除 s2 的元组，但同时删除了 s2 的名字。

(4) 插入异常：没有提供零件的供应商无法插入。

按照一般性的方法，还是通过模式分解的方法来解决图 6-9 模式的问题。图 6-10 分别给出了主码为{S#, P#}和{Sname, P#}时的分解结果。



(a) 主码为{S#, P#}　　(b) 主码为{Sname, P#}

图 6-10　多候选码时 3NF 分解到 BCNF

经过分析，发现分解后原来存在的那些模式设计问题消除了。但问题是，在原有的 1NF、2NF 和 3NF 理论体系下，现在分解后的关系模式仍然还只是 3NF。很显然，分解后的模式要优于 3NF。因此，这里出现了一个理论上的缺口，BCNF 就是在这样的背景下提出的。

BCNF 范式(Boyce/Codd 范式)是由 Ray Boyce 和 Edgar F. Codd 提出的，BCNF 就以他们名字的首字母命名。BCNF 扩充了 3NF，可以处理 *R* 有多个候选码的情形。与 3NF 相比，BCNF 不仅考虑了非主属性和主码之间的函数依赖，还考虑了主属性和主码之间的函数依赖。下面给出 BCNF 的定义。

定义 6-18　如果关系模式 R 的所有不平凡的、完全的函数依赖的决定因素(左边的属性集)都是候选码，则 $R\in$BCNF。

BCNF 的定义也隐含了 2NF 和 3NF 中的要求，因此满足 BCNF 的模式肯定满足 2NF 和 3NF。在 BCNF 模式的函数依赖图中，所有有向边都是从候选码中引出的，所有不平凡函数依赖的左边属性集都是候选码。

下面讨论一个 3NF 分解到 BCNF 的示例。设有教学关系模式 $R(S, J, T)$，S、J、T 分别表示学号、课程号和教师姓名。已知每个教师只教一门课程，每门课程有若干任课教师，学生选定一门课程就对应一个固定的教师。因此知道 R 满足不平凡函数依赖 $T\rightarrow J$ 和$\{S, J\}\rightarrow T$。很容易发现，R 中唯一的候选码是$\{S, J\}$，并且 R 满足 3NF。这是因为 R 中只有一个非主属性 T，而 T 和主码$\{S, J\}$之间不存在局部函数依赖与传递函数依赖。但 R 不满足 BCNF，因为函数依赖 $T\rightarrow J$ 违背了 BCNF 的定义。

图 6-11 给出了该关系模式分解到 BCNF 的结果。需要注意的是，在分解过程中，函数依赖$\{S, J\}\rightarrow T$ 无法保持下来。这个示例说明，在将 3NF 分解到 BCNF 的过程中，有时候是无法保持函数依赖的。

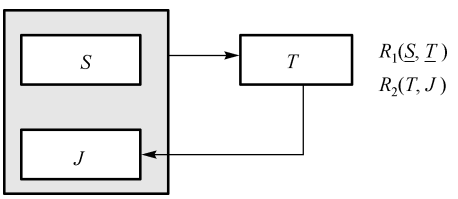


图 6-11　3NF 分解到 BCNF

6.5　模式分解的算法

在前面的讨论中，一般要求对模式分解结果进行无损连接和保持函数依赖的测试，以保证模式分解过程的正确性。本节将介绍几个规范化算法，它们可以使模式分解时不必每次都要人工验证是否无损连接和保持函数依赖。这些算法包括：

(1)无损连接并且保持函数依赖地分解到 3NF 的算法；

(2)无损连接地分解到 BCNF 的算法。

掌握了这两个算法之后，在实际应用中可以直接将关系模式一步分解到 3NF，并且保证无损连接和保持函数依赖。对于 BCNF，由于不一定能在分解过程保持函数依赖，所以目前的算法只能保证分解的无损连接。

6.5.1　无损连接并且保持函数依赖地分解到 3NF 的算法

一个关系模式总是可以分解到 3NF，并且可以保证分解过程是无损连接并且保持函数依赖的。在介绍这一算法之前，首先介绍将一个关系模式保持函数依赖地分解到 3NF 的算法。这一算法是无损连接并且保持函数依赖地分解到 3NF 的算法的基础。

算法 6-1　保持函数依赖地分解到 3NF。

输入：$R<U, F>$，其中 U 是属性集，F 是函数依赖集。

输出：R 的一个模式分解ρ，ρ 中的所有模式满足 3NF，且ρ 是保持函数依赖的分解。

过程如下。

(1)求出 $R<U, F>$的最小函数依赖集(仍记为 F)。

(2)把所有不在 F 中出现的属性组成一个关系模式 R'，并在 U 中去掉这些属性(剩余属性仍记为 U)。

(3)若 F 中存在 $X\rightarrow A$，且 $XA=U$，则输出 $R(U)$和 R'，算法结束，否则对 F 按相同的左部进行分组，将所有 $X\rightarrow A_1, X\rightarrow A_2,\cdots, X\rightarrow A_k$ 形式的函数依赖分为一组，并将每组涉及的所有属性作为一个关系模式输出。若某个关系模式 R_i 的属性集是另一个关系模式的属性集的子集，则在结果中去掉 R_i。设最后得到关系模式 $R_1, R_2,\cdots, R_k$，则$\rho=\{R_1, R_2,\cdots, R_k, R'\}$是一个保持函数依赖的分解，并且满足 3NF。

例如，对于关系模式 $R(ABCDEF)$，设 $F=\{A\rightarrow B, AC\rightarrow E\}$，将 R 保持函数依赖地分解到 3NF 的过程如下。

(1)求最小函数依赖集 $F=\{A\rightarrow B, AC\rightarrow E\}$。

(2)求出 $R'(DF)$。

(3)按左部分组，得到 $R_1(AB)$, $R_2(ACE)$。

(4)$\rho=\{R'(DF), R_1(AB), R_2(ACE)\}$。

ρ 就是 R 保持函数依赖分解到 3NF 的结果。

算法 6-2 给出了无损连接并且保持函数依赖地分解到 3NF 的算法。

【例 6-7】 给定关系模式 R(S#, SN, P, C, S, Z)，和 $F=\{$S#→SN, S#→P, S#→C, S#→S, S#→Z, $\{P, C, S\}\rightarrow Z$, $Z\rightarrow P$, $Z\rightarrow C\}$，将 R 无损连接并且保持函数依赖地分解到 3NF。

(1)求出最小 FD 集：F = {S#→SN, S#→P, S#→C, S#→S, $\{P, C, S\}\rightarrow Z$, $Z\rightarrow P$, $Z\rightarrow C\}$。这一步去掉了冗余的函数依赖 S# →Z。

(2)$\rho_1=\{R_1(\underline{\text{S\#}}, \text{SN}, P, C, S), R_2(\underline{P}, \underline{C}, \underline{S}, Z), R_3(\underline{Z}, P, C)\}$。

(3)R_3 是 R_2 的子集，所以去掉 R_3，$\rho_1=\{R_1(\underline{\text{S\#}}, \text{SN}, P, C, S), R_2(\underline{P}, \underline{C}, \underline{S}, Z)\}$。

(4)R 的主码为 S#，于是$\rho=\rho_1\cup R''(\text{S\#})=\{R_1(\underline{\text{S\#}}, \text{SN}, P, C, S), R_2(\underline{P}, \underline{C}, \underline{S}, Z), R''(\text{S\#})\}$。

(5)因为 S#是 R_1 的子集，所以从ρ 中去掉 R''(S#)。

(6)$\rho=\{R_1(\underline{\text{S\#}}, \text{SN}, P, C, S), R_2(\underline{P}, \underline{C}, \underline{S}, Z)\}$即最终结果。

算法 6-2　无损连接并且保持函数依赖地分解到 3NF。

输入：$R<U, F>$，其中 U 是属性集，F 是函数依赖集。

输出：R 的一个模式分解ρ，ρ 中的所有模式满足 3NF，并且ρ 是无损连接且保持函数依赖的分解。

过程如下。

(1)先用算法 6-1 求出 R 的保持函数依赖的 3NF 分解，设为$\rho_1=\{R_1, R_2,\cdots, R_k\}$。

(2)设 X 是 R 的码，则$\rho=\rho_1\bigcup R''(X)$。

(3)若 X 包含在$\rho_1=\{R_1, R_2,\cdots, R_k\}$的某个关系模式中，则在$\rho$ 中去掉 $R''(X)$。

(4)得到的ρ 就是最终结果，分解过程满足无损连接和保持函数依赖，并且所有模式满足 3NF。

在例 6-7 中，求解过程的第(4)步将 R 的主码单独形成一个关系模式。如果主码是单个属性，按照模式分解的定义，该主码必定包含在ρ_1 中的某个关系模式中，因此最终结果实际上与算法 6-1 的结果是相同的。但是，如果 R 的主码不是单个属性，则算法 6-2 的结果不同于算法 6-1。下面给出一个主码不是单属性的示例。

【例 6-8】 给定关系模式 R(S#, SN, P, C, S, Z)，F = {S#→SN, S#→P, S#→C, Z→S, Z→C}，将 R 无损连接并且保持函数依赖地分解到 3NF。

(1) 求出最小函数依赖集：F = {S#→SN, S#→P, S#→C, Z→S, Z→C}。

(2) ρ_1 = {R_1(S#, SN, P, C), R_2(Z, S, C)}。

(3) R 的主码为{S#, Z}，于是ρ = ρ_1 ∪ R''(S#, Z) = {R_1(S#, SN, P, C), R_2(Z, S, C), R''(S#, Z)}。

(4) ρ = {R_1(S#, SN, P, C), R_2(Z, S, C), R''(S#, Z)}即最终结果，分解过程满足无损连接和保持函数依赖。

6.5.2　无损连接地分解到 BCNF 的算法

算法 6-3 给出了无损连接地分解到 BCNF 的具体过程。

算法 6-3　无损连接地分解到 BCNF。

输入： R<U, F>，其中 U 是属性集，F 是函数依赖集。

输出： R 的一个模式分解ρ，ρ 中的所有模式满足 BCNF，并且ρ 是无损连接的分解。

过程如下。

(1) ρ = {R}。

(2) 检查ρ 中各关系模式是否都属于 BCNF，若是，则算法终止。

(3) 设ρ 中 S(Us)非 BCNF 关系模式，则必定存在 X→A，其中 X 不是 S 的候选码：

①将 S 分解为 S_1(XA)和 S_2(Us–A)，此分解是无损连接的，因为({XA} ⋂ {Us–A} = X)→(A = {XA}–{Us–A})；

②ρ = {ρ– S} ⋃ {S_1, S_2}，即用 S_1 和 S_2 替换ρ 中的 S；

③转到第(2)步。

(4) 由于 U 的属性有限，因此有限次循环后算法终止。

【例 6-9】 给定关系模式 R(S#, C#, G, TN, D), F = {{S#, C#}→G, C#→TN, TN→D}，将 R 无损连接地分解到 BCNF。

(1) ρ = {R}。

(2) TN→D 不满足 BCNF 定义，分解 R：ρ = {R_1(S#, C#, G, TN), R_2(TN, D)}。

(3) R_1 中 C#→TN 不满足 BCNF，继续分解 R_1 为 R_3 和 R_4：

$$\rho = \{R_3(\text{S\#, C\#}, G), R_4(\text{C\#}, \text{TN}), R_2(\text{TN}, D)\}$$

(4) ρ 中各模式均满足 BCNF，即最终结果。

在例 6-9 中，函数 C#→TN 和 TN→D 均不满足 BCNF 定义。如果选择 C#→TN 而不是 TN→D 进行分解，将得到ρ = {R_1(S#, C#, G, D), R_2(C#, TN)}。此处，注意 R_1 的主码为{S#, C#, D}，而不是{S#, C#}。因此，R_1 中唯一的不平凡函数依赖{S#, C#}→G 不满足 BCNF，需要继续分解，得到ρ = {R_3(S#, C#, G), R_4(S#, C#, D), R_2(C#, TN)}。由于 R_4 中不存在不平凡的函数依赖，所以它的主码就是整个属性集。至此，ρ 中各模式均满足 BCNF，分解结束。

通过上面的示例可以得出以下两点结论。

(1) 当选择 C#→TN 开始将关系模式分解到 BCNF 时，最终得到的结果与选择 TN→D

开始的分解结果不同。因此，一个关系模式无损连接地分解到 BCNF 有可能会得到不同的结果。

(2) 当选择 C#→TN 开始分解时，最终得到的结果丢失了函数依赖 TN→D，而选择 TN→D 开始的分解结果则保持了所有的函数依赖。因此，将一个关系模式无损连接地分解到 BCNF 的算法不一定能够保持函数(但也有可能保持了全部的函数依赖)。

6.6 本 章 小 结

本章主要介绍了关系数据库模式设计的相关理论，讨论了模式设计过程中容易出现的四类问题，并重点介绍了函数依赖的概念以及其在模式分解中的作用，最后着重讨论了关系数据库规范化理论并给出了将关系模式规范化(即模式分解)到 3NF 或者 BCNF 的算法。

关系模式规范化的过程可以总结为以下几点。

(1) 对 1NF 模式投影，消除非主属性对主码的局部函数依赖，可将关系模式规范化到 2NF。

(2) 对 2NF 模式投影，消除非主属性对主码的传递函数依赖，可将关系模式规范化到 3NF。

(3) 对 3NF 模式投影，消除左边不是候选码的函数依赖，可将关系模式规范化到 BCNF。

(4) 若要求保持函数依赖，则总可以分解到满足 3NF，但不一定满足 BCNF，即 BCNF 可以达到无损连接，但不一定保持函数依赖。

(5) 若要求保持函数依赖和无损连接，则总可以达到 3NF，但不一定满足 BCNF。

通过对本章的学习，读者应掌握函数依赖以及范式的基本概念，了解规范化的基本过程，理解无损连接和保持函数依赖的概念与动机，并能够熟练运用规范化算法进行模式设计和优化。

习　　题

1．数据库模式设计会出现哪些问题？请举例说明。

2．已知 $R(A, B, C, D)$，$F = \{A\rightarrow C, B\rightarrow C, A\rightarrow D, D\rightarrow C\}$，$R$ 的一个模式分解为 $\rho = \{R_1(AB), R_2(\mathrm{AC}), R_3(AD)\}$。

(1) 求 F 在各分解模式上的投影。

(2) 该分解是否无损连接。

(3) 该分解是否保持函数依赖。

3．已知关系模式 $R = ABCDE$，$F = \{AB\rightarrow C, C\rightarrow D, D\rightarrow E\}$，$R$ 的一个模式分解为 $\rho = \{R_1(ABC), R_2(CD), R_3(DE)\}$。该分解是否是无损连接的分解？

4．已知 $R(A, B, C, D, E, F)$，$F = \{E\rightarrow D, C\rightarrow B, CE\rightarrow F, B\rightarrow A\}$。

(1) R 最高属于第几范式。

(2) 请将 R 无损连接并保持函数依赖地分解到 3NF。

(3) 请将 R 无损连接地分解到 BCNF。

第 7 章　数据库设计

在数据库系统中，数据库设计决定了最终建立的数据库结构的质量，其对前端数据库应用程序的功能实现和性能满足也有重要的影响。数据库设计的主要任务是设计数据库的模式结构，其中的理论内容已经在第 6 章有过介绍。数据库设计理论回答了“为什么这么做”的问题，而本章将着重从工程化的角度讨论数据库设计的整个过程，回答“怎么做”的问题，目的是使大家能够掌握使用规范化的方法为不同的数据库应用进行数据库设计的流程和方法。

内容提要：本章首先介绍数据库设计的概念、方法和过程，然后着重介绍数据库设计的各个过程，重点是每一过程中需要完成的任务、采用的方法和需要注意的一些问题。

7.1　数据库设计概述

数据库设计既涉及理论问题，也强调工程化。数据库设计的工程化要求是随着软件工程的发展而逐步提出的。在软件工程研究中，强调用工程化方法来开发应用软件，以保证软件开发的质量。数据库设计也借鉴了这一思路，建立了工程化的过程模型来进行数据库设计。本书讨论的数据库设计也遵循这样的思路。

数据库设计是指对于给定的应用环境，构造最优的数据库模式，并利用现成的 DBMS，设计数据库物理存储结构，进而建立数据库及其应用系统，使之能够有效地存储数据，满足各种用户的需求。

数据库设计的概念包含了三层含义。

(1)数据库设计是面向特定应用的。这是因为不同的应用有不同的数据管理需求，因此设计者需要针对每个应用的特殊需求进行数据库设计。

(2)数据库设计包含了逻辑设计和物理设计的过程。逻辑设计是指构造最优的数据库逻辑结构(模式结构)，物理设计是指设计数据库的物理存储结构。

(3)数据库设计不仅包括数据库结构设计，还包含初始数据库的建立以及其他设计工作，如恢复设计、安全设计、应用系统设计等。

(4)数据库设计的最终目的是保证数据存取上的功能和性能要求。

7.1.1　数据库设计的方法

数据库设计是一种方法而不是工程技术，缺乏科学的理论支持，很难保证质量。数据库设计方法的提出与软件工程技术的发展密不可分。早期的软件开发强调编程技能，当问题规模比较小的时候，这种开发方法也能胜任。但是，随着计算机软硬件技术的快速发展和信息技术应用范围的不断扩大，软件系统的需求越来越复杂，功能要求越来越高，这时候整个软件系统的开发并不是仅仅靠编程就能完成的。这就好比建造房子，如果建一个茅草屋，可能一个瓦工就可以完成，而且基本上可以直接上来就从地基开始建设。但是，如

果建一座高楼大厦，仅靠瓦工的技术很难完成，必须引入细致、规范的设计方法，从理论上保证大厦的质量。现代的软件开发类似于建造高楼大厦，因此设计方法在其中具有非常重要的地位。这是软件工程思想和方法学产生的背景。数据库本身是数据库应用系统的一部分，因此数据库设计也是软件设计中的一个组成部分。随着数据规模和数据应用的不断发展，数据库设计也同样面临着问题规模日益扩大的问题。因此，科学的、规范的数据库设计方法，是决定数据库功能和性能，进而决定整个数据库应用系统质量的重要因素。

数据库设计方法借鉴了软件工程的思想，强调用工程化的方法进行数据库设计。在传统的结构化软件工程中，要求软件设计和开发的每一个过程都要遵循严格的顺序，前一个阶段结束并且验收后才能开展下一个阶段的工作。例如，需求分析没有结束时不能开展概要设计。一般地，把运用软件工程的思想和方法进行数据库设计的方法称为规范化的数据库设计方法。由于软件工程把整个软件开发过程分成了需求分析、概要设计、详细设计、编码、测试、实施等阶段，因此，规范化的数据库设计方法也把整个数据库设计过程分成了有序的几个阶段。目前已经提出了许多规范化的数据库设计方法，下面进行简单介绍。

1. 新奥尔良方法

新奥尔良方法(New Orleans)是比较著名的规范化数据库设计方法。1978 年 10 月，来自 30 多个国家的数据库专家在美国新奥尔良市专门讨论数据库设计问题，他们运用软件工程的思想和方法，提出了数据库设计的规范，这就是著名的新奥尔良方法，它是目前公认的比较完整和权威的一种规范化数据库设计方法。

新奥尔良方法将数据库设计分成需求分析、概念设计、逻辑设计和物理设计四个阶段。其中需求分析阶段主要获取和分析应用的信息需求与功能需求，概念设计阶段主要建立数据库应用系统的概念模型，逻辑设计阶段建立数据库的逻辑结构，物理设计阶段建立数据库的物理结构并加以实现。

目前，其他的规范化数据库设计方法大多基于新奥尔良方法，并在设计的每一阶段通过采用一些辅助方法来具体实现。因此，可以将新奥尔良方法作为规范化数据库设计方法的代表，并借助其他的一些设计技术来完善数据库设计过程。例如，下面介绍的基于 ER 模型的数据库设计方法强调在概念设计阶段采用 ER 模型方法，基于 3NF 的数据库设计方法强调在逻辑设计阶段以 3NF 为设计目标。

2. 基于 ER 模型的数据库设计方法

基于 ER 模型的数据库设计方法在新奥尔良方法的基础上，强调在概念设计阶段使用 ER 模型来建立数据库的概念模型。

ER 模型是由 Peter Chen 于 1976 年提出的数据库概念建模方法，其基本思想是在需求分析的基础上，用 ER(Entity-Relationship，实体-联系)图构造一个反映现实世界实体以及实体间联系的高层信息模型。

基于 ER 模型的数据库设计方法可以看作对新奥尔良方法的一个补充。

3. 基于 3NF 的数据库设计方法

基于 3NF 的数据库设计方法是由 S. Atre 提出的结构化设计方法，其基本思想是在需求分析的基础上，确定数据库模式中的全部属性和属性间的依赖关系，将它们组织在一个单一的关系模式中，然后分析模式中不符合 3NF 的约束条件，将其进行投影分解，规范化为若干个 3NF 关系模式的集合。

其具体设计步骤分为五个阶段：

(1) 设计企业模式，利用规范化得到的 3NF 关系模式画出企业模式；

(2) 设计数据库的概念模式，把企业模式转换成 DBMS 所能接受的概念模式，并根据概念模式导出各个应用的外模式；

(3) 设计数据库的物理模式(存储模式)；

(4) 对物理模式进行评价；

(5) 实现数据库。

4. 基于视图的数据库设计方法

基于视图的数据库设计方法从分析各个应用的数据着手，其基本思想是为每个应用建立自己的视图，然后把这些视图汇总起来合并成整个数据库的概念模式。合并过程中要解决以下问题：

(1) 消除命名冲突；

(2) 消除冗余的实体和联系；

(3) 进行模式重构，在消除了命名冲突和冗余后，需要对整个汇总模式进行调整，使其满足全部完整性约束条件。

规范化数据库设计方法从本质上来说仍然是手工设计方法，其基本思想是过程迭代和逐步求精。

5. 计算机辅助的数据库设计方法

计算机辅助的数据库设计方法是指在数据库设计的某些过程中模拟某一规范化设计，并以人的知识或经验为主导，通过人机交互方式实现设计中的某些部分。

目前许多计算机辅助软件工程(Computer Aided Software Engineering，CASE)工具可以自动或辅助设计人员完成数据库设计过程中的很多任务，如 PowerDesigner(SAP Sybase)、Design 2000(Oracle)、WorkBench(MySQL)等。CASE 工具辅助数据库设计不仅可以方便地建立图形化模型，而且可以更好地维护数据库设计过程中建立的各种模型信息。

从上面的讨论可以看到，目前新奥尔良方法是流行的规范化数据库设计方法，其他的一些设计方法都只侧重于从某个方面改进新奥尔良方法，强调在某个阶段使用某种特定的实现方法。

因此，在实际应用中，建议以新奥尔良方法为基础，并结合其他的一些方法进行数据库设计。例如，就目前的技术而言，以新奥尔良方法为基础，基于 ER 模型和 3NF，采用计算机辅助进行数据库设计，是一种可行的选择。这一方法总体上要求按照新奥尔良方法的过程开展数据库设计，但是在概念设计阶段要求使用 ER 模型进行设计，在逻辑设计阶

段要求以关系模式为基础并以 3NF 为目标进行优化设计，整个设计过程要求在 CASE 工具辅助下进行。

7.1.2　数据库设计的过程

数据库设计过程通常是与软件的开发过程结合在一起进行的。数据库设计过程一般包括六个阶段：需求分析、概念设计、逻辑设计、物理设计、数据库实施、数据库运行与维护，其中新奥尔良方法的四个过程是核心，同时加入了与软件开发过程相衔接的两个过程。

1. 需求分析

需求分析阶段完成需求信息的收集和整理，包括系统的信息需求和处理需求。数据库设计中的需求分析通常和软件工程的需求分析一起完成。因此，软件开发过程中形成的需求规格说明书都可以作为数据库需求分析的参考。此外，由于数据库需求分析强调信息需求，而软件工程需求分析强调处理需求，因此数据库设计者在整个软件开发过程的需求分析过程中要积极参与，从数据库的角度与用户交互以获取确切的数据库设计和使用需求。

2. 概念设计

概念设计阶段对用户的需求进行综合、归纳和抽象，产生一个独立于 DBMS 的概念模型，从用户的角度描述了现实世界的信息结构。

概念设计一般通过一组图形来表达现实世界的数据需求，它是数据库设计中至关重要的阶段，因为它是需求分析和逻辑设计之间的桥梁。需求分析通常没有考虑数据如何组织管理的问题，逻辑设计则与底层的 DBMS 相关。由于直接将现实世界中的需求转换为数据库逻辑结构比较困难，因此引入中间层——概念设计来解决这一问题。从用户角度看，概念模型是与计算机系统无关的，因此它容易理解，可以较好地表达用户的需求，而且用户也能明白概念模型的含义；从数据库设计者的角度看，概念模型最终可以转换为数据库逻辑结构，因此可以达到数据库设计的目的。

3. 逻辑设计

逻辑设计阶段的主要任务是建立数据库的逻辑结构。一般地，数据库逻辑结构指数据库概念模式结构和外模式结构(内模式涉及物理结构，在物理设计阶段进行设计)。在逻辑设计阶段，将概念模型转换为某个 DBMS 支持的数据模型(如关系数据模型)，并建立数据库模式结构，同时，还需要对建立的数据库模式结构进行优化(如规范化)，最终产生一个优化的数据库逻辑结构。

4. 物理设计

物理设计阶段的主要任务是为数据库逻辑结构选取最适合应用环境的数据库物理结构，包括存储结构和存取方法。不同 DBMS 支持的存储结构和存取方法有一定的差异，因此物理设计需要结合底层 DBMS 的特征开展。

5. 数据库实施

数据库实施阶段的主要任务是根据逻辑设计和物理设计的结果，用 DDL 建立数据库并装入初始数据，同时编写与调试应用程序，并完成数据库的一些维护性设计(如故障恢复设计、安全性设计等)。

数据库实施与软件工程的编码阶段有点类似，因为这一阶段也需要编程，当然主要是 SQL 编程，包括用 DDL 建立基本表、视图等结构，导入或编写 SQL 脚本建立初始的数据库(许多系统都有初始的一些数据)。另一个比较重要的任务是编写存储过程和触发器。

6. 数据库运行与维护

数据库运行与维护阶段通常是与数据库应用系统的试运行结合在一起的。因为数据库本身并不能构成一个应用系统，必须和前端的业务处理结合起来才能最终响应用户的需求。在试运行过程中，如果出现数据库方面的问题，则需要数据库设计者进行分析、调试、修改。

图 7-1 给出了数据库设计的过程。从图中可以看到，数据库设计的输入总共有三个方面。

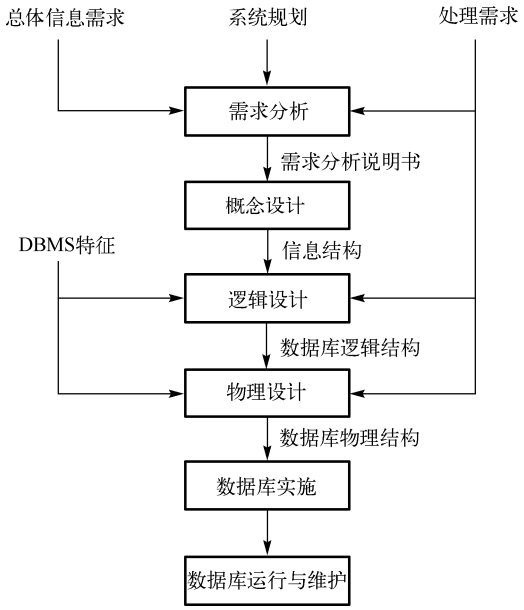


图 7-1　数据库设计的过程

(1) 总体信息需求：包括数据库应用系统的目标、数据元素的定义、数据在组织中的使用描述等。

(2) 处理需求：每个应用需要的数据项、数据量以及处理频率等与处理相关的需求。

(3) DBMS 特征：DBMS 说明、支持的模式、程序语法等。

数据库设计的最终结果是产生一份数据库设计说明书(完整的数据库逻辑结构和物理结构、应用程序设计说明)。数据库设计说明书将作为软件详细设计和编码阶段的重要依据。

7.2　需 求 分 析

在软件工程的需求分析阶段，设计者主要关注系统的处理需求、信息需求以及性能、可靠性等其他需求。数据库设计的需求分析阶段与软件工程的需求分析阶段相比唯一不同的地方就是它主要关注系统的信息需求。

软件工程的需求分析阶段通常会产生一个数据字典，其中包含数据流、数据处理、数据项、数据存储、数据结构以及处理过程的描述。这些信息可以作为数据库设计需求分析阶段的基本参照。例如，数据存储中的内容往往是数据库设计需要重点关注的，因为它们

通常都表示了系统中的实体信息。另外，数据结构、数据流也有可能是形成关系模式的内容，也需要加以仔细分析。数据项一般对应着关系模式中的属性。处理过程对于数据库设计来说意义不大，因此一般不需要在数据库设计阶段对它进行分析。

7.3 概 念 设 计

数据库概念设计的主要目的是建立反映组织信息需求的数据库概念结构，即概念模型。概念模型独立于数据库逻辑结构、DBMS 以及计算机系统，侧重于语义表达，因此有时候也把它称作“语义模型”。概念设计侧重于数据内容的分析和抽象，以用户的观点描述应用中的实体以及实体间的联系，但是它不描述任何应用系统的行为特征(即处理需求)。一般地，在软件工程中，系统的处理需求通过数据流程图等工具表达，而信息需求则由概念模型进行表达。

目前，数据库概念设计阶段流行的方法是 ER 方法。ER 方法通过建立反映系统信息需求的一组 ER 图(ER Diagram)来表达现实应用中的实体以及实体之间的联系。

7.3.1 ER 模型概述

ER 模型是 1976 年由美国路易斯安那州立大学(Louisiana State University)的华人教授 Peter P. Chen(陈品山)提出的。ER 模型的论文 *The Entity-Relationship Model—Toward a Unified View of Data* 发表在第一期的 *ACM Transaction on Database Systems*(数据库领域顶级的期刊之一)上，是目前计算机科学技术领域中引用率最高的论文之一。ER 模型开辟了数据库概念建模这一新的研究方向，并引起了国际数据库界的广泛关注。数据库领域还专门创立了关于概念建模的国际会议(International Conference on Conceptual Modeling)，即 ER 会议。ER 会议现在是数据库领域非常著名的国际会议之一，具有很高的声望。

ER 模型是一个非常简单的概念模型，但它能够用简单的方法表达现实世界的信息需求。在 ER 模型中，现实世界中的所有数据都表示为“实体”，实体之间的关联用术语“联系”来表达。因此，ER 模型的基本要素只有三个。

(1)实体：现实世界的所有数据都抽象为实体。

(2)联系：实体之间的关联，联系有不同的类型。

(3)属性：实体和联系都可以包含若干描述属性，反映实体和联系的特征。

例如，在一个教学系统中，实体有教师、学生、课程等，每个实体都有一些描述属性，例如，学生有学号、姓名、专业等属性，课程有课程编号、名称、学分等属性。这些属性都是根据前面的需求分析结果确定的。然后，学生与课程之间存在选课联系，并且是多对多类型，表示一个学生可以选修多门课程，同时一门课程允许多名学生选。

正如其名称所示，ER 模型的核心要素就是实体和联系。另外，实体和联系都有相应的属性。因此，ER 模型的组成总共包括三个元素：实体、联系和属性。

1. 实体

实体是现实世界中可标识的对象。实体的一个重要特征是它在现实世界(或应用范围内)是可以唯一标识的。此外，实体具有相应的实体名。

现实世界中的实体可分为两类。一类是物理实体，如学生、员工等，他们都是物理可见的对象。另一类是抽象实体，如课程、国家等，它们代表的是一类可区分的抽象概念。

由于实体都是可以唯一区分的，因此每一个实体都有唯一的标识。这个唯一的标识称为实体的码(Key)。例如，学生实体的码是学号，课程实体的码是课程编号，国家实体的码是国家名。

在 ER 模型中，实体是通过一系列的属性来描述的。属性反映了一个实体的不同侧面的信息，例如，学生实体可以通过学号、姓名、年龄、专业等属性来描述。实体的码也是属性。一个实体究竟用什么样的属性来描述是由不同的应用环境来决定的。类似的实体在不同的应用环境中可能具有差异很大的属性描述。例如，同样是员工实体，在公司 *A* 的系统中可能使用了 10 个属性来描述，而在公司 *B* 的系统中可能会使用 20 个属性来描述。这完全取决于不同应用环境的需求。

2. 联系

在 ER 模型中，联系是实体与实体之间的某种关联。联系也具有相应的联系名。之所以在 ER 模型中引入联系这一元素，是因为实际应用环境中的实体与实体之间总是存在着相互联系。为了描述现实世界中实体之间的联系，ER 模型使用了联系这一元素来进行表达。

首先，联系必须是建立在实体与实体之间的，不能建立在实体与属性之间。其次，不同实体之间的联系有着不同的类型。在 ER 模型中，实体之间的联系可分为三种类型。

(1)1∶1 联系(一对一联系)：实体 *A* 和实体 *B* 存在 1∶1 联系，是指一个实体 *A* 只能与一个实体 *B* 发生联系，反过来一个实体 *B* 也只能与一个实体 *A* 发生联系。例如，国家和总统之间一般是 1∶1 的联系，因为一个国家只有一个总统，而一个总统也只能是一个国家的总统。

(2)1∶*N* 联系(一对多联系)：实体 *A* 和实体 *B* 存在 1∶*N* 联系，是指一个实体 *A* 可以与一个或多个实体 *B* 发生联系，但一个实体 *B* 只能与一个实体 *A* 发生联系。例如，班级与学生之间是 1∶*N* 联系，因为一个班级有多名学生，但一名学生只属于一个班级。

(3)*M*∶*N* 联系(多对多联系)：实体 *A* 和实体 *B* 存在 *M*∶*N* 联系，是指一个实体 *A* 可以与一个或多个实体 *B* 发生联系，同时一个实体 *B* 可以和一个或多个实体 *A* 发生联系。例如，学生和课程是 *M*∶*N* 的联系，因为一个学生可以选修多门课程，而一门课程也可以被多个学生同时选修。

联系的确定依赖于实体的定义和应用环境的定义。同样的实体在不同应用中可能有不同的联系。例如，考虑部门和职工之间的联系，如果一个职工只能属于一个部门，则部门和职工之间是 1∶*N* 联系；如果一个职工可属于多个部门，则是 *M*∶*N* 联系。再考虑另一个图书馆和图书的例子。如果图书的码定义为索书号，则图书馆和图书之间为 *M*∶*N* 联系(一个索书号可能有几本相同的书)；如果图书的码为图书条码，并且每本书有唯一条码，则图书馆和图书之间为 1∶*N* 联系。

3. 属性

实体和联系都可以拥有一些描述自身特性的数据项，称为属性。实体通常有多个属性，它们构成了一个属性集。在这些属性中，可以唯一标识实体的属性就是实体的码。联系本

身也可以有描述属性。联系的属性是指由于实体与实体之间建立了联系而新产生的描述信息。例如，学生与课程之间建立了选课联系之后一般就需要有“成绩”，这个“成绩”就是联系的一个属性，它既不是描述学生实体的属性，也不是描述课程的属性，而是在学生选修了课程之后新产生的属性。联系是否具有属性，关键要看应用环境是否需要记录与联系相关的信息。

属性一般具有一个属性名和一个域。域代表了属性可以取值的一个范围，它可以是一个类型，也可以是某种自定义的取值范围。由于 ER 模型强调语义表达，并不关心具体的数据库系统实现，因此在 ER 模型中并不强制要求属性的域必须是数据库系统支持的数据类型。

7.3.2　ER 模型的符号

ER 模型通过建立由实体、联系和属性构成的 ER 图来表示现实世界的数据需求。ER 图的基本符号如图 7-2 所示。

图 7-2　ER 模型的基本符号

如图 7-2 所示，实体在 ER 模型中表示为矩形框，联系表示为菱形框，而属性则通过椭圆形框来表示。属性与实体或联系之间通过线段相连。如果某个属性是实体的码，则在属性下方加上下画线。

图 7-3 给出了一个教学应用的 ER 模型示例。在该应用中，有三个实体：教师、学生和课程。学生和课程之间存在着 M∶N 的选课联系，教师和课程之间也存在着 M∶N 的授课联系。

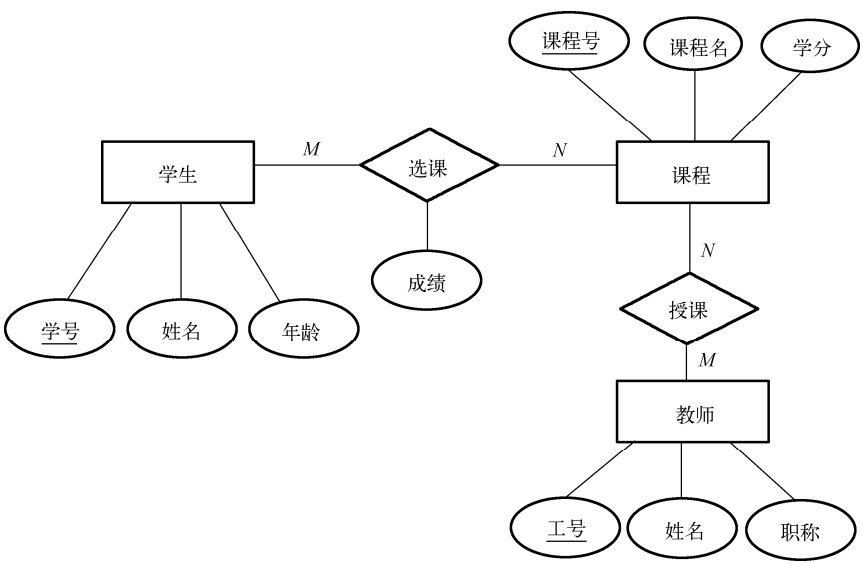


图 7-3　教学应用的 ER 模型示例

7.3.3　ER 模型的设计过程

ER 模型的设计是一个自底向上的过程。通常，应用系统的需求分析是自顶向下进行的，但 ER 模型的设计过程正好相反。需求分析因为要不断细化，所以必须从顶层分析开始。但由于顶层的 ER 模型(即整个应用系统的 ER 模型)在系统数据规模大、复杂程度高时很难直接进行设计，所以必须采用自底向上的设计方法，即先设计底层子系统或模块的分 ER 模型，然后将这些反映系统局部数据需求的分 ER 模型集成为一个反映系统整体数据需求的 ER 模型。

ER 模型的这一设计过程可归纳为以下三个步骤。

(1)分 ER 模型的设计，即设计底层各个子系统的 ER 模型。

(2)ER 模型的集成，即将各个分 ER 模型集成为一个反映整体数据需求的 ER 模型。

(3)ER 模型的优化。由于采用了先设计分 ER 模型然后集成的设计方法，在集成过程中可能会出现一些设计上的问题，因此执行一个优化过程对集成后的 ER 模型进行优化，使得最终得到的全局 ER 模型具有更好的表达效果和性能。

图 7-4 给出了全局 ER 模型设计过程的一个示意图。例如，如果要为某个校园管理信息系统设计 ER 模型，整个设计过程可以大致表示为图 7-5 所显示的顺序。

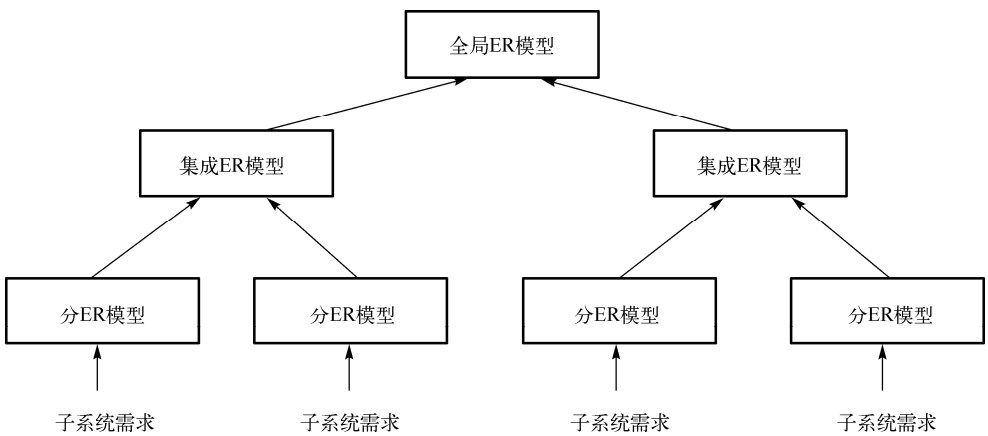


图 7-4　ER 模型的设计过程

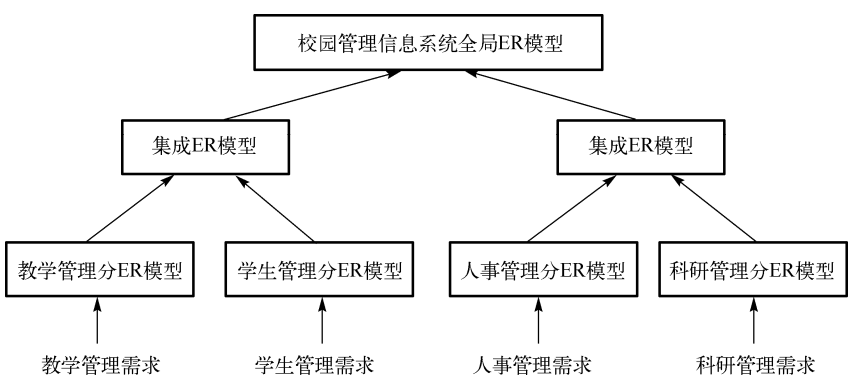


图 7-5　校园管理信息系统 ER 模型设计过程示例

7.3.4　分 ER 模型的设计

分 ER 模型的设计是 ER 模型设计过程中最为重要的一个步骤。它直接表达了底层各个子系统或模块的数据需求。分 ER 模型的要素同样包括实体、联系和属性，因此在设计时也主要从这三个方面进行考虑。由于属性是用来描述实体或者联系的，在设计过程中可以将属性的确定与实体和联系的设计结合在一起，因此分 ER 模型的设计可以分为实体设计和联系设计两个方面。

1. 实体设计

实体设计包括两个方面的工作，一是要确定系统范围内有多少种实体，二是要确定各个实体的属性集合。

实体设计的基本原则有两点。第一点：实体要尽可能少。这是因为实体越多，意味着将来数据库中需要存储的对象也越多(例如，在关系数据库中就会有更多的基本表)，数据库就越复杂，容易使得数据访问编程复杂化，也会导致最终的数据库应用系统性能下降。第二点：现实世界中的事物若能作为属性就尽量作为属性对待。这一点与第一点是相关的，目的是在保证数据表达能力的基础上尽量地减少实体的数量，降低概念数据模型乃至未来数据库的复杂度。

1)确定实体

应用系统中究竟存在哪些实体，唯一的依据就是系统的需求。由于不同应用环境的需求有着较大的差别，因此不能根据主观经验来进行实体设计，必须依据系统需求。

实体的确定有几种不同的方法。

(1)实体是现实世界中可唯一标识的对象，因此现实应用中某些实体很容易区分，如教学信息管理系统中的学生实体、课程实体等。

(2)如果某个对象是一个属性的集合，那么一般要将其确定为实体。例如，如果一个地址是由城市名、街道名、门牌号和邮政编码四个属性来描述的，那么需要将其确定为一个实体。反之，如果应用系统中地址仅仅是一个字符串，那么就只需要将其设计为属性。

(3)实体的设计可以参考软件工程需求分析报告中的数据字典信息。在软件工程需求分析阶段一般会得到应用系统的需求规格说明，其中包含定义了系统中的数据存储、数据项、数据流、数据结构以及数据处理的数据字典。在数据字典中，通常可以考虑将数据存储、数据流、数据结构作为实体的候选。如果这些对象是属性的集合，那么可以将它们确定为实体。

2)确定实体的属性

实体和属性都是现实世界数据的组成，而且很多时候实体和属性之间并没有可以截然区分的特点。事实上，如何准确地区分实体和属性是 ER 模型设计中的一个难点问题。

确定实体属性的首要工作是确定实体的码。另外，只需要考虑系统范围内的实体属性。例如，学生实体可以有许多属性，如学号、姓名、年龄、身高、体重、血型等，但只需要包含那些所针对的应用系统需要的属性。举个例子，如果应用系统的需求中不需要对学生的血型进行管理，那么血型就是系统范围之外的数据，不能将其作为学生实体的属性。如

果不正确地把血型设计成了学生实体的属性会出现什么问题？假设由于系统需求中并不涉及学生的血型(学生登记表上并没有“血型”一栏)，因此“血型”这一数据将来在应用系统中既不会有输入的数据，也不会被用户使用，则成了系统中的垃圾数据。实体的每一个属性还要确定它的域。域同样是需要根据现实世界中属性的取值特点来确定的，这样可以保证在将来的数据库中的属性值与应用中的取值范围相符，使得建立的概念数据模型可以反映真实世界的数据特征。

实体属性还要满足两个准则：一是属性必须是不可分的，即不能包含其他属性集；二是属性不能与实体发生联系，联系必须是发生在实体与实体之间的。

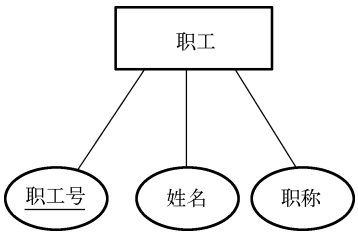


图 7-6　职称属性不可分时的职工实体设计

对于第一个准则，以图 7-6 为例进行说明。职工是一个实体，职工号、姓名、年龄是职工的属性，如果职工的职称没有进一步的特定描述，则可以作为职工的属性。但如果职称与工资、福利等相关，即职称本身还有一些描述属性，则把职称设计为实体比较恰当(图 7-7)。

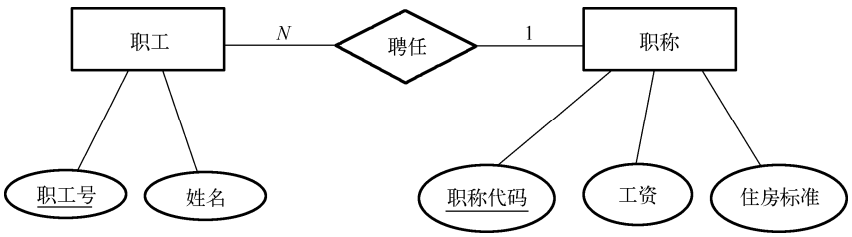


图 7-7　职称属性包含其他属性时的职工实体设计

对于第二个准则，即属性不能与实体发生联系，可以看图 7-8 中的例子。在医院信息系统中，一个病人只能住在一个病房里，因此住院号可以作为病人实体的一个属性。但如果医生实体与病房之间存在负责联系，即一个医生要负责管理多个病房，而一个病房的管理医生只有一个，此时，病房不能再设计为属性，而应当设计为实体(图 7-9)。

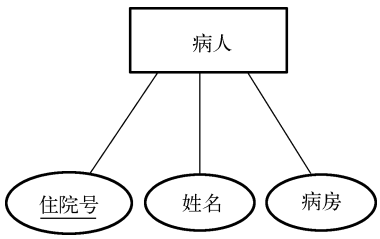


图 7-8　病房作为属性的病人实体设计

2. 联系设计

联系设计包括两个方面的工作，一是要确定哪些实体之间存在联系，二是要确定联系的属性与基数。

联系的确定依据是应用的需求。两个实体在不同的应用环境中的联系可能会完全不同。例如，考虑顾客实体和银行账户实体之间的联系，可能在 *A* 银行是 1∶*N* 的联系，因为一个顾客可以拥有多个账户，但一个账户只有一个所有人，而在 *B* 银行可能是 *M*∶*N* 的联系，因为 *B* 银行中一个账户可以有多个所有人(如夫妻双方共同所有)。因此，要根据应用环境的特点来设计联系的类型。

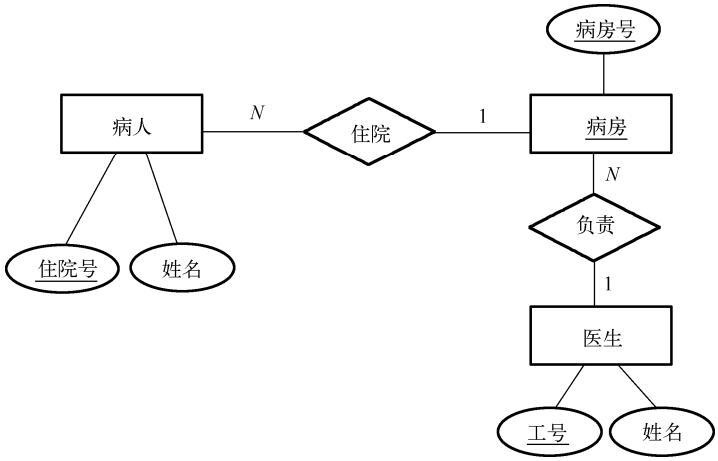


图 7-9　病房作为实体的 ER 模型设计

联系设计中容易出错的地方主要有两处。一是联系的基数。例如，本来是 1∶*N* 的联系设计成了 *M*∶*N* 的联系。二是联系属性的确定。注意，不能将联系的属性设计为关联实体的属性，也不能漏掉联系应该具有的属性。例如，在学生实体和课程实体之间存在选课联系，“成绩”就是选课联系的属性。注意，“成绩”属性既不是描述学生实体的数据，也不是描述课程实体的数据。实体的属性都是实体自身拥有的本质属性，而“成绩”显然不是学生实体或者课程实体所拥有的本质属性。

7.3.5　ER 模型的集成

完成了各个底层子系统的分 ER 模型设计之后，下一步是进行 ER 模型的集成。由于一个应用系统中的数据普遍存在着各种关联，因此一个实体有可能会同时出现在多个分 ER 模型中。例如，在校园管理信息系统中，教师实体既出现在人事管理分 ER 模型中，也出现在教学管理分 ER 模型中。把这些同时出现在多个分 ER 模型中的实体称为公共实体。

ER 模型的集成首先是确定各个分 ER 模型的公共实体，然后是基于公共实体进行 ER 模型的合并，最后是消除合并过程中出现的各种冲突。通常，分 ER 模型设计时因为规模较小，不会出现冲突，但合并时容易出现冲突。这里有两个原因。一是各个子系统的用户对于数据的理解可能存在偏差，例如，教学管理部门会认为只需要存储教师的姓名、部门和职称就可以了，而人事管理部门则可能需要管理教师的更多信息。这样导致的结果就是同样的教师实体在教学管理分 ER 图和人事管理分 ER 图中的属性集不同，形成了属性冲突。二是各个分 ER 模型在实际设计过程中常常是由不同的设计人员完成的，而设计人员之间在设计习惯、设计水平方面往往存在差异，因此也会导致对同样的数据形成了不同的概念建模结果。例如，科研管理子系统的设计人员习惯将科研项目命名为“项目”，而另一些设计人员习惯将科研项目命名为“课题”，这样一来在集成时就会出现“异名同义”的冲突。

ER 模型在集成过程中可能出现的冲突大致可分为三类。

(1) 属性冲突。属性冲突主要表现为属性的类型冲突或者值冲突。类型冲突是指同一

属性的域不同，例如，“性别”属性在某个分 ER 模型中是整数类型，而在另一个分 ER 模型中是字符串类型。对于类型冲突，需要在集成时进行统一化。值冲突是指不同分 ER 模型对于属性的取值要求不同。例如，同样是“性别”属性，在一个分 ER 图中规定 1 是男，0 是女，而在另一个分 ER 图中则规定 1 是女，0 是男。值冲突同样需要在集成时进行取值的统一化处理。

(2) 结构冲突。结构冲突有三种表现形式：实体属性集不同、联系类型不同、同一对象在不同应用中的抽象不同。实体属性集不同是指同一实体在不同的分 ER 模型中的属性集不同，通常可以采用合并属性集等方式加以解决。联系类型不同是指两个实体之间的联系在不同分 ER 模型中不同。这可能有两种原因。一种原因是这两个实体之间只有一种联系，但在不同的分 ER 模型中联系的命名出现了异名同义的情况，此时将这两种联系统一为一种即可。另一种原因是这两个联系本来就是两个实体之间存在的两种不同的联系，此时需要将这两个联系都保留。同一对象在不同应用中的抽象不同是指同样的数据在某个分 ER 模型中设计为属性，而在另一个分 ER 模型中却设计为实体，如图 7-8 和图 7-9 中的病房设计。这种情况需要仔细分析。一般来说，如果确实需要设计为实体，那么需要将属性转变为实体。

(3) 命名冲突。命名冲突包括同名异义和异名同义两种情况。无论实体、联系还是属性，都有可能出现命名冲突，尤其是实体和属性的命名。命名冲突通常通过认真核对都可以发现并解决。

7.3.6　ER 模型的优化

集成后 ER 模型优化的目的是使模型中的实体尽可能少，属性尽可能少，同时保证联系之间没有冗余。由于分开设计的原因，各个分 ER 模型之间有可能会出现冗余属性和联系，这是 ER 模型优化的主要出发点。

ER 模型优化主要包括三个方面：合并实体、消除冗余属性、消除冗余联系。

1. 合并实体

合并实体的目的是减少实体的数量。一般来说，如果两个实体之间是 1∶1 联系，则可以考虑将它们合并为一个实体。如果两个实体在实际应用系统中总是一起查询的，也可以考虑合并。例如，对于病人实体和病历实体，通常查询病人时总是要同时查询病人的病历，因此可以将病历实体合并到病人实体中。这样做的目的是减少查询时的连接开销，提高系统的响应性能。但这种合并并不是只有好处，其副作用是使得数据库设计容易出现比较多的设计问题(这一点将在后面的章节中详细讨论)。因此，能否合并还要看系统的性能需求和设计需求之间如何折中，但这种方式至少提供了一个实体合并思路。

2. 消除冗余属性

通常，分 ER 模型中一般不存在冗余属性，但集成后可能产生冗余属性。例如，在一个教育信息系统中，一个分 ER 模型含有高校毕业生数、在校学生数，另一个分 ER 图含有招生数、在校学生数，每个分 ER 模型没有冗余属性，但集成后“在校学生数”冗余，应加以消除。

冗余属性的出现有几种情形：一是同一非码属性出现在几个实体中；二是一个属性值可从其他属性值中导出，如出生日期和年龄。

3. 消除冗余联系

冗余联系的出现与冗余属性类似，通常都是由于集成而产生的。冗余联系指的是两个实体之间的联系可以通过其他的联系推理得到。例如，图 7-10 中，教师和学生之间的联系就是冗余联系，它可以通过学生与课程、课程与教师之间的联系推理得到。

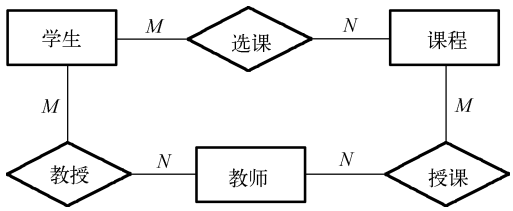


图 7-10　冗余联系示例

7.3.7　ER 模型的扩展

传统的 ER 模型支持实体和实体间三种联系类型的表达，即 1∶1、1∶*N*、*M*∶*N*。但在许多时候，这三种联系类型无法表达实体与实体之间的一些特殊语义，因此在 ER 模型提出以后，数据库领域的研究者针对传统 ER 模型语义表达能力不足的问题，提出了一些 ER 模型的扩展方法。这些扩展方法使得 ER 模型可以表达一些特殊的语义，从而保证 ER 模型可以更合理地表示现实世界的数据语义。

本节主要讨论两种 ER 模型的扩展方法。第一种是子类与超类，第二种是弱实体。这两种扩展方法目前在实际的 ER 建模中都是广泛使用的。

1. 子类与超类

图 7-11 给出了引入子类与超类扩展的原因。在图 7-11 中，部门和职工之间存在两种联系：“工作”和“领导”，其中部门和职工之间 1∶*N* 的“工作”联系指的是每个职工都在一个部门中工作，而一个部门拥有多个职工，1∶1 的“领导”联系指的是每个部门都有一个职工是部门领导。然而事实上，并非所有职工都与部门之间存在着“领导”联系，但传统的 ER 模型无法对实体进行区分，因此不能准确地表达类似的语义。

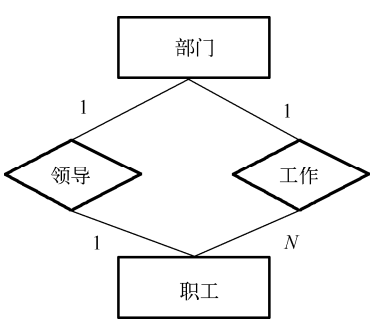


图 7-11　传统 ER 模型无法区分一般职工与部门领导

子类(Subtype)与超类(Supertype)的引入可以解决实体之间的区分问题。给定两个实体 *A* 和 *B*，如果实体 *A* 属于某种特殊类型的实体 *B*，则称实体 *A* 为实体 *B* 的子类，实体 *B* 为实体 *A* 的超类。子类实体是超类实体的特殊化(Specialization)，而超类实体是子类实体的一般化(Generalization)。子

类与超类的概念与面向对象程序设计中的继承概念类似，都反映实体之间的继承关系，不同之处在于 ER 模型中只考虑实体的数据特征，不考虑实体的行为特征(在面向对象程序设计中，一个对象既具有表示数据特征的属性集合，也具有表示行为特征的方法集合)。

子类实体继承了超类实体的全部属性，包括超类实体的码，因此子类实体的码与超类实体的码是一样的。例如，研究生实体是学生实体的子类，那么学生实体就是研究生实体的超类；同时，学生实体的码是学号，那么研究生实体的码也是学号，因为它继承了学生实体的所有属性。

在实际的 ER 模型设计中，一方面可以根据已有的实体设计结果增加子类实体，另一方面也可以抽象出新的超类，从而使模型可以更准确地表达数据的特征。

子类与超类在 ER 模型中通过引入新的图形符号 ISA 进行表达。图 7-12 给出了子类与超类的图形建模示例。

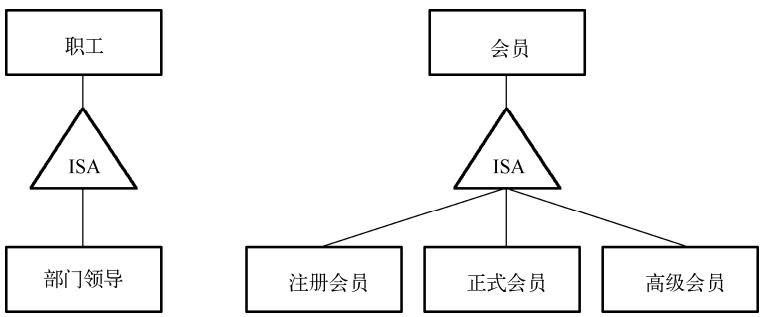


图 7-12　子类与超类的图形建模示例

2. *弱实体*

弱实体(Weak Entity)又称为依赖实体，是应用系统中一类特殊的实体，这类实体在应用系统中的存在必须以系统中另一个实体的存在为前提。如果其所依赖的实体在系统中被删除了，那么弱实体也要随之删除。把弱实体所依赖的实体称为强实体(Strong Entity)或常规实体(Regular Entity)。

弱实体也具有自己的标识(码)，但它的码只是相对于它所依赖的强实体而言是唯一的，在整个系统范围之内可能并不是唯一的。这是弱实体和强实体的主要区别。

举个例子，在一个公司人事系统中，需要同时管理职工以及职工的子女(例如，要给每个职工的子女发放某种补贴)。这里，子女就是弱实体，因为他们在公司系统中存在的前提是他们的父母(即职工实体)必须首先存在于系统中。如果他们所依赖的强实体(即父母)离职了，也就是说不再存在于系统中了，那么他们的信息也会随之被删除，因此公司已经不再需要对他们的信息进行管理了。

一个实体是否是弱实体要看应用系统对它的定义。例如，同样是职工和子女，如果在社区人口管理系统中，则都是强实体，因为即使子女的父母去世了(不再存在于系统中了)，子女实体仍然需要在系统中维护。

弱实体反映了实体之间的一种特殊的依赖关系。弱实体的引入有助于更准确地表达现实世界中实体之间的联系。

弱实体在 ER 模型中表示为双线矩形框，与其所依赖的强实体之间的联系用双线菱形框表示，并且通过箭头指向强实体。图 7-13 给出了弱实体的图形建模示例。

图 7-13　弱实体的图形建模示例

7.4　逻 辑 设 计

完成了概念设计之后，得到了一个概念模型，它描述了整个应用系统的信息需求。逻辑设计阶段以概念模型为基础，结合应用系统的处理需求，将概念模型转换为可以在特定 DBMS 上实现的结构数据模型并加以优化。一般地，使用关系数据模型作为目标数据模型，因此逻辑设计阶段将最终建立优化的关系数据库模式结构。

7.4.1　逻辑设计的任务

数据库逻辑设计的任务和过程如图 7-14 所示，逻辑设计主要包括以下几个方面的任务。

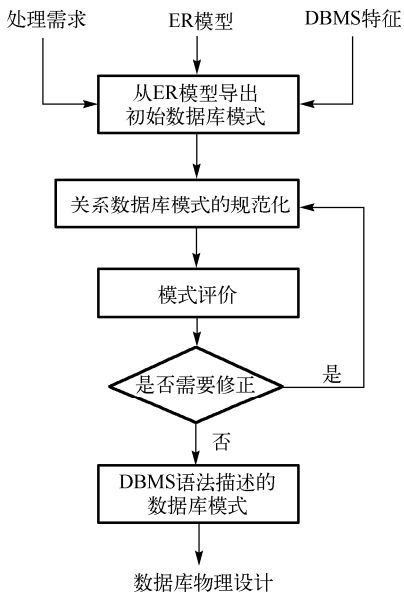


图 7-14　数据库逻辑设计的任务和过程

1. ER 模型转换成关系数据库模式

逻辑设计阶段的第一个任务就是将 ER 模型转换为初始的数据库模式。这一步在整个数据库设计过程中非常关键，因为它完成了从“面向用户的设计”到“面向实现的设计”之间的转换和衔接。

2. 关系数据库模式的规范化

这一步主要完成初始数据库模式的优化。第一步得到的初始数据库模式有可能范式级别较低，因此容易出现模式设计问题并影响到系统的功能和性能。规范化过程就是将初始的数据库模式优化到高级别范式的过程。这一步中首先需要确定初始数据库模式的范式级别，还需要确定数据库模式最终想要达到的范式级别。在实际的数据库设计中，一般以 3NF 为目标范式进行设计。

3. 模式评价

这一步对上一步规范化后的关系数据库模式进行评价。评价的主要指标包括功能指标和性能指标。其中性能指标是评价的重点，这是因为规范化过程本身就是模式分解的过程，模式分解后系统中的连接查询就会增多。如果这些连接查询在整个系统中非常频繁，很显然会影响到平均性能。

4. 模式修正

如果模式评价的结果是现有的关系数据库模式不满足要求，则需要进行模式修正。模式修正包括功能上的修正和性能上的修正。

5. 导出 DBMS 语法描述的数据库模式

完成模式修正后得到了一个优化的全局关系数据库模式，这个全局关系数据库模式就是整个数据库的概念模式。逻辑设计阶段还要完成外模式设计的任务。外模式需要根据前端应用的需求进行设计。

7.4.2　从 ER 模型导出初始数据库模式

在实际中一般转换为关系数据库模式。在 7.3 节中介绍了基本的 ER 模型和扩展的 ER 模型，这两者之间的主要区别是对实体间联系的表达，基本 ER 模型中只有 1∶1、1∶*N* 和 *M*∶*N* 三种联系类型，而扩展的 ER 模型中加入了子类和弱实体两种特殊的实体间联系。下面分别讨论它们的转换方法。

1. 基本 ER 模型的转换

基本 ER 模型的转换过程分为实体转换和联系转换两部分。

1) 实体转换

每个实体转换为一个关系模式，实体的属性成为该关系模式的属性，实体的标识成为该关系模式的主码。关系模式的名字可以和实体名字不同。

2) 联系转换

对于 1∶1 联系，需将任一端的实体的标识和联系属性加入另一实体所对应的关系模式中，两个关系模式的主码保持不变。

对于 1∶*N* 联系，需将 1 端实体的标识和联系属性加入 *N* 端实体所对应的关系模式中，两个关系模式的主码不变。

对于 M∶N 联系，需要新建一个关系模式，该关系模式的属性包括两端实体的标识以及联系的属性，主码为两端实体的码的组合。

2. 扩展 ER 模型的转换

扩展 ER 模型主要包括弱实体和子类两种扩展结果的转换。

1) 弱实体转换

弱实体与某个强实体之间存在依赖关系，因此实际上是一类特殊的实体间联系。在转换时，首先将每个强实体转换为一个关系模式，强实体的属性成为关系模式的属性，实体标识成为主码；然后将每个弱实体转换为一个关系模式，加入其所依赖的强实体的标识，并且将关系模式的主码设置为弱实体的标识加上强实体的标识。

2) 子类转换

父类实体和子类实体都各自转换为关系模式，并在子类关系模式中加入父类的主码，子类关系模式的主码设为父类的主码。

图 7-15 给出了一个示例概念模型，下面将其转换为初始的关系数据库模式。

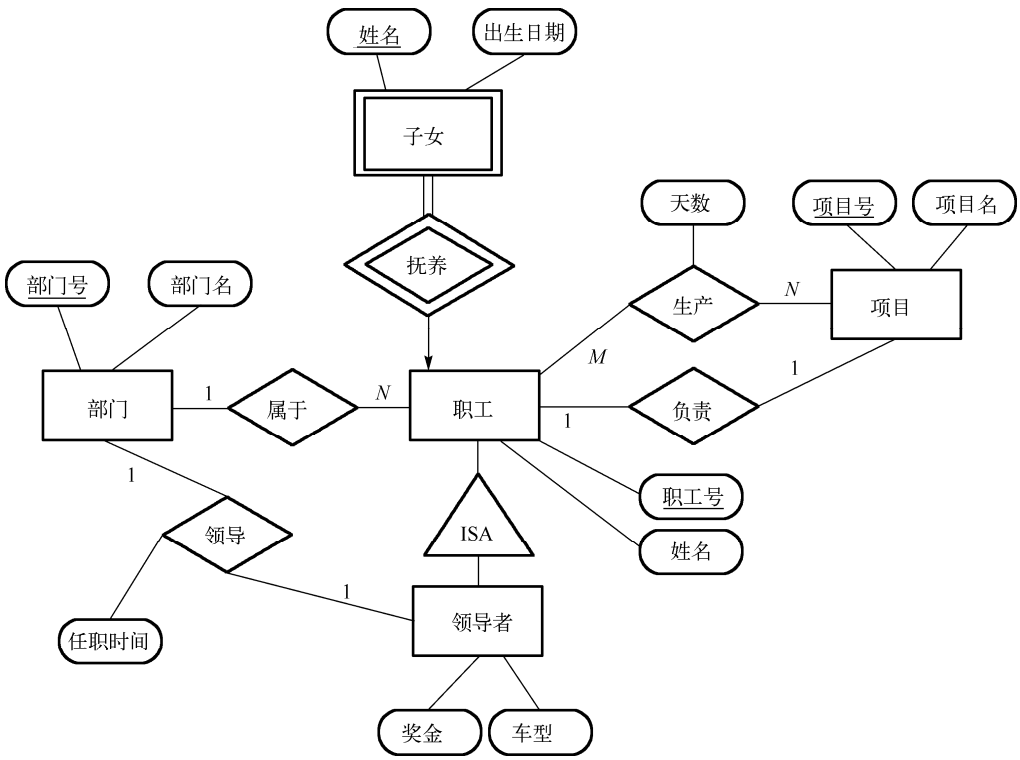


图 7-15　示例概念模型

(1) 实体转换为关系模式。每个实体转换为一个关系模式，转换后得到下面的结果。

①部门(部门号，部门名)。

②职工(职工号，姓名)。

③项目(项目号，项目名)。

④领导者(奖金，车型)。

⑤子女(姓名，出生日期)。

(2)弱实体转换。先执行弱实体转换的目的是将弱实体也转换为与强实体一致的关系模式，因为在关系数据模型中关系模式是没有强弱之分的。示例概念模型中只有一个弱实体“子女”，按照转换方法，在该弱实体中加入“职工号”，并修改其主码为“职工号+姓名”。新的关系模式为子女(姓名，出生日期，职工号)。

(3)子类转换。因为子类实体不含码，所以它和通常的关系模式也有差别，因此需要将子类实体先转换为普通的关系模式。示例概念模型中只有一个子类“领导者”，按照转换方法，在子类实体对应的关系模式中加入父类标识“职工号”，并将“职工号”作为该关系模式的主码。转换后的子类关系模式为领导者(奖金，车型，职工号)。

完成上面的步骤后，得到了下面的关系数据库模式。

①部门(部门号，部门名)。

②职工(职工号，姓名)。

③项目(项目号，项目名)。

④子女(姓名，出生日期，职工号)。

⑤领导者(奖金，车型，职工号)。

下面考虑联系的转换。

(4)联系转换。

示例概念模型中共有四个联系。考虑每个联系，并进行相应的转换。

①部门：领导者(1：1)。

1：1 联系需要在任何一端的实体中加入另一端实体的码以及联系属性，因此，下面的两种结果都是正确的：

领导者(奖金，车型，职工号，部门号，任职时间)

部门(部门号，部门名，职工号，任职时间)

②部门：职工(1：*N*)。

对于 1：*N* 的联系，需要将一端实体的码和联系属性加到 *N* 端实体中，因此得到

职工(职工号，姓名，部门号)

③职工：项目(1：1)。

这个联系表示职工和项目之间的负责关系。虽然这也是个 1：1 联系，但可选的两种转换结果在效果上有些差别。如果将“项目号”加到职工实体中，则得到职工(职工号，姓名，项目号)，由于并非每个职工都是项目负责人，因此最终该关系模式的实例的“项目号”属性上会出现大量的空值，带来存储空间上的浪费和性能上的影响。如果将“职工号”加到项目实体中，则情况会好得多，即项目(项目号，项目名，职工号)。理论上，1：1 联系的两种转换方法的效果应该是相同的，之所以出现这种情况，是因为在示例概念模型中并没有区分一般职工和项目负责人，所以，最好的方法还是增加一个子类实体“项目负责人”，并将此联系建立在项目实体和项目负责人实体之间。如果不增加子类，则要求在设计时对不同的转换结果做细致的分析。

④职工：项目(*M*：*N*)。

对于*M*：*N*联系，需要增加一个新的关系模式：职工_项目(项目号，职工号，天数)。此关系模式的码是两端实体标识的组合，同时也包含联系属性“天数”。

最终，得到的初始数据库模式如下。

①部门(部门号，部门名)。

②职工(职工号，姓名，部门号)。

③项目(项目号，项目名，职工号)。

④领导者(奖金，车型，职工号，部门号，任职时间)。

⑤子女(姓名，出生日期，职工号)。

⑥职工_项目(项目号，职工号，天数)。

7.4.3　关系数据库模式的规范化

关系数据库模式的规范化过程包括两个主要任务。第一个任务是确定范式级别，第二个任务是实施规范化处理。

需要确定的范式级别既包括前面得到的初始数据库模式的范式级别，也包括目标数据库模式的范式级别。

初始数据库模式的范式级别需要根据函数依赖来确定，所以这一步首先需要给出各个关系模式的函数依赖，并求出最小函数依赖集，然后根据 1NF、2NF、3NF 等定义确定每个关系模式的范式。每个关系模式的范式确定之后，整个数据库模式的范式也就确定了。注意，一个数据库模式满足 *x*NF 是指它所包含的所有关系模式都满足 *x*NF。

目标数据库模式的范式级别需要根据实际应用的需要(处理需求)来确定。图 7-1 所示的数据库设计过程指明，逻辑设计阶段的输入有三个：处理需求、ER 模型和 DBMS 特征。其中，处理需求主要在规范化阶段确定目标数据库模式的范式时需要参考。DBMS 特征是在系统规划时确定的，一旦 DBMS 特征确定之后，逻辑设计阶段所针对的结构数据模型也就确定了。例如，DBMS 选择了 Oracle，也就意味着在逻辑设计阶段必须以关系数据库模式为设计目标。一般来说，范式级别的确定是时间效率和模式设计问题之间的权衡。范式级别越高，模式设计问题越少，但连接运算越多，查询效率越低。如果应用对数据只进行查询，没有更新操作，则非 BCNF 范式也不会带来实际影响。但是如果应用对数据的更新操作较频繁，则要考虑高一级范式以避免数据不一致。在实际应用中一般以 3NF 为最高级别范式。因此，一般地，在规范化过程没有特殊情况时都可以以 3NF 为目标范式进行设计。

确定了应用要达到的范式级别后，就可以按照规范化处理过程，将初始数据库模式分解到目标范式。这部分工作在第 6 章中已经有过详细讨论。具体讲，这一步可以用 6.5 节中的几个模式分解算法来将初始的范式转换为目标范式。

7.4.4　模式评价

模式评价的主要任务是检查规范化后的数据库模式是否完全满足用户需求，并确定要修正的部分。

模式评价一般从功能评价和性能评价两个方面进行。

1. 功能评价

检查数据库模式是否支持用户所有的功能要求，例如，数据库模式是否包含用户要存取的所有属性、模式分解时的无损连接有没有满足等。数据库模式功能不满足的一种典型情形是数据库模式中缺少用户应用涉及的部分数据。例如，当为一个手工系统建立一个计算机化的数据库应用系统时，在数据库需求分析阶段往往容易忽略新系统对用户角色和登录信息的获取(因为原手工系统中不存在这些数据)。

2. 性能评价

检查查询响应时间是否满足规定的需求。模式分解通常会引入更多的连接操作，使得一些数据库的查询性能降低。如果查询性能不满足应用要求，要重新考虑模式分解的适当性。另外，数据库模式的性能评价通常可以采取模拟的方法进行，通过模拟应用的数据量和数据操作来分析数据库系统的响应性能。

7.4.5　模式修正

如果模式评价的结果表明当前的数据库模式在功能或性能上不满足设计需求，则需要对已规范化的数据库模式进行修改。

若功能不满足，则可以通过增加关系模式或属性来解决。例如，如果数据库模式中缺少了用户角色和登录信息，则可以增加相应的基本表来满足要求。

若性能不满足，则要考虑增加冗余属性或降低范式(也称为逆规范化)。增加冗余属性在实际数据库设计中是一种常见的提高性能的方法。例如，在一个银行数据库系统中，假设应用经常要求查询当前的存款总额，而银行的账户数目非常多，虽然这一查询通常可以通过账户关系上的聚集查询实现，但因为关系太大，性能较差，此时可以在数据库模式中增加一个冗余属性“当前存款总额”，这样可以快速地完成查询。额外的代价是需要维护数据之间的一致性，例如，可能需要增加触发器来保证当账户关系发生修改时，“当前存款总额”也可以同步更新。逆规范化是指模式合并。一般地，若多个模式具有相同的主码，而应用主要是查询，则可以考虑合并以减少连接开销。

优化性能的另一种方法是对模式进行分解。这里的模式分解与规范化过程的模式分解不同。规范化过程的模式分解的主要目的是减少模式设计问题，而这里的模式分解主要是为了提高性能。

考虑性能优化的模式分解方法有两种：水平分解和垂直分解。水平分解是指将一个关系水平划分为多个关系，每个关系的模式名不同，但属性集相同。通过水平分解，将一个关系的元组划分为多个子集。由于应用往往只需要访问部分元组，因此这种分解方法可以减少不同应用的数据访问量，从而提高性能。水平分解一般有两种形式，一种是根据访问频率来划分元组，另一种是根据应用所访问的数据特性来划分元组。基于访问频率的水平分解的理论依据是帕累托法则(也称为 80/20 法则)。帕累托法则本身是意大利经济学家帕累托提出的一个统计规律，原意是指社会中的大部分财富(80%)掌握在少数人(20%)手里。研究者发现，帕累托法则在许多领域中都有效，例如，图书馆中 80%的读者所借阅的书是图书馆中 20%的图书。在数据库中，帕累托法则可表示为一个关系中 80%的数据访问集中

在 20%的元组上。基于应用访问数据特性的水平分解是指将不同应用所访问的元组集合单独分解出来形成关系。例如，在图 7-16 中，由于不同系的应用系统(如教学管理)通常只访问本系的学生数据，因此可以将每个系的学生元组单独分解出来形成一个关系，从而减少每个系的教学应用的数据访问量，提高整体性能。

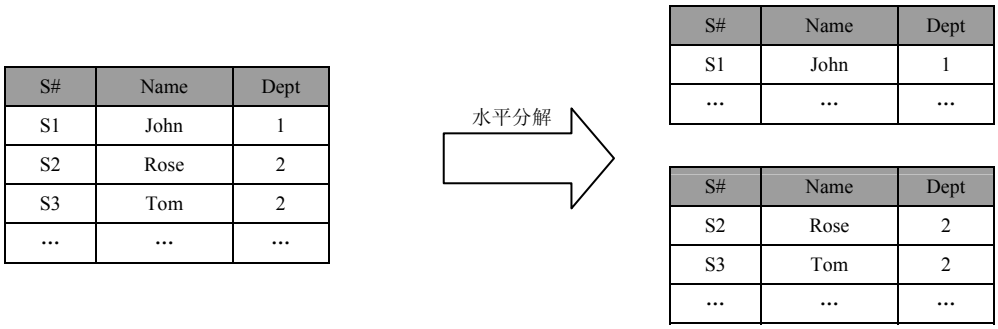


图 7-16　水平分解示例

水平分解从本质上减少了应用处理的数据量(元组数)，因此可以提高查询性能。

垂直分解的出发点与水平分解类似，即希望能够通过分解减少查询涉及的数据量。垂直分解与规范化过程的模式分解在实现方法上类似，但它不以提高范式级别为目的，而以性能优化为目的。垂直分解的依据是应用往往不需要访问一个关系模式的全部属性集，大部分查询只涉及一个关系模式的部分属性集。因此，可以将经常访问的这部分属性集连同主码投影出来形成新的关系模式，并将剩下的属性集以及主码形成另一个关系模式。这样做的目的是使大部分查询可以集中到分解后的某个关系模式上。由于关系模式的属性数目减少了，因此可以减少整个关系的磁盘存储代价，进而提高查询性能。

7.4.6　外模式设计

在逻辑设计阶段还需要针对不同应用模块的访问需求设计相应的外模式。在数据库三级模式结构中，外模式定义了不同用户(应用模块)眼里的数据库逻辑结构，因此应用程序对数据库的访问理论上都应当是访问外模式。

在数据库设计中定义外模式有几个优点。

(1)可以使概念模式名称在应用编程中更符合用户的命名习惯。

在概念设计阶段，由于 ER 模型集成时要消除命名冲突以保证关系和属性名的唯一，因此可能导致概念模式名称不符合某些用户的命名习惯。例如，有的用户习惯用英文单词命名，但概念模式是按汉语拼音缩写来命名的。这种情况会影响用户在前端编写数据库访问程序时的效率。通过外模式，可以重新定义这些模式名称，以符合用户的命名习惯。

(2)可以给不同级别的用户定义不同的外模式，以保证数据的保密性。

例如，假设一个电子商务系统中有概念模式“产品(产品号，产品名，规格，单价，产品成本，产品合格率)”，可以为 Web 用户建立外模式“产品 1(产品号，产品名，规格，单价)”，将产品的成本和合格率信息隐藏；为销售部门建立外模式“产品 2(产品号，名称，规格，单价，成本，合格率)”，以便销售部门了解产品的成本和合格率情况。通过将

Web 用户的访问权限约束在“产品 1”外模式上，可以从理论上避免 Web 用户访问到敏感信息，保证数据的保密性。

(3) 可以简化用户程序对系统的使用。

在实际的数据库应用系统中，可以将某些复杂查询设计为外模式，从而简化前端用户程序的 SQL 编程。

7.5　物 理 设 计

在完成数据库逻辑设计后，需要将数据库逻辑结构映射成物理结构。数据库的物理结构是指数据库在物理设备上的存储结构与存取方法。数据库物理设计依赖于给定的 DBMS，因此设计人员必须充分了解所用 DBMS 的内部特征、存储结构、存取方法。

数据库的物理设计步骤通常分为两步。

(1) 确定数据库的物理结构，主要指存储结构和存取方法。

(2) 对物理结构进行评价，评价的重点是时间效率和空间效率。

1. 确定数据库的物理结构

1) 确定数据库的存储结构

确定数据库存储结构时要综合考虑存取时间、存储空间利用率和维护代价三方面的因素。这三个方面常常是相互矛盾的，例如，消除一切冗余数据虽然能够节约存储空间，但往往会增加存取时间，因此必须进行权衡，选择一个折中方案。

2) 设计数据库的存取方法

在关系数据库中，选择存取方法主要是指确定如何建立索引。例如，主索引应该建在什么属性上，辅助索引应该建在哪些属性上，选择什么类型的索引。不同 DBMS 所支持的索引一般有所不同，例如，Oracle 支持位图索引但不支持散列索引，Microsoft SQL Server 不支持位图索引但支持 B+树索引，所以存取方法设计要根据 DBMS 的特征进行。

索引是影响数据库存取性能的一项重要技术。在数据库物理设计阶段，需要完成下面的索引设计任务。

(1) 哪些表的哪些列需要设计索引：由于索引主要用于加快数据库存取过程，所以索引应当设计在频繁存取的表和列上。

(2) 某个具体的列需要设计什么结构的索引：目前的 DBMS 普遍支持 B+树索引，但不同 DBMS 支持的索引类型往往有所区别。不同的索引结构所支持的查询类型以及适合的负载类型也有所不同。例如，B+树通常适合读多写少的应用，散列索引适合点查询但不适合范围查询。因此，数据库设计者需要了解 DBMS 所支持的不同类型的索引的特点，并结合数据库表和列的实际存取特点，为不同的列选择合适的索引结构。

3) 确定数据的存储位置

为了提高系统性能，应该根据应用情况将数据的易变部分与稳定部分、存取频率较高部分和存取频率较低部分分开存放。此外，表和索引可考虑放在不同的磁盘上，使查询时可以并行读取。日志文件和备份文件由于数据量大，而且只有恢复时使用，可放到磁带上。

4) 确定系统配置

DBMS 产品一般都提供了一些存储分配参数，供设计人员和 DBA 对数据库进行物理优化。初始情况下，系统都为这些变量赋予了合理的缺省值(如并发用户数、同时打开的数据库对象数、缓冲区分配参数等)，但是这些值不一定适合每一种应用环境，在进行物理设计时，需要重新对这些变量赋值以改善系统的性能。

2. 评价物理结构

数据库物理设计过程中需要对时间效率、空间效率、维护代价和各种用户要求进行权衡，其结果可以产生多种方案，数据库设计人员必须对这些方案进行细致的评价，从中选择一个较优的方案作为数据库的物理结构。

评价物理结构的方法完全依赖于所选用的 DBMS，主要从定量估算各种方案的存储空间、存取时间和维护代价入手，对估算结果进行权衡、比较，选择出一个较优的合理的物理结构。如果该结构不符合用户需求，则需要修改设计。

7.6　数据库实施

数据库实施阶段的主要任务是建立实际的数据库结构以及装载初始数据。该阶段的主要工作包括数据库结构定义、建立初始数据库以及编写与调试应用程序。

1. 数据库结构定义

确定了数据库的逻辑结构与物理结构后，就可以用所选用的 DBMS 提供的数据定义语言(DDL)来严格描述数据库结构。例如，用 Create Table 语句建立基本表，用 Create View 建立视图，等等。

2. 建立初始数据库

数据库结构建立好后，就可以向数据库中装载数据了。实际的数据库应用系统一般包含一些初始数据，将这些初始数据入库是数据库实施阶段最主要的工作。对于数据量不是很大的小型系统，可以用人工方法完成数据的入库，其步骤如下。

(1) 筛选数据。需要装入数据库中的数据通常都分散在各个部门的数据文件或原始凭证中，所以首先必须把需要入库的数据筛选出来。

(2) 转换数据格式。筛选出来的需要入库的数据，其格式往往不符合数据库要求，还需要进行转换。这种转换有时可能很复杂。

(3) 输入数据。将转换好的数据输入计算机中。

(4) 校验数据。检查输入的数据是否有误。

对于中大型系统，由于数据量极大，用人工方式组织数据入库将会耗费大量人力物力，而且很难保证数据的正确性。因此应该设计一个数据输入子系统，由计算机辅助数据的入库工作。

3. 编写与调试应用程序

数据库应用程序的设计应该与数据设计并行进行。在数据库实施阶段，当数据库结构建立好后，就可以开始编写与调试数据库的应用程序，也就是说，编写与调试应用程序是与组织数据入库同步进行的。调试应用程序时由于数据入库尚未完成，可先使用模拟数据。

此外，前期设计的存储过程、函数、触发器等过程化 SQL 程序也需要在数据库实施阶段完成编写和调试。

7.7　数据库运行与维护

数据库的运行与维护通常是和整个数据库应用系统一起进行的。数据库投入运行标志着开发任务的基本完成和维护工作的开始，并不意味着设计过程的终结。由于应用环境在不断变化，在数据库运行过程中物理存储也会不断变化，对数据库设计进行评价、调整、修改等维护工作是一个长期的任务，也是设计工作的继续和提高。

在数据库运行阶段，对数据库经常性的维护工作主要是由 DBA 完成的，具体如下。

1. 数据库的备份和恢复

定期对数据库和日志文件进行备份，以保证一旦发生故障，能利用数据库备份及日志文件备份，尽快将数据库恢复到某种一致状态，并尽可能减少对数据库的破坏。数据库的备份和恢复通常借助 DBMS 提供的备份管理器以及恢复管理器来实现。

DBMS 通常支持多种数据库备份策略，常见的有海量备份、增量备份、脱机备份、联机备份等。海量备份是将整个数据库进行备份，增量备份则仅备份自上一次海量备份以来数据库中发生了修改的部分数据，脱机备份指备份过程中不再响应任何用户请求，联机备份则允许在备份过程中继续响应用户请求。具体采用什么样的数据库备份策略，需要数据库设计者根据应用需求决定。例如，对于一个银行信息系统，可以每天 00:00 执行一次海量脱机备份，然后 00:00 之后每隔 1h 执行一次增量联机备份。

2. 数据库的安全性和完整性控制

DBA 必须对数据库安全性和完整性控制负责任。根据用户的实际需要授予用户不同的操作权限。另外，由于应用环境的变化，数据库的完整性约束条件也会变化，需要 DBA 不断修正，以满足用户要求。

3. 数据库性能的监督、分析和改进

目前许多 DBMS 产品都提供了监测系统性能参数的工具，DBA 可以利用这些工具方便地得到系统运行过程中一系列性能参数的值。DBA 应该仔细分析这些数据，通过调整某些参数来进一步改善数据库性能。

4. 数据库的重组织

数据库运行一段时间后，记录的不断增、删、改会使数据库的物理存储变坏，从而降

低数据库存储空间的利用率和数据的存取效率，使数据库的性能下降。这时，DBA 就要对数据库进行重组织或部分重组织(只对频繁增、删的表进行重组织)。

7.8　本 章 小 结

本章主要介绍了数据库设计的概念和基本方法，讨论了规范化的数据库设计过程，并重点讨论其中的概念设计、逻辑设计和物理设计等过程。

通过对本章的学习，读者应了解数据库设计的基本概念和方法，掌握规范化数据库设计的基本过程，并能够熟练运用 CASE 工具针对特定应用需求进行数据库设计。

习　　题

1．试述数据库设计的基本过程。

2．数据库概念设计阶段的主要任务是什么？

3．数据库逻辑设计阶段的主要任务是什么？

4．数据库物理设计阶段的主要任务是什么？

5．某工厂生产多种产品，每种产品要使用多种零件，同一种零件可用在多种产品上。每种零件由一种材料制造，每种材料可用于不同零件的制作。有关产品、零件、材料的数据字段如下。

产品：产品号(GNO)，产品名(GNA)，产品单价(GUP)

零件：零件号(PNO)，零件名(PNA)，单重(UW)，单价(UP)

材料：材料号(MNO)，材料名(MNA)，计量单位(CU)，单价(MUP)

以上各产品需要的各种零件数为 GQTY，各零件需要的材料数为 PQTY。请完成下面的工作。

(1)绘制该应用的 ER 图(实体、属性等名称均使用中文)。

(2)将 ER 模型转换成相应的关系模型(关系模式和属性名称均使用英文)。

第 8 章　数据库应用系统开发

数据库本身并不能建立应用程序，只能完成后台数据存储与管理的功能，因此必须和前端的应用程序结合起来才能进行业务处理。把基于数据库的应用系统(或应用软件)称为数据库应用系统。与其他的应用系统相比，数据库应用系统引入数据库的访问操作，因此在设计和实现过程中必须掌握数据库访问的相关技巧。数据库应用系统在本质上也是软件，因此它的开发过程与一般的软件开发类似。本章的重点放在数据库应用系统的数据库访问编程方法上。由于不同的程序设计语言在数据库访问编程的具体细节上有所不同，因此本章将以 Microsoft Visual Basic 为例阐述数据库访问编程的一般性方法。

内容提要：本章首先简要介绍数据库应用系统开发的整个过程，然后以 Microsoft Visual Basic 和 Microsoft SQL Server 为例讨论数据库访问编程的基本方法和需要注意的一些问题。

8.1　数据库应用系统开发概述

数据库应用系统首先也是一个软件，因此数据库应用系统的开发过程遵循规范化的软件开发模型。但是，由于引入了数据库，数据库应用系统在架构设计上与其他的一些软件(如游戏软件、文字处理软件等)有较大的区别。数据库通常统一存储在数据库服务器上，并且由服务器进行统一管理。这里的服务器不是指物理的计算机服务器硬件，而是软件的概念，一般是指 DBMS(由于 Oracle、Microsoft SQL Server 等都允许在一台计算机上安装多个 DBMS，所以有时也用 DBMS 实例来指代数据库服务器)。因此，要掌握数据库应用系统开发方法，首先必须了解数据库应用系统的架构。

8.1.1　数据库应用系统的架构

数据库应用系统的架构一般指软件体系结构。在数据库应用系统中，可以将所有的业务功能划分为三个方面。

(1)操作界面服务。操作界面服务主要完成数据的输入与显示等业务处理，如输入数据的正确性检查、输出数据的报表显示、图形显示等。

(2)商业服务。商业服务主要完成数据库应用系统中的数据运算以及业务规则处理，如商业规则的检查、对输入数据的加工处理等。

(3)数据服务。数据服务主要完成数据库应用系统中的数据存储与管理功能，如数据的完整性检查、安全性控制等。

根据这三方面的功能在整个架构中位置的不同，数据库应用系统的架构大致可分为两种，即客户端/服务器结构和浏览器/服务器结构。

1. C/S 结构

C/S 结构由客户端和服务器构成，其中服务器指数据库服务器，客户端指完成前端业务处理的应用程序。在 C/S 结构中，客户端可以根据业务处理的要求实时地访问后台的数据库服务器，从而提供对前台数据的增、删、改、查服务。

C/S 结构又有许多种变种，在实际开发中常用的结构主要有前端为主的 C/S 结构和后端为主的 C/S 结构。前端为主的 C/S 结构是指应用系统的三类服务中，操作界面服务和商业服务都在客户端完成，而服务器仅提供数据服务。图 8-1 给出了前端为主的 C/S 结构的示意图。在这种结构中，客户端负担重，服务器负担轻，所以也称为“胖客户端/瘦服务器结构”。

图 8-1 所示的数据库应用系统架构还是目前企业管理信息系统中最常见的体系结构。平常使用的一些程序设计工具(如 Microsoft Visual C++、Visual Basic 等)可以用来开发这种结构的数据库应用系统。

后端为主的 C/S 结构是在前端为主的 C/S 结构的基础上提出来的，可以看成对前端为主的 C/S 结构的一种改进。图 8-2 给出了后端为主的 C/S 结构示意图。可以看到，后端为主的 C/S 结构中，商业服务从客户端迁移到了服务器，因此数据库服务器不仅承担了数据服务，还承担了商业服务。这种结构减轻了客户端的计算负担，增加了服务器的处理任务，所以也称为“瘦客户端/胖服务器结构”。本身数据库服务器并不提供业务处理的能力，所以后端为主的 C/S 结构对 DBMS 有一定的要求，即 DBMS 本身要具备基本的业务处理编程能力，只有这样才能将商业服务迁移到服务器。在现在的 DBMS 中，这一要求主要通过过程化 SQL 来满足。也就是说，可以将商业服务用过程化 SQL 实现成存储过程，然后由客户端调用。实际开发中，一般也用这种方式来建立后端为主的数据库应用系统架构。

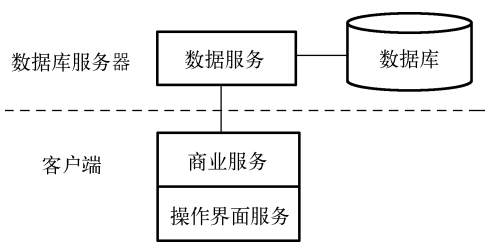


图 8-1　前端为主的 C/S 结构

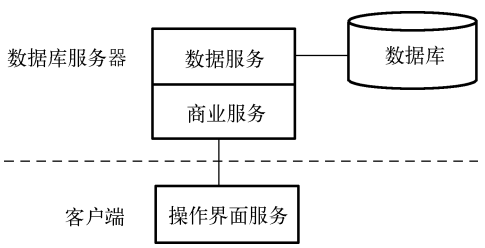


图 8-2　后端为主的 C/S 结构

2. B/S 结构

C/S 结构的主要问题是系统的可维护性差。这是因为在 C/S 结构中，操作界面服务和商业服务通常在客户端上运行，因此一旦系统需要升级，就需要对所有客户端进行更新。在银行、证券、邮电等分布式应用系统中，这种维护性任务的工作量巨大，给系统的升级工作带来了很大的困难。

B/S 结构正是在这样的背景下出现的。当然，B/S 结构出现的基础是互联网和 WWW 服务的出现。在 WWW 服务中，客户端(即浏览器)提供了一个统一的显示和操作界面，它

可以将 Web 服务器上的 HTML 界面动态下载到客户端本地运行。这种方式最大的优点就是可维护性好——如果需要更新界面，只需要在 Web 服务器上将界面的内容更新，所有的客户端都可以自动获取到最新的界面。因此，B/S 结构非常适合那些地域性分布的应用(当然这种架构也有安全性等方面的问题，不展开讨论)。

图 8-3 给出了数据库应用系统的 B/S 结构。这种结构包含了客户端、Web 服务器和数据库服务器三层，所以也称为“三层结构”。如图 8-3 所示，在 B/S 结构中，客户端只提供唯一的浏览器显示功能(这也可以看成操作界面服务的一部分)，商业服务和操作界面服务则通常放在 Web 服务器上。从数据库服务器的角度看，Web 服务器就是它的客户端，所以 Web 服务器和数据库服务器在许多应用中通常是基于 C/S 结构搭建的。因此，在 Web 服务器和数据库服务器这边，仍可以有多种方式来安排商业服务的位置，例如，可以将商业服务迁移到数据库服务器上。

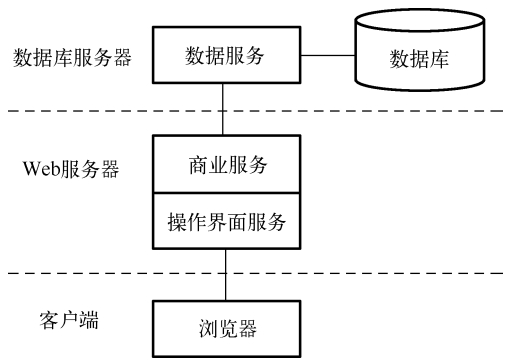


图 8-3　B/S 结构

B/S 结构的数据库应用系统需要用专门的 Web 开发工具开发，如 ASP/ASP.NET 或者 JSP 等。一般的 Web 开发工具都提供数据库访问功能，都可以用来实现 B/S 结构的数据库应用系统。而一些传统的开发工具，如 VC++、VB、PowerBuilder 等，都是针对 C/S 结构的数据库应用系统开发而设计的，因此不能直接用来开发 B/S 结构的数据库应用系统。

C/S 结构和 B/S 结构都各有优缺点，相应的开发工具也各有所长。应用系统应根据各自的需求来决定建立什么样结构的系统。C/S 结构和 B/S 结构后来还产生了许多变种，如三层的 C/S 结构、多层的 B/S 结构等，也有 C/S 和 B/S 的混合实现结构。在这一节只讨论了最常用的结构，其他的结构可以参考一些软件工程方面的书籍。

8.1.2　数据库应用系统的开发过程

在进行数据库应用系统的开发时，一般使用结构化方法(生命周期法)。结构化方法是先对问题进行全面、细致的调查，然后从功能与流程的角度来分析和优化问题，最后设计和实现系统。它的核心思想是结构化的分析、设计与编程，特点是强调自顶向下设计、流程化、文档化。结构化方法一般通过数据流程图分析、模块化技术、结构化程序技术来实现。

图 8-4 给出了基于结构化方法的数据库应用系统开发过程。下面简要介绍每个过程，更详细的内容可参考软件工程的书籍。

1. 可行性分析

可行性分析是研究对于提出的系统开发需求是否存在可行解，以及是否值得去做。可行性分析一般要分析系统开发的经济可行性、技术可行性以及操作可行性，基本的步骤如下。

(1) 复查系统规模和目标。根据系统的开发目标，访问关键人员，改正含糊的、二义的描述以及不正确的描述，核查系统限制和约束。

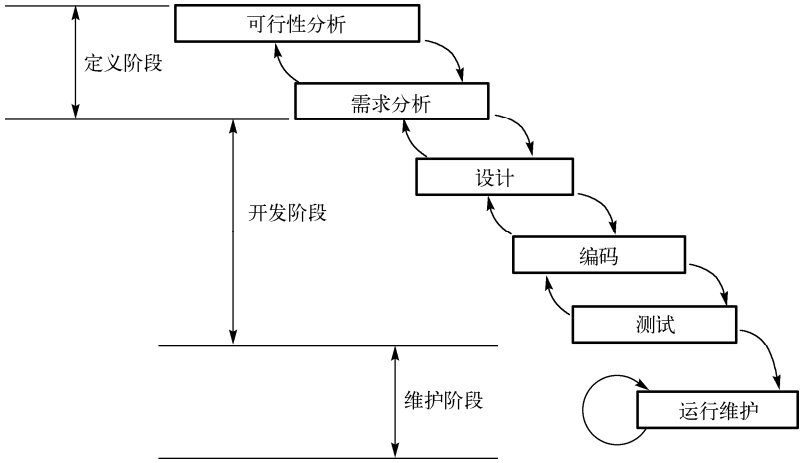


图 8-4　基于结构化方法的数据库应用系统开发过程

(2)研究现有系统功能。分析现有系统的任务和功能，比较新旧系统，例如，新系统必须实现旧系统的基本功能，新系统必须解决旧系统存在的问题，新系统必须比旧系统增收入和减支出。

(3)导出新系统模型。定义新系统的逻辑模型，明确新系统的功能需求和其他指标。

(4)重新定义问题。复查问题定义、规模和目标，根据新系统逻辑模型重新定义问题。这些问题有可能是由于系统分析员误解而产生的，也有可能是之前遗漏的。重新定义问题一般需要循环执行(定义、分析、求解、重定义)，最终明确新系统的问题定义。

(5)导出和分析各种可选解决方案。根据新系统的逻辑模型，从不同角度导出不同的物理模型(物理实现方案)，并分析每一种方案的经济可行性、技术可行性和操作可行性，去掉那些经济上不合算、用户没有能力操作、技术上实现不了的方案。最后为可行的方案制定进度计划。

(6)推荐方案。得出可行性分析的结论——终止还是继续开发。如果继续开发，则对推荐方案进行成本/效益分析。

(7)制定开发计划。为推荐方案制定开发计划，包括进度安排、开发人员、硬件设备、软件工具、各阶段成本估计等。

(8)书写文档，提交审查。提交可行性分析报告，总结各阶段的任务和结果，给出推荐方案及可行性分析结果，描述开发计划等。

2. 需求分析

软件系统的需求是以一种清晰、简洁、一致且无二义性的方式，对一个待开发系统中各个有意义的方面进行陈述的一个集合。常见的需求有功能需求、数据需求、性能需求、环境需求、可靠性需求、安全保密要求、用户界面需求、资源使用需求、成本消耗需求、开发进度需求等，其中最重要的是功能需求、数据需求和性能需求。

需求分析通常包括需求获取和需求规格说明两部分内容，最终的目标是形成软件系统的需求规格说明书。

需求分析常见的工具有数据流程图和 ER 模型。数据流程图(Data Flow Diagram，DFD)

用于分析软件系统的数据流，即数据在整个系统中的流动和处理过程。数据流分析的目的是建立软件系统的功能模型，从而给出系统功能需求的规格说明。ER 模型采用 ER 图的方式分析系统中的数据需求。对于数据库应用系统而言，数据流分析的最终目标是明确系统的数据处理过程，从而导出系统的功能模块结构，而 ER 分析的目标是明确系统的数据需求，最终导出数据库的逻辑结构和物理结构。

3. 设计

设计阶段一般划分为概要设计和详细设计两个阶段。其中概要设计阶段主要完成软件系统的体系结构(功能模块结构)设计、处理流程设计、数据库设计、接口设计等工作，详细设计阶段主要给出每个模块具体的输入/输出、程序流程、数据结构、约束等内容，为模块的编码奠定基础。

软件的功能模块结构一般从需求分析的数据流程图中导出，最终建立层次结构的功能模块。模块之间的关系一般通过控制结构图来分析。模块设计的主要指标是高内聚、低耦合，强调模块的高度封装和独立性。

处理流程设计是指多个模块组合响应系统需求的工作过程。一般地，在需求分析阶段定义的系统功能需求需要借助多个模块的功能才能满足，而处理流程设计给出了针对不同功能需求的模块组合策略和运行流程。

数据库设计是概要设计中的重要内容之一。数据库设计的基础是需求分析阶段得到的 ER 模型(在第 7 章中，ER 模型是数据库概念设计的结果)。数据库设计的过程可以按照第 7 章讨论的规范化数据库设计方法进行，将 ER 模型转换为关系数据模型，然后规范化，一直到建立数据库物理结构。

接口设计主要包括内部接口设计和外部接口设计。内部接口是指模块之间的接口关系，如数据库交互、共享文件交互，等等。外部接口是指系统与外部用户或其他系统之间的接口关系，如外部数据采集接口、输出接口等。

详细设计的主要任务是精确描述每个模块的程序逻辑。详细设计阶段建立了程序设计的蓝本，程序员可以据此进行实际编码。详细设计描述一般要给出每个模块的输入/输出参数，以及其涉及的数据结构、程序流程、出错处理、边界约束等信息，以便使程序员在编码时能够充分明确模块的处理过程。其中最重要的是程序流程。设计程序流程时常见的工具有程序流程图、N-S 图、程序描述语言(伪码)等。在实际开发中，可以根据不同系统的特点选择不同的描述方法。

4. 编码

编码阶段主要的任务是完成详细设计阶段各个模块的编程实现任务，包括人机界面设计和程序编码工作。人机界面设计一般须遵循三条基本原则：置用户于控制之下、减少用户的记忆负担、保持界面一致。程序编码的基本要求是逻辑清楚、清晰易读。

软件系统开发所用的程序设计语言一般要根据自己的特点和需要选择，主要应考虑的因素如下。

(1) 软件的应用领域。

(2) 系统用户的要求。

(3) 可以使用的编译程序。

(4) 可以得到的软件工具。

(5) 工程规模。

(6) 程序员的知识。

(7) 软件可移植性。

下面是常见的一些程序设计语言与其所适用的领域。

(1) C/C++语言：适合系统底层实现以及实时应用。

(2) Fortran：工程领域。

(3) Prolog 和 Lisp：人工智能领域。

(4) Delphi、VB、C#、Python：适合 MIS 应用开发。

(5) VC：适合信息处理与控制等应用开发。

(6) Java：适合平台无关的应用。

(7) JSP、ASP：适合 Web 应用。

5. 测试

软件测试是软件系统开发过程中非常重要的一个步骤。测试是程序的执行过程，目的在于发现错误。一个好的测试用例能发现至今未发现的错误，一个测试的成功在于发现了至今未发现的错误。关于软件测试，必须清楚以下几点。

(1) 软件测试目的是以最少的时间和人力，系统地找出软件中潜在的各种错误。如果成功地进行了测试，就能够发现软件中的错误。

(2) 软件测试的附带收获是，它能够证明软件的功能和性能与需求说明基本符合。

(3) 进行测试时收集到的测试结果数据为可靠性分析提供了依据。

(4) 测试不能说明软件中不存在错误，它只能说明软件中存在错误。

(5) 最严重的错误(从用户角度)是那些导致软件无法满足需求的错误。程序中的问题可能在开发前期的各阶段解决，纠正错误也必须追溯到前期工作。

(6) 软件测试不等于程序测试，软件测试应贯穿于软件定义与开发的整个期间，并且在概要设计阶段就要完成软件测试计划的编写。

软件测试过程一般分为三个阶段。

1) 单元测试(模块测试)

单元测试又称为模块测试，是针对软件设计的最小单位——程序模块进行正确性检验的测试工作。其目的在于发现各模块内部可能存在的各种差错。

单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试，以检验每个模块能否单独工作。

2) 集成测试

在单元测试的基础上，需要将所有模块按照设计要求组装成系统，发现并排除在模块连接中可能出现的问题，最终构成满足要求的软件系统。集成测试需要考虑多方面的问题，例如：

(1) 在把各个模块连接起来的时候，穿越模块接口的数据是否会丢失。

(2) 一个模块的功能是否会对另一个模块的功能产生不利的影响。

(3) 各个子功能组合起来，能否达到预期要求的父功能。

(4) 全局数据结构是否有问题。

(5) 单个模块的误差累积起来，是否会放大，从而达到不能接受的程度。

3) 确认测试

在模拟的环境(可能就是开发的环境)下，验证集成后的被测软件是否满足需求规格说明书列出的需求。

软件测试的基本方法有白盒法和黑盒法。白盒测试(White-Box Testing)也称为玻璃盒法(Glass-Box Testing)，是指测试者完全知道程序的内部结构和处理算法，而黑盒法(Black-Box Testing)是指测试者完全不知道程序的内部结构和处理算法。一般地，单元测试采用白盒法，而集成测试和确认测试采用黑盒法。

6. 运行维护

软件测试通过后就开始试运行，并进入维护阶段。软件维护是指在软件已经交付使用之后，为了改正错误或满足新的需要而修改软件的过程。软件维护包括三种类型的维护工作：改正性维护、适应性维护、完善性维护。

1) 改正性维护

为了识别和纠正软件错误、修复软件性能上的缺陷、排除实施中软件的误使用而进行的诊断和改正错误的过程叫作改正性维护。

在软件交付使用后，因开发时测试得不彻底、不完全，必然会有部分隐藏的错误遗留到运行阶段。这些隐藏下来的错误在某些特定的使用环境下就会暴露出来。

2) 适应性维护

为使软件适应外部环境或数据环境变化而进行的修改软件的过程叫作适应性维护。适应性维护一般是由于外部环境(新的硬、软件配置)变化或者数据环境(数据库、数据格式、数据输入/输出方式、数据存储介质)变化而引起的。

3) 完善性维护

为了满足用户提出的新的功能与性能要求，需要修改或再开发软件，以扩充软件功能、增强软件性能、改进加工效率、提高软件的可维护性，这种情况下进行的维护活动叫作完善性维护。

8.2　数据库访问方法

本节以 Microsoft Visual Basic 为例，介绍数据库访问的基本方法。虽然不同的前端开发语言在数据库访问的具体技术上有所差异，但总的方法和过程是类似的。因此，本节给出的数据库访问方法可作为一般的数据库访问编程的参考。

8.2.1　VB 概述

VB 是 Microsoft Visual Studio 编程套件中的标准工具之一，是 Windows 标准程序的快速开发工具。VB 以工程的方式组织应用程序开发，一个 VB 工程以扩展名为.vbp 的文件为入口，包括下面的基本组成：

(1) 窗体(包括控件)；

(2) 窗体文件(.frm 文件)，一个窗体对应一个窗体文件；

(3) 模块文件(.bas 文件)，是所有窗体公用的程序集合。

VB 工程的一般开发过程如图 8-5 所示。首先分析问题，根据软件的详细设计结果确定系统的界面设计和处理过程。然后根据界面设计的要求设计窗体并确定窗体上的控件。接着确定窗体与控件的属性。VB 窗体的设计采用所见即所得的方式，因此对窗体和控件的属性修改可以直接反映到屏幕上。窗体设计完成后，接下来就是主要的编码工作。VB 程序采用的是事件驱动的编码方式，即程序员通过为窗体、控件等编写特定的事件处理程序来响应界面操作，从而完成相应的处理任务。因此，在事件过程代码编写时，首先要确定应用程序需要响应哪些事件，然后为每一个事件编写相应的响应代码。最后，如果多个窗体存在着一些共性的处理需求，则可以增加一个或多个模块文件。模块文件是一系列过程和函数的集合，其中的过程和函数可以被全部的窗体文件所调用。通过适当地增加模块文件，可以简化窗体文件中的代码，提高程序的清晰度。

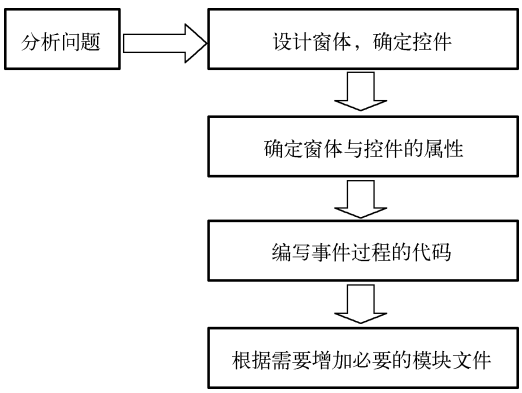


图 8-5　VB 工程的一般开发过程

图 8-6 给出了一个 VB 窗体和控件的例子。该窗体名为 Form1，包含了一个名为 Label1 的 Label 控件(Label 控件所显示的文字在运行时是不可编辑的，一般用来显示一些说明性文字)，两个 Command 控件(Command 控件一般作为动作按钮)，名称分别为 Command1 和 Command2。

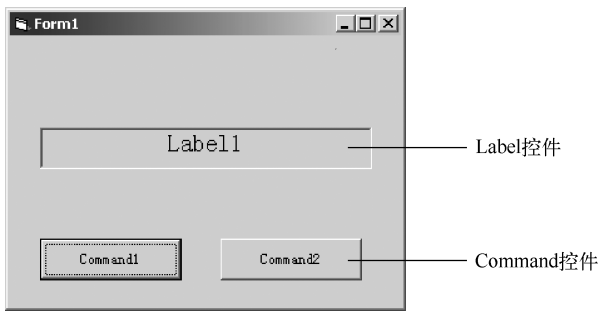


图 8-6　VB 窗体和控件示例

VB 窗体和控件都是具有自己的属性、方法和事件的对象。可以把属性看作一个对象的性质，把方法看作对象的动作，把事件看作对象的响应。VB 程序主要是针对窗体或控件进行的编程。程序员根据应用系统的需求，决定更改窗体和控件的哪些属性、调用哪些方法、对哪些事件做出响应，从而得到希望的外观和行为。

例如，假设希望在图 8-6 窗体的 Command1 按钮上单击时，在 Label 控件上显示“欢迎使用 Visual Basic！”，当单击 Command2 按钮时，在 Label 控件上显示当前时间，即得到图 8-7 所示的效果。

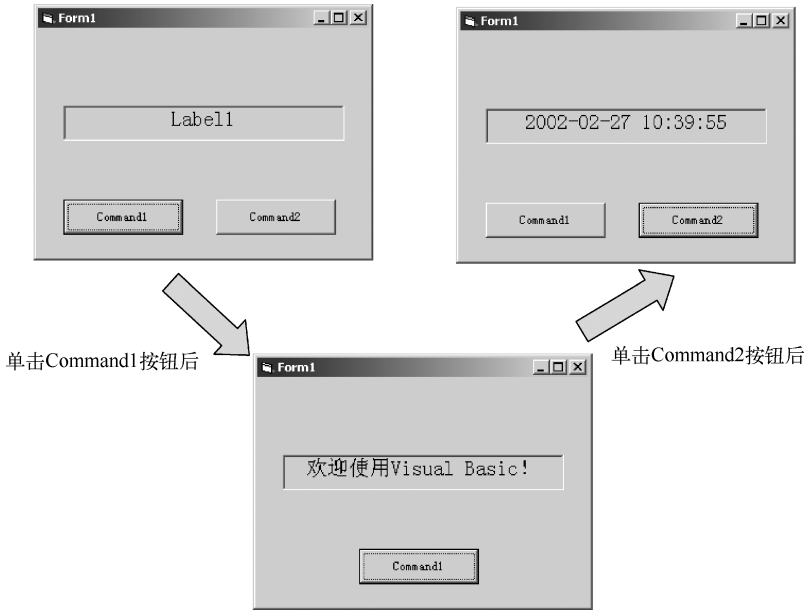


图 8-7　VB 窗体编程示例

为了实现这一效果，首先需要确定对 Command1 和 Command2 上的 Click 事件进行响应，因为 Click 事件代表了 Command 控件上的单击操作。然后，在 Click 事件过程的代码中去修改 Label1 的 Caption 属性。窗体上所有事件过程的代码，包括窗体内控件的事件过程代码，都包含在窗体文件中。在本例中，所有代码都包含在 Form1.frm 文件中。下面给出了 Form1.frm 文件的代码：

```
'Form1.frm 中的代码
Private Sub Command1_Click()  'Click 事件
   Label1.Caption = "欢迎使用 Visual Basic! "
End Sub

Private Sub Command2_Click()
   Label1.Caption = GetTime()
End Sub
```

在上面的示例代码中，Form1.frm 文件包含了两个事件过程，即 Command1_Click()事件和 Command2_Click()事件。其中 Command1_Click()中的语句将 Label1 的 Caption(标题)修改成了“欢迎使用 Visual Basic！”，Command2_Click()中的语句将 Label1 的 Caption 赋予函数 GetTime()的返回值。GetTime 是自定义的一个函数，可以将此类希望被所有窗体共享的函数(如时间处理函数、字符转换函数等)放在一个单独的模块文件中。在本例中，增加了一个模块文件 Module1.bas，并将 GetTime()的源代码放在该模块文件中，如下：

```
'Module1.bas 中的代码
Function GetTime() As String
   GetTime = Format(Now, "yyyy-mm-dd hh:nn:ss")
End Function
```

该模块文件中只包含了一个自定义的函数 GetTime()，在该函数中，获取系统的当前日期并将它格式化后返回。

8.2.2　通过 ADO 数据控件访问数据库

VB 应用程序访问数据库的基本方法有两种：一种是通过 ADO 数据控件；另一种是通过 ADO/DAO/ODBC 等数据库访问中间件。其中，通过 ADO/DAO/ODBC 等数据库访问中间件是常用的数据库访问方法(图 8-8)，它通过一个对象集合(ADO/DAO)或者一个 API 集合(ODBC)来访问数据库。通过 ADO 数据控件的方法一般用于实现数据查询与显示，但不能实现所有的数据库访问功能。DAO 和 ODBC 是早期的数据库访问方法，目前 ADO 是流行的数据库访问方法，所以本节主要介绍 ADO 数据访问方法。

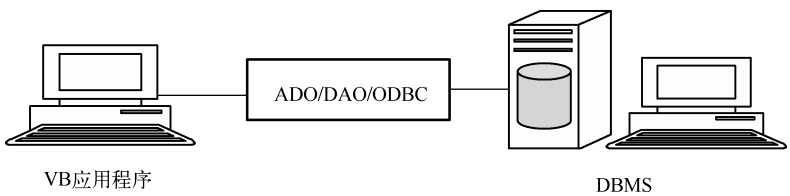


图 8-8　在 VB 中通过 ADO 访问数据库示意图

图 8-9 显示了 VB 中用于数据库访问的两类控件：ADO 数据控件和数据绑定控件。ADO 数据控件(ADO Data Control)是 VB 中用于数据库访问的控件。借助 ADO 数据控件，可以快速建立数据库应用程序。ADO 数据控件一般适用于数据查询与显示，并且不能调用存储过程。

要使用 ADO 数据控件，需要先在 VB 集成开发环境中选择“工程”→“部件”→Microsoft ADO Data Control 6.0 (OLE DB)菜单项，然后在工具箱中就会出现 ADO 数据控件的图标，在窗体设计时可以将其加到窗体上。

ADO 数据控件的主要属性如下。

(1)ConnectString：用于建立客户端到数据库服务器的连接，可以在设计状态下通过向导生成相应的连接字符串。

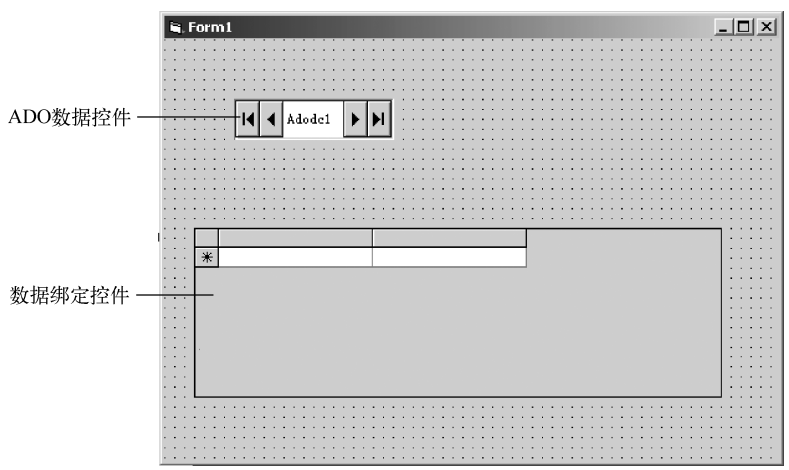


图 8-9　VB 中用于数据库访问的控件

(2) RecordSource：指定数据源，可以是一个基本表、视图或 SQL 查询。

(3) RecordSet：代表了 ADO 数据控件所表示的记录集合。

ADO 数据控件在 VB 程序中代表了一个数据源(一般是一个基本表)。对于其他控件来说，ADO 数据控件就相当于数据库的代理。要想使用 ADO 数据控件所代表的数据，必须要在窗体中再加入若干数据绑定控件，并且将 ADO 数据控件与数据绑定控件绑定，从而使得在程序运行时数据绑定控件可以实时地获取 ADO 数据控件所代表的数据，并显示在窗体上。数据绑定控件与 ADO 数据控件绑定的方法非常简单，只需要将数据绑定控件的 Datasource 属性设置为 ADO 数据控件的名称即可。在 VB 中，有很多类型的控件都可以作为数据绑定控件，一个控件是否可以作为数据绑定控件取决于它是否有 Datasource 属性。

8.2.3　通过 ADO 对象访问数据库

ADO 模型提供了一系列的对象和集合来实现数据库访问。图 8-10 给出了 ADO 中的对象与集合，总共有六个对象(Connection、Command、Recordset、Error、Parameter、Field)和三个集合(Errors、Parameters、Fields)。其中，常用的对象有 Connection、Command 和 Recordset 对象。

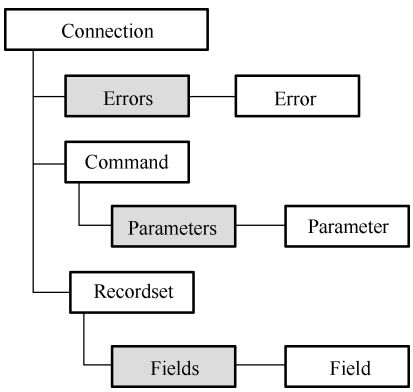


图 8-10　ADO 中的对象与集合

1．Connection 对象

Connection 对象代表与数据源的连接。在 VB 中，可以用下面的语句定义 Connection 对象：

```
Dim cnn as New ADODB.Connection
```

Connection 对象唯一常用的属性是 ConnectString，即连接字符串，它存储了 Connection 对象与数据库服务器的连接信息。要建立客户端与数据库服务器的连接，必须给出正确的 ConnectString。

Connection 对象常用的方法有 Open、Close 和 Execute。其中 Open 方法打开一个数据库连接，Close 方法关闭数据库连接，Execute 方法可以执行一条 SQL 语句或者一个存储过程。但在实际数据库访问中，很少用 Connection 对象的 Execute 方法来执行 SQL 语句或者存储过程，所以只要掌握 Open 和 Close 方法即可。

下面给出了使用 Connection 对象连接 Microsoft SQL Server 数据库的例子。对于其他的 DBMS(如 Oracle 和 MySQL)，Connection 对象使用时主要的差别在于 Connectstring 的内容不同。一般情况下，Connection 对象的使用方法为：定义一个 Connection 对象，为 ConnectString 赋值，调用 Open 方法打开数据库连接，数据库操作完毕后用 Close 方法关闭数据库连接。

```
Dim cnn as New ADODB.Commecntion
Cnn.Connectstring= "Provider=SQLOLEDB.1;Persist Security Info=False;
    User ID=sa;Initial Catalog= sample_db;Data Source=JPQ"
Cnn.Open
```

```
...
Cnn.Close
```

2. Command 对象

Command 对象定义了将对数据源执行的指定命令。使用 Command 对象可以执行一条 SQL 查询语句并返回 Recordset 对象中的记录，也可以执行一个特定的存储过程。由于在 ADO 数据库访问中，对于记录集的操作一般用 Recordset 来完成，所以 Command 对象一般用来调用存储过程。

Command 对象的常用属性有 ActiveConnection 和 CommandText，其中 ActiveConnection 指向一个打开的 Connection 对象，表示 Command 对象所基于的数据库连接信息，CommandText 给出了 SQL 命令，可以是一条 SQL 查询语句，也可以是一个存储过程的名称。

Command 对象的常用方法是 Execute，调用时执行 ComandText 中指定的命令。下面给出了 Command 对象的一个例子。在该例子中，Command 对象执行了一条 SQL 查询语句并将结果返回给一个 Recordset 对象。

```
Dim cmm as New ADODB.Command
Dim rst as New ADODB.Recordset
Cmm.ActiveConnection=cnn
cmm.CommandText="Select * From a Where b<>'asas'"
Set rst=cmm.Execute()
```

Command 对象执行存储过程的内容可参考 5.7.6 节，此处不再赘述。

3. Recordset 对象

Recordset 对象表示的是来自基本表或命令执行结果的记录集。Recordset 对象是 ADO 数据库访问中最常使用的对象，可以操作几乎所有的数据库数据。

Recordset 对象的常用属性如下。

(1) BOF 和 EOF：BOF 表示当前 Recordset 指针指向了记录集的头部，EOF 表示当前 Recordset 指针指向了记录集的尾部。这两个属性一般用来判断记录集是否为空——如果 Recordset 的 BOF 和 EOF 都为 TRUE，则说明该记录集为空。

(2) Source：表示 Recordset 对象所基于的基本表或 SQL 语句。

(3) CursorType：游标类型，一般使用 adOpenKeyset（仅修改可见）或 adOpenDynamic（全部可见）。

(4) LockType：指示编辑过程中对记录使用的锁定类型，一般 adLockOptimistic 表示仅在 Update 时锁定。

(5) Recordcount：记录总数，是指从记录集头部到当前 Recordset 指针位置的记录数，因此如果要获得一个记录集的记录总数，必须将 Recordset 指针移到记录集尾部，再访问 Recordcount 值。

Recordset 对象的常用方法如下。

(1) Open：打开记录集。

(2) Close：关闭记录集。

(3) Addnew：增加一条新记录。

(4) Update：更新记录集。

(5) Delete：删除 Recordset 当前指向的记录。

(6) Movefirst、Movenext、Movelast、Moveprevious：移动 Recordset 指针。

(7) Requery：重新执行 Source 中给出的 SQL 语句，以重新获得记录集。

Recordset 对象可以完成对数据库记录的增、删、改、查等全部操作。一般的数据库操作过程如下。

(1) 创建 Connection 并 Open。

(2) 基于 Open 的 Connection 打开 Recordset。

(3) 使用 Recordset 的 Addnew、Update、Delete、Move 等方法对数据进行增、删、改操作。

(4) 如果要执行查询，则修改 Recordset 的 Source 属性，然后 Open 即可。

Recordset 对象的 Open 方法有两种用法：一种是使用已经打开的 Connection 对象；另一种是直接使用连接字符串。例如，假设 cnn 是一个已经打开的 Connection 对象，则下面的语句使用 cnn 打开了 Recordset 对象 rst：

```
'基于已有的 Connection 的 Open
rst.Open "Employees", cnn, adUseClient, adOpenKeyset, adCmdTable
```

如果不使用已经打开的 Connection 对象，则只需要将上面语句中的 cnn 换成一个连接字符串(ConnectString)即可，格式与 Connection 对象的 ConnectString 属性相同。

8.3　数据库记录操作的具体实现

绝大多数的数据库应用系统中，数据库操作是指记录的增、删、改、查等操作，因为在用户眼里数据库就是记录的集合，而且现实世界中数据操作也是以记录为单位进行的。因此掌握数据库中记录的增、删、改、查方法就意味着基本掌握了数据库访问的方法。由于不同的程序设计语言实现数据库访问的方式有所不同，因此本节以 Microsoft Visual Basic 6.0 为例，结合 Microsoft ADO 模型以及 VB 的 ADO 数据控件，介绍数据库记录操作的基本方法和技巧。

8.3.1　记录插入

记录的插入一般使用 Recordset 对象的 Addnew 方法。下面的代码给出了记录插入的实现过程：

```
Dim cnn as New ADODB.Connection
cnn.Connectstring=...
cnn.Cursorlocation=adUseClient
cnn.Open
Dim rst as New ADODB.Recordset
rst.Open "Employee", cnn, adUseClient, adOpenKeyset, adCmdTable
rst.Addnew
```

```
rst.Fields("Name")=txtName.Text
...
rst.Update
rst.Close
```

记录插入的一般性过程可归纳为以下几个步骤：

(1)打开数据库连接。

(2)使用已经打开的数据库连接打开 Recordset 对象，一般是一个基本表。

(3)调用 Recordset 对象的 Addnew 方法。

(4)将值赋给 Recordset 对象的各个字段。

(5)调用 Recordset 对象的 Update 方法将数据更新到数据库中。

(6)关闭 Recordset 对象。

以上过程代表了使用 ADO 对象在数据库中插入记录的基本过程。其他程序设计语言使用 ADO 插入记录的过程与此类似。因此，上面的过程可以作为基于 ADO 对象的记录插入的常规方法。

8.3.2　记录删除

在数据库应用系统中，记录通常是一条一条删除的，例如，删除学生记录，删除课程记录，等等。一般每次只删除一条记录。

进行记录删除操作时，首先需要在 ADO 中定位要删除的记录，然后调用 Recordset 的 Delete 方法完成删除。

例如，假设在界面上有一个文本框(名称：txtID)，当用户在文本框中输入一个员工号时，将该员工从数据库中删除。下面的代码给出了大致的执行过程。

```
Dim cnn as New ADODB.Connection
cnn.Connectstring=…
cnn.Cursorlocation=adUseClient
cnn.Open
Dim rst as New ADODB.Recordset
rst.Open "Select * From Employee Where EmployeeID='"& txtID.text &"'",
    cnn, adUseClient, adOpenKeyset, adCmdText
If NOT(rst.EOF and rst.BOF) then
rst.Delete
Else
Msgbox “记录不存在”
End If
rst.Close
```

记录删除的一般性过程可归纳为以下几个步骤：

(1)打开数据库连接。

(2)使用 Recordset 的 Open 方法根据给出的删除条件创建一个记录集，该记录集包含了要删除的记录。

(3)检查记录集是否为空(因为一般一次只删除一条记录)。

(4)调用 Recordset 的 Delete 方法删除记录。

(5)关闭 Recordset 对象。

以上过程可以作为基于 ADO 对象的记录删除的一般性方法。

8.3.3　记录修改

在实际应用程序中，一般每次只能修改一条记录，例如，在教学系统中，学生登录后只能修改自己的信息。记录修改通过 Recordset 的 Update 方法来实现。Update 方法将当前 Recordset 的修改更新到数据库中。在执行 Update 方法之前，必须要定位希望修改的记录，并且完成相应的字段修改操作。

下面给出了一个记录修改的例子。在这个例子中，假设要把 EmployeeID 值为 100 的记录的 Name 修改为 aaa，Salary 修改为 2000。

```
Dim cnn as New ADODB.Connection
cnn.Connectstring=…
cnn.Cursorlocation=adUseClient
cnn.Open
Dim rst as New ADODB.Recordset
rst.Open "Select * From Employee Where EmployeeID='100'", cnn, adUseClient,
    adOpenKeyset, adCmdText
rst.Fileds("Name")="aaa"
rst.fields("Salary")=2000
rst.Update
rst.Close
```

记录修改的基本过程可归纳为：

(1)打开数据库连接。

(2)使用 Recordset 的 Open 方法定位要修改的记录(即创建最多只有一条记录的记录集)。

(3)将新值赋予字段，如果有多个字段要修改，则每个字段都要赋值。

(4)调用 Recordset 的 Update 方法。

(5)关闭 Recordset 对象。

以上过程可以作为基于 ADO 对象的记录修改的一般性方法。

8.3.4　记录查询

记录查询通常伴随着查询结果显示的需求，因此在 VB 应用程序中，记录查询操作一般借助 ADO 数据控件来实现，基本的思路是在程序中动态地改变 ADO 数据控件的 RecordSource 属性来动态生成查询结果记录集，并且在使用数据网格(Datagrid)之类的表格时，通过数据绑定控件来显示 ADO 数据控件所生成的最新记录集(即查询结果)。

例如，假设要根据用户输入的一个员工号(用一个名为 txtID 的文本框来获取输入值)来查询相应的员工信息并显示在窗体上，设窗体中的 ADO 数据控件名为 adcEmployee，并且有一个 Datagrid 控件绑定到了 adcEmployee 上，则下面的代码实现了当用户输入一个员工号时自动查询记录并将记录显示在 Datagrid 上的功能。

```
Dim strSQL as string
strSQL="Select * From Employee Where EmployeeID='"& txtID &"'"
```

```
adcEmployee.Recordsource=strSQL   '更新 Recordsource 以生成新的记录集
adcEmployee.Refresh
If adcEmployee.Recordset.BOF and adcEmployee.Recordset.EOF then
    Msgbox "无匹配记录"
End If
```

8.4　本 章 小 结

本章主要介绍了数据库应用系统的基本架构以及开发过程，并以 Microsoft Visual Basic 和 ADO 为例讨论了数据库访问编程的基本方法与技巧，重点介绍了记录操作的编程实现方法。

通过对本章的学习，读者应对数据库应用系统开发的整个过程有大致的了解，掌握数据库访问编程的基本方法，并能够运用 ADO 技术实现对数据库的基本操作。

习　　题

1．简要描述数据库应用系统的开发过程。

2．在数据库应用系统中，C/S 结构和 B/S 结构各有什么优缺点？

3．ADO 数据访问包含哪些对象和集合？

4．简要描述使用 ADO 对象插入记录的过程。

5．简要描述使用 ADO 对象修改记录的过程。

6．如果不使用 ADO 数据控件，直接使用 ADO 对象能否实现记录查询和结果显示功能？如果能，请简要说明实现的过程；如果不能，请说明理由。

第 9 章　数据库事务

在前面的几章中，重点讨论了如何使用数据库、如何设计数据库等问题，但是在实际应用中，数据库的一致性和安全性是一个非常关键的问题。在信息化时代，数据是企业的核心资产，如何保证数据库中的数据正确、可靠是决定数据库技术实用性的一个重要因素。现代 DBMS 的数据一致性保护技术是以事务为基础的，因此本章将重点围绕数据库应用系统中的数据一致性问题，讨论事务的相关概念和技术。

内容提要：本章首先简要介绍事务的概念，然后重点讨论事务的性质、事务的状态、事务的原语操作，最后讨论数据库一致性问题。

9.1　事务的概念

数据库一致性维护的基础是事务理论。事务是一个不可分的操作序列，也是 DBMS 执行和调度的基本单位。在目前的 DBMS 中，前端用户给出的 SQL 语句最终都会转换为事务的执行序列。在 DBMS 中进行事务管理的部件称为事务管理器(Transaction Manager)。事务管理器接收用户给出的事务命令，将操作序列构造成事务，并保证事务的基本特性。本节主要介绍事务的相关概念。

定义 9-1(事务)　数据库系统中的事务是一个不可分的操作序列，其中的操作要么全部都不执行，要把全部都执行。

例如，银行转账操作(如“*A* 账户转账到 *B* 账户 100 元”)可以分解为两个操作。

(1)*A* 账户上减掉 100 元：*A*=*A*−100。

(2)*B* 账户上增加 100 元：*B*=*B*+100。

这两个操作对于转账来说是不可分的，要么都做，要么都不做。如果 DBMS 只执行了其中的某个操作，就会导致数据库的不一致。

在一个应用中，究竟哪些操作序列是一个事务？这一问题需要由用户根据应用的需求来解决。DBMS 一般提供了事务定义的手段，用户可以使用它们来定义事务，并且告诉 DBMS。一旦 DBMS 接收到了事务定义请求，则会保证事务的性质。

在 ANSI SQL 中，事务的定义通过 SQL 命令 Begin Transaction、Commit Transaction 和 Rollback Transaction 来完成。Begin Transaction 表示事务的开始，Commit Transaction 表示事务全部操作都执行完成并提交，Rollback Transaction 表示事务的全部操作都取消了。Begin Transaction 之后的所有数据库操作都是事务内的操作，一直到收到 Commit Transaction 或 Rollback Transaction 命令为止。当然，不同的 DBMS 所支持的事务命令在格式上有所差异。Microsoft SQL Server 的事务命令包括 Begin Transaction、Commit Transaction 和 Rollback Transaction，与 ANSI SQL 一致。MySQL 的事务命令包括 Start Transaction、Commit、Rollback，Oracle 的事务命令包括 Commit Work、RollbackWork(Oracle 没有事务开始的命令，因为它认为一旦连接建立就开始了一个事务)。

图 9-1 给出了在 MySQL 的存储过程中使用事务的例子。该例子实现了简单的银行转账功能。由于实际的银行转账具有事务的性质，因此在实现时需要通过事务命令将转账过程定义为一个事务。第 8 行的 Start Transaction 语句表示开始事务，意味着从第 9 行开始的所有数据库操作都属于一个事务，是不可分的操作序列。第 22 行的 Commit 语句表示事务正常完成并提交，即转账操作顺利完成。此时，DBMS 可以保证转账的结果写回数据库，并保证持久性。第 25 行的 Rollback 语句表示事务取消。这通常是因为事务操作过程中发生了错误，例如，图 9-1 的例子中是转出账户的余额不足导致的转账事务无法完成。事务一旦执行 Rollback 语句，则所有已做的操作全部撤销，数据库回退到事务执行之前的状态。

```
1
2   delimiter //
3   CREATE PROCEDURE transfer(IN id_from INT, IN id_to INT, IN amount INT, OUT state INT)
4   BEGIN
5       DECLARE s INT DEFAULT 0;
6       DECLARE a INT;
7       DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET s = 1;
8       START TRANSACTION;
9       SELECT count(*) FROM account WHERE id = id_from or id=id_to INTO a;
10      IF  a < 2 THEN -- 至少有一个账户不存在
11          SET s = 2;
12      END IF;
13
14      SELECT balance FROM account WHERE id = id_from INTO a;
15      IF  a < amount THEN  -- 余额不足
16          SET s = 3;
17      END IF;
18      UPDATE account SET balance = balance - amount WHERE id = id_from;
19      UPDATE account SET balance = balance + amount WHERE id = id_to;
20      IF s = 0 THEN
21          SET state = 0;
22          COMMIT;
23      ELSE
24          SET state = -1000;
25          ROLLBACK;
26      END IF;
27  END //
28  delimiter;
```

图 9-1　MySQL 事务示例：银行转账

图 9-2 给出了在 VB 中使用 ADO 进行事务编程的示例。ADODB.Connection 对象提供了 BeginTrans、CommitTrans、RollbackTrans 三个方法用于事务编程。一旦建立了数据库连接，就可以使用这三个方法在前端程序中执行事务操作。

```
cnn.Open
On Error Goto RollbackAll '错误陷阱
cnn.BeginTrans '此连接下的所有操作现在开始都属于一个事务
Dim rst1, rst2 as New ADODB.Recordset '执行记录的增删改
rst1.Open "account", cnn, adUseClient, adOpenKeyset, adLockOptimistic, adCmdTable
rst1.AddNew '增加新记录
......
rst2.Open "summary", cnn, adUseClient, adOpenKeyset, adLockOptimistic, adCmdTable
...... '更新关联的summary表
'当发生任何预期错误时，RollbackTrans
If rst2.EOF and rst2.BOF Then
   Goto RollbackAll
End If
......
cnn.CommitTrans '成功到达事务尾部时，提交事务
cnn.Close
RollbackAll: 'Rollback事务的操作统一进行处理
   cnn.RollbackTrans
   cnn.Close
```

图 9-2　使用 ADO 进行事务编程示例（以 VB 为例）

9.2　事务的性质

事务一般应满足四个性质，即原子性、一致性、隔离性和持久性。这四个性质在数据库领域中一般合称为事务的 ACID 性质。

1) 原子性

事务的原子性是指一个事务内部的所有操作要么全部都执行，要么一个也不执行，即所有操作是一个整体。以银行转账为例，如果定义为事务，那么 *A*+100 和 *B*–100 这两个操作要么全部执行，要么全部不执行。这就是事务原子性的含义。

2) 一致性

事务的一致性是指事务的执行保证数据库从一个一致状态转到另一个一致状态，即数据库不会应事务的执行而不一致。但是，事务的内部不一定能够满足数据库的一致性。因此，按照事务的一致性，如果数据库初始状态是一致的，并且所有的操作都以事务执行，则数据库总是可以从一个一致状态迁移到另一个一致状态，保证数据库的一致性。

3) 隔离性

事务的隔离性是指多个事务一起执行时相互独立，互相不影响，任一事务的执行结果直到其成功提交后才对其他事务可见。从另一角度讲，事务的隔离性要求多个事务并发执行的结果与这些事务单独顺序执行的结果相同。

事务的隔离性是数据库中相对较为复杂的一个问题。这是因为不同的应用对于事务的隔离性有着不同的需求，例如，有的应用希望所有的用户之间完全隔离，而有的应用为了提高操作的并发度可以容忍用户之间操作的不隔离。因此，在 SQL 数据库系统中，通过定义不同级别的事务隔离性来满足不同应用的需求。

4) 持久性

事务的持久性是指事务一旦成功提交，其结果就在数据库中持久保存，不会因为关机等而导致数据丢失。

9.3　事务的状态

为了在数据库中有效地管理事务，DBMS 需要了解每个事务的执行状态。通过事务的状态，DBMS 可以知道事务是否已经执行完毕，从而可以进一步采取适当的处理机制(例如，如果事务没有正常结束，则可以选择撤销事务已执行的部分操作)。

事务在执行过程中一般有三种状态。

1) 事务开始

事务开始状态表明事务开始执行。如果 DBMS 的事务管理器接收到 Begin Transaction 命令，则开始一个事务的执行。此时，DBMS 将会记录下事务的开始状态。事务的开始状态可以形式化地表示为<Start *T*>，其中 *T* 是事务标识。

2) 事务提交

事务提交状态意味着事务中的所有操作都已完成，数据库又处于一个新的一致状态。

事务提交一般由 SQL 中的 Commit Transaction 命令来完成。Commit Transaction 操作为事务建立了一个提交点(Commit Point)。事务提交状态可以形式化地表示为<Commit *T*>。

3) 事务回退

事务回退状态表明事务执行时发生故障，事务中已做的操作必须全部撤销。当调用 SQL 的 Rollback Transaction 命令时，DBMS 将记录下事务的回退状态。事务的回退状态可形式化地表示为<Abort *T*>。

当事务开始执行时，DBMS 可以将事务的状态信息存储在一个日志文件中，从而通过读取日志文件中的事务状态信息，DBMS 可以快速地判断某个事务处于什么状态。数据库系统正常运行时，所有事务都是从开始状态到提交状态或者回退状态的，即生命期是完整的。但是，如果一个事务在执行过程中因为各种故障而中断了，日志文件中记录的该事务状态就可能只有开始状态而没有提交状态或者回退状态。此时，DBMS 很容易就知道该事务在执行过程中被中断了。由于事务必须满足原子性，而且在事务执行过程中，数据库有可能处于不一致的状态，所以 DBMS 可以采取一定的恢复策略对这类事务加以处理，例如，取消事务已做的所有操作，使数据库回退到开始之前的状态，这就是采用 Undo 日志进行数据库恢复的基本思想，将在第 10 章详细讨论。

9.4　事务的原语操作

事务内部一般包含一系列的数据库操作，这些数据库操作通常对应着 SQL 中的 Update、Insert、Delete 等操作。事务的这些内部操作在底层最终都将转换为基本的读写操作，这些基本的读写操作称为事务的原语操作，即不可分的基本操作。事务的原语操作可分为四类。

(1) Input(*x*)：将包含 *x* 的磁盘块从外存读入到 DBMS 的缓冲区。

(2) Output(*x*)：将包含 *x* 的内存块写入到相应的磁盘中。

(3) Read(*x*, *t*)：将数据 *x* 赋给内存变量 *t*。如果 *x* 不在缓冲区中，则需要首先执行 Input(*x*)操作将 *x* 读入缓冲区。

(4) Write(*x*, *t*)：将内存变量 *t* 的值赋给缓冲区中的 *x*。如果 *x* 不在缓冲区中，则需要首先执行 Input(*x*)操作将 *x* 读入缓冲区。

图 9-3 给出了事务原语操作的示意图。Input 操作和 Output 操作代表了实际的磁盘 I/O 操作。从用户角度看，任何事务都是由 Read 和 Write 操作构成的序列，因为 Input 和 Output 操作是 DBMS 内部采取的操作，对于客户端是透明的。

此外，由于事务要保证持久性，因此可以假设所有事务最终的结果都需要写回磁盘，即需要对所有修改的数据执行 Output 操作。基于这一分析，可以将事务的四类原语操作简化为 Read 和 Write 两类(Input 操作已经包含在 Read 和 Write 当中了)。如此一来，任何一个事务都可以表示为 Read 操作和 Write 操作的序列。

例如，银行转账事务可以表示为下面的操作序列：

```
T1:   Read (A,t);
      t = t -100;
      Write (A,t);
```

```
Read (B,t);
t = t + 100;
Write (B,t);
Output (A);
Output (B);
```

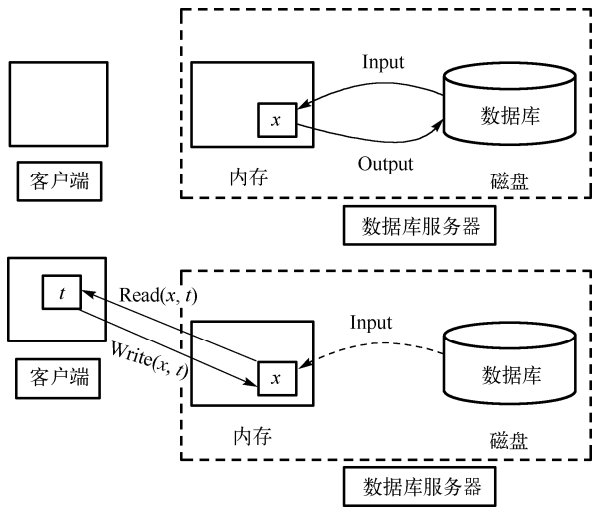


图 9-3　事务原语操作的示意图

由于所有事务在执行结束时都需要把数据写回磁盘，即执行 Output 操作，因此默认事务结束时都执行 Output 操作，并且在表示事务时省略掉 Output 操作。此外，上述例子中的 $t = t - 100$ 和 $t = t + 100$ 并不是 DBMS 端的操作，只是客户端执行的本地操作。因此，DBMS 实际上是无法获知客户端将数据读入本地变量 t 之后的操作语义的。DBMS 唯一能够知道的，就是事务从数据库中读取的值（Read）以及事务写入数据库中的值（Write）。

综上所述，在 DBMS 的事务管理器上，所有事务都可表示为 Read 操作和 Write 操作的序列。例如，上面的事务 T_1 可表示为 Read(A, t), Write(A, t), Read(B, t), Write(B, t)。一般地，可以将事务 T_1 表示为下面的形式：

```
T1: r1(A)w1(A)r1(B)w1(B)
```

操作 $r_i(A)$ 表示事务 T_i 的 Read(A, t) 操作，$w_i(A)$ 表示事务 T_i 的 Write(A, t) 操作。在后面的内容中，均使用这种形式来表示事务。

9.5　数据库一致性

事务是保证数据库一致性的基础。本节讨论数据库一致性的相关概念。需要注意的是，一致性这一术语在不同的领域中有着不同的含义，例如，在数据库中的含义与在分布式系统中的含义是不同的。在本书中，如果不特别说明，一致性默认指数据库一致性。

数据库一致性是定义在数据库的“一致性约束”这一概念上的。一致性约束

(Consistency Constraint)也称为完整性约束(Integrity Constraint)，是指数据库中的数据必须满足的谓词条件。例如，X 是 R 的主码，X 的域是{1, 2, 3, 4, 5}，等等。数据库中的不同数据应满足的谓词条件有所不同，因此一般一个数据库中会定义若干个一致性约束。如果当前数据库中的所有数据都满足所定义的一致性约束，称数据库处于一致状态(Consistent State)，或称为一致性数据库(Consistent Database)。

数据库的一致性并不是永远都成立的。下面给出一个例子。在这个例子中，假设有一个银行数据库，其中账户余额上定义了一个约束：

$$a_1 + a_2 + \cdots + a_n = \text{TOT}$$

式中，$a_1,a_2,\cdots,a_n$ 都是账户余额；TOT 是数据库中的另一个属性，表示所有账户余额之和。TOT 从理论上来说是一个冗余数据，按照传统的规范化理论，在数据库模式设计时不应当保留这种冗余属性。但是，如果在模式评价时发现性能不能满足，则可以通过增加这种冗余属性来提高性能。例如，如果账户非常多，而且分布在不同的部门，同时应用系统又需要频繁地统计账户余额，则通过增加一个账户余额属性 TOT 就可以大大提高此类查询的性能。但这么做的要求是必须满足上面给出的一致性约束条件。

现在，假设 a_2 账户上增加了 100 元，这一操作必须同时执行下面的两步才能保证数据库的一致性：

$$a_2 = a_2 + 100$$

$$\text{TOT} = \text{TOT} + 100$$

但是，这两个步骤执行时会形成一个不一致状态，如图 9-4 所示。当刚执行完其中任何一个操作时，数据库中的数据是不满足前面定义的一致性约束的，即图中的 State 2。通过分析，发现图 9-4 中的 State 2 是无法避免的，因为要保证约束的有效性，必须执行两步操作，但这两步操作在 DBMS 内部必然有一个先后顺序，因此理论上无法保证在任何进行一次操作时数据库一致性都不被破坏。

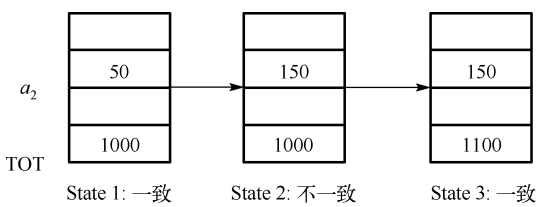


图 9-4　数据库一致性在操作时被破坏

这一个问题目前已经通过事务理论得到了解决。事务的一致性保证数据库的一致性不会因为事务的执行而被破坏，也就是说如果在事务执行之前数据库是一致的，事务执行后数据库必定还是一致的。同时，在事务执行的过程中允许数据库一致性不被满足。因此，像图 9-4 的例子，只要将 a_2 和 TOT 的两个操作构造成一个事务，保证事务执行前(State 1)和事务执行后(State 3)是一致的即可，但事务内部(State 2)可以不满足一致性。实际上，如果事务执行时最终停在了 State 2 上，说明事务执行出现了故障，DBMS 在恢复时需要消除掉 State 2 这种不一致的状态，使数据库状态恢复到 State 1 或者 State 3 的一致状态。这些都是数据库恢复技术中将要讨论的内容，会在第 10 章中讨论。

与数据库一致性类似的另一个概念是数据库的正确性。数据库的正确性一般指数据库中的数据应符合现实世界中真实的数据特征。数据库系统的最本质动机就是将现实世界中的数据真实地表达和存储到计算机系统中，所以数据库的正确性是用户对于数据库系统的一个基本要求。

但是，需要明确的是，数据库的正确性与现实世界中数据的正确性之间是有差异的。以“电话号码”这一数据为例，在现实世界中都知道“abcdefg”形式的号码不是正确的电话号码，因为电话号码是由数字组成的。这种数据的正确性在数据库中可能得到保证，因为可以通过 SQL 的 Check 约束等来实现。“0000000”“1111111”这种类型的电话号码在现实世界中一般也是不正确的，但 DBMS 很难分辨这种类型的错误数据。结论就是，数据库的正确性是依赖于 DBMS 提供的约束定义能力的，因此 DBMS 中数据库的正确性与现实世界中数据库的正确性存在一定的差别。

一般地，如果数据库在事务开始执行时是一致的，并且事务执行结束后数据库仍处于一致状态，则数据库满足正确性。也就是说，如果 DBMS 可以保证数据库的一致性和事务的一致性，那么就可以认为 DBMS 保证了数据库的正确性。因此，本质上数据库的正确性与一致性概念是相通的，其中数据库一致性是基础。在数据库领域中，常常用数据库一致性的概念，而较少使用数据库正确性的概念。

9.6　本 章 小 结

本章主要介绍了事务的基本概念和 ACID 性质，以及事务的状态与原语操作，最后讨论了数据库一致性和正确性的基本概念。事务是数据库系统实施数据保护的理论基础，它为第 10 章和第 11 章讨论数据库故障恢复和并发控制奠定了基础。

通过对本章的学习，读者应了解事务的概念、性质、状态以及原语操作，理解事务的 ACID 性质，并掌握数据库一致性的基本概念。

习　　题

1．事务具有哪几个状态？分别是什么含义？

2．什么是事务的 ACID 性质？请给出违背事务 ACID 性质的具体例子，对于每个性质举一个例子。

3．目前许多 DBMS（如 MySQL）都默认不支持嵌套事务（即在一个事务内部又开始另一个事务）。如果 DBMS 支持嵌套事务，将遇到哪些问题（写出 2 点以上并且要给出自己的分析）？

4．为什么事务非正常结束时会影响数据库数据的一致性？请举例说明。

第 10 章　故 障 恢 复

数据库系统正常运行时，数据库可以始终处于一致状态。但是，如果数据库系统因为各种故障而出现内存数据丢失或者系统挂起等问题，则可能破坏数据库的一致性。数据库故障恢复技术的目的是将数据库恢复到最近的一致状态。由于 DBMS 以事务为单位执行底层操作，而事务管理器保证了事务的一致性，因此如果事务都正常执行并提交，数据库的一致性不会被破坏。只有当发生数据库系统故障时，才有可能使事务的执行被中断，从而使数据库状态停留在事务执行的中间状态。因为事务内部不能保证数据库的一致性，所以此时数据库就有可能处于不一致状态。

内容提要：本章首先概述数据库保护技术，介绍乐观视角和悲观视角的两类数据库保护技术，然后介绍数据库系统故障的三种类型，接着重点讨论基于事务日志的恢复技术，包括 Undo 日志、Redo 日志、Undo/Redo 日志等，最后讨论检查点技术和日志轮转存储技术。

10.1　数据库保护技术概述

数据库保护技术的目的是排除和防止各种对数据库的干扰破坏，确保数据安全可靠，以及在数据库遭到破坏后尽快地恢复。按照不同的视角，数据库保护技术可分为两种类型：一种基于乐观的视角，认为数据库一致性一般不会遭到破坏，因此认为平时 DBMS 不必采取额外的数据库防护措施，只有确认数据库一致性受到破坏时才采取必要的行动将数据库恢复到最近的一致状态；另一种基于悲观的视角，认为数据库一致性在实际运行中很容易遭到破坏，因此有必要提前采取一些措施来防止数据库一致性被破坏。

基于乐观视角的数据库保护技术主要有数据库恢复技术。数据库恢复技术主要解决一旦数据库一致性受到破坏，如何尽快地将其恢复到最近的一致状态的问题。

基于悲观视角的数据库保护技术主要有数据库完整性控制技术、数据库安全性控制技术以及并发控制技术。这三类技术都是通过预先采取数据库保护策略来避免数据库在特定情况下受到破坏的。它们的区别在于所针对的问题：数据库完整性控制技术主要解决如何保证数据库中约束的有效性的问题，数据库安全性控制技术主要解决数据存取中授权与验证的问题，并发控制技术主要解决多用户同时存取同一数据时如何保证数据库一致性的问题。

10.2　数据库系统故障类型

数据库系统在运行过程中有可能发生一些故障，从而导致事务不能正常执行，最终破坏数据库一致性。

数据库系统的故障可分为三种类型，即事务故障、系统故障和介质故障。

1. 事务故障

事务故障指发生在单个事务内部的故障。事务故障只影响当前事务的执行，不会影响到数据库系统中其他事务的执行。事务故障又可分为可预期的事务故障和不可预期的事务故障两种类型。

可预期的事务故障是指应用程序可以发现的故障，如转账时余额不足、账户错误等。可预期的事务故障一般由应用程序负责处理，要求数据库程序员在编码时定义相应的故障类型并有针对性地进行处理(如回退事务、输出错误消息等)。这类事务故障的处理可以在编写存储过程时结合事务编程和异常处理机制加以实现，即在存储过程的程序体中通过错误陷阱捕捉可预期的事务故障，并在异常处理程序体内对捕捉到的故障进行处理。一般的处理方法都是回退事务以及返回相应的错误码和错误消息。

不可预期的事务故障一般指导致事务中断的不可预期故障，如运算溢出等。这类故障难以预测，一旦发生将导致事务被异常中止。应用程序一般无法处理此类故障，由 DBMS 在恢复时统一进行处理。具体而言，发生了不可预期的事务故障时，需要重启数据库实例。在重启的过程中，DBMS 将执行数据库恢复操作对这类故障进行处理。

2. 系统故障

系统故障也称为软故障(Soft Crash)，是指由于操作系统、DBMS 等软件问题或断电等问题而导致内存数据丢失，但磁盘数据仍在。系统故障一旦发生，将导致 DBMS 的缓冲区数据全部丢失。因为缓冲区是被所有事务共享的，因此当前正在运行的所有事务都会受到影响——所有事务的内存数据都将丢失，但系统故障不破坏数据库的物理存储。

系统故障发生时，一般 DBMS 需要通过重新启动实例来执行恢复过程。系统故障的恢复通过基于日志和检查点的恢复方法实现。

3. 介质故障

介质故障也称为硬故障(Hard Crash)，一般指数据库的物理存储被破坏而导致数据库文件损坏。介质故障一般指磁盘损坏，其他的一些故障(如硬盘被盗、自然灾害等)都可以纳入介质故障。

根据一些调查结果，介质故障是实际数据库系统中最为频繁发生的故障。因此，应用系统必须有能够应对介质故障的方案。由于介质故障导致整个磁盘损坏，所以介质故障的恢复需要借助数据库备份来完成。如果没有数据库备份，意味着数据库数据全部丢失。对于介质故障，首先需要重装系统，然后重新将数据库备份安装到数据库系统中，最后借助日志文件进行恢复。

10.3　故障恢复策略

数据库恢复的目的是将数据库恢复到最近的一致状态。数据库恢复的基本原则是冗余(Redundancy)。冗余的含义是指对数据库进行转储(备份)。只有存在数据库备份，才有可能在介质故障时进行数据库恢复。

数据库恢复的具体实现方法可总结为下面三点。

1. 定期转储整个数据库

为了保证数据库恢复的有效性，必须定期备份数据库。目前，商用 DBMS 一般都提供了相应的备份工具(如 Microsoft SQL Server 的备份管理器)和备份策略来允许 DBA 制定特定的数据库备份策略。常见的备份策略如下。

(1)海量备份：备份整个数据库文件。这种策略备份时间长，备份存储代价高，但恢复时过程简单。

(2)增量备份：只备份自上一次海量备份以来所更新的数据。这种方式备份快，备份存储代价低，但恢复时需要根据上一次海量备份和现有的增量备份重建最新的数据库状态，比较耗时。

(3)脱机备份：备份时暂停数据库服务，不能响应用户的事务请求。

(4)联机备份：备份时还可以同时响应事务请求。

一般在实际应用中，DBA 可以根据系统的特点来制定组合式的备份策略。例如，在一个银行系统中，可以规定每天 00:00 做一次脱机的海量备份，之后每 5min 做一次联机的增量备份。

2. 建立事务日志

事务日志记录了事务的状态以及事务对数据库的所有更新细节。通过事务日志，DBMS 可以了解事务的执行进程，并且知道事务对哪些数据做了什么样的修改。这些信息在恢复过程中既可以用来检测数据库一致性有没有被破坏，也可以作为恢复时数据撤销和重做的依据。

3. 通过备份和日志进行恢复

数据库备份和日志是数据库恢复的基础。在恢复时，如果是介质故障，则 DBA 可以重装备份，然后将最新的事务日志应用到备份上以得到最新状态的数据库；如果是系统故障，则可以由 DBMS 来自动扫描事务日志完成恢复(一般在 DBMS 重启时进行恢复)。图 10-1 给出了数据库恢复的基本过程。当发生故障时：①若是介质故障，则首先重装备份；②利用日志进行事务故障恢复和系统故障恢复，一直恢复到故障发生点。

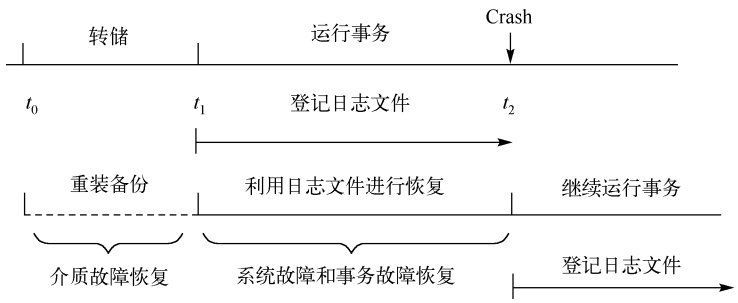


图 10-1　数据库恢复的基本过程

10.4　基于事务日志的恢复技术

日志按事务执行先后顺序记录了事务对数据库所做的每个更新操作的细节。日志中主要包含两方面的内容。一是事务的状态信息，如事务开始状态、提交状态、回退状态等。在每个事务开始执行时，DBMS 在日志中写入事务开始标志，当事务提交后写入事务提交标志或者当事务取消后写入事务回退标志。通过日志中的事务状态，DBMS 可以知道事务的执行进程。二是事务的更新操作记录。根据更新操作记录方式的不同以及随之而来的恢复方法的不同，日志可以分为 Undo 日志、Redo 日志和 Undo/Redo 日志三种类型。

10.4.1　Undo 日志

Undo 日志的事务更新操作记录格式为<*T*, *x*, old_value>。其中 *T* 是事务标识，*x* 是数据对象，old_value 是更新前的值。Undo 日志的登记规则如下。

(1) 事务的每一个修改操作都生成一个 Undo 日志记录<*T*, *x*, old_value>。

(2) 在 *x* 被写到磁盘之前，对应此修改的日志记录必须已被写到磁盘上。

(3) 当事务的所有修改结果都已写入磁盘后，将事务提交状态<Commit, *T*>写到磁盘上。

其中，第二个规则称为先写日志(Write-Ahead Logging，WAL)规则。引入日志后，实际上在执行数据的更新操作时需要写磁盘两次：一次是写数据；另一次是写日志。先写日志规则规定日志必须先写，然后才写数据操作。之所以规定 WAL 规则，是因为考虑到如果先写数据后写日志，一旦这两次写操作之间发生故障导致写数据完成了但写日志没完成，则 DBMS 将无法获知这种故障，从而无法完成一致性恢复。

例如，假设采用后写日志规则，如果 T_1 将 *A* 从 1000 修改为 900 时发生故障，并且写数据操作已经完成(此时 900 已写到数据库)，但还未来得及将表示这一次修改的日志写入磁盘，则日志中与 T_1 相关的记录只有一条：

```
<Start, T1>
```

此时，当 DBMS 重启后执行恢复时，因为 T_1 没有正常结束，所以要撤销其操作，但由于日志中没有 T_1 将 *A* 从 1000 修改为 900 的更新操作记录，所以 DBMS 将无法将 *A* 恢复到 1000。

相反，如果采用先写日志规则，同样假设写日志和写数据之间发生了故障，则故障发生时日志中与 T_1 相关的日志记录为：

```
<Start, T1>
<T1, A, 1000>
```

DBMS 在恢复时就可以发现 T_1 对 *A* 所做的修改，并且可以将 *A* 恢复到 T_1 执行之前的状态，即恢复为 1000。

先写日志规则目前是 DBMS 中普遍遵循的规则。不仅 Undo 日志中采用 WAL 规则，Redo 日志、Undo/Redo 日志也均采用 WAL 规则。

图 10-2 给出了 Undo 日志的正常登记过程。从图中可以看到，在事务开始执行时，日志中首先写入事务开始标志<Start, *T*>，以后每次更新操作(Write)都会生成一条更新记录，

当事务开始将数据写入磁盘时(Output)，先写日志规则强制将缓冲区中的所有日志都写回磁盘，然后开始写数据。把缓冲区中的日志强制写回磁盘的操作称为 Flush Log 操作，在 DBMS 中一般由某个特定的后台进程来完成。

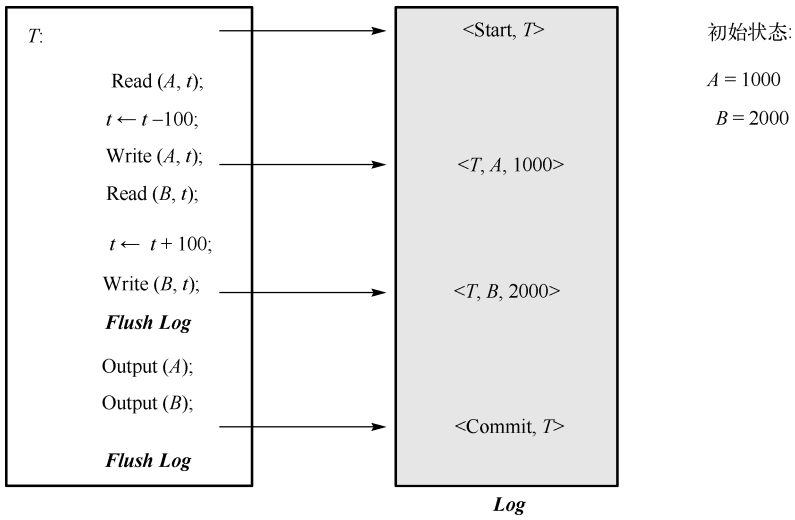


图 10-2　Undo 日志的正常登记过程

图 10-3～图 10-5 分别显示了几种不同故障情形下的 Undo 日志。在这些图中，给出了内存和磁盘中的日志与数据对象。从图中可以看到在 Undo 日志登记过程中，内存和磁盘中数据与日志的修改细节。

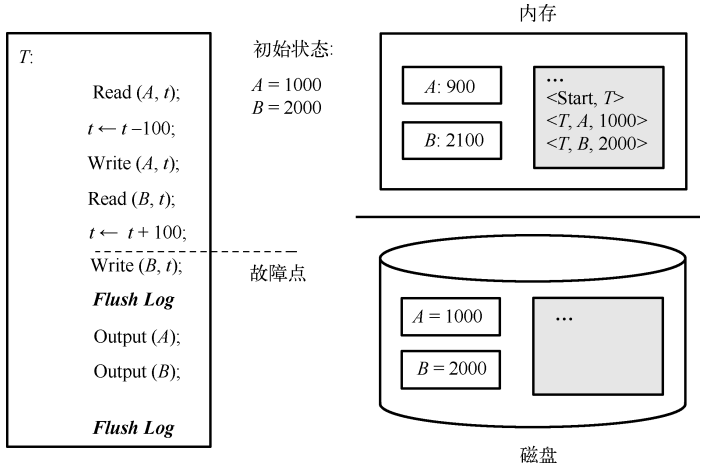


图 10-3　第一种故障情形下的 Undo 日志

在图 10-3 所示例子中，由于故障发生时日志还没有写入磁盘，所以重启后的日志文件是空的，此时并不需要执行任何恢复过程，因为数据库中的数据也没有进行任何修改。

注意在图 10-4 和图 10-5 所示的两种故障情形下，磁盘上的日志是一样的，但数据库中的数据状态不同。因此，在这两种情况下，DBMS 无法确定数据库中的数据是否已经完成了修改，在恢复时都需要执行撤销操作。一种常见的误解是认为此时可以通过扫描数据

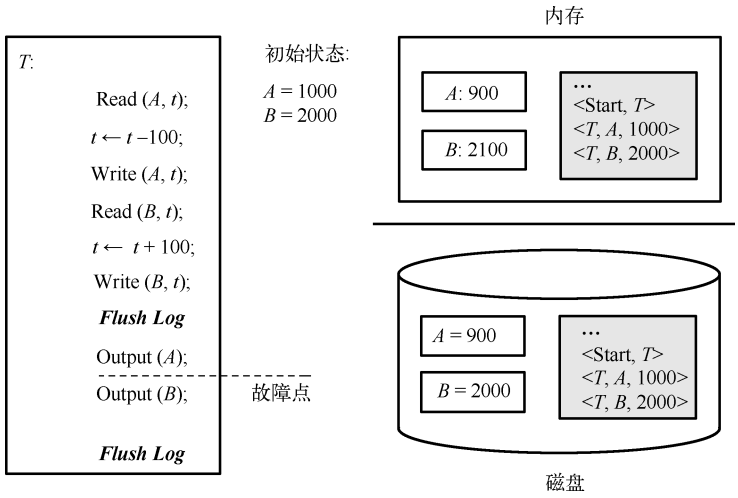


图 10-4　第二种故障情形下的 Undo 日志

库文件来发现是否已经执行了全部的修改(如图 10-5 所示的情形)，但这在实际系统中是不可行的。原因在于数据库文件相对于日志文件而言要大得多，所以如果每次重启恢复时都需要扫描所有的数据库文件，将需要大量的磁盘 I/O 操作，导致恢复时间很长。因此，DBMS 在恢复时只能扫描相对较小的日志文件，通过少量的 I/O 操作来完成恢复。实际上 DBMS 往往同时采用检查点技术和日志轮转存储技术，可以使得扫描日志的 I/O 代价降到很低，使恢复的时间延迟满足应用要求。

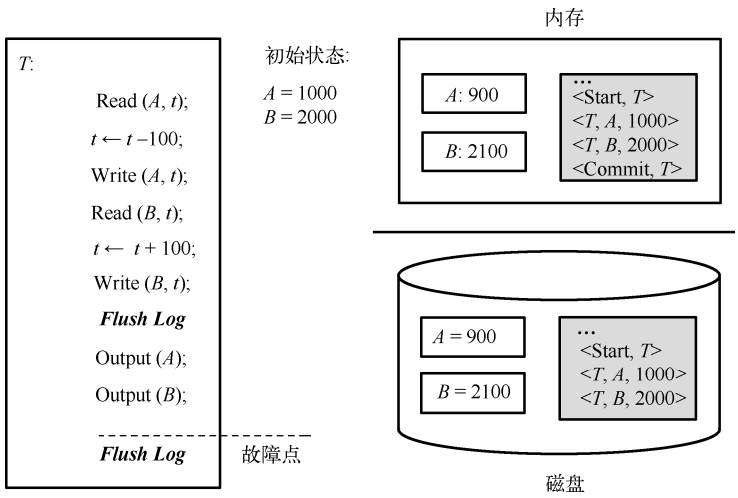


图 10-5　第三种故障情形下的 Undo 日志

图 10-6 给出了事务 *T* 提交时的日志情况，此时数据库数据和日志均已经写回了磁盘。通过图 10-6 可以发现，Undo 日志在写入<Commit, *T*>时所有的数据修改必定已经写回磁盘了，这是由 Undo 日志的登记规则决定的，因此在这种情况下 DBMS 不需要再执行任何恢复操作。

基于 Undo 日志的恢复过程如下。

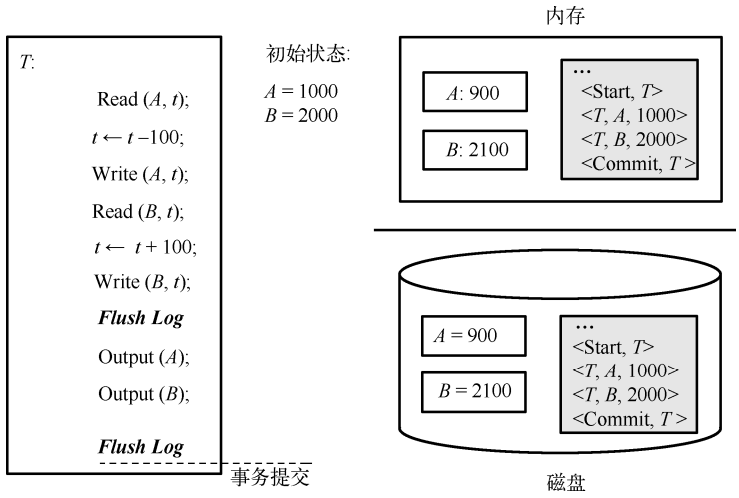


图 10-6　事务提交时的 Undo 日志

(1) 构建 Undo 事务列表：从头扫描日志，找出所有没有<Commit,*T*>或<Abort,*T*>的事务，将其放入一个事务列表 *L* 中。

(2) 执行 Undo 操作：从日志尾部开始扫描日志记录<*T*, *x*, *v*>，如果 $T \in L$，则执行：

```
Write (x, v)
Output (x)
```

即将数据库中的 *x* 值重新改回旧值。

(3) 写事务回退标志：

```
For each T ∈ L do
   Write <Abort,T> to Log
```

下面以图 10-3～图 10-5 所示的故障情形为例讨论 Undo 日志的恢复过程。

(1) 对于图 10-3，由于发生故障时磁盘中的日志并没有实际写入，所以 DBMS 不需要做任何恢复操作，因为此时所有的数据修改还只发生在内存中，磁盘中的数据还没有被修改。

(2) 对于图 10-4 和图 10-5，因为这两种故障情形下磁盘中的日志相同，所以恢复过程也相同。在这两种情形下，故障发生时 *T* 只有<Start, *T*>状态，没有<Commit, *T*>状态或<Abort, *T*>状态，因此需要 Undo。Undo 时从日志尾部开始向前扫描，首先恢复 *B*，然后恢复 *A*，具体的执行操作序列如下：

```
Write(B,2000)
Output(B)
Write(A,1000)
Output(A)
```

最后，在日志中写入<Abort, *T*>记录，表示 *T* 的所有操作都已经被撤销。

另一个问题是：如果在恢复过程中又发生故障怎么办？只要恢复没有最终完成，那么磁盘中的日志就不会变化，所以如果恢复过程中再次发生故障，DBMS 将再次执行恢复操

作。需要注意的是，无论执行多少次恢复操作，最终的数据库状态都是相同的。在该例子中，这个状态就是：*A*=1000，*B*=2000。

基于 Undo 日志的数据库恢复技术可以归纳为以下几点：

(1) 日志中的事务更新操作记录<*T*, *x*, *v*>记录了数据对象修改前的旧值；

(2) 遵循先写日志规则，写数据之前必须要把表示此次修改的日志先写回磁盘；

(3) 在日志中写入<Commit, *T*>之前必须先将数据写入磁盘；

(4) 恢复时忽略已提交的事务，只撤销未提交的事务，因为有<Commit,*T*>的事务肯定已写回磁盘。

10.4.2　Redo 日志

Redo 日志的事务更新操作记录格式为<*T*, *x*, new_value>。其中 *T* 是事务标识，*x* 是数据对象，new_value 是更新后的值。

Redo 日志的登记规则如下：

(1) 事务的每一个修改操作都生成一个 Redo 日志记录<*T*, *x*, new_value>；

(2) 在 *x* 被写到磁盘之前，对应此修改的日志记录必须已被写到磁盘上，即遵循先写日志规则；

(3) 在数据写回磁盘前先将<Commit, *T*>日志记录写回磁盘。

图 10-7 给出了 Redo 日志的正常登记过程。可以看到，Redo 日志与 Undo 日志相比主要的区别在于写<Commit, T>的时机。Undo 日志是将所有数据写回磁盘后再在日志中写入<Commit, T>，而 Redo 日志是先写<Commit, T>记录，然后开始实际的数据写过程。

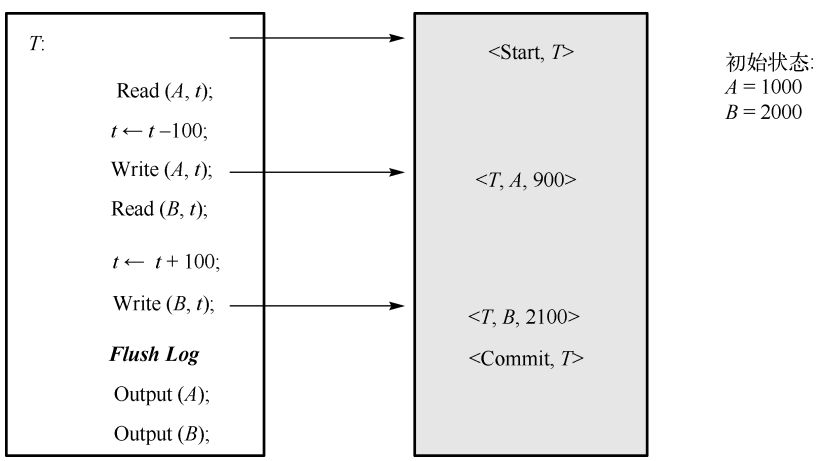


图 10-7　Redo 日志的正常登记过程

图 10-8～图 10-10 分别给出了三种故障情形下的 Redo 日志。图 10-11 给出事务提交时的 Redo 日志。特别需要注意的是，在后两种故障情形下，磁盘上的 Redo 日志和正常结束时的日志是相同的，但是数据库却有可能处于不同的状态，所以 DBMS 无法确定数据库是否已经完成了修改。

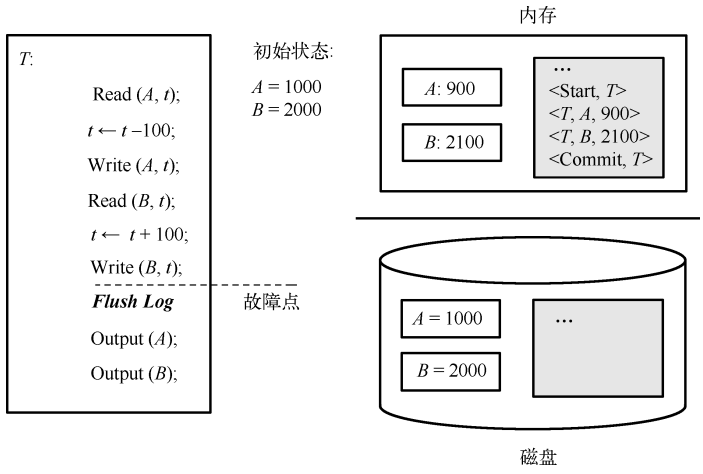


图 10-8　第一种故障情形下的 Redo 日志

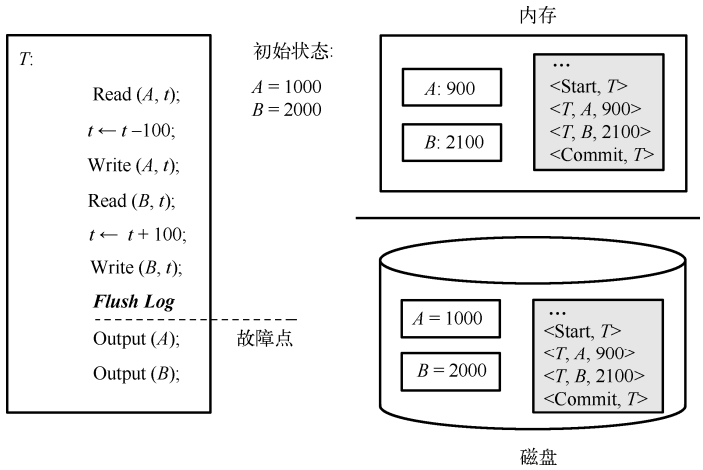


图 10-9　第二种故障情形下的 Redo 日志

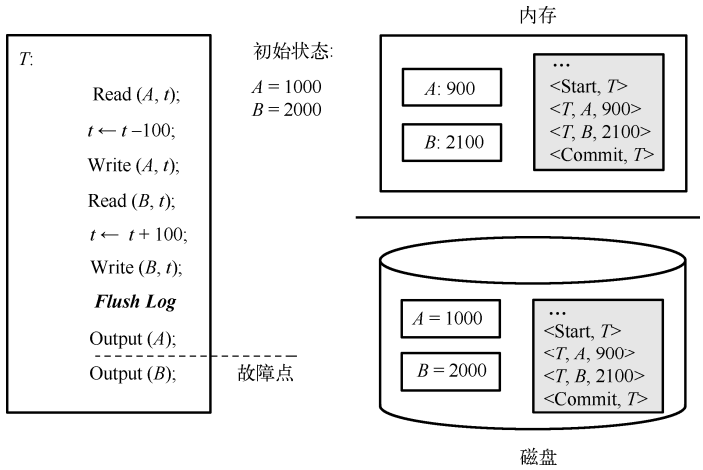


图 10-10　第三种故障情形下的 Redo 日志

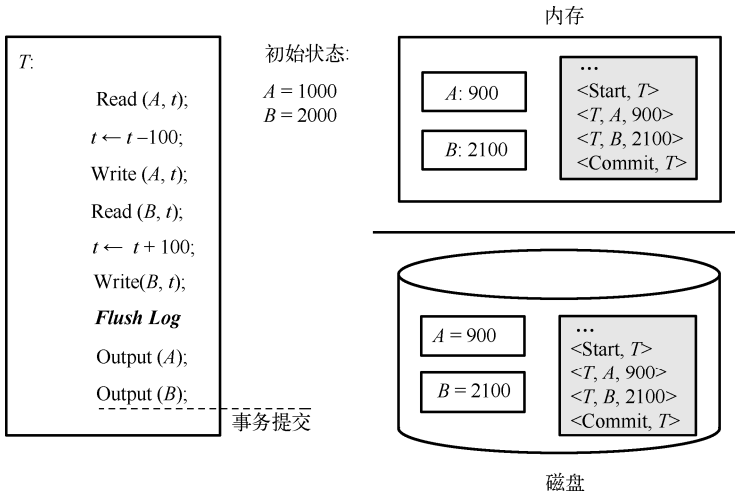


图 10-11　事务提交时的 Redo 日志

基于 Redo 日志的恢复过程如下。

(1) 构建 Redo 事务列表：从头扫描日志，找出所有<Commit,*T*>的事务，将其放入一个事务列表 *L* 中。

(2) 执行 Redo 操作：从日志头部开始扫描日志记录<*T*, *x*, *v*>，如果 $T \in L$，则执行：

```
Write (x, v)
Output (x)
```

即将数据库中的 *x* 值重新改回旧值。

(3) 写事务回退标志：

```
For each T ∉ L do
   Write <Abort,T > to Log
```

以图 10-10 为例，DBMS 在恢复时将把 *T* 放入 Redo 事务列表，并且根据日志记录顺序从前往后执行下面的恢复操作：

```
Write(A,900)
Output(A)
Write(B,2100)
Output(B)
```

与 Undo 日志相比，基于 Redo 日志的恢复的特点可总结为下面几点。

(1) Redo 日志中没有<Commit,*T*>记录的操作必定没有改写磁盘数据，因此在恢复时可以不理会；而 Undo 日志中没有<Commit,*T*>记录的操作有可能改写了数据，在恢复时需要加以撤销。

(2) Redo 日志中有<Commit,*T*>记录的结果可能还未写回磁盘，因此在恢复时要执行 Redo 操作以保证数据写回磁盘；而 Undo 日志中有<Commit,*T*>记录的结果必定已经写回磁盘，所以在恢复时不需要进行任何处理。

(3) Redo 日志中事务的写数据操作必定发生在<Commit, *T*>之后，意味着 Redo 日志中

一个事务内的所有数据更新都必须延迟到事务结束时(Commit)才写回磁盘。把这种将事务内的全部更新操作推迟到事务结束时才写回磁盘的策略称为延迟更新策略。延迟更新策略的优点是如果在事务结束之前发生了故障，恢复时不需要执行任何写数据的 I/O 操作，因为磁盘数据没有被修改。其缺点是在事务提交之前需要将所有更新的数据都缓存在内存中，因此内存代价较大。相对地，Undo 日志要求写<Commit, *T*>之前将全部的数据更新写回磁盘，因此 Undo 日志可以在事务执行了一次 Write 操作后立即将缓冲区的数据写回磁盘，同时释放内存。把这种将 Write 操作修改后的数据立即写回磁盘的策略称为立即更新策略。Undo 日志可以采用立即更新策略，它的优点是内存代价小(因为数据更新写回磁盘后即可释放内存)，其缺点是如果事务中间发生故障，必须要通过磁盘 I/O 操作写数据库才能将数据库状态回退到前一个一致状态，因此恢复时的 I/O 代价更大。当然，Undo 日志仍然可以采用延迟更新策略，但反过来，Redo 日志则无法采用立即更新策略。

10.4.3　Undo/Redo 日志

Undo/Redo 日志是 Undo 日志和 Redo 日志的混合体。Undo/Redo 日志的事务更新操作记录格式为<*T*, *x*, old_value, new_value>。其中 *T* 是事务标识，*x* 是数据对象，old_value 是更新前的值，new_value 是更新后的值。

Undo/Redo 日志的登记规则如下。

(1) 事务的每一个修改操作都生成一个 Undo/Redo 日志记录<*T*, *x*, old_value, new_value>。

(2) 在 *x* 被写到磁盘之前，对应此修改的日志记录必须已被写到磁盘上。

需要注意的是，Undo 日志和 Redo 日志在写<Commit, *T*>上的做法是不同的，Undo 日志是先写数据后写<Commit, *T*>，而 Redo 日志正好相反。因此，作为混合体的 Undo/Redo 日志在登记时不强制要求采用某种写<Commit, *T*>的方式。实际上，无论采用哪种方式，Undo/Redo 日志都可以完成恢复任务，其根本原因在于 Undo/Redo 日志在恢复时不忽略任何事务(Undo 日志在恢复时忽略了有<Commit,*T*>的事务，Redo 日志在恢复时忽略了没有<Commit,*T*>的事务)。

基于 Undo/Redo 日志的恢复过程如下。

(1) 构建 Undo 和 Redo 事务列表：从头扫描日志，找出所有<Commit,*T*>的事务，将其放入一个 Redo 事务列表 L_{Redo} 中。找出所有没有<Commit,*T*>的事务，将其放入一个 Undo 事务列表 L_{Undo} 中。

(2) 执行 Undo 操作：从日志尾部往前扫描日志记录<*T*, *x*, old_value, new_value>，如果 $T \in L_{\text{Undo}}$，则执行：

```
Write (x, old_value)
Output (x)
```

(3) 执行 Redo 操作：从日志头部开始扫描日志记录<*T*, *x*, old_value, new_value>，如果 $T \in L_{\text{Redo}}$，则执行：

```
Write (x, new_value)
Output (x)
```

(4) 写事务回退标志：

```
For each T ∈ L_Undo do
   Write <Abort,T > to log
```

下面通过一个例子讨论 Undo/Redo 日志的具体恢复过程。假设发生故障时的日志记录如下：

```
<Start,T1>
<T1,B,2000,1900>
<Start,T2>
<T2,A,1000,900>
<Commit,T1>
<Start,T3>
<T3,C,3000,2000>
<T3,B,1900,1800>
<Commit,T2>
<Start,T4>
<T4,D,1000,1200>
```

基于 Undo/Redo 日志的恢复过程如下。

(1) 构建 Undo 事务列表和 Redo 事务列表。

上例中，Undo 事务列表 L_{Undo}= {T_3,T_4}, Redo 事务列表 L_{Redo}= {T_1,T_2}。

(2) 执行 Undo 恢复操作。

从后往前依次恢复：

```
Write(D,1000)
Output(D)
Write(B,1900)
Output(B)
Write(C,3000)
Output(C)
```

(3) 执行 Redo 恢复操作。

在 Undo 恢复结果的基础上，从前往后扫描日志执行 Redo 恢复操作

```
Write(B,1900)
Output(B)
Write(A,900)
Output(A)
```

(4) 写<Abort, *T*>日志记录。

在日志尾部写入：

```
<Abort,T3>
<Abort,T4>
```

恢复后的日志记录如下：

```
<Start,T1>
```

```
<T1,B,2000,1900>
<Start,T2>
<T2,A,1000,900>
<Commit,T1>
<Start,T3>
<T3,C,3000,2000>
<T3,B,1900,1800>
<Commit,T2>
<Start,T4>
<T4,D,1000,1200>
<Abort,T3>
<Abort,T4>
```

恢复后数据库的状态为：A=900，B=1900，C=3000，D=1000。

在 Undo/Redo 日志恢复中，特别需要注意的是，必须先执行 Undo 恢复过程，然后执行 Redo 恢复过程。如果先执行 Redo 恢复过程，当 Redo 事务和 Undo 事务修改同一数据对象时将导致恢复结果出错。例如，假设在一个银行系统中，故障时的 Undo/Redo 日志如下：

```
<Start,T1>
<Start,T2>
<T1,A,1000,1200>
<T2,A,1000,1100>
< Commit,T2>
```

可以将 T_1 和 T_2 都看成存款的事务——T_1 在 A 账户上存 200 元，T_2 在 A 账户上存 100 元。根据日志记录，可以发现 T_2 应该提交，T_1 没有完成需要撤销，则最终的结果应当为 1100。但是，如果在恢复时先执行 Redo 后执行 Undo 恢复过程，则 A 在 Redo 后被修改为 1100，接着在 Undo 时被修改为 1000。最终得到了错误的恢复结果。理论上，如果 Undo/Redo 日志恢复时 Redo 和 Undo 事务列表中的事务不是更新同一数据对象，则先执行 Undo 还是先执行 Redo 对最终结果没有影响。但是为了保证恢复过程的正确性和统一性，Undo/Redo 恢复时要求先执行 Undo 恢复过程后执行 Redo 恢复过程。

10.5　检查点技术

基于日志的恢复技术存在的主要问题是当系统故障发生时，必须通过扫描整个日志文件来确定 Undo 事务列表和 Redo 事务列表。由于日志文件增长很快，因此随着系统的运行，日志扫描过程会越来越长，影响恢复的性能。此外，在 Redo 日志和 Undo/Redo 日志中，随着时间的推移，恢复时得到的 Redo 事务列表将越来越大，使得恢复过程变得很长，因为理论上所有 Commit 的事务都需要进行 Redo 操作。Undo 事务列表一般不会太长，这是因为 Undo 事务列表保存了没有正常结束的事务，而在数据库系统中大多数事务都是正常结束的，只有发生故障时正在运行的那些事务才会受到影响。这部分事务的数量跟应用系统的并发用户数有关。例如，假设系统的并发用户数为 50，那么理论上发生故障时最多有 50 个事务会受到影响，也就是说 Undo 事务列表最多只可能有 50 个事务。

对于 Redo 事务列表来说，发现实际上离故障时间点很远的事务应该是已经写回磁盘不需要再做 Redo 恢复的。例如，12:00 发生数据库系统故障，那么 1h 之前执行的事务如果在日志里已经有 Commit 标记，一般来说肯定已经写回磁盘了。因此，如果能够在恢复时将这些修改已经生效的事务从 Redo 事务列表中去除，则可以有效地提高恢复的时间性能。

但是，传统的 Redo 日志和 Undo/Redo 日志缺乏区分事务是否已经写回磁盘的方法，因此研究者提出了检查点(Checkpoint)技术来解决这一问题。

检查点技术的基本思想是在日志中加入一个特殊的检查点标记，指示在该标记之前的所有 Commit 的事务修改已经全部生效。如此一来，在恢复时就不需要再扫描检查点标记之前的日志。

检查点技术周期性地执行下面的工作过程：

(1) DBMS 开始检查点，不再接收新的事务请求；

(2) 等待当前正在执行的事务结束；

(3) 将 Buffer 中的日志记录全部写回磁盘；

(4) 将 Buffer 中的数据全部写回磁盘；

(5) 在日志中写入 Checkpoint 标记；

(6) 重新开始接收事务请求。

图 10-12 显示了有检查点时的恢复过程和无检查点时的恢复过程之间的差别。可以看到，引入检查点之后，t_0～t_1 的所有日志在恢复时不需要再进行扫描，一方面可以减少日志扫描时间，另一方面也可以减少需要 Redo 的事务数量。

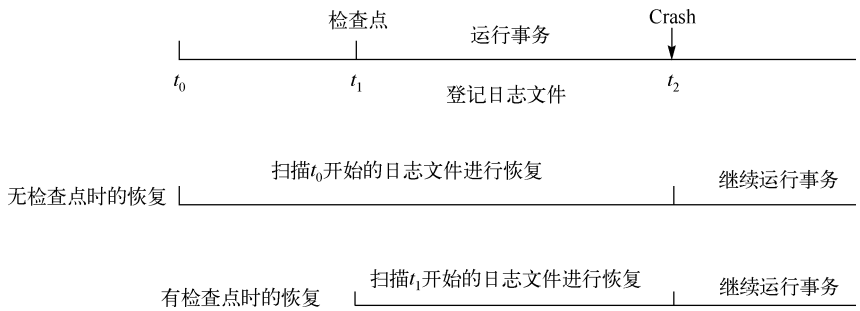


图 10-12　有无检查点时的恢复过程差别

图 10-13 给出了基于检查点的恢复示例。在这个例子中，当开始执行检查点操作时(t_s 时刻)，数据库系统中 T_1 和 T_2 事务已经完成，T_3 事务正在运行，因此系统等待 T_3 结束，并完成检查点操作(t_e 时刻)。其相应的日志记录如下：

```
<Start,T1>
…
<Start,T2>
…
<Start,T3>
…
<Commit,T1>
…
```

```
<Abort,T2>
…
<Commit,T3>
<Checkpoint>
<Start,T4>
…
<Start,T5>
…
<Commit,T4>
…
```

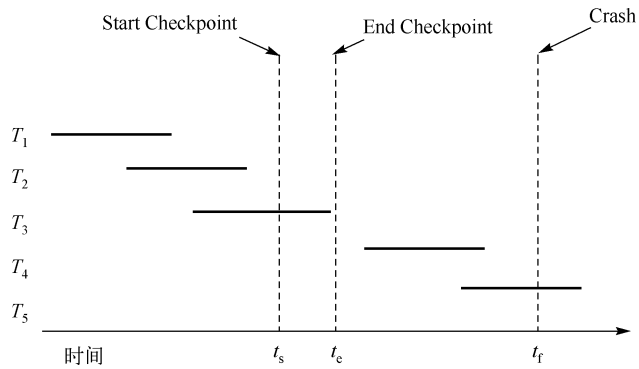


图 10-13　基于检查点的恢复示例

在恢复时，先从日志文件尾部向前扫描到最近的<Checkpoint>标志，然后从<Checkpoint>标志向后扫描产生 Undo 事务列表和 Redo 事务列表。当然这两个过程也可以合并为一轮扫描：从尾部扫描，如果是<Commit,*T*>记录，就将 *T* 放入 Redo 事务列表；如果是<Start,*T*>而且 *T* 不出现在 Redo 事务列表中，就将 *T* 放入 Undo 事务列表；如果是<Checkpoint>标记，就结束扫描过程。在图 10-13 的例子中，可以最终得到 Undo 事务列表为{T_5}，Redo 事务列表为{T_4}，T_1、T_2、T_3 都不需要再执行恢复过程，其中 T_2 是回退的事务(在日志中有<Abort,T_2>记录)，T_1 和 T_3 是在检查点时已经确认修改生效的事务。

10.6　日志轮转存储技术

检查点技术解决了恢复时需要扫描大量日志记录以及 Redo 大量提交事务的问题。在实际 DBMS 中，还存在着另一个与日志相关的问题，即日志文件增长过快导致日志文件写满的问题。

这一问题是指随着日志记录的快速写入，尤其是当同一时刻并发执行的事务数目较多时，日志文件将快速增长。如果 DBMS 采用预先分配一个较小的日志文件，一旦日志文件写满，则日志记录无法再写入，系统将挂起。但如果 DBMS 分配一个很大的日志文件，一是在低负载以及系统开始运行阶段存在存储空间浪费，二是随着时间推移，最终日志文件还会被写满。

为了解决这一问题，目前普遍采用了一种日志轮转存储的技术，也称为轮转日志。轮转日志采用若干个较小的日志文件(如 6 个 100MB 的文件)组成一个逻辑上的循环存储队

列。这些日志文件划分为两类：联机日志文件和归档日志文件。联机日志文件是当前 DBMS 运行时可以写入日志的文件，而归档日志文件不能再写入事务日志。当一个联机日志文件即将写满时，将其类型修改为归档日志文件，使其不再接收新写入的日志，以免文件写满导致系统挂起。同时选择轮转存储队列里可用的一个联机日志文件作为下一个日志写入的文件。对于归档日志文件，由后台的进程将其中的日志迁移到更廉价的存储介质上，迁移完成后的归档日志文件重新成为联机日志文件并加入联机日志文件队列。之所以要把归档日志迁移到廉价存储介质上，是因为日志的主要作用是故障恢复。随着时间的推移，越老的日志越不太可能被系统使用，因此将它存储到低速的廉价存储介质上以节省成本。

图 10-14 显示了日志轮转存储技术的示意图。其中灰色背景的日志文件表示归档日志文件，它们只能读不能写。白色背景的日志文件表示联机日志文件，它们是数据库系统运行时真正接收日志写入请求的日志文件。一旦联机日志文件的空间填充度达到了一定阈值，DBMS 就将其转变为归档日志文件，不再接收日志写入请求。此时，如果有多个联机日志文件存在，则其他联机日志文件可以继续接收日志写入请求。如果只有一个联机日志文件，则可以从已经清空的归档日志文件中选择一个将其作为联机日志文件。

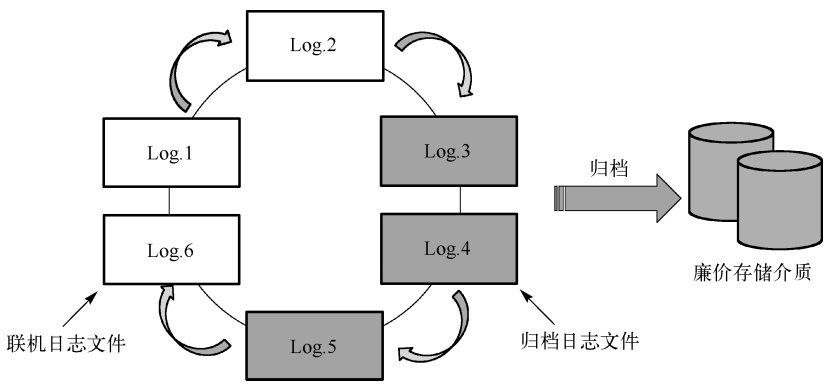


图 10-14　日志轮转存储技术

通过轮转日志技术，DBMS 可以支持 7×24h 不间断地运行，保证数据库系统不会因为日志文件写满而出现挂起的现象。不同的 DBMS 支持的轮转日志数量有所区别，数据库管理员可以在 DBMS 允许的范围内自行确定参与轮转的日志文件数量以及每个日志文件的大小。

10.7　本 章 小 结

本章主要介绍了数据库保护技术的基本概念，并重点介绍了目前数据库系统故障恢复的相关内容，包括数据库系统故障的类型、基于事务日志的恢复技术、检查点技术以及轮转日志技术。

通过对本章的学习，读者应了解事务日志的基本概念以及特征，掌握 Undo 日志、Redo 日志、Undo/Redo 日志的概念、写日志规则以及恢复过程，了解检查点技术的动机和实现方法以及日志轮转存储技术的机理。

习　　题

1．举例说明数据库系统可能出现的故障类型。
2．登记日志文件时为什么必须先写日志后写数据？
3．试述基于 Undo/Redo 日志的恢复过程。
4．具有检查点的恢复技术有什么优点？试举一个具体的例子加以说明。
5．Undo 日志与 Redo 日志的主要区别是什么？

第 11 章　并 发 控 制

并发控制的主要目的是防止多用户同时访问同一数据时破坏数据库的一致性。事务是并发控制的基本单位，保证事务 ACID 性质是事务处理的重要任务，而事务 ACID 性质可能遭到破坏的原因之一是多个事务对数据库的并发操作。对并发操作进行正确调度，避免数据库一致性在并发操作时遭受破坏，是数据库管理系统中并发控制机制的主要任务。

内容提要：本章首先介绍数据库并发操作可能导致的问题，接着介绍并发调度的概念，然后重点讨论可串化调度的相关理论和技术以及基于锁的并发控制机制，最后讨论事务隔离级别、死锁、以及乐观并发控制技术。

11.1　并发操作问题

在数据库系统中，多个事务同时对同一数据库进行操作称为并发操作。如图 11-1 所示，当多个事务同时操作同一个数据库时，由于多个事务访问共享的数据，相互之间可能产生干扰，破坏事务的隔离性和数据库的一致性。

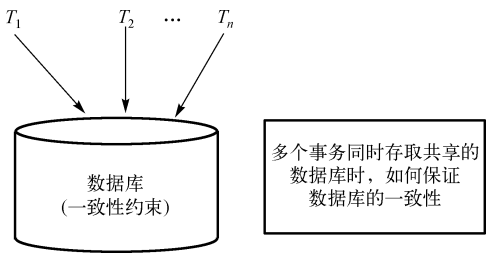


图 11-1　数据库并发操作示例

数据库的并发操作通常会导致三类问题，即丢失更新、脏读和不可重复读。这三类问题是 DBMS 的并发控制子系统需要重点协调并解决的，以避免多个事务并发执行破坏数据库的一致性。

11.1.1　丢失更新

丢失更新问题是指两个事务同时运行时，如果读入同一数据并修改，后提交的事务的提交结果覆盖了先提交的事务的提交结果，导致先提交的事务的修改被丢失。

图 11-2 给出了丢失更新问题的一个例子。在这个例子中，T_1 和 T_2 都读入了数据对象 A，T_1 先写回 A，但是接下来的 T_2 写回 A 时覆盖了 T_1 写入的数据。

丢失更新问题是并发控制中必须避免的问题，因为它直接导致数据修改出错，破坏数据库的一致性。

在 DBMS 中，事务执行数据更新时有两种执行方式：立即更新和延迟更新。立即更新是指执行 Write 操作时立即将数据写回磁盘，延迟更新是指执行 Write 操作时只把数据写入

缓冲区，但不写回磁盘，而是在事务最后提交时才将缓冲区中的所有数据写回磁盘。这两种方式各有特点。立即更新的优点是节省缓冲区，但如果事务回退，将导致较多的 I/O 操作。延迟更新的优点是在事务回退时不需要 I/O 操作，但是要求在缓冲区中保存所有的修改数据，对内存的要求高。

注意，无论立即更新还是延迟更新，如果对并发操作不加以控制，都会导致丢失更新问题。

时间	事务 T_1	事务 T_2	数据库中 A 的值
1	Read(A, t)		1000
2		Read(A, t)	
3	$t = t - 100$		
4		$t = t + 100$	
5	Write(A, t)		
6	Commit		900
7		Write(A, t)	
8		Commit	1100

图 11-2　丢失更新问题示例

11.1.2　脏读

脏读(Dirty Read)是指事务读到了脏数据。脏数据是指写入了缓冲区，但是还没有最终写回磁盘的数据。因此，不能确定脏数据肯定会写入数据库。例如，如果事务在提交之前因为某种原因执行了回退操作，则前面所做的修改全部取消，缓冲区中的脏数据将恢复旧值。

图 11-3 给出了脏读问题的示例。在这个例子中，T_2 读入 A 时读到了 T_1 修改的新值并且继续使用它进行后面的操作，T_1 接着回退了事务，A 被恢复成了旧值 1000，但此时 T_2 仍在使用回退前的 A 值 900。

脏读问题本身只是读到了不正确的数据，但是因为事务读入数据之后常常会继续进行更新操作，因此也会导致对数据库的更新出错，从而破坏数据库的一致性。

时间	事务 T_1	事务 T_2	数据库中 A 的值
1	Read(A, t)		1000
2			
3	$t = t - 100$		
4	Write(A, t)		
5		Read(A, t)	
6	Rollback	$t = t + 100$	900
7		Write(A, t)	
8		Commit	1000

图 11-3　脏读问题示例

11.1.3　不一致分析

不一致分析是指事务 T_1 读取数据后，事务 T_2 执行更新操作修改了数据，使 T_1 无法再现前一次读取结果。换句话讲，T_2 更新之后，T_1 前面读入的数据就成为过时的数据。

具体地讲，不一致分析问题包括三种情况。

(1) 事务 T_1 读取某一数据后，事务 T_2 对其做了修改，当事务 T_1 再次读该数据时，得到与前一次不同的值。例如，在图 11-4 中，T_2 读取 A=1000 进行运算，T_1 读取同一数据 A，对其进行修改后将 A=900 写回数据库。在第 5 个时间点之后，假设 T_2 为了对读取值进行校对而重读 A，A 已经成为 900，与第一次读取值不一致。图 11-4 中的这一问题表现在 T_2 最后得到的 A 和 B 的求和结果 2000 并非第 6 个时间点时数据库中真正的汇总结果(正确的结果应该是 1900)。因此，这一类不一致分析问题也称为“不可重复读”问题。

时间	事务 T_1	事务 T_2	A	B
1		Read(A, x)	1000	1000
2	Read(A, t)			
3	$t = t - 100$			
4	Write(A, t)		900	1000
5		Read(B, y)		
6	Commit	Sum = $x + y$	900	1000
7		Commit		

图 11-4　不一致分析问题示例

(2) 事务 T_1 按一定条件从数据库中读取了某些数据记录后，事务 T_2 删除了其中部分记录，当 T_1 再次按相同条件读取数据时，发现某些记录神秘地消失了。

(3) 事务 T_1 按一定条件从数据库中读取某些数据记录后，事务 T_2 插入了一些记录，当 T_1 再次按相同条件读取数据时，发现多了一些记录。

后两种不一致分析问题也称为幻象(Phantom)问题，即读到了幻象元组。图 11-5 给出了幻象读的一个例子。在这个例子中，事务 T_2 要对当前数据库中的所有数据进行求和。因为在事务开始执行时数据库中只有 A 和 B 两个数据对象，所以 T_2 读入 A 和 B，但在 T_2 读入 A 和 B 之后事务 T_1 新插入了一个数据对象 C，而这个数据对象对于 T_2 来说是不可见的，因此 T_2 后续的汇总结果是错误的。幻象问题产生的情形有两种：一是在 T_1 事务查询时 T_2 事务插入了 T_1 事务的查询语句包含的元组；二是 T_1 事务查询时 T_2 事务修改了其他元组，导致这些元组也被 T_1 的查询语句所包含。无论哪一种情形，T_1 事务再次执行同一查询时都将看到不同的结果。

时间	事务 T_1	事务 T_2
1		Read(A, x)
2		Read(B, y)
3	Write(C, t)	
4	Commit	
5		Sum = $x + y$
6		Commit
7		

图 11-5　幻象读问题示例

并发控制的三类问题的主要产生原因是并发操作破坏了事务的隔离性。如果事务都是串行执行的，那么下一个事务始终看到的是上一个事务产生的一致状态，任何事务的内部执行都不会受到前面事务的影响，可以保证隔离性。但是因为多个事务串行执行从性能角度考虑不可行，所以 DBMS 只能以事务的内部读写操作为最小单位进行并发操作。多个事务并发执行时某些事务的输入和后续行为或许会不一致，即使每一个隔离执行的事务都是正确的。并发控制就是用正确的方式调度并发操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致。

11.2　并 发 调 度

要解决并发操作带来的问题，一种方法是让所有事务串行执行，即一个事务执行完成后再执行下一个事务。由于事务一致性性质可以保证数据库的一致性不会被破坏，因此这种方式从理论上可以保证不会发生并发操作问题。但是，事务的串行执行会导致一个事务在执行时其他事务只能等待，因而不能充分利用系统资源，效率低下。因此，目前的 DBMS 为了充分发挥 DBMS 共享数据的优势，允许事务的内部操作并发执行。

事务的并发执行使得所有事务的内部读写操作交错执行，形成非串行的执行序列。把多个事务的读写操作按时间排序的执行序列称为并发调度。

事务的并发调度要遵循两个基本原则。

(1) 调度中每个事务的读写操作应保持原来的顺序。

一个事务内部的读写操作顺序代表了应用程序的操作语义，它是设计者根据应用需求和设计结果实现完成的。因此它们的顺序不能修改，因为一旦改变了，就意味着事务的操作语义变了，即使最后正确完成也不一定能实现用户要求的功能。

(2) 并发调度不考虑事务的语义和数据库的初始状态。

DBMS 只能看到事务的读写请求，并不知道事务读了数据之后所做的具体操作的细节，因此 DBMS 在做并发调度时不能考虑事务的语义。另外，一个正确的并发调度必须适应任何一致的数据库初始状态，即在不同的数据库状态下都可以保证调度结果的正确性。

图 11-6 给出了两个事务 T_1 和 T_2。假设数据库中的完整性约束是 $A=B$。

事务 T_1	事务 T_2	完整性约束
Read(A, t)	Read(A, t)	$A=B$
$t=t+100$	$t=t*2$	
Write(A, t)	Write(A, t)	
Read(B, t)	Read(B, t)	
$t=t+100$	$t=t*2$	
Write(B, t)	Write(B, t)	

图 11-6　事务 T_1、T_2 以及完整性约束

T_1 和 T_2 两个事务的几种可能的调度方式如图 11-7～图 11-10 所示。其中，图 11-7 和图 11-8 是串行调度的方式，事务的一致性保证了它们最终执行结果的一致性。图 11-9 的调度执行违背了完整性约束条件，因此肯定是不正确的调度。虽然图 11-10 的执行结果在给定的数据库初始状态下是正确的，但它是否能够在其他状态下仍保证执行结

果的一致性还需要进一步证明。可以看到，当多个事务并发执行时，存在着多种并发调度方式，其中有的是正确的，有的则会破坏数据库的一致性。因此，并发控制中的核心问题是：

(1)什么样的并发调度是正确的并发调度？

(2)如何得到一个正确的并发调度？

事务 T_1	事务 T_2	A	B
Read(A, t)		25	25
$t = t + 100$			
Write(A, t)		125	25
Read(B, t)			
$t = t + 100$			
Write(B, t)		125	125
	Read(A, t)	125	125
	$t = t * 2$		
	Write(A, t)	250	125
	Read(B, t)	250	125
	$t = t * 2$		
	Write(B, t)	250	250

图 11-7　事务 T_1、T_2 的并发调度例子(1)

事务 T_1	事务 T_2	A	B
	Read(A, t)	25	25
	$t = t * 2$		
	Write(A, t)	50	25
	Read(B, t)		
	$t = t * 2$		
	Write(B, t)	50	50
Read(A, t)		50	50
$t = t + 100$			
Write(A, t)		150	50
Read(B, t)		150	50
$t = t + 100$			
Write(B, t)		150	150

图 11-8　事务 T_1、T_2 的并发调度例子(2)

事务 T_1	事务 T_2	A	B
Read(A, t)		25	25
$t = t + 100$	Read(A, t)	25	25
Write(A, t)	$t = t * 2$	125	25
	Write(A, t)	50	25
	Read(B, t)	50	25
Read(B, t)	$t = t * 2$	50	25
$t = t + 100$	Write(B, t)	50	50
Write(B, t)		50	125

图 11-9　事务 T_1、T_2 的并发调度例子(3)

事务 T_1	事务 T_2	A	B
Read(A, t)		25	25
$t = t + 100$			
Write(A, t)		125	25
	Read(A, t)	125	25
Read(B, t)	$t = t * 2$	125	25
$t = t + 100$	Write(A, t)	250	25
Write(B, t)		250	125
	Read(B, t)	250	125
	$t = t * 2$		
	Write(B, t)	250	250

图 11-10　事务 T_1、T_2 的并发调度例子(4)

下面重点围绕这两个问题进行讨论。其中，第一个问题主要通过可串化调度理论来回答，第二个问题主要通过基于锁的并发控制机制来回答。

11.3　可串化调度

可串化调度理论回答了“什么是正确的并发调度”问题。由于串行调度不会破坏数据库的一致性，因此如果一个并发调度能够保证与某个串行调度等价，则可以保证此并发调度的正确性。这就是可串化调度理论的基本思想。

11.3.1　可串化调度概念

串行调度是指各个事务之间没有任何操作交错执行，事务一个一个执行。由于事务具有一致性性质，因此事务逐个执行不会影响数据库的一致性，所以串行调度是正确的调度。但是，如前所述，串行调度具有性能上的缺陷，会导致后执行的事务长久等待，而且不同事务之间也缺乏并发性，因此在 DBMS 中通常采取并发调度执行方式。

定义 11-1　如果一个调度的结果与某一串行调度执行的结果等价，则称该调度是可串化调度，否则是不可串化调度。

可串化调度给出了正确的并发调度定义。执行结果等价是指在任何一致的数据库初始状态下，可串化调度的执行结果都与某个串行调度的结果相同。因为串行调度所得到的数据库状态是一致的，所以可串化调度的定义保证了可串化调度执行结果的正确性，即执行后得到的数据库状态始终是一致的。

对于某个特定的并发调度，如何判断它是否是可串化调度？理论上，需要枚举所有可能的串行调度结果(给定 N 个事务，它们的串行调度有 $N!$种)。可串化调度要求在任何数据库初始状态下都和某个串行调度的结果相同，但 DBMS 无法枚举所有的数据库初始状态。

11.3.2　冲突可串性

为了解决可串化调度判定问题，引入冲突可串性的概念和相关方法。冲突可串性对并发调度的可串性进行了进一步精化。因为直接判定一个并发调度非常困难，所以将某些具

有特殊性质的可串化调度从整个可串化调度的集合中分离出来形成一个子集，并且能够用高效的方法来判断一个并发调度是否满足这些特殊性质，即是否属于所形成的子集。

冲突可串性定义了某些可串化调度的特殊性质，并且构建了一个可串化调度的子集——冲突可串调度。由于冲突可串调度具有一些特殊的性质，因此可以建立有效的方法来判断一个并发调度是否是冲突可串的。如果是冲突可串调度，则此调度必定是可串化调度，因此冲突可串调度是可串化调度的一个子集。

冲突可串调度的特殊性质是定义在调度中的冲突操作基础上的。下面给出了冲突操作的定义。

定义 11-2　冲突操作是指并发调度中满足下面条件的一对操作：

(1) 属于不同的事务；

(2) 操作同一个数据对象；

(3) 至少有一个操作是写操作。

图 11-11 给出了冲突操作的三种情况，分别是读写冲突、写写冲突和写读冲突。通过对冲突操作的分析，可以发现：

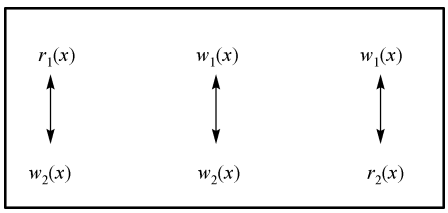


图 11-11　冲突操作

(1) 如果调度中一对连续操作是冲突的，则意味着当它们的执行顺序交换时，至少会改变其中一个事务的最终执行结果；

(2) 如果两个连续操作不冲突，则可以在调度中交换执行顺序而不改变原调度的执行结果，即交换执行顺序后的调度与原调度的执行结果等价。

图 11-12 给出了并发调度在不冲突操作上的一系列交换例子。每一次交换的连续操作都是不冲突的(三次交换的操作对操作的都是不同的数据对象)。通过三次交换，最后发现该并发调度交换成了一个串行调度(T_1T_2)，这意味着该并发调度的执行结果和串行调度 T_1T_2 是等价的，因此此调度是可串化调度。

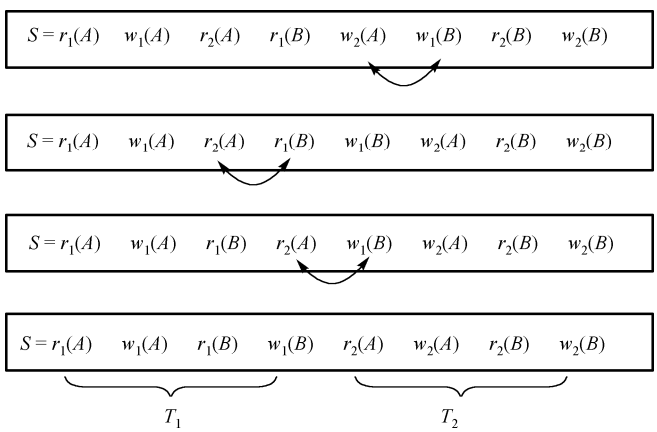


图 11-12　不冲突操作上的连续交换

图 11-13 给出了不可交换的事务操作的例子。其中，$r_1(A)$ 和 $w_1(A)$ 是属于同一个事务

的操作，在调度时不能改变它们的执行顺序，因为一个事务内部的操作顺序代表了事务的执行语义，这种语义是由用户给出的，DBMS 不能对它进行修改。$w_1(B)$和 $r_2(B)$是一对冲突操作，如果对它们进行执行顺序交换，则至少一个事务的执行结果会发生变化，也就是说，交换后的调度与原调度就不可能等价。在图中，如果 $w_1(B)$和 $r_2(B)$交换，则事务 T_2的执行结果将发生改变。在交换之前，事务 T_2的最终结果由 $w_2(A)$和 $w_2(B)$决定，其中 $w_2(A)$的结果跟交换操作没有关系，$w_2(B)$的结果是由 $r_2(B)$读入并修改后决定的，而 $r_2(B)$读入的数据是 $w_1(B)$写入的 B 值。但在交换后，$r_2(B)$读入的数据变成了调度开始时的数据库初始值，由于 $r_2(B)$和 $w_2(B)$之间的事务操作语义不会变化，因此 $r_2(B)$读入的数据变化必将导致 $w_2(B)$写入的 B 值也发生改变，从而导致最终的数据库状态与交换之前不一样。

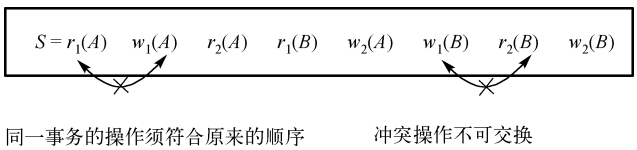


图 11-13　不可交换的事务操作示例

下面给出冲突可串性的准确定义。

定义 11-3　给定 N 个事务的并发调度 S_1和 S_2，如果调度 S_1通过交换一系列不冲突的操作，最终能够转换为调度 S_2，则称 S_1和 S_2冲突等价。

定义 11-4　给定 N 个事务的一个并发调度 S_1，如果 S_1冲突等价于这 N 个事务的某个串行调度，则称 S_1是冲突可串调度，即 S_1满足冲突可串性。

定理 11-1　如果一个并发调度 S_1满足冲突可串性，则 S_1必定是可串化调度。

注意，上面的定理只是一个充分性定理，反之不成立。事实上，冲突可串调度只是可串化调度的一个子集，因此并非所有可串化调度都是冲突可串的。但是，冲突可串性在冲突操作的基础上定义了一类具有特殊性质的可串化调度，并且可以用高效的方法来判断某个调度是否冲突可串。

冲突可串性的判断使用了称为优先图(Precedance Graph)的方法。给定一个并发调度 S，S 的优先图 $P(S)$是一个有向图，其中：

(1) 图的结点(Node)是事务；

(2) 如果结点 T_i和 T_j满足下面的条件，则 T_i和 T_j之间存在有向边(Arc) $T_i \rightarrow T_j$：存在 T_i中的操作 A_1和 T_j中的操作 A_2，满足 A_1在 A_2前，并且 A_1和 A_2是冲突操作。

定理 11-2　给定一个并发调度 S 及其优先图 $P(S)$，如果 $P(S)$不存在回路，则 S 满足冲突可串性。

图 11-14 和图 11-15 分别给出了两个优先图的例子。其中，图 11-14 中的优先图不存在回路，因此是冲突可串的，而且与它冲突等价的串行调度为 $T_1T_2T_3$。图 11-15 中的优先图存在回路，因此该调度不是冲突可串的。

优先图的基本思想是：如果一个调度 S 是冲突可串的，那么在与它等价的串行调度中，必然要满足所有有向边定义的事务执行顺序。例如，图 11-15 中 T_1到 T_2的有向边说明，如果调度 S 是冲突可串的，那么与该调度等价的串行调度中 T_1肯定在 T_2的前面。但是同时

还有 T_2 到 T_1 的有向边，说明如果存在与 S 等价的串行调度，在串行调度中 T_2 肯定在 T_1 的前面。这与前面的“T_1 肯定在 T_2 的前面”相互矛盾，因此 S 不可能是冲突可串调度。

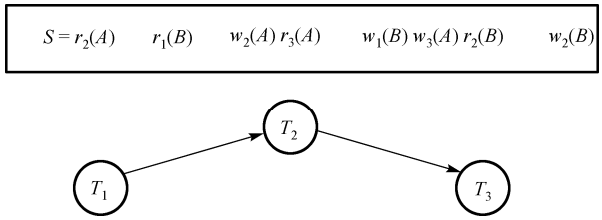


图 11-14　一个冲突可串调度及其优先图

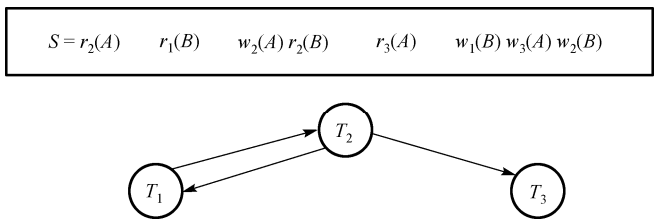


图 11-15　不冲突可串的调度及其带回路的优先图

11.4　基于锁的并发控制机制

锁机制是目前实现并发控制的主要方法之一。现有的商用 DBMS 一般都采用了基于锁的并发控制机制。锁机制的基本思想是通过对数据对象的加锁操作来避免并发冲突。如果所有事务都遵循所使用时的特定规则，那么 DBMS 可以保证事务的并发调度不会破坏数据库的一致性。需要注意的是，11.3 节讨论的冲突可串性理论只能解决并发调度是否正确的问题，无法直接满足 DBMS 的并发控制需求。对于 DBMS 来说，针对多事务并发执行的环境得到一种正确的并发调度才是最终的并发控制目标。

11.4.1　锁机制简介

图 11-16 给出了基于锁的并发控制机制示意图。在基于锁的并发控制机制中，调度器对每个事务都增加了加锁(Lock)和解锁(Unlock)操作，并且要求事务和调度都必须遵循基本的使用规则。

一般用下面的符号来表示事务的加锁和解锁动作。

(1) $L_i(A)$：表示事务 T_i 对数据对象 A 加锁。

(2) $U_i(A)$：表示事务 T_i 释放数据对象 A 上的锁。

加锁和解锁动作增加了事务执行的额外代价，因为锁表如果太大，则只能存储到磁盘上，从而使得加锁和解锁动作带来额外的 I/O 操作，降低了事务执行的时间性能。但这么做的好处是可以保证并发调度的一致性。

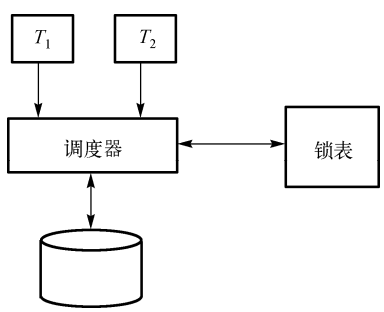


图 11-16　基于锁的并发控制机制示意图

基于锁的并发控制机制需要遵循一些使用锁的基本规则。这些锁的规则称为锁协议(Lock Protocol)。基本的锁协议包括下面两条。

1. 合法事务规则

合法事务规则要求每个事务 T_i 在操作数据对象 A 之前必须先执行加锁操作 $L_i(A)$ 获得 A 上面的锁，在操作完成后必须执行解锁操作 $U_i(A)$ 释放 A 上面的锁，即每个事务 Ti 都必须是下面的形式(其中 $P_i(A)$ 表示对 A 的读写操作)：

$$T_i：\cdots L_i(A)\cdots P_i(A)\cdots U_i(A)\cdots$$

2. 合法调度规则

合法调度规则要求在每个并发调度中，如果事务 T_i 已经持有了数据对象 A 上面的锁，那么在 T_i 释放 A 上面的锁之前，其他事务不得再获得 A 上的锁，即每个调度 S 必须满足图 11-17 所示的形式。

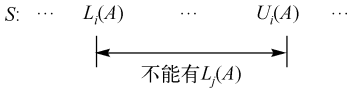


图 11-17　合法调度规则示意

例如，图 11-18 给出了调度 S 在应用锁机制后的调度例子。这个调度虽然满足合法事务规则，但不满足合法调度规则，因为 $L_2(B)$ 操作违背了合法调度规则。

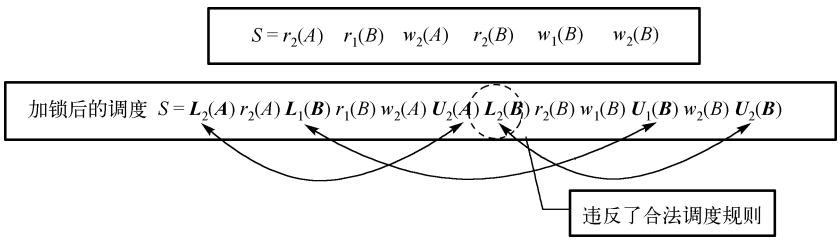


图 11-18　违反了合法调度规则的不正确调度

11.4.2　两阶段锁

目前商用 DBMS 中广泛使用的锁协议称为两阶段锁协议(Two-Phase Locking Protocol)，或称为 2PL 或两阶段锁。两阶段锁是针对单个事务规定的锁协议，其基本规则为：

(1) 事务在对任何数据进行读写之前，首先要获得该数据上的锁；

(2) 在释放一个锁之后，事务不再获得任何锁。

也就是说，遵循两阶段锁的事务必须满足图 11-19 所示的加锁解锁规则。

图 11-19　满足 2PL 的事务加锁和解锁规则

如果把一个事务持有的锁数目作为 Y 轴，事务运行时间作为 X 轴，那么遵循两阶段锁的事务持有的锁就会呈现图 11-20 所示的图形。从图中可以看到，遵循两阶段锁的事务在锁请求上具有两个明显的阶段：第一个阶段称为增长阶段(Growing Phase)，在此阶段事务只请求锁而不释放锁，因此锁的数目一直是增长的；第二阶段称为收缩阶段(Shrinking Phase)，在此阶段事务只释放锁而不请求锁，因此锁的数目是逐步减少的。正是因为遵循

两阶段锁的事务具有这样明显的两阶段加锁解锁特征，所以把这一锁协议称为两阶段锁协议。把遵循两阶段锁协议的事务称为两段式事务。

定理 11-3　如果一个调度 S 中的所有事务都是两段式事务，则该调度是可串化调度。

定理 11-3 为 DBMS 实现并发控制提供了理论基础。现在，DBMS 只要保证到达的每个事务都是两段式事务，就可以保证并发调度的正确性。正是因为两阶段锁具有这一特性，所以成为目前 DBMS 中流行的并发控制技术。

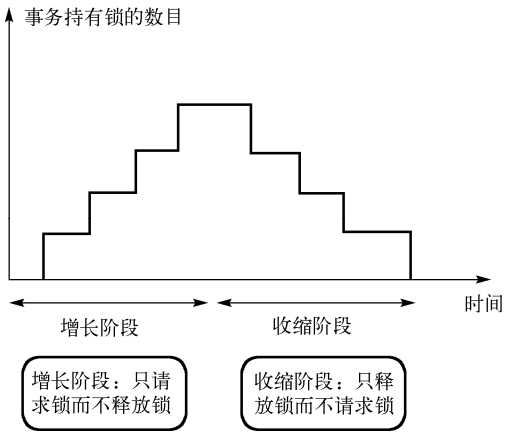


图 11-20　2PL 的增长阶段和收缩阶段

2PL 在实际实现中有两个变种，分别为严格两阶段锁(Strict 2PL，S2PL)和强两阶段锁(Strong Strict 2PL，SS2PL)。强两阶段锁的另一个名称是 Rigorous 2PL。S2PL 协议除了满足 2PL 的要求外，还要求 X 锁不能在事务提交(或撤销)之前释放，即必须保留到事务结束。SS2PL 在 2PL 基础上，要求 S 锁和 X 锁均要保留到事务结束才能释放。

1．X 锁

两阶段锁保证了并发调度的正确性，但是两阶段锁要求事务在请求数据 A 上的锁时，如果已有别的事务持有了 A 上的锁，则必须等待。这一规定与现实应用中的数据存取特点有冲突。在现实应用中，通常如果多个事务同时对同一个数据进行写操作，则要求只能有一个事务持有该数据上的锁，否则会出现丢失更新等问题；但如果多个事务同时对同一个数据进行读操作，则应当允许这些事务都可以获得该数据上的锁，从而保证事务读操作的高并发性，提高整体响应性能，即写操作应当是独占的，而读操作应当是共享的。这是因为并发的读操作不会修改数据，当然也不可能破坏数据库的一致性。

为了实现对读写操作不同的加锁策略，在数据库系统中引入若干不同类型的锁。其中，最常用的锁有两种：X 锁和 S 锁。

X 锁(Exclusive Locks)也称为写锁、排他锁或独占锁。X 锁对事务的写数据操作进行了约束。若事务 T 对数据 R 加 X 锁，那么其他事务要等 T 释放 X 锁以后，才能获准对数据 R 进行封锁。只有获得 R 上的 X 锁的事务，才能对所封锁的数据进行修改。

把 X 锁协议称为 PX 协议。PX 协议规定：

任何企图更新数据 R 的事务 T_i 必须先执行 $xL_i(R)$ 操作，以取得 R 上的 X 锁。如果未获得 X 锁，则事务进入等待状态，一直到获得 X 锁才能继续执行。

PX 协议的一个增强版本称为 PXC 协议。PXC 协议规定：

事务在遵循 PX 协议的基础上，所持有的 X 锁必须要保留到事务结束(Cmmit 或 Abort)。

图 11-21 给出了一个不正确的并发调度。图 11-22 给出了对图 11-21 所示的调度应用 X 锁后的并发调度。通过应用 X 锁，消除了原调度中存在的丢失更新问题，最终得到了一个

正确的并发调度。从图 11-22 可以看到，采用 X 锁解决调度中的丢失更新问题并不一定要求事务都遵循两阶段锁协议。

T_1	T_2	A	B
Read(A, t)		25	25
$t = t + 100$			
Write(A, t)		125	25
	Read(A, t)	125	25
Read(B, t)	$t = t * 2$	125	25
$t = t + 100$	Write(A, t)	250	25
		250	25
	Read(B, t)	250	25
Write(B, t)	$t = t * 2$	250	125
	Write(B, t)	250	50

图 11-21　一个不正确的并发调度

T_1	T_2	A	B
xL$_1$(A)			
Read(A, t)		25	25
$t = t + 100$			
Write(A, t)		125	25
***U*$_1$(A)**			
	xL$_2$(A)		
xL$_1$(B)	Read(A, t)	125	25
Read(B, t)	$t = t * 2$		
$t = t + 100$	Write(A, t)	250	25
	***U*$_2$(A)**		
	xL$_2$(B)	250	25
	Wait		
	Wait		
Write(B, t)	**Wait**	250	125
***U*$_1$(B)**	**Wait**		
	Read(B, t)	250	125
	$t = t * 2$		
	Write(B, t)	250	250
	***U*$_2$(B)**		

图 11-22　应用了 X 锁后的并发调度

图 11-23 显示了基于两阶段锁协议和 X 锁协议得到的并发调度结果。通过对比可以发现，采用两阶段锁协议时事务的并发性会受到一定的影响。

T_1	T_2	A	B
$xL_1(A)$			
Read(A, t)		25	25
$t = t + 100$			
Write(A, t)		125	25
	$xL_2(A)$		
$xL_1(B)$	**Wait**		
Read(B, t)	**Wait**	125	25
$t = t + 100$	**Wait**		
Write(B, t)	**Wait**	125	125
$U_1(A)$	**Wait**		
$U_1(B)$	Read(A, t)	125	125
	$t = t * 2$		
	Write(A, t)	250	125
	$xL_2(B)$		
	$t = t * 2$		
	Write(B, t)	250	250
	$U_2(A)$		
	$U_2(B)$		

图 11-23　基于两阶段锁协议和 X 锁协议的并发调度结果

一般来说，X 锁协议对同一数据的写写冲突做出了严格的限制，因此可以避免并发操作中的丢失更新问题。但是，其他的问题，包括脏读、不可重复读等，是由于读写冲突而导致的，即使 DBMS 采用了基于 X 锁的 2PL，也不能完全避免这些问题的出现。

2. S 锁

X 锁提供了对事务写操作的控制，但 X 锁具有独占性质——任何一个数据 *A* 上同时只能有一个事务持有 X 锁。如果多个事务对数据 *A* 同时进行读操作，则在现实应用中一般允许这些事务同时执行，因为读操作不会破坏数据库的一致性。

由于 X 锁并没有区分事务的读写操作，因此引入另一种锁，即 S 锁，对事务的读操作进行约束。

S 锁(Share Locks)也称为读锁或共享锁。S 锁是在 X 锁之上提出的，如果事务 *T* 对数据 *R* 加了 S 锁，则其他事务对 *R* 的 X 锁请求不能成功，但对 *R* 的 S 锁请求可以成功。这就保证了其他事务可以读取 *R* 但不能修改 *R*，直到事务 *T* 释放 S 锁。

S 锁协议称为 PS 协议。PS 协议的规定如下：

任何事务 T_i 要读取数据 *R*，必须先执行 $sL_i(R)$ 操作，以获得 *R* 上的 S 锁。如果未获得 S 锁，则事务进入等待状态，直到获得 S 锁才能继续执行。当事务获得 S 锁后，如果要对数据 *R* 进行修改，则必须在修改前执行 Upgrade(*R*)操作，将 S 锁升级为 X 锁。

PS 协议的增强版本 PSC 协议规定：

事务在遵循 PS 协议的基础上，所持有的 S 锁必须保持到事务结束(Commit 或 Abort)。

如果在两阶段锁协议的基础上引入 X 锁和 S 锁，则既可以保证并发调度的正确性，也可以对事务的读写操作进行有针对性的控制。基于 X 锁和 S 锁的 2PL 协议可以归纳为以下几点。

(1) 事务在读取数据 R 前必须先获得 R 上的 S 锁。

(2) 事务在更新数据 R 前必须要获得 R 上的 X 锁。如果该事务已具有 R 上的 S 锁，则必须将 S 锁升级为 X 锁。

(3) 如果事务对锁的请求因为与其他事务已具有的锁不相容而被拒绝，则事务进入等待状态，直到其他事务释放锁。

(4) 一旦事务释放一个锁，就不再请求任何锁。

锁的相容性是指锁之间是否相互排斥不能共存。如果两个锁是相容的，则它们可以同时加在同一个数据上。如果两个锁不相容，则不能同时共存于同一个数据上。对于 S 锁和 X 锁而言，X 锁和任何其他锁都是不相容的(这也是 X 锁称为排他锁的原因)，而 S 锁和 S 锁是相容的(这也是 S 锁称为共享锁的原因)。表 11-1 给出了涉及 X 锁和 S 锁的相容性矩阵。

表 11-1　涉及 X 锁和 S 锁的相容性矩阵

持有锁的事务	请求锁的事务		
	X 锁	S 锁	无
X 锁	否	否	是
S 锁	否	是	是
无	是	是	是

3．U 锁

S 锁规定在读数据时先请求 S 锁，如果读完数据后要继续修改，则需要 Upgrade 到 X 锁。可以发现，S 锁的这种锁协议在两个事务同时用 Update 语句更新数据时很容易导致死锁。

图 11-24 给出了 S 锁升级到 X 锁导致的死锁例子。由于 T_1 和 T_2 一开始都读数据 A，所以两个事务都持有了 A 上的 S 锁。接下来，当 T_1 想 Upgrade(A)时，由于 T_2 持有 A 上的 S 锁，所以 T_1 进入 Wait 状态。之后，T_2 想升级 S 锁到 X 锁，同样由于 T_1 还持有 A 上的 S 锁，所以 T_2 也进入了 Wait 状态。最后，从第 7 个时间点开始，两个事务都进入了 Wait 状态，出现了死锁。

时间	T_1	T_2
1	**sL$_1$(A)**	
2		**sL$_2$(A)**
3	Read(A, t)	
4	t=t+100	Read(A, t)
5	Upgrade(A)	t=t+100
6	Wait	Upgrade(A)
7	Wait	Wait
8	…	…

图 11-24　S 锁升级到 X 锁导致的死锁例子

DBMS 中的死锁问题本身并不可怕，因为并发事务出现死锁时，事务的更新都还没有提交，因此还没有修改数据库的状态。但是，像图 11-24 这种死锁的情况，在更新频繁的应用中极容易出现。例如，通常一个企业只有一个对公银行账户；每天在同一个账户上都有很多进出账目，意味着同时执行 Update 语句的可能性非常高。按照图 11-24 的示例，一旦两个 Update 语句同时到达且交错并发执行，则大概率发生死锁。死锁发生后，事务执行被挂起，需要等到 DBMS 执行死锁检测并且解锁后才能继续执行。因此，频繁的死锁会极大地影响系统的整体性能。

为了解决多个 S 锁同时升级到 X 锁引发的频繁死锁问题，引入一种新的锁类型——U 锁。U 锁即更新锁（Update Lock）。将原先的 S 锁划分为两类：一类 S 锁只允许读数据，不允许以后升级到 X 锁，依然将这类 S 锁称为 S 锁；另一类 S 锁允许读取数据且允许以后升级到 X 锁，将这一类 S 锁赋予一个新的名称——U 锁。

如果事务取得了数据 R 上的 U 锁，则可以读 R，并且可以在以后升级为 X 锁。如果事务持有了 R 上的 U 锁，则其他事务不能得到 R 上的 S 锁、X 锁以及 U 锁。如果事务持有了 R 上的 S 锁，则其他事务可以获取 R 上的 U 锁。

表 11-2 给出了涉及 S 锁、X 锁和 U 锁的相容性矩阵。需要注意的是，锁的相容性矩阵并不是对称的。在表 11-2 中，S 锁和 U 锁的相容性是不对称的，表现在以下方面。

（1）<S, U>是相容的：表示如果 T_1 持有了 S 锁，则 T_2 接着请求同一数据上的 U 锁可以成功。由于 U 锁意味着先读后写，所以此时 T_2 可以先执行读数据操作，并且不会影响 T_1 的执行，从而提高多事务执行时的并发性。

（2）<U, S>是不相容的：表示如果 T_1 持有了 U 锁，则 T_2 接着请求同一数据上的 S 锁不能成功。这是因为当 T_2 获得了 S 锁，T_1 在之后想升级到 X 锁时，如果 T_2 还没有释放 S 锁，则 T_1 无法升级。这意味着先开始执行的事务 T_1 受到了后开始执行的事务 T_2 的影响。

表 11-2　涉及 S 锁、X 锁和 U 锁的相容性矩阵

持有锁的事务	请求锁的事务		
	S 锁	X 锁	U 锁
S 锁	是	否	是
X 锁	否	否	否
U 锁	否	否	否

如果 T_1 持有了数据 R 上的 U 锁，然后又将 R 上的 S 锁授予另一个事务，在最坏的情况下，T_1 有可能会永远无法升级到 X 锁。设想，如果 T_1 持有数据 R 上的 U 锁，T_2 请求了数据 R 的 S 锁，然后 T_1 升级到 X 锁不成功进入 Wait 状态；然后 T_2 释放了 R 上的 S 锁，但是在 T_1 重新请求升级锁之前，另一个事务 T_3 正好来请求 R 上的 S 锁；此时，T_3 会获得 R 上的 S 锁，导致 T_1 继续 Wait。极端情况下，如果总是有一个新的事务在上一个事务释放了 R 上的 S 锁之后马上请求 R 的 S 锁，最终导致最早开始的事务 T_1 永远无法升级到 X 锁。这种情况与多个事务之间互相等待锁形成的死锁不同，在并发控制中称为“活锁”（Live Lock）。

图 11-24 所示的死锁情形可以在采用 U 锁后得以解决。图 11-25 给出了基于 U 锁的调度序列。可以看到，由于 T_1 和 T_2 都需要在后面升级到 X 锁，所以两个事务一开始就需要请求 U 锁。

时间	T_1	T_2
1	**uL$_1$(*A*)**	
2		**uL$_2$(*A*)**
3	Read(*A*, *t*)	Wait
4	*t*=*t*+100	Wait
5	Upgrade(*A*)	Wait
6	Write(*A*, *t*)	Wait
7	***U*$_1$(*A*)**	Wait
8		Read(*A*, *t*)
9		…

图 11-25　基于 U 锁的调度序列

11.4.3　多粒度锁与意向锁

在前面讨论的加锁操作中，并没有特别强调 Lock(*A*)时数据 *A* 究竟是什么内容。*A* 称为加锁对象。在数据库中，加锁对象可以是整个数据库，也可以是关系、块、元组、索引、索引项等。把加锁对象的大小称为“锁粒度”(Lock Granularity)。

一般地，锁粒度越细，事务的并发度就越高；锁粒度越粗，并发度越低。例如，如果锁粒度为关系，则事务 T_1 若要删除关系 R_1 的某个元组 t_1，需要对 R_1 加 X 锁，此时任何其他事务都无法再对关系 R_1 进行读写。但是，如果锁粒度为元组，则上述情况下 T_1 只需要对元组 t_1 加 X 锁，此时其他事务可以同时读写除 t_1 之外的任何元组，事务操作的并发度明显得到了提升。

1. 多粒度锁的概念

多粒度锁是指加锁的时候同时支持按不同的锁粒度进行加锁。数据库中基本的锁粒度从大到小包括数据库、关系、磁盘块和元组，而且上一级锁粒度与下一级锁粒度之间存在着包含关系。因此，可以将数据库中不同的锁粒度表示为一棵多粒度树，如图 11-26 所示。

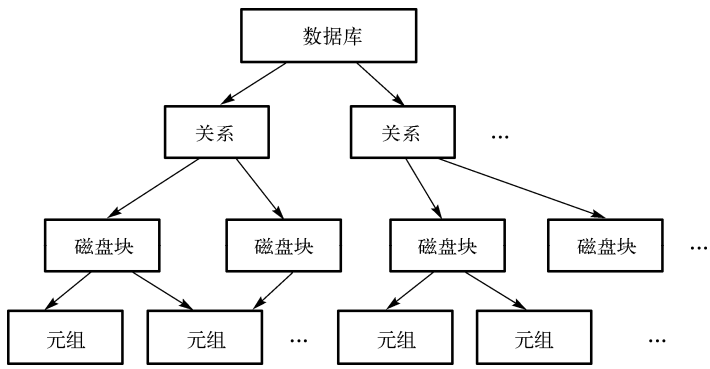


图 11-26　多粒度树

DBMS 支持多粒度锁，意味着支持对多粒度树的每个结点独立加锁。由于上一级锁粒度包含了下一级锁粒度对应的数据对象，因此可以分析得到：如果对某个结点加了锁，意味着以该结点为根的所有下层结点都被加了锁。

2. 多粒度锁的动机

引入多粒度锁之后，很显然加锁操作以及锁管理器都变得更为复杂。那么，能不能不实现多粒度锁，仅用单粒度锁来满足 DBMS 的加锁需求呢？答案显然是不能。下面通过一个例子来说明为什么现在的 DBMS 都需要实现多粒度锁。

假设现有两个银行信息系统中的事务 T_1 和 T_2。其中 T_1 的功能是求当前数据库中所有账户的余额之和，T_2 的功能是增加一个新的账户(假设余额为 1000 元)。如果只支持单粒度锁，最合适的锁粒度就是元组，因为 SQL 中的 DML 操作本身就是针对记录的操作。

假设数据库中现有两个账户 O_1 和 O_2，图 11-27 给出了事务 T_1 和 T_2 的一个调度序列。按照锁协议，T_1 在读 O_1 和 O_2 之前都请求了 S 锁。但是 T_2 因为是新增一个账户 O_3，在元组锁的前提下无法在 Write 操作之前请求锁。原因很简单，加锁操作必须针对已存在的数据对象，但 Write 操作之前 O_3 根本不存在，所以无法执行加锁操作。如此一来，T_2 就无法加锁。图中 T_2 的 Write(O_3)操作在第 5 个时间点提交了，然后 T_1 在第 6 个时间点求出了账户余额之和 Sum。此时 T_1 输出的 Sum 并非数据库的实际状态，因为在第 6 个时间点数据库中实际已经有 3 个账户而不是 Sum 所求的 2 个账户。

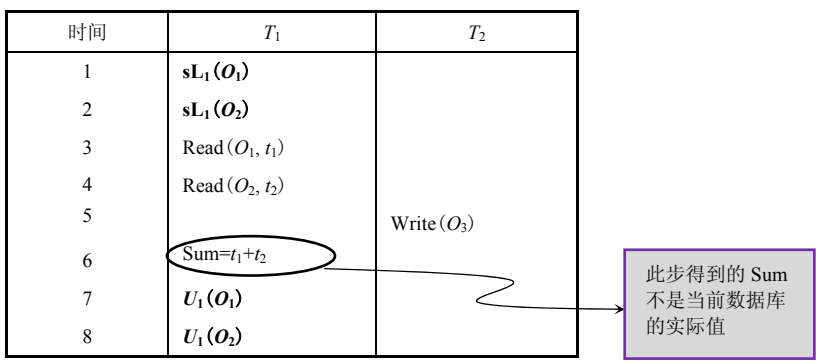


图 11-27　仅支持元组粒度的锁无法避免幻象元组

上述问题出现的根本原因在于加锁操作无法对 T_2 要插入的 O_3 对象进行加锁。在数据库中，像 O_3 这种无法在操作之前被预先加锁的元组称为幻象元组(Phantom Tuple)。幻象元组的存在会导致数据库并发操作时出现不一致分析问题，使得读到的汇总数据与当前数据库中的实际状态不符。

解决图 11-27 中问题的思路是确保 T_2 执行 Write(O_3)时能够将数据对象锁住，为此可以将 T_2 插入 O_3 的操作看成对整个关系的一次写操作，从而在操作之前对整个关系加锁，如图 11-28 所示。这意味着不仅需要支持元组锁，也需要支持关系粒度的锁，即支持多粒度锁。

3. 多粒度锁的新挑战

DBMS 引入了多粒度锁之后，发现现在数据对象上的加锁存在两种不同的方式，即显式加锁和隐式加锁。

(1) 显式加锁：应事务的请求直接加到数据对象上的锁。

时间	T_1	T_2
1	**sL$_1$(O_1)**	
2	**sL$_1$(O_2)**	
3	Read(O_1, t_1)	
4	Read(O_2, t_2)	**xL$_2$(R)**
5		Wait
6	Sum=t_1+t_2	Wait
7	**U_1(O_1)**	Wait
8	**U_1(O_2)**	Wait
9		Write(O_3)

图 11-28　使用多粒度锁解决幻象元组加锁问题

(2) 隐式加锁：数据对象本身没有被显式加锁，但因为其上层结点加了锁而被加锁。

因此，当事务请求给一个结点显式加锁时必须考虑几种情况。

(1) 该结点是否已有不相容锁存在？

(2) 上层结点是否已有不相容的锁(上层结点导致的隐式锁冲突)？

(3) 所有下层结点中是否存在不相容的显式锁？

图 11-29 给出了一个例子。事务 T_1 对关系 R_1 显式加了 S 锁，意味着 R_1 的所有元组被隐式加了 S 锁。其他事务可以在 R_1 的元组上加 S 锁，但不能加 X 锁。例如，图中事务 T_3 请求对元组 t_1 加 X 锁时只能等待。注意，此时元组 t_1 上并没有显式的锁存在，完全是因为其上层结点关系 R_1 被事务 T_1 加了 S 锁而导致的隐式锁冲突。此外，图中事务 T_2 对元组 t_3 加了 X 锁，其他事务不能请求对其上层结点 R_n 的 S 锁或 X 锁。事务 T_4 请求对 R_n 加 X 锁也只能等待。这是由于 R_n 的下层结点 t_3 中已经存在着与其不相容的锁。

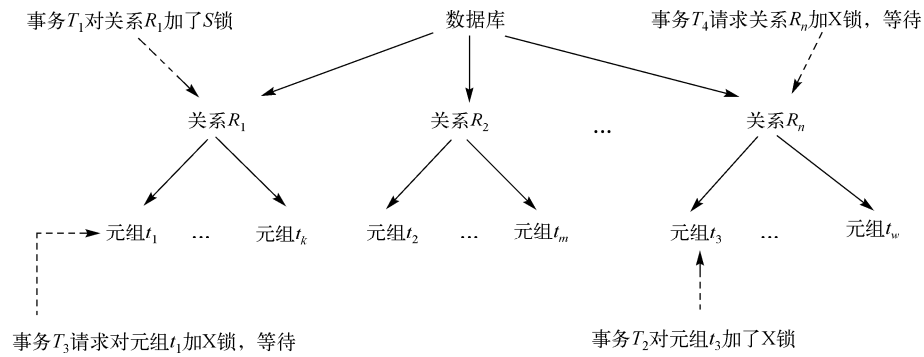


图 11-29　多粒度锁带来的显式加锁和隐式加锁问题

从而，当事务请求对一个结点 P 加锁时，DBMS 必须判断该结点上是否存在不相容的锁。这个不相容的锁有可能是结点 P 上的显式的不相容锁，也有可能是 P 的上层结点导致的隐式不相容锁，还有可能是 P 的下层结点中已存在的某个显式不相容锁。

理论上 DBMS 需要搜索上面全部的可能情况，才能确定 P 上的锁请求能否成功。这里的搜索代价显然是巨大的。例如，假设结点 P 是关系 R_1，R_1 有 50 万个磁盘块和 500 万个元组，则请求加锁时需要搜索 50 万个磁盘块和 500 万个元组上的锁信息。在实际 DBMS

实现中，锁表通常是以链表的形式组织的。在链表中搜索 550 万个元素信息的时间代价是难以接受的(单纯的 C 程序循环执行 500 万次需要的时间一般大于 1s)，因为请求加锁的过程只是用户发出 SQL 查询到返回查询结果之间的一个中间过程，如果这一过程花费了太多的时间，很显然最终会极大地降低整个数据库系统的性能。

为了解决多粒度锁中请求结点加锁时的高搜索代价问题，人们提出了一种新的锁机制——意向锁。因此，意向锁与多粒度锁是捆绑实现的：如果要支持多粒度锁，就需要实现意向锁以保证加锁性能；如果不支持多粒度锁，则不需要实现意向锁。

4. 意向锁

如前所述，引入意向锁的主要目的是提高多粒度锁中结点请求加锁时的效率。基本的意向锁有两种：

(1) IS 锁(Intent Share Lock，意向读锁)；

(2) IX 锁(Intent Exclusive Lock，意向写锁)。

意向锁协议规定：

(1) 如果对某个结点加 IS(IX)锁，则说明事务要对该结点的某个下层结点加 S(X)锁。这也是意向锁名词的来由——它表示了对下层结点加锁的意向；

(2) 对任一结点 P 加 S(X)锁，必须先对从根结点到 P 的路径上的所有结点加 IS(IX)锁。这一条使得在多粒度树上请求对某个结点加锁时，只需要扫描从根结点到目标结点的路径即可。由于多粒度树一般只有四层，所以这一路径只需要不多于四次的计算即可实现。

图 11-30 显示了在多粒度树(此处省略了最上层的数据库结点)上对元组 t_1 请求加 X 锁的过程。首先，对结点 R 请求加 IX 锁，因为现在 R 上没有任何不相容的锁，所以 R 结点上的 IX 锁加锁成功。接着，对结点 B_1 请求加 IX 锁，成功。最后，到达了结点 t_1，此时请求对 t_1 加 X 锁。由于假设多粒度树之前没有任何锁，所以最终 t_1 上的 X 锁也加锁成功。

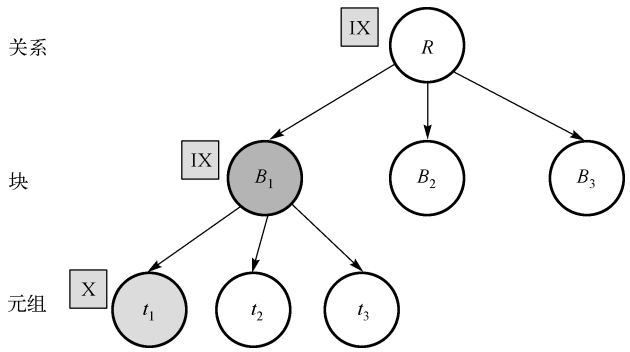


图 11-30　在多粒度树上按照意向锁协议对结点 t_1 加 X 锁

在图 11-30 的基础上，继续对结点 B_1 请求加 S 锁。图 11-31 显示了加锁的过程，其中结点 R 上加 IS 锁成功，然后到达结点 B_1 时请求加 S 锁，但 S 锁与 B_1 上已有的 IX 锁不相容，所以 B_1 请求加 S 锁没有成功。

图 11-30 和图 11-31 的加锁过程表明，按照意向锁协议对多粒度树上的结点请求加锁

时，只需要遍历从根结点到目标结点的路径上的结点，从而可以将请求加锁的判断代价控制在 $O(1)$ 的时间复杂度里。

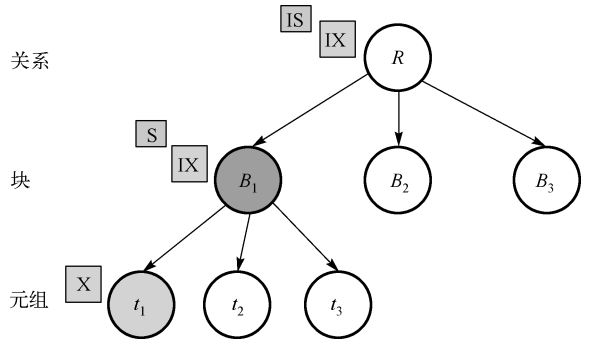


图 11-31　在多粒度树上使用意向锁对结点 B_1 请求加 S 锁

表 11-3 给出了涉及 IS 锁、IX 锁、S 锁和 X 锁的相容性矩阵。由于 DBMS 支持的锁类型有限，而且不同锁之间的相容性也是已知的，所以给定了锁类型之后，DBMS 就可以将对应的相容性矩阵直接存储在内核中。

表 11-3　涉及 IS 锁、IX 锁、S 锁和 X 锁的相容性矩阵

持有锁的事务	请求锁的事务			
	IS 锁	IX 锁	S 锁	X 锁
IS 锁	是	是	是	否
IX 锁	是	是	否	否
S 锁	是	否	是	否
X 锁	否	否	否	否

11.5　事务的隔离级别

在事务的 ACID 性质中，原子性(A)、一致性(C)和持久性(D)是 DBMS 必须保证百分百实现的性质，即要么有，要么没有。例如，对于持久性，要么持久化到磁盘，要么没能持久化到磁盘。如果没能持久化到磁盘，则认为事务的持久性没有实现。

另外，人们发现事务的隔离性(I)与其他三种性质有所不同。某些现实应用并不强制要求事务之间完全隔离。如果系统性能是应用的第一需求，而事务之间的隔离性排第二，那么通过放松对事务隔离性的要求，从而提高事务的并发性，进而提升性能，也是实际应用中的一种可行方案。这么做的原因是事务的完全隔离需要对事务实施严格的锁协议，导致事务之间的并发性降低，影响系统性能。

以银行的 ATM 应用为例，一种选择是当用户在 ATM 查询余额时，如果要求事务之间完全隔离，那么用户登录后无论查询多少次余额，结果都应该是统一的金额，实现的方式就是将账户上的 S 锁一直保持到事务解锁。此时，其他事务可以读取同一账户但无法进行修改，这意味着一些转账、自动扣款等业务无法在用户查询余额时并发执行，导致事务并发性低，系统吞吐下降。另一种选择是用户在查询余额时，不要求事务之间的完全隔离。

此时，若用户登录后执行多次同样的余额查询操作，可能会得到不同的金额，这意味着事务之间没有完全隔离，因为正在执行查询的事务，从中可以发现另外的事务正在修改账户余额。这种情况仅影响 ATM 应用的用户友好性。如果系统的性能重要性高于用户友好性(ATM 应用可以视作这样的例子：与漫长的等待时间相比，用户更愿意接受偶发的余额查询不一致性)，则采用这种不完全隔离的方案对于应用而言更合适。从锁机制角度分析，如果 DBMS 允许一个事务读取数据后即可释放 S 锁，就会出现上述的不完全隔离情形。这个例子表明，如果事务的隔离性与其他三种性质一样设计成非此即彼的 0-1 选择问题，用户就无法自行根据并发性、性能、用户友好性等因素来决定是否实施完全事务隔离。DBMS 必须提供对多种事务隔离级别的支持，才能使得用户可以根据实际应用来选择使用哪一种事务隔离级别。

ANSI SQL 通过定义四种事务隔离级别为用户提供了自定义事务隔离级别的支持。表 11-4 显示了这四种事务隔离级别，其中未提交读是最低的隔离级别，可串行读是最高的隔离级别。对于表中的事务隔离级别，需要注意以下两点。

(1) 事务的隔离级别是针对连接(会话)而设置的，不是针对单个事务。因此，一旦设置了某种隔离级别，在整个会话期间所有事务都将遵循该隔离级别。

(2) 事务的隔离级别只影响读操作，即只负责解决读引起的并发操作问题。这意味着写操作之间的冲突，如丢失更新问题，并不能通过设置事务隔离级别来解决。

表 11-4　SQL 中定义的四种事务隔离级别

隔离级别	并发操作问题		
	脏读	不可重复读	幻象
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

在不同的事务隔离级别下，事务的读操作受影响的程度有所不同。表 11-4 显示了不同事务隔离级别下出现三类读问题的可能性。当事务隔离级别为未提交读时，所有的读操作问题都会出现；而当隔离级别设置为可串行读时，三类读操作问题都可以避免。

目前，MySQL 默认采用的事务隔离级别为可重复读，而 Oracle 和 Microsoft SQL Server 默认的事务隔离级别为提交读。此外，MySQL 和 Microsoft SQL Server 都支持四种事务隔离级别，而 Oracle 只支持提交读和可串行读。

11.5.1　未提交读

当事务隔离级别设置为未提交读(Read Uncommitted)时，事务允许读取当前数据页上的任何数据，不管数据是否已提交。此时，读事务不必等待任何锁，也不需要对读取的数据加锁。

未提交读只有理论意义，在实际应用中很少使用，但因为它不要求读数据时加锁，所以并发性能相比其他几种事务隔离级别最高。

图 11-32 给出了未提交读的例子。由于连接 2 设置了未提交读隔离级别，所以第 3 步

和第 5 步的两次 Select 查询都不会请求 S 锁。因此第 5 步的 Select 查询会读入连接 1 在第 4 步更新的脏数据。由于连接 1 在第 4 步更新的数据未提交，所以一旦它在第 6 步回滚了事务，就导致连接 2 中读入的“王红”记录实际已经是错误的数据，这就是脏读带来的后果。

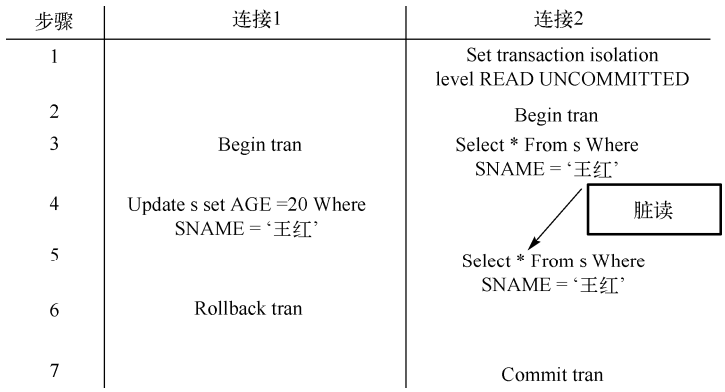


图 11-32　未提交读隔离级别示例

11.5.2　提交读

提交读(Read Committed)的含义是事务只能读到其他事务提交的数据，即不能读其他事务修改了但未提交的数据。因此，当事务隔离级别设置为提交读时，可以防止出现脏读问题。提交读要求事务必须在所读取的数据上加 S 锁，但是数据一旦读出，允许马上释放事务持有的 S 锁。提交读也隐含了要求 X 锁必须保持到事务提交，即满足 S2PL 协议。这也是实际 DBMS 普遍采用 S2PL 协议的主要原因。

图 11-33 演示了提交读隔离级别的例子(以 Microsoft SQL Server 为例，与 MySQL 的差别在于事务命令语句格式不同)。在第 3 步时，连接 2 中的事务执行 Select 查询，意味着将“王红”记录加了 S 锁。因为此时其他事务没有对姓名为“王红”的记录加锁，所以这一步加 S 锁成功。注意，如果此时有别的事务正在修改“王红”这条记录，连接 2 对“王红”请求 S 锁将不成功，从而可以避免读到脏数据。接着，在第 4 步，连接 1 要修改“王红”

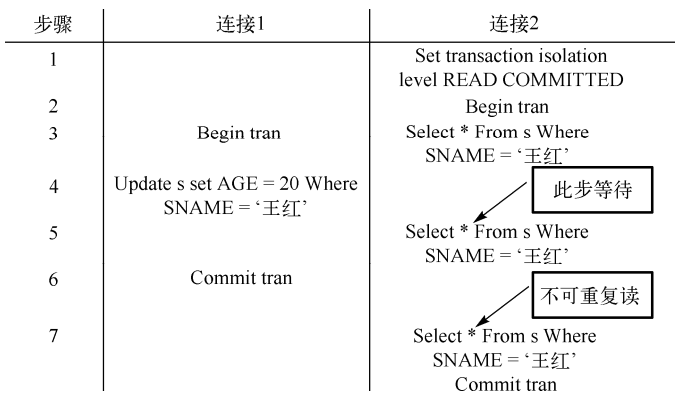


图 11-33　提交读隔离级别示例

记录，此步需要请求 X 锁。由于第 3 步连接 2 请求的 S 锁已经释放，所以连接 1 可以获得“王红”记录上的 X 锁。第 5 步连接 2 再次查询时，由于连接 1 对“王红”记录加的 X 锁还未释放，所以连接 2 请求 S 锁不能成功，进入等待状态。之后第 6 步连接 1 提交事务。第 7 步连接 2 再次执行同一查询时，其结果与第 3 步中的第一次查询结果不相同，因为第 3 步查询得到的元组也已经被第 4 步连接 1 的 Update 操作修改了，出现了不可重复读问题。

11.5.3　可重复读

可重复读(Repeatable Read)保证事务在事务内部如果重复访问同一数据(记录集)，数据不会发生改变，即事务在访问数据时，其他事务不能修改正在访问的那部分数据。可重复读可以防止脏读和不可重复读，但不能防止幻像。在可重复读隔离级别下，事务必须在所访问数据上加 S 锁，以防止其他事务修改数据，而且 S 锁必须保持到事务结束。

与提交读相比，可重复读主要防止了不可重复读问题的出现。特别需要注意的是，可重复读只保证事务一开始执行的 Select 查询所限定的元组集合全部用 S 锁锁定，在事务结束之前都不会被其他事务修改。这并不意味着在事务内部重复执行同一条 Select 语句总是能够返回相同的元组集合。这是因为可重复读无法防止幻象元组的出现。一旦幻象元组满足了事务内部 Select 语句的查询条件，也会被 Select 语句返回，从而造成同一 Select 语句在事务内部重复执行时返回的结果不同。例如。事务第一次执行 Select 语句时，按条件 Age=20 返回了 R_1 和 R_2 两条记录，可重复读保证这两条记录在事务结束之前不会被任何事务修改，但可以在事务内部重复读取。但是，如果另一个事务新插入一条记录 R_3，并且 R_3 的 Age 值也为 20，则 R_3 也满足 Select 语句中的条件 Age=20。此时，事务如果再次执行同一 Select 语句，将返回 R_1、R_2 和 R_3 三条记录。虽然结果不同，但依然满足可重复读隔离级别的要求，因为 R_1 和 R_2 这两条记录没有被修改。

图 11-34 给出了可重复读的示例。连接 2 的事务隔离级别设置为了可重复读。第 3 步

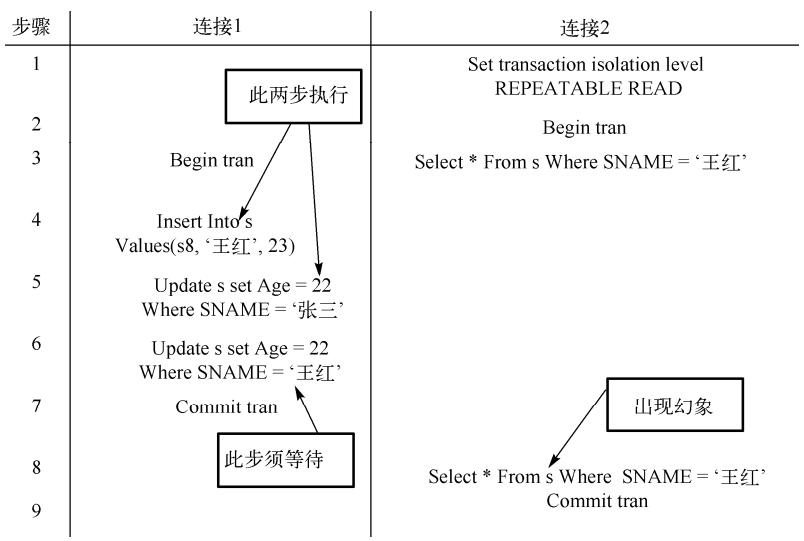


图 11-34　可重复读隔离级别示例

连接 2 执行 Select 查询，意味着对“王红”记录加了 S 锁并且锁会持有到事务提交。因此，当连接 1 在第 6 步想更新“王红”记录时无法获得 X 锁，只能等待。但是，连接 1 的第 4 步和第 5 步得以执行，其中第 5 步因为更新的并不是“王红”记录，所以加锁对象不同，锁没有冲突，可以正常获得锁并执行，第 4 步插入了幻象元组。最终，连接 2 的第 8 步再次执行同一 Select 语句时将出现幻象问题。

11.5.4　可串行读

可串行读(Serializable Read)可以保证事务调度是可串化的。事务在访问数据时，其他事务不能修改数据，也不能插入新元组(从而避免出现幻象)。事务必须在所访问数据上加 S 锁，防止其他事务修改数据，而且 S 锁必须保持到事务结束。此外，事务还必须锁住访问的整个表。

图 11-35 显示了可串行读的例子。当连接 2 设置为可串行读隔离级别后，连接 1 在第 4 步企图插入幻象元组时失败，因为连接 2 中的事务对整个表也加了锁。

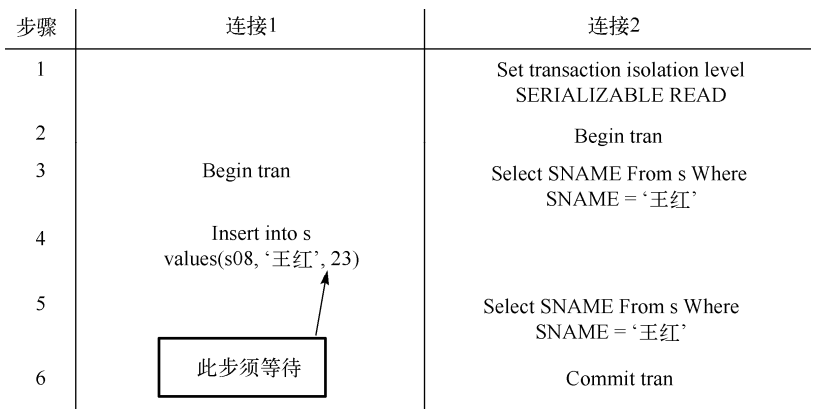


图 11-35　可串行读隔离级别示例

11.6　死　　锁

基于锁的并发控制机制虽然能够解决并发操作带来的问题，但也有一些副作用。其中最主要的一个副作用就是死锁问题。

DBMS 中的死锁有两种情形(图 11-36)。

(1)事务 T_1 持有 A 上的锁且请求 B 上的锁，事务 T_2 持有 B 上的锁且请求 A 上的锁，即两个事务分别持有了对方请求的锁，同时又请求对方持有的锁。

(2)事务 T_1 持有 A 的 S 锁，请求升级到 X 锁。事务 T_2 也持有 A 上的 S 锁，也请求升级到 X 锁。

无论哪一种情况，都会导致两个事务都无法获得执行后续操作所需的锁，从而都进入永远的等待状态。其中第二种情形可以采用 U 锁来解决，已经在 11.4.2 节进行了讨论，所以本节重点讨论第一种死锁情形的解决。

时间	T_1	T_2
1	**sL₁(*A*)**	
2		**sL₂(*B*)**
3	Read(*A*, *t*)	
4	*t*=*t*+100	Read(*B*, *w*)
5	**xL₁(*B*)**	*w*=*w*+100
6	Wait	**xL₁(*A*)**
7	Wait	Wait
8	…	…

时间	T_1	T_2
1	**sL₁(*A*)**	
2		**sL₂(*A*)**
3	Read(*A*, *t*)	
4	*t*=*t*+100	Read(*A*, *t*)
5	Upgrade(*A*)	*t*=*t*+100
6	Wait	Upgrade(*A*)
7	Wait	Wait
8	…	…

图 11-36　死锁的两种情形

需要强调的是，死锁主要的缺点是影响系统性能，因为死锁的事务都会被挂起无法执行。通常需要 DBMS 采取一定的死锁检测策略(如超时机制)来发现死锁，然后将死锁的某个事务回滚并释放其持有的锁，从而使得其他事务得以继续执行。但是，由于死锁的事务并未提交，因此事务的更新并未持久生效，并不会影响数据库的最终状态，也就是说，不会破坏数据库的一致性。从另一个角度看，DBMS 的并发控制的首要目标是保证数据库的一致性，所以往往一些措施与 DBMS 的其他目标(如性能)背道而驰。因此，DBMS 需要从性能、一致性，以及本书后续要讨论的安全性等多个方面综合考虑，最终在各个方面进行折中，才能满足实际应用的需要。

DBMS 中的死锁处理策略主要有两种。第一种是死锁检测策略，即 DBMS 采用某种技术检测是否出现了死锁，如果出现死锁，则解锁。第二种是死锁预防策略，即在加锁阶段提前采取措施预防出现死锁。

11.6.1　死锁检测

死锁检测(Deadlock Detecting)的简单策略是采用超时(Timeout)机制，即监控每个事务的执行时间，如果某个事务的执行时间超过了某个阈值，则认为出现了死锁。此时，将该事务取消并释放其持有的锁。

超时机制实现简单，但主要的问题是适应性差。由于实际应用中的事务有长事务(执行时间长)和短事务(执行时间短)，所以很难设置合适的超时阈值。

另一种死锁检测策略是使用等待图(Waiting Graph)。等待图的思路是使用一个有向图存储事务之间的资源等待关系，并通过检测图中是否存在回路来检测死锁。等待图的结点是事务，边是事务之间的资源等待关系——如果 T_i 等待 T_j 持有的锁，则 T_i 到 T_j 存在一条有向边。很容易证明，如果等待图中存在回路，则事务执行出现了死锁。

图 11-37 显示了构建等待图检测死锁的例子。每当并发调度器接收到事务的加锁请求时，它可以通过查看锁表更新等待图。如果检测到等待图中出现了回路，则说明当前事务的调度出现了死锁。当出现死锁时，DBMS 可以随机选择回路中的某个事务将其撤销，从而破坏回路实现解锁。当然，选择撤销哪个事务也可以考虑其他的方法，如持有锁最多的事务、最早开始的事务等，但这些都不影响死锁处理的最终效果。

时间	T_1	T_2	T_3	T_4
1	$L_1(A)$; $r_1(A)$			
2		$L_2(C)$; $r_2(C)$		
3			$L_3(B)$; $r_3(B)$	
4				$L_4(D)$; $r_4(D)$
5		$L_2(A)$; Wait		
6			$L_3(C)$; Wait	
7				$L_4(A)$; Wait
8	$L_1(B)$; Wait			

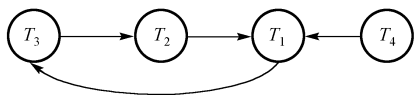


图 11-37　构建等待图检测死锁

11.6.2　死锁预防

死锁预防(Deadlock Prevention)是通过对加锁过程加以限制来保证并发操作过程中不会出现死锁。虽然从消除死锁的角度看死锁预防更有吸引力，但死锁预防通常有其他的代价，包括空间代价和时间代价。

目前常用的死锁预防策略包括优先顺序方法和时间戳方法。优先顺序方法是通过人为规定数据对象的加锁顺序来预防死锁，而时间戳方法则是通过为事务额外添加时间戳并基于时间戳先后顺序来控制加锁操作，从而达到死锁预防的目的。

1. 优先顺序方法

优先顺序方法是把要加锁的数据库元素按某种顺序排序，然后规定事务只能按照元素顺序请求锁。

图 11-38 给出了按优先顺序加锁的示例。这里规定事务必须按照 $A\rightarrow B\rightarrow C\rightarrow D\rightarrow$的字典顺序对数据对象进行加锁。因此，$T_1$ 必须先请求 A 上的锁，然后对 B 加锁，T_2 先请求 A 上的锁，然后请求 C 上的锁，以此类推。

时间	T_1	T_2	T_3	T_4
1	$L_1(A)$; $r_1(A)$			
2		$L_2(A)$; Wait		
3			$L_3(B)$; $r_3(B)$	
4				$L_4(A)$; Wait
5			$L_3(C)$; $w_3(C)$	
6	$L_1(B)$; $w_1(B)$		$U_3(B)$; $U_3(C)$	
7	$U_1(A)$; $U_1(B)$			
8		$L_2(A)$; $L_2(C)$		
9		$r_2(C)$; $w_2(A)$		
10	T_1: A, B	$U_2(A)$; $u_2(C)$		
11	T_2: A, C			
12	T_3: B, C			$L_4(A)$; $L_4(D)$
13	T_4: A, D			$r_4(D)$; $w_4(A)$
14				$U_4(A)$; $U_4(D)$

图 11-38　基于优先顺序的加锁过程示例

如图 11-39 所示，假设事务 T_1 持有了 A 上的锁，然后请求 B 上的锁，按照死锁的定义，必定存在一个事务 T_2 持有了 B 上的锁，然后正在请求 A 上的锁。这不满足按序加锁的规定，因此不可能出现这种情形。

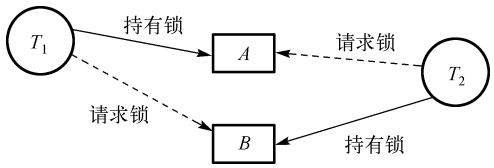


图 11-39　按序加锁可以预防死锁

2. 时间戳方法

时间戳方法是对每个事务在开始执行时都赋予一个时间戳。时间戳是数据库技术中常用的一种策略。时间戳实际上是一个整数，它只能增加，不能减小。因为这与时间只能往前无法倒流的性质类似，所以习惯称这样的数字为时间戳。

在数据库领域，时间戳有很多种用途。其中最常见的用途是标识数据的版本，例如，每当数据被修改一次，其时间戳就加一。如此一来，通过判断数据对象上当前的时间戳，就可以判断该数据自上次读取后有没有被其他事务修改过。

若基于时间戳方法进行死锁预防，当事务 T 请求被事务 U 持有的锁时，DBMS 根据 T 和 U 的时间戳来决定锁的授予方式。同时规定，如果事务 T 被取消后重新启动，T 的时间戳不变。

当一个事务 T 请求另一个事务 U 持有的锁时，有两种锁的授予机制，即等待-死亡(Wait-Die)和伤害-等待(Wound-Wait)。

1)等待-死亡

在等待-死亡机制下，当事务 T 请求被事务 U 持有的锁时：

(1)如果 T 的时间戳早于 U 的时间戳，即 timestamp(T)< timestamp(U)，则 T 进入等待状态，而 U 不受影响；

(2)如果 T 的时间戳晚于 U 的时间戳，即 timestamp(T)>timestamp(U)，则 T 死亡(即回滚)；

(3)死亡的事务在将来重新启动时其时间戳继续保持原先的时间戳。

图 11-40 给出了等待-死亡机制的一个示例。最左侧的列表示事务启动的时间。当在第 2 个时间点事务 T_2 请求对 A 加锁时，由于 T_1 已经持有了 A 的锁并且 T_1 时间戳早于 T_2，所以 T_2 死亡(回滚)。类似地，T_4 在第 4 个时间点请求对 A 加锁时也同样死亡。然后，在第 10 个时间点 T_2 重新启动并再度请求 A 的锁。此时，T_4 持有了 A 上的锁。但由于 T_4 的时间戳晚于 T_2(注意，事务重新启动后继续保持原先的时间戳)，所以 T_2 进入等待状态。

在等待-死亡机制下，事务的回滚总是发生在请求锁阶段，因此要回滚的事务操作比较少，但回滚的事务会比较多。

等待-死亡机制可以保证不会出现死锁。可以通过反证法进行证明：假设出现了死锁，形成了等待回路 $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$，因为在等待-死亡机制中，只有当 timestamp(T_i)< timestamp(T_j)时才会形成等待关系 $T_i \rightarrow T_j$，因此可以推出 $T_1 < T_2 < \cdots < T_k < T_1$，不成立。

时间	T_1	T_2	T_3	T_4
1	$L_1(A)$; $r_1(A)$			
2		$L_2(A)$; **Die**		
3			$L_3(B)$; $r_3(B)$	
4				$L_4(A)$; **Die**
5			$L_3(C)$; $w_3(C)$	
6			$U_3(B)$; $U_3(C)$	
7	$L_1(B)$; $w_1(B)$			
8	$U_1(A)$; $U_1(B)$			
9				$L_4(A)$; $L_4(D)$
10		$L_2(A)$; **Wait**		
11				$r_4(D)$; $w_4(A)$
12				$U_4(A)$; $U_4(D)$
13		$L_2(A)$; $L_2(C)$		
14		$r_2(C)$; $w_2(A)$		
15		$U_2(A)$; $U_2(C)$		

图 11-40　等待-死亡机制下的事务加锁示例

2) 伤害-等待

在伤害-等待机制下，当事务 T 请求被事务 U 持有的锁时：

(1) 如果 T 的时间戳早于 U 的时间戳，即 timestamp(T)< timestamp(U)，则 T “伤害”U——表示 U 被强制回滚并释放其持有的锁，然后 T 获得锁；

(2) 如果 T 的时间戳晚于 U 的时间戳，即 timestamp(T)>timestamp(U)，则 T 进入等待状态；

(3) 被伤害的事务在将来重新启动时其时间戳继续保持原先的时间戳。

图 11-41 给出了伤害-等待机制的一个示例。最左侧的列表示事务启动的时间。当在第 2 个时间点事务 T_2 请求对 A 加锁时，由于 T_1 已经持有了 A 的锁并且 T_1 时间戳早于 T_2，所以 T_2 进入等待状态。类似地，T_4 在第 4 个时间点请求对 A 加锁时也同样等待。在第 5 个时间点事务 T_1 请求对 B 加锁，此时 B 上的锁被 T_3 持有。由于 T_1 的时间戳早于 T_3，所以 T_3 被伤害，即 T_3 被回滚并释放 B 上的锁，从而 T_1 获得了 B 上的锁继续执行。

时间	T_1	T_2	T_3	T_4
1	$L_1(A)$; $r_1(A)$			
2		$L_2(A)$; **Wait**		
3			$L_3(B)$; $r_3(B)$	
4				$L_4(A)$; **Wait**
5	$L_1(B)$; $w_1(B)$		**Wounded**	
6	$U_1(A)$; $U_1(B)$			
7		$L_2(A)$; $L_2(C)$		
8		$r_2(C)$; $w_2(A)$		
9		$U_2(A)$; $U_1(C)$		
10				$L_4(A)$; $L_4(D)$
11				$r_4(D)$; $w_4(A)$
12				$U_4(A)$; $U_4(D)$
13			$L_3(B)$; $r_3(B)$	
14			$L_3(C)$; $w_3(C)$	
15			$U_3(b)$; $U_3(C)$	

图 11-41　伤害-等待机制下的事务加锁示例

在伤害-等待机制下，当发生某个事务被伤害的情形时，该事务已经获得了锁，因此有可能已经执行了较长时间，因此事务的回滚操作会较多。但是，需要回滚的事务数预期较少，因为一般情况下事务总是在开始时先请求锁。此外，请求锁时“等待”要比“伤害”更普遍，因为一般情况下一个新事务要请求的锁总是被一个较早的事务所持有。

伤害-等待机制可以保证不会出现死锁。可以通过反证法进行证明：假设出现了死锁，形成了等待回路 $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$,因为在伤害-等待机制中，只有当 timestamp(T_i)>timestamp(T_j)时才会形成等待关系 $T_i \rightarrow T_j$，因此可以推出 $T_1 > T_2 > \cdots > T_k > T_1$，不成立。

11.7　乐观并发控制

基于锁的并发控制实质上是一种“悲观”的做法。它立足于预防事务冲突，事务访问数据前都要请求锁。锁机制影响性能，容易带来死锁、活锁等副作用。因此，数据库领域通常把锁机制直接称为“悲观锁”。

如果每次访问数据都加锁能否也同样实现并发控制？答案是肯定的。这就是乐观并发控制。乐观并发控制假定不太可能(但不是不可能)在多个用户间发生资源冲突，允许不锁定任何资源而执行事务，只有试图更改数据时才检查资源以确定是否发生冲突。如果发生冲突，应用程序必须读取数据并再次尝试进行更改。

乐观并发控制的动机可归纳为八个字，即“读不加锁、写时协调”。如果大部分事务都是只读事务，则并发冲突的概率比较低，即使不加锁，也不会破坏数据库的一致性。但是，由于事务并发操作依然存在冲突的可能性，所以乐观并发控制虽然在事务读数据时不加锁，但是在写回数据时需要确定是否发生了冲突。如果发生冲突，则通过回滚冲突事务加以解决。

乐观并发控制中的核心问题是如何确定哪些事务发生了冲突。目前 DBMS 普遍采用的是有效性确认(Validation)技术。有效性确认技术要求所有事务都遵循有效性确认协议，分三个阶段对数据库进行操作。

(1)读阶段：数据被读入到事务的局部变量中。此时所有写操作都针对局部变量，并不对数据库进行更新。

(2)有效性确认阶段：事务进行有效性检查，判定是否可以将写操作所更新的局部变量值写回数据库而不违反可串性。

(3)写阶段：若事务通过了有效性检查，则进行实际的写数据库操作，否则回滚事务。

其中，有效性确认阶段是关键的步骤，目前 MySQL、Oracle 等采用两种方式进行实现。

(1)基于行版本的方式：MySQL 通过 Version 机制进行实现，而 Oracle 和 Microsoft SQL Server 则通过 Timestamp 机制进行实现。

(2)基于值比较的方式：只有 Microsoft SQL Server 支持。

下面以 Microsoft SQL Server 为例介绍乐观并发控制。虽然 MySQL、Oracle、Microsoft SQL Server 等普遍都支持乐观并发控制，但这些 DBMS 默认采用的仍是悲观并发控制，只有在某些特定的情况下才允许使用乐观并发控制。这里很重要的一个原因是乐观并发控制的性能不稳定：在读多写少的应用负载下，乐观并发控制可能性能表现较好，但在写多读少的场景下，由于大多数事务在提交时都可能出现冲突而导致回滚，所以乐观并发控制的

性能会变差。相比之下，基于 2PL 的悲观并发控制能够提供相对较稳定的性能输出。

Microsoft SQL Server 只允许在游标中使用并发控制。它在定义游标时允许指定以下的游标并发选项。

(1) READ_ONLY。

只读游标，不允许通过游标进行更新。这种情况下游标返回的记录集不加任何锁。

(2) OPTIMISTIC WITH VALUES。

基于值比较的乐观并发控制。当游标以此选项打开时，游标返回的记录集都不加锁。如果用户试图修改某一行，则此行的当前值会与最后一次提取此行时获取的值进行比较。如果任何值发生改变，则服务器就会知道其他人已更新了此行，并会返回一个错误。如果值是一样的，服务器就执行修改。

(3) OPTIMISTIC WITH ROW VERSIONING。

基于行版本的乐观并发控制。使用基于行版本的乐观并发控制时，其中的表必须具有 timestamp 列，用于表示行版本。数据库服务器通过 timestamp 来确定该行在读入游标后是否有所更改。

(4) SCROLL LOCKS。

这个选项实现悲观并发控制，有些文献中将其翻译为“滚动锁”。在悲观并发控制中，在把数据库的行读入游标结果集时，应用程序将试图锁定数据库行。滚动锁在提取时在每行上获取，并保持到下次提取或者游标关闭。

上述游标并发选项中，第二种和第三种采用的都是乐观并发控制，只不过第 3 种采用了基于值比较的乐观并发控制，而第三种则采用了基于行版本的乐观并发控制。

当 Microsoft SQL Server 选择以 OPTIMISTIC WITH ROW VERSIONING 选项定义游标时，要求相关的数据库表上必须要有特殊的 timestamp 列(数据库范围内唯一的 8 字节二进制数)。Microsoft SQL Server 使用 timestamp 来确定一行记录的版本。如果一个表包含 timestamp 列，则每次由 Insert、Update 或 Delete 语句修改一行时，此行的 timestamp 值就被置为当前的@@DBTS 值，然后@@DBTS 加 1。这里的@@DBTS 是系统定义的全局变量，它返回当前数据库最后所使用的时间戳。服务器可以比较某行的当前 timestamp 值和游标提取时的 timestamp 值，进行有效性确认。

当用户打开游标时，Microsoft SQL Server 保存行的当前 timestamp 值；当在游标中想更新一行时，Microsoft SQL Server 为更新数据自动添加一条 Where 子句：Where timestamp 列<= <old timestamp>。如果 Where 子句的条件不满足，则报错并回滚事务。

图 11-42 给出了在 Microsoft SQL Server 游标中使用基于行版本的乐观并发控制的例子。在这个例子中，先创建了一个包含 timestamp 列的表 test，然后在游标外更新了所有记录(这一步的目的是增加 timestamp 值，使得后面游标内的更新无法通过有效性检查)，最后当在游标内执行记录更新时，因为 test 表中被更新的记录的当前 timestamp 值跟游标打开时读取的 timestamp 值不相等，所以系统提示“乐观并发检查失败。已在此游标之外修改了该行。”图 11-43 显示了 test 表中的 timestamp 列在 Insert 操作之后的值。可以看到，每次 Insert 之后 timestamp 的值就自动加一，而 DBMS 就是利用 timestamp 的这一性质实现乐观并发控制时的有效性检查的。

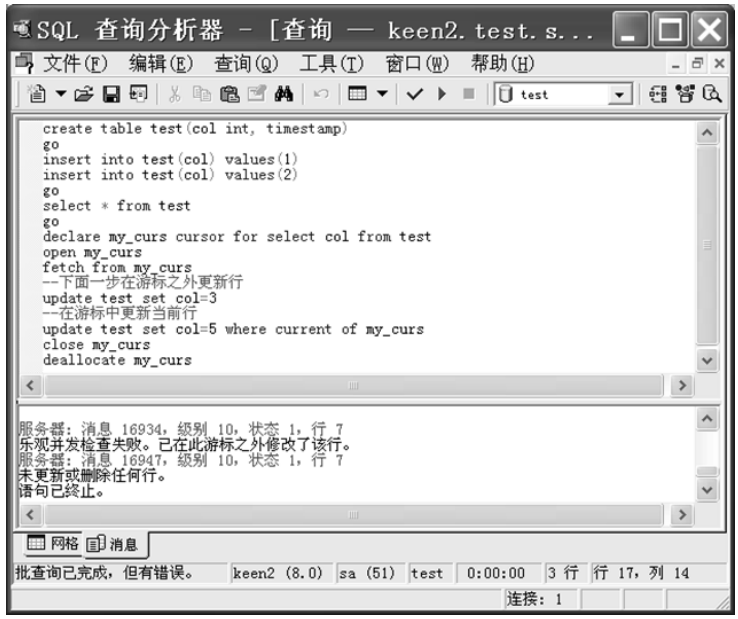


图 11-42　基于行版本的乐观并发控制示例

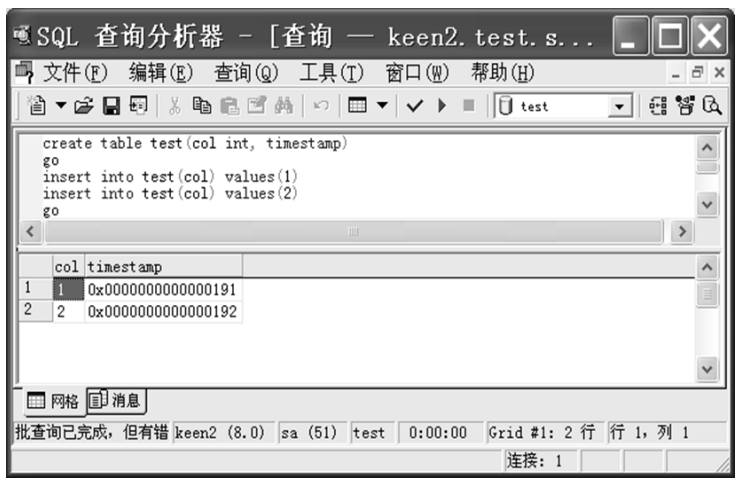


图 11-43　行版本(timestamp)的变化

如果表中没有 timestamp 列，Microsoft SQL Server 在游标并发中可以采用基于值比较的乐观并发控制(OPTIMISTIC WITH VALUES)。如果用户试图修改某一行，则此行的当前值会与最后一次提取此行时获取的值进行比较。如果任何值发生改变，则服务器就会知道其他人已更新了此行，并会返回一个错误并取消修改。如果值是一样的，服务器就执行修改。

11.8　本 章 小 结

本章主要介绍了数据库中的一个重要主题：并发控制。由于现在的 DBMS 基本都是支持多事务并发的多用户数据库系统，因此事务的并发控制是保证多事务并发执行时数据库

一致性的重要技术保障。本章首先介绍了并发操作可能带来的问题，然后重点讨论了可串化调度的概念以及冲突可串性的理论与判断方法。在此基础上，本章讨论了基于锁的并发控制机制以及乐观并发控制。此外，本章还讨论了与并发控制相关的事务隔离级别以及死锁等问题。

通过对本章的学习，读者应能够判断并发调度可能出现的问题，掌握冲突可串性的概念和优先图构造方法，重点掌握 2PL、X 锁、S 锁、U 锁、多粒度锁、意向锁等基本概念，了解事务的四种隔离级别、死锁检测与预防的方法以及乐观并发控制的相关技术。

习　　题

1．并发操作可能会产生哪几类问题？

2．什么是冲突可串？

3．举例说明，对并发事务的一个调度是可串化的，而这些并发事务不一定遵守两段锁协议。

4．证明：如果参与并发调度的所有事务都遵循 2PL，则这些事务产生的任何一个调度均满足冲突可串性。

5．基于时间戳的死锁预防策略有哪两种？相互之间有何区别？

第 12 章　数据库完整性

数据完整性是为了防止数据库中出现不符合语义的数据。为了维护数据的完整性，数据库管理系统必须提供一种机制来检查数据库的数据是否满足语义约束规定的条件。这些加在数据库之上的语义约束条件就是数据库中的数据完整性约束。因此，数据库完整性控制的主要任务是保证完整性约束的有效性。

内容提要：本章首先介绍数据库完整性控制的概念，然后讨论数据库完整性约束的定义和分类，最后介绍数据库完整性约束实施的主要途径。

12.1　数据库完整性控制的概念

数据库完整性控制是保证数据库中数据的正确性、有效性和相容性的技术。其中，正确性是指数据的合法性，例如，年龄由数字组成；有效性是指数据的取值应在有效范围内，例如，月份应为 1～12；相容性是指表示同一个事实的两个数据应该一致，例如，一个人的性别只有一个。

数据库完整性控制的核心是完整性约束，即数据的正确性、有效性和相容性都是通过一定的完整性约束来实现的。目前的 DBMS 提供了一系列的完整性约束类型和定义手段，DBA 可以在数据库设计阶段根据应用系统的需求定义相应的完整性约束。

数据库完整性对于数据库应用系统非常关键，其作用主要体现在以下几个方面。

(1) 数据库完整性约束能够防止合法用户使用数据库时向数据库中添加不合语义的数据。这种情况往往是因为用户的误操作而引起的，例如，在输入课程成绩“96”时多输入一位数字“6”就会使数据库中出现“966”这样的错误数据。

(2) DBMS 提供的完整性控制机制可以有效地实现应用程序的业务规则，而且易于定义，容易理解，可以降低应用程序的复杂性，提高应用程序的运行效率。同时，基于 DBMS 的完整性控制机制是集中管理的，因此比应用程序更容易实现数据库的完整性。

(3) 合理的数据库完整性设计，能够同时兼顾数据库的完整性和系统的效能。

(4) 在应用软件的功能测试中，数据库完整性有助于尽早地发现应用软件的错误。例如，如果应用程序的输入接口容易导致输入错误，在测试时很容易通过数据库完整性约束发现这类问题。

DBMS 的数据库完整性控制机制应具有以下三个功能。

(1) 定义功能：提供定义完整性约束条件的机制。

(2) 检查功能：检查用户发出的操作请求是否违背了约束条件。一般有两种检查方式：一种是立即执行约束(即一条语句执行完成后立即检查约束)；另一种是延迟执行约束(即在整个事务执行完毕后再检查约束)。

(3) 违约响应功能：如果操作请求使数据不满足完整性约束条件，则采取一定的动作来保证数据的完整性。

12.2　数据库完整性约束的定义

DBA 向 DBMS 提出一组完整性约束来检查数据库中的数据是否满足语义约束，DBMS 完整性控制系统会维护这些完整性约束的有效性。

数据库完整性约束主要由三部分构成。

(1)触发条件：系统什么时候使用规则来检查数据。

(2)约束条件：系统检查用户发出的错误操作不能满足完整性约束条件。

(3)违约响应：违约时要做的事情。

完整性约束是由 DBMS 提供的语句来描述的，并且存储在数据字典中，但违约由 DBMS 来处理。

下面给出完整性约束的形式化定义。

定义 12-1(完整性约束)　数据库完整性约束是一个五元组(*D*,*O*,*A*,*C*,*P*)，其中各部分具体如下。

D(Data)：约束作用的数据对象。

O(Operation)：触发完整性检查的数据库操作，即当用户发出什么操作请求时需要检查该完整性规则，是立即检查还是延迟检查。

A(Assertion)：数据对象要遵守的断言或语义规则。

C(Condition)：选择 *A* 作用的数据对象值的谓词。

P(Procedure)：违反完整性规则时触发的过程。

表 12-1 给出了完整性约束的两个例子。

表 12-1　完整性约束示例

条件	学号不能为空	教授工资不得低于 1000 元
D	约束的对象为 SNO 属性	约束的对象为 SAL 属性
O	插入和修改 STUDENT 元组时	插入和修改职工元组时
A	SNO 不能为空	SAL 不能小于 1000
C	无(作用于所有记录的 SNO 属性)	职称=‘教授’(作用于职称=‘教授’的记录)
P	拒绝执行	拒绝执行

12.3　数据库完整性约束的分类

完整性约束条件作用的对象可以是表、元组、列三种。完整性约束条件涉及的这三类对象，其状态可以是静态的，也可以是动态的。

完整性约束的分类有几种不同的观点，下面分别进行讨论。

12.3.1　按约束对象的粒度分类

按约束对象的粒度，完整性约束可分为关系级约束、列级约束和元组级约束。

1) 关系级约束

关系级约束指若干元组间、关系中以及关系之间联系的约束。

2) 列级约束

列级约束指针对列的类型、取值范围、精度、排序等而制定的约束。

3) 元组级约束

元组级约束指元组中的字段组和字段间联系的约束。

例如，在下面的 Create Table 语句中，age 列上的 Check 约束是列级约束，它指定了 age 列的取值范围。sno 列上的 Primary Key 约束是关系级约束，它要求整个关系的所有元组的 sno 都不可重复。gender 列上的 Check 约束是元组级约束，它定义了元组中 sname 和 gender 两个属性值之间的约束条件，即当性别(gender)是男(M)时，要求学生姓名(sname)不能以 Ms.打头。

```
Create Table Student
(   sno     Char(10),
    sname   Char(30),
    age     Int Check(age > 16),
    gender  Char(1),
    constraint PK_S Primary Key (sno),
    constraint CK_S Check (gender='F' or sname NOT LIKE 'Ms.%')
)
```

12.3.2　按约束对象的状态分类

按约束对象的状态，数据库完整性约束可分为静态约束和动态约束两类。

1) 静态约束

静态约束指数据库每一个确定状态时的数据对象所应满足的约束条件，它是反映数据库状态合理性的约束，这是数据库中最重要的一类完整性约束。

例如，学生表中“学生的年龄不能小于 16 岁”这一约束要求任何时刻的学生表状态都必须满足所有学生年龄都小于 16 岁这一约束条件。

2) 动态约束

动态约束指数据库从一种状态转变为另一种状态时新、旧值之间所应满足的约束条件，它是反映数据库状态变迁的约束。

例如，在员工表中要求“新工资=原工资+工龄×1.5”就是一个动态约束，它要求在修改员工工资时必须遵循与原工资之间的特定计算关系。

12.3.3　按约束的作用类型分类

按约束的作用类型，数据库完整性约束可分为域完整性约束、实体完整性约束和参照完整性约束三类。图 12-1 给出了这三类完整性约束在作用域上的差别。

1) 域完整性约束

域完整性约束是最简单、最基本的约束，它作用在列上，保证列取值的合理性。在当今的关系 DBMS 中，一般都有域完整性约束检查功能。在 SQL 数据库中，域完整性主要在列定义时通过 Check 约束和默认值(DEFAULT)为列指定一个有效的数据集，并确定该列是否允许为空。

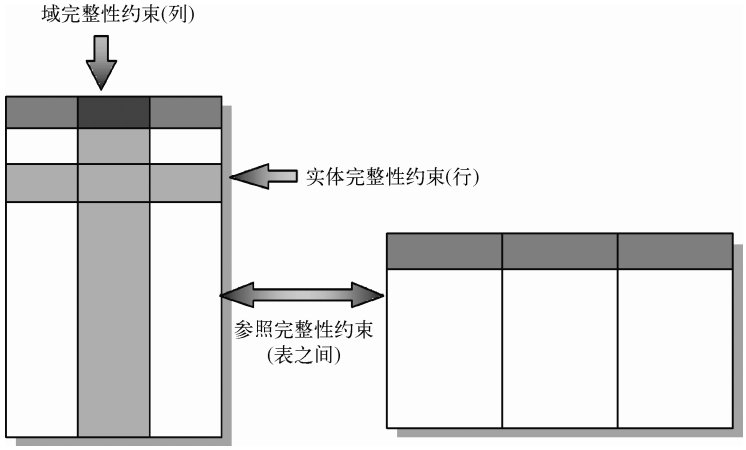


图 12-1　域完整性约束、实体完整性约束和参照完整性约束在作用域上的差别

需要注意的是，域与类型这两个概念之间是有区别的。域代表了列的取值范围，而类型是一个值集。一般来说，域的概念比类型要小一些，因为有许多类型都是无限集，如整型、浮点类型等。

在 SQL 数据库中，可以用 Create Domain 语句建立一个域以及该域应该满足的完整性约束条件，并且可以在定义列时使用自定义的域。例如，下面的 SQL 语句建立了一个表示性别的域 GenderDomain：

```
Create Domain GenderDomain Char(2)
Constraint CK_gender Check (Value IN ('男', '女') )
```

建立的域可以在定义表时使用，以实现自定义域中的完整性约束。例如，下面的 Create Table 语句中使用了上面定义的 GenderDomain 域：

```
Create Table Student
(   sno Char(10),
    sname Char(20),
    gender GenderDomain,
    age Int
)
```

使用自定义域的好处是可以在多个表定义中重复使用自定义域，从而避免在 Create Table 时频繁地定义约束。

2) 实体完整性约束

实体完整性约束要求关系的主码不能重复，也不能为空。一个关系对应现实世界中一个实体集。现实世界中的实体是可以相互区分、识别的，即它们应具有某种唯一性标识。实体完整性表明关系模式中不存在不可标识的实体。如果出现主码重复或者为空的情况，说明实体是不可标识的，这与现实世界的实际情况相矛盾，这样的实体就不是一个完整实体。

3) 参照完整性约束

参照完整性约束是定义建立关系之间联系的主码与外码时引用的约束条件。关系数据库中通常都包含多个存在相互联系的关系，关系与关系之间的联系是通过外码来实现的。如果一个关系 R(称为被参照关系或目标关系)的主码同时又是另一关系 K(称为参照关系)

的外码，则参照完整性约束要求参照关系 *K* 中外码的取值，要么与被参照关系 *R* 中某元组主码的值相同，要么取空值。如果参照关系 *K* 的外码也是主码的一部分，根据实体完整性约束的要求，主码不得取空值，因此，参照关系 *K* 外码的取值实际上只能取相应被参照关系 *R* 中已经存在的主码值。

12.4　数据库完整性约束实施途径

在 SQL 数据库中，数据库完整性约束的实施途径包括约束(Constraint)、规则(Rule)和触发器(Trigger)和断言四种类型。

1. 约束

SQL 提供了 Primary Key、Unique 等约束。这些约束是在实际应用中实施数据库完整性约束最常用的途径。表 12-2 给出了不同类型的完整性约束在 SQL 数据库中的实现方法。具体的约束类型在介绍 SQL 时已经有过详细讨论，此处不再赘述。

表 12-2　域完整性约束、实体完整性约束和参照完整性约束在 SQL 数据库中的实现

完整性类型	约束类型	完整性功能描述
域完整性	Default	插入数据时，如果没有明确提供列值，则用缺省值作为该列的值
	Check	指定某个列可以接受的值范围，或指定数据应满足的条件
实体完整性	Primary Key	指定主码，确保主码值不重复，并不允许主码为空值
	Unique	指出数据应具有唯一值，防止出现冗余
参照完整性	Foreign Key	定义外码，确保外码值出现在被参照表的候选码中

2. 规则

规则是一组用过程化 SQL(如 Microsoft SQL Server T-SQL)书写的条件语句，它可以和列或用户自定义类型捆绑在一起，用来实施完整性约束。规则一旦定义后，可以被多个列共享，表示实施了同一种完整性约束条件。

在 SQL 数据库中，SQL 的 Where 子句中合法的语句一般都可以用在规则定义中，如算术运算符、关系运算符、IN、LIKE、BETWEEN 等。

下面以 Microsoft SQL Server 为例，介绍利用 T-SQL 定义规则来实施完整性约束的方法。整个实施方法包括两个步骤：定义规则和绑定规则。

1)定义规则

规则的定义通过 Create Rule 语句来完成，其语法如下：

```
Create Rule rule_name
As condition_expression
```

其中，rule_name 是所定义规则的名称。规则名称必须符合标识符规则。condition_expression 是规则所包含的条件，它可以是 Where 子句中任何有效的表达式。在 condition_expression 中可以使用本地变量来表示规则所要约束的列。

2）绑定规则

规则定义完成后，需要将规则与特定的列和用户自定义类型进行绑定，从而将规则应用到列上实施完整性约束。在 Microsoft SQL Server 中，规则绑定通过系统存储过程 sp_bindrule 来实现，其语法如下：

```
sp_bindrule [ @rulename = ] 'rule' , [ @objname = ] 'object_name '
```

其中，@rulename 是规则名，@objname 是要绑定的列名或用户自定义类型的名称。如果规则定义时在 condition_expression 中使用了本地变量，绑定规则后本地变量就会自动映射成所绑定的列名。

【例 12-1】 设有学生表 Student（sno, sname, email），在数据库中创建一个 E-mail 的规则对象，其值为包含@的字符串，并将它应用到学生表的 email 列上。

（1）创建规则。

```
Create Rule rl_email
As
@val LIKE '%@%'
```

（2）绑定规则到学生表的 email 列上。

```
sp_bindrule 'rl_email', 'student.email'
```

3. 触发器

触发器是实施数据库完整性约束的另一种常用的途径。正如在第 5 章所讨论的，触发器是一类特殊的存储过程，它关联在某个特定的表上，并且由表上的更新操作自动触发运行。正因为触发器是自动触发运行的，所以触发器没有参数，也不能像存储过程那样被用户显式调用。

触发器特别适合用来实现元组级的动态约束，即当某个列发生修改时，要求其他列与所修改的列值之间满足一定的约束条件。

【例 12-2】 假设在一个银行系统中有账户表 Account（AccountID, Account_Name, Balance, BranchID），其中存储了账户、用户名、余额和所在的支行号，支行表为 Branch（BranchID, Branch_Name, SumSaving），其中存储了支行号、支行名称和当前的存款总额。要求在 Account 表中增加一个新的账户或者修改某个账户的余额时，同步更新 Branch 表中的 SumSaving 列。下面是 Microsoft SQL Server 中基于触发器的实现结果：

```
Create Trigger Update_Sum On Account
For Insert, Update
As
Declare @new Int
Declare @old Int
Declare @ID Int
Select @new = (Select Balance From inserted)
If Update(Balance)
   Select @old= (Select Balance From deleted)
Else
```

```
        @old = 0
    Select @ID=(Select BranchID From inserted)
    -- 要根据它确定 Branch 表中需更新的行
    Update Branch Set SumSaving=SumSaving + @new – @old Where BranchID=@ID
```

4. 断言

断言(Assertion)也是一种实施数据库完整性约束的途径(但 Oracle 不支持断言)。一个断言对应了一个谓词条件，表明数据应满足的约束。用户可以通过 Create Assertion 语句创建一个断言实施约束，之后对断言涉及的数据进行操作时会触发断言。如果数据操作不满足断言规定的条件，操作将被拒绝。

【例 12-3】 现有一个选课表 SC(S#, C#, Score)。希望限制每一门课程选修人数不超过 60。下面给出了断言的定义：

```
    Create Assertion asser1
    Check(60>=All (Select Count(*) From SC Group By C#));
```

12.5　本 章 小 结

本章主要介绍了数据库完整性控制的基本概念以及数据库完整性约束的分类，并重点介绍了目前数据库系统中实施完整性约束的几种主要途径。完整性控制的内容与 SQL 密切相关，因此本章的相关内容建议与第 4 章结合起来理解。

通过对本章的学习，读者应了解数据库完整性的概念和类型，掌握实体完整性、参照完整性等重要概念的区分，并了解几种常见的数据库完整性约束实施途径。

习　　题

1．DBMS 的完整性控制机制应具有哪些功能？

2．数据库中的静态约束与动态约束分别是什么含义？

3．举例说明什么是域完整性和参照完整性。

4．数据库完整性约束的实施途径有哪些？

第 13 章　数据库安全性

数据库的安全性是指保护数据库，以防止不合法的使用造成的数据泄密、更改或破坏。数据库管理系统安全保护就是通过种种防范措施以防止用户越权使用数据库。安全保护措施是否有效是衡量数据库系统的主要性能指标之一。

内容提要：本章首先介绍数据库安全性控制的基本概念，然后讨论用户标识与鉴别方法，接着重点介绍自主访问控制机制和强制访问控制机制，最后介绍视图与安全性控制。

13.1　数据库安全性控制概述

数据库安全性控制的目标是保证数据的保密性、完整性、可用性和一致性。保密性是指要防止非授权的数据读取，避免信息泄露。完整性是指防止非授权的数据篡改。可用性是指正常的数据存取不能受到影响，要能够正常得到响应。一致性是指要保证数据库一致性约束的有效性。

数据库的安全性会因为各种原因而受到威胁，几种典型的情形如下。

(1) 用户编写一段合法的程序绕过 DBMS 及其授权机制，通过操作系统直接存取、修改或备份数据库中的数据。

(2) 直接或编写应用程序执行非授权操作。

(3) 通过多次合法查询从数据库中推导出一些保密数据。

例如，某数据库应用系统禁止查询单个人的工资，但允许查任意一组人的平均工资。用户甲想了解乙的工资，于是他执行下面的推理操作：

(1) 查询包括乙在内的一组人的平均工资。

(2) 查用自己替换乙后这组人的平均工资。

从而可以推导出乙的工资，完成信息窃取。

数据库安全性控制的常用技术包括用户标识与鉴别、访问控制、审计、操作系统安全保护、密码存储等。这些技术从高到低形成了对数据库系统的保护。图 13-1 给出了这些安全性控制的层次。

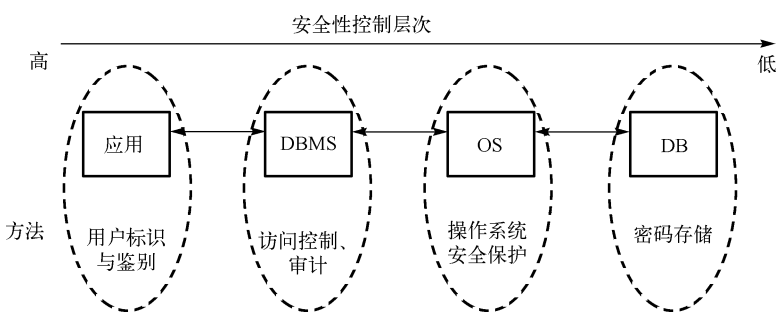


图 13-1　安全性控制的层次

13.2　用户标识与鉴别

用户标识与鉴别(Identification and Authentication)是 DBMS 提供的最外层保护措施。用户每次登录数据库时都要输入用户标识，DBMS 进行核对后，合法的用户获得进入系统最外层的权限。

用户标识与鉴别的方法很多，常用的方法有身份认证、口令认证、随机数认证等。

1. 身份认证

用户的身份是系统管理员为用户定义的用户名(也称为用户标识、用户账户、用户 ID)，并记录在计算机系统或 DBMS 中。用户名是用户在计算机系统中或 DBMS 中的唯一标识。因此，一般不允许用户自行修改用户名。

身份认证是指系统对输入的用户名与合法用户名进行对照，鉴别此用户是否为合法用户。若是，则可以进入下一步的核实；否则，不能使用系统。

2. 口令认证

用户的口令是合法用户自己定义的密码。为保密起见，口令由合法用户自己定义并可以随时变更。因此，口令可以认为是用户私有的钥匙。口令记录在数据库中。

口令认证是为了进一步对用户进行核实。通常系统要求用户输入口令，只有口令正确才能进入系统。为防止口令被人窃取，用户在终端上输入口令时，口令的内容是不显示的，在屏幕上用特定字符(用*较为常见)替代。

通过用户名和口令来鉴定用户的方法简单易行，但用户名与口令容易被人窃取，因此还需要用更复杂的方法。

3. 随机数认证

随机数认证实际上是非固定口令的认证，即用户的口令每次都是不同的。鉴别时系统提供一个随机数，用户根据预先约定的计算过程或计算函数进行计算，并将计算结果输送到计算机，系统根据用户计算结果判定用户是否合法。例如，算法为“口令=随机数平方的后三位”，出现的随机数是 36，则口令是 296。

13.3　访问控制机制

数据库安全性所关心的主要是 DBMS 的访问控制机制(Access Control)。数据库安全最重要的一点就是确保只授权给有资格的用户访问数据库的权限，同时令所有未被授权的人员无法接近数据，这主要通过数据库系统的访问控制机制实现。

访问控制机制主要包括两部分。

(1)授权(Authorization)：定义用户权限，并将用户权限登记到数据字典中。用户权限是指不同的用户对于不同的数据对象允许执行的操作权限。要进行用户权限定义，DBMS 必须提供有关定义用户权限的语言，即数据控制语言(DCL)，例如，SQL 中 Grant、Revoke

等语句提供了定义用户权限和撤回用户权限的功能。具有授权资格的用户使用 DCL 描述授权决定，并把授权决定告知计算机。授权决定描述中包括将哪些数据对象的哪些操作权限授予哪些用户，计算机分析授权决定，并将编译后的授权决定存放在数据字典中，从而完成了对用户权限的定义和登记。这些定义经过编译后存放在数据字典中，称为安全规则或授权规则。

(2)验证(Authentification)：当用户发出存取数据库的操作请求后(请求一般应包括操作类型、操作对象和操作用户等信息)，DBMS 查找数据字典，根据安全法则进行合法权限检查，若用户的操作请求超出了定义的权限，系统将拒绝执行此操作。

授权机制与验证机制一起组成了 DBMS 的安全子系统。

目前提出的访问控制机制主要有自主访问控制(Discretionary Access Control，DAC)和强制访问控制(Mandatory Access Control，MAC)两种类型。其中，自主访问控制机制对应着计算机系统安全等级中的 C2 级，是目前操作系统和数据库管理系统中最常见的访问控制机制；强制访问控制机制对应着计算机系统安全等级的 B1 级，实现了强制访问控制的系统可以称为“可信的”或者“安全的”系统。因此，“安全 DBMS”的概念一般指实现了强制访问控制的 DBMS。目前国内外商业领域普遍使用的 DBMS 产品都是 C2 级的，还不能称为安全 DBMS。Oracle、Sybase 等都推出了安全级产品，但都是美国禁止出口的产品，一般在军事应用中使用。

13.3.1　自主访问控制机制

当前大型的 DBMS 一般都支持 C2 级中的自主访问控制(DAC)，目前的 SQL 标准也对自主访问控制提供支持，这主要通过 SQL 的 Grant、Revoke 等语句来实现。在自主访问控制机制中，用户对于不同的数据对象有不同的存取权限，不同的用户对同一对象也有不同的权限，而且用户还可将其拥有的存取权限转授给其他用户。因此，自主访问控制机制非常灵活。

存取权限是由两个要素组成的：数据对象和操作类型。定义一个用户的存取权限就是定义这个用户可以在哪些数据对象上进行哪些类型的操作。在数据库系统中，授权操作就是指定义存取权限。

在非关系系统中，用户只能对数据进行操作，访问控制的数据对象也仅限于数据本身。而关系数据库系统中，DBA 可以把建立、修改基本表的权限授予用户，用户获得此权限后可以建立和修改基本表、索引、视图。因此，关系数据库系统中访问控制的数据对象不仅有数据本身，如表、属性等，还有模式、外模式、内模式等数据字典中的内容。表 13-1 给出了关系数据库系统中的存取权限定义。

表 13-1　关系数据库系统中的存取权限定义

类别	数据对象	操作类型
模式	概念模式	建立、修改、删除、检索
	内模式	建立、删除、检索
	外模式	建立、删除、检索
数据	基本表	插入、删除、修改、查询
	列	插入、删除、修改、查询

在自主访问控制的基础上，现代的 DBMS 还可以通过定义存取谓词来实现一些特殊的访问控制。例如，规定用户只能在特定的时间里存取某个基本表。存取谓词定义用户能够存取的数据筛选条件。表 13-2 给出了存取谓词的几个例子。

表 13-2　存取谓词示例

用户名	数据对象	操作类型	存取谓词
John	Employee	UPDATE	EmpName = ‘John’
Mary	Department	SELECT	DeptName = ‘Sales’
Rose	Supplier	SELECT	无

用户还可以通过触发器等来自定义一些存取谓词以实现特殊目的的授权管理。在定义存取谓词时，可以引用系统变量，如终端设备号、系统时钟等，实现与时间地点有关的存取权限，这样用户只能在某段时间内，在某台终端上存取有关数据。

例如，下面的 Microsoft SQL Server 触发器规定了只能在周一至周五的工作时间 9:00～17:00 更新员工表(Employee)。

```
Create Trigger secure_ck
On Employee
For Insert, Delete, Update
As
If Datename(weekday, getdate())='星期六'
   or Datename(weekday, getdate())='星期日'
   or (convert(Int, Datename(hour, getdate())) Not Between 9 and 17)
Begin
   Raiserror ('只许在工作时间更新员工表! ', 16, 1)
   Rollback Transaction
End
```

上面的触发器代码中，Raiserror 语句人工生成一个错误对象，这个错误对象可以最终返回给用户应用程序。这样，当用户应用程序在非工作时间访问 Employee 表时就会收到这个错误对象，并且可以得到触发器中自定义的错误信息“只许在工作时间更新员工表!”

自主访问控制能够通过授权机制有效地控制其他用户对敏感数据的存取。但是由于用户对数据的存取权限是“自主”的，用户可以自由地决定将数据的存取权限授予何人，以及是否也将“授权”的权限授予别人，而系统对此无法控制。在这种授权机制下，仍可能存在数据的“无意泄露”。例如，甲将自己权限范围内的某些数据存取权限授权给乙，甲的意图是只允许乙本人操纵这些数据，但甲的这种安全性要求并不能得到保证，因为乙一旦获得了对数据的存取权限，就可以将数据备份，获得自身权限内的副本，并在不征得甲同意的前提下传播副本。

造成这一问题的根本原因就在于这种机制仅仅通过对数据的存取权限来进行安全控制，而数据本身并无安全标记。要解决这一问题，就需要对系统控制下的所有主客体实施强制访问控制机制。

13.3.2　强制访问控制机制

在强制访问控制机制中，DBMS 所管理的全部实体分为主体和客体两大类。

主体是系统中的活动实体，既包括 DBMS 所管理的实际用户，也包括代表用户的各进程。客体是系统中的被动实体，是受主体操纵的，包括文件、基本表、索引、视图等。对于主体和客体，DBMS 为它们每个实例(值)指派一个安全标记(Label)。

安全标记分成若干级别，如绝密(Top Secret)、机密(Secret)、可信(Confidential)、公开(Public)等。主体的安全标记称为存取级(Clearance Level)，客体的安全标记称为密级(Classification Level)。强制访问控制机制就是通过对比主体和客体的安全标记，最终确定主体是否能够存取客体。

当某一用户(或某一主体)以安全标记注册进入系统时，系统要求他对任何客体的存取必须遵循如下规则。

规则 1：仅当主体的许可证级别大于或等于客体的密级时，该主体才能读取相应的客体。

规则 2：仅当主体的许可证级别等于客体的密级时，该主体才能写相应的客体。

规则 1 的意义是明显的，而规则 2 需要解释。在某些系统中，第二条规则与这里的规则 2 有些差别。这些系统规定：仅当主体的许可证级别小于或等于客体的密级时，该主体才能写相应的客体，即用户可以为写入的数据对象赋予高于自己的安全级别的密级。这样一旦数据被写入，该用户自己也不能再读该数据对象了。这两种规则的共同点在于它们均禁止了拥有高安全级别的主体更新低密级的数据对象，从而防止了敏感数据的泄露。

强制访问控制(MAC)是对数据本身进行密级标记，无论数据如何复制，安全标记与数据是一个不可分的整体，只有符合安全标记要求的用户才可以操纵数据，从而提供了更高级别的安全性。

强制访问控制的实现要求首先实现自主访问控制，这是因为强制访问控制是高级别的安全保护方法。在安全性控制中，高安全级别要包含较低级别提供的所有保护。因此，在实现了强制访问控制的 DBMS 中，强制访问控制和自主访问控制共同构成 DBMS 的安全机制。

13.4　视图与安全性控制

在 SQL 数据库中，视图对应着外模式。视图最初的目的是实现数据的逻辑数据独立性，以提高数据库应用系统的可维护性。但是，也可以利用视图机制来实现一定的安全保护功能。

视图机制实现安全保护功能的基本思路是为不同的用户定义不同的视图，把数据对象限制在一定的范围内，即通过视图机制把要保密的数据对无权存取的用户隐藏起来，从而自动地对数据提供一定程度的安全保护。

实际中，视图机制需要与授权机制配合使用，具体的实现过程如下。

(1)为特定的用户定义视图，通过视图定义屏蔽掉需要对该用户保密的数据。例如，对于 Web 用户，要求他们不能查看员工的工资信息，则可以在定义 Web 用户的视图时去掉员工表上的工资属性。

(2)在视图上面进一步定义存取权限，使得用户可以在指定权限范围内对视图进行存取。

(3) 去掉用户直接存取视图所基于的基本表的相应权限，从而使得用户不可能直接去存取基本表信息。这样做的另一个好处是，即使该用户的用户名和口令信息泄露，也不会导致基本表上的保密信息的泄露，因为该用户没有存取基本表的权限。

例如，假设要求用户 John 只能查询其所在部门 Sales 的员工信息(Employee 表的 EmpID、Ename、Age、Dept 等信息)，但是不能查询 Salary 信息，则可以定义下面的视图：

```
Create View Emp_view
As
Select EmpID, Ename, Age, Dept
From Employee
Where Dept = 'Sales'
```

然后，将该视图上的 Select 权限通过 Grant 语句授权给用户 John：

```
Grant Select on Emp_view to John
```

同时，去掉用户 John 对基本表 Employee 的直接存取权限：

```
Deny Select On Employee To John
```

通过上面的这种方式，间接地实现了自定义的安全性控制功能。

13.5　本 章 小 结

本章介绍了数据库安全性的相关内容。数据库安全性控制是 DBMS 中的一个重要功能，它为保障数据库数据的保密性、完整性等提供了保证。本章主要介绍了数据库安全性控制的基本概念和用户标识与鉴别方法，并重点介绍了目前数据库系统中常用的两种访问控制机制，包括自主访问控制机制和强制访问控制机制。

通过对本章的学习，读者应了解数据库安全性控制的基本概念，了解自主访问控制和强制访问控制之间的区别，以及利用视图增强数据库安全性的基本方法。

习　　题

1．试述实现数据库安全性控制的常用方法和技术。

2．什么是数据库中的自主访问控制机制和强制访问控制机制？

3．为什么视图可以在一定程度上增强数据库的安全性？请举例说明。

第 14 章　数据库技术新发展

数据库技术经过了 40 多年的发展，在理论与应用方面都取得了许多成果。关系数据库技术一直是数据库领域中占据最主要地位的技术，但是随着技术的发展、数据类型的复杂化、数据量的日益增长以及用户需求的不断演化，数据库技术也出现了许多新的发展。本章将重点围绕数据库领域中的新技术展开讨论。这些新技术中大多是目前已经得到了实用性验证并产品化的技术，对于其他一些目前尚处于理论研究阶段的技术，在本书中不进行专门讨论。

内容提要：本章首先介绍分布式数据库技术，然后讨论面向对象数据库技术，接着对对象关系数据库技术进行简要概述，最后介绍 NoSQL 数据库技术。

14.1　分布式数据库技术

分布式数据库技术是随着网络技术和关系数据库技术的发展而产生的。在现实世界中，许多应用系统的数据天然地具有物理分布的特点，如连锁超市、银行系统、邮电系统、大型企业的应用系统等。网络技术的发展使得这些物理分布的站点可以相互通信并进行信息共享和交互。与此同时，随着数据库技术日益成为主流的企业数据管理手段，如何使数据库技术适应具有物理分布特点的应用系统成为一个新的问题。分布式数据库技术正是在这样的背景下产生的。它为分布式应用提供了有效的数据库技术支持。

14.1.1　分布式数据库技术的产生与发展

20 世纪 70 年代，计算机网络技术的发展使不同地点的计算机互联互通成为可能。同时，在这一段时间里，数据库技术也逐步走向了成熟，关系数据库技术得到了快速发展和应用。因此，结合网络技术和关系数据库技术的分布式数据库技术开始出现。它将不同地点的数据库服务器通过网络进行互连，同时向上层应用提供一个统一的数据视图，从而既实现了对分布式应用的支持，也保证了数据库系统的数据统一管理和共享的特点。

分布式管理也符合许多企业应用需求，如银行、保险、跨国企业、连锁业、交通运输业等。在这类应用中，数据通常分散存储在不同地域的站点上，但在管理上要求集中管理系统范围内的所有数据。

历史上第一个分布式数据库系统是美国计算机公司 CCA 于 1976～1978 年设计完成的 SDD-1 系统。此后，在 20 世纪 80 年代，由于计算机与网络技术的发展，分布式数据库技术在世界范围内得到了研究和应用。在这一时期出现了一些著名的分布式数据库系统，如德国斯图加特大学的 POREL 系统、IBM 的 R*系统（分布式 System R）、美国加利福尼亚大学伯克利分校的 INGRES/STAR 系统、法国 INRIA 的 SIRIUS-DELTA 以及 IMAG 的 MICROBE 系统等。

1986 年，IBM 的 C.J. Date 提出了理想的分布式数据库系统应满足的 12 条原则。这些原则对于后来的分布式数据库研究产生了巨大的影响。这 12 条原则如下。

(1) 局部结点自治性：网络中的每个结点是独立的数据库系统。它有自己的数据库，运行它的局部 DBMS，执行局部应用，具有高度的自治性。

(2) 不依赖中心结点：每个结点具有全局字典管理、查询处理、并发控制和恢复控制等功能。

(3) 能连续操作：使中断分布式数据库服务的情况减至最少，当一个新站点合并到现有的分布式系统或在分布式系统中撤离一站点时，不会导致任何不必要的服务中断；在分布式系统中可动态地建立和消除片段，而不中止任何组成部分的站点或数据库的操作；应尽可能在不使整个系统停机的情况下对组成分布式系统的站点的 DBMS 进行升级。

(4) 具有位置独立性(或称位置透明性)：用户不必知道数据的物理存储地，可工作得像数据全部存储在局部站点一样。一般位置独立性需要有分布式数据命名模式和字典子系统的支持。

(5) 分片独立性(或称为分片透明性)：分布式系统可将给定的关系分成若干块或片，可提高系统的处理性能。利用分片将数据存储在最频繁使用它的位置上，使大部分操作是局部操作，减少网络的信息流量。如果系统支持分片独立性，用户工作起来就像数据全然不是分片的一样。

(6) 数据复制独立性：将给定的关系(或片段)复制存储到不同站点上。支持数据复制的系统应当支持复制独立性，要求数据复制对用户透明，使用户不必关心复制细节。

(7) 支持分布式查询处理：在分布式数据库系统中有三类查询，即局部查询、远程查询和全局查询。局部查询和远程查询仅涉及单个结点的数据(本地的或远程的)，查询优化采用的技术是集中式数据库的查询优化技术。全局查询涉及多个结点上的数据，其查询处理和优化要复杂得多。

(8) 支持分布事务管理：事务管理有两个主要方面，即恢复控制和并发控制。在分布式系统中，单个事务会涉及多个站点上的代码执行，也会涉及多个站点上的更新，可以说每个事务是由多个代理组成的，每个代理代表在给定站点上的给定事务执行的过程。在分布式系统中须保证事务的代理集全部一致交付，或者全部一致回滚。

(9) 具有硬件独立性：可以在不同硬件系统上运行同样的 DBMS。

(10) 具有操作系统独立性：希望在不同的操作系统上运行 DBMS。

(11) 具有网络独立性：系统能够支持多个不同的站点，每个站点有不同的硬件和不同的操作系统，能支持各种不同的通信网络。

(12) 具有 DBMS 独立性：理想的分布式系统应该提供 DBMS 独立性，能够支持异构型分布式系统。

20 世纪 90 年代开始，DB2、Oracle 等开始提供对分布式数据库技术的部分支持，例如，Oracle 推出了 Distributed Database Option，DB2 也推出了 Distributed Data Facility。但目前完全支持 12 条规则的商用系统还未实现。目前，各个主流的数据库软件都支持分布式数据库技术，包括 Oracle、DB2、Microsoft SQL Server 等。

国内从 20 世纪 80 年代开始研究分布式数据库技术，主要的研究单位有中国科学院、华东师范大学、武汉大学、东北大学、华中科技大学等。例如，中国科学院、上海科技大

学、华东师范大学等联合研制了 C-POREL 系统，武汉大学研制了分布式数据库系统 WDDB，东北大学设计和实现了分布式数据库系统 DMU/FO，华中科技大学在国产关系数据库系统 DM2(达梦 2)的基础上推出了分布式版本 DM2-C。

14.1.2　分布式数据库的概念

分布式数据库系统(Distributed Database Systems，DDBS)通常指物理上分布而逻辑上集中的数据库系统。分布式数据库系统通常使用较小的计算机系统，每台计算机可单独放在一个地方，每台计算机中都有 DBMS 的一份完整副本，并具有自己局部的数据库，位于不同地点的许多计算机通过网络互相连接，共同组成一个完整的、全局的大型数据库。

在分布式数据库系统中，每个站点(Site)自身具有完全的本地数据库系统，所有站点协同工作，组成了一个逻辑上的统一数据库。站点数据由分布式 DBMS(DDBMS)管理——DBMS+分布式扩展模块。分布式数据库系统中的应用可分为本地应用(局部应用)和全局应用，用户也可以相应地分为本地用户和全局用户。本地应用和本地用户只访问其所注册的那个站点上的数据，而全局应用和全局用户则访问其涉及的多个站点上的数据。以银行系统为例，目前很多银行都建立了全国通存通兑系统。利用这些系统，不仅可以使一个支行的用户通过访问支行的账目数据库来完成现金的存取等交易，实现局部应用，还可以通过计算机网络实现异地异行现金转账等业务，从一个支行的账户中转出若干金额到另一个支行的账户中，实现同时访问两个支行(异地)上的数据库的全局应用。

图 14-1 给出了分布式数据库系统的示意图。

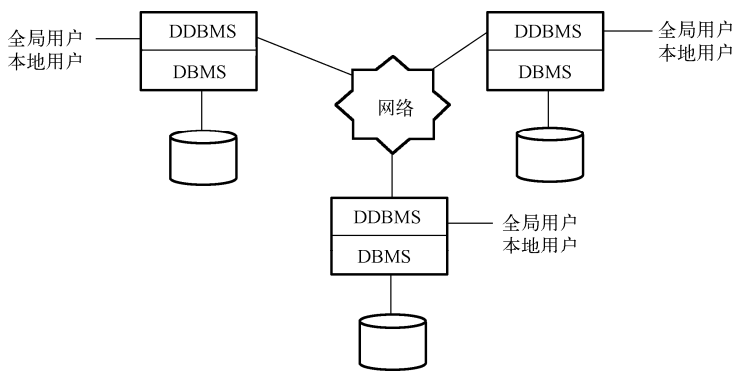


图 14-1　分布式数据库系统示意

1. 分布式数据库系统的特点

分布式数据库系统的特点如下。

1)物理分布性

在分布式数据库系统中，数据是物理分布在不同的站点上的。不同站点之间相距甚远(如在几十千米以上)，也可以相距甚近(如在同一个大楼里)。站点之间都用通信网络联系，在每一个站点上一般使用一个集中式数据库系统。

2)逻辑整体性

分布式数据库系统中的数据虽然在物理上分布在不同的站点，但各个站点上的数据在

逻辑上属于同一个系统，对用户而言是一个整体，就像一个集中式数据库一样。物理分布、逻辑整体是分布式数据库系统最主要的特点。因此，如果简单地将多个集中式数据库系统通过网络相连接，并不能构成分布式数据库系统，因为这种方法不能对用户应用提供一个统一的数据库视图，用户在访问数据时必须要显式地说明数据的存储站点等信息。

3) 站点自治性

分布式数据库系统强调站点自治，即每个站点都拥有自己独立的本地数据库系统、操作系统、CPU 等，也有专门的数据库管理人员，具有高度的自治能力。在一些分布式数据库系统中，还允许每个站点有异构的数据库系统。站点自治性使得本地应用都可以在单个站点上完成，有利于系统性能的提升。

4) 数据透明性

由于分布式数据库系统的物理分布特性，要使其能够支持涉及多个站点的全局应用，以便于全局应用的用户使用分布式数据库系统时，将主要精力集中在应用的逻辑上，而不是数据的位置分布上，因此，分布式数据应提供了数据透明性，即用户不需要知道数据的物理位置以及如何访问某个特定站点的数据。

数据透明性包括位置透明性、复制透明性和分片透明性。位置透明性是指用户和应用程序不必知道它所用的数据在什么站点。用户所要使用的数据很可能在本地的数据库中，也可能在外地的数据库中。系统提供位置透明性时，用户就不必关心数据在本地还是外地的数据库中，应用程序的逻辑变得简单，而且允许在数据使用的方式改变时，不必重写程序，这样避免了应用程序的频繁变更，也降低了应用程序的复杂程度。复制透明性是指在分布式数据库系统中，为了提高系统的性能和可用性，将部分数据同时重复地存放在不同的站点，这样，本地数据库中也可能包含外地数据库中的数据。应用程序运行时，就可以在本地数据库的基础上运行，尽量不借助通信网络与外地数据库联系，而用户还以为在使用外地数据库中的数据。这样可以避免站点之间的通信开销，加快应用程序的运行速度，对查询操作比较有利。但是，各个站点大量复制其他站点的数据会使数据的更新操作涉及所有复制数据，带来数据一致性维护的代价，也会加大系统的维护开销。分片透明性是指用户不需要知道数据库中的数据是如何分片的。数据分片是指将数据分割成不同的片段，例如，将学生数据分为计算机系学生片段和电子系学生片段，并且可以将计算机系学生片段存在站点 *A*，将电子系学生片段存在站点 *B*。但这种数据分片对于用户是透明的。图 14-2 给出了分片透明性的一个示例。在这个例子中，学生数据被分成了两个片段 CS_Student 和 EE_Student，并且存储在站点 *A* 和站点 *B* 上，当用户在站点 *C* 上发出学生查询时，系统会自动完成分片上的查询以及结果汇总工作，用户并不知道这些与分片相关的处理细节。

2. 分布式数据库系统的分类

分布式数据库系统可以按照许多方式进行分类。例如，可以根据各个站点本地 DBMS 及其依赖的数据模型来分，也可以根据分布式数据库系统的全局控制方式来分。

根据构成各个站点本地 DBMS 及其依赖的数据模型，可以把分布式数据库系统分为下面三类。

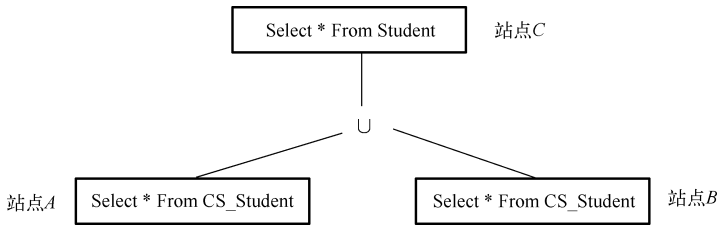


图 14-2　分片透明性示例

(1) 同构同质型 DDBS。各个站点都采用同一类型的数据模型(如都是关系数据模型)，并且都是同种型号的数据库管理系统(如都是 Oracle 系统)。

(2) 同构异质型 DDBS。各个站点都采用同一类型的数据模型，但是数据库管理系统的类型(或型号)是不同的，如 DB2、Sybase、Oracle 等。

(3) 异构型 DDBS。各个站点的本地 DBMS 采用了不同类型的数据模型。例如，有的站点是关系 DBMS，有的站点是面向对象 DBMS。

按照分布式数据库系统的全局控制方式，可以将分布式数据库系统分为以下三类。

(1) 全局控制集中的 DDBS。全局控制集中的 DDBS 的特点是全局控制信息集中在某一站点上，由该站点完成全局事务的协调和局部数据库转换等一切控制功能。全局数据字典只有一个，也存放在该站点上，它是分布式数据库系统进行全局控制的主要依据。

这类结构的优点是控制简单，容易实现更新一致性，但由于控制集中在某一特定的站点上，不仅容易形成瓶颈，而且系统比较脆弱，一旦该结点出故障，整个系统就将瘫痪。

(2) 全局控制分散的 DDBS。全局控制分散的 DDBS 的特点是全局控制信息分散在网络的每一个站点上，全局数据字典也在每个站点上存放一份。每个站点都能完成全局事务的协调和局部数据库转换的控制功能，每个站点既是全局事务的参与者，又是全局事务的协调者。一般称这类结构为完全分布的 DDBS。

这类结构的优点是站点独立，自治性强，单个站点退出或进入系统均不会影响整个系统的运行，但是全局控制的协调机制和一致性的维护都比较复杂。

(3) 全局控制部分分散的 DDBS。全局控制部分分散的 DDBS 的特点是根据应用的需要将全局控制信息和全局数据字典分散在某些站点上，它是介于前两种情况之间的体系结构。

14.1.3　分布式数据库管理系统的组成

分布式数据库管理系统(简称 DDBMS)是建立、管理、维护分布式数据库的一组软件，一般由四部分组成，如图 14-3 所示。

(1) LDBMS。LDBMS(Local DBMS)是局部站点上的数据库管理系统，其功能是建立和管理局部数据库，提供站点自治能力，执行局部应用及全局查询的子查询。

(2) GDBMS。GDBMS(Global DBMS)是全局数据库管理系统，其主要功能是提供数据透明性，协调全局事务的执行，协调各局部 DBMS 以完成全局应用，保证数据库的全局一致性，执行并发控制，实现更新同步，提供全局恢复功能等。

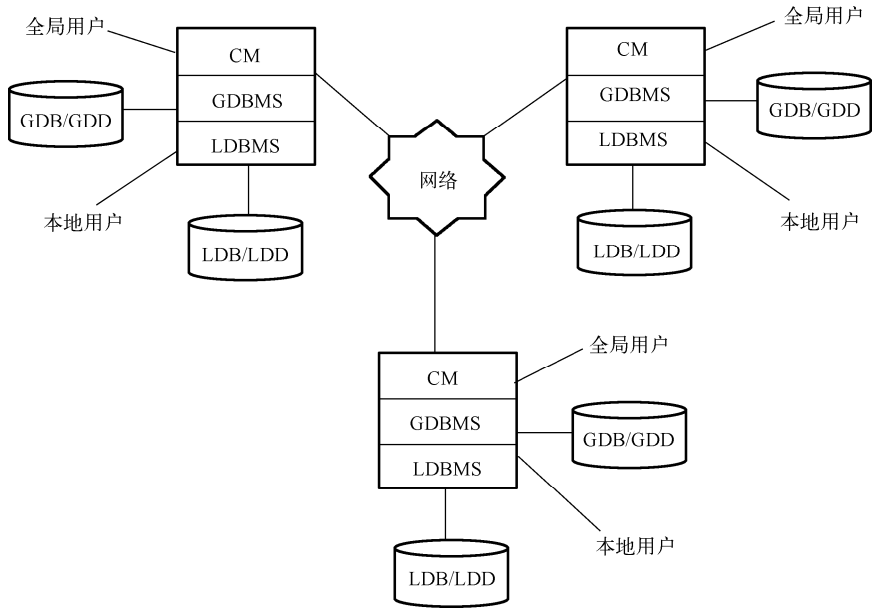


图 14-3　分布式数据库管理系统的组成
CM：通信管理器

(3) 全局数据字典。全局数据字典(Global Data Directory，GDD)用来存放全局概念模式、分片模式、分配模式的定义以及各模式之间映像的定义，也存放用户存取权限的定义，以保证全部用户的合法权限和数据库的安全性；另外，还存放数据完整性约束条件的定义，其功能与集中式数据库的数据字典类似。

(4) 通信管理器。通信管理器(Communication Manager，CM)负责在分布式数据库的各站点之间传送消息和数据，完成通信功能。

14.1.4　数据分片与分配

数据分片与分配是分布式数据库系统中的核心问题。分布式数据库系统中的复制透明性、分片透明性等特点都和数据的分片与分配有关系。

1. 数据分片

在分布式数据库系统中，全局数据库是由各个局部数据库逻辑组合而成的，局部数据库是全局数据库的某种逻辑分割。

数据分片的方法主要有下面几种。

(1) 水平分片：将数据分割成不相交的元组集合，每个子集为一个逻辑片段。水平分片可通过关系代数中的选择运算来实现，全局关系可通过分片的 Union 操作来得到。

(2) 垂直分片：把全局关系的属性集分成若干子集，可通过投影运算来实现。

(3) 混合分片：前面几种方法的混合，可以是先水平后垂直，或者先垂直后水平，两种混合方式得到的结果有差别。

关系数据库上的数据分片和数据库模式分解一样，必须满足一定的约束条件。这些约束条件如下。

(1)完备性条件：必须把全局关系的所有数据映射到片段中，决不允许有属于全局关系的数据却不属于它的任何一个片段。

(2)可重构条件：必须保证能够由同一个全局关系的各个片段来重建该全局关系。对于水平分片，可用并操作重构全局关系；对于垂直分片，可用连接操作重构全局关系。

(3)不相交条件：要求一个全局关系被分割后所得的各个数据片段互不重叠(垂直分片的主键除外)。

例如，假设有员工关系模式 *E*(E#, Ename, Age, Sex, Salary)，下面给出了该关系模式上的几种分片结果。

(1)水平分片结果：将员工关系分割为男员工和女员工两个片段。

```
Define Fragment SHF1
AS Select * From E Where Sex='M'

Define Fragment SHF2
AS Select * From E Where Sex='F'
```

(2)垂直分片结果：将员工的 E#、Ename、Age、Sex 定义为一个片段，将 E#、Salary 定义为另一个片段。

```
Define Fragment SVF1
AS Select E#, Ename, Age, Sex From E

Define Fragment SVF2
AS Select E#, Salary From E
```

(3)混合分片结果：将男员工的 E#、Ename 分割为一个片段，将女员工的 E#、Salary 定义为另一个片段。

```
Define Fragment SF1
AS Select E#, Ename From SHF1

Define Fragment SF2
AS Select * From SVF2 Where Sex='F'
```

2. 数据分配

数据分配是指将数据分片分散存储到各个站点的策略。图 14-4 显示了数据分片与数据分配之间的关系。

数据分配可以采用以下几种不同的方式。

(1)集中式分配：将所有分片都存储在同一站点上。这种分配方式容易管理，一致性易保证，但可靠性差。

(2)分割式分配：每个分片分配存储到某个特定站点上。这种方式可充分利用站点存储资源，部分站点故障时系统仍可运行，可靠性较好，但全局查询代价高。

(3)复制式分配：每个分片在所有站点上都有副本。这种方式可靠性高，响应快，但数据同步代价高，冗余大。

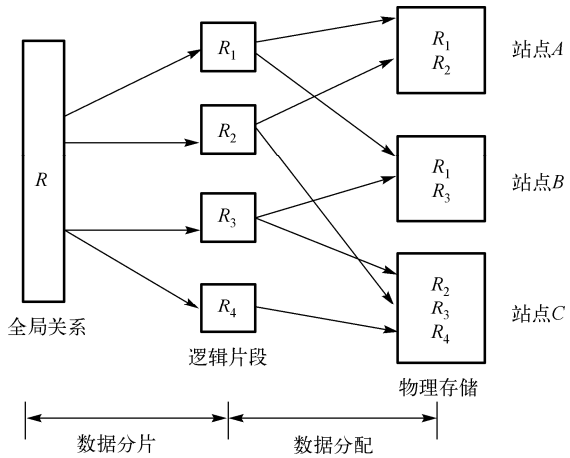


图 14-4　数据分片与数据分配之间的关系

(4) 混合式分配：所有分片划分成若干子集，每个子集存储于一个或多个站点上。这种方式灵活性好，兼有复制式分配和分割式分配的优点，但同时也兼有两者的缺点。

14.1.5　分布式数据库的模式结构

分布式数据库的模式结构分为四层，分别是全局外层、全局概念层、局部概念层和局部内层，如图 14-5 所示。其中全局外层和全局概念层是分布式数据库特有的模式结构，而局部概念层则对应着集中式数据库的概念模式层，局部内层对应着集中式数据库的内模式层。

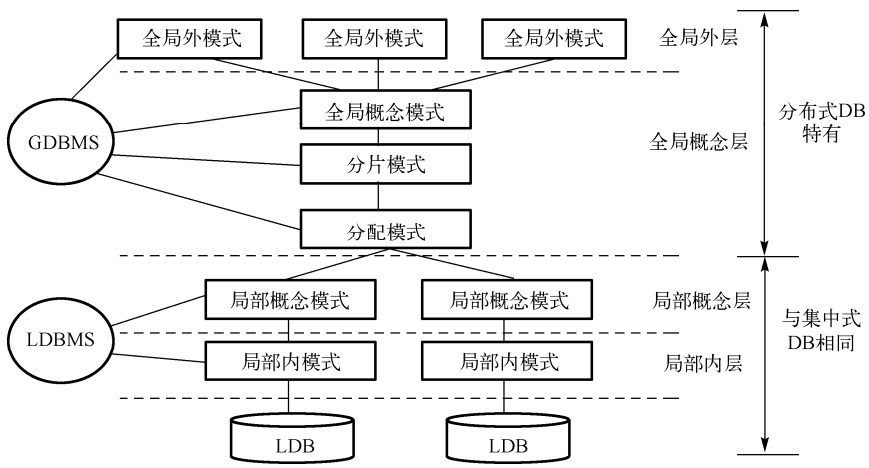


图 14-5　分布式数据库模式结构

1. 全局外层

全局外层是分布式数据库特定的全局用户对分布式数据库的最高层抽象，由多个用户视图组成。对于完全透明的分布式数据库，其全局外模式与集中式数据库的外模式概念类似，定义方式和使用方式也相同。但集中式数据库的外模式只是对具体站点上的局部数据

库的抽象，而分布式数据库的全局外模式是定义在由各局部数据库逻辑集合组成的全局概念模式之上的。

2. 全局概念层

全局概念层是分布式数据库的整体抽象，包含了全部数据特征和逻辑结构，定义了全局数据的逻辑结构、逻辑分布性和物理分布性，不涉及全局数据在每个局部站点上的物理存储细节，是全局的数据库视图。

全局概念层包括三种模式。

(1) 全局概念模式：描述分布式数据库全局数据的逻辑结构，是分布式数据库的全局概念视图，包括模式名、属性名、属性的数据类型的定义和长度等。

(2) 分片模式：描述全局数据的逻辑片段划分视图。

(3) 分配模式：描述数据分片与站点之间的局部物理存储映射，是划分后的片段的物理分配。

3. 局部概念层

局部概念层由局部概念模式描述，它是全局概念模式的子集。全局概念模式经逻辑划分后被分配在各局部站点上。

对于只支持全局应用的分布式数据库而言，其局部概念模式可理解为局部数据库的概念模式和外模式的组合。

对于支持本地应用的 DDB 而言，其局部概念模式与传统的集中式数据库一样还要划分为局部外模式和概念模式。

4. 局部内层

局部内层是分布式数据库中关于物理数据存储结构的描述，相当于集中式数据库的内模式层。

14.1.6　分布式数据库的优缺点

分布式数据库系统是在集中式数据库系统的基础上发展来的，其主要出发点是解决集中式数据库不能适应分布式应用的问题。比较分布式数据库系统与集中式数据库系统，可以发现分布式数据库系统具有下列优点。

(1) 更适合分布式的管理与控制。分布式数据库系统的结构更适合具有地理分布特性的组织或机构使用，允许分布在不同区域、不同级别的各个部门对其自身的数据实行局部控制。例如，实现全局数据在本地录入、查询、维护，由于计算机资源靠近用户，这样做可以降低通信代价，提高响应速度，而涉及其他站点数据库的数据只是少量的，从而可以大大减少网络上的信息传输量；同时，局部数据的安全性也可以做得更好。

(2) 具有灵活的体系结构。集中式数据库系统强调的是集中式控制，物理数据库是存放在一个站点上的，由一个 DBMS 集中管理。多个用户只可以通过近程终端或远程终端在多用户操作系统的支持下运行该 DBMS 来共享集中式数据库中的数据。而分布式数据库系统的站点局部 DBMS 的自治性，使得大部分的局部事务管理和控制问题都能就地解决，只

有在涉及其他站点的数据时才需要通过网络作为全局事务来管理。分布式 DBMS 可以设计成具有不同程度的自治性，从具有充分的站点自治到几乎是完全集中式的控制。

(3) 系统经济，可靠性高，可用性好。与一个大型计算机支持的一个大型的集中式数据库再加一些近程终端和远程终端相比，由超级微型计算机或超级小型计算机支持的分布式数据库系统往往具有更高的性价比和实施灵活性。分布式系统比集中式系统具有更高的可靠性和更好的可用性。例如，由于数据分布在多个站点并有许多复制数据，个别站点或个别通信链路发生故障不至于导致整个系统的崩溃，而且系统的局部故障不会引起全局失控。

(4) 在某些情况下可提高系统性能。如果存取的数据在本地数据库中，那么就可以由用户所在的计算机来执行，速度较快。

(5) 可扩展性好，易于集成现有系统，也易于扩充。对于一个企业或组织，可以采用分布式数据库技术在已建立的若干数据库的基础上开发全局应用，对原有的局部数据库系统做某些改动，形成一个分布式系统。这比重建一个大型数据库系统要简单，既省时间，又省财力、物力。也可以通过增加站点数的办法，迅速扩充已有的分布式数据库系统。

但是，分布式数据库系统也存在一些缺点。

(1) 通信开销较大，故障率高。例如，在网络通信传输速度不高时，系统的响应速度慢，与通信相关的因素往往导致系统故障，同时系统本身的复杂性也容易导致较高的故障率。当故障发生后，系统恢复也比较复杂，可靠性有待提高。

(2) 数据的存取结构复杂。一般来说，在分布式数据库中存取数据比在集中式数据库中存取数据更复杂，开销更大。

(3) 数据的安全性和保密性较难控制。在具有高度站点自治的分布时数据库中，不同站点的局部数据库管理员可以采用不同的安全措施，但是无法保证全局数据都是安全的。安全性问题是分布式系统固有的问题。因为分布式系统是通过通信网络来实现分布控制的，而通信网络本身在保护数据的安全性和保密性方面存在弱点，数据很容易被窃取。

(4) 分布式数据库的设计、站点划分及数据在不同站点的分配比较复杂。分布式数据库系统中，数据的划分和分配对系统的性能、响应速度及可用性等具有极大的影响。不同站点的通信速度与局部数据库系统的存取部件的存取速度相比，是非常慢的。通信系统有较高的延迟，在 CPU 上处理通信信息的代价很高。分布式数据库系统中要注意解决分布式数据库的设计、查询处理和优化、事务管理及并发控制和目录管理等问题。

14.2　面向对象数据库技术

关系数据库技术从 20 世纪 70 年代提出之后，在 80 年代得到了快速发展，很好地满足了各行各业的数据组织和管理需求。但是，随着计算机软硬件技术的不断发展以及新需求的不断涌现，关系数据库技术在复杂数据管理方面也呈现出了一些缺点，如语义表达能力弱、不能很好地支持复杂数据表达等。在面向对象程序设计语言的启发下，研究者提出了面向对象数据库技术。面向对象数据库技术具有比关系数据库技术更强的语义表达能力，适合多媒体、GIS 等领域中的复杂数据管理。本节主要讨论面向对象数据库技术的特点，并对面向对象数据库技术和关系数据库技术的优缺点进行对比分析。

14.2.1　面向对象数据库的产生与发展

面向对象数据库(Object-Oriented Database，OODB)技术是在面向对象概念、数据库系统概念和新型应用需求的共同促进下产生的新型数据库技术。

1. 面向对象数据库的概念

传统的关系数据库技术假设数据具有很强的结构化特征。关系数据库所存储的数据的特点可归纳为如下几点。

(1)结构统一。关系数据库中的数据具有相同的结构(模式)，还具有很强的格式化特点。

(2)面向记录。在关系数据库中，对用户而言，数据库是一个记录的集合。所有数据都是以记录的形式存在的。

(3)数据小。关系数据库中的记录一般都比较短。

(4)原子属性。关系数据库中的每个属性值是无结构的，是不可分的原子值，满足 1NF 定义。

随着计算机技术的不断发展，一些新的应用开始出现，如 GIS、多媒体、超媒体、CAD 等。这些新型应用中的数据大都不具备传统的数据特征，因此，使用传统的关系数据库技术来表达和存储这些新型数据就遇到了困难。例如，在关系数据库中，如果要存储一段视频，现有的做法只能是将视频作为 BLOB(大二进制对象)存储在关系数据库的某个字段中。但是，视频上的一些操作，如快进、后退、片段截取等，在 SQL 中难以得到支持。而且 BLOB 数据也无法使用传统的 SQL 进行操作(Insert、Update 等)，必须借助额外的方法才能实现。因此，虽然关系数据库技术能够支持新型复杂数据的存储，但很难有效地支持这些复杂数据上的操作，更不用说支持其他的功能了，如复杂数据的索引、查询优化等。

另外，关系数据库技术在数据表达方面也存在低效的问题。根据关系数据模型，现实世界中的实体对应着关系中的一个元组，但是 1NF 的要求容易导致关系中出现冗余数据。例如，图 14-6 显示了一个关系数据库 1NF 导致冗余属性值的例子，在图中可以看到，由于 1NF 要求属性值必须是不可分的原子值，因此出现了很多冗余的属性值。

ISBN	Title	Author	Keyword	Publisher
062001	Compilers	J.Ullman	Compiler	Springer
062001	Compilers	J.Ullman	Grammar	Springer
062001	Compilers	A.Aho	Compiler	Springer
062001	Compilers	A.Aho	Compiler	Springer
062001	Compilers	J.Hopcroft	Compiler	Springer
062001	Compilers	J.Hopcroft	Compiler	Springer

图 14-6　关系数据库的 1NF 导致的冗余属性值

按照关系数据库的规范化技术，可以将图 14-6 的关系模式分解为三个关系模式，从而减少冗余，优化模式设计。图 14-7 给出了图 14-6 的规范化模式分解结果。

模式分解结果虽然解决了冗余表达的问题，但仍然存在以下问题。

(1)一本书在现实世界中是一个独立对象，而在图 14-7 的数据库设计中被人为地分割成几个关系。也就是说，关系数据模型不得不通过多个实体集(关系)来表达现实世界中的一类实体(图 14-8)。

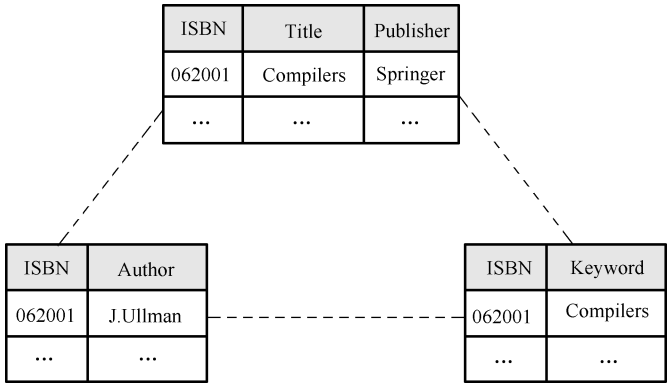


图 14-7　图 14-6 的规范化模式分解结果

图 14-8　现实世界与关系数据模型之间的认知鸿沟

(2)不能表达作者之间的顺序关系。

(3)SQL 可以执行分组操作，但只能返回单个的聚集函数值，不能返回一组结果，如“查询所有作者及其所写作的书列表”。

一种自然的解决方法是将图 14-6 所示的数据表示为图 14-9 所示的形式。这种表示和存储方式符合人们对数据的认知，并且也没有数据冗余问题。图 14-9 所示的结构可以称为嵌套关系(Nested Relation)。在嵌套关系中，关系中的值可以是一个集合或列表。但这不满足传统关系数据模型的 1NF 要求，因此需要一个允许在关系中自由使用集合和列表的类型系统。

ISBN	Title	Author	Keyword	Publisher
062001	Compilers	{J.Ullman, A.Aho, J.Hopcroft}	{Compiler, Grammar}	Springer

图 14-9　基于嵌套关系的自然解决方法

面向对象数据库技术就是在这样的背景下产生的。面向对象的概念最早出现在面向对象程序设计语言(Object-Oriented Programming Language，OOPL)中。面向对象概念提供了复杂类型系统，支持集合、列表等类型。面向对象概念与数据库系统概念以及新型应用需求相结合，最终产生了面向对象数据库技术，即在 DBMS 中可以支持复杂数据的高效表达和管理。

2. 面向对象数据库系统的实现方式

面向对象数据库系统有多种可选的实现方式，包括扩充命令式编程语言、扩充 OO 语言、OO 语言的底层演化以及对象关系数据库技术等。

(1)扩充命令式编程语言：通过扩充库使语言自身具有数据库功能，如面向对象数据库系统 O2 的实现采用了扩充 C 语言的方式。

(2)扩充 OO 语言：通过额外的类来提供持久化对象存储、查询等数据库功能。这种方

式是目前实现面向对象数据库系统最流行的方式，许多面向对象数据库系统(如 Versant、Ontos、ObjectStore 等)都采用了这种实现方式。

(3)OO 语言的底层演化：将数据库功能直接添加到 OO 语言中，使语言自身具备数据库支持能力，不需要额外的库，例如，面向对象数据库系统 GemStone 的实现采用了对 SmallTalk、Java、C++语言进行底层扩充的方式。

(4)对象关系数据库技术：通过扩充关系数据模型，使其支持对象概念。对象关系数据库技术允许用户在 DBMS 内核中添加自定义的数据类型和操作，以此来实现面向对象的数据管理。

3. 面向对象数据库技术的发展历史

面向对象数据库技术从 20 世纪 80 年代中期开始发展，1989 年 Atkinson 等发表了《面向对象数据库系统宣言》，提出了面向对象数据库的 13 条规则。

从 20 世纪 80 年代到 90 年代，出现了多个面向对象数据库系统，如 Jasmine、O2、Versant、Ontos 等。

1990 年，加利福尼亚大学伯克利分校的 M.Stonebraker 等发表了《第三代数据库宣言》，提出了对象关系数据库技术，并将对象关系数据库技术定义为关系数据库技术之后的下一代数据库技术。

自 20 世纪 90 年代，Oracle、DB2、Informix 等均开始在其关系数据库引擎中加入对对象关系数据库技术的支持，并提供了多个对象扩展模块，其中 Informix 还提供了对象关系扩展的集成开发环境。

1993 年，对象数据管理组织(Object Data Management Group，ODMG)推出了面向对象数据库标准，通常称为 ODMG93。

1997 年，ODMG 颁布了第二版面向对象数据库标准 ODMG 2.0。

1999 年，对象关系数据库技术被写入新版 SQL 标准 SQL3，成为目前商业化的主流技术。

14.2.2　面向对象数据模型

面向对象数据库系统支持面向对象数据模型(简称 OO 模型)。OO 模型以对象为基本的数据结构来表达现实世界中的实体以及实体间的联系，以 OO 模型为基础的面向对象数据库系统实质上是一个持久的、可共享的对象库的存储和管理者；而一个对象库是由一个 OO 模型所定义的对象的集合体。

1. 面向对象数据模型的基本概念

(1)对象与对象标识：现实世界的任一实体都被统一地模型化为一个对象(Object)，每个对象有唯一的标识，称为对象标识(Object Identifier，OID)。

(2)封装(Encapsulation)：每一个对象是其状态与行为的封装，其中状态是该对象一系列属性值的集合，而行为是在对象状态上操作的集合，操作也称为方法(Method)。

(3)类(Class)：共享相同属性和方法集的所有对象构成了一个对象类(简称类)，一个对象是某一类的一个实例(Instance)。例如，学生是一个类，李枫、张晨、杨敏等是学生中

的对象。在数据库系统中，要注意区分“型”和“值”的概念。在 OODB 中，类是“型”，对象是某一类的一个“值”。类属性的定义域可以是任何类，既可以是基本类，如整数、字符串、布尔型，也可以是包含属性和方法的一般类。特别地，一个类的某一属性的定义也可以是这个类自身。

(4)类层次(结构)：在一个面向对象数据库模式中，可以定义一个类(如 C_1)的子类(如 C_2)，类 C_1 称为类 C_2 的超类(或父类)。子类(如 C_2)还可以再定义子类(如 C_3)。

(5)消息(Message)：由于对象是封装的，对象与外部的通信一般只能通过消息传递，即消息从外部传送给对象，存取和调用对象中的属性与方法，在内部执行所要求的操作，操作的结果仍以消息的形式返回。

2. 对象结构与标识

对象是由一组数据结构和在这组数据结构上的操作程序所封装起来的基本单位，对象之间的联系是通过一组消息来定义的，包括如下内容。

(1)属性集合。属性描述对象的状态、组成和特性，所有属性的集合构成对象数据的数据结构。对象可以嵌套，组成复杂的对象。

(2)方法集合。方法描述对象的行为特性，包括方法的接口(方法调用的说明)和方法的实现(对象操作的算法)。

(3)消息集合。对象之间操作请示的集合。

在面向对象数据库中，每个对象有唯一的、不变的标识，对象中的属性、方法会随时间变化，但对象的标识始终不变。对象标识主要有如下几种。

(1)值标识。用值表示的标识，如关系数据库中元组的码。

(2)名标识。用名字表示的标识，如变量的名字。

(3)内标识。由系统自动给出的标识。在面向对象数据库中，内标识是最常见的对象标识形式。

3. 类结构与继承

在面向对象数据库模式中，一组类可以形成一个类层次。一个面向对象数据库模式可能有多个类层次。在一个类层次中，一个类继承其所有超类的属性、方法和消息。

图 14-10 给出了类层次的一个例子。类层次中底层的类和上层的类之间形成了自上而下的继承关系。

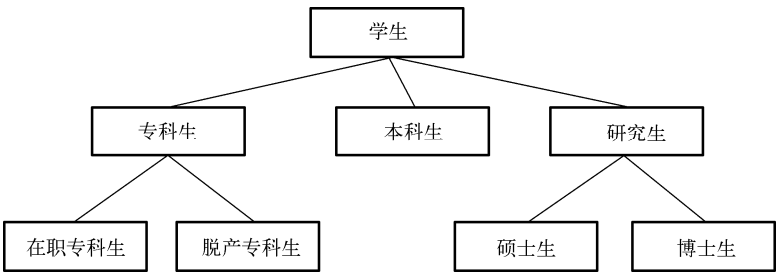


图 14-10　类层次示例

在面向对象数据库模式中，继承分为单继承和多重继承。其中，单继承是指一个子类只能继承一个超类的特性，多重继承是指一个子类能够继承多个超类的特性。图 14-11 给出了单继承和多重继承的示例。继承性是建立数据库模式的有力工具，它提供了对现实世界简洁而精确的描述。同时，继承性也提供了信息重用的机制。

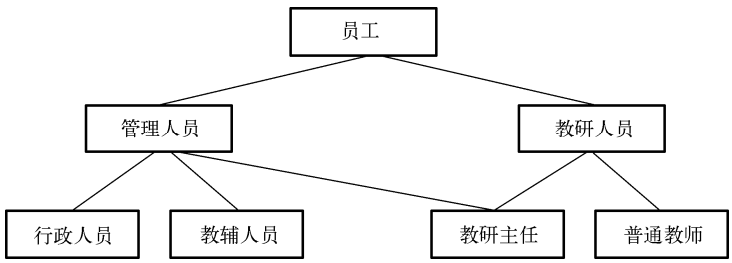


图 14-11　单继承和多重继承示例

在面向对象数据库模式中，对象的属性不仅可以是单值或多值的，还可以是一个对象，这就是对象的嵌套关系。如果对象 *B* 是对象 *A* 的某个属性，则称 *A* 是复合对象，*B* 是 *A* 的子对象。

对象的嵌套关系为用户提供了从不同的粒度观察数据库的方法。粒度就是数据库中数据细节的详细程度，细节越详细，粒度越高。

表 14-1 给出了关系数据模型和面向对象数据模型之间的对比。由于采用了面向对象的数据建模方法，面向对象数据库系统能够提供更高层次的数据抽象能力，从而呈现给用户一个与现实世界相符的数据模型。总体而言，面向对象数据模型在建模能力、语义表达能力等方面要优于关系数据模型。但关系数据模型的数据结构和操作方法更简单，而且拥有几十年的应用背景，已经成为行业标准，也很适合结构化数据的管理。另外，现实世界中有许多应用并不需要数据库系统提供像面向对象数据模型那样的复杂数据管理能力。因此，关系数据库技术并没有因为面向对象数据库技术的出现而退出市场。

表 14-1　关系数据模型和面向对象数据模型的对比

内容	关系数据模型	面向对象数据模型
基本数据结构	二维表	类
数据标识符	码	OID
静态性质	属性	属性
动态行为	关系操作	方法
抽象数据类型	无	有
封装性	无	有
数据间关系	外码，数据依赖	继承、组合
模式演化能力	弱	强

14.2.3　面向对象数据库语言

面向对象数据库语言(OODB 语言)用于描述面向对象数据库模式，说明并操纵类定义与实例对象。OODB 语言主要包括对象定义语言(ODL)和对象操纵语言(OML)。对象操纵语言中的一个重要子集是对象查询语言(OQL)。

1. 面向对象数据库语言的功能

(1)类的操纵与定义。面向对象数据库语言可以操纵类，包括定义、生成、存取、修改与撤销类。其中类的定义包括定义类的属性、操作特征、继承性与约束等。

(2)操作方法的定义。面向对象数据库语言可用于对象操作方法的定义与实现。在操作实现中，语言的命令可用于操作对象的局部数据结构。对象模型中的封装性允许操作方法由不同程序设计语言来实现，并且隐藏不同程序设计语言实现的事实。

(3)对象的操纵。面向对象数据库语言可以用于操纵(即生成、存取、修改与删除)实例对象。

2. 面向对象数据库语言的组成

面向对象数据库语言分为类的定义和操纵语言、对象的定义和操作语言、方法的定义和操作语言三类。

类的定义和操纵语言用于实现类的定义与操纵功能。对象的定义和操纵语言用于描述对象和实例的结构，并实现对对象和实例的生成、存取、修改以及删除等操作。方法的定义和操纵语言用于实现操作方法的定义功能。

14.3　对象关系数据库技术

面向对象数据库技术基于 OO 模型，从而可以提供比关系数据模型更为强的语义表达能力，如继承、聚合等。但是，目前面向对象数据库系统中通常采用了扩充 OO 语言的方法，使得现有的 OODBMS 与 OOPL 之间存在着很强的关联性。从某种意义上讲，OODBMS 就是具备了持久化能力的 OOPL。这种方法给用户应用程序的开发带来了很大的局限性。首先，前端的应用程序开发工具受限，例如，Versant 虽然提供了 Java 版本和 C++版本，但不支持其他多种流行的开发语言(如 C#)。其次，OODB 采用了不同于 SQL 的面向对象数据库语言，使得原来流行的 SQL 在 OODB 上得不到继承，而目前大多数的数据库程序员都是熟练的 SQL 程序员，因此从开发的角度分析，采用 OODB 显然会增加人员培训等方面的成本。

因此，一种更合理的方式是将面向对象特性扩充到关系数据模型当中，使得传统的关系数据模型也能够支持对象特性。这样一来，既可以使数据库具备 OO 模型较强的语义表达能力，也可以继续兼容 SQL，提高面向对象数据库技术的实用性。对象关系数据库(Object-Relational Database，ORDB)技术正是在这样的思路下提出的。由于 ORDB 同时具有面向对象数据模型和关系数据模型的特性，因此得到 IBM、Oracle 等关系数据库厂商的广泛支持。

对象关系数据库系统除了具有原来关系数据库系统的各种特点外，还应该具有以下特点。

(1)关系表的列可以是新的抽象数据类型(Abstract Data Types，ADT)。

(2)用户可以增加新的 ADT 以及定义 ADT 上的操作。

(3)支持引用类型、继承。

(4)传统的 SQL 仍可以在 ORDB 中使用。

下面给出了在 ORDB 中扩充 Address 抽象数据类型的例子。首先使用 SQL 中的 Create Type 语句创建 Address 抽象数据类型，然后就可以在表定义中像传统的数据类型一样使用 Address 类型。

```
Create Type Address
(   street Varchar(30),
    city Varchar(30),
    house_number  Varchar(20)
)
Employee (name: Char(10), salary Int, addr: Address)
```

上例中定义的 Address 抽象数据类型代表了对象的概念，可以在 SQL 语句中访问它的成员变量，例如：

```
Select name, addr.city From Employee
```

抽象数据类型的操作可以通过ORDB中的用户自定义操作来实现。例如，假设在ORDB中扩充了新的类型 Polygong，代表多边形对象。可以定义 Polygon 上的 Meet 操作来实现判断两个多边形是否存在相接拓扑关系的功能，从而可以通过下面的 SQL 查询来回答“求与中国相邻的国家名称”。

```
Country(name:Char(30), boundary: polygon)
Select C.name From Country C, Country D
Where D.name='China' and Meet(C.boundary, D.boundary)
```

对象关系数据库技术和面向对象数据库技术都提供了对面向对象概念的支持，可以看成面向对象数据模型的两种不同的实现方式。表 14-2 总结了对象关系数据库技术和面向对象数据库技术之间的相同点与不同点。

表 14-2　对象关系数据库技术和面向对象数据库技术的相同点与不同点

异同	对象关系数据库技术	面向对象数据库技术
相同点	两者都支持对象概念；都提供了 DBMS 的特征，如并发、恢复等	
不同点	具有对象扩展能力的 RDBMS，兼容 RDBMS，面向复杂数据应用，可以很好地支持 SQL	支持对象持久化的 OO 语言，与 OO 语言集成，面向以对象为中心的应用，一般以 OQL 为查询语言

14.4　NoSQL 数据库技术

关系数据库技术虽然满足许多应用的数据管理需求，但在某些功能的支持上存在不足，如大量数据的写入处理、为有数据更新的表做索引或表结构(Schema)变更、字段不固定时应用、对于简单查询需要快速返回结果等。为了弥补关系数据库的不足(特别是最近几年)，NoSQL 数据库出现了。关系数据库应用广泛，能进行事务处理和 Join 等复杂处理。相对地，NoSQL 数据库只应用在特定领域，基本上不进行复杂的处理，但它恰恰弥补了之前所列举的关系数据库的不足。

14.4.1　NoSQL 数据库的概念

NoSQL(是 Not only SQL 的缩写，早期的说法是 Not SQL，即非关系数据库)是指不使用传统的关系数据模型，而是使用如 Key-Value 存储、文档型的、列存储、图形数据库、XML 等方式存储数据的数据库技术。之所以不使用传统的关系数据模型，主要是因为它存储数据的方式发生了变化。例如，当需要存储发票的数据时，在传统的关系数据模型中，需要设计表的结构，然后使用服务器语言将其转化为实体对象，再传递到客户端(这就是对象关系映射)，而在 NoSQL 中，只要保存发票数据就可以了。NoSQL 不需要预先设计表和结构就可以储存新的数值。当然，如果项目中要保存的数据的确需要关系数据库才能完成，那么应该坚持使用关系数据库。

在关于 NoSQL 的网站 http://nosql-database.org 中，对 NoSQL 的描述是：非关系的，分布式的，开源的，而且可以垂直扩展。大多数 NoSQL 数据库具有以下特点：模式自由(Schema-Free)，支持数据冗余，简单的 API，基于最终一致性(Eventually Consistency)和 BASE 原则(而非 ACID 原则)。目前，Google 的 BigTable 与 Amazon 的 Dynamo 是非常成功的商业 NoSQL 实现。一些开源的 NoSQL 体系，如 Facebook 的 Cassandra、Apache 的 HBase，也得到了广泛认同。

大多数 NoSQL 数据库系统从名字上看不出什么相同之处，但是，它们通常在某些方面相同：它们可以处理超大量的数据。

总体而言，与关系数据库技术相比，NoSQL 数据库技术具有下面的一些特点。

(1)可以处理超大量的数据。

(2)通常运行在便宜的 PC 服务器集群上。

(3)高性能。NoSQL 的支持者称，通过 NoSQL 架构可以省去将 Web 或 Java 应用和数据转换成 SQL 查询的时间，执行速度变得更快。SQL 对于处理繁重的重复数据操作比较合适。但是，当数据库结构非常简单时，SQL 可能没有太大用处。

(4)没有过多的操作。虽然 NoSQL 的支持者也承认关系数据库提供了无可比拟的功能集合，而且在数据完整性上的发挥也绝对稳定，他们同时也表示，企业的具体需求可能没有那么多。

(5)Bootstrap 支持。因为 NoSQL 项目都是开源的，所以它们缺乏供应商提供的正式支持。关于这一点，它们与大多数开源项目一样，不得不从社区中寻求支持。

14.4.2　NoSQL 兴起的原因

随着互联网 Web 2.0 网站的兴起，非关系数据库现在成了一个极其热门的新领域，非关系数据库产品的发展非常迅速。而传统的关系数据库在应付 Web 2.0 网站，特别是超大规模和高并发的社交网络网站方面，已经显得力不从心，暴露了很多难以解决的问题。

Web 应用对数据库的需求可简单总结为如下几个方面。

(1)对数据库高并发读写的需求。Web 2.0 网站要根据用户个性化信息来实时生成动态界面和提供动态信息，所以基本上无法使用动态界面静态化技术，因此数据库并发负载非常高，往往要达到每秒上万次读写请求。关系数据库勉强可以承受上万次 SQL 查询，但是应付上万次 SQL 写数据请求，就无法承受了。其实对于普通的 BBS 网站，往往也存在对高并发写请求的需求。

(2)对海量数据的高效率存储和访问的需求。对于大型的社交网络网站，每天用户产生海量的用户动态。例如，国外的 Friendfeed 一个月就达到了 2.5 亿条用户动态，对于关系数据库来说，在一张 2.5 亿条记录的表里面进行 SQL 查询，效率是极其低下乃至不可忍受的。又如，大型 Web 网站的用户登录系统中，如腾讯、盛大等，动辄数以亿计的账户，关系数据库也很难应付。

(3)对数据库的高可扩展性和高可用性的需求。在基于 Web 的架构当中，数据库是最难进行横向扩展的。当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像 Web 服务器和应用服务器那样简单通过添加更多的硬件和服务结点来提升系统性能。对于很多需要提供 24h 不间断服务的网站来说，对数据库系统进行升级和扩展是非常艰难的事情，往往需要停机维护和数据迁移。

在上面提到的需求面前，关系数据库遇到了难以克服的障碍，而对于 Web 2.0 网站来说，关系数据库的很多主要特性往往无用武之地，主要表现在以下方面。

(1)无法满足数据库事务一致性的需求。很多 Web 实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性的要求也不高。因此数据库事务管理成了数据库高负载下一个沉重的负担。

(2)无法满足海量数据的管理需求。关系数据库解决海量数据存取问题的主要手段是索引、缓存等技术，但在 Web 应用尤其是 Web 2.0 应用中，数据量往往以 PB 计算。面对如此海量的数据，关系数据库系统的传统优化技术往往难以取得理想的性能。目前一般的理解是，一个关系数据库的基本表中的记录一般不能超过 1000 万条，否则性能就很难保证，需要人工分库分表。虽然 MySQL 等也提供了数据库集群的支持，但是关系数据库集群部署和配置复杂，主备之间备份和恢复不方便，动态扩容能力弱，而且动态数据迁移难以自动化实现。

(3)无法满足对复杂的 SQL 查询，特别是多表关联查询的需求。任何大数据量的 Web 系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的 SQL 报表查询，特别是 SNS 类型的网站，从需求以及产品设计角度上，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，以至于 SQL 的功能就被极大地弱化了。

(4)One Size Fits All 模式很难适用于截然不同的业务场景。关系数据库技术的核心思想是“统一”，即使用一个数据模型和一个数据库统一存储和管理所有的数据并且支持所有的应用。这种 One Size Fits All 的思路在互联网时代很难适应快速发展的应用模式。例如，联机数据分析(OLAP)和联机事务处理(OLTP)这两类应用中，一个强调高吞吐，另一个强调低延时，已经演化出完全不同的架构，使用同一套模型来抽象显然是不合适的。

14.4.3　关系数据库与 NoSQL 的对比

总体而言，关系数据库与 NoSQL 相比的优势在于：以完善的关系代数理论作为基础，有严格的标准，支持事务 ACID，提供严格的数据一致性，借助索引机制可以实现高效的查询，技术成熟，有专业公司的技术支持。劣势在于：可扩展性较差，无法较好地支持海量数据存储，采用固定的数据库模式，无法较好地支持 Web 2.0 应用，事务机制影响系统的整体性能等。NoSQL 的优势是可以支持超大规模数据存储，数据分布和复制容易，灵活的数据模型可以很好地支持 Web 2.0 应用，具有强大的横向扩展能力等。其劣势是缺乏数学理论

基础，复杂查询性能不高，大都不能实现事务强一致性，很难实现数据完整性，技术尚不成熟，缺乏专业团队的技术支持，维护较困难，目前处于百花齐放的状态，用户难以选择等。

表 14-3 总结了关系数据库与 NoSQL 之间的对比。可以看到，目前关系数据库和 NoSQL 各有优缺点，彼此都无法取代。关系数据库适合电信、银行等领域的关键业务系统，需要保证事务强一致性。NoSQL 应用场景主要是互联网企业、传统企业的非关键业务(如数据分析)。

表 14-3　关系数据库与 NoSQL 对比

比较标准	关系数据库	NoSQL	备注
数据库原理	完全支持	部分支持	RDBMS 以关系代数理论作为基础，NoSQL 没有统一的理论基础
数据规模	大	超大	RDBMS 很难实现横向扩展，纵向扩展的空间也有限，性能会随着数据规模的增大而降低。 NoSQL 易于通过添加更多的设备来支持更大规模的数据
数据库模式	固定	灵活	RDBMS 需要定义数据库模式，严格遵守数据定义和相关约束条件。 NoSQL 不存在数据库模式，可以自由灵活定义并存储各种不同类型的数据
查询效率	快	可以实现高效的简单查询，但是不具备高度结构化查询等特性，复杂查询的支持弱	RDBMS 借助于索引机制可以实现快速查询(包括记录查询和范围查询)。 很多 NoSQL 数据库没有面向复杂查询的索引，虽然 NoSQL 可以使用 MapReduce 来加速查询，但是，在复杂查询方面的性能仍然不如 RDBMS
一致性	强一致性	弱一致性	RDBMS 严格遵守事务 ACID 模型，可以保证事务强一致性。 NoSQL 放松了对事务 ACID 的要求，而是遵守 BASE 模型，只能保证最终一致性
数据完整性	容易实现	很难实现	RDBMS 易于实现数据的完整性，例如，通过主键来实现实体完整性，通过外键来实现参照完整性，通过 Check 约束或者触发器来实现用户自定义完整性。 NoSQL 无法实现数据完整性
扩展性	一般	好	RDBMS 难以实现横向扩展，纵向扩展的空间也比较有限。 NoSQL 在设计之初就考虑了横向扩展的需求，可以很容易通过添加廉价设备实现扩展
可用性	好	很好	RDBMS 为了保证严格的数据一致性，只能提供相对较弱的可用性。 大多数 NoSQL 都能提供较高的可用性
标准化	是	否	RDBMS 已经标准化(SQL) NoSQL 还没有行业标准，不同的 NoSQL 数据库都有自己的查询语言和应用程序接口。NoSQL 缺乏统一查询语言，NoSQL 发展缓慢
技术支持	高	低	RDBMS 经过几十年的发展，已经非常成熟，Oracle 等大型厂商都可以提供很好的技术支持。 NoSQL 在技术支持方面仍然处于起步阶段，还不成熟，缺乏有力的技术支持
可维护性	复杂	复杂	RDBMS 需要专门的数据库管理员(DBA)维护。 NoSQL 数据库虽然没有 RDBMS 复杂，但也难以维护

此外，越来越多的企业开始在实际业务系统中采用混合架构。例如，亚马逊公司使用不同类型的数据库来支撑它的电子商务应用：对于“购物篮”这种临时性数据，采用键值存储会更加高效，当前的产品和订单信息则适合存放在关系数据库中，大量的历史订单信息则适合保存在类似 MongoDB 的文档数据库中。

14.4.4　NoSQL 数据库的主要类型

NoSQL 数据库目前主要的类型有键值数据库、列存储数据库、文档数据库和图数据库等。当然，由于 NoSQL 数据库目前没有规范的定义和分类，这几种类型只能代表最常见的类型。本节简要介绍有代表性的几种实现方式。有些网站将所有非关系数据库技术都归入 NoSQL 范畴，包括面向对象数据库技术、对象关系数据库技术等，但这些分类方法还有待商榷，本书不对此进行定论。

图 14-12 给出了键值数据库、列存储数据库、文档数据库和图数据库的数据结构的简单对比。

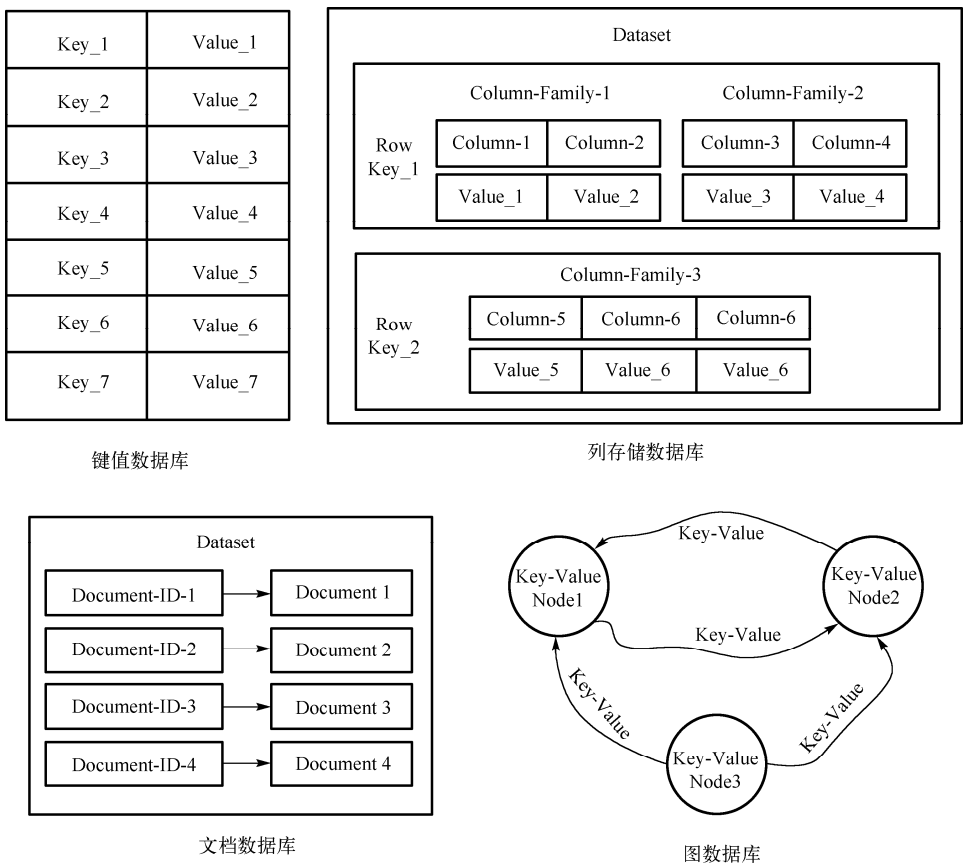


图 14-12　键值数据库、列存储数据库、文档数据库和图数据库的数据结构对比

1. 键值数据库

键值数据库是最常见的 NoSQL 数据库，它的数据是以 Key-Value 的形式存储的。虽然

它的处理速度非常快，但是基本上只能通过 Key 的完全一致查询获取数据。这里的键是一个字符串对象，值可以是任意类型的数据，如整型、字符型、数组、列表、集合等。

根据数据的保存方式，其可以分为临时性、永久性和两者兼具三种。

1) 临时性

Memcached 属于这种类型。临时性就是“数据有可能丢失”的意思。Memcached 把所有数据都保存在内存中，这样保存和读取的速度非常快，但是当 Memcached 停止的时候，数据就不存在了。由于数据保存在内存中，所以无法操作超出内存容量的数据(旧数据会丢失)。

2) 永久性

Tokyo Tyrant、Flare、ROMA 等属于这种类型。和临时性相反，永久性就是“数据不会丢失”的意思。这里的键值存储不像 Memcached 那样在内存中保存数据，而是把数据保存在硬盘上。它与 Memcached 在内存中处理数据比起来，由于必然要发生对硬盘的 I/O 操作，所以性能上还是有差距的，但数据不会丢失是它最大的优势。

3) 两者兼具

Redis 属于这种类型。Redis 有些特殊，临时性和永久性兼具，且集合了临时性 Key-Value 存储和永久性 Key-Value 存储的优点。Redis 首先把数据保存到内存中，在满足特定条件(默认是 15min 内 1 个以上，5min 内 10 个以上，1min 内 10000 个以上的 Key 发生变更)的时候将数据写入到硬盘中。这样既确保了内存中数据的处理速度，又可以通过写入硬盘来保证数据的永久性。这种类型的数据库特别适合处理数组类型的数据。

键值数据库适用于涉及频繁读写、拥有简单数据模型的应用，如内容缓存、会话、配置文件、参数、购物车等存储配置和用户数据信息的移动应用。键值数据库的缺点是不提供通过值进行查询的途径，不能通过两个或两个以上的键来关联数据，另外，也缺乏对事务的支持。

目前，键值数据库已经成为理想的数据缓冲区解决方案，Memcached、Redis 等在互联网领域广泛应用。图 14-13 给出了利用键值数据库 Memcached 构建数据缓冲区的典型解决方案。所有的数据都存储在关系数据库中，当首次访问数据时，将数据读出后保存在 Memcached 中，之后再次访问数据时直接从 Memcached 中读取。与传统的关系数据库中的缓冲区相比，Memcached 提供了更多的数据类型和数据结构的支持，并且支持持久化、恢复、高并发、高扩展等功能，因此更适合海量数据上的高并发应用。

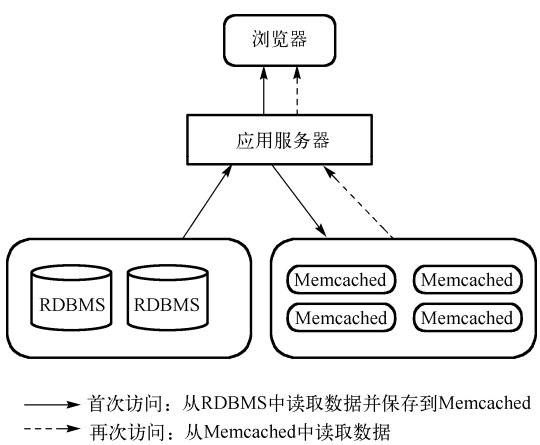


图 14-13　利用键值数据库构建数据缓冲区

2. 列存储数据库

Cassandra、HBase、HyperTable 属于列存储数据库。由于近年来数据量出现爆发性增长，这种类型的 NoSQL 数据库尤其引人注目。面向列存储的 NoSQL 数据库具有下面的特性。

1) 列存储

普通的关系数据库都是以行为单位来存储数据的，擅长进行以行为单位的读入处理，如特定条件数据的获取。因此，关系数据库也称为面向行的数据库。相反，面向列存储的数据库是以列为单位来存储数据的，擅长以列为单位读入数据。

图 14-14 给出了传统行存储数据库和列存储数据库之间的对比。图 14-15 给出了一个具体例子。例子中给出了一个存储员工的表格。按照关系数据库的行存储模式，例子中的三条员工记录存储为三行记录，这些记录最终按记录结构一行一行地存储在磁盘块上。但是，按照列存储模式，这些记录的同一列数据组织成一行，然后存储到磁盘块上。

2) 高扩展性

基于列存储的 NoSQL 数据库具有高扩展性，即使数据增加，也不会降低相应的处理速度(特别是写入速度)，所以它主要应用于需要处理大量数据的情况。另外，利用列存储数据库的优势，把它作为批处理程序的存储器来对大量数据进行更新也是非常有用的。但由于列存储数据库跟现行数据库存储的思维方式有很大的不同，应用起来十分困难。

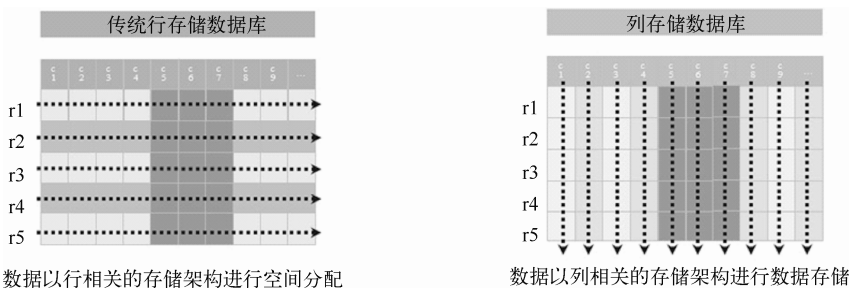


图 14-14　行存储数据库与列存储数据库的对比

EmpID	LastName	FirstName	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

数据格式(行存储)：
1, Smith, Joe, 40000;
2, Jones, Mary, 50000;
3, Johnson, Cathy, 44000;

数据格式(列存储)：
1, 2, 3;
Smith; Jones; Johnson;
Joe, Mary, Cathy;
40000; 50000; 44000;

图 14-15　行存储与列存储示例

最近，像 Twitter 和 Facebook 这样需要对大量数据进行更新与查询的网络服务不断增加，基于列存储的 NoSQL 数据库的优势对其中一些服务是非常有用的。

列存储数据库适合的应用包括：

(1) 分布式数据存储与管理；

(2) 数据在地理上分布于多个数据中心的应用程序；

(3) 可以容忍副本中存在短期不一致情况的应用程序；

(4) 拥有动态字段的应用程序；

(5) 拥有潜在大量数据的应用程序，大到几百 TB 的数据。

列存储数据库的优点是查找速度快，可扩展性强，容易进行分布式扩展，复杂性低；缺点是功能较少，大都不支持事务强一致性。

3. 文档数据库

MongoDB、CouchDB 属于文档数据库。它们属于 NoSQL 数据库，但与 Key-Value 存储相异。基于文档存储的 NoSQL 数据库具有下面的特性。

1) 不定义表结构

文档数据库即使不定义表结构，也可以像定义了表结构一样使用。关系数据库在变更表结构时比较费事，为了保持一致性还需修改程序。然而 NoSQL 数据库则可省去这些麻烦(通常程序都是正确的)，确实方便快捷。

2) 可以使用复杂的查询条件

跟键值数据库不同的是，基于文档存储的 NoSQL 数据库可以通过复杂的查询条件来获取数据。虽然它不具备事务处理和连接关系数据库所具有的数据处理能力，但除此以外的其他处理基本上都能实现。这是非常容易使用的 NoSQL 数据库。

文档数据库存储的基本对象是文档(Document)。文档实质上也是由键值对构成的。图 14-16 给出了一个文档的例子。图 14-17 给出了文档数据库 MongoDB 与关系数据库之间一些关键概念的对比。文档数据库中的文档是数据库中的基本对象，相当于关系数据库中的元组。文档的集合相当于关系数据库中的表。

```
{"_id":      "37010"
 "city":     "ADAMS",
 "pop":      2660,
 "styate":   "TN",
 "councilman": {name: "John Smith"
           address: "13 Scenic Way"
              }
}
```

图 14-16　文档数据库中的文档示例

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard (数据水平切分到不同物理节点)

图 14-17　文档数据库 MongoDB 与关系数据库之间的概念对比

文档数据库是目前使用最广泛的一类 NoSQL 数据库。它适合存储、索引并管理面向文档的数据或者类似的半结构化数据，如用于后台具有大量读写操作的网站、使用 JSON 数据结构的应用、使用嵌套结构等非规范化数据的应用程序等。由于现实应用中需要存储

多源异构数据的场景越来越多，因此使用文档数据库作为异构数据的统一数据库是一种流行的解决方案。但是，与其他 NoSQL 数据库一样，文档数据库缺乏统一的查询语法，对于事务的支持较弱，并且也不支持文档间的事务。

4. 图数据库

图数据库是另一种流行的 NoSQL 数据库。随着社交网络、生物信息学等应用的发展，图数据(或者称为图谱数据)成为应用中的一类主要数据。图数据库以点、边为基础存储单元，以高效存储、查询图数据为主要的目标。如果把社交网络中的用户看作图的结点，可以利用用户的关注、转发、点赞等行为构建结点之间的边，从而形成如图 14-18 所示的图数据结构。由于社交网络用户数多，数据量大，因此所形成的图数据也远远超出传统关系数据库所能处理的范围。而且，关系数据库的基本结构是二维表，并不适合执行专门的图查询操作，如子图查询、图相似性查询等。

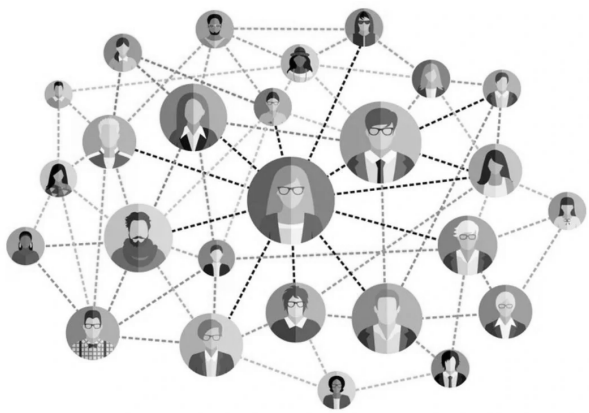


图 14-18　社交网络中的图数据结构示例

图数据库对数据的存储、查询以及其数据结构都和关系数据库有很大的不同。图数据库直接存储了结点之间的依赖关系，而关系数据库和其他类型的非关系数据库则以非直接的方式来表示数据之间的关系。图数据库把数据间的关联作为数据的一部分进行存储，关联上可添加标签、方向以及属性，而其他数据库针对关系的查询必须在运行时进行具体化操作，这也是图数据库在关系查询上相比其他类型的数据库有巨大性能优势的原因。

目前常用的图模型主要包含属性图、RDF 图两种。属性图由顶点、边及其属性构成。顶点和边都可以带有属性，结点可以通过“标签”进行分组。表示关系的边总是从一个开始点指向一个结束点，而且边一定是有方向的，这使得图成为有向图。关系上的属性可以为结点的关系提供额外的元数据和语义。RDF(Resource Description Framework)图在顶点和边上没有属性，只有一个资源描述符，这是 RDF 图与属性图间最根本的区别。在 RDF 图中每增加一条信息都要用一个单独的结点表示。例如，若在图中给表示人的结点添加姓名，在属性图中只需要在结点添加属性即可，而在 RDF 图中必须添加一个新的结点，并用 hasName 与原始结点相连。

目前常见的图数据库包括 Neo4J、OrientDB、InfoGrid、InfiniteGraph、GraphDB 等。

图数据库适合处理具有高度相互关联关系的数据，比较适用于解决社交网络、模式识别、依赖分析、推荐系统以及路径寻找等问题。其优点是灵活性高，支持复杂的图算法，可用于构建复杂的关系图谱。

14.4.5　常见的 NoSQL 开源数据库

NoSQL 数据库目前采用的是开源软件维护方式。下面简要介绍几种目前常见的 NoSQL 开源数据库。

1．LevelDB 和 RocksDB

LevelDB 是 Google 开源的持久化键值单机数据库，可看作开源版的 BigTable。LevelDB 具有很高的随机写、顺序读写性能，但是随机读的性能很一般，也就是说，LevelDB 很适合应用在查询较少，而写很多的场景。LevelDB 底层存储结构与 BigTable 一样使用了 LSM-Tree（Log Structured Merge Tree）。LSM-Tree 通过一种类似于归并排序的方式高效地将随机写转换为顺序写并批量写回磁盘，因此具有很高的写性能。

图 14-19 显示了 LevelDB 的架构。它包括内存结构（Memtable、Immutable Memtable、读缓存）和磁盘结构（SSTable 构成的 Level）。当写入数据时，数据先写入 Memtable。如果 Memtable 满了，则转换为 Immutable Memtable。Immutable Memtable 只允许读，不允许写入。内存中的 Immutable Memtable 超过一定数量后 Flush 到磁盘的 Level 0。Flush 操作是完全顺序写，所以写性能很高。后续如果 Level 0 的数据量或者文件数超过阈值，LevelDB 采用合并（Compact）操作选择一个文件与下一层 Key 范围重叠的所有文件进行合并，然后写入下一层。这个过程对于磁盘的写也是顺序写操作的，因此也同样具有很高的写性能。

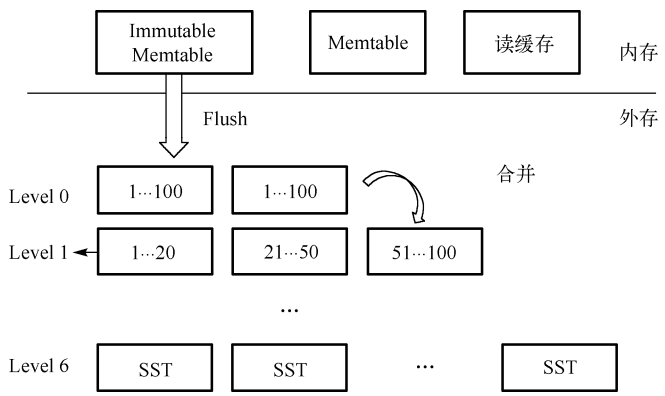


图 14-19　LevelDB 的架构

LevelDB 的主要特点：

（1）Key 和 Value 都是任意长度的字节数组。

（2）数据按 Key 值排序存储。

（3）调用者可以提供一个自定义的比较函数来重写排序顺序。

（4）提供基本的 Put（Key,Value）、Get（Key）、Delete（Key）操作。

（5）多个更改可以在一个原子批处理中生效。

(6) 用户可以创建一个瞬时快照(Snapshot)，以获得数据的一致性视图。

(7) 在数据上支持向前和向后迭代。

(8) 使用 Snappy 压缩库对数据进行自动压缩。

(9) 与外部交互的操作都被抽象成了接口(如文件系统操作等)，因此用户可以根据接口自定义的操作系统交互。

LevelDB 的主要缺点：

(1) 不支持 SQL 语句，也不支持索引。

(2) 一次只允许一个进程访问一个特定的数据库。

(3) 该数据库没有内置的 C/S 支持，有需要的用户必须自己封装。

RocksDB 是 Facebook 公司在 LevelDB 基础之上开发的键值数据库系统。图 14-20 显示了 RocksDB 的架构。它主要针对 LevelDB 进行了优化，具体如下。

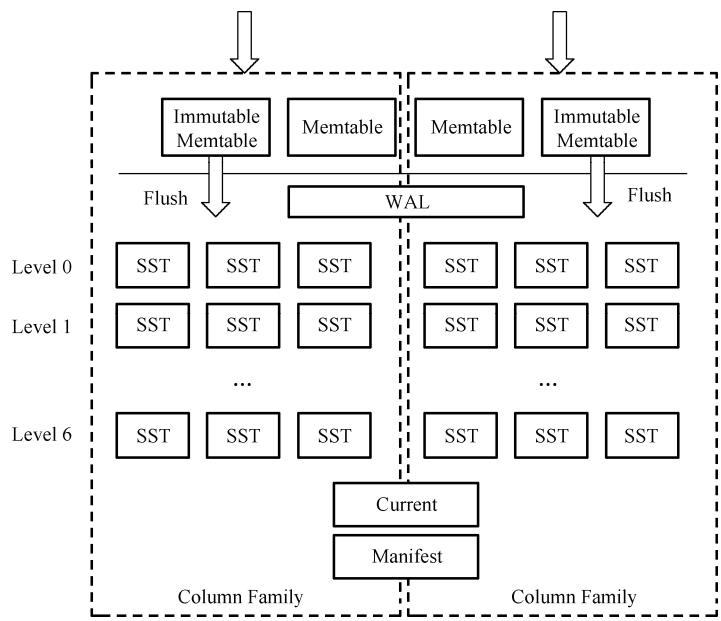


图 14-20　RocksDB 的架构

(1) LevelDB 仅支持单线程合并文件，RocksDB 可以支持多线程合并文件，充分利用多核的特性，加快文件合并的速度，避免文件合并期间的系统停顿。

(2) LevelDB 只有一个 Memtable，若 Memtable 满了还没有来得及持久化，则会减缓 Put 操作，从而引起系统停顿；RocksDB 支持管道式的 Memtable，也就说允许根据需要开辟多个 Memtable，以解决 Put 与合并操作速度差异的性能瓶颈问题。

(3) LevelDB 只能获取单个键值对；RocksDB 支持一次获取多个键值对，还支持 Key 范围查找。

(4) LevleDB 不支持备份；RocksDB 支持全量和增量备份。RocksDB 允许将已删除的数据备份到指定的目录，供后续恢复。

(5) 压缩方面，RocksDB 可采用多种压缩算法，除了 LevelDB 用的 Snappy，还有 Zlib、Bzip2。

(6) RocksDB 增加了对列族(Column Family)的支持，支持将多个不相关的数据集隔离存储在同一个数据库中。

LevelDB 和 RocksDB 是目前学术界研究较多的开源键值数据库系统。实际中，可以将 LevleDB 或者 RocksDB 作为数据存储系统引擎，在其上面实现分片和多副本，从而实现一个真正的分布式存储系统。国内许多开源数据库产品也都是以 RocksDB 为蓝本进行优化和二次开发得到的。

2. MongoDB

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似 JSON 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。它的特点是高性能、易部署、易使用，存储数据非常方便。

MongoDB 的主要功能特性如下。

(1) 面向集合，可以存储对象类型的数据。面向集合(Collection-Oriented)是指数据被分组存储在数据集中，称为一个集合(Collection)。每个集合在数据库中都有唯一的标识名，并且可以包含无限数目的文档。集合的概念类似关系数据库里的表，不同的是它不需要定义任何模式。

(2) 模式自由。模式自由，意味着对于存储在 MongoDB 中的文件，不需要知道它的任何结构定义。如果需要，完全可以把不同结构的文件存储在同一个数据库里。

(3) 支持动态查询。

(4) 支持完全索引，包含内部对象。

(5) 支持查询。

(6) 支持复制和故障恢复。

(7) 使用高效的二进制数据存储，包括大型对象(如视频等)。

(8) 支持自动处理碎片，以支持云计算层次的扩展性。

(9) 支持 Ruby、Python、Java、C++、PHP 等多种语言。

(10) 文件存储格式为 BSON(一种 JSON 的扩展)。BSON(Binary Serialized Document Format)存储形式是指存储在集合中的文档被存储为 Key-Value 对的形式。Key 用于唯一标识一个文档，为字符串类型，而 Value 则可以是各种复杂的文件类型。

(11) 可通过网络访问。MongoDB 服务器可运行在 Linux、Windows 或 OS X 平台，支持 32 位和 64 位应用，默认端口为 27017。推荐其运行在 64 位平台，因为 MongoDB 在 32 位平台运行时支持的最大文件尺寸为 2GB。

3. HyperTable

HyperTable 是一个开源、高性能、可伸缩的数据库，它采用与 Google 的 BigTable 相似的模型。在过去数年中，Google 为在 PC 集群上运行的可伸缩计算基础设施设计建造了三个关键部分。第一个关键的基础设施是 Google File System(GFS)，这是一个高可用的文件系统，提供了一个全局的命名空间。它通过跨机器(和跨机架)的文件数据复制来达到高

可用性，并因此免受传统文件存储系统无法避免的许多失败的影响，如电源、内存和网络端口等失败。第二个关键的基础设施是名为 Map-Reduce 的计算框架，它与 GFS 紧密协作，帮助处理收集到的海量数据。第三个关键的基础设施是 BigTable，它是传统数据库的替代。BigTable 可以通过一些主键来组织海量数据，并实现高效的查询。HyperTable 是 BigTable 的一个开源实现，并且根据人们的想法进行一些改进。

4. Cassandra

Cassandra 是一套开源分布式 Key-Value 存储系统。它最初由 Facebook 开发，用于储存特别大的数据。Facebook 目前在使用此系统。

Cassandra 的主要特性有分布式、基于列存储结构和高伸展性。

Cassandra 是由一堆数据库结点共同构成的一个分布式 Key-Value 存储系统。Cassandra 上的一个写操作会被复制到其他结点上，对 Cassandra 的读操作也会被路由到某个结点上读取。对于一个 Cassandra 集群来说，扩展性能是比较简单的事情，只在集群里面添加结点就可以了。

Cassandra 以 Amazon 专有的分布式存储系统 Dynamo 为基础，结合了 BigTable 基于列族的数据模型以及 P2P 去中心化的存储架构。某种程度上可以将它称为 Dynamo 2.0。类似于 BigTable，Cassandra 也支持行列混合存储。它的功能比 Dynomo 更丰富，但支持度不如文档数据库 MongoDB。

和其他数据库比较，Cassandra 的特点如下。

⑴ 模式自由：使用 Cassandra，不必提前设定记录格式，如记录的字段类型、字段数目等。用户可以在系统运行时随意添加或删除字段，因此能够适应记录格式灵活多变的应用。

⑵ 真正的可扩展性：Cassandra 是纯粹意义上的水平扩展。为给集群添加更多容量，可以指向另一台计算机，不必重启任何进程，改变应用查询，或手动迁移任何数据。

⑶ 多数据中心识别：可以通过调整结点布局来避免某一个数据中心发生故障时，备用的数据中心保存了所有的数据。

⑷ 范围查询：如果不喜欢全部的键值查询，则可以设置键的范围来查询。

⑸ 列表数据结构：在混合模式下可以将超级列添加到 5 维。对于每个用户的索引，这是非常方便的。

⑹ 分布式写操作：可以在任何地方、任何时间集中读或写任何数据，并且不会有任何单点失败。

14.5　本 章 小 结

本章主要介绍了数据库技术方面的一些新进展，包括分布式数据库技术、面向对象数据库技术、对象关系数据库技术以及 NoSQL 数据库技术。

通过对本章的学习，读者应了解分布式数据库的基本概念和组成，明白数据分片与分配之间的区别和联系，了解面向对象数据库技术和对象关系数据库技术的概念、区别与联系，并能够初步了解 NoSQL 数据库技术的动机、概念和实现方式。

习　　题

1．分布式数据库系统的特点是什么？
2．简述分布式数据库系统的体系结构。
3．分布式数据库中的数据分片有哪些方式？数据分配有哪些方式？
3．简述面向对象数据模型的基本概念。
4．对象关系数据库技术与面向对象数据库技术有哪些异同点？
5．简述关系数据库技术在 Web 应用上的局限性。
6．NoSQL 数据库目前主要有哪些实现方式？

参 考 文 献

DATE C J, 2007. 数据库系统导论[M]. 8 版. 孟小峰, 王珊, 等译. 北京: 机械工业出版社.

GARCIA-MOLINA H, ULLMAN J D, WIDOM J, 2002. 数据库系统全书[M]. 岳丽华, 杨冬青, 龚育昌, 等译. 北京: 机械工业出版社.

GRAY J, REUTER A, 2004. 事务处理概念与技术[M]. 孟小峰, 于戈, 等译. 北京: 机械工业出版社.

黄德才, 许芸, 王文娟, 等, 2011. 数据库原理及其应用教程——学习指导、例题分析、习题解答与标准试题[M]. 北京: 科学出版社.

刘晖, 彭智勇, 2007. 数据库安全[M]. 武汉: 武汉大学出版社.

邵佩英, 2000. 分布式数据库系统及其应用[M]. 北京: 科学出版社.

SILBERSCHATZ A, KORTH H, SUDARSHAN S, 2012. 数据库系统概念(原书第 6 章)[M]. 杨冬青, 李红燕, 唐世渭, 等译. 北京: 机械工业出版社.

ULLMAN J D, WIDOM J, 2009. 数据库系统基础教程[M]. 3 版. 岳丽华, 金培权, 万寿红, 译. 北京: 机械工业出版社.

王珊, 萨师煊, 2014. 数据库系统概论[M]. 5 版. 北京: 高等教育出版社.

王珊, 朱青, 2005. 数据库系统概论学习指导与习题解答[M]. 北京: 高等教育出版社.

吴宗岱, 2017. 数据库系统概论(第 5 版)同步辅导及习题全解[M]. 北京: 中国水利水电出版社.

张敏, 徐震, 冯登国, 2005. 数据库安全[M]. 北京: 科学出版社.

ABITEBOUL S, HULL R, VIANU V, 1995. Foundations of databases: the logical level[M]. Boston: Addison Wesley.

HELLERSTEIN J M, STONEBRAKER M, HAMILTON J，2007. Architecture of a database system[M]. Hanover: Now Publishers.

GARCIA-MOLINA H, ULLMAN J D, WIDOM J, 2009. Database system implementation(影印版)[M]. 2nd ed. 北京: 机械工业出版社.

RAMAKRISHNAN R, GEHRKE J, 2002. Database management systems[M]. 3rd ed. New York: McGraw-Hill.

TIWARI S, 2011. Professional NoSQL[M]. Indianapolis: John Wiley & Sons.

WEIKUM G, VOSSEN G, 2001. Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery[M]. San Francisco: Morgan Kaufmann.

(TP-9237.31)

数据库系统及应用

本书配有教学课件，可供任课教师参考。申请方式：关注“科学EDU”微信公众号，点击“教学服务”栏目中的“课件申请”。

科学出版社互联网入口　科学EDU

科学出版社　工科分社
联系电话：010-64005312
销售电话：010-64031535
E-mail：gk@mail.sciencep.com

www.sciencep.com

ISBN 978-7-03-075532-2

定 价：69.00元